Visual Studio 2005 Visual C++

Copyright© 2016 Microsoft Corporation

The content in this document is retired and is no longer updated or supported. Some links might not work. Retired content represents the latest updated version of this content.

Visual C++

Microsoft Visual C++ 2005 provides a powerful and flexible development environment for creating Microsoft Windows–based and Microsoft .NET–based applications. It can be used as an integrated development system, or as a set of individual tools. Visual C++ is comprised of these components:

The Visual C++ 2005 compiler tools. The compiler has new features supporting developers that target virtual machine platforms like the Common Language Runtime (CLR). There are now compilers to target x64 and Itanium. The compiler continues to support targeting x86 machines directly, and optimizes performance for both platforms.

- The Visual C++ 2005 Libraries. This includes the industry-standard Active Template Library (ATL), the Microsoft Foundation Class (MFC) libraries, and standard libraries such as the Standard C++ Library, and the C RunTime Library (CRT), which has been extended to provide security enhanced alternatives to functions known to pose security issues. A new library, the C++ Support Library, is designed to simplify programs that target the CLR.
- The Visual C++ 2005 Development Environment. Although the C++ compiler tools and libraries can be used from the command-line, the development environment provides powerful support for project management and configuration (including better support for large projects), source code editing, source code browsing, and debugging tools. This environment also supports IntelliSense, which makes informed, context-sensitive suggestions as code is being authored.

In addition to conventional graphical user-interface applications, Visual C++ enables developers to build Web applications, smart-client Windows-based applications, and solutions for thin-client and smart-client mobile devices. C++ is the world's most popular systems-level language, and Visual C++ gives developers a world-class tool with which to build software.

In This Section

Getting Started with Visual C++

Programming Guide

Deployment (C++)

Porting and Upgrading Programs

Security Best Practices for C++

Creating and Managing Visual C++ Projects

Building a C/C++ Program

Visual C++ Reference

Visual C++ Samples

Related Sections

What's New in Visual C++

Debugging Native Code

Programming for Devices Using Visual C++

Introducing Visual Studio

Visual Studio Walkthroughs

Integrated Development Environment for Visual Studio

Product Support and Accessibility

Getting Started with Visual C++

This section of the documentation helps you get started with using Visual C++ 2005.

In This Section

What's New in Visual C++

Outlines the new features in Visual C++, including the language, libraries, and development environment.

Previous Product Changes

Describes the changes in Visual C++ in previous releases.

Supported Platforms (Visual C++)

Lists the platforms that applications built using Microsoft Visual C++ 2005 can be targeted to.

Visual C++ Settings

Discusses the default settings when starting up Visual Studio the first time and selecting the Visual C++ settings.

Visual C++ Editions

Describes features available in different editions of Visual C++.

Visual C++ Guided Tour

Provides introductory topics on the various types of projects you can create using Visual C++, as well as an introduction to the Visual Studio IDE.

How Do I in Visual C++

Provides categorized lists of Help pages that address key programming tasks of interest to Visual C++ developers.

Related Sections

Integrated Development Environment for Visual Studio

Creating and Managing Visual C++ Projects

.NET Framework Programming in Visual Studio

Automation and Extensibility for Visual Studio

See Also

Other Resources

Visual C++

What's New in Visual C++

This topic has been updated for Visual Studio 2005 SP1.

For Visual C++ 2005, the Visual C++ language has been expanded to include support for developing components and applications that run on a virtual machine with garbage collection. These additions have been approved by the C++/CLI standards committee. Visual C++ 2005 supports targeting the .NET Framework CLR (Common Language Runtime), Microsoft's implementation of a garbage-collected virtual machine, but it can be used to target any platform that conforms to the CLI standard.

The Visual C++ syntax supported in Visual C++ 2005 for .NET Programming is now much more natural than that of the Managed Extensions for C++ introduced in Microsoft Visual C++ .NET 2003. Managed Extensions for C++ is still supported, but has been deprecated in favor of the new syntax. For more information, see New Language Features in Visual C++.

There are also numerous new features for native programming, including targeting 64-bit processor architectures, new library functions with improved security, and more. For the complete list of new features, follow the links below.

Visual C++ 2005 Service Pack 1

- What's New in Visual C++ Compiler, Language, and Tools

Visual C++ 2005

- What's New in the Visual C++ Libraries
- What's New in the Development Environment (C++)
- What's New in Visual C++ Compiler, Language, and Tools

Related Sections

What's New in Visual Studio 2005

Describes the new features of Visual Studio 2005.

Porting and Upgrading Programs

Provides links to information on porting from other platforms, and upgrading from previous releases of Visual C++.

How to: Upgrade Projects from Previous Versions of Visual C++

Provides links to upgrading Visual C++ projects from previous versions.

What's New in Developing Visual C++ Device Applications

Provides information about new features in Visual C++ for Devices.

See Also

Other Resources

Previous Product Changes

What's New in the Visual C++ Libraries

The following sections discuss the updates and breaking changes to the libraries that are shipped with Visual C++.

In this Section

- New MFC and ATL Features
- New CRT Features
- New Standard C++ Library Features
- New C++ Support Library
- New Visual C++ Code DOM Features

General Libraries Changes

Breaking Changes

In mixed mode, native constructors are called before managed constructors. For a single executable, the constructors of all native global objects will be called before the constructor of any managed global object, regardless of whether the native and managed objects exist in the same compilation unit.

Therefore, applications relying on managed constructors being called before native constructors may function in an incorrect manner.

If there are dependencies between global objects, you should ensure that only dependencies from managed objects to native objects exist. The best way of ensuring that an object is managed or native is to put it into a compilation unit that is built that way.

Libraries Changes to Support Manifest-Based Assembly Generation

As of Visual C++ 2005, the Standard C++ Library and the ATL, MFC, and CRT libraries are side-by-side shared assemblies with manifests. Similarly, the compiler and linker settings now default to manifest generation using mt.exe. This will ensure that all Visual C++ executables have a manifest that refers to their dependence on ATL, MFC, and CRT. For detailed information, see Visual C++ Libraries as Shared Side-by-Side Assemblies, Deployment (C++), and Building C/C++ Isolated Applications and Side-by-side Assemblies.

If you use makefiles to build your projects, see How to: Embed a Manifest Inside a C/C++ Application.

Library Support for Mixed Assemblies

Visual C++ 2005 supports the use of the Standard C++ Library and the ATL, MFC, and CRT libraries as mixed assemblies compiled with /clr. These mixed libraries allow you to use all of their existing functions when your code contains a mixture of native code and MSIL-based __clrcall code. For more information, see Mixed (Native and Managed) Assemblies.

Initialization of Mixed Assemblies

The library support changes in Visual C++ 2005 resolve the "loader lock" issues that applied to mixed DLLs in Visual C++ .NET and Visual C++ .NET 2003. However, there is still a restriction that your DllMain code must never access the CLR. For more information, see Mixed (Native and Managed) Assemblies.

Libraries not Supported by Pure Mode Compilation

In Visual C++ 2005, the ATL and MFC libraries are not supported by pure mode compilation (/clr:pure).

See Also

Concepts

What's New in Visual C++

New MFC and ATL Features

This topic contains a list of new MFC and ATL features in Visual C++ 2005.

ATL/MFC Shared Classes

Breaking Changes

See Breaking Changes (ATL/MFC).

ATL

Breaking Changes

See Breaking Changes (ATL).

New Global Functions

- AtlComPtrAssign
- AtlComQIPtrAssign

New Symbols

- _ATL_MIXED

New Methods

- CArray::RelocateElements
- Sids::NetworkService

Deprecated Functions

Some ATL functions are now marked as _ATL_INSECURE_DEPRECATE, deprecated because of a security-related issue. You can use _ATL_SECURE_NO_WARNINGS to suppress the compiler warnings that will be generated when you use these functions.

Deprecated Global Functions

The following functions are obsolete in Visual C++ .NET 2002 and later, but are available for backward compatibility with projects created with previous versions of Visual C++.

- AtlModuleRegisterTypeLib
- AtlModuleUnregisterTypeLib
- AtlModuleRegisterServer
- AtlModuleUnregisterServer
- AtlModuleUnregisterServerEx

ATL Server

/nowchar_t has been added to sproxy.exe. See SPROXY.EXE: XML Web Service Proxy Generator for more information.

Breaking Changes

See Breaking Changes (ATL Server)

Deprecated Functions

_set_security_error_handler has been removed from the CRT, so AtlsSecErrHandlerFunc should no longer be used.

MFC

Breaking Changes

See Breaking Changes (MFC).

MFC Windows Forms Support

Windows Forms support has been added to MFC. Using the MFC Windows Forms support classes, you can host Windows Forms controls within your MFC applications as ActiveX controls within MFC dialog boxes or views. In addition, Windows

Forms forms can be hosted as MFC dialog boxes. For more information, see Using a Windows Form User Control in MFC.

The following code elements have been added for MFC Windows Forms support:

CWinFormsControl Class

CWinFormsView Class

ICommandSource Interface

ICommandTarget Interface

ICommandUI Interface

IView Interface

CommandHandler Delegate

CommandUIHandler Delegate

DDX_ManagedControl

UICheckState Enumeration

Note
MFC Windows Forms integration works only in projects which link dynamically with MFC (projects in which AFXDLL is defined).

GDI+ Shipped with Windows XP

GDI+ is included with Windows XP and is available as a redistributable for Windows NT 4.0 SP6, Windows 2000, Windows 98, and Windows Me. To download the latest redistributable, see http://www.microsoft.com/msdownload/platformsdk/sdkupdate/psdkredist.htm.

For more information on GDI+, see the GDI+ SDK documentation in MSDN: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/gdicpp/GDIPlus/GDIPlus.asp.

New Methods

The following wrapper methods have been added to CDC:

- CDC::SetGraphicsMode
- CDC::GetGraphicsMode
- CDC::SetWorldTransform
- CDC::ModifyWorldTransform
- CDC::GetWorldTransform

The following method has been added to CStdioFile:

- CStdioFile::Seek

Deprecated Methods

- AfxOleTypeMatchGuid
- CArchive::IsByteSwapping
- CInternetConnection::QueryOption
- CInternetFile::QueryOption

Some MFC functions are now marked as _AFX_INSECURE_DEPRECATE, deprecated because of a security-related issue. You can use _AFX_SECURE_NO_WARNINGS to suppress the compiler warnings that will be generated when you use these functions.

See Also

Concepts

What's New in the Visual C++ Libraries

New CRT Features

This topic contains a list of new CRT features in Visual C++ 2005.

CRT

Breaking Changes

See Breaking Changes (CRT).

New features

- Secure versions of several functions have been added. These functions handle errors in a better way and enforce stricter controls on buffers to help avoid common security flaws. The new secure versions are identified by the **_s** suffix.
- Existing less secure versions of many functions have been deprecated. To disable the deprecation warnings, define **_CRT_SECURE_NO_WARNINGS**. For more information, see Security-Enhanced Versions of CRT Functions.
- Many existing functions now validate their parameters and invoke the invalid parameter handler when an invalid parameter is passed. For more information, see the individual function references and the Parameter Validation topic.
- Many existing functions now set **errno** where they did not before. See individual function references for specific **errno** information.
- The **typedef errno_t** with type integer was added. **errno_t** is used whenever a function return type or parameter deals with error codes from **errno**. **errno_t** replaces **errcode**.
- Locale-dependent functions now have versions which take the locale as a parameter rather than using the current locale. These new functions have the **_l** suffix. Several new functions were added to work with locale objects. New functions include _get_current_locale, _create_locale and _free_locale. See individual function references for more information.
- New functions were added to support locking and unlocking file handles. For more information, see _lock_file and _unlock_file.
- The **_spawn** family of functions does not reset **errno** to zero on success, as it did in previous versions. For more information, see _spawn, _wspawn Functions.
- Versions of the **printf** family of functions that allow you to specify the order in which the arguments are used are available. See printf_p Positional Parameters, _cprintf_p, _cwprintf_p, _printf_p, _wprintf_p, _sprintf_p, _swprintf_p, _fprintf_p, _fwprintf_p, _vprintf_p, _vwprintf_p, _vsprintf_p, _vswprintf_p, and _vfprintf_p, _vfwprintf_p.
- Unicode is now a supported text format. The function **_open** supports **_O_TEXTW**, **_O_UTF8** and **_O_UTF16** attributes. The fopen function supports the "ccs=ENCODING" method of specifying a Unicode format. For more information, see _open, _wopen and fopen, _wfopen and fopen_s, _wfopen_s.
- A new version of the CRT libraries built in managed code (MSIL) is now available and is used when compiling with the /clr (Common Language Runtime Compilation) option. See C Run-Time Libraries for more information.
- _fileinfo has been removed.
- The default size for **time_t** is now 64 bits, which expands the range of **time_t** and several of the time functions out to the year 3000. See Time Management, and the individual time functions.
- The CRT now supports setting the locale on a per thread basis. The function _configthreadlocale was added to support this feature.
- The _statusfp2 and __control87_2 functions were added to allow access to and control of the floating point control word on both the x87 and SSE2 floating point processor.
- The_mkgmtime and _mkgmtime64 functions were added to provide support for converting times (**struct tm**) to Greenwich Mean Time (GMT).
- Changes were made to **swprintf** and vswprintf to better conform with the standard. For more information, see sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l.
- A new header file, INTRIN.H, provides prototypes for some intrinsic functions.
- The fopen function now has an N attribute.

- The _open function now has an _O_NOINHERIT attribute.
- The atoi function now returns **INT_MAX** and sets **errno** to **ERANGE** on overflow. In previous versions, the overflow behavior was undefined.
- The printf family of functions supports hexadecimal floating point output implemented according to the ANSI C99 standard using the format type specifiers **%a** and **%A**. For more information, see printf Type Field Characters.
- The **printf** family now supports the "ll" (long long) size prefix. For more information, see Size and Distance Specification.
- The _controlfp function has been optimized for better performance.
- Debug versions of some functions have been added. For more information, see _strdup_dbg, _wcsdup_dbg, _tempnam_dbg, _wtempnam_dbg, _getcwd_dbg, _wgetcwd_dbg, _getdcwd_dbg, _wgetdcwd_dbg,_fullpath_dbg, _wfullpath_dbg.
- Added **_chgsignl** and **_cpysignl** (long double versions).
- Added **_locale_t** type to type table.
- New macro _countof Macro added for computing number of elements in an array.
- In each function topic, a section on .NET Framework equivalents has been added.
- Several string functions now have the option of truncating strings rather than failing when output buffers are too small; see _TRUNCATE.
- _set_se_translator now requires the use of the /EHa compiler option.
- **fpos_t** is now **__int64** under /Za (for C code) and when __STDC__ is set manually (for C++ code). It used to be a struct.
- _CRT_DISABLE_PERFCRIT_LOCKS can improve the I/O performance of single-threaded programs.
- POSIX names have been deprecated in favor of ISO C++ conformant names (for example, use _getch rather than getch).
- New link options .obj files are available for pure mode. See Link Options for more details.
- _recalloc combines features of realloc and calloc.

See Also

Concepts

What's New in the Visual C++ Libraries

New Standard C++ Library Features

This topic contains a list of new Standard C++ Library features in Visual C++ 2005.

Standard C++ Library

- The Standard C++ Library now has support to detect incorrect iterator use. See Debug Iterator Support and Checked Iterators for more information.
- The **for each** keyword works on STL collections. For more information, see How to: Iterate Over STL Collection with for each.
- The uncaught_exception function now conforms to the C++ standard. Previous implementations of this function returned false in all cases.
- For a list of breaking changes in the Standard C++ Library, see Breaking Changes in the Standard C++ Library.

See Also

Concepts

What's New in the Visual C++ Libraries

New C++ Support Library

- A new library to support managed programming in Visual C++: C++ Support Library, featuring auto_gcroot, auto_handle, lock, call_in_appdomain Function, com::ptr.

See Also

Concepts

What's New in the Visual C++ Libraries

New Visual C++ Code DOM Features

This topic contains a list of new Visual C++ Code DOM features in Visual C++ 2005.

Visual C++ Code DOM

- CppCodeGenerator provides functions to generate Visual C++ code that is compliant with CLR features in Visual C++ 2005.
- CppCodeGenerator7 provides functions to generate Visual C++ code that is compliant with Managed Extensions for C++ syntax, which is CLR programming syntax prior to Visual C++ 2005.

See Also

Concepts

What's New in the Visual C++ Libraries

What's New in the Development Environment (C++)

In the Visual Studio Integrated Development Environment (IDE), the following features are new or are enhanced for Visual C++ 2005.

Browsing Source Code

- The Call Browser window, which helps you easily navigate to code that either makes calls to a function, or makes calls from a function.
- Inheritance browsing from Class View. For more information, see How to: Display Inheritance Graphs.
- Live browsing enables features like Call Browser, Find Symbol Results Window, and all tool windows to operate for Visual C++ without generating a BSC file.

IntelliSense

- Identifiers that are defined with the The #define Directive directive are now supported in IntelliSense.
- Symbols from namespaces that are specified with the using Directive (C++) directive are now supported in IntelliSense.
- List Members no longer populates completion lists with symbols from all common libraries, such as Win32, ATL, STL, and MFC. Instead, it populates them with symbols from header files included in your program with the #include directive.
- Templates symbols are now fully supported in IntelliSense. Furthermore, Explicit Template Specializations and Partial Template Specializations are also fully supported in IntelliSense.
- The scalability of IntelliSense has increased from a maximum of 16,000 files per solution to 65,535 files per solution, with a limitation of 65,536 symbols per file.

Application Wizards and Project Templates

- **Create New Project from Existing Code Files Wizard**, which helps you port existing code into a new project. For more information, see How to: Create a C++ Project from Existing Code.
- SQL Server Project Template, which helps you create class library projects for SQL Server 2005.

Note
When you open a .NET Class Library or ASP.NET Web Service project created in a previous version of Visual C++, **/Zl** will be added to the **Command Line** property page for the compiler in the project setting dialog box, but msvcrt.lib and msvcmrt.lib will not be added to the linker's **Additional Dependencies** property. This will result in a linker error (LNK2001) when you b uild the project. To resolve the error, add msvcrt.lib and msvcmrt.lib to the linker's **Additional Dependencies** property or d elete **/Zl** from the compiler's **Command Line** property page.

Project and Build System

- VCBUILD.EXE, which builds Visual C++ projects and solutions from the command line. For more information, see VCBUILD Reference.
- 64-bit platforms support. For more information, see How to: Configure Visual C++ Projects to Target 64-Bit Platforms.
- Property Sheets (C++), which enable you save project settings in files that you can apply to additional projects on multiple computers. Property sheets also allow you to create User-defined Macros.
- Property Manager, which helps you manage property sheets. To display this feature, select the **Property Manager** menu item from the **View** menu.
- Custom build rules, which are defined in Rule Files. This feature facilitates building file extensions that require external build tools.
- Physical view in Solution Explorer, which is available through the **Show All Files** button. Now you can drag files from Windows Explorer onto project nodes in Solution Explorer. The **Show All Files** button displays all file references in your project.

- The **References** node was removed from Solution Explorer. The new References, Common Properties, <Projectname> Property Pages Dialog Box enables you to add references to .NET assemblies, COM components, or project components, to your .NET projects.
- Profile-Guided Optimizations (PGO) build commands available through the project context menu (right-click a project node) in Solution Explorer; and PGO project property settings available through project property pages.
- Multiprocessor Builds, which help you build multiple projects simultaneously.

General Features

- Visual C++ Settings, which are settings that customize the IDE for Visual C++ developers.
- Unicode in the C/C++ Code and Text Editor, Resource Editors, IntelliSense, the Object Browser, wizards for Visual C++ Projects, and all tool windows.
- RAD features (Dataset Designer and Server Explorer Window) for remote databases.

See Also

Reference

What's New in the Visual Studio 2005 Debugger

Concepts

What's New in Visual C++

What's New in Managed Device Projects

What's New in Visual C++ Compiler, Language, and Tools

This topic has been updated for Visual Studio 2005 SP1.

The following sections discuss the updates to the compiler, linker, and other build tools that are included with Visual C++.

Visual C++ 2005 Service Pack 1

The following topics list the new features in Visual C++ 2005 Service Pack 1:

- New Compiler Intrinsics
- New Language Keywords

Visual C++ 2005

The following topics list the new features in Visual C++ 2005:

- New Compiler Features
- New Language Features
- New Preprocessor Features
- New Linker Features
- New Linker Utility Features
- New NMAKE Features
- New MASM Features

See Also

Concepts

What's New in Visual C++

New Compiler Features

This topic lists the new features in the Visual C++ 2005 compiler.

Note
The Visual C++ compiler can only be run on the following operating systems: Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003.

Remarks

The Visual C++ 2005 compiler contains the following new features:

- The compiler has breaking changes in this release. See Breaking Changes in the Visual C++ 2005 Compiler for more information.
- 64-bit native and cross-compilers. For more information, see Compiler Options and x64 Software Conventions.
- /analyze (Enterprise Code Analysis) compiler option has been added.
- /bigobj compiler option has been added.
- **/clr:pure**, **/clr:safe**, and **/clr:oldSyntax** have been added. For more information, see /clr (Common Language Runtime Compilation).
- Deprecated compiler options: many compiler options have been deprecated in this release; see Deprecated Compiler Options for more information.
- Double thunking in **/clr** code is reduced; see Double Thunking (C++) for more information.
- /EH (Exception Handling Model) or **/EHs** can no longer be used to catch an exception that is raised with something other than a throw; use **/EHa**.
- /errorReport (Report Internal Compiler Errors) compiler option has been added.
- /favor (Optimize for 64) compiler option has been added.
- /FA, /Fa (Listing File) compiler option has been added.
- /FC (Full Path of Source Code File in Diagnostics) compiler option has been added.
- /fp (Specify Floating-Point Behavior) compiler option has been added.
- /G (Optimize for Processor) Options compiler option has been added.
- /G (Optimize for Processor) Options compiler option has been added.
- **/G3**, **/G4**, **/G5**, **/G6**, **/G7**, and **/GB** compiler options have been removed. The compiler now uses a "blended model" that attempts to create the best output file for all architectures.
- **/Gf** has been removed. Use /GF (Eliminate Duplicate Strings) instead.
- /GL (Whole Program Optimization) is now compatible with /CLRHEADER. For more information, see /LTCG (Link-time Code Generation).
- **/GR** is now on by default. See /GR (Enable Run-Time Type Information) for more information.
- /GS (Buffer Security Check) now provides security protection for vulnerable pointer parameters. **/GS** is now on by default. **/GS** now also works on functions compiled to MSIL with /clr (Common Language Runtime Compilation).
- /homeparams (Copy Register Parameters to Stack) compiler option has been added.
- /hotpatch (Create Hotpatchable Image) compiler option has been added.
- Inline function heuristics have been updated; see inline, __inline, __forceinline and inline_depth for more information
- Many new intrinsic functions have been added, and many previously undocumented intrinsics are now documented. For more information, see Alphabetical Listing of Intrinsic Functions.

- By default, any call to **new** that fails will throw an exception. For more information, see The new and delete Operators.
- **/ML** and **/MLd** compiler options have been removed. Visual C++ no longer supports single-threaded, statically linked CRT library support. See C Run-Time Libraries for more information.
- The compiler implemented the Named Return Value Optimization, which is enabled when you compile with /O1, /O2 (Minimize Size, Maximize Speed), /Og (Global Optimizations), and /Ox (Full Optimization).
- **/Oa** compiler option has been removed but will be silently ignored; use the noalias or restrict **__declspec** modifiers to specify how the compiler does aliasing.
- **/Op** compiler option had been removed. Use /fp (Specify Floating-Point Behavior) instead.
- OpenMP is now supported by Visual C++. For more information, see OpenMP in Visual C++.
- /openmp (Enable OpenMP 2.0 Support) compiler option has been added.
- **/Ow** compiler option has been removed but will be silently ignored. Use the noalias or restrict **__declspec** modifiers to specify how the compiler does aliasing.
- Profile-Guided Optimizations
- **/QI0f** has been removed.
- **/QIfdiv** has been removed.
- /QIPF_B (Errata for B CPU Stepping) compiler option has been added.
- /QIPF_C (Errata for C CPU Stepping) compiler option has been added.
- /QIPF_fr32 (Do Not Use Upper 96 Floating Point Registers) compiler option has been added.
- /QIPF_noPIC (Generate Position Dependent Code) compiler option has been added.
- /QIPF_restrict_plabels (Assume No Functions Created at Run Time) compiler option has been added.
- Unicode Support in the Compiler and Linker
- /vd (Disable Construction Displacements) now allows you to use dynamic_cast Operator on an object being constructed (**/vd2**)
- **/YX** compiler option has been removed. Use /Yc (Create Precompiled Header File) or /Yu (Use Precompiled Header File) instead. If you remove **/YX** from your build configurations and replace it with nothing, it can result in faster builds.
- **/Zc:forScope** is now on by default. See /Zc:forScope (Force Conformance in for Loop Scope) for more information.
- **/Zc:wchar_t** is now on by default. See /Zc:wchar_t (wchar_t Is Native Type) for more information.
- **/Zd** compiler option has been removed. Line-number only debugging information is no longer supported. Use **/Zi** instead (see /Z7, /Zi, /ZI (Debug Information Format) for more information).
- **/Zg** is now only valid on C source code files, and not on C++ source code files. See /Zg (Generate Function Prototypes) for more information.
- /Zx (Debug Optimized Itanium Code) compiler option has been added.

See Also

Concepts

What's New in Visual C++ Compiler, Language, and Tools

Deprecated Compiler Options

In Visual C++ 2005, the following compiler options have been marked as deprecated. These compiler options will continue to work as expected in the current release, but may be removed in a future release.

Remarks

The following compiler options are now deprecated.

Deprecated option	Use the following option instead	For more information, see
/clr:noAssembly	/LN (Create MSIL Module)	/clr (Common Language Runtime Compilation)
/Fr	**/FR**	/FR, /Fr (Create .sbr File)
/Ge	None	/Ge (Enable Stack Probes)
/GX	/EH (Exception Handling Model)	/GX (Enable Exception Handling)
/GZ	/RTC (Run-Time Error Checks)	/GZ (Enable Stack Frame Run-Time Error Checking)
/H	None	/H (Restrict Length of External Names)
/Og	/O1, /O2 (Minimize Size, Maximize Speed)	/Og (Global Optimizations)

/Qlfist	No compiler option is needed. The compiler has made significant improvements in float to int conversion speed.
	
	Regarding rounding, **/Qlfist** results in generation of the fistp instruction, which uses whatever ambient rounding mode is to do the conversion.
	
	A routine to do this is:
	
	`int ftol_ambient(double d) {`
	`  int i;`
	
	`  __asm {`
	`  fld d`
	`  fistp i`
	`  }`
	`  return i;`
	`}`
		/Qlfist (Suppress _ftol)

/V	None	/V (Version Number)
/Yd	/Z7, /Zi, /ZI (Debug Information Format)	/Yd (Place Debug Information in Object File)
/Ze	None	/Za, /Ze (Disable Language Extensions)
/Zg	None	/Zg (Generate Function Prototypes)

See Also

Concepts

New Language Features

This topic lists the new features in the Visual C++ 2005 language syntax.

Remarks

Visual C++ 2005 features new syntax to develop applications for the Common Language Runtime. For more information, see New Language Features in Visual C++ and Breaking Changes in the Visual C++ 2005 Compiler.

The Visual C++ 2005 language syntax contains the following additional new features:

- The attributeattribute is now deprecated. See User-Defined Attributes for information on how to create attributes.
- appdomain **__declspec** modifier has been added.
- __clrcall calling convention has been added.
- deprecated (C++) **declspec** modifier now allows you to specify a string that will be displayed at compile time, when a user tries to access a deprecated class or function.
- dynamic_cast Operator has breaking changes.
- Native enums now allow you to specify the underlying type. For more information, see C++ Enumeration Declarations.
- jitintrinsic **declspec** modifier has been added.
- noalias **declspec** modifier has been added.
- process **__declspec** modifier has been added.
- **abstract**, **override**, and **sealed** are valid for native compilations. For more information, see How to: Declare Override Specifiers in Native Compilations.
- __restrict keyword has been added.
- restrict **declspec** modifier has been added.
- __thiscall is now a keyword.
- __unaligned keyword is now documented.
- volatile (C++) has updated behavior with respect to optimizations.

See Also

Concepts

What's New in Visual C++ Compiler, Language, and Tools

New Linker Features

This topic lists the new features in the Visual C++ 2005 linker.

Remarks

The Visual C++ 2005 linker contains the following new features:

- Modules (non-assembly MSIL output files) are now allowed as input to the linker. For more information, see .netmodule Files as Linker Input.
- /ALLOWISOLATION (Manifest Lookup) linker option has been added.
- /ASSEMBLYRESOURCE (Embed a Managed Resource) has been updated to now allow you to specify the name of the resource in the assembly, and to specify that the resource is private in the assembly.
- /CLRIMAGETYPE (Specify Type of CLR Image) linker option has been added.
- /CLRSUPPORTLASTERROR (Preserve Last Error Code for PInvoke Calls) linker option has been added.
- /CLRTHREADATTRIBUTE (Set CLR Thread Attribute) linker option has been added.
- /CLRUNMANAGEDCODECHECK (Add SupressUnmanagedCodeSecurityAttribute) linker option has been added.
- /ERRORREPORT (Report Internal Linker Errors) linker option has been added.
- **/EXETYPE** linker option has been removed. The linker no longer supports creating Windows 95 and Windows 98 device drivers. Use an appropriate DDK to create these device drivers. The **EXETYPE** keyword is no longer valid for module definition files.
- /FUNCTIONPADMIN (Create Hotpatchable Image) linker option has been added.
- **/LTCG** linker option is now supported on modules compiled with **/clr**. /LTCG has also been updated to support profile-guided optimizations. See /LTCG (Link-time Code Generation), Profile-Guided Optimizations, and /clr (Common Language Runtime Compilation) for more information.
- /MANIFEST (Create Side-by-Side Assembly Manifest) linker option has been added.
- /MANIFESTDEPENDENCY (Specify Manifest Dependencies) linker option has been added.
- /MANIFESTFILE (Name Manifest File) linker option has been added.
- **/MAPINFO:LINES** linker option has been removed.
- /NXCOMPAT (Compatible with Data Execution Prevention) linker option has been added.
- /PGD (Specify Database for Profile-Guided Optimizations) linker option has been added.
- /PROFILE (Performance Tools Profiler) linker option has been added.
- /SECTION (Specify Section Attributes) linker option now supports attribute negation and no longer supports the L or D (VxD-related) attributes.
- Unicode Support in the Compiler and Linker
- /VERBOSE (Print Progress Messages) linker option now also accepts ICF and REF.
- **/VXD** linker option has been removed. The linker no longer supports creating Windows 95 and Windows 98 device drivers. Use an appropriate DDK to create these device drivers. The VXD keyword is no longer valid for module definition files.
- **/WS** linker option has been removed. **/WS** was used to modify images targeted for Windows NT 4.0. `IMAGECFG.exe -R` *filename* can be used instead of **/WS**. IMAGECFG.exe can be found on the Windows NT 4.0 CD-ROM in SUPPORT\DEBUG\I386\IMAGECFG.EXE.
- /WX (Treat Linker Warnings as Errors) linker option is now documented.

See Also

Concepts

New Linker Utility Features

This topic lists the new features in the Visual C++ 2005 linker utilities.

Remarks

- /ALLOWISOLATION editbin option had been added
- DESCRIPTION module definition file statement is removed. The linker no longer supports building virtual device drivers.
- **/ERRORREPORT** option has been added to bscmake.exe, dumpbin.exe, editbin.exe, and lib.exe.
- **/LTCG** lib option has been added.
- /NXCOMPAT editbin option has been added.
- /RANGE dumpbin option has been added.
- /TLS dumpbin option has been added.
- **/WS** editbin option has been removed. **/WS** was used to modify images targeted for Windows NT 4.0. `IMAGECFG.exe -R` *filename* can be used instead of **/WS**. IMAGECFG.exe can be found on the Windows NT 4.0 CD-ROM in SUPPORT\DEBUG\I386\IMAGECFG.EXE.
- **/WX[:NO]** lib option has been added. See Running LIB for more information.

See Also

Concepts

What's New in Visual C++ Compiler, Language, and Tools

New MASM Features

This topic lists the new MASM features in Visual C++ 2005.

MASM

The following features are new for MASM in Visual C++ 2005:

- MASM expressions are now 64-bit values. In previous versions MASM expressions were 32-bit values.
- The instruction `__asm int 3` now causes a function to be compiled to native. For more information, see __asm.
- ALIAS (MASM) is now documented.
- /ERRORREPORT ml.exe and ml64.exe option is added.
- .FPO is now documented.
- H2INC.exe will not ship in Visual C++ 2005. If you need to continue to use H2INC, use H2INC.exe from a previous version of Visual C++.
- operator IMAGEREL has been added.
- operator HIGH32 has been added.
- operator LOW32 has been added.
- ml64.exe is a version of MASM for the x64 architecture. It assembles x64 .asm files into x64 object files. Inline assembly language is not supported in the x64 compiler. For more information, see MASM for x64 (ml64.exe).

  The following MASM directives have been added for ml64.exe (x64):

  - .ALLOCSTACK
  - .ENDPROLOG
  - .PUSHFRAME
  - .PUSHREG
  - .SAVEREG
  - .SAVEXMM128
  - .SETFRAME

  In addition, the PROC directive was updated with x64-only syntax.
- MMWORD directive has been added
- **/omf** (ML.exe command line option) now implies **/c**. ML.exe does not support linking OMF format objects.
- The SEGMENT directive now supports additional attributes.
- operator SECTIONREL has been added.
- XMMWORD directive has been added

See Also

Concepts

What's New in Visual C++ Compiler, Language, and Tools

New NMAKE Features

This topic lists the new NMAKE features in Visual C++ 2005.

Remarks

NMAKE in Visual C++ 2005 contains the following new features:

- **/ERRORREPORT** has been added. For more information, see NMAKE Options.
- **/G** has been added. For more information, see NMAKE Options.
- The predefined rules have been updated. For more information, see Predefined Rules.
- The $(MAKE) macro, which is documented in Recursion Macros, now gives the full path to nmake.exe.

See Also

Concepts

What's New in Visual C++ Compiler, Language, and Tools

New Preprocessor Features

This topic lists the new features in the Visual C++ 2005 preprocessor.

For more information, see Breaking Changes in the Visual C++ 2005 Compiler.

Remarks

The Visual C++ 2005 preprocessor contains the following new features:

- **__CLR_VER** predefined macro added. For more information, see Predefined Macros.
- The comment (C/C++) pragma now accepts **/MANIFESTDEPENDENCY** as a linker comment. The exestr option to **comment** is now deprecated.
- **embedded_idl** attribute (The #import Directive) now takes an optional parameter.
- fenv_access pragma
- float_control pragma
- fp_contract pragma
- Global variables will not be initialized in the order they are declared if you have global variables in pragma managed, unmanaged and unmanaged sections. This is a potential breaking change if, for example, an unmanaged global variable is initialized with a managed global variables, and a fully constructed managed object is required.
- Sections specified with init_seg are now read only, and not read/write as in previous versions.
- inline_depth default is now 16. A default of 16 was also in effect in Visual C++ .NET 2003.
- **_INTEGRAL_MAX_BITS** predefined macro added, see Predefined Macros.
- **_M_CEE**, **_M_CEE_PURE**, and **_M_CEE_SAFE** predefined macros added, see Predefined Macros.
- **_M_IX86_FP** predefined macro added. For more information, see Predefined Macros.
- **_M_X64** predefined macro added. For more information, see Predefined Macros.
- make_public pragma
- managed, unmanaged pragma syntax updated (now has **push** and **pop**)
- mscorlib.dll is now implicitly referenced by The #using Directive in all **/clr** compilations.
- **_OPENMP** predefined macro added. For more information, see Predefined Macros.
- optimize pragma has been updated, a and w are no longer valid parameters.
- no_registry **#import** attribute has been added.
- region, endregion pragmas added
- **_VC_NODEFAULTLIB** predefined macro added. For more information, see Predefined Macros.
- Variadic Macros are now implemented.
- vtordisp is deprecated and will be removed in a future release of Visual C++.
- The warning pragma now has the **suppress** specifier.

See Also

Concepts

What's New in Visual C++ Compiler, Language, and Tools

New Compiler Intrinsics

This topic is new for Visual Studio 2005 SP1.

The following intrinsics are included beginning with Visual C++ 2005 Service Pack 1:

Intrinsics for x86 and x64

- __halt
- __lidt
- __nop
- __readcr8
- __sidt
- __svm_clgi
- __svm_invlpga
- __svm_skinit
- __svm_stgi
- __svm_vmload
- __svm_vmrun
- __svm_vmsave
- __ud2
- __vmx_off
- __vmx_vmptrst
- __writecr8

Intrinsics for x64 Only

- __vmx_on
- __vmx_vmclear
- __vmx_vmlaunch
- __vmx_vmptrld
- __vmx_vmread
- __vmx_vmresume
- __vmx_vmwrite

See Also

Concepts

What's New in Visual C++ Compiler, Language, and Tools

New Language Keywords

This topic is new for Visual Studio 2005 SP1.

The following Microsoft-specific keywords are included beginning with Visual C++ 2005 Service Pack 1:

- __sptr, __uptr

See Also

Concepts

What's New in Visual C++ Compiler, Language, and Tools

Previous Product Changes

This section contains topics that list the changes in Visual C++ in previous versions.

In This Section

Product Changes in Visual C++ Releases

Provides links to topics describing the changes in each release of Visual C++.

Library Changes in Visual C++ Releases

Provides links to topics describing changes to the libraries that ship with Visual C++, including the Active Template Library (ATL), Microsoft Foundation Class (MFC) Library, and the Standard C++ Library.

Related Sections

How to: Upgrade Projects from Previous Versions of Visual C++

Discusses how to use projects from previous versions of Visual C++.

Product Changes in Visual C++ Releases

Visual C++ undergoes changes from release to release. Read the following topics for more information about the changes to Visual C++ and in which release the changes were made.

Changes in the libraries are catalogued separately. See Library Changes in Visual C++ Releases.

In This Section

Visual C++ .NET 2003

Describes the major changes in this release.

Visual C++ 6.0 Default Shortcut Keys

Provides a list of the default shortcut key mappings for Visual C++ 6.0.

Visual C++ 2.0 Default Shortcut Keys

Provides a list of the default shortcut key mappings for Visual C++ 2.0.

Related Sections

What's New in Visual C++

Describes the major changes in this release.

Product Changes: Visual C++ .NET 2003

Note
Some features mentioned in this topic may not still exist in the current version of Visual C++. For more information, see What's New in Visual C++.

Microsoft Visual C++ .NET 2003 provides many improvements and several new features:

- Visual C++ Compiler, Language, and Linker
- Visual C++ Libraries
- Development Environment
- Windows Forms Designer for Managed Extensions for C++
- Project Build Automation Model

Visual C++ Compiler, Language, and Linker

The following compiler, linker, other build tools, and Visual C++ language features are new for Visual C++ .NET 2003.

Compiler

- Information on how to run a Managed Extensions for C++ application built with the current version's compiler on a previous version of the runtime.
- A walkthrough has been added showing how to port an existing, native application to use Managed Extensions for C++: Walkthrough: Porting an Existing Native C++ Application to Interoperate with .NET Framework Components
- You can now create a delegate on a method of a value type.
- The compiler's conformance with the C++ standard has been significantly enhanced for Visual C++ .NET 2003.
- **/arch** compiler option is added.
- **/Gf** is deprecated and will be removed in the next version of Visual C++.
- **/G7** compiler option is added.
- The **/GS** compiler option has been enhanced to help protect local variables from direct buffer overruns.
- The /**noBool** compiler option has been removed. The compiler now allows **bool** to appear only as a keyword (and not an identifier) in a Visual C++ source code file.
- The long **long** type is now available as a typedef of **__int64**. Note that there is not yet support for long **long** in the CRT.
- The **/Zm** compiler option now specifies the precompiled header memory allocation limit.
- **_InterlockedCompareExchange** intrinsic now documented.
- **_InterlockedDecrement** intrinsic now documented.
- **_InterlockedExchange** intrinsic now documented.
- **_InterlockedExchangeAdd** intrinsic now documented.
- **_InterlockedIncrement** intrinsic now documented.
- **_ReadWriteBarrier** intrinsic added.

Visual C++ Attributes

- **implements** attribute is now documented.

Linker

The following linker options have been added:

- **/ASSEMBLYDEBUG**
- **/ASSEMBLYLINKRESOURCE**
- **/DELAYSIGN**
- **/KEYFILE**
- **/KEYCONTAINER**
- **/SAFESEH**

Preprocessor

- The _STATIC_CPPLIB symbol is now documented for use with **/MD**.
- The _CPPLIB_VER symbol is now documented.
- The **#import** directive now has the following attributes documented:
  - auto_search
  - auto_rename
  - no_search_namespace
  - rename_search_namespace
  - tlbid

MASM

The **.SAFESEH** directive and /safeseh ml.exe option were added.

Visual C++ Libraries

Old iostream Library

The old iostream library has been removed from Visual C++ in this release. Use the Standard C++ Library for iostream programming.

C Run-Time Library

- The documentation for the C Run-Time Library now includes information for .NET Framework equivalents.
- **_get_heap_handle** has been added.
- The _CRTDBG_CHECK_DEFAULT_DF macro, used in **_CrtSetDbgFlag** is now defined to zero, meaning that no heap checking is done by default.
- **swprintf** now is compatible with the ISO C standard (C++ only).
- **vswprintf** now has a C++-only form that lets you specify the maximum number of characters to store.
- For several of the multibyte functions, such as **_mbsset**, the CRT was not always detecting invalid multibyte strings, where a lead byte would be followed by a null trail byte. The CRT now has more checks for invalid multibyte characters, where a lead byte is followed by a null trail byte.
- **_set_purecall_handler** has been added.
- Previously, when a process loaded a DLL that statically referenced the CRT library, floating-point precision would be initialized to 53 bits. In Visual C++ .NET 2003, floating-point precision is not initialized in this scenario. This may cause a breaking change in some existing applications that relied on the CRT to initialize floating-point precision.
- The CRT now has float and double forms for all math functions. These new functions are only callable from Visual C++.

Standard C++ Library

In previous releases, input and output of characters to a stream may have resulted in character or unsigned short values being stored if **wchar_t** was not defined as a native type. Now, an unsigned short is always treated as a character.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace.

The Standard C++ Library has been updated to take advantage of the enhanced Visual C++ compiler support for the C++

standard. For example, the Standard C++ Library implementation shipping in Visual C++ now makes use of partial template specialization.

ATL

- In Visual Studio .NET 2003, the MIDL compiler has a default setting of **/robust**, which causes projects running under Windows NT 4 to stop responding.

To change the MIDL compiler flag to /no_robust

1. Right-click your project, and on the shortcut menu, click **Properties**.

   The **Project Properties** dialog box appears.

2. In the left pane, click **MIDL**, and then select **Command Line**.

3. Enter **/no_robust** in the **Additional Options** text box.

- The ATL string conversion macro **USES_CONVERSION** has been superseded by **USES_CONVERSION_EX**. It will try to allocate space on the stack. If there is no available room on the stack, it will try the heap. If there is no room on the heap, it will return **NULL**. **USES_CONVERSION_EX** has an extra parameter (*threshold*) that you can use: if the request is larger than the size of *threshold*, the macro will go to the heap directly.

- **_alloca** has been superseded by **_atl_safe_alloca**.

- The class **CSocketAddr Class** has been added to provide protocol independent methods for dealing with both IPv6 and IPv4 addresses.

- When building a project that outputs an executable, ATL automatically adds quotation marks around the path name created at run time with the **%MODULE%** registrar script parameter. If you do not want the path name to include the quotation marks, use the new **%MODULE_RAW%** parameter instead.

  When building a project that outputs a DLL, ATL will not add quotation marks to the path name if **%MODULE%** or **%MODULE_RAW%** is used.

MFC

- Connection maps are no longer compacted when a connection point is deleted; instead, the deleted connection point is replaced with a NULL. Therefore, you should check for NULL when using **CConnectionPoint::GetConnections** or **CConnectionPoint::GetNextConnection**.

- The following functions may now throw exceptions: **CSimpleString::FreeExtra**, **CSimpleString::GetAt**, **CSimpleString::operator []**, **CSimpleString::ReleaseBuffer**, **CSimpleString::ReleaseBufferSetLength**, **CStrBuf::SetLength**, and some **CTime::CTime** constructors.

- The behaviors of **AfxIsValidAddress** and **AfxIsValidString** have changed. In non-debug builds, both functions just test for a non-NULL argument.

- **CHeaderCtrl::GetOrderArray** no longer has a default second parameter; -1 is no longer a valid value for the second parameter.

- **CAsyncSocket** has new IPv6-aware members: **GetPeerNameEx**, **GetSockNameEx**, **ReceiveFromEx**, and **SendToEx**.

ATL and MFC

- The **CImage** class now keeps track of the number of objects created. Whenever the count goes to 0, the function **GdiplusShutdown** is automatically called to release resources used by GDI+. This prevents resources from not being freed when **CImage** objects are created by a DLL.

- The MFC **CString** class has been rewritten as a template class, **CStringT**. This allows you to use **CString** in ATL projects without linking in the larger MFC static library or DLL.

  Note

This release fixes the problem described in the Knowledge Base article, "PRB: Linking Errors When You Import CString-Derived Classes" (Q309801). You can find Knowledge Base articles on the MSDN Library CD-ROM or at http://support.microsoft.com/default.aspx. If you encountered linker errors when exporting a **CString**-derived class from an MFC extension DLL in Visual C++ .NET 2002 and have applied the workaround described in this article, you should remove the workaround code, because the problem has been fixed in Visual C++ .NET 2003.

- **CStrBufT** and **CSimpleStringT** now have an additional template parameter that tells you whether **CString** is to be used from the MFC DLL. Use the typedef found in the **CStringT** or **CSimpleStringT** class in atlsimplstr.h (rather than the global typedef).

ATL Server

- In an XML Web service created with ATL Server, the default action is to validate SOAP parameters after they are read. To switch off the validation, define the macro **_ATL_SOAP_NO_PARAMETER_VALIDATIONS**.
- SPROXY.EXE can now process a .discomap file or a .wsdl file. Specify the new `/wsdl` option when you use a .wsdl file as the input:

  ```
  sproxy /wsdl <input_location>
  ```

  where `<input_location>` is the path of the .wsdl file to use.

  SPROXY.EXE can now process a results.discomap file:

  ```
  sproxy results.discomap
  ```

  Note that the .discomap file contains a link to a local copy of the .wsdl file and also uses a local copy of the schema file.

Development Environment

Note

Some features mentioned in this topic may not still exist in the current version of Visual C++. For more information, see What's New in Visual C++.

When you open a Visual C++ project from Visual Studio .NET in Visual C++ .NET 2003, the old project file will be renamed and a new project file for the Visual C++ .NET 2003 environment will be created.

The format of a .vcproj file is now documented.

The following new features are in the development environment:

- $(WebDeployPath), $(WebDeployRoot), $(ParentName), $(RootNameSpace), $(SafeParentName), and $(SafeInputName) macros have been added.
- It is now possible to build a project without building any of the projects dependent projects.

Managed Extensions for C++ Project Templates

This release includes several new project templates that you can use to create applications in Managed Extensions for C++:

- **Windows Controls Library (.NET)**
- **Windows Forms Application (.NET)**
- **Windows Service (.NET)**

In addition, the templates for managed projects in Visual C++ .NET 2002, were renamed for this release:

- **Managed C++ Application** is now called **Console Application (.NET)**
- **Managed C++ Class Library** is now called **Class Library (.NET)**
- **Managed C++ Empty Project** is now called **Empty Project (.NET)**

- **Managed C++ Web Service** is now called **ASP.NET Web Service**

Windows Forms Designer for Managed Extensions for C++

With the addition of the Windows Forms Designer for this release, Visual C++ introduces a rapid application development solution for creating your Windows Forms applications in Managed Extensions for C++.

This feature includes full support of the Toolbox and Server Explorer, which allows you to drag and drop, or cut and paste, controls and components directly onto your Windows Forms application. In addition, you can easily manipulate the properties of your controls and components through the Property Grid.

Project Build Automation Model

Visual C++ .NET offers new property pages and a number of new objects, as well as new properties and methods for existing objects that enhance the project build model.

New Property Pages

Managed Resources Property Page

XML Data Generator Tool Property Page

Managed Wrapper Property Page

Auxiliary Managed Wrapper Property Page

New Objects

VCALinkTool Object

VCActiveXReference Object

VCAssemblyReference Object

VCAuxiliaryManagedWrapperGeneratorTool Object

VCManagedResourceCompilerTool Object

VCManagedWrapperGeneratorTool Object

VCPrimaryInteropTool Object

VCProjectReference Object

VCReference Object

VCReferences Collection

VCReferenceConfiguration Object

VCXMLDataGeneratorTool Object

New or Changed Members for Existing Objects

Several new or updated properties and methods were added to existing objects. Note that changes can encompass several things, such as an updated signature or the addition of new members.

Object	New or changed member
VCCLCompilerTool Object	**EnableEnhancedInstructionSet** Property
	ToolKind Property
VCLinkerTool Object	**FixedBaseAddress** Property
	ToolKind Property
VCLibrarianTool Object	**ToolKind** Property
VCCustomBuildTool Object	**ToolKind** Property

VCMidlTool Object	**ToolKind** Property
VCResourceCompilerTool Object	**ToolKind** Property
VCPreBuildEventTool Object	**ToolKind** Property
VCPreLinkEventTool Object	**ToolKind** Property
VCPostBuildEventTool Object	**ToolKind** Property
VCBscMakeTool Object	**ToolKind** Property
VCNMakeTool Object	**ToolKind** Property
VCWebServiceProxyGeneratorTool Object	**Namespace** Property (**VCProjectEngine**)
	References Property
	ToolKind Property
VCWebDeploymentTool Object	**ToolKind** Property
VCProject Object	**AddAssemblyReference** Method
	AddActiveXReference Method
	AddProjectReference Method
	CanAddAssemblyReference Method
	CanAddActiveXReference Method
	CanAddProjectReference Method
	Object Property
	References Property
	ReferencesConsumableByDesigners Property
	RemoveReference Method
	RootNamespace Property
	VCReferences Property
VCFile Object	**AddFile** Method
	CanAddFile Method
	FileType Property
	Items Property

	Object Property
	RemoveFile Method
VCFilter Object	**Object** Property
VCConfiguration Object	**FullReferencesPath** Property
	ReferencesPath Property
	ReferenceTools Property
	SatelliteDLLs Property
	StopBuild Method
	StyleSheets Property
VCStyleSheet Object	**FullReferencesPath** Property
	ReferencesPath Property
	ReferenceTools Property
	StyleSheetDirectory Property
	StyleSheetName Property
VCFileConfiguration Object	**ProjectConfiguration** Property
	Compile Method
	OutputUpToDate Property
VCProjectEngine Object	**ShowEnvironmentInBuildLog** Property

See Also

Other Resources

Product Changes in Visual C++ Releases

Product Changes: Visual C++ .NET 2002

Note
Some features mentioned in this topic may not still exist in the current version of Visual C++. For more information, see What's New in Visual C++.

Microsoft Visual C++ .NET 2002 provides many new features and improvements:

- Managed Code and Targeting the .NET Framework
- Attributed Programming
- Visual C++ Compiler, Linker, and Visual C++ Language
- Development Environment
- Resource Editors
- Libraries
- Deployment and Redistribution of an Application
- Visual C++ .NET Samples

Managed Code and Targeting the .NET Framework

Visual C++ introduces two ways for you to write managed code. Both are designed to make COM programming easier and to allow your programs to access the services provided by the .NET common language runtime, such as garbage collection and language interoperability.

- **Managed Extensions for C++** By using these extensions, you can continue writing C++ code in existing programs.
- **Visual C#** C# is a new programming language that is simple, type safe, and object oriented. An evolution of C and C++, C# is designed for developers to create new .NET Framework applications.

Attributed Programming

Attributes are designed to provide an efficient and quick method to simplify COM programming in Visual C++. Attributes, like C++ keywords, are used in your source files and interpreted by the compiler. Attributes can modify the behavior of your existing code and even insert additional framework code to accomplish basic tasks, such as implementing an ActiveX control, creating a class factory, or formatting a database command. You can apply attributes to nearly any Visual C++ object, such as classes, data members, and member functions, or you can insert attributes into your source code as stand-alone statements.

Visual C++ Help includes attributes walkthroughs that cover various aspects of attributed programming.

Visual C++ Compiler, Linker, and Visual C++ Language

The following compiler, linker, other build tools, and Visual C++ language features are new for Visual C++ .NET. There is also an update to the Standard C++ Library.

Compiler

- Standard compliance issues in Visual C++.
- **/AI** compiler option.
- **/clr** compiler option.
- **/FU** compiler option.
- **/Fx** compiler option.
- **/GH** compiler option.
- **/Gi** compiler option was removed from the compiler; the Visual C++ compiler no longer does incremental compilation. **/Gi** will be silently ignored by the compiler.
- **/GD** compiler option was removed from the compiler.

- **/GL** compiler option.
- **/GS** compiler option.
- New keywords associated with Managed Extensions for C++.
- **/Qlfist** is now documented.
- **/O1** and **/O2** now use **/GF** instead of **/Gf**.
- **/Ox**, **/O1**, and **/O2** now use **/Ob2** instead of **/Ob1**.
- **/RTC** compiler option.
- **/showIncludes** compiler option.
- **/w** compiler option now lets you specify the behavior of specific warnings.
- **/Wall** compiler option now lets you enable all warnings, including warnings that are disabled by default.
- **/WL** compiler option.
- **/Wp64** compiler option.
- **/Y-** compiler option.
- **/Zc:forScope** compiler option.
- **/Zc:wchar_t** compiler option.
- A list of warnings that are off by default.
- Boolean expressions are now of type **bool**, not **BOOL**.

Linker

- **/ALLOWBIND** linker option.
- **/ASSEMBLYMODULE** linker option.
- **/ASSEMBLYRESOURCE** linker option.
- **/COMMENT** linker option is now deprecated. Use the **comment** pragma.
- The CVPACK utility is removed.
- **/DEBUGTYPE** linker option has been removed. Symbol support in COFF format is removed; symbols must now be placed in .pdb files.
- **/GPSIZE** linker option has been removed. The linker now determines the optimal location for communal variables (uninitialized global data items) based on the presence of GP relative relocations to the data.
- **/IDLOUT** linker option.
- **/IGNOREIDL** linker option.
- **/LINK50COMPAT** has been removed. Import libraries created with Visual C++ .NET are compatible with Visual C++ 6.0.
- **/LTCG** linker option.
- **/MAPINFO:FIXUPS** has been removed.
- **/MIDL** linker option.
- **/NOASSEMBLY** linker option.
- **/PDB:NONE** linker option has been removed. Debug information in the .exe is no longer supported.
- **/PDBSTRIPPED** linker option.
- **/PDBTYPE** linker option has been removed. Debug information will be placed in a single .pdb file.
- The **/PROFILE** linker option has been removed. If you are using a third party profiler that needs **/PROFILE**, you should instead use **/FIXED:NO**. In this version, the linker will still accept **/PROFILE** as a synonym for **/FIXED:NO**.

- The source profiler, a linker utility, is removed.
- **/TLBID** linker option.
- **/TLBOUT** linker option.
- **/TSAWARE** linker option.
- **/WARN** linker option has been removed.

Dumpbin

- **/CLRHEADER** option.
- **/IMPORTS** option for the DUMPBIN utility now takes an optional parameter that lets you display imports from a specified DLL.
- **/PDBPATH** option.
- **/RAWDATA** parameters are renamed, but syntax from previous releases is still valid.
- **/UNWINDINFO** option.

Visual C++ Language

- **_AddressOfReturnAddress** keyword is now documented.
- **__alignof** keyword.
- Covariant return types are now supported.
- **__declspec(align)** keyword.
- **__declspec(deprecated)** keyword.
- **__declspec(noinline)** keyword.
- **__declspec(selectany)** keyword now supports dynamically allocated objects (objects initialized by a constructor).
- **__event** keyword.
- The **for** statement under **/Ze** has been modified to adhere more closely to standard behavior.
- **__if_exists** and **__if_not_exists** keywords.
- **__interface** keyword.
- Function-try blocks are now supported.
- **__m64** data type.
- **__m128** data type.
- **__m128d** data type.
- **__m128i** data type.
- MMX, SSE and SSE2 intrinsics.
- **__super** keyword.
- **template** keyword now allows template parameters.
- **throw(...)** syntax.
- **__assume** compiler intrinsic.
- **__debugbreak** compiler intrinsic.
- **__noop** compiler intrinsic.
- Explicit overrides.
- Limit on levels of nested parentheses in a single expression is now 256.

- **_ReturnAddress** keyword is now documented.
- **__sealed** keyword.
- **__w64** keyword.
- **wchar_t** is now a native data type.
- New keywords associated with Managed Extensions for C++.

Preprocessor

- **conform** pragma.
- **__COUNTER__** predefined macro.
- **deprecated** pragma.
- **__FUNCDNAME__** predefined macro.
- **__FUNCSIG__** predefined macro.
- **__FUNCTION__** predefined macro.
- **#import** directive:
- You can now specify type libraries by **progid** or **libid**.
- The following are new attributes in the **#import** directive:
    - **embedded_idl**
    - **no_dual_interfaces**
    - **no_smartpointers**
- **_MANAGED** preprocessor directive.
- **managed** and **unmanaged** pragmas.
- **pack** pragma now has a **show** option to display current pack value.
- **pop_macro** is now documented.
- **push_macro** is now documented.
- **runtime_checks** pragma.
- **section** pragma.
- **#using** directive.
- **_WCHAR_T_DEFINED** preprocessor directive.

Development Environment

The following development environment features and changes are new for Visual C++ .NET.

- Class View has been extended.
- Excluding Files When Dependency Checking (new procedure).
- The Properties window provides a customizable grid for viewing and modifying properties of objects in your project.
- Project property pages let you specify build settings for projects.
- Wizards are now designed in HTML and JScript, with supporting text files, to make custom wizards more flexible and extensible.
- WizardBar and ClassWizard have been removed.
- Customizing the build process is a new pre-build event.
- The New Database Wizard has been removed from Visual C++ .NET.

- Exporting an NMAKE makefile is no longer available. Use devenv command line switches to build a Visual C++ project from the command line.
- Visual C++ 6.0 projects that use **/Gf** and that are imported into the new version will have their **/Gf** options converted to **/GF**.

Resource Editors

Image Editor Toolbar and the Option Selector

The Image Editor toolbar was called the Graphics toolbar in Visual C++ 6.0. While the buttons on the toolbar are essentially the same (tools for drawing, painting, entering text, erasing, and manipulating views), the behavior and presentation of the toolbar has changed slightly.

In Visual C++ 6.0, the Graphics toolbar was initially a docked window on the left side of the workspace and the Options selector appeared as a box within the toolbar. In Visual C++ .NET, the Image Editor toolbar appears at the top of the workspace. The Option selector now appears on the far right in the Image Editor toolbar as a button with a drop-down arrow. The icon on the Option selector button changes depending on the tool you have selected.

The Text Tool Dialog Box on the Image Editor Toolbar

The behavior of the Text Tool dialog box has changed from Visual C++ 6.0 to Visual Studio .NET.

Action	Result in Visual C++ 6.0	Result in Visual Studio .NET
Press ENTER	Entered new line into text.	Exits the Text tool and commits the changes.
Click the **Close** button	Exited the Text tool and committed the changes.	Exits the Text tool and commits the changes.
Press ESC	Exited the Text tool and committed the changes.	Exits the Text tool and discards the changes.
Press SHIFT + ENTER	(Not applicable)	Enters a new line into text.
Press CTRL + ENTER	(Not applicable)	Enters a new line into text.

These changes make the Text tool behave more like standard dialog boxes and the other Image editor tools.

Libraries

Note
Some features mentioned in this topic may not still exist in the current version of Visual C++. For more information, see What's New in Visual C++.

The following libraries included in Visual C++ .NET are either new or have changed.

Active Template Library (ATL)

- The default settings for ATL projects have changed.
- More reference topics for ATL methods contain code examples.
- Support for collections and enumerators: **ICollectionOnSTLImpl**, **IEnumOnSTLImpl**, **CComEnumImpl**, **CComEnumOnSTL**, **CComEnum**
- A new class, **CImage**, that provides enhanced bitmap support, including the ability to load and save images in JPEG, GIF, BMP, and Portable Network Graphics (PNG) formats.
- New classes for managing arrays, lists and trees: **CAtlArray**, **CAtlList**, **CAtlMap**, **CRBMap**, and **CRBMultiMap**.
- New string conversion macros and classes.
- Utility classes: **CAdapt**
- Hosting ActiveX controls: **IAxWinAmbientDispatch**, **IAxWinHostWindow**, **CAxWindow2**
- Enhancements to **CComBSTR** and **CComVariant**
- New macros: **OBJECT_ENTRY_AUTO** and **OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO**

- Simplified object creation: **CComCoClass::CreateInstance**
- Improved debugging support: **_ATL_DEBUG_INTERFACES**
- Registry support for **REG_MULTI_SZ** values: **CRegKey**
- New security classes: **CAcl**, **CDacl**, **CSacl**, **CSecurityAttributes**, **CSecurityDesc**, **CSid**, **CTokenGroups** and **CTokenPrivileges**.
- New array management classes: **CSimpleArray**, **CSimpleMap**, **CComSafeArray**, **CComSafeArrayBound**.
- A Currency object class: **CComCurrency**.
- The following ATL macros should be considered obsolete: **atlTraceFlags**, **ATLTrace**, **BEGIN_OBJECT_MAP**, **OBJECT_ENTRY**, **END_OBJECT_MAP**.
- Several ATL methods have been replaced.
- The functionality of **CComModule** has been distributed into several new classes.

ATL Server

ATL Server is a set of native C++ classes that allows developers to create Web applications, XML Web services, and other server applications. Many classes can also be used in client applications or components.

C Run-Time Library

- Run-time error checks functions
- The **new** operator can throw an exception or return zero if the memory allocation fails.
- Recover from stack overflows with **_resetstkoflw**
- New functions to allow dates beyond the year 2038:

_ctime64	**_wfindnext64**	**_gmtime64**	**_wstat64**
_wctime64	**_fstat64**	**_localtime64**	**_time64**
_findfirst64	**_ftime64**	**_mktime64**	**_utime64**
_wfindfirst64	**_futime64**	**_stat64**	**_wutime64**
_findnext64			

- Configure debug-heap checks with **_CrtSetDbgFlag**.
- Determine block and subblock types **_CrtReportBlockType**
- Math constants
- New wide-character functions:

_cwprintf	**_putwch**	**_putws**	**_wtof**

- New **__int64** versions of string functions:

_strtoi64	**_strtoui64**	**_wctoi64**	**_wcstoui64**

- New Unicode versions of console functions:

_getwch	**_ungetwch**	**cgetws**	**_getwche**
_cwscanf			

- New string functions that calculate the number of characters needed to print formatted data:

_scprintf	**_scwprintf**	**_vscprintf**	**_vscwprintf**
_snscanf	**_snwscanf**		

- Data alignment functions.
- **_set_security_error_handler**, for registering a security error handler.
- Because the timing of the release of C99, this version of Visual C++ is not conformant with that standard.
- New byte-swapping functions: **_byteswap_uint64**, **_byteswap_ulong**, **_byteswap_ushort**.
- **_set_SSE2_enable** to use SSE2 instructions.
- **_CrtSetReportHook2**.

Microsoft Foundation Class (MFC) Library

- Reference topics for MFC contain hundreds of new code examples.
- Static Casting and MFC Message Maps Beginning with Visual C++ .NET, MFC provides stricter type checking for return and parameter types of message handler functions. This new behavior notifies the developer of potential problems by flagging potentially unsafe message handlers with an error message. MFC now uses static casts for ON_MESSAGE, ON_REGISTERED_MESSAGE, ON_THREAD_MESSAGE, and ON_REGISTERED_THREAD_MESSAGE.

  For example, in the past a developer could use a member function that returned **void** instead of LRESULT for ON_MESSAGE or ON_REGISTERED_MESSAGE and compile without any errors. With Visual C++ .NET, the potential miscast is caught and flagged as an error. The developer can fix the potential problem by replacing the return type (with LRESULT) and recompiling.
- DHTML editing component: **CHtmlEditCtrl**, **CHtmlEditView**, **CHtmlEditDoc**
- DHTML dialog boxes: **CDHtmlDialog**, **CMultiPageDHtmlDialog**
- ISAPI support for parsing argument lists: **CHttpArg**, **CHttpArgList**
- Support for windowless controls: **COleControlSite**, **COleControlContainer**, and **COccManager**
- Enhanced support for using HTML Help in an MFC application.
- Windows 2000 print property sheet: **CPrintDialogEx**.
- DAO support: The Visual C++ MFC Application Wizard and MFC DLL Application Wizard no longer support DAO database projects. You can still add DAO-derived classes using the Add Class Wizard. Microsoft recommends using OLE DB or ODBC for new native C++ projects. You should use DAO only in maintaining existing applications.
- **TRACE0**, **TRACE1**, **TRACE2**, and **TRACE3** are now considered obsolete; use **ATLTRACE2**. **TRACE** now has the same functionality as **ATLTRACE2**.
- When selecting a dialog box font, use MS Shell Dlg, not MS Sans Serif or Helv fonts. Previous versions of MFC would automatically replace MS Sans Serif or Helv with DEFAULT_GUI_FONT or the system font, but MFC no longer does that.
- Enhanced support for localized resources in satellite DLLs.

OLE DB Templates

The following OLE DB Templates are provided in Visual C++:

- New consumer classes: **CDataConnection**, **CDynamicAccessor**, **CDynamicParameterAccessor**, **CDynamicStringAccessor**, **CDynamicStringAccessorA**, **CDynamicStringAccessorW**, **CStreamRowset**, **CXMLAccessor**.
- New consumer macros: **COLUMN_NAME*** macros for binding to a specific column in the database by name and **BLOB*** macros for binding binary large objects (BLOB).
- New provider classes: **IDBSchemaRowsetImpl**, **IErrorRecordsImpl**, **IRowsetChangeImpl**, **IRowsetCreatorImpl**, **IRowsetNotifyCP**, **IRowsetNotifyImpl**, **IRowsetLocateImpl**, **IRowsetUpdateImpl**.

New 64-bit Data Types

Methods in several OLE DB Templates classes have been changed to use the new 64-bit data types instead of the old 32-bit ones. The documentation has been updated to reflect these changes. In case of discrepancy between the documentation and the header files (atldb.h, atldbcli.h, and atldbsch.h), the header files contain the most current parameter types.

Shared Classes

A number of new and revised classes can now be used in any Visual C++ program.

Standard C++ Library

The Standard C++ Library has been updated:

- hash_map Class
- hash_set Class
- Optional exception handling support.
- Enhanced code formatting (readability).
- Enhanced DLL support (passing objects between process boundaries).
- Enhanced multithreading support.
- Enhanced compliance with the standard.

Old iostream Library

For Visual C++, the use of the old iostream library is deprecated; you will get a warning each time you use an iostream function. It is possible that the old iostream library will be removed in a subsequent release.

Deployment and Redistribution of an Application

Note
Some features mentioned in this topic may not still exist in the current version of Visual C++. For more information, see What's New in Visual C++.

To redistribute a Visual C++ application, you can use deployment in Visual Studio, which allows you to create setup executables, packaging files, and publish to Web sites.

If you do not want to use Visual Studio deployment and you have a native (not managed) Visual C++ application, you can discover your application's dependencies and develop your own setup process.

Visual C++ .NET Samples

Note
Some features mentioned in this topic may not still exist in the current version of Visual C++. For more information, see What's New in Visual C++.

Visual C++ .NET includes all new samples for the following areas:

- C Run-Time Library
- ATL Server
- Attributes
- Managed Extensions for C++
- MASM
- Debugging
- Custom Wizard

Note

Because the wizard architecture has changed, the samples previously provided for Custom Wizards were retired, and new Custom Wizard samples have been provided.

- Code Model
- Project Model
- Event Handling
- International
- Tailspin Toys Application

Additionally, Visual C++ .NET includes new samples for the following areas plus the samples that shipped in previous releases:

- MFC
- ATL
- COM support
- Standard Template Library (STL)

See Also

Other Resources

Product Changes in Visual C++ Releases

Visual C++ 6.0 Default Shortcut Keys

This section lists the default key combinations available for the Visual C++ 6.0 keyboard scheme. For information on changing default combinations, see How to: Work with Shortcut Key Combinations.

Note
To see a list of shortcut keys that have changed or are no longer supported in Visual C++ .NET, see Obsolete Visual C++ 6.0 Key Commands and Changes to Visual C++ 6.0 Key Commands.

Control Manipulation Shortcut Keys	Miscellaneous Editor Shortcut Keys
Database Tools Shortcut Keys	Object Browser Shortcut Keys
Debug Shortcut Keys	Project Shortcut Keys
Dialog Editor Shortcut Keys	Search and Replace Shortcut Keys
Global Shortcut Keys	Text Manipulation Shortcut Keys
HTML Designer Shortcut Keys	Text Navigation Shortcut Keys
Image Editor Shortcut Keys	Text Selection Shortcut Keys
Integrated Help Shortcut Keys	Tool Window Shortcut Keys
Macro Shortcut Keys	Window Management Shortcut Keys

See Also

Other Resources

Shortcut Keys

Obsolete Visual C++ 6.0 Key Commands

Changes to Visual Studio have required changes to shortcut keys. For a list of keys that have been reassigned, see Changes to Visual C++ 6.0 Key Commands.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

The following Visual C++ 6.0 key combinations are no longer available in Visual Studio.

Visual C++ 6.0 key combination	Visual C++ 6.0 command and scope	Comment
Alt + 3	ActivateWatchWindow (View)	No longer has a keyboard shortcut. See How to: Watch an Expression in the Debugger for more information. You can view the **QuickWatch** dialog box using SHIFT + F9.
Alt + F2	Bookmarks (Editing)	To move to the next bookmark, type F2. To move to the previous bookmark, type SHIFT + F2. To insert a bookmark, type CTRL + SHIFT + L. To clear bookmarks, type CTRL + SHIFT + F2. To set or remove a bookmark on the selected line, type CTRL + F2.
Ctrl + K	ConditionalDown	This keyboard shortcut is no longer available.
Ctrl + Shift + J	ConditionalUpExtend	This keyboard shortcut is no longer available.
Alt + Shift + F11	DebugMemoryPrevFormat (Debugging)	This keyboard shortcut is no longer available. To change to a different format, right-click the **Memory** window and select a format from the shortcut menu.
Ctrl + 7	InsertAcceleratorTable (Inserting)	In the Accelerator Editor, type INSERT.
Ctrl + 5	InsertBitmap (Inserting)	Use the New File or Add New Item dialog box.
Ctrl + 8	InsertStringTable (Inserting)	Use the Add Resource dialog box.
Ctrl + 6	InsertToolbar (Inserting)	Use the Add Resource dialog box.
Ctrl + 9	InsertVersionInfo (Inserting)	Use the Add Resource dialog box.
Ctrl + Shift + B	LayoutArrangeButtonsBottom (Layout)	In the Dialog Editor, use CTRL + B (Format.ButtonBottom)
Alt + F7	ProjectSettings (Project)	Use the Options dialog box to set options for your project.
Ctrl + Shift + F8	SelectColumn	No longer supported. See Changing and Selecting Text.

Ctrl + G	SelectDialogGuideType	No longer supported. See Changing and Selecting Text.
Ctrl + F8	SelectLine (Editing)	No longer supported. See Changing and Selecting Text.
Alt + Shift + L	SentenceCut (Editing)	No longer supported. See Changing and Selecting Text.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 6.0 Default Shortcut Keys

Shortcut Keys

Changes to Visual C++ 6.0 Key Commands

Due to changes and addition of features in Visual C++, the following key combinations have been reassigned.

Note
Many key combinations have been removed. For more information, see Obsolete Visual C++ 6.0 Key Commands.

For information on changing default combinations, see How to: Work with Shortcut Key Combinations.

File, Project, View, and Window Management Keyboard Shortcut Changes

Visual C++ 6.0 key combination	Version 6.0 command(s) and scope	Comment
CTRL + SHIFT + G	FileGoTo (File)	Obsolete.
CTRL + ALT + P	ProjectSelectTool (Project)	Obsolete.
ALT + 4	ActivateVariablesWindow (View)	Displays the Locals window. See Variable Windows for more information.
CTRL + W	ClassWizard (View)	Obsolete. See Where Are ClassWizard and WizardBar in Visual C++ .NET? for more information.
ALT + F6	WindowDockingView (Window management)	Obsolete. This key command now displays the next window pane. A window's ability to dock depends on the current view and the window type. See Window Types for more information.

Debugger Keyboard Shortcut Changes

Visual C++ 6.0 key combination	Version 6.0 command(s)	Comment
CTRL + F11	DebugToggleMixedMode	This command still toggles between the code and the disassembler, but you must be in Break mode to use the key combination.
Alt + F11	DebugMemoryNextFormat	Obsolete. This key combination now opens the Macro Explorer.

Editing and Dialog Layout Keyboard Shortcut Changes

Visual C++ 6.0 key combination	Version 6.0 command(s) and scope	Comment
F4	GoToNextErrorTag (Edit)	You can double-click an error in the Output Window to locate it in the code editor. This global key command displays the next error.
SHIFT + F4	GoToPrevErrorTag (Edit)	You can double-click an error in the Output Window to locate it in the code editor. This global key command displays the previous error.
CTRL + J	ConditionalUp (Edit)	Obsolete.
F8	SelectChar (Edit)	Obsolete. You can use the StickyKeys accessibility feature to toggle selection mode. See Microsoft Windows Help for using this feature.
CTRL + T	TypeInfo (Edit)	Obsolete. In the text editor, this key combination now transposes selected characters. (Edit.CharTranspose)

ALT + UP ARROW	LayoutSpaceEvenlyDown (Dialog Layout)	Obsolete. Use CTRL + DOWN ARROW.

Image Editor and Tools Keyboard Shortcut Changes

The Visual C++ 6.0 keyboard shortcuts in the Image Editor are no longer available. For an updated list, see Accelerator Keys for the Image Editor.

Key combination	Version 6.0 command(s) and scope	Comment
CTRL + B	ImagePickupBrush (Image editor)	Obsolete.
INSERT	ImageNewDevice (Image editor)	Obsolete.
CTRL + SHIFT + N	ImageOutlinedRoundRectTool (Image editor)	Obsolete.
CTRL + SHIFT + E	ImageOutlinedEllipseTool (Image editor)	Obsolete.
ALT + F12	Browse (Tools)	Obsolete.

Insert Keyboard Shortcut Changes

Visual C++ 6.0 key combination	Version 6.0 command(s)	Comment
CTRL + 1	InsertDialog	Obsolete. Use the Add Resource dialog box.
CTRL + 2	InsertMenu	Obsolete. Use the Add Resource dialog box.
CTRL + 3	InsertCursor	Obsolete. Use the Add Resource dialog box or use the Add New Item dialog box (CTRL + SHIFT + A) and from the Templates pane, select Cursor (.cur).
CTRL + 4	InsertIcon	Obsolete. Use the Add Resource dialog box, or use the Add New Item dialog box (CTRL + SHIFT + A) and from the Templates pane, select Icon (.ico).
CTRL + R	InsertResource	Obsolete. Use the Add Resource dialog box, or use the Add New Item dialog box (CTRL + SHIFT + A) and from the Templates pane, select Resource file (.rc). To add an existing resource file, Use the Add Existing Item dialog box and locate the .rc file to add.

See Also

Other Resources

Visual C++ 6.0 Default Shortcut Keys

Shortcut Keys

Control Manipulation Shortcut Keys, Visual C++ 6.0 Default Shortcut Option

Use the following shortcut key combinations to move, select, and change the size of controls on design surfaces.

Note

The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Note

To see a list of shortcut keys that have changed or are no longer supported in Visual C++ .NET, see Obsolete Visual C++ 6.0 Key Commands and Changes to Visual C++ 6.0 Key Commands.

Command name	Shortcut keys	Description
Edit.MoveControlDown	DOWN ARROW CTRL + DOWN ARROW	Moves the selected control down in increments of 1 on the design surface.
Edit.MoveControlDownGrid	DOWN ARROW	Moves the selected control down in increments of 8 on the design surface.
Edit.MoveControlLeft	LEFT ARROW CTRL + LEFT ARROW	Moves the control to the left in increments of 1 on the design surface.
Edit.MoveControlLeftGrid	LEFT ARROW	Moves the control to the left in increments of 8 on the design surface.
Edit.MoveControlRight	RIGHT ARROW CTRL + RIGHT ARROW	Moves the control to the right in increments of 1 on the design surface.
Edit.MoveControlRightGrid	RIGHT ARROW	Moves the control to the right in increments of 8 on the design surface.
Edit.MoveControlUp	UP ARROW CTRL + UP ARROW	Moves the control up in increments of 1 on the design surface.
Edit.MoveControlUpGrid	UP ARROW	Moves the control up in increments of 8 on the design surface.
Edit.SelectNextControl	TAB	Moves to the next control on the page.
Edit.SelectPreviousControl	SHIFT + TAB	Moves back to the previously selected control on the page.
Edit.SizeControlDown	SHIFT + DOWN ARROW CTRL + SHIFT + DOWN ARROW	Increases the height of the control in increments of 1 on the design surface.
Edit.SizeControlDownGrid	SHIFT + DOWN ARROW	Increases the height of the control in increments of 8 on the design surface.

Edit.SizeControlLeft	SHIFT + LEFT ARROW CTRL + SHIFT + LEFT ARROW	Reduces the width of the control in increments of 1 on the design surface.
Edit.SizeControlLeftGrid	SHIFT + LEFT ARROW	Reduces the width of the control in increments of 8 on the design surface.
Edit.SizeControlRight	SHIFT + RIGHT ARROW CTRL + SHIFT + RIGHT ARROW	Increases the width of the control in increments of 1 on the design surface.
Edit.SizeControlRightGrid	SHIFT + RIGHT ARROW	Increases the width of the control in increments of 8 on the design surface.
Edit.SizeControlUp	SHIFT + UP ARROW CTRL + SHIFT + UP ARROW	Decreases the height of the control in increments of 1 on the design surface.
Edit.SizeControlUpGrid	SHIFT + UP ARROW	Decreases the height of the control in increments of 8 on the design surface.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 6.0 Default Shortcut Keys

Shortcut Keys

Database Tools Shortcut Keys, Visual C++ 6.0 Default Shortcut Option

Use the following shortcut key combinations in the Database Designer or Query Designer.

Note

The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Note

To see a list of shortcut keys that have changed or are no longer supported in Visual C++ .NET, see Obsolete Visual C++ 6.0 Key Commands and Changes to Visual C++ 6.0 Key Commands.

Command name	Shortcut keys	Description
Database.Run	CTRL + E	Executes the currently active database object.
Database.RunSelection	CTRL + Q	Executes the current selection in the SQL editor.
Database.StepInto	ALT + F5	Steps into debug mode for the currently active database object.
Query.Run	CTRL + R	Executes the query.
View.Diagram	CTRL + 1	Displays the Diagram pane of the Query Designer.
View.Grid	CTRL + 2	Displays the Grid pane of the Query Designer. Available only in the Query Designer.
View.Results	CTRL + 4	Displays the Results pane of the Query Designer.
View.SQL	CTRL + 3	Displays the SQL pane of the Query Designer.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 6.0 Default Shortcut Keys

Shortcut Keys

Debug Shortcut Keys, Visual C++ 6.0 Default Shortcut Option

Use the following shortcut key combinations while debugging your code.

Note

The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Note

To see a list of shortcut keys that have changed or are no longer supported in Visual C++ .NET, see Obsolete Visual C++ 6.0 Key Commands and Changes to Visual C++ 6.0 Key Commands.

Command name	Shortcut keys	Description
Debug.ApplyCodeChanges	ALT + F10	Apply changes made to code without stopping debug mode. See Edit and Continue for more information.
Debug.Autos	CTRL + ALT + V, A	Displays the **Autos** window to view the values of variables currently in the scope of the current line of execution within the current procedure. See Breaking Execution for more information.
Debug.BreakAll	CTRL + ALT + BREAK	Temporarily stops execution of all processes in a debugging session. Available only in run mode.
Debug.Breakpoints	ALT + F9 CTRL + ALT + B	Displays the **New Breakpoint** dialog box, where you can add and modify breakpoints. See Breakpoints Overview for more information.
Debug.CallStack	ALT + 7 CTRL + ALT + C	Displays the **Call Stack** window to display a list of all active procedures or stack frames for the current thread of execution. Available only in run mode. See How to: Use the Call Stack Window for more information.
Debug.ClearAllBreakpoints	CTRL + SHIFT + F9	Clears all the breakpoints in the project.
Debug.Disassembly	ALT + 8	Displays the **Disassembly** window. See How to: Use the Disassembly Window for more information.
Debug.EnableBreakpoint	CTRL + F9	Enables breakpoint at the current line.
Debug.Exceptions	CTRL + ALT + E	Displays the Exceptions dialog box. See Continuing Execution After an Exception for more information.
Debug.Immediate	CTRL + ALT + I	Displays the Immediate window, where you can evaluate expressions and execute individual commands.
Debug.Locals	ALT + 4 CTRL + ALT + V, L	Displays the **Locals** window to view the variables and their values for each procedure in the current stack frame.

Debug.Memory1	ALT + 6 CTRL + ALT + M, 1	Displays the Memory 1 window to view large buffers, strings, and other data that do not display clearly in the Watch or Variables window.
Debug.Memory2	CTRL + ALT + M, 2	Displays the Memory 2 window to view large buffers, strings, and other data that do not display clearly in the Watch or Variables window.
Debug.Memory3	CTRL + ALT + M, 3	Displays the Memory 3 window to view large buffers, strings, and other data that do not display clearly in the Watch or Variables window.
Debug.Memory4	CTRL + ALT + M, 4	Displays the Memory 4 window to view large buffers, strings, and other data that do not display clearly in the Watch or Variables window.
Debug.Modules	CTRL + ALT + U	Displays the **Modules** window, which allows you to view the .dll or .exe files used by the program. See How to: Use the Modules Window for more information.
Debug.NewBreakpoint	CTRL + B	Inserts or clears a breakpoint in the current line of code.
Debug.QuickWatch	CTRL + ALT + Q SHIFT + F9	Displays the **QuickWatch** dialog box with the current value of the selected expression. Available only in break mode. Use this command to check the current value of a variable, property, or other expression for which you have not defined a watch expression.
Debug.Registers	ALT + 5 CTRL + ALT + G	Displays the **Registers** window, which displays registers content for debugging native-code applications. See How to: Use the Registers Window for more information.
Debug.Restart	CTRL + SHIFT + F5	Terminates a debugging session, rebuilds, and then starts running the application from the beginning again. Available in break and run modes.
Debug.RunningDocuments	CTRL + ALT + N	Displays the **Running Documents** window, which displays the set of documents that are in the process you are debugging. Available in run mode.
Debug.RunToCursor	CTRL + F10	In break mode, resumes execution of your code from the current statement to the selected statement. The Current Line of Execution margin indicator appears in the Margin Indicator bar. See Run to the Cursor Location for more information.
Debug.SetNextStatement	CTRL + SHIFT + F10	Sets the execution point to the line of code you choose. See Setting the Execution Point for more information.
Debug.ShowNextStatement	ALT + NUM *	Highlights the next statement to be executed.
Debug.Start	F5	Automatically attaches the debugger and runs the application from the startup form specified in the <Project> Properties dialog box. Changes to Continue if in break mode.
Debug.StartWithoutDebugging	CTRL + F5	Runs the code without invoking the debugger.
Debug.StepInto	F11	Executes code one statement at a time, following execution into function calls. See Stepping Into for more information.

Debug.StepOut	SHIFT + F11	Executes the remaining lines of a function in which the current execution point lies. See Stepping for more information.
Debug.StepOver	F10	Executes the next line of code but does not follow execution through any function calls.
Debug.StopDebugging	SHIFT + F5	Stops running the current application in the program. Available in break and run modes.
Debug.This	CTRL + ALT + V, T	Displays the **This** window, which allows you to view the data members of the object associated with the current method.
Debug.Threads	CTRL + ALT + H	Displays the **Threads** window to view all of the threads for the current process and information about them. See How to: Use the Threads Window for more information.
Debug.ToggleBreakpoint	F9	Sets or removes a breakpoint at the current line.
Debug.ToggleDisassembly	CTRL + F11	Displays the disassembly information for the current source file. Available only in break mode.
Tools.DebugProcesses	CTRL + ALT + P	Displays the **Processes** window, which allows you to debug multiple programs at the same time in a single solution. See How to: Use the Processes Window for more information.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 6.0 Default Shortcut Keys

Shortcut Keys

Dialog Editor Shortcut Keys, Visual C++ 6.0 Default Shortcut Option

Use the following shortcut key combinations only when you work in the Dialog editor.

Note

The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Note

To see a list of shortcut keys that have changed or are no longer supported in Visual C++ .NET, see Obsolete Visual C++ 6.0 Key Commands and Changes to Visual C++ 6.0 Key Commands.

Command name	Shortcut keys	Description
Edit.CheckMnemonics	CTRL + M	Checks for duplicate mnemonics in the dialog box.
Format.AlignBottoms	CTRL + DOWN ARROW	Aligns the selected controls horizontally with the bottom-most control selected on the dialog box.
Format.AlignCenters	SHIFT + F9	Aligns the selected controls so that they are centered vertically on the dialog box.
Format.AlignLefts	CTRL + LEFT ARROW	Aligns the selected controls vertically with the left-most control selected on the dialog box.
Format.AlignMiddles	F9	Aligns the selected controls so that they are centered horizontally on the dialog box.
Format.AlignRights	CTRL + RIGHT ARROW	Aligns the selected controls vertically with the right-most control selected on the dialog box.
Format.AlignTops	CTRL + UP ARROW	Aligns the selected controls horizontally with the top most control selected on the dialog box.
Format.ButtonBottom	CTRL + B	Aligns selected controls along the bottom edge of the dialog box.
Format.ButtonRight	CTRL + R	Aligns the selected controls along the right edge of the dialog box.
Format.CenterHorizontal	CTRL + SHIFT + F9	Centers the control with the horizontal center of the dialog box.
Format.CenterVertical	CTRL + F9	Centers the control with the vertical center of the dialog box.
Format.SizetoContent	SHIFT + F7	Reduces or increases the size of the selected control to match the content of the control.

Format.SpaceAcross	ALT + RIGHT ARROW ALT + LEFT ARROW	Aligns the selected controls so that the controls are spaced evenly apart horizontally.
Format.SpaceDown	ALT + UP ARROW ALT + DOWN ARROW	Aligns the selected controls so that the controls are spaced evenly apart vertically.
Format.TabOrder	CTRL + D	Displays a number beside each control, indicating the order in which the controls are selected when you tab.
Format.TestDialog	CTRL + T	Displays a testable version of the current dialog box.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 6.0 Default Shortcut Keys

Shortcut Keys

Global Shortcut Keys, Visual C++ 6.0 Default Shortcut Option

Use the following shortcut key combinations in various places within the integrated development environment (IDE).

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Command name	Shortcut keys	Description
Edit.Copy	CTRL + INSERT CTRL + C	Copies the currently selected item to the system clipboard.
Edit.Cut	SHIFT + DELETE CTRL + X	Removes the currently selected item and places it on the system clipboard.
Edit.CycleClipboardRing	CTRL + SHIFT + V CTRL + SHIFT + INSERT	Removes the currently selected item and places it on the system clipboard.
Edit.ListMembers	CTRL + ALT + T	Lists members of current class for statement completion when editing code.
Edit.GotoNextLocation	F8	Moves the cursor to the next item, such as a task in the Task List window or a search match in the Find Results window. Each time you press F8, you move to the next item in the list.
Edit.GotoPreviousLocation	SHIFT + F8	Moves the cursor to the previous item in the Task List window or Find Results window.
Edit.GoToReference	SHIFT + F12	Displays the reference of the selection in the code.
Edit.Paste	CTRL + V SHIFT + INSERT	Inserts the Clipboard contents at the insertion point.
Edit.Redo	CTRL + Y CTRL + SHIFT + Z SHIFT + ALT + BACKSPACE	Restores the previously undone action.
Edit.SelectionCancel	ESC	Closes a menu or dialog box, cancels an operation in progress, or places focus in the current document window.
Edit.Undo	CTRL + Z ALT + BACKSPACE	Reverses the last editing action.
File.Print	CTRL + P	Displays the Print dialog box, where you can choose printer settings.
File.SaveAll	CTRL + SHIFT + S	Saves all documents in the current solution and any files in the external files project.
File.SaveSelectedItems	CTRL + S	Saves the selected items in the current project.
Tools.GoToCommandLine	CTRL + /	Places the caret in the Find/Command box on the Standard toolbar.

View.NextTask	F4 CTRL + SHIFT + F12	Moves to the next task in the Task List window.
View.PopBrowseContext	CTRL + NUM *	Bookmarks your latest Go To definition or reference search.
View.PreviousTask	SHIFT + F4	Moves to the previous task in the Task List window.
View.ViewCode	CTRL + ALT + 0	Displays the selected item in Code view of the editor.
View.ViewDesigner	SHIFT + F7	Displays the selected item in Design view of the editor.
View.WebNavigateBack	ALT + LEFT ARROW	Displays the previous page in the viewing history. Only available in the Web Browser window.
View.WebNavigateForward	ALT + RIGHT ARROW	Displays the next page in the viewing history. Only available in the Web Browser window.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 6.0 Default Shortcut Keys

Shortcut Keys

HTML Designer Shortcut Keys, Visual C++ 6.0 Default Shortcut Option

Use the following shortcut key combinations only while editing in the HTML Designer. Most keys combinations are available only in either HTML view or Design view.

Note

The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Note

To see a list of shortcut keys that have changed or are no longer supported in Visual C++ .NET, see Obsolete Visual C++ 6.0 Key Commands and Changes to Visual C++ 6.0 Key Commands.

Command name	Shortcut keys	Description
Edit.ListMembers	CTRL + J	Lists members of current class for statement completion when editing code. Available only in HTML view.
Edit.ParameterInfo	CTRL + SHIFT + SPACE	Displays a tool tip that contains information for the current parameter, based on the current language. Available only in HTML view.
Edit.View Bottom	CTRL + PAGE DOWN	Moves to the bottom of the current document. Available only in HTML view.
Edit.View Top	CTRL + PAGE UP	Moves to the top of the current window. Available only in HTML view.
Format.Bold	CTRL + B	Toggles the selected text between bold and normal. Available only in Design view.
Format.DecreaseIndent	CTRL + SHIFT + T	Decreases the selected paragraph by one indent unit. Available only in Design view.
Format.IncreaseIndent	CTRL + T	Indents the selected paragraph by one indent unit. Available only in Design view.
Format.Italic	CTRL + I	Toggles the selected text between italic and normal. Available only in Design view.
Format.LockElement	CTRL + SHIFT + K	Prevents an absolutely positioned element from being inadvertently moved. Available only in Design view.
Format.ShowGrid	CTRL + G	Toggles show grid. Only available in Design view.

Format.SnaptoGrid	CTRL + SHIFT + G	Specifies that elements are aligned using an invisible grid. You can set grid spacing on the HTML Designer Display page of the Options dialog box, and the grid will be changed the next time you open a document. Available only in Design view.
Format.Underline	CTRL + U	Toggles the selected text between underlined and normal. Available only in Design view.
Insert.Bookmark	CTRL + SHIFT + L	Inserts a bookmark at the selected location. Available only in Design view.
Insert.DIV	CTRL + J	Inserts `<div></div>` in the current HTML document. Available only in Design view.
Insert.Hyperlink	CTRL + L	When text is selected, displays the Hyperlink dialog box. Available only in Design view.
Insert.Image	CTRL + SHIFT + W	Displays the Insert Image dialog box. Available only available in Design view.
View.Details	CTRL + SHIFT + Q	Displays signal icons for HTML elements that do not have a visual representation, such as comments, scripts, and anchors for absolutely positioned elements. Available only in Design view.
View.NextView	CTRL + PAGE DOWN	HTML Editor: Toggles between Design view and HTML view. XML Editor: Toggles between Data view and Schema view.
View.VisibleBorders	CTRL + Q	Displays a 1-pixel border around HTML elements that support a BORDER attribute and has it set to zero, such as tables, table cells, and divisions. Available only in Design view.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 6.0 Default Shortcut Keys

Shortcut Keys

Image Editor Shortcut Keys, Visual C++ 6.0 Default Shortcut Option

Use the following shortcut key combinations in the Image editor.

Note

The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Note

To see a list of shortcut keys that have changed or are no longer supported in Visual C++ .NET, see Obsolete Visual C++ 6.0 Key Commands and Changes to Visual C++ 6.0 Key Commands.

Command name	Shortcut keys	Description
Image.DrawOpaque	CTRL + J	Toggles the current selection to be rendered as either opaque or transparent.
Image.FlipHorizontal	CTRL + H	Flips the image from right to left on the horizontal axis. Available only in the Image editor. See Flipping an Image for more information.
Image.FlipVertical	SHIFT + ALT + H	Flips the image from top to bottom on the vertical axis. Available only in the Image editor. See Flipping an Image for more information.
Image.NewImageType	INSERT	Displays the New Icon Image Type dialog box, which allows you to select a new image type to create. Available only in the Image editor.
Image.Rotate90Degrees	CTRL + SHIFT + H	Rotates the image 90 degrees clockwise. Available only in the Image editor. See Flipping an Image for more information.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 6.0 Default Shortcut Keys

Shortcut Keys

Integrated Help Shortcut Keys, Visual C++ 6.0 Default Shortcut Option

Use the following shortcut key combinations to view and move among topics in Help.

Note

The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Note

To see a list of shortcut keys that have changed or are no longer supported in Visual C++ .NET, see Obsolete Visual C++ 6.0 Key Commands and Changes to Visual C++ 6.0 Key Commands.

Command name	Shortcut keys	Description
Help.Contents	CTRL + ALT + F1	Displays the Contents window for the documentation contained in MSDN.
Help.DynamicHelp	CTRL + F1	Displays the **Dynamic Help** window, which displays links to topics that are related to the item with current focus in the product. See How to: Customize Dynamic Help for more information.
Help.F1Help	F1	Displays a topic from Help that corresponds to the current user interface selected.
Help.Index	CTRL + ALT + F2	Displays the **Index** window for the documentation contained in MSDN.
Help.Indexresults	SHIFT + ALT + F2	Displays the **Index Results** window, which lists the topics that contain the keyword selected in the Index window.
Help.Nexttopic	ALT + DOWN ARROW	Displays the next topic in the table of contents. Available only in the Help (Web) browser window.
Help.Previoustopic	ALT + UP ARROW	Displays the previous topic in the table of contents. Available only in the Help (Web) browser window.
Help.Search	CTRL + ALT + F3	Displays the **Search** window, which allows you to search for words or phrases in the documentation contained in MSDN.
Help.Searchresults	SHIFT + ALT + F3	Displays the **Search Results** window, which displays a list of topics that contain the string specified in the **Look for** box of the Search window.
Help.WindowHelp	SHIFT + F1	Displays a topic from help that corresponds to the current user interface selected.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 6.0 Default Shortcut Keys

Shortcut Keys

Macro Shortcut Keys, Visual C++ 6.0 Default Shortcut Option

Use the following shortcut key combinations when you work with macros.

Note

The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Note

To see a list of shortcut keys that have changed or are no longer supported in Visual C++ .NET, see Obsolete Visual C++ 6.0 Key Commands and Changes to Visual C++ 6.0 Key Commands.

Command name	Shortcut keys	Description
Tools.MacrosIDE	ALT + F11	Launches the Macros IDE, Visual Studio Macros.
Tools.RecordTemporaryMacro	CTRL + SHIFT + R	Places the environment in macro record mode.
Tools.RunTemporaryMacro	CTRL + SHIFT + P	Plays back a recorded macro.
View.MacroExplorer	ALT + `	Displays the Macro Explorer window, which lists all available macros in the current solution.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 6.0 Default Shortcut Keys

Shortcut Keys

Miscellaneous Editor Shortcut Keys, Visual C++ 6.0 Default Shortcut Option

Use the following shortcut key combinations in either the Accelerator Editor or the String Editor.

Note
To see a list of shortcut keys that have changed or are no longer supported in Visual C++ .NET, see Obsolete Visual C++ 6.0 Key Commands and Changes to Visual C++ 6.0 Key Commands.

Command name	Shortcut keys	Description
Edit.NewAccelerator	INSERT	Adds a new entry for an accelerator key. Available only in the Accelerator editor.
Edit.NewString	INSERT	Adds a new entry in the string table. Available only in the String editor.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Concepts

Visual Studio Settings

Other Resources

Shortcut Keys

Object Browser Shortcut Keys, Visual C++ 6.0 Default Shortcut Option

Use the following shortcut key combinations while you work in the Object Browser window.

Note
To see a list of shortcut keys that have changed or are no longer supported in Visual C++ .NET, see Obsolete Visual C++ 6.0 Key Commands and Changes to Visual C++ 6.0 Key Commands.

Command name	Shortcut keys	Description
Edit.FindSymbol	ALT + F12 CTRL + SHIFT + Y	Displays the Find Symbol dialog box.
Edit.GoToDeclaration	CTRL + F12 CTRL + ALT + F12	Displays the definition of the selected symbol in the code.
Edit.GoToDefinition	F12	Displays the declaration for the selected symbol in code.
View.FindSymbolResults	CTRL + ALT + Y	Displays the Find Symbol Results window.
View.ObjectBrowser	CTRL + ALT + J	Displays the Object Browser to view the classes, properties, methods, events, and constants available for packages and object libraries and the procedures in your project.
View.ObjectBrowserBack	ALT + -	Moves back to the previously selected object in the selection history of the Object Browser.
View.ObjectBrowserForward	SHIFT + ALT + -	Moves forward to the next object in the selection history of the Object Browser.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 6.0 Default Shortcut Keys

Shortcut Keys

Project Shortcut Keys, Visual C++ 6.0 Default Shortcut Option

Use the following shortcut key combinations to add new items to a project, build a project, or open files or projects.

Note

The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Note

To see a list of shortcut keys that have changed or are no longer supported in Visual C++ .NET, see Obsolete Visual C++ 6.0 Key Commands and Changes to Visual C++ 6.0 Key Commands.

Command name	Shortcut keys	Description
Build.BuildSolution	F7	Builds the solution.
Build.Cancel	CTRL + BREAK	Cancels the build in progress.
Build.Compile	CTRL + F7	Creates an object file containing machine code, linker directives, sections, external references, and function/data names for the selected file.
File.NewFile	CTRL + N	Displays the New File dialog box where you can select a new file to add to the current project.
File.NewProject	CTRL + SHIFT + N	Displays the New Project dialog box where you can create projects and add them to the current solution.
File.OpenFile	CTRL + O	Displays the Open File dialog box where you can select to open an existing file.
File.OpenProject	CTRL + SHIFT + O	Displays the Open Project dialog box where you can add existing projects to your solution.
File.AddExistingItem	SHIFT + ALT + A	Displays the Add Existing Item dialog box, which allows you to add an existing file to the current project.
File.AddNewItem	CTRL + SHIFT + A	Displays the Add New Item dialog box, which allows you to add a new file to the current project.
Project.Override	CTRL + ALT + INSERT	Allows you to override base class methods in a derived class. Available for C#.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 6.0 Default Shortcut Keys

Shortcut Keys

Search and Replace Shortcut Keys, Visual C++ 6.0 Default Shortcut Option

Use the following shortcut key combinations with the Find and Replace dialog box.

Note

The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Note

To see a list of shortcut keys that have changed or are no longer supported in Visual C++ .NET, see Obsolete Visual C++ 6.0 Key Commands and Changes to Visual C++ 6.0 Key Commands.

Command name	Shortcut keys	Description
Edit.Find	CTRL + F	Displays the Find dialog box.
Edit.FindinFiles	CTRL + SHIFT + F	Displays the Find in Files dialog box.
Edit.FindNext	F3	Finds the next occurrence of the previous search text.
Edit.FindNextSelected	CTRL + F3	Finds the next occurrence of the currently selected text in the document.
Edit.FindPrevious	SHIFT + F3	Finds the previous occurrence of the search text.
Edit.FindPreviousSelected	CTRL + SHIFT + F3	Finds the previous occurrence of the currently selected text, or the word at the caret.
Edit.GoToFindCombo	CTRL + D	Places the caret in the Find/Command line on the Standard toolbar.
Edit.HiddenText	ALT + F3, H	Selects or clears the Search Hidden Text option for Find dialog box.
Edit.IncrementalSearch	CTRL + I	Starts incremental search. If incremental search is started but you have not typed any characters, recalls the previous pattern. If text has been found, searches for the next occurrence.
Edit.MatchCase	ALT + F3, C	Selects or clears the Match Case option for find and replace operations.
Edit.RegularExpression	ALT + F3, R	Selects or clears the Regular expression option so that special characters can be used in find and replace operations.
Edit.Replace	CTRL + H	Displays the Replace dialog box.
Edit.ReplaceinFiles	CTRL + SHIFT + H	Displays the Replace in Files dialog box.

Edit.ReverseIncrementalSearch	CTRL + SHIFT + I	Changes the direction of incremental search to begin at the bottom of the file and progress towards the top.
Edit.StopSearch	ALT + F3, S	Halts the current Find in Files operation.
Edit.Up	ALT + F3, B	Selects or clears the Search Up option for find and replace operations.
Edit.WholeWord	ALT + F3, W	Selects or clears the Match whole word option for find and replace operations.
Edit.Wildcard	ALT + F3, P	Selects or clears the Wildcard option for find and replace operations.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 6.0 Default Shortcut Keys

Shortcut Keys

Text Manipulation Shortcut Keys, Visual C++ 6.0 Default Shortcut Option

Use the following shortcut key combinations in text editors to delete, move, and format text in an open document.

Note

The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Note

To see a list of shortcut keys that have changed or are no longer supported in Visual C++ .NET, see Obsolete Visual C++ 6.0 Key Commands and Changes to Visual C++ 6.0 Key Commands.

Command name	Shortcut keys	Description
Edit.BreakLine	ENTER SHIFT + ENTER	Inserts a new line.
Edit.CharTranspose	CTRL + T	Swaps the characters on either side of the insertion point. For example, `AC|BD` becomes `AB|CD`. Available only in text editors.
Edit.ClearBookmarks	CTRL + SHIFT + F2 CTRL + K, CTRL + L	Removes all unnamed bookmarks in the current document.
Edit.CollapsetoDefinitions	CTRL + M, CTRL + O	Automatically determines logical boundaries for creating regions in code, such as procedures, and then hides them.
Edit.CommentSelection	CTRL + K, CTRL + C	Marks the current line of code as a comment, using the correct comment syntax for the programming language.
Edit.CompleteWord	CTRL + SPACE ALT + RIGHT ARROW	Displays Word Completion based on the current language.
Edit.Delete	DELETE	Deletes one character to the right of the cursor.
Edit.DeleteBackwards	BACKSPACE SHIFT + BACKSPACE	Deletes one character to the left of the cursor.
Edit.DeleteHorizontalWhiteSpace	CTRL + K, CTRL + \	Collapses white space in the selection, or deletes white space adjacent to the cursor if no selection.
Edit.FormatDocument	CTRL + K, CTRL + D	Applies the indenting and space formatting for the language as specified on the Formatting pane of the language in the Text Editor section of the Options dialog box.
Edit.FormatSelection	ALT + F8 CTRL + K, CTRL + F	Correctly indents the selected lines of code based on the surrounding lines of code.
Edit.HideSelection	CTRL + M, CTRL + H	Hides the selected text. A signal icon marks the location of the hidden text in the file.
Edit.InsertTab	TAB	Indents the line of text a specified number of spaces, such as five.

Edit.LineCut	CTRL + L SHIFT + ALT + L	Cuts all selected lines, or the current line if nothing has been selected, to the clipboard.
Edit.LineDelete	CTRL + SHIFT + L	Deletes all selected lines, or the current line if no selection has been made.
Edit.LineOpenAbove	CTRL + ENTER	Inserts a blank line above the insertion point.
Edit.LineOpenBelow	CTRL + SHIFT + ENTER	Inserts a blank line below the insertion point.
Edit.LineTranspose	SHIFT + ALT + T	Moves the line containing the insertion point below the next line.
Edit.MakeLowercase	CTRL + U	Changes the selected text to lowercase characters.
Edit.MakeUppercase	CTRL + SHIFT + U	Changes the selected text to uppercase characters.
Edit.OvertypeMode	INSERT	Toggles between insert and overtype insertion modes. Available only when working in text editors.
Edit.StopHidingCurrent	CTRL + M, CTRL + U	Removes the outlining information for the currently selected region.
Edit.StopOutlining	CTRL + M, CTRL + P	Removes all outlining information from the entire document.
Edit.SwapAnchor	CTRL + R, CTRL + P	Swaps the anchor and end points of the current selection.
Edit.TabLeft	SHIFT + TAB	Moves selected lines to the left one tab stop.
Edit.ToggleAllOutlining	CTRL + M, CTRL + L	Toggles all previously marked hidden text sections between hidden and display states.
Edit.ToggleBookmark	CTRL + F2 CTRL + K, CTRL + K	Sets or removes a bookmark at the current line.
Edit.ToggleOutliningExpansion	CTRL + M, CTRL + M	Toggles the currently selected hidden text section between the hidden and display state.
Edit.ToggleTaskListShortcut	CTRL + K, CTRL + H	Sets or removes a shortcut at the current line.
Edit.ToggleWordWrap	CTRL + R, CTRL + R	Enables or disables word wrap in an editor.
Edit.UncommentSelection	CTRL + K, CTRL + U	Removes the comment syntax from the current line of code.
Edit.ViewWhiteSpace	CTRL + SHIFT + 8 CTRL + R, CTRL + W	Shows or hides spaces and tab marks.

Edit.WordDelete ToEnd	CTRL + DELETE	Deletes the word to the right of the insertion point.
Edit.WordDelete ToStart	CTRL + BACKSPACE	Deletes the word to the left of the insertion point.
Edit.WordTransp ose	CTRL + SHIFT + T	Transposes the words on either side of the insertion point. For example, `main int` woul d be changed to read `int main`.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 6.0 Default Shortcut Keys

Shortcut Keys

Text Navigation Shortcut Keys, Visual C++ 6.0 Default Shortcut Option

Use the following shortcut key combinations in text editors to move in an open document.

Note

The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Note

To see a list of shortcut keys that have changed or are no longer supported in Visual C++ .NET, see Obsolete Visual C++ 6.0 Key Commands and Changes to Visual C++ 6.0 Key Commands.

Command name	Shortcut keys	Description
Edit.CharLeft	LEFT ARROW	Moves the cursor one character to the left.
Edit.CharRight	RIGHT ARROW	Moves the cursor one character to the right.
Edit.DocumentEnd	CTRL + END	Moves the insertion point to the last line of the document.
Edit.DocumentStart	CTRL + HOME	Moves the insertion point to the first line of the document.
Edit.GoTo	CTRL + G	Displays the Go To Line dialog box.
Edit.GotoBrace	CTRL +]	Moves the insertion point to the next brace in the document.
Edit.LineDown	DOWN ARROW	Moves the cursor down one line.
Edit.LineEnd	END	Moves the cursor to the end of the current line.
Edit.LineStart	HOME	Moves the cursor to the beginning of the line.
Edit.LineUp	UP ARROW	Moves the cursor up one line.
Edit.NextBookmark	F2 CTRL + K, CTRL + N	Moves to the next bookmark in the document.
Edit.PageDown	PAGE DOWN	Scrolls down one screen in the editor window.
Edit.PageUp	PAGE UP	Scrolls up one screen in the editor window.
Edit.PreviousBookmark	SHIFT + F2 CTRL + K, CTRL + P	Moves to the previous bookmark.
Edit.QuickInfo	CTRL + K, CTRL + I	Displays Quick Info, based on the current language.
Edit.ScrollLineDown	CTRL + DOWN ARROW	Scrolls text down one line. Available in text editors only.
Edit.ScrollLineUp	CTRL + UP ARROW	Scrolls text up one line. Available in text editors only.
Edit.ShowTileGrid	Ctrl + Shift + G	Exposes the tile grid. Available in text editors only.

Edit.WordNext	CTRL + RIGHT ARROW	Moves the insertion point to the right one word.
Edit.WordPrevious	CTRL + LEFT ARROW	Moves the insertion point to the left one word.
View.BrowseNext	CTRL + NUM + CTRL + SHIFT + 1	Displays the next symbol reference or definition.
View.BrowsePrevious	CTRL + NUM – CTRL + SHIFT + 2	Displays the previous symbol reference or definition.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 6.0 Default Shortcut Keys

Shortcut Keys

Text Selection Shortcut Keys, Visual C++ 6.0 Default Shortcut Option

Use the following shortcut key combinations in text editors to select text in an open document.

Note

The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Note

To see a list of shortcut keys that have changed or are no longer supported in Visual C++ .NET, see Obsolete Visual C++ 6.0 Key Commands and Changes to Visual C++ 6.0 Key Commands.

Command name	Shortcut keys	Description
Edit.CharLeftExtend	SHIFT + LEFT ARROW	Moves the cursor to the left one character, extending the selection.
Edit.CharLeftExtendColumn	SHIFT + ALT + LEFT ARROW	Moves the cursor to the left one character, extending the column selection.
Edit.CharRightExtend	SHIFT + RIGHT ARROW	Moves the cursor to the right one character, extending the selection.
Edit.CharRightExtendColumn	SHIFT + ALT + RIGHT ARROW	Moves the cursor to the right one character, extending the column selection.
Edit.DocumentEndExtend	CTRL + SHIFT + END	Selects the text from the insertion point to the last line of the document.
Edit.DocumentStartExtend	CTRL + SHIFT + HOME	Selects the text from the insertion point to the first line of the document.
Edit.GotoBraceExtend	CTRL + SHIFT +]	Moves the insertion point to the next brace, extending the selection.
Edit.LineDownExtend	SHIFT + DOWN ARROW	Extends text selection down one line, starting at the location of the insertion point.
Edit.LineDownExtendColumn	SHIFT + ALT + DOWN ARROW	Moves the cursor down one line, extending the column selection.
Edit.LineEndExtend	SHIFT + END	Selects text from the insertion point to the end of the current line.
Edit.LineEndExtendColumn	SHIFT + ALT + END	Moves the insertion point to the end of the line, extending the column selection.
Edit.LineStartExtend	SHIFT + HOME	Selects text from the insertion point to the start of the line.
Edit.LineStartExtendColumn	SHIFT + ALT + HOME	Moves the insertion point to the start of the line, extending the column selection.
Edit.LineUpExtend	SHIFT + DOWN ARROW SHIFT + UP ARROW	Selects text up line by line starting from the location of the insertion point.

Edit.LineUpExtendColumn	SHIFT + ALT + UP ARROW	Moves the cursor up one line, extending the column selection.
Edit.PageDownExtend	SHIFT + PAGE DOWN	Extends selection down one page.
Edit.PageUpExtend	SHIFT + PAGE UP	Extends selection up one page.
Edit.SelectAll	CTRL + A	Selects everything in the current document.
Edit.SelectCurrentWord	CTRL + W	Selects the word containing the insertion point or the word to the right of the insertion point.
Edit.SelectToLastGoBack	CTRL + =	Select from the current location in the editor back to the previous location in the editor.
Edit.ViewBottomExtend	CTRL + SHIFT + PAGE DOWN	Moves the cursor to the last line in view, extending the selection.
Edit.ViewTopExtend	CTRL + SHIFT + PAGE UP	Extends the selection to the top of the current window.
Edit.WordNextExtend	CTRL + SHIFT + RIGHT ARROW	Extends the selection one word to the right.
Edit.WordNextExtendColumn	CTRL + SHIFT + ALT + RIGHT ARROW	Moves the cursor to the right one word, extending the column selection.
Edit.WordPreviousExtend	CTRL + SHIFT + LEFT ARROW	Extends the selection one word to the left.
Edit.WordPreviousExtendColumn	CTRL + SHIFT + ALT + LEFT ARROW	Moves the cursor to the left one word, extending the column selection.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 6.0 Default Shortcut Keys

Shortcut Keys

Tool Window Shortcut Keys, Visual C++ 6.0 Default Shortcut Option

Use the following shortcut key combinations to display specific tool windows.

Note

The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Note

To see a list of shortcut keys that have changed or are no longer supported in Visual C++ .NET, see Obsolete Visual C++ 6.0 Key Commands and Changes to Visual C++ 6.0 Key Commands.

Command name	Shortcut keys	Description
Tools.CommandWindowMarkMode	CTRL + SHIFT + M	Places the Command Window in a mode allowing for selection of text within the window.
View.ClassView	CTRL + SHIFT + C	Displays the Class View window.
View.CommandWindow	CTRL + ALT + A	Displays the Command window, which allows you to type commands that manipulate the IDE.
View.DocumentOutline	CTRL + ALT + D	Displays the Document Outline window to view the flat or hierarchical outline of the current document.
View.Favorites	CTRL + ALT + F	Displays **Help Favorites**, which lists shortcuts to Web pages. See How to: Save Links to Favorite Information or more information.
View.Output	ALT + 2 CTRL + ALT + O	Displays the Output window to view status messages at run time.
View.PropertiesWindow	ALT + ENTER	Displays the Properties window, which lists the design-time properties and events for the currently selected item.
View.ResourceView	CTRL + SHIFT + E	Displays the Resource View window.
View.ServerExplorer	CTRL + ALT + S	Displays the Server Explorer windows, which allows you to view and manipulate database servers, event logs, message queues, XML Web services, and many other operating system services.
View.SolutionExplorer	CTRL + ALT + L	Displays Solution Explorer, which lists the projects and files in the current solution.
View.TaskList	CTRL + ALT + K	Displays the Task List window where you customize, categorize, and manage tasks, comments, shortcuts, warnings and error messages.

View.Toolbox	CTRL + ALT + X	Displays the Toolbox, which contains controls and other items that can be included or used with y our code.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 6.0 Default Shortcut Keys

Shortcut Keys

Window Management Shortcut Keys, Visual C++ 6.0 Default Shortcut Option

Use the following shortcut key combinations to move, close, or navigate among tool and document windows.

Note

The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Note

To see a list of shortcut keys that have changed or are no longer supported in Visual C++ .NET, see Obsolete Visual C++ 6.0 Key Commands and Changes to Visual C++ 6.0 Key Commands.

Command name	Shortcut keys	Description
View.FullScreen	SHIFT + ALT + ENTER	Toggles Full Screen mode on and off.
View.NavigateBackward	CTRL + -	Goes back to the previous document or window in the navigation history.
View.NavigateForward	CTRL + SHIFT + -	Moves forward to the document or window next in the navigation history.
Window.ActivateDocumentWindow	ESC ALT + 0	Closes a menu or dialog box, cancels an operation in progress, or places focus in the current document window.
Window.CloseDocumentWindow	CTRL + F4	Closes the current MDI child window.
Window.CloseToolWindow	SHIFT + ESC	Closes the current tool window.
Window.NextDocumentWindow	CTRL + F6 CTRL + TAB	Cycle through the MDI child windows one window at a time.
Window.NextPane	ALT + F6	Moves to the next tool window.
Window.NextSplitPane	F6	Moves to the next pane of a split pane view of a single document.
Window.NextTab	CTRL + PAGE DOWN	Moves to the next tab in the document or window.
Window.PreviousDocumentWindow	CTRL + SHIFT + F6 CTRL + SHIFT + TAB	Moves to the previous document in the editor or designer.
Window.PreviousPane	SHIFT + ALT + F6	Moves to the previously selected window.
Window.PreviousSplitPane	SHIFT + F6	Moves to the previous pane of a document in split pane view.
Window.PreviousTab	CTRL + PAGE UP	Moves to the previous tab in the document or window.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 6.0 Default Shortcut Keys
Shortcut Keys

Visual C++ 2.0 Default Shortcut Keys

This section lists the default key combinations available for the Visual C++ 2.0 keyboard scheme. For information on changing default combinations, see How to: Work with Shortcut Key Combinations.

Control Manipulation Shortcut Keys	Object Browser Shortcut Keys
Database Tools Shortcut Keys	Project Shortcut Keys
Debug Shortcut Keys	Search and Replace Shortcut Keys
Dialog Editor Shortcut Keys	Text Manipulation Shortcut Keys
Global Shortcut Keys	Text Navigation Shortcut Keys
HTML Designer Shortcut Keys	Text Selection Shortcut Keys
Integrated Help Shortcut Keys	Tool Window Shortcut Keys
Macro Shortcut Keys	Window Management Shortcut Keys

See Also

Reference

Visual C++ 2.0 Default Shortcut Keys

Concepts

Visual Studio Settings

Other Resources

Shortcut Keys

Control Manipulation Shortcut Keys, Visual C++ 2.0 Default Shortcut Option

Use the following shortcut key combinations to move, select, and change the size of controls on design surfaces.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Command name	Shortcut keys	Description
Edit.MoveControlDown	DOWN ARROW	Moves the selected control down in increments of 1 on the design surface.
Edit.MoveControlLeft	LEFT ARROW	Moves the control to the left in increments of 1 on the design surface.
Edit.MoveControlRight	RIGHT ARROW	Moves the control to the right in increments of 1 on the design surface.
Edit.MoveControlUp	UP ARROW	Moves the control up in increments of 1 on the design surface.
Edit.SizeControlDown	SHIFT + DOWN ARROW	Increases the height of the control in increments of 1 on the design surface.
Edit.SizeControlLeft	SHIFT + LEFT ARROW	Reduces the width of the control in increments of 1 on the design surface.
Edit.SizeControlRight	SHIFT + RIGHT ARROW	Increases the width of the control in increments of 1 on the design surface.
Edit.SizeControlUp	SHIFT + UP ARROW	Decreases the height of the control in increments of 1 on the design surface.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 2.0 Default Shortcut Keys

Shortcut Keys

Database Tools Shortcut Keys, Visual C++ 2.0 Default Shortcut Option

Use the following shortcut key combinations in the Database Designer or Query Designer.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Command name	Shortcut keys	Description
Database.StepInto	CTRL + E	Steps into debug mode for the currently active database object.
Query.Run	CTRL + R	Executes the query.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 2.0 Default Shortcut Keys

Shortcut Keys

Debug Shortcut Keys, Visual C++ 2.0 Default Shortcut Option

Use the following shortcut key combinations while debugging your code.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Command name	**Shortcut keys**	**Description**
Debug.ApplyCodeChanges	ALT + F10	Apply changes made to code without stopping debug mode.
Debug.Breakpoints	CTRL + B	Displays the **Breakpoints** window. See Breakpoints Overview for more information.
Debug.CallStack	ALT + 6 CTRL + K	Displays the **Call Stack** window to display a list of all active procedures or stack frames for the current thread of execution. Available only in run mode. See How to: Use the Call Stack Window for more information.
Debug.Disassembly	ALT + 7	Displays the **Disassembly** window. See How to: Use the Disassembly Window for more information.
Debug.EnableBreakpoint	CTRL + SHIFT + F9	Enables breakpoint at the current line.
Debug.Locals	ALT + 3	Displays the **Locals** window to view the variables and their values for each procedure in the current stack frame.
Debug.Memory1	ALT + 5	Displays the **Memory 1** window to view large buffers, strings, and other data that do not display clearly in the Watch or Variables window.
Debug.QuickWatch	SHIFT + F9	Displays the **QuickWatch** dialog box with the current value of the selected expression. Available only in break mode. Use this command to check the current value of a variable, property, or other expression for which you have not defined a watch expression.
Debug.Registers	ALT + 4	Displays the **Registers** window, which displays registers content for debugging native-code applications. See How to: Use the Registers Window for more information.
Debug.Restart	SHIFT + F5	Terminates a debugging session, rebuilds, and then starts running the application from the beginning again. Available in break and run modes.
Debug.RunToCursor	F7	In break mode, resumes execution of your code from the current statement to the selected statement. The Current Line of Execution margin indicator appears in the Margin Indicator bar.
Debug.SetNextStatement	CTRL + SHIFT + F7	Sets the execution point to the line of code you choose.

Debug.ShowNextStatement	ALT + NUM *	Highlights the next statement to be executed.
Debug.Start	F5	Automatically attaches the debugger and runs the application from the startup form specified in the <Project> Properties dialog box. Changes to Continue if in break mode.
Debug.StartWithoutDebugging	CTRL + F5	Runs the code without invoking the debugger.
Debug.StepInto	F8	Executes code one statement at a time, following execution into function calls.
Debug.StepOut	SHIFT + F7	Executes the remaining lines of a function in which the current execution point lies.
Debug.StepOver	F10	Executes the next line of code but does not follow execution through any function calls.
Debug.StopDebugging	ALT + F5	Stops running the current application in the program. Available in break and run modes.
Debug.ToggleBreakpoint	F9	Sets or removes a breakpoint at the current line.
Debug.ToggleDisassembly	CTRL + F7	Displays the disassembly information for the current source file. Available only in break mode.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 2.0 Default Shortcut Keys

Shortcut Keys

Dialog Editor Shortcut Keys, Visual C++ 2.0 Default Shortcut Option

Use the following shortcut key combinations only while you work in the Dialog Editor.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Command name	Shortcut keys	Description
Edit.CheckMnemonics	CTRL + M	Checks for duplicate mnemonics in the dialog box.
Format.AlignBottoms	CTRL + DOWN ARROW	Aligns the selected controls horizontally with the bottom-most control s elected on the dialog box.
Format.AlignCenters	SHIFT + F9	Aligns the selected controls so that they are centered vertically on the di alog box.
Format.AlignLefts	CTRL + LEFT ARROW	Aligns the selected controls vertically with the left-most control selected on the dialog box.
Format.AlignMiddles	F9	Aligns the selected controls so that they are centered horizontally on the dialog box.
Format.AlignRights	CTRL + RIGHT ARROW	Aligns the selected controls vertically with the right-most control selecte d on the dialog box.
Format.AlignTops	CTRL + UP ARROW	Aligns the selected controls horizontally with the top most control select ed on the dialog box.
Format.ButtonBottom	CTRL + SHIFT + B	Aligns selected controls along the bottom edge of the dialog box.
Format.ButtonRight	CTRL + B	Aligns the selected controls along the right edge of the dialog box.
Format.CenterHorizontal ly	CTRL + SHIFT + F9	Centers the selected controls horizontally on the dialog box.
Format.CenterVertically	CTRL + F9	Centers the selected controls vertically on the dialog box.
Format.MakeHorizontal SpacingEqual	ALT + LEFT ARROW ALT + RIGHT ARROW	Spaces controls horizontally equal on the dialog box.
Format.MakeVerticalSpa cingEqual	ALT + DOWN ARROW ALT + UP ARROW	Spaces controls vertically equal on the dialog box.
Format.SizetoContent	F7	Reduces or increases the size of the selected control to match the conten t of the control.
Format.TestDialog	CTRL + T	Displays a testable version of the current dialog box.

View.TabOrder	CTRL + D	Displays a number beside each control, indicating the order in which the controls are selected when you tab.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 2.0 Default Shortcut Keys

Shortcut Keys

Global Shortcut Keys, Visual C++ 2.0 Default Shortcut Option

Use the following shortcut key combinations in various places within the integrated development environment (IDE).

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Command name	Shortcut keys	Description
Edit.Copy	CTRL + C CTRL + INSERT	Copies the currently selected item to the system clipboard.
Edit.Cut	CTRL + X SHIFT + DELETE CTRL + ALT + W	Removes the currently selected item to the system clipboard.
Edit.GotoNextLocation	F4	Moves the cursor to the next item, such as a task in the Task List window or a search match in the Find Results window. Each time you press F12, you move to the next item in the list.
Edit.GotoPreviousLocation	SHIFT + F4	Moves the cursor to the previous item in the Task List window or Find Results window.
Edit.GoToReference	SHIFT + F11 SHIFT + ALT + F11	Displays the reference of the selection in the code.
Edit.Paste	CTRL + V SHIFT + INSERT	Inserts the Clipboard contents at the insertion point.
Edit.Redo	CTRL + A CTRL + SHIFT + Z SHIFT + ALT + BACKSPACE	Restores the previously undone action.
Edit.SelectionCancel	ESC	Closes a menu or dialog box, cancels an operation in progress, or places focus in the current document window.
Edit.Undo	CTRL + Z ALT + BACKSPACE	Reverses the last editing action.
File.Print	CTRL + P CTRL + SHIFT + F12 CTRL + SHIFT + ALT + F2	Displays the Print dialog box where you can choose printer settings.
File.SaveAll	CTRL + SHIFT + S	Saves all documents in the current solution and any files in the external files project.
File.SaveSelectedItems	CTRL + S SHIFT + F12	Saves the selected items in the current project.
View.PopBrowseContext	CTRL + NUM *	Returns to the location preceding the last browse operation.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 2.0 Default Shortcut Keys

Shortcut Keys

HTML Designer Shortcut Keys, Visual C++ 2.0 Default Shortcut Option

Use the following shortcut key combinations only while editing in the HTML designer. Certain keys combinations are available only in a specific view of that designer.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Command name	**Shortcut keys**	**Description**
Edit.ListMembers	CTRL + ALT + T	Lists members of current class for statement completion when editing code. Available only in HTML view.
Edit.ParameterInfo	CTRL + SHIFT + SPACE	Displays a tool tip that contains information for the current parameter, based on the current language. Available only in HTML view.
Format.Bold	CTRL + B	Toggles the selected text between bold and normal. Available only in Design view.
Format.DecreaseIndent	CTRL + SHIFT + T	Decreases the selected paragraph by one indent unit. Available only in Design view.
Format.IncreaseIndent	CTRL + T	Indents the selected paragraph by one indent unit. Available only in Design view.
Format.Italic	CTRL + I	Toggles the selected text between italic and normal. Available only in Design view.
Format.Underline	CTRL + U	Toggles the selected text between underlined and normal. Available only in Design view.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 2.0 Default Shortcut Keys

Shortcut Keys

Integrated Help Shortcut Keys, Visual C++ 2.0 Default Shortcut Option

Use the following shortcut key combinations to view and move among topics in Help.

Command name	Shortcut keys	Description
Help.F1Help	F1	Displays a topic from Help that corresponds to the current user interface selected.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 2.0 Default Shortcut Keys

Shortcut Keys

Macro Shortcut Keys, Visual C++ 2.0 Default Shortcut Option

Use the following shortcut key combinations when you work with macros.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Command name	Shortcut keys	Description
Tools.RecordTemporaryMacro	CTRL + SHIFT + R	Places the environment in macro record mode.
Tools.RunTemporaryMacro	CTRL + SHIFT + P	Plays back a recorded macro.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 2.0 Default Shortcut Keys

Shortcut Keys

Object Browser Shortcut Keys, Visual C++ 2.0 Default Shortcut Option

Use the following shortcut key combinations while you work in the Object Browser window.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Command name	Shortcut keys	Description
Edit.FindSymbol	CTRL + F11 CTRL + ALT + F11	Displays the Find Symbol dialog box.
Edit.GoToDefinition	F11 ALT + F1	Displays the declaration for the selected symbol in code.
View.ObjectBrowser	SHIFT + ALT + F1	Displays the Object Browser to view the classes, properties, methods, events, and constants available for packages and object libraries and the procedures in your project.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 2.0 Default Shortcut Keys

Shortcut Keys

Project Shortcut Keys, Visual C++ 2.0 Default Shortcut Option

Use the following shortcut key combinations to add new items to a project, build a project, or open files or projects.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Command name	Shortcut keys	Description
Build.BuildSolution	SHIFT + F8	Builds the solution.
Build.Cancel	CTRL + BREAK	Cancel the build in progress.
Build.Compile	CTRL + F8	Creates an object file containing machine code, linker directives, sections, external references, and function/data names for the selected file.
Build.RebuildSolution	ALT + F8	Rebuilds the solution
Project.AddResource	CTRL + R	Add resource to current project.
File.NewFile	CTRL + N	Displays the New File dialog box where you can select a new file to add to the current project.
File.NewProject	CTRL + SHIFT + N	Displays the New Project dialog box where you can create projects and add them to the current solution.
File.OpenFile	CTRL + O CTRL + F12	Displays the Open File dialog box where you can select to open an existing file.
File.OpenProject	CTRL + SHIFT + O	Displays the Open Project dialog box where you can add existing projects to your solution.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 2.0 Default Shortcut Keys

Shortcut Keys

Search and Replace Shortcut Keys, Visual C++ 2.0 Default Shortcut Option

Use the following shortcut key combinations with the Find, **Replace**, Find in Files, and **Replace in Files** dialog boxes.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Command name	Shortcut keys	Description
Edit.Find	ALT + F3	Displays the Find dialog box.
Edit.FindNext	F3	Finds the next occurrence of the previous search text.
Edit.FindNextSelected	CTRL + F3	Finds the next occurrence of the currently selected text in the document.
Edit.FindPrevious	SHIFT + F3	Finds the previous occurrence of the search text.
Edit.FindPreviousSelected	CTRL + SHIFT + F3	Finds the previous occurrence of the currently selected text, or the word at the caret.
Edit.GoToFindCombo	ALT + A CTRL + F	Places the caret in the Find/Command line on the Standard toolbar.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 2.0 Default Shortcut Keys

Shortcut Keys

Finding and Replacing

Text Manipulation Shortcut Keys, Visual C++ 2.0 Default Shortcut Option

Use the following shortcut key combinations in text editors to delete, move, and format text in an open document.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Command name	Shortcut keys	Description
Edit.BreakLine	ENTER SHIFT + ENTER	Inserts a new line.
Edit.CompleteWord	CTRL + SPACE	Displays Word Completion based on the current language.
Edit.Delete	DELETE	Deletes one character to the right of the cursor.
Edit.DeleteBackwards	BACKSPACE SHIFT + BACKSPACE	Deletes one character to the left of the cursor.
Edit.FormatSelection	CTRL + SHIFT + F CTRL + ALT + I	Correctly indents the selected lines of code based on the surrounding lines of code.
Edit.InsertTab	TAB	Indents the line of text a specified number of spaces, such as five.
Edit.LineCut	CTRL + Y	Cuts all selected lines, or the current line if nothing has been selected, to the clipboard.
Edit.LineOpenAbove	CTRL + ENTER	Inserts a blank line above the insertion point.
Edit.LineOpenBelow	CTRL + SHIFT + ENTER	Inserts a blank line below the insertion point.
Edit.MakeLowercase	CTRL + U	Changes the selected text to lowercase characters.
Edit.MakeUppercase	CTRL + SHIFT + U	Changes the selected text to uppercase characters.
Edit.OvertypeMode	INSERT	Toggles between insert and overtype insertion modes. Available only when working in text editors.
Edit.SwapAnchor	CTRL + SHIFT + X	Swaps the anchor and end points of the current selection.
Edit.TabLeft	SHIFT + TAB	Moves selected lines to the left one tab stop.
Edit.ToggleBookmark	CTRL + F2	Sets or removes a bookmark at the current line.

Edit.ViewWhiteSpace	CTRL + ALT + T	Shows or hides spaces and tab marks.
Edit.WordDeleteToEnd	CTRL + DELETE	Deletes the word to the right of the insertion point.
Edit.WordDeleteToStart	CTRL + BACKSPACE	Deletes the word to the left of the insertion point.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 2.0 Default Shortcut Keys

Shortcut Keys

Text Navigation Shortcut Keys, Visual C++ 2.0 Default Shortcut Option

Use the following shortcut key combinations in text editors to move in an open document.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Command name	Shortcut keys	Description
Edit.CharLeft	LEFT ARROW	Moves the cursor one character to the left.
Edit.CharRight	RIGHT ARROW	Moves the cursor one character to the right. Available only in the .NET Framework Designer
Edit.DocumentEnd	CTRL + END	Moves the insertion point to the last line of the document.
Edit.DocumentStart	CTRL + HOME	Moves the insertion point to the first line of the document.
Edit.GoTo	CTRL + G	Displays the Go To Line dialog box.
Edit.GotoBrace	CTRL +] CTRL + M	Moves the insertion point to the next brace in the document.
Edit.LineDown	DOWN ARROW	Moves the cursor down one line.
Edit.LineEnd	END	Moves the cursor to the end of the current line.
Edit.LineStart	HOME	Moves the cursor to the beginning of the line.
Edit.LineUp	UP ARROW	Moves the cursor up one line.
Edit.NextBookmark	F2	Moves to the next bookmark in the document.
Edit.PageDown	PAGE DOWN	Scrolls down one screen in the editor window.
Edit.PageUp	PAGE UP	Scrolls up one screen in the editor window.
Edit.PreviousBookmark	SHIFT + F2	Moves to the previous bookmark.
Edit.QuickInfo	CTRL + T	Displays Quick Info, based on the current language.
Edit.ScrollLineDown	CTRL + DOWN ARROW	Scrolls text down one line. Available in text editors only.
Edit.ScrollLineUp	CTRL + UP ARROW	Scrolls text up one line. Available in text editors only.
Edit.WordNext	CTRL + RIGHT ARROW	Moves the insertion point to the right one word.
Edit.WordPrevious	CTRL + LEFT ARROW	Moves the insertion point to the left one word.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 2.0 Default Shortcut Keys

Shortcut Keys

Text Selection Shortcut Keys, Visual C++ 2.0 Default Shortcut Option

Use the following shortcut key combinations in text editors to select text in an open document.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Command name	Shortcut keys	Description
Edit.CharLeftExtend	SHIFT + LEFT ARROW	Moves the cursor to the left one character, extending the selection.
Edit.CharRightExtend	SHIFT + RIGHT ARROW	Moves the cursor to the right one character, extending the selection.
Edit.DocumentEndExtend	CTRL + SHIFT + END	Selects the text from the insertion point to the last line of the document.
Edit.DocumentStartExtend	CTRL + SHIFT + HOME	Selects the text from the insertion point to the first line of the document.
Edit.GotoBraceExtend	CTRL + SHIFT +] CTRL + SHIFT + M	Moves the insertion point to the next brace, extending the selection.
Edit.LineDownExtend	SHIFT + DOWN ARROW	Extends text selection down one line, starting at the location of the insertion point.
Edit.LineEndExtend	SHIFT + END	Selects text from the insertion point to the end of the current line.
Edit.LineStartExtend	SHIFT + HOME	Selects text from the insertion point to the start of the line.
Edit.LineUpExtend	SHIFT + UP ARROW	Selects text up, line by line, starting from the location of the insertion point.
Edit.PageDownExtend	SHIFT + PAGE DOWN	Extends selection down one page.
Edit.PageUpExtend	SHIFT + PAGE UP	Extends selection up one page
Edit.WordNextExtend	CTRL + SHIFT + RIGHT ARROW	Extends the selection one word to the right.
Edit.WordPreviousExtend	CTRL + SHIFT + LEFT ARROW	Extends the selection one word to the left.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 2.0 Default Shortcut Keys

Shortcut Keys

Tool Window Shortcut Keys, Visual C++ 2.0 Default Shortcut Option

Use the following shortcut key combinations to display specific tool windows.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Command name	Shortcut keys	Description
View.Output	ALT + 1	Displays the Output window to view status messages at run time.
View.PropertiesWindow	ALT + ENTER CTRL + W	Displays the Properties window, which lists the design-time properties and events for the currently selected item.
View.SolutionExplorer	ALT + 0	Displays Solution Explorer, which lists the projects and files in the current solution.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 2.0 Default Shortcut Keys

Shortcut Keys

Window Management Shortcut Keys, Visual C++ 2.0 Default Shortcut Option

Use the following shortcut key combinations to move, close, or navigate among tool and document windows.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

Command name	Shortcut keys	Description
View.NavigateBackward	CTRL + NUM *	Goes back to the previous document or window in the navigation history.
Window.ActivateDocumentWindow	ESC	Closes a menu or dialog box, cancels an operation in progress, or places focus in the current document window.
Window.CloseDocumentWindow	CTRL + F4	Closes the current MDI child window.
Window.CloseToolWindow	SHIFT + ESC	Closes the current tool window.
Window.Dockable	ALT + F6	Makes the current tool window dockable.
Window.NextDocumentWindow	CTRL + F6 CTRL + TAB	Cycle through the MDI child windows one window at a time.
Window.NextPane	F6	Moves to the next tool window.
Window.NextTab	CTRL + PAGE DOWN	Moves to the next tab in the document or window.
Window.PreviousDocumentWindow	CTRL + SHIFT + F6 CTRL + SHIFT + TAB	Moves to the previous document in the editor or designer.
Window.PreviousPane	SHIFT + F6	Moves to the previously selected window.
Window.PreviousTab	CTRL + PAGE UP	Moves to the previous tab in the document or window.

See Also

Tasks

How to: Work with Shortcut Key Combinations

Other Resources

Visual C++ 2.0 Default Shortcut Keys

Shortcut Keys

Library Changes in Visual C++ Releases

In This Section

ATL and MFC Changes

Standard C++ Library Changes

Reference

ATL Reference

ATL Server Reference

MFC Reference

.NET Framework Reference

OLE DB Templates Reference

Run-Time Library Reference

Shared Classes Reference

Standard C++ Library Reference

Related Sections

What's New in Visual C++

Porting and Upgrading Programs

Product Changes in Visual C++ Releases

See Also

Other Resources

Previous Product Changes

ATL and MFC Changes

The following topics describe changes in ATL and MFC.

- ATL and MFC Version Numbers
- ATL 7.1 and MFC 7.1
- ATL 7.0 and MFC 7.0

See Also

Other Resources

Library Changes in Visual C++ Releases

ATL and MFC Version Numbers

The versions of ATL shipped with Visual C++ as listed in the following table.

ATL version	Visual C++ version
1.0, 1.1, 2.0	None. Released to Web in Visual C++ 4.x time frame.
3.0	Visual C++ 6.0
7.0	Visual C++ .NET 2002
7.1	Visual C++ .NET 2003
8.0	Visual C++ 2005

The versions of MFC shipped with Visual C++ as listed in the following table.

MFC version	Visual C++ version
1.0	Microsoft C/C++ 7.0
2.0	Visual C++ 1.0
2.5	Visual C++ 1.5
3.0	Visual C++ 2.0
3.1	Visual C++ 2.1
3.2	Visual C++ 2.2
4.0	Visual C++ 4.0
4.1	Visual C++ 4.1
4.2	Visual C++ 4.2
4.21 (mfc42.dll)	Visual C++ 5.0
6.0 (mfc42.dll)	Visual C++ 6.0
7.0 (mfc70.dll)	Visual C++ .NET 2002
7.1 (mfc71.dll)	Visual C++ .NET 2003
8.0 (mfc80.dll)	Visual C++ 2005

See Also

Reference

ATL and MFC Changes

ATL and MFC Changes: ATL 7.1 and MFC 7.1

Note Some features mentioned in this topic may not still exist in the current version of Visual C++.

Changes have been made to the ATL and MFC libraries and wizards in Visual C++ .NET 2003, some of which may break existing code.

- Wizard issues when migrating a project from ATL 6.0 to ATL 7.1 (ATL)
- **CAtlServiceModuleT::InitializeSecurity** has been removed from **CAtlServiceModuleT** (ATL)
- **SetRfc822Time** function updated
- Projects created with default ATL Wizard will fail on Windows NT

Wizard issues when migrating a project from ATL 6.0 to ATL 7.1

When converting a Visual C++ 6.0 project to a Visual C++ .NET 2003 project, message handlers are added to the end of the message map. If the message map has a **CHAIN_*_MAP(...)** entry, items added by the wizard will be included at the end of the entry, and may not be processed.

For example, when migrating a project containing a Visual C++ 6.0 ATL control to a Visual C++ .NET 2003 project, the **OnInitDialog** handler is never called. This is because the **MESSAGE_HANDLER** is placed after the **CHAIN_MSG_MAP(...)** entry, instead of before.

To ensure the **OnInitDialog** handler is called, the line `CHAIN_MSG_MAP(CComControl<Name of class>)` must be included at the end of the **CHAIN_MSG_MAP(...)** entry. Alternately, the **OnInitDialog** base class implementation should perform the necessary initialization.

CAtlServiceModuleT::InitializeSecurity has been removed from CAtlServiceModuleT

In Visual Studio .NET 2003, this method is not implemented in the base class. For details, see CAtlServiceModuleT::InitializeSecurity.

SetRfc822Time function updated

To avoid potential buffer overruns, the SetRfc822Time function now has an extra parameter and a return value that indicates the length of string required to store the returned time data.

Projects created with default ATL Wizard will fail on Windows NT

In Visual Studio .NET 2003, the MIDL compiler has a default setting of **/robust**, which causes projects running under Windows NT 4 to stop responding.

To change the MIDL compiler flag to /no_robust

1. Right-click your project, and on the shortcut menu, click **Properties**.

   The **Project Properties** dialog box appears.

2. In the left pane, click **MIDL**, and then select **Command Line**.
3. Enter **/no_robust** in the **Additional Options** text box.

See Also
Reference
ATL and MFC Changes

ATL and MFC Changes: ATL 7.0 and MFC 7.0

Note Some features mentioned in this topic may not still exist in the current version of Visual C++..

Many improvements have been made to the ATL and MFC libraries since Visual C++ 6.0. Some of these changes may break existing code.

DLL Incompatibilities

The ATL and MFC DLL files shipping as part of Visual C++ .NET 2002 have been renamed to ATL70.dll and MFC70.dll, respectively.

The Visual C++ .NET ATL and MFC classes are not binary compatible with the same classes from previous releases, and therefore any source code built using mfc42.dll must be rebuilt with Visual Studio .NET. Any DLL or LIB files used by your application must also be rebuilt with Visual Studio .NET.

For example, a library containing an exported function taking **CString** as a parameter that has been built using Visual C++ 6.0 will give an unresolved external during linking with a Visual C++ .NET project.

ATL Module Classes

ATL 3.0 provided the class **CComModule**. In ATL 7.0, the functionality previously provided by **CComModule** is handled by several new classes. See ATL Module Classes for more information.

String Conversions

In versions of ATL up to and including ATL 3.0 in Visual C++ 6.0, string conversions using the macros in atlconv.h were always performed using the ANSI code page of the system (CP_ACP). Starting with ATL 7.0 in Visual C++ .NET, string conversions are performed using the default ANSI code page of the current thread, unless **_CONVERSION_DONT_USE_THREAD_LOCALE** is defined, in which case the ANSI code page of the system is used as before.

Note that the string conversion classes, such as **CW2AEX**, allow you to pass a code page to use for the conversion to their constructors. If a code page is not specified, the classes use the same code page as the macros.

For more information, see ATL and MFC String Conversion Macros.

CException Is Now an Abstract Base Class

CException is the base class for all exceptions in the Microsoft Foundation Class Library. Because **CException** is now an abstract base class, you cannot create **CException** objects directly; you must create objects of derived classes. If you do create an object directly, you will receive an error. For more information, see CException.

Converting from BSTR to CString

In Visual C++ 6.0, it was acceptable to use the following code:

```
BSTR bstr = SysAllocString(L"Hello");
CString str = bstr;
SysFreeString(bstr);
```

With new projects under Visual C++ .NET, this will cause the following error under ANSI builds:

```
error C2440: 'initializing' : cannot convert from 'BSTR' to
'ATL::CStringT<BaseType,StringTraits>'
```

There are now UNICODE and ANSI versions of **CString** (**CStringW** and **CStringA**). To flag any unnecessary overhead incurred by implicit conversions, constructors that take the inverse type (for example, **CStringA** taking a UNICODE argument, or **CStringW** taking an ANSI argument) are now tagged as explicit using the following entry in stdafx.h:

```
#define _ATL_CSTRING_EXPLICIT_CONSTRUCTORS
```

To work around this error, do one of the following:

- Use **CStringW** to avoid conversion:

  ```
  BSTR bstr = SysAllocString(L"Hello");
  ```

```
CStringW str = bstr;
SysFreeString(bstr);
```

- Explicitly call the constructor:

```
BSTR bstr = SysAllocString(L"Hello");
CString str = CString(bstr);
SysFreeString(bstr);
```

- Remove the line `#define _ATL_CSTRING_EXPLICIT_CONSTRUCTORS` from stdafx.h.

CTime Changes

CTime Class uses the underlying **__time64_t** data type. In MFC 6.0, CTime used the **time_t** data type which was then a 32-bit type. The reason for this change is to support times beyond 3:14:07 on January 19, 2038.

CComEnumImpl::Skip Changes

The CComEnumImpl::Skip method in versions before ATL 7.0 would not return the correct error code for an input value of 0. It would also handle large input values in an inconsistent manner. These behaviors were fixed in ATL 7.0.

CWnd::DestroyWindow Assertions

When a tooltip was displayed in CWnd::DestroyWindow, an assertion error would occur. As a result, in MFC 7.0, the following member variables were moved from **AFX_THREAD_STATE** to **AFX_MODULE_THREAD_STATE**:

- **CToolTipCtrl* m_pToolTip**
- **CWnd* m_pLastHit**
- **int m_nLastHit**
- **TOOLINFO m_lastInfo**
- **int m_nLastStatus**
- **CControlBar* m_pLastStatus**

LNK2001 Unresolved External Symbol Error

When calling a function in a static library or DLL that takes a **wchar_t** type (note that BSTR and LPWSTR resolve to **wchar_t***), you may get an LNK2001 unresolved external symbol error.

This error is caused by the /Zc:wchar_t compiler option, which is set to on by default in new MFC projects. This option causes the compiler to treat **wchar_t** as a native type. Before Visual C++ .NET, **wchar_t** was treated as an **unsigned short**.

If the main project and library do not use the same setting for /Zc:wchar_t, this will cause a mismatch of function signatures. To avoid this problem, rebuild the library with the /Zc:wchar_t compiler option, or turn it off in the main project using the **Treat wchar_t as Built-in Type** setting on the **Language** property page in the **Property Pages** dialog box.

Boolean Expressions Are Now of Type bool, Not BOOL

Consider the following class:

```
class CMyClass : public CObject
{
   BOOL bFlag;

   void Serialize (CArchive& ar))
   {
      if (ar.IsStoring())
         ar << (bFlag != FALSE); // breaking change
      else
         ar >> bFlag;
   }
};
```

Before Visual C++ .NET, the expression `bFlag != FALSE` evaluated as a **BOOL** and four bytes were written; in Visual C++ .NET,

it evaluates as a **bool** and one byte is written. This means that programs compiled with different versions of the compiler may produce mutually incompatible data files.

To avoid the problem, cast the expression to **BOOL**:

```
ar << (BOOL)(bFlag != FALSE);
```

CColorPropPage and CFontPropPage Have Been Removed

In previous versions of MFC, an ActiveX control displayed property pages for color or font properties by specifying the GUID **CLSID_CColorPropPage** or **CLSID_CFontPropPage**, respectively. These GUIDs pointed to the classes **CColorPropPage** and **CFontPropPage**, which are no longer implemented. Instead, use the GUIDs **CLSID_StockColorPage** and **CLSID_StockFontPage**. These are implemented by msstkprp.dll, so you must redistribute that DLL with your application.

ON_MESSAGE Changes

The function parameter in the ON_MESSAGE macro must match the type `afx_msg LRESULT (CWnd::*)(WPARAM, LPARAM)`.

OLE DB Templates Changes

You can find more information on changes in the OLE DB Templates as described in the Knowledge Base article "INFO: Porting Issues with Visual Studio .NET OLE DB Provider Template Classes" (Q321743). You can find Knowledge Base articles on the MSDN Library CD-ROM or at http://support.microsoft.com/support.

OLE DB Consumer Classes and Templates

As a general note, the accessor class must implement additional members. This is only necessary if you implement your own accessor class manually. If your accessor class derives from **CAccessor**, you need not do this.

Old behavior	**New behavior**
CRowset is a class.	**CRowset** is a class template and takes one parameter, *TAccessor*, an accessor class.
CBulkRowset is a class.	**CBulkRowset** is a class template.
The base class for **CArrayRowset** was a template parameter (default value was **CRowset**).	**CArrayRowset** always derives from **CBulkRowset**.
CDynamicAccessor::GetColumnInfo took three parameters.	**CDynamicAccessor::GetColumnInfo** has a new form that takes an additional parameter, *ppStringsBuffer*. Using this parameter eliminates a memory leak. The old method is deprecated.
Rowset, second parameter of the **CAccessorRowset** template, is a rowset class.	*TRowset*, second parameter of the **CAccessorRowset** template, is a rowset class template.
Rowset, second parameter of the **CTable** template, is a rowset class.	*TRowset*, second parameter of the **CTable** template, is a rowset class template.
Rowset, second parameter of the **CCommand** template, is a rowset class.	*TRowset*, second parameter of the **CCommand** template, is a rowset class template.
DEFINE_COMMAND macro	**DEFINE_COMMAND** macro is deprecated. Use **DEFINE_COMMAND_EX** instead.

OLE DB Provider Classes and Templates:

The internal implementation of many interfaces and methods has changed since Visual C++ 6.0. This may cause compatibility issues depending on whether your application overrides these methods.

Old behavior	**New behavior**

The rowset/accessor implementation used **CSimpleMap**/**CSimpleArray** classes. User-provided collection classes had to be **CSimpleMap**/**CSimpleArray** compatible.	The rowset/accessor implementation uses **CAtlMap**/**CAtlArray** classes. User-provided collection classes have to be **CAtlMap**/**CAtlArray** compatible. In addition, code that calls methods of these collection classes should be reviewed as there are significant differences between the **CAtl*** and **CSimple*** classes (parameters, return values, and so on) that can result in run-time errors.
ICommandImpl derived from **ICommand**.	**ICommandImpl** is a template that derives from the template's *CommandBase* argument (the default is **ICommand**).
ICommandTextImpl derived from **ICommandImpl<ICommandImpl<*T*>**.	**ICommandTextImpl** derives from **ICommandImpl<ICommandImpl<*T*, ICommandText>**. Note that here **ICommandImpl** derives from **ICommandText** (not the default **ICommand**).

See Also

Reference

ATL and MFC Changes

Standard C++ Library Changes

The Standard C++ Library encompasses the latest ANSI C++ extensions, including the Standard Template Library and a new iostream library. The Standard C++ Library provides new functionality, such as numerous algorithms that manipulate C++ objects, and a migration path for developers who want to move to the standard iostream.

Starting in Visual Studio .NET 2003, Visual C++ no longer ships the old iostream libraries.

The Standard C++ Library is a set of header files. A list of the header files can be found in the Standard C++ Library Overview. The new header files do not have the .h extension.

Additional upgrading issues are as follows:

- Standard C++ Library Changes: Visual C++ .NET 2003
- Standard C++ Library Changes: Upgrading Issues

See Also

Other Resources

Previous Product Changes

Standard C++ Library Changes: Visual C++ .NET 2003

If you are upgrading a program that uses Standard C++ Library code that compiled in Visual C++ 6.0, you should be aware of the following issues:

- reverse_iterator Changes
- Some Iterators Are No Longer the Same as Pointers
- MIN/MAX #define Change

reverse_iterator Changes

The names for some of the types defined by the Standard C++ Library reverse_iterator class have changed. Also, there are different template arguments for this class.

The following sample compiled in Visual C++ 6.0, but fails in the current version:

```
#include <iterator>
#include <vector>
typedef std::iterator<std::random_access_iterator_tag, char, int> random_it;

int main( )
{
   char c;

   // too many template args for VC7
   typedef std::reverse_iterator< random_it, char, char&, char *, int > rev_it;
   rev_it::reference_type x = c;
   rev_it::pointer_type y = 0;
   rev_it::iter_type z;

   /*
   // try the following code instead
   typedef std::reverse_iterator< random_it > rev_it;
   rev_it::reference a = c;
   rev_it::pointer b = 0;
   rev_it::iterator_type c1;
   */
}
```

Some Iterators Are No Longer the Same as Pointers

In some Standard C++ Library classes, iterators are no longer defined as pointer types.

The following sample compiled in Visual C++ 6.0, but fails on the indicated lines in the current version:

```
#include <string>
#include <vector>
#include <algorithm>

bool pred(int i) {
   return true;
};

int main()
{
   std::string str("test");
   const char *pszstr = str.begin();       // LINE 8: INCORRECT
   const char *pszStr2 = str.c_str();      // OK
   const char *pszStr3 = &(*str.begin());  // OK

   std::vector<int> v;
   int *pint = std::remove_if(v.begin(), v.end(), pred);   // LINE 13: INCORRECT
   std::vector<int>::iterator iint = std::remove_if(v.begin(), v.end(), pred);   // OK
}
```

MIN/MAX #define Change

The Standard C++ Library definition of **_MIN** and **_MAX** changed from:

```
#define _MAX   _cpp_max
#define _MIN   _cpp_min
```

to:

```
#define _MAX   (max)
#define _MIN   (min)
```

This makes definitions of **std::_MIN** invalid.

The following sample compiled in Visual C++ 6.0, but fails on the indicated line in the current version:

```
#include <xutility>
#include <stdlib.h>
using namespace std;
int main()
{
   std::_MAX(3,4);   // error
   _MAX(4,5);
}
```

See Also

Concepts

Standard C++ Library Changes

Standard C++ Library Changes: Upgrading Issues

The following topic discuss difficulties you may encounter with an earlier application and the new run-time libraries.

- Differences in iostream Implementation
- Earlier Projects Built with No Default Libraries
- C++ Exception Handling Must Be Enabled for the Standard C++ Library

Differences in iostream Implementation

The old iostream library was removed beginning in Visual C++ .NET 2003.

The main difference between the Standard C++ Library and previous run-time libraries is in the iostream library. Details of the iostream implementation have changed, and it may be necessary to rewrite parts of your code that use iostream if you want to link with the Standard C++ Library.

You will have to remove any old iostream headers (fstream.h, iomanip.h, ios.h, iostream.h, istream.h, ostream.h, streamb.h, and strstrea.h) you have included in your code and add one or more of the new Standard C++ iostream headers (<fstream>, <iomanip>, <ios>, <iosfwd>, <iostream>, <istream>, <ostream>, <sstream>, <streambuf>, and <strstream>, all without the .h extension).

The following list describes behavior in the new Standard C++ iostream library that differs from behavior in the old iostream library.

In the new Standard C++ iostream library:

- **open** functions do not take a third parameter (the protection parameter).
- You cannot create streams from file handles.
- With a couple of exceptions, all names in the new Standard C++ Library are in the **std** namespace. See Using C++ Library Headers for more information.
- You cannot open **ofstream** objects with the **ios::out** flag alone. The **ios::out** flag must be combined with another **ios** enumerator in a logical **OR**; for example, with **ios::in** or **ios::app**.
- **ios::good** no longer returns a nonzero value after reaching the end-of-file because the **eofbit** state is set.
- **ios::setf**(*_IFlags*) should not be used with a flag value of **ios::dec**, **ios::oct**, or **ios::hex** unless you know that none of the base flags are currently set. The formatted input/output functions and operators assume that only one base is set. Instead, use **ios_base**. For example, **setf**(**ios_base::oct, ios_base::basefield**) clears all base information and sets the base to octal.
- **ios::unsetf** returns **void** instead of the previous value.
- **istream::get**(**char&** *_Rch*) does not assign to *Rch* if there is an error.
- **istream::get**(**char*** *_Pch*, **int** *_Ncount*, **char** *_Delim*) is different in three ways:
    - When nothing is read, **failbit** is set.
    - An **eos** is always stored after characters extracted (this happens regardless of the outcome).
    - A value of -1 for *_Ncount* is an error.
- **istream::seekg** with an invalid parameter does not set **failbit**.
- The return type **streampos** is a class with overloaded operators. In functions that return a **streampos** value (such as **istream::tellg**, **ostream::tellp**, **strstreambuf::seekoff**, and **strstreambuf::seekpos**), you should cast the return value to the type required: **streamoff**, **fpos_t**, or **mbstate_t**.
- The first function parameter (*_Falloc*) in **strstreambuf::strstreambuf**(*_Falloc*, *_Ffree*) takes a **size_t** argument, not a **long**.

In addition to the above changes, the following functions, constants, and enumerators that are elements of the old iostream library are not elements of the new iostream library:

- **attach** member function of **filebuf**, **fstream ifstream**, and **ofstream**
- **fd** member function of **filebuf**, **fstream ifstream**, and **ofstream**
- **filebuf::openprot**
- **filebuf::setmode**
- **ios::bitalloc**
- **ios::nocreate**
- **ios::noreplace**
- **ios::sync_with_stdio**
- **streambuf::out_waiting**
- **streambuf::setbuf** (use **rdbuf** -> **pubsetbuf** for the same behavior)

Earlier Projects Built with No Default Libraries

You can build a project without default libraries by selecting /NODEFAULTLIB. If your previous project was built with no default libraries and you want to make iostream calls, you must name one of the new Standard C++ run-time libraries (Libcp.lib, Libcpmt.lib, Msvcprt.lib, and so on) or one of the old iostream run-time libraries (Libci.lib, Libcimt.lib, Msvcirt.lib, and so on) in order to link with the proper library.

In previous Visual C++ versions (4.1 and earlier), the run-time library names were Libc.lib, Libcmt.lib, and Msvcrt.lib. These libraries included the old iostream library. The old iostream library has now been removed from these libraries. If you do not choose to ignore default libraries and you include old iostream header files in your code, the old iostream run-time libraries (Libci.lib, Libcimt.lib, Msvcirt.lib, and so on) are linked by default. However, if you have chosen to ignore default libraries and have manually added one of the early run-time libraries, your iostream calls will now break.

C++ Exception Handling Must Be Enabled for the Standard C++ Library

Any file that uses the Standard C++ Library must be compiled with C++ exception handling enabled. For more information, see /GX.

See Also

Concepts

Standard C++ Library Changes

Supported Platforms (Visual C++)

Applications built using Microsoft Visual C++ 2005 can be targeted to any of the following platforms:

Operating System	x86	x64	Itanium
Windows 98, Windows 98 Second Edition	X		
Windows Millennium Edition	X		
Windows 2000	X		
Windows XP	X	X	
Windows Server 2003	X	X	X

For information on how to set the target platform, see How to: Configure Visual C++ Projects to Target 64-Bit Platforms.

See Also

Concepts

Visual Studio Hardware Requirements

Other Resources

Getting Started with Visual C++

Visual C++ Settings

Visual C++ settings are designed to optimize the user experience and to maximize the productivity of Visual C++ developers. It is part of the Visual Studio Settings feature, which enables you to customize and save your Visual Studio Integrated Development Environment (IDE) settings.

You can also port or reload your IDE settings on other computers. For more information, see How to: Share Settings Between Computers.

Windows and Views

Feature	Displayed by default?	Notes
Class View	Yes	• Filtering is enabled • Class View folders are enabled • Hides **Implemented Interfaces** nodes on classes
Command Window	No	
How to: Control the Dynamic Help Window	No	Does not display when you press F1.
Object Browser	Yes, in physical view	• Does not show inherited members • Displays all class members by default
Output Window	Yes, when build starts	Exposes **GotoNextError** commands through shortcut keys.
Solution Explorer	Yes	**Synchronize selected file with open file** in the Code Editor is OFF by default.
Start Page	Yes	Only displays the first time you start the IDE.
Task List (Visual Studio)	No	It will not display because of any build command.
Toolbox	Yes	• Displays components in alphabetical order. • Displays **Clipboard Ring** tab.

In addition, the following elements have certain behaviors enabled when you select the Visual C++ profile.

Dialog Boxes

Feature	Behavior
Quick Find, Find and Replace Window	
New Project Dialog Box	When other platforms are installed on the machine, the IDE enables the combo box that allows you to specify the targeted platform.
Options Dialog Box (Visual Studio)	The **More Options** button displays options for Source Control, Device Tools, Database Tools, HTML Designer, and XML Designer.

Keyboard

Feature	Behavior

Shortcut Keys	Visual C++ supports: • Visual C++ 6.0 Default Shortcut Keys, which is the default shortcut keys setting • BRIEF Emulation • EMACS Emulation • Visual C++ 2.0 Default Shortcut Keys

Miscellaneous IDE Elements

Feature	Behavior
Designer settings	The default view is the Code Editor, not the component designer view.
Main menus, shortcut menus, toolbars, commands	• Tool tips will appear for all menu items and tool bar buttons when the mouse hovers above them and will contain the corresponding shortcut key. • A platform selection drop-down list box will appear in the main tool bar.
Shell appearance	
Window management and layouts	Default layout of windows: • The Solution Explorer, Class View, and Resource View tool windows will display on the left side, at full length. • The Output window and Property Grid will not display by default. Default layout of debugger windows: • Autos, Locals, Watch1, and Registers windows will display on the left side. • Call Stack, Breakpoints, Threads, Modules, and Output will display on the right side. • The Output window will not display.

See Also

Tasks

How to: Change Select Settings

How to: Specify Settings for a Team

Reference

Import and Export Settings, Environment, Options Dialog Box

Visual C++ Editions

Note
Documentation for features that are not available in Visual C++ Express Edition or Visual C++ Standard Edition may be included in your documentation set.

Compilers

Feature	Microsoft Visual C++ Express Edition	Visual Studio 2005 Standard	Visual Studio 2005 Professional	Visual Studio 2005 Team System
32-bit x86 compiler	X	X	X	X
64-bit x64 compiler and cross-compiler		X	X	X
64-bit Itanium compiler and cross-compiler				X

Libraries and Headers

Feature	Microsoft Visual C++ Express Edition	Visual Studio 2005 Standard	Visual Studio 2005 Professional	Visual Studio 2005 Team System
Win32 SDK	(core only)	X	X	X
CRT library	X	X	X	X
CRT debug library		X	X	X
CRT source code		X	X	X
STL	X	X	X	X
ATL		X	X	X
ATL Server		X	X	X
MFC		X	X	X
.NET Framework class library	X	X	X	X
C++ Support Library		X	X	X
OpenMP			X	X

Integrated Development Environment

Feature	Microsoft Visual C++ Express Edition	Visual Studio 2005 Standard	Visual Studio 2005 Professional	Visual Studio 2005 Team System
Graphical integrated development environment	X	X	X	X

Tools

Feature	Microsoft Visual C++ Express Edition	Visual Studio 2005 Standard	Visual Studio 2005 Professional	Visual Studio 2005 Team System
Incremental Linker (Link.exe)	X	X	X	X
Program Maintenance Utility (Nmake.exe)	X	X	X	X
Lib Generator (Lib.exe)	X	X	X	X
Windows Resource Compiler (Rc.exe)	X	X	X	X
Windows Resource to Object Converter (CvtRes.exe)	X	X	X	X
Native Web Service Proxy Generator (SProxy.exe)			X	X
Browse Information Maintenance Utility (BscMake.exe)		X	X	X
C++ Name Undecorator (Undname.exe)		X	X	X
ClStencil.exe			X	X
COFF/PE Dumper (Dumpbin.exe)		X	X	X
COFF/PE Editor (Editbin.exe)		X	X	X
MASM (Ml.exe)			X	X
Visual C++ Web Deployment Tool (VCDeploy.exe)			X	X
Spy++		X	X	X
ErrLook		X	X	X
AtlTrace		X	X	X
Devenv.com		X	X	X
Inference Rules	X	X	X	X
Command Line Tool (Vcbuild.exe)	X	X	X	X
Profile Guided Optimizations			X	X

Debugging Features

Feature	Microsoft Visual C++ Express Edition	Visual Studio 2005 Standard	Visual Studio 2005 Professional	Visual Studio 2005 Team System
Native Debugging	X	X	X	X

Managed Debugging	X	X	X	X
Remote Debugging			X	X
SQL Debugging			X	X

Designers and Editors

Feature	Microsoft Visual C++ Express Edition	Visual Studio 2005 Standard	Visual Studio 2005 Professional	Visual Studio 2005 Team System
CSS Style Designer/Editor	X	X	X	X
HTML Designer/Editor	X	X	X	X
XML Editor	X	X	X	X
Source Code Editor	X	X	X	X
Windows Forms Designer	X	X	X	X
Data Designer		X	X	X
Resource Editors		X	X	X

Data Features

Feature	Microsoft Visual C++ Express Edition	Visual Studio 2005 Standard	Visual Studio 2005 Professional	Visual Studio 2005 Team System
Data Designer		X	X	X
Data Objects	X	X	X	X
Web Services	(Can consume Web services only)	X	X	X
Server Explorer			X	X

Automation and Extensibility

Feature	Microsoft Visual C++ Express Edition	Visual Studio 2005 Standard	Visual Studio 2005 Professional	Visual Studio 2005 Team System
Extensibility Object Models	X	X	X	X
Code Model	X	X	X	X
Project Model	X	X	X	X

Resource Editor Model		X	X	X
Wizard Model	X	X	X	X
Debugger Object Model	X	X	X	X

See Also

Concepts

Installation and Setup Essentials
What's New in Visual Studio 2005
What's New in Visual C++
Visual Database Tools Editions
Visual Studio Hardware Requirements

Project Templates in Visual C++ Editions

When you create a new project, icons in the **New Project** dialog box and **Add Project** dialog box represent the available project types and their templates. The project template that is associated with the project icon you choose determines the output type and other options available for that project. Not all project templates are available in all editions of Visual C++.

Note
Documentation for features that are not available in Visual C++ Express Edition or Visual C++ Standard Edition may be included in the documentation set for these editions.

Visual C++ Project Templates

The following table shows which Visual C++ project templates are available in the different editions of Visual Studio.

Template	**Microsoft Visual C++ Express Edition**	**Visual Studio 2005 Standard**	**Visual Studio 2005 Professional and above**
ASP.NET Web Service		X	X
ATL Project		X	X
ATL Server Project		X	X
ATL Server Web Service		X	X
ATL Smart Device Project		X	X
Class Library	X	X	X
CLR Console Application	X	X	X
CLR Empty Project	X	X	X
Custom Wizard		X	X
Empty Project	X	X	X
Makefile Project	X	X	X
MFC ActiveX Control		X	X
MFC Application		X	X
MFC DLL		X	X
MFC Smart Device ActiveX Control		X	X
MFC Smart Device Application		X	X
MFC Smart Device DLL		X	X
SQL Server Project			X

Test Project			X
Win32 Console Application	X	X	X
Win32 Project		X	X
Win32 Smart Device Project		X	X
Windows Forms Application	X	X	X
Windows Forms Control Library		X	X
Windows Service		X	X

See Also

Concepts

Visual C++ Projects

Installation and Setup Essentials

What's New in Visual Studio 2005

What's New in Visual C++

Visual C++ Guided Tour

There are several topics that introduce the different types of projects you can create with Microsoft Visual C++ 2005.

In This Section

How to: Compile a Native C++ Program from the Command Line

Creates a simple C++ program with a text editor and compiles it on the command line.

How to: Compile a C++ Program that Targets the CLR in Visual Studio

Creates a Visual C++ program with .NET classes and compiles it using the Visual Studio Development Environment.

How to: Compile a Code Example from the Help Topics

Describes how to locate sample Visual C++ programs in the help topics and how to compile and run them.

How to: Create a Standard C++ Program

In this procedure you can type in your own C++ program or use one of the sample programs.

How to: Create a Windows Forms Application

Creates a Windows Forms application using standard controls from the Toolbox.

How to: Convert Between Various String Types

Describes how to convert various Visual C++ strings from one type to another.

How to: Use Visual C++ Express Edition with the Microsoft Platform SDK

Details how to install the Microsoft Platform SDK for use in building Win32 application using Visual C++ Express Edition.

Walkthrough: Using the Visual Studio IDE

Creates a new standard C++ program and tests it functionality using Visual Studio.

Walkthrough: Creating a Windows Forms Control

Creates a Windows Forms control and an application to test it.

Walkthrough: Creating and Using a Managed Assembly

Creates a managed assembly and an application that references the assembly.

Walkthrough: Creating and Using a Dynamic Link Library

Creates a dynamic link library and an application that references the DLL.

Walkthrough: Creating and Using a Static Library

Creates a static library and an application that references it.

Introduction to Visual C++ for UNIX Users

Provides information for UNIX users who are new to Visual C++.

Reference

Visual C++ Reference

Provides information on the C and C++ languages as well as the various libraries they incorporate.

.NET Framework Class Library

Provides information on all the interfaces and types that are included in the .NET Framework.

C++/CLI Language Specification Standard

The Ecma specification of the C++/CLI language.

Related Sections

Integrated Development Environment for Visual Studio

Provides support for creating projects, writing and debugging code, and deploying applications.

Programming Guide

Contains conceptual and task-based topics on Visual C++ programming.

Building a C/C++ Program

Describes the various ways you can build projects using Visual Studio.

See Also

Other Resources

Getting Started with Visual C++
Getting Started with Visual C++ Express Edition

How to: Compile a Native C++ Program from the Command Line

Visual C++ includes a C++ compiler you can use to create everything from simple Visual C++ programs to Windows Forms applications and components.

In this procedure, you create simple Visual C++ programs with a text editor and compile them on the command line.

You can also compile Visual C++ programs you create with a text editor using the Visual Studio development environment. See How to: Compile a C++ Program that Targets the CLR in Visual Studio for more information.

You can use your own Visual C++ programs instead of typing in the simple programs shown in these procedures. You can also use any of the Visual C++ code sample programs included in the help topics. See How to: Compile a Code Example from the Help Topics for more information.

To create a Visual C++ source file and compile it on the command line

1. Open the **Visual Studio 2005 Command Prompt** window.

   Click the **Start** button, then point to **All Programs**, **Microsoft Visual Studio 2005**, **Visual Studio Tools**, and click **Visual Studio 2005 Command Prompt**.

Note
The **Visual Studio 2005 Command Prompt** automatically sets up the correct path to the Visual C++ compiler and any needed libraries, so it is used instead of the regular Command Prompt window. For more information, see Setting the Path and Environment Variables for Command-Line Builds.

2. At the command prompt, type **notepad simple.cpp** and press Enter.

   Click **Yes** when prompted to create a new file.

3. In Notepad, type the following lines:

   ```
   #include <iostream>

   int main()
   {
       std::cout << "This is a native C++ program." << std::endl;
       return 0;
   }
   ```

4. On the **File** menu, click **Save**. You have created a Visual C++ source file.

5. On the **File** menu, click **Exit** to close Notepad.

6. At the command line prompt, type **cl /EHsc simple.cpp** and press Enter. The **/EHsc** command line option instructs the compiler to enable C++ exception handling. For more information, see /EH (Exception Handling Model).

   The **cl.exe** compiler generates an executable program **simple.exe**.

   You can see the executable program name in the lines of output information the compiler displays.

7. To see a list of all files in the directory named **simple** with any extension, type **dir simple.*** and press Enter.

   The **.obj** file is an intermediate format file you can ignore.

8. To run the **simple.exe** program, type **simple** and press Enter.

   The program displays this text and exits:

   ```
   This is a native C++ program.
   ```

9. To close the command prompt window, type **exit** and press Enter.

Compiling a Visual C++ Program with .NET Classes

This procedure shows the command line you use to compile a Visual C++ program using .NET Framework classes.

The /clr (Common Language Runtime Compilation) compiler option is necessary because this program uses .NET classes and requires the Visual C++ compiler to include the necessary .NET libraries. The Visual C++ compiler generates an .exe that contains MSIL code rather than machine executable instructions.

You can use the steps in this procedure to compile any of the sample Visual C++ programs included in the help topics. For example, the How to: Compile a Code Example from the Help Topics topic refers to a sample program that uses .NET classes to calculate the amount of time since the last Windows start-up occurred.

To compile a Visual C++ .NET console application on the command line

1. Open the **Visual Studio 2005 Command Prompt** window.

   Click the **Start** button, point to **All Programs**, **Microsoft Visual Studio 2005**, **Visual Studio Tools**, and click **Visual Studio 2005 Command Prompt**.

2. At the command prompt, type **notepad simpleclr.cpp** and press Enter:

   Click **Yes** when prompted to create a new file.

3. In Notepad, type the following lines:

   ```
   int main()
   {
       System::Console::WriteLine("This is a Visual C++ program.");
   }
   ```

4. On the **File** menu, click **Save**.

   You have created a Visual C++ source file using a .NET class (Console) that is located in the System namespace.

5. On the **File** menu, click **Exit** to close Notepad.

6. At the command line prompt, type **cl /clr simpleclr.cpp** and press Enter:

7. The **cl.exe** compiler generates an executable program **simpleclr.exe**.

8. To see a list of all files in the directory named **simpleclr** with any extension, type **dir simpleclr.*** and press Enter.

   The **.obj** file is an intermediate format file you can ignore.

   The **.manifest** file is an XML file that contains information about the assembly (an assembly is the .NET unit of deployment, such as an **.exe** program or **.dll** component or library).

9. To run the **simpleclr.exe** program, type **simpleclr** and press Enter.

   The program displays this text and exits:

   ```
   This is a Visual C++ program.
   ```

10. To close the command prompt window, type **exit** and press Enter.

See Also

Other Resources

Visual C++ Guided Tour
C++ Language Reference
Building a C/C++ Program

How to: Compile a C++ Program that Targets the CLR in Visual Studio

You can create Visual C++ programs that use .NET classes and compile them using the Visual Studio Development Environment.

For this procedure you can type in your own Visual C++ program or use one of the sample programs. The sample program referred to in this procedure creates a text file named textfile.txt and saves it to the project directory.

To create a new project in Visual Studio and add a new source file

1. Create a new project. On the **File** menu, point to **New**, then click **Project...**.
2. From the Visual C++ project types, click **CLR**, then click **CLR Empty Project**.
3. Enter a project name.

   By default, the solution that contains the project has the same name as the new project, though you can enter a different name. You can enter a different location for the project if you wish.

   Click **OK** to create the new project.
4. If Solution Explorer is not visible, click **Solution Explorer** on the **View** menu.
5. Add a new source file to the project:

   - Right-click on the **Source Files** folder in Solution Explorer and point to **Add** and click **New Item...**.
   - Click **C++ File (.cpp)** and enter a file name then click **Add**.

   The **.cpp** file appears in the **Source Files** folder in Solution Explorer and a tabbed window appears where you type the code you want in that file.
6. Click in the newly created tab in Visual Studio and type in a valid Visual C++ program or copy and paste one of the sample programs.

   For example, you can use the How to: Write a Text File sample program (in the **File Handling and I/O** node of the Programming Guide). See How to: Compile a Code Example from the Help Topics for information on copying a sample program to the Clipboard.

   If you use the sample program, notice that the new Visual C++ syntax uses the **gcnew** keyword instead of **new** when creating a .NET object, and that **gcnew** returns a handle (^) rather than a pointer (*):

   ```
   StreamWriter^ sw = gcnew StreamWriter(fileName);
   ```

   For more information on the new Visual C++ syntax, see New Language Features in Visual C++.
7. On the **Build** menu, click **Build Solution**.

   The **Output** window displays information about the compilation progress, such as the location of the build log and a message indicating that the build succeeded.

   If you make changes and run the program without doing a build, a dialog may appear indicating the project is out of date. Select the checkbox on this dialog before you click OK if you want Visual Studio to always use the current versions of files when building without prompting you.
8. On the **Debug** menu, click **Start without Debugging**.
9. If you used the sample program, a command window is displayed that indicates the text file has been created. Press any key to close the command window.

   The **textfile.txt** text file that was created is located in your project directory. You can open this file with Notepad.

   Note

Choosing the empty CLR project template automatically set the **/clr** compiler option. You can verify this by right-clicking on the project in **Solution Explorer** and clicking **Properties**, then checking the **Common Language Runtime support** option found within the **General** node of **Configuration Properties**.

See Also

Other Resources

Visual C++ Guided Tour
C++ Language Reference
Building a C/C++ Program

How to: Compile a Code Example from the Help Topics

The documentation for Visual Studio 2005 includes code examples that demonstrate the types of programs you can create. You can view the source code for these example programs from the Visual Studio Help menu.

These example programs are generally console applications, since these are concise complete programs. The code examples are ready for you to copy and paste into your own projects.

This procedure describes how to locate the Visual C++ code example programs in the help topics and how to compile and run them.

To copy example Visual C++ programs to the Clipboard

1. On the **Help** menu, click **Contents**.
2. Set the **Filtered by** list to **Visual C++**.
3. Navigate to one of the example programs:

   In the **Contents** area, open the tree nodes for **Development Tools and Languages**, **Visual Studio**, **Visual C++** and **Programming Guide**.

   If you are using Visual C++ Express Edition, open the nodes **Express Library for Visual Studio**, **Visual C++ Express Edition**, and **Programming Guide**.
4. There are many example programs in this area of the help.

   For example, open **.NET Programming in C++** and **Windows Operations** and select the How to: Retrieve Time Elapsed Since Startup topic.

   This opens the topic in a window in Visual Studio or in the Help Browser, depending on how your environment is configured.
5. Click the **Copy Code** button to place the example program on the Clipboard.
6. With the source code on the Clipboard, you can either:
   - Add a new **.cpp** file to the current project in Visual Studio and paste the source code into the file, then compile it in the Visual Studio Development Environment, as described in the How to: Compile a C++ Program that Targets the CLR in Visual Studio topic.
   - Or, paste the source code into a text editor and save it with a **.cpp** extension, then compile it on the command line, as described below.

Compiling the Code

Since the example programs are simple Visual C++ console applications, you may find it easier to compile them on the command line, as described in this procedure.

To compile a code example copied to the Clipboard

1. Open the Visual Studio Command Prompt window by clicking the **Start** button, point to **All Programs**, **Microsoft Visual Studio 2005**, **Visual Studio Tools**, then click **Visual Studio 2005 Command Prompt**.
2. At the command prompt, type the following and press Enter:

   notepad timeelapsed.cpp

   Click **Yes** when prompted to create a new file.
3. On the **Edit** menu, click **Paste** to place the program you copied into Notepad.
4. On the **File** menu, click **Save.**
5. On the **File** menu, click **Exit** to exit Notepad.
6. At the command line prompt, type the following and press Enter:

cl /clr timeelapsed.cpp

7. To run the **timeelapsed.exe** program, type **timeelapsed** and press Enter.

   The program runs and displays how much time has elapsed since Windows last started.

8. Type **exit** and press Enter to close the command prompt window.

See Also

Other Resources

Visual C++ Guided Tour
Building a C/C++ Program

How to: Create a Standard C++ Program

With Visual C++ 2005, you can create Standard C++ programs using the Visual Studio Development Environment. In this procedure, you create a new project, add a new file to the project, edit the file to add C++ code, and then compile and run the program using Visual Studio.

You can type in your own C++ program or use one of the sample programs. The sample program referred to in this procedure is a console application that uses the **set** container in the Standard Template Library (STL), which is part of the ISO C++ 98 standard.

Visual C++ conforms to these standards:

- ISO C 95
- ISO C++ 98
- Ecma C++/CLI 05

Note
You should use the **/Za** compiler option if you want to enforce ANSI C++ and ANSI C compliance checking of your program (the default option is **/Ze** which allows Microsoft extensions to the standard). See /Za, /Ze (Disable Language Extensions) for more information.

To create a new project and add a source file

1. Create a new project:

   On the **File** menu, point to **New**, then click **Project...**.

2. From the **Visual C++** project types, click **Win32**, then click **Win32 Console Application**.

3. Enter a project name.

   By default, the solution that contains the project has the same name as the new project, though you can enter a different name. You can enter a different location for the project if you wish.

   Click OK to create the new project.

4. In the **Win32 Application Wizard**, select **Empty Project** and click **Finish**.

5. If Solution Explorer is not visible, click **Solution Explorer** on the **View** menu.

6. Add a new source file to the project:

   - Right-click on the **Source Files** folder in Solution Explorer and point to **Add** and click **New Item**.
   - Click **C++ File (.cpp)** from the **Code** node, enter a file name, and then click **Add**.

   The **.cpp** file appears in the Source Files folder in Solution Explorer and a tabbed window appears where you type in the code.

7. Click in the newly created tab in Visual Studio and type in a valid C++ program that uses the Standard C++ Library, or copy and paste one of the sample programs.

   For example, you can use the set::find (STL Samples) sample program in the **Standard Template Library Samples** topics in the help. See How to: Compile a Code Example from the Help Topics for information on copying a sample program to the Clipboard.

   If you use the sample program for this procedure, notice the `using namespace std;` directive. This allows the program to use **cout** and **endl** without requiring fully qualified names (**std::cout** and **std::endl**).

8. On the **Build** menu, click **Build Solution**.

   The **Output** window displays information about the compilation progress, such as the location of the build log and a message indicating that the build succeeded.

9. On the **Debug** menu, click **Start without Debugging**.

   If you used the sample program, a command window is displayed that shows whether certain integers are found in the set.

See Also

Other Resources

Visual C++ Guided Tour
C++ Language Reference
Standard C++ Library Reference

How to: Create a Windows Forms Application

In .NET development, a Windows GUI application is called a Windows Forms (or *Winforms*) application. Developing a Windows Forms project with Visual C++ is generally the same as with any other .NET language, such as Visual Basic.NET or C#.

Windows Forms applications in Visual C++ use the .NET Framework classes and other .NET features with the new Visual C++ syntax. For more information, see New Language Features in Visual C++.

In this procedure, you create a Windows Forms application using several standard controls from the Toolbox. In the finished application, a user can select a date, and a text label shows what date was chosen.

To create a new Windows Forms project

1. On the **File** menu, click **New**, and then click **Project...**.
2. In the **Project Types** pane, select **CLR** in the **Visual C++** node, then select **Windows Forms Application** in the **Templates** pane.

   Type a name for the project, such as **winformsapp**. You can accept the default location, type in a location, or browse to a directory where you want to save the project.
3. The Windows Forms Designer opens, showing **Form1** of the project you created.

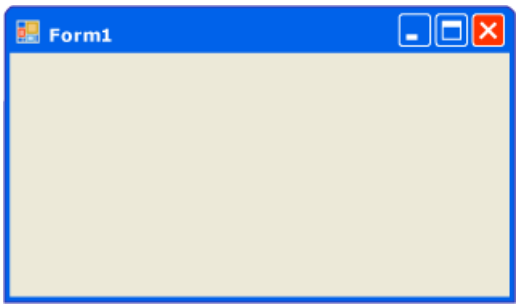

To add controls to a form

1. If the **Toolbox** window is not visible, click **Toolbox** on the **View** menu.
2. Place three controls from the **Toolbox** on the **Form1** design surface:
   - Drag a Label control to near the top-left corner of **Form1**.
   - Drag a DateTimePicker control just below the **Label** control.
   - Drag a Button control to the bottom of the form near the center.

   Your form should look something like this:

To set properties of forms and controls

1. Select the form by clicking on an empty area on its surface.
2. If the **Properties** window is not visible, click **Properties** on the **View** menu (or press F4).

   You may want to close the **Toolbox** for more room.
3. Set the form's **Text** property (shown in the form Title Bar) by clicking to the right of the **Text** property in the **Properties**

Window and typing:

Date Chooser

4. Select the label by clicking on it and set its **Text** property to:

   Choose a date:

5. Select the button by clicking on it and set its **Text** property to:

   OK

   The form should look similar to this:

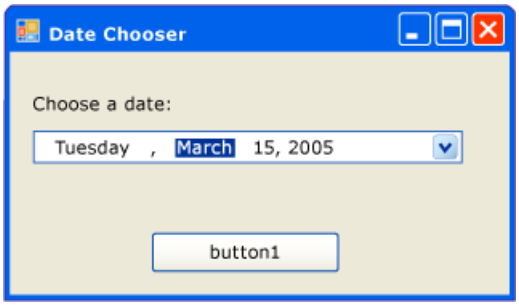

Writing Event Handler Code

In this section, you write the code to run when these events occur:

- A Click event on the **Button** control
- A ValueChanged event on the **DateTimePicker** control

To write code to handle events

1. Double-click on the button to add a button click event handler (the default event for a button is a Click event).
2. An empty event handler method is generated for you in the code view of the form displayed in a tabbed page in the editing area.
3. Press Enter after the opening brace of the **button1_Click** method and type the code to run when that event occurs:

   ```
   Application::Exit();
   ```

   IntelliSense™ displays a list of valid possible choices after you type the scope resolution operator (::). You can select a choice from the list and press Tab, double-click it, or continue typing.

4. Return to the Design view by clicking the **Form1.h [Design]** tab in the editing area or on the **View** menu, click **Designer**.
5. Select the **DateTimePicker** control by clicking on it.
6. To add a **ValueChanged** event handler for the **DateTimePicker** control, click the lightning bolt icon in the **Properties** window to display events for that control.
7. Double-click on the **ValueChanged** event to open generate an empty event handler in the Code view.

Note
ValueChanged is the default event for the **DateTimePicker** control, so you could also double-click the **DateTimePicker** control to generate an empty event handler.

8. Press Enter after the opening brace of the **dateTimePicker1_ValueChanged** method, then type the code to run when the event occurs:

   ```
   label1->Text=String::Format("New date: {0}", dateTimePicker1->Text);
   ```

   When a user of the application selects a new date, the Text property of the label is set to the literal string `New date:` with the Text property of the **DateTimePicker** appended to that string.

Visual Studio provides several features that simplify typing code:

- When you type an arrow operator (->), IntelliSense displays valid choices you can select from the list.
- When you type an opening parenthesis for a method, a tooltip window shows valid arguments for each overload of that method. To view the different overloads, use the UP or DOWN arrow keys.
- Auto-completion can finish typing a variable name or member from what you have typed. For example, if you type `String::Fo` and press Ctrl-Spacebar or Tab, Visual Studio will complete typing `String::Format` for you.

To build and run the program

1. On the **Build** menu, click **Build Solution**.

   If there are any errors, click the **Go to Next Message** button in the **Output** window. The error message text appears in the status bar. You can double-click on any error to go directly to the line with that error in the source code.

2. On the **Debug** menu, click **Run without Debugging**. The application you built is displayed.

3. Test the application by clicking the down arrow on the **DateTimePicker** and selecting a date. The label text changes to show the date that was selected.

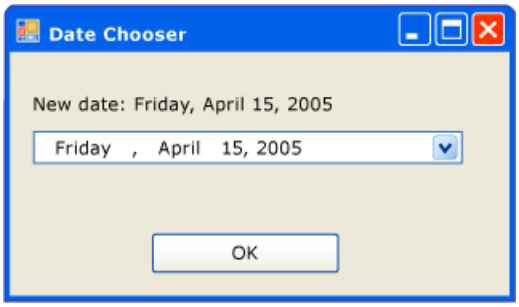

4. You can add more features to this application, such as menus, other forms, and help files. Experiment.

See Also

Reference

System.Windows.Forms

Concepts

Overview of Windows-based Applications

Other Resources

Visual C++ Guided Tour

Creating Windows-based Applications

How to: Convert Between Various String Types

This topic demonstrates how to convert various Visual C++ string types into other strings. The strings types that are covered include **char ***, **wchar_t***, _bstr_t, CComBSTR, CString, basic_string, and System.String. In all cases, a copy of the string is made when converted to the new type. Any changes made to the new string will not affect the original string, and vice versa.

Converting from char *

Example

This example demonstrates how to convert from a **char *** to the other string types listed above.

```
// convert_from_char.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"

using namespace std;
using namespace System;

int main()
{
    char *orig = "Hello, World!";
    cout << orig << " (char *)" << endl;

    // Convert to a wchar_t*
    size_t origsize = strlen(orig) + 1;
    const size_t newsize = 100;
    size_t convertedChars = 0;
    wchar_t wcstring[newsize];
    mbstowcs_s(&convertedChars, wcstring, origsize, orig, _TRUNCATE);
    wcscat_s(wcstring, L" (wchar_t *)");
    wcout << wcstring << endl;

    // Convert to a _bstr_t
    _bstr_t bstrt(orig);
    bstrt += " (_bstr_t)";
    cout << bstrt << endl;

    // Convert to a CComBSTR
    CComBSTR ccombstr(orig);
    if (ccombstr.Append(L" (CComBSTR)") == S_OK)
    {
        CW2A printstr(ccombstr);
        cout << printstr << endl;
    }

    // Convert to a CString
    CString cstring(orig);
    cstring += " (CString)";
    cout << cstring << endl;

    // Convert to a basic_string
    string basicstring(orig);
    basicstring += " (basic_string)";
    cout << basicstring << endl;

    // Convert to a System::String
    String ^systemstring = gcnew String(orig);
```

```
    systemstring += " (System::String)";
    Console::WriteLine("{0}", systemstring);
    delete systemstring;
}
```

Output

```
Hello, World! (char *)
Hello, World! (wchar_t *)
Hello, World! (_bstr_t)
Hello, World! (CComBSTR)
Hello, World! (CString)
Hello, World! (basic_string)
Hello, World! (System::String)
```

Converting from wchar_t *

Example

This example demonstrates how to convert from a **wchar_t *** to the other string types listed above.

```
// convert_from_wchar_t.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"

using namespace std;
using namespace System;

int main()
{
    wchar_t *orig = L"Hello, World!";
    wcout << orig << L" (wchar_t *)" << endl;

    // Convert to a char*
    size_t origsize = wcslen(orig) + 1;
    const size_t newsize = 100;
    size_t convertedChars = 0;
    char nstring[newsize];
    wcstombs_s(&convertedChars, nstring, origsize, orig, _TRUNCATE);
    strcat_s(nstring, " (char *)");
    cout << nstring << endl;

    // Convert to a _bstr_t
    _bstr_t bstrt(orig);
    bstrt += " (_bstr_t)";
    cout << bstrt << endl;

    // Convert to a CComBSTR
    CComBSTR ccombstr(orig);
    if (ccombstr.Append(L" (CComBSTR)") == S_OK)
    {
        CW2A printstr(ccombstr);
        cout << printstr << endl;
    }

    // Convert to a CString
    CString cstring(orig);
    cstring += " (CString)";
    cout << cstring << endl;

    // Convert to a basic_string
```

```
    wstring basicstring(orig);
    basicstring += L" (basic_string)";
    wcout << basicstring << endl;

    // Convert to a System::String
    String ^systemstring = gcnew String(orig);
    systemstring += " (System::String)";
    Console::WriteLine("{0}", systemstring);
    delete systemstring;
}
```

Output

```
Hello, World! (wchar_t *)
Hello, World! (char *)
Hello, World! (_bstr_t)
Hello, World! (CComBSTR)
Hello, World! (CString)
Hello, World! (basic_string)
Hello, World! (System::String)
```

Converting from _bstr_t

Example

This example demonstrates how to convert from a **_bstr_t** to the other string types listed above.

```
// convert_from_bstr_t.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"

using namespace std;
using namespace System;

int main()
{
    _bstr_t orig("Hello, World!");
    wcout << orig << " (_bstr_t)" << endl;

    // Convert to a char*
    const size_t newsize = 100;
    char nstring[newsize];
    strcpy_s(nstring, (char *)orig);
    strcat_s(nstring, " (char *)");
    cout << nstring << endl;

    // Convert to a wchar_t*
    wchar_t wcstring[newsize];
    wcscpy_s(wcstring, (wchar_t *)orig);
    wcscat_s(wcstring, L" (wchar_t *)");
    wcout << wcstring << endl;

    // Convert to a CComBSTR
    CComBSTR ccombstr((char *)orig);
    if (ccombstr.Append(L" (CComBSTR)") == S_OK)
    {
        CW2A printstr(ccombstr);
        cout << printstr << endl;
    }

    // Convert to a CString
```

```
    CString cstring((char *)orig);
    cstring += " (CString)";
    cout << cstring << endl;

    // Convert to a basic_string
    string basicstring((char *)orig);
    basicstring += " (basic_string)";
    cout << basicstring << endl;

    // Convert to a System::String
    String ^systemstring = gcnew String((char *)orig);
    systemstring += " (System::String)";
    Console::WriteLine("{0}", systemstring);
    delete systemstring;
}
```

Output

```
Hello, World! (_bstr_t)
Hello, World! (char *)
Hello, World! (wchar_t *)
Hello, World! (CComBSTR)
Hello, World! (CString)
Hello, World! (basic_string)
Hello, World! (System::String)
```

Converting from CComBSTR

Example

This example demonstrates how to convert from a **CComBSTR** to the other string types listed above.

```
// convert_from_ccombstr.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"
#include "vcclr.h"

using namespace std;
using namespace System;
using namespace System::Runtime::InteropServices;

int main()
{
    CComBSTR orig("Hello, World!");
    CW2A printstr(orig);
    cout << printstr << " (CComBSTR)" << endl;

    // Convert to a char*
    const size_t newsize = 100;
    char nstring[newsize];
    CW2A tmpstr1(orig);
    strcpy_s(nstring, tmpstr1);
    strcat_s(nstring, " (char *)");
    cout << nstring << endl;

    // Convert to a wchar_t*
    wchar_t wcstring[newsize];
    wcscpy_s(wcstring, orig);
    wcscat_s(wcstring, L" (wchar_t *)");
    wcout << wcstring << endl;
```

```
    // Convert to a _bstr_t
    _bstr_t bstrt(orig);
    bstrt += " (_bstr_t)";
    cout << bstrt << endl;

    // Convert to a CString
    CString cstring(orig);
    cstring += " (CString)";
    cout << cstring << endl;

    // Convert to a basic_string
    wstring basicstring(orig);
    basicstring += L" (basic_string)";
    wcout << basicstring << endl;

    // Convert to a System::String
    String ^systemstring = gcnew String(orig);
    systemstring += " (System::String)";
    Console::WriteLine("{0}", systemstring);
    delete systemstring;
}
```

Output

```
Hello, World! (CComBSTR)
Hello, World! (char *)
Hello, World! (wchar_t *)
Hello, World! (_bstr_t)
Hello, World! (CString)
Hello, World! (basic_string)
Hello, World! (System::String)
```

Converting from CString

Example

This example demonstrates how to convert from a **CString** to the other string types listed above.

```
// convert_from_cstring.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"

using namespace std;
using namespace System;

int main()
{
    CString orig("Hello, World!");
    wcout << orig << " (CString)" << endl;

    // Convert to a char*
    const size_t newsize = 100;
    char nstring[newsize];
    strcpy_s(nstring, orig);
    strcat_s(nstring, " (char *)");
    cout << nstring << endl;

    // Convert to a wchar_t*
    // You must first convert to a char * for this to work.
    size_t origsize = strlen(orig) + 1;
    size_t convertedChars = 0;
```

```
    wchar_t wcstring[newsize];
    mbstowcs_s(&convertedChars, wcstring, origsize, orig, _TRUNCATE);
    wcscat_s(wcstring, L" (wchar_t *)");
    wcout << wcstring << endl;

    // Convert to a _bstr_t
    _bstr_t bstrt(orig);
    bstrt += " (_bstr_t)";
    cout << bstrt << endl;

    // Convert to a CComBSTR
    CComBSTR ccombstr(orig);
    if (ccombstr.Append(L" (CComBSTR)") == S_OK)
    {
        CW2A printstr(ccombstr);
        cout << printstr << endl;
    }

    // Convert to a basic_string
    string basicstring(orig);
    basicstring += " (basic_string)";
    cout << basicstring << endl;

    // Convert to a System::String
    String ^systemstring = gcnew String(orig);
    systemstring += " (System::String)";
    Console::WriteLine("{0}", systemstring);
    delete systemstring;
}
```

Output

```
Hello, World! (CString)
Hello, World! (char *)
Hello, World! (wchar_t *)
Hello, World! (_bstr_t)
Hello, World! (CComBSTR)
Hello, World! (basic_string)
Hello, World! (System::String)
```

Converting from basic_string

Example

This example demonstrates how to convert from a **basic_string** to the other string types listed above.

```
// convert_from_basic_string.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"

using namespace std;
using namespace System;

int main()
{
    string orig("Hello, World!");
    cout << orig << " (basic_string)" << endl;

    // Convert to a char*
    const size_t newsize = 100;
    char nstring[newsize];
```

```
    strcpy_s(nstring, orig.c_str());
    strcat_s(nstring, " (char *)");
    cout << nstring << endl;

    // Convert to a wchar_t*
    // You must first convert to a char * for this to work.
    size_t origsize = strlen(orig.c_str()) + 1;
    size_t convertedChars = 0;
    wchar_t wcstring[newsize];
    mbstowcs_s(&convertedChars, wcstring, origsize, orig.c_str(), _TRUNCATE);
    wcscat_s(wcstring, L" (wchar_t *)");
    wcout << wcstring << endl;

    // Convert to a _bstr_t
    _bstr_t bstrt(orig.c_str());
    bstrt += " (_bstr_t)";
    cout << bstrt << endl;

    // Convert to a CComBSTR
    CComBSTR ccombstr(orig.c_str());
    if (ccombstr.Append(L" (CComBSTR)") == S_OK)
    {
        CW2A printstr(ccombstr);
        cout << printstr << endl;
    }

    // Convert to a CString
    CString cstring(orig.c_str());
    cstring += " (CString)";
    cout << cstring << endl;

    // Convert to a System::String
    String ^systemstring = gcnew String(orig.c_str());
    systemstring += " (System::String)";
    Console::WriteLine("{0}", systemstring);
    delete systemstring;
}
```

Output

```
Hello, World! (basic_string)
Hello, World! (char *)
Hello, World! (wchar_t *)
Hello, World! (_bstr_t)
Hello, World! (CComBSTR)
Hello, World! (CString)
Hello, World! (System::String)
```

Converting from System::String

Example

This example demonstrates how to convert from a **System.String** to the other string types listed above.

```
// convert_from_system_string.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"
#include "vcclr.h"

using namespace std;
using namespace System;
```

```
using namespace System::Runtime::InteropServices;

int main()
{
    String ^orig = gcnew String("Hello, World!");
    Console::WriteLine("{0} (System::String)", orig);

    pin_ptr<const wchar_t> wch = PtrToStringChars(orig);

    // Convert to a char*
    size_t origsize = wcslen(wch) + 1;
    const size_t newsize = 100;
    size_t convertedChars = 0;
    char nstring[newsize];
    wcstombs_s(&convertedChars, nstring, origsize, wch, _TRUNCATE);
    strcat_s(nstring, " (char *)");
    cout << nstring << endl;

    // Convert to a wchar_t*
    wchar_t wcstring[newsize];
    wcscpy_s(wcstring, wch);
    wcscat_s(wcstring, L" (wchar_t *)");
    wcout << wcstring << endl;

    // Convert to a _bstr_t
    _bstr_t bstrt(wch);
    bstrt += " (_bstr_t)";
    cout << bstrt << endl;

    // Convert to a CComBSTR
    CComBSTR ccombstr(wch);
    if (ccombstr.Append(L" (CComBSTR)") == S_OK)
    {
        CW2A printstr(ccombstr);
        cout << printstr << endl;
    }

    // Convert to a CString
    CString cstring(wch);
    cstring += " (CString)";
    cout << cstring << endl;

    // Convert to a basic_string
    wstring basicstring(wch);
    basicstring += L" (basic_string)";
    wcout << basicstring << endl;

    delete orig;
}
```

Output

```
Hello, World! (System::String)
Hello, World! (char *)
Hello, World! (wchar_t *)
Hello, World! (_bstr_t)
Hello, World! (CComBSTR)
Hello, World! (CString)
Hello, World! (basic_string)
```

See Also

Tasks

How to: Convert Standard String to System::String

How to: Convert System::String to Standard String

How to: Convert System::String to wchar_t* or char*

Reference

mbstowcs_s, _mbstowcs_s_l

wcstombs_s, _wcstombs_s_l

strcpy_s, wcscpy_s, _mbscpy_s
strcat_s, wcscat_s, _mbscat_s
pin_ptr

Concepts

ATL and MFC String Conversion Macros
CString Operations Relating to C-Style Strings
Programming with CComBSTR

Other Resources

Visual C++ Guided Tour

How to: Use Visual C++ Express Edition with the Microsoft Platform SDK

You can use Visual C++ 2005 Express Edition to build powerful .NET Framework applications immediately after installation. In order to use Visual C++ Express Edition to build Win32 applications, you will need to take just a few more steps, as detailed below.

Installing the Platform SDK for use with Visual C++ Express Edition

1. Install the Microsoft Platform SDK over the Web from the Platform SDK Update site. At a minimum, you will need to install the Windows Core SDK. However, several samples utilize the functionality in other SDKs, so it is recommended that you perform a complete install.

2. From the **Tools** menu in Visual Studio, select **Options**. The **Options** dialog box appears.

   From the **Options** dialog box, expand the **Projects and Solutions** node and select **VC++ Directories**. In that section, add the following paths to the appropriate subsection:

   - Executable files: `C:\Program Files\Microsoft SDK\Bin`
   - Include files: `C:\Program Files\Microsoft SDK\include`
   - Library files: `C:\Program Files\Microsoft SDK\lib`

Note
The location of the Platform SDK might be different on your system.

3. Update the corewin_express.vsprops file (found in %VSINSTALLDIR%\VC\VCProjectDefaults) and change the string that reads:

   AdditionalDependencies="kernel32.lib"

   to:

   AdditionalDependencies="kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib"

4. Update the default.js file (found in %VSINSTALLDIR%\VC\VCWizards\AppWiz\Generic\Application\scripts\1033) and change the two lines that read:

   LinkTool.AdditionalDependencies = "kernel32.lib $(NoInherit)";

   to:

   // LinkTool.AdditionalDependencies = "kernel32.lib $(NoInherit)";

5. To integrate the Platform SDK documentation into the Visual C++ Express documentation, follow the instructions at Visual Studio 2005 Express Edition Combined Help Collection Manager.

6. Generate and build a Win32 application to test your paths.

   For this, you can use one of the many samples provided in the Help files, such as CPUID Sample: Determines CPU Capabilities. To verify that everything is working, build and run the sample as documented in the help topic.

See Also

Other Resources

Visual C++ Guided Tour

Walkthrough: Using the Visual Studio IDE

The Visual Studio Integrated Development Environment (IDE) offers a set of tools that help you write and modify the code for your programs, and also detect and correct errors in your programs.

In this topic, you create a new standard C++ program and test its functionality by using features available in Visual Studio for the C++ developer.

This walkthrough covers the following:

- Working with Projects and Solutions
- Using Solution Explorer
- Adding a Source File
- Fixing Compilation Errors
- Testing a Program
- Debugging a Program

Prerequisites

This topic assumes you understand the fundamentals of the C++ language.

Working with Projects and Solutions

Visual Studio organizes your work in projects and solutions. A solution can contain more than one project, such as a DLL and an executable that references that DLL. For more information, see Introduction to Solutions, Projects, and Items.

The first step in writing a Visual C++ program with Visual Studio is to choose the type of project. For each project type, Visual Studio sets compiler settings and generates starter code for you.

To create a new project

1. On the **File** menu, point to **New**, and then click **Project...**.
2. In the **Project Types** area, click **Win32**,and then, in the **Visual Studio installed templates** pane, click **Win32 Console Application**.
3. Type **game** as the project name.

   When you create a new project, Visual Studio puts the project in a solution. Accept the default name for the solution, which by default is the same name as the project.

   You can accept the default location, type a different location, or browse to a directory where you want to save the project.

   Press **OK** to start the **Win32 Application Wizard**.
4. On the **Overview** page of the **Win32 Application Wizard** dialog box, press **Next**.
5. On the **Application Settings** page under **Application type**, select **Console Application**. Select the **Empty Project** setting under **Additional options** and click **Finish**.

   You now have a project without source code files.

Using Solution Explorer

Solution Explorer makes it easy for you to work with files and other resources in your solution.

In this step, you add a class to the project and Visual Studio adds the **.h** and **.cpp** files to your project. You then add a new source code file to the project for the main program that tests the class.

To add a class to a project

1. If the **Solution Explorer** window is not visible, on the **View** menu click **Solution Explorer**.
2. Right-click the **Header Files** folder in **Solution Explorer** and point to **Add**. Then click **Class**.

In the **Visual C++** category, click **C++** and click **C++ Class** in the **Visual Studio installed templates** area. Click **Add**.

3. In the **Generic C++ Class Wizard**, type **Cardgame** as the **Class name** and accept the default file names and settings. Then click **Finish**.

4. Make these changes to the **Cardgame.h** file displayed in the editing area:

   - Add two private data members after the opening brace of the class definition:

     ```
     int players;
     static int totalparticipants;
     ```

   - Add a public constructor prototype that takes one parameter of type `int`:

     ```
     Cardgame(int p);
     ```

   - Delete the default constructor generated for you. A default constructor is a constructor that takes no arguments. The default constructor looks similar to the following:

     ```
     Cardgame(void);
     ```

5. The **Cardgame.h** file should resemble this after your changes:

   ```
   #pragma once
   class Cardgame
   {
           int players;
           static int totalparticipants;
       public:
           Cardgame(int p);
           ~Cardgame(void);
   };
   ```

   The line `#pragma once` indicates that the file will be included only one time by the compiler. For more information, see once.

   For information about other C++ keywords included in this header file, see class (C++), int, Static (C++), and public (C++).

6. Double-click **Cardgame.cpp** in the **Source Files** folder to open it for editing.

7. Add the code for the constructor that takes one **int** argument:

   ```
   Cardgame::Cardgame(int p)
   {
       players = p;
       totalparticipants += p;
   }
   ```

   When you begin typing `pl` or `to`, you can press Ctrl-Spacebar and auto-completion will finish typing `players` or `totalparticipants` for you.

8. Delete the default constructor that was generated for you:

   ```
   Cardgame::Cardgame(void);
   ```

9. The **Cardgame.cpp** file should resemble this after your changes:

```
#include "Cardgame.h"
Cardgame::Cardgame(int p)
{
    players = p;
    totalparticipants += p;
}
Cardgame::~Cardgame(void)
{
}
```

For an explanation of `#include`, see The #include Directive.

Adding a Source File

In this step, you add a source code file for the main program that tests the class.

To add a new source file

1. From the **Project** menu, click **Add New Item**.

   Alternatively, to use Solution Explorer to add a new file to the project, right-click the **Source Files** folder in Solution Explorer and point to **Add**. Then click **New Item**.

   In the **Visual C++** area, select **Code**. Then click **C++ File (.cpp)**.

2. Type **testgames** as the Name and click **Add**.

3. In the **testgames.cpp** editing window, type the following code:

```
#include "Cardgame.h"
int Cardgame::totalparticipants = 0;
int main()
{
    Cardgame *bridge = 0;
    Cardgame *blackjack = 0;
    Cardgame *solitaire = 0;
    Cardgame *poker = 0;

    bridge = new Cardgame(4);
    blackjack = new Cardgame(8);
    solitaire = new Cardgame(1);
    delete blackjack;
    delete bridge;
    poker = new Cardgame(5);
    delete solitaire;
    delete poker;

    return 0;
}
```

   For information about C++ keywords included in this source file, see new Operator (C++), delete Operator (C++), The if-else Statement, and The try, catch, and throw Statements.

4. On the **Build** menu, click **Build Solution**.

   You should see output from the build in the **Output** window indicating that the project compiled without errors.

Fixing Compilation Errors

In this step, you deliberately introduce a Visual C++ syntax error in your code to see what a compilation error looks like and how to fix it. When you compile the project, an error message indicates what the problem is and where it occurred.

To fix compilation errors using the IDE

1. In **testgames.cpp**, delete the semicolon in the last line so that it resembles this:

   ```
   return 0
   ```

2. On the **Build** menu, click **Build Solution**.

3. A message in the **Output** window indicates that building the project failed.

   Click on the **Go To Next Message** button (the green, right-pointing arrow) in the **Output** window. The error message in the **Output** window and status bar area indicates there is a missing semicolon before the closing brace.

   You can press the F1 key to view more help information about an error.

4. Add the semicolon back to the end of the line with the syntax error:

   ```
   return 0;
   ```

5. On the **Build** menu, click **Build Solution**.

   A message in the **Output** window indicates that the project compiled correctly.

Testing a Program

Running a program in Debug mode enables you to use breakpoints to pause the program to examine the state of variables and objects.

In this step, you watch the value of a variable as the program runs and deduce why the value is not what you might expect.

To run a program in Debug mode

1. Click on the **testgames.cpp** tab in the editing area if that file is not visible.

2. Set the current line in the editor by clicking the following line:

   ```
   solitaire = new Cardgame(1);
   ```

3. To set a breakpoint on that line, on the **Debug** menu, click **Toggle Breakpoint**, or press F9. Alternatively, you can click in the area to the left of a line of code to set or clear a breakpoint.

   A red circle appears to the left of a line with a breakpoint set.

4. On the **Debug** menu, click **Start Debugging** (or press F5).

   When the program reaches the line with the breakpoint, execution stops temporarily (because your program is in Break mode). A yellow arrow to the left of a line of code indicates that is the next line to be executed.

5. To examine the value of the `totalparticipants` variable, hover over it with the mouse. The variable name and its value of 12 is displayed in a tooltip window.

   Right-click the `totalparticipants` variable and click **Add Watch** to display that variable in the **Watch** window. You can also select the variable and drag it to the **Watch** window.

6. On the **Debug** menu, click **Step Over** or press F10 to step to the next line of code.

   The value of `totalparticipants` is now displayed as 13.

7. Right-click the last line of the `main` method (`return 0;`) and click **Run to Cursor**. The yellow arrow to the left of the code points to the next statement to be executed.

8. The `totalparticipants` number should decrease when a Cardgame terminates. At this point, `totalparticipants` should equal 0 because all Cardgame pointers have been deleted, but the **Watch 1** window indicates `totalparticipants` equals 18.

   There is a bug in the code that you will detect and fix in the next section.

9. On the **Debug** menu, click **Stop Debugging** or press Shift-F5 to stop the program.

Debugging a Program

In this step, you modify the program to fix the problem that was discovered above.

To fix a program that has a bug

1. To see what occurs when a Cardgame object is destroyed, view the destructor for the Cardgame class.
2. On the **View** menu, click **Class View** or click the **Class View** tab in the **Solution Explorer** window.
3. Expand the **game** project tree and click the **Cardgame** class.

   The area underneath shows the class members and methods.
4. Right-click the **~Cardgame(void)** destructor and click **Go To Definition**.
5. To decrease the `totalparticipants` when a card game terminates, type the following code between the opening and closing braces of the `Cardgame::~Cardgame` destructor:

   ```
   totalparticipants -= players;
   ```

6. The **Cardgame.cpp** file should resemble this after your changes:

   ```
   #include "Cardgame.h"

   Cardgame::Cardgame(int p)
   {
       players = p;
       totalparticipants += p;
   }
   Cardgame::~Cardgame(void)
   {
       totalparticipants -= players;
   }
   ```

7. On the **Build** menu, click **Build Solution**.
8. On the **Debug** menu, click **Run** or press F5 to run the program in Debug mode. The program pauses at the first breakpoint.
9. On the **Debug** menu, click **Step Over** or press F10 to step through the program.

   Note that after each Cardgame constructor executes, the value of `totalparticipants` increases and after each pointer is deleted (and the destructor is called), `totalparticipants` decreases.

   If you step to the last line of the program, just before the **return** statement is executed, `totalparticipants` equals 0.
10. Continue stepping through the program until it exits or on the **Debug** menu, click **Run** or press F5 to allow the program to continue to run until it exits.

See Also

Other Resources

Visual C++ Guided Tour
Integrated Development Environment for Visual Studio
Managing Solutions, Projects, and Files
Building, Debugging, and Testing

Walkthrough: Creating a Windows Forms Control

Windows Forms controls are components that can be placed in Windows Forms applications (GUI applications that target the Common Language Runtime). Windows Forms applications in Visual C++ use .NET Framework classes and other .NET features with the new Visual C++ syntax.

In this procedure, you create a Windows Forms control that displays a number which increments each time the label is clicked in an application. You will also create a Windows Forms application project to test the control.

This walkthrough covers the following:

- Creating a New Project
- Designing the Control
- Adding a Custom Property to the Control
- Adding a Project to Test the Control
- Placing the Control in an Application
- Running the Application

Create a New Project

The Windows Forms Control project template you use in this section creates a User Control, which is a composite control that contains other controls.

Alternatively, you can create a Windows Forms control by deriving a class directly from the Control class (where your code is responsible for drawing the control) or the Component class (a control with no UI).

To create a new Windows Forms control project

1. On the **File** menu, click **New**, and then click **Project...**.
2. In the **Project Types** pane, select **CLR** in the **Visual C++** node, then select **Windows Forms Control Library** in the **Visual Studio installed templates** pane.

   Type a name for the project, such as **clickcounter**.

   Type a different name for the solution, such as **controlandtestapp**.

   You can accept the default location, type in a location or browse to a directory where you want to save the project.
3. The Windows Forms Designer opens, showing an area where you add the controls you want to place on the control design surface.

Design the Control

In this step, you add a Label control to the control design surface. You then set some properties on the control itself and on the **Label** control it contains.

To set the properties of a User Control

1. If the **Properties** window is not visible, on the **View** menu, click **Properties Window**.

   Click on the control to select it and set its properties as follows:

   - Set the **Size** property to 100, 100.
   - Set the **BorderStyle** to **Fixed3D**

     The label boundaries will be visible when the control placed in an application.
2. If the **Toolbox** window is not visible, click **Toolbox** on the **View** menu.

   Drag a **Label** control from the **Toolbox** to the design surface and place it near the middle of the control.

   Set these properties for the label:

- Set the **BorderStyle** to **FixedSingle**.
- Set the **Text** to the digit **0** (zero).
- Set the **Autosize** to **False**.
- Set the **Size** to **30, 20**.
- Set the **TextAlign** to **MiddleCenter**.

Leave the **Name** property (how you refer to it in code) unchanged as **label1**. The control should look like this:

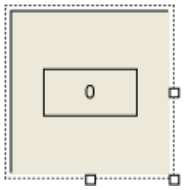

3. Add an event handler for the label Click event (the default event for a label) by double-clicking on the label.
4. The **clickcounter.h** file is displayed in the editing area with an empty event handler method generated for you.

Note
Close the **Toolbox** or **Properties** window if you need more room by clicking on their Close boxes or unpinning them so they auto-hide.

5. Press Enter after the opening brace of the **label1_Click** method and type:

```
int temp = System::Int32::Parse(label1->Text);
temp++;
label1->Text = temp.ToString();
```

IntelliSense™ displays a list of valid choices after you type a scope resolution operator (::), dot operator (.) or arrow operator (->). You can highlight an item and press Tab or Enter or double-click an item to insert that item into your code.

Also, when you type an opening parenthesis for a method, Visual Studio displays valid argument types for each overload of the method.

Add a Custom Property to the Control

In this step, you define a custom property that allows an application developer to determine whether the number displayed on the control increments when the label is clicked or when any location on the control is clicked.

To add a custom property to a control

1. Place the cursor after the colon of the first **public** scope indicator at the top of the **clickcounterControl.h** file, press Enter, then type the following:

```
property bool ClickAnywhere {
    bool get() {
        return (label1->Dock == DockStyle::Fill);
    }
    void set(bool val) {
        if (val)
            label1->Dock = DockStyle::Fill;
        else
            label1->Dock = DockStyle::None;
    }
}
```

When the **ClickAnywhere** property of the control is set to **true**, the Dock property of the label is set to **DockStyle::Fill**, so the label fills the entire control surface. A click anywhere on the control surface will then cause a label **Click** event,

incrementing the number on the label.

When the **ClickAnywhere** property is **false** (the default), the **Dock** property of the label is set to **DockStyle::None**. The label does not fill the control, and a click on the control must be inside the label boundaries to cause a label **Click** event, incrementing the number.

2. Build the User Control. On the **Build** menu, select **Build Solution**.

   If there are no errors, a Windows Forms control is generated with a file name of **clickcounter.dll**. You can locate this file in your project directory structure.

Add a Project to Test the Control

In this step, you create a Windows Forms application project where you will place instances of the **clickcounter** control on a form.

Note
The Windows Forms application you create to test the control can be written with Visual C++ or another .NET language, such as C# or Visual Basic .NET.

To create a Windows Forms application project

1. On the **File** menu, click **New**, and then click **Project...**.

   You can also add a project to the solution by right-clicking on the **controlandtestapp** solution in **Solution Explorer**, pointing to **Add,** then clicking **New Project...**.

2. In the **Project Types** pane, select **CLR** in the **Visual C++** node, then select **Windows Forms Application** in the **Visual Studio installed templates** pane.

   Type a name for the project, such as **testapp**.

   Be sure to select **Add to Solution** instead of accepting the default **Create New Solution** setting in the **Solution** drop-down list, then click **OK**.

3. The Windows Forms Designer for the new project opens, showing a new form called **Form1**.

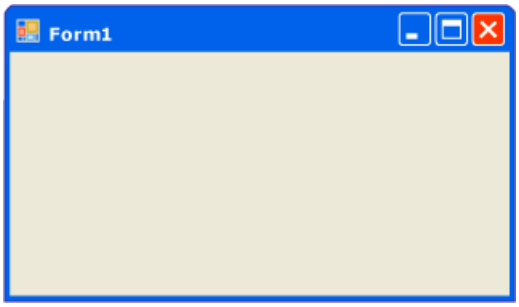

To add a control to the Toolbox

1. Add a reference to the control. On the **Project** menu, click **References** or right-click on the **testapp** project in **Solution Explorer** and click **References**.

   Click the **Add New Reference** button, then click the **Projects** tab (you are adding a reference to another project in this solution) and select the **clickcounter** project. Click **OK** twice.

2. If the **Toolbox** window is not visible, click **Toolbox** on the **View** menu.

3. Right-click on the **Toolbox** and click **Choose Items**.

   Click the **Browse** button and locate the **clickcounter.dll file in your solution directory structure. Select it and click Open.**

   The **clickcounter** control is shown in the**.NET Framework Components** list with a check mark. Click OK.

   The control appears in the **Toolbox** with the default "gear" icon.

Place the Control in an Application

In this step, you place two instances of the control on an application form and set their properties.

To place instances of a control on a form

1. Drag two instances of the **clickcounter** control from the **Toolbox**. Place them on the form so they don't overlap.

   If you need to make the form wider, click on the form to select it and drag one of the selection handles outward.

2. If the **Properties** window is not visible, click **Properties** on the **View** menu.

   The **ClickAnywhere** property is in the **Misc.** section of the **Property Window** if properties are organized by category.

3. Click one instance of the **clickcounter** control on the form to select it, then set its **ClickAnywhere** property to **true**.
4. Leave the **ClickAnywhere** property of the other instance of the **clickcounter** control set to **false** (the default).
5. Right-click on the **testapp** project in Solution Explorer and select **Set As StartUp Project**.
6. On the **Build** menu, click **Rebuild Solution**.

   You should see that the two projects built with no errors.

Run the Application

In this step, you run the application, and click on the controls to test them.

To place instances of a control on a form

1. On the **Debug** menu, click **Start Debugging**.

   The form appears with the two instances of the control visible.

2. Run the application and click on both **clickcounter** controls:

   - Click on the control with **ClickAnywhere** set to **true**.

     The number on the label increments when you click anywhere on the control.

   - Click on the control with **ClickAnywhere** set to **false**.

     The number on the label increments only when you click within the visible boundary of the label.

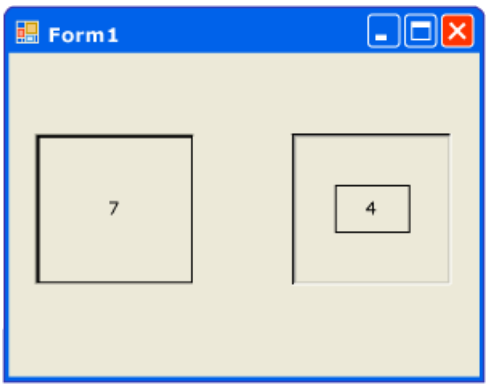

1. Close the test application by clicking its Close box in the upper right corner of the Form1 window.

See Also

Reference

System.Windows.Forms

Other Resources

Visual C++ Guided Tour

Creating Windows-based Applications

Windows Forms Controls

Walkthrough: Creating and Using a Managed Assembly

In this walkthrough, you will create a managed assembly containing useful routines that can be used by other applications. Using managed assemblies is a great way to reuse code. Rather than re-implementing these routines in every program you create, you write them once and reference them from applications that need the functionality.

This walkthrough uses Visual C++ and targets the Common Language Runtime. For a walkthrough using native C++ code, see Walkthrough: Creating and Using a Dynamic Link Library or Walkthrough: Creating and Using a Static Library.

This walkthrough covers the following:

- Creating a new class library project
- Adding a class to the class library
- Creating an application that references the class library
- Using the functionality from the class library in the console application
- Running the application

Prerequisites

This topic assumes you understand the fundamentals of the C++ language.

To create a new class library project

1. From the **File** menu, select **New** and then select **Project...**.
2. From the **Project types** pane, under **Visual C++**, select **CLR**. This creates a project that targets the Common Language Runtime.
3. From the **Templates** pane, select **Class Library**.
4. Choose a name for the project, such as **MathFuncsAssembly**, and enter it in the **Name** field. Choose a name for the solution, such as **ManagedAssemblies**, and enter it in the **Solution Name** field.
5. Press **OK** to create the project.
6. By default, when new projects are created, they are set up to use precompiled headers. To disable precompiled header, select **Properties** from the **Project** menu. Expand the **Configuration Properties** node, then expand the **C/C++** node, and select **Precompiled Headers**. From the dropdown list next to **Create/Use Precompiled Header**, select **Not Using Precompiled Header**. Press **OK** to save these changes. For more information on precompiled headers, see Creating Precompiled Header Files.

To add a class to the class library

1. After you create a new CLR Class Library, a simple class is created for you. The names for the header file and source file will be the same as the name you chose for the project above. In this example, they are named **MathFuncsAssembly.h** and **MathFuncsAssembly.cpp**.
2. Replace the existing code in **MathFuncsAssembly.h** with a simple class named **MyMathFuncsAssembly** to do common mathematical operations, such as addition, subtraction, multiplication, and division. The code should resemble the following:

```
// MathFuncsAssembly.h

using namespace System;

namespace MathFuncs
{
    public ref class MyMathFuncs
    {
    public:
```

```
        // Returns a + b
        static double Add(double a, double b);

        // Returns a - b
        static double Subtract(double a, double b);

        // Returns a * b
        static double Multiply(double a, double b);

        // Returns a / b
        // Throws DivideByZeroException if b is 0
        static double Divide(double a, double b);
    };
}
```

3. Implement the functionality for **MyMathFuncs** in the source file. The code should resemble the following:

```
// MathFuncsAssembly.cpp
// compile with: /clr /LD

#include "MathFuncsAssembly.h"

namespace MathFuncs
{
    double MyMathFuncs::Add(double a, double b)
    {
        return a + b;
    }

    double MyMathFuncs::Subtract(double a, double b)
    {
        return a - b;
    }

    double MyMathFuncs::Multiply(double a, double b)
    {
        return a * b;
    }

    double MyMathFuncs::Divide(double a, double b)
    {
        if (b == 0)
        {
            throw gcnew DivideByZeroException("b cannot be zero!");
        }

        return a / b;
    }
}
```

4. Compile the class library by selecting **Build Solution** from the **Build** menu. This creates a dynamic link library (DLL) that can be used by other programs. For more information on DLLs, see DLLs.

To create an application that references the class library

1. To create an application that will reference and use the class library that was just created, from the **File** menu, select **New** and then select **Project...**.
2. From the **Project types** pane, under **Visual C++**, select **CLR**. This creates a project that targets the Common Language

Runtime.

3. From the **Templates** pane, select **CLR Console Application**.

4. Choose a name for the project, such as **MyExecRefsAssembly**, and enter it in the **Name** field. Next to **Solution**, select **Add to Solution** from the drop down list. This will add the new project to the same solution as the class library.

5. Press **OK** to create the project.

6. By default, when new projects are created, they are set up to use precompiled headers. To disable precompiled header, select **Properties** from the **Project** menu. Expand the **Configuration Properties** node, then expand the **C/C++** node, and select **Precompiled Headers**. From the dropdown list next to **Create/Use Precompiled Header**, select **Not Using Precompiled Header**. Press **OK** to save these changes. For more information on precompiled headers, see Creating Precompiled Header Files.

To use the functionality from the class library in the console application

1. After you create a new CLR Console Application, a program is created for you that simply writes "Hello World" to the console. The name for the source file will be the same as the name you chose for the project above. In this example, it is named **MyExecRefsAssembly.cpp**.

2. To use the math routines that were created in the class library, you must reference it. To do this, select **References...** from the **Project** menu. From the **Property Pages** dialog, expand the **Common Properties** node and select **References**. Then select the **Add New Reference...** button. For more information on the **References...** dialog, see References, Common Properties, <Projectname> Property Pages Dialog Box.

3. The **Add Reference** dialog is displayed. This dialog lists all the libraries that you can reference. The **.NET** tab lists the libraries that are included with the .NET Framework. The **COM** tab lists all the COM components on your computer. The **Project** tab lists all the projects in the current solution and any libraries they contain. From the **Projects** tab, select **MathFuncsAssembly**. Then select **OK**. For more information on the **Add Reference** dialog, see Add Reference Dialog Box.

   Note

   You can also reference an assembly directly from the source file by including the **#using** directive, as in `#using <MathFuncsAssembly.dll>`. For more information on this directive, see The #using Directive.

4. You can now use the **MyMathFuncs** class in this application. In **MyExecRefsAssembly.cpp**, replace the contents of the file function with the following code:

```
// MyExecRefsAssembly.cpp
// compile with: /clr /FUMathFuncsAssembly.dll

using namespace System;

int main(array<System::String ^> ^args)
{
    double a = 7.4;
    int b = 99;

    Console::WriteLine("a + b = {0}",
        MathFuncs::MyMathFuncs::Add(a, b));
    Console::WriteLine("a - b = {0}",
        MathFuncs::MyMathFuncs::Subtract(a, b));
    Console::WriteLine("a * b = {0}",
        MathFuncs::MyMathFuncs::Multiply(a, b));
    Console::WriteLine("a / b = {0}",
        MathFuncs::MyMathFuncs::Divide(a, b));

    return 0;
}
```

5. Build the executable by selecting **Build Solution** from the **Build** menu.

To run the application

1. Make sure **MyExecRefsAssembly** is selected as the default project. From the **Solution Explorer**, select **MyExecRefsAssembly**, and then select **Set As StartUp Project** from the **Project** menu.
2. To run the project, select **Start Without Debugging** from the **Debug** menu. The output should look like this:

```
a + b = 106.4
a - b = -91.6
a * b = 732.6
a / b = 0.0747474747474748
```

See Also

Other Resources

Visual C++ Guided Tour
Assemblies in the Common Language Runtime
Application Domains

Walkthrough: Creating and Using a Dynamic Link Library

In this walkthrough, you will create a dynamic link library (DLL) containing useful routines that can be used by other applications. Using DLLs is a great way to reuse code. Rather than re-implementing these routines in every program you create, you write them once and reference them from applications that need the functionality.

This walkthrough uses native C++. For a walkthrough using native C++ to create a static library, see Walkthrough: Creating and Using a Static Library. For a walkthrough using Visual C++ that targets the Common Language Runtime, see Walkthrough: Creating and Using a Managed Assembly.

This walkthrough covers the following:

- Creating a new dynamic link library (DLL) project
- Adding a class to the dynamic link library
- Creating an application that references the dynamic link library
- Using the functionality from the class library in the console application
- Running the application

Prerequisites

This topic assumes you understand the fundamentals of the C++ language.

To create a new dynamic link library (DLL) project

1. From the **File** menu, select **New** and then **Project...**.
2. From the **Project types** pane, under **Visual C++**, select **Win32**.
3. From the **Templates** pane, select **Win32 Console Application**.
4. Choose a name for the project, such as **MathFuncsDll**, and enter it in the **Name** field. Choose a name for the solution, such as **DynamicLibrary**, and enter it in the **Solution Name** field.
5. Press **OK** to start the Win32 application wizard. From the **Overview** page of the **Win32 Application Wizard** dialog, press **Next**.
6. From the **Application Settings** page of the **Win32 Application Wizard**, under **Application type**, select **DLL** if it is available or **Console application** if **DLL** is not available. Some versions of Visual Studio do not support creating a DLL project using wizards. You can change this later to make your project compile into a DLL.
7. From the **Application Settings** page of the **Win32 Application Wizard**, under **Additional options**, select **Empty project**.
8. Press **Finish** to create the project.

To add a class to the dynamic link library

1. To create a header file for a new class, from the **Project** menu, select **Add New Item...**. The **Add New Item** dialog will be displayed. From the **Categories** pane, under **Visual C++**, select **Code**. From the **Templates** pane, select **Header File (.h)**. Choose a name for the header file, such as **MathFuncsDll.h**, and press **Add**. A blank file will be displayed.
2. Add a simple class named **MyMathFuncs** to do common mathematical operations, such as addition, subtraction, multiplication, and division. The code should resemble the following:

```
// MathFuncsDll.h

namespace MathFuncs
{
    class MyMathFuncs
    {
    public:
```

```
        // Returns a + b
        static __declspec(dllexport) double Add(double a, double b);

        // Returns a - b
        static __declspec(dllexport) double Subtract(double a, double b);

        // Returns a * b
        static __declspec(dllexport) double Multiply(double a, double b);

        // Returns a / b
        // Throws DivideByZeroException if b is 0
        static __declspec(dllexport) double Divide(double a, double b);
    };
}
```

3. Note the **__declspec(dllexport)** modifier in the method declarations above. These modifiers enable the method to be exported by the DLL so they can be used by other applications. For more information, see dllexport, dllimport.

4. To create a source file for a new class, from the **Project** menu, select **Add New Item...**. The **Add New Item** dialog will be displayed. From the **Categories** pane, under **Visual C++**, select **Code**. From the **Templates** pane, select **C++ File (.cpp)**. Choose a name for the source file, such as **MathFuncsDll.cpp**, and press **Add**. A blank file will be displayed.

5. Implement the functionality for **MyMathFuncs** in the source file. The code should resemble the following:

```
// MathFuncsDll.cpp
// compile with: /EHsc /LD

#include "MathFuncsDll.h"

#include <stdexcept>

using namespace std;

namespace MathFuncs
{
    double MyMathFuncs::Add(double a, double b)
    {
        return a + b;
    }

    double MyMathFuncs::Subtract(double a, double b)
    {
        return a - b;
    }

    double MyMathFuncs::Multiply(double a, double b)
    {
        return a * b;
    }

    double MyMathFuncs::Divide(double a, double b)
    {
        if (b == 0)
        {
            throw new invalid_argument("b cannot be zero!");
        }

        return a / b;
    }
```

```
}
```

6. To build the project into a DLL, from the **Project** menu, select **MathFuncsDll Properties...**. From the left pane, under **Configuration Properties**, select **General**. From the right pane, change the **Configuration Type** to **Dynamic Library (.dll)**. Press **OK** to save the changes.

   Note

   If building from the command line, use the **/LD** compiler option to specify that the output file should be a DLL. For more information, see /MD, /MT, /LD (Use Run-Time Library).

7. Compile the dynamic link library by selecting **Build Solution** from the **Build** menu. This creates a DLL that can be used by other programs. For more information on DLLs, see DLLs.

To create an application that references the dynamic link library

1. To create an application that will reference and use the dynamic link library that was just created, from the **File** menu, select **New** and then **Project...**.
2. From the **Project types** pane, under **Visual C++**, select **Win32**.
3. From the **Templates** pane, select **Win32 Console Application**.
4. Choose a name for the project, such as **MyExecRefsDll**, and enter it in the **Name** field. Next to **Solution**, select **Add to Solution** from the drop down list. This will add the new project to the same solution as the dynamic link library.
5. Press **OK** to start the **Win32 Application Wizard**. From the **Overview** page of the **Win32 Application Wizard** dialog, press **Next**.
6. From the **Application Settings** page of the **Win32 Application Wizard**, under **Application type**, select **Console application**.
7. From the **Application Settings** page of the **Win32 Application Wizard**, under **Additional options**, deselect **Precompiled header**.
8. Press **Finish** to create the project.

To use the functionality from the class library in the console application

1. After you create a new Console Application, an empty program is created for you. The name for the source file will be the same as the name you chose for the project above. In this example, it is named **MyExecRefsDll.cpp**.
2. To use the math routines that were created in the dynamic link library, you must reference it. To do this, select **References...** from the **Project** menu. From the **Property Pages** dialog, expand the **Common Properties** node and select **References**. Then select the **Add New Reference...** button. For more information on the **References...** dialog, see References, Common Properties, <Projectname> Property Pages Dialog Box.
3. The **Add Reference** dialog is displayed. This dialog lists all the libraries that you can reference. The **Project** tab lists all the projects in the current solution and any libraries they contain. From the **Projects** tab, select **MathFuncsDll**. Then select **OK**. For more information on the **Add Reference** dialog, see Add Reference Dialog Box.
4. To reference the header files of the dynamic link library, you must modify the include directories path. To do this, from the **Property Pages** dialog, expand the **Configuration Properties** node, then the **C/C++** node, and select **General**. Next to **Additional Include Directories**, type in the path to the location of the **MathFuncsDll.h** header file.
5. Dynamic link libraries are not loaded by the executable until runtime. You must tell the system where to locate **MathFuncsDll.dll**. This is done using the **PATH** environment variable. To do this, from the **Property Pages** dialog, expand the **Configuration Properties** node and select **Debugging**. Next to **Environment**, type in the following: `PATH=<path to MathFuncsDll.dll file>`, where *<path to MathFuncsDll.dll file>* is replaced with the actual location of **MathFuncsDll.dll**. Press **OK** to save all the changes made.

   Note

If you intend to run the executable from the command line rather than from within Visual Studio, then you must manually update the **PATH** environment variable from the command prompt as follows: `set PATH=%PATH%;<path to MathFuncsDll.dll file>`, where <*path to MathFuncsDll.dll file*> is replaced with the actual location of **MathFuncsDll.dll**.

6. You can now use the **MyMathFuncs** class in this application. Replace the contents of **MyExecRefsDll.cpp** with the following code:

```
// MyExecRefsDll.cpp
// compile with: /EHsc /link MathFuncsDll.lib

#include <iostream>

#include "MathFuncsDll.h"

using namespace std;

int main()
{
    double a = 7.4;
    int b = 99;

    cout << "a + b = " <<
        MathFuncs::MyMathFuncs::Add(a, b) << endl;
    cout << "a - b = " <<
        MathFuncs::MyMathFuncs::Subtract(a, b) << endl;
    cout << "a * b = " <<
        MathFuncs::MyMathFuncs::Multiply(a, b) << endl;
    cout << "a / b = " <<
        MathFuncs::MyMathFuncs::Divide(a, b) << endl;

    return 0;
}
```

7. Build the executable by selecting **Build Solution** from the **Build** menu.

To run the application

1. Make sure **MyExecRefsDll** is selected as the default project. From the **Solution Explorer**, select **MyExecRefsDll**, and then select **Set As StartUp Project** from the **Project** menu.

2. To run the project, select **Start Without Debugging** from the **Debug** menu. The output should look like this:

```
a + b = 106.4
a - b = -91.6
a * b = 732.6
a / b = 0.0747475
```

See Also

Concepts

DLLs

Other Resources

Visual C++ Guided Tour

General Concepts

Walkthrough: Creating and Using a Static Library

In this walkthrough, you will create a static library (LIB) containing useful routines that can be used by other applications. Using static libraries is a great way to reuse code. Rather than re-implementing these routines in every program you create, you write them once and reference them from applications that need the functionality.

This walkthrough uses native C++. For a walkthrough using native C++ to create a dynamic link library (DLL), see Walkthrough: Creating and Using a Dynamic Link Library. For a walkthrough using Visual C++ that targets the Common Language Runtime, see Walkthrough: Creating and Using a Managed Assembly.

This walkthrough covers the following:

- Creating a new static library project
- Adding a class to the static library
- Creating an application that references the static library
- Using the functionality from the static library in the console application
- Running the application

Prerequisites

This topic assumes you understand the fundamentals of the C++ language.

To create a new static library project

1. From the **File** menu, select **New** and then **Project...**.
2. From the **Project types** pane, under **Visual C++**, select **Win32**.
3. From the **Templates** pane, select **Win32 Console Application**.
4. Choose a name for the project, such as **MathFuncsLib**, and enter it in the **Name** field. Choose a name for the solution, such as **StaticLibrary**, and enter it in the **Solution Name** field.
5. Press **OK** to start the Win32 application wizard. From the **Overview** page of the **Win32 Application Wizard** dialog, press **Next**.
6. From the **Application Settings** page of the **Win32 Application Wizard**, under **Application type**, select **Static library.**
7. From the **Application Settings** page of the **Win32 Application Wizard**, under **Additional options**, deselect **Precompiled header**.
8. Press **Finish** to create the project.

To add a class to the static library

1. To create a header file for a new class, from the **Project** menu, select **Add New Item...**. The **Add New Item** dialog will be displayed. From the **Categories** pane, under **Visual C++**, select **Code**. From the **Templates** pane, select **Header File (.h)**. Choose a name for the header file, such as **MathFuncsLib.h**, and press **Add**. A blank file will be displayed.
2. Add a simple class named **MyMathFuncs** to do common mathematical operations, such as addition, subtraction, multiplication, and division. The code should resemble the following:

```
// MathFuncsLib.h

namespace MathFuncs
{
    class MyMathFuncs
    {
    public:
        // Returns a + b
```

```
        static double Add(double a, double b);

        // Returns a - b
        static double Subtract(double a, double b);

        // Returns a * b
        static double Multiply(double a, double b);

        // Returns a / b
        // Throws DivideByZeroException if b is 0
        static double Divide(double a, double b);
    };
}
```

3. To create a source file for a new class, from the **Project** menu, select **Add New Item...**. The **Add New Item** dialog will be displayed. From the **Categories** pane, under **Visual C++**, select **Code**. From the **Templates** pane, select **C++ File (.cpp)**. Choose a name for the source file, such as **MathFuncsLib.cpp**, and press **Add**. A blank file will be displayed.

4. Implement the functionality for **MyMathFuncs** in the source file. The code should resemble the following:

```
// MathFuncsLib.cpp
// compile with: /c /EHsc
// post-build command: lib MathFuncsLib.obj

#include "MathFuncsLib.h"

#include <stdexcept>

using namespace std;

namespace MathFuncs
{
    double MyMathFuncs::Add(double a, double b)
    {
        return a + b;
    }

    double MyMathFuncs::Subtract(double a, double b)
    {
        return a - b;
    }

    double MyMathFuncs::Multiply(double a, double b)
    {
        return a * b;
    }

    double MyMathFuncs::Divide(double a, double b)
    {
        if (b == 0)
        {
            throw new invalid_argument("b cannot be zero!");
        }

        return a / b;
    }
}
```

5. To build the project into a static library, from the **Project** menu, select **MathFuncsLib Properties...**. From the left pane,

under **Configuration Properties**, select **General**. From the right pane, change the **Configuration Type** to **Static Library (.lib)**. Press **OK** to save the changes.

> **Note**
>
> If building from the command line, you must build the program in two steps. First, compile the code using **Cl.exe** with the **/c** compiler option (`cl /c /EHsc MathFuncsLib.cpp`). This will create an object file named **MathFuncsLib.obj**. For more information, see /c (Compile Without Linking). Second, link the code using the Library Manager **Lib.exe** (`lib MathFuncsLib.obj`). This will create the static library **MathFuncsLib.lib**. For more information on the Library Manager, see LIB Reference.

6. Compile the static library by selecting **Build Solution** from the **Build** menu. This creates a static library that can be used by other programs.

To create an application that references the static library

1. To create an application that will reference and use the static library that was just created, from the **File** menu, select **New** and then **Project...**.
2. From the **Project types** pane, under **Visual C++**, select **Win32**.
3. From the **Templates** pane, select **Win32 Console Application**.
4. Choose a name for the project, such as **MyExecRefsLib**, and enter it in the **Name** field. Next to **Solution**, select **Add to Solution** from the drop down list. This will add the new project to the same solution as the static library.
5. Press **OK** to start the **Win32 Application Wizard**. From the **Overview** page of the **Win32 Application Wizard** dialog, press **Next**.
6. From the **Application Settings** page of the **Win32 Application Wizard**, under **Application type**, select **Console application**.
7. From the **Application Settings** page of the **Win32 Application Wizard**, under **Additional options**, deselect **Precompiled header**.
8. Press **Finish** to create the project.

To use the functionality from the static library in the console application

1. After you create a new Console Application, an empty program is created for you. The name for the source file will be the same as the name you chose for the project above. In this example, it is named **MyExecRefsLib.cpp**.
2. To use the math routines that were created in the static library, you must reference it. To do this, select **References...** from the **Project** menu. From the **Property Pages** dialog, expand the **Common Properties** node and select **References**. Then select the **Add New Reference...** button. For more information on the **References...** dialog, see References, Common Properties, <Projectname> Property Pages Dialog Box.
3. The **Add Reference** dialog is displayed. This dialog lists all the libraries that you can reference. The **Project** tab lists all the projects in the current solution and any libraries they contain. From the **Projects** tab, select **MathFuncsLib**. Then select **OK**. For more information on the **Add Reference** dialog, see Add Reference Dialog Box.
4. To reference the header files of the static library, you must modify the include directories path. To do this, from the **Property Pages** dialog, expand the **Configuration Properties** node, then the **C/C++** node, and select **General**. Next to **Additional Include Directories**, type in the path to the location of the **MathFuncsLib.h** header file.
5. You can now use the **MyMathFuncs** class in this application. Replace the contents of **MyExecRefsLib.cpp** with the following code:

```
// MyExecRefsLib.cpp
// compile with: /EHsc /link MathFuncsLib.lib

#include <iostream>

#include "MathFuncsLib.h"
```

```
using namespace std;

int main()
{
    double a = 7.4;
    int b = 99;

    cout << "a + b = " <<
        MathFuncs::MyMathFuncs::Add(a, b) << endl;
    cout << "a - b = " <<
        MathFuncs::MyMathFuncs::Subtract(a, b) << endl;
    cout << "a * b = " <<
        MathFuncs::MyMathFuncs::Multiply(a, b) << endl;
    cout << "a / b = " <<
        MathFuncs::MyMathFuncs::Divide(a, b) << endl;

    return 0;
}
```

6. Build the executable by selecting **Build Solution** from the **Build** menu.

To run the application

1. Make sure **MyExecRefsLib** is selected as the default project. From the **Solution Explorer**, select **MyExecRefsLib**, and then select **Set As StartUp Project** from the **Project** menu.
2. To run the project, select **Start Without Debugging** from the **Debug** menu. The output should look like this:

```
a + b = 106.4
a - b = -91.6
a * b = 732.6
a / b = 0.0747475
```

See Also

Other Resources

Visual C++ Guided Tour

General Concepts

Introduction to Visual C++ for UNIX Users

This topic provides information for UNIX users who are new to Visual C++ and want to become productive with Visual C++.

Getting Started on the Command Line

You can use Visual C++ from the command line in a similar way that you would use a UNIX command-line environment. You compile from the command prompt with the command-line C and C++ compiler (CL.EXE) and tools, including NMAKE.EXE, the Microsoft version of the UNIX make utility.

In UNIX, commands are installed in a common folder, such as /usr/bin. In Visual C++, the command-line tools are installed in your installation directory at VC\bin (on a typical installation at Program Files\Microsoft Visual Studio 8\VC\bin). To use the command-line tools, run vsvars32.bat, which is located in your installation directory at Common7\Tools. This adds your bin directory to your path and sets up other paths that are necessary to compile Visual C++ programs from the command line.

Note
If you open a command prompt with the **Visual Studio Command Line Prompt** from the **Start** menu, then vsvars32.bat is run for you.

To take advantage of more powerful features, such as the debugger, statement completion, and so on, you need to use the development environment. For more information, see Building on the Command Line and How to: Compile a Native C++ Program from the Command Line.

Debugging Your Code

If you use the command line and run your applications on your development workstation, you will see that a dialog box to run the Visual Studio debugger is displayed when your code encounters a memory access violation, unhandled exception, or other unrecoverable errors. If you click **OK**, then the Visual Studio development environment is started, and the debugger will open to the point of failure. It is possible to debug your applications this way, and, in this case, your source code would only be available if you compiled with the /Z7, /Zi, /ZI (Debug Information Format) switch. For more information, see Debugging Visual C++ and Walkthrough: Using the Visual Studio IDE.

Using the Development Environment

It is easier to use the development environment to edit and build your source code in a *project*. A project is a collection of source and related files that will be compiled into a single unit, such as a library or executable. A project also contains information on how the files are to be built. Information about projects is stored in a project file with the extension .prj.

An application that consists of multiple libraries and executables, each potentially built with a different set of compiler options or even in a different language, are stored in multiple projects that are part of a single *solution*. A solution is an abstraction for a container to group multiple projects together. Information about solutions is stored in a solution file with the extension .sln. For more information, see Managing Solutions, Projects, and Files and Walkthrough: Using the Visual Studio IDE.

Importing Your Existing Code

You can use Visual C++ to use existing code that is set up to compile with or without a makefile and put it into a Visual Studio project. For more information, see the **Create Project From Existing Code Files Wizard**. For more information, see How to: Create a C++ Project from Existing Code.

Creating a New Project

You can create new projects in the development environment. Visual C++ provides numerous templates that provide standard code for various common projects. You can use application wizards to generate projects with code outlines for various application types.

You can start with an empty project by using the **Console Application (Win32) Wizard**. Select the **Empty Project** check box. You can then add new and existing files to the project later.

When you create a project, you must name the project. By default, the project name equals the name of the dynamic-link library (DLL) or executable that is build from the project. For more information, see How to: Create Solutions and Projects.

Microsoft-Specific Modifiers

Visual C++ contains several extensions to the standard C++ programming language. These extensions are used to specify storage class attributes, function calling conventions, and based addressing, among other things. For a complete list of all Visual C++ extensions, see Microsoft-Specific Modifiers.

You can disable all Microsoft-specific extensions to C++ by using the **/Za** compiler option. This option is recommended if you want to write code to run on multiple platforms. For more information on the **/Za** compiler option, see /Za, /Ze (Disable Language Extensions). For more information on Visual C++ conformance, see Compatibility and Compliance Issues in Visual C++.

Precompiled Headers

The Microsoft C and C++ compilers provide options for precompiling any C or C++ code, including inline code. Using this performance feature, you can compile a stable body of code, store the compiled state of the code in a file, and, during subsequent compilations, combine the precompiled code with code that is still under development. Each subsequent compilation is faster because the stable code does not need to be recompiled.

By default, all precompiled code is specified in the files **stdafx.h** and **stdafx.cpp**. The **New Project** wizard will automatically create these files for you unless you deselect the **Precompiled header** option. For more information on precompiled headers, see Creating Precompiled Header Files.

Related Sections

For more information, see Porting from UNIX to Win32.

See Also

Other Resources

Visual C++ Guided Tour

How Do I in Visual C++

This page links to help on popular Visual C++ tasks.

Getting Started

Using Visual C++... Creating Managed Applications... Creating Native Applications... Visual C++ Examples... File Handling with Visual C++... Windows Operations with Visual C++... Help... more...

Development Environment

Visual Studio Basics... Solutions and Projects... Precompiled Headers... Editor... more...

Windows Applications

Windows Forms... MFC... more...

Web Applications

Web Services... ATL Server... MFC and WinInet... .NET Remoting... ASP.NET... more...

CLR Development

Windows Forms... Mixed Assemblies... Pure / Verifiable Assemblies... Reflection... more...

Native Development

Win32... COM/ATL/MFC... Native Programming... Precompiled Headers... more...

Interop

CLR/Native Interoperability... C++ Interop... COM Interop... P/Invoke... Using Windows Forms in MFC... more...

Migrating, Porting, and Upgrading

Migrating to Visual C++... Migrating from UNIX... Porting to 64-bit... Upgrading from Previous Versions of Visual C++... more...

Security

Security Basics... Secure C Runtime (CRT)... Standard C++ Library Security Improvements... more...

64-bit Programming

64-bit Programming... more...

Command Line

Visual C++ on the Command Line... Visual C++ Examples... File Handling with Visual C++... Windows Operations with Visual C++... more...

Data Access

Programming... ADO.NET... ADO... ODBC... OLE... more...

Debugging, Testing, and Deployment

Debugging... Testing... Deployment... more...

Performance

Performance... more...

C++ Language

Differences from Managed Extensions for C++... Visual C++ Examples... File Handling with Visual C++... Windows Operations with Visual C++... more...

Reference

C/C++ Languages... CRT... Standard C++ Library... .NET Framework... ATL... ATL Server... MFC... more...

See Also

Reference

Visual C++ (How Do I in Smart Devices)

Other Resources

Visual C++

Getting Started with Visual C++

Visual C++ Guided Tour

Getting Started (How Do I in Visual C++)

This page links to help on introductory tasks related to Visual C++. To view other categories of popular tasks covered in Help, see How Do I in Visual C++.

Using Visual C++

Walkthrough: Using the Visual Studio IDE

Introduces the Visual Studio Integrated Development Environment (IDE). Demonstrates useful features for Visual C++ developers.

How to: Compile a C++ Program that Targets the CLR in Visual Studio

Demonstrates how to compile a Visual C++ program using the Visual Studio IDE.

How to: Compile a Code Example from the Help Topics

Demonstrates how to compile a code sample from the Help topics that accompany Visual Studio.

How to: Compile a Native C++ Program from the Command Line

Demonstrates how to compile a Visual C++ program from the command line.

How to: Use Visual C++ Express Edition with the Microsoft Platform SDK

Provides instructions for installing the Microsoft Platform SDK for use with Visual C++ Express Edition.

Introduction to Visual C++ for UNIX Users

Introduces Visual Studio and Visual C++ for developers new to the Microsoft environment.

How to: Create a C++ Project from Existing Code

Demonstrates how to import existing code into Visual Studio.

Creating Managed Applications

How to: Create a Windows Forms Application

Explains how to use Visual Studio to create a Windows Forms application.

Walkthrough: Creating a Windows Forms Control

Explains how to use Visual Studio to create a Windows Forms control and how to use that control in an application.

ASP.NET Web Service Template (C++)

Introduces the ASP.NET Web Services Template used to author XML Web services.

Creating and Developing Managed Device Projects

Discusses some of the differences between developing for managed devices and developing for the desktop.

How to: Create a SQL Server Project (C++)

Provides an overview of the SQL Server Class Library template.

CLR Console Application Template (C++)

Introduces the Console Application template used to create applications that are run from the command line.

Walkthrough: Creating and Using a Managed Assembly

Demonstrates how to create a managed assembly and how to use that assembly from another application.

Creating Native Applications

How to: Create a Standard C++ Program

Explains how to create and compile a standard C++ program using Visual Studio.

MFC Application Wizard

Explains how to use the MFC Application Wizard to create an MFC application.

MFC ActiveX Control Wizard

Explains how to use the MFC ActiveX Control Wizard to create an MFC ActiveX control.

Win32 Application Wizard

Explains how to use the Win32 Application Wizard to create a standard C++ program.

MFC DLL Wizard

Explains how to use the MFC DLL Wizard to create an MFC Dynamic Link Library.

ATL Project Wizard

Explains how to use the ATL Project Wizard to create an Active Template Library application.

ATL Server Project Wizard

Explains how to use the ATL Server Project Wizard to create a native XML Web service application.

Windows Service Template (C++)

Explains how to use the Windows Service Template to create a Windows service, a long-running executable application that runs in its own Windows session.

Creating and Porting Visual C++ Device Projects

Includes topics on creating new Visual C++ device projects, and allowing existing projects to target devices in the Visual Studio 2005 environment.

Walkthrough: Creating and Using a Dynamic Link Library

Demonstrates how to create a Dynamic Link Library and how to use that DLL from another application.

Visual C++ Examples

C++ Language (How Do I in Visual C++)

Introduces the new syntax for C++ used to target the Common Language Runtime (CLR).

File Handling with Visual C++

How to: Read a Binary File

Demonstrates reading binary data from a file.

How to: Write a Binary File

Demonstrates writing binary data to a file.

How to: Read a Text File

Demonstrates how to open and read a text file one line at a time.

How to: Write a Text File

Demonstrates how to create a text file and write text to it using the StreamWriter class.

How to: Enumerate Files in a Directory

Demonstrates how to retrieve a list of the files in a directory.

How to: Monitor File System Changes

Uses FileSystemWatcher to register for events corresponding to files being created, changed, deleted, or renamed.

How to: Retrieve File Information

Demonstrates the FileInfo class. When you have the name of a file, you can use this class to retrieve information about the file such as the file size, directory, full name, and date and time of creation and of the last modification.

Windows Operations with Visual C++

How to: Write Data to the Windows Registry

Uses the CurrentUser key to create a writable instance of the RegistryKey class.

How to: Read Data from the Windows Registry

Uses the **CurrentUser** key to read data from the Windows registry.

How to: Retrieve Text from the Clipboard

Uses the GetDataObject member function to return a pointer to the IDataObject interface, which can then be queried for the format of the data and used to retrieve the actual data.

How to: Store Text in the Clipboard

Uses the Clipboard object defined in the System.Windows.Forms namespace to store a string.

How to: Retrieve the Windows Version

Demonstrates how to retrieve the platform and version information of the current operating system.

How to: Retrieve Time Elapsed Since Startup

Demonstrates how to determine the tick count, or milliseconds that have elapsed since Windows was started.

Help

Help on Help (Microsoft Document Explorer Help)

Provides information about using Microsoft Document Explorer and the various Help collections available to you.

Techniques for Locating Help

Provides useful strategies for finding the type of information you are looking for.

See Also

Concepts

How Do I in Visual C++

Other Resources

Getting Started with Visual C++

Visual C++ Guided Tour

Development Environment (How Do I in Visual C++)

This page links to help on tasks related to the development environment for Visual C++. To view other categories of popular tasks covered in Help, see How Do I in Visual C++.

Visual Studio Basics

Help on Help (Microsoft Document Explorer Help)

Provides information about using Microsoft Document Explorer and the various Help collections available to you.

Introduction to Visual C++ for UNIX Users

Provides information for UNIX users who are new to Visual C++ and want to become productive with Visual C++.

Walkthrough: Using the Visual Studio IDE

Demonstrates how to create a new standard C++ program and test its functionality using features available in Visual Studio for the C++ developer.

How to: Compile a C++ Program that Targets the CLR in Visual Studio

Demonstrates how to use Visual Studio to compile a Visual C++ program.

How to: Compile a Code Example from the Help Topics

Describes how to locate the Visual C++ code example programs in the help topics and how to compile and run them.

How to: Use Visual C++ Express Edition with the Microsoft Platform SDK

Explains how Visual C++ Express Edition users can install the Microsoft Platform SDK.

Debugging in Visual Studio

Demonstrates how you can use Visual Studio to stop at procedure locations, inspect memory and register values, change variables, observe message traffic, and get a close look at how your code works or does not work.

Solutions and Projects

How to: Create a Standard C++ Program

Demonstrates how to create a new project, add a new file to the project, edit the file to add C++ code, and then compile and run the program using Visual Studio.

How to: Create a Windows Forms Application

Creates a Windows Forms application using several standard controls from the Toolbox.

Walkthrough: Creating a Windows Forms Control

Creates a Windows Forms control, a component that can be placed in a Windows Forms application.

How to: Create a C++ Project from Existing Code

Discusses how to create a Visual C++ project using the Create Project From Existing Code Files wizard.

Creating a Makefile Project

Demonstrates how to create a Makefile Project using the Makefile Project Wizard.

Walkthrough: Using Profile-Guided Optimizations

Demonstrates how to use profile-guided optimization (PGO) for varied scenarios that range from simple implementations using PGO build menus in the Visual Studio Integrated Development Environment (IDE), to more advanced scenarios that require custom configurations.

How to: Display Call Graphs

Discusses how to display call graphs using the Call Browser.

How to: Display Inheritance Graphs

Discusses how to use the Class View for inheritance browsing.

How to: Run Multiprocessor Builds in the IDE

Describes how to run Multiprocessor Builds from the IDE.

How to: Add New Property Sheets to C++ Projects

Describes how to create new Property Sheets (C++), and add them to your Visual C++ projects.

How to: Inherit Project Settings from Property Sheets

Describes how to specify that your Visual C++ project or project property sheet will inherit its project configuration from a property sheet.

How to: Organize Project Output Files for Builds

Describes best practices for organizing project output files.

Precompiled Headers

Creating Precompiled Header Files

Describes how the Microsoft C and C++ compilers provide options for precompiling any C or C++ code, including inline code.

When to Precompile Source Code

Describes when it is appropriate to use precompiled headers.

Two Choices for Precompiling Code

Describes the two choices for precompiling code: automatic precompiling and manual precompiling.

Precompiled Header Consistency Rules

Discusses guidelines that will help you use precompiled headers more efficiently.

Using Precompiled Headers in a Project

Describes a method for using the manual precompiled-header options in a project.

Editor

Code and Text Editor

Introduces the Code and Text Editor, the word processor of the integrated development environment (IDE).

Using IntelliSense

Explains how you can use IntelliSense to find the information you need and insert language elements directly into your code.

See Also

Reference

Debugging, Testing, and Deployment (How Do I in Visual C++)

Concepts

How Do I in Visual C++

Other Resources

Integrated Development Environment for Visual Studio

Windows Applications (How Do I in Visual C++)

This page links to help on tasks related to creating Windows applications using Visual C++. To view other categories of popular tasks covered in Help, see How Do I in Visual C++.

Windows Forms

Windows Forms Walkthroughs and How-to Topics

Lists the Windows Forms walkthroughs that will help you effectively implement Windows Forms in your Visual C++ application.

Code for Windows Forms Applications

Lists the code examples that demonstrate how to perform common programming tasks in Windows Forms applications.

MFC

MFC Fundamentals

Introduces MFC and the fundamental support for programming for Microsoft Windows that it provides.

CWinApp: The Application Class

Discuses **CWinApp**, the main application class in MFC that encapsulates the initialization, running, and termination of an application for the Windows operating system.

Collections

Introduces the MFC collections used to manage groups of objects.

MFC Internet Programming Tasks

Contains detailed steps for adding Internet support to your applications.

Serialization: Making a Serializable Class

Lists the five main steps required to make a class serializable.

Document Template Creation

Describes creating a document using a Document Template.

Document/View Creation

Summarizes the types of documents that are created by an application object, a document template, a newly created document, and a newly created frame window.

Creating New Documents, Windows, and Views

Illustrates the creation process for documents, views, and frame windows.

Exported DLL Function Entry Points

Describes the AFX_MANAGE_STATE macro used to maintain the proper global state when switching from the DLL module to the calling application's DLL.

Window Procedure Entry Points

Describes how the Windows entry point is determined.

Idle Loop Processing

Focuses on using idle processing to perform lengthy processing "in the background."

Exception Handling in MFC

Explains the exception-handling mechanisms available in MFC.

MAPI

Describes the Microsoft Messaging Application Programming Interface (MAPI) for client message application developers.

MFC ISAPI Classes

Describes the MFC classes that wrap the Internet Server API (ISAPI).

How MFC Makes It Easier to Create Internet Client Applications

Describes how MFC abstracts the Internet protocols for HTTP, FTP, and gopher into a high-level application programming interface, providing a fast and straightforward path to making applications Internet-aware.

Declaring Message Handler Functions

Lists the rules and conventions that govern the names of message-handler functions.

How to: Display Command Information in the Status Bar

Describes how to use the status bar to provide helpful feedback as the user of your application moves the mouse through items in the menus.

Deriving a Class from CObject

Describes the minimum steps necessary to derive a class from **CObject**.

Do I Have to Derive New Classes from CObject?

Explains when you might want to derive your class from **CObject**.

What Does it Cost me to Derive a Class from CObject?

Describes the overhead associated with deriving from **CObject**.

See Also

Concepts

How Do I in Visual C++

Web Applications (How Do I in Visual C++)

This page links to help on tasks related to creating Web applications using Visual C++. To view other categories of popular tasks covered in Help, see How Do I in Visual C++.

Web Services

Introduction to Programming XML Web Services in Managed Code

Describes the benefits of using XML Web Services in managed code.

XML Web Services Created Using ASP.NET and XML Web Service Clients

Provides an overview of XML Web Services using ASP.NET.

How to: Create ASP.NET Web Services

Describes how to create an ASP.NET Web Service using Visual Studio.

How to: Debug XML Web Services in Managed Code

Describes several scenarios for debugging XML Web Services applications.

How to: Deploy XML Web Services in Managed Code

Describes how to deploy XML Web Services applications using Visual Studio.

How to: Enable Discovery for XML Web Services

Describes several scenarios for making your XML Web Service discoverable.

How to: Explore Existing XML Web Services Created Using ASP.NET

Describes how to access the service help page for an XML Web Service.

How to: Throw Exceptions from a Web Service Created Using ASP.NET

Demonstrates how to throw a SoapException from an XML Web Service.

Testing XML Web Services

Describes how to test XML Web Services created using ATL Server.

ATL Server

ATL Server Tutorial

Describes how to create a simple online store using ATL Server, a set of native C++ classes for developing XML Web services, Web applications, and other server-based applications.

ATL Server Tasks

Provides instructions for common tasks that you may wish to perform in your ATL Server Web applications.

XML Web Services Created with ATL Server

Focuses on how to create, distribute, and consume XML Web services using ATL Server and Visual C++.

Adding an XML Web Service with ATL Server

Describes how to add an XML Web service created using ATL Server to your project.

Debugging XML Web Services Created with ATL Server

Describes additional steps to take to debug an XML Web Service created using ATL Server.

MFC and WinInet

WinInet Basics

Provides an introduction to WinInet.

How WinInet Makes It Easier to Create Internet Client Applications

Describes how, using WinInet, you can write Internet client applications at a higher level of programming, without having to deal with WinSock, TCP/IP, or the details of specific Internet protocols.

How MFC Makes It Easier to Create Internet Client Applications

Describes the Microsoft Foundation Classes that encapsulate the Win32 Internet Extension (WinInet) functions.

Writing an Internet Client Application Using MFC WinInet Classes

Describes how to write an internet client application using MFC.

Steps in a Typical Internet Client Application

Shows the steps you might perform in a typical Internet client application.

Steps in a Typical HTTP Client Application

Shows the steps you might perform in a typical HTTP client application.

Steps in a Typical FTP Client Application

Shows the steps you might perform in a typical FTP client application.

.NET Remoting

Remote Objects

Describes how .NET remoting enables client applications to use objects in other processes on the same computer or on any other computer available on its network.

.NET Framework Remoting Overview

Summarizes how .NET remoting enables you to build widely distributed applications easily, whether application components are all on one computer or spread out across the entire world.

Remote Objects How-to Topics

Lists all How to topics that demonstrate the use of .NET Remoting.

Remoting Examples

Lists all examples that demonstrate the use of .NET Remoting.

ASP.NET

Building ASP.NET Applications

Describes how to build ASP.NET applications.

How to: Create Performance Session for ASP.NET Applications

Describes how to use the **Performance Wizard** to create a performance session for ASP.NET applications.

How to: Debug ASP.NET Exceptions

Describes how to set up Visual Studio to debug ASP.NET applications.

How to: Enable Debugging for ASP.NET Applications

Describes how to enable debugging in the project properties and in the application's web.config file.

See Also

Concepts

How Do I in Visual C++

CLR Development (How Do I in Visual C++)

This page links to help on tasks related to CLR development using Visual C++. To view other categories of popular tasks covered in Help, see How Do I in Visual C++.

Windows Forms

Windows Forms Walkthroughs and How-to Topics

Lists the Windows Forms walkthroughs that will help you effectively implement Windows Forms in your Visual C++ application.

Code for Windows Forms Applications

Lists the code examples that demonstrate how to perform common programming tasks in Windows Forms applications.

Mixed Assemblies

How To: Migrate to /clr

Discusses issues that arise when compiling native code with **/clr** and when converting a project to managed.

How to: Compile MFC and ATL Code with /clr

Discusses how to compile existing MFC and ATL programs to target the Common Language Runtime.

How To: Remove Dependency on _vcclrit.h

Contains the recommended modifications required to remove dependency on this header file.

Performance Considerations for Interop (C++)

Provides guidelines for reducing the effect of managed/unmanaged interop transitions on run-time performance.

Pure / Verifiable Assemblies

Pure and Verifiable Code

Describes the differences between mixed, pure, and verifiable code.

How to: Create Verifiable C++ Projects

Describes how to set project properties and modify project source files to transform your Visual C++ projects to produce verifiable applications.

How to: Migrate to /clr:safe

Lists the issues that will generate verifiability errors when compiling with /clr:safe.

Walkthrough: Using /clr:pure Features

Demonstrates the use of reflection in a pure MSIL application.

How to: Create a Partially Trusted Application by Removing Dependency on msvcm80.dll

Discusses how to create a partially trusted Common Language Runtime application using Visual C++ by removing dependency on msvcm80.dll.

Reflection

Reflection in C++

Explains how to use Visual C++ to discover, examine, and invoke data types at runtime through reflection.

How to: Implement a Plug-In Component Architecture using Reflection

Demonstrate the use of reflection to implement a simple "plug-in" architecture.

How to: Enumerate Data Types in Assemblies using Reflection

Demonstrates the enumeration of public types and members using the System.Reflection API.

typeid

Explains how to return the Type of a specified type using **typeid**.

See Also

Reference

Interop (How Do I in Visual C++)

Concepts

How Do I in Visual C++

Native Development (How Do I in Visual C++)

This page links to help on tasks related to native development using Visual C++. To view other categories of popular tasks covered in Help, see How Do I in Visual C++.

Win32

Microsoft Platform SDK Introduction

Provides information about the application programming interfaces (API) supported by Microsoft Windows.

What's New in the Platform SDK

Lists the latest additions to the Platform SDK.

Platform SDK Documentation Contents

Lists all technologies that are covered by the Platform SDK documentation.

Using the Platform SDK

Describes how to use the Platform SDK with Visual Studio.

Platform SDK Samples

Lists the samples that use the Platform SDK.

COM/ATL/MFC

COM (Component Object Model)

Introduces COM, a platform-independent, distributed, object-oriented system for creating binary software components that can interact.

COM+ (Component Services)

Describes how COM+ builds on and extends applications written using COM, MTS, and other COM-based technologies.

Compiler COM Support Samples

Lists the samples that demonstrate the Visual C++ compiler's built-in support for COM.

ATL Tutorial

Leads you through the creation of an ActiveX control, demonstrating many ATL and COM fundamentals.

Attributes Tutorial

Uses attributes to implement both a client and a server application.

Introduction to COM and ATL

Provides a brief introduction to COM and ATL.

Programming with ATL and C Run-Time Code

Discusses the benefits of using the C Run-Time Library (CRT) with either static or dynamic linking, or alternatively using the _ATL_MIN_CRT macro to reduce your dependency on CRT code.

ATL Samples

Lists the samples that demonstrate ATL.

ATL Server Samples

Lists the samples that demonstrate ATL Server.

MFC Overview

Introduces the Microsoft Foundation Class (MFC) library.

Hierarchy Chart

Graphically displays the MFC library.

MFC COM

Describes MFC's support for COM.

MFC Samples

Lists the samples that demonstrate MFC.

Native Programming

DLLs

Discusses Dynamic Link Libraries (DLLs).

DLL Frequently Asked Questions

Answers common questions concerning DLLs.

STL Samples

Lists the samples that demonstrate the use of the Standard Template Library.

Precompiled Headers

Creating Precompiled Header Files

Describes how the Microsoft C and C++ compilers provide options for precompiling any C or C++ code, including inline code.

When to Precompile Source Code

Describes when it is appropriate to use precompiled headers.

Two Choices for Precompiling Code

Describes the two choices for precompiling code: automatic precompiling and manual precompiling.

Precompiled Header Consistency Rules

Discusses guidelines that will help you use precompiled headers more efficiently.

Using Precompiled Headers in a Project

Describes a method for using the manual precompiled-header options in a project.

See Also

Concepts

How Do I in Visual C++

Interop (How Do I in Visual C++)

This page links to help on tasks related to interop using Visual C++. To view other categories of popular tasks covered in Help, see How Do I in Visual C++.

CLR/Native Interoperability

Mixed, Pure, and Verifiable Feature Comparison

Compares features across the three difference /clr compilation modes.

How To: Migrate to /clr

Discusses issues that arise when compiling native code with **/clr** and when converting a project to managed.

How to: Compile MFC and ATL Code with /clr

Discusses how to compile existing MFC and ATL programs to target the Common Language Runtime.

Mixed (Native and Managed) Assemblies

Describes how developers can author applications using a mixture of managed and unmanaged functionality.

Performance Considerations for Interop (C++)

Provides guidelines for reducing the effect of managed/unmanaged interop transitions on run-time performance.

C++ Interop

How to: Marshal ANSI Strings Using C++ Interop

Demonstrates using C++ Interop to marshal ANSI strings between managed and native code.

How to: Marshal Unicode Strings Using C++ Interop

Demonstrates using C++ Interop to marshal Unicode strings between managed and native code.

How to: Marshal COM Strings Using C++ Interop

Demonstrates using C++ Interop to marshal COM strings between managed and native code.

How to: Marshal Structures Using C++ Interop

Demonstrates using C++ Interop to marshal **struct**s between managed and native code.

How to: Marshal Arrays Using C++ Interop

Demonstrates using C++ Interop to marshal arrays between managed and native code.

How to: Marshal Callbacks and Delegates Using C++ Interop

Demonstrates using C++ Interop to marshal callbacks and delegates between managed and native code.

How to: Marshal Embedded Pointers Using C++ Interop

Demonstrates using C++ Interop to marshal embedded pointers between managed and native code.

COM Interop

Using Native COM Servers from .NET

Describes the available options for using existing COM components from .NET applications and outlines the advantages and disadvantages of each approach.

How to: Use Native COM Servers with TLBIMP

Demonstrates how COM objects can be used from managed code using the Type Library Importer (Tlbimp.exe) tool.

How to: Use Native COM Servers with CRCWs

Demonstrates how COM objects can be used from managed code using Custom Runtime Callable Wrappers.

Exposing .NET Framework Components to COM

Describes several tips for writing managed code that interoperates with COM clients.

P/Invoke

How to: Marshal Strings Using PInvoke

Explains how native functions that accept C-style strings can be called using the CLR string type System.String using Platform Invoke (P/Invoke) functionality.

How to: Call Native DLLs from Managed Code Using PInvoke

Demonstrates how functions that are implemented in unmanaged DLLs can be called from managed code using Platform Invoke (P/Invoke) functionality.

How to: Marshal Structures Using PInvoke

Explains how native functions that accept C-style **struct**s can be called from managed code using Platform Invoke (P/Invoke) functionality.

How to: Marshal Arrays Using PInvoke

Explains how native functions that accept C-style arrays can be called from managed code using Platform Invoke (P/Invoke) functionality.

How to: Marshal Function Pointers Using PInvoke

Explains how managed delegates can be used in place of function pointers when interoperating with unmanaged functions using Platform Invoke (P/Invoke) functionality.

How to: Marshal Embedded Pointers Using PInvoke

Explains how to marshal embedded pointers using Platform Invoke (P/Invoke) functionality.

Using Windows Forms in MFC

Using a Windows Form User Control in MFC

Explains how you can use the MFC Windows Forms support classes to host Windows Forms controls within your MFC applications as ActiveX controls within MFC dialog boxes or views.

Windows Forms/MFC Programming Differences

Provides background information about programming differences between Windows Forms and MFC.

Hosting a Windows Form User Control as an MFC Dialog Box

Describes how to host a Windows Forms Control as an MFC Dialog Box using the **CWinFormsDialog** class.

How to: Create the User Control and Host MDI View

Shows how to create a .NET Frameworks user control, author the user control in a control class library (specifically, a Windows Control Library project), and then compile the project into an assembly.

How to: Add Command Routing to the Windows Forms Control

Demonstrates how to use **CWinFormsView** to route commands and update command UI messages to the user control to allow it to handle MFC commands.

How to: Call Properties and Methods of the Windows Forms Control

Explains why it is advisable to add a member of the user control type and initialize it in **IView::OnInitialUpdate**.

Hosting a Windows Form User Control as an MFC View

Describes how to host a Windows Forms User Control as an MFC View.

How to: Create the User Control and Host in a Dialog Box

Explains how to add a user control to a new dialog-based MFC project.

How to: Do DDX/DDV Data Binding with Windows Forms

Shows how to bind a native C++ string to a .NET user control.

How to: Sink Windows Forms Events from Native C++ Classes

Shows how to enable native C++ classes to receive callbacks from managed events raised from Windows Forms controls or other forms with the MFC macro map format. Sinking events in views and dialogs is similar to doing the same task for controls.

Hosting a Windows Form User Control in an MFC Dialog Box

Describes how to host a Windows Forms user control in an MFC dialog box.

See Also

Concepts

How Do I in Visual C++

Migrating, Porting, and Upgrading (How Do I in Visual C++)

This page links to help on tasks related to migrating, porting, and upgrading using Visual C++. To view other categories of popular tasks covered in Help, see How Do I in Visual C++.

Migrating to Visual C++

How To: Migrate to /clr

Discusses issues that arise when compiling native code with **/clr** and when converting a project to managed.

How to: Compile MFC and ATL Code with /clr

Discusses how to compile existing MFC and ATL programs to target the Common Language Runtime.

Managed Extensions for C++ Syntax Upgrade Checklist

Lists the syntactic differences between Managed Extensions for C++ and the new C++ syntax.

C++/CLI Migration Primer

A detailed guide to migrating from Managed Extensions for C++ to Visual C++ 2005.

How to: Migrate to /clr:pure

Discusses issues likely to arise when migrating to pure MSIL using the /clr:pure compiler option.

How to: Migrate to /clr:safe

Discusses issues likely to arise when migrating to verifiable MSIL using the /clr:safe compiler option.

Migrating from UNIX

Introduction to Visual C++ for UNIX Users

Provides information for UNIX users who are new to Visual C++ and want to become productive with Visual C++.

Porting from UNIX to Win32

Describes the options available when migrating applications from UNIX to Windows.

Porting to 64-bit

64-Bit Programming with Visual C++

Contains topics about targeting 64-bit hardware with the Visual C++ build tools.

Common Visual C++ 64-bit Migration Issues

Lists many of the issues you will likely encounter when creating applications to run on a 64-bit Windows operating system.

Visual Studio Development Environment 64-Bit Support

Describes support for creating 64-bit applications using Visual Studio.

Upgrading from Previous Versions of Visual C++

Breaking Changes in the Visual C++ 2005 Compiler

Discusses the behavior changes in Visual C++ 2005 that can cause code that worked in a previous release to either not compile, or to behave differently at run time.

What's New in Visual C++

Lists the new additions to

Visual C++ 2005.

What's New in the Visual C++ Libraries

Discusses the updates and breaking changes to the libraries that are shipped with Visual C++.

Breaking Changes in Visual C++ .NET 2003

Discusses the behavior changes in Visual C++.NET 2003 that can cause code that worked in a previous release to either not compile, or to behave differently at run time.

Visual C++ .NET 2003 Enhanced Compiler Conformance

Lists areas where the conformance of the Visual C++ compiler was significantly improved.

Product Changes: Visual C++ .NET 2002

Discusses the behavior changes in Visual C++.NET 2002 that can cause code that worked in a previous release to either not compile, or to behave differently at run time.

See Also

Concepts

How Do I in Visual C++

Security (How Do I in Visual C++)

This page links to help on tasks related to security using Visual C++. To view other categories of popular tasks covered in Help, see How Do I in Visual C++.

Security Basics

Security Best Practices for C++

Contains information on recommended security tools and practices that is specific to Visual C++.

Security in Native and .NET Framework Code

Contains information on recommended security tools and practices.

/analyze (Enterprise Code Analysis)

C/C++ Code Analysis tool provides information to developers about possible defects in their C/C++ source code.

Code Access Security

Code Access Security is a feature of the .NET Common Language Runtime (CLR) that helps limit the access code has to protected resources and operations.

How to: Debug a Partial Trust Application

Describes how to debug a partial trust application using Visual Studio.

How to: Create a Partially Trusted Application by Removing Dependency on msvcm80.dll

Discusses how to create a partially trusted Common Language Runtime application using Visual C++ by removing dependency on msvcm80.dll.

Secure C Runtime (CRT)

Security Enhancements in the CRT

Describes the significant enhancements that have been made to make the CRT more secure.

Security-Enhanced Versions of CRT Functions

Lists all security enhanced functions available in Visual C++ 2005.

Parameter Validation

Describes how the security enhanced functions handle parameter validation.

Secure Template Overloads

Describes the template overloads that the CRT provides to help ease the transition to the more secure variants.

Standard C++ Library Security Improvements

Checked Iterators

Describes how checked iterators ensure that you do not overwrite the bounds of your container in retail code.

Debug Iterator Support

Describes how debug iterators ensure that you do not overwrite the bounds of your container in debug code.

See Also

Concepts

How Do I in Visual C++

64-bit Programming (How Do I in Visual C++)

This page links to help on tasks related to 64-bit programming using Visual C++. To view other categories of popular tasks covered in Help, see How Do I in Visual C++.

64-bit Programming

64-Bit Programming with Visual C++

Contains topics about targeting 64-bit hardware with the Visual C++ build tools.

How to: Enable a 64-Bit Visual C++ Toolset at the Command Line

Describes the various versions of cl.exe (the Visual C++ compiler) that can be used to create x86 or x64 applications.

Common Visual C++ 64-bit Migration Issues

Lists issues you will likely encounter when creating applications to run on a 64-bit Windows operating system.

How to: Configure Visual C++ Projects to Target 64-Bit Platforms

Describes how to set up Visual C++ applications to target 64-bit platforms using project configurations available in the Visual Studio Integrated Development Environment (IDE).

See Also

Concepts

How Do I in Visual C++

Command Line (How Do I in Visual C++)

This page links to help on tasks related to command line development using Visual C++. To view other categories of popular tasks covered in Help, see How Do I in Visual C++.

Visual C++ on the Command Line

How to: Compile a Native C++ Program from the Command Line

Demonstrates how to create a simple Visual C++ program with a text editor and compile it on the command line.

How to: Compile a Code Example from the Help Topics

Describes how to locate Visual C++ code example programs in the help topics and how to compile and run them.

Compiler Options

Introduces cl.exe, a tool that controls the Microsoft C and C++ compilers and linker.

Linker Options

Introduces LINK, a tool that links Common Object File Format (COFF) object files and libraries to create an executable (.exe) file or dynamic-link library (DLL).

NMAKE Reference

Introduces the Microsoft Program Maintenance Utility (NMAKE.EXE), a tool that builds projects based on commands contained in a description file.

New NMAKE Features

Lists the new NMAKE features in Visual C++ 2005.

VCBUILD Reference

Describes how you can use VCBUILD.exe to build Visual C++ projects and Visual Studio solutions from the command line.

How to: Run Multiprocessor Builds with VCBUILD

Describes how you can use VCBUILD to run Multiprocessor Builds from the command line.

Introduction to Visual C++ for UNIX Users

Provides information for UNIX users who are new to Visual C++ and want to become productive with Visual C++.

Visual C++ Examples

Arrays

How to: Create Single-Dimension Arrays

Shows how to create single-dimension arrays of reference, value, and native pointer types.

How to: Create Multidimension Arrays

Shows how to create multi-dimension arrays of reference, value, and native pointer types.

How to: Iterate Over Arrays with for each

Shows how to use the for each, in keyword on different types of arrays.

How to: Create Arrays of Managed Arrays (Jagged Arrays)

Shows how to create single-dimension arrays of managed array of reference, value, and native pointer types.

How to: Sort Arrays

Demonstrates how to use the **Sort** method to sort the elements of an array.

How to: Sort Arrays Using Custom Criteria

Demonstrates how to sort arrays by implementing the IComparable interface.

How to: Make Typedefs for Managed Arrays

Shows how to make a typedef for a managed array.

How to: Accept Variable Arguments

Shows how functions with a variable number of arguments can be implemented in Visual C++ using the **...** syntax.

How to: Use Managed Arrays as Template Type Parameters

Shows how to use a managed array as a parameter to a template.

How to: Declare and Use Interior Pointers and Managed Arrays

Shows how you can declare and use an interior pointer to an array.

Boxing and Casting

How to: Use gcnew to Create Value Types and Use Implicit Boxing

Shows how to use gcnew on a value type to create a boxed value type, which can then be placed on the managed, garbage-collected heap.

How to: Unbox

Shows how to unbox and modify a value.

How to: Explicitly Request Boxing

Shows how to explicitly request boxing by assigning a variable to a variable of type Object.

How to: Downcast with safe_cast

Shows how to downcast from a base class to a class derived from the base class using **safe_cast**.

How to: Use safe_cast and Boxing

Shows how to use **safe_cast** to box a value on the CLR heap.

How to: Use safe_cast and Generic Types

Shows how to use **safe_cast** to perform a downcast with a generic type.

How to: Use safe_cast and Unboxing

Shows how to use **safe_cast** to unbox a value on the CLR heap.

How to: Use safe_cast and User-Defined Conversions

Shows how to invoke user-defined conversions by using **safe_cast**.

How to: Upcast with safe_cast

Shows how to do an upcast—a cast from a derived type to one of its base classes—using **safe_cast**.

Data Types and Interfaces

How to: Instantiate Classes and Structs

Demonstrates that reference types and value types can only be instantiated on the managed heap, not on the stack or on the native heap.

How to: Convert with Constructors

Introduces converting constructors, constructors that take a type and use it to create an object.

How to: Define an Interface Static Constructor

Introduces static constructors, constructors which can be used to initialize static data members.

How to: Define Static Constructors in a Class or Struct

Demonstrates how to create a static constructor.

How to: Write Template Functions that Take Native, Value, or Reference Parameters

Demonstrates that by using a tracking reference in the signature of a template function, you can ensure that the function can be called with parameters whose type are native, CLR value, or CLR reference.

Enumerations

How to: Specify Underlying Types of Enums

Shows how to specify the underlying type of an **enum**.

How to: Convert Between Managed and Standard Enumerations

Demonstrates how to convert between an **enum** and an integral type by using a cast.

Events and Delegates

How to: Compose Delegates

Demonstrates how to compose delegates.

How to: Define and Use Delegates

Demonstrates how to define and use a delegate.

How to: Define and Use Static Events

Shows how to define and use static events.

How to: Define Event Accessor Methods

Shows how you can define an event's behavior when handlers are added or removed, and for when an event is raised.

How to: Implement Abstract Events

Shows how to implement an abstract event.

How to: Implement Managed Virtual Events

Shows how to implement virtual, managed events in an interface and class.

How to: Access Events in Interfaces

Shows how to access an event in an interface.

How to: Add Multiple Handlers to Events

Demonstrates that an event receiver, or any other client code, can add one or more handlers to an event.

How to: Associate Delegates to Members of a Value Class

Shows how to associate a delegate with a member of a value class.

How to: Associate Delegates to Unmanaged Functions

Shows how to associate a delegate with a native function by wrapping the native function in a managed type, and declaring the function to be invoked through P/Invoke.

How to: Override Default Access of add, remove, and raise Methods

Shows how to override the default access on the add, remove, and raise events methods.

How to: Raise Events Defined in a Different Assembly

Shows how to consume an event and event handler defined in one assembly by another assembly.

Exceptions

Basic Concepts in Using Managed Exceptions

Discusses the basic concepts for exception handling in managed applications.

Differences in Exception Handling Behavior Under /CLR

Discusses differences from the standard behavior of exception handling and some restrictions in detail.

How to: Define and Install a Global Exception Handler

Demonstrates how unhandled exceptions can be captured.

How to: Catch Exceptions in Native Code Thrown from MSIL

Shows how to catch CLR exceptions in native code with **__try** and **__except**.

finally

Discusses the CLR exception handling **finally** clause.

For Each

How to: Iterate Over Arrays with for each

Shows how to use the for each, in keyword on different types of arrays.

How to: Iterate Over a Generic Collection with for each

Demonstrates how to create generic collections and iterate over them using for each, in.

How to: Iterate Over a User-Defined Collection with for each

Demonstrates how to iterate over a user-defined collection using for each, in.

How to: Iterate Over STL Collection with for each

Demonstrates how to iterate over STL collections using for each, in.

Generics

Overview of Generics in Visual C++

Provides an overview of generics, parameterized types supported by the Common Language Runtime.

Generic Functions

Discusses generic functions, a function that is declared with type parameters.

Generic Classes (Visual C++)

Describes how to create a generic class.

Generic Interfaces (Visual C++)

Describes how to create a generic interface.

Generic Delegates (Visual C++)

Describes how to create a generic delegate.

Constraints

Describes that constraints are a requirement that types used as type arguments must satisfy.

Consuming Generics from Other .NET Languages

Discusses how generics authored in one .NET language may be used in other .NET languages.

Generics and Templates

Provides an overview of the many differences between generics and templates.

How to: Convert Generic Classes

Shows how to convert a generic class to T.

Pointers

How to: Declare Interior Pointers with the const Keyword

Shows how to use **const** in the declaration of an interior pointer.

How to: Overload Functions with Interior Pointers and Native Pointers

Demonstrates that functions can be overloaded depending on whether the parameter type is an interior pointer or a native pointer.

How to: Cannot Use Tracking References and Unary "Take-Address" Operator

Shows that a tracking reference cannot be used as a unary take-address operator.

How to: Declare Pinning Pointers and Value Types

Shows that you can declare a pinning pointer to a value type object and use a **pin_ptr** to the boxed value type.

How to: Declare Value Types with the interior_ptr Keyword

Demonstrates that an **interior_ptr** can be used with a value type.

How to: Define the Scope of Pinning Pointers

Demonstrates that an object is pinned only while a **pin_ptr** points to it.

How to: Pin Pointers and Arrays

Shows how to pin an array by declaring a pinning pointer to its element type, and pinning one of its elements.

Properties

How to: Use Simple Properties

Demonstrates that for simple properties—those that merely assign and retrieve a private data member—it is not necessary to explicitly define the get and set accessor functions.

How to: Use Indexed Properties

Shows how to use default and user defined indexed properties.

How to: Use Multidimensional Properties

Shows how to create multidimension properties that take a non-standard number of parameters.

How to: Declare and Use Static Properties

Shows how to declare and use a static property.

How to: Declare and Use Virtual Properties

Shows how to declare and use virtual properties.

How to: Declare Abstract and Sealed Properties

Shows how to declare a sealed or abstract property by defining a non-trivial property and specifying the abstract or sealed keywords on the get and set accessor functions.

How to: Overload Property Accessor Methods

Demonstrates how to overload indexed properties.

Tracking References

How to: Use Tracking References and Value Types

Shows simple boxing through a tracking reference to a value type.

How to: Using Tracking References and Interior Pointers

Shows that taking the address of a tracking reference returns an **interior_ptr** and how to modify and access data through a tracking reference.

How to: Pass CLR Types by Reference with Tracking References

Shows how to pass CLR types by reference with tracking references.

File Handling with Visual C++

How to: Read a Binary File

Demonstrates reading binary data from a file.

How to: Write a Binary File

Demonstrates writing binary data to a file.

How to: Read a Text File

Demonstrates how to open and read a text file one line at a time.

How to: Write a Text File

Demonstrates how to create a text file and write text to it using the StreamWriter class.

How to: Enumerate Files in a Directory

Demonstrates how to retrieve a list of the files in a directory.

How to: Monitor File System Changes

Uses FileSystemWatcher to register for events corresponding to files being created, changed, deleted, or renamed.

How to: Retrieve File Information

Demonstrates the FileInfo class. When you have the name of a file, you can use this class to retrieve information about the file such as the file size, directory, full name, and date and time of creation and of the last modification.

Windows Operations with Visual C++

How to: Write Data to the Windows Registry

Uses the CurrentUser key to create a writable instance of the RegistryKey class.

How to: Read Data from the Windows Registry

Uses the **CurrentUser** key to read data from the Windows registry.

How to: Retrieve Text from the Clipboard

Uses the GetDataObject member function to return a pointer to the IDataObject interface, which can then be queried for the format of the data and used to retrieve the actual data.

How to: Store Text in the Clipboard

Uses the Clipboard object defined in the System.Windows.Forms namespace to store a string.

How to: Retrieve the Windows Version

Demonstrates how to retrieve the platform and version information of the current operating system.

How to: Retrieve Time Elapsed Since Startup

Demonstrates how to determine the tick count, or milliseconds that have elapsed since Windows was started.

See Also

Concepts

How Do I in Visual C++

Data Access (How Do I in Visual C++)

This page links to help on tasks related to data access using Visual C++. To view other categories of popular tasks covered in Help, see How Do I in Visual C++.

Programming

Data Access Programming

Describes the several ways to work with databases.

Installing Database Support

Describes the database components that are installed automatically with Visual C++.

Record (Data Access)

Describes a record, a collection of data about a single entity, such as an account or a customer, stored in a table (a row of the table).

Schema (Data Access)

Describes how a database schema describes the current structure of the tables and database views in the database.

Catalog Information

Describes what information can be stored in a table in a data source, such as the names of tables and the columns in them, table privileges, names of primary and foreign keys, information about predefined queries or stored procedures, information about indexes on tables, and statistics about tables.

Transactions

Describes transactions, a series of operations performed on a database.

Record Views

Describes the classes that MFC provides to access record views.

Data Access Frequently Asked Questions

Provides answers to the most frequently asked questions about data access.

ADO.NET

Data Access Using ADO.NET in C++

Provides an overview of using ADO.NET in Visual C++.

How to: Marshal ANSI Strings for ADO.NET

Demonstrates how to add a native string (**char ***) to an ADO.NET database and how to marshal a System.String from an ADO.NET database to a native string.

How to: Marshal BSTR Strings for ADO.NET

Demonstrates how to add a COM string (**BSTR**) to an ADO.NET database and how to marshal a **System.String** from an ADO.NET database to a COM string.

How to: Marshal Unicode Strings for ADO.NET

Demonstrates how to add a Unicode string (**wchar_t ***) to an ADO.NET database and how to marshal a **System.String** from an ADO.NET database to a Unicode string.

How to: Marshal a VARIANT for ADO.NET

Demonstrates how to add a **VARIANT** to an ADO.NET database and how to marshal a System.Object from an ADO.NET database to a **VARIANT**.

How to: Marshal a SAFEARRAY for ADO.NET

Demonstrates how to add a **SAFEARRAY** to an ADO.NET database and how to marshal an array from an ADO.NET database to a **SAFEARRAY**.

ADO.NET Sample Application

Demonstrates how to return results from a data source and write the output to the console or command prompt.

ADO

Data-Bound Controls (ADO and RDO)

Describes databinding, a mechanism that allows users to work with data retrieved from a database.

Data Access: ADO and RDO

Summarizes the two underlying technologies that support data-source or data-bound controls.

ADO Databinding

Discusses the use of the databinding ActiveX controls that are built with ADO.

Using ADO Databinding in Visual C++

Describes the steps needed to use ADO databinding in Visual C++.

ODBC

Open Database Connectivity (ODBC)

Provides an overview of the classes that the Microsoft Foundation Classes (MFC) library supplies for programming with Open Database Connectivity (ODBC).

Installing and Getting Started with ODBC

Helps you install the components needed to start working with Open Database Connectivity (ODBC).

Design and Create an ODBC Database Application

Helps you design and set up an ODBC database application.

ODBC Basics

Provides the basics of Open Database Connectivity (ODBC).

ODBC and MFC

Introduces the main concepts of the Microsoft Foundation Classes (MFC) library's ODBC-based database classes and provides an overview of how the classes work together.

OLE

OLE DB Programming

Represents the starting point for information about the Microsoft OLE DB database technology and the OLE DB Template Library.

OLE DB Programming Overview

Describes what OLE DB is, and what makes it distinct from other database technologies.

OLE DB Consumer Templates (C++)

Lists the OLE DB Consumer Templates that the OLE DB version 2.6 specification supports.

OLE DB Provider Templates (C++)

Describes the OLE DB Provider and describes what services they must support.

ATL Database Classes (OLE DB Templates)

Describes the templates in ATL that make the high-performance OLE DB database technology easier to use by providing classes that implement many of the commonly used OLE DB interfaces.

See Also

Concepts

How Do I in Visual C++

Other Resources

ADO.NET

Debugging, Testing, and Deployment (How Do I in Visual C++)

This page links to help on tasks related to debugging, testing, and deployment using Visual C++. To view other categories of popular tasks covered in Help, see How Do I in Visual C++.

Debugging

Walkthrough: Debugging an XML Web Service

Shows how you debug an XML Web service.

Debugging Preparation: Windows Forms Applications

Discusses how to debug a Windows Forms application.

Making an Image Easier to Debug

Describes how to simplify debugging of managed code.

Testing

Runtime Profiling

Discusses ways of gathering information about application performance.

Debugging and Profiling Applications

Discusses techniques for debugging and profiling applications using various tools available to developers.

Enabling Profiling

Describes how to enable profiling.

Deployment

Deployment (C++)

Discusses the two different technologies for deploying Windows applications: ClickOnce deployment or Windows Installer deployment.

Deployment Concepts

Discusses the main considerations for deploying Visual C++ applications.

Deployment Examples

Contains examples demonstrating how to deploy Visual C++ applications built with Visual Studio.

Version Checking of Redistributable Files

Demonstrates one method of checking between two dynamic libraries or executables.

ClickOnce Deployment for Visual C++ Applications

Describes the tools available for using ClickOnce with Visual C++.

Running a C++ /clr Application on a Previous Runtime Version

Describes how to make your application run on a different runtime than it was built with.

See Also

Concepts

How Do I in Visual C++

Performance (How Do I in Visual C++)

This page links to help on tasks related to performance using Visual C++. To view other categories of popular tasks covered in Help, see How Do I in Visual C++.

Performance

Profile-Guided Optimizations

Provides an overview of the process of using profile-guided optimizations.

Walkthrough: Using Profile-Guided Optimizations

Demonstrates how to use profile-guided optimization (PGO) for varied scenarios that range from simple implementations using PGO build menus in the Visual Studio Integrated Development Environment (IDE), to more advanced scenarios that require custom configurations.

Tools for Profile-Guided Optimization

Discusses the tools that support profile-guided optimizations.

How to: Merge Multiple PGO Profiles into a Single Profile

Describes how to create a single profile that PGO can use from several different scenarios.

Environment Variables for Profile-Guided Optimizations

Contains information about environment variables that affect test scenarios on an image created with **/LTCG:PGI**.

/PGD (Specify Database for Profile-Guided Optimizations)

Discusses the **/PGD** command line option.

/O Options (Optimize Code)

Discusses the /O option, which lets you control various optimizations to create code for maximum speed or minimum size.

OpenMP in Visual C++

Describes how the OpenMP API lets you write applications that effectively use multiple processors.

See Also

Concepts

How Do I in Visual C++

C++ Language (How Do I in Visual C++)

This page links to help on tasks related to the Visual C++ language using Visual C++. To view other categories of popular tasks covered in Help, see How Do I in Visual C++.

Differences from Managed Extensions for C++

Managed Extensions for C++ Syntax Upgrade Checklist

Lists the syntactic differences between Managed Extensions for C++ and the new C++ syntax.

C++/CLI Migration Primer

A detailed guide to migrating code from Managed Extensions for C++ to Visual C++ 2005.

Visual C++ Examples

Arrays

How to: Create Single-Dimension Arrays

Shows how to create single-dimension arrays of reference, value, and native pointer types.

How to: Create Multidimension Arrays

Shows how to create multi-dimension arrays of reference, value, and native pointer types.

How to: Iterate Over Arrays with for each

Shows how to use the for each, in keyword on different types of arrays.

How to: Create Arrays of Managed Arrays (Jagged Arrays)

Shows how to create single-dimension arrays of managed array of reference, value, and native pointer types.

How to: Sort Arrays

Demonstrates how to use the **Sort** method to sort the elements of an array.

How to: Sort Arrays Using Custom Criteria

Demonstrates how to sort arrays by implementing the IComparable interface.

How to: Make Typedefs for Managed Arrays

Shows how to make a typedef for a managed array.

How to: Accept Variable Arguments

Shows how functions with a variable number of arguments can be implemented in Visual C++ using the **...** syntax.

How to: Use Managed Arrays as Template Type Parameters

Shows how to use a managed array as a parameter to a template.

How to: Declare and Use Interior Pointers and Managed Arrays

Shows how you can declare and use an interior pointer to an array.

Boxing and Casting

How to: Use gcnew to Create Value Types and Use Implicit Boxing

Shows how to use gcnew on a value type to create a boxed value type, which can then be placed on the managed, garbage-collected heap.

How to: Unbox

Shows how to unbox and modify a value.

How to: Explicitly Request Boxing

Shows how to explicitly request boxing by assigning a variable to a variable of type Object.

How to: Downcast with safe_cast

Shows how to downcast from a base class to a class derived from the base class using **safe_cast**.

How to: Use safe_cast and Boxing

Shows how to use **safe_cast** to box a value on the CLR heap.

How to: Use safe_cast and Generic Types

Shows how to use **safe_cast** to perform a downcast with a generic type.

How to: Use safe_cast and Unboxing

Shows how to use **safe_cast** to unbox a value on the CLR heap.

How to: Use safe_cast and User-Defined Conversions

Shows how to invoke user-defined conversions by using **safe_cast**.

How to: Upcast with safe_cast

Shows how to do an upcast—a cast from a derived type to one of its base classes—using **safe_cast**.

Data Types and Interfaces

How to: Instantiate Classes and Structs

Demonstrates that reference types and value types can only be instantiated on the managed heap, not on the stack or on the native heap.

How to: Convert with Constructors

Introduces converting constructors, constructors that take a type and use it to create an object.

How to: Define an Interface Static Constructor

Introduces static constructors, constructors which can be used to initialize static data members.

How to: Define Static Constructors in a Class or Struct

Demonstrates how to create a static constructor.

How to: Write Template Functions that Take Native, Value, or Reference Parameters

Demonstrates that by using a tracking reference in the signature of a template function, you can ensure that the function can be called with parameters whose type are native, CLR value, or CLR reference.

Enumerations

How to: Specify Underlying Types of Enums

Shows how to specify the underlying type of an **enum**.

How to: Convert Between Managed and Standard Enumerations

Demonstrates how to convert between an **enum** and an integral type by using a cast.

Events and Delegates

How to: Compose Delegates

Demonstrates how to compose delegates.

How to: Define and Use Delegates

Demonstrates how to define and use a delegate.

How to: Define and Use Static Events

Shows how to define and use static events.

How to: Define Event Accessor Methods

Shows how you can define an event's behavior when handlers are added or removed, and for when an event is raised.

How to: Implement Abstract Events

Shows how to implement an abstract event.

How to: Implement Managed Virtual Events

Shows how to implement virtual, managed events in an interface and class.

How to: Access Events in Interfaces

Shows how to access an event in an interface.

How to: Add Multiple Handlers to Events

Demonstrates that an event receiver, or any other client code, can add one or more handlers to an event.

How to: Associate Delegates to Members of a Value Class

Shows how to associate a delegate with a member of a value class.

How to: Associate Delegates to Unmanaged Functions

Shows how to associate a delegate with a native function by wrapping the native function in a managed type, and declaring the function to be invoked through P/Invoke.

How to: Override Default Access of add, remove, and raise Methods

Shows how to override the default access on the add, remove, and raise events methods.

How to: Raise Events Defined in a Different Assembly

Shows how to consume an event and event handler defined in one assembly by another assembly.

Exceptions

Basic Concepts in Using Managed Exceptions

Discusses the basic concepts for exception handling in managed applications.

Differences in Exception Handling Behavior Under /CLR

Discusses differences from the standard behavior of exception handling and some restrictions in detail.

How to: Define and Install a Global Exception Handler

Demonstrates how unhandled exceptions can be captured.

How to: Catch Exceptions in Native Code Thrown from MSIL

Shows how to catch CLR exceptions in native code with **__try** and **__except**.

finally

Discusses the CLR exception handling **finally** clause.

Basic Concepts in Using Managed Exceptions

Discusses the basic concepts for exception handling in managed applications.

For Each

How to: Iterate Over Arrays with for each

Shows how to use the for each, in keyword on different types of arrays.

How to: Iterate Over a Generic Collection with for each

Demonstrates how to create generic collections and iterate over them using for each, in.

How to: Iterate Over a User-Defined Collection with for each

Demonstrates how to iterate over a user-defined collection using for each, in.

How to: Iterate Over STL Collection with for each

Demonstrates how to iterate over STL collections using for each, in.

Generics

Overview of Generics in Visual C++

Provides an overview of generics, parameterized types supported by the Common Language Runtime.

Generic Functions

Discusses generic functions, a function that is declared with type parameters.

Generic Classes (Visual C++)

Describes how to create a generic class.

Generic Interfaces (Visual C++)

Describes how to create a generic interface.

Generic Delegates (Visual C++)

Describes how to create a generic delegate.

Constraints

Describes that constraints are a requirement that types used as type arguments must satisfy.

Consuming Generics from Other .NET Languages

Discusses how generics authored in one .NET language may be used in other .NET languages.

Generics and Templates

Provides an overview of the many differences between generics and templates.

How to: Convert Generic Classes

Shows how to convert a generic class to T.

Pointers

How to: Declare Interior Pointers with the const Keyword

Shows how to use **const** in the declaration of an interior pointer.

How to: Overload Functions with Interior Pointers and Native Pointers

Demonstrates that functions can be overloaded depending on whether the parameter type is an interior pointer or a native pointer.

How to: Cannot Use Tracking References and Unary "Take-Address" Operator

Shows that a tracking reference cannot be used as a unary take-address operator.

How to: Declare Pinning Pointers and Value Types

Shows that you can declare a pinning pointer to a value type object and use a **pin_ptr** to the boxed value type.

How to: Declare Value Types with the interior_ptr Keyword

Demonstrates that an **interior_ptr** can be used with a value type.

How to: Define the Scope of Pinning Pointers

Demonstrates that an object is pinned only while a **pin_ptr** points to it.

How to: Pin Pointers and Arrays

Shows how to pin an array by declaring a pinning pointer to its element type, and pinning one of its elements.

Properties

How to: Use Simple Properties

Demonstrates that for simple properties—those that merely assign and retrieve a private data member—it is not necessary to explicitly define the get and set accessor functions.

How to: Use Indexed Properties

Shows how to use default and user defined indexed properties.

How to: Use Multidimensional Properties

Shows how to create multidimension properties that take a non-standard number of parameters.

How to: Declare and Use Static Properties

Shows how to declare and use a static property.

How to: Declare and Use Virtual Properties

Shows how to declare and use virtual properties.

How to: Declare Abstract and Sealed Properties

Shows how to declare a sealed or abstract property by defining a non-trivial property and specifying the abstract or sealed keywords on the get and set accessor functions.

How to: Overload Property Accessor Methods

Demonstrates how to overload indexed properties.

Tracking References

How to: Use Tracking References and Value Types

Shows simple boxing through a tracking reference to a value type.

How to: Using Tracking References and Interior Pointers

Shows that taking the address of a tracking reference returns an **interior_ptr** and how to modify and access data through a tracking reference.

How to: Pass CLR Types by Reference with Tracking References

Shows how to pass CLR types by reference with tracking references.

File Handling with Visual C++

How to: Read a Binary File

Demonstrates reading binary data from a file.

How to: Write a Binary File

Demonstrates writing binary data to a file.

How to: Read a Text File

Demonstrates how to open and read a text file one line at a time.

How to: Write a Text File

Demonstrates how to create a text file and write text to it using the StreamWriter class.

How to: Enumerate Files in a Directory

Demonstrates how to retrieve a list of the files in a directory.

How to: Monitor File System Changes

Uses FileSystemWatcher to register for events corresponding to files being created, changed, deleted, or renamed.

How to: Retrieve File Information

Demonstrates the FileInfo class. When you have the name of a file, you can use this class to retrieve information about the file such as the file size, directory, full name, and date and time of creation and of the last modification.

Windows Operations with Visual C++

How to: Write Data to the Windows Registry

Uses the CurrentUser key to create a writable instance of the RegistryKey class.

How to: Read Data from the Windows Registry

Uses the **CurrentUser** key to read data from the Windows registry.

How to: Retrieve Text from the Clipboard

Uses the GetDataObject member function to return a pointer to the IDataObject interface, which can then be queried for the format of the data and used to retrieve the actual data.

How to: Store Text in the Clipboard

Uses the Clipboard object defined in the System.Windows.Forms namespace to store a string.

How to: Retrieve the Windows Version

Demonstrates how to retrieve the platform and version information of the current operating system.

See Also

Concepts

How Do I in Visual C++

Reference (How Do I in Visual C++)

This page links to reference topics for the Visual C++ language and its libraries. To view other categories of popular tasks covered in Help, see How Do I in Visual C++.

C/C++ Languages
C Language Reference

Describes the C programming language as implemented in Microsoft C.

C++ Language Reference

Explains the C++ programming language as implemented in Microsoft C++.

C/C++ Preprocessor Reference

Explains the preprocessor as it is implemented in Microsoft C/C++.

CRT
C Run-Time Libraries

Discusses the various .lib files that comprise the C run-time libraries as well as their associated compiler options and preprocessor directives.

Standard C++ Library
Standard C++ Library Reference

Discusses the ways a C++ program can call on a large number of functions from this conforming implementation of the Standard C++ Library. These functions perform essential services such as input and output and provide efficient implementations of frequently used operations.

.NET Framework
.NET Framework Programming in Visual Studio

Discusses application development with the .NET Framework in Visual Basic, Visual C#, Visual J#, or Visual C++.

.NET Framework Class Library Reference

Introduces the .NET Framework class library, a library of classes, interfaces, and value types that are included in the .NET Framework.

ATL
ATL Reference

Documents the Active Template Library (ATL), a set of template-based Visual C++ classes that simplify the programming of Component Object Model (COM) objects.

ATL/MFC Shared Classes

Documents the classes that are shared between ATL and MFC.

ATL Server
ATL Server Reference

Documents ATL Server, a set of native C++ classes that allow developers to create Web applications, XML Web services, and other server applications.

MFC
MFC Reference

Covers the classes, global functions, global variables, and macros that make up the Microsoft Foundation Class Library.

See Also
Concepts
How Do I in Visual C++

Programming Guide

This section contains conceptual and task-based topics on C++ programming.

In This Section

.NET Programming in C++

Covers .NET features available through the C++/CLI syntax, including the creation of pure MSIL and verifiable assemblies

Native and .NET Interoperability

Covers techniques for interfacing with non .NET Framework functionality, including options available only to C++.

General Concepts

Covers C++ topics such as data access, Exception handling, and multithreading.

64-Bit Programming with Visual C++

Describes the various 64-bit toolsets available to C++ programmers, and provides guidance for common migration issues.

See Also

Other Resources

Visual C++

.NET Programming in C++

The following topics describe how Visual C++ can target the .NET platform.

In This Section

Pure and Verifiable Code

Regular Expressions in C++

File Handling and I/O in C++

Graphics Operations Using the .NET Framework in C++

Windows Operations in C++

Data Access Using ADO.NET in C++

Interoperability with Other .NET Languages in C++

Serialization in C++

Managed Types

Reflection in C++

Strong Name Assemblies (Assembly Signing)

The Debug Class in Visual C++

See Also

Other Resources

Native and .NET Interoperability

Pure and Verifiable Code

For .NET Programming, Visual C++ supports the creation of three distinct types of components and applications: mixed, pure, and verifiable. All three are available through the /clr (Common Language Runtime Compilation) compiler option.

Remarks

For more information about verifiable assemblies, see:

- Mixed, Pure, and Verifiable Feature Comparison
- How to: Migrate to /clr:pure
- How to: Create Verifiable C++ Projects
- How to: Migrate to /clr:safe
- Using Verifiable Assemblies with SQL Server
- Security Best Practices for C++

Mixed (/clr)

Mixed assemblies (compiled with **/clr**), contain both unmanaged and managed parts, making it possible for them to use .NET features, but still contain unmanaged code. This allows applications and components to be updated to use .NET features without requiring that the entire project be rewritten. Using Visual C++ to mix managed and unmanaged code in this fashion is called C++ Interop. For more information, see Mixed (Native and Managed) Assemblies and Native and .NET Interoperability.

Pure (/clr:pure)

Pure assemblies (compiled with **/clr:pure**) can contain both native and managed data types, but only managed functions. Like mixed assemblies, pure assemblies allow interop with native DLLs through P/Invoke (see Using Explicit PInvoke in C++ (DllImport Attribute)), but C++ Interop features are not available. Moreover, pure assemblies cannot export functions that are callable from native functions because entry points in a pure assembly use the __clrcall calling convention.

Advantages of /clr:pure

- Better Performance: Because pure assemblies contain only MSIL, there are no native functions, and therefore no managed/unmanaged transitions are necessary. (Function calls made through P/Invoke are an exception to this rule.)
- AppDomain Awareness: Managed functions and CLR data types exist inside Application Domains, which affects their visibility and accessibility. Pure assemblies are domain-aware (__declspec(appdomain) is implied for each type) so accessing their types and functionality from other .NET components is easier and safer. As a result, pure assemblies interoperate more easily with other .NET components than mixed assemblies.
- Non-disk loading: Pure assemblies can be loaded in-memory and even streamed. This is essential for using .NET assemblies as stored procedures. This differs from mixed assemblies, which due to a dependency on the Windows loading mechanisms, must exist on disk in order to execute.
- Reflection: It is not possible to reflect over mixed executables, whereas pure assemblies provide full reflection support. For more information, see Reflection in C++.
- Host Controllability: Because pure assemblies contain only MSIL, they behave more predictably and flexibly than mixed assemblies when used in applications that host the CLR and modify its default behavior.

Limitations of /clr:pure

This section covers features not currently supported by **/clr:pure**.

- Pure assemblies cannot be called by unmanaged functions. Therefore pure assemblies cannot implement COM interfaces or expose native callbacks. Pure assemblies cannot export functions via __declspec(dllexport) or .DEF files. Also, functions declared with the __clrcall convention cannot be imported via __declspec(dllimport). Functions in a native module can be called from a pure assembly, but pure assemblies cannot expose native-callable functions, so exposing functionality in a pure assembly must be done through managed functions in a mixed assembly. See How to: Migrate to /clr:pure for more information.

- ATL and MFC libraries are not supported by pure mode compilation in Visual C++ 2005.
- Pure .netmodules are not accepted as input to the Visual C++ linker. However, pure .obj files are accepted by the linker, and .obj files contain a superset of information contained in netmodules. See .netmodule Files as Linker Input for more information.
- Compiler COM support (#import) is not supported, as this would introduce unmanaged instructions into the pure assembly.
- Floating point options for alignment and exception-handling are not adjustable for pure assemblies. As a result, __declspec(align) cannot be used. This renders some header files, such as fpieee.h, incompatible with /clr:pure.
- The GetLastError function in the PSDK can give undefined behavior when compiling with **/clr:pure**.

Verifiable (/clr:safe)

The **/clr:safe** compiler option generates verifiable assemblies, like those written in Visual Basic and C#, conforming to requirements that allow the common language runtime (CLR) to guarantee that the code does not violate current security settings. For example, if security settings prohibit a component from writing to disk, the CLR can determine if a verifiable component meets this criterion before executing any of the code. There is no CRT support for verifiable assemblies. (CRT support is available to pure assemblies through a Pure MSIL version of the C Runtime library.)

Verifiable assemblies offer these advantages over pure and mixed assemblies:

- Increased security.
- Some situations require it (SQL components, for example).
- Future versions of Windows will increasingly require components and applications to be verifiable.

One disadvantage is that C++ interop features are not available. Verifiable assemblies cannot contain any unmanaged functions or native data types, even if they are not referenced by the managed code.

Despite the use of the word "safe", compiling applications with **/clr:safe** does not mean there are no bugs; it just means that the CLR can verify the security settings at run time.

Regardless of assembly type, calls made from managed assemblies to native DLLs via P/Invoke will compile, but may fail at runtime depending on security settings.

Note
There is one coding scenario that will pass the compiler but that will result in an unverifiable assembly: calling a virtual function through an object instance using the scope resolution operator. For example: `MyObj -> A::VirtualFunction();`.

See Also

Other Resources

.NET Programming in C++

Mixed, Pure, and Verifiable Feature Comparison

This topic compares features among the different **/clr** compilation modes. For more information, see /clr (Common Language Runtime Compilation).

Feature Comparison

Feature	Mixed (/clr)	Pure (/clr:pure)	Safe (/clr:safe)	Related Information
.NET Framework class library	supported	supported	supported	.NET Framework Class Library Reference
CRT library	supported	supported		Run-Time Routines by Category
MFC/ATL	supported			MFC Reference | ATL Class Overview
Unmanaged Functions	supported			Mixed (Native and Managed) Assemblies
Unmanaged Data	supported	supported		Pure and Verifiable Code
Callable from Unmanaged Functions	supported			How to: Migrate to /clr:pure
Supports calling unmanaged Functions	supported	C-style functions only	P/Invoke only	Using C++ Interop (Implicit PInvoke)
Supports Reflection	DLLs only	supported	supported	Reflection in C++

See Also

Reference

Pure and Verifiable Code

How to: Migrate to /clr:pure

This topic discusses issues likely to arise when migrating to pure MSIL using **/clr:pure** (see /clr (Common Language Runtime Compilation) for more information). This topic assumes that the code being migrated is currently complied as mixed assembly using the **/clr** option, as the migration path from unmanaged code to pure MSIL is not a direct one. For unmanaged code, see How To: Migrate to /clr before attempting to migrate to pure MSIL.

Basic Changes

Pure MSIL is comprised of MSIL instructions, so code containing functions that cannot be expressed in MSIL will prevent compilation. This includes functions defined as using calling conventions other than __clrcall. (Non __clrcall functions can be invoked in a pure MSIL component, but not defined.)

To ensure no runtime errors, you should enable the C4412 warnnig. Enable C4412 by adding `#pragma warning (default : 4412)` to each compiland that you compile with **/clr:pure** and that passes C++ types to and from IJW (**/clr)** or native code. See Compiler Warning (level 2) C4412 for more information.

Architectural Considerations

Some of the limitations of pure MSIL assemblies listed in Pure and Verifiable Code have high-level implications for application design and migration strategy. Most notably, unlike mixed assemblies, pure MSIL assemblies don't enjoy full compatibility with unmanaged modules.

Pure MSIL assemblies can call unmanaged functions, but cannot be called by unmanaged functions. As a result, pure MSIL is a better candidate for client code that uses unmanaged functions than it is for server code that is used by unmanaged functions. If functionality contained in a pure MSIL assembly is to be used by unmanaged functions, a mixed assembly must be used as an interface layer.

Applications that use ATL or MFC are not good candidates for migration to pure MSIL, as these libraries are not supported in this release. Likewise, the Platform SDK contains header files that do not compile under **/clr:pure**.

While pure MSIL assemblies can call unmanaged functions, this ability is limited to simple C-style functions. The use of more complex unmanaged APIs is likely to require the unmanaged functionality to be exposed in the form of a COM interface, or a mixed assembly that can act as an interface between the pure MSIL and unmanaged components. Using a mixed assembly layer is the only way to use unmanaged functions that take callback functions, for example, as a pure assembly is unable to provide a native callable function for use as a callback.

Application Domains and Calling Conventions

Although it is possible for pure MSIL assemblies use unmanaged functionality, functions and static data are handled differently. In pure assemblies, functions are implemented with the __clrcall calling convention, and static data is stored per-application domain. This differs from the default for unmanaged and mixed assemblies, which use the __cdecl calling convention for functions and store static data on a per-process basis.

Within the context of pure MSIL (and verifiable code compiled with /clr:safe) these defaults are transparent, as __clrcall is the default calling convention of the CLR, and application domains are the native scope for static and global data in .NET applications. However, when interfacing with unmanaged or mixed components, the differing treatment of functions and global data can cause problems.

For example, if a pure MSIL component is to call functions in an unmanaged or mixed DLL, a header file for the DLL will be used to compile the pure assembly. However, unless the calling convention for each function in the header is indicated explicitly, they will all be assumed to be __clrcall. This will later cause runtime failures, as these functions are likely implemented with the __cdecl convention. The functions in the unmanaged header file can be explicitly marked as __cdecl, or the entire DLL source code must be recompiled under **/clr:pure**.

Similarly, function pointers are assumed to point to __clrcall functions under **/clr:pure** compilation. These too must be explicitly annotated with the correct calling convention.

For more information, see Application Domains and Visual C++.

Linking Limitations

The Microsoft Visual C++ 2005 linker will not attempt to link mixed and pure OBJ files, as the storage scope and calling conventions are different.

See Also

Reference

Pure and Verifiable Code

How to: Create Verifiable C++ Projects

Visual C++ application wizards do not create verifiable projects, but projects can be converted to be verifiable. This topic describes how to set project properties and modify project source files to transform your Visual C++ projects to produce verifiable applications.

Compiler and Linker Settings

By default, .NET projects use the /clr compiler flag and configure the linker to target x86 hardware. For verifiable code, you must use the /clr:safe flag, and you must instruct the linker to generate MSIL instead of native machine instructions.

To change the compiler and linker settings

1. Display the project Property Page. For more information, see How to: Open Project Property Pages.
2. On the **General** page under the **Configuration Properties** node, set the **Common Language Runtime Support** property to **Safe MSIL Common Language Runtime Support (/clr:safe)**.
3. On the **Advanced** page under the **Linker** node, set the **CLR Image Type** property to **Force safe IL image (/CLRIMAGETYPE:SAFE)**.

Removing Native Data Types

Because native data types are non-verifiable, even if they are not actually used, you must remove all header files containing native types.

Note
The following procedure applies to Windows Forms Application (.NET) and Console Application (.NET) projects.

To remove references to native data types

- Comment out everything in the Stdafx.h file.

Configuring an Entry Point

Because verifiable applications cannot use the C run-time libraries (CRT), they cannot depend on the CRT to call the main function as the standard entry point. This means that you must explicitly provide the name of the function to be called initially to the linker. (In this case, Main() is used instead of main() or _tmain() to indicate a non-CRT entry point, but because the entry point must be specified explicitly, this name is arbitrary.)

Note
The following procedures apply to Console Application (.NET) projects.

To configure an entry point

1. Change _tmain() to Main() in the project's main .cpp file.
2. Display the project Property Page. For more information, see How to: Open Project Property Pages.
3. On the **Advanced** page under the **Linker** node, enter **Main** as the **Entry Point** property value.

See Also

Reference

Pure and Verifiable Code

How to: Migrate to /clr:safe

Visual C++ 2005 can generate verifiable components with using **/clr:safe**, which causes the compiler to generate errors for each non-verifiable code construct.

Remarks

The following issues generate verifiability errors:

- Native types. Even if it isn't used, the declaration of native classes, structures, pointers, or arrays will prevent compilation.
- Global variables
- Function calls into any unmanaged library, including common language runtime function calls
- A verifiable function cannot contain a static_cast Operator for down-casting. The static_cast Operator can be used for casting between primitive types, but for down-casting, safe_cast or a C-Style cast (which is implemented as a safe_cast) must be used.
- A verifiable function cannot contain a reinterpret_cast Operator (or any C-style cast equivalent).
- A verifiable function cannot perform arithmetic on an interior_ptr. It may only assign to it and dereference it.
- A verifiable function can only throw or catch pointers to reference types, so value types must be boxed before throwing.
- A verifiable function can only call verifiable functions (such that calls to the common language runtime are not allowed, include **AtEntry**/**AtExit**, and so global constructors are disallowed).
- A verifiable class cannot use Explicit.
- If building an EXE, a main function cannot declare any parameters, so GetCommandLineArgs must be used to retrieve command-line arguments.
- Making a non-virtual call to a virtual function. For example:

```
// not_verifiable.cpp
// compile with: /clr
ref struct A {
   virtual void Test() {}
};

ref struct B : A {};

int main() {
   B^ b1 = gcnew B;
   b1->A::Test();   // Non-virtual call to virtual function
}
```

Also, the following keywords cannot be used in verifiable code:

- unmanaged and pack pragmas
- naked and align __declspec modifiers
- __asm
- __based
- __try and **__except**

See Also

Reference

Pure and Verifiable Code

Using Verifiable Assemblies with SQL Server

Extended stored procedures, packaged as dynamic-link libraries (DLLs), provide a way to extend SQL Server functionality through functions developed with Visual C++. Extended stored procedures are implemented as functions inside DLLs. In addition to functions, extended stored procedures can also define user-defined types and aggregate functions (such as SUM or AVG).

When a client executes an extended stored procedure, SQL Server searches for the DLL associated with the extended stored procedure and loads the DLL. SQL Server calls the requested extended stored procedure and executes it under a specified security context. The extended stored procedure then passes result sets and returns parameters back to the server.

SQL Server 2005 provides extensions to Transact-SQL (T-SQL) to allow you to install verifiable assemblies into SQL Server. The SQL Server permission set specifies the security context, with the following levels of security:

- Unrestricted mode: Run code at your own risk; code does not have to be verifiably type-safe.
- Safe mode: Run verifiably typesafe code; compiled with /clr:safe.

Safe mode requires the executed assemblies to be verifiably typesafe.

To create and load a verifiable assembly into SQL Server, use the Transact-SQL commands CREATE ASSEMBLY and DROP ASSEMBLY as follows:

```
CREATE ASSEMBLY <assemblyName> FROM <'Assembly UNC Path'> WITH
  PERMISSION_SET <permissions>
DROP ASSEMBLY <assemblyName>
```

The PERMISSION_SET command specifies the security context, and can have the values UNRESTRICTED, SAFE, or EXTENDED.

In addition, you can use the CREATE FUNCTION command to bind to method names in a class:

```
CREATE FUNCTION <FunctionName>(<FunctionParams>)
RETURNS returnType
[EXTERNAL NAME <AssemblyName>:<ClassName>::<StaticMethodName>]
```

Example

The following SQL script (for example, named "MyScript.sql") loads an assembly into SQL Server and makes a method of a class available:

```
-- Create assembly without external access
drop assembly stockNoEA
go
create assembly stockNoEA
from
'c:\stockNoEA.dll'
with permission_set safe

-- Create function on assembly with no external access
drop function GetQuoteNoEA
go
create function GetQuoteNoEA(@sym nvarchar(10))
returns real
external name stockNoEA:StockQuotes::GetQuote
go

-- To call the function
select dbo.GetQuoteNoEA('MSFT')
go
```

SQL scripts can be executed interactively in SQL Query Analyzer or with the command-line option using Isqlw.exe. For information about Isqlw.exe, see isqlw Utility. The following command line connects to MyServer, uses the default database, uses a trusted connection, inputs MyScript.sql, and outputs MyResult.txt.

```
isqlw –S MyServer -E –i myScript.sql –o myResult.txt
```

See Also

Reference

How to: Migrate to /clr:safe

Transact-SQL Overview

Programming Extended Stored Procedures

Classes, Structures, and Unions

isqlw Utility

Other Resources

Extended Stored Procedures

Regular Expressions in C++

Demonstrates various string operations using regular expressions classes in the .NET Framework.

The following topics demonstrate the use of the .NET Framework System.Text.RegularExpressions namespace (and in one case the System.String.Split method) to search, parse, and modify strings.

In This Section

- How to: Parse Strings Using Regular Expressions
- How to: Parse Strings Using the Split Method
- How to: Use Regular Expressions for Simple Matching
- How to: Use Regular Expressions to Extract Data Fields
- How to: Use Regular Expressions to Rearrange Data
- How to: Use Regular Expressions to Search and Replace
- How to: Use Regular Expressions to Validate Data Formatting

Related Sections

.NET Framework Regular Expressions

See Also

Other Resources

.NET Programming in C++

How to: Use Regular Expressions for Simple Matching

The following code example uses regular expressions to look for exact substring matches. The search is performed by the static IsMatch method, which takes two strings as input. The first is the string to be searched, and the second is the pattern to be searched for.

Example

```
// regex_simple.cpp
// compile with: /clr
#using <System.dll>

using namespace System;
using namespace System::Text::RegularExpressions;

int main()
{
   array<String^>^ sentence =
   {
      "cow over the moon",
      "Betsy the Cow",
      "cowering in the corner",
      "no match here"
   };

   String^ matchStr = "cow";
   for (int i=0; i<sentence->Length; i++)
   {
      Console::Write( "{0,24}", sentence[i] );
      if ( Regex::IsMatch( sentence[i], matchStr,
                           RegexOptions::IgnoreCase ) )
         Console::WriteLine("  (match for '{0}' found)", matchStr);
      else
         Console::WriteLine("");
   }
   return 0;
}
```

See Also

Other Resources

.NET Framework Regular Expressions

.NET Programming in C++

How to: Use Regular Expressions to Extract Data Fields

The following code example demonstrates the use of regular expressions to extract data from a formatted string. The following code example uses the Regex class to specify a pattern that corresponds to an e-mail address. This patter includes field identifiers that can be used to retrieve the user and host name portions of each e-mail address. The Match class is used to perform the actual pattern matching. If the given e-mail address is valid, the user name and host names are extracted and displayed.

Example

```
// Regex_extract.cpp
// compile with: /clr
#using <System.dll>

using namespace System;
using namespace System::Text::RegularExpressions;

int main()
{
    array<String^>^ address=
    {
        "jay@southridgevideo.com",
        "barry@adatum.com",
        "treyresearch.net",
        "karen@proseware.com"
    };

    Regex^ emailregex = gcnew Regex("(?<user>[^@]+)@(?<host>.+)");

    for (int i=0; i<address->Length; i++)
    {
        Match^ m = emailregex->Match( address[i] );
        Console::Write("\n{0,25}", address[i]);

        if ( m->Success )
        {
            Console::Write("   User='{0}'",
            m->Groups["user"]->Value);
            Console::Write("   Host='{0}'",
            m->Groups["host"]->Value);
        }
        else
            Console::Write("   (invalid email address)");
        }

    Console::WriteLine("");
    return 0;
}
```

See Also

Other Resources

.NET Framework Regular Expressions

.NET Programming in C++

How to: Use Regular Expressions to Rearrange Data

The following code example demonstrates how the .NET Framework regular expression support can be used to rearrange, or reformat data. The following code example uses the Regex and Match classes to extract first and last names from a string and then display these name elements in reverse order.

The **Regex** class is used to construct a regular expression that describes the current format of the data. The two names are assumed to be separated by a comma and can use any amount of white-space around the comma. The **Match** method is then used to analyze each string. If successful, first and last names are retrieved from the **Match** object and displayed.

Example

```
// regex_reorder.cpp
// compile with: /clr
#using <System.dll>
using namespace System;
using namespace Text::RegularExpressions;

int main()
{
   array<String^>^ name =
   {
      "Abolrous, Sam",
      "Berg,Matt",
      "Berry , Jo",
      "www.contoso.com"
   };

   Regex^ reg = gcnew Regex("(?<last>\\w*)\\s*,\\s*(?<first>\\w*)");

   for ( int i=0; i < name->Length; i++ )
   {
      Console::Write( "{0,-20}", name[i] );
      Match^ m = reg->Match( name[i] );
      if ( m->Success )
      {
         String^ first = m->Groups["first"]->Value;
         String^ last = m->Groups["last"]->Value;
         Console::WriteLine("{0} {1}", first, last);
      }
      else
         Console::WriteLine("(invalid)");
   }
   return 0;
}
```

See Also

Other Resources

.NET Framework Regular Expressions

.NET Programming in C++

How to: Use Regular Expressions to Search and Replace

The following code example demonstrates how the regular expression class Regex can be used to perform search and replace. This is done with the Replace method. The version used takes two strings as input: the string to be modified, and the string to be inserted in place of the sections (if any) that match the pattern given to the **Regex** object.

This code replaces all the digits in a string with underscores (_) and then replaces those with an empty string, effectively removing them. The same effect can be accomplished in a single step, but two steps are used here for demonstration purposes.

Example

```
// regex_replace.cpp
// compile with: /clr
#using <System.dll>
using namespace System::Text::RegularExpressions;
using namespace System;

int main()
{
   String^ before = "The q43uick bro254wn f0ox ju4mped";
   Console::WriteLine("original  : {0}", before);

   Regex^ digitRegex = gcnew Regex("(?<digit>[0-9])");
   String^ after = digitRegex->Replace(before, "_");
   Console::WriteLine("1st regex : {0}", after);

   Regex^ underbarRegex = gcnew Regex("_");
   String^ after2 = underbarRegex->Replace(after, "");
   Console::WriteLine("2nd regex : {0}", after2);

   return 0;
}
```

See Also

Other Resources

.NET Framework Regular Expressions

.NET Programming in C++

How to: Use Regular Expressions to Validate Data Formatting

The following code example demonstrates the use of regular expressions to verify the formatting of a string. In the following code example, the string should contain a valid phone number. The following code example uses the string "\d{3}-\d{3}-\d{4}" to indicate that each field represents a valid phone number. The "d" in the string indicates a digit, and the argument after each "d" indicates the number of digits that must be present. In this case, the number is required to be separated by dashes.

Example

```
// regex_validate.cpp
// compile with: /clr
#using <System.dll>

using namespace System;
using namespace Text::RegularExpressions;

int main()
{
   array<String^>^ number =
   {
      "123-456-7890",
      "444-234-22450",
      "690-203-6578",
      "146-893-232",
      "146-839-2322",
      "4007-295-1111",
      "407-295-1111",
      "407-2-5555",
   };

   String^ regStr = "^\\d{3}-\\d{3}-\\d{4}$";

   for ( int i = 0; i < number->Length; i++ )
   {
      Console::Write( "{0,14}", number[i] );

      if ( Regex::IsMatch( number[i], regStr ) )
         Console::WriteLine(" - valid");
      else
         Console::WriteLine(" - invalid");
   }
   return 0;
}
```

See Also

Other Resources

.NET Framework Regular Expressions

.NET Programming in C++

How to: Parse Strings Using the Split Method

The following code example demonstrates using the System.String.Split method to extract each word from a string. A string containing multiple types of word delineators is constructed and then parsed by calling **Split** with a list of the delineators. Then, each word in the sentence is displayed separately.

Example

```
// regex_split.cpp
// compile with: /clr
using namespace System;

int main()
{
   String^ delimStr = " ,.:\t";
   Console::WriteLine( "delimiter : '{0}'", delimStr );
   array<Char>^ delimiter = delimStr->ToCharArray( );
   array<String^>^ words;
   String^ line = "one\ttwo three:four,five six seven";

   Console::WriteLine( "text : '{0}'", line );
   words = line->Split( delimiter );
   Console::WriteLine( "Number of Words : {0}", words->Length );
   for (int word=0; word<words->Length; word++)
      Console::WriteLine( "{0}", words[word] );

   return 0;
}
```

See Also

Other Resources

.NET Framework Regular Expressions

.NET Programming in C++

How to: Parse Strings Using Regular Expressions

The following code example demonstrates simple string parsing using the Regex class in the System.Text.RegularExpressions namespace. A string containing multiple types of word delineators is constructed. The string is then parsed using the **Regex** class in conjunction with the Match class. Then, each word in the sentence is displayed separately.

Example

```
// regex_parse.cpp
// compile with: /clr
#using <system.dll>

using namespace System;
using namespace System::Text::RegularExpressions;

int main( )
{
   int words = 0;
   String^ pattern = "[a-zA-Z]*";
   Console::WriteLine( "pattern : '{0}'", pattern );
   Regex^ regex = gcnew Regex( pattern );

   String^ line = "one\ttwo three:four,five six  seven";
   Console::WriteLine( "text : '{0}'", line );
   for( Match^ match = regex->Match( line );
        match->Success; match = match->NextMatch( ) )
   {
      if( match->Value->Length > 0 )
      {
         words++;
         Console::WriteLine( "{0}", match->Value );
      }
   }
   Console::WriteLine( "Number of Words : {0}", words );

   return 0;
}
```

See Also

Other Resources

.NET Framework Regular Expressions

.NET Programming in C++

File Handling and I/O in C++

Demonstrates various file operations using the .NET Framework.

The following topics demonstrate the use of classes defined in the System.IO namespace to perform various file operations.

In This Section

- How to: Enumerate Files in a Directory
- How to: Monitor File System Changes
- How to: Read a Binary File
- How to: Read a Text File
- How to: Retrieve File Information
- How to: Write a Binary File
- How to: Write a Text File

Related Sections

File and Stream I/O

See Also

Other Resources

.NET Programming in C++

How to: Read a Binary File

The following code example demonstrates reading binary data from a file. Two classes from the System.IO namespace are used: FileStream and BinaryReader. **FileStream** represents the actual file. **BinaryReader** provides an interface to the stream that allows binary access.

The following code example uses a file created by the code in How to: Write a Binary File called data.bin.

Example

```
// binary_read.cpp
// compile with: /clr
#using<system.dll>
using namespace System;
using namespace System::IO;

int main()
{
   String^ fileName = "data.bin";
   try
   {
      FileStream^ fs = gcnew FileStream(fileName, FileMode::Open);
      BinaryReader^ br = gcnew BinaryReader(fs);

      Console::WriteLine("contents of {0}:", fileName);
      while (br->BaseStream->Position < br->BaseStream->Length)
         Console::WriteLine(br->ReadInt32().ToString());

      fs->Close( );
   }
   catch (Exception^ e)
   {
      if (dynamic_cast<FileNotFoundException^>(e))
         Console::WriteLine("File '{0}' not found", fileName);
      else
         Console::WriteLine("Exception: ({0})", e);
      return -1;
   }
   return 0;
}
```

See Also

Other Resources

File and Stream I/O

.NET Programming in C++

How to: Write a Binary File

The following code example demonstrates writing binary data to a file. Two classes from the System.IO namespace are used: FileStream and BinaryWriter. **FileStream** represents the actual file, while **BinaryWriter** provides an interface to the stream that allows binary access.

The following code example writes a file containing integers in binary format. This file can be read with the code in How to: Read a Binary File.

Example

```
// binary_write.cpp
// compile with: /clr
#using<system.dll>
using namespace System;
using namespace System::IO;

int main()
{
   array<Int32>^ data = {1, 2, 3, 10000};

   FileStream^ fs = gcnew FileStream("data.bin", FileMode::Create);
   BinaryWriter^ w = gcnew BinaryWriter(fs);

   try
   {
      Console::WriteLine("writing data to file:");
      for (int i=0; i<data->Length; i++)
      {
         Console::WriteLine(data[i]);
         w->Write(data[i]);
      }
   }
   catch (Exception^)
   {
     Console::WriteLine("data could not be written");
     fs->Close();
     return -1;
   }

   fs->Close();
   return 0;
}
```

See Also

Other Resources

File and Stream I/O

.NET Programming in C++

How to: Read a Text File

The following code example demonstrates how to open and read a text file one line at a time. This is accomplished with the StreamReader class defined within the System.IO namespace. An instance of this class is used to open a text file and then the System.IO.StreamReader.ReadLine method is used to retrieve each line.

This code can be used with any file named textfile.txt that contains text or with the file generated in How to: Write a Text File.

Example

```
// text_read.cpp
// compile with: /clr
#using<system.dll>
using namespace System;
using namespace System::IO;

int main()
{
   String^ fileName = "textfile.txt";
   try
   {
      Console::WriteLine("trying to open file {0}...", fileName);
      StreamReader^ din = File::OpenText(fileName);

      String^ str;
      int count = 0;
      while ((str = din->ReadLine()) != nullptr)
      {
         count++;
         Console::WriteLine("line {0}: {1}", count, str );
      }
   }
   catch (Exception^ e)
   {
      if (dynamic_cast<FileNotFoundException^>(e))
         Console::WriteLine("file '{0}' not found", fileName);
      else
         Console::WriteLine("problem reading file '{0}'", fileName);
   }

   return 0;
}
```

See Also

Other Resources

File and Stream I/O

.NET Programming in C++

How to: Write a Text File

The following code example demonstrates how to create a text file and write text to it using the StreamWriter class, which is defined in the System.IO namespace. The **StreamWriter** constructor takes the name of the file to be created. If the file exists, it is overwritten (unless you pass True as the second StringWriter constructor argument).

The file is then filed using the Write and WriteLine functions.

Example

```
// text_write.cpp
// compile with: /clr
using namespace System;
using namespace System::IO;

int main()
{
   String^ fileName = "textfile.txt";

   StreamWriter^ sw = gcnew StreamWriter(fileName);
   sw->WriteLine("A text file is born!");
   sw->Write("You can use WriteLine");
   sw->WriteLine("...or just Write");
   sw->WriteLine("and do {0} output too.", "formatted");
   sw->WriteLine("You can also send non-text objects:");
   sw->WriteLine(DateTime::Now);
   sw->Close();
   Console::WriteLine("a new file ('{0}') has been written", fileName);

   return 0;
}
```

See Also

Other Resources

File and Stream I/O

.NET Programming in C++

How to: Enumerate Files in a Directory

The following code example demonstrates how to retrieve a list of the files in a directory. Additionally, the subdirectories are enumerated. The following code example uses the GetFiles **GetFiles** and GetDirectories methods to display the contents of the C:\Windows directory.

Example

```
// enum_files.cpp
// compile with: /clr
using namespace System;
using namespace System::IO;

int main()
{
   String^ folder = "C:\\";
   array<String^>^ dir = Directory::GetDirectories( folder );
   Console::WriteLine("--== Directories inside '{0}' ==--", folder);
   for (int i=0; i<dir->Length; i++)
      Console::WriteLine(dir[i]);

   array<String^>^ file = Directory::GetFiles( folder );
   Console::WriteLine("--== Files inside '{0}' ==--", folder);
   for (int i=0; i<file->Length; i++)
      Console::WriteLine(file[i]);

   return 0;
}
```

See Also

Other Resources

File and Stream I/O

.NET Programming in C++

How to: Monitor File System Changes

The following code example uses FileSystemWatcher to register for events corresponding to files being created, changed, deleted, or renamed. Instead of periodically polling a directory for changes to files, you can use the **FileSystemWatcher** class to fire events when a change is detected.

Example

```
// monitor_fs.cpp
// compile with: /clr
#using <system.dll>

using namespace System;
using namespace System::IO;

ref class FSEventHandler
{
public:
    void OnChanged (Object^ source, FileSystemEventArgs^ e)
    {
        Console::WriteLine("File: {0} {1}",
               e->FullPath, e->ChangeType);
    }
    void OnRenamed(Object^ source, RenamedEventArgs^ e)
    {
        Console::WriteLine("File: {0} renamed to {1}",
                e->OldFullPath, e->FullPath);
    }
};

int main()
{
   array<String^>^ args = Environment::GetCommandLineArgs();

   if(args->Length < 2)
   {
      Console::WriteLine("Usage: Watcher.exe <directory>");
      return -1;
   }

   FileSystemWatcher^ fsWatcher = gcnew FileSystemWatcher( );
   fsWatcher->Path = args[1];
   fsWatcher->NotifyFilter = static_cast<NotifyFilters>
              (NotifyFilters::FileName |
               NotifyFilters::Attributes |
               NotifyFilters::LastAccess |
               NotifyFilters::LastWrite |
               NotifyFilters::Security |
               NotifyFilters::Size );

    FSEventHandler^ handler = gcnew FSEventHandler();
    fsWatcher->Changed += gcnew FileSystemEventHandler(
            handler, &FSEventHandler::OnChanged);
    fsWatcher->Created += gcnew FileSystemEventHandler(
            handler, &FSEventHandler::OnChanged);
    fsWatcher->Deleted += gcnew FileSystemEventHandler(
            handler, &FSEventHandler::OnChanged);
    fsWatcher->Renamed += gcnew RenamedEventHandler(
            handler, &FSEventHandler::OnRenamed);

    fsWatcher->EnableRaisingEvents = true;

    Console::WriteLine("Press Enter to quit the sample.");
    Console::ReadLine( );
}
```

See Also

Reference

System.IO namespace

Other Resources

File and Stream I/O

.NET Programming in C++

How to: Retrieve File Information

The following code example demonstrates the FileInfo class. When you have the name of a file, you can use this class to retrieve information about the file such as the file size, directory, full name, and date and time of creation and of the last modification.

This code retrieves file information for Notepad.exe.

Example

```
// file_info.cpp
// compile with: /clr
using namespace System;
using namespace System::IO;

int main()
{
   array<String^>^ args = Environment::GetCommandLineArgs();
   if (args->Length < 2)
   {
      Console::WriteLine("\nUSAGE : file_info <filename>\n\n");
      return -1;
   }

   FileInfo^ fi = gcnew FileInfo( args[1] );

   Console::WriteLine("file size: {0}", fi->Length );

   Console::Write("File creation date:  ");
   Console::Write(fi->CreationTime.Month.ToString());
   Console::Write(".{0}", fi->CreationTime.Day.ToString());
   Console::WriteLine(".{0}", fi->CreationTime.Year.ToString());

   Console::Write("Last access date:  ");
   Console::Write(fi->LastAccessTime.Month.ToString());
   Console::Write(".{0}", fi->LastAccessTime.Day.ToString());
   Console::WriteLine(".{0}", fi->LastAccessTime.Year.ToString());

   return 0;
}
```

See Also

Other Resources

File and Stream I/O

.NET Programming in C++

Graphics Operations Using the .NET Framework in C++

Demonstrates image manipulation using the .NET Framework SDK.

The following topics demonstrate the use of the System.Drawing.Image class to perform image manipulation.

In This Section

How to: Convert Image File Formats with the .NET Framework

How to: Display Images with the .NET Framework

How to: Draw Shapes with the .NET Framework

How to: Rotate Images with the .NET Framework

Related Sections

Getting Started with Graphics Programming

About GDI+ Managed Code

System.Drawing

See Also

Other Resources

.NET Programming in C++

How to: Convert Image File Formats with the .NET Framework

The following code example demonstrates the System.Drawing.Image class and the System.Drawing.Imaging.ImageFormat enumeration used to convert and save image files. The following code loads an image from a .jpg file and then saves it in both .gif and .bmp file formats.

Note

GDI+ is included with Windows XP and is available as a redistributable for Windows NT 4.0 SP 6, Windows 2000, Windows 98, and Windows Me. To download the latest redistributable, see http://go.microsoft.com/fwlink/?linkid=11232. For more information, see GDI+.

Example

```
#using <system.drawing.dll>

using namespace System;
using namespace System::Drawing;
using namespace System::Drawing::Imaging;

int main()
{
   Image^ image = Image::FromFile("SampleImage.jpg");
   image->Save("SampleImage.png", ImageFormat::Png);
   image->Save("SampleImage.bmp", ImageFormat::Bmp);

   return 0;
}
```

See Also

Reference

System.Drawing

Other Resources

.NET Programming in C++

How to: Display Images with the .NET Framework

The following code example modifies the OnPaint event handler to retrieve a pointer to the Graphics object for the main form. The OnPaint function is intended for a Windows Forms application, most likely created with a Visual Studio application wizard.

The image is represented by the Image class. The image data is loaded from a .jpg file using the System.Drawing.Image.FromFile method. Before the image is drawn to the form, the form is resized to accommodate the image. The drawing of the image is performed with the System.Drawing.Graphics.DrawImage method.

The **Graphics** and **Image** classes are both in the System.Drawing namespace.

Note

GDI+ is included with Windows XP and is available as a redistributable for Windows NT 4.0 SP 6, Windows 2000, Windows 98, and Windows Me. To download the latest redistributable, see http://go.microsoft.com/fwlink/?linkid=11232. For more information, see the GDI+ SDK documentation at GDI+.

Example

```
#using <system.drawing.dll>

using namespace System;
using namespace System::Drawing;

protected:
Void Form1::OnPaint(PaintEventArgs^ pe)
{
    Graphics^ g = pe->Graphics;
    Image^ image = Image::FromFile("SampleImage.jpg");
    Form::ClientSize = image->Size;
    g->DrawImage( image, 0, 0, image->Size.Width, image->Size.Height );
}
```

See Also

Reference

System.Drawing

Other Resources

.NET Programming in C++

How to: Rotate Images with the .NET Framework

The following code example demonstrates the use of the System.Drawing.Image class to load an image from disk, rotate it 90 degrees, and save it as a new .jpg file.

Note

GDI+ is included with Windows XP and is available as a redistributable for Windows NT 4.0 SP 6, Windows 2000, Windows 98, and Windows Millennium Edition. To download the latest redistributable, see http://go.microsoft.com/fwlink/?linkid=11232. For more information, see GDI+.

Example

```
#using <system.drawing.dll>

using namespace System;
using namespace System::Drawing;

int main()
{
   Image^ image = Image::FromFile("SampleImage.jpg");
   image->RotateFlip( RotateFlipType::Rotate90FlipNone );
   image->Save("SampleImage_rotated.jpg");
   return 0;
}
```

See Also

Other Resources

.NET Programming in C++

How to: Draw Shapes with the .NET Framework

The following code example uses the Graphics class to modify the OnPaint event handler to retrieve a pointer to the **Graphics** object for the main form. This pointer is then used to set the background color of the form and draw a line and an arc using the System.Drawing.Graphics.DrawLine and DrawArc methods.

Note

GDI+ is included with Windows XP and is available as a redistributable for Windows NT 4.0 SP 6, Windows 2000, Windows 98, and Windows Me. To download the latest redistributable, see http://go.microsoft.com/fwlink/?linkid=11232. For more information, see GDI+.

Example

```
#using <system.drawing.dll>
using namespace System;
using namespace System::Drawing;
// ...
protected:
Void Form1::OnPaint(PaintEventArgs^ pe )
{
   Graphics^ g = pe->Graphics;
   g->Clear(Color::AntiqueWhite);

   Rectangle rect = Form::ClientRectangle;
   Rectangle smallRect;
   smallRect.X = rect.X + rect.Width / 4;
   smallRect.Y = rect.Y + rect.Height / 4;
   smallRect.Width = rect.Width / 2;
   smallRect.Height = rect.Height / 2;

   Pen^ redPen = gcnew Pen(Color::Red);
   redPen->Width = 4;
   g->DrawLine(redPen, 0, 0, rect.Width, rect.Height);

   Pen^ bluePen = gcnew Pen(Color::Blue);
   bluePen->Width = 10;
   g->DrawArc( bluePen, smallRect, 90, 270 );
}
```

See Also

Reference

System::Drawing namespace

Other Resources

.NET Programming in C++

Windows Operations in C++

Demonstrates various Windows-specific tasks using the .NET Framework SDK.

The following topics demonstrate various Windows operations performed with the .NET Framework SDK using Visual C++.

In This Section

How to: Determine if Shutdown Has Started

How to: Determine the User Interactive State

How to: Read Data from the Windows Registry

How to: Read Windows Performance Counters

How to: Retrieve Text from the Clipboard

How to: Retrieve the .NET Framework Version

How to: Retrieve the Current Username

How to: Retrieve the Local Machine Name

How to: Retrieve the Windows Version

How to: Retrieve Time Elapsed Since Startup

How to: Store Text in the Clipboard

How to: Write Data to the Windows Registry

Related Sections

Environment

See Also

Other Resources

.NET Programming in C++

How to: Write Data to the Windows Registry

The following code example uses the CurrentUser key to create a writable instance of the RegistryKey class corresponding to the **Software** key. The CreateSubKey method is then used to create a new key and add to key/value pairs.

Example

```
// registry_write.cpp
// compile with: /clr
using namespace System;
using namespace Microsoft::Win32;

int main()
{
   // The second OpenSubKey argument indicates that
   // the subkey should be writable.
   RegistryKey^ rk;
   rk  = Registry::CurrentUser->OpenSubKey("Software", true);
   if (!rk)
   {
      Console::WriteLine("Failed to open CurrentUser/Software key");
      return -1;
   }

   RegistryKey^ nk = rk->CreateSubKey("NewRegKey");
   if (!nk)
   {
      Console::WriteLine("Failed to create 'NewRegKey'");
      return -1;
   }

   String^ newValue = "NewValue";
   try
   {
      nk->SetValue("NewKey", newValue);
      nk->SetValue("NewKey2", 44);
   }
   catch (Exception^)
   {
      Console::WriteLine("Failed to set new values in 'NewRegKey'");
      return -1;
   }

   Console::WriteLine("New key created.");
   Console::Write("Use REGEDIT.EXE to verify ");
   Console::WriteLine("'CURRENTUSER/Software/NewRegKey'\n");
   return 0;
}
```

Remarks

You can use the .NET Framework to access the registry with the Registry and RegistryKey classes, which are both defined in the Microsoft.Win32 namespace. The **Registry** class is a container for static instances of the RegistryKey class. Each instance represents a root registry node. The instances are ClassesRoot, CurrentConfig, CurrentUser, LocalMachine, and Users.

See Also

Concepts

How to: Read Data from the Windows Registry

Other Resources

.NET Programming in C++

How to: Read Data from the Windows Registry

The following code example uses the CurrentUser key to read data from the Windows registry. First, the subkeys are enumerated using the GetSubKeyNames method and then the Identities subkey is opened using the OpenSubKey method. Like the root keys, each subkey is represented by the RegistryKey class. Finally, the new **RegistryKey** object is used to enumerate the key/value pairs.

Example

```
// registry_read.cpp
// compile with: /clr
using namespace System;
using namespace Microsoft::Win32;

int main( )
{
   array<String^>^ key = Registry::CurrentUser->GetSubKeyNames( );

   Console::WriteLine("Subkeys within CurrentUser root key:");
   for (int i=0; i<key->Length; i++)
   {
      Console::WriteLine("   {0}", key[i]);
   }

   Console::WriteLine("Opening subkey 'Identities'...");
   RegistryKey^ rk = nullptr;
   rk = Registry::CurrentUser->OpenSubKey("Identities");
   if (rk==nullptr)
   {
      Console::WriteLine("Registry key not found - aborting");
      return -1;
   }

   Console::WriteLine("Key/value pairs within 'Identities' key:");
   array<String^>^ name = rk->GetValueNames( );
   for (int i=0; i<name->Length; i++)
   {
      String^ value = rk->GetValue(name[i])->ToString();
      Console::WriteLine("   {0} = {1}", name[i], value);
   }

   return 0;
}
```

Remarks

The Registry class is merely a container for static instances of **RegistryKey**. Each instance represents a root registry node. The instances are ClassesRoot, CurrentConfig, **CurrentUser**, LocalMachine, and Users.

In addition to being static, the objects within the **Registry** class are read-only. Furthermore, instances of the RegistryKey class that are created to access the contents of the registry objects are read-only as well. For an example of how to override this behavior, see How to: Write Data to the Windows Registry.

There are two additional objects in the **Registry** class: DynData and PerformanceData. Both are instances of the **RegistryKey** class. The **DynData** object contains dynamic registry information, which is only supported in Windows 98 and Windows Me. The **PerformanceData** object can be used to access performance counter information for applications that use the Windows Performance Monitoring System. The **PerformanceData** node represents information that is not actually stored in the registry and therefore cannot be viewed using Regedit.exe.

See Also

Concepts

How to: Write Data to the Windows Registry

Other Resources

Windows Operations in C++

How to: Retrieve Text from the Clipboard

The following code example uses the GetDataObject member function to return a pointer to the IDataObject interface. This interface can then be queried for the format of the data and used to retrieve the actual data.

Example

```
// read_clipboard.cpp
// compile with: /clr
#using <system.dll>
#using <system.Drawing.dll>
#using <system.windows.forms.dll>

using namespace System;
using namespace System::Windows::Forms;

[STAThread] int main( )
{
   IDataObject^ data = Clipboard::GetDataObject( );

   if (data)
   {
      if (data->GetDataPresent(DataFormats::Text))
      {
         String^ text = static_cast<String^>
           (data->GetData(DataFormats::Text));
         Console::WriteLine(text);
      }
      else
         Console::WriteLine("Nontext data is in the Clipboard.");
   }
   else
   {
      Console::WriteLine("No data was found in the Clipboard.");
   }

   return 0;
}
```

See Also

Other Resources

Windows Operations in C++
.NET Programming in C++

How to: Store Text in the Clipboard

The following code example uses the Clipboard object defined in the System.Windows.Forms namespace to store a string. This object provides two member functions: SetDataObject and GetDataObject. Data is stored in the Clipboard by sending any object derived from Object to **SetDataObject**.

Example

```
// store_clipboard.cpp
// compile with: /clr
#using <System.dll>
#using <System.Drawing.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

[STAThread] int main()
{
   String^ str = "This text is copied into the Clipboard.";

   // Use 'true' as the second argument if
   // the data is to remain in the clipboard
   // after the program terminates.
   Clipboard::SetDataObject(str, true);

   Console::WriteLine("Added text to the Clipboard.");

   return 0;
}
```

See Also

Tasks

How to: Retrieve Text from the Clipboard

Other Resources

Windows Operations in C++

.NET Programming in C++

How to: Retrieve Time Elapsed Since Startup

The following code example demonstrates how to determine the tick count, or number of milliseconds that have elapsed since Windows was started. This value is stored in the System.Environment.TickCount member and, because it is a 32-bit value, resets to zero approximately every 24.9 days.

Example

```
// startup_time.cpp
// compile with: /clr
using namespace System;

int main( )
{
   Int32 tc = Environment::TickCount;
   Int32 seconds = tc / 1000;
   Int32 minutes = seconds / 60;
   float hours = static_cast<float>(minutes) / 60;
   float days = hours / 24;

   Console::WriteLine("Milliseconds since startup: {0}", tc);
   Console::WriteLine("Seconds since startup: {0}", seconds);
   Console::WriteLine("Minutes since startup: {0}", minutes);
   Console::WriteLine("Hours since startup: {0}", hours);
   Console::WriteLine("Days since startup: {0}", days);

   return 0;
}
```

See Also

Other Resources

Windows Operations in C++
.NET Programming in C++

How to: Retrieve the Windows Version

The following code example demonstrates how to retrieve the platform and version information of the current operating system. This information is stored in the System.Environment.OSVersion property and consists of an enumeration that describes the version of Windows in broad terms and a Version object that contains the exact build of the operating system.

Example

```
// os_ver.cpp
// compile with: /clr
using namespace System;

int main()
{
   OperatingSystem^ osv = Environment::OSVersion;
   PlatformID id = osv->Platform;
   Console::Write("Operating system: ");

   if (id == PlatformID::Win32NT)
      Console::WriteLine("Win32NT");
   else if (id == PlatformID::Win32S)
      Console::WriteLine("Win32S");
   else if (id == PlatformID::Win32Windows)
      Console::WriteLine("Win32Windows");
   else
      Console::WriteLine("WinCE");

   Version^ version = osv->Version;
   if (version)
   {
      int build = version->Build;
      int major = version->Major;
      int minor = version->Minor;
      int revision = Environment::Version->Revision;
      Console::Write("OS Version: ");
      Console::WriteLine("{0}.{1}.{2}.{3}",
                   build, major, minor, revision);
   }

   return 0;
}
```

See Also

Other Resources

Windows Operations in C++
.NET Programming in C++

How to: Retrieve the Local Machine Name

The following code example demonstrates the retrieval of the local machine name (the name of the computer as it appears on a network). You can accomplish this by getting the MachineName string, which is defined in the Environment namespace.

Example

```
// machine_name.cpp
// compile with: /clr
using namespace System;

int main()
{
   Console::WriteLine("\nMachineName: {0}", Environment::MachineName);
   return 0;
}
```

See Also

Other Resources

Windows Operations in C++
.NET Programming in C++

How to: Retrieve the Current Username

The following code example demonstrates the retrieval of the current user name (the name of the user logged into Windows). The name is stored in the UserName string, which is defined in the Environment namespace.

Example

```
// username.cpp
// compile with: /clr
using namespace System;

int main()
{
   Console::WriteLine("\nCurrent user: {0}", Environment::UserName);
   return 0;
}
```

See Also

Other Resources

Windows Operations in C++
.NET Programming in C++

How to: Retrieve the .NET Framework Version

The following code example demonstrates how to determine the version of the currently installed .NET Framework with the Version property, which is a pointer to a Version object that contains the version information.

Example

```
// dotnet_ver.cpp
// compile with: /clr
using namespace System;
int main()
{
   Version^ version = Environment::Version;
   if (version)
   {
      int build = version->Build;
      int major = version->Major;
      int minor = version->Minor;
      int revision = Environment::Version->Revision;
      Console::Write(".NET Framework version: ");
      Console::WriteLine("{0}.{1}.{2}.{3}",
            build, major, minor, revision);
   }
   return 0;
}
```

See Also

Other Resources

Windows Operations in C++
.NET Programming in C++

How to: Read Windows Performance Counters

Some applications and Windows subsystems expose performance data through the Windows performance system. These counters can be accessed using the PerformanceCounterCategory and PerformanceCounter classes, which reside in the System.Diagnostics namespace.

The following code example uses these classes to retrieve and display a counter that is updated by Windows to indicate the percentage of time that the processor is busy.

Example

```
// processor_timer.cpp
// compile with: /clr
#using <system.dll>

using namespace System;
using namespace System::Threading;
using namespace System::Diagnostics;
using namespace System::Timers;

ref struct TimerObject
{
public:
   static String^ m_instanceName;
   static PerformanceCounter^ m_theCounter;

public:
   static void OnTimer(Object^ source, ElapsedEventArgs^ e)
   {
      try
      {
         Console::WriteLine("CPU time used: {0,6} ",
          m_theCounter->NextValue( ).ToString("f"));
      }
      catch(Exception^ e)
      {
         if (dynamic_cast<InvalidOperationException^>(e))
         {
            Console::WriteLine("Instance '{0}' does not exist",
                  m_instanceName);
            return;
         }
         else
         {
            Console::WriteLine("Unknown exception... ('q' to quit)");
            return;
         }
      }
   }
};

int main()
{
   String^ objectName = "Processor";
   String^ counterName = "% Processor Time";
   String^ instanceName = "_Total";

   if ( !PerformanceCounterCategory::Exists(objectName) )
   {
      Console::WriteLine("Object {0} does not exist", objectName);
      return -1;
   }

   if ( !PerformanceCounterCategory::CounterExists(
          counterName, objectName) )
   {
```

```
        Console::WriteLine("Counter {0} does not exist", counterName);
        return -1;
    }

    TimerObject::m_instanceName = instanceName;
    TimerObject::m_theCounter = gcnew PerformanceCounter(
           objectName, counterName, instanceName);

    System::Timers::Timer^ aTimer = gcnew System::Timers::Timer();
    aTimer->Elapsed += gcnew ElapsedEventHandler(&TimerObject::OnTimer);
    aTimer->Interval = 1000;
    aTimer->Enabled = true;
    aTimer->AutoReset = true;

    Console::WriteLine("reporting CPU usage for the next 10 seconds");
    Thread::Sleep(10000);
    return 0;
}
```

See Also

Concepts

Introduction to Monitoring Performance Thresholds

Other Resources

Windows Operations in C++

.NET Programming in C++

How to: Determine if Shutdown Has Started

The following code example demonstrates how to determine whether the application or the .NET Framework is currently terminating. This is useful for accessing static elements in the .NET Framework because, during shutdown, these constructs are finalized by the system and cannot be reliably used. By checking the HasShutdownStarted property first, you can avoid potential failures by not accessing these elements.

Example

```
// check_shutdown.cpp
// compile with: /clr
using namespace System;
int main()
{
   if (Environment::HasShutdownStarted)
      Console::WriteLine("Shutting down.");
   else
      Console::WriteLine("Not shutting down.");
   return 0;
}
```

See Also

Other Resources

Windows Operations in C++
.NET Programming in C++

How to: Determine the User Interactive State

The following code example demonstrates how to determine whether code is being run in a user-interactive context. If UserInteractive is false, then the code is running as a service process or from inside a Web application, in which case you should not attempt to interact with the user.

Example

```
// user_interactive.cpp
// compile with: /clr
using namespace System;

int main()
{
   if ( Environment::UserInteractive )
      Console::WriteLine("User interactive");
   else
      Console::WriteLine("Noninteractive");
   return 0;
}
```

See Also

Other Resources

Windows Operations in C++
.NET Programming in C++

Data Access Using ADO.NET in C++

ADO.NET is the .NET Framework API for data access and provides power and ease of use unmatched by previous data access solutions. This section describes some of the issues involving ADO.NET that are unique to Visual C++ users, such as marshaling native types.

ADO.NET runs under the Common Language Runtime (CLR). Therefore, any application that interacts with ADO.NET must also target the CLR. However, that does not mean that native applications cannot use ADO.NET. These examples will demonstrate how to interact with an ADO.NET database from native code.

In This Section

How to: Marshal ANSI Strings for ADO.NET

How to: Marshal BSTR Strings for ADO.NET

How to: Marshal Unicode Strings for ADO.NET

How to: Marshal a VARIANT for ADO.NET

How to: Marshal a SAFEARRAY for ADO.NET

Related Sections

Section	Description
ADO.NET	Provides an overview of ADO.NET, a set of classes that expose data acces s services to the .NET programmer.
Creating SQL Server 2005 Objects in Managed Code	Describes how to use .NET languages, including Visual C++, to create dat abase objects such as stored procedures, aggregates, triggers, user-defin ed functions, and user-defined types; and to retrieve and update data for Microsoft SQL Server 2005 databases.

See Also

Other Resources

.NET Programming in C++
Native and .NET Interoperability

How to: Marshal ANSI Strings for ADO.NET

Demonstrates how to add a native string (**char ***) to a database and how to marshal a System.String from a database to a native string.

Example

In this example, the class DatabaseClass is created to interact with an ADO.NET DataTable object. Note that this class is a native C++ **class** (as compared to a **ref class** or **value class**). This is necessary because we want to use this class from native code, and you cannot use managed types in native code. This class will be compiled to target the CLR, as is indicated by the `#pragma managed` directive preceding the class declaration. For more information on this directive, see managed, unmanaged.

Note the private member of the DatabaseClass class: `gcroot<DataTable ^> table`. Since native types cannot contain managed types, the **gcroot** keyword is necessary. For more information on **gcroot**, see How to: Declare Handles in Native Types.

The rest of the code in this example is native C++ code, as is indicated by the `#pragma unmanaged` directive preceding `main`. In this example, we are creating a new instance of DatabaseClass and calling its methods to create a table and populate some rows in the table. Note that native C++ strings are being passed as values for the database column StringCol. Inside DatabaseClass, these strings are marshaled to managed strings using the marshaling functionality found in the System.Runtime.InteropServices namespace. Specifically, the method PtrToStringAnsi is used to marshal a **char *** to a **String**, and the method StringToHGlobalAnsi is used to marshal a **String** to a **char ***.

Note
The memory allocated by **StringToHGlobalAnsi** must be deallocated by calling either FreeHGlobal or **GlobalFree**.

```
// adonet_marshal_string_native.cpp
// compile with: /clr /FU System.dll /FU System.Data.dll /FU System.Xml.dll
#include <comdef.h>
#include <gcroot.h>
#include <iostream>
using namespace std;

#using <System.Data.dll>
using namespace System;
using namespace System::Data;
using namespace System::Runtime::InteropServices;

#define MAXCOLS 100

#pragma managed
class DatabaseClass
{
public:
    DatabaseClass() : table(nullptr) { }

    void AddRow(char *stringColValue)
    {
        // Add a row to the table.
        DataRow ^row = table->NewRow();
        row["StringCol"] = Marshal::PtrToStringAnsi(
            (IntPtr)stringColValue);
        table->Rows->Add(row);
    }

    void CreateAndPopulateTable()
    {
        // Create a simple DataTable.
        table = gcnew DataTable("SampleTable");

        // Add a column of type String to the table.
        DataColumn ^column1 = gcnew DataColumn("StringCol",
            Type::GetType("System.String"));
        table->Columns->Add(column1);
    }
```

```
    int GetValuesForColumn(char *dataColumn, char **values,
        int valuesLength)
    {
        // Marshal the name of the column to a managed
        // String.
        String ^columnStr = Marshal::PtrToStringAnsi(
                (IntPtr)dataColumn);

        // Get all rows in the table.
        array<DataRow ^> ^rows = table->Select();
        int len = rows->Length;
        len = (len > valuesLength) ? valuesLength : len;
        for (int i = 0; i < len; i++)
        {
            // Marshal each column value from a managed string
            // to a char *.
            values[i] = (char *)Marshal::StringToHGlobalAnsi(
                (String ^)rows[i][columnStr]).ToPointer();
        }

        return len;
    }

private:
    // Using gcroot, you can use a managed type in
    // a native class.
    gcroot<DataTable ^> table;
};

#pragma unmanaged
int main()
{
    // Create a table and add a few rows to it.
    DatabaseClass *db = new DatabaseClass();
    db->CreateAndPopulateTable();
    db->AddRow("This is string 1.");
    db->AddRow("This is string 2.");

    // Now retrieve the rows and display their contents.
    char *values[MAXCOLS];
    int len = db->GetValuesForColumn(
        "StringCol", values, MAXCOLS);
    for (int i = 0; i < len; i++)
    {
        cout << "StringCol: " << values[i] << endl;

        // Deallocate the memory allocated using
        // Marshal::StringToHGlobalAnsi.
        GlobalFree(values[i]);
    }

    delete db;

    return 0;
}
```

Output

```
StringCol: This is string 1.
StringCol: This is string 2.
```

Compiling the Code

- To compile the code from the command line, save the code example in a file named adonet_marshal_string_native.cpp and enter the following statement:

```
cl /clr /FU System.dll /FU System.Data.dll /FU System.Xml.dll adonet_marshal_string_na
tive.cpp
```

Security

For information on security issues involving ADO.NET, see Securing ADO.NET Applications.

See Also

Reference

System.Runtime.InteropServices

Other Resources

Data Access Using ADO.NET in C++

ADO.NET

Interoperability

Native and .NET Interoperability

How to: Marshal BSTR Strings for ADO.NET

Demonstrates how to add a COM string (**BSTR**) to a database and how to marshal a System.String from a database to a **BSTR**.

Example

In this example, the class DatabaseClass is created to interact with an ADO.NET DataTable object. Note that this class is a native C++ **class** (as compared to a **ref class** or **value class**). This is necessary because we want to use this class from native code, and you cannot use managed types in native code. This class will be compiled to target the CLR, as is indicated by the `#pragma managed` directive preceding the class declaration. For more information on this directive, see managed, unmanaged.

Note the private member of the DatabaseClass class: `gcroot<DataTable ^> table`. Since native types cannot contain managed types, the **gcroot** keyword is necessary. For more information on **gcroot**, see How to: Declare Handles in Native Types.

The rest of the code in this example is native C++ code, as is indicated by the `#pragma unmanaged` directive preceding `main`. In this example, we are creating a new instance of DatabaseClass and calling its methods to create a table and populate some rows in the table. Note that COM strings are being passed as values for the database column StringCol. Inside DatabaseClass, these strings are marshaled to managed strings using the marshaling functionality found in the System.Runtime.InteropServices namespace. Specifically, the method PtrToStringBSTR is used to marshal a **BSTR** to a **String**, and the method StringToBSTR is used to marshal a **String** to a **BSTR**.

Note
The memory allocated by **StringToBSTR** must be deallocated by calling either FreeBSTR or **SysFreeString**.

```
// adonet_marshal_string_bstr.cpp
// compile with: /clr /FU System.dll /FU System.Data.dll /FU System.Xml.dll
#include <comdef.h>
#include <gcroot.h>
#include <iostream>
using namespace std;

#using <System.Data.dll>
using namespace System;
using namespace System::Data;
using namespace System::Runtime::InteropServices;

#define MAXCOLS 100

#pragma managed
class DatabaseClass
{
public:
    DatabaseClass() : table(nullptr) { }

    void AddRow(BSTR stringColValue)
    {
        // Add a row to the table.
        DataRow ^row = table->NewRow();
        row["StringCol"] = Marshal::PtrToStringBSTR(
            (IntPtr)stringColValue);
        table->Rows->Add(row);
    }

    void CreateAndPopulateTable()
    {
        // Create a simple DataTable.
        table = gcnew DataTable("SampleTable");

        // Add a column of type String to the table.
        DataColumn ^column1 = gcnew DataColumn("StringCol",
            Type::GetType("System.String"));
        table->Columns->Add(column1);
    }
```

```
    int GetValuesForColumn(BSTR dataColumn, BSTR *values,
        int valuesLength)
    {
        // Marshal the name of the column to a managed
        // String.
        String ^columnStr = Marshal::PtrToStringBSTR(
                (IntPtr)dataColumn);

        // Get all rows in the table.
        array<DataRow ^> ^rows = table->Select();
        int len = rows->Length;
        len = (len > valuesLength) ? valuesLength : len;
        for (int i = 0; i < len; i++)
        {
            // Marshal each column value from a managed string
            // to a BSTR.
            values[i] = (BSTR)Marshal::StringToBSTR(
                (String ^)rows[i][columnStr]).ToPointer();
        }

        return len;
    }

private:
    // Using gcroot, you can use a managed type in
    // a native class.
    gcroot<DataTable ^> table;
};

#pragma unmanaged
int main()
{
    // Create a table and add a few rows to it.
    DatabaseClass *db = new DatabaseClass();
    db->CreateAndPopulateTable();

    BSTR str1 = SysAllocString(L"This is string 1.");
    db->AddRow(str1);

    BSTR str2 = SysAllocString(L"This is string 2.");
    db->AddRow(str2);

    // Now retrieve the rows and display their contents.
    BSTR values[MAXCOLS];
    BSTR str3 = SysAllocString(L"StringCol");
    int len = db->GetValuesForColumn(
        str3, values, MAXCOLS);
    for (int i = 0; i < len; i++)
    {
        wcout << "StringCol: " << values[i] << endl;

        // Deallocate the memory allocated using
        // Marshal::StringToBSTR.
        SysFreeString(values[i]);
    }

    SysFreeString(str1);
    SysFreeString(str2);
    SysFreeString(str3);
    delete db;

    return 0;
}
```

Output

```
StringCol: This is string 1.
StringCol: This is string 2.
```

Compiling the Code

- To compile the code from the command line, save the code example in a file named adonet_marshal_string_native.cpp and enter the following statement:

```
cl /clr /FU System.dll /FU System.Data.dll /FU System.Xml.dll adonet_marshal_string_na
tive.cpp
```

Security

For information on security issues involving ADO.NET, see Securing ADO.NET Applications.

See Also

Reference

System.Runtime.InteropServices

Other Resources

Data Access Using ADO.NET in C++

ADO.NET

Interoperability

Native and .NET Interoperability

How to: Marshal Unicode Strings for ADO.NET

Demonstrates how to add a native Unicode string (**wchar_t ***) to a database and how to marshal a System.String from a database to a native Unicode string.

Example

In this example, the class DatabaseClass is created to interact with an ADO.NET DataTable object. Note that this class is a native C++ **class** (as compared to a **ref class** or **value class**). This is necessary because we want to use this class from native code, and you cannot use managed types in native code. This class will be compiled to target the CLR, as is indicated by the `#pragma managed` directive preceding the class declaration. For more information on this directive, see managed, unmanaged.

Note the private member of the DatabaseClass class: `gcroot<DataTable ^> table`. Since native types cannot contain managed types, the **gcroot** keyword is necessary. For more information on **gcroot**, see How to: Declare Handles in Native Types.

The rest of the code in this example is native C++ code, as is indicated by the `#pragma unmanaged` directive preceding `main`. In this example, we are creating a new instance of DatabaseClass and calling its methods to create a table and populate some rows in the table. Note that Unicode C++ strings are being passed as values for the database column StringCol. Inside DatabaseClass, these strings are marshaled to managed strings using the marshaling functionality found in the System.Runtime.InteropServices namespace. Specifically, the method PtrToStringUni is used to marshal a **wchar_t *** to a **String**, and the method StringToHGlobalUni is used to marshal a **String** to a **wchar_t ***.

Note
The memory allocated by **StringToHGlobalUni** must be deallocated by calling either FreeHGlobal or **GlobalFree**.

```
// adonet_marshal_string_wide.cpp
// compile with: /clr /FU System.dll /FU System.Data.dll /FU System.Xml.dll
#include <comdef.h>
#include <gcroot.h>
#include <iostream>
using namespace std;

#using <System.Data.dll>
using namespace System;
using namespace System::Data;
using namespace System::Runtime::InteropServices;

#define MAXCOLS 100

#pragma managed
class DatabaseClass
{
public:
    DatabaseClass() : table(nullptr) { }

    void AddRow(wchar_t *stringColValue)
    {
        // Add a row to the table.
        DataRow ^row = table->NewRow();
        row["StringCol"] = Marshal::PtrToStringUni(
            (IntPtr)stringColValue);
        table->Rows->Add(row);
    }

    void CreateAndPopulateTable()
    {
        // Create a simple DataTable.
        table = gcnew DataTable("SampleTable");

        // Add a column of type String to the table.
        DataColumn ^column1 = gcnew DataColumn("StringCol",
            Type::GetType("System.String"));
        table->Columns->Add(column1);
    }
```

```
    int GetValuesForColumn(wchar_t *dataColumn, wchar_t **values,
        int valuesLength)
    {
        // Marshal the name of the column to a managed
        // String.
        String ^columnStr = Marshal::PtrToStringUni(
                (IntPtr)dataColumn);

        // Get all rows in the table.
        array<DataRow ^> ^rows = table->Select();
        int len = rows->Length;
        len = (len > valuesLength) ? valuesLength : len;
        for (int i = 0; i < len; i++)
        {
            // Marshal each column value from a managed string
            // to a wchar_t *.
            values[i] = (wchar_t *)Marshal::StringToHGlobalUni(
                (String ^)rows[i][columnStr]).ToPointer();
        }

        return len;
    }

private:
    // Using gcroot, you can use a managed type in
    // a native class.
    gcroot<DataTable ^> table;
};

#pragma unmanaged
int main()
{
    // Create a table and add a few rows to it.
    DatabaseClass *db = new DatabaseClass();
    db->CreateAndPopulateTable();
    db->AddRow(L"This is string 1.");
    db->AddRow(L"This is string 2.");

    // Now retrieve the rows and display their contents.
    wchar_t *values[MAXCOLS];
    int len = db->GetValuesForColumn(
        L"StringCol", values, MAXCOLS);
    for (int i = 0; i < len; i++)
    {
        wcout << "StringCol: " << values[i] << endl;

        // Deallocate the memory allocated using
        // Marshal::StringToHGlobalUni.
        GlobalFree(values[i]);
    }

    delete db;

    return 0;
}
```

Output

```
StringCol: This is string 1.
StringCol: This is string 2.
```

Compiling the Code

- To compile the code from the command line, save the code example in a file named adonet_marshal_string_wide.cpp and enter the following statement:

```
cl /clr /FU System.dll /FU System.Data.dll /FU System.Xml.dll adonet_marshal_string_wi
de.cpp
```

Security

For information on security issues involving ADO.NET, see Securing ADO.NET Applications.

See Also

Reference

System.Runtime.InteropServices

Other Resources

Data Access Using ADO.NET in C++

ADO.NET

Interoperability

Native and .NET Interoperability

How to: Marshal a VARIANT for ADO.NET

Demonstrates how to add a native **VARIANT** to a database and how to marshal a System.Object from a database to a native **VARIANT**.

Example

In this example, the class DatabaseClass is created to interact with an ADO.NET DataTable object. Note that this class is a native C++ **class** (as compared to a **ref class** or **value class**). This is necessary because we want to use this class from native code, and you cannot use managed types in native code. This class will be compiled to target the CLR, as is indicated by the `#pragma managed` directive preceding the class declaration. For more information on this directive, see managed, unmanaged.

Note the private member of the DatabaseClass class: `gcroot<DataTable ^> table`. Since native types cannot contain managed types, the **gcroot** keyword is necessary. For more information on **gcroot**, see How to: Declare Handles in Native Types.

The rest of the code in this example is native C++ code, as is indicated by the `#pragma unmanaged` directive preceding `main`. In this example, we are creating a new instance of DatabaseClass and calling its methods to create a table and populate some rows in the table. Note that native **VARIANT** types are being passed as values for the database column ObjectCol. Inside DatabaseClass, these **VARIANT** types are marshaled to managed objects using the marshaling functionality found in the System.Runtime.InteropServices namespace. Specifically, the method GetObjectForNativeVariant is used to marshal a **VARIANT** to an **Object**, and the method GetNativeVariantForObject is used to marshal an **Object** to a **VARIANT**.

```
// adonet_marshal_variant.cpp
// compile with: /clr /FU System.dll /FU System.Data.dll /FU System.Xml.dll
#include <comdef.h>
#include <gcroot.h>
#include <iostream>
using namespace std;

#using <System.Data.dll>
using namespace System;
using namespace System::Data;
using namespace System::Runtime::InteropServices;

#define MAXCOLS 100

#pragma managed
class DatabaseClass
{
public:
    DatabaseClass() : table(nullptr) { }

    void AddRow(VARIANT *objectColValue)
    {
        // Add a row to the table.
        DataRow ^row = table->NewRow();
        row["ObjectCol"] = Marshal::GetObjectForNativeVariant(
            IntPtr(objectColValue));
        table->Rows->Add(row);
    }

    void CreateAndPopulateTable()
    {
        // Create a simple DataTable.
        table = gcnew DataTable("SampleTable");

        // Add a column of type String to the table.
        DataColumn ^column1 = gcnew DataColumn("ObjectCol",
            Type::GetType("System.Object"));
        table->Columns->Add(column1);
    }

    int GetValuesForColumn(wchar_t *dataColumn, VARIANT *values,
        int valuesLength)
    {
```

```
        // Marshal the name of the column to a managed
        // String.
        String ^columnStr = Marshal::PtrToStringUni(
                (IntPtr)dataColumn);

        // Get all rows in the table.
        array<DataRow ^> ^rows = table->Select();
        int len = rows->Length;
        len = (len > valuesLength) ? valuesLength : len;
        for (int i = 0; i < len; i++)
        {
            // Marshal each column value from a managed object
            // to a VARIANT.
            Marshal::GetNativeVariantForObject(
                rows[i][columnStr], IntPtr(&values[i]));
        }

        return len;
    }

private:
    // Using gcroot, you can use a managed type in
    // a native class.
    gcroot<DataTable ^> table;
};

#pragma unmanaged
int main()
{
    // Create a table and add a few rows to it.
    DatabaseClass *db = new DatabaseClass();
    db->CreateAndPopulateTable();

    BSTR bstr1 = SysAllocString(L"This is a BSTR in a VARIANT.");
    VARIANT v1;
    v1.vt = VT_BSTR;
    v1.bstrVal = bstr1;
    db->AddRow(&v1);

    int i = 42;
    VARIANT v2;
    v2.vt = VT_I4;
    v2.lVal = i;
    db->AddRow(&v2);

    // Now retrieve the rows and display their contents.
    VARIANT values[MAXCOLS];
    int len = db->GetValuesForColumn(
        L"ObjectCol", values, MAXCOLS);
    for (int i = 0; i < len; i++)
    {
        switch (values[i].vt)
        {
            case VT_BSTR:
                wcout << L"ObjectCol: " << values[i].bstrVal << endl;
                break;
            case VT_I4:
                cout << "ObjectCol: " << values[i].lVal << endl;
                break;
            default:
                break;
        }

    }

    SysFreeString(bstr1);
    delete db;
```

```
    return 0;
}
```

Output

```
ObjectCol: This is a BSTR in a VARIANT.
ObjectCol: 42
```

Compiling the Code

- To compile the code from the command line, save the code example in a file named adonet_marshal_variant.cpp and enter the following statement:

```
cl /clr /FU System.dll /FU System.Data.dll /FU System.Xml.dll adonet_marshal_variant.c
pp
```

Security

For information on security issues involving ADO.NET, see Securing ADO.NET Applications.

See Also

Reference

System.Runtime.InteropServices

Other Resources

Data Access Using ADO.NET in C++

ADO.NET

Interoperability

Native and .NET Interoperability

How to: Marshal a SAFEARRAY for ADO.NET

Demonstrates how to add a native **SAFEARRAY** to a database and how to marshal a managed array from a database to a native **SAFEARRAY**.

Example

In this example, the class DatabaseClass is created to interact with an ADO.NET DataTable object. Note that this class is a native C++ **class** (as compared to a **ref class** or **value class**). This is necessary because we want to use this class from native code, and you cannot use managed types in native code. This class will be compiled to target the CLR, as is indicated by the `#pragma managed` directive preceding the class declaration. For more information on this directive, see managed, unmanaged.

Note the private member of the DatabaseClass class: `gcroot<DataTable ^> table`. Since native types cannot contain managed types, the **gcroot** keyword is necessary. For more information on **gcroot**, see How to: Declare Handles in Native Types.

The rest of the code in this example is native C++ code, as is indicated by the `#pragma unmanaged` directive preceding `main`. In this example, we are creating a new instance of DatabaseClass and calling its methods to create a table and populate some rows in the table. Note that native **SAFEARRAY** types are being passed as values for the database column ArrayIntsCol. Inside DatabaseClass, these **SAFEARRAY** types are marshaled to managed objects using the marshaling functionality found in the System.Runtime.InteropServices namespace. Specifically, the method Copy is used to marshal a **SAFEARRAY** to a managed array of integers, and the method Copy is used to marshal a managed array of integers to a **SAFEARRAY**.

```
// adonet_marshal_safearray.cpp
// compile with: /clr /FU System.dll /FU System.Data.dll /FU System.Xml.dll
#include <comdef.h>
#include <gcroot.h>
#include <iostream>
using namespace std;

#using <System.Data.dll>
using namespace System;
using namespace System::Data;
using namespace System::Runtime::InteropServices;

#define MAXCOLS 100

#pragma managed
class DatabaseClass
{
public:
    DatabaseClass() : table(nullptr) { }

    void AddRow(SAFEARRAY *arrayIntsColValue)
    {
        // Add a row to the table.
        DataRow ^row = table->NewRow();
        int len = arrayIntsColValue->rgsabound[0].cElements;
        array<int> ^arr = gcnew array<int>(len);

        int *pData;
        SafeArrayAccessData(arrayIntsColValue, (void **)&pData);
        Marshal::Copy(IntPtr(pData), arr, 0, len);
        SafeArrayUnaccessData(arrayIntsColValue);

        row["ArrayIntsCol"] = arr;
        table->Rows->Add(row);
    }

    void CreateAndPopulateTable()
    {
        // Create a simple DataTable.
        table = gcnew DataTable("SampleTable");

        // Add a column of type String to the table.
        DataColumn ^column1 = gcnew DataColumn("ArrayIntsCol",
```

```
            Type::GetType("System.Int32[]"));
        table->Columns->Add(column1);
    }

    int GetValuesForColumn(wchar_t *dataColumn, SAFEARRAY **values,
        int valuesLength)
    {
        // Marshal the name of the column to a managed
        // String.
        String ^columnStr = Marshal::PtrToStringUni(
                (IntPtr)dataColumn);

        // Get all rows in the table.
        array<DataRow ^> ^rows = table->Select();
        int len = rows->Length;
        len = (len > valuesLength) ? valuesLength : len;
        for (int i = 0; i < len; i++)
        {
            // Marshal each column value from a managed array
            // of Int32s to a SAFEARRAY of type VT_I4.
            values[i] = SafeArrayCreateVector(VT_I4, 0, 10);
            int *pData;
            SafeArrayAccessData(values[i], (void **)&pData);
            Marshal::Copy((array<int> ^)rows[i][columnStr], 0,
                IntPtr(pData), 10);
            SafeArrayUnaccessData(values[i]);
        }

        return len;
    }

private:
    // Using gcroot, you can use a managed type in
    // a native class.
    gcroot<DataTable ^> table;
};

#pragma unmanaged
int main()
{
    // Create a table and add a few rows to it.
    DatabaseClass *db = new DatabaseClass();
    db->CreateAndPopulateTable();

    // Create a standard array.
    int originalArray[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    // Create a SAFEARRAY.
    SAFEARRAY *psa;
    psa = SafeArrayCreateVector(VT_I4, 0, 10);

    // Copy the data from the original array to the SAFEARRAY.
    int *pData;
    HRESULT hr = SafeArrayAccessData(psa, (void **)&pData);
    memcpy(pData, &originalArray, 40);
    SafeArrayUnaccessData(psa);
    db->AddRow(psa);

    // Now retrieve the rows and display their contents.
    SAFEARRAY *values[MAXCOLS];
    int len = db->GetValuesForColumn(
        L"ArrayIntsCol", values, MAXCOLS);
    for (int i = 0; i < len; i++)
    {
        int *pData;
        SafeArrayAccessData(values[i], (void **)&pData);
        for (int j = 0; j < 10; j++)
        {
```

```
            cout << pData[j] << " ";
        }
        cout << endl;
        SafeArrayUnaccessData(values[i]);

        // Deallocate the memory allocated using
        // SafeArrayCreateVector.
        SafeArrayDestroy(values[i]);
    }

    SafeArrayDestroy(psa);
    delete db;

    return 0;
}
```

Output

```
0 1 2 3 4 5 6 7 8 9
```

Compiling the Code

- To compile the code from the command line, save the code example in a file named adonet_marshal_safearray.cpp and enter the following statement:

  ```
  cl /clr /FU System.dll /FU System.Data.dll /FU System.Xml.dll adonet_marshal_safearray
  .cpp
  ```

Security

For information on security issues involving ADO.NET, see Securing ADO.NET Applications.

See Also

Reference

System.Runtime.InteropServices

Other Resources

Data Access Using ADO.NET in C++

ADO.NET

Interoperability

Native and .NET Interoperability

Interoperability with Other .NET Languages in C++

The topics in this section show how to create assemblies in Visual C++ that consume from or provide functionality to assemblies written in C# or Visual Basic.

In This Section

How to: Implement is and as C# Keywords in C++

How to: Implement the lock C# Keyword in C++

How to: Consume a C# Indexer

See Also

Other Resources

.NET Programming in C++

How to: Implement is and as C# Keywords in C++

This topic shows how to implement the functionality of the **is** and **as** C# keywords in Visual C++.

For more information, see is (C# Reference) and as (C# Reference).

Example

```
// CS_is_as.cpp
// compile with: /clr
using namespace System;

interface class I {
public:
   void F();
};

ref struct C : public I {
   virtual void F( void ) { }
};

template < class T, class U >
Boolean isinst(U u) {
   return dynamic_cast< T >(u) != nullptr;
}

int main() {
   C ^ c = gcnew C();
   I ^ i = safe_cast< I ^ >(c);   // is (maps to castclass in IL)
   I ^ ii = dynamic_cast< I ^ >(c);   // as (maps to isinst in IL)

   // simulate 'as':
   Object ^ o = "f";
   if ( isinst< String ^ >(o) )
      Console::WriteLine("o is a string");
}
```

Output

```
o is a string
```

See Also

Other Resources

Interoperability with Other .NET Languages in C++

How to: Implement the lock C# Keyword in C++

This topic shows how to implement the C# **lock** keyword in Visual C++. For more information, see lock Statement (C# Reference).

You can also use the **lock** class in the C++ Support Library. See Synchronization (lock Class) for more information.

Example

```
// CS_lock_in_CPP.cpp
// compile with: /clr /c
using namespace System::Threading;
ref class Lock {
   Object^ m_pObject;
   Lock % operator=( Lock const % );
   Lock( Lock const % );
public:
   Lock( Object ^ pObject ) : m_pObject( pObject ) {
      Monitor::Enter( m_pObject );
   }
   ~Lock() {
      Monitor::Exit( m_pObject );
   }
};

ref struct LockHelper {
   void DoSomething();
};

void LockHelper::DoSomething() {
   // Note: Reference type with stack allocation semantics to provide
   // deterministic finalization

   Lock lock( this );
   // LockHelper instance is locked
}
```

See Also

Other Resources

Interoperability with Other .NET Languages in C++

How to: Consume a C# Indexer

Visual C++ does not contain indexers; it has indexed properties. To consume a C# indexer, access the indexer as if it were an indexed property.

For more information about indexers, see:

- Indexers (C# Programming Guide)
- How to: Use Indexed Properties

Example

The following C# program defines an indexer.

```
// consume_cs_indexers.cs
// compile with: /target:library
using System;
public class IndexerClass {
   private int [] myArray = new int[100];
   public int this [int index] {   // Indexer declaration
      get {
         // Check the index limits.
         if (index < 0 || index >= 100)
            return 0;
         else
            return myArray[index];
      }
      set {
         if (!(index < 0 || index >= 100))
            myArray[index] = value;
      }
   }
}
/*
// code to consume the indexer
public class MainClass {
   public static void Main() {
      IndexerClass b = new IndexerClass();

      // Call indexer to initialize elements 3 and 5
      b[3] = 256;
      b[5] = 1024;
      for (int i = 0 ; i <= 10 ; i++)
         Console.WriteLine("Element #{0} = {1}", i, b[i]);
   }
}
*/
```

This Visual C++ program consumes the indexer.

```
// consume_cs_indexers_2.cpp
// compile with: /clr
#using "consume_cs_indexers.dll"
using namespace System;

int main() {
   IndexerClass ^ ic = gcnew IndexerClass;
   ic->default[0] = 21;
   for (int i = 0 ; i <= 10 ; i++)
      Console::WriteLine("Element #{0} = {1}", i, ic->default[i]);
}
```

Output

```
Element #0 = 21
Element #1 = 0
Element #2 = 0
Element #3 = 0
Element #4 = 0
Element #5 = 0
Element #6 = 0
Element #7 = 0
Element #8 = 0
Element #9 = 0
Element #10 = 0
```

See Also

Other Resources

Interoperability with Other .NET Languages in C++

Serialization in C++

Serialization (the process of storing the state of an object or member to a permanent medium) of managed classes (including individual fields or properties) is supported by the SerializableAttribute and NonSerializedAttribute classes.

Remarks

Apply the **SerializableAttribute** custom attribute to a managed class to serialize the entire class or apply only to specific fields or properties to serialize parts of the managed class. Use the **NonSerializedAttribute** custom attribute to exempt fields or properties of a managed class from being serialized.

Example

In the following example, the class `MyClass` (and the property `m_nCount`) is marked as serializable. However, the `m_nData` property is not serialized as indicated by the **NonSerialized** custom attribute:

```
// serialization_and_mcpp.cpp
// compile with: /LD /clr
using namespace System;

[ Serializable ]
public ref class MyClass {
public:
   int m_nCount;
private:
   [ NonSerialized ]
   int m_nData;
};
```

Note that both attributes can be referenced using their "short name" (**Serializable** and **NonSerialized**). This is further explained in Applying Attributes.

See Also

Other Resources

.NET Programming in C++

Managed Types

Visual C++ allows access to .NET features through managed types, which provide support for features of the common language runtime and are subject to the advantages and restrictions of the runtime.

In This Section

- Managed Types and the main Function
- .NET Framework Equivalents to C++ Native Types
- Version Issues for Value Types Nested in Native Types
- How to: Test for Equality
- How to: Diagnose and Fix Assembly Compatibility Problems

See Also

Other Resources

.NET Programming in C++

Managed Types and the main Function

When writing an application using **/clr**, the arguments of the **main()** function cannot be of a managed type.

An example of a proper signature is:

```
// managed_types_and_main.cpp
// compile with: /clr
int main(int, char*[], char*[]) {}
```

See Also

Other Resources

Managed Types

.NET Framework Equivalents to C++ Native Types

The following table shows the keywords for built-in Visual C++ types, which are aliases of predefined types in the **System** namespace.

Visual C++ type	.NET Framework type
bool	**System.Boolean**
signed char (see /J for more information)	**System.SByte**
unsigned char	**System.Byte**
wchar_t	**System.Char**
double and **long double**	**System.Double**
float	**System.Single**
int, **signed int**, **long**, and **signed long**	**System.Int32**
unsigned int and **unsigned long**	**System.UInt32**
__int64 and **signed __int64**	**System.Int64**
unsigned __int64	**System.UInt64**
short and **signed short**	**System.Int16**
unsigned short	**System.UInt16**
void	**System.Void**

See Also

Other Resources

Managed Types

Version Issues for Value Types Nested in Native Types

Consider a signed (strong name) assembly component used to build a client assembly. The component contains a value type that is used in the client as the type for a member of a native union, a class, or an array. If a future version of the component changes the size or layout of the value type, the client must be recompiled.

Create a keyfile with sn.exe (`sn -k mykey.snk`).

Example

The following sample is the component.

```
// nested_value_types.cpp
// compile with: /clr /LD
using namespace System::Reflection;
[assembly:AssemblyVersion("1.0.0.*"),
assembly:AssemblyKeyFile("mykey.snk")];

public value struct S {
   int i;
   void Test() {
      System::Console::WriteLine("S.i = {0}", i);
   }
};
```

This sample is the client:

```
// nested_value_types_2.cpp
// compile with: /clr
#using <nested_value_types.dll>

struct S2 {
   S MyS1, MyS2;
};

int main() {
   S2 MyS2a, MyS2b;
   MyS2a.MyS1.i = 5;
   MyS2a.MyS2.i = 6;
   MyS2b.MyS1.i = 10;
   MyS2b.MyS2.i = 11;

   MyS2a.MyS1.Test();
   MyS2a.MyS2.Test();
   MyS2b.MyS1.Test();
   MyS2b.MyS2.Test();
}
```

Output

```
S.i = 5
S.i = 6
S.i = 10
S.i = 11
```

However, if you add another member to `struct S` in nested_value_types.cpp, (for example, `double d;`) and recompile the component without recompiling the client, the result is an unhandled exception (of type System.IO.FileLoadException).

See Also

Other Resources

Managed Types

How to: Test for Equality

In the following sample, a test for equality using Managed Extensions for C++ is based on what the pointers point to.

For more information, see Breaking Changes in the Visual C++ 2005 Compiler.

Example

```
// mcppv2_equality_test.cpp
// compile with: /clr:oldSyntax /LD
using namespace System;

bool Test1() {
   String * str1 = S"test";
   String * str2 = S"test";
   return (str1 == str2);
}
```

The IL for this program shows that the return value is implemented with this instruction:

```
  IL_0012:  ceq
```

which compares the addresses of the two String objects.

Using the new syntax,

```
// mcppv2_equality_test_2.cpp
// compile with: /clr /LD
using namespace System;

bool Test1() {
   String ^ str1 = "test";
   String ^ str2 = "test";
   return (str1 == str2);
}
```

The IL for this program shows that the return value is implemented with a call to op_Equality.

```
  IL_0012:  call       bool [mscorlib]System.String::op_Equality(string,
                                                                 string)
```

See Also

Other Resources

Managed Types

How to: Diagnose and Fix Assembly Compatibility Problems

This topic explains what can happen when the version of an assembly referenced at compile time doesn't match the version of the assembly referenced at runtime, and how to avoid the problem.

When an assembly is compiled, other assemblies may be referenced with the **#using** syntax. During the compilation, these assemblies are accessed by the compiler. Information from these assemblies is used to make optimization decisions.

However, if the referenced assembly is changed and recompiled, and you do not recompile the referencing assembly that is dependent on it, the assemblies might not still be compatible. Optimization decisions that were valid at first might not be correct with respect to the new assembly version. Various runtime errors might occur due to these incompatibilities. There is no specific exception that will be produced in such cases. The way the failure is reported at runtime depends on the nature of the code change that caused the problem.

These errors should not be a problem in your final production code as long as the entire application is rebuilt for the released version of your product. Assemblies that are released to the public should be marked with an official version number, which will ensure that these problems are avoided. For more information, see Assembly Versioning.

Diagnosing and fixing an incompatibility error

1. If you encounter runtime exceptions or other error conditions that occur in code that references another assembly, and have no other identified cause, you may be dealing with an out of date assembly.
2. First, isolate and reproduce the exception or other error condition. A problem that occurs due to an outdated exception should be reproducible.
3. Check the timestamp of any assemblies referenced in your application.
4. If the timestamps of any referenced assemblies are later than the timestamp of your application's last compilation, then your application is out of date. If this occurs, recompile your application with the most recent assembly, and make any code changes required.
5. Rerun the application, perform the steps that reproduce the problem, and verify that the exception does not occur.

Example

The following program illustrates the problem by reducing the accessibility of a method, and trying to access that method in another assembly without recompiling. Try compiling `changeaccess.cpp` first. This is the referenced assembly which will change. Then compile `referencing.cpp`. The compilation succeeds. Now, reduce the accessibility of the called method. Recompile `changeaccess.cpp` with the flag `/DCHANGE_ACCESS`. This makes the method protected, rather than private, so it can longer be called legally. Without recompiling `referencing.exe`, rerun the application. An exception MethodAccessException will result.

```
// changeaccess.cpp
// compile with: /clr:safe /LD
// After the initial compilation, add /DCHANGE_ACCESS and rerun
// referencing.exe to introduce an error at runtime. To correct
// the problem, recompile referencing.exe

public ref class Test {
#if defined(CHANGE_ACCESS)
protected:
#else
public:
#endif

  int access_me() {
    return 0;
  }

};
```

```
// referencing.cpp
```

```
// compile with: /clr:safe
#using <changeaccess.dll>

// Force the function to be inline, to override the compiler's own
// algorithm.
__forceinline
int CallMethod(Test^ t) {
  // The call is allowed only if access_me is declared public
  return t->access_me();
}

int main() {
  Test^ t = gcnew Test();
  try {
    CallMethod(t);
  } catch (System::Exception ^ e) {
    System::Console::WriteLine("Exception!");
  }
  System::Console::WriteLine("No exception.");
  return 0;
}
```

See Also

Reference

The #using Directive

Other Resources

Managed Types

Reflection in C++

Reflection allows known data types to be inspected at runtime. Reflection allows the enumeration of data types in a given assembly, and the members of a given class or value type can be discovered. This is true regardless of whether the type was known or referenced at compile time. This makes reflection a useful feature for development and code management tools.

Note that the assembly name provided is the strong name (see Strong-Named Assemblies), which includes the assembly version, culture, and signing information. Note also that the name of the namespace in which the data type is defined can be retrieved, along with the name of the base class.

The most common way to access reflection features is through the GetType method. This method is provided by System::Object, from which all garbage-collected classes derive.

Reflection on an .exe built with the Visual C++ compiler is allowed if the .exe is built with the **/clr:pure** or **/clr:safe** compiler options. See /clr (Common Language Runtime Compilation) for more information.

Topics in this section:

- How to: Implement a Plug-In Component Architecture using Reflection
- How to: Enumerate Data Types in Assemblies using Reflection

For more information, see

- Reflection
- System.Reflection Namespace

Example

The **GetType** method returns a pointer to a Type class object, which describes the type upon when the object is based. (The **Type** object does not contain any instance-specific information.) One such item is the full name of the type, which can be displayed as follows:

Note that the type name includes the full scope in which the type is defined, including the namespace, and that it is displayed in .NET syntax, with a dot as the scope resolution operator.

```
// vcpp_reflection.cpp
// compile with: /clr
using namespace System;
int main() {
   String ^ s = "sample string";
   Console::WriteLine("full type name of '{0}' is '{1}'", s, s->GetType());
}
```

Output

```
full type name of 'sample string' is 'System.String'
```

Value types can be used with the **GetType** function as well, but they must be boxed first.

```
// vcpp_reflection_2.cpp
// compile with: /clr
using namespace System;
int main() {
   Int32 i = 100;
   Object ^ o = i;
   Console::WriteLine("type of i = '{0}'", o->GetType());
}
```

Output

```
type of i = 'System.Int32'
```

As with the **GetType** method, the typeid operator returns a pointer to a **Type** object, so this code indicates the type name **System.Int32**. Displaying type names is the most basic feature of reflection, but a potentially more useful technique is to

inspect or discover the valid values for enumerated types. This can be done by using the static **Enum::GetNames** function, which returns an array of strings, each containing an enumeration value in text form. The following sample retrieves an array of strings that describes the value enumeration values for the **Options** (CLR) enum and displays them in a loop.

If a fourth option is added to the **Options** enumeration, this code will report the new option without recompilation, even if the enumeration is defined in a separate assembly.

```
// vcpp_reflection_3.cpp
// compile with: /clr
using namespace System;

enum class Options {   // not a native enum
   Option1, Option2, Option3
};

int main() {
   array<String^>^ names = Enum::GetNames(Options::typeid);

   Console::WriteLine("there are {0} options in enum '{1}'",
               names->Length, Options::typeid);

   for (int i = 0 ; i < names->Length ; i++)
      Console::WriteLine("{0}: {1}", i, names[i]);

   Options o = Options::Option2;
   Console::WriteLine("value of 'o' is {0}", o);
}
```

Output

```
there are 3 options in enum 'Options'
0: Option1
1: Option2
2: Option3
value of 'o' is Option2
```

The **GetType** object supports a number of members and properties that can be used to examine a type. This code retrieves and displays some of this information:

```
// vcpp_reflection_4.cpp
// compile with: /clr
using namespace System;
int main() {
   Console::WriteLine("type information for 'String':");
   Type ^ t = String::typeid;

   String ^ assemblyName = t->Assembly->FullName;
   Console::WriteLine("assembly name: {0}", assemblyName);

   String ^ nameSpace = t->Namespace;
   Console::WriteLine("namespace: {0}", nameSpace);

   String ^ baseType = t->BaseType->FullName;
   Console::WriteLine("base type: {0}", baseType);

   bool isArray = t->IsArray;
   Console::WriteLine("is array: {0}", isArray);

   bool isClass = t->IsClass;
   Console::WriteLine("is class: {0}", isClass);
}
```

Sample Output

```
type information for 'String':
assembly name: mscorlib, Version=1.0.5000.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
```

```
namespace: System
base type: System.Object
is array: False
is class: True
```

Reflection also allows the enumeration of types within an assembly and the members within classes. To demonstrate this feature, define a simple class:

```
// vcpp_reflection_5.cpp
// compile with: /clr /LD
using namespace System;
public ref class TestClass {
   int m_i;
public:
   TestClass() {}
   void SimpleTestMember1() {}
   String ^ SimpleMember2(String ^ s) { return s; }
   int TestMember(int i) { return i; }
   property int Member {
      int get() { return m_i; }
      void set(int i) { m_i = i; }
   }
};
```

If the code above is compiled into a DLL called vcpp_reflection_6.dll, you can then use reflection to inspect the contents of this assembly. This involves using the static reflection API function Assembly::Load to load the assembly. This function returns the address of an **Assembly** object that can then be queried about the modules and types within.

Once the reflection system successfully loads the assembly, an array of **Type** objects is retrieved with the Assembly::GetTypes function. Each array element contains information about a different type, although in this case, only one class is defined. Using a loop, each **Type** in this array is queried about the type members using the **Type::GetMembers** function. This function returns an array of **MethodInfo** objects, each object containing information about the member function, data member, or property in the type.

Note that the list of methods includes the functions explicitly defined in **TestClass** and the functions implicitly inherited from the **System::Object** class. As part of being described in .NET rather than in Visual C++ syntax, properties appear as the underlying data member accessed by the get/set functions. The get/set functions appear in this list as regular methods. Reflection is supported through the common language runtime, not by the Visual C++ compiler.

Although you used this code to inspect an assembly that you defined, you can also use this code to inspect .NET assemblies. For example, if you change TestAssembly to mscorlib, then you will see a listing of every type and method defined in mscorlib.dll.

```
// vcpp_reflection_6.cpp
// compile with: /clr
using namespace System;
using namespace System::IO;
using namespace System::Reflection;
int main() {
   Assembly ^ a = nullptr;
   try {
      // load assembly -- do not use file extension
      // will look for .dll extension first
      // then .exe with the filename
      a = Assembly::Load("vcpp_reflection_5");
   }
   catch (FileNotFoundException ^ e) {
      Console::WriteLine(e->Message);
      return -1;
   }

   Console::WriteLine("assembly info:");
   Console::WriteLine(a->FullName);
   array<Type^>^ typeArray = a->GetTypes();

   Console::WriteLine("type info ({0} types):", typeArray->Length);
```

```
    int totalTypes = 0;
    int totalMembers = 0;
    for (int i = 0 ; i < typeArray->Length ; i++) {
       // retrieve array of member descriptions
       array<MemberInfo^>^ member = typeArray[i]->GetMembers();

       Console::WriteLine("  members of {0} ({1} members):",
       typeArray[i]->FullName, member->Length);
       for (int j = 0 ; j < member->Length ; j++) {
          Console::Write("       ({0})",
          member[j]->MemberType.ToString() );
          Console::Write("{0}  ", member[j]);
          Console::WriteLine("");
          totalMembers++;
       }
       totalTypes++;
    }
    Console::WriteLine("{0} total types, {1} total members",
    totalTypes, totalMembers);
 }
```

See Also

Other Resources

.NET Programming in C++

How to: Implement a Plug-In Component Architecture using Reflection

The following code examples demonstrate the use of reflection to implement a simple "plug-in" architecture. The first listing is the application, and the second is the plug-in. The application is a multiple document form that populates itself using any form-based classes found in the plug-in DLL provided as a command-line argument.

The application attempts to load the provided assembly using the System.Reflection.Assembly.Load method. If successful, the types inside the assembly are enumerated using the System.Reflection.Assembly.GetTypes method. Each type is then checked for compatibility using the System.Type.IsAssignableFrom(System.Type) method. In this example, classes found in the provided assembly must be derived from the Form class to qualify as a plug-in.

Compatible classes are then instantiated with the System.Activator.CreateInstance method, which accepts a Type as an argument and returns a pointer to a new instance. Each new instance is then attached to the form and displayed.

Note that the **Load** method does not accept assembly names that include the file extension. The main function in the application trims any provided extensions, so the following code example works in either case.

Example

The following code defines the application that accepts plug-ins. An assembly name must be provided as the first argument. This assembly should contain at least one public **Form** derived type.

```
// plugin_application.cpp
// compile with: /clr /c
#using <system.dll>
#using <system.drawing.dll>
#using <system.windows.forms.dll>

using namespace System;
using namespace System::Windows::Forms;
using namespace System::Reflection;

ref class PluggableForm : public Form  {
public:
   PluggableForm() {}
   PluggableForm(Assembly^ plugAssembly) {
      Text = "plug-in example";
      Size = Drawing::Size(400, 400);
      IsMdiContainer = true;

      array<Type^>^ types = plugAssembly->GetTypes( );
      Type^ formType = Form::typeid;

      for (int i = 0 ; i < types->Length ; i++) {
         if (formType->IsAssignableFrom(types[i])) {
            // Create an instance given the type description.
            Form^ f = dynamic_cast<Form^> (Activator::CreateInstance(types[i]));
            if (f) {
               f->Text = types[i]->ToString();
               f->MdiParent = this;
               f->Show();
            }
         }
      }
   }
};

int main() {
   Assembly^ a = Assembly::LoadFrom("plugin_application.exe");
   Application::Run(gcnew PluggableForm(a));
}
```

The following code defines three classes derived from **Form**. When the name of the resulting assembly name is passed to the

executable in the previous listing, each of these three classes will be discovered and instantiated, despite the fact that they were all unknown to the hosting application at compile time.

```
// plugin_assembly.cpp
// compile with: /clr /LD
#using <system.dll>
#using <system.drawing.dll>
#using <system.windows.forms.dll>

using namespace System;
using namespace System::Windows::Forms;
using namespace System::Reflection;
using namespace System::Drawing;

public ref class BlueForm : public Form {
public:
   BlueForm() {
      BackColor = Color::Blue;
   }
};

public ref class CircleForm : public Form {
protected:
   virtual void OnPaint(PaintEventArgs^ args) override {
      args->Graphics->FillEllipse(Brushes::Green, ClientRectangle);
   }
};

public ref class StarburstForm : public Form {
public:
   StarburstForm(){
      BackColor = Color::Black;
   }
protected:
   virtual void OnPaint(PaintEventArgs^ args) override {
      Pen^ p = gcnew Pen(Color::Red, 2);
      Random^ r = gcnew Random( );
      Int32 w = ClientSize.Width;
      Int32 h = ClientSize.Height;
      for (int i=0; i<100; i++) {
         float x1 = w / 2;
         float y1 = h / 2;
         float x2 = r->Next(w);
         float y2 = r->Next(h);
         args->Graphics->DrawLine(p, x1, y1, x2, y2);
      }
   }
};
```

See Also

Reference

Reflection in C++

How to: Enumerate Data Types in Assemblies using Reflection

The following code demonstrates the enumeration of public types and members using System.Reflection.

Given the name of an assembly, either in the local directory or in the GAC, the code below attempts to open the assembly and retrieve descriptions. If successful, each type is displayed with its public members.

Note that System.Reflection.Assembly.Load requires that no file extension is used. Therefore, using "mscorlib.dll" as a command-line argument will fail, while using just "mscorlib" will result the display of the .NET Framework types. If no assembly name is provided, the code will detect and report the types within the current assembly (the EXE resulting from this code).

Example

```
// self_reflection.cpp
// compile with: /clr
using namespace System;
using namespace System::Reflection;
using namespace System::Collections;

public ref class ExampleType {
public:
   ExampleType() {}
   void Func() {}
};

int main() {
   String^ delimStr = " ";
   array<Char>^ delimiter = delimStr->ToCharArray( );
   array<String^>^ args = Environment::CommandLine->Split( delimiter );

   Assembly^ a = Assembly::LoadFrom("self_reflection.exe");
   Console::WriteLine(a);

   int count = 0;
   array<Type^>^ types = a->GetTypes();
   IEnumerator^ typeIter = types->GetEnumerator();

   while ( typeIter->MoveNext() ) {
      Type^ t = dynamic_cast<Type^>(typeIter->Current);
      Console::WriteLine("   {0}", t->ToString());

      array<MemberInfo^>^ members = t->GetMembers();
      IEnumerator^ memberIter = members->GetEnumerator();
      while ( memberIter->MoveNext() ) {
         MemberInfo^ mi = dynamic_cast<MemberInfo^>(memberIter->Current);
         Console::Write("      {0}", mi->ToString( ) );
         if (mi->MemberType == MemberTypes::Constructor)
            Console::Write("   (constructor)");

         Console::WriteLine();
      }
      count++;
   }
   Console::WriteLine("{0} types found", count);
}
```

See Also

Reference

Reflection in C++

Strong Name Assemblies (Assembly Signing)

This topic discusses how you can sign your assembly, often referred to as giving your assembly a strong name.

Remarks

When using Visual C++, use linker options to sign your assembly to avoid issues related to the CLR attributes for assembly signing:

- AssemblyDelaySignAttribute
- AssemblyKeyFileAttribute
- AssemblyKeyNameAttribute

Reasons for not using the attributes include the fact that the key name is visible in assembly metadata, which can be a security risk if the file name includes confidential information. Also, the build process used by the Visual C++ development environment will invalidate the key with which the assembly is signed if you use CLR attributes to give an assembly a strong name, and then run a post-processing tool like mt.exe on the assembly.

If you build at the command line, use linker options to sign your assembly, and then run a post-processing tool (like mt.exe), you will need to re-sign the assembly with sn.exe. Alternatively, you can build and delay sign the assembly and after running post-processing tools, complete the signing.

If you use the signing attributes when building in the development environment, you can successfully sign the assembly by explicitly calling sn.exe (Strong Name Tool (Sn.exe)) in a post-build event. For more information, see Specifying Build Events. Build times may be less if you use attributes and a post-build event, compared to using a linker options.

The following linker options support assembly signing:

- /DELAYSIGN (Partially Sign an Assembly)
- /KEYFILE (Specify Key or Key Pair to Sign an Assembly)
- /KEYCONTAINER (Specify a Key Container to Sign an Assembly)

For more information on strong assemblies, see Creating and Using Strong-Named Assemblies.

See Also

Other Resources

.NET Programming in C++

The Debug Class in Visual C++

When using Debug in a Visual C++ application, the behavior does not change between a debug and a release build.

Remarks

The behavior for Trace is identical to the behavior for the Debug class, but is dependent on the symbol TRACE being defined. This means that you must **#ifdef** any Trace-related code to prevent debug behavior in a release build.

Example

The following sample always executes the output statements, regardless of whether you compile with **/DDEBUG** or **/DTRACE**.

```
// mcpp_debug_class.cpp
// compile with: /clr
#using <system.dll>
using namespace System::Diagnostics;
using namespace System;

int main() {
   Trace::Listeners->Add( gcnew TextWriterTraceListener( Console::Out ) );
   Trace::AutoFlush = true;
   Trace::Indent();
   Trace::WriteLine( "Entering Main" );
   Console::WriteLine( "Hello World." );
   Trace::WriteLine( "Exiting Main" );
   Trace::Unindent();

   Debug::WriteLine("test");
}
```

Output

```
    Entering Main
Hello World.
    Exiting Main
test
```

Example

To get the expected behavior (that is, no "test" output printed for a release build), you must use the **#ifdef** and **#endif** directives. The previous code sample is modified below to demonstrate this fix:

```
// mcpp_debug_class2.cpp
// compile with: /clr
#using <system.dll>
using namespace System::Diagnostics;
using namespace System;

int main() {
   Trace::Listeners->Add( gcnew TextWriterTraceListener( Console::Out ) );
   Trace::AutoFlush = true;
   Trace::Indent();

#ifdef TRACE   // checks for a debug build
   Trace::WriteLine( "Entering Main" );
   Console::WriteLine( "Hello World." );
   Trace::WriteLine( "Exiting Main" );
#endif
   Trace::Unindent();

#ifdef DEBUG   // checks for a debug build
   Debug::WriteLine("test");
#endif   //ends the conditional block
}
```

See Also

Other Resources

.NET Programming in C++

Native and .NET Interoperability

Visual C++ supports interoperability features that allow managed and unmanaged constructs to co-exist and interoperate within the same assembly, and even in the same file. A small subset of this functionality, such as P/Invoke, is supported by other .NET languages as well, but most of the interoperability support provided by Visual C++ is not available in other languages.

In This Section

Mixed (Native and Managed) Assemblies

Describes assemblies generated with the /clr (Common Language Runtime Compilation) compiler option that contain both managed and unmanaged functionality.

Using Native COM Servers from .NET

Describes how COM components can be used from .NET applications.

Using a Windows Form User Control in MFC

Discusses how to use the MFC Windows Forms support classes to host Windows Forms controls within your MFC applications.

Calling Native Functions from Managed Code

Describes how non-CLR DLLs can be used from .NET applications.

See Also

Other Resources

Programming Guide

Mixed (Native and Managed) Assemblies

Mixed assemblies are capable of containing both unmanaged machine instructions and MSIL instructions. This allows them to call and be called by .NET components, while retaining compatibility with components that are entirely unmanaged. Using mixed assemblies, developers can author applications using a mixture of managed and unmanaged functionality. This makes mixed assemblies ideal for migrating existing Visual C++ applications to the .NET Platform.

For example, an existing application consisting entirely of unmanaged functions can be brought to the .NET platform by recompiling just one module with the **/clr** compiler switch. This module is then able to use .NET features, but remains compatible with the remainder of the application. In this way, an application can be converted to the .NET platform in a gradual, piece-by-piece fashion. It is even possible to decide between managed and unmanaged compilation on a function-by-function basis within the same file (see managed, unmanaged).

Visual C++ 2005 supports the generation of three distinct types of managed assemblies: mixed, pure, and verifiable. The latter two are discussed in Pure and Verifiable Code.

In This Section

How To: Migrate to /clr

Describes the recommended steps for either introducing or upgrading .NET functionality in your application.

How to: Compile MFC and ATL Code with /clr

Discusses how to compile existing MFC and ATL programs to target the Common Language Runtime.

Initialization of Mixed Assemblies

Describes the "loader lock" problem and solutions.

How To: Remove Dependency on _vcclrit.h

Describes how to revert the loader lock solution recommended for Visual C++ 2003.

Library Support for Mixed Assemblies

Discusses how to use native libraries in **/clr** compilations.

Performance Considerations for Interop (C++)

Describes the performance implications of mixed assemblies and data marshaling.

Application Domains and Visual C++

Discusses Visual C++ support for application domains.

Double Thunking (C++)

Discusses the performance implications of a native entry point for a managed function.

Avoiding Exceptions on CLR Shutdown When Consuming COM Objects Built with /clr

Discusses how to ensure proper shutdown of a managed application that consumes a COM object compiled with **/clr**.

How to: Create a Partially Trusted Application by Removing Dependency on msvcm80.dll

Discusses how to create a partially trusted Common Language Runtime application using Visual C++ by removing dependency on msvcm80.dll.

For more information about coding guidelines for mixed assemblies, see the MSDN article "An Overview of Managed/Unmanaged Code Interoperability" at http://msdn.microsoft.com/netframework/default.aspx?pull=/library/en-us/dndotnet/html/manunmancode.asp.

See Also

Other Resources

Native and .NET Interoperability

How To: Migrate to /clr

This topic discusses issues that arise when compiling native code with **/clr** (see /clr (Common Language Runtime Compilation) for more information). **/clr** allows Visual C++ modules to invoke and be invoked from .NET assemblies while retaining compatibility with unmanaged modules. See Mixed (Native and Managed) Assemblies and Native and .NET Interoperability for more information on the advantages of compiling with **/clr**.

Known Issues Compiling Library Projects with /clr

Visual C++ 2005 contains some known issues when compiling library projects with **/clr**:

- When compiling an MFC ActiveX control project with **/clr** in the development environment**,** the build system will attempt to register the .dll with regasm and not regsvr32. You must manually register the control with regsvr32.
- When you create an ATL project and then enable **/clr**, the .c files will generate an error, because .c files cannot be compiled with **/clr**. However, if you change the file settings to compile the file with **/TP**, you will generate linker errors. The resolution is to compile the .c files as native (without **/clr**).
- Your code may query types at runtime with CRuntimeClass::FromName. However, if a type is in an MSIL .dll (compiled with **/clr**), the call to CRuntimeClass::FromName may fail if it occurs before the static constructors run in the managed .dll (you will not see this problem if the FromName call happens after code has executed in the managed .dll). To work around this problem, you can force the construction of the managed static constructor by defining a function in the managed .dll, exporting it, and invoking it from the native MFC application. For example:

```
// Extention DLL Header file:
__declspec( dllexport ) void EnsureManagedInitialization () {
   // managed code that won't be optimized away
   System::GC::KeepAlive(System::Int32::MaxValue);
}
```

Compile with Visual C++ 2005

Before using **/clr** on any module in your project, first compile and link your native project with Visual C++ 2005.

The following steps, followed in order, provide the easiest path to a **/clr** compilation. It is important to compile and run your project after each of these steps.

Versions Prior to Visual C++ 2003

If you are upgrading to Visual C++ 2005 from a version prior to Visual C++ 2003, you may see compiler errors related to the enhanced C++ standard conformance in Visual C++ 2003. See Breaking Changes in the Visual C++ Compiler for more information.

Upgrading from Visual C++ 2003

Projects previous built with Visual C++ 2003 should also first be compiled without **/clr** as Visual C++ 2005 has increased ANSI/ISO compliance and some breaking changes. See Breaking Changes in the Visual C++ Compiler for more information. The change that is likely to require the most attention is Security Enhancements in the CRT. Code that uses the CRT is very likely to produce deprecation warnings. These warnings can be suppressed, but migrating to the new Security-Enhanced Versions of CRT Functions is preferred, as they provide better security and may reveal security issues in your code.

Upgrading from Managed Extensions for C++

Projects built with Visual C++ .NET or Visual C++ 2003 that used Managed Extensions for C++ will require at least one change to project settings, as these extensions are now deprecated. As a result, code written with Managed Extentions for C++ won't compile under **/clr**. Use **/clr:oldSyntax** instead.

Convert C Code to C++

Although Visual C++ 2005 will compile C files, it is necessary to convert them to C++ for a **/clr** compilation. The actual filename doesn't have to be changed, you can use **/Tp** (see /Tc, /Tp, /TC, /TP (Specify Source File Type).) Note that, although C++ source code files are required for **/clr**, it is not necessary to re-factor your code to use object-oriented paradigms.

C code is very likely to require changes when compiled as a C++ file. The C++ type-safety rules are strict, so type conversions

must be made explicit with casts. For example, malloc returns a void pointer, but can be assigned to a pointer to any type in C with a cast:

```
int* a = malloc(sizeof(int));   // C code
int* b = (int*)malloc(sizeof(int));   // C++ equivalent
```

Function pointers are also strictly type-safe in C++, so the following C code requires modification. In C++ it's best to create a **typedef** that defines the function pointer type, and then use that type to cast function pointers:

```
NewFunc1 = GetProcAddress( hLib, "Func1" );   // C code
typedef int(*MYPROC)(int);   // C++ equivalent
NewFunc2 = (MYPROC)GetProcAddress( hLib, "Func2" );
```

C++ also requires that functions either be prototyped or fully defined before they can be referenced or invoked.

Identifiers used in C code that happen to be keywords in C++ (such as **virtual**, **new**, **delete**, **bool**, **true**, **false**, etc.) must be renamed. This can generally be done with simple search-and-replace operations.

Finally, whereas C-style COM calls require explicit use of the v-table and **`this`** pointer, C++ does not:

```
COMObj1->lpVtbl->Method(COMObj, args);  // C code
COMObj2->Method(args);  // C++ equivalent
```

Reconfigure Project Settings

After your project compiles and runs in Visual C++ 2005 you should create new project configurations for **/clr** rather than modifying the default configurations. **/clr** is incompatible with some compiler options and creating separate configurations lets you build your project as native or managed. When **/clr** is selected in the property pages dialog box, project settings not compatible with **/clr** are disabled (and disabled options are not automatically restored if **/clr** is subsequently unselected).

Create New Project Configurations

You can use **Copy Settings From** option in the New Project Configuration Dialog Box to create a project configuration based on your existing project settings. Do this once for the Debug configuration and once for Release configuration. Subsequent changes can then be applied to the **/clr** -specific configurations only, leaving the original project configurations intact.

Projects that use custom build rules may require extra attention.

This step has different implications for projects that use makefiles. In this case a separate build-target can be configured, or version specific to **/clr** compilation can be created from a copy of the original.

Change Project Settings

/clr can be selected in the development environment by following the instructions in /clr (Common Language Runtime Compilation). As mentioned previously, this step will automatically disable conflicting project settings.

Note
When upgrading a managed library or web service project from Visual C++ 2003 to Visual C++ 2005, the **/Zl** compiler opti on will added to the **Command Line** property page. This will cause LNK2001. Remove **/Zl** from the **Command Line** propert y page to resolve. See /Zl (Omit Default Library Name) and How to: Open Project Property Pages for more information. Or, a dd msvcrt.lib and msvcmrt.lib to the linker's **Additional Dependencies** property.

For projects built with makefiles, incompatible compiler options must be disabled manually once **/clr** is added. See //clr Restrictions for information on compiler options that are not compatible with **/clr**.

Precompiled Headers

Precompiled headers are supported under **/clr**. However, if you only compile some of your CPP files with **/clr** (compiling the rest as native) some changes will be required because precompiled headers generated with **/clr** are not compatible with those generated without **/clr**. This incompatibility is due to the fact that **/clr** generates and requires metadata. Modules compiled **/clr** can therefore not use precompiled headers that don't include metadata, and non **/clr** modules can't use precompiled header files that do contain meta data.

The easiest way to compile a project where some modules are compiled **/clr** is to disable precompiled headers entirely. (In the

project Property Pages dialog, open the C/C++ node, and select Precompiled Headers. Then change the Create/Use Precompiled Headers property to "Not Using Precompiled Headers".)

However, particularly for large projects, precompiled headers provide much better compilation speed, so disabling this feature is not desirable. In this case it's best to configure the **/clr** and non **/clr** files to use separate precompiled headers. This can be done in one step by multi-selecting the modules to be compiled **/clr** using the Solution Explorer, right-clicking on the group, and selecting Properties. Then change both the Create/Use PCH Through File and Precompiled Header File properties to use a different header file name and PCH file respectively.

Fixing Errors

Compiling with **/clr** may result in compiler, linker or runtime errors. This section discusses the most common problems.

Metadata Merge

Differing versions of data types can cause the linker to fail because the metadata generated for the two types doesn't match. (This is usually caused when members of a type are conditionally defined, but the conditions are not the same for all CPP files that use the type.) In this case the linker fails, reporting only the symbol name and the name of the second OBJ file where the type was defined. It is often useful to rotate the order that OBJ files are sent to the linker to discover the location of the other version of the data type.

Loader Lock Deadlock

In Visual C++ .NET and Visual C++ 2003, initialization under **/clr** was susceptible to non-deterministic deadlock. This issue is known as "loader lock deadlock". In Visual C++ 2005 , this deadlock is easier to avoid, it is detected and reported at runtime, and is no longer non-deterministic. Encountering the loader lock problem is still possible, but now it's much easier to avoid and fix. See Initialization of Mixed Assemblies for detailed background, guidance, and solutions.

Data Exports

Exporting DLL data is error-prone, and not recommended. This is because the data section of a DLL is not guaranteed to be initialized until some managed portion of the DLL has been executed. Reference metadata with The #using Directive.

Type Visibility

Native types are now private by default. In Visual C++ .NET 2002 and Visual C++ 2003, native types were public by default. This can result in a native type not being visible outside the DLL. Resolve this error by adding **public** to these types. See Type and Member Visibility for more information.

Floating Point and Alignment Issues

__controlfp is not supported on the common language runtime (see _control87, _controlfp, __control87_2 for more information). The CLR will also not respect align (C++).

COM Initialization

The Common Language Runtime initializes COM automatically when a module is initialized (when COM is initialized automatically it’s done so as MTA). As a result, explicitly initializing COM yields return codes indicating that COM is already initialized. Attempting to explicitly initialize COM with one threading model when the CLR has already initialized COM to another threading model can cause your application to fail.

COM initialization and associated error code should either allow for the case where COM is already initialized, or calls to CoInitialize and CoUninitialize can generally simply be removed. The common language runtime starts COM as MTA by default; use /CLRTHREADATTRIBUTE (Set CLR Thread Attribute) to modify this.

Performance Issues

You may see decreased performance when native C++ methods generated to MSIL are called indirectly (virtual function calls or using function pointers). To learn more about this, see Double Thunking (C++).

When moving from native to MSIL, you will notice an increase in the size of your working set. This is because the common language runtime provides many features to ensure that programs run correctly. If your **/clr** application is not running correctly, you may want to enable C4793 (off by default), see Compiler Warning (level 1 and 3) C4793 for more information.

Program Crashes on Shutdown

In some cases, the CLR can shutdown before your managed code is finished running. Using **std::set_terminate** and **SIGTERM** can cause this. See signal Constants and set_terminate (<exception>) for more information.

Using New Visual C++ Features

After your application compiles, links, and runs, you can begin using .NET features in any module compiled with **/clr**. For more

information, see New Language Features in Visual C++.

If you used Managed Extensions for C++, you can convert your code to use the new syntax. For a summary of syntactical differences, see the Managed Extensions for C++ Syntax Upgrade Checklist. For details on converting Managed Extensions for C++, see C++/CLI Migration Primer.

For information on .NET programming in Visual C++ see:

- .NET Programming in C++
- Native and .NET Interoperability
- New Language Features in Visual C++

See Also

Concepts

Mixed (Native and Managed) Assemblies

How to: Compile MFC and ATL Code with /clr

This topic discusses how to compile existing MFC and ATL programs to target the Common Language Runtime.

To compile an MFC executable or regular DLL with /clr

1. Open the **Project Properties** dialog by right clicking the project in the **Solution Explorer** and selecting **Properties**.
2. Expand the node next to **Configuration Properties** and select **General**. In the right pane, under **Project Defaults**, set **Common Language Runtime support** to **Common Language Runtime Support (/clr)**.

   From the same pane, make sure **Use of MFC** is set to **Use MFC in a Shared DLL**.
3. Under **Configuration Properties**, expand the node next to **C/C++** and select **General**. Make sure **Debug Information Format** is set to **Program Database /Zi** (not **/ZI**).
4. Under **Configuration Properties**, select **C/C++** and then **Code Generation**. Make sure **Runtime Library** is set to one of **Multi-threaded Debug DLL (/MDd)** or **Multi-threaded DLL (/MD)**.

To compile an MFC extension DLL with /clr

1. Follow the steps above in the section "To compile an MFC executable or regular DLL with /clr".
2. Do **one of** the following:
   a. Under **Configuration Properties**, expand the node next to **C/C++** and select **Precompiled Headers**. Set **Create/Use Precompiled Header** to **Not using Precompiled Headers**.
   b. From the **Solution Explorer**, right click on the file stdafx.cpp and select **Properties**. Under **Configuration Properties**, expand the node next to **C/C++** and select **General**. Set **Compile with Common Language Runtime support** to **No Common Language Runtime support**.
3. For the file containing DllMain and anything it calls, open the **Property Pages** for that file by right clicking the file from the **Solution Explorer** and selecting **Properties**. Under **Configuration Properties**, expand the node next to **C/C++** and select **General**. In the right pane, under **Project Defaults**, set **Compile with Common Language Runtime support** to **No Common Language Runtime support**.

To compile an ATL executable with /clr

1. Open the **Project Properties** dialog by right clicking the project in the **Solution Explorer** and selecting **Properties**.
2. Expand the node next to **Configuration Properties** and select **General**. In the right pane, under **Project Defaults**, set **Common Language Runtime support** to **Common Language Runtime Support (/clr)**.
3. Under **Configuration Properties**, expand the node next to **C/C++** and select **General**. Make sure **Debug Information Format** is set to **Program Database /Zi** (not **/ZI**).
4. Under **Configuration Properties**, select **C/C++** and then **Code Generation**. Make sure **Runtime Library** is set to one of **Multi-threaded Debug DLL (/MDd)** or **Multi-threaded DLL (/MD)**.
5. For every MIDL generated file (C files), right click the file in the **Solution Explorer** and select **Properties**. Under **Configuration Properties**, expand the node next to **C/C++** and select **General**. Set **Compile with Common Language Runtime support** to **No Common Language Runtime support**.

To compile an ATL DLL with /clr

1. Follow the steps above in the section "To compile an ATL executable with /clr".
2. Do **one of** the following:
   a. Under **Configuration Properties**, expand the node next to **C/C++** and select **Precompiled Headers**. Set **Create/Use Precompiled Header** to **Not using Precompiled Headers**.
   b. From the **Solution Explorer**, right click on the file stdafx.cpp and select **Properties**. Under **Configuration Properties**, expand the node next to **C/C++** and select **General**. Set **Compile with Common Language**

Runtime support to **No Common Language Runtime support**.

3. For the file containing DllMain and anything it calls, open the **Property Pages** for that file by right clicking the file from the **Solution Explorer** and selecting **Properties**. Under **Configuration Properties**, expand the node next to **C/C++** and select **General**. In the right pane, under **Project Defaults**, set **Compile with Common Language Runtime support** to **No Common Language Runtime support**.

See Also

Concepts

Mixed (Native and Managed) Assemblies

Initialization of Mixed Assemblies

In Visual C++ .NET and Visual C++ 2003, DLLs compiled with the **/clr** compiler option could non-deterministically deadlock when loaded; this issue was called the mixed DLL loading or loader lock issue. In Visual C++ 2005, almost all non-determinism has been removed from the mixed DLL loading process. However, there are a few remaining scenarios for which loader lock can (deterministically) occur. For more information about this issue, see:

- "Mixed DLL Loading Problem" at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vcconMixedDLLLoadingProblem.asp.

In Visual C++ 2005 there is still a restriction that code within DllMain must not access the CLR. This means that **DllMain** should make no calls to managed functions, directly or indirectly; no managed code should be declared or implemented in **DllMain**; and no garbage collection or automatic library loading should take place within **DllMain**.

Note
Visual C++ 2003 provided _vcclrit.h to facilitate DLL initialization while minimizing opportunity for deadlock. For Visual C++ 2005, using _vcclrit.h is no longer necessary, and causes deprecation warnings to be produced during compilation. The recommended strategy is to remove dependencies on this file using the steps in How To: Remove Dependency on _vcclrit.h. Less ideal solutions include suppressing the warnings by defining **_CRT_VCCLRIT_NO_DEPRECATE** prior to including _vcclrit.h, or merely ignoring the deprecation warnings.

Causes of Loader Lock

With the introduction of the .NET platform there are two distinct mechanisms for loading an execution module (EXE or DLL): one for Windows, which is used for unmanaged modules, and one for the .NET Common Language Runtime (CLR) which loads .NET assemblies. The mixed DLL loading problem centers around the Microsoft Windows OS loader.

When an assembly containing only .NET constructs is loaded into a process, the CLR loader can perform all of the necessary loading and initialization tasks itself. However, for mixed assemblies, because they can contain native code and data, the Windows loader must be used as well.

The Windows loader guarantees that no code can access code or data in that DLL before it has been initialized, and that no code can redundantly load the DLL while it is partially initialized. To do this, the Windows loader uses a process-global critical section (often called the "loader lock") that prevents unsafe access during module initialization. As a result, the loading process is vulnerable to many classic deadlock scenarios. For mixed assemblies, the following two scenarios increase the risk of deadlock:

- First, if users attempt to execute functions compiled to Microsoft intermediate language (MSIL) when the loader lock is held (from **DllMain** or in static initializers, for example), this can cause deadlock. Consider the case in which the MSIL function references a type in an assembly that has not been loaded. The CLR will attempt to automatically load that assembly, which may require the Windows loader to block on the loader lock. Since the loader lock is already held by code earlier in the call sequence, a deadlock results. However, executing MSIL under loader lock does not guarantee that a deadlock will occur, making this scenario difficult to diagnose and fix. In some circumstances, such as where the DLL of the referenced type contains no native constructs and all of its dependencies contain no native constructs, the Windows loader is not required to load the .NET assembly of the referenced type. Additionally, the required assembly or its mixed native/.NET dependencies may have already been loaded by other code. Consequently, the deadlocking can be difficult to predict, and can vary depending on the configuration of the target machine.
- Second, when loading DLLs in versions 1.0 and 1.1 of the .NET Framework, the CLR assumed that the loader lock was not held and performed several actions that are invalid under loader lock. Assuming the loader lock is not held is a valid assumption for purely .NET DLLs, but, because mixed DLLs execute native initialization routines, they require the native Windows loader and therefore the loader lock. Consequently, even if the developer was not attempting to execute any MSIL functions during DLL initialization, there was still a small possibility of nondeterministic deadlock with versions 1.0 and 1.1 of the .NET Framework.

In Visual C++ 2005, all non-determinism has been removed from the mixed DLL loading process. This was accomplished with these changes:

- The CLR no longer makes false assumptions when loading mixed DLLs.
- Unmanaged and managed initialization is performed in two separate and distinct stages. Unmanaged initialization takes

place first (via DllMain), and managed initialization takes place afterwards, through a .NET-supported construct called a *.cctor*. The latter is completely transparent to the user unless **/Zl** or **/NODEFAULTLIB** are used. See/NODEFAULTLIB (Ignore Libraries) and /Zl (Omit Default Library Name) for more information.

Loader lock can still occur, but now it occurs reproducibly, and is detected. If DllMain contains MSIL instructions, the compiler will generate warning Compiler Warning (level 1) C4747. Furthermore, either the CRT or the CLR will try to detect and report attempts to execute MSIL under loader lock. CRT detection results in runtime diagnostic C Run-Time Error R6033.

The remainder of this document describes the remaining scenarios for which MSIL can execute under the loader lock, resolutions for the issue under each of those scenarios, and debugging techniques.

Scenarios and Workarounds

There are several different situations under which user code can execute MSIL under loader lock. The developer must ensure that the user code implementation does not attempt to execute MSIL instructions under each of these circumstances. The following subsections describe all possibilities with a discussion of how to resolve issues in the most common cases.

- **DllMain**
- Static Initializers
- User-Supplied Functions Affecting Startup
- Custom Locales

DllMain

The **DllMain** function is a user defined entry point for a DLL. Unless the user specifies otherwise, **DllMain** is invoked every time a process or thread attaches to or detaches from the containing DLL. Since this invocation can occur while the loader lock is held, no user-supplied **DllMain** function should be compiled to MSIL. Furthermore, no function in the call tree rooted at **DllMain** can be compiled to MSIL. To resolve issues here, the code block that defines **DllMain** should be modified with #pragma **unmanaged**. The same should be done for every function that **DllMain** calls.

In cases where these functions must call a function that requires an MSIL implementation for other calling contexts, a duplication strategy can be used where both a .NET and a native version of the same function are created.

Alternatively, if **DllMain** is not required or if it does not need to be executed under loader lock, the user-provided **DllMain** implementation can be removed, which will eliminate the problem.

If DllMain attempts to execute MSIL directly, Compiler Warning (level 1) C4747 will result. However, the compiler cannot detect cases where DllMain calls a function in another module that in turn attempts to execute MSIL.

Please see "Impediments to Diagnosis" for more information on this scenario.

Initializing Static Objects

Initializing static objects can result in deadlock if a dynamic initializer is required. For simple cases, such as when a static variable is simply assigned to a value known at compile time, no dynamic initialization is required, so there is no risk of deadlock. However, static variables initialized by function calls, constructor invocations, or expressions that cannot be evaluated at compile time all require code to execute during module initialization.

The code below shows examples of static initializers that require dynamic initialization: a function call, object construction, and a pointer initialization. (These examples aren't static, but are assumed to be defined in the global scope, which has the same effect.)

```
// dynamic initializer function generated
int a = init();
CObject o(arg1, arg2);
CObject* op = new CObject(arg1, arg2);
```

This risk of deadlock depends on whether the containing module is compiled with **/clr** and whether MSIL will be executed. Specifically, if the static variable is compiled without **/clr** (or resides in a #pragma **unmanaged** block), and the dynamic initializer required to initialize it results in the execution of MSIL instructions, deadlock may occur. This is because, for modules compiled without **/clr**, the initialization of static variables is performed by DllMain. In contrast, static variables compiled with **/clr** are initialized by the .cctor, after the unmanaged initialization stage has completed and the loader lock has been released.

There are a number of solutions to deadlock caused by the dynamic initialization of static variables (arranged roughly in order of time required to fix the problem):

- The source file containing the static variable can be compiled with **/clr**.
- All functions called by the static variable can be compiled to native code using the #pragma **unmanaged** directive.
- Manually clone the code that the static variable depends upon, providing both a .NET and a native version with different names. Developers can then call the native version from native static initializers and call the .NET version elsewhere.

User-Supplied Functions Affecting Startup

There are several user-supplied functions on which libraries depend for initialization during startup. For example, when globally overloading operators in C++ such as the **new** and **delete** operators, the user-provided versions are used everywhere, including in STL initialization and destruction. As a result, STL and user-provided static initializers will invoke any user-provided versions of these operators.

If the user-provided versions are compiled to MSIL, then these initializers will be attempting to execute MSIL instructions while the loader lock is held. A user-supplied malloc has the same consequences. To resolve this problem, any of these overloads or user-supplied definitions must be implemented as native code using the #pragma **unmanaged** directive.

Please see "Impediments to Diagnosis" for more information on this scenario.

Custom Locales

If the user provides a custom global locale, this locale will be used for initializing all future I/O streams, including those that are statically initialized. If this global locale object is compiled to MSIL, then locale-object member functions compiled to MSIL may be invoked while the loader lock is held.

There are three options for solving this problem:

The source files containing all global I/O stream definitions can be compiled using the **/clr** option. This will prevent their static initializers from being executed under loader lock.

The custom locale function definitions can be compiled to native code by using the #pragma **unmanaged** directive.

Refrain from setting the custom locale as the global locale until after the loader lock is released. Then explicitly configure I/O streams created during initialization with the custom locale.

Impediments to Diagnosis

In some cases it is difficult to detect the source of deadlocks. The following subsections discuss these scenarios and ways to work around these issues.

Implementation in Headers

With Visual C++ .NET and Visual C++ .NET 2003, and in select cases with Visual C++ 2005, function implementations inside header files can complicate diagnosis. Inline functions and template code both require that functions be specified in a header file. The C++ language specifies the One Definition Rule, which forces all implementations of functions with the same name to be semantically equivalent. Consequently, the C++ linker need not make any special considerations when merging object files that have duplicate implementations of a given function.

In Visual C++ .NET and Visual C++ .NET 2003, the linker simply chooses the largest of these semantically equivalent definitions, to accommodate forward declarations and scenarios when different optimization options are used for different source files. This creates a problem for mixed native/.NET DLLs.

Because the same header may be included both by a CPP files with **/clr** enabled and disabled, or a #include can be wrapped inside a #pragma **unmanaged** block, it is possible to have both MSIL and native versions of functions that provide implementations in headers. MSIL and native implementations have different semantics with respect to initialization under the loader lock, which effectively violates the one definition rule. Consequently, when the linker chooses the largest implementation, it may choose the MSIL version of a function, even if it was explicitly compiled to native code elsewhere using the #pragma unmanaged directive. To ensure that an MSIL version of a template or inline function is never called under loader lock, every definition of every such function called under loader lock must be modified with the #pragma **unmanaged** directive. If the header file is from a third party, the easiest way to achieve this is to push and pop the #pragma unmanaged directive around the #include directive for the offending header file. (See managed, unmanaged for an example.) However, this strategy will not work for headers that contain other code that must directly call .NET APIs.

In Visual C++ 2005, as a convenience for users dealing with loader lock, the linker will choose the native implementation over the managed when presented with both. This avoids the above issues. However, there are two exceptions to this rule in this release due to two unresolved issues with the compiler:

- The call is to an inline function is through a global static function pointer. This scenario is particularly notable because virtual functions are called through global function pointers. For example,

```
#include "definesfoo.h"
#include "definesclassC.h"

typedef void (*function_pointer_t)();

function_pointer_t foo_p = &foo;

#pragma unmanaged
void DuringLoaderlock(C & c)
{
    // Either of these calls could resolve to a managed implementation,
    // at link-time, even if a native implementation also exists.
    c.VirtualMember();
    foo_p();
}
```

- With Itanium-targeted compilation, there is a bug in the implementation of all function pointers. In the code snippet above, if foo_p were defined locally inside during_loaderlock(), the call might also resolve to a managed implementation.

Diagnosing in Debug Mode

All diagnoses of loader lock problems should be done with Debug builds. Release builds may not produce diagnostics, and the optimizations performed in Release mode may mask some of the MSIL under loader lock scenarios.

How to Debug Loader Lock Issues

The diagnostic that the CLR generates when an MSIL function is invoked causes the CLR to suspend execution. In turn, this causes the Visual C++ 2005 mixed-mode debugger to be suspended as well when running the debuggee in-process. However, when attaching to the process, it is not possible to obtain a managed callstack for the debuggee using the mixed debugger.

To identify the specific MSIL function that was called under loader lock, developers should complete the following steps:

1. Ensure that symbols for mscoree.dll and mscorwks.dll are available.

   This can be done in two ways. First, the PDBs for mscoree.dll and mscorwks.dll can be added to the symbol search path. To do this, open the symbol search path options dialog. (From the Tools menu, click Options. In the left pane of the Options dialog box, Open the Debugging node and click Symbols.) Add the path to the mscoree.dll and mscorwks.dll PDB files to the search list. These PDBs are installed to the %VSINSTALLDIR%\SDK\v2.0\symbols. Click OK.

   Second, the PDBs for mscoree.dll and mscorwks.dll can be downloaded from the Microsoft Symbol Server. To configure Symbol Server, open the symbol search path options dialog. (From the Tools menu, click Options. In the left pane of the Options dialog box, Open the Debugging node and click Symbols.) Add the following search path to the search list: http://msdl.microsoft.com/download/symbols. Add a symbol cache directory to the symbol server cache text box. Click OK.

2. Set debugger mode to native-only mode.

   To do this, open the Properties grid for the startup project in the solution. Under the Configuration Properties subtree, select the Debugging Node. Set the Debugger Type Field to Native-Only.

3. Start the Debugger (F5).

4. When the **/clr** diagnostic is generated, click Retry and then click Break.

5. Open the call stack window. (From the Debug menu, click Windows, then Call Stack.) If the offending **DllMain** or static initializer is identified with a green arrow. If the offending function is not identified, the following steps must be taken to find it.

6. Open the immediate window (From the Debug menu, click Windows, then Immediate.)

7. Type .load sos.dll into the immediate window to load the SOS debugging service.

8. Type !dumpstack into the immediate window to obtain a complete listing of the internal **/clr** stack.

9. Look for the first instance (closest to the bottom of the stack) of either _CorDllMain (if **DllMain** causes issue) or _VTableBootstrapThunkInitHelperStub or GetTargetForVTableEntry (if static initializer causes issue). The stack entry just below this call is the invocation of the MSIL implemented function that attempted to execute under loader lock.

10. Go to the source file and line number identified in Step 9 and correct the problem using the scenarios and solutions described in the Scenarios section.

Example

The following sample shows how to avoid loader lock by moving code from DllMain into the constructor of a global object.

In this sample, there is a global managed object whose constructor contains the managed object that was originally in DllMain. The second part of this sample references the assembly, creating an instance of the managed object to invoke the module constructor which does the initialization.

```
// initializing_mixed_assemblies.cpp
// compile with: /clr /LD
#pragma once
#include <stdio.h>
#include <windows.h>
struct __declspec(dllexport) A {
   A() {
      System::Console::WriteLine("Module ctor initializing based on global instance of clas
s.\n");
   }

   void Test() {
      printf_s("Test called so linker does not throw away unused object.\n");
   }
};

#pragma unmanaged
// Global instance of object
A obj;

extern "C"
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved) {
   // Remove all managed code from here and put it in constructor of A.
   return true;
}
```

Example

```
// initializing_mixed_assemblies_2.cpp
// compile with: /clr initializing_mixed_assemblies.lib
#include <windows.h>
using namespace System;
#include <stdio.h>
#using "initializing_mixed_assemblies.dll"
struct __declspec(dllimport) A {
   void Test();
};

int main() {
   A obj;
   obj.Test();
}
```

Output

```
Module ctor initializing based on global instance of class.

Test called so linker does not throw away unused object.
```

See Also

Concepts

Mixed (Native and Managed) Assemblies

How To: Remove Dependency on _vcclrit.h

In Visual C++ .NET and Visual C++ 2003, DLLs compiled with the /clr compiler option could non-deterministically deadlock when loaded. This problem is often called the loader lock issue, and is described in detail in Initialization of Mixed Assemblies.

To address this problem, _vcclrit.h was provided in Visual C++ 2003 to help initialize DLLs in a manner that avoided loader lock. For Visual C++ 2005, initialization is handled in a completely different manner in order to dramatically reduce the chances of deadlock, so this header file is no longer required.

For backward compatibility, _vcclrit.h is still included, and code that uses it continues to work, but the contents of _vcclrit.h have been deprecated. This topic contains the recommended modifications required to remove dependency on this header file.

Note
Although not an ideal solution, the deprecation warnings that result from using _vcclrit.h can be suppressed by defining **_CRT_VCCLRIT_NO_DEPRECATE** prior to including _vcclrit.h.

Linker Setting Changes

The following are changes that should be made to remove _vcclrit.h from projects that employed the recommended Visual C++ 2003 loader lock solution. They all involve changes to the linker settings. See Modifying Project Settings for instructions on changing these settings through Visual Studio.

- Remove the /NOENTRY linker switch. Using _vcclrit.h required that the /NOENTRY (No Entry Point) switch be provided to the linker. This is no longer necessary.
- Do not explicitly input MSVCRT.LIB to the linker. Because the /NOENTRY switch was required, it was necessary to explicitly link to the CRT library file MSVCRT.LIB. With /NOENTRY removed, this library is linked in by default under /clr compilation.
- Remove of NOCHKCLR.OBJ from linker input. This file is no longer required.
- Remove Forced Symbol References. Previously, it was necessary to force the following symbols with the /INCLUDE switch: __DllMainCRTStartup@12, __crt_dll_initialize, and __crt_dll_terminate. None of these symbols are necessary in Visual C++ 2005, so they can safely be removed.

Code Changes

Other than removing #include <_vcclrit.h> from your code, some additional changes may be necessary depending on the nature of your code. At the minimum, __crt_dll_initialize and __crt_dll_terminate will simply no longer be invoked, and can therefore be removed.

See Also

Concepts

Mixed (Native and Managed) Assemblies

Library Support for Mixed Assemblies

Visual C++ 2005 supports the use of the Standard C++ Library, the Common RunTime library (CRT), ATL, and MFC for applications compiled with /clr (Common Language Runtime Compilation). This allows existing applications that use these libraries to use .NET Framework features as well.

This support introduces the following new DLL and import libraries:

- Msvcmrt[d].lib if you compile with /clr. Mixed assemblies links to this import library.
- Msvcm80[d].dll and Msvcurt[d].lib if you compile with /clr:pure. The DLL is a mixed assembly providing managed C Run Time (CRT) support, and is part of a managed assembly installed in the global assembly cache (GAC). Pure assemblies link to this import library and end up bound to Msvcm80.dll.

This support provides several related benefits:

- The CRT and Standard C++ Library are available to both mixed and pure code. The CRT and Standard C++ Library provided are not verifiable; ultimately, your calls are still routed to the same CRT and Standard C++ Library as you are using from native code.
- Correct unified exception handling in pure and mixed images.
- Static initialization of C++ variables in pure and mixed images.
- Support for per-AppDomain and per-process variables in managed code.
- Resolves the loader lock issues that applied to mixed DLLs in Visual C++ .NET and Visual C++ .NET 2003.

In addition, this support presents the following limitations:

- Only the CRT DLL model is supported (both for code compiled with /clr or /clr:pure).
- You cannot mix pure and mixed objects in a single image if those objects use the Visual C++ libraries (because all objects must be pure in a pure image). If you do this, you receive link-time errors.

You should update your common language runtime (CLR) to the current version as it is not guaranteed to work with earlier versions. Code built with these changes will not run on CLR version 1.x.

See Also

Concepts

Mixed (Native and Managed) Assemblies

Performance Considerations for Interop (C++)

This topic provides guidelines for reducing the effect of managed/unmanaged interop transitions on run-time performance.

Visual C++ supports the same interoperability mechanisms as other .NET languages such as Visual Basic and C# (P/Invoke), but it also provides interop support that is specific to Visual C++ (C++ interop). For performance-critical applications, it is important to understand the performance implications of each interop technique.

Regardless of the interop technique used, special transition sequences, called thunks, are required each time a managed function calls an unmanaged function and vice versa. These thunks are inserted automatically by the Visual C++ compiler, but it is important to keep in mind that cumulatively, these transitions can be expensive in terms of performance.

Reducing Transitions

One way to avoid or reduce the cost of interop thunks is to refactor the interfaces involved to minimize managed/unmanaged transitions. Dramatic performance improvements can be made by targeting chatty interfaces, which are those that involved frequent calls across the managed/unmanaged boundary. A managed function that calls an unmanaged function in a tight loop, for example, is a good candidate for refactoring. If the loop itself is moved to the unmanaged side, or if a managed alternative to the unmanaged call is created (perhaps be queuing data on the managed side and then marshaling it to the unmanaged API all at once after the loop), the number of transitions can be reduced significantly.

P/Invoke vs. C++ Interop

For .NET languages, such as Visual Basic and C#, the prescribed method for interoperating with native components is P/Invoke. Because P/Invoke is supported by the .NET Framework, Visual C++ supports it as well, but Visual C++ also provides its own interoperability support, which is referred to as C++ Interop. C++ Interop is preferred over P/Invoke because P/Invoke is not type-safe. As a result, errors are primarily reported at run time, but C++ Interop also has performance advantages over P/Invoke.

Both techniques require several things to happen whenever a managed function calls an unmanaged function:

- The function call arguments are marshaled from CLR to native types.
- A managed-to-unmanaged thunk is executed.
- The unmanaged function is called (using the native versions of the arguments).
- An unmanaged-to-managed thunk is executed.
- The return type and any "out" or "in,out" arguments are marshaled from native to CLR types.

The managed/unmanaged thunks are necessary for interop to work at all, but the data marshaling that is required depends on the data types involved, the function signature, and how the data will be used.

The data marshaling performed by C++ Interop is the simplest possible form: the parameters are simply copied across the managed/unmanaged boundary in a bitwise fashion; no transformation is performed at all. For P/Invoke, this is only true if all parameters are simple, blittable types. Otherwise, P/Invoke performs very robust steps to convert each managed parameter to an appropriate native type, and vice versa if the arguments are marked as "out", or "in,out".

In other words, C++ Interop uses the fastest possible method of data marshaling, whereas P/Invoke uses the most robust method. This means that C++ Interop (in a fashion typical for C++) provides optimal performance by default, and the programmer is responsible for addressing cases where this behavior is not safe or appropriate.

C++ Interop therefore requires that data marshaling must be provided explicitly, but the advantage is that the programmer is free to decide what is appropriate, given the nature of the data, and how it is to be used. Furthermore, although the behavior of P/Invoke data marshaling can be modified at customized to a degree, C++ Interop allows data marshaling to be customized on a call-by-call basis. This is not possible with P/Invoke.

For more information about C++ Interop, see Using C++ Interop (Implicit PInvoke).

See Also

Concepts

Mixed (Native and Managed) Assemblies

Application Domains and Visual C++

If you have a **__clrcall** virtual function, the vtable will be per application domain (appdomain). If you create an object in one appdomain, you can only call the virtual function from within that appdomain. All functions defined in **/clr:pure** compilands use the **__clrcall** calling convention. Therefore, all vtables defined in **/clr:pure** compilands are per appdomain. In mixed mode (**/clr**) you will have per process vtables if your type has no **__clrcall** virtual functions.

For more information, see

- appdomain
- Application Domains
- __clrcall
- How to: Migrate to /clr:pure
- process

See Also

Concepts

Mixed (Native and Managed) Assemblies

Double Thunking (C++)

Double thunking refers to the loss of performance you can experience when a function call in a managed context calls a Visual C++ managed function and where program execution calls the function's native entry point in order to call the managed function. This topic discusses where double thunking occurs and how you can avoid it to improve performance.

Remarks

By default, when compiling with **/clr** (not **/clr:pure**), the definition of a managed function causes the compiler to generate a managed entry point and a native entry point. This allows the managed function to be called from native and managed call sites. However, when a native entry point exists, it can be the entry point for all calls to the function. If a calling function is managed, the native entry point will then call the managed entry point. In effect, two calls are required to invoke the function (hence, double thunking). For example, virtual functions are always called through a native entry point.

One resolution is to tell the compiler not to generate a native entry point for a managed function, that the function will only be called from a managed context, by using the __clrcall calling convention.

Similarly, if you export (dllexport, dllimport) a managed function, a native entry point is generated and any function that imports and calls that function will call through the native entry point. To avoid double thunking in this situation, do not use native export/import semantics; simply reference the metadata via **#using** (see The #using Directive).

In Visual C++ 2005 the compiler was updated to reduce unnecessary double thunking. For example, any function with a managed type in the signature (including return type) will implicitly be marked as **__clrcall**. For more information on double thunk elimination, see http://msdn.microsoft.com/msdnmag/issues/05/01/COptimizations/default.aspx.

Example

The following sample demonstrates double thunking. When compiled native (without **/clr**), the call to the virtual function in `main` generates one call to `T`'s copy constructor and one call to the destructor. Similar behavior is achieved when the virtual function is declared with **/clr** and **__clrcall**. However, when just compiled with **/clr**, the function call generates a call to the copy constructor but there is another call to the copy constructor due to the native-to-managed thunk.

```
// double_thunking.cpp
// compile with: /clr
#include <stdio.h>
struct T {
   T() {
      puts(__FUNCSIG__);
   }

   T(const T&) {
      puts(__FUNCSIG__);
   }

   ~T() {
      puts(__FUNCSIG__);
   }

   T& operator=(const T&) {
      puts(__FUNCSIG__);
      return *this;
   }
};

struct S {
   virtual void /* __clrcall */ f(T t) {};
} s;

int main() {
   S* pS = &s;
   T t;

   printf("calling struct S\n");
   pS->f(t);
   printf("after calling struct S\n");
```

```
}
```

Sample Output

```
__thiscall T::T(void)
calling struct S
__thiscall T::T(const struct T &)
__thiscall T::T(const struct T &)
__thiscall T::~T(void)
__thiscall T::~T(void)
after calling struct S
__thiscall T::~T(void)
```

Example

The previous sample demonstrated the existence of double thunking. This sample shows its effect. The **for** loop calls the virtual function and the program reports execution time. The slowest time is reported when the program is compiled with **/clr**. The fastest times are reported when compiling without **/clr** or if the virtual function is declared with **__clrcall**.

```
// double_thunking_2.cpp
// compile with: /clr
#include <time.h>
#include <stdio.h>

#pragma unmanaged
struct T {
   T() {}
   T(const T&) {}
   ~T() {}
   T& operator=(const T&) { return *this; }
};

struct S {
   virtual void /* __clrcall */ f(T t) {};
} s;

int main() {
   S* pS = &s;
   T t;
   clock_t start, finish;
   double  duration;
   start = clock();

   for ( int i = 0 ; i < 1000000 ; i++ )
      pS->f(t);

   finish = clock();
   duration = (double)(finish - start) / (CLOCKS_PER_SEC);
   printf( "%2.1f seconds\n", duration );
   printf("after calling struct S\n");
}
```

Sample Output

```
4.2 seconds
after calling struct S
```

See Also

Concepts

Mixed (Native and Managed) Assemblies

Avoiding Exceptions on CLR Shutdown When Consuming COM Objects Built with /clr

Once the common language runtime (CLR) enters shutdown mode, native functions have limited access to CLR services. When attempting to call Release on a COM object compiled with **/clr**, the CLR transitions to native code and then transitions back into managed code to service the IUnknown::Release call (which is defined in managed code). The CLR prevents the call back into managed code since it is in shutdown mode.

To resolve this, ensure that destructors called from Release methods only contain native code.

See Also

Concepts

Mixed (Native and Managed) Assemblies

How to: Create a Partially Trusted Application by Removing Dependency on msvcm80.dll

This topic discusses how to create a partially trusted Common Language Runtime application using Visual C++ by removing dependency on msvcm80.dll.

A Visual C++ application built with **/clr** will have a dependency on msvcm80.dll, which is part of the C-Runtime Library. When you want your application to be used in a partial trust environment, the CLR will enforce certain code access security rules on your DLL. Therefore, it will be necessary to remove this dependency because msvcm80.dll contains native code and code access security policy cannot be enforced on it.

If your application does not use any functionality of the C-Runtime Library and you would like to remove the dependency on this library from your code, you will have to use the **/NODEFAULTLIB:msvcmrt.lib** linker option and link with either ptrustm.lib or ptrustmd.lib. These libraries contain object files for initialization and uninitialization of an application, exception classes used by the initialization code, and managed exception handling code. Linking in one of these libraries will remove any dependency on msvcm80.dll.

Note
The order of assembly uninitialization might differ for applications that use the ptrust libraries. For normal applications, assemblies are usually unloaded in the reverse order that they are loaded, but this is not guaranteed. For partial trust applications, assemblies are usually unloaded in the same order that they are loaded. This, also, is not guaranteed.

To create a partially trusted mixed (/clr) application

1. To remove the dependency on msvcm80.dll, you must specify to the linker not to include this library by using the **/NODEFAULTLIB:msvcmrt.lib** linker option. For information on how to do this using the Visual Studio development environment or programmatically, see /NODEFAULTLIB (Ignore Libraries).
2. Add one of the ptrustm libraries to the linker input dependencies. Use ptrustm.lib if you are building your application in release mode. For debug mode, use ptrustmd.lib. For information on how to do this using the Visual Studio development environment or programmatically, see .lib Files as Linker Input.

See Also

Reference

/link (Pass Options to Linker)

Concepts

Mixed (Native and Managed) Assemblies

Initialization of Mixed Assemblies

Library Support for Mixed Assemblies

Other Resources

Security in Native and .NET Framework Code

Using Native COM Servers from .NET

This section describes the available options for using existing COM components from .NET applications and outlines the advantages and disadvantages of each approach. Generally, the recommended method is C++ Interop.

Using TLBIMP

The .NET Framework SDK Type Library Importer (Tlbimp.exe) is a tool that exposes a COM type library as an assembly called an interop assembly. This assembly defines managed equivalents, or wrappers, for each COM interface in a given type library.

When methods of the interop assembly are called, a managed-to-unmanaged transition is executed and control is passed to the COM component. Likewise, when the unmanaged COM function returns, an unmanaged-to-managed transition is executed. By default, the COM HRESULT is checked for failure, and an exception is thrown if the HRESULT does not indicate success. Similarly, COM component initialization and interface queries are performed by the interop assembly and therefore hidden from the calling code.

Interop assemblies do not replace the COM components they represent; the unmanaged COM functions remain in the COM component, so the component must be installed and registered on target computers or calls to the interop assembly fail.

Using **`Tlbimp`** is the easiest way to use a COM component from managed code, but there are some serious disadvantages, especially for large and/or complex COM interfaces. These disadvantages are:

- **`Tlbimp`** generates managed interfaces for every COM interface in the type library. There is no way to suppress this behavior, so the resulting assemblies can be very large. (For example, the **`Tlbimp`**-generated interop assembly for Mshtml.dll is more than 8 MB.) There is also no way to hide an interface that is meant only for use within the COM component.
- **`Tlbimp`** supports a limited number of data types. Usually unsupported types are imported into the managed world as the generic, non type-safe IntPtr type, requiring error-prone and tedious marshaling code to use the assembly, but sometimes Tlbimp.exe cannot expose members of an interface at all.
- **`Tlbimp`** generates a separate interop assembly, which it must be deployed with the final application.

If these disadvantages are acceptable, see How to: Use Native COM Servers with TLBIMP for an example.

Modifying MSIL

The disadvantages of **`Tlbimp`** can be mitigated somewhat by disassembling the interop assembly — using the MSIL Disassembler (Ildasm.exe) — editing the MSIL to remove unnecessary interface definitions and replace argument types, and then reassembling the MSIL with the MSIL Assembler (Ilasm.exe). This process is error-prone and requires knowledge of MSIL, unmanaged types, and .NET types. Furthermore, this has to be done again if the COM interfaces are updated.

C++ Interop

The disadvantages of **`Tlbimp`** — and of editing MSIL— can be avoided entirely in Visual C++ because, unlike Visual Basic and C#, Visual C++ can use COM objects directly using the usual COM mechanisms (such as **CoCreateInstance** and **QueryInterface**). This is possible due to C++ Interop features that cause the compiler to automatically insert the transition code to move from managed to unmanaged functions and back again.

Using C++ Interop, COM components can be used as they are normally used or they can be wrapped inside C++ classes. These wrapper classes are called custom runtime callable wrappers, or CRCWs, and they have two advantages over using COM directly in application code:

- The resulting class can be used from languages other than Visual C++.
- The details of the COM interface can be hidden from the managed client code. .NET data types can be used in place of native types, and the details of data marshaling can be performed transparently inside the CRCW.

Using Visual C++ to wrap COM interfaces is demonstrated in How to: Use Native COM Servers with CRCWs.

Regardless of whether COM is used directly or through a CRCW, argument types other than simple, blittable types must be marshaled. For information about data marshaling, see Using C++ Interop (Implicit PInvoke).

Note

MFC applications must be initialized as single threaded apartment (STA). If you call CoInitializeEx in your **InitInstance** override, specify **COINIT_APARTMENTTHREADED** (rather than **COINIT_MULTITHREADED**). For more information, see PRB: MFC Application Stops Responding When You Initialize the Application as a Multithreaded Apartment (828643) at http://support.microsoft.com/default.aspx?scid=kb;en-us;828643.

See Also

Other Resources

Native and .NET Interoperability

How to: Use Native COM Servers with TLBIMP

Unlike other .NET languages, Visual C++ provides interop features that allow unmanaged functions, including COM interfaces, to be used directly. This allows developers to avoid Tlbimp.exe and its disadvantages. For more information, see Using Native COM Servers from .NET.

This topic demonstrates how COM objects can be used from managed code the Type Library Importer (Tlbimp.exe) tool. This .NET Framework tool, given a COM type library (or just the DLL containing both the COM component and the type library), generates an managed interop assembly that can be used from any .NET language. However, this technique, especially for large and/or complex COM type libraries, is not recommended for Visual C++ programmers.

The following example contains the steps and code required to use COM interfaces defined in Quartz.dll, which is installed in the Windows/System32 directory in Windows XP. These interfaces encapsulate DirectShow functionality to allow the playback of AVI files. To execute Tlbimp.exe from the command line, you need to add the .NET Framework tools to the system path by executing Sdkvars.bat in the C:\Program Files\Microsoft.NET\SDK\v2.0\Bin directory.

To generate the interop assembly

1. In a command prompt window, in the c:\windows\system32 directory, execute the command **`tlbimp quartz.dll`**. (The name of the resulting interop assembly is based on the name of the COM type library; in this case the resulting file is **`QuartzTypeLib.dll`**.)

2. Move the interop assembly to a directory where the application that uses it will be executed.

Example

The following code defines a console application that uses the previously generated interop assembly to display an AVI file.

Execute the resulting .exe file with the name of a valid AVI file, and the file is rendered in a window.

```
// AVIPlayer.cpp
// compile with: /clr
#using <QuartzTypeLib.dll>
using namespace QuartzTypeLib;

using namespace System;

void DisplayUsage() {
   Console::WriteLine("AVIPlayer: Plays AVI files.");
   Console::WriteLine("Usage: AVIPlayer.EXE <filename>");
}

int main() {
   array<String^>^ args = Environment::GetCommandLineArgs();
   if (args->Length != 2) {
      DisplayUsage();
      return 0;
   }

   String^ filename = args[1];
   if (filename->Equals("/?")) {
      DisplayUsage();
      return 0;
   }

   FilgraphManager^ graphManager = gcnew FilgraphManager();
   IMediaControl^ mc = dynamic_cast<IMediaControl^>(graphManager);

   mc->RenderFile(filename);
   mc->Run();

   Console::WriteLine("press any key");
   Console::ReadLine();
}
```

See Also

Reference

Using Native COM Servers from .NET

How to: Use Native COM Servers with CRCWs

Unlike other .NET languages, Visual C++ provides interop features that allow unmanaged APIs, including COM interfaces, to be accessed directly and seamlessly. For COM interop in particular, this provides powerful advantages.

Example

Like the example in How to: Use Native COM Servers with TLBIMP, this example uses the COM interfaces defined in Quartz.dll (found in the `C:\window\System32` directory) to play AVI files. In this case, however, C++ interop is used instead of a separate interop assembly generated with Tlbimp.exe. There are several advantages to this technique. In this case the interop code is built into the application, so there is no dependency on a separate assembly. Also, the exposed managed interface is customized to be more .NET-like. For example, the RenderFile method takes a System.String instead of a char*. The managed version of the COM interface is called a Custom Runtime Callable Wrapper (CRCW).

Writing CRCWs does not require an interop assembly, but it does require header files that define the COM interfaces. For COM components that include type libraries, these headers can be generated using the MIDL Compiler.

The first portion of the following code example defines the custom wrapper, which exposes the members that will be exposed to the managed application. The second portion is a console application that uses the custom wrapper to play AVI files.

Execute the resulting .exe file with the name of a valid AVI file and the file is rendered in a window.

```
// use_native_COM_servers_with_CRCWs.cpp
// compile with: /clr
// processor: x86
#include <comdef.h>
#import "quartz.tlb" no_namespace

using namespace System;
using namespace System::Runtime::InteropServices;

//_COM_SMARTPTR_TYPEDEF(IMediaControl, IID_IMediaControl);

ref struct Player : public IDisposable {
   Player() : fm((new IMediaControlPtr())) {
      fm->CreateInstance(__uuidof(FilgraphManager), 0, CLSCTX_INPROC_SERVER);
   if ((*fm) == 0)
      throw gcnew Exception("Could not create COM object");
   }

   ~Player() {
      this->!Player();
   }

   !Player() {
    (*fm).Release();
      delete fm;
   }

   void RenderFile(String^ file) {
      IntPtr ip = Marshal::StringToBSTR(file);
      BSTR b = static_cast<BSTR>(ip.ToPointer());
       (*fm)->RenderFile(b);
      Marshal::FreeBSTR(ip);
   }

   void Run() {
      (*fm)->Run();
   }

private:
   IMediaControlPtr* fm;
};

void DisplayUsage() {
```

```
   Console::WriteLine("AVIPlayer2: Plays AVI files.");
   Console::WriteLine("Usage: AVIPlayer2.EXE <filename>");
}

int main() {
   array<String^>^ args = Environment::GetCommandLineArgs();

   if (args->Length != 2) {
      DisplayUsage();
      return 0;
   }

   String^ filename = args[1];
   if (filename->Equals("/?")) {
      DisplayUsage();
      return 0;
   }

   Player^ player = gcnew Player;

   player->RenderFile(filename);
   player->Run();

   Console::WriteLine("press any key");
   Console::ReadLine();
}
```

See Also

Reference

Using Native COM Servers from .NET

Using a Windows Form User Control in MFC

Using the MFC Windows Forms support classes, you can host Windows Forms controls within your MFC applications as an ActiveX control within MFC dialog boxes or views. In addition, Windows Forms forms can be hosted as MFC dialog boxes.

The following sections describe how to:

- Host a Windows Forms control in an MFC dialog box.
- Host a Windows Forms user control as an MFC view.
- Host a Windows Forms form as an MFC dialog box.

Note
MFC Windows Forms integration works only in projects that link dynamically with MFC (projects in which AFXDLL is defined) .

Note
When you build your application using a private (modified) copy of the MFC Windows Forms interfaces DLL (mfcmifc80.dll), it will fail to install in the GAC unless you replace the Microsoft key with your own vendor key. For more information on asse mbly signing, see Programming with Assemblies and Strong Name Assemblies (Assembly Signing).

For sample applications using Windows Forms, see
BirthdayPicker Sample: Demonstrates .NET Framework Resources with Windows Forms,
Calculator Sample: Windows Forms Pocket Calculator, and Scribble Sample: MDI Drawing Application.

For a sample application that shows Windows Forms used with MFC, see MFC and WinForms Int e gration.

In This Section

Hosting a Windows Form User Control in an MFC Dialog Box

Hosting a Windows Form User Control as an MFC View

Hosting a Windows Form User Control as an MFC Dialog Box

Reference

CWinFormsControl Class

CWinFormsDialog Class

CWinFormsView Class

ICommandSource Interface

ICommandTarget Interface

ICommandUI Interface

IView Interface

CommandHandler Delegate

CommandUIHandler Delegate

DDX_ManagedControl

UICheckState Enumeration

Related Sections

Windows Forms

Windows Forms Controls

ASP.NET User Controls

See Also

Concepts

Form Views (MFC)

Other Resources

User Interface

Windows Forms/MFC Programming Differences

The topics in Using a Windows Form User Control in MFC describe the MFC support for Windows Forms. If you are not familiar with .NET Frameworks or MFC programming, this topic provides background information about programming differences between the two.

Windows Forms is for creating Microsoft Windows applications on the .NET Framework. This framework provides a modern, object-oriented, extensible set of classes that enable you to develop rich Windows-based applications. With Windows Forms, you are able to create a rich client application that can access a wide variety of data sources and provide data-display and data-editing facilities using Windows Forms controls.

However, if you are accustomed to MFC, you might be used to creating certain types of applications that are not yet explicitly supported in Windows Forms. Windows Forms applications are equivalent to MFC dialog applications. However, they do not provide the infrastructure to directly support other MFC application types like OLE document server/container, ActiveX documents, the Document/View support for single-document interface (SDI), multiple-document interface (MDI), and multiple top-level interface (MTI). You can write your own logic to create these applications.

For more information about Windows Forms applications, see Introduction to Windows Forms.

For a sample application that shows Windows Forms used with MFC, see MFC and WinForms Integration.

The following MFC view or document and command routing features have no equivalents in Windows Forms:

- Shell integration

  MFC handles the dynamic data exchange (DDE) commands and command-line arguments that the shell uses when you right-click a document and select such verbs as Open, Edit, or Print. Windows Forms has no shell integration and does not respond to shell verbs.

- Document templates

  In MFC, document templates associate a view, which is contained in a frame window (in MDI, SDI, or MTI mode), with the document you opened. Windows Forms has no equivalent to document templates.

- Documents

  MFC registers document file types and processes the document type when opening a document from the shell. Windows Forms has no document support.

- Document states

  MFC maintains dirty states for the document. Therefore, when you close the application, close the last view that contains the application, or exit from Windows, MFC prompts you to save the document. Windows Forms has no equivalent support.

- Commands

  MFC has the concept of commands. The menu bar, toolbar, and context menu can all invoke the same command, for example, Cut and Copy. In Windows Forms, commands are tightly bound events from a particular UI element (such as a menu item); therefore, you have to hook up all the command events explicitly. You can also handle multiple events with a single handler in Windows Forms. For more information, see
  Connecting Multiple Events to a Single Event Handler in Windows Forms.

- Command routing

  MFC command routing enables the active view or document to process commands. Because the same command often has different meanings for different views (for example, Copy behaves differently in text edit view than in a graphics editor), the commands need to be handled by the active view. Because Windows Forms menus and toolbars have no inherent understanding of the active view, you cannot have a different handler for each view type for your **MenuItem.Click** events without writing additional internal code.

- Command update mechanism

  MFC has a command update mechanism. Therefore, the active view or document is responsible for the state of the UI elements (for example, enabling or disabling a menu item or tool button, and checked states). Windows Forms has no equivalent of a command update mechanism.

See Also

Reference

Windows Forms Walkthroughs and How-to Topics

Other Resources

Using a Windows Form User Control in MFC

Hosting a Windows Form User Control in an MFC Dialog Box

MFC hosts a Windows Forms control as a special kind of ActiveX control and communicates with the control using ActiveX interfaces (for example, IOleObject) and properties and methods of the Control class. In general, you should use .NET Framework properties and methods to operate on the control.

For a sample application that shows Windows Forms used with MFC, see MFC and WinForms Integration.

Note
In the current release, a **CDialogBar** object cannot host Windows Forms controls.

In This Section

How to: Create the User Control and Host in a Dialog Box

How to: Do DDX/DDV Data Binding with Windows Forms

How to: Sink Windows Forms Events from Native C++ Classes

Reference

CWinFormsControl Class | CDialog Class | CWnd Class | **Control**

See Also

Tasks

Hosting a Windows Form User Control as an MFC Dialog Box

Concepts

Windows Forms/MFC Programming Differences

Other Resources

Using a Windows Form User Control in MFC

Hosting a Windows Form User Control as an MFC View

How to: Create the User Control and Host in a Dialog Box

The procedure in this topic assumes you are creating a new dialog-based (CDialog Class) MFC project, but you can also add support for a Windows Form control to an existing MFC dialog.

To create the .NET user control

1. Create a new Visual C# Windows Control Library project named WindowsControlLibrary1.

   From the **File** menu, select **New**, then **Project**. In the **Visual C#** folder, Select the **Windows Control Library** icon.

   Accept the default project name of WindowsControlLibrary1 by clicking **OK**.

   The default name of the .NET control will be `UserControl1`.

2. Add child controls to `UserControl1`.

   In **Toolbox**, open the **All Windows Forms** list. Drag a Button control to the UserControl1 design surface.

   Also add a TextBox control.

3. Change the declarations of the TextBox and Button to public from private in the file UserControl1.Designer.cs.

4. Build the project.

   From the **Build** menu, click **Build Solution**.

   You may want to note the full path with file name of the generated DLL in the Build log because you will enter that information into the MFC application.

To create the MFC host application

1. Create a new MFC Application project.

   From the **File** menu, select **New**, then **Project**. In the **Visual C++** folder, select the **MFC Application** icon.

   In the **Name** box, enter `MFC01`. Change the Solution setting to **Add to Solution**. Click **OK**. The **MFC Application Wizard** appears.

   In the **MFC Application Wizard**, select Application Type. Choose **Dialog based**. Accept the remaining defaults and click **Finish**. This will create an MFC application with an MFC Dialog.

2. Add a placeholder control to the MFC dialog box.

   Click the **Resource View** tab. In Resource View, double-click on `IDD_MFC01_DIALOG`. The dialog resource appears in Resource Editor.

   In **Toolbox**, open the **Dialog Editor** list. Drag a Static Text control to the dialog resource. The Static Text control will serve as a placeholder for the .NET Windows Forms control. Resize it to approximately the size of the Windows Forms control.

   Change the ID of the Static Text control to `IDC_CTRL1` in the **Properties** window and change the **TabStop** property to **True**.

3. Configure the project.

   In **Solution Explorer**, right-click the `MFC01` project node, and select **Properties** from the context menu. The **Property Pages** dialog box appears.

   In the **Property Pages** dialog box, in the **Configuration Properties** tree control, select **General**, then in the **Project Defaults** section, set **Common Language Runtime support** to **Common Language Runtime Support (/clr)**. Click **OK**.

4. Add a reference to the .NET control.

   In **Solution Explorer**, right-click the `MFC01` project node and select **References**. In the **Property Page**, click **Add New Reference**,select WindowsControlLibrary1 (under the **Projects** tab), and click **OK**. This adds a reference in the form of a

/FU compiler option so that the program will compile; it also copies WindowsControlLibrary1.dll into the `MFC01` project directory so that the program will run.

5. In stdafx.h, find this line:

```
#endif // _AFX_NO_AFXCMN_SUPPORT
```

Add these lines above it:

```
#include <afxwinforms.h>   // MFC Windows Forms support
```

6. Add code to create the managed control.

First, declare the managed control. In MFC01Dlg.h, go to the declaration of the dialog class, and add a data member for the user control in Protected scope as follows:

```
class CMFC01Dlg : public CDialog
{
   // ...
   // Data member for the .NET User Control:
   CWinFormsControl<WindowsControlLibrary1::UserControl1> m_ctrl1;
```

Next, provide an implementation for the managed control. In MFC01Dlg.cpp, in the dialog override of `CMFC01Dlg::DoDataExchange` generated by the MFC Application wizard (not `CAboutDlg::DoDataExchange`, which is in the same file), add the following code to create the managed control and associate it with the static place holder IDC_CTRL1:

```
void CMFC01Dlg::DoDataExchange(CDataExchange* pDX)
{
   CDialog::DoDataExchange(pDX);
   DDX_ManagedControl(pDX, IDC_CTRL1, m_ctrl1);
}
```

7. Build and run the project.

In **Solution Explorer**, right-click MFC01 and select **Set as StartUp Project**.

From the **Build** menu, click **Build Solution**.

From the **Debug** menu, click **Start without debugging**. You will now see the MFC dialog box displaying the Windows Form control.

See Also

Other Resources

Hosting a Windows Form User Control in an MFC Dialog Box

How to: Do DDX/DDV Data Binding with Windows Forms

DDX_ManagedControl calls CWinFormsControl::CreateManagedControl to create a control matching the resource control ID. If you use **DDX_ManagedControl** for a **CWinFormsControl** control (in wizard-generated code), you should not call **CreateManagedControl** explicitly for the same control.

Call **DDX_ManagedControl** in CWnd::DoDataExchange to create controls from resource IDs. For data exchange, you do not need to use the DDX/DDV functions with Windows Forms controls. Instead, you can place code to access the properties of the managed control in the **DoDataExchange** method of your dialog (or view) class, as in the following example.

The following example shows how to bind a native C++ string to a .NET user control.

Example

The following is an example of DDX/DDV data binding of an MFC string `m_str` with the user-defined `NameText` property of a .NET user control.

The control is created when CDialog::OnInitDialog calls `CMyDlg::DoDataExchange` for the first time, so any code that references `m_UserControl` must come after the **DDX_ManagedControl** call.

You can implement this code in the MFC01 application you created in How to: Create the User Control and Host in a Dialog Box.

Put the following code in the declaration of CMFC01Dlg:

```
class CMFC01Dlg : public CDialog
{
   CWinFormsControl<WindowsControlLibrary1::UserControl1> m_MyControl;
   CString m_str;
};
```

Put the following code in the implementation of CMFC01Dlg:

```
void CMFC01Dlg::DoDataExchange(CDataExchange* pDX)
{
   CDialog::DoDataExchange(pDX);
   DDX_ManagedControl(pDX, IDC_CTRL1, m_MyControl);

   if (pDX->m_bSaveAndValidate) {
      m_str = m_MyControl->textBox1->Text;
   } else
   {
      m_MyControl->textBox1->Text = gcnew System::String(m_str);
   }
}
```

Now we will add the handler method for a click on the OK button. Click the **Resource View** tab. In Resource View, double-click on `IDD_MFC01_DIALOG`. The dialog resource appears in Resource Editor. Then double click the OK button..

Define the handler as follows.

```
void CMFC01Dlg::OnBnClickedOk()
{
   AfxMessageBox(CString(m_MyControl.GetControl()->textBox1->Text));
   OnOK();
}
```

And add the following line to the implementation of BOOL CMFC01Dlg::OnInitDialog().

```
   m_MyControl.GetControl()->textBox1->Text = "hello";
```

You can now build and run the application. Notice that any text in the text box will be displayed in a pop-up message box when

the application closes.

See Also

Reference

CWinFormsControl Class
DDX_ManagedControl
CWnd::DoDataExchange

How to: Sink Windows Forms Events from Native C++ Classes

You can enable native C++ classes to receive callbacks from managed events raised from Windows Forms controls or other forms with the MFC macro map format. Sinking events in views and dialogs is similar to doing the same task for controls.

To do this, you need to:

- Attach an **OnClick** event handler to the control using MAKE_DELEGATE.
- Create a delegate map using BEGIN_DELEGATE_MAP, END_DELEGATE_MAP, and EVENT_DELEGATE_ENTRY.

Example

This sample continues the work you did in How to: Do DDX/DDV Data Binding with Windows Forms.

Now, you will associate your MFC control (`m_MyControl`) with a managed event handler delegate called **OnClick** for the managed Click event.

Add the first line of code to implementation of BOOL CMFC01Dlg::OnInitDialog.

Add the delegate map and OnClick definition to the public section in the declaration of class CMFC01Dlg : public CDialog.

```
m_MyControl.GetControl()->button1->Click += MAKE_DELEGATE( System::EventHandler, OnClick );

// delegate map
BEGIN_DELEGATE_MAP( CMFC01Dlg )
EVENT_DELEGATE_ENTRY( OnClick, System::Object^, System::EventArgs^ )
END_DELEGATE_MAP()

   void OnClick( System::Object^ sender, System::EventArgs^ e ) {}
```

See Also

Reference

MAKE_DELEGATE
BEGIN_DELEGATE_MAP
END_DELEGATE_MAP
EVENT_DELEGATE_ENTRY

Hosting a Windows Form User Control as an MFC View

MFC uses the CWinFormsView class to host a Windows Forms user control within an MFC view. MFC Windows Forms views are ActiveX controls. The user control is hosted as a child of the native view and occupies the entire client area of the native view.

The end result is similar to the model used by the CFormView Class. This allows you to take advantage of the Windows Forms designer and run time to create rich form-based views.

Because MFC Windows Forms views are ActiveX controls, they do not have the same **hwnd** as MFC views. Also they cannot be passed as a pointer to a CView view. In general, use .NET Frameworks methods to work with Windows Forms views and rely less on Win32.

For a sample application that shows Windows Forms used with MFC, see MFC and WinForms Integration.

In This Section

How to: Create the User Control and Host MDI View

How to: Add Command Routing to the Windows Forms Control

How to: Call Properties and Methods of the Windows Forms Control

See Also

Tasks

How to: Author Composite Controls

Other Resources

Using a Windows Form User Control in MFC

How to: Create the User Control and Host MDI View

The following procedures show how to create a .NET Frameworks user control, author the user control in a control class library (specifically, a Windows Control Library project), and then compile the project into an assembly. You will then consume the control from an MFC application that uses classes derived from CView Class and CWinFormsView Class.

For information about how to create a Windows Forms user control and author a control class library, see How to: Author User Controls.

Note
In some cases, WinForms controls, such as a third-party Grid control, might not behave reliably when hosted in an MFC application. A recommended workaround is to place a Windows Forms User Control in the MFC application and place the third-party Grid control inside the User control.

This procedure assumes that you created a Windows Forms Controls Library project named WindowsControlLibrary1, as per the procedure in How to: Create the User Control and Host in a Dialog Box.

To create the MFC host application

1. Create a new MFC Application project.

   On the **File** menu, select **New**, and then click **Project**. In the **Visual C++** folder, select **MFC Application**.

   In the **Name** box, enter `MFC02` and change the Solution setting to **Add to Solution**. Click **OK**.

   In the **MFC Application Wizard**, accept all the defaults, and then click **Finish**. This creates an MFC application with a Multiple Document Interface.

2. Configure the project.

   In **Solution Explorer**, right-click the `MFC02` project node, and select **Properties** from the context menu. The **Property Pages** dialog box appears.

   In the **Property Pages** dialog box, in the **Configuration Properties** tree control, select **General**, then in the **Project Defaults** section, set **Common Language Runtime support** to **Common Language Runtime Support (/clr)**. Click **OK**.

3. Add a reference to the .NET control.

   In **Solution Explorer**, right-click the `MFC02` project node and select **References**. In the **Property Page**, click **Add New Reference**,select WindowsControlLibrary1 (under the **Projects** tab), and click **OK**. This adds a reference in the form of a /FU compiler option so that the program will compile; it also copies WindowsControlLibrary1.dll into the `MFC02` project directory so that the program will run.

4. In stdafx.h, find this line:

   ```
   #endif // _AFX_NO_AFXCMN_SUPPORT
   ```

   Add these lines above it:

   ```
   #include <afxwinforms.h>   // MFC Windows Forms support
   ```

5. Modify the view class so that it inherits from CWinFormsView.

   In MFC02View.h, replace CView with CWinFormsView so that the code appears as follows:

   ```
   class CMFC02View : public CWinFormsView
   {
   };
   ```

If you want add additional views to your MDI application, you will need to call CWinApp::AddDocTemplate for each view you create.

6. Modify the MFC02View.cpp file to change CView to CWinFormsView in the IMPLEMENT_DYNCREATE macro and message map and replace the existing empty constructor with the constructor shown below:

```
IMPLEMENT_DYNCREATE(CMFC02View, CWinFormsView)

CMFC02View::CMFC02View(): CWinFormsView(WindowsControlLibrary1::UserControl1::typeid)
{
}
BEGIN_MESSAGE_MAP(CMFC02View, CWinFormsView)
//leave existing body as is
END_MESSAGE_MAP()
```

7. Build and run the project.

   In **Solution Explorer**, right-click MFC02 and select **Set as StartUp Project**.

   On the **Build** menu, click **Build Solution**.

   On the **Debug** menu, click **Start without debugging**.

See Also

Other Resources

Hosting a Windows Form User Control as an MFC View

How to: Add Command Routing to the Windows Forms Control

CWinFormsView routes commands and update command UI messages to the user control to allow it to handle MFC commands (for example, frame menu items and toolbar buttons).

The user control uses ICommandTarget::Initialize to store a reference to the command source object in `m_CmdSrc`, as shown in the following example. To use **ICommandTarget** you need to add a reference to mfcmifc80.dll.

CWinFormsView handles several of the common MFC view notifications by forwarding them to the managed user control. These notifications include the OnInitialUpdate, OnUpdate and OnActivateView methods of the IView Interface.

To create the MFC host application

1. Open Windows Forms Control Library you created in How to: Create the User Control and Host in a Dialog Box.
2. Add a reference to mfcmifc80.dll, which you can do by right-clicking on the project node in **Solution Explorer**, selecting **Add Reference**, and then browsing to Microsoft Visual Studio 8\VC\atlmfc\lib.
3. Open UserControl1.Designer.cs and add the following using statement:

   ```
   using namespace Microsoft::VisualC::MFC;
   ```

4. Also, in UserControl1.Designer.cs, change this line:

   ```
   partial class UserControl1
   ```

   to this:

   ```
   partial class UserControl1 : System.Windows.Forms.UserControl, ICommandTarget
   ```

5. Add this as the first line of the class definition for `UserControl1`:

   ```
   private ICommandSource m_CmdSrc;
   ```

6. Add the following method definitions to `UserControl1` (we will create the ID of the MFC control in the next step):

   ```
   public void Initialize (ICommandSource cmdSrc)
   {
      m_CmdSrc = cmdSrc;
      // need ID of control in MFC dialog and callback function
      m_CmdSrc.AddCommandHandler(32771, new CommandHandler (singleMenuHandler));
   }

   private void singleMenuHandler (uint cmdUI)
   {
      // User command handler code
      System.Windows.Forms.MessageBox.Show("Got FILE_NEW");
   }
   ```

7. Open the MFC application you created in How to: Create the User Control and Host in a Dialog Box.
8. Add a menu option that will invoke `singleMenuHandler`.

   Go to **Resource View** (CTRL+SHIFT+E), expand the **Menu** folder, and then double click on **IDR_MFC02TYPE**. This displays the menu editor.

Add a menu option at the bottom of the **View** menu. Save the file.

In **Solution Explorer**, open the Resource.h file, copy the value for the menu option you just added, and paste that value as the first parameter to the `m_CmdSrc.AddCommandHandler` call in the C# project's `Initialize` method.

9. Build and run the project.

   On the **Build** menu, click **Build Solution**.

   On the **Debug** menu, click **Start without debugging**.

   Select the menu option you added. Notice that the method in the .dll is called.

See Also

Reference

ICommandSource Interface
ICommandTarget Interface
CommandHandler Delegate

Other Resources

Hosting a Windows Form User Control as an MFC View

How to: Call Properties and Methods of the Windows Forms Control

Because CWinFormsView::GetControl returns a pointer to System.Windows.Forms.Control, and not a pointer to `WindowsControlLibrary1::UserControl1`, it is advisable to add a member of the user control type and initialize it in IView::OnInitialUpdate. Now you can call methods and properties using `m_ViewControl`.

To create the MFC host application

1. Open the MFC application you created in How to: Create the User Control and Host in a Dialog Box.
2. Add the following line to the public overrides section of the `CMFC02View` class declaration in MFC02View.h.

   ```
   gcroot<WindowsControlLibrary1::UserControl1 ^> m_ViewControl;
   ```

3. Add an override for OnInitialupdate.

   Display the **Properties** window (F4). In **Class View** (CTRL+SHIFT+C), select CMFC02View class. In the **Properties** window, select the icon for Overrides. Scoll down the list to OnInitialUpdate. Click on the drop down list and select <Add>. In MFC02View.cpp. make sure the body of the OnInitialUpdate function is as follows:

   ```
   CWinFormsView::OnInitialUpdate();
   m_ViewControl = safe_cast<WindowsControlLibrary1::UserControl1 ^>(this->GetControl());
   m_ViewControl->textBox1->Text = gcnew System::String("hi");
   ```

4. Build and run the project.

   On the **Build** menu, click **Build Solution**.

   On the **Debug** menu, click **Start without debugging**.

   Notice that the text box is now initialized.

See Also

Other Resources

Hosting a Windows Form User Control as an MFC View

Hosting a Windows Form User Control as an MFC Dialog Box

MFC provides the template class CWinFormsDialog to allow you to host a Windows Forms user control (UserControl) in a modal or modeless MFC dialog box. **CWinFormsDialog** is derived from the MFC class CDialog, so the dialog box can be launched as modal or modeless.

The process that **CWinFormsDialog** uses to host the user control is the similar to that described in Hosting a Windows Form User Control in an MFC Dialog Box. However, **CWinFormsDialog** manages the initialization and hosting of the user control so that it does not have to be programmed manually.

For a sample application that shows Windows Forms used with MFC, see MFC and WinForms Integration.

To create the MFC host application

1. Create a new MFC Application project.

   On the **File** menu, select **New**, and then click **Project**. In the **Visual C++** folder, select **MFC Application**.

   In the **Name** box, enter `MFC03` and change the Solution setting to **Add to Solution**. Click **OK**.

   In the **MFC Application Wizard**, accept all the defaults, and then click **Finish**. This creates an MFC application with a Multiple Document Interface.

2. Configure the project.

   In **Solution Explorer**, right-click the `MFC03` project node, and select **Properties** from the context menu. The **Property Pages** dialog box appears.

   In the **Property Pages** dialog box, in the **Configuration Properties** tree control, select **General**, then in the **Project Defaults** section, set **Common Language Runtime support** to **Common Language Runtime Support (/clr)**. Click **OK**.

3. Add a reference to the .NET control.

   In **Solution Explorer**, right-click the `MFC03` project node and select **References**. In the **Property Page**, click **Add New Reference**,select WindowsControlLibrary1 (under the **Projects** tab), and click **OK**. This adds a reference in the form of a /FU compiler option so that the program will compile; it also copies WindowsControlLibrary1.dll into the `MFC03` project directory so that the program will run.

4. Add `#include <afxwinforms.h>` to stdafx.h, at the end of the existing **#include** statements.

5. Add a new class that subclasses **CDialog**.

   Right click on project name and add an MFC class (called CHostForWinForm) that subclasses **CDialog**. Since you do not need the dialog box resource, you can delete the resource ID (select the Resource View, expand the Dialog folder and delete IDD_HOSTFORWINFORM resource. Then, remove any references to the ID in code.).

6. Replace **CDialog** in CHostForWinForm.h and CHostForWinForm.cpp files with **CWinFormsDialog<WindowsControlLibrary1::UserControl1>**.

7. Call DoModal on the CHostForWinForm class.

   In MFC03.cpp, add `#include "HostForWinForm.h"`.

   Before the return statement in the definition of CMFC03App::InitInstance, add:

   ```
   CHostForWinForm m_HostForWinForm;

   m_HostForWinForm.DoModal();
   ```

8. Build and run the project.

   On the **Build** menu, click **Build Solution**.

   On the **Debug** menu, click **Start without debugging**.

   Next you will add code to monitor the state of a control on the Windows Forms from the MFC application.

9. Add a handler for OnInitDialog.

   Display the **Properties** window (F4). In **Class View**, select CHostForWinForm. In the **Properties** window, select overrides and in the row for OnInitDialog, click in the left hand column and select < Add >. This adds the following line to CHostForWinForm.h:

```
virtual BOOL OnInitDialog();
```

10. Define OnInitDialog (in CHostForWinForm.cpp) as follows:

```
BOOL CHostForWinForm::OnInitDialog() {
   CWinFormsDialog<WindowsControlLibrary1::UserControl1>::OnInitDialog();
   GetControl()->button1->Click += MAKE_DELEGATE(System::EventHandler, OnButton1);
   return TRUE;
}
```

11. Next add the OnButton1 handler. Add the following lines to the public section of the CHostForWinForm class in CHostForWinForm.h:

```
virtual void OnButton1( System::Object^ sender, System::EventArgs^ e );

BEGIN_DELEGATE_MAP( CHostForWinForm )
   EVENT_DELEGATE_ENTRY( OnButton1, System::Object^, System::EventArgs^ );
END_DELEGATE_MAP()
```

   In CHostForWinForm.cpp, add this definition:

```
void CHostForWinForm::OnButton1( System::Object^ sender, System::EventArgs^ e )
{
   System::Windows::Forms::MessageBox::Show("test");
}
```

12. Build and run the project. When you click the button, which is on the Windows Form, code in the MFC application will run.

    Next you will add code to display from the MFC code the value in the text box on the Windows Form.

13. In the public section of the CHostForWinForm class in CHostForWinForm.h, add the following declaration:

```
CString m_sEditBoxOnWinForm;
```

14. In the definition of DoDataExchange in CHostForWinForm.cpp, add the following three lines to the end of the function:

```
if (pDX->m_bSaveAndValidate)
   m_sEditBoxOnWinForm = CString( GetControl()->textBox1->Text);
else
   GetControl()->textBox1->Text = gcnew System::String(m_sEditBoxOnWinForm);
```

15. In the definition of OnButton1 in CHostForWinForm.cpp, add the following three lines to the end of the function:

```
this->UpdateData(TRUE);
System::String ^ z = gcnew System::String(m_sEditBoxOnWinForm);
System::Windows::Forms::MessageBox::Show(z);
```

16. Build and run the project.

See Also

Reference

System.Windows.Forms.UserControl

Other Resources

Using a Windows Form User Control in MFC

Calling Native Functions from Managed Code

The common language runtime provides Platform Invocation Services, or PInvoke, that enables managed code to call C-style functions in native dynamic-linked libraries (DLLs). The same data marshaling is used as for COM interoperability with the runtime and for the "It Just Works," or IJW, mechanism.

For more information, see:

- Using Explicit PInvoke in C++ (DllImport Attribute)
- Using C++ Interop (Implicit PInvoke)
- A Closer Look at Platform Invoke

The samples in this section just illustrate how PInvoke can be used. PInvoke can simplify customized data marshaling because you provide marshaling information declaratively in attributes instead of writing procedural marshaling code.

PInvoke and the DllImport Attribute

The following example shows the use of PInvoke in a Visual C++ program. The native function puts is defined in msvcrt.dll. The DllImport attribute is used for the declaration of puts.

```
// platform_invocation_services.cpp
// compile with: /clr
using namespace System;
using namespace System::Runtime::InteropServices;

[DllImport("msvcrt", CharSet=CharSet::Ansi)]
extern "C" int puts(String ^);

int main() {
   String ^ pStr = "Hello World!";
   puts(pStr);
}
```

The following sample is equivalent to the previous sample, but uses IJW.

```
// platform_invocation_services_2.cpp
// compile with: /clr
using namespace System;
using namespace System::Runtime::InteropServices;

#include <stdio.h>

int main() {
   String ^ pStr = "Hello World!";
   char* pChars = (char*)Marshal::StringToHGlobalAnsi(pStr).ToPointer();
   puts(pChars);

   Marshal::FreeHGlobal((IntPtr)pChars);
}
```

Advantages of IJW

- There is no need to write **DLLImport** attribute declarations for the unmanaged APIs the program uses. Just include the header file and link with the import library.
- The IJW mechanism is slightly faster (for example, the IJW stubs do not need to check for the need to pin or copy data items since that is done explicitly by the developer).
- It clearly illustrates performance issues. In this case, the fact that you are translating from a Unicode string to an ANSI string and that you have an attendant memory allocation and deallocation. In this case, a developer writing the code using IJW would realize that calling **_putws** and using **PtrToStringChars** would be better for performance.

- If you need to call many unmanaged APIs using the same data, marshaling it once up front and passing the marshaled copy around is much more efficient than re-marshaling every time.

Disadvantages of IJW

- Marshaling needs to be specified explicitly in code rather than by attributes (which in many cases like this one have suitable defaults).
- The marshaling code is inline, where it is more invasive in the flow of the application logic.
- Since the explicit marshaling APIs return **IntPtr** types for 32-bit to 64-bit portability, extra **ToPointer** calls need to be used.

Like many places, the specific method exposed by C++ is the more efficient, explicit method, at the cost of some additional complexity.

If the application uses mainly unmanaged data types or if it calls more unmanaged APIs than .NET Framework APIs, using the IJW feature will generally be preferable. To call an occasional unmanaged API in a mostly managed application, the choice is more subtle.

Marshaling Arguments

With PInvoke, no marshaling is required between managed and C++ native primitive types with the same form. For example, no marshaling is required between Int32 and int, and Double and double.

Marshaling is required for types that do not have the same form. This includes char, string, and struct types. The following table shows the mappings used by the marshaler for various types.

wtypes.h	Visual C++	Visual C++ with /clr	Common Language Runtime
HANDLE	void *	void *	IntPtr, UIntPtr
BYTE	unsigned char	unsigned char	Byte
SHORT	short	short	Int16
WORD	unsigned short	unsigned short	UInt16
INT	int	int	Int32
UINT	unsigned int	unsigned int	UInt32
LONG	long	long	Int32
BOOL	long	bool	Boolean
DWORD	unsigned long	unsigned long	UInt32
ULONG	unsigned long	unsigned long	UInt32
CHAR	char	char	Char
LPCSTR	char *	String ^ [in], StringBuilder ^ [in, out]	String ^ [in], StringBuilder ^ [in, out]
LPCSTR	const char *	String ^	String
LPWSTR	wchar_t *	String ^ [in], StringBuilder ^ [in, out]	String ^ [in], StringBuilder ^ [in, out]
LPCWSTR	const wchar_t *	String ^	String

FLOAT	float	float	Single
DOUBLE	double	double	Double

The marshaler automatically pins memory allocated on the runtime heap if its address is passed to an unmanaged function. Pinning prevents the garbage collector from moving the allocated block of memory during compaction.

In the example shown earlier in this topic, the CharSet parameter of DllImport specifies how managed Strings should be marshaled; in this case, they should be marshaled to ANSI strings for the native side.

Marshaling information for individual arguments of a native function can be specified using the MarshalAs attribute. There are several choices for marshaling a String * argument: BStr, ANSIBStr, TBStr, LPStr, LPWStr, and LPTStr. The default is LPStr.

```
// platform_invocation_services_3.cpp
// compile with: /clr
using namespace System;
using namespace System::Runtime::InteropServices;

[DllImport("msvcrt", EntryPoint="puts")]
extern "C" int puts([MarshalAs(UnmanagedType::LPWStr)] String ^);

int main() {
   String ^ pStr = "Hello World!";
   puts(pStr);
}
```

In this example, the string is marshaled as a double-byte Unicode character string, LPWStr. The output is just the first letter of Hello World! because the second byte of the marshaled string is null, and puts interprets this as the end-of-string marker.

The MarshalAs attribute is in the System::Runtime::InteropServices namespace. The attribute can be used with other data types such as arrays.

PInvoke with Windows APIs

PInvoke is also convenient for calling functions in Windows.

In this example, a Visual C++ program interoperates with the MessageBox function that is part of the Win32 API.

```
// platform_invocation_services_4.cpp
// compile with: /clr /c
using namespace System;
using namespace System::Runtime::InteropServices;
typedef void* HWND;
[DllImport("user32", CharSet=CharSet::Ansi)]
extern "C" int MessageBox(HWND hWnd, String ^ pText, String ^ pCaption, unsigned int uType)
;

int main() {
   String ^ pText = "Hello World! ";
   String ^ pCaption = "PInvoke Test";
   MessageBox(0, pText, pCaption, 0);
}
```

The output is a message box with the title PInvoke Test containing the text Hello World!.

The marshaling information is also used by PInvoke to look up functions in the DLL. In user32.dll there is in fact no MessageBox function, but CharSet=CharSet::Ansi enables PInvoke to use MessageBoxA, the ANSI version, instead of MessageBoxW, which is the Unicode version. In general, using Unicode versions of unmanaged APIs should be preferred because that eliminates the translation overhead from the native Unicode format of .NET Framework string objects to ANSI.

When Not to Use PInvoke

Using PInvoke is not suitable for all C-style functions in DLLs. For example, suppose there is a function MakeSpecial in mylib.dll declared as follows.

```
char * MakeSpecial(char * pszString);
```

If we use PInvoke in a Visual C++ application, we might write something similar to:

```
[DllImport("mylib")]
extern "C" String * MakeSpecial([MarshalAs(UnmanagedType::LPStr)] String ^);
```

The difficulty here is that we cannot delete the memory for the unmanaged string returned by MakeSpecial. Other functions called via PInvoke return a pointer to an internal buffer that does not need to be deallocated by the user. In this case, using the IJW feature is the obvious choice.

Limitations of PInvoke

You cannot return the same exact pointer from a native function that you took as a parameter. If a native function returns the pointer that has been marshaled to it by PInvoke, memory corruption and exceptions may ensue.

```
__declspec(dllexport)
char* fstringA(char* param) {
   return param;
}
```

The following sample exhibits this problem, and even though the program may seem to give the correct output, the output is coming from memory that had been freed.

```
// platform_invocation_services_5.cpp
// compile with: /clr /c
using namespace System;
using namespace System::Runtime::InteropServices;
#include <limits.h>

ref struct MyPInvokeWrap {
public:
   [ DllImport("user32.dll", EntryPoint = "CharLower", CharSet = CharSet::Ansi) ]
   static String^ CharLower([In, Out] String ^);
};

int main() {
   String ^ strout = "AabCc";
   Console::WriteLine(strout);
   strout = MyPInvokeWrap::CharLower(strout);
   Console::WriteLine(strout);
}
```

Performance Considerations

PInvoke has an overhead of between 10 and 30 x86 instructions per call. In addition to this fixed cost, marshaling creates additional overhead. There is no marshaling cost between blittable types that have the same representation in managed and unmanaged code. For example, there is no cost to translate between int and Int32.

For higher performance, it may be necessary to have fewer PInvoke calls that marshal as much data as possible, rather than have more calls that marshal less data per call.

See Also

Other Resources

Native and .NET Interoperability

Using Explicit PInvoke in C++ (DllImport Attribute)

The .NET Framework provides explicit Platform Invoke (or PInvoke) features with the **Dllimport** attribute to allow managed applications to call unmanaged functions packaged inside DLLs. Explicit PInvoke is required for situations where unmanaged APIs are packaged as DLLs and the source code is not available. Calling Win32 functions, for example, requires PInvoke. Otherwise, use implicit P{Invoke; see Using C++ Interop (Implicit PInvoke) for more information.

PInvoke works by using DllImportAttribute. This attribute, which takes the name of the DLL as its first argument, is placed before a function declaration for each DLL entry point that will be used. The signature of the function must match the name of a function exported by the DLL (but some type conversion can be performed implicitly by defining the **DllImport** declarations in terms of managed types.)

The result is a managed entry point for each native DLL function that contains the necessary transition code (or thunk) and simple data conversions. Managed functions can then call into the DLL through these entry points. The code inserted into a module as the result of PInvoke is entirely managed and explicit PInvoke is supported for **/clr**, **/clr:pure**, and **/clr:safe** compilations. For more information, see Pure and Verifiable Code.

In This Section

- Calling Native Functions from Managed Code
- How to: Call Native DLLs from Managed Code Using PInvoke
- How to: Marshal Strings Using PInvoke
- How to: Marshal Structures Using PInvoke
- How to: Marshal Arrays Using PInvoke
- How to: Marshal Function Pointers Using PInvoke
- How to: Marshal Embedded Pointers Using PInvoke

See Also

Concepts

Calling Native Functions from Managed Code

How to: Call Native DLLs from Managed Code Using PInvoke

Functions that are implemented in unmanaged DLLs can be called from managed code using Platform Invoke (P/Invoke) functionality. If the source code for the DLL is not available, P/Invoke is the only option for interoperating. However, unlike other .NET languages, Visual C++ provides an alternative to P/Invoke. For more information, see Using C++ Interop (Implicit PInvoke).

Example

The following code example uses the Win32 GetSystemMetrics function to retrieve the current resolution of the screen in pixels.

For functions that use only intrinsic types as arguments and return values, no extra work is required. Other data types, such as function pointers, arrays, and structures, require additional attributes to ensure proper data marshaling.

Although it is not required, it is good practice to make P/Invoke declarations static members of a value class so that they do not exist in the global namespace, as demonstrated in this example.

```
// pinvoke_basic.cpp
// compile with: /clr
using namespace System;
using namespace System::Runtime::InteropServices;

value class Win32 {
public:
   [DllImport("User32.dll")]
   static int GetSystemMetrics(int);

   enum class SystemMetricIndex {
      // Same values as those defined in winuser.h.
      SM_CXSCREEN = 0,
      SM_CYSCREEN = 1
   };
};

int main() {
   int hRes = Win32::GetSystemMetrics( safe_cast<int>(Win32::SystemMetricIndex::SM_CXSCREEN
) );
   int vRes = Win32::GetSystemMetrics( safe_cast<int>(Win32::SystemMetricIndex::SM_CYSCREEN
) );
   Console::WriteLine("screen resolution: {0},{1}", hRes, vRes);
}
```

See Also

Other Resources

Using Explicit PInvoke in C++ (DllImport Attribute)

How to: Marshal Strings Using PInvoke

This topic explains how native functions that accept C-style strings can be called using the CLR string type System::String using .NET Framework Platform Invoke support. Visual C++ programmers are encouraged to use the C++ Interop features instead (when possible) because P/Invoke provides little compile-time error reporting, is not type-safe, and can be tedious to implement. If the unmanaged API is packaged as a DLL, and the source code is not available, then P/Invoke is the only option, but otherwise see Using C++ Interop (Implicit PInvoke).

Managed and unmanaged strings are laid out differently in memory, so passing strings from managed to unmanaged functions requires the MarshalAsAttribute attribute to instruct the compiler to insert the required conversion mechanisms for marshaling the string data correctly and safely.

As with functions that use only intrinsic data types, DllImportAttribute is used to declare managed entry points into the native functions, but--for passing strings--instead of defining these entry points as taking C-style strings, a handle to the String type can be used instead. This prompts the compiler to insert code that performs the required conversion. For each function argument in an unmanaged function that takes a string, the **MarshalAsAttribute** attribute should be used to indicate that the String object should be marshaled to the native function as a C-style string.

Example

The following code consists of a unmanaged and a managed module. The unmanaged module is a DLL that defines a function called TakesAString that accepts a C-style ANSI string in the form of a char*. The managed module is a command-line application that imports the TakesAString function, but defines it as taking a managed System.String instead of a char*. The **MarshalAsAttribute** attribute is used to indicate how the managed string should be marshaled when TakesAString is called.

The managed module is compiled with /clr, but /clr:pure works as well.

```
// TraditionalDll2.cpp
// compile with: /LD /EHsc
#include <windows.h>
#include <stdio.h>
#include <iostream>

using namespace std;

#define TRADITIONALDLL_EXPORTS
#ifdef TRADITIONALDLL_EXPORTS
#define TRADITIONALDLL_API __declspec(dllexport)
#else
#define TRADITIONALDLL_API __declspec(dllimport)
#endif

extern "C" {
   TRADITIONALDLL_API void TakesAString(char*);
}

void TakesAString(char* p) {
   printf_s("[unmanaged] %s\n", p);
}
```

```
// MarshalString.cpp
// compile with: /clr
using namespace System;
using namespace System::Runtime::InteropServices;

value struct TraditionalDLL
{
   [DllImport("TraditionalDLL2.dll")]
      static public void
      TakesAString([MarshalAs(UnmanagedType::LPStr)]String^);
};

int main() {
   String^ s = gcnew String("sample string");
```

```
     Console::WriteLine("[managed] passing managed string to unmanaged function...");
    TraditionalDLL::TakesAString(s);
    Console::WriteLine("[managed] {0}", s);
  }
```

This technique causes a copy of the string to be constructed on the unmanaged heap, so changes made to the string by the native function will not be reflected in the managed copy of the string. For passing strings by-reference, See How to: Modify System::String in Native Function Using PInvoke.

Note that no portion of the DLL is exposed to the managed code via the traditional #include directive. In fact, the DLL is accessed at runtime only, so problems with functions imported with **DllImport** will not be detected at compile time.

See Also

Other Resources

Using Explicit PInvoke in C++ (DllImport Attribute)

How to: Marshal Structures Using PInvoke

This topic explains how native functions that accept C-style strings can be called from managed functions providing an instance of String using P/Invoke. Visual C++ programmers are encouraged to use the C++ Interop features instead because P/Invoke provides little compile-time error reporting, is not type-safe, and can be tedious to implement. If the unmanaged API is packaged as a DLL and the source code is not available, P/Invoke is the only option. Otherwise, see the following topics:

- Using C++ Interop (Implicit PInvoke)
- How to: Marshal Structures Using PInvoke

By default, native and managed structures are laid out differently in memory, so successfully passing structures across the managed/unmanaged boundary requires extra steps to preserve data integrity.

This topic explains the steps required to define managed equivalents of native structures and how the resulting structures can be passed to unmanaged functions. This topic assumes that simple structures — those that do not contain strings or pointers — are used. For information on non-blittable interoperability, see Using C++ Interop (Implicit PInvoke).

Marshaling simple, blittable structures across the managed/unmanaged boundary first requires that managed versions of each native structure be defined. These structures can have any legal name; there is no relationship between the native and managed version of the two structures other than their data layout. Therefore, it is vital that the managed version contains fields that are the same size and in the same order as the native version. (There is no mechanism for ensuring that the managed and native versions of the structure are equivalent, so incompatibilities will not become apparent until run time. It is the programmer's responsibility to ensure that the two structures have the same data layout.)

Because the members of managed structures are sometimes rearranged for performance purposes, it is necessary to use the StructLayoutAttribute attribute to indicate that the structure are laid out sequentially. It is also a good idea to explicitly set the structure packing setting to be the same as that used by the native structure. (Although by default, Visual C++ uses an 8-byte structure packing for both managed code.)

1. Next, use DllImportAttribute to declare entry points that correspond to any unmanaged functions that accept the structure, but use the managed version of the structure in the function signatures, which is a moot point if you use the same name for both versions of the structure.
2. Now managed code can pass the managed version of the structure to the unmanaged functions as though they are actually managed functions. These structures can be passed either by value or by reference, as demonstrated in the following example.

Example

The following code consists of an unmanaged and a managed module. The unmanaged module is a DLL that defines a structure called Location and a function called GetDistance that accepts two instances of the Location structure. The second module is a managed command-line application that imports the GetDistance function, but defines it in terms of a managed equivalent of the Location structure, MLocation. In practice the same name would probably be used for both versions of the structure; however, a different name is used here to demonstrate that the DllImport prototype is defined in terms of the managed version.

The managed module is compiled with /clr, but /clr:pure works as well.

Note that no portion of the DLL is exposed to the managed code using the traditional #include directive. In fact, the DLL is accessed at run time only, so problems with functions imported with DllImport will not be detected at compile time.

```
// TraditionalDll3.cpp
// compile with: /LD /EHsc
#include <iostream>
#include <stdio.h>
#include <math.h>

#define TRADITIONALDLL_EXPORTS
#ifdef TRADITIONALDLL_EXPORTS
   #define TRADITIONALDLL_API __declspec(dllexport)
#else
   #define TRADITIONALDLL_API __declspec(dllimport)
#endif
```

```
#pragma pack(push, 8)
struct Location {
   int x;
   int y;
};
#pragma pack(pop)

extern "C" {
   TRADITIONALDLL_API double GetDistance(Location, Location);
   TRADITIONALDLL_API void InitLocation(Location*);
}

double GetDistance(Location loc1, Location loc2) {
   printf_s("[unmanaged] loc1(%d,%d)", loc1.x, loc1.y);
   printf_s(" loc2(%d,%d)\n", loc2.x, loc2.y);

   double h = loc1.x - loc2.x;
   double v = loc1.y = loc2.y;
   double dist = sqrt( pow(h,2) + pow(v,2) );

   return dist;
}

void InitLocation(Location* lp) {
   printf_s("[unmanaged] Initializing location...\n");
   lp->x = 50;
   lp->y = 50;
}
```

```
// MarshalStruct_pi.cpp
// compile with: /clr
using namespace System;
using namespace System::Runtime::InteropServices;

[StructLayout(LayoutKind::Sequential, Pack=8)]
value struct MLocation {
   int x;
   int y;
};

value struct TraditionalDLL {
   [DllImport("TraditionalDLL3.dll")]
   static public double GetDistance(MLocation, MLocation);
   [DllImport("TraditionalDLL3.dll")]
   static public double InitLocation(MLocation*);
};

int main() {
   MLocation loc1;
   loc1.x = 0;
   loc1.y = 0;

   MLocation loc2;
   loc2.x = 100;
   loc2.y = 100;

   double dist = TraditionalDLL::GetDistance(loc1, loc2);
   Console::WriteLine("[managed] distance = {0}", dist);

   MLocation loc3;
   TraditionalDLL::InitLocation(&loc3);
   Console::WriteLine("[managed] x={0} y={1}", loc3.x, loc3.y);
}
```

Sample Output

```
[unmanaged] loc1(0,0) loc2(100,100)
```

```
[managed] distance = 141.42135623731
[unmanaged] Initializing location...
[managed] x=50 y=50
```

See Also

Other Resources

Using Explicit PInvoke in C++ (DllImport Attribute)

How to: Marshal Arrays Using PInvoke

This topic explains how native functions that accept C-style strings can be called using the CLR string type String using .NET Framework Platform Invoke support. Visual C++ programmers are encouraged to use the C++ Interop features instead (when possible) because P/Invoke provides little compile-time error reporting, is not type-safe, and can be tedious to implement. If the unmanaged API is packaged as a DLL and the source code is not available, P/Invoke is the only option (otherwise, see Using C++ Interop (Implicit PInvoke)).

Example

Because native and managed arrays are laid out differently in memory, passing them successfully across the managed/unmanaged boundary requires conversion, or marshaling. This topic demonstrates how an array of simple (blitable) items can be passed to native functions from managed code.

As is true of managed/unmanaged data marshaling in general, the DllImportAttribute attribute is used to create a managed entry point for each native function that will be used. In the case of functions that take arrays as arguments, the MarshalAsAttribute attribute must be used as well to specify to the compiler how the data will be marshaled. In the following example, the LPArray enumeration is used to indicate that the managed array will be marshaled as a C-style array.

The following code consists of an unmanaged and a managed module. The unmanaged module is a DLL that defines a function that accepts an array of integers. The second module is a managed command-line application that imports this function, but defines it in terms of a managed array, and uses the **MarshalAsAttribute** attribute to specify that the array should be converted to a native array when called.

The managed module is compiled with /clr, but /clr:pure works as well.

```
// TraditionalDll4.cpp
// compile with: /LD /EHsc
#include <iostream>

#define TRADITIONALDLL_EXPORTS
#ifdef TRADITIONALDLL_EXPORTS
#define TRADITIONALDLL_API __declspec(dllexport)
#else
#define TRADITIONALDLL_API __declspec(dllimport)
#endif

extern "C" {
   TRADITIONALDLL_API void TakesAnArray(int len, int[]);
}

void TakesAnArray(int len, int a[]) {
   printf_s("[unmanaged]\n");
   for (int i=0; i<len; i++)
      printf("%d = %d\n", i, a[i]);
}
```

```
// MarshalBlitArray.cpp
// compile with: /clr
using namespace System;
using namespace System::Runtime::InteropServices;

value struct TraditionalDLL {
   [DllImport("TraditionalDLL4.dll")]
   static public void TakesAnArray(
   int len,[MarshalAs(UnmanagedType::LPArray)]array<int>^);
};

int main() {
   array<int>^ b = gcnew array<int>(3);
   b[0] = 11;
   b[1] = 33;
   b[2] = 55;
   TraditionalDLL::TakesAnArray(3, b);
```

```
    Console::WriteLine("[managed]");
    for (int i=0; i<3; i++)
       Console::WriteLine("{0} = {1}", i, b[i]);
 }
```

Note that no portion of the DLL is exposed to the managed code through the traditional #include directive. In fact, because the DLL is accessed at run time only, problems with functions imported with **DllImportAttribute** will not be detected at compile time.

See Also

Other Resources

Using Explicit PInvoke in C++ (DllImport Attribute)

How to: Marshal Function Pointers Using PInvoke

This topic explains how managed delegates can be used in place of function pointers when interoperating with unmanaged functions using .NET Framework P/Invoke features. However, Visual C++ programmers are encouraged to use the C++ Interop features instead (when possible) because P/Invoke provides little compile-time error reporting, is not type-safe, and can be tedious to implement. If the unmanaged API is packaged as a DLL and the source code is not available, P/Invoke is the only option. Otherwise, see the following topics:

- Using C++ Interop (Implicit PInvoke)
- How to: Marshal Callbacks and Delegates Using C++ Interop

Unmanaged APIs that take functions pointers as arguments can be called from managed code with a managed delegate in place of the native function pointer. The compiler automatically marshals the delegate to unmanaged functions as a function pointer and inserts the necessary managed/unmanaged transition code.

Example

The following code consists of an unmanaged and a managed module. The unmanaged module is a DLL that defines a function called TakesCallback that accepts a function pointer. This address is used to execute the function.

The managed module defines a delegate that is marshaled to the native code as a function pointer and uses the DllImportAttribute attribute to expose the native TakesCallback function to the managed code. In the main function, an instance of the delegate is created and passed to the TakesCallback function. The program output demonstrates that this function gets executed by the native TakesCallback function.

The managed function suppresses garbage collection for the managed delegate to prevent .NET Framework garbage collection from relocating the delegate while the native function executes.

The managed module is compiled with /clr, but /clr:pure works as well.

```
// TraditionalDll5.cpp
// compile with: /LD /EHsc
#include <iostream>
#define TRADITIONALDLL_EXPORTS
#ifdef TRADITIONALDLL_EXPORTS
#define TRADITIONALDLL_API __declspec(dllexport)
#else
#define TRADITIONALDLL_API __declspec(dllimport)
#endif

extern "C" {
   /* Declare an unmanaged function type that takes two int arguments
      Note the use of __stdcall for compatibility with managed code */
   typedef int (__stdcall *CALLBACK)(int);
   TRADITIONALDLL_API int TakesCallback(CALLBACK fp, int);
}

int TakesCallback(CALLBACK fp, int n) {
   printf_s("[unmanaged] got callback address, calling it...\n");
   return fp(n);
}
```

```
// MarshalDelegate.cpp
// compile with: /clr
using namespace System;
using namespace System::Runtime::InteropServices;

public delegate int GetTheAnswerDelegate(int);
public value struct TraditionalDLL {
   [DllImport("TraditionalDLL5.dll")]
   static public int TakesCallback(GetTheAnswerDelegate^ pfn, int n);
};
```

```
int GetNumber(int n) {
   Console::WriteLine("[managed] callback!");
   static int x = 0;
   ++x;
   return x + n;
}

int main() {
   GetTheAnswerDelegate^ fp = gcnew GetTheAnswerDelegate(GetNumber);
   pin_ptr<GetTheAnswerDelegate^> pp = &fp;
   Console::WriteLine("[managed] sending delegate as callback...");

   int answer = TraditionalDLL::TakesCallback(fp, 42);
}
```

Note that no portion of the DLL is exposed to the managed code using the traditional #include directive. In fact, the DLL is accessed at run time only, so problems with functions imported with **DllImportAttribute** will not be detected at compile time.

See Also

Other Resources

Using Explicit PInvoke in C++ (DllImport Attribute)

How to: Marshal Embedded Pointers Using PInvoke

Functions that are implemented in unmanaged DLLs can be called from managed code using Platform Invoke (P/Invoke) functionality. If the source code for the DLL is not available, P/Invoke is the only option for interoperating. However, unlike other .NET languages, Visual C++ provides an alternative to P/Invoke. For more information, see Using C++ Interop (Implicit PInvoke) and How to: Marshal Embedded Pointers Using C++ Interop.

Example

Passing structures to native code requires that a managed structure that is equivalent in terms of data layout to the native structure is created. However, structures that contain pointers require special handling. For each embedded pointer in the native structure, the managed version of the structure should contain an instance of the IntPtr type. Also, memory for these instances must be explicitly allocated, initialized, and released using the AllocCoTaskMem, StructureToPtr, and FreeCoTaskMem methods.

The following code consists of an unmanaged and a managed module. The unmanaged module is a DLL that defines a function that accepts a structure called ListString that contains a pointer, and a function called TakesListStruct. The managed module is a command-line application that imports the TakesListStruct function and defines a structure called MListStruct that is equivalent to the native ListStruct except that the double* is represented with an **IntPtr** instance. Before calling TakesListStruct, the main function allocates and initializes the memory that this field references.

The managed module is compiled with /clr, but /clr:pure works as well.

```
// TraditionalDll6.cpp
// compile with: /EHsc /LD
#include <stdio.h>
#include <iostream>
#define TRADITIONALDLL_EXPORTS
#ifdef TRADITIONALDLL_EXPORTS
#define TRADITIONALDLL_API __declspec(dllexport)
#else
#define TRADITIONALDLL_API __declspec(dllimport)
#endif

#pragma pack(push, 8)
struct ListStruct {
   int count;
   double* item;
};
#pragma pack(pop)

extern "C" {
   TRADITIONALDLL_API void TakesListStruct(ListStruct);
}

void TakesListStruct(ListStruct list) {
   printf_s("[unmanaged] count = %d\n", list.count);
   for (int i=0; i<list.count; i++)
      printf_s("array[%d] = %f\n", i, list.item[i]);
}
```

```
// EmbeddedPointerMarshalling.cpp
// compile with: /clr
using namespace System;
using namespace System::Runtime::InteropServices;

[StructLayout(LayoutKind::Sequential, Pack=8)]
value struct MListStruct {
   int count;
   IntPtr item;
};

value struct TraditionalDLL {
    [DllImport("TraditionalDLL6.dll")]
```

```
   static public void TakesListStruct(MListStruct);
};

int main() {
   array<double>^ parray = gcnew array<double>(10);
   Console::WriteLine("[managed] count = {0}", parray->Length);

   Random^ r = gcnew Random();
   for (int i=0; i<parray->Length; i++) {
      parray[i] = r->NextDouble() * 100.0;
      Console::WriteLine("array[{0}] = {1}", i, parray[i]);
   }

   int size = Marshal::SizeOf(double::typeid);
   MListStruct list;
   list.count = parray->Length;
   list.item = Marshal::AllocCoTaskMem(size * parray->Length);

   for (int i=0; i<parray->Length; i++) {
      IntPtr t = IntPtr(list.item.ToInt32() + i * size);
      Marshal::StructureToPtr(parray[i], t, false);
   }

   TraditionalDLL::TakesListStruct( list );
   Marshal::FreeCoTaskMem(list.item);
}
```

Note that no portion of the DLL is exposed to the managed code using the traditional #include directive. In fact, the DLL is accessed at run time only, so problems with functions imported with DllImportAttribute will not be detected at compile time.

See Also

Other Resources

Using Explicit PInvoke in C++ (DllImport Attribute)

Using C++ Interop (Implicit PInvoke)

Unlike other .NET languages, Visual C++ has interoperability support that allows managed and unmanaged code to exist in the same application and even in the same file (with the managed, unmanaged pragmas). This allows Visual C++ developers to integrate .NET functionality into existing Visual C++ applications without disturbing the rest of the application.

You can also call unmanaged functions from a managed compiland using dllexport, dllimport.

Implicit PInvoke is useful when you do not need to specify how function parameters will be marshaled, or any of the other details that can be specified when explicitly calling DllImportAttribute.

Visual C++ provides two ways for managed and unmanaged functions to interoperate:

- Using Explicit PInvoke in C++ (DllImport Attribute)

Explicit PInvoke is supported by the .NET Framework and is available in most .NET languages. But as its name implies, C++ Interop is specific to Visual C++.

C++ Interop

C++ Interop is recommended over explicit PInvoke because it provides better type safety, is typically less tedious to implement, is more forgiving if the unmanaged API is modified, and makes performance enhancements possible that are not possible with explicit PInvoke. However, C++ Interop is not possible if the unmanaged source code is not available or when compiling with **/clr:safe** (see Pure and Verifiable Code for more information).

C++ COM Interop

The interoperability features supported by Visual C++ offer a particular advantage over other .NET languages when it comes to interoperating with COM components. Instead of being limited to the restrictions of the .NET Framework Type Library Importer (Tlbimp.exe), such as limited support for data types and the mandatory exposure of every member of every COM interface, C++ Interop allows COM components to be accessed at will and does not require separate interop assemblies. For more information, see Using Native COM Servers from .NET.

Blittable Types

For unmanaged APIs that use simple, intrinsic types (see Blittable and Non-Blittable Types), no special coding is required because these data types have the same representation in memory, but more complex data types require explicit data marshaling. For an example, see How to: Call Native DLLs from Managed Code Using PInvoke.

Example

```
// vcmcppv2_impl_dllimp.cpp
// compile with: /clr:pure user32.lib
using namespace System::Runtime::InteropServices;

// Implicit DLLImport specifying calling convention
extern "C" int __stdcall MessageBeep(int);

// explicit DLLImport needed here to use P/Invoke marshalling because
// System::String ^ is not the type of the first parameter to printf
[DllImport("msvcrt.dll", EntryPoint = "printf", CallingConvention = CallingConvention::Cdec
l,  CharSet = CharSet::Ansi)]
// or just
// [DllImport("msvcrt.dll")]
int printf(System::String ^, ...);

int main() {
   // (string literals are System::String by default)
   printf("Begin beep\n");
   MessageBeep(100000);
   printf("Done\n");
}
```

Output

```
Begin beep
Done
```

In This Section

- How to: Marshal ANSI Strings Using C++ Interop
- How to: Marshal Unicode Strings Using C++ Interop
- How to: Marshal COM Strings Using C++ Interop
- How to: Marshal Structures Using C++ Interop
- How to: Marshal Arrays Using C++ Interop
- How to: Marshal Callbacks and Delegates Using C++ Interop
- How to: Marshal Embedded Pointers Using C++ Interop
- How to: Access Characters in a System::String
- How to: Convert char * String to System::Byte Array
- How to: Convert System::String to wchar_t* or char*
- How to: Convert System::String to Standard String
- How to: Convert Standard String to System::String
- How to: Modify System::String in Native Function Using PInvoke
- How to: Obtain a Pointer to Byte Array
- How to: Load Unmanaged Resources into a Byte Array
- How to: Modify Reference Class in a Native Function
- How to: Determine if an Image is Native or CLR
- How to: Add Native DLL to Global Assembly Cache
- How to: Hold Reference to Value Type in Native Type
- How to: Hold Object Reference in Native Function
- How to: Detect /clr Compilation
- How to: Convert Between System::Guid and _GUID
- How to: Specify an out Parameter
- How to: Use a Native Type in a /clr Compilation
- How to: Declare Handles in Native Types
- How to: Wrap Native Class for Use by C#

For information on using delegates in an interop scenario, see delegate.

See Also

Concepts

Calling Native Functions from Managed Code

How to: Marshal ANSI Strings Using C++ Interop

This topic demonstrates how ANSI strings can be passed using C++ Interop, but the .NET Framework String represents strings in Unicode format, so conversion to ANSI is an extra step. For interoperating with other string types, see the following topics:

- How to: Marshal Unicode Strings Using C++ Interop
- How to: Marshal COM Strings Using C++ Interop

The following code examples use the managed, unmanaged #pragma directives to implement managed and unmanaged functions in the same file, but these functions interoperate in the same manner if defined in separate files. Because files containing only unmanaged functions do not need to be compiled with /clr (Common Language Runtime Compilation), they can retain their performance characteristics.

Example

The example demonstrates passing an ANSI string from a managed to an unmanaged function using StringToHGlobalAnsi. This method allocates memory on the unmanaged heap and returns the address after performing the conversion. This means that no pinning is necessary (because memory on the GC heap is not being passed to the unmanaged function) and that the IntPtr returned from **StringToHGlobalAnsi** must be explicitly released or a memory leak results.

```
// MarshalANSI1.cpp
// compile with: /clr
#include <iostream>
#include <stdio.h>

using namespace std;
using namespace System;
using namespace System::Runtime::InteropServices;

#pragma unmanaged

void NativeTakesAString(const char* p) {
   printf_s("(native) received '%s'\n", p);
}

#pragma managed

int main() {
   String^ s = gcnew String("sample string");
   IntPtr ip = Marshal::StringToHGlobalAnsi(s);
   const char* str = static_cast<const char*>(ip.ToPointer());

   Console::WriteLine("(managed) passing string...");
   NativeTakesAString( str );

   Marshal::FreeHGlobal( ip );
}
```

The following example demonstrates the data marshaling required to access an ANSI string in a managed function that is called by an unmanaged function. The managed function, on receiving the native string, can either use it directly or convert it to a managed string using the PtrToStringAnsi method, as shown.

```
// MarshalANSI2.cpp
// compile with: /clr
#include <iostream>
#include <vcclr.h>

using namespace std;

using namespace System;
using namespace System::Runtime::InteropServices;

#pragma managed
```

```
void ManagedStringFunc(char* s) {
   String^ ms = Marshal::PtrToStringAnsi(static_cast<IntPtr>(s));
   Console::WriteLine("(managed): received '{0}'", ms);
}

#pragma unmanaged

void NativeProvidesAString() {
   cout << "(native) calling managed func...\n";
   ManagedStringFunc("test string");
}

#pragma managed

int main() {
   NativeProvidesAString();
}
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Marshal Unicode Strings Using C++ Interop

This topic demonstrates one facet of Visual C++ interoperability. For more information, see Using C++ Interop (Implicit PInvoke).

The following code examples use the managed, unmanaged #pragma directives to implement managed and unmanaged functions in the same file, but these functions interoperate in the same manner if defined in separate files. Files containing only unmanaged functions do not need to be compiled with /clr (Common Language Runtime Compilation).

This topic demonstrates how Unicode strings can be passed from a managed to an unmanaged function, and vice versa. For interoperating with other strings types, see the following topics:

- How to: Marshal ANSI Strings Using C++ Interop
- How to: Marshal COM Strings Using C++ Interop

Example

To pass a Unicode string from a managed to an unmanaged function, the PtrToStringChars function (declared in Vcclr.h) can be used to access in the memory where the managed string is stored. Because this address will be passed to a native function, it is important that the memory be pinned with pin_ptr to prevent the string data from being relocated, should a garbage collection cycle take place while the unmanaged function executes.

```
// MarshalUnicode1.cpp
// compile with: /clr
#include <iostream>
#include <stdio.h>
#include <vcclr.h>

using namespace std;

using namespace System;
using namespace System::Runtime::InteropServices;

#pragma unmanaged

void NativeTakesAString(const wchar_t* p) {
   printf_s("(native) recieved '%S'\n", p);
}

#pragma managed

int main() {
   String^ s = gcnew String("test string");
   pin_ptr<const wchar_t> str = PtrToStringChars(s);

   Console::WriteLine("(managed) passing string to native func...");
   NativeTakesAString( str );
}
```

The following example demonstrates the data marshaling required to access a Unicode string in a managed function called by an unmanaged function. The managed function, on receiving the native Unicode string, converts it to a managed string using the PtrToStringUni method.

```
// MarshalUnicode2.cpp
// compile with: /clr
#include <iostream>

using namespace std;
using namespace System;
using namespace System::Runtime::InteropServices;

#pragma managed

void ManagedStringFunc(wchar_t* s) {
```

```
   String^ ms = Marshal::PtrToStringUni((IntPtr)s);
   Console::WriteLine("(managed) recieved '{0}'", ms);
}

#pragma unmanaged

void NativeProvidesAString() {
   cout << "(unmanaged) calling managed func...\n";
   ManagedStringFunc(L"test string");
}

#pragma managed

int main() {
   NativeProvidesAString();
}
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Marshal COM Strings Using C++ Interop

This topic demonstrates how a BSTR (the basic string format favored in COM programming) can be passed from a managed to an unmanaged function, and vice versa. For interoperating with other strings types, see the following topics:

- How to: Marshal Unicode Strings Using C++ Interop
- How to: Marshal ANSI Strings Using C++ Interop

The following code examples use the managed, unmanaged #pragma directives to implement managed and unmanaged functions in the same file, but these functions interoperate in the same manner if defined in separate files. Files containing only unmanaged functions do not need to be compiled with /clr (Common Language Runtime Compilation).

Example

The following example demonstrates how a BSTR (a string format used in COM programming) can be passed from a managed to an unmanaged function. The calling managed function uses StringToBSTR to obtain the address of a BSTR representation of the contents of a .NET System.String. This pointer is pinned using pin_ptr to ensure that its physical address is not changed during a garbage collection cycle while the unmanaged function executes. The garbage collector is prohibited from moving the memory until the pin_ptr goes out of scope.

```
// MarshalBSTR1.cpp
// compile with: /clr
#define WINVER 0x0502
#define _AFXDLL
#include <afxwin.h>

#include <iostream>
using namespace std;

using namespace System;
using namespace System::Runtime::InteropServices;

#pragma unmanaged

void NativeTakesAString(BSTR bstr) {
   printf_s("%S", bstr);
}

#pragma managed

int main() {
   String^ s = "test string";

   IntPtr ip = Marshal::StringToBSTR(s);
   BSTR bs = static_cast<BSTR>(ip.ToPointer());
   pin_ptr<BSTR> b = &bs;

   NativeTakesAString( bs );
   Marshal::FreeBSTR(ip);
}
```

The following example demonstrates how a BSTR can be passed from an unmanaged to an unmanaged function. The receiving managed function can either use the string in as a BSTR or use PtrToStringBSTR to convert it to a String for use with other managed functions. Because the memory representing the BSTR is allocated on the unmanaged heap, no pinning is necessary, because there is no garbage collection on the unmanaged heap.

```
// MarshalBSTR2.cpp
// compile with: /clr
#define WINVER 0x0502
#define _AFXDLL
#include <afxwin.h>

#include <iostream>
```

```
using namespace std;

using namespace System;
using namespace System::Runtime::InteropServices;

#pragma managed

void ManagedTakesAString(BSTR bstr) {
   String^ s = Marshal::PtrToStringBSTR(static_cast<IntPtr>(bstr));
   Console::WriteLine("(managed) convered BSTR to String: '{0}'", s);
}

#pragma unmanaged

void UnManagedFunc() {
   BSTR bs = SysAllocString(L"test string");
   printf_s("(unmanaged) passing BSTR to managed func...\n");
   ManagedTakesAString(bs);
}

#pragma managed

int main() {
   UnManagedFunc();
}
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Marshal Structures Using C++ Interop

This topic demonstrates one facet of Visual C++ interoperability. For more information, see Using C++ Interop (Implicit PInvoke).

The following code examples use the managed, unmanaged #pragma directives to implement managed and unmanaged functions in the same file, but these functions interoperate in the same manner if defined in separate files. Files containing only unmanaged functions do not need to be compiled with /clr (Common Language Runtime Compilation).

Example

The following example demonstrates passing a structure from a managed to an unmanaged function, both by value and by reference. Because the structure in this example contains only simple, intrinsic data types (see Blittable and Non-Blittable Types), no special marshaling is required. To marshal non-blittable structures, such as those that contain pointers, see How to: Marshal Embedded Pointers Using C++ Interop.

```
// PassStruct1.cpp
// compile with: /clr

#include <stdio.h>
#include <math.h>

using namespace System;
using namespace System::Runtime::InteropServices;

#pragma unmanaged

struct Location {
   int x;
   int y;
};

double GetDistance(Location loc1, Location loc2) {
   printf_s("[unmanaged] loc1(%d,%d)", loc1.x, loc1.y);
   printf_s(" loc2(%d,%d)\n", loc2.x, loc2.y);

   double h = loc1.x - loc2.x;
   double v = loc1.y = loc2.y;
   double dist = sqrt( pow(h,2) + pow(v,2) );

   return dist;
}

void InitLocation(Location* lp) {
   printf_s("[unmanaged] Initializing location...\n");
   lp->x = 50;
   lp->y = 50;
}

#pragma managed

int main() {
   Location loc1;
   loc1.x = 0;
   loc1.y = 0;

   Location loc2;
   loc2.x = 100;
   loc2.y = 100;

   double dist = GetDistance(loc1, loc2);
   Console::WriteLine("[managed] distance = {0}", dist);

   Location loc3;
   InitLocation(&loc3);
   Console::WriteLine("[managed] x={0} y={1}", loc3.x, loc3.y);
```

```
}
```

The following example demonstrates passing a structure from an unmanaged to a managed function, both by value and by reference. Because the structure in this example contains only simple, intrinsic data types (see Blittable and Non-Blittable Types), no special marshalling is required. To marshal non-blittable structures, such as those that contain pointers, see How to: Marshal Embedded Pointers Using C++ Interop.

```
// PassStruct2.cpp
// compile with: /clr
#include <stdio.h>
#include <math.h>
using namespace System;

// native structure definition
struct Location {
   int x;
   int y;
};

#pragma managed

double GetDistance(Location loc1, Location loc2) {
   Console::Write("[managed] got loc1({0},{1})", loc1.x, loc1.y);
   Console::WriteLine(" loc2({0},{1})", loc2.x, loc2.y);

   double h = loc1.x - loc2.x;
   double v = loc1.y = loc2.y;
   double dist = sqrt( pow(h,2) + pow(v,2) );

   return dist;
}

void InitLocation(Location* lp) {
   Console::WriteLine("[managed] Initializing location...");
   lp->x = 50;
   lp->y = 50;
}

#pragma unmanaged

int UnmanagedFunc() {
   Location loc1;
   loc1.x = 0;
   loc1.y = 0;

   Location loc2;
   loc2.x = 100;
   loc2.y = 100;

   printf_s("(unmanaged) loc1=(%d,%d)", loc1.x, loc1.y);
   printf_s(" loc2=(%d,%d)\n", loc2.x, loc2.y);

   double dist = GetDistance(loc1, loc2);
   printf_s("[unmanaged] distance = %f\n", dist);

   Location loc3;
   InitLocation(&loc3);
   printf_s("[unmanaged] got x=%d y=%d\n", loc3.x, loc3.y);

    return 0;
}

#pragma managed

int main() {
   UnmanagedFunc();
}
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Marshal Arrays Using C++ Interop

This topic demonstrates one facet of Visual C++ interoperability. For more information, see Using C++ Interop (Implicit PInvoke).

The following code examples use the managed, unmanaged #pragma directives to implement managed and unmanaged functions in the same file, but these functions interoperate in the same manner if defined in separate files. Files containing only unmanaged functions do not need to be compiled with /clr (Common Language Runtime Compilation).

Example

The following example demonstrates how to pass a managed array to an unmanaged function. The managed function uses pin_ptr to suppress garbage collection for the array before calling the unmanaged function. By providing the unmanaged function with a pinned pointer into the GC heap, the overhead of making a copy of the array can be avoided. To demonstrate that the unmanaged function is accessing GC heap memory, it modifies the contents of the array and the changes are reflected when the managed function resumes control.

```
// PassArray1.cpp
// compile with: /clr
#include <iostream>
using namespace std;

using namespace System;

#pragma unmanaged

void TakesAnArray(int* a, int c) {
   cout << "(unmanaged) array recieved:\n";
   for (int i=0; i<c; i++)
      cout << "a[" << i << "] = " << a[i] << "\n";

   cout << "(unmanaged) modifying array contents...\n";
   for (int i=0; i<c; i++)
      a[i] = rand() % 100;
}

#pragma managed

int main() {
   array<int>^ nums = gcnew array<int>(5);

   nums[0] = 0;
   nums[1] = 1;
   nums[2] = 2;
   nums[3] = 3;
   nums[4] = 4;

   Console::WriteLine("(managed) array created:");
   for (int i=0; i<5; i++)
      Console::WriteLine("a[{0}] = {1}", i, nums[i]);

   pin_ptr<int> pp = &nums[0];
   TakesAnArray(pp, 5);

   Console::WriteLine("(managed) contents:");
   for (int i=0; i<5; i++)
      Console::WriteLine("a[{0}] = {1}", i, nums[i]);
}
```

The following example demonstrates passing an unmanaged array to a managed function. The managed function accesses the array memory directly (as opposed to creating a managed array and copying the array content), which allows changes made by the managed function to be reflected in the unmanaged function when it regains control.

```
// PassArray2.cpp
```

```
// compile with: /clr
#include <iostream>
using namespace std;

using namespace System;

#pragma managed

void ManagedTakesAnArray(int* a, int c) {
   Console::WriteLine("(managed) array recieved:");
   for (int i=0; i<c; i++)
      Console::WriteLine("a[{0}] = {1}", i, a[i]);

   cout << "(managed) modifying array contents...\n";
   Random^ r = gcnew Random(DateTime::Now.Second);
   for (int i=0; i<c; i++)
      a[i] = r->Next(100);
}

#pragma unmanaged

void NativeFunc() {
   int nums[5] = { 0, 1, 2, 3, 4 };

   printf_s("(unmanaged) array created:\n");
   for (int i=0; i<5; i++)
      printf_s("a[%d] = %d\n", i, nums[i]);

   ManagedTakesAnArray(nums, 5);

   printf_s("(ummanaged) contents:\n");
   for (int i=0; i<5; i++)
      printf_s("a[%d] = %d\n", i, nums[i]);
}

#pragma managed

int main() {
   NativeFunc();
}
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Marshal Callbacks and Delegates Using C++ Interop

This topic demonstrates the marshalling of callbacks and delegates (the managed version of a callback) between managed and unmanaged code using Visual C++.

The following code examples use the managed, unmanaged #pragma directives to implement managed and unmanaged functions in the same file, but the functions could also be defined in separate files. Files containing only unmanaged functions do not need to be compiled with the /clr (Common Language Runtime Compilation).

Example

The following example demonstrates how to configure an unmanaged API to trigger a managed delegate. A managed delegate is created and one of the interop methods, GetFunctionPointerForDelegate, is used to retrieve the underlying entry point for the delegate. This address is then passed to the unmanaged function, which calls it with no knowledge of the fact that it is implemented as a managed function.

Notice that is it possible, but not necessary, to pin the delegate using pin_ptr to prevent it from being re-located or disposed of by the garbage collector. Protection from premature garbage collection is needed, but pinning provides more protection than is necessary, as it prevents collection but also prevents relocation.

If a delegate is re-located by a garbage collection, it will not affect the underlaying managed callback, so Alloc is used to add a reference to the delegate, allowing relocation of the delegate, but preventing disposal. Using GCHandle instead of pin_ptr reduces fragmentation potential of the managed heap.

```
// MarshalDelegate1.cpp
// compile with: /clr
#include <iostream>

using namespace System;
using namespace System::Runtime::InteropServices;

#pragma unmanaged

// Declare an unmanaged function type that takes two int arguments
// Note the use of __stdcall for compatibility with managed code
typedef int (__stdcall *ANSWERCB)(int, int);

int TakesCallback(ANSWERCB fp, int n, int m) {
   printf_s("[unmanaged] got callback address, calling it...\n");
   return fp(n, m);
}

#pragma managed

public delegate int GetTheAnswerDelegate(int, int);

int GetNumber(int n, int m) {
   Console::WriteLine("[managed] callback!");
   return n + m;
}

int main() {
   GetTheAnswerDelegate^ fp = gcnew GetTheAnswerDelegate(GetNumber);
   GCHandle gch = GCHandle::Alloc(fp);
   IntPtr ip = Marshal::GetFunctionPointerForDelegate(fp);
   ANSWERCB cb = static_cast<ANSWERCB>(ip.ToPointer());
   Console::WriteLine("[managed] sending delegate as callback...");

// force garbage collection cycle to prove
// that the delegate doesn't get disposed
   GC::Collect();

   int answer = TakesCallback(cb, 243, 257);

// release reference to delegate
```

```
   gch.Free();
}
```

The following example is similar to the previous example, but in this case the provided function pointer is stored by the unmanaged API, so it can be invoked at any time, requiring that garbage collection be suppressed for an arbitrary length of time. As a result, the following example uses a global instance of GCHandle to prevent the delegate from being relocated, independent of function scope. As discussed in the first example, using pin_ptr is unnecessary for these examples, but in this case wouldn't work anyway, as the scope of a pin_ptr is limited to a single function.

```
// MarshalDelegate2.cpp
// compile with: /clr
#include <iostream>

using namespace System;
using namespace System::Runtime::InteropServices;

#pragma unmanaged

// Declare an unmanaged function type that takes two int arguments
// Note the use of __stdcall for compatibility with managed code
typedef int (__stdcall *ANSWERCB)(int, int);
static ANSWERCB cb;

int TakesCallback(ANSWERCB fp, int n, int m) {
   cb = fp;
   if (cb) {
      printf_s("[unmanaged] got callback address (%d), calling it...\n", cb);
      return cb(n, m);
   }
   printf_s("[unmanaged] unregistering callback");
   return 0;
}

#pragma managed

public delegate int GetTheAnswerDelegate(int, int);

int GetNumber(int n, int m) {
   Console::WriteLine("[managed] callback!");
   static int x = 0;
   ++x;

   return n + m + x;
}

static GCHandle gch;

int main() {
   GetTheAnswerDelegate^ fp = gcnew GetTheAnswerDelegate(GetNumber);

   gch = GCHandle::Alloc(fp);

   IntPtr ip = Marshal::GetFunctionPointerForDelegate(fp);
   ANSWERCB cb = static_cast<ANSWERCB>(ip.ToPointer());
   Console::WriteLine("[managed] sending delegate as callback...");

   int answer = TakesCallback(cb, 243, 257);

   // possibly much later (in another function)...

   Console::WriteLine("[managed] releasing callback mechanisms...");
   TakesCallback(0, 243, 257);
   gch.Free();
}
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Marshal Embedded Pointers Using C++ Interop

The following code examples use the managed, unmanaged #pragma directives to implement managed and unmanaged functions in the same file, but these functions interoperate in the same manner if defined in separate files. Files containing only unmanaged functions do not need to be compiled with /clr (Common Language Runtime Compilation).

Example

The following example demonstrates how an unmanaged function that takes a structure containing pointers can be called from a managed function. The managed function creates an instance of the structure and initializes the embedded pointer with the new keyword (instead of the gcnew keyword). Because this allocates the memory on the native heap, there is no need to pin the array to suppress garbage collection. However, the memory must be explicitly deleted to avoid memory leakage.

```
// marshal_embedded_pointer.cpp
// compile with: /clr
#include <iostream>

using namespace System;
using namespace System::Runtime::InteropServices;

// unmanaged struct
struct ListStruct {
   int count;
   double* item;
};

#pragma unmanaged

void UnmanagedTakesListStruct(ListStruct list) {
   printf_s("[unmanaged] count = %d\n", list.count);
   for (int i=0; i<list.count; i++)
      printf_s("array[%d] = %f\n", i, list.item[i]);
}

#pragma managed

int main() {
   ListStruct list;
   list.count = 10;
   list.item = new double[list.count];

   Console::WriteLine("[managed] count = {0}", list.count);
   Random^ r = gcnew Random(0);
   for (int i=0; i<list.count; i++) {
      list.item[i] = r->NextDouble() * 100.0;
      Console::WriteLine("array[{0}] = {1}", i, list.item[i]);
   }

   UnmanagedTakesListStruct( list );
   delete list.item;
}
```

Output

```
[managed] count = 10
array[0] = 72.624326996796
array[1] = 81.7325359590969
array[2] = 76.8022689394663
array[3] = 55.8161191436537
array[4] = 20.6033154021033
array[5] = 55.8884794618415
array[6] = 90.6027066011926
array[7] = 44.2177873310716
array[8] = 97.754975314138
array[9] = 27.370445768987
[unmanaged] count = 10
```

```
array[0] = 72.624327
array[1] = 81.732536
array[2] = 76.802269
array[3] = 55.816119
array[4] = 20.603315
array[5] = 55.888479
array[6] = 90.602707
array[7] = 44.217787
array[8] = 97.754975
array[9] = 27.370446
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Access Characters in a System::String

You can access characters of a String object for high-performance calls to unmanaged functions that take `wchar_t*` strings. The method yields an interior pointer to the first character of the **String** object. This pointer can be manipulated directly or pinned and passed to a function expecting an ordinary `wchar_t` string.

Example

PtrToStringChars returns a Char, which is an interior pointer (also known as a **byref**). As such, it is subject to garbage collection. You don't have to pin this pointer unless you're going to pass it to a native function.

Consider the following code. Pinning is not needed because `ppchar` is an interior pointer, and if the garbage collector moves the string it points to, it will also update `ppchar`. Without a pin_ptr, the code will work and not have the potential performance hit caused by pinning.

If you pass `ppchar` to a native function, then it must be a pinning pointer; the garbage collector will not be able to update any pointers on the unmanaged stack frame.

```
// PtrToStringChars.cpp
// compile with: /clr
#include<vcclr.h>
using namespace System;

int main() {
   String ^ mystring = "abcdefg";

   interior_ptr<const Char> ppchar = PtrToStringChars( mystring );

   for ( ; *ppchar != L'\0'; ++ppchar )
      Console::Write(*ppchar);
}
```

Output

```
abcdefg
```

This example shows where pinning is needed.

```
// PtrToStringChars_2.cpp
// compile with: /clr
#include <string.h>
#include <vcclr.h>
// using namespace System;

size_t getlen(System::String ^ s) {
   // make sure it doesn't move during the unmanaged call
   pin_ptr<const wchar_t> pinchars = PtrToStringChars(s);
   return wcslen(pinchars);
};

int main() {
   System::Console::WriteLine(getlen("testing"));
}
```

Output

```
7
```

An interior pointer has all the properties of a native C++ pointer. For example, you can use it to walk a linked data structure and do insertions and deletions using only one pointer:

```
// PtrToStringChars_3.cpp
// compile with: /clr /LD
using namespace System;
```

```
ref struct ListNode {
   Int32 elem;
   ListNode ^ Next;
};

void deleteNode( ListNode ^ list, Int32 e ) {
   interior_ptr<ListNode ^> ptrToNext = &list;
   while (*ptrToNext != nullptr) {
      if ( (*ptrToNext) -> elem == e )
         *ptrToNext = (*ptrToNext) -> Next;   // delete node
      else
         ptrToNext = &(*ptrToNext) -> Next;   // move to next node
   }
}
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Convert char * String to System::Byte Array

The most efficient way to convert a **char *** string to a Byte array is to use Marshal class.

Example

```
// convert_native_string_to_Byte_array.cpp
// compile with: /clr
#include <string.h>

using namespace System;
using namespace System::Runtime::InteropServices;

int main() {
   char buf[] = "Native String";
   int len = strlen(buf);

   array< Byte >^ byteArray = gcnew array< Byte >(len + 2);

   // convert native pointer to System::IntPtr with C-Style cast
   Marshal::Copy((IntPtr)buf,byteArray, 0, len);

   for ( int i = byteArray->GetLowerBound(0); i <= byteArray->GetUpperBound(0); i++ ) {
      char dc =  *(Byte^)   byteArray->GetValue(i);
      Console::Write((Char)dc);
   }

   Console::WriteLine();
}
```

```
Native String
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Convert System::String to wchar_t* or char*

You can use **PtrToStringChars** in Vcclr.h to convert String to native **wchar_t *** or **char ***. This always returns a wide Unicode string pointer because CLR strings are internally Unicode. You can then convert from wide as shown in the following example.

Example

```
// convert_string_to_wchar.cpp
// compile with: /clr
#include < stdio.h >
#include < stdlib.h >
#include < vcclr.h >

using namespace System;

int main() {
   String ^str = "Hello";

   // Pin memory so GC can't move it while native function is called
   pin_ptr<const wchar_t> wch = PtrToStringChars(str);
   printf_s("%S\n", wch);

   // Conversion to char* :
   // Can just convert wchar_t* to char* using one of the
   // conversion functions such as:
   // WideCharToMultiByte()
   // wcstombs_s()
   // ... etc
   size_t convertedChars = 0;
   size_t  sizeInBytes = ((str->Length + 1) * 2);
   errno_t err = 0;
   char    *ch = (char *)malloc(sizeInBytes);

   err = wcstombs_s(&convertedChars,
                    ch, sizeInBytes,
                    wch, sizeInBytes);
   if (err != 0)
      printf_s("wcstombs_s  failed!\n");

    printf_s("%s\n", ch);
}
```

Output

```
Hello
Hello
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Convert System::String to Standard String

You can convert a String to **std::string** or **std::wstring**, without using **PtrToStringChars** in Vcclr.h.

Example

```
// convert_system_string.cpp
// compile with: /clr
#include <string>
#include <iostream>
using namespace std;
using namespace System;

void MarshalString ( String ^ s, string& os ) {
   using namespace Runtime::InteropServices;
   const char* chars =
      (const char*)(Marshal::StringToHGlobalAnsi(s)).ToPointer();
   os = chars;
   Marshal::FreeHGlobal(IntPtr((void*)chars));
}

void MarshalString ( String ^ s, wstring& os ) {
   using namespace Runtime::InteropServices;
   const wchar_t* chars =
      (const wchar_t*)(Marshal::StringToHGlobalUni(s)).ToPointer();
   os = chars;
   Marshal::FreeHGlobal(IntPtr((void*)chars));
}

int main() {
   string a = "test";
   wstring b = L"test2";
   String ^ c = gcnew String("abcd");

   cout << a << endl;
   MarshalString(c, a);
   c = "efgh";
   MarshalString(c, b);
   cout << a << endl;
   wcout << b << endl;
}
```

```
test
abcd
efgh
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Convert Standard String to System::String

This topic shows how convert a Standard C++ Library string (<string>) to a String.

Example

```
// convert_standard_string_to_system_string.cpp
// compile with: /clr
#include <string>
#include <iostream>
using namespace System;
using namespace std;

int main() {
   string str = "test";
   cout << str << endl;
   String^ str2 = gcnew String(str.c_str());
   Console::WriteLine(str2);

   // alternatively
   String^ str3 = gcnew String(str.c_str());
   Console::WriteLine(str3);
}
```

Output

```
test
test
test
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Modify System::String in Native Function Using PInvoke

You can pass an array of String to a native function and modify the array in the native function using **PInvoke** services with DllImportAttribute and MarshalAsAttribute.

Example

Compile the native library.

```
// string_in_native_function.cpp
// compile with: /LD
#include <stdio.h>
#include <windows.h>

extern "C" {
   __declspec(dllexport) int f(wchar_t* buff[]) {
      printf_s("in f: %S\n", buff[0]);
      fflush(stdin);
      fflush(stdout);
      buff[0] = L"changed";
      return wcslen(buff[0]);
   }
};
```

Compile the following assembly that consumes the native DLL.

```
// string_in_native_function_2.cpp
// compile with: /clr
using namespace System;
using namespace System::Runtime::InteropServices;

[DllImport("string_in_native_function.dll", CharSet=CharSet::Unicode)]
int f([In][Out][MarshalAsAttribute(UnmanagedType::LPArray,
                ArraySubType=UnmanagedType::LPWStr, SizeConst=2)] array<String ^> ^ str);

int main() {
   array<String ^> ^ myarr = gcnew array<String^>(2);
   myarr[0] = "one";
   myarr[1] = "two";
   f(myarr);   // call to imported native function.
   Console::WriteLine("in managed: {0}", myarr[0]);
}
```

Output

```
in f: one
in managed: changed
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Obtain a Pointer to Byte Array

You can obtain a pointer to the array block in a Byte array by taking the address of the first element and assigning it to a pointer.

Example

```
// pointer_to_Byte_array.cpp
// compile with: /clr
using namespace System;
int main() {
   Byte bArr[] = {1, 2, 3};
   Byte* pbArr = &bArr[0];

   array<Byte> ^ bArr2 = gcnew array<Byte>{1,2,3};
   interior_ptr<Byte> pbArr2 = &bArr2[0];
}
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Load Unmanaged Resources into a Byte Array

This topic discusses several ways to load unmanaged resources into a Byte array.

Example

If you know the size of your unmanaged resource, you can preallocate a CLR array and then load the resource into the array using a pointer to the array block of the CLR array.

```
// load_unmanaged_resources_into_Byte_array.cpp
// compile with: /clr
using namespace System;
void unmanaged_func( unsigned char * p ) {
   for ( int i = 0; i < 10; i++ )
      p[ i ] = i;
}

public ref class A {
public:
   void func() {
      array<Byte> ^b = gcnew array<Byte>(10);
      pin_ptr<Byte> p =  &b[ 0 ];
      Byte * np = p;
      unmanaged_func( np );   // pass pointer to the block of CLR array.
      for ( int i = 0; i < 10; i++ )
         Console::Write( b[ i ] );
      Console::WriteLine();
   }
};

int main() {
   A^ g = gcnew A;
   g->func();
}
```

Output

```
0123456789
```

This sample shows how to copy data from an unmanaged memory block to a managed array.

```
// load_unmanaged_resources_into_Byte_array_2.cpp
// compile with: /clr
using namespace System;
using namespace System::Runtime::InteropServices;

#include <string.h>
int main() {
   char buf[] = "Native String";
   int len = strlen(buf);
   array<Byte> ^byteArray = gcnew array<Byte>(len + 2);

   // convert any native pointer to IntPtr by doing C-Style cast
   Marshal::Copy( (IntPtr)buf, byteArray, 0, len );
}
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Modify Reference Class in a Native Function

You can pass a reference class with a CLR array to a native function, and modify the class, using PInvoke services.

Example

Compile the following native library.

```
// modify_ref_class_in_native_function.cpp
// compile with: /LD
#include <stdio.h>
#include <windows.h>

struct S {
   wchar_t* str;
   int intarr[2];
};

extern "C"  {
   __declspec(dllexport) int bar(S* param) {
      printf_s("str: %S\n", param->str);
      fflush(stdin);
      fflush(stdout);
      printf_s("In native: intarr: %d, %d\n",
                param->intarr[0], param->intarr[1]);
      fflush(stdin);
      fflush(stdout);
      param->intarr[0]=300;param->intarr[1]=400;
      return 0;
    }
};
```

Compile the following assembly.

```
// modify_ref_class_in_native_function_2.cpp
// compile with: /clr
using namespace System;
using namespace System::Runtime::InteropServices;

[StructLayout(LayoutKind::Sequential, CharSet = CharSet::Unicode)]
ref class S  {
public:
   [MarshalAsAttribute(UnmanagedType::LPWStr)]
   String ^ str;
   [MarshalAsAttribute(UnmanagedType::ByValArray,
        ArraySubType=UnmanagedType::I4, SizeConst=2)]
   array<Int32> ^ intarr;
};

[DllImport("modify_ref_class_in_native_function.dll",
           CharSet=CharSet::Unicode)]
int bar([In][Out] S ^ param);

int main() {
   S ^ param = gcnew S;
   param->str = "Hello";
   param->intarr = gcnew array<Int32>(2);
   param->intarr[0] = 100;
   param->intarr[1] = 200;
   bar(param);   // Call to native function
   Console::WriteLine("In managed: intarr: {0}, {1}",
                      param->intarr[0], param->intarr[1]);
}
```

Output

```
str: Hello
In native: intarr: 100, 200
In managed: intarr: 300, 400
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Determine if an Image is Native or CLR

One way to determine whether an image was built for the common language runtime is to use **dumpbin** /CLRHEADER.

You can also programmatically check whether an image was built for the common language runtime. For more information, see How to: Detect /clr Compilation.

Example

The following sample determines whether an image was built to run on the common language runtime.

```
// detect_image_type.cpp
// compile with: /clr
using namespace System;
using namespace System::IO;

enum class CompilationMode {Invalid, Native, CLR };

static CompilationMode IsManaged(String^ filename) {
   try {
      array<Byte>^ data = gcnew array<Byte>(4096);
      FileInfo^ file = gcnew FileInfo(filename);
      Stream^ fin = file->Open(FileMode::Open, FileAccess::Read);
      Int32 iRead = fin->Read(data, 0, 4096);
      fin->Close();

      // Verify this is a executable/dll
      if ((data[1] << 8 | data[0]) != 0x5a4d)
         return CompilationMode::Invalid;

      // This will get the address for the WinNT header
      Int32 iWinNTHdr = data[63]<<24 | data[62]<<16 | data[61] << 8 | data[60];

      // Verify this is an NT address
      if ((data[iWinNTHdr+3] << 24 | data[iWinNTHdr+2] << 16 | data[iWinNTHdr+1] << 8 | dat
a[iWinNTHdr]) != 0x00004550)
         return CompilationMode::Invalid;

      Int32 iLightningAddr = iWinNTHdr + 24 + 208;
      Int32 iSum = 0;
      Int32 iTop = iLightningAddr + 8;

      for (int i = iLightningAddr; i < iTop; ++i)
         iSum |= data[i];

      if (iSum == 0)
         return CompilationMode::Native;
      else
         return CompilationMode::CLR;
   }
   catch(Exception ^e) {
      throw(e);
   }
}

int main() {
   array<String^>^ args = Environment::GetCommandLineArgs();

   if (args->Length < 2) {
      Console::WriteLine("USAGE : detect_clr <assembly_name>\n");
      return -1;
   }

   Console::WriteLine("{0} is compiled {1}", args[1], IsManaged(args[1]));
}
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Add Native DLL to Global Assembly Cache

You can put a native DLL (not COM) into the Global Assembly Cache.

Example

/ASSEMBLYLINKRESOURCE lets you embed a native DLL in an assembly.

For more information, see /ASSEMBLYLINKRESOURCE (Link to .NET Framework Resource).

```
/ASSEMBLYLINKRESOURCE:MyComponent.dll
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Hold Reference to Value Type in Native Type

Use **gcroot** on the boxed type to hold a reference to a value type in a native type.

Example

```
// reference_to_value_in_native.cpp
// compile with: /clr
#using <mscorlib.dll>
#include <vcclr.h>

using namespace System;

public value struct V {
   String ^str;
};

class Native {
public:
   gcroot< V^ > v_handle;
};

int main() {
   Native native;
   V v;
   native.v_handle = v;
   native.v_handle->str = "Hello";
   Console::WriteLine("String in V: {0}", native.v_handle->str);
}
```

Output

```
String in V: Hello
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Hold Object Reference in Native Function

You can use gcroot.h, which wraps GCHandle, to hold a CLR object reference in unmanaged memory. Alternatively, you can use **GCHandle** directly.

Example

```
// hold_object_reference.cpp
// compile with: /clr
#include "gcroot.h"
using namespace System;

#pragma managed
class StringWrapper {

private:
   gcroot<String ^ > x;

public:
   StringWrapper() {
      String ^ str = gcnew String("ManagedString");
      x = str;
   }

   void PrintString() {
      String ^ targetStr = x;
      Console::WriteLine("StringWrapper::x == {0}", targetStr);
   }
};
#pragma unmanaged
int main() {
   StringWrapper s;
   s.PrintString();
}
```

Output

```
StringWrapper::x == ManagedString
```

GCHandle gives you a means to hold a managed object reference in unmanaged memory. You use the Alloc method to create an opaque handle to a managed object and Free to release it. Also, the Target method allows you to obtain the object reference back from the handle in managed code.

```
// hold_object_reference_2.cpp
// compile with: /clr
using namespace System;
using namespace System::Runtime::InteropServices;

#pragma managed
class StringWrapper {
   IntPtr m_handle;
public:
   StringWrapper() {
      String ^ str = gcnew String("ManagedString");
      m_handle = static_cast<IntPtr>(GCHandle::Alloc(str));
   }
   ~StringWrapper() {
      static_cast<GCHandle>(m_handle).Free();
   }


   void PrintString() {
      String ^ targetStr = safe_cast< String ^ >(static_cast<GCHandle>(m_handle).Target);
      Console::WriteLine("StringWrapper::m_handle == {0}", targetStr);
   }
```

```
};

#pragma unmanaged
int main() {
   StringWrapper s;
   s.PrintString();
}
```

Output

```
StringWrapper::m_handle == ManagedString
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Detect /clr Compilation

Use the **_MANAGED** or **_M_CEE** macro to see if a module is compiled with **/clr**. For more information, see /clr (Common Language Runtime Compilation).

For more information about macros, see Predefined Macros.

Example

```
// detect_CLR_compilation.cpp
// compile with: /clr
#include <stdio.h>

int main() {
   #if (_MANAGED == 1) || (_M_CEE == 1)
      printf_s("compiling with /clr\n");
   #else
      printf_s("compiling without /clr\n");
   #endif
}
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Convert Between System::Guid and _GUID

The following code sample shows how to convert between a Guid and a **_GUID**.

Example

```
// convert_guids.cpp
// compile with: /clr
#include <windows.h>
#include <stdio.h>

using namespace System;

Guid FromGUID( _GUID& guid ) {
   return Guid( guid.Data1, guid.Data2, guid.Data3,
                guid.Data4[ 0 ], guid.Data4[ 1 ],
                guid.Data4[ 2 ], guid.Data4[ 3 ],
                guid.Data4[ 4 ], guid.Data4[ 5 ],
                guid.Data4[ 6 ], guid.Data4[ 7 ] );
}

_GUID ToGUID( Guid& guid ) {
   array<Byte>^ guidData = guid.ToByteArray();
   pin_ptr<Byte> data = &(guidData[ 0 ]);

   return *(_GUID *)data;
}

int main() {
   _GUID ng = {0x11111111,0x2222,0x3333,0x44,0x55,0x55,0x55,0x55,0x55,0x55,0x55};
   Guid mg;

   Console::WriteLine( (mg = FromGUID( ng )).ToString() );
   _GUID ng2 = ToGUID( mg );

   printf_s(  "%x-%x-%x-", ng2.Data1, ng2.Data2, ng2.Data3 );
   for (int i = 0 ; i < 8 ; i++) {
      if (i == 2)
         printf_s("-");
      printf_s("%x", ng2.Data4[i]);
   }
   printf_s("\n");
}
```

Output

```
11111111-2222-3333-4455-555555555555
11111111-2222-3333-4455-555555555555
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Specify an out Parameter

This sample shows how to specify that a function parameter is an out parameter and how to call that function from a C# program.

An out parameter is specified in Visual C++ with OutAttribute .

Example

The first part of this sample is a Visual C++ DLL with a type that contains a function with an out parameter.

```
// cpp_out_param.cpp
// compile with: /LD /clr:safe
using namespace System;
public value struct TestStruct {
   static void Test([Runtime::InteropServices::Out] String^ %s) {
      s = "a string";
   }
};
```

This is a C# client that consumes the Visual C++ component created in the previous example.

```
// cpp_out_param_2.cs
// compile with: /reference:cpp_out_param.dll
using System;
class TestClass {
   public static void Main() {
      String t;
      TestStruct.Test(out t);
      System.Console.WriteLine(t);
   }
}
```

Output

```
a string
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Use a Native Type in a /clr Compilation

You can define a native type in a **/clr** compilation and any use of that native type from within the assembly is valid. However, native types will not be available for use from referenced metadata.

Each assembly must contain the definition of every native type it will use.

For more information, see /clr (Common Language Runtime Compilation).

Example

This sample creates a component that defines and uses a native type.

```
// use_native_type_in_clr.cpp
// compile with: /clr /LD
public struct NativeClass {
   static int Test() { return 98; }
};

public ref struct ManagedClass {
   static int i = NativeClass::Test();
   void Test() {
      System::Console::WriteLine(i);
   }
};
```

This sample defines a client that consumes the component. Notice that it is an error to access the native type, unless it is defined in the compiland.

```
// use_native_type_in_clr_2.cpp
// compile with: /clr
#using "use_native_type_in_clr.dll"
// Uncomment the following 3 lines to resolve.
// public struct NativeClass {
//    static int Test() { return 98; }
// };

int main() {
   ManagedClass x;
   x.Test();

   System::Console::WriteLine(NativeClass::Test());   // C2653
}
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Declare Handles in Native Types

You cannot declare a handle type in a native type. vcclr.h provides the type-safe wrapper template gcroot, to refer to a CLR object from the C++ heap. This template lets you embed a virtual handle in a native type and treat it as if it were the underlying type.

The gcroot template is implemented using the facilities of the value class System::Runtime::InteropServices::GCHandle, which provides "handles" into the garbage-collected heap. Note that the handles themselves are not garbage collected and are freed when no longer in use by the destructor in the gcroot class (this destructor cannot be called manually). If you instantiate a gcroot object on the native heap, you must call delete on that resource.

The runtime will maintain an association between the handle and the CLR object, which it references. When the CLR object moves with the garbage-collected heap, the handle will return the new address of the object. A variable does not have to be pinned before it is assigned to a gcroot template.

Example

This sample shows how to create a gcroot object on the native stack.

```
// mcpp_gcroot.cpp
// compile with: /clr
#include <vcclr.h>
using namespace System;

class CppClass {
public:
   gcroot<String^> str;   // can use str as if it were String^
   CppClass() {}
};

int main() {
   CppClass c;
   c.str = gcnew String("hello");
   Console::WriteLine( c.str );   // no cast required
}
```

Output

```
hello
```

This sample shows how to create a gcroot object on the native heap.

```
// mcpp_gcroot_2.cpp
// compile with: /clr
// compile with: /clr
#include <vcclr.h>
using namespace System;

struct CppClass {
   gcroot<String ^> * str;
   CppClass() : str(new gcroot<String ^>) {}

   ~CppClass() { delete str; }

};

int main() {
   CppClass c;
   *c.str = gcnew String("hello");
   Console::WriteLine( *c.str );
}
```

Output

```
hello
```

This sample shows how to gcroot to hold references to value types (not reference types) in a native type by using gcroot on the boxed type.

```
// mcpp_gcroot_3.cpp
// compile with: /clr
#include < vcclr.h >
using namespace System;

public value struct V {
   String^ str;
};

class Native {
public:
   gcroot< V^ > v_handle;
};

int main() {
   Native native;
   V v;
   native.v_handle = v;
   native.v_handle->str = "Hello";
   Console::WriteLine("String in V: {0}", native.v_handle->str);
}
```

Output

```
String in V: Hello
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

How to: Wrap Native Class for Use by C#

This sample shows how to wrap a native C++ class so it can be consumed by code authored in C#, or other .NET language.

Example

```
// wrap_native_class_for_mgd_consumption.cpp
// compile with: /clr /LD
#include <windows.h>
#include <vcclr.h>
#using <System.dll>

using namespace System;

class UnmanagedClass {
public:
   LPCWSTR GetPropertyA() { return 0; }
   void MethodB( LPCWSTR ) {}
};

public ref class ManagedClass {
public:
   // Allocate the native object on the C++ Heap via a constructor
   ManagedClass() : m_Impl( new UnmanagedClass ) {}

   // Deallocate the native object on a destructor
   ~ManagedClass() {
      delete m_Impl;
   }

protected:
   // Deallocate the native object on the finalizer just in case no destructor is called
   !ManagedClass() {
      delete m_Impl;
   }

public:
   property String ^  get_PropertyA {
      String ^ get() {
         return gcnew String( m_Impl->GetPropertyA());
      }
   }

   void MethodB( String ^ theString ) {
      pin_ptr<const WCHAR> str = PtrToStringChars(theString);
      m_Impl->MethodB(str);
   }

private:
   UnmanagedClass * m_Impl;
};
```

See Also

Reference

Using C++ Interop (Implicit PInvoke)

General Concepts

You can add functionality to your Visual C++ program using the Visual C++ libraries and various technologies and techniques.

In This Section

- Data Access (C++)
- DLLs
- International Programming
- Multithreading

See Also

Other Resources

Programming Guide

Data Access (C++)

This section covers the legacy technologies you can use for database programming in Visual C++. For information on data access programming using Visual C++ 2005 and SQL Server 2005, see Data Access Using ADO.NET in C++, Accessing Data (Visual Studio), and Creating SQL Server 2005 Objects in Managed Code.

In This Section

Data Access Programming

Describes data access programming with Visual C++, where the preferred way is to use one of the class libraries such as the Active Template Class Library (ATL) or Microsoft Foundation Class (MFC) Library, which simplify working with the database APIs.

OLE DB Programming

Provides links to conceptual topics discussing the OLE DB database technology and the OLE DB Template Library.

Open Database Connectivity (ODBC)

Provides links to conceptual topics discussing targeting ODBC with MFC.

Data-Bound Controls (ADO and RDO)

Discusses a reusable databinding mechanism through ActiveX controls in MFC projects that lets you rapidly develop database applications.

Related Sections

Adding Functionality

Provides links to topics describing conceptual information about the Visual C++ libraries and topics discussing various coding technologies and techniques.

OLE DB Templates

Provides reference material for the OLE DB consumer and provider templates, a set of template classes that implement many commonly used OLE DB interfaces.

MFC Reference

Provides reference material for the MFC Library, a set of classes that constitute an application framework, which is the framework of an application written for the Windows API.

Data Access Programming

Visual C++ provides several ways to work with databases. The preferred way is to use one of the class libraries such as the Active Template Class Library (ATL) or Microsoft Foundation Class (MFC) Library, which simplify working with the database APIs.

Note
This topic covers the legacy technologies you can use for database programming in Visual C++. For information on data access programming using Visual C++ 2005 and SQL Server 2005, see Data Access Using ADO.NET in C++, Accessing Data (Visual Studio), and Creating SQL Server 2005 Objects in Managed Code.

The library classes support the following kinds of data access:

- ATL provides OLE DB templates and database attributes.
- MFC provides Open Database Connectivity (ODBC) and an ODBC driver.

These libraries supply abstractions that simplify working with databases, complete with the speed, power, and flexibility of C++. They integrate your data access work with the library's application framework.

Alternately, you can directly call database API functions from the COM, ODBC, or DAO software development kits (SDKs). For information about programming directly with the COM, DAO, or ODBC API functions, see the COM SDK, DAO SDK, or ODBC SDK.

Use ATL OLE DB if you need to access data, regardless of the form in which it is stored. Use the MFC ODBC classes when you are not using Microsoft Jet (.mdb) databases and want to work with the ODBC API for complete data-source independence. Use the MFC DAO classes when you want to work with Microsoft Jet (.mdb) databases or with external databases such as ODBC data sources.

Note
Microsoft recommends using OLE DB or ODBC for new projects. DAO should only be used in maintaining existing applications.

Besides writing stand-alone database applications, you can often use a database effectively in other kinds of programs as a convenient storage and retrieval medium.

To learn more about	See
Selecting a database technology	
ODBC vs. DAO	Should I Use DAO or ODBC?
Using the Microsoft Knowledge Base to find additional articles on database topics written by product support engineers	Microsoft Knowledge Base
ATL Database Support (OLE DB)	
OLE DB programming (conceptual topics)	OLE DB Programming Overview
Using the OLE DB consumer templates (conceptual topics)	OLE DB Consumer Templates
OLE DB consumer attributes	OLE DB Consumer Attributes
Using the OLE DB provider templates (conceptual topics)	OLE DB Provider Templates
Adding an OLE DB consumer to an MFC project	Creating an OLE DB Consumer

MFC Database Support (ODBC and DAO)	
What DAO and ODBC are	What Are DAO and ODBC?
When to use the MFC database classes	When Should I Use the Database Classes?
Learn about the MFC database programming model	What Is the MFC Database Programming Model?.
Choose between the MFC DAO classes and the MFC ODBC classes	Should I Use DAO or ODBC?.
Data sources you can access with DAO and ODBC	What Data Sources Can I Access with DAO and ODBC?
Open Database Connectivity (ODBC)	ODBC and MFC
Whether you can call DAO or ODBC APIs directly while using the classes	Can I Call DAO or ODBC Directly?
What ODBC drivers are provided	ODBC Driver List
How the database classes work with the MFC document/view architecture	MFC: Using Database Classes with Documents and Views
Installing MFC database support; what ODBC drivers are installed in Visual C++ by default; what ODBC and DAO SDK components are installed	Installing MFC Database Support
Data-Bound Controls (ADO and RDO)	
Writing a program that uses data-bound controls	Data-Bound Controls (ADO and RDO)
Data binding using ActiveX controls	MFC ActiveX Controls: Using Data Binding in an ActiveX Control
Distributing ActiveX controls	MFC ActiveX Controls: Distributing ActiveX Controls

See Also

Other Resources

Data Access (C++)

Installing Database Support

When you run Setup for Visual C++ .NET, the following database components are installed automatically:

- All the necessary ATL OLE DB components. For more information, see Installing ATL Database Support.
- A set of ODBC drivers with the ODBC driver manager and the ODBC administrator program. For more information, see "ODBC Drivers Installed" and "ODBC SDK Components Installed" in Installing MFC Database Support.
- Necessary components from the DAO software development kit (SDK). This includes Help files, which are not integrated with this documentation. However, if you work with DAO, you need to install a version of Jet compatible with your operating system. For more information, see "DAO SDK Components Installed" in Installing MFC Database Support.

As part of the baseline installation, Setup also installs Microsoft Data Access Components (MDAC), which are necessary to support data access programming in Visual C++ .NET.

Visual C++ .NET installs the MDAC 2.7 SDK. You should check the Microsoft Universal Data Access Web site at http://microsoft.com/data/ for updates to and news about the MDAC SDK.

If you are redistributing data access applications, you should also have the MDAC 2.7 redistribution program. The MDAC 2.7 SDK is designed for use with the MDAC 2.7 redistribution program (Mdac_typ.exe) included in the MDAC directory on the Visual Studio .NET Prerequisites CD-ROM. You can also download Mdac_typ.exe from the MDAC 2.7 SDK download link previously listed. For more information about redistributing components, see Redistributing Controls.

See Also

Other Resources

Data Access (C++)

Installing ATL Database Support

Setup installs all the necessary components to support OLE DB programming as part of the baseline installation. The OLE DB interfaces are installed as part of the MDAC SDK. The OLE DB templates, which support OLE DB programming in Visual C++ .NET, are installed as part of the ATL library. Visual C++ .NET Setup installs both the MDAC SDK (version 2.7) and the ATL library.

For updates to and news about OLE DB, see the Universal Data Access Web site at http://microsoft.com/data/.

See Also

Concepts

OLE DB Programming
OLE DB Consumer Templates (C++)
OLE DB Provider Templates (C++)

Other Resources

Data Access (C++)

Installing MFC Database Support

ODBC Drivers Installed

Setup installs the following ODBC drivers:

- Microsoft Access driver
- Microsoft dBASE driver
- Microsoft Excel driver
- Microsoft FoxPro VFP driver
- Microsoft Visual FoxPro driver
- Microsoft ODBC for Oracle driver
- Microsoft Paradox driver
- Microsoft Text driver
- Microsoft SQL Server driver

For information about obtaining additional drivers and for a list of ODBC drivers included in this version of Visual C++, see ODBC Driver List.

ODBC SDK Components Installed

Visual C++ includes many key ODBC components, including the required header files, libraries, DLLs, and tools. These include the ODBC Administrator Control Panel application, which you use to configure ODBC data sources, and the ODBC driver manager. Also included are ODBC drivers for many popular DBMSs, as listed in ODBC Drivers Installed.

The ODBC SDK gives you additional information and tools for writing and testing ODBC drivers. Note that as of Visual C++ .NET, the ODBC SDK is no longer included on the Visual C++ .NET installation media and is available as part of the Platform SDK installed with the Visual Studio .NET Prerequisites.

DAO SDK Components Installed

Note
Microsoft recommends using OLE DB or ODBC for new projects. DAO should only be used in maintaining existing applications.

The following components of the DAO SDK are installed by default:

- Microsoft Jet (4.0 SP3)
- Microsoft Jet (3.x MDB)
- Microsoft Jet (1.x, 2.x)
- All the database formats listed under What Databases Can I Access with DAO and ODBC?

In Visual C++ .NET, the DAO SDK is installed with the Visual Studio .NET Prerequisites. Run \Jet\Jetsetup.exe.

Note
Microsoft recommends that you use Jet 4.0 Service Pack 3 (version 2927.04) or later. Jet 4.0 Service Pack 3 shipped with Windows 2000 and Windows ME. Using this version of Jet reduces the number of Jet versions that must be tested with your application. Windows XP might ship with another version of Jet. See "Note on Redistributing DAO Components" in Redistributing Controls.

If you want to install other DAO SDK components, such as the DAO SDK C++ classes, example files, or the Windows Help version of the DAO Help file, install the DAO SDK from the Visual C++ 6.0 CD-ROM.

For updates to and news about OLE DB, see the Universal Data Access Web site at http://microsoft.com/data/.

See Also

Concepts

Data Access Programming

ATL Database Classes (OLE DB Templates)

Microsoft provides several implementations of OLE DB, a set of COM interfaces that provide uniform access to data in diverse information sources and formats.

The OLE DB Templates are C++ templates in ATL that make the high-performance OLE DB database technology easier to use by providing classes that implement many of the commonly used OLE DB interfaces.

This template library contains two parts:

- OLE DB consumer templates Used to implement an OLE DB client (consumer) application.
- OLE DB provider templates Used to implement an OLE DB server (provider) application.

In addition, the OLE DB consumer attributes provide a convenient way to create OLE DB consumers. The OLE DB attributes inject code based on the OLE DB consumer templates to create working OLE DB consumers.

Note that the MFC library contains one class, COleDBRecordView, that displays database records in controls. The view is a form view directly connected to a **CRowset** object, and displays the fields of the **CRowset** object in the dialog template's controls. For more information, see **COleDBRecordView**.

For more information, see OLE DB Programming.

For updates to and news about OLE DB, see the Universal Data Access Web site at http://microsoft.com/data/.

See Also

Reference

Creating an OLE DB Consumer
Creating an OLE DB Provider

Concepts

OLE DB Consumer Templates Reference
OLE DB Provider Templates Reference

Other Resources

OLE DB Templates Samples
OLE DB Programmer's Reference

MFC Database Classes (ODBC and DAO)

Note
Microsoft recommends using OLE DB or ODBC for new projects. DAO should only be used in maintaining existing applications.

Because MFC implements DAO and ODBC differently, the documentation for each class is almost completely separate. The DAO SDK is provided on the Visual C++ CD-ROM, and the ODBC SDK is provided on the MSDN Library CD-ROM. The following table lists the next topics to read if you are just beginning to study the MFC DAO classes or the MFC ODBC classes.

For more information, see the following topics:

- ODBC and MFC
- MFC Database Documentation
- Using Database Classes with Documents and Views
- Using Database Classes Without Documents and Views
- Check the Microsoft Universal Data Access Web site at http://microsoft.com/data/ for updates to and news about ODBC and DAO.

See Also

Concepts

Data Access Programming

MFC Database Documentation

The MFC documentation for DAO and ODBC classes consists of the components listed in the following table.

MFC Database Documentation

For documentation about	See
Classes for both DAO and ODBC	The class name in the *MFC Reference*
Global functions and macros for both DAO and ODBC	The function or macro name in the *MFC Reference*
Programming with the MFC ODBC classes	ODBC and MFC
Technical notes for both DAO and ODBC	MFC Technical Notes

MFC Documentation and DAO Documentation

Throughout the MFC documentation for the MFC DAO classes, you will find references to topics in the DAO SDK documentation, included in the online documentation. Because MFC encapsulates (or wraps) DAO, the MFC documentation:

- Focuses on MFC and how it differs from the underlying DAO implementation.
- Points you to the DAO SDK Help topics for the underlying details. These cross-references are always worded as "topic *X* in DAO Help."
- Points out differences where MFC does things differently from the way DAO does them. MFC does not wrap all of DAO. For example, MFC does not supply objects for DAO security functionality.

Note
The DAO SDK Help is a separate Help file. There are no hypertext links from this documentation to DAO help in this version o f Visual C++.

Note
You might have to do some translating when you are browsing topics in DAO SDK Help. Examples in the primary DAO SDK d ocumentation are written in the Basic programming language, not C++. (But the DAO SDK supplies a set of C++ examples th at do not use MFC.)

MFC Documentation and ODBC Documentation

The MFC documentation for the MFC ODBC classes is organized differently. The MFC ODBC classes supply a high-level abstraction that rests on ODBC rather than a wrapper of the ODBC API. Therefore, the two documentation sets are less connected than are the MFC and DAO documentation sets. The ODBC documentation uses the C language, which is much closer to C++ than is Basic.

See Also

Concepts

MFC Database Classes (ODBC and DAO)
ODBC Basics

MFC: Using Database Classes with Documents and Views

You can use the MFC database classes — DAO or ODBC — with or without the document/view architecture. This topic emphasizes working with documents and views. It explains:

- How to write a form-based application using a **CRecordView** or **CDaoRecordView** object as the main view on your document.
- How to use recordset objects in your documents and views.
- Other considerations.

For alternatives, see MFC: Using Database Classes Without Documents and Views.

Writing a Form-Based Application

Many data-access applications are based on forms. The user interface is a form containing controls in which the user examines, enters, or edits data. To make your application form based, use class **CRecordView** or **CDaoRecordView**. When you run the MFC Application Wizard and select **ODBC** client type on the **Database Support** page, the project uses **CRecordView** for the view class. The wizards no longer support DAO, so if you want to use **CDaoRecordView**, you have to code it manually.

In a form-based application, each record view object stores a pointer to a **CRecordset** or **CDaoRecordset** object. The framework's record field exchange (RFX) mechanism exchanges data between the recordset and the data source. The dialog data exchange (DDX) mechanism exchanges data between the field data members of the recordset object and the controls on the form. **CRecordView** or **CDaoRecordView** also provides default command handler functions for navigating from record to record on the form.

To create a form-based application with the application wizard, see Creating a Forms-Based MFC Application and Database Support, MFC Application Wizard.

For a full discussion of forms, see Record Views.

Using Recordsets in Documents and Views

Many simple form-based applications do not need documents. If your application is more complex, you probably want to use a document as a proxy for the database, storing a **CDatabase** or **CDaoDatabase** object that connects to the data source. Form-based applications usually store a pointer to a recordset object in the view. Other kinds of database applications store recordsets and **CDatabase** or **CDaoDatabase** object in the document. Here are some possibilities for using documents in database applications:

- If you are accessing a recordset in a local context, create a **CRecordset** or **CDaoRecordset** object locally in member functions of the document or the view, as needed.

  Declare a recordset object as a local variable in a function. Pass **NULL** to the constructor, which causes the framework to create and open a temporary **CDatabase** or **CDaoDatabase** object for you. As an alternative, pass a pointer to a **CDatabase** or **CDaoDatabase** object. Use the recordset within the function and let it be destroyed automatically when the function exits.

  When you pass **NULL** to a recordset constructor, the framework uses information returned by the recordset's **GetDefaultConnect** member function to create a **CDatabase** or **CDaoDatabase** object and open it. The wizards implement **GetDefaultConnect** for you.

- If you are accessing a recordset during the lifetime of your document, embed one or more **CRecordset** or **CDaoRecordset** objects in your document.

  Construct the recordset objects either when you initialize the document or as needed. You might write a function that returns a pointer to the recordset if it already exists or constructs and opens the recordset if it does not exist yet. Close, delete, and recreate the recordset as needed, or call its **Requery** member function to refresh the records.

- If you are accessing a data source during the lifetime of your document, embed a **CDatabase** or **CDaoDatabase** object or store a pointer to a **CDatabase** or **CDaoDatabase** object in it.

  The **CDatabase** or **CDaoDatabase** object manages a connection to your data source. The object is constructed automatically during document construction, and you call its **Open** member function when you initialize the document. When you construct recordset objects in document member functions, you pass a pointer to the document's **CDatabase**

or **CDaoDatabase** object. This associates each recordset with its data source. The database object is usually destroyed when the document closes. The recordset objects are typically destroyed when they exit the scope of a function.

Other Factors

Form-based applications often do not have any use for the framework's document serialization mechanism, so you might want to remove, disable, or replace the **New** and **Open** commands on the **File** menu. See the article Serialization: Serialization vs. Database Input/Output.

You might also want to make use of the many user-interface possibilities that the framework can support. For example, you could use multiple **CRecordView** or **CDaoRecordView** objects in a splitter window, open multiple recordsets in different multiple document interface (MDI) child windows, and so on.

You might want to implement printing of whatever is in your view, whether it is a form implemented with **CRecordView** or **CDaoRecordView** or something else. As classes derived from **CFormView**, **CRecordView** and **CDaoRecordView** do not support printing, but you can override the **OnPrint** member function to allow printing. For more information, see class CFormView.

You might not want to use documents and views at all. In that case, see MFC: Using Database Classes Without Documents and Views.

See Also

Concepts

MFC Database Classes (ODBC and DAO)

MFC: Using Database Classes Without Documents and Views

Sometimes you might not want to use the framework's document/view architecture in your database applications. This topic explains:

- When you do not need to use documents such as document serialization.
- Application wizard options to support applications without serialization and without document-related **File** menu commands such as **New**, **Open**, **Save**, and **Save As**.
- How to work with an application that uses a minimal document.
- How to structure an application with no document or view.

When You Do Not Need Documents

Some applications have a distinct concept of a document. These applications typically load all or most of a file from storage into memory with a **File Open** command. They write the updated file back to storage all at once with a **File Save** or **Save As** command. What the user sees is a data file.

Some categories of applications, however, do not require a document. Database applications operate in terms of transactions. The application selects records from a database and presents them to the user, often one at a time. What the user sees is usually a single current record, which might be the only one in memory.

If your application does not require a document for storing data, you can dispense with some or all of the framework's document/view architecture. How much you dispense with depends on the approach you prefer. You might:

- Use a minimal document as a place to store a connection to your data source but dispense with normal document features such as serialization. This is useful when you want several views of the data and want to synchronize all the views, updating them all at once and so on.
- Use a frame window, into which you draw directly, rather than using a view. In this case, you omit the document and store any data or data connections in the frame-window object.

Application Wizard Options for Documents and Views

The MFC Application Wizard has several options in **Select database support**, which are listed in the following table. If you use the MFC Application Wizard to create an application, all these options produce applications with documents and views. Some options provide documents and views that omit unneeded document functionality. For more information, see Database Support, MFC Application Wizard.

Option	View	Document
None	Derived from **CView**.	Provides no database support. This is the default option. If you select the **Document/view architecture support** option on the Application Type page, you get full document support including serialization and **New**, **Open**, **Save**, and **Save As** commands on the **File** menu. See Applications with No Document.
Header files only	Derived from **CView**.	Provides the basic level of database support for your application. Includes Afxdb.h. Adds link libraries, but does not create any database-specific classes. You can create record sets later and use them to examine and update records.
Database view without file support	Derived from **CRecordView**	Provides document support but no serialization support. Document can store recordset and coordinate multiple views; does not support serialization or the **New**, **Open**, **Save**, and **Save As** commands. See Applications with Minimal Documents. If you include a database view, you must specify the source of the data. Includes database header files, link libraries, a record view, and a recordset. (Available only for applications with the **Document/view architecture support** option selected on the Application Type page.)

Database view with file support	Derived from **CRecordView**	Provides full document support, including serialization and document-related **File** menu commands. Database applications typically operate on a per-record basis rather than on a per-file basis and so do not need serialization. However, you might have a special use for serialization. See Applications with Minimal Documents. If you include a database view, you must specify the source of the data. Includes database header files, link libraries, a record view, and a recordset. (Available only for applications with the **Document/view architecture support** option selected on the Application Type page.)

For a discussion of alternatives to serialization and alternative uses for serialization, see Serialization: Serialization vs. Database Input/Output.

Applications with Minimal Documents

The MFC Application Wizard has two options that support form-based data access applications. Each option creates a **CRecordView**-derived view class and a document. They differ in what they leave out of the document.

Document Without File Support

Select the application wizard database option **Database view without file support** if you do not need document serialization. The document serves the following useful purposes:

- It is a convenient place to store a **CRecordset** object.

  This usage parallels ordinary document concepts: the document stores the data (or, in this case, a set of records) and the view is a view of the document.

- If your application presents multiple views (such as multiple record views), a document supports coordinating the views.

  If multiple views show the same data, you can use the **CDocument::UpdateAllViews** member function to coordinate updates to all views when any view changes the data.

You usually use this option for simple form-based applications. The application wizard supports a convenient structure for such applications automatically.

Document with File Support

Select the application wizard database option **Database view with file support** when you have an alternative use for the document-related **File** menu commands and document serialization. For the data-access portion of your program, you can use the document in the same way as described in Document Without File Support. You can use the document's serialization capability, for example, to read and write a serialized user profile document that stores the user's preferences or other useful information. For more ideas, see Serialization: Serialization vs. Database Input/Output.

The application wizard supports this option, but you must write the code that serializes the document. Store the serialized information in document data members.

Applications with No Document

You might sometimes want to write an application that does not use documents or views. Without documents, you store your data (such as a **CRecordset** object) in your frame-window class or your application class. Any additional requirements depend on whether the application presents a user interface.

Database Support with a User Interface

If you have a user interface (other than, for example, a console command-line interface), your application draws directly into the frame window's client area rather than into a view. Such an application does not use **CRecordView**, **CFormView**, or **CDialog** for its main user interface, but it normally use **CDialog** for ordinary dialogs.

Writing Applications Without Documents

Because the application wizard does not support creating applications without documents, you must write your own **CWinApp**-derived class and, if needed, also create a **CFrameWnd** or **CMDIFrameWnd** class. Override **CWinApp::InitInstance** and declare an application object as:

```
CYourNameApp theApp;
```

The framework still supplies the message-map mechanism and many other features.

Database Support Separate from the User Interface

Some applications need no user interface or only a minimal one. For example, suppose you are writing:

- An intermediate data-access object that other applications (clients) call for special processing of data between the application and the data source.
- An application that processes data without user intervention, such as an application that moves data from one database format to another or one that does calculations and performs batch updates.

Because no document owns the **CRecordset** or **CDaoRecordset** object, you probably want to store it as an embedded data member in your **CWinApp**-derived application class. Alternatives include:

- Not keeping a permanent **CRecordset** or **CDaoRecordset** object at all. You can pass **NULL** to your recordset class constructors. In that case, the framework creates a temporary **CDatabase** or **CDaoDatabase** object using the information in the recordset's **GetDefaultConnect** member function. This is the most likely alternative approach.
- Making the **CRecordset** or **CDaoRecordset** object a global variable. This variable should be a pointer to a recordset object that you create dynamically in your **CWinApp::InitInstance** override. This avoids attempting to construct the object before the framework is initialized.
- Using recordset objects as you would within the context of a document or a view. Create recordsets in the member functions of your application or frame-window objects.

See Also

Concepts

MFC Database Classes (ODBC and DAO)

Record (Data Access)

A record is a collection of data about a single entity, such as an account or a customer, stored in a table (a row of the table). A record consists of a group of contiguous columns (sometimes called fields) that contain data of various types. A set of records selected from a data source — often called a result set in database terms — is called a recordset in MFC. For more information, see Recordset (ODBC).

See Also

Concepts

Data Access Programming

Schema (Data Access)

A database schema describes the current structure of the tables and database views in the database. In general, wizard-generated code assumes that the schema for the table or tables accessed by a recordset will not change, but the database classes can deal with some schema changes, such as adding, reordering, or deleting unbound columns. If a table changes, you must manually update the recordset for the table, then recompile your application.

You can also supplement the wizard-generated code to deal with a database whose schema is not entirely known at compile time. For more information, see Recordset: Dynamically Binding Data Columns (ODBC).

See Also

Concepts

Data Access Programming
SQL
Recordset (ODBC)

Catalog Information

Information about the tables in a data source can include the names of tables and the columns in them, table privileges, names of primary and foreign keys, information about predefined queries or stored procedures, information about indexes on tables, and statistics about tables.

For more information, see Data Source: Determining the Schema of the Data Source (ODBC).

Note
In the MFC DAO classes, you can get catalog information as follows: Use CDaoDatabase::GetTableDefCount and CDaoDatabase::GetTableDefInfo to enumerate the tables in the database and obtain information for each table in a CDaoTableDefInfo structure.

See Also

Concepts

Data Access Programming

Transactions

The concept of a transaction was developed to handle cases in which the resulting state of the database depends on the total success of a series of operations. This could come about because successive operations might modify the results of previous operations. In such cases, if any one operation fails, the resulting state could be indeterminate.

To solve this problem, transactions group a series of operations in such a way that the integrity of the final result can be assured. Either all the operations must succeed and then be committed (written to the database) or the entire transaction fails. The cancellation of the transaction is called a rollback. Rollback allows a recovery from the changes and returns the database to the pretransaction state.

For example, in an automated banking transaction, if you transfer money from account A to account B, both the withdrawal from A and the deposit to B must succeed to process the funds correctly, or the whole transaction must fail.

A transaction must have ACID properties, which stand for the following:

- **Atomicity** A transaction is an atomic unit of work and executes exactly once; either all the work is done or none of it is.
- **Consistency** A transaction preserves the consistency of data, transforming one consistent state of data into another consistent state of data. Data bound by a transaction must be semantically preserved.
- **Isolation** A transaction is a unit of isolation and each occurs separately and independently of concurrent transactions. A transaction should never see the intermediate stages of another transaction.
- **Durability** A transaction is a unit of recovery. If a transaction succeeds, its updates persist, even if the system crashes or is shut down. If a transaction fails, the system remains in the state previous to committing the transaction.

You can support transactions in OLE DB (see Supporting Transactions in OLE DB) or ODBC (see Transaction (ODBC)).

A distributed transaction is a transaction that updates distributed data, that is, data on more than one networked computer system. If you want to support transactions over a distributed system, you should use the Microsoft .NET Framework rather than the OLE DB transaction support.

For information about transactions in the Microsoft .NET Framework, see Processing Transactions in the .NET Framework SDK.

See Also

Concepts

Data Access Programming

Record Views

This section applies only to the MFC ODBC and DAO classes. For information about OLE DB record views, see COleDBRecordView and Using OLE DB Record Views.

To support form-based data-access applications, the class library provides class CRecordView and class CDaoRecordView. A record view is a form view object whose controls are mapped directly to the field data members of a recordset object (and indirectly to the corresponding columns in a query result or table on the data source). Like their base class CFormView, **CRecordView** and **CDaoRecordView** are based on a dialog template resource.

Form Uses

Forms are useful for a variety of data-access tasks:

- Entering data
- Performing read-only examination of data
- Updating data

Further Reading About Record Views

The material in topics applies to both the ODBC-based and the DAO-based classes. Use **CRecordView** for ODBC and **CDaoRecordView** for DAO.

Topics include:

- Features of Record View Classes
- Data Exchange for Record Views
- Your Role in Working with a Record View
- Designing and Creating a Record View
- Using a Record View

See Also

Concepts

Data Access Programming
ODBC Driver List

Features of Record View Classes

You can do form-based data-access programming with class CFormView, but CRecordView and CDaoRecordView are generally better classes to derive from. In addition to their **CFormView** features, **CRecordView** and **CDaoRecordView**:

- Provide dialog data exchange (DDX) between the form controls and the associated recordset object.
- Handle Move First, Move Next, Move Previous, and Move Last commands for navigating through the records in the associated recordset object.
- Update changes to the current record when the user moves to another record.

For more information about navigation, see Record Views: Supporting Navigation in a Record View.

See Also

Concepts

Record Views
ODBC Driver List

Data Exchange for Record Views

When you use Add Class to map the controls in a record view's dialog template resource to the fields of a recordset, the framework manages data exchange in both directions — from recordset to controls and from controls to recordset. Using the DDX mechanism means that you do not have to write the code to transfer data back and forth yourself.

DDX for record views works in conjunction with:

- RFX for recordsets of class **CRecordset** (ODBC).
- DFX for recordsets of class **CDaoRecordset** (DAO).

Although they differ in implementation, at the interface level RFX and DFX are very similar data exchange mechanisms. The DAO version, DFX, is modeled closely on the earlier ODBC version, RFX. If you know how to use RFX, you know how to use DFX.

RFX and DFX move data between the current record of the data source and the field data members of a recordset object. DDX moves the data from the field data members to the controls in the form. This combination fills the form controls initially and as the user moves from record to record. It can also move updated data back to the recordset and then to the data source.

The following figure shows the relationship between DDX and RFX (or DFX) for record views.

Dialog Data Exchange and Record Field Exchange

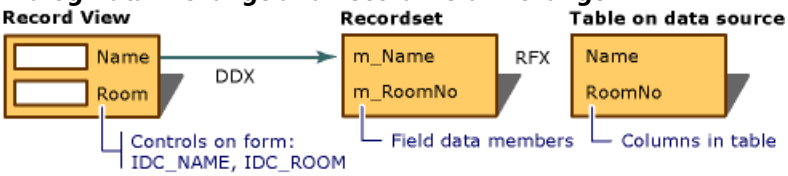

For more information about DDX, see Dialog Data Exchange and Validation. For more information about RFX, see Record Field Exchange (RFX).

See Also

Concepts

Record Views

ODBC Driver List

Your Role in Working with a Record View

The following table shows what you typically must do to work with a record view and what the framework does for you.

Working with a Record View: You and the Framework

You	The framework
Use the Visual C++ Dialog editor to design the form.	Creates a dialog template resource with controls.
Use the MFC Application Wizard to create classes derived from CRecordView and CRecordset or from CDaoRecordView and CDaoRecordset.	Writes the classes for you.
Map record view controls to recordset field data members.	Provides DDX between the controls and the recordset fields.
	Provides default command handlers for **Move First**, **Move Last**, **Move Next**, and **Move Previous** commands from menus or toolbar buttons.
	Updates changes to the data source.
[Optional] Write code to fill list boxes or combo boxes or other controls with data from a second recordset.	
[Optional] Write code for any special validations.	
[Optional] Write code to add or delete records.	

Form-based programming is only one approach to working with a database. For information about applications using some other user interface, or no user interface, see MFC: Using Database Classes with Documents and Views and MFC: Using Database Classes Without Documents and Views. For alternative approaches to displaying database records, see classes CListView and CTreeView.

See Also

Concepts

Record Views

ODBC Driver List

Designing and Creating a Record View

You can create your record view class with the MFC Application Wizard. If you use an application wizard, it creates the record view class and a dialog template resource for it (without controls). You must use the Visual C++ Dialog editor to add controls to the dialog template resource. On the other hand, if you use **Add Class**, you must first create the dialog template resource in the Dialog editor and then create the record view class.

This information applies to both **CRecordView** and **CDaoRecordView**.

To create your record view with the MFC Application Wizard

- See Database Support, MFC Application Wizard.

To design your form

- See Dialog Editor.

To create your record view class

- See Adding an MFC ODBC Consumer.

The following topics explain additional details of using record views:

- Record Views: Supporting Navigation in a Record View
- Record Views: Using a Record View
- Record Views: Filling a List Box from a Second Recordset

See Also

Concepts

Record Views
Recordset (ODBC)
ODBC Driver List

Supporting Navigation in a Record View

This topic explains how to support movement from record to record in your record view, including information about:

- Command handling for record scrolling commands.
- User-interface update handlers for scrolling commands.

The information in these topics applies to both CRecordView (ODBC) and CDaoRecordView (DAO).

See Also

Concepts

Record Views
ODBC Driver List

Command Handlers for Record Scrolling

Classes CRecordView and CDaoRecordView provide default command handling for the following standard commands:

- **ID_RECORD_MOVE_FIRST**
- **ID_RECORD_MOVE_LAST**
- **ID_RECORD_MOVE_NEXT**
- **ID_RECORD_MOVE_PREV**

The **OnMove** member function of classes **CRecordView** and **CDaoRecordView** provides default command handling for all four commands, which move from record to record. As these commands are issued, RFX (or DFX) loads the new record into the recordset's fields and DDX moves the values into the record form's controls. For information about RFX, see Record Field Exchange (RFX).

Note
Be sure to use these standard command IDs for any user-interface objects associated with the standard record navigation commands.

See Also

Concepts

Supporting Navigation in a Record View

User-Interface Updating for Record Views

CRecordView and **CDaoRecordView** also provide default user-interface update handlers for the navigation commands. These handlers automate enabling and disabling the user-interface objects — menu items and toolbar buttons. The application wizard supplies standard menus and, if you choose the **Dockable Toolbar** option, a set of toolbar buttons for the commands. If you create a record view class using **CRecordView**, you might want to add similar user-interface objects to your application.

To create menu resources with the menu editor

- Referring to the information about using the menu editor, create your own menu with the same four commands.

To create toolbar buttons with the graphics editor

- Referring to the information about using the toolbar editor, edit the toolbar resource to add toolbar buttons for your record navigation commands.

See Also

Concepts

Supporting Navigation in a Record View
Using a Record View

Using a Record View

This topic explains how you might commonly customize the default code for record views that the wizard writes for you. Typically, you want to constrain the record selection with a filter or parameters, perhaps sort the records, customize the SQL statement.

This information applies to both CRecordView (ODBC) and CDaoRecordView (DAO).

Using **CRecordView** or **CDaoRecordView** is much the same as using CFormView. The basic approach is to use the record view to display and perhaps update the records of a single recordset. Beyond that, you might want to use other recordsets as well, as discussed in Record Views: Filling a List Box from a Second Recordset.

See Also

Concepts

Record Views
ODBC Driver List

Record View Code Created by Application Wizard

The MFC Application Wizard overrides the view's **OnInitialUpdate** and **OnGetRecordset** member functions. After the framework creates the frame window, document, and view, it calls `OnInitialUpdate` to initialize the view. `OnInitialUpdate` obtains a pointer to the recordset from the document. A call to the base class CView::OnInitialUpdate function opens the recordset. The following code shows this process for a **CRecordView** — the code for a **CDaoRecordView** is similar:

```
void CSectionForm::OnInitialUpdate()
{
   m_pSet = &GetDocument()->m_sectionSet;
   CRecordView::OnInitialUpdate();
}
```

When the recordset opens, it selects records. CRecordset::Open or CDaoRecordset::Open makes the first record the current record, and DDX moves data from the recordset's field data members to the corresponding form controls in the view. For more information about RFX, see Record Field Exchange (RFX). For more information about DDX, see Dialog Data Exchange and Validation. For information about the document/view creation process, see Using the Classes to Write Applications for Windows.

Note

You should give your end users the capability to refresh the record view controls from the recordset. Without this capability, if a user changes a control's value to an illegal value, the user can be permanently stuck on the current record. To refresh the controls, you call the **CWnd** member function UpdateData with a parameter of **FALSE**.

See Also

Concepts

Using a Record View

Changes You Might Make to the Default Code

The MFC Application Wizard writes a recordset class for you that selects all records in a single table. You will often want to modify that behavior in one or more of the following ways:

- Set a filter or a sort order for the recordset. Do this in `OnInitialUpdate` after the recordset object is constructed but before its **Open** member function is called. For more information, see Recordset: Filtering Records (ODBC) and Recordset: Sorting Records (ODBC).
- Parameterize the recordset. Specify the actual run-time parameter value after the filter. For more information, see Recordset: Parameterizing a Recordset (ODBC)
- Pass a customized SQL string to the Open member function. For a discussion of what you can accomplish with this technique , see SQL: Customizing Your Recordset's SQL Statement (ODBC).

See Also

Concepts

Using a Record View

Filling a List Box from a Second Recordset

By default, a record view is associated with a single recordset object, whose fields are mapped to the record view's controls. Sometimes you might want to put a list box or combo box control in your record view and fill it with values from a second recordset object. The user can use the list box to select a new category of information to display in the record view. This topic explains how and when to do that.

Tip
Be aware that filling a combo box or list box from a data source might be slow. Take precautions against trying to fill a control from a recordset with a large number of records.

The model for this topic consists of a primary recordset that fills the controls of your form, while a secondary recordset fills a list box or combo box. Selecting a string from the list box causes your program to requery the primary recordset based on what was selected. The following procedure uses a combo box but applies equally to a list box.

To fill a combo box or list box from a second recordset

1. Create the recordset object (CRecordset for ODBC, CDaoRecordset for DAO).
2. Obtain a pointer to the CComboBox object for the combo box control.
3. Empty the combo box of any previous contents.
4. Move through all records in the recordset, calling CComboBox::AddString for each string from the current record you want to add to the combo box.
5. Initialize the selection in the combo box.

```
void CSectionForm::OnInitialUpdate()
{
    // ...

    // Fill the combo box with all of the courses
    CENROLLDoc* pDoc = GetDocument();
    if (!pDoc->m_courseSet.Open())
        return;

    // ...

    m_ctlCourseList.ResetContent();
    if (pDoc->m_courseSet.IsOpen())
    {
        while (!pDoc->m_courseSet.IsEOF() )
        {
            m_ctlCourseList.AddString(
                pDoc->m_courseSet.m_CourseID);
            pDoc->m_courseSet.MoveNext();
        }
    }
    m_ctlCourseList.SetCurSel(0);
}
```

This function uses a second recordset, `m_courseSet`, which contains a record for each course offered, and a **CComboBox** control, `m_ctlCourseList`, which is stored in the record view class.

The function gets `m_courseSet` from the document and opens it. Then it empties `m_ctlCourseList` and scrolls through `m_courseSet`. For each record, the function calls the combo box's **AddString** member function to add the course ID value from the record. Finally, the code sets the combo box's selection.

See Also

Concepts

Record Views

Data Access Frequently Asked Questions

The following are frequently asked questions (FAQ) about data access programming:

- What Are DAO and ODBC?
- When Should I Use the Database Classes?
- What Is the MFC Data Access Programming Model?
- Should I Use DAO or ODBC?
- What data sources can I access with DAO and ODBC?
- Can I call DAO or ODBC APIs directly?
- Are DDL and DML Supported?
- Where Can I Find Microsoft Knowledge Base Articles on Database Topics?

See Also

Concepts

Data Access Programming

What Are DAO and ODBC?

Both Data Access Objects (DAOs) and Open Database Connectivity (ODBC) are APIs that give you the ability to write applications that are independent of any particular database management system (DBMS).

DAO is familiar to database programmers using Microsoft Access Basic or Microsoft Visual Basic. DAO uses the Microsoft Jet database engine to provide a set of data access objects: database objects, tabledef and querydef objects, recordset objects, and others. DAO works best with .mdb files like those created by Microsoft Access, but you can also access ODBC data sources through DAO and the Microsoft Jet database engine.

ODBC provides an API that different database vendors implement through ODBC drivers specific to a particular DBMS. Your program uses this API to call the ODBC Driver Manager, which passes the calls to the appropriate driver. The driver, in turn, interacts with the DBMS using SQL.

Note
ODBC is a major part of the Microsoft Windows Open Standards Architecture (WOSA). DAO is optimized around the Microsoft Jet database engine, but you can still access ODBC and other external data sources with that engine, and the distinct ODBC API and the MFC classes based on it are still available and still have their role to play in your selection of database tools.

See Also

Other Resources

Data Access Frequently Asked Questions

When Should I Use the Database Classes?

Many applications require data storage in a database, and many other applications could benefit from using a database. A database gives you a flexible data repository that can be accessed, in many cases, by multiple users and multiple applications. Databases can store large amounts of data and provide fast data access for queries and updates.

See Also

Other Resources

Data Access Frequently Asked Questions

What Is the MFC Database Programming Model?

Although MFC implements DAO and ODBC quite differently underneath, they have similar interfaces that it relatively easy to port your applications from one to the other, particularly from ODBC to DAO. For information about porting from ODBC to DAO, see Technical Note 55. The DAO and ODBC interfaces in MFC are also very similar to that in Visual Basic.

The MFC programming model provides a database object for each open database. The database object represents your connection to the database. You make queries and updates using recordset objects. DAO provides additional objects, for working with table structure, saving queries for reuse, and so on, described later. MFC supplies classes for each of these objects: one set of classes for DAO and another set for ODBC.

Using MFC makes data access easier. The DAO and ODBC database classes supply high-level abstractions that free you from using DAO or ODBC directly. Writing to their APIs is more complex than using the MFC classes. This is especially true if you are writing small, relatively simple applications.

The database classes add the following components to the MFC class library:

- C++ database classes that supply a high-level API for accessing databases through either DAO or ODBC
- Extensions to the application wizard and **Add Class** for creating application-specific database classes
- Sample programs that illustrate use of the classes and the wizards
- Online documentation that includes overviews, articles about programming topics, and class reference materials

For information about these components, see ODBC and MFC.

For more information, see:

- Choosing between DAO and ODBC.
- Availability of database definition language (DDL) and database manipulation language (DML) in DAO and ODBC.
- Installing MFC database support.
- The ODBC classes in MFC.
- MFC data access programming documentation.
- How MFC implements ODBC.

See Also

Other Resources

Data Access Frequently Asked Questions

Should I Use DAO or ODBC?

Note
As of Visual C++ .NET, the Visual C++ environment and wizards no longer support DAO (although the DAO classes are included and you can still use them). Microsoft recommends that you use OLE DB Templates or ODBC for new projects. You should only use DAO in maintaining existing applications.

Which set of MFC classes should you use? This depends on your needs:

- Use the ODBC classes if you are working strictly with ODBC data sources, particularly in client/server situations, where the MFC ODBC classes provide better performance.
- Use the DAO classes if you are working primarily with Microsoft Jet (.mdb) databases or with other database formats that the Microsoft Jet database engine can read directly. For a list of these, see What Databases Can I Access with DAO and ODBC?
- Access ODBC data sources through the DAO classes when you want the speed of the Microsoft Jet database engine and the extra functionality of the DAO classes.

Note
DAO requires additional hard disk space.

The DAO classes have the following advantages:

- Better performance in some cases, particularly when using Microsoft Jet (.mdb) databases.
- Compatibility with the ODBC classes and with Microsoft Access Basic and Microsoft Visual Basic.
- Access to validation rules.
- Ability to specify relations between tables.
- A richer data access model, with support for Data Definition Language (DDL) and Data Manipulation Language (DML). For more information, see Database Definition and Manipulation.

The following table summarizes the key differences to help you choose.

Choosing Between MFC DAO and ODBC Classes

Can I	With DAO classes?	With ODBC classes?
Access .MDB files	Yes	Yes
Access ODBC data sources	Yes	Yes
Available for 16 Bit	No	Yes
Available for 32 Bit	Yes	Yes
Available for 64 Bit	No	Yes
Database compaction	Yes	No
Database engine support	Microsoft Jet database engine	Target DBMS
DDL support	Yes	Only through direct ODBC calls
DML support	Yes	Yes

Nature of the MFC implementation	"Wrapper" of DAO core functions	Simplified abstraction rather than a "wrapper" of the ODBC API
Optimal for	.mdb files (Microsoft Access)	Any DBMS for which you have a driver, especially in client/server situations
Transaction support	Per solution, or for ODBC data, per database	Per database

Keep in mind that the capabilities of ODBC drivers vary. For more information, see the ODBC *Programmer's Reference* and the Help file for your ODBC driver.

See Also

Other Resources

Data Access Frequently Asked Questions

What Data Sources Can I Access with DAO and ODBC?

Both sets of MFC classes let you access a wide variety of data sources and make it possible to write applications that are independent of the data source.

Databases You Can Access with DAO

Using DAO and the MFC DAO classes, you can access the following sources of data:

- Databases using the Microsoft Jet database engine, created with Microsoft Access or Microsoft Visual Basic, versions 1.x, 2.x, and 3.0 of the database engine.
- Installable ISAM databases, including:
  - dBASE III, dBASE IV, and dBASE 5.0
  - Paradox, versions 3.x, 4.x, and 5.x
- Open Database Connectivity (ODBC) databases, including but not limited to Microsoft SQL Server, SYBASE SQL Server, and ORACLE Server. To access an ODBC database, you must have an appropriate ODBC driver for the database you want to access. For a list of ODBC drivers included in this version of Visual C++ and for information about obtaining additional drivers, see ODBC Driver List.
- Microsoft Excel, versions 3.0, 4.0, 5.0, and 7.0 worksheets.
- Lotus WKS, WK1, WK3, and WK4 spreadsheets.
- Text files.

DAO is best used with databases that the Microsoft Jet database engine can read, which includes all the above except ODBC data sources. Best performance is with Microsoft Jet (.mdb) databases. Attaching external tables, especially in ODBC data sources, to an .mdb database is more efficient than opening the external database directly through the MFC DAO classes without attaching.

Databases You Can Access with ODBC

Using ODBC and the MFC ODBC classes, you can access any data source, local or remote, for which the user of your application has an ODBC driver. 16-bit, 32-bit, and 64-bit ODBC drivers are available for a wide range of data sources. If you are working with a Microsoft Jet (.mdb) database, it is more efficient to use the DAO classes than the Microsoft Access ODBC driver.

See Also

Other Resources

Data Access Frequently Asked Questions

Can I Call DAO or ODBC Directly?

As usual in MFC, if you need finer control, you can call DAO or ODBC directly in addition to accessing them through the classes. MFC attempts to simplify programming for Windows, but it also stays out of your way if you need access to the underlying APIs.

See Also

Other Resources

Data Access Frequently Asked Questions

Are DDL and DML Supported?

The MFC DAO classes support two kinds of access to databases:

- **Data definition language (DDL)** You can create and delete databases, create and delete tables, define table fields and indexes, and take other actions that affect the structure of your database.
- **Data manipulation language (DML)** You can run queries, add, delete, and edit records, and otherwise manipulate the content of your database.

The MFC ODBC classes support only DML, but you can call ODBC API functions directly to carry out DDL tasks.

See Also

Other Resources

Data Access Frequently Asked Questions

Where Can I Find Microsoft Knowledge Base Articles on Database Topics?

To find Knowledge Base articles about MFC ODBC topics, search the Knowledge Base for the keyword MFCDATABASE.

To find Knowledge Base articles about MFC DAO topics, search the Knowledge Base for the keyword MFCDAO.

See Also

Other Resources

Data Access Frequently Asked Questions

OLE DB Programming

This home page is your starting point for information about the Microsoft OLE DB database technology and the OLE DB Template Library.

It is important to know that Microsoft provides several implementations of OLE DB. OLE DB is a set of COM interfaces that provide uniform access to data in diverse information sources and formats.

The OLE DB templates are C++ templates that make the high-performance OLE DB database technology easier to use by providing classes that implement many commonly used OLE DB interfaces. This template library is divided into consumer templates and provider templates.

Visual C++ also has wizard support for creating OLE DB starter applications.

In addition, you can use attributes to implement the OLE DB consumer templates.

To learn more about	**See**
Using the OLE DB consumer templates (conceptual topics)	OLE DB Consumer Templates
Using the OLE DB provider templates (conceptual topics)	OLE DB Provider Templates
OLE DB templates classes and macros	OLE DB Templates Reference (Visual C++)
OLE DB consumer attributes	OLE DB Consumer Attributes
OLE DB interfaces	OLE DB programmer's reference (in the *Platform SDK*)
OLE DB templates samples	OLE DB Templates Samples
Data access programming overview (Visual C++)	Data Access Programming
ODBC vs. DAO	Should I Use DAO or ODBC?
ODBC conceptual topics	Open Database Connectivity (ODBC)
Data-bound controls (ADO and RDO) conceptual topics	Data-Bound Controls (ADO and RDO)

See Also

Other Resources

Data Access (C++)

OLE DB Programming Overview

What is OLE DB, and what makes it distinct from other database technologies? OLE DB is a high-performance, COM-based database technology produced by Microsoft. What sets OLE DB apart from Microsoft's other database technologies is how it provides Universal Data Access.

Universal Data Access

Universal Data Access provides a common way to access data regardless of the form in which it is stored. In a typical business situation, a vast amount of information is stored outside of corporate databases. This information is found in file systems (such as FAT or NTFS), indexed-sequential files, personal databases (such as Access), spreadsheets (such as Excel), project planning applications (such as Project), and e-mail (such as Outlook).

Accessing this data using the various associated applications presents a major bottleneck in workflow or at least an annoyance. Most companies find themselves in this situation and deal with the problem by consolidating the information in a database management system (DBMS). However, such a move is expensive, time consuming, and in many cases not practical.

The alternative is to develop a Universal Data Access solution. OLE DB and ADO provide Universal Data Access capability. Of the two, OLE DB is the more performance intensive and is recommended for use with Visual C++ applications.

Universal Data Access implies two capabilities: the first is distributed query or uniform access to multiple (distributed) data sources and the second is the ability to make non-DBMS data sources accessible to database applications.

- Distributed query

  The ability to access data uniformly on multiple (that is, distributed) data sources. The data sources can be of the same type (such as two separate Access databases) or of different types (such as a SQL Server database and an Access database). Uniformly means that you can meaningfully run the same query on all data sources.

- Non-DBMS access

  The ability to make non-DBMS data sources accessible to database applications. Examples of DBMS data sources include IMS, DB2, Oracle, SQL Server, Access, and Paradox. Examples of non-DBMS data sources include information in file systems, e-mail, spreadsheets, and project management tools.

Consider a scenario in which a sales department needs to find all e-mail messages received within a one-week period from customers in a certain area. This query might require a search on an e-mail application's mailbox file and a search on an Access table of customers to specify the customers' names. While Access is a DBMS application, Outlook is not.

OLE DB allows you to develop applications that access diverse data sources, whether they are DBMS or not. OLE DB makes universal access possible by using COM interfaces that support the appropriate DBMS functionality for a given data source. COM reduces unnecessary duplication of services and maximized interoperability not only among data sources but also among other applications.

Benefits of COM

This is where COM comes in. OLE DB is a set of COM interfaces. By accessing data through a uniform set of interfaces, you can organize a database into a matrix of cooperating components.

Based on the COM specification, OLE DB defines an extensible and maintainable collection of interfaces that factor and encapsulate consistent, reusable portions of DBMS functionality. These interfaces define the boundaries of DBMS components such as row containers, query processors, and transaction coordinators, which enable uniform transactional access to diverse information sources.

Typically, OLE DB applications are written as DLLs, but its COM implementation overcomes the disadvantages of DLLs (such as naming and version problems) by using componentized code. In OLE DB, you call interfaces or access other components using their globally unique identifiers (GUIDs).

Finally, COM keeps track of component usage by using reference counting. When you call a method on an interface, the reference count is incremented; when the method returns, the reference count is decremented. When the count equals zero, the component to which the method belongs is released.

See Also

Concepts

OLE DB Programming

OLE DB Consumer Templates (C++)
OLE DB Provider Templates (C++)

Other Resources

OLE DB Templates

OLE DB Consumers and Providers

The OLE DB architecture uses the model of consumers and providers. A consumer makes requests for data. A provider responds to these requests by placing data in a tabular format and returning it to the consumer. Any call that the consumer can make must be implemented in the provider.

Technically defined, a consumer is any system or application code (not necessarily an OLE DB component) that accesses data through OLE DB interfaces. The interfaces are implemented in a provider. Thus, a provider is any software component that implements OLE DB interfaces to encapsulate access to data and expose it to other objects (that is, consumers).

In terms of roles, a consumer calls methods on OLE DB interfaces; an OLE DB provider implements the needed OLE DB interfaces.

OLE DB avoids the terms client and server because these roles do not always make sense, especially in an n-tier situation. Because a consumer could be a component on a tier that serves another component, to call it a client component would be confusing. Also, a provider sometimes acts more like a database driver than a server.

See Also

Concepts

OLE DB Programming
OLE DB Programming Overview

OLE DB Object Model

The OLE DB object model comprises the following objects or components. The first four objects or components listed (data sources, sessions, commands, and rowsets) allow you to connect to a data source and view it. The rest, starting with accessors, relate to working with the data when it is displayed.

Data Sources

Data source objects allow you to connect to a data source such as a file or DBMS. A data source object creates and manages the connection and contains permissions and authentications information (such as login name and password). A data source object can create one or more sessions.

Sessions

A session manages a particular interaction with the data source to query and retrieve data. Each session is a single transaction. A transaction is an indivisible work unit defined by the ACID test. For a definition of ACID, see Transactions.

Sessions perform important tasks such as opening rowsets and returning the data source object that created it. Sessions can also return metadata, or information about the data source itself (such as table information).

A session can create one or more commands.

Commands

Commands execute a text command such as a SQL statement. If the text command specifies a rowset, such as a SQL **SELECT** statement, the command creates the rowset.

A command is simply a container for a text command, which is a string (such as a SQL statement) passed from a consumer to a data source object for execution by the provider's underlying data store. Typically, the text command is a SQL **SELECT** statement (in which case, because SQL **SELECT** specifies a rowset, the command automatically creates a rowset).

Rowsets

Rowsets expose data in tabular format. An index is a special case of a rowset. You can create rowsets from the session or the command.

Schema Rowsets

Schemas contain metadata (structural information) about a database. Schema rowsets are rowsets that contain schema information. Some OLE DB providers for DBMS support schema rowset objects. For more information about schema rowsets, see Obtaining Metadata with Schema Rowsets and Schema Rowset Classes and Typedef Classes.

View Objects

A view object defines a subset of the rows and columns from a rowset. It contains no data of its own. View objects cannot combine data from multiple rowsets.

Accessors

Only OLE DB uses the concept of accessors. An accessor describes how data is stored in a consumer. It contains a set of bindings (called a column map) between rowset fields (columns) and data members that you declare in the consumer.

Transactions

Transaction objects are used when committing or aborting nested transactions at other than the lowest level. A transaction is an indivisible work unit defined by the ACID test. ACID stands for:

- Atomicity: cannot be divided into smaller work units.
- Concurrency: more than one transaction can occur at a time.
- Isolation: one transaction has limited knowledge about changes made by another.
- Durability: the transaction makes persistent changes.

Enumerators

Enumerators search for available data sources and other enumerators. Consumers that are not customized for a particular data

source use enumerators to search for a data source to use.

A root enumerator, shipped in the Microsoft Data Access SDK, traverses the registry looking for data sources and other enumerators. Other enumerators traverse the registry or search in a provider-specific manner.

Errors

Any interface on any OLE DB object can generate errors. Errors contain additional information about an error, including an optional custom error object. This information is contained in an HRESULT.

Notifications

Notifications are used by groups of cooperating consumers sharing a rowset (where sharing means that the consumers are assumed to be working within the same transaction). Notifications enable cooperating consumers sharing a rowset to be informed about actions on the rowset performed by their peers.

See Also

Concepts

OLE DB Programming
OLE DB Programming Overview

OLE DB Templates, Attributes, and Other Implementations

ATL OLE DB Templates

The OLE DB Templates, which are part of ATL (Active Template Library), make the high-performance OLE DB database technology easier to use by providing classes that implement many of the commonly used OLE DB interfaces. Along with this template library comes wizard support for creating OLE DB starter applications.

This template library contains two parts:

- **OLE DB Consumer Templates** Used to implement an OLE DB client (consumer) application.
- **OLE DB Provider Templates** Used to implement an OLE DB server (provider) application.

To use the OLE DB Templates, you should be familiar with C++ templates, COM, and the OLE DB interfaces. If you are not familiar with OLE DB, see OLE DB Programmer's Reference.

For more information, you can:

- Read the topics about the OLE DB Consumer Templates or OLE DB Provider Templates.
- Create an OLE DB consumer or OLE DB provider.
- See the list of OLE DB consumer classes or OLE DB provider classes.
- See the list of OLE DB templates samples.
- See OLE DB Programmer's Reference (in the *Platform SDK*).

OLE DB Attributes

The OLE DB consumer attributes provide a convenient way to create OLE DB consumers. The OLE DB attributes inject code based on the OLE DB consumer templates to create working OLE DB consumers and providers. If you need to specify functionality not supported by the attributes, you can use the OLE DB Templates in conjunction with attributes in your code.

MFC OLE DB Classes

The MFC library has one class, COleDBRecordView, that displays database records in controls. The view is a form view directly connected to a **CRowset** object and displays the fields of the **CRowset** object in the dialog template's controls. It also supplies a default implementation for moving to the first, next, previous, or last record and an interface for updating the record currently on view. For more information, see **COleDBRecordView**.

OLE DB SDK Interfaces

In the cases where the OLE DB Templates do not support OLE DB functionality, you need to use the OLE DB interfaces themselves. For more information, see OLE DB Programmer's Reference in the *Platform SDK*.

See Also

Concepts

OLE DB Programming
OLE DB Programming Overview

OLE DB Architectural Design Issues

You should consider the following issues before starting your OLE DB application:

What programming implementation will you use to write your OLE DB application?

Microsoft offers several libraries to accomplish this: an OLE DB Template library, OLE DB attributes, and the raw OLE DB interfaces in the OLE DB SDK. In addition, there are wizards that help you write your program. These implementations are described in OLE DB Templates, Attributes, and Other Implementations.

Do you need to write your own provider?

Most developers do not need to write their own provider. Microsoft provides several providers. Whenever you create a data connection (for example, when you add a consumer to your project using the ATL OLE DB Consumer Wizard), the **Data Link Properties** dialog box lists all the available providers registered on your system. If one of these providers is appropriate for your own data store and data access application, the easiest thing to do is use one of these. However, if your data store does not fit one of these categories, you have to create your own provider. For information about creating providers, see OLE DB Provider Templates.

What level of support do you need for your consumer?

Some consumers can be very basic; while others can be very complex. The functionality of OLE DB objects is specified by properties. When you use the ATL OLE DB Consumer Wizard to create a consumer or the Database Provider Wizard to create a provider, it sets the appropriate object properties for you to give you a standard set of functionalities. However, if the wizard-generated consumer or provider classes do not support everything you need them to do, you need to refer to the interfaces for those classes in the OLE DB Templates Library. These interfaces wrap the raw OLE DB interfaces, providing extra implementation to make using them easier for you.

For example, if you want to update data in a rowset, but forgot to specify this when you created the consumer with the wizard, you can specify the functionality after the fact by setting the **DBPROP_IRowsetChange** and **DBPROP_UPDATABILITY** properties on the command object. Then, when the rowset is created, it has the **IRowsetChange** interface.

Do you have legacy code using another data access technology (ADO, ODBC, or DAO)?

Given the possible combinations of technologies (such as using ADO components with OLE DB components and migrating ODBC code to OLE DB), covering all situations is beyond the scope of the Visual C++ documentation. However, many articles covering various scenarios are available on the following Microsoft Web sites:

Knowledge Base articles

http://support.microsoft.com/search/default.asp

MSDN technical articles on data access

http://msdn.microsoft.com/visualc/technical/articles/data.asp

Visual Studio Solution Center

http://msdn.microsoft.com/vstudio/centers/solution.asp

Search Microsoft.com

http://search.microsoft.com/us/SearchMS.asp

When you perform a search, enter a combination of keywords that best fits your scenario; for example: if you were using ADO objects with an OLE DB provider, try a Boolean search with **ADO AND "OLE DB"**. If you wanted to migrate legacy DAO code to ODBC, select "all words" and specify strings such as **migrating DAO** or **DAO legacy**.

See Also

Concepts

OLE DB Programming
OLE DB Programming Overview

OLE DB Consumer Templates (C++)

The OLE DB Consumer Templates support the OLE DB version 2.6 specification. (The OLE DB Consumer Templates are tested against OLE DB 2.6 but do not support every interface in the specification.) The Consumer Templates minimize the amount of code you must write to implement an OLE DB consumer. The templates provide:

- Easy access to OLE DB features and easy integration with ATL and MFC.
- An easy binding model for database parameters and columns.
- Native C/C++ data types for OLE DB programming.

To use the OLE DB Templates, you should be familiar with C++ templates, COM, and the OLE DB interfaces. If you are not familiar with OLE DB, see OLE DB Programmer's Reference.

The OLE DB Templates support the existing OLE DB object model rather than adding a new object model. The top-layer classes in the OLE DB Consumer Templates parallel the components defined in the OLE DB specification. The design of the OLE DB Consumer Templates includes advanced features such as multiple accessors on a rowset. The use of templates and multiple inheritance makes the library small and flexible.

How OLE DB Consumers Access Data

Consumers use several kinds of objects, which are described in the following topics:

- Data Sources and Sessions
- Accessors and Rowsets
- Commands and Tables
- User Records

Before the consumer does anything, you first select an OLE DB provider that is appropriate for the type of database you need to access (for example, SQL, Oracle, ODBC, and MSDS). To do this, you typically use an enumerator (see CEnumerator as mentioned in Data Sources and Sessions).

The data source object provides the connection information necessary to connect to the data source you selected. The data source object also contains authentication information (such as login names and passwords), which is used to give users permission to access the data source. The data source object makes a connection to the database and then creates one or more session objects. Each session object manages its own interactions with the database (that is, querying and retrieving data) and performs these transactions independently of other existing sessions.

The session creates the rowset and command objects. The command object allows users to interact with the database, for example, using SQL commands. The rowset object is a set of data through which you can navigate and in which you can update, delete, and insert rows.

An OLE DB consumer binds columns in the database tables with local variables; to do this, it uses an accessor, which contains a map of how data is stored within the consumer. The map consists of a set of bindings between table columns and local buffers (variables) in the consumer application.

One important concept when working with consumers is that you declare two classes in a consumer: the command (or table) class and the user record class. You access the rowset through the command (or table) class, which inherits from both an accessor class and a rowset class. The user record class contains the rowset binding map previously described.

For more information, see the following topics:

- OLE DB Consumer Templates Architecture
- Creating an OLE DB Consumer
- Common OLE DB Consumer Scenarios (Examples)

See Also

Reference

OLE DB SDK Documentation
OLE DB Programmer's Reference

Concepts

OLE DB Programming

Other Resources

Data Access (C++)

Data Sources and Sessions

The following figure shows the classes that support connecting to and accessing a data source. Each class is based on a standard OLE DB component implementation, as described in the OLE DB Reference.

Data Source and Session Classes

The classes are:

- CDataSource This class instantiates the data source object, which creates and manages a connection to a data source through an OLE DB provider. The data source takes information such as the data source address and authentication information in the form of a connection string.

  It is also worth noting that the helper class CEnumerator is often used before any connection is established to obtain a list of available providers registered on a system. This allows you to select a provider as a data source. For example, the **Data Link Properties** dialog box uses this class to populate the list of providers on the **Providers** tab. It is equivalent to the **SQLBrowseConnect** or **SQLDriverConnect** function.

- CSession This class instantiates the session object, which represents a single access session to the data source. However, you can create multiple sessions on a data source. For each session, you can create rowsets, commands, and other objects to access data from the data source. The session handles transactions.

See Also

Concepts

OLE DB Consumer Templates (C++)

Accessors and Rowsets

To set and retrieve data, OLE DB Templates use an accessor and a rowset through the CAccessorRowset class. This class can handle multiple accessors of different types.

Accessor Types

All accessors derive from CAccessorBase. **CAccessorBase** provides both parameter and column binding.

The following figure shows the accessor types.

Accessor Classes

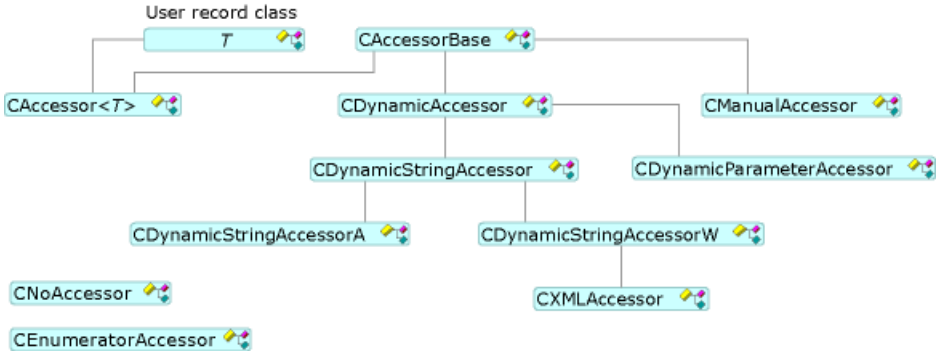

- CAccessor Use this accessor when you know the structure of the database source at design time. **CAccessor** statically binds a database record, which contains the buffer, to the data source.
- CDynamicAccessor Use this accessor when you do not know the structure of the database at design time. **CDynamicAccessor** calls IColumnsInfo::GetColumnInfo to get the database column information. It creates and manages an accessor and the buffer.
- CDynamicParameterAccessor Use this accessor to handle unknown command types. When you prepare the commands, **CDynamicParameterAccessor** can get parameter information from the ICommandWithParameters interface, if the provider supports **ICommandWithParameters**.
- CDynamicStringAccessor, CDynamicStringAccessorA, and CDynamicStringAccessorW Use these classes when you have no knowledge of the database schema. **CDynamicStringAccessorA** retrieves data as ANSI strings; **CDynamicStringAccessorW** retrieves data as Unicode strings.
- CManualAccessor With this class, you can use whatever data types you want if the provider can convert the type. It handles both result columns and command parameters.

The following table summarizes the support in the OLE DB Template accessor types.

Accessor type	Dynamic	Handles params	Buffer	Multiple accessors
CAccessor	No	Yes	User	Yes
CDynamicAccessor	Yes	No	OLE DB Templates	No
CDynamicParameterAccessor	Yes	Yes	OLE DB Templates	No
CDynamicStringAccessor[A,W]	Yes	No	OLE DB Templates	No
CManualAccessor	Yes	Yes	User	Yes

Rowset Types

The OLE DB Templates support three kinds of rowsets (see the preceding figure): single rowsets (implemented by CRowset), bulk rowsets (implemented by CBulkRowset), and array rowsets (implemented by CArrayRowset). Single rowsets fetch a single row handle when **MoveNext** is called. Bulk rowsets can fetch multiple row handles. Array rowsets are rowsets that can be accessed using array syntax.

The following figure shows the rowset types.

Rowset Classes

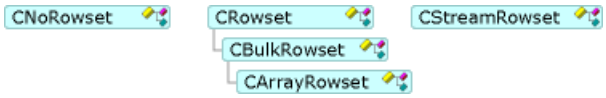

Schema rowsets do not access data in the data store but instead access information about the data store, called metadata. Schema rowsets are typically used in situations in which the database structure is not known at compile time and must be obtained at run time.

See Also

Concepts

OLE DB Consumer Templates (C++)

Commands and Tables

Commands and tables allow you to access rowsets; that is, open rowsets, execute commands, and bind columns. The CCommand and CTable classes instantiate the command and table objects, respectively. These classes derive from CAccessorRowset as shown in the following figure.

Command and Table Classes

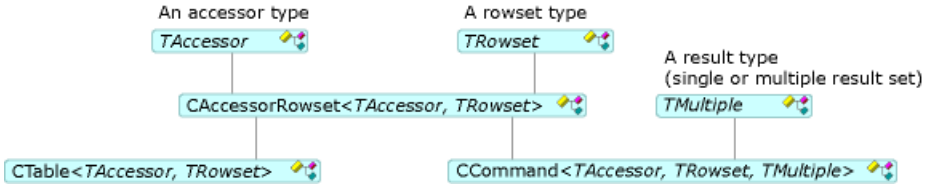

In the previous table, *TAccessor* can be any accessor type listed in Accessor Types. *TRowset* can be any rowset type listed in Rowset Types. *TMultiple* specifies the result type (a single or multiple result set).

The ATL OLE DB Consumer Wizard lets you specify whether you want a command or table object.

- For data sources without commands, you can use the **CTable** class. You generally use it for simple rowsets that specify no parameters and require no multiple results. This simple class opens a table on a data source using a table name that you specify.
- For data sources that support commands, you can use the **CCommand** class instead. To execute a command, call Open on this class. As an alternative, you can call **Prepare** to prepare a command that you want to execute more than once.

  CCommand has three template arguments: an accessor type, a rowset type, and a result type (**CNoMultipleResults**, by default, or **CMultipleResults**). If you specify **CMultipleResults**, the **CCommand** class supports the **IMultipleResults** interface and handles multiple rowsets. The DBVIEWER sample shows how to handle the multiple results.

See Also

Concepts

OLE DB Consumer Templates (C++)

User Records

To use a static accessor (that is, an accessor derived from **CAccessor)**, your consumer must have a user record. The user record is a C++ class that contains data elements to handle input or output. The ATL OLE DB Consumer Wizard generates a user record for your consumer. You can add methods to the user record for optional tasks like handling commands.

The following code shows a sample record that handles commands. In the user record, **BEGIN_COLUMN_MAP** represents a data rowset passed to the consumer from a provider. **BEGIN_PARAM_MAP** represents a set of command parameters. This example uses a CCommand class to handle the command parameters. The data members in the map entries represent offsets into one contiguous block of memory for each instance of the class. The **COLUMN_ENTRY** macros correspond to the **PROVIDER_COLUMN_ENTRY** macros on the provider side.

For more information about the **COLUMN_MAP** and **PARAM_MAP** macros, see Macros for OLE DB Consumer Templates.

```
class CArtists
{
public:
// Data Elements
   CHAR m_szFirstName[20];
   CHAR m_szLastName[30];
   short m_nAge;

// Column binding map
BEGIN_COLUMN_MAP(CArtists)
   COLUMN_ENTRY(1, m_szFirstName)
   COLUMN_ENTRY(2, m_szLastName)
   COLUMN_ENTRY(3, m_nAge)
END_COLUMN_MAP()

// Parameter binding map
BEGIN_PARAM_MAP(CArtists)
   COLUMN_ENTRY(1, m_nAge)
END_PARAM_MAP()
};
```

Wizard-Generated User Records

If you use the ATL OLE DB Consumer Wizard to generate a consumer, you have the choice of using OLE DB Templates or OLE DB Attributes. The generated code is different in each case. For more information about this code, see Consumer Wizard-Generated Classes.

User Record Support for Multiple Accessors

For a detailed discussion of the scenarios in which you need to use multiple accessors, see Using Multiple Accessors on a Rowset.

The following example shows the user record modified to support multiple accessors on the rowset. Instead of **BEGIN_COLUMN_MAP** and **END_COLUMN_MAP**, it uses BEGIN_ACCESSOR_MAP and BEGIN_ACCESSOR for each accessor. The **BEGIN_ACCESSOR** macro specifies the accessor number (offset from zero) and whether the accessor is an autoaccessor. Autoaccessors call **GetData** to retrieve data automatically on a call to MoveNext. Nonautomatic accessors require you to explicitly retrieve the data. Use a nonautomatic accessor if you are binding to a large data field (like a bitmap image) that you might not want to retrieve for every record.

```
class CMultiArtists
{
public:
// Data Elements
   CHAR m_szFirstName[20];
   CHAR m_szLastName[30];
   short m_nAge;

// Column binding map
BEGIN_ACCESSOR_MAP(CMultiArtists, 2)
   BEGIN_ACCESSOR(0, true)    // true specifies an auto accessor
```

```
     COLUMN_ENTRY(1, m_szFirstName)
     COLUMN_ENTRY(2, m_szLastName)
    END_ACCESSOR()
    BEGIN_ACCESSOR(1, false)   // false specifies a manual accessor
     COLUMN_ENTRY(3, m_nAge)
    END_ACCESSOR()
  END_ACCESSOR_MAP()
  };
```

See Also

Concepts

OLE DB Consumer Templates (C++)

Schema Rowsets

Schema (or metadata) is any information about a database's structure or organization, such as information on the provider, rowset, table, columns, or other information aside from the actual database content. OLE DB allows you to obtain such information without opening the rowset by using schema rowsets.

For information about how to use schema rowsets, see Obtaining Metadata with Schema Rowsets.

See Also

Concepts

OLE DB Consumer Templates (C++)

Creating an OLE DB Consumer

You can create an OLE DB Templates consumer with or without the ATL OLE DB Consumer Wizard, though using the wizard is the recommended method. A consumer can operate in any application that supports ATL.

This section covers the following topics:

- Creating an OLE DB Consumer Using a Wizard
- Creating an OLE DB Consumer Without Using a Wizard

See Also

Concepts

OLE DB Consumer Templates (C++)

Creating an OLE DB Consumer Using a Wizard

This procedure demonstrates how you can use the ATL Project Wizard and ATL OLE DB Consumer Wizard to generate an OLE DB Templates consumer and then modify the main code of the console application to retrieve and display data from a database table.

The last two sections explain the wizard-generated OLE DB consumer code.

This section covers the following topics:

- Creating a Simple Consumer
- Implementing a Simple Consumer
- Consumer Wizard-Generated Classes
- Consumer Wizard-Generated Methods

See Also

Reference

Creating an OLE DB Consumer

Creating a Simple Consumer

Use the ATL Project Wizard and ATL OLE DB Consumer Wizard to generate an OLE DB Templates consumer.

To create a console application for an OLE DB consumer

1. On the **File** menu, click **New**, and then click **Project**.

   The **New Project** dialog box appears.

2. In the Project Types pane, click the **Visual C++ Projects** folder, and then click the **Win32 Project** icon in the Templates pane. In the **Name** box, enter the name of your project, for example, **MyCons**.

3. Click **OK**.

   The Win32 Project Wizard appears.

4. On the **Application Settings** page, select **Console application**, and then select **Add support for ATL**.

5. Click **Finish** to close the wizard and generate the project.

Next, use the ATL OLE DB Consumer Wizard to add an OLE DB consumer object.

To create a consumer with the ATL OLE DB Consumer Wizard

1. In Class View, right-click the `MyCons` project.

2. On the shortcut menu, click **Add**, and then click **Add Class**.

   The **Add Class** dialog box appears.

3. In the Categories pane, click **Visual C++**, click the **ATL OLE DB Consumer** icon in the Templates pane, and then click **Open**.

   The ATL OLE DB Consumer Wizard appears.

4. Click the **Data Source** button.

   The **Data Link Properties** dialog box appears.

5. In the **Data Link Properties** dialog box, do the following:

   - On the **Provider** tab, specify an OLE DB provider.
   - On the **Connection** tab, specify the server name, logon ID, and password for your data source and database on the server.

   Note

   There is a security issue with the **Allow saving of password** feature of the **Data Link Properties** dialog box. In **Enter information to log on to the server**, there are two radio buttons: **Use Windows NT integrated security** and **Use a specific user name and password**.

   Note

   If you select **Use a specific user name and password**, you have the option of saving the password (using the **Allow saving password** check box); however, this option is not secure. It is recommended that you select **Use Windows NT integrated security**; this option uses Windows NT to verify your identity.

   Note

   If you cannot use Windows NT integrated security, you should use a middle-tier application to prompt the user for the password or to store the password in a location with security mechanisms to help protect it (instead of in source code).

   After selecting your provider and other settings, click **Test Connection** to verify the selections made on the previous

dialog box pages. If the **Results** box reports `Test connection succeeded`, click **OK** to create the data link.

The **Select Database Object** dialog box appears.

6. Use the tree control to select a table, view, or stored procedure. For the purpose of this procedure, select the Products table from the Northwind database.
7. Click **OK**. This returns you to the ATL OLE DB Consumer Wizard.
8. The wizard completes the names for **Class** and **.h file** based on the name of the table, view, or stored procedure that you selected. You can edit these names if you want.
9. Clear the **Attributed** check box so that the wizard creates the consumer code using OLE DB Template classes instead of the default OLE DB consumer attributes.
10. Under **Type**, select **Command**.

    The wizard creates a CCommand-based consumer if you select **Command** or a CTable-based consumer if you select **Table**. The table or command class is named after the selected object, but you can edit the name.
11. Under **Support**, leave the **Change**, **Insert**, and **Delete** boxes cleared.

    Select the **Change**, **Insert**, and **Delete** check boxes to support the changing, inserting, and deleting of records in the rowset, if required. For more information about writing data to the data store, see Updating Rowsets.
12. Click **Finish** to create the consumer.

The wizard generates a command class and a user record class, as shown in Consumer Wizard-Generated Classes. The command class will have the name that you entered in the **Class** box in the wizard (in this case, `CProducts`), and the user record class will have a name of the form "*ClassName*Accessor" (in this case, `CProductsAccessor`).

Note

The wizard puts the following line into Products.h:

```
#error Security Issue: The connection string may contain a password
```

Note

This line prevents the consumer application from compiling and reminds you to check your connection string for hard-coded passwords. After checking your connection string, you can remove this line of code.

See Also

Concepts

Creating an OLE DB Consumer Using a Wizard

Implementing a Simple Consumer

The following topics show how to edit the files created by the MFC Application Wizard and ATL OLE DB Consumer Wizard to create a simple consumer. This example has the following parts:

- "Retrieving Data with the Consumer" shows how to implement code in the consumer that reads all the data, row by row, from a database table.
- "Adding Bookmark Support to the Consumer" shows how to add bookmark support to the consumer.
- "Adding XML Support to the Consumer" shows how to modify the consumer code to output the retrieved rowset data as XML data.

Note

You can use the consumer application described in this section to test the MyProv and Provider sample providers.

Note

To build a consumer application to test MyProv (the same provider described in Enhancing the Simple Read-Only Provider), you must include bookmark support as described in "Adding Bookmark Support to the Consumer."

Note

To build a consumer application to test Provider, leave out the bookmark support described in "Adding Bookmark Support to the Consumer" and skip to "Adding XML Support to the Consumer."

Retrieving Data with the Consumer

To modify the console application to use the OLE DB consumer

- In MyCons.cpp, change the main code by inserting the bold text as follows:

```
// MyCons.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
#include "Products.h"
...
int main(int argc, char* argv[])
{
   HRESULT hr = CoInitialize(NULL);

   // Instantiate rowset
   CProducts rs;

   hr = rs.OpenAll();
   ATLASSERT( SUCCEEDED( hr ) );
   hr = rs.MoveFirst();

   // Iterate through the rowset
   while( SUCCEEDED(hr) && hr != DB_S_ENDOFROWSET )
   {
      // Print out the column information for each row
      printf("Product ID: %d, Name: %s, Unit Price: %d, Quantity per Unit: %d, Units i
n Stock %d, Reorder Level %d\n",
            rs.m_ProductID, rs.m_ProductName, rs.m_UnitPrice, rs.m_QuantityPerUnit, rs
.m_UnitsInStock, rs.m_ReorderLevel );
      hr = rs.MoveNext();
```

```
    }

    rs.Close();
    rs.ReleaseCommand();

    CoUninitialize();

    return 0;
}
```

Adding Bookmark Support to the Consumer

A bookmark is a column that uniquely identifies rows in the table. Typically it is the key column, but not always; it is provider-specific. This section shows you how to add bookmark support. To do so, you need to do the following in the user record class:

- Instantiate the bookmarks. These are objects of type CBookmark.
- Request a bookmark column from the provider by setting the **DBPROP_IRowsetLocate** property.
- Add a bookmark entry to the column map by using the BOOKMARK_ENTRY macro.

The previous steps give you bookmark support and a bookmark object with which to work. This code example demonstrates a bookmark as follows:

- Open a file for writing.
- Output rowset data to the file row by row.
- Move the rowset cursor to the bookmark by calling MoveToBookmark.
- Output the bookmarked row, appending it to the end of the file.

Note
If you use this consumer application to test the Provider sample provider application, leave out the bookmark support described in this section.

To instantiate the bookmark

- The accessor needs to contain an object of type CBookmark. The *nSize* parameter specifies the size of the bookmark buffer in bytes (typically 4 for 32-bit platforms and 8 for 64-bit platforms). Add the following declaration to the column data members in the user record class:

```
////////////////////////////////////////////////////////////////////
// Products.h
class CProductsAccessor
{
public:
    CBookmark<4> m_bookmark;    // Add bookmark declaration
    LONG m_ProductID;
    ...
```

To request a bookmark column from the provider

- Add the following code in the `GetRowsetProperties` method in the user record class:

```
// Set the DBPROP_IRowsetLocate property.
void GetRowsetProperties(CDBPropSet* pPropSet)
{
    pPropSet->AddProperty(DBPROP_CANFETCHBACKWARDS, true, DBPROPOPTIONS_OPTIONAL);
    pPropSet->AddProperty(DBPROP_CANSCROLLBACKWARDS, true, DBPROPOPTIONS_OPTIONAL);
```

```
   // Add DBPROP_IRowsetLocate property to support bookmarks
   pPropSet->AddProperty(DBPROP_IRowsetLocate, true);
}
```

To add a bookmark entry to the column map

- Add the following entry to the column map in the user record class:

```
// Set a bookmark entry in the column map.
BEGIN_COLUMN_MAP(CProductsAccessor)
   BOOKMARK_ENTRY(m_bookmark)   // Add bookmark entry
   COLUMN_ENTRY_LENGTH_STATUS(1, m_ProductID, m_dwProductIDLength, m_dwProductIDStatus
)
   COLUMN_ENTRY_LENGTH_STATUS(2, m_ProductName, m_dwProductNameLength, m_dwProductName
Status)
...
END_COLUMN_MAP()
```

To use a bookmark in your main code

- In the MyCons.cpp file from the console application you previously created, change the main code to read as follows. To use bookmarks, the main code needs to instantiate its own bookmark object (`myBookmark`); this is a different bookmark from the one in the accessor (`m_bookmark`).

```
//////////////////////////////////////////////////////////////////////
// MyCons.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "Products.h"
#include <iostream>
#include <fstream>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
   HRESULT hr = CoInitialize(NULL);

   // Instantiate rowset
   CProducts rs;

   hr = rs.OpenAll();
   hr = rs.MoveFirst();

   // Cast CURRENCY m_UnitPrice to a long value
   LONGLONG lPrice = rs.m_UnitPrice.int64;

   // Open file output.txt for writing in overwrite mode
   ofstream outfile( "C:\\output.txt", ios::out );

   if (!outfile)      // Test for invalid file
      return -1;

   // Instantiate a bookmark object myBookmark for the main code
   CBookmark<4> myBookmark;
   int nCounter = 0;

   // Iterate through the rowset and output column data to output.txt row by row
```

```
   // In the file, mark the beginning of this set of data:
   outfile << "initial row dump" << endl;
   while( SUCCEEDED(hr) && hr != DB_S_ENDOFROWSET )
   {
      nCounter++;
      if( nCounter == 5 )
         myBookmark = rs.bookmark;
      // Output the column information for each row:
      outfile << rs.m_ProductID << rs.m_ProductName << lPrice << rs.m_QuantityPerUnit
<< rs.m_UnitsInStock << rs.m_ReorderLevel << endl;
      hr = rs.MoveNext();
   }

   // Move cursor to bookmark
   hr = rs.MoveToBookmark(myBookmark);

   // Iterate through the rowset and output column data to output.txt row by row
   // In the file, mark the beginning of this set of data:
   outfile << "row dump starting from bookmarked row" << endl;
   while( SUCCEEDED(hr) && hr != DB_S_ENDOFROWSET )
   {
      // Output the column information for each row
      outfile << rs.m_ProductID << rs.m_ProductName << lPrice << rs.m_QuantityPerUnit
<< rs.m_UnitsInStock << rs.m_ReorderLevel << endl;
      hr = rs.MoveNext();
   }

   rs.CloseAll();
   CoUninitialize();

   return 0;
}
```

For more information about bookmarks, see Using Bookmarks. Examples of bookmarks are also shown in Updating Rowsets.

Adding XML Support to the Consumer

As discussed in Accessing XML Data, there are two ways to retrieve XML data from a data source: using CStreamRowset or using CXMLAccessor. This example uses **CStreamRowset**, which is more efficient, but requires you to have SQL Server 2000 running on the computer on which you execute this sample application.

To modify the command class to inherit from CStreamRowset

- In the consumer application you previously created, change your **CCommand** declaration to specify **CStreamRowset** as the rowset class as follows:

```
class CProducts : public CCommand<CAccessor<CProductsAccessor>, CStreamRowset >
```

To modify the main code to retrieve and output the XML data

- In the MyCons.cpp file from the console application you previously created, change the main code to read as follows:

```
////////////////////////////////////////////////////////////////////////
// MyCons.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "Products.h"
#include <iostream>
```

```
#include <fstream>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
   HRESULT hr = CoInitialize(NULL);

   // Instantiate rowset
   CProducts rs;

   // Add variable declarations for the Read method to handle sequential stream data
   CHAR buffer[1001];  // Pointer to buffer into which data stream is read
   ULONG cbRead;       // Actual number of bytes read from the data stream

   hr = rs.OpenAll();

   // Open file output.txt for writing in overwrite mode
   ofstream outfile( "C:\\output.txt", ios::out );

   if (!outfile)      // Test for invalid file
      return -1;

   // The following loop reads 1000 bytes of the data stream at a time
   // until it reaches the end of the data stream
   for (;;)
   {
      // Read sequential stream data into buffer
      HRESULT hr = rs.m_spStream->Read(buffer, 1000, &cbRead);
      if (FAILED (hr))
         break;
      // Output buffer to file
      buffer[cbRead] = 0;
      outfile << buffer;
      // Test for end of data stream
      if (cbRead < 1000)
         break;
   }

   rs.CloseAll();
   CoUninitialize();

   return 0;
}
```

See Also

Concepts

Creating an OLE DB Consumer Using a Wizard

Consumer Wizard-Generated Classes

When you use the ATL OLE DB Consumer Wizard to generate a consumer, you have the choice of using OLE DB Templates or OLE DB attributes. In both cases, the wizard generates a command class and a user record class. The command class contains code to open the data source and rowset you specified in the wizard. The user record class contains a column map for the database table you selected. However, the generated code differs in each case:

- If you select a templated consumer, the wizard generates a command class and a user record class. The command class will have the name that you enter in the Class box in the wizard (for example, `CProducts`), and the user record class will have a name of the form "*ClassName*Accessor" (for example, `CProductsAccessor`). Both classes are placed in the consumer's header file.
- If you select an attributed consumer, the user record class will have a name of the form "_*ClassName*Accessor" and will be injected. That is, you will be able to view only the command class in the text editor; you can only view the user record class as injected code. For information about viewing injected code, see Debugging Injected Code.

The following examples use a command class created on the Products table of the Northwind database to demonstrate the wizard-generated consumer code for the command class and user record class.

Templated User Record Classes

If you create an OLE DB consumer using the OLE DB Templates (rather than the OLE DB attributes), the wizard generates code as described in this section.

Column Data Members

The first part of the user record class includes the data member declarations and the status and length data members for each data-bound column. For information about these data members, see Field Status Data Members in Wizard-Generated Accessors.

Note

If you modify the user record class or write your own consumer, the data variables must come before the status and length v ariables.

Note

The ATL OLE DB Consumer Wizard uses the **DB_NUMERIC** type to bind numeric data types. It formerly used **DBTYPE_VAR NUMERIC** (the format of which is described by the **DB_VARNUMERIC** type; see Oledb.h). If you do not use the wizard to cre ate consumers, it is recommended that you use **DB_NUMERIC**.

```
// Products.H : Declaration of the CProducts class

class CProductsAccessor
{
public:
   // Column data members:
   LONG m_ProductID;
   TCHAR m_ProductName[41];
   LONG m_SupplierID;
   LONG m_CategoryID;
   TCHAR m_QuantityPerUnit[21];
   CURRENCY m_UnitPrice;
   SHORT m_UnitsInStock;
   SHORT m_UnitsOnOrder;
   SHORT m_ReorderLevel;
   VARIANT_BOOL m_Discontinued;

   // Column status data members:
   DBSTATUS m_dwProductIDStatus;
   DBSTATUS m_dwProductNameStatus;
   DBSTATUS m_dwSupplierIDStatus;
   DBSTATUS m_dwCategoryIDStatus;
```

```
    DBSTATUS m_dwQuantityPerUnitStatus;
    DBSTATUS m_dwUnitPriceStatus;
    DBSTATUS m_dwUnitsInStockStatus;
    DBSTATUS m_dwUnitsOnOrderStatus;
    DBSTATUS m_dwReorderLevelStatus;
    DBSTATUS m_dwDiscontinuedStatus;

    // Column length data members:
    DBLENGTH m_dwProductIDLength;
    DBLENGTH m_dwProductNameLength;
    DBLENGTH m_dwSupplierIDLength;
    DBLENGTH m_dwCategoryIDLength;
    DBLENGTH m_dwQuantityPerUnitLength;
    DBLENGTH m_dwUnitPriceLength;
    DBLENGTH m_dwUnitsInStockLength;
    DBLENGTH m_dwUnitsOnOrderLength;
    DBLENGTH m_dwReorderLevelLength;
    DBLENGTH m_dwDiscontinuedLength;
```

Rowset Properties

Next, the wizard sets rowset properties. If you selected **Change**, **Insert**, or **Delete** in the ATL OLE DB Consumer Wizard, the appropriate properties are set here (DBPROP_IRowsetChange is always set, then one or more of DBPROPVAL_UP_CHANGE, DBPROPVAL_UP_INSERT, and/or DBPROPVAL_UP_DELETE, respectively).

```
    void GetRowsetProperties(CDBPropSet* pPropSet)
    {
       pPropSet->AddProperty(DBPROP_CANFETCHBACKWARDS, true, DBPROPOPTIONS_OPTIONAL);
       pPropSet->AddProperty(DBPROP_CANSCROLLBACKWARDS, true, DBPROPOPTIONS_OPTIONAL);
       pPropSet->AddProperty(DBPROP_IRowsetChange, true, DBPROPOPTIONS_OPTIONAL);
       pPropSet->AddProperty(DBPROP_UPDATABILITY, DBPROPVAL_UP_CHANGE | DBPROPVAL_UP_INSERT
 | DBPROPVAL_UP_DELETE);
    }
```

Command or Table Class

If you specify a command class, the wizard declares the command class; for templated code, the command looks like this:

```
    DEFINE_COMMAND_EX(CProductsAccessor, L" \
    SELECT \
       ProductID, \
       ProductName, \
       SupplierID, \
       CategoryID, \
       QuantityPerUnit, \
       UnitPrice, \
       UnitsInStock, \
       UnitsOnOrder, \
       ReorderLevel, \
       Discontinued \
       FROM dbo.Products")
```

Column Map

The wizard then generates the column bindings or column map. To fix several issues with some providers, the following code might bind columns in a different order than that reported by the provider.

```
    BEGIN_COLUMN_MAP(CProductsAccessor)
       COLUMN_ENTRY_LENGTH_STATUS(1, m_ProductID, m_dwProductIDLength, m_dwProductIDStatus)
       COLUMN_ENTRY_LENGTH_STATUS(2, m_ProductName, m_dwProductNameLength, m_dwProductNameSt
atus)
       COLUMN_ENTRY_LENGTH_STATUS(3, m_SupplierID, m_dwSupplierIDLength, m_dwSupplierIDStatu
s)
       COLUMN_ENTRY_LENGTH_STATUS(4, m_CategoryID, m_dwCategoryIDLength, m_dwCategoryIDStatu
s)
       COLUMN_ENTRY_LENGTH_STATUS(5, m_QuantityPerUnit, m_dwQuantityPerUnitLength, m_dwQuant
```

```
ityPerUnitStatus)
      _COLUMN_ENTRY_CODE(6, DBTYPE_CY, _SIZE_TYPE(m_UnitPrice), 0, 0, offsetbuf(m_UnitPrice
), offsetbuf(m_dwUnitPriceLength), offsetbuf(m_dwUnitPriceStatus))
      COLUMN_ENTRY_LENGTH_STATUS(7, m_UnitsInStock, m_dwUnitsInStockLength, m_dwUnitsInStoc
kStatus)
      COLUMN_ENTRY_LENGTH_STATUS(8, m_UnitsOnOrder, m_dwUnitsOnOrderLength, m_dwUnitsOnOrde
rStatus)
      COLUMN_ENTRY_LENGTH_STATUS(9, m_ReorderLevel, m_dwReorderLevelLength, m_dwReorderLeve
lStatus)
      _COLUMN_ENTRY_CODE(10, DBTYPE_BOOL, _SIZE_TYPE(m_Discontinued), 0, 0, offsetbuf(m_Dis
continued), offsetbuf(m_dwDiscontinuedLength), offsetbuf(m_dwDiscontinuedStatus))
   END_COLUMN_MAP()
};
```

Class Declaration

Finally, the wizard generates a command class declaration such as the following:

```
class CProducts : public CCommand<CAccessor<CProductsAccessor> >
```

Attribute-Injected User Record Classes

If you create an OLE DB consumer using the database attributes (db_command or db_table), the attributes inject a user record class with a name of the form "_*ClassName*Accessor." For example, if you named your command class `COrders`, the user record class will be `_COrdersAccessor`. Although the user record class appears in Class View, double-clicking it navigates to the command or table class in the header file instead. In these cases, you can only view the actual declaration of the user record class by viewing the attribute-injected code.

There can be potential complications if you add or override methods in attributed consumers. For example, you could add a `_COrdersAccessor` constructor to the `COrders` declaration, but note that in reality this adds a constructor to the injected `COrdersAccessor` class. Such a constructor can initialize the columns/parameters, but you cannot create a copy constructor this way, because it cannot directly instantiate the `COrdersAccessor` object. If you need a constructor (or other method) directly on the `COrders` class, it is recommended that you define a new class deriving from `COrders` and add the necessary methods there.

In the following example, the wizard generates a declaration for the class `COrders`, but the user record class `COrdersAccessor` does not appear, because the attributes inject it.

```
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atldbcli.h>
[
   db_source(L"your connection string"),
   db_command(L"Select ShipName from Orders;")
]
class COrders
{
public:

   // COrders()            // incorrect constructor name
   _COrdersAccessor()      // correct constructor name
   {
   }
      [db_column(1) ] TCHAR m_ShipName[41];
   };
```

The injected command class declaration looks like this:

```
class CProducts : public CCommand<CAccessor<_CProductsAccessor> >
```

Most of the injected code is the same as or similar to the templated version. The main differences are in the injected methods, which are described in Consumer Wizard-Generated Methods.

For information about viewing injected code, see Debugging Injected Code.

See Also

Concepts

Creating an OLE DB Consumer Using a Wizard

Consumer Wizard-Generated Methods

The ATL OLE DB Consumer Wizard and the MFC Application Wizard generate certain functions of which you should be aware. Note that some methods are implemented differently in attributed projects, so there are a few caveats; each case is covered below. For information about viewing injected code, see Debugging Injected Code.

- `OpenAll` opens the data source, rowsets, and turns on bookmarks if they are available.
- `CloseAll` closes all open rowsets and releases all command executions.
- `OpenRowset` is called by OpenAll to open the consumer's rowset or rowsets.
- `GetRowsetProperties` retrieves a pointer to the rowset's property set with which properties can be set.
- `OpenDataSource` opens the data source using the initialization string you specified in the **Data Link Properties** dialog box.
- `CloseDataSource` closes the data source in an appropriate manner.

OpenAll and CloseAll

```
HRESULT OpenAll();
void CloseAll();
```

The following example shows how you can call `OpenAll` and `CloseAll` when you execute the same command repeatedly. Compare the code example in CCommand::Close, which shows a variation that calls **Close** and **ReleaseCommand** instead of `CloseAll`.

```
int main(int argc, char* argv[])
{
   HRESULT hr;

   hr = CoInitialize(NULL);

   CCustOrdersDetail rs;      // Your CCommand-derived class
   rs.m_OrderID = 10248;      // Open order 10248
   hr = rs.OpenAll();         // (Open also executes the command)
   hr = rs.MoveFirst();         // Move to the first row and print it
   printf( "Name: %s, Unit Price: %d, Quantity: %d, Discount %d, Extended Price %d\n", rs.m
_ProductName, rs.m_UnitPrice.int64, rs.m_Quantity, rs.m_Discount, rs.m_ExtendedPrice.int64
);

   // Close the first command execution
   rs.Close();


   rs.m_OrderID = 10249;      // Open order 10249 (a new order)
   hr = rs.Open();            // (Open also executes the command)
   hr = rs.MoveFirst();         // Move to the first row and print it
   printf( "Name: %s, Unit Price: %d, Quantity: %d, Discount %d, Extended Price %d\n", rs.m
_ProductName, rs.m_UnitPrice.int64, rs.m_Quantity, rs.m_Discount, rs.m_ExtendedPrice.int64
);

   // Close the second command execution;
   // Instead of rs.CloseAll() you could call
   // rs.Close() and rs.ReleaseCommand():
   rs.CloseAll();

   CoUninitialize();
   return 0;
}
```

Remarks

Note that if you define a `HasBookmark` method, the `OpenAll` code sets the DBPROP_IRowsetLocate property; make sure you only do this if your provider supports that property.

OpenRowset

```
// OLE DB Template version:
HRESULT OpenRowset(DBPROPSET* pPropSet = NULL)
// Attribute-injected version:
HRESULT OpenRowset(const CSession& session, LPCWSTR szCommand = NULL);
```

OpenAll calls this method to open the rowset or rowsets in the consumer. Typically, you do not need to call **OpenRowset** unless you want to work with multiple data sources/sessions/rowsets. **OpenRowset** is declared in the command or table class header file:

```
// OLE DB Template version:
HRESULT OpenRowset(DBPROPSET *pPropSet = NULL)
{
   HRESULT hr = Open(m_session, NULL, pPropSet);
   #ifdef _DEBUG
   if(FAILED(hr))
      AtlTraceErrorRecords(hr);
   #endif
   return hr;
}
```

The attributes implement this method differently. This version takes a session object and a command string that defaults to the command string specified in db_command, although you can pass a different one. Note that if you define a `HasBookmark` method, the `OpenRowset` code sets the DBPROP_IRowsetLocate property; make sure you only do this if your provider supports that property.

```
// Attribute-injected version:
HRESULT OpenRowset(const CSession& session, LPCWSTR szCommand=NULL)
{

   DBPROPSET *pPropSet = NULL;
   CDBPropSet propset(DBPROPSET_ROWSET);
   __if_exists(HasBookmark)

   {
      propset.AddProperty(DBPROP_IRowsetLocate, true);
      pPropSet= &propset;
      }
...
}
```

GetRowsetProperties

```
void GetRowsetProperties(CDBPropSet* pPropSet);
```

This method retrieves a pointer to the rowset's property set; you can use this pointer to set properties such as DBPROP_IRowsetChange. `GetRowsetProperties` is used in the user record class as follows. You can modify this code to set additional rowset properties:

```
void GetRowsetProperties(CDBPropSet* pPropSet)
{
   pPropSet->AddProperty(DBPROP_CANFETCHBACKWARDS, true, DBPROPOPTIONS_OPTIONAL);
   pPropSet->AddProperty(DBPROP_CANSCROLLBACKWARDS, true, DBPROPOPTIONS_OPTIONAL);
   pPropSet->AddProperty(DBPROP_IRowsetChange, true, DBPROPOPTIONS_OPTIONAL);
   pPropSet->AddProperty(DBPROP_UPDATABILITY, DBPROPVAL_UP_CHANGE | DBPROPVAL_UP_INSERT | D
BPROPVAL_UP_DELETE);
}
```

Remarks

You should not define a global `GetRowsetProperties` method because it could conflict with the one defined by the wizard. Note that this is a wizard-generated method that you get with templated and attributed projects; the attributes do not inject this code.

OpenDataSource and CloseDataSource

```
HRESULT OpenDataSource();
void CloseDataSource();
```

Remarks

The wizard defines the methods `OpenDataSource` and `CloseDataSource`; `OpenDataSource` calls CDataSource::OpenFromInitializationString.

See Also

Concepts

Creating an OLE DB Consumer Using a Wizard

Creating a Consumer Without Using a Wizard

The following example assumes that you are adding OLE DB consumer support to an existing ATL project. If you want to add OLE DB consumer support to an MFC application, you should run the MFC Application Wizard, which creates all the support necessary and invokes MFC routines necessary to execute the application.

To add OLE DB consumer support without using the ATL OLE DB Consumer Wizard:

- In your Stdafx.h file, append the following **#include** statements:

```
#include <atlbase.h>
#include <atldbcli.h>
#include <atldbsch.h> // if you are using schema templates
```

Programmatically, a consumer typically performs the following sequence of operations:

- Create a user record class that binds columns to local variables. In this example, `CMyTableNameAccessor` is the user record class (see User Records). This class contains the column map and parameter map. Declare a data member in the user record class for each field you specify in your column map; for each of these data members, also declare a status data member and a length data member. For more information, see Field Status Data Members in Wizard-Generated Accessors.

  Note

  If you write your own consumer, the data variables must come before the status and length variables.

- Instantiate a data source and a session. Decide what type of accessor and rowset to use and then instantiate a rowset using CCommand or CTable:

```
CDataSource ds;
CSession ss;
class CMyTableName : public CCommand<CAccessor<CMyTableNameAccessor> >
```

- Call **CoInitialize** to initialize COM. This is usually called in the main code. For example:

```
HRESULT hr = CoInitialize(NULL);
```

- Call CDataSource::Open or one of its variations.
- Open a connection to the data source, open the session, and open and initialize the rowset (and if a command, also execute it):

```
hr = ds.Open();
hr = ss.Open(ds);
hr = rs.Open();            // (Open also executes the command)
```

- Optionally, set rowset properties using **CDBPropSet::AddProperty** and pass them as a parameter to `rs.Open`. For an example of how this is done, see GetRowsetProperties in Consumer Wizard-Generated Methods.
- You can now use the rowset to retrieve/manipulate the data.
- When your application is done, close the connection, session, and rowset:

```
rs.Close();
ss.Close();
ds.Close();
```

If you are using a command, you might want to call **ReleaseCommand** after **Close**. The code example in CCommand::Close shows how to call **Close** and **ReleaseCommand**.

- Call **CoUnInitialize** to uninitialize COM. This is usually called in the main code.

```
CoUninitialize();
```

See Also

Reference

Creating an OLE DB Consumer

Working with OLE DB Consumer Templates

The following topics provide some examples of how to use the OLE DB Consumer Templates in common scenarios:

- Simplifying Data Access with Database Attributes
- Field Status Data Members in Wizard-Generated Accessors
- Traversing a Simple Rowset
- Issuing a Parameterized Query
- Fetching Data
- Updating Rowsets
- Using Stored Procedures
- Using Accessors
- Obtaining Metadata with Schema Rowsets
- Supporting Transactions in OLE DB
- Using OLE DB Record Views
- Using an Existing ADO Recordset
- Updating a Column When Another Table Contains a Reference to the Row
- Using Bookmarks
- Retrieving a BLOB
- Receiving Notifications

For an example of creating and implementing an OLE DB Consumer, see Creating a Simple Consumer.

You can also find examples of how to use the OLE DB Consumer Templates in the following samples:

- CatDB
- DBViewer
- MultiRead

See Also
Concepts
OLE DB Consumer Templates (C++)

Simplifying Data Access with Database Attributes

This topic demonstrates the use of database attributes to simplify database operations.

The basic way to access information from a database is to create a command (or table) class and a user record class for a particular table in the database. The database attributes simplify some of the template declarations that you previously had to do.

To demonstrate the use of database attributes, the following sections show two equivalent table and user record class declarations: the first uses attributes and the second uses OLE DB Templates. Such declaration code is typically placed in a header file named for the table or command object, for example, Authors.h.

By comparing the two files, you can see how much simpler it is to use attributes. Among the differences are:

- Using attributes, you only have to declare one class: `CAuthors`, while with templates you have to declare two: `CAuthorsNoAttrAccessor` and `CAuthorsNoAttr`.
- The **db_source** call in the attributed version is equivalent to the `OpenDataSource()` call in the template declaration.
- The **db_table** call in the attributed version is equivalent to the following template declaration:

  ```
  class CAuthorsNoAttr : public CTable<CAccessor<CAuthorsNoAttrAccessor> >
  ```

- The **db_column** calls in the attributed version are equivalent to the column map (see `BEGIN_COLUMN_MAP ... END_COLUMN_MAP`) in the template declaration.

The attributes inject a user record class declaration for you. The user record class is equivalent to `CAuthorsNoAttrAccessor` in the template declaration. If your table class is `CAuthors`, the injected user record class is named `CAuthorsAccessor`, and you can only view its declaration in injected code. For more information, see "Attribute-Injected User Record Classes" in User Records.

Note that in both the attributed and the templated code, you must set rowset properties using **CDBPropSet::AddProperty**.

For information about the attributes discussed in this topic, see OLE DB Consumer Attributes.

Table and Accessor Declaration Using Attributes

The following code calls **db_source** and **db_table** on the table class. **db_source** specifies the data source and connection to be used. **db_table** injects the appropriate template code to declare a table class. **db_column** specify the column map and inject the accessor declaration. You can use OLE DB consumer attributes in any project that supports ATL.

Here is the table and accessor declaration using attributes:

```
//////////////////////////////////////////////////////////////////////
// Table and accessor declaration using attributes
// authors.h
//////////////////////////////////////////////////////////////////////

// Table class declaration
// (Note that you must provide your own connection string for db_source.)
[
   db_source(L"your connection string"),
   db_table("Authors")
]
class CAuthors
{
public:
   DWORD m_dwAuIDStatus;
   DWORD m_dwAuthorStatus;
   DWORD m_dwYearBornStatus;
   DWORD m_dwAuIDLength;
   DWORD m_dwAuthorLength;
   DWORD m_dwYearBornLength;
   [ db_column(1, status=m_dwAuIDStatus, length=m_dwAuIDLength) ] LONG m_AuID;
   [ db_column(2, status=m_dwAuthorStatus, length=m_dwAuthorLength) ] TCHAR m_Author[51];
   [ db_column(3, status=m_dwYearBornStatus, length=m_dwYearBornLength) ] SHORT m_YearBorn;
```

```
   void GetRowsetProperties(CDBPropSet* pPropSet)
   {
      pPropSet->AddProperty(DBPROP_CANFETCHBACKWARDS, true);
      pPropSet->AddProperty(DBPROP_CANSCROLLBACKWARDS, true);
      pPropSet->AddProperty(DBPROP_IRowsetChange, true);
   }
};
```

Table and Accessor Declaration Using Templates

Here is the table and accessor declaration using templates.

```
////////////////////////////////////////////////////////////////////
// Table and user record class declaration using templates
// authors.h
////////////////////////////////////////////////////////////////////

// User record class declaration
class CAuthorsNoAttrAccessor
{
public:
   DWORD m_dwAuIDStatus;
   DWORD m_dwAuthorStatus;
   DWORD m_dwYearBornStatus;
   DWORD m_dwAuIDLength;
   DWORD m_dwAuthorLength;
   DWORD m_dwYearBornLength;
   LONG m_AuID;
   TCHAR m_Author[51];
   SHORT m_YearBorn;
   void GetRowsetProperties(CDBPropSet* pPropSet)
   {
      pPropSet->AddProperty(DBPROP_CANFETCHBACKWARDS, true);
      pPropSet->AddProperty(DBPROP_CANSCROLLBACKWARDS, true);
      pPropSet->AddProperty(DBPROP_IRowsetChange, true);
   }
   HRESULT OpenDataSource()
   {
      CDataSource _db;
      HRESULT hr;
      hr = _db.OpenFromInitializationString(L"your connection string");
      if (FAILED(hr))
      {
#ifdef _DEBUG
         AtlTraceErrorRecords(hr);
#endif
         return hr;
      }
      return m_session.Open(_db);
   }
   void CloseDataSource()
   {
      m_session.Close();
   }
   operator const CSession&()
   {
      return m_session;
   }
   CSession m_session;
   BEGIN_COLUMN_MAP(CAuthorsNoAttrAccessor)
      COLUMN_ENTRY_LENGTH_STATUS(1, m_AuID, m_dwAuIDLength, m_dwAuIDStatus)
      COLUMN_ENTRY_LENGTH_STATUS(2, m_Author, m_dwAuthorLength, m_dwAuthorStatus)
      COLUMN_ENTRY_LENGTH_STATUS(3, m_YearBorn, m_dwYearBornLength, m_dwYearBornStatus)
   END_COLUMN_MAP()
};
class CAuthorsNoAttr : public CTable<CAccessor<CAuthorsNoAttrAccessor> >
{
public:
```

```
    HRESULT OpenAll()
    {
       HRESULT hr;
       hr = OpenDataSource();
       if (FAILED(hr))
          return hr;
       __if_exists(GetRowsetProperties)
       {
          CDBPropSet propset(DBPROPSET_ROWSET);
          __if_exists(HasBookmark)
          {
             propset.AddProperty(DBPROP_IRowsetLocate, true);
          }
          GetRowsetProperties(&propset);
          return OpenRowset(&propset);
       }
       __if_not_exists(GetRowsetProperties)
       {
          __if_exists(HasBookmark)
          {
             CDBPropSet propset(DBPROPSET_ROWSET);
             propset.AddProperty(DBPROP_IRowsetLocate, true);
             return OpenRowset(&propset);
          }
       }
       return OpenRowset();
    }
    HRESULT OpenRowset(DBPROPSET *pPropSet = NULL)
    {
       HRESULT hr = Open(m_session, "Authors", pPropSet);
#ifdef _DEBUG
       if(FAILED(hr))
          AtlTraceErrorRecords(hr);
#endif
       return hr;
    }
    void CloseAll()
    {
       Close();
       CloseDataSource();
    }
};
```

See Also

Other Resources

OLE DB Consumer Attributes

Attributes Walkthroughs

Field Status Data Members in Wizard-Generated Accessors

When you use the ATL OLE DB Consumer Wizard to create a consumer, the wizard generates a data member in the user record class for each field that you specify in your column map. Each data member is of type **DWORD** and contains a status value corresponding to its respective field.

For example, for a data member *m_OwnerID*, the wizard generates an additional data member for field status (*dwOwnerIDStatus*) and another one for field length (*dwOwnerIDLength*). It also generates a column map with **COLUMN_ENTRY_LENGTH_STATUS** entries.

This is shown in the following code:

```
[db_source("insert connection string")]
[db_command(" \
   SELECT \
      Au_ID, \
      Author, \
      `Year Born`, \
      FROM Authors")]

class CAuthors
{
public:

   // The following wizard-generated data members contain status
   // values for the corresponding fields in the column map. You
   // can use these values to hold NULL values that the database
   // returns or to hold error information when the compiler returns
   // errors. See "Field Status Data Members in Wizard-Generated
   // Accessors" in the Visual C++ documentation for more information
   // on using these fields.
   DWORD m_dwAuIDStatus;
   DWORD m_dwAuthorStatus;
   DWORD m_dwYearBornStatus;

   // The following wizard-generated data members contain length
   // values for the corresponding fields in the column map.
   DWORD m_dwAuIDLength;
   DWORD m_dwAuthorLength;
   DWORD m_dwYearBornLength;

BEGIN_COLUMN_MAP(CAuthorsAccessor)
   COLUMN_ENTRY_LENGTH_STATUS(1, m_AuID, dwAuIDLength, dwAuIDStatus)
   COLUMN_ENTRY_LENGTH_STATUS(2, m_Author, dwAuthorLength, dwAuthorStatus)
   COLUMN_ENTRY_LENGTH_STATUS(3, m_YearBorn, dwYearBornLength, dwYearBornStatus)
END_COLUMN_MAP()

   [ db_column(1, status=m_dwAuIDStatus, length=m_dwAuIDLength) ] LONG m_AuID;
   [ db_column(2, status=m_dwAuthorStatus, length=m_dwAuthorLength) ] TCHAR m_Author[51];
   [ db_column(3, status=m_dwYearBornStatus, length=m_dwYearBornLength) ] SHORT m_YearBorn;

   void GetRowsetProperties(CDBPropSet* pPropSet)
   {
      pPropSet->AddProperty(DBPROP_IRowsetChange, true);
      pPropSet->AddProperty(DBPROP_UPDATABILITY, DBPROPVAL_UP_CHANGE | DBPROPVAL_UP_INSERT
| DBPROPVAL_UP_DELETE);
   }
};
```

Note

If you modify the user record class or write your own consumer, the data variables must come before the status and length v ariables.

You can use the status values for debugging purposes. If code generated by the ATL OLE DB Consumer Wizard generates compilation errors such as **DB_S_ERRORSOCCURRED** or **DB_E_ERRORSOCCURRED**, you should first look at the current values of the field status data members. Those that have nonzero values correspond to the offending columns.

You can also use the status values to set a NULL value for a particular field. Doing so helps you in cases in which you want to distinguish a field value as NULL rather than zero. It is up to you to decide whether NULL is a valid value or a special value and decide how your application should handle it. OLE DB defines **DBSTATUS_S_ISNULL** as the correct means of specifying a generic NULL value. If the consumer reads data and the value is null, the status field is set to **DBSTATUS_S_ISNULL**. If the consumer wants to set a NULL value, the consumer sets the status value to **DBSTATUS_S_ISNULL** before calling the provider.

Next, open Oledb.h and search for **DBSTATUSENUM**. You can then match the numerical value of the nonzero status against the **DBSTATUSENUM** enumeration values. If the enumeration name is not sufficient to tell you what is wrong, see Status in the OLE DB 2.0 Programmer's Reference; this topic contains tables of status values used when getting or setting data. For information about length values, see Length.

Retrieving the Length or Status of a Column

You can retrieve the length of a variable-length column or the status of a column (to check for **DBSTATUS_S_ISNULL**, for example):

- To get the length, use the **COLUMN_ENTRY_LENGTH** macro.
- To get the status, use the **COLUMN_ENTRY_STATUS** macro.
- To get both, use **COLUMN_ENTRY_LENGTH_STATUS**, as shown below.

```
class CProducts
{
public:
   char      szName[40];
   long      nNameLength;
   DBSTATUS   nNameStatus;

BEGIN_COLUMN_MAP(CProducts)
// Bind the column to CProducts.m_ProductName.
// nOrdinal is zero-based, so the column number of m_ProductName is 1.
   COLUMN_ENTRY_LENGTH_STATUS(1, szName, nNameLength, nNameStatus)
END_COLUMN_MAP()
};

CTable<CAccessor<CProducts > > product;

product.Open(session, "Product");
while (product.MoveNext() == S_OK)
{
   // Check the product name isn't NULL before tracing it
   if (product.nNameStatus == DBSTATUS_S_OK)
      ATLTRACE("%s is %d characters\n", szName, nNameLength);
}
```

When you use **CDynamicAccessor**, the length and status are bound for you automatically. To retrieve the length and status values, use the **GetLength** and **GetStatus** member functions.

See Also

Reference

Working with OLE DB Consumer Templates

Traversing a Simple Rowset

The following example shows a quick and easy database access that does not involve commands. The following consumer code, in an ATL project, retrieves records from a table called *Artists* in a Microsoft Access database using the Microsoft OLE DB Provider for ODBC. The code creates a CTable table object with an accessor based on the user record class `CArtists`. It opens a connection, opens a session on the connection, and opens the table on the session.

```
#include <atldbcli.h>

CDataSource connection;
CSession session;
CTable<CAccessor<CArtists> > artists;

// Open the connection, session, and table, specifying authentication
// using Windows NT integrated security. Hard-coding a password is a major
// security weakness.
connection.Open(CLSID_MSDASQL, "NWind", NULL, NULL,
DBPROP_AUTH_INTEGRATED);
session.Open(connection);
artists.Open(session, "Artists");

// Get data from the rowset
while (artists.MoveNext() == S_OK)
{
   cout << artists.m_szFirstName;
   cout << artists.m_szLastName;
}
```

The user record, `CArtists`, looks like this:

```
class CArtists
{
public:
// Data Elements
   CHAR m_szFirstName[20];
   CHAR m_szLastName[30];
   short m_nAge;

// Column binding map
BEGIN_COLUMN_MAP(CArtists)
   COLUMN_ENTRY(1, m_szFirstName)
   COLUMN_ENTRY(2, m_szLastName)
   COLUMN_ENTRY(3, m_nAge)
END_COLUMN_MAP()
```

See Also

Reference

Working with OLE DB Consumer Templates

Issuing a Parameterized Query

The following example issues a simple parameterized query that retrieves records with an age field (that is greater than 30) from a table in a Microsoft Access database. To support the parameter, the user record must have an additional map. The following code, in an ATL project, uses the **CCommand** class instead of the **CTable** class used in the previous example, Traversing a Simple Rowset.

```
#include <atldbcli.h>

CDataSource connection;
CSession session;
CCommand<CAccessor<CArtists> > artists;

// Open the connection, session, and table, specifying authentication
// using Windows NT integrated security. Hard-coding a password is a major
// security weakness.
connection.Open(CLSID_MSDASQL, "NWind", NULL, NULL,
DBPROP_AUTH_INTEGRATED);
session.Open(connection);

// Set the parameter for the query
artists.m_nAge = 30;
artists.Open(session, "select * from artists where age > ?");

// Get data from the rowset
while (artists.MoveNext() == S_OK)
{
   cout << artists.m_szFirstName;
   cout << artists.m_szLastName;
}
```

The user record, `CArtists`, looks like this:

```
class CArtists
{
public:
// Data Elements
   CHAR m_szFirstName[20];
   CHAR m_szLastName[30];
   short m_nAge;

// Column binding map
BEGIN_COLUMN_MAP(CArtists)
   COLUMN_ENTRY(1, m_szFirstName)
   COLUMN_ENTRY(2, m_szLastName)
   COLUMN_ENTRY(3, m_nAge)
END_COLUMN_MAP()

// Parameter binding map
BEGIN_PARAM_MAP(CArtists)
   SET_PARAM_TYPE(DBPARAMIO_INPUT)
   COLUMN_ENTRY(1, m_nAge)
END_PARAM_MAP()
};
```

See Also

Reference

Working with OLE DB Consumer Templates

Fetching Data

After you open the data source, session, and rowset objects, you can fetch data. Depending on the type of accessor you are using, you might need to bind columns.

To fetch data

1. Open the rowset using the appropriate **Open** command.
2. If you are using **CManualAccessor**, bind the output columns if you have not already done so. To bind the columns, call **GetColumnInfo**, and then create an accessor with the bindings, as shown in the following example:

```
// From the DBViewer Sample CDBTreeView::OnQueryEdit
// Get the column information
ULONG ulColumns       = 0;
DBCOLUMNINFO* pColumnInfo  = NULL;
LPOLESTR pStrings      = NULL;
if (rs.GetColumnInfo(&ulColumns, &pColumnInfo, &pStrings) != S_OK)
    ThrowMyOLEDBException(rs.m_pRowset, IID_IColumnsInfo);
struct MYBIND* pBind = new MYBIND[ulColumns];
rs.CreateAccessor(ulColumns, &pBind[0], sizeof(MYBIND)*ulColumns);
for (ULONG l=0; l<ulColumns; l++)
rs.AddBindEntry(l+1, DBTYPE_STR, sizeof(TCHAR)*40, &pBind[l].szValue, NULL, &pBind[l].
dwStatus);
rs.Bind();
```

3. Write a **while** loop to retrieve the data. In the loop, call **MoveNext** to advance the cursor and test the return value against S_OK, as shown in the following example:

```
while (rs.MoveNext() == S_OK)
{
    // Add code to fetch data here
    // If you are not using an auto accessor, call rs.GetData()
}
```

4. Within the **while** loop, you can fetch the data according to your accessor type.

   - If you use the CAccessor class, you should have a user record that contains data members. You can access your data using those data members, as shown in the following example:

```
while (rs.MoveNext() == S_OK)
{
    // Use the data members directly. In this case, m_nFooID
    // is declared in a user record that derives from
    // CAccessor
    wsprintf_s("%d", rs.m_nFooID);
}
```

   - If you use the **CDynamicAccessor** or **CDynamicParameterAccessor** class, you can fetch data by using the accessing functions **GetValue** and **GetColumn**, as shown in the following example. If you want to determine the type of data you are using, use **GetType**.

```
while (rs.MoveNext() == S_OK)
{
    // Use the dynamic accessor functions to retrieve your data.
```

```
    ULONG ulColumns = rs.GetColumnCount();
    for (ULONG i=0; i<ulColumns; i++)
    {
        rs.GetValue(i);
    }
}
```

- If you use **CManualAccessor**, you must specify your own data members, bind them yourself, and access them directly, as shown in the following example:

```
while (rs.MoveNext() == S_OK)
{
    // Use the data members you specified in the calls to
    // AddBindEntry.

    wsprintf_s("%s", szFoo);
}
```

See Also

Reference

Working with OLE DB Consumer Templates

Updating Rowsets

A very basic database operation is to update, or write data to, the data store. In OLE DB, the update mechanism is simple: your consumer application sets the values of bound data members and then writes those values to the rowset; the consumer then requests that the provider update the data store.

Consumers can perform the following kinds of updates on rowset data: setting column values within a row, inserting a row, and deleting a row. To perform these operations, the OLE DB Template class CRowset implements the IRowsetChange interface and overrides the following interface methods:

- SetData changes column values in a row of a rowset; it is equivalent to the SQL UPDATE command.
- Insert inserts a row into a rowset; it is equivalent to the SQL INSERT command.
- Delete deletes rows from a rowset; it is equivalent to the SQL DELETE command.

Supporting Update Operations

When you create a consumer with the ATL OLE DB Consumer Wizard, you can support the update operations by selecting one or more of the three check boxes **Change**, **Insert**, and **Delete**. If you select these, the wizard modifies the code appropriately to support the type of changes you choose. However, if you do not use the wizard, you need to set the following rowset properties to **VARIANT_TRUE** to support updates:

- **DBPROPVAL_UP_CHANGE** allows you to change the data values in a row.
- **DBPROPVAL_UP_INSERT** allows you to insert a row.
- **DBPROPVAL_UP_DELETE** allows you to delete a row.

You set the properties as follows:

```
CDBPropSet ps(DBPROPSET_ROWSET);
ps.AddProperty(DBPROP_IRowsetChange, true)
ps.AddProperty(DBPROP_UPDATABILITY, DBPROPVAL_UP_CHANGE | DBPROPVAL_UP_INSERT | DBPROPVAL_U
P_DELETE)
```

Change, insert, or delete operations might fail if one or more columns is not writable. Modify your cursor map to correct this.

Setting Data in Rows

CRowset::SetData sets data values in one or more columns of the current row. The following code sets the values of data members bound to the columns "Name" and "Units in Stock" of the table Products and then calls **SetData** to write those values to the 100th row of the rowset:

```
// Instantiate a rowset based on the user record class
CTable<CAccessor<CProductAccessor> > product;
CSession session;

// Open the rowset and move to the 100th row
product.Open(session, "Product", &ps, 1);  // ps is the property set
product.MoveToBookmark(&bookmark, 0);      // Assume that bookmark is set to 100th row

// Change the values of columns "Name" and "Units in Stock" in the current row of the Produ
ct table
_tcscpy_s( product.m_ProductName, product.m_sizeOfProductName,
           _T( "Candle" ) );
product.m_UnitsInStock = 10000;

// Set the data
HRESULT hr = product.SetData( );
```

Inserting Rows into Rowsets

CRowset::Insert creates and initializes a new row using data from the accessor. **Insert** creates an entirely new row after the

current row; you need to specify whether to increment the current row to the next row or leave it unchanged. You do this by setting the *bGetRow* parameter:

```
HRESULT Insert(int nAccessor = 0, bool bGetRow = false)
```

- **false** (the default value) specifies that the current row increment to the next row (in which case it points to the inserted row).
- **true** specifies that the current row remain where it is.

The following code sets the values of data members bound to the columns of the table Products and then calls **Insert** to insert a new row with those values after the 100th row of the rowset. It is recommended that you set all column values to avoid undefined data in the new row:

```
// Instantiate a rowset based on the user record class
CTable<CAccessor<CProductAccessor> > product;
CSession session;

// Open the rowset and move to the 100th row
product.Open(session, "Product", &ps, 1);  // ps is the property set
product.MoveToBookmark(&bookmark, 0);      // Assume that bookmark is set to 100th row

// Set the column values for a row of the Product table, then insert the row
product.m_ProductID = 101;
_tcscpy_s( product.m_ProductName, product.m_sizeOfProductName,
           _T( "Candle" ) );
product.m_SupplierID = 27857;
product.m_CategoryID = 372;
_tcscpy_s( product.m_QuantityPerUnit, product.m_sizeOfQuantityPerUnit,
           _T( "Pack of 10" ) );
product.m_UnitPrice = 20;
product.m_UnitsInStock = 10000;
product.m_UnitsOnOrder = 5201;
product.m_ReorderLevel = 5000;
product.m_Discontinued = false;

// You must also initialize the status and length fields before setting/inserting data
// Set the column status values
m_dwProductIDStatus = DBSTATUS_S_OK;
m_dwProductNameStatus = DBSTATUS_S_OK;
m_dwSupplierIDStatus = DBSTATUS_S_OK;
m_dwCategoryIDStatus = DBSTATUS_S_OK;
m_dwQuantityPerUnitStatus = DBSTATUS_S_OK;
m_dwUnitPriceStatus = DBSTATUS_S_OK;
m_dwUnitsInStockStatus = DBSTATUS_S_OK;
m_dwUnitsOnOrderStatus = DBSTATUS_S_OK;
m_dwReorderLevelStatus = DBSTATUS_S_OK;
m_dwDiscontinuedStatus = DBSTATUS_S_OK;

// Set the column length value for column data members that are not fixed-length types.
// The value should be the length of the string that you are setting.
m_dwProductNameLength = 6;             // "Candle" has 6 characters
m_dwQuantityPerUnitLength = 10;        // "Pack of 10" has 10 characters

// Insert the data
HRESULT hr = product.Insert( );
```

For a more detailed example, see CRowset::Insert.

For more information about setting the status and length data members, see Field Status Data Members in Wizard-Generated Accessors.

Deleting Rows from Rowsets

CRowset::Delete deletes the current row from the rowset. The following code calls **Delete** to remove the 100th row of the rowset:

```
// Instantiate a rowset based on the user record class
CTable<CAccessor<CProductAccessor> > product;
CSession session;

// Open the rowset and move to the 100th row
product.Open(session, "Product", &ps, 1);  // ps is the property set
product.MoveToBookmark(&bookmark, 0);      // Assume that bookmark is set to 100th row

// Delete the row
HRESULT hr = product.Delete( );
```

Immediate and Deferred Updates

Unless you specify otherwise, calls to the **SetData**, **Insert**, and **Delete** methods update the data store immediately. You can, however, defer updates so that the consumer stores all changes in a local cache and then transfers them to the data store when you call one of the following update methods:

- CRowset::Update transfers any pending changes made to the current row since the last fetch or **Update** call on it.
- CRowset::UpdateAll transfers any pending changes made to all rows since the last fetch or **Update** call on it.

Note that update, as used by the update methods, has the specific meaning of making changes on command and is not to be confused with the SQL UPDATE command (**SetData** is equivalent to the SQL UPDATE command).

Deferred updates are useful, for example, in situations such as a series of banking transactions; if one transaction is canceled, you can undo the change, because you do not send the series of changes until after the last one is committed. Also, the provider can bundle the changes into one network call, which is more efficient.

To support deferred updates, you must set the **DBPROP_IRowsetChange** property in addition to the properties described in "Supporting Update Operations":

```
    pPropSet->AddProperty(DBPROP_IRowsetUpdate, true);
```

When you call **Update** or **UpdateAll**, the methods transfer changes from the local cache to the data store and then wipe out the local cache. Because update transfers changes only for the current row, it is important that your application keep track of which row to update and when to update it. The following example shows how to update two consecutive rows:

```
// Instantiate a rowset based on the user record class
CTable<CAccessor<CProductAccessor> > product;
CSession session;

// Open the rowset and move to the 100th row
product.Open(session, "Product", &ps, 1);  // ps is the property set
product.MoveToBookmark(&bookmark, 0);      // Assume that bookmark is set to 100th row

// Change the values of columns "Name" and "Units in Stock" in the 100th row of the Product
table
_tcscpy_s( product.m_ProductName, product.m_sizeOfProductName,
           _T( "Wick" ) );
product.m_UnitsInStock = 10000;
HRESULT hr = product.SetData( );  // No changes made to row 100 yet
product.Update();                 // Update row 100 now

// Change the values of columns "Name" and "Units in Stock" in the 101st row of the Product
table
product.MoveNext( );
_tcscpy_s( product.m_ProductName, product.m_sizeOfProductName
           _T( "Wax" ) );
product.m_UnitsInStock = 500;
HRESULT hr = product.SetData( );  // No changes made to row 101 yet
product.Update();                 // Update row 101 now
```

To ensure that pending changes are transferred, you should call **Update** before moving to another row. However, when this is tedious or inefficient, for example, when your application needs to update hundreds of rows, you can use **UpdateAll** to update all the rows at once.

For example, if the first **Update** call were missing from the above code, row 100 would remain unchanged, while row 101 would be changed. After that point, your application would either have to call **UpdateAll** or move back to row 100 and call **Update** for that row to be updated.

Finally, one main reason to defer changes is to be able to undo them. Calling CRowset::Undo rolls back the state of the local change cache to the state of the data store before any pending changes were made. It is important to note that **Undo** does not roll back the state of the local cache by one step (the state before only the latest change); instead, it clears the local cache for that row. Also, **Undo** affects only the current row.

See Also

Reference

Working with OLE DB Consumer Templates
CRowset Class
IRowsetChange

Using Stored Procedures

A stored procedure is an executable object stored in a database. Calling a stored procedure is similar to invoking a SQL command. Using stored procedures on the data source (instead of executing or preparing a statement in the client application) can provide several advantages, including higher performance, reduced network overhead, and improved consistency and accuracy.

A stored procedure can have any number of (including zero) input or output parameters and can pass a return value. You can either hard code parameter values as specific data values or use a parameter marker (a question mark '?'), as shown below.

This topic covers regular stored procedures. For CLR stored procedures using Visual C++ 2005 and SQL Server 2005, see Creating SQL Server 2005 Objects in Managed Code.

Note
CLR SQL Server stored procedures created using Visual C++ must be compiled with the **/clr:safe** compiler option.

The OLE DB provider for SQL Server (SQLOLEDB) supports the following mechanisms that stored procedures use to return data:

- Every SELECT statement in the procedure generates a result set.
- The procedure can return data through output parameters.
- The procedure can have an integer return code.

Note
You cannot use stored procedures with the OLE DB provider for Jet because that provider does not support stored procedures; only constants are allowed in query strings.

See Also

Reference

Working with OLE DB Consumer Templates
Running Stored Procedures (OLE DB)
Calling a Stored Procedure (OLE DB)

Defining Stored Procedures

Before calling a stored procedure, you must first define it, using the DEFINE_COMMAND macro. When you define the command, denote parameters with a question mark (?) as the parameter marker:

```
DEFINE_COMMAND(CMySProcAccessor, _T("{INSERT {name, phone} into shippers  (?,?)}")
```

Note that the syntax (the use of braces and so on) used in the code examples in this topic is specific to SQL Server. The syntax that you use in your stored procedures might vary according to the provider you use.

Next, in the parameter map, specify the parameters that you used in the command, listing the parameters in the order that they occur in the command:

```
BEGIN_PARAM_MAP(CMySProcAccessor)
   SET_PARAM_TYPE(DBPARAMIO_INPUT)
   COLUMN_ENTRY(1, m_Name)   // name corresponds to first '?' param
   SET_PARAM_TYPE(DBPARAMIO_INPUT)
   COLUMN_ENTRY(2, m_Phone)  // phone corresponds to second '?' param
END_PARAM_MAP()
```

The previous example defines a stored procedure as it goes. Typically, for efficient reuse of code, a database contains a set of predefined stored procedures with names such as "Sales by Year" or "dt_adduserobject." You can view their definitions using SQL Server Enterprise Manager. You call them as follows (the placement of the '?' parameters depends on the stored procedure's interface):

```
DEFINE_COMMAND(CMySProcAccessor, _T("{CALL \"Sales by Year\" (?,?) }")
DEFINE_COMMAND(CMySProcAccessor, _T("{CALL dbo.dt_adduserobject (?,?) }")
```

Next, declare the command class:

```
class CMySProc : public CCommand<CAccessor<CMySProcAccessor> >
```

Finally, call the stored procedure in `OpenRowset` as follows:

```
CSession m_session;
HRESULT OpenRowset()
{
   return CCommand<CAccessor<CMySProcAccessor> >::Open(m_session);
}
```

Also note that you can define a stored procedure using the database attribute db_command as follows:

```
db_command("{ ? = CALL dbo.dt_adduserobject }")
```

See Also

Concepts

Using Stored Procedures

Output Parameters

Calling a stored procedure is similar to invoking a SQL command. The main difference is that stored procedures use output parameters (or "outparameters") and return values.

In the following stored procedure, the first '?' is the return value (phone) and the second '?' is the input parameter (name):

```
DEFINE_COMMAND(CMySProcAccessor, _T("{ ? = SELECT phone FROM shippers WHERE name = ? }")
```

You specify the in and out parameters in the parameter map:

```
BEGIN_PARAM_MAP(CMySProcAccessor)
   SET_PARAM_TYPE(DBPARAMIO_OUTPUT)
   COLUMN_ENTRY(1, m_Phone)   // Phone is the return value
   SET_PARAM_TYPE(DBPARAMIO_INPUT)
   COLUMN_ENTRY(2, m_Name)   // Name is the input parameter
END_PARAM_MAP()
```

Your application must handle the output returned from stored procedures. Different OLE DB providers return output parameters and return values at different times during result processing. For example, the Microsoft OLE DB provider for SQL Server (SQLOLEDB) does not supply output parameters and return codes until after the consumer has retrieved or canceled the result sets returned by the stored procedure. The output is returned in the last TDS packet from the server.

Row Count

If you use the OLE DB Consumer Templates to execute a stored procedure that has outparameters, the row count is not set until you close the rowset.

For example, consider a stored procedure with a rowset and an outparameter:

```
create procedure sp_test
   @_rowcount integer output
as
   select top 50 * from test
   @_rowcount = @@rowcount
return 0
```

The @_rowcount outparameter reports how many rows were actually returned from the test table. However, this stored procedure limits the number of rows to a maximum of 50. For example, if there were 100 rows in test, the rowcount would be 50 (because this code retrieves only the top 50 rows). If there were only 30 rows in the table, the rowcount would be 30. You must call **Close** or **CloseAll** to populate the outparameter before you fetch its value.

See Also

Concepts

Using Stored Procedures

Using Multiple Result Sets from One Stored Procedure

Most stored procedures return multiple result sets. Such a stored procedure usually includes one or more select statements. The consumer needs to consider this to handle all the result sets.

To handle multiple result sets

1. Create a **CCommand** class with **CMultipleResults** as a template argument and with the accessor of your choice. Usually, this is a dynamic or manual accessor. If you use another type of accessor, you might not be able to determine the output columns for each rowset.
2. Execute the stored procedure as usual and bind the columns (see How Do I Fetch Data?).
3. Use the data.
4. Call **GetNextResult** on the **CCommand** class. If another result rowset is available, **GetNextResult** returns S_OK and you should rebind your columns if you are using a manual accessor. If **GetNextResult** returns an error, there are no further result sets available.

See Also

Concepts

Using Stored Procedures

Using Accessors

The following topics explain how to determine which type of accessor is best for your data application and then how to use the different types of accessors available:

- Determining Which Accessor to Use
- Using Multiple Accessors on a Rowset
- Using Dynamic Accessors
- Using Manual Accessors
- Accessing XML Data

See Also

Reference

Working with OLE DB Consumer Templates

Determining Which Type of Accessor to Use

You can determine data types on a rowset at compile time or at run time.

If you need to determine data types at compile time, use a static accessor (such as **CAccessor**). You can determine the data types manually or by using the ATL OLE DB Consumer Wizard.

If you need to determine the data types at run time, use a dynamic (**CDynamicAccessor** or its children) or manual accessor (**CManualAccessor**). In these cases, you can call **GetColumnInfo** on the rowset to return the column binding information, from which you can determine types.

The following table lists the types of accessors provided in the consumer templates. Each accessor has advantages and disadvantages. Depending on your situation, one accessor type should suit your needs.

Accessor class	**Binding**	**Parameter**	**Comment**
CAccessor	Create a user record with **COLUMN_ENTRY** macros. The macros bind a data member in that record to the accessor. When the rowset is created, columns cannot be unbound.	Yes, by using a **PARAM_MAP** macro entry. Once bound, parameters cannot be unbound.	Fastest accessor because of small amount of code.
CDynamicAccessor	Automatic.	No.	Useful if you do not know the type of data in a rowset.
CDynamicParameterAccessor	Automatic, but can be overridden.	Yes, if the provider supports **ICommandWithParameters**. Parameters bound automatically.	Slower than **CDynamicAccessor** but useful for calling generic stored procedures.
CDynamicStringAccessor[A,W]	Automatic.	No.	Retrieves data accessed from the data store as string data.
CManualAccessor	Manual using **AddBindEntry**.	Manually using **AddParameterEntry**.	Very fast; parameters and columns bound only once. You determine the type of data to use. (See DBVIEWER sample for an example.) Requires more code than **CDynamicAccessor** or **CAccessor**. It is more like calling OLE DB directly.
CXMLAccessor	Automatic.	No.	Retrieves data accessed from the data store as string data and formats it as XML-tagged data.

See Also

Concepts

Using Accessors

Using Multiple Accessors on a Rowset

There are three basic scenarios in which you need to use multiple accessors:

- **Multiple read/write rowsets.** In this scenario, you have a table with a primary key. You want to be able to read all the columns in the row, including the primary key. You also want to be able to write data to all the columns except the primary key (because you cannot write to the primary key column). In this case, you set up two accessors:
  - Accessor 0 contains all columns.
  - Accessor 1 contains all columns except the primary key.
- **Performance.** In this scenario, one or more columns contain a large amount of data, for example, graphics, sound, or video files. Every time you move to a row, you probably do not want to retrieve the column with the large data file, because doing so would slow down your application's performance.

  You can set up separate accessors in which the first accessor contains all columns except the one with large data, and it retrieves data from these columns automatically; this is the auto accessor. The second accessor retrieves only the column containing large data, but it does not retrieve data from this column automatically. You can have other methods update or fetch the large data on demand.
  - Accessor 0 is an automatic accessor; it retrieves all the columns except the one with large data.
  - Accessor 1 is not an automatic accessor; it retrieves the column with large data.

  Use the auto argument to specify whether the accessor is an auto accessor.
- **Multiple ISequentialStream columns.** In this scenario, you have more than one column containing **ISequentialStream** data. However, each accessor is limited to one **ISequentialStream** data stream. To solve this problem, set up several accessors, each containing one **ISequentialStream** pointer.

You normally create accessors using the BEGIN_ACCESSOR and END_ACCESSOR macros. You can also use the db_accessor attribute. (Accessors are described further in User Records.) The macros or the attribute specify whether an accessor is an automatic or a non-automatic accessor:

- In an automatic accessor, move methods such as **MoveFirst**, **MoveLast**, **MoveNext**, and **MovePrev** retrieve data for all specified columns automatically. Accessor 0 should be the automatic accessor.
- In a non-automatic accessor, the retrieval does not occur until you explicitly call a method such as **Update**, **Insert**, **Fetch**, or **Delete**. In the scenarios described above, you might not want to retrieve all the columns on every move. You can place one or more columns in a separate accessor and make that a non-automatic accessor, as shown below.

The following example uses multiple accessors to read and write to the jobs table of the SQL Server pubs database using multiple accessors. This is the most common use of multiple accessors; see the "multiple read/write rowsets" scenario above.

The user record class is as follows. It sets up two accessors: accessor 0 contains only the primary key column (ID) and accessor 1 contains other columns.

```
class CJobs
{
public:
    enum {
        sizeOfDescription = 51
    };

    short nID;
    char szDescription[ sizeOfDescription ];
    short nMinLvl;
    short nMaxLvl;

    DWORD dwID;
    DWORD dwDescription;
    DWORD dwMinLvl;
    DWORD dwMaxLvl;
```

```
BEGIN_ACCESSOR_MAP(CJobs, 2)
    // Accessor 0 is the automatic accessor
    BEGIN_ACCESSOR(0, true)
        COLUMN_ENTRY_STATUS(1, nID, dwID)
    END_ACCESSOR()
    // Accessor 1 is the non-automatic accessor
    BEGIN_ACCESSOR(1, true)
        COLUMN_ENTRY_STATUS(2, szDescription, dwDescription)
        COLUMN_ENTRY_STATUS(3, nMinLvl, dwMinLvl)
        COLUMN_ENTRY_STATUS(4, nMaxLvl, dwMaxLvl)
    END_ACCESSOR()
END_ACCESSOR_MAP()
};
```

The main code is as follows. Calling **MoveNext** automatically retrieves data from the primary key column ID using accessor 0. Note how the **Insert** method near the end uses accessor 1 to avoid writing to the primary key column.

```
int main(int argc, char* argv[])
{
    // Initalize COM
    ::CoInitialize(NULL);

    // Create instances of the data source and session
    CDataSource source;
    CSession session;
    HRESULT hr = S_OK;

    // Set initialization properties
    CDBPropSet dbinit(DBPROPSET_DBINIT);
    dbinit.AddProperty(DBPROP_AUTH_USERID, OLESTR("my_user_id"));
    dbinit.AddProperty(DBPROP_INIT_CATALOG, OLESTR("pubs"));
    dbinit.AddProperty(DBPROP_INIT_DATASOURCE, OLESTR("(local)"));

    hr = source.Open("SQLOLEDB.1", &dbinit);
    if (hr == S_OK)
    {
        hr = session.Open(source);
        if (hr == S_OK)
        {
            // Ready to fetch/access data
            CTable<CAccessor<CJobs> > jobs;

            // Set properties for making the rowset a read/write cursor
            CDBPropSet dbRowset(DBPROPSET_ROWSET);
            dbRowset.AddProperty(DBPROP_CANFETCHBACKWARDS, true);
            dbRowset.AddProperty(DBPROP_CANSCROLLBACKWARDS, true);
            dbRowset.AddProperty(DBPROP_IRowsetChange, true);
            dbRowset.AddProperty(DBPROP_UPDATABILITY,
                DBPROPVAL_UP_INSERT | DBPROPVAL_UP_CHANGE |
                DBPROPVAL_UP_DELETE);

            hr = jobs.Open(session, "jobs", &dbRowset);
            if (hr == S_OK)
            {
                // Calling MoveNext automatically retrieves ID
                // (using accessor 0)
                while(jobs.MoveNext() == S_OK)
                   printf_s("Description = %s\n", jobs.szDescription);

                hr = jobs.MoveFirst();
                if (hr == S_OK)
                {
                    jobs.nID = 25;
                    strcpy_s(&jobs.szDescription[0],
                             jobs.sizeOfDescription,
                             "Developer");
                    jobs.nMinLvl = 10;
```

```
                    jobs.nMaxLvl = 20;

                    jobs.dwDescription = DBSTATUS_S_OK;
                    jobs.dwID = DBSTATUS_S_OK;
                    jobs.dwMaxLvl = DBSTATUS_S_OK;
                    jobs.dwMinLvl = DBSTATUS_S_OK;

                    // Insert method uses accessor 1
                    // (to avoid writing to the primary key column)
                    hr = jobs.Insert(1);
                }
                jobs.Close();
            }
            session.Close();
        }
        source.Close();
    }

    // Uninitialize COM
    ::CoUninitialize();
    return 0;
}
```

See Also

Concepts

Using Accessors

User Records

Using Dynamic Accessors

Dynamic accessors allow you to access a data source when you have no knowledge of the database schema (underlying structure). The OLE DB Templates library provides several classes to help you do this.

The DynamicConsumer sample shows how to use the dynamic accessor classes to obtain column information and dynamically create accessors.

Using CDynamicAccessor

CDynamicAccessor allows you to access a data source when you have no knowledge of the database schema (the database's underlying structure). **CDynamicAccessor** methods obtain column information such as column names, count, and data type. You use this column information to create an accessor dynamically at run time. The column information is stored in a buffer that is created and managed by this class. Obtain data from the buffer using the GetValue method.

Example

```
// Using_Dynamic_Accessors.cpp
// compile with: /c /EHsc
#include <stdio.h>
#include <objbase.h>
#include <atldbcli.h>


int main( int argc, char* argv[] )
{
    HRESULT hr = CoInitialize( NULL );

    CDataSource ds;
    CSession ss;

    CTable<CDynamicAccessor> rs;

    // The following is an example initialization string:
    hr = ds.OpenFromInitializationString(L"Provider=SQLOLEDB.1;"
      L"Integrated Security=SSPI;Persist Security Info=False;"
      L"Initial Catalog=Loginname;Data Source=your_data_source;"
      L"Use Procedure for Prepare=1;Auto Translate=True;"
      L"Packet Size=4096;Workstation ID=LOGINNAME01;"
      L"Use Encryption for Data=False;"
      L"Tag with column collation when possible=False");

    hr = ss.Open( ds );
    hr = rs.Open( ss, "Shippers" );

    hr = rs.MoveFirst( );
    while( SUCCEEDED( hr ) && hr != DB_S_ENDOFROWSET )
    {
        for( size_t i = 1; i <= rs.GetColumnCount( ); i++ )
        {
            DBTYPE type;
            rs.GetColumnType( i, &type );
            printf_s( "Column %d [%S] is of type %d\n",
                      i, rs.GetColumnName( i ), type );

            switch( type )
            {
                case DBTYPE_WSTR:
                    printf_s( "value is %S\n",
                              (WCHAR*)rs.GetValue( i ) );
                break;
                case DBTYPE_STR:
                    printf_s( "value is %s\n",
                              (CHAR*)rs.GetValue( i ) );
                default:
                    printf_s( "value is %d\n",
```

```
                                *(long*)rs.GetValue( i ) );
            }
        }
        hr = rs.MoveNext( );
    }

    rs.Close();
    ss.Close();
    ds.Close();
    CoUninitialize();

    return 0;
}
```

Using CDynamicStringAccessor

CDynamicStringAccessor works like CDynamicAccessor, except in one important way. While **CDynamicAccessor** requests data in the native format reported by the provider, **CDynamicStringAccessor** requests that the provider fetch all data accessed from the data store as string data. This is especially useful for simple tasks that do not require calculation of values in the data store, such as displaying or printing the data store's contents.

Use **CDynamicStringAccessor** methods to obtain column information. You use this column information to create an accessor dynamically at run time. The column information is stored in a buffer created and managed by this class. Obtain data from the buffer using CDynamicStringAccessor::GetString or store it to the buffer using CDynamicStringAccessor::SetString.

Example

```
// Using_Dynamic_Accessors_b.cpp
// compile with: /c /EHsc
#include <stdio.h>
#include <objbase.h>
#include <atldbcli.h>

int main( int argc, char* argv[] )
{
    HRESULT hr = CoInitialize( NULL );
    if (hr != S_OK)
    {
        exit (-1);
    }

    CDataSource ds;
    CSession ss;

    CTable<CDynamicStringAccessor> rs;

    // The following is an example initialization string:
    hr = ds.OpenFromInitializationString(L"Provider=SQLOLEDB.1;"
      L"Integrated Security=SSPI;Persist Security Info=False;"
      L"Initial Catalog=Loginname;Data Source=your_data_source;"
      L"Use Procedure for Prepare=1;Auto Translate=True;"
      L"Packet Size=4096;Workstation ID=LOGINNAME01;"
      L"Use Encryption for Data=False;"
      L"Tag with column collation when possible=False");

    hr = ss.Open( ds );
    hr = rs.Open( ss, "Shippers" );

    hr = rs.MoveFirst( );
    while( SUCCEEDED( hr ) && hr != DB_S_ENDOFROWSET )
    {
        for( size_t i = 1; i <= rs.GetColumnCount( ); i++ )
        {
            printf_s( "column %d value is %s\n",
                      i, rs.GetString( i ) );
        }
        hr = rs.MoveNext( );
```

```
    }

    rs.Close();
    ss.Close();
    ds.Close();
    CoUninitialize();

   return 0;

}
```

Using CDynamicParameterAccessor

CDynamicParameterAccessor is similar to CDynamicAccessor, except that **CDynamicParameterAccessor** obtains parameter information to be set by calling the ICommandWithParameters interface. The provider must support **ICommandWithParameters** for the consumer to use this class.

The parameter information is stored in a buffer created and managed by this class. Obtain parameter data from the buffer by using CDynamicParameterAccessor::GetParam and CDynamicParameterAccessor::GetParamType.

For an example demonstrating how to use this class to execute a SQL Server stored procedure and get the output parameter values, see Knowledge Base article Q058860, "HOWTO: Execute Stored Procedure using CDynamicParameterAccessor." Knowledge Base articles are available in the MSDN Library Visual Studio documentation or at http://support.microsoft.com.

See Also

Tasks

DynamicConsumer Sample: Uses Dynamic Accessor and Schema Rowset Classes to Read Metadata from a Database

Reference

CDynamicAccessor Class
CDynamicStringAccessor Class
CDynamicParameterAccessor Class

Concepts

Using Accessors

Overriding a Dynamic Accessor

When you use a dynamic accessor such as **CDynamicAccessor**, the command **Open** method creates an accessor for you automatically, based on the column information of the opened rowset. You can override the dynamic accessor to control exactly how the columns are bound.

To override the dynamic accessor, pass **false** as the last parameter to the **CCommand::Open** method. This prevents **Open** from creating an accessor automatically. You can then call **GetColumnInfo** and call **AddBindEntry** for each column that you want to bind. The following code shows how to do it:

```
USES_CONVERSION;
double   dblProductID;

CCommand<CDynamicAccessor> product;
// Open the table, passing false to prevent automatic binding
product.Open(session, _T("Select * FROM Products"), NULL, NULL, DBGUID_DEFAULT, false);

ULONG         nColumns;
DBCOLUMNINFO*   pColumnInfo;
// Get the column information from the opened rowset.
product.GetColumnInfo(&nColumns, &pColumnInfo);

// Bind the product ID as a double.
pColumnInfo[0].wType          = DBTYPE_R8;
pColumnInfo[0].ulColumnSize = 8;
product.AddBindEntry(pColumnInfo[0]);

// Bind the product name as it is.
product.AddBindEntry(pColumnInfo[1]);

// Bind the reorder level as a string.
pColumnInfo[8].wType          = DBTYPE_STR;
pColumnInfo[8].ulColumnSize = 10;
product.AddBindEntry(pColumnInfo[8]);

// You have finished specifying the bindings. Go ahead and bind.
product.Bind();
// Free the memory for the column information that was retrieved in
// previous call to GetColumnInfo.
CoTaskMemFree(pColumnInfo);

char*   pszProductName;
char*   pszReorderLevel;
bool   bRC;

// Loop through the records tracing out the information.
while (product.MoveNext() == S_OK)
{
   bRC = product.GetValue(1, &dblProductID);
   pszProductName   = (char*)product.GetValue(2);
   pszReorderLevel  = (char*)product.GetValue(9);

   ATLTRACE(_T("Override = %lf \"%s\" \"%s\"\n"), dblProductID,
      A2T(pszProductName), A2T(pszReorderLevel));
}
```

See Also

Concepts

Using Accessors

Using Manual Accessors

There are four things to do when handling an unknown command:

- Determine the parameters
- Execute the command
- Determine the output columns
- See if there are multiple return rowsets

To do this with the OLE DB Consumer Templates, use the **CManualAccessor** class and follow these steps:

1. Open a **CCommand** object with **CManualAccessor** as a template parameter.

   ```
   CCommand<CManualAccessor, CRowset, CMultipleResults> rs;
   ```

2. Query the session for the **IDBSchemaRowset** interface and use the procedure parameters rowset. If the **IDBSchemaRowset** interface is not available, query for the **ICommandWithParameters** interface. Call **GetParameterInfo** for information. If neither interface is available, you can assume there are no parameters.
3. For each parameter, call **AddParameterEntry** to add the parameters and set them.
4. Open the rowset but set the bind parameter to **false**.
5. Call **GetColumnInfo** to retrieve the output columns. Use **AddBindEntry** to add the output column to the binding.
6. Call **GetNextResult** to determine if more rowsets are available. Repeat steps 2 through 5.

For an example of a manual accessor, see **CDBListView::CallProcedure** in the DBVIEWER sample.

See Also
Concepts
Using Accessors

Accessing XML Data

There are two separate methods of retrieving XML data from a data source: one uses CStreamRowset and the other uses CXMLAccessor.

Functionality	CStreamRowset	CXMLAccessor
Amount of data transferred	Retrieves data from all columns and rows at once.	Retrieves data from all columns but only one row at a time. You must navigate rows using methods such as **MoveNext**.
Formatting the string	SQL Server formats the XML string and sends it to the consumer.	Retrieves rowset data in its native format (requests that the provider send it as Unicode strings) and then builds the string containing the data in XML format.
Control over formatting	You have some level of control over how the XML string is formatted by setting some SQL Server 2000-specific properties.	You have no control over the format of the generated XML string.

While **CStreamRowset** provides a more overall efficient way of retrieving data in XML format, it is only supported by SQL Server 2000.

Retrieving XML Data Using CStreamRowset

You specify CStreamRowset as the rowset type in your **CCommand** or **CTable** declaration. You can use it with your own accessor or no accessor, for example:

```
CCommand< CAccessor<CMyAccessor>, CStreamRowset > myCmd;
```

-or-

```
CCommand< CNoAccessor, CStreamRowset > myCmd;
```

Normally when you call **CCommand::Open** (specifying, for example, **CRowset** as the *TRowset* class), it obtains an **IRowset** pointer. **ICommand::Execute** returns an **IRowset** pointer, which is stored in the **m_spRowset** member of the **CRowset** object. Methods such as **MoveFirst**, **MoveNext**, and **GetData** use that pointer to retrieve the data.

By contrast, when you call **CCommand::Open** (but specify **CStreamRowset** as the *TRowset* class), **ICommand::Execute** returns an ISequentialStream pointer, which is stored in the **m_spStream** data member of CStreamRowset. You then use the **Read** method to retrieve the (Unicode string) data in XML format. For example:

```
myCmd.m_spStream->Read()
```

SQL Server 2000 performs the XML formatting and returns all columns and all rows of the rowset as one XML string.

For an example using the **Read** method, see "Adding XML Support to the Consumer" in Implementing a Simple Consumer.

Note
XML support using **CStreamRowset** works with SQL Server 2000 only and requires that you have the OLE DB Provider for SQL Server 2000 (installed with MDAC).

Retrieving XML Data Using CXMLAccessor

CXMLAccessor allows you to access data from a data source as string data when you have no knowledge of the data store's schema. **CXMLAccessor** works like **CDynamicStringAccessorW** except that the former converts all data accessed from the data store as XML-formatted (tagged) data. The XML tag names match the data store's column names as closely as possible.

Use **CXMLAccessor** as you would any other accessor class, passing it as a template parameter to **CCommand** or **CTable**:

```
CTable<CXMLAccessor, CRowset> rs;
```

Use GetXMLRowData to retrieve data from the table one row at a time, and navigate rows using methods such as **MoveNext**, for example:

```
// Open data source, session, and rowset
hr = rs.MoveFirst();
while( SUCCEEDED(hr) && hr != DB_S_ENDOFROWSET )
{
    CStringW strRowData;
    myCmd.GetXMLRowData(strRowData);

    printf_s( "%S\n", strRowData );

    hr = rs.MoveNext();
}
```

You can use GetXMLColumnData to retrieve the column (data type) information as XML-formatted string data.

See Also

Concepts

Using Accessors

Obtaining Metadata with Schema Rowsets

Sometimes you need to obtain information about the provider, rowset, table, columns, or other database information without opening the rowset. Data about the database structure is called metadata and can be retrieved by a number of different methods. One method is to use schema rowsets.

OLE DB Templates provide a set of classes to retrieve schema information; these classes create predefined schema rowsets and are listed in Schema Rowset Classes and Typedef Classes.

Note
If you are using OLAP and some of your rowsets are not supported by the schema rowset classes (for example, you have a variable number of columns), you should consider using **CManualAccessor** or **CDynamicAccessor**. You can scroll through the columns and use case statements to handle the possible data types for each column.

Catalog/Schema Model

ANSI SQL defines a catalog/schema model for data stores; OLE DB uses this model. In this model, catalogs (databases) contain schemas and schemas contain tables.

- **Catalog** A catalog is another name for a database. It is a collection of related schemas. To list the catalogs (databases) belonging to a given data source, use CCatalog. Because many databases have only one catalog, metadata is sometimes simply called schema information.
- **Schema** A schema is a collection of database objects that are owned or have been created by a particular user. To list the schemas owned by a given user, use CSchemata.

  In Microsoft SQL Server and ODBC 2.x terms, a schema is an owner (for example, dbo is a typical schema name). Also, SQL Server stores metadata in a set of tables: one table contains a list of all the tables and another table contains a list of all the columns. There is no equivalent to a schema in a Microsoft Access database.
- **Table** Tables are collections of columns arranged in specific orders. To list the tables defined in a given catalog (database) and information about those tables, use CTables).

Restrictions

When you query for schema information, you can use restrictions to specify the type of information in which you are interested. You can think of restrictions as a filter or qualifier in a query. For example, in the query:

```
SELECT * FROM authors where l_name = 'pivo'
```

`l_name` is a restriction. This is a very simple example with only one restriction; the schema rowset classes support several restrictions.

The schema rowset typedef classes encapsulate all the OLE DB schema rowsets so that you can access a schema rowset just like any other rowset by instantiating and opening it. For example, the typedef class CColumns is defined as:

```
CRestrictions<CAccessor<CColumnsInfo>
```

The CRestrictions class supplies the restriction support. After you create an instance of the schema rowset, call CRestrictions::Open. This method returns a result set based on the restrictions that you specify.

To specify restrictions, refer to Appendix B: Schema Rowsets and look up the rowset that you are using. For example, **CColumns** corresponds to the COLUMNS Rowset; that topic lists the restriction columns in the COLUMNS rowset: TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME. You must follow that order in specifying your restrictions.

So, for example, if you want to restrict by table name, note that TABLE_NAME is the third restriction column, and then call **Open**, specifying the desired table name as the third restriction parameter, as shown in the following example.

To use schema rowsets

1. You must include the header file Atldbsch.h (of course, you need Atldbcli.h for consumer support as well).

2. Instantiate a schema rowset object in the consumer's or the document's header file. If you want table information, declare a **CTables** object; if you want column information, declare a **CColumns** object. This example shows how to retrieve the columns in the authors table:

```
CDataSource ds;
ds.Open();
CSession ss;
ss.Open();
CColumns ColumnSchemaRowset;
// TABLE_NAME is the third restriction column, so
// specify "authors" as the third restriction parameter:
hr = ColumnSchemaRowset.Open(ss, NULL, NULL, "authors");
hr = ColumnSchemaRowset.MoveFirst();
while (hr == S_OK)
{
   hr = ColumnSchemaRowset.MoveNext();
}
```

3. To fetch the information, access the appropriate data member of the schema rowset object, for example, `ColumnSchemaRowset.m_szColumnName`. This corresponds to COLUMN_NAME. To see which OLE DB column each data member corresponds to, see CColumns.

For the reference of the schema rowset, typedef classes provided in the OLE DB Templates (see Schema Rowset Classes and Typedef Classes).

For more information about OLE DB schema rowsets, including restriction columns, see Appendix B: Schema Rowsets in the OLE DB Programmer's Reference.

For more complex examples of how to use schema rowset classes, see the CatDB and DBViewer samples.

For information about provider support for schema rowsets, see Supporting Schema Rowsets.

See Also

Concepts

Using Accessors

Supporting Transactions in OLE DB

A transaction is a way to group, or batch, a series of updates to a data source so that either all succeed and are committed at once or (if any one of them fails) none are committed and the entire transaction is rolled back. This process ensures the integrity of the result on the data source.

OLE DB supports transactions with the following three methods:

- ITransactionLocal::StartTransaction
- ITransaction::Commit
- ITransaction::Abort

Relationship of Sessions and Transactions

A single data source object can create one or more session objects, each of which can be inside or outside the scope of a transaction at a given time.

When a session does not enter a transaction, all work done within that session on the data store is immediately committed on each method call. (This is sometimes referred to as autocommit mode or implicit mode.)

When a session enters a transaction, all work done within that session on the data store is part of that transaction and is committed or aborted as a single unit. (This is sometimes referred to as manual-commit mode.)

Transaction support is provider-specific. If the provider you are using supports transactions, a session object that supports **ITransaction** and **ITransactionLocal** can enter a simple (that is, non-nested) transaction. The OLE DB Templates class CSession supports these interfaces and is the recommended way to implement transaction support in Visual C++.

Starting and Ending the Transaction

You call the **StartTransaction**, **Commit**, and **Abort** methods in the rowset object in the consumer.

Calling **ITransactionLocal::StartTransaction** starts a new local transaction. When you start the transaction, any changes mandated by subsequent operations are not actually applied to the data store until you commit the transaction.

Calling **ITransaction::Commit** or **ITransaction::Abort** ends the transaction. **Commit** causes all changes within the scope of the transaction to be applied to the data store. **Abort** causes all changes within the scope of the transaction to be canceled and the data store is left in the state it had before the transaction started.

Nested Transactions

A nested transaction occurs when you start a new local transaction when an active transaction already exists on the session. The new transaction is started as a nested transaction below the current transaction. If the provider does not support nested transactions, calling **StartTransaction** when there is already an active transaction on the session returns **XACT_E_XTIONEXISTS**.

Distributed Transactions

A distributed transaction is a transaction that updates distributed data; that is, data on more than one networked computer system. If you want to support transactions over a distributed system, you should use the .NET Framework rather than the OLE DB transaction support.

For information about transactions in the Microsoft .NET Framework, see Processing Transactions in the .NET Framework SDK.

See Also

Concepts

Using Accessors

Using OLE DB Record Views

If you want to display OLE DB rowset data in an MFC application, you should use the MFC class COleDBRecordView. A record view object created from **COleDBRecordView** allows you to display database records in MFC controls. The record view is a dialog form view directly connected to an OLE DB Rowset object created from the **CRowset** template class. Obtaining a handle to the rowset object is simple:

```
COleDBRecordView myRecordView;
...
// CProductAccessor is a user record class
CRowset<CAccessor<CProductAccessor>> myRowSet = myRecordView.OnGetRowset();
```

The view displays the fields of the **CRowset** object in the dialog's controls. The **COleDBRecordView** object uses Dialog Data Exchange (DDX) and the navigational functionality built into **CRowset** (**MoveFirst**, **MoveNext**, **MovePrev**, and **MoveLast**) to automate the movement of data between the controls on the form and the fields of the rowset. **COleDBRecordView** keeps track of the user's position in the rowset so that the record view can update the user interface and supplies an OnMove method for updating the current record before moving to another.

You can use DDX functions with **COleDbRecordView** to get data directly from the database recordset and display it in a dialog control. You should use the **DDX_*** methods (such as **DDX_Text**), not the **DDX_Field*** functions (such as **DDX_FieldText**) with **COleDbRecordView**.

See Also

Reference

COleDBRecordView Class

Concepts

Using Accessors

Using an Existing ADO Recordset

To mix OLE DB consumer templates and Active Data Objects (ADO), use ADO to open a recordset (corresponding to a rowset in the OLE DB Consumer Templates). When you have a recordset, do the following to connect to an OLE DB rowset:

1. Call **QueryInterface** for the **IRowset** and **IAccessor** pointers.

```
IRowset* lpRowset = NULL;
IAccessor* lpAccessor = NULL;
lpUnk->QueryInterface(IID_IRowset, (void**)&lpRowset);
lpUnk->QueryInterface(IID_IAccessor, (void**)&lpAccessor);
```

Note

lpUnk points to the **IUnknown** object of the ADO recordset.

2. Attach the accessor and rowset to their appropriate OLE DB consumer template classes.

```
CRowset rs;
CAccessor accessor;

accessor.AddAccessorInfo(0ul);      // 0 is the ordinal of an ADO accessor
rs.m_spRowset.Attach(lpRowset);      // use the Attach method of CComPtr<>
rs.SetAccessor(accessor);
```

See Also

Concepts

Using Accessors

Updating a Column When Another Table Contains a Reference to the Row

Some providers can detect which columns in the row change, but many providers cannot. As a result, updating a column can result in an error when there is a reference to the row you are attempting to update. To solve this problem, create a separate accessor containing only the columns you want to change. Pass the number of that accessor to **SetData**.

See Also

Concepts

Using Accessors

Using Bookmarks

Before you open the rowset, you must tell the provider that you want to use bookmarks. To do this, set the **DBPROP_BOOKMARKS** property to **true** in your property set. The provider retrieves bookmarks as column zero, so you must use the special macro **BOOKMARK_ENTRY** and the **CBookmark** class if you are using a static accessor. **CBookmark** is a template class where the argument is the length in bytes of the bookmark buffer. The length of the buffer required for a bookmark depends on the provider. If you are using the ODBC OLE DB provider, as shown in the following example, the buffer must be 4 bytes.

```
class CProducts
{
public:
   CBookmark<4>   bookmark;

   BEGIN_COLUMN_MAP(CProducts)
      BOOKMARK_ENTRY(bookmark)
   END_COLUMN_MAP()
};

CDBPropSet propset(DBPROPSET_ROWSET);
propset.AddProperty(DBPROP_BOOKMARKS, true);

CTable<CAccessor<CProducts> > product;
product.Open(session, "Products", &propset);
```

If you use **CDynamicAccessor**, the buffer is dynamically allocated at run time. In this case, you can use a specialized version of **CBookmark** for which you do not specify a buffer length. Use the function **GetBookmark** to retrieve the bookmark from the current record, as shown in this code sample:

```
CTable<CDynamicAccessor> product;
CBookmark<>              bookmark;
CDBPropSet propset(DBPROPSET_ROWSET);

propset.AddProperty(DBPROP_BOOKMARKS, true);
product.Open(session, "Products", &propset);
product.MoveNext();
product.GetBookmark(&bookmark);
```

For information about supporting bookmarks in providers, see Provider Support for Bookmarks.

See Also

Concepts

Using Accessors

Retrieving a BLOB

You can retrieve a binary large object (BLOB) in various ways. You can use **DBTYPE_BYTES** to retrieve the BLOB as a sequence of bytes or use an interface like **ISequentialStream**. For more information, see BLOBS and OLE Objects in the *OLE DB Programmer's Reference*.

The following code shows how to retrieve a BLOB using **ISequentialStream**. The macro BLOB_ENTRY allows you to specify the interface and the flags used for the interface. After opening the table, the code calls **Read** repeatedly on **ISequentialStream** to read bytes from the BLOB. The code calls **Release** to dispose of the interface pointer before calling **MoveNext** to obtain the next record.

```
class CCategories
{
public:
   ISequentialStream*   pPicture;

BEGIN_COLUMN_MAP(CCategories)
   BLOB_ENTRY(4, IID_ISequentialStream, STGM_READ, pPicture)
END_COLUMN_MAP()
};

CTable<CAccessor<CCategories> > categories;
ULONG          cb;
BYTE            myBuffer[65536];

categories.Open(session, "Categories");
while (categories.MoveNext() == S_OK)
{
   do
   {
      categories.pPicture->Read(myBuffer, 65536, &cb);
      // Do something with the buffer
   } while (cb > 0);
   categories.pPicture->Release();
}
```

For more information about macros that handle BLOB data, see "Column Map Macros" in Macros and Global Functions for OLE DB Consumer Templates.

See Also

Concepts

Using Accessors

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

Receiving Notifications

OLE DB provides interfaces for receiving notifications when events occur. These are described in OLE DB Object Notifications in the *OLE DB Programmer's Reference*. Setup of these events uses the standard COM connection-point mechanism. For example, an ATL object that wants to retrieve events through **IRowsetNotify** implements the **IRowsetNotify** interface by adding **IRowsetNotify** to the class-derived list and exposing it through a **COM_INTERFACE_ENTRY** macro.

IRowsetNotify has three methods, which can be called at various times. If you want to respond to only one of these methods, you can use the IRowsetNotifyImpl class, which returns **E_NOTIMPL** for the methods you are not interested in.

When you create the rowset, you must tell the provider that you want the returned rowset object to support **IConnectionPointContainer**, which is needed to set up the notification.

The following code shows how to open the rowset from an ATL object and use the **AtlAdvise** function to set up the notification sink. **AtlAdvise** returns a cookie that is used when you call **AtlUnadvise**.

```
CDBPropSet propset(DBPROPSET_ROWSET);
propset.AddProperty(DBPROP_IConnectionPointContainer, true);

product.Open(session, _T("Products"), &propset);

AtlAdvise(product.m_spRowset, GetUnknown(), IID_IRowsetNotify, &m_dwCookie);
```

See Also

Concepts

Using Accessors

OLE DB Provider Templates (C++)

OLE DB is an important part of Microsoft's Universal Data Access strategy. The OLE DB design allows high-performance data access from any data source. Any tabular data is viewable through OLE DB regardless of whether it came from a database. The flexibility gives you a tremendous amount of power.

As explained in OLE DB Consumers and Providers, OLE DB uses the concept of consumers and providers. The consumer makes requests for data; the provider returns data in a tabular format to the consumer. From a programming perspective, the most important implication of this model is that the provider must implement any call the consumer can make.

What Is a Provider?

An OLE DB provider is a set of COM objects that serve interface calls from a consumer object, transferring data in a tabular format from a durable source (called a data store) to the consumer.

Providers can be simple or complex. The provider can support a minimal amount of functionality or a full-blown production quality provider by implementing more interfaces. A provider can return a table, allow the client to determine the format of that table, and perform operations on that data.

Each provider implements a standard set of COM objects to handle requests from the client, with standard meaning that any OLE DB consumer can access data from any provider, regardless of language (such as C++ and Basic).

Each COM object contains several interfaces, some of which are required and some of which are optional. By implementing the mandatory interfaces, a provider guarantees a minimum level of functionality (called compliance) that any client should be able to use. A provider can implement optional interfaces to provide additional functionality.
The OLE DB Provider Template Architecture describes these interfaces in detail. The client should always call **QueryInterface** to determine if a provider supports a given interface.

OLE DB Specification Level Support

The OLE DB provider templates support the OLE DB version 2.7 specification. Using the OLE DB provider templates, you can implement a level 0 compliant provider. (For an explanation of level 0, see "Minimum Provider Functionality" in "OLE DB Leveling: Choosing the Right Interfaces" at http://www.microsoft.com/data/oledb/techinfo/oledbleveling2.htm.) The Provider sample, for example, uses the templates to implement a non-SQL (MS-DOS) command server that executes the DOS DIR command to query the file system. The Provider sample returns the directory information in a rowset, which is the standard OLE DB mechanism for returning tabular data.

The simplest type of provider supported by the OLE DB templates is a read-only provider with no commands. Providers with commands are also supported, as are bookmarking and read/write capability. You can implement a read/write provider by writing additional code. Dynamic rowsets and transactions are not supported by the current version, but you can add them if you want.

When Do You Need to Create an OLE DB Provider?

You do not always need to create your own provider; Microsoft provides several prepackaged, standard providers in the **Data Link Properties** dialog box in Visual C++. The main reason to create an OLE DB provider is to take advantage of the Universal Data Access strategy. Some of advantages of doing so are:

- Accessing data through any language such as C++, Basic, and Visual Basic Scripting Edition. It allows different programmers in your organization to access the same data in the same way, regardless of what language they use.
- Exposing your data to other data sources such as SQL Server, Excel, and Access. This can be very useful if you want to transfer data among different formats.
- Participating in cross–data source (heterogeneous) operations. This can be a very effective way of data warehousing. By using OLE DB providers, you can keep data in its native format and still be able to access it in a simple operation.
- Adding additional capabilities to your data, such as query processing.
- Increasing performance of accessing data by controlling how it is manipulated.
- Increasing robustness. If you have a proprietary data format that only one programmer can access, you are at risk. Using OLE DB providers, you can open that proprietary format to all your programmers.

Read-Only and Updatable Providers

Providers can vary greatly in complexity and functionality. It is useful to categorize providers into read-only providers and updatable providers:

- Visual C++ 6.0 supported only read-only providers. Creating an OLE DB Provider discusses how to create a read-only provider.
- Visual C++ .NET supports updatable providers, which can update (write to) the data store. For information about updatable providers, see Creating an Updatable Provider; the UpdatePV sample is an example of an updatable provider.

For more information, see:

- The OLE DB Provider Template Architecture
- Creating an OLE DB Provider
- OLE DB Programming

See Also

Reference

OLE DB SDK Documentation
OLE DB Programmer's Reference

Other Resources

Data Access (C++)

The OLE DB Provider Template Architecture

Data Sources and Sessions

The OLE DB provider architecture includes a data source object and one or more sessions. The data source object is the initial object that every provider must instantiate. When a consumer application needs data, it co-creates the data source object to start the provider. The data source object creates a session object (using the **IDBCreateSession** interface) through which the consumer connects to the data source object. ODBC programmers can think of the data source object as being equivalent to the **HENV** and the session object as equivalent to the **HDBC**.

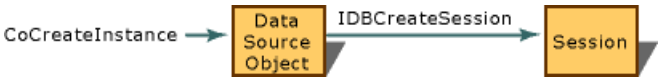

Together with the source files created by the OLE DB Provider Wizard, the OLE DB templates implement a data source object. A session is an object that corresponds to the OLE DB **TSession**.

Mandatory and Optional Interfaces

The OLE DB provider templates give you prepackaged implementations for all the required interfaces. Mandatory and optional interfaces are defined by OLE DB for several types of objects:

- Data source
- Session
- Rowset
- Command
- Transaction

Note that the OLE DB provider templates do not implement the row and storage objects.

The following table lists mandatory and optional interfaces for the objects listed above, according to the OLE DB 2.6 SDK Documentation.

Component	Interface	Comment

Data source (CDataSource)	[mandatory] **IDBCreateSession** [mandatory] **IDBInitialize** [mandatory] **IDBProperties** [mandatory] **IPersist** [optional] **IConnectionPointContainer** [optional] **IDBAsynchStatus** [optional] **IDBDataSourceAdmin** [optional] **IDBInfo** [optional] **IPersistFile** [optional] **ISupportErrorInfo**	Connection from the consumer to the provider. The object is used to specify properties on the connection such as user ID, password, and data source name. The object can also be used to administer a data source (create, update, delete, tables, and so on).

Session (CSession)	[mandatory] **IGetDataSource** [mandatory] **IOpenRowset** [mandatory] **ISessionProperties** [optional] **IAlterIndex** [optional] **IAlterTable** [optional] **IBindResource** [optional] **ICreateRow** [optional] **IDBCreateCommand** [optional] **IDBSchemaRowset** [optional] **IIndexDefinition** [optional] **ISupportErrorInfo** [optional] **ITableCreation** [optional] **ITableDefinition** [optional] **ITableDefinitionWithConstraints** [optional] **ITransaction** [optional] **ITransactionJoin** [optional] **ITransactionLocal** [optional] **ITransactionObject**	The session object represents a single conversation between a consumer and provider. It is somewhat similar to the ODBC **HSTMT** in that there can be many simultaneous sessions active. The session object is the primary link to get to OLE DB functionality. To get to a command, transaction, or rowset object, you go through the session object.
Rowset (CRowset)	[mandatory] **IAccessor** [mandatory] **IColumnsInfo** [mandatory] **IConvertType** [mandatory] **IRowset** [mandatory] **IR**	The rowset object represents the data from the data source. The object is responsible for the bindings of that data and any basic operations (update, fetch, movement, and others) on the data. You always have a rowset object to contain and manipulate data.

	owsetInfo [optional] **IChapteredRowset** [optional] **IColumnsInfo2** [optional] **IColumnsRowset** [optional] **IConnectionPointContainer** [optional] **IDBAsynchStatus** [optional] **IGetRow** [optional] **IRowsetChange** [optional] **IRowsetChapterMember** [optional] **IRowsetCurrentIndex** [optional] **IRowsetFind** [optional] **IRowsetIdentity** [optional] **IRowsetIndex** [optional] **IRowsetLocate** [optional] **IRowsetRefresh** [optional] **IRowsetScroll** [optional] **IRowsetUpdate** [optional] **IRowsetView** [optional] **ISupportErrorInfo** [optional] **IRowsetBookmark**	

Command (CCommand)	[mandatory] **IAccessor** [mandatory] **IColumnsInfo** [mandatory] **ICommand** [mandatory] **ICommandProperties** [mandatory] **ICommandText** [mandatory] **IConvertType** [optional] **IColumnsRowset** [optional] **ICommandPersist** [optional] **ICommandPrepare** [optional] **ICommandWithParameters** [optional] **ISupportErrorInfo** [optional] **ICommandStream**	The command object handles operations on data such as queries. It can handle parameterized or non-parameterized statements. The command object is also responsible for handling bindings for parameters and output columns. A binding is a structure that contains information about how a column, in a rowset, should be retrieved. It contains information such as ordinal, data type, length, and status.
Transaction (optional)	[mandatory] **IConnectionPointContainer** [mandatory] **ITransaction** [optional] **ISupportErrorInfo**	The transaction object defines an atomic unit of work on a data source and determines how those units of work relate to each other. This object is not directly supported by the OLE DB provider templates (that is, you create your own object).

For more information, see the following topics:

- Property Maps
- The User Record

See Also

Reference

OLE DB Interfaces

Concepts

OLE DB Provider Templates (C++)

Data Source Object Interfaces

The following table shows the mandatory and optional interfaces defined by OLE DB for a data source object.

Interface	Required?	Implemented by OLE DB templates?
IDBCreateSession	Mandatory	Yes
IDBInitialize	Mandatory	Yes
IDBProperties	Mandatory	Yes
IPersist	Mandatory	Yes
IConnectionPointContainer	Optional	No
IDBDataSourceAdmin	Optional	No
IDBInfo	Optional	No
IPersistFile	Optional	No
ISupportErrorInfo	Optional	No

The data source object implements the **IDBProperties**, **IDBInitialize**, and **IDBCreateSession** interfaces through inheritance. You can choose to support additional functionality by inheriting or not inheriting from one of these implementation classes. If you want to support the **IDBDataSourceAdmin** interface, you must inherit from the **IDBDataSourceAdminImpl** class.

See Also

Concepts

The OLE DB Provider Template Architecture

Session Object Interfaces

The following table shows the mandatory and optional interfaces defined by OLE DB for a session object.

Interface	Required?	Implemented by OLE DB templates?
IGetDataSource	Mandatory	Yes
IOpenRowset	Mandatory	Yes
ISessionProperties	Mandatory	Yes
IAlterIndex	Optional	No
IAlterTable	Optional	No
IBindResource	Optional	No
ICreateRow	Optional	No
IDBCreateCommand	Optional	Yes
IDBSchemaRowset	Optional	Yes
IIndexDefinition	Optional	No
ISupportErrorInfo	Optional	Yes
ITableCreation	Optional	No
ITableDefinition	Optional	No
ITableDefinitionWithConstraints	Optional	No
ITransaction	Optional	No
ITransactionJoin	Optional	No
ITransactionLocal	Optional	No
ITransactionObject	Optional	No

The session object creates a rowset object. If the provider supports commands, the session also creates a command object (**CCommand**, implementing the OLE DB **TCommand**). The command object implements the **ICommand** interface and uses the **ICommand::Execute** method to execute commands on the rowset, as shown in the following figure.

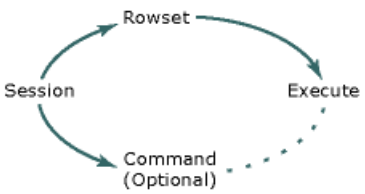

See Also

Concepts

The OLE DB Provider Template Architecture

Rowset Object Interfaces

The following table shows the mandatory and optional interfaces defined by OLE DB for a rowset object.

Interface	Required?	Implemented by OLE DB templates?
IAccessor	Mandatory	Yes
IColumnsInfo	Mandatory	Yes
IConvertType	Mandatory	Yes
IRowset	Mandatory	Yes
IRowsetInfo	Mandatory	Yes
IChapteredRowset	Optional	No
IColumnsInfo2	Optional	No
IColumnsRowset	Optional	No
IConnectionPointContainer	Optional	Yes (through ATL)
IDBAsynchStatus	Optional	No
IGetRow	Optional	No
IRowsetChange	Optional	Yes
IRowsetChapterMember	Optional	No
IRowsetCurrentIndex	Optional	No
IRowsetFind	Optional	No
IRowsetIdentity	Optional (but required for level 0 providers)	Yes
IRowsetIndex	Optional	No
IRowsetLocate	Optional	Yes
IRowsetRefresh	Optional	No
IRowsetScroll	Optional	No
IRowsetUpdate	Optional	Yes
IRowsetView	Optional	No
ISupportErrorInfo	Optional	Yes
IRowsetBookmark	Optional	No

The wizard-generated rowset object implements **IAccessor**, **IRowset**, and **IRowsetInfo** through inheritance. The **IAccessorImpl** binds both output columns. The **IRowset** interface handles fetches rows and data. The **IRowsetInfo** interface handles the rowset properties.

See Also

Concepts

The OLE DB Provider Template Architecture

Command Object Interfaces

The following table shows the mandatory and optional interfaces defined by OLE DB for a command object.

Interface	Required?	Implemented by OLE DB templates?
IAccessor	Mandatory	Yes
IColumnsInfo	Mandatory	Yes
ICommand	Mandatory	Yes
ICommandProperties	Mandatory	Yes
ICommandText	Mandatory	Yes
IConvertType	Mandatory	Yes
IColumnsRowset	Optional	No
ICommandPersist	Optional	No
ICommandPrepare	Optional	No
ICommandWithParameters	Optional	No
ISupportErrorInfo	Optional	No
ICommandStream	Optional	No

The command object uses the **IAccessor** to specify parameter bindings. The consumer calls **IAccessor::CreateAccessor**, passing it an array of **DBBINDING** structures. **DBBINDING** contains information about the column bindings (such as type and length). The provider receives the structures and determines how the data should be transferred and whether conversions are necessary.

The **ICommandText** interface provides a way to specify a text command. The **ICommandProperties** interface handles all the command properties.

See Also

Concepts

The OLE DB Provider Template Architecture

Transaction Object Interfaces

The transaction object defines an atomic unit of work on a data source and determines how those units of work relate to each other. This object is not directly supported by the OLE DB provider templates (that is, you must create your own object).

The following table shows the mandatory and optional interfaces defined by OLE DB for a transaction object.

Interface	Required?	Implemented by OLE DB templates?
IConnectionPointContainer	Mandatory	No
ITransaction	Mandatory	No
ISupportErrorInfo	Optional	No

See Also

Concepts

The OLE DB Provider Template Architecture

Property Maps

In addition to the session, rowset, and optional command object, each provider supports one or more properties. These properties are defined in property maps contained in the header files created by the OLE DB Provider Wizard. Each header file contains a map for the properties in the OLE DB property group defined for the object or objects defined in that file. The header file that contains the data source object also contains the property map for the DataSource properties. Session.h contains the property map for the Session properties. The rowset and command objects reside in a single header file, called *projectname*RS.h. These properties are members of the Rowset properties group.

See Also

Concepts

The OLE DB Provider Template Architecture

The User Record

The user record provides the code and data structure that represents the column data for a rowset. A user record can be created at compile time or at run time. When you create a provider using the ATL OLE DB Provider Wizard, the wizard creates a default user record that looks like this (assuming you specified a provider name [short name] of "MyProvider"):

```
class CWindowsFile:
   public WIN32_FIND_DATA
{
public:

BEGIN_PROVIDER_COLUMN_MAP(CMyProviderWindowsFile)
   PROVIDER_COLUMN_ENTRY("FileAttributes", 1, dwFileAttributes)
   PROVIDER_COLUMN_ENTRY("FileSizeHigh", 2, nFileSizeHigh)
   PROVIDER_COLUMN_ENTRY("FileSizeLow", 3, nFileSizeLow)
   PROVIDER_COLUMN_ENTRY_STR("FileName", 4, cFileName)
   PROVIDER_COLUMN_ENTRY_STR("AltFileName", 5, cAlternateFileName)
END_PROVIDER_COLUMN_MAP()

};
```

The OLE DB provider templates handle all OLE DB specifics regarding interactions with the client. To acquire the column data needed for a response, the provider calls the **GetColumnInfo** function, which you must place in the user record. You can explicitly override **GetColumnInfo** in the user record, for example, by declaring it in the .h file as shown here:

```
template <class T>
static ATLCOLUMNINFO* GetColumnInfo(T* pThis, ULONG* pcCols)
```

This is equivalent to:

```
static ATLCOLUMNINFO* GetColumnInfo(CommandClass* pThis, ULONG* pcCols)
static ATLCOLUMNINFO* GetColumnInfo(RowsetClass* pThis, ULONG* pcCols)
```

You must also implement **GetColumnInfo** in the user record's .cpp file.

The PROVIDER_COLUMN_MAP macros aid in creating a **GetColumnInfo** function:

- BEGIN_PROVIDER_COLUMN_MAP defines the function prototype and a static array of **ATLCOLUMNINFO** structures.
- PROVIDER_COLUMN_ENTRY defines and initializes a single **ATLCOLUMNINFO**.
- END_PROVIDER_COLUMN_MAP closes the array and the function. It also places the array element count into the *pcCols* parameter.

When a user record is created at run time, **GetColumnInfo** uses the `pThis` parameter to receive a pointer to a rowset or command instance. Commands and rowsets must support the **IColumnsInfo** interface, so column information can be obtained from this pointer.

CommandClass and **RowsetClass** are the command and rowset that use the user record.

For a more detailed example of how to override **GetColumnInfo** in a user record, see Dynamically Determining Columns Returned to the Consumer.

See Also

Concepts

The OLE DB Provider Template Architecture

Creating an OLE DB Provider

The recommended way to create an OLE DB provider is to use the wizards to create an ATL COM project and a provider and then modify the files using the OLE DB templates. As you customize your provider, you can comment out unwanted properties and add optional interfaces.

The basic steps are as follows:

1. Use the ATL Project Wizard to create the basic project files and the ATL OLE DB Provider Wizard to create the provider (select **ATL OLE DB Provider** from the Visual C++ folder in **Add Class**).
2. Modify the code in the `Execute` method in CMyProviderRS.h. For an example, see Reading Strings Into an OLE DB Provider.
3. Edit the property maps in MyProviderDS.h, MyProviderSess.h, and MyProviderRS.h. The wizard creates property maps that contain all properties that a provider might implement. Go through the property maps and remove or comment out properties that your provider does not need to support.
4. Update the PROVIDER_COLUMN_MAP, which can be found in MyProviderRS.h. For an example, see Storing Strings In the OLE DB Provider.
5. When you are ready to test your provider, you can test it by trying to find the provider in a provider enumeration. For examples of test code that finds a provider in an enumeration, see the CATDB and DBVIEWER samples or the example in Implementing A Simple Consumer.
6. Add any additional interfaces you want. For an example, see Enhancing the Simple Read-Only Provider.

Note
By default, the wizards generate code that is OLE DB level 0 compliant. To ensure that your application remains level 0 compliant, do not remove any of the wizard-generated interfaces from the code.

See Also

Tasks

CatDB Sample: Data Source Schema Browser

DBViewer Sample: Database Browser

Creating a Project for the Provider

To create a project in which the OLE DB provider will reside

1. From the **File** menu, click **New**, and then click **Project**.

   The **New Project** dialog box appears.

2. In the Project Types pane, click the Visual C++ Projects folder. In the Templates pane, click **ATL Project**.

3. In the **Name** box, enter a name for the project, and then click **OK**.

   The ATL Project Wizard appears.

4. In the ATL Project Wizard, choose **Dynamic-Link Library (DLL)** for **Server Type**.

5. Click **Finish**.

See Also

Reference

Creating an OLE DB Provider

Creating the Provider

To create an OLE DB provider with the ATL OLE DB Provider Wizard

1. Right-click the project.
2. On the shortcut menu, click **Add**, and then click **Add Class**.
3. In the **Add Class** dialog box, select the **ATL OLE DB Provider** icon, and then click **Open**.
4. In the ATL OLE DB Provider Wizard, enter a short name for your provider in the **Short Name** box. The following topics use the short name "MyProvider", but you can use another name. The other name boxes populate according to the name you enter.
5. Edit the other name boxes, if needed. In addition to the object and file names, you can edit the following:
   - **Coclass**: The name that COM uses to create the provider.
   - **ProgID**: The programmatic identifier, which is a text string that can be used instead of a GUID.
   - **Version**: Used with the ProgID and coclass to generate a version-dependent programmatic ID.
6. Click **Finish**.

See Also

Reference

Creating an OLE DB Provider

Provider Wizard-Generated Files

The ATL OLE DB Provider Wizard generates the following files. The following topics use the short name "MyProvider", but the exact file names depend on the choice you made when creating the provider.

File name	Description
MyProviderRS.cpp	Contains the command helper `Execute` method and the provider column map.
MyProviderDS.h	Implements the data source object. This header file contains the property map for data source properties.
MyProviderRS.h	Implements the command and rowset objects. This header file contains the property map for rowset and command properties.
MyProviderSess.h	Implements the session object. This header file contains the property map for session properties.
MyProvider.rgs	Contains the registered objects generated by the OLE DB Provider Wizard.

See Also

Reference

Creating an OLE DB Provider

CMyProviderSource (MyProviderDS.H)

The provider classes use multiple inheritance. The following code shows the inheritance chain for the data source object:

```
/////////////////////////////////////////////////////////////////////////////
// CMyProviderSource
class ATL_NO_VTABLE CMyProviderSource :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<CMyProviderSource, &CLSID_MyProvider>,
   public IDBCreateSessionImpl<CMyProviderSource, CMyProviderSession>,
   public IDBInitializeImpl<CMyProviderSource>,
   public IDBPropertiesImpl<CMyProviderSource>,
   public IPersistImpl<CMyProviderSource>,
   public IInternalConnectionImpl<CMyProviderSource>
```

All the COM components derive from **CComObjectRootEx** and **CComCoClass**. **CComObjectRootEx** provides all the implementation for the **IUnknown** interface. It can handle any threading model. **CComCoClass** handles any error support required. If you want to send richer error information to the client, you can use some of the error APIs in **CComCoClass**.

The data source object also inherits from several 'Impl' classes. Each class provides the implementation for an interface. The data source object implements the **IPersist**, **IDBProperties**, **IDBInitialize**, and **IDBCreateSession** interfaces. Each interface is required by OLE DB to implement the data source object. You can choose to support or not support particular functionality by inheriting or not inheriting from one of these 'Impl' classes. If you want to support the **IDBDataSourceAdmin** interface, you inherit from the **IDBDataSourceAdminImpl** class to get the functionality required.

COM Map

Whenever the client calls **QueryInterface** for an interface on the data source, it goes through the following COM map:

```
BEGIN_COM_MAP(CMyProviderSource)
   COM_INTERFACE_ENTRY(IDBCreateSession)
   COM_INTERFACE_ENTRY(IDBInitialize)
   COM_INTERFACE_ENTRY(IDBProperties)
   COM_INTERFACE_ENTRY(IPersist)
   COM_INTERFACE_ENTRY(IInternalConnection)
END_COM_MAP()
```

The COM_INTERFACE_ENTRY macros are from ATL and tell the implementation of **QueryInterface** in **CComObjectRootEx** to return the appropriate interfaces.

Property Map

The property map specifies all the properties designated by the provider:

```
BEGIN_PROPSET_MAP(CMyProviderSource)
   BEGIN_PROPERTY_SET(DBPROPSET_DATASOURCEINFO)
      PROPERTY_INFO_ENTRY(ACTIVESESSIONS)
      PROPERTY_INFO_ENTRY(ASYNCTXNABORT)
      PROPERTY_INFO_ENTRY(ASYNCTXNCOMMIT)
      PROPERTY_INFO_ENTRY(BYREFACCESSORS)
      PROPERTY_INFO_ENTRY_VALUE(CATALOGLOCATION, DBPROPVAL_CL_START)
      PROPERTY_INFO_ENTRY(CATALOGTERM)
      PROPERTY_INFO_ENTRY(CATALOGUSAGE)
      PROPERTY_INFO_ENTRY(COLUMNDEFINITION)
      PROPERTY_INFO_ENTRY(CONCATNULLBEHAVIOR)
      PROPERTY_INFO_ENTRY(DATASOURCENAME)
      PROPERTY_INFO_ENTRY(DATASOURCEREADONLY)
      PROPERTY_INFO_ENTRY(DBMSNAME)
      PROPERTY_INFO_ENTRY(DBMSVER)
      PROPERTY_INFO_ENTRY_VALUE(DSOTHREADMODEL, DBPROPVAL_RT_FREETHREAD)
      PROPERTY_INFO_ENTRY(GROUPBY)
      PROPERTY_INFO_ENTRY(HETEROGENEOUSTABLES)
      PROPERTY_INFO_ENTRY(IDENTIFIERCASE)
```

```
      PROPERTY_INFO_ENTRY(MAXINDEXSIZE)
      PROPERTY_INFO_ENTRY(MAXROWSIZE)
      PROPERTY_INFO_ENTRY(MAXROWSIZEINCLUDESBLOB)
      PROPERTY_INFO_ENTRY(MAXTABLESINSELECT)
      PROPERTY_INFO_ENTRY(MULTIPLEPARAMSETS)
      PROPERTY_INFO_ENTRY(MULTIPLERESULTS)
      PROPERTY_INFO_ENTRY(MULTIPLESTORAGEOBJECTS)
      PROPERTY_INFO_ENTRY(MULTITABLEUPDATE)
      PROPERTY_INFO_ENTRY(NULLCOLLATION)
      PROPERTY_INFO_ENTRY(OLEOBJECTS)
      PROPERTY_INFO_ENTRY(ORDERBYCOLUMNSINSELECT)
      PROPERTY_INFO_ENTRY(OUTPUTPARAMETERAVAILABILITY)
      PROPERTY_INFO_ENTRY(PERSISTENTIDTYPE)
      PROPERTY_INFO_ENTRY(PREPAREABORTBEHAVIOR)
      PROPERTY_INFO_ENTRY(PREPARECOMMITBEHAVIOR)
      PROPERTY_INFO_ENTRY(PROCEDURETERM)
      PROPERTY_INFO_ENTRY(PROVIDERNAME)
      PROPERTY_INFO_ENTRY(PROVIDEROLEDBVER)
      PROPERTY_INFO_ENTRY(PROVIDERVER)
      PROPERTY_INFO_ENTRY(QUOTEDIDENTIFIERCASE)
      PROPERTY_INFO_ENTRY(ROWSETCONVERSIONSONCOMMAND)
      PROPERTY_INFO_ENTRY(SCHEMATERM)
      PROPERTY_INFO_ENTRY(SCHEMAUSAGE)
      PROPERTY_INFO_ENTRY(STRUCTUREDSTORAGE)
      PROPERTY_INFO_ENTRY(SUBQUERIES)
      PROPERTY_INFO_ENTRY(TABLETERM)
      PROPERTY_INFO_ENTRY(USERNAME)
   END_PROPERTY_SET(DBPROPSET_DATASOURCEINFO)
   BEGIN_PROPERTY_SET(DBPROPSET_DBINIT)
      PROPERTY_INFO_ENTRY(AUTH_PASSWORD)
      PROPERTY_INFO_ENTRY(AUTH_PERSIST_SENSITIVE_AUTHINFO)
      PROPERTY_INFO_ENTRY(AUTH_USERID)
      PROPERTY_INFO_ENTRY(INIT_DATASOURCE)
      PROPERTY_INFO_ENTRY(INIT_HWND)
      PROPERTY_INFO_ENTRY(INIT_LCID)
      PROPERTY_INFO_ENTRY(INIT_LOCATION)
      PROPERTY_INFO_ENTRY(INIT_MODE)
      PROPERTY_INFO_ENTRY(INIT_PROMPT)
      PROPERTY_INFO_ENTRY(INIT_PROVIDERSTRING)
      PROPERTY_INFO_ENTRY(INIT_TIMEOUT)
   END_PROPERTY_SET(DBPROPSET_DBINIT)
   BEGIN_PROPERTY_SET(DBPROPSET_DATASOURCE)
      PROPERTY_INFO_ENTRY(CURRENTCATALOG)
   END_PROPERTY_SET(DBPROPSET_DATASOURCE)
   CHAIN_PROPERTY_SET(CMyProviderSession)
END_PROPSET_MAP()
```

Properties in OLE DB are grouped. The data source object has two groups of properties: one for the **DBPROPSET_DATASOURCEINFO** set and one for the **DBPROPSET_DBINIT** set. The **DBPROPSET_DATASOURCEINFO** set corresponds to properties about the provider and its data source. The **DBPROPSET_DBINIT** set corresponds to properties used at initialization. The OLE DB Provider Templates handle these sets with the PROPERTY_SET macros. The macros create a block that contains an array of properties. Whenever the client calls the **IDBProperties** interface, the provider uses the property map.

You do not need to implement every property in the specification. However, you must support the required properties; see the level 0 conformance specification for more information. If you do not want to support a property, you can remove it from the map. If you want to support a property, add it into the map by using a PROPERTY_INFO_ENTRY macro. The macro corresponds to the **UPROPINFO** structure as shown in the following code:

```
struct UPROPINFO
{
   DBPROPID    dwPropId;
   ULONG       ulIDS;
   VARTYPE     VarType;
   DBPROPFLAGS dwFlags;
   union
   {
```

```
        DWORD dwVal;
        LPOLESTR szVal;
    };
    DBPROPOPTIONS dwOption;
};
```

Each element in the structure represents information to handle the property. It contains a **DBPROPID** to determine the GUID and ID for the property. It also contains entries to determine the type and value of the property.

If you want to change the default value of a property (note that a consumer can change the value of a writable property at any time), you can use either the PROPERTY_INFO_ENTRY_VALUE or PROPERTY_INFO_ENTRY_EX macro. These macros allow you to specify a value for a corresponding property. To learn more about a property and its values, consult Properties (OLE DB). The PROPERTY_INFO_ENTRY_VALUE macro is a shorthand notation that allows you to change the value. The PROPERTY_INFO_ENTRY_VALUE macro calls the PROPERTY_INFO_ENTRY_EX macro. This macro allows you to add or change all of the attributes in the **UPROPINFO** structure.

If you want to define your own property set, you can add one by making an additional BEGIN_PROPSET_MAP/END_PROPSET_MAP combination. You need to define a GUID for the property set and then define your own properties. If you have provider-specific properties, add them to a new property set instead of using an existing one. This avoids problems in later versions of OLE DB.

User-Defined Property Sets

Visual C++ .NET supports user-defined property sets. You no longer have to override **GetProperties** or **GetPropertyInfo**. Instead, the templates detect any user-defined property set and add it to the appropriate object.

If you have a user-defined property set that needs to be available at initialization time (that is, before the consumer calls **IDBInitialize::Initialize**), you can specify this by using the **UPROPSET_USERINIT** flag in conjunction with the BEGIN_PROPERTY_SET_EX macro. The property set must be in the data source object for this to work (as the OLE DB specification requires). For example:

```
BEGIN_PROPERTY_SET_EX(DBPROPSET_MYPROPSET, UPROPSET_USERINIT)
   PROPERTY_INFO_ENTRY(DBPROP_MYPROP)
END_PROPERTY_SET_EX(DBPROPSET_MYPROPSET)
```

See Also

Concepts

Provider Wizard-Generated Files

CMyProviderSession (MyProviderSess.H)

MyProviderSess.H contains the declaration and implementation for the OLE DB session object. The data source object creates the session object and represents a conversation between a consumer and provider. Several simultaneous sessions can be open for one data source. The inheritance list for `CMyProviderSession` follows:

```
/////////////////////////////////////////////////////////////////////////////
// CMyProviderSession
class ATL_NO_VTABLE CMyProviderSession :
   public CComObjectRootEx<CComSingleThreadModel>,
   public IGetDataSourceImpl<CMyProviderSession>,
   public IOpenRowsetImpl<CMyProviderSession>,
   public ISessionPropertiesImpl<CMyProviderSession>,
   public IObjectWithSiteSessionImpl<CMyProviderSession>,
   public IDBSchemaRowsetImpl<CMyProviderSession>,
   public IDBCreateCommandImpl<CMyProviderSession, CMyProviderCommand>
```

The session object inherits from **IGetDataSource**, **IOpenRowset**, **ISessionProperties**, and **IDBCreateCommand**. The **IGetDataSource** interface allows a session to retrieve the data source that created it. This is useful if you need to get properties from the data source that you created or other information that the data source can provide. The **ISessionProperties** interface handles all the properties for the session. The **IOpenRowset** and **IDBCreateCommand** interfaces are used to do the database work. If the provider supports commands, it implements the **IDBCreateCommand** interface. It is used to create the command object that can execute commands. The provider always implements the **IOpenRowset** object. It is used to generate a simple rowset from a provider. It is a default rowset (for example, "`select * from mytable`") from a provider.

The wizard also generates three session classes: `CMyProviderSessionColSchema`, `CMyProviderSessionPTSchema`, and `CMyProviderSessionTRSchema`. These sessions are used for schema rowsets. Schema rowsets allow the provider to return metadata to the consumer without the consumer having to execute a query or fetch data. Fetching metadata can be far quicker than discovering a providers capabilities.

The OLE DB specification requires that providers implementing the **IDBSchemaRowset** interface support three schema rowset types: **DBSCHEMA_COLUMNS**, **DBSCHEMA_PROVIDER_TYPES**, and **DBSCHEMA_TABLES**. The wizard generates implementations for each schema rowset. Each class generated by the wizard contains an `Execute` method. In this `Execute` method, you can return data to the provider about which tables, columns, and data types you support. This data is usually known at compile time.

See Also

Concepts

Provider Wizard-Generated Files

CMyProviderCommand (MyProviderRS.H)

The `CMyProviderCommand` class is the implementation for the provider command object. It provides the implementation for the **IAccessor**, **ICommandText**, and **ICommandProperties** interfaces. The **IAccessor** interface is the same as the one in the rowset. The command object uses the accessor to specify bindings for parameters. The rowset object uses them to specify bindings for output columns. The **ICommandText** interface is a useful way to specify a command textually. This example uses the **ICommandText** interface later when it adds custom code; it also overrides the **ICommand::Execute** method. The **ICommandProperties** interface handles all of the properties for the command and rowset objects.

```
// CMyProviderCommand
class ATL_NO_VTABLE CMyProviderCommand :
class ATL_NO_VTABLE CMyProviderCommand :
   public CComObjectRootEx<CComSingleThreadModel>,
   public IAccessorImpl<CMyProviderCommand>,
   public ICommandTextImpl<CMyProviderCommand>,
   public ICommandPropertiesImpl<CMyProviderCommand>,
   public IObjectWithSiteImpl<CMyProviderCommand>,
   public IConvertTypeImpl<CMyProviderCommand>,
   public IColumnsInfoImpl<CMyProviderCommand>
```

The **IAccessor** interface manages all the bindings used in commands and rowsets. The consumer calls **IAccessor::CreateAccessor** with an array of **DBBINDING** structures. Each **DBBINDING** structure contains the information about how the column bindings should be handled (such as type and length). The provider receives the structures and then determines how the data should be transferred and whether any conversions are necessary. The **IAccessor** interface is used in the command object to handle any parameters in the command.

The command object also provides an implementation of **IColumnsInfo**. OLE DB requires the **IColumnsInfo** interface. The interface allows the consumer to retrieve information about parameters from the command. The rowset object uses the **IColumnsInfo** interface to return information about the output columns to the provider.

The provider also contains an interface called **IObjectWithSite**. The **IObjectWithSite** interface was implemented in ATL 2.0 and allows the implementer to pass information about itself to its child. The command object uses the **IObjectWithSite** information to tell any generated rowset objects about who created them.

See Also

Concepts

Provider Wizard-Generated Files

CMyProviderRowset (MyProviderRS.H)

The wizard generates an entry for the rowset object. In this case, it is called `CMyProviderRowset`. The `CMyProviderRowset` class inherits from an OLE DB provider class called **CRowsetImpl**, which implements all the necessary interfaces for the rowset object. The following code shows the inheritance chain for **CRowsetImpl**:

```
template <class T, class Storage, class CreatorClass,
   class ArrayType = CAtlArray<Storage> >
class CMyRowsetImpl:
   public CRowsetImpl<T, Storage, CreatorClass, ArrayType,
      CSimpleRow, IRowsetLocateImpl< T > >
```

CRowsetImpl also uses the **IAccessor** and **IColumnsInfo** interfaces. It uses these interfaces for output fields in tables. The class also provides an implementation for **IRowsetIdentity**, which allows the consumer to determine if two rows are identical. The **IRowsetInfo** interface implements properties for the rowset object. The **IConvertType** interface allows the provider to resolve differences between data types requested by the consumer and those used by the provider.

The **IRowset** interface actually handles data retrieval. The consumer first calls a method called **GetNextRows** to return a handle to a row, known as an **HROW**. The consumer then calls **IRowset::GetData** with that **HROW** to retrieve the requested data.

CRowsetImpl also takes several template parameters. These parameters allow you to determine how the **CRowsetImpl** class handles data. The *ArrayType* argument allows you to determine what storage mechanism is used to store the row data. The **RowClass** parameter specifies what class contains an **HROW**.

The **RowsetInterface** parameter allows you to also use the **IRowsetLocate** or **IRowsetScroll** interface. The **IRowsetLocate** and **IRowsetScroll** interfaces both inherit from **IRowset**. Therefore, the OLE DB provider templates must provide special handling for these interfaces. If you want to use either of these interfaces, you need to use this parameter.

See Also

Concepts

Provider Wizard-Generated Files

CMyProviderWindowsFile

The wizard creates a class to contain one row of data; in this case, it is called `CMyProviderWindowsFile`. The following code for `CMyProviderWindowsFile` is wizard generated and lists all the files in a directory by using the **WIN32_FIND_DATA** structure. `CMyProviderWindowsFile` inherits from the **WIN32_FIND_DATA** structure:

```
/////////////////////////////////////////////////////////////////////
// MyProviderRS.H

class CMyProviderWindowsFile:
   public WIN32_FIND_DATA
{
public:
BEGIN_PROVIDER_COLUMN_MAP(CMyProviderWindowsFile)
   PROVIDER_COLUMN_ENTRY("FileAttributes", 1, dwFileAttributes)
   PROVIDER_COLUMN_ENTRY("FileSizeHigh", 2, nFileSizeHigh)
   PROVIDER_COLUMN_ENTRY("FileSizeLow", 3, nFileSizeLow)
   PROVIDER_COLUMN_ENTRY_STR("FileName", 4, cFileName)
   PROVIDER_COLUMN_ENTRY_STR("AltFileName", 5, cAlternateFileName)
END_PROVIDER_COLUMN_MAP()
};
```

`CMyProviderWindowsFile` is called the user record class because it also contains a map describing the columns in the provider's rowset. The provider column map contains one entry for each field in the rowset using the PROVIDER_COLUMN_ENTRY macros. The macros specify column name, ordinal, and offset to a structure entry. The provider column entries in the above code contain offsets into the **WIN32_FIND_DATA** structure. When the consumer calls **IRowset::GetData**, data is transferred in one contiguous buffer. Rather than making you do pointer arithmetic, the map allows you to specify a data member.

The `CMyProviderRowset` class also contains the `Execute` method. `Execute` is what actually reads the data in from the native source. The following code shows the wizard-generated `Execute` method. The function uses the Win32 **FindFirstFile** and **FindNextFile** APIs to retrieve information about the files in the directory and place them in instances of the `CMyProviderWindowsFile` class.

```
/////////////////////////////////////////////////////////////////////
// MyProviderRS.H

HRESULT Execute(DBPARAMS * pParams, LONG* pcRowsAffected)
{
   USES_CONVERSION;
   BOOL bFound = FALSE;
   HANDLE hFile;
   LPTSTR  szDir = (m_strCommandText == _T("")) ? _T("*.*") :
       OLE2T(m_strCommandText);
   CMyProviderWindowsFile wf;
   hFile = FindFirstFile(szDir, &wf);
   if (hFile == INVALID_HANDLE_VALUE)
      return DB_E_ERRORSINCOMMAND;
   LONG cFiles = 1;
   BOOL bMoreFiles = TRUE;
   while (bMoreFiles)
   {
      if (!m_rgRowData.Add(wf))
         return E_OUTOFMEMORY;
      bMoreFiles = FindNextFile(hFile, &wf);
      cFiles++;
   }
   FindClose(hFile);
   if (pcRowsAffected != NULL)
      *pcRowsAffected = cFiles;
   return S_OK;
}
```

The directory to search is represented by **m_strCommandText**; this contains the text represented by the **ICommandText** interface in the command object. If no directory is specified, it uses the current directory.

The method creates one entry for each file (corresponding to a row) and places it in the **m_rgRowData** data member. The **CRowsetImpl** class defines the **m_rgRowData** data member. The data in this array represents the entire table and is used throughout the templates.

See Also

Concepts

Provider Wizard-Generated Files

Creating a Simple Read-Only Provider

When you have created an OLE DB provider using the ATL Project Wizard and ATL OLE DB Provider Wizard, you can add other functionality that you want to support. Start designing your provider by examining what kind of data you will be sending to the consumer and under what conditions. It is especially important to determine whether you need to support commands, transactions, and other optional objects. A good design up front will speed implementation and testing.

The example is presented in two parts:

- The first part shows how to create a simple read-only provider that reads a pair of strings.
- The second part shows how to enhance the simple read-only provider by adding the **IRowsetLocate** interface.

See Also

Reference

Creating an OLE DB Provider

Implementing the Simple Read-Only Provider

The extended example in this topic shows how to edit the wizard-created files to create a simple read-only provider that reads a set of two strings from a text file. To create this provider from the wizard files, add code to complete the following tasks:

- Read the strings into the provider
- Store the strings in the provider

See Also

Reference

Creating a Simple Read-Only Provider

Reading Strings into the OLE DB Provider

The `RMyProviderRowset::Execute` function opens a file and reads strings. The consumer passes the file name to the provider by calling ICommandText::SetCommandText. The provider receives the file name and stores it in the member variable `m_szCommandText`. `Execute` reads the file name from `m_szCommandText`. If the file name is invalid or the file is unavailable, `Execute` returns an error. Otherwise, it opens the file and calls **fgets** to retrieve the strings. For each set of strings it reads, `Execute` creates an instance of the user record (`CAgentMan`) and places it into an array.

If the file cannot be opened, `Execute` must return **DB_E_NOTABLE**. If it returns **E_FAIL** instead, the provider will not work with many consumers and will not pass the OLE DB conformance tests.

Example

The edited `Execute` function looks like this:

```
////////////////////////////////////////////////////////////////////////
// MyProviderRS.h
class RMyProviderRowset : public CRowsetImpl< RMyProviderRowset, CAgentMan, CRMyProviderCom
mand>
{
public:
    HRESULT Execute(DBPARAMS * pParams, LONG* pcRowsAffected)
    {
        enum {
            sizeOfBuffer = 256,
            sizeOfFile = MAX_PATH
        };
        USES_CONVERSION;
        FILE* pFile = NULL;
        TCHAR szString[sizeOfBuffer];
        TCHAR szFile[sizeOfFile];
        size_t nLength;        errcodeerr;

        ObjectLock lock(this);

        // From a filename, passed in as a command text, scan the file
        // placing data in the data array.
        if (!m_szCommandText)
        {
            ATLTRACE("No filename specified");
            return E_FAIL;
        }

        // Open the file
        _tcscpy_s(szFile, sizeOfFile, m_szCommandText);
        if (szFile[0] == _T('\0') ||
            ((err = fopen_s(&pFile, &szFile[0], "r")) == 0))
        {
            ATLTRACE("Could not open file");
            return DB_E_NOTABLE;
        }

        // Scan and parse the file.
        // The file should contain two strings per record
        LONG cFiles = 0;
        while (fgets(szString, sizeOfBuffer, pFile) != NULL)
        {
            nLength = strlen(szString);
            szString[nLength-1] = '\0';   // Strip off trailing CR/LF
            CAgentMan am;
            _tcscpy_s(am.szCommand, am.sizeOfCommand, szString);
            _tcscpy_s(am.szCommand2, am.sizeOfCommand2, szString);

            if (fgets(szString, sizeOfBuffer, pFile) != NULL)
            {
```

```
                nLength = strlen(szString);
                szString[nLength-1] = '\0'; // Strip off trailing CR/LF
                _tcscpy_s(am.szText, am.sizeOfText, szString);
                _tcscpy_s(am.szText2, am.sizeOfText2, szString);
            }

            am.dwBookmark = ++cFiles;
            if (!m_rgRowData.Add(am))
            {
                ATLTRACE("Couldn't add data to array");
                fclose(pFile);
                return E_FAIL;
            }
        }

        if (pcRowsAffected != NULL)
            *pcRowsAffected = cFiles;
        return S_OK;
    }
}
```

See Also

Reference

Implementing the Simple Read-Only Provider

Storing Strings in the OLE DB Provider

In MyProviderRS.h, the ATL OLE DB Provider Wizard creates a default user record called `CWindowsFile`. To handle the two strings, either modify `CWindowsFile` or add your own user record as shown in the following code:

```
////////////////////////////////////////////////////////////////////////
class CAgentMan:
   public WIN32_FIND_DATA
   DWORD dwBookmark;              // Add this
   TCHAR szCommand[256];          // Add this
   TCHAR szText[256];             // Add this
   TCHAR szCommand2[256];         // Add this
   TCHAR szText2[256];            // Add this

{
public:
BEGIN_PROVIDER_COLUMN_MAP()
   PROVIDER_COLUMN_ENTRY_STR(OLESTR("Command"), 1, 256, GUID_NULL, CAgentMan, szCommand)
   PROVIDER_COLUMN_ENTRY_STR(OLESTR("Text"), 2, 256, GUID_NULL, CAgentMan, szText)
   PROVIDER_COLUMN_ENTRY_STR(OLESTR("Command2"), 3, 256, GUID_NULL, CAgentMan, szCommand2)
   PROVIDER_COLUMN_ENTRY_STR(OLESTR("Text2"),4, 256, GUID_NULL, CAgentMan, szText2)
END_PROVIDER_COLUMN_MAP()
   bool operator==(const CAgentMan& am) // This is optional
   {
      return (lstrcmpi(cFileName, wf.cFileName) == 0);
   }
};
```

The data members `szCommand` and `szText` represent the two strings, with `szCommand2` and `szText2` providing additional columns if needed. The data member `dwBookmark` is not needed for this simple read-only provider but is used later to add an **IRowsetLocate** interface; see Enhancing the Simple Read Only Provider. The **==** operator compares instances (implementing this operator is optional).

When this is done, your provider should be ready to compile and run. To test the provider, you need a consumer with matching functionality. Implementing a Simple Consumer shows how to create such a test consumer. Run the test consumer with the provider. Verify that the test consumer retrieves the proper strings from the provider when you click the **Run** button in the **Test Consumer** dialog box.

When you have successfully tested your provider, you might want to enhance its functionality by implementing additional interfaces. An example is shown in Enhancing the Simple Read-Only Provider.

See Also

Reference

Implementing the Simple Read-Only Provider

Enhancing the Simple Read-Only Provider

This section shows how to enhance the simple read-only provider created in the previous section. **IRowsetLocateImpl** creates an implementation for the **IRowsetLocate** interface and adds bookmark support for you.

When you have a working provider, you might want to enhance it to make the provider update, handle transactions, or enhance the performance of the row-fetching algorithm. Most provider enhancements involve adding an interface to an existing COM object.

The example in the following topics enhances the row-fetching mechanism by adding the **IRowsetLocate** interface to `CAgentRowset`. The topics show you how to:

- Make RMyProviderRowset inherit from IRowsetLocate.
- Dynamically determine the columns returned to the consumer.

See Also

Reference

Creating a Simple Read-Only Provider

Modifying the Inheritance of RMyProviderRowset

To add the **IRowsetLocate** interface to the simple read-only provider example, modify the inheritance of **RMyProviderRowset**. Initially, **RMyProviderRowset** inherits from **CRowsetImpl**. You need to modify it to inherit from **CRowsetBaseImpl**.

To do this, create a new class, `CMyRowsetImpl`, in MyProviderRS.h:

```
////////////////////////////////////////////////////////////////////
// MyProviderRS.h

template <class T, class Storage, class CreatorClass, class ArrayType = CAtlArray<Storage>
>
class CMyRowsetImpl:
   public CRowsetImpl<T, Storage, CreatorClass, ArrayType, CSimpleRow, IRowsetLocateImpl< T
, IRowsetLocate > >
{
...
};
```

Now, edit the COM interface map in MyProviderRS.h to be as follows:

```
BEGIN_COM_MAP(CMyRowsetImpl)
   COM_INTERFACE_ENTRY(IRowsetLocate)
   COM_INTERFACE_ENTRY_CHAIN(_RowsetBaseClass)
END_COM_MAP()
```

This creates a COM interface map that tells `CMyRowsetImpl` to call **QueryInterface** for both the **IRowset** and **IRowsetLocate** interfaces. To get all of the implementation for the other rowset classes, the map links the `CMyRowsetImpl` class back to the **CRowsetBaseImpl** class defined by the OLE DB Templates; the map uses the COM_INTERFACE_ENTRY_CHAIN macro, which tells OLE DB templates to scan the COM map in **CRowsetBaseImpl** in response to a **QueryInterface** call.

Finally, link `RAgentRowset` to `CMyRowsetBaseImpl` by modifying `RAgentRowset` to inherit from `CMyRowsetImpl`, as follows:

```
class RAgentRowset : public CMyRowsetImpl<RAgentRowset, CAgentMan, CMyProviderCommand>
```

`RAgentRowset` can now use the **IRowsetLocate** interface while taking advantage of the rest of the implementation for the rowset class.

When this is done, you can dynamically determine columns returned to the consumer.

See Also

Reference

Enhancing the Simple Read-Only Provider

Dynamically Determining Columns Returned to the Consumer

The PROVIDER_COLUMN_ENTRY macros normally handle the **IColumnsInfo::GetColumnsInfo** call. However, because a consumer might choose to use bookmarks, the provider must be able to change the columns returned depending on whether the consumer asks for a bookmark.

To handle the **IColumnsInfo::GetColumnsInfo** call, delete the PROVIDER_COLUMN_MAP, which defines a function `GetColumnInfo`, from the `CAgentMan` user record in MyProviderRS.h and replace it with the definition for your own `GetColumnInfo` function:

```
////////////////////////////////////////////////////////////////////////
// MyProviderRS.H
class CAgentMan
{
public:
   DWORD dwBookmark;
   TCHAR szCommand[256];
   TCHAR szText[256];
   TCHAR szCommand2[256];
   TCHAR szText2[256];

   static ATLCOLUMNINFO* GetColumnInfo(void* pThis, ULONG* pcCols);
   bool operator==(const CAgentMan& am)
   {
      return (lstrcmpi(szCommand, am.szCommand) == 0);
   }

};
```

Next, implement the `GetColumnInfo` function in MyProviderRS.cpp, as shown in the following code.

`GetColumnInfo` checks first to see if the OLE DB property **DBPROP_BOOKMARKS** is set. To get the property, `GetColumnInfo` uses a pointer (`pRowset`) to the rowset object. The `pThis` pointer represents the class that created the rowset, which is the class where the property map is stored. `GetColumnInfo` typecasts the `pThis` pointer to an `RMyProviderRowset` pointer.

To check for the **DBPROP_BOOKMARKS** property, `GetColumnInfo` uses the **IRowsetInfo** interface, which you can obtain by calling **QueryInterface** on the `pRowset` interface. As an alternative, you can use an ATL CComQIPtr method instead.

```
////////////////////////////////////////////////////////////////////
// MyProviderRS.cpp
ATLCOLUMNINFO* CAgentMan::GetColumnInfo(void* pThis, ULONG* pcCols)
{
   static ATLCOLUMNINFO _rgColumns[5];
   ULONG ulCols = 0;

   // Check the property flag for bookmarks; if it is set, set the zero
   // ordinal entry in the column map with the bookmark information.
   CAgentRowset* pRowset = (CAgentRowset*) pThis;
   CComQIPtr<IRowsetInfo, &IID_IRowsetInfo> spRowsetProps = pRowset;

   CDBPropIDSet set(DBPROPSET_ROWSET);
   set.AddPropertyID(DBPROP_BOOKMARKS);
   DBPROPSET* pPropSet = NULL;
   ULONG ulPropSet = 0;
   HRESULT hr;

   if (spRowsetProps)
      hr = spRowsetProps->GetProperties(1, &set, &ulPropSet, &pPropSet);

   if (pPropSet)
   {
      CComVariant var = pPropSet->rgProperties[0].vValue;
      CoTaskMemFree(pPropSet->rgProperties);
      CoTaskMemFree(pPropSet);
```

```
      if (SUCCEEDED(hr) && (var.boolVal == VARIANT_TRUE))
      {
         ADD_COLUMN_ENTRY_EX(ulCols, OLESTR("Bookmark"), 0, sizeof(DWORD),
         DBTYPE_BYTES, 0, 0, GUID_NULL, CAgentMan, dwBookmark,
         DBCOLUMNFLAGS_ISBOOKMARK)
         ulCols++;
      }
   }

   // Next, set the other columns up.
   ADD_COLUMN_ENTRY(ulCols, OLESTR("Command"), 1, 256, DBTYPE_STR, 0xFF, 0xFF,
      GUID_NULL, CAgentMan, szCommand)
   ulCols++;
   ADD_COLUMN_ENTRY(ulCols, OLESTR("Text"), 2, 256, DBTYPE_STR, 0xFF, 0xFF,
      GUID_NULL, CAgentMan, szText)
   ulCols++;

   ADD_COLUMN_ENTRY(ulCols, OLESTR("Command2"), 3, 256, DBTYPE_STR, 0xFF, 0xFF,
      GUID_NULL, CAgentMan, szCommand2)
   ulCols++;
   ADD_COLUMN_ENTRY(ulCols, OLESTR("Text2"), 4, 256, DBTYPE_STR, 0xFF, 0xFF,
      GUID_NULL, CAgentMan, szText2)
   ulCols++;

   if (pcCols != NULL)
      *pcCols = ulCols;

   return _rgColumns;
}
```

This example uses a static array to contain the column information. If the consumer does not want the bookmark column, one entry in the array is unused. To handle the information, you create two array macros: ADD_COLUMN_ENTRY and ADD_COLUMN_ENTRY_EX. ADD_COLUMN_ENTRY_EX takes an extra parameter, `flags`, that is needed if you designate a bookmark column.

```
//////////////////////////////////////////////////////////////////////
// MyProviderRS.h

#define ADD_COLUMN_ENTRY(ulCols, name, ordinal, colSize, type, precision,
scale, guid, dataClass, member) \
   _rgColumns[ulCols].pwszName = (LPOLESTR)name; \
   _rgColumns[ulCols].pTypeInfo = (ITypeInfo*)NULL; \
   _rgColumns[ulCols].iOrdinal = (ULONG)ordinal; \
   _rgColumns[ulCols].dwFlags = 0; \
   _rgColumns[ulCols].ulColumnSize = (ULONG)colSize; \
   _rgColumns[ulCols].wType = (DBTYPE)type; \
   _rgColumns[ulCols].bPrecision = (BYTE)precision; \
   _rgColumns[ulCols].bScale = (BYTE)scale; \
   _rgColumns[ulCols].cbOffset = offsetof(dataClass, member);

#define ADD_COLUMN_ENTRY_EX(ulCols, name, ordinal, colSize, type,
precision, scale, guid, dataClass, member, flags) \
   _rgColumns[ulCols].pwszName = (LPOLESTR)name; \
   _rgColumns[ulCols].pTypeInfo = (ITypeInfo*)NULL; \
   _rgColumns[ulCols].iOrdinal = (ULONG)ordinal; \
   _rgColumns[ulCols].dwFlags = flags; \
   _rgColumns[ulCols].ulColumnSize = (ULONG)colSize; \
   _rgColumns[ulCols].wType = (DBTYPE)type; \
   _rgColumns[ulCols].bPrecision = (BYTE)precision; \
   _rgColumns[ulCols].bScale = (BYTE)scale; \
   _rgColumns[ulCols].cbOffset = offsetof(dataClass, member); \
   memset(&(_rgColumns[ulCols].columnid), 0, sizeof(DBID)); \
   _rgColumns[ulCols].columnid.uName.pwszName = (LPOLESTR)name;
```

In the `GetColumnInfo` function, the bookmark macro is used like this:

```
ADD_COLUMN_ENTRY_EX(ulCols, OLESTR("Bookmark"), 0, sizeof(DWORD),
   DBTYPE_BYTES, 0, 0, GUID_NULL, CAgentMan, dwBookmark,
   DBCOLUMNFLAGS_ISBOOKMARK)
```

You can now compile and run the enhanced provider. To test the provider, modify the test consumer as described in Implementing a Simple Consumer. Run the test consumer with the provider. Verify that the test consumer retrieves the proper strings from the provider when you click the **Run** button in the **Test Consumer** dialog box.

See Also

Reference

Enhancing the Simple Read-Only Provider

Testing the Read-Only Provider

To test a provider, you need a consumer. It helps if the consumer can match up with the provider. The OLE DB consumer templates are a thin wrapper around OLE DB and match with the provider COM objects. Because the source is shipped with the consumer templates, it is easy to debug a provider with them. The consumer templates are also a very small and fast way to develop consumer applications.

The example in this topic creates a default MFC Application Wizard application for a test consumer. The test application is a simple dialog with OLE DB consumer template code added.

To create the test application

1. On the **File** menu, click **New**, and then click **Project**.
2. In the Project Types pane, select the **Visual C++ Projects** folder. In the Templates pane, select **MFC Application**.
3. For the project name, enter **TestProv**, and then click **OK**.

   The MFC Application Wizard appears.
4. On the **Application Type** page, select **Dialog based**.
5. On the **Advanced Features** page, select **Automation**, and then click **Finish**.

Note
The application does not require Automation support if you add **CoInitialize** in **CTestProvApp::InitInstance**.

You can view and edit the TestProv dialog box (IDD_TESTPROV_DIALOG) by selecting it in Resource View. Place two list boxes, one for each string in the rowset, in the dialog box. Turn off the sort property for both list boxes by pressing ALT+Enter when a list box is selected, clicking the **Styles** tab, and clearing the **Sort** check box. Also, place a **Run** button on the dialog box to fetch the file. The finished TestProv dialog box should have two list boxes labeled "String 1" and "String 2", respectively; it also has **OK**, **Cancel**, and **Run** buttons.

Open the header file for the dialog class (in this case TestProvDlg.h). Add the following code to the header file (outside of any class declarations):

```
/////////////////////////////////////////////////////////////////////////////
// TestProvDlg.h

class CProvider
{
// Attributes
public:
   char   szField1[16];
   char   szField2[16];

   // Binding Maps
BEGIN_COLUMN_MAP(CProvider)
   COLUMN_ENTRY(1, szField1)
   COLUMN_ENTRY(2, szField2)
END_COLUMN_MAP()
};
```

The code represents a user record that defines what columns will be in the rowset. When the client calls **IAccessor::CreateAccessor**, it uses these entries to specify which columns to bind. The OLE DB consumer templates also allow you to bind columns dynamically. The COLUMN_ENTRY macros are the client-side version of the PROVIDER_COLUMN_ENTRY macros. The two COLUMN_ENTRY macros specify the ordinal, type, length, and data member for the two strings.

Add a handler function for the **Run** button by pressing CTRL and double-clicking the **Run** button. Place the following code in the function:

```
/////////////////////////////////////////////////////////////////////////////
```

```
// TestProvDlg.cpp

void CtestProvDlg::OnRun()
{
   CCommand<CAccessor<CProvider> > table;
   CDataSource source;
   CSession   session;

   if (source.Open("MyProvider.MyProvider.1", NULL) != S_OK)
      return;

   if (session.Open(source) != S_OK)
      return;

   if (table.Open(session, _T("c:\\samples\\myprov\\myData.txt")) != S_OK)
      return;

   while (table.MoveNext() == S_OK)
   {
      m_ctlString1.AddString(table.szField1);
      m_ctlString2.AddString(table.szField2);
   }
}
```

The **CCommand**, **CDataSource**, and **CSession** classes all belong to the OLE DB consumer templates. Each class mimics a COM object in the provider. The **CCommand** object takes the `CProvider` class, declared in the header file, as a template parameter. The `CProvider` parameter represents bindings that you use to access the data from the provider. Here is the `Open` code for the data source, session, and command:

```
   if (source.Open("MyProvider.MyProvider.1", NULL) != S_OK)
      return;

   if (session.Open(source) != S_OK)
      return;

   if (table.Open(session, _T("c:\\samples\\myprov\\myData.txt")) != S_OK)
      return;
```

The lines to open each of the classes create each COM object in the provider. To locate the provider, use the ProgID of the provider. You can get the ProgID from the system registry or by looking in the MyProvider.rgs file (open the provider's directory and search for the ProgID key).

The MyData.txt file is included with the MyProv sample. To create a file of your own, use an editor and type an even number of strings, pressing ENTER between each string. Change the path name if you move the file.

Pass in the string "c:\\samples\\myprov\\MyData.txt" in the `table.Open` line. If you step into the `Open` call, you see that this string is passed to the `SetCommandText` method in the provider. Note that the `ICommandText::Execute` method used that string.

To fetch the data, call **MoveNext** on the table. **MoveNext** calls the **IRowset::GetNextRows**, **GetRowCount**, and **GetData** functions. When there are no more rows (that is, the current position in the rowset is greater than **GetRowCount**), the loop terminates:

```
   while (table.MoveNext() == S_OK)
   {
      m_ctlString1.AddString(table.szField1);
      m_ctlString2.AddString(table.szField2);
   }
```

Note that when there are no more rows, providers return **DB_S_ENDOFROWSET**. The **DB_S_ENDOFROWSET** value is not an error. You should always check against **S_OK** to cancel a data fetch loop and not use the SUCCEEDED macro.

You should now be able to build and test the program.

See Also

Reference

Creating an Updatable Provider

Visual C++ 6.0 supported only read-only providers. Visual C++ .NET supports updatable providers or providers that can update (write to) the data store. This topic discusses how to create updatable providers using OLE DB templates.

This topic assumes that you are starting with a workable provider. There are two steps to creating an updatable provider. You must first decide how the provider will make changes to the data store; specifically, whether changes are to be done immediately or deferred until an update command is issued. The section "Making Providers Updatable" describes the changes and settings you need to do in the provider code.

Next, you must make sure your provider contains all the functionality to support anything the consumer might request of it. If the consumer wants to update the data store, the provider has to contain code that persists data to the data store. For example, you might use the C Run-Time Library or MFC to perform such operations on your data source. The section "Writing to the Data Source" describes how to write to the data source, deal with **NULL** and default values, and set column flags.

Note
UpdatePV is an example of an updatable provider. UpdatePV is the same as MyProv but with updatable support.

Making Providers Updatable

The key to making a provider updatable is understanding what operations you want your provider to perform on the data store and how you want the provider to carry out those operations. Specifically, the major issue is whether updates to the data store are to be done immediately or deferred (batched) until an update command is issued.

You must first decide whether to inherit from **IRowsetChangeImpl** or **IRowsetUpdateImpl** in your rowset class. Depending on which of these you choose to implement, the functionality of three methods will be affected: **SetData**, **InsertRows**, and **DeleteRows**.

- If you inherit from IRowsetChangeImpl, calling these three methods immediately changes the data store.
- If you inherit from IRowsetUpdateImpl, the methods defer changes to the data store until you call **Update**, **GetOriginalData**, or **Undo**. If the update involves several changes, they are performed in batch mode (note that batching changes can add considerable memory overhead).

Note that **IRowsetUpdateImpl** derives from **IRowsetChangeImpl**. Thus, **IRowsetUpdateImpl** gives you change capability plus batch capability.

To support updatability in your provider

1. In your rowset class, inherit from **IRowsetChangeImpl** or **IRowsetUpdateImpl**. These classes provide appropriate interfaces for changing the data store:

   Adding IRowsetChange

   Add **IRowsetChangeImpl** to your inheritance chain using this form:

   ```
   IRowsetChangeImpl< rowset-name, storage-name >
   ```

   Also add `COM_INTERFACE_ENTRY(IRowsetChange)` to the **BEGIN_COM_MAP** section in your rowset class.

   Adding IRowsetUpdate

   Add **IRowsetUpdate** to your inheritance chain using this form:

   ```
   IRowsetUpdateImpl< rowset-name, storage>
   ```

Note

You should remove the **IRowsetChangeImpl** line from your inheritance chain. This one exception to the directive previously mentioned must include the code for **IRowsetChangeImpl**.

2. Add the following to your COM map (**BEGIN_COM_MAP ... END_COM_MAP**):

If you implement	Add to COM map
IRowsetChangeImpl	`COM_INTERFACE_ENTRY(IRowsetChange)`
IRowsetUpdateImpl	`COM_INTERFACE_ENTRY(IRowsetChange)COM_INTERFACE_ENTRY(IRowsetUpdate)`

3. In your command, add the following to your property set map (**BEGIN_PROPSET_MAP ... END_PROPSET_MAP**):

If you implement	Add to property set map
IRowsetChangeImpl	`PROPERTY_INFO_ENTRY_VALUE(IRowsetChange, VARIANT_FALSE)`
IRowsetUpdateImpl	`PROPERTY_INFO_ENTRY_VALUE(IRowsetChange, VARIANT_FALSE)PROPERTY_INFO_ENTRY_VALUE(IRowsetUpdate, VARIANT_FALSE)`

4. In your property set map, you should also include all of the following settings as they appear below:

```
PROPERTY_INFO_ENTRY_VALUE(UPDATABILITY, DBPROPVAL_UP_CHANGE |
  DBPROPVAL_UP_INSERT | DBPROPVAL_UP_DELETE)
PROPERTY_INFO_ENTRY_VALUE(CHANGEINSERTEDROWS, VARIANT_TRUE)
PROPERTY_INFO_ENTRY_VALUE(IMMOBILEROWS, VARIANT_TRUE)

PROPERTY_INFO_ENTRY_EX(OWNINSERT, VT_BOOL, DBPROPFLAGS_ROWSET |
  DBPROPFLAGS_READ, VARIANT_TRUE, 0)
PROPERTY_INFO_ENTRY_EX(OWNUPDATEDELETE, VT_BOOL, DBPROPFLAGS_ROWSET |
  DBPROPFLAGS_READ, VARIANT_TRUE, 0)
PROPERTY_INFO_ENTRY_EX(OTHERINSERT, VT_BOOL, DBPROPFLAGS_ROWSET |
  DBPROPFLAGS_READ, VARIANT_TRUE, 0)
PROPERTY_INFO_ENTRY_EX(OTHERUPDATEDELETE, VT_BOOL, DBPROPFLAGS_ROWSET |
  DBPROPFLAGS_READ, VARIANT_TRUE, 0)
PROPERTY_INFO_ENTRY_EX(REMOVEDELETED, VT_BOOL, DBPROPFLAGS_ROWSET |
  DBPROPFLAGS_READ, VARIANT_FALSE, 0)
```

You can find the values used in these macro calls by looking in Atldb.h for the property IDs and values (if Atldb.h differs from the online documentation, Atldb.h supersedes the documentation).

Note
Many of the **VARIANT_FALSE** and **VARIANT_TRUE** settings are required by the OLE DB templates; the OLE DB specification says they can be read/write, but the OLE DB templates can only support one value.

If you implement IRowsetChangeImpl

If you implement **IRowsetChangeImpl**, you must set the following properties on your provider. These properties are primarily used to request interfaces through **ICommandProperties::SetProperties**.

- DBPROP_IRowsetChange: Setting this automatically sets **DBPROP_IRowsetChange**.
- DBPROP_UPDATABILITY: A bitmask specifying the supported methods on **IRowsetChange**: **SetData**, **DeleteRows**, or **InsertRow**.
- DBPROP_CHANGEINSERTEDROWS: Consumer can call **IRowsetChange::DeleteRows** or **SetData** for newly inserted rows.

- DBPROP_IMMOBILEROWS: Rowset will not reorder inserted or updated rows.

If you implement IRowsetUpdateImpl

If you implement **IRowsetUpdateImpl**, you must set the following properties on your provider, in addition to setting all the properties for **IRowsetChangeImpl** previously listed:

- DBPROP_IRowsetUpdate.
- DBPROP_OWNINSERT: Must be READ_ONLY AND VARIANT_TRUE.
- DBPROP_OWNUPDATEDELETE: Must be READ_ONLY AND VARIANT_TRUE.
- DBPROP_OTHERINSERT: Must be READ_ONLY AND VARIANT_TRUE.
- DBPROP_OTHERUPDATEDELETE: Must be READ_ONLY AND VARIANT_TRUE.
- DBPROP_REMOVEDELETED: Must be READ_ONLY AND VARIANT_TRUE.
- DBPROP_MAXPENDINGROWS.

Note
If you support notifications, you might also have some other properties as well; see the section on **IRowsetNotifyCP** for this list.

For example of how the properties are set, see the property set map in **CUpdateCommand** (in Rowset.h) in UpdatePV.

Writing to the Data Source

To read from the data source, call the **Execute** function. To write to the data source, call the **FlushData** function. (In a general sense, flush means to save modifications you make to a table or index to disk.)

```
FlushData(HROW, HACCESSOR);
```

The row handle (*HROW*) and accessor handle (*HACCESSOR*) arguments allow you to specify the region to write. Typically, you write a single data field at a time.

The **FlushData** method writes data in the format in which it was originally stored. If you do not override this function, your provider will function correctly but changes will not be flushed to the data store.

When to Flush

The provider templates call **FlushData** whenever data needs to be written to the data store; this usually (but not always) occurs as a result of calls to the following functions:

- **IRowsetChange::DeleteRows**
- **IRowsetChange::SetData**
- **IRowsetChange::InsertRows** (if there is new data to insert in the row)
- **IRowsetUpdate::Update**

How It Works

The consumer makes a call that requires a flush (such as **Update**) and this call is passed to the provider, which always does the following:

- Calls **SetDBStatus** whenever you have a status value bound (see *OLE DB Programmers Reference*, Chapter 6, *Data Parts: Status*).
- Checks column flags.
- Calls **IsUpdateAllowed**.

These three steps help provide security. Then the provider calls **FlushData**.

How to Implement FlushData

To implement **FlushData**, you need to take into account several issues:

- Making sure that the data store can handle changes.
- Handling **NULL** values.
- Handling default values.

To implement your own **FlushData** method, you need to:

- Go to your rowset class.
- In the rowset class put the declaration of:

```
    HRESULT FlushData(HROW, HACCESSOR)
    {
        // Insert your implementation here and return an HRESULT.
    }
```

- Provide an implementation of **FlushData**.

A good implementation of **FlushData** stores only the rows and columns that are actually updated. You can use the *HROW* and *HACCESSOR* parameters to determine the current row and column being stored for optimization.

Typically, the biggest challenge is working with your own native data store. If possible, try to:

- Keep the method of writing to your data store as simple as possible.
- Handle **NULL** values (optional but advised).
- Handle default values (optional but advised).

The best thing to do is to have actual specified values in your data store for **NULL** and default values. It is best if you can extrapolate this data. If not, you are advised not to allow **NULL** and default values.

The following example shows how **FlushData** is implemented in the `RUpdateRowset` class in the UpdatePV sample (see Rowset.h in the sample code):

```
/////////////////////////////////////////////////////////////////////////////
// class RUpdateRowset (in rowset.h)
...
HRESULT FlushData(HROW, HACCESSOR)
{
    ATLTRACE2(atlTraceDBProvider, 0, "RUpdateRowset::FlushData\n");

    USES_CONVERSION;
    enum {
        sizeOfString = 256,
        sizeOfFileName = MAX_PATH
    };
    FILE*    pFile = NULL;
    TCHAR    szString[sizeOfString];
    TCHAR    szFile[sizeOfFileName];
    errcode  err = 0;

    ObjectLock lock(this);

    // From a filename, passed in as a command text,
    // scan the file placing data in the data array.
    if (m_strCommandText == (BSTR)NULL)
    {
        ATLTRACE( "RRowsetUpdate::FlushData -- "
                  "No filename specified\n");
        return E_FAIL;
    }

    // Open the file
    _tcscpy_s(szFile, sizeOfFileName, OLE2T(m_strCommandText));
```

```
    if ((szFile[0] == _T('\0')) ||
        ((err = _tfopen_s(&pFile, &szFile[0], _T("w"))) != 0))
    {
        ATLTRACE("RUpdateRowset::FlushData -- Could not open file\n");
        return DB_E_NOTABLE;
    }

    // Iterate through the row data and store it.
    for (long l=0; l<m_rgRowData.GetSize(); l++)
    {
        CAgentMan am = m_rgRowData[l];

        _putw((int)am.dwFixed, pFile);

        if (_tcscmp(&am.szCommand[0], _T("")) != 0)
            _stprintf_s(&szString[0], _T("%s\n"), am.szCommand);
        else
            _stprintf_s(&szString[0], _T("%s\n"), _T("NULL"));
        _fputts(szString, pFile);

        if (_tcscmp(&am.szText[0], _T("")) != 0)
            _stprintf_s(&szString[0], _T("%s\n"), am.szText);
        else
            _stprintf_s(&szString[0], _T("%s\n"), _T("NULL"));
        _fputts(szString, pFile);

        if (_tcscmp(&am.szCommand2[0], _T("")) != 0)
            _stprintf_s(&szString[0], _T("%s\n"), am.szCommand2);
        else
            _stprintf_s(&szString[0], _T("%s\n"), _T("NULL"));
        _fputts(szString, pFile);

        if (_tcscmp(&am.szText2[0], _T("")) != 0)
            _stprintf_s(&szString[0], _T("%s\n"), am.szText2);
        else
            _stprintf_s(&szString[0], _T("%s\n"), _T("NULL"));
        _fputts(szString, pFile);
    }

    if (fflush(pFile) == EOF || fclose(pFile) == EOF)
    {
        ATLTRACE("RRowsetUpdate::FlushData -- "
                 "Couldn't flush or close file\n");
    }

    return S_OK;
}
```

Handling Changes

For your provider to handle changes, you first need to make sure your data store (such as a text file or video file) has facilities that enable you to make changes on it. If it does not, you should create that code separately from the provider project.

Handling NULL Data

It is possible that an end user will send **NULL** data. When you write **NULL** values to fields in the data source, there can be potential problems. Imagine an order-taking application that accepts values for city and postal code; it could accept either or both values, but not neither, because in that case delivery would be impossible. You therefore have to restrict certain combinations of **NULL** values in fields that make sense for your application.

As the provider developer, you have to consider how you will store that data, how you will read that data from the data store, and how you specify that to the user. Specifically, you must consider how to change the data status of rowset data in the data source (for example, DataStatus = **NULL**). You decide what value to return when a consumer accesses a field containing a **NULL** value.

Look at the code in the UpdatePV sample; it illustrates how a provider can handle **NULL** data. In UpdatePV, the provider stores **NULL** data by writing the string "NULL" in the data store. When it reads **NULL** data from the data store, it sees that string and then empties the buffer, creating a **NULL** string. It also has an override of **IRowsetImpl::GetDBStatus** in which it returns

DBSTATUS_S_ISNULL if that data value is empty.

Marking Nullable Columns

If you also implement schema rowsets (see **IDBSchemaRowsetImpl**), your implementation should specify in the **DBSCHEMA_COLUMNS** rowset (usually marked in your provider by **C***xxx***SchemaColSchemaRowset**) that the column is nullable.

You also need to specify that all nullable columns contain the **DBCOLUMNFLAGS_ISNULLABLE** value in your version of the **GetColumnInfo**.

In the OLE DB templates implementation, if you fail to mark columns as nullable, the provider assumes that they must contain a value and will not allow the consumer to send it null values.

The following example shows how the **CommonGetColInfo** function is implemented in **CUpdateCommand** (see UpProvRS.cpp) in UpdatePV. Note how the columns have this **DBCOLUMNFLAGS_ISNULLABLE** for nullable columns.

```
////////////////////////////////////////////////////////////////////////////
// CUpdateCommand (in UpProvRS.cpp)

ATLCOLUMNINFO* CommonGetColInfo(IUnknown* pPropsUnk, ULONG* pcCols, bool bBookmark)
{
    static ATLCOLUMNINFO _rgColumns[6];
    ULONG ulCols = 0;

    if (bBookmark)
    {
        ADD_COLUMN_ENTRY_EX(ulCols, OLESTR("Bookmark"), 0,
                            sizeof(DWORD), DBTYPE_BYTES,
                            0, 0, GUID_NULL, CAgentMan, dwBookmark,
                            DBCOLUMNFLAGS_ISBOOKMARK)
        ulCols++;
    }

    // Next set the other columns up.
    // Add a fixed length entry for OLE DB conformance testing purposes
    ADD_COLUMN_ENTRY_EX(ulCols, OLESTR("Fixed"), 1, 4, DBTYPE_UI4,
                        10, 255, GUID_NULL, CAgentMan, dwFixed,
                        DBCOLUMNFLAGS_WRITE |
                        DBCOLUMNFLAGS_ISFIXEDLENGTH)
    ulCols++;

    ADD_COLUMN_ENTRY_EX(ulCols, OLESTR("Command"), 2, 16, DBTYPE_STR,
                        255, 255, GUID_NULL, CAgentMan, szCommand,
                        DBCOLUMNFLAGS_WRITE | DBCOLUMNFLAGS_ISNULLABLE)
    ulCols++;
    ADD_COLUMN_ENTRY_EX(ulCols, OLESTR("Text"), 3, 16, DBTYPE_STR,
                        255, 255, GUID_NULL, CAgentMan, szText,
                        DBCOLUMNFLAGS_WRITE | DBCOLUMNFLAGS_ISNULLABLE)
    ulCols++;

    ADD_COLUMN_ENTRY_EX(ulCols, OLESTR("Command2"), 4, 16, DBTYPE_STR,
                        255, 255, GUID_NULL, CAgentMan, szCommand2,
                        DBCOLUMNFLAGS_WRITE | DBCOLUMNFLAGS_ISNULLABLE)
    ulCols++;
    ADD_COLUMN_ENTRY_EX(ulCols, OLESTR("Text2"), 5, 16, DBTYPE_STR,
                        255, 255, GUID_NULL, CAgentMan, szText2,
                        DBCOLUMNFLAGS_WRITE | DBCOLUMNFLAGS_ISNULLABLE)
    ulCols++;

    if (pcCols != NULL)
    {
        *pcCols = ulCols;
    }

    return _rgColumns;
}
```

Default Values

As with **NULL** data, you have the responsibility to deal with changing default values.

The default of **FlushData** and **Execute** is to return **S_OK**. Therefore, if you do not override this function, the changes appear to succeed (**S_OK** will be returned), but they will not be transmitted to the data store.

In the UpdatePV sample (in Rowset.h), the `SetDBStatus` method handles default values as follows:

```
virtual HRESULT SetDBStatus(DBSTATUS* pdbStatus, CSimpleRow* pRow,
                            ATLCOLUMNINFO* pColInfo)
{
    ATLASSERT(pRow != NULL && pColInfo != NULL && pdbStatus != NULL);

    void* pData = NULL;
    char* pDefaultData = NULL;
    DWORD* pFixedData = NULL;

    switch (*pdbStatus)
    {
        case DBSTATUS_S_DEFAULT:
            pData = (void*)&m_rgRowData[pRow->m_iRowset];
            if (pColInfo->wType == DBTYPE_STR)
            {
                pDefaultData = (char*)pData + pColInfo->cbOffset;
                strcpy(pDefaultData, "Default");
            }
            else
            {
                pFixedData = (DWORD*)((BYTE*)pData +
                                      pColInfo->cbOffset);
                *pFixedData = 0;
                return S_OK;
            }
            break;
        case DBSTATUS_S_ISNULL:
        default:
            break;
    }
    return S_OK;
}
```

Column Flags

If you support default values on your columns, you need to set it using metadata in the **<***provider class***>SchemaRowset** class. Set *m_bColumnHasDefault* = **VARIANT_TRUE**.

You also have the responsibility to set the column flags, which are specified using the **DBCOLUMNFLAGS** enumerated type. The column flags describe column characteristics.

For example, in the `CUpdateSessionColSchemaRowset` class in UpdatePV (in Session.h), the first column is set up this way:

```
// Set up column 1
trData[0].m_ulOrdinalPosition = 1;
trData[0].m_bIsNullable = VARIANT_FALSE;
trData[0].m_bColumnHasDefault = VARIANT_TRUE;
trData[0].m_nDataType = DBTYPE_UI4;
trData[0].m_nNumericPrecision = 10;
trData[0].m_ulColumnFlags = DBCOLUMNFLAGS_WRITE |
                            DBCOLUMNFLAGS_ISFIXEDLENGTH;
lstrcpyW(trData[0].m_szColumnDefault, OLESTR("0"));
m_rgRowData.Add(trData[0]);
```

This code specifies, among other things, that the column supports a default value of 0, that it be writeable, and that all data in the column have the same length. If you want the data in a column to have variable length, you would not set this flag.

For a complete list of column flag values, see "DBCOLUMNFLAGS enumerated type" in IColumnsInfo::GetColumnInfo.

See Also

Reference

Creating an OLE DB Provider

Other Resources

Chapter 5: Updating Data in Rowsets

Working with OLE DB Provider Templates

To learn how to work with the OLE DB provider templates, see the following topics:

- Adding an Interface to Your Provider
- Referencing a Property in Your Provider
- Setting Properties in Your Provider
- Dynamically Binding Columns in Your Provider
- Supporting Free Threading in Your Provider
- Testing Your Provider
- Debugging Your Provider
- Converting Data Not Supported by the Provider

See Also

Concepts

OLE DB Provider Templates Reference
OLE DB Consumer Templates (C++)

Adding an Interface to Your Provider

Determine which object you want to add the interface to (usually data source, rowset, command, or session objects created by the OLE DB Provider Wizard). It is possible that the object you need to add the interface to is one that your provider does not currently support. In that case, run the ATL OLE DB Provider Wizard to create the object. Right-click the project in Class View, click **Add Class** from the **Add** menu, and then click **ATL OLE DB Provider**. You might want to put the interface code in a separate directory and then copy the files to your provider project.

If you created a new class to support the interface, make the object inherit from that class. For example, you might add the class **IRowsetIndexImpl** to a rowset object:

```
template <class Creator>
class CAgentRowset :
public CRowsetImpl< CAgentRowset<Creator>, CAgentMan, Creator>,
   public IRowsetIndexImpl< ... >
```

Add the interface to **COM_MAP** in the object using the COM_INTERFACE_ENTRY macro. If there is no map, create one. For example:

```
BEGIN_COM_MAP(CAgentRowset)
     COM_INTERFACE_ENTRY(IRowsetIndex)
END_COM_MAP()
```

For the rowset object, chain the map of its parent object so that the object can delegate to the parent class. In this example, add the COM_INTERFACE_ENTRY_CHAIN macro to the map:

```
BEGIN_COM_MAP(CAgentRowset)
     COM_INTERFACE_ENTRY(IRowsetIndex)
     COM_INTERFACE_ENTRY_CHAIN(CRowsetImpl)
END_COM_MAP()
```

See Also

Concepts

Working with OLE DB Provider Templates

Referencing a Property in Your Provider

Find the property group and property ID for the property you want. For more information, see OLE DB Properties in the *OLE DB Programmer's Reference*.

The following example assumes that you are trying to get a property from the rowset. The code for using the session or command is similar, but uses a different interface.

Create a CDBPropSet object using the property group as the parameter to the constructor. For example:

```
CDBPropSet propset(DBPROPSET_ROWSET);
```

Call AddProperty, passing it the property ID and a value to be assigned to the property. The type of the value depends on the property you are using.

```
CDBPropSet propset(DBPROPSET_ROWSET);
propset.AddProperty(DBPROP_IRowsetChange, true);
propset.AddProperty(DBPROP_UPDATABILITY,
DBPROPVAL_UP_INSERT | DBPROPVAL_UP_CHANGE | DBPROPVAL_UP_DELETE);
```

Use the **IRowset** interface to call **GetProperties**. Pass the property set as a parameter. Here is the final code:

```
CAgentRowset<CMyProviderCommand>* pRowset = (CAgentRowset<CMyProviderCommand>*) pThis;

CComQIPtr<IRowsetInfo, &IID_IRowsetInfo> spRowsetProps = pRowset;

DBPROPIDSET set;
set.AddPropertyID(DBPROP_BOOKMARKS);
DBPROPSET* pPropSet = NULL;
ULONG ulPropSet = 0;
HRESULT hr;

if (spRowsetProps)
   hr = spRowsetProps->GetProperties(1, &set, &ulPropSet, &pPropSet);

if (pPropSet)
{
   CComVariant var = pPropSet->rgProperties[0].vValue;
   CoTaskMemFree(pPropSet->rgProperties);
   CoTaskMemFree(pPropSet);

   if (SUCCEEDED(hr) && (var.boolVal == VARIANT_TRUE))
   {
      ...  // Use property here
   }
}
```

See Also

Concepts

Working with OLE DB Provider Templates

Setting Properties in Your Provider

Find the property group and property ID for the property you want. For more information, see OLE DB Properties in the *OLE DB Programmer's Reference*.

In the provider code generated by the wizard, find the property map corresponding to the property group. The name of the property group usually corresponds to the name of the object. Command and rowset properties can be found in the command or rowset; data source and initialization properties can be found in the data source object.

In the property map, add a PROPERTY_INFO_ENTRY_EX macro. PROPERTY_INFO_ENTRY_EX takes four parameters:

- The property ID corresponding to your property. You must remove the first seven characters ("DBPROP_") from the front of the property name. For example, if you want to add **DBPROP_MAXROWS**, pass `MAXROWS` as the first element. If this is a custom property, pass the full GUID name (for example, `DBMYPROP_MYPROPERTY`).
- The variant type of the property (in OLE DB Properties in the *OLE DB Programmer's Reference*). Enter the **VT_** type (such as **VT_BOOL** or **VT_I2**) corresponding to the data type.
- Flags to indicate whether the property is readable and writable and the group to which it belongs. For example, the following code indicates a read/write property belonging to the rowset group:

  ```
  DBPROPFLAGS_ROWSET | DBPROPFLAGS_READ | DBPROPFLAGS_WRITE
  ```

- The base value of the property. This might be **VARIANT_FALSE** for a Boolean type or zero for an integer type, for example. The property has this value unless it is changed.

  Note

  Some properties are connected or chained to other properties, such as bookmarks or updating. When a consumer sets one property to true, another property might also be set. The OLE DB provider templates support this through the met hod CUtlProps::OnPropertyChanged.

Properties Ignored by Microsoft OLE DB Providers

The Microsoft OLE DB Providers ignore the following OLE DB properties:

- **DBPROP_MAXROWS** only works for read-only providers (that is, where DBPROP_IRowsetChange and DBPROP_IRowsetUpdate are false); otherwise this property is not supported.
- **DBPROP_MAXPENDINGROWS** is ignored; the provider specifies its own limit.
- **DBPROP_MAXOPENROWS** is ignored; the provider specifies its own limit.
- **DBPROP_CANHOLDROWS** is ignored; the provider specifies its own limit.

See Also

Concepts

Working with OLE DB Provider Templates

Dynamically Binding Columns in Your Provider

Make sure you really need dynamic column binding. You might need it because:

- Your rowset columns are not defined at compile time.
- You support an element such as bookmarks that adds columns.

To implement dynamic column binding

1. Remove any **PROVIDER_COLUMN_MAP**s from your code.
2. In the user record (your structure), add the following declaration:

   ```
   static ATLCOLUMNINFO* GetColumnInfo(void* pThis, ULONG* pcCols);
   ```

3. Implement the **GetColumnInfo** function. This function lays out how the information is stored. You might need to get properties or other information for this function. You might want to create a macro, similar to the COLUMN_ENTRY macro, to add your own information.

   The following example shows a **GetColumnInfo** function.

   ```
   // Check the property flag for bookmarks, if it is set, set the zero
   // ordinal entry in the column map with the bookmark information.
   CAgentRowset* pRowset = (CAgentRowset*) pThis;
   CComQIPtr<IRowsetInfo, &IID_IRowsetInfo> spRowsetProps = pRowset;

   CDBPropIDSet set(DBPROPSET_ROWSET);
   set.AddPropertyID(DBPROP_BOOKMARKS);
   DBPROPSET* pPropSet = NULL;
   ULONG ulPropSet = 0;
   HRESULT hr;

   if (spRowsetProps)
      hr = spRowsetProps->GetProperties(1, &set, &ulPropSet, &pPropSet);

   if (pPropSet)
   {
      CComVariant var = pPropSet->rgProperties[0].vValue;
      CoTaskMemFree(pPropSet->rgProperties);
      CoTaskMemFree(pPropSet);

      if (SUCCEEDED(hr) && (var.boolVal == VARIANT_TRUE))
      {
         ADD_COLUMN_ENTRY_EX(ulCols, OLESTR("Bookmark"), 0, sizeof(DWORD), DBTYPE_BYTES,
            0, 0, GUID_NULL, CAgentMan, dwBookmark, DBCOLUMNFLAGS_ISBOOKMARK)
         ulCols++;
      }
   }

   // Next, set up the other columns.
   ADD_COLUMN_ENTRY(ulCols, OLESTR("Command"), 1, 256, DBTYPE_STR, 0xFF, 0xFF,
      GUID_NULL, CAgentMan, szCommand)
   ulCols++;
   ADD_COLUMN_ENTRY(ulCols, OLESTR("Text"), 2, 256, DBTYPE_STR, 0xFF, 0xFF,
      GUID_NULL, CAgentMan, szText)
   ulCols++;
   ```

```
ADD_COLUMN_ENTRY(ulCols, OLESTR("Command2"), 3, 256, DBTYPE_STR, 0xFF, 0xFF,
   GUID_NULL, CAgentMan, szCommand2)
ulCols++;
ADD_COLUMN_ENTRY(ulCols, OLESTR("Text2"), 4, 256, DBTYPE_STR, 0xFF, 0xFF,
   GUID_NULL, CAgentMan, szText2)
ulCols++;


if (pcCols != NULL)
   *pcCols = ulCols;

return _rgColumns;
}
```

See Also

Concepts

Working with OLE DB Provider Templates

Supporting Free Threading in Your Provider

All the OLE DB provider classes are thread-safe, and registry entries are set accordingly. It is a good idea to support free threading to help provide a high level of performance in multiuser situations. To help keep your provider thread-safe, you must verify that your code is blocked properly. Whenever you write or store data, you must block the access with critical sections.

Each OLE DB provider template object has its own critical section. To make blocking easier, each new class you create should be a template class taking the parent class name as an argument.

The following example shows how to block your code:

```
template <class T>
class CMyObject<T> : public...

HRESULT MyObject::MyMethod(void)
{
   T* pT = (T*)this;      // this gets the parent class

// You don't need to do anything if you are only reading information

// If you want to write information, do the following:
   pT->Lock();         // engages critical section in the object
   ...;                // write whatever information you wish
   pT->Unlock();       // disengages the critical section
}
```

For more information about how to protect critical sections with **Lock** and **Unlock**, see Multithreading: How To Use the Synchronization Classes.

You must also verify that any methods you override (such as `Execute`) are thread-safe.

See Also

Concepts

Working with OLE DB Provider Templates

Testing Your Provider

Before you release a provider, you should perform the following tests, in the order indicated. These tests ensure that the provider functions properly for most potential users.

1. Test the provider using a consumer application written with the OLE DB consumer templates. The test consumer should cover all functional areas of your provider (all code that you have added or modified).
2. Test the provider using a consumer application written with ADO. Most developers (especially Microsoft Visual Basic and Microsoft C# developers) use ADO or ADO.NET for consumer applications. The test consumer should cover all functional areas of your provider. For an example of an ADO consumer application, see ADO Code Examples in Microsoft Visual Basic.
3. Run the OLE DB conformance tests (including ADO conformance tests) to ensure that your provider meets the level 0 standard for OLE DB providers. (For an explanation of level 0, see "Minimum Provider Functionality" in "OLE DB Leveling: Choosing the Right Interfaces" at http://www.microsoft.com/data/oledb/techinfo/oledbleveling2.htm.) These tests and associated documentation are included with Visual C++ in the Data Access SDK. These tests also help to ensure that your provider runs well when aggregated by other service providers and are especially useful if you modify or add properties. For more information about the conformance tests, see the Readme file for the Data Access SDK, which is located on one of the Visual Studio CDs.

See Also

Concepts

Working with OLE DB Provider Templates

Debugging Your Provider

There are two ways to debug your provider:

- Because providers are created in process, you can create some consumer code using the OLE DB consumer templates and step into the provider normally.
- You can use the ITEST utility that comes with Visual C++.

To use the ITEST utility

1. Open the provider project.
2. On the **Projects** menu, click **Settings**.
3. In the **Property Pages** dialog box, click the **Debug** tab.
4. In the **Executable for Debug Session** box, select the ITEST application.
5. Set breakpoints, and then debug as usual.

See Also

Concepts

Working with OLE DB Provider Templates

Converting Data Not Supported by the Provider

When the consumer requests a data type that is not supported by the provider, the OLE DB provider template code for **IRowsetImpl::GetData** calls Msdadc.dll to convert the data type.

If you implement an interface like **IRowsetChange** that requires data conversion, you can call Msdaenum.dll to do the conversion. Use **GetData**, defined in Atldb.h, as an example.

See Also

Concepts

Working with OLE DB Provider Templates

Advanced Provider Techniques

For more information about advanced provider techniques, see the following topics:

- Supporting Notifications
- Supporting Schema Rowsets
- Provider Support for Bookmarks
- Passing OLE DB Conformance Tests
- OLE DB Resource Pooling and Services

See Also

Concepts

OLE DB Provider Templates Reference
OLE DB Consumer Templates (C++)

Supporting Notifications

Implementing Connection Point Interfaces on the Provider and Consumer

To implement notifications, a provider class must inherit from IRowsetNotifyCP and IConnectionPointContainer.

IRowsetNotifyCP implements the provider site for the connection point interface IRowsetNotify. **IRowsetNotifyCP** implements broadcast functions to advise listeners on the connection point **IID_IRowsetNotify** of changes to the contents of the rowset.

Note that you must also implement and register **IRowsetNotify** on the consumer (also known as the sink) using IRowsetNotifyImpl so that the consumer can handle notifications. For information about implementing the connection point interface on the consumer, see Receiving Notifications.

In addition, the class must also contain a map that defines the connection point entry, like this:

```
BEGIN_CONNECTION_POINT_MAP
   CONNECTIONPOINT_ENTRY (IID_IRowsetNotify)
END_CONNECTION_POINT_MAP
```

Adding IRowsetNotify

To add **IRowsetNotify**, you need to add `IConnectionPointContainerImpl<rowset-name>` and `IRowsetNotifyCP<rowset-name>` to your inheritance chain.

For example, here is the inheritance chain for `RUpdateRowset` in UpdatePV:

Note

The sample code might differ from what is listed here; you should regard the sample code as the more up-to-date version.

```
/////////////////////////////////////////////////////////////////////////////
// class RUpdateRowset (in rowset.h)

class RUpdateRowset :
public CRowsetImpl< RUpdateRowset, CAgentMan, CUpdateCommand,
         CAtlArray< CAgentMan, CAtlArray<CAgentMan> >, CSimpleRow,
         IRowsetScrollImpl< RUpdateRowset, IRowsetScroll > >,
      public IRowsetUpdateImpl< RUpdateRowset, CAgentMan >,
      public IConnectionPointContainerImpl<RUpdateRowset>,
      public IRowsetNotifyCP<RUpdateRowset>
```

Setting COM Map Entries

You also need to add the following to the COM map in your rowset:

```
COM_INTERFACE_ENTRY(IConnectionPointContainer)
COM_INTERFACE_ENTRY_IMPL(IConnectionPointContainer)
```

These macros allow anyone calling **QueryInterface** for your connection point container (the basis of **IRowsetNotify**) to find the requested interface on your provider. For an example of how to use connection points, see the ATL POLYGON sample and tutorial.

Setting Connection Point Map Entries

You also need to add a connection point map. It should look something like:

```
BEGIN_CONNECTION_POINT_MAP(rowset-name)
     CONNECTION_POINT_ENTRY(_uuidof(IRowsetNotify))
END_CONNECTION_POINT_MAP()
```

This connection point map allows a component looking for the **IRowsetNotify** interface to find it in your provider.

Setting Properties

You also need to add the following properties to your provider. You only need to add properties based on the interfaces that you support.

Property	Add if you support
DBPROP_IConnectionPointContainer	Always
DBPROP_NOTIFICATIONGRANULARITY	Always
DBPROP_NOTIFICATIONPHASES	Always
DBPROP_NOTIFYCOLUMNSET	**IRowsetChange**
DBPROP_NOTIFYROWDELETE	**IRowsetChange**
DBPROP_NOTIFYROWINSERT	**IRowsetChange**
DBPROP_NOTIFYROWSETFETCHPOSITIONCHANGE	Always
DBPROP_NOTIFYROWFIRSTCHANGE	**IRowsetUpdate**
DBPROP_NOTIFYROWSETRELEASE	Always
DBPROP_NOTIFYROWUNDOCHANGE	**IRowsetUpdate**
DBPROP_NOTIFYROWUNDODELETE	**IRowsetUpdate**
DBPROP_NOTIFYROWUNDOINSERT	**IRowsetUpdate**
DBPROP_NOTIFYROWUPDATE	**IRowsetUpdate**

Most of the implementation for the notifications is already embedded in the OLE DB Provider Templates. Due to a compiler feature in Visual C++ .NET, if you do not add **IRowsetNotifyCP** to your inheritance chain, the compiler removes all that code from your compilation stream, thus making your code size smaller.

See Also

Concepts

Advanced Provider Techniques

Supporting Schema Rowsets

Schema rowsets allow consumers to obtain information about a data store without knowing its underlying structure, or schema. For example, a data store might have tables organized into a user-defined hierarchy, so there would be no way to ensure knowledge of the schema except by reading it. (As another example, note that the Visual C++ wizards use schema rowsets to generate accessors for the consumer.) To allow the consumer to do this, the provider's session object exposes methods on the IDBSchemaRowset interface. In Visual C++ applications, you use the IDBSchemaRowsetImpl class to implement **IDBSchemaRowset**.

IDBSchemaRowsetImpl supports the following methods:

- CheckRestrictions checks the validity of restrictions against a schema rowset.
- CreateSchemaRowset implements a COM object creator function for the object specified by the template parameter.
- SetRestrictions specifies which restrictions you support on a particular schema rowset.
- IDBSchemaRowset::GetRowset returns a schema rowset (inherited from interface).
- GetSchemas returns a list of schema rowsets accessible by **IDBSchemaRowsetImpl::GetRowset** (inherited from interface).

ATL OLE DB Provider Wizard Support

The ATL OLE DB Provider Wizard creates three schema classes in the session header file:

- **C** *ShortName* **SessionTRSchemaRowset**
- **C** *ShortName* **SessionColSchemaRowset**
- **C** *ShortName* **SessionPTSchemaRowset**

These classes respond to consumer requests for schema information; note that the OLE DB specification requires that these three schema rowsets be supported:

- **C** *ShortName* **SessionTRSchemaRowset** handles requests for table information (the **DBSCHEMA_TABLES** schema rowset).
- **C** *ShortName* **SessionColSchemaRowset** handles requests for column information (the **DBSCHEMA_COLUMNS** schema rowset). The wizard supplies sample implementations for these classes, which return schema information for a DOS provider.
- **C** *ShortName* **SessionPTSchemaRowset** handles requests for schema information about the provider type (the **DBSCHEMA_PROVIDER_TYPES** schema rowset). The default implementation provided by the wizard returns **S_OK**.

You can customize these classes to handle schema information appropriate to your provider:

- In **C***ShortName***SessionTRSchemaRowset**, you must fill out the catalog, table, and description fields (**trData.m_szType**, **trData.m_szTable**, and **trData.m_szDesc**). The wizard-generated example uses only one row (table). Other providers might return more than one table.
- In **C***ShortName***SessionColSchemaRowset**, you pass the name of the table as a **DBID**.

Setting Restrictions

An important concept in schema rowset support is setting restrictions, which you do using **SetRestrictions**. Restrictions allow consumers to fetch only matching rows (for example, find all the columns in the table "MyTable"). Restrictions are optional, and in the case in which none are supported (the default), all data is always returned. For an example of a provider that does support restrictions, see the UpdatePV sample.

Setting up the Schema Map

Set up a schema map such as this one in Session.h in UpdatePV:

```
BEGIN_SCHEMA_MAP(CUpdateSession)
    SCHEMA_ENTRY(DBSCHEMA_TABLES, CUpdateSessionTRSchemaRowset)
```

```
    SCHEMA_ENTRY(DBSCHEMA_COLUMNS, CUpdateSessionColSchemaRowset)
    SCHEMA_ENTRY(DBSCHEMA_PROVIDER_TYPES, CUpdateSessionPTSchemaRowset)
END_SCHEMA_MAP()
```

To support **IDBSchemaRowset**, you must support **DBSCHEMA_TABLES**, **DBSCHEMA_COLUMNS**, and **DBSCHEMA_PROVIDER_TYPES**. You can add additional schema rowsets at your discretion.

Declare a schema rowset class with an `Execute` method such as `CUpdateSessionTRSchemaRowset` in UpdatePV:

```
class CUpdateSessionTRSchemaRowset :
    public CSchemaRowsetImpl < CUpdateSessionTRSchemaRowset,
                               CTABLESRow, CUpdateSession >
...
// Execute looks like this; what pointers does the consumer use?
    HRESULT Execute(DBROWCOUNT* pcRowsAffected,
                    ULONG cRestrictions, const VARIANT* rgRestrictions)
```

Note that `CUpdateSession` inherits from **IDBSchemaRowsetImpl**, so it has all the restriction handling methods. Using `CSchemaRowsetImpl`, declare three child classes (listed in the schema map above): `CUpdateSessionTRSchemaRowset`, `CUpdateSessionColSchemaRowset`, and `CUpdateSessionPTSchemaRowset`. Each of these child classes has an `Execute` method that handles its respective set of restrictions (search criteria). Each `Execute` method compares the values of the `cRestrictions` and `rgRestrictions` parameters. See the description of these parameters in SetRestrictions.

For more information about which restrictions correspond to a particular schema rowset, consult the table of schema rowset GUIDs in IDBSchemaRowset in the *OLE DB Programmer's Reference* in the Platform SDK.

For example, if you supported the **TABLE_NAME** restriction on **DBSCHEMA_TABLES**, you would do the following:

First, look up **DBSCHEMA_TABLES** and see that it supports four restrictions (in order).

Schema rowset restriction	Restriction value
TABLE_CATALOG	0x1 (binary 1)
TABLE_SCHEMA	0x2 (binary 10)
TABLE_NAME	0x4 (binary 100)
TABLE_TYPE	0x8 (binary 1000)

Next, note that there is one bit for each restriction. Because you want to support **TABLE_NAME** only, you would return 0x4 in the *rgRestrictions* element. If you supported **TABLE_CATALOG** and **TABLE_NAME**, you would return 0x5 (binary 101).

By default, the implementation returns 0 (does not support any restrictions) for any request. UpdatePV is an example of a provider that does support restrictions.

Example

This code is taken from the UpdatePV sample. UpdatePv supports the three required schema rowsets: **DBSCHEMA_TABLES**, **DBSCHEMA_COLUMNS**, and **DBSCHEMA_PROVIDER_TYPES**. As an example of how to implement schema support in your provider, this topic takes you through implementing the **DBSCHEMA_TABLE** rowset.

Note
The sample code might differ from what is listed here; you should regard the sample code as the more up-to-date version.

The first step in adding schema support is to determine which restrictions you are going to support. To determine which restrictions are available for your schema rowset, look at the OLE DB specification for the definition of **IDBSchemaRowset**. Following the main definition, you see a table containing the schema rowset name, the number of restrictions, and the restriction columns. Select the schema rowset you want to support and make a note of the number of restrictions and restriction columns. For example, **DBSCHEMA_TABLES** supports four restrictions (**TABLE_CATALOG**, **TABLE_SCHEMA**, **TABLE_NAME**, and **TABLE_TYPE**):

```
void SetRestrictions(ULONG cRestrictions, GUID* rguidSchema,
```

```
    ULONG* rgRestrictions)
{
    for (ULONG l=0; l<cRestrictions; l++)
    {
        if (InlineIsEqualGUID(rguidSchema[l], DBSCHEMA_TABLES))
            rgRestrictions[l] = 0x0C;
        else if (InlineIsEqualGUID(rguidSchema[l], DBSCHEMA_COLUMNS))
                rgRestrictions[l] = 0x04;
             else if (InlineIsEqualGUID(rguidSchema[l],
                                        DBSCHEMA_PROVIDER_TYPES))
                    rgRestrictions[l] = 0x00;
    }
}
```

A bit represents each restriction column. If you want to support a restriction (that is, you can query by it), set that bit to a 1. If you do not want to support a restriction, set that bit to zero. From the line of code above, UpdatePV supports the **TABLE_NAME** and **TABLE_TYPE** restrictions on the **DBSCHEMA_TABLES** rowset. These are the third (bit mask 100) and fourth (bit mask 1000) restrictions. Therefore, the bitmask for UpdatePv is 1100 (or 0x0C):

```
    if (InlineIsEqualGUID(rguidSchema[l], DBSCHEMA_TABLES))
        rgRestrictions[l] = 0x0C;
```

The following `Execute` function is similar to those in regular rowsets. You have three arguments: `pcRowsAffected`, `cRestrictions`, and `rgRestrictions`. The `pcRowsAffected` variable is an output parameter that the provider can return the count of rows in the schema rowset. The `cRestrictions` parameter is an input parameter containing the number of restrictions passed by the consumer to the provider. The `rgRestrictions` parameter is an array of **VARIANT** values that contain the restriction values.

```
    HRESULT Execute(DBROWCOUNT* pcRowsAffected, ULONG cRestrictions,
                    const VARIANT* rgRestrictions)
```

The `cRestrictions` variable is based on the total number of restrictions for a schema rowset, regardless of whether the provider supports them. Because UpdatePv supports two restrictions (the third and fourth), this code only looks for a `cRestrictions` value greater than or equal to three.

The value for the **TABLE_NAME** restriction is stored in `rgRestrictions[2]` (again, the third restriction in a zero-based array is 2). You need to check that the restriction is not **VT_EMPTY** to actually support it. Note that **VT_NULL** is not equal to **VT_EMPTY**. **VT_NULL** specifies a valid restriction value.

The UpdatePv definition of a table name is a fully qualified path name to a text file. Extract the restriction value and then attempt to open the file to ensure that the file does actually exist. If the file does not exist, return **S_OK**. This might seem a bit backwards but what the code is really telling the consumer is that there were no supported tables by the name specified. The **S_OK** return means that the code executed correctly.

```
USES_CONVERSION;
enum {
            sizeOfszFile = 255
};
CTABLESRow  trData;
FILE        *pFile = NULL;
TCHAR       szFile[ sizeOfszFile ];
errcode     err = 0;

// Handle any restrictions sent to us. This only handles
// the TABLE_NAME & TABLE_TYPE restictions (the 3rd and 4th
// restrictions in DBSCHEMA_TABLES...look in IDBSchemaRowsets
// in part 2 of the prog. ref) so your restrictions are 0x08 & 0x04
// for a total of (0x0C)
if (cRestrictions >= 3 && rgRestrictions[2].vt != VT_EMPTY)
{
    CComBSTR bstrName = rgRestrictions[2].bstrVal;
    if ((rgRestrictions[2].vt == VT_BSTR) && (bstrName != (BSTR)NULL))
    {
        // Check to see if the file exists
```

```
        _tcscpy_s(&szFile[0], sizeOfszFile, OLE2T(bstrName));
        if (szFile[0] == _T('\0') ||
           ((err = _tfopen(&pFile, &szFile[0], _T("r"))) == 0))
        {
            return S_OK;// Their restriction was invalid return no data
        }
        else
        {
            fclose(pFile);
        }
    }
}
```

Supporting the fourth restriction (**TABLE_TYPE**) is similar to the third restriction. Check to see that the value is not **VT_EMPTY**. This restriction only returns the table type, **TABLE**. To determine the valid values for the **DBSCHEMA_TABLES**, look in Appendix B of the *OLE DB Programmer's Reference* in the **TABLES** rowset section.

```
// TABLE_TYPE restriction:
if (cRestrictions >=4 && rgRestrictions[3].vt != VT_EMPTY)
{
    CComBSTR bstrType = rgRestrictions[3].bstrVal;
    if ((rgRestrictions[3].vt == VT_BSTR) && (bstrType != (BSTR)NULL))
    {
        // This is kind of a blind restriction.
        // This only actually supports
        // TABLES so if you get anything else,
        // just return an empty rowset.
        if (_tcscmp(_T("TABLE"), OLE2T(bstrType)) != 0)
            return S_OK;
    }
}
```

This is where you actually create a row entry for the rowset. The variable `trData` corresponds to **CTABLESRow**, a structure defined in the OLE DB provider templates. **CTABLESRow** corresponds to the **TABLES** rowset definition in Appendix B of the OLE DB specification. You only have one row to add because you can only support one table at a time.

```
// Bring over the data:
lstrcpyW(trData.m_szType, OLESTR("TABLE"));
lstrcpyW(trData.m_szDesc, OLESTR("The Directory Table"));
lstrcpynW(trData.m_szTable, T2OLE(szFile),
          SIZEOF_MEMBER(CTABLESRow, m_szTable));
```

UpdatePV sets only three columns: **TABLE_NAME**, **TABLE_TYPE**, and **DESCRIPTION**. You should make a note of the columns for which you return information, because you need this information when you implement **GetDBStatus**:

```
    _ATLTRY
    {
        m_rgRowData.Add(trData);
    }
    _ATLCATCHALL()
    {
        return E_OUTOFMEMORY;
    }
    //if (!m_rgRowData.Add(trData))
    //    return E_OUTOFMEMORY;
    *pcRowsAffected = 1;
    return S_OK;
}
```

The **GetDBStatus** function is very important to the correct operation of the schema rowset. Because you do not return data for every column in the **TABLES** rowset, you need to specify which columns you return data for and which you do not.

```
virtual DBSTATUS GetDBStatus(CSimpleRow* , ATLCOLUMNINFO* pColInfo)
{
```

```
    ATLASSERT(pColInfo != NULL);

    switch(pColInfo->iOrdinal)
    {
    case 3:     // TABLE_NAME
    case 4:     // TABLE_TYPE
    case 6:     // DESCRIPTION
        return DBSTATUS_S_OK;
        break;
    default:
        return DBSTATUS_S_ISNULL;
    break;
    }
}
```

Because your `Execute` function returns data for the **TABLE_NAME**, **TABLE_TYPE**, and **DESCRIPTION** fields from the **TABLES** rowset, you can look in Appendix B of the OLE DB specification and determine (by counting from the top down) that they are ordinals 3, 4, and 6. For each of those columns, return **DBSTATUS_S_OK**. For all the other columns, return **DBSTATUS_S_ISNULL**. It is important to return this status, because a consumer might not understand that the value you return is **NULL** or something else. Again, note that **NULL** is not equivalent to empty.

For more information about the OLE DB schema rowset interface, see the IDBSchemaRowset interface in the OLE DB Programmer's Reference.

For information about how consumers can use **IDBSchemaRowset** methods, see Obtaining Metadata with Schema Rowsets.

For an example of an provider that supports schema rowsets, see the UpdatePV sample.

See Also

Concepts

Advanced Provider Techniques

Provider Support for Bookmarks

The example in this topic adds the **IRowsetLocate** interface to the `CMyProviderRowset` class. In almost all cases, you start by adding an interface to an existing COM object. You can then test it by adding more calls from the consumer templates. The example demonstrates how to:

- Add an interface to a provider.
- Dynamically determine the columns to return to the consumer.
- Add bookmark support.

The **IRowsetLocate** interface inherits from the **IRowset** interface. To add the **IRowsetLocate** interface, inherit `CMyProviderRowset` from IRowsetLocateImpl.

Adding the **IRowsetLocate** interface is a bit different from most interfaces. To make the VTABLEs line up, the OLE DB provider templates have a template parameter to handle the derived interface. The following code shows the new inheritance list:

```
/////////////////////////////////////////////////////////////////////
// MyProviderRS.h

// CMyProviderRowset
class CMyProviderRowset : public CRowsetImpl< CMyProviderRowset,
      CTextData, CMyProviderCommand, CAtlArray<CTextData>,
      CSimpleRow,
          IRowsetLocateImpl<CMyProviderRowset, IRowsetLocate> >
```

The fourth, fifth, and sixth parameters are all added. This example uses the defaults for the fourth and fifth parameters but specify **IRowsetLocateImpl** as the sixth parameter. **IRowsetLocateImpl** is an OLE DB template class that takes two template parameters: these hook up the **IRowsetLocate** interface to the `CMyProviderRowset` class. To add most interfaces, you can skip this step and move to the next one. Only the **IRowsetLocate** and **IRowsetScroll** interfaces need to be handled in this way.

You then need to tell the `CMyProviderRowset` to call **QueryInterface** for the **IRowsetLocate** interface. Add the line `COM_INTERFACE_ENTRY(IRowsetLocate)` to the map. The interface map for `CMyProviderRowset` should appear as shown in the following code:

```
/////////////////////////////////////////////////////////////////////
// MyProviderRS.h

typedef CRowsetImpl< CMyProviderRowset, CTextData, CMyProviderCommand, CAtlArray<CTextData>
, CSimpleRow, IRowsetLocateImpl<CMyProviderRowset, IRowsetLocate> > _RowsetBaseClass;

BEGIN_COM_MAP(CMyProviderRowset)
   COM_INTERFACE_ENTRY(IRowsetLocate)
   COM_INTERFACE_ENTRY_CHAIN(_RowsetBaseClass)
END_COM_MAP()
```

You also need to hook your map into the **CRowsetImpl** class. Add in the COM_INTERFACE_ENTRY_CHAIN macro to hook in the **CRowsetImpl** map. Also, create a typedef called `RowsetBaseClass` that consists of the inheritance information. This typedef is arbitrary and can be ignored.

Finally, handle the **IColumnsInfo::GetColumnsInfo** call. You would normally use the PROVIDER_COLUMN_ENTRY macros to do this. However, a consumer might want to use bookmarks. You must be able to change the columns the provider returns depending on whether the consumer asks for a bookmark.

To handle the **IColumnsInfo::GetColumnsInfo** call, delete the **PROVIDER_COLUMN** map in the `CTextData` class. The PROVIDER_COLUMN_MAP macro defines a function `GetColumnInfo`. You need to define your own `GetColumnInfo` function. The function declaration should look like this:

```
/////////////////////////////////////////////////////////////////////
// MyProviderRS.H

class CTextData
```

```
{
   ...
     // NOTE: Be sure you removed the PROVIDER_COLUMN_MAP!
   static ATLCOLUMNINFO* GetColumnInfo(CMyProviderRowset* pThis,
        ULONG* pcCols);
   static ATLCOLUMNINFO* GetColumnInfo(CMyProviderCommand* pThis,
        ULONG* pcCols);
...
};
```

Then, implement the `GetColumnInfo` function in the MyProviderRS.cpp file as follows:

```
////////////////////////////////////////////////////////////////////
// MyProviderRS.cpp

template <class TInterface>
ATLCOLUMNINFO* CommonGetColInfo(IUnknown* pPropsUnk, ULONG* pcCols)
{
   static ATLCOLUMNINFO _rgColumns[5];
   ULONG ulCols = 0;

   CComQIPtr<TInterface> spProps = pPropsUnk;

   CDBPropIDSet set(DBPROPSET_ROWSET);
   set.AddPropertyID(DBPROP_BOOKMARKS);
   DBPROPSET* pPropSet = NULL;
   ULONG ulPropSet = 0;
   HRESULT hr;

   if (spProps)
      hr = spProps->GetProperties(1, &set, &ulPropSet, &pPropSet);

   // Check the property flag for bookmarks, if it is set, set the
// zero ordinal entry in the column map with the bookmark
// information.

   if (pPropSet)
   {
      CComVariant var = pPropSet->rgProperties[0].vValue;
      CoTaskMemFree(pPropSet->rgProperties);
      CoTaskMemFree(pPropSet);

      if ((SUCCEEDED(hr) && (var.boolVal == VARIANT_TRUE)))
      {
         ADD_COLUMN_ENTRY_EX(ulCols, OLESTR("Bookmark"), 0,
                     sizeof(DWORD), DBTYPE_BYTES,
            0, 0, GUID_NULL, CAgentMan, dwBookmark,
                        DBCOLUMNFLAGS_ISBOOKMARK)
         ulCols++;
      }

   }


   // Next set the other columns up.
   ADD_COLUMN_ENTRY_EX(ulCols, OLESTR("Field1"), 1, 16, DBTYPE_STR,
          0xFF, 0xFF, GUID_NULL, CTextData, szField1)
   ulCols++;
   ADD_COLUMN_ENTRY_EX(ulCols, OLESTR("Field2"), 2, 16, DBTYPE_STR,
       0xFF, 0xFF, GUID_NULL, CTextData, szField2)
   ulCols++;

   if (pcCols != NULL)
      *pcCols = ulCols;

   return _rgColumns;
}
```

```
ATLCOLUMNINFO* CTextData::GetColumnInfo(CMyProviderCommand* pThis,
     ULONG* pcCols)
{
   return CommonGetColInfo<ICommandProperties>(pThis->GetUnknown(),
        pcCols);
}


ATLCOLUMNINFO* CAgentMan::GetColumnInfo(RUpdateRowset* pThis, ULONG* pcCols)
{
   return CommonGetColInfo<IRowsetInfo>(pThis->GetUnknown(), pcCols);
}
```

`GetColumnInfo` first checks to see whether a property called **DBPROP_IRowsetLocate** is set. OLE DB has properties for each of the optional interfaces off the rowset object. If the consumer wants to use one of these optional interfaces, it sets a property to true. The provider can then check this property and take special action based on it.

In your implementation, you get the property by using the pointer to the command object. The `pThis` pointer represents the rowset or command class. Because you use templates here, you have to pass this in as a **void** pointer or the code does not compile.

Specify a static array to contain the column information. If the consumer does not want the bookmark column, an entry in the array is wasted. You can dynamically allocate this array, but you would need to make sure to destroy it properly. This example defines and uses the macros ADD_COLUMN_ENTRY and ADD_COLUMN_ENTRY_EX to insert the information into the array. You can add the macros to the MyProviderRS.H file as shown in the following code:

```
//////////////////////////////////////////////////////////////////////
// MyProviderRS.h

#define ADD_COLUMN_ENTRY(ulCols, name, ordinal, colSize, type, precision, scale, guid, data
Class, member) \
   _rgColumns[ulCols].pwszName = (LPOLESTR)name; \
   _rgColumns[ulCols].pTypeInfo = (ITypeInfo*)NULL; \
   _rgColumns[ulCols].iOrdinal = (ULONG)ordinal; \
   _rgColumns[ulCols].dwFlags = 0; \
   _rgColumns[ulCols].ulColumnSize = (ULONG)colSize; \
   _rgColumns[ulCols].wType = (DBTYPE)type; \
   _rgColumns[ulCols].bPrecision = (BYTE)precision; \
   _rgColumns[ulCols].bScale = (BYTE)scale; \
   _rgColumns[ulCols].cbOffset = offsetof(dataClass, member);

#define ADD_COLUMN_ENTRY_EX(ulCols, name, ordinal, colSize, type, precision, scale, guid, d
ataClass, member, flags) \
   _rgColumns[ulCols].pwszName = (LPOLESTR)name; \
   _rgColumns[ulCols].pTypeInfo = (ITypeInfo*)NULL; \
   _rgColumns[ulCols].iOrdinal = (ULONG)ordinal; \
   _rgColumns[ulCols].dwFlags = flags; \
   _rgColumns[ulCols].ulColumnSize = (ULONG)colSize; \
   _rgColumns[ulCols].wType = (DBTYPE)type; \
   _rgColumns[ulCols].bPrecision = (BYTE)precision; \
   _rgColumns[ulCols].bScale = (BYTE)scale; \
   _rgColumns[ulCols].cbOffset = offsetof(dataClass, member); \
   memset(&(_rgColumns[ulCols].columnid), 0, sizeof(DBID)); \
   _rgColumns[ulCols].columnid.uName.pwszName = (LPOLESTR)name;
```

To test the code in the consumer, you need to make a few changes to the `OnRun` handler. The first change to the function is that you add code to add a property to the property set. The code sets the **DBPROP_IRowsetLocate** property to true, thus telling the provider that you want the bookmark column. The `OnRun` handler code should appear as follows:

```
//////////////////////////////////////////////////////////////////////
// TestProv Consumer Application in TestProvDlg.cpp

void CTestProvDlg::OnRun()
{
```

```
   CCommand<CAccessor<CProvider> > table;
   CDataSource source;
   CSession   session;

   if (source.Open("MyProvider.MyProvider.1", NULL, NULL, NULL,
          NULL) != S_OK)
      return;

   if (session.Open(source) != S_OK)
      return;

   CDBPropSet propset(DBPROPSET_ROWSET);
   propset.AddProperty(DBPROP_IRowsetLocate, true);
   if (table.Open(session, _T("c:\\public\\testprf2\\myData.txt"),
          &propset) != S_OK)
      return;

   CBookmark<4> tempBookmark;
   ULONG ulCount=0;
   while (table.MoveNext() == S_OK)
   {

      DBCOMPARE compare;
      if (ulCount == 2)
         tempBookmark = table.bookmark;
      HRESULT hr = table.Compare(table.dwBookmark, table.dwBookmark,
                 &compare);
      if (FAILED(hr))
         ATLTRACE(_T("Compare failed: 0x%X\n"), hr);
      else
         _ASSERTE(compare == DBCOMPARE_EQ);

      m_ctlString1.AddString(table.szField1);
      m_ctlString2.AddString(table.szField2);
      ulCount++;
   }

   table.MoveToBookmark(tempBookmark);
   m_ctlString1.AddString(table.szField1);
   m_ctlString2.AddString(table.szField2);
}
```

The while loop contains code to call the `Compare` method in the **IRowsetLocate** interface. The code you have should always pass because you are comparing the exact same bookmarks. Also, store one bookmark in a temporary variable so that you can use it after the while loop finishes to call the **MoveToBookmark** function in the consumer templates. The **MoveToBookmark** function calls the **GetRowsAt** method in **IRowsetLocate**.

You also need to update the user record in the consumer. Add an entry in the class to handle a bookmark and an entry in the **COLUMN_MAP**:

```
////////////////////////////////////////////////////////////////////
// TestProvDlg.cpp

class CProvider
{
// Attributes
public:
   CBookmark<4>    bookmark;  // Add this line
   char   szCommand[16];
   char   szText[256];

   // Binding Maps
BEGIN_ACCESSOR_MAP(CProvider, 1)
   BEGIN_ACCESSOR(0, true)   // auto accessor
      BOOKMARK_ENTRY(bookmark)  // Add this line
      COLUMN_ENTRY(1, szField1)
      COLUMN_ENTRY(2, szField2)
```

```
    END_ACCESSOR()
END_ACCESSOR_MAP()
};
```

When you have updated the code, you should be able to build and execute the provider with the **IRowsetLocate** interface.

See Also

Concepts

Advanced Provider Techniques

Passing OLE DB Conformance Tests

To make providers more consistent, the Data Access SDK provides a set of OLE DB conformance tests. The tests check all aspects of your provider and give you reasonable assurance that your provider functions as expected. You can find the OLE DB conformance tests on the Microsoft Data Access SDK. This section focuses on things you should do to pass the conformance tests. For information about running the OLE DB conformance tests, see the SDK.

Running the Conformance Tests

In Visual C++ 6.0, the OLE DB provider templates added a number of hooking functions to allow you to check values and properties. Most of these functions were added in response to the conformance tests.

Note
You need to add several validation functions for your provider to pass the OLE DB conformance tests.

This provider requires two validation routines. The first routine, **CRowsetImpl::ValidateCommandID**, is part of your rowset class. It is called during the creation of the rowset by the provider templates. The sample uses this routine to tell consumers that it does not support indexes. The first call is to **CRowsetImpl::ValidateCommandID** (note that the provider uses the **_RowsetBaseClass** typedef added in the interface map for `CMyProviderRowset` in Provider Support for Bookmarks, so you do not have to type that long line of template arguments). Next, return **DB_E_NOINDEX** if the index parameter is not **NULL** (this indicates the consumer wants to use an index on us). For more information about command IDs, see the OLE DB specification and look for **IOpenRowset::OpenRowset**.

The following code is the **ValidateCommandID** validation routine:

```
/////////////////////////////////////////////////////////////////////
// MyProviderRS.H
// Class: CMyProviderRowset

HRESULT ValidateCommandID(DBID* pTableID, DBID* pIndexID)
{
   HRESULT hr = _RowsetBaseClass::ValidateCommandID(pTableID, pIndexID);
   if (hr != S_OK)
      return hr;

   if (pIndexID != NULL)
      return DB_E_NOINDEX;    // Doesn't support indexes

   return S_OK;
}
```

The provider templates call the **OnPropertyChanged** method whenever someone changes a property on the **DBPROPSET_ROWSET** group. If you want to handle properties for other groups, you add them to the appropriate object (that is, **DBPROPSET_SESSION** checks go in the `CMyProviderSession` class).

The code first checks to see whether the property is linked to another. If the property is being chained, it sets the **DBPROP_BOOKMARKS** property to True. Appendix C of the OLE DB specification contains information about properties. This information also tells you whether the property is chained to another one.

You might also want to add the `IsValidValue` routine to your code. The templates call `IsValidValue` when attempting to set a property. You would override this method if you require additional processing when setting a property value. You can have one of these methods for each property set.

Threading Issues

By default, the OLE DB Provider Wizard in the ATL OLE DB Provider Wizard generates code for the provider to run in an apartment model. If you attempt to run this code with the conformance tests, you initially get failures. This is because Ltm.exe, the tool used to run the OLE DB conformance tests, defaults to free threaded. The OLE DB Provider Wizard code defaults to apartment model for performance and ease of use.

To correct this problem, you can either change LTM or change the provider.

To change LTM to run in apartment threaded mode

1. On the LTM main menu, click **Tools**, and then click **Options**.
2. On the **General** tab, change the threading model from **Free Threaded** to **Apartment Threaded**.

To change your provider to run in free threaded mode:

- In your provider project, search for all instances of **CComSingleThreadModel** and replace it with **CComMultiThreadModel**, which should be in your data source, session, and rowset headers.
- In your .rgs file, change the threading model from **Apartment** to **Both**.
- Follow correct programming rules for free threaded programming (that is, lock on writes).

See Also

Concepts

Advanced Provider Techniques

OLE DB Resource Pooling and Services

To work well with OLE DB pooling, or with any OLE DB service, your provider must support aggregation of all objects. This is a requirement of any OLE DB 1.5 or later provider. It is critical for leveraging services. Providers that do not support aggregation cannot be pooled and no additional services are provided.

To be pooled, providers must support the free thread model. The resource pool determines the provider's thread model according to the **DBPROP_THREADMODEL** property.

If the provider has a global connection state that might change while the data source is in an initialized state, it should support the new **DBPROP_RESETDATASOURCE** property. This property is called before a connection is reused and gives the provider the opportunity to clean up state before its next use. If the provider cannot clean up some state associated with the connection, it can return **DBPROPSTATUS_NOTSETTABLE** for the property and the connection will not be reused.

Providers that connect to a remote database and can detect whether that connection might be lost should support the **DBPROP_CONNECTIONSTATUS** property. This property gives the OLE DB services the ability to detect dead connections and make sure they are not returned to the pool.

Finally, automatic transaction enlistment generally does not work unless it is implemented at the same level that pooling occurs. Providers that support automatic transaction enlistment themselves should support disabling this enlistment by exposing the **DBPROP_INIT_OLEDBSERVICES** property and disabling enlistment if the **DBPROPVAL_OS_TXNENLISTMENT** is deselected.

See Also

Concepts

Advanced Provider Techniques

Resource Pooling in Your OLE DB Application

To leverage pooling in your application, you must make sure OLE DB services are invoked by obtaining your data source through **IDataInitialize** or **IDBPromptInitialize**. If you directly use **CoCreateInstance** to invoke the provider based on the provider's CLSID, no OLE DB services are invoked.

The OLE DB services maintain pools of connected data sources as long as a reference to **IDataInitialize** or **IDBPromptInitialize** is held or as long as there is a connection in use. Pools are also maintained automatically within a COM+ 1.0 Services or Internet Information Services (IIS) environment. If your application takes advantage of pooling outside of a COM+ 1.0 Services or IIS environment, you should keep a reference to **IDataInitialize** or **IDBPromptInitialize** or hold onto at least one connection. To make sure that the pool does not get destroyed when the last connection is released by the application, keep the reference or hold on to the connection for the lifetime of your application.

OLE DB services identify the pool from which the connection is drawn at the time that **Initialize** is called. After the connection is drawn from a pool, it cannot be moved to a different pool. Therefore, avoid doing things in your application that change initialization information, such as calling **UnInitialize** or calling **QueryInterface** for a provider-specific interface before calling **Initialize**. Also, connections established with a prompt value other than **DBPROMPT_NOPROMPT** are not pooled. However, the initialization string retrieved from a connection established through prompting can be used to establish additional pooled connections to the same data source.

Some providers must make a separate connection for each session. These additional connections must be separately enlisted in the distributed transaction, if one exists. OLE DB services caches and reuses a single session per data source, but if the application requests more than one session at a time from a single data source, the provider might end up making additional connections and doing additional transaction enlistments that are not pooled. It is actually more efficient to create a separate data source for each session in a pooled environment than to create multiple sessions from a single data source.

Finally, because ADO automatically makes use of OLE DB services, you can use ADO to establish connections and the pooling and enlistment happen automatically.

See Also

Concepts

OLE DB Resource Pooling and Services

Enabling and Disabling OLE DB Services

The OLE DB Service Component Manager compares the properties specified by the consumer to those supported by the provider to determine whether individual service components could be invoked to satisfy extended functionality requested by the consumer. For example, if an application requests a scrollable cursor and the provider only supports a forward-only cursor, the Service Component Manager invokes the Client Cursor Engine service component to provide scrollable functionality. If the application is relying on extended functionality supported by default on the provider's rowset, and the application does not explicitly set the properties to request that functionality, the functionality might not appear on the rowset returned by the Client Cursor Engine. To be interoperable, applications should always set properties to explicitly request extended functionality where needed.

In some cases, it might be necessary to disable individual OLE DB services to work well with existing applications that make assumptions about the characteristics of a provider. OLE DB services provide the ability to disable individual services, or all services, either on a connection-by-connection basis or for all applications using a single provider.

See Also

Concepts

OLE DB Resource Pooling and Services

Enabling and Disabling Services for a Provider

Individual OLE DB services can be enabled or disabled by default for all applications that access a single provider. This is done by adding an **OLEDB_SERVICES** registry entry under the provider's CLSID, with a **DWORD** value specifying the services to enable or disable, as shown in the following table.

Default services enabled	Keyword value
All services (default)	0xffffffff
All except Pooling and AutoEnlistment	0xfffffffe
All except Client Cursor	0xfffffffb
All except Pooling, AutoEnlistment, and Client Cursor	0xfffffff0
No services	0x00000000
No aggregation, all services disabled	<missing key>

See Also

Concepts

Enabling and Disabling OLE DB Services

Overriding Provider Service Defaults

The provider's registry value for **OLEDB_SERVICES** is returned as the default value for the DBPROP_INIT_OLEDBSERVICES initialization property on the data source object.

As long as the registry entry exists, the provider's objects is aggregated and the user can override the provider's default setting for enabled services by setting the **DBPROP_INIT_OLEDBSERVICES** property before initialization. To enable or disable a particular service, the user generally gets the current value of the **DBPROP_INIT_OLEDBSERVICES** property, sets or clears the bit for the particular property to be enabled or disabled, and resets the property. **DBPROP_INIT_OLEDBSERVICES** can be set directly in OLE DB or in the connection string passed to ADO or **IDataInitialize::GetDatasource**. The corresponding values to enable/disable individual services are listed in the following table.

Default services enabled	**DBPROP_INIT_OLEDBSERVICES property value**	**Value in connection string**
All services (default)	**DBPROPVAL_OS_ENABLEALL**	"OLE DB Services = -1;"
All except Pooling and AutoEnlistment	**DBPROPVAL_OS_ENABLEALL &** **~DBPROPVAL_OS_RESOURCEPOOLING &** **~DBPROPVAL_OS_TXNENLISTMENT**	"OLE DB Services = -4;"
All except Client Cursor	**DBPROPVAL_OS_ENABLEALL** & ~**DBPROPVAL_OS_CLIENTCURSOR**	"OLE DB Services = -5;"
All except Pooling, AutoEnlistment, and Client Cursor	**DBPROPVAL_OS_ENABLEALL &** **~DBPROPVAL_OS_TXNENLISTMENT &** **~DBPROPVAL_OS_CLIENTCURSOR**	"OLE DB Services = -7;"
No services	~**DBPROPVAL_OS_ENABLEALL**	"OLE DB Services = 0;"

If the registry entry does not exist for the provider, the Component Managers will not aggregate the provider's objects, and no services will be invoked, even if explicitly requested by the user.

See Also

Reference

Resource Pooling
How Consumers Use Resource Pooling
How Providers Work Effectively with Resource Pooling

Concepts

Enabling and Disabling OLE DB Services

Open Database Connectivity (ODBC)

The Microsoft Foundation Classes (MFC) library supplies classes for programming with Open Database Connectivity (ODBC).

For more information about using the ODBC API without MFC, see the Open Database Connectivity (ODBC) SDK in MSDN.

Frequently Asked Questions About MFC Database Support

- What data sources can I access with DAO and ODBC?
- Can I call DAO or ODBC APIs directly while using the classes?
- What ODBC drivers are provided?
- What ODBC drivers are installed by default?

In This Section

- ODBC— the Open Database Connectivity standard and API
- What MFC Support Is Supplied for ODBC programming
- What ODBC Drivers Are Available
- Installing MFC ODBC Support
- Should I Use DAO or ODBC?
- Connecting to ODBC Data Sources
- Record Field Exchange (RFX) between a database and a recordset object
- ODBC Recordsets
- Structured Query Language (SQL) for ODBC
- Database Transactions
- Determining the Schema of an ODBC Data Source at run time
- Programmatically Configuring an ODBC Data Source
- Programmatically Creating a Table in an ODBC Data Source
- Fetching Records in Bulk
- Customizing my Recordset's SQL Statement
- Making Direct SQL Calls
- Making Direct Calls to the ODBC API
- How do the database classes work with MFC's document/view architecture?
- ODBC Driver List

ODBC Database Tasks

These tasks are based on the MFC ODBC classes. For more information about using the ODBC API without MFC, see the Open Database Connectivity (ODBC) SDK.

- Install and Get Started with ODBC
- Design and Create an ODBC Database Application
- Use Database Forms (Record Views) with ODBC
- Use ODBC to Work with Other Users
- Work with ODBC Database Connections

- Work with ODBC and Drivers
- Use the ODBC Cursor Library
- Use MFC ODBC Recordsets

See Also

Concepts

Data Access Programming

Installing and Getting Started with ODBC

The following topics help you install the components that you need to start working with Open Database Connectivity (ODBC).

In This Section

- Install MFC ODBC database support
- Install the ODBC drivers that ship with Visual C++
- Redistribute ODBC components to your customers

See Also

Concepts

Open Database Connectivity (ODBC)

Design and Create an ODBC Database Application

These topics help you design and set up your ODBC database application.

In This Section

- Find Microsoft Knowledge Base Articles on ODBC
- ODBC and MFC
- Use MFC ODBC Recordsets
- Learn About ODBC
- Design and Create an ODBC Database Application

See Also

Concepts

Open Database Connectivity (ODBC)

Use Database Forms (Record Views) with ODBC

Some applications, such as data entry or data viewing, require a form, a view with dialog-style controls.

In This Section

- Display database data in a form
- Features of record view classes
- Data exchange for record views
- Your role in working with a record view
- Designing and creating a record view
- Using a record view

See Also

Concepts

Open Database Connectivity (ODBC)

Use ODBC to Work with Other Users

Use ODBC in environments where multiple users are working with the same ODBC database.

In This Section

See Also

Concepts

Open Database Connectivity (ODBC)

Work with ODBC Database Connections

The following topics explain how to connect to ODBC data sources.

In This Section

- Manage ODBC database connections
- Programmatically configure an ODBC data source

See Also

Concepts

Open Database Connectivity (ODBC)

Work with ODBC and Drivers

ODBC allows you to write applications that are not dependent on just one database management system (DBMS). You can write one application that can operate on multiple DBMSs for which your users have the correct DBMS-specific ODBC driver.

In This Section

- Determine the schema of an ODBC data source at run time
- Get a driver description using ODBC Administrator
- Obtain online help for your ODBC driver
- Programmatically create a table in an ODBC data source
- Call ODBC API functions directly in MFC

See Also

Concepts

Open Database Connectivity (ODBC)

Use the ODBC Cursor Library

ODBC manages scrolling through recordsets through the Cursor Library. These topics explain how to work with the Cursor Library.

In This Section

- Use the ODBC Cursor Library

See Also

Concepts

Open Database Connectivity (ODBC)

Use MFC ODBC Recordsets

These topics explain how to work with ODBC recordsets through MFC.

In This Section

General Recordset Tasks

- Choose an ODBC recordset type
- Requery a recordset
- Declare a recordset class for a table
- Declare a recordset class for a predefined query

Recordset Design and Implementation

- Learn about record field exchange (RFX)
- Exchange data between a database and a recordset object
- Customize recordset code produced by the wizards
- Use the record field exchange (RFX) functions

Recordset Performance and Convenience

- Add records in bulk with a recordset
- Filter records in a recordset

Recordset Operations

- Add records in bulk with a recordset
- Requery a recordset
- Filter records in a recordset
- Sort records in a recordset
- Parameterize a recordset
- Pass parameter values to a recordset at run time
- Perform a join with a recordset
- Lock records in a recordset
- Work with large data items in a recordset
- Obtain SUMs and other aggregate results in a recordset

Recordset Selection and SQL

- Select records from an ODBC data source
- Customize the SQL string for a recordset

Dynamic Binding of Recordsets

- Bind data columns in a recordset dynamically at run time

See Also

Concepts

Open Database Connectivity (ODBC)

ODBC Basics

This topic provides the basics of Open Database Connectivity (ODBC):

- How ODBC Works with the Database Classes
- How ODBC Drivers Work with Dynasets
- What ODBC Components You Need to Redistribute with Your Applications

You will also want to read the related topic ODBC: The ODBC Cursor Library.

Note
ODBC data sources are accessible through the MFC ODBC classes, as described in this topic, or through the MFC Data Access Object (DAO) classes.

Note
The MFC ODBC classes support Unicode and multithreading. For more information about the multithreading support, see ODBC Classes and Threads

ODBC is a call-level interface that allows applications to access data in any database for which there is an ODBC driver. Using ODBC, you can create database applications with access to any database for which your end user has an ODBC driver. ODBC provides an API that allows your application to be independent of the source database management system (DBMS).

ODBC is the database portion of the Microsoft Windows Open Services Architecture (WOSA), which is an interface that allows Windows-based desktop applications to connect to multiple computing environments without rewriting the application for each platform.

The following are components of ODBC:

- ODBC API

  A library of function calls, a set of error codes, and a standard Structured Query Language (SQL) syntax for accessing data on DBMSs.

- ODBC Driver Manager

  A dynamic-link library (Odbc32.dll) that loads ODBC database drivers on behalf of an application. This DLL is transparent to your application.

- ODBC database drivers

  One or more DLLs that process ODBC function calls for specific DBMSs. For a list of supplied drivers, see ODBC Driver List.

- ODBC Cursor Library

  A dynamic-link library (Odbccr32.dll) that resides between the ODBC Driver Manager and the drivers and handles scrolling through the data.

- ODBC Administrator

  A tool used for configuring a DBMS to make it available as a data source for an application.

An application achieves independence from DBMSs by working through an ODBC driver written specifically for a DBMS rather than working directly with the DBMS. The driver translates the calls into commands its DBMS can use, simplifying the developer's work and making it available for a wide range of data sources.

The database classes support any data source for which you have an ODBC driver. This might, for example, include a relational database, an Indexed Sequential Access Method (ISAM) database, a Microsoft Excel spreadsheet, or a text file. The ODBC drivers manage the connections to the data source, and SQL is used to select records from the database.

For a list of ODBC drivers included in this version of Visual C++ and for information about obtaining additional drivers, see ODBC Driver List.

See Also

Concepts

Open Database Connectivity (ODBC)

ODBC and the Database Classes

The MFC ODBC database classes encapsulate the ODBC API function calls you would normally make yourself in the member functions of the CDatabase and CRecordset classes. For example, the complex ODBC call sequences, binding of returned records to storage locations, handling of error conditions, and other operations are managed for you by the database classes. As a result, you use a considerably simpler class interface to manipulate records through a recordset object.

Note
ODBC data sources are accessible through the MFC ODBC classes, as described in this TOPIC, or through the MFC Data Acces s Object (DAO) classes.

Although the database classes encapsulate ODBC functionality, they do not provide a one-to-one mapping of ODBC API functions. The database classes provide a higher level of abstraction, modeled after data-access objects found in Microsoft Access and Microsoft Visual Basic. For more information, see What Is the MFC Database Programming Model?.

See Also

Concepts

ODBC Basics

ODBC Driver Requirements for Dynasets

In the MFC ODBC database classes, dynasets are recordsets with dynamic properties; they remain synchronized with the data source in certain ways. MFC dynasets (but not forward-only recordsets) require an ODBC driver with Level 2 API conformance. If the driver for your data source conforms to the Level 1 API set, you can still use both updateable and read-only snapshots and forward-only recordsets, but not dynasets. However, a Level 1 driver can support dynasets if it supports extended fetch and keyset-driven cursors.

In ODBC terminology, dynasets and snapshots are referred to as cursors. A cursor is a mechanism used for keeping track of its position in a recordset. For more information about driver requirements for dynasets, see Dynaset. For more information about cursors, see the Open Database Connectivity (ODBC) SDK in the MSDN documentation.

Note
For updateable recordsets, your ODBC driver must support either positioned update statements or the **::SQLSetPos** ODBC API function. If both are supported, MFC uses **::SQLSetPos** for efficiency. Alternatively, for snapshots, you can use the cursor library, which provides the required support for updateable snapshots (static cursors and positioned update statements).

See Also

Concepts

ODBC Basics

Redistributing ODBC Components to Your Customers

If you incorporate the functionality of the ODBC Administrator programs into your application, you must also distribute to your users the files that run these programs. These ODBC files reside in the \OS\System directory of the Visual C++ CD-ROM. The Redistrb.wri file and the license agreement in the same directory contain a list of ODBC files that you can redistribute.

Consult the documentation for any ODBC drivers you plan to ship. You need to determine which DLLs and other files to ship.

You should also read Installing Database Support for information about ODBC components and drivers and read Redistributing Controls, which explains how to redistribute ActiveX controls.

In addition, you need to include one other file in most cases. The Odbccr32.dll is the ODBC Cursor Library. This library gives Level 1 drivers the capability of forward and backward scrolling. It also provides the capability of supporting snapshots. For more information about the ODBC Cursor Library, see ODBC: The ODBC Cursor Library.

The following topics provide more information about using ODBC with the database classes:

- ODBC: The ODBC Cursor Library
- ODBC: Configuring an ODBC Data Source
- ODBC: Calling ODBC API Functions Directly

See Also

Concepts

ODBC Basics
ODBC Administrator

ODBC: The ODBC Cursor Library

This topic describes the ODBC Cursor Library and explains how to use it. For more information, see:

- Cursor Library and Level 1 ODBC Drivers
- Positioned Updates and Timestamp Columns
- Using the Cursor Library

The ODBC Cursor Library is a dynamic-link library (DLL) that resides between the ODBC Driver Manager and the driver. In ODBC terms, a driver maintains a cursor to keep track of its position in the recordset. The cursor marks the position in the recordset to which you have already scrolled — the current record.

Cursor Library and Level 1 ODBC Drivers

The ODBC Cursor Library gives Level 1 drivers the following new capabilities:

- Forward and backward scrolling. Level 2 drivers do not need the cursor library because they are already scrollable.
- Support for snapshots. The cursor library manages a buffer containing the snapshot's records. This buffer reflects your program's deletions and edits to records but not the additions, deletions, or edits of other users. Therefore, the snapshot is only as current as the cursor library's buffer. The buffer also does not reflect your own additions until you call **Requery**. Dynasets do not use the cursor library.

The cursor library gives you snapshots (static cursors) even if they are not normally supported by your driver. If your driver already supports static cursors, you do not need to load the cursor library to get snapshot support. If you do use the cursor library, you can use only snapshots and forward-only recordsets. If your driver supports dynasets (KEYSET_DRIVEN cursors) and you want to use them, you must not use the cursor library. If you want to use both snapshots and dynasets, you must base them on two different **CDatabase** objects (two different connections) unless your driver supports both.

Positioned Updates and Timestamp Columns

Note
ODBC data sources are accessible through the MFC ODBC classes, as described in this topic, or through the MFC Data Access Object (DAO) classes.

Note
If your ODBC driver supports **SQLSetPos**, which MFC uses if available, this topic does not apply to you.

Most Level 1 drivers do not support positioned updates. Such drivers rely on the cursor library to emulate the capabilities of Level 2 drivers in this regard. The cursor library emulates positioned update support by doing a searched update on the unchanging fields.

In some cases, a recordset might contain a timestamp column as one of those unchanging fields. Two issues arise in using MFC recordsets with tables that contain timestamp columns.

The first issue concerns updateable snapshots on tables with timestamp columns. If the table to which your snapshot is bound contains a timestamp column, you should call **Requery** after you call **Edit** and **Update**. If not, you might not be able to edit the same record again. When you call **Edit** and then **Update**, the record is written to the data source and the timestamp column is updated. If you do not call **Requery**, the timestamp value for the record in your snapshot no longer matches the corresponding timestamp on the data source. When you try to update the record again, the data source might disallow the update because of the mismatch.

The second issue concerns limitations of class CTime when used with the **RFX_Date** function to transfer time and date information to or from a table. Processing the **CTime** object imposes some overhead in the form of extra intermediate processing during the data transfer. The date range of **CTime** objects might also be too limiting for some applications. A new version of the **RFX_Date** function takes an ODBC **TIMESTAMP_STRUCT** parameter instead of a **CTime** object. For more information, see **RFX_Date** in Macros and Globals in the *MFC Reference*.

Using the Cursor Library

When you connect to a data source — by calling CDatabase::OpenEx or CDatabase::Open — you can specify whether to use the cursor library for the data source. If you will be creating snapshots on that data source, specify the **CDatabase::useCursorLib** option in the *dwOptions* parameter to **OpenEx** or specify **TRUE** for the **bUseCursorLib** parameter to **Open** (the default value is **TRUE**). If your ODBC driver supports dynasets and you want to open dynasets on the data source, do not use the cursor library (it masks some driver functionality needed for dynasets). In that case, do not specify **CDatabase::useCursorLib** in **OpenEx** or specify **FALSE** for the **bUseCursorLib** parameter in **Open**.

See Also

Concepts

ODBC Basics

ODBC: Configuring an ODBC Data Source

To use a data source with an application you have developed, you must use ODBC Administrator to configure it. ODBC Administrator keeps track of available data sources and their connection information in the Windows registry. Use ODBC Administrator to add, modify, and delete data sources in the **Data Sources** dialog box and to add and delete ODBC drivers.

Note
This information applies when you use MFC Data Access Object (DAO) classes for ODBC access and when you use MFC ODBC classes.

ODBC Administrator is automatically installed with the Microsoft Foundation Classes (MFC) library database support. For more information about the ODBC Administrator program, see ODBC Administrator and the online ODBC API Reference Help system.

For information about how to write ODBC Setup and Administration programs for MFC database applications, Technical Note 48.

See Also

Concepts

ODBC Basics

ODBC: Calling ODBC API Functions Directly

ODBC: Calling ODBC API Functions Directly

The database classes provide a simpler interface to a data source than does ODBC. As a result, the classes do not encapsulate all the ODBC API. For any functionality that falls outside the abilities of the classes, you must call ODBC API functions directly. For example, you must call the ODBC catalog functions (**::SQLColumns**, **::SQLProcedures**, **::SQLTables**, and others) directly.

Note
ODBC data sources are accessible through the MFC ODBC classes, as described in this topic, or through the MFC Data Access Object (DAO) classes.

To call an ODBC API function directly, you must take the same steps you would take if you were making the calls without the framework. They steps are:

- Allocate storage for any results the call returns.
- Pass an ODBC **HDBC** or **HSTMT** handle, depending on the parameter signature of the function. Use the AFXGetHENV macro to retrieve the ODBC handle.

  Member variables **CDatabase::m_hdbc** and **CRecordset::m_hstmt** are available so that you do not need to allocate and initialize these yourself.
- Perhaps call additional ODBC functions to prepare for or follow up the main call.
- Deallocate storage when you finish.

For more information about these steps, see the Open Database Connectivity (ODBC) SDK in the MSDN documentation.

In addition to these steps, you need to take extra steps to check function return values, ensure that your program is not waiting for an asynchronous call to finish, and so on. You can simplify these last steps by using the **AFX_SQL_ASYNC** and **AFX_SQL_SYNC** macros. For more information, see Macros and Globals in the *MFC Reference*.

See Also

Concepts

ODBC Basics

ODBC and MFC

Note
To use the MFC database classes for targeting a Win32 platform (such as Windows NT), you must have the appropriate ODBC driver for your data source. Some drivers are included with Visual C++; others can be obtained from Microsoft and other vendors. For more information, see ODBC Driver List.

This topic introduces the main concepts of the Microsoft Foundation Classes (MFC) library's ODBC-based database classes and provides an overview of how the classes work together. For more information about ODBC and MFC, see the following topics:

- Connecting to a Data Source
- Selecting and Manipulating Records
- Displaying and Manipulating Data in a Form
- Working with Documents and views
- Access to ODBC and SQL
- Further Reading About the MFC ODBC Classes

The MFC database classes based on ODBC are designed to provide access to any database for which an ODBC driver is available. Because the classes use ODBC, your application can access data in many different data formats and different local/remote configurations. You do not have to write special-case code to handle different database management systems (DBMSs). As long as your users have an appropriate ODBC driver for the data they want to access, they can use your program to manipulate data in tables stored there.

See Also

Concepts

Open Database Connectivity (ODBC)

Connecting to a Data Source

An ODBC data source is a specific set of data, the information required to access that data, and the location of the data source, which can be described using a data-source name. From your program's point of view, the data source includes the data, the DBMS, the network (if any), and ODBC.

To access data provided by a data source, your program must first establish a connection to the data source. All data access is managed through that connection.

Data-source connections are encapsulated by class CDatabase. When a **CDatabase** object is connected to a data source, you can:

- Construct recordsets, which select records from tables or queries.
- Manage transactions, batching updates so all are committed to the data source at once (or the whole transaction is rolled back so the data source is unchanged) — if the data source supports the required level of transactions.
- Directly execute Structured Query Language (SQL) statements.

When you finish working with a data-source connection, you close the **CDatabase** object and either destroy it or reuse it for a new connection. For more information about data-source connections, see Data Source (ODBC).

See Also

Concepts

ODBC and MFC

Selecting and Manipulating Records

Normally when you select records from a data source using a SQL **SELECT** statement, you get a result set, which is a set of records from a table or a query. With the database classes, you use a recordset object to select and access the result set. This is an object of an application-specific class that you derive from class CRecordset. When you define a recordset class, you specify the data source to associate it with, the table to use, and the columns of the table. The MFC Application Wizard or **Add Class** (as described in Adding an MFC ODBC Consumer) creates a class with a connection to a specific data source. The wizards write the GetDefaultSQL member function of class **CRecordset** to return the table name. For more information about using the wizards to create recordset classes, see Database Support, MFC Application Wizard and Adding an MFC ODBC Consumer.

Using a CRecordset object at run time, you can:

- Examine the data fields of the current record.
- Filter or sort the recordset.
- Customize the default SQL **SELECT** statement.
- Scroll through the selected records.
- Add, update, or delete records (if both the data source and the recordset are updateable).
- Test whether the recordset allows requerying and refresh the recordset's contents.

When you finish using the recordset object, you close and destroy it. For more information about recordsets, see Recordset (ODBC).

See Also

Concepts

ODBC and MFC

Displaying and Manipulating Data in a Form

Many data-access applications select data and display it in fields in a form. The database class CRecordView gives you a CFormView object directly connected to a recordset object. The record view uses dialog data exchange (DDX) to move the values of the fields of the current record from the recordset to the controls on the form and to move updated information back to the recordset. The recordset, in turn, uses record field exchange (RFX) to move data between its field data members and the corresponding columns in a table on the data source.

You can use the MFC Application Wizard or **Add Class** (as described in Adding an MFC ODBC Consumer) to create the view class and its associated recordset class in conjunction.

The record view and its recordset are destroyed when you close the document. For more information about record views, see Record Views. For more information about RFX, see Record Field Exchange (RFX).

See Also

Concepts

ODBC and MFC

Working with Documents and Views

The Microsoft Foundation Classes (MFC) library relies on a document/view architecture for many of its features. Typically, a document stores your data and a view displays it within the client area of a frame window and manages user interaction with the data. The view communicates with the document to obtain and update the data. You can use the database classes with the framework or without it.

For more information about using database classes in the framework, see MFC: Using Database Classes with Documents and Views.

By default, the MFC Application Wizard creates a skeleton application with no database support. However, you can select options to include minimal database support or more complete form-based support. For more information about application wizard options, see Database Support, MFC Application Wizard.

You can also use the database classes without using the full document/view architecture. For more information, see MFC: Using Database Classes Without Documents and Views.

See Also

Concepts

ODBC and MFC

Access to ODBC and SQL

The Microsoft Foundation Class Library encapsulates many Windows API calls and still lets you call any Windows API function directly. The database classes give you the same flexibility with regard to the ODBC API. While the database classes shield you from much of the complexity of ODBC, you can call ODBC API functions directly from anywhere in your program.

Similarly, the database classes shield you from having to work much with SQL, but you can use SQL directly if you want. You can customize recordset objects by passing a custom SQL statement (or setting portions of the default statement) when you open the recordset. You can also make SQL calls directly using the ExecuteSQL member function of class CDatabase.

For more information, see ODBC: Calling ODBC API Functions Directly and SQL: Making Direct SQL Calls (ODBC).

See Also

Concepts

ODBC and MFC

Further Reading About the MFC ODBC Classes

The following topics further explain the concepts and techniques introduced in this topic:

- Database Support, MFC Application Wizard
- Adding an MFC ODBC Consumer
- Data Source (ODBC)
- Dynaset
- Exceptions: Database Exceptions
- MFC: Using Database Classes with Documents and Views
- MFC: Using Database Classes Without Documents and Views
- ODBC
- ODBC Administrator
- Record Field Exchange (RFX)
- Recordset (ODBC)
- Record Views
- Serialization: Serialization vs. Database Input/Output
- Snapshot
- SQL
- Transaction (ODBC)

A good place to start your reading is with Recordset (ODBC).

In the *MFC Reference*, see CDatabase, CRecordset, CRecordView, CFieldExchange, and CDBException.

See Also

Concepts

ODBC and MFC
Data Access Programming

ODBC Classes and Threads

Beginning with MFC 4.2, there is multithreading support for the MFC ODBC classes. Note, however, that MFC does not provide multithreading support for the DAO classes.

The multithreading support for the ODBC classes has some limitations. Because these classes wrap the ODBC API, they are restricted to the multithreading support of the components on which they are built. For example, many ODBC drivers are not thread-safe; therefore, the MFC ODBC classes are not thread-safe if you use them with one of these drivers. You should verify whether your particular driver is thread-safe.

When creating a multithreaded application, you should be very careful in using multiple threads to manipulate the same object. For example, using the same **CRecordset** object in two threads might cause problems when retrieving data; a fetch operation in one thread might overwrite the data fetched in the other thread. A more common use of the MFC ODBC classes in separate threads is to share an open **CDatabase** object across threads to use the same ODBC connection, with a separate **CRecordset** object in each thread. Note that you should not pass an unopened **CDatabase** object to a **CRecordset** object in another thread.

Note
If you must have multiple threads manipulate the same object, you should implement the appropriate synchronization mechanisms, such as critical sections. Be aware that certain operations, such as **Open**, are not protected. You should be sure that these operations will not be called concurrently from separate threads.

For more information about creating multithreaded applications, see Multithreading Topics.

See Also

Concepts

Open Database Connectivity (ODBC)
Data Access Programming

ODBC Administrator

ODBC Administrator registers and configures the data sources available to you either locally or across a network. The wizards use information supplied by ODBC Administrator to create code in your applications that connects your users to data sources.

To set up an ODBC data source for use with either the MFC ODBC classes or the MFC Data Access Object (DAO) classes, you must first register and configure the data source. Use ODBC Administrator to add and remove data sources. Depending on the ODBC driver, you can also create new data sources.

ODBC Administrator is installed during setup. If you chose **Custom** Installation and did not select any ODBC drivers in the **Database Options** dialog box, you need to run Setup again to install the necessary files.

During setup, you select the ODBC drivers you want to install. You can later install additional drivers that ship with Visual C++ using the Visual C++ Setup program.

If you want to install ODBC drivers that do not ship with Visual C++, you must run the Setup program that accompanies the driver.

To install ODBC drivers that ship with Visual C++

1. Run Setup from your Visual C++ distribution CD.

   The opening dialog box in the Setup program is appears.

2. Click **Next** on each dialog box until you reach the **Installation Options** dialog box. Select **Custom**, and then click **Next**.

3. Clear all the check boxes in the **Microsoft Visual C++ Setup** dialog box except the **Database Options** check box, and then click **Details** to display the **Database Options** dialog box.

4. Clear the **Microsoft Data Access Objects** check box, select the **Microsoft ODBC Drivers** check box, and then click **Details**.

   The **Microsoft ODBC Drivers** dialog box appears.

5. Select the drivers you want to install, and then click **OK** twice.

6. Click **Next** on the remaining dialog boxes to begin the installation. Setup notifies you when the installation is complete.

When the drivers are installed, you can configure the data source using the ODBC Administrator. You will find the ODBC icon in Control Panel.

See Also

Concepts

Open Database Connectivity (ODBC)
Data Source (ODBC)

ODBC Driver List

Visual C++ provides ODBC drivers for the following databases:

- SQL Server
- Microsoft Access
- Microsoft Excel
- dBASE
- Paradox
- Microsoft Oracle ODBC
- Text files

For information about ODBC drivers available from Microsoft and other companies, including the ODBC Driver Pack, contact Microsoft Product Support Services.

See Also

Concepts

Open Database Connectivity (ODBC)

Data Source (ODBC)

This topic applies to the MFC ODBC classes.

In database terms, a data source is a specific set of data, the information required to access that data, and the location of the data source, which can be described using a data-source name. To work with class CDatabase, the data source must be one that you have configured through Open Database Connectivity (ODBC) Administrator. Examples of data sources include a remote database running on Microsoft SQL Server across a network or a Microsoft Access file in a local directory. From your application, you can access any data source for which you have an ODBC driver.

You can have one or more data sources active in your application at one time, each represented by a **CDatabase** object. You can also have multiple simultaneous connections to any data source. You can connect to remote as well as to local data sources, depending on the drivers you have installed and the capabilities of your ODBC drivers. For more information about data sources and ODBC Administrator, see ODBC and ODBC Administrator.

The following topics explain more about data sources:

- Data Source: Managing Connections (ODBC)
- Data Source: Determining the Schema of the Data Source (ODBC)

See Also

Concepts

Open Database Connectivity (ODBC)

Data Source: Managing Connections (ODBC)

This topic applies to the MFC ODBC classes.

This topic explains:

- How to configure a data source.
- How a multiuser environment affects a data source and its recordsets.
- Why you generalize a connection string to a data source.
- How to connect to a data source.
- How to disconnect from a data source.
- How to reuse a CDatabase object.

Connecting to a data source means establishing communications with a DBMS to access the data. When you connect to a data source from an application through an ODBC driver, the driver makes the connection for you, either locally or across a network.

You can connect to any data source for which you have an ODBC driver. Users of your application must also have the same ODBC driver for their data source. For more information about redistributing ODBC drivers, see Redistributing ODBC Components to Your Customers.

Configuring a Data Source

ODBC Administrator is used to configure your data sources. You can also use ODBC Administrator after installation to add or remove data sources. When you create applications, you can either direct your users to the ODBC Administrator to let them add data sources or you can build this functionality into your application by making direct ODBC installation calls. For more information, see ODBC Administrator.

You can use an Excel file as a data source, and you need to configure the file so that it is registered and appears in the **Select Data Source** dialog box.

To use an Excel file as a data source

1. Configure the file with the ODBC Data Source Administrator.
2. On the **File DSN** tab, click **Add**.
3. In the **Create New Data Source** dialog box, select an Excel driver, and then click **Next**.
4. Click **Browse**, and select the name of the file to be used as a date source.

Note
You might need to select **All Files** in the drop-down menu to view the .xls files.

1. Click **Next**, and then click **Finish**.
2. In the **ODBC Microsoft Excel Setup** dialog box, select the database Version and Workbook.

Working in a Multiuser Environment

If multiple users are connected to a data source, they can change data while you are manipulating it in your recordsets. Similarly, your changes might affect other users' recordsets. For more information, see Recordset: How Recordsets Update Records (ODBC) and Transaction (ODBC).

Generalizing the Connection String

The wizards use a default connection string to establish a connection to a data source. You use this connection to view tables and columns while you are developing your application. However, this default connection string might not be appropriate for your users' connections to the data source through your application. For example, their data source and the path to its location might be different from the one used in developing your application. In that case, you should reimplement the

CRecordset::GetDefaultConnect member function in a more generic fashion and discard the wizard implementation. For example, use one of the following approaches:

- Register and manage the connection strings using ODBC Administrator.
- Edit the connection string and remove the data source name. The framework supplies ODBC as the data source; at run time, ODBC displays a dialog box asking for the data source name and any other required connection information.
- Supply the data source name only. ODBC asks for the user ID and password, if required. For example, before generalizing, the connection string looks like this:

  ```
  CString CApp1Set::GetDefaultConnect()
  {
     return "ODBC;DSN=afx;Trusted_Connection=Yes;";
  }
  ```

  This connection string specifies a trusted connection, which uses Windows NT integrated security. You should avoid hard-coding a password or specifying a blank password, because doing so creates a major security weakness. Instead, you can give **GetDefaultConnect** a new connection string so that it queries for a user ID and password.

  ```
  // User must select data source and supply user ID and password:
      return "ODBC;";
  // User ID and password required:
      return "ODBC;DSN=mydb;";
  // Password required (myuserid must be replaced with a valid user ID):
      return "ODBC;DSN=mydb;UID=myuserid;";
  // Hard-coded user ID and password (SECURITY WEAKNESS--AVOID):
      return "ODBC;DSN=mydb;UID=sa;PWD=777;";
  ```

Connecting to a Specific Data Source

To connect to a specific data source, your data source must already have been configured with ODBC Administrator.

To connect to a specific data source

1. Construct a **CDatabase** object.
2. Call its **OpenEx** or **Open** member function.

For more information about how to specify the data source if it is something other than the one you specified with a wizard, see CDatabase::OpenEx or CDatabase::Open in the *MFC Reference*.

Disconnecting from a Data Source

You must close any open recordsets before calling the **Close** member function of **CDatabase**. In recordsets associated with the **CDatabase** object you want to close, any pending **AddNew** or **Edit** statements are canceled and all pending transactions are rolled back.

To disconnect from a data source

1. Call the **CDatabase** object's Close member function.
2. Destroy the object unless you want to reuse it.

Reusing a CDatabase Object

You can reuse a **CDatabase** object after disconnecting from it, whether you use it to reconnect to the same data source or to connect to a different data source.

To reuse a CDatabase object

1. Close the object's original connection.
2. Instead of destroying the object, call its **OpenEx** or **Open** member function again.

See Also

Reference

CRecordset Class

Concepts

Data Source (ODBC)

Data Source: Determining the Schema of the Data Source (ODBC)

Data Source: Determining the Schema of the Data Source (ODBC)

This topic applies to the MFC ODBC classes.

To set up data members in your **CRecordset** objects, you need to know the schema of the data source to which you are connecting. Determining the schema of a data source involves obtaining a list of the tables in the data source, a list of the columns in each table, the data type of each column, and the existence of any indexes.

See Also

Concepts

Data Source (ODBC)
Data Source: Managing Connections (ODBC)

Data Source: Programmatically Configuring an ODBC Data Source

This topic explains how you can configure Open Database Connectivity (ODBC) data source names programmatically. This gives you flexibility to access data without forcing the user to explicitly use the ODBC Administrator or other programs to specify the names of data sources.

Typically, a user runs ODBC Administrator to create a data source if the associated database management system (DBMS) supports this operation.

When creating a Microsoft Access ODBC data source through ODBC Administrator, you are given two choices: you can select an existing .mdb file or you can create a new .mdb file. There is no programmatic way of creating the .mdb file from your MFC ODBC application. Therefore, if your application requires that you place data into a Microsoft Access data source (.mdb file), you most likely want to have an empty .mdb file that you can use or copy whenever you need it.

However, many DBMSs allow programmatic data source creation. Some data sources maintain a directory specification for databases. That is, a directory is the data source and each table within the data source is stored in a separate file (in the case of dBASE, each table is a .dbf file). Drivers for other ODBC databases, such as Microsoft Access and SQL Server, require that some specific criteria be satisfied before a data source can be established. For example, when using the SQL Server ODBC driver, you need to have established a SQL Server computer.

SQLConfigDataSource Example

The following example uses the **::SQLConfigDataSource** ODBC API function to create a new Excel data source called New Excel Data Source:

```
SQLConfigDataSource(NULL,ODBC_ADD_DSN, "Excel Files (*.xls)",
                    "DSN=New Excel Data Source\0"
                    "Description=New Excel Data Source\0"
                    "FileType=Excel\0"
                    "DataDirectory=C:\\EXCELDIR\0"
                    "MaxScanRows=20\0");
```

Note that the data source is actually a directory (C:\EXCELDIR); this directory must exist. The Excel driver uses directories as its data sources and files as the individual tables (one table per .xls file).

For more information about creating tables, see Data Source: Programmatically Creating a Table in an ODBC Data Source.

The following information discusses the parameters that need to be passed to the **::SQLConfigDataSource** ODBC API function. To use **::SQLConfigDataSource**, you must include the Odbcinst.h header file and use the Odbcinst.lib import library. Also, Odbccp32.dll must be in the path at run time (or Odbcinst.dll for 16 bit).

You can create an ODBC data source name using ODBC Administrator or a similar utility. However, sometimes it is desirable to create a data source name directly from your application to obtain access without requiring the user to run a separate utility.

ODBC Administrator (typically installed in Control Panel) creates a new data source by putting entries in the Windows registry (or, for 16 bit, in the Odbc.ini file). The ODBC Driver Manager queries this file to obtain the required information about the data source. It is important to know what information needs to be placed in the registry because you need to supply it with the call to **::SQLConfigDataSource**.

Although this information could be written directly to the registry without using **::SQLConfigDataSource**, any application that does so is relying on the current technique that the Driver Manager uses to maintain its data. If a later revision to the ODBC Driver Manager implements record keeping about data sources in a different way, any application that uses this technique is broken. It is generally advisable to use an API function when one is provided. For example, your code is portable from 16 bit to 32 bit if you use the **::SQLConfigDataSource** function, because the function correctly writes to the Odbc.ini file or to the registry.

SQLConfigDataSource Parameters

The following explains the parameters of the **::SQLConfigDataSource** function. Much of the information is taken from the ODBC API *Programmer's Reference* supplied with Visual C++ version 1.5 and later.

Function Prototype

```
BOOL SQLConfigDataSource(HWND hwndParent,UINT fRequest, LPCSTR lpszDriver, LPCSTR lpszAttri
butes);
```

Remarks
Parameters and Usage
hwndParent

The window specified as the owner of any dialog boxes that either the ODBC Driver Manager or the specific ODBC driver creates to obtain additional information from the user about the new data source. If the *lpszAttributes* parameter does not supply enough information, a dialog box appears. The *hwndParent* parameter might be **NULL**.

lpszDriver

The driver description. This is the name presented to users rather than the physical driver name (the DLL).

lpszAttributes

List of attributes in the form "keyname=value". These strings are separated by null terminators with two consecutive null terminators at the end of the list. These attributes are primarily default driver-specific entries, which go into the registry for the new data source. One important key that is not mentioned in the ODBC API reference for this function is "DSN" ("data source name"), which specifies the name of the new data source. The rest of the entries are specific to the driver for the new data source. Often it is not necessary to supply all of the entries because the driver can prompt the user with dialog boxes for the new values. (Set *hwndParent* to **NULL** to cause this.) You might want to explicitly supply default values so that the user is not prompted.

To determine the description of a driver for the lpszDriver parameter using ODBC Administrator

1. Run ODBC Administrator.
2. Click **Add**.

This gives you a list of installed drivers and their descriptions. Use this description as the *lpszDriver* parameter. Note that you use the entire description, such as "Excel Files (*.xls)", including the file name extension and parentheses if they exist in the description.

As an alternative, you can examine the registry (or, for 16 bit, the file Odbcinst.ini), which contains a list of all driver entries and descriptions under the registry key "ODBC Drivers" (or the section [ODBC Drivers] in Odbcinst.ini).

One way to find the keynames and values for the *lpszAttributes* parameter is to examine the Odbc.ini file for an already configured data source (perhaps one that has been configured by ODBC Administrator).

To find keynames and values for the lpszAttributes parameter

1. Run the Windows registry editor (or, for 16 bit, open the Odbc.ini file).
2. Find the ODBC data sources information using one of the following:
   - For 32 bit, find the key **HKEY_CURRENT_USER\Software\ODBC\ODBC.INI\ODBC Data Sources** in the left pane.

     The right pane lists entries of the form: "pub: REG_SZ:<*data source name*>", where <*data source name*> is a data source that has already been configured with the desired settings for the driver you intend to use. Select the data source you want, for example, SQL Server. The items following the string "pub:" are, in order, the keyname and value to use in your *lpszAttributes* parameter.
   - For 16 bit, find the section in the Odbc.ini file marked by [<*data source name*>].

     The lines following this line are of the form "keyname=value". These are exactly the entries to use in your *lpszAttributes* parameter.

You might also want to examine the documentation for the specific driver you are going to use. You can find useful information in the online Help for the driver, which you can access by running ODBC Administrator. These Help files are usually placed in the WINDOWS\SYSTEM directory for Windows NT, Windows 3.1, or Windows 95.

To obtain online Help for your ODBC driver

1. Run ODBC Administrator.

2. Click **Add**.
3. Select the driver name.
4. Click **OK**.

When ODBC Administrator displays the information for creating a new data source for that particular driver, click **Help**. This opens the Help file for that particular driver, which generally contains important information concerning the use of the driver.

For more information, see the Installer DLL Function Reference.

See Also

Concepts

Data Source (ODBC)

Data Source: Programmatically Creating a Table in an ODBC Data Source

This topic explains how to create a table for your data source, using the **ExecuteSQL** member function of class **CDatabase**, passing the function a string that contains a **CREATE TABLE** SQL statement.

For general information about ODBC data sources in MFC, see Data Source (ODBC). The topic Data Source: Programmatically Configuring an ODBC Data Source describes creating data sources.

When you have the data source established, you can easily create tables using the **ExecuteSQL** member function and the **CREATE TABLE** SQL statement. For example, if you had a **CDatabase** object called `myDB`, you could use the following MFC code to create a table:

```
myDB.ExecuteSQL("CREATE TABLE OFFICES (OfficeID TEXT(4)" ",
                        OfficeName TEXT(10))");
```

This code example creates a table called "OFFICES" in the Microsoft Access data source connection maintained by `myDB`; the table contains two fields "OfficeID" and "OfficeName." For more information about creating tables as well as primary keys and indexes for them, see Appendix C in the ODBC Programmer's Reference.

Note

The field types specified in the **CREATE TABLE** SQL statement might vary according to the ODBC driver that you are using. The Microsoft Query program (distributed with Visual C++ 1.5) is one way to discover what field types are available for a data source. In Microsoft Query, click **File**, click **Table_Definition**, select a table from a data source, and look at the type shown in the **Type** combo box. Appendix C in the ODBC Programmer's Reference describes the supported SQL syntax. SQL syntax also exists to create indexes.

See Also

Concepts

Data Source (ODBC)

Dynaset

This topic describes dynasets and discusses their availability.

Note
This topic applies to the MFC ODBC classes, including CRecordset. For information about dynasets in the DAO classes, see CDaoRecordset. With DAO, you can open dynaset-type recordsets.

A dynaset is a recordset with dynamic properties. During its lifetime, a recordset object in dynaset mode (usually called a dynaset) stays synchronized with the data source in the following way. In a multiuser environment, other users might edit or delete records that are in your dynaset or add records to the table your dynaset represents. Records your application adds to or deletes from the recordset are reflected in your dynaset. Records that other users add to the table will not be reflected in your dynaset until you rebuild the dynaset by calling its **Requery** member function. When other users delete records, MFC code skips over the deletions in your recordset. Other users' editing changes to existing records are reflected in your dynaset as soon as you scroll to the affected record.

Similarly, edits you make to records in a dynaset are reflected in dynasets in use by other users. Records you add are not reflected in other users' dynasets until they requery their dynasets. Records you delete are marked as "deleted" in other users' recordsets. If you have multiple connections to the same database (multiple **CDatabase** objects), recordsets associated with those connections have the same status as the recordsets of other users.

Dynasets are most valuable when data must be dynamic, as (for example) in an airline reservation system.

Note
To use dynasets, you must have an ODBC driver for your data source that supports dynasets and the ODBC cursor library mu st not be loaded. For more information, see Availability of Dynasets.

To specify that a recordset is a dynaset, pass **CRecordset::dynaset** as the first parameter to the **Open** member function of your recordset object.

Note
For updatable dynasets, your ODBC driver must support either positioned update statements or the **::SQLSetPos** ODBC API f unction. If both are supported, MFC uses **::SQLSetPos** for efficiency.

Availability of Dynasets

The MFC database classes support dynasets if the following requirements are met:

- The ODBC cursor library DLL must not be in use for this data source.

  If the cursor library is used, it masks some functionality of the underlying ODBC driver that is necessary for dynaset support. If you want to use dynasets (and your ODBC driver has the functionality required for dynasets, as described in the rest of this section), you can cause MFC not to load the cursor library when you create a **CDatabase** object. For more information, see ODBC and the OpenEx or Open member function of class **CDatabase**.

  In ODBC terminology, dynasets and snapshots are referred to as cursors. A cursor is a mechanism used for keeping track of its position in a recordset. For more information about cursors, see ODBC Programmer's Reference.

- The ODBC driver for your data source must support keyset-driven cursors.

  Keyset-driven cursors manage data from a table by getting and storing a set of keys. The keys are used to obtain current data from the table when the user scrolls onto a particular record. To determine whether your driver provides this support, call the **::SQLGetInfo** ODBC API function with the **SQL_SCROLL_OPTIONS** parameter.

  If you try to open a dynaset without keyset support, you get a **CDBException** with the return code value **AFX_SQL_ERROR_DYNASET_NOT_SUPPORTED**.

- The ODBC driver for your data source must support extended fetching.

  Extended fetching is the ability to scroll backward as well as forward over the resulting records of your SQL query. To

determine whether your driver supports this ability, call the **::SQLGetFunctions** ODBC API function with the **SQL_API_SQLEXTENDEDFETCH** parameter.

If you want updateable dynasets (or snapshots, for that matter), your ODBC driver must also support either the **::SQLSetPos** ODBC API function or positioned updates. The **::SQLSetPos** function allows MFC to update the data source without sending SQL statements. If this support is available, MFC uses it in preference to making updates using SQL. To determine whether your driver supports **::SQLSetPos**, call **::SQLGetInfo** with the **SQL_POS_OPERATIONS** parameter.

Positioned updates use SQL syntax (of the form **WHERE CURRENT OF** <cursorname>) to identify a particular row in the table on the data source. To determine whether your driver supports positioned updates, call **::SQLGetInfo** with the **SQL_POSITIONED_STATEMENTS** parameter.

Generally, MFC dynasets (but not forward-only recordsets) require an ODBC driver with level 2 API conformance. If the driver for your data source conforms to the level 1 API set, you can still use both updateable and read-only snapshots and forward-only recordsets, but not dynasets. However, a level 1 driver can support dynasets if it supports extended fetching and keyset-driven cursors. For more information about ODBC conformance levels, see ODBC.

Note
If you want to use both snapshots and dynasets, you must base them on two different **CDatabase** objects (two different connections).

Unlike snapshots, which use intermediate storage maintained by the ODBC cursor library, dynasets fetch a record directly from the data source as soon as you scroll to it. This keeps the records originally selected by the dynaset synchronized with the data source.

For a list of ODBC drivers included in this version of Visual C++ and for information about obtaining additional drivers, see ODBC Driver List.

See Also

Concepts

Open Database Connectivity (ODBC)

Snapshot

A snapshot is a recordset that reflects a static view of the data as it existed at the time the snapshot was created. When you open the snapshot and move to all the records, the set of records it contains and their values do not change until you rebuild the snapshot by calling **Requery**.

Note
This topic applies to the MFC ODBC classes. If you are using the MFC DAO classes instead of the MFC ODBC classes, see CDaoRecordset::Open for a description of snapshot-type recordsets.

You can create updateable or read-only snapshots with the database classes. Unlike a dynaset, an updateable snapshot does not reflect changes to record values made by other users, but it does reflect updates and deletions made by your program. Records added to a snapshot do not become visible to the snapshot until you call **Requery**.

Tip
A snapshot is an ODBC static cursor. Static cursors do not actually get a row of data until you scroll to that record. To ensure that all records are immediately retrieved, you can scroll to the end of your recordset and then scroll to the first record you want to see. Note, however, that scrolling to the end entails extra overhead and can lower performance.

Snapshots are most valuable when you need the data to remain fixed during your operations, as when you are generating a report or performing calculations. Even so, the data source can diverge considerably from your snapshot, so you might want to rebuild it from time to time.

Snapshot support is based on the ODBC Cursor Library, which provides static cursors and positioned updates (needed for updateability) for any Level 1 driver. The cursor library DLL must be loaded in memory for this support. When you construct a **CDatabase** object and call its **OpenEx** member function, you must specify the **CDatabase::useCursorLib** option of the *dwOptions* parameter. If you call the **Open** member function, the cursor library is loaded by default. If you are using dynasets instead of snapshots, you do not want to cause the cursor library to be loaded.

Snapshots are available only if the ODBC Cursor Library was loaded when the **CDatabase** object was constructed or the ODBC driver you are using supports static cursors.

Note
For some ODBC drivers, snapshots (static cursors) might not be updateable. Check your driver documentation for cursor types supported and the concurrency types they support. To guarantee updateable snapshots, make sure you load the cursor library into memory when you create a **CDatabase** object. For more information, see ODBC: The ODBC Cursor Library.

Note
If you want to use both snapshots and dynasets, you must base them on two different **CDatabase** objects (two different connections).

For more information about the properties snapshots share with all recordsets, see Recordset (ODBC). For more information about ODBC and snapshots, including the ODBC Cursor Library, see ODBC.

See Also

Concepts

Open Database Connectivity (ODBC)

Record Field Exchange (RFX)

The MFC ODBC database classes automate moving data between the data source and a recordset object. When you derive a class from CRecordset and do not use bulk row fetching, data is transferred by the record field exchange (RFX) mechanism.

Note
If you have implemented bulk row fetching in a derived **CRecordset** class, the framework uses the bulk record field exchang e (Bulk RFX) mechanism to transfer data. For more information, see Recordset: Fetching Records in Bulk (ODBC).

RFX is similar to dialog data exchange (DDX). Moving data between a data source and the field data members of a recordset requires multiple calls to the recordset's DoFieldExchange function and considerable interaction between the framework and ODBC. The RFX mechanism is type-safe and saves you the work of calling ODBC functions such as **::SQLBindCol**. For more information about DDX, see Dialog Data Exchange and Validation.

RFX is mostly transparent to you. If you declare your recordset classes with the MFC Application Wizard or **Add Class** (as described in Adding an MFC ODBC Consumer), RFX is built into them automatically. Your recordset class must be derived from the base class **CRecordset** supplied by the framework. The MFC Application Wizard lets you create an initial recordset class. **Add Class** lets you add other recordset classes as you need them. For more information and examples, see Adding an MFC ODBC Consumer.

You must manually add a small amount of RFX code in three cases, when you want to:

- Use parameterized queries. For more information, see Recordset: Parameterizing a Recordset (ODBC).
- Perform joins (using one recordset for columns from two or more tables). For more information, see Recordset: Performing a Join (ODBC).
- Bind data columns dynamically. This is less common than parameterization. For more information, see Recordset: Dynamically Binding Data Columns (ODBC).

If you need a more advanced understanding of RFX, see Record Field Exchange: How RFX Works.

The following topics explain the details of using recordset objects:

- Record Field Exchange: Using RFX
- Record Field Exchange: Using the RFX Functions
- Record Field Exchange: How RFX Works

See Also

Tasks

Adding an MFC ODBC Consumer

Reference

Database Support, MFC Application Wizard

CRecordset Class

Concepts

Open Database Connectivity (ODBC)

Recordset (ODBC)

Record Field Exchange: Using RFX

This topic explains what you do to use RFX in relation to what the framework does.

Note
This topic applies to classes derived from CRecordset in which bulk row fetching has not been implemented. If you are using bulk row fetching, bulk record field exchange (Bulk RFX) is implemented. Bulk RFX is similar to RFX. To understand the differences, see Recordset: Fetching Records in Bulk (ODBC).

The following topics contain related information:

- Record Field Exchange: Working with the Wizard Code introduces the main components of RFX and explains the code that the MFC Application Wizard and **Add Class** (as described in Adding an MFC ODBC Consumer) write to support RFX and how you might want to modify the wizard code.
- Record Field Exchange: Using the RFX Functions explains writing calls to the RFX functions in your **DoFieldExchange** override.

The following table shows your role in relation to what the framework does for you.

Using RFX: You and the Framework

You	The framework
Declare your recordset classes with a wizard. Specify names and data types of field data members.	The wizard derives a **CRecordset** class and writes a DoFieldExchange override for you, including an RFX function call for each field data member.
(Optional) Manually add any needed parameter data members to the class. Manually add an RFX function call to **DoFieldExchange** for each parameter data member, add a call to CFieldExchange::SetFieldType for the group of parameters, and specify the total number of parameters in m_nParams. See Recordset: Parameterizing a Recordset (ODBC).	
(Optional) Manually bind additional columns to field data members. Manually increment m_nFields. See Recordset: Dynamically Binding Data Columns (ODBC).	
Construct an object of your recordset class. Before using the object, set the values of its parameter data members, if any.	For efficiency, the framework prebinds the parameters, using ODBC. When you pass parameter values, the framework passes them to the data source. Only the parameter values are sent for requeries, unless the sort and/or filter strings have changed.
Open a recordset object using CRecordset::Open.	Executes the recordset's query, binds columns to field data members of the recordset, and calls **DoFieldExchange** to exchange data between the first selected record and the recordset's field data members.
Scroll in the recordset using CRecordset::Move or a menu or toolbar command.	Calls **DoFieldExchange** to transfer data to the field data members from the new current record.
Add, update, and delete records.	Calls **DoFieldExchange** to transfer data to the data source.

See Also

Reference

CRecordset Class
CFieldExchange Class

Macros, Global Functions, and Global Variables

Concepts

Record Field Exchange (RFX)

Record Field Exchange: How RFX Works

Recordset: Obtaining SUMs and Other Aggregate Results (ODBC)

Record Field Exchange: Working with the Wizard Code

This topic explains the code that the MFC Application Wizard and **Add Class** (as described in Adding an MFC ODBC Consumer) write to support RFX and how you might want to alter that code.

Note
This topic applies to classes derived from **CRecordset** in which bulk row fetching has not been implemented. If you are using bulk row fetching, bulk record field exchange (Bulk RFX) is implemented. Bulk RFX is similar to RFX. To understand the differences, see Recordset: Fetching Records in Bulk (ODBC).

When you create a recordset class with the MFC Application Wizard or **Add Class**, the wizard writes the following RFX-related elements for you, based on the data source, table, and column choices you make in the wizard:

- Declarations of the recordset field data members in the recordset class
- An override of **CRecordset::DoFieldExchange**
- Initialization of recordset field data members in the recordset class constructor

Field Data Member Declarations

The wizards write a recordset class declaration in an .h file that resembles the following for class `CSections`:

```
class CSections : public CRecordset
{
public:
   CSections(CDatabase* pDatabase = NULL);
   DECLARE_DYNAMIC(CSections)

// Field/Param Data
   CString   m_strCourseID;
   CString   m_strInstructorID;
   CString   m_strRoomNo;
   CString   m_strSchedule;
   CString   m_strSectionNo;

// Overrides
   // Wizard generated virtual function overrides
   protected:
   virtual CString GetDefaultConnect();  // Default connection string
   virtual CString GetDefaultSQL();      // Default SQL for Recordset
   virtual void DoFieldExchange(CFieldExchange* pFX);  // RFX support

// Implementation
#ifdef _DEBUG
   virtual void AssertValid() const;
   virtual void Dump(CDumpContext& dc) const;
#endif

};
```

If you add parameter data members or new field data members that you bind yourself, add them after the wizard-generated ones.

Also, notice that the wizard overrides the **DoFieldExchange** member function of class **CRecordset**.

DoFieldExchange Override

DoFieldExchange is the heart of RFX. The framework calls **DoFieldExchange** any time it needs to move data either from data source to recordset or from recordset to data source. **DoFieldExchange** also supports obtaining information about field data members through the IsFieldDirty and IsFieldNull member functions.

The following **DoFieldExchange** override is for the `CSections` class. The wizard writes the function in the .cpp file for your

recordset class.

```
void CSections::DoFieldExchange(CFieldExchange* pFX)
{
   pFX->SetFieldType(CFieldExchange::outputColumn);
   RFX_Text(pFX, "CourseID", m_strCourseID);
   RFX_Text(pFX, "InstructorID", m_strInstructorID);
   RFX_Text(pFX, "RoomNo", m_strRoomNo);
   RFX_Text(pFX, "Schedule", m_strSchedule);
   RFX_Text(pFX, "SectionNo", m_strSectionNo);
}
```

Notice the following key features of the function:

- This section of the function is called the field map.
- A call to **CFieldExchange::SetFieldType**, through the *pFX* pointer. This call specifies that all RFX function calls up to the end of **DoFieldExchange** or the next call to **SetFieldType** are output columns. For more information, see CFieldExchange::SetFieldType.
- Several calls to the **RFX_Text** global function — one per field data member (all of which are **CString** variables in the example). These calls specify the relationship between a column name on the data source and a field data member. The RFX functions do the actual data transfer. The class library supplies RFX functions for all the common data types. For more information about RFX functions, see Record Field Exchange: Using the RFX Functions.

  Note

  The order of the columns in your result set must match the order of the RFX function calls in `DoFieldExchange`.

- The *pFX* pointer to a CFieldExchange object that the framework passes when it calls **DoFieldExchange**. The **CFieldExchange** object specifies the operation that **DoFieldExchange** is to perform, the direction of transfer, and other context information.

Recordset Constructor

The recordset constructor that the wizards write contains two things related to RFX:

- An initialization for each field data member
- An initialization for the m_nFields data member, which contains the number of field data members

The constructor for the `CSections` recordset example looks like this:

```
CSections::CSections(CDatabase* pdb)
   : CRecordset(pdb)
{
   m_strCourseID = "";
   m_strInstructorID = "";
   m_strRoomNo = "";
   m_strSchedule = "";
   m_strSectionNo = "";
   m_nFields = 5;
}
```

Note

If you add any field data members manually, as you might if you bind new columns dynamically, you must increment **m_nFields**. Do so by appending another line of code, such as:

```
m_nFields += 3;
```

This is the code for adding three new fields. If you add any parameter data members, you must initialize the m_nParams data member, which contains the number of parameter data members. Put the **m_nParams** initialization outside the brackets.

See Also

Concepts

Record Field Exchange (RFX)

Record Field Exchange: Using the RFX Functions

This topic explains how to use the RFX function calls that make up the body of your **DoFieldExchange** override.

Note
This topic applies to classes derived from CRecordset in which bulk row fetching has not been implemented. If you are using bulk row fetching, bulk record field exchange (Bulk RFX) is implemented. Bulk RFX is similar to RFX. To understand the differences, see Recordset: Fetching Records in Bulk (ODBC).

The RFX global functions exchange data between columns on the data source and field data members in your recordset. You write the RFX function calls in your recordset's DoFieldExchange member function. This topic describes the functions briefly and shows the data types for which RFX functions are available. Technical Note 43 describes how to write your own RFX functions for additional data types.

RFX Function Syntax

Each RFX function takes three parameters (and some take an optional fourth or fifth parameter):

- A pointer to a CFieldExchange object. You simply pass along the *pFX* pointer passed to **DoFieldExchange**.
- The name of the column as it appears on the data source.
- The name of the corresponding field data member or parameter data member in the recordset class.
- (Optional) In some of the functions, the maximum length of the string or array being transferred. This defaults to 255 bytes, but you might want to change it. The maximum size is based on the maximum size of a **CString** object — **INT_MAX** (2,147,483,647) bytes — but you will probably encounter driver limits before that size.
- (Optional) In the **RFX_Text** function, you sometimes use a fifth parameter to specify the data type of a column.

For more information, see the RFX functions under Macros and Globals in the *Class Library Reference*. For an example of when you might make special use of the parameters, see Recordset: Obtaining SUMs and Other Aggregate Results (ODBC).

RFX Data Types

The class library supplies RFX functions for transferring many different data types between the data source and your recordsets. The following list summarizes the RFX functions by data type. In cases where you must write your own RFX function calls, select from these functions by data type.

Function	Data type
RFX_Bool	**BOOL**
RFX_Byte	**BYTE**
RFX_Binary	**CByteArray**
RFX_Double	**double**
RFX_Single	**float**
RFX_Int	**int**
RFX_Long	**long**
RFX_LongBinary	**CLongBinary**
RFX_Text	**CString**

RFX_Date	**CTime**

For more information, see the RFX function documentation under Macros and Globals in the *Class Library Reference*. For information about how C++ data types map to SQL data types, see the table ANSI SQL Data Types Mapped to C++ Data Types in SQL: SQL and C++ Data Types (ODBC).

See Also

Reference

CRecordset Class

CFieldExchange Class

Concepts

Record Field Exchange (RFX)

Record Field Exchange: How RFX Works

Recordset: Parameterizing a Recordset (ODBC)

Recordset: Dynamically Binding Data Columns (ODBC)

Record Field Exchange: How RFX Works

This topic explains the RFX process. This is an advanced topic covering:

- RFX and the recordset
- The RFX process

Note
This topic applies to classes derived from **CRecordset** in which bulk row fetching has not been implemented. If you are using bulk row fetching, bulk record field exchange (Bulk RFX) is implemented. Bulk RFX is similar to RFX. To understand the differences, see Recordset: Fetching Records in Bulk (ODBC).

RFX and the Recordset

The recordset object's field data members, taken together, constitute an edit buffer that holds the selected columns of one record. When the recordset is first opened and is about to read the first record, RFX binds (associates) each selected column to the address of the appropriate field data member. When the recordset updates a record, RFX calls ODBC API functions to send a SQL **UPDATE** or **INSERT** statement to the driver. RFX uses its knowledge of the field data members to specify the columns to write.

The framework backs up the edit buffer at certain stages so it can restore its contents if necessary. RFX backs up the edit buffer before adding a new record and before editing an existing record. It restores the edit buffer in some cases, for example, after an **Update** call following **AddNew**. The edit buffer is not restored if you abandon a newly changed edit buffer by, for example, moving to another record before calling **Update**.

Besides exchanging data between the data source and the recordset's field data members, RFX manages binding parameters. When the recordset is opened, any parameter data members are bound in the order of the "?" placeholders in the SQL statement that **CRecordset::Open** constructs. For more information, see Recordset: Parameterizing a Recordset (ODBC).

Your recordset class's override of **DoFieldExchange** does all the work, moving data in both directions. Like dialog data exchange (DDX), RFX needs information about the data members of your class. The wizard provides the necessary information by writing a recordset-specific implementation of **DoFieldExchange** for you, based on the field data member names and data types you specify with the wizard.

Record Field Exchange Process

This section describes the sequence of RFX events as a recordset object is opened and as you add, update, and delete records. The table Sequence of RFX Operations During Recordset Open and the table Sequence of RFX Operations During Scrolling in this topic show the process as RFX processes a **Move** command in the recordset and as RFX manages an update. During these processes, DoFieldExchange is called to perform many different operations. The **m_nOperation** data member of the CFieldExchange object determines which operation is requested. You might find it helpful to read Recordset: How Recordsets Select Records (ODBC) and Recordset: How Recordsets Update Records (ODBC) before you read this material.

RFX: Initial Binding of Columns and Parameters

The following RFX activities occur, in the order shown, when you call a recordset object's Open member function:

- If the recordset has parameter data members, the framework calls **DoFieldExchange** to bind the parameters to parameter placeholders in the recordset's SQL statement string. A data type-dependent representation of the value of the parameter is used for each placeholder found in the **SELECT** statement. This occurs after the SQL statement is prepared but before it is executed. For information about statement preparation, see the **::SQLPrepare** function in the ODBC *Programmer's Reference*.
- The framework calls **DoFieldExchange** a second time to bind the values of selected columns to corresponding field data members in the recordset. This establishes the recordset object as an edit buffer containing the columns of the first record.
- The recordset executes the SQL statement and the data source selects the first record. The record's columns are loaded into the recordset's field data members.

The following table shows the sequence of RFX operations when you open a recordset.

Sequence of RFX Operations During Recordset Open

Your operation	DoFieldExchange operation	Database/SQL operation
1. Open recordset.		
	2. Build a SQL statement.	
		3. Send the SQL.
	4. Bind parameter data members.	
	5. Bind field data members to columns.	
		6. ODBC does the move and fills in the data.
	7. Fix up the data for C++.	

Recordsets use ODBC's prepared execution to allow for fast requerying with the same SQL statement. For more information about prepared execution, see the ODBC SDK *Programmer's Reference* in the MSDN Library.

RFX: Scrolling

When you scroll from one record to another, the framework calls **DoFieldExchange** to replace the values previously stored in the field data members with values for the new record.

The following table shows the sequence of RFX operations when the user moves from record to record.

Sequence of RFX Operations During Scrolling

Your operation	DoFieldExchange operation	Database/SQL operation
1. Call **MoveNext** or one of the other Move functions.		
		2. ODBC does the move and fills in the data.
	3. Fix up the data for C++.	

RFX: Adding New Records and Editing Existing Records

If you add a new record, the recordset operates as an edit buffer to build up the contents of the new record. As with adding records, editing records involves changing the values of the recordset's field data members. From the RFX perspective, the sequence is as follows:

1. Your call to the recordset's AddNew or Edit member function causes RFX to store the current edit buffer so it can be restored later.

2. **AddNew** or **Edit** prepares the fields in the edit buffer so RFX can detect changed field data members.

   Because a new record has no previous values to compare new ones with, **AddNew** sets the value of each field data member to a **PSEUDO_NULL** value. Later, when you call **Update**, RFX compares each data member's value with the **PSEUDO_NULL** value. If there is a difference, the data member has been set. (**PSEUDO_NULL** is not the same as a record column with a true Null value nor is either of these the same as C++ **NULL**.)

   Unlike the **Update** call for **AddNew**, the **Update** call for **Edit** compares updated values with previously stored values rather than using **PSEUDO_NULL**. The difference is that **AddNew** has no previously stored values for comparison.

3. You directly set the values of field data members whose values you want to edit or that you want filled for a new record. This can include calling **SetFieldNull**.

4. Your call to Update checks for changed field data members, as described in step 2 (see the table Sequence of RFX Operations During Scrolling). If none have changed, **Update** returns 0. If some field data members have changed, **Update** prepares and executes a SQL **INSERT** statement that contains values for all updated fields in the

record.

5. For **AddNew**, **Update** concludes by restoring the previously stored values of the record that was current before the **AddNew** call. For **Edit**, the new, edited values remain in place.

The following table shows the sequence of RFX operations when you add a new record or edit an existing record.

Sequence of RFX Operations During AddNew and Edit

Your operation	DoFieldExchange operation	Database/SQL operation
1. Call **AddNew** or **Edit**.		
	2. Back up the edit buffer.	
	3. For **AddNew**, mark field data members as "clean" and Null.	
4. Assign values to recordset field data members.		
5. Call **Update**.		
	6. Check for changed fields.	
	7. Build SQL **INSERT** statement for **AddNew** or **UPDATE** statement for **Edit**.	
		8. Send the SQL.
	9. For **AddNew**, restore the edit buffer to its backed-up contents. For **Edit**, delete the backup.	

RFX: Deleting Existing Records

When you delete a record, RFX sets all the fields to **NULL** as a reminder that the record is deleted and you must move off it. You do not need any other RFX sequence information.

See Also

Tasks

Adding an MFC ODBC Consumer

Reference

Macros, Global Functions, and Global Variables

CFieldExchange Class

CRecordset::DoFieldExchange

Concepts

Record Field Exchange (RFX)

Recordset (ODBC)

This topic applies to the MFC ODBC classes.

A CRecordset object represents a set of records selected from a data source. The records can be from:

- A table.
- A query.
- A stored procedure that accesses one or more tables.

An example of a recordset based on a table is "all customers," which accesses a Customer table. An example of a query is "all invoices for Joe Smith." An example of a recordset based on a stored procedure (sometimes called a predefined query) is "all of the delinquent accounts," which invokes a stored procedure in the back-end database. A recordset can join two or more tables from the same data source, but not tables from different data sources.

Note
For information about deriving recordset classes with the wizards, see Adding an MFC ODBC Consumer and Database Support, MFC Application Wizard.

Note
Some ODBC drivers support views of the database. A view in this sense is a query originally created with the SQL **CREATE VIEW** statement. The wizards currently do not support views, but it is possible to code this support yourself.

Recordset Capabilities

All recordset objects share the following capabilities:

- If the data source is not read-only, you can specify that your recordset be updatable, appendable, or read-only. If the recordset is updateable, you can choose either pessimistic or optimistic locking methods, provided the driver supplies the appropriate locking support. If the data source is read-only, the recordset will be read-only.
- You can call member functions to scroll through the selected records.
- You can filter the records to constrain which records are selected from those available.
- You can sort the records in ascending or descending order, based on one or more columns.
- You can parameterize the recordset to qualify the recordset selection at run time.

Snapshots and Dynasets

There are two principal types of recordsets: snapshots and dynasets. Both are supported by class **CRecordset**. Each shares the common characteristics of all recordsets, but each also extends the common functionality in its own specialized way. Snapshots provide a static view of the data and are useful for reports and other situations in which you want a view of the data as it existed at a particular time. Dynasets are useful when you want updates made by other users to be visible in the recordset without having to requery or refresh the recordset. Snapshots and dynasets can be updateable or read-only. To reflect records added or deleted by other users, call CRecordset::Requery.

CRecordset also allows for two other types of recordsets: dynamic recordsets and forward-only recordsets. Dynamic recordsets are similar to dynasets; however, dynamic recordsets reflect any records added or deleted without calling **CRecordset::Requery**. For this reason, dynamic recordsets are generally expensive with respect to processing time on the DBMS, and many ODBC drivers do not support them. In contrast, forward-only recordsets provide the most efficient method of data access for recordsets that do not require updates or backward scrolling. For example, you might use a forward-only recordset to migrate data from one data source to another, where you only need to move through the data in a forward direction. To use a forward-only recordset, you must do both of the following:

- Pass the option **CRecordset::forwardOnly** as the *nOpenType* parameter of the Open member function.
- Specify **CRecordset::readOnly** in the *dwOptions* parameter of **Open**.

Note

For information about ODBC driver requirements for dynaset support, see ODBC. For a list of ODBC drivers included in this version of Visual C++ and for information about obtaining additional drivers, see ODBC Driver List.

Your Recordsets

For every distinct table, view, or stored procedure that you want to access, you typically define a class derived from **CRecordset**. (The exception is a database join, in which one recordset represents columns from two or more tables.) When you derive a recordset class, you enable the record field exchange (RFX) mechanism or the bulk record field exchange (Bulk RFX) mechanism, which are similar to the dialog data exchange (DDX) mechanism. RFX and Bulk RFX simplify the transfer of data from the data source into your recordset; RFX additionally transfers data from your recordset to the data source. For more information, see Record Field Exchange (RFX) and Recordset: Fetching Records in Bulk (ODBC).

A recordset object gives you access to all the selected records. You scroll through the multiple selected records using **CRecordset** member functions, such as **MoveNext** and **MovePrev**. At the same time, a recordset object represents only one of the selected records, the current record. You can examine the fields of the current record by declaring recordset class member variables that correspond to columns of the table or of the records that result from the database query. For information about recordset data members, see Recordset: Architecture (ODBC).

The following topics explain the details of using recordset objects. The topics are listed in functional categories and a natural browse order to permit sequential reading.

Topics about the mechanics of opening, reading, and closing recordsets

- Recordset: Architecture (ODBC)
- Recordset: Declaring a Class for a Table (ODBC)
- Recordset: Creating and Closing Recordsets (ODBC)
- Recordset: Scrolling (ODBC)
- Recordset: Bookmarks and Absolute Positions (ODBC)
- Recordset: Filtering Records (ODBC)
- Recordset: Sorting Records (ODBC)
- Recordset: Parameterizing a Recordset (ODBC)

Topics about the mechanics of modifying recordsets

- Recordset: Adding, Updating, and Deleting Records (ODBC)
- Recordset: Locking Records (ODBC)
- Recordset: Requerying a Recordset (ODBC)

Topics about somewhat more advanced techniques

- Recordset: Performing a Join (ODBC)
- Recordset: Declaring a Class for a Predefined Query (ODBC)
- Recordset: Dynamically Binding Data Columns (ODBC)
- Recordset: Fetching Records in Bulk (ODBC)
- Recordset: Working with Large Data Items (ODBC)
- Recordset: Obtaining SUMs and Other Aggregate Results (ODBC)

Topics about how recordsets work

- Recordset: How Recordsets Select Records (ODBC)
- Recordset: How Recordsets Update Records (ODBC)

See Also

Tasks

Adding an MFC ODBC Consumer

Concepts

Open Database Connectivity (ODBC)

Transaction (ODBC)

Recordset: Architecture (ODBC)

This topic applies to the MFC ODBC classes.

This topic describes the data members that comprise the architecture of a recordset object:

- Field data members
- Parameter data members
- Using m_nFields and m_nParams data members

Note
This topic applies to objects derived from **CRecordset** in which bulk row fetching has not been implemented. If bulk row fetching is implemented, the architecture is similar. To understand the differences, see Recordset: Fetching Records in Bulk (ODBC).

Sample Class

When you use the MFC ODBC Consumer Wizard from **Add Class** wizard to declare a recordset class derived from **CRecordset**, the resulting class has the general structure shown in the following simple class:

```
class CCourse : public CRecordset
{
public:
   CCourse(CDatabase* pDatabase = NULL);
   ...
   CString m_strCourseID;
   CString m_strCourseTitle;
   CString m_strIDParam;
};
```

At the beginning of the class, the wizard writes a set of field data members. When you create the class, you must specify one or more field data members. If the class is parameterized, as the sample class is (with the data member `m_strIDParam`), you must manually add parameter data members. The wizard does not support adding parameters to a class.

Field Data Members

The most important members of your recordset class are the field data members. For each column you select from the data source, the class contains a data member of the appropriate data type for that column. For example, the sample class shown at the beginning of this topic has two field data members, both of type **CString**, called `m_strCourseID` and `m_strCourseTitle`.

When the recordset selects a set of records, the framework automatically binds the columns of the current record (after the **Open** call, the first record is current) to the field data members of the object. That is, the framework uses the appropriate field data member as a buffer in which to store the contents of a record column.

As the user scrolls to a new record, the framework uses the field data members to represent the current record. The framework refreshes the field data members, replacing the previous record's values. The field data members are also used for updating the current record and for adding new records. As part of the process of updating a record, you specify the update values by assigning values directly to the appropriate field data member or members.

Parameter Data Members

If the class is parameterized, it has one or more parameter data members. A parameterized class lets you base a recordset query on information obtained or calculated at run time.

Typically, the parameter helps narrow the selection, as in the following example. Based on the sample class at the beginning of this topic, the recordset object might execute the following SQL statement:

```
SELECT CourseID, CourseTitle FROM Course WHERE CourseID = ?
```

The "?" is a placeholder for a parameter value that you supply at run time. When you construct the recordset and set its

`m_strIDParam` data member to MATH101, the effective SQL statement for the recordset becomes:

```
SELECT CourseID, CourseTitle FROM Course WHERE CourseID = MATH101
```

By defining parameter data members, you tell the framework about parameters in the SQL string. The framework binds the parameter, which lets ODBC know where to get values to substitute for the placeholder. In the example, the resulting recordset contains only the record from the Course table with a CourseID column whose value is MATH101. All specified columns of this record are selected. You can specify as many parameters (and placeholders) as you need.

Note

MFC does nothing itself with the parameters — in particular, it does not perform a text substitution. Instead, MFC tells ODBC where to get the parameter; ODBC retrieves the data and performs the necessary parameterization.

Note

The order of parameters is important. For information about this and more information about parameters, see Recordset: Parameterizing a Recordset (ODBC).

Using m_nFields and m_nParams

When a wizard writes a constructor for your class, it also initializes the m_nFields data member, which specifies the number of field data members in the class. If you add any parameters to your class, you must also add an initialization for the m_nParams data member, which specifies the number of parameter data members. The framework uses these values to work with the data members.

For more information and examples, see Record Field Exchange: Using RFX.

See Also

Concepts

Recordset (ODBC)
Recordset: Declaring a Class for a Table (ODBC)
Record Field Exchange (RFX)

Recordset: Declaring a Class for a Table (ODBC)

This topic applies to the MFC ODBC classes.

The most common recordset class opens a single table. To declare a recordset class for a single table, use the MFC ODBC Consumer Wizard from **Add Class** and choose each column you want by naming a corresponding recordset field data member.

Other uses for recordsets include:

- Joining two or more tables.
- Containing the results of a predefined query.

See Also

Concepts

Recordset (ODBC)
Recordset: Creating and Closing Recordsets (ODBC)
Recordset: Declaring a Class for a Predefined Query (ODBC)
Recordset: Performing a Join (ODBC)

Recordset: Creating and Closing Recordsets (ODBC)

This topic applies to the MFC ODBC classes.

To use a recordset, construct a recordset object and then call its **Open** member function to run the recordset's query and select records. When you finish with the recordset, close and destroy the object.

This topic explains:

- When and how to create a recordset object.
- When and how you can qualify the recordset's behavior by parameterizing, filtering, sorting, or locking it.
- When and how to close a recordset object.

Creating Recordsets at Run Time

Before you can create recordset objects in your program, you typically write application-specific recordset classes. For more information about this preliminary step, see Adding an MFC ODBC Consumer.

Open a dynaset or snapshot object when you need to select records from a data source. The type of object to create depends on what you need to do with the data in your application and on what your ODBC driver supports. For more information, see Dynaset and Snapshot.

To open a recordset

1. Construct an object of your **CRecordset**-derived class.

   You can construct the object on the heap or on the stack frame of a function.

2. Optionally modify the default recordset behavior. For the available options, see Setting Recordset Options.

3. Call the object's Open member function.

In the constructor, pass a pointer to a **CDatabase** object or pass **NULL** to use a temporary database object that the framework constructs and opens based on the connection string returned by the GetDefaultConnect member function. The **CDatabase** object might already be connected to a data source.

The call to **Open** uses SQL to select records from the data source. The first record selected (if any) is the current record. The values of this record's fields are stored in the recordset object's field data members. If any records were selected, both the **IsBOF** and **IsEOF** member functions return 0.

In your Open call, you can:

- Specify whether the recordset is a dynaset or snapshot. Recordsets open as snapshots by default. Or, you can specify a forward-only recordset, which allows only forward scrolling, one record at a time.

  By default, a recordset uses the default type stored in the **CRecordset** data member **m_nDefaultType**. Wizards write code to initialize **m_nDefaultType** to the recordset type you choose in the wizard. Rather than accepting this default, you can substitute another recordset type.

- Specify a string to replace the default SQL **SELECT** statement that the recordset constructs.

- Specify whether the recordset is read-only or append-only. Recordsets allow full updating by default, but you can limit that to adding new records only or you can disallow all updates.

The following example shows how to open a read-only snapshot object of class `CStudentSet`, an application-specific class:

```
// Construct the snapshot object
CStudentSet rsStudent( NULL );
// Set options if desired, then open the recordset
if(!rsStudent.Open(CRecordset::snapshot, NULL, CRecordset::readOnly))
    return FALSE;
// Use the snapshot to operate on its records...
```

After you call **Open**, use the member functions and data members of the object to work with the records. In some cases, you

might want to requery or refresh the recordset to include changes that have occurred on the data source. For more information, see Recordset: Requerying a Recordset (ODBC).

Tip
The connect string you use during development might not be the same connect string that your eventual users need. For ide as about generalizing your application in this regard, see Data Source: Managing Connections (ODBC).

Setting Recordset Options

After you construct your recordset object but before you call **Open** to select records, you might want to set some options to control the recordset's behavior. For all recordsets, you can:

- Specify a filter to constrain record selection.
- Specify a sort order for the records.
- Specify parameters so you can select records using information obtained or calculated at run time.

You can also set the following option if conditions are right:

- If the recordset is updateable and supports locking options, specify the locking method used for updates.

Note
To affect record selection, you must set these options before you call the **Open** member function.

Closing a Recordset

When you finish with your recordset, you must dispose of it and deallocate its memory.

To close a recordset

1. Call its Close member function.
2. Destroy the recordset object.

   If you declared it on the stack frame of a function, the object is destroyed automatically when the object goes out of scope. Otherwise, use the **delete** operator.

Close frees the recordset's **HSTMT** handle. It does not destroy the C++ object.

See Also

Concepts

Recordset (ODBC)
Recordset: Scrolling (ODBC)
Recordset: Adding, Updating, and Deleting Records (ODBC)

Recordset: Scrolling (ODBC)

This topic applies to the MFC ODBC classes.

After you open a recordset, you need to access the records to display values, do calculations, generate reports, and so on. Scrolling lets you move from record to record within your recordset.

This topic explains:

- How to scroll from one record to another in a recordset.
- Under what circumstances scrolling is and is not supported.

Scrolling from One Record to Another

Class **CRecordset** provides the **Move** member functions for scrolling within a recordset. These functions move the current record by rowsets. If you have implemented bulk row fetching, a **Move** operation repositions the recordset by the size of the rowset. If you have not implemented bulk row fetching, a call to a **Move** function repositions the recordset by one record each time. For more information about bulk row fetching, see Recordset: Fetching Records in Bulk (ODBC).

Note
When moving through a recordset, deleted records might not be skipped. For more information, see the IsDeleted member function.

In addition to the **Move** functions, **CRecordset** provides member functions for checking whether you have scrolled past the end or ahead of the beginning of your recordset.

To determine whether scrolling is possible in your recordset, call the **CanScroll** member function.

To scroll

1. Forward one record or one rowset: call the MoveNext member function.
2. Backward one record or one rowset: call the MovePrev member function.
3. To the first record in the recordset: call the MoveFirst member function.
4. To the last record in the recordset or to the last rowset: call the MoveLast member function.
5. *N* records relative to the current position: call the Move member function.

To test for the end or the beginning of the recordset

1. Have you scrolled past the last record? Call the IsEOF member function.
2. Have you scrolled ahead of the first record (moving backward)? Call the IsBOF member function.

The following code example uses **IsBOF** and **IsEOF** to detect the limits of a recordset when scrolling in either direction.

```
// Open a recordset; first record is current
CCustSet rsCustSet( NULL );
rsCustSet.Open( );

if( rsCustSet.IsBOF( ) )
    return;
    // The recordset is empty

// Scroll to the end of the recordset, past
// the last record, so no record is current
while ( !rsCustSet.IsEOF( ) )
    rsCustSet.MoveNext( );

// Move to the last record
rsCustSet.MoveLast( );
```

```
// Scroll to beginning of the recordset, before
// the first record, so no record is current
while( !rsCustSet.IsBOF( ) )
    rsCustSet.MovePrev( );

// First record is current again
rsCustSet.MoveFirst( );
```

IsEOF returns a nonzero value if the recordset is positioned past the last record. **IsBOF** returns a nonzero value if the recordset is positioned ahead of the first record (before all records). In either case, there is no current record to operate on. If you call **MovePrev** when **IsBOF** is already **TRUE** or call **MoveNext** when **IsEOF** is already **TRUE**, the framework throws a **CDBException**. You can also use **IsBOF** and **IsEOF** to check for an empty recordset.

For more information about recordset navigation, see Recordset: Bookmarks and Absolute Positions (ODBC).

When Scrolling Is Supported

As originally designed, SQL provided only forward scrolling, but ODBC extends scrolling capabilities. The available level of support for scrolling depends on the ODBC drivers your application works with, your driver's ODBC API conformance level, and whether the ODBC Cursor Library is loaded into memory. For more information, see ODBC and ODBC: The ODBC Cursor Library.

Tip

You can control whether the cursor library is used. See the *bUseCursorLib* and *dwOptions* parameters to CDatabase::Open.

Note

Unlike the MFC DAO classes, the MFC ODBC classes do not provide a set of **Find** functions for locating the next (or previous) record that meets specified criteria.

See Also

Reference

CRecordset::CanScroll
CRecordset::CheckRowsetError

Concepts

Recordset (ODBC)
Recordset: Filtering Records (ODBC)

Recordset: Bookmarks and Absolute Positions (ODBC)

This topic applies to the MFC ODBC classes.

When navigating through a recordset, you often need a way of returning to a particular record. A record's bookmark and absolute position provide two such methods.

This topic explains:

- How to use bookmarks.
- How to set the current record using absolute positions.

Bookmarks in MFC ODBC

A bookmark uniquely identifies a record. When you navigate through a recordset, you cannot always rely on the absolute position of a record because records can be deleted from the recordset. The reliable way to keep track of the position of a record is to use its bookmark. Class **CRecordset** supplies member functions for:

- Getting the bookmark of the current record, so you can save it in a variable (GetBookmark).
- Moving quickly to a given record by specifying its bookmark, which you saved earlier in a variable (SetBookmark).

The following example illustrates how to use these member functions to mark the current record and later return to it:

```
// rs is a CRecordset or
// CRecordset-derived object

CDBVariant varRecordToReturnTo;
rs.GetBookmark( varRecordToReturnTo );

// More code in which you
// move to other records

rs.SetBookmark( varRecordToReturnTo );
```

You do not need to extract the underlying data type from the CDBVariant object. Assign the value with **GetBookmark** and return to that bookmark with **SetBookmark**.

Note

Depending on your ODBC driver and recordset type, bookmarks might not be supported. You can easily determine whether bookmarks are supported by calling CRecordset::CanBookmark. Furthermore, if bookmarks are supported, you must explicitly choose to implement them by specifying the **CRecordset::useBookmarks** option in the Open member function. You should also check the persistence of bookmarks after certain recordset operations. For example, if you **Requery** a recordset, bookmarks might no longer be valid. Call CDatabase::GetBookmarkPersistence to check whether you can safely call **SetBookmark**.

Absolute Positions in MFC ODBC

Besides bookmarks, class **CRecordset** allows you to set the current record by specifying an ordinal position. This is called absolute positioning.

Note

Absolute positioning is not available on forward-only recordsets. For more information about forward-only recordsets, see Recordset (ODBC).

To move the current record pointer using absolute position, call CRecordset::SetAbsolutePosition. When you pass a value to **SetAbsolutePosition**, the record corresponding to that ordinal position becomes the current record.

Note

The absolute position of a record is potentially unreliable. If the user deletes records from the recordset, the ordinal position of any subsequent record changes. Bookmarks are the recommended method for moving the current record. For more information, see Bookmarks in MFC ODBC.

For more information about recordset navigation, see Recordset: Scrolling (ODBC).

See Also

Concepts

Recordset (ODBC)

Recordset: Filtering Records (ODBC)

This topic applies to the MFC ODBC classes.

This topic explains how to filter a recordset so that it selects only a particular subset of the available records. For example, you might want to select only the class sections for a particular course, such as MATH101. A filter is a search condition defined by the contents of a SQL **WHERE** clause. When the framework appends it to the recordset's SQL statement, the **WHERE** clause constrains the selection.

You must establish a recordset object's filter after you construct the object but before you call its **Open** member function (or before you call the **Requery** member function for an existing recordset object whose **Open** member function has been called previously).

To specify a filter for a recordset object

1. Construct a new recordset object (or prepare to call **Requery** for an existing object).
2. Set the value of the object's m_strFilter data member.

   The filter is a null-terminated string that contains the contents of the SQL **WHERE** clause but not the keyword **WHERE**. For example, use:

   ```
   m_pSet->m_strFilter = "CourseID = 'MATH101'";
   ```

   not

   ```
   m_pSet->m_strFilter = "WHERE CourseID = 'MATH101'";
   ```

   Note

   The literal string "MATH101" is shown with single quotation marks above. In the ODBC SQL specification, single quotes are used to denote a character string literal. Check your ODBC driver documentation for the quoting requirements of y our DBMS in this situation. This syntax is also discussed further near the end of this topic.

3. Set any other options you need, such as sort order, locking mode, or parameters. Specifying a parameter is especially useful. For information about parameterizing your filter, see Recordset: Parameterizing a Recordset (ODBC).
4. Call **Open** for the new object (or **Requery** for a previously opened object).

Tip

Using parameters in your filter is potentially the most efficient method for retrieving records.

Tip

Recordset filters are useful for joining tables and for using parameters based on information obtained or calculated at run ti me.

The recordset selects only those records that meet the search condition you specified. For example, to specify the course filter described above (assuming a variable `strCourseID` currently set, for instance, to "MATH101"), do the following:

```
// Using the recordset pointed to by m_pSet

// Set the filter
m_pSet->m_strFilter = "CourseID = " + strCourseID;

// Run the query with the filter in place
if ( m_pSet->Open( CRecordset::snapshot, NULL, CRecordset::readOnly ) )

// Use the recordset
```

The recordset contains records for all class sections for MATH101.

Notice how the filter string was set in the example above, using a string variable. This is the typical usage. But suppose you wanted to specify the literal value 100 for the course ID. The following code shows how to set the filter string correctly with a literal value:

```
m_strFilter = "StudentID = '100'";   // correct
```

Note the use of single quote characters; if you set the filter string directly, the filter string is **not**:

```
m_strFilter = "StudentID = 100";   // incorrect for some drivers
```

The quoting shown above conforms to the ODBC specification, but some DBMSs might require other quote characters. For more information, see SQL: Customizing Your Recordset's SQL Statement (ODBC).

Note

If you choose to override the recordset's default SQL string by passing your own SQL string to **Open**, you should not set a filter if your custom string has a **WHERE** clause. For more information about overriding the default SQL, see SQL: Customizing Your Recordset's SQL Statement (ODBC).

See Also

Concepts

Recordset (ODBC)
Recordset: Sorting Records (ODBC)
Recordset: How Recordsets Select Records (ODBC)
Recordset: How Recordsets Update Records (ODBC)
Recordset: Locking Records (ODBC)

Recordset: Sorting Records (ODBC)

This topic applies to the MFC ODBC classes.

This topic explains how to sort your recordset. You can specify one or more columns on which to base the sort, and you can specify ascending or descending order (**ASC** or **DESC**; **ASC** is the default) for each specified column. For example, if you specify two columns, the records are sorted first on the first column named and then on the second column named. A SQL **ORDER BY** clause defines a sort. When the framework appends the **ORDER BY** clause to the recordset's SQL query, the clause controls the selection's ordering.

You must establish a recordset's sort order after you construct the object but before you call its **Open** member function (or before you call the **Requery** member function for an existing recordset object whose **Open** member function has been called previously).

To specify a sort order for a recordset object

1. Construct a new recordset object (or prepare to call **Requery** for an existing one).
2. Set the value of the object's m_strSort data member.

   The sort is a null-terminated string. It contains the contents of the **ORDER BY** clause but not the keyword **ORDER BY**. For example, use:

   ```
   recordset.m_strSort = "LastName DESC, FirstName DESC";
   ```

   not

   ```
   recordset.m_strSort = "ORDER BY LastName DESC, FirstName DESC";
   ```

3. Set any other options you need, such as a filter, locking mode, or parameters.
4. Call **Open** for the new object (or **Requery** for an existing object).

The selected records are ordered as specified. For example, to sort a set of student records in descending order by last name, then first name, do the following:

```
// Construct the recordset
CStudentSet rsStudent( NULL );
// Set the sort
rsStudent.m_strSort = "LastName DESC, FirstName DESC";
// Run the query with the sort in place
rsStudent.Open( );
```

The recordset contains all of the student records, sorted in descending order (Z to A) by last name, then by first name.

Note
If you choose to override the recordset's default SQL string by passing your own SQL string to **Open**, do not set a sort if you r custom string has an **ORDER BY** clause.

See Also

Concepts

Recordset (ODBC)
Recordset: Parameterizing a Recordset (ODBC)
Recordset: Filtering Records (ODBC)

Recordset: Parameterizing a Recordset (ODBC)

This topic applies to the MFC ODBC classes.

Sometimes you might want to be able to select records at run time, using information you have calculated or obtained from your end user. Recordset parameters let you accomplish that goal.

This topic explains:

- The purpose of a parameterized recordset.
- When and why you might want to parameterize a recordset.
- How to declare parameter data members in your recordset class.
- How to pass parameter information to a recordset object at run time.

Parameterized Recordsets

A parameterized recordset lets you pass parameter information at run time. This has two valuable effects:

- It might result in better execution speed.
- It lets you build a query at run time, based on information not available to you at design time, such as information obtained from your user or calculated at run time.

When you call **Open** to run the query, the recordset uses the parameter information to complete its **SQL SELECT** statement. You can parameterize any recordset.

When to Use Parameters

Typical uses for parameters include:

- Passing run-time arguments to a predefined query.

  To pass parameters to a stored procedure, you must specify a complete custom ODBC **CALL** statement — with parameter placeholders — when you call **Open**, overriding the recordset's default SQL statement. For more information, see CRecordset::Open in the *Class Library Reference* and SQL: Customizing Your Recordset's SQL Statement (ODBC) and Recordset: Declaring a Class for a Predefined Query (ODBC).

- Efficiently performing numerous requeries with different parameter information.

  For example, each time your end user looks up information for a particular student in the student registration database, you can specify the student's name or ID as a parameter obtained from the user. Then, when you call your recordset's **Requery** member function, the query selects only that student's record.

  Your recordset's filter string, stored in **m_strFilter**, might look like this:

  ```
  "StudentID = ?"
  ```

  Suppose you obtain the student ID in the variable `strInputID`. When you set a parameter to `strInputID` (for example, the student ID 100) the value of the variable is bound to the parameter placeholder represented by the "?" in the filter string.

  Assign the parameter value as follows:

  ```
  strInputID = "100";
  ...
  m_strParam = strInputID;
  ```

  You would not want to set up a filter string this way:

  ```
  m_strFilter = "StudentID = 100";   // 100 is incorrectly quoted
                                     // for some drivers
  ```

For a discussion of how to use quotes correctly for filter strings, see Recordset: Filtering Records (ODBC).

The parameter value is different each time you requery the recordset for a new student ID.

> **Tip**
>
> Using a parameter is more efficient than simply a filter. For a parameterized recordset, the database must process a SQL **SELECT** statement only once. For a filtered recordset without parameters, the **SELECT** statement must be processed each time you **Requery** with a new filter value.

For more information about filters, see Recordset: Filtering Records (ODBC).

Parameterizing Your Recordset Class

> **Note**
>
> This section applies to objects derived from **CRecordset** in which bulk row fetching has not been implemented. If you are using bulk row fetching, implementing parameters is a similar process. For more information, see Recordset: Fetching Records in Bulk (ODBC).

Before you create your recordset class, determine what parameters you need, what their data types are, and how the recordset uses them.

To parameterize a recordset class

1. Run the MFC ODBC Consumer Wizard from **Add Class** to create the class.
2. Specify field data members for the recordset's columns.
3. After the wizard writes the class to a file in your project, go to the .h file and manually add one or more parameter data members to the class declaration. The addition might look something like the following example, part of a snapshot class designed to answer the query "Which students are in the senior class?"

   ```
   class CStudentSet : public CRecordset
   {
   // Field/Param Data
       CString m_strFirstName;
       CString m_strLastName;
       CString m_strStudentID;
       CString m_strGradYear;

       CString m_strGradYrParam;
   };
   ```

   Add your parameter data members after the wizard-generated field data members. The convention is to append the word "Param" to each user-defined parameter name.

4. Modify the DoFieldExchange member function definition in the .cpp file. Add an RFX function call for each parameter data member you added to the class. For information about writing your RFX functions, see Record Field Exchange: How RFX Works. Precede the RFX calls for the parameters with a single call to:

   ```
   pFX->SetFieldType( CFieldExchange::param );
   // RFX calls for parameter data members
   ```

5. In the constructor of your recordset class, increment the count of parameters, **m_nParams**.

   For information, see Record Field Exchange: Working with the Wizard Code.

6. When you write the code that creates a recordset object of this class, place a "?" (question mark) symbol in each place in your SQL statement strings where a parameter is to be replaced.

At run time, "?" placeholders are filled, in order, by the parameter values you pass. The first parameter data member set after the SetFieldType call replaces the first "?" in the SQL string, the second parameter data member replaces the second "?", and so on.

Note

Parameter order is important: the order of RFX calls for parameters in your `DoFieldExchange` function must match the order of the parameter placeholders in your SQL string.

Tip

The most likely string to work with is the string you specify (if any) for the class's m_strFilter data member, but some ODBC drivers might allow parameters in other SQL clauses.

Passing Parameter Values at Run Time

You must specify parameter values before you call **Open** (for a new recordset object) or **Requery** (for an existing one).

To pass parameter values to a recordset object at run time

1. Construct the recordset object.
2. Prepare a string or strings, such as the **m_strFilter** string, containing the SQL statement, or parts of it. Put "?" placeholders where the parameter information is to go.
3. Assign a run-time parameter value to each parameter data member of the object.
4. Call the **Open** member function (or **Requery**, for an existing recordset).

For example, suppose you want to specify a filter string for your recordset using information obtained at run time. Assume you have constructed a recordset of class `CStudentSet` earlier — called `rsStudents` — and now want to requery it for a particular kind of student information.

```
// Set up a filter string with
// parameter placeholders
rsStudents.m_strFilter = "GradYear <= ?";

// Obtain or calculate parameter values
// to pass--simply assigned here
CString strGradYear = GetCurrentAcademicYear( );

// Assign the values to parameter data members
rsStudents.m_strGradYrParam = strGradYear;

// Run the query
if( !rsStudents.Requery( ) )
    return FALSE;
```

The recordset contains records for those students whose records meet the conditions specified by the filter, which was constructed from run-time parameters. In this case, the recordset contains records for all senior students.

Note

If needed, you can set the value of a parameter data member to Null, using SetParamNull. You can likewise check whether a parameter data member is Null, using IsFieldNull.

See Also

Concepts

Recordset (ODBC)
Recordset: Adding, Updating, and Deleting Records (ODBC)
Recordset: How Recordsets Select Records (ODBC)

Recordset: Adding, Updating, and Deleting Records (ODBC)

This topic applies to the MFC ODBC classes.

Note
You can now add records in bulk more efficiently. For more information, see Recordset: Adding Records in Bulk (ODBC).

Note
This topic applies to objects derived from **CRecordset** in which bulk row fetching has not been implemented. If you are usin g bulk row fetching, see Recordset: Fetching Records in Bulk (ODBC).

Updateable snapshots and dynasets allow you to add, edit (update), and delete records. This topic explains:

- How to determine whether your recordset is updatable.
- How to add a new record.
- How to edit an existing record.
- How to delete a record.

For more information about how updates are carried out and how your updates appear to other users, see Recordset: How Recordsets Update Records (ODBC). Normally, when you add, edit, or delete a record, the recordset changes the data source immediately. You can instead batch groups of related updates into transactions. If a transaction is in progress, the update does not become final until you commit the transaction. This allows you to take back or roll back the changes. For information about transactions, see Transaction (ODBC).

The following table summarizes the options available for recordsets with different update characteristics.

Recordset Read/Update Options

Type	Read	Edit record	Delete record	Add new (append)
Read-only	Y	N	N	N
Append-only	Y	N	N	Y
Fully updatable	Y	Y	Y	Y

Determining Whether Your Recordset is Updateable

A recordset object is updateable if the data source is updateable and you opened the recordset as updateable. Its updateability also depends on the SQL statement you use, the capabilities of your ODBC driver, and whether the ODBC Cursor Library is in memory. You cannot update a read-only recordset or data source.

To determine whether your recordset is updatable

- Call the recordset object's CanUpdate member function.

  CanUpdate returns a nonzero value if the recordset is updateable.

By default, recordsets are fully updateable (you can perform **AddNew**, **Edit**, and **Delete** operations). But you can also use the appendOnly option to open updateable recordsets. A recordset opened this way allows only the addition of new records with **AddNew**. You cannot edit or delete existing records. You can test whether a recordset is open only for appending by calling the CanAppend member function. **CanAppend** returns a nonzero value if the recordset is either fully updateable or open only for appending.

The following code shows how you might use **CanUpdate** for a recordset object called `rsStudentSet`:

```
if( !rsStudentSet.Open( ) )
    return FALSE;
if( !rsStudentSet.CanUpdate( ) )
```

```
{
    AfxMessageBox( "Unable to update the Student recordset." );
    return;
}
```

Caution

When you prepare to update a recordset by calling **Update**, take care that your recordset includes all columns making up the primary key of the table (or all of the columns of any unique index on the table). In some cases, the framework can use only the columns selected in your recordset to identify which record in your table to update. Without all the necessary columns, multiple records might be updated in the table, possibly damaging the referential integrity of the table. In this case, the framework throws exceptions when you call **Update**.

Adding a Record to a Recordset

You can add new records to a recordset if its CanAppend member function returns a nonzero value.

To add a new record to a recordset

1. Make sure the recordset is appendable.
2. Call the recordset object's AddNew member function.

   AddNew prepares the recordset to act as an edit buffer. All field data members are set to the special value Null and marked as unchanged so only changed (dirty) values are written to the data source when you call Update.
3. Set the values of the new record's field data members.

   Assign values to the field data members. Those you do not assign are not written to the data source.
4. Call the recordset object's **Update** member function.

   Update completes the addition by writing the new record to the data source. For information about happens if you fail to call **Update**, see Recordset: How Recordsets Update Records (ODBC).

For information about how adding records works and about when added records are visible in your recordset, see Recordset: How AddNew, Edit, and Delete Work (ODBC).

The following example shows how to add a new record:

```
if( !rsStudent.Open( ) )
    return FALSE;
if( !rsStudent.CanAppend( ) )
    return FALSE;                    // no field values were set
rsStudent.AddNew( );
rsStudent.m_strName = strName;
rsStudent.m_strCity = strCity;
rsStudent.m_strStreet = strStreet;
if( !rsStudent.Update( ) )
{
    AfxMessageBox( "Record not added; no field values were set." );
    return FALSE;
}
```

Tip

To cancel an **AddNew** or **Edit** call, simply make another call to **AddNew** or **Edit** or call **Move** with the **AFX_MOVE_REFRESH** parameter. Data members are reset to their previous values and you are still in **Edit** or **Add** mode.

Editing a Record in a Recordset

You can edit existing records if your recordset's CanUpdate member function returns a nonzero value.

To edit an existing record in a recordset

1. Make sure the recordset is updateable.
2. Scroll to the record you want to update.
3. Call the recordset object's Edit member function.

   Edit prepares the recordset to act as an edit buffer. All field data members are marked so that the recordset can tell later whether they were changed. The new values for changed field data members are written to the data source when you call Update.
4. Set the values of the new record's field data members.

   Assign values to the field data members. Those you do not assign values remain unchanged.
5. Call the recordset object's **Update** member function.

   Update completes the edit by writing the changed record to the data source. For information about happens if you fail to call **Update**, see Recordset: How Recordsets Update Records (ODBC).

After you edit a record, the edited record remains the current record.

The following example shows an **Edit** operation. It assumes the user has moved to a record he or she wants to edit.

```
rsStudent.Edit( );
rsStudent.m_strStreet = strNewStreet;
rsStudent.m_strCity = strNewCity;
rsStudent.m_strState = strNewState;
rsStudent.m_strPostalCode = strNewPostalCode;
if( !rsStudent.Update( ) )
{
    AfxMessageBox( "Record not updated; no field values were set." );
    return FALSE;
}
```

Tip

To cancel an **AddNew** or **Edit** call, simply make another call to **AddNew** or **Edit** or call **Move** with the **AFX_MOVE_REFRESH** parameter. Data members are reset to their previous values and you are still in **Edit** or **Add** mode.

Deleting a Record from a Recordset

You can delete records if your recordset's CanUpdate member function returns a nonzero value.

To delete a record

1. Make sure the recordset is updateable.
2. Scroll to the record you want to update.
3. Call the recordset object's Delete member function.

   Delete immediately marks the record as deleted, both in the recordset and on the data source.

   Unlike **AddNew** and **Edit**, **Delete** has no corresponding **Update** call.
4. Scroll to another record.

   Note

   When moving through the recordset, deleted records might not be skipped. For more information, see the IsDeleted member function.

The following example shows a **Delete** operation. It assumes that the user has moved to a record that the user wants to delete. After **Delete** is called, it is important to move to a new record.

```
rsStudent.Delete( );
```

```
rsStudent.MoveNext( );
```

For more information about the effects of the **AddNew**, **Edit**, and **Delete** member functions, see Recordset: How Recordsets Update Records (ODBC).

See Also

Concepts

Recordset (ODBC)

Recordset: Locking Records (ODBC)

Recordset: Adding Records in Bulk (ODBC)

This topic applies to the MFC ODBC classes.

The MFC CRecordset class has a new optimization that improves efficiency when you are adding new records in bulk to a table.

Note
This topic applies to objects derived from **CRecordset** in which bulk row fetching has not been implemented. If you are usin g bulk row fetching, see Recordset: Fetching Records in Bulk (ODBC).

A new option for the **dwOptions** parameter to the CRecordset::Open member function, **optimizeBulkAdd**, improves performance when you are adding multiple records consecutively without calling **Requery** or **Close**. Only those fields that are dirty before the first **Update** call are marked as dirty for subsequent **AddNew**/**Update** calls.

If you are using the database classes to take advantage of the **::SQLSetPos** ODBC API function for adding, editing, and deleting records, this optimization is unnecessary.

If the ODBC Cursor Library is loaded or the ODBC driver does not support adding, editing, and deleting through **::SQLSetPos**, this optimization should improve bulk add performance. To turn on this optimization, set the **dwOptions** parameter in the **Open** call for your recordset to the following:

```
appendOnly | optimizeBulkAdd
```

See Also

Concepts

Recordset (ODBC)
Recordset: Adding, Updating, and Deleting Records (ODBC)
Recordset: Locking Records (ODBC)

Recordset: Locking Records (ODBC)

This topic applies to the MFC ODBC classes.

This topic explains:

- The kinds of record locking available.
- How to lock records in your recordset during updates.

When you use a recordset to update a record on the data source, your application can lock the record so no other user can update the record at the same time. The state of a record updated by two users at the same time is undefined unless the system can guarantee that two users cannot update a record simultaneously.

Note
This topic applies to objects derived from **CRecordset** in which bulk row fetching has not been implemented. If you have implemented bulk row fetching, some of the information does not apply. For example, you cannot call the **Edit** and **Update** member functions. For more information about bulk row fetching, see Recordset: Fetching Records in Bulk (ODBC).

Record-Locking Modes

The database classes provide two record-locking modes:

- Optimistic locking (the default)
- Pessimistic locking

Updating a record occurs in three steps:

1. You begin the operation by calling the Edit member function.
2. You change the appropriate fields of the current record.
3. You end the operation — and normally commit the update — by calling the Update member function.

Optimistic locking locks the record on the data source only during the **Update** call. If you use optimistic locking in a multiuser environment, the application should handle an **Update** failure condition. Pessimistic locking locks the record as soon as you call **Edit** and does not release it until you call **Update** (failures are indicated through the **CDBException** mechanism, not by a value of **FALSE** returned by **Update**). Pessimistic locking has a potential performance penalty for other users, because concurrent access to the same record might have to wait until completion of your application's **Update** process.

Locking Records in Your Recordset

If you want to change a recordset object's locking mode from the default, you must change the mode before you call **Edit**.

To change the current locking mode for your recordset

- Call the SetLockingMode member function, specifying either **CRecordset::pessimistic** or **CRecordset::optimistic**.

The new locking mode remains in effect until you change it again or the recordset is closed.

Note
Relatively few ODBC drivers currently support pessimistic locking.

See Also

Concepts

Recordset (ODBC)
Recordset: Performing a Join (ODBC)
Recordset: Adding, Updating, and Deleting Records (ODBC)

Recordset: Performing a Join (ODBC)

This topic applies to the MFC ODBC classes.

What a Join Is

The join operation, a common data-access task, lets you work with data from more than one table using a single recordset object. Joining two or more tables yields a recordset that can contain columns from each table, but appears as a single table to your application. Sometimes the join uses all columns from all tables, but sometimes the SQL **SELECT** clause in a join uses only some of the columns from each table. The database classes support read-only joins but not updateable joins.

To select records containing columns from joined tables, you need the following items:

- A table list containing the names of all tables being joined.
- A column list containing the names of all participating columns. Columns with the same name but from different tables are qualified by the table name.
- A filter (SQL **WHERE** clause) that specifies the columns on which the tables are joined. This filter takes the form "Table1.KeyCol = Table2.KeyCol" and actually accomplishes the join.

You can join more than two tables in the same way by equating multiple pairs of columns, each pair joined by the SQL keyword **AND**.

See Also

Concepts

Recordset (ODBC)
Recordset: Declaring a Class for a Predefined Query (ODBC)
Recordset: Declaring a Class for a Table (ODBC)
Recordset: Requerying a Recordset (ODBC)

Recordset: Declaring a Class for a Predefined Query (ODBC)

This topic applies to the MFC ODBC classes.

This topic explains how to create a recordset class for a predefined query (sometimes called a stored procedure, as in Microsoft SQL Server).

Note

This topic applies to objects derived from **CRecordset** in which bulk row fetching has not been implemented. If bulk row fetc hing is implemented, the process is very similar. To understand the differences between recordsets that implement bulk row fetching and those that do not, see Recordset: Fetching Records in Bulk (ODBC).

Some database management systems (DBMSs) allow you to create a predefined query and call it from your programs like a function. The query has a name, might take parameters, and might return records. The procedure in this topic describes how to call a predefined query that returns records (and perhaps takes parameters).

The database classes do not support updating predefined queries. The difference between a snapshot predefined query and a dynaset predefined query is not updateability but whether changes made by other users (or other recordsets in your program) are visible in your recordset.

Tip

You do not need a recordset to call a predefined query that does not return records. Prepare the SQL statement as described below, but execute it by calling the **CDatabase** member function ExecuteSQL.

You can create a single recordset class to manage calling a predefined query, but you must do some of the work yourself. The wizards do not support creating a class specifically for this purpose.

To create a class for calling a predefined query (stored procedure)

1. Use the MFC ODBC Consumer Wizard from **Add Class** to create a recordset class for the table that contributes the most columns returned by the query. This gives you a head start.

2. Manually add field data members for any columns of any tables that the query returns but that the wizard did not create for you.

   For example, if the query returns three columns each from two additional tables, add six field data members (of the appropriate data types) to the class.

3. Manually add RFX function calls in the DoFieldExchange member function of the class, one corresponding to the data type of each added field data member.

   ```
   Immediately before these RFX calls, call <MSHelp:link keywords="_mfc_CFieldExchange.3a
   3a.SetFieldType" TABINDEX="0">SetFieldType</MSHelp:link>, as shown here:
   pFX->SetFieldType( CFieldExchange::outputColumn );
   ```

   Note

   You must know the data types and the order of columns returned in the result set. The order of RFX function calls in `DoFieldExchange` must match the order of result set columns.

4. Manually add initializations for the new field data members in the recordset class constructor.

   You must also increment the initialization value for the m_nFields data member. The wizard writes the initialization, but it only covers the field data members it adds for you. For example:

   ```
   m_nFields += 6;
   ```

Some data types should not be initialized here, for example, **CLongBinary** or byte arrays.

5. If the query takes parameters, add a parameter data member for each parameter, an RFX function call for each, and an initialization for each.

6. You must increment **m_nParams** for each added parameter, as you did **m_nFields** for added fields in step 4 of this procedure. For more information, see Recordset: Parameterizing a Recordset (ODBC).

7. Manually write a SQL statement string with the following form:

   ```
   {CALL proc-name [(? [, ?]...)]}
   ```

   where **CALL** is an ODBC keyword, **proc-name** is the name of the query as it is known on the data source, and the "?" items are placeholders for the parameter values you supply to the recordset at run time (if any). The following example prepares a placeholder for one parameter:

   ```
   CString mySQL = "{CALL Delinquent_Accts (?)}";
   ```

8. In the code that opens the recordset, set the values of the recordset's parameter data members and then call the **Open** member function, passing your SQL string for the **lpszSQL** parameter. Or instead, replace the string returned by the `GetDefaultSQL` member function in your class.

The following examples show the procedure for calling a predefined query, named `Delinquent_Accts`, which takes one parameter for a sales district number. This query returns three columns: `Acct_No`, `L_Name`, `Phone`. All columns are from the Customers table.

The following recordset specifies field data members for the columns the query returns and a parameter for the sales district number requested at run time.

```
class CDelinquents : public CRecordset
{
// Field/Param Data
    LONG m_lAcct_No;
    CString m_strL_Name;
    CString m_strPhone;
    LONG m_lDistParam;
    // ...
};
```

This class declaration is as the wizard writes it, except for the `m_lDistParam` member added manually. Other members are not shown here.

The next example shows the initializations for the data members in the `CDelinquents` constructor.

```
CDelinquents::CDelinquents(CDatabase* pdb)
   : CRecordset(pdb)
{
    // Wizard-generated params:
    m_lAcct_No = 0;
    m_strL_Name = "";
    m_strPhone = "";
    m_nFields = 3;
    // User-defined params:
    m_nParams = 1;
    m_lDistParam = 0;
}
```

Note the initializations for m_nFields and m_nParams. The wizard initializes **m_nFields**; you initialize **m_nParams**.

The next example shows the RFX functions in `CDelinquents::DoFieldExchange`:

```
void CDelinquents::DoFieldExchange(CFieldExchange* pFX)
{
```

```
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Long(pFX, "Acct_No", m_lAcct_No);
    RFX_Text(pFX, "L_Name", m_strL_Name);
    RFX_Text(pFX, "Phone", m_strPhone);
    pFX->SetFieldType(CFieldExchange::param);
    RFX_Long(pFX, "Dist_No", m_lDistParam);
}
```

Besides making the RFX calls for the three returned columns, this code manages binding the parameter you pass at run time. The parameter is keyed to the `Dist_No` (district number) column.

The next example shows how to set up the SQL string and how to use it to open the recordset.

```
// Construct a CDelinquents recordset object
CDelinquents rsDel( NULL );
CString strSQL = "{CALL Delinquent_Accts (?)}"
// Specify a parameter value (obtained earlier from the user)
rsDel.m_lDistParam = lDistrict;
// Open the recordset and run the query
if( rsDel.Open( CRecordset::snapshot, strSQL ) )
    // Use the recordset ...
```

This code constructs a snapshot, passes it a parameter obtained earlier from the user, and calls the predefined query. When the query runs, it returns records for the specified sales district. Each record contains columns for the account number, customer's last name, and customer's phone number.

Tip

You might want to handle a return value (output parameter) from a stored procedure. For more information and an example, see CFieldExchange::SetFieldType.

See Also

Concepts

Recordset (ODBC)
Recordset: Requerying a Recordset (ODBC)
Recordset: Declaring a Class for a Table (ODBC)
Recordset: Performing a Join (ODBC)

Recordset: Requerying a Recordset (ODBC)

This topic applies to the MFC ODBC classes.

This topic explains how you can use a recordset object to requery (that is, refresh) itself from the database and when you might want to do that with the Requery member function.

The principal reasons for requerying a recordset are to:

- Bring the recordset up to date with respect to records added by you or by other users and records deleted by other users (those you delete are already reflected in the recordset).
- Refresh the recordset based on changing parameter values.

Bringing the Recordset Up to Date

Frequently, you will want to requery your recordset object to bring it up to date. In a multiuser database environment, other users can make changes to the data during the life of your recordset. For more information about when your recordset reflects changes made by other users and when other users' recordsets reflect your changes, see Recordset: How Recordsets Update Records (ODBC) and Dynaset.

Requerying Based on New Parameters

Another frequent — and equally important — use of Requery is to select a new set of records based on changing parameter values.

Tip
Query speed is probably significantly faster if you call **Requery** with changing parameter values than if you call **Open** again.

Requerying Dynasets vs. Snapshots

Because dynasets are meant to present a set of records with dynamic up-to-date data, you want to requery dynasets often if you want to reflect other users' additions. Snapshots, on the other hand, are useful because you can safely rely on their static contents while you prepare reports, calculate totals, and so on. Still, you might sometimes want to requery a snapshot as well. In a multiuser environment, snapshot data might lose synchronization with the data source as other users change the database.

To requery a recordset object

- Call the Requery member function of the object.

Alternatively, you can close and reopen the original recordset. In either case, the new recordset represents the current state of the data source.

For an example, see Record Views: Filling a List Box from a Second Recordset.

Tip
To optimize **Requery** performance, avoid changing the recordset's filter or sort. Change only the parameter value before call ing **Requery**.

If the **Requery** call fails, you can retry the call; otherwise, your application should terminate gracefully. A call to **Requery** or **Open** might fail for any of a number of reasons. Perhaps a network error occurs; or, during the call, after the existing data is released but before the new data is obtained, another user might get exclusive access; or the table on which your recordset depends could be deleted.

See Also

Concepts

Recordset (ODBC)
Recordset: Dynamically Binding Data Columns (ODBC)
Recordset: Creating and Closing Recordsets (ODBC)

Recordset: Dynamically Binding Data Columns (ODBC)

This topic applies to the MFC ODBC classes.

Recordsets manage binding table columns that you specify at design time, but there are cases when you might want to bind columns that were unknown to you at design time. This topic explains:

- When you might want to bind columns dynamically to a recordset.
- How to bind columns dynamically at run time.

Note
This topic applies to objects derived from **CRecordset** in which bulk row fetching has not been implemented. The techniques described generally are not recommended if you are using bulk row fetching. For more information about bulk row fetching, see Recordset: Fetching Records in Bulk (ODBC).

When You Might Bind Columns Dynamically

At design time, the MFC Application Wizard or MFC ODBC Consumer Wizard (from **Add Class**) creates recordset classes based on the known tables and columns on your data source. Databases can change between when you design them and later when your application uses those tables and columns at run time. You or another user might add or drop a table or add or drop columns from a table that your application's recordset relies on. This probably is not a concern for all data-access applications, but if it is for yours, how can you cope with changes in the database schema, other than by redesigning and recompiling? The purpose of this topic is to answer that question.

This topic describes the most common case in which you might bind columns dynamically — having begun with a recordset based on a known database schema, you want to handle additional columns at run time. The topic further assumes that the additional columns map to **CString** field data members, the most common case, although suggestions are supplied to help you manage other data types.

With a small amount of extra code, you can:

- Determine what columns are available at run time.
- Bind additional columns to your recordset dynamically, at run time.

Your recordset still contains data members for the columns you knew about at design time. It also contains a small amount of extra code that dynamically determines whether any new columns have been added to your target table and, if so, binds these new columns to dynamically allocated storage (rather than to recordset data members).

This topic does not cover other dynamic binding cases, such as dropped tables or columns. For those cases, you need to use ODBC API calls more directly. For information, see the ODBC SDK *Programmer's Reference* on the MSDN Library CD.

How to Bind Columns Dynamically

To bind columns dynamically, you must know (or be able to determine) the names of the additional columns. You must also allocate storage for the additional field data members, specify their names and their types, and specify the number of columns you are adding.

The following discussion mentions two different recordsets. The first is the main recordset that selects records from the target table. The second is a special column recordset used to get information about the columns in your target table.

General Process

At the most general level, you follow these steps:

1. Construct your main recordset object.

   Optionally, pass a pointer to an open **CDatabase** object or be able to supply connection information to the column recordset in some other way.

2. Take steps to add columns dynamically.

   See the process described in Adding the Columns below.

3. Open your main recordset.

   The recordset selects records and uses record field exchange (RFX) to bind both the static columns (those mapped to recordset field data members) and the dynamic columns (mapped to extra storage that you allocate).

Adding the Columns

Dynamically binding added columns at run time requires the following steps:

1. Determine at run time what columns are in the target table. Extract from that information a list of the columns that have been added to the table since your recordset class was designed.

   A good approach is to use a column recordset class designed to query the data source for column information for the target table (such as column name and data type).

2. Provide storage for the new field data members. Because your main recordset class does not have field data members for unknown columns, you must provide a place to store the names, result values, and possibly data type information (if the columns are different data types).

   One approach is to build one or more dynamic lists, one for the new columns' names, another for their result values, and a third for their data types (if necessary). These lists, particularly the value list, provide the information and the necessary storage for binding. The following figure illustrates building the lists.

   Building Lists of Columns to Bind Dynamically

   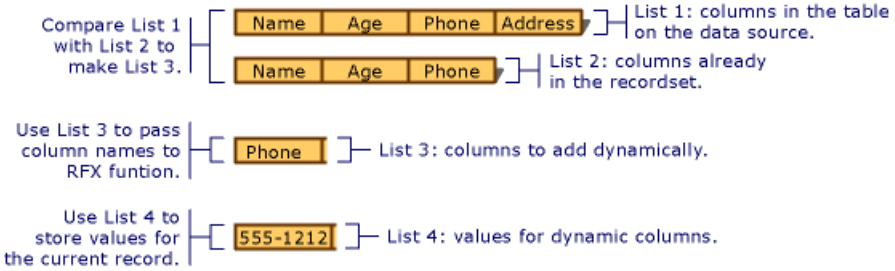

3. Add an RFX function call in your main recordset's `DoFieldExchange` function for each added column. These RFX calls do the work of fetching a record, including the additional columns, and binding the columns to recordset data members or to your dynamically supplied storage for them.

   One approach is to add a loop to your main recordset's `DoFieldExchange` function that loops through your list of new columns, calling the appropriate RFX function for each column in the list. On each RFX call, pass a column name from the column name list and a storage location in the corresponding member of the result value list.

Lists of Columns

The four lists you need to work with are shown in the following table.

Current-Table-Columns (List 1 in the illustration)

A list of the columns currently in the table on the data source. This list might match the list of columns currently bound in your recordset.

Bound-Recordset-Columns (List 2 in the illustration)

A list of the columns bound in your recordset. These columns already have RFX statements in your `DoFieldExchange` function.

Columns-To-Bind-Dynamically (List 3 in the illustration)

A list of columns in the table but not in your recordset. These are the columns you want to bind dynamically.

Dynamic-Column-Values (List 4 in the illustration)

A list containing storage for the values retrieved from the columns you bind dynamically. Elements of this list correspond to those in Columns-to-Bind-Dynamically, one to one.

Building Your Lists

With a general strategy in mind, you can turn to the details. The procedures in the rest of this topic show you how to build the lists shown in Lists of Columns. The procedures guide you through:

- Determining the names of columns not in your recordset.

- Providing dynamic storage for columns newly added to the table.
- Dynamically adding RFX calls for new columns.

Determining Which Table Columns Are Not in Your Recordset

Build a list (Bound-Recordset-Columns, as in List 2 in the illustration) that contains a list of the columns already bound in your main recordset. Then build a list (Columns-to-Bind-Dynamically, derived from Current-Table-Columns and Bound-Recordset-Columns) that contains column names that are in the table on the data source but not in your main recordset.

To determine the names of columns not in the recordset (Columns-to-Bind-Dynamically)

1. Build a list (Bound-Recordset-Columns) of the columns already bound in your main recordset.

   One approach is to create Bound-Recordset-Columns at design time. You can visually examine the RFX function calls in the recordset's `DoFieldExchange` function to get these names. Then, set up your list as an array initialized with the names.

   For example, the illustration shows Bound-Recordset-Columns (List 2) with three elements. Bound-Recordset-Columns is missing the Phone column shown in Current-Table-Columns (List 1).

2. Compare Current-Table-Columns and Bound-Recordset-Columns to build a list (Columns-to-Bind-Dynamically) of the columns not already bound in your main recordset.

   One approach is to loop through your list of columns in the table at run time (Current-Table-Columns) and your list of columns already bound in your recordset (Bound-Recordset-Columns) in parallel. Into Columns-to-Bind-Dynamically put any names in Current-Table-Columns that do not appear in Bound-Recordset-Columns.

   For example, the illustration shows Columns-to-Bind-Dynamically (List 3) with one element: the Phone column found in Current-Table-Columns (List 1) but not in Bound-Recordset-Columns (List 2).

3. Build a list of Dynamic-Column-Values (as in List 4 in the illustration) in which to store the data values corresponding to each column name stored in your list of columns to bind dynamically (Columns-to-Bind-Dynamically).

   The elements of this list play the role of new recordset field data members. They are the storage locations to which the dynamic columns are bound. For descriptions of the lists, see Lists of Columns.

Providing Storage for the New Columns

Next, set up storage locations for the columns to be bound dynamically. The idea is to provide a list element in which to store each column's value. These storage locations parallel the recordset member variables, which store the normally bound columns.

To provide dynamic storage for new columns (Dynamic-Column-Values)

- Build Dynamic-Column-Values, parallel to Columns-to-Bind-Dynamically, to contain the value of the data in each column.

  For example, the illustration shows Dynamic-Column-Values (List 4) with one element: a **CString** object containing the actual phone number for the current record: "555-1212".

  In the most common case, Dynamic-Column-Values has elements of type **CString**. If you are dealing with columns of varying data types, you need a list that can contain elements of a variety of types.

The result of the preceding procedures is two main lists: Columns-to-Bind-Dynamically containing the names of columns and Dynamic-Column-Values containing the values in the columns for the current record.

Tip
If the new columns are not all of the same data type, you might want an extra parallel list containing items that somehow def ine the type of each corresponding element in the column list. (You can use the values **AFX_RFX_BOOL**, **AFX_RFX_BYTE**, and so on, for this if you want. These constants are defined in AFXDB.H.) Choose a list type based on how you represent the colu mn data types.

Adding RFX Calls to Bind the Columns

Finally, arrange for the dynamic binding to occur by placing RFX calls for the new columns in your `DoFieldExchange` function.

To dynamically add RFX calls for new columns

- In your main recordset's `DoFieldExchange` member function, add code that loops through your list of new columns (Columns-to-Bind-Dynamically). In each loop, extract a column name from Columns-to-Bind-Dynamically and a result value for the column from Dynamic-Column-Values. Pass these items to an RFX function call appropriate to the data type of the column. For descriptions of the lists, see Lists of Columns.

In the common case, in your **RFX_Text** function calls you extract **CString** objects from the lists, as in the following lines of code, where Columns-to-Bind-Dynamically is a **CStringList** called `m_listName` and Dynamic-Column-Values is a **CStringList** called `m_listValue`:

```
RFX_Text( pFX,
            m_listName.GetNext( posName ),
            m_listValue.GetNext( posValue ));
```

For more information about RFX functions, see Macros and Globals in the *Class Library Reference*.

Tip
If the new columns are different data types, use a switch statement in your loop to call the appropriate RFX function for each type.

When the framework calls `DoFieldExchange` during the **Open** process to bind columns to the recordset, the RFX calls for the static columns bind those columns. Then your loop repeatedly calls RFX functions for the dynamic columns.

See Also

Concepts

Recordset (ODBC)

Recordset: Working with Large Data Items (ODBC)

Recordset: Fetching Records in Bulk (ODBC)

This topic applies to the MFC ODBC classes.

Class **CRecordset** provides support for bulk row fetching, which means that multiple records can be retrieved at once during a single fetch, rather than retrieving one record at a time from the data source. You can implement bulk row fetching only in a derived **CRecordset** class. The process of transferring data from the data source to the recordset object is called bulk record field exchange (Bulk RFX). Note that if you are not using bulk row fetching in a **CRecordset**-derived class, data is transferred via record field exchange (RFX). For more information, see Record Field Exchange (RFX).

This topic explains:

- How CRecordset supports bulk row fetching.
- Some special considerations when using bulk row fetching.
- How to implement bulk record field exchange.

How CRecordset Supports Bulk Row Fetching

Before opening your recordset object, you can define a rowset size with the **SetRowsetSize** member function. The rowset size specifies how many records should be retrieved during a single fetch. When bulk row fetching is implemented, the default rowset size is 25. If bulk row fetching is not implemented, the rowset size remains fixed at 1.

After you have initialized the rowset size, call the Open member function. Here you must specify the **CRecordset::useMultiRowFetch** option of the **dwOptions** parameter to implement bulk row fetching. You can additionally set the **CRecordset::userAllocMultiRowBuffers** option. The bulk record field exchange mechanism uses arrays to store the multiple rows of data retrieved during a fetch. These storage buffers can be allocated automatically by the framework or you can allocate them manually. Specifying the **CRecordset::userAllocMultiRowBuffers** option means that you will do the allocation.

The following table lists the member functions provided by **CRecordset** to support bulk row fetching.

Member function	Description
CheckRowsetError	Virtual function that handles any errors that occur during fetching.
DoBulkFieldExchange	Implements bulk record field exchange. Called automatically to transfers multiple rows of data from the data source to the recordset object.
GetRowsetSize	Retrieves the current setting for the rowset size.
GetRowsFetched	Tells how many rows were actually retrieved after a given fetch. In most cases, this is the rowset size, unless an incomplete rowset was fetched.
GetRowStatus	Returns a fetch status for a particular row within a rowset.
RefreshRowset	Refreshes the data and status of a particular row within a rowset.
SetRowsetCursorPosition	Moves the cursor to a particular row within a rowset.
SetRowsetSize	Virtual function that changes the setting for the rowset size to the specified value.

Special Considerations

Although bulk row fetching is a performance gain, certain features operate differently. Before you decide to implement bulk row fetching, consider the following:

- The framework automatically calls the **DoBulkFieldExchange** member function to transfer data from the data source to the recordset object. However, data is not transferred from the recordset back to the data source. Calling the **AddNew**, **Edit**, **Delete**, or **Update** member functions results in a failed assertion. Although **CRecordset** currently does not provide

a mechanism for updating bulk rows of data, you can write your own functions by using the ODBC API function **SQLSetPos**. For more information about **SQLSetPos**, see the *ODBC SDK Programmer's Reference* in the MSDN documentation.

- The member functions **IsDeleted**, **IsFieldDirty**, **IsFieldNull**, **IsFieldNullable**, **SetFieldDirty**, and **SetFieldNull** cannot be used on recordsets that implement bulk row fetching. However, you can call **GetRowStatus** in place of **IsDeleted**, and **GetODBCFieldInfo** in place of **IsFieldNullable**.
- The **Move** operations repositions your recordset by rowset. For example, suppose you open a recordset that has 100 records with an initial rowset size of 10. **Open** fetches rows 1 through 10, with the current record positioned on row 1. A call to **MoveNext** fetches the next rowset, not the next row. This rowset consists of rows 11 through 20, with the current record positioned on row 11. Note that **MoveNext** and **Move(1)** are not equivalent when bulk row fetching is implemented. **Move(1)** fetches the rowset that begins 1 row from the current record. In this example, calling **Move(1)** after calling **Open** fetches the rowset consisting of rows 2 through 11, with the current record positioned on row 2. For more information, see the Move member function.
- Unlike record field exchange, the wizards do not support bulk record field exchange. This means that you must manually declare your field data members and manually override **DoBulkFieldExchange** by writing calls to the Bulk RFX functions. For more information, see Record Field Exchange Functions in the *Class Library Reference*.

How to Implement Bulk Record Field Exchange

Bulk record field exchange transfers a rowset of data from the data source to the recordset object. The Bulk RFX functions use arrays to store this data, as well as arrays to store the length of each data item in the rowset. In your class definition, you must define your field data members as pointers to access the arrays of data. In addition, you must define a set of pointers to access the arrays of lengths. Any parameter data members should not be declared as pointers; declaring parameter data members when using bulk record field exchange is the same as declaring them when using record field exchange. The following code shows a simple example:

```
class MultiRowSet : public CRecordset
{
public:
   // Field/Param Data
      // field data members
      long* m_rgID;
      LPSTR m_rgName;

      // pointers for the lengths
      // of the field data
      long* m_rgIDLengths;
      long* m_rgNameLengths;

      // input parameter data member
      CString m_strNameParam;

   .
   .
   .
}
```

You can either allocate these storage buffers manually or have the framework do the allocation. To allocate the buffers yourself, you must specify the **CRecordset::userAllocMultiRowBuffers** option of the **dwOptions** parameter in the **Open** member function. Be sure to set the sizes of the arrays at least equal to the rowset size. If you want to have the framework do the allocation, you should initialize your pointers to **NULL.** This is typically done in the recordset object's constructor:

```
MultiRowSet::MultiRowSet( CDatabase* pDB )
   : CRecordset( pDB )
{
   m_rgID = NULL;
   m_rgName = NULL;
   m_rgIDLengths = NULL;
   m_rgNameLengths = NULL;
   m_strNameParam = "";

   m_nFields = 2;
```

```
    m_nParams = 1;

    .
    .
    .
}
```

Finally, you must override the **DoBulkFieldExchange** member function. For the field data members, call the Bulk RFX functions; for any parameter data members, call the RFX functions. If you opened the recordset by passing a SQL statement or stored procedure to **Open**, the order in which you make the Bulk RFX calls must correspond to the order of the columns in the recordset; similarly, the order of the RFX calls for parameters must correspond to the order of parameters in the SQL statement or stored procedure.

```
void MultiRowSet::DoBulkFieldExchange( CFieldExchange* pFX )
{
   // call the Bulk RFX functions
   // for field data members
   pFX->SetFieldType( CFieldExchange::outputColumn );
   RFX_Long_Bulk( pFX, _T( "[colRecID]" ),
                  &m_rgID, &m_rgIDLengths );
   RFX_Text_Bulk( pFX, _T( "[colName]" ),
                  &m_rgName, &m_rgNameLengths, 30 );


   // call the RFX functions for
   // for parameter data members
   pFX->SetFieldType( CFieldExchange::inputParam );
   RFX_Text( pFX, "NameParam", m_strNameParam );
}
```

Note

You must call the **Close** member function before your derived **CRecordset** class goes out of scope. This ensures that any memory allocated by the framework are freed. It is good programming practice to always explicitly call **Close**, regardless of whether you have implemented bulk row fetching.

For more information about record field exchange (RFX), see Record Field Exchange: How RFX Works. For more information about using parameters, see CFieldExchange::SetFieldType and Recordset: Parameterizing a Recordset (ODBC).

See Also

Reference

CRecordset::m_nFields

CRecordset::m_nParams

Concepts

Recordset (ODBC)

Recordset: Working with Large Data Items (ODBC)

This topic applies to both the MFC ODBC classes and the MFC DAO classes.

Note
If you are using the MFC DAO classes, manage your large data items with class CByteArray rather than class CLongBinary. If you are using the MFC ODBC classes with bulk row fetching, use **CLongBinary** rather than **CByteArray**. For more informati on about bulk row fetching, see Recordset: Fetching Records in Bulk (ODBC).

Suppose your database can store large pieces of data, such as bitmaps (employee photographs, maps, pictures of products, OLE objects, and so on). This kind of data is often referred to as a Binary Large Object (or BLOB) because:

- Each field value is large.
- Unlike numbers and other simple data types, it has no predictable size.
- The data is formless from the perspective of your program.

This topic explains what support the database classes provide for working with such objects.

Managing Large Objects

Recordsets have two ways to solve the special difficulty of managing binary large objects. You can use class CByteArray or you can use class CLongBinary. In general, **CByteArray** is the preferred way to manage large binary data.

CByteArray requires more overhead than **CLongBinary** but is more capable, as described in The CByteArray Class. **CLongBinary** is described briefly in The CLongBinary Class.

For detailed information about using **CByteArray** to work with large data items, see Technical Note 45.

CByteArray Class

CByteArray is one of the MFC collection classes. A **CByteArray** object stores a dynamic array of bytes — the array can grow if needed. The class provides fast access by index, as with built-in C++ arrays. **CByteArray** objects can be serialized and dumped for diagnostic purposes. The class supplies member functions for getting and setting specified bytes, inserting and appending bytes, and removing one byte or all bytes. These facilities make parsing the binary data easier. For example, if the binary object is an OLE object, you might have to work through some header bytes to reach the actual object.

Using CByteArray in Recordsets

By giving a field data member of your recordset the type **CByteArray**, you provide a fixed base from which RFX can manage the transfer of such an object between your recordset and the data source and through which you can manipulate the data inside the object. RFX needs a specific site for retrieved data, and you need a way to access the underlying data.

For detailed information about using **CByteArray** to work with large data items, see Technical Note 45.

CLongBinary Class

A CLongBinary object is a simple shell around an **HGLOBAL** handle to a block of storage allocated on the heap. When it binds a table column containing a binary large object, RFX allocates the **HGLOBAL** handle when it needs to transfer the data to the recordset and stores the handle in the **CLongBinary** field of the recordset.

In turn, you use the **HGLOBAL** handle, **m_hData**, to work with the data itself, operating on it as you would on any handle data. This is where CByteArray adds capabilities.

Caution
CLongBinary objects cannot be used as parameters in function calls. In addition, their implementation, which calls **::SQLGetD ata**, necessarily slows scrolling performance for a scrollable snapshot. This might also be true when you use an **::SQLGetDat a** call yourself to retrieve dynamic schema columns.

See Also

Concepts

Recordset (ODBC)
Recordset: Obtaining SUMs and Other Aggregate Results (ODBC)
Record Field Exchange (RFX)

Recordset: Obtaining SUMs and Other Aggregate Results (ODBC)

This topic applies to the MFC ODBC classes.

This topic explains how to obtain aggregate results using the following SQL keywords:

- **SUM** Calculates the total of the values in a column with a numeric data type.
- **MIN** Extracts the smallest value in a column with a numeric data type.
- **MAX** Extracts the largest value in a column with a numeric data type.
- **AVG** Calculates an average value of all the values in a column with a numeric data type.
- **COUNT** Counts the number of records in a column of any data type.

You use these SQL functions to obtain statistical information about the records in a data source rather than to extract records from the data source. The recordset that is created usually consists of a single record (if all columns are aggregates) that contains a value. (There might be more than one record if you used a **GROUP BY** clause.) This value is the result of the calculation or extraction performed by the SQL function.

Tip
To add a SQL **GROUP BY** clause (and possibly a **HAVING** clause) to your SQL statement, append it to the end of **m_strFilter**. For example:

```
m_strFilter = "sales > 10 GROUP BY SALESPERSON_ID";
```

You can limit the number of records you use to obtain aggregate results by filtering and sorting the columns.

Caution
Some aggregation operators return a different data type from the columns over which they are aggregating.

- **SUM** and **AVG** might return the next larger data type (for example, calling with **int** returns **LONG** or **double**).
- **COUNT** usually returns **LONG** regardless of target column type.
- **MAX** and **MIN** return the same data type as the columns they calculate.

  For example, the **Add Class** wizard creates `long m_lSales` to accommodate a Sales column, but you need to replace this with a `double m_dblSumSales` data member to accommodate the aggregate result. See the following example.

To obtain an aggregate result for a recordset

1. Create a recordset as described in Adding an MFC ODBC Consumer containing the columns from which you want to obtain aggregate results.
2. Modify the DoFieldExchange function for the recordset. Replace the string representing the column name (the second argument of the RFX function calls) with a string representing the aggregation function on the column. For example, replace:

   ```
   RFX_Long(pFX, "Sales", m_lSales);
   ```

   with:

   ```
   RFX_Double(pFX, "Sum(Sales)", m_dblSumSales)
   ```

3. Open the recordset. The result of the aggregation operation is left in `m_dblSumSales`.

Note

The wizard actually assigns data member names without Hungarian prefixes. For example, the wizard would produce `m_Sales` for a Sales column, rather than the `m_lSales` name used earlier for illustration.

If you are using a CRecordView class to view the data, you have to change the DDX function call to display the new data member value; in this case, changing it from:

```
DDX_FieldText(pDX, IDC_SUMSALES, m_pSet->m_lSales, m_pSet);
```

To:

```
DDX_FieldText(pDX, IDC_SUMSALES, m_pSet->m_dblSumSales, m_pSet);
```

See Also

Concepts

Recordset (ODBC)

Recordset: How Recordsets Select Records (ODBC)

Recordset: How Recordsets Select Records (ODBC)

This topic applies to the MFC ODBC classes.

This topic explains:

- Your role and your options in selecting records.
- How a recordset constructs its SQL statement and selects records.
- What you can do to customize the selection.

Recordsets select records from a data source through an ODBC driver by sending SQL statements to the driver. The SQL sent depends on how you design and open your recordset class.

Your Options in Selecting Records

The following table shows your options in selecting records.

How and When You Can Affect a Recordset

When you	You can
Declare your recordset class with the **Add Class** wizard	Specify which table to select from. Specify which columns to include. See Adding an MFC ODBC Consumer.
Complete your recordset class implementation	Override member functions such as **OnSetOptions** (advanced) to set application-specific options or to change defaults. Specify parameter data members if you want a parameterized recordset.
Construct a recordset object (before you call **Open**)	Specify a search condition (possibly compound) for use in a **WHERE** clause that filters the records. See Recordset: Filtering Records (ODBC). Specify a sort order for use in an **ORDER BY** clause that sorts the records. See Recordset: Sorting Records (ODBC). Specify parameter values for any parameters you added to the class. See Recordset: Parameterizing a Recordset (ODBC).
Run the recordset's query by calling **Open**	Specify a custom SQL string to replace the default SQL string set up by the wizard. See CRecordset::Open in the *Class Library Reference* and SQL: Customizing Your Recordset's SQL Statement (ODBC).
Call **Requery** to requery the recordset with the latest values on the data source	Specify new parameters, filter, or sort. See Recordset: Requerying a Recordset (ODBC).

How a Recordset Constructs Its SQL Statement

When you call a recordset object's Open member function, **Open** constructs a SQL statement using some or all of the following ingredients:

- The **lpszSQL** parameter passed to **Open**. If not **NULL**, this parameter specifies a custom SQL string or part of one. The framework parses the string. If the string is a SQL **SELECT** statement or an ODBC **CALL** statement, the framework uses the string as the recordset's SQL statement. If the string does not begin with "SELECT" or "{CALL", the framework uses what is supplied to construct a SQL **FROM** clause.
- The string returned by GetDefaultSQL. By default, this is the name of the table you specified for the recordset in the wizard, but you can change what the function returns. The framework calls **GetDefaultSQL** — if the string does not begin with "SELECT" or "{CALL", it is assumed to be a table name, which is used to construct a SQL string.
- The field data members of the recordset, which are to be bound to specific columns of the table. The framework binds

record columns to the addresses of these members, using them as buffers. The framework determines the correlation of field data members to table columns from the RFX or Bulk RFX function calls in the recordset's DoFieldExchange or DoBulkFieldExchange member function.

- The filter for the recordset, if any, contained in the m_strFilter data member. The framework uses this string to construct a SQL **WHERE** clause.
- The sort order for the recordset, if any, contained in the m_strSort data member. The framework uses this string to construct a SQL **ORDER BY** clause.

Tip
To use the SQL **GROUP BY** clause (and possibly the **HAVING** clause), append the clauses to the end of your filter string .

- The values of any parameter data members you specify for the class. You set parameter values just before you call **Open** or **Requery**. The framework binds the parameter values to "?" placeholders in the SQL string. At compile time, you specify the string with placeholders. At run time, the framework fills in the details based on the parameter values you pass.

Open constructs a SQL **SELECT** statement from these ingredients. See Customizing the Selection for details about how the framework uses the ingredients.

After constructing the statement, **Open** sends the SQL to the ODBC Driver Manager (and the ODBC Cursor Library if it is in memory), which sends it on to the ODBC driver for the specific DBMS. The driver communicates with the DBMS to carry out the selection on the data source and fetches the first record. The framework loads the record into the field data members of the recordset.

You can use a combination of these techniques to open tables and to construct a query based on a join of multiple tables. With additional customization, you can call predefined queries (stored procedures), select table columns not known at design time and bind them to recordset fields or you can perform most other data-access tasks. Tasks you cannot accomplish by customizing recordsets can still be accomplished by calling ODBC API functions or directly executing SQL statements with CDatabase::ExecuteSQL.

Customizing the Selection

Besides supplying a filter, a sort order, or parameters, you can take the following actions to customize your recordset's selection:

- Pass a custom SQL string in **lpszSQL** when you call Open for the recordset. Anything you pass in **lpsqSQL** takes precedence over what the GetDefaultSQL member function returns.

  For more information, see SQL: Customizing Your Recordset's SQL Statement (ODBC), which describes the types of SQL statements (or partial statements) you can pass to **Open** and what the framework does with them.

Note
If the custom string you pass does not begin with "SELECT" or "{CALL", MFC assumes it contains a table name. This also applies to the next bulleted item.

- Alter the string that the wizard writes in your recordset's **GetDefaultSQL** member function. Edit the function's code to change what it returns. By default, the wizard writes a **GetDefaultSQL** function that returns a single table name.

  You can have **GetDefaultSQL** return any of the items that you can pass in the **lpszSQL** parameter to **Open**. If you do not pass a custom SQL string in **lpszSQL**, the framework uses the string that **GetDefaultSQL** returns. At a minimum, **GetDefaultSQL** must return a single table name. But you can have it return multiple table names, a full **SELECT** statement, an ODBC **CALL** statement, and so on. For a list of what you can pass to **lpszSQL** — or have **GetDefaultSQL** return — see SQL: Customizing Your Recordset's SQL Statement (ODBC).

  If you are performing a join of two or more tables, rewrite **GetDefaultSQL** to customize the table list used in the SQL **FROM** clause. For more information, see Recordset: Performing a Join (ODBC).

- Manually bind additional field data members, perhaps based on information you obtain about the schema of your data source at run time. You add field data members to the recordset class, RFX or Bulk RFX function calls for them to the DoFieldExchange or DoBulkFieldExchange member function, and initializations of the data members in the class

constructor. For more information, see Recordset: Dynamically Binding Data Columns (ODBC).

- Override recordset member functions, such as **OnSetOptions**, to set application-specific options or to override defaults.

If you want to base the recordset on a complex SQL statement, you need to use some combination of these customization techniques. For example, perhaps you want to use SQL clauses and keywords not directly supported by recordsets or perhaps you are joining multiple tables.

See Also

Concepts

Recordset (ODBC)
Recordset: How Recordsets Update Records (ODBC)
ODBC Basics
SQL
Recordset: Locking Records (ODBC)

Recordset: How Recordsets Update Records (ODBC)

This topic applies to the MFC ODBC classes.

Besides their ability to select records from a data source, recordsets can (optionally) update or delete the selected records or add new records. Three factors determine a recordset's updateability: whether the connected data source is updateable, the options you specify when you create a recordset object, and the SQL that is created.

Note
The SQL on which your **CRecordset** object is based can affect your recordset's updateability. For example, if your SQL contains a join or a **GROUP BY** clause, MFC sets the updateability to **FALSE**.

Note
This topic applies to objects derived from **CRecordset** in which bulk row fetching has not been implemented. If you are using bulk row fetching, see Recordset: Fetching Records in Bulk (ODBC).

This topic explains:

- Your role in recordset updating and what the framework does for you.
- The recordset as an edit buffer and the differences between dynasets and snapshots.

Recordset: How AddNew, Edit, and Delete Work (ODBC) describes the actions of these functions from the point of view of the recordset.

Recordset: More About Updates (ODBC) completes the recordset update story by explaining how transactions affect updates, how closing a recordset or scrolling affects updates in progress, and how your updates interact with the updates of other users.

Your Role in Recordset Updating

The following table shows your role in using recordsets to add, edit, or delete records, along with what the framework does for you.

Recordset Updating: You and the Framework

You	The framework
Determine whether the data source is updateable (or appendable).	Supplies CDatabase member functions for testing the data source's updateability or appendability.
Open an updatable recordset (of any type).	
Determine whether the recordset is updatable by calling **CRecordset** update functions such as **CanUpdate** or **CanAppend**.	
Call recordset member functions to add, edit, and delete records.	Manages the mechanics of exchanging data between your recordset object and the data source.
Optionally, use transactions to control the update process.	Supplies **CDatabase** member functions to support transactions.

For more information about transactions, see Transaction (ODBC).

The Edit Buffer

Taken collectively, the field data members of a recordset serve as an edit buffer that contains one record — the current record. Update operations use this buffer to operate on the current record.

- When you add a record, the edit buffer is used to build a new record. When you finish adding the record, the record that was previously current becomes current again.

- When you update (edit) a record, the edit buffer is used to set the field data members of the recordset to new values. When you finish updating, the updated record is still current.

When you call AddNew or Edit, the current record is stored so it can be restored later as needed. When you call Delete, the current record is not stored but is marked as deleted and you must scroll to another record.

Note
The edit buffer plays no role in record deletion. When you delete the current record, the record is marked as deleted, and the recordset is "not on a record" until you scroll to a different record.

Dynasets and Snapshots

Dynasets refresh a record's contents as you scroll to the record. Snapshots are static representations of the records, so a record's contents are not refreshed unless you call Requery. To use all the functionality of dynasets, you must be working with an ODBC driver that conforms to the correct level of ODBC API support. For more information, see ODBC and Dynaset.

See Also

Concepts

Recordset (ODBC)

Recordset: How AddNew, Edit, and Delete Work (ODBC)

Recordset: How AddNew, Edit, and Delete Work (ODBC)

This topic applies to the MFC ODBC classes.

This topic explains how the **AddNew**, **Edit**, and **Delete** member functions of class **CRecordset** work. Topics covered include:

- How Adding Records Works
- Visibility of Added Records
- How Editing Records Works
- How Deleting Records Works

Note
This topic applies to objects derived from **CRecordset** in which bulk row fetching has not been implemented. If you are usin g bulk row fetching, see Recordset: Fetching Records in Bulk (ODBC).

As a supplement, you might want to read Record Field Exchange: How RFX Works, which describes the corresponding role of RFX in update operations.

Adding a Record

Adding a new record to a recordset involves calling the recordset's AddNew member function, setting the values of the new record's field data members, and calling the Update member function to write the record to the data source.

As a precondition for calling **AddNew**, the recordset must not have been opened as read-only. The **CanUpdate** and **CanAppend** member functions let you determine these conditions.

When you call **AddNew**:

- The record in the edit buffer is stored, so its contents can be restored if the operation is canceled.
- The field data members are flagged so it is possible to detect changes in them later. The field data members are also marked clean (unchanged) and set to a Null.

After you call **AddNew**, the edit buffer represents a new, empty record, ready to be filled in with values. To do this, you manually set the values by assigning to them. Instead of specifying an actual data value for a field, you can call **SetFieldNull** to specify the value Null.

To commit your changes, you call **Update**. When you call **Update** for the new record:

- If your ODBC driver supports the **::SQLSetPos** ODBC API function, MFC uses the function to add the record on the data source. With **::SQLSetPos**, MFC can add a record more efficiently because it does not have to construct and process a SQL statement.
- If **::SQLSetPos** cannot be used, MFC does the following:
  1. If no changes are detected, **Update** does nothing and returns 0.
  2. If there are changes, **Update** constructs a SQL **INSERT** statement. The columns represented by all dirty field data members are listed in the **INSERT** statement. To force a column to be included, call the SetFieldDirty member function:

     ```
     SetFieldDirty( &m_dataMember, TRUE );
     ```

  3. **Update** commits the new record — the **INSERT** statement is executed and the record is committed to the table on the data source (and the recordset, if not a snapshot) unless a transaction is in progress.
  4. The stored record is restored to the edit buffer. The record that was current before the **AddNew** call is current again regardless of whether the **INSERT** statement was successfully executed.

Tip

For complete control of a new record, take the following approach: set the values of any fields that will have values and then explicitly set any fields that will remain Null by calling **SetFieldNull** with a pointer to the field and the parameter **TRUE** (the default). If you want to ensure that a field is not written to the data source, call **SetFieldDirty** with a pointer to the field and the parameter **FALSE**, and do not modify the field's value. To determine whether a field is allowed to be Null, call **IsFieldNullable**.

Tip

To detect when recordset data members change value, MFC uses a **PSEUDO_NULL** value appropriate to each data typ e that you can store in a recordset. If you must explicitly set a field to the **PSEUDO_NULL** value and the field happens a lready to be marked Null, you must also call **SetFieldNull**, passing the address of the field in the first parameter and **F ALSE** in the second parameter.

Visibility of Added Records

When is an added record visible to your recordset? Added records sometimes show up and sometimes are not visible, depending on two things:

- What your driver is capable of.
- What the framework can take advantage of.

If your ODBC driver supports the **::SQLSetPos** ODBC API function, MFC uses the function to add records. With **::SQLSetPos**, added records are visible to any updatable MFC recordset. Without support for the function, added records are not visible and you must call **Requery** to see them. Using **::SQLSetPos** is also more efficient.

Editing an Existing Record

Editing an existing record in a recordset involves scrolling to the record, calling the recordset's Edit member function, setting the values of the new record's field data members, and calling the Update member function to write the changed record to the data source.

As a precondition for calling **Edit**, the recordset must be updatable and on a record. The **CanUpdate** and **IsDeleted** member functions let you determine these conditions. The current record also must not already have been deleted, and there must be records in the recordset (both **IsBOF** and **IsEOF** return 0).

When you call **Edit**, the record in the edit buffer (the current record) is stored. The stored record's values are later used to detect whether any fields have changed.

After you call **Edit**, the edit buffer still represents the current record but is now ready to accept changes to the field data members. To change the record, you manually set the values of any field data members you want to edit. Instead of specifying an actual data value for a field, you can call **SetFieldNull** to specify the value Null. To commit your changes, call **Update**.

Tip

To get out of **AddNew** or **Edit** mode, call **Move** with the parameter **AFX_MOVE_REFRESH**.

As a precondition for calling **Update**, the recordset must not be empty and the current record must not have been deleted. **IsBOF**, **IsEOF**, and **IsDeleted** should all return 0.

When you call **Update** for the edited record:

- If your ODBC driver supports the **::SQLSetPos** ODBC API function, MFC uses the function to update the record on the data source. With **::SQLSetPos**, the driver compares your edit buffer with the corresponding record on the server, updating the record on the server if the two are different. With **::SQLSetPos**, MFC can update a record more efficiently because it does not have to construct and process a SQL statement.

  -or-

- If **::SQLSetPos** cannot be used, MFC does the following:

  1. If there have been no changes, **Update** does nothing and returns 0.
  2. If there are changes, **Update** constructs a SQL **UPDATE** statement. The columns listed in the **UPDATE** statement

are based on the field data members that have changed.

3. **Update** commits the changes — executes the **UPDATE** statement — and the record is changed on the data source, but not committed if a transaction is in progress (see Transaction: Performing a Transaction in a Recordset (ODBC) for information about how the transaction affects the update). ODBC keeps a copy of the record, which also changes.

4. Unlike the process for **AddNew**, the **Edit** process does not restore the stored record. The edited record remains in place as the current record.

Caution
When you prepare to update a recordset by calling **Update**, take care that your recordset includes all columns making up the primary key of the table (or all of the columns of any unique index on the table, or enough columns to uniquely identify the row). In some cases, the framework can use only the columns selected in your recordset to identify which record in your table to update. Without all the necessary columns, multiple records might be updated in the table. In this case, the framework throws exceptions when you call **Update**.

Tip
If you call **AddNew** or **Edit** after having called either function previously but before you call **Update**, the edit buffer is refreshed with the stored record, replacing the new or edited record in progress. This behavior gives you a way to abort an **AddNew** or **Edit** and begin a new one: if you determine that the record-in-progress is faulty, simply call **Edit** or **AddNew** again.

Deleting a Record

Deleting a record from a recordset involves scrolling to the record and calling the recordset's Delete member function. Unlike **AddNew** and **Edit**, **Delete** does not require a matching call to **Update**.

As a precondition for calling **Delete**, the recordset must be updatable and it must be on a record. The **CanUpdate**, **IsBOF**, **IsEOF**, and **IsDeleted** member functions let you determine these conditions.

When you call **Delete**:

- If your ODBC driver supports the **::SQLSetPos** ODBC API function, MFC uses the function to delete the record on the data source. Using **::SQLSetPos** is usually more efficient than using SQL.

  -or-

- If **::SQLSetPos** cannot be used, MFC does the following:

  1. The current record in the edit buffer is not backed up as in **AddNew** and **Edit**.

  2. **Delete** constructs a SQL **DELETE** statement that removes the record.

     The current record in the edit buffer is not stored as in **AddNew** and **Edit**.

  3. **Delete** commits the deletion — executes the **DELETE** statement. The record is marked deleted on the data source and, if the record is a snapshot, in ODBC.

  4. The deleted record's values are still in the field data members of the recordset, but the field data members are marked Null and the recordset's **IsDeleted** member function returns a nonzero value.

Note
After deleting a record, you should scroll to another record to refill the edit buffer with the new record's data. It is an error to call **Delete** again or to call **Edit**.

For information about the SQL statements used in update operations, see SQL.

See Also

Concepts

Recordset (ODBC)

Recordset: More About Updates (ODBC)

Recordset: More About Updates (ODBC)

This topic applies to the MFC ODBC classes.

This topic explains:

- How other operations, such as transactions, affect updates.
- Your updates and those of other users.
- More about the Update and Delete member functions.

Note
This topic applies to objects derived from **CRecordset** in which bulk row fetching has not been implemented. If you have implemented bulk row fetching, some of the information does not apply. For example, you cannot call the **AddNew**, **Edit**, **Delete**, and **Update** member functions; however, you can perform transactions. For more information about bulk row fetching, see Recordset: Fetching Records in Bulk (ODBC).

How Other Operations Affect Updates

Your updates are affected by transactions in effect at the time of the update, by closing the recordset before completing a transaction, and by scrolling before completing a transaction.

How Transactions Affect Updates

Beyond understanding how **AddNew**, **Edit**, and **Delete** work, it is important to understand how the **BeginTrans**, **CommitTrans**, and **Rollback** member functions of CDatabase work with the update functions of CRecordset.

By default, calls to **AddNew** and **Edit** affect the data source immediately when you call **Update**. **Delete** calls take effect immediately. But you can establish a transaction and execute a batch of such calls. The updates are not permanent until you commit them. If you change your mind, you can roll back the transaction instead of committing it.

For more information about transactions, see Transaction (ODBC).

How Closing the Recordset Affects Updates

If you close a recordset, or its associated **CDatabase** object, with a transaction in progress (you have not called CommitTrans or Rollback), the transaction is rolled back automatically (unless your database backend is the Microsoft Jet database engine).

Caution
If you are using the Microsoft Jet database engine, closing a recordset inside an explicit transaction does not result in releasing any of the rows that were modified or locks that were placed until the explicit transaction is committed or rolled back. It is recommended that you always both open and close recordsets inside or outside of an explicit Jet transaction.

How Scrolling Affects Updates

When you scroll in a recordset, the edit buffer is filled with each new current record (the previous record is not stored first). Scrolling skips over records previously deleted. If you scroll after an **AddNew** or **Edit** call without calling **Update**, **CommitTrans**, or **Rollback** first, any changes are lost (with no warning to you) as a new record is brought into the edit buffer. The edit buffer is filled with the record scrolled to, the stored record is freed, and no change occurs on the data source. This applies to both **AddNew** and **Edit**.

Your Updates and the Updates of Other Users

When you use a recordset to update data, your updates affect other users. Similarly, the updates of other users during the lifetime of your recordset affect you.

In a multiuser environment, other users can open recordsets that contain some of the same records you have selected in your recordset. Changes to a record before you retrieve it are reflected in your recordset. Because dynasets retrieve a record each time you scroll to it, dynasets reflect changes each time you scroll to a record. Snapshots retrieve a record the first time you scroll to it, so snapshots reflect only those changes that occur before you scroll to the record initially.

Records added by other users after you open the recordset do not show up in your recordset unless you requery. If your recordset is a dynaset, edits to existing records by other users do show up in your dynaset when you scroll to the affected

record. If your recordset is a snapshot, edits do not show up until you requery the snapshot. If you want to see records added or deleted by other users in your snapshot, or records added by other users in your dynaset, call Requery to rebuild the recordset. (Note that the deletions of other users show up in your dynaset.) You might also call **Requery** to see records you add, but not to see your deletions.

Tip
To force caching of an entire snapshot at once, call **MoveLast** immediately after you open the snapshot. Then call **MoveFirst** to begin working with the records. **MoveLast** is equivalent to scrolling over all the records, but it retrieves them all at once. Note, however, that this can lower performance and might not be required for some drivers.

The effects of your updates on other users are similar to their effects on you.

More About Update and Delete

This section provides additional information to help you work with **Update** and **Delete**.

Update Success and Failure

If **Update** succeeds, the **AddNew** or **Edit** mode ends. To begin an **AddNew** or **Edit** mode again, call **AddNew** or **Edit**.

If **Update** fails (returns **FALSE** or throws an exception), you remain in **AddNew** or **Edit** mode, depending on which function you called last. You can then do one of the following:

- Modify a field data member and try the **Update** again.
- Call **AddNew** to reset the field data members to Null, set the values of the field data members, and then call **Update** again.
- Call **Edit** to reload the values that were in the recordset before the first call to **AddNew** or **Edit**, set the values of the field data members, and then call **Update** again. After a successful **Update** call (except after an **AddNew** call), the field data members retain their new values.
- Call **Move** (including **Move** with a parameter of **AFX_MOVE_REFRESH**, or 0), which flushes any changes and ends any **AddNew** or **Edit** mode in effect.

Update and Delete

This section applies to both **Update** and **Delete**.

On an **Update** or **Delete** operation, one and only one record should be updated. That record is the current record, which corresponds to the data values in the fields of the recordset. If for some reason no records are affected or more than one record is affected, an exception is thrown containing one of the following **RETCODE** values:

- **AFX_SQL_ERROR_NO_ROWS_AFFECTED**
- **AFX_SQL_ERROR_MULTIPLE_ROWS_AFFECTED**

When these exceptions are thrown, you remain in the **AddNew** or **Edit** state you were in when you called **Update** or **Delete**. Here are the most common scenarios in which you would see these exceptions. You're most likely to see:

- **AFX_SQL_ERROR_NO_ROWS_AFFECTED** when you are using optimistic locking mode and another user has modified the record in a way that prevents the framework from identifying the correct record to update or delete.
- **AFX_SQL_ERROR_MULTIPLE_ROWS_AFFECTED** when the table you are updating has no primary key or unique index and you do not have enough columns in the recordset to uniquely identify a table row.

See Also

Concepts

Recordset (ODBC)
Recordset: How Recordsets Select Records (ODBC)
Record Field Exchange (RFX)
SQL
Exceptions: Database Exceptions

SQL

Structured Query Language (SQL) is a way to communicate with a relational database that lets you define, query, modify, and control the data. Using SQL syntax, you can construct a statement that extracts records according to criteria you specify.

Note
This information applies to the MFC ODBC classes. If you are working with the MFC DAO classes, see the topic Comparison o f Microsoft Jet Database Engine SQL and ANSI SQL in DAO Help.

SQL statements begin with a keyword verb such as **CREATE** or **SELECT**. SQL is a very powerful language; a single statement can affect an entire table.

Many versions of SQL exist, each developed with a particular DBMS in mind. The MFC database classes recognize a set of SQL statements that corresponds to the X/Open and SQL Access Group Common Applications Environment (CAE) SQL draft specification (1991). For information about the syntax of these statements, see Appendix C in the *ODBC SDK Programmer's Reference* on the MSDN Library CD.

This topic explains:

- The relationship between ODBC and SQL.
- The most common SQL keywords used by the database classes.
- How the database classes use SQL.

Open Database Connectivity (ODBC)

The database classes are implemented with ODBC, which uses SQL in a call-level interface rather than embedding SQL commands in the code. ODBC uses SQL to communicate with a data source through ODBC drivers. These drivers interpret the SQL and translate it, if necessary, for use with a particular database format, such as Microsoft Access. For more information about how ODBC uses SQL, see ODBC and the ODBC SDK *Programmer's Reference* on the MSDN Library CD.

Database Classes

The database classes are designed to let you manipulate and update data in an existing data source. The MFC Application Wizard, the MFC ODBC Consumer Wizard (accessed through **Add Class**), and the database classes construct most of the SQL statements for you.

The database classes use a portion of SQL known as the Data Manipulation Language (DML). These commands let you work with all or part of the data source, add new records, edit records, and delete records. The following table lists the most common SQL keywords and the ways the database classes use them.

Some Common SQL Keywords

SQL keyword	The wizards and database classes use it
SELECT	To identify which tables and columns in the data source are to be used.
WHERE	To apply a filter that narrows the selection.
ORDER BY	To apply a sort order to the recordset.
INSERT	To add new records to a recordset.
DELETE	To delete records from a recordset.
UPDATE	To modify the fields of a record.

In addition, the database classes recognize ODBC **CALL** statements, which you can use to call a predefined query (or stored procedure) on some data sources. The ODBC database driver interprets these statements and substitutes the command appropriate for each DBMS.

Note
Not all DBMSs support **CALL** statements.

If the classes cannot recognize a user-supplied statement in **CRecordset::Open**, it is interpreted as a table name.

For an explanation of how the framework constructs SQL statements, see Recordset: How Recordsets Select Records (ODBC) and SQL: Customizing Your Recordset's SQL Statement (ODBC).

SQL databases use data types similar to those used in C and C++. For a discussion of these similarities, see SQL: SQL and C++ Data Types (ODBC).

You can find more information about SQL, including a list of supported SQL statements, data types, SQL core grammar, and a reading list of recommended publications about SQL, in the *ODBC SDK Programmer's Reference* on the MSDN Library CD.

How the Database Classes Use SQL

The recordsets you derive from the database classes use ODBC to communicate with a data source, and ODBC retrieves records from the data source by sending SQL statements. This topic explains the relationship between the database classes and SQL.

A recordset constructs a SQL statement by building up the pieces of a SQL statement into a **CString**. The string is constructed as a **SELECT** statement, which returns a set of records.

When the recordset calls ODBC to send a SQL statement to the data source, the ODBC Driver Manager passes the statement to the ODBC driver and the driver sends it to the underlying DBMS. The DBMS returns a result set of records, and the ODBC driver returns the records to the application. The database classes let your program access the result set in a type-safe C++ class derived from **CRecordset**.

The following topics provide more information about how the database classes use SQL:

- SQL: Customizing Your Recordset's SQL Statement (ODBC)
- SQL: SQL and C++ Data Types (ODBC)
- SQL: Making Direct SQL Calls (ODBC)

See Also

Concepts

Open Database Connectivity (ODBC)
ODBC Basics

SQL: Customizing Your Recordset's SQL Statement (ODBC)

This topic explains:

- How the framework constructs a SQL statement
- How to override the SQL statement

Note

This information applies to the MFC ODBC classes. If you are working with the MFC DAO classes, see the topic "Comparison of Microsoft Jet Database Engine SQL and ANSI SQL" in DAO Help.

SQL Statement Construction

Your recordset bases record selection primarily on a SQL **SELECT** statement. When you declare your class with a wizard, it writes an overriding version of the **GetDefaultSQL** member function that looks something like this (for a recordset class called `CAuthors`).

```
CString CAuthors::GetDefaultSQL()
{
    return "AUTHORS";
}
```

By default, this override returns the table name you specified with the wizard. In the example, the table name is "AUTHORS." When you later call the recordset's **Open** member function, **Open** constructs a final **SELECT** statement of the form:

```
SELECT rfx-field-list FROM table-name [WHERE m_strFilter]
       [ORDER BY m_strSort]
```

where `table-name` is obtained by calling **GetDefaultSQL** and `rfx-field-list` is obtained from the RFX function calls in **DoFieldExchange**. This is what you get for a **SELECT** statement unless you replace it with an overriding version at run time, although you can also modify the default statement with parameters or a filter.

Note

If you specify a column name that contains (or could contain) spaces, you must enclose the name in square brackets. For exa mple, the name "First Name" should be "[First Name]".

To override the default **SELECT** statement, pass a string containing a complete **SELECT** statement when you call **Open**. Instead of constructing its own default string, the recordset uses the string you supply. If your replacement statement contains a **WHERE** clause, do not specify a filter in **m_strFilter** because you would then have two filter statements. Similarly, if your replacement statement contains an **ORDER BY** clause, do not specify a sort in **m_strSort** so that you will not have two sort statements.

Note

If you use literal strings in your filters (or other parts of the SQL statement), you might have to "quote" (enclose in specified d elimiters) such strings with a DBMS-specific literal prefix and literal suffix character (or characters).

You might also encounter special syntactic requirements for operations such as outer joins, depending on your DBMS. Use ODBC functions to obtain this information from your driver for the DBMS. For example, call **::SQLGetTypeInfo** for a particular data type, such as **SQL_VARCHAR**, to request the **LITERAL_PREFIX** and **LITERAL_SUFFIX** characters. If you are writing database-independent code, see Appendix C in the *ODBC SDK Programmer's Reference* on the MSDN Library CD for detailed syntax information.

A recordset object constructs the SQL statement that it uses to select records unless you pass a custom SQL statement. How this is done depends mainly on the value you pass in the *lpszSQL* parameter of the **Open** member function.

The general form of a SQL **SELECT** statement is:

```
SELECT [ALL | DISTINCT] column-list FROM table-list
    [WHERE search-condition][ORDER BY column-list [ASC | DESC]]
```

One way to add the **DISTINCT** keyword to your recordset's SQL statement is to embed the keyword in the first RFX function call in **DoFieldExchange**. For example:

```
...
    RFX_Text(pFX, "DISTINCT CourseID", m_strCourseID);
...
```

Note
Use this technique only with a recordset opened as read-only.

Overriding the SQL Statement

The following table shows the possibilities for the *lpszSQL* parameter to **Open**. The cases in the table are explained following the table.

The lpszSQL Parameter and the Resulting SQL String

Case	What you pass in lpszSQL	The resulting SELECT statement
1	**NULL**	**SELECT** *rfx-field-list* **FROM** *table-name* **CRecordset::Open** calls **GetDefaultSQL** to get the table name. The resulting string is one of cases 2 through 5, depending on what **GetDefaultSQL** returns.
2	A table name	**SELECT** *rfx-field-list* **FROM** *table-name* The field list is taken from the RFX statements in **DoFieldExchange**. If **m_strFilter** and **m_strSort** are not empty, adds the **WHERE** and/or **ORDER BY** clauses.
3 *	A complete **SELECT** statement but without a **WHERE** or **ORDER BY** clause	As passed. If **m_strFilter** and **m_strSort** are not empty, adds the **WHERE** and/or **ORDER BY** clauses.
4 *	A complete **SELECT** statement with a **WHERE** and/or **ORDER BY** clause	As passed. **m_strFilter** and/or **m_strSort** must remain empty, or two filter and/or sort statements are produced.
5 *	A call to a stored procedure	As passed.

* **m_nFields** must be less than or equal to the number of columns specified in the **SELECT** statement. The data type of each column specified in the **SELECT** statement must be the same as the data type of the corresponding RFX output column.

Case 1 lpszSQL = NULL

The recordset selection depends on what **GetDefaultSQL** returns when **CRecordset::Open** calls it. Cases 2 through 5 describe the possible strings.

Case 2 lpszSQL = a Table Name

The recordset uses record field exchange (RFX) to build the column list from the column names provided in the RFX function calls in the recordset class's override of **DoFieldExchange**. If you used a wizard to declare your recordset class, this case has the same result as case 1 (provided that you pass the same table name you specified in the wizard). If you do not use a wizard to write your class, case 2 is the simplest way to construct the SQL statement.

The following example constructs a SQL statement that selects records from an MFC database application. When the framework calls the **GetDefaultSQL** member function, the function returns the name of the table, `SECTION`.

```
CString CEnrollSet::GetDefaultSQL()
```

```
{
    return "SECTION";
}
```

To obtain the names of the columns for the SQL **SELECT** statement, the framework calls the **DoFieldExchange** member function.

```
void CEnrollSet::DoFieldExchange(CFieldExchange* pFX)
{
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Text(pFX, "CourseID", m_strCourseID);
    RFX_Text(pFX, "InstructorID", m_strInstructorID);
    RFX_Text(pFX, "RoomNo", m_strRoomNo);
    RFX_Text(pFX, "Schedule", m_strSchedule);
    RFX_Text(pFX, "SectionNo", m_strSectionNo);
}
```

When complete, the SQL statement looks like this:

```
SELECT CourseID, InstructorID, RoomNo, Schedule, SectionNo
    FROM SECTION
```

Case 3 lpszSQL = a SELECT/FROM Statement

You specify the column list by hand rather than relying on RFX to construct it automatically. You might want to do this when:

- You want to specify the **DISTINCT** keyword following **SELECT**.

  Your column list should match the column names and types in the same order as they are listed in **DoFieldExchange**.

- You have reason to manually retrieve column values using the ODBC function **::SQLGetData** rather than relying on RFX to bind and retrieve columns for you.

  You might, for example, want to accommodate new columns a customer of your application added to the database tables after the application was distributed. You need to add these extra field data members, which were not known at the time you declared the class with a wizard.

  Your column list should match the column names and types in the same order as they are listed in **DoFieldExchange**, followed by the names of the manually bound columns. For more information, see Recordset: Dynamically Binding Data Columns (ODBC).

- You want to join tables by specifying multiple tables in the **FROM** clause.

  For information and an example, see Recordset: Performing a Join (ODBC).

Case 4 lpszSQL = SELECT/FROM Plus WHERE and/or ORDER BY

You specify everything: the column list (based on the RFX calls in **DoFieldExchange**), the table list, and the contents of a **WHERE** and/or an **ORDER BY** clause. If you specify your **WHERE** and/or **ORDER BY** clauses this way, do not use **m_strFilter** and/or **m_strSort**.

Case 5 lpszSQL = a Stored Procedure Call

If you need to call a predefined query (such as a stored procedure in a Microsoft SQL Server database), you must write a **CALL** statement in the string you pass to *lpszSQL*. The wizards do not support declaring a recordset class for calling a predefined query. Not all predefined queries return records.

If a predefined query does not return records, you can use the **CDatabase** member function **ExecuteSQL** directly. For a predefined query that does return records, you must also manually write the RFX calls in **DoFieldExchange** for any columns the procedure returns. The RFX calls must be in the same order and return the same types, as the predefined query. For more information, see Recordset: Declaring a Class for a Predefined Query (ODBC).

See Also

Concepts

SQL: SQL and C++ Data Types (ODBC)
SQL: Making Direct SQL Calls (ODBC)

SQL: SQL and C++ Data Types (ODBC)

Note
This information applies to the MFC ODBC classes. If you are working with the MFC DAO classes, see the topic "Comparison of Microsoft Jet Database Engine SQL and ANSI SQL" in DAO Help.

The following table maps ANSI SQL data types to C++ data types. This augments the C language information given in Appendix D of the *ODBC SDK Programmer's Reference* on the MSDN Library CD. The wizards manage most data type mapping for you. If you do not use a wizard, you can use the mapping information to help you write the field exchange code manually.

ANSI SQL Data Types Mapped to C++ Data Types

ANSI SQL data type	C++ data type
CHAR	**CString**
DECIMAL	**CString** 1
SMALLINT	**int**
REAL	**float**
INTEGER	**long**
FLOAT	**double**
DOUBLE	**double**
NUMERIC	**CString** 1
VARCHAR	**CString**
LONGVARCHAR	**CLongBinary**, **CString** 2
BIT	**BOOL**
TINYINT	**BYTE**
BIGINT	**CString** 1
BINARY	**CByteArray**
VARBINARY	**CByteArray**
LONGVARBINARY	**CLongBinary**, **CByteArray** 3
DATE	**CTime**, **CString**
TIME	**CTime, CString**
TIMESTAMP	**CTime, CString**

1. ANSI **DECIMAL** and **NUMERIC** map to **CString** because **SQL_C_CHAR** is the default ODBC transfer type.

2. Character data beyond 255 characters is truncated by default when mapped to **CString**. You can extend the truncation length by explicitly setting the *nMaxLength* argument of **RFX_Text**.

3. Binary data beyond 255 characters is truncated by default when mapped to **CByteArray**. You can extend the truncation length by explicitly setting the *nMaxLength* argument of **RFX_Binary**.

If you are not using the ODBC cursor library, you might encounter a problem when attempting to update two or more long variable-length fields using the Microsoft SQL Server ODBC driver and the MFC ODBC database classes. The ODBC types, **SQL_LONGVARCHAR** and **SQL_LONGVARBINARY**, map to text and image SQL Server types. A **CDBException** is thrown if you update two or more long variable-length fields on the same call to **CRecordset::Update**. Therefore, do not update multiple long columns simultaneously with **CRecordset::Update**. You can update multiple long columns simultaneously with the ODBC API **SQLPutData**. You can also use the ODBC cursor library, but this is not recommended for drivers, like the SQL Server driver, that support cursors and do not need the cursor library.

If you are using the ODBC cursor library with the MFC ODBC database classes and the Microsoft SQL Server ODBC driver, an **ASSERT** might occur along with a **CDBException** if a call to **CRecordset::Update** follows a call to **CRecordset::Requery**. Instead, call **CRecordset::Close** and **CRecordset::Open** rather than **CRecordset::Requery**. Another solution is not to use the ODBC cursor library, because the SQL Server and the SQL Server ODBC driver provide native support for cursors natively and the ODBC cursor library is not needed.

See Also

Concepts

SQL

SQL: Making Direct SQL Calls (ODBC)

SQL: Making Direct SQL Calls (ODBC)

This topic explains:

- When to use direct SQL calls.
- How you make direct SQL calls to the data source.

Note
This information applies to the MFC ODBC classes. If you are working with the MFC DAO classes, see the topic "Comparison of Microsoft Jet Database Engine SQL and ANSI SQL" in DAO Help.

When to Call SQL Directly

To create new tables, drop (delete) tables, alter existing tables, create indexes, and perform other SQL functions that change the data source schema, you must issue a SQL statement directly to the data source using Database Definition Language (DDL). When you use a wizard to create a recordset for a table (at design time), you can choose which columns of the table to represent in the recordset. This does not allow for columns you or another user of the data source add to the table later, after your program has been compiled. The database classes do not support DDL directly, but you can still write code to bind a new column to your recordset dynamically, at run time. For information about how to do this binding, see Recordset: Dynamically Binding Data Columns (ODBC).

You can use the DBMS itself to alter the schema or another tool that lets you perform DDL functions. You can also use ODBC function calls for sending SQL statements, such as calling a predefined query (stored procedure) that does not return records.

Making Direct SQL Function Calls

You can directly execute a SQL call using a CDatabase object. Set up your SQL statement string (usually in a **CString**) and pass it to the ExecuteSQL member function of your **CDatabase** object. If you use ODBC function calls to send a SQL statement that normally returns records, the records are ignored.

See Also

Concepts

SQL

Transaction (ODBC)

This topic applies to the MFC ODBC classes.

A transaction is a way to group, or batch, a series of updates to a data source so that all are committed at once or none are committed if you roll back the transaction. If you do not use a transaction, changes to the data source are committed automatically rather than being committed on demand.

Note
Not all ODBC database drivers support transactions. Call the **CanTransact** member function of your CDatabase or CRecordset object to determine whether your driver supports transactions for a given database. Note that **CanTransact** does not tell you whether the data source provides full transaction support. You must also call **CDatabase::GetCursorCommitBehavior** and **CDatabase::GetCursorRollbackBehavior** after **CommitTrans** and **Rollback** to check the effect of the transaction on the open **CRecordset** object.

Calls to the **AddNew** and **Edit** member functions of a **CRecordset** object affect the data source immediately when you call **Update**. **Delete** calls also take effect immediately. In contrast, you can use a transaction consisting of multiple calls to **AddNew**, **Edit**, **Update**, and **Delete**, which are performed but not committed until you call **CommitTrans** explicitly. By establishing a transaction, you can execute a series of such calls while retaining the ability to roll them back. If a critical resource is unavailable or some other condition prevents the entire transaction from being completed, you can roll back the transaction instead of committing it. In that case, none of the changes belonging to the transaction affect the data source.

Note
Currently, class **CRecordset** does not support updates to the data source if you have implemented bulk row fetching. This means you cannot make calls to **AddNew**, **Edit**, **Delete**, or **Update**. However, you can write you own functions to perform updates and then call those functions within a given transaction. For more information about bulk row fetching, see Recordset: Fetching Records in Bulk (ODBC).

Note
Besides affecting your recordset, transactions affect SQL statements that you execute directly as long as you use the ODBC **HDBC** associated with your **CDatabase** object or an ODBC **HSTMT** based on that **HDBC**.

Transactions are particularly useful when you have multiple records that must be updated simultaneously. In this case, you want to avoid a half-completed transaction, such as might happen if an exception was thrown before the last update was made. Grouping such updates into a transaction allows a recovery (rollback) from the changes and returns the records to the pretransaction state. For example, if a bank transfers money from account A to account B, both the withdrawal from A and the deposit to B must succeed to process the funds correctly or the whole transaction must fail.

In the database classes, you perform transactions through **CDatabase** objects. A **CDatabase** object represents a connection to a data source, and one or more recordsets associated with that **CDatabase** object operate on tables of the database through recordset member functions.

Note
Only one level of transactions is supported. You cannot nest transactions nor can a transaction span multiple database objects.

The following topics provide more information about how transactions are performed:

- Transaction: Performing a Transaction in a Recordset (ODBC)
- Transaction: How Transactions Affect Updates (ODBC)

See Also

Concepts

Open Database Connectivity (ODBC)

Transaction: Performing a Transaction in a Recordset (ODBC)

This topic explains how to perform a transaction in a recordset.

Note
Only one level of transactions is supported; you cannot nest transactions.

To perform a transaction in a recordset

1. Call the **CDatabase** object's **BeginTrans** member function.
2. If you have not implemented bulk row fetching, call the **AddNew/Update**, **Edit/Update**, and **Delete** member functions of one or more recordset objects of the same database as many times as needed. For more information, see Recordset: Adding, Updating, and Deleting Records (ODBC). If you have implemented bulk row fetching, you must write your own functions to update the data source.
3. Finally, call the **CDatabase** object's **CommitTrans** member function. If an error occurs in one of the updates or you decide to cancel the changes, call its **Rollback** member function.

The following example uses two recordsets to delete a student's enrollment from a school registration database, removing the student from all classes in which the student is enrolled. Because the **Delete** calls in both recordsets must succeed, a transaction is required. The example assumes the existence of `m_dbStudentReg`, a member variable of type **CDatabase** already connected to the data source, and the recordset classes `CEnrollmentSet` and `CStudentSet`. The `strStudentID` variable contains a value obtained from the user.

```
BOOL CEnrollDoc::RemoveStudent( CString strStudentID )
{
    // remove student from all the classes
    // the student is enrolled in

    if ( !m_dbStudentReg.BeginTrans( ) )
        return FALSE;

    CEnrollmentSet rsEnrollmentSet(&m_dbStudentReg);
    rsEnrollmentSet.m_strFilter = "StudentID = " + strStudentID;

    if ( !rsEnrollmentSet.Open(CRecordset::dynaset) )
        return FALSE;

    CStudentSet rsStudentSet(&m_dbStudentReg);
    rsStudentSet.m_strFilter = "StudentID = " + strStudentID;

    if ( !rsStudentSet.Open(CRecordset::dynaset) )
        return FALSE;

    TRY
    {
        while ( !rsEnrollmentSet.IsEOF( ) )
        {
            rsEnrollmentSet.Delete( );
            rsEnrollmentSet.MoveNext( );
        }

        // delete the student record
        rsStudentSet.Delete( );

        m_dbStudentReg.CommitTrans( );
    }

    CATCH_ALL(e)
    {
        m_dbStudentReg.Rollback( );
```

```
        return FALSE;
    }
    END_CATCH_ALL

    rsEnrollmentSet.Close( );
    rsStudentSet.Close( );

    return TRUE;

}
```

Note

Calling **BeginTrans** again without calling **CommitTrans** or **Rollback** is an error.

See Also

Reference

CDatabase Class

CRecordset Class

Concepts

Transaction (ODBC)

Transaction: How Transactions Affect Updates (ODBC)

Transaction: How Transactions Affect Updates (ODBC)

Updates to the data source are managed during transactions through the use of an edit buffer (the same method used outside of transactions). The field data members of a recordset collectively serve as an edit buffer that contains the current record, which the recordset backs up temporarily during an **AddNew** or **Edit**. During a **Delete** operation, the current record is not backed up within a transaction. For more information about the edit buffer and how updates store the current record, see Recordset: How Recordsets Update Records (ODBC).

Note
If you have implemented bulk row fetching, you cannot call **AddNew**, **Edit**, or **Delete**. You must instead write your own functions to perform updates to the data source. For more information about bulk row fetching, see Recordset: Fetching Records in Bulk (ODBC).

During transactions, **AddNew**, **Edit**, and **Delete** operations can be committed or rolled back. The effects of **CommitTrans** and **Rollback** might cause the current record to not be restored to the edit buffer. To make sure that the current record is properly restored, it is important to understand how the **CommitTrans** and **Rollback** member functions of **CDatabase** work with the update functions of **CRecordset**.

How CommitTrans Affects Updates

The following table explains the effects of **CommitTrans** on transactions.

How CommitTrans Affects Updates

Operation	Status of data source
AddNew and **Update**, and then **CommitTrans**	New record are added to data source.
AddNew (without **Update**), and then **CommitTrans**	New record is lost. Record not added to data source.
Edit and **Update**, and then **CommitTrans**	Edits committed to data source.
Edit (without **Update**), and then **CommitTrans**	Edits to the record are lost. Record remains unchanged on the data source.
Delete then **CommitTrans**	Records deleted from data source.

How Rollback Affects Transactions

The following table explains the effects of **Rollback** on transactions.

How Rollback Affects Transactions

Operation	Status of current record	You must also	Status of data source
AddNew and **Update**, then **Rollback**	Content of the current record is stored temporarily to make room for new record. New record is entered into edit buffer. After **Update** is called, the current record is restored to the edit buffer.		Addition to data source made by **Update** is reversed.
AddNew (without **Update**), then **Rollback**	Content of the current record is stored temporarily to make room for new record. Edit buffer contains new record.	Call **AddNew** again to restore the edit buffer to an empty, new record. Or call **Move**(0) to restore the old values to the edit buffer.	Because **Update** was not called, there were no changes made to the data source.

Edit and **Update**, then **Rollback**	An unedited version of the current record is stored temporarily. Edits are made to the content of the edit buffer. After **Update** is called, the unedited version of the record is still temporarily stored.	*Dynaset*: Scroll off the current record then back to restore the unedited version of the record to the edit buffer. *Snapshot*: Call **Requery** to refresh the recordset from the data source.	Changes to data source made by **Update** are reversed.
Edit (without **Update**), then **Rollback**	An unedited version of the current record is stored temporarily. Edits are made to the content of the edit buffer.	Call **Edit** again to restore the unedited version of the record to the edit buffer.	Because **Update** was not called, there were no changes made to the data source.
Delete then **Rollback**	Content of the current record is deleted.	Call **Requery** to restore the content of the current record from the data source.	Deletion of data from data source is reversed.

See Also

Reference

CDatabase Class

CRecordset Class

Concepts

Transaction (ODBC)

Transaction (ODBC)

Transaction: Performing a Transaction in a Recordset (ODBC)

Data-Bound Controls (ADO and RDO)

Databinding is a mechanism that allows users to work with data retrieved from a database.

Visual C++ provides a reusable databinding mechanism through ActiveX controls in MFC projects that lets you rapidly develop database applications.

This section is composed of the following sections about how to implement databinding in your Visual C++ application:

- Using ActiveX Controls
- Databinding with ActiveX Controls in Visual C++
- Creating Database Connections

For more information about data access programming:

To learn more about	See
Data access programming overview (Visual C++)	Data Access Programming
OLE DB programming (conceptual topics)	OLE DB Programming Overview
Open Database Connectivity (ODBC)	ODBC and MFC
Microsoft Universal Data Access Web site	http://www.microsoft.com/data

See Also

Concepts

Data Access Programming

Using ActiveX Controls

The topics in this section present an overview of using ActiveX controls.

An ActiveX control is a COM component that supports standard interfaces relating to persistence, connection points, and hosting. These standard interfaces define a protocol by which a control can be hosted in a control container, exchange messages, and handle events.

As COM servers, ActiveX controls have the following.

Term	Description
Properties	Controls have member variables to represent internal state and are implemented as **Get** and **Set** accessor functions. A **Get** function is generated for each accessor method with a propget tag in the .idl file. A **Set** function is generated for each accessor method with either a propput or propputref IDL tag. Use wrapper classes or the OLE/COM Object Viewer to determine how accessor functions are defined.
Methods	A control's behavior is defined by its public methods. Wrapper classes give you access to a control's methods. If you do not use wrapper classes (the default), you get access to a control's methods by obtaining a pointer to an interface. An example of a public method is the **Refresh** method in the ADO data control, which updates the retrieved rowset.
Events	A control can generate an event to notify the host that something happened. An example is the **OnClick** event for the Button control. When the button gets clicked, the button generates an **OnClick** event. If the control's host has a handler for that event, it executes.
Type Library	A type library tells a control container what properties, methods, and events are supported by a control. Type libraries can exist either as separate files (with a .tlb extension) or internally within the control. Type libraries also contain the control's coclass information. A coclass is a COM class that is identified with a GUID. A coclass contains one or more interfaces that are defined by the control. To examine type libraries, use the OLE/COM Object Viewer.

The following topics describe the use of an ActiveX control:

- Inserting the Control into a Visual C++ Application
- Wrapper Classes
- Creating a Database Connection
- Setting Control Properties at Design Time
- Setting Event Handlers on ActiveX Controls
- Modifying a Control's Run-Time Behavior
- Redistributing Controls

See Also

Reference

Data-Bound Controls (ADO and RDO)

Inserting the Control into a Visual C++ Application

You can insert an ActiveX control into a dialog box from the toolbar or using the Insert ActiveX Control dialog box.

To insert an ActiveX control from the Toolbox

1. Right-click an empty section of the Toolbox.
2. On the shortcut menu, click **Customize Toolbox**, and then select the controls that you want.
3. Drag controls to your dialog box in the Dialog editor.

To insert an ActiveX control from the Dialog editor

1. Right-click the dialog box.
2. On the shortcut menu, click Insert ActiveX Control.

Note
When you insert an ActiveX control from **Insert ActiveX Control** into a project, wrapper classes are not included in the project. You are responsible for creating a wrapper class to customize the control's functionality.

See Also

Reference

Using ActiveX Controls

Wrapper Classes

When you insert a control into a Visual C++ project, wrapper classes for the control are not included by default. However, if you want to modify the control's behavior, you can write a wrapper class for the control. Depending on how you intend to manipulate the control programmatically, you need to write one or more of the control's wrapper classes.

A wrapper class is available for each of the coclasses in the control's type library (.tlb) file. The control's wrapper class should be the control name prefixed by the letter C.

For more information about the functionality of the wrapper classes, see the object model for the control's base technology.

Using CWnd::GetDlgItem also requires wrapper classes because the return value must be cast to the control class. For example:

```
CDBList* pDBList = 0;
pDBList = static_cast<CDBList*>(GetDlgItem(IDC_DBLIST));
```

By reading the generated .idl file, you can determine what properties, methods, and events are exposed by a control, as well as see method and accessor function declarations directly. Additional information can be obtained from the control using the OLE/COM Object Viewer.

See Also

Reference

Using ActiveX Controls

Modifying a Control's Run-Time Behavior

Using the OLE/COM Object Viewer

By reading a control's type library, the OLE/COM Object Viewer lets you view a control's interfaces.

To use the OLE/COM Object Viewer

1. Start the OLE/COM Object Viewer by either clicking **OLE/COM Object Viewer** on the **Tools** menu or by typing **oleview** at the command line.
2. Display all registered Automation objects by opening the **Automation Objects** folder from the **Object Classes, Grouped by Component Category**.
3. Scroll down and select one of the control. Several tabs appear in the right pane and the interfaces implemented by the control display on the **Registry** tab.
   - If you right-click a control in the left pane and select **View Type Information**, the ITypeInfo Viewer displays a reconstructed .idl or .odl file.
   - If you expand the control node in the left pane, you see a list of the interfaces in the object. If you click an interface, the registry entry for that interface is displayed in the right pane.
   - If you right-click an interface and click **View**, the OLE/COM Object Viewer displays a dialog box showing the GUID for the interface and an option to view type library information, if it is available. Selecting **View Type Info** displays a portion of a reconstructed .idl file specific to the interface in the ITypeInfo Viewer.
   - In the ITypeInfo Viewer, expanding the tree view and clicking an interface member displays the accessor method signatures in the right pane.

See Also

Reference

Using ActiveX Controls

Setting Control Properties at Design Time

Properties of controls can be set at design time using the Dialog editor. When you set a property, the Resource editor initializes the control with the specified value. The property still can be changed programmatically.

The **DataSource** property, which is found in all data-bound controls, allows you to specify the data source control to which you want to bind.

When binding simple bound ADO data-bound controls to an ADO data control (ADODC), you must associate the control with a column by setting the **DataField** property to a valid field in the rowset. Otherwise, the compiled application asserts in a running Debug build (the assertion is simply marking that the control has been bound to a null column).

Note
The **General** properties tab allows you to specify a control identifier and other properties needed for the .rc file. (The .rc file is later compiled to generate a Windows program resource code.)

To set a property on the All tab

1. Insert an ActiveX control into a dialog box.
2. Right-click the control in the Dialog editor, and then click **Properties**.
3. Click the **All** tab to display the control's properties. For a given property, enter the initialization value.

To set control properties at run time, see Modifying a Control's Run-Time Behavior.

See Also

Reference

Using ActiveX Controls

Setting Event Handlers on ActiveX Controls

ActiveX controls can be programmed to respond to events. You can use **ControlEvents** in properties to view the events that are available in a control and to create event handlers. Event handling is commonly done to trap changes in the data source query. Changes include:

- Implementing a lookup. When a control (like a DBCombo box) changes value, the change event is trapped to update a data control's query.
- Implementing a master detail relationship. Two data controls are added to a dialog box, one for the master portion and a second for the detail portion. Whenever the first data source changes, the second data source's query is updated through an event handler.
- Trapping errors. Most controls have an error event from which you can write an error handler (see Error Trapping).

For more information, see Mapping Messages to Functions.

See Also
Reference
Using ActiveX Controls

Modifying a Control's Run-Time Behavior

After you insert a control and generate one or more wrapper classes, you can invoke the control's methods and program the control's event handlers.

The control's wrapper classes specify the functions you can use to modify the run-time behavior of the control. Include the appropriate wrapper class header file and use the methods. To set a property, look for an accessor method with the property name prefixed by Set. To retrieve a property, look for an accessor method with the property name prefixed by Get. Event handlers can be written later.

Because the controls are implemented using Automation, the types passed can only be Automation-safe types such as BSTR and VARIANT. While you can use system calls to allocate and set BSTRs and VARIANTs, you might want to use the ATL wrapper classes (CComBSTR, CComVariant), the Visual C++ COM Compiler support wrapper classes (_bstr_t, _variant_t), or the MFC wrapper class (COleVariant).

If you add a data control, the Insert ActiveX Control Wizard generates wrapper classes for the data control's coclasses that manage its internal data objects. These classes do not include all of RDO or ADO, but rather represent internal objects declared in the type library.

If you want to use ADO and RDO directly, you should connect to the ADO or RDO DLLs directly (Msado15.dll or Msrdo20.dll), either with the compiler COM support classes, which support the #import directive, or with the respective SDK.

To Set Control Properties at Run Time

Note that some properties of an ActiveX control might be read-only at run time, which makes dynamic creation difficult. You can temporarily simulate design mode for property initialization by overriding the control container's OnAmbientPropertyChange handler, as described in the Knowledge Base article, "How to: Set ActiveX Control Design-Time Properties at Run Time (Q260744)". You can find Knowledge Base articles at http://support.microsoft.com/support.

See Also

Reference

Using ActiveX Controls

Redistributing Controls

Visual C++ .NET supplies ActiveX controls that you can use in applications. You can then redistribute the controls along with the applications. In the **Insert ActiveX Control** dialog box, highlighting a control will display its .ocx or .dll file.

For a list of the redistributable Visual C++–supplied ActiveX controls, see Program Files\Microsoft Visual Studio .NET 2003\redist.txt on Disc 2 of the Visual C++ .NET product CD-ROM; any .ocx files in the Win\System folder are redistributable.

MFC ActiveX Controls: Distributing ActiveX Controls explains how to install and register redistributable ActiveX controls.

Merge Module Projects explains how Visual Studio .NET deployment handles the redistribution of files through merge modules.

Redistributing Database Support Files discusses how to redistribute support files for the database technologies found in the Microsoft Data Access SDK.

If your application uses an ActiveX control that connects to a database, you need to install or do the following:

- **DCOM for Windows.** You need to run Dcom98.exe or Dcom95.exe on any computer running versions of Windows earlier than Windows 2000. (Dcom98.exe is specifically for Windows 98; Dcom95.exe is specifically for Windows 95.) You can download these files from http://www.microsoft.com/com/resources/downloads.asp.
- **MDAC 2.7 SDK.** You should install the Microsoft Data Access 2.7 SDK on the target computer. You can download this from http://microsoft.com/data/download.htm; on this page, scroll down to "MDAC 2.7 Software Development Kit" and click the download link.
- **MDAC 2.7 redistribution program.** The MDAC 2.7 SDK is designed for use with the MDAC 2.7 redistribution program (Mdac_typ.exe) included in the MDAC27 directory on the Visual Studio .NET Prerequisites CD-ROM. You can also download Mdac_typ.exe from the MDAC 2.7 SDK download link listed above.
- **Replicate the DSN.** You also need to replicate the data source name on the target computer. You can do this programmatically with functions such as ConfigDSN.

Important Notes on Component Redistribution

- **Redistributing DAO components.** Microsoft recommends that you use Jet 4.0 SP3 (version 2927.04) or later. Jet 4.0 SP3 ships with Windows 2000 and Windows Me. Using this version of Jet reduces the number of Jet versions that must be tested with your application.

  Windows XP ships with an upgraded service pack version of Jet not included in earlier versions of Windows. Testing your application on Windows XP automatically tests the version of Jet that ships with Windows XP. You need to test DAO applications on both versions of Jet 4.0 before releasing them.

  The only difference in the Windows XP version is fixes for problems found since Windows 2000 was released. If users of your application do not encounter problems, there is no need to upgrade beyond Jet 4.0 SP3.

  If you use Windows 2000 and want to upgrade your version of Jet, you need to use Windows service packs. If you use a Windows version earlier than Windows 2000, you can obtain the latest version of Jet, as described in the Knowledge Base article, "ACC2000: Updated Version of Microsoft Jet 4.0 Available in Download Center" (Q239114). You can find Knowledge Base articles on the MSDN Library CD-ROM or at http://support.microsoft.com/support.
- **Known problems with ActiveX controls.** There is a known problem with dynamically creating instances of redistributable ActiveX controls on computers on which Visual C++ has not been installed, as described in the Knowledge Base article, "PRB: Dynamic Creation of Redistributable Control Fails" (Q151804). You can find Knowledge Base articles on the MSDN Library CD-ROM or at http://support.microsoft.com/support. There is also a known problem with placing some ActiveX controls on a dialog box; you get a message box stating that the control requires a design-time license, as described in the Knowledge Base article, "PRB: Need Design-Time License for Microsoft ActiveX Controls" (Q155059). You can find Knowledge Base articles on the MSDN Library CD-ROM or at http://support.microsoft.com/support.
- **Visual Studio licensed controls.** Visual Studio licensees can redistribute additional ActiveX controls specific to the other Visual Studio development tools. For example, the Chart control is distributed with Visual Basic, which also ships in Visual Studio. Thus, if you are using Visual C++ as a part of a Visual Studio license, you can redistribute the Chart control. However, if you purchased only Visual C++, you do not have a license to redistribute it.

See Also

Reference

Using ActiveX Controls

MFC ActiveX Controls: Distributing ActiveX Controls

Databinding with ActiveX Controls in Visual C++

Databinding is implemented through two types of ActiveX controls: data controls and data-bound controls.

Data controls

A data control is responsible for encapsulating a database query and the retrieved rowset. The Microsoft data controls provide a user interface consisting of a series of buttons to iterate through the data. Visual C++ offers two data access technologies for data controls: ADO and RDO.

Data-bound controls

A data-bound control is responsible for presenting the data. Data-bound controls connect to data controls to receive data and present the data through a variety of user interfaces. A Visual C++ application can also bind variables to data values set in the data-bound controls; see CWnd::BindProperty.

For more information about databinding, see:

- MFC ActiveX Controls: Using Data Binding in an ActiveX Control
- Data Access: ADO and RDO
- ADO Databinding
- RDO Databinding
- Wrapper Classes
- Setting Event Handlers on ActiveX Controls
- Error Trapping
- Limitations of Databinding

See Also
Reference
Data-Bound Controls (ADO and RDO)

Data Access: ADO and RDO

The following table shows two underlying technologies that support data-source or data-bound controls.

ADO

ADO is a COM wrapper of OLE DB that facilitates writing data access applications (consumers). OLE DB is a COM-based universal data access technology, allowing you to use any data source, not just indexed, sequential access methods (ISAM) and SQL-based databases.

OLE DB providers can access data from a variety of data sources and is not limited to SQL queries to retrieve data but rather can use queries as defined in the provider.

RDO

RDO is the COM wrapper of ODBC. ODBC, a C-based API, allows general-purpose (heterogeneous) data access. However, RDO relies on SQL as the command language to access data.

You might consider using the ADO-based data-access controls rather than the RDO data-access controls.

The following table shows the main differences between ADO and RDO data controls.

Data-bound controls

RDO data-bound controls use the **ICursor** interfaces; ADO controls use the OLE DB **IRowset** interface. In both cases, the interfaces used by the controls return a rowset.

The RDO-based data-bound controls were designed to work best with Visual Basic. As such, some functionality of RDO data-bound controls, most notably in formatting, is not available in Visual C++ applications. This problem is not present in the ADO databinding controls.

Data controls

ADO data controls implement the **IDataSource** interface and RDO data controls implement the **IVBDSC** interface. You can call an **IDataSource** method to receive an **IRowset** interface pointer. Similarly, you can call an IVBDSC method to get an **ICursor** interface pointer.

See Also

Reference

Databinding with ActiveX Controls in Visual C++

ADO Databinding

This section discusses the use of the databinding ActiveX controls that are built with ADO.

- Using ADO Databinding in Visual C++
- Setting Event Handlers on ActiveX Controls
- Updating Data with the ADO Data Control

See Also

Reference

Databinding with ActiveX Controls in Visual C++

Using ADO Databinding in Visual C++

Using ADO databinding in Visual C++ requires the following steps:

- Add an ADO data control.
- Point to a data source.
- Specify the record source (SQL query or data retrieval language).
- Add an ADO data-bound control.
- Connect the data-bound control to an ADO data control.
- Select the fields to bind to the ADO data control's record source.

To use ADO databinding in Visual C++

1. Create an MFC dialog application or MFC Formview application using the MFC Application Wizard.
2. Add the Microsoft ADO data control to the dialog box; see Inserting the Control into a Visual C++ Application.
3. Point the ADO data control to your OLE DB data source.
   a. Right-click the ADO data control, and then click **Properties**.
   b. On the **Control** tab, click **Use Connection String**. You can use the supplied provider or you can delete it.
   c. Click **Build**. If you deleted the provider from **Use Connection String**, you are now able to define one. After you define the provider, access the properties of the ADO data control again, and then select **Build** again to continue.

      If a provider is defined in **Use Connection String** before you select **Build**, you are able to define the data link properties. This displays the DataLink Wizard.
   d. Change the **Provider** if necessary, and define **Location** and **Data Source** values, as appropriate for your provider. For example, if you are using a SQL Server provider, **Location** specifies the database server and **Data Source** specifies the database. If you are using an ODBC provider, the **Data Source** corresponds to the ODBC DSN.
   e. Click the **Authentication** tab, and then set values for **User Name** and **Password**, if required by the data source.
   f. Click the **Connection** tab, and then click **Test Connection** to test the data source. Scroll to the end of the Results window to see if the test passed. If it failed, check the configuration of your data source. Common errors include invalid passwords and incorrect values for the **Location** and **Data Source** fields.
   g. Exit the DataLink Wizard and return to the property sheet for the ADO data control.
4. On the **RecordSource** tab, enter a query into **Command Text (SQL)**. The data-bound controls can bind to the results of this query. The query is usually SQL. However, some OLE DB providers do not use SQL.
5. Set any other ADO data control properties as needed and close the property sheet for the ADO data control.
6. Add a data-bound control. For example, add the DataGrid control, which is different from the RDO DBGrid control.)
7. Set the DataGrid's properties.
   a. Right-click the DataGrid, and then click **Properties**.
   b. Click the **All** tab, and then set the **DataSource** property to the ADO data control. Click the **DataSource** drop-down list and find the ID of the ADO data control. The default ID name is IDC_ADODC1.
8. To run in test mode, press CTRL+T. You are able to scroll through the data. Press the Esc key or close the dialog box to end test mode.

If you compile and run the program, you are able to scroll through the data as well.

See Also

Reference

ADO Databinding

Updating Data with the ADO Data Control

ADO data control data can be either read-only or modifiable.

To create an application that modifies data using ADO data control

1. Set ADO data control **CursorLocation** property. Options are:
   - Server Side
   - Client Side
2. Set ADO data control **LockType** property. The optimistic concurrency is recommended.
3. Set ADO data control **CursorType** property. Options are:
   - Keyset Cursor
   - Dynamic Cursor
   - Static Cursor

   Make sure the OLE DB provider supports the chosen option.
4. Set the data-bound control's properties, as needed, to allow updateability. Note that some controls do not allow updating.

For more information about these properties, see the ADO documentation.

See Also

Tasks

Using ADO Databinding in Visual C++

Reference

ADO Databinding

RDO Databinding

This section discusses the use of the databinding ActiveX controls that are built with RDO.

- Using RDO Databinding in Visual C++
- Updating Data with the RDO RemoteData Control
- Model for Hosting RDO Data Controls in a Container

See Also

Reference

Databinding with ActiveX Controls in Visual C++

Using RDO Databinding in Visual C++

To use RDO databinding in Visual C++, you need to add an RDO RemoteData control and point to a data source and a record source (SQL query). You also need to add an RDO data-bound control, point it to an RDO RemoteData control, and select the fields to bind to the RDO RemoteData control's record source.

To use RDO databinding in Visual C++

1. Configure an ODBC data source, if you have not already done so.
2. Create an MFC Dialog application or MFC Formview application using the MFC Application Wizard.
3. Add the Microsoft RemoteData control (RDO RemoteData control) to the dialog box; see Inserting the Control into a Visual C++ Application.
4. Point the RDO RemoteData control to your ODBC data source.
   a. Right-click the control, and then click **Properties**.
   b. Click the **Control** tab.
   c. Set **DataSource** to your ODBC data source.
   d. As needed, set the user name and password for your ODBC data source. Leave it blank if the data source does not require a user name or password.
   e. Enter an SQL query into the **SQL** property. The data-bound controls can bind to the results of this query.
5. Set any other RDO RemoteData control properties as needed.
6. Add a data-bound control. For example, add the **DBGrid Control** and set the data source as follows.
   a. Right-click the DBGrid, and then click **Properties**.
   b. Click the **All** tab.
   c. Set the **DataSource** property to the RDO RemoteData control. Click the drop-down combo box for the property and find the ID of the RDO RemoteData control. The default ID name is IDC_REMOTEDATACTL1.
7. To run in test mode, use CTRL+T. You are able to scroll through the data. Press the Esc key or close the dialog box to end test mode.

If you compile and run the program, you are able to scroll through the data as well.

See Also

Reference

RDO Databinding
Databinding with ActiveX Controls in Visual C++

Updating Data with the RDO RemoteData Control

RDO RemoteData control data can be either read-only or modifiable.

To create an application that modifies data using RDO RemoteData control

1. Set the RDO RemoteData control **CursorDriver** property.
   - Server Side cursors store returned data on the server.
   - ODBC Client Side cursors store data in the client's local storage.
   - Client Batch Side cursors use a cursor library designed to deal with concurrency issues.
   - No Cursor option is used when an action query is performed, and thus no cursor is necessary.
2. Set the RDO RemoteData control **LockType** property. The optimistic concurrency based on resultset option is recommended.
   - Read-only concurrency should not be used if you want data to be modifiable.
   - Pessimistic concurrency locks data during update so that other users are not in danger of committing incompatible data changes.
   - Optimistic concurrency does not lock data, but if during commit, it detects that another client has posted an incompatible state, the RDO RemoteData control throws an error.
3. Set the RDO RemoteData control **ResultsetType** property. Make sure the ODBC driver supports the chosen options:
   - When Create Static Cursor is chosen, the changes are not detected until the cursor is closed and reopened.
   - When Create Keyset Cursor is chosen, the cursor allows inserts, updates, and deletes, at will, within the keyset cursor itself.
4. Set the data-bound control as needed to allow updateability. Note that some controls do not allow updating.

For information about how to use these objects, see the documentation about the RDO RemoteData control.

See Also

Tasks

Using RDO Databinding in Visual C++

Reference

RDO Databinding

Model for Hosting RDO Data Controls in a Container

A container hosts an RDO data control as follows:

- The container obtains an IVBDSC interface from the data control. If it cannot find IVBDSC, it is not a data control.
- The container obtains the **ICursor** interfaces from the data control. These interfaces supply a Cursor object that can be manipulated by a client.
- The container hooks up to the data control's **INotifyDBEvents** interface. This interface allows the container to receive notifications from the data control. The container should support the **INotifyDBEventsSink** interface to do this.

A container hosts an RDO data-bound control as follows:

- The control supports the **IBoundObject** interface and the container supports the **IBoundObjectSite** interface. The control obtains the container's **IBoundObjectSite** interface, and the container obtains the **IBoundObject** interface from the control.
- The control supports the **IPropNotifySink** interface and hooks up with the container. This allows the container to receive notifications from the control.
- If the control supports **INotifyDBEventsSink**, it can receive notifications from an RDO data control after connecting with the data control's **INotifyDBEvents** interface.
- The control then can receive cursor objects from the data control (directly or through the container). The cursors can then be manipulated and scrolled. At this point, the RDO data-bound control is successfully bound.

See Also
Tasks
Using RDO Databinding in Visual C++
Reference
RDO Databinding

Error Trapping

In databinding, error trapping comes from two sources: error events or error objects.

Error Trapping via Error Events

Both the ADO data control and the RDO RemoteData control data controls have error events. Typically, you set an error event handler. The event handlers have the following signature.

```
void CMyDlg::OnErrorAdodc1(long ErrorNumber,
                           BSTR* FAR Description,
                           long Scode,
                           LPCTSTR Source,
                           LPCTSTR HelpFile,
                           long HelpContext,
                           BOOL FAR* fCancelDisplay)
```

Usually the Description field is populated, but the ErrorNumber and Scode fields are only populated in the event of COM errors. A standard event handler is to display the Description field in a message box. For example:

```
{
   USES_CONVERSION;
// note: have to include the ATL file ATLConv.h to use the ATL conversion macros
   ::AfxMessageBox(OLE2T(*Description), MB_OK);
}
```

However, because the ADO data control and RDO RemoteData control are already set up to trap error events, no coding is necessary.

Error Trapping Through Error Objects

Both ADO and RDO have error objects. When generating wrapper classes, the RDO RemoteData control generates wrappers for error objects, but the ADO data control does not.

The ADO data control automatically displays ADO error messages.

See Also
Reference
Databinding with ActiveX Controls in Visual C++

Limitations of Databinding

Databinding is a powerful way to create data applications quickly. However, the current databinding controls architecture is inherently two tier.

Scalability

ADO data-bound controls can only access data from the ADO data control. RDO data-bound controls can only access data from the RDO RemoteData control. For RDO RemoteData control, there is no workaround but to use a two-tier architecture, which results in the database server receiving all data retrieval requests directly. To avoid directly connecting to the database server, write a provider that allows access to middle-tier business and data objects. The ADO data control connects to these objects, rather than the database server. Such middle-tier objects can be cached and managed in a transaction server such as COM+ 1.0 services.

Versioning and Distribution

When new versions of controls are released, the application will have to be tested with the new versions. If another application is installed on a user's computer, and it has a different version of the controls, the application will have to be checked. Finally, when new versions of controls are released, the new controls will have to be distributed to application users.

Drivers and Providers

Databinding is only as good as the ODBC driver or OLE DB provider you are using. Because the drivers and providers are responsible for exposing data to data controls, it is important to ensure that the driver or provider supports the functionality that you need. When you select a driver or provider, you must also ensure that your users have the driver or provider installed. This includes installing any middleware that the driver or provider requires. For example, for ODBC Oracle connectivity, the user should have installed not only an ODBC Oracle driver, but also Oracle's SQL*Net middleware. For connectivity to Oracle 7.3 servers, the Microsoft Oracle ODBC driver is recommended.

Programmability

Because ActiveX controls were designed to be black-box components, programmability is limited to a developer's access to the control's interfaces. In the databinding model in the resource editor, this is implemented through wrapper classes generated by the Insert ActiveX Control Wizard. If the wizard is unable to detect a coclass, no wrapper class is generated and there is no programmatic access.

Despite these limitations, databinding affords a way to rapidly prototype data applications using Visual C++. If speed of development is important, databinding should be considered when designing your application.

See Also

Reference

Databinding with ActiveX Controls in Visual C++

Creating Database Connections

To use databinding, you must configure a data source. When using the ADO data control, you must configure an OLE DB connection. When using the RDO RemoteData control, you must create an Open Database Connectivity (ODBC) connection. Because Visual C++ ships with an OLE DB provider for ODBC data sources, the ADO data control can also make use of ODBC connections.

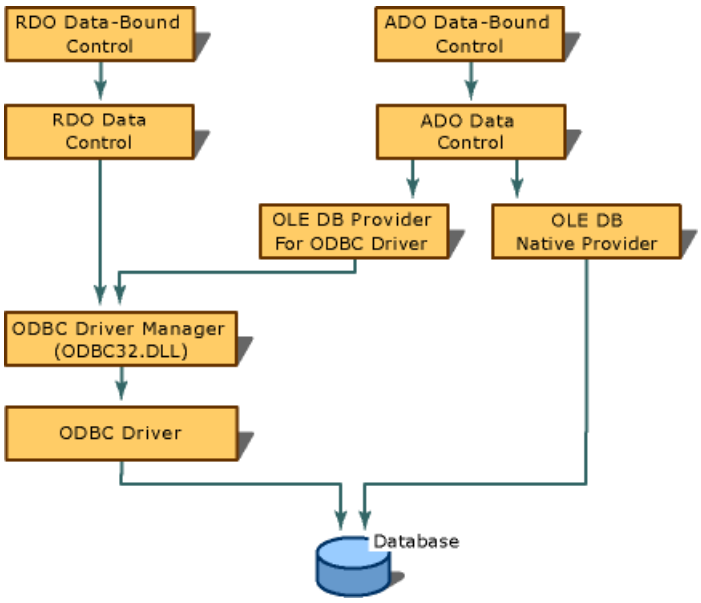

See Also

Tasks

ODBC Connections

Reference

Data-Bound Controls (ADO and RDO)

Oracle Connections

ODBC Connections

ODBC connections are configured in the system Control Panel. ODBC connections can be made against any data source for which an ODBC driver has been installed. Visual C++ 6.0 or later ships drivers for text files, Access, FoxPro, Paradox, dBASE, Excel, SQL Server, and Oracle. When you create an ODBC connection, it automatically receives a data source name (DSN). The DSN is subsequently used to identify connections in data controls, such as ADO data control and RDO RemoteData control.

OLE DB Connections No additional work is necessary to configure an OLE DB connection. For example, if an ODBC data source is created, the OLE DB provider for ODBC automatically detects it.

To configure an ODBC data source

1. Click **Start**, click **Settings**, and then click **Control Panel**.
2. In Control Panel, select 32bit ODBC (Windows 95 or 98) or ODBC (Windows NT or 2000).
3. Click the **User DSN** or **System DSN** tab. **User DSN** lets you create user-specific data source names and **System DSN** lets you create data sources available to all users.
4. Click **Add** to display a list of locally installed ODBC drivers.
5. Select the driver corresponding to the type of indexed sequential access method (ISAM) or database you want to connect to, and then click **Finish**.
6. Follow the instructions specific to the driver. After closing, the DSN is available for use.

When generating a DSN for some ODBC driver types, you need to know the location of the actual file. For example when creating an Access DSN, you need to know the location of the .mdb file. Also, you should have a valid user name and password. For example, the system user name for most Access systems is *admin*.

When creating an Oracle DSN, you should know the SQL*Net connection string.

See Also

Reference

Creating Database Connections

Oracle Connections

When configuring either an Oracle ODBC DSN or an OLE DB connection, you must have Oracle's SQL*Net client-side components installed. SQL*Net is Oracle's network transport layer. Most Oracle ODBC drivers, including Microsoft's driver and the Microsoft OLE DB Oracle provider map, call directly to this SQL*Net layer.

After SQL*Net is configured, note your SQL*Net connection string. Typically, this consists of a specifier for the Oracle database server name and the network protocol type (such as TCP/IP and named pipes). Often, the SQL*Net connection string is mapped to an alias. In this case, it is only necessary to note the alias. For information about how to configure SQL*Net, contact your Oracle DBA, refer to the SQL*Net documentation, or refer to your Oracle ODBC driver Help file for an example of a connection string.

See Also

Reference

Creating Database Connections

DLLs

A dynamic-link library (DLL) is an executable file that acts as a shared library of functions. Dynamic linking provides a way for a process to call a function that is not part of its executable code. The executable code for the function is located in a DLL, which contains one or more functions that are compiled, linked, and stored separately from the processes that use them. DLLs also facilitate the sharing of data and resources. Multiple applications can simultaneously access the contents of a single copy of a DLL in memory.

Dynamic linking differs from static linking in that it allows an executable module (either a .dll or .exe file) to include only the information needed at run time to locate the executable code for a DLL function. In static linking, the linker gets all of the referenced functions from the static link library and places it with your code into your executable.

Using dynamic linking instead of static linking offers several advantages. DLLs save memory, reduce swapping, save disk space, upgrade easier, provide after-market support, provide a mechanism to extend the MFC library classes, support multilanguage programs, and ease the creation of international versions.

The following topics provide detailed information about programming DLLs.

In This Section

Differences Between Applications and DLLs

Describes the fundamental differences between applications and DLLs.

Advantages of Using DLLs

Describes the advantages of dynamic linking.

Differences Between Win16 and Win32 DLLs

Describes the changes between Win16 and Win32 DLLs.

DLL Frequently Asked Questions

Provides answers to frequently asked questions about DLLs.

Linking an Executable to a DLL

Describes explicit and implicit linking to a DLL.

Initializing a DLL

Discusses DLL initialization code (such as allocating memory) that must execute when your DLL loads.

Run-Time Library Behavior

Describes how the run-time library performs the DLL startup sequence.

LoadLibrary and AfxLoadLibrary

Discusses using **LoadLibrary** and **AfxLoadLibrary** to explicitly link to a DLL.

GetProcAddress

Discusses using **GetProcAddress** to obtain the address of an exported function in the DLL.

FreeLibrary and AfxFreeLibrary

Discusses using **FreeLibrary** and **AfxFreeLibrary** when the DLL module is no longer needed.

Search Path Used by Windows to Locate a DLL

Describes the search path that the Windows operating system uses to locate a DLL on your system.

Module States of a Regular DLL Dynamically Linked to MFC

Describes the module states of a regular DLL dynamically linked to MFC.

Extension DLLs

Explains DLLs that typically implements reusable classes derived from the existing Microsoft Foundation Class Library classes.

Creating a Resource-Only DLL

Discusses a resource-only DLL, which contains nothing but resources, such as icons, bitmaps, strings, and dialog boxes.

Localizing Resources in MFC Applications: Satellite DLLs

Provides enhanced support for satellite DLLs, a feature that helps in creating applications localized for multiple languages.

Importing and Exporting

Describes importing public symbols into an application or exporting functions from a DLL

Active Technology and DLLs

Allows object servers to be completely implemented inside a DLL.

Automation in a DLL

Describes what the Automation option in the MFC DLL Wizard supplies.

Naming Conventions for MFC DLLs

Discusses how the DLLs and libraries included in MFC follow a structured naming convention.

Calling DLL functions from Visual Basic Applications

Describes how to call DLL functions from Visual Basic applications.

Related Sections

Using MFC as Part of a DLL

Describes regular DLLs, which allow you to use the MFC library as part of a Windows dynamic-link library.

DLL Version of MFC

Describes how you can use the MFCxx.dll and MFCxxD.dll (where x is the MFC version number) shared dynamic-link libraries with MFC applications and extension DLLs.

Adding Functionality

Provides links to topics describing conceptual information about the Visual C++ libraries and topics discussing various coding technologies and techniques.

Differences Between Applications and DLLs

Even though DLLs and applications are both executable program modules, they differ in several ways. To the end user, the most obvious difference is that DLLs are not programs that can be directly executed. From the system's point of view, there are two fundamental differences between applications and DLLs:

- An application can have multiple instances of itself running in the system simultaneously, whereas a DLL can have only one instance.
- An application can own things such as a stack, global memory, file handles, and a message queue, but a DLL cannot.

What do you want to do?

- Export from a DLL
- Link an executable to a DLL
- Initialize a DLL

What do you want to know more about?

- The advantages of using DLLs
- Non-MFC DLLs: Overview
- Regular DLLs statically linked to MFC
- Regular DLLs dynamically linked to MFC
- Extension DLLs: Overview

See Also

Concepts

DLLs

Advantages of Using DLLs

Dynamic linking has the following advantages:

- Saves memory and reduces swapping. Many processes can use a single DLL simultaneously, sharing a single copy of the DLL in memory. In contrast, Windows must load a copy of the library code into memory for each application that is built with a static link library.
- Saves disk space. Many applications can share a single copy of the DLL on disk. In contrast, each application built with a static link library has the library code linked into its executable image as a separate copy.
- Upgrades to the DLL are easier. When the functions in a DLL change, the applications that use them do not need to be recompiled or relinked as long as the function arguments and return values do not change. In contrast, statically linked object code requires that the application be relinked when the functions change.
- Provides after-market support. For example, a display driver DLL can be modified to support a display that was not available when the application was shipped.
- Supports multilanguage programs. Programs written in different programming languages can call the same DLL function as long as the programs follow the function's calling convention. The programs and the DLL function must be compatible in the following ways: the order in which the function expects its arguments to be pushed onto the stack, whether the function or the application is responsible for cleaning up the stack, and whether any arguments are passed in registers.
- Provides a mechanism to extend the MFC library classes. You can derive classes from the existing MFC classes and place them in an MFC extension DLL for use by MFC applications.
- Eases the creation of international versions. By placing resources in a DLL, it is much easier to create international versions of an application. You can place the strings for each language version of your application in a separate resource DLL and have the different language versions load the appropriate resources.

A potential disadvantage to using DLLs is that the application is not self-contained; it depends on the existence of a separate DLL module.

What do you want to do?

- Export from a DLL
- Link an executable to a DLL
- Initialize a DLL

What do you want to know more about?

- Non-MFC DLLs: Overview
- Regular DLLs statically linked to MFC
- Regular DLLs dynamically linked to MFC
- Extension DLLs: Overview

See Also

Concepts

DLLs

Kinds of DLLs

This topic provides information to help you determine the kind of DLL to build.

Different Kinds of DLLs Available

Using Visual C++, you can build Win32 DLLs in C or C++ that do not use the Microsoft Foundation Class (MFC) library. You can create a non-MFC DLL project with the Win32 Application Wizard.

The MFC library itself is available, in either static link libraries or in a number of DLLs, with the MFC DLL Wizard. If your DLL is using MFC, Visual C++ supports three different DLL development scenarios:

- Building a regular DLL that statically links MFC
- Building a regular DLL that dynamically links MFC
- Building an MFC extension DLL, which always dynamically link MFC

What do you want to know more about?

- Non-MFC DLLs: Overview
- Regular DLLs statically linked to MFC
- Regular DLLs dynamically linked to MFC
- Extension DLLs: Overview
- Which kind of DLL to use

Deciding Which Kind of DLL to Use

If your DLL does not use MFC, use Visual C++ to build a non-MFC Win32 DLL. Linking your DLL to MFC (either statically or dynamically) takes up significant disk space and memory. You should not link to MFC unless your DLL actually uses MFC.

If your DLL will be using MFC, and will be used by either MFC or non-MFC applications, you must build either a regular DLL that dynamically links to MFC or a regular DLL that statically links to MFC. In most cases, you probably want to use a regular DLL that dynamically links to MFC because the file size of the DLL will be much smaller and the savings in memory from using the shared version of MFC can be significant. If you statically link to MFC, the file size of your DLL will be larger and potentially take up extra memory because it loads its own private copy of the MFC library code.

Building a DLL that dynamically links to MFC is faster than building a DLL that statically links to MFC because it is not necessary to link MFC itself. This is especially true in debug builds where the linker must compact the debug information. By linking with a DLL that already contains the debug information, there is less debug information to compact within your DLL.

One disadvantage to dynamically linking to MFC is that you must distribute the shared DLLs Mfcx0.dll and Msvcrt.dll (or similar files) with your DLL. The MFC DLLs are freely redistributable, but you still must install the DLLs in your setup program. In addition, you must ship the Msvcrt.dll, which contains the C run-time library that is used both by your program and the MFC DLLs themselves.

If your DLL will only be used by MFC executables, you have a choice between building a regular DLL or an extension DLL. If your DLL implements reusable classes derived from the existing MFC classes or you need to pass MFC-derived objects between the application and the DLL, you must build an extension DLL.

If your DLL dynamically links to MFC, the MFC DLLs might be redistributed with your DLL. This architecture is particularly useful for sharing the class library between multiple executable files to save disk space and minimize memory usage.

Prior to version 4.0, Visual C++ only supported two kinds of DLLs that used MFC: USRDLLs and AFXDLLs. Regular DLLs statically linked to MFC have the same characteristics as the former USRDLL. MFC extension DLLs have the same characteristics as the former AFXDLLs.

What do you want to know more about?

- Non-MFC DLLs: Overview
- Regular DLLs statically linked to MFC
- Regular DLLs dynamically linked to MFC

- Extension DLLs: Overview

See Also

Concepts

DLLs

Non-MFC DLLs: Overview

A non-MFC DLL is a DLL that does not use MFC internally, and the exported functions in the DLL can be called by either MFC or non-MFC executable files. Functions are usually exported from a non-MFC DLL using the standard C interface.

For more information about non-MFC DLLs, see Dynamic-Link Libraries in the *Platform SDK*.

What do you want to do?

- Create a Win32 DLL
- Export from a DLL
- Link an executable to a DLL
- Initialize a DLL

What do you want to know more about?

- Regular DLLs statically linked to MFC
- Regular DLLs dynamically linked to MFC
- Extension DLLs: Overview

See Also

Concepts

Kinds of DLLs

Regular DLLs Statically Linked to MFC

A regular DLL statically linked to MFC is a DLL that uses MFC internally, and the exported functions in the DLL can be called by either MFC or non-MFC executables. As the name describes, this kind of DLL is built using the static link library version of MFC. Functions are usually exported from a regular DLL using the standard C interface. For an example of how to write, build, and use a regular DLL, see the sample DLLScreenCap.

Note that the term USRDLL is no longer used in the Visual C++ documentation. A regular DLL that is statically linked to MFC has the same characteristics as the former USRDLL.

A regular DLL, statically linked to MFC, has the following features:

- The client executable can be written in any language that supports the use of DLLs (C, C++, Pascal, Visual Basic, and so on); it does not have to be an MFC application.
- The DLL can link to the same MFC static link libraries used by applications. There is no longer a separate version of the static link libraries for DLLs.
- Before version 4.0 of MFC, USRDLLs provided the same type of functionality as regular DLLs statically linked to MFC. As of Visual C++ version 4.0, the term USRDLL is obsolete.

A regular DLL, statically linked to MFC, has the following requirements:

- This type of DLL must instantiate a class derived from **CWinApp**.
- This type of DLL uses the **DllMain** provided by MFC. Place all DLL-specific initialization code in the **InitInstance** member function and termination code in **ExitInstance** as in a normal MFC application.
- Even though the term USRDLL is obsolete, you must still define "**_USRDLL**" on the compiler command line. This definition determines which declarations is pulled in from the MFC header files.

Regular DLLs must have a **CWinApp**-derived class and a single object of that application class, as does an MFC application. However, the **CWinApp** object of the DLL does not have a main message pump, as does the **CWinApp** object of an application.

Note that the **CWinApp::Run** mechanism does not apply to a DLL, because the application owns the main message pump. If the DLL opens modeless dialogs or has a main frame window of its own, the application's main message pump must call a routine exported by the DLL that in turn calls the **CWinApp::PreTranslateMessage** member function of the DLL's application object.

For an example of this function, see the DLLScreenCap sample.

Symbols are usually exported from a regular DLL using the standard C interface. The declaration of a function exported from a regular DLL would look something like this:

```
extern "C" __declspec(dllexport) MyExportedFunction( );
```

All memory allocations within a regular DLL should stay within the DLL; the DLL should not pass to or receive from the calling executable any of the following:

- Pointers to MFC objects
- Pointers to memory allocated by MFC

If you need to do any of the above or need to pass MFC-derived objects between the calling executable and the DLL, you must build an extension DLL.

It is safe to pass pointers to memory that were allocated by the C run-time libraries between an application and a DLL only if you make a copy of the data. You must not delete or resize these pointers or use them without making a copy of the memory.

A DLL that is statically linked to MFC cannot also dynamically link to the shared MFC DLLs. A DLL that is statically linked to MFC is dynamically bound to an application just like any other DLL; applications link to it just like any other DLL.

The standard MFC static link libraries are named according to the convention described in Naming Conventions for MFC DLLs. However, with MFC version 3.0 and later, it is no longer necessary to manually specify to the linker the version of the MFC library you want linked in. Instead, the MFC header files automatically determine the correct version of the MFC library to link

in based on preprocessor defines, such as **_DEBUG** or **_UNICODE**. The MFC header files add /DEFAULTLIB directives instructing the linker to link in a specific version of the MFC library.

What do you want to do?

- Initialize regular DLLs

What do you want to know more about?

- Using MFC as Part of a DLL
- Using Database, OLE, and Sockets Extension DLLs in Regular DLLs
- Creating an MFC DLL
- Regular DLLs Dynamically Linked to MFC
- Extension DLLs

See Also

Concepts

Kinds of DLLs

Regular DLLs Dynamically Linked to MFC

A regular DLL dynamically linked to MFC is a DLL that uses MFC internally, and the exported functions in the DLL can be called by either MFC or non-MFC executables. As the name describes, this kind of DLL is built using the dynamic-link library version of MFC (also known as the shared version of MFC). Functions are usually exported from a regular DLL using the standard C interface.

You must add the **AFX_MANAGE_STATE** macro at the beginning of all the exported functions in regular DLLs that dynamically link to MFC to set the current module state to the one for the DLL. This is done by adding the following line of code to the beginning of functions exported from the DLL:

```
AFX_MANAGE_STATE(AfxGetStaticModuleState( ))
```

A regular DLL, dynamically linked to MFC has the following features:

- This is a new type of DLL introduced by Visual C++ 4.0.
- The client executable can be written in any language that supports the use of DLLs (C, C++, Pascal, Visual Basic, and so on); it does not have to be an MFC application.
- Unlike the statically linked regular DLL, this type of DLL is dynamically linked to the MFC DLL (also known as the shared MFC DLL).
- The MFC import library linked to this type of DLL is the same one used for extension DLLs or applications using the MFC DLL: MFCxx(D).lib.

A regular DLL, dynamically linked to MFC has the following requirements:

- These DLLs are compiled with **_AFXDLL** defined, just like an executable that is dynamically linked to the MFC DLL. But **_USRDLL** is also defined, just like a regular DLL that is statically linked to MFC.
- This type of DLL must instantiate a **CWinApp**-derived class.
- This type of DLL uses the **DllMain** provided by MFC. Place all DLL-specific initialization code in the **InitInstance** member function and termination code in **ExitInstance** as in a normal MFC application.

Because this kind of DLL uses the dynamic-link library version of MFC, you must explicitly set the current module state to the one for the DLL. To do this, use the AFX_MANAGE_STATE macro at the beginning of every function exported from the DLL.

Regular DLLs must have a **CWinApp**-derived class and a single object of that application class, as does an MFC application. However, the **CWinApp** object of the DLL does not have a main message pump, as does the **CWinApp** object of an application.

Note that the **CWinApp::Run** mechanism does not apply to a DLL, because the application owns the main message pump. If your DLL brings up modeless dialogs or has a main frame window of its own, your application's main message pump must call a DLL-exported routine that calls **CWinApp::PreTranslateMessage**.

Place all DLL-specific initialization in the **CWinApp::InitInstance** member function as in a normal MFC application. The **CWinApp::ExitInstance** member function of your **CWinApp** derived class is called from the MFC provided **DllMain** function before the DLL is unloaded.

You must distribute the shared DLLs MFCx0.dll and Msvcrt.dll (or similar files) with your application.

A DLL that is dynamically linked to MFC cannot also statically link to MFC. Applications link to regular DLLs dynamically linked to MFC it just like any other DLL.

Symbols are usually exported from a regular DLL using the standard C interface. The declaration of a function exported from a regular DLL looks something like this:

```
extern "C" __declspec(dllexport) MyExportedFunction( );
```

All memory allocations within a regular DLL should stay within the DLL; the DLL should not pass to or receive from the calling executable any of the following:

- pointers to MFC objects
- pointers to memory allocated by MFC

If you need to do any of the above, or if you need to pass MFC-derived objects between the calling executable and the DLL, then you must build an extension DLL.

It is safe to pass pointers to memory that were allocated by the C run-time libraries between an application and a DLL only if you make a copy of the data. You must not delete or resize these pointers or use them without making a copy of the memory.

When building a regular DLL that dynamically links to MFC, you need to use the macro AFX_MANAGE_STATE to switch the MFC module state correctly. This is done by adding the following line of code to the beginning of functions exported from the DLL:

```
AFX_MANAGE_STATE(AfxGetStaticModuleState( ))
```

The **AFX_MANAGE_STATE** macro should not be used in regular DLLs that statically link to MFC or in extension DLLs. For more information, see Managing the State Data of MFC Modules.

For an example of how to write, build, and use a regular DLL, see the sample DLLScreenCap. For more information about regular DLLs that dynamically link to MFC, see the section titled "Converting DLLScreenCap to Dynamically Link with the MFC DLL" in the abstract for the sample.

What do you want to do?

- Initialize regular DLLs

What do you want to know more about?

- The module states of a regular DLL dynamically linked to MFC
- Managing the state data of MFC modules
- Using Database, OLE, and Sockets Extension DLLs in Regular DLLs
- Using MFC as Part of a DLL

See Also

Concepts

Kinds of DLLs

Extension DLLs: Overview

An MFC extension DLL is a DLL that typically implements reusable classes derived from existing Microsoft Foundation Class Library classes. Extension DLLs are built using the dynamic-link library version of MFC (also known as the shared version of MFC). Only MFC executables (either applications or regular DLLs) that are built with the shared version of MFC can use an extension DLL. With an extension DLL, you can derive new custom classes from MFC and then offer this extended version of MFC to applications that call your DLL.

Extension DLLs can also be used for passing MFC-derived objects between the application and the DLL. The member functions associated with the passed object exist in the module where the object was created. Because these functions are properly exported when using the shared DLL version of MFC, you can freely pass MFC or MFC-derived object pointers between an application and the extension DLLs it loads.

For an example of a DLL that fulfills the basic requirements of an extension DLL, see the MFC sample DLLHUSK. In particular, look at the Testdll1.cpp and Testdll2.cpp files.

Note that the term AFXDLL is no longer used in the Visual C++ documentation. An extension DLL has the same characteristics as the former AFXDLL.

What do you want to do?

- Initialize an extension DLL

What do you want to know more about?

- Extension DLLs
- Using Database, OLE, and Sockets Extension DLLs in Regular DLLs
- Non-MFC DLLs: Overview
- Regular DLLs statically linked to MFC
- Regular DLLs dynamically linked to MFC
- Creating an MFC DLL

See Also

Concepts

Kinds of DLLs

Differences Between Win16 and Win32 DLLs

If you have built 16-bit DLLs for Windows 3.*x*, you should find that building 32-bit DLLs is more convenient. The compiler offers more direct support, which can save you several steps in DLL creation. The specific differences between Win16 and Win32 DLLs are:

- There is no separate startup module that you have to link with. The DLL startup sequence is handled directly by C/C++ run-time library code linked into your DLL.
- The run-time library code initializes any static non-local C++ objects by calling the appropriate constructors. Each process gets its own copy of all the DLL's static data, including objects.
- You no longer need the function **LibMain** or a **WEP** (Windows Exit Procedure). Where you add initialization and termination code for your DLL depends on the type of DLL you are building. Instead of **LibMain**, you provide **DllMain**, which is called for both for both entry and exit.
- You can import and export symbols directly in your source code. If you use the **__declspec(dllexport)** attribute (similar to **__export** in Windows 3.*x*), you do not need to use a separate module-definition file for exports.
- Executables that use **__declspec(dllimport)** to import data, objects, and functions from a DLL cause the compiler to generate more efficient code.
- The timing of calls to routines registered with atexit can differ.
- In addition to Win32 non-MFC DLLs, Visual C++ offers three kinds of MFC DLLs.

What do you want to do?

- Export from a DLL
- Initialize a DLL
- Link an executable to a DLL

What do you want to know more about?

- Importing and exporting
- Run-time library behavior

See Also

Concepts

DLLs

DLL Frequently Asked Questions

Following are some frequently asked questions (FAQ) about DLLs.

- How do I port my 16-bit DLL to a Win32 DLL?
- How do I convert my USRDLL to a regular DLL that uses the MFC shared library?
- How do I export data from a DLL?
- How do I share data in my DLL with an application or with other DLLs?
- Can a multithreaded application access an MFC DLL in different threads?
- Can an MFC DLL create multiple threads?
- Are there any MFC classes or functions that cannot be used in an MFC DLL?
- How do I debug my DLL?
- What optimization techniques should I use to improve the client application's performance when loading?
- There's a memory leak in my regular DLL, but my code looks fine. How can I find the memory leak?
- How do I create a modal dialog box from within a regular DLL?

See Also

Concepts

DLLs

How do I port my 16-bit DLL to a Win32 DLL?

If you have built 16-bit DLLs for Windows 3.*x*, you should find that building 32-bit DLLs is more convenient. Visual C++ offers more direct support, which can save you several steps in DLL creation. For more information, see Differences Between Win16 and Win32 DLLs.

The differences between Win16 and Win32 DLLs require more than just a simple recompilation to turn your Win16 DLL into a Win32 DLL. For more information about porting 16-bit DLLs to Win32 DLLs, see the Knowledge Base article, "How to Port a 16-bit DLL to a Win32 DLL" (Q125688).

See Also

Concepts

DLL Frequently Asked Questions

How do I convert my USRDLL to a regular DLL that uses the MFC shared library?

Note that the term USRDLL is no longer used in the Visual C++ documentation. A regular DLL that is statically linked to MFC has the same characteristics as the former USRDLL.

Converting your DLL to dynamically link to the MFC libraries requires more than just rebuilding the DLL with the MFC shared library. For more information about converting a regular DLL that statically links to MFC to a DLL that dynamically links to MFC, see the section "Converting DLLScreenCap to Dynamically Link with the MFC DLL" in the abstract for the DLLScreenCap sample.

See Also

Concepts

DLL Frequently Asked Questions

How do I export data from a DLL?

It is possible for a Win32-based application to be able to address DLL global variables directly by name from within the executable. This is done by exporting global data names in a way that is similar to the way you export a DLL function name. For more information about how to export data from a DLL, see Exporting from a DLL and the Knowledge Base article "Exporting Data from a DLL or an Application" (Q90530).

See Also

Concepts

DLL Frequently Asked Questions

How do I share data in my DLL with an application or with other DLLs?

Win32 DLLs are mapped into the address space of the calling process. By default, each process using a DLL has its own instance of all the DLLs global and static variables. If your DLL needs to share data with other instances of it loaded by other applications, you can use either of the following approaches:

- Create named data sections using the data_seg pragma.
- Use memory mapped files. See the Win32 documentation about memory mapped files.

Here is an example of using the **data_seg** pragma:

```
#pragma data_seg (".myseg")
   int i = 0;
   char a[32]n = "hello world";
#pragma data_seg()
```

data_seg can be used to create a new named section (.myseg in this example). The most typical usage is to call the data segment .shared for clarity. You then must specify the correct sharing attributes for this new named data section in your .def file or with the linker option /SECTION:.MYSEC,RWS.

There are restrictions to consider before using a shared data segment:

- Any variables in a shared data segment must be statically initialized. In the above example, i is initialized to 0 and a is 32 characters initialized to hello world.
- All shared variables are placed in the compiled DLL in the specified data segment. Very large arrays can result in very large DLLs. This is true of all initialized global variables.
- Never store process-specific information in a shared data segment. Most Win32 data structures or values (such as HANDLEs) are really valid only within the context of a single process.
- Each process gets its own address space. It is very important that pointers are never stored in a variable contained in a shared data segment. A pointer might be perfectly valid in one application but not in another.
- It is possible that the DLL itself could get loaded at a different address in the virtual address spaces of each process. It is not safe to have pointers to functions in the DLL or to other shared variables.

Note that these last three points apply to memory-mapped files and shared data segments.

Memory-mapped files have an advantage over shared data sections because the start of the memory-mapped file is known. Developers can implement pointer-like behavior by using "offset from the start of the shared memory section" in all data located inside the shared memory. __based pointers are highly recommended for making this fast and easy. However, it is important to remember that the base (or start of the memory mapped file) can be different in each process, so the variable storing the base for **__based** pointers cannot itself be in the shared memory.

These restrictions have important implications to C++ classes.

- Classes with virtual functions always contain function pointers. Classes with virtual functions should never be stored in shared data segments nor in memory mapped files. This is particularly important to MFC classes or classes that inherit from MFC.
- Static data members are implemented as the equivalent of global variables. This means that each process would have its own copy of that class's static data members. Classes with static data members should not be shared.
- The initialization requirement of a shared data segment causes a particular problem for C++ classes. If you have something like `CTest Counter(0);` in a shared data segment, the `Counter` object gets initialized in each process as they load the DLL, potentially zeroing out the object's data each time. This is very different than intrinsic data types that are initialized by the linker when it creates the DLL.

Because of these restrictions, Microsoft does not recommend sharing C++ objects between processes. In general, if you want to use C++ to share data between processes, write a class that internally uses a memory-mapped file to share data, but do not

share the class instances themselves. This might require special care in developing such a class, but it enables application developers to fully control the side effects of sharing data.

For more information about creating named data sections, see the following Knowledge Base articles at http://support.microsoft.com:

- "How to Share Data Between Different Mappings of a DLL" (Q125677).
- "Specifying Shared and Nonshared Data in a DLL" (Q100634).
- "Sharing All Data in a DLL" (Q109619).
- "Memory in Shared Code Sections Is Not Shared Across Terminal Server Sessions" (Q251045)

See Also

Concepts

DLL Frequently Asked Questions

Can a multithreaded application access an MFC DLL in different threads?

Multithreaded applications can access regular DLLs that dynamically link to MFC and extension DLLs from different threads. And as of Visual C++ version 4.2, an application can access regular DLLs that statically link to MFC from multiple threads created in the application.

Prior to version 4.2, only one external thread could attach to a regular DLL that statically linked to MFC. For more information about restrictions accessing regular DLLs that statically link to MFC from multiple threads (prior to Visual C++ version 4.2), see the Knowledge Base article, "Multiple Threads and MFC _USRDLLs" (Q122676).

Note that the term USRDLL is no longer used in the Visual C++ documentation. A regular DLL that is statically linked to MFC has the same characteristics as the former USRDLL.

See Also

Concepts

DLL Frequently Asked Questions

Can an MFC DLL create multiple threads?

Except during initialization, an MFC DLL can safely create multiple threads as long as it uses the Win32 thread local storage (TLS) functions such as **TlsAlloc** to allocate thread local storage. However, if an MFC DLL uses **__declspec(thread)** to allocate thread local storage, the client application must be implicitly linked to the DLL. If the client application explicitly links to the DLL, the call to **LoadLibrary** will not successfully load the DLL. For more information about creating multiple threads inside MFC DLLs, see the Knowledge Base article, "PRB: Calling LoadLibrary() to Load a DLL That Has Static TLS" (Q118816).

An MFC DLL that creates a new MFC thread during startup will hang when it is loaded by an application. This includes whenever a thread is created by calling **AfxBeginThread** or **CWinThread::CreateThread** inside:

- The **InitInstance** of a **CWinApp**-derived object in a regular DLL.
- A supplied **DllMain** or **RawDllMain** function in a regular DLL.
- A supplied **DllMain** or **RawDllMain** function in an Extension DLL.

For more information about creating threads during initialization, see the Knowledge Base article, "PRB: Cannot Create an MFC Thread During DLL Startup" (Q142243).

See Also

Concepts

DLL Frequently Asked Questions

Are there any MFC classes or functions that cannot be used in an MFC DLL?

Extension DLLs use the **CWinApp**-derived class of the client application. They must not have their own **CWinApp**-derived class.

Regular DLLs must have a **CWinApp**-derived class and a single object of that application class, as does an MFC application. Unlike the **CWinApp** object of an application, the **CWinApp** object of the DLL does not have a main message pump.

Note that because the **CWinApp::Run** mechanism does not apply to a DLL, the application owns the main message pump. If the DLL opens modeless dialog boxes or has a main frame window of its own, the application's main message pump must call a routine exported by the DLL, which in turn calls the **CWinApp::PreTranslateMessage** member function of the DLL's application object.

See Also

Concepts

DLL Frequently Asked Questions

How do I debug my DLL?

To debug a DLL using Visual C++, you must build a debugging version of the DLL and call it from an application. However, it is not necessary to build the debugging version of the calling application or to build the calling application with Visual C++. For more information about debugging a DLL, see Debugging DLLs or the following Knowledge Base articles:

- "Debugging a Dynamic-Link Library (DLL) in Windows" (Q85221).
- "Problems Loading a Debuggee That Uses a DLL" (Q119518).

See Also

Concepts

DLL Frequently Asked Questions

What optimization techniques should I use to improve the client application's performance when loading?

If your DLL is a regular DLL that is statically linked to MFC, changing it to a regular DLL that is dynamically linked to MFC reduces the file size.

If the DLL has a large number of exported functions, use a .def file to export the functions (instead of using **__declspec(dllexport)**) and use the .def file NONAME attribute on each exported function. The NONAME attribute causes only the ordinal value and not the function name to be stored in the DLL's export table, which reduces the file size.

DLLs that are implicitly linked to an application are loaded when the application loads. To improve the performance when loading, try dividing the DLL into different DLLs. Put all the functions that the calling application needs immediately after loading into one DLL and have the calling application implicitly link to that DLL. Put the other functions that the calling application does not need right away into another DLL and have the application explicitly link to that DLL. For more information, see Determining Which Linking Method to Use.

See Also

Concepts

DLL Frequently Asked Questions

There's a memory leak in my regular DLL, but my code looks fine. How can I find the memory leak?

One possible cause of the memory leak is that MFC creates temporary objects that are used inside message handler functions. In regular DLLs, MFC does not automatically release memory allocated for these objects. For more information, see Memory Management and the Debug Heap or the Knowledge Base article, "Cleaning Up Temporary MFC Objects in _USRDLL DLLs" (Q105286).

Note that the term USRDLL is no longer used in the Visual C++ documentation. A regular DLL that is statically linked to MFC has the same characteristics as the former USRDLL. The advice in the Knowledge Base article also applies to regular DLLs that are dynamically linked to MFC. The information in the above Knowledge Base article applies to both regular DLLs that statically link to MFC and regular DLLs that dynamically link to MFC.

See Also

Concepts

DLL Frequently Asked Questions

How do I create a modal dialog box from within a regular DLL?

When creating a modal dialog box from within a regular DLL statically linked to the MFC, you must pass a valid parent window object to the **CDialog** constructor. For more information about creating a modal dialog box in a regular DLL that statically links to MFC, see the Knowledge Base article, "How to Create a Modal Dialog from Within a USRDLL" (Q121947). For sample code that creates a modal dialog box, see the sample DLLScreenCap.

Remarks

Tthe term USRDLL is no longer used in the Visual C++ documentation. A regular DLL that is statically linked to MFC has the same characteristics as the former USRDLL. The advice in the Knowledge Base article also applies to regular DLLs that are dynamically linked to MFC.

See Also

Concepts

DLL Frequently Asked Questions

Linking an Executable to a DLL

An executable file links to (or loads) a DLL in one of two ways:

- Implicit linking
- Explicit linking

Implicit linking is sometimes referred to as static load or load-time dynamic linking. Explicit linking is sometimes referred to as dynamic load or run-time dynamic linking.

With implicit linking, the executable using the DLL links to an import library (.lib file) provided by the maker of the DLL. The operating system loads the DLL when the executable using it is loaded. The client executable calls the DLL's exported functions just as if the functions were contained within the executable.

With explicit linking, the executable using the DLL must make function calls to explicitly load and unload the DLL and to access the DLL's exported functions. The client executable must call the exported functions through a function pointer.

An executable can use the same DLL with either linking method. Furthermore, these mechanisms are not mutually exclusive, as one executable can implicitly link to a DLL and another can attach to it explicitly.

What do you want to know more about?

- Working with Import Libraries and Export Files
- Determining which linking method to use
- The search path used by Windows to locate a DLL

See Also

Concepts

DLLs

Determining Which Linking Method to Use

There are two types of linking: implicit linking and explicit linking.

Implicit Linking

Implicit linking occurs when an application's code calls an exported DLL function. When the source code for the calling executable is compiled or assembled, the DLL function call generates an external function reference in the object code. To resolve this external reference, the application must link with the import library (.lib file) provided by the maker of the DLL.

The import library only contains code to load the DLL and to implement calls to functions in the DLL. Finding an external function in an import library informs the linker that the code for that function is in a DLL. To resolve external references to DLLs, the linker simply adds information to the executable file that tells the system where to find the DLL code when the process starts up.

When the system starts a program that contains dynamically linked references, it uses the information in the program's executable file to locate the required DLLs. If it cannot locate the DLL, the system terminates the process and displays a dialog box that reports the error. Otherwise, the system maps the DLL modules into the process's address space.

If any of the DLLs has an entry-point function (for initialization and termination code), the operating system calls the function. One of the parameters passed to the entry-point function specifies a code that indicates the DLL is attaching to the process. If the entry-point function does not return TRUE, the system terminates the process and reports the error.

Finally, the system modifies the executable code of the process to provide starting addresses for the DLL functions.

Like the rest of a program's code, DLL code is mapped into the address space of the process when the process starts up and it is loaded into memory only when needed. As a result, the **PRELOAD** and **LOADONCALL** code attributes used by .def files to control loading in previous versions of Windows no longer have meaning.

Explicit Linking

Most applications use implicit linking because it is the easiest linking method to use. However, there are times when explicit linking is necessary. Here are some common reasons to use explicit linking:

- The application does not know the name of a DLL that it will have to load until run time. For example, the application might need to obtain the name of the DLL and the exported functions from a configuration file.
- A process using implicit linking is terminated by the operating system if the DLL is not found at process startup. A process using explicit linking is not terminated in this situation and can attempt to recover from the error. For example, the process could notify the user of the error and have the user specify another path to the DLL.
- A process using implicit linking is also terminated if any of the DLLs it is linked to have a **DllMain** function that fails. A process using explicit linking is not terminated in this situation.
- An application that implicitly links to many DLLs can be slow to start because Windows loads all the DLLs when the application loads. To improve startup performance, an application can implicitly link to those DLLs needed immediately after loading and wait to explicitly link to the other DLLs when they are needed.
- Explicit linking eliminates the need to link the application with an import library. If changes in the DLL cause the export ordinals to change, applications using explicit linking do not have to relink (assuming they are calling **GetProcAddress** with a name of a function and not with an ordinal value), whereas applications using implicit linking must relink to the new import library.

Here are two hazards of explicit linking to be aware of:

- If the DLL has a **DllMain** entry point function, the operating system calls the function in the context of the thread that called **LoadLibrary**. The entry-point function is not called if the DLL is already attached to the process because of a previous call to **LoadLibrary** with no corresponding call to the **FreeLibrary** function. Explicit linking can cause problems if the DLL is using a **DllMain** function to perform initialization for each thread of a process because threads existing when **LoadLibrary** (or **AfxLoadLibrary**) is called will not be initialized.
- If a DLL declares static-extent data as **__declspec(thread)**, it can cause a protection fault if explicitly linked. After the DLL is loaded with **LoadLibrary**, it causes a protection fault whenever the code references this data. (Static-extent data includes both global and local static items.) Therefore, when you create a DLL, you should either avoid using thread-local storage or inform DLL users about potential pitfalls (in case they attempt dynamic loading).

What do you want to do?

- Link implicitly
- Link explicitly

What do you want to know more about?

- The search path used by Windows to locate a DLL
- LoadLibrary and AfxLoadLibrary
- GetProcAddress
- FreeLibrary and AfxFreeLibrary
- Using thread local storage in a dynamic-link library (Platform SDK)

See Also

Concepts

Linking an Executable to a DLL

Linking Implicitly

To implicitly link to a DLL, executables must obtain the following from the provider of the DLL:

- A header file (.h file) containing the declarations of the exported functions and/or C++ classes. The classes, functions, and data should all have **__declspec(dllimport)**, for more information, see dllexport, dllimport.
- An import library (.LIB files) to link with. (The linker creates the import library when the DLL is built.)
- The actual DLL (.dll file).

Executables using the DLL must include the header file containing the exported functions (or C++ classes) in each source file that contains calls to the exported functions. From a coding perspective, the function calls to the exported functions are just like any other function call.

To build the calling executable file, you must link with the import library. If you are using an external makefile, specify the file name of the import library where you list other object (.obj) files or libraries that you are linking with.

The operating system must be able to locate the DLL file when it loads the calling executable.

What do you want to do?

- Link explicitly
- Determine which linking method to use

What do you want to know more about?

- Working with Import Libraries and Export Files
- The search path used by Windows to locate a DLL

See Also

Concepts

Linking an Executable to a DLL

Linking Explicitly

With explicit linking, applications must make a function call to explicitly load the DLL at run time. To explicitly link to a DLL, an application must:

- Call **LoadLibrary** (or a similar function) to load the DLL and obtain a module handle.
- Call **GetProcAddress** to obtain a function pointer to each exported function that the application wants to call. Because applications are calling the DLL's functions through a pointer, the compiler does not generate external references, so there is no need to link with an import library.
- Call **FreeLibrary** when done with the DLL.

For example:

```
typedef UINT (CALLBACK* LPFNDLLFUNC1)(DWORD,UINT);
...

HINSTANCE hDLL;               // Handle to DLL
LPFNDLLFUNC1 lpfnDllFunc1;    // Function pointer
DWORD dwParam1;
UINT  uParam2, uReturnVal;

hDLL = LoadLibrary("MyDLL");
if (hDLL != NULL)
{
   lpfnDllFunc1 = (LPFNDLLFUNC1)GetProcAddress(hDLL,
                                          "DLLFunc1");
   if (!lpfnDllFunc1)
   {
      // handle the error
      FreeLibrary(hDLL);
      return SOME_ERROR_CODE;
   }
   else
   {
      // call the function
      uReturnVal = lpfnDllFunc1(dwParam1, uParam2);
   }
}
```

What do you want to do?

- Link implicitly
- Determine which linking method to use

What do you want to know more about?

- LoadLibrary and AfxLoadLibrary
- GetProcAddress
- FreeLibrary and AfxFreeLibrary
- The search path used by Windows to locate a DLL

See Also

Concepts

Linking an Executable to a DLL

Initializing a DLL

Typically, your DLL has initialization code (such as allocating memory) that must execute when your DLL loads. When using Visual C++, where you add code to initialize your DLL depends on the type of DLL you are building. If you do not need to add initialization or termination code, there is nothing special you have to do when building your DLL. If you need to initialize your DLL, the following table describes where to add your code.

DLL type	Where to add initialization and termination code
Regular DLL	In the DLL's **CWinApp** object's **InitInstance** and **ExitInstance**.
Extension DLL	In the **DllMain** function generated by the MFC DLL Wizard.
Non-MFC DLL	In a function called **DllMain** that you provide.

In Win32, all DLLs might contain an optional entry-point function (usually called **DllMain**) that is called for both initialization and termination. This gives you an opportunity to allocate or release additional resources as needed. Windows calls the entry-point function in four situations: process attach, process detach, thread attach, and thread detach.

The C run-time library provides an entry-point function called **_DllMainCRTStartup**, and it calls **DllMain**. Depending on the type of DLL, you should have a function called **DllMain** in your source code or you should use the **DllMain** provided in the MFC library.

What do you want to do?

- Initialize regular DLLs
- Initialize extension DLLs
- Initialize non-MFC DLLs

What do you want to know more about?

- The C run-time library behavior and _DllMainCRTStartup
- The function specification for DllMain (Platform SDK)
- Dynamic-link library entry-point function (Platform SDK)

See Also

Concepts

DLLs

Initializing Regular DLLs

Because regular DLLs have a **CWinApp** object, they should perform their initialization and termination tasks in the same location as an MFC application: in the **InitInstance** and **ExitInstance** member functions of the DLL's **CWinApp**-derived class. Because MFC provides a **DllMain** function that is called by **_DllMainCRTStartup** for **PROCESS_ATTACH** and **PROCESS_DETACH**, you should not write your own **DllMain** function. The MFC-provided **DllMain** function calls **InitInstance** when your DLL is loaded and it calls **ExitInstance** before the DLL is unloaded.

A regular DLL can keep track of multiple threads by calling TlsAlloc and TlsGetValue in its **InitInstance** function. These functions allow the DLL to track thread-specific data.

In your regular DLL that dynamically links to MFC, if you are using any MFC OLE, MFC Database (or DAO), or MFC Sockets support, respectively, the MFC debug extension DLLs MFCOxxD.dll, MFCDxxD.dll, and MFCNxxD.dll (where xx is the version number) are linked in automatically. You must call one of the following predefined initialization functions for each of these DLLs that you are using in your regular DLL's **CWinApp::InitInstance**.

Type of MFC support	Initialization function to call
MFC OLE (MFCOxxD.dll)	**AfxOleInitModule**
MFC Database (MFCDxxD.dll)	**AfxDbInitModule**
MFC Sockets (MFCNxxD.dll)	**AfxNetInitModule**

What do you want to do?

- Initialize extension DLLs
- Initialize non-MFC DLLs

What do you want to know more about?

- The C run-time library behavior and _DllMainCRTStartup
- Using Database, OLE, and Sockets Extension DLLs in Regular DLLs
- Processes and threads (Platform SDK)
- Thread local storage wrappers (MFC Technical Note 58)

See Also

Concepts

Initializing a DLL

Initializing Extension DLLs

Because extension DLLs do not have a **CWinApp**-derived object (as do regular DLLs), you should add your initialization and termination code to the **DllMain** function that the MFC DLL Wizard generates.

The wizard provides the following code for extension DLLs. In the code, `PROJNAME` is a placeholder for the name of your project.

```
#include "stdafx.h"
#include <afxdllx.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
static AFX_EXTENSION_MODULE PROJNAMEDLL = { NULL, NULL };

extern "C" int APIENTRY
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
   if (dwReason == DLL_PROCESS_ATTACH)
   {
      TRACE0("PROJNAME.DLL Initializing!\n");

      // Extension DLL one-time initialization
      AfxInitExtensionModule(PROJNAMEDLL,
                             hInstance);

      // Insert this DLL into the resource chain
      new CDynLinkLibrary(Dll3DLL);
   }
   else if (dwReason == DLL_PROCESS_DETACH)
   {
      TRACE0("PROJNAME.DLL Terminating!\n");
   }
   return 1;   // ok
}
```

Creating a new **CDynLinkLibrary** object during initialization allows the extension DLL to export **CRuntimeClass** objects or resources to the client application.

If you are going to use your extension DLL from one or more regular DLLs, you must export an initialization function that creates a **CDynLinkLibrary** object. That function must be called from each of the regular DLLs that use the extension DLL. An appropriate place to call this initialization function is in the **InitInstance** member function of the regular DLL's **CWinApp**-derived object before using any of the extension DLL's exported classes or functions.

In the **DllMain** that the MFC DLL Wizard generates, the call to **AfxInitExtensionModule** captures the module's run-time classes (**CRuntimeClass** structures) as well as its object factories (**COleObjectFactory** objects) for use when the **CDynLinkLibrary** object is created. You should check the return value of **AfxInitExtensionModule**; if a zero value is returned from **AfxInitExtensionModule**, return zero from your **DllMain** function.

If your extension DLL will be explicitly linked to an executable (meaning the executable calls **AfxLoadLibrary** to link to the DLL), you should add a call to **AfxTermExtensionModule** on **DLL_PROCESS_DETACH**. This function allows MFC to clean up the extension DLL when each process detaches from the extension DLL (which happens when the process exits or when the DLL is unloaded as a result of a **AfxFreeLibrary** call). If your extension DLL will be linked implicitly to the application, the call to **AfxTermExtensionModule** is not necessary.

Applications that explicitly link to extension DLLs must call **AfxTermExtensionModule** when freeing the DLL. They should also use **AfxLoadLibrary** and **AfxFreeLibrary** (instead of the Win32 functions **LoadLibrary** and **FreeLibrary**) if the application uses multiple threads. Using **AfxLoadLibrary** and **AfxFreeLibrary** ensures that the startup and shutdown code that executes when the extension DLL is loaded and unloaded does not corrupt the global MFC state.

Because the MFCx0.dll is fully initialized by the time **DllMain** is called, you can allocate memory and call MFC functions within **DllMain** (unlike the 16-bit version of MFC).

Extension DLLs can take care of multithreading by handling the **DLL_THREAD_ATTACH** and **DLL_THREAD_DETACH** cases in the **DllMain** function. These cases are passed to **DllMain** when threads attach and detach from the DLL. Calling TlsAlloc when a DLL is attaching allows the DLL to maintain thread local storage (TLS) indexes for every thread attached to the DLL.

Note that the header file Afxdllx.h contains special definitions for structures used in extension DLLs, such as the definition for **AFX_EXTENSION_MODULE** and **CDynLinkLibrary**. You should include this header file in your extension DLL.

Note
It is important that you neither define nor undefine any of the _AFX_NO_XXX macros in Stdafx.h. For more information, see the Knowledge Base article "PRB: Problems Occur When Defining _AFX_NO_XXX" (Q140751). You can find Knowledge Base articles in the MSDN Library or at http://search.support.microsoft.com/.

A sample initialization function that handles multithreading is included in Using Thread Local Storage in a Dynamic-Link Library in the *Platform SDK*. Note that the sample contains an entry-point function called **LibMain**, but you should name this function **DllMain** so that it works with the MFC and C run-time libraries.

The MFC sample DLLHUSK demonstrates the use of initialization functions.

What do you want to do?

- Initialize regular DLLs
- Initialize non-MFC DLLs

What do you want to know more about?

- The C run-time library behavior and _DllMainCRTStartup
- Using Database, OLE, and Sockets Extension DLLs in Regular DLLs
- The function specification for DllMain (Platform SDK)
- Dynamic-link library entry-point function (Platform SDK)

See Also

Concepts

Initializing a DLL

Initializing Non-MFC DLLs

To initialize non-MFC DLLs, your DLL source code must contain a function called **DllMain**. The following code presents a basic skeleton showing what the definition of **DllMain** might look like:

```
BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD  ul_reason_for_call,
                      LPVOID lpReserved)
{
    switch( ul_reason_for_call ) {
    case DLL_PROCESS_ATTACH:
    ...
    case DLL_THREAD_ATTACH:
    ...
    case DLL_THREAD_DETACH:
    ...
    case DLL_PROCESS_DETACH:
    ...
    }
    return TRUE;
}
```

Note

The Platform SDK documentation for **DllEntryPoint** says that the actual name of the entry-point function must be specified on the linker command-line with the /ENTRY option. With Visual C++, you do not need to use the /ENTRY option if the name of your entry-point function is **DllMain**. In fact, if you do use the /ENTRY option and name your entry-point function something other than **DllMain**, the C run-time library will not get initialized properly.

What do you want to know more about?

- The function specification for DllMain (Platform SDK)
- Dynamic-link library entry-point function (Platform SDK)
- The C run-time library behavior and _DllMainCRTStartup

See Also

Concepts

Initializing a DLL

Run-Time Library Behavior

The C/C++ run-time library code performs the DLL startup sequence, eliminating the need to link with a separate module as was necessary in Windows 3.x. Included in the C/C++ run-time library code is the DLL entry-point function called **_DllMainCRTStartup**. The **_DllMainCRTStartup** function does several things, including calling **_CRT_INIT**, which initializes the C/C++ run-time library and invokes C++ constructors on static, non-local variables. Without this function, the run-time library would be left in an uninitialized state. **_CRT_INIT** is available for both a statically linked CRT or linking to the CRT DLL Msvcrt.dll, from a user DLL.

While it is possible to specify another entry-point function using the /ENTRY: linker option, it is not recommended because your entry-point function would have to duplicate everything that **_DllMainCRTStartup** does. When building DLLs in Visual C++, **_DllMainCRTStartup** is linked in automatically and you do not need to specify an entry-point function using the /ENTRY: linker option.

In addition to initializing the C run-time library, **_DllMainCRTStartup** calls a function called **DllMain**. Depending on the kind of DLL you are building, Visual C++ provides **DllMain** for you and it gets linked in so that **_DllMainCRTStartup** always has something to call. In this way, if you do not need to initialize your DLL, there is nothing special you have to do when building your DLL. If you need to initialize your DLL, where you add your code depends on the kind of DLL you are writing. For more information, see Initializing a DLL.

The C/C++ run-time library code calls constructors and destructors on static, non-local variables. For example, in the following DLL source code, `Equus` and `Sugar` are two static, non-local objects of class `CHorse`, defined in Horses.h. There is no function in source code that contains calls to a constructor function for `CHorse` or to the destructor function because these objects are defined outside of any function. Therefore, calls to these constructors and destructors must be performed by the run-time code. The run-time library code for applications also performs this function.

```
#include "horses.h"

CHorse  Equus( ARABIAN, MALE );
CHorse  Sugar( THOROUGHBRED, FEMALE );

BOOL    WINAPI   DllMain (HANDLE hInst,
                          ULONG ul_reason_for_call,
                          LPVOID lpReserved)
...
```

Each time a new process attempts to use the DLL, the operating system creates a separate copy of the DLL's data: this is called process attach. The run-time library code for the DLL calls the constructors for all of the global objects, if any, and then calls the **DllMain** function with process attach selected. The opposite situation is process detach: the run-time library code calls **DllMain** with process detach selected and then calls a list of termination functions, including **atexit** functions, destructors for the global objects, and destructors for the static objects. Note that the order of events in process attach is the reverse of that in process detach.

The run-time library code is also called during thread attach and thread detach, but the run-time code does no initialization or termination on its own.

What do you want to do?

- Initialize a DLL

See Also

Concepts

DLLs

LoadLibrary and AfxLoadLibrary

Processes call **LoadLibrary** (or **AfxLoadLibrary**) to explicitly link to a DLL. If successful, the function maps the specified DLL into the address space of the calling process and returns a handle to the DLL that can be used with other functions used in explicit linking such as **GetProcAddress** and **FreeLibrary**.

LoadLibrary attempts to locate the DLL using the same search sequence used for implicit linking. If the system cannot find the DLL or if the entry-point function returns FALSE, **LoadLibrary** returns NULL. If the call to **LoadLibrary** specifies a DLL module already mapped into the address space of the calling process, the function simply returns a handle of the DLL and increments the module's reference count.

If the DLL has an entry-point function, the operating system calls the function in the context of the thread that called **LoadLibrary**. The entry-point function is not called if the DLL is already attached to the process because of a previous call to **LoadLibrary** with no corresponding call to the **FreeLibrary** function.

MFC applications loading extension DLLs should use **AfxLoadLibrary** instead of **LoadLibrary**. **AfxLoadLibrary** handles thread synchronization before calling **LoadLibrary**. The interface (function prototype) to **AfxLoadLibrary** is the same as **LoadLibrary**.

If for some reason Windows cannot load the DLL, the process can attempt to recover from the error. For example, the process could notify the user of the error and have the user specify another path to the DLL.

Security Note
If the code is to run under Windows NT 4, Windows 2000, or Windows XP (prior to SP1), make sure to specify the full path n ame of any DLLs. On these operating systems, the current directory is searched first when loading files. If you do not quality t he path to the file, a file can be loaded that was not intended.

What do you want to do?

- Link implicitly
- Determine which linking method to use

What do you want to know more about?

- GetProcAddress
- FreeLibrary
- The search path used by Windows to locate a DLL

See Also

Concepts

DLLs

GetProcAddress

Processes explicitly linking to a DLL call GetProcAddress to obtain the address of an exported function in the DLL. You use the returned function pointer to call the DLL function. **GetProcAddress** takes as parameters the DLL module handle (returned by either **LoadLibrary**, **AfxLoadLibrary**, or **GetModuleHandle**) and takes either the name of the function you want to call or the function's export ordinal.

Because you are calling the DLL function through a pointer and there is no compile-time type checking, make sure that the parameters to the function are correct so that you do not overstep the memory allocated on the stack and cause an access violation. One way to help provide type-safety is to look at the function prototypes of the exported functions and create matching typedefs for the function pointers. For example:

```
typedef UINT (CALLBACK* LPFNDLLFUNC1)(DWORD,UINT);
...

HINSTANCE hDLL;               // Handle to DLL
LPFNDLLFUNC1 lpfnDllFunc1;    // Function pointer
DWORD dwParam1;
UINT  uParam2, uReturnVal;

hDLL = LoadLibrary("MyDLL");
if (hDLL != NULL)
{
   lpfnDllFunc1 = (LPFNDLLFUNC1)GetProcAddress(hDLL,
                                        "DLLFunc1");
   if (!lpfnDllFunc1)
   {
      // handle the error
      FreeLibrary(hDLL);
      return SOME_ERROR_CODE;
   }
   else
   {
      // call the function
      uReturnVal = lpfnDllFunc1(dwParam1, uParam2);
   }
}
```

How you specify the function you want when calling **GetProcAddress** depends on how the DLL was built.

You can only obtain the export ordinal if the DLL you are linking to is built with a module definition (.def) file and if the ordinals are listed with the functions in the **EXPORTS** section of the DLL's .def file. Calling **GetProcAddress** with an export ordinal, as opposed to the function name, is slightly faster if the DLL has many exported functions because the export ordinals serve as indexes into the DLL's export table. With an export ordinal, **GetProcAddress** can locate the function directly as opposed to comparing the specified name to the function names in the DLL's export table. However, you should call **GetProcAddress** with an export ordinal only if you have control over assigning the ordinals to the exported functions in the .def file.

What do you want to do?

- Link implicitly
- Determine which linking method to use

What do you want to know more about?

- LoadLibrary and AfxLoadLibrary
- FreeLibrary

See Also

Concepts

DLLs

FreeLibrary and AfxFreeLibrary

Processes explicitly linking to a DLL call the **FreeLibrary** function when the DLL module is no longer needed. This function decrements the module's reference count and, if the reference count is zero, unmaps it from the address space of the process.

MFC applications should use **AfxFreeLibrary** instead of **FreeLibrary** to unload an extension DLL. The interface (function prototype) for **AfxFreeLibrary** is the same as **FreeLibrary**.

What do you want to do?

- Link implicitly
- Determine which linking method to use

What do you want to know more about?

- LoadLibrary and AfxLoadLibrary
- GetProcAddress

See Also

Concepts

DLLs

Search Path Used by Windows to Locate a DLL

With both implicit and explicit linking, Windows first searches for "known DLLs", such as Kernel32.dll and User32.dll. Windows then searches for the DLLs in the following sequence:

1. The directory where the executable module for the current process is located.
2. The current directory.
3. The Windows system directory. The **GetSystemDirectory** function retrieves the path of this directory.
4. The Windows directory. The **GetWindowsDirectory** function retrieves the path of this directory.
5. The directories listed in the PATH environment variable.

Note
The LIBPATH environment variable is not used.

What do you want to do?

- Link implicitly
- Link explicitly
- Determine which linking method to use

See Also

Concepts

DLLs

Module States of a Regular DLL Dynamically Linked to MFC

The ability to dynamically link a regular DLL to the MFC DLL allows some configurations that are very complicated. For example, a regular DLL and the executable that uses it can both dynamically link to the MFC DLL and to any extension DLLs.

This configuration poses a problem with regard to the MFC global data, such as the pointer to the current **CWinApp** object and handle maps.

Before MFC version 4.0, this global data resided in the MFC DLL itself and was shared by all the modules in the process. Because each process using a Win32 DLL gets its own copy of the DLL's data, this scheme provided an easy way to track per-process data. Also, because the AFXDLL model presumed that there would be only one **CWinApp** object and only one set of handle maps in the process, these items could be tracked in the MFC DLL itself.

But with the ability to dynamically link a regular DLL to the MFC DLL, it is now possible to have two or more **CWinApp** objects in a process — and also two or more sets of handle maps. How does MFC keep track of which ones it should be using?

The solution is to give each module (application or regular DLL) its own copy of this global state information. Thus, a call to **AfxGetApp** in the regular DLL returns a pointer to the **CWinApp** object in the DLL, not the one in the executable. This per-module copy of the MFC global data is known as a module state and is described in MFC Tech Note 58.

The MFC common window procedure automatically switches to the correct module state, so you do not need to worry about it in any message handlers implemented in your regular DLL. But when your executable calls into the regular DLL, you do need to explicitly set the current module state to the one for the DLL. To do this, use the **AFX_MANAGE_STATE** macro in every function exported from the DLL. This is done by adding the following line of code to the beginning of functions exported from the DLL:

```
AFX_MANAGE_STATE(AfxGetStaticModuleState( ))
```

What do you want to know more about?

- Managing the state data of MFC modules
- Regular DLLs dynamically linked to MFC
- Extension DLLs

See Also

Concepts

DLLs

Extension DLLs

An MFC extension DLL is a DLL that typically implements reusable classes derived from the existing Microsoft Foundation Class Library classes.

An MFC extension DLL has the following features and requirements:

- The client executable must be an MFC application compiled with **_AFXDLL** defined.
- An extension DLL can also be used by a regular DLL that is dynamically linked to MFC.
- Extension DLLs should be compiled with **_AFXEXT** defined. This forces **_AFXDLL** to be also defined and ensures that the proper declarations is pulled in from the MFC header files. It also ensures that **AFX_EXT_CLASS** is defined as **__declspec(dllexport)** while building the DLL, which is necessary if you are using this macro to declare the classes in your extension DLL.
- Extension DLLs should not instantiate a class derived from **CWinApp**, but should rely on the client application (or DLL) to provide this object.
- Extension DLLs should, however, provide a **DllMain** function and do any necessary initialization there.

Extension DLLs are built using the dynamic-link library version of MFC (also known as the shared version of MFC). Only MFC executables (either applications or regular DLLs) that are built with the shared version of MFC can use an extension DLL. Both the client application and the extension DLL must use the same version of MFCx0.dll. With an extension DLL, you can derive new custom classes from MFC and then offer this extended version of MFC to applications that call your DLL.

Extension DLLs can also be used for passing MFC-derived objects between the application and the DLL. The member functions associated with the passed object exist in the module where the object was created. Because these functions are properly exported when using the shared DLL version of MFC, you can freely pass MFC or MFC-derived object pointers between an application and the extension DLLs it loads.

An MFC extension DLL uses a shared version of MFC in the same way an application uses the shared DLL version of MFC, with a few additional considerations:

- It does not have a **CWinApp**-derived object. It must work with the **CWinApp**-derived object of the client application. This means that the client application owns the main message pump, the idle loop, and so on.
- It calls **AfxInitExtensionModule** in its **DllMain** function. The return value of this function should be checked. If a zero value is returned from **AfxInitExtensionModule**, return 0 from your **DllMain** function.
- It creates a **CDynLinkLibrary** object during initialization if the extension DLL wants to export **CRuntimeClass** objects or resources to the application.

Before version 4.0 of MFC, this type of DLL was called an AFXDLL. AFXDLL refers to the **_AFXDLL** preprocessor symbol that is defined when building the DLL.

The import libraries for the shared version of MFC are named according to the convention described in Naming conventions for MFC DLLs. Visual C++ supplies prebuilt versions of the MFC DLLs, plus a number of non-MFC DLLs that you can use and distribute with your applications. These are documented in Redist.txt, which is installed to the Program Files\Microsoft Visual Studio folder.

If you are exporting using a .def file, place the following code at the beginning and end of your header file:

```
#undef AFX_DATA
#define AFX_DATA AFX_EXT_DATA
// <body of your header file>
#undef AFX_DATA
#define AFX_DATA
```

These four lines ensure that your code is compiled correctly for an extension DLL. Leaving out these four lines might cause your DLL to either compile or link incorrectly.

If you need to pass an MFC or MFC-derived object pointer to or from an MFC DLL, the DLL should be an extension DLL. The member functions associated with the passed object exist in the module where the object was created. Because these functions are properly exported when using the shared DLL version of MFC, you can freely pass MFC or MFC-derived object pointers

between an application and the extension DLLs it loads.

Due to C++ name mangling and export issues, the export list from an extension DLL might be different between the debug and retail versions of the same DLL and DLLs for different platforms. The retail MFCx0.dll has about 2,000 exported entry points; the debug MFCx0D.dll has about 3,000 exported entry points.

Memory Management

MFCx0.dll and all extension DLLs loaded into a client application's address space use the same memory allocator, resource loading, and other MFC global states as if they were in the same application. This is significant because the non-MFC DLL libraries and the regular DLLs do the exact opposite and have each DLL allocating out of its own memory pool.

If an extension DLL allocates memory, that memory can freely intermix with any other application-allocated object. Also, if an application that dynamically links to MFC fails, the protection of the operating system maintains the integrity of any other MFC application sharing the DLL.

Similarly other global MFC states, like the current executable file to load resources from, are also shared between the client application and all MFC extension DLLs as well as MFCx0.dll itself.

Sharing Resources and Classes

Exporting resources is done through a resource list. Each application contains a singly linked list of **CDynLinkLibrary** objects. When looking for a resource, most of the standard MFC implementations that load resources look first at the current resource module (**AfxGetResourceHandle**) and if the resource is not found walk the list of **CDynLinkLibrary** objects attempting to load the requested resource.

Walking the list has the disadvantages that it is slightly slower and requires managing resource ID ranges. It has the advantage that a client application that links to several extension DLLs can use any DLL-provided resource without having to specify the DLL instance handle. **AfxFindResourceHandle** is an API used for walking the resource list to look for a given match. It takes the name and type of a resource and returns the resource handle where it was first found (or NULL).

If you do not want to walk the list and only load resources from a specific place, use the functions **AfxGetResourceHandle** and **AfxSetResourceHandle** to save the old handle and set the new handle. Be sure to restore the old resource handle before you return to the client application. For an example of using this approach to explicitly load a menu, see Testdll2 .cpp in the MFC sample DLLHUSK.

Dynamic creation of MFC objects given an MFC name is similar. The MFC object deserialization mechanism needs to have all of the **CRuntimeClass** objects registered so that it can reconstruct by dynamically creating C++ objects of the required type based on what was stored earlier.

In the case of the MFC sample DLLHUSK, the list looks something like:

```
head ->   DLLHUSK.EXE   - or -   DLLHUSK.EXE
               |                     |
          TESTDLL2.DLL          TESTDLL2.DLL
               |                     |
          TESTDLL1.DLL          TESTDLL1.DLL
               |                     |
           MFCOxxD.DLL               |
               |                     |
           MFCDxxD.DLL               |
               |                     |
            MFCxxD.DLL            MFCxx.DLL
```

where *xx* is the version number; for example, 42 represents version 4.2.

The MFCxx.dll is usually last on the resource and class list. MFCxx.dll includes all of the standard MFC resources, including prompt strings for all of the standard command IDs. Placing it at the end of the list allows DLLs and the client application itself not to have their own copy of the standard MFC resources, but to rely on the shared resources in the MFCxx.dll instead.

Merging the resources and class names of all DLLs into the client application's name space has the disadvantage of requiring you to be careful with what IDs or names you pick.

The DLLHUSK sample manages the shared resource name space by using multiple header files.

If your MFC extension DLL needs to maintain extra data for each application, you can derive a new class from **CDynLinkLibrary** and create it in **DllMain**. When running, the DLL can check the current application's list of **CDynLinkLibrary** objects to find the one for that particular extension DLL.

What do you want to do?

- Initialize an extension DLL

What do you want to know more about?

- Tips on using shared resource files
- DLL Version of MFC
- Regular DLLs statically linked to MFC
- Regular DLLs dynamically linked to MFC
- Using Database, OLE, and Sockets Extension DLLs in Regular DLLs

See Also

Concepts

DLLs

Using Database, OLE, and Sockets Extension DLLs in Regular DLLs

When using an extension DLL from a regular DLL, if the extension DLL is not wired into the **CDynLinkLibrary** object chain of the regular DLL, you might run into one or more of a set of related problems. Because the debug versions of the MFC Database, OLE, and Sockets support DLLs are implemented as extension DLLs, you might see similar problems if you are using these MFC features, even if you are not explicitly using any of your own extension DLLs. Some symptoms are:

- When attempting to deserialize an object of a type of class defined in the extension DLL, the message "Warning: Cannot load CYourClass from archive. Class not defined." appears in the TRACE debug window and the object fails to serialize.
- An exception indicating bad class might be thrown.
- Resources stored in the extension DLL fail to load because **AfxFindResourceHandle** returns **NULL** or an incorrect resource handle.
- **DllGetClassObject**, **DllCanUnloadNow**, and the **UpdateRegistry**, **Revoke**, **RevokeAll**, and **RegisterAll** member functions of **COleObjectFactory** fail to locate a class factory defined in the extension DLL.
- **AfxDoForAllClasses** does not work for any classes in the extension DLL.
- Standard MFC database, sockets, or OLE resources fail to load. For example, **AfxLoadString**(**AFX_IDP_SQL_CONNECT_FAIL**) returns an empty string, even when the regular DLL is properly using the MFC Database classes.

The solution to these problems is to create and export an initialization function in the extension DLL that creates a **CDynLinkLibrary** object. Call this initialization function exactly once from each regular DLL that uses the extension DLL.

MFC OLE, MFC Database (or DAO), or MFC Sockets Support

If you are using any MFC OLE, MFC Database (or DAO), or MFC Sockets support in your regular DLL, respectively, the MFC debug extension DLLs MFCOxxD.dll, MFCDxxD.dll, and MFCNxxD.dll (where xx is the version number) are linked automatically. You must call a predefined initialization function for each of these DLLs that you are using.

For database support, add a call to **AfxDbInitModule** to your regular DLL's **CWinApp::InitInstance** function. Make sure this call occurs before any base-class call or any added code which accesses the MFCDxxD.dll. This function takes no parameters and returns void.

For OLE support, add a call to **AfxOleInitModule** to your regular DLL's **CWinApp::InitInstance**. Note that the **COleControlModule InitInstance** function calls **AfxOleInitModule** already, so if you are building an OLE control and are using **COleControlModule**, you should not add this call to **AfxOleInitModule**.

For Sockets support, add a call to **AfxNetInitModule** to your regular DLL's **CWinApp::InitInstance**.

Note that release builds of MFC DLLs and applications do not use separate DLLs for database, sockets, or OLE support. However, it is safe to call these initialization functions in release mode.

CDynLinkLibrary Objects

During each of the operations mentioned at the beginning of this topic, MFC needs to search for a desired value or object. For example, during deserialization, MFC needs to search through all the currently available run-time classes to match objects in the archive with their proper run-time class.

As a part of these searches, MFC scans through all the extension DLLs in use by walking a chain of **CDynLinkLibrary** objects. **CDynLinkLibrary** objects attach automatically to a chain during their construction and are created by each extension DLL in turn during initialization. In addition, every module (application or regular DLL) has its own chain of **CDynLinkLibrary** objects.

For an extension DLL to get wired into a **CDynLinkLibrary** chain, it must create a **CDynLinkLibrary** object in the context of every module that uses the extension DLL. Therefore, if an extension DLL is going to be used from regular DLLs, it must provide an exported initialization function that creates a **CDynLinkLibrary** object. Every regular DLL that uses the extension DLL must call the exported initialization function.

If an extension DLL is only going to be used from an MFC application (.exe) and never from a regular DLL, then it is sufficient to create the **CDynLinkLibrary** object in the extension DLL's **DllMain**. This is what the MFC DLL Wizard extension DLL code does. When loading an extension DLL implicitly, **DllMain** loads and executes before the application ever starts. Any

CDynLinkLibrary creations are wired into a default chain that the MFC DLL reserves for an MFC application.

Note that it is a bad idea to have multiple **CDynLinkLibrary** objects from one extension DLL in any one chain, especially if the extension DLL will be dynamically unloaded from memory. Do not call the initialization function more than once from any one module.

Sample Code

This sample code assumes that the regular DLL is implicitly linking to the extension DLL. This is accomplished by linking to the import library (.lib) of the extension DLL when building the regular DLL.

The following lines should be in the source of the extension DLL:

```
// YourExtDLL.cpp:

// standard MFC extension DLL routines
#include "afxdllx.h"

static AFX_EXTENSION_MODULE NEAR extensionDLL = { NULL, NULL };

extern "C" int APIENTRY
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        // extension DLL one-time initialization
        if (!AfxInitExtensionModule(extensionDLL, hInstance))
           return 0;
    }
    return 1;   // ok
}

// Exported DLL initialization is run in context of
// application or regular DLL
extern "C" void WINAPI InitYourExtDLL()
{
    // create a new CDynLinkLibrary for this app
    new CDynLinkLibrary(extensionDLL);

    // add other initialization here
}
```

Be sure to export the **InitYourExtDLL** function. This could be done using **__declspec(dllexport)** or in your DLL's .def file as follows:

```
// YourExtDLL.Def:
LIBRARY      YOUREXTDLL
CODE         PRELOAD MOVEABLE DISCARDABLE
DATA         PRELOAD SINGLE
EXPORTS
    InitYourExtDLL
```

Add a call to the **InitInstance** member of the **CWinApp**-derived object in each regular DLL using the extension DLL:

```
// YourRegularDLL.cpp:

class CYourRegularDLL : public CWinApp
{
public:
    virtual BOOL InitInstance(); // Initialization
    virtual int ExitInstance();  // Termination

    // nothing special for the constructor
    CYourRegularDLL(LPCTSTR pszAppName) : CWinApp(pszAppName) { }
};
```

```
BOOL CYourRegularDLL::InitInstance()
{
    // any DLL initialization goes here
    TRACE0("YOUR regular DLL initializing\n");

    // wire any extension DLLs into CDynLinkLibrary chain
    InitYourExtDLL();

    return TRUE;
}
```

What do you want to do?

- Initialize an extension DLL
- Initialize regular DLLs

What do you want to know more about?

- Extension DLLs
- Regular DLLs Statically Linked to MFC
- Regular DLLs Dynamically Linked to MFC
- Using MFC as Part of a DLL
- DLL Version of MFC

See Also

Concepts

Extension DLLs

Creating a Resource-Only DLL

A resource-only DLL is a DLL that contains nothing but resources, such as icons, bitmaps, strings, and dialog boxes. Using a resource-only DLL is a good way to share the same set of resources among multiple programs. It is also a good way to provide an application with resources localized for multiple languages (see Localized Resources in MFC Applications: Satellite DLLs).

To create a resource-only DLL, you create a new Win32 DLL (non-MFC) project and add your resources to the project.

- Select Win32 Project in the **New Project** dialog box and specify a DLL project type in the Win32 Project Wizard.
- Create a new resource script that contains the resources (such as a string or a menu) for the DLL and save the .rc file.
- On the **Project** menu, click **Add Existing Item**, and then insert the new .rc file into the project.
- Specify the /NOENTRY linker option. /NOENTRY prevents the linker from linking a reference to _main into the DLL; this option is required to create a resource-only DLL.
- Build the DLL.

The application that uses the resource-only DLL should call **LoadLibrary** to explicitly link to the DLL. To access the resources, call the generic functions **FindResource** and **LoadResource**, which work on any kind of resource, or call one of the following resource-specific functions:

- **FormatMessage**
- **LoadAccelerators**
- **LoadBitmap**
- **LoadCursor**
- **LoadIcon**
- **LoadMenu**
- **LoadString**

The application should call **FreeLibrary** when it is finished using the resources.

What do you want to know more about?

- Editing Resources

See Also

Concepts

DLLs

Localized Resources in MFC Applications: Satellite DLLs

MFC version 7.0 and later provides enhanced support for satellite DLLs, a feature that helps in creating applications localized for multiple languages. A satellite DLL is a resource-only DLL that contains an application's resources localized for a particular language. When the application begins executing, MFC automatically loads the localized resource most appropriate for the environment. For example, you could have an application with English language resources with two satellite DLLs, one containing a French translation of your resources and the other containing a German translation. When the application is run on an English language system, it uses the English resources. If run on a French system, it uses the French resources; if run on a German system, it uses the German resources.

To support localized resources in an MFC application, MFC attempts to load a satellite DLL containing resources localized to a specific language. Satellite DLLs are named *ApplicationNameXXX*.dll, where *ApplicationName* is the name of the .exe or .dll using MFC, and *XXX* is the three-letter code for the language of the resources (for example, 'ENU' or 'DEU').

MFC attempts to load the resource DLL for each of the following languages in order, stopping when it finds one:

1. (Windows 2000 or later only) The current user's default UI language, as returned from the GetUserDefaultUILanguage() Win32 API.
2. (Windows 2000 or later only) The current user's default UI language, without any specific sublanguage (that is, ENC [Canadian English] becomes ENU [U.S. English]).
3. The system's default UI language. On Windows 2000 or later, this is returned from the GetSystemDefaultUILanguage() API. On other platforms, this is the language of the OS itself.
4. The system's default UI language, without any specific sublanguage.
5. A fake language with the 3-letter code LOC.

If MFC does not find any satellite DLLs, it uses whatever resources are contained in the application itself.

As an example, suppose that an application LangExample.exe uses MFC and is running on a Windows 2000 multiple user-interface system; the system UI language is ENU [U.S. English] and the current user's UI language is set to FRC [Canadian French]. MFC looks for the following DLLs in the following order:

1. LangExampleFRC.dll (user's UI language).
2. LangExampleFRA.dll (user's UI language without the sublanguage, in this example French (France).
3. LangExampleENU.dll (system's UI language).
4. LangExampleLOC.dll.

If none of these DLLs are found, MFC uses the resources in LangExample.exe.

See Also

Reference

TN057: Localization of MFC Components

Concepts

DLLs

Importing and Exporting

You can import public symbols into an application or export functions from a DLL using two methods:

- Use a module definition (.def) file when building the DLL
- Use the keywords **__declspec(dllimport)** or **__declspec(dllexport)** in a function definition in the main application

Using a .def file

A module-definition (.def) file is a text file containing one or more module statements that describe various attributes of a DLL. If you do not use **__declspec(dllimport)** or **__declspec(dllexport)** to export a DLL's functions, the DLL requires a .def file.

You can use .def files to import into an application or to export from a DLL.

Using __declspec

Visual C++ uses **__declspec(dllimport)** and **__declspec(dllexport)** to replace the **__export** keyword previously used in 16-bit versions of Visual C++.

You do not need to use **__declspec(dllimport)** for your code to compile correctly, but doing so allows the compiler to generate better code. The compiler is able to generate better code because it can determine whether a function exists in a DLL or not, which allows the compiler to produce code that skips a level of indirection that would normally be present in a function call that crossed a DLL boundary. However, you must use **__declspec(dllimport)** to import variables used in a DLL.

With the proper .def file EXPORTS section, **__declspec(dllexport)** is not required. **__declspec(dllexport)** was added to provide an easy way to export functions from an .exe or .dll file without using a .def file.

The Win32 Portable Executable format is designed to minimize the number of pages that must be touched to fix imports. To do this, it places all the import addresses for any program in one place called the Import Address Table. This allows the loader to modify only one or two pages when accessing these imports.

What do you want to do?

- Import into an Application
- Export from a DLL

See Also

Concepts

DLLs

Importing into an Application

You can import functions into an application using two methods:

- Use the keywords **__declspec(dllimport)** in a function definition in the main application
- Use a module definition (.def) file along with **__declspec(dllimport)**

What do you want to do?

- Import into an Application Using __declspec(dllimport)
- Import Function Calls Using __declspec(dllimport)
- Import Data Using __declspec(dllimport)
- Import Using DEF Files

See Also

Concepts

Importing and Exporting

Importing into an Application Using __declspec(dllimport)

A program that uses public symbols defined by a DLL is said to import them. When you create header files for applications that use your DLLs to build with, use **__declspec(dllimport)** on the declarations of the public symbols. The keyword **__declspec(dllimport)** works whether you export with .def files or with the **__declspec(dllexport)** keyword.

To make your code more readable, define a macro for **__declspec(dllimport)** and then use the macro to declare each imported symbol:

```
#define DllImport   __declspec( dllimport )

DllImport int  j;
DllImport void func();
```

Using **__declspec(dllimport)** is optional on function declarations, but the compiler produces more efficient code if you use this keyword. However, you must use **__declspec(dllimport)** for the importing executable to access the DLL's public data symbols and objects. Note that the users of your DLL still need to link with an import library.

You can use the same header file for both the DLL and the client application. To do this, use a special preprocessor symbol that indicates whether you are building the DLL or building the client application. For example:

```
#ifdef _EXPORTING
   #define CLASS_DECLSPEC    __declspec(dllexport)
#else
   #define CLASS_DECLSPEC    __declspec(dllimport)
#endif

class CLASS_DECLSPEC CExampleA : public CObject
{ ... class definition ... };
```

What do you want to do?

- Initialize a DLL

What do you want to know more about?

- Importing and exporting inline functions
- Mutual imports

See Also

Concepts

Importing into an Application

Importing Function Calls Using __declspec(dllimport)

The following code example shows how to use **_declspec(dllimport)** to import function calls from a DLL into an application. Assume that `func1` is a function that resides in a DLL separate from the .exe file that contains the **main** function.

Without **__declspec(dllimport)**, given this code:

```
int main(void)
{
   func1();
}
```

the compiler generates code that looks like this:

```
call func1
```

and the linker translates the call into something like this:

```
call 0x4000000         ; The address of 'func1'.
```

If `func1` exists in another DLL, the linker cannot resolve this directly because it has no way of knowing what the address of `func1` is. In 16-bit environments, the linker adds this code address to a list in the .exe file that the loader would patch at run time with the correct address. In 32-bit and 64-bit environments, the linker generates a thunk of which it does know the address. In a 32-bit environment the thunk looks like:

```
0x40000000:    jmp DWORD PTR __imp_func1
```

Here `imp_func1` is the address for the `func1` slot in the import address table of the .exe file. All the addresses are thus known to the linker. The loader only has to update the .exe file's import address table at load time for everything to work correctly.

Therefore, using **__declspec(dllimport)** is better because the linker does not generate a thunk if it is not required. Thunks make the code larger (on RISC systems, it can be several instructions) and can degrade your cache performance. If you tell the compiler the function is in a DLL, it can generate an indirect call for you.

So now this code:

```
__declspec(dllimport) void func1(void);
int main(void)
{
   func1();
}
```

generates this instruction:

```
call DWORD PTR __imp_func1
```

There is no thunk and no `jmp` instruction, so the code is smaller and faster.

On the other hand, for function calls inside a DLL, you do not want to have to use an indirect call. You already know a function's address. Because time and space are required to load and store the address of the function before an indirect call, a direct call is always faster and smaller. You only want to use **__declspec(dllimport)** when calling DLL functions from outside the DLL itself. Do not use **__declspec(dllimport)** on functions inside a DLL when building that DLL.

See Also

Concepts

Importing into an Application

Importing Data Using __declspec(dllimport)

In the case of data, using **__declspec(dllimport)** is a convenience item that removes a layer of indirection. When you import data from a DLL, you still have to go through the import address table. Before **__declspec(dllimport)**, this meant you had to remember to do an extra level of indirection when accessing data exported from the DLL:

```
// project.h
#ifdef _DLL   // If accessing the data from inside the DLL
   ULONG ulDataInDll;

#else         // If accessing the data from outside the DLL
   ULONG *ulDataInDll;
#endif
```

You would then export the data in your .DEF file:

```
// project.def
LIBRARY project
EXPORTS
   ulDataInDll   CONSTANT
```

and access it outside the DLL:

```
if (*ulDataInDll == 0L)
{
   // Do stuff here
}
```

When you mark the data as **__declspec(dllimport)**, the compiler automatically generates the indirection code for you. You no longer have to worry about the steps above. As stated previously, do not use **__declspec(dllimport)** declaration on the data when building the DLL. Functions within the DLL do not use the import address table to access the data object; therefore, you will not have the extra level of indirection present.

To export the data automatically from the DLL, use this declaration:

```
__declspec(dllexport) ULONG ulDataInDLL;
```

See Also

Concepts

Importing into an Application

Importing Using DEF Files

If you choose to use **__declspec(dllimport)** along with a .def file, you should change the .def file to use DATA in place of CONSTANT to reduce the likelihood that incorrect coding will cause a problem:

```
// project.def
LIBRARY project
EXPORTS
   ulDataInDll   DATA
```

The following table shows why.

Keyword	Emits in the import library	Exports
CONSTANT	_imp_ulDataInDll_ulDataInDll	_ulDataInDll
DATA	_imp_ulDataInDll	_ulDataInDll

Using **__declspec(dllimport)** and CONSTANT lists both the `imp` version and the undecorated name in the .lib DLL import library that is created to allow explicit linking. Using **__declspec(dllimport)** and DATA lists just the `imp` version of the name.

If you use CONSTANT, either of the following code constructs can be used to access `ulDataInDll`:

```
__declspec(dllimport) ULONG ulDataInDll; /*prototype*/
if (ulDataInDll == 0L)   /*sample code fragment*/
```

-or-

```
ULONG *ulDataInDll;      /*prototype*/
if (*ulDataInDll == 0L)  /*sample code fragment*/
```

However, if you use DATA in your .def file, only code compiled with the following definition can access the variable `ulDataInDll`:

```
__declspec(dllimport) ULONG ulDataInDll;

if (ulDataInDll == 0L)   /*sample code fragment*/
```

Using CONSTANT is more risky because if you forget to use the extra level of indirection, you could potentially access the import address table's pointer to the variable — not the variable itself. This type of problem can often manifest as an access violation because the import address table is currently made read-only by the compiler and linker.

The current Visual C++ linker issues a warning if it sees CONSTANT in the .def file to account for this case. The only real reason to use CONSTANT is if you cannot recompile some object file where the header file did not list **__declspec(dllimport)** on the prototype.

See Also

Concepts

Importing into an Application

Exporting from a DLL

A DLL file has a layout very similar to an .exe file, with one important difference — a DLL file contains an exports table. The exports table contains the name of every function that the DLL exports to other executables. These functions are the entry points into the DLL; only the functions in the exports table can be accessed by other executables. Any other functions in the DLL are private to the DLL. The exports table of a DLL can be viewed by using the DUMPBIN tool with the /EXPORTS option.

You can export functions from a DLL using two methods:

- Create a module definition (.def) file and use the .def file when building the DLL. Use this approach if you want to export functions from your DLL by ordinal rather than by name.
- Use the keyword **__declspec(dllexport)** in the function's definition.

When exporting functions with either method, make sure to use the __stdcall calling convention.

What do you want to do?

- Export from a DLL using .def files
- Export from a DLL using __declspec(dllexport)
- Export and import using AFX_EXT_CLASS
- Export C++ functions for use in C-language executables
- Export C functions for use in C or C++-language executables
- Export functions from a DLL by ordinal rather than by name
- Determine which exporting method to use
- Determine which linking method to use
- Initialize a DLL

What do you want to know more about?

- Importing into an application
- Importing and exporting inline functions
- Mutual imports

See Also

Concepts

Importing and Exporting

Exporting from a DLL Using DEF Files

A module-definition (.def) file is a text file containing one or more module statements that describe various attributes of a DLL. If you are not using the **__declspec(dllexport)** keyword to export the DLL's functions, the DLL requires a .def file.

A minimal .def file must contain the following module-definition statements:

- The first statement in the file must be the LIBRARY statement. This statement identifies the .def file as belonging to a DLL. The LIBRARY statement is followed by the name of the DLL. The linker places this name in the DLL's import library.
- The EXPORTS statement lists the names and, optionally, the ordinal values of the functions exported by the DLL. You assign the function an ordinal value by following the function's name with an at sign (@) and a number. When you specify ordinal values, they must be in the range 1 through N, where N is the number of functions exported by the DLL. If you want to export functions by ordinal, see Exporting Functions from a DLL by Ordinal Rather Than by Name as well as this topic.

For example, a DLL that contains the code to implement a binary search tree might look like the following:

```
LIBRARY   BTREE
EXPORTS
   Insert   @1
   Delete   @2
   Member   @3
   Min   @4
```

If you use the MFC DLL Wizard to create an MFC DLL, the wizard creates a skeleton .def file for you and automatically adds it to your project. Add the names of the functions to be exported to this file. For non-MFC DLLs, you must create the .def file yourself and add it to your project.

If you are exporting functions in a C++ file, you have to either place the decorated names in the .def file or define your exported functions with standard C linkage by using extern "C". If you need to place the decorated names in the .def file, you can obtain them by using the DUMPBIN tool or by using the linker /MAP option. Note that the decorated names produced by the compiler are compiler specific. If you place the decorated names produced by the Visual C++ compiler into a .def file, applications that link to your DLL must also be built using the same version of Visual C++ so that the decorated names in the calling application match the exported names in the DLL's .def file.

If you are building an extension DLL, and exporting using a .def file, place the following code at the beginning and end of your header files that contain the exported classes:

```
#undef AFX_DATA
#define AFX_DATA AFX_EXT_DATA
// <body of your header file>
#undef AFX_DATA
#define AFX_DATA
```

These lines ensure that MFC variables that are used internally or that are added to your classes are exported (or imported) from your extension DLL. For example, when deriving a class using **DECLARE_DYNAMIC**, the macro expands to add a **CRuntimeClass** member variable to your class. Leaving out these four lines might cause your DLL to compile or link incorrectly or cause an error when the client application links to the DLL.

When building the DLL, the linker uses the .def file to create an export (.exp) file and an import library (.lib) file. The linker then uses the export file to build the DLL file. Executables that implicitly link to the DLL link to the import library when they are built.

Note that MFC itself uses .def files to export functions and classes from the MFCx0.dll.

What do you want to do?

- Export from a DLL using __declspec(dllexport)
- Export and import using AFX_EXT_CLASS
- Export C++ functions for use in C-language executables
- Export C functions for use in C or C++-language executables

- Determine which exporting method to use
- Import into an application using __declspec(dllimport)
- Initialize a DLL

What do you want to know more about?

- .def files
- Rules for module-definition statements
- Decorated names
- Importing and exporting inline functions
- Mutual imports

See Also

Concepts

Exporting from a DLL

Exporting from a DLL Using __declspec(dllexport)

Microsoft introduced **__export** in the 16-bit compiler version of Visual C++ to allow the compiler to generate the export names automatically and place them in a .lib file. This .lib file can then be used just like a static .lib to link with a DLL.

In newer compiler versions, you can export data, functions, classes, or class member functions from a DLL using the **__declspec(dllexport)** keyword. **__declspec(dllexport)** adds the export directive to the object file so you do not need to use a .def file.

This convenience is most apparent when trying to export decorated C++ function names. Because there is no standard specification for name decoration, the name of an exported function might change between compiler versions. If you use **__declspec(dllexport)**, recompiling the DLL and dependent .exe files is necessary only to account for any naming convention changes.

Many export directives, such as ordinals, NONAME, and PRIVATE, can be made only in a .def file, and there is no way to specify these attributes without a .def file. However, using **__declspec(dllexport)** in addition to using a .def file does not cause build errors.

To export functions, the **__declspec(dllexport)** keyword must appear to the left of the calling-convention keyword, if a keyword is specified. For example:

```
__declspec(dllexport) void __cdecl Function1(void);
```

To export all of the public data members and member functions in a class, the keyword must appear to the left of the class name as follows:

```
class __declspec(dllexport) CExampleExport : public CObject
{ ... class definition ... };
```

When building your DLL, you typically create a header file that contains the function prototypes and/or classes you are exporting and add **__declspec(dllexport)** to the declarations in the header file. To make your code more readable, define a macro for **__declspec(dllexport)** and use the macro with each symbol you are exporting:

```
#define DllExport   __declspec( dllexport )
```

__declspec(dllexport) stores function names in the DLL's export table. If you want to optimize the table's size, see Exporting Functions from a DLL by Ordinal Rather Than by Name.

Note
When porting DLL source code from Win16 to Win32, replace each instance of **__export** with **__declspec(dllexport)**.

As a reference, search through the Win32 Winbase.h header file. It contains examples of **__declspec(dllimport)** usage.

What do you want to do?

- Export from a DLL using .def files
- Export and import using AFX_EXT_CLASS
- Export C++ functions for use in C-language executables
- Export C functions for use in C or C++-language executables
- Determine which exporting method to use
- Import into an application using __declspec(dllimport)
- Initialize a DLL

What do you want to know more about?

- The __declspec keyword
- Importing and exporting inline functions
- Mutual imports
- How to export data from a DLL
- How to share data in my DLL with an application or with other DLLs

See Also

Concepts

Exporting from a DLL

Exporting and Importing Using AFX_EXT_CLASS

Extension DLLs use the macro **AFX_EXT_CLASS** to export classes; the executables that link to the extension DLL use the macro to import classes. With the **AFX_EXT_CLASS** macro, the same header files that are used to build the extension DLL can be used with the executables that link to the DLL.

In the header file for your DLL, add the **AFX_EXT_CLASS** keyword to the declaration of your class as follows:

```
class AFX_EXT_CLASS CMyClass : public CDocument
{
// <body of class>
};
```

This macro is defined by MFC as **__declspec(dllexport)** when the preprocessor symbols **_AFXDLL** and **_AFXEXT** are defined. But the macro is defined as **__declspec(dllimport)** when **_AFXDLL** is defined and **_AFXEXT** is not defined. When defined, the preprocessor symbol **_AFXDLL** indicates that the shared version of MFC is being used by the target executable (either a DLL or an application). When both **_AFXDLL** and **_AFXEXT** are defined, this indicates that the target executable is an extension DLL.

Because **AFX_EXT_CLASS** is defined as **__declspec(dllexport)** when exporting from an extension DLL, you can export entire classes without placing the decorated names for all of that class's symbols in the .def file. This method is used by the MFC sample DLLHUSK.

Although you can avoid creating a .def file and all of the decorated names for the class with this method, creating a .def file is more efficient because the names can be exported by ordinal. To use the .def file method of exporting, place the following code at the beginning and end of your header file:

```
#undef AFX_DATA
#define AFX_DATA AFX_EXT_DATA
// <body of your header file>
#undef AFX_DATA
#define AFX_DATA
```

Caution

Be careful when exporting inline functions, because they can create the possibility of version conflicts. An inline function gets expanded into the application code; therefore, if you later rewrite the function, it does not get updated unless the application i tself is recompiled. Normally, DLL functions can be updated without rebuilding the applications that use them.

Exporting Individual Members in a Class

Sometimes you might want to export individual members of your class. For example, if you are exporting a **CDialog**-derived class, you might only need to export the constructor and the **DoModal** call. You can use **AFX_EXT_CLASS** on the individual members you need to export.

For example:

```
class CExampleDialog : public CDialog
{
public:
   AFX_EXT_CLASS CExampleDialog();
   AFX_EXT_CLASS int DoModal();
   ...
   // rest of class definition
   ...
};
```

Because you are no longer exporting all members of the class, you may run into an additional problem because of the way that MFC macros work. Several of MFC's helper macros actually declare or define data members. Therefore, these data members must also be exported from your DLL.

For example, the **DECLARE_DYNAMIC** macro is defined as follows when building an extension DLL:

```
#define DECLARE_DYNAMIC(class_name) \
protected: \
   static CRuntimeClass* PASCAL _GetBaseClass(); \
public: \
   static AFX_DATA CRuntimeClass class##class_name; \
   virtual CRuntimeClass* GetRuntimeClass() const; \
```

The line that begins with static **AFX_DATA** is declaring a static object inside of your class. To export this class correctly and access the run-time information from a client executable, you must export this static object. Because the static object is declared with the modifier **AFX_DATA**, you only need to define **AFX_DATA** to be **__declspec(dllexport)** when building your DLL and define it as **__declspec(dllimport)** when building your client executable. Because **AFX_EXT_CLASS** is already defined in this way, you just need to redefine **AFX_DATA** to be the same as **AFX_EXT_CLASS** around your class definition.

For example:

```
#undef  AFX_DATA
#define AFX_DATA AFX_EXT_CLASS

class CExampleView : public CView
{
   DECLARE_DYNAMIC()
   // ... class definition ...
};

#undef  AFX_DATA
#define AFX_DATA
```

Because MFC always uses the **AFX_DATA** symbol on data items it defines within its macros, this technique works for all such scenarios. For example, it works for **DECLARE_MESSAGE_MAP**.

Note
If you are exporting the entire class rather than selected members of the class, static data members are automatically exported.

What do you want to do?

- Export from a DLL using .def files
- Export from a DLL using __declspec(dllexport)
- Export C++ functions for use in C-language executables
- Export C functions for use in C or C++-language executables
- Determine which exporting method to use
- Import into an application using __declspec(dllimport)
- Initialize a DLL

What do you want to know more about?

- Decorated names
- Importing and exporting inline functions
- Mutual imports

See Also

Concepts

Exporting from a DLL

Exporting C++ Functions for Use in C-Language Executables

If you have functions in a DLL written in C++ that you want to access from a C-language module, you should declare these functions with C linkage instead of C++ linkage. Unless otherwise specified, the C++ compiler uses C++ type-safe naming (also known as name decoration) and C++ calling conventions, which can be difficult to call from C.

To specify C linkage, specify **extern** "**C**" for your function declarations. For example:

```
extern "C" __declspec( dllexport ) int MyFunc(long parm1);
```

What do you want to do?

- Export from a DLL using .def files
- Export from a DLL using __declspec(dllexport)
- Export and import using AFX_EXT_CLASS
- Export C functions for use in C or C++-language executables
- Determine which exporting method to use
- Import into an application using __declspec(dllimport)
- Initialize a DLL

What do you want to know more about?

- Decorated names
- Linkage specifications

See Also

Concepts

Exporting from a DLL

Exporting C Functions for Use in C or C++ Language Executables

If you have functions in a DLL written in C that you want to access from a C language or C++ language module, you should use the **__cplusplus** preprocessor macro to determine which language is being compiled, and then declare these functions with C linkage if being used from a C++ language module. If you use this technique and provide header files for your DLL, these functions can be used by C and C++ users with no change.

The following code shows a header file that can be used by C and C++ client applications:

```
// MyCFuncs.h
#ifdef __cplusplus
extern "C" {  // only need to export C interface if
              // used by C++ source code
#endif

__declspec( dllimport ) void MyCFunc();
__declspec( dllimport ) void AnotherCFunc();

#ifdef __cplusplus
}
#endif
```

If you need to link C functions to your C++ executable and the function declaration header files have not used the above technique, in the C++ source file, do the following to prevent the compiler from decorating the C function names:

```
extern "C" {
#include "MyCHeader.h"
}
```

What do you want to do?

- Export from a DLL using .def files
- Export from a DLL using __declspec(dllexport)
- Export and import using AFX_EXT_CLASS
- Determine which exporting method to use
- Import into an application using __declspec(dllimport)
- Initialize a DLL

What do you want to know more about?

- Decorated names
- Linkage specifications

See Also

Concepts

Exporting from a DLL

Determining Which Exporting Method to Use

To determine which method to use to export functions (a .def file or the **__declspec(dllexport)** keyword), answer the following questions:

- Will you be continuing to add additional exported functions?
- Who is using your DLL? For example, is it a third-party DLL used by many executables that you cannot rebuild or is the DLL used only by applications that you can easily rebuild?

Pros and Cons of Using .DEF Files

Exporting functions in a .def file gives you control over what the export ordinals are. When you add additional exported functions to your DLL, you can assign them higher ordinal values (higher than any other exported function). When you do this, applications using implicit linking do not have to relink with the new import library that contains the new functions. This is very important, for example, if you are designing a third-party DLL for use by many applications. You can continue to enhance your DLL by adding additional functionality while at the same time ensuring that existing applications continue to work properly with the new DLL. The MFC DLLs are built using .def files.

Another advantage to using a .def file is that you can export functions using the NONAME attribute, which places only the ordinal in the exports table in the DLL. For DLLs with a large number of exported functions, using the NONAME attribute can reduce the size of the DLL file. For information about writing a module definition statement, see Rules for Module-Definition Statements. For more information about ordinal export, see Exporting Functions from a DLL by Ordinal Rather Than by Name.

The major disadvantage of using .a def file is that if you are exporting functions in a C++ file, you either have to place the decorated names in the .def file or define your exported functions with standard C linkage by using extern "C" to avoid the name decoration done by the compiler.

If you need to place the decorated names in the .def file, you can obtain them by using the DUMPBIN tool or by using the linker /MAP option. Note that the decorated names produced by the compiler are compiler specific. If you place the decorated names produced by the Visual C++ compiler into a .def file, applications that link to your DLL must also be built using the same version of Visual C++ so that the decorated names in the calling application match the exported names in the DLL's .def file.

Pros and Cons of Using __declspec(dllexport)

Using **__declspec(dllexport)** is convenient because you do not have to worry about maintaining a .def file and obtaining the decorated names of the exported functions. This method is suitable if, for example, you are designing a DLL for use with an application that you control. If you rebuild the DLL with new exports, you also have to rebuild the application because the decorated names for exported C++ functions might change if you recompile with a different version of the compiler.

What do you want to do?

- Export from a DLL using .DEF files
- Export from a DLL using __declspec(dllexport)
- Export and import using AFX_EXT_CLASS
- Export C++ functions for use in C-language executables
- Export C functions for use in C or C++-language executables
- Import into an application using __declspec(dllimport)
- Initialize a DLL

What do you want to know more about?

- Importing and exporting inline functions
- Mutual imports
- Decorated names

See Also

Concepts

Exporting Functions from a DLL by Ordinal Rather Than by Name

The simplest way to export functions from your DLL is to export them by name. This is what happens when you use **__declspec(dllexport)**, for example. But you can instead export functions by ordinal. With this technique, you must use a .def file instead of **__declspec(dllexport)**. To specify a function's ordinal value, append its ordinal to the function name in the .def file. For information about specifying ordinals, see Exporting from a DLL Using .def Files.

Tip
If you want to optimize your DLL's file size, use the **NONAME** attribute on each exported function. With the **NONAME** attribute, the ordinals are stored in the DLL's export table rather than the function names. This can be a considerable savings if you are exporting many functions.

What do you want to do?

- Use a .def file so I can export by ordinal
- Use __declspec(dllexport)

See Also

Concepts

Exporting from a DLL

Mutual Imports

Exporting or importing to another executable file presents complications when the imports are mutual (or circular). For example, two DLLs import symbols from each other, similar to mutually recursive functions.

The problem with mutually importing executable files (usually DLLs) is that neither can be built without building the other first. Each build process requires, as input, an import library produced by the other build process.

The solution is to use the LIB utility with the /DEF option, which produces an import library without building the executable file. Using this utility, you can build all the import libraries you need, no matter how many DLLs are involved or how complicated the dependencies are.

The general solution for handling mutual imports is:

1. Take each DLL in turn. (Any order is feasible, although some orders are more optimal.) If all the needed import libraries exist and are current, run LINK to build the executable file (DLL). This produces an import library. Otherwise, run LIB to produce an import library.

   Running LIB with the /DEF option produces an additional file with an .EXP extension. The .EXP file must be used later to build the executable file.

2. After using either LINK or LIB to build all of the import libraries, go back and run LINK to build any executable files that were not built in the previous step. Note that the corresponding .exp file must be specified on the LINK line.

   If you had run the LIB utility earlier to produce an import library for DLL1, LIB would have produced the file DLL1.exp as well. You must use DLL1.exp as input to LINK when building DLL1.dlll.

The following illustration shows a solution for two mutually importing DLLs, DLL1 and DLL2. Step 1 is to run LIB, with the /DEF option set, on DLL1. Step 1 produces DLL1.lib, an import library, and DLL1.exp. In step 2, the import library is used to build DLL2, which in turn produces an import library for DLL2's symbols. Step 3 builds DLL1, by using DLL1.exp and DLL2.lib as input. Note that an .exp file for DLL2 is not necessary because LIB was not used to build DLL2's import library.

Linking Two DLLs with Mutual Imports

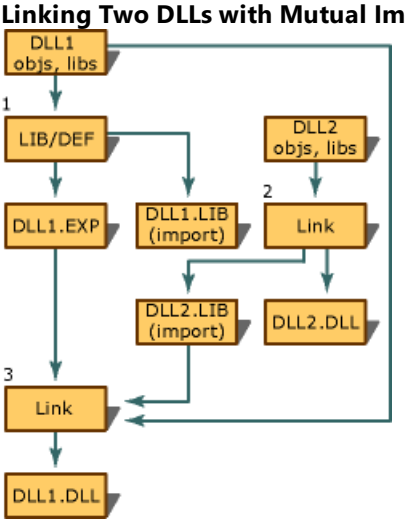

Limitations of _AFXEXT

You can use the **_AFXEXT** preprocessor symbol for your extension DLLs as long as you do not have multiple layers of extension DLLs. If you have extension DLLs that call or derive from classes in your own extension DLLs, which then derive from the MFC classes, you must use your own preprocessor symbol to avoid ambiguity.

The problem is that in Win32, you must explicitly declare any data as **__declspec(dllexport)** if it is to be exported from a DLL, and **__declspec(dllimport)** if it is to be imported from a DLL. When you define **_AFXEXT**, the MFC headers make sure that **AFX_EXT_CLASS** is defined correctly.

When you have multiple layers, one symbol such as **AFX_EXT_CLASS** is not sufficient, because an extension DLL might be exporting new classes as well as importing other classes from another extension DLL. To solve this problem, use a special preprocessor symbol that indicates that you are building the DLL itself versus using the DLL. For example, imagine two extension DLLs, A.dll and B.dll. They each export some classes in A.h and B.h, respectively. B.dll uses the classes from A.dll. The header files would look something like this:

```
/* A.H */
#ifdef A_IMPL
   #define CLASS_DECL_A   __declspec(dllexport)
#else
   #define CLASS_DECL_A   __declspec(dllimport)
#endif

class CLASS_DECL_A CExampleA : public CObject
{ ... class definition ... };

// B.H
#ifdef B_IMPL
   #define CLASS_DECL_B   __declspec(dllexport)
#else
   #define CLASS_DECL_B   __declspec(dllimport)
#endif

class CLASS_DECL_B CExampleB : public CExampleA
{ ... class definition ... };
...
```

When A.dll is built, it is built with `/D A_IMPL` and when B.dll is built, it is built with `/D B_IMPL`. By using separate symbols for each DLL, `CExampleB` is exported and `CExampleA` is imported when building B.dll. `CExampleA` is exported when building A.dll and imported when used by B.dll (or some other client).

This type of layering cannot be done when using the built-in **AFX_EXT_CLASS** and **_AFXEXT** preprocessor symbols. The technique described above solves this problem in a manner not unlike the mechanism MFC itself uses when building its Active technologies, Database, and Network extension DLLs.

Not Exporting the Entire Class

When you are not exporting an entire class, you have to ensure that the necessary data items created by the MFC macros are exported correctly. This can be done by redefining **AFX_DATA** to your specific class's macro. This should be done any time you are not exporting the entire class.

For example:

```
/* A.H */
#ifdef A_IMPL
   #define CLASS_DECL_A  _declspec(dllexport)
#else
   #define CLASS_DECL_A  _declspec(dllimport)
#endif

#undef  AFX_DATA
#define AFX_DATA CLASS_DECL_A

class CExampleA : public CObject
{
   DECLARE_DYNAMIC()
   CLASS_DECL_A int SomeFunction();
   //... class definition ...
};

#undef AFX_DATA
#define AFX_DATA
```

What do you want to do?

- Export from a DLL
- Export from a DLL using .DEF files
- Export from a DLL using __declspec(dllexport)
- Export and import using AFX_EXT_CLASS
- Export C++ functions for use in C-language executables

- Determine which exporting method to use
- Import into an application using __declspec(dllimport)

What do you want to know more about?

- The LIB utility and the /DEF option

See Also

Concepts

Importing and Exporting

Importing and Exporting Inline Functions

Imported functions can be defined as inline. The effect is roughly the same as defining a standard function inline; calls to the function are expanded into inline code, much like a macro. This is principally useful as a way of supporting C++ classes in a DLL that might inline some of their member functions for efficiency.

One feature of an imported inline function is that you can take its address in C++. The compiler returns the address of the copy of the inline function residing in the DLL. Another feature of imported inline functions is that you can initialize static local data of the imported function, unlike global imported data.

Caution
You should exercise care when providing imported inline functions because they can create the possibility of version conflicts. An inline function gets expanded into the application code; therefore, if you later rewrite the function, it does not get updated unless the application itself is recompiled. (Normally, DLL functions can be updated without rebuilding the applications that use them.)

What do you want to do?

- Export from a DLL
- Export from a DLL using .DEF files
- Export from a DLL using __declspec(dllexport)
- Export and import using AFX_EXT_CLASS
- Export C++ functions for use in C-language executables
- Determine which exporting method to use
- Import into an application using __declspec(dllimport)

See Also

Concepts

Importing and Exporting

Active Technology and DLLs

Active technology allows object servers to be completely implemented inside a DLL. This type of server is called an in-process server. MFC does not completely support in-process servers for all the features of visual editing, mainly because Active technology does not provide a way for a server to hook into the container's main message loop. MFC requires access to the container application's message loop to handle accelerator keys and idle-time processing.

If you are writing an Automation server and your server has no user interface, you can make your server an in-process server and put it completely into a DLL.

What do you want to know more about?

- Automation Servers

See Also

Concepts

DLLs

Automation in a DLL

When you choose the Automation option in the MFC DLL Wizard, the wizard provides you with the following:

- A starter object description language (.ODL) file
- An include directive in the STDAFX.h file for Afxole.h
- An implementation of the **DllGetClassObject** function, which calls the **AfxDllGetClassObject** function
- An implementation of the **DllCanUnloadNow** function, which calls the **AfxDllCanUnloadNow** function
- An implementation of the **DllRegisterServer** function, which calls the COleObjectFactory::UpdateRegistryAll function

What do you want to know more about?

- Automation Servers

See Also

Concepts

DLLs

Naming Conventions for MFC DLLs

The DLLs and libraries included in MFC follow a structured naming convention. This makes it easier to know which DLL or library you should be using for which purpose.

The import libraries needed to build applications or extension DLLs that use these DLLs have the same base name as the DLL but have a .lib file name extension.

Shared DLL Naming Convention

DLL	Description
MFCx0.DLL	MFC DLL, ANSI Release version
MFCx0U.DLL	MFC DLL, Unicode Release version
MFCx0D.DLL	MFC DLL, ANSI Debug version
MFCx0UD.DLL	MFC DLL, Unicode Debug version

If you are dynamically linking to the shared DLL version of MFC, whether it is from an application or from an extension DLL, you must include MFCx0.DLL with your product. If you require Unicode support in your application, include MFCx0U.DLL instead.

Note
The MFCx0.DLL and MFCx0U.DLL Retail version of the DLLs contain Active technologies, database, and network support in a single DLL. The Debug version maintains separate DLLs for these functional areas.

If you are statically linking your DLL to MFC, you must link it with one of the static MFC libraries. These versions are named according to the convention [N|U]AFXCW[D].LIB. For more information, see the table "Static-Link Library Naming Conventions" in Library Naming Conventions (MFC).

For a list of Visual C++ DLLs that can be distributed with your applications, see Redist.txt in your Visual Studio installation.

What do you want to know more about?

- The naming convention for libraries

See Also

Concepts

DLLs

Calling DLL Functions from Visual Basic Applications

For Visual Basic applications (or applications in other languages such as Pascal or Fortran) to call functions in a C/C++ DLL, the functions must be exported using the correct calling convention without any name decoration done by the compiler.

__stdcall creates the correct calling convention for the function (the called function cleans up the stack and parameters are passed from right to left) but decorates the function name differently. So, when **__declspec(dllexport)** is used on an exported function in a DLL, the decorated name is exported.

The **__stdcall** name decoration prefixes the symbol name with an underscore (_) and appends the symbol with an at sign (@) character followed by the number of bytes in the argument list (the required stack space). As a result, the function when declared as:

```
int __stdcall func (int a, double b)
```

is decorated as:

```
_func@12
```

The C calling convention (**__cdecl**) decorates the name as `_func`.

To get the decorated name, use /MAP. Use of **__declspec(dllexport)** does the following:

- If the function is exported with the C calling convention (**_cdecl**), it strips the leading underscore (_) when the name is exported.
- If the function being exported does not use the C calling convention (for example, **__stdcall**), it exports the decorated name.

Because there is no way to override where the stack cleanup occurs, you must use **__stdcall**. To undecorate names with **__stdcall**, you must specify them by using aliases in the EXPORTS section of the .def file. This is shown as follows for the following function declaration:

```
int  __stdcall MyFunc (int a, double b);
void __stdcall InitCode (void);
```

In the .DEF file:

```
EXPORTS
   MYFUNC=_MyFunc@12
   INITCODE=_InitCode@0
```

For DLLs to be called by programs written in Visual Basic, the alias technique shown in this topic is needed in the .def file. If the alias is done in the Visual Basic program, use of aliasing in the .def file is not necessary. It can be done in the Visual Basic program by adding an alias clause to the Declare statement.

What do you want to know more about?

- Exporting from a DLL
- Exporting from a DLL using .DEF files
- Exporting from a DLL using __declspec(dllexport)
- Exporting C++ functions for use in C-language executables
- Determining which exporting method to use
- Decorated names

See Also

Concepts

International Programming

An important aspect of developing applications for international markets is the adequate representation of local character sets. The ASCII character set defines characters in the range 0x00 to 0x7F. There are other character sets, primarily European, that define the characters within the range 0x00 to 0x7F identically to the ASCII character set and also define an extended character set from 0x80 to 0xFF. Thus, an 8-bit, single-byte-character set (SBCS) is sufficient to represent the ASCII character set, as well as the character sets for many European languages. However, some non-European character sets, such as Japanese Kanji, include many more characters than a single-byte coding scheme can represent, and therefore require multibyte-character set (MBCS) encoding.

In This Section

Unicode and MBCS

Discusses Visual C++ support for Unicode and MBCS programming.

Support for Unicode

Describes Unicode, a specification for supporting all character sets, including character sets that cannot be represented in a single byte.

Support for Multibyte Character Sets (MBCS)

Discusses MBCS, an alternative to Unicode for supporting character sets, like Japanese and Chinese, that cannot be represented in a single byte.

Generic-Text Mappings in Tchar.h

Provides Microsoft-specific generic-text mappings for many data types, routines, and other objects.

Related Sections

Internationalization

Discusses international support in the C run-time library.

International Samples

Provides links to samples demonstrating internationalization in Visual C++.

Language and Country/Region Strings

Provides the language and country/region strings in the C run-time library.

Adding Functionality

Provides links to topics describing conceptual information about the Visual C++ libraries and topics discussing various coding technologies and techniques.

Unicode and MBCS

The Microsoft Foundation Classes (MFC) library, the C run-time library for Visual C++, and the Visual C++ development environment are enabled to assist your international programming. They provide:

- Support for the Unicode standard on Windows 2000 (formerly Windows NT).

  Unicode is a 16-bit character encoding, providing enough encodings for all languages. All ASCII characters are included in Unicode as widened characters.

Note
The Unicode standard is not supported on Windows 95, Windows 98, or Windows Millennium Edition.

- Support for a form of multibyte character set (MBCS) called double-byte character set (DBCS) on all platforms.

  DBCS characters are composed of 1 or 2 bytes. Some ranges of bytes are set aside for use as lead bytes. A lead byte specifies that it and the following trail byte comprise a single 2-byte-wide character. You must keep track of which bytes are lead bytes. In a particular multibyte-character set, the lead bytes fall within a certain range, as do the trail bytes. When these ranges overlap, it might be necessary to evaluate the context to determine whether a given byte is functioning as a lead byte or a trail byte.

- Support for tools that simplify MBCS programming of applications written for international markets.

  When run on an MBCS-enabled version of the Windows operating system, the Visual C++ development system — including the integrated source code editor, debugger, and command-line tools — is completely MBCS-enabled. For more information, see MBCS Support in Visual C++.

Note
In this documentation, MBCS is used to describe all non-Unicode support for multibyte characters. In Visual C++, MBCS always means DBCS. Character sets wider than 2 bytes are not supported.

By definition, the ASCII character set is a subset of all multibyte-character sets. In many multibyte character sets, each character in the range 0x00 – 0x7F is identical to the character that has the same value in the ASCII character set. For example, in both ASCII and MBCS character strings, the 1-byte **NULL** character ('\0') has value 0x00 and indicates the terminating null character.

See Also

Concepts

International Programming
International Enabling

International Enabling

Most traditional C and C++ code makes assumptions about character and string manipulation that do not work well for international applications. While both MFC and the run-time library support Unicode or MBCS, there is still work for you to do. To guide you, this section explains the meaning of "international enabling" in Visual C++:

- Both Unicode and MBCS are enabled by means of portable data types in MFC function parameter lists and return types. These types are conditionally defined in the appropriate ways, depending on whether your build defines the symbol **_UNICODE** or the symbol **_MBCS** (which means DBCS). Different variants of the MFC libraries are automatically linked with your application, depending on which of these two symbols your build defines.
- Class library code uses portable run-time functions and other means to ensure correct Unicode or MBCS behavior.
- You still must handle certain kinds of internationalization tasks in your code:
  - Use the same portable run-time functions that make MFC portable under either environment.
  - Make literal strings and characters portable under either environment, using the **_T** macro. For more information, see Generic-Text Mappings in Tchar.h.
  - Take precautions when parsing strings under MBCS. These precautions are not needed under Unicode. For more information, see MBCS Programming Tips.
  - Take care if you mix ANSI (8-bit) and Unicode (16-bit) characters in your application. It is possible to use ANSI characters in some parts of your program and Unicode characters in others, but you cannot mix them in the same string.
  - Do not hard-code strings in your application. Instead, make them STRINGTABLE resources by adding them to the application's .rc file. Your application can then be localized without requiring source code changes or recompilation. For more information about STRINGTABLE resources, see String Editor.

Note
European and MBCS character sets have some characters, such as accented letters, with character codes greater than 0x80. Because most code uses signed characters, these characters greater than 0x80 are sign-extended when converted to **int**. This is a problem for array indexing because the sign-extended characters, being negative, indexes outside the array. Languages that use MBCS, such as Japanese, are also unique. Because a character might consist of 1 or 2 bytes, you should always manipulate both bytes at the same time.

See Also

Concepts

Unicode and MBCS

Internationalization Strategies

Internationalization Strategies

Depending on your target operating systems and markets, you have several internationalization strategies:

- Your application uses Unicode and therefore runs on Windows 2000 and Windows NT, but not on Windows 95 or Windows 98.

  You use Unicode-specific functionality and all characters are 16 bits wide (although you can use ANSI characters in some parts of your program for special purposes). The C run-time library provides functions, macros, and data types for Unicode-only programming. MFC is fully Unicode-enabled.

- Your application uses MBCS and can be run on any Win32 platform.

  You use MBCS-specific functionality. Strings can contain single-byte characters, double-byte characters, or both. The C run-time library provides functions, macros, and data types for MBCS-only programming. MFC is fully MBCS-enabled.

- The source code for your application is written for complete portability — by recompiling with the symbol **_UNICODE** or the symbol **_MBCS** defined, you can produce versions that use either. For more information, see Generic-Text Mappings in Tchar.h.

- Your application uses a wrapper library for missing Unicode functions on Windows 95, Windows 98, and Windows ME like the one described in Design a Single Unicode App that Runs on Both Windows 98 and Windows 2000. Wrapper libraries are also available commercially.

  You use fully portable C run-time functions, macros, and data types. MFC's flexibility supports any of these strategies.

The remainder of these topics focus on writing completely portable code that you can build as Unicode or as MBCS.

See Also

Concepts

Unicode and MBCS
Locales and Code Pages

Locales and Code Pages

A locale ID reflects the local conventions and language for a particular geographical region. A given language might be spoken in more than one Country/Region; for example, Portuguese is spoken in Brazil as well as in Portugal. Conversely, a Country/Region might have more than one official language. For example, Canada has two languages: English and French. Thus, Canada has two distinct locales: Canadian-English and Canadian-French. Some locale-dependent categories include the formatting of dates and the display format for monetary values.

The language determines the text and data formatting conventions, while the Country/Region determines the national conventions. Every language has a unique mapping, represented by code pages, which includes characters other than those in the alphabet (such as punctuation marks and numbers). A code page is a character set and is related to the language. As such, a locale is a unique combination of language, Country/Region, and code page. The locale and code page setting can be changed at run time by calling the setlocale function.

Different languages might use different code pages. For example, the ANSI code page 1252 is used for English and most European languages, and the ANSI code page 932 is used for Japanese Kanji. Virtually all code pages share the ASCII character set for the lowest 128 characters (0x00 to 0x7F).

Any single-byte code page can be represented in a table (with 256 entries) as a mapping of byte values to characters (including numbers and punctuation marks), or glyphs. Any multibyte code page can also be represented as a very large table (with 64K entries) of double-byte values to characters. In practice, however, it are usually represented as a table for the first 256 (single-byte) characters and as ranges for the double-byte values.

For more information about code pages, see Code Pages.

The C run-time library has two types of internal code pages: locale and multibyte. You can change the current code page during program execution (see the documentation for the setlocale and _setmbcp functions). Also, the run-time library might obtain and use the value of the operating system code page. In Windows 2000, the operating system code page is the "system default ANSI" code page. This code page is constant for the duration of the program's execution.

When the locale code page changes, the behavior of the locale-dependent set of functions changes to that dictated by the chosen code page. By default, all locale-dependent functions begin execution with a locale code page unique to the "C" locale. You can change the internal locale code page (as well as other locale-specific properties) by calling the **setlocale** function. A call to **setlocale**(LC_ALL, "") sets the locale to that indicated by the operating system user locale.

Similarly, when the multibyte code page changes, the behavior of the multibyte functions changes to that dictated by the chosen code page. By default, all multibyte functions begin execution with a multibyte code page corresponding to the operating system's default code page. You can change the internal multibyte code page by calling the **_setmbcp** function.

The C run-time function **setlocale** sets, changes, or queries some or all of the current program's locale information. The _wsetlocale routine is a wide-character version of **setlocale**; the arguments and return values of **_wsetlocale** are wide-character strings.

See Also

Concepts

Unicode and MBCS
Benefits of Character Set Portability

Benefits of Character Set Portability

You can benefit from using MFC and C run-time portability features even if you do not currently intend to internationalize your application:

- Coding portably makes your code base flexible. You can later move it easily to Unicode or MBCS.
- Using Unicode makes your applications for Windows 2000 more efficient. Because Windows 2000 uses Unicode, non-Unicode strings passed to and from the operating system must be translated, which incurs overhead.
- Using MBCS enables you to support international markets on Win32 platforms other than Windows 2000, such as Windows 95 or Windows 98.

See Also

Concepts

Unicode and MBCS
Support for Unicode

Support for Unicode

Unicode is a specification for supporting all character sets, including character sets that cannot be represented in a single byte. If you are programming for an international market, consider using either Unicode or multibyte character sets (MBCSs) or enabling your program so you can build it for either by changing a switch.

A wide character is a 2-byte multilingual character code. Most characters used in modern computing worldwide, including technical symbols and special publishing characters, can be represented according to the Unicode specification as a wide character. Characters that cannot be represented in 1 wide character can be represented in a Unicode pair with Unicode's surrogate feature. Because each wide character is always represented in a fixed size of 16 bits, using wide characters simplifies programming with international character sets.

A wide-character string is represented as a **wchar_t[]** array and is pointed to by a **wchar_t*** pointer. Any ASCII character can be represented as a wide character by prefixing the letter L to the character. For example, L'\0' is the terminating wide (16-bit) **NULL** character. Similarly, any ASCII string literal can be represented as a wide-character string literal by prefixing the letter L to the ASCII literal (L"Hello").

Generally, wide characters take more space in memory than multibyte characters but are faster to process. In addition, only one locale can be represented at a time in multibyte encoding, whereas all character sets in the world are represented simultaneously by the Unicode representation.

The MFC framework is Unicode-enabled throughout, except for the database classes. (ODBC is not Unicode-enabled.) MFC accomplishes Unicode enabling by using portable macros throughout, as shown in the following table.

Portable Data Types in MFC

Non-portable data type	Replaced by this macro
char	**_TCHAR**
char*, **LPSTR (Win32 data type)**	**LPTSTR**
const char*, LPCSTR (Win32 data type)	**LPCTSTR**

Class **CString** uses **_TCHAR** as its base and provides constructors and operators for easy conversions. Most string operations for Unicode can be written by using the same logic used for handling the Windows ANSI character set, except that the basic unit of operation is a 16-bit character instead of an 8-bit byte. Unlike working with multibyte character sets (MBCSs), you do not have to (and should not) treat a Unicode character as if it were two distinct bytes.

What do you want to do?

- Install Unicode support via MFC
- Enable Unicode in my program
- Enable both Unicode and MBCS in my program
- Use Unicode to create an internationalized program
- Learn the benefits of Unicode, including how using Unicode makes my program more efficient on Windows 2000
- Use wmain so I can pass wide-character arguments to my program
- See a summary of Unicode programming
- Learn about generic-text mappings for byte-width portability

See Also

Concepts

International Programming
Support for Using wmain

Support for Using wmain

Visual C++ supports defining a **wmain** function and passing wide-character arguments to your Unicode application. You declare formal parameters to **wmain**, using a format similar to **main**. You can then pass wide-character arguments and, optionally, a wide-character environment pointer to the program. The *argv* and *envp* parameters to **wmain** are of type **wchar_t***. For example:

```
wmain( int argc, wchar_t *argv[ ], wchar_t *envp[ ] )
```

Note

MFC Unicode applications use **wWinMain** as the entry point. In this case, **CWinApp::m_lpCmdLine** is a Unicode string. Be sure to set **wWinMainCRTStartup** with the /ENTRY linker option.

If your program uses a **main** function, the multibyte-character environment is created by the run-time library at program startup. A wide-character copy of the environment is created only when needed (for example, by a call to the **_wgetenv** or **_wputenv** functions). On the first call to **_wputenv**, or on the first call to **_wgetenv** if an MBCS environment already exists, a corresponding wide-character string environment is created. The environment is then pointed to by the **_wenviron** global variable, which is a wide-character version of the **_environ** global variable. At this point, two copies of the environment (MBCS and Unicode) exist simultaneously and are maintained by the run-time system throughout the life of the program.

Similarly, if your program uses a **wmain** function, a wide-character environment is created at program startup and is pointed to by the **_wenviron** global variable. An MBCS (ASCII) environment is created on the first call to **_putenv** or **getenv** and is pointed to by the **_environ** global variable.

See Also

Concepts

Support for Unicode

Unicode Programming Summary

Unicode Programming Summary

To take advantage of the MFC and C run-time support for Unicode, you need to:

- Define **_UNICODE**.

  Define the symbol **_UNICODE** before you build your program.

- Specify entry point.

  On the **Output** page of the Linker folder in the project's Property Pages dialog box, set the Entry Point symbol to **wWinMainCRTStartup**.

- Use portable run-time functions and types.

  Use the proper C run-time functions for Unicode string handling. You can use the **wcs** family of functions, but you might prefer the fully portable (internationally enabled) **_TCHAR** macros. These macros are all prefixed with **_tcs**; they substitute, one for one, for the **str** family of functions. These functions are described in detail in the Internationalization section of the *Run-Time Library Reference*. For more information, see Generic-Text Mappings in Tchar.h.

  Use **_TCHAR** and the related portable data types described in Support for Unicode.

- Handle literal strings properly.

  The Visual C++ compiler interprets a literal string coded as:

  ```
  L"this is a literal string"
  ```

  to mean a string of Unicode characters. You can use the same prefix for literal characters. Use the **_T** macro to code literal strings generically, so they compile as Unicode strings under Unicode or as ANSI strings (including MBCS) without Unicode. For example, instead of:

  ```
  pWnd->SetWindowText( "Hello" );
  ```

  use:

  ```
  pWnd->SetWindowText( _T("Hello") );
  ```

  With **_UNICODE** defined, **_T** translates the literal string to the L-prefixed form; otherwise, **_T** translates the string without the L prefix.

Tip
The **_T** macro is identical to the **_TEXT** macro.

- Be careful passing string lengths to functions.

  Some functions want the number of characters in a string; others want the number of bytes. For example, if **_UNICODE** is defined, the following call to a **CArchive** object will not work (`str` is a **CString**):

  ```
  archive.Write( str, str.GetLength( ) );    // invalid
  ```

  In a Unicode application, the length gives you the number of characters but not the correct number of bytes, because each character is 2 bytes wide. Instead, you must use:

  ```
  archive.Write( str, str.GetLength( ) * sizeof( _TCHAR ) );    // valid
  ```

  which specifies the correct number of bytes to write.

However, MFC member functions that are character-oriented, rather than byte-oriented, work without this extra coding:

```
pDC->TextOut( str, str.GetLength( ) );
```

CDC::TextOut takes a number of characters, not a number of bytes.

- Use fopen_s, _wfopen_s to open Unicode files.

To summarize, MFC and the run-time library provide the following support for Unicode programming under Windows 2000:

- Except for database class member functions, all MFC functions are Unicode-enabled, including **CString**. **CString** also provides Unicode/ANSI conversion functions.
- The run-time library supplies Unicode versions of all string-handling functions. (The run-time library also supplies portable versions suitable for Unicode or for MBCS. These are the **_tcs** macros.)
- Tchar.h supplies portable data types and the **_T** macro for translating literal strings and characters. For more information, see Generic-Text Mappings in Tchar.h.
- The run-time library provides a wide-character version of **main**. Use **wmain** to make your application Unicode-aware.

See Also

Concepts

Support for Unicode

Support for Multibyte Character Sets (MBCSs)

Multibyte character sets (MBCSs) are an alternative to Unicode for supporting character sets, like Japanese and Chinese, that cannot be represented in a single byte. If you are programming for an international market, consider using either Unicode or MBCS, or enabling your program so you can build it for either by changing a switch.

The most common MBCS implementation is double-byte character sets (DBCSs). Visual C++ in general, and MFC in particular, is fully enabled for DBCS.

For samples, see the MFC source code files.

For platforms used in markets whose languages use large character sets, the best alternative to Unicode is MBCS. MFC supports MBCS by using internationalizable data types and C run-time functions. You should do the same in your code.

Under MBCS, characters are encoded in either 1 or 2 bytes. In 2-byte characters, the first, or lead byte, signals that both it and the following byte are to be interpreted as one character. The first byte comes from a range of codes reserved for use as lead bytes. Which ranges of bytes can be lead bytes depends on the code page in use. For example, Japanese code page 932 uses the range 0x81 through 0x9F as lead bytes, but Korean code page 949 uses a different range.

Consider all the following in your MBCS programming.

MBCS characters in the environment

MBCS characters can appear in strings such as file and directory names.

Editing operations

Editing operations in MBCS applications should operate on characters, not bytes. The caret should not split a character, the RIGHT ARROW key should move right one character, and so on. **Delete** should delete a character; **Undo** should reinsert it.

String handling

In an application that uses MBCS, string handling poses special problems. Characters of both widths are mixed in a single string; therefore, you must remember to check for lead bytes.

Run-time library support

The C run-time library and MFC support single-byte, MBCS, and Unicode programming. Single-byte strings are processed with the *str* family of run-time functions, MBCS strings are processed with corresponding *_mbs* functions, and Unicode strings are processed with corresponding *wcs* functions. MFC class member function implementations use portable run-time functions that map, under the right circumstances, to the normal *str* family of functions, the MBCS functions, or the Unicode functions, as described in "MBCS/Unicode portability."

MBCS/Unicode portability

Using the Tchar.h header file, you can build single-byte, MBCS, and Unicode applications from the same sources. Tchar.h defines macros prefixed with *_tcs* , which map to *str*, *_mbs*, or *wcs* functions, as appropriate. To build MBCS, define the symbol **_MBCS**. To build Unicode, define the symbol **_UNICODE**. By default, **_MBCS** is defined for MFC applications. For more information, see Generic-Text Mappings in Tchar.h.

Note
Behavior is undefined if you define both **_UNICODE** and **_MBCS**.

The Mbctype.h and Mbstring.h header files define MBCS-specific functions and macros, which you might need in some cases. For example, **_ismbblead** tells you whether a specific byte in a string is a lead byte.

For international portability, code your program with Unicode or multibyte character sets (MBCSs).

What do you want to do?

- Enable MBCS in my program
- Enable both Unicode and MBCS in my program
- Use MBCS to create an internationalized program
- See a summary of MBCS programming

- Learn about generic-text mappings for byte-width portability

See Also

Concepts

International Programming
MBCS Support in Visual C++

MBCS Support in Visual C++

When run on an MBCS-enabled version of the Windows 2000 or Windows XP operating system, the Visual C++ development system (including the integrated source code editor, debugger, and command line tools) is MBCS-enabled, with the exception of the memory window.

The memory window does not interpret bytes of data as MBCS characters, even though it can interpret them as ANSI or Unicode characters. ANSI characters are always 1 byte in size and Unicode characters are 2 bytes in size. With MBCS, characters can be 1 or 2 bytes in size and their interpretation depends on which code page is in use. Because of this, it is difficult for the memory window to reliably display MBCS characters. The memory window cannot know which byte is the start of a character. The developer can view the byte values in the memory window and look up the value in tables to determine the character representation. This is possible because the developer knows the starting address of a string based on the source code.

Visual C++ accepts double-byte characters wherever it is appropriate to do so. This includes path names and file names in dialog boxes and text entries in the Visual C++ resource editor (for example, static text in the dialog editor and static text entries in the icon editor). In addition, the preprocessor recognizes some double-byte directives — for example, file names in **#include** statements, and as arguments to the **code_seg** and **data_seg** pragmas. In the source code editor, double-byte characters in comments and string literals are accepted, although not in C/C++ language elements (such as variable names).

Support for the Input Method Editor (IME)

Applications written for East Asian markets that use MBCS (for example, Japan) normally support the Windows IME for entering both single- and double-byte characters. The Visual C++ development environment contains full support for the IME. For more information, see IME Sample: Demonstrates How to Control IME Mode and Implement IME Level 3.

Japanese keyboards do not directly support Kanji characters. The IME converts a phonetic string, entered in one of the other Japanese alphabets (Romaji, Katakana, or Hiragana) into its possible Kanji representations. If there is ambiguity, you can select from several alternatives. When you have selected the intended Kanji character, the IME passes two **WM_CHAR** messages to the controlling application.

The IME, activated by the ALT+` key combination, appears as a set of buttons (an indicator) and a conversion window. The application positions the window at the text insertion point. The application must handle **WM_MOVE** and **WM_SIZE** messages by repositioning the conversion window to conform to the new location or size of the target window.

If you want users of your application to have the ability to enter Kanji characters, the application must handle Windows IME messages. For more information about IME programming, see Input Method Editor.

Visual C++ Debugger

The Visual C++ debugger provides the ability to set breakpoints on IME messages. In addition, the Memory window can display double-byte characters.

Command-Line Tools

The Visual C++ command-line tools, including the compiler, NMAKE, and the resource compiler (RC.EXE), are MBCS-enabled. You can use the resource compiler's /c option to change the default code page when compiling your application's resources.

To change the default locale at source code compile time, use #pragma setlocale.

Graphical Tools

The Visual C++ Windows-based tools, such as Spy++ and the resource editing tools, fully support IME strings.

See Also

Concepts

Support for Multibyte Character Sets (MBCSs)
MBCS Programming Tips

MBCS Programming Tips

This section supplies tips for successful multibyte character set (MBCS) programming. The advice applies to MFC applications and applications written without MFC. Topics include:

- General MBCS Programming Advice
- Incrementing and Decrementing Pointers
- Byte Indices
- Last Character in a String
- Character Assignment
- Character Comparison
- Buffer Overflow

See Also

Concepts

Support for Multibyte Character Sets (MBCSs)

General MBCS Programming Advice

Use the following tips:

- For flexibility, use run-time macros such as **_tcschr** and **_tcscpy** when possible. For more information, see Generic-Text Mappings in Tchar.h.
- Use the C run-time **_getmbcp** function to get information about the current code page.
- Do not reuse string resources. Depending on the target language, a given string might have a different meaning when translated. For example, "File" on the application's main menu might translate differently from the string "File" in a dialog box. If you need to use more than one string with the same name, use different string IDs for each.
- You might want to find out whether your application is running on an MBCS-enabled operating system. To do so, set a flag at program startup; do not rely on API calls.
- When designing dialog boxes, allow approximately 30% extra space at the end of static text controls for MBCS translation.
- Be careful when selecting fonts for your application, because some fonts are not available on all systems. For example, the Japanese version of Windows 2000 does not support the Helvetica font.
- When selecting the font for dialog boxes, use MS Shell Dlg instead of MS Sans Serif or Helvetica. MS Shell Dlg is replaced with the correct font by the system before creating the dialog box. Using MS Shell Dlg ensures that any changes in the operating system to deal with this font will automatically be available. (MFC replaces MS Shell Dlg with the DEFAULT_GUI_FONT or the System font on Windows 95, Windows 98, and Windows NT 4 because those systems do not handle MS Shell Dlg correctly.)
- When designing your application, decide which strings can be localized. If in doubt, assume that any given string will be localized. As such, do not mix strings that can be localized with those that cannot.

See Also

Concepts

MBCS Programming Tips
Incrementing and Decrementing Pointers

Incrementing and Decrementing Pointers

Use the following tips:

- Point to lead bytes, not trail bytes. It is usually unsafe to have a pointer to a trail byte. It is usually safer to scan a string forward rather than in reverse.
- There are pointer increment/decrement functions and macros available that move over a whole character:

  ```
  sz1++;
  ```

  becomes:

  ```
  sz1 = _mbsinc( sz1 );
  ```

  The **_mbsinc** and **_mbsdec** functions correctly increment and decrement in *character* units, regardless of the character size.
- For decrements, you need a pointer to the head of the string, as in the following:

  ```
  sz2--;
  ```

  becomes:

  ```
  sz2 = _mbsdec( sz2Head, sz2 );
  ```

  Alternatively, your head pointer could be to a valid character in the string, such that:

  ```
  sz2Head < sz2
  ```

  You must have a pointer to a known valid lead byte.
- You might want to maintain a pointer to the previous character for faster calls to **_mbsdec**.

See Also

Concepts

MBCS Programming Tips

Byte Indices

Byte Indices

Use the following tips:

- Working with a bytewise index into a string presents problems similar to those posed by pointer manipulation. Consider this example, which scans a string for a backslash character:

  ```
  while ( rgch[ i ] != '\\' )
      i++;
  ```

  This might index a trail byte, not a lead byte, and thus it might not point to a *character*.

- Use the _mbclen function to solve the preceding problem:

  ```
  while ( rgch[ i ] != '\\' )
      i += _mbclen ( rgch + i );
  ```

  This correctly indexes to a lead byte, hence to a *character*. The **_mbclen** function determines the size of a character (1 or 2 bytes).

See Also

Concepts

MBCS Programming Tips
The Last Character in a String

The Last Character in a String

Use the following tips:

- Trail byte ranges overlap the ASCII character set in many cases. You can safely use bytewise scans for any control characters (less than 32).
- Consider the following line of code, which might be checking to see if the last character in a string is a backslash character:

  ```
  if ( sz[ strlen( sz ) - 1 ] == '\\' )    // Is last character a '\'?
      // . . .
  ```

  Because **strlen** is not MBCS-aware, it returns the number of bytes, not the number of characters, in a multibyte string. Also, note that in some code pages (932, for example), '\' (0x5c) is a valid trail byte (`sz` is a C string).

  One possible solution is to rewrite the code this way:

  ```
  char *pLast;
  pLast = _mbsrchr( sz, '\\' );    // find last occurrence of '\' in sz
  if ( pLast && ( *_mbsinc( pLast ) == '\0' ) )
      // . . .
  ```

  This code uses the MBCS functions **_mbsrchr** and **_mbsinc**. Because these functions are MBCS-aware, they can distinguish between a '\' character and a trail byte '\'. The code performs some action if the last character in the string is a null ('\0').

See Also

Concepts

MBCS Programming Tips
Character Assignment

Character Assignment

Consider the following example, in which the **while** loop scans a string, copying all characters except 'X' into another string:

```
while( *sz2 )
{
    if( *sz2 != 'X' )
        *sz1++ = *sz2++;
    else
        sz2++;
}
```

The code copies the byte at `sz2` to the location pointed to by `sz1`, then increments `sz1` to receive the next byte. But if the next character in `sz2` is a double-byte character, the assignment to `sz1` copies only the first byte. The following code uses a portable function to copy the character safely and another to increment `sz1` and `sz2` correctly:

```
while( *sz2 )
{
    if( *sz2 != 'X' )
    {
        _mbscpy_s( sz1, 1, sz2 );
        sz1 = _mbsinc( sz1 );
        sz2 = _mbsinc( sz2 );
    }
    else
        sz2 = _mbsinc( sz2 );
}
```

See Also

Concepts

MBCS Programming Tips

Character Comparison

Character Comparison

Use the following tips:

- Comparing a known lead byte with an ASCII character works correctly:

```
if( *sz1 == 'A' )
```

- Comparing two unknown characters requires the use of one of the macros defined in Mbstring.h:

```
if( !_mbccmp( sz1, sz2) )
```

  This ensures that both bytes of a double-byte character are compared for equality.

See Also

Concepts

MBCS Programming Tips
Buffer Overflow

Buffer Overflow

Varying character sizes can cause problems when you put characters into a buffer. Consider the following code, which copies characters from a string, `sz`, into a buffer, `rgch`:

```
cb = 0;
while( cb < sizeof( rgch ) )
    rgch[ cb++ ] = *sz++;
```

The question is: Was the last byte copied a lead byte? The following does not solve the problem because it can potentially overflow the buffer:

```
cb = 0;
while( cb < sizeof( rgch ) )
{
    _mbccpy( rgch + cb, sz );
    cb += _mbclen( sz );
    sz = _mbsinc( sz );
}
```

The **_mbccpy** call attempts to do the correct thing — copy the full character, whether it is 1 or 2 bytes. But it does not take into account that the last character copied might not fit the buffer if the character is 2 bytes wide. The correct solution is:

```
cb = 0;
while( (cb + _mbclen( sz )) <= sizeof( rgch ) )
{
    _mbccpy( rgch + cb, sz );
    cb += _mbclen( sz );
    sz = _mbsinc( sz );
}
```

This code tests for possible buffer overflow in the loop test, using **_mbclen** to test the size of the current character pointed to by `sz`. By making a call to the **_mbsnbcpy** function, you can replace the code in the **while** loop with a single line of code. For example:

```
_mbsnbcpy( rgch, sz, sizeof( rgch ) );
```

See Also

Concepts

MBCS Programming Tips

Generic-Text Mappings in Tchar.h

To simplify transporting code for international use, the Microsoft run-time library provides Microsoft-specific generic-text mappings for many data types, routines, and other objects. You can use these mappings, which are defined in Tchar.h, to write generic code that can be compiled for single byte, multibyte, or Unicode, depending on a manifest constant you define using a **#define** statement. Generic-text mappings are Microsoft extensions that are not ANSI compatible.

Using the Tchar.h, you can build single-byte, MBCS, and Unicode applications from the same sources. Tchar.h defines macros prefixed with **_tcs**, which, with the correct preprocessor definitions, map to **str**, **_mbs**, or **wcs** function as appropriate. To build MBCS, define the symbol **_MBCS**. To build Unicode, define the symbol **_UNICODE**. To build a single-byte application, define neither (the default). By default, **_MBCS** is defined for MFC applications.

The **_TCHAR** data type is defined conditionally in Tchar.h. If the symbol **_UNICODE** is defined for your build, **_TCHAR** is defined as **wchar_t**; otherwise, for single-byte and MBCS builds, it is defined as **char**. (**wchar_t**, the basic Unicode wide character data type, is the 16-bit counterpart to an 8-bit signed **char**.) For international applications, use the **_tcs** family of functions, which operate in **_TCHAR** units, not bytes. For example, **_tcsncpy** copies *n* **_TCHAR**s, not *n* bytes.

Because some SBCS string-handling functions take (signed) **char*** parameters, a type mismatch compiler warning results when **_MBCS** is defined. There are three ways to avoid this warning, listed in order of efficiency:

1. Use the type-safe inline function thunks in Tchar.h. This is the default behavior.
2. Use the direct macros in Tchar.h by defining **_MB_MAP_DIRECT** on the command line. If you do this, you must manually match types. This is the fastest method, but is not type-safe.
3. Use the type-safe statically linked library function thunks in Tchar.h. To do so, define the constant **_NO_INLINING** on the command line. This is the slowest method, but the most type-safe.

Preprocessor Directives for Generic-Text Mappings

# define	Compiled version	Example
_UNICODE	Unicode (wide-character)	**_tcsrev** maps to **_wcsrev**
_MBCS	Multibyte-character	**_tcsrev** maps to **_mbsrev**
None (the default: neither **_UNICODE** nor **_MBCS** defined)	SBCS (ASCII)	**_tcsrev** maps to **strrev**

For example, the generic-text function **_tcsrev**, defined in Tchar.h, maps to **_mbsrev** if you defined **_MBCS** in your program or to **_wcsrev** if you defined **_UNICODE**. Otherwise **_tcsrev** maps to **strrev**. Other data type mappings are provided in Tchar.h for programming convenience, but **_TCHAR** is the most useful.

Generic-Text Data Type Mappings

Generic-Text Data Type Name	_UNICODE & _MBCS Not Defined	_MBCS Defined	_UNICODE Defined
_TCHAR	**char**	**char**	**wchar_t**
_TINT	**int**	**int**	**wint_t**
_TSCHAR	**signed char**	**signed char**	**wchar_t**
_TUCHAR	**unsigned char**	**unsigned char**	**wchar_t**
_TXCHAR	**char**	**unsigned char**	**wchar_t**
_T or **_TEXT**	No effect (removed by preprocessor)	No effect (removed by preprocessor)	**L** (converts following character or string to its Unicode counterpart)

For a complete list of generic-text mappings of routines, variables, and other objects, see Generic-Text Mappings in the *Run-Time Library Reference*.

Note

Do not use the *str* family of functions with Unicode strings, which are likely to contain embedded null bytes. Similarly, do not use the *wcs* family of functions with MBCS (or SBCS) strings.

The following code fragments illustrate the use of **_TCHAR** and **_tcsrev** for mapping to the MBCS, Unicode, and SBCS models.

```
_TCHAR *RetVal, *szString;
RetVal = _tcsrev(szString);
```

If **_MBCS** has been defined, the preprocessor maps this fragment to the code:

```
char *RetVal, *szString;
RetVal = _mbsrev(szString);
```

If **_UNICODE** has been defined, the preprocessor maps this fragment to the code:

```
wchar_t *RetVal, *szString;
RetVal = _wcsrev(szString);
```

If neither **_MBCS** nor **_UNICODE** has been defined, the preprocessor maps the fragment to single-byte ASCII code:

```
char *RetVal, *szString;
RetVal = strrev(szString);
```

Thus, you can write, maintain, and compile a single source code file to run with routines that are specific to any of the three kinds of character sets.

See Also

Concepts

International Programming

Using TCHAR.H Data Types with _MBCS Code

Using TCHAR.H Data Types with _MBCS Code

When the manifest constant **_MBCS** is defined, a given generic-text routine maps to one of the following kinds of routines:

- An SBCS routine that handles multibyte bytes, characters, and strings appropriately. In this case, the string arguments are expected to be of type **char***. For example, **_tprintf** maps to **printf**; the string arguments to **printf** are of type **char***. If you use the **_TCHAR** generic-text data type for your string types, the formal and actual parameter types for **printf** match because **_TCHAR*** maps to **char***.
- An MBCS-specific routine. In this case, the string arguments are expected to be of type **unsigned char***. For example, **_tcsrev** maps to **_mbsrev**, which expects and returns a string of type **unsigned char***. If you use the **_TCHAR** generic-text data type for your string types, there is a potential type conflict because **_TCHAR** maps to type **char**.

Following are three solutions for preventing this type conflict (and the C compiler warnings or C++ compiler errors that would result):

- Use the default behavior. Tchar.h provides generic-text routine prototypes for routines in the run-time libraries, as in the following example.

  ```
  char * _tcsrev(char *);
  ```

  In the default case, the prototype for **_tcsrev** maps to **_mbsrev** through a thunk in Libc.lib. This changes the types of the **_mbsrev** incoming parameters and outgoing return value from **_TCHAR *** (that is, **char ***) to **unsigned char ***. This method ensures type matching when you are using **_TCHAR**, but it is relatively slow due to the function call overhead.

- Use function inlining by incorporating the following preprocessor statement in your code.

  ```
  #define _USE_INLINING
  ```

  This method causes an inline function thunk, provided in Tchar.h, to map the generic-text routine directly to the appropriate MBCS routine. The following code excerpt from Tchar.h provides an example of how this is done.

  ```
  __inline char *_tcsrev(char *_s1)
  {return (char *)_mbsrev((unsigned char *)_s1);}
  ```

  If you can use inlining, this is the best solution, because it guarantees type matching and has no additional time cost.

- Use direct mapping by incorporating the following preprocessor statement in your code.

  ```
  #define _MB_MAP_DIRECT
  ```

  This approach provides a fast alternative if you do not want to use the default behavior or cannot use inlining. It causes the generic-text routine to be mapped by a macro directly to the MBCS version of the routine, as in the following example from Tchar.h.

  ```
  #define _tcschr _mbschr
  ```

  When you take this approach, you must be careful to ensure use of appropriate data types for string arguments and string return values. You can use type casting to ensure proper type matching or you can use the **_TXCHAR** generic-text data type. **_TXCHAR** maps to type **char** in SBCS code but maps to type **unsigned char** in MBCS code. For more information about generic-text macros, see Generic-Text Mappings in the *Run-Time Library Reference*.

See Also

Concepts

Generic-Text Mappings in Tchar.h

Multithreading

Visual C++ allows you to have multiple concurrent threads of execution running simultaneously. With multithreading, you can spin off background tasks, manage simultaneous streams of input, manage a user interface, and much more.

In This Section

Multithreading with C and Win32

Provides support for creating multithread applications with Microsoft Windows

Multithreading with C++ and MFC

Describes what processes and threads are and what the MFC approach to multithreading is.

Multithreading and Locales

Discusses issues that arise when using the locale functionality of both the C Runtime Library and the Standard C++ Library in a multithreaded application.

Related Sections

CWinThread

Represents a thread of execution within an application.

CSyncObject

Describes a pure virtual class that provides functionality common to the synchronization objects in Win32.

CSemaphore

Represents a semaphore, which is a synchronization object that allows a limited number of threads in one or more processes to access a resource.

CMutex

Represents a mutex, which is a synchronization object that allows one thread mutually exclusive access to a resource.

CCriticalSection

Represents a critical section, which is a synchronization object that allows one thread at a time to access a resource or section of code.

CEvent

Represents an event, which is a synchronization object that allows one thread to notify another that an event has occurred.

CMultiLock

Represents the access-control mechanism used in controlling access to resources in a multithreaded program.

CSingleLock

Represents the access-control mechanism used in controlling access to a resource in a multithreaded program.

Adding Functionality

Provides links to topics describing conceptual information about the Visual C++ libraries and topics discussing various coding technologies and techniques.

Multithreading with C and Win32

Microsoft Visual C++ provides support for creating multithread applications with Microsoft Windows: Windows XP, Windows 2000, Windows NT, Windows Me, and Windows 98. You should consider using more than one thread if your application needs to manage multiple activities, such as simultaneous keyboard and mouse input. One thread can process keyboard input while a second thread filters mouse activities. A third thread can update the display screen based on data from the mouse and keyboard threads. At the same time, other threads can access disk files or get data from a communications port.

With Visual C++, there are two ways to program with multiple threads: use the Microsoft Foundation Class (MFC) library or the C run-time library and the Win32 API. For information about creating multithread applications with MFC, see Multithreading with C++ and MFC after reading the following topics about multithreading in C.

These topics explain the features in Visual C++ that support the creation of multithread programs.

What do you want to know more about?

- What multithreading is about
- Library support for multithreading
- Include files for multithreading
- C Run-Time library functions for thread control
- Sample multithread program in C
- Writing a Multithread Win32 Program
- Compiling and linking multithread programs
- Avoiding problem areas with multithread programs
- Thread local storage (TLS)

See Also

Concepts

Multithreading

Multithread Programs

A thread is basically a path of execution through a program. It is also the smallest unit of execution that Win32 schedules. A thread consists of a stack, the state of the CPU registers, and an entry in the execution list of the system scheduler. Each thread shares all the process's resources.

A process consists of one or more threads and the code, data, and other resources of a program in memory. Typical program resources are open files, semaphores, and dynamically allocated memory. A program executes when the system scheduler gives one of its threads execution control. The scheduler determines which threads should run and when they should run. Threads of lower priority might have to wait while higher priority threads complete their tasks. On multiprocessor machines, the scheduler can move individual threads to different processors to balance the CPU load.

Each thread in a process operates independently. Unless you make them visible to each other, the threads execute individually and are unaware of the other threads in a process. Threads sharing common resources, however, must coordinate their work by using semaphores or another method of interprocess communication. For more information about synchronizing threads, see Writing a Multithreaded Win32 Program.

See Also

Concepts

Multithreading with C and Win32

Library Support for Multithreading

All versions of the CRT now support multi threading with the exception of the non-locking versions of some functions. See Multithreaded Libraries Performance for more information.

See C Run-Time Libraries for more information on CRT versions.

See Also

Concepts

Multithreading with C and Win32

Include Files for Multithreading

Standard include files declare C run-time library functions as they are implemented in the libraries. If you use the Full Optimization (/Ox) or fastcall Calling Convention (/Gr) option, the compiler assumes that all functions should be called using the register calling convention. The run-time library functions were compiled using the C calling convention, and the declarations in the standard include files tell the compiler to generate correct external references to these functions.

See Also

Concepts

Multithreading with C and Win32

C Run-Time Library Functions for Thread Control

All Win32 programs have at least one thread. Any thread can create additional threads. A thread can complete its work quickly and then terminate, or it can stay active for the life of the program.

The LIBCMT and MSVCRT C run-time libraries provide the following functions for thread creation and termination: _beginthread, _beginthreadex and _endthread, _endthreadex.

The **_beginthread** and **_beginthreadex** functions create a new thread and return a thread identifier if the operation is successful. The thread terminates automatically if it completes execution, or it can terminate itself with a call to **_endthread** or **_endthreadex**.

Note
If you are going to call C run-time routines from a program built with Libcmt.lib, you must start your threads with the **_beginthread** or **_beginthreadex** function. Do not use the Win32 functions **ExitThread** and **CreateThread**. Using **SuspendThread** can lead to a deadlock when more than one thread is blocked waiting for the suspended thread to complete its access to a C run-time data structure.

The _beginthread and _beginthreadex Functions

The **_beginthread** and **_beginthreadex** functions create a new thread. A thread shares the code and data segments of a process with other threads in the process but has its own unique register values, stack space, and current instruction address. The system gives CPU time to each thread, so that all threads in a process can execute concurrently.

_beginthread and **_beginthreadex** are similar to the CreateThread function in the Win32 API but has these differences:

- **_beginthread** and **_beginthreadex** let you pass multiple arguments to the thread.
- They initialize certain C run-time library variables. This is important only if you use the C run-time library in your threads.
- **CreateThread** helps provide control over security attributes. You can use this function to start a thread in a suspended state.

_beginthread and **_beginthreadex** return a handle to the new thread if successful or an error code if there was an error.

The _endthread and _endthreadex Functions

The _endthread function terminates a thread created by **_beginthread** (and similarly, **_endthreadex** terminates a thread created by **_beginthreadex**). Threads terminate automatically when they finish. **_endthread** and **_endthreadex** are useful for conditional termination from within a thread. A thread dedicated to communications processing, for example, can quit if it is unable to get control of the communications port.

See Also

Concepts

Multithreading with C and Win32

Sample Multithread C Program

Bounce.c is a sample multithread program that creates a new thread each time the letter **a** or **A** is typed. Each thread bounces a happy face of a different color around the screen. Up to 32 threads can be created. The program's normal termination occurs when **q** or **Q** is typed. For information about compiling and linking Bounce.c, see Compiling and Linking Multithread Programs.

Example

```
// sample_multithread_c_program.c
// compile with: /c
//
//  Bounce - Creates a new thread each time the letter 'a' is typed.
//  Each thread bounces a happy face of a different color around
//  the screen. All threads are terminated when the letter 'Q' is
//  entered.
//

#include <windows.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <process.h>

#define MAX_THREADS  32

// The function getrandom returns a random number between
// min and max, which must be in integer range.
#define getrandom( min, max ) (SHORT)((rand() % (int)(((max) + 1) - \
                               (min))) + (min))

int main( void );                    // Thread 1: main
void KbdFunc( void  );               // Keyboard input, thread dispatch
void BounceProc( void * MyID );      // Threads 2 to n: display
void ClearScreen( void );            // Screen clear
void ShutDown( void );               // Program shutdown
void WriteTitle( int ThreadNum );    // Display title bar information

HANDLE  hConsoleOut;                 // Handle to the console
HANDLE  hRunMutex;                   // "Keep Running" mutex
HANDLE  hScreenMutex;                // "Screen update" mutex
int     ThreadNr;                    // Number of threads started
CONSOLE_SCREEN_BUFFER_INFO csbiInfo; // Console information


int main() // Thread One
{
    // Get display screen information & clear the screen.
    hConsoleOut = GetStdHandle( STD_OUTPUT_HANDLE );
    GetConsoleScreenBufferInfo( hConsoleOut, &csbiInfo );
    ClearScreen();
    WriteTitle( 0 );

    // Create the mutexes and reset thread count.
    hScreenMutex = CreateMutex( NULL, FALSE, NULL );  // Cleared
    hRunMutex = CreateMutex( NULL, TRUE, NULL );      // Set
    ThreadNr = 0;

    // Start waiting for keyboard input to dispatch threads or exit.
    KbdFunc();

    // All threads done. Clean up handles.
    CloseHandle( hScreenMutex );
    CloseHandle( hRunMutex );
    CloseHandle( hConsoleOut );
}
```

```
void ShutDown( void ) // Shut down threads
{
    while ( ThreadNr > 0 )
    {
        // Tell thread to die and record its death.
        ReleaseMutex( hRunMutex );
        ThreadNr--;
    }

    // Clean up display when done
    WaitForSingleObject( hScreenMutex, INFINITE );
    ClearScreen();
}

void KbdFunc( void ) // Dispatch and count threads.
{
    int         KeyInfo;

    do
    {
        KeyInfo = _getch();
        if ( tolower( KeyInfo ) == 'a' &&
             ThreadNr < MAX_THREADS )
        {
            ThreadNr++;
            _beginthread( BounceProc, 0, &ThreadNr );
            WriteTitle( ThreadNr );
        }
    } while( tolower( KeyInfo ) != 'q' );

    ShutDown();
}

void BounceProc( void *pMyID )
{
    char    MyCell, OldCell;
    WORD    MyAttrib, OldAttrib;
    char    BlankCell = 0x20;
    COORD   Coords, Delta;
    COORD   Old = {0,0};
    DWORD   Dummy;
    char    *MyID = (char*)pMyID;

    // Generate update increments and initial
    // display coordinates.
    srand( (unsigned int) *MyID * 3 );

    Coords.X = getrandom( 0, csbiInfo.dwSize.X - 1 );
    Coords.Y = getrandom( 0, csbiInfo.dwSize.Y - 1 );
    Delta.X = getrandom( -3, 3 );
    Delta.Y = getrandom( -3, 3 );

    // Set up "happy face" & generate color
    // attribute from thread number.
    if( *MyID > 16)
        MyCell = 0x01;          // outline face
    else
        MyCell = 0x02;          // solid face
    MyAttrib =  *MyID & 0x0F;   // force black background

    do
    {
        // Wait for display to be available, then lock it.
        WaitForSingleObject( hScreenMutex, INFINITE );

        // If we still occupy the old screen position, blank it out.
        ReadConsoleOutputCharacter( hConsoleOut, &OldCell, 1,
```

```
                                Old, &Dummy );
        ReadConsoleOutputAttribute( hConsoleOut, &OldAttrib, 1,
                                Old, &Dummy );
        if (( OldCell == MyCell ) && (OldAttrib == MyAttrib))
            WriteConsoleOutputCharacter( hConsoleOut, &BlankCell, 1,
                                         Old, &Dummy );

        // Draw new face, then clear screen lock
        WriteConsoleOutputCharacter( hConsoleOut, &MyCell, 1,
                                     Coords, &Dummy );
        WriteConsoleOutputAttribute( hConsoleOut, &MyAttrib, 1,
                                     Coords, &Dummy );
        ReleaseMutex( hScreenMutex );

        // Increment the coordinates for next placement of the block.
        Old.X = Coords.X;
        Old.Y = Coords.Y;
        Coords.X += Delta.X;
        Coords.Y += Delta.Y;

        // If we are about to go off the screen, reverse direction
        if( Coords.X < 0 || Coords.X >= csbiInfo.dwSize.X )
        {
            Delta.X = -Delta.X;
            Beep( 400, 50 );
        }
        if( Coords.Y < 0 || Coords.Y > csbiInfo.dwSize.Y )
        {
            Delta.Y = -Delta.Y;
            Beep( 600, 50 );
        }
    }
    // Repeat while RunMutex is still taken.
    while ( WaitForSingleObject( hRunMutex, 75L ) == WAIT_TIMEOUT );
}

void WriteTitle( int ThreadNum )
{
    enum {
        sizeOfNThreadMsg = 80
    };
    char    NThreadMsg[sizeOfNThreadMsg];

    sprintf_s( NThreadMsg, sizeOfNThreadMsg,
               "Threads running: %02d.  Press 'A' "
               "to start a thread,'Q' to quit.", ThreadNum );
    SetConsoleTitle( NThreadMsg );
}

void ClearScreen( void )
{
    DWORD    dummy;
    COORD    Home = { 0, 0 };
    FillConsoleOutputCharacter( hConsoleOut, ' ',
                                csbiInfo.dwSize.X * csbiInfo.dwSize.Y,
                                Home, &dummy );
}
```

Input

```
a
q
```

See Also

Concepts

Multithreading with C and Win32

Writing a Multithreaded Win32 Program

When you write a program with multiple threads, you must coordinate their behavior and use of the program's resources. You must also make sure that each thread receives its own stack.

Sharing Common Resources Between Threads

Note

For a similar discussion from the MFC point of view, see Multithreading: Programming Tips and Multithreading: When to Use the Synchronization Classes.

Each thread has its own stack and its own copy of the CPU registers. Other resources, such as files, static data, and heap memory, are shared by all threads in the process. Threads using these common resources must be synchronized. Win32 provides several ways to synchronize resources, including semaphores, critical sections, events, and mutexes.

When multiple threads are accessing static data, your program must provide for possible resource conflicts. Consider a program where one thread updates a static data structure containing *x,y* coordinates for items to be displayed by another thread. If the update thread alters the *x* coordinate and is preempted before it can change the *y* coordinate, the display thread might be scheduled before the *y* coordinate is updated. The item would be displayed at the wrong location. You can avoid this problem by using semaphores to control access to the structure.

A mutex (short for *mut*ual *ex*clusion) is a way of communicating among threads or processes that are executing asynchronously of one another. This communication is usually used to coordinate the activities of multiple threads or processes, typically by controlling access to a shared resource by locking and unlocking the resource. To solve this *x,y* coordinate update problem, the update thread sets a mutex indicating that the data structure is in use before performing the update. It would clear the mutex after both coordinates had been processed. The display thread must wait for the mutex to be clear before updating the display. This process of waiting for a mutex is often called blocking on a mutex because the process is blocked and cannot continue until the mutex clears.

The Bounce.c program shown in Sample Multithread C Program uses a mutex named `ScreenMutex` to coordinate screen updates. Each time one of the display threads is ready to write to the screen, it calls **WaitForSingleObject** with the handle to `ScreenMutex` and constant **INFINITE** to indicate that the **WaitForSingleObject** call should block on the mutex and not time out. If `ScreenMutex` is clear, the wait function sets the mutex so other threads cannot interfere with the display and continues executing the thread. Otherwise, the thread blocks until the mutex clears. When the thread completes the display update, it releases the mutex by calling **ReleaseMutex**.

Screen displays and static data are only two of the resources requiring careful management. For example, your program might have multiple threads accessing the same file. Because another thread might have moved the file pointer, each thread must reset the file pointer before reading or writing. In addition, each thread must make sure that it is not preempted between the time it positions the pointer and the time it accesses the file. These threads should use a semaphore to coordinate access to the file by bracketing each file access with **WaitForSingleObject** and **ReleaseMutex** calls. The following code example illustrates this technique:

```
HANDLE    hIOMutex= CreateMutex (NULL, FALSE, NULL);

WaitForSingleObject( hIOMutex, INFINITE );
fseek( fp, desired_position, 0L );
fwrite( data, sizeof( data ), 1, fp );
ReleaseMutex( hIOMutex);
```

Thread Stacks

All of an application's default stack space is allocated to the first thread of execution, which is known as thread 1. As a result, you must specify how much memory to allocate for a separate stack for each additional thread your program needs. The operating system allocates additional stack space for the thread, if necessary, but you must specify a default value.

The first argument in the **_beginthread** call is a pointer to the **BounceProc** function, which executes the threads. The second argument specifies the default stack size for the thread. The last argument is an ID number that is passed to **BounceProc**. **BounceProc** uses the ID number to seed the random number generator and to select the thread's color attribute and display character.

Threads that make calls to the C run-time library or to the Win32 API must allow sufficient stack space for the library and API

functions they call. The C **printf** function requires more than 500 bytes of stack space, and you should have 2K of stack space available when calling Win32 API routines.

Because each thread has its own stack, you can avoid potential collisions over data items by using as little static data as possible. Design your program to use automatic stack variables for all data that can be private to a thread. The only global variables in the Bounce.c program are either mutexes or variables that never change after they are initialized.

Win32 also provides Thread-Local Storage (TLS) to store per-thread data. For more information, see Thread Local Storage.

See Also

Concepts

Multithreading with C and Win32

Compiling and Linking Multithread Programs

The Bounce.c program is introduced in Sample Multithread C Program.

Programs are compiled multithreaded by default.

To compile and link the multithread program Bounce.c from within the development environment

1. On the **File** menu, click **New**, and then click **Project**.
2. In the **Project Types** pane, click **Win32**.
3. In the **Templates** pane, click **Win32 Console Application**, and then name the project.
4. Add the file containing the C source code to the project.
5. On the **Build** menu, build the project by clicking the **Build** command.

To compile and link the multithread program Bounce.c from the command line

- Compile and link the program:

```
CL BOUNCE.C
```

See Also

Concepts

Multithreading with C and Win32

Avoiding Problem Areas with Multithread Programs

There are several problems you might encounter in creating, linking, or executing a multithread C program. Some of the more common problems are described in the following table. (For a similar discussion from the MFC point of view, see Multithreading: Programming Tips.)

Problem	Probable cause
You get a message box s howing that your progra m caused a protection vi olation.	Many Win32 programming errors cause protection violations. A common cause of protection violati ons is the indirect assignment of data to null pointers. Because this results in your program trying to access memory that does not belong to it, a protection violation is issued. An easy way to detect the cause of a protection violation is to compile your program with debugging information and then run it through the debugger in the Visual C++ environment. When the protecti on fault occurs, Windows transfers control to the debugger and the cursor is positioned on the line t hat caused the problem.
Your program generates numerous compile and li nk errors.	You can eliminate many potential problems by setting the compiler's warning level to one of its high est values and heeding the warning messages. By using the level 3 or level 4 warning level options, y ou can detect unintentional data conversions, missing function prototypes, and use of non-ANSI feat ures.

See Also

Concepts

Multithreading with C and Win32

Thread Local Storage (TLS)

Thread Local Storage (TLS) is the method by which each thread in a given multithreaded process can allocate locations in which to store thread-specific data. Dynamically bound (run-time) thread-specific data is supported by way of the TLS API (TlsAlloc, TlsGetValue, TlsSetValue, and TlsFree). Win32 and the Visual C++ compiler now support statically bound (load-time) per-thread data in addition to the existing API implementation.

API Implementation for TLS

Thread Local Storage is implemented through the Win32 API layer and the compiler. For more information, see the Win32 API documentation for TlsAlloc, TlsGetValue, TlsSetValue, and TlsFree.

The Visual C++ compiler includes a keyword to make TLS operations more automatic, rather than through the API layer. This syntax is described in the next section, Compiler Implementation for TLS.

Compiler Implementation for TLS

To support TLS, a new attribute, **thread**, has been added to the C and C++ languages and is supported by the Visual C++ compiler. This attribute is an extended storage class modifier, as described in the previous section. Use the **__declspec** keyword to declare a **thread** variable. For example, the following code declares an integer thread local variable and initializes it with a value:

```
__declspec( thread ) int tls_i = 1;
```

See Also

Concepts

Multithreading with C and Win32
Rules and Limitations for TLS

Rules and Limitations for TLS

The following guidelines must be observed when declaring statically bound thread local objects and variables:

- The **thread** attribute can be applied only to data declarations and definitions. It cannot be used on function declarations or definitions. For example, the following code generates a compiler error:

```
#define Thread  __declspec( thread )
Thread void func();     // This will generate an error.
```

- The **thread** modifier might be specified only on data items with **static** extent. This includes global data objects (both **static** and **extern**), local static objects, and static data members of C++ classes. Automatic data objects cannot be declared with the **thread** attribute. The following code generates compiler errors:

```
#define Thread  __declspec( thread )
void func1()
{
    Thread int tls_i;            // This will generate an error.
}

int func2( Thread int tls_i )    // This will generate an error.
{
    return tls_i;
}
```

- The declarations and the definition of a thread local object must all specify the **thread** attribute. For example, the following code generates an error:

```
#define Thread  __declspec( thread )
extern int tls_i;        // This will generate an error, since the
int Thread tls_i;        // declaration and definition differ.
```

- The **thread** attribute cannot be used as a type modifier. For example, the following code generates a compiler error:

```
char __declspec( thread ) *ch;        // Error
```

- C++ classes cannot use the **thread** attribute. However, C++ class objects can be instantiated with the **thread** attribute. For example, the following code generates a compiler error:

```
#define Thread  __declspec( thread )
class Thread C       // Error: classes cannot be declared Thread.
{
// Code
};
C CObject;
```

  Because the declaration of C++ objects that use the **thread** attribute is permitted, the following two examples are semantically equivalent:

```
#define Thread  __declspec( thread )
Thread class B
{
// Code
} BObject;               // OK--BObject is declared thread local.
```

```
class B
{
// Code
};
Thread B BObject;        // OK--BObject is declared thread local.
```

- The address of a thread local object is not considered constant, and any expression involving such an address is not considered a constant expression. In standard C, the effect of this is to forbid the use of the address of a thread local variable as an initializer for an object or pointer. For example, the following code is flagged as an error by the C compiler:

```
#define Thread  __declspec( thread )
Thread int tls_i;
int *p = &tls_i;       //This will generate an error in C.
```

  This restriction does not apply in C++, however. Because C++ permits dynamic initialization of all objects, you can initialize an object with an expression that uses the address of a thread local variable. This is accomplished in the same way as the construction of thread local objects. For example, the code shown above does not generate an error when compiled as a C++ source file. Note that the address of a thread local variable is valid only as long as the thread in which the address was taken still exists.

- Standard C permits the initialization of an object or variable with an expression involving a reference to itself, but only for objects of nonstatic extent. Although C++ normally permits such dynamic initialization of objects with an expression involving a reference to itself, this type of initialization is not permitted with thread local objects. For example:

```
#define Thread  __declspec( thread )
Thread int tls_i = tls_i;                // Error in C and C++
int j = j;                               // OK in C++, error in C
Thread int tls_i = sizeof( tls_i )       // Legal in C and C++
```

  Note that a `sizeof` expression that includes the object being initialized does not constitute a reference to itself and is legal in both C and C++.

  C++ does not allow such dynamic initialization of thread data because of possible future enhancements to the thread local storage facility.

- If a DLL declares any nonlocal data or object as **__declspec**(thread), it can cause a protection fault if dynamically loaded. After the DLL is loaded with LoadLibrary, it causes system failure whenever the code references the nonlocal **__declspec**(thread) data. Because the global variable space for a thread is allocated at run time, the size of this space is based on a calculation of the requirements of the application plus the requirements of all the DLLs that are statically linked. When you use **LoadLibrary**, there is no way to extend this space to allow for the thread local variables declared with **__declspec**(thread). Use the TLS APIs, such as TlsAlloc, in your DLL to allocate TLS if the DLL might be loaded with **LoadLibrary**.

See Also

Concepts

Thread Local Storage (TLS)

Multithreading with C++ and MFC

The Microsoft Foundation Class (MFC) library provides support for multithreaded applications. This topic describes processes and threads and the MFC approach to multithreading.

A process is an executing instance of an application. For example, when you double-click the Notepad icon, you start a process that runs Notepad.

A thread is a path of execution within a process. When you start Notepad, the operating system creates a process and begins executing the primary thread of that process. When this thread terminates, so does the process. This primary thread is supplied to the operating system by the startup code in the form of a function address. Usually, it is the address of the **main** or **WinMain** function that is supplied.

You can create additional threads in your application if you want. You might want to do this to handle background or maintenance tasks when you do not want the user to wait for them to complete. All threads in MFC applications are represented by CWinThread objects. In most situations, you do not even have to explicitly create these objects; instead call the framework helper function AfxBeginThread, which creates the **CWinThread** object for you.

MFC distinguishes two types of threads: user-interface threads and worker threads. User-interface threads are commonly used to handle user input and respond to events and messages generated by the user. Worker threads are commonly used to complete tasks, such as recalculation, that do not require user input. The Win32 API does not distinguish between types of threads; it just needs to know the thread's starting address so it can begin to execute the thread. MFC handles user-interface threads specially by supplying a message pump for events in the user interface. **CWinApp** is an example of a user-interface thread object, because it derives from **CWinThread** and handles events and messages generated by the user.

Special attention should be given to situations where more than one thread might require access to the same object. Multithreading: Programming Tips describes techniques that you can use to get around problems that might arise in these situations. Multithreading: How to Use the Synchronization Classes describes how to use the classes that are available to synchronize access from multiple threads to a single object.

Writing and debugging multithreaded programming is inherently a complicated and tricky undertaking, because you must ensure that objects are not accessed by more than one thread at a time. The multithreading topics do not teach the basics of multithreaded programming, only how to use MFC in your multithreaded program. The multithreaded MFC samples included in Visual C++ illustrate a few multithreaded Adding Functionality and Win32 APIs not encompassed by MFC; however, they are only intended to be a starting point.

For more information about how the operating system handles processes and threads, see Processes and Threads in the Platform SDK.

For more information about MFC multithreading support, see the following topics:

- Multithreading: Creating User-Interface Threads
- Multithreading: Creating Worker Threads
- Multithreading: How to Use the Synchronization Classes
- Multithreading: Terminating Threads
- Multithreading: Programming Tips
- Multithreading: When to Use the Synchronization Classes

See Also

Concepts

Multithreading

Multithreading: Creating User-Interface Threads

A user-interface thread is commonly used to handle user input and respond to user events independently of threads executing other portions of the application. The main application thread (provided in your **CWinApp**-derived class) is already created and started for you. This topic describes the steps necessary to create additional user-interface threads.

The first thing you must do when creating a user-interface thread is derive a class from CWinThread. You must declare and implement this class, using the DECLARE_DYNCREATE and IMPLEMENT_DYNCREATE macros. This class must override some functions and can override others. These functions and what they should do are presented in the following table.

Functions to Override When Creating a User-Interface Thread

Function	Purpose
ExitInstance	Perform cleanup when thread terminates. Usually overridden.
InitInstance	Perform thread instance initialization. Must be overridden.
OnIdle	Perform thread-specific idle-time processing. Not usually overridden.
PreTranslateMessage	Filter messages before they are dispatched to **TranslateMessage** and **DispatchMessage**. Not usu ally overridden.
ProcessWndProcException	Intercept unhandled exceptions thrown by the thread's message and command handlers. Not usual ly overridden.
Run	Controlling function for the thread. Contains the message pump. Rarely overridden.

MFC provides two versions of **AfxBeginThread** through parameter overloading: one for user-interface threads and the other for worker threads. To start your user-interface thread, call AfxBeginThread, providing the following information:

- The RUNTIME_CLASS of the class you derived from **CWinThread**.
- (Optional) The desired priority level. The default is normal priority. For more information about the available priority levels, see SetThreadPriority in the *Platform SDK*.
- (Optional) The desired stack size for the thread. The default is the same size stack as the creating thread.
- (Optional) **CREATE_SUSPENDED** if you want the thread to be created in a suspended state. The default is 0, or start the thread normally.
- (Optional) The desired security attributes. The default is the same access as the parent thread. For more information about the format of this security information, see SECURITY_ATTRIBUTES in the *Platform SDK*.

AfxBeginThread does most of the work for you. It creates a new object of your class, initializes it with the information you supply, and calls CWinThread::CreateThread to start executing the thread. Checks are made throughout the procedure to make sure all objects are deallocated properly should any part of the creation fail.

What do you want to know more about?

- Multithreading: Terminating Threads
- Multithreading: Creating Worker Threads
- Processes and Threads

See Also

Concepts

Multithreading with C++ and MFC

Multithreading: Creating Worker Threads

A worker thread is commonly used to handle background tasks that the user should not have to wait for to continue using your application. Tasks such as recalculation and background printing are good examples of worker threads. This topic details the steps necessary to create a worker thread. Topics include:

- Starting the thread
- Implementing the controlling function
- Example

Creating a worker thread is a relatively simple task. Only two steps are required to get your thread running: implementing the controlling function and starting the thread. It is not necessary to derive a class from CWinThread. You can derive a class if you need a special version of **CWinThread**, but it is not required for most simple worker threads. You can use **CWinThread** without modification.

Starting the Thread

There are two overloaded versions of **AfxBeginThread**: one for user-interface threads and one for worker threads. To begin execution of your worker thread, call AfxBeginThread, providing the following information:

- The address of the controlling function.
- The parameter to be passed to the controlling function.
- (Optional) The desired priority of the thread. The default is normal priority. For more information about the available priority levels, see SetThreadPriority in the *Platform SDK*.
- (Optional) The desired stack size for the thread. The default is the same size stack as the creating thread.
- (Optional) **CREATE_SUSPENDED** if you want the thread to be created in a suspended state. The default is 0, or start the thread normally.
- (Optional) The desired security attributes. The default is the same access as the parent thread. For more information about the format of this security information, see SECURITY_ATTRIBUTES in the *Platform SDK*.

AfxBeginThread creates and initializes a **CWinThread** object for you, starts it, and returns its address so you can refer to it later. Checks are made throughout the procedure to make sure all objects are deallocated properly should any part of the creation fail.

Implementing the Controlling Function

The controlling function defines the thread. When this function is entered, the thread starts, and when it exits, the thread terminates. This function should have the following prototype:

```
UINT MyControllingFunction( LPVOID pParam );
```

The parameter is a single value. The value the function receives in this parameter is the value that was passed to the constructor when the thread object was created. The controlling function can interpret this value in any manner it chooses. It can be treated as a scalar value or a pointer to a structure containing multiple parameters, or it can be ignored. If the parameter refers to a structure, the structure can be used not only to pass data from the caller to the thread, but also to pass data back from the thread to the caller. If you use such a structure to pass data back to the caller, the thread needs to notify the caller when the results are ready. For information about communicating from the worker thread to the caller, see Multithreading: Programming Tips.

When the function terminates, it should return a **UINT** value indicating the reason for termination. Typically, this exit code is 0 to indicate success with other values indicating different types of errors. This is purely implementation dependent. Some threads might maintain usage counts of objects and return the current number of uses of that object. To see how applications can retrieve this value, see Multithreading: Terminating Threads.

There are some restrictions on what you can do in a multithreaded program written with the MFC library. For descriptions of these restrictions and other tips about using threads, see Multithreading: Programming Tips.

Controlling Function Example

The following example shows how to define a controlling function and use it from another portion of the program.

```
UINT MyThreadProc( LPVOID pParam )
{
    CMyObject* pObject = (CMyObject*)pParam;

    if (pObject == NULL ||
        !pObject->IsKindOf(RUNTIME_CLASS(CMyObject)))
    return 1;   // if pObject is not valid

    // do something with 'pObject'

    return 0;   // thread completed successfully
}

// inside a different function in the program
.
.
.
pNewObject = new CMyObject;
AfxBeginThread(MyThreadProc, pNewObject);
.
.
.
```

What do you want to know more about?

- Multithreading: Creating User-Interface Threads

See Also

Concepts

Multithreading with C++ and MFC

Multithreading: When to Use the Synchronization Classes

The multithreaded classes provided with MFC fall into two categories: synchronization objects (CSyncObject, CSemaphore, CMutex, CCriticalSection, and CEvent) and synchronization access objects (CMultiLock and CSingleLock).

Synchronization classes are used when access to a resource must be controlled to ensure integrity of the resource. Synchronization access classes are used to gain access to these controlled resources. This topic describes when to use each class.

To determine which synchronization class you should use, ask the following series of questions:

1. Does the application have to wait for something to happen before it can access the resource (for example, data must be received from a communications port before it can be written to a file)?

   If yes, use **CEvent**.

2. Can more than one thread within the same application access this resource at one time (for example, your application allows up to five windows with views on the same document)?

   If yes, use **CSemaphore**.

3. Can more than one application use this resource (for example, the resource is in a DLL)?

   If yes, use **CMutex**.

   If no, use **CCriticalSection**.

CSyncObject is never used directly. It is the base class for the other four synchronization classes.

Example 1: Using Three Synchronization Classes

As an example, take an application that maintains a linked list of accounts. This application allows up to three accounts to be examined in separate windows, but only one can be updated at any particular time. When an account is updated, the updated data is sent over the network to a data archive.

This example application uses all three types of synchronization classes. Because it allows up to three accounts to be examined at one time, it uses **CSemaphore** to limit access to three view objects. When an attempt to view a fourth account occurs, the application either waits until one of the first three windows closes or it fails. When an account is updated, the application uses **CCriticalSection** to ensure that only one account is updated at a time. After the update succeeds, it signals **CEvent**, which releases a thread waiting for the event to be signaled. This thread sends the new data to the data archive.

Example 2: Using Synchronization Access Classes

Choosing which synchronization access class to use is even simpler. If your application is concerned with accessing a single controlled resource only, use **CSingleLock**. If it needs access to any one of a number of controlled resources, use **CMultiLock**. In example 1, **CSingleLock** would have been used, because in each case only one resource is needed at any particular time.

For information about how the synchronization classes are used, see Multithreading: How to Use the Synchronization Classes. For information about synchronization, see Synchronization in the *Platform SDK*. For information about multithreading support in MFC, see Multithreading with C++ and MFC.

See Also

Concepts

Multithreading with C++ and MFC

Multithreading: How to Use the Synchronization Classes

Synchronizing resource access between threads is a common problem when writing multithreaded applications. Having two or more threads simultaneously access the same data can lead to undesirable and unpredictable results. For example, one thread could be updating the contents of a structure while another thread is reading the contents of the same structure. It is unknown what data the reading thread will receive: the old data, the newly written data, or possibly a mixture of both. MFC provides a number of synchronization and synchronization access classes to aid in solving this problem. This topic explains the classes available and how to use them to create thread-safe classes in a typical multithreaded application.

A typical multithreaded application has a class that represents a resource to be shared among threads. A properly designed, fully thread-safe class does not require you to call any synchronization functions. Everything is handled internally to the class, allowing you to concentrate on how to best use the class, not about how it might get corrupted. An effective technique for creating a fully thread-safe class is to merge the synchronization class into the resource class. Merging the synchronization classes into the shared class is a straightforward process.

As an example, take an application that maintains a linked list of accounts. This application allows up to three accounts to be examined in separate windows, but only one can be updated at any particular time. When an account is updated, the updated data is sent over the network to a data archive.

This example application uses all three types of synchronization classes. Because it allows up to three accounts to be examined at one time, it uses CSemaphore to limit access to three view objects. When an attempt to view a fourth account occurs, the application either waits until one of the first three windows closes or it fails. When an account is updated, the application uses CCriticalSection to ensure that only one account is updated at a time. After the update succeeds, it signals CEvent, which releases a thread waiting for the event to be signaled. This thread sends the new data to the data archive.

Designing a Thread-Safe Class

To make a class fully thread-safe, first add the appropriate synchronization class to the shared classes as a data member. In the previous account-management example, a **CSemaphore** data member would be added to the view class, a **CCriticalSection** data member would be added to the linked-list class, and a **CEvent** data member would be added to the data storage class.

Next, add synchronization calls to all member functions that modify the data in the class or access a controlled resource. In each function, you should create either a CSingleLock or CMultiLock object and call that object's **Lock** function. When the lock object goes out of scope and is destroyed, the object's destructor calls **Unlock** for you, releasing the resource. Of course, you can call **Unlock** directly if you want.

Designing your thread-safe class in this fashion allows it to be used in a multithreaded application as easily as a non-thread-safe class, but with a higher level of safety. Encapsulating the synchronization object and synchronization access object into the resource's class provides all the benefits of fully thread-safe programming without the drawback of maintaining synchronization code.

The following code example demonstrates this method by using a data member, `m_CritSection` (of type **CCriticalSection**), declared in the shared resource class and a **CSingleLock** object. The synchronization of the shared resource (derived from **CWinThread**) is attempted by creating a **CSingleLock** object using the address of the `m_CritSection` object. An attempt is made to lock the resource and, when obtained, work is done on the shared object. When the work is finished, the resource is unlocked with a call to **Unlock**.

```
CSingleLock singleLock(&m_CritSection);
singleLock.Lock();
// resource locked
//.usage of shared resource...

singleLock.Unlock();
```

Note

CCriticalSection, unlike other MFC synchronization classes, does not have the option of a timed lock request. The waiting period for a thread to become free is infinite.

The drawbacks to this approach are that the class will be slightly slower than the same class without the synchronization objects added. Also, if there is a chance that more than one thread might delete the object, the merged approach might not always work. In this situation, it is better to maintain separate synchronization objects.

For information about determining which synchronization class to use in different situations, see Multithreading: When to Use the Synchronization Classes. For more information about synchronization, see Synchronization in the *Platform SDK*. For more information about multithreading support in MFC, see Multithreading with C++ and MFC.

See Also

Concepts

Multithreading with C++ and MFC

Multithreading: Terminating Threads

Two normal situations cause a thread to terminate: the controlling function exits or the thread is not allowed to run to completion. If a word processor used a thread for background printing, the controlling function would terminate normally if printing completed successfully. If the user wants to cancel the printing, however, the background printing thread has to be terminated prematurely. This topic explains both how to implement each situation and how to get the exit code of a thread after it terminates.

- Normal Thread Termination
- Premature Thread Termination
- Retrieving the Exit Code of a Thread

Normal Thread Termination

For a worker thread, normal thread termination is simple: Exit the controlling function and return a value that signifies the reason for termination. You can use either the AfxEndThread function or a **return** statement. Typically, 0 signifies successful completion, but that is up to you.

For a user-interface thread, the process is just as simple: from within the user-interface thread, call PostQuitMessage in the *Platform SDK*. The only parameter that **PostQuitMessage** takes is the exit code of the thread. As for worker threads, 0 typically signifies successful completion.

Premature Thread Termination

Terminating a thread prematurely is almost as simple: Call AfxEndThread from within the thread. Pass the desired exit code as the only parameter. This stops execution of the thread, deallocates the thread's stack, detaches all DLLs attached to the thread, and deletes the thread object from memory.

AfxEndThread must be called from within the thread to be terminated. If you want to terminate a thread from another thread, you must set up a communication method between the two threads.

Retrieving the Exit Code of a Thread

To get the exit code of either the worker or the user-interface thread, call the GetExitCodeThread function. For information about this function, see the *Platform SDK*. This function takes the handle to the thread (stored in the `m_hThread` data member of **CWinThread** objects) and the address of a **DWORD**.

If the thread is still active, **GetExitCodeThread** places **STILL_ACTIVE** in the supplied **DWORD** address; otherwise, the exit code is placed in this address.

Retrieving the exit code of CWinThread objects takes an extra step. By default, when a **CWinThread** thread terminates, the thread object is deleted. This means you cannot access the `m_hThread` data member because the **CWinThread** object no longer exists. To avoid this situation, do one of the following:

- Set the `m_bAutoDelete` data member to **FALSE**. This allows the **CWinThread** object to survive after the thread has been terminated. You can then access the `m_hThread` data member after the thread has been terminated. If you use this technique, however, you are responsible for destroying the **CWinThread** object because the framework will not automatically delete it for you. This is the preferred method.
- Store the thread's handle separately. After the thread is created, copy its `m_hThread` data member (using **::DuplicateHandle**) to another variable and access it through that variable. This way the object is deleted automatically when termination occurs and you can still find out why the thread terminated. Be careful that the thread does not terminate before you can duplicate the handle. The safest way to do this is to pass **CREATE_SUSPENDED** to AfxBeginThread, store the handle, and then resume the thread by calling ResumeThread.

Either method allows you to determine why a **CWinThread** object terminated.

See Also

Reference

_endthread, _endthreadex
_beginthread, _beginthreadex
ExitThread

Concepts

Multithreading with C++ and MFC

Multithreading: Programming Tips

Multithreaded applications require stricter care than single-threaded applications when accessing data. Because there are multiple, independent paths of execution in use simultaneously in multithreaded applications, either the algorithms, the data, or both must be aware that data could be used by more than one thread at a time. This topic explains techniques for avoiding potential problems when programming multithreaded applications with the Microsoft Foundation Class (MFC) library.

- Accessing Objects from Multiple Threads
- Accessing MFC Objects from Non-MFC Threads
- Windows Handle Maps
- Communicating Between Threads

Accessing Objects from Multiple Threads

For size and performance reasons, MFC objects are not thread-safe at the object level, only at the class level. This means that you can have two separate threads manipulating two different **CString** objects, but not two threads manipulating the same **CString** object. If you absolutely must have multiple threads manipulating the same object, protect such access with appropriate Win32 synchronization mechanisms, such as critical sections. For more information about critical sections and other related objects, see Synchronization in the Platform SDK.

The class library uses critical sections internally to protect global data structures, such as those used by the debug memory allocation.

Accessing MFC Objects from Non-MFC Threads

If you have a multithreaded application that creates a thread in a way other than using a CWinThread object, you cannot access other MFC objects from that thread. In other words, if you want to access any MFC object from a secondary thread, you must create that thread with one of the methods described in Multithreading: Creating User-Interface Threads or Multithreading: Creating Worker Threads. These methods are the only ones that allow the class library to initialize the internal variables necessary to handle multithreaded applications.

Windows Handle Maps

As a general rule, a thread can access only MFC objects that it created. This is because temporary and permanent Windows handle maps are kept in thread local storage to help maintain protection from simultaneous access from multiple threads. For example, a worker thread cannot perform a calculation and then call a document's **UpdateAllViews** member function to have the windows that contain views on the new data modified. This has no effect at all, because the map from **CWnd** objects to **HWND**s is local to the primary thread. This means that one thread might have a mapping from a Windows handle to a C++ object, but another thread might map that same handle to a different C++ object. Changes made in one thread would not be reflected in the other.

There are several ways around this problem. The first is to pass individual handles (such as an **HWND**) rather than C++ objects to the worker thread. The worker thread then adds these objects to its temporary map by calling the appropriate **FromHandle** member function. You could also add the object to the thread's permanent map by calling **Attach**, but this should be done only if you are guaranteed that the object will exist longer than the thread.

Another method is to create new user-defined messages corresponding to the different tasks your worker threads will be performing and post these messages to the application's main window using **::PostMessage**. This method of communication is similar to two different applications conversing except that both threads are executing in the same address space.

For more information about handle maps, see Technical Note 3. For more information about thread local storage, see Thread Local Storage and Using Thread Local Storage in the *Platform SDK*.

Communicating Between Threads

MFC provides a number of classes that allow threads to synchronize access to objects to maintain thread safety. Usage of these classes is described in Multithreading: How to Use the Synchronization Classes and Multithreading: When to Use the Synchronization Classes. For more information about these objects, see Synchronization in the *Platform SDK*.

See Also

Concepts

Multithreading and Locales

Both the C Runtime Library and the Standard C++ Library provide support for changing the locale of your program. This topic discusses issues that arise when using the locale functionality of both libraries in a multithreaded application.

Remarks

With the C Runtime Library, you can create multithreaded applications using the **_beginthread** and **_beginthreadex** functions. This topic only covers multithreaded applications created using these functions. For more information, see _beginthread, _beginthreadex.

To change the locale using the C Runtime Library, use the setlocale function. In previous versions of Visual C++, this function would always modify the locale throughout the entire application. There is now support for setting the locale on a per-thread basis. This is done using the _configthreadlocale function. To specify that setlocale should only change the locale in the current thread, call `_configthreadlocale(_ENABLE_PER_THREAD_LOCALE)` in that thread. Conversely, calling `_configthreadlocale(_DISABLE_PER_THREAD_LOCALE)` will cause that thread to use the global locale, and any call to setlocale in that thread will change the locale in all threads that have not explicitly enabled per-thread locale.

To change the locale using the C++ Runtime Library, use the locale Class. By calling the locale::global method, you change the locale in every thread that has not explicitly enabled per-thread locale. To change the locale in a single thread or portion of an application, simply create an instance of a **locale** object in that thread or portion of code.

Note
Calling locale::global changes the locale for both the Standard C++ Library and the C Runtime Library. However, calling setlocale only changes the locale for the C Runtime Library; the Standard C++ Library is not affected.

The following examples show how to use the setlocale function, the locale Class, and the _configthreadlocale function to change the locale of an application in several different scenarios.

Example

In this example, the main thread spawns two child threads. The first thread, Thread A, enables per-thread locale by calling `_configthreadlocale(_ENABLE_PER_THREAD_LOCALE)`. The second thread, Thread B, as well as the main thread, do not enable per-thread locale. Thread A then proceeds to change the locale using the setlocale function of the C Runtime Library.

Since Thread A has per-thread locale enabled, only the C Runtime Library functions in Thread A start using the "french" locale. The C Runtime Library functions in Thread B and in the main thread continue to use the "C" locale. Also, since setlocale does not affect the Standard C++ Library locale, all Standard C++ Library objects continue to use the "C" locale.

```
// multithread_locale_1.cpp
// compile with: /EHsc /MD
#include <clocale>
#include <cstdio>
#include <locale>
#include <process.h>
#include <windows.h>

#define NUM_THREADS 2
using namespace std;

unsigned __stdcall RunThreadA(void *params);
unsigned __stdcall RunThreadB(void *params);

BOOL localeSet = FALSE;
HANDLE printMutex = CreateMutex(NULL, FALSE, NULL);

int main()
{
    HANDLE threads[NUM_THREADS];

    unsigned aID;
    threads[0] = (HANDLE)_beginthreadex(
        NULL, 0, RunThreadA, NULL, 0, &aID);
```

```
    unsigned bID;
    threads[1] = (HANDLE)_beginthreadex(
        NULL, 0, RunThreadB, NULL, 0, &bID);

    WaitForMultipleObjects(2, threads, TRUE, INFINITE);

    printf_s("[Thread main] Per-thread locale is NOT enabled.\n");
    printf_s("[Thread main] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread main] locale::global is set to \"%s\"\n",
        locale().name().c_str());

    CloseHandle(threads[0]);
    CloseHandle(threads[1]);
    CloseHandle(printMutex);

    return 0;
}

unsigned __stdcall RunThreadA(void *params)
{
    _configthreadlocale(_ENABLE_PER_THREAD_LOCALE);
    setlocale(LC_ALL, "french");
    localeSet = TRUE;

    WaitForSingleObject(printMutex, INFINITE);
    printf_s("[Thread A] Per-thread locale is enabled.\n");
    printf_s("[Thread A] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread A] locale::global is set to \"%s\"\n\n",
        locale().name().c_str());
    ReleaseMutex(printMutex);

    return 1;
}

unsigned __stdcall RunThreadB(void *params)
{
    while (!localeSet)
        Sleep(100);

    WaitForSingleObject(printMutex, INFINITE);
    printf_s("[Thread B] Per-thread locale is NOT enabled.\n");
    printf_s("[Thread B] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread B] locale::global is set to \"%s\"\n\n",
        locale().name().c_str());
    ReleaseMutex(printMutex);

    return 1;
}
```

Output

```
[Thread A] Per-thread locale is enabled.
[Thread A] CRT locale is set to "French_France.1252"
[Thread A] locale::global is set to "C"

[Thread B] Per-thread locale is NOT enabled.
[Thread B] CRT locale is set to "C"
[Thread B] locale::global is set to "C"

[Thread main] Per-thread locale is NOT enabled.
[Thread main] CRT locale is set to "C"
[Thread main] locale::global is set to "C"
```

In this example, the main thread spawns two child threads. The first thread, Thread A, enables per-thread locale by calling `_configthreadlocale(_ENABLE_PER_THREAD_LOCALE)`. The second thread, Thread B, as well as the main thread, do not enable

per-thread locale. Thread A then proceeds to change the locale using the locale::global method of the Standard C++ Library.

Since Thread A has per-thread locale enabled, only the C Runtime Library functions in Thread A start using the "french" locale. The C Runtime Library functions in Thread B and in the main thread continue to use the "C" locale. However, since the locale::global method changes the locale "globally", all Standard C++ Library objects in all threads start using the "french" locale.

```
// multithread_locale_2.cpp
// compile with: /EHsc /MD
#include <clocale>
#include <cstdio>
#include <locale>
#include <process.h>
#include <windows.h>

#define NUM_THREADS 2
using namespace std;

unsigned __stdcall RunThreadA(void *params);
unsigned __stdcall RunThreadB(void *params);

BOOL localeSet = FALSE;
HANDLE printMutex = CreateMutex(NULL, FALSE, NULL);

int main()
{
    HANDLE threads[NUM_THREADS];

    unsigned aID;
    threads[0] = (HANDLE)_beginthreadex(
        NULL, 0, RunThreadA, NULL, 0, &aID);

    unsigned bID;
    threads[1] = (HANDLE)_beginthreadex(
        NULL, 0, RunThreadB, NULL, 0, &bID);

    WaitForMultipleObjects(2, threads, TRUE, INFINITE);

    printf_s("[Thread main] Per-thread locale is NOT enabled.\n");
    printf_s("[Thread main] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread main] locale::global is set to \"%s\"\n",
        locale().name().c_str());

    CloseHandle(threads[0]);
    CloseHandle(threads[1]);
    CloseHandle(printMutex);

    return 0;
}

unsigned __stdcall RunThreadA(void *params)
{
    _configthreadlocale(_ENABLE_PER_THREAD_LOCALE);
    locale::global(locale("french"));
    localeSet = TRUE;

    WaitForSingleObject(printMutex, INFINITE);
    printf_s("[Thread A] Per-thread locale is enabled.\n");
    printf_s("[Thread A] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread A] locale::global is set to \"%s\"\n\n",
        locale().name().c_str());
    ReleaseMutex(printMutex);

    return 1;
}
```

```
unsigned __stdcall RunThreadB(void *params)
{
    while (!localeSet)
        Sleep(100);

    WaitForSingleObject(printMutex, INFINITE);
    printf_s("[Thread B] Per-thread locale is NOT enabled.\n");
    printf_s("[Thread B] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread B] locale::global is set to \"%s\"\n\n",
        locale().name().c_str());
    ReleaseMutex(printMutex);

    return 1;
}
```

Output

```
[Thread A] Per-thread locale is enabled.
[Thread A] CRT locale is set to "French_France.1252"
[Thread A] locale::global is set to "French_France.1252"

[Thread B] Per-thread locale is NOT enabled.
[Thread B] CRT locale is set to "C"
[Thread B] locale::global is set to "French_France.1252"

[Thread main] Per-thread locale is NOT enabled.
[Thread main] CRT locale is set to "C"
[Thread main] locale::global is set to "French_France.1252"
```

In this example, the main thread spawns two child threads. The first thread, Thread A, enables per-thread locale by calling `_configthreadlocale(_ENABLE_PER_THREAD_LOCALE)`. The second thread, Thread B, as well as the main thread, do not enable per-thread locale. Thread B then proceeds to change the locale using the setlocale function of the C Runtime Library.

Since Thread B does not have per-thread locale enabled, the C Runtime Library functions in Thread B and in the main thread start using the "french" locale. The C Runtime Library functions in Thread A continue to use the "C" locale because Thread A has per-thread locale enabled. Also, since setlocale does not affect the Standard C++ Library locale, all Standard C++ Library objects continue to use the "C" locale.

```
// multithread_locale_3.cpp
// compile with: /EHsc /MD
#include <clocale>
#include <cstdio>
#include <locale>
#include <process.h>
#include <windows.h>

#define NUM_THREADS 2
using namespace std;

unsigned __stdcall RunThreadA(void *params);
unsigned __stdcall RunThreadB(void *params);

BOOL localeSet = FALSE;
BOOL configThreadLocaleCalled = FALSE;
HANDLE printMutex = CreateMutex(NULL, FALSE, NULL);

int main()
{
    HANDLE threads[NUM_THREADS];

    unsigned aID;
    threads[0] = (HANDLE)_beginthreadex(
        NULL, 0, RunThreadA, NULL, 0, &aID);

    unsigned bID;
    threads[1] = (HANDLE)_beginthreadex(
        NULL, 0, RunThreadB, NULL, 0, &bID);
```

```
    WaitForMultipleObjects(2, threads, TRUE, INFINITE);

    printf_s("[Thread main] Per-thread locale is NOT enabled.\n");
    printf_s("[Thread main] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread main] locale::global is set to \"%s\"\n",
        locale().name().c_str());

    CloseHandle(threads[0]);
    CloseHandle(threads[1]);
    CloseHandle(printMutex);

    return 0;
}

unsigned __stdcall RunThreadA(void *params)
{
    _configthreadlocale(_ENABLE_PER_THREAD_LOCALE);
    configThreadLocaleCalled = TRUE;
    while (!localeSet)
        Sleep(100);

    WaitForSingleObject(printMutex, INFINITE);
    printf_s("[Thread A] Per-thread locale is enabled.\n");
    printf_s("[Thread A] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread A] locale::global is set to \"%s\"\n\n",
        locale().name().c_str());
    ReleaseMutex(printMutex);

    return 1;
}

unsigned __stdcall RunThreadB(void *params)
{
    while (!configThreadLocaleCalled)
        Sleep(100);
    setlocale(LC_ALL, "french");
    localeSet = TRUE;

    WaitForSingleObject(printMutex, INFINITE);
    printf_s("[Thread B] Per-thread locale is NOT enabled.\n");
    printf_s("[Thread B] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread B] locale::global is set to \"%s\"\n\n",
        locale().name().c_str());
    ReleaseMutex(printMutex);

    return 1;
}
```

Output

```
[Thread B] Per-thread locale is NOT enabled.
[Thread B] CRT locale is set to "French_France.1252"
[Thread B] locale::global is set to "C"

[Thread A] Per-thread locale is enabled.
[Thread A] CRT locale is set to "C"
[Thread A] locale::global is set to "C"

[Thread main] Per-thread locale is NOT enabled.
[Thread main] CRT locale is set to "French_France.1252"
[Thread main] locale::global is set to "C"
```

In this example, the main thread spawns two child threads. The first thread, Thread A, enables per-thread locale by calling `_configthreadlocale(_ENABLE_PER_THREAD_LOCALE)`. The second thread, Thread B, as well as the main thread, do not enable

per-thread locale. Thread B then proceeds to change the locale using the locale::global method of the Standard C++ Library.

Since Thread B does not have per-thread locale enabled, the C Runtime Library functions in Thread B and in the main thread start using the "french" locale. The C Runtime Library functions in Thread A continue to use the "C" locale because Thread A has per-thread locale enabled. However, since the locale::global method changes the locale "globally", all Standard C++ Library objects in all threads start using the "french" locale.

```
// multithread_locale_4.cpp
// compile with: /EHsc /MD
#include <clocale>
#include <cstdio>
#include <locale>
#include <process.h>
#include <windows.h>

#define NUM_THREADS 2
using namespace std;

unsigned __stdcall RunThreadA(void *params);
unsigned __stdcall RunThreadB(void *params);

BOOL localeSet = FALSE;
BOOL configThreadLocaleCalled = FALSE;
HANDLE printMutex = CreateMutex(NULL, FALSE, NULL);

int main()
{
    HANDLE threads[NUM_THREADS];

    unsigned aID;
    threads[0] = (HANDLE)_beginthreadex(
        NULL, 0, RunThreadA, NULL, 0, &aID);

    unsigned bID;
    threads[1] = (HANDLE)_beginthreadex(
        NULL, 0, RunThreadB, NULL, 0, &bID);

    WaitForMultipleObjects(2, threads, TRUE, INFINITE);

    printf_s("[Thread main] Per-thread locale is NOT enabled.\n");
    printf_s("[Thread main] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread main] locale::global is set to \"%s\"\n",
        locale().name().c_str());

    CloseHandle(threads[0]);
    CloseHandle(threads[1]);
    CloseHandle(printMutex);

    return 0;
}

unsigned __stdcall RunThreadA(void *params)
{
    _configthreadlocale(_ENABLE_PER_THREAD_LOCALE);
    configThreadLocaleCalled = TRUE;
    while (!localeSet)
        Sleep(100);

    WaitForSingleObject(printMutex, INFINITE);
    printf_s("[Thread A] Per-thread locale is enabled.\n");
    printf_s("[Thread A] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread A] locale::global is set to \"%s\"\n\n",
        locale().name().c_str());
    ReleaseMutex(printMutex);

    return 1;
```

```
}

unsigned __stdcall RunThreadB(void *params)
{
    while (!configThreadLocaleCalled)
        Sleep(100);
    locale::global(locale("french"));
    localeSet = TRUE;

    WaitForSingleObject(printMutex, INFINITE);
    printf_s("[Thread B] Per-thread locale is NOT enabled.\n");
    printf_s("[Thread B] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread B] locale::global is set to \"%s\"\n\n",
        locale().name().c_str());
    ReleaseMutex(printMutex);

    return 1;
}
```

Output

```
[Thread B] Per-thread locale is NOT enabled.
[Thread B] CRT locale is set to "French_France.1252"
[Thread B] locale::global is set to "French_France.1252"

[Thread A] Per-thread locale is enabled.
[Thread A] CRT locale is set to "C"
[Thread A] locale::global is set to "French_France.1252"

[Thread main] Per-thread locale is NOT enabled.
[Thread main] CRT locale is set to "French_France.1252"
[Thread main] locale::global is set to "French_France.1252"
```

See Also

Reference

_beginthread, _beginthreadex
_configthreadlocale
setlocale
Internationalization
Locale
<clocale>
<locale>
locale Class

Concepts

Multithreading

64-Bit Programming with Visual C++

This section contains topics about targeting 64-bit hardware with the Visual C++ build tools.

In This Section

- How to: Configure Visual C++ Projects to Target 64-Bit Platforms
- How to: Enable a 64-Bit Visual C++ Toolset at the Command Line
- Common Visual C++ 64-bit Migration Issues
- Itanium Software Conventions
- x64 Software Conventions

Related Sections

64-bit Applications

align (C++)

/clr (Common Language Runtime Compilation) for information on how to run an application compiled with **/clr:safe** under WOW64 on a 64-bit operating system.

/favor (Optimize for 64)

http://msdn.microsoft.com/visualc/using/building/64bit/default.aspx

Introducing 64-bit Windows

Itanium Intrinsics

MASM for x64 (ml64.exe)

/Wp64 (Detect 64-Bit Portability Issues)

__w64

x64 Intrinsics

See Also

Other Resources

Programming Guide

How to: Configure Visual C++ Projects to Target 64-Bit Platforms

This topic describes how to set up C++ applications to target 64-bit platforms using project configurations available in the Visual Studio Integrated Development Environment (IDE).

To develop 64-bit applications you must install one or both of the Visual C++ 64-bit compilers, otherwise 64-bit project configurations will not be available in the IDE. See Installing Visual Studio 64-bit Components for information.

Note
64-bit tools are not available on Visual C++ Express Edition by default. To enable 64-bit tools on Visual C++ Express Edition, install the .NET Framework SDK in addition to Visual C++ Express Edition. Otherwise, an error occurs when you attempt to configure a project to target a 64-bit platform using Visual C++ Express Edition.

The first procedure describes how to change the active project configuration to target 64-bit platforms using the Visual Studio IDE. The second procedure describes how to migrate Win32 project settings into a 64-bit project configuration.

To set up C++ applications to target 64-bit platforms

1. Open the C++ project that you want to configure to target a 64-bit platform.
2. Open the property pages for that project. For more information, see How to: Open Project Property Pages.

Note
For .NET projects, make sure the **Configuration Properties** node, or one of its child nodes, is selected in the **<Project name> Property Pages** dialog box. Otherwise, the **Configuration Manager** button remains disabled.

3. Click **Configuration Manager** to open the Configuration Manager Dialog Box.
4. Click the **Active Solution Platform** list, and then select the **<New...>** option to open the New Solution Platform Dialog Box.
5. Click the **Type or select the new platform** drop-down arrow, and then select a 64-bit platform.

Note
In the **New Solution Platform** dialog box, you can copy existing project settings into the new 64-bit project configuration using the **Copy settings from** option.

6. Click **OK**. The platform you selected in the preceding step will appear under **Active Solution Platform** in the **Configuration Manager** dialog box.
7. Click **Close** in the **Configuration Manager** dialog box, and then click **OK** in the **<Projectname> Property Pages** dialog box.

To copy Win32 project settings into a 64-bit project configuration

- When the **New Solution Platform** dialog box is open while you set up your project to target a 64-bit platform, click the **Copy settings from** drop-down arrow, and then select **Win32**. The following project settings are automatically updated on the project level:
  - /MACHINE (Specify Target Platform) is set to **/MACHINE:IA64** or **/MACHINE:X64**.
  - **Register Output** is turned OFF. For more information, see Linker Property Pages.
  - **Target Environment** is set to **/env x64** or **/env ia64**. For more information, see MIDL Property Pages: General.
  - **Validate Parameters** is cleared and reset to the default value. For more information, see MIDL Property Pages: Advanced.

- If **Debug Information Format** was set to **/ZI** in the Win32 project configuration, then it is set to **/Zi** in the 64-bit project configuration. For more information, see /Z7, /Zi, /ZI (Debug Information Format).
- Values of **WIN32** are replaced by **WIN64** for /D (Preprocessor Definitions).

Note
None of these project properties are changed if they are overridden on the file level.

See Also

Concepts

64-bit Applications

Other Resources

64-Bit Programming with Visual C++

How to: Enable a 64-Bit Visual C++ Toolset at the Command Line

Although not installed by default and not available in all editions (for more information, see Visual C++ Editions.), Visual C++ 2005 includes compilers that let you create applications that can run on a 64-bit Windows operating system.

See Installing Visual Studio 64-bit Components for information on installing the 64-bit compilers.

For information on creating 64-bit applications inside the Visual Studio development environment, see How to: Configure Visual C++ Projects to Target 64-Bit Platforms.

The following list describes the various versions of cl.exe (the Visual C++ compiler):

x86 on x86

Allows you to create output files for x86 machines. This version of cl.exe runs as a 32-bit process, native on an x86 machine and under WOW64 on a 64-bit Widows operating system.

Itanium on x86 (Itanium cross-compiler)

Allows you to create output files for Itanium. This version of cl.exe runs as a 32-bit process, native on an x86 machine and under WOW64 on a 64-bit Widows operating system.

x64 on x86 (x64 cross-compiler)

Allows you to create output files for x64. This version of cl.exe runs as a 32-bit process, native on an x86 machine and under WOW64 on a 64-bit Widows operating system.

Itanium on Itanium

Allows you to create output files for Itanium. This version of cl.exe runs as a native process on an Itanium machine.

x64 on x64

Allows you to create output files for x64. This version of cl.exe runs as a native process on an x64 machine.

When the 64-bit compilers are installed on a 64-bit Windows operating system, you will see command prompts for the various 64-bit native and cross compilers. To access these command prompts, go to **Start** and then **All Programs**. From the **Microsoft Visual Studio 2005** menu option, select **Visual Studio Tools** and then a command prompt.

Vcvarsall.bat

Any of the five compilers can be used from the command line by running the vcvarsall.bat batch file. By default, the full path for this file is C:\Program Files\Microsoft Visual Studio 8\VC\Vcvarsall.bat.

(Another way to enable the Visual C++ toolset is to use vcvars32.bat, but this only enables the 32-bit compiler. For more information, see Setting the Path and Environment Variables for Command-Line Builds.)

If no arguments are provided, this batch file configures the necessary environment variables for using the x86, 32-bit compiler, but it can be used to configure any of the 64-bit compilers. The following table shows the supported arguments for vcvarsall.bat.

Vcvarsall.bat argument	Compiler	Host (either natively or using emulation)	Target architecture
"x86" (default)	32-bit Native	x86, x64, Itanium	x86
"x86_amd64"	x64 Cross	X86, x64, Itanium	x64
"x86_IPF"	Itanium Cross	X86, x64, Itanium	Itanium
"amd64"	x64 Native	x64	x64
"IPF" or "itanium"	Itanium Native	Itanium	Itanium

See Also

Other Resources

64-Bit Programming with Visual C++

Common Visual C++ 64-bit Migration Issues

When you use Visual C++ to create applications to run on a 64-bit Windows operating system, you should be aware of the following issues:

- An **int** and a **long** are 32-bit values on 64-bit Windows operating systems. For programs that you plan to compile for 64-bit platforms, you should be careful not to assign pointers to 32-bit variables. Pointers are 64-bit on 64-bit platforms, and you will truncate the pointer value if you assign it to a 32-bit variable.
- **size_t**, **time_t**, and **ptrdiff_t** are 64-bit values on 64-bit Windows operating systems.
- **time_t** is a 32-bit value on 32-bit Windows operating systems in Visual C++ versions before Visual C++ 2005. In Visual C++ 2005 and later, **time_t** is a 64-bit integer by default. For more information, see Time Management.

  You should be aware of where your code takes an **int** value and processes it as a **size_t** or **time_t** value. It is possible that the number could grow to be larger than a 32-bit number and data will be truncated when it is passed back to the **int** storage.

The %x (hex **int** format) **printf** modifier will not work as expected on a 64-bit Windows operating system. It will only operate on the first 32 bits of the value that is passed to it.

- Use %I32x to display an integer on a Windows 32-bit operating system.
- Use %I64x to display an integer on a Windows 64-bit operating system.
- The %p (hex format for a pointer) will work as expected on a 64-bit Windows operating system.

For more information, see:

- Compiler Options
- /Wp64
- __w64
- Migration Tips

See Also

Other Resources

64-Bit Programming with Visual C++

Porting and Upgrading Programs

Itanium Software Conventions

For information on software conventions when developing for the Itanium Processor Family (IPF) see,

- http://developer.intel.com/design/itanium/downloads/245358.htm
- http://msdn.microsoft.com/visualc/using/building/64bit/default.aspx?pull=/library/en-us/dnnetserv/html/ws03-64-bitwindevover.asp

See Also

Other Resources

64-Bit Programming with Visual C++

x64 Software Conventions

This section describes the Visual C++ calling convention methodology for x64, 64-bit extension to the x86 architecture.

- Overview of x64 Calling Conventions
- Types and Storage
- Register Usage
- Calling Convention
- Stack Usage
- Prolog and Epilog
- Exception Handling in Visual C++
- Intrinsics and Inline Assembly
- Image Format

The following compiler option helps you optimize your application for x64:

- /favor (Optimize for 64)

See Also
Reference
Calling Conventions

Overview of x64 Calling Conventions

Two important modifications from x86 to x64 are the 64-bit addressing capability and a flat set of 16 64-bit registers for general use. Given the expanded register set, x64 just uses the __fastcall calling convention and a RISC-based exception-handling model. The **__fastcall** model uses registers for the first four arguments and the stack frame to pass the other parameters.

The following compiler option helps you optimize your application for x64:

- /favor (Optimize for 64)

Calling convention

The x64 Application Binary Interface (ABI) is a 4 register fast-call calling convention, with stack-backing for those registers. There is a strict one-to-one correspondence between arguments in a function, and the registers for those arguments. Any argument that doesn't fit in 8 bytes, or is not 1, 2, 4, or 8 bytes, must be passed by reference. There is no attempt to spread a single argument across multiple registers. The x87 register stack is unused. It may be used, but must be considered volatile across function calls. All floating point operations are done using the 16 XMM registers. The arguments are passed in registers RCX, RDX, R8, and R9. If the arguments are float/double, they are passed in XMM0L, XMM1L, XMM2L, and XMM3L. 16 byte arguments are passed by reference. Parameter passing is described in detail in Parameter Passing. In addition to these registers, RAX, R10, R11, XMM4, and XMM5 are volatile. All other registers are non-volatile. Register usage is documented in detail in Register Usage and Caller/Callee Saved Registers.

The caller is responsible for allocating space for parameters to the callee, and must always allocate sufficient space for the 4 register parameters, even if the callee doesn't have that many parameters. This aids in the simplicity of supporting C unprototyped functions, and vararg C/C++ functions. For vararg or unprototyped functions, any float values must be duplicated in the corresponding general-purpose register. Any parameters above the first 4 must be stored on the stack, above the backing store for the first 4, prior to the call. Vararg function details can be found in Varargs. Unprototyped function information is detailed in Unprototyped Functions.

Alignment

Most structures are aligned to their natural alignment. The primary exceptions are the stack pointer and malloc or alloca memory, which are aligned to 16 byte, in order to aid performance. Alignment above 16 bytes must be done manually, but since 16 bytes is a common alignment size for XMM operations, this should suffice for most code. For more information about structure layout and alignment see Types and Storage. For information about the stack layout, see Stack Usage.

Unwindability

All non-leaf functions [functions that neither call a function, nor allocate any stack space themselves] must be annotated with data [referred to as xdata or ehdata, which is pointed to from pdata] that describes to the operating system how to properly unwind them, to recover non-volatile registers. Prologs and epilogs are highly restricted, so that they can be properly described in xdata. The stack pointer must be aligned to 16 bytes, except for leaf functions, in any region of code that isn't part of an epilog or prolog. For details about the proper structure of function prolog and epilogs, see Prolog and Epilog. For more information about exception handling and the exception handling/unwinding pdata and xdata see Exception Handling (x64).

See Also

Reference

x64 Software Conventions

Types and Storage

This section describes the enumeration and storage of data types for the x64 architecture.

- Scalar Types
- Aggregates and Unions
- Structure Alignment Examples
- Bitfields

See Also

Reference

x64 Software Conventions

Scalar Types

Although the access of data can stem from any alignment, it is recommended that data be aligned on its natural boundary to avoid performance loss (or a multiple thereof). Enums are constant integers and are treated as 32-bit integers. The following table describes the type definition and recommended storage for it as it pertains to alignment using the following alignment values:

- Byte – 8 bits
- Word – 16 bits
- Double Word – 32 bits
- Quad Word – 64 bits
- Octa Word – 128 bits

Scalar Type	C Data Type	Storage Size (in bytes)	Recommended Alignment
INT8	**char**	1	Byte
UINT8	**unsigned char**	1	Byte
INT16	**short**	2	Word
UINT16	**unsigned short**	2	Word
INT32	**int, long**	4	Doubleword
UINT32	**unsigned int, unsigned long**	4	Doubleword
INT64	**__int64**	8	Quadword
UINT64	**unsigned __int64**	8	Quadword
FP32 (single precision)	**float**	4	Doubleword
FP64 (double precision)	**double**	8	Quadword
POINTER	*****	8	Quadword
__m64	**struct __m64**	8	Quadword
__m128	**struct __m128**	16	Octaword

See Also

Reference

Types and Storage

Aggregates and Unions

Other types such as arrays, structs, and unions have stricter alignment requirements that ensure consistent aggregate and union storage and data retrieval. Here are the definitions for array, structure, and union:

Array

Contains an ordered group of adjacent data objects. Each object is called an element. All elements within an array have the same size and data type.

Structure

Contains an ordered group of data objects. Unlike the elements of an array, the data objects within a structure can have different data types and sizes. Each data object in a structure is called a member.

Union

An object that holds any one of a set of named members. The members of the named set can be of any type. The storage allocated for a union is equal to the storage required for the largest member of that union, plus any padding required for alignment.

The following table shows the strongly suggested alignment for the scalar members of unions and structures.

Scalar Type	C Data Type	Required Alignment
INT8	**char**	Byte
UINT8	**unsigned char**	Byte
INT16	**short**	Word
UINT16	**unsigned short**	Word
INT32	**int, long**	Doubleword
UINT32	**unsigned int, unsigned long**	Doubleword
INT64	**__int64**	Quadword
UINT64	**unsigned __int64**	Quadword
FP32 (single precision)	**float**	Doubleword
FP64 (double precision)	**double**	Quadword
POINTER	*****	Quadword
__m64	**struct __m64**	Quadword
__m128	**struct __m128**	Octaword

The following aggregate alignment rules apply:

- The alignment of an array is the same as the alignment of one of the elements of the array.
- The alignment of the beginning of a structure or a union is the maximum alignment of any individual member. Each member within the structure or union must be placed at its proper alignment as defined in the previous table, which may require implicit internal padding, depending on the previous member.

- Structure size must be an integral multiple of its alignment, which may require padding after the last member. Since structures and unions can be grouped in arrays, each array element of a structure or union must begin and end at the proper alignment previously determined.
- It is possible to align data in such a way as to be greater than the alignment requirements as long as the previous rules are maintained.
- An individual compiler may adjust the packing of a structure for size reasons. For example /Zp (Struct Member Alignment) allows for adjusting the packing of structures.

See Also

Reference

Types and Storage

Structure Alignment Examples

The following represent a set of examples showing structure requirements.

Example 1

```
struct  S1 {
  short a;       //size = 2 bytes, alignment = 2 bytes;
}
```

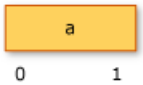

Example 2

```
struct
{   // size = 24 bytes, alignment = Quadword
   int a;
   double b;
   short c;
};
```

```
struct {         // size = 24 bytes, alignment = Quadword
  int a;
  double b;
  short c;
};
```

Example 3

```
struct {         // size = 12 bytes, alignment = Doubleword
  char a;
  short b;
  char c;
  int d;
};
```

Example 4

```
union {          // size = 8 bytes; alignment = Quadword
  char *p;
  short s;
  long l;
};
```

See Also

Reference

Types and Storage

Bitfields

Structure bit fields are limited to 64 bits and can be of type signed int, unsigned int, int64, or unsigned int64. Bit fields that cross the type boundary will skip bits to align the bitfield to the next type alignment. For example, integer bitfields may not cross a 32-bit boundry.

See Also

Reference

Types and Storage

Register Usage

The x64 architecture provides for 16 general-purpose registers (hereafter referred to as integer registers) as well as 16 XMM registers available for floating-point use. Volatile registers are scratch registers presumed by the caller to be destroyed across a call. Nonvolatile registers are required to retain their values across a function call and must be saved by the callee if used.

The following table describes how each register is used across function calls:

Register	Status	Use
RAX	**Volatile**	**Return value register**
RCX	**Volatile**	**First integer argument**
RDX	**Volatile**	**Second integer argument**
R8	**Volatile**	**Third integer argument**
R9	**Volatile**	**Fourth integer argument**
R10:R11	**Volatile**	**Must be preserved as needed by caller; used in syscall/sysret instructions**
R12:R15	**Nonvolatile**	**Must be preserved by callee**
RDI	**Nonvolatile**	**Must be preserved by callee**
RSI	**Nonvolatile**	**Must be preserved by callee**
RBX	**Nonvolatile**	**Must be preserved by callee**
RBP	**Nonvolatile**	**May be used as a frame pointer; must be preserved by callee**
RSP	**Nonvolatile**	**Stack pointer**
XMM0	**Volatile**	**First FP argument**
XMM1	**Volatile**	**Second FP argument**
XMM2	**Volatile**	**Third FP argument**
XMM3	**Volatile**	**Fourth FP argument**
XMM4:XMM5	**Volatile**	**Must be preserved as needed by caller**
XMM6:XMM15	**Nonvolatile**	**Must be preserved as needed by callee.**

See Also

Reference

x64 Software Conventions

Calling Convention

This section describes the process that one function (caller) makes call into another function (callee).

For prototyped functions, all arguments are converted to the expected callee types before passing.

- Parameter Passing
- Varargs
- Unprototyped Functions
- Return Values
- Caller/Callee Saved Registers
- Function Pointers
- Legacy Floating-Point Support
- FpCsr
- MXCSR
- setjmp/longjump

See Also

Reference

x64 Software Conventions

Parameter Passing

The first four integer arguments will be passed in registers. Integer values will be passed (in order left to right) in RCX, RDX, R8, and R9. Arguments four and higher will passed onto the stack. All arguments are right justified in registers. This is done so the callee can ignore the upper bits of the register if need be and can access only the portion of the register necessary.

Floating-point and double-precision arguments are passed in XMM0 – XMM3 (up to 4) with the integer slot (RCX, RDX, R8, and R9) that would normally be used for that cardinal slot being ignored (see example) and vice versa.

__m128 types, arrays and strings are never passed by immediate value but rather a pointer will be passed to memory allocated by the caller. Structs/unions of size 8, 16, 32, or 64 bits and __m64 will be passed as if they were integers of the same size. Structs/unions other than these sizes will be passed as a pointer to memory allocated by the caller. For these aggregate types passed as a pointer (including __m128), the caller-allocated temporary memory will be 16-byte aligned.

Intrinsic functions that do not allocate stack space and do not call other functions can use other volatile registers to pass additional register arguments because there is a tight binding between the compiler and the intrinsic function implementation. This is a further opportunity for improving performance.

The callee has the responsibility of dumping the register parameters into their shadow space if needed.

The following table summarizes how parameters are passed:

Parameter type	How passed
Floating point	First 4 parameters – XMM0 through XMM3. Others passed on stack.
Integer	First 4 parameters – RCX, RDX, R8, R9. Others passed on stack.
Aggregates (8, 16, 32, or 64 bits) and __m64	First 4 parameters – RCX, RDX, R8, R9. Others passed on stack.
Aggregates (other)	By pointer. First 4 parameters passed as pointers in RCX, RDX, R8, and R9
__m128	By pointer. First 4 parameters passed as pointers in RCX, RDX, R8, and R9

Example of argument passing 1 – all integers

```
func1(int a, int b, int c, int d, int e);
// a in RCX, b in RDX, c in R8, d in R9, e pushed on stack
```

Example of argument passing 2 – all floats

```
func2(float a, double b, float c, double d, float e);
// a in XMM0, b in XMM1, c in XMM2, d in XMM3, e pushed on stack
```

Example of argument passing 3 – mixed ints and floats

```
func3(int a, double b, int c, float d);
// a in RCX, b in XMM1, c in R8, d in XMM3
```

Example of argument passing 4 –__m64, __m128, and aggregates

```
func4(__m64 a, _m128 b, struct c, float d);
// a in RCX, ptr to b in RDX, ptr to c in R8, d in XMM3
```

See Also

Reference

Calling Convention

Varargs

If parameters are passed via varargs (for example, ellipsis arguments), then essentially the normal parameter passing applies including spilling the fifth and subsequent arguments. It is again the callee's responsibility to dump arguments that have their address taken. For floating-point values only, both the integer and the floating-point register will contain the float value in case the callee expects the value in the integer registers.

See Also

Reference

Calling Convention

Unprototyped Functions

For functions not fully prototyped, the caller will pass integer values as integers and floating-point values as double precision. For floating-point values only, both the integer register and the floating-point register will contain the float value in case the callee expects the value in the integer registers.

```
func1();
func2() {   // RCX = 2, RDX = XMM1 = 1.0, and R8 = 7
   func1(2, 1.0, 7);
}
```

See Also

Reference

Calling Convention

Return Values (C++)

Return values that can fit into 64-bits are returned through RAX (including __m64 types), except for __m128, __m128i, __m128d, floats, and doubles, which are returned in XMM0. If the return value does not fit within 64 bits, then the caller assumes the responsibility of allocating and passing a pointer for the return value as the first argument. Subsequent arguments are then shifted one argument to the right. That same pointer must be returned by the callee in RAX. User defined types to be returned must be 1, 2, 4, 8, 16, 32, or 64 bits in length.

See Also

Reference

Calling Convention

Caller/Callee Saved Registers

The registers RAX, RCX, RDX, R8, R9, R10, R11 are considered volatile and must be considered destroyed on function calls (unless otherwise safety-provable by analysis such as whole program optimization).

The registers RBX, RBP, RDI, RSI, R12, R13, R14, and R15 are considered nonvolatile and must be saved and restored by a function that uses them.

See Also

Reference

Calling Convention

Function Pointers

Function pointers are simply pointers to the label of the respective function. There are no table of contents (TOC) requirements for function pointers.

See Also

Reference

Calling Convention

Legacy Floating-Point Support

The MMX and floating-point stack registers (MM0-MM7/ST0-ST7) are preserved across context switches. There is no explicit calling convention for these registers. The use of these registers is strictly prohibited in kernel mode code.

See Also

Reference

Calling Convention

FpCsr

The register state also includes the x87 FPU control word. The calling convention dictates this register to be nonvolatile.

The x87 FPU control word register is set to the following standard values at the start of program execution:

```
FPCSR[0:6]: Exception masks all 1's (all exceptions masked)
FPCSR[7]: Reserved – 0
FPCSR[8:9]: Precision Control – 10B (double precision)
FPCSR[10:11]: Rounding  control - 0 (round to nearest)
FPCSR[12]: Infinity control – 0 (not used)
```

A callee that modifies any of the fields within FPCSR must restore them before returning to its caller. Furthermore, a caller that has modified any of these fields must restore them to their standard values before invoking a callee unless by agreement the callee expects the modified values.

There are two exceptions to the rules regarding the non-volatility of the control flags:

1. In functions where the documented purpose of the given function is to modify the nonvolatile FpCsr flags.
2. When it is provably correct that the violation of these rules results in a programs that behaves/means the same as a program where these rules are not violated, for example, through whole-program analysis.

See Also

Reference

Calling Convention

MxCsr

The register state also includes MxCsr. The calling convention divides this register into a volatile portion and a nonvolatile portion. The volatile portion consists of the 6 status flags, MXCSR[0:5], while the remainder of the register, MXCSR[6:15], is considered nonvolatile.

The nonvolatile portion is set to the following standard values at the start of program execution:

```
MXCSR[6]         : Denormals are zeros - 0
MXCSR[7:12]      : Exception masks all 1's (all exceptions masked)
MXCSR[13:14]   : Rounding  control - 0 (round to nearest)
MXCSR[15]      : Flush to zero for masked underflow - 0 (off)
```

A callee that modifies any of the nonvolatile fields within MXCSR must restore them before returning to its caller. Furthermore, a caller that has modified any of these fields must restore them to their standard values before invoking a callee unless by agreement the callee expects the modified values.

There are two exceptions to the rules regarding the non-volatility of the control flags:

- In functions where the documented purpose of the given function is to modify the nonvolatile MxCsr flags.
- When it is provably correct that the violation of these rules results in a programs that behaves/means the same as a program where these rules are not violated, for example, through whole-program analysis.

No assumptions can be made about the state of the volatile portion of MXCSR across a function boundary, unless specifically described in a function's documentation.

See Also
Reference
Calling Convention

setjmp/longjump

When you include setjmpex.h or setjmp.h, all calls to setjmp or longjmp will result in an unwind that invokes destructors and finally calls. This differs from x86, where including setjmp.h results in finally clauses and destructors not being invoked.

A call to **setjmp** preserves the current stack pointer, non-volatile registers, and MxCsr registers. Calls to **longjmp** return to the most recent **setjmp** call site and resets the stack pointer, non-volatile registers, and MxCsr registers, back to the state as preserved by the most recent **setjmp** call.

See Also

Reference

Calling Convention

Stack Usage

All memory beyond the current address of RSP is considered volatile: The OS, or a debugger, may overwrite this memory during a user debug session, or an interrupt handler. Thus, RSP must always be set before attempting to read or write values to a stack frame.

This section discusses the allocation of stack space for local variables and the **alloca** intrinsic.

- Stack Allocation
- Dynamic Parameter Stack Area Construction
- Function Types
- malloc Alignment
- alloca

See Also

Reference

x64 Software Conventions

Stack Allocation

A function's prolog is responsible for allocating stack space for local variables, saved registers, stack parameters, and register parameters.

The parameter area is always at the bottom of the stack (even if alloca is used), so that it will always be adjacent to the return address during any function call. It contains at least four entries, but always enough space to hold all the parameters needed by any function that may be called. Note that space is always allocated for the register parameters, even if the parameters themselves are never homed to the stack; a callee is guaranteed that space has been allocated for all its parameters. Home addresses are required for the register arguments so a contiguous area is available in case the called function needs to take the address of the argument list (va_list) or an individual argument. This area also provides a convenient place to save register arguments during thunk execution and as a debugging option (for example, it makes the arguments easy to find during debugging if they are stored at their home addresses in the prolog code). Even if the called function has fewer than 4 parameters, these 4 stack locations are effectively owned by the called function, and may be used by the called function for other purposes besides saving parameter register values. Thus the caller may not save information in this region of stack across a function call.

If space is dynamically allocated (alloca) in a function, then a nonvolatile register must be used as a frame pointer to mark the base of the fixed part of the stack and that register must be saved and initialized in the prolog. Note that when alloca is used, calls to the same callee from the same caller may have different home addresses for their register parameters.

The stack will always be maintained 16-byte aligned, except within the prolog (for example, after the return address is pushed), and except where indicated in Function Types for a certain class of frame functions.

The following is an example of the stack layout where function A calls a non-leaf function B. Function A's prolog has already allocated space for all the register and stack parameters required by B at the bottom of the stack. The call pushes the return address and B's prolog allocates space for its local variables, nonvolatile registers, and the space needed for it to call functions. If B uses alloca, the space is allocated between the local variable/nonvolatile register save area and the parameter stack area.

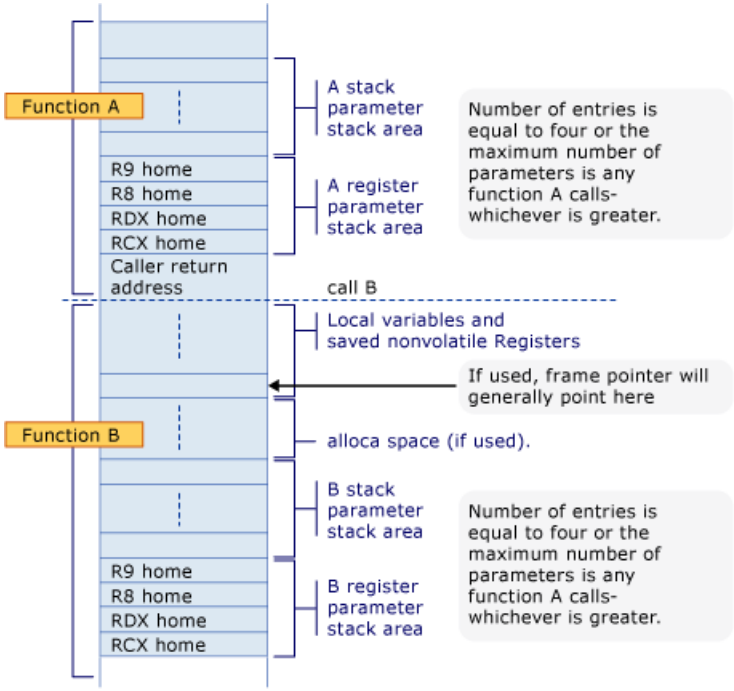

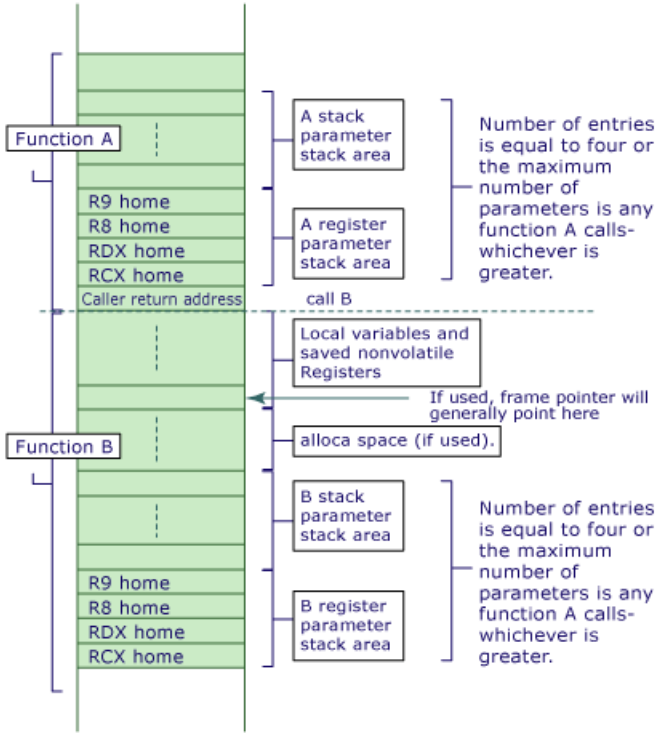

When the function B calls another function, the return address is pushed just below the home address for RCX.

See Also

Reference

Stack Usage

Dynamic Parameter Stack Area Construction

If a frame pointer is used, the option exists to dynamically create the parameter stack area. This is not currently done in the x64 compiler.

See Also

Reference

Stack Usage

Function Types

There are basically two types of functions. A function that requires a stack frame is called a frame function. A function that does not require a stack frame is called a leaf function.

A frame function is a function that allocates stack space, calls other functions, saves nonvolatile registers, or uses exception handling. It also requires a function table entry. A frame function requires a prolog and an epilog. A frame function can dynamically allocate stack space and can employ a frame pointer. A frame function has the full capabilities of this calling standard at its disposal.

If a frame function does not call another function then it is not required to align the stack (referenced in Section Stack Allocation).

A leaf function is one that does not require a function table entry. It cannot call any functions, allocate space, or save any nonvolatile registers. It is allowed to leave the stack unaligned while it executes.

See Also
Reference
Stack Usage

malloc Alignment

malloc is required to return memory on a 16-byte boundary.

See Also

Reference

Stack Usage

alloca

_alloca is required to be 16-byte aligned and additionally required to use a frame pointer.

The stack that is allocated needs to allocate space below it for parameters of subsequently called functions, as discussed in Stack Allocation.

See Also

Reference

Stack Usage

Prolog and Epilog

Every function that allocates stack space, calls other functions, saves nonvolatile registers, or uses exception handling must have a prolog whose address limits are described in the unwind data associated with the respective function table entry (see Exception Handling (x64)). The prolog saves argument registers in their home addresses if required, pushes nonvolatile registers on the stack, allocates the fixed part of the stack for locals and temporaries, and optionally establishes a frame pointer. The associated unwind data must describe the action of the prolog and must provide the information necessary to undo the effect of the prolog code.

If the fixed allocation in the stack is more than one page (that is, greater than 4096 bytes), then it is possible that the stack allocation could span more than one virtual memory page and, therefore, the allocation must be checked before it is actually allocated. A special routine that is callable from the prolog and which does not destroy any of the argument registers is provided for this purpose.

The preferred method for saving nonvolatile registers is to move them onto the stack before the fixed stack allocation. If the fixed stack allocation were performed before the nonvolatile registers were saved, then most probably a 32-bit displacement would be required to address the saved register area (reportedly, pushes of registers are just as fast as moves and should remain so for the foreseeable future in spite of the implied dependency between pushes). Nonvolatile registers can be saved in any order. However, the first use of a nonvolatile register in the prolog must be to save it.

The code for a typical prolog might be:

```
mov       [RSP + 8], RCX
push   R15
push   R14
push   R13
sub      RSP, fixed-allocation-size
lea      R13, 128[RSP]
...
```

This prolog stores the argument register RCX in its home location, saves nonvolatile registers R13-R15, allocates the fixed part of the stack frame, and establishes a frame pointer that points 128 bytes into the fixed allocation area. Using an offset allows more of the fixed allocation area to be addressed with one-byte offsets.

If the fixed allocation size is greater than or equal to one page of memory, then a helper function must be called before modifying RSP. This helper, __chkstk, is responsible for probing the to-be-allocated stack range, to ensure that the stack is extended properly. In that case, the previous prolog example would instead be:

```
mov       [RSP + 8], RCX
push   R15
push   R14
push   R13
mov      RAX,  fixed-allocation-size
call   __chkstk
sub      RSP, RAX
lea      R13, 128[RSP]
...
```

The __chkstk helper will not modify any registers other than R10, R11, and the condition codes. In particular, it will return RAX unchanged and leave all nonvolatile registers and argument-passing registers unmodified.

Epilog code exists at each exit to a function. Whereas there is normally only one prolog, there can be many epilogs. Epilog code trims the stack to its fixed allocation size (if necessary), deallocates the fixed stack allocation, restores nonvolatile registers by popping their saved values from the stack, and returns.

The epilog code must follow a strict set of rules for the unwind code to reliably unwind through exceptions and interrupts. This reduces the amount of unwind data required, because no extra data is needed to describe each epilog. Instead, the unwind code can determine that an epilog is being executed by scanning forward through a code stream to identify an epilog.

If no frame pointer is used in the function, then the epilog must first deallocate the fixed part of the stack, the nonvolatile registers are popped, and control is returned to the calling function. For example,

```
add      RSP, fixed-allocation-size
```

```
pop      R13
pop      R14
pop      R15
ret
```

If a frame pointer is used in the function, then the stack must be trimmed to its fixed allocation prior to the execution of the epilog. This is technically not part of the epilog. For example, the following epilog could be used to undo the prolog previously used:

```
lea      RSP, -128[R13]
; epilogue proper starts here
add      RSP, fixed-allocation-size
pop      R13
pop      R14
pop      R15
ret
```

In practice, when a frame pointer is used, there is no good reason to adjust RSP in two steps, so the following epilog would be used instead:

```
lea      RSP, fixed-allocation-size - 128[R13]
pop      R13
pop      R14
pop      R15
ret
```

These are the only legal forms for an epilog. It must consist of either an `add RSP,constant` or `lea RSP,constant[FPReg]`, followed by a series of zero or more 8-byte register pops and a return or a jmp. (Only a subset of jmp statements are allowable in the epilog. These are exclusively of the class of jmps with ModRM memory references where ModRM mod field value 00. The use of jmps in the epilog with ModRM mod field value 01 or 10 is prohibited. See Table A-15 in the AMD x86-64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions, for more info on the allowable ModRM references.). No other code can appear. In particular, nothing can be scheduled within an epilog, including loading of a return value.

Note that, when a frame pointer is not used, the epilog must use `add RSP,constant` to deallocate the fixed part of the stack. It may not use `lea RSP,constant[RSP]` instead. This restriction exists so the unwind code has fewer patterns to recognize when searching for epilogs.

Following these rules allows the unwind code to determine that an epilog is currently being executed and to simulate execution of the remainder of the epilog to allow recreating the context of the calling function.

See Also

Reference

x64 Software Conventions

Exception Handling (x64)

This section discusses structured exception handling and C++ exception handling behavior on the x64.

- Unwind Data for Exception Handling, Debugger Support
- The Unwind Procedure
- The Language Specific Handler
- Unwind Helpers for MASM
- Unwind Data Definitions in C

For more information, see Exception Handling in Visual C++.

See Also

Reference

x64 Software Conventions

Unwind Data for Exception Handling, Debugger Support

This section discusses the data structures required for exception handling and debugging support.

- struct RUNTIME_FUNCTION
- struct UNWIND_INFO
- struct UNWIND_CODE
- Chained Unwind Info Structures

See Also

Reference

Exception Handling (x64)

struct RUNTIME_FUNCTION

Table-based exception handling requires a table entry for all functions that allocate stack space or call another function (for example, nonleaf functions). Function table entries have the format:

ULONG	Function start address
ULONG	Function end address
ULONG	Unwind info address

The RUNTIME_FUNCTION structure must be DWORD aligned in memory. All addresses are image relative, that is, they are 32-bit offsets from the starting address of the image that contains the function table entry. These entries are sorted, and put in the .pdata section of a PE32+ image. For dynamically generated functions [JIT compilers], the runtime to support these functions must either use RtlInstallFunctionCallback or RtlAddFunctionTable to provide this information to the operating system. Failure to do so will result in unreliable exception handling and debugging of processes.

See Also

Reference

Unwind Data for Exception Handling, Debugger Support

struct UNWIND_INFO

The unwind data info structure is used to record the effects a function has on the stack pointer and where the nonvolatile registers are saved on the stack:

UBYTE: 3	Version
UBYTE: 5	Flags
UBYTE	Size of prolog
UBYTE	Count of unwind codes
UBYTE: 4	Frame Register
UBYTE: 4	Frame Register offset (scaled)
USHORT * n	Unwind codes array
variable	Can either be of form (1) or (2) below

(1) Exception Handler

ULONG	Address of exception handler
variable	Language-specific handler data (optional)

(2) Chained Unwind Info

ULONG	Function start address
ULONG	Function end address
ULONG	Unwind info address

The UNWIND_INFO structure must be DWORD aligned in memory. The meaning of each field is as follows:

Version

Version number of the unwind data, currently 1.

Flags

Three flags are currently defined:

UNW_FLAG_EHANDLER The function has an exception handler that should be called when looking for functions that need to examine exceptions.

UNW_FLAG_UHANDLER The function has a termination handler that should be called when unwinding an exception.

UNW_FLAG_CHAININFO This unwind info structure is not the primary one for the procedure. Instead, the chained unwind info entry is the contents of a previous RUNTIME_FUNCTION entry. See the following text for an explanation of chained unwind info structures. If this flag is set, then the UNW_FLAG_EHANDLER and UNW_FLAG_UHANDLER flags must be cleared. Also, the frame register and fixed-stack allocation fields must have the same values as in the primary unwind info.

Size of prolog

Length of the function prolog in bytes.

Count of unwind codes

This is the number of slots in the unwind codes array. Note that some unwind codes (for example, UWOP_SAVE_NONVOL) require more than one slot in the array.

Frame register

If nonzero, then the function uses a frame pointer, and this field is the number of the nonvolatile register used as the frame pointer, using the same encoding for the operation info field of UNWIND_CODE nodes.

Frame register offset (scaled)

If the frame register field is nonzero, then this is the scaled offset from RSP that is applied to the FP reg when it is established. The actual FP reg is set to RSP + 16 * this number, allowing offsets from 0 to 240. This permits pointing the FP reg into the middle of the local stack allocation for dynamic stack frames, allowing better code density through shorter instructions (more instructions can use the 8-bit signed offset form).

Unwind codes array

This is an array of items that explains the effect of the prolog on the nonvolatile registers and RSP. See the section on UNWIND_CODE for the meanings of individual items. For alignment purposes, this array will always have an even number of entries, with the final entry potentially unused (in which case the array will be one longer than indicated by the count of unwind codes field).

Address of exception handler

This is an image-relative pointer to either the function's language-specific exception/termination handler (if flag UNW_FLAG_CHAININFO is clear and one of the flags UNW_FLAG_EHANDLER or UNW_FLAG_UHANDLER is set).

Language-specific handler data

This is the function's language-specific exception handler data. The format of this data is unspecified and completely determined by the specific exception handler in use.

Chained Unwind Info

If flag UNW_FLAG_CHAININFO is set then the UNWIND_INFO structure ends with three UWORDs. These UWORDs represent the RUNTIME_FUNCTION information for the function of the chained unwind.

See Also

Reference

Unwind Data for Exception Handling, Debugger Support

struct UNWIND_CODE

The unwind code array is used to record the sequence of operations in the prolog that affect the nonvolatile registers and RSP. Each code item has the following format:

UBYTE	Offset in prolog
UBYTE: 4	Unwind operation code
UBYTE: 4	Operation info

The array is sorted by descending order of offset in the prolog.

Offset in prolog

Offset from the beginning of the prolog of the end of the instruction that performs this operation, plus 1 (that is, the offset of the start of the next instruction).

Unwind operation code

Note: Certain operation codes require an unsigned offset to a value in the local stack frame. This offset is from the start (lowest address) of the fixed stack allocation. If the Frame Register field in the UNWIND_INFO is zero, this offset is from RSP. If the Frame Register field is nonzero, this is the offset from where RSP was located when the FP reg was established. This equals the FP reg minus the FP reg offset (16 * the scaled frame register offset in the UNWIND_INFO). If an FP reg is used, then any unwind code taking an offset must only be used after the FP reg is established in the prolog.

For all opcodes except UWOP_SAVE_XMM128 and UWOP_SAVE_XMM128_FAR, the offset will always be a multiple of 8, because all stack values of interest are stored on 8 byte boundaries (the stack itself is always 16-byte aligned). For operation codes that take a short offset (less than 512K), the final USHORT in the nodes for this code holds the offset divided by 8. For operation codes that take a long offset (512K <= offset < 4GB), the final two USHORT nodes for this code hold the offset (in little-endian format).

For the opcodes UWOP_SAVE_XMM128 and UWOP_SAVE_XMM128_FAR, the offset will always be a multiple of 16, since all 128-bit XMM operations must occur on 16-byte aligned memory. Therefore, a scale factor of 16 is used for UWOP_SAVE_XMM128, permitting offsets of less than 1M.

The unwind operation code is one of the following:

UWOP_PUSH_NONVOL (0)1 node

Push a nonvolatile integer register, decrementing RSP by 8. The operation info is the number of the register. Note that, because of the constraints on epilogs, UWOP_PUSH_NONVOL unwind codes must appear first in the prolog and correspondingly, last in the unwind code array. This relative ordering applies to all other unwind codes except UWOP_PUSH_MACHFRAME.

UWOP_ALLOC_LARGE (1)2 or 3 nodes

Allocate a large-sized area on the stack. There are two forms. If the operation info equals 0, then the size of the allocation divided by 8 is recorded in the next slot, allowing an allocation up to 512K – 8. If the operation info equals 1, then the unscaled size of the allocation is recorded in the next two slots in little-endian format, allowing allocations up to 4GB – 8.

UWOP_ALLOC_SMALL (2)1 node

Allocate a small-sized area on the stack. The size of the allocation is the operation info field * 8 + 8, allowing allocations from 8 to 128 bytes.

The unwind code for a stack allocation should always use the shortest possible encoding:

Allocation Size	**Unwind Code**
8 to 128 bytes	UWOP_ALLOC_SMALL
136 to 512K-8 bytes	UWOP_ALLOC_LARGE, operation info = 0

512K to 4G–8 bytes	UWOP_ALLOC_LARGE, operation info = 1

UWOP_SET_FPREG (3)1 node

Establish the frame pointer register by setting the register to some offset of the current RSP. The offset is equal to the Frame Register offset (scaled) field in the UNWIND_INFO * 16, allowing offsets from 0 to 240. The use of an offset permits establishing a frame pointer that points to the middle of the fixed stack allocation, helping code density by allowing more accesses to use short instruction forms. Note that the operation info field is reserved and should not be used.

UWOP_SAVE_NONVOL (4)2 nodes

Save a nonvolatile integer register on the stack using a MOV instead of a PUSH. This is primarily used for shrink-wrapping, where a nonvolatile register is saved to the stack in a position that was previously allocated. The operation info is the number of the register. The scaled-by-8 stack offset is recorded in the next unwind operation code slot, as described in the note above.

UWOP_SAVE_NONVOL_FAR (5)3 nodes

Save a nonvolatile integer register on the stack with a long offset, using a MOV instead of a PUSH. This is primarily used for shrink-wrapping, where a nonvolatile register is saved to the stack in a position that was previously allocated. The operation info is the number of the register. The unscaled stack offset is recorded in the next two unwind operation code slots, as described in the note above.

UWOP_SAVE_XMM128 (8)2 nodes

Save all 128 bits of a nonvolatile XMM register on the stack. The operation info is the number of the register. The scaled-by-16 stack offset is recorded in the next slot.

UWOP_SAVE_XMM128_FAR (9)3 nodes

Save all 128 bits of a nonvolatile XMM register on the stack with a long offset. The operation info is the number of the register. The unscaled stack offset is recorded in the next two slots.

UWOP_PUSH_MACHFRAME (10)1 node

Push a machine frame. This is used to record the effect of a hardware interrupt or exception. There are two forms. If the operation info equals 0, the following has been pushed on the stack:

RSP+32	SS
RSP+24	Old RSP
RSP+16	EFLAGS
RSP+8	CS
RSP	RIP

If the operation info equals 1, then the following has instead been pushed:

RSP+40	SS
RSP+32	Old RSP
RSP+24	EFLAGS
RSP+16	CS
RSP+8	RIP
RSP	Error code

This unwind code will always appear in a dummy prolog, which is never actually executed but instead appears before the real entry point of an interrupt routine, and exists only to provide a place to simulate the push of a machine frame. UWOP_PUSH_MACHFRAME records that simulation, which indicates the machine has conceptually done the following:

Pop RIP return address from top of stack into *Temp*

Push SS

Push old RSP

Push EFLAGS

Push CS

Push *Temp*

Push Error Code (if op info equals 1)

The simulated UWOP_PUSH_MACHFRAME operation decrements RSP by 40 (op info equals 0) or 48 (op info equals 1).

Operation info

The meaning of these 4 bits depends upon the operation code. To encode a general-purpose (integer) register, the following mapping is used:

0	RAX
1	RCX
2	RDX
3	RBX
4	RSP
5	RBP
6	RSI
7	RDI
8 to 15	R8 to R15

See Also

Reference

Unwind Data for Exception Handling, Debugger Support

Chained Unwind Info Structures

If the UNW_FLAG_CHAININFO flag is set, then an unwind info structure is a secondary one, and contains an image-relative pointer in the shared exception handler/chained info address field which points to the RUNTIME_FUNCTION entry pointing to the primary unwind info.

Chained info is useful in two situations. First, it can be used for noncontiguous code segments, like those produced by BBT. Using chained info can reduce the size of the necessary unwind info, because there is no need to duplicate the unwind codes array from the primary unwind info.

Chained info can also be used to handle shrink-wrapping of volatile register saves. The compiler may choose to delay saving some volatile registers until outside of the function entry prolog. This can be recorded by having primary unwind info for the portion of the function before the shrink-wrapped code, and then setting up chained info with a non-zero size of prolog, where the unwind codes in the chained info reflect saves of the nonvolatile registers. In that case, the unwind codes will all be instances of UWOP_SAVE_NONVOL. Shrink-wrapping that saves nonvolatile registers using a PUSH is not supported, and shrink-wrapping that modifies RSP with an additional fixed stack allocation is also not supported.

It is possible for an UNWIND_INFO item with UNW_FLAG_CHAININFO set to point to a RUNTIME_FUNCTION entry whose UNWIND_INFO item also has UNW_FLAG_CHAININFO set (multiple shrink-wrapping). Eventually, following the chained unwind info pointers will arrive at an UNWIND_INFO item with UNW_FLAG_CHAININFO cleared, which is the primary UNWIND_INFO item pointing to the actual procedure entry point.

See Also

Reference

Unwind Data for Exception Handling, Debugger Support

The Unwind Procedure

The unwind code array is sorted into descending order. When an exception occurs, the complete context is stored by the operating system in a context record. The exception dispatch logic is then invoked, which repeatedly executes the following steps to find an exception handler.

1. Use the current RIP stored in the context record to search for a RUNTIME_FUNCTION table entry that describes the current function (or function portion, in the case of chained UNWIND_INFO entries).
2. If no function table entry is found, then it is in a leaf function, and RSP will directly address the return pointer. The return pointer at [RSP] is stored in the updated context, the simulated RSP is incremented by 8, and step 1 is repeated.
3. If a function table entry is found, RIP can lie within three regions a) in an epilog, b) in the prolog, or c) in code that may be covered by an exception handler.
   - Case a) If the RIP is within an epilog, then control is leaving the function, there can be no exception handler associated with this exception for this function, and the effects of the epilog must be continued to compute the context of the caller function. To determine if the RIP is within an epilog, the code stream from RIP on is examined. If that code stream can be matched to the trailing portion of a legitimate epilog, as described in section CNDJ6nn5us4RjlIAqgBLqQsCAAAACAAAAA4AAABfAFIAZQBmADQAOQA2ADAAOQAyADQAMgA1AAAA REF _Ref496092425 \r \h 0, then it is in an epilog, and the remaining portion of the epilog is simulated, with the context record updated as each instruction is processed. After this, step 1 is repeated.
   - Case b) If the RIP lies within the prologue, then control has not entered the function, there can be no exception handler associated with this exception for this function, and the effects of the prolog must be undone to compute the context of the caller function. The RIP is within the prolog if the distance from the function start to the RIP is less than or equal to the prolog size encoded in the unwind info. The effects of the prolog are unwound by scanning forward through the unwind codes array for the first entry with an offset less than or equal to the offset of the RIP from the function start, then undoing the effect of all remaining items in the unwind code array. Step 1 is then repeated.
   - Case c) If the RIP is not within a prolog or epilog and the function has an exception handler (UNW_FLAG_EHANDLER is set), then the language specific handler is called. The handler scans its data and calls filter functions as appropriate. The language specific handler can return that the exception was handled or that the search is to be continued. It can also initiate an unwind directly.
4. If the language specific handler returns a handled status, then execution is continued using the original context record.
5. If there is no language specific handler or the handler returns a “continue search” status, then the context record must be unwound to the state of the caller. This is accomplished by processing all of the unwind code array elements, undoing the effect of each. Step 1 is then repeated.

When chained unwind info is involved, these basic steps are still followed. The only difference is that, while walking the unwind code array to unwind a prolog’s effects, once the end of the array is reached, it is then linked to the parent unwind info and the entire unwind code array found there is walked. This linking continues until arriving at an unwind info without the UNW_CHAINED_INFO flag and finish walking its unwind code array.

The smallest set of unwind data is 8 bytes. This would represent a function that only allocated 128 bytes of stack or less, and possibly saved one nonvolatile register. This is also the size of a chained unwind info structure for a zero-length prolog with no unwind codes.

See Also

Reference

Exception Handling (x64)

The Language Specific Handler

The relative address of the language specific handler is present in the UNWIND_INFO whenever flags UNW_FLAG_EHANDLER or UNW_FLAG_UHANDLER are set. As described in the previous section, the language specific handler is called as part of the search for an exception handler or as part of an unwind. It has the following prototype:

```
typedef EXCEPTION_DISPOSITION (*PEXCEPTION_ROUTINE) (
    IN PEXCEPTION_RECORD ExceptionRecord,
    IN ULONG64 EstablisherFrame,
    IN OUT PCONTEXT ContextRecord,
    IN OUT PDISPATCHER_CONTEXT DispatcherContext
);
```

ExceptionRecord supplies a pointer to an exception record, which has the standard Win64 definition.

EstablisherFrame is the address of the base of the fixed stack allocation for this function.

ContextRecord points to the exception context at the time the exception was raised (in the exception handler case) or the current "unwind" context (in the termination handler case).

DispatcherContext points to the dispatcher context for this function. It has the following definition:

```
typedef struct _DISPATCHER_CONTEXT {
    ULONG64 ControlPc;
    ULONG64 ImageBase;
    PRUNTIME_FUNCTION FunctionEntry;
    ULONG64 EstablisherFrame;
    ULONG64 TargetIp;
    PCONTEXT ContextRecord;
    PEXCEPTION_ROUTINE LanguageHandler;
    PVOID HandlerData;
} DISPATCHER_CONTEXT, *PDISPATCHER_CONTEXT;
```

ControlPc is the value of RIP within this function. This is either an exception address or the address at which control left the establishing function. This is the RIP that will be used to determine if control is within some guarded construct within this function (for example, a __try block for __try/__except or __try/__finally).

ImageBase is the image base (load address) of the module containing this function, to be added to the 32-bit offsets used in the function entry and unwind info to record relative addresses.

FunctionEntry supplies a pointer to the RUNTIME_FUNCTION function entry holding the function and unwind info image-base relative addresses for this function.

EstablisherFrame is the address of the base of the fixed stack allocation for this function.

TargetIp Supplies an optional instruction address that specifies the continuation address of the unwind. This address is ignored if **EstablisherFrame** is not specified.

ContextRecord points to the exception context, for use by the system exception dispatch/unwind code.

LanguageHandler points to the language-specific language handler routine being called.

HandlerData points to the language-specific handler data for this function.

See Also

Reference

Exception Handling (x64)

Unwind Helpers for MASM

In order to write proper assembly routines, there are a set of pseudo-operations that can be used in parallel with the actual assembly instructions to create the appropriate .pdata and .xdata. There are also a set of macros that provide simplified use of the pseudo-operations for their most common uses.

Remarks

The following topics are in this section:

- Raw Pseudo Operations
- MASM Macros

See Also

Reference

Exception Handling (x64)

Raw Pseudo Operations

This topic lists the pseudo operations.

Remarks

Pseudo operation	Description
PROC FRAME [:ehandler]	Causes MASM to generate a function table entry in .pdata and unwind information in .xdata for a function's structured exception handling unwind behavior. If ehandler is present, this proc is entered in the .xdata as the language specific handler. When the FRAME attribute is used, it must be followed by an .ENDPROLOG directive. If the function is a leaf function (as defined in Function Types) the FRAME attribute is unnecessary, as are the remainder of these pseudo-operations.
.PUSHREG reg	Generates a UWOP_PUSH_NONVOL unwind code entry for the specified register number using the current offset in the prologue. This should only be used with nonvolatile integer registers. For pushes of volatile registers, use an .ALLOCSTACK 8, instead
.SETFRAME reg, offset	Fills in the frame register field and offset in the unwind information using the specified register and offset. The offset must be a multiple of 16 and less than or equal to 240. This directive also generates a UWOP_SET_FPREG unwind code entry for the specified register using the current prologue offset.
.ALLOCSTACK size	Generates a UWOP_ALLOC_SMALL or a UWOP_ALLOC_LARGE with the specified size for the current offset in the prologue. The size operand must be a multiple of 8.
.SAVEREG reg, offset	Generates either a UWOP_SAVE_NONVOL or a UWOP_SAVE_NONVOL_FAR unwind code entry for the specified register and offset using the current prologue offset. MASM will choose the most efficient encoding. Offset must be positive, and a multiple of 8. Offset is relative to the base of the procedure’s frame, which is generally in RSP, or, if using a frame pointer, the unscaled frame pointer.
.SAVEXMM128 reg, offset	Generates either a UWOP_SAVE_XMM128 or a UWOP_SAVE_XMM128_FAR unwind code entry for the specified XMM register and offset using the current prologue offset. MASM will choose the most efficient encoding. Offset must be positive, and a multiple of 16. Offset is relative to the base of the procedure’s frame, which is generally in RSP, or, if using a frame pointer, the unscaled frame pointer.
.PUSHFRAME [code]	Generates a UWOP_PUSH_MACHFRAME unwind code entry. If the optional code is specified, the unwind code entry is given a modifier of 1. Otherwise the modifier is 0.
.ENDPROLOG	Signals the end of the prologue declarations. Must occur in the first 255 bytes of the function.

Here is a sample function prolog with proper usage of most of the opcodes:

```
sample PROC FRAME
   db      048h; emit a REX prefix, to enable hot-patching
push rbp
.pushreg rbp
sub rsp, 040h
.allocstack 040h
lea rbp, [rsp+020h]
.setframe rbp, 020h
movdqa [rbp], xmm7
```

```
.savexmm128 xmm7, 020h;the offset is from the base of the frame
;not the scaled offset of the frame
mov [rbp+018h], rsi
.savereg rsi, 038h
mov [rsp+010h], rdi
.savereg rdi, 010h; you can still use RSP as the base of the frame
; or any other register you choose
.endprolog

; you can modify the stack pointer outside of the prologue (similar to alloca)
; because we have a frame pointer.
; if we didn’t have a frame pointer, this would be illegal
; if we didn’t make this modification,
; there would be no need for a frame pointer

sub rsp, 060h

; we can unwind from the following AV because of the frame pointer

mov rax, 0
mov rax, [rax] ; AV!

; restore the registers that weren’t saved with a push
; this isn’t part of the official epilog, as described in section 2.5

movdqa xmm7, [rbp]
mov rsi, [rbp+018h]
mov rdi, [rbp-010h]

; Here’s the official epilog

lea rsp, [rbp-020h]
pop rbp
ret
sample ENDP
```

See Also

Reference

Unwind Helpers for MASM

MASM Macros

In order to simplify the use of the Raw Pseudo Operations, there are a set of macros, defined in ksamd64.inc, which can be used to create typical procedure prologues and epilogues.

Remarks

Macro	Description
alloc_stack (n)	Allocates a stack frame of n bytes (using sub rsp, n), and emits the appropriate unwind information (.allocstack n)
save_reg reg, loc	Saves a nonvolatile register reg on the stack at RSP offset loc, and emits the appropriate unwind information. (.save reg reg, loc)
push_reg reg	Pushes a nonvolatile register reg on the stack, and emits the appropriate unwind information. (.pushreg reg)
rex_push_reg reg	Save a nonvolatile register on the stack using a 2 byte push, and emits the appropriate unwind information (.pushreg reg) This should be used if the push is the first instruction in the function to ensure that the function is hot-patchable.
save_xmm128 reg, loc	Saves a nonvolatile XMM register reg on the stack at RSP offset loc, and emits the appropriate unwind information (.savexmm128 reg, loc)
set_frame reg, offset	Sets the frame register reg to be the RSP + offset (using a mov, or an lea), and emits the appropriate unwind information (.set_frame reg, offset)
push_eflags	Pushes the eflags with a pushfq instruction, and emits the appropriate unwind information (.alloc_stack 8)

Here is a sample function prolog with proper usage of the macros:

```
SkFrame struct
Fill    dq ?; fill to 8 mod 16
SavedRdi dq ?; saved register RDI
SavedRsi dq ?; saved register RSI
SkFrame ends

sampleFrame struct
Filldq?; fill to 8 mod 16
SavedRdidq?; Saved Register RDI
SavedRsi  dq?; Saved Register RSI
sampleFrame ends

sample2 PROC FRAME
alloc_stack(sizeof sampleFrame)
save_reg rdi, sampleFrame.SavedRdi
save_reg rsi, sampleFrame.SavedRsi
.end_prolog

; function body

mov rsi, sampleFrame.SavedRsi[rsp]
mov rdi, sampleFrame.SavedRdi[rsp]

; Here’s the official epilog

add rsp, (sizeof sampleFrame)
ret
sample2 ENDP
```

See Also

Reference

Unwind Helpers for MASM

Unwind Data Definitions in C

The following is a C description of the unwind data:

```
typedef enum _UNWIND_OP_CODES {
    UWOP_PUSH_NONVOL = 0, /* info == register number */
    UWOP_ALLOC_LARGE,     /* no info, alloc size in next 2 slots */
    UWOP_ALLOC_SMALL,     /* info == size of allocation / 8 - 1 */
    UWOP_SET_FPREG,       /* no info, FP = RSP + UNWIND_INFO.FPRegOffset*16 */
    UWOP_SAVE_NONVOL,     /* info == register number, offset in next slot */
    UWOP_SAVE_NONVOL_FAR, /* info == register number, offset in next 2 slots */
    UWOP_SAVE_XMM128,     /* info == XMM reg number, offset in next slot */
    UWOP_SAVE_XMM128_FAR, /* info == XMM reg number, offset in next 2 slots */
    UWOP_PUSH_MACHFRAME   /* info == 0: no error-code, 1: error-code */
} UNWIND_CODE_OPS;

typedef union _UNWIND_CODE {
    struct {
        UBYTE CodeOffset;
        UBYTE UnwindOp : 4;
        UBYTE OpInfo   : 4;
    };
    USHORT FrameOffset;
} UNWIND_CODE, *PUNWIND_CODE;

#define UNW_FLAG_EHANDLER  0x01
#define UNW_FLAG_UHANDLER  0x02
#define UNW_FLAG_CHAININFO 0x04

typedef struct _UNWIND_INFO {
    UBYTE Version       : 3;
    UBYTE Flags         : 5;
    UBYTE SizeOfProlog;
    UBYTE CountOfCodes;
    UBYTE FrameRegister : 4;
    UBYTE FrameOffset   : 4;
    UNWIND_CODE UnwindCode[1];
/*  UNWIND_CODE MoreUnwindCode[((CountOfCodes + 1) & ~1) - 1];
*   union {
*       OPTIONAL ULONG ExceptionHandler;
*       OPTIONAL ULONG FunctionEntry;
*   };
*   OPTIONAL ULONG ExceptionData[]; */
} UNWIND_INFO, *PUNWIND_INFO;

typedef struct _RUNTIME_FUNCTION {
    ULONG BeginAddress;
    ULONG EndAddress;
    ULONG UnwindData;
} RUNTIME_FUNCTION, *PRUNTIME_FUNCTION;

#define GetUnwindCodeEntry(info, index) \
    ((info)->UnwindCode[index])

#define GetLanguageSpecificDataPtr(info) \
    ((PVOID)&GetUnwindCodeEntry((info),((info)->CountOfCodes + 1) & ~1))

#define GetExceptionHandler(base, info) \
    ((PEXCEPTION_HANDLER)((base) + *(PULONG)GetLanguageSpecificDataPtr(info)))

#define GetChainedFunctionEntry(base, info) \
    ((PRUNTIME_FUNCTION)((base) + *(PULONG)GetLanguageSpecificDataPtr(info)))

#define GetExceptionDataPtr(info) \
    ((PVOID)((PULONG)GetLanguageSpecificData(info) + 1)
```

See Also

Reference

Exception Handling (x64)

Intrinsics and Inline Assembly

One of the constraints for the x64 compiler is to have no inline assembler support. This means that functions that cannot be written in C or C++ will either have to be written as subroutines or as intrinsic functions supported by the compiler. Certain functions are performance sensitive while others are not. Performance-sensitive functions should be implemented as intrinsic functions. In general, this will be the same list of intrinsic functions implemented for ALPHA and the Itanium but will include x64 specific functions as well.

The intrinsics supported by the compiler are described in Compiler Intrinsics.

See Also

Reference

x64 Software Conventions

Image Format

The executable image format is PE32+. Executable images (both DLLs and EXEs) are restricted to a maximum size of 2 gigabytes, so relative addressing with a 32-bit displacement can be used to address static image data. This data includes the import address table, string constants, static global data, and so on.

See Also

Reference

x64 Software Conventions

Deployment (C++)

Deployment is the process by which you distribute a finished application or component to be installed on other computers. Deployment is a multi-stage process that starts when an application is created on a developer's computer and ends when it is installed and ready to run on a user's computer.

Visual Studio provides two different technologies for deploying Windows applications: ClickOnce deployment or Windows Installer deployment.

- ClickOnce can be used to deploy C++ applications that target the Common Language Runtime (mixed, pure, and verifiable assemblies). You can use Windows Installer to deploy a managed application also, but ClickOnce takes advantage of .NET Framework security features (such as manifest signing) and is therefore preferred over Windows Installer deployment. ClickOnce does not support deployment of native C++ applications. For more information, see ClickOnce Deployment for Visual C++ Applications.
- Windows Installer technology can be used for deploying both native C++ applications and C++ applications that target the Common Language Runtime.

For a detailed comparison of ClickOnce and Windows Installer deployment, see Choosing a Deployment Strategy.

This section discusses how to ensure that a native Visual C++ application runs on any computer that provides a supported target platform, which files you need to include in your installation package, and what are the recommended ways to redistribute the Visual C++ components on which your application depends.

If you are familiar with the deployment process and are primarily interested in learning about changes made in Visual Studio 2005, you may start with Choosing a Deployment Method. For examples of deployment please see, Deployment Examples.

The following topics discuss deployment of Visual C++ applications in more detail.

In This Section

Deployment Concepts

Understanding Dependencies of a Visual C++ Application

Determining Which DLLs to Redistribute

Choosing a Deployment Method

Redistributing Visual C++ Files

Deployment Examples

Deploying Windows-Client Applications

Redistributing Web Applications

Deploying ATL Server Applications

Version Checking of Redistributable Files

ClickOnce Deployment for Visual C++ Applications

Running a C++ /clr Application on a Previous Runtime Version

See Also

Concepts

Troubleshooting C/C++ Isolated Applications and Side-by-side Assemblies

Other Resources

Building C/C++ Isolated Applications and Side-by-side Assemblies
Deploying .NET Framework Applications
Packaging Device Solutions for Deployment

Deployment Concepts

This section discusses the main considerations for deploying C++ applications.

Windows Installer Deployment in C++

Visual C++ projects typically use the traditional Windows Installer setup for deployment. To prepare a Windows Installer deployment, you package your application in a setup.exe file and distribute that file, together with an installer package (.msi). Users then run setup.exe to install your application.

You package your application by adding a Setup project to your solution; when built, it creates the setup and installer package files that you distribute to users. For more information, see Choosing a Deployment Method.

Library Dependencies

When a C/C++ application is built using functionality provided by the Visual C++ libraries, it becomes dependent on the presence of those libraries at runtime. In order for the application to run, it must link, either statically or dynamically, to the necessary Visual C++ libraries. If an application dynamically links to a Visual C++ library, then when it runs that library must be present so it can be loaded. On the other hand, if the application statically links to a Visual C++ library, then it does not need the corresponding DLLs to be present on the user's computer. Static linking, however, has some negative effects, such as the increasing the size of the application files, and making maintenance potentially more difficult. For more information, see Advantages of Using DLLs.

Visual C++ libraries are packaged as DLLs, and all necessary libraries for C/C++ applications are installed by Visual Studio on the developer's computer. However, when deploying your application to your users, it is not feasible in most cases to require them to install Visual Studio in order to run your application. It is important to be able to redistribute just the parts of Visual C++ that are required by your application to run correctly.

To understand what files from Visual C++ you need to redistribute with your applications, please see Determining Which DLLs to Redistribute. To explore what packaging and deployment alternatives are available, please see Choosing a Deployment Method. For examples and suggestions on how to resolve some other related issues, please see Deployment Examples and Troubleshooting C/C++ Isolated Applications and Side-by-side Assemblies.

See Also

Concepts

Understanding Dependencies of a Visual C++ Application

Other Resources

Deployment (C++)

Windows Installer Deployment

Understanding Dependencies of a Visual C++ Application

One way to determine what Visual C++ libraries your applications depends on is to look at the **General Tab** in the **Project Settings** dialog box for your project. You might see, for example, that your project is using the MFC and ATL libraries. If you chose **Use MFC in a Dynamic Library**, your application at runtime depends on MFC DLLs, such as mfc80.dll. If your application does not use MFC or ATL, it might still depend on the CRT library if you have chosen **Multi-Threaded Debug DLL (/MDd)** for the **Runtime Library** on the **Code Generation** tab in the C/C++ branch of the **Project Properties**.

The most comprehensive way to determine which DLLs your application depends on is to open the application using the Dependency Walker (depends.exe), which ships with Visual C++. Depends.exe is installed to \Microsoft Visual Studio 2005\Common7\Tools\bin. Note that depends.exe is only installed if you select the Win32 Platform SDK Tool, which is in the Visual C++ Tools category of the Visual C++ custom installation.

By using depends.exe, or the DUMPBIN utility with the /DEPENDENTS option, you can see a list of DLLs that statically link to your application and a list of the application's delay-loaded DLLs.

To see which DLLs, such as ActiveX controls, are dynamically loaded, use the profiling feature of depends.exe. Then, test your application until you are sure that all code paths have been exercised. When you end the profiling session, depends.exe shows which DLLs were dynamically loaded.

When using depends.exe, be aware that a DLL might have a dependency on another DLL or on a specific version of a DLL. You can use depends.exe on either the development computer or on a target computer. On the development computer, depends.exe reports the DLLs that are required to support an application. If you have trouble getting an application to run on a target computer, you can copy depends.exe to the target computer and open the application in depends.exe. Depends.exe reports which of the application's DLLs are either missing or present with an incorrect version.

After you have a complete list of DLLs your application depends on, you can determine which of these DLLs you have to redistribute with your application when deploying to another computer. In most cases you do not need to redistribute system DLLs, but you may have to redistribute DLLs for Visual C++ libraries. For more information see Determining Which DLLs to Redistribute.

See Also

Other Resources

Deployment (C++)

Determining Which DLLs to Redistribute

The first step in determining which files you need to redistribute with your application is to clearly understand the dependencies of this application and collect a list of dependent DLLs by using, for example, Dependency Walker (depends.exe) as described in Understanding Dependencies of a Visual C++ Application.

As you use DUMPBIN or depends.exe to view your dependencies, use the file list in Redist.txt to see which of the DLLs that your application depends on are Microsoft-supplied DLLs that are part of Visual Studio. Redist.txt is located in the Program Files\Microsoft Visual Studio 2005 directory on the second Visual Studio 2005 product CD or on the DVD. Below is a list of Visual C++ files your application might depend on. However, only files specified in Redist.txt may be redistributed with your application. Note that debug versions of an application are not redistributable and that none of the debug versions of the various Visual C++ dynamic-link libraries (DLLs) are redistributable. For additional information, please see Choosing a Deployment Method.

DLL Name	Visual C++ Library
atl80.dll	Active Template Library
msvcm80.dll msvcp80.dll msvcr80.dll	C Runtime and Standard C++ Libraries
mfc80.dll mfc80u.dll mfcm80.dll mfcm80u.dll	Microsoft Foundation Classes

For additional information on how redistribute these DLLs with your application please see Redistributing Visual C++ Files. For examples, see Deployment Examples.

Normally, you should not need to redistribute system DLLs (for example, Kernel32.dll, User32.dll, Ole32.dll, or ShDocVW.dll) because these DLLs are part of the operating system and are present on any computer with the same version of the operating system installed. However, there might be exceptions, such as in the case where your application is going to run on several versions of Microsoft operating systems. Be sure to read the corresponding licensing agreements if you encounter such a need. If possible, try to get the system DLLs upgraded either through service packs or by small redistributable packages made available by Microsoft. You can search and identify any such available packages from the Knowledge Base articles in the MSDN Library or at http://support.microsoft.com.

See Also

Concepts

Choosing a Deployment Method

Other Resources

Deployment (C++)

Choosing a Deployment Method

In most cases deployment of Visual C++ applications is done with Windows Installer deployment. For more information on deployment methods supported within Visual Studio and alternatives, see Choosing a Deployment Strategy and Deployment Alternatives. ClickOnce deployment for Visual C++ native applications is not supported in Visual Studio 2005; however, it is possible to deploy Visual C++ applications via ClickOnce on the command line. For more information, see ClickOnce Deployment for Visual C++ Applications.

Visual C++ libraries are side-by-side assemblies

Visual Studio 2005 installs Visual C++ libraries as shared side-by-side assemblies. By default, all applications built with Visual Studio 2005 are built as isolated applications with a manifest either embedded as a resource, or accompanying the final binary as an external file. In order to ensure that your Visual C++ application will run on a computer without Visual C++ installed, you may have to redistribute Visual C++ assemblies with your application and ensure they are installed on the target computer.

Redistributing Visual C++ libraries

There are three ways to redistribute Visual C++ DLLs:

1. Using Visual C++ Redistributable Merge Modules to install a particular Visual C++ library as shared side-by-side assemblies into the native assembly cache (WinSxS folder). This is the primary recommended way for redistributing Visual C++ libraries. Access to this folder requires that the installer application be run by a user with administrative rights. For more information please see Redistributing Using Merge Modules. An example of this deployment may be found in How to: Deploy using a Setup and Deployment Project.

2. Using Visual C++ Redistributable Package (VCRedist_x86.exe, VCRedist_x64.exe, VCRedist_ia64.exe) to install all Visual C++ libraries as shared side-by-side assemblies into the native assembly cache (WinSxS folder). This package is installed by Visual Studio into %PROGDIR%\Microsoft Visual Studio 8\SDK\v2.0\Bootstrapper\Packages\ folder and can also be downloaded from the Microsoft download site Microsoft Visual C++ 2005 Redistributable Package (x86). Redistributing Visual C++ libraries using this package is recommended for applications built with Visual C++ Express and for cases when deployment of all Visual C++ libraries at once is desirable. For an example of how to use this package please see How to: Deploy using XCopy.

3. Install a particular Visual C++ assembly as a private assembly for the application using files provide in the Program Files\Microsoft Visual Studio 8\VC\Redist directory. This way is recommended to enable installation of applications by users who do not have administrative rights or when it should be possible to run an application from a share. For an example, please see How to: Deploy using XCopy.

Note
On Windows Server 2000, Windows 95 and Windows 98, the only recommended and supported way to redistribute Visual C++ libraries is by using Redistributable Merge Modules.

When installing Visual C++ libraries using redistributable merge modules, the assemblies are deployed as shared side-by-side assemblies into the native assembly cache (WinSxS folder). Access to this folder requires that the installer application be run by a user with administrative rights.

If an installation is run by a user who does not have administrative rights, the installation fails to deploy the Visual C++ assemblies and the application will not run. Also, some products may allow installation on a per-user basis, but merge modules install libraries into shared locations and impact all users of the system. In both these and similar scenarios, the supported technique is to install the required assemblies as private side-by-side assemblies for a particular user's application.

With this technique, it is enough to copy a folder with DLLs and manifests for dependent assemblies to the application's local folder. The operating system's loader, when executing the application, still looks for dependent assemblies in the WinSxS folder first; however when a corresponding assembly cannot be found, the loader loads a private assembly from this sub-directory.

Incorrect redistribution of Visual C++ libraries may result in run-time errors during execution of an application that depends on them. A list of potential errors and resolutions to these errors can be found in Troubleshooting C/C++ Isolated Applications and Side-by-side Assemblies.

Manifest required

It is not supported to redistribute C/C++ applications that are built without a manifest. Visual C++ libraries cannot be used by

C/C++ applications without a manifest binding the application to these libraries. All C/C++ binaries built in Visual C++ 2005 have to include a manifest describing their dependencies on Visual C++ libraries. This is the default setup of projects in Visual Studio and the default behavior of the linker building the final binary from object code.

It is recommended in all cases that the manifest be internal. However, in the case of an EXE file, the manifest can be external (this scenario is supported but not recommended).

Applications that expect dependent DLLs to be present in either the application-local folder or in a folder indicated by an environment variable may also be vulnerable to security-related exploits. It is also more difficult to service such applications after they have been deployed.

Prefer dynamic linking over static linking

It is not recommended to redistribute C/C++ applications that statically link to Visual C++ libraries. It is often mistakenly assumed that by statically linking your program to Visual C++ libraries it is possible to significantly improve the performance of an application. However the impact on performance of dynamically loading Visual C++ libraries is insignificant in almost all cases. Furthermore, static linking does not allow for servicing the application and its dependent libraries by either the application's author or Microsoft. For example, consider an application that is statically linked to a particular library, running on a client computer with a new version of this library. The application still uses code from the previous version of this library, and does not benefit from library improvements, such as security enhancements. Authors of C/C++ applications are strongly advised to think through the servicing scenario before deciding to statically link to dependent libraries, and use dynamic linking whenever possible.

See Also

Concepts

Choosing a Deployment Strategy
Deployment Examples

Other Resources

Deployment (C++)
Windows Installer Deployment Overview
ClickOnce Deployment

Redistributing Visual C++ Files

When you develop an application using Visual C++, you have the advantage of working on a computer that is not only a good development environment, but is also a convenient environment on which to run and test applications. However, when you want to deploy your application to other computers, you need to redistribute all the files needed to support the application on the target system. For more information on which files you may need to redistribute with your application, see Deployment Examples and Determining Which DLLs to Redistribute.

Only some Visual C++ files are allowed to be redistributed with your application. Please consult the EULA for Visual Studio 2005 and the Redist.txt file to see which files may be redistributed with your application. EULA.txt can be found in the \Setup directory on the first Visual Studio 2005 product CD or on the DVD, and Redist.txt is located in the Program Files\Microsoft Visual Studio 2005 directory on the second CD or on the DVD.

Visual C++ files can be redistributed using either the provided Redistributable Merge Modules, or the Visual C++ Redistributable Package, or by deploying specific Visual C++ assemblies as private side-by-side assemblies in the application local folder.

In the first case, a merge module with a specific Visual C++ assembly has to be included in a Windows Installer package or similar installation package used to deploy this application to another computer. For more information, see Redistributing Using Merge Modules. An example of this deployment may be found in How to: Deploy using a Setup and Deployment Project.

The Visual C++ Redistributable Package (VCRedist_x86.exe, VCRedist_x64.exe, VCRedist_ia64.exe) has to be executed on the target system as a prerequisite to installation of the application. This package installs and registers all Visual C++ libraries.

Note
Redistributing Visual C++ libraries using this package is recommended for applications built with Visual C++ Express and for cases when deployment of all Visual C++ libraries at once is desirable.

For an example of how to use this package please see How to: Deploy using XCopy.

When deploying Visual C++ libraries as a private side-by-side assembly, all DLLs and the manifest file that form this assembly have to be added to the Windows Installer or similar installation package and deployed to the target computer in a subdirectory in the application local folder, following the rules for deploying private assemblies. Information on which Visual C++ DLLs and manifest files are part of which Visual C++ assemblies can be found in Visual C++ Libraries as Shared Side-by-Side Assemblies. A description of the rules that must be followed when deploying private assemblies for loading at runtime can be found in Assembly Searching Sequence. For an example of how to deploy Visual C++ libraries as a private assembly, please see How to: Deploy using XCopy.

Potential run-time errors

If a manifest is present in your application but a required Visual C++ library is not installed in the WinSxS folder, you may get one of the following error messages depending on the version of Windows on which you try to run your application:

- **The application failed to initialize properly (0xc0000135).**
- **This application has failed to start because the application configuration is incorrect. Reinstalling application may fix this problem.**
- **The system cannot execute the specified program.**

If no manifest is present in your application, the error you get depends on whether the Visual C++ library your application depends on is deployed in the application local folder or in a shared folder such as the System folder or WinSxS folder.

Note
It is not supported to redistribute C/C++ applications that are built without a manifest. Visual C++ libraries cannot be used by C/C++ applications without a manifest binding the application to these libraries. For more information, please see Choosing a Deployment Method.

If a Visual C++ library DLL (for example, MSVCR80.DLL) is reachable (ether installed in the application-local folder or in the System folder), you may get

R6034 An application has made an attempt to load the C runtime library incorrectly.

If the DLL is not reachable and Windows cannot load this DLL for your application, you may get

This application has failed to start because MSVCR80.dll was not found. Re-installing the application may fix this problem.

To resolve these errors, you need to make sure that your application is built correctly and Visual C++ libraries are correctly deployed on the target system. To identify the root cause of these run-time errors, follow the steps outlined in Troubleshooting C/C++ Isolated Applications and Side-by-side Assemblies.

See Also

Concepts

Deployment Examples

Troubleshooting C/C++ Isolated Applications and Side-by-side Assemblies

Other Resources

Deployment (C++)

Redistributing Using Merge Modules

Visual Studio installs several merge modules for each Visual C++ component that may be redistributed with your application. They may be found in the Program Files\Common Files\Merge Modules folder. Each of these merge modules may be used to deploy Visual C++ DLLs to the target system. However only non-debug versions of Visual C++ DLLs are allowed to be redistributed. For a complete list of merge modules that are allowed to be used redistributed, please see Redist.txt.

Using Visual C++ redistributable merge modules, you can install Visual C++ assemblies as shared side-by-side assemblies into the native assembly cache (WinSxS folder). Visual Studio itself uses this technique to install these DLLs. Installation of the WinSxS folder requires administrative user rights. If an installation is run by a user who does not have administrative rights, the Visual C++ assemblies cannot be installed, and applications that depend on those DLLs cannot run.

The alternative redistribution approach is to install private side-by-side assemblies of a particular user application. For more information on deploying Visual C++ files as private assemblies see Redistributing Visual C++ Files.

Related Sections

Deploying Windows-Client Applications

See Also

Tasks

How to: Add Merge Modules to a Deployment Project
How to: Deploy using a Setup and Deployment Project

Concepts

Redistributing Visual C++ Files

Redistributing Visual C++ ActiveX Controls

Visual C++ 6.0 supplies ActiveX controls you can use in applications that you then redistribute. These controls are not included in Visual C++ 2005. Per the licensing agreements for Visual C++ 6.0, you can redistribute these controls with applications developed in Visual C++ 2005.

For a list of the redistributable Visual C++ 6.0 ActiveX controls, see Common\Redist\Redist.txt on Disc 1 of the Visual C++ 6.0 product CD.

When distributing applications, you must install and register the .ocx for the ActiveX control (using Regsvr32.exe). In addition, you should make sure the target computer has current versions of the following system files (an asterisk indicates the file needs to be registered):

- Asycfilt.dll
- Comcat.dll *
- Oleaut32.dll *
- Olepro32.dll *
- Stdole2.tlb

If these DLLs are not available on the target system, you need to get them updated using the prescribed mechanism for updating the corresponding operating system. You can download the latest service packs for Windows operating systems from http:// windowsupdate.microsoft.com.

If your application uses one of the ActiveX controls that connects to a database, you must have the Microsoft Data Access Components (MDAC) installed on the target system. For more information, see Redistributing Database Support Files.

When using an ActiveX control that connects to a database, you also need to replicate the data source name on the target computer. You can do this programmatically with functions such as **ConfigDSN**.

Some redistributable ActiveX controls have additional dependencies. For each .ocx file in the Os\System folder on the Visual C++ 6.0 product CD, there is also a .dep file. For each .ocx file that you want to redistribute, look for one or more USES entries in the corresponding .dep file. If a file is listed, you must ensure that the file is on the target computer. Any DLLs directly supporting an .ocx file need to be registered. (For Regsvr32.exe to succeed, the target computer must first contain all of the DLLs the control statically loads.) Furthermore, if a DLL that is listed as a dependency also has a .dep file in the Os\System folder on the Visual C++ 6.0 CD, you must also investigate that .dep file for USES entries.

See Also

Concepts

Redistributing Visual C++ Files

Redistributing Database Support Files

This topic discusses how to redistribute support files for Data Access Objects (DAO) and the database technologies found in the Microsoft Data Access SDK.

Installing DAO Support Files

To get and work with the latest version of DAO, see the following Knowledge Base article:

- Q239114, ACC2000: Updated Version of Microsoft Jet 4.0 Available in Download Center (http://support.microsoft.com/support/kb/articles/q239/1/14.asp)

You can find Knowledge Base articles in the MSDN Library or at http://support.microsoft.com.

Installing Microsoft Data Access SDK Support Files

You should use Mdac_typ.exe to install support for ODBC, OLE DB, ActiveX Data Objects (ADO), and Remote Data Services (RDS). Mdac_typ.exe is in the WCU\MDAC27 folder on the DVD or in the MDAC27 folder on the Visual Studio .NET Prerequisites CD of the Visual C++ .NET product CDs. You can also download Mdac_typ.exe from the Universal Data Access Web site at http://www.microsoft.com/data.

For more information and the latest release details about MDAC, see the Universal Data Access Web site at http://www.microsoft.com/data.

For more information, see Redistributing MDAC. If you are using Visual Studio .NET Setup projects to deploy your application, see Deployment and Dependencies.

See Also

Concepts

Redistributing Visual C++ Files

Redistributing the MFC Library

If you dynamically link your application to the MFC library, you will, at a minimum, need to redistribute Mfc80.dll and Msvcr80.dll with corresponding manifest files for MFC and CRT assemblies. For more information on Visual C++ Assemblies and the DLLs that are part of them, see Visual C++ Libraries as Shared Side-by-Side Assemblies. All MFC DLLs use the shared version of the C run-time library (CRT); thus, Msvcr80.dll is required.

Mfc80.dll does not need to be redistributed with MFC applications if you statically link to the MFC DLL (that is, if you specified **Use MFC in a Static Library** on the **General** tab in the **Project Settings** dialog box). Even though this may work for testing and internal deployment of your applications, it is not a recommended way to redistribute MFC. For more information on the recommended solution for deploying Visual C++ libraries, see Choosing a Deployment Method.

If your application uses the MFC classes implementing the WebBrowser control (for example, CHtmlView Class or CHtmlEditView Class), Microsoft Internet Explorer 4.0 or later must be installed on the target computer. Installing the latest version of Internet Explorer also ensures that the target computer has the latest common control files. Information about installing minimal Internet Explorer components is available in Knowledge Base article Q185375, HOWTO: Create a Single EXE Install of Internet Explorer (http://support.microsoft.com/support/kb/articles/q185/3/75.asp).

If your application uses the MFC database classes, such as CRecordset Class and CRecordView Class, you need to redistribute ODBC and any ODBC drivers that your application uses. For more information, see Redistributing Database Support Files.

If you redistribute an MFC DLL, be sure you redistribute the retail version rather than the debug version. Debug versions of the DLLs are not redistributable. Debug versions of the MFC DLLs have a trailing "d" in their file names, as in Mfc80d.dll.

If you modify MFC sources in any way and rebuild the MFC DLL, you must rename the modified MFC DLL so it will not conflict with the MFC DLL from Visual Studio. Rebuilding and renaming the MFC DLL is not a recommended procedure. For more information, see MFC Technical Note 33.

You may redistribute MFC by using either merge modules that are installed with Visual Studio or by deploying the MFC assembly in a sub-folder of your application-local folder as a private assembly. For more information about redistributing MFC, see Redistributing Visual C++ Files.

Installation of Localized MFC Components

If you decide to localize your application's use of the MFC resources by installing an MFC80 localization DLL, follow the instructions in this topic. Like other shared DLLs, an MFC localization DLL should be installed only if its version is newer than the version that is already installed. Therefore, installing this DLL is similar to installing the other DLLs. For more information, see Version Checking of Redistributable Files.

There are several MFC DLLs supplied for various locales. For example, MFC80deu.dll is the German version and contains version information that identifies it as the German locale. If you install any locale DLL, you must ensure that the locale for which the DLL is intended matches the locale of the installed Windows system. When you install the DLL, you must rename it to MFC80loc.dll.

Visual C++ provides two merge modules related to localization:

- MFC_LOC_E.msm, which contains the localization DLLs for the European versions.
- MFC_LOC_FE.msm, which contains the localization DLLs for the Asian versions.

When you use one of these merge modules with an appropriate installer that supports merge modules, it identifies and installs the appropriate localized version of the DLL in the target system and renames it to MFC80loc.dll.

You should never install an MFC80loc.dll on an English system. English resources are built into MFC80.dll, and it is faster to load them from that DLL instead of searching (and loading) an MFC localization DLL first.

For more information about localizing MFC applications, see MFC Technical Note 57 and Knowledge Base article Q208983, "How to: Using MFC LOC DLLs" (http://support.microsoft.com/support/kb/articles/q208/9/83.asphttp://support.microsoft.com/support/kb/articles/q208/9/83.asp)

You may redistribute MFC localized DLLs by deploying the MFC assembly in your application-local folder as a private assembly. For more information about redistributing Visual C++ libraries as private assemblies, see Redistributing Visual C++ Files.

See Also

Concepts

Redistributing Visual C++ Files

Redistributing ATL and OLE DB Templates

An Active Template Library (ATL) or OLE DB Templates project can be built for a MinDependency or a MinSize configuration.

A MinDependency configuration is what you get when you set the **Use of ATL** property to **Static Link to ATL** on the **General** property page and set the **Runtime Library** property to **Multi-threaded (/MT)** on the **Code Generation** property page (C/C++ folder).

A MinSize configuration is what you get when you set the **Use of ATL** property to **Dynamic Link to ATL** and the **Minimize CRT Use in ATL** to **Yes** on the **General** property page, or set the **Runtime Library** property to **Multi-threaded DLL (/MD)** on the **Code Generation** property page (C/C++ folder). See Using _ATL_MIN_CRT in Your ATL Project.

MinSize makes the output file as small as possible but requires that ATL80.dll and Msvcr80.dll (if you selected the **Multi-threaded DLL (/MD)** option) are on the target computer. ATL80.dll should be registered on the target computer to ensure that all ATL functionality is present. ATL80.dll contains ANSI and Unicode exports.

If you build your ATL or OLE DB Templates project for a MinDependency target, you do not need to install and register ATL80.dll on the target computer, although you might get a larger program image.

If you redistribute an ATL executable application, you must register the .exe file (and any controls inside it) by issuing the following command:

```
filename /regserver
```

where `filename` is the name of the executable file.

For OLE DB Templates applications, ensure that the target computer has the latest versions of Microsoft Data Access Components (MDAC) files. For more information, see Redistributing Database Support Files.

Installation of Localized ATL

If you use an ActiveX control that depends on Msstkprp.dll (typically they will be ATL-based controls) and you want to ship a localized version of the ActiveX control, you can find a localized version of Msstkprp.dll at http://msdn.microsoft.com/vbasic/downloads/ipdk.asp

See Also

Concepts

Redistributing Visual C++ Files

Redistributing Common Controls

Microsoft Visual C++ 2005 supports developing applications using the latest Windows common controls (Comctl32.dll) available with the latest version of Internet Explorer. If you use any of the newer controls from the latest Comctl32.dll, you need the latest version of Internet Explorer on the target computer to ensure that your application runs properly. You can install a version of Comctl32.dll that is compatible with Visual C++ applications by accessing http://www.microsoft.com/msdownload/platformsdk/sdkupdate/psdkredist.htm.

See Also

Concepts

Redistributing Visual C++ Files

Deployment Examples

This section contains examples demonstrating how to deploy Visual C++ applications built with Visual Studio 2005.

Deployment scenarios

The first two examples are based on the scenarios in Scenarios for Deployment Examples. The third example covers the common situation of preparing a test machine for remote debugging.

Examples

How to: Deploy using a Setup and Deployment Project

How to: Deploy using XCopy

Preparing a Test Machine To Run a Debug Executable

See Also

Tasks

Walkthrough: Deploying a ClickOnce Application Manually

Other Resources

Deployment (C++)

Scenarios for Deployment Examples

Let us assume we have MyApplication.exe and MyLibrary.DLL, both built with MFC. MyApplication.exe depends on MyLibray.DLL, both use MFC in a shared DLL, and both can be native or mixed (/clr) binaries. In the simplest case, both are generated by the wizard with no change to default settings. The examples in this section describe how to deploy this application to another computer where Visual Studio is not installed. This section primarily covers deploying the release version of the application; however, changes required for deploying the debug version of an application are pointed out.

Note
Redistributing debug VC++ programs is not permitted by the EULA. You can only do this for testing purposes internally. See the EULA for Visual Studio 2005.

Initial setup

There are three computers to consider in this scenario.

The development computer is the one on which the application is built. It has Visual Studio 2005 (STD, PRO, or TS) installed.

The two deployment target computers do not have Visual Studio 2005 installed. Deployment target 1 is a computer running an operating system that supports manifest-based binding of applications to their dependencies (Windows XP Home Edition, Windows XP Professional, Windows Server 2003, Windows Vista). Deployment target 2 runs an operating system without such support (Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows 2000).

The goal is to build the application on the development computer, deploy it to the two target computers, and run it.

Preparation

When you have built a Visual C++ binary that is going to be run on another computer, you need to determine which DLLs this binary depends on. A useful tool for this is Dependency Walker. In this scenario, you must consider the Visual C++ DLLs, in particular CRT and MFC. If you open the debug version of MyApplication.exe in Visual Studio and browse through its resources, you can see the RT_MANIFEST resource. This is the manifest embedded inside the binary. If you export it and open as an XML file, you see the following:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <dependency>
    <dependentAssembly>
      <assemblyIdentity type="win32" name="Microsoft.VC80.DebugCRT" version="8.0.50727.42"
processorArchitecture="x86" publicKeyToken="1fc8b3b9a1e18e3b"></assemblyIdentity>
    </dependentAssembly>
  </dependency>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity type="win32" name="Microsoft.VC80.DebugMFC" version="8.0.50727.42"
processorArchitecture="x86" publicKeyToken="1fc8b3b9a1e18e3b"></assemblyIdentity>
    </dependentAssembly>
  </dependency>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity type="win32" name="Microsoft.Windows.Common-Controls" version="6.0.
0.0" processorArchitecture="x86" publicKeyToken="6595b64144ccf1df" language="*"></assemblyI
dentity>
    </dependentAssembly>
  </dependency>
</assembly>
```

This means that this application depends on these assemblies:

- Assembly Microsoft.VC80.DebugCRT, version 8.0.50727.42 for x86
- Assembly Microsoft.VC80.DebugMFC, version 8.0.50727.42 for x86
- Assembly Microsoft.Windows.Common-Controls, version 6.0.0.0 for x86

In the release binary, you will see this:

```
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <dependency>
    <dependentAssembly>
      <assemblyIdentity type="win32" name="Microsoft.VC80.CRT" version="8.0.50727.42" proce
ssorArchitecture="x86" publicKeyToken="1fc8b3b9a1e18e3b"></assemblyIdentity>
    </dependentAssembly>
  </dependency>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity type="win32" name="Microsoft.VC80.MFC" version="8.0.50727.42" proce
ssorArchitecture="x86" publicKeyToken="1fc8b3b9a1e18e3b"></assemblyIdentity>
    </dependentAssembly>
  </dependency>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity type="win32" name="Microsoft.Windows.Common-Controls" version="6.0.
0.0" processorArchitecture="x86" publicKeyToken="6595b64144ccf1df" language="*"></assemblyI
dentity>
    </dependentAssembly>
  </dependency>
</assembly>
```

This means that this application depends on these assemblies:

- Assembly Microsoft.VC80.CRT, version 8.0.50727.42 for x86
- Assembly Microsoft.VC80.MFC, version 8.0.50727.42 for x86
- Assembly Microsoft.Windows.Common-Controls, version 6.0.0.0 for x86

You would see similar manifests in MyLibrary.dll, both for debug and release. Note that the manifest ID is 1 for an EXE, 2 for a DLL. Also, if the manifest is not embedded in the binary, it is stored as <binaryname>.<extension>.manifest and has the same content.

Note

Visual Studio 2005 does not support building C++ applications without a manifest and binding to Visual C++ libraries the old way using %PATH%. Moreover, Visual C++ DLLs can detect this, prevent the DLL from loading and report the unsupported scenario and the necessary changes. Do not use /manifest:no or delete the manifest.

Deployment methods

In this example, you will install MyApplication.exe to a folder %TARGET% that can be specified by the customer during installation. MyLibrary.dll will be installed in %TARGET%\MyLibrary, and \MyLibrary added to the path.

We will examine two methods to deploy VC++ applications:

1. Building a setup package using a Setup and Deployment Project.
2. XCopy Deployment.

For each method, we will explore two scenarios:

1. Deploying Visual C++ libraries as shared assemblies.
2. Deploying Visual C++ libraries as private assemblies.

In scenario 1, there is only one copy of the Visual C++ DLLs in the WinSxS folder. In scenario 2, you will have two copies of the Visual C++ DLLs, installed in the application EXE's and DLL's local folders.

Note

Scenario 2 is not supported on Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows 2000 because deployment in application-local folders creates problems when it comes to servicing.

See Also

Tasks

How to: Deploy using a Setup and Deployment Project

How to: Deploy using XCopy

Concepts

Deployment Examples

How to: Deploy using a Setup and Deployment Project

This topic demonstrates deployment using a Setup and Deployment Project. For background on this deployment scenario, see Scenarios for Deployment Examples.

Procedures for deploying Visual C++ library DLLs as shared assemblies

Create a Setup and Deployment Project

- From the **File** menu, click **New Project**, open the **Other Project Types** node and select **Setup and Deployment**, then click **Setup Project**. Supply a name and click **OK**.

Note
Visual C++ Express does not support Setup Projects. To redistribute Visual C++ libraries with applications built in Visual C++ Express, use the Visual C++ Redistributable Package. Please see How to: Deploy using XCopy for more information.

Add the EXE and DLL to the project and specify where they should be installed on the target computer

1. From the **Project** menu, choose **Add** and click **File**.
2. Find the folder containing MyApplication.exe and MyLibrary.DLL and select them both.
3. In the **File System** window, right click on **Application Folder**, point to **Add** and click **Create** to create a new folder. Call it MyLibrary.
4. Click on **Application Folder** again, select MyLibrary.DLL and drag it to the MyLibrary folder.In **Solution Explorer**, under your project in **Detect Dependencies** you should see that the Visual Studio detects dependencies on MFC80.dll and MSVCR80.dll. You need to add the corresponding Merge Modules for these DLLs.
5. From the **Project** menu, point to **Add** and click **Merge Module**. Select Microsoft_VC80_CRT_x86.msm and Microsoft_VC80_MFC_x86.msm, and click **OK**.

Note
The debug versions of these merge modules are named Microsoft_VC80_DebugCRT_x86.msm and Microsoft_VC80_DebugMFC_x86.msm.

For deploying 64-bit applications to a 64-bit operating system, select the merge module for the corresponding platform. For x64, select Microsoft_VC80_CRT_x86_x64.msm and Microsoft_VC80_MFC_x86_x64.msm; for Itanium, Microsoft_VC80_CRT_x86_ia64.msm and Microsoft_VC80_MFC_x86_ia64.msm.

Build setup.exe

- Click **Build Solution** from the **Build** menu.

Run setup.exe

- Your application is installed in the specified target folder.
- Merge modules install the CRT and MFC assemblies as shared side-by-side assemblies. You can verify this by checking folders %WINDIR%\WinSxS\x86_Microsoft.VC80.CRT_1fc8b3b9a1e18e3b_8.0.50613.12800_x-ww_b7acac55\ and %WINDIR%\WinSxS\x86_Microsoft.VC80.MFC_1fc8b3b9a1e18e3b_8.0.50613.12800_x-ww_88931f5a\
- The application and DLL use the CRT and MFC assemblies at runtime. You must add %TARGET%\MyLibrary to the PATH environment variable so that MyApplication.exe can find Mylibrary.Dll. (Or you could rebuild it as a side-by-side assembly.)

On an operating system that does not support manifest-based binding (Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows 2000), this is what happens:

- Your application is installed in the specified target folder.
- Merge modules install the CRT and MFC assemblies in two locations:
  - %WINDIR%\System32—This ensures that applications run on this operating system even though it does not support manifest-based binding.
  - %WINDIR%\WinSxS (similar to Windows XP Home Edition, Windows XP Professional, Windows Server 2003)—This ensures that when the operating system is upgraded to Windows XP Home Edition, Windows XP Professional, Windows Server 2003, your application will still run and start using the DLLs in WinSxS. The DLLs in System32 will be removed in the process by which Windows Upgrade informs already-installed applications about operating system upgrades.
- The application and DLL use the CRT and MFC DLLs from System32 at runtime.

Procedures for deploying Visual C++ library DLLs as private assemblies

Create a Setup and Deployment Project

- From the **File** menu, click **New Project**, open the **Other Project Types** node and select **Setup and Deployment**, then click **Setup Project**. Specify a name and click **OK**.

Add EXE and DLL to the project and specify where they should be installed on the target computer

1. From the **Project** menu, choose **Add** and click **File**.
2. Find the folder containing MyApplication.exe and MyLibrary.DLL and select them both.
3. In the **File System** window, right click **Application Folder**, point to **Add** and click **Create** to create a new folder. Call it MyLibrary.
4. Click **Application Folder** again, select MyLibrary.DLL and drag it to the MyLibrary folder.In **Solution Explorer**, under your project in **Detect Dependencies** you should see that Visual Studio detects dependencies on MFC80.dll and MSVCR80.dll. You need to add a corresponding folder for these assemblies from the \vc\redist folder.

Note
For debug builds, use \vc\redist\debug_nonredist.

5. Open %PROGDIR%\Microsoft Visual Studio 8\VC\Redist\x86 in Windows Explorer.
6. Holding the Ctrl key, click the folders Microsoft.VC80.CRT and Microsoft.VC80.MFC. Drag these folders to Visual Studio and drop them into the Application folder.
7. Repeat step 7, but this time drag the folders to the MyLibrary folder.

   You may see a message from Visual Studio saying that you are including a DLL that is part of a Merge Module. This is exactly what you want to do, so click **No** to indicate that you do not want to use an MSM for this DLL.

   You need mfcm80.dll and its Unicode version mfcm80u.dll only if you use MFC/Winforms integration. Otherwise you can remove these DLLs from your setup.

   You need msvcm80.dll only if you are using managed code in your applications, for example, if your projects are built with /clr or /clr:pure.

   If mfcm80.dll, mfcm80u.dll, or msvcm80.dll is included in the setup project, an installation of the .NET Framework is required. Your setup won't work without .NET Framework 2.0 installed on the target machine.

   For debug installation, change CRT to DebugCRT and MFC to DebugMFC in the previous step.

   For deploying 64-bit applications to 64-bit operating systems, use \vc\redist\amd64 or \vc\redist\ia64.

Build setup.exe

- Click **Build Solution** from the **Build** menu.

Run setup.exe

- Your application is installed in the specified target folder.
- The application and DLL use the CRT and MFC assemblies from the Microsoft.VC80.CRT and Microsoft.VC80.MFC folders at runtime.

On an operating system that does not support manifest-based binding (Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows 2000), this deployment method is not recommended. You are strongly urged to use MSMs instead. If you attempt to install, this is what happens:

- Your application is installed in the specified target folder.
- By default, the operating system loader won't find the CRT and MFC DLLs. Microsoft.VC80.CRT and Microsoft.VC80.MFC have to be added to the PATH.

Note
MSMs allow you to do custom steps depending on the target machine's operating system. You may want to add these directories to the PATH only on operating systems that don't support manifest-based binding; that is, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition and Windows 2000, but not Windows XP Home Edition, Windows XP Professional, Windows Server 2003.

See Also

Tasks

How to: Deploy using XCopy

Concepts

Deployment Examples

How to: Deploy using XCopy

This topic demonstrates how to deploy applications by copying application files from one computer to another. This technique is useful for debugging applications on another computer. For background on this deployment scenario, see Scenarios for Deployment Examples.

Deploying Visual C++ library DLLs as shared assemblies

Visual C++ Redistributable Package can be used to deploy release versions of all Visual C++ libraries. These are three versions of this package—x86, x64, and IA64–that can be used on the corresponding versions of Windows. This package only installs release versions of Visual C++ libraries. Debug versions can either be deployed as private assemblies or by using the debug versions of Visual C++ Merge Modules. Debug versions of merge modules can be packaged together with the debug version of the application using Setup and Deployment project as described in How to: Deploy using a Setup and Deployment Project. Alternatively it is possible to install a merge module by using Windows Installer (msiexec.exe). For more information please see How to: Prepare a Test Machine to Run a Debug Executable.

To deploy Visual C++ libraries by using Visual C++ Redistributable Package, perform the following steps:

Steps

1. Create a folder structure on the development computer that matches the folder structure to be used on the target computer. For this example, create a \bin folder and copy myapplication.exe there. Then create a \bin\mylibrary folder and copy MyLibrary.dll there.
2. Copy this folder to the target computer.
3. Copy Visual C++ Redistributable Package (vcredist_x86.exe) from %PROGDIR%\Microsoft Visual Studio 8\SDK\v2.0\Bootstrapper\Packages\vcredist_x86\vcredist_x86.exe to the target computer

   Note

   For applications built in Visual C++ Express, it is recommended to redistribute Visual C++ libraries using the Visual C++ Redistributable Package (vcredist_x86.exe). Visual C++ Express does not install this package; however, it can be downloaded from Microsoft Visual C++ 2005 Redistributable Package (x86).

4. Run vcredist_x86.exe on the target computer.This installs all Visual C++ libraries as shared assemblies. On a target computer with support for manifest-based binding of applications to their dependencies (Windows XP Home Edition, Windows XP Professional, Windows Server 2003, Windows Vista), the libraries are installed in the WinSxS folder. On a computer without such support (Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows 2000), the libraries are installed to both the WinSxS and System32 folders.

   Note

   For deploying 64-bit applications to 64-bit operating systems, use vcredist_x64.exe or vcredist_ia64.exe. These packages can also be downloaded from Microsoft Visual C++ 2005 Redistributable Package (x64) and Microsoft Visual C++ 2005 Redistributable Package (ia64)

   Note

   vcredist_*.exe installs only the release versions of Visual C++ DLLs. Debug versions should only be deployed as private assemblies; see the next procedure for more details.

5. Your application is ready to be run.

Deploying Visual C++ library DLLs as private assemblies

In this scenario, the application is deployed by just copying a folder from the development computer to the target computer, and installing dependent assemblies as private side-by-side assemblies. To deploy Visual C++ libraries as private assemblies, perform the following steps:

Steps

1. Create a folder structure on the development computer that matches the folder structure to be used on the target computer. For this example, create a \bin folder and copy myapplication.exe there. Then create a \bin\mylibrary folder and copy MyLibrary.dll there.
2. On the development computer, from %PROGDIR%\Microsoft Visual Studio 8\VC\Redist\x86, copy Microsoft.VC80.CRT and Microsoft.VC80.MFC to \bin and to \bin\MyLibrary.

Note
Visual C++ Express does not create %PROGDIR%\Microsoft Visual Studio 8\VC\Redist\ during installation on the development computer. To redistribute Visual C++ libraries with applications built with Visual C++ Express, please use Visual C++ Redistributable Packages (VCRedist_*.exe). See section "Deploying Visual C++ library DLLs as shared assemblies" above for more information.

Note
For deploying 64-bit applications to 64-bit operating systems, use \vc\redist\amd64 or \vc\redist\ia64.

3. Copy the \bin folder to the target computer. On a target computer with manifest-based binding support (Windows XP Home Edition, Windows XP Professional, Windows Server 2003, Windows Vista) no further preparation is necessary. On a computer without such support (Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows 2000), Microsoft.VC80.CRT and Microsoft.VC80.MFC must be on the path.

Note
For debug applications, use debug DLLs from \vc\redist\debug_nonredist\. However, remote debugging may not work for an application that deploys debug versions of Visual C++ libraries as private assemblies. For more information, see Preparing a Test Machine To Run a Debug Executable.

Note
If you want to do remote debugging on the target machine, you must copy over the debug DLLs or follow the procedure to install debug DLLs in Preparing a Test Machine To Run a Debug Executable.

See Also

Tasks

How to: Deploy using a Setup and Deployment Project

Concepts

Deployment Examples

Preparing a Test Machine To Run a Debug Executable

To prepare a computer to test the debug version of an application built with Visual C++ you need to deploy debug versions of Visual C++ libraries that the application depends on. To identify what libraries have to be deployed, follow the steps outlined in Understanding Dependencies of a Visual C++ Application. Debug versions of Visual C++ library DLLs usually have names that end in "d"; for example, the debug version of the CRT DLL msvcr80.dll is named msvcr80d.dll.

Note

Debug versions of an application are not redistributable and none of the debug versions of the various Visual C++ dynamic-link libraries (DLLs) are redistributable. Debug versions of an application and Visual C++ libraries can only be deployed to another computer internal to your development site for the sole purpose of debugging and testing your application on a computer that does not have Visual C++ 2005 installed. For more information please see, Redistributing Visual C++ Files.

There are two ways of deploying debug versions of Visual C++ libraries along with the debug version of an application:

1. Use Merge Modules to install a debug version of a particular Visual C++ library as shared side-by-side assemblies into the native assembly cache (WinSxS folder). An example of this deployment may be found in How to: Deploy using a Setup and Deployment Project.

   Note

   Visual C++ Express does not support Setup Projects. An alternative way of installing debug merge modules is to use Windows Installer (msiexec.exe). For example, to install debug version of the CRT library, one may execute **msiexec /i microsoft_vc80_debugcrt_x86.msm** on the target machine.

2. Install a particular Visual C++ assembly as a private assembly for the application using files provide in the Program Files\Microsoft Visual Studio 8\VC\Redist\debug_nonredist\ directory. An example may be found in How to: Deploy using XCopy.

   Note

   For remote debugging your application on another computer, you have to deploy debug versions of Visual C++ libraries as shared side-by-side assemblies. You may either use Setup projects or Windows Installer to install corresponding merge modules using the steps described above. After Visual C++ libraries are installed you may run a remote debugger off a network share. For more information on remote debugging, please see How to: Set Up Remote Debugging.

See Also

Tasks

How to: Set Up Remote Debugging
How to: Deploy using a Setup and Deployment Project
How to: Deploy using XCopy

Concepts

Deployment Examples

Other Resources

Windows Installer Command line options

Deploying Windows-Client Applications

With Visual Studio 2005, Visual C++ applications can be deployed in two ways: publishing an application using ClickOnce technologies, or deploying it using the traditional Setup application using Windows Installer technologies. With Windows Installer deployment, you package the application into a setup.exe file and distribute that file to users; they run the setup.exe file to install the application. With ClickOnce deployment, you publish the application to a centralized location and the user installs or runs the application from that location. For a comparison of these two deployment strategies see Choosing a Deployment Strategy.

For information on ClickOnce Deployment see ClickOnce Deployment.

For deployment using Windows Installer, you need to package all of the parts of your application and any other dependent libraries and components into an installation package that properly installs and registers all files on the target computer. For more information see Windows Installer Deployment

For more information on how to determine what dependent components you have to package with your application see Determining Which DLLs to Redistribute.

Visual Studio allows you to build an installation utility for your application using Setup and Deployment Projects. Using a wizard, you may specify how your application is deployed to another computer, and in what ways its dependent DLLs and assemblies are to be installed on the target computer.

If you choose to redistribute Visual C++ Libraries as Shared Side-by-Side Assemblies using merge modules provided with Visual Studio 2005, you need to include merge modules for the Visual C++ assemblies your application depends on in your Setup and Deployment Project. In this way, these assemblies will be installed on the target computer as shared side-by-side assemblies and can be used by your application at runtime. For more information see Redistributing Using Merge Modules and How to: Add Merge Modules to a Deployment Project.

If you decide that you need to deploy a Visual C++ assembly as a private assembly for your application, you need to include the DLLs and manifest files that are part of the assembly inside your deployment project. For more information on which DLLs and manifest files form each Visual C++ assembly see Visual C++ Libraries as Shared Side-by-Side Assemblies. You need to ensure that after Visual C++ DLLs and corresponding manifest files are deployed to the target computer, they are located in the application-local folder in such a way that they can be found and loaded at runtime. For rules on how private assemblies have to be located relative to the application that uses them, see Assembly Searching Sequence.

Be sure to include the DLLs and manifest files for a specific assembly. Deploying just the DLLs on the target computer does not enable loading these DLLs. The manifest file for an assembly is used to uniquely identify the assembly and DLLs that are part of the assembly. It is used at runtime to search for assemblies and if missing DLLs cannot be identified as part of assembly, they are not loaded.

Related Sections

Setup and Deployment Projects

Windows Installer Deployment Tasks

Isolated application and side-by-side assemblies

Redistributing Visual C++ Files

See Also

Concepts

Deployment Examples

Other Resources

Deployment (C++)

Redistributing Web Applications

If your application uses the MFC classes implementing the WebBrowser control (for example, **CHtmlView** or **CHtmlEditView**), Microsoft Internet Explorer 4.0 or later must at least be minimally installed on the target computer.

Installing the latest version of Internet Explorer also ensures that the target computer has the latest common control files.

Information about installing minimal Internet Explorer components is available in the following Knowledge Base article:

- Q185375, HOWTO: Create a Single EXE Install of Internet Explorer (http://support.microsoft.com/support/kb/articles/q185/3/75.asp)

You can find Knowledge Base articles in the MSDN Library or at http://support.microsoft.com.

See Also

Tasks

Deploying ATL Server Applications

Other Resources

Deployment (C++)

Deploying ATL Server Applications

Note
Deployment of ATL Server projects to IIS is not supported on a Web server running on any 64-bit platform.

Web applications are typically deployed by copying files to the server using xcopy or ftp. However, Visual Studio uses VCDeploy.exe to deploy Visual C++ (ATL Server or ISAPI) Web applications. VCDeploy.exe is built into the development environment, so you typically do not have to access it directly. When you create the application and specify the publishing location (for example, C:\Inetpub\wwwroot) in the Application Wizard, then VCDeploy.exe will deploy your application to that location. The actual publishing of files occurs at build time.

VCDeploy.exe will only deploy applications to localhost (the computer used to build the Web application), which is useful for testing. If your development computer (localhost) is not your server, you will have to copy the server files manually and register them with IIS.

Note
When you deploy to a 64-bit platform, you must ensure that Visual Studio spawns the 64-bit version of VCDeploy.exe; otherwise, virtual directories in IIS will not be created, and registration of your ISAPI extensions will fail. You can ensure that the 64-bit version of VCDeploy.exe is spawned by setting your options as follows: From the **Tools** menu, click **Options**; open the **Projects and Solutions** node and select **VC++ Directories**; in **Show directories for**, select **Executable files**; in **Platform**, select **x64** or **Itanium**. Then enter a new directory **$(VCInstallDir)bin\amd64** and make it first in the list. Alternatively, you can open a command prompt window and put this directory first in your path, then launch devenv.exe from the same command prompt.

When deploying ATL Server applications, you normally will not have any system dependencies other than the C run-time libraries (Msvcr71.dll) and your own application DLLs. However, because this depends on your project configuration, you need to verify before deployment that you have identified all components that your application depends on. For information about locating dependencies, see Determining Which DLLs to Redistribute.

XML Web services created using ATL Server require at a minimum that MSXML 3.0 be installed on the target computer. You can download the latest XML Parser from the MSDN Online XML/XSL Website (http://msdn.microsoft.com/xml).

Be aware that the definition of `WIN32_WINNT` generated by the application wizard in StdAfx.h causes your projects to depend on a minimum level of operating system support.

To deploy an ATL Server application manually

1. Build your application and locate all of the components that you need to distribute. An ATL Server Web application or XML Web service could include any of the following items:
   - ISAPI extension DLL
   - Web application DLLs
   - Dependencies (MSVCR71, MSXML, or other components specific to your project)
   - Server response files
   - Discovery documents
   - HTML pages, style sheets, images, and other content
2. Copy the files and dependencies to an appropriate location on the target computer.

   You can use multiple folders to store your files so that you can configure different permissions for each folder. It can be worthwhile to store static content (HTML pages, style sheets, images), server response files, and DLLs in three or more different folders.

   Because some DLLs (such as XML Web service DLLs) are accessed directly by the user, while others (such as Web application DLLs that are only used by server response files) are not be accessed in this way, you might find site management simpler if each of these categories has its own folder.

Whatever folder structure you choose, you need to be sure that the relative and absolute paths used in your files are correct. In particular, you need to be sure that the handler tags in your server response files point to the correct locations.

3. Set NTFS file security settings appropriately. For information about the security contexts used in a typical ATL Server application or XML Web service, see ATL Server Security.
4. Start Internet Services Manager (for example, on the **Start** menu, click **Programs**, click **Administrative Tools**, and then click **Internet Services Manager**).
5. Choose a Web site on which to install your application. Typically, this will be the **Default Web Site**.
6. Right-click the site, click **New**, and then click **Virtual Directory**.
7. In the **Alias** box, enter a name for the virtual directory.
8. Select the folder used in step 2 as the **Web Site Content Directory** for this new virtual directory.
9. Give the new directory **Run scripts**, and then **Execute** permissions.

   This assumes the directory only contains server response files and XML Web service DLLs. In general, follow these guidelines for giving permissions to your directories:

   - Give **Read** permissions if the directory contains static content (such as HTML pages, style sheets, images, and other content).
   - Give permissions to **Run scripts** if the directory contains server response files and the ISAPI extension is registered as a script engine. It is not necessary or desirable to give **Read** permissions.
   - Give **Execute** permissions if the directory contains server response files and the ISAPI extension is not registered as a script engine or the directory contains XML Web service or Web application DLLs that will be accessed directly by the user. It is not necessary or desirable to give **Read** permissions.
   - If the directory contains DLLs and other files that are never to be accessed directly by the user of the Web site, disable **Read**, **Run scripts**, and **Execute** permissions. These DLLs can still be called from server response files or used from code, but they cannot be called or downloaded directly by users of the Web site.
10. Click **Next**, and then click **Finish** to end the wizard and create the virtual directory.
11. Right-click the newly created virtual directory, and then select **Properties**.
12. Click the **Configuration** button on the **Virtual Directory** page of the **Properties** dialog box.
13. Use the **App Mappings** page of the **Application Configuration** dialog box to register your ISAPI extension DLL as the handler for DLLs and server response files as described in Associating SRF Files with Your ISAPI DLL by Hand.

    You do not need to register the ISAPI extension as the handler for a file extension if you never use it. If you only allow users to access your site using server response files, you do not need to register the ISAPI extension as the handler for DLLs. If you only provide XML Web services that are accessible directly as DLLs, you do not need to register the ISAPI extension as the handler for server response files.
14. Optional. Set **Application Protection** to an appropriate level.
15. Optional. Set the appropriate default documents on the **Documents** page.
16. Optional. On the **Directory Security** page, set the appropriate security for your application.
17. Optional. Configure other settings using Internet Services Manager.

To deploy an ATL Server application using a setup project

- It is recommended that you use a Web Setup project to create an .msi file that automates the process of setting up an ATL Server application. For more information, see Deployment Projects.

To deploy an ATL Server application to the local computer during development

- You can use the Web Deployment Property Page to automate the process of setting up an ATL Server application when developing on a computer that has IIS installed.

See Also

Other Resources

Deployment (C++)

Version Checking of Redistributable Files

It is important to install only the newest version of a file. In almost all cases, you want to install a newer version of a DLL or component on top of an older version, but not the other way around. System DLLs (for example, Kernel32.dll, User32.dll, Ole32.dll, and ShDocVW.dll) are not to be redistributed. If you have to redistribute any files, make sure to read the corresponding license agreements and ensure that the files can be redistributed for corresponding operating system versions and location of the installation. Typically, version checking is the responsibility of the Setup program.

If you write your own custom Setup program, it must manually check version information when installing redistributable files. This topic includes a sample program that checks for version numbers or, if version numbers are missing from a file, timestamps.

Example

The following example demonstrates a method of checking between two dynamic libraries or executables.

```
// checkversion.cpp
// compile with: /link version.lib

#include <windows.h>
#include <stdio.h>

void EmitErrorMsg (HRESULT hr);
HRESULT GetFileVersion (char *filename, VS_FIXEDFILEINFO *vsf);
HRESULT GetFileDate (char *filename, FILETIME *pft);
HRESULT LastError();
int WhichIsNewer (char *fname1, char *fname2);

void printtime (FILETIME *t)
{
    FILETIME lft;
    FILETIME *ft = &lft;
    FileTimeToLocalFileTime(t,ft);
    printf_s("%08x %08x",ft->dwHighDateTime,ft->dwLowDateTime);
    {
        SYSTEMTIME stCreate;
        BOOL bret = FileTimeToSystemTime(ft,&stCreate);
        printf_s("    %02d/%02d/%d  %02d:%02d:%02d\n",
                        stCreate.wMonth, stCreate.wDay, stCreate.wYear,
                        stCreate.wHour, stCreate.wMinute,
                        stCreate.wSecond);
    }
}

int main(int argc, char* argv[])
{
    printf_s("usage: checkversion file1 file2\n"
             "\tReports which file is newer, first by checking "
             "the file version in "
             "\tthe version resource, then by checking the date\n\n" );

    if (argc != 3)
        return 1;

    int newer = WhichIsNewer(argv[1],argv[2]);
    switch(newer)
    {
        case 1:
        case 2:
            printf_s("%s is newer\n",argv[newer]);
        break;
        case 3:
            printf_s("they are the same version\n");
        break;
        case 0:
```

```
        default:
            printf_s("there was an error\n");
        break;
    }

    return !newer;
}

int WhichIsNewer (char *fname1, char *fname2)
{
    // 1 if argv[1] is newer
    // 2 if argv[2] is newer
    // 3 if they are the same version
    // 0 if there is an error

    int ndxNewerFile;
    HRESULT ret;
    VS_FIXEDFILEINFO vsf1,vsf2;

    if ( SUCCEEDED((ret=GetFileVersion(fname1,&vsf1))) &&
         SUCCEEDED((ret=GetFileVersion(fname2,&vsf2))))
    {
        // both files have a file version resource
        // compare by file version resource
        if (vsf1.dwFileVersionMS > vsf2.dwFileVersionMS)
        {
            ndxNewerFile = 1;
        }
        else
            if (vsf1.dwFileVersionMS < vsf2.dwFileVersionMS)
            {
                ndxNewerFile = 2;
            }
            else {
              // if (vsf1.dwFileVersionMS == vsf2.dwFileVersionMS)
              if (vsf1.dwFileVersionLS > vsf2.dwFileVersionLS) {
                   ndxNewerFile = 1;
              }
              else
                  if (vsf1.dwFileVersionLS < vsf2.dwFileVersionLS) {
                      ndxNewerFile = 2;
              }
            else
            {
                // if (vsf1.dwFileVersionLS == vsf2.dwFileVersionLS)
                ndxNewerFile = 3;
            }
        }
    }
    else
    {
        // compare by date
        FILETIME ft1,ft2;
        if (SUCCEEDED((ret=GetFileDate(fname1,&ft1))) &&
            SUCCEEDED((ret=GetFileDate(fname2,&ft2))))
        {
            LONG x = CompareFileTime(&ft1,&ft2);
            if (x == -1)
                ndxNewerFile = 2;
            else
                if (x == 0)
                   ndxNewerFile = 3;
                else
                   if (x == 1)
                       ndxNewerFile = 1;
                   else
                   {
                       EmitErrorMsg(E_FAIL);
```

```
                        return 0;
                    }
        }
        else
        {
            EmitErrorMsg(ret);
            return 0;
        }
    }
    return ndxNewerFile;
}

HRESULT GetFileDate (char *filename, FILETIME *pft)
{
    // we are interested only in the create time
    // this is the equiv of "modified time" in the
    // Windows Explorer properties dialog
    FILETIME ct,lat;
    HANDLE hFile = CreateFile(filename, GENERIC_READ,FILE_SHARE_READ |
                        FILE_SHARE_WRITE,0,OPEN_EXISTING,0,0);
    if (hFile == INVALID_HANDLE_VALUE)
        return LastError();
    BOOL bret = GetFileTime(hFile,&ct,&lat,pft);
    if (bret == 0)
        return LastError();
    return S_OK;
}

// This function gets the file version info structure
HRESULT GetFileVersion (char *filename, VS_FIXEDFILEINFO *pvsf)
{
    DWORD dwHandle;
    DWORD cchver = GetFileVersionInfoSize(filename,&dwHandle);
    if (cchver == 0)
        return LastError();
    char* pver = new char[cchver];
    BOOL bret = GetFileVersionInfo(filename,dwHandle,cchver,pver);
    if (!bret)
        return LastError();
    UINT uLen;
    void *pbuf;
    bret = VerQueryValue(pver,"\\",&pbuf,&uLen);
    if (!bret)
        return LastError();
    memcpy(pvsf,pbuf,sizeof(VS_FIXEDFILEINFO));
    delete[] pver;
    return S_OK;
}

HRESULT LastError ()
{
    HRESULT hr = HRESULT_FROM_WIN32(GetLastError());
    if (SUCCEEDED(hr))
        return E_FAIL;
    return hr;
}

// This little function emits an error message
// based on WIN32 error messages
void EmitErrorMsg (HRESULT hr)
{
    char szMsg[1024];
    FormatMessage(
        FORMAT_MESSAGE_FROM_SYSTEM,
        NULL,
        hr,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        szMsg,
```

```
        1024,
        NULL
    );

    printf_s("%s\n",szMsg);
}
```

See Also

Other Resources

Deployment (C++)

ClickOnce Deployment for Visual C++ Applications

Visual Studio provides two different technologies for deploying Windows applications: ClickOnce deployment or Windows Installer deployment. For a detailed comparison of ClickOnce and Windows Installer deployment, see Choosing Between ClickOnce and Windows Installer

ClickOnce Deployment in C++

The Visual C++ development environment does not directly support deployment of Visual C++ projects with ClickOnce, but tools are available to use it.

Note
Visual Studio does support ClickOnce in the Visual C# and Visual Basic development environments. If your Visual C++ project is a dependency of a Visual C# project, you can publish the application (including its dependencies) using ClickOnce deployment from the Visual C# development environment.

To deploy a Visual C++ application using ClickOnce, you first have to build a ClickOnce Application Manifest and a ClickOnce Deployment Manifest using the Manifest Generation and Editing Tool (Mage.exe) or its graphical user interface version (for information, see Manifest Generation and Editing Tool, Graphical Client (MageUI.exe)).

You first use Mage.exe to build the application manifest; the resulting file will have the extension .manifest. You then use Mage.exe to build the deployment manifest; the resulting file will have the extension .application. You then sign the manifests.

The application manifest must specify the target processor (**x86**, **x64**, or **Itanium**). See Deploying 64-bit Applications for information on these options.

Also, the name of the application and deployment manifests must be different from the name of the C++ application. This avoids conflict between the application manifest created by Mage.exe and the external manifest that is part of the C++ application.

Your deployment will need to install any Visual C++ libraries on which your application depends. To determine the dependencies for a particular application, you can use depends.exe or the DUMPBIN utility with the /DEPENDENTS option. For more information on dependencies, see Understanding Dependencies of a Visual C++ Application. You might need to run VCRedist.exe; this utility installs Visual C++ libraries on the target computer.

You may also need to build a bootstrapper (prerequisites installer) for your application to deploy prerequisite components; for information on the bootstrapper, see Adding Custom Prerequisites.

For a more detailed description of the technology, see ClickOnce Deployment. For a detailed example of ClickOnce deployment, see Walkthrough: Deploying a ClickOnce Application Manually.

See Also

Tasks

Deploying ATL Server Applications

Reference

Manifest Generation and Editing Tool (Mage.exe)
Manifest Generation and Editing Tool, Graphical Client (MageUI.exe)
Certificate Creation Tool (Makecert.exe)

Concepts

Adding Custom Prerequisites

Other Resources

Deployment (C++)
Deploying Applications and Components
Windows Installer Deployment
ClickOnce Deployment
.NET Programming in C++
Native and .NET Interoperability

Running a C++ /clr Application on a Previous Runtime Version

Unless otherwise specified, a Visual C++ .NET application is built to run with the common language runtime version that the compiler uses to build the application. However, it is possible for an .exe application built with one version of the runtime to run on any other version of the runtime that provides all of the necessary functionality.

To accomplish this, an .exe application needs an app.config file containing runtime version information (with the **supportedRuntime** tag). The Visual C++ development environment does not support modification of the app.config file, but other Visual Studio languages do through the project's **Property Pages** dialog box. For example, you can modify the **Supported Runtimes** property of a Visual C# Windows application and use that updated app.config file in your C++ application.

At run time, the name of the app.config file must be *filename.ext*.config (where *filename.ext* is the name of the executable that started the application) and the file must be in the same directory as the executable. For example, if your application is called TestApp.exe, the app.config file would be named TestApp.exe.config.

If you specify more than one runtime version and the application runs on a computer with more than one installed runtime version, the application uses the first version specified in the config file that matches an installed runtime that is available on the system.

For more information, see Targeting a .NET Framework Version.

Windows applications built with the Visual C++ compiler need to be compiled with /clr:initialAppDomain to run on the previous version of the common language runtime.

Targeting the previous version of the runtime for ASP.NET Web applications is not supported in this release of Visual C++.

See Also

Other Resources

Deployment (C++)

Redeploying an Application Containing a Visual C++ Service Pack or Hotfix

This topic is new for Visual Studio 2005 SP1.

Visual C++ 2005 SP1, similar to other service packs and hotfixes, installs a new version of the Visual C++ libraries. These new libraries are backwards binary-compatible with the libraries in Visual C++ 2005. Changes in the Visual C++ libraries for Visual C++ 2005 SP1 are expected to have no effect on symbols exported from runtime DLLs associated with the libraries. Visual C++ applications that are built with the Visual C++ 2005 version of the libraries might behave differently under Visual C++ 2005 SP1 because of updates to the Visual C++ libraries. However, applications will be able to load libraries at runtime and resolve calls to methods in the Visual C++ libraries regardless of which version of the libraries is installed. The libraries shipped with Visual C++ 2005 SP1 are a complete replacement of the libraries shipped in Visual C++ 2005, and you should use both for application development and for runtime execution.

We recommended that you update all computers that you use in application development to Visual C++ 2005 SP1. Once an application is rebuilt, it will depend on the Visual C++ 2005 SP1 libraries. If you install such an application on end-users computers, you will have to redistribute the libraries included with Visual C++ 2005 SP1. This topic describes how to do this.

After you apply the service pack or hotfix and rebuild your application, verify that the application is working as expected. Only when you are satisfied with the results of your testing, deploy the update to your application as follows:

- If you originally deployed your application with a Visual C++ merge module, you should deploy your update with the new Visual C++ merge module that was included with the service pack or hotfix.
- If you originally deployed your application with the Visual C++ Redistributable Package, you should deploy your update with the new Visual C++ Redistributable Package that was included with the service pack or hotfix.
- If you originally deployed your application with an application local copy of a Visual C++ DLL, you should deploy your update with an application local copy of the Visual C++ DLL that was included with the service pack or hotfix.

Visual C++ 2005 Service Pack 1

Once an application is rebuilt with Visual C++ 2005 Service Pack 1, it will need the Visual C++ 2005 Service Pack 1 version of the Visual C++ libraries during runtime. Follow the instructions above in the previous section to deploy an update to your application.

See Also

Other Resources

Deployment (C++)

Porting and Upgrading Programs

Porting an application refers to opening it for the first time in Visual C++. Upgrading refers to opening a project from a previous version of Visual C++ in the current version.

Note
An ActiveX control compiled with Visual C++ 6.0 when embedded in a dialog box in a project developed with Visual C++ 2005 may cause your program to assert at runtime. In this situation, recompile the ActiveX control using Visual C++ 2005. The assert will be in the file occcont.cpp, on the line in source: `ASSERT(IsWindow(pTemp->m_hWnd)).`

For more information, see What's New in Visual C++.

In This Section

Port from UNIX to Win32

Discusses options for migrating UNIX applications to Windows.

How to: Upgrade Wizard-Generated Managed Extensions for C++ Code

Discusses how you can upgrade wizard-generated code.

Managed Extensions for C++ Syntax Upgrade Checklist

Lists the syntactic differences between Managed Extensions for C++ and the new Visual C++ syntax for targeting the common language runtime.

How to: Upgrade Projects from Previous Versions of Visual C++

Discusses how to use projects created in previous versions of Visual C++.

C++/CLI Migration Primer

A detailed guide to upgrade your Managed Extensions for C++ syntax to use the new syntax (see New Language Features for more information on the new syntax).

See Also

Other Resources

Visual C++

Creating and Porting Visual C++ Device Projects

Porting from UNIX to Win32

When migrating applications from UNIX to Windows, you have several options:

- Using UNIX libraries to port applications from UNIX to Win32
- Porting applications from UNIX to Win32 natively
- Running UNIX applications on Windows NT and Windows 2000 using the POSIX subsystem

UNIX libraries

One option UNIX programmers normally consider is using third-party UNIX-like libraries to let their UNIX code compile as a Win32 executable. Several commercial (and at least one public domain) libraries do this. This is an option for some applications. The advantage of these porting libraries is that they minimize the initial porting effort. The main disadvantage, for a competitive software product, is that a native Win32 port of an application will generally be faster and will inevitably have more functionality. It can be awkward for the application to step outside of its UNIX shell if it needs to make Win32 calls to get more power from Windows NT and Windows 2000.

The following list provides Microsoft and third-party resources for porting and supporting UNIX migration to Visual C++:

UNIX Migration Guides

The UNIX Code Migration Guide provides technical help on code migration from UNIX to the Win32 environment.

UNIX Code Migration Guide

The Unix Project Migration Guide supplements the UNIX Code Migration Guide by providing high-level help on migrating substantial projects from UNIX to Win32. The Guide provides advice on issues to consider at each stage of project migration. The Guide may be downloaded from:

http://go.microsoft.com/fwlink/?linkid=20012

Microsoft Windows Services For UNIX (SFU)

Microsoft Windows Services For UNIX (SFU) version 3.0 provides a full range of cross-platform services for integrating Windows into existing UNIX-based environments. Services for UNIX provides file sharing, remote access and administration, password synchronization, common directory management, a common set of utilities, and a shell.

http://www.microsoft.com/windows/sfu/default.asp

InteropSystems.com

http://www.interopsystems.com/

A third party site for a company providing software supporting porting UNIX to Win32.

C++ Boost Web Site

http://boost.sourceforge.net/regression-logs/

http://boost.sourceforge.net/boost-build2/

Porting UNIX applications directly to Win32

Another option is porting UNIX applications directly to Win32. Using ANSI C/C++ libraries, and commercial C compiler libraries, many of the traditional system calls relied on by UNIX applications are available in Win32 applications.

The output model of **stdio**-based applications does not need to be changed, since the Win32 console APIs mimic the **stdio** model, and versions of *curses* exist that use the Win32 console APIs. For more information, see SetConsoleCursorPosition.

Berkeley socket-based applications need very few changes to work as Win32 applications. The Windows Sockets interface was designed for portability with BSD sockets, with minimal changes that are noted in the introductory sections of the WinSock specification.

Windows NT and Windows 2000 supports DCE-compliant RPC, so RPC-based applications are easily usable. See RPC Functions.

One of the largest areas of difference is in the process model. UNIX has **fork**; Win32 does not. Depending on the use of **fork** and the code base, Win32 has two APIs that can be used: **CreateProcess** and **CreateThread**. A UNIX application that forks multiple copies of itself can be reworked in Win32 to have either multiple processes or a single process with multiple threads.

If multiple processes are used, there are multiple methods of IPC that can be used to communicate between the processes (and perhaps to update the code and data of the new process to be like the parent, if the functionality that **fork** provides is needed). For more on IPC, see Interprocess Communications.

Windows and UNIX graphical models are very different. UNIX uses the X Window System GUI, while Windows uses GDI. Though similar in concept, there is no simple mapping of the X API to the GDI API. However, OpenGL support is available for migrating UNIX OpenGL-based applications. And there are X clients and X servers for Windows. See Device Contexts for information on GDI.

Basic UNIX applications, including many CGI applications, should port easily to Visual C++ running on Windows NT and Windows 2000. Functions like **open**, **fopen**, **read**, **write** and others are available in the Visual C++ run-time library. Also, there is a one-to-one mapping between C UNIX APIs and Win32 APIs: **open** to **CreateFile**, **read** to **ReadFile**, **write** to **WriteFile**, **ioctl** to **DeviceIOControl**, **close** to **CloseFile**, and so on.

Windows NT and Windows 2000 POSIX subsystem

Another option UNIX programmers look at is the Windows NT and Windows 2000 POSIX subsystem. However, it only supports POSIX 1003.1, which was the only POSIX version standardized when Windows NT was created. Since then, there has been little demand for extending this subsystem, because most applications have been converted to Win32. The 1003.1 system is of limited interest for fully featured applications, because it does not include many capabilities (such as those in 1003.2, network support, and so on). Full featured applications run under the Windows NT and Windows 2000 POSIX subsystem do not have access to Windows NT and Windows 2000 features available to Win32 applications, such as memory-mapped files, networking, and graphics. Applications such as VI, LS, and GREP are the main targets for the Windows NT and Windows 2000 POSIX subsystem.

See Also

Reference

UNIX

UNIX Code Migration Guide

Concepts

Inference Rules

Other Resources

Porting and Upgrading Programs

UNIX Migration Project Guide

How to: Upgrade Wizard-Generated Managed Extensions for C++ Code

Visual Studio wizards generate code in your source files, either when the project is created, or when functionality is added via a code wizard.

When upgrading a project, it is usually necessary to upgrade the wizard-generated code as well. For Microsoft Visual C++ 2005, a common scenario is upgrading code written with Managed Extensions for C++ to use new Visual C++ language features instead.

Upgrading Managed Extensions for C++

To upgrade code that uses Managed Extensions C++ code:

- Convert the code manually, using the Managed Extensions for C++ Syntax Upgrade Checklist.
- Use Microsoft Visual C++ 2005 to generate a new project of the same type, then either move your existing code into the new project or cut-and-paste the wizard-generated portions of the new project into the existing one.

For the most part, upgrading from Managed Extensions for C++ is just a matter of changing the syntax, but there are some semantic differences.

See Also

Reference

/clr (Common Language Runtime Compilation)

Other Resources

Porting and Upgrading Programs

Managed Extensions for C++ Syntax Upgrade Checklist

For Visual C++ 2005, Visual C++ includes new features for targeting virtual machines with garbage collection, such as the .NET Framework. In a future release, support for Managed Extensions for C++ will be removed.

Code using Managed Extensions for C++ can still be compiled with **/clr:oldSyntax**. See /clr (Common Language Runtime Compilation) for more information For more information on using the new syntax, see:

- .NET Programming in C++
- How To: Migrate to /clr
- C++/CLI Migration Primer
- New Language Features in Visual C++

This topic lists the syntactic differences between Managed Extensions for C++ and the new Visual C++ syntax.

To Update Managed Extensions for C++ Code

- Remove #using <mscorlib.dll>. (This assembly is now referenced by default.)
- Replace **__gc class** with **ref class**.
- Replace **__gc struct** with **ref struct**.
- Replace **__value class** with **value class**.
- Replace **__value struct** with **value struct**.
- Remove default constructors from value classes. (Default constructors for value types are allowed in Managed Extensions for C++, but the nature of the CLR prevents them from being called reliably, so this feature has been removed.)

For more information about reference and value types, see Classes and Structs (Managed).

- Replace **__gc __interface** with **interface class**.

For more information about the interface keyword, see interface class.

- Replace __abstract with **abstract**, and move it after the class name. For more information, see abstract (Visual C++).
- Replace __sealed with **sealed**, and move it after the class name. For more information, see sealed.
- Replace __property with property, combine the get and set methods into a single property block, and remove the trailing underbar/name suffix from these accessors. For simple properties, the accessors need not be defined explicitly. The syntax for indexed properties requires bracket pairs after the property type. For more information, see, How to: Use Simple Properties and How to: Use Indexed Properties.
- Replace __event with event (Visual C++).
- Replace **__value enum** with enum class.
- Replace pointers to reference types followed (defined with **__gc***) with **^**. For more information, see ^ (Handle to Object on Managed Heap).
- Replace pointers to values types that potentially live in the garbage collected heap with interior_ptr.
- Replace pointers declared with __pin with instances of pin_ptr, unless a whole object is being pinned, in which case use interior_ptr, and then use pin_ptr on the interior pointer. For Visual C++ 2005, this template is defined in the cli Namespace.
- Replace **new** with gcnew.
- Replace 0 or null with nullptr when used to indicate that a pointer or handle into the garbage collected heap does not indicate an object and when using comparisons to check pointers and handles.
- Update garbage-collected arrays declared with **__gc[]** to use the array (Visual C++) type. For Visual C++ 2005, this template is defined in the cli Namespace.

- Remove all instances of __box. For more information, see Implicit Boxing.
- Remove the "S" prefix from string literals (the compiler now determines if a string is literal based on the context in which it is used.)
- Replace the explicit declaration and usage of overloaded operators required in Managed Extensions for C++ (such as **op_Addition**, **op_Subtraction**) with the traditional C++ syntax. For more information, see User-Defined Operators and Operator Overloading.
- Update conversion operators (which use **op_Implicit** and **op_Explicit** in Managed Extensions for C++ syntax.) For more information, see User-Defined Conversions.
- Replace __typeof with typeid.
- Replace __try_cast with safe_cast. For Visual C++ 2005, this template is defined in the cli Namespace.
- If you were signing your assembly (also known as giving your assembly a strong name) with the CLR's assembly-signing attributes, you should remove those attributes from your code and instead use the assembly-signing linker options. For more information, see Strong Name Assemblies (Assembly Signing).

See Also

Concepts

Mixed (Native and Managed) Assemblies
New Language Features in Visual C++

How to: Upgrade Projects from Previous Versions of Visual C++

Projects built in Visual C++ since version 4.*x* can be opened and saved as projects.

When a project in the format of a previous version of Visual C++ is opened, a backup copy of the project file is made. Note that any change to the new project file will cause the old project file to become obsolete. Also, note that attempting to rename the old project file so it can be opened will cause you to lose the changes that were made to the project file created in the upgrade.

If you have many projects that you would like to convert in a batch process, see How to: Upgrade Visual C++ Projects to Visual Studio .NET in Batch Mode.

You may want or need to modify the new project:

- To fix WINVER-related compilation errors
- Standard C++ Library Changes
- ATL and MFC Version Numbers

For additional information, see:

- Product Changes in Visual C++ Releases
- Library Changes in Visual C++ Releases
- Visual C++ .NET 2003 Enhanced Compiler Conformance
- Breaking Changes in the Visual C++ Compiler
- Nonstandard Behavior

Upgrade Previous Visual C++ Enterprise Edition Projects

Data sources have their own project in Visual C++. You do not need a C++ project; your project can consist solely of data sources, and you can edit and debug the stored procedures within the data sources.

The data sources appear as a separate database project in Server Explorer. You can switch to Server Explorer by clicking the **Server Explorer** tab. When you open a project built in a previous version of Visual C++ Enterprise Edition, a database project is automatically created for the data sources. This project appears in Solution Explorer, along with the C++ project that once contained the data sources.

See Also

Other Resources

Previous Product Changes

How to: Upgrade Visual C++ Projects to Visual Studio .NET in Batch Mode

The following JScript sample can be run at the command line to upgrade one or more Visual C++ 6.0 projects.

Example

```
// when running this at the command line,
// call it with CScript so you don't get UI.
// example: CScript convert.js e:\yourprojects\old.dsp e:\yourprojects\new.vcproj
// NOTE: full path required to both input and output files

// or set default script engine to the command line doing this first
// example: CScript //H:CScript

// Once you set the environment, run the .js file like a .bat file

// To have a batch file loop through all the .dsp files
// in a directory, write a batch file that looked like this
// (Windows NT 4 or Windows 2000 only)
// CScript //H:CScript //Nologo
// for /R %%i in (*.dsp) do convert.js %%i >> .\Convert.log

var vcProj = new ActiveXObject("VisualStudio.VCProjectEngine.8.0");
var objFile = new ActiveXObject("Scripting.FileSystemObject");
var objArgs = WScript.Arguments;

// check the arguments to be sure it's right
if (objArgs.Count() < 2)
{
   WScript.Echo("VC6 or 5 DSP Project File Conversion");
   WScript.Echo("Opens specified .dsp and converts to VC8 Format.");
   WScript.Echo("Will create project file with .vcproj extension");
   WScript.Echo("\n\tusage: <full path\project.dsp> <full path\project.vcproj>");
   WScript.Quit(1);
}

WScript.Echo("\nConverting: "+ objArgs.Item(0));
// If there is a file name of the .vcproj extension, do not convert
var vcProject = vcProj.LoadProject(objArgs.Item(0));
if (!objFile.FileExists(vcProject.ProjectFile))
{
   // specify name and location of new project file
   vcProject.ProjectFile = objArgs.Item(1);

   // call the project engine to save this off.
   // when no name is shown, it will create one with the .vcproj name
   vcProject.Save();
   WScript.Echo("New Project Name: "+vcProject.ProjectFile+"\n");
}
else
{
   WScript.Echo("ERROR!: "+vcProject.ProjectFile+" already exists!\n");
}
```

See Also

Other Resources

Previous Product Changes

How to: Modify WINVER

When you build an MFC project from a previous release in Visual C++ .NET, you may see compilation errors because your project is now using a different value for WINVER.

Remarks

When your project was being built in a previous version of Visual C++, WINVER may have been set to 400, to allow compilation using an earlier version of Platform SDK header files, such as may have been found on Windows 95 or Windows NT 4.0.

You may have implicitly gotten your WINVER=400 setting in one of two ways:

- By using the earlier version of the Platform SDK header files that were available with the earliest versions of Visual C++ 6.0.
- By including winres.h in your Visual C++ 6.0 project. This file contained a definition of 400 for WINVER.

If you got your WINVER=400 setting by either of these means, if you still need WINVER=400 to be set, and if your project is now getting compiler errors in Visual C++ .NET, you need to explicitly define WINVER=400. For example, in a header file, you could add the following line:

```
#define  WINVER  0x0400
```

You can also define this macro with the **/D** compiler option; see /D (Preprocessor Definitions)for more information.

See Also

Other Resources

Previous Product Changes

Conversion failed Dialog Box

The project system was not able to open a project from a previous version of Visual C++ in Visual C++ .NET. Possible reasons include:

- A valid project file was not specified.
- The project did not contain a Win32 platform configuration. In order for a Visual C++ project from a previous version of Visual C++ to be opened in Visual C++ .NET, there needs to be at least one configuration with a Win32 platform target.

See Also

Other Resources

Previous Product Changes

How to: Access ClassWizard and WizardBar Functionality

In previous versions of Visual C++, ClassWizard and WizardBar helped you develop your MFC application. ClassWizard and WizardBar were removed in Visual C++ .NET 2002. This topic discusses where ClassWizard and WizardBar functionality can be accessed in Visual C++.

WizardBar

To add a new class

- In **Class View** or **Solution Explorer**, right-click the project node. On the shortcut menu, click **Add** and then click **Add Class**.

To add a function to a class

- In **Class View**, right-click the class. On the shortcut menu, click **Add** and then click **Add Function**.

To go to a definition

- Use the Navigation Bar, which is at the top of the text editor window. You can turn the Navigation Bar off from the **General** tab of the **C/C++** folder, which is in the **Text Editor** folder of the **Options** dialog box (**Tools** menu).

  You can also use the Object Browser. See Searching for Symbols: Objects, Definitions and References for more information.

ClassWizard

For more information about wizards that add code to a project, see Adding Functionality with Code Wizards.

To add a member variable

- In **Class View**, right-click the class to which you want to add a member variable. On the shortcut menu, click **Add** and then click **Add Variable**.

  For more information, see Adding a Member Variable.

To add a handler for a message

- See Mapping Messages to Functions.

To add methods and properties to automation interfaces

- In Class View, right-click an interface that supports automation. On the shortcut menu, click **Add** and then click **Add Method** or **Add Property**.

  For more information, see Adding a Method or Adding a Property.

To add ActiveX events

- In Class View for an ActiveX control project, right-click the class to which you want to add an event. On the shortcut menu, click **Add** and then click **Add Event**.

  For more information, see Adding an Event.

To add a handler (function) for a user-interface item

- Select the item in Resource View. For example, select a button. Press F4 to open the **Properties** window and click the **ControlEvents** button. Select a message in the left column and click **Add** from the right column. The function will appear in Class View.

  For more information, see Adding an MFC Message Handler.

See Also

Concepts

How to: Upgrade Projects from Previous Versions of Visual C++

C++/CLI Migration Primer

This is a guide to moving your Visual C++ programs from Managed Extensions for C++ to Visual C++ 2005. For a checklist summary of syntactic changes, see Managed Extensions for C++ Syntax Upgrade Checklist.

C++/CLI extends a dynamic component programming paradigm to the ISO-C++ standard language. The new language offers a number of significant improvements over Managed Extensions. This section provides an enumerated listing of the Managed Extensions for C++ language features and their mapping to Visual C++ 2005 where such a mapping exists, and points out those constructs for which no mapping exists.

In This Section

Outline of Changes

A high-level outline for quick reference, providing a listing of the changes under five general categories.

Language Keywords

Discusses changes in language keywords, including the removal of the double underscore and the introduction of both contextual and spaced keywords.

The Managed Types

Looks at syntactic changes in the declaration of the Common Type System (CTS) – this includes changes in the declaration of classes, arrays (including the parameter array), enums, and so on.

Member Declarations within a Class or Interface

Presents the changes involving class members such as scalar properties, index properties, operators, delegates, and events.

Value Types and Their Behaviors

Focuses on value types and the new family of interior and pinning pointers. It also discusses a number of significant semantics changes such as the introduction of implicit boxing, immutability of boxed value types, and the removal of support for default constructors within value classes.

General Language Changes

Details semantic changes such as support for cast notation, string literal behavior, and changes in the semantics between ISO-C++ and C++/CLI.

See Also

Concepts

Mixed (Native and Managed) Assemblies

New Language Features in Visual C++

Outline of Changes

This outline shows you examples of some of the changes in the language from Managed Extensions for C++ to Visual C++ 2005. Follow the link that accompanies each item for more information.

No Double Underscore Keywords

The double underscore in front of all keywords has been removed, with one exception. Thus, **__value** becomes **value**, and **__interface** becomes **interface**, and so on. To prevent name clashes between keywords and identifiers in user code, keywords are primarily treated as contextual.

See Language Keywords for more information.

Class Declarations

Managed Extensions syntax:

```
__gc class Block {};                          // reference class
__value class Vector {};                      // value class
__interface I {};                        // interface class
__gc __abstract class Shape {};               // abstract class
__gc __sealed class Shape2D : public Shape {}; // derived class
```

New syntax:

```
ref class Block {};                // reference class
value class Vector {};             // value class
interface class I {};        // interface class
ref class Shape abstract {};       // abstract class
ref class Shape2D sealed: Shape{}; // derived class
```

See The Managed Types for more information.

Object Declaration

Managed Extensions syntax:

```
public __gc class Form1 : public System::Windows::Forms::Form {
private:
   System::ComponentModel::Container __gc *components;
   System::Windows::Forms::Button   __gc *button1;
   System::Windows::Forms::DataGrid __gc *myDataGrid;
   System::Data::DataSet  __gc *myDataSet;
};
```

New syntax:

```
public ref class Form1 : System::Windows::Forms::Form {
   System::ComponentModel::Container^ components;
   System::Windows::Forms::Button^ button1;
   System::Windows::Forms::DataGrid^ myDataGrid;
   System::Data::DataSet^ myDataSet;
};
```

See Declaration of a CLR Reference Class Object for more information.

Managed Heap Allocation

Managed Extensions syntax:

```
Button* button1 = new Button; // managed heap
int *pi1 = new int;           // native heap
```

```
Int32 *pi2 = new Int32;       // managed heap
```

New syntax:

```
Button^ button1 = gcnew Button;        // managed heap
int * pi1 = new int;                   // native heap
Int32^ pi2 = gcnew Int32;              // managed heap
```

See Declaration of a CLR Reference Class Object for more information.

A Tracking Reference to No Object

Managed Extensions syntax:

```
// OK: we set obj to refer to no object
Object * obj = 0;

// Error: no implicit boxing
Object * obj2 = 1;
```

New syntax:

```
// Incorrect Translation
// causes the implicit boxing of both 0 and 1
Object ^ obj = 0;
Object ^ obj2 = 1;

// Correct Translation
// OK: we set obj to refer to no object
Object ^ obj = nullptr;

// OK: we initialize obj2 to an Int32^
Object ^ obj2 = 1;
```

See Declaration of a CLR Reference Class Object for more information.

Array Declaration

The CLR array has been redesigned. It is similar to the stl `vector` template collection, but maps to the underlying `System::Array` class – that is, it is not a template implementation.

See Declaration of a CLR Array for more information.

Array as Parameter

Managed Extensions array syntax:

```
void PrintValues( Object* myArr __gc[]);
void PrintValues( int myArr __gc[,,]);
```

New array syntax:

```
void PrintValues( array<Object^>^ myArr );
void PrintValues( array<int,3>^ myArr );
```

Array as Return Type

Managed Extensions array syntax:

```
Int32 f() [];
int GetArray() __gc[];
```

New array syntax:

```
array<Int32>^ f();
array<int>^ GetArray();
```

Shorthand Initialization of Local CLR Array

Managed Extensions array syntax:

```
int GetArray() __gc[] {
   int a1 __gc[] = { 1, 2, 3, 4, 5 };
   Object* myObjArray __gc[] = { __box(26), __box(27), __box(28),
                                 __box(29), __box(30) };

   return a1;
}
```

New array syntax:

```
array<int>^ GetArray() {
   array<int>^ a1 = {1,2,3,4,5};
   array<Object^>^ myObjArray = {26,27,28,29,30};

   return a1;
}
```

Explicit CLR Array Declaration

Managed Extensions array syntax:

```
Object* myArray[] = new Object*[2];
String* myMat[,] = new String*[4,4];
```

New array syntax:

```
array<Object^>^ myArray = gcnew array<Object^>(2);
array<String^,2>^ myMat = gcnew array<String^,2>(4,4);
```

New to language: explicit array initialization that follows gcnew

```
// explicit initialization list follow gcnew
// is not supported in Managed Extensions
array<Object^>^ myArray =
   gcnew array<Object^>(4){ 1, 1, 2, 3 };
```

Scalar Properties

Managed Extensions property syntax:

```
public __gc __sealed class Vector {
   double _x;

public:
   __property double get_x(){ return _x; }
   __property void set_x( double newx ){ _x = newx; }
};
```

New property syntax:

```
public ref class Vector sealed {
   double _x;

public:
```

```
   property double x
   {
      double get()             { return _x; }
      void   set( double newx ){ _x = newx; }
   } // Note: no semi-colon …
};
```

New to language: trivial properties

```
public ref class Vector sealed {
public:
   // equivalent shorthand property syntax
   // backing store is not accessible
   property double x;
};
```

See Property Declaration for more information.

Indexed Properties

Managed Extensions indexed property syntax:

```
public __gc class Matrix {
   float mat[,];

public:
   __property void set_Item( int r, int c, float value) { mat[r,c] = value; }
   __property int get_Item( int r, int c ) { return mat[r,c]; }
};
```

New indexed property syntax:

```
public ref class Matrix {
   array<float, 2>^ mat;

public:
   property float Item [int,int] {
      float get( int r, int c ) { return mat[r,c]; }
      void set( int r, int c, float value ) { mat[r,c] = value; }
   }
};
```

New to language: class-level indexed property

```
public ref class Matrix {
   array<float, 2>^ mat;

public:
   // ok: class level indexer now
   //     Matrix mat;
   //     mat[ 0, 0 ] = 1;
   //
   // invokes the set accessor of the default indexer

   property float default [int,int] {
      float get( int r, int c ) { return mat[r,c]; }
      void set( int r, int c, float value ) { mat[r,c] = value; }
   }
};
```

See Property Index Declaration for more information.

Overloaded Operators

Managed Extensions operator overload syntax:

```
public __gc __sealed class Vector {
public:
   Vector( double x, double y, double z );

   static bool    op_Equality( const Vector*, const Vector* );
   static Vector* op_Division( const Vector*, double );
};

int main() {
   Vector *pa = new Vector( 0.231, 2.4745, 0.023 );
   Vector *pb = new Vector( 1.475, 4.8916, -1.23 );

   Vector *pc = Vector::op_Division( pa, 4.8916 );

   if ( Vector::op_Equality( pa, pc ))
      ;
}
```

New operator overload syntax:

```
public ref class Vector sealed {
public:
   Vector( double x, double y, double z );

   static bool    operator ==( const Vector^, const Vector^ );
   static Vector^ operator /( const Vector^, double );
};

int main() {
   Vector^ pa = gcnew Vector( 0.231, 2.4745, 0.023 );
   Vector^ pb = gcnew Vector( 1.475, 4.8916, -1.23 );

   Vector^ pc = pa / 4.8916;
   if ( pc == pa )
      ;
}
```

See Overloaded Operators for more information.

Conversion Operators

Managed Extensions conversion operator syntax:

```
__gc struct MyDouble {
   static MyDouble* op_Implicit( int i );
   static int op_Explicit( MyDouble* val );
   static String* op_Explicit( MyDouble* val );
};
```

New conversion operator syntax:

```
ref struct MyDouble {
public:
   static operator MyDouble^ ( int i );
   static explicit operator int ( MyDouble^ val );
   static explicit operator String^ ( MyDouble^ val );
};
```

See Changes to Conversion Operators for more information.

Explicit Override of an Interface Member

Managed Extensions explicit override syntax:

```
public __gc class R : public ICloneable {
   // to be used through ICloneable
   Object* ICloneable::Clone();

   // to be used through an R
   R* Clone();
};
```

New explicit override syntax:

```
public ref class R : public ICloneable {
   // to be used through ICloneable
   virtual Object^ InterfaceClone() = ICloneable::Clone;

   // to be used through an R
   virtual R^ Clone();
};
```

See Explicit Override of an Interface Member for more information.

Private Virtual Functions

Managed Extensions private virtual function syntax:

```
__gc class Base {
private:
   // inaccessible to a derived class
   virtual void g();
};

__gc class Derived : public Base {
public:
   // ok: g() overrides Base::g()
   virtual void g();
};
```

New private virtual function syntax

```
ref class Base {
private:
   // inaccessible to a derived class
   virtual void g();
};

ref class Derived : public Base {
public:
   // error: cannot override: Base::g() is inaccessible
   virtual void g() override;
};
```

See Private Virtual Functions for more information.

CLR Enum Type

Managed Extensions enum syntax:

```
__value enum e1 { fail, pass };
public __value enum e2 : unsigned short  {
   not_ok = 1024,
   maybe, ok = 2048
};
```

New enum syntax:

```
enum class e1 { fail, pass };
public enum class e2 : unsigned short {
   not_ok = 1024,
   maybe, ok = 2048
};
```

Apart from this small syntactic change, the behavior of the CLR enum type has been changed in a number of ways:

- A forward declaration of a CLR enum is no longer supported.
- The overload resolution between the built-in arithmetic types and the Object class hierarchy has reversed between Managed Extensions and Visual C++ 2005. As a side-effect, CLR enums are no longer implicitly converted to arithmetic types.
- In the new syntax, a CLR enum maintains its own scope, which is not the case in Managed Extensions. Previously, enumerators were visible within the containing scope of the enum; now, enumerators are encapsulated within the scope of the enum.

See CLR Enum Type for more information.

Removal of __box Keyword

Managed Extensions boxing syntax:

```
Object *o = __box( 1024 ); // explicit boxing
```

New boxing syntax:

```
Object ^o = 1024; // implicit boxing
```

See A Tracking Handle to a Boxed Value for more information.

Pinning Pointer

Managed Extensions pinning pointer syntax:

```
__gc struct H { int j; };

int main() {
   H * h = new H;
   int __pin * k = & h -> j;
};
```

New pinning pointer syntax:

```
ref struct H { int j; };

int main() {
   H^ h = gcnew H;
   pin_ptr<int> k = &h->j;
}
```

See Value Type Semantics for more information.

__typeof Keyword becomes typeid

Managed Extensions typeof syntax:

```
Array* myIntArray =
   Array::CreateInstance( __typeof(Int32), 5 );
```

New typeid syntax:

```
Array^ myIntArray =
   Array::CreateInstance( Int32::typeid, 5 );
```

See typeof Goes to T::typeid for more information.

See Also

Concepts

C++/CLI Migration Primer
Managed Extensions for C++ Syntax Upgrade Checklist
New Language Features in Visual C++

Language Keywords

Several language keywords changed from Managed Extensions for C++ to Visual C++ 2005.

In the new Visual C++ 2005 syntax, the double underscore is removed as a prefix from all keywords (with one exception: **__identifier** is retained). For example, a property is now declared as **property**, not **__property**.

There were two primary reasons for using the double-underscore prefix in Managed Extensions:

- It is the conformant method of providing local extensions to the ISO-C++ Standard. A primary goal of the Managed Extensions design was to not introduce incompatibilities with the standard language, such as new keywords and tokens. It was this reason, in large part, which motivated the choice of pointer syntax for the declaration of objects of managed reference types.
- The use of the double underscore, apart from its conformant aspect, is also a reasonable guarantee of being non-invasive with the existing code base of the language users. This was a second primary goal of the Managed Extensions design.

In spite of removing the double underscores, Microsoft remains committed to being conformant. However, support for the CLR dynamic object model represents a new and powerful programming paradigm. Support of this new paradigm requires its own high-level keywords and tokens. We have sought to provide a first-class expression of this new paradigm while integrating it and supporting the standard language. The new syntax design provides a first class programming experience of these two disparate object models.

Similarly, we are concerned with maximizing the non-invasive nature of these new language keywords. This has been accomplished with the use of contextual and spaced keywords. Before we look at the actual new language syntax, let’s try to make sense of these two special keyword flavors.

A contextual keyword has a special meaning within specific program contexts. Within the general program, for example, **sealed** is treated as an ordinary identifier. However, when it occurs within the declaration portion of a managed reference class type, it is treated as a keyword within the context of that class declaration. This minimizes the potential invasive impact of introducing a new keyword in the language, something that we feel is very important to users with an existing code base. At the same time, it allows users of the new functionality to have a first-class experience of the additional language feature – something that wasn't possible with Managed Extensions. For an example of how **sealed** is used see Declaration of a Managed Class Type.

A spaced keyword, such as **value class**, is a special case of a contextual keyword. It pairs an existing keyword with a contextual modifier separated by a space. The pair is treated as a single unit rather than as two separate keywords.

See Also

Concepts

C++/CLI Migration Primer
New Language Features in Visual C++

The Managed Types

The syntax for the declaration of managed types and the creation and use of objects of these types has been significantly altered from Managed Extensions for C++ to Visual C++ 2005. This was done to promote their integration within the ISO-C++ type system. These changes are presented in detail in the following subsections.

In This Section

Declaration of a Managed Class Type

Discusses how to declare a managed **class**, **struct**, or **interface**.

Declaration of a CLR Reference Class Object

Discusses how to declare a reference class type object using a tracking handle.

Declaration of a CLR Array

Explains how to declare and initialize an array.

Changes in Constructor Initialization Order

Discusses key changes in class constructor initialization order.

Changes in Destructor Semantics

Discusses non-deterministic finalization, `Finalize` versus `Dispose`, ramifications for reference objects, and use of an explicit `Finalize`.

Note: The discussion of delegates is deferred until Delegates and Events in order to present them with event members within a class, the general topic of Member Declarations within a Class or Interface. (For a detailed discussion of the rationale behind the introduction of the tracking reference syntax and the general shift in design, see Motivating the New Language Design.)

See Also

Reference

Classes and Structs (Managed)

array (Visual C++)

Concepts

C++/CLI Migration Primer

Declaration of a Managed Class Type

The way to declare a reference class type changed from Managed Extensions for C++ to Visual C++ 2005.

In Managed Extensions, a reference class type is prefaced with the **__gc** keyword. In the new syntax, the **__gc** keyword is replaced by one of two spaced keywords: **ref class** or **ref struct**. The choice of **struct** or **class** indicates the public (for **struct**) or private (for **class**) default access level of its members declared in an initial unlabeled section of the body of the type.

Similarly, in Managed Extensions, a value class type is prefaced with the **__value** keyword. In the new syntax, the **__value** keyword is replaced by one of two spaced keywords: **value class** or **value struct**.

An interface type, in Managed Extensions, was indicated with the keyword **__interface**. In the new syntax, this is replaced with **interface class**.

For example, the following class declarations in Managed Extensions:

```
public __gc class Block {};     // reference class
public __value class Vector {}; // value class
public __interface IFooBar {};  // interface class
```

Under the new syntax these are equivalently declared as follows:

```
public ref class Block {};         // reference class
public value class Vector {};      // value class
public interface class IFooBar {}; // interface class
```

Specifying the Class as Abstract

Under Managed Extensions, you put the keyword **__abstract** before the **class** keyword (either before or after the **__gc**) to indicate that the class is incomplete and that objects of the class cannot be created within the program:

```
public __gc __abstract class Shape {};
public __gc __abstract class Shape2D: public Shape {};
```

Under the new syntax, you specify the **abstract** contextual keyword following the class name and before either the class body, base class derivation list, or semicolon.

```
public ref class Shape abstract {};
public ref class Shape2D abstract : public Shape{};
```

Of course, the semantic meaning is unchanged.

Specifying the Class as Sealed

Under Managed Extensions, you put the keyword **__sealed** before the **class** keyword (either before or after **__gc**) to indicate that objects of the class cannot be inherited from:

```
public __gc __sealed class String {};
```

Under the new syntax, you specify the **sealed** contextual keyword following the class name and before either the class body, base class derivation list, or semicolon.

You can both derive a class and seal it. For example, the `String` class is implicitly derived from `Object`. The benefit of sealing a class is that it supports the static resolution (that is, at compile-time) of all virtual function calls through the sealed reference class object. This is because the **sealed** specifier guarantees that the `String` tracking handle cannot refer to a subsequently derived class that might provide an overriding instance of the virtual method being invoked. Here is an example of a sealed class in new syntax:

```
public ref class String sealed {};
```

One can also specify a class as both abstract and sealed – this is a special condition that indicates a static class. This is described in the CLR documentation as follows:

"A type that is both **abstract** and **sealed** should have only static members, and serves as what some languages call a namespace."

For example, here is a declaration of an abstract sealed class using the Managed Extensions syntax:

```
public __gc __sealed __abstract class State {
public:
   static State() {}
   static bool inParamList();

private:
   static bool ms_inParam;
};
```

and here is this declaration translated into the new syntax:

```
public ref class State abstract sealed {
public:
   static State();
   static bool inParamList();

private:
   static bool ms_inParam;
};
```

CLR Inheritance: Specifying the Base Class

Under the CLR object model, only public single inheritance is supported. However, Managed Extensions retained the ISO-C++ default interpretation of a base class without an access keyword as specifying a private derivation. This meant that each CLR inheritance declaration had to provide the **public** keyword to override the default interpretation.

```
// Managed Extensions: error: defaults to private derivation
__gc class Derived : Base {};
```

In the new syntax definition, the absence of an access keyword indicates a public derivation in a CLR inheritance definition. Thus, the **public** access keyword is now optional. While this does not require any modification of Managed Extensions for C++ code, I list this change here for completeness.

```
// New syntax: ok: defaults to public derivation
ref class Derived : Base{};
```

See Also

Reference

Classes and Structs (Managed)
abstract (Visual C++)
sealed

Concepts

The Managed Types

Declaration of a CLR Reference Class Object

The syntax to declare and instantiate an object of a reference class type has changed from Managed Extensions for C++ to Visual C++ 2005.

In Managed Extensions, a reference class type object is declared by using the ISO-C++ pointer syntax, with an optional use of the **__gc** keyword to the left of the star (*****). For example, here are a variety of reference class type object declarations under the Managed Extensions syntax:

```
public __gc class Form1 : public System::Windows::Forms::Form {
private:
   System::ComponentModel::Container __gc *components;
   Button __gc *button1;
   DataGrid __gc *myDataGrid;
   DataSet __gc *myDataSet;

   void PrintValues( Array* myArr ) {
      System::Collections::IEnumerator* myEnumerator =
         myArr->GetEnumerator();

      Array *localArray;
      myArr->Copy(myArr, localArray, myArr->Length);
   }
};
```

Under the new syntax, you declare a reference class type object by using a new declarative token (**^**) referred to formally as a *tracking handle* and more informally as a *hat*. (The tracking adjective means that a reference type sits in the CLR heap, and can therefore transparently move locations during garbage collection heap compaction. A tracking handle is transparently updated during runtime. Two similar concepts are the *tracking reference* (**%**), and the *interior pointer* (**interior_ptr<>**), discussed in Value Type Semantics.

The primary reasons to move the declarative syntax away from a reuse of the ISO-C++ pointer syntax are as follows:

- The use of the pointer syntax did not permit overloaded operators to be directly applied to a reference object. Rather, one had to call the operator by using its internal name, such as `rV1->op_Addition(rV2)` instead of the more intuitive `rV1+rV2`.
- A number of pointer operations, such as casting and pointer arithmetic, not permitted for objects stored on a garbage collected heap. The notion of a tracking handle better captures the nature of a CLR reference type.

The **__gc** modifier on a tracking handle is unnecessary and is not supported. The use of the object itself is not changed; it still accesses members through the pointer member selection operator (**->**). For example, here is the previous Managed Extensions code sample translated into the new syntax:

```
public ref class Form1: public System::Windows::Forms::Form {
private:
   System::ComponentModel::Container^ components;
   Button^ button1;
   DataGrid^ myDataGrid;
   DataSet^ myDataSet;

   void PrintValues( Array^ myArr ) {
      System::Collections::IEnumerator^ myEnumerator =
         myArr->GetEnumerator();

      Array ^localArray;
      myArr->Copy(myArr, localArray, myArr->Length);   }
};
```

Dynamic Allocation of an Object on the CLR Heap

In Managed Extensions, the existence of two **new** expressions to allocate between the native and managed heap was largely transparent. In almost all instances, the compiler is able to use the context to determine whether to allocate memory from the native or managed heap. For example,

```
Button *button1 = new Button; // OK: managed heap
int *pi1 = new int;           // OK: native heap
Int32 *pi2 = new Int32;       // OK: managed heap
```

When you do not want the contextual heap allocation, you could direct the compiler with either the **__gc** or **__nogc** keyword. In the new syntax, the separate nature of the two new expressions is made explicit with the introduction of the **gcnew** keyword. For example, the previous three declarations look as follows in the new syntax:

```
Button^ button1 = gcnew Button;        // OK: managed heap
int * pi1 = new int;                   // OK: native heap
Int32^ pi2 = gcnew Int32; // OK: managed heap
```

Here is the Managed Extensions initialization of the `Form1` members declared in the previous section:

```
void InitializeComponent() {
   components = new System::ComponentModel::Container();
   button1 = new System::Windows::Forms::Button();
   myDataGrid = new DataGrid();

   button1->Click +=
      new System::EventHandler(this, &Form1::button1_Click);
}
```

Here is the same initialization recast to the new syntax. Note that the hat is not required for the reference type when it is the target of a **gcnew** expression.

```
void InitializeComponent() {
   components = gcnew System::ComponentModel::Container;
   button1 = gcnew System::Windows::Forms::Button;
   myDataGrid = gcnew DataGrid;

   button1->Click +=
      gcnew System::EventHandler( this, &Form1::button1_Click );
}
```

A Tracking Reference to No Object

In the new syntax, `0` no longer represents a null address but is treated as an integer, the same as `1`, `10`, or `100`. A new special token represents a null value for a tracking reference. For example, in Managed Extensions, we initialize a reference type to address no object as follows:

```
// OK: we set obj to refer to no object
Object * obj = 0;

// Error: no implicit boxing
Object * obj2 = 1;
```

In the new syntax, any initialization or assignment of a value type to an `Object` causes an implicit boxing of that value type. In the new syntax, both `obj` and `obj2` are initialized to addressed boxed Int32 objects holding the values 0 and 1, respectively. For example:

```
// causes the implicit boxing of both 0 and 1
Object ^ obj = 0;
Object ^ obj2 = 1;
```

Therefore, in order to perform the explicit initialization, assignment, and comparison of a tracking handle to null, use a new keyword, **nullptr**. The correct revision of the original example looks as follows:

```
// OK: we set obj to refer to no object
Object ^ obj = nullptr;
```

```
// OK: we initialize obj2 to a Int32^
Object ^ obj2 = 1;
```

This complicates somewhat the porting of existing code into the new syntax. For example, consider the following value class declaration:

```
__value struct Holder {
   Holder( Continuation* c, Sexpr* v ) {
      cont = c;
      value = v;
      args = 0;
      env = 0;
   }

private:
   Continuation* cont;
   Sexpr * value;
   Environment* env;
   Sexpr * args __gc [];
};
```

Here, both `args` and `env` are CLR reference types. The initialization of these two members to `0` in the constructor cannot remain unchanged in the transition to the new syntax. Rather, they must be changed to **nullptr**:

```
value struct Holder {
   Holder( Continuation^ c, Sexpr^ v )
   {
      cont = c;
      value = v;
      args = nullptr;
      env = nullptr;
   }

private:
   Continuation^ cont;
   Sexpr^ value;
   Environment^ env;
   array<Sexpr^>^ args;
};
```

Similarly, tests against those members comparing them to `0` must also be changed to compare the members to `nullptr`. Here is the Managed Extensions syntax:

```
Sexpr * Loop (Sexpr* input) {
   value = 0;
   Holder holder = Interpret(this, input, env);

   while (holder.cont != 0) {
      if (holder.env != 0) {
         holder=Interpret(holder.cont,holder.value,holder.env);
      }
      else if (holder.args != 0) {
         holder =
         holder.value->closure()->
         apply(holder.cont,holder.args);
      }
   }

   return value;
}
```

Here is the revision, replacing each `0` instance with a `nullptr`. The translation tool helps in this transformation by automating many if not all occurrences, including use of the `NULL` macro.

```
Sexpr ^ Loop (Sexpr^ input) {
   value = nullptr;
   Holder holder = Interpret(this, input, env);

   while ( holder.cont != nullptr ) {
      if ( holder.env != nullptr ) {
         holder=Interpret(holder.cont,holder.value,holder.env);
      }
      else if (holder.args != nullptr ) {
         holder =
         holder.value->closure()->
         apply(holder.cont,holder.args);
      }
   }

   return value;
}
```

The **nullptr** is converted into any pointer or tracking handle type but is not promoted to an integral type. For example, in the following set of initializations, the `nullptr` is valid only as an initial value to the first two.

```
// OK: we set obj and pstr to refer to no object
Object^ obj = nullptr;
char*   pstr = nullptr; // 0 would also work here

// Error: no conversion of nullptr to 0 …
int ival = nullptr;
```

Similarly, given an overloaded set of methods such as the following:

```
void f( Object^ ); // (1)
void f( char* );   // (2)
void f( int );     // (3)
```

An invocation with `nullptr` literal, such as the following,

```
// Error: ambiguous: matches (1) and (2)
f(  nullptr );
```

is ambiguous because the `nullptr` matches both a tracking handle and a pointer, and there is no preference given to one type over the other. (This situation requires an explicit cast in order to disambiguate.)

An invocation with `0` exactly matches instance (3):

```
// OK: matches (3)
f( 0 );
```

because `0` is of type integer. Were `f(int)` not present, the call would unambiguously match `f(char*)` through a standard conversion. The matching rules give precedence of an exact match over a standard conversion. In the absence of an exact match, a standard conversion is given precedence over an implicit boxing of a value type. That is why there is no ambiguity.

See Also

Reference

Classes and Structs (Managed)
^ (Handle to Object on Managed Heap)
nullptr

Concepts

The Managed Types

Declaration of a CLR Array

The syntax for declaring, instantiating, and initializing a managed array has changed from Managed Extensions for C++ to Visual C++ 2005.

The declaration of a CLR array object in Managed Extensions was an extension of the standard array declaration in which a **__gc** keyword was placed between the name of the array object and its possibly comma-filled dimension, as in the following pair of examples:

```
void PrintValues( Object* myArr __gc[]);
void PrintValues( int myArr __gc[,,]);
```

This has been simplified in the new syntax, in which we use a template-like declaration similar to the STL `vector` declaration. The first parameter indicates the element type. The second parameter specifies the array dimension (with a default value of 1, so only multiple dimensions require a second argument). The array object itself is a tracking handle, and so must be given a hat. If the element type is also a reference type, it also requires a hat. For example, the above example, when expressed in the new syntax, looks as follows:

```
void PrintValues( array<Object^>^ myArr );
void PrintValues( array<int,3>^ myArr );
```

Because a reference type is a tracking handle rather than an object, it is possible to specify a CLR array as the return type of a function. (In contrast, it is not possible to specify the native array as the return type of a function.) The syntax for doing this in Managed Extensions was somewhat non-intuitive. For example:

```
Int32 f() [];
int GetArray() __gc[];
```

In Visual C++ 2005, the declaration is much simpler. For example,

```
array<Int32>^ f();
array<int>^ GetArray();
```

The shorthand initialization of a local managed array is supported in both versions of the language. For example:

```
int GetArray() __gc[] {
   int a1 __gc[] = { 1, 2, 3, 4, 5 };
   Object* myObjArray __gc[] = {
      __box(26), __box(27), __box(28), __box(29), __box(30)
   };
   return a1;
}
```

is considerably simplified in the new syntax (note that because boxing is implicit in the new syntax, the **__box** operator has been eliminated – see Value Types and Their Behaviors for a discussion):

```
array<int>^ GetArray() {
   array<int>^ a1 = {1,2,3,4,5};
   array<Object^>^ myObjArray = {26,27,28,29,30};
   return a1;
}
```

Because an array is a CLR reference type, the declaration of each array object is a tracking handle. Therefore, array objects must be allocated on the CLR heap. (The shorthand notation hides the managed heap allocation.) Here is the explicit form of an array object initialization under Managed Extensions:

```
Object* myArray[] = new Object*[2];
String* myMat[,] = new String*[4,4];
```

Under the new syntax, the **new** expression is replaced with **gcnew**. The dimension sizes are passed as parameters to the **gcnew** expression, as follows:

```
array<Object^>^ myArray = gcnew array<Object^>(2);
array<String^,2>^ myMat = gcnew array<String^,2>(4,4);
```

In the new syntax, an explicit initialization list can follow the **gcnew** expression; this was not supported in Managed Extensions. For example:

```
// explicit initialization list following gcnew
// was not supported in Managed Extensions
array<Object^>^ myArray =
   gcnew array<Object^>(4){ 1, 1, 2, 3 };
```

See Also

Reference

array (Visual C++)

Concepts

The Managed Types

Changes in Constructor Initialization Order

The order of initialization for class constructors has changed from Managed Extensions for C++ to Visual C++ 2005.

Comparison of Constructor Initialization Order

Under Managed Extensions for C++, constructor initialization occurred in the following order:

1. The constructor of the base class, if any, is invoked.
2. The initialization list of the class is evaluated.
3. The code body of the class constructor is executed.

This order of execution follows the same conventions as in native C++ programming. The new Visual C++ language prescribes the following execution order for CLR classes:

1. The initialization list of the class is evaluated.
2. The constructor of the base class, if any, is invoked.
3. The code body of the class constructor is executed.

Note this change applies only to CLR classes; native classes in Visual C++ 2005 still follow the previous conventions. In both cases, these rules cascade upward throughout the whole hierarchy chain of a given class.

Consider the following code example using Managed Extensions for C++:

```
__gc class A
{
public:
   A() : _n(1)
   {
   }

protected:
   int _n;
};

__gc class B : public A
{
public:
   B() : _m(_n)
   {
   }
private:
   int _m;
};
```

Following the constructor initialization order prescribed above, we should see the following order of execution when new instances of class `B` are constructed:

1. The constructor of the base class `A` is invoked. The `_n` member is initialized to `1`.
2. The initialization list for class `B` is evaluated. The `_m` member is initialized to `1`.
3. The code body of class `B` is executed.

Now consider the same code in the new Visual C++ syntax:

```
ref class A
{
public:
   A() : _n(1)
   {
   }
```

```
protected:
   int _n;
};

ref class B : A
{
public:
   B() : _m(_n)
   {
   }
private:
   int _m;
};
```

The order of execution when new instances of class `B` are constructed under the new syntax is:

1. The initialization list for class `B` is evaluated. The `_m` member is initialized to `0` (`0` is the uninitialized value of the `_m` class member).
2. The constructor of the base class `A` is invoked. The `_n` member is initialized to `1`.
3. The code body of class `B` is executed.

Note that a similar syntax produces different results for these code examples. The constructor of class `B` depends on a value from base class `A` to initialize its member. However, the constructor for class `A` has not yet been invoked. Such a dependency can be especially dangerous when the inherited class depends on a memory or resource allocation to occur in the base class constructor.

What This Means Going from Managed Extensions for C++ to Visual C++ 2005

In many cases the changes to the execution order of class constructors should be transparent to the programmer because base classes have no notion of the behavior of inherited classes. However, as these code examples illustrate, the constructors of inherited classes can be greatly affected when their initialization lists depend on the values of base class members. When you move your code from Managed Extensions for C++ to the new syntax, consider moving such constructs to the body of the class constructor, where execution is guaranteed to occur last.

See Also

Reference

Constructors (C++)

Constructor Initializers

Concepts

The Managed Types

Changes in Destructor Semantics

Semantics for class destructors have changed significantly from Managed Extensions for C++ to Visual C++ 2005.

In Managed Extensions, a class destructor was permitted within a reference class but not within a value class. This has not changed in the new syntax. However, the semantics of the class destructor have changed. This topic focuses on the reasons of that change and discusses how it affects the translation of existing CLR code. It is probably the most important programmer-level change between the two versions of the language.

Non-deterministic Finalization

Before the memory associated with an object is reclaimed by the garbage collector, an associated `Finalize` method, if present, is invoked. You can think of this method as a kind of super-destructor because it is not tied to the program lifetime of the object. We refer to this as finalization. The timing of just when or even whether a `Finalize` method is invoked is undefined. This is what we mean when we say that garbage collection exhibits non-deterministic finalization.

Non-deterministic finalization works well with dynamic memory management. When available memory becomes scarce, the garbage collector kicks in. Under a garbage collected environment, destructors to free memory are unnecessary. Non-deterministic finalization does not work well, however, when an object maintains a critical resource such as a database connection or a lock of some sort. In this case, we should release that resource as soon as possible. In the native world, that is achieved by using a constructor/destructor pair. As soon as the lifetime of the object ends, either when the local block within which it is declared ends, or when the stack unravels because of a thrown exception, the destructor executes and the resource is automatically released. This approach works very well, and its absence under Managed Extensions was sorely missed.

The solution provided by the CLR is for a class to implement the `Dispose` method of the `IDisposable` interface. The problem here is that `Dispose` requires an explicit invocation by the user. This is error-prone. The C# language provides a modest form of automation in the form of a special **using** statement. The Managed Extensions design provided no special support.

Destructors in Managed Extensions for C++

In Managed Extensions, the destructor of a reference class is implemented by using the following two steps:

1. The user-supplied destructor is renamed internally to `Finalize`. If the class has a base class (remember, under the CLR Object Model, only single inheritance is supported), the compiler injects a call to its finalizer following execution of the user-supplied code. For example, consider the following simple hierarchy taken from the Managed Extensions language specification:

```
__gc class A {
public:
   ~A() { Console::WriteLine(S"in ~A"); }
};

__gc class B : public A {
public:
   ~B() { Console::WriteLine(S"in ~B");  }
};
```

In this example, both destructors are renamed `Finalize`. `B`'s `Finalize` has an invocation of `A`'s `Finalize` method added following the invocation of `WriteLine`. This is what the garbage collector will by default invoke during finalization. Here is what this internal transformation might look like:

```
// internal transformation of destructor under Managed Extensions
__gc class A {
public:
   void Finalize() { Console::WriteLine(S"in ~A"); }
};

__gc class B : public A {
public:
   void Finalize() {
      Console::WriteLine(S"in ~B");
      A::Finalize();
   }
```

```
};
```

1. In the second step, the compiler synthesizes a virtual destructor. This destructor is what our Managed Extensions user programs invoke either directly or through an application of the delete expression. It is never invoked by the garbage collector.

   Two statements are placed within this synthesized destructor. One is a call to `GC::SuppressFinalize` to make sure that there are no more invocations of `Finalize`. The second is the actual invocation of `Finalize`, which represents the user-supplied destructor for that class. Here is what this might look like:

```
__gc class A {
public:
   virtual ~A() {
      System::GC::SuppressFinalize(this);
      A::Finalize();
   }
};

__gc class B : public A {
public:
   virtual ~B() {
      System::GC::SuppressFinalize(this);
      B::Finalize();
   }
};
```

While this implementation allows the user to explicitly invoke the class `Finalize` method now rather than at a time you have no control over, it does not really tie in with the `Dispose` method solution. This is changed in Visual C++ 2005.

Destructors in New Syntax

In the new syntax, the destructor is renamed internally to the `Dispose` method and the reference class is automatically extended to implement the `IDispose` interface. That is, under Visual C++ 2005, our pair of classes is transformed as follows:

```
// internal transformation of destructor under the new syntax
__gc class A : IDisposable {
public:
   void Dispose() {
      System::GC::SuppressFinalize(this);
      Console::WriteLine( "in ~A");
   }
};

__gc class B : public A {
public:
   void Dispose() {
      System::GC::SuppressFinalize(this);
      Console::WriteLine( "in ~B");
      A::Dispose();
   }
};
```

When either a destructor is invoked explicitly under the new syntax, or when `delete` is applied to a tracking handle, the underlying `Dispose` method is invoked automatically. If it is a derived class, a call of the `Dispose` method of the base class is inserted at the close of the synthesized method.

But this does not get us all the way to deterministic finalization. In order to reach that, we need the additional support of local reference objects. (This has no analogous support within Managed Extensions, and so it is not a translation issue.)

Declaring a Reference Object

Visual C++ 2005 supports the declaration of an object of a reference class on the local stack or as a member of a class as if it were directly accessible. When combined with the association of the destructor with the `Dispose` method, the result is the automated invocation of finalization semantics on reference types.

First, we define our reference class such that object creation functions as the acquisition of a resource through its class constructor. Secondly, within the class destructor, we release the resource acquired when the object was created.

```
public ref class R {
public:
   R() { /* acquire expensive resource */ }
   ~R() { /* release expensive resource */ }

   // … everything else …
};
```

The object is declared locally by using the type name but without the accompanying hat. All uses of the object, such as invoking a method, are done through the member selection dot (**.**) instead of arrow (**->**). At the end of the block, the associated destructor, transformed into `Dispose`, is invoked automatically, as shown here:

```
void f() {
   R r;
   r.methodCall();

   // r is automatically destructed here –
   // that is, r.Dispose() is invoked
}
```

As with the **using** statement within C#, this does not defy the underlying CLR constraint that all reference types must be allocated on the CLR heap. The underlying semantics remain unchanged. The user could equivalently have written the following (and this is likely the internal transformation performed by the compiler):

```
// equivalent implementation
// except that it should be in a try/finally clause
void f() {
   R^ r = gcnew R;
   r->methodCall();

   delete r;
}
```

In effect, under the new syntax, destructors are again paired with constructors as an automated acquisition/release mechanism tied to a local object's lifetime.

Declaring an Explicit Finalize

In the new syntax, as we've seen, the destructor is synthesized into the `Dispose` method. This means that when the destructor is not explicitly invoked, the garbage collector, during finalization, will not as before find an associated `Finalize` method for the object. To support both destruction and finalization, we have introduced a special syntax for providing a finalizer. For example:

```
public ref class R {
public:
   !R() { Console::WriteLine( "I am the R::finalizer()!" ); }
};
```

The **!** prefix is analogous to tilde (**~**) that introduces a class destructor – that is, both post-lifetime methods have a token prefixing the name of the class. If the synthesized `Finalize` method occurs within a derived class, an invocation of the base class `Finalize` method is inserted at its end. If the destructor is explicitly invoked, the finalizer is suppressed. Here is what the transformation might look like:

```
// internal transformation under new syntax
public ref class R {
public:
   void Finalize() {
      Console::WriteLine( "I am the R::finalizer()!" );
   }
};
```

Moving from Managed Extensions for C++ to Visual C++ 2005

The runtime behavior of a Managed Extensions for C++ program is changed when it is compiled under Visual C++ 2005 whenever a reference class contains a non-trivial destructor. The required translation algorithm is similar to the following:

1. If a destructor is present, rewrite that to be the class finalizer.
2. If a `Dispose` method is present, rewrite that into the class destructor.
3. If a destructor is present but there is no `Dispose` method, retain the destructor while performing the first item.

In moving your code from Managed Extensions to the new syntax, you might miss performing this transformation. If the application depended in some way on the execution of associated finalization methods, the behavior of the application will silently differ from the one you intended.

See Also

Reference

Destructors and Finalizers in Visual C++

Concepts

The Managed Types

Member Declarations within a Class or Interface

The declaration of properties and operators has been extensively reworked from Managed Extensions for C++ to Visual C++ 2005, hiding the underlying implementation details that were exposed in the Managed Extensions design. Event declarations have been modified as well.

Under the category of changes that have no Managed Extensions support, static constructors can now be defined out-of-line (they were required to be defined inline within Managed Extensions), and the notion of a delegating constructor has been introduced.

In This Section

Property Declaration

Discusses changes to property declarations.

Property Index Declaration

Discusses changes to indexed property declarations.

Delegates and Events

Discusses changes to the syntax for declaring delegates and events.

Sealing a Virtual Function

Discusses changes to the syntax for sealing a function.

Overloaded Operators

Discusses changes to operator overloading.

Changes to Conversion Operators

Discusses changes to conversion operators.

Explicit Override of an Interface Member

Discusses changes to the method for explicitly overriding an interface member.

Private Virtual Functions

Discusses changes in the way private virtual functions are handled in derived classes.

Static Const Int Linkage Is No Longer Literal

Discusses changes in the way **static const** integral members are linked and how to explicitly declare a constant using the new **literal** keyword.

See Also

Concepts

C++/CLI Migration Primer

Property Declaration

The way to declare a property in a managed class has changed from Managed Extensions for C++ to Visual C++ 2005.

In the Managed Extensions design, each `set` or `get` property accessor is specified as an independent method. The declaration of each method is prefixed with the **__property** keyword. The method name begins with either `set_` or `get_` followed by the actual name of the property (as visible to the user). Thus, a `Vector` providing an `x` coordinate `get` property would name it `get_x` and the user would invoke it as `x`. This naming convention and separate specification of methods actually reflects the underlying runtime implementation of the property. For example, here is our `Vector` with a set of coordinate properties:

```
public __gc __sealed class Vector {
public:
   __property double get_x(){ return _x; }
   __property double get_y(){ return _y; }
   __property double get_z(){ return _z; }

   __property void set_x( double newx ){ _x = newx; }
   __property void set_y( double newy ){ _y = newy; }
   __property void set_z( double newz ){ _z = newz; }
};
```

This spreads out the functionality associated with a property and requires the user to lexically unify the associated sets and gets. Moreover, it is verbose. In the new syntax, which is more like that of C#, the **property** keyword is followed by the type of the property and its unadorned name. The `set` and `get` access methods are placed within a block following the property name. Note that unlike C#, the signature of the access method is specified. For example, here is the code example above translated into the new syntax.

```
public ref class Vector sealed {
public:
   property double x {
      double get() {
         return _x;
      }

      void set( double newx ) {
         _x = newx;
      }
   } // Note: no semi-colon
};
```

If the access methods of the property reflect distinct access levels – such as a **public** `get` and a **private** or **protected** `set`, an explicit access label can be specified. By default, the access level of the property reflects that of the enclosing access level. For example, in the above definition of `Vector`, both the `get` and `set` methods are **public**. To make the `set` method **protected** or **private**, the definition would be revised as follows:

```
public ref class Vector sealed {
public:
   property double x {
      double get() {
         return _x;
      }

   private:
      void set( double newx ) {
         _x = newx;
      }

   } // note: extent of private culminates here …

// note: dot is a public method of Vector
double dot( const Vector^ wv );
```

```
// etc.
};
```

The scope of an access keyword within a property extends until either the closing brace of the property or the specification of an additional access keyword. It does not extend beyond the definition of the property to the enclosing access level within which the property is defined. In the above declaration, for example, `Vector::dot()` is a public method.

Writing the set/get properties for the three `Vector` coordinates involves three steps:

1. declare a private state member of the appropriate type.
2. return it when the user wishes to get its value.
3. assign it the new value.

In the new syntax, a shorthand property syntax is available which automates this usage pattern:

```
public ref class Vector sealed {
public:
   // equivalent shorthand property syntax
   property double x;
   property double y;
   property double z;
};
```

The interesting side effect of the shorthand property syntax is that although the backstage state member is generated by the compiler, it is not accessible within the class except through the set/get accessors.

See Also

Reference

property

Concepts

Member Declarations within a Class or Interface

Property Index Declaration

The syntax for declaring an indexed property has changed from Managed Extensions for C++ to Visual C++ 2005.

The two primary shortcoming of the Managed Extensions language support of indexed properties is the inability to provide class-level subscripting; that is, all indexed properties are required to be given a name, and thus there is no way, for example, to provide a managed subscript operator that can be directly applied to a `Vector` or `Matrix` class object. A second less significant shortcoming is that it is visually difficult to distinguish a property from an indexed property – the number of parameters is the only indication. Finally, indexed properties suffer from the same problems as those of non-indexed properties – the accessors are not treated as an atomic unit, but separated into individual methods. For example:

```
public __gc class Vector;
public __gc class Matrix {
   float mat[,];

public:
   __property void set_Item( int r, int c, float value);
   __property float get_Item( int r, int c );

   __property void set_Row( int r, Vector* value );
   __property Vector* get_Row( int r );
};
```

As you can see here, the indexers are distinguished only by the additional parameters to specify a two or single dimension index. In the new syntax, the indexers are distinguished by the bracket ([,]) following the name of the indexer and indicating the number and type of each index:

```
public ref class Vector {};
public ref class Matrix {
private:
   array<float, 2>^ mat;

public:
   property float Item [int,int] {
      float get( int r, int c );
      void set( int r, int c, float value );
   }

   property Vector^ Row [int] {
      Vector^ get( int r );
      void set( int r, Vector^ value );
   }
};
```

To indicate a class level indexer that can be applied directly to objects of the class in the new syntax, the **default** keyword is reused to substitute for an explicit name. For example:

```
public ref class Matrix {
private:
   array<float, 2>^ mat;

public:
   // ok: class level indexer now
   //
   //     Matrix mat …
   //     mat[ 0, 0 ] = 1;
   //
   // invokes the set accessor of the default indexer …

   property float default [int,int] {
      float get( int r, int c );
      void set( int r, int c, float value );
   }
```

```
    property Vector^ Row [int] {
       Vector^ get( int r );
       void set( int r, Vector^ value );
    }
 };
```

In the new syntax, when the default indexed property is specified, the two following names are reserved: `get_Item` and `set_Item`. This is because these are the underlying names generated for the default indexed property.

Note that there is no simple index syntax analogous to the simple property syntax.

See Also

Concepts

Member Declarations within a Class or Interface
How to: Use Indexed Properties

Delegates and Events

The way to declare delegates and events has changed from Managed Extensions for C++ to Visual C++ 2005.

The double underscore is no longer needed, as shown in the following sample. Here a sample code in Managed Extensions:

```
__delegate void ClickEventHandler(int, double);
__delegate void DblClickEventHandler(String*);

__gc class EventSource {
   __event ClickEventHandler* OnClick;
   __event DblClickEventHandler* OnDblClick;
};
```

The same code in the new syntax looks as follows:

```
delegate void ClickEventHandler( int, double );
delegate void DblClickEventHandler( String^ );

ref class EventSource {
   event ClickEventHandler^ OnClick;
   event DblClickEventHandler^ OnDblClick;
};
```

Events (and delegates) are reference types, which is clear in the new syntax because of the use of the hat (**^**). Events support both an explicit declaration syntax and the trivial form shown in the preceding code. In the explicit form, the user specifies the `add`, `raise`, and `remove` methods associated with the event. (Only the `add` and `remove` methods are required; the `raise` method is optional.)

Under Managed Extensions, if you provide these methods, you do not also provide an explicit event declaration, but you must decide on a name for the event that is not present. Each method is specified in the form `add_EventName`, `raise_EventName`, and `remove_EventName`, as in the following example taken from the Managed Extensions specification:

```
// explicit implementations of add, remove, raise
public __delegate void f(int);
public __gc struct E {
   f* _E;
public:
   E() { _E = 0; }

   __event void add_E1(f* d) { _E += d; }

   static void Go() {
      E* pE = new E;
      pE->E1 += new f(pE, &E::handler);
      pE->E1(17);
      pE->E1 -= new f(pE, &E::handler);
      pE->E1(17);
   }

private:
   __event void raise_E1(int i) {
      if (_E)
         _E(i);
   }

protected:
   __event void remove_E1(f* d) {
      _E -= d;
   }
};
```

The new syntax simplifies the declaration, as the following translation demonstrates. An event specifies the two or three

methods enclosed in a pair of braces and placed immediately after the declaration of the event and its associated delegate type, as shown here:

```
public delegate void f( int );
public ref struct E {
private:
   f^ _E; // delegates are also reference types

public:
   E() {  // note the replacement of 0 with nullptr!
      _E = nullptr;
   }

   // the new aggregate syntax of an explicit event declaration
   event f^ E1 {
   public:
      void add( f^ d ) {
         _E += d;
      }

   protected:
      void remove( f^ d ) {
         _E -= d;
      }

   private:
      void raise( int i ) {
         if ( _E )
            _E( i );
      }
   }

   static void Go() {
      E^ pE = gcnew E;
      pE->E1 += gcnew f( pE, &E::handler );
      pE->E1( 17 );
      pE->E1 -= gcnew f( pE, &E::handler );
      pE->E1( 17 );
   }
};
```

See Also

Reference

delegate

event (Visual C++)

Concepts

Member Declarations within a Class or Interface

Sealing a Virtual Function

The syntax for sealing a virtual function has changed from Managed Extensions for C++ to Visual C++ 2005.

The **__sealed** keyword is used in Managed Extensions to modify either a reference type, disallowing subsequent derivation from it (see Declaration of a Managed Class Type), or to modify a virtual function, disallowing subsequent overriding of the method in a derived class. For example:

```
__gc class base { public: virtual void f(); };
__gc class derived : public base {
public:
   __sealed void f();
};
```

In this example, `derived::f()` overrides the `base::f()` instance based on the exact match of the function prototype. The **__sealed** keyword indicates that a subsequent class inherited from the derived class cannot provide an override of `derived::f()`.

In the new syntax, **sealed** is placed after the signature rather than being allowed to appear anywhere before the actual function prototype, as was previously allowed. In addition, the use of **sealed** requires an explicit use of the **virtual** keyword as well. That is, the correct translation of `derived`, above, is as follows:

```
ref class derived: public base {
public:
   virtual void f() override sealed;
};
```

The absence of the **virtual** keyword in this instance results in an error. In the new syntax, the contextual keyword **abstract** can be used in place of the **=0** to indicate a pure virtual function. This was not supported within Managed Extensions. For example:

```
__gc class base { public: virtual void f()=0; };
```

can be rewritten as

```
ref class base { public: virtual void f() abstract; };
```

See Also

Reference

sealed

Concepts

Member Declarations within a Class or Interface

Overloaded Operators

Operator overloading has changed significantly from Managed Extensions for C++ to Visual C++ 2005.

In the declaration of a reference type, for example, rather than using the native `operator+` syntax, you explicitly write out the underlying internal name of the operator – in this case, `op_Addition`. In addition, the invocation of an operator has to be explicitly invoked through that name, thus precluding the two primary benefits of operator overloading: (a) the intuitive syntax, and (b) the ability to intermix new types with existing types. For example:

```
public __gc __sealed class Vector {
public:
   Vector( double x, double y, double z );

   static bool    op_Equality( const Vector*, const Vector* );
   static Vector* op_Division( const Vector*, double );
   static Vector* op_Addition( const Vector*, const Vector* );
   static Vector* op_Subtraction( const Vector*, const Vector* );
};

int main()
{
   Vector *pa = new Vector( 0.231, 2.4745, 0.023 );
   Vector *pb = new Vector( 1.475, 4.8916, -1.23 );

   Vector *pc1 = Vector::op_Addition( pa, pb );
   Vector *pc2 = Vector::op_Subtraction( pa, pc1 );
   Vector *pc3 = Vector::op_Division( pc1, pc2->x );

   if ( Vector::op_Equality( pc1, pc2 ))
      ;
}
```

In the new syntax, the usual expectations of a native C++ programmer are restored, both in the declaration and use of the static operators. Here is the `Vector` class translated into the new syntax:

```
public ref class Vector sealed {
public:
   Vector( double x, double y, double z );

   static bool    operator ==( const Vector^, const Vector^ );
   static Vector^ operator /( const Vector^, double );
   static Vector^ operator +( const Vector^, const Vector^ );
   static Vector^ operator -( const Vector^, const Vector^ );
};

int main()
{
   Vector^ pa = gcnew Vector( 0.231, 2.4745, 0.023 );
   Vector^ pb = gcnew Vector( 1.475,4.8916,-1.23 );

   Vector^ pc1 = pa + pb;
   Vector^ pc2 = pa - pc1;
   Vector^ pc3 = pc1 / pc2->x;

   if ( pc1 == pc2 )
      ;
}
```

See Also

Concepts

Member Declarations within a Class or Interface

Changes to Conversion Operators

The syntax for conversion operators has changed from Managed Extensions for C++ to Visual C++ 2005.

One example is to write `op_Implicit` to specify a conversion. Here is a definition of `MyDouble` taken from the language specification:

```
__gc struct MyDouble {
   static MyDouble* op_Implicit( int i );
   static int op_Explicit( MyDouble* val );
   static String* op_Explicit( MyDouble* val );
};
```

This says that, given an integer, the algorithm for converting that integer into a `MyDouble` is provided by the `op_Implicit` operator. Moreover, that conversion will be performed implicitly by the compiler. Similarly, given a `MyDouble` object, the two `op_Explicit` operators provide the respective algorithms for converting that object into either an integer or a managed `String` entity. However, the compiler will not perform the conversion unless explicitly requested by the user.

In C#, this looks as follows:

```
class MyDouble {
   public static implicit operator MyDouble( int i );
   public static explicit operator int( MyDouble val );
   public static explicit operator string( MyDouble val );
};
```

The C# code looks more like C++ than Managed Extensions for C++ does. That is not the case in the new syntax.

The ISO-C++ committee introduced a keyword, **explicit**, to mitigate unintended consequences – for example, an `Array` class which takes a single integer argument as a dimension will implicitly convert any integer into an `Array` object which is not what you want. One way to prevent this is a design idiom of a dummy second argument to a constructor

On the other hand, you should not provide a conversion pair when designing a class type within C++. The best example for that is the standard string class. The implicit conversion is the single-argument constructor taking a C-style string. However, it does not provide the corresponding implicit conversion operator – that of converting a string object to a C-style string, but rather requires the user to explicitly invoke a named function – in this case, `c_str()`.

So, associating an implicit/explicit behavior on a conversion operator (and as encapsulating the set of conversions to a single form of declaration) appears to be an improvement on the original C++ support for conversion operators, which eventually led to the **explicit** keyword. The Visual C++ 2005 language support for conversion operators looks as follows, which is slightly less verbose than that of C# because of the default behavior of the operator supporting an implicit application of the conversion algorithm:

```
ref struct MyDouble {
public:
   static operator MyDouble^ ( int i );
   static explicit operator int ( MyDouble^ val );
   static explicit operator String^ ( MyDouble^ val );
};
```

Another change is that a single argument constructor is treated as if it is declared as **explicit**. This means that in order to trigger its invocations, an explicit cast is required. Note, however, that if an explicit conversion operator is defined, it and not the single-argument constructor, is invoked.

See Also

Concepts

Member Declarations within a Class or Interface

Explicit Override of an Interface Member

The syntax for declaring an explicit override of an interface member within a class has changed from Managed Extensions for C++ to Visual C++ 2005.

You often want to provide two instances of an interface member within a class that implements the interface – one that is used when class objects are manipulated through an interface handle, and one that is used when class objects are used through the class interface. For example:

```
public __gc class R : public ICloneable {
   // to be used through ICloneable
   Object* ICloneable::Clone();

   // to be used through an R
   R* Clone();
};
```

In Managed Extensions we do this by providing an explicit declaration of the interface method with the method's name qualified with the name of the interface. The class-specific instance is unqualified. This eliminates the need to downcast the return value of `Clone`, in this example, when explicit called through an instance of `R`.

In the new syntax, a general overriding mechanism has been introduced that replaces the Managed Extensions syntax. Our example would be rewritten as follows:

```
public ref class R : public ICloneable {
public:
   // to be used through ICloneable
   virtual Object^ InterfaceClone() = ICloneable::Clone;

   // to be used through an R
   virtual R^ Clone();
};
```

This revision requires that the interface member being explicitly overridden be given a unique name within the class. Here, I've provided the awkward name of `InterfaceClone`. The behavior is still the same – an invocation through the `ICloneable` interface invokes the renamed `InterfaceClone`, while a call through an object of type `R` invokes the second `Clone` instance.

See Also

Reference

Explicit Overrides

Concepts

Member Declarations within a Class or Interface

Private Virtual Functions

The way private virtual functions are handled in derived classes has changed from Managed Extensions for C++ to Visual C++ 2005.

In Managed Extensions, the access level of a virtual function does not constrain its ability to be overridden within a derived class. In the new syntax, a virtual function cannot override a base class virtual function that it cannot access. For example:

```
__gc class Foo {
   // inaccessible to a derived class
   virtual void g();
};

__gc class Bar : public Foo {
public:
   // in Managed Extensions, ok: g() overrides Foo::g()
   // in new syntax, error: cannot override: Foo::g() is inaccessible …
   void g();
};
```

There is no real mapping of this sort of design onto the new syntax. One simply has to make the base class members accessible – that is, non-private. The inherited methods do not have to bear the same access. In this example, the least invasive change is to make the `Foo` member **protected**. This way the general program's access to the method through `Foo` is still prohibited,

```
ref class Foo {
protected:
   virtual void g();
};

ref class Bar : Foo {
public:
   virtual void g() override;
};
```

Note that the absence of the explicit **virtual** keyword in the base class, under the new syntax, generates a warning message.

See Also

Reference

Member Visibility

Concepts

Member Declarations within a Class or Interface

Static Const Int Linkage Is No Longer Literal

Declaration of a constant member of a class has changed from Managed Extensions for C++ to Visual C++ 2005.

Although **static const** integral members are still supported, their linkage attribute has changed. Their former linkage attribute is now carried in a literal integral member. For example, consider the following Managed Extensions class:

```
public __gc class Constants {
public:
   static const int LOG_DEBUG = 4;
};
```

This generates the following underlying CIL attributes for the field (note the literal attribute):

```
.field public static literal int32
modopt([Microsoft.VisualC]Microsoft.VisualC.IsConstModifier) STANDARD_CLIENT_PRX = int32(0x
00000004)
```

While this still compiles under the new syntax:

```
public ref class Constants {
public:
   static const int LOG_DEBUG = 4;
};
```

it no longer emits the literal attribute, and therefore is not viewed as a constant by the CLR runtime:

```
.field public static int32 modopt([Microsoft.VisualC]Microsoft.VisualC.IsConstModifier) STA
NDARD_CLIENT_PRX = int32(0x00000004)
```

In order to have the same inter-language literal attribute, the declaration should be changed to the newly supported **literal** data member, as follows,

```
public ref class Constants {
public:
   literal int LOG_DEBUG = 4;
};
```

See Also

Reference

literal (Visual C++)

Concepts

Member Declarations within a Class or Interface

Value Types and Their Behaviors

Value types have changed in various ways from Managed Extensions for C++ to Visual C++ 2005. In this section, we look at the CLR enum type and the value class type, together with a look at boxing and access to the boxed instance on the CLR heap, as well as a look at interior and pinning pointers. There have been extensive language changes in this area.

In This Section

Value Types and Their Behaviors

Discusses changes in the declaration and behavior of enums.

Implicit Boxing of Value Types

Discusses the motivation for implicit boxing of value types and the consequent changes in behavior.

A Tracking Handle to a Boxed Value

Discusses how implicit boxing of value types translates to a tracking handle to the boxed value object.

Value Type Semantics

Discusses changes to value type semantics, including inherited virtual methods, class default constructors, interior pointers, and pinning pointers.

See Also

Reference

Classes and Structs (Managed)

Concepts

C++/CLI Migration Primer

CLR Enum Type

The declaration and behavior of enums has changed from Managed Extensions for C++ to Visual C++ 2005.

The Managed Extensions enum declaration is preceded by the **__value** keyword. The idea here is to distinguish the native enum from the CLR enum which is derived from `System::ValueType`, while suggesting an analogous functionality. For example:

```
__value enum e1 { fail, pass };
public __value enum e2 : unsigned short  {
   not_ok = 1024,
   maybe, ok = 2048
};
```

The new syntax solves the problem of distinguishing native and CLR enums by emphasizing the class nature of the latter rather than its value type roots. As such, the **__value** keyword is discarded, replaced with the spaced keyword pair of **enum class**. This provides a paired keyword symmetry to the declarations of the reference, value, and interface classes:

```
enum class ec;
value class vc;
ref class rc;
interface class ic;
```

The translation of the enumeration pair `e1` and `e2` in the new syntax looks as follows:

```
enum class e1 { fail, pass };
public enum class e2 : unsigned short {
   not_ok = 1024,
   maybe, ok = 2048
};
```

Apart from this small syntactic change, the behavior of the CLR enum type has been changed in a number of ways:

- A forward declaration of a CLR enum is no longer supported. There is no mapping. It is simply flagged as a compile-time error.

```
__value enum status; // Managed Extensions: ok
enum class status;   // new syntax: error
```

- The overload resolution between the built-in arithmetic types and the `Object` class hierarchy has reversed between the two language versions! As a side-effect, CLR enums are no longer implicitly converted to arithmetic types.
- In the new syntax, a CLR enum maintains its own scope, which is not the case in Managed Extensions. Previously, the enumerators were visible within the containing scope of the enum. Now, the enumerators are encapsulated within the scope of the enum.

CLR Enums are a Kind of Object

Consider the following code fragment:

```
__value enum status { fail, pass };

void f( Object* ){ Console::WriteLine("f(Object)\n"); }
void f( int ){ Console::WriteLine("f(int)\n"); }

int main()
{
   status rslt = fail;

   f( rslt ); // which f is invoked?
```

```
}
```

For the native C++ programmer, the natural answer to the question of which instance of the overloaded `f` is invoked is that of `f(int)`. An enum is a symbolic integral constant, and it participates in the standard integral promotions which take precedence in this case. And in fact in Managed Extensions this was the instance to which the call resolves. This caused a number of surprises – not when we used them in a native C++ frame of mind – but when we needed them to interact with the existing BCL (Base Class Library) framework, where an `Enum` is a class indirectly derived from `Object`. In the Visual C++ 2005 language design, the instance of `f` invoked is that of `f(Object^)`.

The way Visual C++ 2005 has chosen to enforce this is to not support implicit conversions between a CLR enum type and the arithmetic types. This means that any assignment of an object of a CLR enum type to an arithmetic type will require an explicit cast. So, for example, given

```
void f( int );
```

as a non-overloaded method, in Managed Extensions, the call

```
f( rslt ); // ok: Managed Extensions; error: new syntax
```

is ok, and the value contained within `rslt` is implicitly converted into an integer value. In Visual C++ 2005, this call fails to compile. To correctly translate it, we must insert a conversion operator:

```
f( safe_cast<int>( rslt )); // ok: new syntax
```

The Scope of the CLR Enum Type

One of the changes between the C and C++ languages was the addition in C++ of scope within the struct facility. In C, a struct is just a data aggregate without support of either an interface or an associated scope. This was quite a radical change at the time and was a contentious issue for many new C++ users coming from the C language. The relationship between the native and CLR enum is analogous.

In Managed Extensions, an attempt was made to define weakly injected names for the enumerators of a CLR enum in order to simulate the absence of scope within the native enum. This did not prove successful. The problem is that this causes the enumerators to spill into the global namespace, resulting in difficult to manage name-collisions. In the new syntax, we have conformed to the other CLR languages in supporting scopes within the CLR enum.

This means that any unqualified use of an enumerator of a CLR enum will not be recognized by the new syntax. Let's look at a real-world example.

```
// Managed Extensions supporting weak injection
__gc class XDCMake {
public:
   __value enum _recognizerEnum {
      UNDEFINED,
      OPTION_USAGE,
      XDC0001_ERR_PATH_DOES_NOT_EXIST = 1,
      XDC0002_ERR_CANNOT_WRITE_TO = 2,
      XDC0003_ERR_INCLUDE_TAGS_NOT_SUPPORTED = 3,
      XDC0004_WRN_XML_LOAD_FAILURE = 4,
      XDC0006_WRN_NONEXISTENT_FILES = 6,
   };

   ListDictionary* optionList;
   ListDictionary* itagList;

   XDCMake() {
      optionList = new ListDictionary;

      // here are the problems …
      optionList->Add(S"?", __box(OPTION_USAGE)); // (1)
      optionList->Add(S"help", __box(OPTION_USAGE)); // (2)

      itagList = new ListDictionary;
```

```
      itagList->Add(S"returns",
         __box(XDC0004_WRN_XML_LOAD_FAILURE)); // (3)
   }
};
```

Each of the three unqualified uses of the enumerator names (*(1)*, *(2)*, and *(3)*) will need to be qualified in the translation to the new syntax in order for the source code to compile. Here is a correct translation of the original source code:

```
ref class XDCMake {
public:
   enum class _recognizerEnum {
      UNDEFINED, OPTION_USAGE,
      XDC0001_ERR_PATH_DOES_NOT_EXIST = 1,
      XDC0002_ERR_CANNOT_WRITE_TO = 2,
      XDC0003_ERR_INCLUDE_TAGS_NOT_SUPPORTED = 3,
      XDC0004_WRN_XML_LOAD_FAILURE = 4,
      XDC0006_WRN_NONEXISTENT_FILES = 6
   };

   ListDictionary^ optionList;
   ListDictionary^ itagList;

   XDCMake() {
      optionList = gcnew ListDictionary;
      optionList->Add("?",_recognizerEnum::OPTION_USAGE); // (1)
      optionList->Add("help",_recognizerEnum::OPTION_USAGE); //(2)
      itagList = gcnew ListDictionary;
      itagList->Add( "returns",
         _recognizerEnum::XDC0004_WRN_XML_LOAD_FAILURE); //(3)
   }
};
```

This changes the design strategy between a native and a CLR enum. With a CLR enum maintaining an associated scope in Visual C++ 2005, it is neither necessary nor effective to encapsulate the declaration of the enum within a class. This idiom evolved around the time of cfront 2.0 within Bell Laboratories also in order to solve the global name pollution problem.

In the original beta release of the new iostream library by Jerry Schwarz at Bell Laboratories, Jerry did not encapsulate all the associated enums defined for the library, and the common enumerators such as `read`, `write`, `append`, and so on, made it nearly impossible for users to compile their existing code. One solution would have been to mangle the names, such as `io_read`, `io_write`, etc. A second solution would have been to modify the language by adding scope to an enum, but this was not practicable at the time. The middle solution was to encapsulate the enum within the class, or class hierarchy, where both the tag name and enumerators of the enum populate the enclosing class scope.) That is, the motivation for placing enums within classes, at least originally, was not philosophical, but a practical response to the global name-space pollution problem.

With the Visual C++ 2005 enum, there is no longer any compelling benefit to encapsulating an enum within a class. In fact, if you look at the `System` namespaces, you will see that enums, classes, and interfaces all inhabit the same declaration space.

See Also

Reference

enum class

Concepts

Value Types and Their Behaviors

Implicit Boxing of Value Types

The boxing of value types has changed from Managed Extensions for C++ to Visual C++ 2005.

In language design, we imposed a philosophical position in lieu of practical experience with the feature and, in practice, it was a mistake. As an analogy, in the original multiple inheritance language design, Stroustrup decided that a virtual base class sub-object could not be initialized within a derived class constructor, and therefore the language required that any class serving as a virtual base class must define a default constructor. It is that default constructor that would be invoked by any subsequent virtual derivation.

The problem of a virtual base class hierarchy is that responsibility for the initialization of the shared virtual sub-object shifts with each subsequent derivation. For example, if I define a base class for which initialization requires the allocation of a buffer, the user-specified size of that buffer might be passed as an argument to the constructor. If I then provide two subsequent virtual derivations, call them `inputb` and `outputb`, each provides a particular value to the base class constructor. Now, when I derived an `in_out` class from both `inputb` and `outputb`, neither of those values to the shared virtual base class sub-object can sensibly be allowed to evaluate.

Therefore, in the original language design, Stroustrup disallowed the explicit initialization of a virtual base class within the member initialization list of the derived class constructor. While this solved the problem, in practice the inability to direct the initialization of the virtual base class proved impracticable. Keith Gorlen of the National Institute of Health, who had implemented a freeware version of the SmallTalk collection library called nihcl, was a principle voice in convincing Stroustrup that he had to come up with a more flexible language design.

A principle of Object-Oriented hierarchical design holds that a derived class should concern itself only with the non-private implementation of its immediate base classes. In order to support a flexible initialization design for virtual inheritance, Stroustrup had to violate this principle. The most derived class in a hierarchy assumes responsibility for all virtual sub-object initialization regardless of how deep into the hierarchy it occurs. For example, `inputb` and `outputb` are both responsible for explicitly initializing their immediate virtual base class. When `in_out` derives from both `inputb` and `outputb`, `in_out` becomes responsible for the initialization of the once removed virtual base class, and the initialization made explicit within `inputb` and `outputb` is suppressed.

This provides the flexibility required by language developers, but at the cost of a complicated semantics. This burden of complication is stripped away if we restrict a virtual base class to be without state and simply allow it to specify an interface. This is a recommended design idiom within C++. Within CLR programming, it is raised to policy with the Interface type.

Here is a simple code sample– and in this case, the explicit boxing is unnecessary:

```
// Managed Extensions for C++ requires explicit __box operation
int my1DIntArray __gc[] = { 1, 2, 3, 4, 5 };
Object* myObjArray __gc[] = {
   __box(26), __box(27), __box(28), __box(29), __box(30)
};

Console::WriteLine( "{0}\t{1}\t{2}", __box(0),
   __box(my1DIntArray->GetLowerBound(0)),
   __box(my1DIntArray->GetUpperBound(0)) );
```

As you can see, there is a whole lot of boxing going on. Under Visual C++ 2005, value type boxing is implicit:

```
// new syntax makes boxing implicit
array<int>^ my1DIntArray = {1,2,3,4,5};
array<Object^>^ myObjArray = {26,27,28,29,30};

Console::WriteLine( "{0}\t{1}\t{2}", 0,
   my1DIntArray->GetLowerBound( 0 ),
   my1DIntArray->GetUpperBound( 0 ) );
```

See Also

Reference

Implicit Boxing

Concepts

Value Types and Their Behaviors

A Tracking Handle to a Boxed Value

The usage of a tracking handle to reference a value type has changed from Managed Extensions for C++ to Visual C++ 2005.

Boxing is a peculiarity of the CLR unified type system. Value types directly contain their state, while reference types are an implicit pair: the named entity is a handle to an unnamed object allocated on the managed heap. Any initialization or assignment of a value type to an `Object`, for example, requires that the value type be placed within the CLR heap – this is where the image of boxing it arises – first by allocating the associated memory, then by copying the value type's state, and then returning the address of this anonymous Value/Reference hybrid. Thus, when one writes in C#

```
object o = 1024; // C# implicit boxing
```

there is a great deal more going on than is made apparent by the simplicity of the code. The design of C# hides the complexity not only of what operations are taking place under the hood, but also of the abstraction of boxing itself. Managed Extensions for C++, on the other hand, concerned that this would lead to a false sense of efficiency, puts it in the user's face by requiring an explicit instruction:

```
Object *o = __box( 1024 ); // Managed Extensions explicit boxing
```

Boxing is implicit in Visual C++ 2005:

```
Object ^o = 1024; // new syntax implicit boxing
```

The **__box** keyword serves a vital service within Managed Extensions, one that is absent by design from languages such as C# and Visual Basic: it provides both a vocabulary and tracking handle for directly manipulating a boxed instance on the managed heap. For example, consider the following small program:

```
int main() {
   double result = 3.14159;
   __box double * br = __box( result );

   result = 2.7;
   *br = 2.17;
   Object * o = br;

   Console::WriteLine( S"result :: {0}", result.ToString() ) ;
   Console::WriteLine( S"result :: {0}", __box(result) ) ;
   Console::WriteLine( S"result :: {0}", br );
}
```

The underlying code generated for the three invocations of `WriteLine` show the various costs of accessing the value of a boxed value type (thanks to Yves Dolce for pointing out these differences), where the indicated lines show the overhead associated with each invocation.

```
// Console::WriteLine( S"result :: {0}", result.ToString() ) ;
ldstr      "result :: {0}"
ldloca.s   result  // ToString overhead
call       instance string  [mscorlib]System.Double::ToString()  // ToString overhead
call       void [mscorlib]System.Console::WriteLine(string, object)

// Console::WriteLine( S"result :: {0}", __box(result) ) ;
Ldstr    " result :: {0}"
ldloc.0
box    [mscorlib]System.Double // box overhead
call    void [mscorlib]System.Console::WriteLine(string, object)


// Console::WriteLine( S"result :: {0}", br );
ldstr    "result :: {0}"
ldloc.0
```

```
call     void [mscorlib]System.Console::WriteLine(string, object)
```

Passing the boxed value type directly to `Console::WriteLine` eliminates both the boxing and the need to invoke `ToString()`. (Of course, there is the earlier boxing to initialize `br`, so we don't gain anything unless we really put `br` to work.

In the new syntax, the support for boxed value types is considerably more elegant and integrated within the type system while retaining its power. For example, here is the translation of the earlier small program:

```
int main()
{
   double result = 3.14159;
   double^ br = result;
   result = 2.7;
   *br = 2.17;
   Object^ o = br;
   Console::WriteLine( "result :: {0}", result.ToString() );
   Console::WriteLine( "result :: {0}", result );
   Console::WriteLine( "result :: {0}", br );
}
```

See Also

Tasks

How to: Explicitly Request Boxing

Concepts

Value Types and Their Behaviors

Value Type Semantics

Value type semantics have changed from Managed Extensions for C++ to Visual C++ 2005.

Here is the canonical simple value type used in the Managed Extensions for C++ spec:

```
__value struct V { int i; };
__gc struct R { V vr; };
```

In Managed Extensions, we can have four syntactic variants of a value type (where forms 2 and 3 are the same semantically):

```
V v = { 0 };       // Form (1)
V *pv = 0;         // Form (2) an implicit form of (3)
V __gc *pvgc = 0;  // Form (3)
__box V* pvbx = 0; // Form (4) must be local
```

Invoking Inherited Virtual Methods

Form (1) is the canonical value object, and it is reasonably well understood, except when someone attempts to invoke an inherited virtual method such as `ToString()`. For example:

```
v.ToString(); // error!
```

In order to invoke this method, because it is not overridden in `V`, the compiler must have access to the associated virtual table of the base class. Because value types are in-state storage without the associated pointer to its virtual table (vptr), this requires that `v` be boxed. In the Managed Extensions language design, implicit boxing is not supported but must be explicitly specified by the programmer, as in

```
__box( v )->ToString(); // Managed Extensions: note the arrow
```

The primary motive behind this design is pedagogical: underlying mechanism needs to be visible to the programmer so that she will understand the 'cost' of not providing an instance within her value type. Were `V` to contain an instance of `ToString`, the boxing would not be necessary.

The lexical complexity of explicitly boxing the object, but not the underlying cost of the boxing itself, is removed in the new syntax:

```
v.ToString(); // new syntax
```

but at the cost of possibly misleading the class designer as to the cost of not having provided an explicit instance of the `ToString` method within `V`. The reason for preferring the implicit boxing is that while there is usually just one class designer, there are an unlimited number of users, none of whom would have the freedom to modify `V` to eliminate the possibly onerous explicit box.

The criteria by which to determine whether or not to provide an overriding instance of `ToString` within a value class should be the frequency and location of its uses. If it is called very rarely, there is of course little benefit in its definition. Similarly, if it is called in non-performant areas of the application, adding it will also not measurably add to the general performance of the application. Alternatively, one can keep a tracking handle to the boxed value, and calls through that handle would not require boxing.

There is No Longer a Value Class Default Constructor

Another difference with a value type between Managed Extensions and the new syntax is the removal of support for a default constructor. This is because there are occasions during execution in which the CLR can create an instance of the value type without invoking the associated default constructor. That is, the attempt under Managed Extensions to support a default constructor within a value type could not in practice be guaranteed. Given that absence of guarantee, it was felt to be better to drop the support altogether rather than have it be non-deterministic in its application.

This is not as bad as it might initially seem. This is because each object of a value type is zeroed out automatically (that is, each type is initialized to its default value). As a result, the members of a local instance are never undefined. In this sense, the loss of

the ability to define a trivial default constructor is really not a loss at all – and in fact is more efficient when performed by the CLR.

The problem is when a user of Managed Extensions defines a non-trivial default constructor. This has no mapping to the new syntax. The code within the constructor will need to be migrated into a named initialization method that would then need to be explicitly invoked by the user.

The declaration of a value type object within the new syntax is otherwise unchanged. The down side of this is that value types are not satisfactory for the wrapping of native types for the following reasons:

- There is no support for a destructor within a value type. That is, there is no way to automate a set of actions triggered by the end of an object's lifetime.
- A native class can be contained only within a managed type as a pointer which is then allocated on the native heap.

We would like to wrap a small native class in a value type rather than a reference type to avoid a double heap allocation: the native heap to hold the native type, and the CLR heap to hold the managed wrapper. Wrapping a native class within a value type allows you to avoid the managed heap, but provides no way to automate the reclamation of the native heap memory. Reference types are the only practicable managed type within which to wrap non-trivial native classes.

Interior Pointers

Forms (2) and *(3)* above can address nearly anything in this world or the next (that is, anything managed or native). So, for example, all the following are permitted in Managed Extensions:

```
__value struct V { int i; };
__gc struct R { V vr; };

V v = { 0 };  // Form (1)
V *pv = 0;  // Form (2)
V __gc *pvgc = 0;  // Form (3)
__box V* pvbx = 0;  // Form (4)

R* r;

pv = &v;            // address a value type on the stack
pv = __nogc new V;  // address a value type on native heap
pv = pvgc;          // we are not sure what this addresses
pv = pvbx;          // address a boxed value type on managed heap
pv = &r->vr;        // an interior pointer to value type within a
                    //    reference type on the managed heap
```

So, a `V*` can address a location within a local block (and therefore can be dangling), at global scope, within the native heap (for example, if the object it addresses has already been deleted), within the CLR heap (and therefore will be tracked if it should be relocated during garbage collection), and within the interior of a reference object on the CLR heap (an interior pointer, as this is called, is also transparently tracked).

In Managed Extensions, there is no way to separate out the native aspects of a `V*`; that is, it is treated at its inclusive, which handles the possibility of it addressing an object or subobject on the managed heap.

In the new syntax, a value type pointer is factored into two types: `V*`, which is limited to non-CLR heap locations, and the interior pointer, `interior_ptr<V>`, which allows for but does not require an address within the managed heap.

```
// may not address within managed heap
V *pv = 0;

// may or may not address within managed heap
interior_ptr<V> pvgc = nullptr;
```

Forms (2) and *(3)* of Managed Extensions map into `interior_ptr<V>`. *Form (4)* is a tracking handle. It addresses the whole object that has been boxed within the managed heap. It is translated in the new syntax into a `V^`,

```
V^ pvbx = nullptr; // __box V* pvbx = 0;
```

The following declarations in Managed Extensions all map to interior pointers in the new syntax. (They are value types within

the `System` namespace.)

```
Int32 *pi;   // => interior_ptr<Int32> pi;
Boolean *pb; // => interior_ptr<Boolean> pb;
E *pe;       // => interior_ptr<E> pe; // Enumeration
```

The built-in types are not considered managed types, although they do serve as aliases to the types within the `System` namespace. Thus the following mappings hold true between Managed Extensions and the new syntax:

```
int * pi;     // => int* pi;
int __gc * pi2; // => interior_ptr<int> pi2;
```

When translating a `V*` in your existing program, the most conservative strategy is to always turn it to an `interior_ptr<V>`. This is how it was treated under Managed Extensions. In the new syntax, the programmer has the option of restricting a value type to non-managed heap addresses by specifying `V*` rather than an interior pointer. If, on translating your program, you can do a transitive closure of all its uses and be sure that no assigned address is within the managed heap, then leaving it as `V*` is fine.

Pinning Pointers

The garbage collector may optionally move objects that reside on the CLR heap to different locations within the heap, usually during a compaction phase. This movement is not a problem to tracking handles, tracking references, and interior pointers which update these entities transparently. This movement is a problem, however, if the user has passed the address of an object on the CLR heap outside of the runtime environment. In this case, the volatile movement of the object is likely to cause a runtime failure. To exempt objects such as these from being moved, we must locally pin them to their location for the extent of their outside use.

In Managed Extensions, a *pinning pointer* is declared by qualifying a pointer declaration with the **__pin** keyword. Here is an example slightly modified from the Managed Extensions specification:

```
__gc struct H { int j; };

int main()
{
   H * h = new H;
   int __pin * k = & h -> j;

   // …
};
```

In the new language design, a pinning pointer is declared with syntax analogous to that of an interior pointer.

```
ref struct H
{
public:
   int j;
};

int main()
{
   H^ h = gcnew H;
   pin_ptr<int> k = &h->j;

   // …
}
```

A pinning pointer under the new syntax is a special case of an interior pointer. The original constraints on a pinning pointer remain. For example, it cannot be used as a parameter or return type of a method; it can be declared only on a local object. A number of additional constraints, however, have been added in the new syntax.

The default value of a pinning pointer is `nullptr`, not `0`. A `pin_ptr<>` cannot be initialized or assigned `0`. All assignments of `0` in existing code will need to be changed to `nullptr`.

A pinning pointer under Managed Extensions was permitted to address a whole object, as in the following example taken from the Managed Extensions specification:

```
__gc class G {
public:
   void incr(int* pi) { pi += 1; }
};
__gc struct H { int j; };
void f( G * g ) {
   H __pin * pH = new H;
   g->incr(& pH -> j);
};
```

In the new syntax, pinning the whole object returned by the `new` expression is not supported. Rather, the address of the interior member needs to be pinned. For example,

```
ref class G {
public:
   void incr(int* pi) { *pi += 1; }
};
ref struct H { int j; };
void f( G^ g ) {
   H ^ph = gcnew H;
   Console::WriteLine(ph->j);
   pin_ptr<int> pj = &ph->j;
   g->incr(  pj );
   Console::WriteLine(ph->j);
}
```

See Also

Reference

Classes and Structs (Managed)

interior_ptr

pin_ptr

Concepts

Value Types and Their Behaviors

General Language Changes

A number of CLR language features changed from Managed Extensions for C++ to Visual C++ 2005.

The changes described in this section are a sort of language miscellany. It includes a change in the handling of string literals, a change in the overload resolution between an ellipsis and the `Param` attribute, the change of **typeof** to **typeid**, a change in the calling of constructor initializer lists, and the introduction of a new cast notation, that of **safe_cast**.

String Literal

Discusses how the handling of string literals has changed.

Param Array and Ellipsis

Discusses how `ParamArray` is now given precedence over the ellipsis (...) for resolving function calls with varying numbers of arguments.

typeof Goes to T::typeid

Discusses how the **typeof** operator has been supplanted by **typeid**.

Initializer Lists

Discusses changes in the calling order of initializer lists.

Cast Notation and Introduction of safe_cast<>

Discusses changes to cast notation and in particular the introduction of **safe_cast**.

See Also

Concepts

C++/CLI Migration Primer

String Literal

The handling of string literals has changed from Managed Extensions for C++ to Visual C++ 2005.

In the Managed Extensions for C++ language design, a managed string literal was indicated by prefacing the string literal with an `S`. For example:

```
String *ps1 = "hello";
String *ps2 = S"goodbye";
```

The performance overhead between the two initializations turns out to be non-trivial, as the following CIL representation demonstrates as seen through **ildasm**:

```
// String *ps1 = "hello";
ldsflda    valuetype $ArrayType$0xd61117dd
     modopt([Microsoft.VisualC]Microsoft.VisualC.IsConstModifier)
     '?A0xbdde7aca.unnamed-global-0'

newobj instance void [mscorlib]System.String::.ctor(int8*)
stloc.0

// String *ps2 = S"goodbye";
ldstr      "goodbye"
stloc.0
```

That's a remarkable savings for just remembering (or learning) to prefix a literal string with an `S`. In the new syntax, the handling of string literals is made transparent, determined by the context of use. The `S` no longer needs to be specified.

What about cases in which we need to explicitly direct the compiler to one interpretation or another? In these cases, we apply an explicit cast. For example:

```
f( safe_cast<String^>("ABC") );
```

Moreover, the string literal now matches a `String` with a simple conversion rather than a standard conversion. While this may not sound like much it changes the resolution of overloaded function sets which include a `String` and a `const char*` as competing formal parameters. The resolution that once resolved to a `const char*` instance is now flagged as ambiguous. For example:

```
ref struct R {
   void f(const char*);
   void f(String^);
};

int main () {
   R r;
   // old syntax: f( const char* );
   // new syntax: error: ambiguous
   r.f("ABC");
}
```

Why is there a difference? Since more than one instance named `f` exists within the program, this requires the function overload resolution algorithm to be applied to the call. The formal resolution of an overload function involves three steps.

1. The collection of the candidate functions. The candidate functions are those methods within the scope that lexically match the name of the function being invoked. For example, since `f()` is invoked through an instance of `R`, all named functions `f` that are not a member of `R` (or of its base class hierarchy) are not candidate functions. In our example, there are two candidate functions. These are the two member functions of `R` named `f`. A call fails during this phase if the candidate function set is empty.

2. The set of viable functions from among the candidate functions. A viable function is one that can be invoked with the

arguments specified in the call, given the number of arguments and their types. In our example, both candidate functions are also viable functions. A call fails during this phase if the viable function set is empty.

3. Select the function that represents the best match of the call. This is done by ranking the conversions applied to transform the arguments to the type of the viable function parameters. This is relatively straight-forward with a single parameter function; it becomes somewhat more complex when there are multiple parameters. A call fails during this phase if there is no best match. That is, if the conversions necessary to transform the type of the actual argument to the type of the formal parameter are equally good. The call is flagged as ambiguous.

In Managed Extensions, the resolution of this call invoked the `const char*` instance as the best match. In the new syntax, the conversion necessary to match `"abc"` to `const char*` and `String^` are now equivalent – that is, equally good – and so the call is flagged as bad – that is, as ambiguous.

This leads us to two questions:

- What is the type of the actual argument, `"abc"`?
- What is the algorithm for determining when one type conversion is better than another?

The type of the string literal `"abc"` is `const char[4]` – remember, there is an implicit null terminating character at the end of every string literal.

The algorithm for determining when one type conversion is better than another involves placing the possible type conversions in a hierarchy. Here is my understanding of that hierarchy – all these conversions, of course, are implicit. Using an explicit cast notation overrides the hierarchy similar to the way parentheses overrides the usual operator precedence of an expression.

1. An exact match is best. Surprisingly, for an argument to be an exact match, it does not need to exactly match the parameter type; it just needs to be close enough. This is the key to understanding what is going on in this example, and how the language has changed.
2. A promotion is better than a standard conversion. For example, promoting a `short int` to an `int` is better than converting an `int` into a `double`.
3. A standard conversion is better than a boxing conversion. For example, converting an `int` into a `double` is better that boxing an `int` into an `Object`.
4. A boxing conversion is better than an implicit user-defined conversion. For example, boxing an `int` into an `Object` is better than applying a conversion operator of a `SmallInt` value class.
5. An implicit user-defined conversion is better than no conversion at all. An implicit user-defined conversion is the last exit before Error (with the caveat that the formal signature might contain a param array or ellipsis at that position).

So, what does it mean to say that an exact match isn't necessarily exactly a match? For example, `const char[4]` does not exactly match either `const char*` or `String^`, and yet the ambiguity of our example is between two conflicting exact matches!

An exact match, as it happens, includes a number of trivial conversions. There are four trivial conversions under ISO-C++ that can be applied and still qualify as an exact match. Three are referred to as lvalue transformations. A fourth type is called a qualification conversion. The three lvalue transformations are treated as a better exact match than one requiring a qualification conversion.

One form of the lvalue transformation is the native-array-to-pointer conversion. This is what is involved in matching a `const char[4]` to `const char*`. Therefore, the match of `f("abc")` to `f(const char*)` is an exact match. In the earlier incarnations of our language, this was the best match, in fact.

For the compiler to flag the call as ambiguous, therefore, requires that the conversion of a `const char[4]` to a `String^` also be an exact match through a trivial conversion. This is the change that has been introduced in the new language version. And this is why the call is now flagged as ambiguous.

See Also

Reference

System::String Handling in Visual C++

Concepts

General Language Changes

Param Array and Ellipsis

Precedence of the param array for resolving overloaded function calls has changed from Managed Extensions for C++ to Visual C++ 2005.

In both Managed Extensions and the new syntax, there is no explicit support for the param array that C# and Visual Basic support. Instead, one flags an ordinary array with an attribute, as follows:

```
void Trace1( String* format, [ParamArray]Object* args[] );
void Trace2( String* format, Object* args[] );
```

While these both look the same, the `ParamArray` attribute tags this for C# or other CLR languages as an array taking a variable number of elements with each invocation. The change in behavior in programs between Managed Extensions and the new syntax is in the resolution of an overloaded function set in which one instance declares an ellipsis and a second declares a `ParamArray` attribute, as in the following example provided by Artur Laksberg.

```
int foo(...); // 1
int foo( [ParamArray] Int32[] ); // 2
```

In Managed Extensions, the ellipsis was given precedence over the attribute which is reasonable since the attribute is not a formal aspect of the language. However, in the new syntax, the param array is now supported directly within the language, and it is given precedence over the ellipsis because it is more strongly typed. Thus, in Managed Extensions, the call

```
foo( 1, 2 );
```

resolves to `foo(...)` while in the new syntax, it resolves to the `ParamArray` instance. On the off chance that your program behavior depends on the invocation of the ellipsis instance over that of the `ParamArray`, you will need to modify either the signature or the call.

See Also

Concepts

General Language Changes

typeof Goes to T::typeid

The **typeof** operator used in Managed Extensions for C++ has been supplanted by the **typeid** keyword in Visual C++ 2005.

In Managed Extensions, the **__typeof()** operator returns the associated `Type*` object when passed the name of a managed type. For example:

```
// Creates and initializes a new Array instance.
Array* myIntArray =
   Array::CreateInstance( __typeof(Int32), 5 );
```

In the new syntax, **__typeof** has been replaced by an additional form of **typeid** that returns a `Type^` when a managed type is specified.

```
// Creates and initializes a new Array instance.
Array^ myIntArray =
   Array::CreateInstance( Int32::typeid, 5 );
```

See Also

Reference

typeid

Concepts

General Language Changes

Initializer Lists

Initializer lists in constructors are now called before the base class constructor.

Remarks

Prior to Visual C++ 2005, the base class constructor was called before the initializer list when compiling with Managed Extensions for C++. Now, when compiling with **/clr**, the initializer list is called first.

See Also

Concepts

General Language Changes

Cast Notation and Introduction of safe_cast<>

The cast notation has changed from Managed Extensions for C++ to Visual C++ 2005.

Modifying an existing structure is a different and more difficult experience than crafting the initial structure. There are fewer degrees of freedom, and the solution tends towards a compromise between an ideal restructuring and what is practicable given the existing structural dependencies.

Language extension is another example. Back in the early 1990s as Object-Orienting programming became an important paradigm, the need for a type-safe downcast facility in C++ became pressing. Downcasting is the user-explicit conversion of a base-class pointer or reference to a pointer or reference of a derived class. Downcasting requires an explicit cast. The reason is that the actual type of the base class pointer is an aspect of the runtime; the compiler therefore cannot check it. Or, to rephrase that, a downcast facility, just like a virtual function call, requires some form of dynamic resolution. This raises two questions:

- Why should a downcast be necessary in the Object-Oriented paradigm? Isn't the virtual function mechanism sufficient? That is, why can't one claim that any need for a downcast (or a cast of any sort) is a design failure?
- Why should support of a downcast be a problem in C++? After all, it is not a problem in object-oriented languages such as Smalltalk (or, subsequently, Java and C#)? What is it about C++ that makes supporting a downcast facility difficult?

A virtual function represents a type-dependent algorithm common to a family of types. (We are not considering interfaces, which are not supported in ISO-C++ but are available in CLR programming and which represent an interesting design alternative). The design of that family is typically represented by a class hierarchy in which there is an abstract base class declaring the common interface (the virtual functions) and a set of concrete derived classes which represent the actual family types in the application domain.

A `Light` hierarchy in a Computer Generated Imagery (CGI) application domain, for example, will have common attributes such as `color`, `intensity`, `position`, `on`, `off`, and so on. One can control several lights, by using the common interface without worrying whether a particular light is a spotlight, a directional light, a non-directional light (think of the sun), or perhaps a barn-door light. In this case, downcasting to a particular light-type to exercise its virtual interface is unnecessary. In a production environment, however, speed is essential. One might downcast and explicitly invoke each method if by doing so inline execution of the calls can be performed instead of using the virtual mechanism.

So, one reason to downcast in C++ is to suppress the virtual mechanism in return for a significant gain in runtime performance. (Note that the automation of this manual optimization is an active area of research. However, it is more difficult to solve than replacing the explicit use of the **register** or **inline** keyword.)

A second reason to downcast falls out of the dual nature of polymorphism. One way to think of polymorphism is being divided into a passive and dynamic pair of forms.

A virtual invocation (and a downcast facility) represents dynamic uses of polymorphism: one is performing an action based on the actual type of the base class pointer at that particular instance in the execution of the program.

Assigning a derived class object to its base class pointer, however, is a passive form of polymorphism; it is using the polymorphism as a transport mechanism. This is the main use of `Object`, for example, in pre-generic CLR programming. When used passively, the base class pointer chosen for transport and storage typically offers an interface that is too abstract. `Object`, for example, provides roughly five methods through its interface; any more specific behavior requires an explicit downcast. For example, if we want to adjust the angle of our spotlight or its rate of fall off, we would have to downcast explicitly. A virtual interface within a family of sub-types cannot practicably be a superset of all the possible methods of its many children, and so a downcast facility will always be needed within an object-oriented language.

If a safe downcast facility is needed in an object-oriented language, then why did it take C++ so long to add one? The problem is in how to make the information as to the run-time type of the pointer available. In the case of a virtual function, the run-time information is set up in two parts by the compiler:

- The class object contains an additional virtual table pointer member (either at the beginning or end of the class object; that's has an interesting history in itself) that addresses the appropriate virtual table. For example, a spotlight object addresses a spotlight virtual table, a directional light, a directional light virtual table, and so on
- Each virtual function has an associated fixed slot in the table, and the actual instance to invoke is represented by the address stored within the table. For example, the virtual `Light` destructor might be associated with slot 0, `Color` with slot 1, and so on. This is an efficient if inflexible strategy because it is set up at compile-time and represents a minimal overhead.

The problem, then, is how to make the type information available to the pointer without changing the size of C++ pointers, either by adding a second address or by directly adding some sort of type encoding. This would not be acceptable to those programmers (and programs) that decide not to use the object-oriented paradigm – which was still the predominant user community. Another possibility was to introduce a special pointer for polymorphic class types, but this would be confusing, and make it difficult to inter-mix the two, particularly with issues of pointer arithmetic. It would also not be acceptable to maintain a run-time table that associates each pointer with its currently associated type, and dynamically updating it.

The problem then is a pair of user communities which have different but legitimate programming aspirations. The solution has to be a compromise between the two communities, allowing each not only their aspiration but the ability to interoperate. This means that the solutions offered by either side are likely to be infeasible and the solution implemented finally to be less than perfect. The actual resolution revolves around the definition of a polymorphic class: a polymorphic class is one that contains a virtual function. A polymorphic class supports a dynamic type-safe downcast. This solves the maintain-the-pointer-as-address problem because all polymorphic classes contain that additional pointer member to their associated virtual table. The associated type information, therefore, can be stored in an expanded virtual table structure. The cost of the type-safe downcast is (almost) localized to users of the facility.

The next issue with the type-safe downcast was its syntax. Because it is a cast, the original proposal to the ISO-C++ committee used the unadorned cast syntax, as in this example:

```
spot = ( SpotLight* ) plight;
```

but this was rejected by the committee because it did not allow the user to control the cost of the cast. If the dynamic type-safe downcast has the same syntax as the previously unsafe but static cast notation, then it becomes a substitution, and the user has no ability to suppress the runtime overhead when it is unnecessary and perhaps too costly.

In general, in C++, there is always a mechanism by which to suppress compiler-supported functionality. For example, we can turn off the virtual mechanism by either using the class scope operator (`Box::rotate(angle)`) or by invoking the virtual method through a class object (rather than a pointer or reference of that class). This latter suppression is not required by the language but is a quality of implementation issue, similar to the suppression of the construction of a temporary in a declaration of the form:

```
// compilers are free to optimize away the temporary
X x = X::X( 10 );
```

So the proposal was taken back for further consideration, and several alternative notations were considered, and the one brought back to the committee was of the form (`?type`), which indicated its undetermined – that is, dynamic nature. This gave the user the ability to toggle between the two forms – static or dynamic – but no one was too pleased with it. So it was back to the drawing board. The third and successful notation is the now standard `dynamic_cast<type>`, which was generalized to a set of four new-style cast notations.

In ISO-C++, `dynamic_cast` returns `0` when applied to an inappropriate pointer type, and throws a `std::bad_cast` exception when applied to a reference type. In Managed Extensions for C++, applying `dynamic_cast` to a managed reference type (because of its pointer representation) always returned `0`. `__try_cast<type>` was introduced as an analog to the exception throwing variant of the `dynamic_cast`, except that it throws `System::InvalidCastException` if the cast fails.

```
public __gc class ItemVerb;
public __gc class ItemVerbCollection {
public:
   ItemVerb *EnsureVerbArray() [] {
      return __try_cast<ItemVerb *[]>
         (verbList->ToArray(__typeof(ItemVerb *)));
   }
};
```

In the new syntax, `__try_cast` has been recast as `safe_cast`. Here is the same code fragment in the new syntax:

```
public ref class ItemVerb;
public ref class ItemVerbCollection {
public:
   array<ItemVerb^>^ EnsureVerbArray() {
      return safe_cast<array<ItemVerb^>^>
         ( verbList->ToArray( ItemVerb::typeid ));
   }
```

```
};
```

In the managed world, it is important to allow for verifiable code by limiting the ability of programmers to cast between types in ways that leave the code unverifiable. This is a critical aspect of the dynamic programming paradigm represented by the new syntax. For this reason, instances of old-style casts are recast internally as run-time casts, so that, for example:

```
// internally recast into the
// equivalent safe_cast expression above
( array<ItemVerb^>^ ) verbList->ToArray( ItemVerb::typeid );
```

On the other hand, because polymorphism provides both an active and a passive mode, it is sometimes necessary to perform a downcast just to gain access to the non-virtual API of a subtype. This can occur, for example, with the member(s) of a class that want to address any type within the hierarchy (passive polymorphism as a transport mechanism) but for which the actual instance within a particular program context is known. In this case, having a run-time check of the cast can be an unacceptable overhead. If the new syntax is to serve as the managed systems programming language, it must provide some means of allowing a compile-time (that is, static) downcast. That is why the application of the `static_cast` notation is allowed to remain a compile-time downcast:

```
// ok: cast performed at compile-time.
// No run-time check for type correctness
static_cast< array<ItemVerb^>^>(verbList->ToArray(ItemVerb::typeid));
```

The problem is that there is no way to guarantee that the programmer doing the `static_cast` is correct and well-intentioned; that is, there is no way to force managed code to be verifiable. This is a more urgent concern under the dynamic program paradigm than under native, but is not sufficient within a system programming language to disallow the user the ability to toggle between a static and run-time cast.

There is a performance trap and pitfall in the new syntax, however. In native programming, there is no difference in performance between the old-style cast notation and the new-style `static_cast` notation. But in the new syntax, the old-style cast notation is significantly more expensive than the use of the new-style `static_cast` notation. The reason is that the compiler internally transforms the use of the old-style notation into a run-time check that throws an exception. Moreover, it also changes the execution profile of the code because it causes an uncaught exception bringing down the application – perhaps wisely, but the same error would not cause that exception if the `static_cast` notation were used. One might argue this will help prod users into using the new-style notation. But only when it fails; otherwise, it will cause programs that use the old-style notation to run significantly slower without a visible understanding of why, similar to the following C programmer pitfalls:

```
// pitfall # 1:
// initialization can remove a temporary class object,
// assignment cannot
Matrix m;
m = another_matrix;

// pitfall # 2: declaration of class objects far from their use
Matrix m( 2000, 2000 ), n( 2000, 2000 );
if ( ! mumble ) return;
```

See Also

Reference

C-Style Casts with /clr

safe_cast

Concepts

General Language Changes

Security Best Practices for C++

This topic contains information on recommended security tools and practices. Using these resources and tools doesn't make applications immune from attack, but it makes successful attacks less likely.

Visual C++ Security Features

This section discusses security features that are built into the Visual C++ compiler and linker.

/GS (Buffer Security Check)

This compiler option (which is on by default), instructs the compiler to inject overrun detection code into functions that are at risk of being exploited. When an overrun is detected, execution is halted.

/SAFESEH (Image has Safe Exception Handlers)

This linker option (which is not enabled by default), instructs the linker to include into the output image, a table containing the address of each exception handler. At runtime, the operating system uses this table to insure that only legitimate exception handlers are executed. This helps prevent the execution of exception handlers introduced by a runtime hacker attack.

/analyze (Enterprise Code Analysis)

This compiler option (which is not enabled by default) activates code analysis that reports potential security issues such as buffer overrun, un-initialized memory, null pointer dereferencing, and memory leaks. See Code Analysis for C/C++ Overview for more information.

Security-Enhanced CRT

For Visual C++ 2005, the C Runtime Library (CRT) has been augmented to include secure versions of functions that pose security risks. (The unchecked strcpy string copy function, for example.) The older, insecure versions of these functions are now deprecated, so their use results in compile-time warnings. Programmers are strongly encouraged to use the secure versions of these CRT functions rather than suppress these compilation warnings. See Security Enhancements in the CRT for more information.

Checked Iterators

With checked iterators, users of the Standard C++ Library container classes are notified of attempts to access elements outside the bounds of a container. See Checked Iterators for more information.

Code Analysis for Managed Code

Code Analysis for Managed Code, also known as FxCop, is a tool which checks assemblies for conformance to the Microsoft .NET Framework Design Guidelines. FxCop analyzes the code and metadata within each assembly to check for defects in the following areas:

- Library design
- Localization
- Naming conventions
- Performance
- Security

Code Analysis for Managed Code is included in Visual Studio Team System, and can also be downloaded at http://www.gotdotnet.com/team/fxcop/.

Windows Application Verifier

Available as part of the Application Compatibility Toolkit, the Application Verifier (AppVerifier) is a tool that can help developers identify potential application compatibility, stability, and security issues.

The AppVerifier works by monitoring an application's use of the operating system, including the file system, registry, memory, and APIs, while the application is being run. The tool then provides guidance for source-code level fixes of the issues it uncovers.

The verifier lets you:

- Test for potential application compatibility errors caused by common programming mistakes.
- Examine an application for memory-related issues.
- Determine an application's compliance with various requirements of the Designed for Windows XP or Certified for Windows Server™ 2003 Logo Programs.
- Identify potential security issues in an application.

The Windows Application Verifier is available at http://www.microsoft.com/windows/appcompatibility/appverifier.mspx.

.NET Framework Security Features

This section gives an overview of two related .NET Framework security features.

Code Access Security

Describes the .NET Framework security system and how it interfaces with application code.

Configuring Security Policy

Describes guidelines and tools for adjusting .NET Framework security policies.

Windows User Accounts

Using Windows user accounts that belong to the Administrators group exposes developers and--by extension--customers to security risks. See Running as a Member of the Users Group for more information.

Increasing Security by Using User Account Control under Windows Vista

User Account Control (UAC) is a feature of Windows Vista in which user accounts have limited privileges. Detailed information about UAC can be found at http://go.microsoft.com/fwlink/?linkid=53781.

Note
If you build a Visual C++ project on Windows Vista with UAC disabled, and you later enable UAC, you will need to clean and rebuild the project for it to work correctly.

See Also

Reference

System.Security

Other Resources

Securing Applications

Running as a Member of the Users Group

This topic explains how configuring Windows user accounts as a member of the Users Group (as opposed to the Administrators Group) increases security by reducing the chances of being infected with malicious code.

Security Risks

Running as an administrator makes your system vulnerable to several kinds of security attack, such as "Trojan horse" and "buffer overrun." Merely visiting an Internet site as an administrator can be damaging to the system, as malicious code that is downloaded from an Internet site may attack your computer. If successful, it inherits your administrator permissions and can then perform actions such as deleting all your files, reformatting your hard drive, and creating a new user accounts with administrative access.

Non Administrator User Groups

The Windows user accounts that developers use normally should be added to either the Users or Power Users Groups. Developers should also be added to the Debugging Group. Being a member of the Users group allows you to perform routine tasks including running programs and visiting Internet sites without exposing your computer to unnecessary risk. As a member of the Power Users group, you can also perform tasks such as application installation, printer installation, and most Control Panel operations. If you need to perform administrative tasks such as upgrading the operating system or configuring system parameters, you should log into an administrator account for just long enough to perform the administrative task. Alternatively, the Windows **runas** command can be used to launch specific applications with Administrative access.

Exposing Customers to Security Risks

Not being part of the Administrators group is particularly important for developers because, in addition to protecting development machines, it prevents developers from inadvertently writing code that requires customers to join the Administrators Group in order to execute the applications you develop. If code that requires administrator access is introduced during development, it will fail at runtime, alerting you to the fact that your application now requires customers to run as Administrators.

Code That Requires Administrator Privileges

Some code requires Administrator access in order to execute. If possible, alternatives to this code should be pursued. Examples of code operations that require Administrator access are:

- Writing to protected areas of the file system, such as the Windows or Program Files directories
- Writing to protected areas of the registry, such as HKEY_LOCAL_MACHINE
- Installing assemblies in the Global Assembly Cache (GAC)

Generally, these actions should be limited to application installation programs. This allows users to use administrator status only temporarily.

Debugging

You can debug any applications that you launch within Visual Studio (native and unmanaged) as a non-administrator by becoming part of the Debugging Group. This includes the ability to attach to a running application using the Attach to Process command. However, it is necessary to be part of the Administrator Group in order to debug native or managed applications that were launched by a different user.

See Also

Concepts

Security Best Practices for C++

Creating and Managing Visual C++ Projects

You can create Visual C++ projects in several ways:

- You can use project templates, such as the CLR Console Application Template (C++), to help you quickly create simple projects.
- You can also use application wizards to help you create a solution. A solution can contain many projects and can be coded in any language that is included with Visual Studio. The Visual Studio development environment can handle dependencies among projects, individual project configurations, project deployment, and source code control. This method works well for larger scale applications.
- You can create a simple text file and then save it with a .cpp extension. Using the Win32 Application Wizard, create an empty project. Add your .cpp file into the Visual Studio environment. This method would work well for a very simple, console-style application.

Once you have created your project, you can manage it using code wizards and property pages to manage the specifics of your project.

In This Section

Using Projects to Create Applications

Describes how to create projects with Visual C++ application wizards, which allow you to create projects that employ the various libraries.

Visual C++ Projects

Lists all the different types of project templates or application wizards that you can use to build applications with Visual C++.

Modifying Project Settings

Describes how to use Property Pages and Property Sheets to specify your project settings.

Adding Functionality with Code Wizards

Describes how to add classes, methods, variables, and other elements to your project to add functionality.

Designing a Wizard

Describes concepts on how to create your own wizard that generates code and a user interface for other users.

Related Sections

Project Templates in Visual C++ Editions

Provides an overview of which project templates are available in Visual C++ editions.

Managing Solutions, Projects, and Files

Provides links to topics describing common tasks that you can perform when working with solutions and projects.

Building a C/C++ Program

Provides links to topics describing building your program from the command line or from the integrated development environment of Visual Studio.

User Interface

Provides links to topics describing adding a user interface to your program.

General Concepts

Provides links to topics describing conceptual information about the Visual C++ libraries and topics discussing various coding technologies and techniques.

Creating and Porting Visual C++ Device Projects

Explains how to create a new Visual C++ device project and how to port existing Visual C++ desktop projects and eMbedded Visual C++ 4.0 projects into a Visual C++ device project.

Developing Visual C++ Device Projects

Describes how to use the features of Visual Studio 2005 to develop a device application using Visual C++.

Visual C++ Reference

Provides links to topics describing the C and C++ language references, the libraries provided with Visual C++, the Visual C++ Extensibility Object Model, and the Microsoft Macro Assembler (MASM).

Visual C++ Samples

Provides links to sample code showing the capabilities of Visual C++ and the libraries and technologies it supports.

See Also

Other Resources

Visual Studio SDK

Using Projects to Create Applications

A project is defined as a configuration and a group of files that produce a program or final binary file or files. Each project contains at least two configurations — a debug configuration and a retail or release configuration.

Projects are the individual parts of your solution that address different application requirements. Solution definitions include the dependency relationships among projects. They also include information about how to deploy the projects that make up your solution. Between them, the projects in your solution make up an application.

Application Wizards

The best way to create new projects is by using Visual C++ application wizards. For detailed information see, Creating a Project with a Visual C++ Application Wizard.

You can always write programs without using application wizards, but because the wizards provide a complete set of project files and structureby using a wizard to start your project lets you continue immediately to programming functionality. To create an empty project, use the Win32 Project Wizard. You can add your own .cpp files to this project. This method works for simple console applications.

Project Templates

Visual C++ provides project templates for targeting the common language runtime and the .NET Framework class library. These project templates do not provide a user interface; instead, they automatically add the important project references and files.

Code Wizards

After you create the framework of your project, you can use the Visual C++ code wizards to help you add items like classes, members or functions into your programs. For more information, see Adding Functionality with Code Wizards.

Property Pages

Once you create a project and add items, you will also want to specify how your project is built and debugged. To modify your project's settings, see:

- Specifying Project Settings with Property Pages
- Setting Visual C++ Project Properties.

Projects have also external aspects that control how they are compiled and deployed. Projects are part of solutions. They are managed by Visual Studio. Various projects in a solution may be coded in any of the languages included with Visual Studio. Four aspects of each project are defined in the larger Visual Studio development environment:

- Project configurations
- Project dependencies
- Deployment
- Source control

See Also

Other Resources

Creating and Managing Visual C++ Projects
Managing Solutions, Projects, and Files

Browsing Projects and Solutions

How to: Display Inheritance Graphs

Inheritance browsing, or viewing type and member hierarchies using inheritance graphs, is a feature that is available through Class View and the Object Browser. It enables you to view the base types and derived types of the current type that is selected in the **Class View** tree or **Object Browser**. It also enables you to view inherited members of the currently selected type.

To display base types

1. Open your project in the integrated development environment (IDE).
2. In **Class View**, on the **Class View Settings** menu, select **Show Base Types**.

   — or —

   In the **Object Browser**, on the **Object Browser Settings** menu, select **Show Base Types**.

   For each node in the tree that is a type, a **Base Types** sub node appears that contains a list of all the base types of a type.

To display derived types

1. Open your project in the IDE.
2. In **Class View**, on the **Class View Grouping** menu, select **Show Derived Types**.

   — or —

   In the **Object Browser**, on the **Object Browser Settings** menu, select **Show Derived Types**.

   For each node in the tree that is a type, a **Derived Types** sub node appears that contains a list of all the derived types of a type.

To display inherited members

1. Open your project in the IDE.
2. In **Class View**, on the **Class View Grouping** menu, select **Show Inherited Members**.

   — or —

   In the **Object Browser**, on the **Object Browser Settings** menu, select **Show Inherited Members**.

   When you select a type in the tree, inherited members are displayed in the members pane along with the members defined within that type.

See Also

Tasks

How to: Display Call Graphs

Other Resources

Viewing Classes, Members, and Symbols

Converting Projects from Mixed Mode to Pure Intermediate Language

All Visual C++ CLR projects link to the C run-time libraries by default. Consequently, these projects are classified as mixed-mode applications, because they combine native code with code that targets the common language runtime (managed code). When they are compiled, they are compiled into intermediate language (IL), also known as Microsoft intermediate language (MSIL).

To convert your mixed-mode application into pure intermediate language

1. Remove links to the C run-time libraries (CRT):

   a. In the .cpp file defining the entry point of your application, change the entry point to `Main()`. Using `Main()` indicates that your project does not link to the CRT.

   b. In Solution Explorer, right-click your project and select **Properties** on the shortcut menu to open the property pages for your application.

   c. In the **Advanced** project property page for the **Linker**, select the **Entry Point** and then enter **Main** in this field.

   d. For console applications, in the **System** project property page for the **Linker**, select the **SubSystem** field and change this to **Console (/SUBSYSTEM:CONSOLE)**.

   > **Note**
   >
   > You do not have to set this property for Windows Forms applications because the **SubSystem** field is set to **Windows (/SUBSYSTEM:WINDOWS)** by default.

   e. In stdafx.h, comment out all the `#include` statements. For example, in console applications:

   ```
   // #include <iostream>
   // #include <tchar.h>
   ```

   -or-

   For example, in Windows Forms applications:

   ```
   // #include <stdlib.h>
   // #include <malloc.h>
   // #include <memory.h>
   // #include <tchar.h>
   ```

   f. For Windows Forms applications, in Form1.cpp, comment out the `#include` statement that references windows.h. For example:

   ```
   // #include <windows.h>
   ```

2. Add the following code to stdafx.h:

   ```
   #ifndef __FLTUSED__
   #define __FLTUSED__
      extern "C" __declspec(selectany) int _fltused=1;
   #endif
   ```

3. Remove all unmanaged types:

   a. Wherever appropriate, replace unmanaged types with references to structures from the System namespace.

Common managed types are listed in the following table:

Structure	Description
Boolean	Represents a Boolean value.
Byte	Represents an 8-bit unsigned integer.
Char	Represents a Unicode character.
DateTime	Represents an instant in time, typically expressed as a date and time of day.
Decimal	Represents a decimal number.
Double	Represents a double-precision floating-point number.
Guid	Represents a globally unique identifier (GUID).
Int16	Represents a 16-bit signed integer.
Int32	Represents a 32-bit signed integer.
Int64	Represents a 64-bit signed integer.
IntPtr	A platform-specific type that is used to represent a pointer or a handle.
SByte	Represents an 8-bit signed integer.
Single	Represents a single-precision floating-point number.
TimeSpan	Represents a time interval.
UInt16	Represents a 16-bit unsigned integer.
UInt32	Represents a 32-bit unsigned integer.
UInt64	Represents a 64-bit unsigned integer.
UIntPtr	A platform-specific type that is used to represent a pointer or a handle.
Void	Indicates a method that does not return a value; that is, the method has the void return type.

See Also

Concepts

Compiling to MSIL

How to: Build Isolated Applications to Consume COM Components

Isolated applications are applications that have manifests built into the program. You can create isolated applications to consume COM components.

To add COM references to manifests of isolated applications

1. Open the project property pages for the isolated application.
2. Expand the **Configuration Properties** node, and then expand the **Manifest Tool** node.
3. Select the **Isolated COM** property page, and then set the **Component File Name** property to the name of the COM component that you want the isolated application to consume.
4. Click **OK**.

To build manifests into isolated applications

1. Open the project property pages for the isolated application.
2. Expand the **Configuration Properties** node, and then expand the **Manifest Tool** node.
3. Select the **Input and Output** property page, and then set the **Embed Manifest** property equal to **Yes**.
4. Click **OK**.
5. Build the solution.

See Also

Reference

Isolated Applications
Side-by-side Assemblies

How to: Build Registration-Free COM Components

Registration-free COM components are COM components that have manifests built into the DLLs.

To build manifests into COM components

1. Open the project property pages for the COM component.
2. Expand the **Configuration Properties** node, and then expand the **Manifest Tool** node.
3. Select the **Input and Output** property page, and then set the **Embed Manifest** property equal to **Yes**.
4. Click **OK**.
5. Build the solution.

See Also

Tasks

How to: Build Isolated Applications to Consume COM Components

Reference

Isolated Applications

Side-by-side Assemblies

Visual C++ Projects

An application wizard provides a user interface that you use to create a project, modeled after a project template, and generates source files and directories for the application. The wizard provides program structure, basic menus, toolbars, icons, and appropriate **#include** statements. Visual C++ application wizards work in conjunction with application frameworks and libraries to create starter programs for you.

You can always write programs without using application wizards, but because the wizards provide a complete set of project files and structure, using a wizard to begin your project allows you to proceed immediately to programming functionality.

Project Types

Visual Studio contains a project template or application wizard for the following project types. Each wizard helps you create projects:

CLR

Project template
ASP.NET Web Service Template (C++)
Class Library Template (C++)
CLR Console Application Template (C++)
CLR Empty Project Template (C++)
Windows Control Library Template (C++)
Windows Forms Application Template
Windows Service Template (C++)

ATL

Project template	How to create a project
ATL Project	Creating an ATL Project
ATL Server Project	Creating an ATL Server Project
ATL Server Web Service	Creating an ATL Server Web Service

MFC

Project template	How to create a project
MFC ActiveX Control	Creating an MFC ActiveX Control Project
MFC Application	Creating an MFC Application
MFC DLL	Creating an MFC DLL Project

Win32

Project template	How to create a project
Win32 Console Project	Creating a Win32 Console Application
Win32 Project	Creating a Win32 Windows Application

General

Project template	How to create a project

Custom Wizard	Creating a Custom Wizard
Makefile Project	Creating a Makefile Project

TODO Comments

The files generated by an application wizard contain TODO comments in the form of comments. These are areas where you provide your own source code or where you can use more wizards to develop your application. A list of the new project's TODO comments appears in the Task List.

See the following topics for more information about using these features:

- Adding Functionality with Code Wizards
- Working with Resources

After running the application wizard, you can immediately build the project successfully and run the application.

See Also

Tasks

Creating Projects with Application Wizards

Reference

Project Templates in Visual C++ Editions

Using Projects to Create Applications

Files Created for CLR Projects

Other Resources

Managing Solutions, Projects, and Files

Creating Projects with Application Wizards

Every type of Visual C++ project has an application wizard that helps you generate new projects quickly and easily, modeled from the project template. For details, see Visual C++ Projects.

To open an application wizard, use the New Project dialog box to specify your project properties like the name, or the directory and solution where your project will reside.

To open a Visual C++ application wizard

1. On the **File** menu, click **New**, and then click **Project**.

   The **New Project** dialog box appears.

2. In the Project Types pane, select the **Visual C++ Projects** folder. An icon for every type of C++ project appears in the Templates pane.

3. In the Templates pane, select an icon to choose a project type. A message appears under both panes indicating the type of project you are going to create.

4. Specify your project properties, or skip this step to use Visual Studio default project properties. See Projects as Containers for details.

5. Click **OK**, and the wizard for your project type opens.

When you create a project you can either create a new solution or you can add the project to an existing solution. For more information, see Solutions as Containers.

See Also

Concepts

Introduction to Solutions, Projects, and Items

Other Resources

Creating and Managing Visual C++ Projects

Wizard Support for Other Languages

When you install Visual Studio, the setup application detects the locale listed in your system and installs the appropriate language template or templates for that locale. For example, for Western European locales, setup installs English, French, Italian, Spanish, and German. These languages appear in the **Resource language** list on the Application Type page of the MFC Application Wizard.

Language Templates

Not all templates are installed on all systems, because the templates are ANSI encoding based, and not all resources can be edited on all systems. For example, by default, you cannot edit Japanese resources on a French system.

If you are using Windows 2000 or later and you want to create an MFC application in another language, then you must copy the template directory for the appropriate language from the Visual Studio installer media (Disk 1) to your system.

Note
To edit the created project, you must set your system locale to the appropriate locale for the selected language.

The templates are each assigned a folder in the \Microsoft Visual Studio .NET 2003\Vc7\VCWizards\mfcappwiz\templates\ directory, as listed in the following table . To access the desired language template, copy the appropriate folder to the \mfcappwiz\templates\ directory on your computer. Once you have copied the folder, the language will appear in the **Resource language** list on the **Application Type** page of the MFC Application Wizard.

Language Templates Provided in Visual Studio .NET

Language	Template
Chinese (Traditional)	1028
Chinese (Simplified)	2052
English	1033
French	1036
German	1031
Italian	1040
Japanese	1041
Korean	1042
Spanish	3082

Format of Visual C++ Wizard-generated Files

The Visual C++ wizards will generate projects in Unicode when the installed language version of Visual Studio does not match the system locale. For example, when the Japanese version of Visual Studio is installed on a computer that has regional settings set to any language other than Japanese, then the Visual C++ wizards will generate projects comprised of Unicode files. This is common on machines set up with Windows multi-language (MUI) packs.

This behavior differs from systems set up such that the system locale is the same as the language version of Visual Studio. In this case, project files will be created in ANSI in the system code page.

See Also

Concepts

File Types Created for Visual C++ Projects

Other Resources

File Types Created for Visual C++ Projects

This topic describes all the types of files that are associated with Visual C++ projects. The actual files included in your project depend on the project type and the options you select when using a wizard.

- Project Files and Makefiles
- Files Created for CLR Projects
- ATL Program or Control Source and Header Files
- MFC Program or Control Source and Header Files
- Precompiled Header Files
- Resource Files
- Help Files (WinHelp)
- Miscellaneous Project Files

When you create a Visual C++ project, you might be creating a new solution, or you might be adding a project to a solution. Non-trivial applications are commonly developed with multiple projects in a solution.

Projects usually produce either an EXE or a DLL. Projects can be dependent on each other; during the build process, the Visual C++ environment checks dependencies both within and between projects. Each project has core source code, and depending on the kind of project, it may have many other files containing various aspects of the project. The contents of these files are indicated by the file extension. The Visual Studio development environment uses the file extensions to determine how to handle the file contents during a build.

The following table shows common files in a Visual C++ project, and identifies them with their file extension. Throughout this section, *Projname* represents the name of the project.

File extension	Type	Contents
.asmx	Source	Deployment file.
.asp	Source	Active Server Page file.
.atp	Project	Application template project file.
.bmp, .dib, .gif, .jpg, .jpe, .png	Resource	General image files.
.bsc	Compiling	The browser code file.
.cpp; .c	Source	Main source code files for your application.

.cur	Resource	Cursor bitmap graphic file.
.dbp	Project	Database project file.
.disco	Source	The dynamic discovery document file. Handles XML Web service discovery.
.exe, .dll	Project	Executable or dynamic-link library files.
.h	Source	The header, or include, file.
.htm, .html, .xsp, .asp, .htc, .hta, .xml	Resource	Common Web files.
.HxC	Project	Help project file.
.ico	Resource	Icon bitmap graphic file.
.idb	Compiling	The state file, containing dependency information between source files and class definitions, which can be used by the compiler during minimal rebuild and incremental compilation. Use the /Fd compiler option to specify the name of the .idb file. See /Gm (Enable Minimal Rebuild) for more information.
.idl	Compiling	The interface definition language file. See Interface Definition (IDL) File in the *Platform SDK* for more information.
.ilk	Linking	Incremental link file. See /INCREMENTAL for more information.
.map	Linking	A text file containing linker information. Use the /Fm compiler option to name the map file. See /MAP for more information.
.ncb	Solution	The no compile browser file.
.obj, .o		Object files, compiled but not linked.

.pch	Debug	Precompiled header file.
.pdb	Debug	The program debug database file. See What Are .pdb Files? for more information.
.rc, .rc2	Resource	Resource script files to generate resources.
.sbr	Compiling	Source browser intermediate file. The input file for BSCMAKE.
.sln	Solution	The solution file.
.suo	Solution	The solution options file.
.srf	Project	The server response file. This file contains the HTML code for an ATL Server application.
.txt	Resource	Text file, usually the "readme" file.
.vap	Project	Visual Studio Analyzer project file.
.vbg	Solution	Compatible project group file.
.vbp, .vip, .vbproj	Project	The Visual Basic project file.
.vcproj	Project	The Visual C++ project file. See Project Files and Makefiles for more information.
.vdproj	Project	The Visual Studio deployment project file.
.vmx	Project	The macro project file.
.vup	Project	The utility project file.

For information on other files associated with Visual Studio, see File Types and File Extensions in Visual Studio .NET.

Project files are organized into folders in Solution Explorer. Visual C++ creates a folder for source files, header files, and resource files, but you can reorganize these folders or create new ones. You can use folders to organize explicitly logical clusters of files within the hierarchy of a project. For example, you could create folders to contain all your user interface source files, or specifications, documentation, or test suites. All file folder names should be unique.

When you add an item to a project, you add the item to all configurations for that project, regardless of whether or not the item is buildable. For example, if you have a project named MyProject, adding an item adds it to both the Debug and Release project configurations.

See Also

Concepts

Visual C++ Projects

Wizard Support for Other Languages

Other Resources

Creating and Managing Visual C++ Projects

Project Files and Makefiles

The following files are created when you create a project in Visual Studio. They are used to manage project files within the solution.

File name	Directory location	Solution Explorer location	Description
Solname.sln	*Projname*	Not displayed in Solution Explorer	The solution file used within the development environment. It organizes all elements of a project or multiple projects into a single solution. This file is stored in the parent project directory.
Projname.suo	*Projname*	Not displayed in Solution Explorer	The solution options file used within the development environment. It stores all the user options you create for your solution, so that each time you open the project's solution it has the look and feel you want and includes any customizations you have made. This file is stored in the parent project directory.
Projname.vcproj	*Projname*	Not displayed in Solution Explorer	The project file used within the development environment. In previous versions, this file was called *Projname*.dsp. It stores the information specific to your project. Each project has a separate .vcproj file. For more information, see Format of a .vcproj File.This file is stored in the parent project directory.
Projname.idl	*Projname*	Source	The file containing the Interface Description Language source code for a control type library. This file is used by Visual C++ to generate a type library. The generated library exposes the control's interface to other Automation clients. SeeInterface Definition (IDL) File in the *Platform SDK* for more information.
Projname.ncb	*Projname*	Not displayed in Solution Explorer	The no compile browser file. It contains information generated by the parser, which is used by Class View. If the file is accidentally or deliberately deleted, it is automatically regenerated.
Readme.txt	*Projname*	Project	A file describing each file in a project, using the actual filenames created by the application wizard. It is located in the parent directory of the project.

See Also

Concepts

File Types Created for Visual C++ Projects

Precompiled Header Files

These files are used to build a precompiled header file *Projname*.pch and a precompiled types file Stdafx.obj.

These files are located in the *Projname* directory. In Solution Explorer, Stdafx.h is in the Header Files folder, and Stdafx.cpp is located in the Source Files folder.

File name	**Description**
Stdafx.h	An include file for standard system include files and for project-specific include files that are used frequently but are changed infrequently. You should not define or undefine any of the _AFX_NO_XXX macros in stdafx.h; see the Knowledge Base article "PRB: Problems Occur When Defining _AFX_NO_XXX". You can find Knowledge Base articles in the MSDN Library or at http://search.support.microsoft.com/.
Stdafx.cpp	Contains the preprocessor directive `#include "stdafx.h"` and adds include files for precompiled types. Precompiled files of any type, including header files, support faster compilation times by restricting compilation only to those files that require it. Once your project is built the first time, you will notice much faster build times on subsequent builds because of the presence of the precompiled header files.

See Also

Tasks

How to: Specify Project Properties with Property Pages

Concepts

File Types Created for Visual C++ Projects

Resource Files (C++)

Resources are interface elements that provide information to the user. Bitmaps, icons, toolbars, and cursors are all resources. Some resources can be manipulated to perform an action such as selecting from a menu or entering data in dialog box.

See Working with Resources for more information.

File name	Directory location	Solution Explorer location	Description
Projname.rc	*Projname*	Source Files	The resource script file for the project. The resource script file contains the following, depending on the type of project, and the support selected for the project (for example, toolbars, dialog boxes, or HTML): • Default menu definition. • Accelerator and string tables. • Default **About** dialog box. • Other dialog boxes. • Icon file (res*Projname*.ico). • Version information. • Bitmaps. • Toolbar. • HTML files. The resource file includes the file Afxres.rc for standard Microsoft Foundation Class resources.
Resource.h	*Projname*	Header Files	The resource header file that includes definitions for the resources used by the project.
Projname.rc2	*Projname*\res	Source Files	The script file containing additional resources used by the project. You can include the .rc2 file at the top of the project's .rc file. An .rc2 file is useful for including resources used by several different projects. Instead of having to create the same resources several times for different projects, you can put them in an .rc2 file and include the .rc2 file into the main .rc file.
Projname.def	*Projname*	Source Files	The module definition file for a DLL project. For a control, it provides the name and description of the control, as well as the size of the run-time heap.
Projname.ico	*Projname*\res	Resource Files	The icon file for the project or control. This icon appears when the application is minimized. It is also used in the application's **About** box. By default, MFC provides the MFC icon, and ATL provides the ATL icon.
*Projname*Doc.ico	*Projname*\res	Resource Files	The icon file for an MFC project that includes support for the document/view architecture.
Toolbar.bmp	*Projname*\res	Resource Files	The bitmap file representing the application or control in a toolbar or palette. This bitmap is included in the project's resource file. The initial toolbar and status bar are constructed in the **CMainFrame** class.

See Also

Concepts

File Types Created for Visual C++ Projects

Files Created for CLR Projects

When you use Visual C++ templates to create your projects, several files are created, depending on which template you use. The following table lists all the files that are created by project templates for .NET Framework projects.

File name	File description
AssemblyInfo.cpp	The file that contains information (that is, attributes, files, resources, types, versioning information, signing information, and so on) for modifying the project's assembly metadata. For more information see Assembly Concepts in the .NET Framework SDK.
projname.asmx	A text file that references managed classes that encapsulate the functionality of the XML Web service.
projname.cpp	The main source file and entry point into the application that Visual Studio created for you. Identifies the project .dll file and the project namespace. Provide your own code in this file.
projname.vsdisco	An XML deployment file containing links to other resources that describe the XML Web service.
projname.h	The main include file for the project, which contains all declarations, global symbols, and **#include** directives for other header files.
projname.ncb	The no compile browser file containing information generated by the parser and used by Class View
projname.sln	The solution file used within the development environment to organize all elements of your project into a single solution.
projname.suo	The solution options file used within the development environment.
projname.vcproj	The project file used within the development environment that stores the information specific to this project.
ReadMe.txt	A file describing each file in your project using the actual filenames created by the template.

See Also

Tasks

Walkthrough: Creating an XML Web Service Using C++ and the CLR

ATL Program or Control Source and Header Files

The following files are created when you create an ATL project in Visual Studio, depending on the options you select for the project you create.

All of these files are located in the *Projname* directory, and in either the Header Files (.h files) folder or Source Files (.cpp files) folder in Solution Explorer.

Your project includes *ProjnameIsapi*.vcproj, *ProjnameIsapi*.h, *ProjnameIsapi*.cpp, and ProjName.srf files only if you create an ATL Server project.

File name	Description
Projname.h	The main include file containing the C++ interface definitions and GUID declarations of the items defined in ATLSample.idl. It is regenerated by MIDL during compilation.
Projname.cpp	The main program source file. It contains the implementation of your DLL's exports for an in-process server and the implementation of **WinMain** for a local server. For a service, this additionally implements all the service management functions.
*Projname*Isapi.cpp	The root ISAPI Extension file containing the ISAPI Extension code and additional ATL Server code. You can customize the ATL Server code to the specific needs of your projects.
*Projname*Isapi.vcproj	The main project file for Visual C++ projects generated using an application wizard. It contains information about the version of Visual C++ that generated the file, and information about the platforms, configurations, and project features selected with the application wizard.
*Projname*Isapi.def	Contains the functions that will be exported from your ISAPI Extension DLL, including the ISAPI Extension functions **HttpExtensionProc**, **GetExtensionVersion**, and **TerminateExtension**. These functions are delegated to the instance of your **CIsapiExtension** class.
Resource.h	The header file for the resource file.
StdAfx.cpp	Includes the files StdAfx.h and Atlimpl.cpp.
StdAfx.h	Includes the ATL header files.
ProjName.srf	The server response file. Contains the HTML code for an ATL Server application.

See Also

Reference

MFC Program or Control Source and Header Files

Files Created for CLR Projects

Concepts

File Types Created for Visual C++ Projects

MFC Program or Control Source and Header Files

The following files are created when you create an MFC project in Visual Studio, depending on the options you select for the project you create. For example, your project contains *Projname*dlg.cpp and *Projname*dlg.h files only if you create a dialog-based project or class.

All of these files are located in the *Projname* directory, and in either the Header Files (.h files) folder or Source Files (.cpp files) folder in Solution Explorer.

File name	Description
Projname.h	The main include file for the program or DLL. It contains all global symbols and **#include** directives for other header files. It derives the `CPrjnameApp` class from **CWinApp** and declares an `InitInstance` member function. For a control, the `CPrjnameApp` class is derived from **COleControlModule**.
Projname.cpp	The main program source file. It creates one object of the class `CPrjnameApp`, which is derived from **CWinApp**, and overrides the **InitInstance** member function. For executables, `CPrjnameApp::InitInstance` does several things. It registers document templates, which serve as a connection between documents and views; creates a main frame window; and creates an empty document (or opens a document if one is specified as a command-line argument to the application). For DLLs and ActiveX (formerly OLE) controls, `CProjNameApp::InitInstance` registers the control's object factory with OLE by calling **COleObjectFactory::RegisterAll** and makes a call to **AfxOLEInit**. In addition, the member function `CProjNameApp::ExitInstance` is used to unload the control from memory with a call to **AfxOleTerm**. This file also registers and unregisters the control in the Windows registration database by implementing the **DllRegisterServer** and **DllUnregisterServer** functions.
*Projname*ctrl.h, *Projname*ctrl.cpp	Declare and implement the `CProjnameCtrl` class. `CProjnameCtrl` is derived from **COleControl**, and skeleton implementations of some member functions are defined that initialize, draw, and serialize (load and save) the control. Message, event, and dispatch maps are also defined.
*Projname*dlg.cpp, *Projname*dlg.h	Created if you choose a dialog-based application. The files derive and implement the dialog class, named `CProjnameDlg`, and include skeleton member functions to initialize a dialog and perform dialog data exchange (DDX). Your About dialog class is also placed in these files instead of in *Projname*.cpp.
Dlgproxy.cp, Dlgproxy.h	In a dialog-based program, the implementation and header file for the project's Automation proxy class for the main dialog. This is only used if you have chosen Automation support.
*Projname*doc.cpp, *Projname*doc.h	Derive and implement the document class, named `CProjnameDoc`, and include skeleton member functions to initialize a document, serialize (save and load) a document, and implement debugging diagnostics.
*Projname*set.h/.cpp	Created if you create a program that supports a database and contains the recordset class.

*Projname*view.cpp, *Projname*view.h	Derive and implement the view class, named `CProjnameView`, which is used to display and print the document data. The `CProjnameView` class is derived from one of the following MFC classes: • CEditView • CFormView • CRecordView • COleDBRecordView • CTreeView • CListView • CRichEditView • CScrollView • CView • CHTMLView • CHTMLEditView The project's view class contains skeleton member functions to draw the view and implement debugging diagnostics. If you have enabled support for printing, then message-map entries are added for print, print setup, and print preview command messages. These entries call the corresponding member functions in the base view class.
*Projname*PropPage.h, *Projname*PropPage.cpp	Declare and implement the `CProjnamePropPage` class. `CProjnamePropPage` is derived from **COlePropertyPage** and a skeleton member function, `DoDataExchange`, is provided to implement data exchange and validation.
IPframe.cpp, IPframe.h	Created if the Mini-Server or Full-Server option is selected in the application wizard's **Automation Options** page (step 3 of 6). The files derive and implement the in-place frame window class, named **CInPlaceFrame**, used when the server is in place activated by a container program.
Mainfrm.cpp, Mainfrm.h	Derive the **CMainFrame** class from either CFrameWnd (for SDI applications) or CMDIFrameWnd (for MDI applications). The **CMainFrame** class handles the creation of toolbar buttons and the status bar, if the corresponding options are selected in the application wizard's **Application Options** page (step 4 of 6). For information on using **CMainFrame**, see The Frame-Window Classes Created by the Application Wizard.
Childfrm.cpp, Childfrm.h	Derive the **CChildFrame** class from CMDIChildWnd. The **CChildFrame** class is used for MDI document frame windows. These files are always created if you select the MDI option.

See Also

Reference

ATL Program or Control Source and Header Files

Files Created for CLR Projects

Concepts

File Types Created for Visual C++ Projects

Help Files (HTML Help)

The following files are created when you add the HTML Help type of Help support to your application by selecting the **Context-sensitive help** check box and then selecting **HTML Help format** in the Advanced Features page of the MFC Application Wizard.

File name	Directory location	Solution Explorer location	Description
Projname.hhp	*Projname*\hlp	HTML Help files	The help project file. It contains the data needed to compile the help files into an .hxs file or a .chm file.
Projname.hhk	*Projname*\hlp	HTML Help files	Contains an index of the help topics.
Projname.hhc	*Projname*\hlp	HTML Help files	The contents of the help project.
Makehtmlhelp.bat	*Projname*	Source Files	Used by the system to build the Help project when the project is compiled.
Afxcore.htm	*Projname*\hlp	HTML Help Topics	Contains the standard help topics for standard MFC commands and screen objects. Add your own help topics to this file.
Afxprint.htm	*Projname*\hlp	HTML Help Topics	Contains the help topics for the printing commands.
*.jpg; *.gif	*Projname*\hlp\Images	Resource Files	Contain images for the different generated help file topics.

See Also

Concepts

File Types Created for Visual C++ Projects

Help Files (WinHelp)

The following files are created when you add the WinHelp type of Help support to your application by selecting the **Context-sensitive help** check box and then selecting **WinHelp format** in the Advanced Features page of the MFC Application Wizard.

File name	Directory location	Solution Explorer location	Description
Projname.hpj	*Projname*\hlp	Source Files	The Help project file used by the Help compiler to create your program or control's Help file.
Projname.rtf	*Projname*\hlp	Help Files	Contains template topics that you can edit and information on customizing your .hpj file.
Projname.cnt	*Projname*\hlp	Help Files	Provides the structure for the **Contents** window in Windows Help.
Makehelp.bat	*Projname*	Source Files	Used by the system to build the Help project when the project is compiled.
Print.rtf	*Projname*\hlp	Help Files	Created if your project includes printing support (the default). Describes the printing commands and dialog boxes.
*.bmp	*Projname*\hlp	Resource Files	Contain images for the different generated help file topics.

You can add WinHelp support to an MFC ActiveX Control project by selecting **Generate help files** in the Application Settings tab of the MFC ActiveX Control Wizard. The following files are added to your project when you add Help support to an MFC ActiveX control:

File name	Directory location	Solution Explorer location	Description
Projname.hpj	*Projname*\hlp	Source files	The project file used by the Help compiler to create your program or control's Help file.
Projname.rtf	*Projname*\hlp	Project	Contains template topics that you can edit and information on customizing your .hpj file.
Makehelp.bat	*Projname*	Source Files	Used by the system to build the Help project when the project is compiled.
Bullet.bmp	*Projname*	Resource Files	Used by standard Help file topics to represent bulleted lists.

See Also

Concepts

File Types Created for Visual C++ Projects

Miscellaneous Project Files

The following files are created when you create a project with certain options in Visual Studio. The options are described with their related file type:

File name	Directory location	Solution Explorer location	Description
Projname.lic	*Projname*	Not displayed in Solution Explorer	The user license file created for a control project. This file must be present in the same directory as the program or DLL to allow an instance of the control to be created in a design-time environment. Typically, you distribute this file with a control, but your customers do not distribute it.
Projname.reg	*Projname*	Not displayed in Solution Explorer	The project's registration file. This file is created in two cases: • You selected any Automation server option or an Automation option. • You selected a document file extension (one of the options available in the **Advanced Options** dialog box). The file demonstrates the kind of registration settings the framework sets for you.
ITolbar.bmp	*Projname*\res	Resource Files	This file is created only if you have chosen any Automation server support and have also chosen the **Dockable Toolbar** option. The file contains tiled images for the toolbar when the server application is in-place activated inside a container application. The file is similar to the standard Toolbar.bmp, except that many nonserver commands are removed.

See Also

Concepts

File Types Created for Visual C++ Projects

How to: Create ASP.NET Web Services

To create an XML Web service with ASP.NET, use the ASP.NET Web Service template, which is available from the New Project dialog box.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

To create an ASP.NET Web service

1. On the **File** menu, click **New**, and then click **Project**.

   The New Project dialog box appears.

2. In the **Project types** pane, click the **Visual C++** node. In the **Templates** pane, click the ASP.NET Web Service icon.

3. In the **Name** box, enter a unique name for your application.

Note
You can also specify other project and solution settings from the New Project dialog box, but these settings are not required.

4. Click **OK**.

See Also

Concepts

Debugging Preparation: XML Web Services (C++)
Visual C++ Projects

ASP.NET Web Service Template (C++)

Use the ASP.NET Web Service template to author an XML Web service that can be consumed by other Web services or applications on a network. For more information, see XML Web Services in Managed Code.

Requirements

You must have:

- A computer with Internet Information Services (IIS) installed.
- If you want to use FrontPage Server Extensions as your access method, the server extensions must also be installed on that computer.
- You must have the correct Web permissions settings for your application.

Files and References Added by the Template

The template automatically adds the essential project references and files to use as a starting point for your application:

- References to these .NET Framework namespaces:
  - System.EnterpriseServices - Contains classes that provide infrastructure for enterprise applications.
  - System.XML - Contains classes that provide standards-based support for processing XML.
  - System.Web.Services - Contains classes that enable you to build and use XML Web services, programmable entities residing on a Web server and exposed via standard Internet protocols.
  - System.Web - Contains classes and interfaces that enable browser/server communication.
  - System.Data - Contains classes that constitute the ADO.NET architecture, which is the primary data access method for managed applications.
  - System - Contains fundamental classes and base classes that define commonly used values and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
  - mscorlib - The assembly DLL that provides .NET Framework support.
- Source files:
  - Service (.cpp file) - The main source file and entry point into the application that Visual Studio created for you. Identifies the project .dll file and the project namespace. Provide your own code in this file.
  - Service (.asmx file) - A text file that references managed classes that encapsulate the functionality of the XML Web service.
  - AssemblyInfo.cpp - The file that contains information (that is, attributes, files, resources, types, versioning information, signing information, and so on) for modifying the project's assembly metadata. For more information, see Assembly Concepts.
  - Stdafx.cpp - Used to build a precompiled header file named Win32.pch and a precompiled types file named StdAfx.obj.
- Header files:
  - Service (.h file) - The main include file for the project, which contains all declarations, global symbols, and **#include** directives for other header files.
  - Service (.resx file) - An XML resource file whose **BuildAction** property is set to **Embedded Resource**.
  - Stdafx.h - Used to build a precompiled header file named Win32.pch and a precompiled types file named StdAfx.obj.
  - resource.h - A generated include file for app.rc.
- Resource files:
  - app.rc - The resource script file of a program.

    - app.ico -The icon file of a program.
- ReadMe.txt - A file describing each file in your project using the actual filenames created by the template.
- Web.config – A file with application-specific settings. For more information, see Application Configuration Files.
- Global.asax – A file with code for responding to application-level events raised by ASP.NET. For more information, see Global.asax Syntax.

See Also

Tasks

How to: Create ASP.NET Web Services

How to: Create ASP.NET Web Service Projects

Reference

Files Created for CLR Projects

Concepts

Introduction to Programming XML Web Services in Managed Code

Creating an ATL Project

The easiest way to create an ATL project is to use the ATL Project Wizard, located in the Win32 Projects folder of the New Project dialog box.

To create an ATL project using the ATL Project Wizard

1. Follow the instructions in the topic Creating a Project with a Visual C++ Application Wizard.
2. Select the **ATL Project** icon in the Templates pane to open the ATL Project Wizard.
3. Define your application settings using the Application Settings page of the ATL Project Wizard.

Note
Skip this step to keep the wizard default settings.

4. Click **Finish** to close the wizard and open your new project in the development environment.

Once your project is created, you can view the files created in Solution Explorer. For more information about the files the wizard creates for your project, see the project-generated file ReadMe.txt. For more information about the file types, see File Types Created for Visual C++ Projects. For more information about the configurations for the new ATL project, and how to change them, see Default ATL Project Configurations.

See Also

Reference

Property Pages (C++)

Concepts

Adding Functionality with Code Wizards

Other Resources

Windows Installer Deployment Tasks

ATL Project Wizard

The Active Template Library (ATL) is a set of template-based C++ classes that simplify writing small and fast COM objects. The ATL Project Wizard creates a project with the structures to contain COM objects.

Overview

This wizard page describes the current application settings for the ATL project you are creating. By default, the project has the following settings:

- Dynamic-link library Specifies that your server is a DLL and therefore an in-process server.
- Attributed Specifies that your project uses attributes.

To change these defaults, click **Application Settings** in the left column of the wizard and make changes in that page of the ATL Project Wizard.

For information on the default project settings, including the choice of character set, and linking defaults, see Default ATL Project Configurations.

After you create an ATL project, you can add objects or controls to your project using Visual C++ code wizards. You can make the following types of enhancements to a basic ATL project using code wizards:

- Add object and controls to an ATL project
- Add a new interface in an ATL project
- Add a COM+ 1.0 component to an ATL project

Additionally, consider these tasks when you create and enhance an ATL project:

- Make an ATL object noncreatable
- Optimize the compiler for an ATL project

You can specify project properties (for example, whether to link statically to the CRT) in the Project Properties page, and you can set build configurations for an ATL project. See Using Projects to Build Applications for more information.

See Also

Tasks

Creating Projects with Application Wizards

Reference

Fundamentals of ATL COM Objects

ATL Tutorial

Concepts

Visual C++ Projects

Programming with ATL and C Run-Time Code

Other Resources

Creating and Managing Visual C++ Projects

Application Settings, ATL Project Wizard

Use the **Application Settings** page of the ATL Project Wizard to design and add basic features to a new ATL project.

Attributed

This option is not selected by default. It specifies to create an ATL project that uses attributes. When you select this option, the module attribute is inserted into your project's .cpp file. This attribute automatically implements DllMain, DllRegisterServer, DllUnregisterServer, DllGetClassObject, and DllCanUnloadNow. The ATL Project Wizard generates two projects: a framework ATL DLL project with attribute support (called *ProjName*) and a proxy/stub project (called *ProjNamePS*).

An attributed project does not allow support for MFC or merging of proxy/stub code.

Other ATL wizards that insert objects and controls into a project use attributed code by default.

See Basic Mechanics of Attributes for more information.

Server type

Choose from one of three server types:

Dynamic-link library (DLL)

Select to create an in-process server.

Executable (EXE)

Select to create a local out-of-process server. This option does not allow support for MFC or COM+ 1.0. It does not allow for the merging of proxy/stub code.

Service (EXE)

Select to create a Windows application that runs in the background when Windows starts. This option does not allow support for MFC or COM+ 1.0 or does not allow for the merging of proxy/stub code.

Additional options

Note
All additional options are available for DLL projects only.

Allow merging of proxy/stub code

Select the **Allow merging of proxy/stub code** check box as a convenience when marshaling interfaces is required. This option places the MIDL-generated proxy and stub code in the same DLL as the server.

Support MFC

Select to specify that your object includes MFC support. This option links your project to the MFC libraries so that you can access any of the classes and functions they contain.

Support COM+ 1.0

Select to modify the project build settings to support COM+ 1.0 components. In addition to the standard list of libraries, the wizard adds the COM+ 1.0 component-specific library comsvcs.lib

In addition, the mtxex.dll is delay loaded on the host system when your application is launched.

- **Support component registrar** If your ATL project contains support for COM+ 1.0 components, you can set this option. The component registrar allows your COM+ 1.0 object to get a list of components, register components, or unregister components (individually or all at once).

See Also

Tasks

Creating an ATL Project

Reference

ATL Project Wizard

Default ATL Project Configurations

Default ATL Project Configurations

This topic compares the default ATL project configurations in Visual C++ .NET with the default project configurations in Visual C++ 6.0.

Visual C++ .NET Default Configurations

In Visual C++ .NET, the ATL Project Wizard creates two project configurations by default.

Visual C++ .NET Configurations

Configuration	Character set	Use of ATL	Minimize CRT
Release	MBCS	DLL	No
Debug	MBCS	DLL	No

Character set, **Use of ATL** and **Minimize CRT** can all be changed in the **Project Settings** dialog under the **General** tab. You can also add your own configurations using the Configuration Manager. For details, see Build Configurations.

Using **_ATL_MIN_CRT** introduces repercussions in terms of exception handling and use of CStrings. See the following topic for more information: Exception Handling, CString and _ATL_MIN_CRT

Version 6.0 Default Configurations

In Visual C++ version 6.0, the ATL COM AppWizard (now called the ATL Project Wizard) created six project configurations by default. The configurations were variations on Release, Debug, Unicode, and use of CRT and ATL settings. All these configurations can be duplicated in Visual C++ .NET using the Configuration Manager, if so desired.

Version 6.0 Configurations

Configuration	Character set	Use of ATL	Minimize CRT
Debug	MBCS	Static	No
Debug Unicode	UNICODE	Static	No
Release Min Dependency	MBCS	Static	Yes
Release Min Dependency Unicode	UNICODE	Static	Yes
Release Min Size	MBCS	DLL	Yes
Release Min Size Unicode	UNICODE	DLL	Yes

See Also

Reference

Modifying Project Settings

Configuration Manager Dialog Box

Concepts

Programming with ATL and C Run-Time Code

Other Resources

Building in Visual Studio

MFC Support in ATL Projects

If you select **Support MFC** in the ATL Project Wizard, your project declares the application as an MFC application object (class). The project initializes the MFC library and instantiates a class (class *ProjName*) that is derived from CWinApp.

This option is available for nonattributed ATL DLL projects only.

```
class CProjNameApp : public CWinApp
{
public:

// Overrides
   virtual BOOL InitInstance();
   virtual int ExitInstance();
   DECLARE_MESSAGE_MAP()
};

BEGIN_MESSAGE_MAP(CProjNameApp, CWinApp)
END_MESSAGE_MAP()

CProjNameApp theApp;

BOOL CProjNameApp::InitInstance()
{
   return CWinApp::InitInstance();
}

int CProjNameApp::ExitInstance()
{
   return CWinApp::ExitInstance();
}
```

You can view the application object class and its **InitInstance** and **ExitInstance** functions in Class View.

See Also

Tasks

Creating an ATL Project

Reference

Default ATL Project Configurations

Concepts

Adding a Class

COM+ 1.0 Support in ATL Projects

You can use the ATL Project Wizard to create a project with basic support for COM+ 1.0 components.

COM+ 1.0 is designed for developing distributed component-based applications. It also provides a run-time infrastructure for deploying and managing these applications.

If you select the **Support COM+ 1.0** check box, the wizard modifies the build script in the link step. Specifically, the COM+ 1.0 project links to the following libraries:

- comsvcs.lib
- Mtxguid.lib

If you select the **Support COM+ 1.0** check box, you can also select **Support component registrar**. The component registrar allows your COM+ 1.0 object to get a list of components, register components, or unregister components (individually or all at once).

See Also

Reference

Fundamentals of ATL COM Objects

Default ATL Project Configurations

Concepts

Programming with ATL and C Run-Time Code

Adding Objects and Controls to an ATL Project

You can use one of the ATL code wizards to add an object or a control to your ATL- or MFC-based projects. For each COM object or control you add, the wizard generates .cpp and .h files, as well as an .rgs file for script-based registry support. The following ATL code wizards are available in Visual Studio:

ATL Simple Object	ATL Dialog	ATL Performance Monitor
ATL Control	ATL Server Web Service	ATL Active Server Page Component
ATL OLE DB Consumer	Add ATL Support to MFC	ATL COM+ 1.0 Component Wizard
ATL OLE DB Provider	ATL Property Page	

Note

Before adding an ATL object to your project, you should review the details and requirements for the object in its related Help topics.

To add an object or a control using the ATL Control Wizard

1. In Solution Explorer, right-click the project node and click **Add** from the shortcut menu. Click **Add Class**.

   The Add Class dialog box appears.

2. With the ATL folder selected in the Categories pane, select an object to insert from the Templates pane. Click **Open**. The code wizard for the selected object appears.

   Note

   If you want to add an ATL object to an MFC project, you must add ATL support to the existing project. You can do this b y following the instructions in Adding ATL Support to Your MFC Project.

   Alternately, if you attempt to add an ATL object to your MFC project without previously adding ATL support, Visual Studio prompts you to specify whether you want ATL support added to your project. Click **Yes** to add ATL support to the project and open the selected ATL wizard.

See Also

Tasks

Creating Projects with Application Wizards

Reference

ATL Project Wizard

Fundamentals of ATL COM Objects

Default ATL Project Configurations

Concepts

Visual C++ Projects

Programming with ATL and C Run-Time Code

Specifying Compiler Optimization for an ATL Project

By default, the ATL Control Wizard generates new classes with the ATL_NO_VTABLE macro, as follows:

```
class ATL_NO_VTABLE CProjName
{
   ...
};
```

ATL then defines _ATL_NO_VTABLE as follows:

```
#ifdef _ATL_DISABLE_NO_VTABLE
   #define ATL_NO_VTABLE
#else
   #define ATL_NO_VTABLE __declspec(novtable)
#endif
```

If you do not define _ATL_DISABLE_NO_VTABLE, the ATL_NO_VTABLE macro expands to `declspec(novtable)`. Using `declspec(novtable)` in a class declaration prevents the vtable pointer from being initialized in the class constructor and destructor. When you build your project, the linker eliminates the vtable and all functions to which the vtable points.

You must use ATL_NO_VTABLE, and consequently `declspec(novtable)`, with only base classes that are not directly creatable. You must not use `declspec(novtable)` with the most-derived class in your project, because this class (usually CComObject, CComAggObject, or CComPolyObject) initializes the vtable pointer for your project.

You must not call virtual functions from the constructor of any object that uses `declspec(novtable)`. You should move those calls to the FinalConstruct method.

If you are unsure whether you should use the `declspec(novtable)` modifier, you can remove the ATL_NO_VTABLE macro from any class definition, or you can globally disable it by specifying

```
#define _ATL_DISABLE_NO_VTABLE
```

in stdafx.h, before all other ATL header files are included.

See Also

Tasks

Creating Projects with Application Wizards

Reference

ATL Project Wizard

Fundamentals of ATL COM Objects

novtable

Default ATL Project Configurations

Concepts

Visual C++ Projects

Programming with ATL and C Run-Time Code

Adding a New Interface in an ATL Project

When you add an interface to your object or control, you create stubbed-out functions for each method in that interface. In your object or control, you can add only interfaces currently found in an existing type library. Also, the class in which you add the interface must implement the BEGIN_COM_MAP macro or, if the project is attributed, it must have the `coclass` attribute.

You can add a new interface to your control in one of two ways: manually or using code wizards in Class View.

To use code wizards in Class View to add an interface to an existing object or control

1. In Class View, right-click the class name of a control. For example, a full control or composite control, or any other control class that implements a BEGIN_COM_MAP macro in its header file.
2. On the shortcut menu, click **Add**, and then click **Implement Interface**.
3. Select the interfaces to implement in the Implement Interface Wizard. If the interface does not exist in any available typelib, then you must add it manually to the .idl file.

To add a new interface manually

1. Add the definition of your new interface to the .idl file.
2. Derive your object or control from the interface.
3. Create a new COM_INTERFACE_ENTRY for the interface or, if the project is attributed, add the `coclass` attribute.
4. Implement methods on the interface.

See Also

Tasks

Creating Projects with Application Wizards

Reference

ATL Project Wizard
Fundamentals of ATL COM Objects
Default ATL Project Configurations

Concepts

Visual C++ Projects
Programming with ATL and C Run-Time Code

Making an ATL Object Noncreatable

You can change the attributes of an ATL-based COM object so that a client cannot directly create the object. In this case, the object would be returned through a method call on another object rather than created directly.

To make an object noncreatable

1. Remove the OBJECT_ENTRY_AUTO for the object. If you want the object to be noncreatable but the control to be registered, replace OBJECT_ENTRY_AUTO with OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO.
2. Add the noncreatable attribute to the coclass in the .idl file. For example:

```
[
   uuid(A1992E3D-3CF0-11D0-826F-00A0C90F2851),
   helpstring("MyObject"),
  noncreatable
]
coclass MyObject
{
   [default] interface IMyInterface;
}
```

See Also

Tasks

Creating Projects with Application Wizards

Reference

ATL Project Wizard

Fundamentals of ATL COM Objects

Default ATL Project Configurations

Concepts

Visual C++ Projects

Programming with ATL and C Run-Time Code

Creating an ATL Server Project

The easiest way to create an ATL Server project is to use the ATL Server Project Wizard.

To create an ATL Server project using the ATL Server Project Wizard

1. Follow the instructions in the help topic Creating a Project with a Visual C++ Application Wizard.

   Note When prompted, you must select either the **ATL Server Project** or the **ATL Server Web Service** icon in the Templates pane for the ATL Server Project Wizard to open.

   Note Deployment of ATL Server projects to IIS is not supported on a Web server running on any 64-bit platform.

2. Define your application settings using the ATL Server Project Wizard or skip this step to keep the wizard's default settings.

3. Click **Finish** to close the wizard, and your new project will open in the development environment.

See Also

Reference

Property Pages (C++)

Concepts

Adding Functionality with Code Wizards

Other Resources

Windows Installer Deployment Tasks

ATL Server Project Wizard

The ATL Server Project Wizard provides an easy way to generate Web application or XML Web service projects that use ATL Server, a set of unmanaged C++ classes designed for such applications.

The ATL Server Project Wizard has two icons in the Templates pane of the **New Project** dialog box: **ATL Server Project** and **ATL Server Web Service**. The wizard is the same in both cases, but the default settings are different.

After the wizard generates the files and structure for the project, you can add classes, objects, controls, or resources using the programming assistants.

Overview

This wizard page describes the current application settings for the ATL Server project that you are creating. By default, the wizard creates a project as follows:

- Project Settings
  - The project is created with a Web application DLL and an ISAPI extension DLL. Both projects will be deployed to a virtual directory with the same name as the solution.
- Server Options
  - The project is created without additional support options for the ISAPI extension DLL.
- Application Options
  - The project is created with validation support and stencil processing support.
- Developer Support Options
  - The project is created with TODO comments, attributed code, and custom assert and trace handling support.

To change these defaults, click the tab titles in the left column of the wizard and make the desired changes on those pages.

After you create your ATL Server project, you can add objects or controls to your project using Visual C++ code wizards.

See Also

Tasks

Creating an ATL Server Project
Creating Projects with Application Wizards

Concepts

Visual C++ Projects

Other Resources

ATL Server Concepts
Creating and Managing Visual C++ Projects

Project Settings, ATL Server Project Wizard

Choose the project types to be generated along with their names and locations. Use this page to generate up to two projects: Web application DLL and ISAPI extension DLL.

Generate Web application DLL

Generates a Web application DLL that contains a request handler. The name and location of this project is taken from the information provided to the **New Project** dialog box. You can further customize the Web application DLL project on the Application Options page.

Generate ISAPI extension DLL

Generates an ISAPI extension DLL that contains the code necessary to dispatch HTTP requests to request handlers in Web application DLLs. The name and location of this project is taken from the text boxes provided on this page. You can further customize the ISAPI extension DLL project on the Server Options page.

ISAPI project name

Provide the name of the generated ISAPI extension DLL project.

ISAPI project directory

Provide the location of the generated ISAPI extension DLL project.

Generate combined DLL

Generates a single DLL project that can act as both an ISAPI extension and a Web application DLL with a request handler. The name and location of this project is taken from the information provided to the **New Project** dialog box.

Deployment support

Enables deployment support for your project. Deployment support allows your project to be installed and registered on your local machine as part of the build process. Deployment requires IIS to be installed on the build machine. If you do not select deployment support in the wizard, you can enable it later using the Web Deployment property page.

Virtual root

Provides the name of the virtual directory in which your project will be installed. If the virtual directory does not exist, the Web deployment tool will create it.

See Also

Reference

ATL Server Project Wizard

Other Resources

ATL Server Concepts

Server Options, ATL Server Project Wizard

Choose the options to be supported by the ISAPI extension DLL. Use this page to add caching support and other services to the extension DLL so that all request handlers routed through that DLL can use them.

Additional support options

Generates code in the ISAPI extension DLL that supports additional services.

Option	Description
Blob cache	Adds code that allows the efficient storage and retrieval of arbitrary data.
File cache	Adds code that allows the efficient storage and retrieval of files and file data.
Data source cache	Adds code that allows the per-thread caching of OLE DB data source connections.
Predefined performance counters	Adds performance counters for monitoring request statistics to the ISAPI extension. The extension DLL must be registered if this option is selected.
Browser capabilities support	Exposes the browser capabilities service from the ISAPI extension.

Session services

Adds support for session-state services to the ISAPI extension, allowing the creation, storage and retrieval of data across requests.

Option	Description
OLE DB-backed session-state services	Persists session data to an OLE DB data source. Suitable for Web farms.
Memory-backed session-state services	Persists session data to memory. Not usually suitable for Web farms without server affinity.

Data source

If **OLE DB-backed session-state services** is selected, specifies an existing database as the store for the session data. The database should conform to the schema contained in \Vc7\bin\SessionServices.sql.

Resource language

Specifies the language for resources generated or included by the wizard.

See Also

Reference

ATL Server Project Wizard

Concepts

Predefined ISAPI Performance Counters

Other Resources

Caching

Session-State Services (ATL Server)

Application Options, ATL Server Project Wizard

Choose the options to be supported by the Web application DLL. Use this page to add support for validation or stencil processing, to specify options for the initial server response file, or to specify whether the Web application DLL should contain an XML Web service SOAP handler or a standard request handler.

Application options

Generates code in the Web application DLL that supports additional services.

Option	Description
Validation support	Inserts the method necessary for validation of query parameters and form variables.
Stencil processing support	Inserts a replacement method that demonstrates how to use the tag_name attribute or REPLACEMENT_METHOD_ENTRY macro.
Create as Web Service	Generates skeleton code for an XML Web service. If this box is not checked a standard request handler is generated instead.

Options for initial server response file (.SRF)

Adds tags to the generated server response file.

Option	Description
Use locale	Select this box and select an item from the list to have the corresponding locale tag added to the generated server response file.
Use codepage	Select this box and select an item from the list to have the corresponding codepage tag added to the generated server response file.

See Also

Reference

ATL Server Project Wizard

Other Resources

ATL Server Concepts

ATL Server Response File Reference

Developer Support Options, ATL Server Project Wizard

Choose the options that configure the generated projects to the way that you work best. Use this page to specify whether you want comments, attributed code, or integration with WebDbg.

Developer support options

Customizes the generated projects.

Option	Description
Generate TODO comments	Inserts TODO comments in the generated code.
Attributed code	Generates code using Visual C++ attributes. If this box is not select, the code is generated without attributes.
Custom assert and trace handling support	Adds a CDebugReportHook object to the project or projects to allow trace and assert statements to be passed to a named pipe and enable debugging using WebDbg.

See Also

Reference

ATL Server Project Wizard

Other Resources

ATL Server Attributes

How to: Create Class Libraries

To create a class library, use the Class Library template, which is available from the New Project dialog box.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

To create a class library

1. On the **File** menu, click **New**, and then click **Project**.

   The New Project dialog box appears.

2. In the **Project Types** pane, click the **Visual C++** node. In the **Templates** pane, click the Class Library icon.

3. In the **Name** box, enter a unique name for your application.

Note
You can also specify other project and solution settings from the **New Project** dialog box, but these settings are not required.

4. Click **OK**.

See Also

Other Resources

Debugging Preparation: Visual C++ Project Types

Class Library Template (C++)

You can use the Class Library template to quickly create reusable classes and components that can be shared with other projects.

The template automatically adds the essential project references and files:

- References to these .NET Framework namespaces:
  - System - Contains fundamental classes and base classes that define commonly used values and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
  - mscorlib - The assembly DLL that provides .NET Framework support.
- Source files:
  - Library (.cpp file) - The main source file and entry point into the application that Visual Studio created for you. Identifies the project .dll file and the project namespace. Provide your own code in this file.
  - AssemblyInfo.cpp - The file that contains information (that is, attributes, files, resources, types, versioning information, signing information, and so on) for modifying the project's assembly metadata. For more information, see Assembly Concepts in the .NET Framework SDK.
  - Stdafx.cpp - Used to build a precompiled header file named Win32.pch and a precompiled types file named StdAfx.obj.
- Header files:
  - Library (.h file) - The main include file for the project, which contains all declarations, global symbols, and **#include** directives for other header files.
  - Stdafx.h - Used to build a precompiled header file named Win32.pch and a precompiled types file named StdAfx.obj.
  - resource.h - A generated include file for app.rc.
- Resource files:
  - app.rc - The resource script file of a program.
  - app.ico -The icon file of a program.
- ReadMe.txt - A file describing each file in your project using the actual filenames created by the template.

See Also

Tasks

How to: Create Class Libraries

How to: Add New Project Items

Reference

Files Created for CLR Projects

Concepts

Item Management in Projects

Other Resources

Managing Solutions, Projects, and Files

How to: Create CLR Console Applications

To create a console application, use the Console Application template, which is available from the New Project dialog box.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

To create a CLR console application

1. On the **File** menu, click **New**, and then click **Project**.

   The New Project dialog box appears.

2. In the **Project types** pane, click the **Visual C++** node. In the **Templates** pane, select the CLR Console Application project template.

3. In the **Name** box, enter a unique name for your application.

Note
You can also specify other project and solution settings from the New Project dialog box, but these settings are not required.

4. Click the **OK** button.

See Also

Concepts

Visual C++ Projects

Other Resources

Debugging Preparation: Visual C++ Project Types

CLR Console Application Template (C++)

The Console Application template adds the necessary items needed to create a console application. Console applications are typically designed without a graphical user interface and are compiled into a stand-alone executable file. A console application is run from the command line with input and output information being exchanged between the command prompt and the running application. Because information can be written to and read from the console window, this makes the console application a great way to learn new programming techniques without having to be concerned with the user interface.

The template automatically adds the essential project references and files to use as a starting point for your application:

- References to these .NET Framework namespaces:
  - System - Contains fundamental classes and base classes that define commonly used values and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
  - mscorlib - The assembly DLL that provides .NET Framework support.
- Source files:
  - Console (.cpp file) - The main source file and entry point into the application that Visual Studio created for you. Identifies the project .dll file and the project namespace. Provide your own code in this file.
  - AssemblyInfo.cpp - The file that contains information (that is, attributes, files, resources, types, versioning information, signing information, and so on) for modifying the project's assembly metadata. For more information, see Assembly Concepts in the .NET Framework SDK.
  - Stdafx.cpp - Used to build a precompiled header file named Win32.pch and a precompiled types file named StdAfx.obj.
- Header files:
  - Stdafx.h - Used to build a precompiled header file named Win32.pch and a precompiled types file named StdAfx.obj.
  - resource.h - A generated include file for app.rc.
- Resource files:
  - app.rc - The resource script file of a program.
  - app.ico -The icon file of a program.
- ReadMe.txt - A file describing each file in your project using the actual filenames created by the template.

See Also

Tasks

How to: Create CLR Console Applications

Reference

Files Created for CLR Projects

Concepts

Item Management in Projects

Other Resources

Managing Solutions, Projects, and Files

Creating a Custom Wizard

The Visual C++ Custom Wizard is the tool to use when you need to create a new custom wizard. The easiest way to create a custom wizard is to use the Custom Wizard.

To create a wizard using the Custom Wizard

1. Follow the instructions in the help topic Creating a Project with a Visual C++ Application Wizard.
2. In the New Project dialog box, select the **Custom Wizard** icon in the Templates pane to open the Custom Wizard.
3. Define your application settings using the Custom Wizard.
4. Click **Finish** to close the wizard and open your new project in the development environment.

After you have created your project, you can view the files created in Solution Explorer. For more information about the files created for the Custom Wizard, see Files Created for Your Wizard.

See Also

Concepts

Designing a Wizard

Custom Wizard

The Visual C++ Custom Wizard is a specific type of wizard, designed to help you create your own wizards. Once configured, your wizard generates the starter files you need for your projects.

The ConsoleAppWiz Sample sample can help you understand how to configure your wizard once you have created it.

Overview

This wizard page describes the current application settings for the Custom Wizard project you are creating. By default, the wizard creates a project as follows:

- The wizard's friendly name is set to the name of the project you indicated in the New Project dialog box.
- The wizard has a user interface.
- The wizard user interface has one page.

You can change these default settings on the Application Settings page of the wizard.

See Also

Tasks

Creating a Custom Wizard

Concepts

Visual C++ Projects

Files Created for Your Wizard

Designing a Wizard

Application Settings, Custom Wizard

Use this page of the Custom Wizard to design a wizard.

Wizard friendly name

Sets the name of the wizard that you are designing. This name is displayed in the ReadMe.txt file and as the title in the HTML file.

User interface

The default. Specifies that the custom wizard displays a user interface. If you want the user to select options specifying what code that your wizard generates, the wizard must have a user interface. The wizard does not need a user interface if its purpose is to generate template code with no user options.

Number of pages

Sets the number of pages your custom wizard requires. This option applies only to a wizard that has a user interface.

See Also

Tasks

Creating a Custom Wizard

Reference

Custom Wizard

Concepts

Files Created for Your Wizard

Designing a Wizard

How to: Create CLR Empty Projects

To create a CLR empty project, use the CLR Empty Project template, which is available from the New Project dialog box.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

To create a CLR empty project

1. On the **File** menu, click **New**, and then click **Project**.

   The New Project dialog box appears.

2. In the **Project types** pane, click the **Visual C++** node. In the **Templates** pane, click the CLR Empty Project icon.

3. In the **Name** box, enter a unique name for your application.

Note
You can also specify other project and solution settings from the **New Project** dialog box, but these settings are not required.

4. Click **OK**.

See Also

Concepts

Visual C++ Projects

Other Resources

Debugging Preparation: Visual C++ Project Types

CLR Empty Project Template (C++)

The Empty Project template can be used when you want to create your own project type. The template creates the necessary file structure needed to store application information. Any references, files, or components must be added manually.

See Also

Tasks

How to: Create CLR Empty Projects

How to: Add New Project Items

Reference

Files Created for CLR Projects

Concepts

Project Items

Creating a Makefile Project

If you have a project that you build from the command line with a makefile, then the Visual Studio development environment will not recognize your project. To open and build your project using Visual Studio, first create an empty project containing the appropriate build settings using the Makefile Project Wizard. You can then use this project to build your project from the Visual Studio development environment.

The project displays no files in Solution Explorer. The project specifies the build settings, which are reflected in the project's property page.

The output file that you specify in the project has no effect on the name that the build script generates; it declares only an intention.

To create a Makefile project

1. Follow the instructions in the help topic Creating a Project with a Visual C++ Application Wizard.
2. In the New Project dialog box, select **Makefile Project** in the Templates pane to open the wizard.
3. In the Application Settings page, provide the command, output, clean, and rebuild information.
4. Click **Finish** to close the wizard and open the newly created project in Solution Explorer.

You can view and edit the project's properties in its property page. See Setting Visual C++ Project Properties for information about displaying the property page.

See Also

Reference

Makefile Project Wizard

Concepts

Special Characters in a Makefile

Contents of a Makefile

Makefile Project Wizard

Use this wizard to create a command line for the build engine to execute. Once you create the project, you can view and edit the project's properties in the project's property page.

Overview

This wizard page describes the current project settings for the project you are creating. By default, the project is set up to build both debug and release configurations of *ProjName*.exe.

To change these defaults, click the Application Settings tab in the left column of the wizard and make the desired changes.

See Also

Tasks

Creating a Makefile Project

Concepts

NMAKE Reference

Special Characters in a Makefile

Contents of a Makefile

Application Settings, Makefile Project Wizard

Provide the information for the project. The options you specify are reflected in the project's property page. Once you create the project, you can view and edit each of the following options in the **Nmake** page of the project's property page.

Build command line

Specifies the command line to run when the user selects **Build** from the **Build** menu. Displayed in the **Build command line** field on the **Nmake** page of the project's property page.

Output

Specifies the name of the file that will contain the output for the command line. By default, this option is based on the project name. Displayed in the **Output** field on the **Nmake** page of the project's property page.

Clean commands

Specifies the command line to run when the user selects **Clean** from the **Build** menu. Displayed in the **Clean command line** field on the **Nmake** page of the project's property page.

Rebuild command line

Specifies the command line to run when the user selects **Rebuild** from the **Build** menu. Displayed in the **Rebuild all command line** field on the **Nmake** page of the project's property page.

See Also

Tasks

Creating a Makefile Project
Creating Projects with Application Wizards

Reference

Makefile Project Wizard
Property Pages (C++)

Concepts

Visual C++ Projects

Other Resources

Windows Installer Deployment Tasks

How to: Enable IntelliSense for Makefile Projects

IntelliSense fails to operate in the IDE for Visual C++ makefile projects when certain project settings, or compiler options, are set up incorrectly. Use this procedure to configure Visual C++ makefile projects, so that IntelliSense works when makefile projects are open in the Visual Studio development environment.

To enable IntelliSense for makefile projects in the IDE

1. Open the **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Expand the **Configuration Properties** node.
3. Select the **NMake** property page, and then modify properties under **IntelliSense** as appropriate.
   - Set the **Common Language Runtime Support** property for projects (or files) that contain managed code. See /clr (Common Language Runtime Compilation), for more information.
   - Set the **Preprocessor Definitions** property to define any preprocessor symbols in your makefile project. See /D (Preprocessor Definitions), for more information.
   - Set the **Additional Include Directories** property to specify the list of directories that the compiler will search to resolve file references that are passed to preprocessor directives in your makefile project. See /I (Additional Include Directories), for more information.

     For projects that are built using CL.EXE from a Command Window, set the **INCLUDE** environment variable to specify directories that the compiler will search to resolve file references that are passed to preprocessor directives in your makefile project.
   - Set the **Forced Includes** property to specify which header files to process when building your makefile project. See /FI (Name Forced Include File), for more information.
   - Set the **Assembly Search Path** property to specify the list of directories that the compiler will search to resolve references to .NET assemblies in your project. See /AI (Specify Metadata Directories), for more information.
   - Set the **Forced Using Assemblies** property to specify which .NET assemblies to process when building your makefile project. See /FU (Name Forced #using File), for more information.
4. Click **OK** to close the property pages.
5. Use the **Save All** command to save the modified project settings.

Note
In order for IntelliSense to work, you must close the solution that contains your makefile project and then delete any pr eviously generated .ncb files.

The next time you open your makefile project in the Visual Studio development environment, run the **Clean Solution** command and then the **Build Solution** command on your makefile project. IntelliSense should work properly in the IDE.

See Also

Tasks

How to: Create a C++ Project from Existing Code

Concepts

NMAKE Reference

Other Resources

Using IntelliSense

Creating an MFC ActiveX Control

ActiveX control programs are modular programs designed to give a specific type of functionality to a parent application. For example, you can create a control such as a button for use in a dialog, or toolbar for use in a Web page.

The easiest way to create an MFC ActiveX control is to use the MFC ActiveX Control Wizard.

To create an MFC ActiveX Control using the MFC ActiveX Control Wizard

1. Follow the instructions in the help topic Creating a Project with a Visual C++ Application Wizard.
2. In the New Project dialog box, select the **MFC ActiveX Control** icon in the Templates pane to open the MFC ActiveX Control Wizard.
3. Define your application settings, control names, and control settings using the MFC ActiveX Control Wizard.

Note
Skip this step to keep the wizard default settings.

4. Click **Finish** to close the wizard and open your new project in the development environment.

After you have created your project, you can view the files created in Solution Explorer. For more information about the files the wizard creates for your project, see the project-generated file ReadMe.txt. For more information about the file types, see File Types Created for Visual C++ Projects.

After you have created your project, you can use the code wizards to add functions, variables, events, properties, and methods. For more information about customizing your ActiveX control, see MFC ActiveX Controls.

See Also

Reference

Property Pages (C++)

Concepts

Adding Functionality with Code Wizards

Other Resources

Windows Installer Deployment Tasks

MFC ActiveX Control Wizard

An ActiveX control is a specific type of automation server; it is a reusable component. The application hosting the ActiveX control is the automation client of that control. If your goal is to create such a reusable component, then use this wizard to create your control. See MFC ActiveX Controls for more information.

Alternately, you can create an automation server MFC application using the MFC Application Wizard.

An ActiveX control created with this wizard can have a user interface, or it can be invisible. You can indicate this option in the Control Settings page in the wizard. A timer control is an example of an ActiveX control that you would want to be invisible.

ActiveX controls can have a complex user interface. Some controls might be like encapsulated forms: a single control containing many fields, each a Windows control in its own right. For example, an auto parts object implemented as an MFC ActiveX control might present a form-like user interface through which users could read and edit the part number, part name, and other information. See MFC ActiveX Controls for more information.

If you need to create a container for your ActiveX objects, see Create an ActiveX Control Container.

The MFC starter program includes C++ source (.cpp) files, resource (.rc) files, and a project (.vcproj) file. The code generated in these starter files is based on MFC.

The following sample list shows tasks and types of enhancements for your ActiveX control:

- Optimizing an ActiveX Control
- Adding Stock Events to an ActiveX Control
- Adding Custom Events
- Adding Stock Methods
- Adding Custom Methods
- Adding Stock Properties
- Adding Custom Properties
- Programming ActiveX Controls in an ActiveX Control Container

Overview

This wizard page describes the current application settings for the MFC ActiveX control project you are creating. By default, the wizard creates a project as follows:

- The default project generates no run-time license or help files. You can change these default settings on the Application Settings page. Only the selections you make on this page of the ActiveX Control Wizard are reflected on the **Overview** page.
- The project includes a control class and a property page class, based on the name of the project. You can edit the names of your project and file names on the Control Names page.
- The control is based on no existing Windows control, activates when it becomes visible, has a user interface, and includes an **About** dialog box. You can change these default settings on the Control Settings page.

See Also

Concepts

Visual C++ Projects

Other Resources

Creating and Managing Visual C++ Projects

ATL Concepts

Application Settings, MFC ActiveX Control Wizard

Use this page of the MFC ActiveX Control Wizard to design and add basic features to a new MFC ActiveX project. These settings apply to the application itself and not to any specific feature or element of the control.

Run-time license

Select this option to generate a user license file to distribute with the control. The license is a text file, *projname*.lic. This file must be in the same directory as the control's DLL to allow an instance of the control to be created in a design-time environment. You usually distribute this file with your control, but your customers do not distribute it.

Generate help files

Select this option to generate stubbed help files and configure the project to include help for your control. A default project, created without this option, generates only an **About** box that is displayed when the user right clicks the control, uses F1, or clicks **Help** on the control's container.

Note
How help is displayed depends on how your control interacts with its container. If you include help with your container, you must handle messages between the control and the container to display the help appropriately. See Adding Control Help in a Dialog Box and Message-Map Support for more information.

When you generate help files using the wizard, your project includes the following:

- The file .vcproj contains code to build and configure the help file when the project is built.
- The file *projnamePropPage*.cpp file includes a SetHelpInfo function in the constructor.
- The file projname.hpj, is the help project file used by the help compiler to create the ActiveX control's help file. The .hpj file is a text file containing the information about building your help file and the paths to the additional files (for example, bitmaps) the help file includes.
- The project includes the HLP directory to contain the project help bitmap files and the help topic file (*projname*.rtf). This help topic file contains the standard help topics for the common properties, events, and methods supported by many ActiveX controls. You can edit the .rtf file to add or remove specific help topics.

See Also

Reference

MFC ActiveX Control Wizard
Control Names, MFC ActiveX Control Wizard
Control Settings, MFC ActiveX Control Wizard

Control Names, MFC ActiveX Control Wizard

Specify the names for the control class and property page class, the type names, and type identifiers for your control. With the exception of **Short name**, all other fields can be edited independently. If you change the text for **Short name**, the change is reflected in the names of all other fields in this page. This naming behavior is designed to make all the names easily identifiable for you as you develop your control.

Short name

Provide an abbreviated name for the control. By default, this name is based on the project name you provided in the New Project dialog box. The name you provide determines the class names, the type names, and the type identifiers, unless you change those fields individually.

Control class name

By default, the name of the control class is based on the short name, with `C` as a prefix and `Ctrl` as a suffix. For example, if your control's short name is `Price`, the control class name is `CPriceCtrl`.

Control .h file

By default, the name of the header file is based on the short name, with `Ctrl` as a suffix and `.h` as the file extension. For example, if your control's short name is `Price`, the header file name is `PriceCtrl.h`. The name in this field should match the control class name.

Control .cpp file

By default, the name of the header file is based on the short name, with `Ctrl` as a suffix and `.cpp` as the file extension. For example, if your control's short name is `Price`, the header file name is `PriceCtrl.cpp`. The name in this field should match the header name.

Control type name

By default, the name of the control type is based on the short name, followed by `Control`. For example, if your control's short name is `Price`, the control class type name is `Price Control`. If you change the value in this field, make sure the name indicates an inheritance.

Control type ID

Sets the control type ID of the control class. The control writes this string to the registry when it is added to a project. Container applications use this string to create an instance of the control.

By default, the control type ID is based on the project name, which you indicated in the **New Project** dialog box, and the short name. This name should match the type name.

By default, the control type ID appears as follows:

ProjectName.*ShortName*Ctrl.1

If you change the short name in this dialog box, the control type ID appears as follows:

ProjectName.*NewShortName*Ctrl.1

PropPage class name

By default, the name of the property page class is based on the short name, with `C` as a prefix and `PropPage` as a suffix. For example, if your control's short name is `Price`, the property page class name is `CPricePropPage`. This name should match the control class name, appended with `PropPage`.

PropPage .h file

By default, the name of the property page header file is based on the short name, with as a `PropPage` as a suffix and `.h` as the file extension. For example, if your control's short name is `Price`, the property page header file name is `PricePropPage.h`. This name should match the class name.

PropPage .cpp file

By default, the name of the property page implementation file is based on the short name, with as a `PropPage` as a suffix and `.cpp` as the file extension. For example, if your control's short name is `Price`, the property page header file name is `PricePropPage.cpp`. This name should match the header file name.

PropPage type name

By default, the property page type name is based on the short name, followed by `Property Page`. For example, if your control's short name is `Price`, the property page type name is `Price Property Page`. If you change the value in this field, make sure the name indicates the control class.

PropPage type ID

Sets the ID of the property page class. The control writes this string in the registry when it is applied to a project. A container application uses this string to create an instance of the control's property page.

By default, the property page type ID is based on the project name, which you indicated in the **New Project** dialog box, and the short name. This name should match the type name.

By default, the property page type ID appears as follows:

ProjectName.*ShortName*PropPage.1

If you change the short name in this dialog box, the property page type ID appears as follows:

ProjectName.*NewShortName*PropPage.1

See Also

Reference

MFC ActiveX Control Wizard
Application Settings, MFC ActiveX Control Wizard
Control Settings, MFC ActiveX Control Wizard

Concepts

File Types Created for Visual C++ Projects

Control Settings, MFC ActiveX Control Wizard

Insert "Search Results" summary here.

Specify the names for the control class and property page class, the type names, and type identifiers for your control. With the exception of **Short name**, all other fields can be edited independently. If you change the text for **Short name**, the change is reflected in the names of all other fields in this page. This naming behavior is designed to make all the names easily identifiable for you as you develop your control.

Short name

Provide an abbreviated name for the control. By default, this name is based on the project name you provided in the New Project Dialog Box. The name you provide determines the class names, the type names, and the type identifiers, unless you change those fields individually.

Control class name

By default, the name of the control class is based on the short name, with `C` as a prefix and `Ctrl` as a suffix. For example, if your control's short name is `Price`, the control class name is `CPriceCtrl`.

Control .h file

By default, the name of the header file is based on the short name, with `Ctrl` as a suffix and `.h` as the file extension. For example, if your control's short name is `Price`, the header file name is `PriceCtrl.h`. The name in this field should match the control class name.

Control .cpp file

By default, the name of the .cpp file is based on the short name, with `Ctrl` as a suffix and `.cpp` as the file extension. For example, if your control's short name is `Price`, the .cpp file name is `PriceCtrl.cpp`. The name in this field should match the header name.

Control type name

By default, the name of the control type is based on the short name, followed by `Control`. For example, if your control's short name is `Price`, the control class type name is `Price Control`. If you change the value in this field, make sure the name indicates an inheritance.

Creating an MFC ActiveX Control Container

An ActiveX control container is a parent program that supplies the environment for an ActiveX (formerly OLE) control to run. You can create an application capable of containing ActiveX controls with or without MFC, but it is much easier to do with MFC.

Creating an MFC container program using the MFC Application Wizard allows you to access the many features of ActiveX controls and Automation that are implemented by the MFC and ActiveX classes. These features include visual editing, Automation, creating compound files, and support for controls. The MFC Application Wizard visual editing options that your parent program will support include creating a container, a mini-server, a full-server, and a program that is both a container and a server.

- **New MFC Application**. To create a new MFC program that includes Automation, visual editing, compound files, or control support, use the MFC Application Wizard and choose the appropriate Automation options.
- **Existing MFC Application**. If you are adding control containment to an existing MFC application, see OLE Control Containers: Manually Enabling OLE Control Containment.

To create an ActiveX container for any of the following types of applications

1. Containers
2. Visual editing
3. MFC ActiveX controls

See Also

Concepts

Visual C++ Projects

Creating an MFC Application

An MFC application is an executable application for Windows that is based on the Microsoft Foundation Class (MFC) Library. The easiest way to create an MFC application is to use the MFC Application Wizard.

MFC executables generally fall into five types: standard Windows applications, dialog boxes, forms-based applications, Explorer-style applications, and Web browser–style applications. For more information, see:

- Using the Classes to Write Windows Applications
- Creating and Displaying Dialog Boxes
- Creating a Forms-Based MFC Application
- Creating a Windows Explorer-Style MFC Application
- Creating a Web Browser-Style MFC Application

The MFC Application Wizard generates the appropriate classes and files for any of these types of applications, depending on the options you select in the wizard.

To create an MFC application using the MFC Application Wizard

1. Follow the instructions in the help topic Creating a Project with a Visual C++ Application Wizard.
2. In the New Project dialog box, select **MFC Application** in the Templates pane to open the wizard.
3. Define your application settings using the MFC Application Wizard.

Note
Skip this step to keep the wizard default settings.

4. Click **Finish** to close the wizard and open your new project in the development environment.

Once your project is created, you can view the files created in Solution Explorer. For more information about the files the wizard creates for your project, see the project-generated file ReadMe.txt. For more information about the file types, see File Types Created for Visual C++ Projects.

See Also

Tasks

Debugging Preparation: Windows Forms Applications (.NET)

Reference

Property Pages (C++)

Concepts

Adding Functionality with Code Wizards

Other Resources

Windows Installer Deployment Tasks

MFC Application Wizard

The MFC Application Wizard generates an application having built-in functionality that, when compiled, implements the basic features of a Windows executable (.exe) application. The MFC starter application includes C++ source (.cpp) files, resource (.rc) files, header (.h) files, and a project (.vcproj) file. The code generated in these starter files is based on MFC.

Note
Depending on the options you select, the wizard creates additional files in your project. For example, if you select **Context-sensitive help** in the Advanced Features page, the wizard creates the files necessary to compile the project's Help files. For more information on the files created by the wizard, see File Types Created for Visual C++ Projects, and see the Readme.txt file in the project.

Overview

This wizard page describes the current application settings for the MFC application you are creating. By default, the wizard creates a project as follows:

- **Application Type**
  - The project is created with multiple-document interface (MDI) support.
  - The project uses the document/view architecture.
  - The project is created as an MFC standard (Windows) project.
  - The project uses MFC in a shared DLL.
- **Compound Document Support**
  - The project provides no support for compound documents.
- **Document Template Strings**
  - The project uses the project name for the default document template strings.
- **Database Support**
  - The project provides no support for databases.
- **User Interface Features**
  - The project implements standard Windows user interface features such as a system menu, a status bar, maximize and minimize boxes, an "About" box, a standard docking toolbar, and child frames.
- **Advanced Features**
  - The project supports printing and print preview.
  - The project supports ActiveX controls.
  - The project provides no support for Automation, MAPI, Windows sockets, or Active Accessibility.
- **Generated Classes**
  - The project's view class is derived from CView.
  - The project's application class is derived from CWinApp.
  - The project's document class is derived from CDocument.
  - The project's main frame class is derived from CMDIFrameWnd.
  - The project's child frame class is derived from CMDIChildWnd.

To change these defaults, click the tab titles in the left column of the wizard and make the desired changes on those pages.

After you create an MFC application project, you can add objects or controls to your project using Visual C++ code wizards.

See Also

Tasks

Creating an MFC Application

Concepts

Using the Classes to Write Applications for Windows

Other Resources

MFC Reference

Application Type, MFC Application Wizard

Use this page of the MFC Application Wizard to design and add basic features to a new MFC project.

Application type

Indicate the type of document support you want to create in your application. The type of application you select determines the user interface options available for your application. See User Interface Features, MFC Application Wizard for more information.

For more information about the types of documents, see:

- SDI and MDI
- Frame Windows
- Frame-Window Classes
- Documents, Views, and the Framework
- Dialog Boxes

Option	Description
Single document	Creates a single document interface (SDI) architecture for your application, with a view class based on CView. You can change the base class for the view in the Generated Classes page of the wizard. To create a form-based application, for example, you could use CFormView for the view class. In this type of application, the document's frame window can hold only one document in the frame.
Multiple documents	Creates a multiple document interface (MDI) architecture for your application, with a view class based on **CView**. You can change the base class for the view in the **Generated Classes** page of the wizard. To create a form-based application, for example, you could use **CFormView** for the view class. In this type of application, the document's frame window can hold multiple child windows in the frame.
Dialog based	Creates a dialog-based architecture for your application, with a dialog class based on **CDialog**. (To create an HTML dialog, select the box **Use HTML dialog**.)
Multiple top-level documents	Creates a multiple top-level architecture for your application, with a view class based on **CView**. In this type of application, when a user clicks **New** (or **New Frame**) on the **File** menu, the application creates a window whose parent is implicitly the desktop. The new document frame appears in the taskbar and is not restricted to the application window's client area.

Project style

Indicates whether your application has a Windows Explorer architecture and display or a standard MFC architecture and display. See Creating a Windows Explorer-Style MFC Application for more information.

Option	Description
Windows Explorer	Implements a Windows Explorer–like application using a splitter window where the left pane is a CTreeView and the right pane is a CListView.
MFC standard	Provides a standard MFC application architecture (default).

Use of MFC

Indicates how to link to the MFC library. By default, MFC is linked as a shared DLL.

Option	Description

Use MFC in a shared DLL	Links the MFC library to your application as a shared DLL. Your application makes calls to the MFC library at run time. This option reduces the disk and memory requirements of your application if it is composed of multiple executable files that use the MFC library. Both Win32 and MFC applications can call functions in your DLL.
Use MFC in a static library	Links your application to the static MFC library at build time.

Document/view architecture support

Includes document/view architecture in your application using the CDocument and CView base classes (default). Clear this check box if you are porting a non-MFC application, for example, or if you want reduce the size of your compiled executable. An application without document/view architecture is derived from CWinApp by default, and it does not include MFC support for opening a document from a disk file.

Use HTML dialog

For dialog box applications only. Derives the dialog class from CDHtmlDialog instead of CDialog. If you check this box, **CDHtmlDialog** is listed in the **Base class** box in the Generated Classes page of the wizard.

A **CDHtmlDialog**-derived dialog box displays HTML-based dialog boxes and does data exchange with HTML controls and handles HTML events.

Resource language

Sets the language to use for your resources. The list displays the languages available on your system, as installed by Visual Studio. If you want to select a language other than your system language, then the appropriate template folder for that language must already be installed. For more information on installing language resources different from the defaults available in the **Resource language** list, see Wizard Support for Other Languages.

The language you select is reflected in the **Localized strings** option of the Document Template Strings page of the wizard.

See Also

Reference

MFC Application Wizard

Concepts

File Types Created for Visual C++ Projects

Compound Document Support, MFC Application Wizard

In this page of the MFC Application Wizard, indicate to what level your application provides compound and active document support. Your application must support the document/view architecture to support compound documents and document templates.

By default, the application contains no compound document support. If you accept this default, your application cannot support active documents or compound files.

Compound document support

Determines whether your application provides container support, server support, or both. For more information about this area, see:

- Containers: Implementing a Container
- Servers: Implementing a Server

Option	Description
None	Indicates no support for Object Linking and Embedding (OLE). By default, the application wizard creates an application without ActiveX support.
Container	Contains linked and embedded objects.
Mini server	Indicates the application can create and manage compound document objects. Note that mini-servers cannot run stand alone and only support embedded items.
Full server	Indicates the application can create and manage compound document objects. Full servers are able to run stand alone and support both linked and embedded items.
Container/full server	Indicates the application can be both a container and a server. A container is an application that can incorporate embedded or linked items into its own documents. A server is an application that can create Automation items for use by container applications.

Additional options

Indicates whether your application supports active documents. See Active Documents for more information about this feature.

Option	Description
Active document server	Indicates the application can create and manage active documents. If you select this option, you must specify a file extension for your active document server in the **File extension** box in the Document Template Strings page of the wizard. See Active Document Servers for more information.
Active document container	Indicates the application can contain active documents within its frame. Active documents may include, for example, Internet Explorer documents, or Office documents such as Microsoft Word files or Excel spreadsheets. See Active Document Containment for more information.
Support for compound files	Does not serialize the container application's documents using the compound-file format. This option forces the loading of an entire file containing objects into memory. Incremental saves to individual objects are not available. If one object is changed and subsequently saved, then all objects in the file are saved.

See Also

Reference

MFC Application Wizard

Document Template Strings, MFC Application Wizard

In this page of the MFC Application Wizard, provide or refine the following options to help with document management and localization. Document template strings are available for applications that include **Document/view architecture support** in the Application Type. They are not available for dialog boxes. Because most document template strings are visible and used by the application's users, they are localized into the **Resource language** indicated in the **Application Type** page of the wizard.

Nonlocalized strings

Applies to applications that create user documents. Users can open, print, and save documents more easily if you provide a file extension and a file type ID. These items are not localized because they are used by the system rather than by the user.

Option	Description
File extension	Sets the file extension associated with the documents that the user saves when using the application. For example, if your project is named Widget, you could name the file extension .wgt. (When you enter the file extension, do not include the period.) If you provide a file extension, the Explorer can print your application's documents without launching your application when the user drops the document icon on a printer icon. If you do not specify an extension, a user must specify a file extension when saving files. The wizard does not provide a default file extension.
File type ID	Sets the label for your document type in the system registry.

Localized strings

Produces strings associated with the application and document that are read and used by the application's users, so the strings are localized.

Option	Description
Language	Indicates the language in which strings are displayed for all the boxes under **Localized strings**. To change the value in this box, select the appropriate language under **Resource language** in the Application Type page of the MFC Application Wizard.
Main frame caption	Sets the text appearing at the top of the main application frame. By default, the project name.
Doc type name	Identifies the type of document under which a document of the application can be grouped. By default, the project name. Changing the default does not change any other options in this dialog box.
Filter name	Sets the name your users can indicate to find files of your file type. This option is available from the **Files of type** and **Save as type** options in the standard Windows **Open** and **Save as** dialog boxes. By default, the project name plus Files, followed by the extension provided in **File extension**. For example, if your project is named Widget, and the file extension is .wgt, the **Filter name** is Widget Files (*.wgt) by default.
File new short name	Sets the name appearing in the standard Windows **New** dialog box, if there is more than one new document template. If your application is an Automation server, this name is used as the short name of your Automation object. By default, the project name.

File type long name	Sets the file type name in the system registry. If your application is an Automation server, this name is used as the long name of your Automation object. By default, the project name plus .Document.

See Also

Reference

MFC Application Wizard

Database Support, MFC Application Wizard

This page provides options that allow you to specify the level of database support (plus a data source, if necessary) for your project.

Database support

Sets the level of database support for your project.

Option	Description
None	Provides no database support. This is the default option.

Header files only	Provides the basic level of database support for your application.
	• If you select ODBC support under **Client type**, the MFC Application Wizard includes in your project the header file AFXDB.H. It adds link libraries, but it does not create any database-specific classes. You can create recordsets later and use them to examine and update records.
	• If you select OLE DB support under **Client type**, the following header files are included:
	  • ATLBASE.H
	  • AFXOLEDB.H
	  • ATLPLUS.H

Database view without file support	Includes database header files, link libraries, a record view and a recordset. (Available only for applications with the **Document/view architecture support** option selected in the Application Type page.) This option includes document support but no serialization support. If you choose to include a database view, you must specify the source of the data.
Database view with file support	Includes database header files, link libraries, a record view and a recordset. (Available only for applications with the **Document/view architecture support** option selected in the **Application Type** page.) This option supports document serialization, which you can use, for example, to update a user profile file. Database applications typically operate on a per-record basis rather than on a per-file basis and so do not need serialization. However, you may have a special use for serialization. If you choose to include a database view, you must specify the source of the data.

Note

Under **Database Support**, if you select either **Database view without file support** or **Database view with file support**, the view class derivation differs, depending on your **Client type** selection, as follows:

- If you select **ODBC** under **Client type**, then the application's view class derives from CRecordView. This class is associated with a CRecordset-derived class, which the MFC Application Wizard also creates for you. This option gives you a form-based application in which the record view is used to view and update records through its recordset.
- If you select **OLE DB** under **Client type**, then the view class derives from COleDBRecordView, and it is associated with a CTable or CCommand-derived class.

Client type

Indicates whether your project uses OLE DB or ODBC classes.

Option	Description
OLE DB	When this option is selected, clicking the **Data Source** button invokes the **Data Link Properties** wizard to help you create a connection to an OLE DB data source.

ODBC	When this option is selected, clicking the **Data Source** button invokes the **Select Data Source** wizard to help you create a connection to an ODBC data source.

Data Source

Click the **Data Source** button to set up a data source using the specified driver or provider and database. If you selected OLE DB in the **Client type** option, this button displays the **Data Link Properties** dialog box. If you selected ODBC in the **Client type** option, this button provides the **Select Data Source** dialog box. This option is available only if you choose to include a database view in your application.

Option	Description
Data Link Properties (OLE DB)	Establishes the specified data source using the specified OLE DB provider. You must specify the OLE DB provider, the location of the data, the data source, logon ID, and (optionally) a password. For details on this dialog box, see **Data source** in ATL OLE DB Consumer Wizard.
Select Data Source (ODBC)	Establishes the specified data source using the specified ODBC driver. You must select a data source name to choose a table for the data source. The wizard binds all columns of the table to the member variables of a **CRecordset**-derived class. For details on this dialog box, see **Data source** in MFC ODBC Consumer Wizard.

Note
In previous releases, Shift-clicking the **Data Source** button opened a File Open dialog to allow you to select a Data Link (.udl) file. This functionality is no longer supported.

Generate attributed database class

Available for OLE DB client only. Specifies whether the database classes in the generated project use attributes.

Bind all columns

Available for ODBC client only. Specifies whether all columns in the selected table are bound. If you select this box, all columns are bound; if you do not select this box, no columns are bound, and you must bind them manually in the recordset class.

Type

Available for ODBC client only. Specifies whether the recordset is a dynaset or a snapshot, as described in the following table.

Option	Description
Dynaset	Specifies that the recordset is a dynaset. A dynaset is the result of a query that provides an indexed view into the queried database's data. A dynaset caches only an integral index to the original data and thus offers a performance gain over a snapshot. The index points directly to each record found as a result of a query and indicates if a record is removed. You also have access to updated information in the queried records.
Snapshot	Specifies that the recordset is a snapshot. A snapshot is the result of a query and is a view into a database at one point in time. All records found as a result of the query are cached, so you do not see any changes to the original records.

See Also

Reference

MFC Application Wizard

User Interface Features, MFC Application Wizard

This page provides the options by which you indicate the look and feel of your application. The user interface features available for your project depend on the type of application you specified in the Application Type page of the MFC Application Wizard. (For example, if you create a single document interface application, you cannot add child frame styles.)

Main frame styles

Sets the features of your application's main window frame.

Option	Description
Thick frame	Creates a window that has a sizing border. The default.
Minimize box	Includes a minimize box in the main frame window. The default.
Maximize box	Includes a maximize box in the main frame window. The default.
Minimized	Opens the main frame window as an icon.
Maximized	Opens the main frame window to the full size of the display.
System menu	Includes a system menu in the main frame window. The default.
About box	Includes an **About** box for the application, available from the application's **Help** menu. The default, and unchangeable unless you select **Dialog based**, in the Application Type page. **Note** Usually, an unavailable option indicates that the wizard does not apply the option to the project, whether the unavailable item's check box is selected or cleared. In this case, the wizard always adds an **About** box to the project unless you first specify the project as dialog based and then uncheck the box.
Initial status bar	Adds a status bar to your application. The status bar contains automatic indicators for the keyboard's CAPS LOCK, NUM LOCK, and SCROLL LOCK keys and a message line that displays help strings for menu commands and toolbar buttons. Clicking this option also adds menu commands to display or hide the status bar. By default, applications have a status bar. Not available for dialog-based application types.
Split window	Provides a splitter bar. The splitter bar splits the application's main views. In a multiple document interface (MDI) application, the MDI child frame's client window is a splitter window, and in a single document interface (SDI) application and multiple top level document application, the main frame's client window is a splitter window. Not available for dialog-based application types.

Child frame styles

Specifies the appearance and initial state of the child frames in your application. Child frame styles are available for MDI

applications only.

Option	Description
Child minimize box	Specifies that a child window has a minimize button (default).
Child maximize box	Specifies that a child window has a maximize button (default).
Child minimized	Specifies that a child window is initially minimized by setting the cs.style flag **WS_MINIMIZE** in the PreCreateWindow member function of **CChildFrame**. You can set this option or **Child maximized**; you cannot set both at once.
Child maximized	Specifies that a child window is initially maximized by setting the cs.style flag **WS_MAXIMIZE** in the PreCreateWindow member function of **CChildFrame**.

Toolbars

Indicates whether your application includes toolbars. Not available for dialog-based applications.

Option	Description
None	Indicates that your application does not include a toolbar.
Standard docking	Adds a standard Windows toolbar to your application. The toolbar contains buttons for creating a new document; opening and saving document files; cutting copying, pasting, or printing text; displaying the About box; and entering Help mode. Enabling this option also adds menu commands to display or hide the toolbar. By default, applications have a docking toolbar.
Browser style	Adds an Internet Explorer–style toolbar to your application.

Dialog title

For CDialog-based applications only, this title appears in the title bar of the dialog box. To edit this field, the option **Dialog based** under **Application type** must be selected. For more information, see Application Type, MFC Application Wizard.

See Also

Reference

MFC Application Wizard

Advanced Features, MFC Application Wizard

This page provides options for additional features for your application, such as Help, printing support, and so on. In each section, specify additional support for these advanced features.

Context-sensitive help

Generates a set of help files for context-sensitive help, available through F1 and a Help menu, or a Help button on a dialog box. Help support requires the help compiler. If you do not have the help compiler, you can install it by rerunning Setup.

Once you have indicated that you want Help, select the Help format for your application. The format you select determines the format of the starter files the wizard creates for you. See the project's ReadMe.txt in Solution Explorer for a description of the files created to add Help to your project.

Option	Description
WinHelp format	Creates WinHelp-based starter help files. See WinHelp: Context-Sensitive Help for Your Programs and Help Files (WinHelp) for more information.
HTML Help format	Creates HTML-based starter help files. See HTML Help: Context-Sensitive Help for Your Programs and Help Files (HTML Help) for more information.

Printing and print preview

Generates the code to handle the print, print setup, and print preview commands by calling member functions in the CView class from the MFC library. The wizard also adds commands for these functions to the application's menu. Printing support is available only for applications specifying **Document/view architecture support** in the Application Type page of the wizard. By default, document/view applications have printing support.

Automation

Specifies that the application can manipulate objects implemented in another application, or exposes the application to Automation clients.

ActiveX controls

Supports ActiveX controls (default). If you do not select this option and, at a later time, want to insert ActiveX controls into your project, you must add a call to AfxEnableControlContainer in your application's InitInstance member function.

MAPI (Messaging API)

Specifies that the application can create, manipulate, transfer, and store mail messages.

Windows sockets

Supports Windows sockets. Windows sockets allow you to write applications that communicate over TCP/IP networks.

Active Accessibility

Adds support for IAccessible to CWnd-derived classes, which allows you to customize the user interface for better interaction with accessibility clients.

Common Control Manifest

The default. Generates an application manifest to enable the new Common Control DLL shipped with Microsoft Windows XP.

Version 6 of the Common Control DLL does not automatically update the previous version of the Common Controls used by existing applications. To use version 6 of the Common Control DLL, you must create an application manifest that directs your application to load the new DLL. This Common Control DLL also supports the Windows XP themes.

An application manifest can also specify other DLLs and versions that your application needs. Windows XP helps guarantee that your application is loaded only with the specified DLLs. For more information on application manifests, see Isolated Applications and Side-by-Side Assemblies in the *Platform SDK*.

Number of files on recent file list

Specifies the number of files to be listed on the most recently used list. The default number is 4.

See Also

Reference

MFC Application Wizard

Generated Classes, MFC Application Wizard

This page lists the names of base classes and files that your project generates. By default, the names are based on the project name you specified in the New Project dialog box. You can change most of these names, as described below.

Generated classes

Indicates the names of the classes created for the project. By default, the names are based on the project name. The default MFC project creates a C*ProjName*View class, a C*ProjName*App class, a C*ProjName*Doc class, a C*ProjName*MainFrame class, and a C*ProjName*ChildFrame class. All other boxes on this page reflect information about the class currently selected in the **Generated classes** list box.

To change a class name, use the **Class Name** box.

Class name

Indicates the name of the class currently selected in the **Generated classes** list box. If the box is active, you can change the class name. When you change the focus from the **Class Name** box, any change to the selected class name appears in the **Generated classes** list box.

.h file

Indicates the name of the header file of the class currently selected in the **Generated classes** list box. If the text box is active, you can change the name of the header file.

Base class

Indicates the name of the base class for the currently selected class in the **Generated classes** list box. If the box is active, you can select from the list another class for the base class.

.cpp file

Indicates the name of the source code file associated with the selected class. If the text box is active, you can change the name of the implementation file.

See Also

Reference

Advanced Features, MFC Application Wizard

Concepts

File Types Created for Visual C++ Projects

Creating a Forms-Based MFC Application

A form is a dialog box with controls that let a user access and possibly change data. You may want to develop an application in which the user selects from a selection of forms. Commonly, a forms-based application lets the user access forms by click **New** from the **File** menu. A dialog-based application, which does not give users access to a **New** option in the **File** menu, is also considered a forms-based application.

A single document interface (SDI), forms-based application allows only one instance of a particular form to run at a time. It is possible to run different forms at the same time from an SDI forms-based application by selecting a new form from the **New** option in the **File** menu.

If you create a multiple document interface (MDI), forms-based application, the application will be able to support multiple instances of the same form.

If you create an application with multiple top-level document support, the desktop is the implicit parent for the document and the document's frame is not restricted to the client area of the application. You can open multiple instances of the document, each with its own frame, menu, and task bar icon. You can close subsequent instances of documents individually, but if you select the **Exit** option from the **File** menu of the initial instance, the application closes all instances.

SDI, MDI, and multiple top-level document applications are all forms based and use the document/view architecture.

Any dialog-based application, by definition, is forms based. A dialog-based application does not use the document/view architecture, so you must manage the creation and access methods for your own additional forms.

The base class for form-based applications is CFormView. If your application has database support, then you can also select any class that derives from **CFormView**. A form is any window derived from **CFormView** or from any class that inherits from **CFormView**.

Even if you use a base class such as CView, you can later make your applications forms-based by adding an MFC class derived from **CFormView** and checking the **Generate DocTemplate resources** checkbox in the MFC Class Wizard.

Once you finish with the wizard, your project opens, and if you selected **CFormView** (or a class that inherits from **CFormView**) as your base class or if you created a dialog-based application, Visual C++ opens the dialog editor. At this point, you are ready to design your first form.

To begin creating a forms-based MFC executable

1. Follow the directions in Creating an MFC Application.
2. In the MFC Application Wizard Application Type page, select the **Document/view architecture support** check box.
3. Select **Single document**, **Multiple documents**, or **Multiple top-level documents**.

Note
If you chose a SDI, MDI, or multiple top-level document interface application, by default, **CView** is set as the base class for your application's view in the Generated Classes page of the wizard. To create a forms-based application, you must select **CFormView** as the base class for the application's view. Note that the wizard provides no printing support for a forms-based application.

4. Set any other project options you want on the other pages of the wizard.
5. Click **Finish** to generate the skeleton application.

For more information, see:

- Derived View Classes
- Alternatives to the Document/View Architecture
- Application Design Choices

See Also

Tasks

Creating a Windows Explorer-Style MFC Application
Creating a Web Browser-Style MFC Application

Reference

MFC Application Wizard

Concepts

Form Views (MFC)

Creating a Windows Explorer-Style MFC Application

Many Windows system applications use the user interface (UI) for Windows Explorer. When you start Windows Explorer, for example, you see an application with a vertical splitter bar dividing the client area. The left side of the client area provides navigation and browsing features, and the right side of the client area shows details pertinent to the selection in the left pane. When a user clicks an item in the left pane, the application repopulates the right pane. In an MDI application, you can use commands on the **View** menu to change the amount of detail shown in the right pane. (In an SDI or multiple top-level document application, you can change the detail using the toolbar buttons only.)

The contents of the panes depend on the application. In a file-system browser, the left pane shows a hierarchical view of directories or machines, or machine groups, while the right pane displays folders, individual files, or machines, and details about them. The contents do not necessarily have to be files. They could be e-mail messages, error reports, or other items in a database.

The wizard creates the following classes for you:

- The **CLeftView** class defines the left pane of the client area. It is always derived from CTreeView.
- The *CProjName*View class defines the right pane of the client area. By default, it is derived from CListView but can be another type of view depending on the class you specify from the **Base class** list in the Generated Classes page of the wizard.

The generated application can have a single document interface (SDI), a multiple document interface (MDI), or a multiple top-level documents architecture. Each frame window the application creates is vertically split using CSplitterWnd. Coding this application type is similar to coding a normal MFC application that uses a splitter, except that this type of application has separate control views within each splitter pane.

If you use the default list view in the right pane, the wizard creates additional menu choices (in MDI applications only) and toolbar buttons to switch the view's style among large icons, small icons, list, and detail modes.

To begin creating a Windows Explorer-style MFC executable

1. Follow the directions in Creating an MFC Application.
2. In the MFC Application Wizard Application Type page, select the **Windows Explorer** project style.
3. Set any other options you desire on the other pages of the wizard.
4. Click **Finish** to generate the skeleton application.

For more information, see:

- Multiple Document Types, Views, and Frame Windows
- Derived View Classes
- Application Design Choices

See Also

Tasks

Creating a Web Browser-Style MFC Application
Creating a Forms-Based MFC Application

Reference

MFC Application Wizard

Creating a Web Browser-Style MFC Application

A Web browser–style application can access information from the Internet (such as HTML or active documents) or an intranet, as well as folders in the local file system and on a network. By deriving the application's view class from CHtmlView, effectively you make the application a Web browser by providing the view with the WebBrowser control.

To create a Web browser application based on the MFC document/view architecture

1. Follow the directions in Creating an MFC Application.
2. In the MFC Application Wizard Application Type page, make certain that the **Document/view architecture** box is selected. (You can choose either **Single document** or **Multiple documents**, but not **Dialog based**.)
3. On the Review Generated Classes page, use the **Base class** drop-down menu to select **CHtmlView**.
4. Select any other options you want built into the skeleton application.
5. Click **Finish**.

The WebBrowser control supports Web browsing through hyperlinks and Uniform Resource Locator (URL) navigation. The control maintains a history list that allows the user to browse forward and backward through previously browsed sites, folders, and documents. The control directly handles the navigation, hyperlinks, history lists, favorites, and security. Applications can use the WebBrowser control as an active document container to host active documents as well. Thus, richly formatted documents such as Microsoft Excel spreadsheets or Word documents can be opened and edited in place from within the WebBrowser control. The WebBrowser control is also an ActiveX control container that can host any ActiveX control.

Note

The WebBrowser ActiveX control (and therefore **CHtmlView**) is available only to applications running under Windows versio ns in which Internet Explorer 4.0 or later has been installed.

Because **CHtmlView** simply implements the Microsoft Web browser control, its support for printing is not like other CView-derived classes. Rather, the WebBrowser control implements the printer user interface and printing. As a result, **CHtmlView** does not support print preview, and the framework does not provide for other printing support functions: for example, CView::OnPreparePrinting, CView::OnBeginPrinting, and CView::OnEndPrinting, which are available in other MFC applications.

CHtmlView acts as a wrapper for the Web browser control, which gives your application a view onto a Web or an HTML page. The wizard creates an override to the OnInitialUpdate function in the view class, providing a navigational link to the Microsoft Visual C++ Web site:

```
void CWebView::OnInitialUpdate()
{
   CHtmlView::OnInitialUpdate();

   // TODO: This code navigates to a popular spot on the web.
   //  change the code to go where you'd like.
   Navigate2(_T("http://www.msdn.microsoft.com/vstudio/"),NULL,NULL);
}
```

You can replace this site with one of your own, or you can use the LoadFromResource member function to open an HTML page that resides in the project's resource script as the default content for the view. For example:

```
void CWebView::OnInitialUpdate()
{
   CHtmlView::OnInitialUpdate();

   // TODO: This code navigates to a popular spot on the web.
   //  change the code to go where you'd like.
   LoadFromResource(IDR_HTML1);
}
```

See Also

Tasks

MFCIE Sample: Demonstrates the MFC CHtmlView and CReBar Classes

Reference

MFC Application Wizard

Modifying Project Settings

Property Pages (C++)

Modifying Project Settings

Other Resources

Windows Installer Deployment Tasks

Creating an MFC DLL Project

An MFC DLL is a binary file that acts as a shared library of functions that can be used simultaneously by multiple applications. The easiest way to create an MFC DLL project is to use the MFC DLL Wizard.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

To create an MFC DLL Project using the MFC DLL Wizard

1. Follow the instructions in the help topic Creating a Project with a Visual C++ Application Wizard.

Note
In the New Project dialog box, select the **MFC DLL** icon in the Templates pane to open the MFC DLL Wizard.

2. Define your application settings using the application settings page of the MFC DLL Wizard.

Note
Skip this step to keep the wizard default settings.

3. Click **Finish** to close the wizard and open your new project in Solution Explorer

Once your project is created, you can view the files created in the Solution Explorer. For more information about the files the wizard creates for your project, see the project-generated file ReadMe.txt. For more information about the file types, see File Types Created for Visual C++ Projects.

See Also

Reference

Property Pages (C++)

Concepts

Adding Functionality with Code Wizards

Other Resources

Debugging Preparation: Visual C++ Project Types

Windows Installer Deployment Tasks

MFC DLL Wizard

When you use the MFC DLL wizard to create an MFC DLL project, you get a working starter application with built-in functionality that, when compiled, will implement the basic features of a DLL. The MFC starter program includes C++ source (.cpp) files, resource (.rc) files, and a project (.vcproj) file. The code generated in these starter files is based on MFC. For more detailed information, see the file details in Readme.txt that is generated for your project in Visual Studio, and Classes and Functions Generated by the MFC DLL Wizard

Overview

This wizard page describes the current application settings for the MFC DLL project you are creating. By default, the project is created as a regular DLL (MFC Shared) project with no additional settings.

To change these defaults, click **Application Settings** in the left column of the wizard and make changes in that page of the MFC DLL Wizard.

After you create an MFC DLL project, you can add objects or controls to your project using Visual C++ code wizards.

You can perform the following tasks and types of enhancements to a basic MFC DLL project:

- Export from a DLL
- Link an Executable to a DLL
- Initialize a DLL

See Also

Tasks

Adding an MFC Class
Adding a Member Function
Implementing an Interface

Reference

Property Pages (C++)
Modifying Project Settings

Concepts

Wizard Support for Other Languages

Other Resources

Creating and Managing Visual C++ Projects
Windows Installer Deployment Tasks

Application Settings, MFC DLL Wizard

Use this page of the MFC DLL wizard to design and add basic features to a new MFC DLL project.

DLL type

Select the type of DLL you want to create.

Regular DLL using shared MFC DLL

Select this option to link the MFC library to your program as a shared DLL. Using this option, you cannot share MFC objects between your DLL and the calling application. Your program makes calls to the MFC library at run time. This option reduces the disk and memory requirements of your program if it is composed of multiple execution files that use the MFC library. Both Win32 and MFC programs can call functions in your DLL. You must redistribute the MFC DLL with this type of project.

Regular DLL with MFC statically linked

Select this option to link your program statically to the MFC library at build time. Both Win32 and MFC programs can call functions in your DLL. While this option increases the size of your program, you do not need to redistribute the MFC DLL with this type of project. You cannot share MFC objects between your DLL and the calling application.

MFC extension DLL

Select this option if you want your program to make calls to the MFC library at run time, and if you want to share MFC objects between your DLL and the calling application. This option reduces the disk and memory requirements of your program, if it is composed of multiple executable files that all use the MFC library. Only MFC programs can call functions in your DLL. You must redistribute the MFC DLL with this type of project.

Additional features

Select whether your MFC DLL should support automation and whether it should support Windows sockets.

Automation

Select **Automation** to allow your program to manipulate objects implemented in another program. Selecting **Automation** also exposes your program to other Automation clients. See Automation for more information.

Windows sockets

Select this option to indicate that your program supports Windows sockets. Windows sockets allow you to write programs that communicate over TCP/IP networks.

When your MFC DLL with Windows sockets support is created, CWinApp::InitInstance initializes support for sockets and the MFC header file StdAfx.h includes AfxSock.h.

See Also

Tasks

Creating an MFC DLL Project

Concepts

MFC DLL Wizard

Classes and Functions Generated by the MFC DLL Wizard

The code that the MFC DLL Wizard generates depends on the kind of DLL you are creating and the options you have selected. The MFC DLL Wizard generates the same code for both forms of regular DLLs.

Kind of DLL	Option	Classes	Functions
Extension	None	None	**DllMain**
Regular	None	Application class derived from **CWinApp**	None
Regular	Automation	Application class derived from **CWinApp**	**DllGetClassObjectDllCanUnloadNowDllRegisterServer**
Extension	Window Sockets	None	**DllMain**
Regular	Window Sockets	Application class derived from **CWinApp**	**InitInstance** contains call to **AfxSocketInit**

See Also

Concepts

MFC DLL Wizard

Creating a Win32 Windows Application

The easiest way to create a Win32 Windows application is to use the Win32 Application Wizard.

To create a Win32 Windows application

1. Follow the instructions in the help topic Creating a Project with a Visual C++ Application Wizard.
2. In the New Project dialog box, select **Win32 Project** in the Templates pane to open the wizard.
3. Define your application settings using the Win32 Application Wizard.

Note
Skip this step to keep the wizard default settings.

4. Click **Finish** to close the wizard, and your newly created project opens in Solution Explorer.

See Also

Reference

Property Pages (C++)

Concepts

Adding Functionality with Code Wizards

Other Resources

Windows Installer Deployment Tasks

Win32 Application Wizard

The Visual C++ Win32 Application Wizard allows you to create any of four types of projects (listed in the heading in the table below). In each case, you can specify additional options that are appropriate for the type of project you open. The following table indicates which options are available for each application type.

Type of support	Console application	Executable (Windows) application	Dynamic-link library	Static library
Empty project	Yes	Yes	Yes	No
Export symbols	No	No	Yes	No
Precompiled header	No	No	No	Yes
ATL support	Yes	No	No	No
MFC support	Yes	No	No	Yes

Overview

This wizard page describes the current project settings for the Win32 application you are creating. By default, the following options are set:

- The project is a Windows application.
- The project is not empty.
- The project contains no export symbols.
- The project does not use a precompiled header file (this option is available for static library projects only).
- The project includes support for neither MFC nor ATL.

To change these defaults, click the Application Settings tab in the left column of the wizard and make the desired changes.

Once you have created a Win32 application, you can add generic C++ classes using the Generic Code Wizard. You can add other items, such as HTML files, header files, resources, or text files.

Note
You cannot add ATL classes, and you can add MFC classes only to those Win32 application types that support MFC (see the previous table).

You can view the files the wizard creates for your project in Solution Explorer. For more information about the files the wizard creates for your project, see the project-generated file ReadMe.txt. For more information about the file types, File Types Created for Visual C++ Projects.

See Also

Tasks

Creating an Empty Win32 Application

Concepts

Visual C++ Projects

Application Settings, Win 32 Project Wizard

Use this page of the wizard to set options for the Win32 project.

Application type

Creates the specified application type.

Option	Description
Console application	Creates a console application. Console programs are developed with Console Functions, which provide character-mode support in console windows. The Visual C++ run-time libraries also provide output and input from console windows with standard I/O functions, such as **printf()** and **scanf()**. A console application has no graphical user interface. It compiles into an .exe file and can be run as a stand-alone application from the command line. You can add MFC and ATL support to a console application.
Windows application	Creates a Win32 program. A Win32 program is an executable application (EXE) written in C or C++, using calls to the Win32 API to create a graphical user interface. You cannot add MFC or ATL support to a Windows application.
DLL	Creates a Win32 dynamic-link library (DLL). A Win32 DLL is a binary file, written in C or C++, that uses calls to the Win32 API rather than to MFC classes, and that acts as a shared library of functions that can be used simultaneously by multiple applications. You cannot add MFC or ATL support to a DLL application. You can indicate that the DLL exports symbols.
Static library	Creates a static library. A static library is a file containing objects and their functions and data that links into your program when the executable file is built. This topic explains how to create the starter files and project properties for a static library. A static library file provides the following benefits: • A Win32 static library is useful if the application you are working on makes calls to the Win32 API rather than to MFC classes. • The linking process is the same whether the rest of your Windows application is written in C or in C++. • You can link a static library to an MFC-based program or to a non-MFC program.

Additional options

Defines the support and options for the application, depending on its type.

Option	Description
Empty project	Specifies that the project files are blank. If you have a set of source code files (such as .cpp files, header files, icons, toolbars, dialog boxes, and so on) and want to create a project in the Visual C++ development environment, you must first create a blank project, then add the files to the project. This selection is unavailable for static library projects.
Export symbols	Specifies that the DLL project exports symbols.
Precompiled header	Specifies that the static library project uses a pre-compiled header.

Add support for

Add support for one of the libraries supplied in Visual C++.

Option	Description
ATL	Builds into the project support for classes in the Active Template Library (ATL). For Win32 console applications only. **Note** This option does not indicate support for adding ATL objects using the ATL code wizards. You can add ATL objects only to ATL projects or MFC projects with ATL support.
MFC	Builds into the project support for the Microsoft Foundation Class (MFC) Library. For Win32 console applications and static libraries only.

See Also

Tasks

Creating a Win32 Windows Application

Reference

Win32 Application Wizard

Creating an Empty Win32 Application

To create an empty Win32 application

1. Enter a name for your new project, a path to the project directory, and then click **OK**.
2. In the Win32 Application Wizard, click the **Application Settings** page. Select the **Application type** you want to create with your source code file, and then select the **Empty Project** check box under **Additional options**.
3. Click **OK**.

The project appears in Solution Explorer with three directories to contain source files, header files, and resource files.

Next, you can add files to your empty Visual C++ project.

See Also

Concepts

Solutions as Containers

Other Resources

Windows Installer Deployment Tasks

Source Control for Visual Studio

Adding Files to an Empty Win32 Applications

To add your files to an empty Win32 applications

1. Select the directory in Solution Explorer.
2. Right-click the directory name, click **Add** from the shortcut menu, and then click **Existing Item**.
3. In the Add Existing Item dialog box, navigate to the files you want to add to your project.
4. Click **OK**.

To add files that are neither source, header, or resource files to your project, right-click the Solution node in Solution Explorer and add the files to the project in the same manner. A Miscellaneous folder will be created to hold the other files in your project.

Note
Before building your project, you will need to specify build options for these files so that they are included correctly in your finished application. For more information, see Specifying Project Settings with Property Pages and Building a C/C++ Program.

See Also

Tasks

Creating an Empty Win32 Application

Concepts

Solutions as Containers

Other Resources

Windows Installer Deployment Tasks

Source Control for Visual Studio

Creating a Win32 Console Application

The easiest way to create a Win32 console application is to use the Win32 Application Wizard.

To create a Win32 console application

1. Follow the instructions in the help topic Creating a Project with a Visual C++ Application Wizard.
2. In the New Project dialog box, select **Win32 Console Project** in the Templates pane to open the wizard.
3. Define your application settings using the Win32 Application Wizard.

Note
Skip this step to keep the wizard default settings.

4. Click **Finish** to close the wizard, and your newly created project opens in Solution Explorer.

See Also

Reference

Property Pages (C++)

Concepts

Adding Functionality with Code Wizards

Other Resources

Debugging Preparation: Console Projects

Windows Installer Deployment Tasks

How to: Create Windows Control Libraries

To create a Windows control library, use the Windows Control Library template, which is available from the New Project dialog box.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

To create a Windows control library

1. On the **File** menu, click **New**, and then click **Project**.

   The New Project dialog box appears.

2. In the **Project types** pane, click the **Visual C++** node. In the **Templates** pane, click the Windows Control Library icon.

3. In the **Name** box, enter a unique name for your application.

Note
You can also specify other project and solution settings from the New Project dialog box, but these settings are not required.

4. Click **OK**.

See Also

Concepts

Visual C++ Projects

Other Resources

Debugging Preparation: Visual C++ Project Types

Windows Control Library Template (C++)

The Windows Control Library template is used to create custom controls to use on Windows Forms. For more information, see Windows Forms Controls.

The template automatically adds the essential project references and files to use as a starting point for your application:

- References to these .NET Framework namespaces:
  - System.XML - Contains classes that provide standards-based support for processing XML.
  - System.Windows.Forms - Contains classes for creating Windows-based applications that take full advantage of the rich user interface features available in the Microsoft Windows operating system.
  - System.Drawing - Contains classes that provide access to GDI+ basic graphics functionality.
  - System.Data - Contains classes that constitute the ADO.NET architecture, which is the primary data access method for managed applications.
  - System - Contains fundamental classes and base classes that define commonly used values and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
  - mscorlib - The assembly DLL that provides .NET Framework support.
- Source files:
  - Control (.cpp file) - The main source file and entry point into the application that Visual Studio created for you. Identifies the project .dll file and the project namespace. Provide your own code in this file.
  - AssemblyInfo.cpp - The file that contains information (that is, attributes, files, resources, types, versioning information, signing information, and so on) for modifying the project's assembly metadata. For more information, see Assembly Concepts.
  - Stdafx.cpp - Used to build a precompiled header file named Win32.pch and a precompiled types file named StdAfx.obj.
- Header files:
  - Control (.h file) - The main include file for the project, which contains all declarations, global symbols, and **#include** directives for other header files.
  - Control (.resx file) - An XML resource file whose **BuildAction** property is set to **Embedded Resource**.
  - Stdafx.h - Used to build a precompiled header file named Win32.pch and a precompiled types file named StdAfx.obj.
  - resource.h - A generated include file for app.rc.
- Resource files:
  - app.rc - The resource script file of a program.
  - app.ico -The icon file of a program.
- ReadMe.txt - A file describing each file in your project using the actual filenames created by the template.

See Also

Tasks

How to: Create Windows Control Libraries

Reference

Files Created for CLR Projects

Other Resources

Component Classes

Component Model Namespaces in Visual Studio

How to: Create Windows Forms Applications

To create a Windows Forms application, use the Windows Forms Application template, which is available from the New Project dialog box.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

To create a Windows Forms application

1. On the **File** menu, click **New**, and then click **Project**.

   The New Project dialog box appears.

2. In the **Project types** pane, click the **Visual C++** node. In the **Templates** pane, select the Windows Forms Application project template.

3. In the **Name** box, enter a unique name for your application.

Note
You can also specify other project and solution settings from the New Project dialog box, but these settings are not required.

4. Click **OK**.

See Also

Tasks

Debugging Preparation: Windows Forms Applications (.NET)

Concepts

Visual C++ Projects

Windows Forms Application Template

The Windows Forms Application template in Visual C++ lays the groundwork for a standard Windows application. For more information on creating a Windows application, see Creating a Windows Application Project.

The template automatically adds the essential project references and files to use as a starting point for your application:

- References to these .NET Framework namespaces:
  - System.XML - Contains classes that provide standards-based support for processing XML.
  - System.Windows.Forms - Contains classes for creating Windows-based applications that take full advantage of the rich user interface features available in the Microsoft Windows operating system.
  - System.Drawing - Contains classes that provide access to GDI+ basic graphics functionality.
  - System.Data - Contains classes that constitute the ADO.NET architecture, which is the primary data access method for managed applications.
  - System - Contains fundamental classes and base classes that define commonly used values and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
  - mscorlib - The assembly DLL that provides .NET Framework support.
- Source files:
  - Form (.cpp file) - The main source file and entry point into the application that Visual Studio created for you. Identifies the project .dll file and the project namespace. Provide your own code in this file.
  - AssemblyInfo.cpp - The file that contains information (that is, attributes, files, resources, types, versioning information, signing information, and so on) for modifying the project's assembly metadata. For more information, see Assembly Concepts in the .NET Framework SDK.
  - Stdafx.cpp - Used to build a precompiled header file named Win32.pch and a precompiled types file named StdAfx.obj.
- Header files:
  - Form (.h file) - The main include file for the project, which contains all declarations, global symbols, and **#include** directives for other header files.
  - Form (.resx file) - An XML resource file whose **BuildAction** property is set to **Embedded Resource**.
  - Stdafx.h - Used to build a precompiled header file named Win32.pch and a precompiled types file named StdAfx.obj.
  - resource.h - A generated include file for app.rc.
- Resource files:
  - app.rc - The resource script file of a program.
  - app.ico -The icon file of a program.
- ReadMe.txt - A file describing each file in your project using the actual filenames created by the template.

See Also

Tasks

How to: Create Windows Forms Applications

How to: Create a Windows Application Project

Reference

Files Created for CLR Projects

Other Resources

Managing Solutions, Projects, and Files

How to: Create Windows Services (C++ Programming)

To create a Windows service, use the Windows Service template, which is available from the New Project dialog box.

Note
The appearance of features in the IDE can depend on your active settings or edition, and might differ from those described in Help. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see Visual Studio Settings.

To create a Windows Service

1. On the **File** menu, click **New**, and then click **Project**.

   The New Project dialog box appears.

2. In the Project Types pane, click the **Visual C++ Projects** node. In the Templates pane, click the Windows Service icon.

3. In the **Name** box, enter a unique name for your application.

Note
You can also specify other project and solution settings from the **New Project** dialog box, but these settings are not required.

4. Click **OK**.

See Also

Other Resources

Windows Service Applications

Windows Service Template (C++)

The Windows Service template adds the necessary items needed to create a Windows Service application, a long-running executable application that runs in its own Windows session. For more information, see Introduction to Windows Service Applications.

The template automatically adds the essential project references and files to use as a starting point for your application:

- References to these .NET Framework namespaces:
  - System.XML - Contains classes that provide standards-based support for processing XML.
  - System.Configuration.Install - Contains classes that allow you to write custom installers for your own components.
  - System.ServiceProcess - Contains classes that allow you to install and run services, which are long-running executables with no user interface.
  - System.Data - Contains classes that constitute the ADO.NET architecture, which is the primary data access method for managed applications.
  - System - Contains fundamental classes and base classes that define commonly used values and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
  - mscorlib - The assembly DLL that provides .NET Framework support.
- Source files:
  - Service (.cpp file) - The main source file and entry point into the application that Visual Studio created for you. Identifies the project .dll file and the project namespace. Provide your own code in this file.
  - AssemblyInfo.cpp - The file that contains information (that is, attributes, files, resources, types, versioning information, signing information, and so on) for modifying the project's assembly metadata. For more information, see Assembly Concepts.
  - Stdafx.cpp - Used to build a precompiled header file named Win32.pch and a precompiled types file named StdAfx.obj.
- Header files:
  - Service (.h file) - The main include file for the project, which contains all declarations, global symbols, and **#include** directives for other header files.
  - Service (.resx file) - An XML resource file whose **BuildAction** property is set to **Embedded Resource**.
  - Stdafx.h - Used to build a precompiled header file named Win32.pch and a precompiled types file named StdAfx.obj.
  - resource.h - A generated include file for app.rc.
- Resource files:
  - app.rc - The resource script file of a program.
  - app.ico -The icon file of a program.
- ReadMe.txt - A file describing each file in your project using the actual filenames created by the template.

See Also

Tasks

How to: Create Windows Services (C++ Programming)

Reference

Files Created for CLR Projects

Concepts

Introduction to Monitoring Performance Thresholds

Other Resources

Managing Solutions, Projects, and Files

How to: Create a SQL Server Project (C++)

To create a SQL Server class library application, use the **SQL Server Project** template, which is available from the New Project Dialog Box.

To create a SQL Server Class Library

1. On the **File** menu, point to **New**, and then click **Project**.

   The **New Project** dialog box appears.

2. In the **Project Types** pane, select the **Visual C++** node. In the **Templates** pane, select the **SQL Server Project** template.

3. In the **Name** box, enter a unique name for your application.

4. Click **OK**.

   A new solution, which contains the new SQL Server class library application, will open in the Visual Studio Integrated Development Environment (IDE), which automatically prompts you to add a database reference. For more information, see Add Database Reference Dialog Box.

See Also

Reference

SQL Server Project Template

Concepts

Visual C++ Projects

SQL Server Project Template

Use the SQL Server Class Library template to create SQL Server class library applications.

The template automatically adds the essential project references and files to use as a starting point for your application:

- References to these Microsoft .NET Framework namespaces:
  - System — Contains fundamental classes and base classes that define commonly used values and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
  - System.Data — Contains classes that constitute the ADO.NET architecture, which is the primary data access method for managed applications.
  - System.Data.Sql — Contains classes for managing available instances of SQL Server within the local network.
  - System.Data.SqlTypes — Contains classes for native data types within SQL Server 2005 that provide a safer, faster alternative to the data types provided by the .NET Framework common language runtime.
  - Microsoft.SqlServer.Server — Contains classes that simulate communication channels between the client and server.
- Source files:
  - *<SQLServerProjectName>*.cpp — The default source file. It contains the declaration for a function named `MyStoredProc`, which is declared with the [Microsoft.SqlServer.Server.SqlProcedure] attribute to denote that it is a common language runtime enabled SQL stored procedure. Provide your own code in this file.
  - AssemblyInfo.cpp — The file that contains information about the assembly metadata of your project. It includes information about attributes, files, resources, types, versioning information, and signing. For more information, see Assembly Contents.
  - stdafx.cpp — A file that is used to build a precompiled header file named Win32.pch, and a file named stdafx.obj that contains precompiled types.
- Header files:
  - *<SQL Server Project Name>*.h — The main include file for the project, which contains all declarations, global symbols, and **#include** directives for other header files.
  - stdafx.h — A file that is used to build a precompiled header file named Win32.pch, and a file named stdafx.obj that contains precompiled types.
  - resource.h - A generated include file for app.rc.
- Resource files:
  - app.rc — The resource script file of a program.
  - app.ico —The icon file of a program.
- ReadMe.txt — A file that describes each file in your project using the file names that are created by the template.

See Also

Tasks

How to: Create a SQL Server Project (C++)

How to: Deploy a C++ SQL Server Project to a Server

How to: Add a SQL Server 2005 Reference

Use the **Connect to Database** command to add a SQL Server 2005 database reference to your Visual C++ project. This command opens either the **Choose Data Source** dialog box or the **Add Connection** dialog box.

To add a database reference to your C++ project

1. Open your SQL Server Project.
2. On the **Tools** menu, click **Connect to Database**.

   If this is the first connection you make, the **Choose Data Source** dialog box appears.

   a. In this dialog box, under **Data source**, select **Microsoft SQL Server**.

   b. Under **Data provider**, choose the provider appropriate to the application you are working with.

   c. Click **Continue**.
3. In the **Add Connection** dialog box, enter the requested information.

   This information is different for each provider. For more help on this dialog box, click the **Help** icon at the top of the dialog box or press the F1 key while the dialog box is selected.
4. Click **Advanced** to display a list of properties that enable you to change settings for the selected provider.
5. Click **Test Connection** to check the connection without closing the dialog box. This way you can make adjustments to the settings if the connection does not succeed.
6. Click **OK**.

   Your connection appears in **Server Explorer** under the **Data Connections** node.

See Also

Tasks

How to: Connect to a Database from Server Explorer

Reference

SQL Server Project Template
Add Database Reference Dialog Box
Choose/Change Data Source Dialog Box
Add/Modify Connection (Microsoft SQL Server)

How to: Deploy a C++ SQL Server Project to a Server

To deploy a C++ SQL Server Project (or stored procedure), use the **Deploy** command. This command is available on the **Build** menu and on the shortcut menu of the solution in **Solution Explorer**.

To deploy a C++ class library to a server

1. Open your C++ SQL Server Project.
2. On the **Build** menu, click **Deploy** <*Solutionname*>.

   — or —

   In **Solution Explorer**, right-click your solution and then click **Deploy** on the shortcut menu.

The **Output** window contains information about whether the deployment of your stored procedure succeeded or failed.

Project Settings for Deployment

You might want to modify the permission level of the stored procedure that you want to deploy. By default, Visual C++ SQL Server project property settings are set up to deploy stored procedures that have the **SAFE** permissions level. The following procedures describe how to modify your project property settings so that you can deploy stored procedures that have different permission levels.

To set up safe SQL server projects for deployment

1. Open your C++ SQL Server Project.
2. Open the property pages dialog box for your project.
3. Under the **Configuration Properties** node, select the **General** page.
4. Set the **Common Language Runtime support** property to **Safe MSIL Common Language Runtime Support (/clr:safe)**.
5. Set the **Assembly Permission Level** property to **Safe**.

See Also

Tasks

How to: Open Project Property Pages
How to: Create a SQL Server Project (C++)
How to: Add a SQL Server 2005 Reference

Reference

SQL Server Project Template

How to: Create a C++ Project from Existing Code

You can port your existing code files into a Visual C++ project using the Create New Project From Existing Code Files Wizard.

Porting your existing code files into a Visual C++ project enables you to use IDE features such as IntelliSense, Call Browser, and Property Manager.

To create a C++ project from existing code

1. On the **File** menu, point to **New**, and then click **Project From Existing Code**.
2. On the first page of the **Create New Project from Existing Code Files** wizard, select **Visual C++** in the **What type of project would you like to create** list.
3. In subsequent pages of the wizard, specify settings for the new project that you want to create.
4. Click **Finish** to generate the new project.

See Also

Reference

Specify Project Location and Source Files, Create New Project From Existing Code Files Wizard
Specify Project Settings, Create New Project From Existing Code Files Wizard
Specify Debug Configuration Settings, Create New Project From Existing Code Files Wizard
Specify Release Configuration Settings, Create New Project From Existing Code Files Wizard

Specify Project Location and Source Files, Create New Project From Existing Code Files Wizard

Use this page of the Create New Project From Existing Code Files wizard to specify:

- The directory path of the new project
- The directories to search for existing source files
- The kinds of files the wizard will import into the new project

How to: Create a C++ Project from Existing Code

Project file location

Specifies the directory path of the new project. This location is where the wizard will deposit all the files (and subdirectories) of the new project.

Browse

Displays the **Project file location** dialog, which helps you specify the directory that will contain the new project. This control enables you to navigate to the desired folder.

Project name

Specifies the name of the new project. Project files, which have file extentions such as .vcproj, .ncb, etc. will adopt this name. Existing code files will keep their original name.

Add files to the project from these folders

When checked, sets the wizard to copy existing code files from their original directories (that are specified in the list box below this control) into the new project.

Add Subfolders

Specifies to copy code files from all subdirectories of the directory listed **Folder** column into the new project.

Folder

Indicates the path to the directory that contains existing code files to copy into the new project. This column lists all the directories that the wizard will search for existing code files.

Add

Displays the **Add files to the project from this folder** dialog box, which helps you the specify directories that the wizard will search for existing code files.

Remove

Deletes the directory path that is selected in the list box left of this control.

File types to add to the project

Specifies the kinds of files that the wizard will add to the new project based on the given file extensions. File extensions are preceded with the asterisk wildcard character, and are delimited in the list of file extensions by a semicolon.

Show all files in Solution Explorer

Specifies that all files in the new project to be visible and displayed in the Solution Explorer window. This option is enabled by default.

Specify Project Settings, Create New Project From Existing Code Files Wizard

Use this page of the Create New Project From Existing Code Files wizard to specify:

- The build environment for the new project
- Build settings to match a specific type of new project to generate

How to: Create a C++ Project from Existing Code

Use Visual Studio

Specifies to use build tools that are included in Visual Studio for building the new project. This option is selected by default.

Project Type

Specifies the type of project that the wizard will generate.

Windows application project

Indicates that the wizard will generate a project for an executable Windows application. This option is available from the **Project Type** drop-down list box.

Console application project

Indicates that the wizard will generate a project for a console application. This option is available from the **Project Type** drop-down list box.

Dynamically linked library (DLL) project

Indicates that the wizard will generate a project for an empty dynamic link library application. This option is available from the **Project Type** drop-down list box.

Static library (LIB) project

Indicates that the wizard will generate a project for a static library application. This option is available from the **Project Type** drop-down list box.

Add support for ATL

Adds ATL support to the new project.

Add support for MFC

Adds MFC support to the new project.

Add support for the Common Language Runtime

Adds CLR programming support to the new project.

Common Language Runtime

Specifies the new project to be compliant with CLR features.

Common Language Runtime (old syntax)

Specifies the new project to be compliant with Managed Extensions for C++ syntax, which is CLR programming syntax prior to Visual C++ 2005.

Use external build system

Specifies to use build tools that are not included in Visual Studio for building the new project. When this option is selected, you can specify build command lines on the **Specify Debug Configuration Settings** and **Specify Release Configuration Settings** pages.

Note

When the **Use External Build System** option is checked, the IDE does not build the new project, so the /D, /I, /FI, /AI, or /FU options are not required for compilation. However, these options must be set correctly in order for IntelliSense to function properly.

See Also

Reference

Specify Debug Configuration Settings, Create New Project From Existing Code Files Wizard

Specify Release Configuration Settings, Create New Project From Existing Code Files Wizard

Specify Debug Configuration Settings, Create New Project From Existing Code Files Wizard

Use this page of the Create New Project From Existing Code Files wizard to specify Debug configuration project settings.

How to: Create a C++ Project from Existing Code

Build command line

Specifies the command line that builds the new project. For example, enter the name of the compiler (plus any switches and arguments) or the build scripts that you want to use to build the new project. This option is enabled when the **Use external build system** option is selected in the **Specify Project Settings** page; otherwise it is unavailable.

Rebuild command line

Specifies the command line that rebuilds the new project. This option is enabled when the **Use external build system** option is selected in the **Specify Project Settings** page; otherwise it is unavailable.

Clean command line

Specifies the command line to delete support files generated by the build tools for the new project. This option is enabled when the **Use external build system** option is selected in the **Specify Project Settings** page; otherwise it is unavailable.

Output (for debugging)

Specifies the directory path of the output files for the Debug configuration of the new project. This option is enabled when the **Use external build system** option is selected in the **Specify Project Settings** page; otherwise it is unavailable.

Preprocessor definitions (/D)

Defines preprocessor symbols for the new project. For more information, see /D (Preprocessor Definitions).

Include search path (/I)

Specifies directory paths to add to the list of directories that the compiler will search to resolve file references passed to preprocessor directives in the new project. For more information, see /I (Additional Include Directories).

Forced included files (/FI)

Specifies header files to process when building the new project. For more information, see /FI (Name Forced Include File).

.NET assembly search path (/AI)

Specifies the directory paths that the compiler will search to resolve .NET assembly references passed to preprocessor directives in the new project. For more information, see /AI (Specify Metadata Directories).

Forced using .NET assemblies (/FU)

Specifies .NET assemblies to process when building the new project. For more information, see /FU (Name Forced #using File).

See Also

Reference

Specify Project Settings, Create New Project From Existing Code Files Wizard

Specify Release Configuration Settings, Create New Project From Existing Code Files Wizard

Use this page of the Create New Project From Existing Code Files wizard to specify Release configuration project settings.

How to: Create a C++ Project from Existing Code

Same as Debug configuration

Specifies that the wizard will generate Release configuration project settings identical to Debug configuration project settings. This option is checked by default. All other options on this page are inactive unless you uncheck this box.

Build command line

Specifies the command line that builds the new project. Enter the name of the compiler plus any switches or arguments that you want to use to build the new project. This option is enabled when the **Use external build system** option is selected in the **Specify Project Settings** page.

Rebuild command line

Specifies the command line that rebuilds the new project. This option is enabled when the **Use external build system** option is selected in the **Specify Project Settings** page.

Clean command line

Specifies the command line to delete support files generated by the build tools for the new project. This option is enabled when the **Use external build system** option is selected in the **Specify Project Settings** page.

Output (for debugging)

Specifies the directory path of the output files for the Debug configuration of the new project. This option is enabled when the **Use external build system** option is selected in the **Specify Project Settings** page.

Preprocessor definitions (/D)

Defines preprocessor symbols for the new project. For more information, see /D (Preprocessor Definitions).

Include search path (/I)

Specifies directory paths to add to the list of directories that the compiler will search to resolve file references passed to preprocessor directives in the new project. For more information, see /I (Additional Include Directories).

Forced included files (/FI)

Specifies header files to process when building the new project. For more information, see /FI (Name Forced Include File).

.NET assembly search path (/AI)

Specifies the directory paths that the compiler will search to resolve .NET assembly references passed to preprocessor directives in the new project. For more information, see /AI (Specify Metadata Directories).

Forced using .NET assemblies (/FU)

Specifies .NET assemblies to process when building the new project. For more information, see /FU (Name Forced #using File).

See Also

Reference

Specify Project Settings, Create New Project From Existing Code Files Wizard

How to: Organize Project Output Files for Builds

This topic describes best practices for organizing project output files. Build errors can occur when you set up project output files incorrectly. This topic also outlines the advantages and disadvantages of each alternative for organizing your project output files.

Referencing CLR Assemblies

To reference assemblies with #using

- You can reference an assembly directly from your code using the #using directive, such as `#using <System.Data.dll>`. For more information, see The #using Directive.

  The file specified can be a .dll, .exe, .netmodule, or .obj, as long as it is in MSIL. The referenced component can be built in any language. Using this option, you will have access to Intellisense since the metadata will be extracted from the MSIL. The file in question must be in the path for the project; otherwise, the project will not compile and Intellisense will not be available. An easy way to determine whether the file is in the path is to right-click on the #using line and choose the **Open document** command. You will be notified if the file cannot be found.

  If you do not want to put the full path to the file, you can use the **/AI** compiler option to edit the search path for #using references. For more information, see /AI (Specify Metadata Directories).

To reference assemblies with /FU

- Instead of referencing an assembly directly from a code file as described above, you can use the **/FU** compiler option. The advantage to this method is that you do not need to add a separate #using statement to every file that references a given assembly.

  To set this option, open the **Properties Pages** for the project. Expand the **Configuration Properties** node, and then expand the **C/C++** node and select **Advanced**. Add the desired assemblies next to **Force #using**. For more information, see /FU (Name Forced #using File).

To reference assemblies with Add New Reference

1. This is the easiest way to make use of CLR assemblies. First, make sure the project is compiled with the **/clr** compiler option. Then, right click the project from the **Solution Explorer** and select **References**. The **Property Pages** dialog will appear.

2. From the **Property Pages** dialog, select **Add New Reference**. A dialog will appear listing all .NET, COM, and other assemblies available in the current project. Select the desired assembly and click **OK**. For more information, see Add Reference Dialog Box.

   Once a project reference is set, the corresponding dependencies are automatically handled. In addition, since metadata is part of an assembly, there is no need to add a header file or prototype the elements that are being used from managed assemblies.

Referencing Native DLLs or Static Libraries

To reference native DLLs or static libraries

1. Reference the appropriate header file in your code using the #include directive. The header file must be in the include path or part of the current project. For more information, see The #include Directive.

2. You can also set project dependencies. Setting project dependencies guarantees two things. First, it ensures that projects are built in the right order so that a project can always find the dependent files it needs. Second, it implicitly adds the dependent project's output directory to the path so that files can be found easily at link-time.

3. To deploy the application, you will need to place the DLL in an appropriate place. This can be one of the following:

   a. The same path as the executable.

   b. Anywhere in the system path (the **path** environment variable).

c. In the side-by-side assembly. For more information, see Building C/C++ Side-by-side Assemblies.

Working with Multiple Projects

By default, projects are built such that all output files are created in a subdirectory of the project directory. The directory is named based on the build configuration (e.g. Debug or Release). In order for sibling projects to refer to each other, each project must explicitly add the other project output directories to their path in order for linking to succeed. This is done automatically when you set the project dependencies. However, if you do not use dependencies, you must carefully handle this because builds can become very difficult to manage. For example, when a project has Debug and Release configurations, and it includes an external library from a sibling project, it should use a different library file depending on which configuration is being built. Thus, hard-coding these paths can be tricky.

Starting with Visual C++ 2005, all essential output files (such as executables, incremental linker files, and PDB files) are copied into a common solution directory. Thus, when working with a solution that contains a number of C++ projects with equivalent configurations, all the output files are centralized for simplified linking and deployment. You can be sure that their application/library will work as expected if they keep those files together (since the files are guaranteed to be in the path).

The location of output files can be a major issue when deploying to a production environment. While running projects in the IDE, the paths to included libraries are not necessarily the same as in the production environment. For example, if you have `#using "../../lib/debug/mylib.dll"` in your code but then deploy mylib.dll into a different relative position, the application will fail at runtime. To prevent this, you should avoid using relative paths in #include statements in your code. It is better to ensure that the necessary files are in the project build path and similarly ensuring that the corresponding production files are properly placed.

How to specify where output files go

- The location of project output settings can be found in the project's **Property Pages**. Expand the node next to **Configuration Properties** and select **General**. The output location is specified next to **Output Directory**. For more information, see General Property Page (Project).

See Also

Concepts

Visual C++ Projects

Modifying Project Settings

A project configuration is a collection of properties that determine the behavior of the build process and the debugger. This collection of properties (or project settings) is stored in one or more of the following files:

- Project Files
- Property Sheets
- Tool Files

The Visual C++ project system provides the following mechanisms to help you define a project configuration or override the default project configuration:

- Property Pages
- Project Property Sheet Wizard

See Also

Other Resources

Creating and Managing Visual C++ Projects

Property Pages (C++)

A project property page is a Visual Studio user interface element that allows you to specify and apply project settings to your projects.

You access property pages by selecting a project and clicking Properties on the shortcut menu or clicking Properties on the Project menu. When you specify values in the Property Pages dialog box, they are applied to the selected project or projects in Solution Explorer.

You can specify and apply project settings to all build configurations, or you can specify separate project properties for each build configuration. For example, you can specify settings for the Release configuration that do not apply to the Debug configuration.

Property pages are dynamic, and the pages that are displayed depend on the file types in the project. For example, if you do not have an .idl file in your project, then the IDL property page will not be displayed in the Property Pages dialog box.

For more information, see How to: Specify Project Properties with Property Pages.

Default vs. Modified Properties

Some property values appear in bold, which means that the value was changed. The default setting for a property at the project level when the project node is selected in Solution Explorer is commonly the default value of that property for the tool. For example, in a Win32 project on the compiler's **Code Generation** property page, the property **Enable String Pooling** is initially set to No (not bold). This means that No agrees with the project's default setting. If you were to use CL.exe at the command line, /GF would not be in effect unless you specified it.

If you or the application wizard change a property, the value appears in bold. For example, on the same **Code Generation** property page, **/RTC1** is in effect for the **Full Runtime Checks** property. At the command line, **/RTC1** is not in effect by default; in this case, the application wizard changed the project's default setting for the property.

A subset of the project defaults is exposed and available for you to modify. For more information, see the Project Defaults section in General Property Page (Project).

Resetting Properties

When you view a project's Property Pages dialog box and when the project node is selected in Solution Explorer, for each property you will also have the option to select **<inherit from project defaults>**.

Click **Apply** to ensure that the **Property Pages** dialog box is refreshed with your latest choices when you reset a property to the project's default.

When you view a project's **Property Pages** dialog box and when a file is selected in Solution Explorer, a property value that appears in bold indicates that the value of the property at the file level has been changed. Accordingly, for each property you will also have the option to select **<inherit from project>**. If many files have properties that were modified, the time to build the project will increase.

The project defaults are system (platform) defaults, for the most part. Some project defaults derive from the style sheets that are applied when you update properties in the Project Defaults section of the general project configuration page (see General Property Page (Project)).

Entering User-Defined Values

Other properties require you to specify values for the property. Some of these properties take one or more values (multi-properties) and other properties take one value (single properties).

It is possible to type directly in the right column of the property page. Separate values with a semicolon. Use only alphabetic and numeric characters for these properties. No error checking is performed on these values.

You also can select **<Edit...>** from the drop-down list (for single properties) or the browse button (for a selected multi-property). This opens a dialog box where you can enter a value or values for the property. This dialog box's title will be the same as the name of the property you are editing. For multi-properties, enter one value per line, using a carriage return after each value.

Edit boxes for multi-properties will have a check box labeled **inherit from project** (for a file) or **inherit from project defaults** (at the project level). This check box is selected by default, meaning that the property can inherit values from the project, if a file is selected, or from the project defaults, if the project is selected in **Solution Explorer**.

Note that the displayed values for a multi-property in the property page dialog box only reflect the property settings at the current level. For example, if a file is selected in the **Solution Explorer** and you select the C/C++ **Preprocessor Definitions** property, the property page will not display any preprocessor definitions at the project level. However, if you open the edit box for the multi-property, you can see all inherited values for the property.

If you use the Visual C++ Project Model, be aware that this behavior is also in effect for the objects on files and projects. That is, when querying for the values on a property at the file level, you will not get the values for that same property at the project level; you'll need to explicitly get the values of the property at the project level. Be aware that some inherited values of a property may come from a style sheet, which is not accessible programmatically for this release.

$(Inherit) and $(NoInherit) Macros

Note that two macros can be used for properties where you specify user-defined values:

- **$(Inherit)**
- **$(NoInherit)**

Clearing the check box (for example, **inherit from project defaults**) is the same as specifying **$(NoInherit)**; using the check box is easier. If you specify **$(NoInherit)**, you may very well want to still specify your own values.

Regardless of the setting for the **inherit from project** (or **inherit from project defaults**) check box, if you use either of these macros in the edit box, the setting of the check box is ignored.

If you do use **$(NoInherit)** on a property, be aware if one of the project defaults affects the property. If it does, you can unintentionally override a project default that you specified.

See Also

Tasks

How to: Create and Remove Project Dependencies
How to: Create and Edit Configurations

Other Resources

Windows Installer Deployment Tasks
Source Control for Visual Studio

References, Common Properties, <Projectname> Property Pages Dialog Box

Use this property page dialog box to set up C++ projects with references to .NET Framework assemblies, COM components, or external projects.

Note
This property page dialog box is only available for CLR projects.

To open this property page dialog box, select **References** on the **Project** menu or on the project context menu in Solution Explorer. Alternatively, you can open the property pages for your project, expand the **Common Properties** node, and then select **References**.

How to: Open Project Property Pages

References

The **References** pane displays a list of all the components that your project currently references, such as .NET Framework assemblies, COM components, and external projects.

Add New Reference

Opens the **Add Reference** dialog box. For more information, see Add Reference Dialog Box.

Remove Reference

Deletes the reference to the assembly that is selected in the **References** list.

ActiveX Reference Properties

Note
ActiveX reference properties are only available for references to COM components. These properties are only displayed when a COM component is selected in the **References** pane. These properties are not modifiable.

Control Full Path

The full path to the referenced control.

Control GUID

The globally unique identifier for the ActiveX control.

Control Version

The version of the referenced ActiveX control.

Type Library Name

The name of the referenced type library.

Wrapper Tool

The tool used to build the interop assembly from the referenced COM library or ActiveX control.

Assembly Reference Properties

Note
Assembly reference properties are only available for references to .NET Framework assemblies. These properties are only displayed when a .NET Framework assembly is selected in the **References** pane. These properties are not modifiable.

Relative Path

The relative path from the project directory to the referenced assembly.

Build Properties

The following properties enable you to specify how to build with references.

Copy Local

Specifies whether to automatically copy the referenced assembly to the target location during a build.

Copy Local Dependencies

Specifies whether to automatically copy the referenced assembly's non-system dependencies to the target location during a build. Only used if **Copy Local** is **true**.

Copy Local Satellite Assemblies

Specifies whether to automatically copy the referenced assembly's satellite assemblies to the target location during a build. Only used if **Copy Local** is **true**.

Use Dependencies In Build

Specifies that the dependencies of this assembly are used in the build process. If **true**, this assembly's dependencies will be used on the compiler's command line during the build.

Use In Build

Specifies that this assembly is used in the build process. If **true**, the assembly will be used on the compiler's command line during the build.

Reference Properties

The following properties are not modifiable.

Assembly Name

Indicates the assembly name for the referenced assembly.

Culture

Indicates the culture of the selected reference.

Description

Displays the description of the selected reference.

Full Path

Indicates the directory path to the referenced assembly.

Identity

Indicates the full path for .NET Framework assemblies. Indicates the GUID for COM components.

Label

Indicates the label of the reference.

Name

Indicates the name of the reference.

Public Key Token

Indicates the public key token used to identify the referenced assembly.

Strong Name

true if the referenced assembly has a strong name. A strong named assembly is uniquely versioned.

Version

Indicates the version of the referenced assembly.

Additional reference search paths

The **Additional reference search paths** pane displays a list of additional paths to search when resolving assembly references.

Add Path

Opens the **Add Reference Search Path** dialog box, which makes it possible for you to add search paths. For more information, see Add Reference Search Path Dialog Box.

Remove Path

Removes the selected search path from the list of reference search paths.

Move Up

Moves the selected search path up in the list.

Move Down

Moves the selected search path down in the list.

See Also

Reference

Property Pages (C++)

Add Reference Search Path Dialog Box

Use this dialog box to specify an additional directory path that the project system will search to resolve references to DLLs, assemblies, or modules in your application.

To access this dialog box, click **Add Path** on the References, Common Properties, <Projectname> Property Pages Dialog Box.

Search path

Specifies the path to the directory that the project system will search to resolve references to DLLs, assemblies, or modules in your application.

Browse

Opens the **Add Reference Search Path** dialog box, which helps you locate a file directory and specify its directory path.

Make this path relative to the project directory (for paths with no macros)

When selected, specifies that the reference search path is determined from the project directory, rather than from the root directory.

See Also

Reference

Project Files

VC++ Directories, Projects and Solutions, Options Dialog Box

General Property Page (Project)

When a project is selected in Solution Explorer, the **General** property page under the **Configuration Properties** node, displays two sections of properties:

- General
- Project Defaults

General

The properties in the General section affect the location of files that are created in the build process and which files to delete when the **Clean** option (**Build** menu) is selected.

Output Directory

Specifies the directory where tools such as the linker will place all final output files that are created during the build process. Typically, this includes the output of tools such as the linker, librarian, or BSCMake.

To programmatically access this property, see OutputDirectory.

Intermediate Directory

Specifies the directory where tools such as the compiler will place all intermediate files created during the build process. Typically, this includes the output of tools such as the C/C++ compiler, MIDL, and the resource compiler.

To programmatically access this property, see IntermediateDirectory.

Extensions to Delete on Clean

The **Clean** option (**Build** menu) deletes files from the intermediate directory where a project's configuration is built. Files with extensions specified with this property will be deleted when **Clean** is run or when you perform a rebuild. In addition to files of these extensions in the intermediate directory, the build system will also delete any known output of the build regardless of where it is located (including intermediate outputs such as .obj files). Note that you can specify wildcard characters.

To programmatically access this property, see DeleteExtensionsOnClean.

Build Log Files

Allows you to specify a non-default location for the log file that is created whenever you build a project.

You can use project macros to change the directory location. See Macros for Build Commands and Properties.

Inherited Project Property Sheets

Specifies which property sheets you want to use. See How to: Inherit Project Settings from Property Sheets for more information.

Project Defaults

The properties in the Project Default section represent default properties that you can modify. See Specifying Project Settings with Property Pages for a discussion of default properties. The definition for these properties can be found in the .vsprops files in Program Files\Microsoft Visual Studio 8\VC\VCProjectDefaults.

Configuration Type

There are several configuration types from which to choose:

- Application (.exe), displays linker toolset (C/C++ Compiler, MIDL, Resource Compiler, Linker, BSCMake, XML Web Service Proxy Generator, custom build, prebuild, prelink, postbuild events).
- Dynamic Library (.dll), displays linker toolset, specifies /DLL linker option, and adds the _WINDLL define to CL.
- Makefile, displays makefile toolset (NMake).
- Static Library (.lib), displays librarian toolset (same as linker toolset except substitute librarian for linker and omit XML Web Service Proxy Generator).
- Utility, displays utility toolset (MIDL, custom build, prebuild, postbuild events).

To programmatically access this property, see ConfigurationType.

Use of MFC

Specifies whether the MFC project will statically or dynamically link to the MFC DLL. Non-MFC projects can select **Use Standard Windows Libraries** to link to various Win32 libraries that are included when you use MFC.

To programmatically access this property, see useOfMfc.

Use of ATL

Specifies whether the ATL project will statically or dynamically link to the ATL .DLL. If you specify anything other than **Not Using ATL**, a define will be added to the compiler's **Command Line** property page.

To programmatically access this property, see useOfATL.

Minimize CRT Use in ATL

This property prevents CRT startup code from being linked with an ATL project. When enabled, it also defines _ATL_MIN_CRT. This will reduce the size of your binary but prevents calling any functions that require the CRT startup code. The **Use of ATL** property must be set to something other than **Not Using ATL** in order for **Minimize CRT Use in ATL** to be in effect.

For more information, see Knowledge Base article Q165076 : INFO LNK2001 on CRT Symbols in ATL Release Build for more info.

To programmatically access this property, see ATLMinimizesCRunTimeLibraryUsage.

Character Set

Defines whether _UNICODE or _MBCS should be set. Also affects the linker entry point where appropriate.

To programmatically access this property, see CharacterSet.

Common Language Runtime support

Causes the /clr compiler option to be used.

To programmatically access this property, see ManagedExtensions.

Whole Program Optimization

Specifies the /GL compiler option and /LTCG linker option.

For information on how to access the **General** property page under the **Configuration Properties** node, see How To: Specify Project Properties with Property Pages.

See Also

Reference

Property Pages (C++)

General Property Page (File)

When a file is selected in Solution Explorer, the **General** property page under the **Configuration Properties** node contains the following properties:

Exclude From Build

Specifies whether the file should be in the build for the current configuration.

To programmatically access this property, see ExcludedFromBuild.

Tool

The tool that will be used to build this file. See Specifying Custom Build Steps for more information.

To programmatically access this property, see Tool.

For information on how to access the **General** property page under the **Configuration Properties** node, see How To: Specify Project Properties with Property Pages.

See Also

Reference

Property Pages (C++)

Command Line Property Pages

Most property page folders contain a **Command Line** property page. This page displays which properties are set in the folder. The **Command Line** property page also contains an **Additional Options** box where you can specify properties that are valid for the tool but for which there is no property in the folder.

Any command that you enter in the edit box will be passed through to the tool for the folder. No verification or checks will be done on the input, nor will there be any dependency checking.

For information on how to access the **Command Line** property pages, see How To: Specify Project Properties with Property Pages.

See Also

Tasks

.lib Files as Linker Input

Reference

Property Pages (C++)

NMake Property Page

The **NMake** property page enables you to specify build settings for NMake projects.

For more information about NMake projects, see Creating a Makefile Project.

The **NMake** property page contains the following properties:

Build Command Line

Specifies the command to run when the **Build** command is selected from the **Build** menu.

Rebuild All Command Line

Specifies the command to run when the **Rebuild All** command is selected from the **Build** menu.

Clean Command Line

Specifies the command to run when the **Clean** command is selected from the **Build** menu.

Output

Specifies the name of the file that will contain the output for the command line. By default, this option is based on the project name.

For information on how to access the **NMake** property page, see How To: Specify Project Properties with Property Pages.

For information on how to programmatically access members of this object, see VCNMakeTool.

See Also

Tasks

How to: Enable IntelliSense for Makefile Projects

Reference

Property Pages (C++)

Linker Property Pages

This topic discusses the following properties on the **General** linker property page:

Ignore Import Library

Tells the linker not to try to link any .lib output generated from this build into any dependent project. This allows the project system to handle .dll files that do not produce a .lib file when built. If a project is dependent on another project that produces a DLL, the project system automatically will link the .lib file produced by that child project. This may not be needed by projects that are producing COM DLLs or resource-only DLLs; these DLLs do not have any meaningful exports. If a DLL has no exports, the linker will not generate a .lib file. If no export .lib file is present on the disk, and the project system tells the linker to link with this (missing) DLL, the link will fail.

Use **Ignore Import Library** to resolve this problem. When set to **Yes**, the project system will ignore the presence or absence of that .lib file and cause any project that is dependent on this project to not link with the nonexistent .lib file.

To programmatically access this property, see IgnoreImportLibrary.

Register Output

Run regsvr32.exe /s $(TargetPath), which is valid only on .dll projects. For .exe projects, this property is ignored. If you want to register an .exe output, set a postbuild event on the configuration to do the custom registration that is always required for registered .exe files.

To programmatically access this property, see RegisterOutput.

Link Library Dependencies

Gives you the choice of linking in the .lib files that are produced by dependent projects. Typically, you will want to link in the .lib file.

Use Library Dependency Inputs

In a large project, when a dependent project produces a .lib file, incremental linking is disabled. If there are many dependent projects that produce .lib files, building the application can take a long time. When this property is set to **Yes**, the project system links in the .obj files for .libs produced by dependent projects, thus enabling incremental linking.

For information on how to access the **General** linker property page, see
How To: Specify Project Properties with Property Pages.

See Also

Reference

VC++ Directories, Projects and Solutions, Options Dialog Box
Property Pages (C++)

Manifest Tool Property Pages

Use the **Manifest Tool** property pages to set build options for mt.exe.

For more information on the manifest tool, see Mt.exe.

In This Section

General, Manifest Tool, Configuration Properties, <Projectname> Property Pages Dialog Box

Input and Output, Manifest Tool, Configuration Properties, <Projectname> Property Pages Dialog Box

Isolated COM, Manifest Tool, Configuration Properties, <Projectname> Property Pages Dialog Box

Advanced, Manifest Tool, Configuration Properties, <Projectname> Property Pages Dialog Box

See Also

Reference

Property Pages (C++)

General, Manifest Tool, Configuration Properties, <Projectname> Property Pages Dialog Box

Use this dialog box to specify general options for Mt.exe.

To access this property page dialog box, open the property pages for your project or your property sheet. Expand the **Manifest Tool** node under **Configuration Properties**, and then select **General**.

Suppress Startup Banner

Yes (/nologo) specifies that standard Microsoft copyright data will be concealed when the manifest tool is started. Use this option to suppress unwanted output in log files, when you run mt.exe as part of a build process or from a build environment.

Verbose Output

Yes (/verbose) specifies that additional build information will be displayed during manifest generation.

Merge Rules File

Uses the /mergerules option to specify the XML file that contains merge rules and a list of namespaces. This option is required when there are multiple manifest snippets that are generated by the manifest tool and that need to be merged into a single manifest.

Assembly Identity

Uses the /identity option to specify an identity string, which comprises the attributes for the <assemblyIdentity> Element (ClickOnce Reference). An identity string begins with the value for the **name** attribute, and is followed by *attribute* = *value* pairs. The attributes in an identity string are delimited by a comma.

The following is an example identity string:

```
Microsoft.Windows.Common-Controls, processorArchitecture=x86, version=6.0.0.0, type=win32,
publicKeyToken=6595b64144ccf1df
```

Use UNICODE Response Files

Yes specifies that the project system will generate Unicode response files when it runs the manifest tool. Set this option to **Yes** when files in your project have paths defined in Unicode.

Use FAT32 Work-around

FAT and FAT32 file systems have up to a two second latency between the time that a file is changed and the time that the file's time stamp is updated. This latency prevents the build process from determining whether or not relevant files are up-to-date.

Yes introduces a two second delay to allow the manifest tool (Mt.exe) to properly embed the manifest on FAT and FAT32 file systems. If you are building a project on a FAT or FAT32 file system, select **Yes**.

See Also

Tasks

How to: Open Project Property Pages

How to: Edit Project Property Sheets

Reference

ClickOnce Application Manifest

Other Resources

Manifest Tool Property Pages

Input and Output, Manifest Tool, Configuration Properties, <Projectname> Property Pages Dialog Box

Use this dialog box to specify input and output options for Mt.exe.

To access this property page dialog box, open the property pages for your project or your property sheet. Expand the **Manifest Tool** node under **Configuration Properties**, and then select **Input and Output**.

Additional Manifest Files

Uses the /manifest option to specify the full paths of additional manifest files that the manifest tool will process or merge. Full paths are delimited by a semicolon.

Input Resource Manifests

Uses the /inputresource option to specify the full path of a resource of type RT_MANIFEST, to input into the manifest tool. The path can be followed by the specified resource ID. For example:

```
dll_with_manifest.dll;#1
```

The resource ID is optional and defaults to CREATEPROCESS_MANIFEST_RESOURCE_ID in winuser.h.

Embed Manifest

Yes specifies that the project system will embed the application manifest file into the assembly.

No specifies that the project system will create the application manifest file as a stand-alone file.

Output Manifest File

Specifies the name of the output manifest file. This property is optional when only one manifest file is operated upon by the manifest tool.

Output Resource Manifest

Uses the /outputresource option to specify the file name of a resource of type RT_MANIFEST as the output of the manifest tool. The file name can be followed by the specified resource ID. For example:

```
dll_with_merged_manifest.dll;#3
```

The resource ID is optional and defaults to CREATEPROCESS_MANIFEST_RESOURCE_ID in winuser.h.

Generate Catalog Files

Uses the /makecdfs option to specify that the manifest tool will generate catalog definition files (.cdf files), which are used to make catalogs.

Dependency Information File

Specifies the dependency information file that Visual Studio uses to track build dependency information for the manifest tool.

See Also

Tasks

How to: Open Project Property Pages

How to: Edit Project Property Sheets

Reference

ClickOnce Application Manifest

Other Resources

Manifest Tool Property Pages

Isolated COM, Manifest Tool, Configuration Properties, <Projectname> Property Pages Dialog Box

Use this dialog box to specify **Isolated COM** options for Mt.exe.

To access this property page dialog box, open the property pages for your project or your property sheet. Expand the **Manifest Tool** node under **Common Properties**, and then select **Isolated COM**.

- How to: Build Isolated Applications to Consume COM Components

Type Library File

Uses the /tlb option to specify the name of the type library file (.tlb file) that the manifest tool will use to generate the manifest file.

Registrar Script File

Uses the /rgs option to specify the name of the registrar script file (.rgs file) that the manifest tool will use to generate the manifest file.

Component File Name

Uses the /dll option to specify the name of the resource that the manifest tool will generate. You must enter a value for this property when values for either **Type Library File** or **Registrar Script File** are specified.

Replacements File

Uses the /replacements option to specify the full path to the file that contains values for replaceable strings in the .rgs file.

See Also

Tasks

How to: Open Project Property Pages
How to: Edit Project Property Sheets

Reference

Isolated Applications
Side-by-side Assemblies
ClickOnce Application Manifest

Other Resources

Manifest Tool Property Pages

Advanced, Manifest Tool, Configuration Properties, <Projectname> Property Pages Dialog Box

Use this dialog box to specify advanced options for Mt.exe.

To access this property page dialog box, open the property pages for your project or your property sheet. Expand the **Manifest Tool** node under **Configuration Properties**, and then select **Advanced**.

Update File Hashes

Uses the /hashupdate option to specify that the manifest tool will compute the hash of files specified in `<file>` elements, and then update the hash attributes with the computed value.

Update File Hashes Search Path

Specifies the search path for files that are referenced in `<file>` elements. This option also uses the /hashupdate option.

Key File

Specifies the .snk file that contains the strong name key pair for signing.

Key Container

Specifies the key container that contains the strong name key for signing.

Validate Signature

Yes (/validate_singnatures) specifies that all the signature elements in the manifest will be verified.

See Also

Tasks

How to: Open Project Property Pages

How to: Edit Project Property Sheets

Reference

<file> Element

ClickOnce Application Manifest

Other Resources

Manifest Tool Property Pages

Resources Property Pages

For more information on using the resource compiler, see Using RC (The RC Command Line).

For information on how to access the **Resources** property pages, see How To: Specify Project Properties with Property Pages.

To programmatically access these properties, see VCResourceCompilerTool.

See Also

Reference

Property Pages (C++)

MIDL Property Pages

The **MIDL** node contains several property pages:

- MIDL Property Pages: General
- MIDL Property Pages: Output
- MIDL Property Pages: Advanced

For information on how to programmatically access MIDL options for C++ projects, see VCMidlTool

See Also

Reference

Property Pages (C++)

MIDL Property Pages: General

The **General** property page in the **MIDL** folder specifies the following MIDL compiler options:

- Preprocessor Definitions (/D)
- Additional Include Directories (/I)
- Ignore Standard Include Path (/no_def_idir)
- MkTypLib Compatible (/mktyplib203)
- Warning Level (/W)
- Warn As Error (/WX)
- Suppress Startup Banner (/nologo)
- MIDL Char Type (/char)
- Target Environment (/env)
- Generate Stubless Proxies (/Oicf)

For information on how to access the **General** property page in the **MIDL** folder, see How To: Specify Project Properties with Property Pages.

For information on how to programmatically access MIDL options for C++ projects, see VCMidlTool object.

See Also

Reference

MIDL Property Pages

MIDL Property Pages: Output

The **Output** property page in the **MIDL** folder specifies the following MIDL compiler options:

- Output Directory (/out)
- Header File (/h)
- DLL Data File (/dlldata)
- IID File (/iid)
- Proxy File (/proxy)
- Generate Type Library (/notlb)
- Type Library (/tlb)

For information on how to access the **Output** property page in the **MIDL** folder, see How To: Specify Project Properties with Property Pages.

For information on how to programmatically access MIDL options for C++ projects, see VCMidlTool.

See Also

Reference

MIDL Property Pages

MIDL Property Pages: Advanced

The **Advanced** property page in the **MIDL** folder specifies the following MIDL compiler options:

- Enable Error Checking (/error)
- Check Allocations (/error)
- Check Bounds (/error)
- Check Enum Range (/error)
- Check Reference Pointers (/error)
- Check Stub Data (/error)
- Validate Parameters (/robust) *
- Struct Member Alignment (/Zp)
- Redirect Output (/o)
- C Preprocess Options (/cpp_opt)
- Undefine Preprocessor Definitions (/U)

* /robust is only for use when building for a Windows 2000 or later machine. If you build an ATL project and want to use /robust, change this line in the dlldatax.c file:

```
#define _WIN32_WINNT 0x0400   //for WinNT 4.0 or Win95 with DCOM
to
#define _WIN32_WINNT 0x0500   //for WinNT 4.0 or Win95 with DCOM
```

For information on how to access the **Advanced** property page in the **MIDL** folder, see How To: Specify Project Properties with Property Pages.

For information on how to programmatically access MIDL options for C++ projects, see VCMidlTool.

See Also

Reference

MIDL Property Pages

Web References Property Page

The **Web References** property page specifies how the XML Web service proxy class will be generated. An XML Web service proxy class will be generated if you add a web reference to your project. See Add Web Reference for more information.

The **Web References** property page contains the following properties:

Output file

The name of the file to contain the XML Web service proxy class.

Suppress Startup Banner

Do not display the banner for the Web Services Description Language Tool (Wsdl.exe).

Namespace

Specifies the name of the generated web proxy.

Additional References

Specifies the additional DLLs referenced by the proxy DLL.

See Consuming XML Web Services for more information.

For information on how to access the **Web Reference** property page, see How To: Specify Project Properties with Property Pages.

See Also

Reference

Property Pages (C++)

XML Data Generator Tool Property Page

The **XML Data Generator Tool** property page becomes available when you add a dataset to a project.

The **XML Data Generator Tool** property page contains the following properties:

Output File

Specifies the output file name to use.

Suppress Startup Banner

Suppresses the display of the startup banner and information messages.

Generated Proxy Language

Determines whether or not to emit managed code.

For information on how to access the **XML Data Generator Tool** property page, see How To: Specify Project Properties with Property Pages.

For information on how to programmatically access members of this object, see VCXMLDataGeneratorTool

See Also

Reference

Property Pages (C++)

Web Deployment Property Page

The **Web Deployment** property page specifies how the web deployment tool will install the files produced by your project so that they are served by Internet Information Services (IIS). Enabling web deployment frees you from having to create virtual directories, configure application mappings, and copy content during development of your applications.

Note
Web Deployment can only be carried out by a user who is a member of the Administrators group on the local machine. Web Deployment can only deploy to the first web site on the local machine.

Note
Deployment of ATL Server projects to IIS is not supported on a Web server running on any 64-bit platform.

This property page is available for all C++ projects.

The **Web Deployment** property page contains the following properties:

Excluded From Build

Specifies whether or not deployment occurs when the project is built. Set to **Yes** to disable deployment or set to **No** to enable deployment.

Relative Path

The path relative to the virtual directory to which the primary project output will be copied.

Additional Files

The semicolon-separated list of additional files to be deployed. Files can be specified with project-relative or absolute paths and can make use of the macros for build commands and properties.

The primary project output and files in the project marked as deployable content need not be listed here. You can use the **Content** property exposed by the Properties Window to mark files as deployable content.

Files beneath the project directory that are marked as deployable content or as additional files will be copied to a location relative to the virtual directory directly corresponding to their location relative to the project directory. Files that are not in the project directory or a subdirectory under the project will be placed directly in the virtual directory.

Unload Before Copy

Specifies whether or not to unload the ISAPI extension(s) associated with the virtual directory before deploying.

If this property is set to **Yes** and the ISAPI extension is running in the IIS process, the WWW publishing service will be reset; otherwise, only the virtual directory specified in **Virtual Directory Name** will be affected.

Register Output

Specifies whether or not the primary project output should be registered using **regsvr32** after deployment. This property should only be set to **Yes** when the primary project output is an ISAPI extension.

Virtual Directory Name

The alias of the virtual directory. This virtual directory will be created if it doesn't already exist.

Application Mappings

The semicolon-separated list of file extensions to be associated with the primary project output. Each item in the list must include only the period and the extension. This property should only be set when the primary project output is an ISAPI extension.

Application Protection

The level of process isolation used by the virtual directory.

Note

Medium isolation is treated as high isolation on Windows NT 4 because that version of IIS does not support medium isolation.

See the documentation for VCWebDeploymentTool for information about programmatic access to the properties on this page.

For information on how to access the **Web Deployment** property page, see Setting Visual C++ Project Properties.

Web Deployment is a build step that will only occur if the link step occurs. See Understanding Custom Build Steps and Build Events to see when this build step is executed in relation to the other steps.

See Also

Reference

Modifying Project Settings

How to: Open Project Property Pages

This topic discusses how to open Property Pages for Visual C++ projects. You can do this in one of several ways. Each way begins by selecting a project in Solution Explorer.

To open property pages for Visual C++ projects

1. Select a project in Solution Explorer.
2. Complete one of the following procedures:
   - Select **Property Pages** (SHIFT+F4) on the **View** menu.
   - Select **Properties** on the **Project** menu.
   - Right-click the project node in Solution Explorer and click **Properties**.
   - Select the **Property Pages** icon in the Properties window.
   - Select the **Properties** icon in Solution Explorer.
   - Type View.PropertyPages or Project.Properties in the Command Window.

How to: Specify Project Properties with Property Pages

A project's **Property Pages** dialog box is accessed as follows:

To specify project properties with property pages

1. Display your project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Using the Configuration dialog box, select the configuration for which you want to set this project's properties.
3. In the left pane, select a folder and property page.
4. In the right pane, set the available properties.
5. Click **OK** to save the settings. If you want a change to take effect immediately, click **Apply**.

Note
Properties that are unavailable are defined when the project is created; they are displayed for information only.

Specifying Project Settings with Property Pages discusses editing properties and property inheritance.

For information on the **Build** menu in the Visual Studio shell, see Default and Custom Builds.

Documentation is available for each property; click the **Help** button. For additional information on properties in the **Property Pages** dialog box, see:

- General Property Page (Project)
- General Property Page (File)
- Command Line Property Pages
- NMake Property Page
- Linker Property Pages
- Resources Property Pages
- MIDL Property Pages
- Web Reference Property Page
- XML Data Generator Tool Property Page
- Web Deployment Property Page

See Also

Reference

Property Pages (C++)
Project Settings for a C++ Debug Configuration

How to: Set Environment Variables for Projects

Use the **Environment** or **Merge Environment** project settings, which are available from the <*Projectname*> **Property Pages** dialog box, to set up environment variables for projects. For more information, see Project Settings for a C++ Debug Configuration.

To set environment variables for a project

1. Open the <*Projectname*> **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Expand the **Configuration Properties** node.
3. Click the **Debugging** node to open the debugging property page.
4. Modify the **Environment** or **Merge Environment** project settings.
5. Click **OK** to accept any modified project settings and close the <*Projectname*> **Property Pages** dialog box.

See Also

Reference

Modifying Project Settings

XML Document Generator Tool Property Pages

The XML Document Generator Tool property page exposes the functionality of xdcmake.exe. xdcmake.exe merges .xdc files into an .xml file when your source code contains documentation comments and /doc (Process Documentation Comments) (C/C++) is specified,. See Recommended Tags for Documentation Comments (C++) for information on adding documentation comments to source code.

Note
xdcmake.exe options in the development environment (property pages) differ from the options when xdcmake.exe is used at the command line. For information on using xdcmake.exe at the command line, see XDCMake Reference.

Suppress Startup Banner

Suppress copyright message.

Validate IntelliSense

Perform correctness check on tags that could be used by Visual Studio IntelliSense.

Additional Document Files

Additional directories in which you want the project system to look for .xdc files. xdcmake will always look for .xdc files generated by the project. Multiple directories can be specified.

Output Document File

The name and directory location of the .xml output file. See Macros for Build Commands and Properties for information on using macros to specify directory locations.

Document Library Dependencies

If your project has a dependency on a .lib project in the solution, you can process .xdc files from the .lib project into the .xml files for the current project.

For more information on Project Dependencies Dialog Box.

Use UNICODE Response Files

Enables Unicode characters in filenames. If the .cpp file containing document comments uses Unicode characters, the .xdc file name will use those Unicode characters.

For more information, see Unicode Support in the Compiler and Linker.

See Also

Reference

Property Pages (C++)

Property Pages (C++)

Property Sheets (C++)

A project property sheet is an .xml file with the extension .vsprops. It enables you to specify switches for build tools such as the compiler or linker and create user-defined macros.

You can use property sheets to create project configurations that can be applied to multiple projects since project settings that are defined in .vsprops files are inheritable, unlike project settings defined in Project Files (.vcproj files). Therefore, a project configuration defined in a .vcproj file can inherit project settings from one or more property sheets (.vsprops files). For more information, see Property Inheritance.

For information on tasks that demonstrate this concept, see:

- How to: Create Project Property Sheets
- How to: Edit Project Property Sheets
- How to: Inherit Project Configurations from Property Sheets

Example

The following .vsprops file contains both build tool properties and user-defined macros.

```
<?xml version="1.0" ?>
<VisualStudioPropertySheet ProjectType="Visual C++" Version="8.00"
   Name="Visual C++ Project Properties" OutputDirectory="$(VCPACKAGES)"
   UseMFC="FALSE" UseATL="FALSE">

      <Tool Name="VCCLCompilerTool" UsePrecompiledHeader="3"
         ForcedIncludeFiles="warning.h" />
      <Tool Name="VCMIDLTool" />
      <Tool Name="VCResourceCompilerTool" />
      <Tool Name="VCLinkerTool" OptimizeForWindows98="1" />

      <UserMacro Name="VCROOT" Value="$(DDROOT)\vc" />
      <UserMacro Name="VCPROJDEFAULTS"
         Value="$(BINDIR)\VC8\VCProjectDefaults" />
      <UserMacro Name="VCPACKAGES" Value="$(BINDIR)\VC8\VCPackages" />
      <UserMacro Name="INCLUDEPATH" Value="$(VCROOT)\Inc"
         InheritsFromParent="TRUE" Delimiter=";" />

</VisualStudioPropertySheet>
```

Use the XSD Schema for Property Sheets to validate your own .vsprops files.

See Also

Reference

Modifying Project Settings
Property Pages (C++)

XML Schema Definition for Property Sheets

The following schema is used for property sheets.

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="VisualStudioPropertySheet" type="VisualStudioPropertySheetType" />
   <xs:complexType name="VisualStudioPropertySheetType">
      <xs:sequence minOccurs="0" maxOccurs="1">
         <xs:element name="Tool" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
               <xs:anyAttribute processContents="skip" />
            </xs:complexType>
         </xs:element>
         <xs:element name="UserMacro" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
               <xs:attribute name="Name" type="xs:string" use="required" />
               <xs:attribute name="Value" type="xs:string" use="required" />
               <xs:attribute name="InheritsFromParent" type="xs:string" use="optional" />
               <xs:attribute name="Delimiter" type="xs:string" use="optional" />
               <xs:attribute name="PerformEnvironmentSet" type="xs:string" use="optional" /
>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
   <xs:attribute name="ProjectType" type="xs:string" use="required" fixed="Visual C++" />
   <xs:attribute name="Version" type="xs:string" use="required" fixed="8.00" />
   <xs:attribute name="Name" type="xs:string" use="required" />
   <xs:anyAttribute processContents="skip" />
   </xs:complexType>
</xs:schema>
```

See Also

Concepts

Property Sheets (C++)

How to: Add New Property Sheets to C++ Projects

This topic describes how to create new Property Sheets (C++), and add them to your C++ projects.

To create a project property sheet

1. Open your solution in the Visual Studio IDE.
2. From the **View** menu, select the **Property Manger** menu item. The Property Manager window appears docked on the left of the IDE.
3. In Property Manager, right-click the project that will contain the new property sheet, and then select the **Add New Project Property Sheet** menu item.
4. In the Add New Item dialog box, enter the name of the new property sheet in the **Name** text box.
5. Click **Add**.

The new property sheet will appear as a node in the tree view of the Property Manager window.

See Also

Reference

Property Manager

How to: Edit Project Property Sheets

The easiest way to edit project Property Sheets (C++) is by modifying project settings in property pages for property sheets.

To edit a project property sheet

1. Open your solution in the Visual Studio IDE.
2. From the **View** menu, select the **Property Manger** menu item. The Property Manager window will appear, docked on the left of the IDE.
3. In Property Manager, double-click a property sheet. The property pages for the property sheet will appear.
4. Enter or set new property values in the fields that you want to modify.

Note
Aggregate properties are separated with a semicolon. For more information, see Property Inheritance.

5. Click the **Apply** button.

Note
When you click **Apply**, the new settings will automatically propagate to all project files and property sheets that inherit from the property sheet that you modified.

6. Click the **OK** button.

See Also

Tasks

How to: Add New Property Sheets to C++ Projects

How to: Inherit Project Settings from Property Sheets

This topic describes how to apply project settings that are defined in property sheets, to your C++ projects. You can either use the Property Manager or set the **Inherited Project Property Sheets** property to specify that your Visual C++ project or project property sheet will inherit its project settings from another property sheet.

To apply project settings that are defined in property sheets using the Property Manager

1. Open your project in the Visual Studio IDE.
2. Select **Property Manager** from the **View** menu.
3. In Property Manager, right-click the project, configuration, or user-defined property sheet that you want to apply project settings.

Note
Default project property sheets cannot inherit project settings from user-defined property sheets.

4. Select **Add Existing Property Sheet** from the context menu.

   When you select **Add Existing Property Sheet**, the Add Existing Property Sheet dialog box appears. This dialog box helps you locate and specify which property sheet you want to apply to your project, configuration, or user-defined property sheet.

To apply project settings that are defined in property sheets using the Inherited Project Property Sheets property

1. Open your project in the Visual Studio IDE.
2. In Solution Explorer, right-click the node for the project or the property sheet that will inherit the project configuration.
3. On the shortcut menu, click **Properties**. The property pages for your project will appear.
4. Use the **Configuration** control to specify which project configuration to apply the project settings.
5. In the **Inherited Project Property Sheets** field of the **General** property page, enter the path to the property sheet that contains the settings you want to inherit.
6. Click **Apply**.

Note
When you click **Apply**, the new settings will automatically propagate to all project files and property sheets that inherit from the property sheet that you modified.

See Also

Tasks

How to: Add New Property Sheets to C++ Projects
How to: Edit Project Property Sheets

Reference

General Property Page (Project)
Property Manager

Concepts

Property Sheets (C++)

How to: Add Property Sheets to Source Code Control

When you set up a Visual C++ project to work together with Visual Source Safe, the project system will automatically update the .vcproj file in source code control. However, if you add user defined property sheets to your C++ project, the project system will not automatically update these files in source code control.

To set up property sheets in Visual Source Safe

- Create a directory under the solution directory to store all your project property sheets for that solution.

  - or -

- For broadly-used property sheets, such as property sheets that are common to several teams in a company, create a directory that is a peer directory of the solution directories to store all your project property sheets used by your teams within the company.

See Also

Concepts

Property Sheets (C++)

User-defined Macros

Visual C++ enables you to create custom macros for building projects in the integrated development environment. Macros that you create are called user-defined macros. You can use the Add User Macro Dialog Box to easily create new custom macros.

User-defined macros are defined in .vsprops files. For more information, see XML Schema Definition for Property Sheets.

See Also

Reference

XML Schema Definition for Property Sheets

Other Resources

Macros for Build Commands and Properties

Add User Macro Dialog Box

This dialog box helps you create custom build macros.

Name

Specifies the name of your custom macro.

Value

Specifies the value of your custom macro.

Macro inherits from parent property sheets

Enables the custom macro to inherit from parent property sheets.

Delimiter

Specifies the character that represents the delimiter that separates values for your custom macro. This option is only available when **Macro inherits from parent property sheets** is selected.

Set this macro as an environment variable

When this option is selected, your custom macro serves as an environment variable.

See Also

Reference

User-defined Macros

Property Inheritance

The project system employs a property inheritance model, which extends your ability to manage project configurations. When you define a project configuration in a property sheet or in a tool file, you may want to apply those settings to another project. Property inheritance enables you to reuse project configurations, without having to redefine and reapply each project setting in the property pages.

Property inheritance applies to all files that define a project configuration. The following table summarizes which files can inherit property settings from each other.

File Type	Can inherit from .vcproj files?	Can inherit from .vsprops files?	Can inherit from .tool files?
Project Files (.vcproj)	No	Yes	Yes
Property Sheets (.vsprops)	No	Yes	No
Tool Files (.tool)	No	No	Yes

See Also

Tasks

How to: Inherit Project Settings from Property Sheets

Concepts

Property Types (C++)

Property Types (C++)

The project system differentiates properties into two types: simple properties and aggregate properties. Each type of property is evaluated differently for property inheritance.

Simple Properties

Simple properties are all non-string properties, such as integer, boolean, and enum properties, and only hold a single value. If they are defined at the local level (this file configuration, this configuration, or this property sheet), then the inherited value is ignored. An example of such a property is the WarnAsError, which sets the /WX switch for CL.EXE.

Aggregate Properties

Aggregate properties are always string properties. The value of an aggregate property, defined at any level, is concatenated with the value of its parent. An example of an aggregate property is the AdditionalIncludeDirectories which sets the /I switch for CL.EXE. Path properties are good examples of aggregate properties because they comprise a list of several values, rather than being a single value.

There are three special built-in macros that control property evaluation for aggregate properties:

$(Inherit)

Specifies to evaluate the property values of the parent, and insert those values in place of the macro. When $(Inherit) is not specified anywhere, it is implied at the end of the value, which means the values of the parent will be appended to the value at the current level.

$(NoInherit)

Specifies to not append the property values of the parent to the aggregate property. The $(NoInherit) macro overrides the $(Inherit) macro.

$(StopEvaluating)

Specifies to stop all evaluation on the particular aggregate property where the macro is encountered. Any values after the $(StopEvaluating) macro are lost.

See Also

Concepts

Evaluation of Inherited Properties
Property Inheritance

Evaluation of Inherited Properties

A project configuration can inherit project settings from multiple property sheets.

When the project system evaluates properties that are inherited from multiple property sheets, the last property sheet inherited from will be the first property sheet evaluated against.

For example, if you specify `a.vsprops; b.vsprops; c.vsprops` as the value of the **Inherited Property Sheets** property for a project configuration, then the properties in `c.vsprops` has highest precedence, `b.vsprops` has higher precedence than `a.vsprops`, and `a.vsprops` has lowest precedence. In this case, `a.vsprops`, `b.vsprops`, and `c.vsprops` are peers, which means that they exist at the same level.

Property evaluation is done by a depth first scan, rather than breadth first.

In the preceding example, if `c.vsprops` inherits from `d.vsprops`, then the project system will look in `d.vsprops` before looking in `b.vsprops`.

See Also

Concepts

Property Types (C++)
Property Inheritance

Property Manager

The **Property Manager** window enables you to modify project settings that are defined in property sheets. This window is available from the **View** menu.

Properties

Opens the project settings for the selected node in the hierarchy tree.

Add New Project Property Sheet

Opens the **Add New Item** dialog box. Use this dialog box to add a new property sheet to your project configuration.

Add Existing Project Property Sheet

Opens a dialog box that is similar to the **Open File** dialog box. Use this dialog box to add an existing property sheet to your project configuration.

Move Up

Moves the selected property sheet up the inheritance hierarchy, which increases the precedence of its properties for property evaluation. This option is only available for property sheets that are child nodes of property sheets, and have peers that reside above them in the inheritance hierarchy.

Move Down

Moves the selected property sheet down the inheritance hierarchy, which decreases the precedence of its properties for property evaluation. This option is only available for property sheets that are child nodes of property sheets, and have peers that reside below them in the inheritance hierarchy.

Save

If a property sheet is selected, then this command saves project properties for the selected sheet. If a configuration is selected, such as the **Debug** configuration for the **Win32** platform, then this command saves all properties for the selected configuration.

See Also

Reference

Clear Overridden Properties Dialog Box

Concepts

Property Sheets (C++)

Evaluation of Inherited Properties

Clear Overridden Properties Dialog Box

When you add a user-defined property sheet (.vsprops file) to a project configuration, that project configuration will inherit all of the properties defined in the property sheet by default. Use this dialog box to clear undesirable inherited properties for a project configuration.

See Also

Concepts

Property Sheets (C++)

Property Inheritance

Project Load Warning for <Projectname> Dialog Box

This dialog box will appear if the settings, for projects opened in the IDE, are load incorrectly.

Please correct specified errors and reload project

Provides information regarding the complication such as the invalid path or file name. To resolve errors, remove or replace any invalid references from the property pages of your project. Save your project, and then reload it in the IDE.

Do not show this dialog again this session

When selected, the Project Load Warning for <Projectname> Dialog Box will no longer appear until the next time you open the IDE.

See Also

Concepts

Property Inheritance

Project Load Failure Dialog Box

This dialog box will appear if a project file cannot be loaded.

Please correct the specified error(s) and reload the project.

Provides information regarding the error. Use the information in this dialog box to locate and resolve the error in the project file. Save your project, and then reload it in the IDE.

Do not show this dialog again this session

When selected, the **Project Load Failure** dialog box will not appear until the next time you open the IDE.

See Also

Concepts

Property Inheritance

Rule Files

A rule file, or RULES file, is an XML file that has a .rules file extension. It allows you to define custom build rules and incorporate them into the build process of a Visual C++ project. Custom build rules are associated with one or more file extensions. A custom build rule enables you pass input files to a tool, which results in one or more output files, as part of the build process.

Use the Visual C++ Custom Build Rule Files Dialog Box to manage the rule files in your projects.

Note

Rule files follow the same property inheritance guidelines as property sheets (VSPROPS files). For more information, see Property Inheritance.

When you incorporate custom build rules into the build process, you can also specify the order in which those build rules are run. For more information, see Tool Build Order Dialog Box.

Example

The following example RULES file defines a custom build rule:

```
<?xml version="1.0" encoding="utf-8"?>
<VisualStudioToolFile
   Name="Microsoft Macro Assembler"
   >
   <Rules>
      <CustomBuildRule
         Name="MASM"
         DisplayName="Microsoft Macro Assembler"
         CommandLine="ml.exe /nologo /c [AllOptions] [AdditionalOptions] /Fo&quot;$(IntDir)
\$(InputName).obj&quot; /Ta [inputs]"
         Outputs="$(IntDir)\$(InputName).obj"
         FileExtensions="*.asm"
         ExecutionDescription="Assembling..."
         ShowOnlyRuleProperties="true"
         >
         <Properties>
            <BooleanProperty
               Name="TinyMemoryModelSupport"
               DisplayName="Tiny Memory Model Support"
               PropertyPageName="Advanced"
               Description="Enables tiny-memory-model support. Note that this is not equiva
lent to the .MODEL TINY directive.     (/AT)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/AT"
            />
            <BooleanProperty
               Name="UseCommonObjectFileFormat"
               DisplayName="Use Common Object File Format (COFF)"
               PropertyPageName="Object File"
               Description="Generates a common object file format type of object module.
(/coff)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/coff"
            />
            <BooleanProperty
               Name="PreserveIdentifierCase"
               DisplayName="Preserve Identifier Case"
               PropertyPageName="Identifiers"
               Description="Preserves case of all user identifiers.     (/Cp)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Cp"
            />
```

```
            <BooleanProperty
               Name="MapIdentifiersToUpperCase"
               DisplayName="Map Identifiers To Upper Case"
               PropertyPageName="Identifiers"
               Description="Maps all identifiers to upper case.     (/Cu)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Cu"
               DefaultValue="true"
            />
            <BooleanProperty
               Name="PreservePublicAndExternSymbolCase"
               DisplayName="Preserve Public and Extern Symbol Case"
               PropertyPageName="Identifiers"
               Description="Preserves case in public and extern symbols.     (/Cx)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Cx"
            />
            <StringProperty
               Name="PreprocessorDefinitions"
               DisplayName="Preprocessor Definitions"
               Description="Defines a text macro with the given name.     (/D[symbol])"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/D&quot;[value]&quot;"
               Delimited="true"
               Inheritable="true"
            />
            <BooleanProperty
               Name="GeneratePreprocessedSourceListing"
               DisplayName="Generate Preprocessed Source Listing"
               PropertyPageName="Listing File"
               Description="Generates a preprocessed source listing to the Output Window.
(/EP)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/EP"
            />
            <StringProperty
               Name="AssembledCodeListingFile"
               DisplayName="Assembled Code Listing File"
               PropertyPageName="Listing File"
               Description="Generates an assembled code listing file.     (/Fl[file])"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Fl&quot;[value]&quot;"
               Delimited="true"
            />
            <BooleanProperty
               Name="GenerateEmulatorFixups"
               DisplayName="Generate Emulator Fixups"
               PropertyPageName="Advanced"
               Description="Generates emulator fix-ups for floating-point arithmetic (mixed
language only).     (/FPi)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/FPi"
            />
            <StringProperty
               Name="SourceBrowserFile"
               DisplayName="Source Browser File"
               PropertyPageName="Source Browser File"
               Description="Generates a source browser .sbr file.     (/Fr[file])"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Fr&quot;[value]&quot;"
               Delimited="true"
```

```
            />
            <StringProperty
               Name="ExtendedSourceBrowserFile"
               DisplayName="Extended Source Browser File"
               PropertyPageName="Source Browser File"
               Description="Generates an extended form of a source browser .sbr file.     (
/FR[file])"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/FR&quot;[value]&quot;"
               Delimited="true"
            />
            <BooleanProperty
               Name="UsePascalCallingConvention"
               DisplayName="Use Pascal Calling Convention"
               PropertyPageName="Advanced"
               Description="Specifies use of FORTRAN- or Pascal-style function calling and
naming conventions. Same as OPTION LANGUAGE:PASCAL.     (/Gc)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Gc"
            />
            <BooleanProperty
               Name="UseCStyleCallingConvention"
               DisplayName="Use C-Style Calling Convention"
               PropertyPageName="Advanced"
               Description="Specifies use of C-style function calling and naming convention
s. Same as OPTION LANGUAGE:C.     (/Gd)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Gd"
            />
            <IntegerProperty
               Name="ExternalNameCharacterLimit"
               DisplayName="External Name Character Limit"
               PropertyPageName="Identifiers"
               Description="Restricts external names to number significant characters. The
default is 31 characters.     (/H [number])"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/H [value]"
               DefaultValue="31"
            />
            <StringProperty
               Name="IncludePaths"
               DisplayName="Include Paths"
               Description="Sets path for include file. A maximum of 10 /I options is allow
ed.     (/I [path])"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/I &quot;[value]&quot;"
               Delimited="true"
               Inheritable="true"
            />
            <BooleanProperty
               Name="UseObjectModuleFileFormat"
               DisplayName="Use Object Module File Format"
               PropertyPageName="Object File"
               Description="Generates object module file format (OMF) type of object module
.     (/omf)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/omf"
            />
            <BooleanProperty
               Name="ListAllAvailableInformation"
               DisplayName="List All Available Information"
               PropertyPageName="Listing File"
```

```
               Description="Turns on listing of all available information.     (/Sa)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Sa"
            />
            <BooleanProperty
               Name="UseSafeExceptionHandlers"
               DisplayName="Use Safe Exception Handlers"
               PropertyPageName="Advanced"
               Description="Marks the object as either containing no exception handlers or
containing exception handlers that are all declared with .SAFESEH.     (/safeseh)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/safeseh"
            />
            <BooleanProperty
               Name="AddInstructionTimings"
               DisplayName="Add Instruction Timings"
               PropertyPageName="Listing File"
               Description="Adds instruction timings to listing file.     (/Sc)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Sc"
            />
            <BooleanProperty
               Name="AddFirstPassListing"
               DisplayName="Add First Pass Listing"
               PropertyPageName="Listing File"
               Description="Adds first-pass listing to listing file.     (/Sf)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Sf"
            />
            <BooleanProperty
               Name="EnableAssemblyGeneratedCodeListing"
               DisplayName="Enable Assembly Generated Code Listing"
               PropertyPageName="Listing File"
               Description="Turns on listing of assembly-generated code.     (/Sg)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Sg"
            />
            <IntegerProperty
               Name="SourceListingLineWidth"
               DisplayName="Source Listing Line Width"
               PropertyPageName="Listing File"
               Description="Sets the line width of source listing in characters per line. R
ange is 60 to 255 or 0. Default is 0. Same as PAGE width.     (/Sl [width])"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Sl [value]"
            />
            <BooleanProperty
               Name="DisableSymbolTable"
               DisplayName="Disable Symbol Table"
               PropertyPageName="Listing File"
               Description="Turns off symbol table when producing a listing.     (/Sn)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Sn"
            />
            <IntegerProperty
               Name="SourceListingPageLength"
               DisplayName="Source Listing Page Length"
               PropertyPageName="Listing File"
               Description="Sets the page length of source listing in lines per page. Range
is 10 to 255 or 0. Default is 0. Same as PAGE length.     (/Sp [length])"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
```

```
masm/html/vclrfml.asp"
               Switch="/Sp [value]"
            />
            <StringProperty
               Name="SourceListingSubTitle"
               DisplayName="Source Listing Subtitle"
               PropertyPageName="Listing File"
               Description="Specifies subtitle text for source listing. Same as SUBTITLE te
xt.     (/Ss [subtitle])"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Ss [value]"
               Delimited="true"
            />
            <StringProperty
               Name="SourceListingTitle"
               DisplayName="Source Listing Title"
               PropertyPageName="Listing File"
               Description="Specifies title for source listing. Same as TITLE text.     (/S
t [title])"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/St [value]"
               Delimited="true"
            />
            <BooleanProperty
               Name="EnableFalseConditionalsInListing"
               DisplayName="Enable False Conditionals In Listing"
               PropertyPageName="Listing File"
               Description="Turns on false conditionals in listing.     (/Sx)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Sx"
            />
            <EnumProperty
               Name="WarningLevel"
               DisplayName="Warning Level"
               Description="Sets the warning level, where level = 0, 1, 2, or 3.    (/W0, /
W1, /W2, /W3)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               >
               <Values>
                  <EnumValue
                     Value="0"
                     Switch="/W0"
                     DisplayName="Warning Level 0 (/W0)"
                  />
                  <EnumValue
                     Value="1"
                     Switch="/W1"
                     DisplayName="Warning Level 1 (/W1)"
                  />
                  <EnumValue
                     Value="2"
                     Switch="/W2"
                     DisplayName="Warning Level 2 (/W2)"
                  />
                  <EnumValue
                     Value="3"
                     Switch="/W3"
                     DisplayName="Warning Level 3 (/W3)"
                  />
               </Values>
            </EnumProperty>
            <BooleanProperty
               Name="TreatWarningsAsErrors"
               DisplayName="Treat Warnings As Errors"
```

```
               Description="Returns an error code if warnings are generated.     (/WX)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/WX"
            />
            <BooleanProperty
               Name="GenerateLineInformation"
               DisplayName="Generate Line Information"
               PropertyPageName="Object File"
               Description="Generates line-number information in object file.     (/Zd)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Zd"
            />
            <BooleanProperty
               Name="MakeAllSymbolsPublic"
               DisplayName="Make All Symbols Public"
               PropertyPageName="Object File"
               Description="Makes all symbols public.     (/Zf)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Zf"
            />
            <BooleanProperty
               Name="GenerateCodeViewInformation"
               DisplayName="Generate CodeView Information"
               PropertyPageName="Object File"
               Description="Generates CodeView information in object file.     (/Zi)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Zi"
            />
            <BooleanProperty
               Name="EnableMASM51Compatibility"
               DisplayName="Enable MASM 5.1 Compatibility"
               Description="Enables M510 option for maximum compatibility with MASM 5.1.
(/Zm)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Zm"
            />
            <EnumProperty
               Name="PackAlignmentBoundary"
               DisplayName="Pack Alignment Boundary"
               PropertyPageName="Advanced"
               Description="Packs structures on the specified byte boundary. The alignment
can be 1, 2, or 4.     (/Zp1, /Zp2, /Zp4)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               >
               <Values>
                  <EnumValue
                     Value="0"
                     Switch="/Zp1"
                     DisplayName="One Byte Boundary (/Zp1)"
                  />
                  <EnumValue
                     Value="1"
                     Switch="/Zp2"
                     DisplayName="Two Byte Boundary (/Zp2)"
                  />
                  <EnumValue
                     Value="2"
                     Switch="/Zp4"
                     DisplayName="Four Byte Boundary (/Zp4)"
                  />
               </Values>
            </EnumProperty>
```

```
            <BooleanProperty
               Name="PerformSyntaxCheckOnly"
               DisplayName="Perform Syntax Check Only"
               Description="Performs a syntax check only.     (/Zs)"
               HelpURL="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vc
masm/html/vclrfml.asp"
               Switch="/Zs"
            />
         </Properties>
      </CustomBuildRule>
   </Rules>
</VisualStudioToolFile>
```

Use the XML Schema Definition for Rule Files to validate your own RULES files.

See Also

Reference

XML Schema Definition for Property Sheets

Project Files

Understanding Custom Build Steps and Build Events

XML Schema Definition for Rule Files

Use the following XML Schema Definition (XSD) to validate RULES files.

```
<?xml version="1.0" encoding="utf-8" ?>
   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name="VisualStudioToolFile" type="VisualStudioToolFile" />
      <xs:complexType name="VisualStudioToolFile">
         <xs:choice minOccurs="0" maxOccurs="1">
            <xs:element name="Rules" maxOccurs="1" minOccurs="0">
               <xs:complexType>
                  <xs:sequence maxOccurs="unbounded" minOccurs="0">
                     <xs:element name="CustomBuildRule" type="CustomBuildRule" />
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
         </xs:choice>
         <xs:attribute name="Name" type="xs:string" use="required" />
      </xs:complexType>
   <xs:complexType name="CustomBuildRule">
      <xs:sequence>
         <xs:element name="Properties" maxOccurs="1" minOccurs="0">
            <xs:complexType>
               <xs:choice minOccurs="0" maxOccurs="unbounded">
                  <xs:element name="StringProperty" type="StringProperty" />
                  <xs:element name="EnumProperty" type="EnumProperty" />
                  <xs:element name="BooleanProperty" type="BooleanProperty" />
                  <xs:element name="IntegerProperty" type="IntegerProperty" />
               </xs:choice>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
      <xs:attribute name="Name" type="xs:string" use="required" />
      <xs:attribute name="DisplayName" type="xs:string" use="optional" />
      <xs:attribute name="CommandLine" type="xs:string" use="required" />
      <xs:attribute name="Outputs" type="xs:string" use="required" />
      <xs:attribute name="AdditionalDependencies" type="xs:string" use="optional" />
      <xs:attribute name="FileExtensions" type="xs:string" use="required" />
      <xs:attribute name="ExecutionDescription" type="xs:string" use="optional" default="Ex
ecuting tool..." />
      <xs:attribute name="SupportsFileBatching" type="xs:boolean" use="optional" />
      <xs:attribute name="BatchingSeparator" type="xs:string" use="optional" default="" />
      <xs:attribute name="ShowOnlyRuleProperties" type="xs:boolean" use="optional" default=
"false" />
   </xs:complexType>
   <xs:complexType name="StringProperty">
      <xs:complexContent>
         <xs:extension base="Property">
            <xs:sequence />
            <xs:attribute name="Switch" type="xs:string" use="optional" />
            <xs:attribute name="DefaultValue" type="xs:string" use="optional" />
            <xs:attribute name="Delimited" type="xs:boolean" use="optional" default="false"
/>
            <xs:attribute name="Delimiters" type="xs:string" use="optional" default=";," />

            <xs:attribute name="Inheritable" type="xs:boolean" use="optional" default="fals
e" />
           </xs:extension>
        </xs:complexContent>
     </xs:complexType>
   <xs:complexType name="EnumProperty">
      <xs:complexContent>
         <xs:extension base="Property">
            <xs:sequence>
               <xs:element name="Values" minOccurs="1" maxOccurs="1">
```

```
                  <xs:complexType>
                     <xs:sequence maxOccurs="unbounded" minOccurs="0">
                        <xs:element name="EnumValue" type="EnumValue" />
                     </xs:sequence>
                  </xs:complexType>
               </xs:element>
            </xs:sequence>
            <xs:attribute name="DefaultValue" type="xs:integer" use="optional" default="0"
/>
           </xs:extension>
        </xs:complexContent>
     </xs:complexType>
   <xs:complexType name="EnumValue">
      <xs:sequence />
      <xs:attribute name="Switch" type="xs:string" use="optional" />
      <xs:attribute name="Value" type="xs:integer" use="required" />
      <xs:attribute name="DisplayName" type="xs:string" use="optional" />
   </xs:complexType>
   <xs:complexType name="BooleanProperty">
      <xs:complexContent>
         <xs:extension base="Property">
            <xs:sequence />
            <xs:attribute name="Switch" type="xs:string" use="optional" />
            <xs:attribute name="DefaultValue" type="xs:boolean" use="optional" default="fal
se" />
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
   <xs:complexType name="IntegerProperty">
      <xs:complexContent>
         <xs:extension base="Property">
            <xs:sequence />
            <xs:attribute name="Switch" type="xs:string" use="optional" />
            <xs:attribute name="DefaultValue" type="xs:integer" use="optional" default="0"
/>
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
   <xs:complexType name="Property">
      <xs:sequence />
      <xs:attribute name="Name" type="xs:string" use="required" />
      <xs:attribute name="IsReadOnly" type="xs:boolean" use="optional" default="false" />
      <xs:attribute name="DisplayName" type="xs:string" use="optional" />
      <xs:attribute name="Description" type="xs:string" use="optional" />
      <xs:attribute name="PropertyPageName" type="xs:string" use="optional" default="Genera
l" />
      <xs:attribute name="Category" type="xs:string" use="optional" />
      <xs:attribute name="HelpContext" type="xs:integer" use="optional" />
      <xs:attribute name="HelpFile" type="xs:string" use="optional" />
      <xs:attribute name="HelpURL" type="xs:string" use="optional" />
   </xs:complexType>
</xs:schema>
```

See Also

Reference

Rule Files

Visual C++ Custom Build Rule Files Dialog Box

This dialog box enables you to add, modify, or search for Rule Files, which contain custom build rules for your C++ projects.

To access this dialog box, right-click the project node in Solution Explorer to bring up the project context menu, and then select the **Custom Build Rules** menu item.

-

Available Rule Files

Displays the list of available rule files. In the **Name** column, check the rule files that you want to add to your project.

Note
Use the VC++ Project Settings, Projects and Solutions, Options Dialog Box to modify the **Rule File Search Paths** option.

New Rule File

Opens the New Rule File Dialog Box.

Modify Rule File

Opens a dialog box similar to the **New Rule File** dialog box.

Find Existing

Enables you to search for .rules files.

Refresh List

Refreshes the list of available .rules files.

Contained Custom Build Rules

Displays the custom build rules contained in the rule file that is selected in the **Available Rule Files** list.

New Rule File Dialog Box

Use this dialog box to create new Rule Files.

To access this dialog box:

- Click the **New Rules File** button on the Visual C++ Custom Build Rule Files Dialog Box
- Click the **Create New Rules File** button on the Matching Custom Build Rule Dialog Box
- Click the **Yes** button on the Matching Custom Build Rule Not Found Dialog Box
-

Display Name

Specifies the name of the RULES file, which displays under the **Name** column of the **Available Rules Files** list, in the **Visual C++ Custom Build Rule Files** dialog box.

File Name

Specifies the name of the RULES file as it appears on disk.

Directory

Specifies the directory path that stores the RULES file.

Browse

Helps you to specify the directory path that stores the RULES file.

Custom Build Rules

Lists the custom build rules defined in the new RULES file.

Add Build Rule

Enables you to add custom build rules to the new RULES file. Opens the Add Custom Build Rule Dialog Box.

Modify Build Rule

Enables you to modify the custom build rule that is selected in the **Custom Build Rules** list.

Remove Build Rule

Removes the custom build rule that is selected in the **Custom Build Rules** list.

Add Custom Build Rule Dialog Box

Use this dialog box to add custom build rules to your C++ projects through project Rule Files.

To access this dialog box, click the Add Build Rule button on the New Rule File Dialog Box.

-

Custom Build Rule Properties

Displays the properties of the custom build rule.

User Defined Properties

Lists user-defined properties for the custom build rule.

Add Property

Enables you to add a user-defined property to the custom build rule. Opens the Add User Property Dialog Box.

Modify Property

Enables you to modify the user-defined property that is selected in the **User Defined Properties** list.

Remove Property

Removes the user-defined property that is selected in the **User Defined Properties** list.

Add User Property Dialog Box

Use this dialog box to create a user-defined property for a custom build rule.

To access this dialog box, click the **Add Property** button on the Add Custom Build Rule Dialog Box.

-

User Property Type

Specifies the return type of the property.

User Property Properties

Lists the properties of the user-defined property.

Enum Values

Lists the values for an enumeration user-defined property. This option is available when the **User Property Type** option is set to **Enum**.

Add Enum Value

Enables you to add a constraint to the enumerator list. Opens the Add Enum Value Dialog Box.

Modify Enum Value

Enables you to modify the constraint that is selected in the **Enum Values** list.

Remove Enum Value

Removes the constraint that is selected in the **Enum Values** list.

See Also

Reference

Rule Files

Add Enum Value Dialog Box

Use this dialog box to define a constraint for an enumeration user-defined property.

To access this dialog box, click the Add Enum Value button on the Add User Property Dialog Box.

-

Enum Value Properties

Lists the properties of the constraint that you want to add to an enumeration user-defined property.

See Also

Reference

Rule Files

Matching Custom Build Rule Dialog Box

This dialog box helps you associate existing custom build rules, which are defined in Rule Files, to files that have been newly added to C++ projects and that the project system does not know how to build.

When you add a file to your project, the project system scans **Rule File Search Paths** for custom build rules that match the file's extension. If you add a file that is currently not associated with an existing rule that could specify how the project system should build that type of file, then this dialog box will appear.

-

Please select the rule you wish to use to build files with this extension

Lists all the RULES files that contain custom build rules that enable you to build files that have the specified file extension.

Do not show this again

Sets the Visual Studio Integrated Development Environment (IDE) to no longer display this dialog box when you add a file to a project that has a matching custom build rule.

Create New Rule File

Enables you to create new custom build rules. Click this option to open the New Rule File Dialog Box.

See Also

Reference

VC++ Project Settings, Projects and Solutions, Options Dialog Box

Matching Custom Build Rule Not Found Dialog Box

This dialog box helps you add custom build rules, which are defined in Rule Files, to your C++ projects.

When you add a file to your project, the project system scans **Rule File Search Paths** for custom build rules that match the file's extension. If you add a file to a project that does not have a matching custom build rule that specifies how the project system should build that type of file, then this dialog box will appear.

-

Yes

Enables you to create new custom build rules. Click this option to open the New Rule File Dialog Box.

No

Closes this dialog box. The build process will ignore files that do not have a build rules associated with their file extension.

Do not show this again

Sets the Visual Studio integrated development environment (IDE) to no longer display this dialog box. If you select this option, then you can set the **Show "No Build Rule" Dialog** property in the VC++ Project Settings, Projects and Solutions, Options Dialog Box to **Yes**, to undo this behavior.

See Also

Reference

Tool Build Order Dialog Box
Matching Custom Build Rule Dialog Box

Tool Build Order Dialog Box

The Build Order dialog box allows you to specify the order in which Visual C++ build tools are run.

To access this dialog box, right-click your project in Solution Explorer, and then select the **Build Order** context menu item.

Configuration

Specifies the build configuration for your project.

Platform

Specifies the target platform for your project.

Configuration Manager

Opens Configuration Manager dialog box. For more information, see Configuration Manager Dialog Box.

Build Order

Lists all build tools currently used by the Visual C++ Project System.

Up

Moves the selected build event to the previous execution slot.

Down

Moves the selected build event to the next execution slot.

Restore

Restores the default configuration for the order in which build tools are run.

See Also

Reference

Rule Files

Adding Functionality with Code Wizards

Once you have created a project, you will want to change or add to that project's functionality. Such tasks include creating new classes, adding new member functions and variables, and adding Automation methods and properties. The code wizards are designed to let you do all these things.

Note
You can now add message handlers and map messages to them and override MFC virtual functions using the Properties window.

Accessing Visual C++ Code Wizards

There are three locations where you can access Visual C++ code wizards:

- On the **Project** menu, the **Add New Item** command allows you to bring up the Add New Item dialog box, which helps you to add new files to your project. The **Add Class** command displays the Add Class dialog box, which in turn open wizards for each of the class types you can add to your project. The **Add Resource** command displays the Add Resource dialog box, from which you can create or select a resource to add to your project.

  If you highlight a class or an interface in your project in Class View, the **Project** menu also displays the following commands:

  - **Implement Interface** (from a control class only)
  - **Add Function**
  - **Add Variable**
  - **Add Connection Point** (ATL class only)
  - **Add Method** (from an interface only)
  - **Add Property** (from an interface only)
  - **Add Event** (from a control class only)
  - **Add PerfMon Object** (from a Performance Monitor Object Manager only)
  - **Add PerfMon Counter** (from a Performance Object only)
- In Solution Explorer, right-clicking any folder and clicking **Add** from the shortcut menu allows you to add new or existing files, more folders, items, classes, resources, and Web references to the project.
- From the Class View window, right-clicking the appropriate node and clicking **Add** from the shortcut menu allows you to add functions, variables, classes, properties, methods, events, interfaces, connection points, or other code to your project.

Note
Visual Studio does not provide a wizard to add an interface to a project. You can add an interface to an ATL project or t o an Adding ATL Support to Your MFC Project by adding a simple object using the ATL Simple Object Wizard. Alternate ly, open the project's .idl file and create the interface by typing:

  ```
  interface IMyInterface {
  };
  ```

  See Implementing an Interface and Adding Objects and Controls to an ATL Project for more information.

Access code wizard from	Description
Add New Item	The Add New Item code wizards add source files to your project. If necessary, additional directories are created to contain the files where the project build engine expects to find them. Code wizards available from the Add Item icon include: • Add C++ source files (.cpp, .h, .idl, .rc, .srf, .def, .rgs). • Add Web development files (.html, .asp, .css, .xml). • Add utility and resource files (.bmp, .cur, .ico, .rct, .sql, .txt). These code wizards generally do not ask you for any information but add a file to your development tree. You may rename the file in the property window.
Solution Explorer	The code wizards available from Solution Explorer depend on where your cursor focus is when you right-click an item. If the **Add** option does not appear when you right-click an item, then move your cursor up one level in the development tree and try again. The code wizards will always place the additional code in the appropriate place in the development tree, no matter where your cursor is. Code wizards available from Solution Explorer include: • Add Class (opens the **Add Class** dialog box containing the new code wizards). • Add Resource (New, Import, or Custom). • Add Web Reference.
Class View	The code wizards available from Class View depend on where your cursor focus is when you right click an item. If the **Add** option does not appear when you right click an item, then move your cursor up one level in the class tree and try again. The code wizards will always place the additional code in the appropriate place in the development tree, no matter where your cursor is. Code wizards available from Class View include: • Add Member Function. • Add Member Variable. • Add Class. • Implement Interface (from a control class only) • Add Connection Point (ATL class only) • Add Method (from an interface only) • Add Property (from an interface only) • Add Event (from a control class only) • Add PerfMon Object (from a Performance Monitor Object Manager only) • Add PerfMon Counter (from a Performance Object only) The Add Class selection opens the **Add Class** dialog box, which gives you access to all the new Add Class code wizards.

See Also

Tasks

Overriding a Virtual Function

Creating Projects with Application Wizards

Reference

Navigating the Class Structure

Concepts

Visual C++ Projects

File Types Created for Visual C++ Projects

Using Visual C++ Add New Item Templates

You can easily add items that are common to Visual C++ projects by using the **Add New Item** command. When you use the **Add New Item** command, the Add New Item dialog box appears with a list of item templates, which add the appropriate files to your project.

The following table is an alphabetical list of Visual C++ Add New Item templates.

Template	Description
ASP Page (.asp)	Adds a blank ASP page.
ASP.NET Web Service	Adds an XML Web service created with CLR features.
ATL Server Web Service	Adds an XML Web service created with ATL Server.
Bitmap File (.bmp)	Creates a Win32 bitmap file.
C++ File (.cpp)	Creates a C++ source file.
Component Class	Adds a Component Class using CLR features.
Cursor File (.cur)	Creates a Win32 cursor file.
DEF File (.def)	Creates a DLL export definition file.
Discovery File, Dynamic (.vsdisco)	Creates a dynamic discovery file, which is an XML file that contains links to other resources that describe the XML Web service. This file enables clients to find an XML Web service and to determine its functionality.
Discovery File, Static (.disco)	Creates a static discovery file, which is an XML document that contains links to other resources that describe the XML Web service, enables programmatic discovery of an XML Web service.
Frameset (.htm)	Adds an HTML file that hosts multiple HTML pages.
Header File (.h)	Creates a C++ header file.
HTML Page (.htm)	Creates a blank HTML file.
Icon File (.ico)	Creates a Win32 icon file.
Installer Class	Adds a class that inherits from the Installer Class using CLR features.
MIDL File (.idl)	Creates an Interface Definition Language file.
Registration Script (.rgs)	Creates an ATL registration script file.
Resource File (.rc)	Creates a Win32 resource file.

Resource Template File (.rc)	Creates a resource template file.
SQL Script File (.sql)	Creates an SQL script file. **Note** This template is not a Standard Edition feature.
SRF File (.srf)	Creates a server response file.
Style Sheet (.css)	Adds a cascading style sheet used for rich HTML style definitions.
Text File (.txt)	Adds a blank text file.
User Control	Adds a User Control using CLR features.
Windows Form	Adds a Windows Form using CLR features.
XML File (.xml)	Adds a blank XML file.

See Also

Concepts

Adding Functionality with Code Wizards

Adding a Class

To add a class in your Visual C++ project, display the Add Class dialog box by right-clicking the project in either Solution Explorer or Class View. From the shortcut menu, click **Add**, and then click **Add Class**.

When you add a class, you must specify a name that is different from classes already existing in MFC or ATL. If you specify a name that already exists in either library, Visual C++ displays an error message indicating that the specified name is reserved.

If your project naming convention requires you to use an existing name, then you can change the letter case, because Visual C++ is case sensitive. That is, you can name a class **cdocument**, but you cannot name a class **CDocument**.

What kind of class do you want to add?

- MFC class
- ATL class
- WMI Provider
- Generic class
- Form-based class
- Add ATL support to my MFC project

See Also

Tasks

Creating Projects with Application Wizards

Reference

Add Class Dialog Box

Concepts

Adding Functionality with Code Wizards

Add Class Dialog Box

The Add Class dialog box contains templates that allow you to:

- Open a corresponding code wizard, if one is available. For more information, see Adding Functionality with Code Wizards.

- or -

- Automatically create your new class by adding the appropriate files and source code to your project.

You can access the Add Class dialog box from the **Project** menu, Solution Explorer, or Class View.

Note
When you attempt to add a class that is not suited to your current project, you will receive an error message. Click **OK** to return to the Add Class dialog box.

Add Class Templates

There are four categories of Add Class templates: .NET, ATL, MFC, and Generic. When you select a template in the **Templates** pane, text describing your selection will appear under the **Categories** and **Templates** panes.

.NET

Template	Wizard
ASP.NET Web Service	Not available
Component Class (.NET)	Not available
Installer Class (.NET)	Not available
User Control (.NET)	Not available
Windows Form (.NET)	Not available

ATL

Template	Wizard
Add ATL Support to MFC	Not available
ATL Active Server Page Component	ATL Active Server Page Component Wizard
ATL Control	ATL Control Wizard
ATL Dialog	ATL Dialog Wizard
ATL COM+ 1.0 Component	ATL COM+ 1.0 Component Wizard
ATL OLEDB Consumer	ATL OLE DB Consumer Wizard
ATL OLEDB Provider	ATL OLE DB Provider Wizard
ATL Performance Monitor Object	ATL Performance Monitor Object Manager Wizard
ATL Property Page	ATL Property Page Wizard

ATL Simple Object	ATL Simple Object Wizard
ATL Server Web Service	ATL Server Web Service Object Wizard
WMI Event Provider	WMI Event Provider Wizard
WMI Instance Provider	WMI Instance Provider Wizard

MFC

Template	Wizard
MFC Class	MFC Class Wizard
MFC Class From ActiveX Control	Add Class From ActiveX Control Wizard
MFC Class From TypeLib	Add Class From Typelib Wizard
MFC ODBC Consumer	MFC ODBC Consumer Wizard

Generic Classes

Template	Wizard
Generic C++ Class	Generic C++ Class Wizard

See Also

Tasks

Adding a Member Function
Adding a Member Variable
Overriding a Virtual Function

Reference

Navigating the Class Structure

Concepts

Adding an MFC Message Handler

Adding a Class from an ActiveX Control

Use this wizard to create an MFC class from an interface in an available ActiveX control. You can add an MFC class to an MFC application, an MFC DLL, or an MFC ActiveX control.

Note
You do not need to create your MFC project with Automation enabled to add a class from an ActiveX control.

An ActiveX control is a reusable software component based on the Component Object Model (COM) that supports a wide variety of OLE functionality and can be customized to fit many software needs. ActiveX controls are designed for use both in ordinary ActiveX control containers and on the Internet in World Wide Web pages.

To add an MFC class from an ActiveX control

1. In either Solution Explorer or Class View, right-click the name of the project to which you want to add the ActiveX control class.
2. From the shortcut menu, click **Add**, and then click **Add Class**.
3. In the Add Class dialog box, in the Templates pane, click **MFC Class from ActiveX Control**, and then click **Open** to display the Add Class from ActiveX Control Wizard.

In the wizard, you can add more than one interface in an ActiveX control. Likewise, you can create classes from more than one ActiveX control in a single wizard session.

You can add classes from ActiveX controls registered in your system, or you can add classes from ActiveX controls located in type library files (.tlb, .olb, .dll, .ocx, or .exe) without first registering them in your system. See Registering OLE Controls for more information on registering ActiveX controls.

The wizard creates an MFC class, derived from CWnd or from COleDispatchDriver, for each interface you add from the selected ActiveX control.

See Also

Concepts

MFC ActiveX Controls

Other Resources

Introduction to COM and ATL

Add Class From ActiveX Control Wizard

Use this wizard to add an MFC class from an available ActiveX control. The wizard creates a class for each interface you add from the selected ActiveX control.

Add Class From

Specifies the location of the type library, from which the class is created.

Option	Description
Registry	The type library is registered in the system. Registered type libraries are listed in **Available ActiveX controls**.
File	The type library is not necessarily registered in the system but is contained in a file. You must provide the file location in **Location**.

Available ActiveX controls

Specifies the ActiveX controls currently registered in the system. Select an ActiveX control from this list to display its interfaces in the **Interfaces** list. See MFC ActiveX Controls: Distributing ActiveX Controls for more information about registering ActiveX controls.

If you click **File** under **Add Class From**, this box is unavailable for change.

Location

Specifies the location of the ActiveX control. If you click **File** under **Add Class From**, you can provide the location of the file containing the type library. To browse to the location of the file, click the ellipsis button.

If you click **Registry** under **Add Class From**, this box is unavailable for change.

Interfaces

Specifies the interfaces in the ActiveX control currently selected in **Available ActiveX controls** or in the type library in the file specified in **Location**.

Transfer button	Description
>	Adds the interface currently selected in the **Interfaces** list. Unavailable if no interface is selected.
>>	Adds all the interfaces in the ActiveX control currently selected in **Available ActiveX controls** or in the type library in the file specified in **Location**.
<	Removes the class currently selected in the **Generated classes** list. Unavailable if no class is currently selected in the **Generated classes** list.
<<	Removes all the classes in the **Generated classes** list. Unavailable if the **Generated classes** list is empty.

Generated classes

Specifies the class names to be generated from the interfaces added using the **>** or **>>** button. You can click this box to select a class, and then use the up or down keys to scroll through the list, viewing each class name in the **Class** box and file name in the **.h file** box that the wizard generates when you click **Finish**. You can select only one class at a time in this box.

You can remove a class by selecting it in this list and clicking **<**. You do not need to select a class in the **Generated classes** box to remove all classes; by clicking **<<**, you remove all classes in the **Generated classes** box.

Class

Specifies the name of the class selected in the **Generated classes** box that the wizard adds when you click **Finish**. You can edit the name in the **Class** box.

.h file

Sets the name of the header file for the new object's class. By default, this name is based on the name you provide in **Generated classes**. Click the ellipsis button to save the file name to the location of your choice, or to append the class declaration to an existing file. If you choose an existing file, the wizard will not save it to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class declaration should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

.cpp file

Sets the name of the implementation file for the new object's class. By default, this name is based on the name you provide in **Generated classes**. Click the ellipsis button to save the file name to the location of your choice. The file is not saved to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class implementation should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

See Also

Tasks

Adding a Class from an ActiveX Control
Automation Clients: Using Type Libraries

Adding a Generic C++ Class

A generic C++ class is a class that you define or that is derived from a class that you define. You can add a generic C++ class from Class View.

To add a generic C++ class to your project

1. From Class View, right-click the project name where you want to add the new class.
2. From the shortcut menu, click **Add**, and then click **Add Class**.
3. In the Add Class dialog box, click **Generic C++ Class** in the Templates pane. Click **Open** to display the Generic C++ Class Wizard.
4. Define your settings in the Generic C++ Class Wizard. You must provide at least a class name.
5. Click **Finish** to close the wizard and view the new generic C++ class in your project.

See Also

Tasks

Adding an MFC Class

Adding an ATL Simple Object

Concepts

Adding a Class

Generic C++ Class Wizard

This wizard adds a C++ class to your project. This generic C++ class does not inherit from ATL or MFC.

Class name

Sets the name of the new class.

.h file

Sets the name of the header file for the new object's class. By default, this name is based on the name you provide in **Class name**. Click the ellipsis button to save the file name to the location of your choice, or to append the class declaration to an existing file. If you choose an existing file, the wizard will not save it to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class declaration should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

.cpp file

Sets the name of the implementation file for the new object's class. By default, this name is based on the name you provide in **Class name**. Click the ellipsis button to save the file name to the location of your choice. The file is not saved to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class implementation should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

Base class

Sets the base class for the new class.

Access

Sets the access to the base class members for the new class. Access modifiers are keywords that specify the access other classes have to the class member functions. See Member-Access Control for more information about specifying access. The class access level is set to **public** by default.

- public
- protected
- private
- Default Specifies no access modifier.

Virtual destructor

Specifies whether the class destructor is virtual. Using virtual destructors helps ensure that the correct destructor is called when instances of derived classes are deleted.

Inline

Generates both the class constructor and the class definition as inline functions in the header file.

See Also

Tasks

Adding a Generic C++ Class

Adding a Member Function

You can add a member function to any class using Class View. Using the Add Member Function Wizard adds a declaration to the header file, and adds a stub member function body to the class's implementation file, which you can then edit.

To add a member function using the Add Member Function Wizard

1. In Class View, expand the project node to display the classes in the project.
2. Right click the class to which you want to add a member function.
3. On the shortcut menu, click **Add** then click **Add Function** to display the Add Member Function Wizard.
4. Provide the member function information in the appropriate wizard boxes. See Add Member Function Wizard for details.
5. Click **Finish** to generate the member function code.

See Also

Tasks

Adding a Member Variable
Overriding a Virtual Function

Reference

Navigating the Class Structure

Concepts

Adding Functionality with Code Wizards
Adding a Class
Adding an MFC Message Handler

Add Member Function Wizard

This wizard adds a member function declaration to the header file and a stub member function implementation to the implementation file for the selected class.

Once you have added the member function using the wizard, you can edit the code in the development environment.

Return type

Sets the return type for the member function you are adding. You can provide your own return type, or you can select from the list of available types. For information about the types, see Fundamental Types.

char	**int**	**unsigned int**
double	**long**	**unsigned long**
float	**short**	**void**
HRESULT	**unsigned char**	

Function name

Sets the name of the member function you are adding.

Parameter type

Sets the type of parameter you are adding for the member function, if the member function has parameters. You can provide your own parameter type, or you can select from the list of available types.

char	**int**	**unsigned char**
double	**long**	**unsigned int**
float	**short**	**unsigned long**

Parameter name

Sets the name of a parameter you are adding for the member function, if the member function has parameters.

Parameter list

Displays a list of parameters you have added to the member function. To add a parameter to the list, provide a type and name in the **Parameter type** and **Parameter name** boxes and click **Add**. To remove a parameter from the list, select the parameter and click **Remove**.

Access

Sets the access to the member function. Access modifiers are keywords that specify the access other classes have to the member function. See Member-Access Control for more information about specifying access. The member function access level is set to **public** by default.

- public
- protected
- private

Check whether the new member function is static or virtual, and whether it is inline or pure. If you set the member function to be pure, the **Virtual** check box is selected, and the **Inline** check box becomes unavailable. The default is a nonstatic, nonvirtual member function.

Option	Description

Static	Specifies that the function acts like a global and can be called outside of the class, even with no class instantiation. The member function has no access to non-static members. A member function specified as **Static** cannot be virtual.
Virtual	Ensures that the correct member function is called for an object, regardless of the expression used to make the member function call. A member function specified as **Virtual** cannot be static.
Pure	Indicates that no implementation is supplied for the virtual member function being declared; therefore, **Pure** can be specified only on virtual member functions. See Class-Member Declaration Syntax for more information. A class that contains at least one pure virtual member function is considered an abstract class. Classes derived from the abstract class must implement the pure virtual member function or they, too, are abstract classes.
Inline	Instructs the compiler to insert a copy of the member function body into each place the member function is called. A member function specified as **Inline** cannot be pure.

.cpp file

Sets the file location where the stub member function implementation is written. By default, it is written to the .cpp file for the class to which the member function is added. Click the ellipsis button to change the file name. The member function implementation is added to the contents of the selected file.

Comment

Provides a comment in the header file for the member function.

Function signature

Displays the member function as it appears in the code when you click **Finish**. You cannot edit the text in this box. To change the member function, change the appropriate boxes in the wizard.

See Also

Tasks

Adding a Member Function

Adding a Member Variable

You can add a member variable to a class using Class View. Member variables can be either for data exchange and data validation, or they can be generic. The data member variable wizard is specifically designed to take the relevant information and use it to insert elements in your source files at the appropriate locations. You can add a member variable from the Dialog editor in Resource View, or from Class View.

Note
When you are designing and implementing a dialog box, you might find it more efficient to use the Dialog editor to add the dialog box controls, and then to implement the controls' member variables.

To add a member variable for a dialog control in Resource View using the Add Member Variable Wizard

1. In Resource View, expand the project node and the Dialog node to display the list of the project's dialog boxes.
2. Double-click the dialog box to which you want to add the member variable to open it in the Dialog editor.
3. In the dialog box displayed in the Dialog editor, right-click the control to which you want to add the member variable.
4. On the shortcut menu, click **Add Variable** to display the Add Member Variable Wizard.

Note
A default value is already provided in **Control ID**.

5. Provide the information in the appropriate wizard boxes. See Dialog Box Controls and Variable Types for more information.
6. Click **Finish** to add the definition and implementation code to the project and close the wizard.

To add a member variable from Class View using the Add Member Variable Wizard

1. In Class View, expand the project node to display the classes in the project.
2. Right-click the class to which you want to add a variable.
3. On the shortcut menu, click **Add**, and then click **Add Variable** to display the Add Member Variable Wizard.
4. Provide the information in the appropriate wizard boxes. See Add Member Variable Wizard for details.
5. Click **Finish** to add the definition and implementation code to the project and close the wizard.

See Also

Tasks

Adding a Member Function

Reference

Navigating the Class Structure

Concepts

Adding Functionality with Code Wizards

Adding a Class

Adding an MFC Message Handler

Add Member Variable Wizard

This wizard adds a member variable declaration to the header file and, depending on the options, it can add code to the .cpp file. Once you have added the member variable using the wizard, you can edit the code in the development environment.

Access

Sets the access to the member variable. Access modifiers are keywords that specify the access other classes have to the member variable. See Member-Access Control for more information about specifying access. The member variable access level is set to **public** by default.

- public
- protected
- private

Variable type

Sets the return type for the member variable you are adding.

- If you are adding a member variable that is not a dialog box control, select from the list of available types.

  For information about the types, see Fundamental Types.

char	**short**
double	**unsigned char**
float	**unsigned int**
int	**unsigned long**
long	

- If you are adding a member variable for a dialog box control, this box is filled with the type of object returned for a control or value. If you select **Control**, then **Variable type** specifies the base class of the control you select in the **Control ID** box. If the dialog box control can contain a value, and if you select **Value**, then **Variable type** specifies the appropriate type for the value that control can contain. See Dialog Box Controls and Variable Types for more information.

  This value depends on the selection in **Control ID** and cannot be changed.

Variable name

Sets the name of the member variable you are adding. Member variables typically begin with the identifying string "m_," which is provided for you by default.

Control variable

Indicates that the member variable manages a control within a dialog box with data exchange and data validation support. See DoDataExchange for more information. This option is available only for member variables added to classes derived from CDialog. Select this box to activate the **Control ID** and **Control type** options.

Control ID

Sets the ID for the control variable you are adding. Select from the list the ID for the type of control for which you are adding the member variable. The list is active only when the **Control variable** box is selected, and it is limited to IDs for the controls already added to the dialog box. For example, for the standard **OK** button, the Control ID is **IDOK**.

Option	Description

Control	This option is set by default for the control type. It manages the control itself and not the state or contents (as you might want to do with a list box, combo box, or edit box) of the control.
Value	This option is available only for control types that can contain a value (such as an edit box) or reflect a state (such as a check box), and for which you might manage range, contents, or state. See Dialog Box Controls and Variable Types for more information.

Category

Specifies whether the variable is based on a control type or the value of the control.

Control type

Sets the type of control being added. This box is not available to change. For example, a button has the control type **BUTTON**, and a combo box has the control type **COMBOBOX**. See Dialog Box Controls and Variable Types for more information.

Max characters

Available only when **Variable type** is set to CString. Indicates the maximum number of characters that the control can hold.

Min value

Available only when the variable type is **BOOL**, **int**, **UINT**, **long**, **DWORD**, **float**, **double**, **BYTE**, **short**, COLECurrency or CTime. Indicates the lowest value acceptable for a scale or date range.

Max value

Available only when the variable type is **BOOL**, **int**, **UINT**, **long**, **DWORD**, **float**, **double**, **BYTE**, **short**, **COLECurrency** or **CTime**. Indicates the highest value acceptable for a scale or date range.

.h file

For ActiveX controls, whose member variables require a wrapper class. Sets the name of the header file to add the class declaration.

.cpp file

For ActiveX controls, whose member variables require a wrapper class. Sets the name of the implementation file to add the class definition.

Comment

Provides a comment in the header file for the member variable.

See Also

Tasks

Adding a Member Variable

Dialog Box Controls and Variable Types

You can use the Add Member Variable Wizard to add a member variable to a dialog box control created using MFC. The type of control for which you add the member variable determines the options that appear in the dialog box.

The following table describes all dialog box control types supported in MFC and the Dialog Editor, and their available types and values.

Control	Control type	Control variable type	Value variable type	Min/max values (value type only)
Animation control	SysAnimate32	CAnimateCtrl	None; control only	N/A
Button	BUTTON	CButton	None; control only	N/A
Check box	CHECK	CButton	**BOOL**	Min value/Max value
Combo box	COMBOBOX	CComboBox	CString	Max characters
Date time picker control	SysDateTimePick32	CDateTimeCtrl	CTime	Min value/max value
Edit box	EDIT	CEdit	**CString**, int, UINT, long, DWORD, float, double, BYTE, short, BOOL, **COleDateTime**, or **COleCurrency**	Min value/max value; some support max characters
Hotkey control	msctls_hotkey32	CHotKeyCtrl	None; control only	N/A
List box	LISTBOX	CListBox	**CString**	Max characters
List control	SysListView32	CListCtrl	None; control only	N/A
Month Calendar control	SysMonthCal32	CMonthCalCtrl	**CTime**	Min value/max value
Progress control	msctls_progress32	CProgressCtrl	None; control only	N/A
Rich Edit 2 control	RichEdit20A	CRichEditCtrl	**CString**	Max characters
Rich Edit control	RICHEDIT	**CRichEditCtrl**	**CString**	Max characters
Scroll bar (vertical or horizontal	SCROLLBAR	CScrollBar	**int**	Min value/max value
Slider control	msctls_trackbar32	CSliderCtrl	**int**	Min value/max value
Spin control	msctls_updown32	CSpinButtonCtrl	None; control only	N/A

Tab control	SysTabControl32	CTabCtrl	None; control only	N/A
Tree control	SysTreeView32	CTreeCtrl	None; control only	N/A

See Also

Tasks

Adding a Member Variable

Adding a Method

You can use the Add Method Wizard to add a method to an interface in your project. If the project contains a class associated with the interface, the wizard modifies the class, too.

To add a method to your object

1. In Class View, expand the project node to display the interface to which you want to add the method.

Note
You can also add methods to dispinterfaces, which, unless the project is attributed, are located under the library node.

2. Right-click the name of the interface.
3. On the shortcut menu, click **Add**, and then click **Add Method**.
4. In the Add Method Wizard, provide the information to create the method.
5. Specify any interface definition language settings for this method in the IDL attributes page of the wizard.
6. Click **Finish** to add the method.

See Also

Concepts

Creating a COM Interface
Editing a COM Interface

Add Method Wizard

Use this wizard to add a method to an interface. Depending on the project type or interface type to which you are adding a method, the wizard displays different options.

Names

Return type

The data type returned by the method. **HRESULT** is recommended for all interface types, because it provides a standard way to return errors.

Interface type	Description
Dual interface	**HRESULT**. Unchangeable.
Custom interface	**HRESULT**. Unchangeable.
Local custom interface	Provide your own return type or select from the list.
Dispinterface	Provide your own return type or select from the list.
MFC ActiveX control dispinterface	If you implement a stock method, the return type is set to the appropriate value and is unchangeable. If you select a method from the **Method name** list and click **Custom** under **Select method type**, select a return type from the list.

Method name

Sets the name for the method.

Interface type	Description
ATL dual interface, custom interface, and local custom interface	Provide your own method name.
MFC dispinterface	Provide your own method name or select a suggested method name from the list. If you select a name from the list, the appropriate value appears in the **Return type** box, and it is unchangeable.
MFC ActiveX control dispinterface	Provide your own or select either of the stock methods DoClick and Refresh. See MFC ActiveX Controls: Adding Stock Methods for more information.

Method type

Available only for MFC ActiveX controls. If you provide a method name in the **Method name** box, rather than selecting a method from the list, this box is unavailable.

If you select one of the methods in the **Method name** list, select either the stock implementation or a custom implementation.

Method type	Description
Stock	The default. Inserts the stock implementation of the method you select in the **Method name** list. **Return type** is unchangeable if you select **Stock**.
Custom	Inserts a stub implementation of the method selected in the **Method name** list. For custom method types, you can provide your own return type, or you can select one from the **Return type** list.

Internal name

Available for only custom methods added to an MFC dispinterface. Sets the name used in the dispatch map, the header (.h) file, and the implementation (.cpp) file. By default, this name is the same as **Method name**. You can change the method name if you are working with an MFC dispinterface or if you are adding a custom method to an MFC ActiveX control dispinterface.

Interface type	Description
ATL dual interface, custom interface, and local custom interface	Not available
MFC dispinterface	Set to the method name by default. You can edit the internal name.
MFC ActiveX control dispinterface	You can set the internal name only for custom methods. Stock methods do not use an internal name.

Parameter attributes

Sets any additional attributes for parameter specified in **Parameter name**.

Parameter attribute	Description	Allowed combinations
In	Indicates that the parameter is passed from the calling procedure to the called procedure.	**in** only **in** and **out**
Out	Indicates that the pointer parameter is returned from the called procedure to the calling procedure (from the server to the client).	**out** only **in** and **out** **out** and **retval**
Retval	Indicates that the parameter receives the return value of the member.	**retval** and out

Parameter type

Sets the data type of the parameter. Select the type from the list.

Parameter name

Sets the name of a parameter to pass through your method. After typing the name, you must click **Add** to add it to the list of parameters that will pass through your method. If you do not provide a parameter name, the wizard ignores any parameter attributes (ATL only) or **Parameter type** selections.

Once you click **Add**, the parameter name appears in **Parameter list**.

Note If you supply a parameter name and then click **Finish** before you click **Add**, the parameter is not added to the method. You must find the method and insert the parameter manually.

Add

Adds the parameter you specify in **Parameter name**, and its type and parameter attributes, to **Parameter list**. You must click **Add** to add a parameter to the list.

Remove

Removes the parameter you select in **Parameter list** from the list.

Parameter list

Displays all parameters and their modifiers and types currently added for the method. As you add parameters, the wizard updates **Parameter list** to display each parameter, with its modifier and type.

See Also

Tasks

Adding a Method

Reference

IDL Attributes, Add Method Wizard

Use this page of the Add Method Wizard to specify any interface definition language (IDL) settings for the method.

id

Sets the numerical ID that identifies the method. See id in the *MIDL Reference*.

This box is unavailable for custom interfaces and is not available for MFC dispinterfaces.

call_as

Specifies the name of a remote method to which this local method can be mapped. See call_as in the *MIDL Reference*.

Not available for MFC dispinterfaces.

helpcontext

Specifies a context ID that lets the user view information about this method in the Help file. See helpcontext in the *MIDL Reference*.

Not available for MFC dispinterfaces.

helpstring

Specifies a character string that is used to describe the element to which it applies. By default, it is set to "method *Method name*." See helpstring in the *MIDL Reference*.

Not available for MFC dispinterfaces.

Additional attributes

Not available for MFC dispinterfaces.

Attribute	Description
hidden	Indicates that the method exists but should not be displayed in a user-oriented browser. See hidden in the *MIDL Reference*.
source	Indicates that a member of the method is a source of events. See source in the *MIDL Reference*.
local	Specifies to the MIDL compiler that the method is not remote. See local in the *MIDL Reference*.
restricted	Specifies that the method cannot be called arbitrarily. See restricted in the *MIDL Reference*.
vararg	Specifies that the method takes a variable number of arguments. To accomplish this, the last argument must be a safe array of **VARIANT** type that contains all the remaining arguments. See vararg in the *MIDL Reference*.

See Also

Tasks

Adding a Method

Reference

Add Method Wizard

Adding a Performance Monitor Counter

Adding a performance monitor counter is the third step to adding performance monitoring support to your system or server project. Use the ATL Performance Monitor Counter Wizard to add a performance monitor counter to an ATL DLL project to which you have already added a performance monitor object manager and a performance object.

Note
You can either add performance monitor support into an ATL project created using the ATL Project Wizard, or you can add performance support to an MFC application by adding ATL support to your MFC application.

Counters are the means by which the performance object gathers data from the system or server. Each counter is responsible for a specific area of system functionality, and it is identified by a unique name and type. ATL performance counters provide data in two forms:

- Raw counter values; for example, **Raw Counter**.
- A calculation of raw data and another component, such as a value averaged over time; for example, **Timer**.

See the following topics for more information:

- ULONG Counter Types
- ULONGLONG Counter Types
- Performance Objects and Counters in the *Platform SDK* for more information.

Just as you can add multiple performance objects to a performance monitor object manager, you can add multiple counters to a performance object. Some of these counters provide information used by other counters.

Performance Monitor Object Manager with Performance Objects and Counters

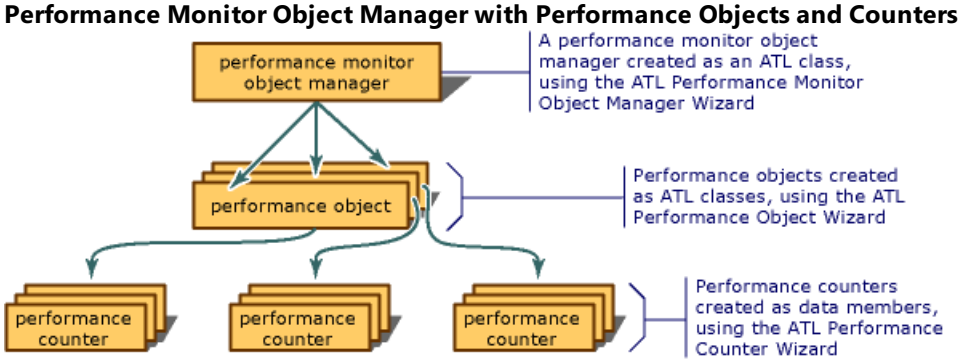

See Performance Monitoring for more information.

To add an ATL performance monitor counter to your project

1. In Class View, expand the project to display the project's classes.
2. Right-click the performance object's class name to which you want to add the performance counter.
3. On the shortcut menu, click **Add**. Click **Add Perfmon Counter** to display the Add Perfmon Counter Wizard.

See Also

Reference

Performance Data Collection

ATL Performance Monitor Counter Wizard

This wizard inserts a performance monitor counter as a variable into an object containing a performance monitor object manager and a performance object. Nonattributed projects must contain a counter map (which is added automatically by the ATL Performance Object Wizard), specifying the name of the variable, its type, and its human-readable name and help string. Additionally, you can specify:

- The level of user you expect to collect the data.
- The exponential value, indicating the top level of the counter scale.

Variable type

Indicate the variable type for the counter. The type you select determines which **Counter type** you can create.

Variable type	Description
ULONG	A 32-bit unsigned integer.
ULONGLONG	A 64-bit unsigned integer.

Variable name

Sets the name of the member variable you are adding. In the MMC Console application provided with Windows 2000, this name appears in the counter list in the **Add Counters** dialog box.

Counter type

Sets the type of calculations the counter performs. Select one of the types from the list. The option you select for **Variable type** determines which counter types are displayed in the list. For a description of each type, see:

- ULONG Counter Types
- ULONGLONG Counter Types

For non-attributed projects, the type you select is implemented as the *countertype* parameter in the DEFINE_COUNTER macro, in the CPerfObject derived class in your project. In an attributed project, *countertype* is passed as a parameter in the perf_counter attribute.

Note
The counter types and data structures are defined in WINPERF.H.

Each counter type is implemented in the counter attribute for attributed projects, or in the counter macro map for nonattributed projects.

Name string

Sets the human-readable name for the counter.

Help string

Sets the help string for the counter. This string length has no upper limit.

In the MMC console application provided with Windows 2000, users can view a counter's help string by opening the **Add Counters** dialog box, highlighting the appropriate counter, and clicking the **Explain** button.

Detail level

Indicates the level of information provided to the user. Your selection is specified in a DWORD and sent to the performance monitor, as is the detail level you specify in the ATL Performance Object Wizard.

- novice
- advanced
- expert

- wizard

Default scale

Indicates the power of 10 by which to multiply the counter value before displaying the counter. For example, if you set **Default scale** to 2, the counter value is multiplied by 100. If you set **Default scale** to –3, the counter value is divided by 1,000.

Default counter

Sets this counter as the default counter for the performance object. In the MMC Console application provided with Windows 2000 and above, this counter name appears as the default selection in the counter list in the **Add Counters** dialog box.

See Also

Tasks

Adding a Performance Monitor Counter

Concepts

Performance Monitoring

ULONG Counter Types, Performance Monitor Counter Wizard

You can add one of the following counter types if you indicate **ULONG** under **Variable type** in the ATL Performance Monitor Counter Wizard.

Counter type	Description
Average Base	Implements PERF_AVERAGE_BASE. Used as the base data in the computation of time or count averages. See **PERF_AVERAGE_TIMER** and **PERF_AVERAGE_BULK**.
Average Timer	Implements PERF_AVERAGE_TIMER. A timer that usually gives time per operation when divided by the number of operations.
Delta	Implements PERF_COUNTER_DELTA. The difference between two counters. The data size is limited to 4 bytes.
Queue Length	Implements PERF_COUNTER_QUEUELEN_TYPE. Count per time interval. Typically used to track number of items queued or waiting.
Rate Counter	Implements PERF_COUNTER_COUNTER. The default. The rate of counts.
Raw Counter	Implements PERF_COUNTER_RAWCOUNT. Instantaneous counter value.
Raw Counter (Hex)	Implements PERF_COUNTER_RAWCOUNT_HEX. Instantaneous counter value intended to be displayed as a hexadecimal number.
Raw Fraction	Implements PERF_RAW_FRACTION. Instantaneous value, to be divided by the base data. See **PERF_RAW_BASE**. A 32-bit value.
Raw Fraction Base	Implements PERF_RAW_BASE. Used as a base data for **PERF_RAW_FRACTION**. A 32-bit value. The raw performance data value holds the denominator of the fraction value. Check that this value is greater than zero before dividing.
Sample Base	Implements PERF_SAMPLE_BASE. Used as a base data for **PERF_SAMPLE_COUNTER** and **PERF_SAMPLE_FRACTION**. Check that this value is greater than zero before dividing.
Sample Counter	Implements PERF_SAMPLE_COUNTER. A count that is either 1 or 0 on each sampling interrupt.
Sample Percentage	Implements PERF_SAMPLE_FRACTION. A count that is either 1 or 0 on each sampling interrupt. For display in terms of a percentage.

See Also

Reference

ULONGLONG Counter Types, Performance Monitor Counter Wizard

ULONGLONG Counter Types, Performance Monitor Counter Wizard

You can add one of the following counter types if you indicate **ULONGLONG** under **Variable type** in the ATL Performance Monitor Counter Wizard.

Counter type	Description
Elapsed Time	Implements PERF_ELAPSED_TIME. The data is the start time of the item being measured. For display, subtract the start time from the snapshot time to yield the elapsed time. The **PerfTime** member of the PERF_OBJECT_TYPE structure contains the sample time. Use the **PerfFreq** member of the **PERF_OBJECT_TYPE** structure to convert the time into seconds.
Large Average	Implements PERF_AVERAGE_BULK. A count that usually gives the bytes per operation when divided by the number of operations.
Large Delta	Implements PERF_COUNTER_LARGE_DELTA. The difference between two counters. The data size can be up to 8 bytes.
Large Queue Length	Implements PERF_COUNTER_LARGE_QUEUELEN_TYPE. Count per time interval. Typically used to track number of items queued or waiting.
Large Queue Length (100 Ns)	Implements PERF_COUNTER_100NS_QUEUELEN_TYPE. Count per time interval using a 100-nanosecond time base. Typically used to track number of items queued or waiting.
Large Queue Length (Object Time)	Implements PERF_COUNTER_OBJ_TIME_QUEUELEN_TYPE. Count per time interval using an object-specific time base. Typically used to track number of items queued or waiting.
Large Rate	Implements PERF_COUNTER_BULK_COUNT. Used to count byte transmission rates.
Large Raw Counter	Implements PERF_COUNTER_LARGE_RAWCOUNT. Instantaneous counter value.
Large Raw Counter (Hex)	Implements PERF_COUNTER_LARGE_RAWCOUNT_HEX. Instantaneous counter value intended to be displayed as a hexadecimal number.
Large Raw Fraction	Implements PERF_LARGE_RAW_FRACTION. Instantaneous value, to be divided by the base data. A 64-bit value. See **PERF_LARGE_RAW_BASE**.
Large Raw Fraction Base	Implements PERF_LARGE_RAW_BASE. Used as a base data for **PERF_LARGE_RAW_FRACTION**. A 64-bit value. The raw performance data value holds the denominator of the fraction value. Check that this value is greater than zero before dividing.
Multi-Item Timer	Implements PERF_COUNTER_MULTI_ T IMER. Timer sampling of multiple but similar items. Result is an average sampling among the items. The number of items is in the base data.
Multi-Item Timer Inverse	Implements PERF_COUNTER_MULTI_TIMER_INV. The inverse of the timer for multiple but similar items. Used when the objects are not in use.

Multi-Item Timer (100 Ns)	Implements PERF_100NSEC_MULTI_TIMER. Timer sampling of multiple but similar items. Result is an average sampling among the items. The number of items is in the base data.
Multi-Item Timer Inverse (100 Ns)	Implements PERF_100NSEC_MULTI_TIMER_INV. The inverse of the timer for multiple but similar items. Used when the objects are not in use.
Multi-Item Timer Base	Implements PERF_COUNTER_MULTI_BASE. Used as the base data for the **MULTI** counters. It defines the number of similar items sampled. See **PERF_COUNTER_MULTI_TIMER**, **PERF_COUNTER_MULTI_TIMER_INV**, **PERF_100NSEC_MULTI_TIMER**, and **PERF_100NSEC_MULTI_TIMER_INV**.
Precision Timer	Implements PERF_PRECISION_SYSTEM_TIMER. The timer used has the same frequency as the system performance timer. When you use this type of timer, the definition of the **PERF_PRECISION_TIMESTAMP** counter must immediately follow the definition of the **PERF_PRECISION_SYSTEM_TIMER** in the Object header.
Precision Timer (100 Ns)	Implements PERF_PRECISION_100NS_TIMER. The timer used has the same frequency as the 100-nanosecond timer. When you use this type of timer, the definition of the **PERF_PRECISION_TIMESTAMP** counter must immediately follow the definition of the **PERF_PRECISION_100NS_TIMER** in the Object header.
Precision Timer (Object Time)	Implements PERF_PRECISION_OBJECT_TIMER. The timer used is of the frequency specified in the Object header's **PerfFreq** field (**PerfTime** is ignored). When you use this type of timer, the definition of the **PERF_PRECISION_TIMESTAMP** counter must immediately follow the definition of the **PERF_PRECISION_OBJECT_TIMER** in the Object header.
Precision Timer Timestamp	Implements PERF_PRECISION_TIMESTAMP. Use in the computation of the timer specified in the previous description block.
Timer	Implements PERF_COUNTER_TIMER. The most common timer.
Timer Inverse	Implements PERF_COUNTER_TIMER_INV. The inverse of the timer. Used when the object is not in use.
Timer (100 Ns)	Implements PERF_100NSEC_TIMER. Timer for when the object is in use.
Timer Inverse (100 Ns)	Implements PERF_100NSEC_TIMER_INV. The inverse of the timer. Used when the object is not in use.
Timer (Object Time)	Implements PERF_OBJ_TIME_TIMER. 64-bit timer in object specific units. Display delta divided by delta time as returned in the object type header structure.

See Also

Reference

ULONG Counter Types, Performance Monitor Counter Wizard

Adding a Performance Object

Creating a performance object is the second step to adding performance monitoring support to your system or server project. Use the ATL Performance Object Wizard to add a performance object to an ATL DLL project to which you have already added a performance monitor object manager. You can add the performance object using either attributed or nonattributed code

Note
You can either add performance monitor support into an ATL project created using the ATL Project Wizard, or you can add p erformance support to an MFC application by adding ATL support to your MFC application.

Performance monitor object manager with performance objects and counters

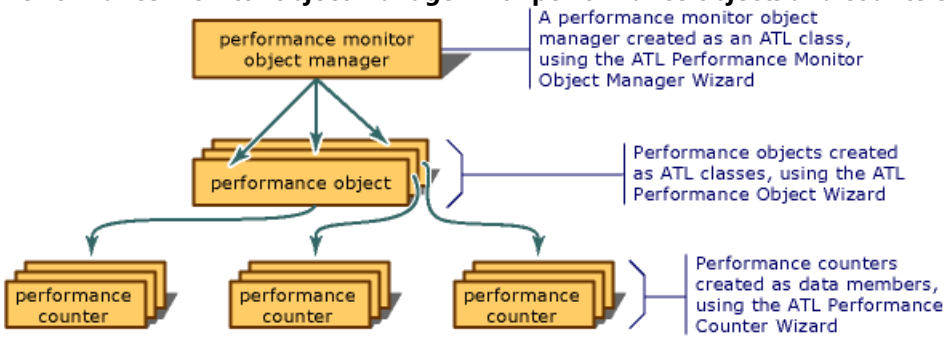

See Performance Monitoring for more information.

Performance monitoring allows your applications and components to publish, capture, and analyze the performance data that applications, services, and drivers provide. You can use this information to determine system bottlenecks and fine-tune system and application performance.

A machine's performance objects can include:

- Physical components (such as processors, disks, and memory).
- System objects (such as processes and threads).
- Server transactions (such as ATL ISAPI server transactions).

Each object is related to a functional element in the system and has a set of counters assigned to it.

Note
After you add a performance object to your performance monitor object manager, you can add the counters either manually or with the ATL Performance Monitor Counter Wizard.

Performance data is grouped by the performance object to which is it related. See Performance Objects and Counters in the *Platform SDK* for a discussion of performance objects.

To add an ATL performance object to your project

1. In Class View, expand the project to display the project's classes.
2. Right-click the performance monitor object manager's class name to which you want to add the performance object.
3. From the shortcut menu, click **Add** and click **Add PerfMon Object** to display the ATL Performance Object Wizard.

See Also

Reference

Performance Objects and Counters

ATL Performance Object Wizard

This wizard inserts a performance object into a project containing performance monitor object. Using this wizard, you can specify the name of the class and its human-readable name and help string.

Class name

Sets the name of the performance object class that you are adding.

Name string

Sets the name of the performance object.

Help string

Sets a help string associated with the performance object.

Detail level

Indicates the level of information provided to the user. Your selection is specified in a DWORD and sent to the performance monitor, as is the detail level you specify in the ATL Performance Monitor Counter Wizard.

- novice
- advanced
- expert
- wizard

No instances

Indicates that only one object of the specified type exists per machine, and it requires no instances. For example, a memory usage object would require no instances.

See Also

Tasks

Adding a Performance Object

Concepts

Performance Monitoring

Adding a Property

You can use the Add Property Wizard to add a method to an interface in your project.

To add a property to your object

1. In Class View, right-click the name of the interface to which you want to add the property.

Note
You can also add properties to dispinterfaces, which, unless the project is attributed, are nested within the library node.

2. From the shortcut menu, click **Add**, and then click **Add Property**.
3. In the Add Property Wizard, provide the information to create the property.
4. Specify any interface definition language (IDL) settings for the property in the IDL Attributes page of the wizard.
5. Click **Finish** to add the property.

The **Get** and **Put** methods of the property are displayed as two icons in Class View, under the interface where it is defined. You can double-click either icon to view the property declaration in the .idl file.

- For ATL interfaces, the **Get** and **Put** functions are added to the .cpp file, and references to these functions are added to the .h file.
- For MFC dispinterfaces, if you select **Member variable** as the implementation type, a method and a variable are added to the class that implements it. If you select **Get/Set methods** as the implementation type, two methods are added to the class that implements it.

See Also

Reference

The Component Object Model
Interface Pointers and Interfaces

Concepts

Creating a COM Interface
Editing a COM Interface

Names, Add Property Wizard

Use this wizard to add a property to an interface.

Property type

Sets the type of property you are adding. For MFC dispinterfaces, provide your own type or select from the predefined list. If you provide a stock implementation of a property, **Property type** is set to the stock type and is unavailable for change.

Property name

Sets the name of the property. For MFC dispinterfaces associated with ActiveX controls, you can supply your own name or you can select a stock property name from the predefined list. If you provide your own property name, the **Stock** implementation type is unavailable. See Stock Properties for a description of the properties in the list.

Interface type	Description
ATL dual interface, custom interface, and local custom interface	Provide a property name.
MFC dispinterface, MFC ActiveX control dispinterface	Provide a property name or select a stock property from the list. If you select a property from the list, the appropriate value appears in the **Property type** box. You can change this type, depending on your selection under **Implementation type**.

Return type

ATL interfaces only. Sets the return type for the property. For dual interfaces, **HRESULT** is always the return type, and this box is unavailable. For custom interfaces, you can select a return type from the list. **HRESULT** is still recommended, because it provides a standard way to return errors.

Variable name

MFC dispinterfaces only. Available only if you specify **Member variable** under **Implementation type**. Sets the name of the member variable with which the property is associated. By default, the variable name is set to m_*PropertyName*. You can edit this name.

Notification function

MFC dispinterfaces only. Available only if you specify **Member variable** under **Implementation type**. Sets the name of the notification function called if the property changes. By default, the name of the notification function is set to On*PropertyName*Changed. You can edit this name.

Get function

For MFC dispinterfaces. Available only if you specify **Get/Set methods** under **Implementation type**. Sets the name of the function to get the property. By default, the name of the Get function is set to Get*PropertyName*. You can edit this name. If you delete the name, the function GetNotSupported is inserted into the interface dispatch map. The Get*PropertyName* function specifies that the property as readable.

Set function

MFC dispinterfaces only. Available only if you specify **Get/Set methods** under **Implementation type**. Sets the name of the function to set the property. By default, the name of the Set function is set to Set*PropertyName*. You can edit this name. If you delete the name, the function SetNotSupported is inserted into the interface dispatch map. The Set*PropertyName* function specifies that the property is writable.

Implementation type

MFC dispinterfaces only. Specifies how to implement the property you are adding.

Implementation type	Description

Stock	Specifies a stock implementation for the property selected in **Property name**. The default. See Stock Properties for more information. If you specify **Stock**, then **Property type**, **Parameter type**, and **Parameter name** are dimmed.
Member variable	Specifies the property is added as a member variable. You can add custom properties or most stock properties as member variables. You cannot specify **Member variable** for **Caption**, **hWnd**, and **Text** properties. Provides default names under **Variable name** and **Notification function**. You can edit this name.
Get/Set methods	Specifies the property is added as Get*PropertyName* and Set*PropertyName* functions, by default. These names appear under **Get function** and **Set function**. You can change the default **Property type**, which passes a value for the Get function. You can specify parameters for the **Get** and **Set** functions.

Get function

For ATL interfaces. Sets the property as readable; that is, it creates the **Get** method for retrieving this property from the object. You must select **Get**, **Put**, or both.

Put function

ATL interfaces only. Sets the property writable; that is, it creates the **Put** method for setting, or "putting," this property of the object. You must select **Get**, **Put**, or both. If you select this option, you can choose from the following two ways to implement the method:

Option	Description
PropPut	The PropPut function returns a copy of the object. This is the default and the most common way to make the property writable.
PropPutRef	The PropPutRef function returns a reference to the object, rather than returning the copy of the object itself. Consider using this option for objects, such as large structs or arrays, that may have initialization overhead.

Parameter attributes

ATL interfaces only. Sets whether the parameter specified by **Parameter name** is **in**, **out**, both, or none.

Option	Description
in	Indicates that the parameter is passed from the calling procedure to the called procedure.
out	Indicates that the pointer parameter is returned from the called procedure to the calling procedure (from the server to the client).

Parameter type

Sets the data type of the parameter. Select the type from the list.

Parameter name

Sets the name of a parameter you are adding for the property, if the property has parameters. Once you click **Add**, the parameter name appears in **Parameter list**.

Parameter list

Displays the list of attributes to be added to the property. Each item in the list consists of the parameter name, parameter type, and attributes. Use **Add** and **Remove** to update the list.

Add

Adds the parameter you specify in **Parameter name** and **Parameter type** to the **Parameter list**. You must click **Add** to add a parameter to the list.

Remove

Removes the parameter you select in **Parameter list**.

Default property

MFC dispinterface only. Sets this property as the default for the interface. An interface can have only one default property; once you specify the default property, for any other properties you add to the interface, this box is unavailable.

See Also

Tasks

Adding a Property

Implementing an Interface

Reference

IDL Attributes, Add Property Wizard

IDL Attributes, Add Property Wizard

Use this page of the Add Property Wizard to specify any interface definition language (IDL) settings for the property.

id

Sets the numerical ID that identifies the property. This option is not available for properties of custom interfaces. See id in the *MIDL Reference*.

helpcontext

Specifies a context ID that lets the user view information about this property in the Help file. See helpcontext in the *MIDL Reference*.

helpstring

Specifies a character string that is used to describe the element to which it applies. By default, it is set to "property *Property name*." See helpstring in the *MIDL Reference*.

Other options

Not all options are available to all property types.

Option	Description
bindable	Indicates that the property supports data binding. See bindable in the *MIDL Reference*. For the stock implementation of the property, this option is set by default and is unchangeable.
defaultbind	Indicates that this the single, bindable property best represents the object. See defaultbind in the *MIDL Reference*.
displaybind	Indicates that this property should be displayed to the user as bindable. See displaybind in the *MIDL Reference*.
immediatebind	Indicates that the database will be notified immediately of all changes to this property of a data-bound object. See immediatebind in the *MIDL Reference*.
defaultcollelem	Indicates that the property is an accessor function for an element of the default collection. See defaultcollelem in the *MIDL Reference*.
nonbrowsable	Tags an interface or dispinterface member that should not be displayed in a properties browser. See nonbrowsable in the *MIDL Reference*.
requestedit	Indicates that the property supports the **OnRequestEdit** notification See requestedit in the *MIDL Reference*. For the stock implementation of the property, this option is set by default and is unchangeable.
source	Indicates that a member of the property is a source of events. See source in the *MIDL Reference*.
hidden	Indicates that the property exists but should not be displayed in a user-oriented browser. See hidden in the *MIDL Reference*.
restricted	Specifies that the property cannot be called arbitrarily. See restricted in the *MIDL Reference*.
local	Specifies to the MIDL compiler that the property is not remote. See local in the *MIDL Reference*.

See Also

Tasks

Adding a Property
Implementing an Interface

Reference

Names, Add Property Wizard

Stock Properties

Stock Properties

If you are adding a property to an MFC dispinterface using the Add Property Wizard, you can choose a stock property from the **Property name** list in the Names page of the wizard. These properties are as follows:

Property name	Description
Appearance	Returns or sets a value that determines the appearance of the control. The control's **Appearance** property can include or omit three-dimensional display effects. This is an ambient read/write property.
BackColor	Returns or sets the control's ambient **BackColor** property to either a palette (RGB) color or a predefined system color. By default, its value corresponds to the foreground color of the control's container. This is an ambient read/write property.
BorderStyle	Returns or sets the border style for a control. This is a read/write property.
Caption	Returns or sets the control's **Caption** property. The caption is the title of the window. **Caption** has no **Member variable** implementation type.
Enabled	Returns or sets the control's **Enabled** property. An enabled control can respond to user-generated events.
Font	Returns or sets the control's ambient font. Null if the control has no font.
ForeColor	Returns or sets the control's ambient **ForeColor** property.
hWnd	Returns or sets the control's **hWnd** property. **hWnd** has no **Member variable** implementation type.
ReadyState	Returns or sets the control's **ReadyState** property. A control can be uninitialized, initialized, loading, interactive, or complete. See READYSTATE in the *Internet SDK* for more information.
Text	Returns or sets the text contained in a control. **Text** has no **Member variable** implementation type.

See Also

Tasks

Adding a Property

Reference

IDL Attributes, Add Property Wizard

Adding an ATL Active Server Page Component

To add an Active Template Library (ATL) object to your project, your project must have been created as an ATL COM application or as an MFC application that contains ATL support. You can use the ATL Project Wizard to create an ATL application, you can select **Add ATL Support to MFC** from the Add Class Dialog Box dialog box, or you can add an ATL object to your MFC application to implement ATL support for an MFC application.

Active Server Pages components are part of the Internet Information Services architecture, which provides the following advanced Web development features:

- You can embed ASP components into your HTML pages to create dynamic, browser-independent content.
- You can use ASP pages to provide standards-based database connectivity.
- You can use the ASP error-handling capabilities for your Web-based applications.

To add an ATL Active Server Pages component to your project

1. In either Solution Explorer or Class View, right-click the name of the project to which you want to add the ATL Active Server Pages component.
2. From the shortcut menu, click **Add**, and then click **Add Class**.
3. In the Add Class dialog box, in the Templates pane, click **ATL Active Server Page Component**, and then click **Open** to display the ATL Active Server Page Component Wizard.

See Also

Tasks

Adding a New Interface in an ATL Project
Adding Connection Points to an Object
Adding a Method
Adding an MFC Class
Adding a Generic C++ Class

Concepts

Adding a Class

ATL Active Server Page Component Wizard

This wizard inserts into the project an Active Server Pages (ASP) component. The Microsoft Internet Information Services (IIS) uses ASP components as part of its enhanced Web page development architecture.

Using this wizard, you can specify the component's threading model and its aggregation support. You can also indicate support for the error information interface, connection points, and free-threaded marshaling.

Names

Specify the names for the object, interface, and classes to be added to your project. With the exception of **Short name**, all other boxes can be edited independently of the others. If you change the text for **Short name**, the change is reflected in the names of all other boxes in this page.

If you change the **Coclass** name in the COM section, the change is reflected in the **Type** and **ProgID** boxes, but the **Interface** name does not change. This naming behavior is designed to make all the names easily identifiable for you as you develop your control.

C++

Provides information for the C++ class created for the object.

Short name

Sets the root name for the object. The name you provide determines the **Class** and **Coclass** names, the **.cpp file** and **.h file** names, the **Interface** name, the **Type** names, and the **ProgID**, unless you change those fields individually.

.h file

Sets the name of the header file for the new object's class. By default, this name is based on the name you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice, or to append the class declaration to an existing file. If you choose an existing file, the wizard will not save it to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class declaration should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

Class

Sets the name of the class to be created. This name is based on the name you provide in **Short name**, preceded by 'C', the typical prefix for a class name.

.cpp file

Sets the name of the implementation file for the new object's class. By default, this name is based on the name you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice. The file is not saved to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class implementation should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

Attributed

Indicates whether the object uses attributes. If you are adding an object to an attributed ATL project, this option is selected and not available to change. That is, you can add only attributed objects to a project created with attribute support.

If you select this option for an ATL project that does not have attribute support, the wizard prompts you to specify whether you want to add attribute support to the project.

For nonattributed projects, any objects you add following setting this option are designated as attributed by default (the check box is selected). You can clear this box to add an object that does not use attributes.

See Application Settings, ATL Project Wizard and Basic Mechanics of Attributes for more information.

COM

Provides information about the COM functionality for the object.

Coclass

Sets the name of the component class that contains a list of interfaces supported by the object. If your project or this object uses attributes, you cannot change this option because ATL does not include the **coclass** attribute.

Type

Sets the object description that will appear in the registry for the coclass.

Interface

Sets the interface you create for your object. This interface contains your custom methods.

ProgID

Sets the name that containers can use instead of the CLSID of the object.

See Also

Tasks

Adding an ATL Active Server Page Component

Options, ATL Active Server Page Component Wizard

Use this page of the ATL Active Server Page Component Wizard to design for increased efficiency and error support for the object.

For more information on ATL projects and ATL COM classes, see ATL Reference.

Threading model

Indicates the method for managing threads. By default, the project uses **Apartment** threading.

See Specifying the Project's Threading Model for more information.

Option	Description
Single	Specifies that the object uses the single threading model. In the single threading model, an object always runs in the primary COM thread. See Single-Threaded Apartments and InprocServer32 for more information.
Apartment	Specifies that the object uses apartment threading. Equivalent to single thread apartment. Each object of an apartment-threaded component is assigned an apartment for its thread, for the life of the object; however, multiple threads can be used for multiple objects. Each apartment is tied to a specific thread and has a Windows message pump (default). See Single-Threaded Apartments for more information.
Both	Specifies that the object can use either apartment or free threading, depending from which kind of a thread it is created.
Free	Specifies that the object uses free threading. Free threading is equivalent to a multithread apartment model. See Multithreaded Apartments for more information.
Neutral (Widows 2000 only)	Specifies that the object follows the guidelines for multithreaded apartments, but it can execute on any kind of thread.

Aggregation

Indicates whether the object uses aggregation. The aggregate object chooses which interfaces to expose to clients, and the interfaces are exposed as if the aggregate object implemented them. Clients of the aggregate object communicate only with the aggregate object.

Option	Description
Yes	Specifies that the object can be aggregated. The default.
No	Specifies that the object is not aggregated.
Only	Specifies that the object must be aggregated.

Support

(Element description to be added)

Option	Description
ISupportErrorInfo	Creates support for the ISupportErrorInfo interface so the object can return error information to the client.
Connection points	Enables connection points for your object by making your object's class derive from IConnectionPointContainerImpl.

Free-threaded marshaler	Creates a free-threaded marshaler object to marshal interface pointers efficiently between threads in the same process. Available to object specifying either **Both** or **Free** as the threading model.

See Also

Tasks

Adding an ATL Active Server Page Component

Reference

ATL Active Server Page Component Wizard

ASP, ATL Active Server Page Component Wizard

Use this page of the ATL Active Server Page Component Wizard to specify optional settings for handling information and state related to your ASP component.

Optional methods

Adds the optional ASP methods, **OnStartPage** and **OnEndPage**, to your object. This option must be selected to set any Active Server Pages intrinsic objects. By default, it is selected.

- **OnStartPage/OnEndPage** OnStartPage is called the first time the script tries to access the object. **OnEndPage** is called when the object is finished processing the script.

Intrinsic object

You must select the **OnStartPage/OnEndPage** option to set any ASP intrinsic objects.

Option	Description
Request	Provides access to the Active Server Pages intrinsic **Request** object. The Request object is used to pass an HTTP request.
Response	Provides access to the Active Server Pages intrinsic **Response** object. In response to a request, the Response object sends information to the browser to display to the user.
Session	Provides access to the Active Server Pages intrinsic **Session** object. The **Session** object maintains information about the current user session, including storing and retrieving state information.
Application	Provides access to the Active Server Pages intrinsic **Application** object. The **Application** object manages state that is shared across multiple ASP objects.
Server	Provides access to the Active Server Pages intrinsic **Server** object. The **Server** object allows you to create other ASP objects.

See Also

Tasks

Adding an ATL Active Server Page Component

Reference

ATL Active Server Page Component Wizard

Adding an ATL COM+ 1.0 Component

This wizard adds an object to your project that supports COM+ 1.0 services, including transactions.

To add an ATL COM+ 1.0 component to your project

1. In either Solution Explorer or Class View, right-click the name of the project to which you want to add the ATL COM+ 1.0 component.
2. On the shortcut menu, click **Add**, and then click **Add Class**.
3. In the Add Class dialog box, in the Templates pane, click **ATL COM+ 1.0 Component**, and then click **Open** to display the ATL COM+ 1.0 Component Wizard.

See Also

Tasks

Adding a Method

Concepts

Adding a Class

ATL COM+ 1.0 Component Wizard

Use this wizard to add an object to your project that supports COM+ 1.0 services, including transactions.

You can specify whether the object supports dual interfaces and Automation. You can also indicate support for the error information interface, enhanced object control, transactions, and asynchronous message queuing.

Names

Specify the names for the object, interface, and classes to be added to your project. With the exception of **Short name**, all other boxes can be edited independently of the others. If you change the text for **Short name**, the change is reflected in the names of all other boxes in this page. If you change the **Coclass** name in the COM section, the change is reflected in the **Type** and **ProgID** boxes, but the **Interface** name does not change. This naming behavior is designed to make all the names easily identifiable for you as you develop your control.

Short name

Sets the abbreviated name for the object. The name you provide determines the **Class** and **Coclass** names, the **.cpp file** and **.h file** names, the **Interface** name, the **Type** names, and the **ProgID**, unless you change those fields individually.

.h file

Sets the name of the header file for the new object's class. By default, this name is based on the name you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice, or to append the class declaration to an existing file. If you choose an existing file, the wizard will not save it to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class declaration should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

Class

Sets the name of the class to be created. This name is based on the name you provide in **Short name**, preceded by 'C', the typical prefix for a class name.

.cpp file

Sets the name of the implementation file for the new object's class. By default, this name is based on the name you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice. The file is not saved to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class implementation should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

Attributed

Indicates whether the object uses attributes. If you are adding an object to an attributed ATL project, this option is selected and not available to change. That is, you can add only attributed objects to a project created with attribute support.

If you select this option for an ATL project that does not have attribute support, the wizard prompts you to specify whether you want to add attribute support to the project.

Any objects you add following setting this option are designated as attributed by default (the check box is selected). You can clear this box to add an object that does not use attributes.

See Application Settings, ATL Project Wizard and Basic Mechanics of Attributes for more information.

COM

Provides information about the COM functionality for the object.

Coclass

Sets the name of the component class that contains a list of interfaces supported by the object.

Note

If you create your project using attributes, or if you indicate on this wizard page that the COM+ 1.0 component uses attributes, you cannot change this option because ATL does not include the **coclass** attribute.

Type

Sets the object description that will appear in the registry

Interface

Sets the interface you create for your object. This interface contains your custom methods.

ProgID

Sets the name that containers can use instead of the CLSID of the object.

See Also

Tasks

Adding an ATL COM+ 1.0 Component

COM+ 1.0, ATL COM+ 1.0 Component Wizard

Use this page of the ATL COM+ 1.0 Component Wizard to specify interface type and additional interfaces to be supported.

For more information on ATL projects and ATL COM classes, see ATL Reference.

Interface

Indicates the type of interface the object supports. By default, the object supports a dual interface.

Option	Description
Dual	Specifies that the object supports a dual interface (its vtable has custom interface functions and late-binding **IDispatch** methods). Allows both COM clients and Automation controllers to access the object.
Custom	Specifies that the object supports a custom interface (its vtable has custom interface functions). A custom interface can be faster than a dual interface, especially across process boundaries. • **Automation compatible** Adds automation support to the custom interface. For attributed projects, sets the **oleautomation** attribute in the coclass.

Queueable

Indicates that clients can call this component asynchronously using message queues. Adds the component attributed macro custom(TLBATTR_QUEUEABLE, 0) to the .h file (attributed projects) or to the .idl file (nonattributed projects).

Support

Indicates additional support for error handling and object control.

Option	Description
ISupportErrorInfo	Creates support for the ISupportErrorInfo interface so the object can return error information to the client.
IObjectControl	Provides your object access to the three IObjectControl methods: Activate, CanBePooled, and Deactivate.
IObjectConstruct	Creates support for the IObjectConstruct interface to manage passing in parameters from other methods or objects.

Transaction

Indicates that the object supports transactions. Includes the file mtxattr.h in the .idl file (nonattributed projects).

Option	Description
Supported	Specifies that the object is never the root of a transaction stream by adding the component attribute macro custom(TLBATTR_TRANS_SUPPORTED,0) to the .h file (attributed projects) or to the .idl file (nonattributed projects).
Required	Specifies that the object may or may not be the root of a transaction stream by adding the component attribute macro custom(TLBATTR_TRANS_REQUIRED,0) to the .h file (attributed projects) or to the .idl file (nonattributed projects).
Not supported	Specifies that the object excludes transactions. Adds the component attribute macro custom(TLBATTR_TRANS_NOTSUPP,0) to the .h file (attributed projects) or to the .idl file (nonattributed projects).
Requires new	Specifies that the object is always the root of a transaction stream by adding the component attribute macro custom(TLBATTR_TRANS_REQNEW,0) to the .h file (attributed projects) or to the .idl file (nonattributed projects).

See Also

Tasks

Adding an ATL COM+ 1.0 Component

Reference

ATL COM+ 1.0 Component Wizard

Adding an ATL Control

Use this wizard to add a user interface object to a project that supports interfaces for all potential containers. To support these interfaces, the project must have been created as an ATL application or as an MFC application that contains ATL support. You can use the ATL Project Wizard to create an ATL application, or add an ATL object to your MFC application to implement ATL support for an MFC application.

To add an ATL control to your project

1. In either Solution Explorer or Class View, right-click the name of the project to which you want to add the ATL simple object.
2. Click **Add** from the shortcut menu, and then click **Add Class**.
3. In the Add Class dialog box, in the Templates pane, click **ATL Control**, and then click **Open** to display the ATL Control Wizard.

Using the ATL Control Wizard, you can create one of three types of controls:

- A standard control
- A composite control
- A DHTML control

Additionally, you can reduce the size of the control and remove interfaces that are not used by most containers by selecting **Minimal control** on the **Options** page of the wizard.

See Also

Tasks

ATLFire Sample: Demonstrates Building a Windowed Control

Reference

Fundamentals of ATL COM Objects

Concepts

Adding Functionality to the Composite Control

ATL Control Wizard

This wizard inserts into an ATL project (or an MFC project with ATL support) an ATL control. You can use this wizard to insert one of three types of controls:

- A standard control
- A composite control
- A DHTML control

Additionally, you can specify a minimal control, removing the interfaces from the Interfaces list, which are provided as defaults for controls to open in most containers. You can set the interfaces you want supported for the control in the **Interfaces** page of the wizard.

Names

Specify the names for the object, interface, and classes to be added to your project. With the exception of **Short name**, all other boxes can be edited independently. If you change the text for **Short name**, the change is reflected in the names of all other boxes in this page. If you change the **Coclass** name in the COM section, then the change is reflected in the **Type** and **ProgID** boxes, but the **Interface** name does not change. This naming behavior is designed to make all the names easily identifiable for you as you develop your control.

Note
Coclass is editable on only nonattributed controls. If your project attributed, you cannot edit **Coclass**.

C++

Provides information for the C++ class created to implement the object.

Short name

Sets the abbreviated name for the object. The name that you provide determines the class and **Coclass** names, the file (**.CPP** and **.H**) names, the interface name, the **Type** names, and the **ProgID**, unless you change those fields individually.

Class

Sets the name of the class that implements the object. This name is based on the name you provide in **Short name**, preceded by 'C', the typical prefix for a class name.

.h file

Sets the name of the header file for the new object's class. By default, this name is based on the name you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice, or to append the class declaration to an existing file. If you choose an existing file, the wizard will not save it to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class declaration should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

.cpp file

Sets the name of the implementation file for the new object's class. By default, this name is based on the name you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice. The file is not saved to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class implementation should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

Attributed

Indicates whether the object uses attributes. If you are adding an object to an attributed ATL project, this option is selected and not available to change. That is, you can add only attributed objects to a project created with attribute support.

You can add an attributed object only to an ATL project that uses attributes. If you select this option for an ATL project that

does not have attribute support, the wizard prompts you to specify whether you want to add attribute support to the project.

Any objects you add after setting this option are designated as attributed by default (the check box is selected). You can clear this box to add an object that does not use attributes.

See Application Settings, ATL Project Wizard and Basic Mechanics of Attributes for more information.

COM

Provides information about the COM functionality for the object.

Coclass

Sets the name of the component class that contains a list of interfaces supported by the object.

Note
If you create your project using attributes, or if you indicate on this wizard page that the control uses attributes, you cannot change this option because ATL does not include the **coclass** attribute.

Interface

Sets the name of the interface for the object. An interface name is prepended with "I" by default.

Type

Sets the object description that will appear in the registry

ProgID

Sets the name that containers can use instead of the CLSID of the object.

See Also

Tasks

Adding an ATL Control

Reference

Fundamentals of ATL COM Objects

Concepts

Adding Functionality to the Composite Control

Options, ATL Control Wizard

Insert "Search Results" summary here.

Use this page of the wizard to define the type of control you are creating and the level of interface support it contains.

Note
If you are using this wizard with a device project, see Wizard Options for Options, ATL Control Wizard.

Control type

The kind of control you want to create.

- **Standard control: An ActiveX control.**
- **Composite control**: An ActiveX control that can contain (similar to a dialog box) other ActiveX controls or Windows controls. A composite control includes the following:
  - A template for the dialog box that implements the composite control.
  - A custom resource, REGISTRY, which automatically registers the composite control when invoked.
  - A C++ class that implements the composite control.
  - A COM interface, exposed by the composite control.
  - An HTML test page containing the composite control.

  By default, this control sets CComControlBase::m_bWindowOnly to true, to indicate that this is a windowed control. It implements a sink map. For more information, see ATL Support for DHTML Controls.
- **DHTML control**: An ATL DHTML control specifies the user interface, using HTML. The DHTML UI class contains a COM map. By default, this control sets CComControlBase::m_bWindowOnly to true, to indicate that this is a windowed control.

  For more information, see Identifying the Elements of the DHTML Control Project.

Minimal control

Supports only the interfaces that are absolutely needed by most containers. You can set **Minimal control** for any of the control types: you can create a minimal standard control, a minimal composite control, or a minimal DHTML control.

Aggregation

Adds aggregation support for the control you are creating. For more information, see Aggregation.

- **Yes**: Create a control that can be aggregated.
- **No**: Create a control that cannot be aggregated.
- **Only**: Create a control that can only be instantiated through aggregation.

Threading model

Specifies that the threading model used by the control.

- **Single**: The control will run only in the primary COM thread.
- **Apartment**: The control can be created in any single thread apartment. The default.

Interface

The type of interface this control exposes to the container.

- **Dual**: Creates an interface that exposes properties and methods through **IDispatch** and directly through the VTBL.
- **Custom**: Creates an interface that exposes methods directly through a VTBL.

  If you select **Custom**, then you can specify that the control is **Automation compatible**. If you select **Automation**

compatible, then the wizard adds the oleautomation attribute to the interface in the IDL, and the interface can be marshaled by the universal marshaler in oleaut32.dll. See Marshaling Details in the *Platform SDK* for more information.

Additionally, if you select **Automation compatible**, then all parameters for all methods in the control must be **VARIANT** compatible.

Support

Sets additional miscellaneous support for the control.

- **Connection points**: Enables connection points for your object by making your object's class derive from IConnectionPointContainerImpl and allowing it to expose a source interface.
- **Licensed**: Adds support to the control for licensing. Licensed controls can only be hosted if the client machine has the correct license.

See Also

Reference

ATL Control Wizard

Interfaces, ATL Control Wizard

This page of the wizard identifies the interfaces that the control supports. By default, the supported interfaces are those typically used by most containers.

Note
If you are using this wizard with a device project, see Wizard Options for Interfaces, ATL Control Wizard.

Note
If you selected **Minimal control** on the Options tab, no interfaces appear by default in the **Supported** list box.

Not supported

Indicates the available interfaces that are not currently supported for the control.

Supported

Indicates the interfaces that are currently supported for the control.

Transfer button	Description
>	Adds to the **Supported** list the interface name currently selected in the **Not Supported** list.
>>	Adds to the **Supported** list all interface names available in the **Not Supported** list.
<	Removes the interface name currently selected in the **Supported** list.
<<	Removes all interface names currently listed in the **Supported** list.

See Also

Reference

ATL Control Wizard

Appearance, ATL Control Wizard

Insert "Search Results" summary here.

Use this page of the wizard to identify additional user element options for the control. This page is available for controls identified as **Standard controls** under **Control type** on the Options, ATL Control Wizard page.

View status

Sets the appearance of the control within the container.

- **Opaque**: Sets the **VIEWSTATUS_OPAQUE** bit in the VIEWSTATUS enumeration and draws the entire control rectangle passed to the CComControlBase::OnDraw method. The control appears completely opaque, and none of the container shows behind the control boundaries.

  This setting helps the container draw the control more quickly. If this option is not selected, the control can contain transparent parts.

  Only an opaque control can have a solid background.

- Sets the **VIEWSTATUS_SOLIDBKGND** bit in the **VIEWSTATUS** enumeration. The control's background appears as a solid color with no pattern.

  This option is available only if the **Opaque** option is also selected.

Add control based on

Sets the control to be based on a Windows control type by adding a CContainedWindow data member to the class implementing the control. It also adds a message map and message handler functions to handle Windows messages for the control. Choose from the list the type of Windows control you want to create, if any.

- **Button**
- **ListBox**
- **SysAnimate32**
- **SysListView32**
- **ComboBox**
- **RichEdit**
- **SysDateTimePick32**
- **SysMonthCal32**
- **ComboBoxEx32**
- **ScrollBar**
- **SysHeader32**
- **SysTabControl32**
- **Edit**
- **Static**
- **SysIPAddress32**
- **SysTreeView32**

Misc status

Sets additional appearance and behavior options for the control.

- **Invisible at run-time**: Sets the control to be invisible at run time. You can use invisible controls to perform operations in the background, such as firing events at timed intervals.

- **Acts like button**: Sets the **OLEMISC_ACTSLIKEBUTTON** bit in the OLEMISC enumeration to enable a control to act like a button. If the container has marked the control's client site as a default button, selecting this option enables your button control to display itself as a default button by drawing itself with a thicker frame. See CComControlBase::GetAmbientDisplayAsDefault for more information.
- **Acts like label**: Sets the **OLEMISC_ACTSLIKELABEL** bit in the **OLEMISC** enumeration to enable a control to replace the container's native label. The container determines what to do with this flag, if anything.

Other

Sets additional behavior options for the control.

- **Normalized DC**: Sets the control to create a normalized device context when it is called to draw itself. This action standardizes the control's appearance, but it makes drawing less efficient.
- **Window only**: Specifies that your control cannot be windowless. If you do not select this option, your control is automatically windowless in containers that support windowless objects, and it is automatically windowed in containers that do not support windowless objects. Selecting this option forces your control to be windowed, even in containers that support windowless objects.
- **Insertable**: Select this option to have your control appear in the **Insert Object** dialog box of applications such as Word and Excel. Your control can then be inserted by any application that supports embedded objects through this dialog box.

See Also

Tasks

SUBEDIT Sample: Superclasses a Standard Windows Control

Reference

ATL Control Wizard

Stock Properties, ATL Control Wizard

This page of the wizard identifies the stock properties supported for the control. By default, no properties are identified.

Note
If you are using this wizard with a device project, see Unsupported Wizard Options for Stock Properties, ATL Control Wizard.

Not supported

Indicates the available properties that are not currently supported for the control.

Supported

Indicates the properties that are currently supported for the control.

Transfer button	Description
>	Adds to the **Supported** list the property name currently selected in the **Not Supported** list.
>>	Adds to the **Supported** list all property names available in the **Not Supported** list.
<	Removes the property name currently selected in the **Supported** list.
<<	Removes all property names currently listed in the **Supported** list.

See Also

Reference

ATL Control Wizard

Adding an ATL Dialog

To add an ATL dialog to your project, your project must be either an ATL project or an MFC project that includes ATL support. You can use the ATL Project Wizard to create an ATL application, or add an ATL object to your MFC application to implement ATL support for an MFC application.

By default, the ATL Dialog Wizard implements a dialog box derived from CAxDialogImpl. This class includes support for hosting ActiveX and Windows controls. If you do not want the overhead of ActiveX control support, once the wizard has generated your code, replace all instances of **CAxDialogImpl** with either CSimpleDialog or CDialogImpl as your base class.

Note
CSimpleDialog creates only modal dialog boxes that support only Windows common controls. **CDialogImpl** creates either modal or modeless dialog boxes.

To add an ATL dialog resource to your project

1. Create an ATL project using the ATL Project Wizard.
2. From Class View, right-click the project name and click **Add** from the shortcut menu. Click **Add Class**.
3. In the Templates pane of the Add Class dialog box, click **ATL Dialog**. Click **Open** to display the ATL Dialog Wizard.

For more information, see Implementing a Dialog Box.

See Also

Reference

Message Maps (ATL)

Concepts

Adding a Class

Other Resources

ATL Window Classes

ATL Dialog Wizard

This wizard inserts into the project an ATL dialog box object, derived from CAxDialogImpl. A dialog box derived from **CAxDialogImpl** can host ActiveX controls.

The wizard creates a dialog resource with default **OK** and **Cancel** buttons. You can edit the dialog resource and add ActiveX controls using the Dialog Editor in Resource View.

The wizard inserts into the header file a message map and declarations for handling the default click events. See Implementing a Dialog Box for more information about ATL dialog boxes.

Short name

Sets the abbreviated name for the ATL dialog object. The name you provide determines the class name and the file (.cpp and .h) names, unless you change those fields individually.

Class

Sets the name of the class to be created. This name is based on the name you provide in **Short name**, preceded by 'C', the typical prefix for a class name.

.h file

Sets the name of the header file for the new object's class. By default, this name is based on the name you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice, or to append the class declaration to an existing file. If you choose an existing file, the wizard will not save it to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class declaration should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

.cpp file

Sets the name of the implementation file for the new object's class. By default, this name is based on the name you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice. The file is not saved to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class implementation should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

See Also

Tasks

Adding an ATL Dialog

Adding an ATL OLE DB Consumer

Use this wizard to add an ATL OLE DB consumer to a project. An ATL OLE DB consumer consists of an OLE DB accessor class and data bindings necessary to access a data source. The project must have been created as an ATL COM or ATL Server application, or as an MFC or Win32 application that contains ATL support (which the ATL OLE DB Consumer Wizard adds automatically).

Note You can add an OLE DB consumer to an MFC project. If you do, the ATL OLE DB Consumer Wizard adds the necessary COM support to your project. This assumes that when you created the MFC project, you selected the **ActiveX controls** check box (in the **Advanced Features** page of the MFC Project Application Wizard), which is checked by default. Selecting this option ensures that the application calls **CoInitialize** and **CoUninitialize**. If you did not select **ActiveX controls** when you created the MFC project, you need to call **CoInitialize** and **CoUninitialize** in your main code.

To add an ATL OLE DB consumer to your project

1. In Class View, right-click the project. On the shortcut menu, click **Add** and then click **Add Class**.
2. In the Visual C++ folder, double-click the **ATL OLE DB Consumer** icon or select it and click **Open**.

   The ATL OLE DB Consumer Wizard opens.
3. Define settings as described in ATL OLE DB Consumer Wizard.
4. Click **Finish** to close the wizard. The newly created OLE DB consumer code will be inserted in your project.

See Also

Concepts

Adding Functionality with Code Wizards

ATL OLE DB Consumer Wizard

This wizard sets up an OLE DB consumer class with the data bindings necessary to access the specified data source through the specified OLE DB provider.

Note
This wizard requires you to click the **Data Source** button to select a data source before entering names in the **Class** and **.h file** fields.

Data Source

The **Data Source** button lets you set up the specified data source using the specified OLE DB provider. When you click this button, the **Data Link Properties** dialog box appears. For more information on building connection strings and the **Data Link Properties** dialog box, see Data Link API Overview in the *Platform SDK* documentation.

Note
In previous releases, Shift-clicking the **Data Source** button opened a File Open dialog to allow you to select a Data Link (.udl) file. This functionality is no longer supported.

The dialog box has four tabs:

- **Provider** tab
- **Connection** tab
- **Advanced** tab
- **All** tab

Class

After you select a data source, this box is populated with a default class name based on the table or stored procedure that you selected (see **Select a data source** below). You can edit the class name.

.h file

After you select a data source, this box is populated with a default header class name based on the table or stored procedure that you selected (see **Select a data source** below). You can edit the header file's name or select an existing header file.

Attributed

This option specifies whether the wizard will create consumer classes using attributes or template declarations. When you select this option, the wizard uses attributes instead of template declarations (this is the default option). When you deselect this option, the wizard uses template declarations instead of attributes.

- If you select a consumer **Type** of Table, the wizard uses the **db_source** and **db_table** attributes to create the table and table accessor class declarations, and uses **db_column** to create the column map, for example:

```
// Inject table class and table accessor class declarations
[
    db_source("<initialization_string>"),
    db_table("dbo.Orders")
]
...
// Column map
    [ db_column(1, status=m_dwOrderIDStatus,        length=m_dwOrderIDLength) ] LONG
m_OrderID;
    [ db_column(2, status=m_dwCustomerIDStatus,        length=m_dwCustomerIDLength)
] TCHAR m_CustomerID[6];
    ...
```

instead of using the **CTable** template class to declare the table and table accessor class, and the BEGIN_COLUMN_MAP and END_COLUMN_MAP macros to create the column map, for example:

```
// Table accessor class
class COrdersAccessor;
// Table class
class COrders : public CTable<CAccessor<COrdersAccessor> >;
...
// Column map
BEGIN_COLUMN_MAP(COrderDetailsAccessor)
    COLUMN_ENTRY_LENGTH_STATUS(1, m_OrderID,         m_dwOrderIDLength, m_dwOrderIDSt
atus)
    COLUMN_ENTRY_LENGTH_STATUS(2, m_CustomerID,         m_dwCustomerIDLength, m_dwCus
tomerIDStatus)
    ...
END_COLUMN_MAP()
```

- If you select a consumer **Type** of Command, the wizard uses the **db_source** and **db_command** attributes, and uses **db_column** to create the column map, for example:

```
[
    db_source("<initialization_string>"),
    db_command("SQL_command")
]
...
// Column map using db_column is the same as for consumer type of 'table'
```

instead of using the command and command accessor class declarations in the command class' .h file, for example:

```
Command accessor class:
class CListOrdersAccessor;
Command class:
class CListOrders : public CCommand<CAccessor<CListOrdersAccessor> >;
...
// Column map using BEGIN_COLUMN_MAP ... END_COLUMN_MAP is the same as
// for consumer type of 'table'
```

See Basic Mechanics of Attributes for more information.

Type

Select one of these radio buttons to specify whether the consumer class will be derived from **CTable** or **CCommand** (default).

Table

Select this option if you want to use **CTable** or **db_table** to create the table and table accessor class declarations.

Command

Select this option if you want to use **CCommand** or **db_command** to create the command and command accessor class declarations. This is the default selection.

Support

Select the check boxes to specify the kinds of updates to be supported in the consumer (the default is none). Each of the following will set DBPROP_IRowsetChange and the appropriate entries for DBPROP_UPDATABILITY in the property set map.

Change

Specifies that the consumer support updates of row data in the rowset.

Insert

Specifies that the consumer support insertion of rows into the rowset.

Delete

Specifies that the consumer support deletion of rows from the rowset.

See Also

Tasks

Adding an ATL OLE DB Consumer

Reference

Connection Strings and Data Links (OLE DB)

Concepts

Adding Functionality with Code Wizards

Adding an ATL OLE DB Provider

Use this wizard to add an ATL OLE DB provider to a project. An ATL OLE DB provider consists of a data source, session, command, and rowset classes. The project must have been created as an ATL COM application.

To add an ATL OLE DB provider to your project

1. In Class View, right-click the project. On the shortcut menu, click **Add** and then click **Add Class**.
2. In the Visual C++ folder, double-click the **ATL OLE DB Provider** icon or select it and click **Open**.

   The ATL OLE DB Provider Wizard opens.
3. Define settings as described in ATL OLE DB Provider Wizard.
4. Click **Finish** to close the wizard, which will insert the newly created OLE DB consumer code in your project.

See Also

Concepts

Adding Functionality with Code Wizards

ATL OLE DB Provider Wizard

This wizard creates the classes that compose an OLE DB provider.

Short name

Type the short name of the provider to be created here. The other edit boxes in the wizard will automatically populate based on what you type here. Edit the other name boxes, if desired.

Coclass

The name of the coclass. The ProgID name will change to match this name.

Attributed

This option specifies whether the wizard will create provider classes using attributes or template declarations. When you select this option, the wizard uses attributes instead of template declarations (this is the default option if you created an attributed project). When you deselect this option, the wizard uses template declarations instead of attributes (this is the default option if you created a non-attributed project).

If you select this option when you created a non-attributed project, the wizard warns you that the project will be converted to an attributed project and asks you whether to continue or not.

ProgID

The ProgID, or programmatic identifier, is a text string that your application can use instead of a GUID. The ProgID name has the form *Projectname*.*Coclassname*.

Version

The version number of your provider. The default is 1.

DataSource class

The name of the data source class, of the form C*Shortname*Source.

DataSource .h file

The header file for the data source class. You can edit this file's name or select an existing header file.

Session class

The name of the session class, of the form C*Shortname*Session.

Session .h file

The header file for the session class. You can edit this file's name or select an existing header file.

Command class

The name of the command class, of the form C*Shortname*Command.

Command .h file

The header file for the command class. This name cannot be edited and depends on the name of the rowset header file.

Rowset class

The name of the rowset class, of the form C*Shortname*Rowset.

Rowset .h file

The header file for the rowset class. You can edit this file's name or select an existing header file.

Rowset .cpp file

The provider's implementation file. You can edit this file's name or select an existing implementation file.

See Also

Tasks

Adding an ATL OLE DB Provider

Adding an ATL Property Page

To add an Active Template Library (ATL) property page to your project, your project must have been created as an ATL application or as an MFC application that contains ATL support. You can use the ATL Project Wizard to create an ATL application or add an ATL object to your MFC application to implement ATL support for an MFC application.

If you are adding a property page for a control, your control must support the ISpecifyPropertyPagesImpl interface. By default, this interface is in the derivation list of your control class when you create an ATL control using the ATL Control Wizard.

Note
If your control class does not have ISpecifyPropertyPagesImpl in its derivation list, you must add it manually.

To add an ATL property page to your project

1. In either Solution Explorer or Class View, right-click the name of the project to which you want to add the ATL property page.
2. From the shortcut menu, click **Add** and then click **Add Class**.
3. In the Add Class dialog box, in the Templates pane, click **ATL Property Page** and then click **Open** to display the ATL Property Page Wizard.

Once you create a property page for a control, you must provide the PROP_PAGE entry in the property map for the control.

See Also

Reference

Fundamentals of ATL COM Objects

Concepts

ATL COM Property Pages

Example: Implementing a Property Page

ATL Property Page Wizard

This wizard adds a property page into an ATL project or to an MFC project with ATL support. An ATL property page provides a user interface for setting the properties (or calling the methods) of one or more COM objects.

Names

Specify the names for the object, interface, and classes to be added to your project. With the exception of **Short name**, all other boxes can be edited independently. If you change the text for **Short name**, the change is reflected in the names of all other boxes in this page. If you change the **Coclass** name in the COM section, then the change is reflected in the **Type** and **ProgID** boxes. This naming behavior is designed to make all the names easily identifiable for you as you develop your property page.

Note
Coclass is editable on only nonattributed projects. If your project attributed, you cannot edit **Coclass**.

C++

Provides information for the C++ class created to implement the object.

Short name

Sets the abbreviated name for the object. The name that you provide determines the class and **Coclass** names, the file (**.cpp** and **.h**) names, the **Type** name, and the **ProgID**, unless you change those fields individually.

.h file

Sets the name of the header file for the new object's class. By default, this name is based on the name you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice, or to append the class declaration to an existing file. If you choose an existing file, the wizard will not save it to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class declaration should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

Class

Sets the name of the class that implements the object. This name is based on the name you provide in **Short name**, preceded by 'C', the typical prefix for a class name.

.cpp file

Sets the name of the implementation file for the new object's class. By default, this name is based on the name you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice. The file is not saved to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class implementation should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

Attributed

Indicates whether the object uses attributes. If you are adding an object to an attributed ATL project, this option is selected and not available to change, that is, you can add only attributed objects to a project created with attribute support.

You can add an attributed object only to an ATL project that uses attributes. If you select this option for an ATL project that does not have attribute support, the wizard prompts you to specify whether you want to add attribute support to the project.

Any objects you add after setting this option are designated as attributed by default (the check box is selected). You can clear this box to add an object that does not use attributes.

See Application Settings, ATL Project Wizard and Basic Mechanics of Attributes for more information.

COM

Provides information about the COM functionality for the object.

Coclass

Sets the name of the component class that contains a list of interfaces supported by the object.

Note
If you create your project using attributes, or if you indicate on this wizard page that the property page uses attributes, you cannot change this option because ATL does not include the **coclass** attribute.

Type

Sets the object description that will appear in the registry

ProgID

Sets the name that containers can use instead of the CLSID of the object.

See Also

Reference

Options, ATL Property Page Wizard

Strings, ATL Property Page Wizard

Concepts

Example: Implementing a Property Page

Options, ATL Property Page Wizard

Use this page of the wizard to define the threading model and aggregation level of property page you are creating.

Note
If you are using this wizard with a device project, see Unsupported Wizard Options for Options, ATL Property Page Wizard.

Threading model

Specifies the threading model used by the property page.

See Specifying the Project's Threading Model for more information.

Option	Description
Single	The property page runs only in the primary COM thread.
Apartment	The property page can be created in any single thread apartment. The default.

Aggregation

Adds aggregation support for the property page you are creating. See Aggregation for more information.

Option	Description
Yes	Create a property page that can be aggregated.
No	Create a property page that cannot be aggregated.
Only	Create a property page that can only be instantiated through aggregation.

See Also

Reference

ATL Property Page Wizard

Strings, ATL Property Page Wizard

Strings, ATL Property Page Wizard

Provides the text associated with the property page.

Title

Sets the text that appears on the tab of the property page.

Doc string

Sets a text string describing the page. This string can be displayed in the property sheet dialog box. The property frame could use the description in a status line or tool tip. The standard property frame currently does not use this string.

Help file

Sets the name of the help file that describes how to use the property page. This name should not include a path. When the user presses **Help**, the frame opens the help file in the directory named in the value of the HelpDir key in the property page registry entries under its CLSID.

See Also

Reference

ATL Property Page Wizard
Options, ATL Property Page Wizard

Adding an ATL Performance Monitor Object Manager

Use the ATL Performance Monitor Object Manager Wizard to add a performance monitor object manager (perfmon), in either attributed (perfmon) or nonattributed (CPerfMon) form to an ATL DLL project.

Note
You can use the ATL Project Wizard to create an ATL project, or you can add an ATL object to your MFC application to implement ATL support for an MFC application.

Creating a performance monitor object manager is the first step to adding performance monitoring support to your system or server project. Performance monitoring allows your applications and components to publish, capture, and analyze the performance data that applications, services, and drivers provide. You can use this information to determine system bottlenecks and fine-tune system and application performance.

You must create a performance monitor object manager in an ATL DLL project before you can add performance objects and implement performance counters for the object. A performance monitor object manager can have multiple performance objects, which in turn can have multiple counters. The following figure illustrates how the performance monitoring architecture works in your ATL project.

Performance Monitor Object Manager with Performance Objects and Counters

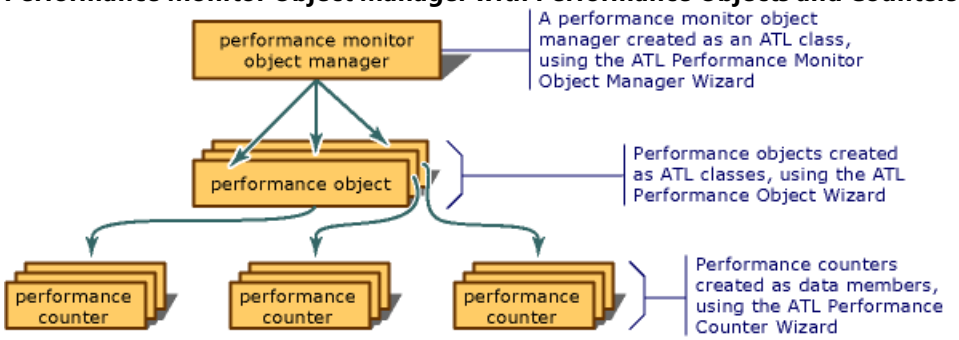

To add an ATL performance monitor object manager to your project

1. In either Solution Explorer or Class View, right-click the name of the project to which you want to add the ATL performance monitor object manager.
2. From the shortcut menu, click **Add**, and then click **Add Class**.
3. In the Add Class dialog box, in the Templates pane, click **ATL Performance Monitor Object Manager**, and then click **Open** to display the ATL Performance Monitor Object Manager Wizard.

By clicking **Generate Sample Object** in the ATL Performance Monitor Object Manager Wizard, you automatically add one sample performance object to the performance monitor object manager and one sample performance counter to the performance object.

Note
A performance monitor can work with an ATL service, but such a service must be added to the ATL DLL project containing the performance monitor. See Performance Monitoring for more information.

See Also

Reference

Performance Objects and Counters

ATL Performance Monitor Object Manager Wizard

This wizard inserts into the project a performance monitor object manager. Using this wizard, you can specify the name of the class and the header file name. By default, the **register** parameter is set to **true**. Additionally, you can add a sample performance object and performance counter, and you can add TODO comments to the project.

Class name

Sets the name of the performance monitor object manager class that you are adding.

.h file

Sets the name of the header file for the new object's class. By default, this name is based on the name you provide in **Class name**. Click the ellipsis button to save the file name to the location of your choice, or to append the class declaration to an existing file. If you choose an existing file, the wizard will not save it to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class declaration should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

Generate TODO comments

Adds comments to the header file indicating the next steps needed to implement a performance monitor and counters.

Note
If you create an attributed object and add TODO comments, these comments stay in place even after you have added attributed performance objects and counters.

Generate sample object

Adds one sample performance object and object class (C*ProjName*SampleObject) to your new performance monitor object manager. To the performance object, it adds one sample performance counter and adds a data member for the counter (`ULONG m_nCounter`) to the performance object. The counter type is set to PERF_COUNTER_RAW C OUNT, a simple instantaneous counter value.

Attributed code

Sets the class to use attributes. Your performance monitor object manager is inserted as follows:

```
[ perfmon(name="Perf_ClassName", register=true) ]
class ClassName
{
public:
   // TODO: First add Performance Monitor objects to your project.
   // You can then add counters to those objects. Use the context
   // menu of MyPMManager in ClassView to add a PerfMon Object
   // to your project. Then use the context menu of the newly
   // created PerfMon Object to add a PerfMon Counter.
};
```

If you select **Attributed code** and your project was not created using attributes, then the wizard adds attribute support to the project.

See Also

Tasks

Adding an ATL Performance Monitor Object Manager
Adding a Performance Object
Adding a Performance Monitor Counter

Concepts

Performance Monitoring

Adding an XML Web Service with ATL Server

To add an XML Web service using ATL Server to your project, your project must have been created using the ATL Server Project Wizard.

To add an XML Web service using ATL Server to your project

1. In either Solution Explorer or Class View, right-click the name of the project to which you want to add the XML Web service using ATL Server.
2. From the shortcut menu, click **Add**, and then click **Add Class**.
3. In the Add Class dialog box, in the Templates pane, click **ATL Server Web Service**, and then click **Open** to display the ATL Server Web Service Object Wizard.
4. Enter the settings required by the wizard.
5. Click **Finish** to add the XML Web service to your project.

Note To do native C++ Web development on Windows NT 4.0, you must have Internet Information Services (IIS) installed.

See Also

Reference

ATL Server Web Service Object Wizard
ATL Server Project Wizard

Concepts

Adding a Class

Other Resources

XML Web Services Created with ATL Server
ATL Server Concepts

ATL Server Web Service Object Wizard

This wizard inserts into the project a minimal XML Web service using ATL Server consisting of the following items:

Item	Description
Class	Provides the implementation of the XML Web service. The request_handler attribute on the class allows it to handle HTTP requests, and the soap_handler attribute allows it to parse incoming SOAP requests. Each SOAP method in the class must use the soap_method attribute and be part of an interface so that marshaling information can be provided for the parameters.
Interface	Declares the SOAP methods provided by the XML Web service and allows marshaling information to be provided for the parameters.
.disco file	Allows potential clients to discover your XML Web service. See XML Web Service Discovery for more details.

The settings for this wizard are described below:

Item name

Specify the name that will be used in the generated code. The text you provide will be used to name the following items:

Item	Name
XML Web service name	*ItemName* **Service**
XML Web service namespace	*ItemName* **Service**
C++ namespace	*ItemName* **Service**
Interface	**I** *ItemName* **Service**
Class	**C** *ItemName* **Service**

.h file

Specify the file name in which the wizard should generate the code. If the file already exists, you will be asked whether the wizard should merge the newly generated code into that file. The name of the .disco file is also determined from the value entered here.

Generate TODO comments

Select this box to insert TODO comments in the generated code.

Generate sample code

Select this box to have a HelloWorld method added to the interface and class generated by the wizard.

See Also

Tasks

Adding an XML Web Service with ATL Server

Reference

ATL Server Project Wizard

Concepts

Adding a Class

Other Resources

XML Web Services Created with ATL Server

ATL Server Concepts

Adding an ATL Simple Object

To add an ATL (Active Template Library) object to your project, your project must have been created as an ATL application or as an MFC application that contains ATL support. You can use the ATL Project Wizard to create an ATL application, or add an ATL object to your MFC application to implement ATL support for an MFC application.

You can define COM interfaces for your new ATL object when you first create it, or add them later by using the Implement Interface command from the Class View shortcut menu.

To add an ATL simple object to your ATL COM project

1. In either Solution Explorer or Class View, right-click the name of the project to which you want to add the ATL simple object.
2. From the shortcut menu, click **Add**, and then click **Add Class**.
3. In the Add Class dialog box, in the Templates pane, click **ATL Simple Object**, and then click **Open** to display the ATL Simple Object Wizard.
4. Set additional options for your project on the Options page of the ATL Simple Object wizard.
5. Click **Finish** to add the object to your project.

See Also

Tasks

Adding a New Interface in an ATL Project
Adding Connection Points to an Object
Adding a Method
Adding an MFC Class
Adding a Generic C++ Class

Concepts

Adding a Class

ATL Simple Object Wizard

This wizard inserts into the project a minimal COM object. Use this page of the wizard to specify the names that identify the C++ class and files for your object and its COM functionality.

Use the Options page of this wizard to specify the object's threading model, its aggregation support, and whether it supports dual interfaces and Automation. You can also indicate support for the error information interface, connection points, Internet Explorer support, and free-threaded marshaling.

Names

Specify the names for the object, interface, and classes to be added to your project. With the exception of **Short name**, all other boxes can be edited independently of the others. If you change the text for **Short name**, the change is reflected in the names of all other boxes in this page. If you change the **Coclass** name in the COM section, the change is reflected in the **Type** and **ProgID** boxes, but the **Interface** name does not change. This naming behavior is designed to make all the names easily identifiable for you as you develop your control.

Note
Coclass is editable on only nonattributed projects. If your project attributed, you cannot edit **Coclass**.

C++

Provides information for the C++ class created for the object.

Short name

Sets the abbreviated name for the object. The name you provide determines the **Class** and **Coclass** names, the **.cpp file** and **.h file** names, the **Interface** name, the **Type** names, and the **ProgID**, unless you change those fields individually.

.h file

Sets the name of the header file for the new object's class. By default, this name is based on the name you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice, or to append the class declaration to an existing file. If you choose an existing file, the wizard will not save it to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class declaration should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

Class

Sets the name of the class to be created. This name is based on the name you provide in **Short name**, preceded by 'C', the typical prefix for a class name.

.cpp file

Sets the name of the implementation file for the new object's class. By default, this name is based on the name you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice. The file is not saved to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class implementation should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

Attributed

Indicates whether the object uses attributes. If you are adding an object to an attributed ATL project, this option is selected and not available to change. That is, you can add only attributed objects to a project created with attribute support.

You can add an attributed object only to an ATL project that uses attributes. If you select this option for an ATL project that does not have attribute support, the wizard prompts you to specify whether you want to add attribute support to the project.

Any objects you add following setting this option are designated as attributed by default (the check box is selected). You can clear this box to add an object that does not use attributes.

See Application Settings, ATL Project Wizard and Basic Mechanics of Attributes for more information.

COM

Provides information about the COM functionality for the object.

Coclass

Sets the name of the component class that contains a list of interfaces supported by the object.

Note
If you create your project using attributes, or if you indicate on this wizard page that the object uses attributes, you cannot change this option because ATL does not include the **coclass** attribute.

Type

Sets the object description that will appear in the registry

Interface

Sets the interface you create for your object. This interface contains your custom methods.

ProgID

Sets the name that containers can use instead of the CLSID of the object.

See Also

Tasks

Adding an ATL Simple Object

Options, ATL Simple Object Wizard

Use this page of the ATL Simple Object Wizard to design for increased efficiency and error support for the object.

Note
If you are using this wizard with a device project, see Wizard Options for Options, ATL Simple Object Wizard.

For more information on ATL projects and ATL COM classes, see ATL Reference.

Threading model

Indicates the method for managing threads. By default, the project uses **Apartment** threading.

See Specifying the Project's Threading Model for more information.

Option	Description
Single	Specifies that the object always runs in the primary COM thread. See Single-Threaded Apartments and InprocServer32 for more information.
Apartment	Specifies that the object uses apartment threading. Equivalent to single thread apartment. Each object of an apartment-threaded component is assigned an apartment for its thread, for the life of the object; however, multiple threads can be used for multiple objects. Each apartment is tied to a specific thread and has a Windows message pump (default). See Single-Threaded Apartments for more information.
Both	Specifies that the object can use either apartment or free threading, depending from which kind of a thread it is created.
Free	Specifies that the object uses free threading. Free threading is equivalent to a multithread apartment model. See Multithreaded Apartments for more information.
Neutral (Windows 2000 only)	Specifies that the object follows the guidelines for multithreaded apartments, but it can execute on any kind of thread.

Aggregation

Indicates whether the object uses aggregation. The aggregate object chooses which interfaces to expose to clients, and the interfaces are exposed as if the aggregate object implemented them. Clients of the aggregate object communicate only with the aggregate object.

Option	Description
Yes	Specifies that the object can be aggregated. The default.
No	Specifies that the object is not aggregated.
Only	Specifies that the object must be aggregated.

Interface

Indicates the type of interface the object supports. By default, the object supports a dual interface.

Option	Description

Dual	Specifies that the object supports a dual interface (its vtable has custom interface functions plus late-binding **IDispatch** methods). Allows both COM clients and Automation controllers to access the object. The default.
Custom	Specifies that the object supports a custom interface (its vtable has custom interface functions). A custom interface can be faster than a dual interface, especially across process boundaries. • **Automation compatible** Allows Automation controllers to access an object that has the custom interface support.

Support

Indicates additional support for the object.

Option	Description
ISupportErrorInfo	Creates support for the ISupportErrorInfo interface so the object can return error information to the client.
Connection points	Enables connection points for your object by making your object's class derive from IConnectionPointContainerImpl.
Free-threaded marshaler	Creates a free-threaded marshaler object to marshal interface pointers efficiently between threads in the same process. Available to object specifying **Both** as the threading model.
IObjectWithSite (IE object support)	Implements IObjectWithSiteImpl, which provides a simple way to support communication between an object and its site in a container.

See Also

Tasks

Adding an ATL Simple Object

Reference

ATL Simple Object Wizard

In-Process Server Threading Issues

Adding an Event

From Class View, you can add an event using the Add Event Wizard only to the control class in your MFC ActiveX control project. If you want to add an event to another type of project, use the **Events** button in the Properties window.

To add an event to your MFC ActiveX control project

1. In Class View, expand the project node to display the classes in the project.
2. Right-click the project's control class.
3. On the shortcut menu, click **Add**, and then click **Add Event** to display the Add Event Wizard.
4. Provide the event information in the appropriate wizard boxes.
5. Click **Finish** to add the event to the project.

See Also

Tasks

Adding a Member Variable
Adding a Member Function

Reference

Navigating the Class Structure

Concepts

Adding Functionality with Code Wizards
Adding a Class
Adding an MFC Message Handler

Add Event Wizard

This wizard adds an event to an MFC ActiveX control project. You can specify your own event, you can customize a typically stock event, or you can select from a list of stock events.

Event name

Sets the name used by Automation clients to request an event from the class. Enter a name or select one from the list.

Event type

Indicates the type of event to add. Available only if you select from the **Event Name** list.

Option	Description
Stock	Specifies that a stock event, such as a button click, will be implemented for this class. Stock events are defined in the Microsoft Foundation Class (MFC) Library.
Custom	Specifies that you are providing your own implementation of the event.

Internal name

Sets the name of the member function that sends the event. Available only for custom events. The name is based on **Event name**. You can change the internal name if you want to provide a name different than **Event name**.

Parameter type

Sets the type for the **Parameter name**. Select the type from the list.

Parameter name

Sets the name of a parameter to pass through your event. After typing the name, you must click **Add** to add it the list of parameters.

Once you click **Add**, the parameter name appears in **Parameter list**.

Note
If you supply a parameter name and then click **Finish** before you click **Add**, then the parameter is not added to the event. Y ou must find the method and insert the parameter manually. **Parameter list**

Add

Adds the parameter you specify in **Parameter name**, and its type, to **Parameter list**. You must click **Add** to add a parameter to the list.

Remove

Removes the parameter you select in **Parameter list** from the list.

Parameter list

Displays all parameters and their types currently added for the method. As you add parameters, the wizard updates **Parameter list** to display each parameter with its type.

See Also

Tasks

Adding an Event

Adding an Event Handler

From the resource editor, you can add a new event handler, or edit an existing event handler, for a dialog box control using the Event Handler Wizard.

You can add an event to the class implementing the dialog box using the Properties window. If you want to add the event to a class other than the dialog box class, use the Event Handler Wizard.

To add an event handler to a dialog box control

1. Double-click the dialog box resource in Resource View to open the dialog box resource that contains the control in the dialog editor.
2. Right-click the control for which you want to handle the notification event.
3. On the shortcut menu, click **Add Event Handler** to display the Event Handler Wizard.
4. Select the event in the **Message type** box to add to the class selected in the **Class list** box.
5. Accept the default name in the **Function handler name** box, or provide the name of your choice.
6. Click **Add and edit** to add the event handler to the project and open the text editor at the new function to add the appropriate event handler code.

   If the selected message type already has an event handler for the selected class, **Add and edit** is unavailable, and **Edit code** is available. Click **Edit code** to open the text editor at the existing function.

Alternately, you can add event handlers from the Properties window. See Adding Event Handlers for Dialog Box Controls for more information.

See Also

Tasks

Adding a Member Variable
Adding a Member Function

Reference

Navigating the Class Structure

Concepts

Adding Functionality with Code Wizards
Adding a Class
Adding an MFC Message Handler

Event Handler Wizard

This wizard adds an event handler for a dialog box control to the class of your choice. If you add an event handler from the Properties window, you can add it only to the class that implements the dialog box. See Adding Event Handlers for Dialog Box Controls for more information.

Command name

Identifies the selected control, for which the event handler is added. This box is unavailable.

Message type

Displays the list of current possible message handlers for the selected control.

Function handler name

Displays the name of the function that is added to handle the event. By default, the name is based on the message type and the command, prepended by "On". For example, for the button called `IDC_BUTTON1`, the message type `BN_CLICKED` displays the function handler name `OnBnClickedButton1`.

Class list

Displays available classes to which you can add an event handler. The class for the selected dialog box is displayed in red.

Handler description

Provides a description for the item selected in the **Message type** box. This box is unavailable.

Add and edit

Adds the message handler to the selected class or object, and then opens the text editor to the new function so you can add the control notification handler code.

Edit code

Opens the text editor to the selected existing function so you can add or edit the control notification handler code.

See Also

Tasks

Adding an Event Handler

Adding an MFC Class

To add classes derived from Microsoft Foundation Class (MFC) library classes to your project, use the **Add Class** command available from Class View. Specify the name of the new class, select the base class, and select the ID of the dialog box with which it is associated (if any). The code wizard creates a header file and an implementation file and adds them to your project.

Note
You can add MFC classes to an ATL COM application if you initially created the application with MFC support. You can also a dd MFC classes to Win32 projects that have MFC support.

To add an MFC class to your project

1. From Class View, right-click the project name. Click **Add** and then click **Add Class** to open the Add Class dialog box.
2. In the Templates pane, select **MFC Class**.
3. Define the settings for the new class in the MFC Class Wizard dialog box.
4. Click **Finish** to close the wizard and view the new class in Class View. You can also view the files created by the wizard in Solution Explorer.

See Also

Concepts

Adding Functionality with Code Wizards
Adding a Class
Class Library Overview

MFC Class Wizard

Use this code wizard to add a class to an existing MFC project, or to add a class to an ATL project that supports MFC. You can also add MFC classes to Win32 projects that have MFC support. The features you specified when you created your project determine the options available in this dialog box.

Note
If you are using this wizard with a device project, see Unsupported Wizard Options for MFC Class Wizard.

Names

In this page, specify the class name, the base class, and file names for the new class.

Class name

Specifies the name of the new class and provides the default basis for the names of IDs and files on this page. C++ classes typically start with "C," so, for example, **CMyClass** becomes MyClass.h, and so on.

Base class

Specifies the name of the base class for the new class. By default, the base class is **CWnd**. The base class you select determines whether other boxes on this page are active.

The type of class you set as the base class determines whether the class has a dialog ID or a resource ID. The general types of classes are as follows:

- Classes such as CButton, CWnd, or CDocument, which do not require a dialog ID or resource ID. These classes do not use a dialog or resource ID. If you select one of these classes for your base class, the **Dialog ID** box and the **DHTML resource ID** box are dimmed.
- Classes such as CDialog, CFormView, or CPropertyPage, which require a dialog ID.
- The class CDHtmlDialog, which requires a dialog ID, a DHTML resource ID, and an HTML file name.

For classes requiring a dialog ID, you might find it more efficient to use the Resource editor to create the dialog resource, assign its ID in the Properties window, and then create a class associated with that resource ID. See Creating a New Dialog Box for more information on creating a standard Windows dialog box.

If you create the class first, you can later associate the class to the resource using the Properties window.

Note
If you create a dialog resource first and derive its new class from **CDHtmlDialog**, delete the standard Windows **OK** and **Cancel** buttons that appear on the default dialog box. The standard Windows dialog box hosts the DHTML form, which contains its own **OK** and **Cancel** buttons.

While your dialog box can contain both Windows controls and DHTML controls, it is not recommended.

Dialog ID

Specifies the ID of the dialog, if you selected **CDialog**, **CFormView**, **CPropertyPage**, or **CDHtmlDialog** as the **Base class**.

.h file

Sets the name of the header file for the new object's class. By default, this name is based on the name you provide in **Class name**. Click the ellipsis button to save the file name to the location of your choice, or to append the class declaration to an existing file. If you choose an existing file, the wizard will not save it to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class declaration should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

.cpp file

Sets the name of the implementation file for the new object's class. By default, this name is based on the name you provide in **Short name**. Click the ellipsis button to save the file name to the location of your choice. The file is not saved to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class implementation should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

Active accessibility

Enables MFC's support for Active Accessibility by calling EnableActiveAccessibility in the constructor. This option is available for classes derived from CWnd.

DHTML resource ID

Applies to classes derived from **CDHtmlDialog** only. Specifies the resource ID of the DHTML dialog box. The resource ID appears in the HTML section of the project's .rc file, along with the HTML dialog box file name. The DHTML resource, identified by this ID, is hosted by the dialog box, identified by **Dialog ID**.

.htm file

Applies to classes derived from **CDHtmlDialog** only. Sets the name of the HTML file for the DHTML dialog box. By default, this file name is based on the class name. The file name appears in the HTML section of the project's .rc file, along with the DHTML dialog box resource ID.

Automation

Sets the class level of support for Automation. Automation at the class level is available for all classes that support Automation. It is also available for projects created with support for Automation (That is, either an MFC project that supports ATL, or an MFC project for which you selected the **Automation** check box in the Advanced Features page of the MFC Application Wizard.)

Option	Description
None	Indicates that the class has no Automation support.
Automation	Indicates that the class supports Automation. If you select this option, the newly created class is available as a programmable object by Automation client applications, such as Microsoft Visual Basic and Microsoft Excel. This option is not available for the base classes listed after this table.
Creatable by type ID	Indicates that both the class and project support other applications creating objects of this class using Automation. With this option, automation clients can directly create an Automation object. The type ID in the text box is used by the client application to specify the object to be created; it is systemwide and must be unique. This option is not available for the base classes listed after this table.

Automation support is not available for the following base classes:

- **CAsyncMonitorFile**
- **CAsyncSocket**
- **CCachedDataPathProperty**
- **CConnectionPoint**
- **CDatabase**
- **CDataPathProperty**
- **CHttpFilter**
- **CHttpServer**
- **CInternetSession**
- **CObject**
- **CSocket**

Type ID

Sets the type ID of the class. The **Type ID** box concatenates the project name and the new class name as follows: *MFCProj.MFCClass*. This ID is changeable only if you selected the **Automation** option **Creatable by type ID**.

Generate DocTemplate resources

Indicates that the documents created by the application have document template resources. To activate this check box, the project must support the MFC document/view architecture, and the base class of this class must be CFormView.

See Document Templates and the Document/View Creation Process for more information.

See Also

Tasks

Adding an MFC Class

Concepts

Adding a Class

Document Template Strings, MFC Class Wizard

This page of the wizard is available only for classes meeting the following criteria:

- The MFC project supports the document/view architecture.
- The base class of the new class is CFormView.
- The check box **Generate DocTemplate resources** is checked on the **Names** section of the MFC Class Wizard.

The wizard provides defaults for the following values to help with forms view design, management, and localization. Because most document template strings are visible and used by the form's users, they are localized into the **Resource language** indicated in the Application Types page of the MFC Application Wizard when the project was created.

Note
The wizard does not provide automatic printing support for classes derived from **CFormView**.

See Document Templates and the Document/View Creation Process for more information.

Nonlocalized strings

Applies to applications that create user documents. Users can open and save documents more easily if the document type has a file extension and a file type ID. These items are not localized because they are used by the system rather than by the user.

File extension

Sets the file extension associated with the document type for this forms application. The file extension default based on the class name. For example, if the new MFC class is named **CWidget**, by default, the file extension is .wid. The file extension is used in file filters and the **Open** and **Save as** dialog boxes.

If you change the file extension, the change is reflected in the **Filter name** box.

Note
If you change the default file extension, do not include the period.

File type ID

Sets the label for your document type in the system registry.

Localized strings

Produces strings associated with the forms and documents that are read and used by the application's users, so the strings are localized.

Doc type name

Identifies the type of document under which a document of the application can be grouped. By default, it is based on the name of the class. For example, if the new MFC class is named **CWidget**, by default, the document type name is Widget. Changing the default does not change any other options in this dialog box.

Filter name

Sets the name that users can indicate to find files of the specified file type. This option is available from the **Files of type** and **Save as type** options in the standard Windows **Open** and **Save as** dialog boxes. By default, the name is based on the project name plus Files, followed by the extension indicated in **File Extension**. For example, if your project is named Widget, and the file extension is .wid, the **Filter name** is Widget Files (*.wid) by default.

File new short name

Sets the name appearing in the standard Windows **New** dialog box, if the project has more than one document template. If your application is an Automation server, this name is used as the short name of your Automation object. By default, this name is based on the class name.

File type long name

Sets the file type name in the system registry. If your application is an Automation server, this name is used as the long name of your Automation object. By default, this name is based on the class name plus .Document. For example, if the class name is **CWidget**, the **File type long name** is Widget Document.

Document class

Indicates the project's document class. By default, this class is the main application's document class, as listed in the Review Generated Classes page of the MFC Application Wizard. You can select another document class from the list, if you have added other document classes in the project.

See Also

Tasks

Adding an MFC Class

Reference

MFC Class Wizard

Concepts

Adding a Class

Adding an MFC Class from a Type Library

Use this wizard to create an MFC class from an interface in an available type library. You can add an MFC class to an MFC application, an MFC DLL, or an MFC ActiveX control.

Note
You do not need to create your MFC project with Automation enabled to add a class from a type library.

A type library contains a binary description of the interfaces exposed by a component, defining the methods along with their parameters and return types. Your type library must be registered for it to appear in the **Available type libraries** list in the Add Class from Typelib Wizard. See Inside Distributed COM: Type Libraries and Language Integration in the MSDN library for more information.

To add an MFC class from a type library

1. In either Solution Explorer or Class View, right-click the name of the project to which you want to add the class.
2. From the shortcut menu, click **Add**, and then click **Add Class**.
3. In the Add Class dialog box, in the Templates pane, click **MFC Class from Typelib**, and then click **Open** to display the Add Class from Typelib Wizard.

In the wizard, you can add more than one class in a type library. Likewise, you can add classes from more than one type library in a single wizard session.

The wizard creates an MFC class, derived from COleDispatchDriver, for each interface you add from the selected type library. **COleDispatchDriver** implements the client side of OLE automation.

See Also

Tasks

Automation Clients: Using Type Libraries

Concepts

Automation Clients

Add Class from Typelib Wizard

Use this wizard to add an MFC class from an available type library. The wizard creates a class for each interface you add from the selected type library.

Add class from

Specifies the location of the type library, from which the class is created.

Option	Description
Registry	The type library is registered in the system. Registered type libraries are listed in **Available type libraries**.
File	The type library is not necessarily registered in the system but is contained in a file. You must provide the file location in **Location**.

Available type libraries

Lists the type libraries currently registered in the system. Select a type library from this list to display its interfaces in the **Interfaces** list.

See Inside Distributed COM: Type Libraries and Language Integration in the MSDN library for more information about registering type libraries.

Location

Specifies the location of the type library. If you click **File** under **Add Class From**, you can provide the location of the file containing the type library. To browse to the location of the file, click the ellipsis button.

Interfaces

Lists the interfaces in the type library currently selected in the **Available type libraries** list.

Transfer button	Description
>	Adds the interface currently selected in the **Interfaces** list. Dimmed if no interface is selected.
>>	Adds all the interfaces in the type library currently selected in the **Available type libraries** list.
<	Removes the class currently selected in the **Generated classes** list. Dimmed if no class is currently selected in the **Generated classes** list.
<<	Removes all the classes in the **Generated classes** list. Dimmed if the **Generated classes** list is empty.

Generated classes

Specifies the class names to be generated from the interfaces added using the **>** or **>>** button. You can click this box to select a class, and then use the up or down keys to scroll through the list, viewing each class name in the **Class** box and file name in the **File** box that the wizard generates when you click **Finish**. You can select only one class at a time in this box.

You can remove a class by selecting it in this list and clicking **<**. You do not need to select a class in the Generated classes box to remove all classes; by clicking **<<**, you remove all classes in the **Generated classes** box.

Class

Specifies the name of the class selected in the **Generated classes** box that the wizard adds when you click **Finish**. You can edit the name in the **Class** box.

File

Sets the name of the header file for the new class. By default, this name is based on the name you provide in **Generated classes**. Click the ellipsis button to save the file name to the location of your choice, or to append the class declaration to an existing file. If you choose an existing file, the wizard will not save it to the selected location until you click **Finish** in the wizard.

The wizard does not overwrite a file. If you select the name of an existing file, when you click **Finish**, the wizard prompts you to indicate whether the class declaration should be appended to the contents of the file. Click **Yes** to append the file; click **No** to return to the wizard and specify another file name.

See Also

Tasks

Adding an MFC Class from a Type Library
Automation Clients: Using Type Libraries

Adding an MFC Message Handler

You can use the Properties window to add a message handler (a member function that handles Windows messages) to a class and map Windows messages to the message handler. You can also add an event handler for any dialog box control.

By using the Properties window to define message- and event-handling functions, you can automatically update the message-dispatch table (or message map) and your class header file.

Note
You can add a message handler to an ATL class using the Properties window; however, some results may vary. For more info rmation, see the ATL topic Adding an ATL Message Handler.

See Also

Tasks

Adding a Member Function
Adding a Member Variable
Overriding a Virtual Function

Reference

Navigating the Class Structure
Dialog Editor

Concepts

Adding Functionality with Code Wizards
Adding a Class

Mapping Messages to Functions

The Properties window enables you to bind message handlers (member functions of MFC user-interface classes) to the messages generated by your application's resources. They use MFC message maps to create the binding.

When you use Class View to create a new class derived from one of the framework classes, it automatically places a complete and functional class in the header (.h) and implementation (.cpp) files that you specify.

Note
To add a new class that does not handle messages, create the class directly in the text editor.

To define or remove a message handler using the Properties window

1. In Class View, click the class.
2. In the Properties window, click the **Messages** button.

Note
The **Messages** button is available when you select either the class name in Class View or when you click within the source window.

   If your project has a handler for a message, then the name of the handler appears in the right column next to the message.
3. If the message has no handler, then click the cell in the right column in the Properties window to display the suggested name of the handler as <add>*HandlerName*. (For example, the **WM_TIMER** message handler suggests <add>**OnTimer**).
4. Click the suggested name to add stub code for the function.
5. To edit a message handler, double-click the message in Class View and edit the code in the source window.

To remove a message handler, double-click the handler in the right column and select <delete>*HandlerName*. The function's code is commented out.

See Also

Tasks

Adding a Member Function
Adding a Member Variable
Overriding a Virtual Function

Reference

Adding Event Handlers for Dialog Box Controls
Navigating the Class Structure

Concepts

Adding an MFC Message Handler
Adding Functionality with Code Wizards
Adding a Class

Message Types Associated with User-Interface Objects

The following table shows the types of objects with which you work, and the types of messages associated with them.

User Interface Objects and Associated Messages

Object ID	Messages
Class name, representing the containing window	Windows messages appropriate to a CWnd-derived class: a dialog box, window, child window, MDI child window, or topmost frame window.
Menu or accelerator identifier	• **COMMAND** message (executes the program function). • **UPDATE_COMMAND_UI** message (dynamically updates the menu item).
Control identifier	Control notification messages for the selected control type.

See Also

Tasks

Mapping Messages to Functions
Adding a Member Function
Adding a Member Variable
Overriding a Virtual Function

Reference

Navigating the Class Structure

Concepts

Adding Functionality with Code Wizards
Adding a Class
Adding an MFC Message Handler

Editing a Message Handler

Once you have defined a message handler, you can go to the member function's definition to add or modify code.

To jump to a member function definition from the dialog editor, double-click a control for which a handler is already defined. This navigates you to the file in which the selected control's message handler is defined.

To jump to a member function definition from Class View, double-click the function name in Class View.

See Also

Tasks

Mapping Messages to Functions
Adding a Member Function
Adding a Member Variable
Overriding a Virtual Function

Reference

Navigating the Class Structure

Concepts

Adding Functionality with Code Wizards
Adding a Class
Adding an MFC Message Handler

Defining a Message Handler for a Reflected Message

Once you have created a new MFC control class, you can define message handlers for it. Reflected message handlers allow your control class to handle its own messages before the message is received by the parent. You can use the MFC CWnd::SendMessage function to send messages from your control to a parent window.

With this functionality you could, for example, create a list box that will redraw itself rather than relying on the parent window to do so (owner drawn). For more information on reflected messages, see Handling Reflected Messages.

To create an ActiveX control with the same functionality, you must create a project for the ActiveX control.

Note
You cannot add a reflected message (OCM_*Message*) for an ActiveX control using the Properties window, as described below. You must add these messages manually.

To define a message handler for a reflected message from the Properties window

1. Add a control, such as a list, a rebar control, a toolbar, or a tree control, to your MFC project.
2. In Class View, click the name of your control class.
3. In the Properties window, the control class name appears in the **Class Name** list.
4. Click the **Messages** button to display the Windows messages available to add to the control.
5. Scroll down the list of messages in the Properties window until you see the heading **Reflected**. Alternately, click the **Categories** button and collapse the view to see the **Reflected** heading.
6. Select the reflected message for which you want to define a handler. Reflected messages are marked with an equal sign (=).
7. Click the cell in the right column in the Properties window to display the suggested name of the handler as <add>*HandlerName*. (For example, the **=WM_CTLCOLOR** message handler suggests <add>**CtlColor**).
8. Click the suggested name to accept. The handler is added to your project.

   Message handler names that you have added appear in the right column of the reflected messages window.
9. To edit or delete a message handler, repeat steps 4 through 7. Click the cell containing the handler name to edit or delete and click the appropriate task.

See Also

Tasks

Mapping Messages to Functions
Adding a Member Function
Adding a Member Variable
Overriding a Virtual Function

Reference

Navigating the Class Structure

Concepts

Adding Functionality with Code Wizards
Adding a Class
Adding an MFC Message Handler

Declaring a Variable Based on Your New Control Class

Once you have created an MFC control class, you can declare a variable based on it. To provide a context for the new variable, you must open the dialog editor and edit the dialog box in which you want to use your reusable control. Also, the dialog box must already have a class associated with it. For information on using the dialog editor, see Dialog Editor.

To declare a variable based on your reusable class

1. While editing the dialog box, drag a control of the same type as the base class of your new control from the Controls toolbar onto the dialog box.
2. Place the mouse pointer over the dropped control.
3. While pressing the CTRL key, double-click the control.

   The Add Member Variable dialog box appears.
4. In the **Access** box, select the correct access for your control.
5. Click the **Control variable** check box.
6. In the **Variable name** box, type a name.
7. Under **Category**, click **Control**.
8. In the **Control ID** list, pick the control that you added. The **Variable type** list should display the correct variable type, and the **Control type** box should display the correct control type.
9. In the **Comment** box, add any comment you want to appear in your code.
10. Click **OK**.

See Also

Tasks

Mapping Messages to Functions
Adding a Member Function
Adding a Member Variable
Overriding a Virtual Function

Reference

Navigating the Class Structure

Concepts

Adding Functionality with Code Wizards
Adding a Class
Adding an MFC Message Handler

Adding an MFC ODBC Consumer

An MFC ODBC consumer consists of an ODBC recordset class and data bindings necessary to access a data source.

To add an MFC ODBC consumer

1. In Class View, right-click the project. On the shortcut menu, click **Add** and then click **Add Class**.
2. In the Visual C++ folder, double-click the **MFC ODBC Consumer** icon or select it and click **Open**.

   The MFC ODBC Consumer Wizard opens.
3. Define settings as described in MFC ODBC Consumer Wizard.
4. Click **Finish** to close the wizard, which will insert the newly created ODBC consumer code in your project.

See Also

Concepts

Adding Functionality with Code Wizards

MFC ODBC Consumer Wizard

Insert "Search Results" summary here.

This wizard sets up an ODBC recordset class and the data bindings necessary to access the specified data source.

Data Source

The **Data Source** button lets you set up the specified data source using the specified ODBC driver. For more information about data source files (DSN), see File Data Sources in the ODBC SDK. The **Select Data Source** dialog box has two tabs:

- **File Data Source** tab: The **Look in** box specifies the directory in which to select files to be used as data sources. The default is \Program Files\Common Files\ODBC\Data Sources. The existing file data sources (.dsn files) appear in the main list box. You can either set up the data sources ahead of time using the **File DSN** tab on the ODBC Data Source Administrator, or create new ones using this dialog box.

  To create a new file data source from this dialog box, click **New** to specify a DSN name; the **Create New Data Source** dialog box appears. In the **Create New Data Source** dialog box, select an appropriate driver and click **Next**; click **Browse**, and select the name of the file to be used as a data source (you have to select "All Files" to view non-DSN files, such as .xls files); click **Next**, and then click **Finish**. (If you selected a non-DSN file, you will get a driver-specific dialog box, such as "ODBC Microsoft Excel Setup," which will convert the file to a DSN.)

Note
You can also create a new file data source beforehand using the ODBC Data Source Administrator. From the **Start** menu, select **Settings**, **Control Panel**, **Administrative Tools**, **Data Sources (ODBC)**, and then **ODBC Data Source Administrator**.

  The **DSN Name** box allows you to specify a name for the file data source. You must ensure that the DSN name ends with the appropriate file extension, such as .xls for Excel files or .mdb for Access files.

  For more information on DSNs, see File Data Sources in the ODBC SDK.

- **Machine Data Source** tab: This tab lists system and User DATA sources. User data sources are specific to a user on this machine. System data sources can be used by all users on this machine or on a systemwide service. See Machine Data Sources in the ODBC SDK

For more information on ODBC data sources, see Data Sources in the ODBC SDK.

Click **OK** to finish. The **Select Database Object** dialog box appears. From this dialog box, select the table or view that the consumer will use. Note that you can select multiple views and tables by holding the control key while clicking on the items.

Class

The name of the consumer class, based by default on the name of the file or machine data source that you selected.

.h file

The name of the consumer class header file, based by default on the name of the file or machine data source that you selected.

.cpp file

The name of the consumer class implementation file, based by default on the name of the file or machine data source that you selected.

Type

Specifies whether the recordset is a dynaset (default) or a snapshot.

- **Dynaset**: Specifies that the recordset is a dynaset. A dynaset is the result of a query that provides an indexed view into the queried database's data. A dynaset caches only an integral index to the original data and thus offers a performance gain over a snapshot. The index points directly to each record found as a result of a query and indicates if a record is removed. You also have access to updated information in the queried records. This is the default.
- **Snapshot**: Specifies that the recordset is a snapshot. A snapshot is the result of a query and is a view into a database at

one point in time. All records found as a result of the query are cached, so you do not see any changes to the original records.

Bind all columns

Specifies whether all columns in the selected table are bound. If you select this box (default), all columns are bound; if you do not select this box, no columns are bound, and you must bind them manually in the recordset class.

See Also

Tasks

Adding an MFC ODBC Consumer

Concepts

Adding Functionality with Code Wizards

Adding ATL Support to Your MFC Project

If you have already created an MFC-based application, then you can add support for the Active Template Library (ATL) easily by running the Add ATL Support to MFC Project Wizard.

Note
This support applies only to simple COM objects added to an MFC executable or DLL project. You can add other COM objects (including ActiveX controls) to MFC projects, but the objects might not operate as expected.

To add ATL support to your MFC project

1. In Solution Explorer, right-click the project to which you want to add ATL support.
2. On the shortcut menu, click **Add**, and then click **Add Class**.
3. Select the **Add ATL Support to MFC Project** icon.

Note
This icon is located in the ATL folder in the **Categories** pane.

4. When prompted, click **Yes** to add ATL support.

For more information about how adding ATL support changes your MFC project's code, see Details of ATL Support Added by the ATL Wizard

See Also

Tasks

Adding a Member Function
Adding a Member Variable
Overriding a Virtual Function

Reference

Navigating the Class Structure

Concepts

Adding a Class
Adding Functionality with Code Wizards
Adding an MFC Message Handler

Details of ATL Support Added by the ATL Wizard

When you add ATL support to an existing MFC executable or DLL, Visual C++ makes the following modifications to the existing MFC project (in this example, the project is called `MFCEXE`):

- Two new files (an .idl file and an .rgs file, used to register the server) are added.
- In the main application header and implementation files (Mfcexe.h and Mfcexe.cpp), a new class (derived from **CAtlMFCModule**) is added. In addition to the new class, code is added to **InitInstance** for registration. Code is also added to the **ExitInstance** function for revoking the class object. In the header file, Finally, two new header files (Initguid.h and Mfcexe_i.c) are included in the implementation file, declaring and initializing the new GUIDs for the **CAtlMFCModule**-derived class.
- To register the server properly, an entry for the new .rgs file is added to the project's resource file.

Notes for DLL Projects

When you add ATL support to an MFC DLL project, you will see some differences. Code is added to the **DLLRegisterServer** and **DLLUnregisterServer** functions for registering and unregistering the DLL. Code is also added to DllCanUnloadNow and DllGetClassObject.

See Also

Tasks

Adding ATL Support to Your MFC Project
Adding a Member Function
Adding a Member Variable
Overriding a Virtual Function

Reference

Navigating the Class Structure

Concepts

Adding Functionality with Code Wizards
Adding a Class
Adding an MFC Message Handler

Creating a COM Interface

Visual C++ provides wizards and templates to create projects that use COM defining interfaces and dispinterfaces for your COM objects and automation classes.

You can use these wizards to perform the following three common tasks:

- Adding ATL Support to Your MFC Project

  Add ATL support to an MFC application after you create an MFC project using the MFC Application Wizard and then running the **Add ATL Support to MFC** code wizard. This support applies only to simple COM objects added to an MFC executable or DLL project. These ATL objects may have multiple interfaces.

- Creating an MFC ActiveX Control

  Open the MFC ActiveX Control Wizard to create an ActiveX control with a dispinterface and an event map defined in the .idl file and the control class, respectively.

- Adding an ATL Control

  Use a combination of the ATL Project Wizard and the ATL Control Wizard to create an ATL ActiveX control.

  You can also add an ATL control to an MFC project to which you have added ATL support, as described above. Additionally, if you select **ATL Control** in the **Add Class** dialog box, and you have not yet added ATL support to your MFC project, Visual Studio displays a dialog box confirming adding ATL support to your MFC project.

  This wizard generates IDL source and a COM map in the project classes.

Once you have an ATL project open, the Add Class dialog box gives you the choice of additional wizards and templates to add COM interfaces to your project. The following wizards allow you to establish one or more interfaces for the object:

- ATL COM+ 1.0 Component Wizard
- ATL Simple Object Wizard
- ATL Active Server Page Component Wizard
- ATL Control Wizard

Additionally, you can implement new interfaces on your COM control by right-clicking the object's control class in Class View and clicking Implement Interface.

Note

Visual Studio does not provide a wizard to add an interface to a project. You can add an interface to an ATL project or to an Adding ATL Support to Your MFC Project by adding a simple object using the ATL Simple Object Wizard. Alternately, open th e project's .idl file and create the interface by typing:

```
interface IMyInterface {
};
```

See Implementing an Interface and Adding Objects and Controls to an ATL Project for more information.

Visual C++ provides several ways to view and edit the COM interfaces defined for your projects. Class View displays icons for any interface or dispinterface defined in an .idl file in your C++ project.

For ATL-based COM object classes, Class View reads the COM map in the ATL class to display the relationship between the ATL class and any interfaces it implements.

In Class View and its shortcut menus, you can work with interfaces as follows:

- Add ATL objects to an MFC-based application.
- Add methods, properties, and events.

- Jump directly to an item's interface code by double-clicking the item.

See Also

Tasks

Creating Projects with Application Wizards

Concepts

Adding Functionality with Code Wizards

Editing a COM Interface

By using commands from the Class View shortcut menu, you can define new methods and properties for the COM interfaces in your Visual C++ projects. In addition, from the Toolbox, you can define events for ActiveX controls.

For ATL- and MFC-based COM object classes, you can edit the class implementation at the same time that you edit the interface.

Note
For interfaces that you have defined outside of the **Add Class** dialog box, Visual C++ adds the methods or properties to the .idl file, and it add stubs to the classes that implement methods, even when the interfaces are added manually.

The following three wizards help you customize existing interfaces. They are available from Class View:

Wizard	Project type
Add Property Wizard	ATL or MFC projects supporting ATL. Right-click the interface to which you want to add the property. Visual C++ detects the project type and modifies the options in the Add Property Wizard accordingly: • For dispinterfaces in projects created by using the MFC Application Wizard, invoking the Add Property Wizard provides options specific to MFC. • For MFC ActiveX control interfaces, the Add Property Wizard provides a list of stock methods and properties that you can use as provided or customize for your control. • For all other interfaces, the Add Property Wizards provide options useful in most situations.
Add Method Wizard	ATL or MFC projects supporting ATL. Right-click the interface to which you want to add the method. Visual C++ detects the project type and modifies the options in the Add Method Wizard accordingly: • For dispinterfaces in projects created by using the MFC Application Wizard, invoking the Add Method Wizard provides options specific to MFC. • For MFC ActiveX control interfaces, the Add Method Wizard provides a list of stock methods and properties that you can use as provided or customize for your control. • For all other interfaces, the **Add Method** wizards provide options useful in most situations.

Additionally, you can implement new interfaces on your COM control by right-clicking the object's control class in Class View and clicking Implement Interface.

See Also

Concepts

Adding Functionality with Code Wizards

Visual C++ Projects

Other Resources

Working with Resource Files

Implementing a Connection Point

To implement a connection point using the Implement Connection Point Wizard, you must have created a project as an ATL COM application or as an MFC application that contains ATL support. You can use the ATL Project Wizard to create an ATL application, or add an ATL object to your MFC application to implement ATL support for an MFC application.

Note
For information about implementing connection points for an MFC project, see Connection Points.

Once you create the project, to implement a connection point, you must first add an ATL object. See Adding Objects and Controls to an ATL Project for a list of wizards that add objects to your ATL project.

Note
The wizard does not support ATL dialogs, XML Web services created with ATL Server, performance objects, or performance c ounters.

A connectable object (that is, a source) can expose a connection point for each of its outgoing interfaces. Each outgoing interface can be implemented by a client on an object (that is, a sink). For more information, see ATL Connection Points.

To implement a connection point

1. In Class View, right-click the class name for your ATL object.
2. Click **Add** from the shortcut menu, and then click **Add Connection Point** to display the Implement Connection Point Wizard.
3. Select the connection point interfaces to implement from the appropriate type libraries and click **Finish**.
4. In Class View, examine the proxy classes created for each connection point. The classes appear as CProxy*InterfaceName*<T> and are derived from IConnectionPointImpl.
5. Double-click the connection point class to display the definition of the connection point's class.
   - If you implement a connection point for your own project's interface, the following definition appears

     ```
     template< class T >
     class CProxyInterfaceName :
        public IConnectionPointImpl< T, &IID_InterfaceName >
     {
     public:
     };
     ```

     If you implement a local interface, methods and properties appear in the class body.
   - If you implement a connection point for another interface, the definition includes the interface's methods, each preceded by `Fire_`.

See Also

Tasks

How to: Create and Implement Interfaces
Adding Connection Points to an Object

Concepts

Adding Functionality with Code Wizards

Implement Connection Point Wizard

This wizard implements a connection point for a COM object. A connectable object (that is, a source) can expose a connection point for its own interfaces or for any outgoing interface. Visual C++ and Windows both provide type libraries that have outgoing interfaces. Each outgoing interface can be implemented by a client on an object (that is, a sink).

For more information, see ATL Connection Points.

Available type libraries

Displays the available type libraries containing the interface definitions for which you can implement connection points. Click the ellipsis button to locate a file containing the type library to use.

Location

Displays the location of the type library currently selected in the **Available type libraries** list.

Interfaces

Displays the interfaces whose definitions are contained in the type library currently selected in the **Available type libraries** box.

Transfer button	Description
>	Adds to the **Implement connection points** list the interface name currently selected in the **Interfaces** list.
>>	Adds to the **Implement connection points** list all interface names available in the **Interfaces** list.
<	Removes the interface name currently selected in the **Implement connection points** list.
<<	Removes all interface names currently listed in the **Implement connection points** list.

Implement connection points

Displays the names of the interfaces for which you implement connection points when you click **Finish**.

See Also

Tasks

Implementing a Connection Point

Implementing an Interface

To implement an interface, you must have created a project as an ATL COM application or as an MFC application that contains ATL support. You can use the ATL Project Wizard to create an ATL application, or add an ATL object to your MFC application to implement ATL support for an MFC application.

Once you create the project, to implement an interface, you must first add an ATL object. See Adding Objects and Controls to an ATL Project for a list of wizards that add objects to your ATL project.

Note
The wizard does not support ATL dialogs, XML Web services using ATL, performance objects, or performance counters.

If you add an ATL control, you can specify whether to implement default interfaces, listed on the Interfaces page of that wizard and defined in atlcom.h.

Once you have added the object or control, you can implement other interfaces, located in any available type library, using the Implement Interface Wizard.

If you are adding a new interface, you must add it manually to the project's .idl file. See Adding a New Interface in an ATL Project for more information.

To implement an interface

1. In Class View, right-click the class name for your ATL object.
2. Click **Add** from the shortcut menu, and then click **Implement Interface** to display the Implement Interface Wizard.
3. Select the interfaces to implement from the appropriate type libraries and click **Finish**.
4. In Class View, expand the object's Bases and Interfaces node to see the interface you have implemented, and then expand the interface's node to see its available properties, methods, and events.

Note
You can also use the Object Browser to examine the members of the interface.

See Also

Concepts

Creating a COM Interface
Editing a COM Interface

Implement Interface Wizard

This wizard implements an interface for a COM object. Implementations of many interfaces are included in the COM libraries available with Visual Studio and Windows. An interface implementation is associated with an object when an instance of that object is created, and it provides the services that the object offers.

For a discussion of interfaces and implementations, see Interfaces and Interface Implementations in the *Platform SDK*.

Implement interface from

Specifies the location of the type library, from which the interface is created.

Option	Description
Project	The type library is part of the project.
Registry	The type library is registered in the system. Registered type libraries are listed in **Available type libraries**.
File	The type library is not necessarily registered in the system but is contained in a file. You must provide the file location in **Location**.

Available type libraries

Displays the available type libraries containing the interface definitions that you can implement. If you click **File** under **Implement interface from**, this box is unavailable for change.

Location

Displays the location of the type library currently selected in the **Available type libraries** list. If you selected **File** under **Implement interface from**, click the ellipsis button to locate a file containing the type library to use.

Interfaces

Displays the interfaces whose definitions are contained in the type library currently selected in the **Available type libraries** box.

Note
Interfaces that have the same name as those already implemented by the selected object are not displayed in the **Interfaces** box.

Transfer button	Description
>	Adds to the **Implement interfaces** list the interface name currently selected in the **Interfaces** list.
>>	Adds to the **Implement interfaces** list all interface names available in the **Interfaces** list.
<	Removes the interface name currently selected in the **Implement interfaces** list.
<<	Removes all interface names currently listed in the **Implement interfaces** list.

Implement Interfaces

Displays the names of the interfaces that you have selected to implement on your object.

Note
If you include more than one interface that derives from **IDispatch**, or if you try to implement an interface that is derived from another interface already on your class, then you must disambiguate the COM_MAP entries. See COM_INTERFACE_ENTRY2 for more information.

See Also

Tasks

Implementing an Interface

Navigating the Class Structure

You can navigate a project class structure, including functions, inheritance and interfaces using either Class View or the Object Browser.

You can go to a function declaration by double-clicking on the function name in Class View. You can go to a class definition by double-clicking on a class name in Class View.

See Also

Tasks

Adding a Member Function
Adding a Member Variable
Overriding a Virtual Function

Concepts

Adding Functionality with Code Wizards
Adding a Class
Adding an MFC Message Handler

Overriding a Virtual Function

You can override virtual functions defined in a base class from the Visual Studio Properties window.

To override a virtual function in the Properties window

1. In Class View, click the class.
2. In the Properties window, click the **Overrides** button.

Note
The **Overrides** button is available when you select either the class name in Class View or when you click within the source window.

   The left column lists the virtual functions. If the name of a virtual function also appears in the right column, then an override has already been implemented.
3. If the function has no override, then click the cell in the right column in the Properties window to display the suggested name of the function override as <add>*FuncName*.
4. Click the suggested name to add stub code for the function.
5. To edit an overriding function, double-click the name of the function in Class View and edit the code in the source window.

To remove an override, click the override function name in the right column and select <delete>*FuncName*. The function's code is commented out.

See Also

Tasks

Adding a Member Function
Adding a Member Variable

Reference

Navigating the Class Structure

Concepts

Adding Functionality with Code Wizards
Adding a Class
Adding an MFC Message Handler

Designing a Wizard

You might need to create projects with standardized features that differ from the application wizards that are provided in Visual C++. For such tasks, you can use the Visual C++ wizard architecture, which is designed for easy extensibility and customization. You can create a wizard using the Visual C++ Custom Wizard. After you create your wizard, you can configure it to generate the starter files you need for your projects.

A Visual C++ wizard is an HTML control. It uses the Visual C++ wizard engine, IVCWizCtlUI, which provides common services, such as NavigateToCommandHandler, OkCancelAlert, and so on. The wizard also uses the script engine, which allows a wizard to understand both VBScript and JScript.

The wizard architecture allows you to access the following object models directly in your wizards, and call their methods, properties, and events from either the JScript or the HTML files. (See the examples in The HTML Files and The JScript File for more information.)

- Developer Tools Environment (DTE) Object Model
- Visual C++ Code Model
- Visual C++ Project Model
- Visual C++ Wizard Model

See Files Created for Your Wizard for a description of each element of designing a wizard.

See Also

Tasks

Creating a Custom Wizard

Concepts

Steps to Designing a Wizard

Customizing Your Wizard

Other Resources

Custom Wizard Samples

Steps to Designing a Wizard

You can use a wizard to create and configure common project starter files. Like any project, creating a wizard takes planning. The following steps describe one way to familiarize yourself with the Visual C++ Custom Wizard and apply it to your own projects.

1. Examine the custom wizard samples included with Visual Studio.
2. Lay the groundwork for the type of project the wizard should create. Like all application construction, this process can go through many hands and many different iterations.
3. Create your project with the Visual C++ Custom Wizard, specifying user interface and page number options.

   Note

   If you indicate no user interface (that is, if you clear **User interface** in Application Settings, Custom Wizard in the Custom Wizard), your wizard sets the custom parameter WIZARD_UI=**FALSE** and creates project template files with no user input and no .htm files. As a result, you do not specify page numbers. See The .vsz File (Project Control) for more information.

4. Examine the basic project that the Custom Wizard created for you.
5. If your wizard has a user interface, run the wizard to learn more about the mechanics of the custom wizard.
6. Customize the basic wizard.
7. Add context-sensitive help.
8. Provide error handling for the JScript and HTML code.
9. Build and test the wizard.
10. Debug your wizard. See Debugging Script and Web Applications for more information.

    Note

    When you are debugging JScript, you cannot perform mixed-mode debugging with native code.

See Also

Tasks

Creating a Custom Wizard

Reference

Custom Wizard

Concepts

Files Created for Your Wizard

Other Resources

Custom Wizard Samples

Examining the Basic Wizard Project

After you have used the Custom Wizard to create the basic project, examine the files it has created for you.

1. In Solution Explorer, expand the project and examine the files the wizard created for your project.
2. Double-click Default.js to open the project's JScript file in the code editor. This file contains JScript functions that create the project when the user clicks **Finish** in the wizard. Review the functions and TODO comments in this file.
3. If your project has a user interface, look in the folder labeled HTML Files and note that you have as many .htm files you as you specified in the Application Settings page in the Custom Wizard. Double-click Default.htm to open the project's main HTML page in the HTML designer. You can view the HTML in either design view or in HTML view, as HTML markup. Switch to HTML markup view and review the HTML markup. Click the **Script Only View** button (located at the upper right corner of the HTML View editing window, next to the **Events** drop-down list) and examine the JScript in the .htm file. By default, this file contains the JScript functions that initialize and load the wizard, and it provides default behavior for the **IVCWizCtrlUI** interface. See the coclass VCWizCtl object for more information.
4. Save and close your wizard project.
5. Open the Visual Studio New Project dialog box and find your wizard icon. Double-click the icon to display your wizard. You can examine the basic wizard pages that the Custom Wizard created for you. Note that the first page contains some sample HTML controls and the standard **Finish**, **Cancel**, and **Help** buttons.
6. Click **Finish** to build a new project with your wizard. By default, your wizard creates two text files: ReadMe.txt and Sample.txt. These files describe the project that your wizard created. Close this project and reopen your wizard project.
7. Read Examining the Mechanics of a Wizard to get a clearer understanding of how the Visual Studio environment and the Visual C++ wizard engine created the project you created when you ran your wizard.

You are ready to start customizing your custom wizard.

See Also

Tasks

Creating a Custom Wizard

Reference

Custom Wizard

Providing Context-Sensitive Help

Concepts

Steps to Designing a Wizard

Files Created for Your Wizard

Customizing Your Wizard

Other Resources

Custom Wizard Samples

Examining the Mechanics of a Wizard

You do not need to compile a wizard project to have users start using it right away. Once you have created the necessary elements, the VSDIR directs the New Project dialog box to display the wizard icon and the Add New Item dialog box to display the wizard name in the shortcut menu. Your customer can launch the wizard immediately by selecting it.

When the user launches the wizard, the environment shell cocreates the wizard engine and queries for IDTWizard. It then calls Execute to launch the wizard.

Note
If the wizard has no interface, the project is created with the supplied defaults and displayed in Solution Explorer, with the no de structure supplied in the .vsz file. The rest of this topic assumes that the wizard has a UI.

If the wizard has a UI, the user accepts or changes the defaults in each control in the wizard's HTML-based UI. As the user navigates through the wizard's pages and makes changes, functions such as Navigate and Next are called in the Script section of the HTML.

Whenever the user selects different options within the wizard, the selections are captured in the symbol table in the wizard control. The symbol table matches the IDs of the controls in the wizard's HTML page to maintain a one-to-one correspondence between user selections and the symbol table.

When the user clicks **Finish** in the wizard UI, the JScript function **OnFinish** is called from the HTML script.

Note
You can customize **OnFinish** in Default.js to perform any additional tasks you require.

The wizard engine then scans through the template files, parsing and rendering them based on the user's choices. It copies the rendered files to the project directory and adds these files to the project. The newly created project is loaded in the Visual Studio environment, and the project's nodes and files are displayed in Solution Explorer.

See Also

Tasks

Creating a Custom Wizard

Reference

VCWizCtl

Concepts

Steps to Designing a Wizard

Customizing Your Wizard

Other Resources

Custom Wizard Samples

Customizing Your Wizard

You must consider the following common tasks as you customize the wizard you created with the Custom Wizard.

- In the .vsz file, specify any custom parameters necessary for your wizard to work. See The .vsz File (Project Control) and Predefined Custom Wizard Symbols for more information.

  If you localize your wizard to several different languages, add those language parameters to the .vsz file. See Localizing a Wizard to Multiple Languages for more information.

- Customize the template files (and Templates.inf) to specify the directives for user selections.
- Customize the Default.js file to specify additional special handling for your wizard. You can write your own functions, and you can use functions provided in Common.js.
- Design icons and other images that your HTML user interface will use.
- Design the HTML user interface.
- Add symbols to the HTML symbol table to match the buttons, controls, text boxes, and other elements that your wizard uses.

  The following shows an excerpt of HTML provided by the Custom Wizard:

  ```
  <SYMBOL NAME="WIZARD_DIALOG_TITLE" TYPE=text VALUE="MyCustomWiz">
        </SYMBOL>
  <SYMBOL NAME="SAMPLE_CHECKBOX" TYPE=checkbox VALUE=true>
        </SYMBOL>
  ```

  This wizard, titled MyCustomWiz, displays a check box that is selected by default.

- In the section marked `<SCRIPT LANGUAGE="JSCRIPT">` in the HTML files, add JScript function calls and access the Visual Studio Object Model to customize the behavior of your wizard. You must call these functions using `window.external`, as follows:

  ```
        window.external.AddSymbol("MAIN_FRAME_DEFAULT_STYLES", true);
        window.external.AddSymbol("MAIN_FRAME_STYLE_FLAGS", "");
  ```

  Note

  The methods, properties, and events exposed through Automation and Extensibility for Visual Studio, Visual C++ Code Model, Project Model, and Wizard Model allow you to programmatically manage all aspects of the wizard project, from creation through build, in both the JScript files and .htm files.

- If necessary, customize the .vsdir file, allowing information about the .vsz file and all other templates to be understood by the shell. For example, indicate the icon resource ID, flags, localized names, and so on.
- Create .htm files and template files in all languages for which your wizard needs to be localized. Add them to the appropriate project directories.
- Provide context-sensitive Help for your wizard.

See Also

Tasks

Creating a Custom Wizard

Reference

Custom Wizard

Providing Context-Sensitive Help

Concepts

Steps to Designing a Wizard

Files Created for Your Wizard

Handling Errors in Wizards

Other Resources

Custom Wizard Samples

Localizing a Wizard to Multiple Languages

You can create a wizard in any language for which Visual Studio provides support. By default, when you install Visual Studio, it identifies the locale from the registry and provides the appropriate templates for that locale.

Visual Studio uses language IDs to identify the language support a wizard requires. By default, the language ID is set to the decimal value of the registry entry HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\7.0\General\UILanguage. If the wizard can find no language entry, it defaults to English (1033).

Note
For a list of decimal language values, see Wizard Support for Other Languages.

The language ID is specified as a custom parameter in the .vsz file in the paths where the HTML files and the Template files reside.

You should specify paths for each language for which you provide an .htm file.

Example

Setting the following custom parameters in the .vsz file indicate that you are providing HTML in English (1033), Japanese (1041), and German (1031):

```
Param="START_PATH\HTML\1033"
Param="START_PATH\HTML\1041"
Param="START_PATH\HTML\1031"
Param="START_PATH\Templates\1033"
Param="START_PATH\Templates\1041"
Param="START_PATH\Templates\1031"
```

Setting the above custom parameters sets up your wizard directory structure as follows:

```
MyWizard1
   HTML
      1033
         default.htm
         myEnglishHTML.htm
      1041
         default.htm
         myJapaneseHTML.htm
      1031
         default.htm
         myGermanHTML.htm
   Templates
      1033
         stdafx.h
         stdafx.cpp
      1041
         stdafx.h
         stdafx.cpp
      1031
         stdafx.h
         stdafx.cpp
   Images
      HtmlPage1.bmp
      HtmlPage2.jpg
   Scripts
      Default.js
```

See Also

Tasks

Creating a Custom Wizard

Reference

Custom Wizard
Custom Parameters in the Wizard .vsz File

Concepts

Files Created for Your Wizard
Designing a Wizard

Providing Context-Sensitive Help

When you use the Custom Wizard to create a wizard, it inserts a **Help** button on your wizard.

Each .htm file in your project includes the following code to provide support for the **Help** button for your wizard.

```
<BUTTON CLASS="BUTTONS" ID="HelpBtn" ACCESSKEY="H"
 onClick="window.external.OnHelp('vc.appwiz.custom.overview');">
```

The OnHelp specifies the keyword of the HTML Help file associated with that page of your wizard. For more information about creating HTML Help files to associate with the page, see HTML Help Start Page. To provide your own help for this wizard page, you must replace the string `'vc.appwiz.custom.overview'` with the keyword that identifies the HTML Help topic for the page.

Note This .htm file cannot be integrated into the compiled MSDN Help.

See Also

Tasks

Creating a Custom Wizard

Reference

Custom Wizard

Concepts

Files Created for Your Wizard

Designing a Wizard

Files Created for Your Wizard

Your wizard uses the name that you specify in the **Name** box in the New Project dialog box to derive names for some files and classes.

The Custom Wizard adds comments to the files it creates for your project. The Custom Wizard also creates a text file, ReadMe.txt, in your new application directory. This file explains the contents and uses of the other new files created by the Custom Wizard.

The following table describes the files created by the Custom Wizard. For more information on how the key elements interact to create a wizard, see Designing a Wizard.

File	Description
Project.vsz	A text file similar to the old .ini format. It identifies the wizard engine and provides context and optional custom parameters.
Project.vsdir	A text file that provides a routing service between the Visual Studio shell and the items in the wizard project.
HTML files (optional)	A wizard can contain a user interface (UI), which is an HTML interface. A file without a UI contains no HTML files. Default.htm is the file that specifies features in the UI. If you specify more than one page in Application Settings of the Custom Wizard, the additional files are each named Page_*PageNum*.htm.
Script files	A wizard accesses the script engine and creates a JScript file, Default.js, for each project. It also includes Common.js. These files contain JScript functions that access the Visual C++ Wizard, Code, and Environment Object Models to customize a wizard. You can customize and add functions in the wizard project's Default.js file.
Templates	A wizard's templates are a collection of text files that contain directives, which are parsed and inserted into the symbol table, depending on the wizard user's selections. The template text files are rendered according to the user input and added to the project. The appropriate information is obtained by directly accessing the wizard control's symbol table.
Templates.inf	A text file that lists all templates associated with the project.
Default.vcproj	An .xml file that contains the information on the project type.
Sample.txt	A template file that shows how your wizard directives are used.
ReadMe.txt	A template file that contains a summary of each file created by the Custom Wizard.
Images (optional)	You can provide any images, such as icons, GIFs, BMPs, and other HTML-supported image formats, to enhance the UI for your wizard. A wizard that has no UI does not need images.
Styles.css (optional)	A file that defines the styles for the UI. If your wizard does not have a user interface, the Custom Wizard does not create a .css file.

Note If you delete your wizard files and directories, you must also delete the following files from the vc7\vcprojects directory. Until you delete these files, icons for your wizard will continue to appear in the New Project dialog box.

- *projectname*.vsz
- *projectname*.ico
- *projectname*.vsdir

Additionally, your wizard includes the file Common.js, which contains commonly used JScript functions and is shared among

all wizards, including the wizards used by Visual C++ to create other project types. See Customizing C++ Wizards with Common JScript Functions for more information.

See Also

Tasks

Creating a Custom Wizard

Reference

Custom Wizard

Concepts

Designing a Wizard

Customizing C++ Wizards with Common JScript Functions

The .vsz File (Project Control)

The starting point of each wizard is the .vsz file. The .vsz file is a text file that determines the wizard to be called and the information to pass to the wizard. The file contains a two-line header, followed by various optional parameters to be passed to the wizard. For a list of optional parameters, see Predefined Custom Wizard Symbols.

The following sample shows the header in a .vsz file:

```
VSWIZARD 7.0
Wizard=VsWizard.VsWizardEngine.8.0
Param="WIZARD_NAME = My AppWizard"
```

- The first line of the header specifies the version number of the template file format. You may specify this number as `6.0`, `7.0`, or `7.1`. No other numbers are valid, and using other numbers results in an "Invalid Format" error.
- The second line sets the **Wizard** variable to the ProgID of the wizard that is cocreated by Visual Studio. A ProgID is a string representation of a CLSID, such as `VsWizard.VsWizardEngine.8.0`.

  If your wizard has a user interface, the ProgID automatically specifies your wizard to implement IVCWizCtlUI. By default, the methods of this interface are used in the .htm files of your project. You can change the behavior of your wizard by using the methods for this interface in the .htm files. See VCWizCtl for more information, which is the coclass for **IVCWizCtlUI**.
- Following these two lines is an optional list of parameters that allow the .vsz file to pass additional custom parameters to the wizard. Each value is passed as a string element in an array of variants in the wizard control's Execute method. By default, a wizard with a user interface produces the following default parameters:

  ```
  Param="START_PATH = <path to the wizard>"
  Param="HTML_PATH = <path to the wizard's HTML file>"
  Param="TEMPLATES_PATH = <path to the wizard's template file>"
  Param="SCRIPT_PATH = <path to the wizard's script files>"
  Param="IMAGES_PATH = <path to the wizard's images>"
  ```

  If your wizard does not have a user interface, it does not have an `IMAGES_PATH` parameter and instead contains the following parameters:

  ```
  Param="WIZARD_UI = FALSE"
  Param="SOURCE_FILTER = txt"
  ```
- The .vsz file can contain the following parameters, which specify functions found in the Common.js file:

  ```
  Param="PREPROCESS_FUNCTION = CanAddATLClass"
  Param="PREPROCESS_FUNCTION = CanAddMFCClass"
  Param="PREPROCESS_FUNCTION = CanAddClass"
  ```

The functions CanAddATLClass, CanAddMFCClass, and CanAddClass are called by the wizard to confirm that the Visual C++ Code Model is available. If one function returns **false**, the wizard is not launched.

You can add your wizard to the Templates pane in the New Project dialog box in Visual Studio by placing the .vsz file in the vc7\vcprojects directory. By default, the Custom Wizard writes the .vsz file to this directory.

Note
If you delete your wizard files and directories, you must also delete the project's .vsz file, .vsdir file, and .ico file from the vc7\vcprojects directory.

See Also

Tasks

Creating a Custom Wizard

Reference

Custom Wizard

Concepts

Files Created for Your Wizard

Adding Wizards to the Add Item and New Project Dialog Boxes Using VSDir Files

Designing a Wizard

Other Resources

Visual C++ Wizard Model

Custom Parameters in the Wizard .vsz File

In its first two lines, the .vsz file identifies the wizard version and the ProgID or CLSID of the wizard to be cocreated. The .vsz file can also include optional context parameters and custom parameters that are added to the symbol table (along with the symbols supplied in the HTML symbol section).

The Execute method displays the wizard, which takes an array of the context and custom parameters defined in the .vsz file as its parameters.

The following commonly used symbols are specified as custom parameters in either the .vsz file or the .htm files and can be used in wizard HTML, script, or template files.

Example

As the following .vsz file entries indicate, the wizard named MyProjWiz contains a user interface.

```
VSWIZARD 7.0
Wizard=VsWizard.VsWizardEngine
Param="WIZARD_NAME = MyProjWiz"
Param="WIZARD_UI = TRUE"
```

Symbols for Custom Parameters in the Wizard .vsz File

Symbol	Definition
ABSOLUTE_PATH	The location of the wizard files.
HTML_FILTER	Specified in the .vsz file. File types that are placed in the HTML Files folder in Solution Explorer. Usually specified as "htm".
HTML_PATH	Specified in the .vsz file. The location of the wizard's HTML files. By default, it is START_PATH\HTML*LANGUAGE* (where *LANGUAGE* is the locale specified by your system registry). **Note** You can specify a different language by setting the <LangID> to the decimal value of HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\7.0\General\UILanguage. See Localizing a Wizard to Multiple Languages for more information. For a list of decimal language values, see Wizard Support for Other Languages.
IMAGE_FILTER	Specified in the .vsz file. File types that are placed in the Image Files folder in Solution Explorer. Usually specified as "bmp;gif".
IMAGES_PATH	Specified in the .vsz file. The location of the image files used in the html files. By default, it is START_PATH\Images.
MISC_FILTER	Specified in the .vsz file. File types that are placed in the Misc folder in Solution Explorer. Usually specified as "vsz;vsdir;ico;vcproj;csproj;css;inf".
PRODUCT	By default, set to Visual C++; however, you can set this value to Visual Basic to create Visual Basic wizards, and so on.
PRODUCT_INSTALLATION_DIR	The directory listed in the registry at HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\7.0\Setup\<Product>\ ProductDir.
PROJECT_TEMPLATE_NAME	Specified in the .vsz file. The project template file that your wizard uses to create projects. Usually specified as "txt".

PROJECT_TEMPLATE_PATH	The directory containing the project's template files. For Visual C++, it is PRODUCT_INSTALLATION_DIR\VCWizards, by default.
RELATIVE_PATH	If ABSOLUTE_PATH is not found, then RELATIVE_PATH is considered. This is the path relative to PRODUCT_INSTALLATION_DIR. For Visual C++, the RELATIVE_PATH is PRODUCT_INSTALLATION_DIR\VCWizards.
SCRIPT_COMMON_PATH	The directory name relative to PRODUCT_INSTALLATION_DIR, where the common script file can be found. For example, for Visual C++, this is VCWizards.
SCRIPT_FILTER	Specified in the .vsz file. File types that are be placed in the Script Files folder in Solution Explorer. Usually specified as either "js" (JScript) or "vbs" (VBScript).
SCRIPT_PATH	The location of the wizard's JScript file. By default, it is START_PATH\Scripts
START_PATH	Specified in the .vsz file. This is not set by the user, but used internally to identify either RELATIVE_PATH or ABSOLUTE_PATH. The wizard name (WIZARD_NAME) is appended to this value.
TEMPLATE_FILTER	Specified in the .vsz file. File types that are placed in the Template Files folder in Solution Explorer. Usually specified as "txt".
TEMPLATES_PATH	Specified in the .vsz file. The location of the wizard's template files. By default, it is START_PATH\Templates\<LangID>. **Note** See HTML_PATH for more information on LangID.
WIZARD_NAME	Specifies the wizard name. Located in the .vsz and used by the rest of the symbols.
WIZARD_UI	Specified in the .vsz file. A Boolean value indicating whether the wizard contains a user interface. Specify **TRUE** for a user interface or **FALSE** for no user interface.

See Also

Tasks

Creating a Custom Wizard

Reference

Custom Wizard

Execute

Concepts

Files Created for Your Wizard

Designing a Wizard

The HTML Files

A wizard can contain a user interface, which is an HTML interface. Along with Default.htm, a wizard can contain any number of .htm files, which you can indicate in the **Number of Pages** box in the Custom Wizard. Each .htm file represents an HTML page of your wizard, and the HTML page is accessible using **Next** and **Back** buttons, tabs, or any other format you specify in your wizard design.

The HTML contains:

- The SYMBOL tag, which identifies the default for user-defined options. Symbols are written into the symbol table when the user clicks **Finish**, such as:

```
<SYMBOL NAME='HEADER_FILE' VALUE='MyHeader.h' TYPE=text></SYMBOL>
```

In the wizard user interface (UI), the text box identified in the symbol table as "HEADER_FILE" contains the default text "MyHeader.h". You can change this value in the wizard UI, and the resulting value is written to the project's symbol table when you click **Finish**, such as:

```
<SYMBOL NAME='CHECKBOX1' TYPE=checkbox VALUE=false></SYMBOL>
```

In the wizard UI, the check box identified in the symbol table as "CHECKBOX1" is cleared by default. You can select this box in the HTML UI, and the resulting value is written to the symbol table when you click **Finish**.

Each .htm file records user selections to the symbols table.

- An include for Common.js, which contains commonly used and helpful JScript functions, and Default.js.
- References to the project's images to display in the HTML.
- HTML text and formatting that customize the appearance of the wizard's user interface
- JScript functions that access the Visual C++ Wizard Object Model to provide customized behavior from within the wizard. These functions are in the HTML page section headed <SCRIPT LANGUAGE='JSCRIPT'>, as shown in the following example.

  Note

  To access the Wizard and Environment Object Models from HTML, prepend the object model item with "window.external."

```
function InitDocument(document)
{
   setDirection();

   if (window.external.FindSymbol('DOCUMENT_FIRST_LOAD'))
   {
      // This function sets the default symbols based
      // on the values specified in the SYMBOL tags above
      //
      window.external.SetDefaults(document);
   }

   // Load the document and initialize the controls
   // with the appropriate symbol values
   //
   window.external.Load(document);
}
```

The following is a sample console application wizard:

```
<SYMBOL NAME='WIZARD_DIALOG_TITLE' TYPE=text VALUE='Console Application Wizard'></SYMBOL>

<SYMBOL NAME='EMPTY_PROJECT' TYPE=checkbox VALUE=false></SYMBOL>
<SYMBOL NAME='SUPPORT_ATL' TYPE=checkbox VALUE=false></SYMBOL>
<SYMBOL NAME='SUPPORT_MFC' TYPE=checkbox VALUE=false></SYMBOL>
```

See Also

Tasks

Creating a Custom Wizard

Reference

Custom Wizard

Concepts

Files Created for Your Wizard

Designing a Wizard

The Template Files

The templates that make up a wizard are a collection of text files that contain some simple directives and are parsed and rendered according to user input and added to the initial project. The appropriate information for parsing the templates is obtained by directly accessing the wizard control's symbol table.

The following example is a very simple template file for a wizard that asks the user to select either A or B.

Example

```
This file has been created by My Custom wizard.
You selected:
[!if TYPE_A]
Type A
[!else]
Type B
[!endif]
The name of this project is [!output PROJECT_NAME].root.cpp:
```

If the user chooses Type B, the above template would be rendered as follows:

Output

```
This file has been created by My Custom wizard.
You selected:
Type B
The name of this project is MyApp8.
```

Note The syntax above is new to Visual C++ .NET. Syntax from previous versions of Visual C++ is not supported in Visual C++ .NET.

See Also

Tasks

Creating a Custom Wizard

Reference

Custom Wizard

Concepts

Files Created for Your Wizard

Designing a Wizard

The Templates.inf File

The Templates.inf File

Templates.inf is a text file that contains a list of templates to render for the project.

Because Templates.inf is a template file itself, you can use directives to indicate which files to include in a project, depending on a user's selections. See Template Directives for a list of directives that you can use in this file.

Example

```
Template1.txt
Template2.txt
  [!if TYPE_A]
TemplateOptionA.h
TemplateOptionA.cpp
  [!else]
TemplateOptionB.h
TemplateOptionB.cpp
  [!endif]
```

CreateCustomInfFile renders Templates.inf into a temporary text file, which must then be deleted after processing the files.

Example

```
var InfFile = CreateCustomInfFile();
AddFilesToProject(selProj, strProjectName, InfFile);
InfFile.Delete();
```

See The JScript File for more information.

See Also

Tasks

Creating a Custom Wizard

Reference

Custom Wizard

Concepts

Files Created for Your Wizard

Designing a Wizard

Template Directives

You can use the following template directives in a wizard template file and the Templates.inf file to customize the wizard.

Directive	Description
[! if]	Begins a control structure to check for a condition.
[! else]	Part of the [! if] control structure. Checks for another condition.
[! endif]	Ends the definition of an [! if] control structure.
[! output]	Can be used in the following two ways: • [! output "string"] gives you the string. • [! output SYMBOL_STRING] gives you the value of the symbol SYMBOL_STRING.
[! loop]	Can be used in the following two ways: • [! loop = 5] • [! loop = *NUM_OF_PAGES*] where *NUM_OF_PAGES* is a symbol with a numeric value.
[! endloop]	Ends a loop structure.

The left bracket ([) followed immediately by an exclamation mark (!) indicates the beginning of a template directive. The right bracket indicates the end of a template directive. This is the required syntax:

```
[!directive params]
```

A space or a non-identifier character is required only between *directive* and *params*.

Example

```
[!if SAMPLE_RADIO_OPTION1]
You have checked the option 'Sample radio button option 1'
[!else]
You have checked the option 'Sample radio button option 2'
[!endif]
```

You can use the following operators with the above directives in a template file.

```
+
-
=
!=
==
||
&&
!
```

Example

```
[!if SYMBOL_STRING != 0]
```

See Also

Tasks

Creating a Custom Wizard

Reference

Custom Wizard

Concepts

Files Created for Your Wizard

Designing a Wizard

The JScript File

The Custom Wizard accesses the script engine and creates a JScript file called Default.js for each project. It also includes Common.js. These files contain JScript functions that give you access to the Visual Studio and Visual C++ object models to customize a wizard. (See Designing a Wizard for a list of these models.) You can add your own functions to the wizard project's Default.js file. To access properties and methods in the wizard object model or the environment model from a JScript file, prepend the object model item with "wizard." and "dte.", respectively.

For example:

```
function CreateCustomProject(strProjectName, strProjectPath)
{
   try
   {
      var strProjTemplatePath = wizard.FindSymbol('PROJECT_TEMPLATE_PATH');
var strProjTemplate = '';
      strProjTemplate = strProjTemplatePath + '\\default.vcproj';

      var Solution = dte.Solution;
      var strSolutionName = "";
      if (wizard.FindSymbol("CLOSE_SOLUTION"))
...
```

When you click **Finish** in the Custom Wizard, the wizard loads the file Default.js in the Script Files folder in Solution Explorer. This JScript file creates projects and renders templates and then adds them to the solution when a user clicks **Finish** in your wizard.

By default, the project's Default.js file includes the following functions:

Function name	Description
AddConfig	Adds the project's configurations. You can supply compiler and linker settings.
AddFilesToCustomProj	When the user clicks **Finish**, adds the specified files to the project.
AddFilters	When the user clicks **Finish**, adds the specified source filters to the project.
CreateCustomProject	When the user clicks **Finish**, creates the project at the specified location.
CreateCustomInfFile	Creates the project's Templates.inf file.
DelFile	Deletes the specified file.
GetTargetName	Get the name of the specified file.
OnFinish	Called by the wizard when the user clicks **Finish** to create the project, add files and filters, render templates, and set the configuration.
PchSettings	Sets the precompiled header settings. See SetCommonPchSettings in the Common.js reference for more information.

Each wizard has a unique Default.js file, which includes TODO comments to help you identify where you must customize the file.

Visual C++ also includes Common.js, a file shared among all wizards and included in your wizard project. You can use the functions in Common.js.

Note
Common.js contains descriptions of each function and its parameters. See the comments in Common.js for more informatio n.

If you have functions that you want to share between your wizard projects, you can add them to Common.js. Create your own version of Common.js and save it in a common path, and then set the SCRIPT_COMMON_PATH to this path in your .vsz file.

Note
The wizards included with Visual C++ use the JScript functions in Common.js. If you change these functions, the Visual C++ wizards can behave unexpectedly.

For more information about JScript, see Writing, Compiling, and Debugging JScript Code.

Debugging Script

To debug script in wizard html files, you must enable script debugging.

To enable script debugging

1. In Internet Explorer, click the **Tools** menu and choose **Internet Options**.
2. Click the **Advanced** tab.
3. Under the **Browsing** category, clear the **Disable Script Debugging** checkbox.

This will also allow common.js and default.js to show up in the **Running Documents** window when you click the Finish button on the wizard. For more information on debugging script in html files and in script files such as default.js and common.js, see Debugging Client-Side Scripts.

See Also

Tasks

Creating a Custom Wizard

Reference

Custom Wizard

Concepts

Files Created for Your Wizard

Designing a Wizard

Customizing C++ Wizards with Common JScript Functions

When you create a wizard project with the Custom Wizard, your project includes Common.js. If you specify a user interface for your wizard, the project also contains HTML pages that specify the JScript script language and include the file Common.js, as follows:

```
<SCRIPT ID="INCLUDE_COMMON" LANGUAGE ="JSCRIPT"></SCRIPT>
```

The wizard also creates a unique file called Default.js. You can customize your own JScript functions in Default.js. See The JScript File for more information.

Common.js contains functions that you can call from each wizard's HTML pages and from Default.js. If you have the same functions that you would like to use across different wizards, you can place these functions in Common.js.

See Also

Tasks

Creating a Custom Wizard

Concepts

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

JScript Functions for C++ Wizards

AddATLSupportToProject	Adds ATL support to an MFC project.
AddCoclassFromFile	Renders and inserts into the project's .idl file a template file that contains a coclass.
AddCommonConfig	Adds the default configurations to the project.
AddFilesToProject	Adds all the files to the project based on the list in the file Templates.inf.
AddInterfaceFromFile	Renders and inserts into the project's IDL file a template file that contains an interface.
CanAddATLClass	Called by the wizard to verify if the project is compatible with the code wizard that is about to be run (in other words, it can accept an ATL class). The wizard calls this function when the parameter PREPROCESS_FUNCTION is in the project control's .vsz file and checks if the Visual C++ Code Model is available. If the code model is not available, the function reports an error and returns **false**.
CanAddClass	The wizard calls this function when the parameter PREPROCESS_FUNCTION is in the project control's .vsz file. It verifies if the Visual C++ Code Model object is available. If the code model is not available, the function reports an error and returns **false**.
CanAddMFCClass	Called by the wizard to verify if the project is compatible with the Code Wizard that is about to be run (in other words, it can accept an MFC class). The wizard calls this function when the parameter PREPROCESS_FUNCTION is in the project control's .vsz file and checks if the Visual C++ Code Model object is available. If the code model is not available, the function reports an error and returns **false**.
CanAddNonAttributed	Indicates whether the project supports both attributed and nonattributed ATL objects.
CanUseFileName	Checks if a file exists. If so, the wizard prompts the user to merge the code to be added to the existing file.
ConvertProjectToAttributed	Converts an ATL project to attributed.
CreateInfFile	Creates the Templates.inf file.
CreateProject	Creates a C++ project.
CreateSafeName	Generates a C++ friendly name.
DeleteFile	Deletes the specified file.
DoesIncludeExist	Indicates whether a **#include** statement exists in a file.
GetCodeForDllCanUnloadNow	Retrieves code needed to unload the DLL.
GetCodeForDllGetClassObject	Retrieves the code for the DLL class object.

GetCodeForDllRegisterServer	Retrieves the code to register a server.
GetCodeForDllUnregisterServer	Retrieves the code to unregister a server.
GetCodeForExitInstance	Helper function to get the text for **ExitInstance**.
GetCodeForInitInstance	Helper function to get the text for InitInstance.
GetExportPragmas	Retrieves the pragmas for exporting functions.
GetInterfaceClasses	Returns the **VCCodeClass** object associated with an interface.
GetInterfaceType	Returns the type of interface (for example, custom, dual, dispinterface, oleautomation).
GetMaxID	Returns the highest *dispid* from members of this interface and all of its bases.
GetMemberfunction	Returns a function object based on the given name.
GetProjectFile	Returns the file name of per-project type of files (.rc, .idl, and so on).
GetProjectPath	Returns the project's directory path.
GetRuntimeErrorDesc	Returns a description for the type of exception.
GetUniqueFileName	Returns a unique file name.
IncludeCodeElementDeclaration	Adds the include statement to *strInFile*, including the header where *strCodeElemName* is implemented, if such a header found is in the project.
InsertIntoFunction	Helper function called in **AddATLSupportToProject** to insert code into **InitInstance**.
IsATLProject	Indicates whether project is ATL based.
IsAttributedProject	Indicates whether a project is attributed.
IsMFCProject	Checks if a project is MFC based.
LineBeginsWith	Helper function called in **InsertIntoFunction** to determine if a line begins with a particular string
OffsetToLineNumber	Finds the line number for a given position in a function body.
OnWizFinish	Called from the wizard HTML script when the user clicks **Finish**. Calls the wizard control's **Finish** method.
RenderAddTemplate	Renders a template file and optionally adds it to the project.
SetCommonPchSettings	Sets up the precompiled header for the project.
SetErrorInfo	Provides error information.
SetFilters	Adds source, include, and resource filters for project folders.

SetMergeProxySymbol	Called by the wizard to add the _MERGE_PROXYSTUB symbol if needed.
SetNoPchSettings	Sets up the project configuration properties when no precompiled header is used.

See Also

Tasks

Creating a Custom Wizard

Reference

JScript Functions for C++ Wizards

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

AddATLSupportToProject

Adds ATL support to an MFC project.

```
function AddATLSupportToProject(
      oProj
);
```

Parameters

oProj

The selected project.

Return Value

true if ATL support was successfully added to the MFC project.

Remarks

Use this function to add ATL support to an MFC project created by the wizard.

The wizard displays a message box to confirm adding ATL support to the MFC project. Upon confirmation, the wizard checks for existing support and adds all the necessary GUIDs, templates, headers, and additional functionality so that the MFC project created by the wizard supports ATL.

Example

```
var oCM = selProj.CodeModel;
var L_TRANSACTION_Text = "Add ATL Support To Project";
oCM.StartTransaction(L_TRANSACTION_Text);
var bRet = AddATLSupportToProject(selProj);
if (bRet)
   oCM.CommitTransaction();
else
   oCM.AbortTransaction();
return bRet;
```

See Also

Tasks

Creating a Custom Wizard

Reference

CanAddATLClass

IsMFCProject

Introduction to ATL

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

AddCoclassFromFile

Renders and inserts into the project's .idl file a template file containing a coclass.

```
function AddCoclassFromFile(
      oCM,
      strCoclassFile
);
```

Parameters

oCM

The Visual C++ Code Model object.

strCoclassFile

The name of the template file, excluding the path.

Remarks

Call this function to render and insert into the project's .idl file a template file containing a coclass.

Example

```
// Render myproj.idl and insert into the project's .idl.
AddCoclassFromFile(oCM, "myproj.idl");
```

See Also

Tasks

Creating a Custom Wizard

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

AddCommonConfig

Adds the default configurations to the project.

```
function AddCommonConfig(
      oProj,
      strProjectName
);
```

Parameters

oProj

The selected project.

strProjectName

The name of the project.

Remarks

Call this function to add the default code model configurations to the project your wizard creates. You can specify either a Release configuration or a Debug configuration. The following tables list the code model object's default property settings for each configuration type.

Visual C++ Compiler Tool Object

Object property	Release configuration setting	Debug configuration setting
UsePrecompiledHeader	pchUseUsingSpecific	pchUseUsingSpecific
WarningLevel	3	3
MinimalRebuild	Not applicable	true
DebugInformationFormat	debugEnabled	debugEditAndContinue
Optimization	optimizeMaxSpeed	Not applicable
BasicRuntimeChecks	Not applicable	runtimeBasicCheckAll
Detect64BitPortabilityProblems	true	true
OmitFramePointers	true	Not applicable
EnableFunctionLevelLinking	true	Not applicable
StringPooling	true	Not applicable

Visual C++ Configuration Object

Object property	Release configuration setting	Debug configuration setting
IntermediateDirectory	"Release"	"Debug"
OutputDirectory	"Release"	"Debug"

Visual C++ Linker Tool Object

Object property	Release configuration setting	Debug configuration setting
SubSystem	subSystemWindows	subSystemWindows

TargetMachine	machineX86	machineX86
GenerateDebugInformation	true	true

Example

```
// Create the Visual C++ project.
selProj = CreateProject(strProjectName, strProjectPath);
// Add the common configuration to the project.
   AddCommonConfig(selProj, strProjectName);
   selProj.Object.keyword = "MyProj";
```

See Also

Tasks

Creating a Custom Wizard

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

AddFilesToProject

Adds to the project all the files listed in Templates.inf.

```
function AddFilesToProject(
      oProj,
      strProjectName,
      InfFile
);
```

Parameters

oProj

The selected project.

strProjectName

The name of the project.

InfFile

The Templates.inf file object. This file contains a list of file names that the wizard creates on completion.

Remarks

Call this function to add to the project all of the files listed in Templates.inf. Using this function, you can add template files, resource files, or help files.

Example

```
// Assign the project path and project name.
var strProjectPath = wizard.FindSymbol("PROJECT_PATH");
var strProjectName = wizard.FindSymbol("PROJECT_NAME");

// Create the Visual C++ project and call it "MyProj"
selProj = CreateProject(strProjectName, strProjectPath);
selProj.Object.Keyword = "MyProj";

// Add common and specific configurations to the project.
AddCommonConfig(selProj, strProjectName);
AddSpecificConfig(selProj, strProjectName);

// Set the project filters
SetFilters(selProj);

// Create the project's Templates.inf file
var InfFile = CreateInfFile();

// Add the files in Templates.inf to the project.
AddFilesToProject(selProj, strProjectName, InfFile);
```

See Also

Tasks

Creating a Custom Wizard

Reference

CreateInfFile

SetCommonPchSettings

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

AddInterfaceFromFile

Renders and inserts a template file that contains an interface.

```
function AddInterfaceFromFile(
      oCM,
      strInterfaceFile
);
```

Parameters

oCM

The Visual C++ Code Model object.

strInterfaceFile

The template file name only, excluding the path.

Remarks

Call this function to render and insert into the project's .idl file a template file that contains an interface.

If the interface is not added successfully, this function displays an error message.

Example

```
// Render myint.idl and insert into strProject.idl
AddInterfaceFromFile(oCM, "myint.idl");
```

See Also

Tasks

Creating a Custom Wizard

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

CanAddATLClass

Called by the wizard to verify that the user can add an ATL class to the project.

```
function CanAddATLClass(
      oProj,
      oObject
);
```

Parameters

oProj

The selected project.

oObject

The selected object. In this case, the current project.

Return Value

true if the class can be added; **false** if the user calls the function for a project that is not an ATL project and does not have ATL support.

Remarks

Called by the wizard to verify if the project is compatible with the code wizard that is about to be run (in other words, it can accept an ATL class).

The wizard calls this function when the parameter PREPROCESS_FUNCTION is in the project control's .vsz file and checks if the Visual C++ Code Model is available. If the code model is not available, the function reports an error and returns **false**.

Example

```
// Determine if an ATL class can be added to the project
if (CanAddATLClass(selProj, selObj))
{
   return true;
}
```

See Also

Tasks

Creating a Custom Wizard

Reference

CanAddClass

IsMFCProject

CanAddMFCClass

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

CanAddClass

Called by the wizard to verify that the project is compatible with the code wizard the user is trying to run.

```
function CanAddClass(
      oProj,
      oObject
);
```

Parameters

oProj

The selected project.

oObject

The selected object. In this case, the current project.

Return Value

true if the class can be added; otherwise **false**.

Remarks

The wizard calls this function when the parameter PREPROCESS_FUNCTION is in the project control's .vsz file.

It verifies if the Visual C++ Code Model object is available. If the code model is not available, the function reports an error and returns **false**.

Example

```
// Determine if a class can be added to the project
if (CanAddClass(selProj, selObj))
{
   return true;
}
```

See Also

Tasks

Creating a Custom Wizard

Reference

CanAddMFCClass

CanAddATLClass

IsMFCProject

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

CanAddMFCClass

Called by the wizard to verify that the user can add an MFC class to the project.

```
function CanAddMFCClass(
      oProj,
      oObject
);
```

Parameters

oProj

Selected project

oObject

The selected object. In this case, the current project.

Return Value

true if the class can be added; **false** if the user calls the function for a project that is not an MFC project and does not have MFC support.

Remarks

Called by the wizard to verify if the project is compatible with the Code Wizard that is about to be run (in other words, it can accept an MFC class).

The wizard calls this function when the parameter PREPROCESS_FUNCTION is in the project control's .vsz file and checks if the Visual C++ Code Model object is available. If the code model is not available, the function reports an error and returns **false**.

Example

```
// Determine if an MFC class can be added to the project
if (CanAddMFCClass(selProj, selObj))
{
   return true;
}
```

See Also

Tasks

Creating a Custom Wizard

Reference

CanAddClass

CanAddATLClass

IsMFCProject

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

CanAddNonAttributed

Indicates that the ATL project supports nonattributed objects.

```
function CanAddNonAttributed( );
```

Return Value

true if the project supports nonattributed and attributed ATL objects; **false** if the project supports only attributed projects.

Remarks

Call this function to indicate whether the project supports both nonattributed and attributed objects.

Example

```
// Check if attributed project using CanAddNonAttributed
window.external.Load(document);
if (IsAttributedProject(window.external))
{
   ATTRIBUTED.checked = true;
   if (!CanAddNonAttributed())
      ATTRIBUTED.disabled = true;
}
```

See Also

Tasks

Creating a Custom Wizard

Reference

CanAddClass

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

Attributed Programming Concepts

CanUseFileName

Checks if a file exists. If the file exists and is not restricted, the wizard prompts the user to merge the code to be added to the existing file.

```
function CanUseFileName(
      strFileName,
      bCheckIfMidlHeader,
      bCannotExist,
      bSetMergeFlag
);
```

Parameters

strFileName

The name of the file to check.

bCheckIfMidlHeader

Set to **true** to check if file name is generated by MIDL.

bCannotExist

Set to **true** to check if the file name already exists and cannot be overwritten.

bSetMergeFlag

Set to **true** to include the MERG_FILE symbol, indicating that the user can merge the code to the existing file name.

Return Value

true if *strFileName* is unique, or if the code can be appended to the existing file; otherwise **false**.

Remarks

Call this function to check if a file name exists. If a file name exists, and it is not created by MIDL or is not otherwise restricted, the function prompts the user to merge the new code to the existing file.

If the file name does not exist and is not restricted, the file of the specified name is created.

If the file name is created by MIDL or is otherwise restricted, the wizard displays an error message.

Example

```
case "HTML_FILE":
if (!HTML_FILE.disabled)
   {
   if (!window.external.FindSymbol("HTML_FILE_VALID"))
      {
      bValid = CanUseFileName(obj.value, false, true);
      if (!bValid)
      break;
      window.external.AddSymbol("HTML_FILE_VALID", true)
      }
   }
   bValid = window.external.ValidateFile(HTML_FILE.value, vsCMValidateFileExtHtml);
   break;
```

See Also

Tasks

Creating a Custom Wizard

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

ConvertProjectToAttributed

Converts an ATL nonattributed project to attributed.

```
function ConvertProjectToAttributed( );
```

Return Value

true if the project was converted successfully; otherwise **false**.

Remarks

Call this function to convert an ATL project from nonattributed to attributed. See Attributed Programming for more information.

Example

```
// Create a function called CheckAddtoProject.
function CheckAddtoProject(oProj)
{
// Is the project attributed already?
   try
   {
      if (!IsAttributedProject(wizard))
      {
         // If the project is not converted to attributed, return false.
         if (!ConvertProjectToAttributed(oProj))
            return false;
      }
   }
   catch (e)
   {
      var L_ErrMsg1_Text = "Error in CheckAddtoProject: ";
      wizard.ReportError( L_ErrMsg1_Text + e.description);
      return false;
   }

   return true;
}
```

See Also

Tasks

Creating a Custom Wizard

Reference

CanAddNonAttributed

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

CreateInfFile

Creates the wizard's Templates.inf file.

```
function CreateInfFile( );
```

Return Value

The wizard template file.

Remarks

Call this function to create the wizard's Templates.inf file from Templatesinf.txt, a temporary text file it first creates in the temporary directory, based on the user's selections. Templates.inf contains the list of file names that the wizard creates. See The Template File for more information.

If this function cannot create the Templatesinf.txt file in the temporary directory, it displays an error.

Example

See AddFilesToProject.

See Also

Tasks

Creating a Custom Wizard

Reference

AddFilesToProject

SetFilters

AddFilesToProject

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

CreateProject

Creates a C++ project.

```
function CreateProject(
      strProjectName,
      strProjectPath
);
```

Parameters

strProjectName

A string containing the project name.

strProjectPath

A string containing the project path.

Return Value

The project object.

Remarks

Call this function to create a C++ project that you can open in Visual Studio. If the wizard's context parameter **WizardType** is specified as **vsWizardAddSubProject**, the project is added as a subproject to the existing project. See ContextParams Enum for more information.

Example

See AddFilesToProject.

See Also

Tasks

Creating a Custom Wizard

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

CreateSafeName

Generates a C++ friendly name.

```
function CreateSafeName(
      strName
);
```

Parameters

strName

The old name.

Return Value

The new, safe name.

Remarks

Call this function to create a C++ friendly name from a name that contains characters that cannot be used by C++. An acceptable C++ name can contain upper- or lowercase letters, digits, or an underscore ("_").

This function checks the old name character by character, skipping any unusable characters. If the old name contains no friendly characters, the function returns the new name as "My". If the new friendly name begins with a digit, this function prepends the new name with "My".

Example

```
// Get the project name
var strProjectName = wizard.FindSymbol("PROJECT_NAME");

// Set the project name to the safe project name.
var strSafeProjectName = CreateSafeName(strProjectName);
```

See Also

Tasks

Creating a Custom Wizard

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

DeleteFile

Deletes the specified file.

```
function DeleteFile(
      oFSO,
      strFile
)
```

Parameters

oFSO

The file system object.

strFile

The name of the file to be deleted.

Remarks

Call this function to delete the specified file.

Example

```
// Declare a temporary file.
var strFile = strTempFolder + "\\" + strTarget;
var strClassName = strTarget.split(".");
wizard.AddSymbol("SAFE_CLASS_NAME", strClassName[0]);
wizard.AddSymbol("SAFE_ITEM_NAME", strClassName[0]);

// Declare the template name.
var strTemplate = strTemplatePath + "\\" + strTpl;

// Render and insert the template.
wizard.RenderTemplate(strTemplate, strFile, bCopyOnly);

// Create a new project file and add the file from the template.
var projfile = projItems.AddFromTemplate(strFile, strTarget);

// Delete the temporary file from the file structure object.
DeleteFile(fso, strFile);
```

See Also

Tasks

Creating a Custom Wizard

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

DoesIncludeExist

Indicates whether a **#include** statement for a specified header file exists in a file.

```
function DoesIncludeExist(
      oProj,
      strHeaderFile,
      strInsertIntoFile
);
```

Parameters

oProj

The selected project.

strHeaderFile

The name of the header file to find.

strInsertIntoFile

The source file containing the **#include** statement for the header file (excluding the path).

Return Value

true if the specified header file is included; otherwise **false**.

Remarks

Indicates whether a #include for a specific header file exists in the file specified by *strInsertIntoFile*.

Example

```
// Check to see if #include for atlbase.h
// is included in the project's stdafx.h.
// and add it if it is not.
if (!DoesIncludeExist(selProj, "<atlbase.h>", strSTDAFX))
   oCM.AddInclude("<atlbase.h>", strSTDAFX, vsCMAddPositionEnd);
```

See Also

Tasks

Creating a Custom Wizard

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

GetCodeForDllCanUnloadNow

Gets the appropriate code for unloading the DLL.

```
function GetCodeForDllCanUnloadNow(
      nLineStart,
      nLineEnd
);
```

Parameters

nLineStart

The zero-based line number for the start of the function.

nLineEnd

The zero-based line number for the end of the function.

Return Value

A string containing the code for unloading the DLL.

Remarks

Call this member function to retrieve the appropriate code for unloading the DLL. Calling this function creates a single string by concatenating the array elements you specify.

The following table shows code for unloading the DLL.

Line number	Code
0	`AFX_MANAGE_STATE(AfxGetStaticModuleState());`
1	`if (_AtlModule.GetLockCount() > 0)`
2	`\treturn S_FALSE;`
3	`return S_OK;`

For each of the lines returned, **GetCodeForDllCanUnloadNow** adds a leading tab (`\t`) and a trailing "CR-LF" (carriage return - linefeed) character pair (`\r\n`).

Example

```
// Get the lines numbered 1 and 2 above
GetCodeForDllCanUnloadNow(1, 2)

// returns the following string
// "\tif (_AtlModule.GetLockCount() > 0)\r\n\t\treturn S_FALSE;\r\n"
```

See Also

Tasks

Creating a Custom Wizard

Reference

GetCodeForDllGetClassObject

GetCodeForExitInstance

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

GetCodeForDllGetClassObject

Retrieves the code for the DLL class object.

```
function GetCodeForDllGetClassObject(
      nLineStart,
      nLineEnd
);
```

Parameters

nLineStart

The zero-based line number for the start of the function.

nLineEnd

The zero-based line number for the end of the function.

Return Value

A string containing the code for getting the class object.

Remarks

Call this member function to retrieve the code for the class object. Calling this function creates a single string by concatenating the array elements you specify.

The following table shows code for getting the code for the class object:

Line number	Code
0	`AFX_MANAGE_STATE(AfxGetStaticModuleState());`
1	`if (S_OK == _AtlModule.GetClassObject(rclsid, riid, ppv))`
2	`\treturn S_OK;`
3	`return AfxDllGetClassObject(rclsid, riid, ppv);`

For each of the lines returned, **GetCodeForDllGetClassObject** adds a leading tab (`\t`) and a trailing "CR-LF" (carriage return - linefeed) character pair (`\r\n`).

Example

```
// Get the lines numbered 1 and 2 above
GetCodeForDllGetClassObject(1, 2)

// returns the following string
// "\tif (S_OK == _AtlModule.GetClassObject(rclsid, riid, ppv))\r\n\t\treturn S_OK;\r\n"
```

See Also

Tasks

Creating a Custom Wizard

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

GetCodeForDllRegisterServer

Gets the appropriate code for registering a server.

```
function GetCodeForDllRegisterServer(
      nLineStart,
      nLineEnd
);
```

Parameters

nLineStart

The zero-based line number for the start of the function.

nLineEnd

The zero-based line number for the end of the function.

Return Value

A string containing the code for registering the server

Remarks

Call this member function to retrieve the appropriate code for registering a server:

Line number	Code
0	`AFX_MANAGE_STATE(AfxGetStaticModuleState());`
1	`_AtlModule.UpdateRegistryAppId(TRUE);`
2	`HRESULT hRes = _AtlModule.RegisterServer(TRUE);`
3	`if (hRes != S_OK)`
4	`\treturn hRes;`
5	`if (!COleObjectFactory::UpdateRegistryAll(TRUE))`
6	`\treturn ResultFromScode(SELFREG_E_CLASS);`
7	`return S_OK;`

For each of the lines returned, **GetCodeForDllRegisterServer** adds a leading tab (`\t`) and a trailing "CR-LF" (carriage return - linefeed) character pair (`\r\n`).

Example

```
// Get the lines numbered 2 and 3 above
GetCodeForDllRegisterServer(2, 3)

// returns the following string
// "\tHRESULT hRes = _AtlModule.RegisterServer(TRUE);\r\n\tif (hRes != S_OK)\r\n"
```

See Also

Tasks

Creating a Custom Wizard

Reference

GetCodeForDllUnregisterServer

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

GetCodeForDllUnregisterServer

Gets the appropriate code for unregistering a server.

```
function GetCodeForDllUnregisterServer(
      nLineStart,
      nLineEnd
);
```

Parameters

nLineStart

The zero-based line number for the start of the function.

nLineEnd

The zero-based line number for the end of the function.

Return Value

A string containing the code for unregistering the server.

Remarks

Call this member function to retrieve the appropriate code for unregistering the server:

Line number	Code
0	`AFX_MANAGE_STATE(AfxGetStaticModuleState());`
1	`_AtlModule.UpdateRegistryAppId(FALSE);`
2	`HRESULT hRes = _AtlModule.UnregisterServer(TRUE);`
3	`if (hRes != S_OK)`
4	`\treturn hRes;`
5	`if (!COleObjectFactory::UpdateRegistryAll(FALSE))`
6	`\treturn ResultFromScode(SELFREG_E_CLASS);`
7	`return S_OK;`

For each of the lines returned, **GetCodeForDllUnregisterServer** adds a leading tab (`\t`) and a trailing "CR-LF" (carriage return - linefeed) character pair (`\r\n`).

Example

```
// Get the lines numbered 2 and 3 above
GetCodeForDllUnregisterServer(2, 3)

// returns the following string
// "\tHRESULT hRes = _AtlModule.UnregisterServer(TRUE);\r\n\tif (hRes != S_OK)\r\n"
```

See Also

Tasks

Creating a Custom Wizard

Reference

GetCodeForDllRegisterServer

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

GetCodeForExitInstance

Gets the ExitInstance code for terminating the wizard.

```
function GetCodeForExitInstance(
      nLineStart,
      nLineEnd
)
```

Parameters

nLineStart

The zero-based line number for the start of the function.

nLineEnd

The zero-based line number for the end of the function.

Return Value

A string containing the code for exiting the wizard instance.

Remarks

Call this member function to retrieve the appropriate code for exiting an instance of the wizard:

Line number	ExitInstance code
0	`_AtlModule.RevokeClassObjects();`
1	`return CWinApp::ExitInstance();`

For each of the lines returned, **GetCodeForExitInstance** adds a leading tab (`\t`) and a trailing "CR-LF" (carriage return - linefeed) character pair (`\r\n`).

Example

```
if (!oExitInstance)
   {
      oExitInstance = oCWinApp.AddFunction("ExitInstance",
      vsCMFunctionFunction, "BOOL", vsCMAddPositionEnd, vsCMAccessPublic,
      strProjectCPP);
      oExitInstance.BodyText = GetCodeForExitInstance(0, 1);
   }
// returns the following string
// "\t_AtlModule.RevokeClassObjects();\r\n
// \treturn CWinApp::ExitInstance();\r\n"
else
   {
   oExitInstance.StartPointOf(vsCMPartBody,
   vsCMWhereDefinition).CreateEditPoint().Insert(GetCodeForExitInstance(0,
   0));
// returns the following string
// "\t_AtlModule.RevokeClassObjects();\r\n
      oCM.Synchronize();
   }
```

See Also

Tasks

Creating a Custom Wizard

Reference

GetCodeForDllCanUnloadNow
GetCodeForInitInstance

Concepts

Customizing C++ Wizards with Common JScript Functions
Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

GetCodeForInitInstance

Retrieves the specified code for InitInstance.

```
function GetCodeForInitInstance(
      nLineStart,
      nLineEnd
);
```

Parameters

nLineStart

The zero-based line number for the start of the function.

nLineEnd

The zero-based line number for the end of the function.

Return Value

A string containing the code for initializing the wizard instance.

Remarks

Call this member function to retrieve the appropriate code for initializing the wizard instance. The line numbers are as follows:

Line number	InitInstance Code
0	`CWinApp::InitInstance();`
1	`return TRUE;`
2	`AfxOleInit();`
3	`// Parse command line for standard shell commands, DDE, file open`
4	`CCommandLineInfo cmdInfo;`
5	`ParseCommandLine(cmdInfo);`
6	`// App was launched with /Embedding or /Automation switch.`
7	`// Run app as automation server.`
8	`if (cmdInfo.m_bRunEmbedded || cmdInfo.m_bRunAutomated)`
9	`{`
10	`\t// Register class factories via CoRegisterClassObject().`
11	`\tif (FAILED(_AtlModule.RegisterClassObjects(CLSCTX_LOCAL_SERVER, REGCLS_MULTIPLEUSE)))`
12	`\t\treturn FALSE;`
13	`\t// Don't show the main window`
14	`\treturn TRUE;`

15	`}`
16	`// App was launched with /Unregserver or /Unregister switch.`
17	`if (cmdInfo.m_nShellCommand == CCommandLineInfo::AppUnregister)`
18	`{`
19	`\t_AtlModule.UpdateRegistryAppId(FALSE);`
20	`\t_AtlModule.UnregisterServer(TRUE);`
21	`\treturn FALSE;`
22	`}`
23	`// App was launched with /Register or /Regserver switch.`
24	`if (cmdInfo.m_nShellCommand == CCommandLineInfo::AppRegister)`
25	`{`
26	`\t_AtlModule.UpdateRegistryAppId(TRUE);`
27	`\t_AtlModule.RegisterServer(TRUE);`
28	`\treturn FALSE;`
29	`}`

For each of the lines returned, **GetCodeForInitInstance** adds a leading tab (`\t`) and a trailing "CR-LF" (carriage return - linefeed) character pair (`\r\n`).

Example

```
// Get the lines numbered 0 through 2 above
GetCodeForInitInstance(0, 2)

// returns the following string
// "\tCWinApp::InitInstance();\r\n\treturn TRUE;\r\n\tAfxOleInit();\r\n"
```

See Also

Tasks

Creating a Custom Wizard

Reference

GetCodeForExitInstance

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

GetExportPragmas

Adds pragmas for exporting functions.

```
function GetExportPragmas( );
```

Return Value

A string containing the export pragmas. Can be one of the following:

- `#pragma comment(linker, "/EXPORT:DllCanUnloadNow=_DllCanUnloadNow@0,PRIVATE")'`
- `#pragma comment(linker, "/EXPORT:DllGetClassObject=_DllGetClassObject@12,PRIVATE")`
- `#pragma comment(linker, "/EXPORT:DllRegisterServer=_DllRegisterServer@0,PRIVATE")`
- `#pragma comment(linker, "/EXPORT:DllUnregisterServer=_DllUnregisterServer@0,PRIVATE")`

Remarks

Call this function to add pragmas for exporting functions.

Example

```
oDllCanUnloadNow.StartPoint.Insert(GetExportPragmas() + "\r\n");
oCM.Synchronize();
```

See Also

Tasks

Creating a Custom Wizard

Reference

comment (C/C++)

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

GetInterfaceClasses

Returns the **VCCodeClass** object associated with an interface.

```
function GetInterfaceClasses(
      strInterface,
      oProject,
      aryClasses
);
```

Parameters

strInterface

The name of the interface containing the class objects.

oProject

The selected project.

aryClasses

[out] An array of class objects in the interface.

Return Value

The VCCodeClass object associated with an interface.

Remarks

Call this function to retrieve the classes associated with an interface.

Example

```
var aryClasses = new Array();
GetInterfaceClasses(oInterface.Name, selProj, aryClasses);
```

See Also

Tasks

Creating a Custom Wizard

Reference

GetInterfaceType

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

GetInterfaceType

Gets the type of interface.

```
function GetInterfaceType(
      oInterface
);
```

Parameters

oInterface

A VCCodeInterface object.

Return Value

The string indicating the type of interface, such as custom, dual, dispinterface, or oleautomation.

Remarks

Call this function to get the type of interface.

Example

See GetMaxID.

See Also

Tasks

Creating a Custom Wizard

Reference

GetInterfaceClasses

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

GetMaxID

Gets the highest dispid from members of this interface and all of its bases.

```
function GetMaxID(
      ointerface
);
```

Parameters

ointerface

A VCCodeInterface object.

Return Value

The highest dispid from the members of *oInterface* and all its bases.

Remarks

Call this function to get the highest dispid from the members of the specified interface and all its bases.

Example

```
if (strInterfaceType == "custom")
      window.external.AddSymbol("DISPID_DISABLED", true);
   else
   {
      var nDispID = window.external.FindSymbol("DISPID");
      if (!nDispID.length)
      {
         nDispID = GetMaxID(oInterface) + 1;
         window.external.AddSymbol("DISPID", nDispID);
      }
   }
```

See Also

Tasks

Creating a Custom Wizard

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

GetMemberfunction

Call this function to get a function object based on the given name.

```
function GetMemberfunction(
      oClass,
      strFuncName,
      oProj
);
```

Parameters

oClass

The selected class object.

strFuncName

The function name to retrieve.

oProj

The selected project.

Return Value

The function indicated by *strFuncName*.

Remarks

Call this function to get a function object based on the given name. See Functions in the Visual C++ Code Model for more information.

Example

```
// Gets the function ExitInstance from the class CWinApp in the
// current project.
var oExitInstance = GetMemberFunction(oCWinApp, "ExitInstance", oProj);
```

See Also

Tasks

Creating a Custom Wizard

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

GetProjectFile

Returns the file name specified type.

```
function GetProjectFile(
      oProj,
      strType,
      bFullPath,
      bMFC
);
```

Parameters

oProj

The selected project.

strType

The type of file to retrieve, such as STDAFX, RC, IDL, CPP, H, ODL, or DEF.

bFullPath

Flag indicating whether to return the full path name.

bMFC

Indicates if the project is an MFC-based project. If *strType* is .cpp or .h, it is considered MFC based. If not, the project is assumed to be ATL based.

Return Value

The file name of the per-project type of files.

Remarks

Call this function to get the file name of the specified type in the specified project.

Example

```
// The selected MFC project's CPP file is retrieved and
// assigned to strFileName.
var strFileName = GetProjectFile(selProj, "CPP", false, true);
```

See Also

Tasks

Creating a Custom Wizard

Reference

GetProjectPath

GetUniqueFileName

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

GetProjectPath

Returns the project's directory path.

```
function GetProjectPath(
      oProj
);
```

Parameters

oProj

The selected project

Return Value

The project's directory path.

Remarks

Call this function to get the project's path.

See Also

Tasks

Creating a Custom Wizard

Reference

GetProjectFile

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

GetRuntimeErrorDesc

Returns a description for the exception type.

```
function GetRuntimeErrorDesc(
      strRuntimeErrorName
);
```

Parameters

strRuntimeErrorName

The name of the type of exception that has occurred.

Return Value

A description of the exception type.

Remarks

Returns a description for the exception type. Can be one of the following exception types:

Exception types	Description
ConversionError	Occurs whenever there is an attempt to convert an object into something to which it cannot be converted.
RangeError	Occurs when a function is supplied with an argument that has exceeded its allowable range. For example, this error occurs if you attempt to construct an **Array** object with a length that is not a valid positive integer.
ReferenceError	Occurs when an invalid reference has been detected. This error will occur, for example, if an expected reference is null.
RegExpError	Occurs when a compilation error occurs with a regular expression. Once the regular expression is compiled, this error cannot occur. This example will occur, for example, when a regular expression is declared with a pattern that has an invalid syntax, or flags other than *i*, *g*, or *m*, or if it contains the same flag more than once.
SyntaxError	Occurs when source text is parsed and that source text does not follow correct syntax. This error will occur, for example, if the **eval** function is called with an argument that is not valid program text.
TypeError	Occurs whenever the actual type of an operand does not match the expected type. An example of when this error occurs is a function call made on something that is not an object or does not support the call.
URIError	Occurs when an illegal Uniform Resource Indicator (URI) is detected. For example, this is error occurs when an illegal character is found in a string being encoded or decoded.

See Also

Tasks

Creating a Custom Wizard

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

GetUniqueFileName

Returns a unique file name.

```
function GetUniqueFileName(
      strDirectory,
      strFileName
);
```

Parameters

strDirectory

Directory to look for file name in

strFileName

File name to check.

Return Value

The file name indicated in *strFileName* if unique; otherwise this function returns *strFileName*, appended with a number from 1 to 9999999, to make it unique. If *strFileName* is not provided, this function returns a unique file name by using GetTempName Method.

Remarks

Returns a unique file name.

See Also

Tasks

Creating a Custom Wizard

Reference

GetProjectFile

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

IncludeCodeElementDeclaration

Adds the include statement to *strInFile*, including the header where *strCodeElemName* is implemented, if such a header found in the *oProj*.

```
function IncludeCodeElementDeclaration(
      oProj,
      strCodeElemName,
      strInFile
);
```

Parameters

oProj

The selected project.

strCodeElemName

The full name of the code element for which you are searching the definition header.

strInFile

The file that will include the definition header, if found.

Remarks

Adds the include statement to *strInFile*, including the header where *strCodeElemName* is implemented, if such a header found in *oProj*.

See Also

Tasks

Creating a Custom Wizard

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

InsertIntoFunction

Used by AddATLSupportToProject to insert code into InitInstance.

```
function InsertIntoFunction(
      oFunction,
      strSearchString,
      nStartLine,
      nEndLine,
      bInsideIfBlock
);
```

Parameters

oFunction

The function object into which the insertion is made.

strSearchString

The string to find to determine insertion point.

nStartLine

The starting line to pass to GetCodeForInitInstance.

nEndLine

Ending line to pass to GetCodeForInitInstance.

bInsideIfBlock

Indicates whether the insertion is into a function block.

Return Value

true if successful; otherwise **false**.

See Also

Tasks

Creating a Custom Wizard

Reference

GetMemberfunction

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

IsATLProject

Indicates whether project is ATL-based.

```
function IsATLProject(
      oProj
);
```

Parameters

oProj

The selected project.

Return Value

true if the project is an ATL project; otherwise, **false**.

Remarks

Indicates whether project is ATL-based.

Example

```
function CanAddATLSupport(selProj, selObj)
{
   if (IsATLProject(selProj))
   {
      var L_ErrMsg1_Text = "Current project already has ATL support.";
      wizard.ReportError(L_ErrMsg1_Text);
      return false;
   }
```

See Also

Tasks

Creating a Custom Wizard

Reference

IsAttributedProject

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

IsAttributedProject

Indicates whether a project is attributed.

```
function IsAttributedProject(
      owizard
);
```

Parameters

owizard

The wizard object.

Return Value

true if the project is attributed; otherwise, **false**.

Remarks

Indicates whether a project is attributed.

Example

```
function CheckAddtoProject(oProj)
{
   try
   {
      if (!IsAttributedProject(wizard))
      {
         if (!ConvertProjectToAttributed(oProj))
            return false;
       }
   }
   catch (e)
   {
      var L_ErrMsg1_Text = "Error in CheckAddtoProject: ";
         wizard.ReportError( L_ErrMsg1_Text + e.description);
         return false;
   }

   return true;
}
```

See Also

Tasks

Creating a Custom Wizard

Reference

IsATLProject

IsMFCProject

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

IsMFCProject

Checks if a project is MFC based.

```
function IsMFCProject(
      oProj,
      bCWinAppRequired
);
```

Parameters

oProj

The selected project

bCWinAppRequired

Indicates whether extension DLLs are included in the check.

Return Value

true if the project is an MFC project; otherwise **false**.

Remarks

Use this function to determine if the selected project is an MFC project.

Example

```
if (!IsMFCProject(selProj, true))
   {
      if (gbExceptionThrown)
         return false;
      var L_ErrMsg2_Text = "ATL support can only be added to MFC EXEs or MFC Regular DLLs."
;
      wizard.ReportError(L_ErrMsg2_Text);
      return false;
   }
```

See Also

Tasks

Creating a Custom Wizard

Reference

IsATLProject

IsAttributedProject

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

LineBeginsWith

Called by InsertIntoFunction to determine if a line begins with the specified string.

```
function LineBeginsWith(
      strBody,
      strSearchString,
      nStartPos
);
```

Parameters

strBody

The body of the function.

strSearchString

The string to find.

nStartPos

The starting position for the search.

Return Value

true if the string is found; otherwise **false**.

Remarks

This function is called by **InsertIntoFunction** to determine if the specified line begins with the specified string.

See Also

Tasks

Creating a Custom Wizard

Reference

OffsetToLineNumber

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

OffsetToLineNumber

Called by InsertIntoFunction to convert an index in a function body to a line number.

```
function OffsetToLineNumber(
      strString,
      nPos
);
```

Parameters

strString

The string containing the function body. The function body is a multi-line string where its lines are delimited by cr-lf character pairs.

nPos

A position within the string.

Return Value

The line within the body function where *nPos* is located. The first line in the function is considered to be line 1 (not 0).

Remarks

Finds the line number for a given position in a function body.

This function is called by **InsertIntoFunction** to convert the index located at *nPos* in a function body to a line number.

Example

```
strString = "function DelFile(fso,
 strWizTempFile)\r\n{\r\n\ttry\r\n\t{\r\nif
(fso.FileExists(strWizTempFile))\r\nreturn true;\r\n";

nLine =  OffsetToLineNumber(strString, 60);

// The return value for the above is 5, because character 60 in the string
// occurs in the 5th line within the string.
```

See Also

Tasks

Creating a Custom Wizard

Reference

LineBeginsWith

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

OnWizFinish

Called from the wizard HTML script when the user clicks **Finish**. This function in turn calls the wizard control's **Finish** function.

```
function OnWizFinish(
      document
);
```

Parameters

document

The HTML document object

Remarks

This function is called from the wizard HTML script when the user clicks **Finish**.

See Also

Tasks

Creating a Custom Wizard

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

RenderAddTemplate

Renders a template file and optionally adds it to the project.

```
function RenderAddTemplate(
      strTemplateFile,
      strProjectFile,
      ProjToAddTo,
      bOpen
);
```

Parameters

strTemplateFile

The template file name only, excluding path, relative to TEMPLATES_PATH.

strProjectFile

The name of new file created. This string may include the path, but relative to PROJECT_PATH.

ProjToAddTo

The project object. Provide the project name if the created file must be added to the project; otherwise, disregard or pass **false** if you are not adding the file to the project.

bOpen

If **true**, open the file in the default editor after adding it to the project.

Remarks

Call this function to render a template file and optionally adds it to the project.

Example

```
// Declare the project path and the template path.
var strProjectPath = wizard.FindSymbol("PROJECT_PATH");
var strTemplatePath = wizard.FindSymbol("TEMPLATES_PATH");
// Declare the template header and implementation files.
var strTemplateHeader = wizard.FindSymbol("TEMPLATE_HEADER");
var strTemplateImpl = wizard.FindSymbol("TEMPLATE_IMPL");
// Render the template strTemplateHeader and open it in the editor.
RenderAddTemplate(strTemplateHeader, strHeaderFile, selProj, true);
// Render the template strTemplateImpl, but do not open it
// in the editor.
RenderAddTemplate(strTemplateImpl, strImplFile, selProj, false);
```

See Also

Tasks

Creating a Custom Wizard

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

SetCommonPchSettings

Establishes the precompiled header settings for the project.

```
function SetCommonPchSettings(
      oProj
);
```

Parameters

oProj

The selected project.

Remarks

Call this function to establish the precompiled header settings for the project.

This function sets the UsePrecompiledHeader property to the following:

- **pchUseUsingSpecific** Uses the /Yu (Use Precompiled Header File) compiler setting.
- **pchCreateUsingSpecific** Uses the /Yc (Create Precompiled Header File) compiler setting.

Example

```
if ((strAppType == "LIB" || ((strAppType == "CONSOLE") &&
   wizard.FindSymbol(SUPPORT_MFC"))) && !Pch)
   {
      AddFilesToProject(selProj, strProjectName, InfFile);
      SetNoCommonPchSettings(selProj);
   }
else
   {
      AddFilesToProject(selProj, strProjectName, InfFile);
      SetcommonPchSettings(selProj);
   }
```

See Also

Tasks

Creating a Custom Wizard

Reference

SetNoPchSettings

AddCommonConfig

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

SetErrorInfo

Called by OnWizFinish and CanUseFileName to provide current error information.

```
function SetErrorInfo(
      oErrorObj
);
```

Parameters

oErrorObj

The error object.

Remarks

Called by OnWizFinish and CanUseFileName to provide current error information. See SetErrorInfo in the Visual C++ Wizard Model documentation for more information.

See Also

Tasks

Creating a Custom Wizard

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

SetFilters

Adds source, include, and resource filters for the project folders.

```
function SetFilters(
      oProj
);
```

Parameters

oProj

The selected project.

Remarks

Call this function to add source, include, and resource filters for the project folders. This function finds the following symbols in the project:

- SOURCE_FILTER
- INCLUDE_FILTER
- RESOURCE_FILTER

These symbols contain the file extensions used in filtering.

Example

```
// Create and set the project name and path.
selProj = CreateProject(strProjectName, strProjectPath);
// Add the previously-identified configurations to the project.
AddConfigurations(selProj, strProjectName);
// Set filters for the project.
SetFilters (selproj);
// Indicate that the project is an ATL project.
selProj.Object.keyword = "AtlProj";
```

See Also

Tasks

Creating a Custom Wizard

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

SetMergeProxySymbol

This function is called by the wizard to add the _MERGE_PROXYSTUB symbol if needed.

```
function SetMergeProxySymbol(
      oProj
);
```

Parameters

oProj

Selected project object

Remarks

This function is called by the wizard to add the _MERGE_PROXYSTUB symbol if needed.

Example

```
SetMergeProxySymbol (selproj);
```

See Also

Tasks

Creating a Custom Wizard

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

SetNoPchSettings

Sets up a project configuration properties when no precompiled header is used.

```
function SetNoPchSettings(
      oProj
);
```

Parameters

oProj

The selected project.

Remarks

Call this function to configure the project when the project uses no precompiled header settings.

Example

See SetCommonPchSettings.

See Also

Tasks

Creating a Custom Wizard

Reference

SetCommonPchSettings

AddCommonConfig

Concepts

Customizing C++ Wizards with Common JScript Functions

Designing a Wizard

Other Resources

JScript Functions for C++ Wizards

Handling Errors in Wizards

This section contains some general rules to consider when handling errors in your wizard. For examples of error handling in both JScript and HTML, see:

- Error Handling in Custom Wizard JScript Files
- Error Handling in Custom Wizard HTML Files

See Also

Concepts

Files Created for Your Wizard
Customizing Your Wizard

Error Handling in Wizard HTML Files

When you create a wizard with a user interface, your project includes .htm files. Use these files to customize your project. See The HTML Files for more information.

Your project should include error handling. The following code provides you with an example of such code.

To handle errors in HTML

1. When you validate the fields, if you call a validation method in a DLL (which should set the error information), call `ReportError` with no parameters.

```
function ValidateInput()
{
   if (!window.external.ValidateFile(HEADER_FILE.value))
   {
      ReportError();
      HEADER_FILE.focus();
      return false;
   }
}
```

2. When you validate fields, if you validate the field using only the HTML script, call `SetErrorInfo` first and then call `ReportError` with no parameters.

```
function OnWhatever()
{
   if (!ValidateInput())
      window.external.ReportErrror();
   ....
}

function ValidateInput()
{
   .....

   if (HEADER_FILE.value == IMPL_FILE.value)
   {
      var L_ErrMsg_Text = "Header and implementation files cannot have the same name."
;
      SetErrorInfo(L_ErrMsg_Text);
      bValid = false;
   }
   if (TYPE.value == "")
   {
      var L_ErrMsg4_Text = "Type cannot be blank.";
      SetErrorInfo(L_ErrMsg4_Text);
      bValid = false;
   }
   return bValid;
}
```

3. Call `ReportError` with parameters:

```
function ValidateInput()
{
```

```
    if (!IsListed(strType))
    {
       var L_Invalid2_Text = "The variable type should be one of the types listed.";
       window.external.ReportError(L_Invalid2_Text);
       VariableType.focus();
       return false;
    }
 }
```

4. If you must go back to the New Project or Add New Item dialog box, return **VS_E_WIZBACKBUTTONPRESS**:

```
 try
 {
    oCM   = window.external.ProjectObject.CodeModel;
 }
 catch(e)
 {
    var L_NCBError_Text = "Cannot access the Class View information
 (.ncb) file. Class View information will not be available.";
    window.external.ReportError(L_NCBError_Text);
    return VS_E_WIZARDBACKBUTTONPRESS;
```

See Also

Concepts

Files Created for Your Wizard

Customizing Your Wizard

Error Handling in Wizard JScript Files

When you create a wizard, your project includes the Default.js and Common.js files. Use these files to customize your project. See The JScript File for more information.

Your project should include error handling. The following code provides you with an example of such code.

To handle errors in JScript

1. To catch errors when the user clicks **Finish**, enter the following code:

```
function OnFinish(selProj, Class)
{
   try
   {
      .....
   }
   catch(e)
   {
      if (e.description.length != 0)
         SetErrorInfo(e.description, e.number);
      return e.number
   }
}
```

2. Throw `e` from any helper script functions called in the script:

```
function ExtenderFromType(strVariableType)
{
   try
   {
      ....
   }
   catch(e)
   {
      throw e;
   }
}
```

3. If the parameter **PREPROCESS_FUNCTION** is in the .vsz file, the wizard calls CanAddATLClass. Use SetErrorInfo in case of failure and return **false**:

```
function CanAddATLClass(oProj, oObject)
{
   try
   {
      if (!IsATLProject(oProj))
      {
         if (!IsMFCProject(oProj, true))
         {
            var L_CanAddATLClass_Text = "ATL classes can only be added
 to ATL, MFC EXE and MFC regular DLL projects.";
            wizard.ReportError(L_CanAddATLClass_Text);
            return false;
         }
         else
         {
```

```
            .....
            var bRet = AddATLSupportToProject(oProj);
            .....
            return bRet;
         }
      }
      return true;
   }
   catch(e)
   {
      throw e;
   }
}
```

4. If you must go back to the New Project or Add New Item dialog box, return **VS_E_WIZBACKBUTTONPRESS**:

```
   function OnFinish(selProj, Class)
   {
      ....
      if (!CheckAddtoProject(selProj))
      {
         return VS_E_WIZARDBACKBUTTONPRESS;
      }
   }
```

See Also

Concepts

Files Created for Your Wizard
Customizing Your Wizard

Building a C/C++ Program

You can build projects in Visual C++ in one of two ways:

- Using Visual Studio.
- Using the command line.

In This Section

Building C++ Projects in Visual Studio

Discusses using the Visual Studio integrated development environment (IDE) and property pages to build your project.

Building on the Command Line

Discusses command-line tools for programmers who prefer to build their applications from the command prompt.

Building C/C++ Isolated Applications and Side-by-side Assemblies

Describes the new deployment model for Windows client applications based on the idea of isolated applications and side-by-side assemblies.

C/C++ Building Reference

Provides links to an overview of building programs in C++, compiler and linker options, and additional build tools.

Related Sections

Visual C++

Provides links to different areas of the Visual Studio and Visual C++ documentation set.

Visual C++ Reference

Provides links to topics describing the C and C++ language references, the libraries provided with Visual C++, the Visual C++ Extensibility Object Model, and the Microsoft Macro Assembler (MASM).

Visual C++ Samples

Provides links to sample code showing the capabilities of Visual C++ and the libraries and technologies it supports.

Building C++ Projects in Visual Studio

When building a Visual C++ application in Visual Studio, you can modify many of the build's settings in the project's Property Pages dialog box. For details, see Setting Visual C++ Project Properties.

In This Section

Understanding Custom Build Steps and Build Events

Describes how to customize the build process in the integrated development environment.

Macros for Build Commands and Properties

Lists macros that you can use where strings are accepted.

Building External Projects

Discusses building projects that use facilities outside the integrated development environment.

Invalid Web Reference

Lists the files that can be use as Web references in a Visual C++ project.

Format of a .vcproj File

Presents the XML structure of a .vcproj file.

Related Sections

VC++ Directories, Projects, Options Dialog Box

Discusses how to modify the search path for executable files, include files, library files, and source code files during a build

Default and Custom Builds

Provides information on building within Visual Studio.

Building a C/C++ Program

Provides links to topics describing building your program from the command line or from the integrated development environment of Visual Studio.

C/C++ Building Reference

Provides links to an overview of building programs in C++, compiler and linker options, and additional build tools.

Upgrade Projects from Previous Versions of Visual C++

Provides links to topics covering issues on upgrading Visual C++ 4.x, 5.0, and 6.0 projects to Visual C++ .NET.

Porting and Upgrading Programs

Provides details about porting applications and discusses makefiles.

Understanding Custom Build Steps and Build Events

From within the Visual C++ development environment, there are two ways customize the build process:

Custom Build Steps

A custom build step is a build rule associated with either a project or one or more files. A custom build step can pass input files to a tool, which results in one or more output files. For example, the help files in an MFC application are built with custom build steps. For more information, see Specifying Custom Build Steps.

Build Events

Build events let you customize a project's build. There are three build events: Pre-Build, Pre-Link, and Post-Build. A build event lets you specify an action to occur at a specific time in the build process. For example, you could use a build event to register a file with regsvr32.exe after the project finishes building. For more information, see Specifying Build Events.

Troubleshooting Custom Build Steps and Build Events can help you ensure that your custom build steps and build events run as expected.

The output format of a custom build step or build event can also enhance the usability of the tool. For more information, see Formatting the Output of a Custom Build Step or Build Event.

Build events and custom build steps run in the following order along with other build steps:

1. Pre-Build event
2. Custom build steps on individual files
3. Proxy generator
4. MIDL
5. Resource compiler
6. The C/C++ compiler
7. Pre-Link event
8. Linker or Librarian (as appropriate)
9. BSCMake
10. Custom build step on the project
11. Web deployment tool. The web deployment tool runs as part of a build only if the linker or librarian tools also run. However, you can run the web deployment tool via the **Build** menu.
12. Post-Build event

A custom build step on the project, the web deployment tool, and a post-build event run (sequentially) at the same point in the build — after all other build processes are completed.

See Also

Reference

Tool Build Order Dialog Box

Other Resources

Building C++ Projects in Visual Studio

Macros for Build Commands and Properties

Specifying Custom Build Steps

A custom build step provides the build system with the information it needs to build output files from an input file. A custom build step is attached to its input file; it contains the commands to run, a list of the output files that are generated by those commands, and (optionally) a description of the command.

For general information on custom build steps, see Understanding Custom Build Steps and Build Events.

To specify a custom build step

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. In the **Configuration** box, select the configuration for which you want to specify a custom build step.
3. In Solution Explorer, select the input file for the custom build step.

   If the **Custom Build Step** folder does not appear, the file you selected has a default step associated with it. For example, .c and .cpp files have a default step to invoke the compiler. To override a default setting, modify the **Tool** property, which is in the **Configuration Settings** folder.
4. Specify the properties associated with the custom build step:

   - In **Additional Dependencies**, specify any additional files beyond the one for which the custom build step is being defined (the file associated with the custom build step is implicitly considered an input to the step). Having additional input files is not a requirement for a custom build step. If you have more than one additional input, separate them with semicolons.

     If an **Additional Dependencies** file's date is later than the input file, then the custom build rule will be run. If all of the **Additional Dependencies** files are older than the input file, and the input file is older than the **Outputs** file, then the custom build step will not run.

     For example, suppose you have a custom build step that takes MyInput.x as input and generates MyInput.cpp, and that MyInput.x includes a header file, MyHeader.h. You can specify MyHeader.h as an input dependency to MyInput.x, and the build system will build MyInput.cpp when it is out-of-date with respect to MyInput.x or MyHeader.h.

     Input dependencies can also ensure that your custom build steps run in the order you need them to. In the preceding example, suppose that MyHeader.h is actually the output of a custom build step. Because MyHeader.h is a dependency of MyInput.x, the build system will first build Myheader.h before running the custom build step on MyInput.x.
   - In **Command Line**, specify the syntax of the custom build rule. This syntax can include any command that is legal at the command line or in a .bat file. You may want to use Macros for Build Commands and Properties to specify locations for files or to get the actual name of the input file in the case of multiple selections. The name of a batch file should be preceded by **call** to ensure that all subsequent commands will be executed.
   - In **Description**, type a description of the custom build step. This will be printed to the **Output** window when the build system processes this step.
   - In **Outputs**, specify the name of the output file. This is a required entry; without a value for this property, the custom build step will not run. If a custom build step has more than one output, separate file names with a semicolon.

     The name of the output file should be what is specified in the **Command Line** property. The project build system will look for the file and check its date. If the file is newer than the input file or if the file is not found, then the custom build step will run. If all of the **Additional Dependencies** files are older than the input file, and the input file is older than the **Outputs** file, then the custom build step will not run.

If you want the build system to operate on an output file generated by the custom build step, you must manually add it to the project. The custom build step will update the file during the build.

Example

Assume that you want to include in your project a file named parser.l. You want a lexical analyzer to process parser.l to produce a .c file with the same base name (parser.c).

First, you add parser.l and parser.c to the project. If the files don't yet exist, you just add a reference to the files. You create a custom build step for parser.l and type the following in the **Commands** property:

```
lexer $(InputPath) .\$(InputName).c
```

This command will run the lexical analyzer on parser.l and output parser.c to the project directory.

In the **Outputs** property, type the following:

```
.\$(InputName).c
```

When you build the project, the build system compares the timestamps of parser.l and parser.c. If parser.l is more recent, or if parser.c doesn't exist, the build system runs the **Command** property to bring parser.c up to date. Since parser.c was also added to the project, the build system will then proceed to compile parser.c.

See Also

Reference

Troubleshooting Custom Build Steps and Build Events

Other Resources

Macros for Build Commands and Properties

Specifying Build Events

You can use build events to specify commands that run before the build starts, before the link process, or after the build finishes.

Build events are executed only if the build successfully reaches those points in the build process. If an error occurs in the build, the Post-Build event will not occur; if the error occurs before the linking phase, neither the Pre-Link nor the Post-Build event will occur. Additionally, if no files need to be linked, the Pre-Link event will not occur. The Pre-Link event is also not available in projects that do not contain a link step.

If no files need to be built, no build events will occur.

For general information on build events, see Understanding Custom Build Steps and Build Events.

To specify a build event

1. In Solution Explorer, select the project for which you want to specify the build event.
2. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
3. In the **Build Events** folder, select a build event property page.
4. Specify the properties associated with the build event:
   - In **Command Line**, specify the syntax of the build event. This syntax can include any command that is legal at the command line or in a .bat file. You may want to use Macros for Build Commands and Properties to specify locations for files or to get the actual name of the input file in the case of multiple selections. The name of a batch file should be preceded by **call** to ensure that all subsequent commands will be executed.
   - In **Description**, type a description for this event. This will be printed to the **Output** window when this event occurs.
   - In **Excluded From Build**, specify **Yes** if you do not want the event to run.

See Also

Reference

Understanding Custom Build Steps and Build Events
Troubleshooting Custom Build Steps and Build Events

Other Resources

Macros for Build Commands and Properties

Troubleshooting Custom Build Steps and Build Events

If your custom build steps or events are not behaving as you expect, there are several things you can do to try to understand what is going wrong.

- Make sure that the files your custom build steps generate match the files you declare as outputs.
- If your custom build steps generate any files that are inputs or dependencies of other build steps (custom or otherwise), make sure that those files are added to your project.
- Add `@echo on` as the first command to see what your custom build step is actually doing. The build events and build steps are placed in a temporary .bat file and run when the project is built, so, you can add error checking to your build event or build step commands.
- Examine the build log (BuildLog.htm) in the intermediate files directory to see what actually executed.

  You can enable the build log by opening the **Options** dialog box (**Tools** menu) and then the **VC++ Build** property page in the **Projects** folder. Ensure that **Build Logging** is set to **Yes**.
- Verify the values of any file-name or directory macros you are using. You can echo macros individually, or you can add `copy %0 command.bat` to the beginning of your custom build step, which will copy your custom build step's commands to command.bat with all macros expanded.
- Run custom build steps and build events individually to check their behavior.

See Also

Reference

Understanding Custom Build Steps and Build Events

Formatting the Output of a Custom Build Step or Build Event

If the output of a custom build step or build event is formatted correctly, users get the following benefits:

- Warnings and errors are counted in the Output window.
- Output appears in the Task List window.
- Clicking on the output in the Output window displays the appropriate topic.
- F1 operations are enabled in the Task List window or Output window.

The format of the output should be:

```
{filename (line# [, column#]) | toolname} :
  [anytext] {error | warning} code####: Localizable String
   [
      any text
   ]
```

Where:

- {*a* | *b*} is a choice of either *a* or *b*.
- [*optional*] is an optional parameter

For example:

C:\sourcefile.cpp(134) : error C2143: syntax error : missing ';' before '}'

LINK : fatal error LNK1104: cannot open file 'somelib.lib'

See Also

Reference

Understanding Custom Build Steps and Build Events

Macros for Build Commands and Properties

You can use these macros anywhere in a project's **Property Pages** dialog box where strings are accepted. These macros are not case sensitive.

Macro	Description
$(RemoteMachine)	Set to the value of the **Remote Machine** property on the Debug property page. See Changing Project Settings for a C/C++ Debug Configuration for more information.
$(References)	A semicolon delimited list of references added to the project.
$(ConfigurationName)	The name of the current project configuration (for example, "Debug").
$(PlatformName)	The name of current project platform (for example, "Win32").
$(Inherit)	Specifies the order in which inherited properties appear in the command line composed by the project build system. By default, inherited properties appear at the end of the current property.1
$(NoInherit)	Causes any properties that would otherwise be inherited, to not be inherited. To also prevent evaluation at the sibling level, use **$(StopEvaluating)**. The use of **$(NoInherit)** causes any occurrences of **$(Inherit)** to be ignored for the same property.1
$(StopEvaluating)	Immediately stops the evaluation of a macro in the evaluation chain. Any values that appear after **$(StopEvaluating)** will not appear in the evaluated value of the macro. If **$(StopEvaluating)** precedes **$(Inherit)**, the inherited value at the current location in the evaluation chain will not be concatenated to the macro value. **$(StopEvaluating)** is a superset of the functionality of **$(NoInherit)**.
$(ParentName)	Name of the item containing this project item. This will be the parent folder name, or project name.
$(RootNameSpace)	The namespace, if any, containing the application.
$(IntDir)	Path to the directory specified for intermediate files relative to the project directory. This resolves to the value for the **Intermediate Directory** property.
$(OutDir)	Path to the output file directory, relative to the project directory. This resolves to the value for the **Output Directory** property.
$(DevEnvDir)	The installation directory of Visual Studio .NET (defined as drive + path); includes the trailing backslash '\'.
$(InputDir)	The directory of the input file (defined as drive + path); includes the trailing backslash '\'. If the project is the input, then this macro is equivalent to $(ProjectDir).

$(InputPath)	The absolute path name of the input file (defined as drive + path + base name + file extension). If the project is the input, then this macro is equivalent to $(ProjectPath).
$(InputName)	The base name of the input file. If the project is the input, then this macro is equivalent to $(ProjectName).
$(InputFileName)	The file name of the input file (defined as base name + file extension). If the project is the input, then this macro is equivalent to $(ProjectFileName).
$(InputExt)	The file extension of the input file. It includes the '.' before the file extension. If the project is the input, then this macro is equivalent to $(ProjectExt).
$(ProjectDir)	The directory of the project (defined as drive + path); includes the trailing backslash '\'.
$(ProjectPath)	The absolute path name of the project (defined as drive + path + base name + file extension).
$(ProjectName)	The base name of the project.
$(ProjectFileName)	The file name of the project (defined as base name + file extension).
$(ProjectExt)	The file extension of the project. It includes the '.' before the file extension.
$(SolutionDir)	The directory of the solution (defined as drive + path); includes the trailing backslash '\'.
$(SolutionPath)	The absolute path name of the solution (defined as drive + path + base name + file extension).
$(SolutionName)	The base name of the solution.
$(SolutionFileName)	The file name of the solution (defined as base name + file extension).
$(SolutionExt)	The file extension of the solution. It includes the '.' before the file extension.
$(TargetDir)	The directory of the primary output file for the build (defined as drive + path); includes the trailing backslash '\'.

$(TargetPath)	The absolute path name of the primary output file for the build (defined as drive + path + base name + file extension).
$(TargetName)	The base name of the primary output file for the build.
$(TargetFileName)	The file name of the primary output file for the build (defined as base name + file extension).
$(TargetExt)	The file extension of the primary output file for the build. It includes the '.' before the file extension.
$(VSInstallDir)	The directory into which you installed Visual Studio .NET.
$(VCInstallDir)	The directory into which you installed Visual C++ .NET.
$(FrameworkDir)	The directory into which the .NET Framework was installed.
$(FrameworkVersion)	The version of the .NET Framework used by Visual Studio. Combined with $(FrameworkDir), the full path to the version of the .NET Framework use by Visual Studio.
$(FrameworkSDKDir)	The directory into which you installed the .NET Framework SDK. The .NET Framework SDK could have been installed as part of Visual Studio .NET or separately.
$(WebDeployPath)	The relative path from the web deployment root to where the project outputs belong. Returns the same value as RelativePath.
$(WebDeployRoot)	The absolute path to the location of <localhost>. For example, c:\inetpub\wwwroot.
$(SafeParentName)	The name of the immediate parent in valid name format. For example, a form is the parent of a .resx file.
$(SafeInputName)	The name of the file as a valid class name, minus file extension.

$(SafeRootNamespace)	The namespace name in which the project wizards will add code. This namespace name will only contain characters that would be permitted in a valid C++ identifier.
$(FxCopDir)	The path to the fxcop.cmd file. The fxcop.cmd file is not installed with all Visual C++ editions.

1. Use the Command Line Property Page for the property to see how properties are inherited. See Specifying Project Settings with Property Pages for more information on property inheritance. See Using $(Inherit) and $(NoInherit) for usage examples.

See Also

Other Resources

Building C++ Projects in Visual Studio

Using $(Inherit) and $(NoInherit)

This topic gives some examples on how to use the **$(Inherit)** and **$(NoInherit)** macros.

- Set the following additional include path (/I) at the project level:

  ```
  c:\test
  ```

- For a particular file, set the following additional include path:

  ```
  c:\test2;c:\mystuff
  ```

- Observe the file's **Command Line** property page. It includes:

  ```
  /I"c:\test2" /I"c:\mystuff" /I"c:\test"
  ```

  This is because there is an implicit `$(Inherit)` at the end of the file configuration's additional include paths if `$(Inherit)` is not placed in the list explicitly.

- However, if you change the file configuration's additional include path to this:

  ```
  c:\test2;$(Inherit);c:\mystuff
  ```

  it will expand to the following on the file's **Command Line** property page:

  ```
  /I"c:\test2" /I"c:\test" /I"c:\mystuff"
  ```

  Notice that the additional include path from the project configuration is where the `$(Inherit)` macro was placed. The `$(Inherit)` macro is used to guide the placement of inherited values that are part of this property. It can go anywhere in the list.

- If you change the file configuration's additional include path to this:

  ```
  $(Inherit);c:\test2;$(Inherit);c:\mystuff
  ```

  it will expand to the following on the file's **Command Line** property page:

  ```
  /I"c:\test" /I"c:\test2" /I"c:\test" /I"c:\mystuff"
  ```

- If you change the file configuration's additional include path to this:

  ```
  c:\test2;c:\mystuff;$(NoInherit)
  ```

  it will expand to the following on the file's **Command Line** property page:

  ```
  /I"c:\test2" /I"c:\mystuff"
  ```

  Notice that `/I"c:\test"` from the project configuration is gone. The location of the `$(NoInherit)` macro has no bearing on how it is used, unlike `$(Inherit)`, which is location-sensitive.

`$(NoInherit)` takes precedence over `$(Inherit)`. If `$(NoInherit)` is present, `$(Inherit)` will be ignored. For example, changing the file configuration's additional include path to this:

```
c:\test2;$(Inherit);c:\mystuff;$(NoInherit)
```

will expand it exactly the same as if it was:

```
c:\test2;c:\mystuff;$(NoInherit)
```

Use care with `$(NoInherit)` in the **Defines** property for tools like the C/C++ compiler or linker; you can cancel the use of project defaults (such as those set by **Use of ATL** and **Use of MFC**).

See Also

Other Resources

Macros for Build Commands and Properties

Building External Projects

An external project is a Visual C++ project that uses a makefile or other facilities that are outside (foreign or external to) the Visual C++ development environment.

If you have an external project (for example, a makefile project) that you want to build in the Visual C++ development environment, create a Makefile project and specify your project's build command and output in the Makefile Application Wizard. For more information, see Creating a Makefile Project.

Note that Visual C++ no longer supports the ability to export a makefile for the active project from the development environment. Use Devenv Command Line Switches to build Visual Studio projects at the command line.

See Also

Other Resources

Building C++ Projects in Visual Studio

Invalid Web Reference

This error message appears when you try to add a Web reference to a Visual C++ project, but the URL specifies a file with an extension that cannot be used by the project. The URL must reference a file with one of the following extensions:

- .disco
- .vsdisco
- .asmx
- .asmx
- .wsdl or ?wsdl

You may be able to locate the appropriate file by removing the file name from the address and using just the server and service name in the URL.

See Also

Reference

Add Web Reference Dialog Box

The proxy settings on this computer are not configured correctly for Web discovery.

Concepts

XML Web Service Discovery

Project Files

A Visual C++ project file is an XML file with the extension .vcproj, which contains information needed to build a Visual C++ project.

Note

.vcproj files are not compatible with NMAKE. For more information on how to build Visual C++ projects with NMAKE, see NMAKE Reference.

Example

The following is a sample .vcproj file:

```
<?xml version="1.0" encoding="Windows-1252"?>
<VisualStudioProject
   ProjectType="Visual C++"
   Version="7.00"
   Name="SomeProjName"
   ProjectGUID="{SomeGuid}"
   {all Read/Write properties of VCProject }
   >
   <Platforms>
      {One or more Platform Objects}
      <Platform
         Name="{PlatformName}"/>
   </Platforms>

   <Configurations>
      {any number of Configuration objects }
      <Configuration
         Name="{ConfigName}|{PlatformName}"
         { All read/write properties on VCConfiguration }
         >
         {List of Tool Objects - See below }
      </Configuration>
   </Configurations>
   <References>
      {any number of assembly, COM, or project references }
      <AssemblyReference
         RelativePath="{reference}"/>
      <ActiveXReference
         ControlGUID="{guid}"
         ControlVersion="{version}"
         WrapperTool="{tool}"/>
      <ProjectReference
         ReferencedProjectIdentifier="{project_reference_guid}"
         Name="project_name"/>
   </References>
   <Files>
      { Any number of Filter or File objects }
      <Filter
         { All Read/Write properties of VCFilter }
         { Any Number of File or Filter objects }
      </Filter>
      <File
         RelativePath="{SomePath}"
         {All read/write properties on VCFile }
         >
         {0 FileConfiguration's or 1 FileConfiguration per project configuration }
         <FileConfiguration
            Name="{ConfigName}|{PlatformName}">
            { Tool object }
         </FileConfiguration>
```

```
      </File>
   </Files>
   <Globals>
      { 0 or more global objects }
      <Global
         Name="SomeName"
         Value="SomeValue"
      />
   </Globals>
</VisualStudioProject>


{ List the list of valid tools can include }

<Tool
   Name="VCCLCompilerTool"
   { all read/write properties of VCCLCompilerTool }
/>
<Tool
   Name="VCCustomBuildTool"
   { all read/write properties of VCCustomBuildTool }
/>
<Tool
   Name="VCLinkerTool"
   { all read/write properties of VCLinkerTool }
/>
<Tool
   Name="VCMIDLTool"
   { all read/write properties of VCMIDLTool }
/>
<Tool
   Name="VCPostBuildEventTool"
   { all read/write properties of VCPostBuildEventTool }
/>
<Tool
   Name="VCPreBuildEventTool"/>
   { all read/write properties of VCPreBuildEventTool }
/>
<Tool
   Name="VCPreLinkEventTool"/>
   { all read/write properties of VCPreLinkEventTool }
/>
<Tool
   Name="VCResourceCompilerTool"
   { all read/write properties of VCResourceCompilerTool }
/>
<Tool
   Name="VCWebServiceProxyGeneratorTool"
   { all read/write properties of VCWebServiceProxyGeneratorTool }
/>
<Tool
   Name="VCWebDeploymentTool"
   { all read/write properties of VCWebDeploymentTool }
/>
<Tool
   Name="VCNMakeTool"
   { all read/write properties of VCNMakeTool }
/>
<Tool
   Name="VCLibrarianTool"
   { all read/write properties of VCLibrarianTool }
/>
```

Use the XSD Schema for Project Files to validate your own .vcproj files.

See Also

Reference

Modifying Project Settings

Other Resources

Building C++ Projects in Visual Studio

XSD Schema for Project Files

This topic presents the schema for a Visual C++ project file.

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="VisualStudioProject" type="VisualStudioProjectType" />
   <xs:complexType name="ConfigurationType">
      <xs:sequence>
         <xs:element name="Tool" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
               <xs:attribute name="Name" type="xs:string" use="required" />
               <!-- NOTE: all other attributes are properties of that particular tool objec
t. -->
               <!--       any unrecognized attribute will be ignored. -->
               <xs:anyAttribute processContents="skip" />
            </xs:complexType>
         </xs:element>
         <xs:element name="DeploymentTool" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
               <!-- NOTE: all other attributes are properties of that particular tool objec
t. -->
               <!--       any unrecognized attribute will be ignored. -->
               <xs:anyAttribute processContents="skip" />
            </xs:complexType>
         </xs:element>
      </xs:sequence>
      <xs:attribute name="Name" type="xs:string" use="required" />
      <!-- NOTE: all other attributes are properties of that particular configuration objec
t. -->
      <!--       any unrecognized attribute will be ignored. -->
      <xs:anyAttribute processContents="skip" />
   </xs:complexType>
   <xs:complexType name="FilterType">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
         <xs:element name="Filter" type="FilterType" />
         <xs:element name="File" type="FileType" />
      </xs:choice>
      <xs:attribute name="Name" type="xs:string" use="required" />
      <!-- NOTE: all other attributes are properties of that particular filter object. -->
      <!--       any unrecognized attribute will be ignored. -->
      <xs:anyAttribute processContents="skip" />
   </xs:complexType>
   <xs:complexType name="FolderType">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
         <xs:element name="Folder" type="FolderType" />
         <xs:element name="Filter" type="FilterType" />
         <xs:element name="File" type="FileType" />
      </xs:choice>
      <xs:attribute name="Name" type="xs:string" use="required" />
      <!-- NOTE: all other attributes are properties of that particular folder object. -->
      <!--       any unrecognized attribute will be ignored. -->
      <xs:anyAttribute processContents="skip" />
   </xs:complexType>
   <xs:complexType name="FileType">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
         <xs:element name="File" type="FileType" />
         <xs:element name="FileConfiguration" type="ConfigurationType" />
      </xs:choice>
      <xs:attribute name="RelativePath" type="xs:string" use="required" />
      <!-- NOTE: all other attributes are properties of that particular file object. -->
      <!--       any unrecognized attribute will be ignored. -->
      <xs:anyAttribute processContents="skip" />
   </xs:complexType>
   <xs:complexType name="VisualStudioProjectType">
```

```
      <xs:sequence>
         <xs:element name="Platforms" minOccurs="1" maxOccurs="1">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="Platform" minOccurs="1" maxOccurs="unbounded">
                     <xs:complexType>
                        <!-- NOTE: the following attribute must match an existing platform
on the system. -->
                        <xs:attribute name="Name" type="xs:string" use="required" />
                     </xs:complexType>
                  </xs:element>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element name="ToolFiles" minOccurs="0" maxOccurs="1">
             <xs:complexType>
                 <xs:choice minOccurs="0" maxOccurs="unbounded">
                     <xs:element name="ToolFile" minOccurs="0" maxOccurs="unbounded">
                         <xs:complexType>
                             <xs:attribute name="RelativePath" type="xs:string" use="requir
ed" />
                         </xs:complexType>
                     </xs:element>
                     <xs:element name="DefaultToolFile" minOccurs="0" maxOccurs="unbounded"
>
                         <xs:complexType>
                             <xs:attribute name="FileName" type="xs:string" use="required"
/>
                         </xs:complexType>
                     </xs:element>
                 </xs:choice>
             </xs:complexType>
         </xs:element>
         <xs:element name="Configurations" minOccurs="1" maxOccurs="1">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="Configuration" minOccurs="1" maxOccurs="unbounded" type
="ConfigurationType" />
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element name="References" minOccurs="0" maxOccurs="1">
            <xs:complexType>
               <xs:choice minOccurs="0" maxOccurs="unbounded">
                  <xs:element name="AssemblyReference">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element name="ReferenceConfiguration" minOccurs="0" maxOccur
s="unbounded" type="ConfigurationType" />
                        </xs:sequence>
                        <xs:attribute name="RelativePath" type="xs:string" use="required" /
>
                        <xs:attribute name="AssemblyName" type="xs:string" use="optional" /
>
                        <xs:attribute name="CopyLocal" type="xs:boolean" use="optional" />
                        <xs:attribute name="UseInBuild" type="xs:boolean" use="optional" de
fault="true" />
                     </xs:complexType>
                  </xs:element>
                  <xs:element name="ProjectReference">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element name="ReferenceConfiguration" minOccurs="0" maxOccur
s="unbounded" type="ConfigurationType" />
                        </xs:sequence>
                        <xs:attribute name="ReferencedProjectIdentifier" type="xs:string" u
se="required" />
                        <xs:attribute name="Name" type="xs:string" use="optional" />
```

```
                        <xs:attribute name="CopyLocal" type="xs:string" use="optional" />
                        <xs:attribute name="UseInBuild" type="xs:boolean" use="optional" de
fault="true" />
                        <xs:attribute name="RelativePathFromSolution" type="xs:string" use=
"optional" />
                     </xs:complexType>
                  </xs:element>
                  <xs:element name="ActiveXReference">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element name="ReferenceConfiguration" minOccurs="0" maxOccur
s="unbounded" type="ConfigurationType" />
                        </xs:sequence>
                        <xs:attribute name="ControlGUID" type="xs:string" use="required" />
                        <xs:attribute name="ControlVersion" type="xs:float" use="required"
/>
                        <xs:attribute name="LocaleID" type="xs:string" use="optional" />
                        <xs:attribute name="WrapperTool" type="xs:string" use="optional" />
                        <xs:attribute name="CopyLocal" type="xs:string" use="optional" />
                        <xs:attribute name="UseInBuild" type="xs:boolean" use="optional" de
fault="true" />
                     </xs:complexType>
                  </xs:element>
               </xs:choice>
            </xs:complexType>
         </xs:element>
         <xs:element name="Files" minOccurs="0" maxOccurs="1">
            <xs:complexType>
               <xs:choice minOccurs="0" maxOccurs="unbounded">
                  <xs:element name="Folder" type="FolderType" />
                  <xs:element name="Filter" type="FilterType" />
                  <xs:element name="File" type="FileType" />
               </xs:choice>
            </xs:complexType>
         </xs:element>
         <xs:element name="Globals" minOccurs="0" maxOccurs="1">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="Global" minOccurs="0" maxOccurs="unbounded">
                     <xs:complexType>
                        <xs:attribute name="Name" type="xs:string" use="required" />
                        <xs:attribute name="Value" type="xs:string" use="required" />
                     </xs:complexType>
                  </xs:element>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
      <!-- NOTE: the ProjectType attribute is only for readability. -->
      <xs:attribute name="ProjectType" type="xs:string" fixed="Visual C++" use="optional" /
>
      <xs:attribute name="Version" use="optional" default="7.00">
         <xs:simpleType>
            <xs:restriction base="xs:string">
               <xs:enumeration value="7.00" />
               <xs:enumeration value="7,00" />
               <xs:enumeration value="7.10" />
               <xs:enumeration value="7,10" />
               <xs:enumeration value="8.00" />
               <xs:enumeration value="8,00" />
            </xs:restriction>
         </xs:simpleType>
      </xs:attribute>
      <!-- NOTE: if the ProjectGUID attribute is missing, a new GUID will be generated. -->
      <xs:attribute name="ProjectGUID" type="xs:string" use="optional" />
      <xs:attribute name="RootNamespace" type="xs:string" use="optional" />
      <xs:attribute name="Keyword" type="xs:string" use="optional" />
      <!-- NOTE: if the Name attribute is missing, one will be created based on the .vcproj
```

```
  filename. -->
        <xs:attribute name="Name" type="xs:string" use="optional" />
        <!-- NOTE: all other attributes matched as project properties will be applied. -->
        <!--       any unrecognized attribute will be ignored. -->
        <xs:anyAttribute processContents="skip" />
     </xs:complexType>
  </xs:schema>
```

See Also

Reference

Modifying Project Settings

Building on the Command Line

Visual C++ provides command-line tools for programmers who prefer to build their applications from the command prompt. If you want to use the command line to build a project created in Visual C++, you can use one of the following:

- DEVENV.EXE
- NMAKE.EXE
- VCBUILD Reference

When you build from the command line, you can get help on warnings, errors, and messages by starting the development environment and clicking either **Index** or **Search** on the **Help** menu.

In This Section

Setting the Path and Environment Variables for Command-Line Builds

Describes using VCVARS32.BAT to set the path and environment variables.

NMAKE Reference

Provides links to topics discussing the Microsoft Program Maintenance Utility (NMAKE.EXE).

VCBUILD Reference

Provides links to topics discussing VCBUILD.EXE.

Related Sections

/MD, /MT, /LD (Use Run-Time Library)

Describes using these compiler options to use a Debug or Release run-time library.

C/C++ Compiler Options

Provides links to topics discussing the C and C++ compiler options and CL.EXE.

Linker Options

Provides links to topics discussing the linker options and LINK.EXE.

C/C++ Build Tools

Provides links to the C/C++ build tools provided with Visual C++.

Building a C/C++ Program

Provides links to topics describing building your program from the command line or from the integrated development environment of Visual Studio.

C# Compiler Options

Provides links to topics discussing the C# compiler options provided with Visual Studio.

Setting the Path and Environment Variables for Command-Line Builds

The vcvars32.bat file sets the appropriate environment variables to enable 32-bit command-line builds. For information on how to enable a 64-bit toolset at the command line, see How to: Enable a 64-Bit Visual C++ Toolset at the Command Line.

Note that if you are doing command-line builds with DEVENV, vcvars32 settings will have no affect on your builds, unless you also specify the /useenv option.

To run vcvars32.bat

1. At the command prompt, change to the \bin subdirectory of your Visual C++ installation.
2. Run vcvars32.bat by typing `VCVARS32`.

Caution
The vcvars32.bat file can vary from computer to computer. Do not replace a missing or damaged vcvars32.bat file with a vcvars32.bat from another computer. Rerun Visual C++ Setup to replace the missing file.

For more information about vcvars32.bat, see the following Knowledge Base article:

- Q248802 : Vcvars32.bat Generates Out of Environment Message

If the current version of Visual C++ is installed on a computer that also has a previous version of Visual C++, you should not run vcvars32.bat from different versions in the same command window.

CL.exe invokes the preprocessor and compiler, and LINK.exe invokes the linker. (CL can invoke the linker as well, so you do not have to invoke LINK directly in most cases.)

CL.exe, LINK.exe, and other command-line tools require that certain environment variables be properly set:

- CL uses CL and INCLUDE.
- LINK uses LINK, LIB, PATH, and TMP.

See Also

Reference

Linker Options

Compiler Options

Concepts

Linking

Other Resources

Building on the Command Line

Compiling a C/C++ Program

NMAKE Reference

The Microsoft Program Maintenance Utility (NMAKE.EXE) is a tool that builds projects based on commands contained in a description file.

What do you want to know more about?

Running NMAKE

Contents of a Makefile

Description blocks

Commands in a Makefile

Macros and NMAKE

Inference rules

Dot directives

Makefile preprocessing

See Also

Reference

C/C++ Build Tools

Other Resources

Building on the Command Line

Creating and Managing Visual C++ Projects

Debugging in Visual Studio

C/C++ Building Reference

Running NMAKE

```
NMAKE [option...] [macros...] [targets...] [@commandfile...]
```

Remarks

NMAKE builds only specified *targets* or, if none is specified, the first target in the makefile. The first makefile target can be a pseudotarget that builds other targets. NMAKE uses makefiles specified with /F; if /F is not specified, it uses the Makefile file in the current directory. If no makefile is specified, it uses inference rules to build command-line *targets*.

The *commandfile* text file (or response file) contains command-line input. Other input can precede or follow @*commandfile*. A path is permitted. In *commandfile*, line breaks are treated as spaces. Enclose macro definitions in quotation marks if they contain spaces.

What do you want to know more about?

NMAKE options

Tools.ini and NMAKE

Exit codes from NMAKE

See Also

Concepts

NMAKE Reference

NMAKE Options

NMAKE options are described in the following table. Options are preceded by either a slash (/) or a dash (–) and are not case sensitive. Use !CMDSWITCHES to change option settings in a makefile or in Tools.ini.

Option	Purpose
/A	Forces build of all evaluated targets, even if not out-of-date with respect to dependents. Does not force build of unrelated targets.
/B	Forces build even if timestamps are equal. Recommended only for very fast systems (resolution of two seconds or less).
/C	Suppresses default output, including nonfatal NMAKE errors or warnings, timestamps, and NMAKE copyright message. Suppresses warnings issued by /K.
/D	Displays timestamps of each evaluated target and dependent and a message when a target does not exist. Useful with /P for debugging a makefile. Use **!CMDSWITCHES** to set or clear /D for part of a makefile.
/E	Causes environment variables to override makefile macro definitions.
/ERRORREPORT[NONE | PROMPT | QUEUE | SEND]	If nmake.exe fails at runtime, you can use /ERRORREPORT to send information to Microsoft about these internal errors. For more information about /ERRORREPORT, see /errorReport (Report Internal Compiler Errors).
/F *filename*	Specifies *filename* as a makefile. Spaces or tabs can precede *filename*. Specify /F once for each makefile. To supply a makefile from standard input, specify a dash (–) for *filename,* and end keyboard input with either F6 or CTRL+Z.
/G	Displays the makefiles included with the !INCLUDE directive. See Makefile Preprocessing Directives for more information.
/HELP, /?	Displays a brief summary of NMAKE command-line syntax.
/I	Ignores exit codes from all commands. To set or clear /I for part of a makefile, use **!CMDSWITCHES**. To ignore exit codes for part of a makefile, use a dash (–) command modifier or .IGNORE. Overrides /K if both are specified.
/K	Continues building unrelated dependencies, if a command returns an error. Also issues a warning and returns an exit code of 1. By default, NMAKE halts if any command returns a nonzero exit code. Warnings from /K are suppressed by /C; /I overrides /K if both are specified.
/N	Displays but does not execute commands; preprocessing commands are executed. Does not display commands in recursive NMAKE calls. Useful for debugging makefiles and checking timestamps. To set or clear /N for part of a makefile, use **!CMDSWITCHES**.
/NOLOGO	Suppresses the NMAKE copyright message.
/P	Displays information (macro definitions, inference rules, targets, .SUFFIXES list) to standard output, and then runs the build. If no makefile or command-line target exists, it displays information only. Use with /D to debug a makefile.
/Q	Checks timestamps of targets; does not run the build. Returns a zero exit code if all targets are up-to-date and a nonzero exit code if any target is not. Preprocessing commands are executed. Useful when running NMAKE from a batch file.

/R	Clears the **.SUFFIXES** list and ignores inference rules and macros that are defined in the Tools.ini file or that are predefined.
/S	Suppresses display of executed commands. To suppress display in part of a makefile, use the **@** command modifier or .SILENT. To set or clear /S for part of a makefile, use **!CMDSWITCHES**.
/T	Updates timestamps of command-line targets (or first makefile target) and executes preprocessing commands but does not run the build.
/U	Must be used in conjunction with /N. Dumps inline NMAKE files so that the /N output can be used as a batch file.
/X *filename*	Sends NMAKE error output to *filename* instead of standard error. Spaces or tabs can precede *filename*. To send error output to standard output, specify a dash (–) for *filename*. Does not affect output from commands to standard error.
/Y	Disables batch-mode inference rules. When this option is selected, all batch-mode inference rules are treated as regular inference rules.

See Also

Reference

Running NMAKE

Tools.ini and NMAKE

NMAKE reads Tools.ini before it reads makefiles, unless /R is used. It looks for Tools.ini first in the current directory and then in the directory specified by the INIT environment variable. The section for NMAKE settings in the initialization file begins with `[NMAKE]` and can contain any makefile information. Specify a comment on a separate line beginning with a number sign (#).

See Also

Reference

Running NMAKE

Exit Codes from NMAKE

NMAKE returns the following exit codes:

Code	Meaning
0	No error (possibly a warning)
1	Incomplete build (issued only when /K is used)
2	Program error, possibly due to one of the following:
	• A syntax error in the makefile
	• An error or exit code from a command
	• An interruption by the user
4	System error — out of memory
255	Target is not up-to-date (issued only when /Q is used)

See Also

Reference

Running NMAKE

Contents of a Makefile

A makefile contains:

- Description blocks
- Commands
- Macros
- Inference rules
- Dot directives
- Preprocessing directives

Remarks

Other features you can use in a makefile are wildcards, long filenames, comments, and special characters.

For a sample, see Sample Makefile.

See Also

Concepts

NMAKE Reference

Wildcards and NMAKE

NMAKE expands filename wildcards (* and ?) in dependency lines. A wildcard specified in a command is passed to the command; NMAKE does not expand it.

See Also

Concepts

Contents of a Makefile

Long Filenames in a Makefile

Enclose long filenames in double quotation marks, as follows:

```
all : "VeryLongFileName.exe"
```

See Also

Concepts

Contents of a Makefile

Comments in a Makefile

Precede a comment with a number sign (#). NMAKE ignores text from the number sign to the next newline character. Examples:

```
# Comment on line by itself
OPTIONS = /MAP  # Comment on macro definition line

all.exe : one.obj two.obj  # Comment on dependency line
    link one.obj two.obj
# Comment in commands block
#   copy *.obj \objects  # Command turned into comment
    copy one.exe \release

.obj.exe:  # Comment on inference rule line
    link $<

my.exe : my.obj ; link my.obj  # Err: cannot comment this
 # Error: # must be the first character
.obj.exe: ; link $<  # Error: cannot comment this
```

To specify a literal number sign, precede it with a caret (**^**), as follows:

```
DEF = ^#define  #Macro for a C preprocessing directive
```

See Also

Concepts

Contents of a Makefile

Special Characters in a Makefile

To use an NMAKE special character as a literal character, place a caret (^) in front of it. NMAKE ignores carets that precede other characters. The special characters are:

```
: ; # ( ) $ ^ \ { } ! @ —
```

A caret (^) within a quoted string is treated as a literal caret character. A caret at the end of a line inserts a literal newline character in a string or macro.

In macros, a backslash (\) followed by a newline character is replaced by a space.

In commands, a percent symbol (%) is a file specifier. To represent % literally in a command, specify a double percent sign (%%) in place of a single one. In other situations, NMAKE interprets a single % literally, but it always interprets a double %% as a single %. Therefore, to represent a literal %%, specify either three percent signs, %%%, or four percent signs, %%%%.

To use the dollar sign ($) as a literal character in a command, specify two dollar signs ($$). This method can also be used in other situations where ^$ works.

See Also

Concepts

Contents of a Makefile

Sample Makefile

This topic contains a sample makefile.

Sample

```
# Sample makefile

!include <win32.mak>

all: simple.exe challeng.exe

.c.obj:
  $(cc) $(cdebug) $(cflags) $(cvars) $*.c

simple.exe: simple.obj
  $(link) $(ldebug) $(conflags) -out:simple.exe simple.obj $(conlibs) lsapi32.lib

challeng.exe: challeng.obj md4c.obj
  $(link) $(ldebug) $(conflags) -out:challeng.exe $** $(conlibs) lsapi32.lib
```

See Also

Concepts

Contents of a Makefile

Description Blocks

A description block is a dependency line optionally followed by a commands block:

```
targets... : dependents...
    commands...
```

A dependency line specifies one or more targets and zero or more dependents. A target must be at the start of the line. Separate targets from dependents by a colon (:); spaces or tabs are allowed. To split the line, use a backslash (\ ) after a target or dependent. If a target does not exist, has an earlier timestamp than a dependent, or is a pseudotarget, NMAKE executes the commands. If a dependent is a target elsewhere and does not exist or is out-of-date with respect to its own dependents, NMAKE updates the dependent before updating the current dependency.

What do you want to know more about?

Targets

Dependents

See Also

Concepts

NMAKE Reference

Targets

In a dependency line, specify one or more targets, using any valid filename, directory name, or pseudotarget. Separate multiple targets with one or more spaces or tabs. Targets are not case sensitive. Paths are permitted with filenames. A target cannot exceed 256 characters. If the target preceding the colon is a single character, use a separating space; otherwise, NMAKE interprets the letter-colon combination as a drive specifier.

What do you want to know more about?

Pseudotargets

Multiple targets

Cumulative dependencies

Targets in multiple description blocks

Dependency side effects

See Also

Concepts

Description Blocks

Pseudotargets

A pseudotarget is a label used in place of a filename in a dependency line. It is interpreted as a file that does not exist, and so is out-of-date. NMAKE assumes a pseudotarget's timestamp is the most recent of all its dependents. If it has no dependents, the current time is assumed. If a pseudotarget is used as a target, its commands are always executed. A pseudotarget used as a dependent must also appear as a target in another dependency. However, that dependency does not need to have a commands block.

Pseudotarget names follow the filename syntax rules for targets. However, if the name does not have an extension (that is, does not contain a period), it can exceed the 8-character limit for filenames and can be up to 256 characters long.

See Also

Concepts

Targets

Multiple Targets

NMAKE evaluates multiple targets in a single dependency as if each were specified in a separate description block.

This...	...is evaluated as this

`bounce.exe leap.exe :`
`jump.obj`
`  echo Building...`
	`bounce.exe : jump.obj`
	`  echo Building...`
	`leap.exe : jump.obj`
	`  echo Building...`

See Also

Concepts

Targets

Cumulative Dependencies

Dependencies are cumulative in a description block if a target is repeated.

This...	...is evaluated as this

`bounce.exe : jump.obj`
`bounce.exe : up.obj`
`  echo Building`
`bounce.exe...`
	`bounce.exe : jump.obj`
	`up.obj`
	`  echo Building`
	`bounce.exe...`

Multiple targets in multiple dependency lines in a single description block are evaluated as if each were specified in a separate description block, but targets that are not in the last dependency line do not use the commands block.

This...	...is evaluated as this

`bounce.exe leap.exe :`
`jump.obj`
`bounce.exe climb.exe :`
`up.obj`
`  echo Building...`
	`bounce.exe : jump.obj`
	`up.obj`
	`  echo Building`
	`bounce.exe...`
	`climb.exe : up.obj`
	`  echo Building`
	`climb.exe...`
	`leap.exe : jump.obj`
	`# invokes an inference rule`

See Also

Concepts

Targets

Targets in Multiple Description Blocks

To update a target in more than one description block using different commands, specify two consecutive colons (::) between targets and dependents.

```
target.lib :: one.asm two.asm three.asm
    ml one.asm two.asm three.asm
    lib target one.obj two.obj three.obj
target.lib :: four.c five.c
    cl /c four.c five.c
    lib target four.obj five.obj
```

See Also

Concepts

Targets

Dependency Side Effects

If a target is specified with a colon (:) in two dependency lines in different locations, and if commands appear after only one of the lines, NMAKE interprets the dependencies as if adjacent or combined. It does not invoke an inference rule for the dependency that has no commands, but instead assumes that the dependencies belong to one description block and executes the commands specified with the other dependency.

This...	...is evaluated as this

`bounce.exe : jump.obj`
`  echo Building`
`bounce.exe...`
 	
`bounce.exe : up.obj`
	`bounce.exe : jump.obj`
	`up.obj`
	`  echo Building`
	`bounce.exe...`

This effect does not occur if a double colon (**::**) is used.

This...	...is evaluated as this

`bounce.exe :: jump.obj`
`  echo Building`
`bounce.exe...`
 	
`bounce.exe :: up.obj`
	`bounce.exe : jump.obj`
	`  echo Building`
	`bounce.exe...`
	
	`bounce.exe : up.obj`
	`# invokes an inference rule`

See Also

Concepts

Targets

Dependents

In a dependency line, specify zero or more dependents after the colon (:) or double colon (::), using any valid filename or pseudotarget. Separate multiple dependents with one or more spaces or tabs. Dependents are not case sensitive. Paths are permitted with filenames.

What do you want to know more about?

Inferred dependents

Search paths for dependents

See Also

Concepts

Description Blocks

Inferred Dependents

An inferred dependent is derived from an inference rule and is evaluated before explicit dependents. If an inferred dependent is out-of-date with respect to its target, NMAKE invokes the commands block for the dependency. If an inferred dependent does not exist or is out-of-date with respect to its own dependents, NMAKE first updates the inferred dependent. For more information about inferred dependents, see Inference Rules.

See Also

Concepts

Dependents

Search Paths for Dependents

Each dependent has an optional search path, specified as follows:

Syntax

```
{directory[;directory...]}dependent
```

Remarks

NMAKE looks for a dependent first in the current directory, and then in directories in the order specified. A macro can specify part or all of a search path. Enclose directory names in braces ({ }); separate multiple directories with a semicolon (;). No spaces or tabs are allowed.

See Also

Concepts

Dependents

Commands in a Makefile

A description block or inference rule specifies a block of commands to run if the dependency is out-of-date. NMAKE displays each command before running it, unless /S, **.SILENT**, **!CMDSWITCHES**, or @ is used. NMAKE looks for a matching inference rule if a description block is not followed by a commands block.

A commands block contains one or more commands, each on its own line. No blank line can appear between the dependency or rule and the commands block. However, a line containing only spaces or tabs can appear; this line is interpreted as a null command, and no error occurs. Blank lines are permitted between command lines.

A command line begins with one or more spaces or tabs. A backslash (\ ) followed by a newline character is interpreted as a space in the command; use a backslash at the end of a line to continue a command onto the next line. NMAKE interprets the backslash literally if any other character, including a space or tab, follows the backslash.

A command preceded by a semicolon (;) can appear on a dependency line or inference rule, whether or not a commands block follows:

```
project.obj : project.c project.h ; cl /c project.c
```

What do you want to know more about?

Command modifiers

Filename-parts syntax

Inline files in a makefile

See Also

Concepts

NMAKE Reference

Command Modifiers

You can specify one or more command modifiers preceding a command, optionally separated by spaces or tabs. As with commands, modifiers must be indented.

Modifier	Purpose
@*command*	Prevents display of the command. Display by commands is not suppressed. By default, NMAKE echoes all executed commands. Use /S to suppress display for the entire makefile; use **.SILENT** to suppress display for part of the makefile.
–[*number*]*command*	Turns off error checking for *command*. By default, NMAKE halts when a command returns a nonzero exit code. If –*number* is used, NMAKE stops if the exit code exceeds *number*. Spaces or tabs cannot appear between the dash and *number*. At least one space or tab must appear between *number* and *command*. Use /I to turn off error checking for the entire makefile; use **.IGNORE** to turn off error checking for part of the makefile.
! *command*	Executes *command* for each dependent file if *command* uses **$**** (all dependent files in the dependency) or **$?** (all dependent files in the dependency with a later timestamp than the target).

See Also

Concepts

Commands in a Makefile

Filename-Parts Syntax

Filename-parts syntax in commands represents components of the first dependent filename (which may be an implied dependent). Filename components are the file's drive, path, base name, and extension as specified, not as it exists on disk. Use **%s** to represent the complete filename. Use **%|**[*parts*]**F** (a vertical bar character follows the percent symbol) to represent parts of the filename, where *parts* can be zero or more of the following letters, in any order.

Letter	Description
No letter	Complete name (same as **%s**)
d	Drive
p	Path
f	File base name
e	File extension

For example, if the filename is c:\prog.exe:

- %s will be c:\prog.exe
- %|F will be c:\prog.exe
- %|dF will be c
- %|pF will be c:\
- %|fF will be prog
- %|eF will be exe

See Also

Concepts

Commands in a Makefile

Inline Files in a Makefile

An inline file contains text you specify in the makefile. Its name can be used in commands as input (for example, a LINK command file), or it can pass commands to the operating system. The file is created on disk when a command that creates the file is run.

What do you want to know more about?

Specifying an inline file

Creating inline file text

Reusing inline files

Multiple inline files

See Also

Concepts

Commands in a Makefile

Specifying an Inline File

Specify two angle brackets (<<) in the command where *filename* is to appear. The angle brackets cannot be a macro expansion.

Syntax

```
<<[filename]
```

Remarks

When the command is run, the angle brackets are replaced by *filename*, if specified, or by a unique NMAKE-generated name. If specified, *filename* must follow angle brackets without a space or tab. A path is permitted. No extension is required or assumed. If *filename* is specified, the file is created in the current or specified directory, overwriting any existing file by that name; otherwise, it is created in the TMP directory (or the current directory, if the TMP environment variable is not defined). If a previous *filename* is reused, NMAKE replaces the previous file.

See Also

Concepts

Inline Files in a Makefile

Creating Inline File Text

Inline files are temporary or permanent.

```
inlinetext
.
.
.
<<[KEEP | NOKEEP]
```

Remarks

Specify *inlinetext* on the first line after the command. Mark the end with double angle brackets (<<) at the beginning of a separate line. The file contains all *inlinetext* before the delimiting brackets. The *inlinetext* can have macro expansions and substitutions, but not directives or makefile comments. Spaces, tabs, and newline characters are treated literally.

A temporary file exists for the duration of the session and can be reused by other commands. Specify **KEEP** after the closing angle brackets to retain the file after the NMAKE session; an unnamed file is preserved on disk with the generated filename. Specify **NOKEEP** or nothing for a temporary file. **KEEP** and **NOKEEP** are not case sensitive.

See Also

Concepts

Inline Files in a Makefile

Reusing Inline Files

To reuse an inline file, specify <<*filename* where the file is defined and first used, then reuse *filename* without << later in the same or another command. The command to create the inline file must run before all commands that use the file.

See Also

Concepts

Inline Files in a Makefile

Multiple Inline Files

A command can create more than one inline file.

```
command << <<
inlinetext
<<[KEEP | NOKEEP]
inlinetext
<<[KEEP | NOKEEP]
```

Remarks

For each file, specify one or more lines of inline text followed by a closing line containing the delimiter. Begin the second file's text on the line following the delimiting line for the first file.

See Also

Concepts

Inline Files in a Makefile

Macros and NMAKE

Macros replace a particular string in the makefile with another string. Using macros, you can:

- Create a makefile that can build different projects.
- Specify options for commands.
- Set environment variables.

You can define your own macros or use NMAKE's predefined macros.

What do you want to know more about?

Defining an NMAKE macro

Using an NMAKE macro

Special NMAKE macros

See Also

Concepts

NMAKE Reference

Defining an NMAKE Macro

```
macroname=string
```

Remarks

The *macroname* is a combination of letters, digits, and underscores (_) up to 1,024 characters, and is case sensitive. The *macroname* can contain an invoked macro. If *macroname* consists entirely of an invoked macro, the macro being invoked cannot be null or undefined.

The *string* can be any sequence of zero or more characters. A null string contains zero characters or only spaces or tabs. The *string* can contain a macro invocation.

What do you want to know more about?

Special characters in macros

Null and undefined macros

Where to define macros

Precedence in macro definitions

See Also

Concepts

Macros and NMAKE

Special Characters in Macros

A number sign (#) after a definition specifies a comment. To specify a literal number sign in a macro, use a caret (^), as in ^#.

A dollar sign ($) specifies a macro invocation. To specify a literal $, use $$.

To extend a definition to a new line, end the line with a backslash (\). When the macro is invoked, the backslash plus newline character is replaced with a space. To specify a literal backslash at the end of the line, precede it with a caret (^), or follow it with a comment specifier (#).

To specify a literal newline character, end the line with a caret (^), as in:

```
CMDS = cls^
dir
```

See Also

Reference

Defining an NMAKE Macro

Null and Undefined Macros

Both null and undefined macros expand to null strings, but a macro defined as a null string is considered defined in preprocessing expressions. To define a macro as a null string, specify no characters except spaces or tabs after the equal sign (=) in a command line or command file, and enclose the null string or definition in double quotation marks (" "). To undefine a macro, use **!UNDEF.** For more information, see Makefile Preprocessing Directives.

See Also

Reference

Defining an NMAKE Macro

Where to Define Macros

Define macros in a command line, command file, makefile, or the Tools.ini file.

In a makefile or the Tools.ini file, each macro definition must appear on a separate line and cannot start with a space or tab. Spaces or tabs around the equal sign are ignored. All string characters are literal, including surrounding quotation marks and embedded spaces.

In a command line or command file, spaces and tabs delimit arguments and cannot surround the equal sign. If *string* has embedded spaces or tabs, enclose either the string itself or the entire macro in double quotation marks (" ").

See Also

Reference

Defining an NMAKE Macro

Precedence in Macro Definitions

If a macro has multiple definitions, NMAKE uses the highest-precedence definition. The following list shows the order of precedence, from highest to lowest:

1. A macro defined on the command line
2. A macro defined in a makefile or include file
3. An inherited environment-variable macro
4. A macro defined in the Tools.ini file
5. A predefined macro, such as CC and AS

Use /E to cause macros inherited from environment variables to override makefile macros with the same name. Use **!UNDEF** to override a command line.

See Also

Reference

Defining an NMAKE Macro

Using an NMAKE Macro

To use a macro, enclose its name in parentheses preceded by a dollar sign ($) as follows.

Syntax

```
$(macroname)
```

Remarks

No spaces are allowed. The parentheses are optional if *macroname* is a single character. The definition string replaces $(*macroname*); an undefined macro is replaced by a null string.

What do you want to know more about?

Macro substitution

See Also

Concepts

Macros and NMAKE

Macro Substitution

When *macroname* is invoked, each occurrence of *string1* in its definition string is replaced by *string2*.

Syntax

```
$(macroname:string1=string2)
```

Remarks

Macro substitution is case sensitive and is literal; *string1* and *string2* cannot invoke macros. Substitution does not modify the original definition. You can substitute text in any predefined macro except `$$@`.

No spaces or tabs precede the colon; any after the colon are interpreted as literal. If *string2* is null, all occurrences of *string1* are deleted from the macro's definition string.

See Also

Concepts

Using an NMAKE Macro

Special NMAKE Macros

NMAKE provides several special macros to represent various filenames and commands. One use for some of these macros is in the predefined inference rules. Like all macros, the macros provided by NMAKE are case sensitive.

- Filename macros
- Recursion macros
- Command macros and options macros
- Environment-Variable Macros

See Also

Concepts

Macros and NMAKE

Filename Macros

Filename macros are predefined as filenames specified in the dependency (not full filename specifications on disk). These macros do not need to be enclosed in parentheses when invoked; specify only a $ as shown.

Macro	Meaning
$@	Current target's full name (path, base name, extension), as currently specified.
$$@	Current target's full name (path, base name, extension), as currently specified. Valid only as a dependent in a dependency.
$*	Current target's path and base name minus file extension.
$**	All dependents of the current target.
$?	All dependents with a later timestamp than the current target.
$<	Dependent file with a later timestamp than the current target. Valid only in commands in inference rules.

To specify part of a predefined filename macro, append a macro modifier and enclose the modified macro in parentheses.

Modifier	Resulting filename part
D	Drive plus directory
B	Base name
F	Base name plus extension
R	Drive plus directory plus base name

See Also

Concepts

Special NMAKE Macros

Recursion Macros

Use recursion macros to call NMAKE recursively. Recursive sessions inherit command-line and environment-variable macros and Tools.ini information. They do not inherit makefile-defined inference rules or **.SUFFIXES** and **.PRECIOUS** specifications. To pass macros to a recursive NMAKE session, either set an environment variable with SET before the recursive call, define a macro in the command for the recursive call, or define a macro in Tools.ini.

Macro	Definition
MAKE	Command used originally to invoke NMAKE. The $(MAKE) macro gives the full path to nmake.exe.
MAKEDIR	Current directory when NMAKE was invoked.
MAKEFLAGS	Options currently in effect. Use as `/$(MAKEFLAGS)`. Note, /F is not included.

See Also

Concepts

Special NMAKE Macros

Command Macros and Options Macros

Command macros are predefined for Microsoft products. Options macros represent options to these products and are undefined by default. Both are used in predefined inference rules and can be used in description blocks or user-defined inference rules. Command macros can be redefined to represent part or all of a command line, including options. Options macros generate a null string if left undefined.

Microsoft product	Command macro	Defined as	Options macro
Macro Assembler	**AS**	ml	**AFLAGS**
Basic Compiler	**BC**	bc	**BFLAGS**
C Compiler	**CC**	cl	**CFLAGS**
C++ Compiler	**CPP**	cl	**CPPFLAGS**
C++ Compiler	**CXX**	cl	**CXXFLAGS**
Resource Compiler	**RC**	rc	**RFLAGS**

See Also

Concepts

Special NMAKE Macros

Environment-Variable Macros

NMAKE inherits macro definitions for environment variables that exist before the start of the session. If a variable was set in the operating-system environment, it is available as an NMAKE macro. The inherited names are converted to uppercase. Inheritance occurs before preprocessing. Use the /E option to cause macros inherited from environment variables to override any macros with the same name in the makefile.

Environment-variable macros can be redefined in the session, and this changes the corresponding environment variable. You can also change environment variables with the SET command. Using the SET command to change an environment variable in a session does not change the corresponding macro, however.

For example:

```
PATH=$(PATH);\nonesuch

all:
    echo %PATH%
```

In this example, changing `PATH` changes the corresponding environment variable `PATH`; it appends `\nonesuch` to your path.

If an environment variable is defined as a string that would be syntactically incorrect in a makefile, no macro is created and no warning is generated. If a variable's value contains a dollar sign ($), NMAKE interprets it as the beginning of a macro invocation. Using the macro can cause unexpected behavior.

See Also

Concepts

Special NMAKE Macros

Inference Rules

Inference rules supply commands to update targets and to infer dependents for targets. Extensions in an inference rule match a single target and dependent that have the same base name. Inference rules are user-defined or predefined; predefined rules can be redefined.

If an out-of-date dependency has no commands, and if .SUFFIXES contains the dependent's extension, NMAKE uses a rule whose extensions match the target and an existing file in the current or specified directory. If more than one rule matches existing files, the **.SUFFIXES** list determines which to use; list priority descends from left to right. If a dependent file does not exist and is not listed as a target in another description block, an inference rule can create the missing dependent from another file with the same base name. If a description block's target has no dependents or commands, an inference rule can update the target. Inference rules can build a command-line target even if no description block exists. NMAKE may invoke a rule for an inferred dependent even if an explicit dependent is specified.

What do you want to know more about?

Defining a rule

Batch-mode rules

Predefined rules

Inferred dependents and rules

Precedence in inference rules

See Also

Concepts

NMAKE Reference

Defining a Rule

The *fromext* represents the extension of a dependent file, and *toext* represents the extension of a target file.

```
.fromext.toext:
   commands
```

Remarks

Extensions are not case sensitive. Macros can be invoked to represent *fromext* and *toext*; the macros are expanded during preprocessing. The period (.) preceding *fromext* must appear at the beginning of the line. The colon (:) is preceded by zero or more spaces or tabs. It can be followed only by spaces or tabs, a semicolon (;) to specify a command, a number sign (#) to specify a comment, or a newline character. No other spaces are allowed. Commands are specified as in description blocks.

What do you want to know more about?

Search paths in rules

See Also

Concepts

Inference Rules

Search Paths in Rules

```
{frompath}.fromext{topath}.toext:
   commands
```

Remarks

An inference rule applies to a dependency only if paths specified in the dependency exactly match the inference-rule paths. Specify the dependent's directory in *frompath* and the target's directory in *topath*; no spaces are allowed. Specify only one path for each extension. A path on one extension requires a path on the other. To specify the current directory, use either a period (.) or empty braces ({ }). Macros can represent *frompath* and *topath*; they are invoked during preprocessing.

Example

```
{dbi\}.cpp{$(ODIR)}.obj::
        $(CC) $(CFLAGS) $(YUDBI) $<

{ilstore\}.cpp{$(ODIR)}.obj::
        $(CC) $(CFLAGS) $<

{misc\}.cpp{$(ODIR)}.obj::
        $(CC) $(CFLAGS) $(YUPDB) $<

{misc\}.c{$(ODIR)}.obj::
        $(CC) $(CFLAGS) $<

{msf\}.cpp{$(ODIR)}.obj::
        $(CC) $(CFLAGS) $<

{bsc\}.cpp{$(ODIR)}.obj::
        $(CC) $(CFLAGS) $(YUPDB) $<

{mre\}.cpp{$(ODIR)}.obj::
        $(CC) $(CFLAGS) $(YUPDB) $<

{namesrvr\}.cpp{$(ODIR)}.obj::
        $(CC) $(CFLAGS) $(YUPDB) $<

{src\cvr\}.cpp{$(ODIR)}.obj::
        $(CC) $(CFLAGS) $<
```

See Also

Reference

Defining a Rule

Batch-Mode Rules

```
{frompath}.fromext{topath}.toext::
   commands
```

Batch-mode inference rules provide only one invocation of the inference rule when N commands go through this inference rule. Without batch-mode inference rules, it would require N commands to be invoked. N is the number of dependents that trigger the inference rule.

Makefiles that contain batch-mode inference rules must use NMAKE version 1.62 or higher. To check the NMAKE version, run the _NMAKE_VER macro available with NMAKE version 1.62 or higher. This macro returns a string representing the Visual C++ product version.

The only syntactical difference from the standard inference rule is that the batch-mode inference rule is terminated with a double colon (::).

Note

The tool being invoked must be able to handle multiple files. The batch-mode inference rule must use `$<` as the macro to acc ess dependent files.

The batch-mode inference rules can speed up the build process. It is faster to supply files to the compiler in batch, because the compiler driver is invoked only once. For example, the C and C++ compiler performs better when handling a set of files because it can remain memory resident during the process.

The following example shows how to use batch-mode inference rules:

```
#
# sample makefile to illustrate batch-mode inference rules
#
O = .
S = .
Objs = $O/foo1.obj $O/foo2.obj $O/foo2.obj $O/foo3.obj $O/foo4.obj
CFLAGS = -nologo

all : $(Objs)

!ifdef NOBatch
{$S}.cpp{$O}.obj:
!else
{$S}.cpp{$O}.obj::
!endif
   $(CC) $(CFLAGS) -Fd$O\ -c $<

$(Objs) :

#end of makefile
```

NMAKE produces the following output without batch-mode inference rules:

```
E:\tmp> nmake -f test.mak -a NOBatch=1

Microsoft (R) Program Maintenance Utility   Version 7.00.0000
Copyright (C) Microsoft Corp 1988-2001. All rights reserved.
        cl -nologo -Fd.\ -c .\foo1.cpp
foo1.cpp
        cl -nologo -Fd.\ -c .\foo2.cpp
foo2.cpp
        cl -nologo -Fd.\ -c .\foo3.cpp
foo3.cpp
        cl -nologo -Fd.\ -c .\foo4.cpp
foo4.cpp
```

NMAKE produces the following result with the batch-mode inference rules:

```
E:\tmp> nmake -f test.mak -a

Microsoft (R) Program Maintenance Utility   Version 7.00.0000
Copyright (C) Microsoft Corp 1988-2001. All rights reserved.

        cl -nologo -Fd.\ -c .\foo1.cpp .\foo2.cpp .\foo3.cpp .\foo4.cpp
foo1.cpp
foo2.cpp
foo3.cpp
foo4.cpp
Generating Code...
```

See Also

Concepts

Inference Rules

Predefined Rules

Predefined inference rules use NMAKE-supplied command and options macros.

Rule	Command	Default action	Batch Rule	Platform nmake runs on
.asm.exe	$(AS) $(AFLAGS) $<	ml $<	no	x86
.asm.obj	$(AS) $(AFLAGS) /c $<	ml /c $<	yes	x86
.asm.exe	$(AS) $(AFLAGS) $<	ml64 $<	no	x64
.asm.obj	$(AS) $(AFLAGS) /c $<	ml64 /c $<	yes	x64
.s.obj	$(AS) $(AFLAGS) $<	ias /c $<	no	Itanium Processor Family (IPF)
.c.exe	$(CC) $(CFLAGS) $<	cl $<	no	all
.c.obj	$(CC) $(CFLAGS) /c $<	cl /c $<	yes	all
.cc.exe	$(CC) $(CFLAGS) $<	cl $<	no	all
.cc.obj	$(CC) $(CFLAGS) /c $<	cl /c $<	yes	all
.cpp.exe	$(CPP) $(CPPFLAGS) $<	cl $<	no	all
.cpp.obj	$(CPP) $(CPPFLAGS) /c $<	cl /c $<	yes	all
.cxx.exe	$(CXX) $(CXXFLAGS) $<	cl $<	no	all
.cxx.obj	$(CXX) $(CXXFLAGS) /c $<	cl /c $<	yes	all
.rc.res	$(RC) $(RFLAGS) /r $<	rc /r $<	no	all

See Also

Concepts

Inference Rules

Inferred Dependents and Rules

NMAKE assumes an inferred dependent for a target if an applicable inference rule exists. A rule applies if:

- *toext* matches the target's extension.
- *fromext* matches the extension of a file that has the target's base name and that exists in the current or specified directory.
- *fromext* is in .SUFFIXES; no other *fromext* in a matching rule has a higher **.SUFFIXES** priority.
- No explicit dependent has a higher **.SUFFIXES** priority.

Inferred dependents can cause unexpected side effects. If the target's description block contains commands, NMAKE executes those commands instead of the commands in the rule.

See Also

Concepts

Inference Rules

Precedence in Inference Rules

If an inference rule is multiply defined, NMAKE uses the highest-precedence definition. The following list shows the order of precedence from highest to lowest:

1. An inference rule defined in a makefile; later definitions have precedence.
2. An inference rule defined in Tools.ini; later definitions have precedence.
3. A predefined inference rule.

See Also

Concepts

Inference Rules

Dot Directives

Specify dot directives outside a description block, at the start of a line. Dot directives begin with a period (.) and are followed by a colon (:). Spaces and tabs are allowed. Dot directive names are case sensitive and are uppercase.

Directive	Purpose
.IGNORE :	Ignores nonzero exit codes returned by commands, from the place it is specified to the end of the makefile. By default, NMAKE halts if a command returns a nonzero exit code. To restore error checking, use **!CMDSWITCHES**. To ignore the exit code for a single command, use the dash (–) modifier. To ignore exit codes for an entire file, use /I.
.PRECIOUS : *targets*	Preserves *targets* on disk if the commands to update them are halted; has no effect if a command handles an interrupt by deleting the file. Separate the target names with one or more spaces or tabs. By default, NMAKE deletes a target if a build is interrupted by CTRL+C or CTRL+BREAK. Each use of **.PRECIOUS** applies to the entire makefile; multiple specifications are cumulative.
.SILENT :	Suppresses display of executed commands, from the place it is specified to the end of the makefile. By default, NMAKE displays the commands it invokes. To restore echoing, use **!CMDSWITCHES**. To suppress echoing of a single command, use the **@** modifier. To suppress echoing for an entire file, use /S.
.SUFFIXES : *list*	Lists extensions for inference-rule matching; predefined to include the following extensions: .exe .obj .asm .c .cpp .cxx .bas .cbl .for .pas .res .rc .f .f90

To change the **.SUFFIXES** list order or to specify a new list, clear the list and specify a new setting. To clear the list, specify no extensions after the colon:

```
.SUFFIXES :
```

To add additional suffixes to the end of the list, specify

```
.SUFFIXES : suffixlist
```

where *suffixlist* is a list of the additional suffixes, separated by one or more spaces or tabs. To see the current setting of **.SUFFIXES**, run NMAKE with /P.

See Also

Concepts

NMAKE Reference

Makefile Preprocessing

You can control the NMAKE session by using preprocessing directives and expressions. Preprocessing instructions can be placed in the makefile or in Tools.ini. Using directives, you can conditionally process your makefile, display error messages, include other makefiles, undefine a macro, and turn certain options on or off.

What do you want to know more about?

Makefile preprocessing directives

Expressions in makefile preprocessing

See Also

Concepts

NMAKE Reference

Makefile Preprocessing Directives

Preprocessing directives are not case sensitive. The initial exclamation point (!) must appear at the beginning of the line. Zero or more spaces or tabs can appear after the exclamation point, for indentation.

!CMDSWITCHES

{**+**| **–**}*option*... Turns each *option* listed on or off. Spaces or tabs must appear before the + or – operator; none can appear between the operator and the option letters. Letters are not case sensitive and are specified without a slash (/). To turn some options on and others off, use separate specifications of **!CMDSWITCHES**.

Only /D, /I, /N, and /S can be used in a makefile. In Tools.ini, all options are allowed except /F, /HELP, /NOLOGO, /X, and /?. Changes specified in a description block do not take effect until the next description block. This directive updates **MAKEFLAGS**; changes are inherited during recursion if **MAKEFLAGS** is specified.

!ERROR *text*

Displays *text* in error U1050, then halts NMAKE, even if /K, /I, **.IGNORE**, **!CMDSWITCHES**, or the dash (–) command modifier is used. Spaces or tabs before *text* are ignored.

!MESSAGE *text*

Displays *text* to standard output. Spaces or tabs before *text* are ignored.

!INCLUDE[**<**] *filename*[**>**]

Reads *filename* as a makefile, then continues with the current makefile. NMAKE searches for *filename* first in the specified or current directory, then recursively through directories of any parent makefiles, then, if *filename* is enclosed by angle brackets (< >), in directories specified by the **INCLUDE** macro, which is initially set to the INCLUDE environment variable. Useful to pass **.SUFFIXES** settings, **.PRECIOUS**, and inference rules to recursive makefiles.

!IF *constantexpression*

Processes statements between **!IF** and the next **!ELSE** or **!ENDIF** if *constantexpression* evaluates to a nonzero value.

!IFDEF *macroname*

Processes statements between **!IFDEF** and the next **!ELSE** or **!ENDIF** if *macroname* is defined. A null macro is considered to be defined.

!IFNDEF *macroname*

Processes statements between **!IFNDEF** and the next **!ELSE** or **!ENDIF** if *macroname* is not defined.

!ELSE[**IF** *constantexpression* | **IFDEF** *macroname*| **IFNDEF** *macroname*]

Processes statements between **!ELSE** and the next **!ENDIF** if the prior **!IF**, **!IFDEF**, or **!IFNDEF** statement evaluated to zero. The optional keywords give further control of preprocessing.

!ELSEIF

Synonym for **!ELSE IF**.

!ELSEIFDEF

Synonym for **!ELSE IFDEF**.

!ELSEIFNDEF

Synonym for **!ELSE IFNDEF**.

!ENDIF

Marks the end of an **!IF**, **!IFDEF**, or **!IFNDEF** block. Any text after **!ENDIF** on the same line is ignored.

!UNDEF *macroname*

Undefines *macroname*.

See Also

Concepts

Expressions in Makefile Preprocessing

The **!IF** or **!ELSE IF** *constantexpression* consists of integer constants (in decimal or C-language notation), string constants, or commands. Use parentheses to group expressions. Expressions use C-style signed long integer arithmetic; numbers are in 32-bit two's-complement form in the range – 2147483648 to 2147483647.

Expressions can use operators that act on constant values, exit codes from commands, strings, macros, and file-system paths.

What do you want to know more about?

Makefile preprocessing operators

Executing a program in preprocessing

See Also

Concepts

Makefile Preprocessing

Makefile Preprocessing Operators

The **DEFINED** operator is a logical operator that acts on a macro name. The expression **DEFINED** (*macroname*) is true if *macroname* is defined. **DEFINED** in combination with **!IF** or **!ELSE IF** is equivalent to **!IFDEF** or **!ELSE IFDEF**. However, unlike these directives, **DEFINED** can be used in complex expressions using binary logical operators.

The **EXIST** operator is a logical operator that acts on a file-system path. **EXIST** (*path*) is true if *path* exists. The result from **EXIST** can be used in binary expressions. If *path* contains spaces, enclose it in double quotation marks.

Integer constants can use the unary operators for numerical negation (–), one's complement (**~**), and logical negation (**!**).

Constant expressions can use the following binary operators.

Operator	Description	Operator	Description
+	Addition	**||**	Logical OR
–	Subtraction	**<<**	Left shift
*****	Multiplication	**>>**	Right shift
/	Division	**==**	Equality
%	Modulus	**!=**	Inequality
&	Bitwise AND	**<**	Less than
|	Bitwise OR	**>**	Greater than
^	Bitwise XOR	**<=**	Less than or equal to
&&	Logical AND	**>=**	Greater than or equal to

To compare two strings, use the equality (**==**) operator and the inequality (**!=**) operator. Enclose strings in double quotation marks.

See Also

Concepts

Expressions in Makefile Preprocessing

Executing a Program in Preprocessing

To use a command's exit code during preprocessing, specify the command, with any arguments, within brackets ([ ]). Any macros are expanded before the command is executed. NMAKE replaces the command specification with the command's exit code, which can be used in an expression to control preprocessing.

See Also

Concepts

Expressions in Makefile Preprocessing

VCBUILD Reference

You can use VCBUILD.exe to build Visual C++ projects and Visual Studio solutions from the command line. Invoking this utility is equivalent to running the **Build Project** or **Build Solution** command from the Visual Studio IDE.

In This Section

VCBUILD Command Line

Defines the command-line syntax for VCBUILD.

VCBUILD Options

Provides links to the VCBUILD options.

Related Sections

Building on the Command Line

Provides links to topics discussing building from the command line.

C/C++ Build Tools

Provides links to build tools that are included with Visual C++.

VCBUILD System Requirements

This topic outlines the system requirements for VCBUILD.EXE.

System Requirements	Client Configuration	Server Configuration
Processor	Minimum: • 400 megahertz (MHz) Pentium processor Recommended: • 800 MHz Pentium processor	Minimum: • 900 MHz Pentium processor Recommended: • 1 gigahertz (GHz) Pentium processor
Operating System	32-bit: • Windows Server 2003 • Windows XP Professional • Windows XP Home • Windows 2000 • Windows Vista	64-bit: • Windows Server 2003 • Windows XP • Windows Vista
Minimum Software	• .NET Framework Redistributable • IE 5.01; IE 6.0 SP1 recommended • MDAC 2.6 for data access applications; MDAC 2.7 SP1 recommended	• .NET Framework Redistributable • IE 5.01; IE 6.0 SP1 recommended • MDAC 2.6 for data access applications; MDAC 2.7 SP1 recommended
RAM	Minimum: • 96 megabytes (MB) Recommended: • 128 MB	Minimum: • 128 MB Recommended: • 256 MB
Hard Disk Space	800 MB	800 MB
CD or DVD Drive	Not required	Not required
Display	Minimum: • 800 x 600 256 colors Recommended: • 1024 x 768 High Color — 16-bit	Minimum: • 800 x 600 256 colors Recommended: • 1024 x 768 High Color — 16-bit
Mouse	Not required	Not required

See Also

Other Resources

VCBUILD Reference

Starting Builds as Part of Job Objects

The most common example of starting the build process as part of a job object is when you start either VCBUILD.EXE or DEVENV.EXE (Visual Studio) using the Windows **Run as** command, and then initiate a build.

Known Issues

When a build is started as part of a job object, the parent job can prevent the project system from creating additional processes as part of that job. Therefore, the project system will start build tools normally, apart from a job object.

- Any attempt to start a build as part of a job on Windows 2000 will always fail.
- Any attempt to cancel a build process that was started as part of a job could fail because the project system cannot cancel build tools after they are started.

See Also

Reference

job objects

Other Resources

VCBUILD Reference

VCBUILD Command Line

VCBUILD uses the following syntax at the command line.

```
VCBUILD [options] [project | solution] [config | $ALL]
```

Flags

options

For more information, see VCBUILD Options.

project

solution

config

$ALL

Remarks

To cancel the build process, press CTRL+C or CTRL+BREAK.

See Also

Other Resources

VCBUILD Reference

VCBUILD Options

VCBUILD has the following options:

Option	Description
/clean (VCBUILD)	Specifies outputting only clean build outputs.
/error	Specifies prefixing error lines that are outputted to stdin, stdout, stderr with the *string* argument.
/implib	Specifies creating an import library for a DLL configuration.
/info	Specifies prefixing information lines that are outputted to stdin, stdout, stderr with the *string* argument.
/link	Specifies performing a link without building sources.
/log	Specifies using *file* as the build log.
/logcommands	Specifies printing commands and response file to the screen.
/M	Specifies the number of concurrent builds to run.
/nocolor	Displays error and warning messages without colorization.
/noimplib	Specifies not generating an import library.
/pass0	Specifies performing pass 0 (MIDL) of Build.exe.
/pass1	Specifies performing pass 1 (compile) of Build.exe (implies /implib).
/pass2	Specifies to perform pass 2 (link) of Build.exe (implies /noimplib).
/platform	Only builds configurations for the given platform.
/rebuild (VCBUILD)	Specifies cleaning build outputs and then performs a build.
/time	Logs the time it takes to complete the build.
/upgrade	Upgrades the project file to the latest supported format.
/useenv	Uses the environment variables for PATH, INCLUDE, LIBS, and LIBPATH.
/warning	Prefixes warning lines with a user-defined string.

See Also

Other Resources

VCBUILD Reference

/clean (VCBUILD)

Specifies outputting only clean build outputs.

```
/clean
```

Remarks

/c is the short form of **/clean**.

See Also

Other Resources

VCBUILD Options

/error

Specifies prefixing error lines that are outputted to stdin, stdout, stderr with the *string* argument.

```
/error:string
```

Flags

string

A user-defined string.

See Also

Other Resources

VCBUILD Options

/implib

Specifies creating an import library for a DLL configuration.

```
/implib
```

See Also

Other Resources

VCBUILD Options

/info

Specifies prefixing information lines that are outputted to stdin, stdout, stderr with the *string* argument.

```
/info:string
```

Arguments

string

A user-defined string.

See Also

Other Resources

VCBUILD Options

/link

Specifies performing a link without building sources.

```
/link
```

Remarks

This option is similar to the /pass2 option.

/l is the short form of **/link**.

See Also

Other Resources

VCBUILD Options

/log

Specifies using *file* as the build log.

```
/log:fil
```

Arguments

file

Determines the filename of the build log.

See Also

Other Resources

VCBUILD Options

/logcommands

Specifies printing commands and response file to the screen.

```
/logcommands
```

See Also

Other Resources

VCBUILD Options

/M

Specifies the number of concurrent builds to run.

```
/Mnumber
```

Arguments

number

Specifies the maximum number of projects to build at the same time.

Remarks

For more information, see Multiprocessor Builds.

Example

The following command line specifies building `ConsoleApp1` (debug configuration) and `ConsoleApp2` (release configuration) simultaneously:

```
D:\>VCBUILD /M2 ConsoleApp1\ConsoleApp1.vcproj debug ConsoleApp2\ConsoleApp2.vcproj release
```

See Also

Other Resources

VCBUILD Options

/nocolor

Displays error and warning messages without colorization.

```
/nocolor
```

See Also

Other Resources

VCBUILD Options

/noimplib

Specifies not generating an import library.

```
/noimplib
```

Remarks

This option overrides **/implib** and is implied by **/pass2**.

See Also

Other Resources

VCBUILD Options

/pass0

Specifies performing pass 0 (MIDL) of Build.exe.

```
/pass0
```

Remarks

Use this switch to generate the pass 0 outputs (typically type libraries, header files, and *_i.c and *_p.c files).

See Also

Other Resources

VCBUILD Options

/pass1

Specifies performing pass 1 (compile) of Build.exe (implies /implib).

```
/pass1
```

Remarks

Use this switch to generate the pass 1 outputs (typically .lib files).

See Also

Other Resources

VCBUILD Options

/pass2

Specifies to perform pass 2 (link) of Build.exe (implies /noimplib).

```
/pass2
```

Remarks

Use this switch to generate the pass 2 outputs (the target files).

See Also

Other Resources

VCBUILD Options

/platform

Only builds configurations for the given platform.

```
/platform:string
```

Parameters

Parameter	Description
string	The name of the platform to target. These values can be found from the Configuration Manager Dialog Box.

See Also

Other Resources

VCBUILD Options

/rebuild (VCBUILD)

Specifies cleaning build outputs and then performs a build.

```
/rebuild
```

Example

The following command line specifies to rebuild two projects, `ConsoleApp1` (debug configuration) and `ConsoleApp2` (release configuration):

```
D:\>VCBUILD /r ConsoleApp1\ConsoleApp1.vcproj debug ConsoleApp2\ConsoleApp2.vcproj release
```

See Also

Other Resources

VCBUILD Options

/time

Logs the time it takes to complete the build.

```
/time
```

See Also

Other Resources

VCBUILD Options

/upgrade

Upgrades the project file to the latest supported format.

```
/upgrade
```

Remarks

See Also

Other Resources

VCBUILD Options

/useenv

Uses the environment variables for PATH, INCLUDE, LIBS, and LIBPATH.

```
/useenv
```

See Also

Other Resources

VCBUILD Options

/warning

Prefixes warning lines with a user-defined string.

```
/warning:string
```

Remarks

See Also

Other Resources

VCBUILD Options

Building C/C++ Isolated Applications and Side-by-side Assemblies

Visual C++ 2005 supports a new deployment model for Windows client applications based on the idea of isolated applications and side-by-side assemblies. In Visual C++ 2005, the ATL, MFC, Standard C++, and CRT libraries have been rebuilt as shared side-by-side assemblies that are installed into the native assembly cache (WinSxS folder). By default, Visual C++ 2005 builds all native C/C++ applications as isolated applications that use manifests to describe their dependencies on Visual C++ libraries.

Building C/C++ programs as isolated applications presents a range of advantages. For example, an isolated application is unaffected when other C/C++ applications install or uninstall Visual C++ libraries. Visual C++ libraries used by isolated applications may still be redistributed in either the application's local folder, or by installation to the native assembly cache (WinSxS); however, servicing of Visual C++ libraries for already deployed applications can be simplified by using a publisher configuration file. The isolated application deployment model makes it easier to ensure that C/C++ applications that are running on a specific computer use the most recent version of Visual C++ libraries, while still leaving open the possibility for system administrators and application authors to control explicit version binding of applications to their dependent DLLs.

This section discusses how you can build your C/C++ application as an isolated application and ensure that it binds to Visual C++ libraries using a manifest. The information in this section primarily applies to native, or unmanaged, Visual C++ applications. For information about deploying native applications built with Visual C++ 2005, see Redistributing Visual C++ Files.

In This Section

Concepts of Isolated Applications and Side-by-side Assemblies

Understanding Manifest Generation for C/C++ Programs

Troubleshooting C/C++ Isolated Applications and Side-by-side Assemblies

Related Sections

Isolated Applications and Side-by-side Assemblies

Deployment (C++)

Concepts of Isolated Applications and Side-by-side Assemblies

An application is considered an isolated application if all of its components are side-by-side assemblies. A side-by-side assembly is a collection of resources—a group of DLLs, windows classes, COM servers, type libraries, or interfaces—available for an application to use at runtime. Typically, a side-by-side assembly is one to several DLLs. For example, the C Runtime Library assembly contains three DLLs—msvcr80.dll, msvcm80.dll and msvcp80.dll—and is used by applications when they are built using functions from the CRT library.

Shared or private

A side-by-side assembly can be either shared or private. Shared side-by-side assemblies may be used by multiple applications that specify, in their manifests, a dependence on the assembly. Multiple versions of side-by-side assemblies can be shared by different applications running at the same time. A private assembly is an assembly that is deployed with an application and is available for the exclusive use of that application. Private assemblies are installed in the folder containing the application's executable file or one of its subfolders.

Manifests and search order

Both isolated applications and side-by-side assemblies are described by manifests. A manifest is an XML document that can be an external XML file or embedded inside an application or an assembly as a resource. The manifest file of an isolated application is used to manage the names and versions of shared side-by-side assemblies to which the application should bind at run time. The manifest of a side-by-side assembly specifies names, versions, resources, and dependent assemblies of side-by-side assemblies. For a shared side-by-side assembly, its manifest is installed in the WinSxS\Manifests folder. In the case of a private assembly, it is recommended to include its manifest in the DLL as a resource with ID equal to 1, and the name of the private assembly may be the same as the name of the DLL. For more information, see Private Assemblies.

At execution time, Windows uses assembly information from the application's manifest to search and load the corresponding side-by-side assembly. If an isolated application specifies an assembly dependency, the operating system first searches for the assembly among the shared assemblies in the WinSxS folder. If the required assembly is not found, the operating system then searches for a private assembly installed in a folder of the application's directory structure. For more information, see Assembly Searching Sequence.

Changing dependencies

It is possible to change side-by-side assembly dependencies after an application has been deployed by modifying the Publisher Configuration Files and Application Configuration Files. A publisher configuration file, also called a publisher policy file, is an XML file that globally redirects applications and assemblies from using one version of a side-by-side assembly to another version of the same assembly. A good example when changing dependency may be used is when a bug fix or security fix is deployed for a particular side-by-side assembly and a publisher of this assembly would like to redirect all applications to use the new version of the assembly. An application configuration file is an XML file that redirects a specific application from using one version of a side-by-side assembly to another version of the same assembly. It may be used to redirect a particular application to use a version of a side-by-side assembly different than the one defined in the global publisher configuration file. For more information, see Configuration.

Visual C++ libraries

In Visual C++ 2005, libraries such as ATL, MFC, CRT, Standard C++, OpenMP, and MSDIA are deployed as shared side-by-side assemblies to the native assembly cache. By default, all applications built with Visual C++ 2005 are built with the manifest embedded inside the final binary, which describes the dependencies of this binary on the Visual C++ libraries. To understand manifest generation for Visual C++ applications, see Understanding Manifest Generation for C/C++ Programs

See Also

Other Resources

Building C/C++ Isolated Applications and Side-by-side Assemblies

Building C/C++ Isolated Applications

An isolated application depends only on side-by-side assemblies and binds to its dependencies using a manifest. It is not required for your application to be fully isolated in order to run properly on Windows; however, by investing in making your application fully isolated, you may save time if you need to service your application in the future. For more information on the advantages of making your application fully isolated, see Isolated Applications.

When you build your native C/C++ application using Visual C++ 2005, by default the Visual Studio project system generates a manifest file that describes your application's dependencies on Visual C++ libraries. If these are the only dependencies your application has, then it becomes an isolated application as soon as it is rebuilt with Visual Studio 2005. If your application is using other libraries at runtime, then you may need to rebuild those libraries as side-by-side assemblies following the steps described in Building C/C++ Side-by-side Assemblies.

See Also

Concepts

Concepts of Isolated Applications and Side-by-side Assemblies

Other Resources

Building C/C++ Isolated Applications and Side-by-side Assemblies

Building C/C++ Side-by-side Assemblies

A side-by-side assembly is a collection of resources—a group of DLLs, windows classes, COM servers, type libraries, or interfaces—available for an application to use at runtime. In Visual C++ 2005, the ATL, MFC, Standard C++, and CRT libraries have been repackaged in several side-by-side assemblies that can be redistributed with Visual C++ applications either as shared or private assemblies. See Visual C++ Libraries as Shared Side-by-Side Assemblies for more information. The primary advantage of repackaging DLLs in assemblies is that multiple versions of assemblies can be used by applications at the same time and it is possible to service currently installed assemblies in case of an update release.

A Visual C++ application may use one or several DLLs in different parts of the application. At runtime, the DLLs are loaded into the main process and the required code is executed. The application relies on the operating system to locate the requested DLLs, understand what other dependent DLLs have to be loaded and then load them together with the requested DLL. On Windows operating systems versions earlier than Windows XP, Windows Server 2003, and Windows Vista, the operating system loader searches for dependent DLLs in either the application's local folder or another folder specified on the system path. On Windows XP, Windows Server 2003, and Windows Vista, the operating system loader can also search for dependent DLLs using a manifest file and search for side-by-side assemblies that contain these DLLs.

By default, when a DLL is built with Visual Studio 2005 it has an application manifest embedded as an RT_MANIFEST resource with ID equal to 2. Just as for an executable, this manifest describes dependencies of this DLL on other assemblies. This assumes that the DLL is not part of a side-by-side assembly and applications that depend on this DLL are not going to use an application manifest to load it, but instead rely on the operating system loader to find this DLL on the system path.

Note

It is important for a DLL that uses an application manifest to have the manifest embedded as a resource with ID equal to 2. If the DLL is dynamically loaded at runtime (for example, using the LoadLibrary function), the operating system loader loads dependent assemblies specified in the DLL's manifest. An external application manifest for DLLs is not checked during a **LoadLibrary** call. If the manifest is not embedded, the loader may attempt to load incorrect versions of assemblies or fail to find dependent assemblies.

One or several related DLLs can be repackaged into a side-by-side assembly with a corresponding assembly manifest, which describes which files form the assembly as well as the dependence of the assembly on other side-by-side assemblies.

Note

If an assembly contains one DLL, it is recommended to embed the assembly manifest into this DLL as a resource with ID equal to 1, and give the private assembly the same name as the DLL. For example, if the name of the DLL is mylibrary.dll, the value of the name attribute used in the <assemblyIdentity> element of the manifest may also be mylibrary. In some cases, when a library has an extension other than .dll (for example, an MFC ActiveX Controls project creates an .ocx library) an external assembly manifest can be created. In this case, the name of the assembly and its manifest must be different than the name of the DLL (for example, MyAssembly, MyAssembly.manifest, and mylibrary.ocx). However it is still recommended to rename such libraries to have the extension.dll and embed the manifest as a resource to reduce the future maintenance cost of this assembly. For more information about how the operating system searches for private assemblies, see Assembly Searching Sequence.

This change may allow deployment of corresponding DLLs as a private assembly in an application local folder or as a shared assembly in the WinSxS assembly cache. Several steps have to be followed in order to achieve correct runtime behavior of this new assembly; they are described in Guidelines for Creating Side-by-side Assemblies. After an assembly is correctly authored it can deployed as either a shared or private assembly together with an application that depends on it. When installing side-by-side assemblies as a shared assembly, you may either follow the guidelines outlined in Installing Win32 Assemblies for Side-by-Side Sharing on Windows XP or use merge modules. When installing side-by-side assemblies as a private assembly, you may just copy the corresponding DLL, resources and assembly manifest as part of the installation process to the application local folder on the target computer, ensuring that this assembly can be found by the loader at runtime (see Assembly Searching Sequence). Another way is to use Windows Installer and follow the guidelines outlined in Installing Win32 Assemblies for the Private Use of an Application on Windows XP.

Related Sections

Deploying Windows-Client Applications

See Also

Concepts

Deployment Examples

Building C/C++ Isolated Applications

Other Resources

Building C/C++ Isolated Applications and Side-by-side Assemblies

Understanding Manifest Generation for C/C++ Programs

A manifest is an XML document that can be an external XML file or a resource embedded inside an application or an assembly. The manifest of an isolated application is used to manage the names and versions of shared side-by-side assemblies to which the application should bind at run time. The manifest of a side-by-side assembly specifies its dependencies on names, versions, resources, and other assemblies.

There are two ways to create a manifest for an isolated application or a side-by-side assembly. First, the author of the assembly can manually create a manifest file following rules and naming requirements. Alternatively, if a program only depends on Visual C++ assemblies such as CRT, MFC, ATL or others, then a manifest can be generated automatically by the linker.

The headers of Visual C++ libraries contain assembly information, and when the libraries are included in application code, this assembly information is used by the linker to form a manifest for the final binary. The linker does not embed the manifest file inside the binary, and can only generate the manifest as an external file. Having a manifest as an external file may not work for all scenarios. For example, it is recommended that private assemblies have embedded manifests. In command line builds such as those that use nmake to build code, a manifest can be embedded using the manifest tool; for more information see Manifest Generation at the Command Line. When building in Visual Studio, a manifest can be embedded by setting a property for the manifest tool in the **Project Properties** dialog; see Manifest Generation in Visual Studio.

See Also

Concepts

Concepts of Isolated Applications and Side-by-side Assemblies

Other Resources

Building C/C++ Isolated Applications and Side-by-side Assemblies

Manifest Generation in Visual Studio

Generation of a manifest file for a particular project can be controlled in the project **Property Pages** dialog. On the **Configuration Properties** tab, click **Linker**, then **Manifest File**, then **Generate Manifest**. By default the project properties of new projects are set to generate a manifest file. However it is possible to disable generation of the manifest for a project using the **Generate Manifest** property of the project. When this property is set to **Yes**, the manifest for this project is generated. Otherwise the linker ignores assembly information when resolving dependencies of the application code, and does not generate the manifest.

The build system in Visual Studio allows the manifest to be embedded in the final binary application file, or generated as an external file. This behavior is controlled by the **Embed Manifest** option in the **Project Properties** dialog. To set this property, open the **Manifest Tool** node, then select **Input and Output**. If the manifest is not embedded, it is generated as an external file and saved in the same directory as the final binary. If the manifest is embedded, Visual Studio embeds the final manifests using the following process:

1. After the source code is compiled to object files, the linker collects dependent assembly information. While linking the final binary, the linker generates an intermediate manifest that is used later to generate the final manifest.
2. After the intermediate manifest and linking are finished, the manifest tool will be executed to merge a final manifest and save it as an external file.
3. The project build system then detects whether the manifest generated by the manifest tool contains different information than the manifest already embedded in the binary.
4. If the manifest embedded in the binary is different from the manifest generated by the manifest tool, or the binary does not contain an embedded manifest, Visual Studio will invoke the linker one more time to embed the external manifest file inside the binary as a resource.
5. If the manifest embedded in the binary is the same as the manifest generated by the manifest tool, the build will continue to the next build steps.

The manifest is embedded inside the final binary as a text resource and it can be viewed by opening the final binary as a file in Visual Studio. To ensure that the manifest points to the correct libraries, follow the steps described in Understanding Dependencies of a Visual C++ Application or follow the suggestions described in the Troubleshooting section.

See Also

Reference

Private Assemblies

Manifest Tool

Concepts

How to: Embed a Manifest Inside a C/C++ Application

Understanding Manifest Generation for C/C++ Programs

Manifest Generation at the Command Line

When building C/C++ applications from the command line using nmake or similar tools, the manifest is generated after the linker has processed all object files and built the final binary. The linker collects assembly information stored in the object files and combines this information into a final manifest file. By default the linker will generate a file named <binary_name>.<extension>.manifest to describe the final binary. The linker does not embed a manifest file inside the binary and can only generate a manifest as an external file. There are several ways to embed a manifest inside the final binary, such as using the Manifest Tool (mt.exe) or compiling the manifest into a resource file. It is important to keep in mind that specific rules have to be followed when embedding a manifest inside the final binary to enable such features as incremental linking, signing, and edit and continue. These and other options are discussed in How to: Embed a Manifest Inside a C/C++ Application when building on the command line.

See Also

Reference

Manifests

/INCREMENTAL (Link Incrementally)

Concepts

Strong Name Assemblies (Assembly Signing)

Edit and Continue

Understanding Manifest Generation for C/C++ Programs

How to: Embed a Manifest Inside a C/C++ Application

It is recommended that a C/C++ application (or library) have its manifest embedded inside the final binary because this ensures proper runtime behavior in most scenarios. Visual Studio by default tries to embed the manifest when building a project from source files; see Manifest Generation in Visual Studio for more details. However if an application is built using nmake, some changes to the existing makefile are necessary. This section demonstrates how to change existing makefiles to automatically embed the manifest inside the final binary.

Two approaches

There are two ways to embed the manifest inside an application or library.

- If you are not doing an incremental build you can directly embed the manifest using a command line similar to the following as a post-build step:

  mt.exe –manifest MyApp.exe.manifest -outputresource:MyApp.exe;1

  or

  mt.exe –manifest MyLibrary.dll.manifest -outputresource:MyLibrary.dll;2

  (1 for an EXE, 2 for a DLL.)

- If you are doing an increment build, directly editing the resource as shown above will disable incremental building and cause a full rebuild; therefore a different approach should be taken:
  - Link the binary to generate the MyApp.exe.manifest file.
  - Convert the manifest to a resource file.
  - Re-link (incrementally) to embed the manifest resource into the binary.

The following examples show how to change makefiles to incorporate both techniques.

Makefiles (Before)

First let's take a look at the nmake script for MyApp.exe, a simple application built from one file:

```
# build MyApp.exe
!if "$(DEBUG)" == "1"
CPPFLAGS=$(CPPFLAGS) /MDd
LFLAGS=$(LFLAGS) /INCREMENTAL
!else
CPPFLAGS=$(CPPFLAGS) /MD
!endif

MyApp.exe : MyApp.obj
    link $** /out:$@ $(LFLAGS)

MyApp.obj : MyApp.cpp

clean :
    del MyApp.obj MyApp.exe
```

The nmake script for MyLibrary.dll looks very similar:

```
# build MyLibrary.dll
!if "$(DEBUG)" == "1"
CPPFLAGS=$(CPPFLAGS) /MDd
LFLAGS=$(LFLAGS) /DLL /INCREMENTAL

!else
CPPFLAGS=$(CPPFLAGS) /MD
LFLAGS=$(LFLAGS) /DLL

!endif
```

```
MyLibrary.dll : MyLibrary.obj
    link $** /out:$@ $(LFLAGS)

MyLibrary.obj : MyLibrary.cpp

clean :
    del MyLibrary.obj MyLibrary.dll
```

If this script is run unchanged with Visual C++ 2005, it successfully creates MyApp.exe. It also creates the external manifest file MyApp.exe.manifest, for use by the operating system to load dependent assemblies at runtime.

Makefiles (After)

To build with embedded manifests we have to make four small changes to each of the original makefiles. For the MyApp.exe makefile:

```
# build MyApp.exe
!include makefile.inc
#^^^^^^^^^^^^^^^^^^^^ Change #1. (Add full path if necessary.)

!if "$(DEBUG)" == "1"
CPPFLAGS=$(CPPFLAGS) /MDd
LFLAGS=$(LFLAGS) /INCREMENTAL
!else
CPPFLAGS=$(CPPFLAGS) /MD
!endif

MyApp.exe : MyApp.obj
    link $** /out:$@ $(LFLAGS)
    $(_VC_MANIFEST_EMBED_EXE)
#^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Change #2

MyApp.obj : MyApp.cpp

clean :
    del MyApp.obj MyApp.exe
    $(_VC_MANIFEST_CLEAN)
#^^^^^^^^^^^^^^^^^^^^^^^^ Change #3

!include makefile.targ.inc
#^^^^^^^^^^^^^^^^^^^^^^^^^ Change #4. (Add full path if necessary.)
```

For the MyLibrary.dll makefile:

```
# build MyLibrary.dll
!include makefile.inc
#^^^^^^^^^^^^^^^^^^^^ Change #1. (Add full path if necessary.)

!if "$(DEBUG)" == "1"
CPPFLAGS=$(CPPFLAGS) /MDd
LFLAGS=$(LFLAGS) /DLL /INCREMENTAL

!else
CPPFLAGS=$(CPPFLAGS) /MD
LFLAGS=$(LFLAGS) /DLL

!endif

MyLibrary.dll : MyLibrary.obj
    link $** /out:$@ $(LFLAGS)
    $(_VC_MANIFEST_EMBED_DLL)
#^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Change #2.

MyLibrary.obj : MyLibrary.cpp
```

```
clean :
    del MyLibrary.obj MyLibrary.dll
    $(_VC_MANIFEST_CLEAN)
#^^^^^^^^^^^^^^^^^^^^^^^^ Change #3.

!include makefile.targ.inc
#^^^^^^^^^^^^^^^^^^^^^^^^^ Change #4. (Add full path if necessary.)
```

The real work is done by the two files that the makefiles now include, makefile.inc and makefile.targ.inc.

Create makefile.inc and copy the following into it:

```
# makefile.inc -- Include this file into existing makefile at the very top.

# _VC_MANIFEST_INC specifies whether build is incremental (1 - incremental).
# _VC_MANIFEST_BASENAME specifies name of a temporary resource file.

!if "$(DEBUG)" == "1"
CPPFLAGS=$(CPPFLAGS) /MDd
LFLAGS=$(LFLAGS) /INCREMENTAL
_VC_MANIFEST_INC=1
_VC_MANIFEST_BASENAME=__VC80.Debug

!else
CPPFLAGS=$(CPPFLAGS) /MD
_VC_MANIFEST_INC=0
_VC_MANIFEST_BASENAME=__VC80

!endif

#####################################################
# Specifying name of temporary resource file used only in incremental builds:

!if "$(_VC_MANIFEST_INC)" == "1"
_VC_MANIFEST_AUTO_RES=$(_VC_MANIFEST_BASENAME).auto.res
!else
_VC_MANIFEST_AUTO_RES=
!endif

#####################################################
# _VC_MANIFEST_EMBED_EXE - command to embed manifest in EXE:

!if "$(_VC_MANIFEST_INC)" == "1"

#MT_SPECIAL_RETURN=1090650113
#MT_SPECIAL_SWITCH=-notify_resource_update
MT_SPECIAL_RETURN=0
MT_SPECIAL_SWITCH=
_VC_MANIFEST_EMBED_EXE= \
if exist $@.manifest mt.exe -manifest $@.manifest -out:$(_VC_MANIFEST_BASENAME).auto.manife
st $(MT_SPECIAL_SWITCH) & \
if "%ERRORLEVEL%" == "$(MT_SPECIAL_RETURN)" \
rc /r $(_VC_MANIFEST_BASENAME).auto.rc & \
link $** /out:$@ $(LFLAGS)

!else

_VC_MANIFEST_EMBED_EXE= \
if exist $@.manifest mt.exe -manifest $@.manifest -outputresource:$@;1

!endif

#####################################################
# _VC_MANIFEST_CLEAN - command to clean resources files generated temporarily:

!if "$(_VC_MANIFEST_INC)" == "1"

_VC_MANIFEST_CLEAN=-del $(_VC_MANIFEST_BASENAME).auto.res \
```

```
    $(_VC_MANIFEST_BASENAME).auto.rc \
    $(_VC_MANIFEST_BASENAME).auto.manifest

!else

_VC_MANIFEST_CLEAN=

!endif

# End of makefile.inc
#####################################################
```

Now create makefile.targ.inc and copy the following into it:

```
# makefile.targ.inc - include this at the very bottom of the existing makefile

#####################################################
# Commands to generate initial empty manifest file and the RC file
# that references it, and for generating the .res file:

$(_VC_MANIFEST_BASENAME).auto.res : $(_VC_MANIFEST_BASENAME).auto.rc

$(_VC_MANIFEST_BASENAME).auto.rc : $(_VC_MANIFEST_BASENAME).auto.manifest
    type <<$@
#include <winuser.h>
1RT_MANIFEST"$(_VC_MANIFEST_BASENAME).auto.manifest"
<< KEEP

$(_VC_MANIFEST_BASENAME).auto.manifest :
    type <<$@
<?xml version='1.0' encoding='UTF-8' standalone='yes'?>
<assembly xmlns='urn:schemas-microsoft-com:asm.v1' manifestVersion='1.0'>
</assembly>
<< KEEP

# end of makefile.targ.inc
```

See Also

Concepts

Understanding Manifest Generation for C/C++ Programs

Troubleshooting C/C++ Isolated Applications and Side-by-side Assemblies

Loading a C/C++ application can fail if dependent Visual C++ libraries cannot be found. A list of potential runtime errors can be found in Redistributing Visual C++ Files. In this section the most common reasons for a C/C++ application failing to load are described, with proposed steps to resolve the problems.

If an application is deployed on a computer without Visual C++ installed, and it crashes with error messages similar to those listed in Redistributing Visual C++ Files, several things have to be checked to understand the reason for the error.

1. Follow the steps described in Understanding Dependencies of a Visual C++ Application. The dependency walker can show most dependencies for any particular application or DLL. If you see that some DLLs are missing, install those DLLs on the computer on which you are trying to run your application.

2. A manifest is used by the operating system loader to load assemblies that your application depends on. It can either be embedded inside the binary as a resource, or saved as an external file in the application's local folder. To check whether the manifest is embedded inside the binary, open your binary in Visual Studio and browse through the resources of this binary. You should be able to find a resource named RT_MANIFEST. If you cannot find a manifest embedded inside the binary, check for an external file named something like <binary_name>.<extension>.manifest.

3. If a manifest is not present, you need to ensure that the linker generates a manifest for your project. You need to check the linker option **Generate manifest** in the **Project Properties** dialog for this project.

Note
Building Visual C++ projects without manifest generation is not supported. Any C/C++ program built in Visual C++ 2005 has to include a manifest describing its dependencies on Visual C++ libraries.

4. If the manifest is embedded inside the binary, ensure that the ID of RT_MANIFEST is correct for this type of the binary. For applications, ID should be equal to 1, for most DLLs ID should be equal to 2. If the manifest is in an external file, open it in an XML editor or text editor. For more information on manifests and rules for deployment, see Manifests.

Note
On Windows XP, if an external manifest is present in the application's local folder, the operating system loader uses this manifest instead of a manifest embedded inside the binary. On Windows Server 2003 and later versions of Windows, the opposite is true—the external manifest is ignored and the embedded manifest is used when present.

5. It is recommended that all DLLs have a manifest embedded inside the binary. External manifests are ignored when a DLL is loaded though a **LoadLibrary** call. For more information, see Assembly manifests.

6. Check that all assemblies enumerated in the manifest are correctly installed on the computer. Each assembly is specified in the manifest by its name, version number and processor architecture. If your application depends on side-by-side assemblies, check that these assemblies are installed on the computer properly, so they can be found by the operating system loader using the steps specified in Assembly Searching Sequence. Remember that 64-bit assemblies cannot be loaded in 32-bit processes and cannot be executed on 32-bit operating systems.

Example

Assume we have an application, appl.exe, built with Visual C++ 2005. This application has its manifest either embedded inside appl.exe as a binary resource RT_MANIFEST with ID equal to 1, or stored as an external file, appl.exe.manifest. The content of the manifest may be something like this:

```
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <dependency>
    <dependentAssembly>
      <assemblyIdentity type="win32" name="Microsoft.VC80.CRT" version="8.0.50608.0" processorA
rchitecture="x86" publicKeyToken="1fc8b3b9a1e18e3b"></assemblyIdentity>
    </dependentAssembly>
  </dependency>
</assembly>
```

To the operating system loader this manifest says that appl.exe depends on an assembly named Microsoft.VC80.CRT, version 8.0.50608.0 and built for 32-bit x86 processor architecture.

The dependent side-by-side assembly can be installed as either a shared assembly or as a private assembly. For example, Visual Studio 2005 installs the CRT assembly as a shared side-by-side assembly which can be found in the directory %WINDIR%\WinSxS\x86_Microsoft.VC80.CRT_1fc8b3b9a1e18e3b_8.0.50608.0_x-ww_b7acac55 when running Windows XP or in the directory %WINDIR%\winsxs\x86_microsoft.vc80.crt_1fc8b3b9a1e18e3b_8.0.50727.42_none_db5f52fb98cb24ad when running Windows Vista.

The **assembly manifest** for a shared Visual C++ CRT assembly is also installed in %WINDIR%\WinSxS\Manifests\x86_microsoft.vc80.crt_1fc8b3b9a1e18e3b_8.0.50608.0_x-ww_b7acac55.manifest when running Windows XP or in %WINDIR%\winsxs\Manifests\x86_microsoft.vc80.crt_1fc8b3b9a1e18e3b_8.0.50727.42_none_db5f52fb98cb24ad.manifest when running Windows Vista and it identifies this assembly and lists its contents (DLLs that are part of this assembly):

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- Copyright ┬⌐ 1981-2001 Microsoft Corporation -->
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
   <noInheritable/>
   <assemblyIdentity type="win32" name="Microsoft.VC80.CRT" version="8.0.50727.42" processorArc
hitecture="x86" publicKeyToken="1fc8b3b9a1e18e3b"/>
   <file name="msvcr80.dll" hash="3ca5156e8212449db6c622c3d10f37d9adb12c66" hashalg="SHA1"/>
   <file name="msvcp80.dll" hash="92cf8a9bb066aea821d324ca4695c69e55b27cff" hashalg="SHA1"/>
   <file name="msvcm80.dll" hash="7daa93e1195940502491c987ff372190bf199395" hashalg="SHA1"/>
</assembly>
```

Side-by-side assemblies can also use publisher configuration files, also called policy files, to globally redirect applications and assemblies from using one version of a side-by-side assembly to another version of the same assembly. You can check policies for the shared Visual C++ CRT assembly in %WINDIR%\WinSxS\Policies\x86_policy.8.0.Microsoft.VC80.CRT_1fc8b3b9a1e18e3b_x-ww_77c24773\8.0.50727.42.policy when running Windows XP or in %WINDIR%\winsxs\Manifests\x86_policy.8.0.microsoft.vc80.crt_1fc8b3b9a1e18e3b_8.0.50727.42_none_5c4003bc63e949f6.manifest when running Windows Vista, which contains something like this:

```
</assembly>
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- Copyright ┬⌐ 1981-2001 Microsoft Corporation -->
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">

   <assemblyIdentity type="win32-policy" name="policy.8.0.Microsoft.VC80.CRT" version="8.0.5072
7.42" processorArchitecture="x86" publicKeyToken="1fc8b3b9a1e18e3b"/>
   <dependency>
      <dependentAssembly>
         <assemblyIdentity type="win32" name="Microsoft.VC80.CRT" processorArchitecture="x86" p
ublicKeyToken="1fc8b3b9a1e18e3b"/>
         <bindingRedirect oldVersion="8.0.41204.256-8.0.50608.0" newVersion="8.0.50727.42"/>
      </dependentAssembly>
   </dependency>
</assembly>
```

The policy above specifies that any application or assembly that asks for version 8.0.41204.256 of this assembly should instead use version 8.0.50727.42 of this assembly, which is the current version installed on the system. If a version of the assembly mentioned in the application's manifest is specified in the policy file, the loader looks for a version of this assembly specified in the manifest in the WinSxS folder, and if this version is not installed, load fails. And if an assembly version 8.0.50727.42 is not installed as well, load fails for applications that ask for assembly version 8.0.41204.256.

However, the CRT assembly can also be installed as a private side-by-side assembly in the application's local folder. If the operating system fails to find the CRT or any other assembly as a shared assembly, it starts looking for the assembly as a private assembly. It searches for private assemblies in the following order:

1. Check the application local folder for a manifest file with name <assemblyName>.manifest. In this example, the loader tries to find Microsoft.VC80.CRT.manifest in the same folder as appl.exe. If the manifest is found, the loader loads the CRT DLL from the application folder. If the CRT DLL is not found, load fails.

2. Try to open folder <assemblyName> in appl.exe's local folder and if it exists, load manifest file <assemblyName>.manifest from this folder. If the manifest is found, the loader loads the CRT DLL from <assemblyName> folder. If the CRT DLL is not found, load fails.

See Assembly Searching Sequence for a more detailed description of how the loader searches for dependent assemblies. If the loader fails to find a dependent assembly as a private assembly, load fails and "The system cannot execute the specified program" is displayed. To resolve this error, dependent assemblies and DLLs that are part of them must be installed on the computer as either private or shared assemblies.

See Also

Concepts

Concepts of Isolated Applications and Side-by-side Assemblies

Other Resources

Building C/C++ Isolated Applications and Side-by-side Assemblies

C/C++ Building Reference

Visual C++ provides two ways of building a C/C++ program. The easiest (and most common) way is to build within the Visual C++ development environment. The other way is to build from a command prompt using command-line tools. In either case, you can create your source files using the Visual C++ source editor or a third-party editor of your choice.

If your program uses a makefile rather than a .vcproj file, you can still build it in the development environment as an external project.

In This Section

Frequently Asked Questions on Building

Provides links to frequently asked questions when building.

Compiling a C/C++ Program

Describes the compiler, which creates an object file containing machine code, linker directives, sections, external references, and function/data names.

Linking

Describes the linker, which combines code from the object files created by the compiler and from statically linked libraries, resolves the name references, and creates an executable file.

Release Builds

Presents information on why and when you would want to change from a debug build to a release build and also discusses some of the problems you may encounter when changing from a debug to a release build.

Optimizing Your Code

Provides links to topics discussing the mechanisms for optimizing code:

C/C++ Build Tools

Provides the following command-line tools for viewing or manipulating build output:

C/C++ Build Errors

Introduces the build errors section in the table of contents.

Related Sections

C/C++ Preprocessor Reference

Discusses the preprocessor, which prepares source files for the compiler by translating macros, operators, and directives.

Understanding Custom Build Steps and Build Events

Discusses customizing the build process.

Building a C/C++ Program

Provides links to topics describing building your program from the command line or from the integrated development environment of Visual Studio.

Setting Compiler Options

Describes setting compiler options in the development environment or on the command line.

Compiler Options

Provides links to topics discussing using compiler options.

Setting Linker Options

Describes setting linker options inside or outside the integrated development environment.

Linker Options

Provides links to topics discussing using linker options.

BSCMAKE Reference

Describes the Microsoft Browse Information Maintenance Utility (BSCMAKE.EXE), which builds a browse information file (.bsc) from .sbr files created during compilation.

LIB Reference

Describes the Microsoft Library Manager (LIB.exe), which creates and manages a library of Common Object File Format (COFF) object files.

EDITBIN Reference

Describes the Microsoft COFF Binary File Editor (EDITBIN.EXE), which modifies Common Object File Format (COFF) binary files.

DUMPBIN Reference

Describes the Microsoft COFF Binary File Dumper (DUMPBIN.EXE), which displays information about Common Object File Format (COFF) binary files.

NMAKE Reference

Describes the Microsoft Program Maintenance Utility (NMAKE.EXE), which is a tool that builds projects based on commands contained in a description file.

Frequently Asked Questions on Building

- How can I build faster?
- How do I compile and link C code, not C++?

See Also

Other Resources

C/C++ Building Reference

How Can I Build Faster?

This topic contains tips for reducing the amount of time it takes to build a project.

- Reduce the number of header files Visual C++ must include and check for every build.
- Use precompiled headers. See /Y (Precompiled Headers) for more information.
- When calling cl.exe directly, do not invoke the compiler once for every source file; pass all source files to cl.exe in the same invocation to create .obj files.
- Use minimal rebuild; see /Gm for more information.
- Use incremental linking; see /INCREMENTAL for more information.
- Use #pragma once, when appropriate. For example, if your headers use include guards.
- Dynamically link to the CRT; see /MD for more information.

Additional topics include:

- Faster Builds and Smaller Header Files
- Excluding Files When Dependency Checking

See Also

Reference

Frequently Asked Questions on Building

Creating Precompiled Header Files

Faster Builds and Smaller Header Files

To speed the build process, Visual C++ provides the following definitions that reduce the size of the Win32 header files by excluding some of the less common APIs:

- VC_EXTRALEAN
- WIN32_LEAN_AND_MEAN

VC_EXTRALEAN defines WIN32_LEAN_AND_MEAN and a number of NO*service* definitions, such as NOCOMM and NOSOUND. (For a list of NO*service* definitions, see the header file Windows.h and the MFC header file afxv_w32.h.)

Applications created with the Visual C++ application wizards use VC_EXTRALEAN automatically. You can manually define VC_EXTRALEAN in legacy MFC applications to speed their build process.

Non-MFC applications can define WIN32_LEAN_AND_MEAN and applicable NO*service* definitions to reduce build times.

Trying to use an API excluded by these definitions causes compiler errors. If a program that defines NOCOMM or VC_EXTRALEAN tries to use **PurgeComm**, for example, the following errors result:

```
error C2065: 'PurgeComm' : undeclared identifier
error C2064: term does not evaluate to a function
```

Fore more information, see http://support.microsoft.com/default.aspx?scid=kb;en-us;166474.

See Also
Reference
How Can I Build Faster?

Excluding Files When Dependency Checking

Adding an include file to one of your source code files creates a dependency. The source file *depends* on the include file. During the build process, the project system checks to see if an include file has changed and if the source file needs to be recompiled. Some include files never change, and you can reduce build time by eliminating dependency checking for those files.

A file called SYSINCL.dat file, which you create, lists the directories that are excluded from dependency checking. Any file in a specified directory or one of its subdirectories will be excluded. Visual C++ reads SYSINCL.dat when you start the product.

You can put a SYSINCL.dat file in *Local Settings*\Application Data\Microsoft\VisualStudio*version* (for example, in C:\Documents and Settings*username*).

In addition to being able to specify directories in which no file will ever be checked for dependencies, Visual C++ will never check dependencies for any file residing in or under the *version* subdirectory.

To exclude files from dependency checking

1. Add directory names (using full, absolute paths) one per line, to SYSINCL.dat.
2. Exit the development environment.
3. Restart the development environment.
4. Click **Rebuild** for changes to take effect.

Note
Excluding dependencies does not increase the speed of the **Rebuild** command because no dependency checking occurs with **Rebuild**.

See Also

Reference

How Can I Build Faster?

Multiprocessor Builds

Multiprocessor builds enable you to build multiple projects concurrently on computers that have more than one CPU. For example, if your computer has two CPUs, then this feature allows you to build up to two projects at the same time.

To optimize the build process, Visual Studio automatically sets the maximum number of concurrent builds equal to the number of CPUs on your computer. You can modify this setting by changing the value of the **Maximum number of parallel project builds** property in the Build and Run property page of the **Options** dialog box. For more information, see How to: Set the Number of Concurrent Builds for Multiprocessor Builds.

Running Multiprocessor Builds

You can run multiprocessor builds from the IDE or command line. To build from the command line, use either VCBUILD or DEVENV. For tasks that demonstrate these concepts, see:

- How to: Run Multiprocessor Builds in the IDE
- How to: Run Multiprocessor Builds with VCBUILD
- How to: Run Multiprocessor Builds with DEVENV Command Line Switches

Multiprocessor Build Output

When you run multiprocessor builds, the project system assigns a number to each project configuration included in the build, which is followed by a right angle-bracket. This prompt precedes each status message for every build step.

The following example build output is the result of a multiprocessor build:

```
1>------ Build started: Project: ConsoleApp2, Configuration: Debug Win32 ------
2>------ Build started: Project: ConsoleApp1, Configuration: Debug Win32 ------
1>Compiling...
2>Compiling...
1>stdafx.cpp
2>stdafx.cpp
2>Compiling...
1>Compiling...
2>AssemblyInfo.cpp
1>AssemblyInfo.cpp
1>ConsoleApp2.cpp
2>ConsoleApp1.cpp
2>Generating Code...
1>Generating Code...
1>Compiling resources...
2>Compiling resources...
2>Linking...
1>Linking...
2>Build log was saved at "file://d:\Documents and Settings\username\My Documents\Visual Stu
dio\Projects\Demo\ConsoleApp1\Debug\BuildLog.htm"
2>ConsoleApp1 - 0 error(s), 0 warning(s)
2>
1>Build log was saved at "file://d:\Documents and Settings\username\My Documents\Visual Stu
dio\Projects\Demo\ConsoleApp2\Debug\BuildLog.htm"
1>ConsoleApp2 - 0 error(s), 0 warning(s)
1>
---------------------- Done ----------------------

    Build: 2 succeeded, 0 failed, 0 up-to-date, 0 skipped
```

The multiprocessor build output illustrates that build steps for multiple projects can spawn concurrently. This behavior is in contrast to non-multiprocessor builds, which do not start build processes for other projects until the build process for the current project has been completed.

The following example build output is the result of a non-multiprocessor build:

```
------ Build started: Project: ConsoleApp1, Configuration: Debug Win32 ------
```

```
Compiling...
stdafx.cpp
Compiling...
AssemblyInfo.cpp
ConsoleApp1.cpp
Generating Code...
Compiling resources...
Linking...
Build log was saved at "file://d:\Documents and Settings\username\My Documents\Visual Studi
o\Projects\Demo\ConsoleApp1\Debug\BuildLog.htm"
ConsoleApp1 - 0 error(s), 0 warning(s)

------ Build started: Project: ConsoleApp2, Configuration: Debug Win32 ------
Compiling...
stdafx.cpp
Compiling...
AssemblyInfo.cpp
ConsoleApp2.cpp
Generating Code...
Compiling resources...
Linking...
Build log was saved at "file://d:\Documents and Settings\username\My Documents\Visual Studi
o\Projects\Demo\ConsoleApp2\Debug\BuildLog.htm"
ConsoleApp2 - 0 error(s), 0 warning(s)

---------------------- Done ----------------------

    Build: 2 succeeded, 0 failed, 0 up-to-date, 0 skipped
```

See Also

Reference

How Can I Build Faster?

How to: Run Multiprocessor Builds in the IDE

This topic describes how to run Multiprocessor Builds from the IDE.

The number of projects that can build concurrently in the IDE is equal to the value of the **Maximum number of parallel project builds** property. For example, if you build a solution that comprises several projects while this property is set to 2, then up to two projects will build concurrently at a time.

Note
To enable multiprocessor builds in the IDE, the **Maximum number of parallel project builds** property must be set to a value greater than 1. For information on how to change this property, see How to: Set the Number of Concurrent Builds for Multiprocessor Builds.

Multiprocessor builds automatically run for build commands (such as **Build Solution**, **Rebuild Solution**, etc.) when they are enabled in the IDE. Build commands are available from the **Build** menu or from the shortcut menu that displays when you right-click project (or solution) nodes in Solution Explorer.

To run multiprocessor builds on all projects in a solution

1. Open the solution containing the projects that you want to build.
2. Click the **Build** menu.

   - OR -

   In **Solution Explorer**, right-click the solution node to bring up the shortcut menu.
3. Select the desired build command.

   The project system will perform the multiprocessor build action and display the build output in the Output Window.

Use the **Batch Build** command to specify which projects (in your solution) and which build configurations (such as Debug or Release), that you would like to build concurrently.

To run multiprocessor builds on a subset of projects in a solution

1. Open the solution containing the projects that you want to build.
2. Click the **Build** menu, and then click **Batch Build**.

   The Batch Build Dialog Box appears.
3. In the **Build** column, check the build configurations of the projects that you want to build.
4. Click the button for the build action that you want to perform (**Build** or **Rebuild**).

   The project system will perform the multiprocessor build action and display the build output in the Output window.

Example

The following example illustrates multiprocessor builds in the IDE. To set up this example, create a new solution named `MultiprocExample` that comprises two C++ console applications, one named `Console1` and the other named `Console2`.

Verify that multiprocessor builds are activated in the IDE, and then use the **Batch Build** command to run a multiprocessor build for `Console1` as Release and `Console2` as Debug. Your build output should appear similar to the following build output:

```
1>------ Build started: Project: Console1, Configuration: Release Win32 ------
2>------ Build started: Project: Console2, Configuration: Debug Win32 ------
2>Compiling...
1>Linking...
2>stdafx.cpp
1>Build log was saved at "file://d:\Documents and Settings\username\My Documents\Visual Stu
dio\Projects\MultiprocExample\Console1\Release\BuildLog.htm"
1>Console1 - 0 error(s), 0 warning(s)
1>
```

```
2>Compiling...
2>AssemblyInfo.cpp
2>Console2.cpp
2>Generating Code...
2>Compiling resources...
2>Linking...
2>Build log was saved at "file://d:\Documents and Settings\username\My Documents\Visual Stu
dio\Projects\MultiprocExample\Console2\Debug\BuildLog.htm"
2>Console2 - 0 error(s), 0 warning(s)
2>
---------------------- Done ----------------------

    Build: 2 succeeded, 0 failed, 0 up-to-date, 0 skipped
```

See Also

Tasks

How to: Run Multiprocessor Builds with VCBUILD

How to: Run Multiprocessor Builds with DEVENV Command Line Switches

How to: Run Multiprocessor Builds with VCBUILD

You can use VCBUILD to run Multiprocessor Builds from the command line. An advantage of building your projects using this utility is that it does not require Visual Studio to be installed on your computer.

The /M option activates multiprocessor builds for the VCBUILD utility.

To run multiprocessor builds using VCBUILD

1. Open a command window.

   Note

   To run VCBUILD, the path of the directory that stores this utility must be included in the PATH environment variable.

2. At the command prompt, use the following command line to build your projects:

   ```
   vcbuild /M<number> [options] [project|solution] [configuration]
   ```

   `<number>` specifies the amount of concurrent builds to run.

Example

The following example illustrates multiprocessor builds using the VCBUILD utility. To set up this example, create a new solution named `MultiprocExample` in an example directory (`C:\MyExamples`) that comprises two managed console applications, one named `Console1` and the other named `Console2`.

Use the following command line to run a multiprocessor build on `Console1` as Debug and `Console2` as Debug. The `/M2` option specifies to build the two projects concurrently.

```
C:\>vcbuild /M2 "C:\MyExamples\MultiprocExample\Console1\Console1.vcproj" Debug "C:\MyExamp
les\MultiprocExample\Console2\Console2.vcproj" Debug
```

Output

See Also

Tasks

How to: Run Multiprocessor Builds in the IDE
How to: Run Multiprocessor Builds with DEVENV Command Line Switches

How to: Run Multiprocessor Builds with DEVENV Command Line Switches

Running Multiprocessor Builds using DEVENV Command Line Switches is similar to running multiprocessor builds from the IDE. The number of projects that can build concurrently depends on the value of the **Maximum number of parallel project builds** property.

> **Note**
>
> To enable multiprocessor builds with DEVENV, the **Maximum number of parallel project builds** property must be set to a value greater than 1. For information on how to change this property, see How to: Set the Number of Concurrent Builds for Multiprocessor Builds.

When Multiprocessor builds are enabled with DEVENV command line switches, they will automatically run when you invoke DEVENV with the following build options:

- /Build (devenv.exe)
- /Rebuild (devenv.exe)
- /Clean (devenv.exe)

To run multiprocessor builds using DEVENV command line switches

1. Open a command window.

   > **Note**
   >
   > To set the appropriate environment variables to run Visual Studio tools in a command session, you may need to run VS VARS32.BAT or VCVARS32.BAT.

2. At the command prompt, enter the following command line to build your solution:

   ```
   devenv <solution>.sln [options] solutionconfig [/project projectnameorfile [/projectco
   nfig name]]
   ```

   `<solution>` specifies the name (and path, if necessary) to the solution file.

Example

The following example illustrates multiprocessor builds using DEVENV command line switches. To set up this example, create a new solution named `MultiprocExample` that comprises two C++ console applications, one named `Console1` and the other named `Console2`.

Use the following command line to run a multiprocessor rebuild for `Console1` and `Console2`, both in Debug configuration.

```
D:\>devenv "D:\Documents and Settings\username\My Documents\Visual Studio\Projects\Multipro
cExample\MultiprocExample.sln" /rebuild Debug
```

Output

```
1>------ Rebuild All started: Project: Console2, Configuration: Debug Win32 ----
--
2>------ Rebuild All started: Project: Console1, Configuration: Debug Win32 ----
--
1>Deleting intermediate and output files for project 'Console2', configuration '
Debug|Win32'
2>Deleting intermediate and output files for project 'Console1', configuration '
Debug|Win32'
1>Compiling...
2>Compiling...
```

```
1>stdafx.cpp
2>stdafx.cpp
1>Compiling...
2>Compiling...
1>Console2.cpp
2>Console1.cpp
1>AssemblyInfo.cpp
2>AssemblyInfo.cpp
1>Generating Code...
2>Generating Code...
1>Compiling resources...
2>Compiling resources...
2>Linking...
1>Linking...
1>Build log was saved at "file://d:\Documents and Settings\username\My Documents
\Visual Studio\Projects\MultiprocExample\Console2\Debug\BuildLog.htm"
1>Console2 - 0 error(s), 0 warning(s)
1>
2>Build log was saved at "file://d:\Documents and Settings\username\My Documents
\Visual Studio\Projects\MultiprocExample\Console1\Debug\BuildLog.htm"
2>Console1 - 0 error(s), 0 warning(s)
2>
---------------------- Done ----------------------

    Rebuild All: 2 succeeded, 0 failed, 0 skipped
```

See Also

Tasks

How to: Run Multiprocessor Builds in the IDE

How to: Run Multiprocessor Builds with VCBUILD

How to: Set the Number of Concurrent Builds for Multiprocessor Builds

This topic describes how to change or specify the number of concurrent builds for Multiprocessor Builds.

To set the number of concurrent builds

1. In the Visual Studio IDE, on the **Tools** menu, click **Options**.

   The Options Dialog Box will appear.

2. Expand the **Projects and Solutions** folder, and then select the **Build and Run** property page.

3. Enter an integer in the text box for the **Maximum number of parallel project builds** property. The highest value that you can set this property is 32.

Note
Build performance does not increase when you set the **Maximum number of parallel project builds** property to a v alue greater than the number of CPUs on your computer.

   To disable Multiprocessor builds, set this property equal to 1.

4. Click **OK**.

See Also

Tasks

How to: Run Multiprocessor Builds with VCBUILD
How to: Run Multiprocessor Builds with DEVENV Command Line Switches

How Do I Compile and Link C Code, Not C++?

Give your files the .c extension, for example `mysource.c`. The Visual C++ compiler automatically assumes that files with the .C extension are C files and not C++ files, and rejects C++ syntax and keywords (such as **public**, **private**, and **class**).

C++ files use the .cpp extension.

See the /TC, /TP compiler options for more information.

See Also

Reference

Frequently Asked Questions on Building

Compiling a C/C++ Program

For information on using the compiler, see the following sections:

- Setting Compiler Options
- Compiler Options
- Creating Precompiled Header Files
- Unicode Support in the Compiler and Linker

For more information, see 64-Bit Programming with Visual C++.

See Also

Other Resources

C/C++ Building Reference

Setting Compiler Options

Compiler options can be set either in the development environment or on the command line.

In the Development Environment

The topic for each compiler option discusses how it can be set in the development environment. See Compiler Options for a complete list.

Outside the Development Environment

You can set compiler (CL.exe) options:

- On the command line
- In command files
- In the CL environment variable

Options specified in the CL environment variable are used every time you invoke CL. If a command file is named in the CL environment variable or on the command line, the options specified in the command file are used. Unlike either the command line or the CL environment variable, a command file allows you to use multiple lines of options and filenames.

Compiler options are processed "left to right," and when a conflict is detected, the last (rightmost) option wins. The CL environment variable is processed before the command line, so in any conflicts between CL and the command line, the command line takes precedence.

Additional Compiler Topics

- Fast Compilation
- CL Invokes the Linker

See Also
Reference
Compiler Options
Other Resources
C/C++ Building Reference

Compiler Command-Line Syntax

The CL command line uses the following syntax:

```
CL [option...] file... [option | file]... [lib...] [@command-file] [/link link-opt...]
```

The following table describes input to the CL command.

Entry	Meaning
option	One or more CL options. Note that all options apply to all specified source files. Options are specified by either a forward slash (/) or a dash (–). If an option takes an argument, the option's description documents whether a space is allowed between the option and the arguments. Option names (except for the /HELP option) are case sensitive. See Order of CL Options for more information.
file	The name of one or more source files, .obj files, or libraries. CL compiles source files and passes the names of the .obj files and libraries to the linker. See CL Filename Syntax for more information.
lib	One or more library names. CL passes these names to the linker.
command-file	A file that contains multiple options and filenames. See CL Command Files for more information.
link-opt	One or more linker options. CL passes these options to the linker.

You can specify any number of options, filenames, and library names, as long as the number of characters on the command line does not exceed 1024, the limit dictated by the operating system.

For information about the return value of cl.exe, see Return Value of cl.exe .

Note
The command-line input limit of 1024 characters is not guaranteed to remain the same in future releases of Windows.

See Also

Reference

Setting Compiler Options

Compiler Options

CL Filename Syntax

CL accepts files with names that follow FAT, HPFS, or NTFS naming conventions. Any filename can include a full or partial path. A full path includes a drive name and one or more directory names. CL accepts filenames separated either by backslashes (\) or forward slashes (/). A partial path omits the drive name, which CL assumes to be the current drive. If you don't specify a path, CL assumes the file is in the current directory.

The filename extension determines how files are processed. C and C++ files, which have the extension .c, .cxx, or .cpp, are compiled. Other files, including .obj files, libraries (.lib), and module-definition (.def) files, are passed to the linker without being processed.

See Also

Reference

Compiler Command-Line Syntax

Order of CL Options

Options can appear anywhere on the CL command line, except for the /link option, which must occur last. The compiler begins with options specified in the CL environment variable and then reads the command line from left to right — processing command files in the order it encounters them. Each option applies to all files on the command line. If CL encounters conflicting options, it uses the rightmost option.

See Also

Reference

Compiler Command-Line Syntax

Return Value of cl.exe

cl.exe returns zero for success (no errors) and non-zero otherwise.

The return value of cl.exe can be useful if you are compiling from a script, or .bat file. You should still plan to capture the output of the compiler, in case there are errors or warnings, so you can resolve them.

Remarks

The following is a sample .bat file that uses the return value of cl.exe.

```
echo off
cl /W4 t.cpp
@if ERRORLEVEL == 0 (
   goto good
)

@if ERRORLEVEL != 0 (
   goto bad
)

:good
   echo "clean compile"
   echo %ERRORLEVEL%
   goto end

:bad
   echo "error or warning"
   echo %ERRORLEVEL%
   goto end

:end
```

See Also

Reference

Compiler Command-Line Syntax

CL Environment Variables

The CL tool uses the following environment variables:

- CL, if defined. The CL tool processes options and arguments defined in the CL environment variable before processing the command line.
- INCLUDE, which must point to the \include subdirectory of your Visual C++ installation.
- LIBPATH, which specifies directories to search for metadata files referenced with #using. See **#using** for more information on LIBPATH.

You can set the CL environment variable using the following syntax:

```
SET CL=[ [option] ... [file] ...] [/link link-opt ...]
```

For details on the arguments to the CL environment variable, see Compiler Command-Line Syntax.

You can use this environment variable to define the files and options you use most often and use the command line to define specific files and options for specific purposes. The CL environment variable is limited to 1024 characters (the command-line input limit).

You cannot use the /D option to define a symbol that uses an equal sign (=). You can substitute the number sign (#) for an equal sign. In this way, you can use the CL environment variable to define preprocessor constants with explicit values (for example, `/DDEBUG#1`).

For related information, see Set Environment Variables.

Examples

The following example of a CL environment variable setting:

```
SET CL=/Zp2 /Ox /I\INCLUDE\MYINCLS \LIB\BINMODE.OBJ
```

is equivalent to the following CL command:

```
CL /Zp2 /Ox /I\INCLUDE\MYINCLS \LIB\BINMODE.OBJ INPUT.C
```

The following example causes CL to compile the source files FILE1.c and FILE2.c, and then link the object files FILE1.obj, FILE2.obj, and FILE3.obj:

```
SET CL=FILE1.C FILE2.C
CL FILE3.OBJ
```

This has the same effect as the following command line:

```
CL FILE1.C FILE2.C FILE3.OBJ
```

See Also

Reference

Setting Compiler Options

Compiler Options

CL Command Files

A command file is a text file that contains options and filenames you would otherwise type on the command line or specify using the CL environment variable. CL accepts a compiler command file as an argument in the CL environment variable or on the command line. Unlike either the command line or the CL environment variable, a command file allows you to use multiple lines of options and filenames.

Options and filenames in a command file are processed according to the location of a command filename within the CL environment variable or on the command line. However, if the /link option appears in the command file, all options on the rest of the line are passed to the linker. Options in subsequent lines in the command file and options on the command line after the command file invocation are still accepted as compiler options. For more information on how the order of options affects their interpretation, see Order of CL Options.

A command file must not contain the CL command. Each option must begin and end on the same line; you cannot use the backslash (\) to combine an option across two lines.

A command file is specified by an at sign (@) followed by a filename; the filename can specify an absolute or relative path.

For example, if the following command is in a file named RESP:

```
/Og /link LIBC.LIB
```

and you specify the following CL command:

```
CL /Ob2 @RESP MYAPP.C
```

the command to CL is as follows:

```
CL /Ob2 /Og MYAPP.C /link LIBC.LIB
```

Note that the command line and the command-file commands are effectively combined.

See Also

Reference

Setting Compiler Options

Compiler Options

Fast Compilation

To increase the speed of your compiles:

- Use minimal rebuild, in which the C++ compiler recompiles a source file only if it is dependent on changes to a class in a header file.
- Create Precompiled Header Files and use the precompiled header options.

See Also

Reference

Compiler Options

Setting Compiler Options

CL Invokes the Linker

CL automatically invokes the linker after compiling unless the /c option is used. CL passes to the linker the names of .obj files created during compiling and the names of any other files specified on the command line. The linker uses the options listed in the LINK environment variable. You can use the /link option to specify linker options on the CL command line. Options that follow the /link option override those in the LINK environment variable. The options in the following table suppress linking.

Option	Description
/c	Compile without linking
/E, /EP, /P	Preprocess without compiling or linking
/Zg	Generate function prototypes
/Zs	Check syntax

For further details about linking, see Linker Options.

Example

Assume that you are compiling three C source files: MAIN.c, MOD1.c, and MOD2.c. Each file includes a call to a function defined in a different file:

- MAIN.c calls the function `func1` in MOD1.c and the function `func2` in MOD2.c.
- MOD1.c calls the standard library functions **printf_s** and **scanf_s**.
- MOD2.c calls graphics functions named `myline` and `mycircle`, which are defined in a library named MYGRAPH.lib.

To build this program, compile with the following command line:

```
CL MAIN.c MOD1.C MOD2.C MYGRAPH.lib
```

CL first compiles the C source files and creates the object files MAIN.obj, MOD1.obj, and MOD2.obj. The compiler places the name of the standard library in each .obj file. For more details, see Use Run-Time Library.

CL passes the names of the .obj files, along with the name MYGRAPH.lib, to the linker. The linker resolves the external references as follows:

1. In MAIN.obj, the reference to `func1` is resolved using the definition in MOD1.obj; the reference to `func2` is resolved using the definition in MOD2.obj.
2. In MOD1.obj, the references to **printf_s** and **scanf_s** are resolved using the definitions in the library that the linker finds named within MOD1.obj.
3. In MOD2.obj, the references to `myline` and `mycircle` are resolved using the definitions in MYGRAPH.lib.

See Also

Reference

Compiler Options

Setting Compiler Options

Compiler Options

cl.exe is a tool that controls the Microsoft C and C++ compilers and linker. cl.exe can only be run on Windows 2000, Windows XP and Windows Server 2003 operating systems.

The compilers produce Common Object File Format (COFF) object (.obj) files. The linker produces executable (.exe) files or dynamic-link libraries (DLLs).

Note that all compiler options are case sensitive.

To compile without linking, use /c.

Finding an Option

To find a particular compiler option, see one of the following lists:

- Compiler Options Listed Alphabetically
- Compiler Options Listed by Category

Specifying Options

For information on specifying options, see:

- Setting Compiler Options
- Fast Compilation
- CL Invokes the Linker

Related Build Tools

Use NMAKE to build your output file.

Use BSCMAKE to support class browsing.

Linker options also affect how your program is built.

See Also

Reference

Setting Compiler Options

Other Resources

C/C++ Building Reference

Compiler Options Listed by Category

The following is a comprehensive categorical list of compiler options. For an alphabetical list, see Compiler Options Listed Alphabetically.

Optimization

Option	Purpose
/O1	Creates small code
/O2	Creates fast code
/Ob	Controls inline expansion
/Od	Disables optimization
/Og	Uses global optimizations
/Oi	Generates intrinsic functions
/Os	Favors small code
/Ot	Favors fast code
/Ox	Uses maximum optimization (/Ob2gity /Gs)
/Oy	Omits frame pointer (x86 only)

Code Generation

Option	Purpose
/arch	Use SSE or SSE2 instructions in code generation (x86 only)
/bigobj	Increases the number of addressable sections in an .obj file.
/clr	Produces an output file to run on the common language runtime
/EH	Specifies the model of exception handling
/favor	Produces code that is optimized for a specific x64 architecture or for the specifics of micro-architectures in both the A MD64 and Extended Memory 64 Technology (EM64T) architectures.
/fp	Specify floating-point behavior.
/G1	Optimize for Itanium processor. Only available in the IPF cross compiler or IPF native compiler.
/G2	Optimize for Itanium2 processor. Only available in the IPF cross compiler or IPF native compiler.
/Gd	Uses the **__cdecl** calling convention (x86 only)
/Ge	Activates stack probes
/GF	Enables string pooling

/Gh	Calls hook function **_penter**
/GH	Calls hook function **_pexit**
/GL	Enables whole program optimization
/Gm	Enables minimal rebuild
/GR	Enables run-time type information (RTTI)
/Gr	Uses the **__fastcall** calling convention (x86 only)
/Gs	Controls stack probes
/GT	Supports fiber safety for data allocated using static thread-local storage
/GX	Enables synchronous exception handling
/Gy	Enables function-level linking
/Gz	Uses the **__stdcall** calling convention (x86 only)
/MD	Creates a multithreaded DLL using MSVCRT.lib
/MDd	Creates a debug multithreaded DLL using MSVCRTD.lib
/MT	Creates a multithreaded executable file using LIBCMT.lib
/MTd	Creates a debug multithreaded executable file using LIBCMTD.lib

Output Files

Option	Purpose
/FA	Creates a listing file Sets listing file name
/Fa	Creates a listing file Sets listing file name
/Fd	Renames program database file
/Fe	Renames the executable file
/Fm	Creates a mapfile
/Fo	Creates an object file
/Fp	Specifies a precompiled header file name
/FR /Fr	Generates browser files
/Fx	Merges injected code with source file

Debugging

Option	Purpose
/GS	Buffers security check

/GZ	Same as /RTC1
/homeparams	Forces parameters passed in registers to be written to their locations on the stack upon function entry. This compiler option is only for the x64 compilers (native and cross compile).
/RTC	Enables run-time error checking
/Wp64	Detects 64-bit portability problems
/Yd	Places complete debugging information in all object files
/Yl	Injects a PCH reference when creating a debug library
/Z7	Generates C 7.0–compatible debugging information
/Zi	Generates complete debugging information
/ZI	Includes debug information in a program database compatible with Edit and Continue (x86 only)
/Zx	Generates debuggable optimized code. Only available in the IPF cross compiler or IPF native compiler.

Preprocessor

Option	Purpose
/AI	Specifies a directory to search to resolve file references passed to the #using directive
/C	Preserves comments during preprocessing
/D	Defines constants and macros
/E	Copies preprocessor output to standard output
/EP	Copies preprocessor output to standard output
/FI	Preprocesses the specified include file
/FU	Forces the use of a file name, as if it had been passed to the #using directive
/I	Searches a directory for include files
/P	Writes preprocessor output to a file
/U	Removes a predefined macro
/u	Removes all predefined macros
/X	Ignores the standard include directory

Language

Option	Purpose
/openmp	Enables #pragma omp in source code.
/vd	Suppresses or enables hidden vtordisp class members

/vmb	Uses best base for pointers to members
/vmg	Uses full generality for pointers to members
/vmm	Declares multiple inheritance
/vms	Declares single inheritance
/vmv	Declares virtual inheritance
/Za	Disables language extensions
/Zc	Specifies standard behavior under /Ze
/Ze	Enables language extensions
/Zg	Generates function prototypes
/Zl	Removes default library name from .obj file
/Zp *n*	Packs structure members
/Zs	Checks syntax only

Linking

Option	Purpose
/F	Sets stack size
/LD	Creates a dynamic-link library
/LDd	Creates a debug dynamic-link library
/LN	Create an MSIL module.
/link	Passes the specified option to LINK
/MD	Compiles to create a multithreaded DLL, using MSVCRT.lib
/MDd	Compiles to create a debug multithreaded DLL, using MSVCRTD.lib
/MT	Compiles to create a multithreaded executable file, using LIBCMT.lib
/MTd	Compiles to create a debug multithreaded executable file, using LIBCMTD.lib

Precompiled Header

Option	Purpose
/Y-	Ignores all other precompiled-header compiler options in the current build
/Yc	Creates a precompiled header file
/Yd	Places complete debugging information in all object files
/Yu	Uses a precompiled header file during build

Miscellaneous

Option	Purpose
@	Specifies a response file
/?	Lists the compiler options
/analyze	Enables code analysis.
/c	Compiles without linking
/doc	Process documentation comments to an XML file.
/errorReport	Allows you to provide internal compiler error (ICE) information directly to the Visual C++ team.
/FC	Display full path of source code files passed to cl.exe in diagnostic text.
/H	Restricts the length of external (public) names
/HELP	Lists the compiler options
/hotpatch	Creates a hotpatchable image.
/J	Changes the default **char** type
/nologo	Suppresses display of sign-on banner
QIfist	Suppresses the call of the helper function **_ftol** when a conversion from a floating-point type to an integral type is required (x86 only)
/QIPF_B	Does not generate sequences of instructions that give unexpected results, according to the errata for the B CPU stepping. (IPF only)
/QIPF_C	Does not generate sequences of instructions that give unexpected results, according to the errata for the C CPU stepping. (IPF only)
/QIPF_fr32	Do not use upper 96 floating-point registers. (IPF only)
/QIPF_noPIC	Generates an image with position dependent code (IPF only).
/QIPF_restrict_plabels	Enhance performance for programs that do not create functions at runtime. (IPF only)
/showIncludes	Displays a list of all include files during compilation
/Tc /TC	Specifies a C source file
/Tp /TP	Specifies a C++ source file
/V	Sets the version string
/W	Sets warning level
/w	Disables all warnings
/Wall	Enables all warnings, including warnings that are disabled by default

/WL	Enables one-line diagnostics for error and warning messages when compiling C++ source code from the command line
/Zm	Specifies the precompiled header memory allocation limit

See Also

Reference

Compiler Options

Setting Compiler Options

Other Resources

C/C++ Building Reference

Compiler Options Listed Alphabetically

The following is a comprehensive alphabetical list of compiler options. For a categorical list, see the Compiler Options Listed by Category.

Option	Purpose
@	Specifies a response file.
/?	Lists the compiler options.
/AI	Specifies a directory to search to resolve file references passed to the #using directive.
/analyze	Enable code analysis.
/arch	Use SSE or SSE2 instructions in code generation (x86 only).
/bigobj	Increases the number of addressable sections in an .obj file.
/C	Preserves comments during preprocessing.
/c	Compiles without linking.
/clr	Produces an output file to run on the common language runtime.
/D	Defines constants and macros.
/doc	Process documentation comments to an XML file.
/E	Copies preprocessor output to standard output.
/EH	Specifies the model of exception handling.
/EP	Copies preprocessor output to standard output.
/errorReport	Allows you to provide internal compiler error (ICE) information directly to the Visual C++ team.
/F	Sets stack size.
/favor	Produces code that is optimized for a specific x64 architecture or for the specifics of micro-architectures in both the AMD64 and Extended Memory 64 Technology (EM64T) architectures.
/FA	Creates a listing file.
/Fa	Sets listing file name.
/FC	Display full path of source code files passed to cl.exe in diagnostic text.
/Fd	Renames program database file.
/Fe	Renames the executable file.
/FI	Preprocesses the specified include file.

/Fm	Creates a mapfile.
/Fo	Creates an object file.
/fp	Specify floating-point behavior.
/Fp	Specifies a precompiled header file name.
/FR /Fr	Generates browser files.
/FU	Forces the use of a file name as if it had been passed to the #using directive.
/Fx	Merges injected code with source file.
/G1	Optimize for Itanium processor. Only available in the IPF cross compiler or IPF native compiler.
/G2	Optimize for Itanium2 processor (default between /G1 and /G2). Only available in the IPF cross compiler or IPF native compiler.
/GA	Optimizes code for Windows application.
/Gd	Uses the **__cdecl** calling convention (x86 only).
/Ge	Activates stack probes.
/GF	Enables string pooling.
/GH	Calls hook function **_pexit**.
/Gh	Calls hook function **_penter**.
/GL	Enables whole program optimization.
/Gm	Enables minimal rebuild.
/GR	Enables run-time type information (RTTI).
/Gr	Uses the **__fastcall** calling convention (x86 only).
/GS	Buffers security check.
/Gs	Controls stack probes.
/GT	Supports fiber safety for data allocated using static thread-local storage.
/GX	Enables synchronous exception handling.
/Gy	Enables function-level linking.
/GZ	Same as /RTC1./RTC (Run-Time Error Checks)

/Gz	Uses the **__stdcall** calling convention (x86 only).
/H	Restricts the length of external (public) names.
/HELP	Lists the compiler options.
/homeparams	Forces parameters passed in registers to be written to their locations on the stack upon function entry. T his compiler option is only for the x64 compilers (native and cross compile).
/hotpatch	Creates a hotpatchable image.
/I	Searches a directory for include files.
/J	Changes the default **char** type.
/LD	Creates a dynamic-link library.
/LDd	Creates a debug dynamic-link library.
/link	Passes the specified option to LINK.
/LN	Creates an MSIL module.
/MD	Creates a multithreaded DLL using MSVCRT.lib.
/MDd	Creates a debug multithreaded DLL using MSVCRTD.lib.
/MT	Creates a multithreaded executable file using LIBCMT.lib.
/MTd	Creates a debug multithreaded executable file using LIBCMTD.lib.
/nologo	Suppresses display of sign-on banner.
/O1	Creates small code.
/O2	Creates fast code.
/Ob	Controls inline expansion.
/Od	Disables optimization.
/Og	Uses global optimizations.
/Oi	Generates intrinsic functions.
/openmp	Enables #pragma omp in source code.
/Os	Favors small code.
/Ot	Favors fast code.
/Ox	Uses maximum optimization (/Ob2gity /Gs).

/Oy	Omits frame pointer (x86 only).
/QIfist	Suppresses **_ftol** when a conversion from a floating-point type to an integral type is required (x86 only).
/QIPF_B	Does not generate sequences of instructions that give unexpected results, according to the errata for the B CPU stepping. (IPF only).
/QIPF_C	Does not generate sequences of instructions that give unexpected results, according to the errata for the C CPU stepping. (IPF only).
/QIPF_fr32	Do not use upper 96 floating-point registers. (IPF only).
/QIPF_noPIC	Generates an image with position dependent code (IPF only).
/QIPF_restrict_plabels	Enhances performance for programs that do not create functions at runtime. (IPF only).
/P	Writes preprocessor output to a file.
/RTC	Enables run-time error checking.
/showIncludes	Displays a list of include files during compilation.
/Tc /TC	Specifies a C source file.
/Tp /TP	Specifies a C++ source file.
/U	Removes a predefined macro.
/u	Removes all predefined macros.
/V	Sets the version string.
/vd	Suppresses or enables hidden vtordisp class members.
/vmb	Uses best base for pointers to members.
/vmg	Uses full generality for pointers to members.
/vmm	Declares multiple inheritance.
/vms	Declares single inheritance.
/vmv	Declares virtual inheritance.
/W	Sets warning level.
/w	Disables all warnings.
/Wall	Enables all warnings, including warnings that are disabled by default.

/WL	Enables one-line diagnostics for error and warning messages when compiling C++ source code from the command line.
/Wp64	Detects 64-bit portability problems.
/X	Ignores the standard include directory.
/Y-	Ignores all other precompiled-header compiler options in the current build.
/Yc	Creates a precompiled header file.
/Yd	Places complete debugging information in all object files.
/Yl	Injects a PCH reference when creating a debug library
/Yu	Uses a precompiled header file during build.
/Z7	Generates C 7.0–compatible debugging information.
/Za	Disables language extensions.
/Zc	Specifies standard behavior under /Ze./Za, /Ze (Disable Language Extensions)
/Ze	Enables language extensions.
/Zg	Generates function prototypes.
/ZI	Includes debug information in a program database compatible with Edit and Continue.
/Zi	Generates complete debugging information.
/Zl	Removes default library name from .obj file (x86 only).
/Zm	Specifies the precompiled header memory allocation limit.
/Zp	Packs structure members.
/Zs	Checks syntax only.
/Zx	Generates debuggable optimized code. Only available in the IPF cross compiler or IPF native compiler.

See Also

Reference

Compiler Options

Setting Compiler Options

Other Resources

C/C++ Building Reference

@ (Specify a Compiler Response File)

Specifies a compiler response file.

```
@response_file
```

Arguments

response_file

A text file containing compiler commands.

Remarks

A response file can contain any commands that you would specify on the command line. This can be useful if you are working on a machine (Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition) and your command-line arguments exceed 127 characters.

It is not possible to specify the **@** option from within a response file. That is, a response file cannot embed another response file.

From the command line you can specify as many response file options (for example, `@respfile.1 @respfile.2`) as you want.

To set this compiler option in the Visual Studio development environment

- A response file cannot be specified from within the development environment and must be specified at the command line.

To set this compiler option programmatically

- This compiler option cannot be changed programmatically.

See Also

Reference

Compiler Options

Setting Compiler Options

/AI (Specify Metadata Directories)

Specifies a directory that the compiler will search to resolve file references passed to the **#using** directive.

```
/AIdirectory
```

Arguments
directory

The directory or path for the compiler to search.

Remarks

Only one directory can be passed to an **/AI** invocation. Specify one **/AI** option for each path you want the compiler to search.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **General** property page.
4. Modify the **Resolve #using References** property.

To set this compiler option programmatically

- See AdditionalUsingDirectories.

See Also
Reference
Compiler Options
Setting Compiler Options
The #using Directive

/analyze (Enterprise Code Analysis)

Enable code analysis.

```
/analyze[:WX-]
```

Remarks

Specifying **/analyze:WX-** means code analysis warnings will not be treated as errors when compiling with **/WX**. See /w, /Wn, /WX, /Wall, /wln, /wdn, /wen, /won (Warning Level) for more information.

/analyze is only available in Enterprise (team development) versions for x86 compilers.

For more information on code analysis, see Code Analysis for C/C++ Overview and Code Analysis for C/C++ Warnings.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Expand the **Configuration Properties** node.
3. Expand the **Code Analysis** node.
4. Select the **General** property page.
5. Modify one or more of the **Code Analysis** properties.

To set this linker option programmatically

- See EnablePREfast.

See Also

Reference

Compiler Options

Setting Compiler Options

/arch (Minimum CPU Architecture)

Specifies the architecture for code generation using the Streaming SIMD Extensions (SSE) and Streaming SIMD Extensions 2 (SSE2) instructions.

```
/arch:[SSE|SSE2]
```

Remarks

Note

/arch is only available when compiling for the x86 platform. This compiler option is not available when compiling for x64 or Itanium.

The SSE instructions exist in various Pentium and AMD Athlon processors. The SSE2 instructions only exist on the Pentium 4 processor.

/arch:SSE allows the compiler to use the SSE instructions, and **/arch:SSE2** allows the compiler to use the SSE2 instructions.

_M_IX86_FP indicates which, if any, **/arch** compiler option was used; see Predefined Macros for more information.

The optimizer will choose when and how to make use of the SSE and SSE2 instructions when **/arch** is specified. SSE and SSE2 instructions will be used for some scalar floating-point computations, when it is determined that it is faster to use the SSE/SSE2 instructions and registers rather than the x87 floating-point register stack. As a result, your code will actually use a mixture of both x87 and SSE/SSE2 for floating-point computations. Additionally, with **/arch:SSE2**, SSE2 instructions can be used for some 64-bit integer operations.

In addition to using the SSE and SSE2 instructions, the compiler will also use other instructions that are present on the processor revisions that support SSE and SSE2. An example is the CMOV instruction that first appeared in the Pentium Pro revision of the Intel processors.

When compiling with /clr (Common Language Runtime Compilation), **/arch** will have no effect on code generation for managed functions. **/arch** only affects code generation for native functions.

/arch and /Qlfist (Suppress _ftol) can not be used on the same compiland.

In particular, if the user does not use **_controlfp** to modify the FP control word, then the run-time startup code will set the x87 FPU control word precision-control field to 53-bits, so all float and double operations within an expression will occur with 53-bit significand and 15-bit exponent. All SSE single-precision operations will, however, use a 24-bit significand/8-bit exponent, and SSE2 double-precision operations will use a 53-bit significand/11-bit exponent. For more information, see _control87, _controlfp, __control87_2.

To illustrate, these differences are possible within a single expression tree, not in cases where there is a user assignment involved after each subexpression:

```
r = f1 * f2 + d;  // Different results are possible on SSE/SSE2.
```

Against:

```
    t = f1 * f2;   // Do f1 * f2, round to the type of t.
    r = t + d;     // This should produce the same overall result
                   // regardless whether x87 stack or SSE/SSE2 is used.
```

controlfp does not change the MXCSR control bits, so with **/arch:SSE2**, any functionality that depends on using **controlfp** will be broken.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.

3. Click the **Code Generation** property page.
4. Modify the **Enable Enhanced Instruction Set** property.

To set this compiler option programmatically

- See EnableEnhancedInstructionSet.

See Also
Reference
Compiler Options
Setting Compiler Options

/bigobj (Increase Number of Sections in .obj file)

/bigobj increases the number of sections that an object file can contain.

```
/bigobj
```

Remarks

By default, an object file can hold up to 65,536 (2^16) addressable sections. **/bigobj** increases that address capacity to 4,294,967,296 (2^32).

Most modules will never generate an .obj file that contains more than 65,536 sections. However, machine generated code, or code that makes heavy use of template libraries may require .obj files that can hold more sections.

An .obj file produced with **/bigobj** can only be consumed by a linker that shipped in Visual C++ 2005 (or later). Linkers that shipped prior to Visual C++ 2005 cannot read .obj files that were produced with **/bigobj**.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also
Reference
Compiler Options
Setting Compiler Options

/C (Preserve Comments During Preprocessing)

Preserves comments during preprocessing.

```
/C
```

Remarks

This compiler option requires the **/E**, **/P**, or **/EP** option.

The following code sample will display the source code comment.

```
// C_compiler_option.cpp
// compile with: /E /C /c
int i;   // a variable
```

This sample will produce the following output.

```
#line 1 "C_compiler_option.cpp"
int i;   // a variable
```

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Preprocessor** property page.
4. Modify the **Keep Comments** property.

To set this compiler option programmatically

- See KeepComments.

See Also

Reference

Compiler Options
Setting Compiler Options
/E (Preprocess to stdout)
/P (Preprocess to a File)
/EP (Preprocess to stdout Without #line Directives)

/c (Compile Without Linking)

Prevents the automatic call to LINK.

```
/c
```

Remarks

Compiling with **/c** creates .obj files only. You must call LINK explicitly with the proper files and options to perform the linking phase of the build.

Any internal project created in the development environment uses the **/c** option by default.

To set this compiler option in the Visual Studio development environment

- This option is not available from within the development environment.

To set this compiler option programmatically

- To set this compiler option programmatically, see CompileOnly.

Example

The following command line creates the object files FIRST.obj and SECOND.obj. THIRD.obj is ignored.

```
CL /c FIRST.C SECOND.C THIRD.OBJ
```

To create an executable file, you must invoke LINK:

```
LINK firsti.obj second.obj third.obj /OUT:filename.exe
```

See Also

Reference

Compiler Options

Setting Compiler Options

/clr (Common Language Runtime Compilation)

Enables applications and components to use features from the common language runtime (CLR).

```
/clr[:options]
```

Arguments
options

One or more of the following, separated by a comma:

/clr

Creates metadata for your application that can be consumed by other CLR applications, and allows your application to consume types and data in the metadata of other CLR components.

For more information, see:

- Mixed (Native and Managed) Assemblies
- How To: Migrate to /clr

/clr:pure

Produces an MSIL-only output file with no native executable code, although it can contain native types compiled to MSIL.

For more information, see Pure and Verifiable Code.

/clr:safe

Produces an MSIL-only (no native executable code) and verifiable output file. **/clr:safe** enables verification diagnostics (PEVerify Tool (Peverify.exe)).

For more information, see Writing Verifiably Type-Safe Code.

/clr:oldSyntax

Enables Managed Extensions for C++ syntax, the original Visual C++ syntax for CLR programming.

Managed Extensions for C++ syntax is deprecated in Microsoft Visual C++ 2005, You should only use **/clr:oldSyntax** if you are maintaining a Visual C++ application that uses Managed Extensions for C++. If you are developing a new application, use the updated syntax; see New Language Features in Visual C++ for more information.

If you have a Managed Extensions for C++ application, you can begin to port your project to use the new syntax; see Porting and Upgrading Programs for more information.

/clr:noAssembly

Specifies that an assembly manifest should not be inserted into the output file. By default, **noAssembly** is not in effect, and an assembly manifest is inserted into the output file.

A managed program that does not have an assembly metadata in the manifest is called a *module*. See /LN (Create MSIL Module) for reasons to generate a module.

noAssembly can only be used when a DLL is the output file of the compilation. If you compile with /c (Compile Without Linking) and **/clr:noAssembly**, then specify the /NOASSEMBLY (Create a MSIL Module) option in the linker phase to create a module.

noAssembly is deprecated in Visual C++ 2005. Use /LN (Create MSIL Module) instead. For more information, see Deprecated Compiler Options.

Before Visual C++ 2005, **/clr:noAssembly** implied **/clr**. However, **/clr** now also supports **/clr:oldSyntax**, so you must specify one **/clr** form when you specify **/clr:noAssembly**. For example, **/clr:noAssembly /clr** creates a module using the new Visual C++ CLR syntax and **/clr:noAssembly,oldSyntax** creates a module using Managed Extensions for C++.

Before Visual C++ 2005, **/clr:noAssembly** required **/LD**. **/LD** is now implied with you specify **/clr:noAssembly**.

/clr:initialAppDomain

Allows a Visual C++ application to run on version 1 of the common language runtime. If you use **initialAppDomain**, then you may see some of the problems discussed in Knowledge Base article Q309694. You can find Knowledge Base articles on the MSDN Library media or at http://support.microsoft.com/support/.

An application compiled with **initialAppDomain** should never be used by an application using ASP.NET. Upgrade to a newer runtime to do ASP.NET work with C++.

Remarks

Managed code is code that can be inspected and managed by the common language runtime. Managed code can access managed objects.

Also see /clr Restrictions.

See New Language Features in Visual C++ for information on how to develop applications that define and consume managed types.

An application compiled with **/clr** may or may not contain managed data.

To allow debugging on a managed application, see /ASSEMBLYDEBUG (Add DebuggableAttribute).

Only CLR types will be instantiated on the garbage-collected heap. For more information, see Classes and Structs (Managed). To compile a function to native code, use the **unmanaged** pragma. For more information, see managed, unmanaged.

By default, **/clr** is not in effect. When **/clr** is in effect, **/MD** is also in effect (see for /MD, /MT, /LD (Use Run-Time Library) more information). **/MD** ensures that the dynamically linked, multithreaded versions of the runtime routines are selected from the standard header (.h) files. Multithreading is necessary for managed programming in part because the CLR garbage collector runs finalizers in an auxiliary thread.

If you compile with **/c**, you can specify the CLR type (IJW, safe, or pure) of the resulting output file with /CLRIMAGETYPE (Specify Type of CLR Image).

/clr implies **/EHa**, and no other **/EH** options are allowed with **/clr**. See /EH (Exception Handling Model) for more information.

For information on how to determine the CLR image type of a file, see /CLRHEADER.

All modules passed to a given invocation of the linker must have been compiled with the same run-time library compiler option (**/MD** or **/LD**).

Use the /ASSEMBLYRESOURCE (Embed a Managed Resource) linker option to embed a resource in an assembly. /DELAYSIGN (Partially Sign an Assembly), /KEYCONTAINER (Specify a Key Container to Sign an Assembly), and /KEYFILE (Specify Key or Key Pair to Sign an Assembly) linker options also let you customize how an assembly is created.

When **/clr** is used, the **_MANAGED** symbol is defined to be 1. For more information, see Predefined Macros.

The global variables in a native object files will be initialized first (during DllMain if the executable is a DLL), and then the global variables in the managed section will be initialized (before any managed code is run). **#pragma** init_seg only affects the order of initialization within the managed and unmanaged categories.

Compiling with **/clr:safe** is analogous to compiling with /platform:anycpu in languages such as C#.

Safe and Pure Images

A pure image will use a CLR version of the C run-time library. However, the CRT is not verifiable, so you cannot use the CRT when compiling with **/clr:safe**. For more information, see C Run-Time Libraries.

Examples of native code that cannot appear in a pure image include inline assembly, setjmp, or longjmp.

Each entry point of a pure or safe image is managed. When compiling with **/clr**, the entry point is native. For more information, see __clrcall.

When compiling with **/clr:safe**, variables are by default appdomain, and cannot be per process. With **/clr:pure**, **appdomain** is the default, but you can use process variables.

When running a 32-bit .exe that was compiled **/clr** or **/clr:pure** on a 64-bit operating system, the application will be run under WOW64, which allows a 32-bit application to run by the 32-bit CLR on a 64-bit operating system. By default, an .exe compiled with **/clr:safe** will be run in the 64-bit CLR on a computer running a 64-bit operating system (on a 32-bit operating system the same .exe would run in the 32-bit CLR). However, it is possible that your safe application loads a 32-bit component. In that case, a safe image running under the operating system's 64-bit support will fail when it loads the 32-bit application (BadFormatException). To ensure that a safe image continues to run when it loads a 32-bit image on a 64-bit operating system, you must use /CLRIMAGETYPE (Specify Type of CLR Image) to change the metadata (.corflags), marking it to be run under WOW64. A sample command line follows (substitute your own entry symbol):

```
cl /clr:safe t.cpp /link /clrimagetype:pure /entry:?main@@$$HYMHXZ /subsystem:console
```

For information on getting a decorated name, see Using a Listing to View Decorated Names. For more information on 64-bit programming, see 64-bit Programming (How Do I in Visual C++).

For samples, walkthroughs and more information, see:

- Walkthrough: Using /clr:pure Features
- How to: Migrate to /clr:pure
- CppLangFeat Sample: Language Features Overview
- Pure and Verifiable Code

Metadata and Unnamed Classes

Unnamed classes will appear in metadata named as follows: **$UnnamedClass$***crc-of-current-file-name***$***index***$**, where *index* is a sequential count of the unnamed classes in the compilation. For example, the following code sample will generate an unnamed class in metadata:

```
// clr_unnamed_class.cpp
// compile with: /clr /LD
class {} x;
```

Use ildasm.exe to view metadata.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **Configuration Properties** folder.
3. Click the **General** property page.
4. Modify the **Common Language Runtime support** property.

   See /NOASSEMBLY (Create a MSIL Module) for information about how to create a module.

Note
When **/clr** is enabled in a project's **Property Pages** dialog box, the compiler option properties that are not compatible with **/clr** will also be adjusted, as necessary. For example, if **/RTC** is set and then **/clr** is enabled, **/RTC** will be turned off.

   Also, when you debug a **/clr** application, the **Debugger Type** property should be set to **Mixed** or **Managed only**. For more information, see Project Settings for a C++ Debug Configuration.

To set this compiler option programmatically

- See CompileAsManaged.

See Also
Reference
Compiler Options
Setting Compiler Options

/clr Restrictions

Note the following restrictions on the use of **/clr**:

- In a structured exception handler, there are restrictions on using **_alloca** when compiling with **/clr**. For more information, see _alloca.
- The use of run-time error checks is not valid with **/clr**. For more information, see Run-Time Error Checks.
- When **/clr** is used to compile a program that only uses standard C++ syntax, the following guidelines apply to the use of inline assembly:
  - Inline assembly code that assumes knowledge of the native stack layout, calling conventions outside of the current function, or other low-level information about the computer may fail if that knowledge is applied to the stack frame for a managed function. Functions containing inline assembly code are generated as unmanaged functions, as if they were placed in a separate module that was compiled without **/clr**.
  - Inline assembly code in functions that pass copy-constructed function parameters is not supported.
- The vprintf Functions cannot be called from a program compiled with **/clr**.
- The naked __declspec modifier is ignored under /clr.
- The translator function set by _set_se_translator will affect only catches in unmanaged code. See Exception Handling under /clr for more information.
- The comparison of function pointers is not permitted under **/clr**.
- The use of functions that are not fully prototyped is not permitted under **/clr**.
- The following compiler options are not supported with **/clr**:
  - **/EHsc** and **/EHs** (**/clr** implies **/EHa** (see /EH (Exception Handling Model))
  - **/fp:strict** and **/fp:except** (see /fp (Specify Floating-Point Behavior).)/Zd
  - /Gm
  - /MT
  - /RTC
  - **/ZI**
- /Jis not supported with **/clr:safe** or **/clr:pure**.
- In Visual C++ 2005, the ATL and MFC libraries are not supported by pure mode compilation (**/clr:pure**). You can use /clr:pure with the Standard C++ Library and the CRT if you also compile with **/MD** or **/MDd**.
- When using **/Zi** with **/clr**, there are performance implications. For more information, see /Zi.
- Passing a wide character to a .NET Framework output routine without also specifying /Zc:wchar_t or without casting the character to **__wchar_t** will cause the output to appear as an **unsigned short int**. For example:

  ```
  Console::WriteLine(L' ')              // Will output 32.
  Console::WriteLine((__wchar_t)L' ')   // Will output a space.
  ```

- /GS is ignored when compiling with **/clr**, unless a function is under **#pragma** unmanaged or if the function must be compiled to native, in which case the compiler will generate warning C4793, which is off by default.
- See /ENTRY for function signature requirements of a managed application.
- Applications compiled with **/openmp** and **/clr** can only be run in a single appdomain process. See /openmp (Enable OpenMP 2.0 Support) for more information.
- Functions that take a variable number of arguments (varargs) will be generated as native functions. Any managed data types in the variable argument position will be marshaled to native types. Note that System.String types are actually

wide-character strings, but they are marshaled to single-byte character strings. So if a printf specifier is %S (wchar_t*), it will marshal to a %s string instead.

- When using the va_arg macro, you may get unexpected results when compiling with **/clr:pure**. For more information, see va_arg, va_end, va_start.
- You should not call, from managed code, any functions that walk the stack to get parameter information (function arguments); the P/Invoke layer causes that information to be further down the stack. For example, do not compile proxy/stub with /clr.
- Functions will be compiled to managed code whenever possible, but not all C++ constructs can be translated to managed code. This determination is made on a function-by-function basis. If any part of a function cannot be converted to managed code, the entire function will be converted to native code instead. The following cases prevent the compiler from generating managed code.
    - Compiler-generated thunks or helper functions. Native thunks are generated for any function call through a function pointer, including virtual function calls.
    - Functions that call `setjmp` or `longjmp`.
    - Functions that use certain intrinsic routines to directly manipulate machine resources. For example, the use of `__enable` and `__disable`, `_ReturnAddress` and `_AddressOfReturnAddress`, or multimedia intrinsics will all result in native code.
    - Functions that follow the `#pragma unmanaged` directive. (Note that the inverse, `#pragma managed`, is also supported.)
    - A function that contains references to aligned types, that is, types declared using `__declspec(align(...))`.
- You cannot use the Compiler COM Support classes with **/clr:pure** or **/clr:safe**.

See Also

Reference

/clr (Common Language Runtime Compilation)

/D (Preprocessor Definitions)

Defines a preprocessing symbol for your source file.

```
/Dname[= | # [{string | number}] ]
```

Remarks

You can use this symbol with **#if** or **#ifdef** to compile source conditionally. The symbol definition remains in effect until a redefinition is encountered in source or the symbol is undefined in source with the **#undef** directive.

/D has much the same effect as using the **#define** directive at the beginning of your source file. However, **/D** strips quotes on the command line and **#define** retains them.

By default, the value associated with a symbol will be 1. That is, **/DTEST** is equivalent to **/DTEST**=1. In the following example, the definition of **TEST** is shown to print 1.

Compiling with **/Dname=** causes the symbol to not have an associated value. While the symbol can still be used to conditionally compile code, the symbol will otherwise evaluate to nothing. For example, in the sample program, compiling with **/DTEST=** will cause a compiler error. This behavior is similar to using **#define** with or without a value.

The following command defines the symbol DEBUG in TEST.c:

```
CL /DDEBUG  TEST.C
```

The following command removes all occurrences of the keyword **__far** in TEST.c:

```
CL /D__far=  TEST.C
```

You cannot set the CL environment variable to a string that contains an equal sign. To use **/D** with the CL environment variable, you must specify a number sign instead of an equal sign:

```
SET CL=/DTEST#0
```

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Preprocessor** property page.
4. Modify the **Preprocessor Definitions** property.

To set this compiler option programmatically

- See PreprocessorDefinitions.

Example

```
// cpp_D_compiler_option.cpp
// compile with: /DTEST
#include <stdio.h>

int main( )
{
    #ifdef TEST
        printf_s("TEST defined %d\n", TEST);
    #else
        printf_s("TEST not defined\n");
    #endif
```

```
}
```

Output

```
TEST defined 1
```

See Also

Reference

Compiler Options
Setting Compiler Options
/U, /u (Undefine Symbols)
The #undef Directive
The #define Directive

/doc (Process Documentation Comments) (C/C++)

Causes the compiler to process documentation comments in source code files and to create an .xdc file for each source code file that has documentation comments.

```
/doc[name]
```

Arguments
name

The name of the .xdc file that the compiler will create. Only valid when one .cpp file is passed in the compilation.

Remarks

The .xdc files are processed into an .xml file with xdcmake.exe. For more information, see XDCMake Reference.

You can add documentation comments to your source code files. For more information, see Recommended Tags for Documentation Comments (Visual C++).

/doc is not compatible with **/clr:oldSyntax**. See /clr (Common Language Runtime Compilation) for more information.

To use the generated .xml file with IntelliSense, make the file name of the .xml file the same as the assembly that you want to support and put the .xml file is in the same directory as the assembly. When the assembly is referenced in the Visual Studio project, the .xml file is also found. For more information, see Using IntelliSense and Supplying XML Code Comments.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Expand the **Configuration Properties** node.
3. Expand the **C/C++** node.
4. Select the **Output Files** property page.
5. Modify the **Generate XML Documentation Files** property.

To set this linker option programmatically

- See GenerateXMLDocumentationFiles.

See Also
Reference
Compiler Options
Setting Compiler Options

/E (Preprocess to stdout)

Preprocesses C and C++ source files and copies the preprocessed files to the standard output device.

```
/E
```

Remarks

In this process, all preprocessor directives are carried out, macro expansions are performed, and comments are removed. To preserve comments in the preprocessed output, use the /C (Preserve Comments During Preprocessing) compiler option as well.

/E adds **#line** directives to the output at the beginning and end of each included file and around lines removed by preprocessor directives for conditional compilation. These directives renumber the lines of the preprocessed file. As a result, errors generated during later stages of processing refer to the line numbers of the original source file rather than lines in the preprocessed file.

The **/E** option suppresses compilation. You must resubmit the preprocessed file for compilation. **/E** also suppresses the output files from the **/FA**, **/Fa**, and **/Fm** options. For more information, see /FA, /Fa (Listing File) and /Fm (Name Mapfile).

To suppress **#line** directives, use the /EP (Preprocess to stdout Without #line Directives) option instead.

To send the preprocessed output to a file instead of to **stdout**, use the /P (Preprocess to a File) option instead.

To suppress **#line** directives and send the preprocessed output to a file, use **/P** and **/EP** together.

You cannot use precompiled headers with the **/E** option.

Note that when preprocessing to a separate file, spaces are not emitted after tokens. This can result in an illegal program or have unintended side effects. The following program compiles successfully:

```
#define m(x) x
m(int)main( )
{
   return 0;
}
```

However, if you compile with:

```
cl -E test.cpp > test2.cpp
```

`int main` in test2.cpp will incorrectly be `intmain`.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See GeneratePreprocessedFile.

Example

The following command line preprocesses `ADD.C`, preserves comments, adds **#line** directives, and displays the result on the standard output device:

```
CL /E /C ADD.C
```

See Also

Reference

Compiler Options

Setting Compiler Options

/EH (Exception Handling Model)

Specifies the model of exception handling to be used by the compiler and destroys C++ objects that will go out of scope as a result of the exception. If **/EH** is not specified, the compiler will catch structured and C++ exceptions, but will not destroy C++ objects that will go out of scope as a result of the exception.

```
/EH{s|a}[c][-]
```

Arguments

a

The exception-handling model that catches asynchronous (structured) and synchronous (C++) exceptions.

s

The exception-handling model that catches C++ exceptions only and tells the compiler to assume that **extern** C functions do throw an exception.

c

If used with **s** (**/EHsc**), catches C++ exceptions only and tells the compiler to assume that **extern** C functions never throw a C++ exception. **/EHca** is equivalent to **/EHa**.

Remarks

Use **/EHs** to specify the synchronous exception handling model (C++ exception handling without structured exception handling exceptions). If you use **/EHs**, then your catch clause will not catch asynchronous exceptions. Also, in Visual C++ 2005, all objects in scope when the asynchronous exception is generated will not be destroyed even if the asynchronous exception is handled. Under **/EHs**, **catch(...)** will only catch C++ exceptions. Access violations and System.Exception exceptions will not be caught.

Use **/EHa** to specify the asynchronous exception handling model (C++ exception handling with structured exception handling exceptions). **/EHa** may result in a less performant image because the compiler will not optimize a catch block as aggressively, even if the compiler does not see a throw.

Use **/EHa** if you want to catch an exception raised with something other than a throw. The following sample will generate an exception:

```
// compiler_options_EHA.cpp
// compile with: /EHa
#include <iostream>
#include <excpt.h>
using namespace std;

void fail() {   // generates SE and attempts to catch it using catch(...)
   try {
      int i = 0, j = 1;
      j /= i;   // This will throw a SE (divide by zero).
   }
   catch(...) {   // catch block will only be executed under /EHa
      cout<<"Caught an exception in catch(...)."<<endl;
   }
}

int main() {
   __try {
      fail();
   }

   // __except will only catch an exception here
   __except(EXCEPTION_EXECUTE_HANDLER) {
   // if the exception was not caught by the catch(...) inside fail()
      cout << "An exception was caught in __except." << endl;
   }
```

```
}
```

The **/EHc** option requires that **/EHs** or **/EHa** is specified. Using /clr (Common Language Runtime Compilation) implies **/EHa** (**/clr /EHa** is redundant). The compiler will generate an error if **/EHs[c]** is used after **/clr**. Optimizations will not affect this behavior. When an exception is caught, the compiler invokes the class destructor or destructors for the object or objects that are in the same scope as the exception. When an exception is not caught, those destructors are not run.

See _set_se_translator for exception handling restrictions under **/clr**.

The option can be cleared by the symbol **-**. For example, **/EHsc-** is interpreted as **/EHs /EHc-** and is equivalent to **/EHs**.

See Exception Handling: Default Synchronous Exception Model for more information.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Code Generation** property page.
4. Modify the **Enable C++ Exceptions** property.

Alternately, you can use the following procedure:

To set this compiler option in the Visual Studio development environment

1. Click the **C/C++** folder.
2. Click the **Code Generation** property page.
3. Set **Enable C++ Exceptions** to **No**.
4. Click the **Command Line** property page.
5. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See ExceptionHandling.

See Also
Reference
Compiler Options
Setting Compiler Options
Exception Specifications

/EP (Preprocess to stdout Without #line Directives)

Preprocesses C and C++ source files and copies the preprocessed files to the standard output device.

```
/EP
```

Remarks

In the process, all preprocessor directives are carried out, macro expansions are performed, and comments are removed. To preserve comments in the preprocessed output, use the /C (Preserve Comments During Preprocessing) option with **/EP**.

The **/EP** option suppresses compilation. You must resubmit the preprocessed file for compilation. **/EP** also suppresses the output files from the **/FA**, **/Fa**, and **/Fm** options. For more information, see /FA, /Fa (Listing File) and /Fm (Name Mapfile).

Errors generated during later stages of processing refer to the line numbers of the preprocessed file rather than the original source file. If you want line numbers to refer to the original source file, use /E (Preprocess to stdout) instead. The **/E** option adds **#line** directives to the output for this purpose.

To send the preprocessed output, with **#line** directives, to a file, use the /P (Preprocess to a File) option instead.

To send the preprocessed output to stdout, with **#line** directives, use **/P** and **/EP** together.

You cannot use precompiled headers with the **/EP** option.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Preprocessor** property page.
4. Modify the **Generate Preprocessed File** property.

To set this compiler option programmatically

- See GeneratePreprocessedFile.

Example

The following command line preprocesses file `ADD.C`, preserves comments, and displays the result on the standard output device:

```
CL /EP /C ADD.C
```

See Also

Reference

Compiler Options

Setting Compiler Options

/errorReport (Report Internal Compiler Errors)

Allows you to provide internal compiler error (ICE) information directly to the Visual C++ team.

```
/errorReport:[ none | prompt | queue | send ]
```

Arguments

none

Reports about internal compiler errors will not be collected or sent to Microsoft.

prompt

Prompts you to send a report when you receive an internal compiler error. **prompt** is the default when compiling an application in the development environment.

queue

Queues the error report. When you log in with administrator privileges, a pop-up window is displayed and allows you to report any failures since the last time you were logged in (you will not be prompted to send reports for failures more than once every three days). **queue** is the default when compiling an application at the command line.

send

Automatically sends reports of internal compiler errors to Microsoft. To enable this option, you must first agree to Microsoft's data collection policy. The first time you specify **/errorReport:send** on a computer, a compiler message will refer you to a website that contains Microsoft's data collection policy.

Remarks

An internal compiler error (ICE) results when the compiler cannot process a source code file. When an ICE occurs, the compiler does not produce an output file or any useful diagnostic that you can use to fix your code.

In previous releases, when you got an ICE, you were encouraged to call Microsoft technical support to report the problem. With **/errorReport**, you can provide ICE information directly to the Visual C++ team. Your error reports can help improve future compiler releases.

A user's ability to send reports depends on machine and user policy permissions.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For more information, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Advanced** property page.
4. Modify the **Error Reporting** property.

To set this compiler option programmatically

- See ErrorReporting.

See Also

Reference

Compiler Options

Setting Compiler Options

/F (Set Stack Size)

Sets the program stack size in bytes.

```
/F[ ]number
```

Arguments

number

The stack size in bytes.

Remarks

Without this option the stack size defaults to 1 MB. The *number* argument can be in decimal or C-language notation. The argument can range from 1 to the maximum stack size accepted by the linker. The linker rounds up the specified value to the nearest 4 bytes. The space between **/F** and *number* is optional.

You may need to increase the stack size if your program gets stack-overflow messages.

You can also set the stack size by:

- Using the **/STACK** linker option. For more information, see /STACK.
- Using EDITBIN on the .exe file. For more information, see EDITBIN Reference.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

Compiler Options

Setting Compiler Options

Output-File (/F) Options

The output-file options create or rename output files. They affect all C or C++ source files specified in the CL environment variable, on the command line, or in any command file.

- /FA, /Fa (Listing File)
- Specifying the Pathname
- /Fd (Name PDB File)
- /Fe (Name EXE File)
- /FI (Name Forced Include File)
- /Fm (Name Mapfile)
- /Fo (Name Object File)
- /Fp (Name .pch File)
- /FR, /Fr (Create .sbr File)
- /FU (Name Forced #using File)
- /Fx (Merge Injected Code)

See Also
Reference
Compiler Options
Setting Compiler Options

/FA, /Fa (Listing File)

Creates a listing file containing assembly code.

```
/FA[c|s|u]
/Fapathname
```

Remarks

The arguments control the generation of source code and machine code and the extension of the listing file.

The following table describes the various values to **/FA**. It is possible to specify more than one value to **/FA**. For example, you can specify **/FAsu**.

Option	Listing contents and file extension
/FA	Assembly code; .asm
/FAc	Machine and assembly code; .cod
/FAs	Source and assembly code; .asm If **/FAcs** is specified, the file extension will be .cod
/FAu	Causes the output file to be created in UTF-8 format, with a byte order marker. By default, the file encoding is ANSI, but use **/FAu** if you want a listing file that displays correctly on any system, or if you are using Unicode source code files as input to the compiler. If **/FAsu** is specified, and if a source code file uses Unicode encoding other than UTF-8, then the code lines in the .asm file may not display correctly.

By default, the listing file gets the same base name as the source file. You can change the name of the listing file and the directory where it is created using the **/Fa** option.

/Fa usage	Result
/Fa	One *source_file*.asm is created for each source code file in the compilation.
/Fa *filename*	*filename*.asm is placed in the current directory. Only valid when compiling a single source code file.
/Fa *filename.extension*	*filename.extension* is placed in the current directory. Only valid when compiling a single source code file.
/Fa *directory*	One *source_file*.asm is created and placed in the specified *directory* for each source code file in the compilation. Note the required trailing backslash. Only paths on the current disk are allowed.
/Fa *directory\filename*	*filename*.asm is placed in the specified *directory*. Only valid when compiling a single source code file.
/Fa *directory\filename.extension*	*filename.extension* is placed in the specified *directory*. Only valid when compiling a single source code file.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **C/C++** folder.
3. Click the **Output Files** property page.
4. Modify the **ASM List Location** (**/Fa**) or **Assembler Output** (**/FA**) property (**/FAu** must be specified in the **Command Line** property page, **Additional Options** box..

To set this compiler option programmatically

- See AssemblerListingLocation or AssemblerOutput. To specify **/FAu**, see AdditionalOptions.

Example

The following command line produces a combined source and machine-code listing called HELLO.cod:

```
CL /FAcs HELLO.CPP
```

See Also

Reference

Output-File (/F) Options
Compiler Options
Setting Compiler Options
Specifying the Pathname

Specifying the Pathname

Each output-file option accepts a *pathname* argument that can specify a location and a name for the output file. The argument can include a drive name, directory, and file name. No space is allowed between the option and the argument.

If *pathname* includes a file name without an extension, the compiler gives the output a default extension. If *pathname* includes a directory but no file name, the compiler creates a file with a default name in the specified directory. The default name is based on the base name of the source file and a default extension based on the type of the output file. If you leave off *pathname* entirely, the compiler creates a file with a default name in a default directory.

Alternatively, the *pathname* argument can be a device name (AUX, CON, PRN, or NUL) rather than a file name. Do not use a space between the option and the device name or a colon as part of the device name.

Device name	**Represents**
AUX	Auxiliary device
CON	Console
PRN	Printer
NUL	Null device (no file created)

Example

The following command line sends a mapfile to the printer:

```
CL /FmPRN HELLO.CPP
```

See Also

Reference

Output-File (/F) Options
Compiler Options
Setting Compiler Options

/FD (IDE Minimal Rebuild)

/FD is not exposed to users except in the Command Line property page of a C++ project's **Property Pages** dialog box, if and only if /Gm (Enable Minimal Rebuild) is not also selected. **/FD** has no effect other than from the development environment. **/FD** is not exposed in the output of **cl /?**.

If you do not enable **/Gm** in the development environment, **/FD** will be used. **/FD** ensures that the .idb file has sufficient dependency information. **/FD** is only used by the development environment, and it should not be used from the command line or a build script.

See Also
Reference
Output-File (/F) Options
Compiler Options
Setting Compiler Options

/Fd (Program Database File Name)

Specifies a file name for the program database (PDB) file created by /Z7, /Zi, /ZI (Debug Information Format).

```
/Fdpathname
```

Remarks

Without **/Fd**, the PDB file name defaults to VC*x*0.pdb., where *x* is the major version of Visual C++ in use.

If you specify a path name that does not include a file name (the path ends in backslash), the compiler creates a .pdb file named VC*x*0.pdb. in the specified directory.

If you specify a file name that does not include an extension, the compiler uses .pdb as the extension.

This option also names the state (.idb) file used for minimal rebuild and incremental compilation.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Output Files** property page.
4. Modify the **Program Database File Name** property.

To set this compiler option programmatically

- See ProgramDataBaseFileName.

Example

This command line creates a .pdb file named PROG.pdb and an .idb file named PROG.idb:

```
CL /DDEBUG /Zi /FdPROG.PDB PROG.CPP
```

See Also

Reference

Output-File (/F) Options
Compiler Options
Setting Compiler Options
Specifying the Pathname

/Fe (Name EXE File)

Specifies a name and a directory for the .exe file or DLL created by the compiler.

```
/Fepathname
```

Remarks

Without this option, the compiler gives the file a default name using the base name of the first source or object file specified on the command line and the extension .exe or .dll.

If you specify the/c (Compile Without Linking), to compile without linking, **/Fe** has no effect.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **Linker** folder.
3. Click the **General** property page.
4. Modify the **Output File** property.

To set this compiler option programmatically

- See OutputFile.

Example

The following command line compiles and links all C source files in the current directory. The resulting executable file is named PROCESS.exe and is created in the directory C:\BIN.

```
CL /FeC:\BIN\PROCESS *.C
```

The following command line creates an executable file in `C:\BIN` with the same base name as the first source or object file:

```
CL /FeC:\BIN\ *.C
```

See Also

Reference

Output-File (/F) Options
Compiler Options
Setting Compiler Options
Specifying the Pathname

/FI (Name Forced Include File)

Causes the preprocessor to process the specified header file.

```
/FI[ ]pathname
```

Remarks

This option has the same effect as specifying the file with double quotation marks in an **#include** directive on the first line of every source file specified on the command line, in the CL environment variable, or in a command file. If you use multiple **/FI** options, files are included in the order they are processed by CL.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Advanced** property page.
4. Modify the **Force Includes** property.

To set this compiler option programmatically

- See ForcedIncludeFiles.

See Also

Reference

Output-File (/F) Options
Compiler Options
Setting Compiler Options
Specifying the Pathname

/Fm (Name Mapfile)

Tells the linker to produce a mapfile containing a list of segments in the order in which they appear in the corresponding .exe file or DLL.

```
/Fmpathname
```

Remarks

By default, the mapfile is given the base name of the corresponding C or C++ source file with a .MAP extension.

Specifying **/Fm** has the same effect as if you had specified the /MAP (Generate Mapfile) linker option.

If you specify /c (Compile Without Linking) to suppress linking, **/Fm** has no effect.

Global symbols in a mapfile usually have one or more leading underscores because the compiler adds a leading underscore to variable names. Many global symbols that appear in the mapfile are used internally by the compiler and the standard libraries.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

Output-File (/F) Options
Compiler Options
Setting Compiler Options
Specifying the Pathname

/Fo (Object File Name)

Specifies an object (.obj) file name or directory to be used instead of the default.

```
/Fopathname
```

Remarks

If you do not use this option, the object file uses the base name of the source file and the .obj extension. You can use any name and extension you want, but the recommended convention is to use .obj.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Output Files** property page.
4. Modify the **Object File Name** property. In the development environment, the object file must have an extension of .obj.

To set this compiler option programmatically

- See ObjectFile.

Example

The following command line creates an object file named THIS.obj in an existing directory, \OBJECT, on drive B.

```
CL /FoB:\OBJECT\ THIS.C
```

See Also

Reference

Output-File (/F) Options
Compiler Options
Setting Compiler Options
Specifying the Pathname

/Fp (Name .pch File)

Provides a path name for a precompiled header instead of using the default path name.

```
/Fppathname
```

Remarks

Use this option with /Yc (Create Precompiled Header File) or /Yu (Use Precompiled Header File) to provide a path name for a precompiled header instead of using the default path name. You can also use **/Fp** with **/Yc** to specify the use of a precompiled header file that differs from the **/Yc** *filename* argument and from the base name of the source file.

If you do not specify an extension as part of the path name, an extension of .pch is assumed. If you specify a directory without a file name, the default file name is VC*x*0.pch., where *x* is the major version of Visual C++ in use.

You can also use the **/Fp** option with **/Yu**.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Precompiled Headers** property page.
4. Modify the **Precompiled Header File** property.

To set this compiler option programmatically

- See PrecompiledHeaderFile.

Example

If you want to create a precompiled header file for a debugging version of your program and you are compiling both header files and source code, you can specify a command such as:

```
CL /DDEBUG /Zi /Yc /FpDPROG.PCH PROG.CPP
```

The following command specifies the use of a precompiled header file named MYPCH.pch. The compiler assumes that the source code in PROG.cpp has been precompiled through MYAPP.h and that the precompiled code resides in MYPCH.pch. It uses the content of MYPCH.pch and compiles the rest of PROG.cpp to create an .obj file. The output of this example is a file named PROG.exe.

```
CL /YuMYAPP.H /FpMYPCH.PCH PROG.CPP
```

See Also

Reference

Output-File (/F) Options
Compiler Options
Setting Compiler Options
Specifying the Pathname

/FR, /Fr (Create .sbr File)

Creates .sbr files.

```
/FR[pathname[\filename]]
/Fr[pathname[\filename]]
```

Remarks

During the build process, the Microsoft Browse Information File Maintenance Utility (BSCMAKE) uses these files to create a .BSC file, which is used to display browse information.

/FR creates an .sbr file with complete symbolic information.

/Fr creates an .sbr file without information on local variables.

If you do not specify *filename*, the .sbr file gets the same base name as the source file.

/Fr is deprecated in Visual C++ 2005; use **/FR** instead. For more information, see Deprecated Compiler Options.

Note
Do not change the .sbr extension. BSCMAKE requires the intermediary files to have that extension.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Browse Information** property page.
4. Modify the **Browse File** or **Enable Browse Information** property.
5. Make sure that the **Build Browser Information** property on the General Property Page (Project) is set to **Yes**.

To set this compiler option programmatically

- See BrowseInformation and BrowseInformationFile.

See Also

Reference

Output-File (/F) Options
Compiler Options
Setting Compiler Options
Specifying the Pathname

/FU (Name Forced #using File)

Gives you a compiler option alternative to passing a file name to The #using Directive in source code.

```
/FUfile
```

Arguments

file

Specifies the metadata file to reference in this compilation.

Remarks

Specify one file per compiler option.

If you are referencing metadata to use the Friend Assemblies feature, you cannot use **/FU**; you must reference the metadata in code with **#using** (and the **[as_friend]** attribute).

See/clr (Common Language Runtime Compilation) for information on how to create an assembly or module for the common language runtime.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Advanced** property page.
4. Modify the **Force #using** property.

To set this compiler option programmatically

- See ForcedUsingFiles.

See Also

Reference

Output-File (/F) Options
Compiler Options
Setting Compiler Options

/Fx (Merge Injected Code)

Produces a copy of each source file with injected code merged into the source.

```
/Fx
```

Remarks

To distinguish a merged source file from an original source file, **/Fx** adds an .mrg extension between the file name and file extension. For example, a file named MyCode.cpp containing attributed code and built with **/Fx** creates a file named MyCode.mrg.cpp containing the following code:

```
//+++ Start Injected Code
[no_injected_text(true)];      // Suppress injected text, it has
                               // already been injected
#pragma warning(disable: 4543) // Suppress warnings about skipping
                               // injected text
#pragma warning(disable: 4199) // Suppress warnings from attribute
                               // providers
//--- End Injected Code
```

In an .mrg file, code that was injected because of an attribute will be delimited as follows:

```
//+++ Start Injected Code
...
//--- End Injected Code
```

The no_injected_text attribute is embedded in an .mrg file, which allows for the compilation of the .mrg file without text being reinjected.

You should be aware that the .mrg source file is intended to be a representation of the source code injected by the compiler. The .mrg file may not compile or run exactly as the original source file.

Macros are not expanded in the .mrg file.

If your program includes a header file that uses injected code, **/Fx** generates an .mrg.h file for that header. **/Fx** does not merge include files that do not use injected code.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Output Files** property page.
4. Modify the **Expand Attributed Source** property.

To set this compiler option programmatically

- See ExpandAttributedSource.

See Also

Reference

Output-File (/F) Options
Compiler Options
Setting Compiler Options

/favor (Optimize for 64)

/favor: *option* is available only in the x64 compiler and produces the code that is optimized for a specific x64 architecture or for the specifics of micro-architectures in both the AMD64 and Extended Memory 64 Technology (EM64T) architectures.

```
/favor:blend
/favor:AMD64
/favor:EM64T
```

Remarks

/favor:blend produces the code that is optimized for the specifics of micro-architectures in both the AMD64 and EM64T architectures. While **/favor:blend** may not give the best performance possible on a specific processor, it is designed to give the best performance across a broad range of x64 processors. **/favor:blend** is in effect by default.

/favor:AMD64 optimizes the generated code for the AMD Opteron, and Athlon processors supporting 64-bit extensions, typically resulting in better performance for that platform. The resulting code can run on all x64 compatible platforms. Code generated with **/favor:AMD64** could result in worse performance on Intel processors supporting EM64T.

/favor:EM64T is available only in the x64 compiler and optimizes the generated code for Intel processors supporting EM64T, typically resulting in better performance for that platform. The resulting code can run on any x64 platform. Code generated with **/favor:EM64T** could result in worse performance on AMD Opteron, and Athlon processors supporting 64-bit extensions.

See x64 Software Conventions for information on programming for the x64 architecture.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

Compiler Options

Setting Compiler Options

/FC (Full Path of Source Code File in Diagnostics)

Causes the compiler to display the full path of source code files passed to the compiler in diagnostics.

```
/FC
```

Remarks

Consider the following code sample:

```
// compiler_option_FC.cpp
int main( ) {
   int i   // C2143
}
```

Without **/FC**, the diagnostic text would look similar to this diagnostic text:

- compiler_option_FC.cpp(5) : error C2143: syntax error : missing ';' before '}'

With **/FC**, the diagnostic text would look similar to this diagnostic text:

- c:\test\compiler_option_FC.cpp(5) : error C2143: syntax error : missing ';' before '}'

/FC is also needed if you want to see the full path of a file name when using the __FILE__ macro. See Predefined Macros for more information on __FILE__.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Expand the **Configuration Properties** node.
3. Expand the **C/C++** node.
4. Select the **Advanced** property page.
5. Modify the **Use Full Paths** property.

To set this linker option programmatically

- See UseFullPaths.

See Also
Reference
Compiler Options
Setting Compiler Options

/fp (Specify Floating-Point Behavior)

Specifies floating-point behavior in a source code file.

```
/fp:[precise | except[-] | fast | strict ]
```

Flags

precise

The default.

Improves the consistency of floating-point tests for equality and inequality by disabling optimizations that could change the precision of floating-point calculations, which is required for strict ANSI conformance. By default, the compiler uses the coprocessor's 80-bit registers to hold the intermediate results of floating-point calculations. This increases program speed and decreases program size. Because the calculation involves floating-point data types that are represented in memory by less than 80 bits, however, carrying the extra bits of precision (80 bits minus the number of bits in a smaller floating-point type) through a lengthy calculation can produce inconsistent results.

With **/fp:precise** on x86 processors, the compiler will perform rounding on variables of type float to the proper precision for assignments and casts and when passing parameters to a function. This rounding guarantees that the data does not retain any significance greater than the capacity of its type. A program compiled with **/fp:precise** can be slower and larger than one compiled without **/fp:precise**. **/fp:precise** disables intrinsics; the standard run-time library routines are used instead. For more information, see /Oi (Generate Intrinsic Functions).

The following floating-point behavior is enabled with **/fp:precise**:

- Contractions, or replacing multiple operations with one composite operation with only a single rounding at the end, is enabled with **/fp:precise**.
- Expression optimizations that are invalid for special values (NaN, +infinity, -infinity, +0, -0) will not be allowed. The optimizations *x-x => 0*, *x*0 => 0*, *x-0 => x*, *x+0 => x*, and *0-x => -x* are all invalid for various reasons (see IEEE 754 and the C99 standard).
- The compiler will properly handle comparisons involving NaN. For example, *x != x* evaluates to **true** if `x` is NaN and ordered comparisons involving NaN raise an exception.
- Expression evaluation will follow the C99 FLT_EVAL_METHOD=2, with one exception. When programming for x86 processors, because the FPU is set to 53-bit precision, this will be considered long double precision.
- Multiplication by exactly 1.0 transformed into a use of the other factor. *x*y*1.0* is transformed into *x*y*. Similarly, *x*1.0*y* is transformed into *x*y*.
- Division by exactly 1.0 is transformed into a use of the dividend. *x*y/1.0* is transformed into *x*y*. Similarly, *x/1.0*y* is transformed into *x*y*.

Using **/fp:precise** with fenv_access ON disables some optimizations such as compile-time evaluations of floating point expressions. For example, if you change rounding mode with _control87, _controlfp, __control87_2, and the compiler performs a floating-point calculation, the rounding mode you specified will not be in effect unless **fenv_access** is ON.

/fp:precise replaces the **/Op** compiler option.

fast

Creates the fastest code in the majority of cases. **/fp:fast** cannot be used with **/fp:strict** or **/fp:precise**, the last option specified on the command line will be used. **/fp:fast** and **/fp:except** will generate a compiler error.

Selecting /Za, /Ze (Disable Language Extensions) (ANSI compatibility) and **/fp:fast** may result in unexpected behavior. For example, single-precision floating-point operations may not be rounded to single precision.

except[-]

Reliable floating-point exception model. Exceptions will be raised immediately after they are triggered. This option is off by default. Appending a minus sign to the option explicitly disables it.

strict

The strictest floating-point model. **/fp:strict** causes fp_contract to be OFF and fenv_access to be ON. **/fp:except** is implied and can be disabled by explicitly specifying **/fp:except-**. When used with **/fp:except-**, **/fp:strict** enforces strict floating-point semantics but without respect for exceptional events.

Remarks

Multiple **/fp** options can be specified in the same compilation.

To control floating-point behavior by function, see the float_control pragma.

Most of the floating-point optimizations related to **/fp:strict**, **/fp:except** (and its corresponding pragmas), and **fp_contract** pragma are machine dependent. **/fp:strict** and **/fp:except** are not compatible with **/clr**.

/fp:precise should address most of an application's floating-point requirements. If necessary, you can use **/fp:except** and **/fp:strict**, but there may be some decrease in performance. If performance is most important, you may want to use **/fp:fast**.

/fp:strict, **/fp:fast**, and **/fp:precise** are precision (correctness) modes. Only one can be in effect at a time. If **/fp:strict** and **/fp:precise** are specified, the compiler use the one that it processes last. You cannot specify both **/fp:strict** and **/fp:fast**.

For more information, see http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/floapoint.asp.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Expand the **Configuration Properties** node.
3. Expand the **C/C++** node.
4. Select the **Code Generation** property page.
5. Modify the **Floating Point Model** property.

To set this compiler option programmatically

- See floatingPointModel.

See Also
Reference
Compiler Options
Setting Compiler Options

/G (Optimize for Processor) Options

Tells the compiler to optimize code generation for the specified processor.

```
/G{1|2}
```

Arguments

/G1

Optimize for Itanium processor. Only available in the Itanium cross compiler or Itanium native compiler.

/G2

Optimize for Itanium2 processor (default between **/G1** and **/G2**) Only available in the Itanium cross compiler or Itanium native compiler.

Remarks

/G is available in all editions of Visual C++, but the compiler can perform more optimizations when **/G** is used with one of the /O compiler options. However, **/O** is not available in the Visual C++ standard edition.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

Compiler Options

Setting Compiler Options

/GA (Optimize for Windows Application)

Results in more efficient code for an .exe file for accessing thread-local storage (TLS) variables.

```
/GA
```

Remarks

/GA speeds access to data declared with __declspec(thread) in a Windows-based program. When this option is set, the __tls_index macro is assumed to be 0.

Using **/GA** for a DLL can result in bad code generation.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also
Reference
Compiler Options
Setting Compiler Options

/Gd, /Gr, /Gz (Calling Convention)

These options determine the order in which function arguments are pushed onto the stack; whether the caller function or called function removes the arguments from the stack at the end of the call; and the name-decorating convention that the compiler uses to identify individual functions.

```
/Gd
/Gr
/Gz
```

Remarks

Note
These compiler options are only available in the compiler targeting x86; these compiler options are not available in the compilers targeting x64 or Itanium.

/Gd, the default setting, specifies the __cdecl calling convention for all functions except C++ member functions and functions marked __stdcall or __fastcall.

/Gr specifies the **__fastcall** calling convention for all functions except C++ member sfunctions and functions marked **__cdecl** or **__stdcall**. All **__fastcall** functions must have prototypes.

/Gz specifies the **__stdcall** calling convention for all functions except C++ member functions and functions marked **__cdecl** or **__fastcall**. All **__stdcall** functions must have prototypes.

Functions that take a variable number of arguments must be marked **__cdecl**.

/Gd, **/Gr**, and **/Gz** are not compatible with /clr:safe or **/clr:pure**.

Note
For x86 processors, by default, C++ member functions use __thiscall.

For all processors, a member function that is explicitly marked as **__cdecl**, **__fastcall**, or **__stdcall** uses the specified calling convention. A member function that takes a variable number of arguments always uses the **__cdecl** calling convention.

__cdecl Specifics

For C, the **__cdecl** naming convention uses the function name preceded by an underscore (_); no case translation is performed. Unless declared as **extern** "C", C++ functions use a different name-decorating scheme. For more information, see Decorated Names.

__fastcall Specifics

Some of a **__fastcall** function's arguments are passed in registers (for x86 processors, ECX and EDX), and the rest are pushed onto the stack from right to left. The called routine pops these arguments from the stack before it returns. Typically, **/Gr** decreases execution time.

Note
Be careful when using the **__fastcall** calling convention for any function written in inline assembly language. Your use of registers could conflict with the compiler's use.

For C, the **__fastcall** naming convention uses the function name preceded by an at sign (**@**) followed by the size of the function's arguments in bytes. No case translation is done. The compiler uses the following template for the naming convention:

```
@function_name@number
```

When using the **__fastcall** naming convention, use the standard include files. Otherwise you will get unresolved external

references.

__stdcall Specifics

A **__stdcall** function's arguments are pushed onto the stack from right to left, and the called function pops these arguments from the stack before it returns.

For C, the **__stdcall** naming convention uses the function name preceded by an underscore (_) and followed by an at sign (@) and the size of the function's arguments in bytes. No case translation is performed. The compiler uses the following template for the naming convention:

```
_functionname@number
```

x86 Specific

This option has no effect on the name decoration of C++ methods and functions. Unless declared as **extern** "C", C++ methods and functions use a different name-decorating scheme. For more information, see Decorated Names.

For more information about calling conventions, see Calling Conventions.

END x86 Specific

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Advanced** property page.
4. Modify the **Calling Convention** property.

To set this compiler option programmatically

- See CallingConvention.

See Also
Reference
Compiler Options
Setting Compiler Options

/Ge (Enable Stack Probes)

Activates stack probes for every function call that requires storage for local variables.

```
/Ge
```

Remarks

This mechanism is useful if you rewrite the functionality of the stack probe. It is recommended that you use /Gh (Enable _penter Hook Function) instead of rewriting the stack probe.

/Gs (Control Stack Checking Calls) has the same effect.

/Ge is deprecated in Visual C++ 2005; the compiler will generate stack checking. For more information, see Deprecated Compiler Options.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also
Reference
Compiler Options
Setting Compiler Options

/GF (Eliminate Duplicate Strings)

Enables the compiler to create a single copy of identical strings in the program image and in memory during execution, resulting in smaller programs, an optimization called string pooling.

```
/GF
```

Remarks

/GF pools strings as read-only.

If you use **/GF**, the operating system does not swap the string portion of memory and can read the strings back from the image file. If you try to modify strings under **/GF**, an application error occurs.

String pooling allows what were intended as multiple pointers to multiple buffers to be as multiple pointers to a single buffer. In the following code, `s` and `t` are initialized with the same string. String pooling causes them to point to the same memory:

```
char *s = "This is a character buffer";
char *t = "This is a character buffer";
```

Note

The/Z7, /Zi, /ZI (Debug Information Format) option, used for Edit and Continue, automatically sets the **/GF** option.

/GF is in effect when/O1, /O2 (Minimize Size, Maximize Speed) or **/O2** is used.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Code Generation** property page.
4. Modify the **Enable String Pooling** property.

To set this compiler option programmatically

- See StringPooling.

See Also
Reference
Compiler Options
Setting Compiler Options

/GH (Enable _pexit Hook Function)

Calls the **_pexit** function at the end of every method or function.

```
/GH
```

Remarks

The **_pexit** function is not part of any library and it is up to you to provide a definition for **_pexit**.

Unless you plan to explicitly call **_pexit**, you do not need to provide a prototype. The function must appear as if it had the following prototype, and it must push the content of all registers on entry and pop the unchanged content on exit:

```
void __declspec(naked) _cdecl _pexit( void );
```

_pexit is similar to **_penter**; see /Gh (Enable _penter Hook Function) for an example of how to write a **_pexit** function.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

Compiler Options

Setting Compiler Options

/Gh (Enable _penter Hook Function)

Causes a call to the **_penter** function at the start of every method or function.

```
/Gh
```

Remarks

The **_penter** function is not part of any library and it is up to you to provide a definition for **_penter**.

Unless you plan to explicitly call **_penter**, you do not need to provide a prototype. The function must appear as if it had the following prototype, and it must push the content of all registers on entry and pop the unchanged content on exit:

```
void __declspec(naked) _cdecl _penter( void );
```

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

Example

The following code, when compiled with **/Gh**, shows how **_penter** is called twice; once when entering function `main` and once when entering function `x`.

```
// Gh_compiler_option.cpp
// compile with: /Gh
// processor: x86
#include <stdio.h>
void x() {}

int main() {
   x();
}

extern "C" void __declspec(naked) _cdecl _penter( void ) {
   _asm {
      push eax
      push ebx
      push ecx
      push edx
      push ebp
      push edi
      push esi
    }

   printf_s("\nIn a function!");

   _asm {
      pop esi
      pop edi
      pop ebp
      pop edx
      pop ecx
```

```
        pop ebx
        pop eax
        ret
      }
  }
```

Sample Output

```
In a function!
In a function!
```

See Also

Reference

Compiler Options

Setting Compiler Options

/GL (Whole Program Optimization)

Enables whole program optimization.

```
/GL[-]
```

Remarks

Whole program optimization allows the compiler to perform optimizations with information on all modules in the program. Without whole program optimization, optimizations are performed on a per module (compiland) basis.

Whole program optimization is off by default and must be explicitly enabled. However, it is also possible to explicitly disable it with **/GL-**.

With information on all modules, the compiler can:

- Optimize the use of registers across function boundaries.
- Do a better job of tracking modifications to global data, allowing a reduction in the number of loads and stores.
- Do a better job of tracking the possible set of items modified by a pointer dereference, reducing the numbers of loads and stores.
- Inline a function in a module even when the function is defined in another module.

.obj files produced with **/GL** will not be available to such linker utilities as EDITBIN Reference and DUMPBIN Reference.

If you compile your program with **/GL** and /c (Compile Without Linking), you should use the /LTCG linker option to create the output file.

/Z7, /Zi, /ZI (Debug Information Format) cannot be used with **/GL**

The format of files produced with **/GL** in the current version may not be readable by subsequent versions of Visual C++. You should not ship a .lib file comprised of .obj files that were produced with **/GL** unless you are willing to ship copies of the .lib file for all versions of Visual C++ you expect your users to use, now and in the future.

.obj files produced with **/GL** and precompiled header files should not be used to build a .lib file unless the .lib file will be linked on the same machine that produced the **/GL** .obj file. Information from the .obj file's precompiled header file will be needed at link time.

For more information on the optimizations available with and the limitations of whole program optimization, see/LTCG (Link-time Code Generation). **/GL** also makes profile guided optimization available; see /LTCG. When compiling for profile guided optimizations and if you want function ordering from your profile guided optimizations, you must compile with /Gy (Enable Function-Level Linking) or a compiler option that implies /Gy.

To set this linker option in the Visual Studio development environment

- See /LTCG (Link-time Code Generation) for information on how to specify **/GL** in the development environment.

To set this linker option programmatically

- See WholeProgramOptimization.

See Also
Reference
Compiler Options
Setting Compiler Options

/Gm (Enable Minimal Rebuild)

Enables minimal rebuild, which determines whether C++ source files that include changed C++ class definitions (stored in header (.h) files) need to be recompiled.

```
/Gm
```

Remarks

The compiler stores dependency information between source files and class definitions in the project's .idb file during the first compile. (Dependency information tells which source file is dependent on which class definition, and which .h file the definition is located in.) Subsequent compiles use the information stored in the .idb file to determine whether a source file needs to be compiled, even if it includes a modified .h file.

Note
Minimal rebuild relies on class definitions not changing between include files. Class definitions must be global for a project (there should be only one definition of a given class), because the dependency information in the .idb file is created for the entire project. If you have more than one definition for a class in your project, disable minimal rebuild.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Code Generation** property page.
4. Modify the **Enable Minimal Rebuild** property.

To set this compiler option programmatically

- See MinimalRebuild.

See Also

Reference

Compiler Options

Setting Compiler Options

/GR (Enable Run-Time Type Information)

Adds code to check object types at run time.

```
/GR[-]
```

Remarks

When **/GR** is on, the compiler defines the **_CPPRTTI** preprocessor macro. In Visual C++ 2005, **/GR** is on by default. **/GR-** disables run-time type information.

Use **/GR** if your code uses dynamic_cast Operator or typeid. **/GR** does, however, cause the .rdata sections of your image to increase in size. If your code does not use **dynamic_cast** or **typeid**, using **/GR-** may produce a smaller image.

For more information on run-time type checking, see Run-Time Type Information in the *C++ Language Reference*.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Language** property page.
4. Modify the **Enable Run-Time Type Info** property.

To set this compiler option programmatically

- See RuntimeTypeInfo.

See Also
Reference
Compiler Options
Setting Compiler Options

/GS (Buffer Security Check)

Detects some buffer overruns that overwrite the return address, a common technique for exploiting code that does not enforce buffer size restrictions. This is achieved by injecting security checks into the compiled code.

```
/GS[-]
```

Remarks

/GS is on by default. Use **/GS**- if you expect your application to have no security exposure.

For additional information on **/GS** see Compiler Security Checks In Depth.

The compiler injects checks in functions with local string buffers or, on x86, functions with exception handling. A string buffer is defined as an array whose element size is one or two bytes, and where the size of the whole array is at least five bytes, or, any buffer allocated with _alloca.

On all platforms, the compiler injects a cookie to protect the function's return address if the function has local string buffers. This cookie is checked on function exit, and during frame unwinding on 64-bit operating systems or on x86 for functions with some form of exception handling. On x86, the compiler also injects a cookie to protect the address of the function's exception handler. This cookie is checked during frame unwinding.

/GS primarily attempts to detect direct buffer overruns into the return address. Buffer overruns are more easily exploited on machines with calling conventions that store the return address of function calls on the stack. For example, x86 uses calling conventions that store the return address of function calls on the stack.

On functions that the compiler thinks might be subject to buffer overrun problems, the compiler will allocate space on the stack before the return address. On function entry, the allocated space is loaded with a security cookie that is computed once at module load. Then, on function exit, a helper function is called to make sure the cookie's value is still the same.

If the value is not the same, an overwrite of the stack has potentially occurred and the process simply terminates. Prior to Visual C++ 2005 a dialog box displayed reporting the stack overwrite.

/GS also protects against vulnerable parameters passed into a function. A vulnerable parameter is a pointer, C++ reference, or a C-structure (C++ POD type) that contains a pointer, string buffer, or C++ reference.

A vulnerable parameter is allocated before the cookie and local variables. A buffer overrun can overwrite these parameters. Code in the function that uses these parameters can lead to an attack before the function returns, thus avoiding the security check. To minimize this danger, the compiler will make a copy of the vulnerable parameters during the function prolog and put them below the storage area for any buffers.

The compiler does not provide security protection for vulnerable parameters in the following situations:

- Functions that do not contain a buffer.
- If optimizations (/O Options (Optimize Code)) are not enabled.
- Functions with a variable argument list (...).
- Functions marked with naked (C++).
- Functions containing inline assembly code in the first statement.
- If a parameter is used only in ways that are less likely to be exploitable in the event of a buffer overrun.

/GS requires initialization of the security cookie. This cookie must be initialized before any function using the cookie runs. The security cookie must be initialized upon entry to an EXE or DLL. This is done automatically when using the default CRT entry points (mainCRTStartup, wmainCRTStartup, WinMainCRTStartup, wWinMainCRTStartup, or _DllMainCRTStartup), but must be done manually, via a call to __security_init_cookie, if you use an alternate entry point.

/GS is supported for managed functions when compiling with /clr (Common Language Runtime Compilation).

/GS does not protect against all buffer overrun security attacks. For example, if you have a buffer and a vtable in an object, the buffer overrun could corrupt the vtable and allow an attack to occur.

Even if you use **/GS**, you should strive to write secure code. That is, make sure that your code has no buffer overruns. **/GS**

might protect your application from buffer overruns that do remain in your code.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Code Generation** property page.
4. Modify the **Buffer Security Check** property.

To set this compiler option programmatically

- See BufferSecurityCheck.

Example

This sample overruns a buffer, which causes the application to fail at runtime.

```
// GS_buffer_security_check.cpp
// compile with: /c
#include <cstring>
#include <stdlib.h>
#pragma warning(disable : 4996)   // for strcpy use

// Vulnerable function
void vulnerable(const char *str) {
   char buffer[10];
   strcpy(buffer, str); // overrun buffer !!!

   // use a secure CRT function to help prevent buffer overruns
   // truncate string to fit a 10 byte buffer
   // strncpy_s(buffer, _countof(buffer), str, _TRUNCATE);
}

int main() {
   // declare buffer that is bigger than expected
   char large_buffer[] = "This string is longer than 10 characters!!";
   vulnerable(large_buffer);
}
```

See Also

Reference

Compiler Options

Setting Compiler Options

/Gs (Control Stack Checking Calls)

Controls stack probes.

```
/Gs[size]
```

Arguments

size

(Optional) The number of bytes that local variables can occupy before a stack probe is activated.

Remarks

This option is an advanced feature to control stack probes. A stack probe is a sequence of code the compiler inserts into every function call. When activated, a stack probe reaches benignly into memory by the amount of space required to store the associated function's local variables.

If a function requires more than *size* bytes of stack space for local variables, its stack probe is activated. The default value of *size* is the size of one page:

- 4 KB for x86
- 4 KB for x64
- 8 KB for Itanium

This value allows an application and the Windows memory manager to increase the amount of memory committed to the program stack at run time.

Note
The default value of *size* allows the program stack of applications for Windows to properly grow at run time. Do not change t he default setting of **/Gs** unless you know exactly why you need to change it.

Some programs, such as virtual device drivers, do not require this default stack-growth mechanism. In such cases, the stack probes are not necessary. You can stop the compiler from generating stack probes by setting *size* to a value that is larger than any function will require for local variable storage. No space is allowed between **/Gs** and *size*.

/Gs0 has the same effect as /Ge (Enable Stack Probes).

You can turn stack probes on or off by using check_stack. Note that **/Gs** and the **check_stack** pragma have no effect on standard C library routines; they affect only the functions you compile.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

Compiler Options

Setting Compiler Options

/GT (Support Fiber-Safe Thread-Local Storage)

Supports fiber safety for data allocated using static thread-local storage, that is, data allocated with **__declspec(thread)**.

```
/GT
```

Remarks

Data declared with **__declspec(thread)** is referenced through a thread-local storage (TLS) array. The TLS array is an array of addresses that the system maintains for each thread. Each address in this array gives the location of thread-local storage data.

A fiber is a lightweight object that consists of a stack and a register context and can be scheduled on various threads. A fiber can run on any thread. Because a fiber may get swapped out and restarted later on a different thread, the address of the TLS array must not be cached or optimized as a common subexpression across a function call (see the /Og (Global Optimizations) option for details). **/GT** prevents such optimizations.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Optimization** property page.
4. Modify the **Enable Fiber-safe Optimizations** property.

To set this compiler option programmatically

- See EnableFiberSafeOptimizations.

See Also

Reference

Compiler Options

Setting Compiler Options

/GX (Enable Exception Handling)

Enables synchronous exception handling with the assumption that **extern** C functions never throw an exception.

```
/GX
```

Remarks

It is equivalent to /EH (Exception Handling Model).

/GX is in effect, by default, when you compile from within the development environment. By default, **/GX-** is enabled when using command-line tools.

For more information, see C++ Exception Handling.

/GX is deprecated in Visual C++ 2005; use /EH (Exception Handling Model) instead. For more information, see Deprecated Compiler Options.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

Compiler Options

Setting Compiler Options

/Gy (Enable Function-Level Linking)

Allows the compiler to package individual functions in the form of packaged functions (COMDATs).

```
/Gy[-]
```

Remarks

The linker requires that functions be packaged separately as COMDATs to exclude or order individual functions in a DLL or .exe file.

You can use the linker option /OPT (Optimizations) to exclude unreferenced packaged functions from the .exe file.

You can use the linker option /ORDER (Put Functions in Order) to place packaged functions in a specified order within the .exe file.

Inline functions are always packaged if they are instantiated as calls (which occurs, for example, if inlining is turned off or you take a function address). In addition, C++ member functions defined within the class declaration are automatically packaged; other functions are not, and selecting this option is required to compile them as packaged functions.

Note
The /Z7, /Zi, /ZI (Debug Information Format) option, used for Edit and Continue, automatically sets the **/Gy** option.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Code Generation** property page.
4. Modify the **Enable Function-Level Linking** property.

To set this compiler option programmatically

- See EnableFunctionLevelLinking.

See Also

Reference

Compiler Options

Setting Compiler Options

/GZ (Enable Stack Frame Run-Time Error Checking)

Performs the same operations as the /RTC (Run-Time Error Checks) option. Deprecated.

```
/GZ
```

Remarks

/GZ is only for use in a nonoptimized (/Od (Disable (Debug))) build.

/GZ is deprecated in Visual C++ 2005; use /RTC (Run-Time Error Checks) instead. For more information, see Deprecated Compiler Options.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also
Reference
Compiler Options
Setting Compiler Options

/H (Restrict Length of External Names)

Restricts the length of external names.

```
/Hnumber
```

Arguments

number

Specifies the maximum length of external names allowed in a program.

Remarks

By default, the length of external (public) names is 2,047 characters. This is true for C and C++ programs. Using **/H** can only decrease the maximum allowable length of identifiers, not increase it. A space between **/H** and *number* is optional.

If a program contains external names longer than *number*, the extra characters are ignored. If you compile a program without **/H** and if an identifier contains more than 2,047 characters, the compiler will generate Fatal Error C1064.

The limit on length includes any compiler-created leading underscore (_) or at sign (@). These characters are part of the identifier and take a significant location.

- The compiler adds a leading underscore (_) to names modified by the **__cdecl** (default) and **__stdcall** calling conventions, and a leading at sign (@) to names modified by the **__fastcall** calling convention.
- The compiler appends argument size information to names modified by the **__fastcall** and **__stdcall** calling conventions, and adds type information to C++ names.

You may find **/H** useful:

- When you create mixed-language or portable programs.
- When you use tools that impose limits on the length of external identifiers.
- When you want to restrict the amount of space that symbols use in a debug build.

The following example shows how using **/H** can actually introduce errors if identifier lengths are limited too much:

```
// compiler_option_H.cpp
// compile with: /H5
// processor: x86
// LNK2005 expected
void func1(void);
void func2(void);

int main() { func1(); }

void func1(void) {}
void func2(void) {}
```

You must also be careful when using the **/H** option because of predefined compiler identifiers. If the maximum identifier length is too small, certain predefined identifiers will be unresolved as well as certain library function calls. For example, if the **printf** function is used and the option **/H5** is specified at compile time, the symbol **_prin** will be created in order to reference **printf**, and this will not be found in the library.

Use of **/H** is incompatible with /GL (Whole Program Optimization).

/H is deprecated in Visual C++ 2005; the maximum length limits have been increased and **/H** is no longer needed. For more information, see Deprecated Compiler Options.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.

2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also
Reference
Compiler Options
Setting Compiler Options

/HELP (Compiler Command-Line Help)

Displays a listing of compiler options to standard output.

```
/HELP
/help
/?
```

Remarks

To set this compiler option in the Visual Studio development environment

- This compiler option should only be accessed from the command line.

To set this compiler option programmatically

- This compiler option cannot be changed programmatically.

See Also

Reference

Compiler Options

Setting Compiler Options

/homeparams (Copy Register Parameters to Stack)

Forces parameters passed in registers to be written to their locations on the stack upon function entry.

```
/homeparams
```

Remarks

This compiler option is only for the x64 compilers (native and cross compile).

When parameters are passed in an x64 compilation, calling conventions require stackspace for parameters, even for parameters passed in registers. For more information, see Parameter Passing. However, by default in a release build, the register parameters will not be written to the stack, into the space that is already provided for the parameters. This makes it difficult to debug an optimized (release) build of your program.

For a release build, use **/homeparams** to ensure that you can debug your application. **/homeparams** does imply a performance disadvantage, because it does require a cycle to load the register parameters on to the stack.

In a debug build, the stack is always populated with parameters passed in registers.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

Compiler Options

Setting Compiler Options

/hotpatch (Create Hotpatchable Image)

Prepares an image for hotpatching. For more information on hotpatching, see http://www.microsoft.com/technet/security/topics/patch/patchmanagement.mspx.

```
/hotpatch
```

Remarks

When **/hotpatch** is used in a compilation, the compiler ensures that first instruction of each function is two bytes, which is required for hot patching.

After compiling with **/hotpatch**, you must link with /FUNCTIONPADMIN (Create Hotpatchable Image), which completes the preparation for making an image hotpatchable. When you compile and link an image with a single invocation of cl.exe, **/hotpatch** implies **/functionpadmin**.

When building an image that will run on one of the chips in the Itanium Processor Family (IPF), it is not necessary to use **/hotpatch**, however you should use **/functionpadmin** on images built for IPF.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also
Reference
Compiler Options
Setting Compiler Options

/I (Additional Include Directories)

Adds a directory to the list of directories searched for include files.

```
/I[ ]directory
```

Arguments

directory

The directory to be added to the list of directories searched for include files.

Remarks

To add more than one directory, use this option more than once. Directories are searched only until the specified include file is found.

You can use this option with the Ignore Standard Include Paths (/X (Ignore Standard Include Paths)) option.

The compiler searches for directories in the following order:

1. Directories containing the source file.
2. Directories specified with the **/I** option, in the order that CL encounters them.
3. Directories specified in the **INCLUDE** environment variable.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **General** property page.
4. Modify the **Additional Include Directories** property.

To set this compiler option programmatically

- See AdditionalIncludeDirectories.

Example

The following command looks for the include files requested by MAIN.c in the following order: first in the directory containing MAIN.c, then in the \INCLUDE directory, then in the \MY\INCLUDE directory, and finally in the directories assigned to the INCLUDE environment variable.

```
CL /I \INCLUDE /I\MY\INCLUDE MAIN.C
```

See Also

Reference

Compiler Options

Setting Compiler Options

/J (Default char Type Is unsigned)

Changes the default **char** type from **signed char** to **unsigned char**, and the **char** type is zero-extended when widened to an **int** type.

```
/J
```

Remarks

If a **char** value is explicitly declared **signed**, the **/J** option does not affect it, and the value is sign-extended when widened to an **int** type.

The **/J** option defines **_CHAR_UNSIGNED**, which is used with **#ifndef** in the LIMITS.h file to define the range of the default **char** type.

ANSI C and C++ do not require a specific implementation of the **char** type. This option is useful when you are working with character data that will eventually be translated into a language other than English.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Language** property page.
4. Modify the **Default Char Unsigned** property.

To set this compiler option programmatically

- See DefaultCharIsUnsigned.

See Also

Reference

Compiler Options

Setting Compiler Options

/link (Pass Options to Linker)

Passes one or more linker options to the linker.

```
/link linkeroptions
```

Arguments

linkeroptions

The linker option or options to be passed to the linker.

Remarks

The **/link** option and its linker options must appear after any file names and CL options. A space is required between **/link** and *linkeroptions*. For more information, see Setting Linker Options.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **Linker** folder.
3. Click a linker property page.
4. Modify one or more properties.

To set this compiler option programmatically

- This compiler option cannot be changed programmatically.

See Also

Reference

Compiler Options

Setting Compiler Options

/LN (Create MSIL Module)

Specifies that an assembly manifest should not be inserted into the output file.

```
/LN
```

Remarks

By default, **/LN** is not in effect (an assembly manifest is inserted into the output file).

When **/LN** is used, one of the /clr (Common Language Runtime Compilation) options must also be used.

A managed program that does not have an assembly metadata in the manifest is called a module. If you compile with /c (Compile Without Linking) and **/LN**, specify /NOASSEMBLY (Create a MSIL Module) in the linker phase to create the output file.

You may want to create modules if you want to take a component-based approach to building assemblies. That is, you can author types and compile them into modules. Then, you can generate an assembly from one or more modules. For more information on creating assemblies from modules, see .netmodule Files as Linker Input or Assembly Linker (Al.exe).

The default file extension for a module is .netmodule.

In Visual C++ releases before Visual C++ 2005, a module was created with **/clr:noAssembly**.

The Visual C++ linker accepts .netmodule files as input and the output file produced by the linker will be an assembly or .netmodule with no run-time dependence on any of the .netmodules that were input to the linker. For more information, see .netmodule Files as Linker Input.

To set this compiler option in the Visual Studio development environment

- Specify /NOASSEMBLY (Create a MSIL Module) in the linker phase to create the output file.

To set this compiler option programmatically

- This compiler option cannot be changed programmatically.

See Also

Reference

Compiler Options

Setting Compiler Options

/MD, /MT, /LD (Use Run-Time Library)

Lets you indicate if a multithreaded module is a DLL and select retail or debug versions of the run-time library.

```
/MD[d]
/MT[d]
/LD[d]
```

Remarks

Option	Description
/MD	Causes your application to use the multithread- and DLL-specific version of the run-time library. Defines **_MT** and **_DLL** and causes the compiler to place the library name MSVCRT.lib into the .obj file. Applications compiled with this option are statically linked to MSVCRT.lib. This library provides a layer of code that allows the linker to resolve external references. The actual working code is contained in MSVCR80.DLL, which must be available at run time to applications linked with MSVCRT.lib. When **/MD** is used with **_STATIC_CPPLIB** defined (**/D_STATIC_CPPLIB**) it will cause the application to link with the static multithread Standard C++ Library (libcpmt.lib) instead of the dynamic version (msvcprt.lib) while still dynamically linking to the main CRT via msvcrt.lib.
/MDd	Defines **_DEBUG**, **_MT**, and **_DLL** and causes your application to use the debug multithread- and DLL-specific version of the run-time library. It also causes the compiler to place the library name MSVCRTD.lib into the .obj file.
/MT	Causes your application to use the multithread, static version of the run-time library. Defines **_MT** and causes the compiler to place the library name LIBCMT.lib into the .obj file so that the linker will use LIBCMT.lib to resolve external symbols.
/MTd	Defines **_DEBUG** and **_MT**. This option also causes the compiler to place the library name LIBCMTD.lib into the .obj file so that the linker will use LIBCMTD.lib to resolve external symbols.
/LD	Creates a DLL. Passes the /DLL option to the linker. The linker looks for, but does not require, a **DllMain** function. If you do not write a **DllMain** function, the linker inserts a **DllMain** function that returns TRUE. Links the DLL startup code. Creates an import library (.lib), if an export (.exp) file is not specified on the command line; you link the import library to applications that call your DLL. Interprets /Fe (Name EXE File) as naming a DLL rather than an .exe file; the default program name becomes *basename*.dll instead of *basename*.exe. Implies **/MT** unless you explicitly specify **/MD**.
/LDd	Creates a debug DLL. Defines **_MT** and **_DEBUG**.

For more information about C run-time libraries and which libraries are used when you compile with /clr (Common Language Runtime Compilation), see C Run-Time Libraries.

All modules passed to a given invocation of the linker must have been compiled with the same run-time library compiler option (**/MD**, **/MT**, **/LD**).

For more information on using the debug versions of the run-time libraries, see Run-Time Library Reference.

Knowledge Base article Q140584 also discusses how to choose the appropriate C run-time library.

For further discussion of DLLs, see DLLs.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Code Generation** property page.
4. Modify the **Runtime Library** property.

To set this compiler option programmatically

- See RuntimeLibrary.

See Also
Reference
Compiler Options
Setting Compiler Options

/nologo (Suppress Startup Banner) (C/C++)

Suppresses the display of the sign-on banner when the compiler starts up and display of informational messages during compiling.

```
/nologo
```

Remarks

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **General** property page.
4. Modify the **Suppress Startup Banner** property.

To set this compiler option programmatically

- See SuppressStartupBanner.

See Also
Reference
Compiler Options
Setting Compiler Options

/O Options (Optimize Code)

The **/O** options control various optimizations that help you create code for maximum speed or minimum size.

- /O1, /O2 (Minimize Size, Maximize Speed) optimizes code for minimum size.
- /O1, /O2 (Minimize Size, Maximize Speed) optimizes code for maximum speed.
- /Ob (Inline Function Expansion) controls inline function expansion.
- /Od (Disable (Debug)) disables optimization, speeding compilation and simplifying debugging.
- /Og (Global Optimizations) enables global optimizations.
- /Oi (Generate Intrinsic Functions) generates intrinsic functions for appropriate function calls.
- /Os, /Ot (Favor Small Code, Favor Fast Code) tells the compiler to favor optimizations for size over optimizations for speed.
- /Os, /Ot (Favor Small Code, Favor Fast Code) (a default setting) tells the compiler to favor optimizations for speed over optimizations for size.
- /Ox (Full Optimization) selects full optimization.
- /Oy (Frame-Pointer Omission) suppresses the creation of frame pointers on the call stack for quicker function calls.

See Also
Reference
Compiler Options
Setting Compiler Options

/O1, /O2 (Minimize Size, Maximize Speed)

Selects a predefined set of options that affect the size and speed of files.

```
/O1
/O2
```

Remarks

The following table describes **/O1** and **/O2**.

Option	Equivalent to	Comment
/O1 (Minimize Size)	**/Og /Os /Oy /Ob2 /Gs /GF /Gy**	Creates the smallest code in the majority of cases.
/O2 (Maximize Speed)	**/Og /Oi /Ot /Oy /Ob2 /Gs /GF /Gy**	Creates the fastest code in the majority of cases. (default setting for release builds)

/O1 and **/O2** also enable the Named Return Value optimization, which eliminates the copy constructor and destructor of a stack based return value. Consider the following sample. The `Test` function will not create the copy constructor or destructor. Add output statements to the constructor, destructor and copy constructor to see the effect of the Named Return Value optimization when you run the program. For more information on Named Return Value Optimization in Visual C++, see http://go.microsoft.com/fwlink?linkid = 55900.

```
// O1_O2_NRVO.cpp
// compile with: /O1
struct A {
   A() {}
   ~A() {}
   A(const A& aa) {}
};

A Test() {
   A a;
   return a;
}
int main() {
   A aa;
   aa = Test();
}
```

x86 Specific

These options imply the use of the Frame-Pointer Omission (/Oy) option.

END x86 Specific

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Optimization** property page.
4. Modify the **Optimization** property.

To set this compiler option programmatically

- See Optimization.

See Also

Reference

/O Options (Optimize Code)

Compiler Options

Setting Compiler Options

/EH (Exception Handling Model)

/Ob (Inline Function Expansion)

Controls inline expansion of functions.

```
/Ob{0|1|2}
```

Arguments

0

Disables inline expansion, which is on by default.

1

Expands only functions marked as inline, __inline, __forceinline or **__inline** or, in a C++ member function, defined within a class declaration.

2

Expands functions marked as **inline** or **__inline** and any other function that the compiler chooses (expansion occurs at the compiler's discretion, often referred to as auto-inlining).

/Ob2 is in effect when /O1, /O2 (Minimize Size, Maximize Speed) or /Ox (Full Optimization) is used.

This option requires that you enable optimizations using **/O1**, **/O2**, **/Ox**, or **/Og**.

Remarks

The compiler treats the inline expansion options and keywords as suggestions. There is no guarantee that functions will be expanded inline. You cannot force the compiler to inline a particular function.

You can also use auto_inline to exclude functions from being considered as candidates for inline expansion. Also see intrinsic.

Note
Information that is gathered from profiling test runs will override optimizations that would otherwise be in effect if you speci fy **/Ob**, **/Os**, or **/Ot**. For more information, Profile-Guided Optimizations.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Optimization** property page.
4. Modify the **Inline Function Expansion** property.

To set this compiler option programmatically

- See InlineFunctionExpansion.

See Also

Reference

/O Options (Optimize Code)
Compiler Options
Setting Compiler Options

/Od (Disable (Debug))

Turns off all optimizations in the program and speeds compilation.

```
/Od
```

Remarks

This option is the default. Because **/Od** suppresses code movement, it simplifies the debugging process. For information on the compile-for-debugging options, see /Z7, /Zi, /ZI (Debug Information Format), /Z7, /Zi, /ZI (Debug Information Format), and /Z7, /Zi, /ZI (Debug Information Format).

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Optimization** property page.
4. Modify the **Optimization** property.

To set this compiler option programmatically

- See Optimization.

See Also

Reference

/O Options (Optimize Code)
Compiler Options
Setting Compiler Options

/Og (Global Optimizations)

Provides local and global optimizations, automatic-register allocation, and loop optimization. Deprecated.

```
/Og
```

Remarks

The following optimizations are available:

- Local and global common subexpression elimination

  In this optimization, the value of a common subexpression is calculated once. In the following example, if the values of `b` and `c` do not change between the three expressions, the compiler can assign the calculation of `b + c` to a temporary variable, and substitute the variable for `b + c`:

  ```
  a = b + c;
  d = b + c;
  e = b + c;
  ```

  For local common subexpression optimization, the compiler examines short sections of code for common subexpressions. For global common subexpression optimization, the compiler searches entire functions for common subexpressions.

- Automatic register allocation

  This optimization allows the compiler to store frequently used variables and subexpressions in registers; the **register** keyword is ignored.

- Loop optimization

  This optimization removes invariant subexpressions from the body of a loop. An optimal loop contains only expressions whose values change through each execution of the loop. In the following example, the expression `x + y` does not change in the loop body:

  ```
  i = -100;
  while( i < 0 ) {
      i += x + y;
  }
  ```

  After optimization, `x + y` is calculated once rather than every time the loop is executed:

  ```
  i = -100;
  t = x + y;
  while( i < 0 ) {
      i += t;
  }
  ```

  Loop optimization is much more effective when the compiler can assume no aliasing, which you set with __restrict, noalias, or restrict.

Note
You can enable or disable global optimization on a function-by-function basis using the **optimize** pragma with the **g** o ption.

/Og also enables the Named Return Value optimization, which eliminates the copy constructor and destructor of a stack based

return value. See /O1, /O2 (Minimize Size, Maximize Speed) for more information.

For related information, see Generate Intrinsic Functions (/Oi) and Full Optimization (/Ox).

/Og is deprecated in Visual C++ 2005; use either /O1 or **/O2**. For more information, see Deprecated Compiler Options.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also
Reference
/O Options (Optimize Code)
Compiler Options
Setting Compiler Options

/Oi (Generate Intrinsic Functions)

Replaces some function calls with intrinsic or otherwise special forms of the function that help your application run faster.

```
/Oi[-]
```

Remarks

Programs that use intrinsic functions are faster because they do not have the overhead of function calls, but may be larger because of the additional code created.

See intrinsic for more information on which functions have intrinsic forms.

/Oi is only a request to the compiler to replace some function calls with intrinsics; the compiler may call the function (and not replace the function call with an intrinsic) if it will result in better performance.

x86 Specific

Note
The _alloca and setjmp functions are always generated inline; this behavior is not affected by **/Oi**.

The intrinsic floating-point functions do not perform any special checks on input values and so work in restricted ranges of input, and have different exception handling and boundary conditions than the library routines with the same name. Using the true intrinsic forms implies loss of IEEE exception handling, and loss of **_matherr** and **errno** functionality; the latter implies loss of ANSI conformance. However, the intrinsic forms can considerably speed up floating-point-intensive programs, and for many programs, the conformance issues are of little practical value.

You can use /Za, /Ze (Disable Language Extensions) to override generation of true intrinsic floating-point options. In this case, the functions are generated as library routines that pass arguments directly to the floating-point chip instead of pushing them onto the program stack.

END x86 Specific

You also use intrinsic to create intrinsic functions, or function (C/C++) to explicitly force a function call.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Optimization** property page.
4. Modify the **Enable Intrinsic Functions** property.

To set this compiler option programmatically

- See EnableIntrinsicFunctions.

See Also

Reference

/O Options (Optimize Code)
Compiler Options
Setting Compiler Options
Compiler Intrinsics

/Os, /Ot (Favor Small Code, Favor Fast Code)

Minimizes or maximizes the size of EXEs and DLLs.

```
/Os
/Ot
```

Remarks

/Os (Favor Small Code) minimizes the size of EXEs and DLLs by instructing the compiler to favor size over speed. The compiler can reduce many C and C++ constructs to functionally similar sequences of machine code. Occasionally these differences offer tradeoffs of size versus speed. The **/Os** and **/Ot** options allow you to specify a preference for one over the other:

/Ot (Favor Fast Code) maximizes the speed of EXEs and DLLs by instructing the compiler to favor speed over size. (This is the default.) The compiler can reduce many C and C++ constructs to functionally similar sequences of machine code. Occasionally, these differences offer tradeoffs of size versus speed. The /Ot option is implied by the Maximize Speed (**/O2**/O1, /O2 (Minimize Size, Maximize Speed)) option. The **/O2** option combines several options to produce very fast code.

If you use **/Os** or **/Ot**, then you must also specify **/Og**/Og (Global Optimizations) to optimize the code.

Note
Information that is gathered from profiling test runs will override optimizations that would otherwise be in effect if you speci fy **/Ob**, **/Os**, or **/Ot**. For more information, Profile-Guided Optimizations.

x86 Specific

The following example code demonstrates the difference between the Favor Small Code (**/Os**) options and the Favor Fast Code (**/Ot**) option:

Note
The following describes the expected behavior when using **/Os** or **/Ot**. However, compiler behavior from release to release may result in different optimizations for the code below.

```
/* differ.c
  This program implements a multiplication operator
  Compile with /Os to implement multiply explicitly as multiply.
  Compile with /Ot to implement as a series of shift and LEA instructions.
*/
int differ(int x)
{
    return x * 71;
}
```

As shown in the fragment of machine code below, when DIFFER.c is compiled for size (**/Os**), the compiler implements the multiply expression in the return statement explicitly as a multiply to produce a short but slower sequence of code:

```
    mov    eax, DWORD PTR _x$[ebp]
    imul   eax, 71                  ; 00000047H
```

Alternately, when DIFFER.c is compiled for speed (**/Ot**), the compiler implements the multiply expression in the return statement as a series of shift and `LEA` instructions to produce a fast but longer sequence of code:

```
    mov    eax, DWORD PTR _x$[ebp]
    mov    ecx, eax
    shl    eax, 3
    lea    eax, DWORD PTR [eax+eax*8]
    sub    eax, ecx
```

END x86 Specific

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Optimization** property page.
4. Modify the **Favor Size or Speed** property.

To set this compiler option programmatically

- See FavorSizeOrSpeed.

See Also
Reference
/O Options (Optimize Code)
Compiler Options
Setting Compiler Options

/Ox (Full Optimization)

Combines optimizing options to produce code that favors execution speed over smaller code size.

```
/Ox
```

Remarks

/Ox can be combined with **/Os** (**/Oxs**) to favor smaller code size (optimize for size).

In general, **/O2** should be preferred over **/Ox** and **/O1** over **/Oxs**.

Using **/Ox** is the same as using the following options:

- **/Ob** *n*, where n = 2
- /Og (Global Optimizations)
- /Oi (Generate Intrinsic Functions)
- /Os, /Ot (Favor Small Code, Favor Fast Code)
- /Oy (Frame-Pointer Omission)

Note
The use of Full Optimization implies the use of the Frame Pointer Omission (/Oy (Frame-Pointer Omission)) option.

/Ox is mutually exclusive from:

- /O1, /O2 (Minimize Size, Maximize Speed)
- /O1, /O2 (Minimize Size, Maximize Speed)
- /Od (Disable (Debug))

/Ox also enables the Named Return Value optimization, which eliminates the copy constructor and destructor of a stack based return value. See /O1, /O2 (Minimize Size, Maximize Speed) for more information.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Optimization** property page.
4. Modify the **Optimization** property.

To set this compiler option programmatically

- See Optimization.

See Also

Reference

/O Options (Optimize Code)
Compiler Options
Setting Compiler Options

/Oy (Frame-Pointer Omission)

Suppresses creation of frame pointers on the call stack.

```
/Oy
```

Remarks

This option speeds function calls, because no frame pointers need to be set up and removed. It also frees one more register, (EBP on the Intel 386 or later) for storing frequently used variables and sub-expressions.

/Oy is only available in x86 compilers.

If your code requires EBP-based addressing, you can specify the /Oy– option after the **/Ox** option or use optimize with the "**y**" and **off** arguments to gain maximum optimization with EBP-based addressing. The compiler detects most situations where EBP-based addressing is required (for instance, with the **_alloca** and **setjmp** functions and with structured exception handling).

The /Ox (Full Optimization) (Full Optimization) and /O1, /O2 (Minimize Size, Maximize Speed) (Fast Code) options imply **/Oy**. Specifying **/Oy–** after the **/Ox**, **/O1**, or **/O2** option disables **/Oy**, whether it is explicit or implied.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Optimization** property page.
4. Modify the **Omit Frame Pointers** property.

To set this compiler option programmatically

- See OmitFramePointers.

See Also

Reference

/O Options (Optimize Code)
Compiler Options
Setting Compiler Options

/openmp (Enable OpenMP 2.0 Support)

Causes the compiler to process **#pragma** omp.

```
/openmp
```

Remarks

#pragma omp is used to specify OpenMP Directives and OpenMP Clauses. If **/openmp** is not specified in a compilation, the compiler ignores OpenMP clauses and directives. OpenMP Function calls are processed by the compiler even if **/openmp** is not specified.

An application compiled with **/openmp** and using the OpenMP Libraries can only be run on Windows 2000 or later operating systems.

Applications compiled with **/openmp** and **/clr** can only be run in a single application domain process; multiple application domains are not supported. That is, when the module constructor (.cctor) is run, it will detect the process is compiled with **/openmp** and if the application is being loaded into a non-default runtime. For more information, see appdomain, /clr (Common Language Runtime Compilation), and Initialization of Mixed Assemblies.

If you attempt to load an application compiled with **/openmp** and **/clr** into a non-default application domain, a TypeInitializationException exception will be thrown outside the debugger and a OpenMPWithMultipleAppdomainsException exception will be thrown in the debugger.

These exceptions can also be raised in the following situations:

- If your application compiled with **/clr**, but not with **/openmp**, is loaded into a non-default application domain but where the process includes an application that was compiled with **/openmp**.
- If you pass your **/clr** application to a utility, such as regasm.exe (Assembly Registration Tool (Regasm.exe)), which loads its target assemblies into a non-default application domain.

The common language runtime's code access security doesn't work in OpenMP regions. If you apply a CLR code access security attribute outside a parallel region, it won't be in effect in the parallel region.

Microsoft advises that you do not write **/openmp** applications that allows partially trusted callers, using AllowPartiallyTrustedCallersAttribute, or any CLR code access security attributes.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Expand the **Configuration Properties** node.
3. Expand the **C/C++** node.
4. Select the **Language** property page.
5. Modify the **OpenMP Support** property.

To set this compiler option programmatically

- See OpenMP.

Example

The following sample shows some of the effects of threadpool startup versus using the threadpool after it started up. Assuming an x64, single core, dual processor the threadpool takes about 16ms to startup. After that though there is very little cost for the threadpool.

When you compile with **/openmp**, the second call to test2 never runs any longer than if you compile with **/openmp-**, as there is no threadpool startup. At a million iterations the **/openmp** version is faster than the **/openmp-** version for the second call to test2, and at 25 iterations both **/openmp-** and **/openmp** versions register less than the clock granularity.

So if you have only one loop in your application and it runs in less than 15ms (adjusted for the approximate overhead on your

machine), **/openmp** may not be appropriate, but if it's anything more than that, you may want to consider using **/openmp**.

```
// cpp_compiler_options_openmp.cpp
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

volatile DWORD dwStart;
volatile int global = 0;

double test2(int num_steps) {
   int i;
   global++;
   double x, pi, sum = 0.0, step;

   step = 1.0 / (double) num_steps;

   #pragma omp parallel for reduction(+:sum) private(x)
   for (i = 1; i <= num_steps; i++) {
      x = (i - 0.5) * step;
      sum = sum + 4.0 / (1.0 + x*x);
   }

   pi = step * sum;
   return pi;
}

int main(int argc, char* argv[]) {
   double   d;
   int n = 1000000;

   if (argc > 1)
      n = atoi(argv[1]);

   dwStart = GetTickCount();
   d = test2(n);
   printf_s("For %d steps, pi = %.15f, %d milliseconds\n", n, d, GetTickCount() - dwStart);

   dwStart = GetTickCount();
   d = test2(n);
   printf_s("For %d steps, pi = %.15f, %d milliseconds\n", n, d, GetTickCount() - dwStart);
}
```

See Also

Reference

Compiler Options

Setting Compiler Options

/P (Preprocess to a File)

Preprocesses C and C++ source files and writes the preprocessed output to a file.

```
/P
```

Remarks

The file has the same base name as the source file and an .i extension. In the process, all preprocessor directives are carried out, macro expansions are performed, and comments are removed. To preserve comments in the preprocessed output, use the /C (Preserve Comments During Preprocessing) option along with **/P**.

/P adds **#line** directives to the output, at the beginning and end of each included file and around lines removed by preprocessor directives for conditional compilation. These directives renumber the lines of the preprocessed file. As a result, errors generated during later stages of processing refer to the line numbers of the original source file rather than lines in the preprocessed file. To suppress the generation of **#line** directives, use /EP (Preprocess to stdout Without #line Directives) as well as **/P**.

The **/P** option suppresses compilation. It does not produce an .obj file, even if you use /Fo (Object File Name). You must resubmit the preprocessed file for compilation. **/P** also suppresses the output files from the /FA, /Fa (Listing File), /FA, /Fa (Listing File), and /Fm (Name Mapfile) options.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Preprocessor** property page.
4. Modify the **Generate Preprocessed File** property.

To set this compiler option programmatically

- See GeneratePreprocessedFile.

Example

The following command line preprocesses `ADD.C`, preserves comments, adds **#line** directives, and writes the result to a file, `ADD.I`:

```
CL /P /C ADD.C
```

See Also

Reference

Compiler Options

Setting Compiler Options

/Q Options (Low-Level Operations)

The **/Q** compiler options let you perform low-level compiler operations.

- /QIfist (Suppress _ftol)
- /QIPF_B (Errata for B CPU Stepping)
- /QIPF_C (Errata for C CPU Stepping)
- /QIPF_fr32 (Do Not Use Upper 96 Floating Point Registers)
- /QIPF_noPIC (Generate Position Dependent Code)
- /QIPF_restrict_plabels (Assume No Functions Created at Run Time)

See Also
Reference
Compiler Options
Setting Compiler Options

/QIfist (Suppress _ftol)

Suppresses the call of the helper function **_ftol** when a conversion from a floating-point type to an integral type is required.

```
/QIfist
```

Remarks

Note
/QIfist is only available in the compiler targeting x86; this compiler option is not available in the compilers targeting x64 or Itanium.

In addition to converting from a floating-point type to integral type, the **_ftol** function ensures the rounding mode of the floating-point unit (FPU) is toward zero (truncate), by setting bits 10 and 11 of the control word. This guarantees that converting from a floating-point type to an integral type occurs as described by the ANSI C standard (the fractional portion of the number is discarded). When using **/QIfist**, this guarantee no longer applies. The rounding mode will be one of four as documented in Intel reference manuals:

- Round toward nearest (even number if equidistant)
- Round toward negative infinity
- Round toward positive infinity
- Round toward zero

You can use the _control87, _controlfp, __control87_2 C Run-Time function to modify the rounding behavior of the FPU. The default rounding mode of the FPU is "Round toward nearest." Using **/QIfist** can improve the performance of your application, but not without risk. You should thoroughly test the portions of your code that are sensitive to rounding modes before relying upon code built with **/QIfist** in production environments.

/arch (Minimum CPU Architecture) and **/QIfist** can not be used on the same compiland.

Note
/QIfist is not in effect by default because the rounding bits also affect floating point to floating point rounding (which occurs after every calculation), so when you set the flags for C-style (toward zero) rounding, your floating point calculations might be different. **/QIfist** should not be used if your code depends upon the expected behavior of truncating the fractional portion of the floating-point number. If you are unsure, do not use **/QIfist**.

In Visual C++ 2005, **/QIfist** is deprecated. The compiler has made significant improvements in float to int conversion speed. For more information, see Deprecated Compiler Options.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

/Q Options (Low-Level Operations)

Compiler Options

/QIPF_B (Errata for B CPU Stepping)

Causes compiler to not generate sequences of instructions that give unexpected results, according to the errata for the specified B CPU stepping.

```
/QIPF_B#[-]
```

Remarks

This option is off by default. Appending a minus sign to the option explicitly disables it.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

/Q Options (Low-Level Operations)
Compiler Options
Setting Compiler Options

/QIPF_C (Errata for C CPU Stepping)

Does not generate sequences of instructions that give unexpected results, according to the errata for the specified C CPU stepping.

```
/QIPF_C#[-]
```

Remarks

This option is off by default. Appending a minus sign to the option explicitly disables it.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

/Q Options (Low-Level Operations)
Compiler Options
Setting Compiler Options

/QIPF_fr32 (Do Not Use Upper 96 Floating Point Registers)

Tells the compiler to only use the lower 32 floating-point registers and to not use the upper 96 floating-point registers when compiling for Itanium.

```
/QIPF_fr32
```

Remarks

Note
/QIPF_fr32 is only available in compilers targeting Itanium. This compiler option is not available in the compilers targeting x 64 or x86.

When writing a device driver, which executes in kernel mode context, compiling with **/QIPF_fr32** is strongly recommended. If an application switches out of kernel mode and one of its upper 96 registers is not intact, as could happen if an interrupt occurred, the application will fail when it switches back to user mode. Consider the following scenario:

- An application runs in user mode, when an interrupt occurs.
- Control passes to the kernel, and the upper 96 floating-point registers are not saved.
- The kernel calls a driver that was compiled without **/QIPF_fr32**.
- The driver destroys one of the upper floating-point registers
- Driver returns to kernel mode.
- Kernel returns from interrupt, and one of the application's upper floating-point registers is destroyed.

In addition to the correctness that it gives applications that switch to user mode from kernel mode, **/QIPF_fr32** also enhances the performance of context changes in user mode (upper 96 floating-point registers do not have to be saved or reloaded).

To set this compiler option in the Visual Studio development environment

- **/QIPF_fr32** is not available in the Visual Studio development environment.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

/Q Options (Low-Level Operations)
Compiler Options
Setting Compiler Options

/QIPF_restrict_plabels (Assume No Functions Created at Run Time)

Assumes that no functions are created at run time.

```
/QIPF_restrict_plabels
```

Remarks

Note
/QIPF_restrict_plabels is only available in compilers targeting Itanium. This compiler option is not available in the compilers targeting x64 or x86.

If your program does not create functions at run time, you can compile with **/QIPF_restrict_plabels** and the compiler will treat pointers to function descriptors as __restrict pointers, so that the contents cannot be overwritten.

With **/QIPF_restrict_plabels**, the compiler can perform optimizations with indirect calls, such as hoisting the load of a virtual method address out of a loop. This can result in performance enhancements when there are a lot of indirect function calls.

To set this compiler option in the Visual Studio development environment

- **/QIPF_restrict_plabels** is not available in the Visual Studio development environment.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

/Q Options (Low-Level Operations)
Compiler Options
Setting Compiler Options

/QIPF_noPIC (Generate Position Dependent Code)

By default, the Visual C++ compiler for Itanium generates position independent code. **/QIPF_noPIC** generates an image with position dependent code.

```
/QIPF_noPIC
```

Remarks

Note
/QIPF_noPIC is only available in compilers targeting Itanium. This compiler option is not available in the compilers targeting x64 or x86.

Code that is position independent can be loaded more quickly than code that is position dependent, if the image cannot be loaded at its preferred address, because relative addresses in position independent code are not patched if the image is loaded at an address other than its preferred address. Also, Windows allows users to share the same code if it is not patched, whereas each user gets a separate instance of patched code.

However, code that is position independent may have poorer performance than code that is position dependent, if the image cannot be loaded at the preferred address. A server application, for example, where users may not care about (potential) longer startup times, and where users rarely run several copies of those applications at the same time, may benefit from the additional performance that position dependent code gives.

To set this compiler option in the Visual Studio development environment

- **/QIPF_noPIC** is not available in the Visual Studio development environment.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

/Q Options (Low-Level Operations)
Compiler Options
Setting Compiler Options

/RTC (Run-Time Error Checks)

Used to enable and disable the run-time error checks feature, in conjunction with the runtime_checks pragma.

```
/RTC1
/RTCc
/RTCs
/RTCu
```

Arguments

1

Equivalent of **/RTC***su*.

c

Reports when a value is assigned to a smaller data type and results in a data loss. For example, if a value of type `short 0x101` is assigned to a variable of type `char`.

This option reports situations in which you intend to truncate, for example, if you want the first eight bits of an **int** returned as a **char**. Because **/RTC***c* causes a run-time error if any information is lost as a result of the assignment, you can mask off the information you need to avoid a run-time error as a result of **/RTC***c*. For example:

```
#include <crtdbg.h>

char get8bits(int value, int position) {
   _ASSERT(position < 32);
   return (char)(value >> position);
   // Try the following line instead:
   // return (char)((value >> position) && 0xff);
}

int main() {
   get8bits(12341235,3);
}
```

s

Enables stack frame run-time error checking, as follows:

- Initialization of local variables to a nonzero value. This helps identify bugs that do not appear when running in debug mode. There is a greater chance that stack variables will still be zero in a debug build compared to a release build because of compiler optimizations of stack variables in a release build. Once a program has used an area of its stack, it is never reset to 0 by the compiler. Therefore, subsequent, uninitialized stack variables that happen to use the same stack area can return values left over from the prior use of this stack memory.
- Detection of overruns and underruns of local variables such as arrays. **/RTC***s* will not detect overruns when accessing memory that results from compiler padding within a structure. Padding could occur by using align (C++), /Zp (Struct Member Alignment), or pack, or if you order structure elements in such a way as to require the compiler to add padding.
- Stack pointer verification, which detects stack pointer corruption. Stack pointer corruption can be caused by a calling convention mismatch. For example, using a function pointer, you call a function in a DLL that is exported as __stdcall but you declare the pointer to the function as __cdecl.

u

Reports when a variable is used without having been initialized. For example, an instruction that generates Compiler Warning (level 4) C4701 may also generate a run-time error under **/RTC***u*. Any instruction that generates Compiler Warning (level 1 and level 4) C4700 will generate a run-time error under **/RTC***u*.

However, consider the following code fragment:

```
int a, *b, c;
if ( 1 )
b = &a;
c = a;  // No run-time error with /RTCu
```

If a variable could have been initialized, it will not be reported at run time by **/RTC***u*. For example, after a variable is aliased through a pointer, the compiler will not track the variable and report uninitialized uses. In effect, you can initialize a variable by taking its address. The & operator works like an assignment operator in this situation.

Remarks

Run-time error checks are a way for you to find problems in your running code; for more information, see How to: Use Native Run-Time Checks.

If you compile your program at the command line using any of the **/RTC** compiler options, any pragma optimize instructions in your code will silently fail. This is because run-time error checks are not valid in a release (optimized) build.

You should use **/RTC** for development builds; **/RTC** should not be used for a retail build. **/RTC** cannot be used with compiler optimizations (/O Options (Optimize Code)). A program image built with **/RTC** will be slightly larger and slightly slower than an image built with **/Od** (up to 5 percent slower than an **/Od** build).

The __MSVC_RUNTIME_CHECKS preprocessor directive will be defined when you use any **/RTC** option or /GZ.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Code Generation** property page.
4. Modify one or both of the following properties: **Basic Runtime Checks** or **Smaller Type Check**.

To set this compiler option programmatically

- See BasicRuntimeChecks and SmallerTypeCheck properties.

Smart Device Developer Notes

For device compilers, the RTC feature is supported only on x86, and is not exposed as a property setting for Visual C++ device projects in the Visual Studio IDE. However, if you are targeting x86, this option may be set in the IDE in the **Additional options** pane of the **C/C++ Command Line** property page.

See Also

Tasks

RTC Sample: Run-Time Error Checks

Reference

Compiler Options

Setting Compiler Options

/showIncludes (List Include Files)

Causes the compiler to output a list of the include files. Nested include files are also displayed (files that are included from the files that you include).

```
/showIncludes
```

Remarks

When an include file is encountered during compilation, a message is output, for example:

```
Note: including file: d:\MyDir\include\stdio.h
```

Nested include files are indicated by an indentation, one space for each level of nesting, for example:

```
Note: including file: d:\temp\1.h
Note: including file:  d:\temp\2.h
```

In this case, `2.h` was included from within `1.h`, hence the indentation.

The **/showIncludes** option emits to **stderr**, not **stdout**.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Advanced** property page.
4. Modify the **Show Includes** property.

To set this compiler option programmatically

- See ShowIncludes.

See Also

Reference

Compiler Options

Setting Compiler Options

/Tc, /Tp, /TC, /TP (Specify Source File Type)

The **/Tc** option specifies that *filename* is a C source file, even if it does not have a .c extension. The **/Tp** option specifies that *filename* is a C++ source file, even if it doesn't have a .cpp or .cxx extension. A space between the option and *filename* is optional. Each option specifies one file; to specify additional files, repeat the option.

/TC and **/TP** are global variants of **/Tc** and **/Tp**. They specify to the compiler to treat all files named on the command line as C source files (**/TC**) or C++ source files (**/TP**), without regard to location on the command line in relation to the option. These global options can be overridden on a single file by means of **/Tc** or **/Tp**.

```
/Tcfilename
/Tpfilename
/TC
/TP
```

Arguments

filename

A C or C++ source file.

Remarks

By default, CL assumes that files with the .c extension are C source files and files with the .cpp or the .cxx extension are C++ source files.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Advanced** property page.
4. Modify the **Compile As** property.

To set this compiler option programmatically

- See CompileAs.

Examples

The following CL command line specifies that MAIN.c, TEST.prg, and COLLATE.prg are all C source files. CL will not recognize PRINT.prg.

```
CL MAIN.C /TcTEST.PRG /TcCOLLATE.PRG PRINT.PRG
```

The following CL command line specifies that TEST1.c, TEST2.cxx, TEST3.huh, and TEST4.o are compiled as C++ files, and TEST5.z is compiled as a C file.

```
CL TEST1.C TEST2.CXX TEST3.HUH TEST4.O /Tc TEST5.Z /TP
```

See Also

Reference

Compiler Options

Setting Compiler Options

/U, /u (Undefine Symbols)

Undefine previously defined symbols. **/U** undefines the specified symbol. **/u** undefines all previously defined symbols.

```
/U[ ]symbol
/u
```

Arguments

symbol

The symbol you want undefined.

Remarks

Neither option can undefine symbols created with the **#define** directive.

Both options undefine the following Microsoft-specific macros.

Symbol	Function
_CHAR_UNSIGNED	Default char type is unsigned. Defined when **/J** is specified.
_CPPRTTI	Defined for code compiled with the **/GR** (Enable Run-Time Type Information) option.
_CPPUNWIND	Defined for code compiled with the **/EH***sc* (Enable Exception Handling) option.
_DLL	Defined when **/MD** is specified.
_M_IX86	Defined to 600, by default, for x86 targets.
_MSC_VER	For more information, see Predefined Macros.
_WIN32	Defined for applications for WIN32. Always defined.
_MT	Defined when **/MD** or **/MT** is specified.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Advanced** property page.
4. Modify the **Undefine Preprocessor Definitions** or **Undefine All Preprocessor Definitions** properties.

To set this compiler option programmatically

- See UndefineAllPreprocessorDefinitions or UndefinePreprocessorDefinitions.

See Also

Reference

Compiler Options

Setting Compiler Options

/V (Version Number)

Embeds a text string in the .obj file. Deprecated.

```
/Vstring
```

Arguments
string

A string specifying the version number or copyright notice to be embedded in an .obj file.

Remarks

The string can label an .obj file with a version number or a copyright notice. Any space or tab characters must be enclosed in double quotation marks (") if they are a part of the string. A backslash (\) must precede any double quotation marks if they are a part of the string. A space between **/V** and *string* is optional.

You can also use comment (C/C++) with the compiler comment-type argument to place the name and version number of the compiler in the .obj file.

/V is deprecated in Visual C++ 2005; **/V** was primarily used to support building virtual device drivers (VxDs), and building VxDs is no longer supported by the Visual C++ toolset. For more information, see Deprecated Compiler Options.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also
Reference
Compiler Options
Setting Compiler Options

/vd (Disable Construction Displacements)

/vd supports incorrect behavior in an early version of Visual C++, and is no longer needed.

```
/vdn
```

Arguments

0

Suppresses the vtordisp constructor/destructor displacement member. Choose this option only if you are certain that all class constructors and destructors call virtual functions virtually.

1

Enables the creation of hidden vtordisp constructor/destructor displacement members. This choice is the default.

2

Allows you to use dynamic_cast Operator on an object being constructed. For example, a dynamic_cast from a virtual base class to a derived class.

/vd2 adds a vtordisp field when you have a virtual base with virtual functions. **/vd1** should be sufficient. The most common case where **/vd2** is necessary is when the only virtual function in your virtual base is a destructor.

Remarks

These options apply only to C++ code that uses virtual bases.

Visual C++ implements C++ construction displacement support in situations where virtual inheritance is used. Construction displacements solve the problem created when a virtual function, declared in a virtual base and overridden in a derived class, is called from a constructor during construction of a further derived class.

The problem is that the virtual function may be passed an incorrect *this* pointer as a result of discrepancies between the displacements to the virtual bases of a class and the displacements to its derived classes. The solution provides a single construction displacement adjustment, called a vtordisp field, for each virtual base of a class.

By default, vtordisp fields are introduced whenever the code defines user-defined constructors and destructors and also overrides virtual functions of virtual bases.

These options affect entire source files. Use vtordisp to suppress and then re-enable vtordisp fields on a class-by-class basis.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

Compiler Options

Setting Compiler Options

/vmb, /vmg (Representation Method)

Select the method that the compiler uses to represent pointers to class members.

Use **/vmb** if you always define a class before you declare a pointer to a member of the class.

Use **/vmg** to declare a pointer to a member of a class before defining the class. This need can arise if you define members in two different classes that reference each other. For such mutually referencing classes, one class must be referenced before it is defined.

```
/vmb
/vmg
```

Remarks

You can also use pointers_to_members or Inheritance Keywords in your code to specify a pointer representation.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

Compiler Options

Setting Compiler Options

/vmm, /vms, /vmv (General Purpose Representation)

Used when /vmb, /vmg (Representation Method) is selected as the /vmb, /vmg (Representation Method). These options indicate the inheritance model of the not-yet-encountered class definition.

```
/vmm
/vms
/vmv
```

Remarks

The options are described in the following table.

Option	Description
/vmm	Specifies the most general representation of a pointer to a member of a class to be one that uses multiple inheritance. The corresponding Inheritance Keywords and argument to pointers_to_members is **multiple_inheritance**. This representation is larger than that required for single inheritance. If the inheritance model of a class definition for which a pointer to a member is declared is virtual, the compiler generates an error.
/vms	Specifies the most general representation of a pointer to a member of a class to be one that uses either no inheritance or single inheritance. The corresponding Inheritance Keywords and argument to pointers_to_members is **single_inheritance**. This is the smallest possible representation of a pointer to a member of a class. If the inheritance model of a class definition for which a pointer to a member is declared is multiple or virtual, the compiler generates an error.
/vmv	Specifies the most general representation of a pointer to a member of a class to be one that uses virtual inheritance. It never causes an error and is the default. The correspondingInheritance Keywords and argument to pointers_to_members is **virtual_inheritance**. This option requires a larger pointer and additional code to interpret the pointer than the other options.

When you specify one of these inheritance-model options, that model is used for all pointers to class members, regardless of their inheritance type or whether the pointer is declared before or after the class. Therefore, if you always use single-inheritance classes, you can reduce code size by compiling with **/vms**; however, if you want to use the most general case (at the expense of the largest data representation), compile with **/vmv**.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

/vmb, /vmg (Representation Method)

/w, /Wn, /WX, /Wall, /wln, /wdn, /wen, /won (Warning Level)

Specify how the compiler generates warnings for a given compilation.

```
/w
/Wn
/WX
/Wall
/wln
/wdn
/wen
/won
```

Remarks

The options and related arguments are described in the following table.

Option	**Description**
/w	Disables all compiler warnings.
/W *n*	Specifies the highest level of warning generated by the compiler. Valid warning levels for *n* range from 0 to 4: • Level 0 disables all warnings. • Level 1 displays severe warnings. Level 1 is the default warning level at the command line. • Level 2 displays all level 1 warnings and warnings less severe than level 1. • Level 3 displays all level 2 warnings and all other warnings recommended for production purposes. • Level 4 displays all level 3 warnings plus informational warnings, which in most cases can be safely ignored. This option should be used only to provide "lint" level warnings and is not recommended as your usual warning level setting. For a new project, it may be best to use `/W4` in all compilations. This will ensure the fewest possible hard-to-find code defects.
/Wall	Enables all warnings, including those disabled by default. See Compiler Warnings That Are Off By Default.
/WX	Treats all compiler warnings as errors. For a new project, it may be best to use **/WX** in all compilations; resolving all warnings will ensure the fewest possible hard-to-find code defects. The linker also has a **/WX** option; see /WX (Treat Linker Warnings as Errors) for more information.
/w *ln*	Specifies the level for a particular warning. The first parameter sets the warning level (same as **/W***n*) and the second parameter is the actual warning number. For example, `/w14326` causes C4326 to be generated as a level 1 warning.
/wd *n*	Disables the specified compiler warning where *n* is the compiler warning number. For example, `/wd4326` disables compiler warning C4326.
/we *n*	Treats the specific compiler warning as an error where *n* is a compiler warning. For example, `/we4326` flags warning number C4326 as an error.

/wo *n*	Reports the error only once where *n* is a compiler warning. For example, `/wo4326` will cause warning C4326 to be reported only once.

If you create a precompiled header (/Yc (Create Precompiled Header File)) with one of the **/w** options, any use of the precompiled header (/Yu (Use Precompiled Header File)) will cause those same **/w** options to be in effect again. You can override the **/w** setting in the precompiled header with another **/w** option at the command line.

Pragma directives in source code are unaffected by the **/w** option.

You can also use warning to control the level of warning reported at compile time.

The C/C++ Build Errors describes the warnings, indicates each warning's level, and indicates potential problems (rather than actual coding errors) with statements that may not compile as you intend.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **General** property page and modify the **Warning Level** or **Treat Warnings as Errors** properties.
4. Click the **Advanced** property page and modify the **Disable Specific Warnings** property.
5. For the remaining options, click the **Command Line** property page and type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See WarningLevel, WarnAsError, DisableSpecificWarnings, and AdditionalOptions.

See Also
Reference
Compiler Options
Setting Compiler Options

/WL (Enable One-Line Diagnostics)

Appends additional information to an error or warning message.

```
/WL
```

Remarks

Error and warning messages from the C++ compiler can be followed by additional information that appears, by default, on a new line. When you compile from the command line, the additional line of information can be appended to the error or warning message. This might be desirable if you capture your build output to a log file and then process that log to find all errors and warnings. A semicolon will separate the error or warning message from the additional line.

Not all error and warning messages have an additional line of information. The following code will generate an error that has an additional line of information; it will let you test the effect when you use **/WL**.

```
// compiler_option_WL.cpp
// compile with: /WL
#include <queue>
int main() {
   std::queue<int> q;
   q.fromthecontinuum();   // C2039
}
```

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also
Reference
Compiler Options
Setting Compiler Options

/Wp64 (Detect 64-Bit Portability Issues)

Detects 64-bit portability problems on types that are also marked with the __w64 keyword.

```
/Wp64
```

Remarks

/Wp64 is off by default in the Visual C++ 32-bit compiler and on by default in the Visual C++ 64-bit compiler.

Variables of the following types are tested on a 32-bit operating system as if they were being used on a 64-bit operating system:

- int
- long
- pointer

If you regularly compile your application with a 64-bit compiler, you may want to disable **/Wp64** in your 32-bit compilations, as the 64-bit compiler will detect all issues. For more information about targeting a Windows 64-bit operating system, see 64-Bit Programming with Visual C++.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **General** property page.
4. Modify the **Detect 64-bit Portability Issues** property.

To set this compiler option programmatically

- See Detect64BitPortabilityProblems.

See Also

Reference

Compiler Options

Setting Compiler Options

/X (Ignore Standard Include Paths)

Prevents the compiler from searching for include files in directories specified in the PATH and INCLUDE environment variables.

```
/X
```

Remarks

You can use this option with the /I (Additional Include Directories) (**/I***directory*) option.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Preprocessor** property page.
4. Modify the **Ignore Standard Include Path** property.

To set this compiler option programmatically

- See IgnoreStandardIncludePath.

Example

In the following command, `/X` tells the compiler to ignore locations specified by the PATH and INCLUDE environment variables, and `/I` specifies the directory in which to look for include files:

```
CL /X /I \ALT\INCLUDE MAIN.C
```

See Also

Reference

Compiler Options

Setting Compiler Options

/Y (Precompiled Headers)

The following compiler options affect the generation and use of precompiled headers:

- /Y- (Ignore Precompiled Header Options)
- /Yc (Create Precompiled Header File)
- /Yd (Place Debug Information in Object File)
- /Yl (Inject PCH Reference for Debug Library)
- /Yu (Use Precompiled Header File)

For details on working with precompiled headers, see Creating Precompiled Header Files.

See Also
Reference
Compiler Options
Setting Compiler Options

/Y- (Ignore Precompiled Header Options)

Causes all other **/Y** compiler options to be ignored (and cannot itself be overridden).

```
/Y-
```

Remarks

For more information on precompiled headers, see:

- /Y (Precompiled Headers)
- Creating Precompiled Header Files

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also
Reference
Compiler Options
Setting Compiler Options

/Yc (Create Precompiled Header File)

Instructs the compiler to create a precompiled header (.pch) file that represents the state of compilation at a certain point.

```
/Yc[filename]
```

Arguments

filename

Specifies a header (.h) file. When this argument is used, the compiler compiles all code up to and including the .h file.

Remarks

When **/Yc** is specified without an argument, the compiler compiles all code up to the end of the base source file, or to the point in the base file where a hdrstop occurs. The resulting .pch file has the same base name as your base source file unless you specify a different file name using the **hdrstop** pragma or the **/Fp** option.

The precompiled code is saved in a file with a name created from the base name of the file specified with the **/Yc** option and a .pch extension. You can also use the /Fp (Name .pch File) option to specify a name for the precompiled header file.

If you use **/Yc***filename*, the compiler compiles all code up to and including the specified file for subsequent use with the **/Yu** option.

If the options **/Yc***filename* and /Yu (Use Precompiled Header File)*filename* occur on the same command line and both reference, or imply, the same file name, **/Yc***filename* takes precedence. This feature simplifies the writing of makefiles.

For more information on precompiled headers, see:

- /Y (Precompiled Headers)
- Creating Precompiled Header Files

To set this compiler option in the Visual Studio development environment

1. Select a .cpp file. The .cpp file must #include the .h file that contains precompiled header information. The project's **/Yc** setting can be overridden at the file level.
2. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
3. Click the **C/C++** folder.
4. Click the **Precompiled Headers** property page.
5. Modify the **Create/Use PCH Through File** property or the **Create/Use Precompiled Header** property.

To set this compiler option programmatically

- See PrecompiledHeaderThrough and UsePrecompiledHeader.

Example

Consider the following code:

```
#include <afxwin.h>   // Include header for class library
#include "resource.h" // Include resource definitions
#include "myapp.h"    // Include information specific to this app
...
```

When this code is compiled with the command `CL /YcMYAPP.H PROG.CPP`, the compiler saves all the preprocessing for AFXWIN.h, RESOURCE.h, and MYAPP.h in a precompiled header file called MYAPP.pch.

See Also

Reference

Compiler Options

Setting Compiler Options

/Yd (Place Debug Information in Object File)

Paces complete debugging information in all object files created from a precompiled header (.pch) file when used with the /Yc and /Z7 options. Deprecated.

```
/Yd
```

Remarks

/Yd is deprecated in Visual C++ 2005; Visual C++ now supports multiple objects writing to a single .pdb file, use **/Zi** instead. For more information, see Deprecated Compiler Options.

Unless you need to distribute a library containing debugging information, use the /Zi option rather than **/Z7** and **/Yd**.

Storing complete debugging information in every .obj file is necessary only to distribute libraries that contain debugging information. It slows compilation and requires considerable disk space. When **/Yc** and **/Z7** are used without **/Yd**, the compiler stores common debugging information in the first .obj file created from the .pch file. The compiler does not insert this information into .obj files subsequently created from the .pch file; it inserts cross-references to the information. No matter how many .obj files use the .pch file, only one .obj file contains the common debugging information.

Although this default behavior results in faster build times and reduces disk-space demands, it is undesirable if a small change requires rebuilding the .obj file containing the common debugging information. In this case, the compiler must rebuild all .obj files containing cross-references to the original .obj file. Also, if a common .pch file is used by different projects, reliance on cross-references to a single .obj file is difficult.

For more information on precompiled headers, see:

- /Y (Precompiled Headers)
- Creating Precompiled Header Files

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

Examples

Suppose you have two base files, F.cpp and G.cpp, each containing these **#include** statements:

```
#include "windows.h"
#include "etc.h"
```

The following command creates the precompiled header file ETC.pch and the object file F.obj:

```
CL /YcETC.H /Z7 F.CPP
```

The object file F.obj includes type and symbol information for WINDOWS.h and ETC.h (and any other header files they include). Now you can use the precompiled header ETC.pch to compile the source file G.cpp:

```
CL /YuETC.H /Z7 G.CPP
```

The object file G.obj does not include the debugging information for the precompiled header but simply references that information in the F.obj file. Note that you must link with the F.obj file.

If your precompiled header was not compiled with **/Z7**, you can still use it in later compilations using **/Z7**. However, the debugging information is placed in the current object file, and local symbols for functions and types defined in the precompiled header are not available to the debugger.

See Also

Reference

Compiler Options

Setting Compiler Options

/Yl (Inject PCH Reference for Debug Library)

Used if creating a debugging library that uses precompiled headers and the build fails.

```
/Ylsymbol
```

Arguments
symbol

An arbitrary symbol to be stored in the object module.

Remarks

When you compile a module with **/Yc** and **/Yl***symbol*, the compiler creates a symbol similar to __@@_PchSym_@00@...@*symbol*, where the ellipsis (...) represents a linker-generated character string, and stores it in the object module. Any source file that you compile with this precompiled header refers to the specified symbol, which causes the linker to include the object module and its debugging information from the library.

Using this option, you may generate LNK1211. When you specify the /Yc (Create Precompiled Header File) and /Z7, /Zi, /ZI (Debug Information Format) options, the compiler creates a precompiled header file that contains debugging information. An error can occur when you store the precompiled header in a library, use the library to build an object module, and the source code does not refer to any of the functions the precompiled header file defines.

To solve the problem, specify **/Yl***symbol*, where *symbol* is the name of an arbitrary symbol in the library, when you create a precompiled header file that does not contain any function definitions. This option directs the compiler to store the debugging information in the precompiled header file.

For more information on precompiled headers, see:

- /Y (Precompiled Headers)
- Creating Precompiled Header Files

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also
Reference
Compiler Options
Setting Compiler Options

/Yu (Use Precompiled Header File)

Instructs the compiler to use an existing precompiled header (.pch) file in the current compilation.

```
/Yu[filename]
```

Arguments

filename

The name of a header file, which is included in the source file using an **#include** preprocessor directive.

Remarks

The name of the include file must be the same for both the **/Yc** option that creates the precompiled header and any subsequent **/Yu** option indicating use of the precompiled header.

For **/Yc**, *filename* specifies the point at which precompilation stops; the compiler precompiles all code though *filename* and names the resulting precompiled header using the base name of the include file and an extension of .pch.

The .pch file must have been created using **/Yc**.

The compiler treats all code occurring before the .h file as precompiled. It skips to just beyond the **#include** directive associated with the .h file, uses the code contained in the .pch file, and then compiles all code after *filename*.

On the command line, no space is allowed between **/Yu** and *filename*.

When you specify the **/Yu** option without a file name, your source program must contain a #pragma hdrstop pragma that specifies the file name of the precompiled header, .pch file. In this case, the compiler will use the precompiled header (.pch file) named by /Fp (Name .pch File). The compiler skips to the location of that pragma, restores the compiled state from the precompiled header file specified by the pragma, and then compiles only code that follows the pragma. If **#pragma hdrstop** does not specify a file name, the compiler looks for a file with a name derived from the base name of the source file with a .pch extension. You can also use the **/Fp** option to specify a different .pch file.

If you specify the **/Yu** option without a file name and fail to specify a **hdrstop** pragma, an error message is generated and the compilation is unsuccessful.

If the **/Yc***filename* and **/Yu***filename* options occur on the same command line and both reference the same file name, **/Yc***filename* takes precedence, precompiling all code up to and including the named file. This feature simplifies the writing of makefiles.

Because .pch files contain information about the machine environment as well as memory address information about the program, you should only use a pch file on the machine where it was created.

For more information on precompiled headers, see:

- /Y (Precompiled Headers)
- Creating Precompiled Header Files

To set this compiler option in the Visual Studio development environment

1. Specify /Yc (Create Precompiled Header File) on a .cpp file in your project.
2. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
3. Click the **C/C++** folder.
4. Click the **Precompiled Headers** property page.
5. Modify the **Create/Use PCH Through File** property or the **Create/Use Precompiled Header** property.

To set this compiler option programmatically

- See PrecompiledHeaderThrough and UsePrecompiledHeader.

Examples

If the following code:

```
#include <afxwin.h>   // Include header for class library
#include "resource.h" // Include resource definitions
#include "myapp.h"    // Include information specific to this app
...
```

is compiled with the command line `CL /YuMYAPP.H PROG.CPP`, the compiler does not process the three include statements but uses precompiled code from MYAPP.pch, thereby saving the time involved in preprocessing all three of the files (and any files they might include).

You can use the /Fp (Name .pch File) option with the **/Yu** option to specify the name of the .pch file if the name is different from either the file name argument to **/Yc** or the base name of the source file, as in the following:

```
CL /YuMYAPP.H /FpMYPCH.pch PROG.CPP
```

This command specifies a precompiled header file named MYPCH.pch. The compiler uses its contents to restore the precompiled state of all header files up to and including MYAPP.h. The compiler then compiles the code that occurs after the MYAPP.h **include** statement.

See Also

Reference

Compiler Options

Setting Compiler Options

/Z7, /Zi, /ZI (Debug Information Format)

Select the type of debugging information created for your program and whether this information is kept in object (.obj) files or in a program database (PDB).

```
/Z{7|i|I}
```

Remarks

The options are described in the following table.

None

Produces no debugging information, so compilation is faster.

/Z7

Produces an .obj file containing full symbolic debugging information for use with the debugger. The symbolic debugging information includes the names and types of variables, as well as functions and line numbers. No .pdb file is produced.

For distributors of third-party libraries, there is an advantage to not having a .pdb file. However, the .obj files for the precompiled headers are necessary during the link phase, and debugging. If there is only type information (and no code) in the .pch object files, you will also have to compile with /Yl (Inject PCH Reference for Debug Library).

/Zi

Produces a program database (PDB) that contains type information and symbolic debugging information for use with the debugger. The symbolic debugging information includes the names and types of variables, as well as functions and line numbers.

/Zi does not affect optimizations. However, **/Zi** does imply **/debug**; see /DEBUG (Generate Debug Info) for more information.

Type information is placed in the .pdb file, and not in the .obj file.

You can use /Gm (Enable Minimal Rebuild) with **/Zi**, whereas **/Gm** is not available when compiling with **/Z7**.

When compiling with **/Zi** and **/clr**, the DebuggableAttribute attribute will not be placed in the assembly metadata; you must specify it in source code, if you want it. This attribute can affect the runtime performance of the application. For more information about how the Debuggable attribute affects performance and how you can modify the performance impact, see Making an Image Easier to Debug.

/ZI

Produces a program database, as described above, in a format that supports the Edit and Continue feature. If you want to use Edit and Continue debugging, you must use this option. Because most optimizations are incompatible with Edit and Continue, using **/ZI** disables any `#pragma optimize` statements in your code.

/ZI causes /Gy (Enable Function-Level Linking) to be used in your compilation.

/ZI is not compatible with /clr (Common Language Runtime Compilation).

Note
/ZI is only available in the compiler targeting x86; this compiler option is not available in the compilers targeting x64 or Itanium Processor Family (IPF).

The compiler names the program database *project*.pdb. If you compile a file without a project, the compiler creates a database named VC*x*0.pdb., where *x* is the major version of Visual C++ in use. The compiler embeds the name of the PDB in each .obj file created using this option, pointing the debugger to the location of symbolic and line-number information. When you use this option, your .obj files will be smaller, because debugging information is stored in the .pdb file rather than in .obj files.

If you create a library from objects that were compiled using this option, the associated .pdb file must be available when the library is linked to a program. Thus, if you distribute the library, you must distribute the PDB.

To create a library that contains debugging information without using .pdb files, you must select the compiler's C 7.0-

Compatible (**/Z7**) option. If you use the precompiled headers options, debugging information for both the precompiled header and the rest of the source code is placed in the PDB. The **/Yd** option is ignored when the Program Database option is specified.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **General** property page.
4. Modify the **Debug Information Format** property.

To set this compiler option programmatically

- See DebugInformationFormat.

See Also
Reference
Compiler Options
Setting Compiler Options

/Za, /Ze (Disable Language Extensions)

/Za flags language constructs not compatible with either ANSI C++ or ANSI C as errors. **/Ze** enables Microsoft extensions.

```
/Za
/Ze
```

Remarks

The Visual C++ compiler offers a number of features beyond those specified in either the ANSI C or ANSI C++ standards. These features are known collectively as Microsoft extensions to C and C++. These extensions are available when the **/Ze** option, the default, is specified and are not available when the s option is specified. See Microsoft Extensions to C and C++ for more information.

Disable language extensions if you plan to port your program to other environments. The compiler treats extended keywords as simple identifiers, disables the other Microsoft extensions, and automatically defines the __STDC__ predefined macro for C programs.

Other compiler options used with **s** can how the compiler ensures ANSI conformance. For example, **s** and /fp (Specify Floating-Point Behavior) may result in unexpected behavior.

See /Zc for ways to get standard behavior with **s**.

/Ze is deprecated in Visual C++ 2005; it is the default. For more information, see Deprecated Compiler Options.

For more information about conformance issues with Visual C++, see Compatibility and Compliance Issues in Visual C++.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Language** property page.
4. Modify the **Disable Language Extensions** property.

To set this compiler option programmatically

- See DisableLanguageExtensions.

See Also
Reference
Compiler Options
Setting Compiler Options

Microsoft Extensions to C and C++

The following are Visual C++ extensions to the ANSI C and ANSI C++ standards:

Keywords

Microsoft extends the C++ language with several additional keywords. For a complete list, see C++ Keywords in the *C++ Language Reference*. Keywords with two leading underscores are Microsoft extensions.

Out of Class Definition of static const Integral (or enum) Members

Under the standard (**/Za**), you need to make an out-of-class definition for data members. For example,

```
class CMyClass  {
   static const int max = 5;
   int m_array[max];
}
...
const int CMyClass::max;   // out of class definition
```

Under **/Ze**, the out-of-class definition is optional for static, const integral, and const enum data members. Only integrals and enums that are static and const can have initializers inside a class; the initializing expression must be a const expression.

To avoid errors when an out-of-class definition is provided (when the out-of-class definition is provided in a header file and the header file is included in multiple source files), you should use selectany. For example:

```
__declspec(selectany) const int CMyClass::max = 5;
```

Casts

The compiler supports the following two non-ANSI casts:

- Use of non-ANSI casts to produce l-values:

  ```
  char *p;
  (( int * ) p )++;
  ```

  The preceding example could be rewritten to conform with the ANSI C standard as follows:

  ```
  p = ( char * )(( int * )p + 1 );
  ```

- Non-ANSI casting of a function pointer to a data pointer:

  ```
  int ( * pfunc ) ();
  int *pdata;
  pdata = ( int * ) pfunc;
  ```

  To perform the same cast while maintaining ANSI compatibility, you must cast the function pointer to an `int` before casting it to a data pointer:

  ```
  pdata = ( int * ) (int) pfunc;
  ```

Variable-Length Argument Lists

The compiler supports use of a function declarator that specifies a variable number of arguments, followed by a function definition that provides a type instead:

```
void myfunc( int x, ... );
void myfunc( int x, char * c )
```

```
{ }
```

Single-Line Comments

The C compiler supports single-line comments, which are introduced with two forward slash (//) characters:

```
// This is a single-line comment.
```

Scope

The C compiler supports the following scope-related features:

- Redefinitions of extern items as static:

```
extern int clip();
static int clip()
{}
```

- Use of benign typedef redefinitions within the same scope:

```
typedef int INT;
typedef int INT;
```

- Function declarators have file scope:

```
void func1()
{
    extern int func2( double );
}
int main( void )
{
    func2( 4 );    //  /Ze passes 4 as type double
}                  //  /Za passes 4 as type int
```

- Use of block-scope variables initialized with nonconstant expressions:

```
int clip( int );
int bar( int );
int main( void )
{
    int array[2] = { clip( 2 ), bar( 4 ) };
}
int clip( int x )
{
    return x;
}
int bar( int x )
{
    return x;
}
```

Data Declarations and Definitions

The C compiler supports the following data declaration and definition features:

- Mixed character and string constants in an initializer:

```
char arr[5] = {'a', 'b', "cde"};
```

- Bit fields with base types other than **unsigned int** or **signed int**.
- Declarators without either a storage class or a type:

```
x;
int main( void )
{
    x = 1;
}
```

- Unsized arrays as the last field in structures and unions:

```
struct zero
{
    char *c;
    int zarray[];
};
```

- Unnamed (anonymous) structures:

```
struct
{
    int i;
    char *s;
};
```

- Unnamed (anonymous) unions:

```
union
{
    int i;
    float fl;
};
```

- Unnamed members:

```
struct s
{
   unsigned int flag : 1;
   unsigned int : 31;
}
```

Intrinsic Floating-Point Functions

The compiler supports inline generation **x86 Specific >** of the **atan**, **atan2**, **cos**, **exp**, **log**, **log10**, **sin**, **sqrt**, and **tan** functions **END x86 Specific** when **/Oi** is specified. For C, ANSI conformance is lost when these intrinsics are used, because they do not set the *errno* variable.

Passing a Non-Const Pointer Parameter to a Function that Expects a Reference to a Const Pointer Parameter

This is an extension to C++. The following code will compile with **/Ze**:

```
typedef   int   T;

const T  acT = 9;      // A constant of type 'T'
const T* pcT = &acT;   // A pointer to a constant of type 'T'

void func2 ( const T*& rpcT )   // A reference to a pointer to a constant of type 'T'
```

```
{
   rpcT = pcT;
}

T*   pT;                // A pointer to a 'T'

void func ()
{
   func2 ( pT );      // Should be an error, but isn't detected
   *pT   = 7;         // Invalidly overwrites the constant 'acT'
}
```

ISO646.H Not Enabled

Under **/Ze**, you have to include iso646.h if you want to use text forms of the following operators:

- && (and)
- &= (and_eq)
- & (bitand)
- | (bitor)
- ~ (compl)
- ! (not)
- != (not_eq)
- || (or)
- |= (or_eq)
- ^ (xor)
- ^= (xor_eq)

Address of string literal has type const char [], not const char (*) []

The following example will output char const (*)[4] under **/Za**, but char const [4] under **/Ze**.

```
#include <stdio.h>
#include <typeinfo>

int main()
{
    printf_s("%s\n", typeid(&"abc").name());
}
```

See Also

Reference

/Za, /Ze (Disable Language Extensions)

Compiler Options

Setting Compiler Options

/Zc (Conformance)

The **/Zc** compiler option lets you specify standard behavior with /Za, /Ze (Disable Language Extensions). The following is a list of the **/Zc** compiler options:

- /Zc:forScope
- /Zc:wchar_t

For more information about conformance issues with Visual C++, see Compatibility and Compliance Issues in Visual C++.

See Also
Reference
Compiler Options
Setting Compiler Options

/Zc:forScope (Force Conformance in for Loop Scope)

Used to implement standard C++ behavior for The for Statement loops with Microsoft extensions (/Za, /Ze (Disable Language Extensions)). **/Zc:forScope** is on by default.

```
/Zc:forScope[-]
```

Remarks

Standard behavior is to let a **for** loop's initializer go out of scope after the **for** loop. Under /Za, /Ze (Disable Language Extensions), the **for** loop's initializer remains in scope until the local scope ends.

The following code will compile under **/Ze** but not under **/Za**:

```
// zc_forScope.cpp
// compile with: /Zc:forScope- /Za
// C2065 expected
int main() {
   // Uncomment the following line to resolve.
   // int i;
   for (int i =0; i < 1; i++)
      ;
   i = 20;   // i has already gone out of scope under /Za
}
```

If you use **/Zc:forScope-**, you will get a warning (off by default) if a variable is in scope because of a declaration that was made in a previous scope. To demonstrate this, remove the `//` characters in the above code to declare `int i`.

You can modify the run-time behavior of **/Zc:forScope** with the conform pragma.

If you use **/Zc:forScope-** in a project with an existing .pch file, **/Zc:forScope-** is ignored (with a warning) and compilation continues with the existing .pch files. If you want a new .pch file generated, use /Yc (Create Precompiled Header File).

For more information about conformance issues with Visual C++, see Compatibility and Compliance Issues in Visual C++.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Modifying Project Settings.
2. Click the **C/C++** folder.
3. Click the **Language** property page.
4. Modify the **Force Conformance in For Loop Scope** property.

To set this compiler option programmatically

- See ForceConformanceInForLoopScope.

See Also

Reference

/Zc (Conformance)

/Za, /Ze (Disable Language Extensions)

/Zc:wchar_t (wchar_t Is Native Type)

When specified, the type **wchar_t** becomes a native type that maps to **__wchar_t** in the same way that **short** maps to **__int16**. **/Zc:wchar_t** is on by default.

```
/Zc:wchar_t[-]
```

Remarks

If **/Zc:wchar_t-** is specified, the compiler requires you to either define **wchar_t** or to include one of the many header files that defines it (for example, wchar.h). Typically, **wchar_t** is defined as an **unsigned short**.

With **/Zc:wchar_t**, the compiler recognizes **wchar_t** as a native type. See Data Type Ranges for more information about **wchar_t**.

__wchar_t is always available.

By providing overloads for both the **unsigned short** and **__wchar_t** variations of **wchar_t**, you can create libraries that can easily be linked with code compiled with or without **/Zc:wchar_t** and avoid the need to provide two different builds of the library (one with and one without **/Zc:wchar_t** enabled).

When **/Zc:wchar_t** is specified, **_WCHAR_T_DEFINED** and **_NATIVE_WCHAR_T_DEFINED** symbols are defined; see Predefined Macros for more information.

Explicit references to comsupp.lib, either from the comment pragma or via the command line, should be changed to now use either comsuppw.lib or comsuppwd.lib, as **/Zc:wchar_t** is now on by default. comsupp.lib should still be used when compiling with **/Zc:wchar_t-**.

For more information about conformance issues with Visual C++, see Compatibility and Compliance Issues in Visual C++.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Modifying Project Settings.
2. Click the **C/C++** folder.
3. Click the **Language** property page.
4. Modify the **Treat wchar_t as Built-in Type** property.

To set this compiler option programmatically

- See TreatWChar_tAsBuiltInType.

See Also

Reference

/Zc (Conformance)

/Zg (Generate Function Prototypes)

Creates a function prototype for each function defined in the source file, but does not compile the source file.

```
/Zg
```

Remarks

The function prototype includes the function return type and an argument type list. The argument type list is created from the types of the formal parameters of the function. Any function prototypes already present in the source file are ignored.

The list of prototypes is written to standard output. You may find this list helpful to verify that actual arguments and formal parameters of a function are compatible. You can save the list by redirecting standard output to a file. Then you can use **#include** to make the list of function prototypes a part of your source file. Doing so causes the compiler to perform argument type checking.

If you use the **/Zg** option and your program contains formal parameters that have struct, enum, or union type (or pointers to such types), the declaration of each struct, enum, or union type must have a tag (name). In the following sample, the tag name is `MyStruct`.

```
// Zg_compiler_option.c
// compile with: /Zg
typedef struct MyStruct { int i; } T2;
void f2(T2 * t) {}
```

/Zg is deprecated in Visual C++ 2005. The Visual C++ compiler plans to remove support for older, C-style code. For more information, see Deprecated Compiler Options.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

Compiler Options

Setting Compiler Options

/Zl (Omit Default Library Name)

Omits the default C runtime library name from the .obj file. By default, the compiler puts the name of the library into the .obj file to direct the linker to the correct library.

```
/Zl
```

Remarks

For more information on the default library, see Use Run-Time Library.

You can use **/Zl** to compile .obj files you plan to put into a library. Although omitting the library name saves only a small amount of space for a single .obj file, the total space saved is significant in a library that contains many object modules.

This option is an advanced option. Setting this option removes certain C Runtime library support that may be required by your application, resulting in link-time errors if your application depends on this support. If you use this option you must provide the required components in some other way.

Use /NODEFAULTLIB (Ignore Libraries). to direct the linker to ignore library references in all .obj files.

For more information, see C Run-Time Libraries.

When compiling with **/Zl**, **_VC_NODEFAULTLIB** is defined. For example:

```
// vc_nodefaultlib.cpp
// compile with: /Zl
void Test() {
   #ifdef _VC_NODEFAULTLIB
      int i;
   #endif

   int i;   // C2086
}
```

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Advanced** property page.
4. Modify the **Omit Default Library Names** property.

To set this compiler option programmatically

- See OmitDefaultLibName.

See Also

Reference

Compiler Options

Setting Compiler Options

/Zm (Specify Precompiled Header Memory Allocation Limit)

Determines the compiler's memory allocation limit for the precompiled header.

```
/Zmnumber
```

Arguments

number

A scaling factor that determines the compiler's memory allocation limit for the precompiled header.

Remarks

The *number* argument is a scaling factor with a default value of 100, which specifies a memory allocation of 50 MB. The maximum value is 2000. The following table shows how number modifies the memory allocation.

Value	Memory allocation
10	5.0 MB
100	50 MB
200	100 MB
1000	500 MB
2000	1000 MB

In prior versions of Visual C++, the compiler used a number of discrete heaps, each of which has a finite limit. The compiler now dynamically grows the heaps as necessary, only requiring a fixed size for the memory allocated to the precompiled header. Exceeding precompiled header's heap size limits occurs only in rare circumstances involving very large or very complex programs. Should your program exceed these limits, use **/Zm** to scale the total size of all of the limits.

In most cases, use of this compiler option is not necessary. Use it if compiling your program causes an error message, and where the error message reports what the value of **/Zm** should be.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

Compiler Options

Setting Compiler Options

/Zp (Struct Member Alignment)

Controls how the members of a structure are packed into memory and specifies the same packing for all structures in a module.

```
/Zp[1|2|4|8|16]
```

Remarks

When you specify this option, each structure member after the first is stored on either the size of the member type or *n*-byte boundaries (where *n* is 1, 2, 4, 8, or 16), whichever is smaller.

The available values are described in the following table.

1

Packs structures on 1-byte boundaries. Same as **/Zp**.

2

Packs structures on 2-byte boundaries.

4

Packs structures on 4-byte boundaries.

8

Packs structures on 8-byte boundaries (default).

16

Packs structures on 16-byte boundaries.

You should not use this option unless you have specific alignment requirements.

You can also use pack to control structure packing. For more information about alignment, see:

- align (C++)
- __alignof Operator
- __unaligned
- Structure Alignment Examples (x64 specific)

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Code Generation** property page.
4. Modify the **Struct Member Alignment** property.

To set this compiler option programmatically

- See StructMemberAlignment.

See Also

Reference

Compiler Options

Setting Compiler Options

/Zs (Syntax Check Only)

Tells the compiler to check only the syntax of the source files on the command line.

```
/Zs
```

Remarks

When using this option, no output files are created, and error messages are written to standard output.

The **/Zs** option provides a quick way to find and correct syntax errors before you compile and link a source file.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

Compiler Options

Setting Compiler Options

/Zx (Debug Optimized Itanium Code)

Disables certain optimizations, such as if-conversions, software pipelining, and global scheduling optimizations, that make it difficult to debug optimized code.

```
/Zx
```

Remarks

This compiler option is only available in the Itanium cross compiler or in the Itanium native compiler.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Click the **C/C++** folder.
3. Click the **Command Line** property page.
4. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See Also

Reference

Compiler Options

Setting Compiler Options

Creating Precompiled Header Files

The Microsoft C and C++ compilers provide options for precompiling any C or C++ code, including inline code. Using this performance feature, you can compile a stable body of code, store the compiled state of the code in a file, and, during subsequent compilations, combine the precompiled code with code that is still under development. Each subsequent compilation is faster because the stable code does not need to be recompiled.

This section covers the following precompiled header issues:

- When to Precompile Source Code
- Two Choices for Precompiling Code
- Precompiled Header Consistency Rules
- Using Precompiled Headers in a Project

For reference information on the compiler options related to precompiled headers, see /Y (Precompiled Headers).

See Also

Reference

Compiler Options

Other Resources

C/C++ Building Reference

When to Precompile Source Code

Precompiled code is useful during the development cycle to reduce compilation time, especially if:

- You always use a large body of code that changes infrequently.
- Your program comprises multiple modules, all of which use a standard set of include files and the same compilation options. In this case, all include files can be precompiled into one precompiled header.

The first compilation — the one that creates the precompiled header file — takes a bit longer than subsequent compilations. Subsequent compilations can proceed more quickly by including the precompiled code.

You can precompile both C and C++ programs. In C++ programming, it is common practice to separate class interface information into header files. These header files can later be included in programs that use the class. By precompiling these headers, you can reduce the time a program takes to compile.

Note
Although you can use only one precompiled header (.pch) file per source file, you can use multiple .pch files in a project.

See Also

Reference

Creating Precompiled Header Files

Two Choices for Precompiling Code

With Visual C++, you can precompile any C or C++ code; you are not limited to precompiling only header files.

Precompiling requires planning, but it offers significantly faster compilations if you precompile source code other than simple header files.

Precompile code when you know that your source files use common sets of header files but don't include them in the same order, or when you want to include source code in your precompilation.

The precompiled-header options are /Yc (Create Precompiled Header File) and /Yu (Use Precompiled Header File). Use **/Yc** to create a precompiled header. When used with the optional **hdrstop** pragma, **/Yc** lets you precompile both header files and source code. Select **/Yu** to use an existing precompiled header in the existing compilation. You can also use **/Fp** with the **/Yc** and **/Yu** options to provide an alternative name for the precompiled header.

The compiler option reference topics for **/Yu** and **/Yc** discuss how to access this functionality in the development environment.

Building a PCH File

You can use one PCH to build another in a fairly simple way.

```
cl -c -Yc"stuff.h" -Fplevel1.pch level1.cpp
cl -c -Yu"stuff.h" -Fplevel1.pch -Yc level2.cpp
```

The `level2.cpp` file looks like this:

```
#include "stuff.h"
#include "morestuff.h"
#pragma hdrstop("level2.pch")
```

The following command will build `level2.pch`, which can be used in further compiles:

```
cl -c -Yu"morestuff.h" -Fplevel2.pch mysource.cpp
```

Note that you only need to put the `#pragma hdrstop` directive in the file that builds the second PCH; you don't have to put it in all .cpp files that use the PCH. Files that use the PCH can name the `morestuff.h` file in the command line (so you don't have to edit all your files to use this). Either method of precompiling code — automatically or manually — stores the resulting precompiled code in a precompiled header.

More Information

- When to Precompile Source Code
- Precompiled Header Consistency Rules
- Using Precompiled Headers in a Project

For further examples using precompiled headers, see the makefiles used to build the sample programs that ship with the Microsoft Foundation Class Library.

See Also
Reference
Creating Precompiled Header Files

Precompiled Header Consistency Rules

This section discusses guidelines that will help you use precompiled headers more efficiently:

- Consistency Rules for Per-File Use of Precompiled Headers
- Consistency Rules for /Yc and /Yu

Because PCH files contain information about the machine environment as well as memory address information about the program, you should only use a PCH file on the machine where it was created.

See Also

Reference

Creating Precompiled Header Files

Consistency Rules for Per-File Use of Precompiled Headers

The /Yu compiler option lets you specify which precompiled header (PCH) file to use.

When you use a PCH, the compiler assumes the same compilation environment — using consistent compiler options, pragmas, and so on — that was in effect when you created the PCH, unless you specify otherwise. If the compiler detects an inconsistency, it issues a warning and identifies the inconsistency where possible. Such warnings do not necessarily indicate a problem with the PCH; they simply warn you of possible conflicts. Consistency requirements for PCHs are described in the following sections.

Compiler Option Consistency

The following compiler options can trigger an inconsistency warning when using a PCH:

- Macros created using the Preprocessor (/D) option must be the same between the compilation that created the PCH and the current compilation. The state of defined constants is not checked, but unpredictable results can occur if these change.
- PCHs do not work with the /E and /EP options.
- PCHs must be created using either the Generate Browse Info (/FR) option or the Exclude Local Variables (/Fr) option before subsequent compilations that use the PCH can use these options.

C 7.0-Compatible (/Z7)

If this option is in effect when the PCH is created, subsequent compilations that use the PCH can use the debugging information.

If the C 7.0-Compatible (/Z7) option is not in effect when the PCH is created, subsequent compilations that use the PCH and /Z7 trigger a warning. The debugging information is placed in the current .obj file, and local symbols defined in the PCH are not available to the debugger.

Include Path Consistency

A PCH does not contain information about the include path that was in effect when it was created. When you use a .pch file, the compiler always uses the include path specified in the current compilation.

Source File Consistency

When you specify the Use Precompiled Header File (/Yu) option, the compiler ignores all preprocessor directives (including pragmas) that appear in the source code that will be precompiled. The compilation specified by such preprocessor directives must be the same as the compilation used for the Create Precompiled Header File (/Yc) option.

Pragma Consistency

Pragmas processed during the creation of a PCH usually affect the file with which the PCH is subsequently used. The comment and message pragmas do not affect the remainder of the compilation.

The following pragmas are retained as part of a PCH, and affect the remainder of a compilation that uses the PCH.

alloc_text	include_alias	pack
auto_inline	init_seg	pointers_to_members
check_stack	inline_depth	setlocale
code_seg	inline_recursion	vtordisp
data_seg	intrinsic	warning
function	optimize	

See Also

Reference

Precompiled Header Consistency Rules

/Yu (Use Precompiled Header File)
Compiler Options

Consistency Rules for /Yc and /Yu

When you use a precompiled header created using /Yc or /Yu, the compiler compares the current compilation environment to the one that existed when you created the .pch file. Be sure to specify an environment consistent with the previous one (using consistent compiler options, pragmas, and so on) for the current compilation. If the compiler detects an inconsistency, it issues a warning and identifies the inconsistency where possible. Such warnings don't necessarily indicate a problem with the .pch file; they simply warn you of possible conflicts. The following sections explain the consistency requirements for precompiled headers.

Compiler Option Consistency

The following table lists compiler options that might trigger an inconsistency warning when using a precompiled header.

Option	Name	Rule
/D	Define constants and macros	Must be the same between the compilation that created the precompiled header and the current compilation. The state of defined constants is not checked, but unpredictable results can occur if your files depend on the values of the changed constants.
/E or /EP	Copy preprocessor output to standard output	Precompiled headers do not work with the /E or /EP option.
/Fr or /FR	Generate Microsoft Source Browser information	For the /Fr and /FR options to be valid with the /Yu option, they must also have been in effect when the precompiled header was created. Subsequent compilations that use the precompiled header also generate Source Browser information. Browser information is placed in a single .sbr file and is referenced by other files in the same manner as CodeView information. You cannot override the placement of Source Browser information.
/GA, /GD, /GE, /Gw, or /GW	Windows protocol options	Must be the same between the compilation that created the precompiled header and the current compilation. If these options differ, a warning message results.
/Zi	Generate complete debugging information	If this option is in effect when the precompiled header is created, subsequent compilations that use the precompilation can use that debugging information. If /Zi is not in effect when the precompiled header is created, subsequent compilations that use the precompilation and the /Zi option trigger a warning. The debugging information is placed in the current object file, and local symbols defined in the precompiled header are not available to the debugger.

Note
The precompiled header facility is intended for use only with a file with C and C++ source files.

Include Path Consistency

A precompiled header created with /Yc does not contain information about the include path that was in effect when you created the .pch file. When you use a .pch file, the compiler always uses the include path specified in the current compilation.

Source File Consistency

When you use a precompiled header, the compiler ignores all preprocessor directives (including pragmas) that appear before the hdrstop pragma. The compilation specified by such preprocessor directives must be the same as the compilation used to create the precompiled header file.

Pragma Consistency

Pragmas processed during the compilation of a precompiled header usually affect the file in which the precompiled header is

subsequently used. The following pragmas affect only the code within the .pch file; they do not affect code that subsequently uses the .pch file:

Comment	page	subtitle
Linesize	pagesize	Title
Message	skip	

The following pragmas are retained as part of a precompiled header, and affect the remainder of a compilation that uses the precompiled header:

alloc_text	function	optimize
auto_inline	inline_depth	Pack
check_pointer	inline_recursion	same_seg
check_stack	intrinsic	warning
code_seg	loop_opt	
data_seg	native_caller	

See Also

Reference

Precompiled Header Consistency Rules

/Yc (Create Precompiled Header File)

/Yu (Use Precompiled Header File)

Using Precompiled Headers in a Project

Previous sections present an overview of precompiled headers: /Yc and /Yu, the /Fp option, and the hdrstop pragma. This section describes a method for using the manual precompiled-header options in a project; it ends with an example makefile and the code that it manages.

For another approach to using the manual precompiled-header options in a project, study one of the makefiles located in the MFC\SRC directory that is created during the default setup of Visual C++. These makefiles take a similar approach to the one presented in this section but make greater use of Microsoft Program Maintenance Utility (NMAKE) macros, and offer greater control of the build process.

This section contains the following topics:

- PCH Files in the Build Process
- Sample Makefile for PCH
- Example Code for PCH

See Also

Reference

Creating Precompiled Header Files

PCH Files in the Build Process

The code base of a software project is usually contained in multiple C or C++ source files, object files, libraries, and header files. Typically, a makefile coordinates the combination of these elements into an executable file. The following figure shows the structure of a makefile that uses a precompiled header file. The NMAKE macro names and the file names in this diagram are consistent with those in the example code found in Sample Makefile for PCH and Example Code for PCH.

The figure uses three diagrammatic devices to show the flow of the build process. Named rectangles represent each file or macro; the three macros represent one or more files. Shaded areas represent each compile or link action. Arrows show which files and macros are combined during the compilation or linking process.

Structure of a Makefile That Uses a Precompiled Header File

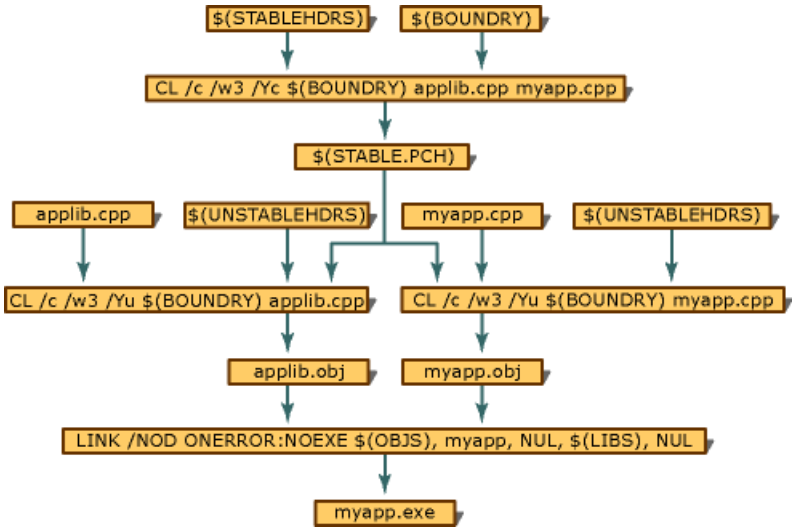

Beginning at the top of the diagram, both STABLEHDRS and BOUNDRY are NMAKE macros in which you list files not likely to need recompilation. These files are compiled using the command string

```
CL /c /W3 /Yc$(BOUNDRY) applib.cpp myapp.cpp
```

only if the precompiled header file (STABLE.pch) does not exist or if you make changes to the files listed in the two macros. In either case, the precompiled header file will contain code only from the files listed in the STABLEHDRS macro. List the last file you want precompiled in the BOUNDRY macro.

The files you list in these macros can be either header files or C or C++ source files. (A single .pch file cannot be used with both C and C++ modules.) Note that you can use the **hdrstop** macro to stop precompilation at some point within the BOUNDRY file. See hdrstop for more information.

Continuing down the diagram, APPLIB.obj represents the support code used in your final application. It is created from APPLIB.cpp, the files listed in the UNSTABLEHDRS macro, and precompiled code from the precompiled header.

MYAPP.obj represents your final application. It is created from MYAPP.cpp, the files listed in the UNSTABLEHDRS macro, and precompiled code from the precompiled header.

Finally, the executable file (MYAPP.EXE) is created by linking the files listed in the OBJS macro (APPLIB.obj and MYAPP.obj).

For a further discussion of the figure, see:

- Sample Makefile for PCH
- Example Code for PCH

See Also
Reference
Using Precompiled Headers in a Project

Sample Makefile for PCH

The following makefile uses macros and an !IF, !ELSE, !ENDIF flow-of-control command structure to simplify its adaptation to your project.

```
# Makefile : Illustrates the effective use of precompiled
#            headers in a project
# Usage:     NMAKE option
# option:    DEBUG=[0|1]
#            (DEBUG not defined is equivalent to DEBUG=0)
#
OBJS = myapp.obj applib.obj
# List all stable header files in the STABLEHDRS macro.
STABLEHDRS = stable.h another.h
# List the final header file to be precompiled here:
BOUNDRY = stable.h
# List header files under development here:
UNSTABLEHDRS = unstable.h
# List all compiler options common to both debug and final
# versions of your code here:
CLFLAGS = /c /W3
# List all linker options common to both debug and final
# versions of your code here:
LINKFLAGS = /NOD /ONERROR:NOEXE
!IF "$(DEBUG)" == "1"
CLFLAGS   = /D_DEBUG $(CLFLAGS) /Od /Zi /f
LINKFLAGS = $(LINKFLAGS) /COD
LIBS      = slibce
!ELSE
CLFLAGS   = $(CLFLAGS) /Oselg /Gs
LINKFLAGS = $(LINKFLAGS)
LIBS      = slibce
!ENDIF
myapp.exe: $(OBJS)
    link $(LINKFLAGS) @<<
$(OBJS), myapp, NUL, $(LIBS), NUL;
<<
# Compile myapp
myapp.obj  : myapp.cpp $(UNSTABLEHDRS)  stable.pch
    $(CPP) $(CLFLAGS) /Yu$(BOUNDRY)    myapp.cpp
# Compile applib
applib.obj : applib.cpp $(UNSTABLEHDRS) stable.pch
    $(CPP) $(CLFLAGS) /Yu$(BOUNDRY)    applib.cpp
# Compile headers
stable.pch : $(STABLEHDRS)
    $(CPP) $(CLFLAGS) /Yc$(BOUNDRY)    applib.cpp myapp.cpp
```

Aside from the STABLEHDRS, BOUNDRY, and UNSTABLEHDRS macros shown in the figure "Structure of a Makefile That Uses a Precompiled Header File" in PCH Files in the Build Process, this makefile provides a CLFLAGS macro and a LINKFLAGS macro. You must use these macros to list compiler and linker options that apply whether you build a debug or final version of the application's executable file. There is also a LIBS macro where you list the libraries your project requires.

The makefile also uses !IF, !ELSE, !ENDIF to detect whether you define a DEBUG symbol on the NMAKE command line:

```
NMAKE DEBUG=[1|0]
```

This feature makes it possible for you to use the same makefile during development and for the final versions of your program — use DEBUG=0 for the final versions. The following command lines are equivalent:

```
NMAKE
NMAKE DEBUG=0
```

For more information on makefiles, see NMAKE Reference. Also see Compiler Options and the Linker Options.

See Also

Reference

Using Precompiled Headers in a Project

Example Code for PCH

The following examples are used in the makefile described in PCH Files in the Build Process and Sample Makefile for PCH. Note that the comments contain important information.

```
// ANOTHER.H : Contains the interface to code that is not
//             likely to change.
//
#ifndef --ANOTHER_H
#define --ANOTHER_H
#include<iostream>
void savemoretime( void );
#endif // --ANOTHER_H

// STABLE.H : Contains the interface to code that is not likely
//            to change. List code that is likely to change
//            in the makefile's STABLEHDRS macro.
//
#ifndef --STABLE_H
#define --STABLE_H
#include<iostream>
void savetime( void );
#endif // --STABLE_H

// UNSTABLE.H : Contains the interface to code that is
//              likely to change. As the code in a header
//              file becomes stable, remove the header file
//              from the makefile's UNSTABLEHDR macro and list
//              it in the STABLEHDRS macro.
//
#ifndef --UNSTABLE_H
#define --UNSTABLE_H
#include<iostream.h>
void notstable( void );
#endif // --UNSTABLE_H

// APPLIB.CPP : This file contains the code that implements
//              the interface code declared in the header
//              files STABLE.H, ANOTHER.H, and UNSTABLE.H.
//
#include"another.h"
#include"stable.h"
#include"unstable.h"
// The following code represents code that is deemed stable and
// not likely to change. The associated interface code is
// precompiled. In this example, the header files STABLE.H and
// ANOTHER.H are precompiled.
void savetime( void )
    { cout << "Why recompile stable code?\n"; }
void savemoretime( void )
    { cout << "Why, indeed?\n\n"; }
// The following code represents code that is still under
// development. The associated header file is not precompiled.
void notstable( void )
    { cout << "Unstable code requires"
            << " frequent recompilation.\n"; }

// MYAPP.CPP : Sample application
//             All precompiled code other than the file listed
//             in the makefile's BOUNDRY macro (stable.h in
//             this example) must be included before the file
//             listed in the BOUNDRY macro. Unstable code must
//             be included after the precompiled code.
//
#include"another.h"
```

```
#include"stable.h"
#include"unstable.h"
int main( void )
{
    savetime();
    savemoretime();
    notstable();
}
```

See Also

Reference

Using Precompiled Headers in a Project

Unicode Support in the Compiler and Linker

This topic describes Unicode support in the Visual C++ build tools.

Filenames

Filenames specified on the command line, and in compiler directives (such as #include) may now contain Unicode characters. In previous versions of Visual C++, only ANSI filenames were supported. Non-ASCII characters should be avoided in output file names so an executable can run on localized versions of Windows 98 and Windows Me.

Source code files

Unicode characters are now supported in identifiers, macros, string and character literals, and in comments. Universal character names are also now supported.

Unicode can be input into a source code file in the following encodings:

- UTF-16 little endian with or without byte order mark (BOM)
- UTF-16 big endian with or without BOM
- UTF-8 with BOM

Output

During compilation, compiler outputs diagnostics to the console in UTF-16. The characters that can be displayed at your console depend on the console window properties. Compiler output redirected to a file is in the current ANSI console codepage.

Linker response files and .DEF files

Response files and DEF files can be either UTF-16 with a Byte Order Mark or ANSI. Previously only ANSI was supported.

.asm and .cod dumps

.asm and .cod dumps are in ANSI by default for compatibility with MASM. Use /FAu to output UTF-8. Note that if you specify /FAs, the intermingled source will just be directly printed and may look garbled, for example if source code is UTF-8 and you didn't specify /FAsu.

You can enable Unicode file names in the development environment (see How to: Open Project Property Pages) by selecting the appropriate tool and by selecting the **Enable Unicode Response Files** property, which is enabled by default. One reason you might change this default is if you modify your development environment to use a compiler that does not have Unicode support.

See Also

Other Resources

Building on the Command Line

Linking

For information on using the linker, see the following sections:

- Setting Linker Options
- Linker Options
- Module-Definition (.def) Files
- Linker Support for Delay-Loaded DLLs

See Also

Other Resources

C/C++ Building Reference

Setting Linker Options

Linker options can be set inside or outside of the development environment. The topic for each linker option discusses how it can be set in the development environment. See Linker Options for a complete list.

When you run LINK outside the development environment, you can specify input in one or more ways:

- On the command line
- Using command files
- In environment variables

LINK first processes options specified in the LINK environment variable, followed by options in the order they are specified on the command line and in command files. If an option is repeated with different arguments, the last one processed takes precedence.

Options apply to the entire build; no options can be applied to specific input files.

See Also

Reference

Linker Options

Other Resources

C/C++ Building Reference

Linker Command-Line Syntax

To run LINK.EXE, use the following command syntax:

```
LINK arguments
```

The *arguments* include options and filenames and can be specified in any order. Options are processed first, then files. Use one or more spaces or tabs to separate arguments.

On the command line, an option consists of an option specifier, either a dash (–) or a forward slash (/), followed by the name of the option. Option names cannot be abbreviated. Some options take an argument, specified after a colon (:). No spaces or tabs are allowed within an option specification, except within a quoted string in the /COMMENT option. Specify numeric arguments in decimal or C-language notation. Option names and their keyword or filename arguments are not case sensitive, but identifiers as arguments are case sensitive.

To pass a file to the linker, specify the filename on the command line after the LINK command. You can specify an absolute or relative path with the filename, and you can use wildcards in the filename. If you omit the dot (.) and filename extension, LINK assumes .obj for the purpose of finding the file. LINK does not use filename extensions or the lack of them to make assumptions about the contents of files; it determines the type of file by examining it, and processes it accordingly.

link.exe returns zero for success (no errors). Otherwise, the linker returns the error number that stopped the link. For example, if the linker generates LNK1104, the linker returns 1104. Accordingly, the lowest error number returned on an error by the linker is 1000. A return value of 128 represents a configuration problem with either the operating system or a .config file; the loader didn't load either link.exe or c2.dll.

See Also

Reference

Setting Linker Options
Linker Options

LINK Command Files

You can pass command-line arguments to LINK in the form of a command file. To specify a command file to the linker, use the following syntax:

```
LINK @commandfile
```

The *commandfile* is the name of a text file. No space or tab is allowed between the at sign (@) and the filename. There is no default extension; you must specify the full filename, including any extension. Wildcards cannot be used. You can specify an absolute or relative path with the filename. LINK does not use an environment variable to search for the file.

In the command file, arguments can be separated by spaces or tabs (as on the command line) and by newline characters.

You can specify all or part of the command line in a command file. You can use more than one command file in a LINK command. LINK accepts the command-file input as if it were specified in that location on the command line. Command files cannot be nested. LINK echoes the contents of command files, unless the /NOLOGO option is specified.

Example

The following command to build a DLL passes the names of object files and libraries in separate command files and uses a third command file for specification of the /EXPORTS option:

```
link /dll @objlist.txt @liblist.txt @exports.txt
```

See Also

Reference

Setting Linker Options

Linker Options

LINK Environment Variables

The LINK tool uses the following environment variables:

- LINK, if defined. The LINK tool processes options and arguments defined in the LINK environment variable before processing the command line.
- LIB, if defined. The LINK tools uses the LIB path when searching for an object, library, or other file specified on the command line or by the /BASE option. It also uses the LIB path to find a .pdb file named in an object. The LIB variable can contain one or more path specifications, separated by semicolons. One path must point to the \lib subdirectory of your Visual C++ installation.
- PATH, if the tool needs to run CVTRES and cannot find the file in the same directory as LINK itself. (LINK requires CVTRES to link a .res file.) PATH must point to the \bin subdirectory of your Visual C++ installation.
- TMP, to specify a directory when linking OMF or .res files.

See Also

Tasks

Setting the Path and Environment Variables for Command-Line Builds

Reference

Setting Linker Options

Linker Options

Linker Options

LINK is a tool that links Common Object File Format (COFF) object files and libraries to create a executable (.exe) file or dynamic-link library (DLL).

The table below is a comprehensive list of options for LINK.exe. This section also includes information on:

- Compiler-Controlled LINK Options
- LINK Input Files
- LINK Output
- Reserved Words

Linker options specified on the command line are not case sensitive: /base and /BASE mean the same thing.

You can specify some linker options via the comment pragma.

Option	Purpose
@	Specifies a response file
/ALIGN	Specifies the alignment of each section
/ALLOWBIND	Specifies that a DLL cannot be bound
/ALLOWISOLATION	Specifies behavior for manifest lookup.
/ASSEMBLYDEBUG	Adds the DebuggableAttribute to a managed image.
/ASSEMBLYLINKRESOURCE	Create a link to a managed resource.
/ASSEMBLYMODULE	Specifies that a Microsoft Intermediate Language (MSIL) module should be imported int o the assembly
/ASSEMBLYRESOURCE	Embeds a managed resource file to an assembly
/BASE	Sets a base address for the program
/CLRIMAGETYPE	Sets the type (IJW, pure, or safe) of a CLR image.
/CLRSUPPORTLASTERROR	Preserves the last error code of functions called through the P/Invoke mechanism.
/CLRTHREADATTRIBUTE	Specify which threading attribute you want applied to the entry point of your CLR progra m.
/CLRUNMANAGEDCODECHECK	/CLRUNMANAGEDCODECHECK specifies whether the linker will apply the SuppressUnm anagedCodeSecurity attribute to linker-generated PInvoke stubs that call from managed code into native DLLs.
/DEBUG	Creates debugging information
/DEF	Passes a module-definition (.def) file to the linker
/DEFAULTLIB	Searches the specified library when resolving external references
/DELAY	Controls the delayed loading of DLLs

/DELAYLOAD	Causes the delayed loading of the specified DLL
/DELAYSIGN	Partially sign an assembly.
/DLL	Builds a DLL
/DRIVER	Creates a Windows NT kernel mode driver
/ENTRY	Sets the starting address
/errorReport	Report internal linker errors to Microsoft.
/EXPORT	Exports a function
/FIXED	Creates a program that can be loaded only at its preferred base address
/FORCE	Forces a link to complete in spite of unresolved or symbols defined more than once
/FUNCTIONPADMIN	Creates a hotpatchable image.
/HEAP	Sets the size of the heap in bytes
/IDLOUT	Specifies the name of the .idl file and other MIDL output files
/IGNOREIDL	Prevents processing attribute information into an .idl file
/IMPLIB	Overrides the default import library name
/INCLUDE	Forces symbol references
/INCREMENTAL	Controls incremental linking
/KEYCONTAINER	Specify a key container to sign an assembly.
/KEYFILE	Specify key or key pair to sign an assembly.
/LARGEADDRESSAWARE	Tells the compiler that the application supports addresses larger than two gigabytes
/LIBPATH	Allows the user to override the environmental library path
/LTCG	Specifies link-time code generation
/MACHINE	Specifies the target platform
/MANIFEST	Create a side-by-side manifest file.
/MANIFESTDEPENDENCY	Specify a <dependentAssembly> section in your manifest file.
/MANIFESTFILE (Name Manifest File)	Change the default name of the manifest file.
/MAP	Creates a mapfile
/MAPINFO	Includes the specified information in the mapfile

/MERGE	Combines sections
/MIDL	Specifies MIDL command line options
/NOASSEMBLY	Suppresses the creation of a .NET Framework assembly
/NODEFAULTLIB	Ignores all (or specified) default libraries when resolving external references
/NOENTRY	Creates a resource-only DLL
/NOLOGO	Suppresses startup banner
/NXCOMPAT	Marks an executable as having been tested to be compatible with Windows Data Executi on Prevention feature.
/OPT	Controls LINK optimizations
/ORDER	Places COMDATs into the image in a predetermined order
/OUT	Specifies the output file name
/PDB	Creates a program database (PDB) file
/PDBSTRIPPED	Creates a program database (PDB) file with no private symbols
/PGD	Specify .pgd file for profile guided optionizations.
/PROFILE	Produces an output file that can be used with the Performance Tools profiler.
/RELEASE	Sets the Checksum in the .exe header
/SAFESEH	Specify that the image will contain a table of safe exception handlers.
/SECTION	Overrides the attributes of a section
/STACK	Sets the size of the stack in bytes
/STUB	Attaches an MS-DOS stub program to a Win32 program
/SUBSYSTEM	Tells the operating system how to run the .exe file
/SWAPRUN	Tells the operating system to copy the linker output to a swap file before running it
/TLBID	Allows you to specify the resource ID of the linker-generated type library
/TLBOUT	Specifies the name of the .tlb file and other MIDL output files
/TSAWARE	Creates an application that is specifically designed to run under Terminal Server
/VERBOSE	Prints linker progress messages
/VERSION	Assigns a version number

/WX	Treat linker warnings as errors.

For related information, see Compiler-Controlled LINK Options.

See Also

Reference

Setting Linker Options

Frequently Asked Questions on Building

Other Resources

C/C++ Building Reference

Compiler-Controlled LINK Options

The CL compiler automatically calls LINK unless you specify the /c option. CL provides some control over the linker through command-line options and arguments. The following table summarizes the features in CL that affect linking.

CL specification	**CL action that affects LINK**
Any file name extension other than .c, .cxx, .cpp, or .def	Passes a file name as input to LINK
filename.def	Passes /DEF:*filename*.def
/F*number*	Passes /STACK:*number*
/Fd*filename*	Passes /PDB:*filename*
/Fe*filename*	Passes /OUT:*filename*
/Fm*filename*	Passes /MAP:*filename*
/Gy	Creates packaged functions (COMDATs); enables function-level linking
/LD	Passes /DLL
/LDd	Passes /DLL
/link	Passes remainder of command line to LINK
/MD or /MT	Places a default library name in the .obj file
/MDd or /MTd	Places a default library name in the .obj file. Defines the symbol **_DEBUG**
/nologo	Passes /NOLOGO
/Zd	Passes /DEBUG
/Zi or /Z7	Passes /DEBUG
/Zl	Omits default library name from .obj file

For more information, see Compiler Options.

See Also
Reference
Setting Linker Options
Linker Options

LINK Input Files

You provide the linker with files that contain objects, import and standard libraries, resources, module definitions, and command input. LINK does not use file extensions to make assumptions about the contents of a file. Instead, LINK examines each input file to determine what kind of file it is.

Object files on the command line are processed in the order they appear on the command line. Libraries are searched in command line order as well, with the following caveat: Symbols that are unresolved when bringing in an object file from a library are searched for in that library first, and then the following libraries from the command line and /DEFAULTLIB (Specify Default Library) directives, and then to any libraries at the beginning of the command line.

Note
LINK no longer accepts a semicolon (or any other character) as the start of a comment in response files and order files. Semicolons are recognized only as the start of comments in module-definition files (.def).

LINK uses the following types of input files:

- .obj files
- .netmodule files
- .lib files
- .exp files
- .def files
- .pdb files
- .res files
- .exe files
- .txt files
- .ilk files

See Also

Reference

Setting Linker Options

Linker Options

.obj Files as Linker Input

LINK accepts .obj files that are either Common Object File Format (COFF) or 32-bit Object Module Format (OMF). Microsoft's Visual C++ compiler creates COFF .obj files.

LINK automatically converts 32-bit OMF objects to COFF. However, there are limitations to OMF to COFF conversions. OMF can represent some things that cannot be represented in COFF. If there are errors when the linker converts from OMF to COFF, then you will need to use COFF .obj files instead of OMF .obj files as input to the linker.

In some circumstances, .obj files can be used instead of .netmodule files. For information on when to use .netmodule files and when to use .obj files, see Choosing the Format of .netmodule Input Files.

See Also

Reference

LINK Input Files

Linker Options

.netmodule Files as Linker Input

link.exe now accepts MSIL .obj and .netmodules as input. The output file produced by the linker will be an assembly or a .netmodule with no run-time dependency on any of the .obj or .netmodules that were input to the linker.

.netmodules are created by the Visual C++ compiler with /LN (Create MSIL Module) or by the linker with /NOASSEMBLY (Create a MSIL Module). .objs are always created in a Visual C++ compilation. For other Visual Studio compilers, use the **/target:module** compiler option.

In most cases, you will need to pass to the linker the .obj file from the Visual C++ compilation that created the .netmodule, unless the .netmodule was created with /clr (Common Language Runtime Compilation). MSIL .netmodules used as input to the linker must be pure MSIL, which can be produced by the Visual C++ compiler using **/clr:safe**. Other Visual Studio compilers produce pure MSIL modules by default.

For information on how to invoke the linker from the command line, see Linker Command-Line Syntax and Setting the Path and Environment Variables for Command-Line Builds.

Passing a .netmodule or .dll file to the linker that was compiled by the Visual C++ compiler with **/clr** or with **/clr:pure** can result in a linker error. For more information, see Choosing the Format of .netmodule Input Files.

The linker accepts native .obj files as well as MSIL .obj files compiled with **/clr**, **/clr:pure**, or **/clr:safe**. When passing mixed .objs in the same build, the verifiability of the resulting output file will, by default, be equal to the lowest level of verifiability of the input modules. For example, if you pass a safe and pure .obj to the linker, the output file will be pure. /CLRIMAGETYPE (Specify Type of CLR Image) lets you specify a lower level of verifiability, if that is what you need.

If you currently have an application that is composed of two or more assemblies and you want the application to be contained in one assembly, you must recompile the assemblies and then link the .objs or .netmodules to produce a single assembly.

You must specify an entry point using /ENTRY (Entry-Point Symbol) when creating an executable image.

When linking with an MSIL .obj or .netmodule file, use /LTCG (Link-time Code Generation), otherwise when the linker encounters the MSIL .obj or .netmodule, it will restart the link with /LTCG.

MSIL .obj or .netmodule files can also be passed to cl.exe.

Input MSIL .obj or .netmodule files cannot have embedded resources. A resource is embedded in an output file (module or assembly) with /ASSEMBLYRESOURCE (Embed a Managed Resource) linker option or with the **/resource** compiler option in other Visual Studio compilers.

When performing MSIL linking, and if you do not also specify /LTCG (Link-time Code Generation), you will see an informational message reporting that the link is restarting. This message can be ignored, but to improve linker performance with MSIL linking, explicitly specify **/LTCG**.

Example

In C++ code the catch block of a corresponding try will be invoked for a non System exception. However, by default, the CLR wraps non System exceptions with RuntimeWrappedException. When an assembly is created from Visual C++ and non Visual C++ modules and you want a catch block in C++ code to be invoked from its corresponding try clause when the try block throws a non System exception, you must add the

[assembly:System::Runtime::CompilerServices::RuntimeCompatibility(WrapNonExceptionThrows=false)] attribute to the source code for the non C++ modules.

```
// MSIL_linking.cpp
// compile with: /c /clr
value struct V {};

ref struct MCPP {
   static void Test() {
      try {
         throw (gcnew V);
      }
      catch (V ^) {
         System::Console::WriteLine("caught non System exception in C++ source code file");
      }
   }
};
```

```
/*
int main() {
   MCPP::Test();
}
*/
```

By changing the Boolean value of the WrapNonExceptionThrows attribute, you modify the ability of the Visual C++ code to catch a non System exception.

```
// MSIL_linking_2.cs
// compile with: /target:module /addmodule:MSIL_linking.obj
// post-build command: link /LTCG MSIL_linking.obj MSIL_linking_2.netmodule /entry:MLinkTes
t.Main /out:MSIL_linking_2.exe /subsystem:console
using System.Runtime.CompilerServices;

// enable non System exceptions
[assembly:RuntimeCompatibility(WrapNonExceptionThrows=false)]

class MLinkTest {
   public static void Main() {
      try {
         MCPP.Test();
      }
      catch (RuntimeWrappedException) {
         System.Console.WriteLine("caught a wrapped exception in C#");
      }
   }
}
```

Output

```
caught non System exception in C++ source code file
```

See Also

Reference

LINK Input Files

Linker Options

Choosing the Format of .netmodule Input Files

An MSIL .obj file (compiled with /clr) can also be used as a .netmodule file. .obj files contain metadata and native symbols. .netmodules only contain metadata.

You can pass an MSIL .obj file to any other Visual Studio compiler via the /addmodule compiler option (but be aware that the .obj file becomes part of the resulting assembly and must be shipped with the assembly). For example, Visual C# and Visual Basic have the /addmodule compiler option.

Note
In most cases, you will need to pass to the linker the .obj file from the compilation that created the .net module. One exceptio n to this is if the .netmodule was created with /clr:pure. Passing a .dll or .netmodule MSIL module file to the linker may result in LNK1107.

.obj files, along with their associated .h files, which you reference via #include in source, allow C++ applications to consume the native types in the module, whereas in a .netmodule file, only the managed types can be consumed by a C++ application. If you attempt to pass a .obj file to #using, information about native types will not be available; #include the .obj file's .h file instead.

Other Visual Studio compilers can only consume managed types from a module.

Use the following to determine whether you need to use a .netmodule or a .obj file as module input to the Visual C++ linker:

- If you are building with a Visual Studio compiler other than Visual C++, produce a .netmodule and use the .netmodule as input to the linker.
- If you are using the Visual C++ compiler to produce modules and if the module(s) will be used to build something other than a library, use the .obj files produced by the compiler as module input to the linker; do not use the .netmodule file as input.
- If your modules will be used to build a native (not a managed) library, use .obj files as module input to the linker and generate a .lib library file.
- If your modules will be used to build a managed library, and if all module input to the linker will be verifiable (produced with /clr:safe), use .obj files as module input to the linker and generate a .dll (assembly) or .netmodule (module) library file.
- If your modules will be used to build a managed library, and if all module input to the linker will be produced with /clr:pure or /clr:safe, use .obj files as module input to the linker and generate a .dll (assembly) or .netmodule (module) if you only want to expose managed types from the library. If you want to expose managed types from the library and if you also want C++ applications to consume the native types in the library, your library will consist of the .obj files for the libraries component modules (you will also want to ship the .h files for each module, so they can be referenced with #include from source code).
- If your modules will be used to build a managed library, and if one or more modules input to the linker will be produced with just /clr, use .obj files as module input to the linker and generate a .dll (assembly). If you want to expose managed types from the library and if you also want C++ applications to consume the native types in the library, your library will consist of the .obj files for the libraries component modules (you will also want to ship the .h files for each module, so they can be referenced with #include from source code).

See Also

Reference

.netmodule Files as Linker Input

.lib Files as Linker Input

LINK accepts COFF standard libraries and COFF import libraries, both of which usually have the extension .lib. Standard libraries contain objects and are created by the LIB tool. Import libraries contain information about exports in other programs and are created either by LINK when it builds a program that contains exports or by the LIB tool. For information on using LIB to create standard or import libraries, see LIB Reference. For details on using LINK to create an import library, see the /DLL option.

A library is specified to LINK as either a file name argument or a default library. LINK resolves external references by searching first in libraries specified on the command line, then in default libraries specified with the /DEFAULTLIB option, and then in default libraries named in .obj files. If a path is specified with the library name, LINK looks for the library in that directory. If no path is specified, LINK looks first in the directory that LINK is running from, and then in any directories specified in the LIB environment variable.

To add .lib files as linker input in the development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Input** property page.
4. Modify the **Additional Dependencies** property.

To programmatically add .lib files as linker input

- See AdditionalDependencies.

Example

The following sample shows how to build and use a .lib file:

```
// lib_link_input_1.cpp
// compile with: /LD
__declspec(dllexport) int Test() {
   return 213;
}
```

And then:

```
// lib_link_input_2.cpp
// compile with: /EHsc lib_link_input_1.lib
__declspec(dllimport) int Test();
#include <iostream>
int main() {
   std::cout << Test() << std::endl;
}
```

Output

```
213
```

See Also

Reference

LINK Input Files

Linker Options

.exp Files as Linker Input

Export (.exp) files contain information about exported functions and data items. When LIB creates an import library, it also creates an .exp file. You use the .exp file when you link a program that both exports to and imports from another program, either directly or indirectly. If you link with an .exp file, LINK does not produce an import library, because it assumes that LIB already created one. For details about .exp files and import libraries, see Working with Import Libraries and Export Files.

See Also

Reference

LINK Input Files

Linker Options

.def Files as Linker Input

See Module-definition (.def) files for more information. Use the /DEF option to specify the .def file name.

See Also

Reference

LINK Input Files

Linker Options

.pdb Files as Linker Input

Object (.obj) files compiled using the /Zi option contain the name of a program database (PDB). You do not specify the object's PDB file name to the linker; LINK uses the embedded name to find the PDB if it is needed. This also applies to debuggable objects contained in a library; the PDB for a debuggable library must be available to the linker along with the library.

LINK also uses a PDB to hold debugging information for the .exe file or the .dll file. The program's PDB is both an output file and an input file, because LINK updates the PDB when it rebuilds the program.

See Also

Reference

LINK Input Files

Linker Options

.res Files as Linker Input

You can specify a .res file when linking a program. The .res file is created by the resource compiler (RC). LINK automatically converts .res files to COFF. The CVTRES.exe tool must be in the same directory as LINK.exe or in a directory specified in the PATH environment variable.

See Also

Reference

LINK Input Files

Linker Options

.exe Files as Linker Input

The MS-DOS Stub File Name (/STUB) option specifies the name of an .exe file that runs with MS-DOS. LINK examines the specified file to be sure that it is a valid MS-DOS program.

See Also

Reference

LINK Input Files

Linker Options

.txt Files as Linker Input

LINK expects various text files as additional input. The command-file specifier (@) and the Base Address (/BASE), /DEF, and /ORDER options all specify text files. These files can have any extension, not just .txt.

See Also

Reference

LINK Input Files

Linker Options

.ilk Files as Linker Input

When linking incrementally, LINK updates the .ilk status file that it created during the first incremental link. This file has the same base name as the .exe file or the .dll file, and it has the extension .ilk. During subsequent incremental links, LINK updates the .ilk file. If the .ilk file is missing, LINK performs a full link and creates a new .ilk file. If the .ilk file is unusable, LINK performs a nonincremental link. For details about incremental linking, see the Link Incrementally (/INCREMENTAL) option.

See Also

Reference

LINK Input Files

Linker Options

LINK Output

Link output includes .exe files, DLLs, mapfiles, and messages.

Output Files

The default output file from LINK is an .exe file. If the /DLL option is specified, LINK builds a .dll file. You can control the output file name with the Output File Name (/OUT) option.

In incremental mode, LINK creates an .ilk file to hold status information for later incremental builds of the program. For details about .ilk files, see .ilk Files. For more information about incremental linking, see the Link Incrementally (/INCREMENTAL) option.

When LINK creates a program that contains exports (usually a DLL), it also builds a .lib file, unless an .exp file was used in the build. You can control the import library file name with the /IMPLIB option.

If the Generate Mapfile (/MAP) option is specified, LINK creates a mapfile.

If the Generate Debug Info (/DEBUG) option is specified, LINK creates a PDB to contain debugging information for the program.

Other Output

When you type `link` without any other command-line input, LINK displays a usage statement that summarizes its options.

LINK displays a copyright and version message and echoes command-file input, unless the Suppress Startup Banner (/NOLOGO) option is used.

You can use the Print Progress Messages (/VERBOSE) option to display additional details about the build.

LINK issues error and warning messages in the form LNK*nnnn*. This error prefix and range of numbers are also used by LIB, DUMPBIN, and EDITBIN.

See Also

Reference

Setting Linker Options

Linker Options

Reserved Words

The following words are reserved by the linker. These names can be used as arguments in module-definition statements only if the name is enclosed in double quotation marks ("").

APPLOADER1	**INITINSTANCE**2	**PRELOAD**
BASE	**IOPL**	**PRIVATE**
CODE	**LIBRARY**1	**PROTMODE**2
CONFORMING	**LOADONCALL**1	**PURE**1
DATA	**LONGNAMES**2	**READONLY**
DESCRIPTION	**MOVABLE**1	**READWRITE**
DEV386	**MOVEABLE**1	**REALMODE**1
DISCARDABLE	**MULTIPLE**	**RESIDENT**
DYNAMIC	**NAME**	**RESIDENTNAME**1
EXECUTE-ONLY	**NEWFILES**2	**SECTIONS**
EXECUTEONLY	**NODATA**1	**SEGMENTS**
EXECUTEREAD	**NOIOPL**1	**SHARED**
EXETYPE	**NONAME**	**SINGLE**
EXPORTS	**NONCONFORMING**1	**STACKSIZE**
FIXED1	**NONDISCARDABLE**	**STUB**
FUNCTIONS2	**NONE**	**VERSION**
HEAPSIZE	**NONSHARED**	**WINDOWAPI**
IMPORTS	**NOTWINDOWCOMPAT**1	**WINDOWCOMPAT**
IMPURE1	**OBJECTS**	**WINDOWS**
INCLUDE2	**OLD**1	

1 The linker emits a warning ("ignored") when it encounters this term. However, the word is still reserved.

2 The linker ignores this word but emits no warning.

See Also

Reference

Setting Linker Options

Linker Options

@ (Specify a Linker Response File)

```
@response_file
```

Remarks

where:

response_file

A text file specifying linker commands.

Remarks

See @ (Specify a Compiler Response File) for more information.

To set this linker option in the Visual Studio development environment

- This linker option is not available from the Visual Studio development environment.

To set this linker option programmatically

- This linker option cannot be changed programmatically.

See Also

Reference

Setting Linker Options

Linker Options

/ALIGN (Section Alignment)

```
/ALIGN[:number]
```

Remarks

where:

number

The alignment value.

Remarks

The /ALIGN option specifies the alignment of each section within the linear address space of the program. The *number* argument is in bytes and must be a power of two. The default is 4K (4096). The linker issues a warning if the alignment produces an invalid image.

Unless you are writing an application such as a device driver, you should not need to modify the alignment.

It is possible to modify the alignment of a particular section with the align parameter to the /SECTION option.

The alignment value that you specify cannot be smaller than the largest section alignment.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Command Line** property page.
4. Type the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See Also

Reference

Setting Linker Options

Linker Options

/ALLOWBIND (Prevent DLL Binding)

```
/ALLOWBIND[:NO]
```

Remarks

/ALLOWBIND:NO sets a bit in a DLL's header that indicates to Bind.exe that the image is not allowed to be bound. You may not want a DLL to be bound if it has been digitally signed (binding invalidates the signature).

You can edit an existing DLL for /ALLOWBIND functionality with the /ALLOWBIND option of the EDITBIN utility.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Command Line** property page.
4. Type the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See Also

Reference

Setting Linker Options
Linker Options
BindImage
BindImageEx

/ALLOWISOLATION (Manifest Lookup)

Specifies behavior for manifest lookup.

```
/ALLOWISOLATION[:NO]
```

Remarks

/ALLOWISOLATION:NO indicates DLLs are loaded as if there was no manifest and causes the linker to set the **IMAGE_DLLCHARACTERISTICS_NO_ISOLATION** bit in the optional header's **DllCharacteristics** field.

/ALLOWISOLATION causes the operating system to do manifest lookups and loads.

/ALLOWISOLATION is the default.

When isolation is disabled for an executable, the Windows loader will not attempt to find an application manifest for the newly created process. The new process will not have a default activation context, even if there is a manifest inside the executable or placed in the same directory as the executable with name *executable-name***.exe.manifest**.

For more information, see Manifest Files Reference.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Expand the **Configuration Properties** node.
3. Expand the **Linker** node.
4. Select the **Manifest File** property page.
5. Modify the **Allow Isolation** property.

See Also

Reference

Setting Linker Options

Linker Options

/ASSEMBLYDEBUG (Add DebuggableAttribute)

```
/ASSEMBLYDEBUG[:DISABLE]
```

/ASSEMBLYDEBUG emits the **DebuggableAttribute** attribute with debug information tracking and disables JIT optimizations. This is the same as specifying the following attribute in source:

```
[assembly:Debuggable(true, true)];   // same as /ASSEMBLYDEBUG
```

/ASSEMBLYDEBUG:DISABLE emits the **DebuggableAttribute** attribute but disables the tracking of debug information and enables JIT optimizations. This is the same as specifying the following attribute in source:

```
[assembly:Debuggable(false, false)];   // same as /ASSEMBLYDEBUG:DISABLE
```

The default is to not emit the **DebuggableAttribute** attribute.

DebuggableAttribute can also be added to an assembly directly in source code. For example,

```
[assembly:Debuggable(true, true)];   // same as /ASSEMBLYDEBUG
```

Remarks

In Visual C++ .NET 2003 and later, it is necessary to explicitly specify that a managed image be debuggable. Using /Zi alone is not sufficient.

Other linker options that affect assembly generation are:

- /ASSEMBLYLINKRESOURCE
- /ASSEMBLYMODULE
- /ASSEMBLYRESOURCE
- /DELAYSIGN
- /KEYCONTAINER
- /KEYFILE
- /NOASSEMBLY

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Debug** property page.
4. Modify the **Debuggable Assembly** property.

To set this linker option programmatically

- See AssemblyDebug.

See Also

Reference

Setting Linker Options

Linker Options

/ASSEMBLYLINKRESOURCE (Link to .NET Framework Resource)

```
/ASSEMBLYLINKRESOURCE:filename
```

Remarks

where:

filename

The .NET Framework resource file to which you want to link from the assembly.

Remarks

The /ASSEMBLYLINKRESOURCE option creates a link to a .NET Framework resource in the output file; the resource file is not placed in the output file. /ASSEMBLYRESOURCE embeds a resource file in the output file.

Linked resources are public in the assembly when created with the linker.

/ASSEMBLYLINKRESOURCE requires that the compilation include /clr; /LN or /NOASSEMBLY is not allowed with /ASSEMBLYLINKRESOURCE.

If *filename* is a .NET Framework resource file created, for example, by Resgen.exe or in the development environment, it can be accessed with members in the **System.Resources** namespace. For more information, see System.Resources.ResourceManager. For all other resources, use the **GetManifestResource*** methods in the **System.Reflection.Assembly** class to access the resource at run time.

filename can be any file format. For example, you may want to make a native DLL part of the assembly, so it can be installed into the Global Assembly Cache and accessed from managed code in the assembly.

Other linker options that affect assembly generation are:

- /ASSEMBLYDEBUG
- /ASSEMBLYMODULE
- /ASSEMBLYRESOURCE
- /DELAYSIGN
- /KEYCONTAINER
- /KEYFILE
- /NOASSEMBLY

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Command Line** property page.
4. Type the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See Also

Reference

Setting Linker Options

Linker Options

/ASSEMBLYMODULE (Add a MSIL Module to the Assembly)

```
/ASSEMBLYMODULE:filename
```

Remarks

where:

filename

The module you want to include in this assembly.

Remarks

The /ASSEMBLYMODULE option allows you to add a module reference to an assembly. Type information in the module will not be available to the assembly program that added the module reference. However, type information in the module will be available to any program that references the assembly.

Use #using to both add a module reference to an assembly and make the module's type information available to the assembly program.

For example, consider the following scenario:

1. Create a module with /LN.
2. Use /ASSEMBLYMODULE in a different project to include the module in the current compilation, which will create an assembly. This project will not reference the module with **#using**.
3. Any project that references this assembly can now also use types from the module.

Other linker options that affect assembly generation are:

- /ASSEMBLYDEBUG
- /ASSEMBLYLINKRESOURCE
- /ASSEMBLYRESOURCE
- /DELAYSIGN
- /NOASSEMBLY
- /KEYFILE
- /KEYCONTAINER

The Visual C++ linker accepts .netmodule files as input and the output file produced by the linker will be an assembly or .netmodule with no run-time dependence on any of the .netmodules that were input to the linker. For more information, see .netmodule Files as Linker Input.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Input** property page.
4. Modify the **Add Module to Assembly** property.

To set this linker option programmatically

- See AddModuleNamesToAssembly.

See Also
Reference
Setting Linker Options

/ASSEMBLYRESOURCE (Embed a Managed Resource)

```
/ASSEMBLYRESOURCE:filename[,[name][,PRIVATE]]
```

Parameters

filename

The managed resource you want to embed in this assembly.

name

Optional. The logical name for the resource; the name used to load the resource. The default is the name of the file.

Optionally, you can specify if the file should be private in the assembly manifest. By default, *name* is public in the assembly.

Remarks

Use the /ASSEMBLYRESOURCE option to embed a resource in an assembly.

Resources are public in the assembly when created with the linker. The linker does not allow you to rename the resource in the assembly.

If *filename* is a .NET Framework resource (.resources) file created, for example, by the Resource File Generator (Resgen.exe) or in the development environment, it can be accessed with members in the **System.Resources** namespace (see System.Resources.ResourceManager for more information). For all other resources, use the **GetManifestResource*** methods in **System.Reflection.Assembly** class to access the resource at run time.

Other linker options that affect assembly generation are:

- /ASSEMBLYDEBUG
- /ASSEMBLYLINKRESOURCE
- /ASSEMBLYMODULE
- /DELAYSIGN
- /KEYFILE
- /KEYCONTAINER
- /NOASSEMBLY

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Input** property page.
4. Modify the **Embed Managed Resource File** property.

To set this linker option programmatically

- See EmbedManagedResourceFile.

See Also

Reference

Setting Linker Options

Linker Options

/BASE (Base Address)

```
/BASE:{address[,size] | @filename,key}
```

The /BASE option sets a base address for the program, overriding the default location for an .exe file (at 0x400000) or a DLL (at 0x10000000). The operating system first attempts to load a program at its specified or default base address. If sufficient space is not available there, the system relocates the program. To prevent relocation, use the /FIXED option.

The linker issues an error if *address* is not a multiple of 64K. You can optionally specify the size of the program, so the linker will warn if the program can not fit in the size you specified.

On the command line, another way to specify the base address is by using the *filename* preceded by an at sign (@), and a *key* into the file. The *filename* is a text file that contains the locations and sizes of all the DLLs your program will use. The linker looks for *filename* in either the specified path or, if no path is specified, in directories specified in the LIB environment variable. Each line in *filename* represents one DLL and has the following syntax:

```
      key address [size] ;comment
```

The *key* is a string of alphanumeric characters and is not case sensitive. It is usually the name of a DLL, but it need not be. The *key* is followed by a base *address* in C-language, hexadecimal, or decimal notation and an optional maximum *size*. All three arguments are separated by spaces or tabs. The linker issues a warning if the specified *size* is less than the virtual address space required by the program. A *comment* is specified by a semicolon (;) and can be on the same or a separate line. The linker ignores all text from the semicolon to the end of the line. This example shows part of such a file:

```
main   0x00010000    0x08000000    ; for PROJECT.exe
one    0x28000000    0x00100000    ; for DLLONE.DLL
two    0x28100000    0x00300000    ; for DLLTWO.DLL
```

If the file that contains these lines is called DLLS.txt, the following example command applies this information:

```
link dlltwo.obj /dll /base:@dlls.txt,two
```

Remarks

You can reduce paging and improve performance of your program by assigning base addresses so that DLLs do not overlap in the address space.

Another way to set the base address is with the *BASE* argument in a NAME or LIBRARY statement. The /BASE and /DLL options together are equivalent to the **LIBRARY** statement.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Advanced** property page.
4. Modify the **Base Address** property.

To set this linker option programmatically

- See BaseAddress.

See Also
Reference
Setting Linker Options
Linker Options

/CLRIMAGETYPE (Specify Type of CLR Image)

```
/CLRIMAGETYPE:{IJW|PURE|SAFE}
```

Remarks

The linker accepts native objects as well as MSIL objects compiled with /clr, /clr:pure, or /clr:safe. When passing mixed objects in the same build, the verifiability of the resulting output file will, by default, be equal to the lowest level of verifiability of the input modules. For example, if you pass a safe and pure module to the linker, the output file will be pure. If you pass a native image and a mixed mode image (compiled with **/clr**), the resulting image will be a mixed mode image.

/CLRIMAGETYPE lets you specify a lower level of verifiability, if that is what you need.

See /clr (Common Language Runtime Compilation) for information on using **/CLRIMAGETYPE** to specify that a safe executable be run under WOW64 on a 64-bit operating system, when that safe image loads a 32-bit resource.

When running an .exe that was compiled with **/clr** or **/clr:pure** on a 64-bit operating system, the application will be run under WOW64, which allows a 32-bit application to run on a 64-bit operating system. By default, an .exe compiled with **/clr:safe** will be run using the operating system's 64-bit support. However, it is possible that your safe application loads a 32-bit component. In that case, a safe image running under the operating system's 64-bit support will fail when it loads the 32-bit application. To ensure that a safe image continues to run when it loads a 32-bit on a 64-bit operating system, you must use the /CLRIMAGETYPE (Specify Type of CLR Image) linker option to change the metadata (.corflags), marking it to be run under WOW64 (and substituting your own entry symbol):

cl /clr:safe t.cpp /link /clrimagetype:pure /entry:?main@@$$HYMHXZ /subsystem:console

For information on how to determine the CLR image type of a file, see /CLRHEADER.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Expand the **Configuration Properties** node.
3. Expand the **Linker** node.
4. Select the **Advanced** property page.
5. Modify the **CLR Image Type** property.

To set this linker option programmatically

- See CLRImageType.

See Also

Reference

Setting Linker Options

Linker Options

/CLRUNMANAGEDCODECHECK (Add SupressUnmanagedCodeSecurityAttribute)

/CLRUNMANAGEDCODECHECK specifies whether the linker will apply SuppressUnmanagedCodeSecurityAttribute to linker-generated PInvoke calls from managed code into native DLLs.

```
/CLRUNMANAGEDCODECHECK[:NO]
```

Remarks

By default, **/CLRUNMANAGEDCODECHECK** is in effect, which means SuppressUnmanagedCodeSecurityAttribute is applied to linker-generated PInvoke calls. Specify **/CLRUNMANAGEDCODECHECK:NO** to not apply this attribute.

The linker only adds the attribute to objects that are compiled with **/clr** or **/clr:pure**. The linker does not generate PInvoke calls in objects compiled with **/clr:safe**. For more information, see /clr (Common Language Runtime Compilation).

A PInovke call is generated by the linker when the linker can't find a managed symbol to satisfy a reference from a managed caller but can find a native symbol to satisfy that reference. For more information on PInvoke, see Calling Native Functions from Managed Code.

Note that if you use AllowPartiallyTrustedCallersAttribute in your code, you should explicitly set **/CLRUNMANAGEDCODECHECK**. It is potential security vulnerability if an image contains both the SuppressUnmanagedCodeSecurity and AllowPartiallyTrustedCallers attributes.

See Security Optimizations for more information on the implications of using SuppressUnmanagedCodeSecurityAttribute.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Expand the **Configuration Properties** node.
3. Expand the **Linker** node.
4. Select the **Advanced** property page.
5. Modify the **CLR Unmanaged Code Check** property.

To set this linker option programmatically

- See CLRUnmanagedCodeCheck.

See Also

Reference

Setting Linker Options

Linker Options

/CLRSUPPORTLASTERROR (Preserve Last Error Code for PInvoke Calls)

/CLRSUPPORTLASTERROR, which is on by default, preserves the last error code of functions called through the P/Invoke mechanism, which allows you to call native functions in DLLS, from code compiled with **/clr**.

```
/CLRSUPPORTLASTERROR{:NO | SYSTEMDLL}
```

Remarks

Preserving the last error code implies a decrease in performance. If you do not want to incur the performance impact of preserving the last error code, link with **/CLRSUPPORTLASTERROR:NO**.

You can minimize the performance impact by linking with **/CLRSUPPORTLASTERROR:SYSTEMDLL**, which only preserves the last error code for functions in system DLLs. A system DLL is defined as one of the following:

ACLUI.DLL	ACTIVEDS.DLL	ADPTIF.DLL	ADVAPI32.DLL
ASYCFILT.DLL	AUTHZ.DLL	AVICAP32.DLL	AVIFIL32.DLL
CABINET.DLL	CLUSAPI.DLL	COMCTL32.DLL	COMDLG32.DLL
COMSVCS.DLL	CREDUI.DLL	CRYPT32.DLL	CRYPTNET.DLL
CRYPTUI.DLL	D3D8THK.DLL	DBGENG.DLL	DBGHELP.DLL
DCIMAN32.DLL	DNSAPI.DLL	DSPROP.DLL	DSUIEXT.DLL
GDI32.DLL	GLU32.DLL	HLINK.DLL	ICM32.DLL
IMAGEHLP.DLL	IMM32.DLL	IPHLPAPI.DLL	IPROP.DLL
KERNEL32.DLL	KSUSER.DLL	LOADPERF.DLL	LZ32.DLL
MAPI32.DLL	MGMTAPI.DLL	MOBSYNC.DLL	MPR.DLL
MPRAPI.DLL	MQRT.DLL	MSACM32.DLL	MSCMS.DLL
MSI.DLL	MSIMG32.DLL	MSRATING.DLL	MSTASK.DLL
MSVFW32.DLL	MSWSOCK.DLL	MTXEX.DLL	NDDEAPI.DLL
NETAPI32.DLL	NPPTOOLS.DLL	NTDSAPI.DLL	NTDSBCLI.DLL
NTMSAPI.DLL	ODBC32.DLL	ODBCBCP.DLL	OLE32.DLL
OLEACC.DLL	OLEAUT32.DLL	OLEDLG.DLL	OPENGL32.DLL
PDH.DLL	POWRPROF.DLL	QOSNAME.DLL	QUERY.DLL
RASAPI32.DLL	RASDLG.DLL	RASSAPI.DLL	RESUTILS.DLL

RICHED20.DLL	RPCNS4.DLL	RPCRT4.DLL	RTM.DLL
RTUTILS.DLL	SCARDDLG.DLL	SECUR32.DLL	SENSAPI.DLL
SETUPAPI.DLL	SFC.DLL	SHELL32.DLL	SHFOLDER.DLL
SHLWAPI.DLL	SISBKUP.DLL	SNMPAPI.DLL	SRCLIENT.DLL
STI.DLL	TAPI32.DLL	TRAFFIC.DLL	URL.DLL
URLMON.DLL	USER32.DLL	USERENV.DLL	USP10.DLL
UXTHEME.DLL	VDMDBG.DLL	VERSION.DLL	WINFAX.DLL
WINHTTP.DLL	WININET.DLL	WINMM.DLL	WINSCARD.DLL
WINTRUST.DLL	WLDAP32.DLL	WOW32.DLL	WS2_32.DLL
WSNMP32.DLL	WSOCK32.DLL	WTSAPI32.DLL	XOLEHLP.DLL

Note

Preserving the last error is not supported for unmanaged functions that are consumed by CLR code, in the same module.

- For more information, see /clr (Common Language Runtime Compilation).

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Command Line** property page.
4. Type the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

Example

The following sample defines a native DLL with one exported function that modifies last error.

```
// CLRSUPPORTLASTERROR_dll.cpp
// compile with: /LD
#include <windows.h>
#include <math.h>

#pragma unmanaged
__declspec(dllexport) double MySqrt(__int64 n) {
   SetLastError(DWORD(-1));
   return sqrt(double(n));
}
```

The following sample consumes the DLL, demonstrating how to use **/CLRSUPPORTLASTERROR**.

```
// CLRSUPPORTLASTERROR_client.cpp
// compile with: /clr CLRSUPPORTLASTERROR_dll.lib /link /clrsupportlasterror:systemdll
// processor: x86
#include <windows.h>
```

```
#include <wininet.h>
#include <stdio.h>
#include <math.h>

#pragma comment(lib, "wininet.lib")

double MySqrt(__int64 n);

#pragma managed
int main() {
   double   d = 0.0;
   __int64 n = 65;
   HANDLE  hGroup = NULL;
   GROUPID groupID;
   DWORD   dwSet = 127, dwGet = 37;

   SetLastError(dwSet);
   d = MySqrt(n);
   dwGet = GetLastError();

   if (dwGet == DWORD(-1))
      printf_s("GetLastError for application call succeeded (%d).\n",
             dwGet);
   else
      printf_s("GetLastError for application call failed (%d).\n",
             dwGet);

   hGroup = FindFirstUrlCacheGroup(0, CACHEGROUP_SEARCH_ALL,
                           0, 0, &groupID, 0);
   dwGet = GetLastError();
   if (dwGet == 183)
      printf_s("GetLastError for system call succeeded (%d).\n",
             dwGet);
   else
      printf_s("GetLastError for system call failed (%d).\n",
             dwGet);
}
```

Sample Output

```
GetLastError for application call failed (127).
GetLastError for system call succeeded (183).
```

See Also

Reference

Setting Linker Options

Linker Options

/CLRTHREADATTRIBUTE (Set CLR Thread Attribute)

Explicitly specify the threading attribute for the entry point of your CLR program.

```
/CLRTHREADATTRIBUTE:{STA|MTA|NONE}
```

Parameters

MTA

Applies the MTAThreadAttribute attribute to the entry point of your program.

NONE

Same as not specifying /CLRTHREADATTRIBUTE. Lets the Common Language Runtime (CLR) set the default threading attribute.

STA

Applies the STAThreadAttribute attribute to the entry point of your program.

Remarks

Setting the thread attribute is only valid when building an .exe, as it affects the entry point of the main thread.

If you use the default entry point (main or wmain, for example) specify the threading model either by using /CLRTHREADATTRIBUTE or by placing the threading attribute (STAThreadAttribute or MTAThreadAttribute) on the default entry function.

If you use a non-default entry point, specify the threading model either by using /CLRTHREADATTRIBUTE or by placing the threading attribute on the non-default entry function, and then specify the non-default entry point with /ENTRY.

If the threading model specified in source code does not agree with the threading model specified with /CLRTHREADATTRIBUTE, the linker will ignore /CLRTHREADATTRIBUTE and apply the threading model specified in source code.

It will be necessary for you to use single-threading, for example, if your CLR program hosts a COM object that uses single-threading. If your CLR program uses multi-threading, it cannot host a COM object that uses single-threading.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Expand the **Configuration Properties** node.
3. Expand the **Linker** node.
4. Select the **Advanced** property page.
5. Modify the **CLR Thread Attribute** property.

To set this linker option programmatically

- See CLRThreadAttribute.

See Also

Reference

Setting Linker Options

Linker Options

/DEBUG (Generate Debug Info)

```
/DEBUG
```

Remarks

The /DEBUG option creates debugging information for the .exe file or DLL.

The linker puts the debugging information into a program database (PDB). It updates the PDB during subsequent builds of the program.

An .exe file or DLL created for debugging contains the name and path of the corresponding PDB. The debugger reads the embedded name and uses the PDB when you debug the program. The linker uses the base name of the program and the extension .pdb to name the program database, and embeds the path where it was created. To override this default, set /PDB and specify a different file name.

The compiler's Line Numbers Only (/Zd) or C7 Compatible (/Z7) option causes the compiler to leave the debugging information in the .obj files. You can also use the Program Database (/Zi) compiler option to store the debugging information in a PDB for the .obj file. The linker looks for the object's PDB first in the absolute path written in the .obj file, and then in the directory that contains the .obj file. You cannot specify an object's PDB file name or location to the linker.

/INCREMENTAL is implied when /DEBUG is specified.

/DEBUG changes the defaults for the /OPT option from REF to NOREF and from ICF to NOICF (so, you will need to explicitly specify /OPT:REF or /OPT:ICF).

See Knowledge Base article Q121366, INFO: PDB and DBG Files - What They Are and How They Work, for more information on .PDB and .DBG files. You can find Knowledge Base articles in the MSDN Library, or at http://support.microsoft.com.

It is not possible to create an .exe or .dll that contains debug information. Debug information is always placed in a .pdb file.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Debug** property page.
4. Modify the **Generate Debug Info** property.

To set this linker option programmatically

- See GenerateDebugInformation.

See Also

Reference

Setting Linker Options

Linker Options

/DEF (Specify Module-Definition File)

```
/DEF:filename
```

Remarks

where:

filename

The name of a module-definition file (.def) to be passed to the linker.

Remarks

The /DEF option passes a module-definition file (.def) to the linker. Only one .def file can be specified to LINK. For details about .def files, see Module-Definition Files.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Input** property page.
4. Modify the **Module Definition File** property.

To specify a .def file from within the development environment, you should add it to the project along with other files and then specify the file to the /DEF option.

To set this linker option programmatically

- See ModuleDefinitionFile.

See Also
Reference
Setting Linker Options
Linker Options

/DEFAULTLIB (Specify Default Library)

```
/DEFAULTLIB:library
```

Remarks

where:

library

The name of a library to search when resolving external references.

Remarks

The /DEFAULTLIB option adds one *library* to the list of libraries that LINK searches when resolving references. A library specified with /DEFAULTLIB is searched after libraries specified on the command line and before default libraries named in .obj files.

The Ignore All Default Libraries (/NODEFAULTLIB) option overrides /DEFAULTLIB:*library*. The Ignore Libraries (/NODEFAULTLIB:*library*) option overrides /DEFAULTLIB:*library* when the same *library* name is specified in both.

To set this linker option in the Visual Studio development environment

- This linker option is not available from the Visual Studio development environment. To add a library to the link phase, use the **Additional Dependencies** property from the **Input** property page.

To set this linker option programmatically

- See AdditionalOptions.

See Also
Reference
Setting Linker Options
Linker Options

/DELAY (Delay Load Import Settings)

```
/DELAY:UNLOAD
/DELAY:NOBIND
```

Remarks

The /DELAY option controls delayed loading of DLLs:

- The UNLOAD qualifier tells the delay-load helper function to support explicit unloading of the DLL. The Import Address Table (IAT) is reset to its original form, invalidating IAT pointers and causing them to be overwritten.

  If you do not select UNLOAD, any call to FUnloadDelayLoadedDLL will fail.

- The NOBIND qualifier tells the linker not to include a bindable IAT in the final image. The default is to create the bindable IAT for delay-loaded DLLs. The resulting image cannot be statically bound. (Images with bindable IATs may be statically bound prior to execution.) See /BIND.

  If the DLL is bound, the helper function will attempt to use the bound information instead of calling GetProcAddress on each of the referenced imports. If either the timestamp or the preferred address does not match those of the loaded DLL, the helper function will assume the bound IAT is out of date and will proceed as if the bound IAT does not exist.

  NOBIND causes your program image to be larger but can speed load time of the DLL. If you never intend to bind the DLL, NOBIND will prevent the bound IAT from being generated.

To specify DLLs to delay load, use the /DELAYLOAD option.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Advanced** property page.
4. Modify the **Delay Loaded DLL** property.

To set this linker option programmatically

- See DelayLoadDLLs.

See Also

Reference

Setting Linker Options
Linker Options
Peering Inside the PE: A Tour of the Win32 Portable Executable File Format

/DELAYLOAD (Delay Load Import)

/DELAYLOAD:*dllname*

Parameters
dllname

The name of a DLL that you want to delay load.

Remarks

The /DELAYLOAD option causes delayed loading of DLLs. The *dllname* specifies a DLL to delay load. You can use this option as many times as necessary to specify as many DLLs as you choose. You must link your program with Delayimp.lib or implement your own delay-load helper function.

The /DELAY option specifies binding and loading options for each delay-loaded DLL.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Input** property page.
4. Modify the **Delay Loaded DLLs** property.

To set this linker option programmatically

- See DelayLoadDLLs.

See Also
Reference
Setting Linker Options
Linker Options

/DELAYSIGN (Partially Sign an Assembly)

```
/DELAYSIGN[:NO]
```

Remarks

where,

NO

Specifies that the assembly should not be partially signed.

Remarks

Use **/DELAYSIGN** if you only want to place the public key in the assembly. The default is **/DELAYSIGN:NO**.

The **/DELAYSIGN** option has no effect unless used with /KEYFILE or /KEYCONTAINER.

When you request a fully signed assembly, the compiler hashes the file that contains the manifest (assembly metadata) and signs that hash with the private key. The resulting digital signature is stored in the file that contains the manifest. When an assembly is delay signed, the linker does not compute and store the signature, but reserves space in the file so the signature can be added later.

For example, using **/DELAYSIGN** allows a tester to put the assembly in the global cache. After testing, you can fully sign the assembly by placing the private key in the assembly.

See Strong Name Assemblies (Assembly Signing) and Delay Signing an Assembly for more information on signing an assembly.

Other linker options that affect assembly generation are:

- /ASSEMBLYDEBUG
- /ASSEMBLYLINKRESOURCE
- /ASSEMBLYMODULE
- /ASSEMBLYRESOURCE
- /NOASSEMBLY

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Command Line** property page.
4. Type the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See Also

Reference

Setting Linker Options

Linker Options

/DLL (Build a DLL)

```
/DLL
```

Remarks

The /DLL option builds a DLL as the main output file. A DLL usually contains exports that can be used by another program. There are three methods for specifying exports, listed in recommended order of use:

1. __declspec(dllexport) in the source code
2. An EXPORTS statement in a .def file
3. An /EXPORT specification in a LINK command

A program can use more than one method.

Another way to build a DLL is with the **LIBRARY** module-definition statement. The /BASE and /DLL options together are equivalent to the **LIBRARY** statement.

Do not specify this option within the development environment; this option is for use only on the command line. This option is set when you create a DLL project with an Application Wizard.

Note that if you create your import library in a preliminary step, before creating your .dll, you must pass the same set of object files when building the .dll, as you passed when building the import library.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Configuration Properties** folder.
3. Click the **General** property page.
4. Modify the **Configuration Type** property.

To set this linker option programmatically

- See ConfigurationType.

See Also

Reference

Setting Linker Options

Linker Options

/DRIVER (Windows NT Kernel Mode Driver)

```
/DRIVER[:UPONLY | :WDM]
```

Remarks

Use the /DRIVER linker option to build a Windows NT kernel mode driver.

/DRIVER:UPONLY causes the linker to add the **IMAGE_FILE_UP_SYSTEM_ONLY** bit to the characteristics in the output header to specify that it is a uniprocessor (UP) driver. The operating system will refuse to load a UP driver on a multiprocessor (MP) system.

/DRIVER:WDM causes the linker to set the **IMAGE_DLLCHARACTERISTICS_WDM_DRIVER** bit in the optional header's DllCharacteristics field.

If **/DRIVER** is not specified, these bits are not set by the linker.

If **/DRIVER** is specified:

- /FIXED:NO is in effect (/FIXED (Fixed Base Address)).
- The extension of the output file will be .sys. Use **/OUT** to change the default filename and extension (/OUT (Output File Name)).

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **System** property page.
4. Modify the **Driver** property.

To set this linker option programmatically

- See **P:Microsoft.VisualStudio.VCProjectEngine.VCLinkerTool.driver**.

See Also

Reference

Setting Linker Options

Linker Options

/ENTRY (Entry-Point Symbol)

```
/ENTRY:function
```

Remarks

where:

function

A function that specifies a user-defined starting address for an .exe file or DLL.

Remarks

The /ENTRY option specifies an entry point function as the starting address for an .exe file or DLL.

The function must be defined with the **__stdcall** calling convention. The parameters and return value must be defined as documented in the Win32 API for **WinMain** (for an .exe file) or **DllEntryPoint** (for a DLL). It is recommended that you let the linker set the entry point so that the C run-time library is initialized correctly, and C++ constructors for static objects are executed.

By default, the starting address is a function name from the C run-time library. The linker selects it according to the attributes of the program, as shown in the following table.

Function name	Default for
mainCRTStartup (or **wmainCRTStartup**)	An application using /SUBSYSTEM:**CONSOLE**; calls **main** (or **wmain**)
WinMainCRTStartup (or **wWinMainCRTStartup**)	An application using /SUBSYSTEM:**WINDOWS**; calls **WinMain** (or **wWinMain**), which must be defined with **__stdcall**
_DllMainCRTStartup	A DLL; calls **DllMain**, which must be defined with **__stdcall**, if it exists

If the /DLL or /SUBSYSTEM option is not specified, the linker selects a subsystem and entry point depending on whether **main** or **WinMain** is defined.

The functions **main**, **WinMain**, and **DllMain** are the three forms of the user-defined entry point.

When creating a managed image, the function specified with /ENTRY must have a signature of (LPVOID *var1*, DWORD *var2*, LPVOID *var3*).

For information on how to define your own DllMain entry point, see Run-Time Library Behavior .

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Advanced** property page.
4. Modify the **Entry Point** property.

To set this linker option programmatically

- See EntryPointSymbol.

See Also
Reference
Setting Linker Options
Linker Options

/ERRORREPORT (Report Internal Linker Errors)

```
/errorReport:[ none | prompt | queue | send ]
```

Remarks

Allows you to provide internal compiler error (ICE) information directly to the Visual C++ team.

For more information about /ERRORREPORT, see //errorReport (Report Internal Compiler Errors).

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For more information, see How to: Open Project Property Pages.
2. Click the **Configuration Properties** folder.
3. Click the **Linker** folder.
4. Click the **Advanced** property page.
5. Modify the **Error Reporting** property.

To set this compiler option programmatically

- See ErrorReporting.

See Also

Reference

Setting Linker Options

Linker Options

/EXPORT (Exports a Function)

```
/EXPORT:entryname[,@ordinal[,NONAME]][,DATA]
```

Remarks

With this option, you can export a function from your program so that other programs can call the function. You can also export data. Exports are usually defined in a DLL.

The *entryname* is the name of the function or data item as it is to be used by the calling program. *ordinal* specifies an index into the exports table in the range 1 through 65,535; if you do not specify *ordinal*, LINK assigns one. The **NONAME** keyword exports the function only as an ordinal, without an *entryname*.

The **DATA** keyword specifies that the exported item is a data item. The data item in the client program must be declared using **extern __declspec(dllimport)**.

There are three methods for exporting a definition, listed in recommended order of use:

1. __declspec(dllexport) in the source code
2. An EXPORTS statement in a .def file
3. An /EXPORT specification in a LINK command

All three methods can be used in the same program. When LINK builds a program that contains exports, it also creates an import library, unless an .exp file is used in the build.

LINK uses decorated forms of identifiers. The compiler decorates an identifier when it creates the .obj file. If *entryname* is specified to the linker in its undecorated form (as it appears in the source code), LINK attempts to match the name. If it cannot find a unique match, LINK issues an error message. Use the DUMPBIN tool to get the decorated names form of an identifier when you need to specify it to the linker.

Note
Do not specify the decorated form of C identifiers that are declared **__cdecl** or **__stdcall**.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Command Line** property page.
4. Type the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See Also

Reference

Setting Linker Options

Linker Options

/FIXED (Fixed Base Address)

```
/FIXED[:NO]
```

Remarks

The /FIXED option tells the operating system to load the program only at its preferred base address. If the preferred base address is unavailable, the operating system will not load the file. For more information, see /BASE (Base Address).

By default, /FIXED:NO is the default when building a DLL, and /FIXED is the default for any other project type.

When /FIXED is specified, LINK does not generate a relocation section in the program. At run time, if the operating system is unable to load the program at that address, it issues an error message and does not load the program.

Specify /FIXED:NO to generate a relocation section in the program.

Do not use /FIXED when building device drivers for Windows NT.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Command Line** property page.
4. Type the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See Also

Reference

Setting Linker Options

Linker Options

/FORCE (Force File Output)

```
/FORCE:[MULTIPLE|UNRESOLVED]
```

Remarks

The /FORCE option tells the linker to create a valid .exe file or DLL even if a symbol is referenced but not defined or is multiply defined.

The /FORCE option can take an optional argument:

- Use /FORCE:MULTIPLE to create an output file whether or not LINK finds more than one definition for a symbol.
- Use /FORCE:UNRESOLVED to create an output file whether or not LINK finds an undefined symbol. /FORCE:UNRESOLVED is ignored if the entry point symbol is unresolved.

/FORCE with no arguments implies both multiple and unresolved.

A file created with this option may not run as expected. The linker will not link incrementally when the /FORCE option is specified.

If a module is compiled with **/clr**, **/FORCE** will not create an image.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Command Line** property page.
4. Type the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See Also

Reference

Setting Linker Options

Linker Options

/FUNCTIONPADMIN (Create Hotpatchable Image)

Prepares an image for hotpatching. For more information on hotpatching, see http://www.microsoft.com/technet/security/topics/patch/patchmanagement.mspx.

```
/FUNCTIONPADMIN[:space]
```

Remarks

where,

space (optional)

The amount of padding to add to the beginning of each function, 5, 6, or 16. x86 images require five bytes of padding, x64 images require 6 bytes, and images built for the Itanium Processor Family require 16 bytes of padding at the beginning of each function.

By default, the compiler will add the correct amount of space to the image, based on the machine type of the image.

In order for the linker to produce a hotpatchable image, the .obj files must have been compiled with /hotpatch (Create Hotpatchable Image).

When you compile and link an image with a single invocation of cl.exe, **/hotpatch** implies **/functionpadmin**.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Command Line** property page.
4. Type the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See Also
Reference
Setting Linker Options
Linker Options

/HEAP (Set Heap Size)

```
/HEAP:reserve[,commit]
```

Remarks

The /HEAP option sets the size of the heap in bytes. This option is only for use when building an .exe file.

The *reserve* argument specifies the total heap allocation in virtual memory. The default heap size is 1 MB. The linker rounds up the specified value to the nearest 4 bytes.

The optional *commit* argument is subject to interpretation by the operating system. In Windows NT/Windows 2000, it specifies the amount of physical memory to allocate at a time. Committed virtual memory causes space to be reserved in the paging file. A higher *commit* value saves time when the application needs more heap space, but increases the memory requirements and possibly the startup time.

Specify the *reserve* and *commit* values in decimal or C-language notation.

This functionality is also available via a module definition file with HEAPSIZE.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **System** property page.
4. Modify the **Heap Commit Size** property.

To set this linker option programmatically

- See HeapReserveSize and HeapCommitSize.

See Also
Reference
Setting Linker Options
Linker Options

/IDLOUT (Name MIDL Output Files)

```
/IDLOUT:[path\]filename
```

Parameters

path

An absolute or relative path specification. By specifying a path, you affect only the location of an .idl file; all other files are placed in the project directory.

filename

Specifies the name of the .idl file created by the MIDL compiler. No file extension is assumed; specify *filename*.idl if you want an .idl extension.

Remarks

The /IDLOUT option specifies the name and extension of the .idl file.

The MIDL compiler is called by the Visual C++ linker when linking projects that have the module attribute.

/IDLOUT also specifies the file names of the other output files associated with the MIDL compiler:

- *filename*.tlb
- *filename*_p.c
- *filename*_i.c
- *filename*.h

filename is the parameter that you pass to /IDLOUT. If /TLBOUT is specified, the .tlb file will get its name from /TLBOUT *filename*.

If you specify neither /IDLOUT nor /TLBOUT, the linker will create vc70.tlb, vc70.idl, vc70_p.c, vc70_i.c, and vc70.h.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Embedded IDL** property page.
4. Modify the **Merge IDL Base File Name** property.

To set this linker option programmatically

- See MergedIDLBaseFileName.

See Also

Reference

Setting Linker Options
Linker Options
/IGNOREIDL (Don't Process Attributes into MIDL)
/MIDL (Specify MIDL Command Line Options)

Other Resources

Building an Attributed Program

/IGNOREIDL (Don't Process Attributes into MIDL)

```
/IGNOREIDL
```

Remarks

The /IGNOREIDL option specifies that any IDL attributes in source code should not be processed into an .idl file.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Embedded IDL** property page.
4. Modify the **Ignore Embedded IDL** property.

To set this linker option programmatically

- See IgnoreEmbeddedIDL.

See Also

Reference

Setting Linker Options
Linker Options
/IDLOUT (Name MIDL Output Files)
/TLBOUT (Name .TLB File)
/MIDL (Specify MIDL Command Line Options)

Other Resources

Building an Attributed Program

/IMPLIB (Name Import Library)

```
/IMPLIB:filename
```

Remarks

where:

filename

A user-specified name for the import library. It replaces the default name.

Remarks

The /IMPLIB option overrides the default name for the import library that LINK creates when it builds a program that contains exports. The default name is formed from the base name of the main output file and the extension .lib. A program contains exports if one or more of the following are specified:

- The __declspec(dllexport) keyword in the source code
- EXPORTS statement in a .def file
- An /EXPORT specification in a LINK command

LINK ignores /IMPLIB when an import library is not being created. If no exports are specified, LINK does not create an import library. If an export file is used in the build, LINK assumes that an import library already exists and does not create one. For information on import libraries and export files, see LIB Reference.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Advanced** property page.
4. Modify the **Import Library** property.

To set this linker option programmatically

- See ImportLibrary.

See Also
Reference
Setting Linker Options
Linker Options

/INCLUDE (Force Symbol References)

```
/INCLUDE:symbol
```

Remarks

where:

symbol

Specifies a symbol to be added to the symbol table.

Remarks

The /INCLUDE option tells the linker to add a specified symbol to the symbol table.

To specify multiple symbols, type a comma (,), a semicolon (;), or a space between the symbol names. On the command line, specify /INCLUDE:*symbol* once for each symbol.

The linker resolves *symbol* by adding the object that contains the symbol definition to the program. This feature is useful for including a library object that otherwise would not be linked to the program.

Specifying a symbol with this option overrides the removal of that symbol by /OPT:REF.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Input** property page.
4. Modify the **Force Symbol References** property.

To set this linker option programmatically

- See ForceSymbolReferences.

See Also

Reference

Setting Linker Options

Linker Options

/INCREMENTAL (Link Incrementally)

```
/INCREMENTAL[:NO]
```

Remarks

The /INCREMENTAL option controls how the linker handles incremental linking.

By default, the linker runs in incremental mode. To override a default incremental link, specify /INCREMENTAL:NO.

An incrementally linked program is functionally equivalent to a program that is nonincrementally linked. However, because it is prepared for subsequent incremental links, an incrementally linked executable (.exe) file or dynamic-link library (DLL):

- Is larger than a nonincrementally linked program because of padding of code and data. (Padding allows the linker to increase the size of functions and data without recreating the .exe file.)
- May contain jump thunks to handle relocation of functions to new addresses.

Note
To ensure that your final release build does not contain padding or thunks, link your program nonincrementally.

To link incrementally regardless of the default, specify /INCREMENTAL. When this option is selected, the linker issues a warning if it cannot link incrementally, and then links the program nonincrementally. Certain options and situations override /INCREMENTAL.

Most programs can be linked incrementally. However, some changes are too great, and some options are incompatible with incremental linking. LINK performs a full link if any of the following options are specified:

- Link Incrementally is not selected (/INCREMENTAL:NO)
- /OPT:REF is selected
- /OPT:ICF is selected
- /ORDER is selected

/INCREMENTAL is implied when /DEBUG is specified.

Additionally, LINK performs a full link if any of the following situations occur:

- The incremental status (.ilk) file is missing. (LINK creates a new .ilk file in preparation for subsequent incremental linking.)
- There is no write permission for the .ilk file. (LINK ignores the .ilk file and links nonincrementally.)
- The .exe or .dll output file is missing.
- The timestamp of the .ilk, .exe, or .dll is changed.
- A LINK option is changed. Most LINK options, when changed between builds, cause a full link.
- An object (.obj) file is added or omitted.
- An object that was compiled with the /Yu /Z7 option is changed.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **General** property page.
4. Modify the **Enable Incremental Linking** property.

To set this linker option programmatically

- See LinkIncremental.

See Also

Reference

Setting Linker Options

Linker Options

/KEYCONTAINER (Specify a Key Container to Sign an Assembly)

```
/KEYCONTAINER:name
```

Remarks

where,

name

Container that contains the key. Place the string in double quotation marks (" ") if it contains a space.

Remarks

The linker creates a signed assembly by inserting a public key into the assembly manifest and signing the final assembly with the private key. To generate a key file, type sn -k *file* at the command line. **sn -i** installs the key pair into a container.

If you compile with /LN, the name of the key file is held in the module and incorporated into the assembly that is created when you compile an assembly that includes an explicit reference to the module, via #using, or when linking with /ASSEMBLYMODULE.

You can also pass your encryption information to the compiler with /KEYFILE. Use /DELAYSIGN if you want a partially signed assembly. See Strong Name Assemblies (Assembly Signing) for more information on signing an assembly.

Other linker options that affect assembly generation are:

- /ASSEMBLYDEBUG
- /ASSEMBLYLINKRESOURCE
- /ASSEMBLYMODULE
- /ASSEMBLYRESOURCE
- /NOASSEMBLY

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Command Line** property page.
4. Type the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See Also
Reference
Setting Linker Options
Linker Options

/KEYFILE (Specify Key or Key Pair to Sign an Assembly)

```
/KEYFILE:filename
```

Remarks

where:

filename

File that contains the key. Place the string in double quotation marks (" ") if it contains a space.

Remarks

The linker inserts the public key into the assembly manifest and then signs the final assembly with the private key. To generate a key file, type sn -k *file* at the command line. A signed assembly is said to have a strong name.

If you compile with /LN, the name of the key file is held in the module and incorporated into the assembly that is created when you compile an assembly that includes an explicit reference to the module, via #using, or when linking with /ASSEMBLYMODULE.

You can also pass your encryption information to the linker with /KEYCONTAINER. Use /DELAYSIGN if you want a partially signed assembly. See Strong Name Assemblies (Assembly Signing) for more information on signing an assembly.

In case both **/KEYFILE** and **/KEYCONTAINER** are specified (either by command line option or by custom attribute), the linker will first try the key container. If that succeeds, then the assembly is signed with the information in the key container. If the linker does not find the key container, it will try the file specified with /KEYFILE. If that succeeds, the assembly is signed with the information in the key file and the key information will be installed in the key container (similar to sn -i) so that on the next compilation, the key container will be valid.

Note that a key file might contain only the public key.

See Creating and Using Strong-Named Assemblies for more information on signing an assembly.

Other linker options that affect assembly generation are:

- /ASSEMBLYDEBUG
- /ASSEMBLYLINKRESOURCE
- /ASSEMBLYMODULE
- /ASSEMBLYRESOURCE
- /NOASSEMBLY

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Command Line** property page.
4. Type the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See Also
Reference
Setting Linker Options
Linker Options

/LARGEADDRESSAWARE (Handle Large Addresses)

```
/LARGEADDRESSAWARE[:NO]
```

Remarks

The /LARGEADDRESSAWARE option tells the linker that the application can handle addresses larger than 2 gigabytes. By default, /LARGEADDRESSAWARE:NO is enabled if /LARGEADDRESSAWARE is not otherwise specified on the linker line.

If an application was linked with /LARGEADDRESSAWARE, DUMPBIN /HEADERS will display information to that effect.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **System** property page.
4. Modify the **Enable Large Addresses** property.

To set this linker option programmatically

- See LargeAddressAware.

See Also

Reference

Setting Linker Options

Linker Options

/LIBPATH (Additional Libpath)

```
/LIBPATH:dir
```

Remarks

where:

dir

Specifies a path that the linker will search before it searches the path specified in the LIB environment option.

Remarks

Use the /LIBPATH option to override the environment library path. The linker will first search in the path specified by this option, and then search in the path specified in the LIB environment variable. You can specify only one directory for each /LIBPATH option you enter. If you want to specify more than one directory, you must specify multiple /LIBPATH options. The linker will then search the specified directories in order.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **General** property page.
4. Modify the **Additional Library Directories** property.

To set this linker option programmatically

- See AdditionalLibraryDirectories.

See Also

Reference

Setting Linker Options

Linker Options

/LTCG (Link-time Code Generation)

```
/LTCG[:NOSTATUS|:STATUS|:PGINSTRUMENT|:PGOPTIMIZE|:PGUPDATE]
```

Remarks

where:

:NOSTATUS | :STATUS (optional)

Specifies whether the linker should display a progress indicator showing what percentage of the link is complete. The default is to not display this status information.

:PGINSTRUMENT (optional)

Specifies that the linker should output a .pgd file in preparation for instrumented test runs on the application. You can optionally specify /PGD to create the .pgd file with a nondefault name or location.

The data collected from the instrumented runs will be used to create an optimized image. For more information, see Profile Guided Optimization. The short form of this option is /LTCG:PGI.

:PGOPTIMIZE (optional)

Specifies that the linker should use the profile data created after running the instrumented binary to create an optimized image. All input files must be identical to the files that were specified with /LTCG:PGI. For more information, see Profile Guided Optimization. The short form of this option is /LTCG:PGO.

:PGUPDATE (optional)

Allows list of input files to be added or modified from what was specified in the :PGINSTRUMENT phase. However, any new input files will not be optimized with profile-guided optimizations and changed portions of a modified input file that invalidate profile data collected during the instrumentation phase for that code will not be optimized with profile-guided optimizations. For more information, see Profile Guided Optimization. The short form of this option is /LTCG:PGU.

Remarks

The /LTCG option tells the linker to call the compiler and perform whole program optimization. You can also do profile guided optimization. For more information, see Profile Guided Optimization.

With the following exceptions, you cannot add additional linker options to the /LTCG:PGOPTIMIZE or /LTCG:PGUPDATE run that were not specified in the /LTCG:PGINSTRUMENT run:

- /BASE
- /FIXED
- /LTCG
- /MAP
- /MAPINFO
- /NOLOGO
- /OUT
- /PGD
- /PDB
- /PDBSTRIPPED
- /STUB
- /VERBOSE

Any linker options specified to /LTCG:PGINSTRUMENT do not have to be specified to /LTCG:PGOPTIMIZE; they are implied.

The rest of this topic will only discuss /LTCG in terms of link-time code generation.

/LTCG is implied with /GL.

The linker invokes link-time code generation if it is passed a module that was compiled with **/GL** or an MSIL module (see .netmodule Files as Linker Input for more information). If you do not explicitly specify **/LTCG** when passing **/GL** or MSIL modules to the linker, the linker will eventually detect this and restart the link with **/LTCG**. Explicitly specify **/LTCG** when passing **/GL** and MSIL modules to the linker for the fastest possible build performance.

/LTCG is not valid for use with /INCREMENTAL.

When /LTCG is used with either /Og, /O1, /O2, or /Ox, the following optimizations are performed:

- Cross-module inlining
- Interprocedural register allocation (64-bit operating systems only)
- Custom calling convention (x86 only)
- Small TLS displacement (x86 only)
- Stack double alignment (x86 only)
- Improved memory disambiguation (better interference information for global variables and input parameters)

Using /LTCG and /Ogt will result in double-alignment optimization.

If /LTCG and /Ogs are specified, double alignment will not be performed. If most of the functions in an application are compiled for speed, with a few functions compiled for size (for example, by using the optimize pragma), the compiler will double align these functions that are optimized for size if they call functions that need double alignment.

If the compiler can identify all the call sites of a function, the compiler will ignore explicit calling-convention modifiers on a function and try to optimize the function's calling convention:

- pass parameters in registers
- reorder parameters for alignment
- remove unused parameters

If a function is called via function pointer, or if a function make be called outside of a module compiled with /GL, the compiler will not attempt to optimize a function's calling convention.

Note
If you use /LTCG and redefine mainCRTStartup, your application can have unpredictable behavior relating to user code that e xecutes before global objects are initialized. There are three ways to address this issue: do not redefine mainCRTStartup, do n ot compile the file containing mainCRTStartup with /LTCG, or initialize global variables and objects statically, if possible.

/LTCG and MSIL Modules

Modules compiled with /GL and /clr can be used as input to the linker when **/LTCG** is specified:

- **/LTCG** can accept native object files; mixed native/managed object files (compiled with **/clr**), pure object files (compiled with **/clr:pure**), and safe object files (compiled with **/clr:safe**)
- **/LTCG** can accept safe .netmodules, which can be created with **/clr:safe /LN** in Visual C++ and **/target:module** with any other Visual Studio compiler. .netmodules produced with **/clr** or **/clr:pure** are not accepted by **/LTCG**.
- /LTCG:PGI does not accept native modules compiled with /GL and /clr, or pure modules (produced with /clr:pure)

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Modifying Project Settings.
2. Click the **Configuration Properties** folder.
3. Click the **General** property page.
4. Modify the **Whole Program Optimization** property.

You can also apply **/LTCG** to specific builds by choosing Profile Guided Optimization from the Build menu, or by right clicking on the project name in Solution Explorer and selecting one of the Profile Guided Optimization options.

To set this compiler option programmatically

- See WholeProgramOptimization.

See Also

Reference

Setting Linker Options

Linker Options

/MACHINE (Specify Target Platform)

```
/MACHINE:{ARM|EBC|IA64|MIPS|MIPS16|MIPSFPU|MIPSFPU16|          SH4|THUMB|X64|X86}
```

Remarks

The /MACHINE option specifies the target platform for the program.

Usually, you do not need to specify the /MACHINE option. LINK infers the machine type from the .obj files. However, in some circumstances, LINK cannot determine the machine type and issues a linker tools error LNK1113. If such an error occurs, specify /MACHINE.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Advanced** property page.
4. Modify the **Target Machine** property.

To set this linker option programmatically

- See TargetMachine.

See Also
Reference
Setting Linker Options
Linker Options

/MANIFEST (Create Side-by-Side Assembly Manifest)

```
/MANIFEST[:NO]
```

Remarks

/MANIFEST specifies that the linker should create a side-by-side manifest file. For more information on manifest files, see Manifest Files Reference.

The default is /MANIFEST.

If /MANIFEST is not specified and the linker is passed /MANIFESTDEPENDENCY comments, a manifest file will be created. A manifest file will not be created if /MANIFEST:NO is specified.

If you specify /MANIFEST, the name of the manifest file will be the same as the name of your output file, with .manifest appended to the file name. For example, if your output file name is MyFile.exe, the manifest file name will be MyFile.exe.manifest. If you specify /MANIFESTFILE:*name*, the name of the manifest will be what you specify in *name*.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Expand the **Configuration Properties** node.
3. Expand the **Linker** node.
4. Select the **Manifest File** property page.
5. Modify the **Generate Manifest** property.

To set this linker option programmatically

- See GenerateManifest.

See Also

Reference

Setting Linker Options

Linker Options

/MANIFESTDEPENDENCY (Specify Manifest Dependencies)

```
/MANIFESTDEPENDENCY:manifest_dependency
```

Remarks

/MANIFESTDEPENDENCY lets you specify attributes that will be placed in the <dependency> section of the manifest file.

See /MANIFEST (Create Side-by-Side Assembly Manifest) for information on how to create a manifest file.

For more information on the <dependency> section of the manifest file, see Publisher Configuration Files.

/MANIFESTDEPENDENCY information can be passed to the linker in one of two ways:

- Directly on the command line (or in a response file) with /MANIFESTDEPENDENCY.
- Via the comment pragma.

The following example shows a /MANIFESTDEPENDENCY comment passed via pragma,

```
#pragma comment(linker, "\"/manifestdependency:type='Win32' name='Test.Research.SampleAssem
bly' version='6.0.0.0' processorArchitecture='X86' publicKeyToken='0000000000000000' langua
ge='*'\"")
```

which results in the following entry in the manifest file:

```
  <dependency>
    <dependentAssembly>
      <assemblyIdentity type='Win32' name='Test.Research.SampleAssembly' version='6.0.0.0'
processorArchitecture='X86' publicKeyToken='0000000000000000' language='*' />
    </dependentAssembly>
  </dependency>
```

The same /MANIFESTDEPENDENCY comments can be passed at the command line as follows:

```
"/manifestdependency:type='Win32' name='Test.Research.SampleAssembly' version='6.0.0.0' pro
cessorArchitecture='X86' publicKeyToken='0000000000000000' language='*'\"
```

The linker will collect /MANIFESTDEPENDENCY comments, eliminate duplicate entries, and then add the resulting XML string to the manifest file. If the linker finds conflicting entries, the manifest file will become corrupt and the application will fail to launch (an entry may be added to the event log, indicating the source of the failure).

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Expand the **Configuration Properties** node.
3. Expand the **Linker** node.
4. Select the **Manifest File** property page.
5. Modify the **Additional Manifest Dependencies** property.

To set this linker option programmatically

- See AdditionalManifestDependencies.

See Also

Reference

Setting Linker Options

Linker Options

/MANIFESTFILE (Name Manifest File)

```
/MANIFESTFILE:filename
```

Remarks

/MANIFESTFILE lets you change the default name of the manifest file. The default name of the manifest file is the file name with .manifest appended.

/MANIFESTFILE will have no effect if you do not also link with /MANIFEST.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Expand the **Configuration Properties** node.
3. Expand the **Linker** node.
4. Select the **Manifest File** property page.
5. Modify the **Manifest File** property.

To set this linker option programmatically

- See ManifestFile.

See Also

Reference

Setting Linker Options

Linker Options

/MAP (Generate Mapfile)

```
/MAP[:filename]
```

Remarks

where:

filename

A user-specified name for the mapfile. It replaces the default name.

Remarks

The /MAP option tells the linker to create a mapfile.

By default, the linker names the mapfile with the base name of the program and the extension .map. The optional *filename* allows you to override the default name for a mapfile.

A mapfile is a text file that contains the following information about the program being linked:

- The module name, which is the base name of the file
- The timestamp from the program file header (not from the file system)
- A list of groups in the program, with each group's start address (as *section*:*offset*), length, group name, and class
- A list of public symbols, with each address (as *section*:*offset*), symbol name, flat address, and .obj file where the symbol is defined
- The entry point (as *section*:*offset*)

The /MAPINFO option specifies additional information to be included in the mapfile.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Debug** property page.
4. Modify the **Generate Map File** property.

To set this linker option programmatically

- See GenerateMapFile and MapFileName.

See Also
Reference
Setting Linker Options
Linker Options

/MAPINFO (Include Information in Mapfile)

```
/MAPINFO:EXPORTS
```

Remarks

The /MAPINFO option tells the linker to include the specified information in a mapfile, which is created if you specify the /MAP option. EXPORTS tells the linker to include exported functions.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Debug** property page.
4. Modify of the **Map Exports** properties:

To set this linker option programmatically

- See MapExports.

See Also
Reference
Setting Linker Options
Linker Options

/MERGE (Combine Sections)

```
/MERGE:from=to
```

Remarks

The /MERGE option combines the first section (*from*) with the second section (*to*), naming the resulting section *to*. For example, `/merge:.rdata=.text`.

If the second section does not exist, LINK renames the section *from* as *to*.

The /MERGE option is useful for creating VxDs and overriding the compiler-generated section names.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Advanced** property page.
4. Modify the **Merge Sections** property.

To set this linker option programmatically

- See MergeSections.

See Also

Reference

Setting Linker Options

Linker Options

/MIDL (Specify MIDL Command Line Options)

```
/MIDL:@file
```

Remarks

where:

file

The name of the file that that contains MIDL command line options.

Remarks

All options for the conversion of an IDL file to a TLB file must be given in *file*; MIDL command-line options cannot be specified on the linker's command line. If /MIDL is not specified, the MIDL compiler will be invoked with only the IDL file name and no other options.

The file should contain one MIDL command-line option per line.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Embedded IDL** property page.
4. Modify the **MIDL Commands** property.

To set this linker option programmatically

- See MidlCommandFile.

See Also

Reference

Setting Linker Options
Linker Options
/IDLOUT (Name MIDL Output Files)
/IGNOREIDL (Don't Process Attributes into MIDL)
/TLBOUT (Name .TLB File)

Other Resources

Building an Attributed Program

/NOASSEMBLY (Create a MSIL Module)

```
/NOASSEMBLY
```

Remarks

The /NOASSEMBLY option tells the linker to create an image for the current output file without a .NET Framework assembly. An MSIL output file without an assembly manifest is called a module.

By default, an assembly is created. You can also use the /LN (Create MSIL Module) compiler option to create a module.

Other linker options that affect assembly generation are:

- /ASSEMBLYDEBUG
- /ASSEMBLYLINKRESOURCE
- /ASSEMBLYMODULE
- /ASSEMBLYRESOURCE
- /DELAYSIGN
- /KEYFILE
- /KEYCONTAINER

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Advanced** property page.
4. Modify the **Turn Off Assembly Generation** property.

To set this linker option programmatically

- See TurnOffAssemblyGeneration.

See Also
Reference
Setting Linker Options
Linker Options

/NODEFAULTLIB (Ignore Libraries)

```
/NODEFAULTLIB[:library]
```

Remarks

where:

library

A library that you want the linker to ignore when it resolves external references.

Remarks

The /NODEFAULTLIB option tells the linker to remove one or more default libraries from the list of libraries it searches when resolving external references.

To create an .obj file that does not contain references to default libraries, use /Zl (Omit Default Library Name).

By default, /NODEFAULTLIB removes all default libraries from the list of libraries it searches when resolving external references. The optional *library* parameter lets you remove a specified library or libraries from the list of libraries it searches when resolving external references. Specify one /NODEFAULTLIB option for each library you want to exclude.

The linker resolves references to external definitions by searching first in libraries that you explicitly specify, then in default libraries specified with the /DEFAULTLIB option, and then in default libraries named in .obj files.

/NODEFAULTLIB:*library* overrides /DEFAULTLIB:*library* when the same *library* name is specified in both.

If you use /NODEFAULTLIB, for example, to build your program without the C run-time library, you may have to also use /ENTRY to specify the entry point (function) in your program. For more information, see C Run-Time Libraries.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Input** property page.
4. Select the **Ignore All Default Libraries** property or specify a list of the libraries you want to ignore in the **Ignore Specific Library** property. The **Command Line** property page will show the effect of the changes you make to these properties.

To set this linker option programmatically

- See IgnoreDefaultLibraryNames and IgnoreAllDefaultLibraries.

See Also

Reference

Setting Linker Options

Linker Options

/NOENTRY (No Entry Point)

```
/NOENTRY
```

Remarks

The /NOENTRY option is required for creating a resource-only DLL.

Use this option to prevent LINK from linking a reference to **_main** into the DLL.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Advanced** property page.
4. Modify the **Resource Only DLL** property.

To set this linker option programmatically

- See ResourceOnlyDLL.

See Also

Reference

Setting Linker Options

Linker Options

/NOLOGO (Suppress Startup Banner) (Linker)

```
/NOLOGO
```

Remarks

The /NOLOGO option prevents display of the copyright message and version number.

This option also suppresses echoing of command files. For details, see LINK Command Files.

By default, this information is sent by the linker to the Output window. On the command line, it is sent to standard output and can be redirected to a file.

To set this linker option in the Visual Studio development environment

- This option should only be used from the command line.

To set this linker option programmatically

- This linker option cannot be changed programmatically.

See Also

Reference

Setting Linker Options

Linker Options

/NXCOMPAT (Compatible with Data Execution Prevention)

Indicates that an executable was tested to be compatible with the Windows Data Execution Prevention feature.

```
/NXCOMPAT[:NO]
```

Remarks

/NXCOMPAT is on by default if a component requires Windows Vista (/SUBSYSTEM 6.0 and greater). See /SUBSYSTEM (Specify Subsystem) for more information.

/NXCOMPAT:NO can be used to explicitly specify an executable as not compatible with Data Execution Prevention.

For more information on Data Execution Prevention, see:

- http://support.microsoft.com/default.aspx?scid=kb;en-us;875352
- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/data_execution_prevention.asp (also available in the MSDN CD library on Data Execution Prevention)
- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/xpehelp/html/xetskconfiguringdataexecutionpreventionsettingsofrun-timeimage.asp
- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/xpehelp/html/xeconNon-ExecuteMemoryProtection.asp

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Command Line** property page.
4. Type the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See Also
Reference
Setting Linker Options
Linker Options

/OPT (Optimizations)

Controls the optimizations that LINK performs during a build.

```
/OPT:{REF | NOREF}
/OPT:{ICF[=iterations] | NOICF}
/OPT:{WIN98 | NOWIN98}
```

Arguments

REF | **NOREF**

/OPT:REF eliminates functions and/or data that are never referenced while **/OPT:NOREF** keeps functions and/or data that are never referenced.

LINK removes unreferenced packaged functions by default. An object contains packaged functions (COMDATs) if it has been compiled with the /Gy option. This optimization is called transitive COMDAT elimination. To override this default and keep unreferenced COMDATs in the program, specify /OPT:NOREF. You can use the /INCLUDE option to override the removal of a specific symbol.

If /DEBUG is specified, the default for **/OPT** is **NOREF** (otherwise, it is **REF**), and all functions are preserved in the image. To override this default and optimize a debugging build, specify **/OPT:REF**. The **/OPT:REF** option disables incremental linking.

You have to explicitly mark data as a COMDAT; use __declspec(selectany).

If **/OPT:REF** is specified, **/OPT:ICF** is on by default. If you want **/OPT:REF** but not **/OPT:ICF**, you must specify the following:

```
link /opt:ref /opt:noicf
```

Specifying **/OPT:ICF** does not activate the **/OPT:REF** option.

ICF[= *iterations* **] | NOICF**

Use **/OPT:ICF[=***iterations***]** to perform identical COMDAT folding. Redundant COMDATs can be removed from the linker output. *iterations* specifies the number of times to traverse the symbols for duplicates. The default number of iterations is two. Additional iterations may locate more duplicates uncovered through folding in the previous iteration.

Note that there is a difference in linker behavior when **ICF** is in effect by default with **/OPT:REF** explicitly specified and when you explicitly specify **/OPT:REF,ICF**. The default **ICF** with **/OPT:REF** does not fold read-only data. This includes any .rdata, .pdata, and .xdata. However, the default **ICF** with **/OPT:REF** results in fewer functions folded when producing images for Itanium and x64 because functions in these modules have more read only data dependency, such as .pdata and .xdata. To get full **ICF**, explicitly specify **/OPT:ICF**.

Functions are placed in COMDATs with the **/Gy** compiler option and const data is placed in COMDATs. See selectany for an example of how to specify data for folding.

ICF is on by default if **REF** is on and needs to be explicitly turned on in a debug build. It is possible to specify **NOICF** if **REF** is specified.

Note
/OPT:ICF can result in the same address being assigned to different functions or read only data members (const variables compiled with **/Gy**). So, **/OPT:ICF** can break a program that depends on the address of functions or read-only data membe rs being different. See /Gy (Enable Function-Level Linking) for more information.

WIN98 | NOWIN98

WIN98 and **NOWIN98** control the section alignment in the final image. For Windows 98 applications, it is optimal to align sections on a 4K boundary to improve load time (allows Windows 98 memory manager to cache executable images with a minimum of wasted space). This is on by default in the linker, so you need to specify **/OPT:NOWIN98** to get a trimmed-down (but slower on Windows 98) version of the application.

WIN98 is on by default. **WIN98** is not on when:

- /ALIGN is used.
- /MACHINE does not target x86.
- /SUBSYSTEM specifies something other than WINDOWS or CONSOLE.

/OPT:WIN98 is not enabled by default for images that would grow (according to the average growth equations, below) by more than 25 percent. In other words, **/OPT:WIN98** would not be enabled for smaller images. You should enable **/OPT:WIN98** explicitly to ensure that you are not affected by this tuning. Specify **/OPT:NOWIN98** to get a smaller (but slower on Windows 98) version of your application.

The enhancements in Windows 98 only work when the sections in a portable executable image begin on a page boundary. The **/OPT:WIN98** option performs the necessary file alignment.

If you are building components that run only on Windows NT or Windows 2000, you should use **/OPT:NOWIN98**.

This change does not impact loading of images or the working set of the process. The only impact is to the on-disk size.

The following helps you compute the average growth of an image using **/OPT:WIN98**:

- The average wasted space for 4096-byte file alignment can be characterized by: *count-of-sections-in-image* * 4096/2
- The average wasted space for the current 512-byte file alignment is: *count-of-sections-in-image* * 512/2

The growth is therefore:

- count-of-bytes-Growth = *count-of-sections-in-image* * (4096/2 - 512/2)

  or, simplified,

  count-of-bytes-Growth = *count-of-sections-in-image* * 1792

  However, this does not take into account the fact that the image header must be padded to the section alignment. Since the header is always 512 byes or less, the extra growth is a constant of 4096 - 512, or 3584.
- average count-of-bytes-Growth = *count-of-sections-in-image* * 1792 + 3584
- maximum count-of-bytes-Growth = *count-of-sections-in-image* * (4096 - 512 + 3584

To get the count of sections, use the DUMPBIN tool on an executable file. The summary will give you a list of sections in that image. Typically, you will see from 3 to 5 sections added to that value.

The only time you should not use **/OPT:WIN98** is when your portable executable image is very small. Even if an image is slated for downloads, the wasted space is zero-filled and compresses well.

Remarks

Optimizations generally decrease the image size and increase the program speed at a cost of increased link time.

You can use the /VERBOSE option to see the functions removed by **/OPT:REF** and the functions that are folded by **/OPT:ICF**.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Optimization** property page.
4. Modify one of the following properties:
   - **Enable COMDAT Folding**
   - **Optimize for Windows98**
   - **References**

To set this linker option programmatically

- See EnableCOMDATFolding, OptimizeForWindows98, and OptimizeReferences properties.

See Also

Reference

Setting Linker Options

Linker Options

/ORDER (Put Functions in Order)

```
/ORDER:@filename
```

Parameters
filename

A text file specifying the linking order for COMDAT functions.

Remarks

The /ORDER option tells LINK to optimize your program by placing certain COMDATs into the image in a predetermined order. LINK places the functions in the specified order within each section in the image.

Specify the order in *filename*, which is a text file (response file) that lists the COMDATs in the order you want to link them. Each line in *filename* contains the name of one COMDAT. An object contains COMDATs if it has been compiled with the /Gy option. Function names are case sensitive.

LINK uses decorated forms of identifiers. The compiler decorates an identifier when it creates the .obj file. Use the DUMPBIN tool to get the decorated form of an identifier when you need to specify it to the linker. For more information on decorated names, see Decorated Names.

If more than one /ORDER specification is used, the last one specified takes effect.

Ordering allows you to optimize your program's paging behavior through swap tuning by grouping a function with the functions it calls. You can also group frequently called functions together. These techniques increase the probability that a called function is in memory when it is needed and will not have to be paged from disk.

The linker will prepend an underscore (_) to every decorated name in *filename* unless the name starts with a question mark (?) or at sign (@). For example, if an object file contains `extern "C" int func(int)` and `int main(void)`, DUMPBIN /SYMBOLS will list these decorated names:

```
009 00000000 SECT3  notype ()    External     | _func
00A 00000008 SECT3  notype ()    External     | _main
```

However, the name specified in the order file should be `func` and `main`.

The /ORDER option disables incremental linking.

Note
LINK cannot order static functions because static function names are not public symbol names. When /ORDER is specified, linker warning LNK4037 is generated for each symbol, in the order file, that is either static or not found.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Optimization** property page.
4. Modify the **Function Order** property.

To set this linker option programmatically

- See FunctionOrder.

See Also
Reference
Setting Linker Options
Linker Options

/OUT (Output File Name)

```
/OUT:filename
```

Remarks

where:

filename

A user-specified name for the output file. It replaces the default name.

Remarks

The /OUT option overrides the default name and location of the program that the linker creates.

By default, the linker forms the file name using the base name of the first .obj file specified and the appropriate extension (.exe or .dll).

This option the default base name for a .mapfile or import library. For details, see Generate Mapfile (/MAP) and /IMPLIB.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **General** property page.
4. Modify the **Output File** property.

To set this linker option programmatically

- See OutputFile.

See Also
Reference
Setting Linker Options
Linker Options

/PDB (Use Program Database)

```
/PDB:filename
```

Remarks

where:

filename

A user-specified name for the program database (PDB) that the linker creates. It replaces the default name.

Remarks

By default, when /DEBUG is specified, the linker creates a program database (PDB) which holds debugging information. The default file name for the PDB has the base name of the program and the extension .pdb.

Use /PDB:*filename* to specify the name of the PDB file. If /DEBUG is not specified, the /PDB option is ignored.

A PDB file can be up to 2GB.

For more information, see .pdb Files as Linker Input.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Debug** property page.
4. Modify the **Generate Program Database File** property.

To set this linker option programmatically

- See ProgramDatabaseFile.

See Also

Reference

Setting Linker Options

Linker Options

/PDBSTRIPPED (Strip Private Symbols)

```
/PDBSTRIPPED:pdb_file_name
```

Remarks

where:

pdb_file_name

A user-specified name for the stripped program database (PDB) that the linker creates.

Remarks

The /PDBSTRIPPED option creates a second program database (PDB) file when you build your program image with any of the compiler or linker options that generate a PDB file (/DEBUG, /Z7, /Zd, or /Zi). This second PDB file omits symbols that you would not want to ship to your customers. The second PDB file will only contain:

- Public symbols
- The list of object files and the portions of the executable to which they contribute
- Frame pointer optimization (FPO) debug records used to traverse the stack

The stripped PDB file will not contain:

- Type information
- Line number information
- Per-object file CodeView symbols such as those for functions, locals, and static data

The full PDB file will still be generated when you use /PDBSTRIPPED.

If you do not create a PDB file, /PDBSTRIPPED is ignored.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Debug** property page.
4. Modify the **Strip Private Symbols** property.

To set this linker option programmatically

- See StripPrivateSymbols.

See Also
Reference
Setting Linker Options
Linker Options

/PGD (Specify Database for Profile-Guided Optimizations)

/PGD:*filename*

Remarks

where:

filename

Specifies the name of the .pgd file that will be used to hold information about the running program.

Remarks

When using /LTCG:PGINSTRUMENT, use /PGD to specify a nondefault name or location for the .pgd file. If you do not specify /PGD, the .pgd file name will be the same as the output file (.exe or .dll) name and will be created in the same directory from which the link was invoked.

When using /LTCG:PGOPTIMIZE, use /PGD to specify the name of the .pgd file to use to create the optimized image.

For more information, see Profile Guided Optimization.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Expand the **Configuration Properties** node.
3. Expand the **Linker** node.
4. Select the **Optimization** property page.
5. Modify the **Profile Guided Database** property.

To set this linker option programmatically

- See ProfileGuidedDatabase.

See Also
Reference
Setting Linker Options
Linker Options

/PROFILE (Performance Tools Profiler)

Produces an output file that can be used with the Performance Tools profiler.

```
/PROFILE
```

Remarks

/PROFILE implies the following linker options:

- /OPT:REF
- /OPT:NOICF
- /INCREMENTAL:NO
- /FIXED:NO

/PROFILE causes the linker to generate a relocation section in the program image. A relocation section allows the profiler to transform the program image to get profile data.

/PROFILE is only available only in Enterprise (team development) versions. For more information on PREfast, see Code Analysis for C/C++ Overview.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Expand the **Configuration Properties** node.
3. Expand the **Linker** node.
4. Select the **Advanced** property page.
5. Modify the **Profile** property.

To set this linker option programmatically

- See Profile.

See Also
Reference
Setting Linker Options
Linker Options

/RELEASE (Set the Checksum)

```
/RELEASE
```

Remarks

The /RELEASE option sets the Checksum in the header of an .exe file.

The operating system requires the Checksum for device drivers. Set the Checksum for release versions of your device drivers to ensure compatibility with future operating systems.

The /RELEASE option is set by default when the /SUBSYSTEM:NATIVE option is specified.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Advanced** property page.
4. Modify the **Set Checksum** property.

To set this linker option programmatically

- See SetChecksum.

See Also
Reference
Setting Linker Options
Linker Options

/SAFESEH (Image has Safe Exception Handlers)

```
/SAFESEH[:NO]
```

When **/SAFESEH** is specified, the linker will only produce an image if it can also produce a table of the image's safe exception handlers. This table specifies for the operating system which exception handlers are valid for the image.

/SAFESEH is only valid when linking for x86 targets. **/SAFESEH** is not supported for platforms that already have the exception handlers noted. For example, on x64 and Itanium, all exception handlers are noted in the PDATA. ML64.exe has support for adding annotations that emit SEH information (XDATA and PDATA) into the image, allowing you to unwind through ml64 functions. See MASM for x64 (ml64.exe) for more information.

If **/SAFESEH** is not specified, the linker will produce an image with a table of safe exceptions handlers if all modules are compatible with the safe exception handling feature. If any modules were not compatible with safe exception handling feature, the resulting image will not contain a table of safe exception handlers. If /SUBSYSTEM specifies WINDOWSCE or one of the EFI_* options, the linker will not attempt to produce an image with a table of safe exceptions handlers, as neither of those subsystems can make use of the information.

If **/SAFESEH:NO** is specified, the linker will not produce an image with a table of safe exceptions handlers even if all modules are compatible with the safe exception handling feature.

The most common reason for the linker not to be able to produce an image is because one or more of the input files (modules) to the linker was not compatible with the safe exception handlers feature. A common reason for a module to not be compatible with safe exception handlers is because it was created with a compiler from a previous version of Visual C++.

You can also register a function as a structured exception handler by using .SAFESEH.

In Microsoft Visual C++ 2005 it is not possible to mark an existing binary as having safe exception handlers (or no exception handlers); information on safe exception handling must be added at build time.

The linker's ability to build a table of safe exception handlers depends on the application using the C runtime library. If you link with /NODEFAULTLIB and you want a table of safe exception handlers, you need to supply a load config struct (such as can be found in loadcfg.c CRT source file) that contains all the entries defined for Visual C++. For example:

```
#include <windows.h>
extern DWORD_PTR __security_cookie;  /* /GS security cookie */

/*
 * The following two names are automatically created by the linker for any
 * image that has the safe exception table present.
*/

extern PVOID __safe_se_handler_table[]; /* base of safe handler entry table */
extern BYTE  __safe_se_handler_count;  /* absolute symbol whose address is
                                          the count of table entries */
typedef struct {
    DWORD       Size;
    DWORD       TimeDateStamp;
    WORD        MajorVersion;
    WORD        MinorVersion;
    DWORD       GlobalFlagsClear;
    DWORD       GlobalFlagsSet;
    DWORD       CriticalSectionDefaultTimeout;
    DWORD       DeCommitFreeBlockThreshold;
    DWORD       DeCommitTotalFreeThreshold;
    DWORD       LockPrefixTable;            // VA
    DWORD       MaximumAllocationSize;
    DWORD       VirtualMemoryThreshold;
    DWORD       ProcessHeapFlags;
    DWORD       ProcessAffinityMask;
    WORD        CSDVersion;
    WORD        Reserved1;
    DWORD       EditList;                   // VA
    DWORD_PTR   *SecurityCookie;
```

```
    PVOID       *SEHandlerTable;
    DWORD       SEHandlerCount;
} IMAGE_LOAD_CONFIG_DIRECTORY32_2;

const IMAGE_LOAD_CONFIG_DIRECTORY32_2 _load_config_used = {
    sizeof(IMAGE_LOAD_CONFIG_DIRECTORY32_2),
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    &__security_cookie,
    __safe_se_handler_table,
    (DWORD)(DWORD_PTR) &__safe_se_handler_count
};
```

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Command Line** property page.
4. Type the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See Also
Reference
Setting Linker Options
Linker Options

/SECTION (Specify Section Attributes)

```
/SECTION:name,[[!]{DEKPRSW}][,ALIGN=#]
```

Remarks

The /SECTION option changes the attributes of a section, overriding the attributes set when the .obj file for the section was compiled.

A section in a portable executable (PE) file is roughly equivalent to a segment or the resources in a new executable (NE) file. Sections contain either code or data. Unlike segments, sections are blocks of contiguous memory with no size constraints. Some sections contain code or data that your program declared and uses directly, while other data sections are created for you by the linker and library manager (lib.exe) and contain information vital to the operating system. For more information on NE files, see Knowledge Base article "Executable-File Header Format" (Q65122). You can find Knowledge Base articles in the MSDN Library, or at http://support.microsoft.com.

Specify a colon (:) and a section *name*. The *name* is case sensitive.

Do not use the following names, as they will conflict with standard names. For example, .sdata is used on RISC platforms:

- .arch
- .bss
- .data
- .edata
- .idata
- .pdata
- .rdata
- .reloc
- .rsrc
- .sbss
- .sdata
- .srdata
- .text
- .xdata

Specify one or more attributes for the section. The attribute characters, listed below, are not case sensitive. You must specify all attributes that you want the section to have; an omitted attribute character causes that attribute bit to be turned off. If you do not specify R, W, or E, the existing read, write, or executable status remains unchanged.

To negate an attribute, precede its character with an exclamation point (!). The meanings of the attribute characters are shown below.

Character	Attribute	Meaning
E	Execute	The section is executable
R	Read	Allows read operations on data
W	Write	Allows write operations on data
S	Shared	Shares the section among all processes that load the image

D	Discardable	Marks the section as discardable
K	Cacheable	Marks the section as not cacheable
P	Pageable	Marks the section as not pageable

K and P are peculiar in that the section flags that correspond to them are in the negative sense. If you specify one of them on the .text section (/SECTION:.text,K), there will be no difference in the section flags when you run DUMPBIN with the /HEADERS option; it was already implicitly cached. To remove the default, specify /SECTION:.text,!K and DUMPBIN will reveal section characteristics, including "Not Cached."

A section in the PE file that does not have E, R, or W set is probably invalid.

The ALIGN=# lets you specify an alignment value for a particular section. See /ALIGN for more information.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Command Line** property page.
4. Type the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See Also
Reference
Setting Linker Options
Linker Options

/STACK (Stack Allocations)

```
/STACK:reserve[,commit]
```

Remarks

The /STACK option sets the size of the stack in bytes. This option is only for use when building an .exe file.

This option specifies the total stack allocation in virtual memory. The default stack size is 1 MB. The linker rounds up the specified value to the nearest 4 bytes.

commit is subject to interpretation by the operating system. In Windows NT and Windows 2000 it specifies the amount of physical memory to allocate at a time. Committed virtual memory causes space to be reserved in the paging file. A higher *commit* value saves time when the application needs more stack space, but increases the memory requirements and possibly the startup time.

Specify the *reserve* and *commit* values in decimal or C-language notation.

Another way to set the size of the stack is with the STACKSIZE statement in a module-definition (.def) file. **STACKSIZE** overrides the Stack Allocations (/STACK) option if both are specified. You can change the stack after the .exe file is built by using the EDITBIN tool.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **System** property page.
4. Modify one of the following properties:
   - **Stack Commit Size**
   - **Stack Reserve Size**

To set this linker option programmatically

- See StackCommitSize and StackReserveSize properties.

See Also
Reference
Setting Linker Options
Linker Options

/STUB (MS-DOS Stub File Name)

```
/STUB:filename
```

Remarks

where:

filename

An MS-DOS application.

Remarks

The /STUB option attaches an MS-DOS stub program to a Win32 program.

A stub program is invoked if the file is executed in MS-DOS. It usually displays an appropriate message; however, any valid MS-DOS application can be a stub program.

Specify a *filename* for the stub program after a colon (:) on the command line. The linker checks *filename* and issues an error message if the file is not an executable. The program must be an .exe file; a .com file is invalid for a stub program.

If this option is not used, the linker attaches a default stub program that issues the following message:

```
This program cannot be run in MS-DOS mode.
```

When building a virtual device driver, *filename* allows the user to specify a file name that contains an IMAGE_DOS_HEADER structure (defined in WINNT.H) to be used in the VXD, rather than the default header.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Command Line** property page.
4. Type the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See Also

Reference

Setting Linker Options

Linker Options

/SUBSYSTEM (Specify Subsystem)

```
/SUBSYSTEM:{CONSOLE|EFI_APPLICATION|EFI_BOOT_SERVICE_DRIVER|
            EFI_ROM|EFI_RUNTIME_DRIVER|NATIVE|POSIX|WINDOWS|WINDOWSCE}
            [,major[.minor]]
```

Remarks

where:

CONSOLE

Win32 character-mode application. Console applications are given a console by the operating system. If **main** or **wmain** is defined, CONSOLE is the default.

Extensible Firmware Interface

The EFI_* subsystems. See the EFI specification for more information. For example, see the Intel web site. The minimum version and default version is 1.0.

NATIVE

Device drivers for Windows NT. If /DRIVER:WDM is specified, NATIVE is the default.

POSIX

Application that runs with the POSIX subsystem in Windows NT.

WINDOWS

Application does not require a console, probably because it creates its own windows for interaction with the user. If **WinMain** or **wWinMain** is defined, WINDOWS is the default.

WINDOWSCE

Application that runs on a Windows CE device.

*major*and *minor*(optional)

Specify the minimum required version of the subsystem. The arguments are decimal numbers in the range 0 through 65,535. See the Remarks for details. There are no upper bounds for version numbers.

Remarks

The /SUBSYSTEM option tells the operating system how to run the .exe file.

The choice of subsystem affects the entry point symbol (or entry point function) that the linker will choose.

The optional minimum and default *major* and *minor* version numbers for the subsystems are as follows.

Subsystem	**Minimum**	**Default**
CONSOLE	4.00 (x86) 5.01 (Itanium)5.02 (x64)	4.00 (x86) 5.01 (Itanium)5.02 (x64)
WINDOWS	4.00 (x86) 5.01 (Itanium) 5.02 (x64)	4.00 (x86) 5.01 (Itanium) 5.02 (x64)
NATIVE (with DRIVER:WDM)	1.00 (x86) 1.10 (Itanium) 1.10 (x64)	1.00 (x86) 1.10 (Itanium) 1.10 (x64)
NATIVE (without /DRIVER:WDM)	4.00 (x86) 5.01 (Itanium) 5.02 (x64)	4.00 (x86) 5.01 (Itanium) 5.02 (x64)
POSIX	1.0	19.90

WINDOWSCE	3.0	4.10
EFI_APPLICATION, EFI_BOOT_SERVICE_DRIVER, EFI_ROM, EFI_RUN TIME_DRIVER	1.0	1.0

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **System** property page.
4. Modify the **SubSystem** property.

To set this linker option programmatically

- See SubSystem.

See Also
Reference
Setting Linker Options
Linker Options

/SWAPRUN (Load Linker Output to Swap File)

```
/SWAPRUN:{NET|CD}
```

Remarks

The /SWAPRUN option tells the operating system to first copy the linker output to a swap file, and then run the image from there. This is a Windows NT 4.0 (and later) feature.

If NET is specified, the operating system will first copy the binary image from the network to a swap file and load it from there. This option is useful for running applications over the network. When CD is specified, the operating system will copy the image on a removable disk to a page file and then load it.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **System** property page.
4. Modify one of the following properties:
   - **Swap Run From CD**
   - **Swap Run From Network**

To set this linker option programmatically

- See SwapRunFromCD and SwapRunFromNet properties.

See Also

Reference

Setting Linker Options

Linker Options

/TLBID (Specify Resource ID for TypeLib)

```
/TLBID:id
```

Remarks

where:

id

A user-specified value for a linker-created type library. It overrides the default resource ID of 1.

Remarks

When compiling a program that uses attributes, the linker will create a type library. The linker will assign a resource ID of 1 to the type library.

If this resource ID conflicts with one of your existing resources, you can specify another ID with /TLBID. The range of values that you can pass to *id* is 1 to 65535.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Embedded IDL** property page.
4. Modify the **TypeLib Resource ID** property.

To set this linker option programmatically

- See TypeLibraryResourceID.

See Also
Reference
Setting Linker Options
Linker Options

/TLBOUT (Name .TLB File)

```
/TLBOUT:[path\]filename
```

Remarks

where:

path

An absolute or relative path specification for where the .tlb file should be created.

filename

Specifies the name of the .tlb file created by the MIDL compiler. No file extension is assumed; specify *filename*.tlb if you want a .tlb extension.

Remarks

The /TLBOUT option specifies the name and extension of the .tlb file.

The MIDL compiler is called by the Visual C++ linker when linking projects that have the module attribute.

If /TLBOUT is not specified, the .tlb file will get its name from /IDLOUT *filename*. If /IDLOUT is not specified, the .tlb file will be called vc70.tlb.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Embedded IDL** property page.
4. Modify the **Type Library** property.

To set this linker option programmatically

- See TypeLibraryFile.

See Also

Reference

Setting Linker Options
Linker Options
/IGNOREIDL (Don't Process Attributes into MIDL)
/MIDL (Specify MIDL Command Line Options)

Other Resources

Building an Attributed Program

/TSAWARE (Create Terminal Server Aware Application)

```
/TSAWARE[:NO]
```

Remarks

The /TSAWARE option sets a flag in the IMAGE_OPTIONAL_HEADER DllCharacteristics field in the program image's optional header. When this flag is set, Terminal Server will not make certain changes to the application.

When an application is not Terminal Server aware (also known as a legacy application), Terminal Server makes certain modifications to the legacy application to make it work properly in a multiuser environment. For example, Terminal Server will create a virtual Windows folder, such that each user gets a Windows folder instead of getting the system's Windows directory. This gives users access to their own INI files. In addition, Terminal Server makes some adjustments to the registry for a legacy application. These modifications slow the loading of the legacy application on Terminal Server.

If an application is Terminal Server aware, it must neither rely on INI files nor write to the **HKEY_CURRENT_USER** registry during setup.

If you use /TSAWARE and your application still uses INI files, the files will be shared by all users of the system. If that is acceptable, you can still link your application with /TSAWARE; otherwise you need to use /TSAWARE:NO.

The /TSAWARE option is enabled by default for Windows 2000 and later, for Windows and console applications. See /SUBSYSTEM and /VERSION for information.

/TSAWARE is not valid for drivers, VxDs, or DLLs.

If an application was linked with /TSAWARE, DUMPBIN /HEADERS will display information to that effect.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **System** property page.
4. Modify the **Terminal Server** property.

To set this linker option programmatically

- See TerminalServerAware.

See Also

Reference

Setting Linker Options
Linker Options
Storing User-Specific Information
Legacy Applications in a Terminal Services Environment

/VERBOSE (Print Progress Messages)

```
/VERBOSE[:ICF |:LIB |:REF |:SAFESEH]
```

Remarks

The linker sends information about the progress of the linking session to the Output window. On the command line, the information is sent to standard output and can be redirected to a file.

Option	Description
/VERBOSE	Displays details about the linking process.
/VERBOSE:ICF	Display information about linker activity resulting from the use of /OPT:ICF.
/VERBOSE:LIB	Displays progress messages indicating just the libraries searched. The displayed information includes the library search process and lists each library and object name (with full path), the symbol being resolved from the library, and a list of objects that reference the symbol.
/VERBOSE:REF	Display information about linker activity resulting from the use of /OPT:REF.
/VERBOSE:SAFESEH	Displays information about which modules are not compatible with safe exception handling when /SAFESEH is not specified.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Command Line** property page.
4. Type the option into the **Additional Options** box.

To set this linker option programmatically

- See ShowProgress.

See Also

Reference

Setting Linker Options

Linker Options

/VERSION (Version Information)

```
/VERSION:major[.minor]
```

Remarks

where:

*major*and *minor*

The version number you want in the header of the .exe or .dll file.

Remarks

The /VERSION option tells the linker to put a version number in the header of the .exe or .dll file. Use DUMPBIN /HEADERS to see the image version field of the OPTIONAL HEADER VALUES to see the effect of /VERSION.

The *major* and *minor* arguments are decimal numbers in the range 0 through 65,535. The default is version 0.0.

The information specified with /VERSION does not affect the version information that appears for an application when you view its properties in Windows Explorer. That version information comes from a resource file that is used to build the application. See Version Information Editor for more information.

Another way to insert a version number is with the VERSION module-definition statement.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **General** property page.
4. Modify the **Version** property.

To set this linker option programmatically

- See Version.

See Also
Reference
Setting Linker Options
Linker Options

/WX (Treat Linker Warnings as Errors)

```
/WX[:NO]
```

Remarks

/WX causes no output file to be generated if the linker generates a warning.

This is similar to **/WX** for the compiler (see /w, /Wn, /WX, /Wall, /wln, /wdn, /wen, /won (Warning Level) for more information). However, specifying **/WX** for the compilation does not imply that **/WX** will also be in effect for the link phase; you must explicitly specify **/WX** for each tool.

By default, **/WX** is not in effect. To treat linker warnings as errors, specify **/WX**. **/WX:NO** is the same as not specifying **/WX**.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Click the **Linker** folder.
3. Click the **Command Line** property page.
4. Type the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See Also
Reference
Setting Linker Options
Linker Options

Module-Definition (.def) Files

Module-definition (.def) files provide the linker with information about exports, attributes, and other information about the program to be linked. A .def file is most useful when building a DLL. Because there are linker options that can be used instead of module-definition statements, .def files are generally not necessary. You can also use __declspec(dllexport) as a way to specify exported functions.

You can invoke a .def file during the linker phase with the /DEF (Specify Module-Definition File) linker option.

If you are building an .exe file that has no exports, using a .def file will make your output file larger and slower loading.

See the following sections for more information:

- Rules for Module-Definition Statements
- EXPORTS
- HEAPSIZE
- LIBRARY
- NAME
- SECTIONS
- STACKSIZE
- STUB
- VERSION
- Reserved words

See Also

Reference

Linker Options

Frequently Asked Questions on Building

Other Resources

C/C++ Building Reference

Rules for Module-Definition Statements

The following syntax rules apply to all statements in a .def file. Other rules that apply to specific statements are described with each statement.

- Statements, attribute keywords, and user-specified identifiers are case sensitive.
- Long file names containing spaces or semicolons (;) must be enclosed in quotation marks (").
- Use one or more spaces, tabs, or newline characters to separate a statement keyword from its arguments and to separate statements from each other. A colon (:) or equal sign (=) that designates an argument is surrounded by zero or more spaces, tabs, or newline characters.
- A **NAME** or **LIBRARY** statement, if used, must precede all other statements.
- The **SECTIONS** and **EXPORTS** statements can appear more than once in the .def file. Each statement can take multiple specifications, which must be separated by one or more spaces, tabs, or newline characters. The statement keyword must appear once before the first specification and can be repeated before each additional specification.
- Many statements have an equivalent LINK command-line option. See the description of the corresponding LINK option for additional details.
- Comments in the .def file are designated by a semicolon (;) at the beginning of each comment line. A comment cannot share a line with a statement, but it can appear between specifications in a multiline statement. (**SECTIONS** and **EXPORTS** are multiline statements.)
- Numeric arguments are specified in base 10 or hexadecimal.
- If a string argument matches a reserved word, it must be enclosed in double quotation marks (").

See Also

Reference

Module-Definition (.def) Files

Frequently Asked Questions on Building

EXPORTS

Introduces a section of one or more *definitions* that are exported functions or data. Each definition must be on a separate line.

```
EXPORTS
definitions
```

Remarks

The **EXPORTS** keyword can be on the same line as the first definition or on a preceding line. The .def file can contain one or more **EXPORTS** statements.

The syntax for export *definitions* is:

```
        entryname[=internalname] [@ordinal [NONAME]] [PRIVATE] [DATA]
```

entryname is the function or variable name that you want to export. This is required. If the name you export is different from the name in the DLL, specify the export's name in the DLL with *internalname*. For example, if your DLL exports a function, `func1()` and you want it to be used as `func2()`, you would specify:

```
EXPORTS
func2=func1
```

@*ordinal* lets you specify that a number, and not the function name, will go into the DLL's export table. This can help you minimize the size of your DLL. The .LIB file will contain the mapping between the ordinal and the function, which allows you to use the function name as you normally would in projects that use the DLL.

The optional **NONAME** keyword allows you to export by ordinal only and reduce the size of the export table in the resulting DLL. However, if you want to use GetProcAddress on the DLL, you must know the ordinal because the name will not be valid.

The optional keyword **PRIVATE** prevents *entryname* from being placed in the import library generated by LINK. It has no effect on the export in the image also generated by LINK.

The optional keyword **DATA** specifies that an export is data, not code. For example, you could export a data variable as follows:

```
EXPORTS
i DATA
```

When you use **PRIVATE** and **DATA** for the same export, **PRIVATE** must precede **DATA**.

There are three methods for exporting a definition, listed in recommended order of use:

1. The **__declspec(dllexport)** keyword in the source code
2. An **EXPORTS** statement in a .def file
3. An /EXPORT specification in a LINK command

All three methods can be used in the same program. When LINK builds a program that contains exports, it also creates an import library, unless an .exp file is used in the build.

The following is an example EXPORTS section:

```
EXPORTS
   DllCanUnloadNow      @1     PRIVATE   DATA
   DllWindowName = Name        DATA
   DllGetClassObject    @4 NONAME   PRIVATE
   DllRegisterServer    @7
   DllUnregisterServer
```

Note that when you export a variable from a DLL with a .def file, you do not need to specify **__declspec(dllexport)** on the

variable. However, in any file that uses the DLL, you must still use **__declspec(dllimport)** on the declaration of data.

See Also

Reference

Rules for Module-Definition Statements

LIBRARY

Tells LINK to create a DLL. At the same time, LINK creates an import library, unless an .exp file is used in the build.

```
LIBRARY [library][BASE=address]
```

Remarks

The *library* argument specifies the name of the DLL. You can also use the /OUT linker option to specify the DLL's output name.

The BASE=*address* argument sets the base address that the operating system uses to load the DLL. This argument overrides the default DLL location of 0x10000000. See the description of the /BASE option for details about base addresses.

Remember to use the /DLL linker option when you build a DLL.

See Also

Reference

Rules for Module-Definition Statements

HEAPSIZE

Exposes the same functionality as the /HEAP linker option.

```
/HEAP:reserve[,commit]
```

See Also

Reference

Rules for Module-Definition Statements

NAME (C/C++)

Specifies a name for the main output file.

```
NAME [application][BASE=address]
```

Remarks

An equivalent way to specify an output file name is with the /OUT linker option, and an equivalent way to set the base address is with the /BASE linker option. If both are specified, /OUT overrides **NAME**.

If you build a DLL, NAME will only affect the DLL name.

See Also

Reference

Rules for Module-Definition Statements

SECTIONS (C/C++)

Introduces a section of one or more *definitions* that are access specifiers on sections in your project's output file.

```
SECTIONS
definitions
```

Remarks

Each definition must be on a separate line. The **SECTIONS** keyword can be on the same line as the first definition or on a preceding line. The .def file can contain one or more **SECTIONS** statements.

This **SECTIONS** statement sets attributes for one or more sections in the image file, and can be used to override the default attributes for each type of section.

The format for *definitions* is:

.section_name specifier

where *.section_name* is the name of a section in your program image and *specifier* is one or more of the following access modifiers:

- **EXECUTE**
- **READ**
- **SHARED**
- **WRITE**

Separate specifier names with a space. For example:

```
SECTIONS
.rdata READ WRITE
```

SECTIONS marks the beginning of a list of section *definitions*. Each *definition* must be on a separate line. The **SECTIONS** keyword can be on the same line as the first *definition* or on a preceding line. The .def file can contain one or more **SECTIONS** statements. The **SEGMENTS** keyword is supported as a synonym for **SECTIONS**.

Older versions of Visual C++ supported:

```
        section [CLASS 'classname'] specifier
```

The **CLASS** keyword is supported for compatibility, but is ignored.

An equivalent way to specify section attributes is with the /SECTION option.

See Also
Reference
Rules for Module-Definition Statements

STACKSIZE

Sets the size of the stack in bytes.

```
STACKSIZE reserve[,commit]
```

Remarks

An equivalent way to set the stack is with the Stack Allocations (/STACK) option. See the documentation on that option for details about the *reserve* and *commit* arguments.

This option has no effect on DLLs.

See Also

Reference

Rules for Module-Definition Statements

STUB

When used in a module definition file that builds a virtual device driver (VxD), allows you to specify a file name that contains an IMAGE_DOS_HEADER structure (defined in WINNT.H) to be used in the virtual device driver (VxD), rather than the default header.

```
STUB:filename
```

Remarks

An equivalent way to specify *filename* is with the /STUB linker option.

STUB is valid in a module definition file only when building a VxD.

See Also

Reference

Rules for Module-Definition Statements

VERSION (C/C++)

Tells LINK to put a number in the header of the .exe file or DLL.

```
VERSION major[.minor]
```

Remarks

The *major* and *minor* arguments are decimal numbers in the range 0 through 65,535. The default is version 0.0.

An equivalent way to specify a version number is with the Version Information (/VERSION) option.

See Also

Reference

Rules for Module-Definition Statements

Linker Support for Delay-Loaded DLLs

The Visual C++ linker now supports the delayed loading of DLLs. This relieves you of the need to use the Platform SDK functions **LoadLibrary** and **GetProcAddress** to implement DLL delayed loading.

Before Visual C++ 6.0, the only way to load a DLL at run time was by using **LoadLibrary** and **GetProcAddress**; the operating system would load the DLL when the executable or DLL using it was loaded.

Beginning with Visual C++ 6.0, when statically linking with a DLL, the linker provides options to delay load the DLL until the program calls a function in that DLL.

An application can delay load a DLL using the /DELAYLOAD (Delay Load Import) linker option with a helper function (default implementation provided by Visual C++). The helper function will load the DLL at run time by calling **LoadLibrary** and **GetProcAddress** for you.

You should consider delay loading a DLL if:

- Your program may not call a function in the DLL.
- A function in the DLL may not get called until late in your program's execution.

The delayed loading of a DLL can be specified during the build of either a .EXE or .DLL project. A .DLL project that delays the loading of one or more DLLs should not itself call a delay-loaded entry point in Dllmain.

The following topics describe delay loading DLLs:

- Specifying DLLs to Delay Load
- Explicitly Unloading a Delay-Loaded DLL
- Loading All Imports for a Delay-Loaded DLL
- Binding Imports
- Error Handling and Notification
- Dumping Delay-Loaded Imports
- Constraints of Delay Loading DLLs
- Understanding the Helper Function
- Developing Your Own Helper Function

See Also

Reference

Peering Inside the PE: A Tour of the Win32 Portable Executable File Format

Concepts

DLLs

Linking

Specifying DLLs to Delay Load

You can specify which DLLs to delay load with the /delayload:*dllname* linker option. If you do not plan to use your own version of a helper function, you must also link your program with Delayimp.lib.

The following is a simple example of delay loading a DLL:

```
// cl t.cpp user32.lib delayimp.lib  /link /DELAYLOAD:user32.dll
#include <windows.h>
// uncomment these lines to remove .libs from command line
// #pragma comment(lib, "delayimp")
// #pragma comment(lib, "user32")

int main() {
   // user32.dll will load at this point
   MessageBox(NULL, "Hello", "Hello", MB_OK);
}
```

Build the DEBUG version of the project. Step through the code using the debugger and you will notice that user32.dll is loaded only when you make the call to **MessageBox**.

See Also

Concepts

Linker Support for Delay-Loaded DLLs

Explicitly Unloading a Delay-Loaded DLL

The /delay:unload linker option allows you to unload a DLL that was delay loaded. By default, when your code unloads the DLL (using /delay:unload and **__FUnloadDelayLoadedDLL2**), the delay-loaded imports remain in the import address table (IAT). However, if you use /delay:unload on the linker command line, the helper function will support the explicit unloading of the DLL, resetting the IAT to its original form; the now-invalid pointers will be overwritten. The IAT is a field in the ImgDelayDescr that contains the address of a copy of the original IAT (if it exists).

Example

```
// link with /link /DELAYLOAD:MyDLL.dll /DELAY:UNLOAD
#include <windows.h>
#include <delayimp.h>
#include "MyDll.h"
#include <stdio.h>

#pragma comment(lib, "delayimp")
#pragma comment(lib, "MyDll")
int main()
{
    BOOL TestReturn;
    // MyDLL.DLL will load at this point
    fnMyDll();

    //MyDLL.dll will unload at this point
    TestReturn = __FUnloadDelayLoadedDLL2("MyDll.dll");

    if (TestReturn)
        printf_s("\nDLL was unloaded");
    else
        printf_s("\nDLL was not unloaded");
}
```

Important notes on unloading a delay-loaded DLL:

- You can find the implementation of the **__FUnloadDelayLoadedDLL2** function in the file \VC7\INCLUDE\DELAYHLP.CPP.
- The name parameter of the **__FUnloadDelayLoadedDLL2** function must exactly match (including case) what the import library contains (that string is also in the import table in the image). You can view the contents of the import library with DUMPBIN /DEPENDENTS. If a case-insensitive string match is desired, you can update **__FUnloadDelayLoadedDLL2** to use one of the CRT string functions or a Windows API call.

See Unloading a Delay-Loaded DLL for more information.

See Also

Concepts

Linker Support for Delay-Loaded DLLs

Binding Imports

The default linker behavior is to create a bindable import address table for the delay-loaded DLL. If the DLL is bound, the helper function will attempt to use the bound information instead of calling **GetProcAddress** on each of the referenced imports. If either the timestamp or the preferred address do not match those of the loaded DLL, the helper function will assume the bound import address table is out of date and will proceed as if it does not exist.

If you never intend to bind the DLL's delay-loaded imports, specifying /delay:nobind on the linker's command line will prevent the bound import address table from being generated and consuming space in the image file.

See Also

Concepts

Linker Support for Delay-Loaded DLLs

Loading All Imports for a Delay-Loaded DLL

The **__HrLoadAllImportsForDll** function, which is defined in delayhlp.cpp, tells the linker to load all imports from a DLL that was specified with the /delayload linker option.

Loading all imports allows you to put error handling in one place in your code and not have to use exception handling around the actual calls to the imports. It also avoids a situation where your application fails partially through a process as a result of the helper code failing to load an import.

Calling **__HrLoadAllImportsForDll** does not change the behavior of hooks and error handling; see Error Handling and Notification for more information.

The following example shows how to call **__HrLoadAllImportsForDll**:

```
if (FAILED(__HrLoadAllImportsForDll("delay1.dll"))) {
   printf ( "failed on snap load, exiting\n" );
   exit(2);
}
```

See Also

Concepts

Linker Support for Delay-Loaded DLLs

Error Handling and Notification

For more information on error handling and notification, see Understanding the Helper Function.

For more information on hook functions, see Structure and Constant Definitions.

If your program uses delay-loaded DLLs, it must handle errors robustly since failures that occur while the program is running will result in unhandled exceptions. Failure handling is comprised of two portions:

Recovery through a hook.

If your code needs to recover or provide an alternate library and/or routine on failure, a hook can be provided to the helper function that can supply or remedy the situation. The hook routine needs to return a suitable value, so that processing can continue (an HINSTANCE or FARPROC) or 0 to indicate that an exception should be thrown. It could also throw its own exception or **longjmp** out of the hook. There are notification hooks and failure hooks.

Reporting via an exception.

If all that is necessary for handling the error is to abort the procedure, no hook is necessary as long as the user code can handle the exception.

The following topics discuss error handling and notification:

- Notification Hooks
- Failure Hooks
- Exceptions

See Also

Concepts

Linker Support for Delay-Loaded DLLs

Notification Hooks

The notification hooks are called just before the following actions are taken in the helper routine:

- The stored handle to the library is checked to see if it has already been loaded.
- **LoadLibrary** is called to attempt the load of the DLL.
- **GetProcAddress** is called to attempt to get the address of the procedure.
- Return to the delay import load thunk.

The notification hook is enabled:

- By supplying a new definition of the pointer **__pfnDliNotifyHook2** that is initialized to point to your own function that receives the notifications.

  -or-

- By setting the pointer **__pfnDliNotifyHook2** to your hook function before any calls to the DLL that the program is delay loading.

If the notification is **dliStartProcessing**, the hook function can return:

NULL

The default helper handles the loading of the DLL. This is useful to be called just for informational purposes.

function pointer

Bypass the default delay-load handling. This lets you supply your own load handler.

If the notification is **dliNotePreLoadLibrary**, the hook function can return:

- 0, if it just wants informational notifications.
- The HMODULE for the loaded DLL, if it loaded the DLL itself.

If the notification is **dliNotePreGetProcAddress**, the hook function can return:

- 0, if it just wants informational notifications.
- The imported function's address, if the hook function gets the address itself.

If the notification is **dliNoteEndProcessing**, the hook function's return value is ignored.

If this pointer is initialized (nonzero), the delay load helper will invoke the function at certain notification points throughout its execution. The function pointer has the following definition:

```
// The "notify hook" gets called for every call to the
// delay load helper.  This allows a user to hook every call and
// skip the delay load helper entirely.
//
// dliNotify == {
//  dliStartProcessing |
//  dliNotePreLoadLibrary  |
//  dliNotePreGetProc |
//  dliNoteEndProcessing}
//  on this call.
//
ExternC
PfnDliHook   __pfnDliNotifyHook2;

// This is the failure hook, dliNotify = {dliFailLoadLib|dliFailGetProc}
ExternC
PfnDliHook   __pfnDliFailureHook2;
```

The notifications pass in a **DelayLoadInfo** structure to the hook function along with the notification value. This data is identical

to that used by the delay load helper routine. The notification value will be one of the values defined in Structure and Constant Definitions.

See Also

Concepts

Error Handling and Notification

Failure Hooks

The failure hook is enabled in the same manner as the notification hook. The hook routine needs to return a suitable value so that processing can continue (an HINSTANCE or FARPROC) or 0 to indicate that an exception should be thrown.

The pointer variable that refers to the user-defined function is:

```
// This is the failure hook, dliNotify = {dliFailLoadLib|dliFailGetProc}
ExternC
PfnDliHook   __pfnDliFailureHook2;
```

The **DelayLoadInfo** structure contains all the pertinent data necessary for accurate reporting of the error, including the value from **GetLastError**.

If the notification is **dliFailLoadLib**, the hook function can return:

- 0, if it cannot handle the failure.
- An HMODULE, if the failure hook fixed the problem and loaded the library itself.

If the notification is **dliFailGetProc**, the hook function can return:

- 0, if it cannot handle the failure.
- A valid proc address (import function address), if the failure hook succeeded in getting the address itself.

See Also

Concepts

Error Handling and Notification

Exceptions (C/C++)

Two exception codes can be raised when failures are encountered:

- For a **LoadLibrary** failure
- For a **GetProcAddress** failure

Here is the exception information:

```
//
// Exception information
//
#define FACILITY_VISUALCPP  ((LONG)0x6d)
#define VcppException(sev,err)  ((sev) | (FACILITY_VISUALCPP<<16) | err)
```

The exception codes thrown are the standard VcppException(ERROR_SEVERITY_ERROR, ERROR_MOD_NOT_FOUND) and VcppException(ERROR_SEVERITY_ERROR, ERROR_PROC_NOT_FOUND) values. The exception passes a pointer to a **DelayLoadInfo** structure in the LPDWORD value that can be retrieved by **GetExceptionInformation** in the EXCEPTION_RECORD structure, ExceptionInformation[0] field.

Additionally, if the incorrect bits are set in the grAttrs field, the exception ERROR_INVALID_PARAMETER is thrown. This exception is, for all intents and purposes, fatal.

See Structure and Constant Definitions for more information.

See Also

Concepts

Error Handling and Notification

Dumping Delay-Loaded Imports

Delay-loaded imports can be dumped using dumpbin /imports and show up with slightly different information than standard imports. They are segregated into their own section of the /imports dumping and are explicitly labeled as delay-loaded imports. If there is unload information present in the image, that is noted. If there is bind information present, the time/date stamp of the target DLL is noted along with the bound addresses of the imports.

See Also

Concepts

Linker Support for Delay-Loaded DLLs

Constraints of Delay Loading DLLs

There are constraints regarding the delay loading of imports.

- Imports of data cannot be supported. A workaround is to explicitly handle the data import yourself using **LoadLibrary** (or **GetModuleHandle** after you know the delay-load helper has loaded the DLL) and **GetProcAddress**.
- Delay loading Kernel32.dll is not supported. This DLL is necessary for the delay-load helper routines to perform the delay loading.
- Binding of entry points that are forwarded is not supported.
- Delay loading of a DLL may not result in the same behavior of the process if there are per-process initializations that occur in the entry point of the delay-loaded DLL. Other cases include static TLS (thread local storage), declared using __declspec(thread), which is not handled when the DLL is loaded via **LoadLibrary**. Dynamic TLS, using TlsAlloc, TlsFree, TlsGetValue, and TlsSetValue, is still available for use in either static or delay-loaded DLLs.
- Static (global) function pointers should be reinitialized to imported functions after the first call to the function. This is because the first use of the function pointer will point to the thunk.
- There is no way currently to delay the loading of only specific procedures from a DLL while using the normal import mechanism.
- Custom calling conventions (such as using condition codes on x86 architectures) are not supported. Also, you cannot use floating-point data types in any of the helper routines or hooks on RISC machines. The floating-point registers are not saved on any platform. If your custom helper routine or hook routines use floating-point types, they need to completely save and restore the floating-point state on machines with register calling conventions with floating-point parameters. Be careful about delay loading the CRT DLL if you call CRT functions that take floating-point parameters on a numeric data processor (NDP) stack in the help function.

See Also

Concepts

Linker Support for Delay-Loaded DLLs

Understanding the Helper Function

The helper function for linker-supported delayed loading is what actually loads the DLL at run time. You can modify the helper function to customize its behavior by writing your own function and linking it to your program instead of using the supplied helper function in Delayimp.lib. One helper function serves all delay loaded DLLs.

You can provide your own version of the helper function if you want to do specific processing based on the names of the DLL or imports.

The helper function performs the following actions:

- Checks the stored handle to the library to see if it has already been loaded
- Calls **LoadLibrary** to attempt loading of the DLL
- Calls **GetProcAddress** to attempt getting the address of the procedure
- Returns to the delay import load thunk to call the now-loaded entry point

The helper function can call back to a notification hook in your program after each of the following actions:

- When the helper function starts up
- Just before **LoadLibrary** is called in the helper function
- Just before **GetProcAddress** is called in the helper function
- If the call to **LoadLibrary** in the helper function failed
- If the call to **GetProcAddress** in the helper function failed
- After the helper function is done processing

Each of these hook points can return a value that will alter normal processing of the helper routine in some manner except the return to the delay import load thunk.

The default helper code can be found in Delayhlp.cpp and Delayimp.h (in vc\include) and is compiled in Delayimp.lib (in vc\lib). You will need to include this library in your compilations unless you write your own helper function.

The following topics describe the helper function:

- Changes in the DLL Delayed Loading Helper Function Since Visual C++ 6.0
- Calling Conventions, Parameters, and Return Type
- Structure and Constant Definitions
- Calculating Necessary Values
- Unloading a Delay-Loaded DLL

See Also
Concepts
Linker Support for Delay-Loaded DLLs

Changes in the DLL Delayed Loading Helper Function Since Visual C++ 6.0

If you have multiple versions of Visual C++ on your computer or if you defined your own helper function, you may be affected by changes made to the DLL delayed loading helper function. For example:

- **__delayLoadHelper** is now **__delayLoadHelper2**
- **__pfnDliNotifyHook** is now **__pfnDliNotifyHook2**
- **__pfnDliFailureHook** is now **__pfnDliFailureHook2**
- **__FUnloadDelayLoadedDLL** is now **__FUnloadDelayLoadedDLL2**

Note
If you are using the default helper function, these changes will not affect you. There are no changes regarding how you invoke the linker.

Multiple Versions of Visual C++

If you have multiple versions of Visual C++ on your computer, make sure the linker matches delayimp.lib. If there is a mismatch, you will get a linker error reporting either `___delayLoadHelper2@8` or `___delayLoadHelper@8` as an unresolved external symbol. The former implies a new linker with an old delayimp.lib, and the latter implies an old linker with a new delayimp.lib.

If you get an unresolved linker error, run dumpbin /linkermember:1 on the delayimp.lib that you expect to contain the helper function to see which helper function is defined instead. The helper function could also be defined in an object file; run dumpbin /symbols and look for `delayLoadHelper(2)`.

If you know you have the Visual C++ 6.0 linker, then:

- Run dumpbin on the delay load helper's .lib or .obj file to determine whether it defines **__delayLoadHelper2**. If not, the link will fail.
- Define **__delayLoadHelper** in the delay load helper's .lib or .obj file.

User-Defined Helper Function

If you defined your own helper function and are using the current version of Visual C++, do the following:

- Rename the helper function to **__delayLoadHelper2**.
- Since the pointers in the delay descriptor (ImgDelayDescr in delayimp.h) have been changed from absolute addresses (VAs) to relative addresses (RVAs) to work as expected in both 32- and 64-bit programs, you need to convert these back to pointers. A new function has been introduced: PFromRva, found in delayhlp.cpp. You can use this function on each of the fields in the descriptor to convert them back to either 32- or 64-bit pointers. The default delay load helper function continues to be a good template to use as an example.

Load All Imports for a Delay-Loaded DLL

The linker can load all imports from a DLL that you specified to be delay loaded. See Loading All Imports for a Delay-Loaded DLL for more information.

See Also
Reference
Understanding the Helper Function

Calling Conventions, Parameters, and Return Type

The helper routine's prototype is:

```
FARPROC WINAPI __delayLoadHelper2(
   PCImgDelayDescr pidd,
   FARPROC * ppfnIATEntry
);
```

where:

pidd

A **const** pointer to a ImgDelayDescr (see delayimp.h) that contains the offsets of various import-related data, a timestamp for binding information, and a set of attributes that provide further information about the descriptor content. Currently there is only one attribute, dlattrRva, which indicates that the addresses in the descriptor are relative virtual addresses (as opposed to virtual addresses).

See Structure and Constant Definitions for the definition of the PCImgDelayDescr structure.

ppfnIATEntry

A pointer to the slot in the delay load import address table (IAT) to be updated with the address of the imported function. The helper routine needs to store the same value that it will be returning into this location.

Expected Return Values

If the function is successful, it returns the address of the imported function.

If the function fails, it raises an exception and returns 0. Three types of exceptions can be raised:

- Invalid parameter, which happens if the attributes in *pidd* aren't specified correctly.
- LoadLibrary failed on the specified DLL.
- Failure of **GetProcAddress**.

It is your responsibility to handle these exceptions.

Remarks

The calling convention for the helper function is **__stdcall**. The type of the return value is not relevant, so FARPROC is used. This function has C linkage.

The return value of the delay load helper needs to be stored in the passed-in function pointer location, unless you want your helper routine to be used as a notification hook. In that case, your code is responsible for finding the appropriate function pointer to return. The thunk code the linker generates then takes that return value as the real target of the import and jumps directly to it.

Sample

The following code shows how to implement a simple hook function.

```
FARPROC WINAPI delayHook(unsigned dliNotify, PDelayLoadInfo pdli)
{
    switch (dliNotify) {
        case dliStartProcessing :

            // If you want to return control to the helper, return 0.
            // Otherwise, return a pointer to a FARPROC helper function
            // that will be used instead, thereby bypassing the rest
            // of the helper.

            break;

        case dliNotePreLoadLibrary :
```

```
            // If you want to return control to the helper, return 0.
            // Otherwise, return your own HMODULE to be used by the
            // helper instead of having it call LoadLibrary itself.

            break;

        case dliNotePreGetProcAddress :

            // If you want to return control to the helper, return 0.
            // If you choose you may supply your own FARPROC function
            // address and bypass the helper's call to GetProcAddress.

            break;

        case dliFailLoadLib :

            // LoadLibrary failed.
            // If you don't want to handle this failure yourself, return 0.
            // In this case the helper will raise an exception
            // (ERROR_MOD_NOT_FOUND) and exit.
            // If you want to handle the failure by loading an alternate
            // DLL (for example), then return the HMODULE for
            // the alternate DLL. The helper will continue execution with
            // this alternate DLL and attempt to find the
            // requested entrypoint via GetProcAddress.

            break;

        case dliFailGetProc :

            // GetProcAddress failed.
            // If you don't want to handle this failure yourself, return 0.
            // In this case the helper will raise an exception
            // (ERROR_PROC_NOT_FOUND) and exit.
            // If you choose you may handle the failure by returning
            // an alternate FARPROC function address.


            break;

        case dliNoteEndProcessing :

            // This notification is called after all processing is done.
            // There is no opportunity for modifying the helper's behavior
            // at this point except by longjmp()/throw()/RaiseException.
            // No return value is processed.

            break;

        default :

            return NULL;
    }

    return NULL;
}

/*
and then at global scope somewhere
PfnDliHook __pfnDliNotifyHook2 = delayHook;
*/
```

See Also

Reference

Understanding the Helper Function

Structure and Constant Definitions

The default helper routine uses several structures to communicate with the hook functions and during any exceptions. Here are the notification and failure values, information structures, and the pointer-to-hook-function type passed to the hooks:

```
//
// Delay load import hook notifications
//
enum {
    dliStartProcessing,        // used to bypass or note helper only
    dliNotePreLoadLibrary,     // called just before LoadLibrary, can
                               //  override w/ new HMODULE return val
    dliNotePreGetProcAddress,  // called just before GetProcAddress, can
                               //  override w/ new FARPROC return value
    dliFailLoadLib,            // failed to load library, fix it by
                               //  returning a valid HMODULE
    dliFailGetProc,            // failed to get proc address, fix it by
                               //  returning a valid FARPROC
    dliNoteEndProcessing,      // called after all processing is done, no
                               //  bypass possible at this point except
                               //  by longjmp()/throw()/RaiseException.
    };

typedef struct DelayLoadProc {
    BOOL                fImportByName;
    union {
        LPCSTR          szProcName;
        DWORD           dwOrdinal;
        };
    } DelayLoadProc;

typedef struct DelayLoadInfo {
    DWORD           cb;         // size of structure
    PCImgDelayDescr pidd;       // raw form of data (everything is there)
    FARPROC *       ppfn;       // points to address of function to load
    LPCSTR          szDll;      // name of dll
    DelayLoadProc   dlp;        // name or ordinal of procedure
    HMODULE         hmodCur;    // the hInstance of the library we have loaded
    FARPROC         pfnCur;     // the actual function that will be called
    DWORD           dwLastError;// error received (if an error notification)
    } DelayLoadInfo, * PDelayLoadInfo;

typedef FARPROC (WINAPI *PfnDliHook)(
    unsigned        dliNotify,
    PDelayLoadInfo  pdli
    );

typedef struct ImgDelayDescr {
    DWORD        grAttrs;        // attributes
    RVA          rvaDLLName;     // RVA to dll name
    RVA          rvaHmod;        // RVA of module handle
    RVA          rvaIAT;         // RVA of the IAT
    RVA          rvaINT;         // RVA of the INT
    RVA          rvaBoundIAT;    // RVA of the optional bound IAT
    RVA          rvaUnloadIAT;   // RVA of optional copy of original IAT
    DWORD        dwTimeStamp;    // 0 if not bound,
                                 // O.W. date/time stamp of DLL bound to (Old BIND)
    } ImgDelayDescr, * PImgDelayDescr;
```

See Also

Reference

Understanding the Helper Function

Calculating Necessary Values

Two critical pieces of information need to be calculated by the delay helper routine. To that end, there are two inline functions in delayhlp.cpp for calculating this information.

- The first calculates the index of the current import into the three different tables (import address table (IAT), bound import address table (BIAT), and unbound import address table (UIAT)).
- The second counts the number of imports in a valid IAT.

```
// utility function for calculating the index of the current import
// for all the tables (INT, BIAT, UIAT, and IAT).
__inline unsigned
IndexFromPImgThunkData(PCImgThunkData pitdCur, PCImgThunkData pitdBase) {
    return pitdCur - pitdBase;
    }

// utility function for calculating the count of imports given the base
// of the IAT. NB: this only works on a valid IAT!
__inline unsigned
CountOfImports(PCImgThunkData pitdBase) {
    unsigned        cRet = 0;
    PCImgThunkData  pitd = pitdBase;
    while (pitd->u1.Function) {
        pitd++;
        cRet++;
        }
    return cRet;
    }
```

See Also

Reference

Understanding the Helper Function

Unloading a Delay-Loaded DLL

The default-supplied delay-load helper checks to see if the delay-load descriptors have a pointer and a copy of the original import address table (IAT) in the pUnloadIAT field. If so, it will save a pointer in a list to the import delay descriptor. This enables the helper function to find the DLL by name to support unloading that DLL explicitly.

Here are the associated structures and functions for explicitly unloading a delay-loaded DLL:

```
//
// Unload support from delayimp.h
//

// routine definition; takes a pointer to a name to unload

ExternC
BOOL WINAPI
__FUnloadDelayLoadedDLL2(LPCSTR szDll);

// structure definitions for the list of unload records
typedef struct UnloadInfo * PUnloadInfo;
typedef struct UnloadInfo {
    PUnloadInfo     puiNext;
    PCImgDelayDescr pidd;
    } UnloadInfo;

// from delayhlp.cpp
// the default delay load helper places the unloadinfo records in the
// list headed by the following pointer.
ExternC
PUnloadInfo __puiHead;
```

The UnloadInfo structure is implemented using a C++ class that uses **LocalAlloc** and **LocalFree** implementations as its operator **new** and operator **delete** respectively. These options are kept in a standard linked list using __puiHead as the head of the list.

Calling __FUnloadDelayLoadedDLL will attempt to find the name you provide in the list of loaded DLLs (an exact match is required). If found, the copy of the IAT in pUnloadIAT is copied over the top of the running IAT to restore the thunk pointers, the library is freed with **FreeLibrary**, the matching **UnloadInfo** record is unlinked from the list and deleted, and TRUE is returned.

The argument to the function __FUnloadDelayLoadedDLL2 is case sensitive. For example, you would specify:

```
__FUnloadDelayLoadedDLL2("user32.DLL");
```

and not:

```
__FUnloadDelayLoadedDLL2("User32.DLL");.
```

See Also

Reference

Understanding the Helper Function

Developing Your Own Helper Function

You may want to provide your own version of the routine to do specific processing based on the names of the DLL or imports. There are two methods of doing this: coding your own, possibly based on the supplied code, or merely hooking the supplied version using the notification hooks detailed previously.

Code Your Own

This is fairly simple since you can essentially use the supplied code as a guideline for the new one. Of course, it must adhere to the calling conventions and if it returns to the linker-generated thunks, it must return a proper function pointer. Once in your code, you can do pretty much whatever you want in order to satisfy the call or get out of the call.

Use the Start Processing Notification Hook

It will probably be easiest to simply provide a new pointer to a user-supplied notification hook function that receives the same values as the default helper on the notification dliStartProcessing. At that point, the hook function can essentially become the new helper function, as a successful return to the default helper will bypass all further processing in the default helper.

See Also

Concepts

Linker Support for Delay-Loaded DLLs

Release Builds

A release build uses optimizations. When you use optimizations to create a release build, the compiler will not produce symbolic debugging information. The absence of symbolic debugging information, along with the fact that code is not generated for TRACE and ASSERT calls, means that the size of your executable file is reduced and will therefore be faster.

This section presents information on why and when you would want to change from a debug build to a release build. It also discusses some of the problems you may encounter when changing from a debug to a release build:

- Creating a Release Build
- Common Problems When Creating a Release Build
- Fixing Release Build Problems
    - Examining ASSERT Statements
    - Using the Debug Build To Check for Memory Overwrite
    - Debugging a Release Build
    - Checking for Memory Overwrites

See Also

Other Resources

Building C++ Projects in Visual Studio

C/C++ Building Reference

How to: Create a Release Build

To generate a release build of your program

1. Select **Release** from the **Solution Configuration** drop-down list, which is on the **Standard** toolbar.
2. On the **Build** menu, click **Build**.

See Also

Reference

Release Builds

Common Problems When Creating a Release Build

During development, you will usually build and test with a debug build of your project. If you then build your application for a release build, you may get an access violation.

The list below shows the primary differences between a debug and a release (nondebug) build. There are other differences, but following are the primary differences that would cause an application to fail in a release build when it works in a debug build.

- Heap Layout
- Compilation
- Pointer Support
- Optimizations

See the /GZ (Catch Release-Build Errors in Debug Build) compiler option for information on how to catch release build errors in debug builds.

Heap Layout

Heap layout will be the cause of about ninety percent of the apparent problems when an application works in debug, but not release.

When you build your project for debug, you are using the debug memory allocator. This means that all memory allocations have guard bytes placed around them. These guard bytes detect a memory overwrite. Because heap layout is different between release and debug versions, a memory overwrite might not create any problems in a debug build, but may have catastrophic effects in a release build.

For more information, see Check for Memory Overwrite and Use the Debug Build To Check for Memory Overwrite.

Compilation

Many of the MFC macros and much of the MFC implementation changes when you build for release. In particular, the ASSERT macro evaluates to nothing in a release build, so none of the code found in ASSERTs will be executed. For more information, see Examine ASSERT Statements.

Some functions are inlined for increased speed in the release build. Optimizations are generally turned on in a release build. A different memory allocator is also being used.

Pointer Support

The lack of debugging information removes the padding from your application. In a release build, stray pointers have a greater chance of pointing to uninitialized memory instead of pointing to debug information.

Optimizations

Depending on the nature of certain segments of code, the optimizing compiler might generate unexpected code. This is the least likely cause of release build problems, but it does arise on occasion. For a solution, see Optimizing Your Code.

See Also

Reference

Release Builds

Fixing Release Build Problems

Fixing Release Build Problems

If your code generates compile errors after switching from debug build to release build, there are some areas you should check.

You may receive compiler warnings during an optimized (release) build that you did not receive during a debug build.

- Examine ASSERT Statements
- Use the Debug Build To Check for Memory Overwrites
- Turn on Generation of Debug Information for the Release Build
- Check for Memory Overwrite

See Also

Reference

Release Builds
Common Problems When Creating a Release Build
Optimizing Your Code

Examining ASSERT Statements

Suppose that when you run the debug version of your MFC application, there are no problems. However, the release version of the same application crashes, returns incorrect results, and/or exhibits some other abnormal behavior.

This problem can be caused when you place important code in an ASSERT statement to verify that it performs correctly. Because ASSERT statements are commented out in a release build of an MFC program, the code does not run in a release build.

If you are using ASSERT to confirm that a function call succeeded, consider using VERIFY instead. The VERIFY macro evaluates its own arguments in both debug and release builds of the application.

Another preferred technique is to assign the function's return value to a temporary variable and then test the variable in an ASSERT statement.

Examine the following code fragment:

```
enum {
    sizeOfBuffer = 20
};
char *buf;
ASSERT(buf = (char *) calloc(sizeOfBuffer, sizeof(char) ));
strcpy_s( buf, sizeOfBuffer, "Hello, World" );
free( buf );
```

This code runs perfectly in a debug version of an MFC application. If the call to `calloc( )` fails, a diagnostic message that includes the file and line number appears. However, in a retail build of an MFC application:

- the call to `calloc( )` never occurs, leaving `buf` uninitialized, or
- `strcpy_s( )` copies "`Hello, World`" into a random piece of memory, possibly crashing the application or hanging the system, or
- `free()` attempts to free memory that was never allocated.

To use ASSERT correctly, the code sample should be changed to the following:

```
enum {
    sizeOfBuffer = 20
};
char *buf;
buf = (char *) calloc(sizeOfBuffer, sizeof(char) );
ASSERT( buf != NULL );
strcpy_s( buf, sizeOfBuffer, "Hello, World" );
free( buf );
```

Or, you can use VERIFY instead:

```
enum {
    sizeOfBuffer = 20
};
char *buf;
VERIFY(buf = (char *) calloc(sizeOfBuffer, sizeof(char) ));
strcpy_s( buf, sizeOfBuffer, "Hello, World" );
free( buf );
```

See Also

Reference

Fixing Release Build Problems

Using the Debug Build to Check for Memory Overwrite

To use the debug build to check for memory overwrite, you must first rebuild your project for debug. Then, go to the very beginning of your application's **InitInstance** function and add the following line:

```
afxMemDF |= checkAlwaysMemDF;
```

The debug memory allocator puts guard bytes around all memory allocations. However, these guard bytes don't do any good unless you check whether they have been changed (which would indicate a memory overwrite). Otherwise, this just provides a buffer that might, in fact, allow you to get away with a memory overwrite.

By turning on the `checkAlwaysMemDF`, you will force MFC to make a call to the **AfxCheckMemory** function every time a call to **new** or **delete** is made. If a memory overwrite was detected, it will generate a TRACE message that looks similar to the following:

```
Damage Occurred! Block=0x5533
```

If you see one of these messages, you need to step through your code to determine where the damage occurred. To isolate more precisely where the memory overwrite occurred, you can make explicit calls to **AfxCheckMemory** yourself. For example:

```
ASSERT(AfxCheckMemory());
    DoABunchOfStuff();
    ASSERT(AfxCheckMemory());
```

If the first ASSERT succeeds and the second one fails, it means that the memory overwrite must have occurred in the function between the two calls.

Depending on the nature of your application, you may find that **afxMemDF** causes your program to run too slowly to even test. The **afxMemDF** variable causes **AfxCheckMemory** to be called for every call to new and delete. In that case, you should scatter your own calls to **AfxCheckMemory**( ) as shown above, and try to isolate the memory overwrite that way.

See Also

Reference

Fixing Release Build Problems

How to: Debug a Release Build

You can debug a release build.

To enable debugging of a release build, change the following release build options

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
2. Enable /Z7 or **/Zi**.
3. Select/INCREMENTAL:NO.
4. Select /DEBUG:Yes.
5. Select /OPT:REF.
6. Select /OPT:ICF.

You can now debug your release build application. To find the problem, step through the code (or of using Just-In-Time debugging) until you find where the failure occurs, and then determine the incorrect parameters or code.

If a program works in a debug build, but fails in a release build, it is likely that one of the compiler optimizations is exposing a defect in your source code. To isolate the problem, you should disable selected optimizations for each source code file, until you locate the file and the optimization that is causing the problem. For example, you may want to divide the files into two groups, disable optimization on one group, and continue dividing until you isolate the problem to a single file.

You may want to use /RTC to try to expose such bugs in your debug builds.

See Optimizing Your Code for more information.

See Also

Reference

Fixing Release Build Problems

Checking for Memory Overwrites

If you get an access violation on a call to a heap manipulation function, it is possible that your program has corrupted the heap. A common symptom of this situation would be:

```
Access Violation in _searchseg
```

The _heapchk function is available in both debug and release builds (Windows NT only) for verifying the integrity of the run time library heap. You can use **_heapchk** in much the same way as the **AfxCheckMemory** function to isolate a heap overwrite, for example:

```
if(_heapchk()!=_HEAPOK)
   DebugBreak();
```

If this function ever fails, you need to isolate at which point the heap was corrupted.

See Also

Reference

Fixing Release Build Problems

Optimizing Your Code

Optimization is the process of fine-tuning executable performance for best performance and smallest code size. Visual C++ provides the following mechanisms for optimizing code:

- Pragmas and keywords you can insert into your source code.
- Compiler options that support optimizations; see Compiler Options by Category.

Improving Program Performance

For details on how good programming practices can improve program performance, see:

- Tips for Improving Time-Critical Code
- Microsoft Visual Studio and Windows 2000 Scalability Center (http://msdn.microsoft.com/vstudio/centers/scale/default.asp)

About Optimizing Code

Because optimization changes the code created by the compiler, it is best to optimize your code after you have fully tested and debugged it. That way, you can debug code that closely matches your source code and not worry about the effects of optimization. Once you apply optimizations, you should test your code again. Occasionally, code will behave differently when optimizations are applied. In that event, you may need to debug optimized code. Also see Common Problems When Creating a Release Build for more information.

Optimized code sometimes gives different answers not because of an error but because optimization changes the order of calculations, resulting in slightly different results due to the limits of floating-point precision.

You may also notice some additional warning messages when you compile your code with optimization. This is normal behavior because some warnings relate only to optimized code. If you pay attention to these warnings, you can avoid most of the problems associated with optimization.

Paradoxically, optimizing a program for speed can sometimes cause code to run slower because some optimizations for speed increase code size. Inlining functions, for example, eliminates the overhead of function calls, but inlining too much code may make your program so large that the number of virtual-memory page faults increases. Therefore, the speed gained from eliminating function calls may be lost to memory swapping. For this reason, it is a good idea to measure the performance of your program before and after you apply optimizations. For general guidelines on how best to optimize your application, see Optimization Best Practices.

The optimize Pragma

If a section of your code causes errors or slowdown due to optimization, you can use the optimize pragma to turn off optimization for that section. Enclose the code between two pragmas, like this:

```
#pragma optimize("", off)
// some code here
#pragma optimize("", on)
```

Additional Topics

For additional topics on optimization, see:

- Compiler Throughput
- Using Function Name Without () Produces No Code
- Optimizing Inline Assembly
- Specifying Compiler Optimization for an ATL Project
- What optimization techniques should I use to improve the client application's performance when loading?
- Optimizing DLL Load Time Performance
- Bugslayer: Improving Runtime Performance with the Smooth Working Set Tool (Part 1) and (Part 2)

See Also

Other Resources

C/C++ Building Reference

Optimization Pragmas and Keywords

Several keywords and pragmas that you can use in your C or C++ code will affect optimization:

- __asm
- __assume
- inline, _inline, or __forceinline
- #pragma auto_inline
- #pragma check_stack
- #pragma function
- #pragma inline_depth
- #pragma inline_recursion
- #pragma intrinsic
- #pragma optimize
- register Keyword

See Also

Reference

Optimizing Your Code

Improving Compiler Throughput

Use precompiled header files to build your project faster. This is important if you are using ATL, MFC, or the Platform SDK header files.

See /Yc and /Yu.

For more information on precompiled headers, see Creating Precompiled Header Files.

See Also

Reference

Optimizing Your Code

Why Floating-Point Numbers May Lose Precision

Floating-point decimal values generally do not have an exact binary representation. This is a side effect of how the CPU represents floating point data. For this reason, you may experience some loss of precision, and some floating-point operations may produce unexpected results.

This behavior is the result of one of the following:

- The binary representation of the decimal number may not be exact.
- There is a type mismatch between the numbers used (for example, mixing float and double).

To resolve the behavior, most programmers either ensure that the value is greater or less than what is needed, or they get and use a Binary Coded Decimal (BCD) library that will maintain the precision.

Binary representation of floating-point values affects the precision and accuracy of floating-point calculations. Microsoft Visual C++ uses IEEE floating-point format.

Example

```
// Floating-point_number_precision.c
// Compile options needed: none. Value of c is printed with a decimal
// point precision of 10 and 6 (printf rounded value by default) to
// show the difference
#include <stdio.h>

#define EPSILON 0.0001   // Define your own tolerance
#define FLOAT_EQ(x,v) (((v - EPSILON) < x) && (x <( v + EPSILON)))

int main() {
   float a, b, c;

   a = 1.345f;
   b = 1.123f;
   c = a + b;
   // if (FLOAT_EQ(c, 2.468)) // Remove comment for correct result
   if (c == 2.468)            // Comment this line for correct result
      printf_s("They are equal.\n");
   else
      printf_s("They are not equal! The value of c is %13.10f "
                "or %f",c,c);
}
```

Output

```
They are not equal! The value of c is  2.4679999352 or 2.468000
```

Comments

For EPSILON, you can use the constants FLT_EPSILON, which is defined for float as 1.192092896e-07F, or DBL_EPSILON, which is defined for double as 2.2204460492503131e-016. You need to include float.h for these constants. These constants are defined as the smallest positive number x, such that x+1.0 is not equal to 1.0. Because this is a very small number, you should employ user-defined tolerance for calculations involving very large numbers.

See Also

Reference

Optimizing Your Code

IEEE Floating-Point Representation

Microsoft Visual C++ is consistent with the IEEE numeric standards. There are three internal varieties of real numbers. Real*4 and real*8 are used in Visual C++. Real*4 is declared using the word **float**. Real*8 is declared using the word **double**. In Windows 32-bit programming, the **long double** data type maps to **double**. There is, however, assembly language support for computations using the real*10 data type.

The values are stored as follows:

Value	Stored as
real*4	sign bit, 8-bit exponent, 23-bit mantissa
real*8	sign bit, 11-bit exponent, 52-bit mantissa
real*10	sign bit, 15-bit exponent, 64-bit mantissa

In real*4 and real*8 formats, there is an assumed leading 1 in the mantissa that is not stored in memory, so the mantissas are actually 24 or 53 bits, even though only 23 or 52 bits are stored. The real*10 format actually stores this bit.

The exponents are biased by half of their possible value. This means you subtract this bias from the stored exponent to get the actual exponent. If the stored exponent is less than the bias, it is actually a negative exponent.

The exponents are biased as follows:

Exponent	Biased by
8-bit (real*4)	127
11-bit (real*8)	1023
15-bit (real*10)	16383

These exponents are not powers of ten; they are powers of two. That is, 8-bit stored exponents can be up to 127. The value 2**127 is roughly equivalent to 10**38, which is the actual limit of real*4.

The mantissa is stored as a binary fraction of the form 1.XXX... . This fraction has a value greater than or equal to 1 and less than 2. Note that real numbers are always stored in normalized form; that is, the mantissa is left-shifted such that the high-order bit of the mantissa is always 1. Because this bit is always 1, it is assumed (not stored) in the real*4 and real*8 formats. The binary (not decimal) point is assumed to be just to the right of the leading 1.

The format, then, for the various sizes is as follows:

Format	BYTE 1	BYTE 2	BYTE 3	BYTE 4	...	BYTE n
real*4	`SXXX XXXX`	`XMMM MMMM`	`MMMM MMMM`	`MMMM MMMM`		
real*8	`SXXX XXXX`	`XXXX MMMM`	`MMMM MMMM`	`MMMM MMMM`	...	`MMMM MMMM`
real*10	`SXXX XXXX`	`XXXX XXXX`	`1MMM MMMM`	`MMMM MMMM`	...	`MMMM MMMM`

`S` represents the sign bit, the `X`'s are the exponent bits, and the `M`'s are the mantissa bits. Note that the leftmost bit is assumed in real*4 and real*8 formats, but is present as "1" in BYTE 3 of the real*10 format.

To shift the binary point properly, you first unbias the exponent and then move the binary point to the right or left the appropriate number of bits.

Examples

The following are some examples in real*4 format:

- In the following example, the sign bit is zero, and the stored exponent is 128, or 100 0000 0 in binary, which is 127 plus 1. The stored mantissa is (1.) 000 0000 ... 0000 0000, which has an implied leading 1 and binary point, so the actual

mantissa is one.

```
                    SXXX XXXX XMMM MMMM ... MMMM MMMM
2   =  1  * 2**1  = 0100 0000 0000 0000 ... 0000 0000 = 4000 0000
```

- Same as +2 except that the sign bit is set. This is true for all IEEE format floating-point numbers.

```
-2  = -1  * 2**1  = 1100 0000 0000 0000 ... 0000 0000 = C000 0000
```

- Same mantissa, exponent increases by one (biased value is 129, or 100 0000 1 in binary.

```
4  =  1  * 2**2  = 0100 0000 1000 0000 ... 0000 0000 = 4080 0000
```

- Same exponent, mantissa is larger by half — it's (1.) 100 0000 ...0000 0000, which, since this is a binary fraction, is 1 1/2 (the values of the fractional digits are 1/2, 1/4, 1/8, and so forth).

```
6  = 1.5 * 2**2  = 0100 0000 1100 0000 ... 0000 0000 = 40C0 0000
```

- Same exponent as other powers of two, mantissa is one less than two at 127, or 011 1111 1 in binary.

```
1  = 1   * 2**0  = 0011 1111 1000 0000 ... 0000 0000 = 3F80 0000
```

- The biased exponent is 126, 011 1111 0 in binary, and the mantissa is (1.) 100 0000 ... 0000 0000, which is 1 1/2.

```
.75 = 1.5 * 2**-1 = 0011 1111 0100 0000 ... 0000 0000 = 3F40 0000
```

- Exactly the same as two except that the bit that represents 1/4 is set in the mantissa.

```
2.5 = 1.25 * 2**1 = 0100 0000 0010 0000 ... 0000 0000 = 4020 0000
```

- 1/10 is a repeating fraction in binary. The mantissa is just shy of 1.6, and the biased exponent says that 1.6 is to be divided by 16 (it is 011 1101 1 in binary, which is 123 in decimal). The true exponent is 123 – 127 = –4, which means that the factor by which to multiply is 2**–4 = 1/16. Note that the stored mantissa is rounded up in the last bit — an attempt to represent the unrepresentable number as accurately as possible. (The reason that 1/10 and 1/100 are not exactly representable in binary is similar to the reason that 1/3 is not exactly representable in decimal.)

```
0.1 = 1.6 * 2**-4 = 0011 1101 1100 1100 ... 1100 1101 = 3DCC CCCD
```

- `0 = 1.0 * 2**-128 = all zeros--a special case.`

See Also

Reference

Why Floating-Point Numbers May Lose Precision

Tips for Improving Time-Critical Code

Writing fast code requires understanding all aspects of your application and how it interacts with the system. This topic suggests alternatives to some of the more obvious coding techniques to help you ensure that the time-critical portions of your code perform satisfactorily.

To summarize, improving time-critical code requires that you:

- Know which parts of your program have to be fast.
- Know the size and speed of your code.
- Know the cost of new features.
- Know the minimum work needed to accomplish the job.

To gather information on the performance of your code, you can use the performance monitor (perfmon.exe).

- Cache Hits and Page Faults
- Sorting and Searching
- MFC and Class Libraries
- Shared Libraries
- Heaps
- Threads
- Small Working Set

Cache Misses and Page Faults

Missed cache hits, on both the internal and external cache, as well as page faults (going to secondary storage for program instructions and data) slow the performance of a program.

A CPU cache hit can cost your program 10–20 clock cycles. An external cache hit can cost 20–40 clock cycles. A page fault can cost one million clock cycles (assuming a processor that handles 500 million instructions/second and a time of 2 millisecond for a page fault). Therefore, it is in the best interest of program execution to write code that will reduce the number of missed cache hits and page faults.

One reason for slow programs is that they take more page faults or miss the cache more often than necessary. To avoid this, it's important to use data structures with good locality of reference, which means keeping related things together. Sometimes a data structure that looks great turns out to be horrible because of poor locality of reference, and sometimes the reverse is true. Here are two examples:

- Dynamically allocated linked lists can reduce program performance because when you search for an item or when you traverse a list to the end, each skipped link could miss the cache or cause a page fault. A list implementation based on simple arrays might actually be much faster because of better caching and fewer page faults even — allowing for the fact that the array would be harder to grow, it still might be faster.
- Hash tables that use dynamically allocated linked lists can degrade performance. By extension, hash tables that use dynamically allocated linked lists to store their contents might perform substantially worse. In fact, in the final analysis, a simple linear search through an array might actually be faster (depending on the circumstances). Array-based hash tables (so-called "closed hashing") is an often-overlooked implementation which frequently has superior performance.

Sorting and Searching

Sorting is inherently time consuming compared to many typical operations. The best way to avoid unnecessary slowdown is to avoid sorting at critical times. You may be able to:

- Defer sorting until a non-performance–critical time.
- Sort the data at an earlier, non-performance–critical time.
- Sort only the part of the data that truly needs sorting.

Sometimes, you can build the list in sorted order. Be careful, because if you need to insert data in sorted order, you may require a more complicated data structure with poor locality of reference, leading to cache misses and page faults. There is no approach that works in all cases. Try several approaches and measure the differences.

Here are some general tips for sorting:

- Use a stock sort to minimize bugs.
- Any work you can do beforehand to reduce the complexity of the sort is worthwhile. If a one-time pass over your data simplifies the comparisons and reduces the sort from O(n log n) to O(n), you will almost certainly come out ahead.
- Think about the locality of reference of the sort algorithm and the data you expect it to run on.

There are fewer alternatives for searches than for sorting. If the search is time-critical, a binary search or hash table lookup is almost always best, but as with sorting, you must keep locality in mind. A linear search through a small array can be faster than a binary search through a data structure with a lot of pointers that causes page faults or cache misses.

MFC and Class Libraries

The Microsoft Foundation Classes (MFC) can greatly simplify writing code. When writing time-critical code, you should be aware of the overhead inherent in some of the classes. Examine the MFC code that your time-critical code uses to see if it meets your performance requirements. The following list identifies MFC classes and functions you should be aware of:

- **CString** MFC calls the C run-time library to allocate memory for a CString dynamically. Generally speaking, **CString** is as efficient as any other dynamically-allocated string. As with any dynamically allocated string, it has the overhead of dynamic allocation and release. Often, a simple **char** array on the stack can serve the same purpose and is faster. Don't use a **CString** to store a constant string. Use **const char *** instead. Any operation you perform with a **CString** object has some overhead. Using the run-time library string functions may be faster.
- **CArray** A CArray provides flexibility that a regular array doesn't, but your program may not need that. If you know the specific limits for the array, you can use a global fixed array instead. If you use **CArray**, use **CArray::SetSize** to establish its size and specify the number of elements by which it grows when a reallocation is necessary. Otherwise, adding elements can cause your array to be frequently reallocated and copied, which is inefficient and can fragment memory. Also be aware that if you insert an item into an array, **CArray** moves subsequent items in memory and may need to grow the array. These actions can cause cache misses and page faults. If you look through the code that MFC uses, you may see that you can write something more specific to your scenario to improve performance. Since **CArray** is a template, for example, you might provide **CArray** specializations for specific types.
- **CList** CList is a doubly linked list, so element insertion is fast at the head, tail, and at a known position (**POSITION**) in the list. Looking up an element by value or index requires a sequential search, however, which can be slow if the list is long. If your code does not require a doubly linked list you may want to reconsider using **CList**. Using a singly linked list saves the overhead of updating an additional pointer for all operations as well as the memory for that pointer. The additional memory is not great, but it is another opportunity for cache misses or page faults.
- **IsKindOf** This function can generate many calls and access a lot of memory in different data areas, leading to bad locality of reference. It is useful for a debug build (in an ASSERT call, for example), but try to avoid using it in a release build.
- **PreTranslateMessage** Use **PreTranslateMessage** when a particular tree of windows needs different keyboard accelerators or when you must insert message handling into the message pump. **PreTranslateMessage** alters MFC dispatch messages. If you override **PreTranslateMessage**, do so only at the level needed. For example, it is not necessary to override **CMainFrame::PreTranslateMessage** if you are interested only in messages going to children of a particular view. Override **PreTranslateMessage** for the view class instead.

  Do not circumvent the normal dispatch path by using **PreTranslateMessage** to handle any message sent to any window. Use window procedures and MFC message maps for that purpose.
- **OnIdle** Idle events can occur at times you do not expect, such as between **WM_KEYDOWN** and **WM_KEYUP** events. Timers may be a more efficient way to trigger your code. Do not force **OnIdle** to be called repeatedly by generating false messages or by always returning **TRUE** from an override of **OnIdle**, which would never allow your thread to sleep. Again, a timer or a separate thread might be more appropriate.

Shared Libraries

Code reuse is desirable. However, if you are going to use someone else's code, you should make sure you know exactly what it does in those cases where performance is critical to you. The best way to understand this is by stepping through the source

code or by measuring with tools such as PView or Performance Monitor.

Heaps

Use multiple heaps with discretion. Additional heaps created with HeapCreate and HeapAlloc let you manage and then dispose of a related set of allocations. Don't commit too much memory. If you're using multiple heaps, pay special attention to the amount of memory that is initially committed.

Instead of multiple heaps, you can use helper functions to interface between your code and the default heap. Helper functions facilitate custom allocation strategies that can improve the performance of your application. For example, if you frequently perform small allocations, you may want to localize these allocations to one part of the default heap. You can allocate a large block of memory and then use a helper function to suballocate from that block. If you do this, you will not have additional heaps with unused memory because the allocation is coming out of the default heap.

In some cases, however, using the default heap can reduce locality of reference. Use Process Viewer, Spy++, or Performance Monitor to measure the effects of moving objects from heap to heap.

Measure your heaps so you can account for every allocation on the heap. Use the C run-time debug heap routines to checkpoint and dump your heap. You can read the output into a spreadsheet program like Microsoft Excel and use pivot tables to view the results. Note the total number, size, and distribution of allocations. Compare these with the size of working sets. Also look at the clustering of related-sized objects.

You can also use the performance counters to monitor memory usage.

Threads

For background tasks, effective idle handling of events may be faster than using threads. It's easier to understand locality of reference in a single-threaded program.

A good rule of thumb is to use a thread only if an operating system notification that you block on is at the root of the background work. Threads are the best solution in such a case because it is impractical to block a main thread on an event.

Threads also present communication problems. You must manage the communication link between your threads, with a list of messages or by allocating and using shared memory. Managing the communication link usually requires synchronization to avoid race conditions and deadlock problems. This complexity can easily turn into bugs and performance problems.

For additional information, see Idle Loop Processing and Multithreading.

Small Working Set

Smaller working sets mean better locality of reference, fewer page faults, and more cache hits. The process working set is the closest metric the operating system directly provides for measuring locality of reference.

- To set the upper and lower limits of the working set, use SetProcessWorkingSetSize.
- To get the upper and lower limits of the working set, use GetProcessWorkingSetSize.
- To view the size of the working set, use Spy++.

See Also

Reference

Optimizing Your Code

Using Function Name Without () Produces No Code

When a function name declared in your program is used without parentheses, the compiler does not produce code. This occurs regardless of whether or not the function takes parameters because the compiler calculates the function address; however, because the function call operator "()" is not present, no call is made. This result is similar to the following:

```
// compile with /Wall to generate a warning
int a;
a;      // no code generated here either
```

In Visual C++, even using warning level 4 generates no diagnostic output. No warning is issued; no code is produced.

The sample code below compiles (with a warning) and links correctly without errors but produces no code in reference to `funcn( )`. For this to work correctly, add the function call operator "()".

```
#include <stdio.h>
void funcn();

int main() {
   funcn;      /* missing function call operator;
                  call will fail.  Use funcn() */
   }

void funcn() {
   printf("\nHello World\n");
}
```

See Also

Reference

Optimizing Your Code

Optimization Best Practices

This document describes some best practices for optimization in Visual C++ 2005. The following topics are discussed:

- Compiler and Linker Options
  - Changes in Visual C++ 2005
  - Which Level of Optimization Should I Use?
  - Floating Point Switches
- Optimization Declspecs
- Optimization Pragmas
- __restrict and __assume
- Intrinsic Support
- Exceptions

Compiler and Linker Options

Changes in Visual C++ 2005

Visual C++ 2005 now supports profile-guided optimization (PGO). This optimization uses profile data from past executions of an instrumented version of an application to drive later optimization of the application. Using PGO can be time consuming, so it may not be something that every developer uses, but we do recommend using PGO for the final release build of a product. For more information, see Profile-Guided Optimizations.

In addition, Whole Program Optimization (also knows as Link Time Code Generation) and the **/O1** and **/O2** optimizations has been improved. In general, an application compiled with one of these options will be faster than the same application compiled with an earlier compiler.

For more information, see /GL (Whole Program Optimization) and /O1, /O2 (Minimize Size, Maximize Speed).

Which Level of Optimization Should I Use?

If at all possible, final release builds should be compiled with Profile Guided Optimizations. If it is not possible to build with PGO, whether due to insufficient infrastructure for running the instrumented builds or not having access to scenarios, then we suggest building with Whole Program Optimization.

The **/Gy** switch is also very useful. It generates a separate COMDAT for each function, giving the linker more flexibility when it comes to removing unreferenced COMDATs and COMDAT folding. The only downside to using **/Gy** is that it can have a minor effect on build time. Therefore, it is generally recommended to use it. For more information, see /Gy (Enable Function-Level Linking).

For linking in 64-bit environments, it is recommended to use the **/OPT:REF,ICF** linker option, and in 32-bit environments, **/OPT:REF** is recommended. For more information, see /OPT (Optimizations).

It is also strongly recommended to generate debug symbols, even with optimized release builds. It doesn't effect the generated code, and it makes it a lot easier to debug your application, if need be.

Floating Point Switches

In Visual C++ 2005, the **/Op** compiler option has been removed and the following four compiler options dealing with floating point optimizations have been added:

/fp:precise	This is the default recommendation and should be used in most cases.
/fp:fast	Recommended if performance is of the utmost importance, for example in games. This will result in the fastest performance.
/fp:strict	Recommended if precise floating-point exceptions and IEEE behavior is desired. This will result in the slowest performance.

/fp:except[-]	Can be used in conjunction with **/fp:strict** or **/fp:precise**, but not **/fp:fast**.

For more information, see /fp (Specify Floating-Point Behavior).

Optimization Declspecs

In this section we will look at two declspecs that can be used in programs to help performance: **__declspec(restrict)** and **__declspec(noalias)**.

The **restrict** declspec can only be applied to function declarations that return a pointer, such as `__declspec(restrict) void *malloc(size_t size);`

The **restrict** declspec is used on functions that return unaliased pointers. This keyword is used for the C-Runtime Library implementation of **malloc** since it will never return a pointer value that is already in use in the current program (unless you are doing something illegal, such as using memory after it has been freed).

The **restrict** declspec gives the compiler more information for performing compiler optimizations. One of the hardest things for a compiler to determine is what pointers alias other pointers, and using this information greatly helps the compiler.

It is worth pointing out that this is a promise to the compiler, not something that the compiler will verify. If your program uses this **restrict** declspec inappropriately, your program may have incorrect behavior.

For more information, see restrict.

The **noalias** declspec is also applied only to functions, and indicates that the function is a semi-pure function. A semi-pure function is one that references or modifies only locals, arguments, and first-level indirections of arguments. This declspec is a promise to the compiler, and if the function references globals or second-level indirections of pointer arguments then the compiler may generate code that breaks the application.

For more information, see noalias.

Optimization Pragmas

There are also several useful pragmas for helping optimize code. The first one we'll discuss is **#pragma optimize**:

```
#pragma optimize("{opt-list}", on | off)
```

This pragma allows you to set a given optimization level on a function-by-function basis. This is ideal for those rare occasions where your application crashes when a given function is compiled with optimization. You can use this to turn off optimizations for a single function:

```
#pragma optimize("", off)
int foo() {...}
#pragma optimize("", on)
```

For more information, see optimize.

Inlining is one of the most important optimizations that the compiler performs and here we talk about a couple of the pragmas that help modify this behavior.

#pragma inline_recursion is useful for specifying whether or not you want the application to be able to inline a recursive call. By default it is off. For shallow recursion of small functions you may to turn this on. For more information, see inline_recursion.

Another useful pragma for limiting the depth of inlining is **#pragma inline_depth**. This is typically useful in situations where you're trying to limit the size of a program or function. For more information, see inline_depth.

__restrict and __assume

There are a couple of keywords in Visual C++ 2005 that can help performance: __restrict and __assume.

First, it should be noted that **__restrict** and **__declspec(restrict)** are two different things. While they are somewhat related, their semantics are different. **__restrict** is a type qualifier, like **const** or **volatile**, but exclusively for pointer types.

A pointer that is modified with **__restrict** is referred to as a *__restrict pointer*. A __restrict pointer is a pointer that can only be accessed through the __restrict pointer. In other words, another pointer cannot be used to access the data pointed to by the __restrict pointer.

__restrict can be a powerful tool for the Visual C++ optimizer, but use it with great care. If used improperly, the optimizer might perform an optimization that would break your application.

The **__restrict** keyword in Visual C++ 2005 replaces the **/Oa** switch from previous versions.

__assume has been in Visual C++ for multiple releases, but it has become much more usable in Visual C++ 2005. With **__assume,** a developer can tell the compiler to make assumptions about the value of some variable.

For example `__assume(a < 5);` tells the optimizer that at that line of code the variable *a* is less than 5. Again this is a promise to the compiler. If *a* is actually 6 at this point in the program then the behavior of the program after the compiler has optimized may not be what you would expect. **__assume** is most useful prior to switch statements and/or conditional expressions.

There are some limitations to **__assume**. First, like **__restrict**, it is only a suggestion, so the compiler is free to ignore it. Also, **__assume** currently works only with variable inequalities against constants. It does not propagate symbolic inequalities, for example, assume(a < b).

Intrinsic Support

Intrinsics are function calls where the compiler has intrinsic knowledge about the call, and rather than calling a function in a library, it emits code for that function. Support for intrinsics has been greatly enhanced in Visual C++ 2005. The header file intrin.h located at <Installation_Directory>\VC\include\intrin.h contains all of the available intrinsics for each of the three supported platforms (x86, x64, and Itanium).

Intrinsics give the programmer the ability to go deep into the code without having to use assembly. There are several benefits to using intrinsics:

1. Your code is more portable. Several of the intrinsics are available on multiple CPU architectures.
2. Your code is easier to read, since the code is still written in C/C++.
3. Your code gets the benefit of compiler optimizations. As the compiler gets better, the code generation for the intrinsics improves.

For more information, see Compiler Intrinsics and Benefits of Using Intrinsics.

Exceptions

There is a performance hit associated with using exceptions. Some restrictions are introduced when using try blocks that inhibit the compiler from performing certain optimizations. On x86 platforms there is additional performance degradation from try blocks due to additional state information that must be generated during code execution. On the 64-bit platforms, try blocks do not degrade performance as much, but once an exception is thrown, the process of finding the handler and unwinding the stack can be expensive.

Therefore, it is recommended to avoid introducing try/catch blocks into code that does not really need it. If you must use exceptions, use synchronous exceptions if possible. For more information, see Structured Exception Handling (C++).

Lastly, throw exceptions for exceptional cases only. Using exceptions for general control flow will likely make performance suffer.

See Also

Reference

Optimizing Your Code

C/C++ Build Tools

Visual C++ provides the following command-line tools for viewing or manipulating build output:

- BSCMAKE.EXE builds a browse information file (.bsc) that contains information about the symbols (classes, functions, data, macros, and types) in your program. You view this information in browse windows within the development environment. (A .bsc file can also be built in the development environment.)
- LIB.EXE is used to create and manage a library of Common Object File Format (COFF) object files. It can also be used to create export files and import libraries to reference exported definitions.
- EDITBIN.EXE is used to modify COFF binary files.
- DUMPBIN.EXE displays information (such as a symbol table) about COFF binary files.
- NMAKE reads and executes makefiles.
- ERRLOOK, the Error Lookup utility, retrieves a system error message or module error message based on the value entered.

See Also

Reference

Decorated Names
Compiler Options
Linker Options

Other Resources

C/C++ Building Reference

BSCMAKE Reference

The Microsoft Browse Information Maintenance Utility (BSCMAKE.EXE) builds a browse information file (.bsc) from .sbr files created during compilation. You view a browse information file in the Object Browser. For information about the Object Browser, see Navigating in the Object Browser.

When you build your program, you can create a browse information file for your program automatically, using BSCMAKE to build the file. You do not need to know how to run BSCMAKE if you create your browse information file in the Visual C++ development environment. However, you may want to read this topic to understand the choices available.

If you build your program outside of the development environment, you can still create a custom .bsc that you can examine in the environment. Run BSCMAKE on the .sbr files that you created during compilation.

This section includes the following topics:

- Building Browse Information Files: Overview
- Building a .bsc file
- BSCMAKE command line
- BSCMAKE command file
- BSCMAKE options
- BSCMAKE exit codes

See Also
Reference
C/C++ Build Tools

Building Browse Information Files: Overview

To create browse information for symbol browsing, the compiler creates an .sbr file for each source file in your project, then BSCMAKE.EXE concatenates the .sbr files into one .bsc file.

Generating .sbr and .bsc files takes time, so Visual C++ turns these functions off by default. If you want to browse current information, you must turn the browse options on and build your project again.

Use /FR or /Fr to tell the compiler to create .sbr files. To create .bsc files, you can call BSCMAKE from the command line. Using BSCMAKE from the command line gives you more precise control over the manipulation of browse information files. See BSCMAKE Reference for more information.

Tip
You can turn on .sbr file generation but leave .bsc file generation turned off. This provides fast builds but also enables you to create a fresh .bsc file quickly by turning on .bsc file generation and building the project.

You can reduce the time, memory, and disk space required to build a .bsc file by reducing the size of the .bsc file.

See General Property Page (Project) for information on how to build a browser file in the development environment.

To create a smaller .bsc file

1. Use BSCMAKE command-line options to exclude information from the browse information file.
2. Omit local symbols in one or more .sbr files when compiling or assembling.
3. If an object file does not contain information needed for your current stage of debugging, omit its .sbr file from the BSCMAKE command when you rebuild the browse information file.

To combine the browse information from several projects into one browser file (.bsc)

1. Either don't build the .bsc file at the project level or use the /n switch to prevent the .sbr files from being truncated.
2. After all the projects are built, run BSCMAKE with all of the .sbr files as input. Wildcards are accepted. For instance, if you had project directories C:\X, C:\Y, and C:\Z with .sbr files in them and you wanted to combine them all into one .bsc file, then use BSCMAKE C:\X*.sbr C:\Y*.sbr C:\Z*.sbr /o c:\whatever_directory\combined.bsc to build the combined .bsc file.

See Also

Reference

C/C++ Build Tools
BSCMAKE Reference

Building a .bsc File

BSCMAKE can build a new browse information file from newly created .sbr files. It can also maintain an existing .bsc file using .sbr files for object files that have changed since the last build.

- How to create an .sbr file
- How BSCMAKE builds a .bsc file

See Also

Reference

BSCMAKE Reference

Creating an .sbr File

The input files for BSCMAKE are .sbr files. The compiler creates an .sbr file for each object file (.obj) it compiles. When you build or update your browse information file, all .sbr files for your project must be available on disk.

To create an .sbr file with all possible information, specify /FR.

To create an .sbr file that doesn't contain local symbols, specify /Fr. If the .sbr files contain local symbols, you can still omit them from the .bsc file by using BSCMAKE's /El option.

You can create an .sbr file without performing a full compile. For example, you can specify the /Zs option to the compiler to perform a syntax check and still generate an .sbr file if you specify /FR or /Fr.

The build process can be more efficient if the .sbr files are first packed to remove unreferenced definitions. The compiler automatically packs .sbr files.

See Also

Reference

Building a .bsc File

How BSCMAKE Builds a .bsc File

BSCMAKE builds or rebuilds a .bsc file in the most efficient way it can. To avoid potential problems, it is important to understand the build process.

When BSCMAKE builds a browse information file, it truncates the .sbr files to zero length. During a subsequent build of the same file, a zero-length (or empty) .sbr file tells BSCMAKE that the .sbr file has no new contribution to make. It lets BSCMAKE know that an update of that part of the file is not required and an incremental build will be sufficient. During every build (unless the /n option is specified), BSCMAKE first attempts to update the file incrementally by using only those .sbr files that have changed.

BSCMAKE looks for a .bsc file that has the name specified with the /o option. If /o is not specified, BSCMAKE looks for a file that has the base name of the first .sbr file and a .bsc extension. If the file exists, BSCMAKE performs an incremental build of the browse information file using only the contributing .sbr files. If the file does not exist, BSCMAKE performs a full build using all .sbr files. The rules for builds are as follows:

- For a full build to succeed, all specified .sbr files must exist and must not be truncated. If an .sbr file is truncated, you must rebuild it (by recompiling or assembling) before running BSCMAKE.
- For an incremental build to succeed, the .bsc file must exist. All contributing .sbr files, even empty files, must exist and must be specified on the BSCMAKE command line. If you omit an .sbr file from the command line, BSCMAKE removes its contribution from the file.

See Also

Reference

Building a .bsc File

BSCMAKE Command Line

To run BSCMAKE, use the following command line syntax:

```
BSCMAKE [options] sbrfiles
```

Options can appear only in the *options* field on the command line.

The *sbrfiles* field specifies one or more .sbr files created by a compiler or assembler. Separate the names of .sbr files with spaces or tabs. You must specify the extension; there is no default. You can specify a path with the filename, and you can use operating-system wildcards (* and ?).

During an incremental build, you can specify new .sbr files that were not part of the original build. If you want all contributions to remain in the browse information file, you must specify all .sbr files (including truncated files) that were originally used to create the .bsc file. If you omit an .sbr file, that file's contribution to the browse information file is removed.

Do not specify a truncated .sbr file for a full build. A full build requires contributions from all specified .sbr files. Before you perform a full build, recompile the project and create a new .sbr file for each empty file.

The following command runs BSCMAKE to build a file called MAIN.bsc from three .sbr files:

```
BSCMAKE main.sbr file1.sbr file2.sbr
```

For related information, see BSCMAKE Command File and BSCMAKE Options.

See Also
Reference
BSCMAKE Reference

BSCMAKE Command File (Response File)

You can provide part or all of the command-line input in a command file. Specify the command file using the following syntax:

```
BSCMAKE @filename
```

Only one command file is allowed. You can specify a path with *filename*. Precede *filename* with an at sign (@). BSCMAKE does not assume an extension. You can specify additional *sbrfiles* on the command line after *filename*. The command file is a text file that contains the input to BSCMAKE in the same order as you would specify it on the command line. Separate the command-line arguments with one or more spaces, tabs, or newline characters.

The following command calls BSCMAKE using a command file:

```
BSCMAKE @prog1.txt
```

The following is a sample command file:

```
/n /v /o main.bsc /El
/S (
toolbox.h
verdate.h c:\src\inc\screen.h
)
file1.sbr file2.sbr file3.sbr file4.sbr
```

See Also

Reference

BSCMAKE Reference

BSCMAKE Options

This section describes the options available for controlling BSCMAKE. Several options control the content of the browse information file by excluding or including certain information. The exclusion options can allow BSCMAKE to run faster and may result in a smaller .bsc file. Option names are case sensitive (except for /HELP and /NOLOGO) .

Only /NOLOGO and /o are available from within the Visual Studio development environment. See Setting Visual C++ Project Properties for information on access a project's property pages.

/Ei (*filename*...)

Excludes the contents of the specified include files from the browse information file. To specify multiple files, separate the names with a space and enclose the list in parentheses. Parentheses are not necessary if you specify only one *filename*. Use /Ei along with the /Es option to exclude files not excluded by /Es.

/El

Excludes local symbols. The default is to include local symbols. For more information about local symbols, see Creating an .sbr File.

/Em

Excludes symbols in the body of macros. Use /Em to include only the names of macros in the browse information file. The default is to include both the macro names and the result of the macro expansions.

/Er (*symbol*...)

Excludes the specified symbols from the browse information file. To specify multiple symbol names, separate the names with a space and enclose the list in parentheses. Parentheses are not necessary if you specify only one *symbol*.

/Es

Excludes from the browse information file every include file specified with an absolute path or found in an absolute path specified in the INCLUDE environment variable. (Usually, these are the system include files, which contain a lot of information that you may not need in your browse information file.) This option does not exclude files specified without a path or with relative paths or files found in a relative path in INCLUDE. You can use the /Ei option along with /Es to exclude files that /Es does not exclude. If you want to exclude only some of the files that /Es excludes, use /Ei instead of /Es and list the files you want to exclude.

/errorreport:[none | prompt | queue | send]

Allows you to send information to Microsoft regarding internal errors in bscmake.exe. For more information on /errorreport, see /errorReport (Report Internal Compiler Errors).

/HELP

Displays a summary of the BSCMAKE command-line syntax.

/Iu

Includes unreferenced symbols. By default, BSCMAKE does not record any symbols that are defined but not referenced. If an .sbr file has been packed, this option has no effect for that input file because the compiler has already removed the unreferenced symbols.

/n

Forces a nonincremental build. Use /n to force a full build of the browse information file whether or not a .bsc file exists and to prevent .sbr files from being truncated. See How BSCMAKE Builds a .bsc File.

/NOLOGO

Suppresses the BSCMAKE copyright message.

/o *filename*

Specifies a name for the browse information file. By default, BSCMAKE gives the browse information file the base name of the first .sbr file and a .bsc extension.

/S (*filename*...)

Tells BSCMAKE to process the specified include file the first time it is encountered and to exclude it otherwise. Use this option to save processing time when a file (such as a header, or .h, file for a .c or .cpp source file) is included in several source files but is unchanged by preprocessing directives each time. You may also want to use this option if a file is changed in ways that are unimportant for the browse information file you are creating. To specify multiple files, separate the names with a space and enclose the list in parentheses. Parentheses are not necessary if you specify only one *filename*. If you want to exclude the file every time it is included, use the /Ei or /Es option.

/v

Provides verbose output, which includes the name of each .sbr file being processed and information about the complete BSCMAKE run.

/?

Displays a brief summary of BSCMAKE command-line syntax.

The following command line tells BSCMAKE to do a full build of MAIN.bsc from three .sbr files. It also tells BSCMAKE to exclude duplicate instances of TOOLBOX.h:

```
BSCMAKE /n /S toolbox.h /o main.bsc file1.sbr file2.sbr file3.sbr
```

See Also

Reference

BSCMAKE Reference

BSCMAKE Exit Codes

BSCMAKE returns an exit code (also called a return code or error code) to the operating system or the calling program.

Code	Meaning
0	No error
1	Command-line error
4	Fatal error during build

See Also

Reference

BSCMAKE Reference

LIB Reference

The Microsoft Library Manager (LIB.exe) creates and manages a library of Common Object File Format (COFF) object files. LIB can also be used to create export files and import libraries to reference exported definitions.

- Overview of LIB
- How to: Set LIB.EXE Options in the Visual Studio Development Environment
- Running LIB
- Managing a Library
- Extracting a Library Member
- Working with Import Libraries and Export Files

See Also

Reference

C/C++ Build Tools

Overview of LIB

LIB creates standard libraries, import libraries, and export files you can use with LINK when building a program. LIB runs from a command prompt.

You can use LIB in the following modes:

- Building or modifying a COFF library
- Extracting a member object to a file
- Creating an export file and an import library

These modes are mutually exclusive; you can use LIB in only one mode at a time.

Lib Options

The following table lists the options for lib.exe, with a link to more information.

/DEF

Create an import library and an export file.

For more information see Building an Import Library and Export File.

/ERRORREPORT

Send information to Microsoft about internal errors with lib.exe.

For more information see Running LIB.

/EXPORT

Exports a function from your program.

For more information see Building an Import Library and Export File.

/EXTRACT

Create an object (.obj) file that contains a copy of a member of an existing library.

For more information see Extracting a Library Member.

/INCLUDE

Adds a symbol to the symbol table.

For more information see Building an Import Library and Export File.

/LIBPATH

Overrides the environment library path.

For more information see Managing a Library.

/LIST

Displays information about the output library to standard output.

For more information see Managing a Library.

/LTCG

Causes the library to be built using link-time code generation.

For more information see Running LIB.

/MACHINE

Specifies the target platform for the program.

For more information see Running LIB.

/NAME

When building an import library, specifies the name of the DLL for which the import library is being built.

For more information see Managing a Library.

/NODEFAULTLIB

Removes one or more default libraries from the list of libraries it searches when resolving external references.

For more information see Managing a Library.

/NOLOGO

Suppresses display of the LIB copyright message and version number and prevents echoing of command files.

For more information see Running LIB.

/OUT

Overrides the default output filename.

For more information see Managing a Library.

/REMOVE

Omits an object from the output library.

For more information see Managing a Library.

/SUBSYSTEM

Tells the operating system how to run a program created by linking to the output library.

For more information see Managing a Library.

/VERBOSE

Displays details about the progress of the session, including names of the .obj files being added.

For more information see Running LIB.

/WX

Treat warnings as errors.

For more information see Running LIB.

See Also

Reference

LIB Reference
LIB Input Files
LIB Output Files
Other LIB Output
Structure of a Library

How to: Set LIB.EXE Options in the Visual Studio Development Environment

To set LIB.EXE options in the Visual Studio development environment

1. Access the project's Property Page dialog box.
2. With a static library project active, select the **Librarian** folder.
3. Select either the General or Input/Output property page.
4. Modify properties as needed.

See Also

Reference

LIB Reference

LIB Input Files

The input files expected by LIB depend on the mode in which it is being used, as shown in the following table.

Mode	Input
Default (building or modifying a library)	COFF object (.obj) files, COFF libraries (.lib), 32-bit Object Model Format (OMF) object (.obj) files
Extracting a member with /EXTRACT	COFF library (.lib)
Building an export file and import library with /DEF	Module-definition (.def) file, COFF object (.obj) files, COFF libraries (.lib), 32-bit OMF object (.obj) files

Note
OMF libraries created by the 16-bit version of LIB cannot be used as input to the 32-bit version of LIB.

See Also

Reference

Overview of LIB

LIB Output Files

The output files produced by LIB depend on the mode in which it is being used, as shown in the following table.

Mode	Output
Default (building or modifying a library)	COFF library (.lib)
Extracting a member with /EXTRACT	Object (.obj) file
Building an export file and import library with /DEF	Import library (.lib) and export (.exp) file

See Also

Reference

Overview of LIB

Other LIB Output

In the default mode, you can use the /LIST option to display information about the resulting library. You can redirect this output to a file.

LIB displays a copyright and version message and echoes command files unless the /NOLOGO option is used.

When you type `lib` with no other input, LIB displays a usage statement that summarizes its options.

Error and warning messages issued by LIB have the form LNK*nnnn*. The LINK, DUMPBIN, and EDITBIN tools also use this range of errors. Help is available by selecting the error in the Output window and pressing F1.

See Also

Reference

Overview of LIB

Structure of a Library

A library contains COFF objects. Objects in a library contain functions and data that can be referenced externally by other objects in a program. An object in a library is sometimes referred to as a library member.

You can get additional information about the contents of a library by running the DUMPBIN tool with the /LINKERMEMBER option. For more information about this option, see DUMPBIN Reference.

See Also

Reference

Overview of LIB

Running LIB

Various command-line options can be used to control LIB.

LIB Command Line

To run LIB, type the command `lib` followed by the options and file names for the task you are using LIB to perform. LIB also accepts command-line input in command files, which are described in the following section. LIB does not use an environment variable.

Note
If you are accustomed to the LINK32.exe and LIB32.exe tools provided with the Microsoft Win32 Software Development Kit f or Windows NT, you may have been using either the command `link32 -lib` or the command `lib32` for managing libraries a nd creating import libraries. Be sure to change your makefiles and batch files to use the `lib` command instead.

LIB Command Files

You can pass command-line arguments to LIB in a command file using the following syntax:

```
LIB @commandfile
```

The file *commandfile* is a text file. No space or tab is allowed between the at sign (@) and the file name. There is no default extension; you must specify the full file name, including any extension. Wildcards cannot be used. You can specify an absolute or relative path with the file name.

In the command file, arguments can be separated by spaces or tabs, as they can on the command line; they can also be separated by newline characters. Use a semicolon (;) to mark a comment. LIB ignores all text from the semicolon to the end of the line.

You can specify either all or part of the command line in a command file, and you can use more than one command file in a LIB command. LIB accepts the command-file input as if it were specified in that location on the command line. Command files cannot be nested. LIB echoes the contents of command files unless the /NOLOGO option is used.

Using LIB Options

An option consists of an option specifier, which is either a dash (–) or a forward slash (/), followed by the name of the option. Option names cannot be abbreviated. Some options take an argument, specified after a colon (:). No spaces or tabs are allowed within an option specification. Use one or more spaces or tabs to separate option specifications on the command line. Option names and their keyword or file name arguments are not case sensitive, but identifiers used as arguments are case sensitive. LIB processes options in the order specified on the command line and in command files. If an option is repeated with different arguments, the last one to be processed takes precedence.

The following options apply to all modes of LIB:

/ERRORREPORT [NONE | PROMPT | QUEUE | SEND]

If lib.exe fails at runtime, you can use /ERRORREPORT to send information to Microsoft about these internal errors.

For more information about /ERRORREPORT, see /errorReport (Report Internal Compiler Errors).

/LTCG

Causes the library to be built using link-time code generation. For more information, see /LTCG.

/MACHINE

Specifies the target platform for the program. Usually, you do not need to specify /MACHINE. LIB infers the machine type from the .obj files. However, in some circumstances, LIB cannot determine the machine type and issues an error message. If such an error occurs, specify /MACHINE. In /EXTRACT mode, this option is for verification only. Use `lib /?` at the command line to see available machine types.

/NOLOGO

Suppresses display of the LIB copyright message and version number and prevents echoing of command files.

/VERBOSE

Displays details about the progress of the session, including names of the .obj files being added. The information is sent to standard output and can be redirected to a file.

/WX[:NO]

Treat warnings as errors. See /WX (Treat Linker Warnings as Errors) for more information.

Other options apply only to specific modes of LIB. These options are discussed in the sections describing each mode.

See Also

Reference

LIB Reference

Managing a Library

The default mode for LIB is to build or modify a library of COFF objects. LIB runs in this mode when you do not specify /EXTRACT (to copy an object to a file) or /DEF (to build an import library).

To build a library from objects and/or libraries, use the following syntax:

```
LIB [options...] files...
```

This command creates a library from one or more input *files*. The *files* can be COFF object files, 32-bit OMF object files, or existing COFF libraries. LIB creates one library that contains all objects in the specified files. If an input file is a 32-bit OMF object file, LIB converts it to COFF before building the library. LIB cannot accept a 32-bit OMF object that is in a library created by the 16-bit version of LIB. You must first use the 16-bit LIB to extract the object; then you can use the extracted object file as input to the 32-bit LIB.

By default, LIB names the output file using the base name of the first object or library file and the extension .lib. The output file is put in the current directory. If a file already exists with the same name, the output file replaces the existing file. To preserve an existing library, use the /OUT option to specify a name for the output file.

The following options apply to building and modifying a library:

/LIBPATH: *dir*

Overrides the environment library path. For details, see the description of the LINK /LIBPATH option.

/LIST

Displays information about the output library to standard output. The output can be redirected to a file. You can use /LIST to determine the contents of an existing library without modifying it.

/NAME: *filename*

When building an import library, specifies the name of the DLL for which the import library is being built.

/NODEFAULTLIB

Removes one or more default libraries from the list of libraries it searches when resolving external references. See /NODEFAULTLIB for more information.

/OUT: *filename*

Overrides the default output filename. By default, the output library is created in the current directory, with the base name of the first library or object file on the command line and the extension .lib.

/REMOVE: *object*

Omits the specified *object* from the output library. LIB creates an output library by combining all objects (whether in object files or libraries), and then deleting any objects specified with /REMOVE.

/SUBSYSTEM:{CONSOLE | EFI_APPLICATION | EFI_BOOT_SERVICE_DRIVER | EFI_ROM | EFI_RUNTIME_DRIVER | NATIVE | POSIX | WINDOWS | WINDOWSCE}[,#[.##]]

Tells the operating system how to run a program created by linking to the output library. For more information, see the description of the LINK /SUBSYSTEM option.

LIB options specified on the command line are not case sensitive.

You can use LIB to perform the following library-management tasks:

- To add objects to a library, specify the file name for the existing library and the filenames for the new objects.
- To combine libraries, specify the library file names. You can add objects and combine libraries with a single LIB command.
- To replace a library member with a new object, specify the library containing the member object to be replaced and the file name for the new object (or the library that contains it). When an object that has the same name exists in more than one input file, LIB puts the last object specified in the LIB command into the output library. When you replace a library member, be sure to specify the new object or library after the library that contains the old object.

- To delete a member from a library, use the /REMOVE option. LIB processes any specifications of /REMOVE after combining all input objects, regardless of command-line order.

Note
You cannot both delete a member and extract it to a file in the same step. You must first extract the member object using /EXTRACT, then run LIB again using /REMOVE. This behavior differs from that of the 16-bit LIB (for OMF libraries) provided in other Microsoft products.

See Also

Reference

LIB Reference

Extracting a Library Member

You can use LIB to create an object (.obj) file that contains a copy of a member of an existing library. To extract a copy of a member, use the following syntax:

```
LIB library /EXTRACT:member /OUT:objectfile
```

This command creates an .obj file called *objectfile* that contains a copy of a *member* of a *library*. The *member* name is case sensitive. You can extract only one member in a single command. The /OUT option is required; there is no default output name. If a file called *objectfile* already exists in the specified directory (or the current directory, if no directory is specified with *objectfile*), the extracted *objectfile* replaces the existing file.

See Also

Reference

LIB Reference

Working with Import Libraries and Export Files

You can use LIB with the /DEF option to create an import library and an export file. LINK uses the export file to build a program that contains exports (usually a dynamic-link library (DLL)), and it uses the import library to resolve references to those exports in other programs.

Note that if you create your import library in a preliminary step, before creating your .dll, you must pass the same set of object files when building the .dll, as you passed when building the import library.

In most situations, you do not need to use LIB to create your import library. When you link a program (either an executable file or a DLL) that contains exports, LINK automatically creates an import library that describes the exports. Later, when you link a program that references those exports, you specify the import library.

However, when a DLL exports to a program that it also imports from, whether directly or indirectly, you must use LIB to create one of the import libraries. When LIB creates an import library, it also creates an export file. You must use the export file when linking one of the DLLs.

See Also

Reference

LIB Reference

Building an Import Library and Export File

To build an import library and export file, use the following syntax:

```
LIB /DEF[:deffile] [options] [objfiles] [libraries]
```

When /DEF is specified, LIB creates the output files from export specifications that are passed in the LIB command. There are three methods for specifying exports, listed in recommended order of use:

1. A **__declspec(dllexport)** definition in one of the *objfiles* or *libraries*
2. A specification of /EXPORT:*name* on the LIB command line
3. A definition in an **EXPORTS** statement in a *deffile*

These are the same methods you use to specify exports when linking an exporting program. A program can use more than one method. You can specify parts of the LIB command (such as multiple *objfiles* or /EXPORT specifications) in a command file in the LIB command, just as you can in a LINK command.

The following options apply to building an import library and export file:

/OUT: *import*

Overrides the default output file name for the *import* library being created. When /OUT is not specified, the default name is the base name of the first object file or library in the LIB command and the extension .lib. The export file is given the same base name as the import library and the extension .exp.

/EXPORT: *entryname*[= *internalname*][,@ *ordinal*[, **NONAME**]][, **DATA**]

Exports a function from your program to allow other programs to call the function. You can also export data (using the **DATA** keyword). Exports are usually defined in a DLL.

The *entryname* is the name of the function or data item as it is to be used by the calling program. Optionally, you can specify the *internalname* as the function known in the defining program; by default, *internalname* is the same as *entryname*. The *ordinal* specifies an index into the export table in the range 1 through 65,535; if you do not specify *ordinal*, LIB assigns one. The **NONAME** keyword exports the function only as an ordinal, without an *entryname*. The **DATA** keyword is used to export data-only objects.

/INCLUDE: *symbol*

Adds the specified symbol to the symbol table. This option is useful for forcing the use of a library object that otherwise would not be included.

Note that if you create your import library in a preliminary step, before creating your .dll, you must pass the same set of object files when building the .dll, as you passed when building the import library.

See Also

Reference

Working with Import Libraries and Export Files

Using an Import Library and Export File

When a program (either an executable file or a DLL) exports to another program that it also imports from, or if more than two programs both export to and import from each other, the commands to link these programs must accommodate circular exports.

In a situation without circular exports, when linking a program that uses exports from another program, you must specify the import library for the exporting program. The import library for the exporting program is created when you link that exporting program. Therefore, you must link the exporting program before the importing program. For example, if TWO.dll imports from ONE.dll, you must first link ONE.dll and get the import library ONE.lib. Then, you specify ONE.lib when linking TWO.dll. When the linker creates TWO.dll, it also creates its import library, TWO.lib. Use TWO.lib when linking programs that import from TWO.dll.

However, in a circular export situation, it is not possible to link all of the interdependent programs using import libraries from the other programs. In the example discussed earlier, if TWO.dll also exports to ONE.dll, the import library for TWO.dll won't exist yet when ONE.dll is linked. When circular exports exist, you must use LIB to create an import library and export file for one of the programs.

To begin, choose one of the programs on which to run LIB. In the LIB command, list all objects and libraries for the program and specify /DEF. If the program uses a .def file or /EXPORT specifications, specify these as well.

After you create the import library (.lib) and the export file (.exp) for the program, you use the import library when linking the other program or programs. LINK creates an import library for each exporting program it builds. For example, if you run LIB on the objects and exports for ONE.dll, you create ONE.lib and ONE.exp. You can now use ONE.lib when linking TWO.dll; this step also creates the import library TWO.lib.

Finally, link the program you began with. In the LINK command, specify the objects and libraries for the program, the .exp file that LIB created for the program, and the import library or libraries for the exports used by the program. To continue the example, the LINK command for ONE.dll contains ONE.exp and TWO.lib, as well as the objects and libraries that go into ONE.dll. Do not specify the .def file or /EXPORT specifications in the LINK command; these are not needed, because the export definitions are contained in the .exp file. When you link using an .exp file, LINK does not create an import library, because it assumes that one was created when the .exp file was created.

See Also

Reference

Working with Import Libraries and Export Files

EDITBIN Reference

The Microsoft COFF Binary File Editor (EDITBIN.EXE) modifies Common Object File Format (COFF) binary files. You can use EDITBIN to modify object files, executable files, and dynamic-link libraries (DLL).

Note
EDITBIN runs only from the command line.

EDITBIN is not available for use on files produced with the /GL compiler option. Any modifications to binary files produced with /GL will have to be achieved by recompiling and linking.

- EDITBIN command line
- EDITBIN options

See Also
Reference
C/C++ Build Tools

EDITBIN Command Line

To run EDITBIN, use the following syntax:

```
EDITBIN [options] files...
```

Specify one or more files for the objects or images to be changed, and one or more options for changing the files.

When you type the command `editbin` without any other command-line input, EDITBIN displays a usage statement that summarizes its options.

See Also

Reference

C/C++ Build Tools

EDITBIN Reference

EDITBIN Options

An option consists of an option specifier, which is either a dash (–) or a forward slash (/), followed by the name of the option. Option names cannot be abbreviated. Some options take arguments, specified after a colon (:). No spaces or tabs are allowed within an option specification. Use one or more spaces or tabs to separate option specifications on the command line. Option names and their keyword or file name arguments are not case sensitive.

EDITBIN has the following options:

- /ALLOWBIND
- /ALLOWISOLATION
- /BIND
- /ERRORREPORT
- /HEAP
- /LARGEADDRESSAWARE
- /NOLOGO
- /NXCOMPAT
- /REBASE
- /RELEASE
- /SECTION
- /STACK
- /SUBSYSTEM
- /SWAPRUN
- /TSAWARE
- /VERSION

See Also

Reference

C/C++ Build Tools

EDITBIN Reference

/ALLOWISOLATION

Allows you to modify a binary in the same way as if you had used the /ALLOWISOLATION (Manifest Lookup) linker option.

```
/ALLOWISOLATION[:NO]
```

See Also

Reference

EDITBIN Options

/ALLOWBIND

```
/ALLOWBIND[:NO]
```

Remarks

The /ALLOWBIND option to the EDITBIN utility allows you to modify a DLL in the same way as if you had used the /ALLOWBIND linker option.

See Also

Reference

EDITBIN Options

/BIND

```
/BIND[:PATH=path]
```

Remarks

This option sets the addresses of the entry points in the import address table for an executable file or DLL. Use this option to reduce load time of a program.

Specify the program's executable file and DLLs in the *files* argument on the EDITBIN command line. The optional *path* argument to /BIND specifies the location of the DLLs used by the specified files. Separate multiple directories with a semicolon (**;**). If *path* is not specified, EDITBIN searches the directories specified in the PATH environment variable. If *path* is specified, EDITBIN ignores the PATH variable.

By default, the program loader sets the addresses of entry points when it loads a program. The amount of time this process takes varies, depending on the number of DLLs and the number of entry points referenced in the program. If a program has been modified with /BIND, and if the base addresses for the executable file and its DLLs do not conflict with DLLs that are already loaded, the operating system does not need to set these addresses. In a situation where the files are incorrectly based, the operating system relocates the program's DLLs and recalculates the entry-point addresses, which adds to the program's load time.

See Also
Reference
EDITBIN Options

/ERRORREPORT (editbin.exe)

```
/ERRORREPORT[NONE | PROMPT | QUEUE | SEND ]
```

Remarks

If editbin.exe fails at runtime, you can use **/ERRORREPORT** to send information to Microsoft about these internal errors.

For more information about **/ERRORREPORT**, see /errorReport (Report Internal Compiler Errors).

See Also

Reference

EDITBIN Options

/HEAP

```
/HEAP:reserve[,commit]
```

Remarks

This option sets the size of the heap in bytes.

The *reserve* argument specifies the total heap allocation in virtual memory. The default heap size is 1 MB. The linker rounds up the specified value to the nearest 4 bytes.

The optional *commit* argument is subject to interpretation by the operating system. On a Windows operating system, it specifies the amount of physical memory to allocate at a time. Committed virtual memory causes space to be reserved in the paging file. A higher *commit* value saves time when the application needs more heap space but increases the memory requirements and possibly the startup time.

Specify the *reserve* and *commit* values in decimal or C-language notation.

See Also

Reference

EDITBIN Options

/LARGEADDRESSAWARE

```
/LARGEADDRESSWARE
```

Remarks

This option edits the image to indicate that the application can handle addresses larger than 2 gigabytes.

See Also

Reference

EDITBIN Options

/NOLOGO (EDITBIN)

```
/NOLOGO
```

Remarks

This option suppresses display of the EDITBIN copyright message and version number.

See Also

Reference

EDITBIN Options

/NXCOMPAT

```
/NXCOMPAT[:NO]
```

Remarks

Indicates that an executable was tested to be compatible with the Windows Data Execution Prevention feature.

For more information, see /NXCOMPAT (Compatible with Data Execution Prevention).

See Also

Reference

EDITBIN Options

/REBASE

```
/REBASE[:modifiers]
```

Remarks

This option sets the base addresses for the specified files. EDITBIN assigns new base addresses in a contiguous address space according to the size of each file rounded up to the nearest 64 KB. For details about base addresses, see the Base Address (/BASE) linker option.

Specify the program's executable files and DLLs in the *files* argument on the EDITBIN command line in the order in which they are to be based. You can optionally specify one or more *modifiers*, each separated by a comma (**,**):

Modifier	**Action**
BASE=*address*	Provides a beginning address for reassigning base addresses to the files. Specify *address* in decimal or C-language notation. If BASE is not specified, the default starting base address is 0x400000. If DOWN is used, BASE must be specified, and *address* sets the end of the range of base addresses.
BASEFILE	Creates a file named COFFBASE.TXT, which is a text file in the format expected by LINK's /BASE option.
DOWN	Tells EDITBIN to reassign base addresses downward from an ending address. The files are reassigned in the order specified, with the first file located in the highest possible address below the end of the address range. BASE must be used with DOWN to ensure sufficient address space for basing the files. To determine the address space needed by the specified files, run EDITBIN with /REBASE on the files and add 64 KB to the displayed total size.

See Also

Reference

EDITBIN Options

/RELEASE

```
/RELEASE
```

Remarks

This option sets the checksum in the header of an executable file.

The operating system requires the checksum for device drivers. It is recommended that you set the checksum for release versions of your device drivers to ensure compatibility with future operating systems.

See Also

Reference

EDITBIN Options

/SECTION (EDITBIN)

```
/SECTION:name[=newname][,attributes][,alignment]
```

Remarks

This option changes the attributes of a section, overriding the attributes that were set when the object file for the section was compiled or linked.

After the colon (**:**), specify the *name* of the section. To change the section name, follow *name* with an equal sign (=) and a *newname* for the section.

To set or change the section's *attributes*, specify a comma (**,**) followed by one or more attributes characters. To negate an attribute, precede its character with an exclamation point (!). The following characters specify memory attributes:

Attribute	**Setting**
c	code
d	discardable
e	executable
i	initialized data
k	cached virtual memory
m	link remove
o	link info
p	paged virtual memory
r	read
s	shared
u	uninitialized data
w	write

To control *alignment*, specify the character **A** followed by one of the following characters to set the size of alignment in bytes, as follows:

Character	**Alignment size in bytes**
1	1
2	2
4	4
8	8
p	16

t	32
s	64
x	no alignment

Specify the *attributes* and *alignment* characters as a string with no white space. The characters are not case sensitive.

See Also

Reference

EDITBIN Options

/STACK

```
/STACK:reserve[,commit]
```

Remarks

This option sets the size of the stack in bytes and takes arguments in decimal or C-language notation. The /STACK option applies only to an executable file.

The *reserve* argument specifies the total stack allocation in virtual memory. EDITBIN rounds up the specified value to the nearest 4 bytes.

The optional *commit* argument is subject to interpretation by the operating system. In Windows NT, Windows 95, and Windows 98, *commit* specifies the amount of physical memory to allocate at a time. Committed virtual memory causes space to be reserved in the paging file. A higher *commit* value saves time when the application needs more stack space but increases the memory requirements and possibly startup time.

See Also

Reference

EDITBIN Options

/SUBSYSTEM

```
/SUBSYSTEM:{CONSOLE|EFI_APPLICATION|EFI_BOOT_SERVICE_DRIVER|
EFI_ROM|EFI_RUNTIME_DRIVER|NATIVE|POSIX|WINDOWS|WINDOWSCE}[,left[,right]]
```

Remarks

This option edits the image to indicate which subsystem the operating system must invoke for execution.

You can specify any of the following subsystems:

- The CONSOLE subsystem handles a Win32 character-mode application that use a console supplied by the operating system.
- Extensible Firmware Interface. The EFI_* subsystems. See the EFI specification for more information. For example, see the Intel web site.
- The WINDOWS subsystem handles an application that does not require a console and creates its own windows, if required.
- The NATIVE subsystem handles a Windows NT device driver.
- The WINDOWSCE subsystem handles Windows CE consumer electronics applications.
- The POSIX subsystem handles a POSIX application in Windows NT.

The optional *left* and *right* values specify the minimum required version of the specified subsystem:

- The whole number part of the version number, the portion to the left of the decimal point, is represented by *left*.
- The fractional part of the version number, the portion to the right of the decimal point, is represented by *right*.
- The values of *left* and *right* must be from 0 through 65,535.

The default is version 4.00 for CONSOLE, WINDOWS, and NATIVE; and version 19.90 for POSIX.

The choice of subsystem affects the default starting address for the program. For more information, see the linker Entry-Point Symbol (/ENTRY:*function*) option.

See the /SUBSYSTEM linker option for more information.

See Also

Reference

EDITBIN Options

/SWAPRUN

```
/SWAPRUN:{[!]NET|[!]CD}
```

Remarks

This option edits the image to tell the operating system to copy the image to a swap file and run it from there. Use this option for images that reside on networks or removable media.

You can add or remove the NET or CD qualifiers:

- NET indicates that the image resides on a network.
- CD indicates that the image resides on a CD-ROM or similar removable medium.
- Use !NET and !CD to reverse the effects of NET and CD.

See Also

Reference

EDITBIN Options

/TSAWARE

```
/TSAWARE[:NO]
```

Remarks

The /TSAWARE option to the EDITBIN utility allows you to modify a program image the same way as if you had used the /TSAWARE linker option.

See Also

Reference

EDITBIN Options

/VERSION

```
/VERSION:left[,right]
```

Remarks

This option places a version number into the header of the image.

The whole number part of the version number, the portion to the left of the decimal point, is represented by *left*. The fractional part of the version number, the portion to the right of the decimal point, is represented by *right*.

See Also

Reference

EDITBIN Options

DUMPBIN Reference

The Microsoft COFF Binary File Dumper (DUMPBIN.EXE) displays information about Common Object File Format (COFF) binary files. You can use DUMPBIN to examine COFF object files, standard libraries of COFF objects, executable files, and dynamic-link libraries (DLLs).

Note
DUMPBIN runs only from the command line.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

- DUMPBIN command line
- DUMPBIN options

See Also
Reference
C/C++ Build Tools

DUMPBIN Command Line

To run DUMPBIN, use the following syntax:

```
DUMPBIN [options] files...
```

Specify one or more binary files, along with any options required to control the information. DUMPBIN displays the information to standard output. You can either redirect it to a file or use the /OUT option to specify a file name for the output.

When you run DUMPBIN on a file without specifying an option, DUMPBIN displays the /SUMMARY output.

When you type the command `dumpbin` without any other command-line input, DUMPBIN displays a usage statement that summarizes its options.

See Also

Reference

C/C++ Build Tools

DUMPBIN Reference

DUMPBIN Options

An option consists of an option specifier, which is either a dash (–) or a forward slash (/), followed by the name of the option. Option names cannot be abbreviated. Some options take arguments, specified after a colon (:). No spaces or tabs are allowed within an option specification. Use one or more spaces or tabs to separate option specifications on the command line. Option names and their keyword or file name arguments are not case sensitive. Most options apply to all binary files; a few apply only to certain types of files. By default, DUMPBIN sends information to standard output. Use the /OUT option to send output to a file.

DUMPBIN has the following options:

- /ALL
- /ARCHIVEMEMBERS
- /CLRHEADER
- /DEPENDENTS
- /DIRECTIVES
- /DISASM
- /ERRORREPORT (dumpbin.exe)
- /EXPORTS
- /FPO
- /HEADERS
- /IMPORTS
- /LINENUMBERS
- /LINKERMEMBER
- /LOADCONFIG
- /OUT
- /PDBPATH
- /PDATA
- /RANGE
- /RAWDATA
- /RELOCATIONS
- /SECTION
- /SUMMARY
- /SYMBOLS
- /TLS
- /UNWINDINFO

See Also

Reference

C/C++ Build Tools

DUMPBIN Reference

/ALL

```
/ALL
```

Remarks

This option displays all available information except code disassembly. Use /DISASM to display disassembly. You can use /RAWDATA:NONE with /ALL to omit the raw binary details of the file.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

/ARCHIVEMEMBERS

```
/ARCHIVEMEMBERS
```

Remarks

This option displays minimal information about member objects in a library.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

/CLRHEADER

```
/CLRHEADER file
```

Remarks

where:

file

An image file built with /clr.

Remarks

CLRHEADER displays information about the .NET headers used in any managed program. The output shows the location and size, in bytes, of the .NET header and sections in the header.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

When /CLRHEADER is used on a file that was compiled with /clr, there will be a **clr Header:** section in the dumpbin output. The value of **flags** indicates which /clr option was used:

- 0 -- /clr (image may contain native code).
- 1 -- /clr:safe (image is MSIL only, able to run on any CLR platform, and possibly verifiable).
- 3 -- /clr:pure (image is MSIL only, but only able to run on x86 platforms).

You can also programmatically check if an image was built for the common language runtime. For more information, see How to: Determine if an Image is Native or CLR.

See Also

Reference

DUMPBIN Options

/DEPENDENTS

```
/DEPENDENTS
```

Remarks

Dumps the names of the DLLs from which the image imports functions. Does not dump the names of the imported functions.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

/DIRECTIVES

```
/DIRECTIVES
```

Remarks

This option dumps the compiler-generated .drective section of an image.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

/DISASM

```
/DISASM
```

Remarks

This option displays disassembly of code sections, using symbols if present in the file.

/DISASM should only be used on native (not managed) images.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

/ERRORREPORT (dumpbin.exe)

```
/ERRORREPORT[NONE | PROMPT | QUEUE | SEND ]
```

Remarks

If dumpbin.exe fails at runtime, you can use **/ERRORREPORT** to send information to Microsoft about these internal errors.

For more information about **/ERRORREPORT**, see /errorReport (Report Internal Compiler Errors).

See Also

Reference

DUMPBIN Options

/EXPORTS

```
/EXPORTS
```

Remarks

This option displays all definitions exported from an executable file or DLL.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

/FPO

```
/FPO
```

Remarks

This option displays frame pointer optimization (FPO) records.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

/HEADERS

```
/HEADERS
```

Remarks

This option displays the file header and the header for each section. When used with a library, it displays the header for each member object.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

/IMPORTS (DUMPBIN)

```
/IMPORTS[:file]
```

This option displays the list of DLLs (both statically linked and delay loaded) that are imported to an executable file or DLL and all the individual imports from each of these DLLs.

The optional *file* specification allows you to specify that the imports for only that DLL will be displayed. For example:

```
dumpbin /IMPORTS:msvcrt.dll
```

Remarks

The output displayed by this option is similar to the /EXPORTS output.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

/LINENUMBERS

```
/LINENUMBERS
```

Remarks

This option displays COFF line numbers. Line numbers exist in an object file if it was compiled with Program Database (/Zi), C7 Compatible (/Z7), or Line Numbers Only (/Zd). An executable file or DLL contains COFF line numbers if it was linked with Generate Debug Info (/DEBUG).

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

/LINKERMEMBER

```
/LINKERMEMBER[:{1|2}]
```

Remarks

This option displays public symbols defined in a library. Specify the 1 argument to display symbols in object order, along with their offsets. Specify the 2 argument to display offsets and index numbers of objects, and then list the symbols in alphabetical order, along with the object index for each. To get both outputs, specify /LINKERMEMBER without the number argument.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

/LOADCONFIG

```
/LOADCONFIG
```

Remarks

This option dumps the IMAGE_LOAD_CONFIG_DIRECTORY structure, an optional structure that is used by the Windows NT loader and defined in WINNT.H.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

/OUT (DUMPBIN)

```
/OUT:filename
```

Remarks

This option specifies a *filename* for the output. By default, DUMPBIN displays the information to standard output.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

/PDATA

```
/PDATA
```

Remarks

RISC processors only.

This option dumps the exception tables (.pdata) from an image or object.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

/PDBPATH

```
/PDBPATH[:VERBOSE] filename
```

Remarks

where:

filename

The name of the .dll or .exe file for which you want to find the matching .pdb file.

VERBOSE (optional)

Reports all directories where an attempt was made to locate the .pdb file.

Remarks

/PDBPATH will search your computer along the same paths that the debugger would search for a .pdb file and will report which, if any, .pdb files correspond to the file specified in *filename*.

When using the Visual Studio debugger, you may experience a problem due to the fact that the debugger is using a .pdb file for a different version of the file you are debugging.

/PDBPATH will search for .pdb files along the following paths:

- Check the location where the executable resides.
- Check the location of the PDB written into the executable. This is usually the location at the time the image was linked.
- Check along the search path configured in the Visual Studio IDE.
- Check along the paths in the _NT_SYMBOL_PATH and _NT_ALT_SYMBOL_PATH environment variables.
- Check in the Windows directory.

See Also

Reference

DUMPBIN Options

/RANGE

Modifies the output of dumpbin when used with other dumpbin options, such as /RAWDATA or /DISASM.

```
/RANGE:vaMin[,vaMax]
```

Flags

vaMin

The virtual address at which you want the dumpbin operation to begin.

vaMax (optional)

The virtual address at which you want the dumpbin operation to end. If not specified, dumpbin will go to the end of the file.

Remarks

To see the virtual addresses for an image, use the map file for the image (RVA + Base), the **/DISASM** or **/HEADERS** option of dumpbin, or the disassembly window in the Visual Studio debugger.

Example

In this example, **/range** is used to modify the display of the **/disasm** option. In this example, the starting value is expressed as a decimal number and the ending value is specified as a hex number.

```
dumpbin /disasm /range:4219334,0x004061CD t.exe
```

See Also

Reference

DUMPBIN Options

/RAWDATA

```
/RAWDATA[:{1|2|4|8|NONE[,number]]
```

Remarks

This option displays the raw contents of each section in the file. The arguments control the format of the display, as shown below:

Argument	**Result**
1	The default. Contents are displayed in hexadecimal bytes, and also as ASCII characters if they have a printed representation.
2	Contents are displayed as hexadecimal 2-byte values.
4	Contents are displayed as hexadecimal 4-byte values.
8	Contents are displayed as hexadecimal 8-byte values.
NONE	Raw data is suppressed. This argument is useful to control the output of /ALL.
Number	Displayed lines are set to a width that holds *number* values per line.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

/RELOCATIONS

```
/RELOCATIONS
```

Remarks

This option displays any relocations in the object or image.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

/SECTION (DUMPBIN)

```
/SECTION:section
```

Remarks

This option restricts the output to information on the specified *section*.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

/SUMMARY

```
/SUMMARY
```

Remarks

This option displays minimal information about sections, including total size. This option is the default if no other option is specified.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

/SYMBOLS

```
/SYMBOLS
```

This option displays the COFF symbol table. Symbol tables exist in all object files. A COFF symbol table appears in an image file only if it is linked with /DEBUG.

The following is a description of the output for /SYMBOLS. Additional information on the meaning of /SYMBOLS output can be found by looking in winnt.h (IMAGE_SYMBOL and IMAGE_AUX_SYMBOL), or COFF documentation.

Given the following sample dump:

```
Dump of file main.obj
File Type: COFF OBJECT

COFF    SYMBOL    TABLE
000    00000000   DEBUG      notype      Filename      | .file
      main.cpp
002   000B1FDB   ABS      notype      Static      | @comp.id
003   00000000   SECT1      notype      Static      | .drectve
      Section length       26, #relocs   0, #linenums    0, checksum 722C964F
005   00000000   SECT2      notype      Static      | .text
      Section length      23, #relocs      1, #linenums    0, checksum 459FF65F, selection
1 (pick no duplicates)
007   00000000   SECT2      notype ()   External      | _main
008   00000000   UNDEF      notype ()   External      | ?MyDump@@YAXXZ (void __cdecl MyDump
(void))

String Table Size = 0x10 bytes

Summary

      26 .drectve
      23 .text
```

Remarks

The following description, for lines that begin with a symbol number, describes columns that have information relevant to users:

- The first three-digit number is the symbol index/number.
- If the third column contains SECT*x*, the symbol is defined in that section of the object file. But if UNDEF appears, it is not defined in that object and must be resolved elsewhere.
- The fifth column (Static, External) tells whether the symbol is visible only within that object, or whether it is public (visible externally). A Static symbol, _sym, wouldn't be linked to a Public symbol _sym; these would be two different instances of functions named _sym.

The last column in a numbered line is the symbol name, both decorated and undecorated.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

/TLS

Displays the IMAGE_TLS_DIRECTORY structure from an executable.

Remarks

/TLS displays the fields of the TLS structure as well as the addresses of the TLS callback functions.

If a program does not use thread local storage, its image will not contain a TLS structure. See thread for more information.

IMAGE_TLS_DIRECTORY is defined in winnt.h.

See Also

Reference

DUMPBIN Options

/UNWINDINFO

```
/UNWINDINFO
```

Remarks

Dumps the unwind descriptors of the structured exception handling (SEH) tables in program images (for example, exes and dlls). /UNWINDINFO only works on Itanium Processor Family (IPF) images.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See Also

Reference

DUMPBIN Options

ERRLOOK Reference

The ERRLOOK utility, which is available from the Tools menu, retrieves a system error message or module error message based on the value entered. ERRLOOK retrieves the error message text automatically if you drag and drop a hexadecimal or decimal value from the Visual Studio debugger into the **Value** edit control. You can also enter a value either by typing it in the **Value** edit control or by pasting it from the Clipboard and clicking **Look Up**.

The accelerator keys for Copy (CTRL+C), Cut (CTRL+X), and Paste (CTRL+V) work for both the **Value** and **Error Message** edit controls if you first highlight the text.

In This Section

Value Edit Control

Describes the Value Edit control in ERRLOOK.

Error Message Edit Control

Describes the Error Message Edit control in ERRLOOK.

Modules Button

Describes the Modules button in ERRLOOK.

Look Up Button

Describes the Look Up button in ERRLOOK.

Related Sections

C/C++ Build Tools

Provides links to topics discussing the C/C++ build tools provided in Visual C++.

Value Edit Control

To use the control, enter a value, paste it from the Clipboard, or drag and drop it from the debugger into this edit control. Enter the value in hexadecimal or decimal form and then click **Look Up**. Hexadecimal values should be preceded by 0x; valid characters are 0-9, A-F, and a-f. Decimal values can be preceded by the minus sign (-); valid characters are 0-9.

See Also

Reference

ERRLOOK Reference

Error Message Edit Control

The **Error Message** box contains the text of the system error message or module error message based on the value entered.

See Also

Reference

Value Edit Control

Modules Button

Click the **Modules** button to bring up the **Additional Modules for Error Searching** dialog. Enter the name of the desired EXE or DLL in the edit box and click **Add** to include the modules in your error message search. Remove a module from the list by highlighting it and clicking the **Remove** button.

See Also

Reference

Value Edit Control

Look Up Button

Click **Look Up** to retrieve the error message that corresponds to the system or module value entered. Values can be entered in hexadecimal or decimal form (including negative decimal values). Modules listed in the **Additional Modules for Error Searching** dialog are also searched.

See Also

Reference

Value Edit Control

Decorated Names

Functions in C and C++ programs are known internally by their decorated names. A decorated name is a string created by the compiler during compilation of the function definition or prototype.

A decorated name is sometimes required when you specify a function name to LINK or other tools. For details about the situations that require decorated names, consult the documentation for the tool you are using.

Note
The decorated naming convention for pointers to member functions was changed in Visual C++ version 4.0. C++ libraries created with Visual C++ version 2.0 should be recompiled to link properly with source files compiled with the current version of Visual C++.

To create or view decorated names, see:

- Using Decorated Names
- Viewing Decorated Names

See Also
Reference
C/C++ Build Tools

Using Decorated Names

In most circumstances, you do not need to know the decorated name of a function. LINK and other tools can usually handle the name in its undecorated form.

However, certain situations require that you specify the name in its decorated form. You must specify the decorated name of C++ functions that are overloaded and special member functions, such as constructor and destructor functions, in order for LINK and other tools to be able to match the name. You must also use decorated names in assembly source files that reference a C or C++ function name.

Note
If you change the function name, class, calling convention, return type, or any parameter, the decorated name is no longer valid. You must get the new version of the function name and use it everywhere the decorated name is specified.

See Also

Reference

Decorated Names

Format of a C++ Decorated Name

Format of a C Decorated Name

Format of a C++ Decorated Name

A decorated name for a C++ function contains the following information:

- The function name.
- The class that the function is a member of, if it is a member function. This may include the class that encloses the function's class, and so on.
- The namespace the function belongs to (if it is part of a namespace).
- The types of the function's parameters.
- The calling convention.
- The return type of the function.

The function and class names are encoded in the decorated name. The rest of the decorated name is a code that has internal meaning only for the compiler and the linker. The following are examples of undecorated and decorated C++ names.

Undecorated name	Decorated name
`int a(char){int i=3;return i;};`	`?a@@YAHD@Z`
`void __stdcall b::c(float){};`	`?c@b@@AAGXM@Z`

See Also

Reference

Using Decorated Names

Format of a C Decorated Name

The form of decoration for a C function depends on the calling convention used in its declaration, as shown below.

Calling convention	Decoration
__cdecl (the default)	Leading underscore (_)
__stdcall	Leading underscore (_) and a trailing at sign (@) followed by a number representing the number of bytes in the parameter list
__fastcall	Same as **__stdcall**, but prepended by an at sign instead of an underscore

See Also

Reference

Using Decorated Names

Viewing Decorated Names

You can get the decorated form of a function name after you compile the source file that contains the function definition or prototype. To examine decorated names in your program, you can do one of the following:

- Use a listing
- Use the DUMPBIN tool

You can use the undname.exe to convert a decorated name to its undecorated form. For example,

```
C:\>undname ?func1@a@@AAEXH@Z
Microsoft (R) C++ Name Undecorator
Copyright (C) Microsoft Corporation 1981-2000. All rights reserved.Undecoration
of :- "?func1@a@@AAEXH@Z"
is :- "private: void __thiscall a::func1(int)"
```

See Also

Reference

Decorated Names

Using a Listing to View Decorated Names

To get the decorated form of a function using a compiler listing, do the following:

1. Generate a listing by compiling the source file that contains the function definition or prototype with the Listing File Type (/FA[c|s]) compiler option set to one of the following three choices: Assembly with Machine Code (/FAc); Assembly with Source Code (/FAs); Assembly, Machine Code, and Source (/FAcs).
2. Find the line that contains the undecorated function definition in the resulting listing.
3. Examine the previous line. The label for the PROC NEAR command is the decorated form of the function name.

See Also

Reference

Viewing Decorated Names

Using DUMPBIN to View Decorated Names

To get the decorated form of a function using DUMPBIN, run it on the .OBJ or .LIB file using the /SYMBOLS option. Find the undecorated function definition in the output. The undecorated name is followed by the decorated name, each enclosed in parentheses.

See Also

Reference

Viewing Decorated Names

Profile-Guided Optimizations

Profile-guided optimization lets you optimize an output file, where the optimizer uses data from test runs of the .exe or .dll file. The data represents how the program is likely to perform in a production environment.

Profile-guided optimizations are only available for native targets: x86, Itanium Processor Family (IPF), or x64. Profile-guided optimizations are not available for output files that will run on the common language runtime. Even if you produce an assembly with mixed native and managed code (compile with **/clr**), you cannot use profile-guided optimization on just the native code.

Note
Information that is gathered from profiling test runs will override optimizations that would otherwise be in effect if you speci fy **/Ob**, **/Os**, or **/Ot**. For more information, see /Ob (Inline Function Expansion) and /Os, /Ot (Favor Small Code, Favor Fast Code).

The following is an overview of the process of using profile-guided optimizations:

- Compile one or more source code files with /GL.

  Each module built with /GL can be examined during profile-guided optimization test runs to capture run-time behavior. Every module in a profile-guided optimization build does not have to be compiled with /GL. However, only those modules compiled with /GL will be instrumented and later available for profile-guided optimizations.

- Link with /LTCG:PGINSTRUMENT.

  /LTCG:PGINSTRUMENT creates an empty .pgd file. After test-run data is added to the .pgd file, it can be used as input to the next link step (creating the optimized image). When specifying /LTCG:PGINSTRUMENT, you can optionally specify /PGD with a nondefault name or location for the .pgd file.

- Profile the application.

  Each time a profiled EXE session ends or a profiled DLL is unloaded, a *appname*!#.pgc file is created. A .pgc file contains information about a particular application test run. # is a number starting with 1 that is incremented based on the number of other *appname*!#.pgc files in the directory. You can delete a .pgc file if the test run does not represent a scenario you want to optimize.

  During a test run, you can force closure of the currently open .pgc file and the creation of a new .pgc file with the pgosweep utility (for example, when the end of a test scenario does not coincide with application shutdown).

- Link with /LTGC:PGOPTIMIZE.

  /LTGC:PGOPTIMIZE creates the optimized image. This step takes as input the .pgd file. For more information, see /LTCG:PGOPTIMIZE.

It is even possible to create the optimized output file and later determine that additional profiling would be useful to create a more optimized image. If the instrumented image and its .pgd file are available, you can do additional test runs and rebuilt the optimized image with the newer .pgd file.

The following is a list of the profile-guided optimizations:

- **Inlining** – For example, if there exists a function A that frequently calls function B, and function B is relatively small, then profile-guided optimizations will inline function B in function A.
- **Virtual Call Speculation** – If a virtual call, or other call through a function pointer, frequently targets a certain function, a profile-guided optimization can insert a conditionally-executed direct call to the frequently-targeted function, and the direct call can be inlined.
- **Register Allocation** – Optimizing with profile data results in better register allocation.
- **Basic Block Optimization** – Basic block optimization allows commonly executed basic blocks that temporally execute within a given frame to be placed in the same set of pages (locality). This minimizes the number of pages used, thus minimizing memory overhead.
- **Size/Speed Optimization** – Functions where the program spends a lot of time can be optimized for speed.

- **Function Layout** – Based on the call graph and profiled caller/callee behavior, functions that tend to be along the same execution path are placed in the same section.
- **Conditional Branch Optimization** – With the value probes, profile-guided optimizations can find if a given value in a switch statement is used more often than other values. This value can then be pulled out of the switch statement. The same can be done with if/else instructions where the optimizer can order the if/else so that either the if or else block is placed first depending on which block is more frequently true.
- **Dead Code Separation** – Code that is not called during profiling is moved to a special section that is appended to the end of the set of sections. This effectively keeps this section out of the often-used pages.
- **EH Code Separation** – The EH code, being exceptionally executed, can often be moved to a separate section when profile-guided optimizations can determine that the exceptions occur only on exceptional conditions.
- **Memory Intrinsics** – The expansion of intrinsics can be decided better if it can be determined if an intrinsic is called frequently. An intrinsic can also be optimized based on the block size of moves or copies.

For more information, see Walkthrough: Using Profile-Guided Optimizations.

In This Section

Tools for Profile-Guided Optimization

How to: Merge Multiple PGO Profiles into a Single Profile

See Also

Reference

C/C++ Build Tools

Tools for Profile-Guided Optimization

The following tools support profile-guided optimizations:

Environment Variables for Profile-Guided Optimizations

Specify run-time behavior of testing scenarios.

pgosweep

Writes all profile data to the .pgc file, closes the .pgc file, and opens a new .pgc file.

pgomgr

Adds profile data from one or more .pgc files to the .pgd file.

See Also

Reference

Profile-Guided Optimizations

Environment Variables for Profile-Guided Optimizations

This section contains information about environment variables that affect test scenarios on an image created with /LTCG:PGI.

Remarks

There are two environment variables that affect test scenarios for profile-guided optimizations:

- VCPROFILE_ALLOC_SCALE
- VCPROFILE_PATH

See Also

Reference

Tools for Profile-Guided Optimization

VCPROFILE_ALLOC_SCALE

Modify the amount of memory allocated to hold the profile data.

```
VCPROFILE_ALLOC_SCALE=scale_value
```

Parameters

scale_value

The scale value for the amount of memory you want for running test scenarios. The default is 1.

Remarks

In rare cases, there will not be enough memory available to support gathering profile data when running test scenarios. In those cases, you can increase the amount of memory with **VCPROFILE_ALLOC_SCALE**.

If you receive an error message during a test run that indicates that you have insufficient memory, assign a larger value to **VCPROFILE_ALLOC_SCALE**, until the test runs complete with no out-of-memory errors.

Example

```
set VCPROFILE_ALLOC_SCALE=2
```

See Also

Reference

Environment Variables for Profile-Guided Optimizations

VCPROFILE_PATH

Specify the directory to create .pgc files.

```
VCPROFILE_PATH=path
```

Parameters

path

The directory path in which .pgc files will be added.

Remarks

By default, .pgc files are created in the same directory as the binary being profiled. However, if the absolute path of the binary does not exist, as may be the case when you run profile scenarios on a different machine from where the binary was built, you can set **VCPROFILE_PATH** to a path that exists.

Example

```
set VCPROFILE_PATH=c:\
```

See Also

Reference

Environment Variables for Profile-Guided Optimizations

pgomgr

Adds profile data from one or more .pgc files to the .pgd file.

```
pgomgr options pgcfiles pgdfile
```

where:

pgcfiles

One or more .pgc files whose profile data you want to merge into the .pgd file. You can specify a single .pgc file or multiple .pgc files. If you do not specify any .pgc files, pgomgr will merge all .pgc files whose filenames are the same as the .pgd file.

pgdfile

The .pgd file into which you are merging data from the .pgc file or files.

options(optional)

The following options can be specified to pgomgr:

/help—Displays available pgomgr options (short for /?).

/clear—Causes the .pgd file to be cleared of all profile information. You cannot specify a .pgc file when **/clear** is specified.

/detail—Displays detailed statistics, including flow graph coverage information.

/summary—Displays per-function statistics.

/unique—when used with **/summary**, causes decorated function names to display. The default, when /unique is not used, is for undecorated function names to be displayed.

/merge[**:***n*]**—**Causes the data in the .pgc file or files to be added to the .pgd file. The optional parameter, *n*, lets you specify hat the data should be added *n* times. For example, if a scenario would commonly be done 6 times, you can do it once in a test run and add it to the .pgd file six times with **pgomgr /merge:6**.

Example

In the following example, the .pgd file was cleared of profile data.

```
pgomgr /clear myapp.pgd
```

In the following example, the profile data in myapp1.pgc was added to the .pgd file 3 times.

```
pgomgr /merge:3 myapp1.pgc myapp.pgd
```

In the following example, profile data from all myapp#.pgc files is added to the myapp.pgd file.

```
pgomgr -merge myapp1.pgd
```

See Also

Reference

Tools for Profile-Guided Optimization

pgosweep

Writes all profile data to the .pgc file, closes the .pgc file, and resets profile data in memory to zero.

```
pgsoweep options image pgcfile
```

where:

image

The .exe or .dll file that was created with /LTCG:PGINSTRUMENT.

pgcfile

The.pgc file where the data counts for this run should be written.

options (optional)

In addition to /? or /help, you can also specify /noreset. /noreset preserves profiling information in memory. By default, profiling information in memory is cleared. If /noreset is specified, subsequent .pgc files will contain profile information that was added to .pgc files that were created using /noreset.

Remarks

When you name your .pgc file (*pgcfile*) you can use the standard format, which is *appname***!***n*.pgc. If you use this format, this data will be located in the /LTCG:PGO phase. If you do not use the standard format, you must then use pgomgr to merge the .pgc files.

Example

```
pgosweep myapp.exe myapp1.pgc
```

In this example, the current profile information for myapp.exe will all be written to myapp1.pgc.

See Also

Reference

Tools for Profile-Guided Optimization

How to: Merge Multiple PGO Profiles into a Single Profile

Profile-guided optimization (PGO) is a great tool for creating optimized binaries based on a scenario that is profiled. But what if you have an application that has several important, yet distinct scenarios; how do you create a single profile that PGO can use from several different scenarios? In Visual Studio 2005, the PGO Manager, Pgomgr.exe, does this job for you.

The syntax for merging profiles is:

```
pgomgr /merge[:num] [.pgc_files] .pgd_files
```

where *num* is an optional weight that is being used for this merge. Weights are commonly used if there are some scenarios that are more important than others or if there are scenarios that are to be run multiple times.

Note
The PGO Manager will not work with stale profile data. To merge a .pgc file into a .pgd file, the .pgc file must be generated by an executable which was created by the same link invocation that generated the .pgd file.

Example

In this example, the PGO Manager will add pgcFile.pgc to pgdFile.pgd six times.

```
pgomgr /merge:6 pgcFile.pgc pgdFile.pgd
```

In this example, the PGO Manager will add pgcFile1.pgc and pgcFile2.pgc to pgdFile.pgd, two times for each .pgc file.

```
pgomgr /merge:2 pgcFile1.pgc pgcFile2.pgc pgdFile.pgd
```

If the PGO Manager is run without a .pgc file it will search the local directory for all .pgc files that have the same name as the .pgd file appended with an exclamation mark (!) followed by arbitrary characters. If the local directory has files test.pgd, test!1.pgc, test2.pgc, and test!hello.pgc, and the following command is run from the local directory, then test!1.pgc and test!hello.pgc will be merged into test.pgd.

```
pgomgr /merge test.pgd
```

See Also

Reference

Profile-Guided Optimizations

C/C++ Build Errors

This section is an alphanumeric reference of the error messages generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, or type the error number in the **Look for** box in the Index.

See Also

Other Resources

C/C++ Building Reference

Debugging in Visual Studio

BSCMAKE Errors BK1500 through BK4503

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, or type the error number in the **Look for** box in the Index.

BSCMAKE Error BK1500

Error Message
UNKNOWN ERROR

BSCMAKE detected an unidentified error condition.

Note the circumstances of the error and try to isolate the problem and create a reproducible test case before Contacting Microsoft Product Support Services.

BSCMAKE Error BK1501

Error Message
unknown character 'character' in option 'option'

BSCMAKE did not recognize the character specified for the option.

BSCMAKE Error BK1502

Error Message
incomplete specification for option 'option'

The option did not contain the correct syntax.

BSCMAKE Error BK1503

Error Message
cannot write to file 'filename' [: reason]

BSCMAKE combines the .sbr files generated during compilation into one browser database. If the resulting browser database exceeds 64 MB, or if the number of input (.sbr) files exceeds 4092, this error will be emitted.

If the problem is caused by more than 4092 .sbr files, you must reduce the number of input files. From within Visual Studio, this can be accomplished by /FR your entire project, then recheck on a file by file basis.

If the problem is caused by a .bsc file larger than 64MB, reducing the number of .sbr files as input will decrease the size of the resulting .bsc file. In addition, the amount of browse information may be reduced through the use of the /Em (Exclude Macro Expanded Symbols), /El (Exclude Local Variables), and /Es (Exclude System Files).

See Also

Reference
BSCMAKE Options

BSCMAKE Error BK1504

Error Message
cannot position in file 'filename' [: reason]

BSCMAKE could not move to a location in the file.

To fix by checking the following possible causes

1. A full disk.
2. A hardware error.
3. File truncation due to insufficient disk space or interruption of the compiler while creating the .sbr file.

BSCMAKE Error BK1505

Error Message
cannot read from file 'filename' [: reason]

BSCMAKE cannot read from the file.

To fix by checking the following possible causes

1. File corruption.
2. File truncation to the compiler running out of disk space or being interrupted while creating the .sbr file.

BSCMAKE Error BK1506

Error Message
cannot open file 'filename' [: reason]

BSCMAKE cannot open the file.

To fix by checking the following possible causes

1. File locked by another process. If *reason* says **Permission denied**, the browser may be using the file. Close the Browse window and retry the build.
2. A full disk.
3. A hardware error.
4. The specified output file has the same name as an existing subdirectory.

BSCMAKE Error BK1507

Error Message
cannot open temporary file 'filename' [: reason]

BSCMAKE cannot open a temporary file.

To fix by checking the following possible causes

1. The **TMP** environment variable is not set to a valid drive and directory.
2. A full disk.

BSCMAKE Error BK1508

Error Message
cannot delete temporary file 'filename' [: reason]

BSCMAKE cannot delete one of its temporary files.

To fix by checking the following possible causes

1. Another process has the file open.
2. A hardware error.

BSCMAKE Error BK1509

Error Message
out of heap space

BSCMAKE ran out of memory, including virtual memory.

To fix by using the following possible solutions

1. Free some disk space.
2. Increase the size of the swap file.
3. Increase the size of the Windows swap file.
4. Reduce the memory BSCMAKE requires by using /Ei or /Es to eliminate some input files or /Em to eliminate macro bodies.

BSCMAKE Error BK1510

Error Message
corrupt .SBR file filename

The given .sbr file does not have the expected format. Recompile to create a new .sbr.

BSCMAKE Error BK1511

Error Message
invalid response file specification

The command-line specification for the response file was incorrect or incomplete, for example:

```
bscmake @
```

BSCMAKE Error BK1512

Error Message
filename: capacity exceeded

BSCMAKE cannot build a browse information file because the number of definitions, references, modules, or other information exceeds the limit.

To fix by using the following possible solutions

1. Exclude some information using /Em, /Es, or /Ei.
2. Omit the /Iu option.

BSCMAKE Error BK1513

Error Message
nonincremental update requires all .SBR files

BSCMAKE cannot build a new browse information (.bsc) file because one or more .sbr files are truncated. To find the names of the truncated .sbr files, read the BK4502 warnings that accompany this error.

BSCMAKE can update a .bsc file with a truncated .sbr file but cannot build a new one. BSCMAKE may build a new .bsc file for the following reasons:

- Missing .bsc file.
- Wrong filename specified for .bsc file.
- Corrupted .bsc file.

BSCMAKE Error BK1514

Error Message
all .SBR files truncated, none found in filename

None of the .sbr files specified for an update were part of the original browse information (.bsc) file. To find the names of the .sbr files that caused this error, read the BK4502 warnings that precede it.

To fix by checking the following possible causes

1. Wrong filename specified for .sbr or .bsc.
2. Corrupted .bsc file required BSCMAKE to rebuild it.

BSCMAKE Error BK1515

Error Message
bscfile: incompatible version: cannot incrementally update

The .bsc file was created with another version of BSCMAKE. Recreate the .bsc file.

BSCMAKE Error BK1516

Error Message
bscfile corrupt; cannot incrementally update

The .bsc file was corrupted, possibly due to a system failure during the build. Delete the .bsc file, rebuild all .sbr files, then rebuild the .bsc file.

BSCMAKE Error BK1517

Error Message
source file for sbrfile compiled with both /Yc and /Yu

The .sbr file refers to itself. It was probably recompiled with /Yu after compiling with /Yc. Reset the compiler option for the source file to /Yc, then select **Rebuild** to generate new .sbr files. Do not recompile the source file with /Yu.

BSCMAKE Warning BK4500

Error Message
UNKNOWN WARNING

BSCMAKE detected an unidentified warning condition.

Note the circumstances of the warning and try to isolate the problem and create a reproducible test case before Contacting Microsoft Product Support Services.

BSCMAKE Warning BK4501

Error Message
ignoring unknown option 'option'

BSCMAKE did not recognize the option.

BSCMAKE Warning BK4502

Error Message
truncated .SBR file 'filename' not in filename

A zero-length .sbr file that was not originally part of the .bsc file was specified during an update.

To fix by checking the following possible causes

1. Wrong filename specified.
2. File deleted. (Error BK1513 results.)
3. File corrupted, requiring BSCMAKE to do a full build.

BSCMAKE Warning BK4503

Error Message
minor error in .SBR file filename ignored

The .sbr file contained an error that did not halt the build, but the resulting .bsc file may be incorrect. Recompile to create a new .sbr.

For more information, see Knowledge Base article Q179270.

BSCMAKE Warning BK4504

Error Message
file contains too many references; ignoring further references from this source

There are more than 64,000 references to a single symbol in the .cpp file.

To correct this problem, you should split your .cpp file into two or more .cpp files.

BSCMAKE Warning BK4505

Error Message
must read and agree to the Data Collection Policy at "url" before using /errorreport:send

Before you can use bscmake's **/errorReport:send** to send information on an internal error you must agree to Microsoft's policy on data collection.

The text of the BK4505 diagnostic will refer you to a website where you can agree to Microsoft's policy on data collection.

For more information on error reporting, see /errorReport (Report Internal Compiler Errors).

Compiler Fatal Errors C999 through C1999

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, type the error number in the **Look for** box in the Index, or click a link below.

Fatal Error C999

Fatal Error C1001

Fatal Error C1002

Fatal Error C1003

Fatal Error C1004

Fatal Error C1005

Fatal Error C1007

Fatal Error C1008

Fatal Error C1009

Fatal Error C1010

Fatal Error C1012

Fatal Error C1013

Fatal Error C1014

Fatal Error C1016

Fatal Error C1017

Fatal Error C1018

Fatal Error C1019

Fatal Error C1020

Fatal Error C1021

Fatal Error C1022

Fatal Error C1023

Fatal Error C1026

Fatal Error C1033

Fatal Error C1034

Fatal Error C1035

Fatal Error C1037

Fatal Error C1038

Fatal Error C1045

Fatal Error C1046

Fatal Error C1047

Fatal Error C1051

Fatal Error C1053

Fatal Error C1054

Fatal Error C1055

Fatal Error C1057

Fatal Error C1059

Fatal Error C1060

Fatal Error C1061

Fatal Error C1064

Fatal Error C1065

Fatal Error C1067

Fatal Error C1068

Fatal Error C1070

Fatal Error C1071

Fatal Error C1073

Fatal Error C1074

Fatal Error C1075

Fatal Error C1076

Fatal Error C1077

Fatal Error C1079

Fatal Error C1080

Fatal Error C1081

Fatal Error C1082

Fatal Error C1083

Fatal Error C1084

Fatal Error C1085

Fatal Error C1086

Fatal Error C1087

Fatal Error C1088

Fatal Error C1089

Fatal Error C1091

Fatal Error C1092

Fatal Error C1093

Fatal Error C1094

Fatal Error C1098

Fatal Error C1099

Fatal Error C1100

Fatal Error C1103

Fatal Error C1104

Fatal Error C1107

Fatal Error C1108

Fatal Error C1109

Fatal Error C1113

Fatal Error C1120

Fatal Error C1121

Fatal Error C1126

Fatal Error C1128

Fatal Error C1189

Fatal Error C1190

Fatal Error C1191

Fatal Error C1192

Fatal Error C1196

Fatal Error C1197

Fatal Error C1201

Fatal Error C1202

Fatal Error C1205

Fatal Error C1206

Fatal Error C1207

Fatal Error C1208

Fatal Error C1209

Fatal Error C1210

Fatal Error C1211

Fatal Error C1305

Fatal Error C1307

Fatal Error C1308

Fatal Error C1309

Fatal Error C1310

Fatal Error C1311

Fatal Error C1312

Fatal Error C1313

Fatal Error C1350

Fatal Error C1351

Fatal Error C1352

Fatal Error C1382

Fatal Error C1383

Fatal Error C1505

Fatal Error C1506

Fatal Error C1507

Fatal Error C1508

Fatal Error C1509

Fatal Error C1600

Fatal Error C1601

Fatal Error C1602

Fatal Error C1603

Fatal Error C1852

Fatal Error C1853

Fatal Error C1854

Fatal Error C1900

Fatal Error C1902

Fatal Error C1903

Fatal Error C1904

Fatal Error C1905

Fatal Error C1999

Fatal Error C999

Error Message

UNKNOWN MESSAGE Please choose the Technical Support command on the Visual C++ Help menu, or open the Technical Support help file for more information

This error usually means that you have error files (.err) from a different version of the compiler. Try reinstalling the product.

Fatal Error C1001

Error Message
INTERNAL COMPILER ERROR(compiler file file, line number)

The compiler cannot generate correct code for a construct, probably due to the combination of an expression and an optimization option. Try removing one or more optimization options and recompiling the function containing the line indicated in the error message.

You can probably fix the problem by removing one or more optimization options. To determine which option is at fault, remove options one at a time and recompile until the error message goes away. The options most commonly responsible are **/Og**, **/Oi**, and **/Oa**. Once you determine which option is responsible, you can disable it using the optimize pragma around the function where the error occurs and continue to use the option for the rest of the module.

The Microsoft Knowledge Base has more information about C1001; see http://support.microsoft.com/default.aspx?scid=kb;en-us;134650.

Try rewriting the line where the error is reported, or several lines of code surrounding that line. If that doesn't work, contact Microsoft Product Support Services.

Fatal Error C1002

Error Message
compiler is out of heap space in pass 2

The compiler ran out of dynamic memory space during its second pass, probably due to a program with too many symbols or complex expressions.

To fix by using the following possible solutions

1. Divide the source file into several smaller files.
2. Break expressions into smaller subexpressions.
3. Remove other programs or drivers that consume memory.

Fatal Error C1003

Error Message

error count exceeds number; stopping compilation

Errors in the program are too numerous to allow recovery. The compiler must terminate.

Fatal Error C1004

Error Message
unexpected end of file found

The compiler reached the end of a source file without resolving a construct. The code may be missing one of the following elements:

- A closing brace
- A closing parenthesis
- A closing comment marker (*/)
- A semicolon

To resolve this error, check for the following:

- The default disk drive has insufficient space for temporary files, which require about twice as much space as the source file.
- An **#if** directive that evaluates to false lacks a closing **#endif** directive.
- A source file does not end with a carriage return and line feed.

The following sample generates C1004:

```
// C1004.cpp
#if TEST
int main() {}
// C1004
```

Possible resolution:

```
// C1004b.cpp
#if TEST
#endif

int main() {}
```

Fatal Error C1005

Error Message
string too big for buffer

A string in a compiler intermediate file overflowed a buffer.

You could get this error when the parameter that you pass to either the /Fd or /Yl compiler options is greater than 256 bytes.

Fatal Error C1007

Error Message
unrecognized flag string in option

The command-line option contains an invalid string. Check the **CL** command line and environment variable for errors.

Fatal Error C1008

Error Message
no input file specified

The compiler was not given a C or C++ source file to compile. Check the **CL** command line and environment variable for filename specifications.

Fatal Error C1009

Error Message
compiler limit : macros nested too deeply

The compiler tried to expand too many macros at the same time. The compiler has a limit of 256 levels of nested macros. Split nested macros into simpler macros.

Fatal Error C1010

Error Message
unexpected end of file while looking for precompiled header. Did you forget to add '#include name' to your source?

An include file specified with /Yu is not listed in the source file. This option is enabled by default in most Visual C++ Project types and "stdafx.h" is the default include file specified by this option.

In the Visual Studio environment, use one of the following methods to resolve this error:

- If you do not use precompiled headers in your project, set the **Create/Use Precompiled Header** property of source files to **Not Using Precompiled Headers**. To set this compiler option, follow these steps:
  1. In the Solution Explorer pane of the project, right-click the project name, and then click **Properties**.
  2. In the left pane, click the **C/C++** folder.
  3. Click the **Precompiled Headers** node.
  4. In the right pane, click **Create/Use Precompiled Header**, and then click **Not Using Precompiled Headers**.
- Make sure you have not inadvertently deleted, renamed or removed header file (by default, stdafx.h) from the current project. This file also needs to be included before any other code in your source files using **#include "stdafx.h"**. (This header file is specified as **Create/Use PCH Through File** project property)

Fatal Error C1012

Error Message
unmatched parenthesis : missing character

The parentheses in a preprocessor directive do not match.

Fatal Error C1013

Error Message
compiler limit : too many open parentheses

An expression contains too many levels of parentheses in a single expression. Simplify the expression or break it into multiple statements.

Prior to Visual C++ 6.0 Service Pack 3, the limit on nested parenthesis in a single expression was 59. Currently, the limit on nested parenthesis is 256.

Fatal Error C1014

Error Message
too many include files : depth = level

The nesting of **#include** directives is too deep. Nested directives can include open files. The source file containing the directive counts as one file.

Fatal Error C1016

Error Message
#ifdef expected an identifier#ifndef expected an identifier

The conditional compilation directive (#ifdef or **#ifndef**) has no identifier to evaluate. To resolve the error, specify an identifier.

The following sample generates C1016:

```
// C1016.cpp
#ifdef   // C1016
#define FC1016
#endif

int main() {}
```

Possible resolution:

```
// C1016b.cpp
#ifdef X
#define FC1016
#endif

int main() {}
```

Fatal Error C1017

Error Message
invalid integer constant expression

The expression in an **#if** directive did not exist or did not evaluate to a constant.

Constants defined using **#define** must have values that evaluate to an integer constant if they are used in an **#if**, **#elif**, or **#else** directive.

The following sample generates C1017:

```
// C1017.cpp
#define CONSTANT_NAME "YES"
#if CONSTANT_NAME   // C1017
#endif
```

Possible resolution:

```
// C1017b.cpp
// compile with: /c
#define CONSTANT_NAME 1
#if CONSTANT_NAME
#endif
```

Because `CONSTANT_NAME` evaluates to a string and not an integer, the **#if** directive generates fatal error C1017.

In other cases, the preprocessor evaluates an undefined constant as zero. This can cause unintended results, as shown in the following sample. `YES` is undefined, so it evaluates to zero. The expression **#if** `CONSTANT_NAME` evaluates to false and the code to be used on `YES` is removed by the preprocessor. `NO` is also undefined (zero), so **#elif** `CONSTANT_NAME==NO` evaluates to true (`0 == 0`), causing the preprocessor to leave the code in the **#elif** portion of the statement — exactly the opposite of the intended behavior.

```
// C1017c.cpp
// compile with: /c
#define CONSTANT_NAME YES
#if CONSTANT_NAME
   // Code to use on YES...
#elif CONSTANT_NAME==NO
   // Code to use on NO...
#endif
```

To see exactly how the compiler handles preprocessor directives, use /P, /E, or /EP.

Fatal Error C1018

Error Message
unexpected #elif

The **#elif** directive appears outside an **#if**, **#ifdef**, or **#ifndef** construct. Use **#elif** only within one of these constructs.

The following sample generates C1018:

```
// C1018.cpp
#elif      // C1018
#endif

int main() {}
```

Possible resolution:

```
// C1018b.cpp
#if 1
#elif
#endif

int main() {}
```

Fatal Error C1019

Error Message
unexpected #else

The **#else** directive appears outside an **#if**, **#ifdef**, or **#ifndef** construct. Use **#else** only within one of these constructs.

The following sample generates C1019:

```
// C1019.cpp
#else   // C1019
#endif

int main() {}
```

Possible resolution:

```
// C1019b.cpp
#if 1
#else
#endif

int main() {}
```

Fatal Error C1020

Error Message
unexpected #endif

The **#endif** directive has no matching **#if**, **#ifdef**, or **#ifndef** directive. Be sure each **#endif** has a matching directive.

The following sample generates C1020:

```
// C1020.cpp
#endif     // C1020
```

Possible resolution:

```
// C1020b.cpp
// compile with: /c
#if 1
#endif
```

Fatal Error C1021

Error Message
invalid preprocessor command 'string'

string is not a valid preprocessor directive. To resolve the error, use a valid preprocessor name for *string*.

The following sample generates C1021:

```
// C1021.cpp
#BadPreProcName    // C1021 delete line
```

Fatal Error C1022

Error Message
expected #endif

An **#if**, **#ifdef**, or **#ifndef** directive has no matching **#endif** directive. Be sure each **#if**, **#ifdef**, or **#ifndef** has a matching **#endif**.

The following sample generates C1022:

```
// C1022.cpp
#define true 1

#if (true)
#else
#else    // C1022
```

Possible resolution:

```
// C1022b.cpp
// compile with: /c
#define true 1

#if (true)
#else
#endif
```

Fatal Error C1023

Error Message

'file' : unexpected error with pch, try rebuilding the pch

C1023 could be caused by one of several problems, the solution to which is a rebuild of the precompiled header file.

C1023 may be caused if you attempt to use a precompiled header file on a machine other than the one on which it was built.

Fatal Error C1026

Error Message
parser stack overflow, program too complex

The space required to parse the program caused a compiler stack overflow.

Decrease the complexity of expressions by:

- Decreasing nesting in **for** and **switch** statements. Put more deeply nested statements in separate functions.
- Breaking up long expressions that involve comma operators or function calls.

Fatal Error C1033

Error Message
cannot open program database pdb

This error can be caused by disk error.

In Visual C++ .NET 2002, the user locale must be set correctly when the file name (or directory path to the file name) contains MBCS characters. Setting the system locale is not sufficient; the user locale must be set to process MBCS characters.

For more information, see http://support.microsoft.com/default.aspx?scid=kb;en-us;246007.

Fatal Error C1034

Error Message
file: no include path set

The **INCLUDE** environment variable is not set.

To set the INCLUDE variable in Visual C++

1. From the **Tools** Menu, click **Options**.
2. In the **Options** dialog box, select the **Directories** tab.
3. In the **Show Directories For** box, select **Include Files**.
4. In the **Directories** box, enter the path for include files.

To set the INCLUDE variable from the command line

- Run the VCVARS32.bat batch file.

Fatal Error C1035

Error Message
expression too complex; simplify expression

The compiler could not generate code for a complex expression. Split the expression into simpler expressions and recompile.

Fatal Error C1037

Error Message
cannot open object file filename

The object file specified by /Fo cannot be opened.

To fix by checking the following possible causes

1. Invalid filename.
2. Insufficient memory to open the file.
3. Another process is using the file.
4. A read-only file has the same name.

In Visual C++ .NET (version 1300 of the compiler), there is a bug that requires the user locale to be set correctly when the file name (or directory path to the file name) contains MBCS characters. Setting the system locale is not sufficient; the user locale must be set to process MBCS characters.

Fatal Error C1038

Error Message

compiler limit : function : control flow state too complex; simplify function

The function has more control-flow states than the compiler can handle. Simplify control flow or split the function into smaller functions.

Fatal Error C1045

Error Message

compiler limit : linkage specifications nested too deeply

Nested externals exceed the compiler limit. Nested externals are allowed with the external linkage type, such as **extern** "C++". Reduce the number of nested externals to resolve the error.

Fatal Error C1046

Error Message
compiler limit : structure nested too deeply

The structure, union, or class exceeded the nesting limit, which is 15 levels. Rewrite the definition to reduce the nesting level. Split the structure, union, or class into two or more parts by using **typedef** to define one or more of the nested structures.

Fatal Error C1047

Error Message
The object or library file 'file' was created with an older compiler than other objects; rebuild old objects and libraries

C1047 is caused when object files or libraries built with **/LTCG** are linked together, but where those object files or libraries are built with different versions of the Visual C++ toolset.

This can happen if you begin using a new version of the compiler but do not do a clean rebuild of existing object files or libraries.

To resolve C1047, rebuild all object files or libraries.

Fatal Error C1051

Error Message

program database file, 'pdbfile', has an obsolete format, delete it and recompile

The compiler cannot update the program database file, which has an older version number. Delete the file and recompile your program with **/Zi** or **/ZI**. For more information, see /Z7, /Zi, /ZI (Debug Information Format)

Fatal Error C1053

Error Message
'<identifier>' : function too large

The function is too large to compile.

To fix by using the following possible solutions

1. Try compiling without optimizations.
2. Divide the function into smaller functions.
3. Reduce calls to inline functions.

Fatal Error C1054

Error Message
compiler limit : initializers nested too deeply

The code exceeds the nesting limit on initializers (10-15 levels, depending on the combination of types being initialized).

To fix by using the following possible solutions

1. Simplify the data types being initialized to reduce nesting.
2. Initialize variables in separate statements after the declaration.

Fatal Error C1055

Error Message
compiler limit : out of keys

The source file contains too many symbols. The compiler ran out of hash keys for the symbol table.

To fix by using the following possible solutions

1. Split the source file into smaller files.
2. Eliminate unnecessary header files.
3. Reuse temporary and global variables instead of creating new ones.

Fatal Error C1057

Error Message
unexpected end of file in macro expansion

The compiler reached the end of the source file while gathering macro-invocation arguments, probably due to a missing right parenthesis in the macro invocation.

Fatal Error C1059

Error Message
compiler is out of near heap space

The compiler ran out of space in its near default data-segment (near) heap.

To fix by using the following possible solutions

1. Eliminate unnecessary include files.
2. Split the current function into smaller functions.
3. Split the current file into smaller files.

Fatal Error C1060

Error Message
compiler is out of heap space

The operating system or run-time library cannot fill a request for memory.

To fix by using the following possible solutions

1. Use /Zm to increase memory allocation limit.
2. Increase the size of the Windows swap-file.
3. Close other running programs.
4. Eliminate unnecessary include files.
5. Eliminate unnecessary global variables, for example, by allocating memory dynamically instead of declaring a large array.
6. Eliminate unused declarations.
7. Split the current file into smaller files.

Fatal Error C1061

Error Message
compiler limit : blocks nested too deeply

Nesting of code blocks exceeds the limit of 128 nesting levels. Simplify nesting.

Fatal Error C1064

Error Message

compiler limit : token overflowed internal buffer

An identifier exceeds the length of the internal buffer used for identifiers. Shorten the name.

Fatal Error C1065

Error Message
compiler limit : out of tags

The source file contains more than 65,523 classes, structs, unions, namespaces, or enums.

Split the file into smaller files.

Fatal Error C1067

Error Message
compiler limit : 64K limit on size of a type record has been exceeded

This error could occur if a symbol has a decorated name exceeding 247 characters. To resolve, shorten the symbol name.

When the compiler generates debug information, it emits type records to define types encountered in source code. For example, type records include simple structures and argument lists of functions. Some of these type records can be large lists.

There is a 64K limit on the size of any type record. If that 64K limit is exceeded then this error will occur.

C1067 can also occur if there are many symbols with long names or if a class, struct, or union has too many members.

Fatal Error C1068

Error Message
cannot open file 'file'

Ensure that *file* is not in use by another program.

Fatal Error C1070

Error Message
mismatched #if/#endif pair in file 'filename'

An **#if**, **#ifdef**, or **#ifndef** directive has no corresponding **#endif**.

The following sample generates C1070:

```
// C1070.cpp
#define TEST

#ifdef TEST

#ifdef TEST
#endif
// C1070
```

Possible resolution:

```
// C1070b.cpp
// compile with: /c
#define TEST

#ifdef TEST
#endif

#ifdef TEST
#endif
```

Fatal Error C1071

Error Message
unexpected end of file found in comment

The compiler reached the end of the file while scanning a comment.

To fix by checking the following possible causes

1. Missing comment terminator (*/).
2. Missing newline character after a comment on the last line of a source file.

The following sample generates C1071:

```
// C1071.cpp
int main() {
}

/* this comment is fine */
/* forgot the closing tag        // C1071
```

Fatal Error C1073

Error Message

Internal error involving incremental compilation (compiler file 'filename', line number)

Recompile the file without using incremental compilation.

Fatal Error C1074

Error Message

'IDB' is illegal extension for PDB file: filename

The compiler expects program databases to have the .pdb extension.

Fatal Error C1075

Error Message
end of file found before the left token at 'filename(linenumber)' was matched

The compiler expected matching *token* before it reached the end of file.

This error can be caused by an unmatched bracket, brace, or other paired character.

Fatal Error C1076

Error Message
compiler limit : internal heap limit reached; use /Zm to specify a higher limit

This error can be caused by too many symbols, or too many template instantiations.

To resolve this error:

1. Use the /Zm option to set the compiler memory limit.
2. Eliminate unnecessary include files.
3. Eliminate unnecessary global variables, for example, by allocating memory dynamically instead of declaring a large array.
4. Eliminate unused declarations.
5. Split large functions into smaller functions.
6. Split large classes into smaller classes.
7. Split the current file into smaller files.

If C1076 occurs immediately after the build starts, the value specified for **/Zm** is probably too high for your program. Reduce the **/Zm** value.

Fatal Error C1077

Error Message

compiler limit : cannot have more than number command line options

The number of command-line options exceeds the internal limit.

There may be too many symbols defined with /D. (Place the definitions in a header file instead.)

Fatal Error C1079

Error Message
compiler limit : PCH file size limit exceeded

The PCH file exceeds the 4 GB size limit.

Fatal Error C1080

Error Message

compiler limit : command line option exceeded limit of number characters

An argument passed to the compiler exceeds 256 characters.

Fatal Error C1081

Error Message
'symbol': file name too long

The length of a file pathname exceeds **_MAX_PATH** (defined by STDLIB.h as 260 characters). Shorten the name of the file.

If you call CL.exe with a short filename, the compiler may need to generate a full pathname. For example, `cl -c myfile.cpp` may cause the compiler to generate:

```
D:\<very-long-directory-path>\myfile.cpp
```

Fatal Error C1082

Error Message

Cannot close filetype file: 'file': message

If the message says "bad file number", the file may have been closing in the foreground while compiling in the background.

Fatal Error C1083

Error Message
Cannot open filetype file: 'file': message

To fix by checking the following possible causes

1. File does not exist.
2. File, subdirectory, or disk is read-only.
3. No access permission for file or directory.
4. Not enough file handles. Close some applications and recompile.
5. The **INCLUDE** environment variable is set incorrectly.
6. An #include directive uses double quotation marks around a path specification, which causes the standard directories to be skipped.
7. You did not specify /clr and your program uses CLR constructs.
8. You attempted to compile a single file in the project without first compiling stdafx.cpp. Before you can compile a single file in the project, you need to compile stdafx.cpp. In the case of the /analyze (Enterprise Code Analysis) compiler option, you will need to use the same **/analyze** setting for stdafx.cpp that you use for the .cpp file.

The following sample generates C1083:

```
// C1083.cpp
// compile with: /c
#include "test.h"   // C1083 test.h does not exist
#include "stdio.h"   // OK
```

Fatal Error C1084

Error Message
Cannot read filetype file: 'file': message

This error is generally the result of a failed internal system API call made by the compiler. The message shown when this error is encountered is often generated by either _wcserror_s or FormatMessage.

Performing the following steps may help resolve C1084:

- Ensure the specified file exists.
- Ensure the appropriate permissions are set in order to access the specified file.
- Ensure the command-line syntax adheres to the rules outlined under Compiler Command-Line Syntax.
- Ensure sure the environment variables **TMP** and **TEMP** are properly set, as well as the appropriate permissions in order to access the directories these environment variables refer to. Also ensure that the drives referenced by the **TMP** and **TEMP** environment variables contain an adequate amount of free space.
- If the message says "bad file number", the specified file may have been closing in the foreground while compiling in the background.

After performing the above diagnostics, perform a clean build.

Fatal Error C1085

Error Message
Cannot write filetype file: 'file': message

To fix by checking the following possible causes

1. Drive is read-only.
2. Drive is full.
3. Sharing violation.
4. If the message says "bad file number", the file may have been closing in the foreground while compiling in the background.

Fatal Error C1086

Error Message

Cannot seek filetype file: 'file': message

The compiler cannot complete an I/O operation.

Fatal Error C1087

Error Message

Cannot tell filetype file: 'file': message

The compiler cannot complete an I/O operation.

Fatal Error C1088

Error Message

Cannot flush filetype file: 'file': message

The compiler cannot complete an I/O operation.

Fatal Error C1089

Error Message

Cannot truncate filetype file: 'file': message

The compiler cannot shrink a file to zero length.

Fatal Error C1091

Error Message
compiler limit: string exceeds 'length' bytes in length

A string constant exceeded the current limit on the length of strings.

You might want to split the static string into two (or more) variables and use strcpy_s to join the result as part of the declaration or during run time.

Fatal Error C1092

Error Message
Edit and Continue does not support changes to data types; build required

You changed or added a data type since the last successful build.

- Edit and Continue does not support changes to existing data types, including class, struct, or enum definitions. You must stop debugging and build the application.
- Edit and Continue does not support the addition of new data types if a program database file, such as vc*x*0.pdb (where *x* is the major version of Visual C++ in use) is read-only. To add data types, the compiler must open the .pdb file in write mode.

To remove this error without ending the current debug session

1. Change the data type back to its state prior to the error.
2. From the **Debug** menu, choose **Apply Code Changes**.

To remove this error without changing your source code

1. From the **Debug** menu, choose **Stop Debugging**.
2. From the **Build** menu, choose **Build**.

For further information, see the Supported Code Changes.

Fatal Error C1093

Error Message

API call 'function name' failed 'location of call' : 'text from run-time'

A call to a .NET function failed. ***text from run-time*** may or may not be supplied by the COM runtime.

For more information about system error messages, see the winerror.h system file, and FormatMessage.

Fatal Error C1094

Error Message
'-Zmval1' : command line option is inconsistent with value used to build precompiled header ('-Zmval2')

The value that is passed to /Yc must be the same value that is passed to /Yu (**/Zm** values must be the same in all compilations that use or create the same precompiled header file).

The following sample generates C1094:

```
// C1094.h
int func1();
```

And then,

```
// C1094.cpp
// compile with: /Yc"C1094.h" /Zm200
int u;
int main() {
   int sd = 32;
}
#include "C1094.h"
```

And then,

```
// C1094b.cpp
// compile with: /Yu"C1094.h" /Zm300 /c
#include "C1094.h"   // C1094 compile with /Zm200
void Test() {}
```

Fatal Error C1098

Error Message
Version mismatch with Edit and Continue engine

The debugger version you are using does not match the compiler used to create the executable. If recompiling does not fix the problem, you may need to reinstall Visual C++ to make sure you have the proper versions of the debugger and compiler.

Fatal Error C1099

Error Message
Edit and Continue engine terminating compile

Edit and Continue loaded a precompiled header file in preparation for compiling code changes, but subsequent actions (such as code changes prior to the precompiled header **#include** statement or stopping the debugger) prevented Edit and Continue from finishing the compile with that process. You do not need to take any action to fix this error.

Fatal Error C1100

Error Message
unable to initialize OLE : system error message

The compiler cannot initialize the Component Object Model (COM) library. See CoInitialize.

Fatal Error C1103

Error Message
fatal error importing progid: 'message'

The compiler detected a problem importing a type library. For example, you cannot specify a type library with progid and also specify **no_registry**.

For more information, see The #import Directive.

The following sample will generate C1103:

```
// C1103.cpp
#import "progid:a.b.id.1.5" no_registry auto_search   // C1103
```

Fatal Error C1104

Error Message
fatal error importing libid: 'message'

The compiler detected a problem importing a type library. For example, you cannot specify a type library with libid and also specify **no_registry**.

For more information, see The #import Directive.

The following sample will generate C1104:

```
// C1104.cpp
#import "libid:11111111.1111.1111.1111.111111111111" version("4.0") lcid("9") no_registry a
uto_search   // C1104
```

Fatal Error C1107

Error Message
could not find assembly 'file': please specify the assembly search path using /AI or by setting the LIBPATH environment variable

A metadata file was passed to a #using directive that the compiler was unable to locate.

LIBPATH, which is described in the topic for **#using**, and the /AI compiler option allow you to specify directories in which the compiler will look for referenced metadata files.

Fatal Error C1108

Error Message
unable to find DLL: 'dll'

The specified DLL (***dll***) could not be found in the path.

To resolve this error, reinstall Visual C++ or copy the appropriate .dll file from the installation CD to your computer.

Fatal Error C1109

Error Message
unable to find 'entry point' in DLL 'dll'

An entry point in a delay-loaded DLL required by the compiler could not be found.

Fatal Error C1113

Error Message
#using failed on 'file'

Only a file in the Microsoft Intermediate Language (MSIL) format can be passed to a #using directive. The /clr compiler option lets you create an MSIL output file. Other Visual Studio languages also produce MSIL files.

Fatal Error C1120

Error Message
call to GetProcAddress failed for 'function'

This error indicates Visual C++ needs to be reinstalled.

Fatal Error C1121

Error Message
call to CryptoAPI failed

The compiler made a call to the CryptoAPI and the call failed. Reinstall Visual Studio and possibly reinstall the operating system.

Fatal Error C1126

Error Message
'identifier' : automatic allocation exceeds size

Space allocated for local variables of a function (plus a limited amount of space used by the compiler, such as an extra 20 bytes for swappable functions) exceeds the limit.

To correct this error, use **malloc** or **new** to allocate large amounts of data.

Fatal Error C1128

Error Message
number of sections exceeded object file format limit : compile with /bigobj

An .obj file exceeded the number of allowable sections, a COFF object file format limitation.

Reaching this section limitation can be the result of using /Gy and a debug build; **/Gy** causes functions to go into their own COMDAT sections. In a debug build, there is a debug info section for each COMDAT function.

C1128 can also be caused when there are too many inline functions.

To correct this error, divide your source file into multiple source code files, compile without **/Gy**, or compile with /bigobj (Increase Number of Sections in .obj file). If you do not compile with **/Gy**, you will need to specify the optimizations individually, since **/O2** and **/O1** both imply **/Gy**.

If possible, compile without debugging information.

You may also need to have specific instantiations of templates in separate source code files, rather than having the compiler emit them.

When porting code, C1128 will likely appear first when using the Itanium (IPF) compiler, and then with the x64 compiler, and much later with the x86 compiler. IPF will have at least 4 sections associated with each function compiled **/Gy** or inlined from templates or class-inline: code, function descriptor, pdata, and debug info, and possibly xdata. x64 will be the same except for the function descriptor. X86 won't have the pdata.

Fatal Error C1189

Error Message
#error : user supplied error message

C1189 is generated by **#error**.

For more information, see The #error Directive.

The following sample generates C1189:

```
// C1189.cpp
#undef _WIN32
#if !defined(_WIN32)
#error _WIN32 must be defined   // C1189
#endif
```

You may also see this error if you build an ATL project with the /robust MIDL compiler option. /robust is only for use when building for a Windows 2000 or later machine. So, either remove /robust or change this line in the dlldatax.c file:

```
#define _WIN32_WINNT 0x0400   // for WinNT 4.0 or Win95 with DCOM
```

to

```
#define _WIN32_WINNT 0x0500   // for WinNT 4.0 or Win95 with DCOM
```

To remove /robust, specify /no_robust in the Advanced property page in the MIDL property page folder. See MIDL Property Pages: Advanced for more information.

Fatal Error C1190

Error Message
managed targeted code requires a '/clr' option

You are using CLR constructs but you did not specify **/clr**.

For more information, see /clr (Common Language Runtime Compilation).

The following sample generates C1190:

```
// C1190.cpp
// compile with: /c
__gc class A {};   // C1190
ref class A {};
```

Fatal Error C1191

Error Message
'dll' can only be imported at global scope

The instruction to import mscorlib.dll into a program that uses /clr programming cannot appear in a namespace or function, but must appear at global scope.

The following sample generates C1191:

```
// C1191.cpp
// compile with: /clr
namespace sample {
   #using <mscorlib.dll>   // C1191 not at global scope
}
```

Possible resolution:

```
// C1191b.cpp
// compile with: /clr /c
#using <mscorlib.dll>
namespace sample {}
```

Fatal Error C1192

Error Message
#using failed on 'file'

Only a file in the Microsoft Intermediate Language (MSIL) format can be passed to a #using directive. The /clr compiler option lets you create an MSIL output file. Other Visual Studio languages also produce MSIL files.

Fatal Error C1196

Error Message

'identifier' : identifier found in type library 'typelib' is not a valid C++ identifier

One of the identifiers in your type library is not a valid C++ identifier. The type library is not available for use with #import.

Fatal Error C1197

Error Message
cannot reference 'mscorlib.dll_1' as the program has already referenced 'mscorlib.dll_2'

The compiler is matched to a version of the common language runtime. However, an attempt was made to reference a version of a common language runtime file from a previous version.

To resolve this error, only reference files from the version of the common language runtime that shipped with the version of Visual C++ you are compiling with.

Example
The following sample generates C1197:

```
// C1197.cpp
// compile with: /clr /c
// processor: x86
#using "C:\Windows\Microsoft.NET\Framework\v1.1.4322\mscorlib.dll"   // C1197
// try the following line instead
// #using "mscorlib.dll"
```

Fatal Error C1201

Error Message
unable to continue after syntax error in class template definition

An unexpected error occurred while parsing a class template definition.

Fix any other errors and recompile. If that fails, note the circumstances of the error, try to isolate the problem and create a reproducible test case, then contact Microsoft Product Support Services.

Fatal Error C1202

Error Message
recursive type or function dependency context too complex

A template definition was recursive or exceeded complexity limits.

Example
The following sample generates C1202.

```
// C1202.cpp
// processor: x86 IPF
template<int n>
class Factorial : public Factorial<n-1>  {   // C1202
public:
   operator int () {
      return Factorial <n-1>::operator int () * n;
   }
};
Factorial<7> facSeven;
```

Possible resolution.

```
// C1202b.cpp
// compile with: /c
template<int n>
class Factorial : public Factorial<n-1> {
public:
   operator int () {
      return Factorial <n-1>::operator int () * n;
   }
};

template <>
class Factorial<0> {
public:
   operator int () {
      return 1;
   }
};

Factorial<7> facSeven;
```

Fatal Error C1205

Error Message

Generics are not supported by the version of the runtime installed

The version of the common language runtime that the compiler is using is not a version of the runtime supported by the current compiler. For example, the generics feature requires a runtime that is matched to the compiler.

Your path specification may need to be modified.

Fatal Error C1206

Error Message
Per-appdomain data is not supported by the version of the runtime installed

Some features, such as per application domain data, are only supported by the common language runtime that supports the feature.

C1206 indicates that the latest version of the runtime is not installed on your computer. Install the common language runtime version that is intended for use with your compiler.

See appdomain for more information.

Fatal Error C1207

Error Message
Managed templates not supported by the version of the runtime installed

C1207 occurs when you have a compiler for the current release, but a common language runtime from a previous release.

Some functionality of the compiler may not work on a previous version of the run time.

To resolve C1207 install the common language runtime version that is intended for use with your compiler.

Fatal Error C1208

Error Message

Allocating reference classes on the stack is not supported by the version of the runtime installed

C1208 occurs when you have a compiler for the current release, but a common language runtime from a previous release.

Some functionality of the compiler may not work on a previous version of the run time.

Install the common language runtime version that is intended for use with your compiler.

Fatal Error C1209

Error Message
Friend assemblies not supported by the version of the runtime installed

C1208 occurs when you have a compiler for the current release, but a common language runtime from a previous release.

Some functionality of the compiler may not work on a previous version of the run time.

To resolve C1209, install the common language runtime that shipped with the compiler you are using.

For more information, see Friend Assemblies.

Fatal Error C1210

Error Message
/clr:pure and /clr:safe are not supported by the version of the runtime installed

C1210 occurs when you have a compiler for the current release, but a common language runtime from a previous release.

Some functionality of the compiler may not work on a previous version of the run time.

To resolve C1210 install the common language runtime version that is intended for use with your compiler.

Fatal Error C1211

Error Message

The TypeForwardedTo Custom Attribute is not supported by the version of the runtime installed

C1211 occurs when you have a compiler for the current release, but a common language runtime from a previous release.

Some functionality of the compiler may not work on a previous version of the run time.

To resolve C1211 install the common language runtime that shipped with the compiler you are using.

For more information, see Type Forwarding.

Fatal Error C1305

Error Message
profile database 'pgd_file' is for a different architecture

A .pgd file that was generated from the /LTCG:PGINSTRUMENT operation for another platform was passed to /LTCG:PGOPTIMIZE . Profile-guided optimizations are available for x86, Itanium Processor Family (IPF), and x64 platforms. However, a .pgd file generated with a /LTCG:PGINSTRUMENT operation for one platform is not valid as input to a /LTCG:PGOPTIMIZE for a different platform.

To resolve this error, only pass a .pgd file that is created with /LTCG:PGINSTRUMENT to /LTCG:PGOPTIMIZE on the same platform.

Fatal Error C1307

Error Message
program has been edited since profile data was collected

When using /LTCG:PGOPTIMIZE, the linker detected an input module that was recompiled after /LTCG:PGINSTRUMENT and that the module was changed to the point where existing profile data is no longer relevant. For example, if the call graph changed in the recompiled module, the compiler will generate C1307.

To resolve this error, run /LTCG:PGINSTRUMENT, redo all test runs, and run /LTCG:PGOPTIMIZE. If you cannot run /LTCG:PGINSTRUMENT and redo all test runs, use /LTCG:PGUPDATE instead of /LTCG:PGOPTIMIZE to create the optimized image.

Fatal Error C1308

Error Message
linking assemblies is not supported

A .netmodule is allowed as input to the linker, but an assembly is not. This error can be generated when an attempt is made to link an assembly compiled with **/clr:safe**.

For more information, see .netmodule Files as Linker Input.

The following sample generates C1308:

```
// C1308.cpp
// compile with: /clr:safe /LD
public ref class MyClass {
public:
   int i;
};
```

and then,

```
// C1308b.cpp
// compile with: /clr /link C1308b.obj C1308.dll
// C1308 expected
#using "C1308.dll"
int main() {
   MyClass ^ my = gcnew MyClass();
}
```

Fatal Error C1309

Error Message
Mismatched versions of C2.DLL and PGODB80.DLL

The toolset you are using to build and use Profile-Guided Optimizations contains mismatched components. If you cannot manually resolve this error, reinstall Visual C++.

Fatal Error C1310

Error Message
profile guided optimizations are not available with OpenMP

You will not be able to link with /LTCG:PGI any module that was compiled with /GL.

The following sample generates C1310:

```
// C1310.cpp
// compile with: /openmp /GL /link /LTCG:PGI
// C1310 expected
int main()
{
   int i = 0, j = 10;

   #pragma omp parallel
   {
      #pragma omp for
      for (i = 0; i < 0; i++)
         --j;
   }
}
```

Fatal Error C1311

Error Message
COFF format cannot statically initialize 'var' with number byte(s) of an address

An address whose value is not known at compile time cannot be statically assigned to a variable whose type has storage of less than four bytes.

This error can occur on code that is otherwise valid C++.

The following snippet shows one condition that might cause C1311.

```
char c = (char)"Hello, world";   // C1311
char *d = (char*)"Hello, world";   // OK
```

Fatal Error C1312

Error Message

Too many conditional branches in function. Simplify or refactor source code.

The code is too complex for the compiler to process without running out of stack memory. Simplify your code.

Fatal Error C1313

Error Message

compiler limit : type blocks may not be nested deeper than number levels

Exception handling (or structured exception handling) blocks were nested too deeply. Simplify your code.

Fatal Error C1350

Error Message
error loading dll 'dll': dll not found

The DLL that supports the attempted operation was not found.

This indicates a problem with your installation and you should reinstall the DLL from your product CD.

Fatal Error C1351

Error Message
error loading dll 'dll': incompatible version

The wrong version of a DLL was found.

This indicates a problem with your installation and you should reinstall the DLL from your product CD.

Fatal Error C1352

Error Message
Invalid or corrupt MSIL in function 'function' from module 'file'

A .netmodule was passed to the compiler, but the compiler detected corruption in the file. Ask the person who produced the .netmodule to investigate.

The compiler does not check .netmodule files for all types of corruption. It does, however, check that all control paths in a function contain a return statement.

For more information, see .netmodule Files as Linker Input.

Fatal Error C1353

Error Message

metadata operation failed: runtime not installed or version mismatch

Reinstall your CLR or Visual Studio.

Fatal Error C1382

Error Message
the PCH file 'file' has been rebuilt since 'obj' was generated. Please rebuild this object

When using /LTCG, the compiler detected a .pch file that is newer than a CIL .obj that points to it. The information in the CIL .obj file is out of date. Rebuild the object.

C1382 can also occur if you compile with **/Yc**, but also pass multiple source code files to the compiler. To resolve, do not use **/Yc** when passing multiple source code files to the compiler. For more information, see /Yc (Create Precompiled Header File).

Fatal Error C1383

Error Message
compiler option /GL is incompatible with the installed version of common language runtime

C1383 occurs when you are using a previous version of the common language runtime with a newer compiler, and when you compile with **/clr** and **/GL.**

To resolve, either do not use **/GL** with **/clr** or install the version of the common language runtime that shipped with your compiler.

For more information, see /clr (Common Language Runtime Compilation) and /GL (Whole Program Optimization).

Fatal Error C1505

Error Message
unrecoverable parser look-ahead error

The compiler cannot evaluate the code. Simplify the code by splitting classes or functions.

Fatal Error C1506

Error Message
unrecoverable block scoping error

The block was too large to compile.

To fix by checking the following possible causes

1. Mismatched braces
2. Unusually large function or class

Fatal Error C1507

Error Message

previous user errors and subsequent error recovery halt further compilation

There are too many errors to continue. Fix some of the errors and compile again.

Fatal Error C1508

Error Message

compiler limit : 'function' : more than 65535 argument bytes

The formal parameters to the function exceed the limit of 65535 bytes.

Fatal Error C1509

Error Message
compiler limit : too many exception handler states in function 'function'. simplify function

The code exceeds an internal limit on exception handler states (32,768 states).

The most common cause is that the function contains a complex expression of user-defined class variables and arithmetic operators.

To fix by using the following possible solutions

1. Simplify expressions by assigning common subexpressions to temporary variables.
2. Split the function into smaller functions.

Fatal Error C1600

Error Message
unsupported data type

This error indicates a mismatch in your compiler .exe files, which may have occurred because of an incomplete installation. For example, you may have installed a service pack but not the Processor Pack. Install all required products.

Fatal Error C1601

Error Message
unsupported inline assembly opcode

This error indicates a mismatch in your compiler .exe files, which may have occurred because of an incomplete installation. For example, you may have installed a service pack but not the Processor Pack. Install all required products.

Fatal Error C1602

Error Message
unsupported intrinsic

This error indicates a mismatch in your compiler .exe files, which may have occurred because of an incomplete installation. For example, you may have installed a service pack but not the Processor Pack. Install all required products.

Fatal Error C1603

Error Message
inline assembly branch target out of range by 'number' bytes

The computed distance between a JCXZ or JECXZ instruction and its specified target label was greater than 128 bytes. Update your code so that the label is closer to the instruction.

Fatal Error C1852

Error Message
'filename' is not a valid precompiled header file

The file is not a precompiled header.

To fix by checking the following possible causes

1. Invalid file specified with **/Yu** or **#pragma hdrstop**.
2. The compiler assumes a file extension of .pch if you do not specify otherwise.

Fatal Error C1853

Error Message
'filename' precompiled header file is from a previous version of the compiler, or the precompiled header is C++ and you are using it from C (or vice versa)

Possible causes

- The precompiled header was compiled with a previous compiler version. Try recompiling the header with the current compiler.
- The precompiled header is C++ and you are using it from C.

Fatal Error C1854

Error Message
cannot overwrite information formed during creation of the precompiled header in object file: 'filename'

You specified the **/Yu** (use precompiled header) option after specifying the **/Yc** (create precompiled header) option for the same file. Certain declarations (such as declarations including **__declspec dllexport**) make this invalid.

Fatal Error C1900

Error Message

ll mismatch between 'tool1' version 'number1' and 'tool2' version 'number2'

Tools run in various passes of the compiler do not match. ***number1*** and ***number2*** refer to the dates on the files. For example, in pass 1, the compiler front end runs (c1.dll) and in pass 2, the compiler back end runs (c2.dll). The dates on the files must match and if they do not, reinstall and use the current version of each tool.

Fatal Error C1902

Error Message
program database manager mismatch; please check your installation

A program database file (.pdb) was created using a newer version of mspdb80.dll than the one found while compiling.

This error usually indicates that mspdbsrv.exe or mspdbcore.dll are missing or have different versions from mspdb80.dll. Ensure matching versions of mspdbsrv.exe, mspdbcore.dll, and mspdb80.dll are installed on your system.

Fatal Error C1903

Error Message

unable to recover from previous error(s); stopping compilation

The compiler found too many errors to continue. Fix the errors and recompile.

Fatal Error C1904

Error Message
bad provider interaction: 'file'

This error indicates the failure of an attribute provider.

Note the circumstances of the error, try to isolate the problem and create a reproducible test case, then contact Getting Help from Microsoft Product Support Services (Visual Studio).

Fatal Error C1905

Error Message

Front end and back end not compatible (must target same processor)

This error occurs when a .obj file is generated by a compiler front end (C1.dll) that targets one processor, such as x86, Itanium Processor Family (IPF), or x64, but is being read by a back end (C2.dll) that targets a different processor.

Ensure that you are using a matching front end and back end.

Fatal Error C1999

Error Message

UNKNOWN FATAL ERRORChoose the Technical Support command on the Visual C++Help menu, or open the Technical Support help file for more information

Note the circumstances of the error, try to isolate the problem and create a reproducible test case, then contact Microsoft Product Support Services.

Compiler Errors C2001 through C2099

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, type the error number in the **Look for** box in the Index, or click a link below.

Compiler Error C2097

Compiler Error C2095

Compiler Error C2094

Compiler Error C2093

Compiler Error C2092

Compiler Error C2091

Compiler Error C2090

Compiler Error C2089

Compiler Error C2088

Compiler Error C2087

Compiler Error C2086

Compiler Error C2085

Compiler Error C2084

Compiler Error C2083

Compiler Error C2082

Compiler Error C2081

Compiler Error C2079

Compiler Error C2078

Compiler Error C2077

Compiler Error C2075

Compiler Error C2074

Compiler Error C2073

Compiler Error C2072

Compiler Error C2071

Compiler Error C2070

Compiler Error C2069

Compiler Error C2067

Compiler Error C2066

Compiler Error C2065

Compiler Error C2064

Compiler Error C2063

Compiler Error C2062

Compiler Error C2061

Compiler Error C2060

Compiler Error C2059

Compiler Error C2058
Compiler Error C2057
Compiler Error C2056
Compiler Error C2055
Compiler Error C2054
Compiler Error C2053
Compiler Error C2052
Compiler Error C2051
Compiler Error C2050
Compiler Error C2048
Compiler Error C2047
Compiler Error C2046
Compiler Error C2045
Compiler Error C2044
Compiler Error C2043
Compiler Error C2042
Compiler Error C2041
Compiler Error C2040
Compiler Error C2039
Compiler Error C2036
Compiler Error C2034
Compiler Error C2033
Compiler Error C2032
Compiler Error C2030
Compiler Error C2028
Compiler Error C2027
Compiler Error C2026
Compiler Error C2022
Compiler Error C2021
Compiler Error C2020
Compiler Error C2019
Compiler Error C2018
Compiler Error C2017
Compiler Error C2015
Compiler Error C2014
Compiler Error C2013
Compiler Error C2012
Compiler Error C2011
Compiler Error C2010
Compiler Error C2009

Compiler Error C2008

Compiler Error C2007

Compiler Error C2006

Compiler Error C2005

Compiler Error C2004

Compiler Error C2003

Compiler Error C2002

Compiler Error C2001

Compiler Error C2001

Error Message
newline in constant

A string constant cannot be continued on a second line unless you do the following:

- End the first line with a backslash.
- Close the string on the first line with a double quotation mark and open the string on the next line with another double quotation mark.

Ending the first line with \n is not sufficient.

Example
The following sample generates C2001:

```
// C2001.cpp
// C2001 expected
#include <stdio.h>

int main()
{
    printf_s("Hello,
             world");
    printf_s("Hello,\n
             world");
}
```

Spaces at the beginning of the next line after a line-continuation character are included in the string constant. None of the examples shown above embed a newline character into the string constant. You can embed a newline character as shown here:

```
// C2001b.cpp
#include <stdio.h>

int main()
{
    printf_s("Hello,\n\
             world");

    printf_s("Hello,\
             \nworld");

    printf_s("Hello,\n"
             "world");

    printf_s("Hello,"
             "\nworld");

    printf_s("Hello,"
             " world");

    printf_s("Hello,\
             world");
}
```

Compiler Error C2002

Error Message
invalid wide-character constant

The multibyte-character constant was not legal.

To fix by checking the following possible causes

1. The wide-character constant contains more bytes than expected.
2. The standard header STDDEF.h is not included.
3. Wide characters cannot be concatenated with ordinary string literals.
4. A wide-character constant must be preceded by the character 'L':

```
L'mbconst'
```

Compiler Error C2003

Error Message
expected 'defined id'

An identifier must follow the preprocessor keyword.

Compiler Error C2004

Error Message
expected 'defined(id)'

An identifier must appear in the parentheses following the preprocessor keyword.

This error can also be generated as a result of compiler conformance work that was done for Visual Studio .NET 2003: missing parenthesis in preprocessor directive. If the closing parenthesis is missing from a preprocessor directive, the compiler will generate an error.

See Summary of Compile-Time Breaking Changes for more information.

Example

The following sample generates C2004:

```
// C2004.cpp
// compile with: /DDEBUG
#include <stdio.h>

int main()
{
    #if defined(DEBUG   // C2004
        printf_s("DEBUG defined\n");
    #endif
}
```

Possible resolution:

```
// C2004b.cpp
// compile with: /DDEBUG
#include <stdio.h>

int main()
{
    #if defined(DEBUG)
        printf_s("DEBUG defined\n");
    #endif
}
```

Compiler Error C2005

Error Message
#line expected a line number, found 'token'

The **#line** directive must be followed by a line number.

The following sample generates C2005:

```
// C2005.cpp
int main() {
   int i = 0;
   #line i   // C2005
}
```

Possible resolution:

```
// C2005b.cpp
int main() {
   int i = 0;
   #line 0
}
```

Compiler Error C2006

Error Message
'directive' expected a filename, found 'token'

Directives such as #include or #import require a filename. To resolve the error, make sure *token* is a valid filename. Also, put the filename in double quotes or angle brackets.

The following sample generates C2006:

```
// C2006.cpp
#include stdio.h   // C2006
```

Possible resolution:

```
// C2006b.cpp
// compile with: /c
#include <stdio.h>
```

Compiler Error C2007

Error Message
#define syntax

No identifier appears after a **#define**. To resolve the error, use an identifier.

The following sample generates C2007:

```
// C2007.cpp
#define   // C2007
```

Possible resolution:

```
// C2007b.cpp
// compile with: /c
#define true 1
```

Compiler Error C2008

Error Message
'character' : unexpected in macro definition

The character appears immediately following the macro name. To resolve the error, there must be a space after the macro name.

The following sample generates C2008:

```
// C2008.cpp
#define TEST1"mytest1"    // C2008
```

Possible resolution:

```
// C2008b.cpp
// compile with: /c
#define TEST2 "mytest2"
```

Compiler Error C2009

Error Message
reuse of macro formal 'identifier'

The formal parameter list of a macro definition uses the identifier more than once. Identifiers in the macro's parameter list must be unique.

Example
The following sample generates C2009:

```
// C2009.cpp
#include <stdio.h>

#define macro1(a,a) (a*a)   // C2009

int main()
{
    printf_s("%d\n", macro1(2));
}
```

Possible resolution:

```
// C2009b.cpp
#include <stdio.h>

#define macro2(a)   (a*a)
#define macro3(a,b) (a*b)

int main()
{
    printf_s("%d\n", macro2(2));
    printf_s("%d\n", macro3(2,4));
}
```

Compiler Error C2010

Error Message
'character' : unexpected in macro formal parameter list

The character is used incorrectly in the formal parameter list of a macro definition. Remove the character to resolve the error.

The following sample generates C2010:

```
// C2010.cpp
// compile with: /c
#define mymacro(a|) (2*a)   // C2010
#define mymacro(a) (2*a)   // OK
```

Compiler Error C2011

Error Message
'identifier' : 'type' type redefinition

The identifier was already defined as *type*. You may also get C2011 if you import a type library more than once into the same file.

The following sample generates C2011:

```
// C2011.cpp
// compile with: /c
struct S;
union S;   // C2011
union S2;   // OK
```

Compiler Error C2012

Error Message
missing name following '<'

An **#include** directive lacks the required filename.

The following sample generates C2012:

```
// C2012.cpp
#include <   // C2012 include the filename to resolve
```

Possible resolution:

```
// C2012b.cpp
// compile with: /c
#include <stdio.h>
```

Compiler Error C2013

Error Message
missing '>'

An **#include** directive lacks a closing angle bracket. Add the closing bracket to resolve the error.

The following sample generates C2013:

```
// C2013.cpp
#include <stdio.h    // C2013
```

Possible resolution:

```
// C2013b.cpp
// compile with: /c
#include <stdio.h>
```

Compiler Error C2014

Error Message
preprocessor command must start as first nonwhite space

The # sign of a preprocessor directive must be the first character on a line that is not white space.

The following sample generates C2014:

```
// C2014.cpp
int k; #include <stdio.h>   // C2014
```

Possible resolution:

```
// C2014b.cpp
// compile with: /c
int k;
#include <stdio.h>
```

Compiler Error C2015

Error Message
too many characters in constant

A character constant contains more than two characters. The limit is one character for standard character constants and two characters for long character constants.

An escape sequence, such as \t, is converted to a single character.

Example
The following sample generates C2015:

```
// C2015.cpp
// compile with: /c

char test1 = 'error';   // C2015
char test2 = 'e';   // OK
```

C2015 can also occur when using a Microsoft extension, character constants converted to integers. The following sample generates C2015:

```
// C2015b.cpp
#include <stdio.h>

int main()
{
    int a = 'abcde';   // C2015

    int b = 'a';   // 'a' = ascii 0x61
    printf_s("%x\n", b);
}
```

Compiler Error C2017

Error Message
illegal escape sequence

An escape sequence, such as \t, appears outside of a character or string constant.

The following sample generates C2017:

```
// C2017.cpp
int main() {
   char test1='a'\n;   // C2017
   char test2='a\n';   // ok
}
```

C2017 can occur when the stringize operator is used with strings that include escape sequences.

The following sample generates C2017:

```
// C2017b.cpp
#define TestDfn(x) AfxMessageBox(#x)
TestDfn(CString("\\") + CString(".h\"\n\n"));   // C2017
```

Compiler Error C2018

Error Message
unknown character 'hexnumber'

The source file contains an unexpected ASCII character, which is identified by its hex number. To resolve the error, remove the character.

Compiler Error C2019

Error Message
expected preprocessor directive, found 'character'

The character followed a # sign but it is not the first letter of a preprocessor directive.

The following sample generates C2019:

```
// C2019.cpp
#!define TRUE 1   // C2019
```

Possible resolution:

```
// C2019b.cpp
// compile with: /c
#define TRUE 1
```

Compiler Error C2020

Error Message
'member' : 'class' member redefinition

A member inherited from a base class or structure is redefined. Inherited members cannot be redefined unless declared as **virtual** in the base class.

Compiler Error C2021

Error Message
expected exponent value, not 'character'

The character used as the exponent of a floating-point constant is not a valid number. Be sure to use an exponent that is in range.

Example
The following sample generates C2021:

```
// C2021.cpp
float test1=1.175494351E;   // C2021
```

Possible resolution:

```
// C2021b.cpp
// compile with: /c
float test2=1.175494351E8;
```

Compiler Error C2022

Error Message
'number' : too big for character

The octal number following a backslash (\) in a character or string constant is too big to represent a character.

Compiler Error C2026

Error Message
string too big, trailing characters truncated

The string was longer than the limit of 16380 single-byte characters.

Prior to adjacent strings being concatenated, a string cannot be longer than 16380 single-byte characters.

A Unicode string of about one half this length would also generate this error.

If you have a string defined as follows, it generates C2026:

```
char sz[] =
"\
imagine a really, really \
long string here\
";
```

You could break it up as follows:

```
char sz[] =
"\
imagine a really, really "
"long string here\
";
```

You may want to store exceptionally large string literals (32K or more) in a custom resource or an external file. See Creating a New Custom or Data Resource for more information.

Compiler Error C2027

Error Message
use of undefined type 'type'

A type cannot be used until it is defined. To resolve the error, be sure the type is fully defined before referencing it.

Example

The following sample generates C2027.

```
// C2027.cpp
class C;
class D {
public:
   void func() {
   }
};

int main() {
   C *pC;
   pC->func();   // C2027

   D *pD;
   pD->func();
}
```

It is possible to declare a pointer to a declared but undefined type. But Visual C++ does not allow a reference to an undefined type.

The following sample generates C2027.

```
// C2027_b.cpp
class A;
A& CreateA();

class B;
B* CreateB();

int main() {
   CreateA();   // C2027
   CreateB();   // OK
}
```

Compiler Error C2028

Error Message
struct/union member must be inside a struct/union

Structure or union members must be declared within the structure or union.

Compiler Error C2030

Error Message
'identifier' : struct/union member redefinition

A structure or union uses the same identifier for more than one member.

Compiler Error C2032

Error Message
'identifier' : function cannot be member of struct/union 'structorunion'

The structure or union has a member function, which is allowed in C++ but not in C. To resolve the error, either compile as a C++ program or remove the member function.

The following sample generates C2032:

```
// C2032.c
struct z {
   int i;
   void func();   // C2032
};
```

Possible resolution:

```
// C2032b.c
// compile with: /c
struct z {
   int i;
};
```

Compiler Error C2033

Error Message
'identifier' : bit field cannot have indirection

The bit field was declared as a pointer, which is not allowed.

The following sample generates C2033:

```
// C2033.cpp
struct S {
   int *b : 1;  // C2033
};
```

Possible resolution:

```
// C2033b.cpp
// compile with: /c
struct S {
   int b : 1;
};
```

Compiler Error C2034

Error Message
'identifier' : type of bit field too small for number of bits

The number of bits in the bit-field declaration exceeds the size of the base type.

The following sample generates C2034:

```
// C2034.cpp
struct A {
   char test : 9;   // C2034, char has 8 bits
};
```

Possible resolution:

```
// C2034b.cpp
// compile with: /c
struct A {
   char test : 8;
};
```

Compiler Error C2036

Error Message
'identifier' : unknown size

An operation on *identifier* requires the size of the data object, which cannot be determined.

Example
The following sample generates C2036.

```
// C2036.c
// a C program
struct A* pA;
struct B { int i; } *pB;
int main() {
   pA++;   // C2036, size of A not known
   ((char*)pA)++;   // OK

   pB++;   // OK
}
```

The following sample generates C2036.

```
// C2036_2.cpp
// a C++ program
struct A* pA;
int main() {
   pA++;   // C2036, size of A not known
   ((char*&)pA)++;   // OK, if sizeof(A) == sizeof(char)
}
```

Compiler Error C2039

Error Message
'identifier1' : is not a member of 'identifier2'

The code incorrectly calls or refers to a member of a structure, class, or union.

Example
The following sample generates C2039.

```
// C2039.cpp
struct S {
   int mem0;
} s, *pS = &s;

int main() {
   pS->mem1 = 0;   // C2039 mem1 is not a member
   pS->mem0 = 0;   // OK
}
```

The following sample generates C2039.

```
// C2039_b.cpp
// compile with: /clr
using namespace System;
int main() {
   Console::WriteLine( "{0}", DateTime::get_Now());   // C2039
   Console::WriteLine( "{0}", DateTime::Now);   // OK
   Console::WriteLine( "{0}", DateTime::Now::get());   // OK
}
```

The following sample generates C2039.

```
// C2039_c.cpp
// compile with: /clr /c
ref struct S {
   property int Count {
     int get();
     void set(int i){}
   };
};

int S::get_Count() { return 0; }   // C2039
int S::Count::get() { return 0; }   // OK
```

C2039 can also occur if you attempt to access a default indexer incorrectly. The following sample defines a component authored in C#.

```
// C2039_d.cs
// compile with: /target:library
// a C# program
[System.Reflection.DefaultMember("Item")]
public class B {
   public int Item {
      get { return 13; }
      set {}
   }
};
```

The following sample generates C2039.

```
// C2039_e.cpp
// compile with: /clr
using namespace System;
#using "c2039_d.dll"

int main() {
   B ^ b = gcnew B;
   int n = b->default;   // C2039
   // try the following line instead
   // int n = b->Item;
   Console::WriteLine(n);
}
```

C2039 can also occur if you use generics. The following sample generates C2039.

```
// C2039_f.cpp
// compile with: /clr
interface class I {};

ref struct R : public I {
   virtual void f3() {}
};

generic <typename T>
where T : I
void f(T t) {
   t->f3();   // C2039
   safe_cast<R^>(t)->f3();   // OK
}

int main() {
   f(gcnew R());
}
```

C2039 can occur when attempting to write code that will deterministic release of your managed or unmanaged resources. For more information, see Destructors and Finalizers in Visual C++.

The following sample generates C2039.

```
// C2039_g.cpp
// compile with: /clr
using namespace System;
using namespace System::Threading;

void CheckStatus( Object^ stateInfo ) {}

int main() {
   ManualResetEvent^ event = gcnew ManualResetEvent( false );
   TimerCallback^ timerDelegate = gcnew TimerCallback( &CheckStatus );
   Timer^ stateTimer = gcnew Timer( timerDelegate, event, 1000, 250 );

   ((IDisposable ^)stateTimer)->Dispose();   // C2039

   stateTimer->~Timer();   // OK
}
```

Compiler Error C2040

Error Message
'operator' : 'identifier1' differs in levels of indirection from 'identifier2'

An expression involving the operators has inconsistent levels of indirection.

If both operands are arithmetic or both are nonarithmetic (such as array or pointer), they are used without change. If one operand is arithmetic and the other is not, the arithmetic operator is converted to the nonarithmetic type.

Compiler Error C2041

Error Message
illegal digit 'character' for base 'number'

The specified character is not a valid digit for the base (such as octal or hex).

The following sample generates C2041:

```
// C2041.cpp
int i = 081;   // C2041  8 is not a base 8 digit
```

Possible resolution:

```
// C2041b.cpp
// compile with: /c
int j = 071;
```

Compiler Error C2042

Error Message
signed/unsigned keywords mutually exclusive

The keywords **signed** and **unsigned** are used in a single declaration.

The following sample generates C2042:

```
// C2042.cpp
unsigned signed int i;   // C2042
```

Possible resolution:

```
// C2042b.cpp
// compile with: /c
unsigned int i;
signed int ii;
```

Compiler Error C2043

Error Message
illegal break

A break is legal only within a **do**, **for**, **while**, or **switch** statement.

Compiler Error C2044

Error Message
illegal continue

A continue is legal only within a **do**, **for**, or **while** statement.

Compiler Error C2045

Error Message
'identifier' : label redefined

The label appears before multiple statements in the same function.

The following sample generates C2045:

```
// C2045.cpp
int main() {
   label: {
   }
   goto label;
   label: {}   // C2045
}
```

Compiler Error C2046

Error Message
illegal case

The keyword **case** can appear only in a **switch** statement.

The following sample generates C2046:

```
// C2046.cpp
int main() {
   case 0:   // C2046
}
```

Possible resolution:

```
// C2046b.cpp
int main() {
   int i = 0;

   switch(i) {
      case 0:
      break;

      default:
      break;
   }
}
```

Compiler Error C2047

Error Message
illegal default

The keyword **default** can appear only in a **switch** statement.

The following sample generates C2047:

```
// C2047.cpp
int main() {
   int i = 0;
   default:   // C2047
   switch(i) {
      case 0:
      break;
   }
}
```

Possible resolution:

```
// C2047b.cpp
int main() {
   int i = 0;
   switch(i) {
      case 0:
      break;
      default:
      break;
   }
}
```

Compiler Error C2048

Error Message
more than one default

A **switch** statement contains multiple **default** labels. Delete one of the **default** labels to resolve the error.

The following sample generates C2048:

```
// C2048.cpp
int main() {
   int a = 1;
   switch (a) {
      case 1:
         a = 0;
      default:
         a = 2;
      default:   // C2048
         a = 3;
   }
}
```

Possible resolution:

```
// C2048b.cpp
int main() {
   int a = 1;
   switch (a) {
      case 1:
         a = 0;
      default:
         a = 2;
   }
}
```

Compiler Error C2050

Error Message
switch expression not integral

The **switch** expression evaluates to a noninteger value. To resolve the error, use only integral values in switch statements.

The following sample generates C2050:

```
// C2050.cpp
int main() {
   int a = 1;
   switch ("a") {   // C2050
      case 1:
         a = 0;
      default:
         a = 2;
   }
}
```

Possible resolution:

```
// C2050b.cpp
int main() {
   int a = 1;
   switch (a) {
      case 1:
         a = 0;
      default:
         a = 2;
   }
}
```

Compiler Error C2051

Error Message
case expression not constant

Case expressions must be integer constants.

The following sample generates C2051:

```
// C2051.cpp
class X {};

int main() {
   static X x;
   int i = 0;

   switch (i) {
      case x:   // C2051 use constant expression to resolve error
         break;
      default:
         break;
   }
}
```

Possible resolution:

```
// C2051b.cpp
class X {};

int main() {
   static X x;
   int i = 0;

   switch (i) {
      case 1:
         break;
      default:
         break;
   }
}
```

Compiler Error C2052

Error Message
'type' : illegal type for case expression

Case expressions must be integer constants.

The following sample generates C2052:

```
// C2052.cpp
int main() {
   int index = 0;
   switch (index) {
      case 1:
         return 24;
      case 1.0:   // C2052
      // try the following line instead
      // case 2:
         return 23;
   }
}
```

Compiler Error C2053

Error Message
'identifier' : wide string mismatch

The wide string is assigned to an incompatible type.

The following sample generates C2053:

```
// C2053.c
int main() {
   char array[] = L"Rika";   // C2053
}
```

Compiler Error C2054

Error Message
expected '(' to follow 'identifier'

The function identifier is used in a context that requires trailing parentheses.

This error can be caused by omitting an equal sign (=) on a complex initialization.

The following sample generates C2054:

```
// C2054.c
int array1[] { 1, 2, 3 };   // C2054, missing =
```

Possible resolution:

```
// C2054b.c
int main() {
   int array2[] = { 1, 2, 3 };
}
```

Compiler Error C2055

Error Message
expected formal parameter list, not a type list

A function definition contains a parameter type list instead of a formal parameter list. ANSI C requires formal parameters to be named unless they are void or an ellipsis (`...`).

The following sample generates C2055:

```
// C2055.c
// compile with: /c
void func(int, char) {}  // C2055
void func (int i, char c) {}   // OK
```

Compiler Error C2056

Error Message
illegal expression

An expression was invalid because of a previous error.

Compiler Error C2057

Error Message
expected constant expression

The context requires a constant expression, an expression whose value is known at compile time.

Example
The following sample generates C2057:

```
// C2057.cpp
int i;
int b[i];   // C2057
int main() {
   const int i = 8;
   int b[i];
}
```

C has more restrictive rules for constant expressions. The following sample generates C2057:

```
// C2057b.c
#define ArraySize1 10
int main() {
   const int ArraySize2 = 10;
   int h[ArraySize2];   // C2057
   int h[ArraySize1];   // OK
}
```

Compiler Error C2058

Error Message
constant expression is not integral

The context requires an integer constant expression.

Compiler Error C2059

Error Message
syntax error : 'token'

The token caused a syntax error.

To determine the cause, examine not only the line listed in the error message, but also the lines above it. The following example generates an error message for the line declaring *j*, but the true source of the error appears on the line just above it.

If examining the lines yields no clue to what the problem might be, try commenting out the line listed in the error message and possibly several lines above it.

If the error message occurs on a symbol immediately following a **typedef** variable, check that the variable is defined in the source code.

You may get C2059 if a symbol evaluates to nothing, as can occur when you compile with **/D***symbol***=**.

```
// C2059a.cpp
// compile with: /DTEST=
#include <stdio.h>

int main() {
   #ifdef TEST
      printf_s("\nTEST defined %d", TEST);   // C2059
   #else
      printf_s("\nTEST not defined");
   #endif
}
```

Another specific reason you can get C2059 is when you compile an application that specifies a structure in the default arguments for a function. The default value for an argument must be an expression. An initializer list, such as that used to initialize a structure, is not an expression. The resolution is to define a constructor to perform the required initialization.

The following example generates C2059:

```
// C2059b.cpp
// compile with: /c
struct ag_type {
   int a;
   float b;
   // Uncomment the following line to resolve.
   // ag_type(int aa, float bb) : a(aa), b(bb) {}
};

void func(ag_type arg = {5, 7.0});   // C2059
void func(ag_type arg = ag_type(5, 7.0));   // OK
```

You can also get C2059 if you define a member template class or function outside the class. See Knowledge Base article Q241949 for more information.

C2059 can occur for an ill-formed cast.

The following sample generates C2059:

```
// C2059c.cpp
// compile with: /clr
using namespace System;
ref class From {};
ref class To : public From {};

int main() {
   From^ refbase = gcnew To();
   To^ refTo = safe_cast<To^>(From^);   // C2059
   To^ refTo2 = safe_cast<To^>(refbase);   // OK
```

```
}
```

C2059 can also occur if you attempt to create a namespace name that contains a period.

The following sample generates C2059:

```
// C2059d.cpp
// compile with: /c
namespace A.B {}   // C2059

// OK
namespace A  {
   namespace B {}
}
```

C2059 can also occur if you define a member template class or function outside the class. See Knowledge Base article Q241949 for more information. You can find Knowledge Base articles on the MSDN Library CD-ROM or at http://support.microsoft.com.

Compiler Error C2060

Error Message

syntax error : end of file found

At least one more token was expected.

Compiler Error C2061

Error Message
syntax error : identifier 'identifier'

The compiler found an identifier where it wasn't expected. Make sure that *identifier* is declared before you use it.

An initializer may be enclosed by parentheses. To avoid this problem, enclose the declarator in parentheses or make it a **typedef**.

This error could also be caused when the compiler detects an expression as a class template argument; use typename to tell the compiler it is a type.

The following sample generates C2061:

```
// C2061.cpp
// compile with: /c
template < A a >   // C2061
// try the following line instead
// template < typename b >
class c{};
```

C2061 can occur if you pass an instance name to typeid:

```
// C2061b.cpp
// compile with: /clr
ref struct G {
   int i;
};

int main() {
   G ^ pG = gcnew G;
   System::Type ^ pType = typeid<pG>;   // C2061
   System::Type ^ pType2 = typeid<G>;   // OK
}
```

Compiler Error C2062

Error Message
type 'type' unexpected

The compiler did not expect a type name.

The following sample generates C2062:

```
// C2062.cpp
// compile with: /c
struct A {  : int l; };   // C2062
struct B { private: int l; };   // OK
```

C2062 can also occur due to the way the compiler handles undefined types in a constructor's parameter list. If the compiler encounters an undefined (misspelled?) type, it assumes the constructor is an expression, and issues C2062. To resolve, only use defined types in a constructor parameter list. The following sample generates C2062:

```
// C2062_b.cpp
// compile with: /c
class DataSent {};

struct MyDataController {
   MyDataController(int age, DataSet* j) {}   // C2062
   // try the following line instead
   // MyDataController(int age, DataSent* j) {}
};
```

Compiler Error C2063

Error Message
'identifier' : not a function

The identifier is used as a function but not declared as a function.

The following sample generates C2063:

```
// C2063.c
int main() {
   int i, j;
   j = i();    // C2063, i is not a function
}
```

Possible resolution:

```
// C2063b.c
int i() { return 0;}
int main() {
   int j;
   j = i();
}
```

Compiler Error C2064

Error Message
term does not evaluate to a function taking 'number' arguments

A call is made to a function through an expression. The expression does not evaluate to a function pointer.

The following sample generates C2064:

```
// C2064.cpp
int i, j;
char* p;
void func() {
   j = i();    // C2064, i is not a function
   p();        // C2064, p doesn't point to a function
}
```

The following sample generates C2064:

```
// C2064b.cpp
struct C {
   void func1(){}
   void func2(){}
};

typedef void (C::*pFunc)();

int main() {
   C c;
   pFunc funcArray[2] = {&C::func1, &C::func2};
   (funcArray[0])();    // C2064
}
```

Possible resolution:

```
// C2064c.cpp
struct C {
   void func1(){}
   void func2(){}
};

typedef void (C::*pFunc)();

int main() {
   C c;
   pFunc funcArray[2] = {&C::func1, &C::func2};
   (c.* funcArray[0])();
}
```

The following sample generates C2064:

```
// C2064d.cpp
struct C {
   typedef void (C::*pFunc)();
   pFunc funcArray[2];
   void func1(){}
   void func2(){}
   C() {
      funcArray[0] = C::func1;
      funcArray[1] = C::func2;
   }
   void func3() {
```

```
      (funcArray[0])();   // C2064
   }
};
```

Possible resolution:

```
// C2064e.cpp
// compile with: /c
struct C {
   typedef void (C::*pFunc)();
   pFunc funcArray[2];
   void func1(){}
   void func2(){}
   C() {
      funcArray[0] = &C::func1;
      funcArray[1] = &C::func2;
   }
   void func3() {
      (this->* funcArray[0])();
   }
};
```

Compiler Error C2065

Error Message
'identifier' : undeclared identifier

A variable's type must be specified in a declaration before it can be used. The parameters that a function uses must be specified in a declaration, or prototype, before the function can be used.

Possible causes:

1. You are compiling with a debug version of the C runtime, declaring a Standard C++ Library iterator variable in a **for** loop, and then trying to use that iterator variable outside the scope of the **for** loop. Compiling Standard C++ Library code with a debug version of the C runtime implies /Zc:forScope. See Debug Iterator Support for more information.
2. You may be calling a function in an SDK header file that is currently not supported in your build environment.
3. Omitting necessary include files, especially if you define **VC_EXTRALEAN**, **WIN32_LEAN_AND_MEAN**, or **WIN32_EXTRA_LEAN**. These symbols exclude some header files from windows.h and afxv_w32.h to speed compiles. (Look in windows.h and afxv_w32.h for an up-to-date description of what's excluded.)
4. Identifier name is misspelled.
5. Identifier uses the wrong uppercase and lowercase letters.
6. Missing closing quote after a string constant.
7. Improper namespace scope. To resolve ANSI C++ Standard Library functions and operators, for example, you must specify the `std` namespace with the **using** directive. The following example fails to compile because the **using** directive is commented out and **cout** is defined in the `std` namespace:

Example

The following sample generates C2065.

```
// C2065.cpp
// compile with: /EHsc
// using namespace std;
#include <iostream>
int main() {
   cout << "Hello" << endl;   // C2065

   // try the following line instead
   std::cout << "Hello" << std::endl;
}
```

When calling a generic function, if the intended type argument cannot be deduced from the parameters used, the compiler will report an error. For more information, see Generic Functions.

The following sample generates C2065.

```
// C2065_b.cpp
// compile with: /clr
generic <typename ItemType>
void G(int i) {}

int main() {
   // global generic function call
   G<T>(10);   // C2065
   G<int>(10);   // OK
}
```

This error can also be generated as a result of compiler conformance work that was done for Visual C++ 2005: parameter checking for Visual C++ attributes. See Breaking Changes in the Visual C++ 2005 Compiler for more information.

The following sample generates C2065.

```
// C2065_c.cpp
// compile with: /c
[module(DLL, name=MyLibrary)];   // C2065
// try the following line instead
// [module(dll, name="MyLibrary")];

[export]
struct MyStruct {
   int i;
};
```

Compiler Error C2066

Error Message
cast to function type is illegal

In ANSI C, it is not legal to cast between a function pointer and a data pointer.

Compiler Error C2067

Error Message
cast to array type is illegal

An object was cast to an array type.

Compiler Error C2069

Error Message

cast of 'void' term to non-'void'

Type **void** cannot be cast to any other type.

Compiler Error C2070

Error Message
'type': illegal sizeof operand

The sizeof operator requires an expression or type name.

The following sample generates C2070:

```
// C2070.cpp
void func() {}
int main() {
   int a;
   a = sizeof(func);   // C2070
}
```

Possible resolution:

```
// C2070b.cpp
void func() {}
int main() {
   int a;
   a = sizeof(a);
}
```

Compiler Error C2071

Error Message
'identifier' : illegal storage class

identifier was declared with an invalid storage class.

Example
The following sample generates C2071.

```
// C2071.cpp
// compile with: /c
struct C {
   extern int i;   // C2071
};
struct D {
   int i;   // OK
};
```

The following sample generates C2071.

```
// C2071_b.cpp
// compile with: /c
typedef int x(int i) { return i; }   // C2071
typedef int x;   // OK
```

Compiler Error C2072

Error Message
'identifier' : initialization of a function

A function initializer was specified incorrectly.

Compiler Error C2073

Error Message
'identifier' : elements of partially initialized array must have a default constructor

Too few initializers were specified for an array of user-defined types or constants. If an explicit initializer and its corresponding constructor are not specified for an array member, a default constructor must be supplied.

The following sample generates C2073:

```
// C2073.cpp
class A {
public:
   A( int );   // constructor for ints only
};
A a[3] = { A(1), A(2) };   // C2073, no default constructor
```

```
// C2073b.cpp
// compile with: /c
class B {
public:
   B();   // default constructor declared
   B( int );
};
B b[3] = { B(1), B(2) };   // OK
```

Compiler Error C2074

Error Message
'identifier' : 'class-key' initialization needs curly braces

There were no curly braces around the specified class, structure, or union initializer.

Compiler Error C2075

Error Message
'identifier' : array initialization needs curly braces

There were no curly braces around the specified array initializer.

The following sample generates C2075:

```
// C2075.c
int main() {
   int i[] = 1, 2, 3 };   // C2075
}
```

Possible resolution:

```
// C2075b.c
int main() {
   int j[] = { 1, 2, 3 };
}
```

Compiler Error C2077

Error Message
non-scalar field initializer 'identifier'

You tried to initialize a bit field with a nonscalar (struct, union, array, or class). Use an integer or floating-point value.

Compiler Error C2078

Error Message
too many initializers

The number of initializers exceeds the number of objects to be initialized.

The following sample generates C2078:

```
// C2078.cpp
int main() {
   int d[2] = {1, 2, 3};   // C2078
   int e[2] = {1, 2};   // OK

   char a[]={"a", "b"};   // C2078
   char *b[]={"a", "b"};   // OK
   char c[]={'a', 'b'};   // OK
}
```

Compiler Error C2079

Error Message
'identifier' uses undefined class/struct/union 'name'

The specified identifier is an undefined class, structure, or union.

This error can be caused by initializing an anonymous union.

The following sample generates C2079:

```
// C2079.cpp
// compile with: /EHsc
#include <iostream>
int main() {
   std::ifstream g;   // C2079
}
```

Possible resolution:

```
// C2079b.cpp
// compile with: /EHsc
#include <fstream>
int main( ) {
   std::ifstream g;
}
```

C2079 can also occur if you attempt to declare an object on the stack of a type whose forward declaration is only in scope.

```
// C2079c.cpp
class A;

class B {
   A a;   // C2079
};

class A {};
```

Possible resolution:

```
// C2079d.cpp
// compile with: /c
class A;
class C {};

class B {
   A * a;
   C c;
};

class A {};
```

Compiler Error C2081

Error Message
'identifier' : name in formal parameter list illegal

The identifier caused a syntax error.

This error can be caused by using the old style for the formal parameter list. You must specify the type of formal parameters in the formal parameter list.

The following sample generates C2081:

```
// C2081.c
void func( int i, j ) {}  // C2081, no type specified for j
```

Possible resolution:

```
// C2081b.c
// compile with: /c
void func( int i, int j ) {}
```

Compiler Error C2082

Error Message
redefinition of formal parameter 'identifier'

A formal parameter to a function is redeclared within the function body. To resolve the error, remove the redefinition.

The following sample generates C2082:

```
// C2082.cpp
void func(int i) {
   int i;   // C2082
   int ii;   // OK
}
```

Compiler Error C2083

Error Message
struct/union comparison illegal

A structure or union is compared directly with another user-defined type. This is not allowed unless a comparison operator has been defined or a conversion to a scalar type exists.

Compiler Error C2084

Error Message
function 'function' already has a body

The function has already been defined.

In previous versions of Visual C++,

- The compiler would accept multiple template specializations that resolved to the same actual type, although the additional definitions would never be available. The compiler will now detect these multiple definitions
- __int32 and int were treated as separate types. The compiler now treats __int32 as a synonym for int. This means that the compiler will detect multiple definitions if a function is overloaded on both __int32 and int and give an error.

The following sample generates C2084:

```
// C2084.cpp
void Func(int);
void Func(int) {}   // define function
void Func(int) {}   // C2084 second definition
```

Possible resolution:

```
// C2084b.cpp
// compile with: /c
void Func(int);
void Func(int) {}
```

Compiler Error C2085

Error Message
'identifier' : not in formal parameter list

The identifier was declared in a function definition but not in the formal parameter list. (ANSI C only)

The following sample generates C2085:

```
// C2085.c
void func1( void )
int main( void ) {}   // C2085
```

Possible resolution:

```
// C2085b.c
void func1( void );
int main( void ) {}
```

With the semicolon missing, `func1()` looks like a function definition, not a prototype, so `main` is defined within `func1()`, generating Error C2085 for identifier `main`.

Compiler Error C2086

Error Message
'identifier' : redefinition

The identifier is defined more than once, or a subsequent declaration differs from a previous one.

C2086 can also be the result of incremental building for a referenced C# assembly. Rebuild the C# assembly to resolve this error.

The following sample generates C2086:

```
// C2086.cpp
main() {
  int a;
  int a;   // C2086 not an error in ANSI C
}
```

Compiler Error C2087

Error Message
'identifier' : missing subscript

The definition of an array with multiple subscripts is missing a subscript value for a dimension higher than one.

The following sample generates C2087:

```
// C2087.cpp
int main() {
   char a[10][];   // C2087
}
```

Possible resolution:

```
// C2087b.cpp
int main() {
   char b[4][5];
}
```

Compiler Error C2088

Error Message
'operator' : illegal for 'class-key'

The operator was not defined for the structure or union. This error is only valid for C code.

The following sample generates C2088 three times:

```
// C2088.c
struct S {
   int m_i;
} s;

int main() {
   int i = s * 1;   // C2088
   struct S s2 = +s;   // C2088
   s++;   // C2088
}
```

Compiler Error C2089

Error Message
'identifier' : 'class-key' too large

The specified structure or union exceeds the 4GB limit.

Compiler Error C2090

Error Message
function returns array

A function cannot return an array. Return a pointer to an array instead.

The following sample generates C2090:

```
// C2090.cpp
int func1(void)[] {}   // C2090
```

Possible resolution:

```
// C2090b.cpp
// compile with: /c
int* func2(int * i) {
   return i;
}
```

Compiler Error C2091

Error Message
function returns function

A function cannot return a function. Return a pointer to a function instead.

Compiler Error C2092

Error Message
'array name' array element type cannot be function

Arrays of functions are not allowed. Use an array of pointers to functions.

Example
The following sample generates C2092:

```
// C2092.cpp
typedef void (F) ();
typedef F AT[10];   // C2092
```

Possible resolution:

```
// C2092b.cpp
// compile with: /c
typedef void (F) ();
typedef F * AT[10];
```

Compiler Error C2093

Error Message
'variable1' : cannot be initialized using address of automatic variable 'variable2'

When compiling with /Za, the program tried to use the address of an automatic variable as an initializer.

The following sample generates C2093:

```
// C2093.c
// compile with: /Za /c
void func() {
   int li;   // an implicit auto variable
   int * s[]= { &li };   // C2093 address is not a constant

   // OK
   static int li2;
   int * s2[]= { &li2 };
}
```

Compiler Error C2094

Error Message
label 'identifier' was undefined

The label used by a goto statement does not exist in the function.

The following sample generates C2094:

```
// C2094.c
int main() {
   goto test;
}   // C2094
```

Possible resolution:

```
// C2094b.c
int main() {
   goto test;
   test:
   {
   }
}
```

Compiler Error C2095

Error Message
'function' : actual parameter has type 'void' : parameter 'number'

The parameter passed to the function is type **void**, which is not allowed. Use a pointer to void (**void ***) instead.

The *number* indicates which parameter is **void**.

Compiler Error C2097

Error Message
illegal initialization

To fix by checking the following possible causes

1. Initialization of a variable using a nonconstant value.
2. Initialization of a short address with a long address.
3. Initialization of a local structure, union, or array with a nonconstant expression when compiling with **/Za**.
4. Initialization with an expression containing a comma operator.
5. Initialization with an expression that is neither constant nor symbolic.

Compiler Error C2099

Error Message
initializer is not a constant

This error is issued only by the C compiler and occurs only for non-automatic variables. The compiler initializes non-automatic variables at the start of the program and the values they are initialized with must be constant.

Example

The following sample generates C2099.

```
// C2099.c
int j;
int *p;
j = *p;   // C2099 *p is not a constant
```

C2099 can also occur because the compiler is not able to perform constant folding on an expression under **/fp:strict** because the floating point precision environment settings (see _controlfp_s for more information) may differ from compile to run time.

When constant folding fails, the compiler invokes dynamic initialization, which is not allowed in C.

To resolve this error, compile the module as a .cpp file or simplify the expression.

For more information, see /fp (Specify Floating-Point Behavior).

The following sample generates C2099.

```
// C2099_2.c
// compile with: /fp:strict /c
float X = 2.0 - 1.0;   // C2099
float X2 = 1.0;   // OK
```

Compiler Errors C2100 through C2199

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, type the error number in the **Look for** box in the Index, or click a link below.

Compiler Error C2100

Compiler Error C2101

Compiler Error C2102

Compiler Error C2103

Compiler Error C2104

Compiler Error C2105

Compiler Error C2106

Compiler Error C2107

Compiler Error C2108

Compiler Error C2109

Compiler Error C2110

Compiler Error C2111

Compiler Error C2112

Compiler Error C2113

Compiler Error C2114

Compiler Error C2115

Compiler Error C2116

Compiler Error C2117

Compiler Error C2118

Compiler Error C2120

Compiler Error C2121

Compiler Error C2122

Compiler Error C2124

Compiler Error C2125

Compiler Error C2126

Compiler Error C2127

Compiler Error C2128

Compiler Error C2129

Compiler Error C2130

Compiler Error C2132

Compiler Error C2133

Compiler Error C2134

Compiler Error C2135

Compiler Error C2137

Compiler Error C2138

Compiler Error C2139
Compiler Error C2140
Compiler Error C2141
Compiler Error C2142
Compiler Error C2143
Compiler Error C2144
Compiler Error C2145
Compiler Error C2146
Compiler Error C2147
Compiler Error C2148
Compiler Error C2149
Compiler Error C2150
Compiler Error C2151
Compiler Error C2152
Compiler Error C2153
Compiler Error C2154
Compiler Error C2155
Compiler Error C2156
Compiler Error C2157
Compiler Error C2158
Compiler Error C2159
Compiler Error C2160
Compiler Error C2161
Compiler Error C2162
Compiler Error C2163
Compiler Error C2164
Compiler Error C2165
Compiler Error C2166
Compiler Error C2167
Compiler Error C2168
Compiler Error C2169
Compiler Error C2170
Compiler Error C2171
Compiler Error C2172
Compiler Error C2173
Compiler Error C2174
Compiler Error C2175
Compiler Error C2177
Compiler Error C2179
Compiler Error C2180

Compiler Error C2181
Compiler Error C2182
Compiler Error C2183
Compiler Error C2184
Compiler Error C2185
Compiler Error C2186
Compiler Error C2188
Compiler Error C2189
Compiler Error C2190
Compiler Error C2191
Compiler Error C2192
Compiler Error C2193
Compiler Error C2194
Compiler Error C2195
Compiler Error C2196
Compiler Error C2197
Compiler Error C2198
Compiler Error C2199

Compiler Error C2100

Error Message
illegal indirection

Indirection operator (*****) is applied to a nonpointer value.

The following sample generates C2100:

```
// C2100.cpp
int main() {
   int r = 0, *s = 0;
   s = &r;
   *r = 200;   // C2100
   *s = 200;   // OK
}
```

Compiler Error C2101

Error Message
'&' on constant

The address-of operator (**&**) must have an l-value as operand.

The following sample generates C2101:

```
// C2101.cpp
int main() {
   char test;
   test = &'a';   // C2101
   test = 'a';   // OK
}
```

Compiler Error C2102

Error Message
'&' requires l-value

The address-of operator (**&**) must have an l-value as operand.

Compiler Error C2103

Error Message
'&' on register variable

You cannot take the address of a register.

Compiler Error C2104

Error Message
'&' on bit field ignored

You cannot take the address of a bit field.

The following sample generates C2104:

```
// C2104.cpp
struct X {
   int sb : 1;
};

int main() {
   X x;
   &x.sb;   // C2104
   x.sb;   // OK
}
```

Compiler Error C2105

Error Message
'operator' needs l-value

The operator must have an l-value as operand.

The following sample generates C2105:

```
// C2105.cpp
int main() {
   unsigned char * p1 = 0;
   unsigned int * p2 = (unsigned int *)p1;
   p2++;

   unsigned int * p = 0;
   p++;   // ok

   p2 = (unsigned int *)p1;
   ((unsigned int *)p1)++;   // C2105
}
```

The following sample generates C2105:

```
// C2105b.cpp
int main() {
   int a[10] = {0};
   int b[10] = {0};
   ++(a , b);   // C2105

   // OK
   ++(a[0] , b[0]);
   ++a[0];
}
```

Compiler Error C2106

Error Message
'operator' : left operand must be l-value

The operator must have an l-value as its left operand.

The following sample generates C2106:

```
// C2106.cpp
int main() {
   int a;
   1 = a;   // C2106
   a = 1;   // OK
}
```

Compiler Error C2107

Error Message
illegal index, indirection not allowed

A subscript is applied to an expression that does not evaluate to a pointer.

Example

C2107 can occur if you incorrectly use the **this** pointer of a value type to access the type's default indexer. For more information, see Semantics of the this Pointer in Value and Reference Types.

The following sample generates C2107.

```
// C2107.cpp
// compile with: /clr
using namespace System;

value struct B {
   property String ^ default[String ^] {
      String ^ get(String ^ data) {
         return "abc";
      }
   }
   void Test() {
      Console::WriteLine("{0}", this["aa"]);   // C2107
      Console::WriteLine("{0}", this->default["aa"]);   // OK
   }
};

int main() {
   B ^ myb = gcnew B();
   myb->Test();
}
```

Compiler Error C2108

Error Message
subscript is not of integral type

The array subscript is a noninteger expression.

Example

C2107 can occur if you incorrectly use the **this** pointer of a value type to access the type's default indexer. For more information, see Semantics of the this Pointer in Value and Reference Types.

The following sample generates C2107.

```
// C2108.cpp
// compile with: /clr
using namespace System;

value struct B {
   property Double default[Double] {
      Double get(Double data) {
         return data*data;
      }
   }
   void Test() {
      Console::WriteLine("{0}", this[3.3]);   // C2108
      Console::WriteLine("{0}", this->default[3.3]);   // OK
   }
};

int main() {
   B ^ myb = gcnew B();
   myb->Test();
}
```

Compiler Error C2109

Error Message
subscript requires array or pointer type

The subscript was used on a variable that was not an array.

The following sample generates C2109:

```
// C2109.cpp
int main() {
   int a, b[10] = {0};
   a[0] = 1;   // C2109
   b[0] = 1;   // OK
}
```

Compiler Error C2110

Error Message
'+' : cannot add two pointers

An attempt was made to add two pointer values using the plus (**+**) operator.

The following sample generates C2110:

```
// C2110.cpp
int main() {
   int a = 0;
   int *pa;
   int *pb;
   a = pa + pb;   // C2110
}
```

Compiler Error C2111

Error Message

'+' : pointer addition requires integral operand

An attempt was made to add a nonintegral value to a pointer using the plus (**+**) operator.

The following sample generates C2111:

```
// C2111.cpp
int main() {
   int *a = 0, *pa = 0, b = 0;
   double d = 0.00;

   a = pa + d;   // C2111
   a = pa + b;   // OK
}
```

Compiler Error C2112

Error Message

'-' : pointer subtraction requires integral or pointer operand

An attempt was made to subtract pointers that point to different types.

Compiler Error C2113

Error Message

'-' : pointer can only be subtracted from another pointer

The right operand in a subtraction operation was a pointer, but the left operand was not.

Compiler Error C2114

Error Message

'operator' : pointer on left; needs integral value on right

The left operand of *operator* was a pointer, so the right operand must be an integer value.

Compiler Error C2115

Error Message
'identifier' : incompatible types

An expression contained incompatible types.

Compiler Error C2116

Error Message
function parameter lists differed

The parameters in the default parameter list do not match the formal parameter list.

Compiler Error C2117

Error Message
'identifier' : array bounds overflow

An array has too many initializers:

- Array elements and initializers do not match in size and quantity.
- No space for the null terminator in a string.

The following sample generates C2117:

```
// C2117.cpp
int main() {
   char abc[4] = "abcd";   // C2117
   char def[4] = "abd";   // OK
}
```

Compiler Error C2118

Error Message
negative subscript

The value defining the array size is larger than the maximum array size or smaller than zero.

The following sample generates C2118:

```
// C2118.cpp
int main() {
   int array1[-1];   // C2118
   int array2[3];   // OK
}
```

Compiler Error C2120

Error Message
'void' illegal with all types

The **void** type is used in a declaration with another type.

The following sample generates C2120:

```
// C2120.cpp
int main() {
   void int i;   // C2120
   int j;   // OK
}
```

Compiler Error C2121

Error Message

'#' : invalid character : possibly the result of a macro expansion

An invalid # character may have been inserted by an incorrect macro that uses the token-pasting operator (##) instead of the stringizing operator (#).

Compiler Error C2122

Error Message

'identifier' : prototype parameter in name list illegal

The parameter is not a legal type. ANSI C does not support user-defined types.

Compiler Error C2124

Error Message
divide or mod by zero

A constant expression has a zero denominator. To resolve the error, do not divide by zero.

The following sample generates C2124:

```
// C2124.cpp
int main() {
  int i = 1 / 0;   // C2124  do not divide by zero
  int i2 = 12 / 2;   // OK
}
```

Compiler Error C2125

Error Message
'identifier' : allocation exceeds 64K

The given item exceeds the 64K size limit.

Compiler Error C2126

Error Message
'operator' : incorrect operand

The operator (usually ++ or --) was incorrectly applied to an enumeration.

Compiler Error C2127

Error Message
parameter allocation exceeds 32K

The storage space for parameters to a function exceeds the 32K limit.

Compiler Error C2128

Error Message
'function' : alloc_text/same_seg applicable only to functions with C linkage

pragma alloc_text can only be used with functions declared to have C linkage.

The following sample generates C2128:

```
// C2128.cpp
// compile with: /c

// Delete the following line to resolve.
void func();
// #pragma alloc_text("my segment", func)   // C2128

extern "C" {
void func();
}

#pragma alloc_text("my segment", func)
void func() {}
```

Compiler Error C2129

Error Message
static function 'function' declared but not defined

A forward reference is made to a **static** function that is never defined.

A **static** function must be defined within file scope. If the function is defined in another file, it must be declared **extern**.

Compiler Error C2130

Error Message
#line expected a string containing the filename, found 'token'

The optional file name token following #line *linenumber* must be a string.

The following sample generates C2130:

```
// C2130.cpp
int main() {
   #line 1000 test   // C2130
   #line 1000 "test"   // OK
}
```

Compiler Error C2132

Error Message

syntax error : unexpected identifier

An identifier appears in an unsupported context.

Compiler Error C2133

Error Message
'identifier' : unknown size

An unsized array is declared as a member of a class, structure, union, or enumeration. The /Za (ANSI C) option does not allow unsized member arrays.

The following sample generates C2133:

```
// C2133.cpp
// compile with: /Za
struct X {
   int a[0];   // C2133 unsized array
};
```

Possible resolution:

```
// C2133b.cpp
// compile with: /c
struct X {
   int a[0];   // no /Za
};
```

Compiler Error C2134

Error Message
'identifier' : struct/union too large

The size of a structure or union exceeds the 64K limit.

Compiler Error C2135

Error Message
'bit operator' : illegal bit field operation

The address-of operator (**&**) cannot be applied to a bit field.

The following sample generates C2135:

```
// C2135.cpp
struct S {
   int i : 1;
};

struct T {
   int j;
};
int main() {
   &S::i;   // C2135 address of a bit field
   &T::j;   // OK
}
```

Compiler Error C2137

Error Message
empty character constant

The empty character constant (' ') is not permitted.

The following sample generates C2137:

```
// C2137.cpp
int main() {
   char c = '';   // C2137
   char d = ' ';   // OK
}
```

Compiler Error C2138

Error Message
illegal to define an enumeration without any members

An enumeration must have at least one member when /Za (disable Microsoft extensions) is selected.

Compiler Error C2139

Error Message

'type' : an undefined class is not allowed as an argument to compiler intrinsic type trait 'trait'

An invalid argument was passed to a type trait.

For more information, see Compiler Support for Type Traits.

Example

The following sample generates C2139.

```
// C2139.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
struct is_polymorphic {
   static const bool value = __is_polymorphic(T);
};

class C;
class D {};

class E {
public:
   virtual void Test() {}
};

int main() {
   cout << is_polymorphic<C>::value << endl;   // C2139
   cout << is_polymorphic<D>::value << endl;   // OK
   cout << is_polymorphic<E>::value << endl;   // OK
}
```

Compiler Error C2140

Error Message

'type' : a type that is dependent on a generic type parameter is not allowed as an argument to compiler intrinsic type trait 'trait'

An invalid type specifier was passed to a type trait.

For more information, see Compiler Support for Type Traits.

Example

The following sample generates C2140.

```
// C2140.cpp
// compile with: /clr /c
template <class T>

struct is_polymorphic {
   static const bool value = __is_polymorphic(T);
};

class x {};

generic <class T>
ref class C {
   void f() {
      System::Console::WriteLine(__is_polymorphic(T));   // C2140
      System::Console::WriteLine(is_polymorphic<T>::value);   // C2140

      System::Console::WriteLine(__is_polymorphic(x));   // OK
      System::Console::WriteLine(is_polymorphic<x>::value);   // OK
   }
};
```

Compiler Error C2141

Error Message
array size overflow

An array exceeds the 2GB limit. Reduce the size of the array.

Example
The following sample generates C2141.

```
// C2141.cpp
// processor: IPF
class A {
   short m_n;
};

int main()
{
   A* pA = (A*)(-1);
   pA = new A[0x8000000000000001];   // C2141

   A* pA2 = (A*)(-1);
   pA2 = new A[0x80000000000000F];   // OK
}
```

Compiler Error C2142

Error Message
function declarations differ, variable parameters specified only in one of them

One declaration of the function contains a variable parameter list. Another declaration does not. ANSI C (/Za) only.

The following sample generates C2142:

```
// C2142.c
// compile with: /Za /c
void func();
void func( int, ... );   // C2142
void func2( int, ... );   // OK
```

Compiler Error C2143

Error Message
syntax error : missing 'token1' before 'token2'

The compiler expected a specific token (a language element other than white space) and found another token instead. C2143 can occur in different situations.

See Knowledge Base article Q241706 for information on this error when using a function-try block.

Check the C++ Language Reference to determine where code is syntactically incorrect. Because the compiler may report this error after the line that causes the problem, check the few lines of code that precede the error.

When using **/clr** and a **using** directive has a syntax error:

```
// C2143a.cpp
// compile with: /clr /c
using namespace System.Reflection;   // C2143
using namespace System::Reflection;
```

When attempting to compile a source code file using CLR syntax without using **/clr**:

```
// C2143b.cpp
ref struct A {   // C2143 error compile with /clr
   void Test() {}
};

int main() {
   A a;
   a.Test();
}
```

The first non-whitespace character following an **if** statement should be a left parenthesis. The compiler cannot translate anything else:

```
// C2143c.cpp
int main() {
   int j = 0;

   // OK
   if (j < 25)
      ;

   if (j < 25)   // C2143
}
```

Missing closing brace, parenthesis, or semicolon on the line where the error is detected or a few lines above:

```
// C2143d.cpp
// compile with: /c
class X {
   int member1;
   int member2   // C2143
} x;
```

Invalid tag in a class declaration:

```
// C2143e.cpp
class X {
   int member;
} x;
```

```
class + {};   // C2143 + is an invalid tag name
class ValidName {};   // OK
```

A label not attached to a statement. If you must place a label by itself (for example, at the end of a compound statement), attach it to a null statement:

```
// C2143f.cpp
// compile with: /c
void func1() {
   // OK
   end1:
      ;

   end2:   // C2143
}
```

An unqualified call is made to a type in the Standard C++ Library:

```
// C2143g.cpp
// compile with: /EHsc /c
#include <vector>
static vector<char> bad;   // C2143
static std::vector<char> good;   // OK
```

Missing **typename** keyword:

```
// C2143h.cpp
template <typename T>
struct X {
   struct Y {
      int i;
   };
   Y memFunc();
};
template <typename T>
X<T>::Y X<T>::memFunc() {   // C2143
// try the following line instead
// typename X<T>::Y X<T>::memFunc() {
   return Y();
}
```

Attempting to define an explicit instantiation:

```
// C2143i.cpp
// compile with: /EHsc /c
// template definition
template <class T>
void PrintType(T i, T j) {}

template void PrintType(float i, float j){}   // C2143
template void PrintType(float i, float j);   // OK
```

C2143 can also occur due to the way the compiler handles undefined types in a constructor's parameter list. If the compiler encounters an undefined (misspelled?) type, it assumes the constructor is an expression, and issues C2143. To resolve, only use defined types in a constructor parameter list. The following sample generates C20143:

```
// C2143j.cpp
// compile with: /c
class DataReceived {};
class DataSent {};

struct MyDataController {
   MyDataController(DataSent* u, DataRecieved* j) {}   // C2143
```

```
    // try the following line instead
    // MyDataController(DataSent* u, DataReceived* j) {}   // OK
};
```

Compiler Error C2144

Error Message
syntax error : 'type' should be preceded by 'token'

The compiler expected *token* and found *type* instead.

This error may be caused by a missing closing brace, right parenthesis, or semicolon.

C2144 can also occur when attempting to create a macro from a CLR keyword that contains a white space character.

You may also see C2144 if you are trying to do type forwarding. See Type Forwarding for more information.

Example
The following sample generates C2144.

```
// C2144.cpp
// compile with: /clr /c
#define REF ref
REF struct MyStruct0;   // C2144

// OK
#define REF1 ref struct
REF1 MyStruct1;
```

The following sample generates C2144.

```
// C2144_2.cpp
// compile with: /clr /c
ref struct X {

   property double MultiDimProp[,,] {   // C2144
   // try the following line instead
   // property double MultiDimProp[int , int, int] {
      double get(int, int, int) { return 1; }
      void set(int i, int j, int k, double l) {}
   }

   property double MultiDimProp2[] {   // C2144
   // try the following line instead
   // property double MultiDimProp2[int] {
      double get(int) { return 1; }
      void set(int i, double l) {}
   }
};
```

Compiler Error C2145

Error Message
syntax error : missing 'token' before identifier

The compiler expected *token* and found identifier instead.

This error may be caused by a missing semicolon after the last declaration in a block.

Compiler Error C2146

Error Message
syntax error : missing 'token' before identifier 'identifier'

The compiler expected *token* and found *identifier* instead. Possible causes:

1. Spelling or capitalization error.
2. Missing type specifier in the declaration of the identifier.

This error may be caused by a typographical error. Error C2065 usually precedes this error.

Example

The following sample generates C2146.

```
// C2146.cpp
class CDeclaredClass {};

class CMyClass {
   CUndeclared m_myClass;   // C2146
   CDeclaredClass m_myClass2;   // OK
};

int main() {
   int x;
   int t x;   // C2146 : missing semicolon before 'x'
}
```

This error can also be generated as a result of compiler conformance work that was done for Visual Studio .NET 2003: missing **typename** keyword. See Summary of Compile-Time Breaking Changes for more information.

The following sample compiles in Visual Studio .NET 2002 but will fail in Visual Studio .NET 2003:

```
// C2146b.cpp
// compile with: /c
template <typename T>
struct X {
   struct Y {
      int i;
   };
   Y memFunc();
};

template <typename T>
X<T>::Y func() { }   // C2146

// OK
template <typename T>
typename X<T>::Y func() { }
```

You will also see this error as a result of compiler conformance work that was done for Visual Studio .NET 2003: explicit specializations no longer find template parameters from primary template.

The use of `T` from the primary template is not allowed in the explicit specialization. For code to be valid in the Visual Studio .NET 2003 and Visual Studio .NET versions of Visual C++, replace all instances of the template parameter in the specialization with the explicitly specialized type.

See Summary of Compile-Time Breaking Changes for more information.

The following sample compiles in Visual Studio .NET but will fail in Visual Studio .NET 2003:

```
// C2146_c.cpp
```

```
// compile with: /c
template <class T>
struct S;

template <>
struct S<int> {
   T m_t;   // C2146
   int m_t2;   // OK
};
```

Compiler Error C2147

Error Message
syntax error : 'identifier' is a new keyword

An identifier was used that is now a reserved keyword in the language.

The following sample generates C2147:

```
// C2147.cpp
// compile with: /clr
int main() {
   int gcnew = 0;   // C2147
   int i = 0;   // OK
}
```

Compiler Error C2148

Error Message
total size of array must not exceed 0x7fffffff bytes

An array exceeds the limit. Reduce the size of the array.

Example
The following sample generates C2148:

```
// C2148.cpp
#include <stdio.h>
#include <stdlib.h>

int main( ) {
   char MyArray[0x7fffffff];   // C2148
   char * MyArray2 = (char *)malloc(0x7fffffff);

   if (MyArray2)
      printf_s("It worked!");
   else
      printf_s("It didn't work.");
}
```

Compiler Error C2149

Error Message
'identifier' : named bit field cannot have zero width

Bit fields can have zero width only if unnamed.

The following sample generates C2149:

```
// C2149.cpp
// compile with: /c
struct C {
   int i : 0;   // C2149
   int j : 2;   // OK
};
```

Compiler Error C2150

Error Message
'identifier' : bit field must have type 'int', 'signed int', or 'unsigned int'

bit fields are required to be **int**, **signed int**, or **unsigned int**.

The following sample generates C2150:

```
// C2150.cpp
// compile with: /c
struct A {
   float a : 8;    // C2150
   int i : 8;   // OK
};
```

Compiler Error C2151

Error Message
more than one language attribute

A function has more than one keyword (**__cdecl**, **__stdcall**, or **__fastcall**) specifying a calling convention.

Compiler Error C2152

Error Message

'identifier' : pointers to functions with different attributes

A pointer to a function with one calling convention (**__cdecl**, **__stdcall**, or **__fastcall**) is assigned to a pointer to a function with another calling convention.

Compiler Error C2153

Error Message
hex constants must have at least one hex digit

Hexadecimal constants 0x, 0X, and \x are not valid. At least one hex digit must follow x or X.

The following sample generates C2153:

```
// C2153.cpp
int main() {
   int a= 0x;    // C2153
   int b= 0xA;   // OK
}
```

Compiler Error C2154

Error Message

'segment' : does not refer to a segment name

When you declare a based variable, you must allocate a segment, unless the variable is **extern** and uninitialized.

Compiler Error C2155

Error Message

'?' : invalid left operand, expected arithmetic or pointer type

An expression on the left hand side of ? cannot be compared to zero. You must use an arithmetic or pointer expression that can be compared to zero.

Compiler Error C2156

Error Message
pragma must be outside function

A pragma that must be specified at a global level (outside a function body) is within a function.

The following sample generates C2156:

```
// C2156.cpp
#pragma optimize( "l", on )   // OK
int main() {
   #pragma optimize( "l", on )   // C2156
}
```

Compiler Error C2157

Error Message
'function' : must be declared before use in pragma list

The function name is not declared before being referenced in the list of functions for an alloc_text pragma.

The following sample generates C2157:

```
// C2157.cpp
// compile with: /c
#pragma alloc_text( "func", func)   // C2157

// OK
extern "C" void func();
#pragma alloc_text( "func", func)
```

Compiler Error C2158

Error Message

'type' : #pragma make_public directive is currently supported for native non-template types only

The make_public pragma can only be applied to a native, non-template type.

Example

The following sample generates C2158.

```
// C2158.cpp
// compile with: /clr /c
ref class A {};
#pragma make_public(A)   // C2158

template< typename T >
class B {};
#pragma make_public(B)   // C2158
#pragma make_public(B<int>)   // C2158

void C () {}
#pragma make_public(C)   // C2158

class D {};
#pragma make_public(D)   // OK
```

Compiler Error C2159

Error Message
more than one storage class specified

A declaration contains more than one storage class.

The following sample generates C2159:

```
// C2159.cpp
// compile with: /c
static int i;   // OK
extern static int i;   // C2159
```

Compiler Error C2160

Error Message
'##' cannot occur at the beginning of a macro definition

A macro definition began with a token-pasting operator (##).

The following sample generates C2160:

```
// C2160.cpp
// compile with: /c
#define mac(a,b) #a   // OK
#define mac(a,b) ##a   // C2160
```

Compiler Error C2161

Error Message
'##' cannot occur at the end of a macro definition

A macro definition ended with a token-pasting operator (##).

The following sample generates C2161:

```
// C2161.cpp
// compile with: /c
#define mac(a,b) a   // OK
#define mac(a,b) a##   // C2161
```

Compiler Error C2162

Error Message
expected macro formal parameter

The token following a stringizing operator (#) is not a formal parameter name.

Example
The following sample generates C2162:

```
// C2162.cpp
// compile with: /c
#include <stdio.h>

#define print(a) printf_s(b)   // OK
#define print(a) printf_s(#b)    // C2162
```

Compiler Error C2163

Error Message
'function' : not available as an intrinsic function

An **intrinsic** or **function** pragma lists a function not available in intrinsic form. For example, certain intrinsics are not available when compiling a program that uses /clr programming.

Compiler Error C2164

Error Message
'function' : intrinsic function not declared

An **intrinsic** pragma uses an undeclared function (only occurs with **/Oi**). Or, one of the compiler intrinsics was used without including its header file.

The following sample generates C2164:

```
// C2164.c
// compile with: /c
// processor: x86
// Uncomment the following line to resolve.
// #include "xmmintrin.h"
void b(float *p) {
   _mm_load_ss(p);   // C2164
}
```

Compiler Error C2165

Error Message
'keyword' : cannot modify pointers to data

The **__stdcall**, **__cdecl**, or **__fastcall** keyword attempts to modify a pointer to data.

The following sample generates C2165:

```
// C2165.cpp
// compile with: /c
char __cdecl *p;   // C2165
char *p;   // OK
```

Compiler Error C2166

Error Message
l-value specifies const object

Code attempts to modify an item declared **const**.

The following sample generates C2166:

```
// C2166.cpp
int f();
int main() {
   ( (const int&) 1 ) = 5;   // C2166
}
```

Compiler Error C2167

Error Message

'function' : too many actual parameters for intrinsic function

A reference to an **intrinsic** function has too many parameters.

Compiler Error C2168

Error Message

'function' : too few actual parameters for intrinsic function

A reference to an **intrinsic** function has too few parameters.

Compiler Error C2169

Error Message

'function' : intrinsic function, cannot be defined

A function definition appears for a function already declared **intrinsic**.

Compiler Error C2170

Error Message
'identifier' : not declared as a function, cannot be intrinsic

To fix by checking the following possible causes

1. Pragma **intrinsic** is used for an item other than a function.
2. Pragma **intrinsic** is used for a function with no intrinsic form.

Compiler Error C2171

Error Message
'operator' : illegal on operands of type 'type'

A unary operator is used with an invalid operand type.

Example
The following sample generates C2171.

```
// C2171.cpp
int main() {
   double d, d1;
   d = ~d1;   // C2171

   // OK
   int d2 = 0, d3 = 0;
   d2 = ~d3;
}
```

The following sample generates C2171.

```
// C2171_b.cpp
// compile with: /c
class A {
public:
   A() { STF( &A::D ); }

   void D() {}
   void DTF() {
      (*TF)();   // C2171
      (this->*TF)();   // OK
   }

   void STF(void (A::*fnc)()) {
      TF = fnc;
   }

private:
   void (A::*TF)();
};
```

Compiler Error C2172

Error Message

'function' : actual parameter is not a pointer : parameter number

Parameter *number* is not a pointer. The function expects a pointer.

Compiler Error C2173

Error Message

'function' : actual parameter is not a pointer : parameter number1, parameter list number2

Parameter *number1* passed to parameter list *number2* is not a pointer. The function expects a pointer.

Compiler Error C2174

Error Message

'function' : actual parameter has type 'void' : parameter number1, parameter list number2

Parameter *number1* passed to parameter list *number2* is a **void** parameter. Parameters cannot have type **void**. Use **void*** instead.

Compiler Error C2175

Error Message
'locale' : invalid locale

The specified locale is not valid. See Language and Country/Region Strings in the *Run-Time Library Reference* for supported locales.

Compiler Error C2177

Error Message
constant too big

A constant value is too large for the variable type it is assigned.

The following sample generates C2177:

```
// C2177.cpp
int main() {
   int a=18446744073709551616;   // C2177
   int b=18446744073709551615;   // OK
}
```

Compiler Error C2179

Error Message
'type' : an attribute argument cannot use type parameters

A generic type parameter is resolved at runtime. However, an attribute parameter must be resolved at compile time. Therefore, you cannot use a generic type parameter as an argument to an attribute.

Example
The following sample generates C2179.

```
// C2179.cpp
// compile with: /clr
using namespace System;

public ref struct Attr : Attribute {
   Attr(Type ^ a) {
      x = a;
   }

   Type ^ x;
};

ref struct G {};

generic<typename T>
public ref class Z {
public:
   Type ^ d;
   [Attr(T::typeid)]   // C2179
   // try the following line instead
   // [Attr(G::typeid)]
   T t;
};
```

Compiler Error C2180

Error Message
controlling expression has type 'type'

The controlling expression in an **if**, **while**, **for**, or **do** statement is a function with return type **void** or an expression cast to **void**.

The following sample generates C2180:

```
// C2180.c
void test() {}
int test2() {
   return 0;
}

int main() {
   if (test()) ;   // C2180
   if (test2()) ;   // OK
}
```

C2180 can also be generated when nullptr is used as the expression in a control statement.

The following sample generates C2180:

```
// C2180b.cpp
// compile with: /clr
int main() {
   int i = 0;
   if (nullptr) {}   // C2180
   if (i == 0) {}   // OK
}
```

Compiler Error C2181

Error Message
illegal else without matching if

Each **else** must have a matching **if**.

The following sample generates C2181:

```
// C2181.cpp
int main() {
   int i = 0;
   else   // C2181
      i = 1;
}
```

Possible resolution:

```
// C2181b.cpp
int main() {
   int i = 0;
   if(i)
      i = 0;
   else
      i = 1;
}
```

Compiler Error C2182

Error Message
'identifier' : illegal use of type 'void'

A variable is declared type **void**.

The following sample generates C2182:

```
// C2182.cpp
// compile with: /c
int main() {
   int i = 10;
   void &ir = i;   // C2182 cannot have a reference to type void
   int &ir = i;   // OK
}
```

Compiler Error C2183

Error Message

syntax error: translation unit is empty

Preprocessing produced an empty source file.

Compiler Error C2184

Error Message
'type' : illegal type for __except expression, must be an integral

A type was used in an __except statement, but the type is not allowed.

The following sample generates C2184:

```
// C2184.cpp
void f() {
   int * p;
   __try{}
   __except(p){};   // C2184
}
```

Possible resolution:

```
// C2184b.cpp
// compile with: /c
void f() {
   int i = 0;
   __try{}
   __except(i){};
}
```

Compiler Error C2185

Error Message
'identifier' : illegal based allocation

A register variable or automatic (local) variable is declared **__based**. Only global variables can be declared **__based**.

Compiler Error C2186

Error Message
'operator' : illegal operand of type 'void'

The operator has a **void** operand.

The following sample generates C2186:

```
// C2186.cpp
// compile with: /c
void func1( void );
int  func2( void );
int i = 2 + func1();   // C2186 func1() is type void
int j = 2 + func2();   // OK both operands are type int
```

Compiler Error C2188

Error Message

'number' : too big for wide character

The number exceeds the size limit for the wide-character type. Choose a larger type.

Compiler Error C2189

Error Message
#error : string

The compiler encountered an **#error** directive and output the user-specified *string*.

Compiler Error C2190

Error Message
first parameter list longer than second

A C function was declared a second time with a shorter parameter list. C does not support overloaded functions.

The following sample generates C2190:

```
// C2190.c
// compile with: /Za /c
void func( int, float );
void func( int  );   // C2190, different parameter list
void func2( int  );   // OK
```

Compiler Error C2191

Error Message
second parameter list longer than first

A C function was declared a second time with a longer parameter list. C does not support overloaded functions.

Example
The following sample generates C2191:

```
// C2191.c
// compile with: /Za /c
void func( int );
void func( int, float );   // C2191 different parameter list
void func2( int, float );   // OK
```

Compiler Error C2192

Error Message
parameter 'number' declaration different

A C function was declared a second time with a different parameter list. C does not support overloaded functions.

The following sample generates C2192:

```
// C2192.c
// compile with: /Za /c
void func( float, int );
void func( int, float );   // C2192, different parameter list
void func2( int, float );   // OK
```

Compiler Error C2193

Error Message
'identifier' : already in a segment

A function was placed in two different segments using **alloc_text** and **code_seg** pragmas.

The following sample generates C2193:

```
// C2193.cpp
// compile with: /c
extern "C" void MYFUNCTION();
#pragma alloc_text(MYCODE, MYFUNCTION)
#pragma code_seg("MYCODE2")
extern "C" void MYFUNCTION() {}   // C2193
extern "C" void MYFUNCTION2() {}
```

Compiler Error C2194

Error Message
'identifier' : is a text segment

The **data_seg** pragma uses a segment name used with **code_seg**.

The following sample generates C2194:

```
// C2194.cpp
// compile with: /c
#pragma code_seg("MYCODE")
#pragma data_seg("MYCODE")   // C2194
#pragma data_seg("MYCODE2")   // OK
```

Compiler Error C2195

Error Message
'identifier' : is a data segment

The **code_seg** pragma uses a segment name used with the **data_seg** pragma.

The following sample generates C2195:

```
// C2195.cpp
#pragma data_seg("MYDATA")
#pragma code_seg("MYDATA")   // C2195
#pragma code_seg("MYDATA2")   // OK
```

Compiler Error C2196

Error Message
case value 'value' already used.

A switch statement uses the same case value more than once.

The following sample generates C2196:

```
// C2196.cpp
int main() {
   int i = 0;
   switch( i ) {
   case 0:
      break;
   case 0:   // C2196
   // try the following line instead
   // case 1:
      break;
   }
}
```

Compiler Error C2197

Error Message
'function' : too many arguments for call

The compiler detected too many parameters for a call to the function, or an incorrect function declaration.

The following sample generates C2197:

```
// C2197.c
// compile with: /Za /c
void func( int );
int main() {
   func( 1, 2 );   // C2197 two actual parameters
   func( 2 );   // OK
}
```

Compiler Error C2198

Error Message
'function' : too few arguments for call

The compiler found too few parameters for a call to the function, or an incorrect function declaration.

The following sample generates C2198:

```
// C2198.c
// compile with: /c
void func( int, int );
int main() {
   func( 1 );   // C2198 only one actual parameter
   func( 1, 1 );   // OK
}
```

Compiler Error C2199

Error Message
syntax error : found 'identifier (' at global scope (was a declaration intended?)

The specified context caused a syntax error. There may be incorrect declaration syntax.

The following sample generates C2199:

```
// C2199.cpp
// compile with: /c
int j = int(1) int(1);   // C2199
int j = 1;   // OK
```

Compiler Errors C2200 through C2299

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, type the error number (C2213, for example) in the index, or click a link below.

Compiler Error C2200

Compiler Error C2201

Compiler Error C2203

Compiler Error C2204

Compiler Error C2205

Compiler Error C2206

Compiler Error C2207

Compiler Error C2208

Compiler Error C2212

Compiler Error C2213

Compiler Error C2216

Compiler Error C2217

Compiler Error C2218

Compiler Error C2219

Compiler Error C2220

Compiler Error C2221

Compiler Error C2222

Compiler Error C2223

Compiler Error C2224

Compiler Error C2226

Compiler Error C2227

Compiler Error C2228

Compiler Error C2229

Compiler Error C2231

Compiler Error C2232

Compiler Error C2233

Compiler Error C2234

Compiler Error C2236

Compiler Error C2238

Compiler Error C2241

Compiler Error C2242

Compiler Error C2243

Compiler Error C2244

Compiler Error C2245

Compiler Error C2246

Compiler Error C2247
Compiler Error C2248
Compiler Error C2249
Compiler Error C2250
Compiler Error C2251
Compiler Error C2252
Compiler Error C2253
Compiler Error C2254
Compiler Error C2255
Compiler Error C2256
Compiler Error C2258
Compiler Error C2259
Compiler Error C2261
Compiler Error C2262
Compiler Error C2264
Compiler Error C2265
Compiler Error C2266
Compiler Error C2267
Compiler Error C2268
Compiler Error C2270
Compiler Error C2271
Compiler Error C2272
Compiler Error C2273
Compiler Error C2274
Compiler Error C2275
Compiler Error C2276
Compiler Error C2277
Compiler Error C2279
Compiler Error C2283
Compiler Error C2284
Compiler Error C2285
Compiler Error C2286
Compiler Error C2287
Compiler Error C2289
Compiler Error C2290
Compiler Error C2292
Compiler Error C2293
Compiler Error C2295
Compiler Error C2296
Compiler Error C2297

Compiler Error C2298

Compiler Error C2299

Compiler Error C2200

Error Message
'function' : function has already been defined

An **alloc_text** pragma uses a function name already defined.

Compiler Error C2201

Error Message
'identifier' : must have external linkage in order to be exported/imported

The exported identifier is **static**.

The following sample generates C2286:

```
// C2201.cpp
// compile with: /c
__declspec(dllexport) static void func() {}   // C2201 func() is static
__declspec(dllexport) void func2() {}   // OK
```

See Also
Reference
Types of Linkage

Compiler Error C2203

Error Message
delete operator cannot specify bounds for an array

With the **/Za** (ANSI) option, the **delete** operator can delete an entire array but not parts or specific members of the array.

The following sample generates C2203:

```
// C2203.cpp
// compile with: /Za
int main() {
   int *ar = new int[10];
   delete [4] ar;   // C2203
   // try the following line instead
   // delete [] ar;
}
```

Compiler Error C2204

Error Message

'type' : type definition found within parentheses

The type is defined as an operand or in prototype scope.

Compiler Error C2205

Error Message
'identifier' : cannot initialize extern variables with block scope

An **extern** variable cannot be initialized in a function.

Compiler Error C2206

Error Message
'function' : typedef cannot be used for function definition

A **typedef** is used to define a function type.

The following sample generates C2206:

```
// C2206.cpp
typedef int functyp();
typedef int MyInt;
functyp func1 {};   // C2206
int main() {
   MyInt i = 0;   // OK
}
```

Compiler Error C2207

Error Message
'member' in struct/union 'tag' has a zero-sized array

The *member* of the structure or union *tag* contains an array with no subscript or a zero subscript. This kind of array is legal only as the last member of a structure or union.

Compiler Error C2208

Error Message
'type' : no members defined using this type

An identifier resolving to a type name is in an aggregate declaration, but the compiler cannot declare a member.

The following sample generates C2208:

```
// C2208.cpp
class C {
   C;   // C2208
   C(){}   // OK
};
```

Compiler Error C2212

Error Message
'identifier' : __based not available for pointers to functions

Pointers to functions cannot be declared **__based**. If you need code-based data, use the **__declspec** keyword or the **data_seg** pragma.

Compiler Error C2213

Error Message
'modifier' : illegal argument to __based

The argument modifying **__based** is invalid.

The following sample generates C2213:

```
// C2213.cpp
// compile with: /c
int i;
int *j;
char __based(i) *p;   // C2213
char __based(j) *p2;   // OK
```

Compiler Error C2216

Error Message
'keyword1' cannot be used with ' keyword2'

Two keywords that are mutually exclusive were used together.

Example
The following sample generates C2216.

```
// C2216.cpp
// compile with: /clr /c
ref struct Y1 {
   literal
   static int staticConst2 = 10;   // C2216
};
```

The following sample generates C2216.

```
// C2216b.cpp
// compile with: /clr /c
public ref class X {
   extern property int i { int get(); }   // C2216 extern not allowed on property
   typedef property int i2;   // C2216 typedef not allowed on property
};
```

The following sample generates C2216.

```
// C2216c.cpp
// compile with: /clr /c
public interface struct I {
   double f();
   double g();
   double h();
};

public ref struct R : I {
   virtual double f() new override { return 0.0; }   // C2216
   virtual double g() new { return 0.0; }   // OK
   virtual double h() override { return 0.0; }   // OK
};
```

Compiler Error C2217

Error Message
'attribute1' requires 'attribute2'

The first function attribute requires the second attribute.

To fix by checking the following possible causes

1. Interrupt (**__interrupt**) function declared as **near**. Interrupt functions must be **far**.
2. Interrupt function declared with **__stdcall**, or **__fastcall**. Interrupt functions must use C calling conventions.

Example

C2217 can also occur if you attempt to bind a delegate to a CLR function that takes a variable number of arguments. If the function also has e param array overload, use that instead. The following sample generates C2217.

```
// C2217.cpp
// compile with: /clr
using namespace System;
delegate void MyDel(String^, Object^, Object^, ...);   // C2217
delegate void MyDel2(String ^, array<Object ^> ^);   // OK

int main() {
   MyDel2^ wl = gcnew MyDel2(Console::WriteLine);
   array<Object ^ > ^ x = gcnew array<Object ^>(2);
   x[0] = safe_cast<Object^>(0);
   x[1] = safe_cast<Object^>(1);

   // wl("{0}, {1}", 0, 1);
   wl("{0}, {1}", x);
}
```

Compiler Error C2218

Error Message
type in __based construct must be 'void'

Only types based on **void** are legal.

Compiler Error C2219

Error Message

syntax error : type qualifier must be after '*'

Type qualifier (**const** or **volatile**) appears where it is not permitted.

Compiler Error C2220

Error Message
warning treated as error - no object file generated

/WX tells the compiler to treat all warnings as errors. Since an error occurred, no object or executable file was generated.

To fix by using the following possible solutions

1. Fix the problem that caused the warning.
2. Compile at a lower warning level.
3. Compile without **/WX**.

Compiler Error C2221

Error Message

unexpected type 'type': must be a native class/struct/union type

A function or macro expected a type.

Compiler Error C2222

Error Message

'->' : left operand has struct/union type, use '.'

The left operand is a **class**, **struct**, or **union** rather than a pointer. To access a member, you must use the period (.) operator rather than the –> operator.

Compiler Error C2223

Error Message
left of '->identifier' must point to struct/union

The operand to the left of `->` is not a pointer to a class, structure, or union.

This error can be caused by a left operand that is an undefined variable (therefore type **int**).

Compiler Error C2224

Error Message
left of '.identifier' must have struct/union type

The operand to the left of the period (.) is not a class, structure, or union.

This error can be caused by a left operand that is an undefined variable (therefore type **int**).

Compiler Error C2226

Error Message

syntax error : unexpected type 'type'

A syntax error occurs before or in the type specifier.

Compiler Error C2227

Error Message
left of '->member' must point to class/struct/union/generic type

The operand to the left of `->` is not a pointer to a class, structure, or union.

The following sample generates C2227:

```
// C2227.cpp
int *pInt;
struct S {
public:
    int member;
} s, *pS = &s;

int main() {
   pInt->member = 0;   // C2227 pInt points to an int
   pS->member = 0;   // OK
}
```

Compiler Error C2228

Error Message
left of '.identifier' must have class/struct/union

The operand to the left of the period (.) is not a class, structure, or union.

The following sample generates C2228:

```
// C2228.cpp
int i;
struct S {
public:
    int member;
} s, *ps = &s;

int main() {
   i.member = 0;   // C2228 i is not a class type
   ps.member = 0;  // C2228 ps is a pointer to a structure

   s.member = 0;   // s is a structure type
   ps->member = 0; // ps points to a structure S
}
```

You will also see this error if you use incorrect syntax when using Managed Extensions. Whereas in other Visual Studio languages, you can use the dot operator to access a member of a managed class, a pointer to the object in C++ means you have to use the -> operator to access the member:

Wrong: `String * myString = checkedListBox1->CheckedItems->Item[0].ToString();`

Right: `String * myString = checkedListBox1->CheckedItems->Item[0]->ToString();`

Compiler Error C2229

Error Message
type 'identifier' has an illegal zero-sized array

A member of a structure or bit field contains a zero-sized array that is not the last member.

Because you can have a zero sized array as the last member of the struct, you must specify its size when you allocate the struct.

If the zero sized array is not the last member of the struct, the compiler can't calculate the offset for the remaining fields.

The following sample generates C2229:

```
// C2229.cpp
struct S {
   int a[0];  // C2229  zero-sized array
   int b[1];
};

struct S2 {
   int a;
   int b[0];
};

int main() {
   // allocate 7 elements for b field
   S2* s2 = (S2*)new int[sizeof(S2) + 7*sizeof(int)];
   s2->b[6] = 100;
}
```

Compiler Error C2231

Error Message

'.' : left operand points to 'class-key', use '–>'

The operand to the left of the member-selection operation (.) is a pointer instead of a class, structure, or union.

The following sample generates C2231:

```
// C2231.c
struct S {
   int member;
} s, *ps = &s;
int main() {
   ps.member = 0;   // C2231

   // OK
   ps->member = 0;   // crash
   s.member = 0;
}
```

Compiler Error C2232

Error Message
'–>' : left operand has 'class-key' type, use '.'

The operand to the left of the –> operator is not a pointer. Use the period (.) operator for a class, structure, or union.

The following sample generates C2232:

```
// C2232.c
struct X {
    int member;
} x, *px;
int main() {
    x->member = 0;   // C2232, x is not a pointer

    px->member = 0;
    x.member = 0;
}
```

Compiler Error C2233

Error Message
'identifier' : arrays of objects containing zero-size arrays are illegal

Each object in an array must contain at least one element.

The following sample generates C2233:

```
// C2233.cpp
// compile with: /c
class A {
   char somearray[1];
};

class B {
   char zeroarray[];
};

A array[100];   // OK
B array2[100];   // C2233
```

Compiler Error C2234

Error Message
'name' : arrays of references are illegal

Because pointers to references are not allowed, arrays of references are not possible.

The following sample generates C2234:

```
// C2234.cpp
int main() {
   int i = 0, j = 0, k = 0, l = 0;
   int &array[4] = {i,j,k,l};   // C2234
   int array2[4] = {i,j,k,l};   // OK
}
```

Compiler Error C2236

Error Message
unexpected 'class-key' 'identifier''. Did you forget a ';'?

The identifier is already defined as a type and cannot be overridden by a user-defined type.

The following sample generates C2236:

```
// C2236.cpp
// compile with: /c
int class A {};  // C2236 "int class" is unexpected
int i;
class B {};
```

Compiler Error C2238

Error Message
unexpected token(s) preceding 'token'

An incorrect token was found.

The following sample generates C2238:

```
// C2238.cpp
// compile with: /c
class v {
virtual: int vvv;   // C2238
};
```

Compiler Error C2241

Error Message
'identifier' : member access is restricted

Code attempts to access a private or protected member.

To fix by using the following possible solutions

1. Change the access level of the member.
2. Declare the member a **friend** of the accessing function.

Compiler Error C2242

Error Message
typedef name cannot follow class/struct/union

A **typedef** name appears at the end of a qualified name.

Compiler Error C2243

Error Message
'conversion type' conversion from 'type1' to 'type2' exists, but is inaccessible

Access protection (**protected** or **private**) prevented conversion from a pointer to a derived class to a pointer to the base class.

The following sample generates C2243:

```
// C2243.cpp
// compile with: /c
class B {};
class D : private B {};
class E : public B {};

D d;
B *p = &d;   // C2243

E e;
B *p2 = &e;
```

Compiler Error C2244

Error Message
'identifier' : unable to match function definition to an existing declaration

An unusual use of the unary + operator was used in front of a function call that did not have parenthesis.

This error only occurs in C++ projects.

The following sample generates C2244:

```
// C2244.cpp
int func(char) {
   return 0;
}

int func(int) {
   return 0;
}

int main() {
   +func;   // C2244
}
```

C2244 can also occur when an incorrect function signature is used for a member function of a class template.

```
// C2244b.cpp
// compile with: /c
template<class T>
class XYZ {
   void func(T t);
};

template<class T>
void XYZ<T>::func(int i) {}   // C2244 wrong function signature
// try the following line instead
// void XYZ<T>::func(T t) {}
```

C2244 can also occur when an incorrect function signature is used for a member function template.

```
// C2244c.cpp
// compile with: /c
class ABC {
   template<class T>
   void func(int i, T t);
};

template<class T>
void ABC::func(int i) {}   // C2244 wrong signature
// try the following line instead
// void ABC::func(int i, T t) {}
```

You cannot partially specialize a function template.

```
// C2244d.cpp
template<class T, class U>
class QRS {
   void func(T t, U u);
};

template<class T>
void QRS<T,int>::func(T t, int u) {}   // C2244
```

Compiler Error C2245

Error Message
non-existent member function 'function' specified as friend (member function signature does not match any overload)

A function specified as a friend was not found by the compiler.

The following sample generates C2245:

```
// C2245.cpp
// compile with: /c
class B {
   void f(int i);
};

class A {
   int m_i;
   friend void B::f(char);   // C2245
   // try the following line instead
   // friend void B::f(int);
};

void B::f(int i) {
   A a;
   a.m_i = 0;
}
```

Compiler Error C2246

Error Message
'identifier' : illegal static data member in locally defined class

A member of a class, structure, or union with local scope is declared **static**.

The following sample generates C2246:

```
// C2246.cpp
// compile with: /c
void func( void ) {
   class A { static int i; };   // C2246  i is local to func
   static int j;   // OK
};
```

Compiler Error C2247

Error Message
'identifier' not accessible because 'class' uses 'specifier' to inherit from 'class'

identifier is inherited from a class declared with private or protected access.

The following sample generates C2247:

```
// C2247.cpp
class A {
public:
   int i;
};
class B : private A {};    // B inherits a private A
class C : public B {} c;   // so even though C's B is public
int j = c.i;               // C2247, i not accessible
```

This error can also be generated as a result of compiler conformance work that was done for Visual Studio .NET 2003: access control with protected members. A protected member (n) can only be accessed through a member function of a class (B) that inherits from the class (A) of which it (n) is a member.

See Summary of Compile-Time Breaking Changes for more information.

For code that is valid in both the Visual Studio .NET 2003 and Visual Studio .NET versions of Visual C++, declare the member to be a friend of the type. Public inheritance could also be used.

```
// C2247b.cpp
// compile with: /c
// C2247 expected
class A {
public:
   void f();
   int n;
};

class B: protected A {
   // Uncomment the following line to resolve.
   // friend void A::f();
};

void A::f() {
   B b;
   b.n;
}
```

C2247 can also be generated as a result of compiler conformance work that was done for Visual Studio .NET 2003: private base classes now inaccessible. A class (A) that is a private base class to a type (B) should not be accessible to a type (C) that uses B as a base class.

See Summary of Compile-Time Breaking Changes for more information.

For code that is valid in both the Visual Studio .NET 2003 and Visual Studio .NET versions of Visual C++, use the scope operator.

```
// C2247c.cpp
// compile with: /c
struct A {};

struct B: private A {};

struct C : B {
   void f() {
      A *p1 = (A*) this;   // C2247
```

```
        // try the following line instead
        // ::A *p2 = (::A*) this;
    }
};
```

Compiler Error C2248

Error Message
'member' : cannot access 'access' member declared in class 'class'

Members of a derived class cannot access **private** members of a base class. You cannot access **private** or **protected** members of class instances.

See Knowledge Base article KB243351 for more information on C2248.

The following sample generates C2248:

```
// C2248.cpp
#include <stdio.h>
class X {
public:
   int  m_pubMemb;
   void setPrivMemb( int i ) {
      m_privMemb = i;
      printf_s("\n%d", m_privMemb);
   }
protected:
   int  m_protMemb;

private:
   int  m_privMemb;
} x;

int main() {
   x.m_pubMemb = 4;
   printf_s("\n%d", x.m_pubMemb);
   x.m_protMemb = 2;   // C2248 m_protMemb is protected
   x.m_privMemb = 3;   // C2248  m_privMemb is private
   x.setPrivMemb(0);   // OK uses public access function
}
```

Another conformance issue that exposes C2248 is the use of template friends and specialization. For more information, see Linker Tools Error LNK2019.

```
// C2248_b.cpp
template<class T>
void f(T t) {
   t.i;   // C2248
}

struct S {
private:
   int i;

public:
   S() {}
   // Delete the following line to resolve.
   friend void f(S);   // refer to the non-template function void f(S)

   // Uncomment the following line to resolve.
   // friend void f<S>(S);
};

int main() {
   S s;
   f<S>(s);
}
```

Another conformance issue that exposes C2248 is when you attempt to declare a friend of a class and when the class is not

visible to the friend declaration in the scope of the class. In this case, grant friendship to the enclosing class to resolve the error. For more information, see Breaking Changes in the Visual C++ 2005 Compiler.

```
// C2248_c.cpp
// compile with: /c
class T {
   class S {
      class E {};
   };
   friend class S::E;   // C2248
};

class A {
   class S {
      class E {};
      friend class A;   // grant friendship to enclosing class
   };
   friend class S::E;   // OK
};
```

Compiler Error C2249

Error Message
'member' : no accessible path to access member declared in virtual base 'class'

The *member* is inherited from a nonpublic **virtual** base class or structure.

Example
The following sample generates C2249.

```
// C2249.cpp
class A {
private:
   void privFunc( void ) {};
public:
   void pubFunc( void ) {};
};

class B : virtual public A {} b;

int main() {
   b.privFunc();    // C2249, private member of A
   b.pubFunc();    // OK
}
```

C2249 can also occur if you try to assign a stream from the Standard C++ Library to another stream. The following sample generates C2249.

```
// C2249_2.cpp
#include <iostream>
using namespace std;
int main() {
   cout = cerr;   // C2249
   #define cout cerr;   // OK
}
```

Compiler Error C2250

Error Message
'identifier' : ambiguous inheritance of 'class::member'

The derived class inherits more than one override of a virtual function of a virtual base class. These overrides are ambiguous in the derived class.

The following sample generates C2286:

```
// C2250.cpp
// compile with: /c
// C2250 expected
struct V {
   virtual void vf();
};

struct A : virtual V {
   void vf();
};

struct B : virtual V {
   void vf();
};

struct D : A, B {
   // Uncomment the following line to resolve.
   // void vf();
};
```

Compiler Error C2251

Error Message
namespace 'namespace' does not have a member 'member' - Did you mean 'member'?

The compiler was not able to find an identifier in the specified namespace.

The following sample generates C2251:

```
// C2251.cpp
// compile with: /c
namespace A {
   namespace B {
      void f1();
   }

   using namespace B;
}

void A::f1() {}   // C2251
void A::B::f1() {}   // OK
```

Compiler Error C2252

Error Message
cannot explicitly instantiate template in current scope

The compiler detected a problem with an explicit instantiation of a template. For example, you cannot explicitly instantiate a template in a function.

The following sample generates C2252:

```
// C2252.cpp
class A {
public:
   template <class T>
   int getit(int i , T * it ) {
      return i;
   }
   template int A::getit<double>(int i, double * it);   // C2252
   // try the following line instead
   // template <> int A::getit<double>(int i, double * it);

};

int main() {
   // cannot explicitly instantiate in function
   template int A::getit<long>(int i, long * it);   // C2252
}
```

Compiler Error C2253

Error Message
'function' : pure specifier or abstract override specifier only allowed on virtual function

A nonvirtual function is specified as pure **virtual**.

The following sample generates C2253:

```
// C2253.cpp
// compile with: /c
class A {
public:
   void func1() = 0;   // C2253 not virtual
   virtual void func2() = 0;   // OK
};
```

The following sample generates C2253:

```
// C2253_2.cpp
// compile with: /clr /c
ref struct A {
   property int Prop_3 {
      int get() abstract;   // C2253
      // try the following line instead
      // virtual int get() abstract;

      void set(int i) abstract;   // C2253
      // try the following line instead
      // virtual void set(int i) abstract;
   }
};
```

Compiler Error C2254

Error Message
'function' : pure specifier or abstract override specifier not allowed on friend function

A **friend** function is specified as pure **virtual**.

The following sample generates C2254:

```
// C2254.cpp
// compile with: /c
class A {
public:
   friend void func1() = 0;   // C2254, func1 is friend
   void virtual func2() = 0;   // OK, pure virtual
   friend void func3();   // OK, friend not virtual nor pure
};

void func1() {};
void func3() {};
```

Compiler Error C2255

Error Message
'element' : not allowed outside of a class definition

For example, a nonmember function is declared a **friend**.

The following sample generates C2255:

```
// C2255.cpp
// compile with: /c
class A {
private:
   void func1();
   friend void func2();
};

friend void func1() {}   // C2255
void func2(){}
```

Compiler Error C2256

Error Message
illegal use of friend specifier on 'function'

A destructor or constructor cannot be specified as a friend.

The following sample generates C2256:

```
// C2256.cpp
// compile with: /c
class C {
public:
   friend ~C();   // C2256
   ~C();   // OK
};
```

Compiler Error C2258

Error Message
illegal pure syntax, must be '= 0'

A pure virtual function is declared with incorrect syntax.

The following sample generates C2258:

```
// C2258.cpp
// compile with: /c
class A {
public:
   void virtual func1() = 1; // C2258
   void virtual func2() = 0;   // OK
};
```

Compiler Error C2259

Error Message
'class' : cannot instantiate abstract class

Code declares an instance of an abstract class or structure.

You cannot instantiate a class or structure with one or more pure virtual functions. To instantiate objects of a derived class, the derived class must override each pure virtual function.

For more information, see Implicitly Abstract Classes.

The following sample generates C2259:

```
// C2259.cpp
// compile with: /c
class V {
public:
   void virtual func() = 0;
};
class A : public V {};
class B : public V {
public:
   void func();
};
V v;  // C2259, V is an abstract class
A a;  // C2259, A inherits func() as pure virtual
B b;  // OK, B defines func()
```

Whenever you derive from an interface and implement the interface methods in the derived class with access permissions other than public, you may receive C2259. This occurs because the compiler expects the interface methods implemented in the derived class to have public access. When you implement the member functions for an interface with more restrictive access permissions, the compiler does not consider them to be implementations for the interface methods defined in the interface, which in turn makes the derived class an abstract class.

There are two possible workarounds for the problem:

- Make the access permissions public for the implemented methods.
- Use the scope resolution operator for the interface methods implemented in the derived class to qualify the implemented method name with the name of the interface.

C2259 can also occur as a result of conformance work that was done in Visual C++ 2005, **/Zc:wchar_t** is now on by default. In this situation, C2599 can be resolved either by compiling with **/Zc:wchar_t-**, to get the behavior from previous versions, or preferably, by updating your types so they are compatible. For more information, see
Breaking Changes in the Visual C++ 2005 Compiler and /Zc:wchar_t (wchar_t Is Native Type).

The following sample generates C2259:

```
// C2259b.cpp
// compile with: /c
#include <windows.h>

class MyClass {
public:
   // WCHAR now typedef'ed to wchar_t
   virtual void func(WCHAR*) = 0;
};

class MyClass2 : MyClass {
public:
   void func(unsigned short*);
};

MyClass2 x;   // C2259
```

```
// OK
class MyClass3 {
public:
   virtual void func(WCHAR*) = 0;
   virtual void func2(wchar_t*) = 0;
   virtual void func3(unsigned short*) = 0;
};

class MyClass4 : MyClass3 {
public:
   void func(WCHAR*) {}
   void func2(wchar_t*) {}
   void func3(unsigned short*) {}
};

MyClass4 y;
```

The following sample generates C2259:

```
// C2259c.cpp
// compile with: /clr
interface class MyInterface {
   void MyMethod();
};

ref class MyDerivedClass: public MyInterface {
private:
   // Uncomment the following line to resolve.
   // public:
   void MyMethod(){}
   // or the following line
   // void MyInterface::MyMethod() {};
};

int main() {
   MyDerivedClass^ instance = gcnew MyDerivedClass; // C2259
}
```

The following sample generates C2259:

```
// C2259d.cpp
// compile with: /clr:oldSyntax
public __gc __interface MyInterface {
   void MyMethod();
};

__gc class MyDerivedClass : public MyInterface {
// Uncomment the following line to resolve.
// public:
   void MyMethod() {};
   // or the following line
   // void MyInterface::MyMethod() {};
};

int main() {
   MyDerivedClass *instance = new MyDerivedClass();   // C2259
}
```

Compiler Error C2261

Error Message
'string' : assembly reference is invalid and cannot be resolved

A value was not valid.

InternalsVisibleToAttribute is used to specify a friend assembly. For example, if a.dll wants to specify b.dll as a friend assembly, you would specify (in a.dll): InternalsVisibleTo("b"). The runtime then allows b.dll to access everything in a.dll (except private types).

For more on the correct syntax when specifying friend assemblies, see Friend Assemblies.

Example
The following sample generates C2261.

```
// C2261.cpp
// compile with: /clr /c
using namespace System::Runtime::CompilerServices;
[assembly: InternalsVisibleTo("a,a,a")];   // C2261
[assembly: InternalsVisibleTo("a.a")];   // OK
[assembly: InternalsVisibleTo("a")];   // OK
```

Compiler Error C2262

Error Message

'attribute_specifiers' : InternalsVisibleTo declarations cannot have a version, culture, or processor architecture specified

The InternalsVisibleToAttribute attribute was not specified correctly.

Example

The following sample generates C2262.

```
// C2262.cpp
// compile with: /clr /c
using namespace System::Runtime::CompilerServices;
[assembly: InternalsVisibleTo("assembly_name, version=1.2.3.7")];   // C2262
[assembly: InternalsVisibleTo("assembly_name ")];   // OK
```

Compiler Error C2264

Error Message
'function' : error in function definition or declaration; function not called

The function cannot be called due to an incorrect definition or declaration.

The following sample generates C2264:

```
// C2264.cpp
struct C {
   // Delete the following line to resolve.
   operator int(int = 0){}   // incorrect declaration
};

struct D {
   operator int(){return 0;}   // OK
};

int main() {
   int i;

   C c;
   i = c;   // C2264

   D d;
   i = d;   // OK
}
```

Compiler Error C2265

Error Message

'identifier' : reference to a zero-sized array is illegal

A reference is declared for an array with zero size. The array must have nonzero, constant bounds.

Compiler Error C2266

Error Message

'identifier' : reference to a non-constant bounded array is illegal

A reference is declared for an array with a nonconstant bound. The array must have constant bounds.

Compiler Error C2267

Error Message
'function' : static functions with block scope are illegal

A local function is declared **static**. Static functions must have global scope.

The following sample generates C2267:

```
// C2267.cpp
static int func2();   // OK
int main() {
    static int func1();   // C2267
}
```

Compiler Error C2268

Error Message
'function' is a compiler predefined library helper. Library helpers are not supported with /GL; compile object file 'file' without /GL.

A function defined in your source code has the same name as an internal compiler function. Compile the module containing the function without /GL.

The following sample generates C2268:

```
// C2268.c
// compile with: /c
// processor: x86
extern int SHFusionLoadLibrary(int lpLibFileName);

int __cdecl _except_handler3(void) {
   return SHFusionLoadLibrary(0);
}

extern int main(void);

void* mainCRTStartup(void* p) {
   p = main;
   return p;
}
```

and then:

```
// C2268b.c
// compile with: C2268.c /EHsc /GL /Ob0 /O2 /Fa /GS- /link /nodefaultlib
// processor: x86
extern int SHFusionLoadLibrary(int lpLibFileName);

extern int __cdecl _except_handler3(void);
extern void mainCRTStartup(void*);
int g = 2;

#define ENTERCONTEXT(fail) \
   int ulCookie = 0;\
   if (!SHActivateContext(&ulCookie)) \
      return fail;\
   __try {

#define LEAVECONTEXT \
    } __finally {SHDeactivateContext(ulCookie);}

int SHActivateContext(int* a) {
    return *a == g || !*a ||_except_handler3();
}

void SHDeactivateContext(int a) {
    g = a;
}

int SHFusionLoadLibrary(int lpLibFileName) {   // C2268
    ENTERCONTEXT(0)
    g = lpLibFileName;
    LEAVECONTEXT

    return lpLibFileName;
}

int main(void) {
```

```
    g = SHFusionLoadLibrary(10);
    return 0;
}
```

Compiler Error C2270

Error Message
'function' : modifiers not allowed on nonmember functions

A nonmember function is declared with const, volatile, or another memory-model modifier.

The following sample generates C2270:

```
// C2270.cpp
// compile with: /c
void func1(void) const;   // C2270, nonmember function

void func2(void);

class CMyClass {
public:
   void func2(void) const;
};
```

Compiler Error C2271

Error Message
'operator' : new/delete cannot have formal list modifiers

The operator (**new** or **delete**) is declared with a memory-model specifier.

The following sample generates C2271:

```
// C2271.cpp
// compile with: /c
void* operator new(size_t) const {   // C2271
// try the following line instead
// void* operator new(size_t) {
   return 0;
}

struct X {
   static void* operator new(size_t) const;   // C2271
   // try the following line instead
   // void * X::operator new(size_t) const;   // static member operator new
};
```

Compiler Error C2272

Error Message
'function' : modifiers not allowed on static member functions

A **static** member function is declared with a memory-model specifier, such as const or volatile, and such modifiers are not allowed on **static** member functions.

The following sample generates C2272:

```
// C2272.cpp
// compile with: /c
class CMyClass {
public:
   static void func1() const volatile;   // C2272  func1 is static
   void func2() const volatile;   // OK
};
```

Compiler Error C2273

Error Message
'type' : illegal as right side of '->' operator

A type appears as the right operand of a –> operator.

This error can be caused by trying to access a user-defined type conversion. Use the keyword **operator** between -> and *type*.

The following sample generates C2273:

```
// C2273.cpp
struct MyClass {
   operator int() {
      return 0;
   }
};
int main() {
   MyClass * ClassPtr = new MyClass;
   int i = ClassPtr->int();   // C2273
   int j = ClassPtr-> operator int();   // OK
}
```

Compiler Error C2274

Error Message
'type' : illegal as right side of '.' operator

A type appears as the right operand of a member-access (.) operator.

This error can be caused by trying to access a user-defined type conversion. Use the keyword **operator** between the period and *type*.

The following sample generates C2286:

```
// C2274.cpp
struct MyClass {
   operator int() {
      return 0;
   }
};

int main() {
   MyClass ClassName;
   int i = ClassName.int();   // C2274
   int j = ClassName.operator int();   // OK
}
```

Compiler Error C2275

Error Message
'identifier' : illegal use of this type as an expression

An expression uses the –> operator with a **typedef** identifier.

The following sample generates C2275:

```
// C2275.cpp
typedef struct S {
    int mem;
} *S_t;
void func1( int *parm );
void func2() {
    func1( &S_t->mem );   // C2275, S_t is a typedef
}
```

Compiler Error C2276

Error Message
'operator' : illegal operation on bound member function expression

The compiler found a problem with the syntax to create a pointer-to-member.

The following sample generates C2276:

```
// C2276.cpp
class A {
public:
   int func(){return 0;}
} a;

int (*pf)() = &a.func;   // C2276
// try the following line instead
// int (A::*pf3)() = &A::func;

class B {
public:
   void mf() {
      &this -> mf;   // C2276
      // try the following line instead
      // &B::mf;
   }
};

int main() {
   A a;
   &a.func;   // C2276
   // try the following line instead
   // &A::func;
}
```

Compiler Error C2277

Error Message
'identifier' : cannot take address of this member function

You cannot take the address of a member function.

The following sample generates C2277:

```
// C2277.cpp
class A {
public:
   A();
};
(*pctor)() = &A::A;   // C2277
```

Compiler Error C2279

Error Message
exception specification cannot appear in a typedef declaration

Under **/Za**, exception specifications are not allowed in a typedef declaration.

The following sample generates C2279:

```
// C2279.cpp
// compile with: /Za /c
typedef int (*xx)() throw(...);   // C2279
typedef int (*xxx)();   // OK
```

Compiler Error C2283

Error Message
'identifier' : pure specifier or abstract override specifier not allowed on unnamed struct

A member function of an unnamed class or structure is declared with a pure specifier, which is not permitted.

The following sample generates C2283:

```
// C2283.cpp
// compile with: /c
struct {
   virtual void func() = 0;   // C2283
};
struct T {
   virtual void func() = 0;   // OK
};
```

Compiler Error C2284

Error Message
'identifier': illegal argument to intrinsic function, parameter 'number'

The parameter is invalid for the given function.

The following sample generates C2284:

```
// C2284.c
// processor: IPF
#include <mmintrin.h>
#include <emmintrin.h>

__m64  __m64_dep_mi(const int, __m64, const int, const int);
#pragma intrinsic(__m64_dep_mi)
__m64 a;

int main(void) {
   __m64 b;
   double d = 78.98;
   b = __m64_dep_mi(0x0, a, 0x1, d);   // C2284
   // try the following line instead
   // b = __m64_dep_mi(0, a, 0x1, 0x1);
}
```

Compiler Error C2285

Error Message

pointers to members representation has already been determined - pragma ignored

Two different representations exist for class.

Compiler Error C2286

Error Message
pointers to members of 'identifier' representation is already set to 'inheritance' – declaration ignored

Two different pointer-to-members representations exist for class.

For more information, see Inheritance Keywords.

Example
The following sample generates C2286:

```
// C2286.cpp
// compile with: /c
class __single_inheritance X;
class __multiple_inheritance X;   // C2286
class  __multiple_inheritance Y;   // OK
```

Compiler Error C2287

Error Message
'class': inheritance representation: 'representation1' is less general than the required 'representation2'

A class is declared with a simpler representation than required.

The following sample generates C2287:

```
// C2287.cpp
// compile with: /vmg /c
class __single_inheritance X;
class __single_inheritance Y;

struct A { };
struct B { };
struct X : A, B { };  // C2287  X uses multiple inheritance
struct Y : A { };  // OK
```

Compiler Error C2289

Error Message
same type qualifier used more than once

A type declaration or definition uses a type qualifier (**const**, **volatile**, **signed**, or **unsigned**) more than once, causing an error under ANSI compatibility (**/Za**).

The following sample generates C2286:

```
// C2289.cpp
// compile with: /Za /c
volatile volatile int i;   // C2289
volatile int j;   // OK
```

Compiler Error C2290

Error Message

C++ asm syntax ignored. Use __asm.

The **asm** syntax is reserved for future use.

Compiler Error C2292

Error Message
'identifier': best case inheritance representation: 'representation1' declared but 'representation2' required

Compiling the following code with /vmb ("Best-case always" representation) causes C2292.

```
// C2292.cpp
// compile with: /vmb
class __single_inheritance X;

struct A { };
struct B { };
struct X : A, B { };  // C2292, X uses multiple inheritance
```

Compiler Error C2293

Error Message
'identifier': illegal to have a member variable as a __based specifier

Specifiers for **__based** modifier must be nonmember pointers.

The following sample generates C2293:

```
// C2293.cpp
// compile with: /c
class A {
   static int *i;
   void __based(i) *bp;   // C2293
   void *bp2;   // OK
};
```

Compiler Error C2295

Error Message
escaped 'character' : is illegal in macro definition

A macro definition cannot contain an escape sequence with the specified character.

Compiler Error C2296

Error Message
'operator' : bad left operand

The left operand used with *operator* is invalid.

For example, the compiler may see a declaration where you intended a function call.

The following sample generates C2296:

```
// C2296.cpp
struct MyStruct {
   struct Help {
      Help(float f) : m_f(f) {}
      float m_f;
   };

   MyStruct(const Help &h) : m_f(h.m_f) {}
   MyStruct(float f) : m_f(f) {}
   MyStruct operator*(const MyStruct &f1) const {
      return MyStruct(m_f * f1.m_f);
   }

private:
   float m_f;
};

int main() {
   float f1 = 1.0f;

   MyStruct m_MyStruct1 ( MyStruct::Help( f1 ) );
   // try the following line instead
   // MyStruct m_MyStruct1 = MyStruct::Help( f1 );

   MyStruct m_MyStruct2 ( MyStruct::Help( f1 ) );
   // try the following line instead
   // MyStruct m_MyStruct2 = MyStruct::Help( f1 );

   MyStruct m_MyStruct3 = m_MyStruct1 * m_MyStruct2;   // C2296
}
```

Compiler Error C2297

Error Message
'operator' : bad right operand

The right operand used with *operator* is invalid.

For example, the compiler may see a declaration where you intended a function call.

The following sample generates C2297:

```
// C2297.cpp
struct MyStruct {
   struct Help {
      Help(float f) : m_f(f) {}
      float m_f;
   };

   MyStruct(const Help &h) : m_f(h.m_f) {}

   MyStruct(float f) : m_f(f) {}

   MyStruct operator*(const MyStruct &f1) const {
      return MyStruct(m_f * f1.m_f);
   }

private:
   float m_f;
};

int main() {
   float f1 = 1.0f;

   MyStruct m_MyStruct1 ( MyStruct::Help( f1 ) );
   // try the following line instead
   // MyStruct m_MyStruct1 = MyStruct::Help( f1 );

   MyStruct m_MyStruct2 ( MyStruct::Help( f1 ) );
   // try the following line instead
   // MyStruct m_MyStruct2 = MyStruct::Help( f1 );

   MyStruct m_MyStruct3 = m_MyStruct1 * m_MyStruct2;   // C2297
}
```

Compiler Error C2298

Error Message
'operation' : illegal operation on pointer to member function expression

A pointer to member-function expression must call the member function.

Example
The following sample generates C2298.

```
// C2298.cpp
#include <stdio.h>

struct X {
   void mf() {
      puts("in X::mf");
   }

   void mf2() {
      puts("in X::mf2");
   }
};

X x;
// pointer to member functions with no params and void return in X
typedef void (X::*pmf_t)();

// a pointer to member function X::mf
void (X::*pmf)() = &X::mf;

int main() {
   int (*pf)();
   pf = x.*pmf;   // C2298
   +(x.*pmf);     // C2298

   pmf_t pf2 = &X::mf2;
   (x.*pf2)();   // uses X::mf2
   (x.*pmf)();   // uses X::mf
}
```

The following sample generates C2298.

```
// C2298_b.cpp
// compile with: /c
void F() {}

class Measure {
public:
   void SetTrackingFunction(void (Measure::*fnc)()) {
      TrackingFunction = this->*fnc;   // C2298
      TrackingFunction = fnc;   // OK
      GlobalTracker = F;   // OK
   }
private:
   void (Measure::*TrackingFunction)(void);
   void (*GlobalTracker)(void);
};
```

Compiler Error C2299

Error Message
'function' : behavior change: an explicit specialization can not be a copy constructor or copy assignment operator

This error can also be generated as a result of compiler conformance work that was done for Visual C++ 2005: previous versions of Visual C++ allowed explicit specializations for a copy constructor or a copy assignment operator.

To resolve C2299, do not make the copy constructor or assignment operator a template function, but rather a non-template function that takes a class type. Any code that calls the copy constructor or assignment operator by explicitly specifying the template arguments needs to remove the template arguments.

See Breaking Changes in the Visual C++ 2005 Compiler for more information.

The following sample generates C2299:

```
// C2299.cpp
// compile with: /c
class C {
   template <class T>
   C (T t);

   template <> C (const C&);   // C2299
   C (const C&);   // OK
};
```

Compiler Errors C2300 Through C2398

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, type the error number (for example, C2322) in the index, or click a link below.

Compiler Error C2300

Compiler Error C2301

Compiler Error C2302

Compiler Error C2307

Compiler Error C2308

Compiler Error C2309

Compiler Error C2310

Compiler Error C2311

Compiler Error C2312

Compiler Error C2313

Compiler Error C2315

Compiler Error C2316

Compiler Error C2317

Compiler Error C2318

Compiler Error C2319

Compiler Error C2320

Compiler Error C2322

Compiler Error C2324

Compiler Error C2325

Compiler Error C2326

Compiler Error C2327

Compiler Error C2332

Compiler Error C2333

Compiler Error C2334

Compiler Error C2337

Compiler Error C2338

Compiler Error C2341

Compiler Error C2344

Compiler Error C2345

Compiler Error C2346

Compiler Error C2347

Compiler Error C2348

Compiler Error C2349

Compiler Error C2350

Compiler Error C2351

Compiler Error C2352
Compiler Error C2353
Compiler Error C2354
Compiler Error C2355
Compiler Error C2356
Compiler Error C2357
Compiler Error C2360
Compiler Error C2361
Compiler Error C2362
Compiler Error C2364
Compiler Error C2365
Compiler Error C2368
Compiler Error C2369
Compiler Error C2370
Compiler Error C2371
Compiler Error C2372
Compiler Error C2373
Compiler Error C2374
Compiler Error C2375
Compiler Error C2376
Compiler Error C2377
Compiler Error C2378
Compiler Error C2379
Compiler Error C2380
Compiler Error C2381
Compiler Error C2382
Compiler Error C2383
Compiler Error C2384
Compiler Error C2385
Compiler Error C2386
Compiler Error C2387
Compiler Error C2388
Compiler Error C2389
Compiler Error C2390
Compiler Error C2391
Compiler Error C2392
Compiler Error C2393
Compiler Error C2394
Compiler Error C2395
Compiler Error C2396

Compiler Error C2300

Error Message

'identifier' : class does not have a destructor called '~identifier'

The class does not have a destructor with the required name.

Compiler Error C2301

Error Message
left of '->~identifier' must point to class/struct/union

The expression to the left of the **->** operator does not evaluate to a pointer to a class, structure, or union.

Compiler Error C2302

Error Message

left of '.~identifier' must have class/struct/union type

The expression to the left of the period (.) operator is not a class, structure, or union.

Compiler Error C2307

Error Message
pragma 'pragma' must be outside function if incremental compilation is enabled

You must place the **data_seg** pragma between functions if you're using incremental compilation.

Compiler Error C2308

Error Message
concatenating mismatched strings

Both wide and non-wide character strings were specified for concatenation. You cannot concatenate a wide character string and non-wide character string.

Compiler Error C2309

Error Message
catch handler expected a parenthesized exception declaration

A catch handler has no parenthesized type.

The following sample generates C2309:

```
// C2309.cpp
// compile with: /EHsc
#include <eh.h>
class C {};
int main() {
   try {
      throw "ooops!";
   }
   catch C {}   // C2309
   // try the following line instead
   // catch( C ) {}
}
```

Compiler Error C2310

Error Message
catch handlers must specify one type

A catch handler specified no type or multiple types.

The following sample generates C2310:

```
// C2310.cpp
// compile with: /EHsc
#include <eh.h>
int main() {
   try {
      throw "Out of memory!";
   }
   catch( int ,int) {}   // C2310 two types
   // try the following line instead
   // catch( int)  {}
}
```

Compiler Error C2311

Error Message
'exception' : is caught by '...' on line number

The catch handler for the ellipsis (...) must be the last handler for a throw.

The following sample generates C2311:

```
// C2311.cpp
// compile with: /EHsc
#include <eh.h>
int main() {
   try {
      throw "ooops!";
   }
   catch( ... ) {}
   catch( int ) {}   // C2311  ellipsis handler not last catch
}
```

Compiler Error C2312

Error Message
'exception1' : is caught by 'exception2' on line number

Two handlers catch the same exception type.

The following sample generates C2312:

```
// C2312.cpp
// compile with: /EHsc
#include <eh.h>
int main() {
    try {
        throw "ooops!";
    }
    catch( signed int ) {}
    catch( int ) {}   // C2312
}
```

Compiler Error C2313

Error Message

'type1' : is caught by reference ('type2') on line number

The exception type has two handlers. The type for the second catch is a reference to the type of the first.

The following sample generates C2313:

```
// C2313.cpp
// compile with: /EHsc
#include <eh.h>
class C {};
int main() {
    try {
        throw "ooops!";
    }
    catch( C& ) {}
    catch( C ) {}   // C2313
}
```

Compiler Error C2315

Error Message

'type1' : reference is caught by 'type2' on line number

The exception type is handled by a previous handler. The reference for the second catch has the type of the first.

Compiler Error C2316

Error Message

'exception' : cannot be caught as the destructor and/or copy constructor are inaccessible

An exception was caught by value or by reference but the copy constructor and/or the assignment operator were inaccessible.

This code was accepted by the previous version's compiler but now gives an error. For more information, see Summary of Compile-Time Breaking Changes.

Example

The following sample generates C2316:

```
// C2316.cpp
// compile with: /EHsc
#include <stdio.h>

extern "C" int printf_s(const char*, ...);

struct B
{
public:
    B() {}
    // Delete the following line to resolve.
private:
    // copy constructor
    B(const B&)
    {
    }
};

void f(const B&)
{
}

int main()
{
    try
    {
        B aB;
        f(aB);
    }
    catch (B b) {   // C2316
        printf_s("Caught an exception!\n");
    }
}
```

Compiler Error C2317

Error Message
'try' block starting on line 'number' has no catch handlers

A **try** block must have at least one catch handler.

The following sample generates C2317:

```
// C2317.cpp
// compile with: /EHsc
#include <eh.h>
int main() {
   try {
      throw "throw an exception";
   }
   // C2317, no catch handler
}
```

Possible resolution:

```
// C2317b.cpp
// compile with: /EHsc
#include <eh.h>
int main() {
   try {
      throw "throw an exception";
   }
   catch(char*) {}
}
```

Compiler Error C2318

Error Message
no try block associated with this catch handler

A **catch** handler is defined but not preceded by a **try** block.

The following sample generates C2318:

```
// C2318.cpp
// compile with: /EHsc
#include <eh.h>
int main() {
   // no try block
   catch( int ) {}   // C2318
}
```

Possible resolution:

```
// C2318b.cpp
// compile with: /EHsc
#include <eh.h>
int main() {
   try{}
   catch( int ) {}
}
```

Compiler Error C2319

Error Message
'try/catch' must be followed by a compound statement. Missing '{'

A **try** or **catch** block is not found following the **try** or **catch** statement. The block must be enclosed in curly braces.

The following sample generates C2319:

```
// C2319.cpp
// compile with: /EHsc
#include <eh.h>
class C {};
int main() {
   try {
      throw "ooops!";
   }
   catch( C ) ;    // C2319
   // try the following line instead
   // catch( C ) {}
}
```

Compiler Error C2320

Error Message
expected ':' to follow access specifier 'specifier'

The keyword **public**, **protected**, or **private** must be followed by a colon.

Compiler Error C2322

Error Message
'identifier' : address of dllimport 'dllimport' is not static

A nonstatic value is given as the address of a function declared with **dllimport**.

Compiler Error C2324

Error Message
'identifier' : unexpected to the right of 'name'

A destructor is called using an incorrect identifier.

The following sample generates C2324:

```
// C2324.cpp
class A {};
typedef A* pA_t;
int i;

int main() {
   pA_t * ppa = new pA_t;
   ppa->~i;   // C2324
   ppa->~pA_t();   // OK
}
```

Compiler Error C2325

Error Message
'type' : unexpected type to the right of 'name'

A call is made to a destructor of incorrect type.

The following sample generates C2325:

```
// C2325.cpp
// compile with: /c
class A {};
typedef A* pA_t;
void f() {
    A** ppa = new A *;
    ppa->~A*;   // C2325

   pA_t *ppa2 = new pA_t;
   ppa2->~pA_t();   // OK
}
```

Compiler Error C2326

Error Message
'declarator' : function cannot access 'name'

The code tries to modify a member variable, which is not possible.

Example
The following sample generates C2326:

```
// C2326.cpp
void MyFunc() {
   int i;

   class MyClass  {
   public:
      void mf() {
         i = 4;   // C2326 i is inaccessible
      }
   };
}
```

Compiler Error C2327

Error Message
'symbol' : is not a type name, static, or enumerator

Code within a nested class attempts to access a member of the enclosing class that is not a type name, a static member, or an enumerator.

When compiling with **/clr**, a common cause for C2327 is a property with the same name as the property type.

The following sample generates C2327:

```
// C2327.cpp
int x;
class enclose {
public:
   int x;
   static int s;
   class inner {
      void f() {
         x = 1;   // C2327; enclose::x is not static
         s = 1;   // ok; enclose::s is static
         ::x = 1;   // ok; ::x refers to global
      }
   };
};
```

C2327 can also occur if the name of a type is hidden by the name of a member:

```
// C2327b.cpp
class X {};

class S {
   X X;
   // try the following line instead
   // X MyX;
   X other;   // C2327, rename member X
};
```

C2327 can also fire in this situation, where you need to fully specify the data type of the parameter:

```
// C2327c.cpp
// compile with: /c
struct A {};

struct B {
   int A;
   void f(A a) {   // C2327
   void f2(struct A a) {}   // OK
   }
};
```

The following sample generates C2327:

```
// C2327d.cpp
// compile with: /clr /c
using namespace System;

namespace NA {
   public enum class E : Int32 {
      one = 1,
      two = 2,
      three = 3
```

```
   };

   public ref class A {
   private:
      E m_e;
   public:
      property E E {
         NA::E get() {
            return m_e;
         }
         // At set, compiler doesn't know whether E is get_E or
         // Enum E, therefore fully qualifying Enum E is necessary
         void set( E e ) {   // C2327
            // try the following line instead
            // void set(NA::E e) {
            m_e = e;
         }
      }
   };
}
```

C2327 can also occur when using Managed Extensions for C++:

```
// C2327e.cpp
// compile with: /clr:oldSyntax /c
using namespace System;
namespace NA {
   public __value enum E : Int32 {
      one = 1, two = 2, three = 3
   };

   public __gc class A {
      E m_e;
      public:
         __property E get_E() {
            return m_e;
         }
         // At set_E compiler doesn't know whether E is get_E or
         // Enum E, therefore fully qualifying Enum E is necessary
         __property void set_E(E e) {   // C2327
         // try the following line instead
         // __property void set_E(NA::E e) {
            m_e = e;
         }
   };
}
```

The following sample shows C2327 when a property has the same name as the property type:

```
// C2327f.cpp
// compile with: /clr /c
public value class Address {};

public ref class Person {
public:
   property Address Address {
      ::Address get() {
         return address;
      }
      void set(Address addr) {   // C2327
      // try the following line instead
      // set(::Address addr) {
         address = addr;
      }
   }
private:
```

```
   Address address;   // C2327
   // try the following line instead
   // ::Address address;
};
```

The following sample shows C2327 when a property has the same name as the property type:

```
// C2327g.cpp
// compile with: /clr:oldSyntax /c
#using <mscorlib.dll>
public __value struct Address {};

public __gc class Person {
public:
   __property ::Address get_Address() {
      return address;
   }

   __property void set_Address(Address addr)   // C2327
   // try the following line instead
   // __property void set_Address(::Address addr) {
      address = addr;
   }

private:
   Address address;   // C2327

   // try the following line instead
   // ::Address address;
};
```

Compiler Error C2332

Error Message
'typedef' : missing tag name

The compiler found an incomplete type definition.

The following sample generates C2332:

```
// C2332.cpp
// compile with: /c
struct S {
   int i;
};

typedef struct * pS;   // C2332
typedef struct S* pS;   // OK

int get_S_i(pS p) {
   return p->i;
}
```

Compiler Error C2333

Error Message
'function' : error in function declaration; skipping function body

This error occurs after another error, for member functions defined inside their class.

The following sample generates C2333:

```
// C2333.cpp
struct s1 {
   s1(s1) {}   // C2333
};
```

Compiler Error C2334

Error Message
unexpected token(s) preceding ': or {'; skipping apparent function body

This error occurs after another error, for member functions defined inside their class.

The following sample generates C2334:

```
// C2334.cpp
// compile with: /c
struct s1 {
   s1( {}   // C2334
   s1() {}   // OK
};
```

Compiler Error C2337

Error Message
'attribute name' : attribute not found

You have used an attribute that is not supported in this version of Visual C++.

The following sample generates C2337:

```
// C2337.cpp
// compile with: /c
[emitidl];
[module(name="x")];
[grasshopper]   // C2337, not a supported attribute
class a{};
```

Compiler Error C2338

Error Message
'identifier' Atl Attribute Provider

The attribute references an undeclared identified. You may see this error associated with ATL Provider Error ATL2010.

Compiler Error C2341

Error Message
'section name' : segment must be defined using #pragma data_seg, code_seg or section prior to use

An allocate statement refers to a segment not yet defined by code_seg, data_seg, or section pragmas.

The following sample generates C2341:

```
// C2341.cpp
// compile with: /c
__declspec(allocate(".test"))   // C2341
int j = 1;
```

Possible resolution:

```
// C2341b.cpp
// compile with: /c
#pragma data_seg(".test")
__declspec(allocate(".test"))
int j = 1;
```

Compiler Error C2344

Error Message
align(#) : alignment must be power of two

When using the align keyword, the value you pass must be a power of two.

For example, the following code generates C2344 because 3 is not a power of two:

```
// C2344.cpp
// compile with: /c
__declspec(align(3)) int a;   // C2344
__declspec(align(4)) int b;   // OK
```

Compiler Error C2345

Error Message
align(value) : illegal alignment value

You passed a value to the align keyword that is outside the allowable range.

The following code generates C2345

```
// C2345.cpp
// compile with: /c
__declspec(align(0)) int a;   // C2345
__declspec(align(1)) int a;   // OK
```

Compiler Error C2346

Error Message
'function' cannot be compiled as native: reason

The compiler was unable to compile a function MSIL.

For more information, see managed, unmanagedand /clr (Common Language Runtime Compilation).

To correct this error

1. Remove the code in the function that cannot be compiled to MSIL.
2. Do not compile the module with /clr or mark the function as unmanaged with the unmanaged pragma.

Example
The following sample generates C2346.

```
// C2346.cpp
// compile with: /clr /c
// C2346 error expected
// processor: x86
#include <dos.h>   // definitions for _disable, _enable
#pragma intrinsic(_disable)
#pragma intrinsic(_enable)

ref class C {
   Object ^ f0;
public:
   Object ^ f1() {
      _disable();
      _enable();
      return f0;
   }

   void f2() {
      _disable();
      _enable();
   }

   static void f3() {
      _disable();
      _enable();
   }
};
```

Compiler Error C2347

Error Message

'__w64' : cannot be used with type 'type'

__w64 only applies to variables of the **int** and **long** data types.

A **short** integer can only be 32 bits.

Compiler Error C2348

Error Message
'type name' : is not a C-style aggregate, cannot be exported in embedded-IDL

To place a **struct** in a .idl file with the export attribute, the **struct** must contain only data.

The following sample generates C2348:

```
// C2348.cpp
// C2348 error expected
[ module(name="SimpleMidlTest") ];

[export]
struct Point {
   // Delete the following two lines to resolve.
   Point() : m_i(0), m_j(0) {}
   Point(int i, int j) : m_i(i), m_j(j) {}

   int m_i;
   int m_j;
};
```

Compiler Error C2349

Error Message

'function' cannot be compiled as managed: 'reason'; use #pragma unmanaged

For more information see Compiler Warning (level 1 and 3) C4793.

Compiler Error C2350

Error Message
'identifier' is not a static member

Nonstatic members of a class or structure cannot be defined.

Compiler Error C2351

Error Message
obsolete C++ constructor initialization syntax

In a new-style initialization list for a constructor, you must explicitly name each direct base class, even if it is the only base class.

The following sample generates C2351:

```
// C2351.cpp
// compile with: /c
class B {
public:
   B() : () {}   // C2351
   B() {}   // OK
};
```

Compiler Error C2352

Error Message
'class::function' : illegal call of non-static member function

A **static** member function called a nonstatic member function. Or, a nonstatic member function was called from outside the class as a static function.

The following sample generates C2352:

```
// C2352.cpp
// compile with: /c
class CMyClass {
public:
   static void func1();
   void func2();
   static void func3() {
      func2();   // C2352 calls nonstatic func2
      func1();   // OK calls static func1
   }
};
```

The following sample generates C2352:

```
// C2352b.cpp
class MyClass {
public:
   void MyFunc() {}
   static void MyFunc2() {}
};

int main() {
   MyClass::MyFunc();   // C2352
   MyClass::MyFunc2();   // OK
}
```

Compiler Error C2353

Error Message
exception specification is not allowed

Exception specifications are not allowed on member functions of managed classes.

The following sample generates C2353:

```
// C2353.cpp
// compile with: /clr /c
ref class X {
   void f() throw(int);   // C2353
   void f();   // OK
};
```

Compiler Error C2354

Error Message
'reference' : initialization of reference member requires a temporary variable

A constructor initializes a reference to a member instead of initializing the member.

It is invalid to initialize a reference member of a class in the class's constructor with a temporary variable. An attempt to do so generates the C2354 error.

The following sample generates C2354:

```
// C2354.cpp
int temp() { return 1; }
class Test
{
public:
    int member;
    int& ref_member;
    Test();
};

Test::Test() : ref_member( temp() )
{   // C2354
}
```

When this error is encountered, the solution is to change the code so that the reference member is not initialized to a temporary variable. The reference must be initialized to an object that will exist for the lifetime of the reference member.

Compiler Error C2355

Error Message
'this' : can only be referenced inside non-static member functions

The **this** pointer is valid only within non-static member functions.

The following sample generates C2355:

```
// C2355.cpp
// compile with: /c
class MyClass {};
MyClass *p = this;   // C2355

// OK
class MyClass2 {
public:
   void Test() {
      MyClass2 *p = this;
   }
};
```

Compiler Error C2356

Error Message
initialization segment must not change during translation unit

Possible causes:

- **#pragma init_seg** preceded by segment initialization code
- **#pragma init_seg** preceded by another **#pragma init_seg**

To resolve, move the segment initialization code to the beginning of the module. If multiple areas must be initialized, move them to separate modules.

The following sample generates C2356:

```
// C2356.cpp
#pragma warning(disable : 4075)

int __cdecl myexit(void (__cdecl *)());
int __cdecl myexit2(void (__cdecl *)());

#pragma init_seg(".mine$m",myexit)
#pragma init_seg(".mine$m",myexit2)   // C2356
```

Compiler Error C2357

Error Message
'identifier' : must be a function of type 'type'

Your code declares a version of the **atexit** function that does not match the version declared internally by the compiler. Declare **atexit** as follows:

```
int __cdecl atexit(void (__cdecl *)());
```

For more information, see init_seg.

The following sample generates C2357:

```
// C2357.cpp
// compile with: /c
// C2357 expected
#pragma warning(disable : 4075)
// Uncomment the following line to resolve.
// int __cdecl myexit(void (__cdecl *)());
#pragma init_seg(".mine$m",myexit)
```

Compiler Error C2360

Error Message
initialization of 'identifier' is skipped by 'case' label

The initialization of *identifier* can be skipped in a **switch** statement. You cannot jump past a declaration with an initializer unless the declaration is enclosed in a block. (Unless it is declared within a block, the variable is within scope until the end of the **switch** statement.)

The following sample generates C2360:

```
// C2360.cpp
int main() {
   int x = 0;
   switch ( x ) {
   case 0 :
      int i = 1;
      { int j = 1; }
   case 1 :   // C2360
      int k = 1;
   }
}
```

Possible resolution:

```
// C2360b.cpp
int main() {
   int x = 0;
   switch ( x ) {
   case 0 :
      { int j = 1; int i = 1;}
   case 1 :
      int k = 1;
   }
}
```

Compiler Error C2361

Error Message
initialization of 'identifier' is skipped by 'default' label

The initialization of *identifier* can be skipped in a **switch** statement. You cannot jump past a declaration with an initializer unless the declaration is enclosed in a block. (Unless it is declared within a block, the variable is within scope until the end of the **switch** statement.)

The following sample generates C2361:

```
// C2361.cpp
void func( void ) {
   int x;
   switch (x) {
   case 0 :
      int i = 1;
      { int j = 1; }
   default :   // C2361 error
      int k = 1;
   }
}
```

Possible resolution:

```
// C2361b.cpp
// compile with: /c
void func( void ) {
   int x = 0;
   switch (x) {
   case 0 :
      { int j = 1; int i = 1;}
   default :
      int k = 1;
   }
}
```

Compiler Error C2362

Error Message
initialization of 'identifier' is skipped by 'goto label'

When compiling with /Za, jumping to the label prevents the identifier from being initialized.

You cannot jump past a declaration with an initializer unless the declaration is enclosed in a block that is not entered, or the variable has already been initialized.

The following sample generates C2326:

```
// C2362.cpp
// compile with: /Za
int main() {
   goto label1;
   int i = 1;      // C2362, initialization skipped
label1:;
}
```

Possible resolution:

```
// C2362b.cpp
// compile with: /Za
int main() {
   goto label1;
   {
      int j = 1;   // OK, this block is never entered
   }
label1:;
}
```

Compiler Error C2364

Error Message
'type': illegal type for custom attribute

Named arguments for custom attributes are limited to compile time constants. For example, integral types (int, char, etc.), System::Type^, and System::Object^.

Example
The following sample generates C2364.

```
// c2364.cpp
// compile with: /clr /c
using namespace System;

[attribute(AttributeTargets::All)]
public ref struct ABC {
public:
   // Delete the following line to resolve.
   ABC( Enum^ ) {}   // C2364
   ABC( int ) {}   // OK
};
```

Compiler Error C2365

Error Message
'class member' : redefinition; previous definition was 'class member'

You attempted to redefine a class member.

The following sample generates C2365.

```
// C2365.cpp
// compile with: /c
class C1 {
   int CFunc();
   char *CFunc;   // C2365, already exists as a member function

   int CMem;
   char *CMem();   // C2365, already exists as a data member
};
```

Compiler Error C2368

Error Message

'identifier' : redefinition; different allocation specifiers

The declaration and definition of the symbol specify different **__declspec** attributes.

Compiler Error C2369

Error Message
'array' : redefinition; different subscripts

The array is already declared with a different subscript.

The following sample generates C2369:

```
// C2369.cpp
// compile with: /c
int a[10];
int a[20];   // C2369
int b[20];   // OK
```

Compiler Error C2370

Error Message
'identifier' : redefinition; different storage class

The identifier is already declared with a different storage class.

Example
The following sample generates C2370:

```
// C2370.cpp
// compile with: /Za /c
extern int i;
static int i;   // C2370
int i;   // OK
```

The following sample generates C2370:

```
// C2370b.cpp
#define Thread __declspec( thread )
extern int tls_i;
int Thread tls_i;   // C2370 declaration and the definition differ
int tls_i;   // OK
```

Compiler Error C2371

Error Message
'identifier' : redefinition; different basic types

The identifier is already declared.

The following sample generates C2371:

```
// C2371.cpp
int main() {
   int i;
   float i;   // C2371, redefinition
   float f;   // OK
}
```

Compiler Error C2372

Error Message
'identifier' : redefinition; different types of indirection

The identifier is already defined with a different derived type.

The following sample generates C2326:

```
// C2372.cpp
// compile with: /c
extern int *fp;
extern int fp[];   // C2372
extern int fp2[];   // OK
```

Compiler Error C2373

Error Message
'identifier' : redefinition; different type modifiers

The identifier is already defined with a different type modifier.

The following sample generates C2373:

```
// C2373.h
void __clrcall func( void );
const int i = 20;
```

And then:

```
// C2373.cpp
// compile with: /c
#include "C2373.h"
extern void __cdecl func( void );   // C2373
extern void __clrcall func( void );   // OK

extern int i;   // C2373
extern const int i;   // OK
```

Compiler Error C2374

Error Message
'identifier' : redefinition; multiple initialization

The identifier is initialized more than once.

The following sample generates C2374:

```
// C2374.cpp
// compile with: /c
int i = 0;
int i = 1;   // C2374
int j = 1;   // OK
```

Compiler Error C2375

Error Message
'function' : redefinition; different linkage

The function is already declared with a different linkage specifier.

The following sample generates C2375:

```
// C2375.cpp
// compile with: /Za /c
extern void func( void );
static void func( void );   // C2375
static void func2( void );   // OK
```

Compiler Error C2376

Error Message

'function' : redefinition; different based allocation

The function is already declared with a different based allocation.

Compiler Error C2377

Error Message
'identifier' : redefinition; typedef cannot be overloaded with any other symbol

A **typedef** identifier is redefined.

The following sample generates C2377:

```
// C2377.cpp
// compile with: /c
typedef int i;
int i;   // C2377
int j;   // OK
```

Compiler Error C2378

Error Message
'identifier' : redefinition; symbol cannot be overloaded with a typedef

The identifier was redefined as a **typedef**.

The following sample generates C2378:

```
// C2378.cpp
// compile with: /c
int i;
typedef int i;   // C2378
typedef int b;   // OK
```

Compiler Error C2379

Error Message
formal parameter number has different type when promoted

The type of the specified parameter is not compatible, through default promotions, with the type in a previous declaration. This is an error in ANSI C (/Za) and a warning with Microsoft extensions (**/Ze**).

The following sample generates C2379:

```
// C2379.c
// compile with: /Za
void func();
void func(char);   // C2379, char promotes to int
```

Compiler Error C2380

Error Message
type(s) preceding 'identifier' (constructor with return type, or illegal redefinition of current class-name?)

A constructor returns a value or redefines the class name.

The following sample generates C2326:

```
// C2380.cpp
// compile with: /c
class C {
public:
   int C();   // C2380, specifies an int return
   int C;   // C2380, redefinition of i
   C();   // OK
};
```

Compiler Error C2381

Error Message
'function' : redefinition; __declspec(noreturn) differs

A function was declared and then defined but the definition used the noreturn **__declspec** modifier. The use of **noreturn** constitutes a redefinition of the function; the declaration and definition need to agree on the use of **noreturn**.

The following sample generates C2381:

```
// C2381.cpp
// compile with: /c
void f1();
void __declspec(noreturn) f1() {}   // C2381
void __declspec(noreturn) f2() {}   // OK
```

Compiler Error C2382

Error Message
'function' : redefinition; different exception specifications

Under /Za, this error indicates that a function overload was attempted only on the exception specification.

The following sample generates C2382:

```
// C2382.cpp
// compile with: /Za /c
void f1(void) throw(int) {}
void f1(void) throw(char) {}   // C2382
void f2(void) throw(char) {}   // OK
```

Compiler Error C2383

Error Message
'symbol' : default-arguments are not allowed on this symbol

The C++ compiler does not allow default arguments on pointers to functions.

This code was accepted by the previous version's compiler but now gives an error. For code that works in all versions of Visual C++, do not assign a default value to a pointer-to-function argument.

For more information, see Summary of Compile-Time Breaking Changes.

The following line generates C2383:

```
// C2383.cpp
// compile with: /c
void (*pf)(int = 0);   // C2383
void (*pf)(int);   // OK
```

Compiler Error C2384

Error Message
'member' : cannot apply __declspec(thread) to a member of a managed class

The thread **__declspec** modifier cannot be used on a member of a managed.

Static TLS can only be used for statically loaded DLLs -- the DLL must be statically loaded when the process starts.

The following line generates C2384:

```
// C2384.cpp
// compile with: /clr /c
public ref class B {
public:
   __declspec( thread ) static int tls_i = 1;   // C2384

   // OK
   [System::ThreadStaticAttribute]
   static int tls_i;
};
```

The following line generates C2384:

```
// C2384_b.cpp
// compile with: /clr:oldSyntax /c
public __gc class B {
public:
   __declspec( thread ) static int tls_i = 1;   // C2384

   // OK
   [System::ThreadStaticAttribute]
   static int tls_i;
};
```

Compiler Error C2385

Error Message
ambiguous access of 'member'

The member can derive from more than one object (it is inherited from more than one object). To resolve this error,

- Make the member unambiguous by providing a cast.
- Rename the ambiguous members in the base classes.

Example
The following sample generates C2385.

```
// C2385.cpp
// C2385 expected
#include <stdio.h>

struct A
{
    void x(int i)
    {
        printf_s("\nIn A::x");
    }
};

struct B
{
    void x(char c)
    {
        printf_s("\nIn B::x");
    }
};

// Delete the following line to resolve.
struct C : A, B {}

// Uncomment the following 4 lines to resolve.
// struct C : A, B
// {
//     using B::x;
//     using A::x;
// };

int main()
{
    C aC;
    aC.x(100);
    aC.x('c');
}

struct C : A, B
{
    using B::x;
    using A::x;
};
```

Compiler Error C2386

Error Message
'symbol' : a symbol with this name already exists in the current scope

You tried to create a namespace alias, but the name you chose already exists.

The following sample generates C2386:

```
// C2386.cpp
namespace A {
   int k;
}

int i;
namespace i = A;   // C2386, i already exists
```

Compiler Error C2387

Error Message
'type' : ambiguous base class

The compiler could not unambiguously resolve a function call because the function exists in more than one base class.

To resolve this error, either remove one of the base classes from the inheritance, or explicitly qualify the function call.

The following sample generates C2387:

```
// C2387.cpp
namespace N1 {
   struct B {
      virtual void f() {
      }
   };
}

namespace N2 {
   struct B {
      virtual void f() {
      }
   };
}

struct D : N1::B, N2::B {
   virtual void f() {
      B::f();   // C2387
      // try the following line instead
      // N1::B::f();
   }
};

int main() {
   D aD;
   aD.f();
}
```

Compiler Error C2388

Error Message
'symbol' : a symbol cannot be declared with both __declspec(appdomain) and __declspec(process)

The **appdomain** and **process** **__declspec** modifiers cannot be used on the same symbol. The storage for a variable exists per process or per application domain.

For more information, see appdomain and process.

The following sample generates C2388:

```
// C2388.cpp
// compile with: /clr /c
__declspec(process) __declspec(appdomain) int i;   // C2388
__declspec(appdomain) int i;   // OK
```

Compiler Error C2389

Error Message
'operator' : illegal operand 'nullptr'

nullptr cannot be an operand.

The following sample generates C2389:

```
// C2389.cpp
// compile with: /clr
int main() {
   throw nullptr;   // C2389
}
```

Compiler Error C2390

Error Message
'identifier' : incorrect storage class 'specifier'

The storage class is not valid for the global-scope identifier. The default storage class is used in place of the invalid class.

Possible resolutions:

- If the identifier is a function, declare it with **extern** storage.
- If the identifier is a formal parameter or local variable, declare it with auto storage.
- If the identifier is a global variable, declare it with no storage class (auto storage).

Example

- The following sample generates C2390:

```
// C2390.cpp
register int i;   // C2390

int main() {
   register int j;   // OK
}
```

Compiler Error C2391

Error Message
'identifier' : 'friend' cannot be used during type definition

The **friend** declaration includes a complete class declaration. A **friend** declaration can specify a member function or an elaborated type specifier, but not a complete class declaration.

The following sample generates C2326:

```
// C2391.cpp
// compile with: /c
class D {
   void func( int );
};

class A {
   friend class B { int i; };   // C2391

   // OK
   friend class C;
   friend void D::func(int);
};
```

Compiler Error C2392

Error Message

'method1' : covariant returns types are not supported in managed types, otherwise 'method2' would be overridden

Covariant return types are not allowed when compiling with the /clr (Common Language Runtime Compilation) option.

Example

The following sample generates C2392.

```
// C2392.cpp
// compile with: /clr
public ref struct B {
public:
   int i;
};

public ref struct D: public B{};

public ref struct B1 {
public:
   virtual B^ mf() {
      B^ pB = gcnew B;
      pB->i = 11;
      return pB;
   }
};

public ref struct D1: public B1 {
public:
   virtual D^ mf() override {  // C2392
   // try the following line instead
   // virtual B^ mf() override {
   // return type D^ is covariant with B^, not allowed with CLR types
      D^ pD = gcnew D;
      pD->i = 12;
      return pD;
   }
};

int main() {
   B1^ pB1 = gcnew D1;
   B^ pB = pB1->mf();
   D^ pD = dynamic_cast<D^>(pB);
}
```

The following sample generates C2392.

```
// C2392_2.cpp
// compile with: /clr:oldSyntax

public __gc struct B {
   int i;
};

public __gc struct D : public B {};

public __gc struct B1 {
   virtual B* mf() {
      B* pB= new B;
      pB->i = 11;
      return pB;
   }
};
```

```
public __gc struct D1 : public B1 {
   virtual D* mf() {   // C2392
   // try the following line instead
   // virtual B* mf() {
   // return type D* is covariant with B*, not allowed with C:R types
      D * pD = new D();
      pD->i = 12;
      return pD;
   }
};

int main() {
   B1 *pB1 = new D1;
   B * pB = pB1->mf();
   D * pD = dynamic_cast<D *>(pB);
}
```

Compiler Error C2393

Error Message
'symbol' : per-appdomain symbol cannot be allocated in segment 'segment'

The use of appdomain variables implies that you are compiling with **/clr:pure** or **/clr:safe**, and a safe or pure image cannot contain data segments.

See /clr (Common Language Runtime Compilation) for more information.

Example
The following sample generates C2393.

```
// C2393.cpp
// compile with: /clr:pure /c
#pragma data_seg("myseg")
int n = 0;   // C2393
```

Compiler Error C2394

Error Message

'your_type::operator'op'" : CLR operator not valid. At least one parameter must be of the following types: 'T^', 'T^%', 'T^&', where T = 'your_type'

An operator in a managed type did not have at least one parameter whose type is the same as the type of the operator return value.

The following sample generates C2394:

```
// C2394.cpp
// compile with: /clr /c
ref struct Y {
   static Y^ operator -(int i, char c);   // C2394

   // OK
   static Y^ operator -(Y^ hY, char c);
   // or
   static Y^ operator -(int i, Y^& rhY);
};
```

Compiler Error C2395

Error Message
'your_type::operator'op'' : CLR operator not valid. At least one parameter must be of the following types: 'T', 'T%', 'T&', 'T^', 'T^%', 'T^&', where T = 'your_type'

An operator in a managed type did not have at least one parameter whose type is the same as the type of the operator return value.

The following sample generates C2395:

```
// C2395.cpp
// compile with: /clr /c
value struct V {
   static V operator *(int i, char c);   // C2395

   // OK
   static V operator *(V v, char c);
   // or
   static V operator *(int i, V& rv);
};
```

Compiler Error C2396

Error Message
'your_type::operator'op'' : CLR UDC operator not valid. Must either convert from or convert to: 'T^', 'T^%', 'T^&', where T = 'your_type'

An operator in a managed type did not have at least one parameter whose type is the same as the type containing the operator.

The following sample generates C2396:

```
// C2396.cpp
// compile with: /clr /c

ref struct Y {
   static operator int(char c);   // C2396

   // OK
   static operator int(Y^ hY);
   // or
   static operator Y^(char c);
};
```

Compiler Errors C2400 Through C2499

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, type the error number (C2421, for example) in the index, or click a link below.

Compiler Error C2400

Compiler Error C2401

Compiler Error C2402

Compiler Error C2403

Compiler Error C2404

Compiler Error C2405

Compiler Error C2406

Compiler Error C2407

Compiler Error C2408

Compiler Error C2409

Compiler Error C2410

Compiler Error C2411

Compiler Error C2412

Compiler Error C2413

Compiler Error C2414

Compiler Error C2415

Compiler Error C2417

Compiler Error C2418

Compiler Error C2419

Compiler Error C2420

Compiler Error C2421

Compiler Error C2422

Compiler Error C2423

Compiler Error C2424

Compiler Error C2425

Compiler Error C2426

Compiler Error C2427

Compiler Error C2428

Compiler Error C2430

Compiler Error C2431

Compiler Error C2432

Compiler Error C2433

Compiler Error C2434

Compiler Error C2435

Compiler Error C2436

Compiler Error C2437
Compiler Error C2438
Compiler Error C2439
Compiler Error C2440
Compiler Error C2441
Compiler Error C2443
Compiler Error C2444
Compiler Error C2446
Compiler Error C2447
Compiler Error C2448
Compiler Error C2449
Compiler Error C2450
Compiler Error C2451
Compiler Error C2452
Compiler Error C2457
Compiler Error C2458
Compiler Error C2459
Compiler Error C2460
Compiler Error C2461
Compiler Error C2462
Compiler Error C2463
Compiler Error C2464
Compiler Error C2465
Compiler Error C2466
Compiler Error C2467
Compiler Error C2469
Compiler Error C2470
Compiler Error C2471
Compiler Error C2473
Compiler Error C2474
Compiler Error C2477
Compiler Error C2479
Compiler Error C2480
Compiler Error C2482
Compiler Error C2483
Compiler Error C2485
Compiler Error C2486
Compiler Error C2487
Compiler Error C2488
Compiler Error C2489

Compiler Error C2490

Compiler Error C2491

Compiler Error C2492

Compiler Error C2493

Compiler Error C2494

Compiler Error C2495

Compiler Error C2496

Compiler Error C2498

Compiler Error C2499

Compiler Error C2400

Error Message
inline assembler syntax error in 'context'; found 'token'

The token caused a syntax error in the specified context.

The following sample generates C2400:

```
// C2400.cpp
// processor: x86
int main() {
   __asm {
      heh ax,bx;   // C2400
      mov ax,bx;   // OK
   }
}
```

Compiler Error C2401

Error Message

'identifier' : register must be base in 'context'

The register used in an indirect memory operand must be a base register in this context.

Compiler Error C2402

Error Message

'identifier' : register must be index in 'context'

The register used in an indirect memory operand must be an index register in this context.

Compiler Error C2403

Error Message

'identifier' : register must be base/index in 'context'

The register used in an indirect memory operand must be a base or index register in this context.

Compiler Error C2404

Error Message

'identifier' : illegal register in 'context'

This register is invalid in this context.

Compiler Error C2405

Error Message
illegal short forward reference with offset

Short forward references must refer to a label only. An additional offset cannot be used.

Compiler Error C2406

Error Message
'identifier' : name undefined in 'context'

An undefined identifier is used with the **SIZE**, **LENGTH**, or member-selection (.) operator.

Compiler Error C2407

Error Message
illegal 'float' register in 'context'

An **NDP** register was specified in an invalid context.

Compiler Error C2408

Error Message
illegal type on PTR operator in 'context'

The first parameter of the **PTR** operator is not a legal type specification.

Compiler Error C2409

Error Message

illegal type used as operator in 'context'

The type is not legal as an operator in this context.

Compiler Error C2410

Error Message
'identifier' : ambiguous member name in 'context'

The identifier is a member of more than one structure or union in this context.

Use a structure or union specifier on the operand that caused the error. A structure or union specifier is an identifier of type **struct** or **union** (a **typedef** name or a variable of the same type as the structure or union being referenced). The specifier must be the left operand of the first member-selection operator (.) to use the operand.

Compiler Error C2411

Error Message
'identifier' : illegal struct/union member in 'context'

To fix by checking the following possible causes

1. The identifier is not a member of a visible structure or union in this context.
2. The identifier is not a member of the structure or union specified with the member-selection (.) operator.

Compiler Error C2412

Error Message

'label' : case-insensitive label redefined

The label is defined more than once in the current function. Change the spelling of the label and its references.

Compiler Error C2413

Error Message
'token' : illegal align size

The size used with the **ALIGN** directive is missing or outside the valid range.

Compiler Error C2414

Error Message
illegal number of operands

To fix by checking the following possible causes

1. The opcode does not support the number of operands used. Check an assembly-language reference manual to determine the correct number of operands.
2. A newer processor supports the instruction with a different number of operands. Adjust the /G option to use the later processor.

Compiler Error C2415

Error Message
improper operand type

The opcode does not use operands of this type.

To fix by checking the following possible causes

1. The opcode does not support the number of operands used. Check an assembly-language reference manual to determine the correct number of operands.
2. A newer processor supports the instruction with additional types. Adjust the /G option to use the later processor.

Compiler Error C2417

Error Message
divide by zero in 'context'

The parameter to the right of the division operator is zero in this context.

Compiler Error C2418

Error Message
cannot delete browser file: filename

The compiler could not delete the browser file.

To fix by checking the following possible causes

1. File is read-only.
2. File is in use by another process.

Compiler Error C2419

Error Message
mod by zero in 'context'

The parameter to the right of the **MOD** operator is zero in this context.

Compiler Error C2420

Error Message
'identifier' : illegal symbol in context

The identifier is invalid in this context.

Compiler Error C2421

Error Message
PTR operator used with register in 'context'

The **PTR** operator cannot be used with a **register** operand.

Compiler Error C2422

Error Message
illegal segment override in 'operand'

Inline assembly code incorrectly uses a segment override operator (colon) on an operand. Possible causes include:

- The register preceding the operator is not a segment register.
- The register preceding the operator is not the only segment register in the operand.
- The segment override operator appears within an indirection operator (brackets).
- The expression following the segment override operator is not an immediate operand or a memory operand.

The following sample generates C2422:

```
// C2422.cpp
// processor: x86
int main() {
   _asm {
      mov AX, [BX:ES]   // C2422
      mov AX, ES   // OK
   }
}
```

Compiler Error C2423

Error Message
'number' : illegal scale

Inline assembly code uses a number other than 1, 2, 4, or 8 to scale a register.

The following sample generates C2423:

```
// C2423.cpp
// processor: x86
int main() {
   _asm {
      lea EAX, [EAX*3]   // C2423
      lea EAX, [EAX+EAX*2]   // OK
   }
}
```

Compiler Error C2424

Error Message

'token' : improper expression in 'context'

The token forms part of an incorrect expression in this context.

Compiler Error C2425

Error Message
'token' :non-constant expression in 'context'

The token forms part of a non-constant expression in this context.

The following sample generates C2425:

```
// C2425.cpp
// processor: x86
int main() {
   int i = 3;
   __asm {
      mov eax, [ebp - i]   // C2425
      mov eax, [ebp - 3]   // OK
   }
}
```

Compiler Error C2426

Error Message
'token' : illegal operator in 'context'

The token cannot be used as an operator in this context. Index operators, for example, cannot be nested.

Compiler Error C2427

Error Message
'class' : cannot define class in this scope

An attempt was made to define a nested class, but the nested class is a member of a base class, not the most containing class.

The following sample generates C2427:

```
// C2427.cpp
// compile with: /c
template <class T>
struct S {
   struct Inner;
};

struct Y : S<int> {};

struct Y::Inner {};   // C2427

// OK
template<typename T>
struct S<T>::Inner {};
```

Compiler Error C2428

Error Message
'operation' : not allowed on operand of type 'bool'

You cannot apply a decrement operator to objects of type **bool**.

The following sample generates C2428:

```
// C2428.cpp
void g(bool fFlag) {
   --fFlag;   // C2428
   fFlag--;   // C2428
}
```

Compiler Error C2430

Error Message

more than one index register in 'identifier'

More than one register is scaled. The compiler supports scaled indexing, but you can only scale one register.

Example

The following sample generates C2430.

```
// C2430.cpp
// processor: x86
int main() {
   _asm mov eax, [ebx*2+ecx*4] // C2430
}
```

Compiler Error C2431

Error Message
illegal index register in 'identifier'

The ESP register is scaled or used as both index and base register. The SIB encoding for the x86 processor does not allow either.

The following sample generates C2431:

```
// C2431.cpp
// processor: x86
int main() {
   _asm mov ax, [ESI + 2*ESP]   // C2431
   _asm mov ax, [esp + esp]   // C2431
}
```

Compiler Error C2432

Error Message
illegal reference to 16-bit data in 'identifier'

A 16-bit register is used as an index or base register. The compiler does not support referencing 16-bit data. 16-bit registers cannot be used as index or base registers when compiling for 32-bit code.

The following sample generates C2432:

```
// C2432.cpp
// processor: x86
int main() {
   _asm mov eax, DWORD PTR [bx]   // C2432
}
```

Compiler Error C2433

Error Message

'identifier' : 'modifier' not permitted on data declarations

The **friend**, **virtual**, and **inline** modifiers cannot be used for data declarations.

Example

The following sample generates C2433.

```
// C2433.cpp
class C{};

int main() {
   inline C c;   // C2433
}
```

Compiler Error C2434

Error Message

'symbol' : a symbol declared with __declspec(process) cannot be dynamically initialized in /clr:pure mode

It is not possible to dynamically initialize a per-process variable under **/clr:pure**. For more information, see /clr (Common Language Runtime Compilation) and process.

Example

The following sample generates C2434.

```
// C2434.cpp
// compile with: /clr:pure /c
int f() { return 0; }
__declspec(process) int i = f();   // C2434
__declspec(process) int i2 = 0;   // OK
```

Compiler Error C2435

Error Message
'var' : dynamic initialization requires managed CRT, cannot compile with /clr:safe

Initialization of global per–application domain variable requires the CRT compiled with **/clr:pure**, which does not produce a verifiable image.

For more information, see appdomain and process.

The following sample generates C2435:

```
// C2435.cpp
// compile with: /clr:safe /c
int globalvar = 0;   // C2435

__declspec(process)
int globalvar2 = 0;
```

Compiler Error C2436

Error Message
'identifier' : member function or nested class in constructor initializer list

Member functions or local classes in the constructor initializer list cannot be initialized.

The following sample generates C2436:

```
// C2436.cpp
struct S{
   int f();
   struct Inner{
      int i;
   };
   S():f(10), Inner(0){}   // C2436
};
```

Compiler Error C2437

Error Message
'identifier' : already initialized

An object can be initialized only once.

The following sample generates C2437:

```
// C2437.cpp
// compile with: /c
class A {
public:
   A(int i) {}
};

class B : A {
   B() : A(1), A(2) {}   // C2437
   B() : A(1) {}   // OK
};
```

Compiler Error C2438

Error Message
'identifier' : cannot initialize static class data via constructor

A constructor is used to initialize a static member of a class. Static members must be initialized in a definition outside the class declaration.

The following sample generates C2438:

```
// C2438.cpp
struct X {
   X(int i) : j(i) {}   // C2438
   static int j;
};

int X::j;

int main() {
   X::j = 1;
}
```

Compiler Error C2439

Error Message
'identifier' : member could not be initialized

A class, structure, or union member cannot be initialized.

To fix by checking the following possible causes

1. Trying to initialize an indirect base class or structure.
2. Trying to initialize an inherited member of a class or structure. An inherited member must be initialized by the constructor of the class or structure.

Compiler Error C2440

Error Message
'conversion' : cannot convert from 'type1' to 'type2'

The compiler cannot cast from '*type1*' to '*type2*'.

C2440 can occur as the result of conformance and update work that was done in the Standard C++ Library. See Standard C++ Library Changes: Visual C++ .NET 2003 and Library Changes in Visual C++ Releases for more information.

Example

C2440 can be caused if you attempt to convert a pointer to member to void*. The following sample generates C2440.

```
// C2440.cpp
class B {
public:
   void  f(){;}

   typedef void (B::*pf)();

   void f2(pf pf) {
       (this->*pf)();
       void* pp = (void*)pf;   // C2440
   }

   void f3() {
      f2(f);
   }
};
```

The C2440 errors on lines 15 and 16 of the next sample will be qualified with the `Incompatible calling conventions for UDT return value` message. (A UDT is a user-defined type, such as a class, struct, or union.) These types of incompatibility errors are caused when the calling convention of a UDT specified in the return type of a forward declaration conflicts with the actual calling convention of the UDT and when a function pointer is involved.

In the example, we first have forward declarations for a struct and for a function that returns the struct; the compiler assumes that the struct uses the C++ calling convention. Then we have the struct definition, which, by default, uses the C calling convention. Since the compiler does not know the calling convention of the struct until it finishes reading the entire struct, the calling convention for the struct in the return type of get_c2 is also assumed to be C++.

The struct is followed by another function declaration that returns the struct, but at this point, the compiler knows that the struct's calling convention is C++. Similarly, the function pointer, which returns the struct, is defined after the struct definition so the compiler knows that the struct uses the C++ calling convention.

To resolve C2440 because of incompatible calling conventions, declare functions that return a UDT after the UDT definition.

```
// C2440b.cpp
struct MyStruct;

MyStruct get_c1();

struct MyStruct {
   int i;
   static MyStruct get_C2();
};

MyStruct get_C3();

typedef MyStruct (*FC)();

FC fc1 = &get_c1;   // C2440, line 15
FC fc2 = &MyStruct::get_C2;   // C2440, line 16
FC fc3 = &get_C3;
```

```
class CMyClass {
public:
   explicit CMyClass( int iBar)
      throw()   {
   }

   static CMyClass get_c2();
};

int main() {
   CMyClass myclass = 2;   // C2440
   // try one of the following
   // CMyClass myclass(2);
   // CMyClass myclass = (CMyClass)2;

   int *i;
   float j;
   j = (float)i;   // C2440, cannot cast from pointer to int to float
}
```

C2440 can also occur if you assign zero to an interior pointer:

```
// C2440c.cpp
// compile with: /clr
int main() {
   array<int>^ arr = gcnew array<int>(100);
   interior_ptr<int> ipi = &arr[0];
   ipi = 0;   // C2440
   ipi = nullptr;   // OK
}
```

C2440 can also occur for an incorrect use of a user-defined conversion. For more information on user-defined conversions, see User-Defined Conversions). The following sample generates C2440.

```
// C2440d.cpp
// compile with: /clr
value struct MyDouble {
   double d;
   // convert MyDouble to Int32
   static explicit operator System::Int32 ( MyDouble val ) {
      return (int)val.d;
   }
};

int main() {
   MyDouble d;
   int i;
   i = d;   // C2440
   // Uncomment the following line to resolve.
   // i = static_cast<int>(d);
}
```

C2440 can also occur if you try to create an instance of a Visual C++ array whose type is a Array. For more information, see array (Visual C++). The following sample generates C2440.

```
// C2440e.cpp
// compile with: /clr
using namespace System;
int main() {
   array<int>^ intArray = Array::CreateInstance(__typeof(int), 1);   // C2440
   // try the following line instead
   // array<int>^ intArray = safe_cast<array<int> ^>(Array::CreateInstance(__typeof(int), 1
));
}
```

C2440 can also occur because of changes in the attributes feature. The following sample generates C2440.

```
// c2440f.cpp
// compile with: /LD
[ module(name="PropDemoLib", version=1.0) ];   // C2440
// try the following line instead
// [ module(name="PropDemoLib", version="1.0") ];
```

This error can also be generated as a result of compiler conformance work that was done for Visual C++ 2005: the Visual C++ compiler no longer allows the const_cast Operator to down cast when compiling source code that uses /clr programming.

To resolve this C2440, use the correct cast operator (for more information, see Casting Operators). See Breaking Changes in the Visual C++ 2005 Compiler for more information.

The following sample generates C2440.

```
// c2440g.cpp
// compile with: /clr
ref class Base {};
ref class Derived : public Base {};
int main() {
   Derived ^d = gcnew Derived;
   Base ^b = d;
   d = const_cast<Derived^>(b);   // C2440
   d = dynamic_cast<Derived^>(b);   // OK
}
```

C2440 can also be generated using **/clr:oldSyntax**. The following sample generates C2440.

```
// c2440h.cpp
// compile with: /clr:oldSyntax
__gc class Base {};
__gc class Derived : public Base {};
int main() {
   Derived *d = new Derived;
   Base *b = d;
   d = const_cast<Derived*>(b);   // C2440
   d = dynamic_cast<Derived*>(b);   // OK
}
```

Compiler Error C2441

Error Message
'variable' : a symbol declared with __declspec(process) must be const in /clr:pure mode

By default, variables are per application domain under **/clr:pure**. A variable marked **__declspec(process)** under **/clr:pure** is prone to errors if modified in one application domain and read in another.

Therefore, the compiler enforces per process variables be **const** under **/clr:pure**, making them read only in all application domains.

For more information, see process and /clr (Common Language Runtime Compilation).

Example
The following sample generates C2441.

```
// C2441.cpp
// compile with: /clr:pure /c
__declspec(process) int i;   // C2441
__declspec(process) const int j = 0;   // OK
```

Compiler Error C2443

Error Message
operand size conflict

The instruction requires operands to be the same size.

The following sample generates C2443:

```
// C2443.cpp
// processor: x86
short var;
int main() {
   __asm xchg ax,bl   // C2443
   __asm mov al,red   // C2443
   __asm mov al,BYTE PTR var   // OK
}
```

Compiler Error C2444

Error Message
'identifier' : used ANSI prototype, found 'type', expected '{' or ';'

The function prototype is followed by a type.

This error can be caused by a missing semicolon or curly brace.

Compiler Error C2446

Error Message

'operator' : no conversion from 'type1' to 'type2'

The compiler cannot convert *type1* to *type2*. The conversion may not make sense because it violates C/C++ semantics.

Compiler Error C2447

Error Message
'{' : missing function header (old-style formal list?)

An open brace occurs at global scope without a corresponding function header.

This error can be caused by an old-style C-language formal list.

The following sample generates C2447:

```
// C2447.cpp
int c;
{}       // C2447
```

Compiler Error C2448

Error Message
'identifier' : function-style initializer appears to be a function definition

The function definition is incorrect.

This error can be caused by an old-style C-language formal list.

The following sample generates C2448:

```
// C2448.cpp
void func(c)
   int c;
{}   // C2448
```

Compiler Error C2449

Error Message
found '{' at file scope (missing function header?)

An open brace occurs at file scope.

This error can be caused by a semicolon between a function header and the opening brace of the function definition.

The following sample generates C2499:

```
// C2449.c
// compile with: /c
void __stdcall func(void) {}   // OK
void __stdcall func(void);  // extra semicolon on this line
{                          // C2449 detected here
```

Compiler Error C2450

Error Message
switch expression of type 'type' is illegal

The **switch** expression evaluates to an invalid type. It must evaluate to an integer type or a class type with unambiguous conversion to an integer type. If it evaluates to a user-defined type, you must supply a conversion operator.

The following sample generates C2450:

```
// C2450.cpp
class X {
public:
   int i;
} x;

class Y {
public:
   int i;
   operator int() { return i; }   // conversion operator
} y;

int main() {
   int j = 1;
   switch ( x ) {   // C2450, x is not type int
   // try the following line instead
   // switch ( y ) {
      default:  ;
   }
}
```

Compiler Error C2451

Error Message
conditional expression of type 'type' is illegal

The conditional expression evaluates to an integer type.

The following sample generates C2451:

```
// C2451.cpp
class B {};

int main () {
   B b1;
   int i = 0;
   if (b1)   // C2451
   // try the following line instead
   // if (i)
      ;
}
```

Compiler Error C2452

Error Message
'type' : invalid source type for safe_cast

The source type for safe_cast was not valid. For example, all types in a **safe_cast** operation must be CLR types.

The following sample generates C2452:

```
// C2452.cpp
// compile with: /clr

struct A {};
struct B : public A {};

ref struct C {};
ref struct D : public C{};

int main() {
   A a;
   safe_cast<B*>(&a);   // C2452

   // OK
   C ^ c = gcnew C;
   safe_cast<D^>(c);
}
```

Compiler Error C2457

Error Message

'macro': predefined macro cannot appear outside of a function body

You attempted to use a predefined macro, such as __FUNCTION__, in a global space.

Example

The following sample generates C2457:

```
// C2457.cpp
#include <stdio.h>

__FUNCTION__;   // C2457, cannot be global

int main()
{
    printf_s("\n%s",__FUNCTION__);   // OK
}
```

Compiler Error C2458

Error Message
'identifier' : redefinition within definition

A class, structure, union, or enumeration is redefined in its own declaration.

The following sample generates C2458:

```
// C2458.cpp
class C {
   enum i { C };   // C2458
};
```

Compiler Error C2459

Error Message
'identifier' : is being defined; cannot add as an anonymous member

A class, structure, or union is redefined in its own scope by a member of an anonymous union.

The following sample generates C2459:

```
// C2459.cpp
// compile with: /c
class C {
   union { int C; };   // C2459
   union { int D; };
};
```

Compiler Error C2460

Error Message
'identifier1' : uses 'identifier2', which is being defined

A class or structure (*identifier2*) is declared as a member of itself (*identifier1*). Recursive definitions of classes and structures are not allowed.

The following sample generates C2460:

```
// C2460.cpp
class C {
   C aC;    // C2460
};
```

Compiler Error C2461

Error Message
'class' : constructor syntax missing formal parameters

The constructor for the class does not specify any formal parameters. The declaration of a constructor must specify a formal parameter. (The list can be null.)

Add a pair of parentheses after *class*::*class*.

The following sample generates C2461:

```
// C2461.cpp
// compile with: /c
class C {
   C::C;   // C2461
   C::C();   // OK
};
```

Compiler Error C2462

Error Message
'identifier' : cannot define a type in a 'new-expression'

You cannot define a type in the operand field of the **new** operator. Put the type definition in a separate statement.

The following sample generates C2462:

```
// C2462.cpp
int main() {
   new struct S { int i; };   // C2462
}
```

Compiler Error C2463

Error Message
cannot define an anonymous type in a 'new-expression'

You cannot define an anonymous type in the operand field of the **new** operator. Create a named type definition in a separate statement, then use the **new** operator.

Compiler Error C2464

Error Message
'identifier' : cannot use 'new' to allocate a reference

A reference identifier was allocated with the **new** operator. References are not memory objects, so **new** cannot return a pointer to them. Use the standard variable declaration syntax to declare a reference.

The following sample generates C2464:

```
// C2464.cpp
int main() {
   new ( int& ir );   // C2464
}
```

Compiler Error C2465

Error Message
cannot define an anonymous type inside parentheses

An anonymous structure, union, or enumerated type is defined inside a parenthetical expression. This is invalid in C++ because the definition is meaningless in function scope.

Compiler Error C2466

Error Message
cannot allocate an array of constant size 0

An array is allocated or declared with size zero. The constant expression for the array size must be an integer greater than zero. An array declaration with a zero subscript is legal only for a class, structure, or union member and only with Microsoft extensions (/Ze).

The following sample generates C2466:

```
// C2466.cpp
// compile with: /c
int i[0];   // C2466
int j[1];   // OK
char *p;
```

Compiler Error C2467

Error Message
illegal declaration of anonymous 'user-defined-type'

A nested user-defined type was declared. This is an error when compiling C source code with the ANSI compatibility option (/Za) enabled.

The following sample generates C2467:

```
//C2467.c
// compile with: /Za
int main() {
   struct X {
      union { int i; };   // C2467, nested declaration
   };
}
```

Compiler Error C2469

Error Message
'operator': cannot allocate 'type' object

An operator was passed an invalid type.

The following sample generates C2469:

```
// C2469.cpp
int main() {
   int *i = new void;   // C2469
   int *i = new int;   // OK
}
```

Compiler Error C2470

Error Message
'function' : looks like a function definition, but there is no parameter list; skipping apparent body

A function definition is missing its argument list.

The following sample generates C2470:

```
// C2470.cpp
int MyFunc {};  // C2470
void MyFunc2() {};  //OK

int main(){
   MyFunc();
   MyFunc2();
}
```

Compiler Error C2471

Error Message
cannot update program database 'file'

The compiler cannot write to the database file.

To fix by checking the following possible causes

1. Insufficient disk space.
2. Volume is read-only.
3. Sharing violation.

Compiler Error C2472

Error Message
'function' cannot be generated in managed code: 'message'; compile with /clr to generate a mixed image

This error will occur when types not supported by managed code are used within a pure common language runtime (CLR) environment. Compile with **/clr** to resolve the error.

Example
The following sample generates C2472.

```
// C2472.cpp
// compile with: /clr:pure
// C2472 expected

#include <cstdlib>

int main()
{
   int * __ptr32 p32;
   int * __ptr64 p64;

   p32 = (int * __ptr32)malloc(4);
   p64 = p32;
}
```

See Also
Reference
/clr (Common Language Runtime Compilation)

Compiler Error C2473

Error Message

'identifier' : looks like a function definition, but there is no parameter list.

The compiler detected what looked like a function, without the parameter list.

Example

The following sample generates C2473.

```
// C2473.cpp
// compile with: /clr /c
class A {
   int i {}   // C2473
};

class B {
   int i() {}   // OK
};
```

Compiler Error C2474

Error Message

'keyword' : missing an adjacent semicolon, could be either keyword or identifier.

The compiler expected to find a semicolon, and was unable to determine your intent. Add the semicolon to resolve this error.

Example

The following sample generates C2474.

```
// C2474.cpp
// compile with: /clr /c

ref class A {
   ref class B {}
   property int i;   // C2474 error
};

// OK
ref class B {
   ref class D {};
   property int i;
};

ref class E {
   ref class F {} property;
   int i;
};
```

Compiler Error C2477

Error Message
'member' : static data member cannot be initialized via derived class

A static data member of a template class was initialized incorrectly. This is a breaking change with versions of the Visual C++ compiler prior to Visual Studio .NET 2003, in order to conform to the ISO C++ standard.

See Summary of Compile-Time Breaking Changes for more information.

The following sample generates C2477:

```
// C2477.cpp
// compile with: /Za /c
template <class T>
struct S {
   static int n;
};

struct X {};
struct A: S<X> {};

int A::n = 0;   // C2477

template<>
int S<X>::n = 0;
```

Compiler Error C2479

Error Message
'identifier' : 'allocate( )' is only valid for data items of static extent

The **__declspec(allocate())** syntax can be used for static data only.

The following sample generates C2479:

```
// C2479.cpp
// compile with: /c
#pragma section("mycode", read)
static __declspec(allocate("mycode")) void DoNothing() {}   // C2479
__declspec(allocate("mycode"))  int i = 0;   // OK
```

Compiler Error C2480

Error Message
'identifier' : 'thread' is only valid for data items of static extent

You cannot use the **thread** attribute with an automatic variable, nonstatic data member, function parameter, or on function declarations or definitions.

Use the **thread** attribute for global variables, static data members, and local static variables only.

The following sample generates C2480:

```
// C2480.cpp
// compile with: /c
__declspec( thread ) void func();   // C2480
__declspec( thread ) static int i;   // OK
```

Compiler Error C2482

Error Message
'identifier' : dynamic initialization of 'thread' data not allowed

Variables declared with the **thread** attribute cannot be initialized with an expression that requires run-time evaluation. A static expression is required to initialize **thread** data.

The following sample generates C2482:

```
// C2482.cpp
// compile with: /c
#define Thread __declspec( thread )
Thread int tls_i = tls_i;   // C2482

int j = j;   // OK in C++; C error
Thread int tls_i = sizeof( tls_i );   // Okay in C and C++
```

Compiler Error C2483

Error Message
'identifier' : object with constructor or destructor cannot be declared 'thread'

Variables declared with the **thread** attribute cannot be initialized with a constructor or other expression that requires run-time evaluation. A static expression is required to initialize **thread** data.

Example
The following sample generates C2483.

```
// C2483.cpp
// compile with: /c
__declspec(thread) struct A {
   A(){}
   ~A(){}
} aa;   // C2483 error

__declspec(thread) struct B {} b;   // OK
```

Compiler Error C2485

Error Message

'identifier' : unrecognized extended attribute

The declaration attribute is not valid.

Compiler Error C2486

Error Message
'__LOCAL_SIZE' only allowed in function with the 'naked' attribute

In inline assembly functions, the name **__LOCAL_SIZE** is reserved for functions declared with the naked attribute.

The following sample generates C2486:

```
// C2486.cpp
// processor: x86
void __declspec(naked) f1() {
   __asm {
      mov   eax,   __LOCAL_SIZE
   }
}
void f2() {
   __asm {
      mov   eax,   __LOCAL_SIZE   // C2486
   }
}
```

Compiler Error C2487

Error Message

'identifier' : member of dll interface class may not be declared with dll interface

You can declare a whole class, or certain members of a non-DLL interface class, with DLL interface. You cannot declare a class with DLL interface and then declare a member of that class with DLL interface.

Compiler Error C2488

Error Message
'identifier' : 'naked' can only be applied to non-member function definitions

The naked attribute was applied to a function declaration.

The following sample generates C2488:

```
// C2488.cpp
// compile with: /c
// processor: x86
__declspec( naked ) void func();   // C2488  declaration, not definition
__declspec( naked ) void i;   // C2488  i is not a function

__declspec( naked ) void func() {}   // OK
```

Compiler Error C2489

Error Message

'identifier' : initialized auto or register variable not allowed at function scope in 'naked' function

For more information, see naked (C++).

The following sample generates C2489:

```
// C2489.cpp
// processor: x86
__declspec( naked ) int func() {
   int i = 1;   // C2489
   register int j = 1;   // C2489
}
```

Compiler Error C2490

Error Message
'keyword' not allowed in function with 'naked' attribute

A function defined as naked cannot use structured exception handling.

The following sample generates C2490:

```
// C2490.cpp
// processor: x86
__declspec( naked ) int func() {
   __try{}   // C2490, structured exception handling
}
```

Compiler Error C2491

Error Message
'identifier' : definition of dllimport function not allowed

Data, static data members, and functions can be declared as **dllimport**s but not defined as **dllimport**s.

The following sample generates C2491:

```
// C2491.cpp
// compile with: /c
// function definition
void __declspec(dllimport) funcB() {}   // C2491

// function declaration
void __declspec(dllimport) funcB();   // OK
```

Compiler Error C2492

Error Message
'variable' : 'thread' data may not have dll interface

The variable is declared with the thread attribute and with the DLL interface. The address of the **thread** variable is not known until run time, so it cannot be linked to a DLL import or export.

The following sample generates C2492:

```
// C2492.cpp
// compile with: /c
class C {
public:
   char   ch;
};

__declspec(dllexport) __declspec(thread) C c_1;   // C2492
__declspec(thread) C c_1;   // OK
```

Compiler Error C2493

Error Message
illegal form of __based

A **__based** expression must be based on a pointer.

The following sample generates C2493:

```
// C2493.cpp
// compile with: /c
char mybase;
int __based(mybase) ptr;   // C2493

// OK
char * mybase;
int __based(mybase) * ptr;
```

Compiler Error C2494

Error Message
'keyword' cannot be called from within a filter expression or __finally/finally block

You cannot use *keyword* in a **__finally** or finally block.

The following sample generates C2494:

```
// C2494.cpp
#include <malloc.h>

int main() {
   __try {}
   __except ( _alloca(100), 1 ) {}   // C2494
   __try {}
   __finally {
      _alloca(100);   // C2494
   }
}
```

C2494 can also occur when using **/clr**.

```
// C2494b.cpp
// compile with: /clr
#include <malloc.h>

int main() {
   char * buf;
   try {}
   catch (char * buf2) {}
   finally {
      _alloca(100);   // C2494
   }
}
```

Compiler Error C2495

Error Message
'identifier' : 'nothrow' can only be applied to function declarations or definitions

The nothrow extended attribute can be applied to function declarations or definitions only.

The following sample generates C2495:

```
// C2495.cpp
// compile with: /c
__declspec(nothrow) class X {   // C2495
   int m_data;
} x;

__declspec(nothrow) void test();   // OK
```

Compiler Error C2496

Error Message
'identifier' : 'selectany' can only be applied to data items with external linkage

The selectany attribute can be applied only to externally visible and global data items.

The following sample generates C2496:

```
// C2496.cpp
// compile with: /c
__declspec(selectany) int x1 = 1;
const __declspec(selectany) int x2 = 2;   // C2496
static __declspec(selectany) int x6 = 6;   // C2496

extern const __declspec(selectany) int x3 = 3;

__declspec(selectany) int x4;

// dynamic initialization of x5
int f();
__declspec(selectany) int x5 = f();

extern const int x7;
// OK - redeclaration of x7 that is extern
const __declspec(selectany) int x7 = 7;
```

Compiler Error C2498

Error Message

'function' : 'novtable' can only be applied to class declarations or definitions

This error can be caused by using **__declspec(novtable)** with a function.

Example

The following sample generates C2498:

```
// C2498.cpp
// compile with: /c
void __declspec(novtable) f() {}   // C2498
class __declspec(novtable) A {};   // OK
```

Compiler Error C2499

Error Message
'class' : a class cannot be its own base class

You attempted to specify the class that you are defining as a base class.

The following sample generates C2499:

```
// C2499.cpp
// compile with: /c
class CMyClass : public CMyClass {};   // C2499
class CMyClass{};   // OK
```

Compiler Errors C2500 Through C2599

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, type the error number (C2521, for example) in the index, or click a link below.

Compiler Error C2500

Compiler Error C2502

Compiler Error C2503

Compiler Error C2504

Compiler Error C2505

Compiler Error C2506

Compiler Error C2507

Compiler Error C2509

Compiler Error C2510

Compiler Error C2511

Compiler Error C2512

Compiler Error C2513

Compiler Error C2514

Compiler Error C2516

Compiler Error C2517

Compiler Error C2518

Compiler Error C2521

Compiler Error C2523

Compiler Error C2524

Compiler Error C2526

Compiler Error C2528

Compiler Error C2529

Compiler Error C2530

Compiler Error C2531

Compiler Error C2532

Compiler Error C2533

Compiler Error C2534

Compiler Error C2535

Compiler Error C2536

Compiler Error C2537

Compiler Error C2538

Compiler Error C2539

Compiler Error C2540

Compiler Error C2541

Compiler Error C2542

Compiler Error C2543
Compiler Error C2544
Compiler Error C2545
Compiler Error C2548
Compiler Error C2549
Compiler Error C2550
Compiler Error C2551
Compiler Error C2552
Compiler Error C2553
Compiler Error C2555
Compiler Error C2556
Compiler Error C2557
Compiler Error C2558
Compiler Error C2561
Compiler Error C2562
Compiler Error C2563
Compiler Error C2564
Compiler Error C2566
Compiler Error C2567
Compiler Error C2568
Compiler Error C2569
Compiler Error C2570
Compiler Error C2571
Compiler Error C2572
Compiler Error C2573
Compiler Error C2574
Compiler Error C2575
Compiler Error C2577
Compiler Error C2579
Compiler Error C2581
Compiler Error C2582
Compiler Error C2583
Compiler Error C2584
Compiler Error C2585
Compiler Error C2586
Compiler Error C2587
Compiler Error C2588
Compiler Error C2589
Compiler Error C2592
Compiler Error C2593

Compiler Error C2594

Compiler Error C2597

Compiler Error C2598

Compiler Error C2599

Compiler Error C2500

Error Message
'identifier1' : 'identifier2' is already a direct base class

A class or structure appears more than once in a list of base classes.

A direct base is one mentioned in the base list. An indirect base is a base class of one of the classes in the base list.

A class cannot be specified as a direct base class more than once. A class can be used as an indirect base class more than once.

The following sample generates C2500:

```
// C2500.cpp
// compile with: /c
class A {};
class B : public A, public A {};    // C2500

// OK
class C : public A {};
class D : public A {};
class E : public C, public D {};
```

Compiler Error C2502

Error Message
'identifier' : too many access modifiers on the base class

The base class has more than one access modifier. Only one access modifier (**public**, **private**, or **protected**) is allowed.

The following sample generates C2502:

```
// C2502.cpp
// compile with: /c
class A { };
class B { };
class C : private public A { };   // C2502

// OK
class D : private A {};
class E : public A, private B {};
```

Compiler Error C2503

Error Message
'class' : base classes cannot contain zero-sized arrays

A base class or structure contains a zero-sized array. An array in a class must have at least one element.

The following sample generates C2503:

```
// C2503.cpp
// compile with: /c
class A {
   public:
   int array [];
};

class B : A {};    // C2503

class C {
public:
   int array [10];
};

class D : C {};
```

Compiler Error C2504

Error Message
'class' : base class undefined

The base class is declared but never defined. Possible causes:

1. Missing include file.
2. External base class not declared with extern.

The following sample generates C2504:

```
// C2504.cpp
// compile with: /c
class A;
class B : public A {};   // C2504
```

// OK

```
class C{};
class D : public C {};
```

Compiler Error C2505

Error Message
'symbol' : '__declspec(modifer)' can only be applied to declarations or definitions of global objects or static data members

A **__declspec** modifier that is designed to only be used at global scope was used in a function.

For more information, see appdomain and process.

The following sample generates C2505:

```
// C2505.cpp
// compile with: /clr

// OK
__declspec(process) int ii;
__declspec(appdomain) int jj;

int main() {
   __declspec(process) int i;   // C2505
   __declspec(appdomain) int j;   // C2505
}
```

Compiler Error C2506

Error Message
'member' : '__declspec(modifier)' cannot be applied to this symbol

You cannot declare per-process or per-appdomain for static members of a managed class.

See appdomain for more information.

Example
The following sample generates C2506.

```
// C2506.cpp
// compile with: /clr /c
ref struct R {
   __declspec(process) static int n;   // C2506
   int o;   // OK
};
```

Compiler Error C2507

Error Message
'identifier' : too many virtual modifiers on the base class

A class or structure is declared as **virtual** more than once. Only one **virtual** modifier can appear for each class in a list of base classes.

The following sample generates C2507:

```
// C2507.cpp
// compile with: /c
class A {};
class B : virtual virtual public A {};   // C2507
class C : virtual public A {};   // OK
```

Compiler Error C2509

Error Message
'identifier' : member function not declared in 'class'

The function is not declared in the specified class.

Example
The following sample generates C2509.

```
// C2509.cpp
// compile with: /c
struct A {
   virtual int vfunc() = 0;
   virtual int vfunc2() = 0;
};

struct B : private A {
   using A::vfunc;
   virtual int vfunc2();
};

int B::vfunc() { return 1; }   // C2509
int B::vfunc2() { return 1; }   // OK
```

Compiler Error C2510

Error Message
'identifier' : left of '::' must be a class/struct/union

A class, structure, or union name must appear on the left side of the scope-resolution operator (**::**) operator.

Compiler Error C2511

Error Message
'identifier' : overloaded member function not found in 'class'

No version of the function is declared with the specified parameters. Possible causes:

1. Wrong parameters passed to function.
2. Parameters passed in wrong order.
3. Incorrect spelling of parameter names.

The following sample generates C2511:

```
// C2511.cpp
// compile with: /c
class C {
   int c_2;
   int Func(char *, char *);
};

int C::Func(char *, char *, int i) {   // C2511
// try the following line instead
// int C::Func(char *, char *) {
   return 0;
}
```

Compiler Error C2512

Error Message
'identifier' : no appropriate default constructor available

No default constructor is available for the specified class, structure, or union. The compiler supplies a default constructor if user-defined constructors are not provided.

If you provide a constructor that takes a non-void parameter, and you want to allow your class to be created with no parameters, you must also provide a default constructor. The default constructor can be a constructor with default values for all parameters.

The following sample generates C2512:

```
// C2512.cpp
// C2512 expected
struct B {
   B (char *);
   // Uncomment the following line to resolve.
   // B() {};
};

int main() {
   B b;
}
```

The following sample shows a more subtle C2512:

```
// C2512b.cpp
// compile with: /c
struct S {
   struct X;

   void f() {
      X *x = new X();   // C2512 X not defined yet
   }

};

struct S::X {};

struct T {
   struct X;
   void f();
};

struct T::X {};

void T::f() {
   X *x = new X();
}
```

Compiler Error C2513

Error Message
'type' : no variable declared before '='

The type specifier appears in declaration with no variable identifier.

The following sample generates C2513:

```
// C2513.cpp
int main() {
   int = 9;   // C2513
   int i = 9;   // OK
}
```

This error can also be generated as a result of a compiler conformance work done for Visual Studio .NET 2003: initialization of a typedef no longer allowed. The initialization of a typedef is not allowed by the standard and now generates a compiler error.

See Summary of Compile-Time Breaking Changes for more information.

```
// C2513b.cpp
// compile with: /c
typedef struct S {
   int m_i;
} S = { 1 };   // C2513
// try the following line instead
// } S;
```

An alternative would be to delete `typedef` to define a variable with aggregate initializer list, but this is not recommended because it will create a variable with the same name as the type and hide the type name.

Compiler Error C2514

Error Message
'class' : class has no constructors

The class, structure, or union has no constructor with a parameter list that matches the parameters being used to instantiate it.

A class must be fully declared before it can be instantiated.

The following sample generates C2514:

```
// C2514.cpp
// compile with: /c
class f;

class g {
public:
    g (int x);
};

class fmaker {
   f *func1() {
      return new f(2);   // C2514
   }

   g *func2() {
      return new g(2);   // OK
   }
};
```

Compiler Error C2516

Error Message
'name' : is not a legal base class

The class is derived from a type name defined by a **typedef** statement.

The following sample generates C2516:

```
// C2516.cpp
typedef unsigned long ulong;
class C : public ulong {}; // C2516
```

Compiler Error C2517

Error Message
'identifier' : right of '::' is undefined

The identifier on the right of the scope-resolution operator (**::**) must be a defined member of the class, structure, or union on the left. If no class, structure, or union is named, the identifier on the right must be declared with global scope.

Compiler Error C2518

Error Message
keyword 'keyword' illegal in base class list; ignored

The keywords **class** and **struct** should not appear in a base class list.

The following sample generates C2518:

```
// C2518.cpp
// compile with: /c
class B {};
class C : public class B {};   // C2518
class D: public B {};   // OK
```

Compiler Error C2521

Error Message
function does not take any arguments

You attempted to use arguments with a destructor or finalizer.

For more information, see Destructors and Finalizers in Visual C++.

Example
The following sample generates C2521.

```
// C2521.cpp
// compile with: /clr
ref class R {
protected:
   !R() {}

public:
   void CleanUp() {
      this->!R(4);   // C2521
      this->!R();   // OK
   }
};

int main() {
   R^ r = gcnew R();
   r->CleanUp();
}
```

Compiler Error C2523

Error Message
'class::~identifier' : destructor/finalizer tag mismatch

The name of the destructor must be the class name preceded by a tilde (**~**). The constructor and destructor are the only members that have the same name as the class.

The following sample generates C2523:

```
// C2523.cpp
// compile with: /c
class A {
   ~B();    // C2523
   ~A();   // OK
};
```

Compiler Error C2524

Error Message
'destructor' : a destructor/finalizer must have a 'void' parameter list

The destructor or finalizer had a parameter list that is not void. Other parameter types are not allowed.

Example
The following code reproduces C2524.

```
// C2524.cpp
// compile with: /c
class A {
   A() {}
   ~A(int i) {}    // C2524
   // try the following line instead
   // ~A() {}
};
```

The following code reproduces C2524.

```
// C2524_b.cpp
// compile with: /clr /c
ref struct I1 {
protected:
   !I1(int i);   // C2524
   // try the following line instead
   // !I1();
};
```

Compiler Error C2526

Error Message

'identifier1' : C linkage function cannot return C++ class 'identifier2'

A function defined with C linkage cannot return a user-defined type.

Compiler Error C2528

Error Message
'name' : pointer to reference is illegal

You cannot declare a pointer to a reference. Dereference the variable before declaring a pointer to it.

The following sample generates C2528:

```
// C2528.cpp
int i;
int &ir = i;
int & (*irptr) = ir;    // C2528
```

Compiler Error C2529

Error Message
'name' : reference to reference is illegal

This error may be fixed by using pointer syntax and declaring a reference to a pointer.

The following sample generates C2529:

```
// C2529.cpp
// compile with: /c
int i;
int &ri = i;
int &(&rri) = ri;   // C2529
```

Compiler Error C2530

Error Message
'identifier' : references must be initialized

You must initialize a reference when it was declared, unless it is declared already:

- With the keyword extern.
- As a member of a class, structure, or union (and it is initialized in the constructor).
- As a parameter in a function declaration or definition.
- As the return type of a function.

The following sample generates C2530:

```
// C2530.cpp
int main() {
   int i = 0;
   int &j;   // C2530
   int &k = i;   // OK
}
```

Compiler Error C2531

Error Message
'identifier' : reference to a bit field illegal

References to bit fields are not allowed.

The following sample generates C2531:

```
// C2531.cpp
// compile with: /c
class P {
   int &b1 : 10;   // C2531
   int b2 : 10;   // OK
};
```

Compiler Error C2532

Error Message
'identifier' : illegal modifier for reference

The reference was changed. References cannot be modified to refer to another object. Use a pointer instead.

Compiler Error C2533

Error Message
'identifier' : constructors not allowed a return type

A constructor cannot have a return type (not even a **void** return type).

The following sample generates C2533:

```
// C2533.cpp
// compile with: /c
class X {
public:
   X();
};

int X::X() {}   // C2533
X::X(){}   // OK
```

Compiler Error C2534

Error Message
'identifier' : constructor cannot return a value

A constructor cannot return a value or have a return type (not even a **void** return type).

This error may be fixed by removing the **return** statement from the constructor definition.

The following sample generates C2534:

```
// C2534.cpp
class A {
public:
   int i;
   A() { return i; }   // C2534
};
```

Compiler Error C2535

Error Message
'identifier' : member function already defined or declared

This error could be caused by using the same formal parameter list in more than one definition or declaration of an overloaded function.

If you get C2535 because of the Dispose function, see Destructors and Finalizers in Visual C++ for more information.

If you are compiling an ATL project, see Knowledge Base article Q241852.

The following sample generates C2535:

```
// C2535.cpp
// compile with: /c
class C {
public:
   void func();   // forward declaration
   void func() {}   // C2535
};
```

Compiler Error C2536

Error Message
'class::identifier' : cannot specify explicit initializer for arrays

A member of a class, structure, or union could not be initialized. Possible causes:

1. A constructor is not available to initialize one or more members of an array. If the size of the array is greater than the number of initializers, a default constructor must be defined.

2. A nonstatic array declared with the **const** specifier. This kind of array cannot be explicitly initialized.

The following sample generates C2536:

```
// C2536.cpp
// compile with: /c
class C {
   int i;
   int j;
   int k;
};

class D {
   C aC[5];
   D() : aC(1,2,3,4,5) {}   // C2536
   // try the following line instead
   // D() {}
};
```

Compiler Error C2537

Error Message
'specifier' : illegal linkage specification

Possible causes:

1. The linkage specifier is not supported. Only the "C" linkage specifier is supported.
2. "C" linkage is specified for more than one function in a set of overloaded functions. This is not allowed.

The following sample generates C2537:

```
// C2537.cpp
// compile with: /c
extern "c" void func();   // C2537
extern "C" void func2();   // OK
```

Compiler Error C2538

Error Message
new : cannot specify initializer for arrays

An initializer is given for an array created with the **new** operator. The **new** operator creates arrays of objects by calling the default constructor for each element of the array. The elements of the array cannot be initialized to distinct values.

Compiler Error C2539

Error Message

new : 'class' : no default constructor to initialize arrays of objects

Initializing an array of objects requires a default constructor (a constructor with no parameters), which is called separately for each object in the array. The constructor was not available. If any constructor is defined, the compiler does not generate a default constructor and you must supply one.

Compiler Error C2540

Error Message
non-constant expression as array bound

An array must have a constant bound.

The following sample generates C2540:

```
// C2540.cpp
void func(int n, int pC[]) {
   int i = ((int [n])pC)[1];   // C2540
}

void func2(int n, int pC[]) {
   int i = (pC)[1];   // OK
}

int main() {
   int pC[100];
   func(100, pC);
   func2(100, pC);
}
```

Compiler Error C2541

Error Message
'delete' : delete : cannot delete objects that are not pointers

The delete operator was used on an object that is not a pointer.

The following sample generates C2541:

```
// C2541.cpp
int main() {
   int i;
   delete i;   // C2541 i not a pointer

   // OK
   int *ip = new int;
   delete ip;
}
```

Compiler Error C2542

Error Message
'identifier' : class object has no constructor for initialization

There is no constructor with a parameter list that matches the initialization.

This error can be caused by incorrect parameters in initialization.

Compiler Error C2543

Error Message
expected ']' for operator '[]'

The subscripting operator is missing a left bracket.

This error can be caused by macro expansion.

Compiler Error C2544

Error Message
expected ')' for operator '()'

The function call operator is missing a left parenthesis.

This error can be caused by macro expansion.

Compiler Error C2545

Error Message
'operator' : unable to find overloaded operator

The operator cannot be used with the operands provided. You must supply an overloaded operator with the required operands.

This error can be caused if operands have incorrect type.

This error may be fixed if you define a conversion operator or constructor that takes a single parameter.

Compiler Error C2548

Error Message
'class::member' : missing default parameter for parameter parameter

The default parameter list is missing a parameter. If you supply a default parameter anywhere in a parameter list, you must define default parameters for all subsequent parameters.

Example
The following sample generates C2548:

```
// C2548.cpp
// compile with: /c
void func( int = 1, int, int = 3);  // C2548

// OK
void func2( int, int, int = 3);
void func3( int, int = 2, int = 3);
```

Compiler Error C2549

Error Message
user-defined conversion cannot specify a return type

The following sample generates C2549:

```
// C2549.cpp
// compile with: /c
class X {
public:
   int operator int() { return value; }   // C2549

   // try the following line instead
   // operator int() { return value; }
private:
   int value;
};
```

Compiler Error C2550

Error Message
'identifier' : constructor initializer lists are only allowed on constructor definitions

A base class initializer list is used on the definition of a function that is not a constructor.

The following sample generates C2550:

```
// C2550.cpp
// compile with: /c
class C {
public:
   C();
};

class D : public C {
public:
   D();
   void func();
};

void D::func() : C() {}  // C2550
D::D() : C() {}   // OK
```

Compiler Error C2551

Error Message
'void *' type needs explicit cast

A **void** pointer is assigned to a nonvoid pointer by implicit conversion. You must use an explicit cast.

Compiler Error C2552

Error Message
'identifier' : non-aggregates cannot be initialized with initializer list

The aggregate identifier was incorrectly initialized.

Aggregates are defined as:

- Arrays
- Classes, structures, and unions that do not have:
    - Constructors
    - Private or protected members
    - Base classes
    - Virtual functions

In addition, Visual C++ does not allow data types in an aggregate that contains constructors.

The following represent the reasons C2552 may fire when an aggregate initialization is attempted on a type:

- The type has one or more user-defined constructors.
- The type has one ore more non-static, private data members.
- The type has one or more virtual functions.
- The type has a base class.
- The type is a ref class or CLR interface.
- The type is a scalar (`int i = {};`)
- The type has a non-fixed dimension array (zero-array) whose elements have destructors.

The following sample generates C2552:

```
// C2552.cpp
// compile with: /clr
#include <string>
using namespace std;

struct Pair_Incorrect {
private:
   string m_name;
   double m_val;
};

struct Pair_Correct1 {
public:
   Pair_Correct1(string name, double val)
      : m_name(name), m_val(val) {}

private:
   string m_name;
   double m_val;
};

struct Pair_Correct2 {
public:
   string m_name;
   double m_val;
};
```

```
int main() {
   // To fix, add a constructor to this class and use it for
   // initializing the data members, see Pair_Correct1 (below)
   // or
   // Do not have any private or protected non-static data members,
   // see Pair_Correct2 (below).  Pair_Correct2 is not recommended in
   // case your object model requires some non-static data members to
   // be private or protected

   string name("John");
   Pair_Incorrect pair1 = { name, 0.0 };   // C2552

   // initialize a CLR immutable value type that has a constructor
   System::DateTime dt = {2001, 4, 12, 22, 16, 49, 844};   // C2552

   Pair_Correct1 pair2( name, 0.0 );
   Pair_Correct1 pair3 = Pair_Correct1( name, 0.0 );
   Pair_Correct2 pair4 = { name, 0.0 };
   System::DateTime dt2(2001, 4, 12, 22, 16, 49, 844);
}
```

Compiler Error C2553

Error Message
'base_function': overriding virtual function return type differs from 'override_function'

A function in a derived class attempted to override a virtual function in a base class, but the derived class function did not have the same return type as the base class function. An override function signature must match the signature of the function being overridden.

The following sample generates C2553:

```
// C2553.cpp
// compile with: /clr /c
ref struct C {
   virtual void f();
};

ref struct D : C {
   virtual int f() override ;   // C2553

   // try the following line instead
   // virtual void f() override;
};
```

Compiler Error C2555

Error Message
' class1::function1': overriding virtual function return type differs and is not covariant from 'class2::function2'

A virtual function and a derived overriding function have identical parameter lists but different return types. An overriding function in a derived class cannot differ from a virtual function in a base class only by its return type.

To resolve this error, cast the return value after the virtual function has been called.

You may also see this error if you compile with /clr. For example, the Visual C++ equivalent to the following C# declaration:

```
Guid[] CheckSources(Guid sourceID, Guid[] carouselIDs);
```

is

```
Guid CheckSources(Guid sourceID, Guid carouselIDs[]) [];
```

For more information on C2555, see Knowledge Base article Q240862.

The following sample generates C2555:

```
// C2555.cpp
// compile with: /c
struct X {
   virtual void func();
};
struct Y : X {
   char func();  // C2555
   void func2();   // OK
};
```

Compiler Error C2556

Error Message
'identifier' : overloaded functions only differ by return type

The overloaded functions have different return types but the same parameter list. Each overloaded function must have a distinct formal parameter list.

The following sample generates C2556:

```
// C2556.cpp
// compile with: /c
class C {
   int func();
   double func();   // C2556
   int func(int i);   // ok parameter lists differ
};
```

Compiler Error C2557

Error Message

'identifier' : private and protected members cannot be initialized without a constructor

Only members and friends can assign a value to a private or protected member. Nonpublic members should be initialized in the class constructor.

Compiler Error C2558

Error Message
'identifier' : no copy constructor available or copy constructor is declared 'explicit'

A copy constructor initializes an object from another object of the same type. (It makes a copy of the object.) The compiler generates a default copy constructor if you do not define any constructors.

To fix by checking the following possible causes

1. Trying to copy a class whose copy constructor is **private**. In most cases, a class with a **private** copy constructor should not be copied. A common programming technique declares a **private** copy constructor to prevent the direct use of a class. The class may be useless by itself or require another class in order to work properly.
2. Trying to copy a class whose copy constructor is explicit. Declaring a copy constructor with **explicit** prevents passing/returning objects of a class to/from functions.

To fix by using the following possible solutions

- If you determine that it is safe to use a class with a **private** copy constructor, derive a new class from the class with a **private** constructor and make a **public** or **protected** copy constructor available in the new class. Use the derived class in place of the original.

For more information, see Compiler Errors when Implementing a CObject-Derived Class.

Compiler Error C2561

Error Message
'identifier' : function must return a value

The function was declared as returning a value, but the function definition does not contain a **return** statement.

This error can be caused by an incorrect function prototype:

1. If the function does not return a value, declare the function with return type void.
2. Check that all possible branches of the function return a value of the type declared in the prototype.
3. C++ functions containing inline assembly routines that store the return value in the `AX` register may need a return statement. Copy the value in `AX` to a temporary variable and return that variable from the function.

The following sample generates C2561:

```
// C2561.cpp
int Test(int x) {
   if (x) {
      return;   // C2561
      // try the following line instead
      // return 1;
   }
   return 0;
}

int main() {
   Test(1);
}
```

Compiler Error C2562

Error Message
'identifier' : 'void' function returning a value

The function is declared as **void** but returns a value.

This error can be caused by an incorrect function prototype.

This error may be fixed if you specify the return type in the function declaration.

The following sample generates C2562:

```
// C2562.cpp
// compile with: /c
void testfunc() {
   int i;
   return i;   // C2562 delete the return to resolve
}
```

Compiler Error C2563

Error Message
mismatch in formal parameter list

The formal parameter list of a function (or a pointer to a function) does not match those of another function (or pointer to a member function). As a result, the assignment of functions or pointers cannot be made.

The following sample generates C2563:

```
// C2563.cpp
void func( int );
void func( int, int );
int main() {
   void *fp();
   fp = func;   // C2563
}
```

Compiler Error C2564

Error Message
'type' : a function-style conversion to a built-in type can only take one argument

Function-style type casts of built-in types take a single argument.

The following sample generates C2564:

```
// C2564.cpp
void g(float f, double d) {
   int j = int(f, d);   // C2564
}
```

Compiler Error C2566

Error Message
overloaded function in conditional expression

An overloaded function in a conditional expression cannot be evaluated.

Compiler Error C2567

Error Message
unable to open metadata in 'file', file may have been deleted or moved

A metadata file that was referenced in source (with **#using**) was not found in the same directory by the compiler back end process as it was by the compiler front end process. See The #using Directive for more information.

C2567 could be caused if you compile with **/c** on one machine and then attempt a link-time code generation on another machine. For more information, see /LTCG (Link-time Code Generation)).

It might also indicate that your computer had no more memory.

To correct this error, make sure that the metadata file is in the same directory location for all phases of the build process.

Compiler Error C2568

Error Message
'identifier1' : unable to resolve function overload

The compiler cannot determine which overloaded function to call. The actual parameters passed to the function must be cast to match the formal parameters for one of the overloaded function, but no one match is unambiguously better than all others.

Compiler Error C2569

Error Message

'EnumOrUnion' : enum/union cannot be used as a base class

If you must derive a type from the specified union or enumeration, change the union or enumeration to a class or structure.

The following sample generates C2569:

```
// C2569.cpp
// compile with: /c
union ubase {};
class cHasPubUBase : public ubase {};   // C2569
// OK
struct sbase {};
class cHasPubUBase : public sbase {};
```

Compiler Error C2570

Error Message
'identifier' : union cannot have base classes

A union derives from a class, structure, or union. This is not allowed. Declare the derived type as a class or structure instead.

The following sample generates C2570:

```
// C2570.cpp
// compile with: /c
class base {};
union hasPubBase : public base {};   // C2570
union hasNoBase {};   // OK
```

Compiler Error C2571

Error Message
'function' : virtual function cannot be in union 'union'

A union is declared with a virtual function. You can declare a virtual function only in a class or structure. Possible resolutions:

1. Change the union to a class or structure.
2. Make the function nonvirtual.

The following sample generates C2571:

```
// C2571.cpp
// compile with: /c
union A {
   virtual void func1();   // C2571
   void func2();   // OK
};
```

Compiler Error C2572

Error Message
'class::member' : redefinition of default parameter : parameter param

Default parameters cannot be redefined. If you require another value for the parameter, the default parameter should be left undefined.

The following sample generates C2572:

```
// C2572.cpp
// compile with: /c
void f(int i = 1);   // function declaration

// function definition
void f(int i = 1) {}   // C2572

// try the following line instead
// void f(int i) {}
```

Compiler Error C2573

Error Message

'class' : cannot delete pointers to objects of this type; the class has no non-placement overload for 'operator delete'.

The class is missing a non-placement delete operator.

Compiler Error C2574

Error Message
'destructor' : cannot be declared static

Neither destructors nor constructors can be declared **static**.

The following sample generates C2574:

```
// C2574.cpp
// compile with: /c
class A {
   virtual static ~A();   // C2574
   //  try the following line instead
   // virtual ~A();
};
```

Compiler Error C2575

Error Message
'identifier' : only member functions and bases can be virtual

A global function or class is declared **virtual**. This is not allowed.

The following sample generates C2575:

```
// C2575.cpp
// compile with: /c
virtual void func() {}   // C2575

void func2() {}
struct A {
   virtual void func2(){}
};
```

Compiler Error C2577

Error Message
'member' : destructor/finalizer cannot have a return type

A destructor or finalizer cannot return a value of **void** or any other type. Remove the **return** statement from the destructor definition.

Example
The following sample generates C2577.

```
// C2577.cpp
// compile with: /c
class A {
public:
   A() {}
   ~A(){
      return 0;   // C2577
   }
};
```

Compiler Error C2579

Error Message

unable to resolve type 'type' (offset). It is expected in file

C2579 always follows C4691. For more information, see Compiler Warning (level 1) C4691.

Compiler Error C2581

Error Message

'type' : static 'operator =' function is illegal

The assignment (**=**) operator is incorrectly declared as **static**. Assignment operators cannot be **static**. For more information, see User-Defined Operators.

Example

The following sample generates C2581.

```
// C2581.cpp
// compile with: /clr /c
ref struct Y {
   static Y ^ operator = (Y^ me, int i);   // C2581
   Y^ operator =(int i);   // OK
};
```

Compiler Error C2582

Error Message
'function' function is unavailable in 'type'

An attempt was made to assign to an object that does not have an assignment operator.

The following sample generates C2583:

```
// C2582.cpp
// compile with: /clr
using namespace System;

struct N {};
ref struct O {};
ref struct R {
   property O prop;   // C2582
   property O ^ prop2;   // OK
};

int main() {
   String ^ st1 = gcnew String("");
   ^st1 = gcnew String("");   // C2582
   st1 = "xxx";   // OK
}
```

Compiler Error C2583

Error Message
'identifier' : 'const/volatile' 'this' pointer is illegal for constructors/destructors

A constructor or destructor is declared **const** or **volatile**. This is not allowed.

The following sample generates C2583:

```
// C2583.cpp
// compile with: /c
class A {
public:
   int i;
   A() const;   // C2583

   // try the following line instead
   // A();
};
```

Compiler Error C2584

Error Message
'Class' : direct base 'Base2' is inaccessible; already a base of 'Base1'

Class already derives directly from *Base1*. *Base2* also derives from *Base1*. *Class* cannot derive from *Base2* because that would mean inheriting (indirectly) from *Base1* again, which is not legal because *Base1* is already a direct base class.

Example
The following sample generates C2584.

```
// C2584.cpp
// compile with: /c
struct A1 {
   virtual int MyFunction();
};

struct A2 {
    virtual int MyFunction();
};

struct B1: public virtual A1, virtual A2 {
    virtual int MyFunction();
};

struct B2: public virtual A2, virtual A1 {
    virtual int MyFunction();
};

struct C: virtual B1, B2 {
    virtual int MyFunction();
};

struct Z : virtual B2, virtual C {   // C2584
// try the following line insted
// struct Z : virtual C {
    virtual int MyFunction();
};
```

Compiler Error C2585

Error Message
explicit conversion to 'type' is ambiguous

The type conversion can produce more than one result.

To fix by checking the following possible causes

1. Converting from a class or structure type based on multiple inheritance. If the type inherits the same base class more than once, the conversion function or operator must use scope resolution (**::**) to specify which of the inherited classes to use in the conversion.
2. A conversion operator and a constructor have been defined making the same conversion.

Compiler Error C2586

Error Message
incorrect user-defined conversion syntax : illegal indirections

Indirection of a conversion operator is not allowed.

The following sample generates C2586:

```
// c2586.cpp
// compile with: /c
struct C {
   * operator int();   // C2586
   operator char();   // OK
};
```

Compiler Error C2587

Error Message
'identifier' : illegal use of local variable as default parameter

Local variables are not allowed as default parameters.

The following sample generates C2587:

```
// C2587.cpp
// compile with: /c
int i;
void func() {
   int j;
   extern void func2( int k = j );  // C2587 -- local variable
   extern void func3( int k = i );   // OK
}
```

Compiler Error C2588

Error Message
'::~identifier' : illegal global destructor

The destructor is defined for something other than a class, structure, or union. This is not allowed.

This error can be caused by a missing class, structure, or union name on the left side of the scope resolution (**::**) operator.

The following sample generates C2588:

```
// C2588.cpp
~F();   // C2588
```

Compiler Error C2589

Error Message
'identifier' : illegal token on right side of '::'

If a class, structure, or union name appears to the left of the scope-resolution operator (double colons), the token on the right must be a class, structure, or union member. Otherwise, any global identifier can appear on the right.

The scope-resolution operator cannot be overloaded.

The following sample generates C2589:

```
// C2589.cpp
void Test(){}
class A {};
void operator :: ();   // C2589

int main() {
   ::Test();
}
```

Compiler Error C2592

Error Message

no legal conversion of initialization expression to type 'type'

This error may be fixed if you supply a conversion operator to make the required cast.

Compiler Error C2593

Error Message
'operator identifier' is ambiguous

More than one possible operator is defined for an overloaded operator.

This error may be fixed if you use an explicit cast on one or more actual parameters.

The following sample generates C2593:

```
// C2593.cpp
struct A {};
struct B : A {};
struct X {};
struct D : B, X {};
void operator+( X, X );
void operator+( A, B );
D d;

int main() {
   d +  d;         // C2593, D has an A, B, and X
   (X)d + (X)d;    // OK, uses operator+( X, X )
}
```

This error can be caused by serializing a floating-point variable using a **CArchive** object. The compiler identifies the **<<** operator as ambiguous. The only primitive C++ types that **CArchive** can serialize are the fixed-size types **BYTE**, **WORD**, **DWORD**, and **LONG**. All integer types must be cast to one of these types for serialization. Floating-point types must be archived using the **CArchive::Write()** member function.

The following example shows how to archive a floating-point variable (`f`) to archive `ar`:

```
ar.Write(&f, sizeof( float ));
```

Compiler Error C2594

Error Message
'operator' : ambiguous conversions from 'type1' to 'type2'

No conversion from *type1* to *type2* was more direct than any other. One possible solution is to define or specify an explicit conversion.

The following sample generates C2594:

```
// C2594.cpp
// compile with: /c
struct A{};
struct I1 : A {};
struct I2 : A {};
struct D : I1, I2 {};

A *f (D *p) {
   return (A*) (p);    // C2594

   // try the following line instead
   // return reinterpret_cast<A*> (p);
}
```

Compiler Error C2597

Error Message
illegal reference to non-static member 'identifier'

Possible causes:

1. A nonstatic member is specified in a static member function. To access the nonstatic member, you must create an instance of the class and use a member-access operator (**.** or **->**).
2. The specified identifier is not a member of a class, structure, or union.
3. A member access operator refers to a nonmember function.
4. The following sample generates C2597:

```
// C2597.cpp
// compile with: /c
struct s1 {
   static void func();
   int i;
};

void s1::func() {
   i = 1;    // C2597

   // OK
   s1 a;
   a.i = 1;
}
```

Compiler Error C2598

Error Message
linkage specification must be at global scope

The linkage specifier is declared at local scope.

The following sample generates C2598:

```
// C2598.cpp
// compile with: /c
void func() {
   extern "C" int func2();   // C2598
}

extern "C" int func( int i );
```

Compiler Error C2599

Error Message
'enum' : forward declaration of enum type is not allowed

Forward declaration of an enum type is not allowed under /Za.

Beginning in Visual C++ 2005, Managed Extensions for C++ no longer support forward declarations of managed enums. See Breaking Changes in the Visual C++ 2005 Compiler for more information.

The following sample generates C2599:

```
// C2599.cpp
// compile with: /clr /c
enum class Status;   // C2599

enum class Status2 { stop2, hold2, go2};

ref struct MyStruct {
   // Delete the following line to resolve.
   Status m_status;

   Status2 m_status2;   // OK
};

enum class Status { stop, hold, go };
```

Compiler Errors C2600 Through C2699

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, type the error number (C2651, for example) in the index, or click a link below.

Compiler Error C2600

Compiler Error C2601

Compiler Error C2602

Compiler Error C2603

Compiler Error C2605

Compiler Error C2611

Compiler Error C2612

Compiler Error C2613

Compiler Error C2614

Compiler Error C2616

Compiler Error C2617

Compiler Error C2619

Compiler Error C2620

Compiler Error C2621

Compiler Error C2622

Compiler Error C2623

Compiler Error C2624

Compiler Error C2626

Compiler Error C2627

Compiler Error C2628

Compiler Error C2630

Compiler Error C2632

Compiler Error C2633

Compiler Error C2634

Compiler Error C2635

Compiler Error C2636

Compiler Error C2637

Compiler Error C2638

Compiler Error C2640

Compiler Error C2645

Compiler Error C2646

Compiler Error C2647

Compiler Error C2648

Compiler Error C2649

Compiler Error C2650

Compiler Error C2651
Compiler Error C2652
Compiler Error C2653
Compiler Error C2654
Compiler Error C2655
Compiler Error C2656
Compiler Error C2657
Compiler Error C2658
Compiler Error C2659
Compiler Error C2660
Compiler Error C2661
Compiler Error C2662
Compiler Error C2663
Compiler Error C2664
Compiler Error C2665
Compiler Error C2666
Compiler Error C2667
Compiler Error C2668
Compiler Error C2669
Compiler Error C2670
Compiler Error C2671
Compiler Error C2673
Compiler Error C2674
Compiler Error C2675
Compiler Error C2676
Compiler Error C2677
Compiler Error C2678
Compiler Error C2679
Compiler Error C2680
Compiler Error C2681
Compiler Error C2682
Compiler Error C2683
Compiler Error C2687
Compiler Error C2688
Compiler Error C2689
Compiler Error C2690
Compiler Error C2691
Compiler Error C2692
Compiler Error C2693
Compiler Error C2694

Compiler Error C2695

Compiler Error C2696

Compiler Error C2697

Compiler Error C2698

Compiler Error C2600

Error Message
'function' : cannot define a compiler-generated special member function (must be declared in the class first)

Before functions such as constructors or destructors can be defined for a class, they must be declared in the class.

The following sample generates C2600:

```
// C2600.cpp
// compile with: /c
class C {};
C::~C() {}   // C2600

class D {
   D::~D();
};

D::~D() {}
```

Compiler Error C2601

Error Message
'function' : local function definitions are illegal

Code tries to define a function within a function.

Or, there may be an extra brace in your source code before the location of the C2601 error.

The following sample generates C2601:

```
// C2601.cpp
int main() {
   int i = 0;

   void funcname(int j) {   // C2601
      j++;
   }
}
```

Compiler Error C2602

Error Message
'class::Identifier' is not a member of a base class of 'class'

Identifier cannot be accessed because it is not a member inherited from any base class.

The following sample generates C2602:

```
// C2602.cpp
// compile with: /c
struct X {
   int x;
};
struct A {
   int a;
};
struct B : public A {
   X::x;   // C2602 B is not derived from X
   A::a;   // OK
};
```

Compiler Error C2603

Error Message
'function' : Too many block scope static objects with constructor/destructors in function

There is a limit of 31 on the number of static objects you can have in an externally visible inline function.

The following code generates C2603:

```
// C2603.cpp
struct C { C() {} };
extern inline void f1() {
   static C C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,C14,C15,C16;
   static C C17,C18,C19,C20,C21,C22,C23,C24,C25,C26,C27,C28,C29,C30,C31,C32;
   static C C33;   // C2603 Comment this line out to avoid error
}
```

Compiler Error C2605

Error Message

'name' : this method is reserved within a managed class

Certain names are reserved by the compiler for internal functions. For more information, see Destructors and Finalizers in Visual C++.

Example

The following sample generates C2605.

```
// C2605.cpp
// compile with: /clr /c
ref class R {
protected:
   void Finalize() {}   // C2605
};
```

Compiler Error C2611

Error Message
'token' : illegal following '~' (expected identifier)

The token is not an identifier.

The following sample generates C2611:

```
// C2611.cpp
// compile with: /c
class C {
   C::~operator int();   // C2611
   ~C();   // OK
};
```

Compiler Error C2612

Error Message
trailing 'char' illegal in base/member initializer list

A character appears after the last base or member in an initializer list.

The following sample generates C2612:

```
// C2612.cpp
class A {
public:
   int i;
   A( int ia ) : i( ia ) + {};   // C2612
};
```

Compiler Error C2613

Error Message
trailing ',' illegal in base class list

A comma appears after the last base in a base class list.

Compiler Error C2614

Error Message

'class1' : illegal member initialization: 'class2' is not a base or member

Only member or base classes can appear in the initialization list for a class or structure.

Example

The following sample generates C2614.

```
// C2614.cpp
// compile with: /c
struct A {
   int i;
   A( int ia ) : B( i ) {};   // C2614 B is not a member of A
};

struct A2 {
   int B;
   int i;
   A2( int ia ) : B( i ) {};   // OK
};
```

Compiler Error C2616

Error Message

'conversion' : cannot implicitly convert a non-lvalue 'type1' to a 'type2' that is not const

A reference cannot be initialized from a non-lvalue.

This is an error under ANSI compatibility (/Za) and a warning under Microsoft extensions (**/Ze**).

Compiler Error C2617

Error Message
'function' : inconsistent return statement

The specified function does not have a declared return type, and a previous return statement did not supply a value.

The following sample generates C2617:

```
// C2617.cpp
int i;
func() {   // no return type prototype
   if( i ) return;   // no return value
   else return( 1 );   // C2617 detected on this line
}
```

Possible resolution:

```
// C2617b.cpp
// compile with: /c
int i;
int MyF() {
   if (i)
      return 0;
   else
      return (1);
}
```

Compiler Error C2619

Error Message
union 'union' : cannot have static member variable 'identifier'

A union member is declared **static**.

The following sample generates C2619

```
// C2619.cpp
int main() {
   union { static int j; };  // C2619
   union { int j; };  // OK
}
```

Compiler Error C2620

Error Message
member 'identifier' of union 'union' has user-defined constructor or non-trivial default constructor

A union member cannot have a default constructor.

The following sample generates C2620:

```
// C2620.cpp
class A {
   A();   // A has a default constructor
};
union U {
   A a;   // C2620
};
```

Compiler Error C2621

Error Message
member 'identifier' of union 'union' has copy constructor

A union member cannot have a copy constructor.

The following sample generates C2621:

```
// C2621.cpp
class A {
   A( const A& );   // A has a copy constructor
};
union U {
   A a;   // C2621
};
```

C2621 can also be generates for code that compiled in Visual C++ 6.0, but that no longer compiles. The Visual C++ 6.0 Standard C++ Library did not use a union of its template parameters. Class types that have a copy constructor cannot be a template parameter for basic_string.

The following sample generates C2621:

```
// C2621b.cpp
// compile with: /EHsc
#include <vector>
#include <string>
using namespace std;
typedef vector<int> mytype;

int main() {
   basic_string < mytype > b;   // C2621 class type (vector) with
                               //  copy constructor as template param
   basic_string < char > c;   // OK
}
```

Compiler Error C2622

Error Message
member 'identifier' of union 'union' has assignment operator

A union member cannot have an assignment operator.

The following sample generates C2622:

```
// C2622.cpp
class A {
   A& operator= ( const A& );  // A's assignment operator
};
union U {
   A a;   // C2622
};
```

Compiler Error C2623

Error Message
member 'identifier' of union 'union' has destructor

A union member cannot have a destructor.

The following sample generates C2623:

```
// C2623.cpp
class A {
   ~A();   // A has a destructor
};
union U {
   A a;    // C2623
};
```

Compiler Error C2624

Error Message
local classes cannot be used to declare 'extern' variables

A local class or structure cannot be used to declare **extern** variables.

The following sample generates C2624:

```
// C2624.cpp
int main() {
   struct C {};
   extern C ac;   // C2624
}
```

Compiler Error C2626

Error Message
anonymous union defines protected/private member 'identifier'

A member of an anonymous union must have public access.

The following sample generates C2626:

```
// C2626.cpp
int main() {
   union {
   protected:
      int j;     // C2626, j is protected
   private:
      int k;     // C2626, k is private
   };
}
```

Possible resolution:

```
// C2626b.cpp
int main() {
   union {
   public:
      int i;   // OK, i is public
   };
}
```

Compiler Error C2627

Error Message
'function' : member function not allowed in anonymous union

An anonymous union cannot have member functions.

The following sample generates C2627:

```
// C2627.cpp
int main() {
   union { void f(){} };   // C2627
   union X { void f(){} };
}
```

Compiler Error C2628

Error Message
'type1' followed by 'type2' is illegal (did you forget a ';'?)

A semicolon may be missing.

The following sample generates C2628:

```
// C2628.cpp
class CMyClass {}
int main(){}   // C2628 error
```

Possible resolution:

```
// C2628b.cpp
class CMyClass {};
int main(){}
```

Compiler Error C2630

Error Message
'symbol' found in what should be a comma-separated list

The symbol appears in a context that requires a comma.

The following sample generates C2630:

```
// C2630.cpp
// compile with: /c
struct D {
   D(int);
};

struct E {
   E(int);
};

class C : public D, public E {
   C();
};

C::C() : D(0) ; E(0) { }   // C2630
C::C() : D(0), E(0) {}   // OK
```

Compiler Error C2632

Error Message
'type1' followed by 'type2' is illegal

This error can be caused if there is missing code between two type specifiers.

The following sample generates C2632:

```
// C2632.cpp
int float i;   // C2632
```

This error can also be generated as a result of compiler conformance work that was done for Visual Studio .NET 2003. **bool** is now a proper type. In previous versions, **bool** was a typedef, and you could create identifiers with that name.

For more information, see Compile-Time Breaking Changes.

The following sample generates C2632:

```
// C2632_2.cpp
// compile with: /LD
void f(int bool);   // C2632
```

To resolve this error so that the code is valid in both the Visual Studio .NET 2003 and Visual Studio .NET versions of Visual C++, rename the identifier.

Compiler Error C2633

Error Message
'identifier' : 'inline' is the only legal storage class for constructors

A constructor is declared as a storage class other than inline.

The following sample generates C2633:

```
// C2633.cpp
// compile with: /c
class C {
   extern C();   // C2633, not inline
   inline C();   // OK
};
```

Compiler Error C2634

Error Message
'&class::member' : pointer to reference member is illegal

A pointer to a reference member is declared.

The following sample generates C2634:

```
// C2634.cpp
int mem;
struct S {
   S() : rf(mem) { }
   int &rf;
};
int (S::*pdm) = &S::rf;   // C2634
```

Compiler Error C2635

Error Message
cannot convert an 'identifier1*' to an 'identifier2*'; conversion from a virtual base class is implied

The conversion requires a cast from a **virtual** base class to a derived class, which is not allowed.

The following sample generates C2635:

```
// C2635.cpp
class B {};
class D : virtual public B {};
class E : public B {};

int main() {
   B b;
   D d;
   E e;

   D * pD = &d;
   E * pE = &e;
   pD = (D*)&b;   // C2635
   pE = (E*)&b;   // OK
}
```

Compiler Error C2636

Error Message
'identifier' : pointer to reference member is illegal

A pointer to a reference member was declared.

The following sample generates C2636:

```
// C2636.cpp
struct S {};
int main() {
   int &S::*prs;   // C2636
   int S::*prs1;   // OK
   int *S::*prs2;   // OK
}
```

Compiler Error C2637

Error Message
'identifier' : cannot modify pointers to data members

A pointer to a data member cannot have a calling convention. To resolve, either remove the calling convention or declare a pointer to member function.

The following sample generates C2637:

```
// C2637.cpp
// compile with: /c
struct S {};
int __stdcall S::*pms1;   // C2637

// OK
int S::*pms2;
int (__stdcall S::*pms3)(...);
```

Compiler Error C2638

Error Message
'identifier' : __based modifier illegal on pointer to member

The **__based** modifier cannot be used for pointers to members.

The following sample generates C2638:

```
// C2638.cpp
void *a;

class C {
public:
   int i;
   int j;
   int func();
};
int __based (a) C::* cpi = &C::i;  // C2638
int (__based (a) C::* cpf)() = &C::func; // c2638
```

Compiler Error C2640

Error Message
'identifier' : __based modifier illegal on reference

The **__based** modifier can be used on pointers only.

The following sample generates C2640:

```
// C2640.cpp
void f(int i) {
    void *vp;
    int _based(vp) &vr = I;  // C2640
}
```

Compiler Error C2645

Error Message
no qualified name for pointer to member (found '::*')

The declaration of a pointer to a member does not specify a class.

The following sample generates C2645:

```
// C2645.cpp
class A {};
int main() {
   int B::* bp;   // C2645 B not defined
   int A::* ap;   // OK
}
```

Compiler Error C2646

Error Message
global anonymous unions must be declared static

An anonymous union has global scope but is not declared **static**.

The following sample generates C2646:

```
// C2646.cpp
// compile with: /c
union { int i; };   // C2646 not static

// OK
static union { int j; };
union U { int i; };
```

Compiler Error C2647

Error Message
'operator': cannot dereference a 'type1' on a 'type2'

The left operand of a pointer-to-member operator (**->*** or **.***) cannot be implicitly converted to a type related to the right operator.

The following sample generates C2647:

```
// C2647.cpp
class C {};
class D {};

int main() {
   D d, *pd;
   C c, *pc = 0;
   int C::*pmc = 0;
   pd->*pmc = 0;   // C2647
   d.*pmc = 0;   // C2647

   // OK
   pc->*pmc = 0;
   c.*pmc = 0;
}
```

Compiler Error C2648

Error Message
'identifier' : use of member as default parameter requires static member

A non-static member is used as a default parameter.

The following sample generates C2648:

```
// C2648.cpp
// compile with: /c
class C {
public:
   int i;
   static int j;
   void func1( int i = i );  // C2648  i is not static
   void func2( int i = j );  // OK
};
```

Compiler Error C2649

Error Message
'identifier' : is not a 'class-key'

A class, structure, or union declaration uses an incorrect tag.

Compiler Error C2650

Error Message

'operator' : cannot be a virtual function

A **new** or **delete** operator is declared **virtual**. These operators are **static** member functions and cannot be **virtual**.

Example

The following sample generates C2650:

```
// C2650.cpp
// compile with: /c
class A {
   virtual void* operator new( unsigned int );   // C2650
   // try the following line instead
   // void* operator new( unsigned int );
};
```

Compiler Error C2651

Error Message
'data type' : left of 'operator' must be a class, struct or union

To use a template parameter as if it is a class, specialize the class template with a class instead of an integral type.

The following sample generates C2651:

```
// C2651.cpp
template<typename T, typename T::PMF pmf>
struct X1 {};

struct Y1 {
   typedef void (Y1::*PMF)();

   void mf() {
   }
};

void f1() {
   X1<int, 0> x11;   // C2651
   X1<Y1, &Y1::mf> x12;   // OK
}
```

Compiler Error C2652

Error Message
'identifier' : illegal copy constructor: first parameter must not be an 'identifier'

The first parameter in the copy constructor has the same type as the class, structure, or union for which it is defined. The first parameter can be a reference to the type but not the type itself.

The following sample generates C2651:

```
// C2652.cpp
// compile with: /c
class A {
   A( A );   // C2652 takes an A
};
class B {
   B( B& );   // OK, reference to B
};
```

Compiler Error C2653

Error Message
'identifier' : is not a class or namespace name

Syntax requires a class, structure, union, or namespace name.

The following sample generates C2653:

```
// C2653.cpp
// compile with: /c
class yy {
   void func1(int i);
};

void xx::func1(int m) {}   // C2653
void yy::func1(int m) {}   // OK
```

C2653 is also possible if you try to define a compound namespace; compound namespaces are not allowed in C++:

```
// C2653b.cpp
namespace a::b {int i;}   // C2653

namespace a {
   namespace b {
      int i;
   }
}

int main() {
   a::b::i = 2;
}
```

Compiler Error C2654

Error Message
'identifier' : attempt to access member outside a member function

A member is accessed in a declaration. Member data can be accessed only in member functions.

This error can be caused when trying to initialize variables in a declaration. Use a constructor for this purpose.

Compiler Error C2655

Error Message
'identifier' : definition or redeclaration illegal in current scope

An identifier can be redeclared only at global scope.

The following sample generates C2655:

```
// C2655.cpp
class A {};
class B {
public:
   static int i;
};

int B::i;  // OK

int main() {
   A B::i;  // C2655
}
```

Compiler Error C2656

Error Message
'function' : function not allowed as a bit field

A function is declared as a member of a bit field.

This error can be caused by a syntax error in a constructor initializer list.

Compiler Error C2657

Error Message
'class::*' found at the start of a statement (did you forget to specify a type?)

The line began with a pointer-to-member identifier.

This error can be caused by a missing type specifier in the declaration of a pointer to a member.

The following sample generates C2657:

```
// C2657.cpp
class C {};
int main() {
   C::* pmc1;        // C2657
   int C::* pmc2;   // OK
}
```

Compiler Error C2658

Error Message
'member': redefinition in anonymous struct/union

Two anonymous structures or unions contained member declarations with the same identifier but with different types. Under /Za, you will also get this error for members with the same identifier and type.

The following sample generates C2658:

```
// C2658.cpp
// compile with: /c
struct X {
   union { // can be struct too
      int i;
   };
   union {
      int i;   // Under /Za, C2658
      // int i not needed here because it is defined in the first union
   };
};

struct Z {
   union {
      char *i;
   };

   union {
      void *i;   // C2658 redefinition of 'i'
      // try the following line instead
      // void *ii;
   };
};
```

Compiler Error C2659

Error Message
'operator' : function as left operand

An overloaded function was on the left side of the specified operator.

The following sample generates C2659:

```
// C2659.cpp
int func( int );
int func( double );
int main() {
   func = 10;   // C2659
}
```

Compiler Error C2660

Error Message
'function' : function does not take number parameters

The function is called with an incorrect number of parameters.

C2660 can occur if you accidentally call a Windows API function rather than an MFC member function of the same name. To solve this problem:

- Adjust the function call to conform to the format of the member function call.
- Use the scope resolution operator (**::**) to tell the compiler to seek the function name in the global name space.

Example
The following sample generates C2660.

```
// C2660.cpp
void func( int, int ) {}

int main() {
   func( 1 );   // C2660 func( int ) not declared
   func( 1, 0 );   // OK
}
```

C2660 can also occur if you attempt to directly call the Dispose method of a managed type. For more information, see Destructors and Finalizers in Visual C++. The following sample generates C2660.

```
// C2660_a.cpp
// compile with: /clr
using namespace System;
using namespace System::Threading;

void CheckStatus( Object^ stateInfo ) {}

int main() {
   ManualResetEvent^ event = gcnew ManualResetEvent( false );
   TimerCallback^ timerDelegate = gcnew TimerCallback( &CheckStatus );
   Timer^ stateTimer = gcnew Timer( timerDelegate, event, 1000, 250 );

   stateTimer->Dispose();   // C2660
   stateTimer->~Timer();   // OK
}
```

C2660 will occur if a derived class hides a function.

```
// C2660b.cpp
// C2660 expected
#include <stdio.h>

class f {
public:
   void bar() {
      printf_s("in f::bar\n");
    }
};

class f2 : public f {
public:
   void bar(int i){printf("in f2::bar\n");}
   // Uncomment the following line to resolve.
   // using f::bar;   // - using declaration added
   // or
```

```
   // void bar(){__super::bar();}
};

int main() {
   f2 fObject;
   fObject.bar();
}
```

C2660 can occur if you invoke an indexed property incorrectly.

```
// C2660c.cpp
// compile with: /clr
ref class X {
   double d;
public:
   X() : d(1.9) {}
   property double MyProp[] {
      double get(int i) {
         return d;
      }
   }   // end MyProp definition
};

int main() {
   X ^ MyX = gcnew X();
   System::Console::WriteLine(MyX->MyProp(1));   // C2660
   System::Console::WriteLine(MyX->MyProp[1]);   // OK
}
```

C2660 can occur if you invoke an indexed property incorrectly.

```
// C2660d.cpp
// compile with: /clr
ref class A{
public:
   property int default[int,int] {
      int get(int a, int b) {
         return a + b;
      }
   }
};

int main() {
   A^ a = gcnew A;
   int x = a[3][5];   // C2660
   int x2 = a[3,5];   // OK
}
```

C2660 can occur if you define a new operator in a template class, but where the new operator creates an object whose type is other than the enclosing type.

```
// C2660e.cpp
// compile with: /c
#include <malloc.h>

template <class T> class CA {
private:
    static T** line;
   void* operator new (size_t, int i) {
      return 0;
   }
   void operator delete(void* pMem, int i) {
      free(pMem);
   }
```

```
public:
   CA () { new (1) T(); }   // C2660
   // try the following line instead
   // CA () { new (1) CA<int>(); }
};

typedef CA <int> int_CA;

void AAA() {
   int_CA  list;
}
```

Compiler Error C2661

Error Message
'function' : no overloaded function takes number parameters

Possible casuses:

1. Incorrect actual parameters in function call.
2. Missing function declaration.

The following sample generates C2661:

```
// C2661.cpp
void func( int ){}
void func( int, int ){}
int main() {
   func( );   // C2661 func( void ) was not declared
   func( 1 );   // OK func( int ) was declared
}
```

Compiler Error C2662

Error Message
'function' : cannot convert 'this' pointer from 'type1' to 'type2'

The compiler could not convert the **this** pointer from *type1* to *type2*.

This error can be caused by invoking a non-**const** member function on a **const** object. Possible resolutions:

- Remove the **const** from the object declaration.
- Add **const** to the member function.

The following sample generates C2662:

```
// C2662.cpp
class C {
public:
   void func1();
   void func2() const{}
} const c;

int main() {
   c.func1();   // C2662
   c.func2();   // OK
}
```

When compiling with **/clr**, you cannot call a function on a **const** or **volatile** qualified managed type. You cannot declare a const member function of a managed class, so you cannot call methods on const managed objects.

```
// C2662_b.cpp
// compile with: /c /clr
ref struct M {
   property M^ Type {
      M^ get() { return this; }
   }

   void operator=(const M %m) {
      M ^ prop = m.Type;   // C2662
   }
};

ref struct N {
   property N^ Type {
      N^ get() { return this; }
   }

   void operator=(N % n) {
      N ^ prop = n.Type;   // OK
   }
};
```

The following sample generates C2662:

```
// C2662_c.cpp
// compile with: /c
// C2662 expected
typedef int ISXVD;
typedef unsigned char BYTE;

class LXBASE {
protected:
    BYTE *m_rgb;
};
```

```
class LXISXVD:LXBASE {
public:
   // Delete the following line to resolve.
   ISXVD *PMin() { return (ISXVD *)m_rgb; }

   ISXVD *PMin2() const { return (ISXVD *)m_rgb; };   // OK
};

void F(const LXISXVD *plxisxvd, int iDim) {
   ISXVD isxvd;
   // Delete the following line to resolve.
   isxvd = plxisxvd->PMin()[iDim];

   isxvd = plxisxvd->PMin2()[iDim];
}
```

Compiler Error C2663

Error Message
'function' : number overloads have no legal conversions for 'this' pointer

The compiler could not convert **this** to any of the overloaded versions of the member function.

This error can be caused by invoking a non-**const** member function on a **const** object. Possible resolutions:

1. Remove the **const** from the object declaration.
2. Add **const** to one of the member function overloads.

The following sample generates C2663:

```
// C2663.cpp
struct C {
   void f() volatile {}
   void f() {}
};

struct D {
   void f() volatile;
   void f() const {}
};

const C *pcc;
const D *pcd;

int main() {
   pcc->f();    // C2663
   pcd->f();    // OK
}
```

Compiler Error C2664

Error Message
'function' : cannot convert parameter number from 'type1' to 'type2'

A parameter cannot be converted to the required type. This might happen if you create an instance of a class and attempt an implicit conversion on a constructor marked with the explicit keyword.

If a temporary object is passed to a function that takes a reference to an object as a parameter, that reference must be a const reference.

If the function is called with a parameter that is not of the type that the function expects, a temporary object is created using the appropriate constructor. This temporary object is then passed to the function. In this case, the temporary object is used to initialize the reference. In earlier versions of the language, all references could be initialized by temporary objects.

To resolve C2664,

- Recheck the prototype for the given function and correct the argument noted in the error message.
- Supply an explicit conversion if necessary.

C2664 can also be generated if a class hides a member in one of its base classes.

For more information, see How to: Convert System::String to wchar_t* or char*.

Example
The following sample generates C2664.

```
// C2664.cpp
// C2664
struct A {
   void f(int i) {};
};

struct B : public A {
   // Uncomment the following line to resolve.
   // using A::f;
   void f(A a) {};
};

int main() {
   B b;
   int i = 1;
   b.f(i);   // B::F hides A::f uncomment using declaration in B
}
```

The following sample generates C2664.

```
// C2664b.cpp
// C2664 expected
struct A {
   // Uncomment the following line to resolve.
   // A(int i){}
};

void func( int, A ) {}

int main() {
   func( 1, 1 );   // no conversion from int to A
}
```

The code below demonstrates this error by calling `Test` with a string literal. Because the parameter is an szString reference, an object must be created by the appropriate constructor. The result is a temporary object that cannot be used to initialize the reference.

```
// C2664c.cpp
// compile with: /EHsc
// C2664 expected
#include <iostream>
#include <string.h>
using namespace std;

class szString {
   int slen;
   char *str;

public:
   szString(const char *);
   int len() const {
      return slen;
   }
};

// simple reference cannot bind to temp var
void Test(szString &a) {}

// Uncomment the following line to resolve.
// void Test(const szString &a) {}

szString::szString(const char * newstr) : slen(0), str(NULL) {
   slen=strlen(newstr);
   str = new char[slen + 1];
   if (str)
      strcpy_s(str, (slen + 1), newstr);
}

int main() {
   Test("hello");
}
```

In Visual C++ 2005, the compiler now enforces the C++ standard requirements for applying **const**. The following sample generates C2664.

```
// C2664d.cpp
// C2664 expected
#include <windows.h>

void func1(LPCSTR &s)
{

}

void func2(LPSTR &s)
{
   func1(s);
}

int main()
{
   return 0;
}
```

The following demonstrates a more complex situation where C2664 is generated.

```
// C2664e.cpp
// compile with: /EHsc
// C2664 expected
#define _INTL
#include <locale>
#include <iostream>
```

```
using namespace std;
#define LEN 90

int main( ) {
   char* pszExt = "This is the string to be converted!";
   wchar_t pwszInt [LEN+1];
   memset(&pwszInt[0], 0, (sizeof(wchar_t))*(LEN+1));

   // Delete the following line to resolve.
   char* pszNext;

   // Uncomment the following line to resolve.
   // const char* pszNext;

   wchar_t* pwszNext;
   mbstate_t state;
   locale loc("C");
   int res = use_facet<codecvt<wchar_t, char, mbstate_t> >
      ( loc ).in( state,
      pszExt, &pszExt[strlen(pszExt)], pszNext,
      pwszInt, &pwszInt[strlen(pszExt)], pwszNext );
   // see comment above
      pwszInt[strlen(pszExt)] = 0;
   wcout << ( (res!=codecvt_base::error) ?
                       L"It worked! " : L"It didn't work! " )
   << L"The converted string is:\n ["
   << &pwszInt[0]
   << L"]" << endl;

   exit(-1);
}
```

An enum variable is not converted to its underlying type, such that, a function call will be satisfied. For more information, see enum class. The following sample generates C2664.

```
// C2664f.cpp
// compile with: /clr
using namespace System;
public enum class A : Char {
   None = 0,
   NonSilent = 1,
};

void Test(Char c) {}

int main() {
   A aa = A::None;
   Test(aa);   // C2664
   Test(Char(aa));   // OK
}
```

A bug in the midl compiler results in a wchar_t type being emitted as an unsigned short in the type library. To resolve this error, either cast the type in your C++ source code or define the type as a string in the idl file.

```
// C2664g.idl
import "prsht.idl";

[ object, uuid(8402B8F1-BF7F-4B49-92D4-C2B9DF4543E9) ]

interface IMyObj1 : IUnknown {
   HRESULT  teststr([in, string] wchar_t *wstr);
   HRESULT  testarr([in, size_is(len)] wchar_t wstr[], [in] int len);
   HRESULT  testbstr([in] BSTR bstr);
};
```

```
[  uuid(44463307-CBFC-47A6-8B4F-13CD0A83B436) ]
library myproj1 {
   [  version(1.0), uuid(D8622C12-5448-42B8-8F0E-E3AD6B8470C1) ]
   coclass CMyObj1 { interface IMyObj1; };
}
```

The following sample generates C2664.

```
// C2664h.cpp
#import "C2664g.tlb"
using namespace myproj1;

int main() {
   IMyObj1Ptr ptr;

   wchar_t * mybuff = 0;
   BSTR bstr = 0;
   int len;
   ptr->teststr(mybuff);
   ptr->testbstr(bstr);
   ptr->testarr(mybuff, len);   // C2664
   ptr->testarr((unsigned short *)mybuff, len);   // OK
}
```

C2664 is also caused if the compiler could not deduce template arguments. The following sample generates C2664.

```
// C2664i.cpp
#include <stdio.h>
template <class T, int iType=0>
class CTypedImg {
public:
   CTypedImg() {}
   void run() {}

   operator CTypedImg<T>& () {
      return *((CTypedImg<T>*)this);
    }
};

template <class t1>
void test(CTypedImg<t1>& myarg) {
   myarg.run();
}

int main() {
   CTypedImg<float,2> img;

   test((CTypedImg<float>&)img);   // ok
   test<float>(img);   // ok
   test(img);   // C2664
}
```

Compiler Error C2665

Error Message
'function' : none of the number1 overloads can convert parameter number2 from type 'type'

A parameter of the overloaded function cannot be converted to the required type. Possible resolutions:

- Supply a conversion operator.
- Use explicit conversion.

Example
The following sample generates C2665.

```
// C2665.cpp
void func(short, char*){}
void func(char*, char*){}

int main() {
   func(0, 1);   // C2665
   func((short)0, (char*)1);   // OK
}
```

Compiler Error C2666

Error Message
'identifier' : number overloads have similar conversions

An overloaded function or operator is ambiguous. Formal parameter lists may be too similar for the compiler to resolve the ambiguity. To resolve this error, explicitly cast one or more of the actual parameters.

The following sample generates C2666:

```
// C2666.cpp
struct complex {
   complex(double);
};

void h(int,complex);
void h(double, double);

int main() {
   h(3,4);   // C2666
}
```

This error can also be generated as a result of compiler conformance work that was done for Visual Studio .NET 2003:

- binary operators and user-defined conversions to pointer types
- qualification conversion is not the same as identity conversion

For the binary operators <, >, <=, and >=, a passed parameter is now implicitly converted to the type of the operand if the parameter's type defines a user-defined conversion operator to convert to the type of the operand. There is now potential for ambiguity.

See Summary of Compile-Time Breaking Changes for more information.

For code that is valid in both the Visual Studio .NET 2003 and Visual Studio .NET versions of Visual C++, call the class operator explicitly using function syntax.

Example

```
// C2666b.cpp
#include <string.h>
#include <stdio.h>

struct T
{
    T( const T& copy )
    {
        m_str = copy.m_str;
    }

    T( const char* str )
    {
        int iSize = (strlen( str )+ 1);
        m_str = new char[ iSize ];
        if (m_str)
            strcpy_s( m_str, iSize, str );
    }

    bool operator<( const T& RHS )
    {
        return m_str < RHS.m_str;
    }

    operator char*() const
    {
        return m_str;
```

```
    }

    char* m_str;
};

int main()
{
    T str1( "ABCD" );
    const char* str2 = "DEFG";

    // Error – Ambiguous call to operator<()
    // Trying to convert str1 to char* and then call
    // operator<( const char*, const char* )?
    //  OR
    // trying to convert str2 to T and then call
    // T::operator<( const T& )?

    if( str1 < str2 )   // C2666

    if ( str1.operator < ( str2 ) )   // Treat str2 as type T
        printf_s("str1.operator < ( str2 )\n");

    if ( str1.operator char*() < str2 )   // Treat str1 as type char*
        printf_s("str1.operator char*() < str2\n");
}
```

The following sample generates C2666

```
// C2666c.cpp
// compile with: /c

enum E
{
    E_A,   E_B
};

class A
{
    int h(const E e) const {return 0; }
    int h(const int i) { return 1; }
    // Uncomment the following line to resolve.
    // int h(const E e) { return 0; }

    void Test()
    {
        h(E_A);   // C2666
        h((const int) E_A);
        h((int) E_A);
    }
};
```

Compiler Error C2667

Error Message
'function' : none of number overloads have a best conversion

An overloaded function call is ambiguous and cannot be resolved.

The conversion required to match the actual parameters in the function call to one of the overloaded functions must be strictly better than the conversions required by all the other overloaded functions.

See Knowledge Base article Q240869 for more information on partial ordering of function templates.

Compiler Error C2668

Error Message
'function' : ambiguous call to overloaded function

The specified overloaded function call could not be resolved. You may want to explicitly cast one or more of the actual parameters.

You can also get this error through template use. If, in the same class, you have a regular member function and a templated member function with the same signature, the templated one must come first. This is a limitation of the current implementation of Visual C++.

See Knowledge Base article Q240869 for more information on partial ordering of function templates.

If you are building an ATL project containing a COM object supporting **ISupportErrorInfo**, see Knowledge Base article Q243298.

Example
The following sample generates C2668:

```
// C2668.cpp
struct A {};
struct B : A {};
struct X {};
struct D : B, X {};

void func( X, X ){}
void func( A, B ){}
D d;
int main() {
   func( d, d );   // C2668 D has an A, B, and X
   func( (X)d, (X)d );   // OK, uses func( X, X )
}
```

Another way to resolve this error is with a using declaration:

```
// C2668b.cpp
// compile with: /EHsc /c
// C2668 expected
#include <iostream>
class TypeA {
public:
   TypeA(int value) {}
};

class TypeB {
   TypeB(int intValue);
   TypeB(double dbValue);
};

class TestCase {
public:
   void AssertEqual(long expected, long actual, std::string
                    conditionExpression = "");
};

class AppTestCase : public TestCase {
public:
   // Uncomment the following line to resolve.
   // using TestCase::AssertEqual;
   void AssertEqual(const TypeA expected, const TypeA actual,
                    std::string conditionExpression = "");
   void AssertEqual(const TypeB expected, const TypeB actual,
                    std::string conditionExpression = "");
```

```
};

class MyTestCase : public AppTestCase {
   void TestSomething() {
      int actual = 0;
      AssertEqual(0, actual, "Value");
   }
};
```

This error can also be generated as a result of compiler conformance work that was done for Visual Studio .NET 2003: ambiguous conversion on cast of constant 0.

Conversion on a cast using constant 0 is ambiguous since int requires a conversion both to long and to void*. To resolve this error, cast 0 to the exact type of the function parameter it is being used for so that no conversions need to take place (this code will be valid in the Visual Studio .NET 2003 and Visual Studio .NET versions of Visual C++).

See Summary of Compile-Time Breaking Changes for more information.

```
// C2668c.cpp
#include "stdio.h"
void f(long) {
   printf_s("in f(long)\n");
}
void f(void*) {
   printf_s("in f(void*)\n");
}
int main() {
   f((int)0);   // C2668

   // OK
   f((long)0);
   f((void*)0);
}
```

This error can occur because the CRT now has float and double forms of all math functions.

```
// C2668d.cpp
#include <math.h>
int main() {
   int i = 0;
   float f;
   f = cos(i);   // C2668
   f = cos((float)i);   // OK
}
```

This error can occur because the pow(int, int) was removed from math.h in the CRT.

```
// C2668e.cpp
#include <math.h>
int main() {
   pow(9,9);   // C2668
   pow((double)9,9);   // OK
}
```

Compiler Error C2669

Error Message
member function not allowed in anonymous union

Anonymous unions cannot have member functions.

The following sample generates C2669:

```
// C2669.cpp
struct X {
   union {
      int i;
      void f() {   // C2669, remove function
         i = 0;
      }
   };
};
```

See Also
Reference
Anonymous Unions

Compiler Error C2670

Error Message

'identifier' : the function template cannot convert parameter number from type 'type'

A parameter could not be converted to the required type.

This error may be fixed if you create an explicit conversion.

Compiler Error C2671

Error Message
'function' : static member functions do not have 'this' pointers

A **static** member function tried to access **this**.

The following sample generates C2671:

```
// C2671.cpp
struct S {
   static S* const func() { return this; }  // C2671
};
```

Compiler Error C2673

Error Message
'function' : global functions do not have 'this' pointers

A global function tried to access **this**.

The following sample generates C2673:

```
// C2673.cpp
int main() {
   this = 0;   // C2673
}
```

Compiler Error C2674

Error Message

a generic declaration is not allowed in this context

A generic was declared incorrectly. For more information, see Generics (Visual C++).

Example

The following sample generates C2674.

```
// C2674.cpp
// compile with: /clr /c
void F(generic <class T> ref class R1);   // C2674
generic <class T> ref class R2 {};   // OK
```

Compiler Error C2675

Error Message
unary 'operator' : 'type' does not define this operator or a conversion to a type acceptable to the predefined operator

C2675 can also occur when using a unary operator, and the type does not define the operator or a conversion to a type acceptable to the predefined operator. To use the operator, you must overload it for the specified type or define a conversion to a type for which the operator is defined.

Example
The following sample generates C2675.

```
// C2675.cpp
struct C {
   C(){}
} c;

struct D {
   D(){}
   void operator-(){}
} d;

int main() {
   -c;   // C2675
   -d;   // OK
}
```

Compiler Error C2676

Error Message
binary 'operator' : 'type' does not define this operator or a conversion to a type acceptable to the predefined operator

To use the operator, you must overload it for the specified type or define a conversion to a type for which the operator is defined.

Example
The following sample generates C2676.

```
// C2676.cpp
// C2676 expected
struct C {
   C();
} c;

struct D {
   D();
   D operator >>( C& ){return * new D;}
   D operator <<( C& ){return * new D;}
} d;

struct E {
   // operator int();
};

int main() {
   d >> c;
   d << c;
   E e1, e2;
   e1 == e2;   // uncomment operator int in class E, then
               // it is OK even though neither E::operator==(E) nor
               // operator==(E, E) defined. Uses the conversion to int
               // and then the builtin-operator==(int, int)
}
```

C2676 can also occur if you attempt to do pointer arithmetic on the **this** pointer of a reference type.

The **this** pointer is of type handle in a reference type. For more information, see
Semantics of the this Pointer in Value and Reference Types.

The following sample generates C2676.

```
// C2676_a.cpp
// compile with: /clr
using namespace System;

ref struct A {
   property Double default[Double] {
      Double get(Double data) {
         return data*data;
      }
   }

   A() {
      Console::WriteLine("{0}", this + 3.3);   // C2676
      Console::WriteLine("{0}", this[3.3]);   // OK
   }
};

int main() {
   A ^ mya = gcnew A();
}
```

Compiler Error C2677

Error Message
binary 'operator' : no global operator found which takes type 'type' (or there is no acceptable conversion)

To use the operator, you must overload it for the specified type or define a conversion to a type for which the operator is defined.

The following sample generates C2677:

```
// C2677.cpp
class C {
public:
   C(){}
} c;

class D {
public:
   D(){}
   operator int(){return 0;}
} d;

int main() {
   int i = 1 >> c;   // C2677
   int j = 1 >> d;   // OK operator int() defined
}
```

Compiler Error C2678

Error Message
binary 'operator' : no operator defined which takes a left-hand operand of type 'type' (or there is no acceptable conversion)

To use the operator, you must overload it for the specified type or define a conversion to a type for which the operator is defined.

C2678 can occur if you do not pin a native member before calling a member function on it.

Example
The following sample generates C2678.

```
// C2678.cpp
// compile with: /clr /c
struct S { int _a; };

ref class C {
public:
   void M( S param ) {
      test = param;   // C2678

      // OK
      pin_ptr<S> ptest = &test;
      *ptest = param;
   }
   S test;
};
```

The following sample generates C2678.

```
// C2678_2.cpp
// compile with: /clr:oldSyntax /c
struct S { int _a; };

__gc class C {
public:
   void M(S param) {
      test = param;   // C2678

      // OK
      S __pin* ptest = &test;
      *ptest = param;
   }

   S test;
};
```

Compiler Error C2679

Error Message
binary 'operator' : no operator found which takes a right-hand operand of type 'type' (or there is no acceptable conversion)

To use the operator, you must overload it for the specified type or define a conversion to a type for which the operator is defined.

The following sample generates C2679:

```
// C2679.cpp
class C {
public:
   C();   // no constructor with an int argument
} c;

class D {
public:
   D(int) {}
   D(){}
} d;

int main() {
   c = 10;   // C2679
   d = 10;   // OK
}
```

Compiler Error C2680

Error Message
'type' : invalid target type for name

A casting operator tried to convert to a type that is not a pointer or reference. The dynamic_cast operator can be used only for pointers or references.

The following sample generates C2680:

```
// C2680.cpp
// compile with: /c
class A { virtual void f(); };
class B : public A {};

void g(B b) {
   A a;
   a = dynamic_cast<A>(b);   // C2680  target not a reference type
   a = dynamic_cast<A&>(b);   // OK
}
```

C2680 can also occur when the target is not defined:

```
// C2680b.cpp
// compile with: /clr /c
// C2680 expected
using namespace System::Collections;

// Delete the following line to resolve.
ref class A;   // not defined

// Uncomment the following line to resolve.
// ref class A{};

public ref class B : ArrayList {
   property A^ C[int] {
      A^ get(int index) {
         return dynamic_cast<A^>(this->default::get(index));
      }
      void set(int index, A^ value) {
         this->default::set(index, value);
      }
   }
};
```

Compiler Error C2681

Error Message
'type' : invalid expression type for name

A casting operator tried to convert from an invalid type. For example, if you use the dynamic_cast operator to convert an expression to a pointer type, the source expression must be a pointer.

The following sample generates C2681:

```
// C2681.cpp
class A { virtual void f(); };

void g(int i) {
    A* pa;
    pa = dynamic_cast<A*>(i);  // C2681
}
```

Compiler Error C2682

Error Message
cannot use casting_operator to convert from 'type1' to 'type2'

A casting operator tried to convert between incompatible types. For example, you cannot use the dynamic_cast operator to convert a pointer to a reference. The **dynamic_cast** operator cannot be used to cast away qualifiers. All qualifiers on the types must match.

You can use the **const_cast** operator to remove attributes such as **const**, **volatile**, or **__unaligned**.

The following sample generates C2682:

```
// C2682.cpp
class A { virtual void f(); };
class B: public A {};

void g(A* pa) {
    B& rb = dynamic_cast<B&>(pa); // C2682
}
```

The following sample generates C2682:

```
// C2682b.cpp
// compile with: /clr
ref struct R{};
ref struct RR : public R{};
ref struct H {
   RR^ r ;
   short s;
   int i;
};

int main() {
   H^ h = gcnew H();
   interior_ptr<int>lr = &(h->i);
   interior_ptr<short>ssr = safe_cast<interior_ptr<short> >(lr);   // C2682
   interior_ptr<short>ssr = reinterpret_cast<interior_ptr<short> >(lr);   // OK
}
```

Compiler Error C2683

Error Message
'cast' : 'type' is not a polymorphic type

You cannot use dynamic_cast to convert from a non-polymorphic class (a class with no virtual functions).

You can use static_cast to perform conversions of non-polymorphic types. However, **static_cast** does not perform a run-time check.

The following sample generates C2683:

```
// C2683.cpp
// compile with: /c
class B { };
class D : public B { };

void f(B* pb) {
   D* pd1 = dynamic_cast<D*>(pb);  // C2683
   D* pd1 = static_cast<D*>(pb);   // OK
}
```

Compiler Error C2687

Error Message
'type' : exception-declaration cannot be 'void' or denote an incomplete type or pointer or reference to an incomplete type

For a type to be part of an exception declaration, it must be defined and not void.

The following sample generates C2687:

```
// C2687.cpp
class C;

int main() {
   try {}
   catch (C) {}   // C2687 error
}
```

Possible resolution:

```
// C2687b.cpp
// compile with: /EHsc
class C {};

int main() {
   try {}
   catch (C) {}
}
```

Compiler Error C2688

Error Message
'C2::fgrv' : covariant returns with multiple or virtual inheritance not supported for varargs functions

Covariant return types are not supported in Visual C++ when a function contains variable arguments.

To resolve this error, either define your functions so that they do not use variable arguments or make the return values the same for all virtual functions.

The following sample generates C2688:

```
// C2688.cpp
struct G1 {};
struct G2 {};
struct G3 : G1, G2 {};
struct G4 {};
struct G5 {};
struct G6 : G4, G5 {};
struct G7 : G3, G6 {};

struct C1 {
   virtual G4& fgrv(int,...);
};

struct C2 : C1 {
   virtual G7& fgrv(int,...);   // C2688, does not return G4&
};
```

Compiler Error C2689

Error Message
'function' : a friend function cannot be defined within a local class

You can declare but not define a friend function in a local class.

The following sample generates C2689:

```
// C2689.cpp
// compile with: /c
void g() {
   void f2();
   class X {
      friend void f2(){}   // C2689
      friend void f2();   // OK
   };
}
```

Compiler Error C2690

Error Message

'operator' : cannot perform pointer arithmetic on a managed array

Pointer arithmetic is not allowed on a managed array. Use array index notation to traverse the array.

The following sample generates C2690:

```
// C2690.cpp
// compile with: /clr
int main() {
   array<int> ^p = gcnew array<int>(5);
   for (int i = 0 ; i < 5 ; i++)
      p[i] = i;
   for (i = 0 ; i < 5 ; i++) {
      System::Console::WriteLine(p++);   // C2690
      System::Console::WriteLine(p[i]);   // OK
   }
}
```

Managed Extensions for C++

Pointer arithmetic is not allowed on a __gc array. Use array index notation to traverse the array.

The following sample generates C2690:

```
// C2690b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;

int main() {
   String* x[] = new String*[10];
   x[0] = "test";
   Console::WriteLine(x[0]);
   x++;   // C2690
}
```

Compiler Error C2691

Error Message
'data type' : a managed array cannot have this element type

The type of a managed array element can be a value type or a reference type.

The following sample generates C2691:

```
// C2691a.cpp
// compile with: /clr
class A {};

int main() {
   array<A>^ a1 = gcnew array<A>(20);   // C2691
   array<int>^ a2 = gcnew array<int>(20);   // value type OK
}
```

The following sample generates C2691:

```
// C2691b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
int main() {
   int * a1 __gc[];   // C2691
   int * a1 = new int [20];   // OK
}
```

C2691 can also occur if you attempt to define a jagged array using Managed Extensions for C++. Jagged arrays are supported in the current syntax, see array (Visual C++) for more information.

The following sample generates C2691:

```
// C2691c.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;
int main() {
   Int32 myJaggedArray[][] = new Int32 [50][];   // C2691
}
```

Compiler Error C2692

Error Message

'function_name' : fully prototyped functions required in C compiler with the '/clr' option

When compiling for .NET managed code, the C compiler requires ANSI function declarations. In addition, if a function takes no parameters, it must explicitly declare **void** as the parameter type.

Compiler Error C2693

Error Message

'operator' : illegal comparison for references to a managed array

You cannot test a managed array for any kind of inequality. For example, you can test to see if managed arrays are equal but you cannot test to see if one array is greater or less than another array.

For example, the following sample generates C2693:

```
// C2693.cpp
// compile with: /clr /c

int func3(array<int> ^a, array<int> ^b) {
   return a < b;   // C2693
}
int func1(array<int> ^a, array<int> ^b) {
   return a != b;   // OK
}
int func2(array<int> ^a, array<int> ^b) {
   return a == b;   // OK
}
```

Managed Extensions for C++

You cannot test a __gc array for any kind of inequality. For example, you can test to see if managed arrays are equal but you cannot test to see if one array is greater or less than another array.

The following sample generates C2693:

```
// C2693b.cpp
// compile with: /clr:oldSyntax /c
#using <mscorlib.dll>
int func3(int a __gc[], int b __gc[]) {
   return a < b;    // C2693
}
int func1(int a __gc[], int b __gc[]) {
   return a != b;   // OK
}

int func2(int a __gc[], int b __gc[]) {
   return a == b;   // OK
}
```

Compiler Error C2694

Error Message
'override': overriding virtual function has less restrictive exception specification than base class virtual member function 'base'

A virtual function was overridden, but under /Za, the overriding function had a less restrictive exception specification.

The following sample generates C2694:

```
// C2694.cpp
// compile with: /Za /c
class MyBase {
public:
   virtual void f(void) throw(int) {
   }
};

class Derived : public MyBase {
public:
   void f(void) throw(...) {}   // C2694
   void f2(void) throw(int) {}   // OK
};
```

Compiler Error C2695

Error Message
'function1': overriding virtual function differs from 'function2' only by calling convention

The signature of a function in a derived class cannot override a function in a base class and change the calling convention.

The following sample generates C2695:

```
// C2695.cpp
class C {
   virtual void __fastcall func();
};

class D : public C {
   virtual void __clrcall func();   // C2695
};
```

Compiler Error C2696

Error Message
Cannot create a temporary object of a managed type 'type'

References to **const** in an unmanaged program cause the compiler to call the constructor and create a temporary object on the stack. However, a managed class can never be created on the stack.

C2696 is only reachable using **/clr:oldSyntax**.

The following sample generates C2696:

```
// C2696b.cpp
// compile with: /clr:oldSyntax

__gc class G {
public:
   G( int i ) {}
};

void func( const G& g );

int main() {
   func( 1 );   // C2696
   G *myG = new G( 1 );   // OK
   func( *myG );
}
```

Compiler Error C2697

Error Message
'array' : must explicitly specify __gc or __nogc for an array declared in a managed type

This error indicates that an array was not explicitly marked as unmanaged or managed in a __gc class.

C2697 is only reachable using **/clr:oldSyntax**.

The following sample generates C2697:

```
// C2697.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
__gc class a {
public:
   int gc_arr[12];  // C2697
   // try the following line instead
   // int gc_arr __gc [];

   // example of a static array
   static int * static_gcarr = gc new int [10];

   // example of a __nogc array
   int nogc_arr __nogc [2];
};

int main() {
   a *mya = new a;
   mya -> gc_arr = new int __gc [2];

   mya -> gc_arr[0] = 33;
   System::Console::WriteLine(mya -> gc_arr[0]);

   mya -> static_gcarr[0] = 44;
   System::Console::WriteLine(mya -> static_gcarr[0]);

   mya -> gc_arr[0] = 55;
   System::Console::WriteLine(mya -> gc_arr[0]);
}
```

Compiler Error C2698

Error Message
the using-declaration for 'declaration 1' cannot co-exist with the existing using-declaration for 'declaration 2'

Once you have a using declaration for a data member, any using declaration in the same scope that uses the same name is not permitted, as only functions can be overloaded.

The following sample generates C2698:

```
// C2698.cpp
struct A {
   int x;
};

struct B {
   int x;
};

struct C : A, B {
   using A::x;
   using B::x;   // C2698
}
```

Compiler Errors C2700 Through C2799

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, or type the error number (C2756, for example) in the index.

Compiler Error C2700

Compiler Error C2701

Compiler Error C2702

Compiler Error C2703

Compiler Error C2704

Compiler Error C2705

Compiler Error C2706

Compiler Error C2707

Compiler Error C2708

Compiler Error C2709

Compiler Error C2710

Compiler Error C2711

Compiler Error C2712

Compiler Error C2713

Compiler Error C2714

Compiler Error C2715

Compiler Error C2716

Compiler Error C2717

Compiler Error C2718

Compiler Error C2719

Compiler Error C2720

Compiler Error C2721

Compiler Error C2722

Compiler Error C2723

Compiler Error C2724

Compiler Error C2725

Compiler Error C2726

Compiler Error C2727

Compiler Error C2728

Compiler Error C2730

Compiler Error C2731

Compiler Error C2732

Compiler Error C2733

Compiler Error C2734

Compiler Error C2735

Compiler Error C2736
Compiler Error C2737
Compiler Error C2738
Compiler Error C2739
Compiler Error C2743
Compiler Error C2745
Compiler Error C2748
Compiler Error C2749
Compiler Error C2750
Compiler Error C2751
Compiler Error C2752
Compiler Error C2753
Compiler Error C2754
Compiler Error C2755
Compiler Error C2756
Compiler Error C2757
Compiler Error C2758
Compiler Error C2760
Compiler Error C2761
Compiler Error C2762
Compiler Error C2764
Compiler Error C2765
Compiler Error C2766
Compiler Error C2767
Compiler Error C2768
Compiler Error C2770
Compiler Error C2771
Compiler Error C2773
Compiler Error C2774
Compiler Error C2775
Compiler Error C2776
Compiler Error C2777
Compiler Error C2778
Compiler Error C2779
Compiler Error C2780
Compiler Error C2781
Compiler Error C2782
Compiler Error C2783
Compiler Error C2784
Compiler Error C2785

Compiler Error C2786

Compiler Error C2787

Compiler Error C2788

Compiler Error C2790

Compiler Error C2791

Compiler Error C2792

Compiler Error C2793

Compiler Error C2794

Compiler Error C2795

Compiler Error C2798

Compiler Error C2700

Error Message

'identifier' : cannot be thrown (use /W4 for more info)

The object cannot be thrown. Compile with /W4 for more diagnostic information.

Compiler Error C2701

Error Message
'function' : a function template cannot be a friend of a local class

A local class cannot have a template function as a friend function.

The following sample generates C2701:

```
// C2701.cpp
// compile with: /c
template<typename T>   // OK
void f1(const T &);

void MyFunction() {
   class MyClass {
      template<typename T> friend void f2(const T &);   // C2701
   };
}
```

Compiler Error C2702

Error Message
__except may not appear in termination block

An exception handler (**__try**/**__except**) cannot be nested inside a **__finally** block.

The following sample generates C2702:

```
// C2702.cpp
// processor: x86 IPF
int Counter;
int main() {
   __try {}
   __finally {
      __try {}   // C2702
      __except( Counter ) {}   // C2702
   }
}
```

Compiler Error C2703

Error Message
illegal __leave statement

A **__leave** statement must be inside a **__try** block.

The following sample generates C2703:

```
// C2703.cpp
int main() {
   __leave;   // C2703
   __try {
      // try the following line instead
      __leave;
   }
   __finally {}
}
```

Compiler Error C2704

Error Message

'identifier' : __va_start intrinsic only allowed in varargs

The **__va_start** intrinsic is used in a declaration for a function with a fixed number of arguments.

Compiler Error C2705

Error Message
'label' : illegal jump into 'exception handler block' scope

Execution jumps to a label within a **try**/**catch**, **__try**/**__except**, **__try**/**__finally** block. For more information, see Exception Handling.

The following sample generates C2705:

```
// C2705.cpp
int main() {
goto trouble;
   __try {
      trouble: ;   // C2705
   }
   __finally {}

   // try the following line instead
   // trouble: ;
}
```

Compiler Error C2706

Error Message
illegal __except without matching __try (missing '}' in __try block?)

The compiler did not find a closing brace for a **__try** block.

The following sample generates C2706:

```
// C2706.cpp
int main() {
   __try {
      void f();
   // C2706  } missing here
   __except(GetExceptionCode() == 0x0) {
   }
}
```

Compiler Error C2707

Error Message
'identifier' : bad context for intrinsic function

Structured exception-handling intrinsics are invalid in certain contexts:

- **_exception_code()** outside an exception filter or **__except** block
- **_exception_info()** outside an exception filter
- **_abnormal_termination()** outside a **__finally** block

To resolve the error, be sure that the exception-handling intrinsics are placed in the appropriate context.

Example
The following sample generates C2707.

```
// C2707.cpp
#include <windows.h>
#include <stdio.h>

LONG MyFilter(LONG excode)
{
    return (excode == EXCEPTION_ACCESS_VIOLATION ?
        EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH);   // OK
}

LONG func(void)
{
    int x, y;
    return(GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ?  // C2707
             EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH);

    __try
    {
        y = 0;
        x = 4 / y;
        return 0;
     }

    __except(MyFilter(GetExceptionCode()))
    {
        return(GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ? // ok
               EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH);
    }
}

int main()
{
    __try
    {
        func();
    } __except(EXCEPTION_EXECUTE_HANDLER)
    {
        printf_s("Caught exception\n");
    }
}
```

Compiler Error C2708

Error Message
'identifier' : actual parameters length in bytes differs from previous call or reference

A __stdcall function must be preceded by a prototype. Otherwise, the compiler interprets the first call to the function as a prototype and this error occurs when the compiler encounters a call that does not match.

To fix this error add a function prototype.

Compiler Error C2709

Error Message

'identifier' : formal parameter's length in bytes differs from previous declaration

The signature in a call to the specified function differs from the prototype.

Compiler Error C2710

Error Message
'construct' : '__declspec(modifier)' can only be applied to a function returning a pointer

A function whose return value is a pointer is the only construct to which *modifier* can be applied.

The following sample generates C2710:

```
// C2710.cpp
__declspec(restrict) void f();   // C2710
// try the following line instead
__declspec(restrict) int * g();
```

Compiler Error C2711

Error Message
'function' : this function cannot be compiled as managed, consider using #pragma unmanaged

Some instructions will prevent the compiler from generating MSIL for the enclosing function.

The following sample generates C2711:

```
// C2711.cpp
// compile with: /clr
// processor: x86
using namespace System;
value struct V {
   static const t = 10;
};

void bar() {
   V::t;
   __asm int 3   // C2711 inline asm can't be compiled managed
}
```

Compiler Error C2712

Error Message
cannot use __try in functions that require object unwinding

With /EHsc, a function with structured exception handling cannot have objects that require unwinding (destruction).

Possible solutions:

- Move code that requires SEH to another function.
- Rewrite functions that use SEH to avoid the use of local variables and parameters that have destructors. Do not use SEH in constructors or destructors.
- Compile without /EHsc.

Example

C2712 can occur if you compile with **/clr:pure** and declare a static array of pointers-to-functions in a **__try** block. A static member requires the compiler to use dynamic initialization under **/clr:pure**, which implies C++ exception handling. However, C++ exception handling is not allowed in a **__try** block.

The following sample generates C2712.

```
// C2712.cpp
// compile with: /clr:pure /c
struct S1 {
   static int smf();
   void fnc();
};

void S1::fnc() {
   __try {
      static int (*array_1[])() = {smf,};   // C2712

      // OK
      static int (*array_2[2])();
      array_2[0] = smf;
    }
    __except(0) {}
}
```

Compiler Error C2713

Error Message

only one form of exception handling permitted per function

You cannot use structured exception handling (**__try**/**__except**) and C++ exception handling (**try**/**catch**) in the same function.

Compiler Error C2714

Error Message

__alignof(void) is not allowed

An invalid value was passed to an operator.

See __alignof Operator for more information.

Example

The following sample generates C2714.

```
// C2714.cpp
int main() {
   return __alignof(void);   // C2714
   return __alignof(char);   // OK
}
```

Compiler Error C2715

Error Message
'type' : cannot throw or catch this type

Value types are not valid arguments when using exception handling in managed code (see Exception Handling under /clr for more information).

```
// C2715a.cpp
// compile with: /clr
using namespace System;

value struct V {
   int i;
};

void f1() {
   V v;
   v.i = 10;
   throw v;   // C2715
   // try the following line instead
   // throw ((V^)v);
}

int main() {
   try {
      f1();
   }

   catch(V v) { if ( v.i == 10 ) {   // C2715
   // try the following line instead
   // catch(V^ pv) { if ( pv->i == 10 ) {
         Console::WriteLine("caught 10 - looks OK");
      }
      else {
         Console::WriteLine("catch looks bad");
      }
   }
   catch(...) {
      Console::WriteLine("catch looks REALLY bad");
   }
}
```

__value types or __gc pointers are not valid arguments when using exception handling in Managed Extensions for C++. To resolve this error, use the __box keyword to box the argument.

The following sample generates C2715:

```
// C2715b.cpp
// compile with: /clr:oldSyntax
using namespace System;

__value struct V {
   int i;
};

void f1() {
   V v;
   v.i = 10;
   throw v;   // C2715
   // try the following line instead
   // throw __box(v);
}
```

```
int main() {
   try {
      f1();
   }

   catch(V v) { if ( v.i == 10 ) {   // C2715
   // try the following line instead
   // catch(__box V *pv) { if ( pv->i == 10 ) {
         Console::WriteLine(S"caught 10 - looks OK");
      }
      else {
         Console::WriteLine(S"catch looks bad");
      }
   }
   catch(...) {
      Console::WriteLine(S"catch looks REALLY bad");
   }
}
```

Compiler Error C2716

Error Message
'operator new' allocates value types on the C++ heap; use '__nogc new type'

The syntax of an object instantiation was incorrect.

C2716 is only reachable using **/clr:oldSyntax**.

The following sample generates C2716:

```
// C2716.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
__value struct V {};

int main() {
   V *pV = new V;   // C2716
   V *pV1 = __nogc new V;   // OK
}
```

Compiler Error C2717

Error Message
'__nogc new' cannot be used to create a managed type object

__nogc new can only be used to create instances of unmanaged classes and value types. For more information, see __nogc.

C2717 is only reachable using **/clr:oldSyntax**.

The following sample generates C2717:

```
// C2717.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;

__gc class G {};

int main() {
   G* pG = __nogc new G;   // C2717
   G* pG2 = new G;   // OK

   G* arr __gc[] = __nogc new G*[10];   // C2717
   G* arr2 __gc[] = new G*[10];   // OK
}
```

Compiler Error C2718

Error Message
'parameter': actual parameter with __declspec(align('#')) won't be aligned

The align **__declspec** modifier is not permitted on function parameters.

The following sample generates C2718:

```
// C2718.cpp
typedef struct __declspec(align(32)) AlignedStruct  {
   int i;
} AlignedStruct;

void f2(int i, ...);

void f4() {
   AlignedStruct as;

   f2(0, as);   // C2718, actual parameter is aligned
}
```

Compiler Error C2719

Error Message
'parameter': formal parameter with __declspec(align('#')) won't be aligned

The align **__declspec** modifier is not permitted on function parameters.

The following sample generates C2719:

```
// C2719.cpp
void func(int __declspec(align(32)) i);   // C2719
// try the following line instead
// void func(int i);
```

Compiler Error C2720

Error Message
'identifier' : 'specifier' storage-class specifier illegal on members

The storage class cannot be used on class members.

The following sample generates C2720

```
// C2720.cpp
struct S {
   static int i;
};
static S::i;   // C2720
```

Compiler Error C2721

Error Message

'specifier' : storage-class specifier illegal between operator keyword and type

User-defined type conversions apply to all storage classes, so you cannot specify a storage class in a type conversion.

Compiler Error C2722

Error Message

'::operator' : illegal following operator command; use 'operator operator'

An **operator** statement redefines **::new** or **::delete**. The **new** and **delete** operators are global, so the scope resolution operator (**::**) is meaningless. Remove the **::** operator.

Compiler Error C2723

Error Message
'function' : 'specifier' storage-class specifier illegal on function definition

The specifier cannot appear with a function definition outside of a class declaration. The **virtual** specifier can be specified only on a member function declaration within a class declaration.

The following sample generates C2723:

```
// C2723.cpp
struct X {
   virtual void f();
   virtual void g();
};

virtual void X::f() {}   // C2723

// try the following line instead
void X::g() {}
```

Compiler Error C2724

Error Message
'identifier' : 'static' should not be used on member functions defined at file scope

Static member functions should be declared with external linkage.

The following sample generates C2724:

```
// C2724.cpp
class C {
   static void func();
};

static void C::func(){};   // C2724
// try the following line instead
// void C::func(){};
```

Compiler Error C2725

Error Message

'exception' : unable to throw or catch a managed object by value or reference

The type of a managed exception was not correct.

Example

The following sample generates C2725.

```
// C2725.cpp
// compile with: /clr
ref class R {
public:
   int i;
};

int main() {
   R % r1 = *gcnew R;
   throw r1;   // C2725

   R ^ r2 = gcnew R;
   throw r2;   // OK
}
```

The following sample generates C2725.

```
// C2725b.cpp
// compile with: /clr
using namespace System;
int main() {
   try {}
   catch( System::Exception%) {}   // C2725
   // try the following line instead
   // catch( System::Exception ^e) {}
}
```

The following sample generates C2725.

```
// C2725c.cpp
// compile with: /clr:oldSyntax
using namespace System;
int main() {
   try {}
   catch( System::Exception&) {}   // C2725
   // try the following line instead
   // catch( System::Exception *e) {}
}
```

Compiler Error C2726

Error Message
'gcnew' may only be used to create an object with managed type

You cannot create an instance of a native type on the garbage-collected heap.

The following sample generates C2726:

```
// C2726.cpp
// compile with: /clr
using namespace System;
class U {};
ref class V {};
value class W {};

int main() {
   U* pU = gcnew U;    // C2726
   U* pU2 = new U;   // OK
   V^ p2 = gcnew V;   // OK
   W p3;   // OK

}
```

The following sample generates C2726:

```
// C2726b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;

class U {};
__gc class V {};

int main() {
   U* pU = __gc new U;    // C2726
   U* pU2 = new U;
   V* p2 = __gc new V;
}
```

Compiler Error C2727

Error Message

'type' : a native array cannot contain a value type with private data members or managed pointer members

Managed Extensions for C++

A __nogc array cannot be of a type that contains __gc types.

C2727 is only reachable using **/clr:oldSyntax**

The following sample generates C2727

```
// C2727b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;
__value struct V {
   String * s;
};

__value struct V2 {
   char s;
};

int main() {
   V v __nogc[5];   // C2727
   V2 v2 __nogc[5];   // OK
}
```

Compiler Error C2728

Error Message
'type' : a native array cannot contain this managed type

Array creation syntax was used to create an array of managed objects. You cannot create an array of managed objects using native array syntax.

For more information, see array.

The following sample generates C2728:

```
// C2728.cpp
// compile with: /clr

int main() {
   int^ arr[5];   // C2728

   // try the following line instead
   array<int>^arr2;
}
```

A __nogc array cannot be of a __gc type.

The following sample generates C2728:

```
// C2728_b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

int main() {
   int __gc* arr __nogc[5];   // C2728

   // try the following line instead
   int arr2 __gc[];
}
```

Compiler Error C2730

Error Message

'class' : cannot be a base class of itself

Recursive base classes are invalid. Specify another class as the base class.

Compiler Error C2731

Error Message
'identifier' : function cannot be overloaded

The functions **main**, **WinMain**, **DllMain**, and **LibMain** cannot be overloaded.

The following sample generates C2731:

```
// C2731.cpp
extern "C" void WinMain(int, char *, char *);
void WinMain(int, short, char *, char*);   // C2731
```

Compiler Error C2732

Error Message
linkage specification contradicts earlier specification for 'function'

The function is already declared with a different linkage specifier.

This error can be caused by include files with different linkage specifiers.

Change the **extern** statements so that the linkages agree.

Example
The following sample generates C2732:

```
// C2732.cpp
extern void func( void );   // implicit C++ linkage
extern "C" void func( void );   // C2732
```

Compiler Error C2733

Error Message
second C linkage of overloaded function 'function' not allowed

More than one overloaded function is declared with C linkage. When using C linkage, only one form of a specified function can be external. Since overloaded functions have the same undecorated name, they cannot be used with C programs.

The following sample generates C2733:

```
// C2733.cpp
extern "C" {
   void F1(int);
}

extern "C" {
   void F1();   // C2733
   // try the following line instead
   // void F2();
}
```

Compiler Error C2734

Error Message
'identifier' : const object must be initialized if not extern

The identifier is declared **const** but not initialized or **extern**.

The following sample generates C2734:

```
// C2734.cpp
const int j;   // C2734
extern const int i;   // OK, declared as extern
```

Compiler Error C2735

Error Message
'keyword' keyword is not permitted in formal parameter type specifier

The keyword is invalid in this context.

The following sample generates C2735:

```
// C2735.cpp
void f(inline int){}   // C2735
```

Compiler Error C2736

Error Message
'keyword' keyword is not permitted in cast

The keyword is invalid in a cast.

The following sample generates C2736:

```
// C2736.cpp
int main() {
   return (virtual) 0;   // C2736
   // try the following line instead
   // return 0;
}
```

Compiler Error C2737

Error Message

'class1' : base class 'class2' must be exported

If a derived class is exported, its base class must also be exported.

Compiler Error C2738

Error Message
'declaration' : is ambiguous or is not a member of 'type'

A function was declared incorrectly.

The following sample generates C2738:

```
// C2738.cpp
struct A {
   template <class T> operator T*();
   // template <class T> operator T();
};

template <>
A::operator int() {   // C2738

// try the following line instead
// A::operator int*() {

// or use the commented member declaration

   return 0;
}
```

Compiler Error C2739

Error Message
'number' : explicit managed array dimensions must be between 1 and 32

An array dimension was not between 1 and 32.

The following sample generates C2739:

```
// C2739.cpp
// compile with: /clr
int main() {
   array<int, -1>^a;   // C2739
   // try the following line instead
   // array<int, 2>^a;
}
```

Compiler Error C2743

Error Message
'type' : cannot catch a native type with __clrcall destructor or copy constructor

A module compiled with **/clr** (not **/clr:pure**) attempted to catch an exception of native type and where the type's destructor or copy constructor uses **__clrcall** calling convention.

When compiled with **/clr** (not **/clr:pure**), exception handling expects the member functions in a native type to be __cdecl and not __clrcall. Native types with member functions using **__clrcall** calling convention cannot be caught in a module compiled with **/clr**.

For more information, see /clr (Common Language Runtime Compilation).

Example
The following sample generates C2743.

```
// C2743.cpp
// compile with: /clr
public struct S {
   __clrcall ~S() {}
};

public struct T {
   ~T() {}
};

int main() {
   try {}
   catch(S) {}   // C2743
   // try the following line instead
   // catch(T) {}

   try {}
   catch(S*) {}   // OK
}
```

Compiler Error C2745

Error Message
'token' : this token cannot be converted to an identifier

Identifiers must be comprised of legal characters.

The following sample generates C2745:

```
// C2745.cpp
// compile with: /clr
int main() {
   int __identifier([]);   // C2745
}
```

Compiler Error C2748

Error Message
managed array creation must have array size or array initializer

A managed array was ill formed. For more information, see array.

The following sample generates C2748:

```
// C2748.cpp
// compile with: /clr
int main() {
   array<int> ^p1 = new array<int>();   // C2748
   // try the following line instead
   array<int> ^p2 = new array<int>(2);
}
```

Compiler Error C2749

Error Message

'type' : can only throw or catch handle to a managed class with /clr:safe

When using **/clr:safe**, you can only throw or catch a reference type.

For more information, see /clr (Common Language Runtime Compilation).

Example

The following sample generates C2749:

```
// C2749.cpp
// compile with: /clr:safe
ref struct MyStruct {
public:
   int i;
};

int main() {
   MyStruct ^x = gcnew MyStruct;

   // Delete the following 4 lines to resolve.
   try {
      throw (1);   // C2749
   }
   catch(int){}

   // OK
   try {
      throw (x);
   }
   catch(MyStruct ^){}
}
```

Compiler Error C2750

Error Message

'type' : cannot use 'new' on the reference type; use 'gcnew' instead

To create an instance of a CLR type, which causes the instance to be placed on the garbage-collected heap, you must use gcnew.

The following sample generates C2750:

```
// C2750.cpp
// compile with: /clr
ref struct Y1 {};

int main() {
   Y1 ^ x = new Y1;   // C2750

   // try the following line instead
   Y1 ^ x2 = gcnew Y1;
}
```

Compiler Error C2751

Error Message

'parameter' : the name of a function parameter cannot be qualified

You cannot use a qualified name as a function parameter.

The following sample generates C2751:

```
// C2751.cpp
namespace std {
   template<typename T>
   class list {};
}

#define list std::list
void f(int &list){}   // C2751
```

Compiler Error C2752

Error Message
'template' : more than one partial specialization matches the template argument list

An instantiation was ambiguous.

The following sample generates C2752:

```
// C2752.cpp
template<class T, class U>
struct A {};

template<class T, class U>
struct A<T*, U> {};

template<class T, class U>
struct A<T,U*> {};

// try the following line instead
// template<class T, class U> struct A<T*,U*> {};

int main() {
   A<char*,int*> a;   // C2752 an instantiation

   // OK
   A<char*,int> a1;
   A<char,int*> a2;
   A<char,int> a3;
}
```

Compiler Error C2753

Error Message
'class' : template class has already been defined

If the template argument list matches the parameter list, the compiler treats it as the same template. Defining the same template twice is not allowed.

The following sample generates C2753:

```
// C2753.cpp
template<class T>
struct A {};

template<class T>
struct A<T> {};   // C2753
// try the following line instead
// struct A<int> {};

template<class T, class U, class V, class W, class X>
struct B {};
```

Compiler Error C2754

Error Message
'specialization' : a partial specialization cannot have a dependent non-type template parameter

An attempt was made to partially specialize a template class that has a dependent non-type template parameter. This is not allowed.

The following sample generates C2754:

```
// C2754.cpp
// compile with: /c

template<class T, T t>
struct A {};

template<class T, int N>
struct B {};

template<class T> struct A<T,5> {};   // C2754
template<> struct A<int,5> {};   // OK
template<class T> struct B<T,5> {};   // OK
```

Compiler Error C2755

Error Message
'param' : non-type parameter of a partial specialization must be a simple identifier

The non-type parameter needs to be a simple identifier, something the compiler can resolve at compile time to a single identifier or a constant value.

The following sample generates C2755:

```
// C2755.cpp
template<int I, int J>
struct A {};

template<int I>
struct A<I,I*5> {};   // C2755
// try the following line instead
// struct A<I,5> {};
```

Compiler Error C2756

Error Message
'template type' : default arguments not allowed on a partial specialization

The template for a partial specialization may not contain a default argument.

The following sample generates C2756:

```
// C2756.cpp
template <class T>
struct S {};

template <class T=int>
// try the following line instead
// template <class T>
struct S<T*> {};   // C2756
```

Compiler Error C2757

Error Message
'symbol' : a symbol with this name already exists and therefore this name cannot be used as a namespace name

A symbol used in the current compilation as a namespace identifier is already being used in a referenced assembly.

The following sample generates C2757:

```
// C2757a.cpp
// compile with: /clr /LD
public ref class Nes {};
```

And then,

```
// C2757b.cpp
// compile with: /clr /c
#using <C2757a.dll>

namespace Nes {    // C2757
// try the following line instead
// namespace Nes2 {
   public ref class X {};
}
```

The following sample generates C2757:

```
// C2757c.cpp
// compile with: /clr:oldSyntax /LD
#using <mscorlib.dll>
public __gc class Nes {};
```

And then,

```
// C2757d.cpp
// compile with: /clr:oldSyntax /c
#using <C2757c.dll>
#using <mscorlib.dll>

namespace Nes {    // C2757
// try the following line instead
// namespace Nes2 {
   public __gc class X {};
}
```

Compiler Error C2758

Error Message
'const' : must be initialized in constructor base/member initializer list

The constructor does not initialize the **const** variable in an initializer list. The compiler leaves the constant undefined. Reference and **const** member variables must be given a value when initialized or in the constructor.

The following sample generates C2758:

```
// C2758.cpp
struct A {
   const int i;

   A(int i) {};   // C2758
   // try the following line instead
   // A(int i) : i(0) {};
};
```

Compiler Error C2760

Error Message
syntax error : expected 'name1' not 'name2'

A casting operator is used with an invalid operator.

The following sample generates C2760:

```
// C2760.cpp
class B {};
class D : public B {};

void f(B* pb) {
    D* pd1 = static_cast<D*>(pb);
    D* pd2 = static_cast<D*>=(pb);  // C2760
    D* pd3 = static_cast<D*=(pb);   // C2760
}
```

Compiler Error C2761

Error Message

'function' : member function redeclaration not allowed

You cannot redeclare a member function. You can define it, but not redeclare it.

Example

The following sample generates C2761.

```
// C2761.cpp
class a {
   int t;
   void test();
};

void a::a;     // C2761
void a::test;  // C2761
```

Nonstatic members of a class or structure cannot be defined. The following sample generates C2761.

```
// C2761_b.cpp
// compile with: /c
struct C {
   int s;
   static int t;
};

int C::s;   // C2761
int C::t;   // OK
```

Compiler Error C2762

Error Message
'class' : invalid expression as a template argument for 'argument'

When using /Za, the compiler will not convert an integral to a pointer.

The following sample generates C2762:

```
// C2762.cpp
// compile with: /Za
template<typename T, T *pT>
class X2 {};

void f2() {
   X2<int, 0> x21;   // C2762
   // try the following line instead
   // X2<int, static_cast<int *>(0)> x22;
}
```

Compiler Error C2764

Error Message

'param' : template parameter not used or deducible in partial specialization 'specialization'

A template parameter is not used in a partial specialization. This makes the partial specialization unusable because the template parameter cannot be deduced.

Example

The following sample generates C2764:

```
// C2764.cpp
#include <stdio.h>
template <class T1, class T2>
struct S  {
   int m_i;
};

template <class T1, class T2>
struct S<int, T2*> {   // C2764
// try the following line instead
// struct S<T1(*)(T2), T2*> {
   char m_c;
};

int main() {
   S<int, char> s1;
   S<void (*)(short), short *> s2;
   s2.m_c = 10;
   s1.m_i = s2.m_c;
   printf_s("%d\n", s1.m_i);
}
```

Compiler Error C2765

Error Message
'function' : an explicit specialization of a function template cannot have any default arguments

Default arguments are not allowed on an explicit specialization of a function template. For more information, see Explicit Specialization of Function Templates.

The following sample generates C2765:

```
// C2765.cpp
template<class T> void f(T t) {};

template<> void f<char>(char c = 'a') {}   // C2765
// try the following line instead
// template<> void f<char>(char c) {}
```

Compiler Error C2766

Error Message
explicit specialization; 'specialization' has already been defined

Duplicate explicit specializations are not allowed. For more information, see Explicit Specialization of Function Templates.

The following sample generates C2766:

```
// C2766.cpp
// compile with: /c
template<class T>
struct A {};

template<>
struct A<int> {};

template<>
struct A<int> {};   // C2766
// try the following line instead
// struct A<char> {};
```

Compiler Error C2767

Error Message
managed array dimension mismatch : expected number argument(s) - number provided

A managed array declaration was ill formed. For more information, see array.

The following sample generates C2767:

```
// C2767.cpp
// compile with: /clr
int main() {
   array<int> ^p1 = new array<int>(2,3); // C2767
   array<int> ^p2 = new array<int>(2);   // OK
}
```

Compiler Error C2768

Error Message
'function' : illegal use of explicit template arguments

The compiler was unable to determine if a function definition was supposed to be an explicit specialization of a function template or if the function definition was supposed to be for a new function.

This error was introduced in Visual Studio .NET 2003, as part of the compiler conformance enhancements.

See Summary of Compile-Time Breaking Changes for more information.

The following sample generates C2768:

```
// C2768.cpp
template<typename T>
void f(T) {}

void f<int>(int) {}   // C2768

// an explicit specialization
template<>
void f<int>(int) {}

// global nontemplate function overload
void f(int) {}
```

Compiler Error C2770

Error Message
invalid explicit template_or_generic argument(s) for 'template'

Function template candidates with explicit template or generic arguments resulted in disallowed function types.

The following sample generates C2770:

```
// C2770.cpp
#include <stdio.h>
template <class T>
int f(typename T::B*);   // expects type with member B

struct Err {};

int main() {
   f<int>(0);   // C2770 int has no B
   // try the following line instead
   f<OK>(0);
}
```

Compiler Error C2771

Error Message
#import only permitted at global or namespace scope

The **#import** directive is not allowed in, for example, a function or structure.

Compiler Error C2773

Error Message
#import and #using available only in C++ compiler

The C compiler does not recognize the **#import** preprocessor directive. Compile the source as C++. Use /TP if necessary.

Compiler Error C2774

Error Message
'identifier' : no 'put' method is associated with this property

A data member declared with property has no **put** function, but an expression tries to set its value.

The following sample generates C2774:

```
// C2774.cpp
struct A {
   __declspec(property(get=GetProp)) int prop;
   int GetProp(void);

   __declspec(property(get=GetProp2, put=PutProp2)) int prop2;
   int GetProp2(void);
   void PutProp2(int);
};

int main() {
   A* pa = new A;
   int val = 0;
   pa->prop = val;   // C2774
   pa->prop++;   // C2774
}
```

Compiler Error C2775

Error Message
'identifier' : no 'get' method is associated with this property

A data member declared with the property extended attribute does not have a **get** function specified, but an expression tries to retrieve its value.

The following sample generates C2775:

```
// C2775.cpp
struct A {
   __declspec(property(put=PutProp2, get=GetProp2)) int prop2;
   int GetProp2(){return 0;}
   void PutProp2(int){}

   __declspec(property(put=PutProp)) int prop;
   int PutProp(void){}

};

int main() {
   A* pa = new A;
   int x;
   x = pa->prop;   // C2775
   x = pa->prop2;
}
```

Compiler Error C2776

Error Message
only one 'get' method can be specified per property

You can only specify one **get** function in the property extended attribute. This error occurs when multiple **get** functions are specified.

The following sample generates C2776:

```
// C2776.cpp
struct A {
   __declspec(property(get=GetProp,get=GetPropToo))
   // try the following line instead
   // __declspec(property(get=GetProp))
      int prop;   // C2776
   int GetProp(void);
   int GetPropToo(void);
};
```

Compiler Error C2777

Error Message
only one 'put' method can be specified per property

A property (C++) declspec modifier had more than one **put** property.

The following sample generates C2777:

```
// C2777.cpp
struct A {
   __declspec(property(put=PutProp,put=PutPropToo))   // C2777
   // try the following line instead
   // __declspec(property(put=PutProp))
      int prop;
   int PutProp(void);
   int PutPropToo(void);
};
```

Compiler Error C2778

Error Message
improperly formed GUID in __declspec(uuid())

An incorrect GUID is supplied to the uuid extended attribute.

The GUID must be a string of hexadecimal numbers with the following format:

```
// C2778a.cpp
// compile with: /c
struct __declspec(uuid("00000000-0000-0000-0000-000000000000")) A {};
struct __declspec(uuid("{00000000-0000-0000-0000-000000000000}")) B{};
```

The **uuid** extended attribute accepts strings recognized by CLSIDFromString, with or without brace delimiters.

The following sample generates C2778:

```
// C2778b.cpp
struct __declspec(uuid(" 00000000-0000-0000-0000-000000000000 ")) C { };   // C2778
struct __declspec(uuid("00000000000000000000000000000000")) D { };   // C2778
```

Compiler Error C2779

Error Message
'declaration' : property methods can only be associated with non-static data members

The **property** extended attribute is incorrectly applied to a static data member.

The following sample generates C2779:

```
// C2779.cpp
struct A {
   static __declspec(property(put=PutProp))
   // try the following line instead
   __declspec(property(put=PutProp))
      int prop;   // C2779
   int PutProp(void);
};
```

Compiler Error C2780

Error Message
'declaration' : expects number1 arguments - number2 provided

A function template has too few arguments.

The following sample generates C2780:

```
// C2780.cpp
template<typename T>
void f(T, T){}

int main() {
   f(1);  // C2780
   // try the following line instead
   // f(1,2);
}
```

Compiler Error C2781

Error Message
'declaration' : expects at least value1 argument - value2 provided

A function template with a variable parameter list has too few arguments.

The following sample generates C2781:

```
// C2781.cpp
template<typename T>
void f(T, T, ...){}

int main() {
   f(1);   // C2781

   // try the following line instead
   f(1,1);
}
```

Compiler Error C2782

Error Message
'declaration' : template parameter 'identifier' is ambiguous

The compiler cannot determine the type of a template argument.

The following sample generates C2782:

```
// C2782.cpp
template<typename T>
void f(T, T) {}

int main() {
   f(1, 'c');   // C2782
   // try the following line instead
   // f<int>(1, 'c');
}
```

C2782 can also occur when using generics:

```
// C2782b.cpp
// compile with: /clr
generic<typename T> void gf(T, T) { }

int main() {
   gf(1, 'c'); // C2782
   // try the following line instead
   // gf<int>(1, 'c');
}
```

C2782 can also occur when you specify the wrong generic type parameter:

```
// C2782c.cpp
// compile with: /clr
#using <System.Drawing.dll>

using namespace System;
using namespace System::Drawing;
using namespace System::Collections::Generic;

Point PointFToPoint(PointF pf) {
   return Point((int) pf.X, (int) pf.Y);
};

int main() {
   array<PointF>^ apf = { PointF(27.8F, 32.62F) };

   array<Point>^ ap =
      Array::ConvertAll<Point>(apf, gcnew Converter<PointF, Point>(&::PointFToPoint));   //
C2782

      // try one of the following lines instead
      // Array::ConvertAll(apf, gcnew Converter<PointF, Point>(PointFToPoint));
      // Array::ConvertAll<PointF>(apf, gcnew Converter<PointF, Point>(PointFToPoint));
      // Array::ConvertAll<PointF, Point>(apf, gcnew Converter<PointF, Point>(PointFToPoint
));

   for each(Point p in ap)
      Console::WriteLine(p);
}
```

Compiler Error C2783

Error Message
'declaration' : could not deduce template argument for 'identifier'

The compiler cannot determine a template argument. Default arguments cannot be used to deduce a template argument.

The following sample generates C2783:

```
// C2783.cpp
template<typename T1, typename T2>
T1 f(T2) {
   return 248;
}

int main() {
   f(1);   // C2783
   // try the following line instead
   int i = f<int>(1);
}
```

C2783 can also occur when using generics:

```
// C2783b.cpp
// compile with: /clr
using namespace System;
generic<typename T1, typename T2>
T1 gf(T2) {
   T1 t1 = safe_cast<T1>( Activator::CreateInstance(T1::typeid));
   return t1;
}

ref class MyClass{};

int main() {
   int i;
   i = gf(9);   // C2783

   // OK
   i = gf<int>(9);
}
```

Compiler Error C2784

Error Message
'declaration' : could not deduce template argument for 'type' from 'type'

The compiler cannot determine a template argument from the supplied function arguments.

The following sample generates C2784:

```
// C2784.cpp
template<class T> class X {};
template<class T> void f(X<T>) {}

int main() {
   X<int> x;
   f(1);   // C2784

   // try the following line instead
   f(x);
}
```

Compiler Error C2785

Error Message
'declaration1' and 'declaration2' have different return types

The return type of function template specialization differs from the return type of the primary function template.

Check all specializations of the function template for consistency.

The following sample generates C2785:

```
// C2785.cpp
// compile with: /c
template<class T> void f(T);

template<> int f(int); // C2785
template<> void f(int); // OK
```

The following sample generates C2785:

```
// C2785_b.cpp
// compile with: /c
template <int u>
class A {
public:
   typedef void T;
};

class B {
public:
   template <typename C>
   typename C::T func (const C &);
};

template <typename C>
typename C::T B::func (const C &) {}

template
A<2>::T
B::func (const A<2> &);   // C2785
// try the following line instead
// B::func<A<2> >(const A<2> &);
```

Compiler Error C2786

Error Message
'type' : invalid operand for __uuidof

The __uuidof operator takes a user-defined type with a GUID attached or an object of such a user-defined type. Possible causes:

1. The argument is not a user-defined type.
2. **__uuidof** cannot extract the GUID from the argument.

The following sample generates C2786:

```
// C2786.cpp
struct __declspec(uuid("00000000-0000-0000-0000-000000000000")) A {};

int main() {
   __uuidof(int);   // C2786
   __uuidof(int *);   // C2786
   __uuidof(A **);   // C2786

   // no error
   __uuidof(A);
   __uuidof(A *);
   __uuidof(A &);
   __uuidof(A[]);

   int i;
   int *pi;
   A **ppa;

   __uuidof(i);      // C2786
   __uuidof(pi);     // C2786
   __uuidof(ppa);    // C2786
}
```

Compiler Error C2787

Error Message
'identifier' : no GUID has been associated with this object

The __uuidof operator takes a user-defined type with a GUID attached or an object of such a user-defined type. This error occurs when the argument is a user-defined type with no GUID.

The following sample generates C2787:

```
// C2787.cpp
#include <windows.h>
struct F {};

struct __declspec(uuid("00000000-0000-0000-c000-000000000046")) F2;

int main() {
   __uuidof(F);   // C2787
   __uuidof(F2);   // OK
}
```

Compiler Error C2788

Error Message
'identifier' : more than one GUID associated with this object

The __uuidof operator takes a user-defined type with a GUID attached or an object of such a user-defined type. This error occurs when the argument is an object with multiple GUIDs.

The following sample generates C2788:

```
// C2788.cpp
#include <windows.h>
struct __declspec(uuid("00000001-0000-0000-0000-000000000000")) A {};
struct __declspec(uuid("{00000002-0000-0000-0000-000000000000}")) B {};
template <class T, class U> class MyClass {};

typedef MyClass<A,B> MyBadClass;
typedef MyClass<A,A> MyGoodClass;

int main() {
   __uuidof(MyBadClass);    // C2788
   // try the following line instead
   __uuidof(MyGoodClass);
}
```

Compiler Error C2790

Error Message
'super' : this keyword can only be used within the body of class member function

This error message appears if the user ever tries to uses the keyword super outside of the context of a member function.

The following sample generates C2790:

```
// C2790.cpp
void f() {
   __super::g();   // C2790
}
```

Compiler Error C2791

Error Message
illegal use of 'super': 'class' does not have any base classes

The keyword super was used within the context of a member function of a class that does not have any base classes.

The following sample generates C2791:

```
// C2791.cpp
struct D {
   void mf() {
      __super::mf();   // C2791
   }
};
```

Compiler Error C2792

Error Message
'super' : this keyword must be followed by '::'

The only token that can follow the keyword **__super** is `::`.

The following sample generates C2792:

```
// C2792.cpp
struct B {
   void mf();
};

struct D : B {
   void mf() {
      __super.();   // C2792

      // try the following line instead
      // __super::mf();
   }
};
```

Compiler Error C2793

Error Message
'token' : unexpected token following '::', identifier or keyword 'operator' expected

The only tokens that can follow **__super::** are an identifier or the keyword **operator**.

The following sample generates C2793

```
// C2793.cpp
struct B {
   void mf();
};

struct D : B {
   void mf() {
      __super::(); // C2793
   }
};
```

Compiler Error C2794

Error Message
'function' : is not a member of any direct or indirect base class of 'class'

You tried to use super to call a nonexistent member function.

The following sample generates C2794

```
// C2794.cpp
struct B {
   void mf();
};

struct D : B {
   void mf() {
      __super::f();  // C2794
   }
};
```

Compiler Error C2795

Error Message
'super::function' is not a member function

This error message appears whenever you try to use super to access a member other than a member function.

Compiler Error C2798

Error Message
'super::member' is ambiguous

Multiple inherited structures contain the member you referenced with super. You could fix the error by either:

- Removing B1 or B2 from the inheritance list of D.
- Changing the name of the data member in B1 or B2.

The following sample generates C2798:

```
// C2798.cpp
struct B1 {
   int i;
};

struct B2 {
   int i;
};

struct D : B1, B2 {
   void g() {
      __super::i = 4; // C2798
   }
};
```

Compiler Errors C2800 Through C2899

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, or type the error number (C2843, for example) below or in the index.

Compiler Error C2800

Compiler Error C2801

Compiler Error C2802

Compiler Error C2803

Compiler Error C2804

Compiler Error C2805

Compiler Error C2806

Compiler Error C2807

Compiler Error C2808

Compiler Error C2809

Compiler Error C2810

Compiler Error C2811

Compiler Error C2812

Compiler Error C2814

Compiler Error C2815

Compiler Error C2817

Compiler Error C2818

Compiler Error C2819

Compiler Error C2821

Compiler Error C2823

Compiler Error C2824

Compiler Error C2825

Compiler Error C2826

Compiler Error C2827

Compiler Error C2828

Compiler Error C2829

Compiler Error C2830

Compiler Error C2831

Compiler Error C2833

Compiler Error C2834

Compiler Error C2835

Compiler Error C2837

Compiler Error C2838

Compiler Error C2839

Compiler Error C2840

Compiler Error C2842
Compiler Error C2843
Compiler Error C2844
Compiler Error C2845
Compiler Error C2846
Compiler Error C2847
Compiler Error C2849
Compiler Error C2850
Compiler Error C2854
Compiler Error C2855
Compiler Error C2856
Compiler Error C2857
Compiler Error C2858
Compiler Error C2859
Compiler Error C2860
Compiler Error C2861
Compiler Error C2862
Compiler Error C2863
Compiler Error C2864
Compiler Error C2865
Compiler Error C2866
Compiler Error C2867
Compiler Error C2868
Compiler Error C2869
Compiler Error C2870
Compiler Error C2871
Compiler Error C2872
Compiler Error C2873
Compiler Error C2874
Compiler Error C2875
Compiler Error C2876
Compiler Error C2877
Compiler Error C2878
Compiler Error C2879
Compiler Error C2881
Compiler Error C2882
Compiler Error C2883
Compiler Error C2884
Compiler Error C2885
Compiler Error C2886

Compiler Error C2888

Compiler Error C2890

Compiler Error C2891

Compiler Error C2892

Compiler Error C2893

Compiler Error C2894

Compiler Error C2896

Compiler Error C2897

Compiler Error C2898

Compiler Error C2899

Compiler Error C2800

Error Message
'operator operator' cannot be overloaded

The following operators cannot be overloaded: class member access (**.**), pointer to member (**.***), scope resolution (**::**), conditional expression (**? :**), and **sizeof**.

The following sample generates C2800:

```
// C2800.cpp
// compile with: /c
class C {
   operator:: ();   // C2800
};
```

Compiler Error C2801

Error Message
'operator operator' must be a non-static member

The following operators can be overloaded only as nonstatic members:

- Assignment **=**
- Class member access **->**
- Subscripting **[]**
- Function call **()**

Possible C2801 causes:

- Overloaded operator is not a class, structure, or union member.
- Overloaded operator is declared **static**.
- The following sample generates C2801:

```
// C2801.cpp
// compile with: /c
operator[]();   // C2801 not a member
class A {
   static operator->();   // C2801 static
   operator()();   // OK
};
```

Compiler Error C2802

Error Message
static member 'operator operator' has no formal parameters

An operator declared by a **static** member function must have at least one parameter.

The following sample generates C2802:

```
// C2802.cpp
// compile with: /clr /c
ref class A {
   static operator+ ();   // C2802
   static operator+ (A^, A^);   // OK
};
```

Compiler Error C2803

Error Message
'operator operator' must have at least one formal parameter of class type

The overloaded operator lacks a parameter of class type.

You need to pass at least one parameter by reference (not using pointers, but references) or by value to be able to write "a < b" (a and b being of type class A).

If both parameters are pointers it will be a pure comparison of pointer addresses and will not use the user-defined conversion.

The following sample generates C2803:

```
// C2803.cpp
// compile with: /c
class A{};
bool operator< (const A *left, const A *right);   // C2803
// try the following line instead
// bool operator< (const A& left, const A& right);
```

Compiler Error C2804

Error Message
binary 'operator operator' has too many parameters

The overloaded binary operator is declared with more than one parameter. The first parameter of a binary operator, whose type is the operator's enclosing type, is implied.

Example
The following sample generates C2804.

```
// C2804.cpp
// compile with: /c
class X {
public:
   X operator+ ( X , X );   // C2804 two parameters
   X operator+ ( X );   // para1 type implicitly X&, para2 type 'X'
};

int main() {
   X x, y;
   x + y;   // equivalent to x.operator+(y)
}
```

The following sample generates C2804.

```
// C2804_2.cpp
// compile with: /clr /c
ref struct Y {
   Y^ operator +(Y^ hY, int i);   // C2804
   static Y^ operator +(Y^ hY, int i);   // OK
   Y^ operator +(int i);   // OK
};
```

Compiler Error C2805

Error Message
binary 'operator operator' has too few parameters

The binary operator has no parameters.

The following sample generates C2805:

```
// C2805.cpp
// compile with: /c
class X {
public:
   X operator< ( void );   // C2805 must take one parameter
   X operator< ( X );   // OK
};
```

Compiler Error C2806

Error Message
'operator operator' has too many formal parameters

An overloaded operator has too many parameters.

The following sample generates C2806:

```
// C2806.cpp
// compile with: /c
class X {
public:
   X operator++ ( int, int );   // C2806 more than 1 parameter
   X operator++ ( int );   // OK
} ;
```

Compiler Error C2807

Error Message
the second formal parameter to postfix 'operator operator' must be 'int'

The second parameter to a postfix operator has the wrong type.

The following sample generates C2807:

```
// C2807.cpp
// compile with: /c
class X {
public:
   X operator++ ( X );   // C2807 nonvoid parameter
   X operator++ ( int );   // OK, int parameter
};
```

Compiler Error C2808

Error Message
unary 'operator operator' has too many formal parameters

The unary operator has a nonvoid parameter list.

The following sample generates C2808:

```
// C2808.cpp
// compile with: /c
class X {
public:
   X operator! ( X );   // C2808 nonvoid parameter list
   X operator! ( void );   // OK
};
```

Compiler Error C2809

Error Message
'operator operator' has no formal parameters

The operator lacks required parameters.

The following sample generates C2809:

```
// C2809.cpp
// compile with: /c
class A{};
int operator+ ();   // C2809
int operator+ (A);   // OK
```

Compiler Error C2810

Error Message
'interface' : an interface can only inherit from another interface

An interface may only inherit from another interface and may not inherit from a class or struct.

The following sample generates C2810:

```
// C2810.cpp
#include <unknwn.h>
class CBase1 {
public:
  HRESULT mf1();
  int  m_i;
};

[object, uuid="40719E20-EF37-11D1-978D-0000F805D73B"]
__interface IDerived : public CBase1 {  // C2810
// try the following line instead
// __interface IDerived {
   HRESULT mf2(void *a);
};

struct CBase2 {
   HRESULT mf1(int a, char *b);
   HRESULT mf2();
};
```

Compiler Error C2811

Error Message

'type1' : cannot inherit from 'type2', a ref class can only inherit from a ref class or interface class

You attempted to use an unmanaged class as a base class for a managed class.

The following sample generates C2811:

```
// C2811.cpp
// compile with: /clr /c
struct S{};
ref struct T {};
ref class C : public S {};   // C2811
ref class D : public T {};   // OK
```

Compiler Error C2812

Error Message
#import is not supported with /clr:pure and /clr:safe

The #import Directive is not supported with **/clr:pure** and **/clr:safe** because **#import** requires the use of native compiler support libraries.

Example
The following sample generates C2812.

```
// C2812.cpp
// compile with: /clr:pure /c
#import "importlib.tlb"   // C2812
```

Compiler Error C2814

Error Message

'member' : a native type cannot be nested within a managed type 'type'

Example

A native type cannot be nested in a CLR type. The following sample generates C2814.

```
// C2814.cpp
// compile with: /clr /c
ref class A {
   class B {};   // C2814
   ref class C {};   // OK
};
```

You must explicitly specify the "managed-ness" of an embedded type using one of the following keywords: __gc, __nogc, or __value.

The following sample generates C2814.

```
// C2814_b.cpp
// compile with: /clr:oldSyntax /c
__gc class A {
   class B {};   // C2814
   __gc class C {};   // OK
};
```

Compiler Error C2815

Error Message
'operator delete' : first formal parameter must be 'void *', but 'param' was used

Any user-defined operator delete function must take a first formal parameter of type `void *`.

The following sample generates C2815:

```
// C2815.cpp
// compile with: /c
class CMyClass {
public:
   void mf1(int *a);
   void operator delete(CMyClass *);   // C2815
   void operator delete(void *);
};
```

Compiler Error C2817

Error Message
return type for 'operator delete' must be 'void'

An overloaded operator delete function cannot return a value.

Compiler Error C2818

Error Message
application of overloaded 'operator ->' is recursive through type 'type'

A redefinition of the class member access operator contains a recursive **return** statement. To redefine the **->** operator with recursion, you must move the recursive routine to a separate function called from the operator override function.

Compiler Error C2819

Error Message
type 'type' does not have an overloaded member 'operator ->'

You need to define **operator->()** to use this pointer operation.

The following sample generates C2819:

```
// C2819.cpp
// compile with: /c
class A {
   public:
      int i;
};

class B {};

void C(B j) {
   j->i;   // C2819
}

class D {
   A* pA;

   public:
      A* operator->() {
         return pA;
      }
};

void F(D j) {
   j->i;
}
```

C2819 can also occur when using C++ Stack Semantics for Reference Types. The following sample generates C2819:

```
// C2819_b.cpp
// compile with: /clr
ref struct R {
   void Test() {}
};

int main() {
   R r;
   r->Test();   // C2819
   r.Test();   // OK
}
```

Compiler Error C2821

Error Message
first formal parameter to 'operator new' must be 'unsigned int'

The first formal parameter of the operator new must be an unsigned **int**.

The following sample generates C2821:

```
// C2821.cpp
// compile with: /c
void * operator new( /* unsigned int,*/ void * );   // C2821
void * operator new( unsigned int, void * );
```

Compiler Error C2823

Error Message
a typedef template is illegal

Templates are not allowed in typedef definitions.

The following sample generates C2823:

```
// C2823.cpp
template<class T>
typedef struct x {
   T i;   // C2823 can't use T, specify data type and delete template
   int i;   // OK
} x1;
```

Compiler Error C2824

Error Message
return type for 'operator new' must be 'void *'

With non-based pointers, overloads of operator **new** must return **void ***.

The following sample generates C2824:

```
// C2824.cpp
// compile with: /c
class   A {
   A* operator new(size_t i, char *m);   // C2824
   // try the following line instead
   // void* operator new(size_t i, char *m);
};
```

Compiler Error C2825

Error Message
var : must be a class or namespace when followed by '::'

An unsuccessful attempt was made to form a qualified name.

For example, make sure that your code does not contain a function declaration where the function name begins with ::.

Example
The following sample generates C2825:

```
// C2825.cpp
typedef int i;
int main() {
   int* p = new int;
   p->i::i();   // C2825
   // try the following line instead
   // p->i::~i();
}
```

Compiler Error C2826

Error Message
'operator' must be declared static

Methods must be declared as static if they implement a managed operator.

C2826 is only reachable using **/clr:oldSyntax**.

The following sample generates C2826:

```
// C2826.cpp
// compile with: /clr:oldSyntax /c
#using<mscorlib.dll>
using namespace System;

__value struct M {
   M op_Addition(M m1, M m2) {   // C2826
   // try the following line instead
   // static M op_Addition(M m1, M m2) {
      return m1;
   }
};
```

Compiler Error C2827

Error Message
'operator operator' cannot be globally overridden with unary form

The operator cannot have a unary form outside of an object.

To fix by using the following possible solutions

1. Make the overloaded operator local to an object.
2. Choose an appropriate unary operator to overload.

Compiler Error C2828

Error Message
'operator operator' cannot be globally overridden with binary form

The operator cannot have a binary form outside of an object.

To fix by using the following possible solutions

1. Make the overloaded operator local to an object.
2. Choose an appropriate unary operator to overload.

Compiler Error C2829

Error Message
'operator operator' cannot have a variable parameter list

Only two operators can take variable parameter lists:

- Function call **()**
- **new**

Compiler Error C2830

Error Message
only placement parameters to 'operator new' can have default values

The standard formal parameters for operator new cannot have default values. Only user-defined placement parameters can specify defaults.

Compiler Error C2831

Error Message
'operator operator' cannot have default parameters

Only three operators can have default parameters:

- new
- Assignment =
- Left parenthesis (

The following sample generates C2831:

```
// C2831.cpp
// compile with: /c
#define BINOP <=
class A {
public:
   int i;
   int operator BINOP(int x = 1) {   // C2831
   // try the following line instead
   // int operator BINOP(int x) {
      return i+x;
   }
};
```

Compiler Error C2833

Error Message
'operator operator' is not a recognized operator or type

The word **operator** must be followed by an operator that you want to override or a type you want to convert.

For a list of the operators that you can define in a managed type, see User-defined Operators.

The following sample generates C2833:

```
// C2833.cpp
// compile with: /c
class A {};

void operator ::* ();   // C2833
void operator :: ();   // OK
```

Compiler Error C2834

Error Message
'operator operator' must be globally qualified

The **new** and **delete** operators are tied to the class where they reside. Scope resolution cannot be used to select a version of **new** or **delete** from a different class. To implement multiple forms of the **new** or **delete** operator, create a version of the operator with extra formal parameters.

Compiler Error C2835

Error Message
user-defined conversion 'type' takes no formal parameters

User-defined type conversions cannot take formal parameters.

The following sample generates C2835:

```
// C2835.cpp
class A {
public:
   char v_char;

   A() {
      v_char = 'A';
   };
   operator char(char a) {   // C2835
   // try the following line instead
   // operator char() {
      return v_char + 1;
   };
};

int main() {
   A a;
}
```

Compiler Error C2837

Error Message
'identifier' : illegal local static variable in exported inline function

An external inline function attempts to declare a local **static** variable. This is not allowed.

Compiler Error C2838

Error Message
'member' : illegal qualified name in member declaration

A class, structure, or union uses a fully qualified name to redeclare a member of another class, structure, or union.

The following sample generates C2838:

```
// C2838.cpp
// compile with: /c
class Bellini {
public:
    void Norma();
};

class Bottesini {
   Bellini::Norma();  // C2838
};
```

Compiler Error C2839

Error Message
invalid return type 'type' for overloaded 'operator ->'

The **->** operator must return a class, struct, or union, or a reference to one.

Compiler Error C2840

Error Message
instruction word argument not constant

An invalid parameter was passed to an **__lfetch** intrinsic. For more information, see __lfetch, __lfetch_excl, __lfetchfault, __lfetchfault_excl.

Example
The following sample generates C2840.

```
// C2840.cpp
// compile with: /c
// processor: IPF
#include <intrin.h>
#pragma intrinsic(__lfetch)
int f(int *p, int x) {
   __lfetch(x, p);   // C2840
   __lfetch(0, p);   // OK
   return *p;
}
```

Compiler Error C2842

Error Message

'class' : a managed type may not define its own 'operator new' or 'operator delete'

You can define your own operator new or operator delete to manage memory allocation on the native heap. However, reference classes cannot define these operators because they are only allocated on the managed heap.

For more information, see User-Defined Operators.

Example

The following sample generates C2842.

```
// C2842.cpp
// compile with: /clr /c
ref class G {
   void* operator new( size_t nSize );   // C2842
};
```

The following sample generates C2842.

```
// C2842_b.cpp
// compile with: /clr:oldSyntax /c
__gc class G {
   void* operator new( size_t nSize );   // C2842
};
```

Compiler Error C2843

Error Message
'member' : cannot take the address of a non-static data member or method of a managed type

An instance is needed to take the address of nonstatic data members of a managed class or interface.

The following sample generates C2843:

```
// C2843_2.cpp
// compile with: /clr
public ref class C {
public:
   int m_i;
};

ref struct MyStruct {
   static void sf() {}
   void f() {}
};

int main() {
   MyStruct ^ps = gcnew MyStruct;
   void (__clrcall MyStruct::*F1)() = & MyStruct::f;   // C2843
   void (__clrcall MyStruct::*F2)() = & ps->f;   // C2843
   void (__clrcall MyStruct::*F3)();   // C2843

   void (__clrcall *F5)() = MyStruct::sf;   // OK
   void (__clrcall *F6)() = & ps->sf;   // OK

   interior_ptr<int> i = &C::m_i; // C2843
   C ^x = gcnew C();
   interior_ptr<int> ii = &x->m_i;
}
```

The following sample generates C2843:

```
// C2843.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
public __gc class C {
public:
   int m_i;
};

__gc struct MyStruct {
   static void sf() {}
   void f() {}
};

int main() {
   MyStruct *ps = new MyStruct;
   void (__clrcall MyStruct::*F1)() = & MyStruct::f;   // C2843
   void (__clrcall MyStruct::*F2)() = & ps->f;   // C2843
   void (__clrcall MyStruct::*F3)();   // C2843

   void (__clrcall *F5)() = MyStruct::sf;   // OK
   void (__clrcall *F6)() = & ps->sf;   // OK

   int __gc *i = &C::m_i; // C2843
   C *x = new C();
   int __gc *i = &x->m_i;
}
```

Compiler Error C2844

Error Message
'member' : cannot be a member of interface 'interface'

An interface class cannot contain a data member unless it is also a property.

Anything other than a property or member function is not allowed in an interface. Furthermore, constructors, destructors, and operators are not allowed.

The following sample generates C2844:

```
// C2844a.cpp
// compile with: /clr /c
public interface class IFace {
   int i;   // C2844
   // try the following line instead
   // property int Size;
};
```

The following sample generates C2844:

```
// C2844b.cpp
// compile with: /clr:oldSyntax /c
#using <mscorlib.dll>
__gc __interface IFace {
   int i;   // C2844
   // try the following line instead
   // __property int Size { get; set; };
};
```

Compiler Error C2845

Error Message
'operator' : pointer arithmetic not allowed on this type

You cannot increment the pointer to a managed class.

The following sample generates C2845:

```
// C2845.cpp
// compile with: /clr
ref class X {};

int main() {
   X ^pX = gcnew X;
   pX++;   // C2845
}
```

Managed Extensions for C++

You cannot increment the pointer to a __gc class. Also, string operators are only valid with **/clr** (not **/clr:oldSyntax**).

The following sample generates C2845:

```
// C2845b.cpp
// compile with: /clr:oldSyntax
using namespace System;
__gc class X {};

int main() {
   X *pX = new X;
   pX++;   // C2845

   String * a = new String("abc");
   String * b = new String("def");
   Console::WriteLine(a + b);   // C2845 not with /clr:oldSyntax
}
```

Compiler Error C2846

Error Message
'constructor' : an interface cannot have a constructor

A Visual C++ interface cannot have a constructor.

The following sample generates C2846:

```
// C2846.cpp
// compile with: /c
__interface C {
   C();   // C2846 constructor not allowed in an interface
};
```

Compiler Error C2847

Error Message
cannot apply sizeof to managed type 'class'

The sizeof operator gets the value of an object at compile time. The size of a managed class, interface, or value type is dynamic and so cannot be known at compile time.

For example, the following sample generates C2847:

```
// C2847.cpp
// compile with: /clr
ref class A {};

int main() {
   A ^ xA = gcnew A;
   sizeof(*xA);   // C2847 cannot use sizeof on managed object
}
```

The following sample generates C2847:

```
// C2847_b.cpp
// compile with: /clr:oldSyntax
__gc class A {};

int main() {
   A *xA = new A;
   sizeof(*xA);   // C2847 cannot use sizeof on managed object
}
```

Compiler Error C2849

Error Message
'destructor' : an interface cannot have a destructor

A Visual C++ interface cannot have a destructor.

The following sample generates C2849:

```
// C2849.cpp
// compile with: /c
__interface C {
   ~C();   // C2849 destructor not allowed in an interface
};
```

Compiler Error C2850

Error Message
'construct' : only allowed at file scope; may not be in a nested construct

Constructs, such as some pragmas, can only appear at global scope.

The following sample generates C2850:

```
// C2850.cpp
// #pragma hdrstop ok
namespace X {
   #pragma hdrstop   // C2850
};
```

Compiler Error C2854

Error Message
syntax error in #pragma hdrstop

The **#pragma hdrstop** gives an invalid filename. The pragma can be followed by an optional filename in parentheses and quotation marks:

The following sample generates C2854:

```
// C2854.cpp
// compile with: /c
#pragma hdrstop( "\\source\\pchfiles\\myheader.pch" ]   // C2854
// try the following line instead
// #pragma hdrstop( "\\source\\pchfiles\\myheader.pch" )
```

Compiler Error C2855

Error Message
command-line option 'option' inconsistent with precompiled header

The given command-line option differs from the option used to create the precompiled header. Regenerate the precompiled header with the given command-line option.

Compiler Error C2856

Error Message
#pragma hdrstop cannot be inside an #if block

The **hdrstop** pragma cannot be placed inside the body of a conditional compilation block.

Move the **#pragma hdrstop** statement to an area that is not contained in an **#if/#endif** block.

Compiler Error C2857

Error Message

'#include' statement specified with the /Ycfilename command-line option was not found in the source file

The /Yc option specifies the name of an include file that is not included in the source file being compiled.

This error can be caused by a **#include** statement in a conditional compilation block that is not compiled.

Compiler Error C2858

Error Message

command-line option '/Yc (/Fdfilename)' inconsistent with precompiled header, which used '/Fdfilename'

The program database specified by the Use Precompiled Header (/Yu) option is not the one specified by the previous Create Precompiled Header (/Yc) option.

Compiler Error C2859

Error Message
filename is not the type file that was used when this precompiled header was created, recreate the precompiled header.

The project database and precompiled header files must be created together to ensure consistent information. Rebuild the project to recreate the precompiled header.

For information on precompiled headers, see /Y (Precompiled Headers).

Compiler Error C2860

Error Message
'void' cannot be an argument type, except for '(void)'

Type **void** cannot be used as an argument type with other arguments.

The following sample generates C2860:

```
// C2860.cpp
// compile with: /c
void profunc1(void, int i);   // C2860
void func10(void);   // OK
```

Compiler Error C2861

Error Message
'function name' : an interface member function cannot be defined

The compiler encountered the interface keyword or deduced a struct as an interface but then found a member function definition. An interface cannot contain a definition for a member function.

Example
The following sample generates C2861:

```
// C2861.cpp
// compile with: /c
#include <objbase.h>   // required for IUnknown definition
[ object, uuid("00000000-0000-0000-0000-000000000001") ]
__interface IMyInterface : IUnknown {
   HRESULT mf(int a);
};

HRESULT IMyInterface::mf(int a) {}   // C2861
```

Compiler Error C2862

Error Message
'interface' : an interface can only have public members

Protected and private members may be accessed only from other member functions. Such members are no use in an interface, since it may not provide implementations for any of its members.

The following sample will generate C2862:

```
// C2862.cpp
// compile with: /c
#include <unknwn.h>

[object, uuid="60719E20-EF37-11D1-978D-0000F805D73B"]
__interface IMyInterface {
   HRESULT mf1(void);   // OK
protected:
   HRESULT mf2(int *b);   // C2862
private:
   HRESULT mf3(int *c);   // C2862
};
```

Compiler Error C2863

Error Message
'interface' : an interface cannot have friends

Declaring friends on an interface is not allowed.

The following sample generates C2863:

```
// C2863.cpp
// compile with: /c
#include <unknwn.h>

class CMyClass {
   void *f();
};

__interface IMyInterface {
   void g();

   friend int h();   // 2863
   friend interface IMyInterface1;  // C2863
   friend void *CMyClass::f();  // C2863
};
```

Compiler Error C2864

Error Message
'variable' : only static const integral data members can be initialized within a class

You attempted to initialize a data member that was not defined as **const** and **static**.

The following sample generates C2864:

```
// C2864.cpp
class B  {
   int i = 3;   // C2864;
   static const int i = 3;   // OK
};
```

The following sample generates C2864:

```
// C2864b.cpp
// compile with: /c
#include <stdio.h>
class BaseCls {
public:
   int Z;
};

class DervCls : public BaseCls {
public:
   char Z;
};

class UseClasses {
public:
   BaseCls * BObj = new DervCls;   // C2864
   DervCls * DObj = new DervCls;   // C2864
   // Uncomment the following 2 lines to resolve.
   // BaseCls * BObj;// = new DervCls;
   // DervCls * DObj;// = new DervCls;

   void ShowZ() {
      // Uncomment the following 2 lines to resolve.
      // BObj = new DervCls;
      // DObj = new DervCls;
      printf("Accessed via base class: %d", BObj->Z);
      printf("Accessed via derived class: %c", DObj->Z);
   }
};
```

Compiler Error C2865

Error Message

'function' : illegal comparison for handle_or_pointer

You can compare references to Classes and Structs (Managed) or __gc types only for equality to see if they refer to the same object (==) or to different objects (!=).

You cannot compare them for ordering because the .NET runtime might move managed objects at any time, changing the outcome of the test.

Compiler Error C2866

Error Message
'variable' : a const static data member of a managed type must be initialized at the point of declaration

It is invalid to declare an uninitialized, **const static** data member in a managed class or struct.

The following sample generates C2866:

```
// C2866.cpp
// compile with: /clr /c
ref class x {
public:
   const static int i;   // C2866
   const static int j = 0;   // OK

   // Delete the following line to resolve.
   void xx() { i = 9; }
};
```

The following sample generates C2866:

```
// C2866b.cpp
// compile with: /clr:oldSyntax /c

__gc class CMyClass {
public:
   const static double d;   // C2866
   const static double d2 = 9;   // OK
};
```

Compiler Error C2867

Error Message
'identifier' : is not a namespace

A **using** directive is applied to something other than a namespace.

The following sample generates C2867:

```
// C2867.cpp
// compile with: /c
namespace N {
   class X {};
}
using namespace N::X;   // C2867
```

Compiler Error C2868

Error Message
'identifier' : illegal syntax for using-declaration; expected qualified-name

A using declaration requires a qualified name.

The following sample generates C2868:

```
// C2868.cpp
class X {
public:
   int i;
};

class Y : X {
public:
   using X::i;   // OK
};

int main() {
   using X;   // C2868
}
```

Compiler Error C2869

Error Message
'name' : has already been defined to be a namespace

You cannot reuse a name already used as a namespace.

The following sample generates C2869:

```
// C2869.cpp
// compile with: /c
namespace A { int i; };

class A {};   // C2869, A is already used
```

Compiler Error C2870

Error Message
'name' : a namespace definition must appear either at file scope or immediately within another namespace definition

You defined namespace *name* incorrectly. Namespaces must be defined at file scope (outside all blocks and classes) or immediately within another namespace.

The following sample generates C2870:

```
// C2870.cpp
// compile with: /c
int main() {
   namespace A { int i; };   // C2870
}
```

Compiler Error C2871

Error Message
'name' : a namespace with this name does not exist

This error will occur when you pass an identifier that is not a namespace to a using directive.

The following sample generates C2871:

```
// C2871.cpp
// compile with: /c
using namespace d;   // C2871 d is not a namespace
using namespace System;   // OK
```

Compiler Error C2872

Error Message
'symbol' : ambiguous symbol

The compiler cannot determine which symbol you are referring to.

C2872 can occur if a header file includes a using Directive (C++), and a subsequent header file is **#include**'d and contains a type that is also in the namespace specified in the **using** directive. Specify a **using** directive only after all your header files are specified with **#include**.

For more information about C2872, see http://support.microsoft.com/default.aspx?scid=kb;en-us;316317.

The following sample generates C2872:

```
// C2872.cpp
namespace A {
   int i;
}

using namespace A;
int i;
int main() {
   ::i++;   // ok
   A::i++;   // ok
   i++;   // C2872 ::i or A::i?
}
```

Compiler Error C2873

Error Message
'symbol' : symbol cannot be used in a using-declaration

A **using** directive is missing a namespace keyword. This causes the compiler to misinterpret the code as a using declaration rather than a using directive.

Compiler Error C2874

Error Message
using-declaration causes a multiple declaration of 'symbol'

The declaration causes the same item to be defined twice.

The following sample generates C2874:

```
// C2874.cpp
namespace Z {
   int i;
}

int main() {
   int i;
   using Z::i;   // C2874, i already declared
}
```

Compiler Error C2875

Error Message
using-declaration causes a multiple declaration of 'class::identifier'

The declaration causes the same item to be defined twice.

The following sample generates C2875:

```
// C2875.cpp
struct A {
   void f(int*);
};

struct B {
   void f(double*);
};

struct AB : A, B {
   using A::f;
   using A::f;   // C2875
   using B::f;
};
```

Compiler Error C2876

Error Message
'class::symbol' : not all overloads are accessible

All overloaded forms of a function in a base class must be accessible to the derived class.

The following sample generates C2876:

```
// C2876.cpp
// compile with: /c
class A {
public:
   double a(double);
private:
   int a(int);
};

class B : public A {
   using A::a;   // C2876 one overload is private in base class
};
```

Compiler Error C2877

Error Message
'symbol' is not accessible from 'class'

All members derived from a base class must be accessible in the derived class.

The following sample generates C2877:

```
// C2877.cpp
// compile with: /c
class A {
private:
   int a;
};

class B : public A {
   using A::a;   // C2877
};
```

Compiler Error C2878

Error Message
'name' : a namespace or class of this name does not exist

You made reference to a namespace or class that is not defined.

The following sample generates C2878:

```
// C2878.cpp
// compile with: /c
namespace A {}
namespace B = C;   // C2878 namespace C doesn't exist
namespace B = A;
```

Compiler Error C2879

Error Message
'symbol' : only an existing namespace can be given an alternative name by a namespace alias definition

You cannot create a namespace alias to a symbol other than a namespace.

The following sample generates C2879:

```
// C2879.cpp
int main() {
   int i;
   namespace A = i;   // C2879 i is not a namespace
}
```

Compiler Error C2881

Error Message
'namespace1' : is already used as an alias for 'namespace2'

You cannot use the same name as an alias for two namespaces.

The following sample generates C2881:

```
// C2881.cpp
// compile with: /c
namespace A {
   int k;
}

namespace B {
   int i;
}

namespace C = A;
namespace C = B;   // C2881 C is already an alias for A
```

Compiler Error C2882

Error Message
'name' : illegal use of namespace identifier in expression

You tried to use the name of a namespace in an expression.

The following sample generates C2882:

```
// C2882.cpp
// compile with: /c
namespace A {
   int k;
}

int i = A;   // C2882, can't assign A to i
```

Compiler Error C2883

Error Message
'name' : function declaration conflicts with 'identifier' introduced by using-declaration

You tried to define a function more than once. The first definition was made from a namespace with a **using** declaration. The second was a local definition.

The following sample generates C2883:

```
// C2883.cpp
namespace A {
   void z(int);
}

int main() {
   using A::z;
   void z(int);   // C2883  z is already defined
}
```

Compiler Error C2884

Error Message
'name' : introduced by using-declaration conflicts with local function 'function'

You tried to define a function more than once. The first definition is a local definition. The second is from a namespace with a **using** declaration.

The following sample generates C2884:

```
// C2884.cpp
namespace A {
   void z(int);
}

void f() {
   void z(int);
   using A::z;   // C2884 z is already defined
}
```

Compiler Error C2885

Error Message
'class::identifier' : not a valid using-declaration at non-class scope

You used a using declaration incorrectly.

Example

This error can be generated as a result of compiler conformance work that was done for Visual C++ 2005: it is no longer valid to have a **using** declaration to a nested type; you must explicitly qualify each reference you make to the nested type, put the type in a namespace, or create a typedef. See Breaking Changes in the Visual C++ 2005 Compiler for more information.

The following sample generates C2885.

```
// C2885.cpp
namespace MyNamespace {
   class X1 {};
}

struct MyStruct {
   struct X1 {
      int i;
   };
};

int main () {
   using MyStruct::X1;   // C2885

   // OK
   using MyNamespace::X1;
   X1 myX1;

   MyStruct::X1 X12;

   typedef MyStruct::X1 abc;
   abc X13;
   X13.i = 9;
}
```

If you use the **using** keyword with a class member, C++ requires you to define that member inside another class (a derived class).

The following sample generates C2885.

```
// C2885_b.cpp
// compile with: /c
class A {
public:
   int i;
};

void z() {
   using A::i;   // C2885 not in a class
}

class B : public A {
public:
   using A::i;
};
```

Compiler Error C2886

Error Message
'class::identifier' : symbol cannot be used in a member using-declaration

A **using** declaration uses a symbol, such as a namespace name. A **using** declaration is for declaring base class members.

The following sample generates C2886:

```
// C2886.cpp
// compile with: /c
namespace Z {
    int i;
}

class B {
protected:
    int i;
};

class D : public B {
    // Error: Z is a namespace
    using Z::i;   // C2886

    // OK: B is a base class
    using B::i;
};
```

Compiler Error C2888

Error Message
'identifier' : symbol cannot be defined within namespace 'namespace'

A symbol belonging to namespace A must be defined in a namespace that encloses A.

The following sample generates C2888:

```
// C2888.cpp
// compile with: /c
namespace M {
   namespace N {
      void f1();
      void f2();
   }

   void N::f1() {}   // OK: namspace M encloses N
}

namespace O {
   void M::N::f2() {}   // C2888 namespace O does not enclose M
}
```

Compiler Error C2890

Error Message

'class' : a ref class can only have one non-interface base class

A reference class can only have one base class.

The following sample generates C2890:

```
// C2890.cpp
// compile with: /clr /c
ref class A {};
ref class B {};
ref class C : public A, public B {};   // C2890
ref class D : public A {};   // OK
```

Managed Extensions for C++

The following sample generates C2890:

```
// C2890b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
__gc class A {};
__gc class B {};

__gc class C : public A, public B {};   // C2890
__gc class D : public A {};   // OK
```

Compiler Error C2891

Error Message
'parameter' : cannot take the address of a template parameter

Instead of using code like this:

```
template <int i> int* f() { return &i; }
```

use this:

```
template <int i> int* f() { return i; }
```

Compiler Error C2892

Error Message
local class shall not have member templates

Templated member functions are not valid in a class that is defined in a function.

The following sample generates C2892:

```
// C2892.cpp
int main() {
   struct local {
      template<class T>   // C2892
      void f() {}
   };
}
```

Compiler Error C2893

Error Message
Failed to specialize function template 'template name'

The compiler failed to specialize a function template. There can be many causes for this error.

In general, the way to resolve a C2893 error is to review the function's signature and make sure you can instantiate every type.

Example

C2893 occurs because `f`'s template parameter `T` is deduced to be `std::map<int,int>`, but `std::map<int,int>` has no member `data_type` (`T::data_type` can not be instantiated with `T = std::map<int,int>`.). The following sample generates C2893.

```
// C2893.cpp
// compile with: /c /EHsc
#include<map>
using namespace std;
class MyClass {};

template<class T>
inline typename T::data_type
// try the following line instead
// inline typename  T::mapped_type
f(T const& p1, MyClass const& p2);

template<class T>
void bar(T const& p1) {
    MyClass r;
    f(p1,r);   // C2893
}

int main() {
   map<int,int> m;
   bar(m);
}
```

Compiler Error C2894

Error Message
templates cannot be declared to have 'C' linkage

This error can be caused by a template defined inside an **extern** "C" block.

The following sample generates C2894:

```
// C2894.cpp
extern "C" {
   template<class T> class stack {};   // C2894 fail

   template<class T> void f(const T &aT) {}   // C2894
}
```

The following sample generates C2894:

```
// C2894b.cpp
// compile with: /c
extern "C" template<class T> void f(const T &aT) {}   // C2894

template<class T> void f2(const T &aT) {}   // OK
```

Compiler Error C2896

Error Message
'function1' : cannot use function template 'function2' as argument

A function template cannot be an argument to another function template.

The following sample generates C2896:

```
// C2896.cpp
template<class T1, class T2> void f1(void(*)(T1, T2));
template<class T1, class T2> void f2(T1, T2);

int main() {
   f1(f2);   // C2896
}
```

C2896 can also occur when you use generics:

```
// C2896b.cpp
// compile with: /clr
generic<class T1> void gf1(T1){}
generic<class T1> void gf2(T1){}

int main() {
   gf1(gf2);   // C2896
   gf1(1);   // OK
}
```

Compiler Error C2897

Error Message
a destructor/finalizer cannot be a function template

Destructors or finalizers cannot be overloaded, so declaring a destructor as a template (which would define a set of destructors) is not allowed.

The following sample generates C2897:

Example

The following sample generates C2897.

```
// C2897.cpp
// compile with: /c
class X {
public:
   template<typename T> ~X() {}   // C2897
};
```

The following sample generates C2897.

```
// C2897_b.cpp
// compile with: /c /clr
ref struct R2 {
protected:
   template<typename T> !R2(){}   // C2897 error
};
```

Compiler Error C2898

Error Message

'declaration' : member function templates cannot be virtual

The following sample generates C2898:

```
// C2898.cpp
// compile with: /c
class X {
public:
   template<typename T> virtual void f(T t) {}   // C2898
};
```

Compiler Error C2899

Error Message
typename cannot be used outside a template declaration

The typename keyword can be used only in a template definition or declaration. In a template declaration, it can be used in two ways:

```
// C2899.cpp
// compile with: /c
template<typename T>
class X {};

// Another way
template<class T>
struct XX {
   typename T::A a;   // T::A is a type
};
```

The following sample generates C2899:

```
// C2899b.cpp
// compile with: /c
struct Y {
   typedef int B;
   typename Y::B b;   // C2899
};
```

Compiler Errors C2900 Through C3499

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, type the error number in the **Look for** box in the Index, or click a link below.

Compiler Error C2902

Compiler Error C2903

Compiler Error C2904

Compiler Error C2906

Compiler Error C2907

Compiler Error C2908

Compiler Error C2909

Compiler Error C2910

Compiler Error C2911

Compiler Error C2912

Compiler Error C2913

Compiler Error C2914

Compiler Error C2917

Compiler Error C2918

Compiler Error C2919

Compiler Error C2920

Compiler Error C2921

Compiler Error C2923

Compiler Error C2926

Compiler Error C2927

Compiler Error C2928

Compiler Error C2929

Compiler Error C2930

Compiler Error C2931

Compiler Error C2932

Compiler Error C2933

Compiler Error C2934

Compiler Error C2935

Compiler Error C2936

Compiler Error C2937

Compiler Error C2939

Compiler Error C2940

Compiler Error C2941

Compiler Error C2942

Compiler Error C2943

Compiler Error C2944
Compiler Error C2945
Compiler Error C2946
Compiler Error C2947
Compiler Error C2948
Compiler Error C2949
Compiler Error C2950
Compiler Error C2951
Compiler Error C2952
Compiler Error C2953
Compiler Error C2954
Compiler Error C2955
Compiler Error C2957
Compiler Error C2958
Compiler Error C2959
Compiler Error C2962
Compiler Error C2963
Compiler Error C2964
Compiler Error C2969
Compiler Error C2970
Compiler Error C2971
Compiler Error C2973
Compiler Error C2974
Compiler Error C2975
Compiler Error C2976
Compiler Error C2977
Compiler Error C2978
Compiler Error C2979
Compiler Error C2988
Compiler Error C2989
Compiler Error C2990
Compiler Error C2991
Compiler Error C2992
Compiler Error C2993
Compiler Error C2994
Compiler Error C2995
Compiler Error C2996
Compiler Error C2998
Compiler Error C2999
Compiler Error C3001

Compiler Error C3002
Compiler Error C3003
Compiler Error C3004
Compiler Error C3005
Compiler Error C3006
Compiler Error C3007
Compiler Error C3008
Compiler Error C3009
Compiler Error C3010
Compiler Error C3011
Compiler Error C3012
Compiler Error C3013
Compiler Error C3014
Compiler Error C3015
Compiler Error C3016
Compiler Error C3017
Compiler Error C3018
Compiler Error C3019
Compiler Error C3020
Compiler Error C3021
Compiler Error C3022
Compiler Error C3023
Compiler Error C3024
Compiler Error C3025
Compiler Error C3026
Compiler Error C3027
Compiler Error C3028
Compiler Error C3029
Compiler Error C3030
Compiler Error C3031
Compiler Error C3032
Compiler Error C3033
Compiler Error C3034
Compiler Error C3035
Compiler Error C3036
Compiler Error C3037
Compiler Error C3038
Compiler Error C3039
Compiler Error C3040
Compiler Error C3041

Compiler Error C3042
Compiler Error C3043
Compiler Error C3044
Compiler Error C3045
Compiler Error C3046
Compiler Error C3047
Compiler Error C3048
Compiler Error C3049
Compiler Error C3050
Compiler Error C3052
Compiler Error C3053
Compiler Error C3054
Compiler Error C3055
Compiler Error C3056
Compiler Error C3057
Compiler Error C3058
Compiler Error C3059
Compiler Error C3060
Compiler Error C3062
Compiler Error C3063
Compiler Error C3065
Compiler Error C3066
Compiler Error C3068
Compiler Error C3069
Compiler Error C3070
Compiler Error C3071
Compiler Error C3072
Compiler Error C3073
Compiler Error C3075
Compiler Error C3076
Compiler Error C3077
Compiler Error C3080
Compiler Error C3082
Compiler Error C3083
Compiler Error C3084
Compiler Error C3085
Compiler Error C3087
Compiler Error C3094
Compiler Error C3095
Compiler Error C3096

Compiler Error C3097
Compiler Error C3099
Compiler Error C3100
Compiler Error C3101
Compiler Error C3103
Compiler Error C3104
Compiler Error C3106
Compiler Error C3110
Compiler Error C3113
Compiler Error C3114
Compiler Error C3115
Compiler Error C3116
Compiler Error C3117
Compiler Error C3118
Compiler Error C3120
Compiler Error C3121
Compiler Error C3126
Compiler Error C3130
Compiler Error C3131
Compiler Error C3132
Compiler Error C3133
Compiler Error C3134
Compiler Error C3135
Compiler Error C3136
Compiler Error C3137
Compiler Error C3138
Compiler Error C3139
Compiler Error C3140
Compiler Error C3141
Compiler Error C3142
Compiler Error C3145
Compiler Error C3146
Compiler Error C3147
Compiler Error C3148
Compiler Error C3149
Compiler Error C3150
Compiler Error C3151
Compiler Error C3152
Compiler Error C3153
Compiler Error C3154

Compiler Error C3155
Compiler Error C3156
Compiler Error C3157
Compiler Error C3159
Compiler Error C3160
Compiler Error C3161
Compiler Error C3162
Compiler Error C3163
Compiler Error C3164
Compiler Error C3166
Compiler Error C3167
Compiler Error C3168
Compiler Error C3170
Compiler Error C3171
Compiler Error C3172
Compiler Error C3173
Compiler Error C3174
Compiler Error C3175
Compiler Error C3176
Compiler Error C3179
Compiler Error C3180
Compiler Error C3181
Compiler Error C3182
Compiler Error C3183
Compiler Error C3185
Compiler Error C3186
Compiler Error C3188
Compiler Error C3189
Compiler Error C3190
Compiler Error C3191
Compiler Error C3192
Compiler Error C3194
Compiler Error C3195
Compiler Error C3196
Compiler Error C3197
Compiler Error C3198
Compiler Error C3199
Compiler Error C3200
Compiler Error C3201
Compiler Error C3202

Compiler Error C3203
Compiler Error C3204
Compiler Error C3205
Compiler Error C3206
Compiler Error C3207
Compiler Error C3208
Compiler Error C3209
Compiler Error C3210
Compiler Error C3211
Compiler Error C3212
Compiler Error C3213
Compiler Error C3214
Compiler Error C3215
Compiler Error C3216
Compiler Error C3217
Compiler Error C3218
Compiler Error C3219
Compiler Error C3222
Compiler Error C3223
Compiler Error C3224
Compiler Error C3225
Compiler Error C3226
Compiler Error C3227
Compiler Error C3228
Compiler Error C3229
Compiler Error C3230
Compiler Error C3231
Compiler Error C3232
Compiler Error C3233
Compiler Error C3234
Compiler Error C3235
Compiler Error C3236
Compiler Error C3237
Compiler Error C3238
Compiler Error C3239
Compiler Error C3240
Compiler Error C3241
Compiler Error C3242
Compiler Error C3243
Compiler Error C3244

Compiler Error C3246
Compiler Error C3247
Compiler Error C3248
Compiler Error C3251
Compiler Error C3252
Compiler Error C3253
Compiler Error C3254
Compiler Error C3255
Compiler Error C3257
Compiler Error C3258
Compiler Error C3261
Compiler Error C3262
Compiler Error C3264
Compiler Error C3265
Compiler Error C3266
Compiler Error C3267
Compiler Error C3268
Compiler Error C3269
Compiler Error C3270
Compiler Error C3271
Compiler Error C3272
Compiler Error C3273
Compiler Error C3274
Compiler Error C3275
Compiler Error C3276
Compiler Error C3277
Compiler Error C3278
Compiler Error C3279
Compiler Error C3280
Compiler Error C3281
Compiler Error C3282
Compiler Error C3283
Compiler Error C3284
Compiler Error C3285
Compiler Error C3286
Compiler Error C3287
Compiler Error C3288
Compiler Error C3289
Compiler Error C3290
Compiler Error C3291

Compiler Error C3292
Compiler Error C3293
Compiler Error C3294
Compiler Error C3295
Compiler Error C3296
Compiler Error C3303
Compiler Error C3304
Compiler Error C3305
Compiler Error C3309
Compiler Error C3320
Compiler Error C3322
Compiler Error C3333
Compiler Error C3334
Compiler Error C3340
Compiler Error C3342
Compiler Error C3347
Compiler Error C3350
Compiler Error C3351
Compiler Error C3352
Compiler Error C3353
Compiler Error C3354
Compiler Error C3356
Compiler Error C3358
Compiler Error C3360
Compiler Error C3363
Compiler Error C3364
Compiler Error C3365
Compiler Error C3366
Compiler Error C3367
Compiler Error C3368
Compiler Error C3369
Compiler Error C3370
Compiler Error C3371
Compiler Error C3372
Compiler Error C3373
Compiler Error C3374
Compiler Error C3375
Compiler Error C3379
Compiler Error C3380
Compiler Error C3381

Compiler Error C3382
Compiler Error C3383
Compiler Error C3384
Compiler Error C3385
Compiler Error C3386
Compiler Error C3387
Compiler Error C3388
Compiler Error C3389
Compiler Error C3390
Compiler Error C3391
Compiler Error C3392
Compiler Error C3393
Compiler Error C3394
Compiler Error C3395
Compiler Error C3396
Compiler Error C3397
Compiler Error C3398
Compiler Error C3399
Compiler Error C3408
Compiler Error C3409
Compiler Error C3412
Compiler Error C3413
Compiler Error C3414
Compiler Error C3415
Compiler Error C3416
Compiler Error C3417
Compiler Error C3418
Compiler Error C3420
Compiler Error C3421
Compiler Error C3450
Compiler Error C3451
Compiler Error C3452
Compiler Error C3453
Compiler Error C3454
Compiler Error C3455
Compiler Error C3456
Compiler Error C3457
Compiler Error C3458
Compiler Error C3459
Compiler Error C3460

Compiler Error C3461

Compiler Error C3462

Compiler Error C3463

Compiler Error C3464

Compiler Error C3465

Compiler Error C3466

Compiler Error C3467

Compiler Error C3468

Compiler Error C3469

Compiler Error C3470

Compiler Error C2902

Error Message
'token' : unexpected token following 'template', identifier expected

The token following the keyword **template** was not an identifier.

The following sample generates C2902:

```
// C2902.cpp
// compile with: /c
namespace N {
   template<class T> class X {};
   class Y {};
}
void g() {
   N::template + 1;   // C2902
}

void f() {
   N::template X<int> x1;   // OK
}
```

C2902 can also occur when using generics:

```
// C2902b.cpp
// compile with: /clr /c
namespace N {
   generic<class T> ref class GC {};
}

void f() {
   N::generic + 1;   // C2902
   N::generic GC<int>^ x;
}
```

Compiler Error C2903

Error Message
'identifier' : symbol is neither a class template nor a function template

Code attempts explicit instantiation of something that is not a template.

The following sample generates C2903:

```
// C2903.cpp
// compile with: /c
namespace N {
   template<class T> class X {};
   class Y {};
}
void g() {
   N::template Y y;   // C2903
   N::X<N::Y> y;   // OK
}
```

C2903 can also occur when using generics:

```
// C2903b.cpp
// compile with: /clr /c
namespace N {
   class Y {};
   generic<class T> ref class Z {};
}

void f() {
   N::generic Y y;   // C2903
   N:: generic Z<int>^ z;   // OK
}
```

Compiler Error C2904

Error Message
'identifier' : name already used for a template in the current scope

Check the code for duplicate names.

The following sample generates C2904:

```
// C2904.cpp
// compile with: /c
void X();  // X is declared as a function
template<class T> class X{};  // C2904
```

Compiler Error C2906

Error Message
'specialization' : explicit specialization requires 'template <>'

You must use the new syntax for explicit specialization of templates.

The following sample generates C2906:

```
// C2906.cpp
// compile with: /c
template<class T> class X{};   // primary template
class X<int> { }   // C2906
template<> class X<int> { };   // new syntax
```

Compiler Error C2907

Error Message

register argument 'arg' does not specify a valid register number

A function argument was not specified correctly.

Example

The following sample generates C2907.

```
// C2907.cpp
// processor: IPF
#include <intrin.h>
#define __CV_IPF_IntR1 1025
int main() {
   __getReg(0);   // C2907
   __getReg(__CV_IPF_IntR1);   // OK
}
```

Compiler Error C2908

Error Message
explicit specialization; 'template' has already been instantiated

A specialization of the primary template occurs before the explicit specialization.

The following sample generates C2908:

```
// C2908.cpp
// compile with: /c
template<class T> class X {};

void f() {
X<int> x;   //specialization and instantiation
            //of X<int>
}

template<> class X<int> {}  // C2908, explicit specialization
```

Compiler Error C2909

Error Message
'identifier': explicit instantiation of function template requires return type

An explicit instantiation of a function template requires explicit specification of its return type. Implicit return type specification does not work.

The following sample generates C2909:

```
// C2909.cpp
// compile with: /c
template<class T> int f(T);
template f<int>(int);         // C2909
template int f<int>(int);   // OK
```

Compiler Error C2910

Error Message
'function' : cannot be explicitly specialized

The compiler detected an attempt to explicitly specialize a function twice.

The following sample generates C2910:

```
// C2910.cpp
// compile with: /c
template <class T>
struct S;

template <> struct S<int> { void f() {} };
template <> void S<int>::f() {}   // C2910 delete this specialization
```

C2910 can also be generated if you try to explicitly specialize a non-template member. That is, you can only explicitly specialize a function template.

The following sample generates C2910:

```
// C2910b.cpp
// compile with: /c
template <class T> struct A {
   A(T* p);
};

template <> struct A<void> {
   A(void* p);
};

template <class T>
inline A<T>::A(T* p) {}

template <> A<void>::A(void* p){}   // C2910
// try the following line instead
// A<void>::A(void* p){}
```

This error will also be generated as a result of compiler conformance work that was done in Visual Studio .NET 2003:.

For code will be valid in the Visual Studio .NET 2003 and Visual Studio .NET versions of Visual C++, remove `template <>`.

See Summary of Compile-Time Breaking Changes for more information.

The following sample generates C2910:

```
// C2910c.cpp
// compile with: /c
template <class T> class A {
   void f();
};

template <> class A<int> {
   void f();
};

template <> void A<int>::f() {}   // C2910
// try the following line instead
// void A<int>::f(){}   // OK
```

Compiler Error C2911

Error Message
'member' : cannot be declared or defined in the current scope

Inside a namespace, class, or function, you can only define a member of the same namespace, class, or function or a member that is enclosed by the same namespace, class, or function.

The following sample generates C2911:

```
// C2911.cpp
struct A;

namespace M {
   struct D;
}

namespace N {
   struct C;

   namespace O {
      struct B;
   }

   // in N
   struct ::A {};   // C2911  A is member of global NS
   struct O::B{};   // OK B is in O, O is inside of N
   struct C {};     // OK C is member of N
   struct M::D {};  // C2911 D is member of M, M not enclosed by N
}
```

Compiler Error C2912

Error Message
explicit specialization; 'declaration' is not a specialization of a function template

You cannot specialize a non-template function.

The following sample generates C2912:

```
// C2912.cpp
// compile with: /c
void f(char);
template<> void f(char);   // C2912
template<class T> void f(T);   // OK
```

This error will also be generated as a result of compiler conformance work that was done in Visual Studio .NET 2003: for every explicit specialization, you must choose the parameters of the explicit specialization, such that, they match the parameters of the primary template.

See Summary of Compile-Time Breaking Changes for more information.

```
// C2912b.cpp
class CF {
public:
   template <class A> CF(const A& a) {}   // primary template

   // attempted explicit specialization
   template <> CF(const char* p) {}   // C2912

   // try the following line instead
   // template <> CF(const char& p) {}
};
```

Compiler Error C2913

Error Message
explicit specialization; 'declaration' is not a specialization of a class template

You cannot specialize a non-template class.

The following sample generates C2913:

```
// C2913.cpp
// compile with: /c
class X{};
template <class T> class Y{};

template<> class X<int> {};   // C2913
template<> class Y<int> {};
```

Compiler Error C2914

Error Message
'identifier' : cannot deduce type argument as function argument is ambiguous

The compiler cannot determine which overloaded functions to use for a generic or template argument.

The following sample generates C2914:

```
// C2914.cpp
// compile with: /c
void f(int);
void f(double);
template<class T> void g(void (*) (T));
void h() { g(f); }   // C2914
// try the following line instead
// void h() { g<int>(f); }
```

C2914 can also occur when using generics. The following sample generates C2914:

```
// C2914b.cpp
// compile with: /clr /c
void f(int);
void f(double);

template<class T>
void gf(void (*) (T));

void h() { gf(f);}   // C2914
// try the following line instead
void h() { gf<int>(f); }
```

Compiler Error C2917

Error Message
'name' : invalid template-parameter

A template parameter list contains an identifier that was not a template parameter.

Example
The following sample generates C2917.

```
// C2917.cpp
// compile with: /c
template<class T> class Vector {
   void sort();
};

template<class T*> void Vector<T>::sort() {}   // C2917
// try the following line instead
// template<class T> void Vector<T>::sort() {}
```

Compiler Error C2918

Error Message
'name' : illegal use of local type in type instantiation

You cannot instantiate a function based on a function template based on a local type. Types used to instantiate must have external linkage.

The following sample generates C2918:

```
// C2918.cpp
// compile with: /Za /c
template<class T>
void f(T t) {};

template<class T>
void g(T t) {};
struct Y {};

void g() {
   struct X {};

   X x;
   f(x);   // C2918

   Y y;
   g(y);
}
```

Compiler Error C2919

Error Message
illegal use of anonymous local type in type instantiation

You cannot instantiate a function from a function template or generic function based on a local type. Types used to instantiate must have external linkage.

The following sample generates C2919:

```
// C2919.cpp
// compile with: /Za
template<class T>
void f(T t) {};

template<class T>
void f2(T t) {};
struct X2 {} x2;

int main() {
   struct {} x;
   f(x);   // C2919
   f2(x2);
}
```

Compiler Error C2920

Error Message
redefinition : 'class' : type class has already been declared as 'identifier'

A generic or template class has multiple declarations, which are not equivalent.

The following sample generates C2920:

```
// C2920.cpp
// compile with: /c
typedef int TC1;
template <class T>
struct TC1 {};   // C2920
struct TC2 {};   // OK
```

C2920 can also occur when using generics:

```
// C2920b.cpp
// compile with: /clr /c
typedef int GC1;
generic <class T>
ref struct GC1 {};   // C2920
ref struct GC2 {};   // OK
```

Compiler Error C2921

Error Message
redefinition : 'class' : class type is being redeclared as 'identifier'

A generic or template class has multiple declarations that are not equivalent.

The following sample generates C2921:

```
// C2921.cpp
// compile with: /c
template <class T> struct TC2 {};
typedef int TC2;   // C2921
// try the following line instead
// typedef struct TC2<int> x;   // OK
```

C2921 can also occur when using generics.

```
// C2921b.cpp
// compile with: /clr /c
generic <class T> ref struct GC2 {};
typedef int GC2;   // C2921
// try the following line instead
// typedef ref struct GC2<int> x;
```

Compiler Error C2923

Error Message
'type' : 'identifier' is not a valid template type argument for parameter 'param'

The argument list is missing a type needed to instantiate the template or generic. Check the template or generic declaration.

The following sample generates C2923:

```
// C2923.cpp
template <class T> struct TC {};
int x;
int main() {
   TC<x>* tc2;   // C2923
   TC<int>* tc2;   // OK
}
```

C2923 can also occur when using generics:

```
// C2923b.cpp
// compile with: /clr /c
generic <class T> ref struct GC {};

int x;

int main() {
   GC<x>^ gc2;   // C2923
   GC<int>^ gc2;   // OK
}
```

Compiler Error C2926

Error Message
'type' : local types or unnamed types cannot be used as type arguments

You cannot instantiate a generic or template class based on a local or unnamed type. Types used to instantiate must have external linkage.

The following sample generates C2926:

```
// C2926.cpp
// compile with: /Za
template<class T>
class X {};
struct Z{};

int main() {
   struct Y{};
    X<Y> x;   // C2926
    X<Z> x2;   // OK
}
```

Compiler Error C2927

Error Message
'function' : a function template must be called with at least one argument

You cannot call a template function without arguments. The type of the template arguments determines what version of the function to generate.

Compiler Error C2928

Error Message

explicit instantiation; 'identifier' is not a function or static data member of template-class 'class'

You cannot explicitly instantiate a member of *class* that is not a function or **static** variable.

Compiler Error C2929

Error Message
'identifier' : explicit instantiation; cannot explicitly force and suppress instantiation of template-class member

You cannot explicitly instantiate an identifier while preventing it from being instantiated.

The following sample generates C2929:

```
// C2929.cpp
// compile with: /c
template<typename T>
class A {
public:
   A() {}
};

template A<int>::A();

extern template A<int>::A();   // C2929
```

Compiler Error C2930

Error Message
'class' : type-class-id redefined as an enumerator of 'enum identifier'

You cannot use a generic or template class as a member of an enumeration.

This error can be caused if braces are improperly matched.

The following sample generates C2930:

```
// C2930.cpp
// compile with: /c
template<class T>
class x{};
enum SomeEnum { x };   // C2930

class y{};
enum SomeEnum { y };
```

C2930 can also occur when using generics:

```
// C2930c.cpp
// compile with: /clr /c
generic<class T>
ref struct GC {};
enum SomeEnum { GC };   // C2930

ref struct GC2 {};
enum SomeEnum2 { GC2 };
```

Compiler Error C2931

Error Message
'class' : type-class-id redefined as a member function of 'identifier'

You cannot use a generic or template class as a member function of another class.

This error can be caused if braces are improperly matched.

The following sample generates C2931:

```
// C2931.cpp
// compile with: /c
template<class T>
struct TC { };
struct MyStruct {
   void TC<int>();   // C2931
};

struct TC2 { };
struct MyStruct2 {
   void TC2();
};
```

C2931 can also occur when using generics:

```
// C2931b.cpp
// compile with: /clr /c
generic<class T> ref struct GC {};
struct MyStruct {
   void GC<int>();   // C2931
   void GC2();   // OK
};
```

Compiler Error C2932

Error Message
'class' : type-class-id redefined as a data member of 'identifier'

You cannot use a generic or template class as a data member.

The following sample generates C2932:

```
// C2932.cpp
// compile with: /c
template<class T>
struct TC {};

struct MyStruct {
   int TC<int>;   // C2932
   int TC;   // OK
};
```

C2932 can also occur when using generics:

```
// C2932b.cpp
// compile with: /clr /c
generic<class T>
ref struct GC {};

struct MyStruct {
   int GC<int>;   // C2932
   int GC;   // OK
};
```

Compiler Error C2933

Error Message
'class' : type-class-id redefined as a typedef member of 'identifier'

You cannot use a generic or template class as a **typedef** member.

The following sample generates C2933:

```
// C2933.cpp
// compile with: /c
template<class T> struct TC { };
struct MyStruct {
   typedef int TC<int>;   // C2933
};

struct TC2 { };
struct MyStruct2 {
   typedef int TC2;
};
```

C2933 can also occur when using generics:

```
// C2933b.cpp
// compile with: /clr /c
generic<class T> ref struct GC { };
struct MyStruct {
   typedef int GC<int>;   // C2933
};

ref struct GC2 { };
struct MyStruct2 {
   typedef int GC2;
};
```

Compiler Error C2934

Error Message

'class' : type-class-id redefined as a nested 'item' of 'identifier'

You cannot use a generic or template class as a nested item.

Compiler Error C2935

Error Message
'class' : type-class-id redefined as a global function

You cannot use a generic or template class as a global function.

This error can be caused if braces are improperly matched.

The following sample generates C2935:

```
// C2935.cpp
// compile with: /c
template<class T>
struct TC {};
void TC<int>() {}   // C2935

// OK
struct TC2 {};
void TC2() {}
```

C2935 can also occur when using generics:

```
// C2935b.cpp
// compile with: /clr /c
generic<class T>
ref struct GC { };
void GC<int>() {}   // C2935
void GC() {}   // OK
```

Compiler Error C2936

Error Message
'class' : type-class-id redefined as a global data variable

You cannot use a generic or template class as a global data variable.

This error can be caused if braces are improperly matched.

The following sample generates C2936:

```
// C2936.cpp
// compile with: /c
template<class T> struct TC { };
int TC<int>;   // C2936

// OK
struct TC2 { };
int TC2;
```

C2936 can also occur when using generics:

```
// C2936b.cpp
// compile with: /clr /c
generic<class T>
ref struct GC {};
int GC<int>;   // C2936

// OK
ref struct GC2 {};
int GC2;
```

Compiler Error C2937

Error Message
'class' : type-class-id redefined as a global typedef

You cannot use a generic or template class as a global **typedef**.

The following sample generates C2937:

```
// C2937.cpp
// compile with: /c
template<class T>
struct TC { };
typedef int TC<int>;   // C2937
typedef TC<int> c;   // OK
```

C2937 can also occur when using generics:

```
// C2937b.cpp
// compile with: /clr
generic<class T>
ref struct GC { };
typedef int GC<int>;   // C2937
typedef GC<int> xx;   // OK
```

Compiler Error C2939

Error Message
'class' : type-class-id redefined as a local data variable

You cannot use a generic or template class as a local data variable.

This error can be caused if braces are improperly matched.

The following sample generates C2939:

```
// C2939.cpp
template<class T>
struct TC { };
int main() {
   int TC<int>;   // C2939
   int TC;   // OK
}
```

C2939 can also occur when using generics:

```
// C2939b.cpp
// compile with: /clr
generic<class T>
ref struct GC { };

int main() {
   int GC<int>;   // C2939
   int GC;   // OK
}
```

Compiler Error C2940

Error Message
'class' : type-class-id redefined as a local typedef

You cannot use a generic or template class as a local **typedef**.

The following sample generates C2940:

```
// C2940.cpp
template<class T>
struct TC {};
int main() {
   typedef int TC<int>;   // C2940
   typedef int TC;   // OK
}
```

C2940 can also occur when using generics:

```
// C2940b.cpp
// compile with: /clr
generic<class T>
ref struct GC { };

int main() {
   typedef int GC<int>;   // C2940
   typedef int GC;
}
```

Compiler Error C2941

Error Message

'class' : type-class-id redefined as a local 'item'

You cannot use a generic or template class as an item.

Compiler Error C2942

Error Message
'class' : type-class-id redefined as a formal argument of a function

You cannot use a generic or template class as a formal argument. You cannot pass an argument directly to the constructor of a generic or template class.

The following sample generates C2942:

```
// C2942.cpp
// compile with: /c
template<class T>
struct TC {};
void f(int TC<int>) {}   // C2942

// OK
struct TC2 {};
void f(TC2 i) {}
```

C2942 can also occur when using generics:

```
// C2942b.cpp
// compile with: /clr /c
generic<class T>
ref struct GC {};
void f(int GC<int>) {}   // C2942
ref struct GC2 { };
void f(int GC2) {}
```

Compiler Error C2943

Error Message
'class' : type-class-id redefined as a type argument of a template

You cannot use a generic or template class, instead of a symbol, as a generic or template type argument.

The following sample generates C2943:

```
// C2943.cpp
// compile with: /c
template<class T>
class List {};

template<class List<int> > class MyList;   // C2943
template<class T >  class MyList;
```

Compiler Error C2944

Error Message
'class' : type-class-id redefined as a value argument of a template

You cannot use a generic or template class, instead of a symbol, as a template value argument.

The following sample generates C2944:

```
// C2944.cpp
// compile with: /c
template<class T>
class TC { };

template <int TC<int> > struct X1 { };   // C2944

template <class T > struct X2 {};
```

C2944 can also occur when using generics:

```
// C2944b.cpp
// compile with: /clr /c
generic<class T>
ref class GC {};

template <int GC<int> > struct X2 { };   // C2944
template <class T> struct X3 {};   // OK
```

Compiler Error C2945

Error Message
explicit instantiation does not refer to a template-class specialization

You cannot explicitly instantiate something that is not templated.

Compiler Error C2946

Error Message
explicit instantiation; 'class' is not a template-class specialization

You cannot explicitly instantiate a nontemplated class.

Example
The following sample generates C2946.

```
// C2946.cpp
class C {};
template C;  // C2946
int main() {}
```

Compiler Error C2947

Error Message
expecting '>' to terminate construct, found 'syntax'

A generic or template argument list may not have been terminated correctly.

C2947 can also be generated by syntax errors.

The following sample generates C2947:

```
// C2947.cpp
// compile with: /c
template <typename T>=   // C2947
// try the following line instead
// template <typename T>
struct A {};
```

Compiler Error C2948

Error Message
explicit instantiation; storage class specifier 'specifier' not permitted on specialization

You cannot use storage-class specifiers (such as **extern**) in a specialization of a template class that was previously explicitly instantiated.

Compiler Error C2949

Error Message
'symbol' : explicit instantiation; cannot use 'auto' and 'extern' on the same template-class specialization

A template class is instantiated once using the **extern** storage-class specifier and once without. If you instantiate a template class more than once, you must use the same storage-class specifier for each declaration.

Example
The following sample generates C2949.

```
// C2949.cpp
template <class T> class X {};
extern template X<int>;
template X<int>;   // C2949
int main() {}
```

Compiler Error C2950

Error Message
'type' : cannot explicitly instantiate an explicit specialization

The compiler found an invalid explicit instantiation.

The following sample generates C2950:

```
// C2950.cpp
// compile with: /c
template <class T>
struct s;

template <>
struct s<int> {};

template struct s<int>;   // C2950 delete this instantiation
```

Compiler Error C2951

Error Message
type declarations are only permitted at global, namespace, or class scope

You cannot declare a generic or template class outside global or namespace scope. If you make your generic or template declarations in an include file, make sure the include file is at global scope.

The following sample generates C2951:

```
// C2951.cpp
template <class T>
class A {};

int main() {
   template <class T>   // C2951
   class B {};
}
```

C2951 can also occur when using generics:

```
// C2951b.cpp
// compile with: /clr /c

// OK
generic <class T>
ref class GC { };

int main() {
   generic <class T> ref class GC2 {};   // C2951
}
```

Compiler Error C2952

Error Message
'declaration' : type declaration missing template parameter list

A template declaration was ill formed.

The following sample generates C2952:

```
// C2952.cpp
// compile with: /c
template <class T>
struct S {
   template <class T1>
   struct S1 {
      void f();
   };
};

template <class T> void S<T>::S1<T>::f() {}   // C2952

// OK
template <class T>
template <class T1>
void S<T>::S1<T1>::f() {}
```

C2952 can also occur when using generics:

```
// C2952b.cpp
// compile with: /clr /c
generic <class T>
ref struct GC {
   generic <class T1>
   ref struct GC1 {
      void f();
   };
};

generic <class T> void GC<T>::GC1<T>::f() {}   // C2952

// OK
generic <class T>
generic <class T1>
void GC<T>::GC1<T1>::f() {}
```

Compiler Error C2953

Error Message
'identifier' : class template has already been defined

Check the source file and include files for other definitions.

The following sample generates C2953:

```
// C2953.cpp
// compile with: /c
template <class T>  class A {};
template <class T>  class A {};   // C2953
template <class T>  class B {};   // OK
```

Compiler Error C2954

Error Message
instruction word argument not in range

An invalid parameter was passed to an **__lfetch** intrinsic. For more inforamtion, see __lfetch, __lfetch_excl, __lfetchfault, __lfetchfault_excl.

Example
The following sample generates C2954.

```
// C2954.cpp
// compile with: /c
// processor: IPF
#include <intrin.h>
#pragma intrinsic(__lfetch)
int f(int *p, int x) {
   __lfetch(5, p);   // C2954
   __lfetch(2, p);   // OK
   return *p;
}
```

Compiler Error C2955

Error Message
'identifier' : use of class type requires type argument list

You cannot use a class template or class generic as an identifier without a template or generic argument list.

For more information, see Class Templates.

The following sample generates C2955:

```
// C2955.cpp
// compile with: /c
template<class T>
class X {};

X x;   // C2955
X<int> x2;   // OK
```

C2955 can also occur when attempting an out-of-line definition for a function declared in a class template:

```
// C2955_b.cpp
// compile with: /c
template <class T>
class CT {
public:
   void CTFunc();
   void CTFunc2();
};

void CT::CTFunc() {}   // C2955

// OK
template <class T>
void CT<T>::CTFunc2() {}
```

C2955 can also occur when using generics:

```
// C2955_c.cpp
// compile with: /clr
generic <class T>
ref struct GC {
   T t;
};

int main() {
   GC^ g;   // C2955
   GC <int>^ g;
}
```

Compiler Error C2957

Error Message
'delim' : invalid left delimiter : expected '<'

A generic class was ill formed.

The following sample generates C2957:

```
// C2957.cpp
// compile with: /clr /LD
generic << class T>   // C2957
// try the following line instead
// generic < class T>
gc class C {};
```

Compiler Error C2958

Error Message
the left delimiter found at 'location' was not matched correctly

A delimiter is not properly matched.

This error can be caused by mismatched parenthesis in a function template declaration.

Compiler Error C2959

Error Message

a generic class or function may not be a member of a template

For more information, see Managed Templates and Generics (Visual C++).

Example

The following sample generates C2959.

```
// C2959.cpp
// compile with: /clr /c
template <class T> ref struct S {
   generic <class U> ref struct GR1;   // C2959
};
```

Compiler Error C2962

Error Message

syntax error : 'token' : expected template-class member function definition to end with '}'

The token caused a syntax error in a template declaration.

This error can be caused by mismatched delimiters

Compiler Error C2963

Error Message

string literals are not permitted as parameter to a template specialization

One or more of the template parameter expressions is a string literal.

This error may be fixed if you use a pointer to a string variable.

Compiler Error C2964

Error Message
invalid expression as template parameter

One or more of the template parameter expressions is not valid as a template parameter. Check the template declaration for parameter types.

Compiler Error C2969

Error Message
syntax error : 'symbol' : expected member function definition to end with '}'

A template member function definition has an unmatched closing brace.

The following sample generates C2969:

```
// C2969.cpp
// compile with: /c
class A {
   int i;
public:
   A(int i) {}
};

A anA(1);

class B {
   A a;
   B() : a(anA);   // C2969
   // try the following line instead
   // B() : a(anA) {}
};
```

Compiler Error C2970

Error Message
'class' : template parameter 'param' : 'arg' : an expression involving objects with internal linkage cannot be used as a non-type argument

You cannot use the name or address of a static variable as a template argument. The template class expects a const value that can be evaluated at compile time.

The following sample generates C2970:

```
// C2970.cpp
// compile with: /c
static int si;
// could declare nonstatic to resolve all errors
// int si;

template <int i>
class X {};

template <int *pi>
class Y {};

X<si> anX;   // C2970 cannot use static variable in templates

// this would also work
const int i = 10;
X<i> anX2;
```

Compiler Error C2971

Error Message
'class' : template parameter 'param' : 'arg' : a local variable cannot be used as a non-type argument

You cannot use the name or address of a local variable as a template argument.

The following sample generates C2971:

```
// C2971.cpp
template <int *pi>
class Y {};

int global_var = 0;

int main() {
   int local_var = 0;
   Y<&local_var> aY;   // C2971
   // try the following line instead
   // Y<&global_var> aY;
}
```

Compiler Error C2973

Error Message
invalid template argument 'number'

Check the template definition to find the correct types.

Compiler Error C2974

Error Message
invalid type argument 'number', type expected

The generic or template argument does not match the generic or template declaration. A type should appear within the angle brackets. Check the generic or template definition to find the correct types.

The following sample generates C2974:

```
// C2974.cpp
// C2974 expected
template <class T>
struct TC {};

template <typename T>
void tf(T){}

int main() {
   // Delete the following 2 lines to resolve
   TC<1>* tc;
   tf<"abc">("abc");

   TC<int>* tc;
   tf<const char *>("abc");
}
```

C2974 can also occur when using generics:

```
// C2974b.cpp
// compile with: /clr
// C2974 expected
using namespace System;
generic <class T>
ref struct GCtype {};

generic <typename T>
void gf(T){}

int main() {
   // Delete the following 2 lines to resolve
   GCtype<"a">^ gc;
   gf<"a">("abc");

   // OK
   GCtype<int>^ gc;
   gf<String ^>("abc");
}
```

Compiler Error C2975

Error Message
'arg' : invalid template argument for 'type', expected compile-time constant expression

The template argument does not match the template declaration; a constant expression should appear within the angle brackets. Variables are not allowed as template actual arguments. Check the template definition to find the correct types.

The following sample generates C2975:

```
// C2975.cpp
template<int I>
class X {};

int main() {
   int i = 4, j = 2;
   X<i + j> x1;   // C2975
   X<6> x2;   // OK
}
```

C2975 will also occur when you use __LINE__ as a compile-time constant with /ZI:

```
// C2975b.cpp
// compile with: /ZI
// processor: x86
template<long line>
void test(void) {}

int main() {
   test<__LINE__>();   // C2975
   test<__LINE__>();   // OK
}
```

Compiler Error C2976

Error Message
'identifier' : too few type arguments

A generic or template is missing one or more actual arguments. Check the generic or template declaration to find the correct number of parameters.

This error can be caused by missing template arguments in STL components.

The following sample generates C2976:

```
// C2976.cpp
template <class T>
struct TC {
   T t;
};
int main() {
   TC<>* t;   // C2976
   TC<int>* t2;   // OK
}
```

C2976 can also occur when using generics:

```
// C2976b.cpp
// compile with: /clr
generic <class T>
ref struct GC {
   T t;
};

int main() {
   GC<>^ g;   // C2976
   GC<int>^ g2;   // OK
}
```

Compiler Error C2977

Error Message
'identifier' : too many type arguments

A generic or template has too many actual arguments. Check the generic or template declaration to find the correct number of parameters.

The following sample generates C2977:

```
// C2977.cpp
// compile with: /c
template<class T, int i>
class MyClass {};

template MyClass< int , 1, 1 >;   // C2977
template MyClass< int , 1 >;   // OK
```

C2977 can also occur when using generics:

```
// C2977b.cpp
// compile with: /clr
// C2977 expected
generic <class T, class U>
void f(){}

generic <class T>
ref struct GC1 {};

int main() {
   // Delete the following 2 lines to resolve.
   GC1<int, char> ^ pgc1;
   f<int,int,int>();

   // OK
   GC1<int> ^ pgc1;
   f<int, int>();
}
```

Compiler Error C2978

Error Message

syntax error : expected 'keyword1' or 'keyword2'; found type 'keyword3'; non-type parameters are not supported in generics

A generic class was declared incorrectly. See Generics (Visual C++)for more information.

Example

The following sample generates C2978.

```
// C2978.cpp
// compile with: /clr /c
generic <ref class T>   // C2978
// try the following line instead
// generic <typename T>   // OK
ref class Utils {
   static void sort(T elems, size_t size);
};

generic <int>
// try the following line instead
// generic <class T>
ref class Utils2 {
   static void sort(T elems, size_t size);
};
```

Compiler Error C2979

Error Message
explicit specializations are not supported in generics

A generic class was declared incorrectly. See Generics (Visual C++) for more information.

Example
The following sample generates C2979.

```
// C2979.cpp
// compile with: /clr /c
generic <>
ref class Utils {};   // C2979 error

generic <class T>
ref class Utils2 {};   // OK
```

Compiler Error C2988

Error Message
unrecognizable template declaration/definition

The template declaration does not parse correctly. Check delimiters.

Compiler Error C2989

Error Message
'class' : class type has already been declared as a non-class type

The class generic or template redefines a non-template or non-generic class. Check header files for conflicts.

If you are using class template partial specializations, see Knowledge Base article Q240866.

For more information, see the Knowledge Base article Q240866. You can find Knowledge Base articles on the MSDN Library CD-ROM or at http://support.microsoft.com/support/http://support.microsoft.com/support/.

The following sample generates C2989:

```
// C2989.cpp
// compile with: /c
class C{};

template <class T>
class C{};  // C2989
class C2{};
```

C2989 can also occur when using generics:

```
// C2989b.cpp
// compile with: /clr /c
ref class GC1;

generic <typename T> ref class GC1;   // C2989
template <typename T> ref class GC2;

generic <typename T> ref class GC2;   // C2989
generic <typename T> ref class GCb;
template <typename T> ref class GC2;
generic <typename T> ref class GCc;
```

Compiler Error C2990

Error Message
'class' : non-class type as already been declared as a class type

The non generic or template class redefines a generic or template class. Check header files for conflicts.

The following sample generates C2990:

```
// C2990.cpp
// compile with: /c
template <class T>
class C{};
class C{};   // C2990
```

C2990 can also occur when using generics:

```
// C2990b.cpp
// compile with: /clr /c
generic <class T>
ref struct GC;

ref struct GC {};   // C2990
```

C2990 can also occur due to a breaking change in the Visual C++ compiler for Visual C++ 2005; the compiler now requires that multiple declarations for the same type be identical with respect to template specification. For more information, see Breaking Changes in the Visual C++ 2005 Compiler.

The following sample generates C2990:

```
// C2990c.cpp
// compile with: /c
template<class T>
class A;

template<class T>
struct A2 {
   friend class A;   // C2990
};

// OK
template<class T>
struct B {
   template<class T>
   friend class A;
};
```

Compiler Error C2991

Error Message
redefinition of type parameter 'parameter'

There was a type conflict between two generic or template definitions of *parameter*. When defining multiple generic or template parameters, you must use equivalent types.

The following sample generates C2991:

```
// C2991.cpp
// compile with: /c
template<class T, class T> struct TC {};   // C2991
// try the following line instead
// template<class T, class T2> struct TC {};
```

C2991 can also occur when using generics:

```
// C2991b.cpp
// compile with: /clr /c
generic<class T,class T> ref struct GC {};   // C2991
// try the following line instead
// generic<class T,class T2> ref struct GC {};
```

Compiler Error C2992

Error Message
'class' : invalid or missing type parameter list

The class is preceded by a **template** or **generic** keyword with missing or invalid parameters.

Example
The following sample generates C2992:

```
// C2992.cpp
// compile with: /c
template <class T>
struct TC1 {
   template <class U>
   struct TC2;
};

template <class T>   struct TC1<T>::TC2 {};   // C2992

// OK
template <class T>
template <class U>
struct TC1<T>::TC2 {};
 // C2992 can also occur when using generics:
// C2992c.cpp
// compile with: /clr /c
generic <class T>
ref struct GC1 {
   generic <class U>
   ref struct GC2;
};

generic <class T> ref struct GC1<T>::GC2 {};   // C2992

// OK
generic <class T>
generic <class U>
ref struct GC1<T>::GC2 {};
```

Compiler Error C2993

Error Message
'identifier' : illegal type for non-type template parameter 'parameter'

You cannot declare a template with a structure or union argument. Use pointers to pass structures and unions as template parameters.

The following sample generates C2993:

```
// C2993.cpp
// compile with: /c
// C2993 expected
struct MyStruct {
   int a;char b;
};

template <class T, struct MyStruct S>   // C2993

// try the following line instead
// template <class T, struct MyStruct * S>
class CMyClass {};
```

This error will also be generated as a result of compiler conformance work that was done in Visual Studio .NET 2003: floating point non-type template parameters no longer allowed. The C++ standard does not allow floating point non-type template parameters.

If it is a function template, use a function argument to pass in the floating point non-type template parameter (this code will be valid in the Visual Studio .NET 2003 and Visual Studio .NET versions of Visual C++). If it is a class template, there is no easy workaround.

See Summary of Compile-Time Breaking Changes for more information.

```
// C2993b.cpp
// compile with: /c
template<class T, float f> void func(T) {}   // C2993

// OK
template<class T>   void func2(T, float) {}
```

Compiler Error C2994

Error Message
unnamed class in template parameter list

You cannot use the **class** keyword as a template argument without specifying a class name.

Compiler Error C2995

Error Message
'function' : function template has already been defined

Make sure that there is only one definition for each member function of a templated class.

The following sample generates C2995:

```
// C2995.cpp
// compile with: /c
template <class T>
void Test(T x){}

template <class T> void Test(T x){}   // C2995
template <class T> void Test2(T x){}   // OK
```

Compiler Error C2996

Error Message

'function' : recursive function template definition

A function definition attempts to instantiate its root templated class. Recursive template instantiations are not allowed.

Compiler Error C2998

Error Message
'identifier' : cannot be a template definition

The compiler could not process the syntax used in the template definition.

The following sample generates C2998:

```
// C2998.cpp
// compile with: /c
template <class T> int x = 1018; // C2998
```

Compiler Error C2999

Error Message
UNKNOWN ERROR

Please choose the Technical Support command on the Visual C++ Help menu, or open the Technical Support help file for more information

Note the circumstances of the error, try to isolate the problem and create a reproducible test case, then contact Microsoft Product Support Services.

Compiler Error C3001

Error Message
'error_text' : expected an OpenMP directive name

The **omp** pragma must be followed by a directive.

The following sample generates C3001:

```
// C3001.c
// compile with: /openmp
int main()
{
   #pragma omp   // C3001 missing token
}
```

Compiler Error C3002

Error Message
'name1 name2' : multiple OpenMP directive names

Multiple directive names are not allowed.

The following sample generates C3002:

```
// C3002.c
// compile with: /openmp
int main()
{
   #pragma omp parallel single   // C3002
}
```

Compiler Error C3003

Error Message
'directive' : OpenMP directive name not allowed after directive clauses

An OpenMP directive name cannot follow an OpenMP directive clause.

The following sample generates C3003:

```
// C3003.c
// compile with: /openmp
int main()
{
   int x, y, z;
   #pragma omp parallel shared(x, y, z) for   // C3003
}
```

Compiler Error C3004

Error Message
'clause' : clause not valid on OpenMP 'directive' directive

An OpenMP clause was used on a directive for which it is not enabled.

The following sample generates C3004:

```
// C3004.c
// compile with: /openmp
int main()
{
   int x, y, z;

   // Shared clause not allowed for 'single' directive.
   #pragma omp single shared(x, y)   // C3004

   x = y;
}
```

Compiler Error C3005

Error Message
'error_text' : unexpected token encountered on OpenMP 'directive' directive

An OpenMP directive was ill formed.

The following sample generates C3005:

```
// C3005.c
// compile with: /openmp
int main()
{
   #pragma omp parallel + for   // C3005
}
```

C3005 can also occur if you put an open brace on the same line as the pragma.

```
// C3005b.c
// compile with: /openmp
int main() {
   #pragma omp parallel {   // C3005 put open brace on next line
   lbl2:;
   }
   goto lbl2;
}
```

Compiler Error C3006

Error Message
'clause' : clause on OpenMP 'directive' directive is missing an expected argument

An OpenMP directive did not have an expected argument.

The following sample generates C3006:

```
// C3006.c
// compile with: /openmp
int main()
{
   #pragma omp parallel shared   // C3006
   // Try the following line instead:
   // #pragma omp parallel shared(x) {}

}
```

Compiler Error C3007

Error Message
'arg' : clause on OpenMP 'directive' directive does not take an argument

An OpenMP directive had an argument, but the directive does not take an argument.

The following sample generates C3007:

```
// C3007.c
// compile with: /openmp
int main()
{
   #pragma omp parallel for ordered(2)   // C3007
}
```

Compiler Error C3008

Error Message
'arg' : argument is missing closing ')' on OpenMP 'directive' directive

An OpenMP directive that takes an argument did not have a closing parenthesis.

The following sample generates C3008:

```
// C3008.c
// compile with: /openmp
int main()
{
   int x, y, z;
   #pragma omp parallel shared(x   // C3008
   // Try the following line instead:
   #pragma omp parallel shared(x)
   {
   }
}
```

Compiler Error C3009

Error Message
'label' : jump into OpenMP structured block not allowed

Code cannot jump into or out of an OpenMP block.

The following sample generates C3009:

```
// C3009.c
// compile with: /openmp
int main() {
   #pragma omp parallel
   {
   lbl2:;
   }
   goto lbl2;   // C3009
}
```

Compiler Error C3010

Error Message
'label' : jump out of OpenMP structured block not allowed

Code cannot jump into or out of an OpenMP block.

The following sample generates C3010:

```
// C3010.c
// compile with: /openmp
int main() {
   #pragma omp parallel
   {
      #pragma omp parallel
      {
         goto lbl3;
      }
   }
   lbl3:;   // C3010
}
```

Compiler Error C3011

Error Message
inline assembly not allowed directly within a parallel region

An **omp** parallel region cannot contain inline assembly instructions.

The following sample generates C3011:

```
// C3011.cpp
// compile with: /openmp
// processor: /x86
int main() {
   int   n = 0;

   #pragma omp parallel
   {
      _asm mov eax, n   // Delete this line to resolve this error.
   }   // C3011
}
```

Compiler Error C3012

Error Message
'intrinsic' : intrinsic function not allowed directly within a parallel region

A compiler intrinsic function is not allowed in an **omp parallel** region.

The following sample generates C3012:

```
// C3012.cpp
// compile with: /openmp
#ifdef __cplusplus
extern "C"
{
#endif

void* _ReturnAddress();

#ifdef __cplusplus
}
#endif

int main()
{
   _ReturnAddress();   // OK

   #pragma omp parallel
   {
      _ReturnAddress();   // C3012
   }
}
```

Compiler Error C3013

Error Message
'clause' : clause may only appear once on OpenMP 'directive' directive

A clause appeared twice on the same directive. Delete one occurrence of the clause.

The following sample generates C3013:

```
// C3013.cpp
// compile with: /openmp
int main() {
   int a, b, c, x, y, z;

   #pragma omp parallel shared(a,b,c) private(x)

   #pragma omp for nowait private(x) nowait   // C3013
   // The previous line generates C3013, with two nowait clauses
   // try the following line instead:
   // #pragma omp for nowait private(x)
   for (a = 0 ; a < 10 ; ++a) {
   }
}
```

Compiler Error C3014

Error Message
expected a for loop following OpenMP 'directive' directive

It is an error for anything other than a **for** loop to immediately follow a **#pragma omp for** directive.

The following sample generates C3014:

```
// C3014.cpp
// compile with: /openmp
int main()
{
   int i = 0;

   #pragma omp parallel
   {
      #pragma omp for
      for (i = 0; i < 10; ++i)   // OK
      {
      }
   }

   #pragma omp parallel for
   for (i = 0; i < 10; ++i)   // OK
   {
   }

   #pragma omp parallel
   {
      #pragma omp for
      {   // C3014
         for (i = 0; i < 10; ++i)
         {
         }
      }
   }

   #pragma omp parallel for
   {   // C3014
      for (i = 0; i < 10; ++i)
      {
      }
   }

   #pragma omp parallel
   {
      #pragma omp for
      i *= 2;   // C3014
      for (i = 0; i < 10; ++i)
      {
      }
   }

   #pragma omp parallel for
   i *= 2;   // C3014
   for (i = 0; i < 10; ++i)
   {
   }
}
```

Compiler Error C3015

Error Message
initialization in OpenMP 'for' statement has improper form

A **for** loop in an OpenMP statement must be fully and explicitly specified.

The following sample generates C3015:

```
// C3015.cpp
// compile with: /openmp
int main()
{
   int i = 0, j = 10;

   #pragma omp parallel
   {
      #pragma omp for
      for (; i < 0; i += j)   // C3015
      // Try the following line instead:
      // for (i = 0; i < 0; i++)
         --j;
   }
}
```

Compiler Error C3016

Error Message
'var' : index variable in OpenMP 'for' statement must have signed integral type

The index variable in an OpenMP **for** statement must be a signed integral type.

The following sample generates C3016:

```
// C3016.cpp
// compile with: /openmp
int main()
{
   #pragma omp parallel
   {
      unsigned int i = 0;
      // Try the following line instead:
      // int i = 0;

      #pragma omp for
      for (i = 0; i <= 10; ++i)   // C3016
      {
      }
   }
}
```

Compiler Error C3017

Error Message
termination test in OpenMP 'for' statement has improper form

A **for** loop in an OpenMP statement must be fully and explicitly specified.

The following sample generates C3017:

```
// C3017.cpp
// compile with: /openmp
int main()
{
   int i = 0, j = 10;

   #pragma omp parallel
   {
      #pragma omp for
      for (i = 0; i; ++i)   // C3017
      // Try the following line instead:
      // for (i = 0; i < 10; ++i)
         ;
   }
}
```

Compiler Error C3018

Error Message
'var1' : OpenMP 'for' test or increment must use index variable 'var2'

A **for** loop in an OpenMP statement must use the same variable for its test and increment as it uses for its index.

The following sample generates C3018:

```
// C3018.cpp
// compile with: /openmp
int main()
{
   int i = 0, j = 5;

   #pragma omp parallel
   {
      #pragma omp for
      for (i = 0; j < 10; ++i)   // C3018
      // try the the following line instead
      // for (i = 0; i < 10; ++i)
         j *= 2;

      #pragma omp for
      for (i = 0; i < 10; j = j + i)   // C3018
      // try the the following line instead
      // for (i = 0; i < 10; i = j + i)
         j *= 2;
   }
}
```

Compiler Error C3019

Error Message
increment in OpenMP 'for' statement has improper form

The increment part of an OpenMP **for** loop must use the index variable both on the left and right side of the operator.

The following sample generates C3019:

```
// C3019.cpp
// compile with: /openmp
int main()
{
   int i = 0, j = 1, n = 3;

   #pragma omp parallel
   {
      #pragma omp for
      for (i = 0; i < 10; i = j + n)   // C3019
      // Try the following line instead:
      // for (i = 0; i < 10; i++)
         j *= 2;
   }
}
```

Compiler Error C3020

Error Message
'var' : index variable of OpenMP 'for' loop cannot be modified in loop body

An OpenMP **for** loop may not modify the index (loop counter) in the body of the **for** loop.

The following sample generates C3020:

```
// C3020.cpp
// compile with: /openmp
int main() {
   int i = 0, n = 3;

   #pragma omp parallel
   {
      #pragma omp for
      for (i = 0; i < 10; i += n)
         i *= 2;   // C3020
         // try the following line instead
         // n++;
   }
}
```

A variable declared with lastprivate cannot be used as the index inside a parallelized loop.

The following sample will give C3020 for the second lastprivate because that lastprivate will trigger a write to idx_a within the outermost for loop. The first lastprivate doesn't give an error because that lastprivate triggers a write to idx_a outside the outermost for loop (technically, at the very end of the last iteration). The following sample generates C3020.

```
// C3020b.cpp
// compile with: /openmp /c
float a[100][100];
int idx_a, idx_b;
void test(int first, int last)
{
   #pragma omp parallel for lastprivate(idx_a)
   for (idx_a = first; idx_a <= last; ++idx_a) {
      #pragma omp parallel for lastprivate(idx_a)   // C3020
      for (idx_b = first; idx_b <= last; ++idx_b) {
         a[idx_a][idx_b] += 1.0f;
      }
   }
}
```

The following sample demonstrates a possible resolution:

```
// C3020c.cpp
// compile with: /openmp /c
float a[100][100];
int idx_a, idx_b;
void test(int first, int last)
{
   #pragma omp parallel for lastprivate(idx_a)
   for (idx_a = first; idx_a <= last; ++idx_a) {
      #pragma omp parallel for lastprivate(idx_b)
      for (idx_b = first; idx_b <= last; ++idx_b) {
         a[idx_a][idx_b] += 1.0f;
      }
   }
}
```

Compiler Error C3021

Error Message
'arg' : argument is empty on OpenMP directive 'directive'

An argument is required for an OpenMP directive.

Example
The following sample generates C3021:

```
// C3021.cpp
// compile with: /openmp
#include <stdio.h>
#include "omp.h"

int g = 0;

int main()
{
    int x, y, i;

    #pragma omp parallel for schedule(static,)   // C3021
    for (i = 0; i < 10; ++i) ;

    #pragma omp parallel for schedule()   // C3021
    for (i = 0; i < 10; ++i)
        printf_s("Hello world, thread %d, iteration %d\n",
                 omp_get_thread_num(), i);

    #pragma omp parallel reduction()   // C3021
      ;

    #pragma omp parallel reduction(+ :)   // C3021
      ;

    //
    // The following shows correct syntax examples.
    //
    #pragma omp parallel reduction(+ : x, y)
      ;

    #pragma omp parallel reduction(* : x, y)
      ;

    #pragma omp parallel reduction(- : x, y)
      ;

    #pragma omp parallel reduction(& : x, y)
      ;

    #pragma omp parallel reduction(^ : x, y)
      ;

    #pragma omp parallel reduction(| : x, y)
      ;

    #pragma omp parallel reduction(&& : x, y)
      ;

    #pragma omp parallel reduction(|| : x, y)
      ;
}
```

Compiler Error C3022

Error Message
'clause' : invalid schedule kind of 'value' on OpenMP 'directive' directive

An unsupported value was passed to a clause.

The following sample generates C3022:

```
// C3022.cpp
// compile with: /openmp /link vcomps.lib
#include <stdio.h>
#include "omp.h"

int main() {
   int i;

   #pragma omp parallel for schedule(10)   // C3022
   for (i = 0; i < 10; ++i) ;

   #pragma omp parallel for schedule(x)   // C3022
   for (i = 0; i < 10; ++i) ;

   // OK
   #pragma omp parallel for schedule(runtime)
   for (i = 0; i < 10; ++i)
   ;
}
```

Compiler Error C3023

Error Message
'value' : unexpected token encountered in argument to OpenMP 'clause' clause

The values passed to a clause were not valid.

The following sample generates C3023:

```
// C3023.cpp
// compile with: /openmp /link vcomps.lib
#include <stdio.h>
#include "omp.h"

int main() {
   int i;

   #pragma omp parallel for schedule(dynamic 10)   // C3023
   for (i = 0; i < 10; ++i) ;

   #pragma omp parallel for schedule(dynamic;10)   // C3023
   for (i = 0; i < 10; ++i) ;

   // OK
   #pragma omp parallel for schedule(dynamic, 10)
   for (i = 0; i < 10; ++i)
   ;
}
```

Compiler Error C3024

Error Message
'schedule(runtime)' : chunk_size expression is not allowed

A value cannot be passed to the run-time parameter of the schedule clause.

The following sample generates C3024:

```
// C3024.cpp
// compile with: /openmp /link vcomps.lib
#include <stdio.h>
#include "omp.h"

int main() {
   int i;

   #pragma omp parallel for schedule(runtime, 10)   // C3024
   for (i = 0; i < 10; ++i) ;

   #pragma omp parallel for schedule(runtime)   // OK
   for (i = 0; i < 10; ++i) ;
}
```

Compiler Error C3025

Error Message
'clause' : integral expression expected

A clause requires an integer expression but was given a noninteger expression.

Example
The following sample generates C3025.

```
// C3025.cpp
// compile with: /openmp /link vcomps.lib
#include <stdio.h>
#include "omp.h"

float f = 2.0F;

int main()
{
    int i = 0;

    // OK
    puts("Test with int");
    #pragma omp parallel for num_threads(i)
    for (i = 1; i <= 2; ++i)
        printf_s("Hello World - thread %d - iteration %d\n",
                 omp_get_thread_num(), i);


    puts("Test with float");
    #pragma omp parallel for num_threads(f)   // C3025
    for (i = 1; i <= 2; ++i)
        printf_s("Hello World - thread %d - iteration %d\n",
                 omp_get_thread_num(), i);
}
```

Compiler Error C3026

Error Message
'clause' : constant expression must be positive

A clause was passed an integer value, but the value was not a positive number. The number must be positive.

Example
The following sample generates C3026:

```
// C3026.cpp
// compile with: /openmp /link vcomps.lib
#include <stdio.h>
#include "omp.h"

int main()
{
    int i;
    const int i1 = 0;

    #pragma omp parallel for num_threads(i1)   // C3026
    for (i = 1; i <= 2; ++i)
        printf_s("Hello World - thread %d - iteration %d\n",
                 omp_get_thread_num(), i);

    #pragma omp parallel for num_threads(i1 + 1)   // OK
    for (i = 1; i <= 2; ++i)
        printf_s("Hello World - thread %d - iteration %d\n",
                 omp_get_thread_num(), i);
}
```

Compiler Error C3027

Error Message
'clause' : arithmetic or pointer expression expected

A clause that requires an arithmetic or pointer expression was passed another kind of expression.

Example
The following sample generates C3027:

```
// C3027.cpp
// compile with: /openmp /link vcomps.lib
#include <stdio.h>
#include "omp.h"

struct MyStruct
{
    int x;
} m_MyStruct;

int main()
{
    int i;

    puts("Test with class MyStruct:\n");
    #pragma omp parallel for if(m_MyStruct)   // C3027
    for (i = 1; i <= 2; ++i)
        printf_s("Hello World - thread %d - iteration %d\n",
                 omp_get_thread_num(), i);

    puts("Test with int:\n");
    #pragma omp parallel for if(9)   // OK
    for (i = 1; i <= 2; ++i)
        printf_s("Hello World - thread %d - iteration %d\n",
                 omp_get_thread_num(), i);
}
```

Compiler Error C3028

Error Message
'member' : only a variable or static data member can be used in a data-sharing clause

A symbol other than a variable or static data member was passed to the reduction clause.

The following sample generates C3028:

```
// C3028.cpp
// compile with: /openmp /link vcomps.lib
int g_i;

class MyClass {
public:
   MyClass();
   MyClass(int x);
   static int x_public;
   int mbr;
private:
   static int x_private;
};

int MyClass::x_public;
int MyClass::x_private;

namespace XyzNS {
   struct xyz { int x; };
   xyz xyz;
}

namespace NS {
   int a1;
   struct Bar {
      static MyClass MyClass;
   };
   struct Baz : public Bar {
      using NS::Bar::MyClass;
   };
}

MyClass NS::Bar::MyClass;

typedef int MyInt;

template <class T, size_t n> class CTempl {
public:
   static T public_array[n];
private:
   static T private_array[n];
};

template<class T,size_t n> T CTempl<T,n>::public_array[n];
template<class T,size_t n> T CTempl<T,n>::private_array[n];

CTempl<int,5> tx;

struct Incomplete;
extern Incomplete inc;

MyClass::MyClass(int x) {

   #pragma omp parallel reduction(+: x, g_i, x_public, x_private)
   // OK
      ;
```

```
   #pragma omp parallel reduction(+: x, g_i, MyClass::x_public,
   MyClass::x_private)
   // OK
      ;

   #pragma omp parallel reduction(+: mbr)
   // C3028, member of a class.
      ;
}

int main() {

   #pragma omp parallel reduction(+:main)
   // C3028, main is a function.
      ;

   #pragma omp parallel reduction(+: XyzNS)
   // C3028, XyzNS is a namespace.
      ;
}
```

Compiler Error C3029

Error Message
'symbol' : can only appear once in data-sharing clauses in an OpenMP directive

A symbol was used more than once in one or more clauses in a directive. The symbol can only be used once in the directive.

The following sample generates C3029:

```
// C3029.cpp
// compile with: /openmp /link vcomps.lib
#include "omp.h"

int g_i;

int main() {
   int i, x;

   #pragma omp parallel reduction(+ : x, x)   // C3029
      ;

   #pragma omp parallel reduction(+ : x)   // OK
      ;

   #pragma omp parallel private(x) reduction(+ : x)   // C3029
      ;

   #pragma omp parallel reduction(+ : x)   // OK
      ;
}
```

Compiler Error C3030

Error Message
'var' : variable in 'reduction' clause/directive cannot have reference type

You can only pass value parameters to certain clauses, such as the reduction clause.

The following sample generates C3030:

```
// C3030.cpp
// compile with: /openmp /link vcomps.lib
#include "omp.h"

void test(int &r) {
   #pragma omp parallel reduction(+ : r)   // C3030
      ;
}

void test2(int r) {
   #pragma omp parallel reduction(+ : r)   // OK
      ;
}

int main(int argc, char** argv) {
   int& r = *((int*)argv);
   int s = *((int*)argv);

   #pragma omp parallel reduction(+ : r)   // C3030
      ;

   #pragma omp parallel reduction(+ : s)   // OK
      ;
}
```

Compiler Error C3031

Error Message
'var' : variable in 'reduction' clause must have scalar arithmetic type

A variable of the wrong type was passed to a reduction clause.

The following sample generates C3031:

```
// C3031.cpp
// compile with: /openmp /link vcomps.lib
#include <stdio.h>
#include "omp.h"

typedef struct {
   int n;
} Incomplete;

extern Incomplete inc;
int i = 9;

int main() {
   #pragma omp parallel reduction(+: inc)   // C3031
      ;

   #pragma omp parallel reduction(+: i)     // OK
      ;
}
```

Compiler Error C3032

Error Message
'var' : variable in 'clause' clause cannot have incomplete type 'type'

Types passed to certain clauses must be fully visible to the compiler.

The following sample generates C3032:

```
// C3032.cpp
// compile with: /openmp /link vcomps.lib
#include "omp.h"

struct Incomplete;
extern struct Incomplete inc;

int main() {
   int i = 9;
   #pragma omp parallel private(inc)   // C3032
      ;

   #pragma omp parallel private(i)     // OK
      ;

}
```

Compiler Error C3033

Error Message
'var' : variable in 'clause' clause cannot have const-qualified type

Values passed to certain clauses cannot be **const** variables.

The following sample generates C3033:

```
// C3033.cpp
// compile with: /openmp /link vcomps.lib
int main() {
   const int val = 1;
   int val2 = 1;

   #pragma omp parallel reduction(+ : val)   // C3033
   ;

   #pragma omp parallel reduction(+ : val2)   // OK
   ;
}
```

Compiler Error C3034

Error Message
OpenMP 'directive1' directive cannot be directly nested within 'directive2' directive

Some directives cannot be nested. To fix this error, you can merge the statements of both directives into the block of one directive, or you can construct consecutive directives.

The following sample generates C3034:

```
// C3034.cpp
// compile with: /openmp /link vcomps.lib
int main() {

   #pragma omp single
   {
      #pragma omp single   // C3034
      {
      ;
      }
   }

   // Two consecutive single clauses are OK.
   #pragma omp single
   {
   }

   #pragma omp single
   {
   }
}
```

Compiler Error C3035

Error Message
OpenMP 'ordered' directive must bind directly to a 'for' or 'parallel for' directive with the 'ordered' clause

An ordered clause was ill formed.

The following sample generates C3035:

```
// C3035.cpp
// compile with: /openmp /link vcomps.lib
int main() {
   int n = 0, x, i;

   #pragma omp parallel private(n)
   {
      #pragma omp ordered   // C3035
      // Try the following line instead:
      // #pragma omp for ordered
       for (i = 0 ; i < 10 ; ++i)
         ;
   }
}
```

Compiler Error C3036

Error Message
'operator' : invalid operator token in OpenMP 'reduction' clause

A reduction clause was not specified correctly.

The following sample generates C3036:

```
// C3036.cpp
// compile with: /openmp
static float a[1000], b[1000], c[1000];
void test1(int first, int last) {
   static float dp = 0.0f;
   #pragma omp for nowait reduction(.:dp)   // C3036
   // try the following line instead
   // #pragma omp for nowait reduction(+: dp)
   for (int i = first ; i <= last ; ++i)
      dp += a[i] * b[i];
}
```

Compiler Error C3037

Error Message
'var' : variable in 'reduction' clause must be shared in enclosing context

A variable specified in a reduction clause may not be private to each thread in the context.

The following sample generates C3037:

```
// C3037.cpp
// compile with: /openmp /c
int g_i;

int main() {
   int i;

   #pragma omp parallel private(g_i)
   // try the following line instead
   // #pragma omp parallel
   {
      #pragma omp for reduction(+:g_i)   // C3037
      for (i = 0 ; i < 10 ; ++i) {
         g_i += i;
      }
   }
}
```

Compiler Error C3038

Error Message
'var' : variable in 'private' clause cannot be a reduction variable in enclosing context

Variables that appear in the reduction clause of a parallel directive cannot be specified in a private clause on a work-sharing directive that binds to the parallel construct.

The following sample generates C3038:

```
// C3038.cpp
// compile with: /openmp /c
int g_i, g_i2;

int main() {
   int i;

   #pragma omp parallel reduction(+: g_i)
   {
      #pragma omp for private(g_i)   // C3038
      // try the following line instead
      // #pragma omp for private(g_i2)
      for (i = 0; i < 10; ++i)
         g_i += i;
   }
}
```

Compiler Error C3039

Error Message

'var' : index variable in OpenMP 'for' statement cannot be a reduction variable

An index variable is implicitly private, so the variable cannot be used in a reduction clause in the enclosing parallel directive.

Example

The following sample generates C3039:

```
// C3039.cpp
// compile with: /openmp /c
int g_i;

int main() {
   int i;

   #pragma omp parallel reduction(+: i)
   {
      #pragma omp for
      for (i = 0; i < 10; ++i)   // C3039
         g_i += i;
   }
}
```

Compiler Error C3040

Error Message
'var' : type of variable in 'reduction' clause is incompatible with reduction operator 'operator'

A variable in a reduction clause cannot be used with the reduction operator.

The following sample generates C3040:

```
// C3040.cpp
// compile with: /openmp /c
#include "omp.h"
double d;

int main() {
   #pragma omp parallel reduction(&:d)   // C3040
      ;

   #pragma omp parallel reduction(-:d)  // OK
      ;
}
```

Compiler Error C3041

Error Message
'var' : variable in 'copyprivate' clause must be private in enclosing context

A variable passed to copyprivate cannot be shared in the enclosing context.

The following sample generates C3041:

```
// C3041.cpp
// compile with: /openmp /c
#include "omp.h"
double d;
int main() {
   #pragma omp parallel shared(d)
   // try the following line instead
   // #pragma omp parallel private(d)
   {
      // or don't make d copyprivate
      #pragma omp single copyprivate(d)   // C3041
      {
      }
   }
}
```

Compiler Error C3042

Error Message
'copyprivate' and 'nowait' clauses cannot appear together on OpenMP 'directive' directive

The copyprivate and nowait clauses are mutually exclusive on the specified directive. To fix this error, remove one or both of the **copyprivate** or **nowait** clauses.

The following sample generates C3042:

```
// C3042.cpp
// compile with: /openmp /c
#include <stdio.h>
#include "omp.h"

double d;

int main() {
    #pragma omp parallel private(d)
   {
      #pragma omp single copyprivate(d) nowait   // C3042
      {
      }
   }
}
```

Compiler Error C3043

Error Message
OpenMP 'critical' directive cannot be nested in 'critical' directive with same name

A critical directive cannot be nested in a **critical** directive that uses the same name.

The following sample generates C3043:

```
// C3043.cpp
// compile with: /openmp /c
#include "omp.h"

int main() {
   int n1 = 1, n2 = 2, n3 = 3;

   #pragma omp parallel
   {
      ++n2;

      #pragma omp critical(MyTest)
      {
         ++n2;

         #pragma omp critical(MyTest)   // C3043
         // try the following line instead
         // #pragma omp critical(MyTest2)
         {
            ++n3;
         }
      }
   }
}
```

Compiler Error C3044

Error Message
'section' : only allowed directly nested under an OpenMP 'sections' directive

The compiler found a **section** directive was used incorrectly. For more information, see sections.

The following sample generates C3044:

```
// C3044.cpp
// compile with: /openmp /c
#include "omp.h"
int main() {
   int n2 = 2, n3 = 3;

   #pragma omp parallel
   {
      ++n2;

      #pragma omp sections
      {
         ++n2;
      }

      #pragma omp section   // C3044
      {
         ++n3;
      }
   }

   #pragma omp parallel
   {
      ++n2;

      #pragma omp sections
      {
         #pragma omp section   // OK
         {
            ++n3;
         }
      }
   }
}
```

Compiler Error C3045

Error Message
Expected a compound statement following OpenMP 'sections' directive. Missing '{'

A code block delimited by braces must follow a sections directive.

The following sample generates C3045:

```
// C3045.cpp
// compile with: /openmp /c
#include "omp.h"

int main() {
   int n2 = 2, n3 = 3;

   #pragma omp parallel
   {
      ++n2;

      #pragma omp sections
         ++n2;   // C3045

      #pragma omp sections   // OK
      {
         ++n3;
      }
   }
}
```

Compiler Error C3046

Error Message
Missing structured block in an OpenMP '#pragma omp sections' region

A sections directive has an empty code block.

The following sample generates C3046:

```
// C3046.cpp
// compile with: /openmp /c
#include "omp.h"

int main() {
   int n2 = 2, n3 = 3;

   #pragma omp parallel
   {
      ++n2;

      #pragma omp sections
      {
/*
         ++n2;

         #pragma omp section
         {
            ++n3;
         }
*/
       }   // C3046 uncomment code to resolve this C3046
   }
}
```

Compiler Error C3047

Error Message
Structured block in an OpenMP 'sections' region must be preceded by '#pragma omp section'

Any code in a code block introduced by a sections directive must be in a code block introduced by a **section** directive.

The following sample generates C3047:

```
// C3047.cpp
// compile with: /openmp /c
#include "omp.h"

int main() {
   int n2 = 2, n3 = 3;

   #pragma omp parallel
   {
      ++n2;

      #pragma omp sections
      {


         #pragma omp section
         {
            ++n3;
         }

         ++n2;   // C3047 not enclosed in #pragma omp section
      }
   }
}
```

Compiler Error C3048

Error Message
Expression following '#pragma omp atomic' has improper form

An atomic directive was incorrectly specified.

The following sample generates C3048:

```
// C3048.cpp
// compile with: /openmp vcomps.lib
#include "omp.h"
#include <stdio.h>

int main() {
   int a[10];
   omp_set_num_threads(4);
   #pragma omp parallel
   {
      #pragma omp atomic
      a[0] = 1;   // C3048
      // try the following line instead
      // a[0] += 1;
   }
}
```

Compiler Error C3049

Error Message
'arg' : invalid argument in OpenMP 'default' clause

An incorrect value was passed to a default clause.

The following sample generates C3049:

```
// C3049.cpp
// compile with: /openmp /c
int main() {
   int n1 = 1;

   #pragma omp parallel default(private)   // C3049
   // try the following line instead
   // #pragma omp parallel default(shared)
   {
      ++n1;
   }
}
```

Compiler Error C3050

Error Message
'type1' : a ref class cannot inherit from 'type1'

`System::ValueType` cannot be a base class for a reference type.

The following sample generates C3050:

```
// C3050.cpp
// compile with: /clr /LD
ref struct X : System::ValueType {};   // C3050
ref struct Y {};   // OK
```

Compiler Error C3052

Error Message
'var' : variable doesn't appear in a data-sharing clause under a default(none) clause

If default(none) is used, any variable used in the structured block must be explicitly specified as either shared or private.

The following sample generates C3052:

```
// C3052.cpp
// compile with: /openmp /c
int main() {
   int n1 = 1;

   #pragma omp parallel default(none) // shared(n1) private(n1)
   {
      n1 = 0;   // C3052 use either a shared or private clause
   }
}
```

Compiler Error C3053

Error Message
'symbol' : 'threadprivate' is only valid for global or static data items

Symbols passed to threadprivate must either be global or static.

The following sample generates C3053:

```
// C3053.cpp
// compile with: /openmp
void Test() {
   int x, y;
   #pragma omp threadprivate(x, y)   // C3053
   #pragma omp parallel copyin(x, y)
   {
      x = y;
   }
}
```

Possible resolution:

```
// C3053b.cpp
// compile with: /openmp /LD
int x, y;
#pragma omp threadprivate(x, y)

void Test() {
   #pragma omp parallel copyin(x, y)
   {
      x = y;
   }
}
```

Compiler Error C3054

Error Message

'#pragma omp parallel' is currently not supported in a generic class or function

For more information, see Generics (Visual C++) and OpenMP in Visual C++.

Example

The following sample generates C3054.

```
// C3054.cpp
// compile with: /openmp /clr /c
#include <omp.h>

ref struct MyBaseClass {
   // Delete the following 7 lines to resolve.
   generic <class ItemType>
   void Test(ItemType i) {   // C3054
      #pragma omp parallel num_threads(4)
      {
         int i = omp_get_thread_num();
      }
   }

   // OK
   void Test2() {
      #pragma omp parallel num_threads(4)
      {
         int i = omp_get_thread_num();
      }
   }
};
```

Compiler Error C3055

Error Message
'symbol' : symbol cannot be referenced before it is used in 'threadprivate' directive

A symbol was referenced and then used in a threadprivate clause, which is not allowed.

The following sample generates C3055:

```
// C3055.cpp
// compile with: /openmp
int x, y;
int z = x;
#pragma omp threadprivate(x, y)   // C3055

void test() {
   #pragma omp parallel copyin(x, y)
   {
      x = y;
   }
}
```

Possible resolution:

```
// C3055b.cpp
// compile with: /openmp /LD
int x, y, z;
#pragma omp threadprivate(x, y)

void test() {
   #pragma omp parallel copyin(x, y)
   {
      x = y;
   }
}
```

Compiler Error C3056

Error Message
'symbol' : symbol is not in the same scope with 'threadprivate' directive

A symbol used in a threadprivate clause must be in the same scope as the **threadprivate** clause.

The following sample generates C3056:

```
// C3056.cpp
// compile with: /openmp
int x, y;
void test() {
   #pragma omp threadprivate(x, y)   // C3056
   #pragma omp parallel copyin(x, y)
   {
      x = y;
   }
}
```

Possible resolution:

```
// C3056b.cpp
// compile with: /openmp /LD
int x, y;
#pragma omp threadprivate(x, y)
void test() {
   #pragma omp parallel copyin(x, y)
   {
      x = y;
   }
}
```

Compiler Error C3057

Error Message
'symbol' : dynamic initialization of 'threadprivate' symbols is not currently supported

The initialized value of a symbol used in a threadprivate clause must be known at compile time.

The following sample generates C3057:

```
// C3057.cpp
// compile with: /openmp /c
extern int f();
int x, y = f();
int a, b;
#pragma omp threadprivate(x, y)   // C3057

#pragma omp threadprivate(a, b)

int main() {
   // Delete the following 4 lines to resolve.
   #pragma omp parallel copyin(x, y)
   {
      x = y;
   }

   #pragma omp parallel copyin(a, b)
   {
      a = b;
   }
}
```

The following sample generates C3057:

```
// C3057b.cpp
// compile with: /openmp /c
extern int Initialize();
int main() {
   #pragma omp parallel
   {
      static int var = Initialize();
      #pragma omp threadprivate(var)   // C3057
   }

   // OK
   #pragma omp parallel
   {
      static int var2;
      static bool initialized2;
      #pragma omp threadprivate(var2, initialized2)
      if (!initialized2) {
         var2 = Initialize();
         initialized2 = true;
      }
   }
}
```

Compiler Error C3058

Error Message
'symbol' : symbol not declared as 'threadprivate' before it is used in the 'copyin' clause

A symbol must first be declared threadprivate before it can be used in a copyin clause.

The following sample generates C3058:

```
// C3058.cpp
// compile with: /openmp
int x, y, z;
#pragma omp threadprivate(x, z)

void test() {
   #pragma omp parallel copyin(x, y)   // C3058
   {
   }
}
```

Possible resolution:

```
// C3058b.cpp
// compile with: /openmp /LD
int x, y, z;
#pragma omp threadprivate(x, y)

void test() {
   #pragma omp parallel copyin(x, y)
   {
   }
}
```

Compiler Error C3059

Error Message
'var' : 'threadprivate' symbol cannot be used in the 'clause' clause

A threadprivate symbol was used in a clause.

The following sample generates C3059:

```
// C3059.cpp
// compile with: /openmp
#include "omp.h"
int x, y;
#pragma omp threadprivate(x, y)

int main() {
   #pragma omp parallel private(x, y)   // C3059
   {
      x = y;
   }
}
```

Possible resolution:

```
// C3059b.cpp
// compile with: /openmp
#include "omp.h"
int x = 0, y = 0;

int main() {
   #pragma omp parallel firstprivate(y) private(x)
   {
      x = y;
   }
}
```

Compiler Error C3060

Error Message
'member' : a friend function may not be defined inside a class using a qualified name (it may only be declared)

A friend function was defined using a qualified name, which is not allowed.

The following sample generates C3060:

```
// C3060.cpp
class A {
public:
   void func();
};

class C {
public:
   friend void A::func() { }   // C3060
   // Try the following line and the out of class definition:
   // friend void A::func();
};

// void A::func(){}
```

Compiler Error C3062

Error Message
'enum': enumerator requires value since the underlying type is 'type'

You can specify an underlying type for an enumeration. However, some types require you to assign values to each enumerator.

For more information on enums, see enum class.

The following sample generates C3062:

```
// C3062.cpp
// compile with: /clr

enum class MyEnum : bool { a };   // C3062
enum class MyEnum2 : bool { a = true};   // OK
```

Compiler Error C3063

Error Message
operator 'operator': all operands must have the same enumeration type

When using operators on enumerators, both operands must be of the enumeration type. For more information, see Using Operators and Enumerations.

The following sample generates C3063:

```
// C3063.cpp
// compile with: /clr
enum class E { a, b } e, mask;
int main() {
   if ( ( e & mask ) != 0 ) ;   // C3063 no operator!= (E, int)

   if ( ( e & mask ) != E() )   // OK
      ;
}
```

Compiler Error C3065

Error Message
property declaration at non-class scope is not allowed

The property __declspec modifier was used outside a class. A property can only be declared inside a class.

The following sample generates C3065:

```
// C3065.cpp
// compile with: /c
__declspec(property(get=get_i)) int i;   // C3065

class x {
public:
   __declspec(property(get=get_i)) int i;   // OK
};
```

Compiler Error C3066

Error Message
there are multiple ways that an object of this type can be called with these arguments

The compiler detected an ambiguous function call involving surrogates.

The following sample generates C3066:

```
// C3066.cpp
template <class T, class U> void func(T*, U*){}

typedef void (*PF)(const int*, const char*);
typedef void (*PF1)(const int*, volatile char*);

struct A {
   operator PF() const {
      return func;
   }

   operator PF1() {
      return func;
   }

   operator PF1() const  {
      return func;
   }

};

int main() {
   A a;
   int i;
   char c;

   a(&i, &c);   // C3066
   a(&i, (const char *) &c);   // OK
}
```

Compiler Error C3068

Error Message
'function' : a 'naked' function cannot contain objects that would require unwinding if a C++ exception occurred

The compiler was unable to perform stack unwinding on a naked function that threw an exception because a temporary object was created in the function and C++ exception handling (/EHsc) was specified.

To resolve this error, do at least one of the following:

- Do not compile with /EHsc.
- Do not mark the function as **naked**.
- Do not create a temporary object in the function.

If a function creates a temporary object on the stack, if the function throws an exception, and if C++ exception handling is enabled, the compiler will clean up the stack if an exception is thrown.

When an exception is thrown, compiler generated code, called the prolog and epilog and which are not present in a naked function, is executed for a function.

Example
The following sample generates C3068:

```
// C3068.cpp
// compile with: /EHsc
// processor: x86
class A {
public:
   A(){}
   ~A(){}
};

void b(A){}

__declspec(naked) void c() {
   b(A());   // C3068
};
```

Compiler Error C3069

Error Message

'operator': not allowed for enumeration type

An operator is not supported for CLR enumerations. For more information, see Using Operators and Enumerations.

Example

The following sample generates C3069:

```
// C3069.cpp
// compile with: /clr
enum struct E { e1 };
enum F { f1 };

int main() {
   E e = E::e1;
   bool tf;
   tf = !e;   // C3069

   // supported for native enums
   F f = f1;
   tf = !f;
}
```

Compiler Error C3070

Error Message
'property': property does not have a 'set' method

A property's set accessor method was not defined. For more information, see property.

The following sample generates C3070:

```
// C3070.cpp
// compile with: /clr
ref class R {
public:
   R(int size) {
      m_data = gcnew array<int>(size);
   }

   property int % MyProp[int] {
      int% get(int index) {
         return m_data[index];
      }
   }

   property int % MyProp2[int] {
      int% get(int index) {
         return m_data[index];
      }
      void set(int index, int % value) {}
   }

   property const int % MyProp3[int] {
      const int% get(int index) {
         return m_data[index];
      }
      void set(int index, const int % value) {}
   }

private:
   array<int>^ m_data;
};

int main() {
   R^ r = gcnew R(10);
   r->MyProp[4] = 4;   // C3070

   int value = 4;
   r->MyProp2[4] = value;   // OK
   r->MyProp3[4] = 4;   // OK
}
```

Compiler Error C3071

Error Message

operator 'operator' can only be applied to an instance of a ref class or a value-type

A CLR operator cannot be used on a native type.

For more information, see % (Tracking Reference).

Example

The following sample generates C3071.

```
// C3071.cpp
// compile with: /clr
class N {};
ref struct R {};

int main() {
   N n;
   %n;   // C3071

   R r;
   R ^ r2 = %r;   // OK
}
```

Compiler Error C3072

Error Message

operator 'operator' cannot be applied to an instance of a ref class

use the unary '*operator* ' operator to convert an instance of a ref class to a handle type

A CLR type requires CLR operators, not native (or standard) operators. For more information, see % (Tracking Reference).

Example

The following sample generates C3072.

```
// C3072.cpp
// compile with: /clr
ref class R {};

int main() {
   R r1;
   R^ r2 = &r1;   // C3072
   R^ r3 = %r1;   // OK
}
```

Compiler Error C3073

Error Message
'type' : ref class does not have a user-defined copy constructor

In a /clr (Common Language Runtime Compilation) compilation, the compiler will not generate a copy constructor for a reference type. In any **/clr** compilation, you must define your own copy constructor for a reference type if you expect an instance of the type to be copied.

For more information, see C++ Stack Semantics for Reference Types.

Example
The following sample generates C3073.

```
// C3073.cpp
// compile with: /clr
ref class R {
public:
   R(int) {}
};

ref class S {
public:
   S(int) {}
   S(const S %rhs) {}   // copy constructor
};

void f(R) {}
void f2(S) {}
void f3(R%){}

int main() {
   R r(1);
   f(r);   // C3073
   f3(r);   // OK

   S s(1);
   f2(s);   // OK
}
```

Compiler Error C3075

Error Message

'instance' : you cannot embed an instance of a reference type, 'type', in a value-type

A value type cannot contain an instance of a reference type.

For more information, see C++ Stack Semantics for Reference Types.

Example

The following sample generates C3075.

```
// C3075.cpp
// compile with: /clr /c
ref struct U {};
value struct X {
   U y;   // C3075
};

ref struct Y {
   U y;   // OK
};
```

Compiler Error C3076

Error Message

'instance' : you cannot embed an instance of a reference type, 'type', in a native type

A native type cannot contain an instance of a CLR type.

For more information, see C++ Stack Semantics for Reference Types.

Example

The following sample generates C3076.

```
// C3076.cpp
// compile with: /clr /c
ref struct U {};

struct V {
   U y;   // C3076
};

ref struct W {
   U y;   // OK
};
```

Compiler Error C3077

Error Message
'finalizer' : a finalizer can only be a member of a reference type

You cannot declare a finalizer in a native or value type.

For more information, see Destructors and Finalizers in Visual C++.

Example
The following sample generates C3077.

```
// C3077.cpp
// compile with: /clr /c
value struct vs {
   !vs(){}   // C3077
};

ref struct rs {
protected:
   !rs(){}   // OK
};
```

Compiler Error C3080

Error Message

'finalizer_function' : a finalizer cannot have a storage-class-specifier

For more information, see Destructors and Finalizers in Visual C++.

Example

The following sample generates C3080.

```
// C3080.cpp
// compile with: /clr /c
ref struct rs {
protected:
   static !rs(){}   // C3080
   !rs(){}   // OK
};
```

Compiler Error C3082

Error Message

'operator' : this operator cannot be applied to a variable whose type is generic

Certain operations are not allowed on generic types. For example, you cannot dereference a generic type parameter.

For more information, see Generics (Visual C++).

Example

The following sample generates C3082.

```
// C3082.cpp
// compile with: /clr
ref class R {};

generic<typename T>
where T : R
ref class GR {
public:
   void mf(T t) {
      R r = *t;   // C3082
   }
};
```

Compiler Error C3083

Error Message

'function': the symbol to the left of a '::' must be a type

A function was called incorrectly.

Example

The following sample generates C3083.

```
// C3083.cpp
// compile with: /c
struct N {
   ~N();
};

struct N1 {
   ~N1();
};

N::N::~N() {}   // C3083
N1::~N1() {}   // OK
```

Compiler Error C3084

Error Message
'function': a finalizer/destructor cannot be 'keyword'

A finalizer or destructor was declared incorrectly.

For example, a destructor should not be marked as sealed. The destructor will be inaccessible to derived types. For more information, see Explicit Overrides and Destructors and Finalizers in Visual C++.

Example
The following sample generates C3084.

```
// C3084.cpp
// compile with: /clr /c
ref struct R {
protected:
   !R() sealed;   // C3084
   !R() abstract;   // C3084
   !R();
};
```

Compiler Error C3085

Error Message
'constructor': a constructor cannot be 'keyword'

A constructor was declared incorrectly. See Override Specifiers for more information.

Example
The following sample generates C3085.

```
// C3085.cpp
// compile with: /c /clr
ref struct S {
   S() abstract;   // C3085
   S(S%) abstract;   // C3085
};

ref struct T {
   T() sealed {}   // C3085
   T(T%) sealed {}   // C3085
};
```

Compiler Error C3087

Error Message
'named_argument': call of 'attribute' already initializes this member

A named argument was specified in the same attribute block as an unnamed argument for the same value. Specify only a named or unnamed argument.

Example
The following sample generates C3087.

```
// C3087.cpp
// compile with: /c
[idl_quote("quote1", text="quote2")];   // C3087
[idl_quote(text="quote3")];   // OK
[idl_quote("quote4")];   // OK
```

Compiler Error C3094

Error Message

'attribute': anonymous usage not allowed

An attribute was not scoped correctly. For more information, see User-Defined Attributes.

Example

The following sample generates C3094.

```
// C3094.cpp
// compile with: /clr /c
using namespace System;
[AttributeUsage(AttributeTargets::Class)]
public ref class AAttribute : Attribute {};

[A];   // C3094

// OK
[A]
ref class x{};
```

Compiler Error C3095

Error Message
'attribute': attribute cannot be repeated

Some attributes are declared such that, multiple occurrences of the attribute cannot be applied to a target.

For more information, see User-Defined Attributes.

Example
The following sample generates C3095.

```
// C3095.cpp
// compile with: /clr /c
using namespace System;

[AttributeUsage(AttributeTargets::All, AllowMultiple=false)]
public ref class Attr : public Attribute {
public:
   Attr(int t) : m_t(t) {}
   const int m_t;
};

[AttributeUsage(AttributeTargets::All, AllowMultiple=true)]
public ref class Attr2 : public Attribute {
public:
   Attr2(int t) : m_t(t) {}
   const int m_t;
};

[Attr(10)]   // C3095
[Attr(11)]
ref class A {};

[Attr2(10)]   // OK
[Attr2(11)]
ref class B {};
```

Compiler Error C3096

Error Message
'attribute': attribute is allowed on data members of attribute classes only

An attribute was applied incorrectly.

For more information, see User-Defined Attributes.

Compiler Error C3097

Error Message
'attribute': attribute must be scoped with 'assembly:' or 'module:'

A global attribute was used incorrectly.

For more information, see User-Defined Attributes.

Example
The following sample generates C3097.

```
// C3097.cpp
// compile with: /clr /c
using namespace System;

[AttributeUsage(AttributeTargets::All, AllowMultiple = true)]
public ref class Attr : public Attribute {
public:
   Attr(int t) : m_t(t) {}
   int m_t;
};


[Attr(10)];   // C3097
[assembly:Attr(10)];   // OK

[Attr(10)]   // OK
public ref class MyClass {};
```

Compiler Error C3099

Error Message

'keyword': use [System::AttributeUsageAttribute] for managed attributes

Use AttributeUsageAttribute to declare **/clr** attributes.

For more information, see User-Defined Attributes.

Example

The following sample generates C3099.

```
// C3099.cpp
// compile with: /clr /c
using namespace System;
[usage(10)]   // C3099
// try the following line instead
// [AttributeUsageAttribute(AttributeTargets::All)]
ref class A : Attribute {};
```

Compiler Error C3100

Error Message

'target' : unknown attribute qualifier

An invalid attribute target was specified.

For more information, see User-Defined Attributes.

Example

The following sample generates C3100.

```
// C3100.cpp
// compile with: /clr /c
using namespace System;
[AttributeUsage(AttributeTargets::All)]
public ref class Attr : public Attribute {
public:
   Attr(int t) : m_t(t) {}
   int m_t;
};

[invalid_target:Attr(10)];   // C3100
[assembly:Attr(10)];   // OK
```

Compiler Error C3101

Error Message
illegal expression for named attribute argument 'field'

When initializing a named attribute argument, the value must be a compile time constant.

For more information on attributes, see User-Defined Attributes.

Example
The following sample generates C3101.

```
// C3101.cpp
// compile with: /clr /c
ref class AAttribute : System::Attribute {
public:
   int Field;
};

extern int i;

[assembly:A(Field = i)];   // C3101
[assembly:A(Field = 0)];   // OK
```

Compiler Error C3103

Error Message
'argument': repeated named argument

An attribute can not repeat named arguments.

For more information, see User-Defined Attributes.

Example
The following sample generates C3103.

```
// C3103.cpp
// compile with: /clr /c
using namespace System;

[AttributeUsage(AttributeTargets::All)]
public ref class Attr : public Attribute {
public:
   int m_t;
};

[Attr(m_t = 10, m_t = 1)]   // C3103
// try the following line instead
// [Attr(m_t = 10)]
ref class A {};
```

Compiler Error C3104

Error Message
illegal attribute argument

You specified an invalid argument to an attribute.

See Attribute Parameter Types for more information.

This error can be generated as a result of compiler conformance work that was done for Visual C++ 2005: when passing managed arrays to custom attributes, the type of the array is no longer deduced from the aggregate initialization list. The compiler now requires you to specify the type of the array as well as the initializer list. See Breaking Changes in the Visual C++ 2005 Compiler for more information.

Example

The following sample generates C3104.

```
// C3104a.cpp
// compile with: /clr /c
using namespace System;

[AttributeUsage(AttributeTargets::Class)]
public ref struct ABC : public Attribute {
   ABC(array<int>^){}
   array<double> ^ param;
};

[ABC( {1,2,3}, param = {2.71, 3.14})]   // C3104
// try the following line instead
// [ABC( gcnew array<int> {1,2,3}, param = gcnew array<double>{2.71, 3.14})]
ref struct AStruct{};
```

The following sample generates C3104.

```
// C3104b.cpp
// compile with: /clr /c
// C3104 expected
using namespace System;

int func() {
   return 0;
}

[attribute(All)]
ref class A {
public:
   A(int) {}
};

// Delete the following 2 lines to resolve.
[A(func())]
ref class B {};

// OK
[A(0)]
ref class B {};
```

The following sample generates C3104.

```
// C3104c.cpp
// compile with: /clr:oldSyntax /c
using namespace System;
```

```
[ attribute(Class) ]
public __gc class AnotherAttr {
public:
   AnotherAttr(Object* arr __gc[]) : var0(arr) {}
   Object* var1 __gc[];
   Object* var0 __gc[];
};

[ AnotherAttr( { __box(3.14159), S"pi" }, var1 = { S"a", S"b" } ) ]   // C3104
public __gc class Class1 {};

// OK
[ AnotherAttr( new Object * __gc[] {__box(3.14159), S"pi" }, var1 = new Object * __gc[] { S
"a", S"b" } ) ]
public __gc class Class2 {};
```

Compiler Error C3106

Error Message

'attribute': unnamed arguments must precede named arguments

Unnamed arguments must be passed to an attribute before named arguments.

For more information, see User-Defined Attributes.

Example

The following sample generates C3106.

```
// C3106.cpp
// compile with: /c
[module(name="MyLib", dll)];   // C3106
[module(dll, name="MyLib")];   // OK
```

Compiler Error C3110

Error Message
'function_name' : you cannot overload a COM interface method

An interface that is prefaced by an interface attribute, such as:

- custom
- dispinterface
- dual
- object

cannot be overloaded. For example:

```
// C3110.cpp
#include <unknwn.h>
[ object, uuid= "4F98A180-EF37-11D1-978D-0000F805D73B" ]
__interface ITestInterface
{
   HRESULT mf1(void);
   HRESULT mf1(BSTR); // C3110
};

int main()
{
}
```

Compiler Error C3113

Error Message
an 'structure' cannot be a template/generic

You attempted to make a class template or class generic out of an interface or an enum.

The following sample generates C3113:

```
// C3113.cpp
// compile with: /c
template <class T>
enum E {};   // C3113
// try the following line instead
// class MyClass{};
```

Compiler Error C3114

Error Message
'argument': not a valid named attribute argument

In order for an attribute class data member to be a valid named argument, it must not be marked **static**, **const**, or **literal**. If a property, the property must not be **static** and must have get and set accessors.

For more information, see property and User-Defined Attributes.

Example
The following sample generates C3114.

```
// C3114.cpp
// compile with: /clr /c
public ref class A : System::Attribute {
public:
   static property int StaticProp {
      int get();
   }

   property int Prop2 {
      int get();
      void set(int i);
   }
};

[A(StaticProp=123)]   // C3114
public ref class R {};

[A(Prop2=123)]   // OK
public ref class S {};
```

Compiler Error C3115

Error Message

'attribute': this attribute is not allowed on 'construct'

An attribute was applied to a construct for which it was not intended. See Attributes by Usage for more information.

Example

The following sample generates C3115.

```
// C3115.cpp
// compile with: /c
#include <unknwn.h>
[module(name="xx")];

[object, helpstringdll(xx.dll), uuid("00000000-0000-0000-0000-000000000001")]   // C3115
// try the following line instead
// [object, uuid("00000000-0000-0000-0000-000000000001")]
__interface IMyI {
   HRESULT xx();
};
```

Compiler Error C3116

Error Message
'storage specifier' : invalid storage class for interface method

You used **typedef**, **register**, or **static** as the storage class for an interface method. These storage classes are not permitted on interface members.

The following sample generates C3116:

```
// C3116.cpp
__interface ImyInterface
{
   static void myFunc();   // C3116
};
```

Compiler Error C3117

Error Message
'%$S' : an interface can only have one base class

You declared an interface that inherits from multiple base classes.

The following sample generates C3117:

```
// C3117.cpp
#include <windows.h>

[ object, uuid("00000000-0000-0000-0000-000000000001") ]
__interface I1
{
};

[ object, uuid("00000000-0000-0000-0000-000000000002") ]
__interface I2
{
};

[ object, uuid("00000000-0000-0000-0000-000000000003") ]
__interface I3 : I1, I2
{   // C3117
};
```

Compiler Error C3118

Error Message
'interface' : interfaces do not support virtual inheritance

You tried to virtually inherit from an interface. For example,

```
// C3118.cpp
__interface I1 {
};

__interface I2 : virtual I1 {   // C3118
};
```

generates this error.

Compiler Error C3120

Error Message
'method_name' : interface methods cannot take a variable argument list

An interface method cannot take a variable argument list. For example, the following interface definition generates C3120:

```
// C3120.cpp
__interface A {
int X(int i, ...);    // C3120
};

int main(void) { return(0); }
```

Compiler Error C3121

Error Message
cannot change GUID for class 'class_name'

You attempted to change the class ID with __declspec(uuid).

For example, the following code generates C3121:

```
// C3121.cpp
[emitidl];
[module(name="MyLibrary")];

[coclass, uuid="00000000-0000-0000-0000-111111111111"]
class __declspec(uuid("00000000-0000-0000-0000-111111111112")) A   // C3121
{
};
int main()
{
}
```

Compiler Error C3126

Error Message
cannot define a union 'union' inside of managed type 'type'

A union cannot be defined inside a managed type.

The following sample generates C3126:

```
// C3126_2.cpp
// compile with: /clr /c
ref class Test
{
   union x
   {   // C3126
      int a;
      int b;
   };
};
```

The following sample generates C3126:

```
// C3126.cpp
// compile with: /clr:oldSyntax /c
#using <mscorlib.dll>

__gc class Test
{
   union x
   {   // C3126
      int a;
      int b;
   };
};
```

Compiler Error C3130

Error Message
Internal Compiler Error: failed to write injected code block to PDB

This error occurs if the compiler failed to write an injected code block to the .pdb file. The most common reason for the failure is lack of disk space.

Compiler Error C3131

Error Message
project must have a 'module' attribute with a 'name' property

The module attribute must have a name parameter.

The following sample generates C3131:

```
// C3131.cpp
[emitidl];
[module];   // C3131
// try the following line instead
// [module (name="MyLib")];

[public]
typedef long int LongInt;
```

Compiler Error C3132

Error Message

'function-parameter' : parameter arrays can only be applied to a formal argument of type 'single-dimensional managed array'

The ParamArray attribute was applied to a parameter that was not a single-dimension array.

The following sample generates C3132:

```
// C3132.cpp
// compile with: /clr /c
using namespace System;
void f( [ParamArray] Int32[,] );   // C3132
void g( [ParamArray] Int32[] );   // C3132

void h( [ParamArray] array<Char ^> ^ MyArray );   // OK
```

Compiler Error C3133

Error Message
Attributes cannot be applied to C++ varargs

An attribute was applied incorrectly. Attributes can not be applied to an ellipsis representing variable arguments.

For more information, see User-Defined Attributes.

Example
The following sample generates C3133.

```
// C3133.cpp
// compile with: /clr /c
ref struct MyAttr: System::Attribute {};
void Func([MyAttr]...);   // C3133
void Func2([MyAttr] int i);   // OK
```

Compiler Error C3134

Error Message

'value' : value of attribute argument 'attribute' does not have valid type 'type'

A syntax error was detected when a value was assigned to an attribute.

See Also

Other Resources

Attributes by Usage

Compiler Error C3135

Error Message

'property' : a property cannot have a 'const' or 'volatile' type

The const and volatile keywords are not permitted on properties.

Compiler Error C3136

Error Message
'interface' : a COM interface can only inherit from another COM interface, 'interface' is not a COM interface

An interface to which you applied an interface attribute inherits from an interface that is not a COM interface. A COM interface ultimately inherits from `IUnknown`. Any interface preceded by an interface attribute is a COM interface.

The following example generates C3136:

```
// C3136.cpp
#include "unknwn.h"

__interface A   // C3136
// try the following line instead
// _interface A : IUnknown
{
   int a();
};

[object]
__interface B : A
{
   int aa();
};
```

Compiler Error C3137

Error Message
'property' : a property cannot be initialized

A property cannot be initialized, for example, in a constructor's initialization list.

The following example generates C3137:

```
// C3137.cpp
// compile with: /clr /c
ref class CMyClass {
public:
   property int Size {
      int get() {
         return 0;
      }
      void set( int i ) {}
   }

   CMyClass() : Size( 1 ) {   // C3137
      // to resolve this C3137, remove the initializer from the
      // ctor declaration and perform the assignment as follows
      // Size = 1;
   }
};
```

The following example generates C3137:

```
// C3137_2.cpp
// compile with: /clr:oldSyntax /c
__gc class CMyClass {
public:
   __property int get_Size() {
      return 0;
   }
   __property void set_Size(int i) {}

   CMyClass() : Size(1) {   // C3137
      // to resolve this C3137, remove the initializer from the
      // ctor declaration and perform the assignment as follows
      // Size = 1;
   }
};
```

Compiler Error C3138

Error Message
'interface' : a 'attribute' interface must inherit from IDispatch, or from an interface that inherits from IDispatch

An interface with the dual or dispinterface attributes does not have **IDispatch** as a direct or indirect base interface.

The following example generates C3138:

```
// C3138.cpp
#include <unknwn.h>

[ object, uuid("77ac9240-6e9a-11d2-97de-0000f805d73b") ]
__interface IMyCustomInterface
{
   HRESULT mf1(void);
};

[ dispinterface, uuid("3536f8a0-6e9a-11d2-97de-0000f805d73b") ]
__interface IMyDispInterface : IUnknown
{
   [id(1)] HRESULT mf2(void);
};

[ object, dual, uuid("34e90a10-6e9a-11d2-97de-0000f805d73b") ]
__interface IMyDualInterface : IMyCustomInterface  // C3138 expected
{
   HRESULT mf3(void);
};
```

Compiler Error C3139

Error Message
'struct' : cannot export a UDT without members

You attempted to apply the export attribute to an empty UDT (user-defined type). For example:

```
// C3139.cpp
#include "unknwn.h"
[emitidl];
[module(name=xx)];

[export] struct MyStruct {   // C3139 empty type
};
int main(){}
```

Compiler Error C3140

Error Message
cannot have multiple 'module' attributes in the same compilation unit

The module attribute can only be defined once per project.

The following sample generates C3140:

```
// C3140.cpp
// compile with: /c
[emitidl];
[module(name = "MyLibrary")];
[module(name = "MyLibrary2")];   // C3140
```

Compiler Error C3141

Error Message
'interface_name' : interfaces only support public inheritance

Interfaces defined with the interface (or __interface) keyword only support public inheritance.

The following sample generates C3141:

```
// C3141.cpp
__interface IBase {};
__interface IDerived1 : protected IBase {};  // C3141
__interface IDerived2 : private IBase {};    // C3141
```

Compiler Error C3142

Error Message
'property_name' : you cannot take the address of a property

The address of a property is not available to the developer.

The following sample generates C3142:

```
// C3142_2.cpp
// compile with: /clr
using namespace System;
ref class CSize {
private:
   property int Size {
      int get();
   }
};

int main() {
    &CSize::Size; // C3142
}
```

The following sample generates C3142:

```
// C3142.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;

__gc class CSize
{
   __property int get_Size();
};

int main()
{
   &CSize::Size;   // C3142
}
```

Compiler Error C3145

Error Message
'object' : global or static variable may not have managed type 'type'

You can only define CLR objects within function scope.

The following sample generates C3145:

```
// C3145.cpp
// compile with: /clr
using namespace System;
ref class G {};

G ^ ptr;   // C3145
G ^ ptr2 = gcnew G;   // C3145

ref class GlobalObjects {
public:
   static G ^ ptr;   // OK
   static G ^ ptr2 = gcnew G;   // OK
};

int main() {
   G ^ ptr;   // OK
   G ^ ptr2 = gcnew G;   // OK
}
```

The following sample generates C3145:

```
// C3145b.cpp
// compile with: /clr
ref class MyClass {
public:
   static int data;
};

interior_ptr<int> p = &(MyClass::data);   // C3145

void Test(interior_ptr<int> x) {}

int main() {
   MyClass ^ h_MyClass = gcnew MyClass;
   interior_ptr<int> p = &(h_MyClass->data);
}
```

Managed Extensions for C++

The following sample generates C3145:

```
// C3145c.cpp
// compile with: /clr:oldSyntax
using namespace System;
__gc class G {};

G *ptr;   // C3145
G *ptr2 = new G;   // C3145

__gc class GlobalObjects {
public:
   static G *ptr;   // OK
   static G *ptr2 = new G;   // OK
};
```

```
int main() {
   G *ptr;   // OK
   G *ptr2 = new G;   // OK
}
```

Compiler Error C3146

Error Message
illegal __nogc reference to managed type 'type'

An attempt was made to create an unmanaged reference to a variable of a managed type.

C3146 is only reachable using **/clr:oldSyntax**.

The following sample generates C3146:

```
// C3146.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
int main() {
   System::String *str1 = new System::String("test");
   System::String __nogc& str2 = *str1;  // C3146
   System::String & str3 = *str1;     // OK equivalent to __gc& str2
}
```

Compiler Error C3147

Error Message
illegal __nogc with multi-dimensional __gc array

A __gc array, which is indicated with a comma inside the array brackets, was also specified to be an unmanaged array with the __nogc keyword. Change the declaration so that the array is unambiguously declared as either managed or unmanaged.

C3147 is only reachable using **/clr:oldSyntax**.

The following sample generates C3147:

```
// C3147.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;
int main()
{
   String* x __nogc[,];   // C3147, remove __nogc
}
```

Compiler Error C3148

Error Message
illegal __nogc pointer to managed type 'type'

A pointer to a managed type cannot be declared with the __nogc keyword.

C3148 is only reachable using **/clr:oldSyntax**.

The following sample generates C3148:

```
// C3148.cpp
// compile with: /clr:oldSyntax
#using<mscorlib.dll>
__gc class G {};

int main() {
   G __nogc *pg;   // C3148
   G *pg2;   // OK
   G __gc *pg3;
}
```

Compiler Error C3149

Error Message
'type' : cannot use this type here without a top-level 'char'

A declaration was not specified correctly.

For example, you may have defined a CLR type at global scope and tried to create a variable of the type as part of the definition. Because global variables of CLR types are not allowed, the compiler will generate C3149.

To resolve this error, declare variables of CLR types inside a function or type definition.

The following sample generates C3149:

```
// C3149.cpp
// compile with: /clr
using namespace System;
int main() {
   // declare an array of value types
   array< Int32 ^> IntArray;   // C3149
   array< Int32>^ IntArray2;   // OK
}
```

The following sample generates C3149:

```
// C3149b.cpp
// compile with: /clr /c
delegate int MyDelegate(const int, int);
void Test1(MyDelegate m) {}   // C3149
void Test2(MyDelegate ^ m) {}   // OK
```

Managed Extensions for C++

A managed object was not used correctly.

The following sample generates C3149:

```
// C3149c.cpp
// compile with: /clr:oldSyntax
__gc class A {};

int main() {
   A a = new A;   // C3149
   A *a = new A;   // OK
}
```

Compiler Error C3150

Error Message
'element' : 'attribute' can only be applied to a class, interface, array or pointer

__gc can only be used on a class, interface, or array.

C3150 is only reachable using **/clr:oldSyntax**.

The following sample generates C3150:

```
// C3150.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc void f()   // C3150; function cannot be managed
{
}
```

Compiler Error C3151

Error Message
'type' cannot apply __gc or __value to a template

A language keyword was applied to a template class, which is not allowed.

C3151 is only reachable using **/clr:oldSyntax**.

The following sample generates C3151:

```
// C3151.cpp
// compile with: /clr:oldSyntax /c
template <class T> __gc class X {};   // C3151
template <class T> class Y {};   // OK
```

Compiler Error C3152

Error Message
'construct' : 'keyword' can only be applied to a class, struct, or member function

Certain keywords can only be applied to a C++ class.

The following sample generates C3152:

```
// C3152.cpp
// compile with: /clr /c
ref class C {
   int (*pfn)() sealed;   // C3152
   virtual int g() sealed;   // OK
};
```

The following sample generates C3152:

```
// C3152_2.cpp
// compile with: /clr:oldSyntax /c
__value __interface A {};   // C3152;
__value class X {};   // OK
```

Compiler Error C3153

Error Message
'interface' : you cannot create an instance of an interface

An interface cannot be instantiated. To use the members of an interface, derive a class from the interface, implement the interface members, and then use the members.

The following sample generates C3153:

```
// C3153.cpp
// compile with: /clr
interface class A {
};

int main() {
   A^ a = gcnew A;   // C3153
}
```

The following sample generates C3153:

```
// C3153b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
__interface A {
};

int main() {
   A *a = new A;   // C3153
}
```

Compiler Error C3154

Error Message
Expected ',' before ellipsis. Non-comma separated ellipsis not supported on parameter array functions.

A variable argument function was not declared correctly.

For more information, see How to: Accept Variable Arguments.

Example
The following sample generates C3154.

```
// C3154.cpp
// compile with: /clr
ref struct R {
   void Func(int ... array<int> ^);   // C3154
   void Func2(int i, ... array<int> ^){}   // OK
   void Func3(array<int> ^){}   // OK
   void Func4(... array<int> ^){}   // OK
};

int main() {
   R ^ r = gcnew R;
   r->Func4(1,2,3);
}
```

Compiler Error C3155

Error Message
attributes are not allowed in a property indexer

An indexed property was declared incorrectly. For more information, see How to: Use Indexed Properties.

Example
The following sample generates C3155.

```
// C3155.cpp
// compile with: /clr /c
using namespace System;
ref struct R {
   property int F[[ParamArray] int] {   // C3155
   // try the following line instead
   // property int F[ int] {   // OK
      int get(int i) {
         return 0;
      }
   }
};
```

Compiler Error C3156

Error Message

'class' : you cannot have a local definition of a managed type

A function cannot contain the definition, or declaration, of a managed class, struct, or interface.

Example

The following sample generates C3156.

```
// C3156.cpp
// compile with: /clr /c
void f() {
   ref class X {};   // C3156
   ref class Y;   // C3156
}
```

The following sample generates C3156.

```
// C3156_b.cpp
// compile with: /clr:oldSyntax /c
void f() {
   __gc class X {};   // C3156
   __gc class Y;   // C3156
}
```

Compiler Error C3157

Error Message

ParamArray attribute can only be applied to the last parameter

The ParamArray attribute was applied to the wrong parameter.

Compiler Error C3159

Error Message
'pointer' : array of pointers to value type cannot be declared

An array of pointers to a value type cannot be declared.

C3159 is only reachable using **/clr:oldSyntax**.

The following sample generates C3159:

```
// C3159.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__value struct B {
};

void f( B*[] );   // C3159

int main() {
}
```

Compiler Error C3160

Error Message
'pointer' : a data member of a managed class cannot have this type

Interior garbage collection pointers may point to the interior of a managed class. Because they are slower than whole-object pointers and require special handling by the garbage collector, you cannot declare interior managed pointers as members of a class.

The following sample generates C3160:

```
// C3160.cpp
// compile with: /clr
ref struct A {
   // cannot create interior pointers inside a class
   interior_ptr<int> pg;   // C3160
   int g;   // OK
   int* pg2;   // OK
};

int main() {
   interior_ptr<int> pg2;   // OK
}
```

Managed Extensions for C++

The following sample generates C3160:

```
// C3160b.cpp
// compile with: /clr:oldSyntax

__gc struct A {
   // cannot create interior pointers inside a class
   int __gc* pg; // C3160
   int g;   // OK
   int __nogc *pg2;   // OK
};

int main() {
   int __gc* pg2;   // OK
}
```

Compiler Error C3161

Error Message

'interface' : nesting class, struct, union or interface in an interface is illegal; nesting interface in a class, struct or union is illegal

An __interface can only appear at global scope or within a namespace. A class, struct, or union cannot appear in an interface.

Example

The following sample generates C3161.

```
// C3161.cpp
// compile with: /c
__interface X {
   __interface Y {};   // C3161 A nested interface
};
```

Compiler Error C3162

Error Message
'type' : a reference type which has a destructor cannot be used as the type of static data member 'member'

The common language runtime cannot know when to run a user-defined destructor when the class also contains static member function.

A destructor will never be run unless the object is deleted explicitly.

For more information, see,

- /clr (Common Language Runtime Compilation)
- Common Visual C++ 64-bit Migration Issues

Example
The following sample generates C3162.

```
// C3162.cpp
// compile with: /clr /c
ref struct A {
   ~A() { System::Console::WriteLine("in destructor"); }
   static A i;   // C3162
   static A^ a = gcnew A;   // OK
};

int main() {
   A ^ a = gcnew A;
   delete a;
}
```

Compiler Error C3163

Error Message
'construct': attributes inconsistent with previous declaration

The attribute(s) applied to a definition conflict with the attribute(s) applied to a declaration.

One way to resolve C3163 is to not to have any attributes on the forward declaration; attributes on a forward declaration should be at most equal to or less than the attributes on the definition.

Example
The following sample generates C3163.

```
// C3163.cpp
// compile with: /clr /c
using namespace System;

[CLSCompliant(true)] void f();
[CLSCompliant(false)] void f() {}   // C3163
// try the following line instead
// [CLSCompliant(true)] void f() {}
```

Compiler Error C3164

Error Message
'member': Copy constructors and destructors on value types are not supported

Copy constructors and destructors are not supported on __value types.

C3164 is only reachable using **/clr:oldSyntax**.

The following sample generates C3164:

```
// C3164.cpp
// compile with: /clr:oldSyntax /c
__value class MyClass {
public:
   ~MyClass();   // C3164
   MyClass(const MyClass& f);   // C3164
};
```

Compiler Error C3166

Error Message
'pointer' : cannot declare a pointer to an interior __gc pointer as a member of 'type'

The compiler found an invalid pointer declaration (a __nogc pointer to a __gc pointer.). This syntax may be supported in a future release.

C3166 is only reachable using **/clr:oldSyntax**.

The following sample generates C3166:

```
// C3166.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;

__gc struct G {
   int __gc* __nogc* p;   // C3166
};

public __gc class H {
public:
   Int32 __gc* __nogc* p;   // C3166
};

public __value struct I {
   int __gc* __nogc* p;   // C3166
};

public __value class J {
public:
   int __gc* __nogc* p;   // C3166
};

int main() {
   G* pG = new G;
   H* pH = new H;
}
```

Compiler Error C3167

Error Message

Unable to initialize .NET Framework: make sure it is installed

The .NET Framework is not installed on this computer; install the .NET Framework.

Compiler Error C3168

Error Message
'type' : illegal underlying type for enum

The underlying type you specified for the **enum** type was not valid. The underlying type must be an integral C++ type or a corresponding CLR type.

The following sample generates C3168:

```
// C3168.cpp
// compile with: /clr /c
ref class G{};

enum class E : G { e };   // C3168
enum class F { f };   // OK
```

The following sample generates C3168:

```
// C3168_2.cpp
// compile with: /clr:oldSyntax /c
__gc class G {};

__value enum E : G {e};   // C3168
__value enum F {f};   // OK
```

Compiler Error C3170

Error Message
cannot have different module identifiers in a project

module attributes with different names were found in two of the files in a compilation. Only one unique **module** attribute can be specified per compilation.

Identical **module** attributes can be specified in more than one source code file.

For example, if the following module attributes were found:

```
// C3170.cpp
[ module(name="MyModule", uuid="373a1a4e-469b-11d3-a6b0-00c04f79ae8f") ];
int main() {}
```

And then,

```
// C3170b.cpp
// compile with: C3170.cpp
// C3170 expected
[ module(name="MyModule1", uuid="373a1a4e-469b-11d3-a6b0-00c04f79ae8f") ];
```

the compiler would generate C3170 (note the different names).

Compiler Error C3171

Error Message
'module': cannot specify different module attributes in a project

module attributes with different parameter lists were found in two of the files in a compilation. Only one unique **module** attribute can be specified per compilation.

Identical **module** attributes can be specified in more than one source code file.

For example, if the following **module** attributes were found:

```
// C3171.cpp
[ module(name="MyModule", uuid="373a1a4e-469b-11d3-a6b0-00c04f79ae8f", version="1.0") ];
int main() {}
```

And then,

```
// C3171b.cpp
// compile with: C3171.cpp
// C3171 expected
[ module(name="MyModule", uuid="373a1a4e-469b-11d3-a6b0-00c04f79ae8f", version="1.1") ];
```

the compiler would generate C3171 (note the different version values).

Compiler Error C3172

Error Message
'module_name': cannot specify different idl_module attributes in a project

idl_module attributes with the same name but different **dllname** or **version** parameters were found in two of the files in a compilation. Only one unique **idl_module** attribute can be specified per compilation.

Identical **idl_module** attributes can be specified in more than one source code file.

For example, if the following **idl_module** attributes were found:

```
// C3172.cpp
[module(name="MyMod")];
[ idl_module(name="x", dllname="file.dll", version="1.1") ];
int main() {}
```

And then,

```
// C3172b.cpp
// compile with: C3172.cpp
// C3172 expected
[ idl_module(name="x", dllname="file.dll", version="1.0") ];
```

the compiler would generate C3172 (note the different version values).

Compiler Error C3173

Error Message
version mismatch in idl merge

This error occurs when an object file contains embedded idl that was generated with a previous version of the compiler. The compiler encodes a version number to ensure that the same compiler used to generate the idl content that is embedded in the .obj files is also the same compiler used to merge the embedded idl.

Update your Visual C++ installation so that all tools are from the latest released version.

Compiler Error C3174

Error Message
module attribute was not specified

A program that uses Visual C++ attributes did not also use the module attribute, which is required in any program that uses attributes.

The following sample generates C3174:

```
// C3174.cpp
// C3174 expected
// uncomment the following line to resolve this C3174
// [module(name="x")];
[export]
struct S
{
   int i;
};

int main()
{
}
```

Compiler Error C3175

Error Message
'function1' : cannot call a method of a managed type from unmanaged function 'function2'

Unmanaged functions cannot call member functions of managed classes.

The following sample generates C3175:

```
// C3175_2.cpp
// compile with: /clr

ref struct A {
   static void func() {
   }
};

#pragma unmanaged   // remove this line to resolve

void func2() {
   A::func();   // C3175
}

#pragma managed

int main() {
   A ^a = gcnew A;
   func2();
}
```

The following sample generates C3175:

```
// C3175.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc struct A
{
   static void func()
   {
   }
};

#pragma unmanaged   // remove this line to resolve

void func2()
{
   A::func();   // C3175
}

#pragma managed

int main()
{
   A *a = new A;
   func2();
}
```

Compiler Error C3176

Error Message

'type' : cannot declare local value type

A class can only be declared as a value type at global scope.

Example

The following sample generates C3176.

```
// C3176.cpp
// compile with: /clr
int main () {
   enum class C {};   // C3176
}
```

Compiler Error C3179

Error Message
an unnamed managed type is not allowed

All CLR classes and structs must have names.

The following sample generates C3179:

```
// C3179a.cpp
// compile with: /clr /c
typedef value struct { // C3179
// try the following line instead
// typedef value struct MyStruct {
   int i;
} V;
```

The following sample generates C3179:

```
// C3179b.cpp
// compile with: /clr:oldSyntax /c
#using <mscorlib.dll>
typedef __value struct {   // C3179
// try the following line instead
// typedef __value struct MyStruct {
   int i;
} V;
```

Compiler Error C3180

Error Message
'type name' : name exceeds meta-data limit of 'limit' characters

The compiler truncated the name for a managed type in metadata. The truncation will make the type unusable with the **#using** directive (or the equivalent in another language).

The type-name limit includes any namespace qualifications.

Compiler Error C3181

Error Message
'type' : invalid operand for operator

An invalid parameter was passed to the __typeof or typeid operator. The parameter must be a managed type.

Note that the compiler uses aliases for native types that map to types in the common language runtime.

The following sample generates C3181:

```
// C3181a.cpp
// compile with: /clr
using namespace System;

int main() {
   Type ^pType1 = interior_ptr<int>::typeid;   // C3181
   Type ^pType2 = int::typeid;   // OK
}
```

The following sample generates C3181:

```
// C3181b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;

int main() {
   Type *pType1 = __typeof(int __gc*);   // C3181
   Type *pType2 = __typeof(int*);   // OK
}
```

Compiler Error C3182

Error Message

'class' : a member using-declaration or access declaration is illegal within a managed type

A using declaration is invalid within all forms of managed classes.

The following sample generates C3182.

```
// C3182a.cpp
// compile with: /clr /c
ref struct B {
   void mf(int) {
   }
};

ref struct D : B {
   using B::mf;   // C3182, delete to resolve
   void mf(char) {
   }
};
```

The following sample generates C3182.

```
// C3182b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc struct B {
   void mf(int)
   {
   }
};

__gc struct D : B {
   using B::mf;   // C3182, delete to resolve
   void mf(char)
   {
   }
};

int main()
{
}
```

Compiler Error C3183

Error Message
cannot define unnamed class, struct or union inside of managed type 'type'

A type that is embedded in a managed type must be named.

The following sample generates C3183:

```
// C3183a.cpp
// compile with: /clr /c
ref class Test
{
   ref class
   {  // C3183, delete class or name it
      int a;
      int b;
   };
};
```

The following sample generates C3183:

```
// C3183b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc class Test
{
   __gc class
   {  // C3183, delete class or name it
      int a;
      int b;
   };
};

int main()
{
}
```

Compiler Error C3185

Error Message

'typeid' used on managed type 'type', use 'operator' instead

You cannot apply the typeid operator to a managed type; use typeid instead.

The following sample generates C3185:

```
// C3185a.cpp
// compile with: /clr
ref class Base {};
ref class Derived : public Base {};

int main() {
   Derived ^ pd = gcnew Derived;
   Base ^pb = pd;
   const type_info & t1 = typeid(pb);   // C3185
   System::Type ^ MyType = Base::typeid;   // OK
};
```

Managed Extensions for C++

You cannot apply typeid to a managed type; use __typeof instead.

The following sample generates C3185:

```
// C3185b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
__gc class Base {};
__gc class Derived : public Base {};

int main() {
   Derived *pd = new Derived;
   Base *pb = pd;
   const type_info & t1 = typeid(*pb);   // C3185

   // OK
   Type * t = __typeof(Base);
   Type * t1 = __typeof(Derived);
};
```

Compiler Error C3186

Error Message
multi-dimensional native array is not allowed

A native multidimensional array was declared incorrectly.

The following sample generates C3186:

```
// C3186.cpp
// compile with: /clr
#using <mscorlib.dll>
int main()
{
   int a[,];   // C3186
   int b[2][3];   // native multidimension array
   array<int,2> ^c = new array<int, 2>(2,5);   // managed multidimension array
}
```

Compiler Error C3188

Error Message

the brace-enclosed form of new-initializer is not supported under /clr:oldSyntax

Example

C3188 is only reachable using **/clr:oldSyntax**.

The following sample generates C3188.

```
// C3188.cpp
// compile with: /clr:oldSyntax
using namespace System;
int main() {
   __gc new Object { 0, 0, 0, 0 };   // C3188
   __gc new Object*[4] { 0, 0, 0, 0 };   // OK
}
```

Compiler Error C3189

Error Message

'typeid<type abstract declarator>': this syntax is no longer supported, use ::typeid instead

An obsolete form of typeid was used, use the new form.

The following sample generates C3189:

```
// C3189.cpp
// compile with: /clr
int main() {
   System::Type^ t  = typeid<System::Object>;   // C3189
   System::Type^ t2  = System::Object::typeid;   // OK
}
```

Compiler Error C3190

Error Message
'instantiation' with the provided template arguments is not the explicit instantiation of any member function of 'type'

The compiler detected an attempt to make an explicit function instantiation; however, the provided type arguments do not match any of the possible functions.

The following sample generates C3190:

```
// C3190.cpp
// compile with: /LD
template<class T>
struct A {
   A(int x = 0) {
   }
   A(int x, int y) {
   }
};

template A<float>::A();   // C3190
// try the following line instead
// template A<int>::A(int);

struct Y {
   template<class T> void f(T);
};

template<class T> void Y::f(T) { }

template void Y::f(int,int);   // C3190

template<class OT> class X {
   template<class T> void f2(T,OT);
};

template<class OT> template<class T> void X<OT>::f2(T,OT) {}

template void X<float>::f2<int>(int,char);   // C3190
// try one of the following lines instead
// template void X<char>::f2(int, char);
// template void X<char>::f2<int>(int,char);
// template void X<char>::f2<>(int,char);
```

Compiler Error C3191

Error Message
'syntax' : incompatible with /clr:oldSyntax

The syntax for Managed Extensions for C++ was requested (**/clr:oldSyntax**). However, the compiler found newer syntax in a source code file.

The following sample generates C3191:

```
// C3191.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;
int main() {
   Object ^ o;   // C3191
   Object * o2;   // OK
}
```

Compiler Error C3192

Error Message
syntax error : '^' is not a prefix operator (did you mean '*'?)

A handle cannot be used as a dereference operator.

The following sample generates C3192:

```
// C3192.cpp
// compile with: /clr
using namespace System;

ref class MyClass {
public:
   MyClass () {}
   MyClass(MyClass%) {}
};

int main() {
   MyClass ^ s = gcnew MyClass;
   MyClass b = ^s;   // C3192

   // OK
   MyClass b2 = *s;
}
```

Compiler Error C3194

Error Message
'member' : a value-type cannot have an assignment operator

Special member functions that require automatic invocation by the compiler, such as a copy constructor or copy assignment operator are not supported within a value class.

Example
The following sample generates C3194.

```
// C3194.cpp
// compile with: /clr /c
value struct MyStruct {
   MyStruct& operator= (const MyStruct& i) { return *this; }   // C3194
};

ref struct MyStruct2 {
   MyStruct2% operator= (const MyStruct2% i) { return *this; }   // OK
};
```

Compiler Error C3195

Error Message
'operator' : is reserved and cannot be used as a member of a ref class or value type. CLR operators must be defined using the 'operator' keyword

The compiler detected an operator definition using the Managed Extensions for C++ syntax.

Either use the new C++ syntax or use the **/clr:oldSyntax** compiler option to enable the Managed Extensions for C++ syntax.

The following sample generates C3195:

```
// C3195.cpp
// compile with: /clr /LD
#using <mscorlib.dll>
value struct V {
   static V op_Addition(V v, int i);   // C3195
   static V operator +(V v, char c);   // OK for new C++ syntax
};
```

Compiler Error C3196

Error Message
'keyword' : used more than once

A keyword was used more than once.

The following sample generates C3196:

```
// C3196.cpp
// compile with: /clr
ref struct R abstract abstract {};   // C3196
ref struct S abstract {};   // OK
```

Compiler Error C3197

Error Message
'keyword' : can only be used in definitions

A keyword was used in a declaration but is only valid in a definition.

The following sample generates C3197:

```
// C3197.cpp
// compile with: /clr /c
ref struct R abstract;   // C3197
ref struct R abstract {};   // OK

public ref class MyObject;   // C3197
ref class MyObject;   // OK
public ref class MyObject {};   // OK
```

Compiler Error C3198

Error Message
invalid use of floating-point pragmas: fenv_access pragma operates only in precise mode

fenv_access pragma was used under an /fp setting other than **/fp:precise**.

The following sample generates C3198:

```
// C3198.cpp
// compile with: /fp:fast
#pragma fenv_access(on)   // C3198
```

Compiler Error C3199

Error Message
invalid use of floating-point pragmas: exceptions are not supported in non-precise mode

The float_control pragma was used to specify floating-point exception model under an /fp setting other than **/fp:precise**.

The following sample generates C3199:

```
// C3199.cpp
// compile with: /fp:fast
#pragma float_control(except, on)   // C3199
```

Compiler Error C3200

Error Message
'template' : invalid template argument for template parameter 'parameter', expected a class template

You passed an invalid argument to a class template. The class template expects template as a parameter. In the following example, calling `Y<int, int> aY` will generate C3200. The first parameter needs to be a template, such as `Y<X, int> aY`.

```
// C3200.cpp
template<typename T>
class X
{
};

template<template<typename U> class T1, typename T2>
class Y
{
};

int main()
{
   Y<int, int> y;   // C3200
}
```

Compiler Error C3201

Error Message
the template parameter list for class template 'template' does not match the template parameter list for template parameter 'template'

You passed a class template in the argument to class template that does not take a template parameter. If you then issue `Y<X3, int> aY;` you will generate C3201. A more appropriate call would be `Y<int, int> aY;`.

```
// C3201.cpp
template<typename T1, typename T2>
class X1
{
};

template<template<typename T> class U = X1>   // C3201
class X2
{
};
```

Compiler Error C3202

Error Message
'arg name' : invalid default argument for template parameter 'parameter', expected a class template

You passed an invalid default argument for a template parameter.

The following sample generates C3202:

```
// C3202.cpp
template<typename T>
class X
{
};

class Z
{
};

template<template<typename U> class T1 = Z, typename T2> // C3202
class Y
{
};
```

Compiler Error C3203

Error Message
'type' : unspecialized class type can't be used as a type argument for type parameter 'param', expected a real type

You passed an invalid argument to a class template or generic. The class template or generic expects a type as a parameter.

This error can be generated as a result of compiler conformance work that was done for Visual C++ 2005: an unspecialized class template can't be used as a template argument in a base class list. To resolve C3203, explicitly add the template type parameter(s) to the template class name when using it as a template parameter in a base class list. See Breaking Changes in the Visual C++ 2005 Compiler for more information.

```
// C3203.cpp
template< typename T >
struct X {
   void f(X) {}
};

template< typename T >
struct Y : public X<Y> {   // C3203
// try the following line instead
// struct Y : public X<Y<T> > {
   void f(Y) {}
};

int main() {
   Y<int> y;
}
```

The following sample generates C3203:

```
// C3203_b.cpp
// compile with: /c
template <class T>
struct S1 {};

template <class T>
class C1 {};

typedef C1<S1> MyC1;   // C3203

// OK
template <template <class> class T>
class C2 {};

typedef C2<S1> MyC1;

template <class T>
class C3 {};

typedef C3<S1<int> > MyC12;
```

C3203 can also occur when using generics:

```
// C3203_c.cpp
// compile with: /clr /c
generic <class T>
value struct GS1 {};

generic <class T>
value struct GC1 {};

typedef GC1<GS1> MyGC1;   // C3203
```

```
typedef GC1<GS1<int> > MyGC2;   // OK
```

Compiler Error C3204

Error Message

'_alloca' cannot be called from within a catch block

This error occurs when you use a call to _alloca from within a catch block.

Example

The following sample generates C3204:

```
// C3204.cpp
// compile with: /EHsc
#include <malloc.h>

void ShowError(void)
{
   try
   {
   }
   catch(...)
   {
      _alloca(1);   // C3204
   }
}
```

Compiler Error C3205

Error Message
argument list for template parameter 'parameter' is missing

A template parameter is missing.

The following sample generates C3205:

```
// C3205.cpp
template<template<class> class T> struct A {
   typedef T unparameterized_type;   // C3205
   // try the following line instead
   // typedef T<int> unparameterized_type;
};

template <class T>
struct B {
   typedef int value_type;
};

int main() {
   A<B> x;
}
```

Compiler Error C3206

Error Message
'function' : invalid type argument for 'param', missing type argument list on class type 'typename'

A template function is defined as taking a template type argument. However, a template template argument was passed.

The following sample generates C3206:

```
// C3206.cpp
template <class T>
void f() {}

template <class T>
struct S {};

void f1() {
   f<S>();   // C3206
   // try the following line instead
   // f<S<int> >();
}
```

Possible resolution:

```
// C3206b.cpp
// compile with: /c
template <class T>
void f() {}

template <class T>
struct S {};

void f1() {
   f<S<int> >();
}
```

C3206 can also occur when using generics:

```
// C3206c.cpp
// compile with: /clr
generic <class GT1>
void gf() {}

generic <class T>
value struct GS {};

int main() {
   gf<GS>();   // C3206
}
```

Possible resolution:

```
// C3206d.cpp
// compile with: /clr
generic <class GT1>
void gf() {}

generic <class T>
value struct GS {};

int main() {
   gf<GS<int> >();
}
```

This error can also be generated as a result of compiler conformance work for Visual C++ .NET 2003, where class templates are not allowed as template type argument.

A class template is not allowed as a template type argument. This worked in Visual C++ .NET 2003, but it is invalid C++.

For more information, see Summary of Compile-Time Breaking Changes.

The following sample compiles in Visual C++ .NET 2002 but will fail in Visual C++ .NET 2003:

```
// C3206e.cpp
template <class T>
struct S {};

template <class T>
void func() {   // takes a type
   T<int> t;
}

int main() {
   func<S>();   // C3206 S is not a type.
}
```

Possible resolution:

```
// C3206f.cpp
template <class T>
struct S {};

template <class T>
void func() {   // takes a type
   T t;
}

int main() {
   func<S<int> >();
}
```

If a template template parameter is necessary, then the resolution for the error that is valid in both the Visual C++ .NET 2003 and Visual C++ .NET 2002 versions require you to wrap the function in a template class that takes a template template parameter:

```
// C3206g.cpp
template <class T>
struct S {};

template<template<class> class TT>
struct X {
   static void func() {
      TT<int> t1;
      TT<char> t2;
   }
};

int main() {
   X<S>::func();
}
```

Compiler Error C3207

Error Message
'function' : invalid template argument for 'arg', class template expected

A template function is defined as taking a template template argument. However, a template type argument was passed.

The following sample generates C3207:

```
// C3207.cpp
template <template <class T> class TT>
void f(){}

template <class T>
struct S
{
};

void f1()
{
   f<S<int> >();   // C3207
   // try the following line instead
   // f<S>();
}
```

Compiler Error C3208

Error Message
'function' : template parameter list for class template 'class' does not match template parameter list for template template parameter 'parameter'

A template template parameter does not have the same number of template parameters as the provided class template.

The following sample generates C3208:

```
// C3208.cpp
template <template <class T> class TT >
int f();

template <class T1, class T2>
struct S;

template <class T1>
struct R;

int i = f<S>();   // C3208
// try the following line instead
// int i = f<R>();
```

Compiler Error C3209

Error Message
'class' : generic class must be a managed class

A generic class must be a managed class.

The following sample generates C3209:

```
// C3209.cpp
// compile with: /clr
generic <class T>
class C {};   // C3209

// OK
generic <class T>
ref class D {};
```

Compiler Error C3210

Error Message

'type' : access declaration can only be applied to a base class member

A using declaration was specified incorrectly.

Example

The following sample generates C3210.

```
// C3210.cpp
// compile with: /c
struct A {
protected:
   int i;
};

struct B {
   using A::i;   // C3210
};

struct C : public A {
   using A::i;   // OK
};
```

Compiler Error C3211

Error Message
'explicit specialization' : explicit specialization is using partial specialization syntax, use template <> instead

An explicit specialization was ill formed.

The following sample generates C3211:

```
// C3211.cpp
// compile with: /LD
template<class T>
struct s;

template<class T>
// use the following line instead
// template<>
struct s<int>{};   // C3211
```

Compiler Error C3212

Error Message
'specialization' : an explicit specialization of a template member must be a member of an explicit specialization

An explicit specialization was ill formed.

The following sample generates C3212:

```
// C3212.cpp
// compile with: /LD
template <class T>
struct S {
   template <class T1>
   struct S1;
};


template <class T>   // C3212
template <>
struct S<T>::S1<int> {};

/*
// try the following instead
template <>
template <>
struct S<int>::S1<int> {};
*/

/*
// or, the following
template <>
struct S<int> {
   template <class T1>
   struct S1;
};

template <>
struct S<int>::S1<int> {
};
*/
```

Compiler Error C3213

Error Message
base class 'base_type' is less accessible than 'derived_type'

A type that will be visible from an assembly must use publicly visible base classes.

The following sample generates C3213:

```
// C3213.cpp
// compile with: /clr
private ref struct privateG {
public:
   int i;
};

public ref struct publicG {
public:
   int i;
};

public ref struct V : public privateG {   // C3213
public:
   int j;
};

public ref struct W: public publicG {   // OK
public:
   int j;
};
```

Compiler Error C3214

Error Message
'type' : invalid type argument for generic parameter 'param' of generic 'generic_type', does not meet constraint 'constraint'

The type was specified for an instantiation of a generic class that does not meet the generic class's constraint.

The following sample generates C3214:

```
// C3214.cpp
// compile with: /clr
interface struct A {};

generic <class T>
where T : A
ref class C {};

ref class X : public A {};

int main() {
   C<int>^ c = new C<int>;   // C3214
   C<X ^> ^ c2 = new C<X^>;   // OK
}
```

Compiler Error C3215

Error Message
'type1' : generic type parameter already constrained by 'type2'

A constraint was specified more than once.

For more information on generics, see Generics (Visual C++).

The following sample generates C3215:

```
// C3215.cpp
// compile with: /clr
interface struct A {};

generic <class T>
where T : A,A
ref class C {};   // C3215
```

Possible resolution:

```
// C3215b.cpp
// compile with: /clr /c
interface struct A {};

generic <class T>
where T : A
ref class C {};
```

Compiler Error C3216

Error Message
constraint must be a generic parameter, not 'type'

A constraint was ill formed.

The following sample generates C3216:

```
// C3216.cpp
// compile with: /clr
interface struct A {};

generic <class T>
where F : A   // C3216
// Try the following line instead:
// where T : A    // C3216
ref class C {};
```

The following example demonstrates a possible resolution:

```
// C3216b.cpp
// compile with: /clr /c
interface struct A {};

generic <class T>
where T : A
ref class C {};
```

Compiler Error C3217

Error Message
'param' : generic parameter cannot be constrained in this declaration

A constraint was ill formed; the constraint generic parameter must agree with the generic class template parameter.

The following sample generates C3217:

```
// C3217.cpp
// compile with: /clr
interface struct A {};

generic <class T>
ref class C {
   generic <class T1>
   where T : A   // C3217
   void f();
};
```

The following sample demonstrates a possible resolution:

```
// C3217b.cpp
// compile with: /clr /c
interface struct A {};

generic <class T>
ref class C {
   generic <class T1>
   where T1 : A
   void f();
};
```

Compiler Error C3218

Error Message

'type' : type not allowed as a constraint

For a type to be a constraint, it must be either a value type or reference to a managed class or interface.

Example

The following sample generates C3218.

```
// C3218.cpp
// compile with: /clr /c
class A {};
ref class B {};

// Delete the following 3 lines to resolve.
generic <class T>
where T : A   // C3218
ref class C {};

// OK
generic <class T>
where  T : B
ref class D {};
```

Compiler Error C3219

Error Message
'param' : generic parameter cannot be constrained by multiple non-interfaces : 'class'

It is not valid to constrain a generic parameter by two or more managed classes.

The following sample generates C3219:

```
// C3219.cpp
// compile with: /clr
ref class A {};
ref class B {};

generic <class T>
where T : A, B
ref class E {};   // C3219
```

The following sample demonstrates a possible resolution:

```
// C3219b.cpp
// compile with: /clr /c
ref class A {};

interface struct C {};

generic <class T>
where T : A
ref class E {};
```

Compiler Error C3222

Error Message
'parameter' : cannot declare default arguments for member functions of a managed type or generic functions

It is not permitted to declare a method parameter with a default argument. An overloaded form of the method is one way to work around this issue. That is, define a method with the same name with no parameters and then initialize the variable in the method body.

The following sample generates C3222:

```
// C3222_2.cpp
// compile with: /clr
public ref class G {
   void f( int n = 0 );   // C3222
};
```

The following sample generates C3222:

```
// C3222.cpp
// compile with: /clr:oldSyntax
public __gc class G {
   void f( int n = 0 );   // C3222
};
```

Compiler Error C3223

Error Message

'property' : you cannot apply 'typeid' to a property

You cannot apply typeid to a property.

Example

The following sample generates C3223.

```
// C3223.cpp
// compile with: /clr
ref class R {
public:
   property int myprop;
};

int main() {
   System::Type^ type2 = R::myprop::typeid;   // C3223
}
```

Compiler Error C3224

Error Message
'type' : no overloaded generic class takes 'number' generic type arguments

The compiler failed to find an appropriate overload.

The following sample generates C3224:

```
// C3224.cs
// compile with: /target:library
public class C<T> {}
public class C<T,U> {}
```

And then,

```
// C3224b.cpp
// compile with: /clr
#using "C3224.dll"
int main() {
   C<int,int,int>^ c = gcnew C<int,int,int>();   // C3224
   C<int,int>^ c2 = gcnew C<int,int>();   // OK
}
```

Compiler Error C3225

Error Message

generic type argument for 'arg' cannot be 'type', it must be a value type or handle type

The generic type argument was not of the correct type.

For more information, see Generics (Visual C++).

Example

You cannot instantiate a generic type with a native type. The following sample generates C3225.

```
// C3225.cpp
// compile with: /clr
class A {};

ref class B {};

generic <class T>
ref class C {};

int main() {
   C<A>^ c = gcnew C<A>;   // C3225
   C<B^>^ c2 = gcnew C<B^>;   // OK
}
```

The following sample creates a component using C#. Notice that the constraint specifies that the generic type can only be instantiated with a value type.

```
// C3225_b.cs
// compile with: /target:library
// a C# program
public class MyList<T> where T: struct {}
```

This sample consumes the C#-authored component, and violates the constraint that MyList can only be instantiated with a value type other than Nullable. The following sample generates C3225.

```
// C3225_c.cpp
// compile with: /clr
#using "C3225_b.dll"
ref class A {};
value class B {};
int main() {
   MyList<A> x;   // C3225
   MyList<B> y;   // OK
}
```

Compiler Error C3226

Error Message
A template declaration is not allowed inside a generic declaration

Use a generic declaration inside a generic class.

The following sample generates C3226:

```
// C3226.cpp
// compile with: /clr
generic <class T>
ref class C {
   template <class T1>   // C3226
   ref struct S1 {};
};
```

The following sample demonstrates a possible resolution:

```
// C3226b.cpp
// compile with: /clr /c
generic <class T>
ref class C {
   generic <class T1>
   ref struct S1 {};
};
```

Compiler Error C3227

Error Message
'parameter' : cannot use 'keyword' to allocate a generic type

In order to instantiate a type, an appropriate constructor is required. However, the compiler is not able to ensure that an appropriate constructor is available.

You can use templates instead of generics to resolve this error, or you can use one of several methods to create an instance of the type.

Example
The following sample generates C3227.

```
// C3227.cpp
// compile with: /clr /c
generic<class T> interface class ICreate {
   static T Create();
};

generic <class T>
where T : ICreate<T>
ref class C {
   void f() {
      T t = new T;   // C3227

      // OK
      T t2 = ICreate<T>::Create();
      T t3 = safe_cast<T>( System::Activator::CreateInstance(T::typeid) );
   }
};
```

Compiler Error C3228

Error Message

'function' : generic type argument for 'param' cannot be 'type', it must be a valuetype or handle type

An incorrect type was passed as a generic type argument.

The following sample generates C3228:

```
// C3228.cpp
// compile with: /clr
class A {};

value class B {};

generic <class T>
void Test() {}

ref class C {
public:
   generic <class T>
   static void f() {}
};

int main() {
   C::f<A>();   // C3228
   C::f<B>();   // OK

   Test<C>();   // C3228
   Test<C ^>();   // OK
}
```

Compiler Error C3229

Error Message

'type' : indirections on a generic type parameter are not allowed

You cannot use generic parameters with *****, **^**, or **&**.

Example

The following sample generates C3229.

```
// C3229.cpp
// compile with: /clr /c
generic <class T>
ref class C {
   T^ t;   // C3229
};

// OK
generic <class T>
ref class D {
   T u;
};
```

The following sample generates C3229.

```
// C3229_b.cpp
// compile with: /clr /c
generic <class T>   // OK
ref class Utils {
   static void sort(T elems[], size_t size);   // C3229
   static void sort2(T elems, size_t size);   // OK
};
```

Compiler Error C3230

Error Message
'function' : template type argument for 'template' cannot contain a generic type parameter: 'param'

Templates are instantiated at compile time, but generics are instantiated at run time. Therefore, it is not possible to generate generic code that can call the template because the template cannot be instantiated at run time when the generic type is finally known.

The following sample generates C3230:

```
// C3230.cpp
// compile with: /clr /LD
template <class S>
void f(S t);

generic <class U>
ref class C {
   void f1(U x) {
      f(x);   // C3230
   }
};
```

Compiler Error C3231

Error Message
'arg' : template type argument cannot use a generic type parameter

Templates are instantiated at compile time, but generics are instantiated at run time. Therefore, it is not possible to generate generic code that can call the template because the template cannot be instantiated at run time when the generic type is finally known.

The following sample generates C3231:

```
// C3231.cpp
// compile with: /clr /LD
template <class T> class A;

generic <class T>
ref class C {
   void f() {
      A<T> a;   // C3231
   }
};
```

Compiler Error C3232

Error Message
'param' : a generic type parameter cannot be used in a qualified name

A generic type parameter was used incorrectly.

The following sample generates C3232:

```
// C3232.cpp
// compile with: /clr
generic <class T>
ref class C {
   typename T::TYPE t;   // C3232
};
```

Compiler Error C3233

Error Message
'type' : generic type parameter already constrained

It is not valid to constrain a generic parameter in more than one **where** clause.

The following sample generates C3233:

```
// C3233.cpp
// compile with: /clr /LD

interface struct C {};
interface struct D {};

generic <class T>
where T : C
where T : D
ref class E {};   // C3233
```

Compiler Error C3234

Error Message

a generic class may not derive from a generic type parameter

A generic class cannot inherit from a generic type parameter.

Example

The following sample generates C3234.

```
// C3234.cpp
// compile with: /clr /c
generic <class T>
public ref class C : T {};   // C3234
// try the following line instead
// public ref class C {};
```

Compiler Error C3235

Error Message

'specialization' : explicit or partial specialization of a generic class is not allowed

Generic classes cannot be used for explicit or partial specializations.

Example

The following sample generates C3235.

```
// C3235.cpp
// compile with: /clr
generic<class T>
public ref class C {};

generic<>
public ref class C<int> {};   // C3235 Remove this specialization to resolve this error.
```

Compiler Error C3236

Error Message
explicit instantiation of a generic is not allowed

The compiler does not allow explicit instantiation of generic classes.

The following sample generates C3236:

```
// C3236.cpp
// compile with: /clr
generic<class T>
public ref class X {};

generic ref class X<int>;   // C3236
```

The following sample demonstrates a possible resolution:

```
// C3236b.cpp
// compile with: /clr /c
generic<class T>
public ref class X {};
```

Compiler Error C3237

Error Message

'generic_class' : a generic class cannot be a custom attribute

Generic classes cannot be user-defined attributes.

Example

The following sample generates C3237.

```
// C3237.cpp
// compile with: /clr /c
// C3237 expected
using namespace System;

generic <class T>
// Delete the following line to resolve.
[attribute(AttributeTargets::All, AllowMultiple=true)]
public ref class GR {};
```

Compiler Error C3238

Error Message

'type' : a type with this name has already been forwarded to assembly 'assembly'

A type was defined in a client application that is also defined, via type forwarding syntax, in a referenced assembly. Both types cannot be defined in the scope of the application.

See Type Forwarding for more information.

Example

The following sample creates an assembly that contains a type that was forwarded from another assembly.

```
// C3238.cpp
// compile with: /clr /LD
public ref class R {};
```

The following sample creates an assembly that used to contain the type definition, but not only contains type forwarding syntax.

```
// C3238_b.cpp
// compile with: /clr /LD
#using "C3238.dll"
[ assembly:TypeForwardedTo(R::typeid) ];
```

The following sample generates C3238.

```
// C3238_c.cpp
// compile with: /clr /c
// C3238 expected
// Delete the following line to resolve.
#using "C3238_b.dll"
public ref class R {};
```

Compiler Error C3239

Error Message

'type' : pointer to interior/pin pointer is disallowed by the common language runtime

The compiler encountered an invalid type.

The following sample generates C3229:

```
// C3239.cpp
// compile with: /clr
int main() {
   interior_ptr<int>* pip0;   // C3239

   // OK
   int * pip1;
   interior_ptr<int> pip2;
   int ** pip;
}
```

Compiler Error C3240

Error Message

'function' : must be a non-overloaded abstract member function of 'type'

A base type contained a function that was defined. Function must be virtual.

Example

The following sample generates C3240.

```
// C3240.cpp
// compile with: /c
__interface I {
   void f();
};

struct A1 : I {
   void f() {}
};

struct A2 : I {
   void f() = 0;
};

template <class T>
struct A3 : T {
   void T::f() {}
};

template <class T>
struct A4 : T {
   void T::f() {}
};

A3<A1> x;   // C3240
A3<I> x2;
A4<A2> x3;
```

Compiler Error C3241

Error Message
'method' : this method was not introduced by 'interface'

When you explicitly override a function, the function signature must exactly match the declaration for the function that you are overriding.

The following sample generates C3241:

```
// C3241.cpp
#pragma warning(disable:4199)

__interface IX12A {
   void mf();
};

__interface IX12B {
   void mf(int);
};

class CX12 : public IX12A, public IX12B { // C3241
   void IX12A::mf(int);
};
```

Compiler Error C3242

Error Message

'function' : you can only explicitly override virtual functions

You tried to explicitly override a nonvirtual method.

Compiler Error C3243

Error Message
none of the overload functions were introduced by 'interface'

You tried to explicitly override a member that does not exist in the specified interface.

The following sample generates C3243:

```
// C3243.cpp
#pragma warning(disable:4199)
__interface IX14A {
   void g();
};

__interface IX14B {
   void f();
   void f(int);
};

class CX14 : public IX14A, public IX14B {
public:
   void IX14A::g();
   void IX14B::f();
   void IX14B::f(int);
};

void CX14::IX14A::f()   // C3243 occurs here
{
}
```

Compiler Error C3244

Error Message
'method' : this method was introduced by 'interface' not by 'interface'

You tried to explicitly override a member that does not exist in the specified interface but does exist in another base class.

The following sample generates C3244:

```
// C3244.cpp
#pragma warning(disable:4199)

__interface IX15A {
   void f();
};

__interface IX15B {
   void g();
};

class CX15 : public IX15A, public IX15B {
public:
   void IX15A::f();
   void IX15B::g();
};

void CX15::IX15A::g()   // C3244 occurs here
{
}
```

Compiler Error C3246

Error Message
'class' : cannot inherit from 'type' as it has been declared as 'sealed'

A class that is marked as sealed cannot be the base class for any other classes.

The following sample generates C3246:

```
// C3246_2.cpp
// compile with: /clr /LD
ref class X sealed {};

ref class Y : public X {}; // C3246
```

The following sample generates C3246:

```
// C3246.cpp
// compile with: /clr:oldSyntax /LD
#using <mscorlib.dll>
__sealed __gc class X {};

__gc class Y : public X {}; // C3246
```

Compiler Error C3247

Error Message
'class1' : a coclass cannot inherit from another coclass 'class2'

A class marked with the coclass attribute cannot inherit from another class marked with the **coclass** attribute.

The following sample generates C3247:

```
// C3247.cpp
[module(name="MyLib")];
[coclass]
class a {
};

[coclass]
class b : public a {   // C3247
};
int main() {
}
```

Compiler Error C3248

Error Message
'function1': function declared as '__sealed' cannot be overridden by 'function2'

A derived class tried to override a __sealed virtual method.

C3248 is only reachable using **/clr:oldSyntax**.

The following sample generates C3248:

```
// C3248b.cpp
// compile with: /clr:oldSyntax
using namespace System;

__gc struct B {
   __sealed virtual void func();
};

__gc struct D : B {
   void func();   // C3248
};
```

Compiler Error C3251

Error Message
cannot invoke base class method on a value type instance

The following error occurs because `GetClass` is a member of `Microsoft.Runtime.Object`, not of `Microsoft.Runtime.Integer4`.

Compiler Error C3252

Error Message
'method' : cannot reduce accessibility of a virtual method in a managed type

A class that implements a virtual method from a base class or any method from an interface cannot reduce the access of that method.

Note that all methods in an interface are public.

The following sample generates C3252:

```
// C3252.cpp
// compile with: /clr /c
ref class A {
public:
   virtual void f1() {}
};

ref class B : public A {
// public:
   virtual void f1() override sealed {}   // C3252, make this method public
};
```

The following sample generates C3252:

```
// C3252_2.cpp
// compile with: /clr:oldSyntax /LD
__gc class A {
public:
   virtual void f1() {}
};

__gc class B : public A {
// public:
   virtual void f1() {}   // C3252, make this method public
};
```

Compiler Error C3253

Error Message
'function' : error with explicit override

An explicit override was specified incorrectly. For example, you cannot specify an implementation for an override that you also specify as pure. For more information, see Explicit Overrides.

The following sample generates C3253:

```
// C3253.cpp
// compile with: /clr
public interface struct I {
   void a();
   void b();
   void c();
};

public ref struct R : I {
   virtual void a() = 0, I::a {}   // C3253
   virtual void b() = I::a {}   // OK
   virtual void c() = 0;   // OK
};
```

Compiler Error C3254

Error Message
'explicit override' : class contains explicit override 'override' but does not derive from an interface that contains the function declaration

When you explicitly override a method, the class that contains the override must derive, directly or indirectly, from the type that contains the function you are overriding.

The following sample generates C3254:

```
// C3254.cpp
__interface I
{
   void f();
};

__interface I1 : I
{
};

struct A /* : I1 */
{
   void I1::f()
   {   // C3254, uncomment : I1 to resolve this C3254
   }
};

int main()
{
}
```

Compiler Error C3255

Error Message

'value type' : cannot dynamically allocate this value type object on native heap

Instances of a value type (see Classes and Structs (Managed)) that contain managed members can be created on the stack but not on the heap.

The following sample generates C3255:

```
// C3255.cpp
// compile with: /clr
using namespace System;
value struct V {
   Object^ o;
};

value struct V2 {
   int i;
};

int main() {
   V* pv = new V;   // C3255
   V2* pv2 = new V2;
   V v2;
}
```

Managed Extensions for C++

The following sample generates C3255:

```
// C3255b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;

__value struct V {
   Object* o;
};

__value struct V2 {
   int i;
};

int main() {
   V* pv = __nogc new V;   // C3255
   V2* pv2 = __nogc new V2;   // OK
}
```

Compiler Error C3257

Error Message
'object' : explicit call to ctor of __gc class type not allowed (did you forget 'new'?)

You cannot create temporary stack-based objects for managed classes. The compiler attempted to create a temporary object that is not directly mentioned in the code. This temporary object is created on the stack because there is no call to new.

C3257 is only reachable using **/clr:oldSyntax**.

The following sample generates C3257:

```
// C3257a.cpp
// compile with: /clr:oldSyntax /c
__gc class X {
public:
   X();
   X(int i) {
      X();   // C3257
      X *MyX = new X();   // OK
   }
};
```

Compiler Error C3258

Error Message
managed operators must be declared using 'static op_<Name>' form ('operator' form is disallowed)

An operator overload in a managed class was specified incorrectly.

C3258 is only reachable using **/clr:oldSyntax**.

The following sample generates C3258:

```
// C3258.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc class A {
public:
   A *operator=(const int _i) { // C3258
   // try the following line instead
   // static bool op_Equality(A & _j, A & _k)
   {
      return (_j.i == _k.i);
   }

   int i;
};

int main() {
        A *a = new A;
        A *b = new A;
        a->i  = 11;
        b->i = 11;
        System::Console::WriteLine(*a == *b);
}
```

Compiler Error C3261

Error Message
a function returning a managed array must have array brackets at the end of the declaration: 'declaration'

A method returning a managed array was not declared correctly.

C3261 is only reachable using **/clr:oldSyntax**.

The following sample generates C3261:

```
// C3261.cpp
// compile with: /clr:oldSyntax /LD
#using <mscorlib.dll>
using namespace System;

Int32[] f()   // C3261
// try the following line instead
// Int32 f()[]
{
   return 0;
}
```

Compiler Error C3262

Error Message
invalid array indexing: '#' dimension(s) specified for '#'-dimensional 'array type'

An array was improperly subscripted. The number of indices may not match the number of dimensions in the array.

The following sample generates C3262:

```
// C3262.cpp
// compile with: /clr
#using <mscorlib.dll>
using namespace System;

#define ARRAY_SIZE 2

ref class MyClass {
public:
   int m_i;
};

// returns a multidimensional managed array of a reference type
array<MyClass^, 2>^ Test0() {
   int i, j;
   array< MyClass^, 2 >^ local = new array< MyClass^, 2 >
      (ARRAY_SIZE, ARRAY_SIZE);

   for (i = 0 ; i < ARRAY_SIZE ; i++)
      for (j = 0 ; j < ARRAY_SIZE ; j++) {
         local[i][j] = new MyClass;   // C3262
         // try the following line instead
         // local[i,j] = new MyClass;
         local[i,j] -> m_i = i;
      }

      return local;
}

int main() {
   int i, j;

   array< MyClass^, 2 >^ MyClass0;
   MyClass0 = Test0();
}
```

Managed Extensions for C++

The following sample generates C3262:

```
// C3262b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
int main() {
int iArr __gc[,] = new int __gc[10,20];
iArr[1,2,3]; // C3262: 3 dimensions specified for 2-dimensional array
}
```

Compiler Error C3264

Error Message
'class' : a class-constructor cannot have a return type

Class constructors cannot have return types.

The following sample generates C3264:

```
// C3264_2.cpp
// compile with: /clr

ref class X {
   public:
      static int X()   { // C3264
      }

      /* use the code below to resolve the error
      static X() {
      }
      */
};
int main() {
}
```

The following sample generates C3264:

```
// C3264.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc class X {
   public:
      static int X()   { // C3264
      }

      /* use the code below to resolve the error
      static X() {
      }
      */
};
int main() {
}
```

Compiler Error C3265

Error Message
cannot declare a managed 'managed construct' in an unmanaged 'unmanaged construct'

You cannot include a managed object in an unmanaged context.

The following sample reproduces C3265:

```
// C3265_2.cpp
// compile with: /clr /LD
#include <vcclr.h>

ref class A { };

class B
// try the following line instead
// ref class B
{
   A ^a;   // C3265
   // or embed the managed handle using gcroot
   // try the following line instead
   // gcroot<A^> a;
};
```

The following sample reproduces C3265:

```
// C3265.cpp
// compile with: /clr:oldSyntax /LD
#using <mscorlib.dll>
__gc class A { };

__nogc class B
// try the following line instead
// __gc class B
{
   A *a;   // C3265
};
```

C3265 can also occur if you embed a managed pointer directly inside an unmanaged class. To fix this error, use **gcroot**:

```
// C3265b.cpp
// compile with: /clr:oldSyntax
#include <vcclr.h>
#using <mscorlib.dll>

namespace TestNS {
   __gc public class Test{};
}

template<class T>
struct Container {
  T* m_px;   // C3265
};
__gc public class ClassA {
public:
  ClassA (){}
   ~ClassA(){}

private:
   Container<TestNS::Test*>  vctTest;
   // try the following line instead
   Container<gcroot<TestNS::Test* > > vctTest2;
};
```

Compiler Error C3266

Error Message
'class' : a class-constructor must have a 'void' parameter list

Class-constructors in a class using /clr programming cannot take parameters.

The following sample generates C3266:

```
// C3266.cpp
// compile with: /clr

ref class X {
   static X(int i) { // C3266
   // try the following line instead
   // static X() {
   }
};

int main() {
}
```

The following sample generates C3266:

```
// C3266b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc class X {
   static X(int i) { // C3266
   // try the following line instead
   // static X() {
   }
};

int main() {
}
```

Compiler Error C3267

Error Message
'class' : a class-constructor must be fully defined within the enclosing class

Class constructors must be fully defined within the enclosing class definition.

C3267 is only reachable using **/clr:oldSyntax**.

The following sample generates C3267:

```
// C3267.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc class X {
   public:
   static X(); // C3267

   /* use the lines below to resolve the error
   static X() {
   }
   */
};

int main() {
}
```

Compiler Error C3268

Error Message

'function' : a generic function or a member-function of a generic class cannot have a variable parameter list

See Generics (Visual C++) for more information.

Example

The following sample generates C3268.

```
// C3268.cpp
// compile with: /clr:pure /c
generic <class ItemType>
void Test(ItemType item, ...) {}   // C3268
// try the following line instead
// void Test(ItemType item) {}

generic <class ItemType2>
ref struct MyStruct { void Test(...){} };   // C3268
// try the following line instead
// ref struct MyStruct { void Test2(){} };   // OK
```

Compiler Error C3269

Error Message
'function' : a member-function of a managed type cannot be declared with '...'

Managed-class member functions cannot declare variable-length parameter lists.

The following sample generates C3269:

```
// C3269_2.cpp
// compile with: /clr

ref struct A
{
   void func(int i, ...)   // C3269
   // try the following line instead
   // void func(int i )
   {
   }
};

int main()
{
}
```

The following sample generates C3269:

```
// C3269.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc struct A
{
   void func(int i, ...)   // C3269
   // try the following line instead
   // void func(int i )
   {
   }
};

int main()
{
}
```

Compiler Error C3270

Error Message
'field': the FieldOffset attribute can only be used in the context of StructLayout(Explicit), in which case it is required

A field was marked with FieldOffset, which is only allowed when StructLayout Explicit is in effect.

The following sample generates C3270:

```
// C3270_2.cpp
// compile with: /clr /c
using namespace System::Runtime::InteropServices;

[ StructLayout(LayoutKind::Sequential) ]
// try the following line instead
// [ StructLayout(LayoutKind::Explicit) ]
public value struct MYUNION
{
   [FieldOffset(0)] int a;   // C3270
   // ...
};
```

The following sample generates C3270:

```
// C3270.cpp
// compile with: /clr:oldSyntax /LD
#using <mscorlib.dll>
using namespace System::Runtime::InteropServices;

[ StructLayout(LayoutKind::Sequential) ]
// try the following line instead
// [ StructLayout(LayoutKind::Explicit) ]
public __value struct MYUNION
{
   [FieldOffset(0)] int a;   // C3270
   // ...
};
```

Compiler Error C3271

Error Message
'member': invalid value 'value' for the FieldOffset attribute

A negative number was passed to the FieldOffset attribute.

The following sample generates C3271:

```
// C3271.cpp
// compile with: /clr /c
using namespace System;
using namespace System::Runtime::InteropServices;

[StructLayout(LayoutKind::Explicit)]
value class MyStruct1 {
   public: [FieldOffset(0)] int a;
   public: [FieldOffset(-1)] long b;   // C3271
};
```

The following sample generates C3271:

```
// C3271_2.cpp
// compile with: /clr:oldSyntax
using namespace System;
using namespace System::Runtime::InteropServices;

[StructLayout(LayoutKind::Explicit)]
__value class MyStruct1 {
   public: [FieldOffset(0)] int a;
   public: [FieldOffset(-1)] long b;   // C3271
};

static int main() {
   MyStruct1* a = __nogc new MyStruct1();
};
```

Compiler Error C3272

Error Message
'symbol' : symbol requires FieldOffset, as it is a member of type typename defined with StructLayout(LayoutKind::Explicit)

When StructLayout **Explicit** is in effect, fields must be marked with FieldOffset.

The following sample generates C3272:

```
// C3272_2.cpp
// compile with: /clr /c
using namespace System;
using namespace System::Runtime::InteropServices;

[StructLayout(LayoutKind::Explicit)]
ref struct X
{
   int data_;   // C3272
   // try the following line instead
   // [FieldOffset(0)] int data_;
};
```

The following sample generates C3272:

```
// C3272.cpp
// compile with: /clr:oldSyntax /LD
#using <mscorlib.dll>
using namespace System;
using namespace System::Runtime::InteropServices;

[StructLayout(LayoutKind::Explicit)]
__gc struct X
{
   int data_;   // C3272
   // try the following line instead
   // [FieldOffset(0)] int data_;
};
```

Compiler Error C3273

Error Message

__finally cannot be used on an exception block in unmanaged code.

The following sample generates C3273:

```
// C3273.cpp
// compile with: /GX
int main()
{
   try
   {
   }
   catch (int)
   {
   }
   __finally   // C3273, remove __finally clause
   {
   }
}
```

Compiler Error C3274

Error Message
__finally/finally without matching try

A __finally or finally statement was found without a matching **try**. To resolve this, either delete the **__finally** statement or add a **try** statement for the **__finally**.

The following sample generates C3274:

```
// C3274.cpp
// compile with: /clr
// C3274 expected
using namespace System;
int main() {
   try {
      try {
         throw gcnew ApplicationException();
      }
      catch(...) {
         Console::Error->WriteLine(L"Caught an exception");
      }
      finally {
         Console::WriteLine(L"In finally");
      }
   } finally {
      Console::WriteLine(L"In finally");
   }

   // Uncomment the following 3 lines to resolve.
   // try {
   //   throw gcnew ApplicationException();
   // }

   finally {
      Console::WriteLine(L"In finally");
   }
   Console::WriteLine(L"**FAIL**");
}
```

Compiler Error C3275

Error Message
'enum member' : cannot use this symbol without qualifier

When using managed code and when two or more enumerations contain an identifier with the same name, you must explicitly qualify references to the identifier.

C3275 is only reachable using **/clr:oldSyntax**.

The following sample shows two situations in which C3275 could be generated:

```
// C3275.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
__value enum Jewelry {
   necklace, brooch, pin, ring, earring
   };

__value enum Phone {
   busy, ring, disconnect
   };

int main() {
   Phone p = ring;   // C3275
   // try the following line instead
   // Phone p = Phone::ring;

   Console::Out->Write(ring);   // C3275
   // try the following line instead
   // Console::Out->Write(Phone::ring);
}
```

Compiler Error C3276

Error Message

'keyword' : jump out of __finally/finally block has undefined behavior during termination handling

This error is the same as the C4532 warning. However, when you are using /clr, this condition cannot be disabled with the warning pragma.

Compiler Error C3277

Error Message
cannot define an unmanaged enum 'enum' inside managed 'type'

An enumeration was defined incorrectly inside a managed type.

The following sample generates C3277:

```
// C3277a.cpp
// compile with: /clr
ref class A
{
   enum E {e1,e2};   // C3277
   // try the following line instead
   // enum class E {e1,e2};
};

int main()
{
}
```

The following sample generates C3277:

```
// C3277b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
__gc class A
{
   enum E {e1,e2};   // C3277
   // try the following line instead
   // __value enum E {e1,e2};
};

int main()
{
}
```

Compiler Error C3278

Error Message
direct call of interface method 'interface method' will fail at runtime

A call was made to an interface method, which is not allowed.

The following sample generates C3278:

```
// C3278_2.cpp
// compile with: /clr
using namespace System;
interface class I
{
   void vmf();
};

public ref class C: public I
{
public:
   void vmf()
   {
      Console::WriteLine( "In C::vmf()" );
      I::vmf(); // C3278
   }

};

int main()
{
   C^ pC = gcnew C;
   pC->vmf();
}
```

The following sample generates C3278:

```
// C3278.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;

public __gc __interface I
{
   void vmf();
};

public __gc class C : public I
{
public:
   void vmf()
   {
      Console::WriteLine(S"In C::vmf()");
      I::vmf();   // C3278
   }
};

int main()
{
   C *pC = new C();
   pC->vmf();
}
```

Compiler Error C3279

Error Message
partial and explicit specializations as well as explicit instantiations of class templates declared in the cli namespace are disallowed

The **cli** namespace is defined by Microsoft and contains pseudo-templates. The Visual C++ compiler does not allow user-defined, partial and explicit specializations, and explicit instantiations of class templates in this namespace.

The following sample generates C3279:

```
// C3279.cpp
// compile with: /clr
namespace cli {
   template <> ref class array<int> {};   // C3279
   template <typename T> ref class array<T, 2> {};   // C3279
}
```

Compiler Error C3280

Error Message
'class' : a member-function of a managed type cannot be compiled as an unmanaged function

Managed class member functions cannot be compiled as unmanaged functions.

The following sample generates C3280:

```
// C3280_2.cpp
// compile with: /clr
ref struct A {
   void func();
};

#pragma managed(push,off)

void A::func()   // C3280
{
}

#pragma managed(pop)
```

The following sample generates C3280:

```
// C3280.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc struct A {
   void func();
};

#pragma managed(push,off)

void A::func()   // C3280
{
}

#pragma managed(pop)
```

Compiler Error C3281

Error Message
'operator' : global operator cannot have managed type 'type' in signature

A global operator cannot have a managed type in the function signature.

C3281 is only reachable using **/clr:oldSyntax**.

The following sample generates C3281:

```
// C3281.cpp
// compile with: /clr:oldSyntax
using namespace System;
__value struct S {
   int i;
   static S op_Addition(S s, char c) { s.i -= c; return s;}
};

S& operator+(S& s, int i) {   // C3281, do not use S in signature
   s.i += i;
   return s;
}

int main() {
   S s = {10};

   s = s + '\x5';   // calls S::op_Addition
   Console::WriteLine(s.i);

   s = s + 10;   // calls global operator+
   Console::WriteLine(s.i);
}
```

Compiler Error C3282

Error Message

generic parameter lists can only appear on managed classes, structs, or functions

A generic parameter list was used incorrectly. For more information, see Generics (Visual C++).

Example

The following sample generates C3282.

```
// C3282.cpp
// compile with: /clr /c
generic <typename T> int x;   // C3282

ref struct GC0 {
   generic <typename T> int x;   // C3282
};

// OK
generic <typename T>
ref class M {};
```

Compiler Error C3283

Error Message
'type' : an interface cannot have an instance constructor

A CLR interface cannot have an instance constructor. A static constructor is allowed.

The following sample generates C3283:

```
// C3283.cpp
// compile with: /clr
interface class I {
   I();   // C3283
};
```

Possible resolution:

```
// C3283b.cpp
// compile with: /clr /c
interface class I {
   static I(){}
};
```

Compiler Error C3284

Error Message
the constraints for generic parameter 'parameter' of function 'function' must match the constraints for generic parameter 'parameter' of function 'function'

A virtual generic function must use the same constraints as a virtual function with the same name and set of arguments in the base class.

The following sample generates C3284:

```
// C3284.cpp
// compile with: /clr /c
// C3284 expected
public interface class IGettable {
   int Get();
};

public interface class B {
   generic<typename T>
   where T : IGettable
   virtual int mf(T t);
};

public ref class D : public B {
public:
   generic<typename T>
   // Uncomment the following line to resolve.
   // where T : IGettable
   virtual int mf(T t) {
      return 4;
   }
};
```

Compiler Error C3285

Error Message
for each statement cannot operate on variables of type 'type'

The **for each** statement repeats a group of embedded statements for each element in an array or an object collection.

See for each, in for more information.

Example
The following sample generates C3285.

```
// C3285.cpp
// compile with: /clr
int main() {
   for each (int i in 0) {}   // C3285

   array<int> ^p = { 1, 2, 3 };
   for each (int j in p) {}   // OK
}
```

Compiler Error C3286

Error Message

'specifier': an iteration variable cannot have any storage-class specifiers

See Storage-Class Specifiers for more information.

See for each, in for more information.

Example

The following sample generates C3286.

```
// C3286.cpp
// compile with: /clr
int main() {
   array<int> ^p = { 1, 2, 3 };
   for each (static int i in p) {}   // C3286
   for each (int j in p) {}   // OK
}
```

Compiler Error C3287

Error Message
the type 'type' (return type of GetEnumerator) must have a suitable public MoveNext member function and public Current property

User-defined collection classes must contain definitions for `MoveNext` and `Current`.

See How to: Iterate Over a User-Defined Collection with for each for more information.

Example
The following sample generates C3287.

```
// C3287.cpp
// compile with: /clr
using namespace System;

ref struct R {
   bool MoveNext() {
      return true;
   }
   property Object^ Current {
      Object^ get() {
         Object ^ o = gcnew Object;
         return o;
      }
   }
};

ref struct R2 {
   R ^GetEnumerator() {
      R^ r = gcnew R;
      return r;
   }
};

ref struct T {};

ref struct T2 {
   T ^GetEnumerator() {
      T^ t = gcnew T;
      return t;
   }
};

int main() {
   for each (int i in gcnew T2) {}   // C3287
   for each (int i in gcnew R2) {}   // OK
}
```

Compiler Error C3288

Error Message

'type' : illegal dereference of a handle type

The compiler detected an illegal dereference of a handle type. You can dereference a handle type and assign it to a reference. For more information, see % (Tracking Reference).

Example

The following sample generates C3288.

```
// C3288.cpp
// compile with: /clr
ref class R {};
int main() {
   *(System::Object^) nullptr;   // C3288

// OK
   (System::Object^) nullptr;   // OK
   R^ r;
   R% pr = *r;
}
```

Compiler Error C3289

Error Message

'property' : a trivial property cannot be indexed

A property was declared incorrectly. Accessors must be defined for an indexed property. See property for more information.

Example

The following sample generates C3289.

```
// C3289.cpp
// compile with: /clr
public ref struct C {
   // user-defined simple indexer
   property int indexer1[int];   // C3289

   // user-defined indexer
   property int indexer2[int] {
      int get(int i) { return 0; }
      void set(int i, int j) {}
   }
};

int main() {
   C ^ MyC = gcnew C();
   MyC->indexer2[0] = 1;
}
```

Compiler Error C3290

Error Message
'type' : a trivial property cannot have reference type

A property was declared incorrectly. When you declare a trivial property, the compiler creates a variable that the property will update, and it is not possible to have a tracking reference variable in a class.

See property and % (Tracking Reference) for more information.

Example
The following sample generates C3290.

```
// C3290.cpp
// compile with: /clr /c
ref struct R {};

ref struct X {
   R^ mr;

   property R % y;   // C3290
   property R ^ x;   // OK

   // OK
   property R% prop {
      R% get() {
         return *mr;
      }

      void set(R%) {}
   }
};

int main() {
   X x;
   R% xp = x.prop;
}
```

Compiler Error C3291

Error Message
'default' : cannot be the name of a trivial property

A trivial property cannot be named **default**. See property for more information.

Example
The following sample generates C3291.

```
// C3291.cpp
// compile with: /clr /c
ref struct C {
   property System::String ^ default;   // C3291
   property System::String ^ Default;   // OK
};
```

Compiler Error C3292

Error Message
the cli namespace cannot be reopened

The cli namespace cannot be declared in your code. For more information, see cli Namespace.

Example
The following sample generates C3292.

```
// C3292.cpp
// compile with: /clr /c
namespace cli {};   // C3292
```

Compiler Error C3293

Error Message

'accessor': use 'default' to access the default property (indexer) for class 'type'

An indexed property was accessed incorrectly. See How to: Use Indexed Properties for more information.

Example

The following sample generates C3293.

```
// C3293.cpp
// compile with: /clr /c
using namespace System;
ref class IndexerClass {
public:
   // default indexer
   property int default[int] {
      int get(int index) { return 0; }
      void set(int index, int value) {}
   }
};

int main() {
   IndexerClass ^ ic = gcnew IndexerClass;
   ic->Item[0] = 21;   // C3293
   ic->default[0] = 21;   // OK

   String ^s = "Hello";
   wchar_t wc = s->Chars[0];   // C3293
   wchar_t wc2 = s->default[0];   // OK
   Console::WriteLine(wc2);
}
```

Compiler Error C3294

Error Message
this syntax is not support with /clr:oldSyntax: please use 'function' instead

Features available in the new syntax for CLR programming are not available when compiling for Managed Extensions for C++.

For more information, see Destructors and Finalizers in Visual C++.

Example
The following sample generates C3294.

```
// C3294.cpp
// compile with: /clr:oldSyntax /c
__gc class R {
protected:
   !R() {}   // C3294
   R() {}   // OK
   ~R() {}   // OK
};
```

Compiler Error C3295

Error Message

'#pragma pragma' can only be used at global or namespace scope

Some pragmas cannot be used in a function. See Pragma Directives for more information.

Example

The following sample generates C3295.

```
// C3295.cpp
int main() {
   #pragma managed   // C3295
}
```

Compiler Error C3296

Error Message

'property' : a property with this name already exists

The compiler encountered more than one property with the same name. Each property in a type must have a unique name.

For more information, see property.

Example

The following sample generates C3296.

```
// C3296.cpp
// compile with: /clr /c
using namespace System;

ref class R {
public:
   property int MyProp[int] { int get(int); }

   property String^ MyProp[int] { void set(int, String^); }   // C3296
};
```

Compiler Error C3297

Error Message

'constraint_2' : cannot use 'constraint_1' as a constraint because 'constraint_1' has the value constraint

Value classes are sealed. If a constraint is a value class, another constraint can never derive from it.

For more information, see Constraints.

Example

The following sample generates C3297.

```
// C3297.cpp
// compile with: /clr /c
generic<class T, class U>
where T : value class
where U : T   // C3297
public ref struct R {};
```

Compiler Error C3298

Error Message

'constraint_1' : cannot use 'constraint_2' as a constraint because 'constraint_2' has the ref constraint and 'constraint_1' has the value constraint

You cannot specify mutually exclusive characteristics for a constraint. For example, a generic type parameter cannot be constrained to both a value type and a reference type.

For more information, see Constraints.

Example

The following sample generates C3298.

```
// C3298.cpp
// compile with: /clr /c
generic<class T, class U>
where T : ref class
where U : T, value class   // C3298
public ref struct R {};
```

Compiler Error C3299

Error Message
'member_function' : cannot specify constraints, they are inherited from the base method

When overriding a generic member function, you cannot specify constraint clauses (repeating the constraints implies that the constraints are not inherited).

The constraint clauses on the generic function you are overriding will be inherited.

For more information, see Constraints.

Example
The following sample generates C3299.

```
// C3299.cpp
// compile with: /clr /c
public ref struct R {
   generic<class T>
   where T : R
   virtual void f();
};

public ref struct S : R {
   generic<class T>
   where T : R   // C3299
   virtual void f() override;
};
```

Compiler Error C3303

Error Message
'attribute': attribute can only be used on 'usage'

An attempt was made to use an attribute where it is not valid.

Compiler Error C3304

Error Message
'attribute': attribute does not allow unnamed arguments

Source annotation attributes do not support unnamed attribute arguments. The attributes must be named.

Compiler Error C3305

Error Message

'attribute' : attribute usage is not allowed on 'construct'

An attribute block was not properly formed.

Compiler Error C3309

Error Message
'macro_name': module name cannot be a macro or a keyword

The value that you pass to the name property of the module attribute cannot be a symbol for the preprocessor to expand; it must be a string literal.

The following sample generates C3309:

```
// C3309.cpp
#define NAME MyModule
[module(name="NAME")];   // C3309
// Try the following line instead
// [module(name="MyModule")];
[coclass]
class MyClass {
public:
   void MyFunc();
};

int main() {
}
```

Compiler Error C3320

Error Message
'type': type cannot have the same name as the module 'name' property

An exported user-defined type (UDT), which could be a struct, class, enum, union or __value, cannot have the same name as the parameter passed to the module attribute's name property.

The following sample generates C3320:

```
// C3320.cpp
#include "unknwn.h"
[module(name="xx")];

[export] struct xx {   // C3320
// Try the following line instead
// [export] struct yy {
   int i;
};
```

Compiler Error C3322

Error Message

'property': is not a valid property for attribute 'attribute'

An invalid or unrecognized property (or parameter) was passed to an attribute. Check the documentation for the attribute.

Compiler Error C3333

Error Message
'type library': cannot #import corrupt type library

The type library specified in the **#import** statement is unreadable by the compiler. You may want to either regenerate the type library, if possible, or request a new type library from your supplier. You may want to use the OLE Viewer, supplied with Visual C++, to view the type library file to see what is the matter with it.

Compiler Error C3334

Error Message
cannot #import corrupt type library

The library file in a **#import** statement is corrupt and can't be consumed by the compiler. Corruption can come from bad generation of the module, disk corruption, or if the binary file was somehow edited and modified.

For more information, see The #import Directive.

Compiler Error C3340

Error Message
'interface': interface cannot be both 'restricted' and 'default' in coclass 'class'

The restricted attribute and the default attribute are mutually exclusive.

The following sample generates C3340:

```
// C3340.cpp
#include <windows.h>
[module(name="MyModule")];

[ object, uuid(373a1a4c-469b-11d3-a6b0-00c04f79ae8f) ]
__interface IMyIface
{
   HRESULT f1();
};

[ coclass, uuid(373a1a4d-469b-11d3-a6b0-00c04f79ae8f),
default(IMyIface),
source(IMyIface),restricted(IMyIface) ]
class CmyClass // C3340
{
};

int main()
{
}
```

Compiler Error C3342

Error Message
'attribute': ambiguous attribute

The compiler found more than one definition of an attribute.

An attribute was defined more than once.

For more information, see User-Defined Attributes.

Example
The following sample generates C3342.

```
// C3342.cpp
// compile with: /clr /c
using namespace System;
using namespace System::Reflection;

[AttributeUsage(AttributeTargets::All)]
public ref class XAttribute : public  Attribute {};

[AttributeUsage(AttributeTargets::All)]
public ref class X : public Attribute {};

[X]   // C3342 could refer to X or XAttribute
// try the following line instead
// [XAttribute]
public ref class Class4 {};
```

Compiler Error C3345

Error Message
'identifier': invalid identifier for module name

The *identifier* for a module contains one or more unacceptable characters. An identifier is valid if the first character is an alphabetical, underscore, or high ANSI (0x80-FF) character, and any subsequent character is an alphanumeric, underscore, or high ANSI character.

To correct this error

- Ensure that *identifier* does not contain blanks or other unacceptable characters.

Example

The following code example causes error message C3345 because the `name` parameter of the `module` attribute contains a blank.

```
// cpp_attr_name_module.cpp
// compile with: /LD /link /OPT:NOREF
#include <atlbase.h>
#include <atlcom.h>
#include <atlwin.h>
#include <atltypes.h>
#include <atlctl.h>
#include <atlhost.h>
#include <atlplus.h>

// C3345 expected
[module(dll, name="My Library", version="1.2", helpfile="MyHelpFile")]
// Try the following line instead
//[module(dll, name="MyLibrary", version="1.2", helpfile="MyHelpFile")]
// Module attribute now applies to this class
class CMyClass {
public:
BOOL WINAPI DllMain(DWORD dwReason, LPVOID lpReserved) {
   // add your own code here
   return __super::DllMain(dwReason, lpReserved);
   }
};
```

See Also

Reference
__iscsym
Character Classification
module (C++)
Other Resources
ASCII Character Codes

Compiler Error C3347

Error Message

'arg': required argument is not specified in attribute idl_module

A required argument was not passed to the idl_module attribute.

The following sample generates C3347:

```
// C3347.cpp
// compile with: /c
[module(name="xx")];

[idl_module(dllname="x")];    // C3347
// try the following line instead
// [idl_module(name="test", dllname="x")];
```

Compiler Error C3350

Error Message
'delegate' : a delegate constructor expects number argument(s)

When you create an instance of a delegate, you must pass two arguments, an instance of the type containing the delegate function, and the function.

The following sample generates C3350:

```
// C3350.cpp
// compile with: /clr
delegate void SumDelegate();

public ref class X {
public:
   void F() {}
   static void F2() {}
};

int main() {
   X ^ MyX = gcnew X();
   SumDelegate ^ pSD = gcnew SumDelegate();   // C3350
   SumDelegate ^ pSD1 = gcnew SumDelegate(MyX, &X::F);
   SumDelegate ^ pSD2 = gcnew SumDelegate(&X::F2);
}
```

Managed Extensions for C++

The following sample generates C3350:

```
// C3350b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
__delegate void SumDelegate();

public __gc class X {
public:
   void F() {}
   static void F2() {}
};

int main() {
   X * MyX = new X();
   SumDelegate *pSD = new SumDelegate();   // C3350
   SumDelegate *pSD1 = new SumDelegate(MyX, &X::F);
   SumDelegate *pSD2 = new SumDelegate(&X::F2);
}
```

Compiler Error C3351

Error Message
'object' : delegate constructor: second argument must be address of a static member function or global function

The compiler expected the address of a function declared static.

The following sample generates C3351:

```
// C3351a.cpp
// compile with: /clr
delegate int D(int, int);

ref class C {
public:
   int mf(int, int) {
      return 1;
   }

   static int mf2(int, int) {
      return 1;
   }
};

int main() {
   System::Delegate ^pD = gcnew D(nullptr, &C::mf);   // C3351
   System::Delegate ^pD2 = gcnew D(&C::mf2);
}
```

The following sample generates C3351:

```
// C3351b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__delegate int D(int, int);

__gc class C {
public:
   int mf(int, int) {
   // declare the function as static
   // static int mf(int, int) {
      return 1;
   }
};

int main() {
   C *pC = new C;
   System::Delegate *pD = new D(0, &C::mf);   // C3351
}
```

Compiler Error C3352

Error Message
'function' : the specified function does not match the delegate type 'type'

The parameter lists for *function* and the delegate do not match.

For more information, see delegate.

The following sample generates C3352:

```
// C3352.cpp
// compile with: /clr
delegate int D( int, int );
ref class C {
public:
   int mf( int ) {
      return 1;
   }

   // Uncomment the following line to resolve.
   // int mf(int, int) { return 1; }
};

int main() {
   C^ pC = gcnew C;
   System::Delegate^ pD = gcnew D( pC, &C::mf );   // C3352
}
```

The following sample generates C3352:

```
// C3352_2.cpp
// compile with: /clr:oldSyntax
__delegate int D(int, int);

__gc class C {
public:
   int mf(int) {
      return 1;
   }

   // Uncomment the following line to resolve.
   // int mf(int, int) { return 1; }
};

int main() {
   C *pC = new C;
   System::Delegate *pD = new D(pC, &C::mf);   // C3352
}
```

Compiler Error C3353

Error Message
'delegate' : a delegate can only be created from a global function or a member function of a managed type

Delegates, declared with the __delegate or delegate keyword, can only be declared at global scope.

The following sample generates C3353:

```
// C3353.cpp
// compile with: /clr
delegate int f;   // C3353
```

Compiler Error C3354

Error Message
'function' : the function used to create a delegate cannot have return type 'type'

The following types are invalid as return types for a delegate:

- Pointer to function
- Pointer to member
- Pointer to member function
- Reference to function
- Reference to member function

The following sample generates C3354:

```
// C3354_2.cpp
// compile with: /clr /c
using namespace System;
typedef void ( *VoidPfn )();

delegate VoidPfn func(); // C3354
// try the following line instead
// delegate  void func();
```

The following sample generates C3354:

```
// C3354.cpp
// compile with: /clr:oldSyntax /c
#using <mscorlib.dll>
using namespace System;

typedef void (*VoidPfn)();

__delegate VoidPfn func();   // C3354
// try the following line instead
// __delegate void func();
```

Compiler Error C3356

Error Message

'attribute': cannot call a multicast attribute with a fully qualified name

An attribute that is processed by more than one process, for example, the compiler and ATL provider, was specified incorrectly.

Compiler Error C3358

Error Message
'symbol': symbol not found

The required symbol was not found.

The following sample generates C3358:

```
// C3358.cpp
#define __ATLEVENT_H__ 1   // remove this line to resolve the error
#define _ATL_ATTRIBUTES 1
#include "atlbase.h"
#include "atlcom.h"

[event_receiver(com)]
struct A   // C3358
{
   void func();
};

int main()
{
}
```

Compiler Error C3360

Error Message
'string': cannot create name

The value that was passed to the uuid attribute was not valid.

The following sample generates C3360:

```
// C3360.cpp
[ uuid("1") ]
// try this line instead
// [ uuid("12341234-1234-1234-1234-123412341234") ]
struct A   // C3360
{
};

int main()
{
}
```

Compiler Error C3363

Error Message

'type' : 'typeid' can only be applied to a type

The typeid operator was used incorrectly.

Example

The following sample generates C3363.

```
// C3363.cpp
// compile with: /clr
int main() {
   System::typeid;   // C3363
}
```

Compiler Error C3364

Error Message
'delegate': delegate constructor: argument must be pointer to member function of managed class or global function

The second parameter of the delegate's constructor takes either the address of a member function or the address of a static member function of any class. Both are treated as simple addresses.

The following sample generates C3364:

```
// C3364_2.cpp
// compile with: /clr

delegate int D( int, int );

ref class C {
public:
   int mf( int, int ) {
      return 1;
   }
};

int main() {
   C^ pC = gcnew C;
   System::Delegate^ pD = gcnew D( pC,pC->mf( 1, 2 ) ); // C3364

   // try the following line instead
   // System::Delegate^ pD = gcnew D(pC, &C::mf);
}
```

The following sample generates C3364:

```
// C3364.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__delegate int D(int, int);

__gc class C {
public:
   int mf(int, int) {
      return 1;
   }
};

int main() {
   C *pC = new C;
   System::Delegate *pD = new D(pC, pC->mf(1, 2));   // C3364
   // try the following line instead
   // System::Delegate *pD = new D(pC, &C::mf);
}
```

Compiler Error C3365

Error Message

operator 'operator' : differing operands of type 'type1' and 'type2'

An attempt was made to compose delegates with different types. See How to: Compose Delegates for more information about delegates.

The following sample generates C3365:

```
// C3365.cpp
// compile with: /clr
delegate void D1();
delegate void D2(int);

ref class R {
public:
   void f(){}
   void g(int){}
};

int main() {
   D1^ d1 = gcnew D1(gcnew R, &R::f);
   D2^ d2 = gcnew D2(gcnew R, &R::g);
   D1^ d3 = gcnew D1(gcnew R, &R::f);

   d1 += d2;   // C3365
   d1 += d3;   // OK
   d1();
}
```

Compiler Error C3366

Error Message
'variable' : static data members of managed types must be defined within the class definition

You attempted to reference a static member of a .NET class or interface outside the definition of that class or interface.

The compiler needs to know the full definition of the class (to emit the meta-data after one pass) and requires static data members to be initialized within the class.

For example, the following example generates C3366:

```
// C3366.cpp
// compile with: /clr /c
ref class X {
   public:
   static int i;   // initialize i here to avoid C3366
};

int X::i = 5;      // C3366
```

The following example generates C3366:

```
// C3366_b.cpp
// compile with: /clr:oldSyntax /c
__gc struct X {
   static int i;   // initialize i here to avoid C3366
};

int X::i = 5;      // C3366
```

Compiler Error C3367

Error Message
'static_member_function' : cannot use static function to create an unbound delegate

When you call an unbound delegate, you must pass an instance of an object. Since a static member function is called through the class name, you can only instantiate an unbound delegate with an instance member function.

For more information, see Unbound Delegates.

Example
The following sample generates C3367.

```
// C3367.cpp
// compile with: /clr
ref struct R {
   void b() {}
   static void f() {}
};

delegate void Del(R^);

int main() {
   Del ^ a = gcnew Del(&R::b);   // OK
   Del ^ b = gcnew Del(&R::f);   // C3367
}
```

Compiler Error C3368

Error Message
'function declaration' : invalid calling convention for IDL

You can only use the __stdcall or __cdecl calling conventions in an .idl file.

The following sample generates C3368:

```
// C3368.cpp
// processor: x86
[idl_module(name="Name", dllname="Some.dll")];

[idl_module(name="Name")]
int __fastcall f1();   // C3368
```

Compiler Error C3369

Error Message
'module name': idl_module already defined

The idl_module usage where you define the DLL can only occur once in a program.

The following sample generates C3369:

```
// C3369.cpp
// compile with: /c
[idl_module(name="name1", dllname="x.dll")]; // C3369
[idl_module(name="name1", dllname="x.dll")];
```

Compiler Error C3370

Error Message
'idl_module name': idl_module not yet defined

Before you can use idl_module to specify an entry point in a DLL, you must first use **idl_module** to specify the DLL name.

The following sample generates C3370:

```
// C3370.cpp
[module(name=MyLibrary)];
// uncomment the following line to resolve the error
// [idl_module(name="name1", dllname=x.dll)];
[idl_module(name="name1"), entry(100)] // C3370
int f1();

int main()
{
}
```

Compiler Error C3371

Error Message
'idl_module': only the 'name' property is allowed here

idl_module usage directly on a function declaration cannot have any parameters other than name.

The following sample generates C3371:

```
// C3371.cpp
[idl_module(name="Name", dllname="Some.dll")];
[idl_module(name="Name", helpstring="Some help")]   // C3371
int f1();
// try
// [idl_module(name="Name")]
// int f1();

int main()
{
}
```

Compiler Error C3372

Error Message
must specify at least 1 interface for attribute 'source' on a coclass

For certain attributes, you must pass an interface name as a parameter.

The following sample generates C3372:

```
// C3372.cpp
#include <windows.h>
[module(name="MyModule")];

[ object, uuid(373a1a4c-469b-11d3-a6b0-00c04f79ae8f) ]
__interface IMyIface {
   HRESULT f1();
};
// to resolve, pass an interface name to the source attribute
// for example, source(IMyIface)
[ coclass, uuid(373a1a4d-469b-11d3-a6b0-00c04f79ae8f), source,
  default(IMyIface) ] // C3372
class CMyClass {
};

int main() {
}
```

Compiler Error C3373

Error Message
attribute 'attribute' takes no arguments except on a coclass

Some attributes can be applied to more than one C++ construct, but arguments to the attribute may only be allowed on some constructs.

The following sample generates C3373:

```
// C3373.cpp
#include <windows.h>

[module(name="MyModule")];

[ object, uuid(373a1a4c-469b-11d3-a6b0-00c04f79ae8f) ]
__interface IMyIface
{
   // arguments to source and restricted are not allowed when
   // these attributes are applied to an interface
   [source(IMyIface)] HRESULT f1();
   [restricted(IMyIface)] HRESULT f2(); // C3373
};

[ coclass, uuid(373a1a4d-469b-11d3-a6b0-00c04f79ae8f) ]
class CMyClass : public IMyIface {
};

int main() {
}
```

Compiler Error C3374

Error Message
can't take address of 'function' unless creating delegate instance

The address of a function was taken in a context other than the creation of a delegate instance.

The following sample generates C3374:

```
// C3374.cpp
// compile with: /clr
public delegate void MyDel(int i);

ref class A {
public:
   void func1(int i) {
      System::Console::WriteLine("in func1 {0}", i);
   }
};

int main() {
   &A::func1;   // C3374

   // OK
   A ^ a = gcnew A;
   MyDel ^ StaticDelInst = gcnew MyDel(a, &A::func1);
}
```

Compiler Error C3375

Error Message
'function' : ambiguous delegate function

A delegate instantiation could have been to a static member function, or as an unbound delegate to an instance function, so the compiler issued this error.

For more information, see delegate.

Example
The following sample generates C3375.

```
// C3375.cpp
// compile with: /clr
ref struct R {
   static void f(R^) {}
   void f() {}
};

delegate void Del(R^);

int main() {
   Del ^ a = gcnew Del(&R::f);   // C3375
}
```

Compiler Error C3379

Error Message
'class' : a nested class cannot have an assembly access specifier as part of its declaration

When applied to a managed type, such as class or struct, the public and private keywords indicate whether the class will be exposed through assembly metadata. **public** or **private** cannot be applied to a nested class, which will inherit the assembly access of the enclosing class.

When used with /clr, the **ref** and **value** keywords indicate that a class is managed (see Classes and Structs (Managed)).

The following sample generates C3379:

```
// C3379a.cpp
// compile with: /clr
using namespace System;

public ref class A {
public:
   static int i = 9;

   public ref class BA {   // C3379
   // try the following line instead
   // ref class BA {
   public:
      static int ii = 8;
   };
};

int main() {

   A^ myA = gcnew A;
   Console::WriteLine(myA->i);

   A::BA^ myBA = gcnew A::BA;
   Console::WriteLine(myBA->ii);
}
```

The following sample generates C3379:

```
// C3379b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

public __gc class A {
public:
   static int i = 9;

   public __gc class BA {   // C3379
   // try the following line instead
   // __gc class BA {
   public:
      static int ii = 8;
   };
};

int main() {

   A *myA = new A;
   Console::WriteLine(myA->i);

   A::BA *myBA = new A::BA;
   Console::WriteLine(myBA->ii);
}
```

Compiler Error C3380

Error Message
'class' : invalid assembly access specifier - only 'public' or 'private' are allowed

When applied to a managed class or struct, the public and private keywords indicate whether the class will be exposed through assembly metadata. Only **public** or **private** can be applied to a class in a program compiled with /clr.

The **ref** and **value** keywords, when used with /clr, indicate that a class is managed (see Classes and Structs (Managed)).

The following sample generates C3380:

```
// C3380_2.cpp
// compile with: /clr
protected ref class A {   // C3380
// try the following line instead
// ref class A {
public:
   static int i = 9;
};

int main() {
   A^ myA = gcnew A;
   System::Console::WriteLine(myA->i);
}
```

The following sample generates C3380:

```
// C3380.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
protected __gc class A {   // C3380
// try the following line instead
// __gc class A {
public:
   static int i = 9;
};

int main() {
   A *myA = new A;
   Console::WriteLine(myA->i);
}
```

Compiler Error C3381

Error Message
'assembly' : assembly access specifiers are only available in code compiled with a /clr option

Native types can be visible outside the assembly, but you can only specify assembly access for native types in a **/clr** compilation.

For more information, see Type Visibility and /clr (Common Language Runtime Compilation).

Example
The following sample generates C3381.

```
// C3381.cpp
// compile with: /c
public class A {};   // C3381
```

Compiler Error C3382

Error Message
'sizeof' is not supported with /clr:safe

The output file of a **/clr:safe** compilation is a file that is verifiably type safe, and sizeof is not supported because the return value of the sizeof operator is size_t, whose size varies depending on the operating system.

For more information, see,

- sizeof Operator
- /clr (Common Language Runtime Compilation)
- Common Visual C++ 64-bit Migration Issues

Example
The following sample generates C3382.

```
// C3382.cpp
// compile with: /clr:safe
int main() {
   sizeof( char );   // C3382
}
```

Compiler Error C3383

Error Message
'operator new' is not supported with /clr:safe

The output file of a **/clr:safe** compilation is a file that is verifiably type safe, and pointers are not supported.

For more information, see,

- /clr (Common Language Runtime Compilation)
- Common Visual C++ 64-bit Migration Issues

Example
The following sample generates C3383.

```
// C3383.cpp
// compile with: /clr:safe
int main() {
   char* pCharArray = new char[256];  // C3383
}
```

Compiler Error C3384

Error Message

'type_parameter' : the value constraint and the ref constraint are mutually exclusive

You cannot constrain a generic type to both **value class** and **ref class**.

See Constraints for more information.

Example

The following sample generates C3384.

```
// C3384.cpp
// compile with: /c /clr
generic <typename T>
where T : ref class
where T : value class   // C3384
ref class List {};
```

Compiler Error C3385

Error Message
'class::function' : a function that has a DllImport Custom attribute cannot return an instance of a class

A function defined as being in a .dll file specified with the **DllImport** attribute cannot return an instance of a class.

The following sample generates C3385:

```
// C3385.cpp
// compile with: /clr /c
using namespace System;
using namespace System::Runtime::InteropServices;

struct SomeStruct1 {};

public ref struct Wrap {
   [ DllImport("somedll.dll", CharSet=CharSet::Unicode) ]
   static SomeStruct1 f1([In, Out] SomeStruct1 *pS);   // C3385
};
```

The following sample generates C3385:

```
// C3385_2.cpp
// compile with: /clr:oldSyntax /c

using namespace System;
using namespace System::Runtime::InteropServices;

struct SomeStruct1 {};

public __gc struct Wrap {
   [ DllImport("somedll.dll", CharSet=CharSet::Unicode) ]
   static SomeStruct1 f1([In, Out] SomeStruct1 *pS);   // C3385
};
```

Compiler Error C3386

Error Message
'type' : __declspec(dllexport)/__declspec(dllimport) cannot be applied to a managed type

The **dllimport** and dllexport **__declspec** modifiers are not valid on a managed type.

The following sample generates C3386:

```
// C3386a.cpp
// compile with: /clr /c
ref class __declspec(dllimport) X1 {   // C3386
// try the following line instead
// ref class X1 {
};
```

The following sample generates C3386:

```
// C3386b.cpp
// compile with: /clr:oldSyntax /c
#using <mscorlib.dll>
using namespace System;

__gc class __declspec(dllimport) X1 {   // C3386
// try the following line instead
// __gc class X1 {
};
```

Compiler Error C3387

Error Message
'member' : __declspec(dllexport)/__declspec(dllimport) cannot be applied to a member of a managed type

The **dllimport** and dllexport **__declspec** modifiers are not valid on members of a managed type.

The following sample generates C3387:

```
// C3387a.cpp
// compile with: /clr /c
ref class X2 {
   void __declspec(dllexport) mf() {   // C3387
   // try the following line instead
   // void mf() {
   }
};
```

The following sample generates C3387:

```
// C3387b.cpp
// compile with: /clr:oldSyntax /c
#using <mscorlib.dll>
using namespace System;

__gc class X2 {
   void __declspec(dllexport) mf() {   // C3387
   // try the following line instead
   // void mf() {
   }
};
```

Compiler Error C3388

Error Message

'type' : not allowed as a constraint, assuming 'ref class' to continue parsing

A constraint was specified on a generic type, but the constraint was not specified correctly. See Constraints for more information.

Example

The following sample generates C3388.

```
// C3388.cpp
// compile with: /clr /c
interface class AA {};

generic <class T>
where T : interface class   // C3388
ref class C {};

// OK
generic <class T>
where T : AA
ref class D {};
```

Compiler Error C3389

Error Message
__declspec(keyword) cannot be used with /clr:pure or /clr:safe

A __declspec modifier used implies a per process state. /clr:pure implies a per appdomain state. So, declaring a variable with the *keyword* **__declspec** modifier and compiling with **/clr:pure** is not allowed.

The following sample generates C3389:

```
// C3389.cpp
// compile with: /clr:pure /c
__declspec(dllexport) int g2 = 0;   // C3389
```

Compiler Error C3390

Error Message

'type_arg' : invalid type argument for generic parameter 'param' of generic 'generic_type', must be a reference type

A generic type was instantiated incorrectly. Check the type definition. For more information, see Generics (Visual C++).

Example

The following sample, using C#, creates a component that contains a generic type, with certain constraints that are not supported when authoring generic types in Visual C++ 2005. For more information, see .Constraints on Type Parameters (C# Programming Guide).

```
// C3390.cs
// compile with: /target:library
// a C# program
public class GR<C, V, N>
where C : class
where V : struct
where N : new() {}
```

The following sample generates C3390.

```
// C3390_b.cpp
// compile with: /clr
#using <C3390.dll>
ref class R { R(int) {} };
value class V {};
ref struct N { N() {} };

int main () {
   GR<V, V, N^>^ gr2;   // C3390 first type must be a ref type
   GR<R^, V, N^>^ gr2b;   // OK
}
```

Compiler Error C3391

Error Message

'type_arg' : invalid type argument for generic parameter 'param' of generic 'generic_type', must be a non-nullable value type

A generic type was instantiated incorrectly. Check the type definition. For more information, see Nullable and Generics (Visual C++).

Example

The following sample, using C#, creates a component that contains a generic type, with certain constraints that are not supported when authoring generic types in Visual C++ 2005. For more information, see .Constraints on Type Parameters (C# Programming Guide).

```
// C3391.cs
// compile with: /target:library
// a C# program
public class GR<N>
where N : struct {}
```

The following sample generates C3391.

```
// C3391_b.cpp
// compile with: /clr
#using <C3391.dll>
using namespace System;
value class VClass {};

int main() {
   GR< Nullable<VClass> >^ a =
      gcnew GR< Nullable<VClass> >();   // C3391 can't be Nullable
   GR<VClass>^ aa = gcnew GR<VClass>();   // OK
}
```

Compiler Error C3392

Error Message

'type_arg' : invalid type argument for generic parameter 'param' of generic 'generic_type', must have a public parameterless constructor

A generic type was instantiated incorrectly. Check the type definition. For more information Generics (Visual C++).

Example

The following sample, using C#, creates a component that contains a generic type, with certain constraints that are not supported when authoring generic types in Visual C++ 2005. For more information, see .Constraints on Type Parameters (C# Programming Guide).

```
// C3392.cs
// compile with: /target:library
// a C# program
public class GR<C, V, N>
where C : class
where V : struct
where N : new() {}
```

The following sample generates C3392.

```
// C3392_b.cpp
// compile with: /clr
#using <C3392.dll>

ref class R { R(int) {} };
ref class N { N() {} };

value class V {};

ref class N2 { public: N2() {} };
ref class R2 { public: R2() {} };

int main () {
   GR<R^, V, N^>^ gr1;   // C3392
   GR<R^, V, N2^>^ gr1a;   // OK

   GR<R^, N^, N^>^ gr3;   // C3392
   GR<R^, V, N2^>^ gr3a;   // OK

   GR<R^, V, R^>^ gr4;   // C3392
   GR<R^, V, R2^>^ gr4a;   // OK
}
```

Compiler Error C3393

Error Message

syntax error in constraint clause: 'identifier' is not a type

The identifier passed to a constraint, which must be a type, was not a type. For more information, see Constraints.

Example

The following sample generates C3393:

```
// C3393.cpp
// compile with: /clr /c
void MyInterface() {}
interface class MyInterface2 {};

generic<typename T>
where T : MyInterface   // C3393
// try the following line instead
// where T : MyInterface2
ref class R {};
```

Compiler Error C3394

Error Message
syntax error in constraint clause: found 'identifier' expected a type

A constraint was ill formed. For more information, see Constraints.

Example
The following sample generates C3394:

```
// C3394.cpp
// compile with: /clr /c
ref class MyClass {};
ref class R {
   generic<typename T>
   where T : static   // C3394
   // try the following line instead
   // where T : MyClass
   void f() {}
};
```

Compiler Error C3395

Error Message
'function' : __declspec(dllexport) cannot be applied to a function with the __clrcall calling convention

__declspec(dllexport) and __clrcall are not compatible. For more information, see dllexport, dllimport.

The following sample generates C3395:

```
// C3395.cpp
// compile with: /clr /c

__declspec(dllexport) void __clrcall Test(){}   // C3395
void __clrcall Test2(){}   // OK
__declspec(dllexport) void Test3(){}   // OK
```

Compiler Error C3396

Error Message
'attribute' : custom attribute not found in 'namespace'

C3396 indicates that you are not using an up-to-date common language runtime. Reinstall your CLR to resolve.

Compiler Error C3397

Error Message

Aggregate initialization is not allowed in default arguments

An array was declared incorrectly. See array (Visual C++) for more information.

Example

The following sample generates C3397.

```
// C3397.cpp
// compile with: /clr
// /clr /c
void Func(array<int> ^p = gcnew array<int> { 1, 2, 3 });   // C3397
void Func2(array<int> ^p = gcnew array<int> (3));   // OK

int main() {
   array<int> ^p = gcnew array<int> { 1, 2, 3};   // OK
}
```

Compiler Error C3398

Error Message
'operator' : cannot convert from 'function_signature' to 'function_pointer'. Source expression must be a function symbol

When the __clrcall calling convention is not specified when compiling with **/clr**, the compiler generates two entry points (addresses) for each function, a native entry point and a managed entry point.

By default the compiler returns the native entry point, but there are some cases where the managed entry point is desired (for instance when assigning the address to a **__clrcall** function pointer). In order for the compiler to reliably choose the managed entry point in an assignment, the right hand side must be a function symbol.

Compiler Error C3399

Error Message

'type' : cannot provide arguments when creating an instance of a generic parameter

When you specify the `gcnew()` constraint, you specify that the constraint type will have a parameterless constructor. Therefore, it is an error to attempt to instantiate that type and pass a parameter.

See Constraints for more information.

Example

The following sample generates C3399.

```
// C3399.cpp
// compile with: /clr /c
generic <class T>
where T : gcnew()
void f() {
   T t = gcnew T(1);   // C3399
   T t2 = gcnew T();   // OK
}
```

Compiler Error C3400

Error Message
circular constraint dependency involving 'constraint_1' and 'constraint_2'

The compiler detected circular constraints.

For more information, see Constraints.

Example
The following sample generates C3400.

```
// C3400.cpp
// compile with: /clr /c
generic<class T, class U>
where T : U
where U : T   // C3400
public ref struct R {};
```

Compiler Error C3408

Error Message
'attribute': attribute is not allowed on template definitions

Attributes cannot be applied to template definitions.

Example
The following sample generates C3408.

```
// C3408.cpp
// compile with: /c
template <class T> struct PTS {
   enum {
      IsPointer = 0,
      IsPointerToDataMember = 0
   };
};

template <class T>
[coclass]   // C3408
struct PTS<T*> {
   enum {
      IsPointer = 1,
      IsPointerToDataMember = 0
   };
};

template
<class T, class U>
struct PTS<T U::*> {
   enum {
      IsPointer = 1,
      IsPointerToDataMember = 1
   };
};

struct S{};

extern "C" int printf(const char*,...);

int main() {
   S s, *pS;
   int S::*ptm;

   printf("PTS<S>::IsPointer == %d PTS<S>::IsPointerToDataMember == %d\n", PTS<S>::IsPointe
r, PTS<S>:: IsPointerToDataMember);
   printf("PTS<S*>::IsPointer == %d PTS<S*>::IsPointerToDataMember == %d\n", PTS<S*>::IsPoi
nter, PTS<S*>:: IsPointerToDataMember);
   printf("PTS<int S::*>::IsPointer == %d PTS<int S::*>::IsPointerToDataMember == %d\n", PT
S<int S::*>::IsPointer, PTS<int S::*>:: IsPointerToDataMember);
}
```

Compiler Error C3409

Error Message
empty attribute block is not allowed

The square brackets were interpreted by the compiler as an attribute block, but no attributes were found.

Example
The following sample generates C3409.

```
// C3409.cpp
// compile with: /c
#include <windows.h>
[]   // C3409
class a {};

// OK
[object, uuid("00000000-0000-0000-0000-000000000000")]
__interface x {};

[coclass, uuid("00000000-0000-0000-0000-000000000001")]
class b : public x {};
```

Compiler Error C3412

Error Message
'template' : cannot specialize template in current scope

A template cannot be specialized at class scope, only in global or namespace scope.

Example
The following sample generates C3412.

```
// C3412.cpp
template <class T>
struct S {
   template <>
   struct S<int> {};   // C3412 in a class
};
```

The following sample shows a possible resolution.

```
// C3412b.cpp
// compile with: /c
template <class T>
struct S {};

template <>
struct S<int> {};
```

Compiler Error C3413

Error Message
'name' : invalid explicit instantiation

The compiler detected an ill-formed explicit instantiation.

The following sample generates C3413:

```
// C3413.cpp
template
class MyClass {};   // C3413
```

Possible resolution:

```
// C3413b.cpp
// compile with: /c
template <class T>
class MyClass {};

template <>
class MyClass<int> {};
```

Compiler Error C3414

Error Message
'member' : imported member function can't be defined

A member was defined in code that is also defined in a referenced assembly.

The following sample generates C3414:

```
// C3414a2.cpp
// compile with: /clr /LD
public ref class MyClass {
public:
   void Test(){}
};
```

and then:

```
// C3414b2.cpp
// compile with: /clr
#using <C3414a2.dll>

void MyClass::Test() {    // C3414
}

System::Object::Object() {    // C3414
}
```

The following sample generates C3414:

```
// C3414a.cpp
// compile with: /clr:oldSyntax /LD
#using <mscorlib.dll>
public __gc class MyClass
{
public:
   void Test(){}
};
```

and then:

```
// C3414b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
#using <C3414a.dll>

void MyClass::Test()
{    // C3414
}

System::Object::Object()
{    // C3414
}
```

Compiler Error C3415

Error Message
multiple 'section_name' sections found with different attributes ('value')

Conflicting values were specified in section pragmas.

value is the current setting for the section, as specified in ntimage.h. For example:

```
// Section contains extended relocations.
#define IMAGE_SCN_LNK_NRELOC_OVFL            0x01000000
// Section can be discarded.
#define IMAGE_SCN_MEM_DISCARDABLE            0x02000000
// Section is not cachable.
#define IMAGE_SCN_MEM_NOT_CACHED             0x04000000
// Section is not pageable.
#define IMAGE_SCN_MEM_NOT_PAGED              0x08000000
// Section is shareable.
#define IMAGE_SCN_MEM_SHARED                 0x10000000
// Section is executable.
#define IMAGE_SCN_MEM_EXECUTE                0x20000000
// Section is readable.
#define IMAGE_SCN_MEM_READ                   0x40000000
// Section is writeable.
#define IMAGE_SCN_MEM_WRITE                  0x80000000
```

The following sample generates C3415:

```
// C3415.cpp
#pragma section("mysec1",write)
#pragma section("mysec1",read)   // C3415
```

Compiler Error C3416

Error Message
'function' : an explicit specialization may not be explicitly instantiated

A function cannot be both explicitly specialized and explicitly instantiated.

The following sample generates C3416:

```
// C3416.cpp
template <class T>
void f();

template <>
void f<int>() {}
template void f<int>();   // C3416 delete this or previous line to resolve
```

Compiler Error C3417

Error Message
'member' : value types cannot contain user-defined special member functions

Value types cannot contain functions such as a default instance constructor, destructor, or copy constructor.

The following sample generates C3517:

```
// C3417.cpp
// compile with: /clr /c
value class VC {
   VC(){}   // C3417

   // OK
   static VC(){}
   VC(int i){}
};
```

Compiler Error C3418

Error Message
access specifier 'specifier' is not supported

A CLR access specifier was specified incorrectly. For more information, see Type and Member Visibility.

Example
The following sample generates C3418.

```
// C3418.cpp
// compile with: /clr /c
ref struct m {
internal public:   // C3418
   void test(){}
};

ref struct n {
internal:   // OK
   void test(){}
};
```

Compiler Error C3420

Error Message
'finalizer' : a finalizer cannot be virtual

A finalizer can only be called non-virtually from its enclosing type. Therefore, it is an error to declare a virtual finalizer.

For more information, see Destructors and Finalizers in Visual C++.

Example
The following sample generates C3420.

```
// C3420.cpp
// compile with: /clr /c
ref class R {
   virtual !R() {}   // C3420
};
```

Compiler Error C3421

Error Message

'type' : you cannot call the finalizer for this class as it is either inaccessible or it does not exist

A finalizer is implicitly private, so it cannot be called from outside its enclosing type.

For more information, see Destructors and Finalizers in Visual C++.

Example

The following sample generates C3421.

```
// C3421.cpp
// compile with: /clr
ref class A {};

ref class B {
   !B() {}

public:
   ~B() {}
};

int main() {
   A a;
   a.!A();   // C3421

   B b;
   b.!B();   // C3421
}
```

Compiler Error C3450

Error Message

'type': not an attribute; cannot specify [System::AttributeUsageAttribute]

A user-defined attribute must inherit from AttributeCollection.

For more information, see User-Defined Attributes.

Example

The following sample generates C3450.

```
// C3450.cpp
// compile with: /clr
// C3450 expected
using namespace System;
using namespace System::Security;
using namespace System::Security::Permissions;

public ref class MyClass {};

class MyClass2 {};

[attribute(AttributeTargets::All)]
ref struct AtClass {
   AtClass(Type ^) {}
};

[attribute(AttributeTargets::All)]
ref struct AtClass2 {
   AtClass2() {}
};

// Apply the AtClass and AtClass2 attributes to class B
[AtClass(MyClass::typeid), AtClass2]
[AttributeUsage(AttributeTargets::All)]
// Delete the following line to resolve.
ref class B {};
// Uncomment the following line to resolve.
// ref class B : Attribute {};
```

Compiler Error C3451

Error Message
'attribute': cannot apply unmanaged attribute to 'type'

A C++ attribute cannot be applied to a CLR type. See C++ Attributes Reference for more information.

For more information, see User-Defined Attributes.

This error can be generated as a result of compiler conformance work that was done for Visual C++ 2005: the uuid (C++ Attributes) attribute is no longer allowed on a user-defined attribute using CLR programming. Use GuidAttribute instead.

See Breaking Changes in the Visual C++ 2005 Compiler for more information.

Example
The following sample generates C3451.

```
// C3451.cpp
// compile with: /clr /c
using namespace System;
[ attribute(AttributeTargets::All) ]
public ref struct MyAttr {};

[ MyAttr, helpstring("test") ]   // C3451
// try the following line instead
// [ MyAttr ]
public ref struct ABC {};
```

Compiler Error C3452

Error Message
list argument member not constant

An argument was passed to an attribute that expected a constant, a value that can be evaluated at compile time.

Example
The following sample generates C3452.

```
// C3452.cpp
// compile with: /c
int i;
[module( name="mod", type=dll, custom={i} ) ];   // C3452
// try the following line instead
// [module( name="mod", type=dll, custom={"a"} ) ];
```

Compiler Error C3453

Error Message

'attribute': attribute not applied because qualifier 'assembly' did not match

Assembly or module level attributes can only be specified as stand-alone instructions.

Example

The following sample generates C3453.

```
// C3453.cpp
// compile with: /clr /c
[assembly:System::CLSCompliant(true)]   // C3453
// try the following line instead
// [assembly:System::CLSCompliant(true)];
ref class X {};
```

Compiler Error C3454

Error Message
[attribute] not allowed on class declaration

A class must be defined for it to be an attribute.

For more information, see attribute.

Example
The following sample generates C3454.

```
// C3454.cpp
// compile with: /clr /c
using namespace System;

[attribute]   // C3454
ref class Attr1;

[attribute]   // OK
ref class Attr2 {};
```

Compiler Error C3455

Error Message

'attribute': none of the attribute constructors matched the arguments

An invalid value was used to declare an attribute. See attribute for more information.

Example

The following sample generates C3455.

```
// C3455.cpp
// compile with: /clr /c
using namespace System;

[attribute("MyAt")]   // C3455
// try the following line instead
// [attribute(All)]
ref struct MyAttr {
   MyAttr() {}
};
```

Compiler Error C3456

Error Message
[source_annotation_attribute] not allowed on managed class declaration

source_annotation_attribute is used to define custom attributes to be used by code analysis.

These custom attributes are only useful if there is a corresponding plugin that understands the attribute.

Compiler Error C3457

Error Message

'attribute': attribute does not support unnamed arguments

Source annotation attributes, unlike CLR custom attribute or compiler attributes, only support named arguments.

Example

The following sample generates C3457.

```
#include "SourceAnnotations.h"
[vc_attributes::Post( 1 )] int f();   // C3457
[vc_attributes::Post( Valid=vc_attributes::Yes )] int f2();   // OK
```

Compiler Error C3458

Error Message
'attribute1': attribute 'attribute2' already specified for 'construct'

Two attributes that are mutually exclusive were specified for the same construct.

Example
The following sample generates C3458

```
// C3458.cpp
// compile with: /clr /c
[System::Reflection::DefaultMember("Chars")]
public ref class MyString {
public:
   [System::Runtime::CompilerServices::IndexerName("Chars")]   // C3458
   property char default[int] {
      char get(int index);
      void set(int index, char c);
   }
};
```

Compiler Error C3459

Error Message

'attribute': attribute allowed only on class indexer (default indexed property)

An attribute that is designed to be applied to a class indexer property was used incorrectly.

For more information, see How to: Use Indexed Properties.

Example

The following sample generates C3459.

```
// C3459.cpp
// compile with: /clr /c
public ref class MyString {
public:
   [System::Runtime::CompilerServices::IndexerName("Chars")]   // C3459
   property int Prop;
};

// OK
public ref class MyString2 {
   array<int>^ MyArr;
public:
   MyString2() {
      MyArr = gcnew array<int>(5);
   }

   [System::Runtime::CompilerServices::IndexerName("Chars")]   // OK
   property int default[int] {
      int get(int index) {
         return MyArr[index];
      }
      void set(int index, int value) {
         MyArr[index] = value;
      }
   }
};
```

Compiler Error C3460

Error Message

'type': only a user-defined type can be forwarded

For more information, see Type Forwarding.

Example

The following sample creates a component.

```
// C3460.cpp
// compile with: /LD /clr
public ref class R {};
```

The following sample generates C3460.

```
// C3460_b.cpp
// compile with: /clr /c
#using "C3460.dll"
[assembly:TypeForwardedTo(int::typeid)];   // C3460
[assembly:TypeForwardedTo(R::typeid)];
```

Compiler Error C3461

Error Message

'type': only a managed type can be forwarded

Type forwarding can only occur on CLR types. See Classes and Structs (Managed) for more information.

For more information, see Type Forwarding.

Example

The following sample creates a component.

```
// C3461.cpp
// compile with: /clr /LD
public ref class R {};
```

The following sample generates C3461.

```
// C3461b.cpp
// compile with: /clr /c
#using "C3461.dll"
class N {};
[assembly:TypeForwardedTo(N::typeid)];   // C3461
[assembly:TypeForwardedTo(R::typeid)];   // OK
```

Compiler Error C3462

Error Message
'type': only an imported type can be forwarded

The TypeForwardedTo attribute must be applied to a type in referenced metadata.

For more information, see Type Forwarding.

Example
The following sample creates a component.

```
// C3462.cpp
// compile with: /clr /LD
public ref class R {};
```

The following sample generates C3462.

```
// C3462b.cpp
// compile with: /clr /c
#using "C3462.dll"
ref class N {};
[assembly:TypeForwardedTo(N::typeid)];   // C3462
[assembly:TypeForwardedTo(R::typeid)];
```

Compiler Error C3463

Error Message
'type': type not allowed in attribute 'implements'

An invalid type was passed to the implements (C++) attribute. For example, you can pass an interface to **implements**, but you cannot pass a pointer to an interface.

Example

The following sample generates C3463.

```
// C3463.cpp
// compile with: /c
#include <windows.h>
[object, uuid("00000000-0000-0000-0000-000000000001")]
__interface X {};

typedef X* PX;

[ coclass, uuid("00000000-0000-0000-0000-000000000002"), implements(interfaces=PX) ]   // C
3463
// try the following line instead
// [ coclass, uuid("00000000-0000-0000-0000-000000000002"), implements(interfaces=X) ]
class CMyClass : public X {};
```

Compiler Error C3464

Error Message
'type' a nested type cannot be forwarded

Type forwarding does not work on nested types.

For more information, see Type Forwarding.

Example
The following sample creates a component.

```
// C3464.cpp
// compile with: /LD /clr
public ref class R {
public:
   ref class N {};
};
```

The following sample generates C3464.

```
// C3464_b.cpp
// compile with: /clr /c
#using "C3464.dll"
[assembly:TypeForwardedTo(R::N::typeid)];   // C3464
[assembly:TypeForwardedTo(R::typeid)];   // OK
```

Compiler Error C3465

Error Message
to use type 'type' you must reference the assembly 'assembly'

Type forwarding will work for a client application until you recompile the client. When you recompile, you will need a reference for every assembly containing the definition of a type used in your client application.

For more information, see Type Forwarding.

Example

The following sample builds an assembly that contains the new location of a type.

```
// C3465.cpp
// compile with: /clr /LD
public ref class R {
public:
   ref class N {};
};
```

The following sample builds an assembly that used to contain the definition of the type, but now contains forwarding syntax for the type.

```
// C3465_b.cpp
// compile with: /clr /LD
#using "C3465.dll"
[ assembly:TypeForwardedTo(R::typeid) ];
```

The following sample generates C3465.

```
// C3465_c.cpp
// compile with: /clr
// C3465 expected
#using "C3465_b.dll"
// Uncomment the following line to resolve.
// #using "C3465.dll"

int main() {
   R^ r = gcnew R();
}
```

Compiler Error C3466

Error Message

'type' : a specialization of a generic class cannot be forwarded

You cannot use type forwarding on a specialization of a generic class.

For more information, see Type Forwarding.

Example

The following sample creates a component.

```
// C3466.cpp
// compile with: /clr /LD
generic<typename T>
public ref class GR {};

public ref class GR2 {};
```

The following sample generates C3466.

```
// C3466_b.cpp
// compile with: /clr /c
#using "C3466.dll"
[assembly:TypeForwardedTo(GR<int>::typeid)];   // C3466
[assembly:TypeForwardedTo(GR2::typeid)];   // OK
```

Compiler Error C3467

Error Message

'type' : this type has already been forwarded

The compiler found more than one forward type declaration for the same type. Only one declaration per type is allowed.

For more information, see Type Forwarding.

Example

The following sample creates a component.

```
// C3467.cpp
// compile with: /LD /clr
public ref class R {};
```

The following sample generates C3467.

```
// C3467_b.cpp
// compile with: /clr /c
#using "C3467.dll"
[ assembly:TypeForwardedTo(R::typeid) ];
[ assembly:TypeForwardedTo(R::typeid) ];   // C3467
```

Compiler Error C3468

Error Message

'type' : you can only forward a type to an assembly:

'*file*' is not an assembly

Only types in an assembly can be forwarded.

For more information, see Type Forwarding.

Example

The following sample creates a module.

```
// C3468.cpp
// compile with: /LN /clr
public ref class R {};
```

The following sample generates C3468.

```
// C3468_b.cpp
// compile with: /clr /c
#using "C3468.netmodule"
[ assembly:TypeForwardedTo(R::typeid) ];   // C3468
```

Compiler Error C3469

Error Message

'type' : a generic class cannot be forwarded

You cannot use type forwarding on a generic class.

For more information, see Type Forwarding.

Example

The following sample creates a component.

```
// C3469.cpp
// compile with: /clr /LD
generic<typename T>
public ref class GR {};

public ref class GR2 {};
```

The following sample generates C3466.

```
// C3469_b.cpp
// compile with: /clr /c
#using "C3469.dll"
[assembly:TypeForwardedTo(GR::typeid)];   // C3469
[assembly:TypeForwardedTo(GR2::typeid)];   // OK
```

Compiler Error C3470

Error Message

'type' : a class cannot have both an indexer (default indexed property) and an operator[]

A type cannot define both a default indexer and an operator[].

Example

The following sample generates C3470

```
// C3470.cpp
// compile with: /clr
using namespace System;

ref class R {
public:
   property int default[int] {
      int get(int i) {
         return i+1;
      }
   }

   int operator[](String^ s) { return Convert::ToInt32(s); }   // C3470
};

int main() {
   R ^ r = gcnew R;
   // return r[9] + r["32"] - 42;
}
```

Compiler Errors C3500 through C3999

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, type the error number in the **Look for** box in the Index, or click a link below.

Compiler Error C3500

Compiler Error C3501

Compiler Error C3505

Compiler Error C3506

Compiler Error C3507

Compiler Error C3508

Compiler Error C3509

Compiler Error C3510

Compiler Error C3519

Compiler Error C3603

Compiler Error C3607

Compiler Error C3609

Compiler Error C3610

Compiler Error C3611

Compiler Error C3612

Compiler Error C3616

Compiler Error C3618

Compiler Error C3619

Compiler Error C3622

Compiler Error C3623

Compiler Error C3624

Compiler Error C3625

Compiler Error C3626

Compiler Error C3627

Compiler Error C3628

Compiler Error C3630

Compiler Error C3631

Compiler Error C3632

Compiler Error C3633

Compiler Error C3634

Compiler Error C3637

Compiler Error C3638

Compiler Error C3640

Compiler Error C3641

Compiler Error C3642

Compiler Error C3644
Compiler Error C3645
Compiler Error C3646
Compiler Error C3648
Compiler Error C3650
Compiler Error C3651
Compiler Error C3652
Compiler Error C3653
Compiler Error C3654
Compiler Error C3655
Compiler Error C3656
Compiler Error C3657
Compiler Error C3658
Compiler Error C3661
Compiler Error C3662
Compiler Error C3665
Compiler Error C3666
Compiler Error C3668
Compiler Error C3669
Compiler Error C3670
Compiler Error C3671
Compiler Error C3672
Compiler Error C3673
Compiler Error C3675
Compiler Error C3697
Compiler Error C3698
Compiler Error C3699
Compiler Error C3701
Compiler Error C3702
Compiler Error C3703
Compiler Error C3704
Compiler Error C3705
Compiler Error C3706
Compiler Error C3707
Compiler Error C3708
Compiler Error C3709
Compiler Error C3710
Compiler Error C3711
Compiler Error C3712
Compiler Error C3713

Compiler Error C3714
Compiler Error C3715
Compiler Error C3717
Compiler Error C3718
Compiler Error C3719
Compiler Error C3721
Compiler Error C3722
Compiler Error C3723
Compiler Error C3724
Compiler Error C3725
Compiler Error C3727
Compiler Error C3728
Compiler Error C3730
Compiler Error C3731
Compiler Error C3732
Compiler Error C3733
Compiler Error C3734
Compiler Error C3736
Compiler Error C3737
Compiler Error C3738
Compiler Error C3739
Compiler Error C3740
Compiler Error C3741
Compiler Error C3743
Compiler Error C3744
Compiler Error C3745
Compiler Error C3747
Compiler Error C3748
Compiler Error C3749
Compiler Error C3752
Compiler Error C3753
Compiler Error C3754
Compiler Error C3755
Compiler Error C3756
Compiler Error C3761
Compiler Error C3762
Compiler Error C3763
Compiler Error C3764
Compiler Error C3765
Compiler Error C3766

Compiler Error C3767
Compiler Error C3768
Compiler Error C3769
Compiler Error C3797
Compiler Error C3798
Compiler Error C3799
Compiler Error C3800
Compiler Error C3802
Compiler Error C3803
Compiler Error C3804
Compiler Error C3805
Compiler Error C3807
Compiler Error C3808
Compiler Error C3809
Compiler Error C3811
Compiler Error C3812
Compiler Error C3813
Compiler Error C3814
Compiler Error C3815
Compiler Error C3816
Compiler Error C3817
Compiler Error C3818
Compiler Error C3819
Compiler Error C3821
Compiler Error C3822
Compiler Error C3823
Compiler Error C3824
Compiler Error C3825
Compiler Error C3826
Compiler Error C3828
Compiler Error C3830
Compiler Error C3831
Compiler Error C3832
Compiler Error C3833
Compiler Error C3834
Compiler Error C3836
Compiler Error C3838
Compiler Error C3839
Compiler Error C3840
Compiler Error C3841

Compiler Error C3842
Compiler Error C3845
Compiler Error C3846
Compiler Error C3848
Compiler Error C3849
Compiler Error C3850
Compiler Error C3851
Compiler Error C3852
Compiler Error C3853
Compiler Error C3854
Compiler Error C3855
Compiler Error C3856
Compiler Error C3857
Compiler Error C3858
Compiler Error C3859
Compiler Error C3860
Compiler Error C3861
Compiler Error C3862
Compiler Error C3865
Compiler Error C3866
Compiler Error C3867
Compiler Error C3868
Compiler Error C3869
Compiler Error C3872
Compiler Error C3874
Compiler Error C3880
Compiler Error C3883
Compiler Error C3886
Compiler Error C3887
Compiler Error C3890
Compiler Error C3891
Compiler Error C3892
Compiler Error C3893
Compiler Error C3894
Compiler Error C3895
Compiler Error C3896
Compiler Error C3897
Compiler Error C3898
Compiler Error C3899
Compiler Error C3900

Compiler Error C3901
Compiler Error C3902
Compiler Error C3903
Compiler Error C3904
Compiler Error C3908
Compiler Error C3909
Compiler Error C3910
Compiler Error C3911
Compiler Error C3912
Compiler Error C3913
Compiler Error C3914
Compiler Error C3915
Compiler Error C3917
Compiler Error C3918
Compiler Error C3919
Compiler Error C3920
Compiler Error C3921
Compiler Error C3923

Compiler Error C3500

Error Message
invalid ProgID 'progid'

An invalid progid was specified with the **#import** statement. Check the Windows registry to ensure that you are specifying a valid progid.

Compiler Error C3501

Error Message
there is no typelib registered for ProgID 'progid'

The class ID for a given progid does not have an associated type library. Therefore, you cannot pass this progid to the **#import** statement.

Compiler Error C3505

Error Message
cannot load type library 'guid'

C3505 can be caused if you are running the (32-bit to) 64-bit cross-compiler on a 64-bit machine because the compiler is running under WOW64 and can only read from the 32-bit registry hive.

You can resolve this C3505 by building 32-bit and 64-bit versions of the type library you are trying to import and register them both. Or you can use the native 64-bit compiler, but that would require changing your VC++ Directories in the IDE to point to the 64-bit compiler.

For more information, see,

- How to: Enable a 64-Bit Visual C++ Toolset at the Command Line
- How to: Enable a 64-Bit Visual C++ Toolset at the Command Line
- How to: Configure Visual C++ Projects to Target 64-Bit Platforms

Compiler Error C3506

Error Message
there is no typelib registered for LIBID 'id'

A typelib was not registered correctly. Use regsvr32.exe to register a type library.

Compiler Error C3507

Error Message
a ProgID can have no more than 39 characters 'id'; nor contain any punctuation apart from '.'; nor start with a digit

The progid attribute has restrictions on the values that it can take.

The following sample generates C3507:

```
// C3507.cpp
[module(name="x")];
[
coclass,
progid("0123456789012345678901234567890123456789"),
uuid("00000000-0000-0000-0000-000000000001") // C3507 expected
]
struct CMyStruct {
};
int main() {
}
```

Compiler Error C3508

Error Message
'type': is not a valid Automation type

An invalid type was specified.

Example
The following sample generates C3508:

```
// C3508.cpp
#define _ATL_DEBUG_QI

#define WIN32_LEAN_AND_MEAN
#define STRICT
#ifndef _WIN32_WINNT
#define _WIN32_WINNT 0x0400
#endif

#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
extern CComModule _Module;
#include <atlcom.h>
#include <atlctl.h>
#include <atlstr.h>

extern "C" int printf_s(const char*, ...);

[module(name=oso)];

union U
// try the following two lines instead
// [export]
// struct U
{
   int i, j;
};

[dispinterface]
__interface I
{
   [id(1)] HRESULT func(U* u);
};

[coclass]
struct C : I
{
   HRESULT func(U*)   // C3508
   {
      return E_FAIL;
   }
};

int main()
{
}
```

Compiler Error C3509

Error Message
'type': invalid Automation return type; when a parameter is marked 'retval', the return type must be 'void', 'HRESULT' or 'SCODE'

A method in a COM interface must return either void or an HRESULT.

The following sample generates C3509:

```
// C3509.cpp
#define _ATL_DEBUG_QI

#define WIN32_LEAN_AND_MEAN   // Exclude rarely-used stuff from Windows headers
#define STRICT
#ifndef _WIN32_WINNT
#define _WIN32_WINNT 0x0400
#endif

#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
extern CComModule _Module;
#include <atlcom.h>
#include <atlctl.h>
#include <atlstr.h>

[module(name=oso)];

[dispinterface, uuid(00000000-0000-0000-0000-000000000001)]
__interface I {
   [id(1)] int f([out, retval] int*);   // C3509
   // try the following line instead
   // [id(1)] void f([out, retval] int*);
};

[coclass, uuid(00000000-0000-0000-0000-000000000002)]
struct C : I {
   int f(int*) {
   // try the following line instead, and delete return statement
   // void f(int*) {
      return 0;
   }
};

int main() {
}
```

Compiler Error C3510

Error Message
cannot locate dependent type library 'type_lib'

no_registry and auto_search were passed to **#import** but the compiler was not able to find a referenced type library.

To resolve this error, make sure that all type libraries and referenced type libraries are available to the compiler.

The following sample generates C3510:

Assume that the following two type libraries were built, and that C3510a.tlb was deleted or not on the path.

```
// C3510a.idl
[uuid("f87070ba-c6d9-405c-a8e4-8cd9ca25c12b")]
library C3510aLib
{
   [uuid("f87070ba-c6d9-405c-a8e4-8cd9ca25c12c")]
   enum E_C3510
   {
      one, two, three
   };
};
```

And then the source code for the second type library:

```
// C3510b.idl
// post-build command: del /f C3510a.tlb
[uuid("f87070ba-c6d9-405c-a8e4-8cd9ca25c12e")]
library C3510bLib
{
   importlib ("C3510a.tlb");
   [uuid("f87070ba-c6d9-405c-a8e4-8cd9ca25c12d")]
   struct S_C3510 {
      enum E_C3510 e;
   };
};
```

And then the client code:

```
// C3510.cpp
#import "c3510b.tlb" no_registry auto_search   // C3510
int main() {
   C3510aLib::S_C4336 ccc;
}
```

Compiler Error C3519

Error Message
'invalid_param' : invalid parameter to embedded_idl attribute

A parameter was passed to the **embedded_idl** attribute of #import, but the compiler did not recognize the parameter.

The only parameters that are allowed for **embedded_idl** are **emitidl** and **no_emitidl**.

The following sample generates C3519:

```
// C3519.cpp
// compile with: /LD
[module(name="MyLib2")];
#import "C:\testdir\bin\importlib.tlb" embedded_idl("no_emitidcl")
// C3519
#import "C:\testdir\bin\importlib.tlb" embedded_idl("no_emitidl")
// OK
```

Compiler Error C3603

Error Message

'Symbol' : type 'Type' not yet supported

You attempted to use a data type that is not yet supported by the .NET runtime in a managed object.

Compiler Error C3607

Error Message
'calling convention' : calling convention unavailable

When using **DllImport** on a function, the function cannot be declared with the __fastcall or thiscall calling convention.

Compiler Error C3609

Error Message
'member': a sealed function must be virtual

sealed is only allowed on a class, struct, or member function marked **virtual**.

The following sample generates C3609:

```
// C3609.cpp
// compile with: /clr /c
ref class C {
   int f() sealed;   // C3609
   virtual int f2() sealed;   // OK
};
```

Managed Extensions for C++

The following sample generates C3609:

```
// C3609c.cpp
// compile with: /clr:oldSyntax /c
__gc class C {
   __sealed int f();   // C3609
   __sealed virtual int f2();   // OK
};
```

Compiler Error C3610

Error Message
'valuetype': value type must be 'boxed' before method 'method' can be called

By default, a value type is not on the managed heap. Before you can call methods from .NET runtime classes, such as **Object**, you need to move the value type to the managed heap.

C3610 is only reachable using **/clr:oldSyntax**.

The following sample generates C3610:

```
// C3610.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__value class A {};

int main() {
   A a;
   a.GetType(); // C3610

   // OK
   __box A* ovar = __box(a);
   ovar->GetType();
}
```

Compiler Error C3611

Error Message

'function': a sealed function cannot have a pure-specifier

A sealed function was declared incorrectly. For more information, see sealed.

Example

The following sample generates C3611.

```
// C3611.cpp
// compile with: /clr /c

ref struct V {
   virtual void Test() sealed = 0;   // C3611
   virtual void Test2() sealed;   // OK
   virtual void Test3() = 0;   // OK
};
```

Compiler Error C3612

Error Message
'type': a sealed class cannot be abstract

Types defined with **value** (or __value) are sealed by default and a class is abstract unless it implements all methods of its base. A sealed abstract class can neither be a base class nor can it be instantiated.

For more information, see Classes and Structs (Managed).

The following sample generates C3612:

```
// C3612.cpp
// compile with: /clr /c
value struct V: public System::ICloneable {};   // C3612

// OK
value struct V2: public System::ICloneable {
   Object^ Clone();
};
```

Compiler Error C3616

Error Message
'number' : a size cannot be specified in a __gc array declaration

A __gc array was incorrectly declared. Subscript values are allowed on the right side of the expression but not on the left.

C3616 is only reachable using **/clr:oldSyntax**.

The following sample generates C3616 and shows one way to resolve the error:

```
// C3616.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
int main()
{
   int a  __gc[2] = new int __gc [];   // C3616
   int b __gc [] = new int __gc [12];   // ok
}
```

Compiler Error C3618

Error Message

'function': a method marked DllImport cannot be defined

A method marked with DllImportAttribute is defined in the specified.DLL.

Example

The following sample generates C3618.

```
// C3618.cpp
// compile with: /clr /c
using namespace System;
using namespace System::Runtime::InteropServices;

[ DllImport("user32.dll", EntryPoint="MessageBox", CharSet=CharSet::Ansi) ]  // CHANGED
void Func();

void Func() {}   // C3618, remove this function definition to resolve
```

Compiler Error C3619

Error Message
a template cannot be declared within a managed type

Class templates are not allowed in a managed class or interface.

C3619 is only reachable using **/clr:oldSyntax**.

The following sample generates C3619:

```
// C3619.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc class X {
   template<typename T> class Y {   // C3619
   };
};
```

Compiler Error C3622

Error Message
'class' : a class declared as 'keyword' cannot be instantiated

An attempt was made to instantiate a class marked as abstract (Visual C++) (or __abstract). A class marked as abstract can be a base class, but it cannot be instantiated.

Example
The following sample generates C3622.

```
// C3622.cpp
// compile with: /clr
ref class a abstract {};

int main() {
   a aa;   // C3622
}
```

The following sample generates C3622.

```
// C3622_b.cpp
// compile with: /clr:oldSyntax
__abstract class a {
};
int main() {
   a aa;   // C3622
}
```

Compiler Error C3623

Error Message
'variable': bit fields are not supported in managed types

The use of bit fields is not permitted on variables in a managed class.

The following sample generates C3623:

```
// C3623.cpp
// compile with: /clr
using namespace System;
ref class CMyClass {
public:
   int i : 1;   // C3623
};

int main() {
   CMyClass^ pMyClass = gcnew CMyClass();
   pMyClass->i = 3;
   Console::Out->WriteLine(pMyClass->i);
}
```

The following sample generates C3623:

```
// C3623_2.cpp
// compile with: /clr:oldSyntax
using namespace System;
__gc class CMyClass {
   public:
      int i : 1;   // C3623
};

int main() {
   CMyClass *pMyClass = new CMyClass();
   pMyClass->i = 3;
   Console::Out->WriteLine(pMyClass->i);
}
```

Compiler Error C3624

Error Message
'type': use of this type requires a reference to assembly 'assembly'

An assembly (reference) needed to compile your code was not specified; pass the assembly to the #using directive.

The following sample generates C3624:

```
// C3624.cpp
// compile with: /clr /c
#using <System.Windows.Forms.dll>

// Uncomment the following 2 lines to resolve.
// #using <System.dll>
// #using <System.Drawing.dll>

using namespace System;

public ref class MyForm : public Windows::Forms::Form {};   // C3624
```

The following sample generates C3624:

```
// C3624_b.cpp
// compile with: /clr:oldSyntax /c
#using <System.Windows.Forms.dll>

// Uncomment the following 2 lines to resolve.
// #using <System.dll>
// #using <System.Drawing.dll>

using namespace System;

public __gc class MyForm : public Windows::Forms::Form {};   // C3624
```

Compiler Error C3625

Error Message
'unmanaged type': an unmanaged type cannot derive from a managed type 'unmanaged class'

An unmanaged class cannot inherit from a managed class. For more information, see Classes and Structs (Managed).

The following sample generates C3625:

```
// C3625.cpp
// compile with: /clr /c
ref class B {};
class D : public B {};   // C3625 cannot inherit from a managed class
```

The following sample generates C3625:

```
// C3625_b.cpp
// compile with: /clr:oldSyntax /c
__gc class B {};
class D : public B {};   // C3625  cannot inherit from a managed class
```

Compiler Error C3626

Error Message
'keyword': '__event' keyword can only be used on COM interfaces, member functions and data members that are pointers to delegates

A keyword was used incorrectly.

The following sample generates C3626:

```
// C3626.cpp
// compile with: /c
struct A {
   __event int i;   // C3626
// try the following line instead
// __event int i();
};
```

Compiler Error C3627

Error Message
Only a value type can be boxed

Only value classes can be boxed.

The following sample generates C3627:

```
// C3627.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__value class vA {
};

__gc class A {
};

int main()
{
   A* a;
   __box(a);   // C3627

   // box a value type
   vA va;
   __box(va);
}
```

Compiler Error C3628

Error Message
'base class' : managed classes only support public inheritance

An attempt was made to use a managed class as a private or protected base class. A managed class can only be used as a base class with public access.

The following sample generates C3628:

```
// C3628a.cpp
// compile with: /clr
ref class B {
};

ref class D : private B {   // C3628

// The following line resolves the error.
// ref class D : public B {
};

int main() {
}
```

The following sample generates C3628:

```
// C3628b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
__gc class B {
};

__gc class D : B {   // C3628, private is the default access level

// The following line resolves the error.
// __gc class D : public B {
};

int main() {
}
```

Compiler Error C3630

Error Message
error when processing the token 'token'

A token in source code could not be processed.

Compiler Error C3631

Error Message
'function': cannot overload managed events

A managed event cannot be overloaded.

Example
C3631 is only reachable using **/clr:oldSyntax**.

The following sample generates C3631.

```
// C3631.cpp
// compile with: /clr:oldSyntax /c

public __gc struct S2 {
   __event void func1();
   __event void func1(int);   // C3631 delete second declaration of func1
};

public __gc struct S1 {
   __delegate void del1();
   __delegate void del2();
   __event int add_myE(del1*) { return 0; }
   __event int remove_myE(del1*) { return 0; }
   __event int add_myE(del2*) { return 0; }   // C3631
   __event int remove_myE(del2*) { return 0; }   // C3631
};
```

Compiler Error C3632

Error Message
'event': illegal style of event for construct

__event declarations are not valid in all constructs.

C3632 is only reachable using **/clr:oldSyntax**.

The following sample generates C3632:

```
// C3632.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;

public __gc __interface I
{
   __event void sna();   // C3632
};

// use the following as an example
__delegate void MyDel();
public __gc __interface I2
{
   __event MyDel* sna;
};

int main()
{
}
```

Compiler Error C3633

Error Message
cannot define 'member' as a member of managed 'type'

CLR reference class data members cannot be of a non-POD C++ type. You can only instantiate a POD native type in a CLR type. For example, a POD type cannot contain a copy constructor or an assignment operator.

Example
The following sample generates C3633.

```
// C3633.cpp
// compile with: /clr /c
#pragma warning( disable : 4368 )

class A1 {
public:
   A1() { II = 0; }
   int II;
};

ref class B {
public:
   A1 a1;   // C3633
   A1 * a2;   // OK
   B() : a2( new A1 ) {}
    ~B() { delete a2; }
};
```

The following sample generates C3633.

```
// C3633_b.cpp
// compile with: /clr:oldSyntax /c
class A1 {
public:
   A1() { II = 0; }
   int II;
};

__gc class B {
public:
   A1 a1;   // C3633
   A1 * a2;   // OK
   B() : a2( new A1 ) {}
    ~B() { delete a2; }
};
```

Compiler Error C3634

Error Message
'function' : cannot define an abstract method of a managed class

An abstract method can be declared in a managed class, but it cannot be defined.

The following sample generates C3634:

```
// C3634.cpp
// compile with: /clr
ref class C {
   virtual void f() = 0;
};

void C::f() {   // C3634 - don't define managed class' pure virtual method
}
```

Managed Extensions for C++

The following sample generates C3634:

```
// C3634b.cpp
// compile with: /clr:oldSyntax /LD
#using <mscorlib.dll>

__gc class C {
   virtual void f() = 0;
};

void C::f() {   // C3634 - don't define managed class' pure virtual method
}
```

Compiler Error C3637

Error Message
'function' : a friend function definition cannot be a specialization of a function type

A friend function was defined incorrectly for a template or generic.

The following sample generates C3637:

```
// C3637.cpp
template <class T>
void f();

struct S {
   friend void f<int>() {}   // C3637
};
```

Possible resolution:

```
// C3637b.cpp
// compile with: /c
template <class T>
void f();

struct S {
   friend void f() {}
};
```

C3637 can also occur when using generics:

```
// C3637c.cpp
// compile with: /clr

generic <class T>
void gf();

struct S {
   friend void gf<int>() {}   // C3637
};
```

Possible resolution:

```
// C3637d.cpp
// compile with: /clr /c
generic <class T>
void gf();

struct S {
   friend void gf() {}
};
```

Compiler Error C3638

Error Message
'operator' : the standard boxing and unboxing conversion operators cannot be redefined

The compiler defines a conversion operator for each managed class to support implicit boxing. This operator cannot be redefined.

For more information, see Implicit Boxing.

The following sample generates C3638:

```
// C3638.cpp
// compile with: /clr
value struct V {
   V(){}
   static operator V^(V);   // C3638
};

int main() {
   V myV;
   V ^ pmyV = myV;   // operator supports implicit boxing
}
```

Compiler Error C3640

Error Message
'member' : a referenced or virtual member function of a local class must be defined

The compiler requires certain functions to be defined.

The following sample generates C3640:

```
// C3640.cpp
void f()
{
   struct S
   {
      virtual void f1();   // C3640
      // Try the following line instead:
      // virtual void f1(){}
   };
}
```

Compiler Error C3641

Error Message
'function' : invalid calling convention 'calling_convention' for function compiled with /clr:pure or /clr:safe

Only __clrcall calling convention is allowed with /clr:pure.

The following sample generates C3641:

```
// C3641.cpp
// compile with: /clr:pure /c
void __cdecl f() {}   // C3641
```

Compiler Error C3642

Error Message
'return_type/args' : cannot call a function with __clrcall calling convention from native code

A function that is marked with the __clrcall calling convention cannot be called from native (unmanaged) code.

return_type/args is either the name of the function or the type of the **__clrcall** function you are trying to call. A type is used when you're calling through a function-pointer.

To call a managed function from a native context, you can add a "wrapper" function that will call the **__clrcall** function. Or, you can use the CLR marshalling mechanism; see How to: Marshal Function Pointers Using PInvoke for more information.

The following sample generates C3642:

```
// C3642.cpp
// compile with: /clr
using namespace System;
struct S {
   void Test(String ^ s) {   // CLR type in signature, implicitly __clrcall
      Console::WriteLine(s);
   }
   void Test2(char * s) {
      Test(gcnew String(s));
   }
};

#pragma unmanaged
int main() {
   S s;
   s.Test("abc");   // C3642
   s.Test2("abc");   // OK
}
```

Compiler Error C3644

Error Message
'function' : cannot compile the function to generate managed code

The presence of some keywords in a function will cause the function to be compiled to native.

The following sample generates C3644:

```
// C3644.cpp
// compile with: /clr
// processor: x86

void __clrcall Func2(int i) {
   __asm {}   // C3644
}
```

Compiler Error C3645

Error Message

'function' : __clrcall cannot be used on functions compiled to native code

The presence of some keywords in a function will cause the function to be compiled to native.

Example

The following sample generates C3645.

```
// C3645.cpp
// compile with: /clr /c
#pragma unmanaged
int __clrcall dog() {}   // C3645
```

Compiler Error C3646

Error Message
'specifier' : unknown override specifier

The compiler found a token in the position where it expected to find an override specifier, but the token was not recognized by the compiler.

For more information, see Override Specifiers.

The following sample generates C3646:

```
// C3646.cpp
// compile with: /clr /c
ref class C {
   void f() unknown;   // C3646
   // try the following line instead
   // virtual void f() abstract;
};
```

Compiler Error C3648

Error Message
this explicit override syntax requires /clr:oldSyntax

When compiling for the latest managed syntax, the compiler found explicit override syntax for previous versions.

For more information, see Explicit Overrides. For more information on the older syntax, see Explicit Overrides.

The following sample generates C3648:

```
// C3648.cpp
// compile with: /clr
public interface struct I {
   void f();
};

public ref struct R : I {
   virtual void I::f() {}   // C3648
   // try the following line instead
   // virtual void f() = I::f{}
};
```

Compiler Error C3650

Error Message
'interface_method' : cannot be used as an explicit override, must be a virtual member function of a base class

An attempt was made to perform an explicit override on a member that was not virtual.

For more information, see Explicit Overrides.

The following sample generates C3650:

```
// C3650.cpp
// compile with: /clr
public interface struct I {
   void a();
};

public ref class S {
public:
   static int f() { return 0; }
   static int g() { return 0; }
};

public ref struct T1 : public S, I {
   virtual int f() new sealed = S::f;   // C3650
   virtual int g() { return 0; }   // OK does not override S::g
   virtual void a() new sealed = I::a {}   // OK
};
```

Compiler Error C3651

Error Message
'member' : cannot be used as an explicit override, must be a member of a base class

An explicit override was specified, but the function being overridden was in a type that is not a base type.

For more information, see Explicit Overrides.

The following sample generates C3651:

```
// C3651.cpp
// compile with: /clr /c
ref class C {
public:
   virtual void func2();
};

ref class Other {
public:
   virtual void func();
};

ref class D : public C {
public:
   virtual void func() new sealed = Other::func;   // C3651
   virtual void func2() new sealed = C::func2;   // OK
};
```

Compiler Error C3652

Error Message
'override' : a function that explicitly overrides must be virtual

A function that does an explicit override must be virtual. For more information, see Explicit Overrides.

The following sample generates C3652:

```
// C3652.cpp
// compile with: /clr /c
public interface class I {
   void f();
};

public ref struct R : I {
   void f() = I::f {}   // C3652
   // try the following line instead
   // virtual void f() = I::f {}
};
```

Compiler Error C3653

Error Message
'function' : cannot be used as a named override: a function being overridden not found; did you forget to name the function explicitly, using a :: operator?

An explicit override specified a function that was not found in any interface. For more information, see Explicit Overrides.

The following sample generates C3653:

```
// C3653.cpp
// compile with: /clr
public interface struct I {
   void h();
};

public ref struct X : public I {
   virtual void f() new sealed = J {};   // C3653 no J in scope
   virtual void g() {}   // OK
   virtual void h() new sealed = I::h {};   // OK
};
```

Compiler Error C3654

Error Message
'text' : syntax error in explicit override

An unexpected string was in an explicit override. For more information, see Explicit Overrides.

The following sample generates C3654:

```
// C3654.cpp
// compile with: /clr /c
public ref struct B {
   virtual void f() = 0;
   virtual void g() = 0;
   virtual void h() = 0;
};

public ref struct Q : B {
   virtual void f() = B::f, 3 {}   // C3654
   // try the following line instead
   // virtual void g() = B::g, B::h {}
};
```

Compiler Error C3655

Error Message
'function' : function already explicitly overridden

A function can only be explicitly overridden once. For more information, see Explicit Overrides.

The following sample generates C3655:

```
// C3655.cpp
// compile with: /clr /c
public ref struct B {
   virtual void f();
   virtual void g();
};

public ref struct D : B {
   virtual void f() new sealed = B::f;
   virtual void g() new sealed = B::f;   // C3655
   // try the following line instead
   // virtual void g() new sealed = B::g;
};
```

Compiler Error C3656

Error Message
'override' : override specifier cannot be repeated

An explicit override keyword can only be specified once. For more information, see Explicit Overrides.

The following sample generates C3656:

```
// C3656.cpp
// compile with: /clr /c
public interface struct O {
   int f();
};

public ref struct V : O {
   int f() override override { return 0; }   // C3656
   // try the following line instead
   // int f() override { return 0; }
};
```

Compiler Error C3657

Error Message

destructors cannot explicitly override or be explicitly overridden

Destructors or finalizers cannot be explicitly overridden. For more information, see Explicit Overrides.

Example

The following sample generates C3657.

```
// C3657.cpp
// compile with: /clr
public ref struct I {
   virtual ~I() { }
   virtual void a();
};

public ref struct D : I {
   virtual ~D() = I::~I {}   // C3657
   virtual void a() = I::a {}   // OK
};
```

Compiler Error C3658

Error Message
'keyword' : override specifier not allowed with /clr:oldSyntax

An override specifier in the current syntax was used in a compilation enabled for the previous syntax.

The following sample generates C3658:

```
// C3658.cpp
// compile with: /clr:oldSyntax /c
__gc struct C {
   virtual void f() abstract;   // C3658
};
```

The following sample demonstrates a possible resolution:

```
// C3658b.cpp
// compile with: /clr /c
ref struct C {
   virtual void f() abstract;
};
```

Compiler Error C3661

Error Message
explicit override list did not find any methods to override

An explicit override specified one or more type names. However, there was no function with the necessary signature in the type(s) that matched the overriding function's signature. If you attempt to override based on type name, there must be one or more virtual functions in the specified type(s) that match the signature of the overriding function.

For more information, see Explicit Overrides.

Compiler Error C3662

Error Message

'member' : override specifier 'specifier' only allowed on member functions of managed classes

An override specifier was used on a member of native type, which is not allowed.

For more information, see Explicit Overrides.

Example

The following sample generates C3662.

```
// C3662.cpp
// compile with: /clr /c
struct S {
   virtual void f();
};

struct S1 : S {
   virtual void f() new;   // C3662
};

ref struct T {
   virtual void f();
};

ref struct T1 : T {
   virtual void f() new;   // OK
};
```

Compiler Error C3665

Error Message
'destructor' : override specifier 'keyword' not allowed on a destructor/finalizer

A keyword was used that is not allowed on a destructor or finalizer.

For example, a new slot cannot be requested on a destructor or finalizer. For more information, see Explicit Overrides and Destructors and Finalizers in Visual C++.

The following sample generates C3665:

```
// C3665.cpp
// compile with: /clr
public ref struct R {
   virtual ~R() { }
   virtual void a() { }
};

public ref struct S : R {
   virtual ~S() new {}   // C3665
   virtual void a() new {}   // OK
};
```

Compiler Error C3666

Error Message

'constructor' : override specifier 'keyword' not allowed on a constructor

An override specifier was used on a constructor, and that is not allowed. For more information, see Override Specifiers.

Example

The following sample generates C3666.

```
// C3666.cpp
// compile with: /clr /c
ref struct R {
   R() new {}   // C3666
   R(int i) {}   // OK
};
```

Compiler Error C3668

Error Message

'method' : method with override specifier 'override' did not override any base class methods

A function attempted to override a non-existent function.

For more information, see Explicit Overrides.

Example

The following sample generates C3668.

```
// C3668.cpp
// compile with: /c
__interface I {
   void f(int);   // virtual by default
};

class J {
public:
   void g(int);
   virtual void h(int);
};

struct R : I,J {
   virtual void f() override {}   // C3668
   virtual void f(int) override {}   // OK

   virtual void g(int) override {}   // C3668
   virtual void h(int) override {}   // OK
};
```

Compiler Error C3669

Error Message

'member' : override specifier 'override' not allowed on static member functions or constructors

An override was specified incorrectly. For more information, see Explicit Overrides.

Example

The following sample generates C3669.

```
// C3669.cpp
// compile with: /clr
public ref struct R {
   R() override {}   // C3669
};
```

Compiler Error C3670

Error Message
'override' : cannot override inaccessible base class method 'method'

An override can only take place on a function whose access level makes it available in a derived type. For more information, see Explicit Overrides.

The following sample generates C3670:

```
// C3670.cpp
// compile with: /clr /c
public ref class C {
// Uncomment the following line to resolve.
// public:
   virtual void g() { }
};

public ref class D : public C {
public:
   virtual void f() new sealed = C::g {};   // C3670
};
```

Compiler Error C3671

Error Message
'function_1' : function does not override 'function_2'

When using explicit override syntax, the compiler generates an error if a function is not overridden. See Explicit Overrides for more information.

Example
The following sample generates C3671.

```
// C3671.cpp
// compile with: /clr /c
ref struct S {
   virtual void f();
};

ref struct S1 : S {
   virtual void f(int) new sealed = S::f;   // C3671
   virtual void f() new sealed = S::f;
};
```

Compiler Error C3672

Error Message

pseudo-destructor expression can only be used as part of a function call

A destructor was called incorrectly. For more information, see Destructors (C++).

Example

The following sample generates C3672.

```
// C3672.cpp
template<typename T>
void f(T* pT) {
   &pT->T::~T;   // C3672
   pT->T::~T();   // OK
};

int main() {
   int i;
   f(&i);
}
```

Compiler Error C3673

Error Message

'type' : class does not have a copy-constructor

A user-defined constructor is needed to copy objects of CLR ref types. For more information, see C++ Stack Semantics for Reference Types.

Example

The following sample generates C3673.

```
// C3673.cpp
// compile with: /clr
public ref struct R {
public:
   R() {}
   // Uncomment the following line to resolve.
   // R(R% p) {}
};

int main() {
   R r;
   R s = r;   // C3673
}
```

The following sample generates C3673.

```
// C3673_b.cpp
// compile with: /clr /c
// C3673 expected
using namespace System;
[AttributeUsage(AttributeTargets::Class)]
ref class MyAttr : public Attribute {
public:
   MyAttr() {}
   // Uncomment the following line to resolve.
   // MyAttr(int i) {}
   property int Priority;
   property int Version;
};

[MyAttr]
ref class ClassA {};   // OK, no arguments

[MyAttr(Priority = 1)]
ref class ClassB {};   // OK, named argument

[MyAttr(123)]
ref class ClassC {};   // Positional argument

[MyAttr(123, Version = 1)]
ref class ClassD {};   // Positional and named
```

Compiler Error C3675

Error Message
'function' : is reserved because 'property' is defined

When you declare a simple property, the compiler generates the get and set accessor methods, and those names are present in the scope of your program. The compiler-generated names are formed by prepending get_ and set_ to the property name. Therefore, you cannot declare functions with the same name as the compiler-generated accessors.

See property for more information.

Example
The following sample generates C3675.

```
// C3675.cpp
// compile with: /clr /c
ref struct C {
public:
   property int Size;
   int get_Size() { return 0; }   // C3675
};
```

Compiler Error C3697

Error Message
'qualifier' : cannot use this qualifier on '^'

The tracking handle (^) was applied to a qualifier for which it was not designed.

The following sample generates C3697:

```
// C3697.cpp
// compile with: /clr
using namespace System;
int main() {
   String ^__restrict s;   // C3697
   String ^ s2;   // OK
}
```

Compiler Error C3698

Error Message
'type' : cannot use this type as argument of 'operator'

A managed object was declared incorrectly.

The following sample generates C3698:

```
// C3698.cpp
// compile with: /clr

int main() {
   array<int>^a = new array<int>^(20);   // C3698
   array<int>^a2 = gcnew array<int>(20);   // OK
}
```

Compiler Error C3699

Error Message
'operator' : cannot use this indirection on type 'type'

An attempt was made to use indirection that is not allowed on *type*.

Example
The following sample generates C3699.

```
// C3699.cpp
// compile with: /clr /c
using namespace System;
int main() {
   String * s;   // C3699
   // try the following line instead
   // String ^ s2;
}
```

A trivial property cannot have reference type. See property for more information. The following sample generates C3699.

```
// C3699_b.cpp
// compile with: /clr /c
ref struct C {
   property System::String % x;   // C3699
   property System::String ^ y;   // OK
};
```

The equivalent of a "pointer to a pointer" syntax is a handle to a tracking reference. The following sample generates C3699.

```
// C3699_c.cpp
// compile with: /clr /c
using namespace System;
void Test(String ^^ i);   // C3699
void Test2(String ^% i);
```

Compiler Error C3701

Error Message
'function' : event_source has no events

You attempted to use event_source on a class that has no event methods. To fix this error, add one or more events to the class.

The following sample generates C3701:

```
// C3701.cpp
[ event_source(native) ]
class CEventSrc {
public:
   // uncomment the following line to resolve this C3701
   // __event void fireEvent(int i);
};   // C3701

int main() {
}
```

Compiler Error C3702

Error Message
'function' : ATL is required for COM events

You attempted to use COM events without including the necessary ATL header files.

The following sample generates C3702:

```
// C3702.cpp
// uncomment the following line to resolve
// #define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include <atlctl.h>

[module(dll, name=idid, uuid="12341234-1234-1234-1234-123412341234")];

[object]
__interface IEvents : IUnknown
{
   HRESULT event1([in] int i);
};

[dual]
__interface IBase
{
   HRESULT fireEvents();
};

[coclass, event_source(com)]
class CEventSrc : public IBase
{
   public:
   __event __interface IEvents;

   HRESULT fireEvents()
   {
      HRESULT hr = IEvents_event1(123);
      return hr;
   }
};   // C3702

int main() {
}
```

Compiler Error C3703

Error Message
'event handler': an event handler method must have the same storage class as the source 'event'

An event has a different storage class than the event handler to which it is hooked. For example, this error occurs if the event handler is a static member function and the event is not static. To fix this error, give the event and the event handler the same storage class.

The following sample generates C3703:

```
// C3703.cpp
// C3703 expected
#include <stdio.h>

[event_source(type=native)]
class CEventSrc {
public:
   __event static void MyEvent();
};

[event_receiver(type=native)]
class CEventHandler {
public:
   // delete the following line to resolve
   void MyHandler() {}

   // try the following line instead
   // static void MyHandler() {}

   void HookIt(CEventSrc* pSource) {
      __hook(CEventSrc::MyEvent, pSource, &CEventHandler::MyHandler);
   }

   void UnhookIt(CEventSrc* pSource) {
      __unhook(CEventSrc::MyEvent, pSource, &CEventHandler::MyHandler);
   }
};

int main() {
   CEventSrc src;
   CEventHandler hnd;

   hnd.HookIt(&src);
   __raise src.MyEvent();
   hnd.UnhookIt(&src);
}
```

Compiler Error C3704

Error Message
'function' : a vararg method cannot fire events

You attempted to use __event on a vararg method. To fix this error, replace the `fireEvent(int i, ...)` call with the `fireEvent(int i)` call as shown in the following code sample.

The following sample generates C3704:

```
// C3704.cpp
[ event_source(native) ]
class CEventSrc {
   public:
      __event void fireEvent(int i, ...);   // C3704
      // try the following line instead:
      // __event void fireEvent(int i);
};

int main() {
}
```

Compiler Error C3705

Error Message
'function' : cannot find eventing interface

You must define an event interface to use COM events. Note that the **#include** lines of the ATL header files shown in the sample below are required for using COM events. To fix this error, uncomment the definition of the `IEvents` interface in the sample code.

The following sample generates C3705:

```
// C3705.cpp
// compile with: /c
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include <atlctl.h>

[module(dll, name="idid", uuid="12341234-1234-1234-1234-123412341234")];

// Uncomment the following 4 lines to resolve.
// [object, uuid("00000000-0000-0000-0000-000000000003")]
// __interface IEvents : IUnknown {
//    HRESULT event1([in] int i);
// };

[dual, uuid("00000000-0000-0000-0000-000000000001")]
__interface IBase {
   HRESULT fireEvents();
};

[coclass, event_source(com), uuid("00000000-0000-0000-0000-000000000002")]
class CEventSrc : public IBase {
public:
   __event __interface IEvents;   // C3705 uncomment IEvents to resolve
   HRESULT fireEvents() {
      HRESULT hr = IEvents_event1(123);
      return hr;
   }
};
```

Compiler Error C3706

Error Message
'function' : must be a COM interface to fire COM events

The event interface that you use to fire COM events must be a COM interface. In this situation, the interface should either be defined using a Visual C++ attribute, or imported using #import from a type library with #import's embedded_idl attribute.

Note that the **#include** lines of the ATL header files shown in the sample below are required for using COM events. To fix this error, make `IEvents` (the eventing interface) a COM interface by applying one of the following attributes to the interface definition: object, dual, or dispinterface.

If an interface is from a header file generated by MIDL, the compiler will not recognize it as a COM interface.

The following sample generates C3706:

```
// C3706.cpp
// compile with: /c
// C3706 expected
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include <atlctl.h>

[module(dll, name="idid", uuid="12341234-1234-1234-1234-123412341234")];

// Uncomment the following line to resolve.
// [object, uuid="12341234-1234-1234-1234-123412341237"]
__interface IEvents : IUnknown {
   HRESULT event1(/*[in]*/ int i);   // uncomment [in]
};

[dual, uuid="12341234-1234-1234-1234-123412341235"]
__interface IBase {
   HRESULT fireEvents();
};

[coclass, event_source(com), uuid="12341234-1234-1234-1234-123412341236"]
class CEventSrc : public IBase {
   public:
   __event __interface IEvents;

   HRESULT fireEvents() {
      HRESULT hr = IEvents_event1(123);
      return hr;
   }
};
```

Compiler Error C3707

Error Message
'function' : dispinterface method must have a dispid

If you use a **dispinterface** method, you must assign it a **dispid**. To fix this error, assign a **dispid** to the **dispinterface** method, for example, by uncommenting the **id** attribute on the method in the sample below. For more information, see the attributes dispinterface and id.

The following sample generates C3707:

```
// C3707.cpp
#include <atlbase.h>
#include <atlcom.h>
#include <atlctl.h>

[module(name="xx")];
[dispinterface]
__interface IEvents : IDispatch
{
   HRESULT event1([in] int i);   // C3707
   // try the following line instead
   // [id(1)] HRESULT event1([in] int i);
};

int main() {
}
```

Compiler Error C3708

Error Message
'interface': improper use of 'keyword'; must be a member of a compatible event source

To declare an interface as an event, the event declaration must be in an event source.

The following sample generates C3708:

```
// C3708.cpp
// compile with: /c
#define _ATL_ATTRIBUTES 1
#include "atlbase.h"
#include "atlcom.h"

[ module(name="MyLibrary")];

[ object, uuid("00000000-0000-0000-0000-000000000001") ]
__interface I {
   HRESULT func();
};

[ object, uuid("00000000-0000-0000-0000-000000000002") ]
__interface II {
   HRESULT func();
};

__event __interface I;   // C3708

// put the event in an event source
[ coclass, event_source(com), uuid("00000000-0000-0000-0000-000000000003") ]
struct E : II {
   __event __interface II;
};
```

Compiler Error C3709

Error Message
'function': improper syntax for specifying event in __hook/__unhook

When you specify an event source with __hook or __unhook, the first parameter must be a valid event method and the second parameter must be a valid event source object (not a method).

The following sample generates C3709:

```
// C3709.cpp
// compile with: /LD
[event_source(native)]
class CEventSrc
{
public:
   __event void event1();
};

[event_receiver(native)]
class CEventRec
{
public:
   void handler1()
   {
   }

   void HookEvents(CEventSrc* pSrc)
   {
      __hook(bad, pSrc, CEventRec::handler1);   // C3709
      // Try the following line instead:
      // __hook(&CEventSrc::event1, pSrc, CEventRec::handler1);
   }

   void UnhookEvents(CEventSrc* pSrc)
   {
      __unhook(&CEventSrc::event1, pSrc, CEventRec::handler1);
   }
};
```

Compiler Error C3710

Error Message

'function': improper syntax for specifying event handler in __hook/__unhook

When you specify an event handler with __hook or __unhook, the handler must be a valid method.

Example

The following sample generates C3710

```
// C3710.cpp
// compile with: /link /opt:noref
#include <atlbase.h>
#include <atlcom.h>
#include <atlctl.h>
#include <stdio.h>

[event_source(native)]
class CEventSrc
{
public:
    __event void event1();
};

[event_receiver(native)]
class CEventRec
{
public:
    void handler1()
    {
        printf_s("Executing handler1().\n");
    }

    void HookEvents(CEventSrc* pSrc)
    {
        __hook(&CEventSrc::event1, pSrc, 0);   // C3710
        // try the following line instead
        // __hook(&CEventSrc::event1, pSrc, &CEventRec::handler1);
    }

    void UnhookEvents(CEventSrc* pSrc)
    {
        __unhook(&CEventSrc::event1, pSrc, &CEventRec::handler1);
    }
};

int main()
{
    CEventSrc eventSrc;
    CEventRec eventRec;
    eventRec.HookEvents(&eventSrc);
    eventSrc.event1();
    eventRec.UnhookEvents(&eventSrc);
}
```

Compiler Error C3711

Error Message
'method': an non-managed event source method must return void or an integral type

You defined a method in the event source that did not return void or an integral type. To fix this error, make the event and event handler have a return type of **void** or an integral type such as **int** or **long**.

The following sample generates C3711:

```
// C3711.cpp
#include <atlbase.h>
#include <atlcom.h>
#include <atlctl.h>

[event_source(native)]
class CEventSrc {
public:
   __event float event1();   // C3711
   // try the following line instead
   // __event int event1();
   // also change the handler, below
};

[event_receiver(native)]
class CEventRec {
public:
   float handler1() {         // change float to int
      return 0.0;             // change 0.0 to 0
   }
   void HookEvents(CEventSrc* pSrc) {
      __hook(CEventSrc::event1, pSrc, CEventRec::handler1);
   }
   void UnhookEvents(CEventSrc* pSrc) {
      __unhook(CEventSrc::event1, pSrc, CEventRec::handler1);
   }
};

int main() {
}
```

Compiler Error C3712

Error Message
'method': an event handler method must return the same type as the source 'method'

You defined an event handler method that did not return the same type as the source event method. To fix this error, give the event handler method the same return type as that of the source event method.

The following sample generates C3712:

```
// C3712.cpp
// compile with: /c
[event_source(native)]
class CEventSrc {
public:
   __event void event1();
};

[event_receiver(native)]
class CEventRec {
public:
   int handler1() { return 0; }
   // try the following line instead
   // void handler1() {}

   void HookEvents(CEventSrc* pSrc) {
      __hook(&CEventSrc::event1, pSrc, &CEventRec::handler1);   // C3712
   }
   void UnhookEvents(CEventSrc* pSrc) {
      __unhook(&CEventSrc::event1, pSrc, &CEventRec::handler1);   // C3712
   }
};
```

Compiler Error C3713

Error Message
'method': an event handler method must have the same function parameters as the source 'method'

You defined an event handler method that did not use the same parameters as the source event method. To fix this error, give the event handler method the same parameters as those of the source event method.

The following sample generates C3713:

```
// C3713.cpp
// compile with: /c
[event_source(native)]
class CEventSrc {
public:
   __event void event1(int nValue);
   // try the following line instead
   // __event void event1();
};

[event_receiver(native)]
class CEventRec {
public:
   void handler1() {}

   void HookEvents(CEventSrc* pSrc) {
      __hook(&CEventSrc::event1, pSrc, &CEventRec::handler1);   // C3713
   }

   void UnhookEvents(CEventSrc* pSrc) {
      __unhook(&CEventSrc::event1, pSrc, &CEventRec::handler1); // C3713
   }
};
```

Compiler Error C3714

Error Message
'method': an event handler method must have the same calling convention as the source 'method'

You defined an event handler method that did not use the same calling convention as the source event method. To fix this error, give the event handler method the same calling conventions as those of the source event method. For example, in the code below, make the calling conventions of `handler1` and `event1` match (__cdecl or __stdcall or others). Removing calling convention keywords from both declarations will also solve the problem, and cause `event1` and `handler1` to default to the thiscall calling convention. See Calling Conventions for more information.

The following sample generates C3714:

```
// C3714.cpp
// compile with: /c
// processor: x86
[event_source(native)]
class CEventSrc {
public:
   __event void __cdecl event1();
   // try the following line instead
   // __event void __stdcall event1();
};

[event_receiver(native)]
class CEventRec {
public:
   void __stdcall handler1() {}

   void HookEvents(CEventSrc* pSrc) {
      __hook(&CEventSrc::event1, pSrc, &CEventRec::handler1);   // C3714
   }

   void UnhookEvents(CEventSrc* pSrc) {
      __unhook(&CEventSrc::event1, pSrc, &CEventRec::handler1); // C3714
   }
};
```

Compiler Error C3715

Error Message
'pointer': must be a pointer to 'class'

You specified a pointer in __hook or __unhook that did not point to a valid class. To fix this error, ensure that your **__hook** and **__unhook** calls specify pointers to valid classes.

Compiler Error C3717

Error Message
'method': a method that fires events cannot be defined

You declared an event method that includes an implementation. An __event method declaration cannot have a definition. To fix this error, ensure that no event method declarations have definitions. For example, in the code below, remove the function body from the `event1` declaration as indicated by the comments.

The following sample generates C3717:

```
// C3717.cpp
[event_source(native)]
class CEventSrc {
public:
   __event void event1() {   // C3717
   }

   // remove definition for event1 and substitute following declaration
   // __event void event1();
};

[event_receiver(native)]
class CEventRec {
public:
   void handler1() {
   }

   void HookEvents(CEventSrc* pSrc) {
      __hook(CEventSrc::event1, pSrc, CEventRec::handler1);
   }

   void UnhookEvents(CEventSrc* pSrc) {
      __unhook(CEventSrc::event1, pSrc, CEventRec::handler1);
   }
};

int main() {
}
```

Compiler Error C3718

Error Message
can only call 'event' in the context of a member function of the receiving class

The *event* can only be called from the receiving class.

Example
The following sample generates C3718:

```
// C3718.cpp
#define _ATL_ATTRIBUTES 1
#include "atlbase.h"
#include "atlcom.h"

[module(name="test")];

[object, uuid("00000000-0000-0000-0000-000000000001")]
__interface I
{
    HRESULT f();
};

[event_source(com), coclass, uuid("00000000-0000-0000-0000-000000000002")]
struct E
{
    __event __interface I;
};

[event_receiver(com)]
struct R
{
    void b()
    {
        printf_s("B::bar()\n");
    }

    void HookAndRun(E* pE)
    {
        __hook(&I::f, pE->GetUnknown(), &R::b);
        __raise pE->f();
    }
};

int main()
{
    CComObject<E>* pE;
    CComObject<E>::CreateInstance(&pE);

    __hook(&I::f, pE->GetUnknown(), &R::b, &r);   // C3718
    __raise pE->f();
    // try the following lines instead
    // R r;
    // r.HookAndRun(pE);
}
```

Compiler Error C3719

Error Message
'interface': an interface based event source can only be used for COM events

You declared an interface in a non-COM context.

The following sample generates C3719:

```
// C3719a.cpp
#define _ATL_ATTRIBUTES 1
#include "atlbase.h"
#include "atlcom.h"

[module(name="MyLibrary", version="1.2", helpfile="MyHelpFile")];

[object]
__interface I {
   HRESULT func1();
};

[event_source(native), coclass]
struct A {
   __event __interface I;   // C3719

   // try the following line instead
   // __event func2();
};

int main() {
}
```

To fix this error, apply the object, coclass, event_source, and event_receiver attributes appropriately to make the classes in which you are using the interface COM classes. For example:

```
// C3719b.cpp
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>

[module(name="xx")];
[object, uuid("00000000-0000-0000-0000-000000000001")]
__interface I {
   HRESULT f();
};

[coclass, event_source(com) , uuid("00000000-0000-0000-0000-000000000002")]
struct MyStruct {
   __event __interface I;
};

[event_receiver(com)]
struct MyStruct2 {
   void f() {
   }
   MyStruct2(I* pB) {
      __hook(&I::f, pB, &MyStruct2::f);
   }
};

int main()
{
}
```

Compiler Error C3721

Error Message
'signature': incompatible signature for event

An event was declared incorrectly. For more information, see __event.

C3721 is only reachable using **/clr:oldSyntax**.

Example
The following sample generates C3721.

```
// C3721.cpp
// compile with: /clr:oldSyntax /c
using namespace System;

public __delegate void MyDel();

public __gc class X {
   __event void add_E1();      // C3721
   // try the following line instead
   // __event void add_E1(MyDel * p);

   __event void remove_E1();   // C3721
   // __event void remove_E1(MyDel * p);
   // try the following line instead

   __event void raise_E1();
};
```

Compiler Error C3722

Error Message
a generic event is not allowed

The compiler only allows generic classes, structs, and functions. For more information, see Generics (Visual C++).

The following sample generates C3722:

```
// C3722.cpp
// compile with: /clr
generic <typename T>
public delegate void MyEventHandler(System::Object^ sender, System::EventArgs^ e, T optiona
l);

generic <class T>
public ref struct MyButton {
   generic<typename U>
   event MyEventHandler<U>^ Click;   // C3722
};
```

Compiler Error C3723

Error Message
'function': could not resolve event

function could not resolve which event to call.

The following sample generates C3723:

```
// C3723.cpp
struct A {
   // To resolve, comment void f(int); and uncomment the __event function
   void f(int);
   // __event void f(int);
   void f(float);

};

struct B {
   void g(int);
   B(A* a) {
   __hook(&A::f, a, &B::g);   // C3723
   }
};

int main() {
}
```

`__hook` and `__unhook` are not compatible with /clr programming. Use the += and -= operators instead.

The following sample generates C3723:

```
// C3723b.cpp
// compile with: /clr
using namespace System;

public delegate void delegate1();

public ref class CPSource {
public:
   event delegate1^ event1;
};

public ref class CReceiver {
public:
   void Handler1() {
   }

   void UnhookAll(CPSource^ pSrc) {
      __unhook(&CPSource::event1, pSrc, &CReceiver::Handler1); // C3723
      // Try the following line instead.
      // pSrc->event1 -= gcnew delegate1(this, &CReceiver::Handler1);
   }
};

int main() {
}
```

Compiler Error C3724

Error Message
must #include <windows.h> to use multi-threading with events

The windows.h file is required if you use multi-threading with events. To fix this error, add `#include <windows.h>` to the top of the file in which event sources and event receivers are defined.

```
// C3724.cpp
// uncomment the following line to resolve
// #include <windows.h>

[event_source(native), threading(apartment)]
class CEventSrc {
public:
   __event void event1();   // C3724
};

[event_receiver(native)]
class CEventRec {
public:
   void handler1() {
   }

   void HookEvents(CEventSrc* pSrc) {
      __hook(CEventSrc::event1, pSrc, CEventRec::handler1);
   }

   void UnhookEvents(CEventSrc* pSrc) {
      __unhook(CEventSrc::event1, pSrc, CEventRec::handler1);
   }
};

int main() {
}
```

Compiler Error C3725

Error Message

must #include <atlevents.h> to use events

The compiler needs helper classes that are defined in atlevents.h to use events.

Compiler Error C3727

Error Message
'event': a managed event must be a member function or a data member that is a pointer to a delegate

.NET events must be a pointer to a delegate type.

The following sample generates C3727:

```
// C3727.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc class PseudoDelegate
{
};

// use the following declaration to resolve the error.
// __delegate void PseudoDelegate(int);

__gc class MyAttr
{
   __event PseudoDelegate* MyE;
};   // C3727

int main()
{
}
```

Compiler Error C3728

Error Message
'event': event does not have a raise method

Metadata created with a language, such as C#, that does not allow an event to be raised from outside the class in which it was defined, was included with the #using directive, and a Visual C++ program using CLR programming attempted to raise the event.

To raise an event in a program developed in a language such as C#, the class containing the event needs to also define a public method that raises the event.

Compiler Error C3730

Error Message
'event': an event must have both an add and a remove method

When creating user-defined add and remove methods for an event, both add and remove need to be present.

C3730 is only reachable using **/clr:oldSyntax**.

The following sample generates C3730:

```
// C3730.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;

public __delegate void MyDel();

__gc class EventSource
{
public:
   MyDel *pE;

   __event void add_E(MyDel* p)   // C3730, uncomment remove_E to resolve
   {
      pE = static_cast<MyDel*> (Delegate::Combine(pE, p));
   }
/*
   __event void remove_E(MyDel* p)
   {
      pE = static_cast<MyDel*> (Delegate::Remove(pE, p));
   }
*/
   __event void raise_E()
   {
      if (pE != 0)  pE->Invoke();
   }
};

int main()
{
}
```

Compiler Error C3731

Error Message
incompatible event 'function1' and handler 'function2'; event source and event handler must be the same type

The event source and event receiver must have the same type (for example **native** vs. **com** types). To fix this error, make the types of the event source and the event handler match.

The following sample generates C3731:

```
// C3731.cpp
// compile with: /clr
#using <mscorlib.dll>
[event_source(native)]
struct A {
   __event void MyEvent();
};

[event_receiver(managed)]
// try the following line instead
// [event_receiver(native)]
struct B {
   void func();
   B(A a) {
      __hook(&A::MyEvent, &a, &B::func);   // C3731
   }
};

int main() {
}
```

Compiler Error C3732

Error Message
'interface': a custom interface that fires COM events cannot inherit from IDispatch

An interface that supports COM events cannot inherit from **IDispatch**. For more information, see Event Handling in COM.

The following error generates C3732:

```
// C3732.cpp
#define _ATL_ATTRIBUTES 1
#include "atlbase.h"
#include "atlcom.h"

[module(name="test")];

// to resolve this C3732, use dual instead of object
// or inherit from IUnknown
[ object ]
__interface I : IDispatch
{
};

[ event_source(com), coclass ]
struct A
{
   __event __interface I;   // C3732
};

int main()
{
}
```

Compiler Error C3733

Error Message
'event': improper syntax for specifying a COM event; did you forget '__interface'?

The wrong syntax was used for a COM event. To fix this error, change the event type or correct the syntax to comply with the COM event rules.

The following sample generates C3733:

```
#define _ATL_ATTRIBUTES 1
#include "atlbase.h"
#include "atlcom.h"

[coclass, event_source(com), // change 'com' to 'native' to resolve
uuid("00000000-0000-0000-0000-000000000001")]
class A
{
   __event void func();   // C3733
};

int main()
{
}
```

Compiler Error C3734

Error Message
'class': a managed class cannot be a coclass

The coclass attribute cannot be used with managed classes.

The following sample generates C3734:

```
// C3734.cpp
// compile with: /clr /c
[module(name="x")];

[coclass]
ref class CMyClass {   // C3734 remove the ref keyword to resolve
};
```

The following sample generates C3734:

```
// C3734_b.cpp
// compile with: /clr:oldSyntax /c
[module(name="x")];

[coclass]
__gc class CMyClass {   // C3734 remove the __gc keyword to resolve
};
```

Compiler Error C3736

Error Message
'event': must be a method or, in the case of managed events, optionally a data member

Native and COM events must be methods. .NET events can also be data members.

The following sample generates C3736:

```
// C3736.cpp
struct A {
   __event int e();
};

struct B {
   int f;   // C3736
   // The following line resolves the error.
   // int f();
   B(A* a) {
      __hook(&A::e, a, &B::f);
   }
};

int main() {
}
```

Compiler Error C3737

Error Message
'delegate': a delegate may not have an explicit calling convention

You cannot specify the calling convention for a __delegate.

The following sample generates C3737:

```
// C3737a.cpp
// compile with: /clr
delegate void __stdcall MyFunc();   // C3737
// Try the following line instead.
// delegate void MyFunc();

int main() {
}
```

The following sample generates C3737:

```
// C3737b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
__delegate void __stdcall MyFunc();   // C3737
// Try the following line instead.
// __delegate void MyFunc();

int main() {
}
```

Compiler Error C3738

Error Message
'calling_convention': the calling convention of the explicit instantiation must match that of the template being instantiated

It is recommended that you do not specify a calling convention on an explicit instantiation. If you must, though, the calling conventions must match.

Example
The following sample generates C3738.

```
// C3738.cpp
// compile with: /clr
// processor: x86
#include <stdio.h>
template< class T >
void f ( T arg ) {   // by default calling convention is __cdecl
   printf ( "f: %s\n", __FUNCSIG__ );
}

template
void __stdcall f< int > ( int arg );   // C3738
// try the following line instead
// void f< int > ( int arg );

int main () {
   f(1);
   f< int > ( 1 );
}
```

Compiler Error C3739

Error Message
'class': syntax is only supported when the 'layout_dependent' parameter of event_receiver is true

You tried to hook an entire interface of events but **layout_dependent** on event_receiver attribute is not true; you must hook a single event at a time.

The following sample generates C3739:

```
// C3739.cpp
struct A
{
   __event void e();
};

// event_receiver is implied
// [ event_receiver(layout_dependent=false)]

// use the following line instead
// [event_receiver(com, layout_dependent=true), coclass ]
struct B
{
   void f();
   B(A* a)
   {
      __hook(A, a, &B::f);   // C3739
      // use the following line instead to hook a single event
      // __hook(&A::e, a, &B::f);
   }
};

int main()
{
}
```

Compiler Error C3740

Error Message
templates cannot source or receive events

A templated class or struct cannot contain events.

The following sample generates C3740:

```
// C3740.cpp
template <typename T>   // Delete the template specification
struct E {
   __event void f();   // C3740
};

int main() {
}
```

Compiler Error C3741

Error Message
'class': must be a coclass when the 'layout_dependent' parameter of event_receiver = true

When **layout_dependent=true** for an event_receiver class, then the class must also have the coclass attribute.

The following sample generates C3741

```
// C3741.cpp
// compile with: /c
// C3741 expected
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
[module(name="xx")];

[object, uuid("00000000-0000-0000-0000-000000000001")]
__interface I{ HRESULT f(); };

// Delete the following line to resolve.
[ event_receiver(com, layout_dependent=true)]

// class or struct must be declared with coclass
// Uncomment the following line to resolve.
// [ event_receiver(com, layout_dependent=true), coclass, uuid("00000000-0000-0000-0000-000
000000002")]
struct R : I {
   HRESULT f(){ return 0; }
   R(){}
   R(I* a){ __hook(I, a); }
};
```

Compiler Error C3743

Error Message
can only hook/unhook an entire interface when the 'layout_dependent' parameter of event_receiver is true

The __unhook function varies in the number of parameters that it takes based on the value passed to the *layout_dependent* parameter in the event_receiver class.

The following sample generates C3743:

```
// C3743.cpp
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
[module(name="xx")];
[object] __interface I { HRESULT f(); };

[event_receiver(com, layout_dependent=true), coclass]
struct R : I {
        HRESULT f() {
      return 0;
   }
        R() {
   }

   R(I* a) {
      __hook(I, a, &R::f);   // C3743
      // The following line resolves the error.
      // __hook(I, a);
   }
};
```

Compiler Error C3744

Error Message

__unhook must have at least 3 arguments for managed events

The __unhook function must take three parameters when used in a program that is compiled for Managed Extensions for C++.

`__hook` and `__unhook` are not compatible with /clr programming. Use the += and -= operators instead.

C3744 is only reachable using **/clr:oldSyntax**.

The following sample generates C3744:

```
// C3744.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;

public __delegate void delegate1();

[ event_source(managed) ]
public __gc class CPSource {
public:
   __event delegate1* event1;
};

[event_receiver(managed)]
public __gc class CReceiver {
public:
   void Handler1() {
   }

   void UnhookAll1(CPSource* pSrc, CReceiver* pRec) {
      pRec;
      __unhook(pSrc);   // C3744
      // The following line resolves the error.
      // __unhook(&CPSource::event1, pSrc, &CReceiver::Handler1);
   }
};

int main() {
}
```

Compiler Error C3745

Error Message
'function': only an event can be 'raised'

Only a function defined with the __event keyword can be passed to the __raise keyword.

The following sample generates C3745:

```
// C3745.cpp
struct E {
   __event void func();
   void func(int) {
   }

   void func2() {
   }

   void bar() {
      __raise func();
      __raise func(1);   // C3745
      __raise func2();   // C3745
   }
};

int main() {
   E e;
   __raise e.func();
   __raise e.func(1);   // C3745
   __raise e.func2();   // C3745
}
```

Compiler Error C3747

Error Message
missing default type parameter : parameter param

Generic or template parameters with default values cannot be followed in the parameter list by parameters that do not have default values.

The following sample generates C3747:

```
// C3747.cpp
template <class T1 = int, class T2>   // C3747
struct MyStruct {};
```

Possible resolution:

```
// C3747b.cpp
// compile with: /c
template <class T1, class T2 = int>
struct MyStruct {};
```

Compiler Error C3748

Error Message
'interface': managed interfaces may not fire events

The __event keyword cannot appear inside an interface.

The following sample generates C3748:

```
// C3748.cpp
__interface I {
// try the following line instead
// struct I {
   __event void f();   // C3748
};

int main() {
}
```

Compiler Error C3749

Error Message
'attribute': a custom attribute may not be used inside a function

A custom attribute cannot be used inside a function. (For more information on custom attributes, see the topic attribute.)

The following sample generates C3749:

```
// C3749a.cpp
// compile with: /clr /c
using namespace System;

[AttributeUsage(AttributeTargets::All)]
public ref struct ABC : public Attribute {
   ABC() {}
};

void f1() { [ABC]; };  // C3749
```

The following sample generates C3749:

```
// C3749b.cpp
// compile with: /clr:oldSyntax /c
using namespace System;

[attribute(All)]
public __gc struct ABC {
   ABC() {}
};

void f1() { [ABC]; };  // C3749
```

Compiler Error C3752

Error Message

'attribute class': cannot classify attribute; 'keyword' should not be used in this context

A user-defined attribute was applied incorrectly.

Compiler Error C3753

Error Message
a generic property is not allowed

Generic parameter lists can only appear on managed classes, structs, or functions.

For more information, see Generics (Visual C++) and property.

Example
The following sample generates C3753.

```
// C3753.cpp
// compile with: /clr /c
ref struct A {
   generic <typename T>
   property int i;   // C3753 error
};
```

Compiler Error C3754

Error Message
delegate constructor: member function 'function' cannot be called on an instance of type 'type'

A call was made to a function through a pointer to a type that does not contain the function.

The following sample generates C3754:

```
// C3754a.cpp
// compile with: /clr
using namespace System;

delegate void MyDel();

interface class MyInterface {};

ref struct MyClass : MyInterface {
   void f() {}
};

int main() {
   MyInterface^ p = gcnew MyClass;
   MyDel^ q = gcnew MyDel(p, &MyClass::f);   // C3754
   // try the following line instead
//   MyDel^ q = gcnew MyDel(safe_cast<MyClass^>(p), &MyClass::f);
}
```

The following sample generates C3754:

```
// C3754b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;

__delegate void MyDel();

__gc __interface MyInterface {};

__gc struct MyClass : MyInterface {
   void f() {}
};

int main() {
   MyInterface* p = new MyClass;
   MyDel* q = new MyDel(p, &MyClass::f);   // C3754
   // try the following line instead
   // MyDel* q = new MyDel(__try_cast<MyClass*>(p), &MyClass::f);
}
```

Compiler Error C3755

Error Message
'delegate': a delegate may not be defined

A delegate can be declared but not defined.

Example
The following sample generates C3755.

```
// C3755.cpp
// compile with: /clr /c
delegate void MyDel() {};   // C3755
```

C3755 can also occur if you attempt to create a delegate template. The following sample generates C3755.

```
// C3755_b.cpp
// compile with: /clr /c
ref struct R {
   template<class T>
   delegate void D(int) {}   // C3755
};
```

Compiler Error C3756

Error Message
'name': delegate definition conflicts with an existing symbol

A delegate or __delegate name is the same as a class that was previously declared. Either rename the class or delete it.

The following sample generates C3756:

```
// C3756a.cpp
// compile with: /clr

ref class test;
delegate void test();   // C3756, rename class to resolve

int main() {
}
```

The following sample generates C3756:

```
// C3756b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc class test;
__delegate void test();   // C3756, rename class to resolve

int main() {
}
```

Compiler Error C3761

Error Message
'function': 'retval' can only appear on the last argument of a function

The retval attribute was used on a function argument that was not the last argument in the list.

The following sample generates C3761:

```
// C3761.cpp
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>

[ module(name=test) ];

[dispinterface]
__interface I
{
   [id(1)] HRESULT func([out, retval] int* i, [in] int j);
   // try the following line instead
   // [id(1)] HRESULT func([in] int i, [out, retval] int* j);
};

[coclass]
struct C : I {   // C3761
   HRESULT func(int* i, int j)
   // try the following line instead
   // HRESULT func(int j, int* i)
   {
      return S_OK;
   }
};

int main()
{
}
```

Compiler Error C3762

Error Message
unable to process attribute 'attribute'

A user-defined attribute that inherits from `System.Security.Permissions.SecurityAttribute` is being used to define a security attribute. Such an attribute cannot be used in the same assembly where it is defined.

Compiler Error C3763

Error Message
'type': 'retval' and 'out' can only appear on a data-pointer type

The out or retval attributes can only appear on parameters of type pointer. Either remove the attribute or make the parameter of type pointer.

The following sample generates C3763:

```
// C3763.cpp
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlplus.h>
#include <atlcom.h>

[ module(name=test) ];

[ dispinterface, uuid("00000000-0000-0000-0000-000000000001") ]
__interface IF84 : IDispatch
{
   [id(0x00000002)]HRESULT m2([out]unsigned char);
};

[ coclass, uuid("00000000-0000-0000-0000-000000000002") ]
class CF84 : public IF84
{   // C3763
public:
   HRESULT __stdcall m2(unsigned char i)
   {
      return S_OK;
   }
};


int main()
{
}
```

Compiler Error C3764

Error Message
'override_function': cannot override base class method 'base_class_function'

The compiler detected an ill-formed override. For example, the base class function was not **virtual**. For more information, see override.

Example
The following sample generates C3764.

```
// C3764.cpp
// compile with: /clr /c
public ref struct A {
   void g(int);
   virtual void h(int);
};

public ref struct B : A {
   virtual void g(int) override {}   // C3764
   virtual void h(int) override {}   // OK
};
```

C3764 can also occur when a base class method is both explicitly and named overridden. The following sample generates C3764.

```
// C3764_b.cpp
// compile with: /clr /c
ref struct A {
   virtual void Test() {}
};

ref struct B : public A {
   virtual void Test() override {}
   virtual void Test2() = A::Test {}   // C3764
};
```

Compiler Error C3765

Error Message
'event': cannot define an event in a class/struct 'type' marked as an event_receiver

If a class is marked with the event_receiver attribute, the class cannot contain an __event declaration.

The following sample generates C3765:

```
// C3765.cpp
[event_receiver(native)]
struct ER2 {
   __event void f();   // C3765
   __event void b(int);   // C3765
};
```

Compiler Error C3766

Error Message

'type' must provide an implementation for the interface method 'function'

A class that inherits from an interface must implement the interface members.

Example

The following sample generates C3766.

```
// C3766.cpp
// compile with: /clr /c

delegate void MyDel();

interface struct IFace {
   virtual event MyDel ^ E;
};

ref struct Class1 : public IFace {};   // C3766

// OK
ref struct Class2 : public IFace {
   virtual event MyDel ^ E {
      void add(MyDel ^) {}
      void remove(MyDel ^) {}
   }
};
```

Compiler Error C3767

Error Message
'function' candidate function(s) not accessible

A friend function defined in a class is not supposed to be treated as if it were defined and declared in the global namespace scope. It can, however, be found by argument-dependent lookup.

C3767 may also be caused by a breaking change: native types are now private by default in a **/clr** compilation; see Type Visibility for more information.

The following sample generates C3767:

```
// C3767a.cpp
// compile with: /clr
using namespace System;
public delegate void TestDel();

public ref class MyClass {
public:
   static event TestDel^ MyClass_Event;
};

public ref class MyClass2 : public MyClass {
public:
   void Test() {
      MyClass^ patient = gcnew MyClass;
      patient->MyClass_Event();
    }
};

int main() {
   MyClass^ x = gcnew MyClass;
   x->MyClass_Event();   // C3767

   // OK
   MyClass2^ y = gcnew MyClass2();
   y->Test();
};
```

The following sample generates C3767:

```
// C3767b.cpp
// compile with: /clr:oldSyntax
using namespace System;
__delegate void TestDel();

public __gc class MyClass {
public:
   static __event TestDel * MyClass_Event;
};

public __gc class MyClass2 : public MyClass {
public:
   void Test() {
      MyClass* patient = new MyClass;
      patient->MyClass_Event();
    }
};

int main() {
   MyClass* x = new MyClass;
   x->MyClass_Event();   // C3767

   // OK
```

```
   MyClass2 * y = new MyClass2();
   y->Test();
};
```

The following sample generates C3767:

```
// C3767c.cpp
// compile with: /clr /c

ref class Base  {
protected:
   void Method() {
      System::Console::WriteLine("protected");
   }
};

ref class Der : public Base {
   void Method() {
      ((Base^)this)->Method();   // C3767
      // try the following line instead
      // Base::Method();
   }
};
```

The following sample generates C3767:

```
// C3767d.cpp
// compile with: /clr:oldSyntax /c

__gc class Base {
protected:
   void Method() {
      System::Console::WriteLine("protected");
   }
};

__gc class Der : public Base {
   void Method() {
      ((Base*)this)->Method();   // C3767
      // try the following line instead
      // Base::Method();
   }
};
```

In Visual C++ .NET 2002, the compiler changed the way it looked up symbols. In some cases, it would have automatically looked for symbols in a specified namespace. Now, it will use argument-dependent lookup. For more information, see Summary of Compile-Time Breaking Changes.

The following sample generates C3767:

```
// C3767e.cpp
namespace N {
   class C {
      friend void FriendFunc() {}
      friend void AnotherFriendFunc(C* c) {}
   };
}

int main() {
   using namespace N;
   FriendFunc();   // C3767 error
   C* pC = new C();
   AnotherFriendFunc(pC);   // found via argument-dependent lookup
}
```

For code that is valid in Visual C++ .NET 2003 and Visual C++ .NET 2002, declare the friend in class scope and define it in namespace scope:

```
// C3767f.cpp
class MyClass {
   int m_private;
   friend void func();
};

void func() {
   MyClass s;
   s.m_private = 0;
}

int main() {
   func();
}
```

Compiler Error C3768

Error Message
cannot take the address of a virtual vararg function in pure managed code

When compiling with **/clr:pure**, you cannot take the address of a virtual **vararg** function.

The following sample generates C3768:

```
// C3768.cpp
// compile with: /clr:pure
struct A
{
   virtual void f(...);
};

int main()
{
   &(A::f);   // C3768
}
```

Compiler Error C3769

Error Message

'type' : a nested class cannot have the same name as the immediately enclosing class

A nested class cannot have the same name as the immediately enclosing class.

Example

The following sample generates C3769.

```
// C3769.cpp
// compile with: /c
class x {
   class x {};   // C3769
   class y {
      class x{};   // OK
   };
};
```

Compiler Error C3797

Error Message

'override': event declaration cannot have override specifier (should be placed on event add/remove/raise methods instead)

You cannot override a trivial event (an event without explicitly defined accessor methods) with another trivial event. The overriding event must define its behavior with accessor functions.

For more information, see event (Visual C++).

Example

The following sample generates C3797.

```
// C3797.cpp
// compile with: /clr /c
delegate void MyDel();

ref class Class1 {
public:
   virtual event MyDel ^ E;
};

ref class Class2 : public Class1 {
public:
   virtual event MyDel ^ E override;   // C3797
};

// OK
ref class Class3 : public Class1 {
public:
   virtual event MyDel ^ E {
      void add(MyDel ^ d) override {}
      void remove(MyDel ^ d) override {}
   }
};
```

Compiler Error C3798

Error Message
'specifier': property declaration cannot have override specifier (should be placed on property get/set methods instead)

A property was declared incorrectly. For more information, see

- property
- abstract (Visual C++)
- sealed

Example
The following sample generates C3798

```
// C3798.cpp
// compile with: /clr /c
ref struct A {
   property int Prop_1 abstract;   // C3798
   property int Prop_2 sealed;   // C3798

   // OK
   property int Prop_3 {
      virtual int get() abstract;
      virtual void set(int i) abstract;
   }

   property int Prop_4 {
      virtual int get() sealed;
      virtual void set(int i) sealed;
   }
};
```

Compiler Error C3799

Error Message
indexed property cannot have an empty parameter list

An indexed property was declared incorrectly. See How to: Use Indexed Properties for more information.

Example
The following sample generates C3799.

```
// C3799.cpp
// compile with: /clr /c
ref struct C {
   property int default[] {   // C3799
   // try the following line instead
   // property int default[int] {
      int get(int index) { return 0; }
      void set(int index, int value) {}
   }
};
```

Compiler Error C3800

Error Message
'declaration': cannot mix properties and events

You cannot declare a construct to be both a property and an event.

C3800 is only reachable using **/clr:oldSyntax**.

The following sample generates C3800:

```
// C3800.cpp
// compile with: /clr:oldSyntax
#using "mscorlib.dll"

__delegate void MyDel();
public __gc struct S
{
   __property __event void set_E(MyDel*)
   {
   }   // C3800
};

int main()
{
}
```

Compiler Error C3802

Error Message
'name': invalid name for a property

A __property was declared with an invalid name.

C3802 is only reachable using **/clr:oldSyntax**.

The following sample generates C3802

```
// C3802.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;

__gc class Test {
public:
   __property int get_() {       // C3802, empty name for a property
      return 0;
   }

   __property int get_Test() {   // C3802, 'Test' -- parent class name
      return 0;
   }

   __property int get_val() {  // OK
      return _mValue;
   }

private:
   int _mValue;
};

int main() {
}
```

Compiler Error C3803

Error Message
'property': property has a type that is incompatible with one of its accessors 'accessor'

The type of a property defined with property does not match the return type for one of its accessor functions.

The following sample generates C3803:

```
// C3803.cpp
struct A
{
   __declspec(property(get=GetIt)) int i;
   char GetIt()
   {
      return 0;
   }

   /*
   // try the following definition instead
   int GetIt()
   {
      return 0;
   }
   */
}; // C3803

int main()
{
}
```

Compiler Error C3804

Error Message

'property_accessor': the accessor methods for a property must either be all static or all non-static

When defining a non-trivial property, the accessor functions can be either static or instance, but not both.

See property for more information.

Example

The following sample generates C3804.

```
// C3804.cpp
// compile with: /c /clr
ref struct A {

   property int i {
      static int get() {}
      void set(int i) {}
   }   // C3804 error

   // OK
   property int j {
      int get() { return 0; }
      void set(int i) {}
   }

   property int k {
      static int get() { return 0; }
      static void set(int i) {}
   }
};
```

Compiler Error C3805

Error Message

'token' : unexpected token, expected either '}' or an identifier

When defining a property, an invalid token was encountered. Remove the invalid token.

Compiler Error C3807

Error Message

'type' : a class with the ComImport attribute cannot derive from 'type2', only interface implementation is allowed

A type that derived from ComImportAttribute can only implement an interface.

Example

The following sample generates C3807.

```
// C3807.cpp
// compile with: /clr /c
ref struct S {};
interface struct I {};

[System::Runtime::InteropServices::ComImportAttribute()]
ref struct S1 : S {};   // C3807
ref struct S2 : I {};
```

Compiler Error C3808

Error Message

'type' : a class with the ComImport attribute cannot define member 'member', only abstract or dllimport functions are allowed

A type that derived from ComImportAttribute cannot define *member*.

Example

The following sample generates C3808.

```
// C3808.cpp
// compile with: /c /clr:pure user32.lib
using namespace System::Runtime::InteropServices;

[System::Runtime::InteropServices::ComImportAttribute()]
ref struct S1 {
   int f() {}   // C3808
   virtual int g() abstract;   // OK

   [DllImport("msvcrt.dll")]
   int printf(System::String ^, int i);   // OK
};
```

Compiler Error C3809

Error Message
'structure' : a managed type cannot have any friend functions/classes/interfaces

Managed types do not allow friends. To fix this error, do not declare friends inside managed types.

The following sample generates C3809:

```
// C3809a.cpp
// compile with: /clr
ref class A {};

ref class B
{
public:
   friend ref class A;   // C3809
};

int main()
{
}
```

The following sample generates C3809:

```
// C3809b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc class A {};

__gc class B
{
public:
   friend __gc class A;   // C3809
};

int main()
{
}
```

Compiler Error C3811

Error Message
property methods 'get_accessor' and 'set_accessor' must agree on their 'static' specifier

Event accessors must agree on their use of **static**.

C3811 is only reachable using **/clr:oldSyntax**.

The following sample generates C3811:

```
// C3811.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc class X
{
   __property static int get_Size()
   {
      return 0;
   }

   __property void set_Size()
   {   // C3811, add static to set_Size or remove from get_Size
   }
};

int main()
{
}
```

Compiler Error C3812

Error Message
'property' must be the first token in a property declaration

When declaring a property, the __property keyword must be the first token on the line.

C3812 is only reachable using **/clr:oldSyntax**.

The following sample generates C3812:

```
// C3812.cpp
// compile with: /clr:oldSyntax /c
#using <mscorlib.dll>

__gc class X {
   static __property int get_Size() { // C3812, remove static specifier
      return 0;
   }
};
```

Compiler Error C3813

Error Message
a property declaration can only appear within the definition of a managed type

A property can only be declared within a user-defined type.

The following sample generates C3813:

```
// C3813a.cpp
// compile with: /clr
class A
{
   property int Int; // C3813
};

int main()
{
}
```

The following sample generates C3813:

```
// C3813b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
class A
{
   __property int get_Int()   // C3813
   {
   return 10;
   };
};

int main()
{
}
```

Compiler Error C3814

Error Message
'class member' : this member cannot co-exist with a property of the same name

A property and another class member have the same identifier. To fix this error, change one of the identifiers.

C3814 is only reachable using **/clr:oldSyntax**.

The following sample generates C3814:

```
// C3814.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;

__gc class CMyClass {
public:
   char Size;
   __property int get_Size() {return __Size;}   // C3814
   __property void set_Size(int value) {__Size=value;}   // C3814
protected:
   int __Size;
};

int main() {
}
```

Compiler Error C3815

Error Message
return type of method 'get_accessor' must match type of the last parameter of a setter

When declaring properties, the return value of the *get_accessor* method must match the last parameter in the declaration of the set accessor method.

C3815 is only reachable using **/clr:oldSyntax**.

The following sample generates C3815:

```
// C3815.cpp
// compile with: /clr:oldSyntax /LD
#using <mscorlib.dll>
__gc class X
{
public:
   __property char get_N()
   // try the following line instead
   // __property int get_N()
   {
      return m_val;
   }

   __property void set_N( int val)
   {
      m_val = val;
   }

private:
   int m_val;
};   // C3815
```

The following sample shows how you can overload properties, so that the return type of the getter does not match the last parameter of the setter.

```
// C3815b.cpp
// compile with: /clr:oldSyntax /c
#using <mscorlib.dll>
public __gc class MyClass {
public:
// 1st property:
   __property System::Int32 get_p1();
   __property void set_p1(System::Int32 i);

// 2nd property (only setter):
   __property void set_p1(System::Int32* i);

};
```

Compiler Error C3816

Error Message
'declaration' was previously declared or defined with a different managed modifier

A forward declaration and an actual declaration require that there be no conflicts or inconsistencies in the declaration of attributes.

The following sample generates C3816:

```
// C3816a.cpp
// compile with: /clr /c
class C1;
// try the following line instead
// ref class C1;

ref class C1{  // C3816, forward declaration does not use ref
};
```

The following sample generates C3816:

```
// C3816b.cpp
// compile with: /clr:oldSyntax /c
#using <mscorlib.dll>

class C1;

// try the following line to resolve the error
// __gc class C1;

__gc class C1{  // C3816, forward declaration does not use __gc
};
```

Compiler Error C3817

Error Message
'declaration' : property can only be applied to a function

The property keyword can only be a applied to a function definition.

C3817 is only reachable using **/clr:oldSyntax**.

The following sample generates C3817:

```
// C3817.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc class G {
      __property int x;   // C3817
   };

// the following class defines a property
__gc class X {
public:
   __property int get_N( int i ) {
      Console::WriteLine( L"int" );
      return m_val[i];
   }

   __property void set_N( int i, int val ) {
      m_val[i] = val;
   }

private:
   int m_val[10];
};

int main() {
}
```

Compiler Error C3818

Error Message
array property declaration 'property1' shall not overload an index property 'property2'

An overload is not possible for properties when one is an indexer and the other is an array property. See __property for more information.

C3818 is only reachable using **/clr:oldSyntax**.

The following sample generates C3818:

```
// C3818.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;

__gc class X {
public:
   __property int get_Int(int index) {
      Console::WriteLine(S"Called indexed property");
      return m_value;
   }

   __property int get_Int() __gc[] {   // C3818, rename a property
      Console::WriteLine(S"Called array property");
      return m_arr;
   }

   int m_arr __gc[];
   int m_value;
};


int main() {
   X* x = new X;
   x->m_arr = new int __gc[3];
   x->m_value = 3;

   x->Int[0];
}
```

Compiler Error C3819

Error Message
wrong number of parameters in 'property'

An accessor function for a __property had the wrong number of parameters.

C3819 is only reachable using **/clr:oldSyntax**.

The following sample generates C3819:

```
// C3819.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;

__gc struct AtClass {
   __property int get_bad() {
      return _i;
   }

   __property void set_bad() {   // C3819
   }

   // uncomment this access definition to resolve
   /*
   __property void set_bad(int i) {
      _i = i;
   }
   */
public:
   int _i;
};

int main() {
}
```

Compiler Error C3821

Error Message
'function': managed type or function cannot be used in an unmanaged function

Functions with inline assembly or setjmp cannot contain value types or managed classes. To fix this error, remove the inline assembly and **setjmp** or remove the managed objects.

C3821 can also occur if you try to use automatic storage in a vararg function. For more information, see How to: Accept Variable Arguments and C++ Stack Semantics for Reference Types.

Example

The following sample generates C3821.

```
// C3821a.cpp
// compile with: /clr /c
public ref struct R {};
void test1(...) {
   R r;   // C3821
}
```

The following sample generates C3821.

```
// C3821b.cpp
// compile with: /clr
// processor: /x86
ref class A {
   public:
   int i;
};

int main() {
   // cannot use managed classes in this function
   A ^a;

   __asm {
      nop
   }
} // C3821
```

The following sample generates C3821.

```
// C3821c.cpp
// compile with: /clr:oldSyntax
// processor: /x86
__gc  class A {
   public:
   int i;
};

int main() {
   // cannot use managed classes in this function
   A *a;

   __asm {
      nop
   }
} // C3821
```

Compiler Error C3822

Error Message
'property' : name of the property method must start with 'get_' or 'set_'

Property method names must start with either **get_** or **set_**.

C3822 is only reachable using **/clr:oldSyntax**.

The following sample generates C3822:

```
// C3822.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc struct A {
   __property int get_Size() {
      return 0;
   }

   __property void SET_Size(int i) {   // C3822
   }

   // use the property below to resolve the error
   __property void set_Size(int i) {
   }
   */
};

int main() {
}
```

Compiler Error C3823

Error Message
'modifier' : storage class modifier can only be applied to a pointer

modifier can only be applied to pointers.

C3823 is only reachable using **/clr:oldSyntax**.

The following sample generates C3823:

```
// C3823.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__pin struct A {   // C3823, cannot use __pin on a struct
};

struct B {
};

int main() {
   __pin B *a1;   // C3823
   // try ..
   // B __pin* a2;
}
```

Compiler Error C3824

Error Message
'member': this type cannot appear in this context (function parameter, return type, or a static member)

Pinning pointers cannot be function parameters, return types, or declared **static**.

The following sample generates C3824:

```
// C3824a.cpp
// compile with: /clr /c
void func() {
   static pin_ptr<int> a; // C3824
   pin_ptr<int> b; // OK
}
```

Managed Extensions for C++

Local pointers declared with the **__pin** keyword cannot be declared **static** and cannot be interior pointers.

The following sample generates C3824:

```
// C3824b.cpp
// compile with: /clr:oldSyntax /c
#using <mscorlib.dll>

__gc struct A {};

void func() {
   static A __pin* a;   // C3824
   A __pin* b;   // OK
}
```

Compiler Error C3825

Error Message
'class': a managed class can only support managed events

Only .NET events are supported in managed classes. To fix this error, change type parameter of **event_source** and **event_receiver** from **native** to **managed** or remove the attribute.

Example
The following sample generates C3825:

```
// C3825a.cpp
// compile with: /clr
public delegate void del1();

[event_source(native)]           // To fix, change 'native' to 'managed' or delete this lin
e
ref class CEventSrc
{
public:
   event del1^ event1;       // C3825

   void FireEvents() {
      event1();
   }
};

[event_receiver(native)]         // To fix, change 'native' to 'managed' or delete this lin
e
ref class CEventRec
{
public:
   void handler1()
   {
      System::Console::WriteLine("Executing handler1().\n");
   }
   void HookEvents(CEventSrc^ pSrc)
   {
      pSrc->event1 += gcnew del1(this, &CEventRec::handler1);
   }
   void UnhookEvents(CEventSrc^ pSrc)
   {
      pSrc->event1 -= gcnew del1(this, &CEventRec::handler1);
   }
};

int main()
{
   CEventSrc^ pEventSrc = gcnew CEventSrc;
   CEventRec^ pEventRec = gcnew CEventRec;
   pEventRec->HookEvents(pEventSrc);
   pEventSrc->FireEvents();
   pEventRec->UnhookEvents(pEventSrc);
}
```

For Managed Extensions for C++ You can also remove the **__gc** from the class declarations so that they are not garbage-collected classes.

The following sample generates C3825:

```
// C3825b.cpp
// compile with: /clr:oldSyntax
[event_source(native)]           // To fix, change 'native' to 'managed'
                                 // or, remove __gc from class
```

```
__gc class CEventSrc
{
public:
    __event void event1();       // C3825

    void FireEvents() {
       event1();
    }
};

[event_receiver(native)]         // To fix, change 'native' to 'managed'
                                 // or, remove __gc from class
__gc class CEventRec
{
public:
    void handler1()
    {
        System::Console::WriteLine("Executing handler1().\n");
    }
    void HookEvents(CEventSrc* pSrc)
    {
        __hook(&CEventSrc::event1, pSrc, &CEventRec::handler1);
     }
    void UnhookEvents(CEventSrc* pSrc)
    {
        __unhook(&CEventSrc::event1, pSrc, &CEventRec::handler1);
    }
};

int main()
{
    CEventSrc* pEventSrc = new CEventSrc;
    CEventRec* pEventRec = new CEventRec;
    pEventRec->HookEvents(pEventSrc);
    pEventSrc->FireEvents();
    pEventRec->UnhookEvents(pEventSrc);
}
```

Compiler Error C3826

Error Message
'type': declaring pinned formal parameters is illegal

A function parameter cannot be decorated with the __pin keyword.

C3826 is only reachable using **/clr:oldSyntax**.

The following sample generates C3826:

```
// C3826.cpp
// compile with: /clr:oldSyntax /c
__gc class G {};

typedef G __pin * GPinPtrType;
typedef G * GPinPtrType2;

void func1(G __pin * x) {}   // C3826
void func1b(G * x) {}   // OK

void func2(GPinPtrType x) {}   // C3826
void func2(GPinPtrType2 x) {}   // OK
```

Compiler Error C3828

Error Message
'object type': placement arguments not allowed while creating instances of managed classes

When creating an object from a managed type, you cannot use the placement form of operator gcnew or new.

The following sample generates C3828:

```
// C3828a.cpp
// compile with: /clr
ref struct M {
};

ref struct N {
   static array<char>^ bytes = gcnew array<char>(256);
};

int main() {
   M ^m1 = new (&N::bytes) M();   // C3828
   // The following line resolves the error.
   // M ^m1 = gcnew M();
}
```

The following sample generates C3828:

```
// C3828b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc struct M {
};

char bytes[256];

int main() {
   M *m1 = new (&bytes) M();   // C3828
   // The following line resolves the error.
   // M *m1 = new M();
}
```

When using Managed Extensions for C++, the compiler will also generate C3828 if you attempt to create an instance of a managed type in an MFC application. To track memory usage, MFC redefines the **new** operator when used in a debug version of MFC. Because of this redefinition, errors can be caused by creating instances of managed classes in an MFC application. This typically happens when porting existing MFC code to the common language runtime. For release builds, this error does not occur because MFC does not redefine the **new** operator.

In the following example, managed code placed in a .cpp file creates an instance of the **String** class. This causes C3828 when compiled in a debug version of an MFC application.

To avoid this error, temporarily undefine the **new** operator using the #undef and push_macro directives before creating instances of managed types. After the last line of managed code, restore the previous definition of the **new** operator using pop_macro.

```
#pragma push_macro("new")
#undef new

String* s;
s = new String("Hello world!");
#pragma pop_macro("new")
```

Compiler Error C3830

Error Message

'type1': cannot inherit from 'type2', value types can only inherit from interface classes

A value type cannot inherit a base class. For more information, see Classes and Structs (Managed).

The following sample generates C3830:

```
// C3830a.cpp
// compile with: /clr /c
public value struct MyStruct4 {
   int i;
};

public value class MyClass : public MyStruct4 {};   // C3830

// OK
public interface struct MyInterface4 {
   void i();
};

public value class MyClass2 : public MyInterface4 {
public:
   virtual void i(){}
};
```

Managed Extensions for C++

A **__value** type cannot inherit a base class.

The following sample generates C3830:

```
// C3830b.cpp
// compile with: /clr:oldSyntax /c
#using <mscorlib.dll>
__value struct v : public System::Object {};   // C3830
__value struct w {};   // OK
```

Compiler Error C3831

Error Message
'member': 'class' cannot have a pinned data member or a member function returning a pinning pointer

pin_ptr or __pin was used incorrectly.

The following sample generates C3831:

```
// C3831a.cpp
// compile with: /clr
ref class Y
{
public:
   int i;
};

ref class X
{
   pin_ptr<int> mbr_Y;   // C3831
   int^ mbr_Y2;   // OK
};

int main() {
   Y y;
   pin_ptr<int> p = &y.i;
}
```

The following sample generates C3831:

```
// C3831b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc class Y
{
};

__gc class X
{
   Y __pin * mbr_Y;   // C3831
   Y * mbr_Y2;   // OK

   Y __pin * mf_Y()  // C3831
   {
      Y __pin * pY = new Y();
      return pY;
   }

   Y * mf_Y2()   // OK
   {
      Y * pY = new Y();
      return pY;
   }
};
```

Compiler Error C3832

Error Message
'type library': type library looks as if it was built for 32-bit pointers; please change the 'ptrsize' qualifier

Explicit information supplied with the **ptrsize** attribute of the #import directive did not agree with what the compiler found in the type library.

Compiler Error C3833

Error Message
'type' : invalid target type for pointer_type

An interior_ptr or pin_ptr was declared incorrectly.

The following sample generates C3833:

```
// C3833.cpp
// compile with: /clr

ref class MyClass {
public:
   int data;
   MyClass() : data(35) {}
};

int main() {
   interior_ptr<MyClass> p;   // C3833

   // OK
   MyClass ^ h_MyClass = gcnew MyClass;
   interior_ptr<int> i = &(h_MyClass->data);
   System::Console::WriteLine(*i);
}
```

The following sample generates C3833:

```
// C3833b.cpp
// compile with: /clr /c
ref class G {
public:
   int i;
};

int main() {
   G ^ pG = gcnew G;
   pin_ptr<G> ppG = &pG;   // C3833 can't pin a whole object

   // OK
   pin_ptr<int> ppG2 = &pG->i;
   *ppG2 = 54;
   int * pi = ppG2;
   System::Console::WriteLine(*pi);
   System::Console::WriteLine(*ppG2);

   *pi = 55;
   System::Console::WriteLine(*pi);
   System::Console::WriteLine(*ppG2);

   *ppG2 = 56;
   System::Console::WriteLine(*pi);
   System::Console::WriteLine(*ppG2);
}
```

Compiler Error C3834

Error Message
illegal explicit cast to a pinning pointer; use a pinned local variable instead

Explicit casts to a pinned pointer are not allowed.

Example
The following sample generates C3834.

```
// C3834.cpp
// compile with: /clr
int main() {
   int x = 33;

   pin_ptr<int> p = safe_cast<pin_ptr<int> >(&x);   // C3834
   pin_ptr<int> p2 = &x;   // OK
}
```

The following sample generates C3834.

```
// C3834_b.cpp
// compile with: /clr:oldSyntax
using namespace System;

int main() {
   const __wchar_t * pS = reinterpret_cast<const __wchar_t __pin*>(S"hello");   // C3834

   // OK
   // declare a pinning variable
   String __pin* ppin = S"Hello";

   // cast pinning variable to unmanaged type
   const __wchar_t * pS2 = reinterpret_cast<__wchar_t*>(ppin);

   pS2 += 6;
}
```

Compiler Error C3836

Error Message
static constructor is not allowed to have a member initializer list

A managed class cannot have a static constructor that also has a member initialization list. Static class constructors are called by the common language runtime to do class initialization, initializing static data members.

The following sample generates C3836:

```
// C3836a.cpp
// compile with: /clr
ref class M
{
   static int s_i;

public:
   static M() :  s_i(1234)   // C3836, delete initializer to resolve
   {
   }
};

int main()
{
}
```

The following sample generates C3836:

```
// C3836b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc class M
{
   static int s_i;

public:
   static M() :  s_i(1234)   // C3836, delete initializer to resolve
   {
   }
};

int main()
{
}
```

Compiler Error C3838

Error Message
cannot explicitly inherit from 'type'

The specified *type* cannot act as a base class in any class.

The following sample generates C3838:

```
// C3838a.cpp
// compile with: /clr /c
public ref class B : public System::Enum {};   // C3838
```

The following sample generates C3838:

```
// C3838b.cpp
// compile with: /clr:oldSyntax /c
public __gc class B : public System::ValueType {};   // C3838
```

Compiler Error C3839

Error Message
cannot change alignment in a managed type

Alignment of variables is controlled by the common language runtime and cannot be modified with align.

The following sample generates C3839:

```
// C3839a.cpp
// compile with: /clr
ref class C
{
public:
   __declspec(align(32)) int m_j; // C3839
};

int main()
{
}
```

The following sample generates C3839:

```
// C3839b.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>

__gc class C
{
public:
   __declspec(align(32)) int m_j; // C3839
};

int main()
{
}
```

Compiler Error C3840

Error Message
declaration of 'method' not allowed; use destructors to implement finalization

Declare a destructor instead of a `Finalize` method in a managed class.

C3840 is only reachable using **/clr:oldSyntax**.

The following sample generates C3840:

```
// C3840.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
__gc class X
{
protected:

   void Finalize()
   {   // C3840, "declaration of 'X::Finalize' not allowed...
   }

   // declare a destructor instead
   // ~X()
   // {
   // }
};

int main()
{
}
```

Compiler Error C3841

Error Message
illegal delete expression: managed type 'type' does not have a destructor defined

When compiled with /clr:oldSyntax, deleting a pointer only implements finalization. Therefore, if the class has no destructor defined (which implies that no `Finalize` method is provided), it is an error to delete a pointer to the type.

C3841 is only reachable using **/clr:oldSyntax**.

The following sample generates C3841:

```
// C3841.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
__gc class A
{   // implicitly derived from System::Object
public:
   ~A() {};
};

int main()
{
   System::Object* p = new A;
   delete p;   // C3841 since System::Object has no destructor defined
   A* q = static_cast<A*>(p);
   delete q;   // OK since A has a destructor
}
```

Compiler Error C3842

Error Message
'function': 'const' and 'volatile' qualifiers on member functions of managed types are not supported

const and volatile are not supported on member functions of managed types.

The following sample generates C3842:

```
// C3842a.cpp
// compile with: /clr /c
public ref struct A {
   void f() const {}   // C3842
   void f() volatile {}   // C3842

   void f() {}
};
```

The following sample generates C3842:

```
// C3842b.cpp
// compile with: /clr:oldSyntax /c
public __gc struct A {
   void f() const {}   // C3842
   void f() volatile  {}   // C3842

   void f()  {}
};
```

Compiler Error C3845

Error Message
'member': only static data members can be initialized inside a ref class or value type

A data member inside a managed type cannot be initialized unless it is static.

The following sample generates C3845:

```
// C3845.cpp
// compile with: /clr
using namespace System;

ref class A {
public:
   const Int32 n = 7;   // C3845
};
```

Possible resolution:

```
// C3845b.cpp
// compile with: /clr /c
using namespace System;

ref class A {
public:
   static Int32 n = 7;
};
```

Managed Extensions for C++

The following sample generates C3845:

```
// C3845c.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;

__gc class A {
public:
   const Int32 m = 7;   // C3845
};
```

Possible resolution:

```
// C3845d.cpp
// compile with: /clr:oldSyntax /c
#using <mscorlib.dll>
using namespace System;

__gc class A {
public:
   static Int32 n = 7;
};
```

Compiler Error C3846

Error Message
'symbol' : cannot import symbol from 'assembly2': as 'symbol' has already been imported from another assembly 'assembly1'

A symbol could not be imported from a referenced assembly because it was previously imported from a referenced assembly.

The following sample generates C3846:

```
// C3846a.cpp
// compile with: /LD /clr
public ref struct G
{
};
```

And then compile this:

```
// C3846b.cpp
// compile with: /clr
#using "c3846a.dll"
#using "c3846a.obj"   // C3846

int main()
{
}
```

The following sample generates C3846:

```
// C3846c.cpp
// compile with: /LD /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;

public __gc struct G
{
};
```

And then compile this:

```
// C3846d.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System;
#using "c3846c.dll"
#using "c3846c.obj"   // C3846

int main()
{
}
```

Compiler Error C3848

Error Message
expression having type 'type' would lose some const-volatile qualifiers in order to call 'function'

A variable with a specified const-volatile type can only call member functions defined with same or greater const-volatile qualifications.

The following samples generate C3848:

```
// C3848.cpp
void glbFunc1()
{
}

typedef void (* pFunc1)();

struct S3
{
   operator pFunc1() // const
   {
      return &glbFunc1;
   }
};

int main()
{
   const S3 s3;
   s3();   // C3848, uncomment const qualifier
}
```

Compiler Error C3849

Error Message
function-style call on an expression of type 'type' would lose const and/or volatile qualifiers for all number available operator overloads

A variable with a specified const-volatile type can only call member functions defined with same or greater const-volatile qualifications.

To fix this error, provide an appropriate member function. You cannot execute a conversion on a const or volatile qualified object when the conversion causes loss of qualification. You can gain qualifiers but you cannot lose qualifiers in a conversion.

The following samples generate C3849:

```
// C3849.cpp
void glbFunc3(int i, char c)
{
   i;
}
typedef void (* pFunc3)(int, char);

void glbFunc2(int i)
{
   i;
}

typedef void (* pFunc2)(int);

void glbFunc1()
{
}
typedef void (* pFunc1)();

struct S4
{
   operator ()(int i)
   {
      i;
   }

   operator pFunc1() const
   {
      return &glbFunc1;
   }

   operator pFunc2() volatile
   {
      return &glbFunc2;
   }

   operator pFunc3()
   {
      return &glbFunc3;
   }

   // operator pFunc1() const volatile
   // {
   //    return &glbFunc1;
   // }
};

int main()
{
   // Cannot call any of the 4 overloads of "operator ()(.....)" and
   // "operator pFunc()" because none is declared as "const volatile"
   const volatile S4 s4;  // can only call cv member functions of S4
```

```
    s4();   // C3849 to resolve, uncomment member function
}
```

Compiler Error C3850

Error Message
'char': a universal-character-name specifies an invalid character

With the exception of 0x24 and 0x40, characters in the range of 0 to 0x20 and 0x7f to 0x9f cannot be represented with a universal character name (UCN).

Example
The following samples generate C3850:

```
// C3850.cpp
int main() {
   int \u0019 = 0;   // C3850, can't represent this char as UCN
}
```

Compiler Error C3851

Error Message
'char': a universal-character-name cannot designate a character in the basic character set

For characters in the range of 0x20 to 0x7f, except 0x24 and 0x40, you cannot use a universal character name.

The following samples generate C3851:

```
// C3851.cpp
int main()
{
   int \u0021 = 0;   // C3851
   int \u0024 = 0;   // OK
}
```

Compiler Error C3852

Error Message
'member' having type 'type': aggregate initialization could not initialize this member

An attempt was made to assign a default initialization as part of an aggregate initialization to a data member that cannot receive a default initialization in an aggregate initialization.

The following samples generate C3852:

```
// C3852.cpp
struct S
{
   short s;
};

struct S1
{
   int i;
   const S s;
};

struct S2
{
   int i;
   char & rc;
};

int main()
{
   S1 s1 = { 1 };   // C3852 const member
   // try the following line instead
   // S1 s1 = { 1, 2 };

   S2 s2 = { 2 };   // C3852 reference member
   // try the following line instead
   // char c = 'a';
   S2 s2 = { 2, c };
}
```

Compiler Error C3853

Error Message
'=': re-initializing a reference or assignment through a reference-to-function is illegal

Cannot assign to a reference through a function because functions are not lvalues.

The following samples generate C3853:

```
// C3853.cpp
// compile with: /EHsc
#include <iostream>
int afunc(int i)
{
   return i;
}

typedef int (& rFunc_t)(int);

int main()
{
   rFunc_t rf = afunc;   // OK binding a reference to function
   rf = afunc;   // C3853, can't reassign to a ref that's an lvalue
   int i = 99;
   int & ri = i;
   std::cout << i << std::endl;
   ri = 0;   // OK, i = 88;
   std::cout << i << std::endl;
}
```

Compiler Error C3854

Error Message
expression to left of '=' evaluates to a function. Cannot assign to a function (a function is not an l-value)

A reference cannot be reinitialized. Dereferencing a reference to a function yields a function, which is an rvalue, to which you cannot assign. Therefore, you cannot assign through a reference to a function.

The following sample generates C3854:

```
// C3854.cpp
int afunc(int i)
{
   return i;
}

typedef int (& rFunc_t)(int);
typedef int (* pFunc_t)(int);

int main()
{
   rFunc_t rf = afunc;   // OK binding a reference to function
   pFunc_t pf = &afunc;   // OK initializing a pointer to function

   *pf = &afunc;   // C3854
   // try the following line instead
   // pf = &afunc;
   *rf = &afunc;   // C3854
}
```

Compiler Error C3855

Error Message
'class': type parameter 'param' is incompatible with the declaration

The compiler found nontype template or generic parameters with different names. This can occur when a specified template parameter in the definition of a template specialization is incompatible with its declaration.

The following sample generates C3855:

```
// C3855.cpp
template <int N>
struct C {
   void f();
};

template <char N>
void C<N>::f() {}   // C3855
```

Possible resolution:

```
// C3855b.cpp
// compile with: /c
template <int N>
struct C {
   void f();
};

template <int N>
void C<N>::f() {}
```

C3855 can also occur when using generics:

```
// C3855c.cpp
// compile with: /clr
generic <class T>
ref struct GC1 {
   generic <class U>
   ref struct GC2;
};

generic <class T>
generic <class U>
generic <class V>
ref struct GC1<T>::GC2 { };   // C3855
```

Possible resolution:

```
// C3855d.cpp
// compile with: /clr /c
generic <class T>
ref struct GC1 {
   generic <class U>
   ref struct GC2;
};

generic <class T>
generic <class U>
ref struct GC1<T>::GC2 { };
```

Compiler Error C3856

Error Message
'type': class is not a class type

The most common cause for this error is when there are more generic or template parameter lists at the point of definition than there were at the point of declaration.

The following sample generates C3856:

```
// C3856.cpp
template <class T>
struct S {
   template <class T1>
   struct S1;
   void f();
};

template <class T>   // C3856
template <class T1>
template <class T2>  // extra template parameter list in definition
struct S<T>::S1{};
```

Possible resolution:

```
// C3856b.cpp
// compile with: /c
template <class T>
struct S {
   template <class T1>
   struct S1;
   void f();
};

template <class T>
template <class T1>
struct S<T>::S1{};
```

C3856 can also occur when using generics:

```
// C3856c.cpp
// compile with: /clr
generic <class T>
ref struct GS {
   generic <class U>
   ref struct GS2;
};

generic <class T>
generic <class U>
generic <class V>
ref struct GS<T>::GS2 {};   // C3856
```

Possible resolution:

```
// C3856d.cpp
// compile with: /clr /c
generic <class T>
ref struct GS {
   generic <class U>
   ref struct GS2;
};
```

```
generic <class T>
generic <class U>
ref struct GS<T>::GS2 {};
```

Compiler Error C3857

Error Message
'type': multiple type parameter lists are not allowed

More than one template or generic was specified for the same type, which is not allowed.

The following sample generates C3857:

```
// C3857.cpp
template <class T, class TT>
template <class T2>    // C3857
struct B {};
```

Possible resolution:

```
// C3857b.cpp
// compile with: /c
template <class T, class TT, class T2>
struct B {};
```

C3857 can also occur when using generics:

```
// C3857c.cpp
// compile with: /clr
generic <typename T>
generic <typename U>
ref class GC;   // C3857
```

Possible resolution:

```
// C3857d.cpp
// compile with: /clr /c
generic <typename U>
ref class GC;
```

Compiler Error C3858

Error Message
'type': cannot be redeclared in current scope

The type cannot be declared twice in the same scope.

The following sample generates C3858:

```
// C3858.cpp
// compile with: /LD
template <class T>
struct Outer
{
   struct Inner;
};

template <class T>
struct Outer<T>::Inner;   // C3858
// try the following line instead
// struct Outer<T>::Inner{};
```

Compiler Error C3859

Error Message

virtual memory range for PCH exceeded; please recompile with a command line option of '-Zmvalue' or greater

Your precompiled header is too small for the amount of data the compiler is trying to put in it. Use /Zm to specify a larger value for the precompiled header file.

Compiler Error C3860

Error Message
type argument list following class type name must list parameters in the order used in type parameter list

A generic or template argument list was ill formed.

The following sample generates C3860:

```
// C3860.cpp
// compile with: /LD
template <class T1, class T2>
struct A {
   void f();
};

template <class T2, class T1>
void A<T1, T2>::f() {}   // C3860
```

Possible resolution:

```
// C3860b.cpp
// compile with: /c
template <class T1, class T2>
struct A {
   void f();
};

template <class T2, class T1>
void A<T2, T1>::f() {}
```

C3860 can also occur when using generics:

```
// C3860c.cpp
// compile with: /clr
generic<class T,class U>
ref struct GC {
   void f();
};

generic<class T, class U>
void GC<T,T>::f() {}   // C3860
```

Possible resolution:

```
// C3860d.cpp
// compile with: /clr /c
generic<class T,class U>
ref struct GC {
   void f();
};

generic<class T, class U>
void GC<T,U>::f() {}
```

Compiler Error C3861

Error Message
'identifier': identifier not found

The compiler was not able to resolve a reference to an identifier, even using argument-dependent lookup.

Example

The following sample generates C3861.

```
// C3861.cpp
void f2(){}
int main() {
   f();   // C3861
   f2();   // OK
}
```

Exception classes in the Standard C++ Library are now in the **std** namespace. See Breaking Changes in the Standard C++ Library for more information.

```
// C3861_b.cpp
// compile with: /EHsc
#include <iostream>
int main() {
   try {
      throw exception("Exception");   // C3861
      // try the following line instead
      // throw std::exception("Exception");
   }
   catch (...) {
      std::cout << "caught an exception" << std::endl;
   }
}
```

Compiler Error C3862

Error Message
'function': cannot compile an unmanaged function with /clr:pure or /clr:safe

A compilation with **/clr:pure** or **/clr:safe** will produce an MSIL only image, an image with no native (unmanaged) code. Therefore, you cannot use the **unmanaged** pragma in a **/clr:pure** or **/clr:safe** compilation.

For more information, see /clr (Common Language Runtime Compilation) and managed, unmanaged.

Example
The following sample generates C3862:

```
// C3862.cpp
// compile with: /clr:pure /c
#pragma unmanaged
void f() {}   // C3862
```

Compiler Error C3865

Error Message
'calling_convention' : can only be used on native member functions

A calling convention was used on a function that was either a global function or on a managed member function. The calling convention can only be used on a native (not managed) member function.

For more information, see Calling Conventions.

The following sample generates C3865:

```
// C3865.cpp
// compile with: /clr
// processor: x86
void __thiscall Func(){}   // C3865

// OK
struct MyType {
   void __thiscall Func(){}
};
```

Compiler Error C3866

Error Message
function call missing argument list

Inside a nonstatic member function, a destructor or finalizer call did not have an argument list.

The following sample generates C3866:

```
// C3866.cpp
// compile with: /c
class C {
   ~C();
   void f() {
      this->~C;   // C3866
      this->~C();   // OK
   }
};
```

Compiler Error C3867

Error Message
'func': function call missing argument list; use '&func' to create a pointer to member

You tried to take the address of a member function without qualifying the member function with its class name and the address-of operator.

This error can also be generated as a result of compiler conformance work that was done for Visual C++ 2005: enhanced pointer-to-member conformance. Code that compiled prior to Visual C++ 2005 will now generate C3867. See Breaking Changes in the Visual C++ 2005 Compiler for more information.

Example

In Visual C++ 2005, C3867 can be issued from the compiler with a misleading suggested resolution. Whenever possible, use the most derived class.

The following sample generates C3867.

```
// C3867_1.cpp
// compile with: /c
struct Base {
protected:
   void Test() {}
};

class Derived : public Base {
   virtual void Bar();
};

void Derived::Bar() {
   void (Base::*p1)() = Test;   // C3867
   &Derived::Test;   // OK
}
```

The following sample generates C3867.

```
// C3867_2.cpp
#include<stdio.h>

struct S {
   char *func() {
      return "message";
   }
};

class X {
public:
   void f() {}
};

int main() {
   X::f;   // C3867

   // OK
   X * myX = new X;
   myX->f();

   S s;
   printf_s("test %s", s.func);   // C3867
   printf_s("test %s", s.func());   // OK
}
```

The following sample generates C3867.

```
// C3867_3.cpp
class X {
public:
   void mf(){}
};

int main() {
   void (X::*pmf)() = X::mf;   // C3867

   // try the following line instead
   void (X::*pmf2)() = &X::mf;
}
```

The following sample generates C3867.

```
// C3867_4.cpp
// compile with: /c
class A {
public:
   void f(int) {}

   typedef void (A::*TAmtd)(int);

   struct B {
      TAmtd p;
   };

   void g() {
      B b1;
      b1.p = f;   // C3867
   }
};
```

The following sample generates C3867.

```
// C3867_5.cpp
// compile with: /EHsc
#include <iostream>

class Testpm {
public:
   void m_func1() {
      std::cout << m_num << "\tm_func1\n";
    }

   int m_num;
   typedef void (Testpm::*pmfn1)();
   void func(Testpm* p);
};

void Testpm::func(Testpm* p) {
   pmfn1 s = m_func1;   // C3867
   pmfn1 s2 = &Testpm::m_func1;   // OK
   (p->*s2)();
}

int main() {
   Testpm *pTestpm = new Testpm;
   pTestpm->m_num = 10;

   pTestpm->func(pTestpm);
}
```

Compiler Error C3868

Error Message

'type': constraints on generic parameter 'parameter' differ from those on the declaration

Multiple declarations must have the same generic constraints. For more information, see Generics (Visual C++).

Example

The following sample generates C3868.

```
// C3868.cpp
// compile with: /clr /c
interface struct I1;

generic <typename T> ref struct MyStruct;
generic <typename U> where U : I1 ref struct MyStruct;   // C3868

// OK
generic <typename T> ref struct MyStruct2;
generic <typename U> ref struct MyStruct2;

generic <typename T> where T : I1 ref struct MyStruct3;
generic <typename U> where U : I1 ref struct MyStruct3;
```

Compiler Error C3869

Error Message
gcnew constraint is missing empty parameter list '()'

The **gcnew** special constraint was specified without the empty parameter list. See Constraints for more information.

Example
The following sample generates C3869.

```
// C3869.cpp
// compile with: /c /clr
using namespace System;
generic <typename T>
where T : gcnew   // C3869
// try the following line instead
// where T : gcnew()
ref class List {};
```

Compiler Error C3872

Error Message
'char': this character is not allowed in an identifier

The compiler does not allow a universal character name (UCN) if the hex value is less than 0x20 or in the range 0x7F-0x9F. The UCN cannot designate a character in the basic source character set, and UCNs are not allowed in identifiers.

The following sample generates C3872:

```
// C3872.cpp
int main() {
   int \u0040;   // C3872, in base char set
}
```

Compiler Error C3874

Error Message
return type of 'function' should be 'int' instead of 'type'

A function does not have the return type that was expected by the compiler.

The following sample generates C3874:

```
// C3874b.cpp
double main()
{   // C3874
}
```

Compiler Error C3880

Error Message
'var' : cannot be a literal data member

The type of a literal attribute must be, or compile-time convertible to, one of the following types:

- integral type
- string
- enum with an integral or underlying type

The following sample generates C3880:

```
// C3880.cpp
// compile with: /clr /c
ref struct Y1 {
   literal System::Decimal staticConst1 = 10;   // C3880
   literal int staticConst2 = 10;   // OK
};
```

Compiler Error C3883

Error Message
'var' : an initonly static data member must be initialized

A variable marked with initonly was not initialized correctly.

The following sample generates C3883:

```
// C3883.cpp
// compile with: /clr
ref struct Y1 {
   initonly
   static int staticConst1;   // C3883
};
```

The following sample demonstrates a possible resolution:

```
// C3883b.cpp
// compile with: /clr /c
ref struct Y1 {
   initonly
   static int staticConst2 = 0;
};
```

The following sample shows how to initialize in a static constructor:

```
// C3883c.cpp
// compile with: /clr /LD
ref struct Y1 {
   initonly
   static int staticConst1;

   static Y1() {
      staticConst1 = 0;
   }
};
```

Compiler Error C3886

Error Message
'var' : a literal data member must be initialized

A literal variable must be initialized when it is declaraed.

The following sample generates C3886:

```
// C3886.cpp
// compile with: /clr /c
ref struct Y1 {
   literal int staticConst;   // C3886
   literal int staticConst2 = 0;   // OK
};
```

Compiler Error C3887

Error Message
'var' : the initializer for a literal data member must be a constant expression

A literal data member can only be initialized with a constant expresion.

The following sample generates C3887:

```
// C3887.cpp
// compile with: /clr
ref struct Y1 {
   static int i = 9;
   literal
   int staticConst = i;   // C3887
};
```

Possible resolution:

```
// C3887b.cpp
// compile with: /clr /c
ref struct Y1 {
   literal
   int staticConst = 9;
};
```

Compiler Error C3890

Error Message
'var' : you cannot take the address of a literal data member

A literal data member exists on the garbage-collected heap. An object on the garbage-collected heap can be moved, so taking the address is not useful.

The following sample generates C3890:

```
// C3890.cpp
// compile with: /clr
ref struct Y1 {
   literal int staticConst = 9;
};

int main() {
   int p = &Y1::staticConst;   // C3890
   int p2 = Y1::staticConst;   // OK
}
```

Compiler Error C3891

Error Message
'var' : a literal data member cannot be used as a l-value

A literal variable is const, and its value cannot be changed after it is initialized in the declaration.

The following sample generates C3891:

```
// C3891.cpp
// compile with: /clr
ref struct Y1 {
   literal int staticConst = 9;
};

int main() {
   Y1::staticConst = 0;   // C3891
}
```

Compiler Error C3892

Error Message
'var' : you cannot assign to a variable that is const

A const variable cannot be changed after it is declared and initialized.

The following sample generates C3892:

```
// C3892.cpp
// compile with: /clr
ref struct Y1 {
   static const int staticConst = 9;
};

int main() {
   Y1::staticConst = 0;   // C3892
}
```

Compiler Error C3893

Error Message
'var' : l-value use of initonly data member is only allowed in an instance constructor of class 'type_name'

Static initonly data members can only have their address taken in a static constructor.

Instance (non-static) initonly data members can only have their address taken in instance (non-static) constructors.

The following sample generates C3893:

```
// C3893.cpp
// compile with: /clr
ref struct Y1 {
   Y1() : data_var(0) {
      int% i = data_var;   // OK
   }
   initonly int data_var;
};

int main(){
   Y1^ y= gcnew Y1;
   int% i = y->data_var;   // C3893
}
```

Compiler Error C3894

Error Message
'var' : l-value use of initonly static data member is only allowed in the class constructor of class 'class'

Static initonly data members can only be used as l-values at their point of declaration, or in a static constructor.

Instance (non-static) initonly data members can only be used as l-values at their point of declaration, or in instance (non-static) constructors.

The following sample generates C3894:

```
// C3894.cpp
// compile with: /clr
ref struct Y1 {
   initonly static int data_var = 0;

public:
   // class constructor
   static Y1() {
      data_var = 99;   // OK
      System::Console::WriteLine("in static constructor");
   }

   // not the class constructor
   Y1(int i) {
      data_var = i;   // C3894
   }

   static void Test() {}

};

int main() {
   Y1::data_var = 88;   // C3894
   int i = Y1::data_var;
   Y1 ^ MyY1 = gcnew Y1(99);
   Y1::Test();
}
```

Compiler Error C3895

Error Message
'var' : type data members cannot be 'volatile'

Certain kinds of data members, for example literal or initonly, cannot be volatile (C++).

The following sample generates C3895:

```
// C3895.cpp
// compile with: /clr
ref struct Y1 {
   initonly
   volatile int data_var2;   // C3895
};
```

Compiler Error C3896

Error Message
'member' : improper initializer: this literal data member can only be initialized with 'nullptr'

A literal (Visual C++) data member was initialized incorrectly. See nullptr for more information.

The following sample generates C3896:

```
// C3896.cpp
// compile with: /clr /c
ref class R{};

value class V {
   literal R ^ r = "test";   // C3896
   literal R ^ r2 = nullptr;   // OK
};
```

Compiler Error C3897

Error Message
'var' : only static initonly data members can have an initializer expression

An initonly data member can be initialized at declaration only if it is also static.

The following sample generates C3897:

```
// C3897.cpp
// compile with: /clr /c
ref struct Y1 {
   initonly int data_var1 = 9;   // C3897

   initonly static int data_var2 = 9;   // OK

   initonly int data_var3;   // declare initonly instance
   Y1() { data_var3 = 99; }   // initialize initonly instance
};
```

Compiler Error C3898

Error Message
'var' : type data members can only be members of managed types

An initonly data member was declared in a native class. An **initonly** data member can only be declared in a CLR class.

The following sample generates C3898:

```
// C3898.cpp
// compile with: /clr
struct Y1 {
   initonly
   static int data_var = 9;   // C3898
};
```

Possible resolution:

```
// C3898b.cpp
// compile with: /clr /c
ref struct Y1 {
   initonly
   static int data_var = 9;
};
```

Compiler Error C3899

Error Message
'var' : l-value use of initonly data member is not allowed directly within a parallel region in class 'class'

An initonly data member cannot be initialized inside that part of a constructor that is in a parallel region. This is because the compiler does an internal relocation of that code, such that, it is effectively no longer part of the constructor.

To resolve, initialize the initonly data member in the constructor, but outside the parallel region.

Example
The following sample generates C3899.

```
// C3899.cpp
// compile with: /clr /openmp
#include <omp.h>

public ref struct R {
   initonly int x;
   R() {
      x = omp_get_thread_num() + 1000;   // OK
      #pragma omp parallel num_threads(5)
      {
         // cannot assign to 'x' here
         x = omp_get_thread_num() + 1000;   // C3899
         System::Console::WriteLine("thread {0}", omp_get_thread_num());
      }
      x = omp_get_thread_num() + 1000;   // OK
   }
};

int main() {
   R^ r = gcnew R;
   System::Console::WriteLine(r->x);
}
```

Compiler Error C3900

Error Message
'member': not allowed in current scope

Property blocks can contain function declarations and inline function definitions only. No members other than functions are allowed in property blocks. No typedefs, operators, or friend functions are allowed. For more information, see property.

Event definitions can only contain access methods and functions.

The following sample generates C3900:

```
// C3900.cpp
// compile with: /clr
ref class X {
   property int P {
      void set(int);   // OK
      int i;   // C3900 variable declaration
   };
};
```

The following sample generates C3900:

```
// C3900b.cpp
// compile with: /clr
using namespace System;
delegate void H();
ref class X {
   event H^ E {
      int m;   // C3900

      // OK
      void Test() {}

      void add( H^ h ) {}
      void remove( H^ h ) {}
      void raise( ) {}
   }
};
```

Compiler Error C3901

Error Message
'accessor_function': must have return type 'type'

At least one get method's return type must match the property type. For more information, see property.

The following sample generates C3901:

```
// C3901.cpp
// compile with: /clr /c
using namespace System;
ref class X {
   property String^ Name {
      void get();   // C3901
      // try the following line instead
      // String^ get();
   };
};

ref class Y {
   property double values[int, int] {
      int get(int, int);   // C3901
      // try the following line instead
      // double get(int, int);
   };
};
```

Compiler Error C3902

Error Message
'accessor': type of last parameter must be 'type'

The type of the last parameter of at least one set method must match the type of the property. For more information, see property.

The following sample generates C3902:

```
// C3902.cpp
// compile with: /clr /c
using namespace System;
ref class X {
   property String ^Name {
      void set(int);   // C3902
      // try the following line instead
      // void set(String^){}
   }

   property double values[int,int] {
      void set(int, int, float);   // C3902
      // try the following line instead
      // void set(int, int, double){}
   }
};
```

Compiler Error C3903

Error Message
'property': does not have set or get method

A property must have at least a `get` or a `set` method. For more information, see property.

The following sample generates C3903:

```
// C3903.cpp
// compile with: /clr
ref class X {
   property int P {
      void f(int){}
      // Add one or both of the following lines.
      // void set(int){}
      // int get(){return 0;}
   };   // C3903

   property double Q[,,,,] {
      void f(){}
      // Add one or both of the following lines.
      // void set(int, char, int, char, double, double){}
      // double get(int, char, int, char, double){return 1.1;}
   }   // C3903
};
```

Compiler Error C3904

Error Message
'property_accessor': must specify number parameter(s)

Check the number of parameters in your `get` and `set` methods against property dimensions.

- The number of parameters for the `get` method must equal the number of dimensions of the property or be zero for nonindexed properties.
- The number of parameters of the `set` method must be one more than the number of dimensions of the property.

For more information, see property.

Example
The following sample generates C3904.

```
// C3904.cpp
// compile with: /clr /c
ref class X {
   property int P {
      // set
      void set();   // C3904
      // try the following line instead
      // void set(int);

      // get
      int get(int, int);   // C3904
      // try the following line instead
      // int get();
   };
};
```

The following sample generates C3904.

```
// C3904b.cpp
// compile with: /clr /c
ref struct X {
   property int Q[double, double, float, float, void*, int] {
      // set
      void set(double, void*);   // C3904
      // try the following line instead
      // void set(double, double, float, float, void*, int, int);

      // get
      int get();   // C3904
      // try the following line instead
      // int get(double, double, float, float, void*, int);
   }
};
```

Compiler Error C3908

Error Message
access level less restrictive than 'construct'

A property accessor method (get or set) cannot have less restrictive access than the access specified on the property itself. Similarly, for event accessor methods.

For more information, see property and event.

The following sample generates C3908:

```
// C3908.cpp
// compile with: /clr
ref class X {
protected:
   property int i {
   public:   // C3908 property i is protected
      int get();
   private:
      void set(int);   // OK more restrictive
   };
};
```

Compiler Error C3909

Error Message
a managed event declaration must occur in a managed type

A managed event was declared in a native type.

For more information, see event.

The following sample generates C3909:

```
// C3909.cpp
// compile with: /clr /c
delegate void H();
class X {
   event H^ E;   // C3909
};

ref class Y {
   static event H^ E {
      void add(H^) {}
      void remove( H^ h ) {}
      void raise( ) {}
   }
};
```

Compiler Error C3910

Error Message
'event': must define member 'method'

An event was defined, but did not contain the specified, required accessor method.

For more information, see event.

The following sample generates C3910:

```
// C3910.cpp
// compile with: /clr /c
delegate void H();
ref class X {
   event H^ E {
      // uncomment the following lines
      // void add(H^) {}
      // void remove( H^ h ) {}
      // void raise( ) {}
   };   // C3910

   event H^ E2; // OK data member
};
```

Compiler Error C3911

Error Message
'event_accessor_method': function must have type 'signature'

An event's accessor method was not properly declared.

For more information, see event.

The following sample generates C3911:

```
// C3911.cpp
// compile with: /clr
using namespace System;
delegate void H(String^, int);

ref class X {
   event H^ E1 {
      void add() {}   // C3911
      // try the following line instead
      // void add(H ^ h) {}

      void remove(){}
      // try the following line instead
      // void remove(H ^ h) {}

      void raise(){}
      // try the following line instead
      // void raise(String ^ s, int i) {}
   };
};
```

Compiler Error C3912

Error Message
'event': type of event must be a delegate type

An event was declared but was not of the proper type.

For more information, see event.

The following sample generates C3912:

```
// C3912.cpp
// compile with: /clr
delegate void H();
ref class X {
   event int Ev;   // C3912
   event H^ Ev2;   // OK
};
```

Compiler Error C3913

Error Message
default property must be indexed

A default property was defined incorrectly.

For more information, see property.

The following sample generates C3913:

```
// C3913.cpp
// compile with: /clr /c
ref struct X {
   property int default {   // C3913
   // try the following line instead
   // property int default[int] {
      int get(int) { return 0; }
      void set(int, int) {}
   }
};
```

Compiler Error C3914

Error Message
a default property cannot be static

A default property was declared incorrectly. For more information, see How to: Use Indexed Properties.

Example
The following sample generates C3914.

```
// C3914.cpp
// compile with: /clr /c
ref struct X {
   static property int default[int] {   // C3914
   // try the following line instead
   // property int default[int] {
      int get(int) { return 0; }
      void set(int, int) {}
   }
};
```

Compiler Error C3915

Error Message
'type' has no default indexed property (class indexer)

A type does not have a default, indexed property.

For more information, see property.

Example

The following sample generates C3915.

```
// C3915.cpp
// compile with: /clr
ref class X {
public:
// uncomment property to resolve this C3915
//   property int default[]
//   {
//      int get(int i)
//      {
//         return 863;
//      }
//   }
};

int main() {
   X^ x = new X;
   System::Console::WriteLine(x[1]);   // C3915
}
```

C3915 can also occur if you attempt to consume a default indexer in the same compiland where it was defined with DefaultMemberAttribute.

The following sample generates C3915.

```
// C3915_b.cpp
// compile with: /clr
using namespace System;

[Reflection::DefaultMember("XXX")]
ref struct A {
   property Double XXX[Double] {
      Double get(Double data) {
         return data*data;
      }
   }
};

ref struct B {
   property Double default[Double] {
      Double get(Double data) {
         return data*data;
      }
   }
};

int main() {
   A ^ mya = gcnew A();
   Console::WriteLine("{0}", mya[3]);   // C3915

   B ^ myb = gcnew B();
   Console::WriteLine("{0}", myb[3]);   // OK
}
```

Compiler Error C3917

Error Message

'property': obsolete construct declaration style

A property or event definition used syntax from a previous version.

If you want to use syntax from a previous version, use /clr:oldSyntax.

For more information, see property.

Example

```
// C3917.cpp
// compile with: /clr /c
public ref class  C {
private:
   int m_length;
public:
   C() {
      m_length = 0;
   }

   property int get_Length();   // C3917

   // The correct property definition:
   property int Length {
      int get() {
         return m_length;
      }

      void set( int i ) {
         m_length = i;
      }
   }
};
```

Compiler Error C3918

Error Message
usage requires 'member' to be a data member

C3918 can occur for several reasons related to events.

Example

C3918 can occur because a class member is required in the current context. The following sample generates C3918.

```
// C3918.cpp
// compile with: /clr /c
public ref class C {
public:
   System::Object ^ o;
   delegate void Del();

   event Del^ MyEvent {
      void add(Del^ev) {
         if ( MyEvent != nullptr) {}   // C3918
         if ( o != nullptr) {}   // OK
   }
   void remove(Del^){}
   }
};
```

C3918 is also caused if you try to check a trivial event for null (the event name will no longer provide direct access to the backing store delegate for the event).

The following sample generates C3918.

```
// C3918_2.cpp
// compile with: /clr /c
using namespace System;
public delegate int MyDel(int);

interface struct IEFace {
   event MyDel ^ E;
};

ref struct EventSource : public IEFace {
   virtual event MyDel ^ E;
   void Fire_E(int i) {
      if (E)   // C3918
         E(i);
   }
};
```

C3918 can also occur if you incorrectly subscribe to an event. The following sample generates C3918.

```
// C3918_3.cpp
// compile with: /clr /c
using namespace System;

public delegate void del();

public ref class A {
public:
   event del^ e {
      void add(del ^handler ) {
         d += handler;
      }
```

```
      void remove(del ^handler) {
         d -= handler;
      }

      void raise() {
         d();
      }
   }

   del^ d;
   void f() {}

   A() {
      e = gcnew del(this, &A::f);   // C3918
      // try the following line instead
      // e += gcnew del(this, &A::f);
      e();
   }
};

int main() {
   A a;
}
```

Compiler Error C3919

Error Message
'event_method': function must have type 'type'

An event accessor method was not declared correctly.

For more information about events, see event.

The following sample generates C3919:

```
// C3919.cpp
// compile with: /clr /c
using namespace System;
delegate void D(String^);
ref class R {
   event D^ e {
      int add(int);   // C3919
      int remove(int);   // C3919

      void add(D^);   // OK
      void remove(D^);   // OK
   }
};
```

Compiler Error C3920

Error Message
'operator'' : cannot define a postfix increment/decrement CLR operator

The common language runtime does not support the postfix operator and user-defined postfix operators are not allowed. You can define a prefix operator and the prefix operator will be used for both pre- and post-increment operations.

The following sample generates C3920:

```
// C3920.cpp
// compile with: /clr /LD
public value struct V {
   static V operator ++(V me, int)
   // try the following line instead
   // static V operator ++(V me)
   {   // C3920
      me.m_i++;
      return me;
   }

   int m_i;
};
```

Compiler Error C3921

Error Message

Use of S-prefixed strings requires /clr:oldSyntax command line option

In Visual C++ 2005, you no longer need to explicitly specify that a string is a CLR string.

Example

The following sample generates C3921.

```
// C3921.cpp
// compile with: /clr
using namespace System;

int main() {
   String ^ s = S"CLR string literal";   // C3921

   // try the following line instead
   String ^ s2 = "CLR string literal";
}
```

Compiler Error C3923

Error Message

'member' : local class, struct, or union definitions are not allowed in a member function of a managed class

Example

The following sample generates C3923.

```
// C3923.cpp
// compile with: /clr /c
ref struct x {
   void Test() {
      struct a {};   // C3923
      class b {};   // C3923
      union c {};   // C3923
   }
};
```

Compiler Warnings C4001 Through C4099

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, type the error number in the **Look for** box in the Index, or click a link below.

Compiler Warning (level 4) C4001

Compiler Warning (level 1) C4002

Compiler Warning (level 1) C4003

Compiler Warning (level 1) C4005

Compiler Warning (level 1) C4006

Compiler Warning (level 2) C4007

Compiler Warning (levels 2 and 3) C4008

Compiler Warning (level 1) C4010

Compiler Warning (level 3) C4013

Compiler Warning (level 1) C4015

Compiler Warning (level 3) C4018

Compiler Warning (level 4) C4019

Compiler Warning (level 1) C4020

Compiler Warning (level 1) C4022

Compiler Warning (level 3) C4023

Compiler Warning (level 1) C4024

Compiler Warning (level 1) C4025

Compiler Warning (level 1) C4026

Compiler Warning (level 1) C4027

Compiler Warning (level 1) C4028

Compiler Warning (level 1) C4029

Compiler Warning (level 1) C4030

Compiler Warning (level 1) C4031

Compiler Warning (level 4) C4032

Compiler Warning (level 1) C4033

Compiler Warning (level 1) C4034

Compiler Warning (level 1) C4036

Compiler Warning (level 1) C4038

Compiler Warning (level 1) C4041

Compiler Warning (level 1) C4042

Compiler Warning (level 1) C4045

Compiler Warning (level 1) C4047

Compiler Warning (level 1) C4048

Compiler Warning (level 1) C4049

Compiler Warning (level 2) C4051

Compiler Warning (level 1) C4052

Compiler Warning (level 4) C4053

Compiler Warning (level 1) C4054

Compiler Warning (level 1) C4055

Compiler Warning (level 2) C4056

Compiler Warning (level 4) C4057

Compiler Warning (level 3) C4060

Compiler Warning (level 4) C4061

Compiler Warning (level 3) C4062

Compiler Warning (level 4) C4063

Compiler Warning (level 4) C4064

Compiler Warning (level 3) C4065

Compiler Warning (level 3) C4066

Compiler Warning (level 1) C4067

Compiler Warning (level 1) C4068

Compiler Warning (level 3) C4069

Compiler Warning (level 3) C4073

Compiler Warning (level 1) C4074

Compiler Warning (level 1) C4075

Compiler Warning (level 1) C4076

Compiler Warning (level 1) C4077

Compiler Warning (level 1) C4079

Compiler Warning (level 1) C4080

Compiler Warning (level 1) C4081

Compiler Warning (level 1) C4083

Compiler Warning (level 1) C4085

Compiler Warning (level 1) C4086

Compiler Warning (level 1) C4087

Compiler Warning (level 1) C4088

Compiler Warning (level 1) C4089

Compiler Warning (level 1) C4090

Compiler Warning (level 1) C4091

Compiler Warning (level 4) C4092

Compiler Warning (level 2) C4094

Compiler Warning (level 1) C4096

Compiler Warning (level 1) C4097

Compiler Warning (level 1) C4098

Compiler Warning (level 2) C4099

Compiler Warning (level 4) C4001

Error Message
nonstandard extension 'single line comment' was used

Single-line comments are standard in C++ and nonstandard in C. Under strict ANSI compatibility (/Za), C files that contain single-line comments, generate C4001 due to the usage of a nonstandard extension. Since single-line comments are standard in C++, C files containing single-line comments do not produce C4001 when compiling with Microsoft extensions (/Ze).

Example

To disable warning, uncomment #pragma warning(disable:4001).

```
// C4001.cpp
// compile with: /W4 /Za /TC
// #pragma warning(disable:4001)
int main()
{
   // single-line comment in main
   // C4001
}
```

Compiler Warning (level 1) C4002

Error Message
too many actual parameters for macro 'identifier'

The number of actual parameters in the macro exceeds the number of formal parameters in the macro definition. The preprocessor collects the extra parameters but ignores them during macro expansion.

C4002 can occur when incorrectly using Variadic Macros.

The following sample generates C4002:

```
// C4002.cpp
// compile with: /W1
#define test(a) (a)

int main() {
   int a = 1;
   int b = 2;
   a = test(a,b);   // C4002
   // try..
   a = test(a);
}
```

This error can also be generated as a result of compiler conformance work that was done for Visual Studio .NET 2003: extra commas in macro no longer accepted.

The compiler will no longer accept extra commas in a macro. For code to be valid in both the Visual Studio .NET 2003 and Visual Studio .NET versions of Visual C++, remove the extra commas.

See Summary of Compiler-Time Breaking Changes for more information.

```
// C4002b.cpp
// compile with: /W1
#define F(x,y)
int main()
{
   F(2,,,,,,3,,,,,,)   // C4002
   // Try the following line instead:
   // F(2,3)
}
```

Compiler Warning (level 1) C4003

Error Message
not enough actual parameters for macro 'identifier'

The number of formal parameters in the macro definition exceeds the number of actual parameters in the macro. Macro expansion substitutes empty text for the missing parameters.

The following sample generates C4003:

```
// C4003.cpp
// compile with: /WX
#define test(a,b) (a+b)

int main()
{
   int a = 1;
   int b = 2;
   a = test(b);   // C4003
   // try..
   a = test(a,b);
}
```

Compiler Warning (level 1) C4005

Error Message
'identifier' : macro redefinition

The macro identifier is defined twice. The compiler uses the second macro definition.

To fix by checking the following possible causes

1. Defining a macro on the command line and in the code with a **#define** directive.
2. Macros imported from include files.

To fix by using the following possible solutions

1. Remove one of the definitions.
2. Use an #undef directive before the second definition.

The following sample generates C4005:

```
// C4005.cpp
// compile with: /W1 /EHsc
#include <iostream>
using namespace std;

#define TEST "test1"
#define TEST "test2"   // C4005 delete or rename to resolve the warning

int main() {
   cout << TEST << endl;
}
```

Compiler Warning (level 1) C4006

Error Message
#undef expected an identifier

The **#undef** directive did not specify an identifier to undefine. The directive is ignored. To resolve the warning, be sure to specify an identifier. The following sample generates C4006:

```
// C4006.cpp
// compile with: /W1
#undef   // C4006

// try..
// #undef TEST

int main() {
}
```

Compiler Warning (level 2) C4007

Error Message
'function' : must be 'attribute'

A required attribute for a function is not explicitly stated. For example, the function **main** must have the **__cdecl** attribute. The compiler forces the attribute.

Compiler Warning (levels 2 and 3) C4008

Error Message
'identifier' : 'attribute' attribute ignored

The compiler ignored the **__fastcall**, **static**, or **inline** attribute for a function (level 3 warning) or data (level 2 warning).

To fix by checking the following possible causes

1. **__fastcall** attribute with data.
2. **static** or **inline** attribute with **main** function.

Compiler Warning (level 1) C4010

Error Message
single-line comment contains line-continuation character

A single-line comment, which is introduced by //, contains a backslash (\) that serves as a line-continuation character. The compiler considers the next line to be a continuation and treats it as a comment.

Some syntax-directed editors do not indicate the line following the continuation character as a comment. Ignore syntax coloring on any lines that cause this warning.

The following sample generates C4010:

```
// C4010.cpp
// compile with: /WX
int main() {
   // the next line is also a comment because of the backslash \
   int a = 3; // C4010
   a++;
}
```

Compiler Warning (level 3) C4013

Error Message
'function' undefined; assuming extern returning int

The compiler encountered a call to an undefined function.

To fix by checking the following possible causes

1. Incorrect spelling of function name
2. External functions not prototyped as **extern**

Compiler Warning (level 1) C4015

Error Message
'identifier' : type of bit field must be integral

The bit field is not declared as an integer type. The compiler assumes the base type of the bit field to be unsigned. Bit fields must be declared as unsigned integer types.

Compiler Warning (level 3) C4018

Error Message
'expression' : signed/unsigned mismatch

Comparing a signed and unsigned number required the compiler to convert the signed value to unsigned.

This warning may be fixed if you cast one of the two types when testing signed and unsigned types.

The following sample generates C4018:

```
// C4018.cpp
// compile with: /W3
int main() {
   unsigned int uc = 0;
   int c = 0;
   unsigned int c2 = 0;

   if (uc < c) uc = 0;   // C4018

   // OK
   if (uc == c2) uc = 0;
}
```

Compiler Warning (level 4) C4019

Error Message
empty statement at global scope

A semicolon at global scope is not preceded by a statement.

This warning may be fixed if you remove the extra semicolon.

Example

```
// C4019.c
// compile with: /Za /W4
#define declint( varname ) int varname;
declint( a );   // C4019, int a;;
declint( b )   // OK, int b;

int main()
{
}
```

Compiler Warning (level 1) C4020

Error Message
'function' : too many actual parameters

The number of actual parameters in a function call exceeds the number of formal parameters in the function prototype or definition. The compiler passes the extra actual parameters according to the calling convention of the function.

The following sample generates C4020:

```
// C4020.c
// compile with: /W1 /c
void f(int);
int main() {
   f(1,2);   // C4020
}
```

Possible resolution:

```
// C4020b.c
// compile with: /c
void f(int);
int main() {
   f(1);
}
```

Compiler Warning (level 1) C4022

Error Message
'function' : pointer mismatch for actual parameter 'number'

The pointer type of the actual parameter differs from the pointer type of the corresponding formal parameter. The actual parameter is passed without change.

Compiler Warning (level 3) C4023

Error Message
'symbol' : based pointer passed to unprototyped function : parameter number

Passing a based pointer to an unprototyped function causes the pointer to be normalized, with unpredictable results.

This warning may be fixed if you use prototype functions that are passed based pointers.

Compiler Warning (level 1) C4024

Error Message
'function' : different types for formal and actual parameter 'number'

Corresponding formal and actual parameters have different types. The compiler passes the actual parameter without change. The receiving function converts the parameter type to the type expected.

Compiler Warning (level 1) C4025

Error Message
'number' : based pointer passed to function with variable arguments: parameter number

Passing a based pointer to a function with variable arguments causes the pointer to be normalized, with unpredictable results. Do not pass based pointers to functions with variable arguments.

Compiler Warning (level 1) C4026

Error Message
function declared with formal parameter list

The function declaration has formal parameters, but the function definition does not. Subsequent calls to this function assume that the function takes no parameters.

Compiler Warning (level 1) C4027

Error Message
function declared without formal parameter list

The function declaration no formal parameters, but there are formal parameters in the function definition or actual parameters in a call. Subsequent calls to this function assume that the function takes actual parameters of the types found in the function definition or call.

Compiler Warning (level 1) C4028

Error Message
formal parameter 'number' different from declaration

The type of the formal parameter does not agree with the corresponding parameter in the declaration. The type in the original declaration is used.

This warning is only valid for C source code.

Example
The following sample generates C4028.

```
// C4028.c
// compile with: /W1 /Za
void f(int , ...);
void f(int i, int j) {}   // C4028

void g(int , int);
void g(int i, int j) {}   // OK

int main() {}
```

Compiler Warning (level 1) C4029

Error Message
declared formal parameter list different from definition

Formal parameter types in the function declaration do not agree with those in the function definition. The compiler uses the parameter list from the definition.

Compiler Warning (level 1) C4030

first formal parameter list longer than the second list

A function was redeclared with different formal parameters. The compiler uses the formal parameters given in the first declaration.

Compiler Warning (level 1) C4031

Error Message
second formal parameter list longer than the first list

A function is redeclared with different formal parameters. The compiler uses the formal parameters given in the first declaration.

Compiler Warning (level 4) C4032

Error Message
formal parameter 'number' has different type when promoted

The parameter type is not compatible, through default promotions, with the type in a previous declaration.

This is an error in ANSI C (/Za) and a warning under Microsoft extensions (/Ze).

Example

```
// C4032.c
// compile with: /W4
void func();
void func(char);   // C4032, char promotes to int

int main()
{
}
```

Compiler Warning (level 1) C4033

Error Message
'function' must return a value

The function does not return a value. An undefined value is returned.

Functions that use **return** without a return value must be declared as type **void**.

This error is for C language code.

The following sample generates C4033:

```
// C4033.c
// compile with: /W1 /LD
int test_1(int x)   // C4033 expected
{
   if (x)
   {
      return;   // C4033
   }
}
```

Compiler Warning (level 1) C4034

Error Message
sizeof returns 0

The **sizeof** operator is applied to an operand of size zero (an empty structure, union, class, or enumerated type, or type **void**).

Compiler Warning (level 1) C4036

Error Message
unnamed 'type' as actual parameter

No type name is given for a structure, union, enumeration, or class used as an actual parameter. If you are using /Zg to generate function prototypes, the compiler issues this warning and comments out the formal parameter in the generated prototype.

Specify a type name to resolve this warning.

Example

The following sample generates C4036.

```
// C4036.c
// compile with: /Zg /W1
typedef struct { int i; } T;
void f(T* t) {}   // C4036

// OK
typedef struct MyStruct { int i; } T2;
void f2(T2 * t) {}
```

Compiler Warning (level 1) C4038

Error Message
'modifier' : illegal ambient class modifier

This modifier cannot be used for classes with **dllimport** or dllexport attributes.

Compiler Warning (level 1) C4041

Error Message
compiler limit : terminating browser output

Browser information exceeded the compiler limit.

This warning can be caused by compiling with /FR (browser information including local variables).

To fix by using the following possible solutions

1. Compile with /Fr (browser information without local variables).
2. Disable browser output (compile without /FR or /Fr).

Compiler Warning (level 1) C4042

Error Message
'identifier' : has bad storage class

The specified storage class cannot be used with this identifier in this context. The compiler uses the default storage class instead:

- **extern**, if *identifier* is a function.
- **auto**, if *identifier* is a formal parameter or local variable.
- No storage class, if *identifier* is a global variable.

This warning can be caused by specifying a storage class other than **register** in a parameter declaration.

The following sample generates C4042

```
// C4042.cpp
// compile with: /W1 /LD
int func2( __declspec( thread ) int tls_i )    // C4042
// try the following line instead
// int func2( int tls_i )
{
   return tls_i;
}
```

Compiler Warning (level 1) C4045

Error Message
'array' : array bounds overflow

The array has too many initializers. Extra initializers are ignored.

Make sure that array elements and initializers match in size and quantity.

Compiler Warning (level 1) C4047

Error Message
'operator' : 'identifier1' differs in levels of indirection from 'identifier2'

A pointer can point to a variable (one level of indirection), to another pointer that points to a variable (two levels of indirection), and so on.

Example

The following sample generates C4047:

```
// C4047.c
// compile with: /W1

int main() {
   char **p = 0;   // two levels of indirection
   char *q = 0;   // one level of indirection

   char *p2 = 0;   // one level of indirection
   char *q2 = 0;   // one level of indirection

   p = q;   // C4047
   p2 = q2;
}
```

The following sample generates C4047:

```
// C4047b.c
// compile with: /W1
#include <stdio.h>

int main() {
   int i;
   FILE *myFile = NULL;
   errno_t  err = 0;
   char file_name[256];
   char *cs = 0;

   err = fopen_s(&myFile, "C4047.txt", "r");
   if ((err != 0) || (myFile)) {
      printf_s("fopen_s failed!\n");
      exit(-1);
    }
   i = fgets(file_name, 256, myFile);   // C4047
   cs = fgets(file_name, 256, myFile);   // OK
}
```

Compiler Warning (level 1) C4048

Error Message
different declared array subscripts : 'identifier1' and 'identifier2'

An expression involves pointers to arrays of different size. The pointers are used without conversion.

This warning may be fixed if you explicitly cast the arrays to the same or equivalent type.

Compiler Warning (level 1) C4049

Error Message
compiler limit : terminating line number emission

The file contains more than 16,777,215 (2^{24}-1) source lines. The compiler stops numbering at 16,777,215.

For code after line 16,777,215:

- The image will contain no debug information for line numbers.
- Some diagnostics may be reported with incorrect line numbers.
- .asm listings (/FAs) may have incorrect line numbers.

Compiler Warning (level 2) C4051

Error Message
type conversion; possible loss of data

An expression contains two data items with different base types. Converting one type causes the data item to be truncated.

This warning may be fixed if you cast the data items to the appropriate type.

Compiler Warning (level 1) C4052

Error Message
function declarations different; one contains variable arguments

One declaration of the function does not contain variable arguments. It is ignored.

The following sample generates C4052:

```
// C4052.c
// compile with: /W4 /c
int f();
int f(int i, ...);   // C4052
```

Compiler Warning (level 4) C4053

Error Message
one void operand for '?:'

The **?:** operator is given an expression of type **void**. The value of the **void** operand is undefined.

Compiler Warning (level 1) C4054

Error Message
'conversion' : from function pointer 'type1' to data pointer 'type2'

A function pointer is cast (possibly incorrectly) to a data pointer. This is a level 1 warning under /Za and a level 4 warning under /Ze.

The following sample generates C4054:

```
// C4054.c
// compile with: /W1 /Za /c
int (*pfunc)();
int* f() {
   return (int*)pfunc;   // C4054
}
```

Under /Ze, this is a level 4 warning.

```
// C4054b.c
// compile with: /W4 /c
int (*pfunc)();
int* f() {
   return (int*)pfunc;   // C4054
}
```

Compiler Warning (level 1) C4055

Error Message
'conversion' : from data pointer 'type1' to function pointer 'type2'

A data pointer is cast (possibly incorrectly) to a function pointer. This is a level 1 warning under /Za and a level 4 warning under /Ze.

The following sample generates C4055:

```
// C4055.c
// compile with: /Za /W1 /c
typedef int (*PFUNC)();
int *pi;
PFUNC f() {
   return (PFUNC)pi;   // C4055
}
```

Under /Ze, this is a level 4 warning.

```
// C4055b.c
// compile with: /W4 /c
typedef int (*PFUNC)();
int *pi;
PFUNC f() {
return (PFUNC)pi;   // C4055
}
```

Compiler Warning (level 2) C4056

Error Message
overflow in floating point constant arithmetic

Floating-point constant arithmetic generates a result that exceeds the maximum allowable value.

This warning can be caused by compiler optimizations performed during constant arithmetic. You can safely ignore this warning if it goes away when you turn off optimization (/Od).

The following sample generates C4056:

```
// C4056.cpp
// compile with: /W2 /LD
#pragma warning (default : 4056)
float fp_val = 1.0e300 * 1.0e300;   // C4056
```

Compiler Warning (level 4) C4057

Error Message
'operator' : 'identifier1' indirection to slightly different base types from 'identifier2'

Two pointer expressions refer to different base types. The expressions are used without conversion.

To fix by checking the following possible causes

1. Mixing signed and unsigned types.
2. Mixing **short** and **long** types.

Compiler Warning (level 3) C4060

Error Message
switch statement contains no 'case' or 'default' labels

The following sample generates C4060:

```
// C4060.cpp
// compile with: /W3
int main()
{
   int i = 0;
   switch( i );   // C4060
   // correct syntax for switch block
   switch( i )
   {
      case '0':
         i++;
      default:
         i = 9;
   }
}
```

Compiler Warning (level 4) C4061

Error Message
enumerator 'identifier' in switch of enum 'enumeration' is not explicitly handled by a case label

The enumerate has no associated handler in a **switch** statement.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4061:

```
// C4061.cpp
// compile with: /W4
#pragma warning(default : 4061)

enum E { a, b, c };
void func ( E e )
{
   switch(e)
   {
      case a:
      case b:
      default:
         break;
   }   // C4061 c' not handled
}

int main()
{
}
```

Compiler Warning (level 3) C4062

Error Message
enumerator 'identifier' in switch of enum 'enumeration' is not handled

The enumerate has no associated handler in a **switch** statement, and there is no **default** label.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4062:

```
// C4062.cpp
// compile with: /W3
#pragma warning(default : 4062)
enum E { a, b, c };
void func ( E e ) {
   switch(e) {
      case a:
      case b:
      break;   // no default label
   }   // C4062, enumerate 'c' not handled
}

int main() {
}
```

Compiler Warning (level 4) C4063

Error Message
case 'identifier' is not a valid value for switch of enum 'enumeration'

The expression in a switch statement is of an enum type, and one of the case values is not one of the enumerators in that enum.

The following sample generates C4063

```
// C4063.cpp
// compile with: /W4 /c
enum E { a, b };
void func ( E e ) {
   switch(e) {
      case a:
      case b:
      case 7:   // switch value not valid
      break;
   }   // C4063
}
```

Compiler Warning (level 4) C4064

Error Message
switch of incomplete enum 'enumeration'

A **switch** statement specifies a selector of an enumerated type that is not defined.

The following sample generates C4064:

```
// C4064.cpp
// compile with: /W4
enum E;
void func ( E e )
{
   switch(e)
   {
   case 7:
   break;  // define members for E to resolve
   }   // C4064
}

int main()
{
}
```

Compiler Warning (level 3) C4065

Error Message
switch statement contains 'default' but no 'case' labels

The **switch** statement specifies a default case only. This is equivalent to a sequence of statements.

Example

```
// C4065.cpp
// compile with: /W3
void func(int i)
{
   switch(i)
   {
      default:
         i++;
   }   // C4065
}

int main()
{
}
```

Compiler Warning (level 3) C4066

Error Message
characters beyond first in wide-character constant ignored

The compiler processes only the first character of a wide-character constant.

Compiler Warning (level 1) C4067

Error Message
unexpected tokens following preprocessor directive - expected a newline

The compiler found, and ignored, extra characters following a preprocessor directive. This warning appears only under ANSI compatibility (/Za).

```
// C4067a.cpp
// compile with: /DX /Za /W1
#pragma warning(default:4067)
#if defined(X)
#else
#endif v   // C4067
int main()
{
}
```

To fix by using the following possible solutions

1. Compile with **/Ze**.
2. Use comment delimiters:

```
// C4067b.cpp
// compile with: /DX /Za /W1
#if defined(X)
#else
#endif
int main()
{
}
```

Compiler Warning (level 1) C4068

Error Message
unknown pragma

The compiler ignored an unrecognized pragma. Be sure the **pragma** is allowed by the compiler you are using. The following sample generates C4068:

```
// C4068.cpp
// compile with: /W1
#pragma NotAValidPragmaName   // C4068, use valid name to resolve
int main()
{
}
```

Compiler Warning (level 3) C4069

Error Message
long double is the same precision as double

Some platforms, such as the R4000 processor, consider **long double** to have the same precision as **double**.

Compiler Warning (level 3) C4073

Error Message
initializers put in library initialization area

Only third-party library developers should use the library initialization area, which is specified by #pragma init_seg. The following sample generates C4073:

```
// C4073.cpp
// compile with: /W3
#pragma init_seg(lib)   // C4073

// try this line to resolve the warning
// #pragma init_seg(user)

int main() {
}
```

Compiler Warning (level 1) C4074

Error Message
initializers put in compiler reserved initialization area

The compiler initialization area, which is specified by #pragma init_seg, is reserved by Microsoft. Code in this area may be executed before initialization of the C run-time library.

The following sample generates C4074:

```
// C4074.cpp
// compile with: /W1
#pragma init_seg( compiler )   // C4074

// try this line to resolve the warning
// #pragma init_seg(user)

int main() {
}
```

Compiler Warning (level 1) C4075

Error Message
initializers put in unrecognized initialization area

A #pragma init_seg uses an unrecognized section name. The compiler ignores the **pragma** command.

The following sample generates C4075:

```
// C4075.cpp
// compile with: /W1
#pragma init_seg("mysegg")   // C4075

// try..
// #pragma init_seg(user)

int main() {
}
```

Compiler Warning (level 1) C4076

Error Message
'typemod' : can not be used with type 'typename'

A type modifier, whether it is **signed** or **unsigned**, cannot be used with a noninteger type. ***typemod*** is ignored.

Example
The following sample generates C4076:

```
// C4076.cpp
// compile with: /W1 /LD
unsigned double x;   // C4076
```

Compiler Warning (level 1) C4077

Error Message
unknown check_stack option

The old form of the **check_stack** pragma is used with an unknown argument. The argument must be +, –, or empty.

The compiler ignores the pragma and leaves the stack checking unchanged.

Example

```
// C4077.cpp
// compile with: /W1 /LD
#pragma check_stack yes // C4077
#pragma check_stack +    // Correct old form
#pragma check_stack (on) // Correct new form
```

Compiler Warning (level 1) C4079

Error Message
unexpected token 'token'

An unexpected separator token occurs in a pragma argument list. The remainder of the pragma was ignored.

The following sample generates C4079:

```
// C4079.cpp
// compile with: /W1
#pragma warning(disable : 4081)
#pragma pack(c,16)   // C4079

int main() {
}
```

Compiler Warning (level 1) C4080

Error Message
expected identifier for segment name; found 'symbol'

The name of the segment in #pragma alloc_text must be a string or an identifier. The compiler ignores the pragma if a valid identifier is not found.

The following sample generates C4080:

```
// C4080.cpp
// compile with: /W1
extern "C" void func(void);

#pragma alloc_text()   // C4080

// try this line to resolve the warning
// #pragma alloc_text("mysection", func)

int main() {
}

void func(void) {
}
```

Compiler Warning (level 1) C4081

Error Message
expected 'token1'; found 'token2'

The compiler expected a different token in this context.

Example

```
// C4081.cpp
// compile with: /W1 /LD
#pragma optimize) "l", on )   // C4081
```

Compiler Warning (level 1) C4083

Error Message
expected 'token'; found identifier 'identifier'

An identifier occurs in the wrong place in a **#pragma** statement.

Example

```
// C4083.cpp
// compile with: /W1 /LD
#pragma warning disable:4083    // C4083
#pragma warning(disable:4083)   //correct
```

Check the syntax of the #pragma directives.

Compiler Warning (level 1) C4085

Error Message
expected pragma parameter to be 'on' or 'off'

The pragma requires an **on** or **off** parameter. The pragma is ignored.

The following sample generates C4085:

```
// C4085.cpp
// compile with: /W1 /LD
#pragma optimize( "t", maybe )  // C4085
```

Compiler Warning (level 1) C4086

Error Message
expected pragma parameter to be '1', '2', '4', '8', or '16'

The pragma parameter does not have the required value (1, 2, 4, 8, or 16).

Example

```
// C4086.cpp
// compile with: /W1 /LD
#pragma pack( 3 ) // C4086
```

Compiler Warning (level 1) C4087

Error Message
'function' : declared with 'void' parameter list

The function declaration has no formal parameters, but the function call has actual parameters. Extra parameters are passed according to the calling convention of the function.

This warning is for the C compiler.

Example

```
// C4087.c
// compile with: /W1
int f1( void ) {
}

int main() {
   f1( 10 );   // C4087
}
```

Compiler Warning (level 1) C4088

Error Message
'function' : pointer mismatch in actual parameter 'number', formal parameter 'number'

The corresponding formal and actual parameters have a different level of indirection. The actual parameter is passed without change. The called function interprets its value as a pointer.

Compiler Warning (level 1) C4089

Error Message
'function' : different types in actual parameter 'number', formal parameter 'number'

The corresponding formal and actual parameters have different types. The actual parameter is passed without change. The function casts the actual parameter to the type specified in the function definition.

Compiler Warning (level 1) C4090

Error Message
'operation' : different 'modifier' qualifiers

A variable used in an operation is defined with a specified modifier that prevents it from being modified without detection by the compiler. The expression is compiled without modification.

This warning can be caused when a pointer to a **const** or **volatile** item is assigned to a pointer not declared as pointing to **const** or **volatile**.

This warning is issued for C programs. In a C++ program, the compiler issues an error: C2440.

The following sample generates C4090:

```
// C4090.c
// compile with: /W1
int *volatile *p;
int *const *q;
int **r;

int main() {
   p = q;   // C4090
   p = r;
   q = p;   // C4090
   q = r;
   r = p;   // C4090
   r = q;   // C4090
}
```

Compiler Warning (level 1) C4091

Error Message
'keyword' : ignored on left of 'type' when no variable is declared

The compiler detected a situation where the user probably intended a variable to be declared, but the compiler was not able to declare the variable.

Example

A **__declspec** attribute at the beginning of a user-defined type declaration applies to the variable of that type. C4091 indicates no variable is declared. The following sample generates C4091.

```
// C4091.cpp
// compile with: /W1 /c
__declspec(dllimport) class X {}; // C4091

// __declspec attribute applies to varX
__declspec(dllimport) class X2 {} varX;

// __declspec attribute after the class or struct keyword
// applies to user defined type
class __declspec(dllimport) X3 {};
```

If an identifier is a typedef, it cannot also be a variable name. The following sample generates C4091.

```
// C4091_b.cpp
// compile with: /c /W1 /WX
#define LIST 4
typedef struct _LIST {} LIST;   // C4091
```

Compiler Warning (level 4) C4092

Error Message
sizeof returns 'unsigned long'

The operand of the **sizeof** operator was very large, so **sizeof** returned an unsigned **long**. This warning occurs under the Microsoft extensions (/Ze). Under ANSI compatibility (/Za), the result is truncated instead.

Compiler Warning (level 2) C4094

Error Message
untagged 'token' declared no symbols

The compiler detected an empty declaration using an untagged structure, union, or class. The declaration is ignored.

Example

```
// C4094.cpp
// compile with: /W2
struct
{
};   // C4094

int main()
{
}
```

This condition generates an error under ANSI compatibility (/Za).

Compiler Warning (level 1) C4096

Error Message
'a': interface is not a COM interface; will not be emitted to IDL

An interface definition that you may have intended as a COM interface was not defined as a COM interface and therefore will not be emitted to the IDL file.

See Interface Attributes for a list attributes that indicate an interface is a COM interface.

The following sample generates C4096:

```
// C4096.cpp
// compile with: /W1 /LD
#include "windows.h"
[module(name="xx")];

// [object, uuid("00000000-0000-0000-0000-000000000001")]
__interface a
{
};

[coclass, uuid("00000000-0000-0000-0000-000000000002")]
struct b : a
{
};   // C4096, remove coclass or uncomment object
```

Compiler Warning (level 1) C4097

Error Message
expected pragma parameter to be 'restore' or 'off'

A pragma was passed an invalid value.

The following sample generates C4097:

```
// C4097.cpp
// compile with: /W1
#pragma runtime_checks("",test)   // C4097
// try the following line instead
// #pragma runtime_checks("",off)

int main() {
}
```

Compiler Warning (level 1) C4098

Error Message
'function' : void function returning a value

A function declared with return type void has a **return** statement that returns a value. The compiler assumes the function returns a value of type **int**.

Compiler Warning (level 2) C4099

Error Message
'identifier' : type name first seen using 'objecttype1' now seen using 'objecttype2'

An object declared as a structure is defined as a class or an object declares as a class is defined as a structure. The compiler uses the type given in the definition.

Example
The following sample generates C4099.

```
// C4099.cpp
// compile with: /W2 /c
struct A;
class A {};   // C4099, use different identifer or use same object type
```

Compiler Warnings C4100 Through C4192

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, type the error number in the **Look for** box in the Index, or click a link below.

Compiler Warning (level 4) C4100

Compiler Warning (level 3) C4101

Compiler Warning (level 3) C4102

Compiler Warning (level 1) C4103

Compiler Warning (level 1) C4109

Compiler Warning (levels 1 and 4) C4112

Compiler Warning (level 1) C4113

Compiler Warning (level 1) C4114

Compiler Warning (levels 1 and 4) C4115

Compiler Warning (level 1) C4116

Compiler Warning (level 1) C4117

Compiler Warning (level 1) C4119

Compiler Warning (level 1) C4120

Compiler Warning (level 4) C4121

Compiler Warning (level 1) C4122

Compiler Warning (level 1) C4124

Compiler Warning (level 4) C4125

Compiler Warning (level 4) C4127

Compiler Warning (level 1) C4129

Compiler Warning (level 4) C4130

Compiler Warning (level 4) C4131

Compiler Warning (level 4) C4132

Compiler Warning (level 3) C4133

Compiler Warning (level 1) C4137

Compiler Warning (level 1) C4138

Compiler Warning (level 1) C4141

Compiler Warning (level 1) C4142

Compiler Warning (level 1) C4143

Compiler Warning (level 1) C4144

Compiler Warning (level 1) C4145

Compiler Warning (level 2) C4146

Compiler Warning (level 2) C4150

Compiler Warning (level 4) C4152

Compiler Warning (level 1) C4153

Compiler Warning (level 1) C4154

Compiler Warning (level 1) C4155

Compiler Warning (level 2) C4156

Compiler Warning (level 1) C4157

Compiler Warning (level 1) C4158

Compiler Warning (level 3) C4159

Compiler Warning (level 1) C4160

Compiler Warning (level 3) C4161

Compiler Warning (level 1) C4162

Compiler Warning (level 1) C4163

Compiler Warning (level 1) C4164

Compiler Warning (level 1) C4165

Compiler Warning (level 1) C4166

Compiler Warning (level 1) C4167

Compiler Warning (level 1) C4168

Compiler Warning (level 1) C4172

Compiler Warning (level 1) C4174

Compiler Warning (level 1) C4175

Compiler Warning (level 1) C4176

Compiler Warning (level 1) C4177

Compiler Warning (level 1) C4178

Compiler Warning (level 1) C4179

Compiler Warning (level 1) C4180

Compiler Warning (level 1) C4181

Compiler Warning (level 1) C4182

Compiler Warning (level 1) C4183

Compiler Warning (level 1) C4185

Compiler Warning (level 1) C4186

Compiler Warning (level 1) C4187

Compiler Warning (level 4) C4189

Compiler Warning (level 1) C4190

Compiler Warning (level 3) C4191

Compiler Warning (level 3) C4192

Compiler Warning (level 3) C4197

Compiler Warning (level 4) C4100

Error Message
'identifier' : unreferenced formal parameter

The formal parameter is not referenced in the body of the function. The unreferenced parameter is ignored.

C4100 can als be issued when code calls a destructor on a otherwise unreferenced parameter of primitive type. This is a limitation of the Visual C++ compiler.

The following sample generates C4100:

```
// C4100.cpp
// compile with: /W4
void func(int i) {   // C4100, delete the unreferenced parameter to
                     //resolve the warning
   // i;   // or, add a reference like this
}

int main()
{
   func(1);
}
```

Compiler Warning (level 3) C4101

Error Message
'identifier' : unreferenced local variable

The local variable is never used. This warning will occur in the obvious situation:

```
// C4101a.cpp
// compile with: /W3
int main() {
int i;   // C4101
}
```

However, this warning will also occur when calling a **static** member function through an instance of the class:

```
// C4101b.cpp
// compile with:  /W3
struct S {
   static int func()
   {
      return 1;
   }
};

int main() {
   S si;   // C4101, si is never used
   int y = si.func();
   return y;
}
```

In this situation, the compiler uses information about `si` to access the **static** function, but the instance of the class is not needed to call the **static** function; hence the warning. To resolve this warning, you could:

- Add a constructor, in which the compiler would use the instance of `si` in the call to `func`.
- Remove the **static** keyword from the definition of `func`.
- Call the **static** function explicitly: `int y = S::func();`.

Compiler Warning (level 3) C4102

Error Message
'label' : unreferenced label

The label is defined but never referenced. The compiler ignores the label.

The following sample generates C4102:

```
// C4102.cpp
// compile with: /W3
int main() {
   int a;

   test:   // C4102, remove the unreferenced label to resolve

   a = 1;
}
```

Compiler Warning (level 1) C4103

Error Message
'filename' : alignment changed after including header, may be due to missing #pragma pack(pop)

Packing affects the layout of classes, and commonly, if packing changes across header files, there can be problems.

Use #pragma pack(pop) before exiting the header file to resolve this warning.

The following sample generates C4103:

```
// C4103.h
#pragma pack(push, 4)

// defintions and declarations

// uncomment the following line to resolve
// #pragma pack(pop)
```

And then,

```
// C4103.cpp
// compile with: /LD /W1
#include "c4103.h"   // C4103
```

Compiler Warning (level 1) C4109

Error Message
unexpected identifier 'identifier'

The pragma containing the unexpected identifier is ignored.

Example

```
// C4109.cpp
// compile with: /W1 /LD
#pragma init_seg( abc ) // C4109
```

Compiler Warning (levels 1 and 4) C4112

Error Message
#line requires an integer between 1 and number

The #line directive specifies an integer outside the allowable range.

If the specified number is less than 1, the line counter is reset to 1. If the specified number is greater than ***number***, the line counter is unchanged. This is a level 1 warning under ANSI compatibility (/Za) and a level 4 warning with Microsoft extensions (/Ze).

The following sample generates C4112:

```
// C4112.cpp
// compile with: /W4
#line 0   // C4112, value must be between 1 and number

int main() {
}
```

Compiler Warning (level 1) C4113

Error Message
'identifier1' differs in parameter lists from 'identifier2'

A function pointer is assigned to another function pointer, but the formal parameter lists of the functions do not agree. The assignment is compiled without modification.

Compiler Warning (level 1) C4114

Error Message
same type qualifier used more than once

A type declaration or definition uses a type qualifier (**const**, **volatile**, **signed**, or **unsigned**) more than once. This causes a warning with Microsoft extensions (/Ze) and an error under ANSI compatibility (/Za).

The following sample generates C4114:

```
// C4114.cpp
// compile with: /W1 /c
volatile volatile int i;   // C4114
```

The following sample generates C4114:

```
// C4114_b.cpp
// compile with: /W1 /c
static const int const * ii;   // C4114
static const int * const iii;   // OK
```

Compiler Warning (levels 1 and 4) C4115

Error Message
'type' : named type definition in parentheses

The given symbol is used to define a structure, union, or enumerated type inside a parenthetical expression. The scope of the definition may be unexpected.

In a C function call, the definition has global scope. In a C++ call, the definition has the same scope as the function being called.

This warning can also be caused by declarators within parentheses (such as prototypes) that are not parenthetical expressions.

This is a level-1 warning with C++ programs and C programs compiled under ANSI compatibility (/Za). Otherwise, it is level 3.

Compiler Warning (level 1) C4116

Error Message
unnamed type definition in parentheses

A structure, union, or enumerated type with no name is defined in a parenthetical expression. The type definition is meaningless.

In a C function call, the definition has global scope. In a C++ function call, the definition has the same scope as the function being called.

Compiler Warning (level 1) C4117

Error Message
macro name 'name' is reserved; 'Command' ignored

To fix by checking the following possible causes

1. Trying to define or undefine a predefined macro.
2. Trying to define or undefine the preprocessor operator **defined**.

The following sample generates C4117:

```
// C4117.cpp
// compile with: /W1
#define __FILE__ test         // C4117. __FILE__ is a predefined macro
#define ValidMacroName test   // ok

int main() {
}
```

Compiler Warning (level 1) C4119

Error Message
different bases 'base1' and 'base2' specified

Two based pointers are incompatible because they have different bases. The compiler cannot convert between them.

Compiler Warning (level 1) C4120

Error Message
based/unbased mismatch

The compiler cannot convert between the two pointers because one is based and the other is not.

Compiler Warning (level 4) C4121

Error Message
'symbol' : alignment of a member was sensitive to packing

A structure member is aligned on a memory offset whose value is not a multiple of the member's size. For example, the following code snippet will produce this warning:

```
// C4121.cpp
// compile with: /W4 /c
#pragma pack(2) // C4121
struct s
{
   char a;
   int b;
};
```

You could make one of the following changes to prevent this warning:

- Change `pack(2)` to `pack(4)`.
- Reverse the order of the structure members such that the `int` precedes the `char`.

When data is not aligned on boundaries that are multiples of the data's size performance can degrade and if you port your code to a RISC machine it will not compile.

You can specify the structure alignment with #pragma pack or /Zp. Note that the compiler does not generate this warning when /Zp1 is specified.

Compiler Warning (level 1) C4122

Error Message
'function' : alloc_text applicable only to functions with C linkage

The **alloc_text** pragma applies only to functions declared with **extern c**. It cannot be used with external C++ functions.

The pragma is ignored.

Compiler Warning (level 1) C4124

Error Message
__fastcall with stack checking is inefficient

The **__fastcall** keyword was used with stack checking enabled.

The **__fastcall** convention generates faster code, but stack checking causes slower code. When using **__fastcall**, turn off stack checking with the **check_stack** pragma or /Gs.

This warning is issued only for the first function declared under these conditions.

Compiler Warning (level 4) C4125

Error Message
decimal digit terminates octal escape sequence

The compiler evaluates the octal number without the decimal digit and assumes the decimal digit is a character.

Example

```
// C4125a.cpp
// compile with: /W4
char array1[] = "\709"; // C4125
int main()
{
}
```

If the digit 9 is intended as a character, correct the example as follows:

```
// C4125b.cpp
// compile with: /W4
char array[] = "\0709";  // C4125 String containing "89"
int main()
{
}
```

Compiler Warning (level 4) C4127

Error Message
conditional expression is constant

The controlling expression of an **if** statement or **while** loop evaluates to a constant. If the controlling expression of a **while** loop is a constant because the loop will exit in the middle, consider replacing the **while** loop with a **for** loop. You can omit the initialization, termination test and loop increment of a **for** loop, which causes the loop to be infinite (like `while(1)`) and you can exit the loop from the body of the **for** statement.

The following sample generates C4127:

```
// C4127.cpp
// compile with: /W4
#include <stdio.h>
int main() {
   if (1 == 1) {}   // C4127
   while (1) { break; }   // C4127

   // OK
   for ( ; ; ) {
      printf("test\n");
      break;
   }
}
```

Compiler Warning (level 1) C4129

Error Message
'character' : unrecognized character escape sequence

The *character* following a backslash (\) in a character or string constant is not recognized as a valid escape sequence. The backslash is ignored and not printed. The character following the backslash is printed.

To print a single backslash, specify a double backslash (\\).

The C++ standard, in section 2.13.2 discusses escape sequences.

The following sample generates C4129:

```
// C4129.cpp
// compile with: /W1
int main() {
   char array1[] = "\/709";   // C4129
   char array2[] = "\n709";   // OK
}
```

Compiler Warning (level 4) C4130

Error Message
'operator' : logical operation on address of string constant

Using the operator with the address of a string literal produces unexpected code.

The following sample generates C4130:

```
// C4130.cpp
// compile with: /W4
int main()
{
   char *pc;
   pc = "Hello";
   if (pc == "Hello") // C4130
   {
   }
}
```

The **if** statement compares the value stored in the pointer `pc` to the address of the string "Hello", which is allocated separately each time the string occurs in code. The **if** statement does not compare the string pointed to by `pc` with the string "Hello".

To compare strings, use the **strcmp** function.

Compiler Warning (level 4) C4131

Error Message
'function' : uses old-style declarator

The specified function declaration is not in prototype form.

Old-style function declarations should be converted to prototype form.

The following example shows an old-style function declaration:

```
// C4131.c
// compile with: /W4 /c
void addrec( name, id ) // C4131 expected
char *name;
int id;
{ }
```

The following example shows a prototype form:

```
 void addrec( char *name, int id )
 { }
```

Compiler Warning (level 4) C4132

Error Message
'object' : const object should be initialized

The constant is not initialized. The value of a symbol with the **const** attribute cannot be changed after its definition, it should be given an initialization value.

Compiler Warning (level 3) C4133

Error Message
'type' : incompatible types - from 'type1' to 'type2'

This warning can be caused by trying to subtract two pointers of different types.

To avoid this warning, provide an appropriate type cast.

Compiler Warning (level 1) C4137

Error Message
'function' : no return value from floating-point function

The function has no **return** statement. A **long double** function returns its value on the floating-point stack or emulation stack. If the function does not return a value, a run-time floating-point stack underflow error can occur. Make sure all floating-point functions return a floating-point value.

Compiler Warning (level 1) C4138

Error Message
'*/' found outside of comment

The closing-comment delimiter is not preceded by an opening-comment delimiter. The compiler assumes a space between the asterisk (*****) and the forward slash (/).

Example

```
// C4138a.cpp
// compile with: /W1
int */*comment*/ptr;   // C4138 Ambiguous first delimiter causes warning
int main()
{
}
```

This warning can be caused by trying to nest comments.

This warning may be resolved if you comment out sections of code that contain comments, enclose the code in an **#if/#endif** block, and set the controlling expression to zero:

```
// C4138b.cpp
// compile with: /W1
#if 0
int my_variable;   /* Declaration currently not needed */
#endif
int main()
{
}
```

Compiler Warning (level 1) C4141

Error Message

'modifier' : used more than once

Example

```
// C4141.cpp
// compile with: /W1 /LD
inline inline void func ();   // C4141
```

Compiler Warning (level 1) C4142

Error Message
benign redefinition of type

A type is redefined in a manner that has no effect on the generated code.

To fix by checking the following possible causes:

- A member function of a derived class has a different return type from the corresponding member function of the base class.
- A type defined with the **typedef** command is redefined using different syntax.

The following sample generates C4142:

```
// C4142.c
// compile with: /W1
float X2;
X2 = 2.0 + 1.0;   // C4142

int main() {
   float X2;
   X2 = 2.0 + 1.0;   // OK
}
```

Compiler Warning (level 1) C4143

Error Message
pragma 'same_seg' not supported; use __based allocation

The **#pragma same_seg** is no longer supported. Use the __based keyword instead.

Compiler Warning (level 1) C4144

Error Message
'expression' : relational expression as switch expression

The specified relational expression was used as the control expression of a switch statement. The associated case statements will be offered Boolean values. The following sample generates C4144:

```
// C4144.cpp
// compile with: /W1
int main()
{
   int i = 0;
   switch(!i) {   // C4144, remove the ! to resolve
      case 1:
         break;
      default:
         break;
   }
}
```

Compiler Warning (level 1) C4145

Error Message
'expression1' : relational expression as switch expression; possible confusion with 'expression2'

A **switch** statement uses a relational expression as its control expression, which results in a Boolean value for the **case** statements. Did you mean *expression2*?

Example
The following sample generates C4145:

```
// C4145.cpp
// compile with: /W1
int main() {
   int i = 0;
   switch(i == 1) {   // C4145, use i instead of i == 1 to resolve
      case 1:
         break;
      default:
         break;
   }
}
```

Compiler Warning (level 2) C4146

Error Message
unary minus operator applied to unsigned type, result still unsigned

Unsigned types can hold only non-negative values, so unary minus (negation) does not usually make sense when applied to an unsigned type. Both the operand and the result are non-negative.

Practically, this occurs when the programmer is trying to express the minimum integer value, which is -2147483648. This value cannot be written as -2147483648 because the expression is processed in two stages:

1. The number 2147483648 is evaluated. Because it is greater than the maximum integer value of 2147483647, the type of 2147483648 is not int, but **unsigned int**.
2. Unary minus is applied to the value, with an unsigned result, which also happens to be 2147483648.

The unsigned type of the result can cause unexpected behavior. If the result is used in a comparison, then an unsigned comparison might be used, for example, when the other operand is an **int**. This explains why the example program below prints just one line.

The expected second line, `1 is greater than the most negative int`, is not printed because `((unsigned int)1) > 2147483648` is false.

You can avoid C4146 by using INT_MIN from limits.h, which has the type **signed int**.

Example
The following sample generates C4146:

```
// C4146.cpp
// compile with: /W2
#include <stdio.h>

void check(int i)
{
    if (i > -2147483648)   // C4146
        printf_s("%d is greater than the most negative int\n", i);
}

int main()
{
    check(-100);
    check(1);
}
```

Compiler Warning (level 2) C4150

Error Message
deletion of pointer to incomplete type 'type'; no destructor called

The **delete** operator is called to delete a type that was declared but not defined, so the compiler cannot find a destructor.

The following sample generates C4150:

```
// C4150.cpp
// compile with: /W2
class  IncClass;

void NoDestruct( IncClass* pIncClass )
{
   delete pIncClass;
} // C4150, define class to resolve

int main()
{
}
```

Compiler Warning (level 4) C4152

Error Message
non standard extension, function/data ptr conversion in expression

A function pointer is converted to or from a data pointer. This conversion is allowed under Microsoft extensions (/Ze) but not under ANSI C.

Compiler Warning (level 1) C4153

Error Message
function/data pointer conversion in expression

A function pointer is converted to or from a data pointer. This conversion is allowed under Microsoft extensions (/Ze) but not under ANSI C.

Compiler Warning (level 1) C4154

Error Message
deletion of an array expression; conversion to pointer supplied

You cannot use **delete** on an array, so the compiler converts the array to a pointer.

Example

```
// C4154.cpp
// compile with: /W1
int main() {
   int array[ 10 ];
   delete array;   // C4154 can't delete stack object

   int *parray2 = new int [10];
   int (&array2)[10] = (int(&)[10]) parray2;
   delete [] array2;   // C4154

   // try the following line instead
   delete [] &array2;
}
```

Compiler Warning (level 1) C4155

Error Message
deletion of an array expression without using the array form of 'delete'

The array form of **delete** should be used to delete an array. This warning occurs only under ANSI-compatibility (/Za).

Example
The following sample generates C4155:

```
// C4155.cpp
// compile with: /Za /W1
#include <stdio.h>

int main(void)
{
    int (*array)[ 10 ] = new int[ 5 ] [ 10 ];
    array[0][0] = 8;

    printf_s("%d\n", array[0][0]);

   delete array;   // C4155
    // try the following line instead
    // delete [] array;   // C4155
}
```

Compiler Warning (level 2) C4156

Error Message
deletion of an array expression without using the array form of 'delete'; array form substituted

The non-array form of **delete** cannot delete an array. The compiler translated **delete** to the array form.

This warning occurs only under Microsoft extensions (/Ze).

Example

```
// C4156.cpp
// compile with: /W2
int main()
{
   int (*array)[ 10 ] = new int[ 5 ][ 10 ];
   delete array; // C4156, changed by compiler to "delete [] array;"
}
```

Compiler Warning (level 1) C4157

Error Message
pragma was ignored by C compiler

Only the C++ compiler recognizes **init_seg()**.

Compiler Warning (level 1) C4158

Error Message
assuming #pragma pointers_to_members(full_generality, inheritance)

A **#pragma pointers_to_members(***single*|*multiple*|*virtual***)** was issued without an accompanying **#pragma pointers_to_members(full_generality)**.

Compiler Warning (level 3) C4159

Error Message
#pragma pragma(pop,...) : has popped previously pushed identifier 'identifier'

Your source code contains a **push** instruction with an identifier for a pragma followed by a **pop** instruction without an identifier. As a result, ***identifier*** is popped, and subsequent uses of ***identifier*** may cause unexpected behavior.

To avoid this warning, give an identifier in the **pop** instruction. For example:

```
// C4159.cpp
// compile with: /W3
#pragma pack(push, f)
#pragma pack(pop)   // C4159

// using the identifier resolves the warning
// #pragma pack(pop, f)

int main()
{
}
```

Compiler Warning (level 1) C4160

#pragma *pragma* (pop,...) : did not find previously pushed identifier ' *identifier* '

A pragma statement in your source code tries to pop an identifier that has not been pushed. To avoid this warning, be sure that the identifier being popped has been properly pushed.

The following example generates C4160:

```
// C4160.cpp
// compile with: /W1
#pragma pack(push)

#pragma pack(pop, id)   // C4160
// use identifier when pushing to resolve the warning
// #pragma pack(push, id)

int main() {
}
```

Compiler Warning (level 3) C4161

Error Message
#pragma pragma(pop...) : more pops than pushes

Because your source code contains one more pop than pushes for pragma ***pragma***, the stack may not behave as you expect. To avoid the warning, be sure that the number of pops does not exceed the number of pushes.

Example
The following example generates C4161:

```
// C4161.cpp
// compile with: /W3 /LD
#pragma pack(push, id)
#pragma pack(pop, id)
#pragma pack(pop, id)   // C4161, an extra pop

#pragma bss_seg(".my_data1")
int j;

#pragma bss_seg(push, stack1, ".my_data2")
int l;

#pragma bss_seg(pop, stack1)
int m;

#pragma bss_seg(pop, stack1)   // C4161
```

Compiler Warning (level 1) C4162

Error Message
'identifier' : no function with C linkage found

A function with C linkage is declared but cannot be found.

To resolve this warning, compile in a .c file (invoke the C compiler). If you must invoke the C++ compiler, place extern "C" before the function declaration.

The following sample generates C4162

```
// C4162.cpp
// compile with: /c /W1
unsigned char _bittest(long* a, long b);
#pragma intrinsic (_bittest)   // C4162

int main() {
   bool bit;
   long num = 78002;
   bit = _bittest(&num, 5);
}
```

Possible resolution:

```
// C4162b.cpp
// compile with: /c
extern "C"
unsigned char _bittest(long* a, long b);
#pragma intrinsic (_bittest)

int main() {
   bool bit;
   long num = 78002;
   bit = _bittest(&num, 5);
}
```

Compiler Warning (level 1) C4163

Error Message
'identifier' : not available as an intrinsic function

The specified function cannot be used as an intrinsic function. The compiler ignores the invalid function name.

The following sample generates C4163:

```
// C4163.cpp
// compile with: /W1 /LD
#include <math.h>
#pragma intrinsic(mysin)   // C4163
// try the following line instead
// #pragma intrinsic(sin)
```

Compiler Warning (level 1) C4164

Error Message
'identifier' : intrinsic function not declared

The specified intrinsic function is not declared; you may need to #include the appropriate header file.

Compiler Warning (level 1) C4165

Error Message
'HRESULT' is being converted to 'bool'; are you sure this is what you want?

When using an HRESULT in an if statement, the HRESULT will be converted to a bool unless you explicitly test for the variable as an HRESULT. This warning is off by default.

The following sample generates C4165

```
// C4165.cpp
// compile with: /W1
#include <windows.h>
#pragma warning(1:4165)

extern HRESULT hr;
int main() {
   if (hr) {
   // try either of the following ...
   // if (FAILED(hr)) { // C4165 expected
   // if (hr != S_OK) {
   }
}
```

Compiler Warning (level 1) C4166

illegal calling convention for constructor/destructor

Constructors and destructors cannot have calling conventions other than the default for the platform (except when you explicitly specify **__clrcall**).

Compiler Warning (level 1) C4167

Error Message
function : only available as an intrinsic function

The **#pragma function** tries to force the compiler to use a conventional call to a function that must be used in intrinsic form. The pragma is ignored.

To avoid this warning, remove the **#pragma function**.

Example

```
// C4167.cpp
// compile with: /W1
#include <malloc.h>
#pragma function(_alloca )   // C4167: _alloca() is intrinsic only
int main(){}
```

Compiler Warning (level 1) C4168

Error Message

compiler limit : out of debugger types, delete program database 'database' and rebuild

The program database file must be rebuilt to accommodate all types in the program.

Compiler Warning (level 1) C4172

Error Message
returning address of local variable or temporary

A function returns the address of a local variable or temporary object. Local variables and temporary objects are destroyed when a function returns, so the address returned is not valid.

Redesign the function so that it does not return the address of a local object.

The following sample generates C4172:

```
// C4172.cpp
// compile with: /W1 /LD
float f = 10;

const double& bar() {
// try the following line instead
// const float& bar() {
   return f;   // C4172
}
```

Compiler Warning (level 1) C4174

Error Message

'name' : not available as a #pragma component

Example

```
// C4174.cpp
// compile with: /W1
#pragma component(info)  // C4174; unknown
#pragma component(browser, off)  // turn off browse info
int main()
{
}
```

Compiler Warning (level 1) C4175

Error Message

#pragma component(browser, on) : browser info must initially be specified on the command line

To use component pragma, you must generate browse information during compilation (/FR).

Compiler Warning (level 1) C4176

Error Message
'subcomponent' : unknown subcomponent for #pragma component browser

The **component** pragma contains an invalid subcomponent. To exclude references to a particular name, you must use the **references** option before the name.

Example

```
// C4176.cpp
// compile with: /W1 /LD
#pragma component(browser, off, i)  // C4176
#pragma component(browser, off, references, i) // ok
```

Compiler Warning (level 1) C4177

Error Message
#pragma pragma should be at global scope

The pragma pragma should not be used within a local scope. The **pragma** will not be valid until global scope is encountered after the current scope.

The following sample generates C4177:

```
// C4177.cpp
// compile with: /W1
// #pragma bss_seg("global")   // OK

int main() {
   #pragma bss_seg("local")    // C4177
}
```

Compiler Warning (level 1) C4178

Error Message
case constant 'constant' too big for the type of the switch expression

A case constant in a **switch** expression does not fit in the type to which it is assigned.

Example

```
// C4178.cpp
// compile with: /W1
int main()
{
    int i;  // maximum size of unsigned long int is 4294967295
    switch( i )
    {
        case 4294967295:   // OK
            break;
        case 4294967296:   // C4178
            break;
    }
 }
```

Compiler Warning (level 1) C4179

Error Message
'//*' : parsed as '/' and '/*': confusion with standard '//' comments

//* is an incorrect comment delimiter. Use **//** or **/*** instead.

Compiler Warning (level 1) C4180

Error Message
qualifier applied to function type has no meaning; ignored

A qualifier, such as **const**, is applied to a function type defined by **typedef**.

Example

```
// C4180.cpp
// compile with: /W1 /c
typedef int *FuncType(void);

// the const qualifier cannot be applied to the
// function type FuncType
const FuncType f;   // C4180
```

Compiler Warning (level 1) C4181

Error Message
qualifier applied to reference type; ignored

A qualifier, such as **const**, is applied to a reference type defined by **typedef**.

Example

```
// C4181.cpp
// compile with: /W1 /LD
typedef int &RefInt;
int i;
const RefInt cri = i;   // C4181
```

Compiler Warning (level 1) C4182

Error Message
#include nesting level is 'number' deep; possible infinite recursion

The compiler ran out of space on the heap because of the number of nested include files. An include file is nested when it is included from another include file.

This message is informational and precedes error C1076.

Compiler Warning (level 1) C4183

Error Message
'identifier': missing return type; assumed to be a member function returning 'int'

The inline definition of a member function in a class or a structure does not have a return type. This member function is assumed to have a default return type of **int**.

The following sample generates C4183:

```
// C4183.cpp
// compile with: /W1 /c
#pragma warning(disable : 4430)
class MyClass1;
class MyClass2 {
   MyClass1() {};   // C4183
};
```

Compiler Warning (level 1) C4185

Error Message
ignoring unknown #import attribute 'attribute'

The attribute is not a valid attribute of **#import**. It is ignored.

Example

```
// C4185.cpp
// compile with: /W1 /c
#import "stdole2.tlb" no_such_attribute   // C4185
```

Compiler Warning (level 1) C4186

Error Message
#import attribute 'attribute' requires count arguments; ignored

A **#import** attribute has the wrong number of arguments.

Example

```
// C4186.cpp
// compile with: /W1 /c
#import "stdole2.tlb" no_namespace("abc") rename("a","b","c")  // C4186
```

The **no_namespace** attribute takes no arguments. The **rename** attribute takes only two arguments.

Compiler Warning (level 1) C4187

Error Message
#import attributes 'attribute1' and 'attribute2' are incompatible; both ignored

A #import statement specified **no_implementation** and **implementation_only** attributes. Both are ignored.

Compiler Warning (level 4) C4189

Error Message
'identifier' : local variable is initialized but not referenced

A variable is declared and initialized but not used.

The following sample generates C4189:

```
// C4189.cpp
// compile with: /W4
int main() {
   int a = 1;   // C4189, remove declaration to resolve
}
```

Compiler Warning (level 1) C4190

Error Message
'identifier1' has C-linkage specified, but returns UDT 'identifier2' which is incompatible with C

A function or pointer to function has a UDT (user-defined type, which is a class, structure, enum, union, or __value type) as return type and **extern** "C" linkage. This is legal if:

- All calls to this function occur from C++.
- The definition of the function is in C++.

Example

```
// C4190.cpp
// compile with: /W1 /LD
struct X
{
   int i;
   X ();
   virtual ~X ();
};

extern "C" X func ();   // C4190
```

Compiler Warning (level 3) C4191

Error Message
'operator/operation' : unsafe conversion from 'type of expression' to 'type required'

Several operations involving function pointers are considered unsafe:

- Function types with different calling conventions.
- Function types with different return conventions.
- Argument or return types with different sizes, type categories, or classifications.
- Differing argument list lengths (on **__cdecl**, only on cast from longer list to shorter list, even if shorter is varargs).
- Pointer to data (other than **void***) aliased against a pointer to function.
- Any other type difference that would yield an error or warning on a **reinterpret_cast**.

Calling this function through the result pointer might cause your program to crash.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4191:

```
// C4191.cpp
// compile with: /W3 /clr
#pragma warning(default: 4191)

void __clrcall f1() { }
void __cdecl   f2() { }

typedef void (__clrcall * fnptr1)();
typedef void (__cdecl   * fnptr2)();

int main() {
   fnptr1 fp1 = static_cast<fnptr1>(&f1);
   fnptr2 fp2 = (fnptr2) &f2;

   fnptr1 fp3 = (fnptr1) &f2;   // C4191
   fnptr2 fp4 = (fnptr2) &f1;   // C4191
};
```

Compiler Warning (level 3) C4192

Error Message
automatically excluding 'name' while importing type library 'library'

A **#import** library contains an item, *name*, that is also defined in the Win32 system headers. Due to limitations of type libraries, names such as **IUnknown** or GUID are often defined in a type library, duplicating the definition from the system headers. **#import** will detect these items and refuse to incorporate them in the .tlh and .tli header files.

To override this behavior, use **#import** attributes no_auto_exclude and include().

Compiler Warning (level 3) C4197

Error Message
'type' : top-level volatile in cast is ignored

The compiler detected a cast to an r-value type which is qualified with volatile, or a cast of an r-value type to some type that is qualified with volatile. According to the C standard (6.5.3), properties associated with qualified types are meaningful only for l-value expressions.

The following sample generates C4197:

```
// C4197.cpp
// compile with: /W3
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

void sigproc(int);
struct S
{
   int i;
} s;

int main()
{
   signal(SIGINT, sigproc);
   s.i = 1;
   S *pS = &s;
   for ( ; (volatile int)pS->i ; )   // C4197
      break;
   // for ( ; *(volatile int *)&pS->i ; )   // OK
   //    break;
}

void sigproc(int) // ctrl-C
{
   signal(SIGINT, sigproc);
   s.i = 0;
}
```

Compiler Warnings C4200 Through C4584

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, type the error number in the **Look for** box in the Index, or click a link below.

Compiler Warning (levels 2 and 4) C4200

Compiler Warning (level 4) C4201

Compiler Warning (level 4) C4202

Compiler Warning (level 4) C4203

Compiler Warning (level 4) C4204

Compiler Warning (level 4) C4205

Compiler Warning (level 4) C4206

Compiler Warning (level 4) C4207

Compiler Warning (level 4) C4208

Compiler Warning (level 4) C4210

Compiler Warning (level 4) C4211

Compiler Warning (level 4) C4212

Compiler Warning (level 4) C4213

Compiler Warning (level 4) C4214

Compiler Warning (level 1) C4215

Compiler Warning (level 1) C4216

Compiler Warning (level 1) C4218

Compiler Warning (level 4) C4220

Compiler Warning (level 4) C4221

Compiler Warning (levels 1 and 4) C4223

Compiler Warning (level 1) C4224

Compiler Warning (level 1) C4226

Compiler Warning (level 1) C4227

Compiler Warning (level 1) C4228

Compiler Warning (level 1) C4229

Compiler Warning (level 1) C4230

Compiler Warning (level 3) C4231

Compiler Warning (level 4) C4232

Compiler Warning (level 4) C4233

Compiler Warning (level 4) C4234

Compiler Warning (level 4) C4235

Compiler Warning (level 1) C4237

Compiler Warning (level 4) C4238

Compiler Warning (level 4) C4239

Compiler Warning (level 3) C4240

Compiler Warning (level 4) C4242

Compiler Warning (level 3) C4243

Compiler Warning (level 2) C4244

Compiler Warning (levels 3 and 4) C4244

Compiler Warning (level 4) C4245

Compiler Warning (level 2) C4250

Compiler Warning (level 1) C4251

Compiler Warning (level 4) C4254

Compiler Warning (level 4) C4255

Compiler Warning (level 4) C4256

Compiler Warning (level 1) C4258

Compiler Warning (level 4) C4263

Compiler Warning (level 1) C4264

Compiler Warning (level 3) C4265

Compiler Warning (level 4) C4266

Compiler Warning (level 3) C4267

Compiler Warning (level 4) C4268

Compiler Warning (level 1) C4269

Compiler Warning (level 1) C4272

Compiler Warning (level 1) C4273

Compiler Warning (level 2) C4275

Compiler Warning (level 1) C4276

Compiler Warning (level 3) C4278

Compiler Warning (level 3) C4280

Compiler Warning (level 3) C4281

Compiler Warning (level 3) C4282

Compiler Warning (level 3) C4283

Compiler Warning (level 2) C4285

Compiler Warning (level 1) C4286

Compiler Warning (level 3) C4287

Compiler Warning (level 1) C4288

Compiler Warning (level 4) C4289

Compiler Warning (level 3) C4290

Compiler Warning (level 1) C4291

Compiler Warning (level 1) C4293

Compiler Warning (level 4) C4295

Compiler Warning (level 4) C4296

Compiler Warning (level 1) C4297

Compiler Warning (level 1) C4301

Compiler Warning (level 2) C4302

Compiler Warning (level 1) C4303
Compiler Warning (level 1) C4305
Compiler Warning (level 3) C4306
Compiler Warning (level 2) C4307
Compiler Warning (level 2) C4308
Compiler Warning (level 2) C4309
Compiler Warning (level 3) C4310
Compiler Warning (level 1) C4311
Compiler Warning (level 1) C4312
Compiler Warning (level 1) C4313
Compiler Warning (level 1) C4318
Compiler Warning (level 1) C4319
Compiler Warning (level 4) C4324
Compiler Warning (level 1) C4325
Compiler Warning (level 1) C4326
Compiler Warning (level 1) C4329
Compiler Warning (level 1) C4333
Compiler Warning (level 3) C4334
Compiler Warning C4335
Compiler Warning (level 4) C4336
Compiler Warning (level 4) C4337
Compiler Warning (level 4) C4339
Compiler Warning (level 1) C4340
Compiler Warning (level 3) C4341
Compiler Warning (level 1) C4342
Compiler Warning (level 4) C4343
Compiler Warning (level 1) C4344
Compiler Warning (level 2) C4345
Compiler Warning (level 1) C4346
Compiler Warning (level 4) C4347
Compiler Warning (level 1) C4348
Compiler Warning (level 1) C4350
Compiler Warning (level 1) C4351
Compiler Warning (level 1) C4353
Compiler Warning (levels 1 and 4) C4355
Compiler Warning (level 2) C4356
Compiler Warning (level 3) C4357
Compiler Warning (level 1) C4358
Compiler Warning (level 3) C4359
Compiler Warning (level 1) C4364

Compiler Warning (level 4) C4365

Compiler Warning (level 4) C4366

Compiler Warning C4368

Compiler Warning (level 1) C4369

Compiler Warning (level 1) C4374

Compiler Warning (level 1) C4375

Compiler Warning (level 1) C4376

Compiler Warning (level 1) C4377

Compiler Warning (level 1) C4378

Compiler Warning (level 1) C4379

Compiler Warning (level 1) C4381

Compiler Warning (level 1) C4382

Compiler Warning (level 1) C4383

Compiler Warning (level 1) C4384

Compiler Warning (level 4) C4389

Compiler Warning (level 3) C4390

Compiler Warning (level 1) C4391

Compiler Warning (level 1) C4392

Compiler Warning (level 1) C4393

Compiler Warning C4394

Compiler Warning (level 1) C4395

Compiler Warning (level 1) C4397

Compiler Warning (level 3) C4398

Compiler Warning (level 1) C4399

Compiler Warning (level 4) C4400

Compiler Warning (level 1) C4401

Compiler Warning (level 1) C4402

Compiler Warning (level 1) C4403

Compiler Warning (level 3) C4404

Compiler Warning (level 1) C4405

Compiler Warning (level 1) C4406

Compiler Warning (level 1) C4407

Compiler Warning (level 4) C4408

Compiler Warning (level 1) C4409

Compiler Warning (level 1) C4410

Compiler Warning (level 1) C4411

Compiler Warning (level 2) C4412

Compiler Warning (level 3) C4414

Compiler Warning (level 1) C4420

Compiler Warning (level 4) C4428

Compiler Warning (level 4) C4429

Compiler Warning C4430

Compiler Warning (level 4) C4431

Compiler Warning (level 4) C4432

Compiler Warning (level 4) C4433

Compiler Warning (level 4) C4434

Compiler Warning C4439

Compiler Warning (level 1) C4440

Compiler Warning (level 1) C4441

Compiler Warning (level 1) C4445

Compiler Warning (level 4) C4460

Compiler Warning (level 1) C4461

Compiler Warning (level 1) C4470

Compiler Warning (level 4) C4480

Compiler Warning (level 4) C4481

Compiler Warning (level 1) C4482

Compiler Warning C4484

Compiler Warning C4485

Compiler Warning (level 1) C4486

Compiler Warning (level 4) C4487

Compiler Warning (level 1) C4488

Compiler Warning (level 1) C4489

Compiler Warning (level 1) C4490

Compiler Warning (level 1) C4502

Compiler Warning (level 1) C4503

Compiler Warning (level 4) C4505

Compiler Warning (level 1) C4506

Compiler Warning (level 1) C4508

Compiler Warning (level 3) C4509

Compiler Warning (level 4) C4510

Compiler Warning (level 3) C4511

Compiler Warning (level 4) C4512

Compiler Warning (level 4) C4513

Compiler Warning (level 4) C4514

Compiler Warning (level 4) C4515

Compiler Warning (level 4) C4516

Compiler Warning (level 4) C4517

Compiler Warning (level 1) C4518

Compiler Warning (level 1) C4519

Compiler Warning (level 3) C4520

Compiler Warning (level 3) C4521

Compiler Warning (level 3) C4522

Compiler Warning (level 3) C4523

Compiler Warning (level 1) C4526

Compiler Warning (level 1) C4530

Compiler Warning (level 1) C4532

Compiler Warning (level 1) C4533

Compiler Warning (level 3) C4534

Compiler Warning (level 3) C4535

Compiler Warning (level 4) C4536

Compiler Warning (level 1) C4537

Compiler Warning (level 3) C4538

Compiler Warning (level 1) C4539

Compiler Warning (level 1) C4540

Compiler Warning (level 1) C4541

Compiler Warning (level 3) C4543

Compiler Warning (level 1) C4544

Compiler Warning (level 1) C4545

Compiler Warning (level 1) C4546

Compiler Warning (level 1) C4547

Compiler Warning (level 1) C4548

Compiler Warning (level 1) C4549

Compiler Warning (level 1) C4550

Compiler Warning (level 1) C4551

Compiler Warning (level 1) C4552

Compiler Warning (level 1) C4553

Compiler Warning (level 3) C4554

Compiler Warning (level 1) C4555

Compiler Warning (level 1) C4556

Compiler Warning (level 3) C4557

Compiler Warning (level 1) C4558

Compiler Warning (level 4) C4559

Compiler Warning (level 1) C4561

Compiler Warning (level 4) C4564

Compiler Warning (level 4) C4565

Compiler Warning (level 1) C4566

Compiler Warning (level 3) C4570

Compiler Warning (level 4) C4571

Compiler Warning (level 1) C4572

Compiler Warning (level 3) C4580

Compiler Warning (level 1) C4581

Compiler Warning (level 1) C4584

Compiler Warning (levels 2 and 4) C4200

Error Message
nonstandard extension used : zero-sized array in struct/union

A structure or union contains an array with zero size.

Level-2 warning when compiling a C++ file and a Level-4 warning when compiling a C file.

Example

```
// C4200.cpp
// compile with: /W2
#include <stdio.h>
struct A {
int a[0];  // C4200
};
int main() {
}
```

Compiler Warning (level 4) C4201

Error Message
nonstandard extension used : nameless struct/union

Under Microsoft extensions (/Ze), you can specify a structure without a declarator as members of another structure or union. These structures generate an error under ANSI compatibility (/Za).

Example

```
// C4201.cpp
// compile with: /W4
struct S
{
   float y;
   struct
   {
      int a, b, c;  // C4201
   };
} *p_s;

int main()
{
}
```

Compiler Warning (level 4) C4202

Error Message
nonstandard extension used : '...': prototype parameter in name list illegal

An old-style function definition contains variable arguments. These definitions generate an error under ANSI compatibility (/Za).

Example

```
// C4202.c
// compile with: /W4
void func( a, b, ...)   // C4202
int a, b;
{}

int main()
{
}
```

Compiler Warning (level 4) C4203

Error Message
nonstandard extension used : union with static member variable

Static union members are valid using Microsoft extensions (/Ze). Such members are invalid in the ANSI standard (/Za). Conforming to the ANSI standard ensures more portable programs. The following sample generates C4203:

```
// C4203.cpp
// compile with: /W4
union test
{
   static int a;   // C4203, remove static to resolve
};

int main()
{
}
```

Compiler Warning (level 4) C4204

Error Message
nonstandard extension used : non-constant aggregate initializer

With Microsoft extensions (/Ze), you can initialize aggregate types (arrays, structures, unions, and classes) with values that are not constants.

Example

```
// C4204.c
// compile with: /W4
int func1()
{
   return 0;
}
struct S1
{
   int i;
};

int main()
{
   struct S1 s1 = { func1() };   // C4204
   return s1.i;
}
```

Such initializations are invalid under ANSI compatibility (/Za).

Compiler Warning (level 4) C4205

Error Message
nonstandard extension used : static function declaration in function scope

With Microsoft extensions (/Ze), **static** functions can be declared inside another function. The function is given global scope.

Example

```
// C4205.c
// compile with: /W4
void func1()
{
   static int func2();  // C4205
};

int main()
{
}
```

Such initializations are invalid under ANSI compatibility (/Za).

Compiler Warning (level 4) C4206

nonstandard extension used : translation unit is empty

The file was empty after preprocessing.

This extension can prevent your code from being portable to other compilers. It generates an error under ANSI compatibility (/Za) and only applies to C source code.

Compiler Warning (level 4) C4207

Error Message
nonstandard extension used : extended initializer form

With Microsoft extensions (/Ze), you can initialize an unsized array of **char** using a string within braces.

Example

```
// C4207.c
// compile with: /W4
char c[] = { 'a', 'b', "cdefg" }; // C4207

int main()
{
}
```

Such initializations are invalid under ANSI compatibility (/Za).

Compiler Warning (level 4) C4208

Error Message
nonstandard extension used : delete [exp] - exp evaluated but ignored

With Microsoft extensions (/Ze), you can delete an array using a value within brackets with the delete operator. The value is ignored.

```
// C4208.cpp
// compile with: /W4
int main()
{
   int * MyArray = new int[18];
   delete [18] MyArray;      // C4208
   MyArray = new int[18];
   delete [] MyArray;        // ok
}
```

Such values are invalid under ANSI compatibility (/Za).

Compiler Warning (level 4) C4210

Error Message
nonstandard extension used : function given file scope

With the default Microsoft extensions (/Ze), function declarations have file scope.

```
// C4210.c
// compile with: /W4 /c
void func1()
{
   extern int func2( double );   // C4210 expected
}

int main()
{
   func2( 4 );   //  /Ze passes 4 as type double
}                //  /Za passes 4 as type int
```

This extension can prevent your code from being portable to other compilers.

Compiler Warning (level 4) C4211

Error Message
nonstandard extension used : redefined extern to static

With the default Microsoft extensions (/Ze), you can redefine an **extern** identifier as **static**.

Example

```
// C4211.c
// compile with: /W4
extern int i;
static int i;   // C4211

int main()
{
}
```

Such redefinitions are invalid under ANSI compatibility (/Za).

See Also

Reference
Static Storage-Class Specifiers

Compiler Warning (level 4) C4212

Error Message
nonstandard extension used : function declaration used ellipsis

The function prototype has a variable number of arguments. The function definition does not.

The following sample generates C4212:

```
// C4212.c
// compile with: /W4 /Ze /c
void f(int , ...);
void f(int i, int j) {}
```

Compiler Warning (level 4) C4213

Error Message
nonstandard extension used : cast on l-value

With the default Microsoft extensions (/Ze), you can use casts on the left side of an assignment statement.

Example

```
// C4213.c
// compile with: /W4
void *a;
void f()
{
   int   i[3];
   a = &i;
   *(( int * )a )++ = 3;  // C4213
}

int main()
{
}
```

Such casts are invalid under ANSI compatibility (/Za).

Compiler Warning (level 4) C4214

Error Message
nonstandard extension used : bit field types other than int

With the default Microsoft extensions (/Ze), bitfield structure members can be of any integer type.

Example

```
// C4214.c
// compile with: /W4
struct bitfields
{
   unsigned short j:4;  // C4214
};

int main()
{
}
```

Such bit fields are invalid under ANSI compatibility (/Za).

Compiler Warning (level 1) C4215

Error Message
nonstandard extension used : long float

The default Microsoft extensions (/Ze) treat **long float** as **double**. ANSI compatibility (/Za) does not. Use **double** to maintain compatibility.

The following sample generates C4215:

```
// C4215.cpp
// compile with: /W1 /LD
long float a;   // C4215

// use the line below to resolve the warning
// double a;
```

Compiler Warning (level 1) C4216

Error Message
nonstandard extension used : float long

The default Microsoft extensions (/Ze) treat **float long** as **double**. ANSI compatibility (/Za) does not. Use **double** to maintain compatibility. The following sample generates C4216:

```
// C4216.cpp
// compile with: /W1
float long a;   // C4216

// use the line below to resolve the warning
// double a;

int main() {
}
```

Compiler Warning (level 1) C4218

Error Message
nonstandard extension used : must specify at least a storage class or a type

With the default Microsoft extensions (/Ze), you can declare a variable without specifying a type or storage class. The default type is **int**.

Example

```
// C4218.c
// compile with: /W4
i;  // C4218

int main()
{
}
```

Such declarations are invalid under ANSI compatibility (/Za).

Compiler Warning (level 4) C4220

Error Message
varargs matches remaining parameters

Under the default Microsoft extensions (/Ze), a pointer to a function matches a pointer to a function with similar, but variable, arguments.

Example

```
// C4220.c
// compile with: /W4

int ( *pFunc1) ( int a, ... );
int ( *pFunc2) ( int a, int b);

int main()
{
   if ( pFunc1 != pFunc2 ) {};  // C4220
}
```

Such pointers do not match under ANSI compatibility (/Za).

Compiler Warning (level 4) C4221

Error Message
nonstandard extension used : 'identifier' : cannot be initialized using address of automatic variable

With the default Microsoft extensions (/Ze), you can initialize an aggregate type (**array**, **struct**, or **union**) with the address of a local (automatic) variable.

Example

```
// C4221.c
// compile with: /W4
struct S
{
   int *i;
};

void func()
{
   int j;
   struct S s1 = { &j };   // C4221
}

int main()
{
}
```

Such initializations are invalid under ANSI compatibility (/Za).

Compiler Warning (levels 1 and 4) C4223

Error Message
nonstandard extension used : non-lvalue array converted to pointer

In standard C, you cannot convert a non-lvalue array to a pointer. With the default Microsoft extensions (/Ze), you can.

Compiler Warning (level 1) C4224

Error Message
nonstandard extension used : formal parameter 'identifier' was previously defined as a type

The identifier was previously used as a **typedef**. This causes a warning under ANSI compatibility (/Za).

Example

```
// C4224.cpp
// compile with: /Za /W1 /LD
typedef int I;
void func ( int I );  // C4224
```

Compiler Warning (level 1) C4226

Error Message
nonstandard extension used : 'keyword' is an obsolete keyword

The current version of Visual C++ does not use this keyword.

Compiler Warning (level 1) C4227

Error Message
anachronism used : qualifiers on reference are ignored

Using qualifiers like **const** or **volatile** with C++ references is an outdated practice.

Example

```
// C4227.cpp
// compile with: /W1 /c
int j = 0;
int &const i = j;   // C4227
```

Compiler Warning (level 1) C4228

Error Message
nonstandard extension used : qualifiers after comma in declarator list are ignored

Use of qualifiers like **const** or **volatile** after a comma when declaring variables is a Microsoft extension (/Ze).

Example

```
// C4228.cpp
// compile with: /W1
int j, const i = 0;  // C4228
int k;
int const m = 0;  // ok
int main()
{
}
```

Compiler Warning (level 1) C4229

Error Message
anachronism used : modifiers on data are ignored

Using a Microsoft modifier such as **__cdecl** on a data declaration is an outdated practice.

Example

```
// C4229.cpp
// compile with: /W1 /LD
int __cdecl counter;   // C4229 cdecl ignored
```

Compiler Warning (level 1) C4230

Error Message
anachronism used : modifiers/qualifiers interspersed; qualifier ignored

Using a qualifier before a Microsoft modifier such as **__cdecl** is an outdated practice.

Example

```
// C4230.cpp
// compile with: /W1 /LD
int __cdecl const function1();   // C4230 const ignored
```

Compiler Warning (level 3) C4231

Error Message
nonstandard extension used : 'identifier' before template explicit instantiation

When Microsoft extensions are enabled (/Ze), an **extern** template can be instantiated, generating this warning. Under ANSI compatibility (/Za), such instantiations cause an error.

Example

```
// C4231.cpp
// compile with: /W3
template<class T, int i> class MyStack {}; // C4231
extern template MyStack< int, 4>;

int main()
{
}
```

Compiler Warning (level 4) C4232

Error Message
nonstandard extension used : 'identifier' : address of dllimport 'dllimport' is not static, identity not guaranteed

Under Microsoft extensions (/Ze), you can give a nonstatic value as the address of a function declared with the **dllimport** modifier. Under ANSI compatibility (/Za), this causes an error.

The following sample generates C4232:

```
// C4232.c
// compile with: /W4 /Ze /c
int __declspec(dllimport) f();
int (*pfunc)() = &f;   // C4232
```

Compiler Warning (level 4) C4233

Error Message
nonstandard extension used : 'keyword' keyword only supported in C++, not C

The compiler compiled your source code as C rather than C++, and you used a keyword that is only valid in C++. The compiler compiles your source file as C if the extension of the source file is .c or you use /Tc.

This warning is automatically promoted to an error. If you wish to modify this behavior, use #pragma warning. For example, to make C4233 into a level 4 warning issue,

```
#pragma warning(2:4233)
```

in your source code file.

Compiler Warning (level 4) C4234

Error Message
nonstandard extension used: 'keyword' keyword reserved for future use

The compiler does not yet implement the keyword you used.

This warning is automatically promoted to an error. If you wish to modify this behavior, use #pragma warning. For example, to make C4234 into a level 4 warning issue,

```
#pragma warning(2:4234)
```

in your source code file.

Compiler Warning (level 4) C4235

Error Message
nonstandard extension used : 'keyword' keyword not supported on this architecture

The compiler does not support the keyword you used.

For example, the **_asm** keyword is not supported for the Itanium Processor Family (IPF) compiler. All assembly code must be written in a separate file or should be used through an intrinsic. See Intrinsics Available on Intel 64-bit Platforms for more information.

This warning is automatically promoted to an error. If you wish to modify this behavior, use #pragma warning. For example, to make C4235 into a level 2 warning, use the following line of code

```
#pragma warning(2:4235)
```

in your source code file.

Compiler Warning (level 1) C4237

Error Message
'keyword' keyword is not yet supported, but reserved for future use

A keyword in the C++ specification is not implemented in the Visual C++ compiler, but the keyword is not available as a user-defined symbol.

The following sample generates C4237:

```
// C4237.cpp
// compile with: /W1 /c
int export;   // C4237
```

Compiler Warning (level 4) C4238

Error Message
nonstandard extension used : class rvalue used as lvalue

For compatibility with previous versions of Visual C++, Microsoft extensions (**/Ze**) allow you to use a class type as an rvalue in a context that implicitly or explicitly takes its address. In some cases, such as the example below, this can be dangerous.

Example

```
// C4238.cpp
// compile with: /W4 /c
struct C {
   C() {}
};

C * pC = &C();   // C4238
```

This usage causes an error under ANSI compatibility (/Za).

Compiler Warning (level 4) C4239

Error Message
nonstandard extension used : 'token' : conversion from 'type' to 'type'

This type conversion is not allowed by the C++ standard, but it is permitted here as an extension. This warning is always followed by at least one line of explanation describing the language rule being violated.

Example

The following sample generates C4239.

```
// C4239.cpp
// compile with: /W4 /c
struct C {
   C() {}
};

void func(void) {
   C & rC = C();   // C4239
   const C & rC2 = C();   // OK
   rC2;
}
```

Conversion from integral type to enum type is not strictly allowed.

The following sample generates C4239.

```
// C4239b.cpp
// compile with: /W4 /c
enum E { value };
struct S {
   E e : 2;
} s = { 5 };   // C4239
// try the following line instead
// } s = { (E)5 };
```

Compiler Warning (level 3) C4240

Error Message

nonstandard extension used : access to 'classname' now defined to be 'access specifier', previously it was defined to be 'access specifier'

Under ANSI compatibility (/Za), you cannot change the access to a nested class. Under the default Microsoft extensions (/Ze), you can, with this warning.

Example

```
// C4240.cpp
// compile with: /W3
class X
{
private:
   class N;
public:
   class N
   {   // C4240
   };
};

int main()
{
}
```

Compiler Warning (level 4) C4242

Error Message
'identifier' : conversion from 'type1' to 'type2', possible loss of data

The types are different. Type conversion may result in loss of data. The compiler makes the type conversion.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

For additional information on C4242, see Common Compiler Errors.

The following sample generates C4242:

```
// C4242.cpp
// compile with: /W4
#pragma warning(4:4242)
int func() {
   return 0;
}

int main() {
   char a;
   a = func();   // C4242, return type and variable type do not match
}
```

Compiler Warning (level 3) C4243

Error Message
'conversion type' conversion exists from 'type1' to 'type2', but is inaccessible

A pointer to a derived class is converted to a pointer to a base class, but the derived class inherits the base class with private or protected access.

The following sample generates C4243:

```
// C4243.cpp
// compile with: /W3
// C4243 expected
struct B {
   int f() {
      return 0;
   };
};

struct D : private B {};
struct E : public B {};

int main() {
   // Delete the following 2 lines to resolve.
   int (D::* d)() = (int(D::*)()) &B::f;
   d;

   int (E::* e)() = (int(E::*)()) &B::f; // OK
   e;
}
```

Compiler Warning (level 2) C4244

Error Message
'argument' : conversion from 'type1' to 'type2', possible loss of data

A floating point type was converted to an integer type. A possible loss of data may have occurred.

If you get C4244, you should either change your program to use compatible types, or add some logic to your code, to ensure that the range of possible values will always be compatible with the types you are using.

C4244 can also fire at level 3, and 4; see Compiler Warning (levels 3 and 4) C4244 for more information.

Example
The following sample generates C4244:

```
// C4244_level2.cpp
// compile with: /W2

int f(int x){ return 0; }
int main() {
   double x = 10.1;
   int i = 10;
   return (f(x));   // C4244
   // try the following line instead
   // return (f(i));
}
```

Compiler Warning (levels 3 and 4) C4244

Error Message
'conversion' conversion from 'type1' to 'type2', possible loss of data

An integer type is converted to a smaller integer type. This is a level-4 warning if *type1* is **int** and *type2* is smaller than **int**. Otherwise, it is a level 3 (assigned a value of type __int64 to a variable of type **unsigned int**). A possible loss of data may have occurred.

If you get C4244, you should either change your program to use compatible types, or add some logic to your code, to ensure that the range of possible values will always be compatible with the types you are using.

C4244 can also fire at level 2; see Compiler Warning (level 2) C4244 for more information.

The conversion may have a problem due to implicit conversions.

The following sample generates C4244:

```
// C4244_level4.cpp
// compile with: /W4
int aa;
unsigned short bb;
int main() {
   int b = 0, c = 0;
   short a = b + c;   // C4244

   bb += c;   // C4244
   bb = bb + c;   // C4244
   bb += (unsigned short)aa;   // C4244
   bb = bb + (unsigned short)aa;   // OK
}
```

For more information, see Usual Arithmetic Conversions.

```
// C4244_level3.cpp
// compile with: /W3
int main() {
   __int64 i = 8;
   unsigned int ii = i;   // C4244
}
```

C4244 can also occur when compiling with /Wp64. Note that /Wp64 is enabled by default for projects created with the application wizard.

The following sample generates C4244:

```
// C4244_level3_b.cpp
// compile with: /W3 /Wp64
int main() {
   char* p1 = 0;
   char* p2 = 0;
   int x = p2 - p1;   // C4244
}
```

Compiler Warning (level 4) C4245

Error Message
'conversion' : conversion from 'type1' to 'type2', signed/unsigned mismatch

You tried to convert a signed **const** that has a negative value to an **unsigned**.

The following sample generates C4245:

```
// C4245.cpp
// compile with: /W4 /c
const int i = -1;
unsigned int j = i; // C4245

const int k = 1;
unsigned int l = k; // okay

int m = -1;
unsigned int n = m; // okay

void Test(size_t i) {}

int main() {
   Test( -19 );   // C4245
}
```

Compiler Warning (level 2) C4250

Error Message
'class1' : inherits 'class2::member' via dominance

Two or more members have the same name. The one in *class2* is inherited because it is a base class for the other classes that contained this member.

To suppress C4250, use the warning pragma.

Because a virtual base class is shared among multiple derived classes, a name in a derived class dominates a name in a base class. For example, given the following class hierarchy, there are two definitions of func inherited within diamond: the vbc::func() instance through the weak class, and the dominant::func() through the dominant class. An unqualified call of func() through a diamond class object, always calls the dominate::func() instance. If the weak class were to introduce an instance of func(), neither definition would dominate, and the call would be flagged as ambiguous.

```
// C4250.cpp
// compile with: /c /W2
#include <stdio.h>
struct vbc {
   virtual void func() { printf("vbc::func\n"); }
};

struct weak : public virtual vbc {};

struct dominant : public virtual vbc {
   void func() { printf("dominant::func\n"); }
};

struct diamond : public weak, public dominant {};

int main() {
   diamond d;
   d.func();   // C4250
}
```

Example
The following sample generates C4250.

```
// C4250_b.cpp
// compile with: /W2 /EHsc
#include <iostream>
using namespace std;
class A {
public:
   virtual operator int () {
      return 2;
   }
};

class B : virtual public A {
public:
   virtual operator int () {
      return 3;
   }
};

class C : virtual public A {};

class E : public B, public C {};   // C4250

int main() {
   E eObject;
   cout << eObject.operator int() << endl;
```

```
}
```

This sample shows a more complex situation. The following sample generates C4250.

```
// C4250_c.cpp
// compile with: /W2 /EHsc
#include <iostream>
using namespace std;

class V {
public:
   virtual int f() {
      return 1024;
   }
};

class B : virtual public V {
public:
   int b() {
      return f(); // B::b() calls V::f()
   }
};

class M : virtual public V {
public:
   int f() {
      return 7;
   }
};

// because of dominance, f() is M::f() inside D,
// changing the meaning of B::b's f() call inside a D
class D : public B, public M {};   // C4250

int main() {
   D d;
   cout << "value is: " << d.b();   // invokes M::f()
}
```

Compiler Warning (level 1) C4251

Error Message
'identifier' : class 'type' needs to have dll-interface to be used by clients of class 'type2'

To minimize the possibility of data corruption when exporting a class with __declspec(dllexport), ensure that:

- All your static data is access through functions that are exported from the DLL.
- No inlined methods of your class can modify static data.
- No inlined methods of your class use CRT functions or other library functions use static data (see PRB: Potential Errors Passing CRT Objects Across DLL Boundaries for more information).
- No methods of your class (regardless of inlining) can use types where the instantiation in the EXE and DLL have static data differences.

You can avoid exporting classes by defining a DLL that defines a class with virtual functions, and functions you can call to instantiate and delete objects of the type. You can then just call virtual functions on the type.

For more information on exporting templates, see http://support.microsoft.com/default.aspx?scid=KB;EN-US;168958.

C4251 can be ignored in Microsoft Visual C++ 2005 if you are deriving from a type in the Standard C++ Library, compiling a debug release (**/MTd**) and where the compiler error message refers to _Container_base.

```
// C4251.cpp
// compile with: /EHsc /MTd /W2 /c
#include <vector>
using namespace std;
class Node;
class __declspec(dllimport) VecWrapper : vector<Node *> {};   // C4251
```

Compiler Warning (level 4) C4254

Error Message
'operator' : conversion from 'type1' to 'type2', possible loss of data

A larger bit field was assigned to a smaller bit field. There could be a loss of data.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4254:

```
// C4254.cpp
// compile with: /W4
#pragma warning(default: 4254)

struct X {
   int a : 20;
   int b : 12;
};

int main() {
   X *x = new X();
   x->b = 10;
   x->a = 4;
   x->a = x->b;    // OK
   x->b = x->a;    // C4254
};
```

Compiler Warning (level 4) C4255

Error Message
'function' : no function prototype given: converting '()' to '(void)'

The compiler did not find an explicit list of arguments to a function.

This warning applies only to the C compiler.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates the warning C4255:

```
// C4255.c
// compile with: /W4 /WX
#pragma warning (default : 4255)

void f()  { // C4255
// try the following line instead
//void f(void) {
}

int main(int argc, char *argv[]) {
   f();
}
```

Compiler Warning (level 4) C4256

Error Message
'function' : constructor for class with virtual bases has '...'; calls may not be compatible with older versions of Visual C++

Possible incompatibility.

Based on the code sample below, if the definition of the constructor S2::S2(int i, ...) was compiled with a version of Visual C++ prior to 7, but the code snippet here were compiled with current version, the call to the constructor for S3 would not work correctly, due to a special-case calling convention change. Note that if both were compiled with Visual C++ 6.0, the call would not work quite right either, unless no parameters were passed for the ellipsis.

To fix this warning,

1. Don't use ellipsis in a constructor.
2. Make sure that all components in their project are built with the current version (including any libraries that may define or reference this class), then disable the warning using the warning pragma.

The following sample generates C4256:

```
// C4256.cpp
// compile with: /W4
// #pragma warning(disable : 4256)
struct S1
{
};

struct S2: virtual public S1
{
   S2( int i, ... )    // C4256
   {
      i = 0;
   }
   /*
   // try the following line instead
   S2( int i)
   {
      i = 0;
   }
   */
};

void func1()
{
   S2 S3( 2, 1, 2 );   // C4256
   // try the following line instead
   // S2 S3( 2 );
}

int main()
{
}
```

Compiler Warning (level 1) C4258

Error Message
'variable' : definition from the for loop is ignored; the definition from the enclosing scope is used"

Under /Ze and /Zc:forScope, variables defined in a for loop go out of scope after the **for** loop ends. This warning occurs if a variable with the same name as the loop variable, but defined in the enclosing loop, is used again in the scope containing the **for** loop. For example:

```
// C4258.cpp
// compile with: /Zc:forScope /W1
int main()
{
   int i;
   {
      for (int i =0; i < 1; i++)
         ;
      i = 20;   // C4258 i (in for loop) has gone out of scope
   }
}
```

Compiler Warning (level 4) C4263

Error Message
'function' : member function does not override any base class virtual member function

A class function definition has the same name as a virtual function in a base class but not the same number or type of arguments. This effectively hides the virtual function in the base class.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4263:

```
// C4263.cpp
// compile with: /W4
#pragma warning(default:4263)
#pragma warning(default:4264)
class B {
public:
   virtual void func();
};

class D : public B {
   void func(int);   // C4263
};

int main() {
}
```

Compiler Warning (level 1) C4264

Error Message
'virtual_function' : no override available for virtual member function from base 'class'; function is hidden

C4264 is always generated after C4263.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

Compiler Warning (level 3) C4265

Error Message
'class' : class has virtual functions, but destructor is not virtual

When a class has virtual functions but a nonvirtual destructor, objects of the type might not be destroyed properly when the class is destroyed through a base class pointer.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4265:

```
// C4265.cpp
// compile with: /W3 /c
#pragma warning(default : 4265)
class B
{
public:
   virtual void vmf();

   ~B();
   // try the following line instead
   // virtual ~B();
};   // C4265

int main()
{
   B b;
}
```

Compiler Warning (level 4) C4266

Error Message
'function' : no override available for virtual member function from base 'type'; function is hidden

A derived class did not override all overloads of a virtual function.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4266:

```
// C4266.cpp
// compile with: /W4 /c
#pragma warning (default : 4266)
class Engine {
public:
   virtual void OnException(int&,int);
   virtual void OnException(int&,int&,int);
};

class LocalBinding : private Engine {
   virtual void OnException(int&,int);
};   // C4266
```

Possible resolution:

```
// C4266b.cpp
// compile with: /W4 /c
#pragma warning (default : 4266)
class Engine {
public:
   virtual void OnException(int&,int);
   virtual void OnException(int&,int&,int);
};

class LocalBinding : private Engine {
   virtual void OnException(int&,int);
   virtual void OnException(int&, int&, int);
};
```

Compiler Warning (level 3) C4267

Error Message
'var' : conversion from 'size_t' to 'type', possible loss of data

When compiling with /Wp64, or when compiling on a 64-bit operating system, ***type*** is 32 bits but **size_t** is 64 bits when compiling for 64-bit targets.

To fix this warning, use **size_t** instead of a ***type***.

Example
The following sample generates C4267.

```
// C4267.cpp
// compile with: /W3 /Wp64
#include <Windows.h>
void Func(int i) {}
void Func2(DWORD i) {}
void Func3(size_t i) {}

int main() {
   size_t bufferSize = 10;
   Func(bufferSize);   // C4267
   Func2(bufferSize);   // C4267
   Func3(bufferSize);   // OK
}
```

C4267 can also be caused on x86 and this warning cannot be resolved in code, but can be ignored and suppressed with the warning pragma.

In this situation, C4267 is caused by a limitation in **/Wp64** warnings. On x86, **std::cout<<range_index** resolves to the overload of **operator<<** that accepts an **unsigned int**, since **size_t** is an **unsigned int** on Win32. C4267 occurs because you passed a **size_t** as an **unsigned int** argument, which would cause truncation on Win64 where **size_t** is 64-bit, but **unsigned int** is still 32-bit. This can be ignored because if you compiled for Win64, **std::cout<<range_index** would resolve to the overload of **operator<<** that accepts an **unsigned __int64**, since that's what type **size_t** is on Win64. The 32-bit compiler doesn't notice, so it warns.

The following sample generates C4267.

```
// C4267_b.cpp
// compile with: /W3 /Wp64 /EHsc
// processor: x86
#include <fstream>
#include <vector>
using namespace std;

int main() {
   ofstream outputfile;
   vector<char> vecbuffer(10);
   outputfile.write( &vecbuffer[0], vecbuffer.size() );   // C4267
}
```

Compiler Warning (level 4) C4268

Error Message
'identifier' : 'const' static/global data initialized with compiler generated default constructor fills the object with zeros

A **const** global or static instance of a non-trivial class is initialized with a compiler-generated default constructor.

Example

```
// C4268.cpp
// compile with: /c /LD /W4
class X {
public:
   int m_data;
};

const X x1;   // C4268
```

As this instance of the class is **const**, the value of `m_data` cannot be changed.

Compiler Warning (level 1) C4269

Error Message
'identifier' : 'const' automatic data initialized with compiler generated default constructor produces unreliable results

A **const** automatic instance of a non-trivial class is initialized with a compiler-generated default constructor.

Example

```
// C4269.cpp
// compile with: /c /LD /W1
class X {
public:
   int m_data;
};

void g() {
   const X x1;   // C4269
};
```

Since this instance of the class is generated on the stack, the initial value of `m_data` can be anything. Also, since it is a **const** instance, the value of `m_data` can never be changed.

Compiler Warning (level 1) C4272

Error Message
'function' : is marked __declspec(dllimport); must specify native calling convention when importing a function.

It is an error to export a function marked with the __clrcall calling convention, and the compiler issues this warning if you attempt to import a function marked **__clrcall**.

The following sample generates C4272:

```
// C4272.cpp
// compile with: /c /W1 /clr
__declspec(dllimport) void __clrcall Test();   // C4272
__declspec(dllimport) void Test2();   // OK
```

Compiler Warning (level 1) C4273

Error Message
'function' : inconsistent DLL linkage

Two definitions in a file differ in their use of dllimport.

Example

The following sample generates C4273.

```
// C4273.cpp
// compile with: /W1 /c
char __declspec(dllimport) c;
char c;   // C4273, delete this line or the line above to resolve
```

The following sample generates C4273.

```
// C4273_b.cpp
// compile with: /W1 /clr /c
#include <stdio.h>
extern "C" int printf_s(const char *, ...);   // C4273
```

Compiler Warning (level 2) C4275

Error Message
non – DLL-interface classkey 'identifier' used as base for DLL-interface classkey 'identifier'

An exported class was derived from a class that was not exported.

To minimize the possibility of data corruption when exporting a class with __declspec(dllexport), ensure that:

- All your static data is accessed through functions that are exported from the DLL.
- No inlined methods of your class can modify static data.
- No inlined methods of your class use CRT functions or other library functions use static data.
- No inlined class functions use CRT functions, or other library functions, where, for example, you access static data.
- No methods of your class (regardless of inlining) can use types where the instantiation in the EXE and DLL have static data differences.

You can avoid exporting classes by defining a DLL that defines a class with virtual functions, and functions you can call to instantiate and delete objects of the type. You can then just call virtual functions on the type.

For more information on exporting templates, see http://support.microsoft.com/default.aspx?scid=KB;EN-US;168958.

C4275 can be ignored in Microsoft Visual C++ 2005 if you are deriving from a type in the Standard C++ Library, compiling a debug release (**/MTd**) and where the compiler error message refers to _Container_base.

```
// C4275.cpp
// compile with: /EHsc /MTd /W2 /c
#include <vector>
using namespace std;
class Node;
class __declspec(dllimport) VecWrapper : vector<Node *> {};   // C4275
```

Compiler Warning (level 1) C4276

Error Message
'function' : no prototype provided; assumed no parameters

When you take the address of a function with the __stdcall calling convention, you must give a prototype so the compiler can create the function's decorated name. Since *function* has no prototype, the compiler, when creating the decorated name, assumes the function has no parameters.

Compiler Warning (level 3) C4278

Error Message
'identifier': identifier in type library 'tlb' is already a macro; use the 'rename' qualifier

When using #import, an identifier in the typelib you are importing is attempting to declare an identifier ***identifier***. However, this is already a valid symbol.

Use the **#import rename** attribute to assign an alias to the symbol in the type library.

Compiler Warning (level 3) C4280

Error Message
'operator –>' was self recursive through type 'type'

Your code incorrectly allows **operator–>** to call itself.

The following sample generates C4280:

```
// C4280.cpp
// compile with: /W3 /WX
struct A
{
   int z;
   A& operator ->();
};

void f(A y)
{
   int i = y->z; // C4280
}
```

Compiler Warning (level 3) C4281

Error Message
'operator –>' recursion occurred through type 'type'

Your code allows **operator–>** to call itself.

The following sample generates C4281:

```
// C4281.cpp
// compile with: /W3 /WX
struct A;
struct B;
struct C;

struct A
{
   int z;
   B& operator->();
};

struct B
{
   C& operator->();
};

struct C
{
   A& operator->();
};

void f(A p)
{
   int i = p->z; // C4281
}
```

Compiler Warning (level 3) C4282

Error Message
then through type 'type'

This continuation of warning C4281 shows that **operator–>** calls itself through *type*.

Compiler Warning (level 3) C4283

Error Message
and through type 'type'

This continuation of warning C4281 shows that **operator–>** calls itself through *type*.

Compiler Warning (level 2) C4285

Error Message
return type for 'identifier::operator –>' is recursive if applied using infix notation

The specified **operator–>()** function cannot return the type for which it is defined or a reference to the type for which it is defined.

The following sample generates C4285:

```
// C4285.cpp
// compile with: /W2
class C
{
public:
    C operator->();   // C4285
   // C& operator->();  C4285, also
};

int main()
{
}
```

Compiler Warning (level 1) C4286

Error Message
'type1' : is caught by base class ('type2') on line number

The specified exception type is handled by a previous handler. The type for the second catch is derived from the type of the first. Exceptions for a base class catch exceptions for a derived class.

Example

```
//C4286.cpp
// compile with: /W1
#include <eh.h>
class C {};
class D : public  C {};
int main()
{
    try
    {
        throw "ooops!";
    }
    catch( C ) {}
    catch( D ) {}  // warning C4286, D is derived from C
}
```

Compiler Warning (level 3) C4287

Error Message
'operator' : unsigned/negative constant mismatch

An unsigned variable was used in an operation with a negative number.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

Example

The following sample generates C4287:

```
// C4287.cpp
// compile with: /W3
#pragma warning(default : 4287)
#include <stdio.h>

int main()
{
    unsigned int u = 1;
    if (u < -1)   // C4287
        printf_s("u LT -1");
    else
        printf_s("u !LT -1");
    return 0;
}
```

Compiler Warning (level 1) C4288

Error Message
nonstandard extension used : 'var' : loop control variable declared in the for-loop is used outside the for-loop scope; it conflicts with the declaration in the outer scope

When compiling with /Ze and **/Zc:forscope-**, a variable declared in a **for** loop was used after the for-loop scope. A Microsoft extension to the C++ language allows this variable to remain in scope, and C4288 reminds you that the first declaration of the variable is not used.

See /Zc:forScope for information about how to specify the Microsoft extension in **for** loops with /Ze.

The following sample generates C4288:

```
// C4288.cpp
// compile with: /W1 /c /Zc:forScope-
int main() {
   int i = 0;    // not used in this program
   for (int i = 0 ; ; ) ;
   i++;   // C4288 using for-loop declaration of i
}
```

Compiler Warning (level 4) C4289

Error Message
nonstandard extension used : 'var' : loop control variable declared in the for-loop is used outside the for-loop scope

When compiling with /Ze and **/Zc:forScope-**, a variable declared in a for loop was used after the **for**-loop scope.

See /Zc:forScope for information about how to specify standard behavior in **for** loops with **/Ze**.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4289:

```
// C4289.cpp
// compile with: /W4 /Zc:forScope-
#pragma warning(default:4289)
int main() {
   for (int i = 0 ; ; )   // C4289
      break;
   i++;
}
```

Compiler Warning (level 3) C4290

Error Message
C++ exception specification ignored except to indicate a function is not __declspec(nothrow)

A function is declared using exception specification, which Visual C++ accepts but does not implement. Code with exception specifications that are ignored during compilation may need to be recompiled and linked to be reused in future versions supporting exception specifications.

For more information, see Exception Specifications .

You can avoid this warning by using the warning pragma:

```
#pragma warning( disable : 4290 )
```

The following code sample generates C4290:

```
// C4290.cpp
// compile with: /EHs /W3 /c
void f1(void) throw(int) {}   // C4290

// OK
void f2(void) throw() {}
void f3(void) throw(...) {}
```

Compiler Warning (level 1) C4291

Error Message
'declaration' : no matching operator delete found; memory will not be freed if initialization throws an exception

A placement new is used for which there is no placement delete.

When memory is allocated for an object with operator **new**, the object's constructor is called. If the constructor throws an exception, any memory that was allocated for the object should be deallocated. This cannot take place unless an operator **delete** function exists that matches the operator **new**.

If you use the operator **new** without any extra arguments and compile with /GX, /EHs, or /EHa options to enable exception handling, the compiler will generate code to call operator **delete** if the constructor throws an exception.

If you use the placement form of the **new** operator (the form with arguments in addition to the size of the allocation) and the object's constructor throws an exception, the compiler will still generate code to call operator **delete**; but it will only do so if a placement form of operator **delete** exists matching the placement form of the operator **new** that allocated the memory. For example:

```
// C4291.cpp
// compile with: /EHsc /W1
#include <malloc.h>

class CList
{
public:
   CList(int)
   {
      throw "Fail!";
   }
};

void* operator new(size_t size, char* pszFilename, int nLine)
{
   return malloc(size);
}

int main(void)
{
   try
   {
      // This will call ::operator new(unsigned int) to allocate heap
      // memory. Heap memory pointed to by pList1 will automatically be
      // deallocated by a call to ::operator delete(void*) when
      // CList::CList(int) throws an exception.
      CList* pList1 = new CList(10);
   }
   catch (...)
   {
   }

   try
   {
      // This will call the overloaded ::operator new(size_t, char*, int)
      // to allocate heap memory. When CList::CList(int) throws an
      // exception, ::operator delete(void*, char*, int) should be called
      // to deallocate the memory pointed to by pList2. Since
      // ::operator delete(void*, char*, int) has not been implemented,
      // memory will be leaked when the deallocation cannot occur.
      CList* pList2 = new(__FILE__, __LINE__) CList(20);   // C4291
   }
   catch (...)
   {
   }
}
```

The above example generates warning C4291 because no placement form of operator **delete** has been defined that matches the placement form of operator **new**. To solve the problem, insert the following code above **main**. Notice that all of the overloaded operator **delete** function parameters match those of the overloaded operator **new**, except for the first parameter.

```
void operator delete(void* pMem, char* pszFilename, int nLine)
{
   free(pMem);
}
```

Compiler Warning (level 1) C4293

Error Message
'operator' : shift count negative or too big, undefined behavior

If a shift count is negative or too large, the behavior of the resulting image is undefined.

Example
The following sample generates C4293:

```
// C4293.cpp
// compile with: /c /W1
unsigned __int64 combine (unsigned lo, unsigned hi) {

   return (hi << 32) | lo;   // C4293

   // try the following line instead
   // return ( (unsigned __int64)hi << 32) | lo;
}
```

Compiler Warning (level 4) C4295

' *array* ' : array is too small to include a terminating null character

An array was initialized but the last character in the array is not a null; accessing the array may produce unexpected results.

The following sample generates C4295:

```
// C4295.c
// compile with: /W4

int main() {
   char a[3] = "abc";   // C4295
}
```

Compiler Warning (level 4) C4296

Error Message
'operator' : expression is always false

An unsigned variable was used in a comparison operation with zero.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4296:

```
// C4296.cpp
// compile with: /W4
#pragma warning(default : 4296)
int main()
{
   unsigned int u = 9;
   if (u < 0)    // C4296
      u++;
   if (u >= 0)   // C4296
      u++;
}
```

Compiler Warning (level 1) C4297

Error Message
'function' : function assumed not to throw an exception but does

A function contains a nothrow declaration and one or more throw statements. To resolve C4297, do not attempt to throw exceptions in functions that are declared with **nothrow** or to remove the **nothrow** specification.

For more information on exception specifications, see Exception Specifications. Also, see /EH (Exception Handling Model) for information on how to modify exception handling behavior at compile time.

This warning is also generated for __declspec(dllexport) functions marked extern "C", even if they are C++ functions.

The following sample generates C4297:

```
// C4297.cpp
// compile with: /W1 /LD
void __declspec(nothrow) f1()   // declared nothrow
// try the following line instead
// void f1()
{
   throw 1;   // C4297
}
```

Compiler Warning (level 1) C4301

Error Message
' derivedclass::function': overriding virtual function only differs from 'baselcass::function' by const/volatile qualifier

The qualifiers in the parameter lists for the functions in the base and derived classes differ. The compiler implements the function declaration in the derived class. To fix this warning, be sure the qualifiers match.

Example
The following sample generates C4301.

```
// C4301.cpp
// compile with: /W1
// processor: x86 IPF
#include <stdio.h>

class Base {
public:
   virtual void Func(const int i) {
      printf_s("base\n");
    }
};

class Derived : public Base {
public:
   // Delete the following line to resolve.
   void Func(int i) { printf_s("derived\n"); }   // C4301

   // Uncomment the following line to resolve.
   // void Func(const int i) { printf_s("derived\n"); }
};

int main() {
   Base B;
   Derived D;

   B.Func(0);
   D.Func(0);
}
```

Compiler Warning (level 2) C4302

Error Message
'conversion' : truncation from 'type 1' to 'type 2'

The compiler detected a conversion from a larger type to a smaller type. Information may be lost.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4302:

```
// C4302.cpp
// compile with: /W2
#pragma warning(default : 4302)
int main() {
   int i;
   char c = (char) &i;     // C4302
   short s = (short) &i;   // C4302
}
```

Compiler Warning (level 1) C4303

Error Message
'cast' from 'type1' to 'type2' is deprecated, use static_cast, __try_cast or dynamic_cast

C-style type casting or function-style casting is not supported when using Managed Extensions for C++. To cast, use either dynamic_cast Operator or static_cast Operator.

C4303 is not reachable using **/clr** and the new syntax; see C-Style Casts with /clr for more information.

The following sample generates C4303:

```
// C4303.cpp
// compile with: /clr:oldSyntax /W1
__gc struct A { };
__gc struct B { };

int main() {
   B *b = new B;
   try {

   // C4303 old 'c' style cast
   A *a = (A*)b;   // C4303
   // run time exception becaused casting to incompatible types

   // try the following line instead
   // A *a = __try_cast<A*>(b);
   }

   catch (System::InvalidCastException * e) {
      System::Console::WriteLine(e);
   }
}
```

Compiler Warning (level 1) C4305

Error Message
'identifier' : truncation from 'type1' to 'type2'

The identifier is converted to a smaller type, resulting in loss of information.

If you used the ATL Connection Point Wizard, see Knowledge Base article:

- Q250847 : ATL Connection Point Wizard Generated Code for Event with VARIANT Argument Gives C4305 Warning

Compiler Warning (level 3) C4306

' *identifier* ' : conversion from ' *type1* ' to ' *type2* ' of greater size

The identifier is type cast to a larger pointer. The unfilled high bits of the new type will be zero-filled.

This warning may indicate an unwanted conversion. The resulting pointer may not be valid.

Compiler Warning (level 2) C4307

Error Message
'operator' : integral constant overflow

The operator is used in an expression that results in an integer constant overflowing the space allocated for it. You may need to use a larger type for the constant. A **signed int** holds a smaller value than an **unsigned int** because the **signed int** uses one bit to represent the sign.

The following sample generates C4307:

```
// C4307.cpp
// compile with: /W2
int i = 2000000000 + 2000000000;   // C4307
int j = (unsigned)2000000000 + 2000000000;   // OK

int main()
{
}
```

Compiler Warning (level 2) C4308

Error Message
negative integral constant converted to unsigned type

An expression converts a negative integer constant to an unsigned type. The result of the expression is probably meaningless.

Example

```
// C4308.cpp
// compile with: /W2
unsigned int u = (-5 + 3U);   // C4308

int main()
{
}
```

Compiler Warning (level 2) C4309

Error Message
'conversion' : truncation of constant value

The type conversion causes a constant to exceed the space allocated for it. You may need to use a larger type for the constant.

The following sample generates C4309:

```
// C4309.cpp
// compile with: /W2
int main()
{
   char c = 128;   // C4309
}
```

Compiler Warning (level 3) C4310

Error Message
cast truncates constant value

A constant value is cast to a smaller type. The compiler performs the cast, which truncates data. The following sample generates C4310:

```
// C4310.cpp
// compile with: /W4
int main() {
   long int a;
   a = (char) 128;   // C4310, use value 0-127 to resolve
}
```

Compiler Warning (level 1) C4311

Error Message
'variable' : pointer truncation from 'type' to 'type'

This warning detects 64-bit portability issues. For example, if code is compiled on a 64-bit platform, the value of a pointer (64 bits) will be truncated if it is assigned to an int (32 bits).

This warning is only issued when /Wp64 is used. See /Wp64 for more information. Also, see Rules for Using Pointers.

For additional information on C4311, see Common Compiler Errors.

The following code sample generates C4311:

```
// C4311.cpp
// compile with: /W1 /Wp64
#include <stddef.h>
int main() {
   void* p = &p;
   unsigned long i = (unsigned long)p;   // C4311
   intptr_t j = (intptr_t)p;   // OK
}
```

Compiler Warning (level 1) C4312

Error Message
'operation' : conversion from 'type1' to 'type2' of greater size

This warning detects 64-bit portability issues. You attempted to assign a 32-bit value to a 64-bit type. For example, casting a 32-bit int or 32-bit long to a 64-bit pointer.

This can be an unsafe conversion in some circumstances when sign extension occurs. If a negative number is assigned to a pointer type of a size greater than the int, sign extension will occur and the pointer value will refer to a memory address different from the value of the int.

This warning is only issued when /Wp64 is used. See /Wp64 for more information. Also, see Rules for Using Pointers.

The following code sample generates C4312:

```
// C4312.cpp
// compile with: /W1 /Wp64 /LD
void* f(int i) {
   return (void*)i;   // C4312
}

// OK
void* f2(__int64 i) {
   return (void*)i;
}
```

Compiler Warning (level 1) C4313

Error Message
'function' : 'format specifier' in format string conflicts with argument %d of type 'type'

There is a conflict between the format specified and the value that you are passing. For example, you passed a 32-bit pointer to %d format specifier, which can hold a 64-bit integer. This warning is only in effect with /Wp64.

Example
The following code sample generates C4313.

```
// C4313.cpp
// compile with: /W1 /Wp64
#include <stdio.h>
int main() {
   int * pI = 0;
   printf("%d", pI);   // C4313
   // try one of the following lines instead
   // printf("%p\n", pI);
   // printf("%Id\n", pI);   // %I64d expects 64-bits of information
}
```

Compiler Warning (level 1) C4318

Error Message
passing constant zero as the length to memset

You called the C run-time library's memset function with a zero character length. For example, memset(p,size,0). Instead, specify memset(p,0,size).

Compiler Warning (level 1) C4319

Error Message
'operator' : zero extending 'type' to 'type' of greater size

The result of the ~ operator is unsigned and then converted to a larger type.

In the following example, ~(a - 1) is evaluated as a 32-bit unsigned long expression and then converted to 64 bits by zero extension. This leads to pointer truncations.

```
// C4319.cpp
// compile with: /W1 /Wp64 /Tc
void* p;
unsigned long a;
typedef unsigned __int64 ULONG_PTR;

int main() {
  p = (void*)((ULONG_PTR)p & ~(a - 1)); // C4319 expected
}
```

Compiler Warning (level 4) C4324

Error Message
'struct_name' : structure was padded due to __declspec(align())

Padding was added at the end of a structure because you specified a __declspec(align) value.

For example, the following code generates C4324:

```
// C4324.cpp
// compile with: /W4
struct __declspec(align(32)) A
{
   char a;
};   // C4324

int main()
{
}
```

Compiler Warning (level 1) C4325

attributes for standard section ' *section* ' ignored

You may not change the attributes of a standard section. For example:

```
#pragma section(".sdata", long)
```

This would overwrite the `.sdata` standard section which uses the **short** data type with the **long** data type.

Standard sections whose attributes you may not change include,

- .data
- .sdata
- .bss
- .sbss
- .text
- .const
- .sconst
- .rdata
- .srdata

Additional sections may be added later.

See Also

Reference

section

Compiler Warning (level 1) C4326

Error Message
return type of 'function' should be 'type1' instead of 'type2'

A function returned a type other than ***type1***. For example, using /Za, main did not return an **int**.

The following sample generates C4326:

```
// C4326.cpp
// compile with: /Za /W1
char main()
{   // C4326 try int main
}
```

Compiler Warning (level 4) C4328

Error Message
'function' : indirection alignment of formal parameter parameter_number (alignment_value_for_parameter) is greater than the actual argument alignment (actual_alignment)

By default on Itanium and CE (other than x86 CE based) platforms, data must be aligned to ensure that the application will run. This means that the address of a variable must be a multiple of the size of the variable's primitive type. (Two-byte Unicode strings must also be aligned.)

To resolve this warning, do not explicitly specify a struct alignment that will result in unaligned structure members. Or, explicitly specify that your function parameters will take unaligned data. For more information, see __unaligned and **#pragma** pack.

Example
C4328 can occur when you pass an unaligned pointer to a function expecting an aligned pointer.

The following sample generates C4328.

```
// C4328.cpp
// compile with: /W4 /Wp64
// processor: x64 IPF
#pragma pack(push)

#pragma pack(1)
struct S {
   int i;
};

struct Other {
   char c;
   S s;
};

// allow compiler to align structs on natural boundaries
#pragma pack(pop)

struct S2 {
   int i;
};

struct Other2 {
   char c;
   S2 s;
};

void f(int * p) {
   p++;
}

void f2(__unaligned int * p) {
   p++;
}

int main() {
   Other o;
   S* pS = &o.s;
   f(&pS->i);   // C4328
   f2(&pS->i);   // OK

   Other2 o2;
   S2* pS2 = &o2.s;
   f(&pS2->i);   // OK
}
```

The following sample is a more subtle cause of C4328. The following sample generates C4328.

```
// C4328_b.cpp
// compile with: /W4 /Wp64 /c
// processor: x64 IPF

// OK
struct T {
   int i;
};

void f2(int* p2);
void g2(T* t) {
   f2(&t->i);
}

// OK
#pragma pack(1)
struct U {
   int i;
};

void f3(__unaligned int* p3);
void g3(U* u) {
   f3(&u->i);
}

// warning
#pragma pack(1)
struct S {
   int i;
};

void f(int* p);
void g(S* s) {
   f(&s->i);   // C4328
   s->i = 0;
}
```

Compiler Warning (level 1) C4329

Error Message
__declspec(align()) is ignored on enum

Use of the align keyword of the __declspec modifier is not allowed on an **enum**. The following sample generates C4329:

```
// C4329.cpp
// compile with: /W1 /LD
enum __declspec(align(256)) TestEnum {   // C4329
   TESTVAL1,
   TESTVAL2,
   TESTVAL3
};
__declspec(align(256)) enum TestEnum1;
```

Compiler Warning (level 1) C4333

Error Message
'operator' : right shift by too large amount, data loss

A right shift operation was too large an amount. All significant bits are shifted out and the result will always be zero.

Example
The following sample generates C4333.

```
// C4333.cpp
// compile with: /c /W1
unsigned shift8 (unsigned char c) {
   return c >> 8;   // C4333

   // try the following line instead
   // return c >> 4;   // OK
}
```

Compiler Warning (level 3) C4334

Error Message
'operator' : result of 32-bit shift implicitly converted to 64 bits (was 64-bit shift intended?)

The result of 32-bit shift was implicitly converted to 64-bits, and the compiler suspects that a 64-bit shift was intended. To resolve this warning, either use 64-bit shift, or explicitly cast the shift result to 64-bit.

Example
The following sample generates C4334.

```
// C4334.cpp
// compile with: /W3 /c
void SetBit(unsigned __int64 *p, int i) {
   *p |= (1 << i);   // C4334
   *p |= (1i64 << i);   // OK
}
```

Compiler Warning C4335

Error Message
Mac file format detected: please convert the source file to either DOS or UNIX format

The line termination character of the first line of a source file is Mac style ('\r') as opposed to UNIX ('\n') or DOS ('\r\n').

This warning is always issued as an errror. See warning pragma for information on how to disable this warning. Also, this warning is only issued once per compiland, so if there are multiple #include directives that specify files in Mac format, C4335 will only be issued once.

One way to generate files in Mac format would be by using the Advanced Save Options (File menu) in Visual Studio.

Example
The following sample generates C4335.

```
// C4335.cpp
// C4335 expected
#include "c4335.h"   // assume both include files are in Mac format
#include "c4335_2.h"
```

Compiler Warning (level 4) C4336

Error Message
import cross-referenced type library 'type_lib1' before importing 'type_lib1'

A type library was referenced with the #import directive. However, the type library contained a reference to another type library that was not referenced with **#import**. This other .tlb file was found by the compiler.

Given two type libraries on disk created from the following two files (compiled with midl.exe):

```
// c4336a.idl
[uuid("f87070ba-c6d9-405c-a8e4-8cd9ca25c12b")]
library c4336aLib
{
   [uuid("f87070ba-c6d9-405c-a8e4-8cd9ca25c12c")]
   enum E_C4336
   {
      one, two, three
   };
};
```

The second type library:

```
// c4336b.idl
[uuid("f87070ba-c6d9-405c-a8e4-8cd9ca25c12d")]
library C4336bLib
{
   importlib ("c4336a.tlb");
   [uuid("f87070ba-c6d9-405c-a8e4-8cd9ca25c12e")]
   struct S_C4336
   {
      enum E_C4336 e;
   };
};
```

The following sample generates C4336:

```
// C4336.cpp
// compile with: /W4 /LD
// #import "C4336a.tlb"
#import "C4336b.tlb"   // C4336, uncomment previous line to resolve
```

Compiler Warning (level 4) C4337

Error Message
cross-referenced type library 'typelib1' in 'typelib2' is being automatically imported

The auto_search attribute of the #import directive caused a type library to be implicitly imported.

Given two type libraries on disk created from the following two files (compiled with midl.exe):

```
// C4337a.idl
[
  uuid(F87070BA-C6D9-405C-A8E4-8CD9CA25C12B)
]
library C4337aLib
{
   [uuid(F87070BA-C6D9-405C-A8E4-8CD9CA25C12C)]
   enum E_C4337a
   {
      one = 0,
      two = 1,
      three = 2
    };
};
```

and then the second .idl file,

```
// C4337b.idl
[
   uuid(F87070BA-C6D9-405C-A8E4-8CD9CA25C12D)
]

library C4337bLib
{
   importlib("c4337a.tlb");

   [uuid(F87070BA-C6D9-405C-A8E4-8CD9CA25C12E)]
   struct S_C4337b
   {
      enum E_C4337a e;
   };
};
```

The following sample generates C4337:

```
// C4337.cpp
// compile with: /W4 /LD
#import "c4337b.tlb" auto_search   // C4337
// explicitly #import all type libraries to resolve
// #import "C4337a.tlb"
// #import "C4337b.tlb"
```

Compiler Warning (level 4) C4339

Error Message
'type' : use of undefined type detected in CLR meta-data - use of this type may lead to a runtime exception

A type was not defined in code that was compiled for the common language runtime. Define the type to avoid a possible runtime exception.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4339:

```
// C4339.cpp
// compile with: /W4 /clr /c
// C4339 expected
#pragma warning(default : 4339)

// Delete the following line to resolve.
class A;

// Uncomment the following line to resolve.
// class A{};


class X {
public:
   X() {}

   virtual A *mf() {
      return 0;
   }
};

X * f() {
   return new X();
}
```

Compiler Warning (level 1) C4340

Error Message
'value' : value wrapped from positive to negative value

The **enum** value is greater than the largest **enum** positive value wrapped around to a negative value.

Compiler Warning (level 3) C4341

Error Message
'value' : signed value is out of range for enum constant

An enumerated constant exceeds the limit for an **int**. The value of the invalid constant is undefined. Constants must resolve to integers between –4,294,967,295 and +4,294,967,295 (signed).

Note that this is a level 3 warning under /Ze and a level 1 warning under /Za.

Example
The following sample generates C4341:

```
// C4341.cpp
// compile with: /WX /W3 /c
enum ELEMENTS {
   Hydrogen = 1,
   Helium,
   Lithium,
   Beryllium,
   Impossibillium = 4294967296   // C4341
   // try the following line instead
   // Impossibillium = 4294967295
};
```

The following sample shows that C4341 is a level 1 warning under /Za

```
// C4341b.cpp
// compile with: /W1 /LD /Za
enum ELEMENTS
{
   Hydrogen = 1,
   Helium,
   Lithium,
   Beryllium,
   Impossibillium = 4294967296  // C4341
   // try the following line instead
   // Impossibillium = 4294967295
};
```

Compiler Warning (level 1) C4342

Error Message
behavior change: 'function' called, but a member operator was called in previous versions

In previous versions of Visual C++, a member was called, but this behavior has been changed and the compiler will find the best match in namespace scope.

If a member operator was found, the compiler would previously not consider any namespace scope operators. If there is a better match at namespace scope, the current compiler will correctly call it, whereas previous compilers wouldn't consider it.

This warning should be disabled after you successfully port your code to the current version. The compiler may give false positives, generating this warning for code where there is no behavior change.

This warning is off by default. For more information, see Compiler Warnings That Are Off by Default.

The following sample generates C4342:

```
// C4342.cpp
// compile with: /EHsc /W1
#include <fstream>
#pragma warning(default: 4342)
using namespace std;
struct X : public ofstream {
   X();
};

X::X() {
   open( "ofs_bug_ev.txt." );
   if ( is_open() ) {
      *this << "Text" << "<-should be text" << endl;   // C4342
      *this << ' ' << "<-should be space symbol" << endl;   // C4342
   }
}

int main() {
   X b;
   b << "Text" << "<-should be text" << endl;
   b << ' ' << "<-should be space symbol" << endl;
}
```

Compiler Warning (level 4) C4343

Error Message
#pragma optimize("g",off) overrides /Og option

This warning, only valid in the Itanium Processor Family (IPF) compiler, reports that a pragma optimize overrode a /Og compiler option.

The following sample generates C4343:

```
// C4343.cpp
// compile with: /Og /W4 /LD
// processor: IPF
#pragma optimize ("g", off)   // C4343
```

Compiler Warning (level 1) C4344

Error Message
behavior change: use of explicit template arguments results in call to 'function'

A call to a function using explicit template arguments calls a different function than it would if explicit arguments had not been specified

Compiler Warning (level 2) C4345

Error Message
behavior change: an object of POD type constructed with an initializer of the form () will be default-initialized

This warning reports a behavior change from the Visual C++ compiler that shipped in Visual Studio .NET when initializing a POD object with (); the compiler will default-initialize the object.

See Summary of Compile-Time Breaking Changes for more information.

The following sample generates C4345:

```
// C4345.cpp
// compile with: /W2
#include <stdio.h>

struct S* gpS;

struct S
{
   // this class has no user-defined default ctor
   void *operator new (size_t size, void*p, int i)
   {
      ((S*)p)->i = i;   // ordinarily, should not initialize
                        // memory contents inside placement new
      return p;
   }
   int i;
};

int main()
{
   S s;
   // Visual C++ .NET 2003 will default-initialize pS->i
   // by assigning the value 0 to pS->i.
   S *pS2 = new (&s, 10) S();   // C4345
   // try the following line instead
   // S *pS2 = new (&s, 10) S;   // not zero initialized
   printf("%d\n", pS2->i);
}
```

Compiler Warning (level 1) C4346

Error Message
'name' : dependent name is not a type

The typename keyword is required if a dependent name is to be treated as a type. This is a breaking change in the Visual C++ .NET 2003 compiler, made in order to conform to the ISO C++ standard.

See Summary of Compile-Time Breaking Changes for more information.

For code that works the same in all versions of Visual C++, add `typename` to the declaration.

The following sample generates C4346:

```
// C4346.cpp
// compile with: /WX /LD
template<class T>
struct C {
   T::X* x;   // C4346
   // try the following line instead
   // typename T::X* x;
};
```

The following samples shows other examples where the **typename** keyword is required:

```
// C4346b.cpp
// compile with: /LD /W1
template<class T>
const typename T::X& f(typename T::Z* p);   // Required in both places

template<class T, int N>
struct L{};

template<class T>
struct M : public L<typename T::Type, T::Value>
{   // required on type argument, not on non-type argument
   typedef typename T::X   Type;
   Type f();   // OK: "Type" is a type-specifer
   typename T::X g();   // typename required
   operator typename T::Z();   // typename required
};
```

and this,

```
// C4346c.cpp
// compile with: /LD /WX
struct Y {
   typedef int Y_t;
};

template<class T>
struct A {
   typedef Y A_t;
};

template<class T>
struct B {
   typedef /*typename*/ A<T>::A_t B_t;   // C4346 typename needed here
   typedef /*typename*/ B_t::Y_t  B_t2;   // typename also needed here
};
```

Compiler Warning (level 4) C4347

Error Message
behavior change: 'function template' is called instead of 'function'

In Visual Studio .NET, if you had a template function and a nontemplate function with the same name as the template function, the compiler incorrectly treated the nontemplate function as a specialization of the template function.

For code that works the same in all versions of Visual C++, add `template<>` above the nontemplate function, making it a real explicit specialization.

For more information, see Summary of Compile-Time Breaking Changes.

This warning is off by default. For more information, see Compiler Warnings That Are Off by Default.

Example
The following sample generates C4347.

```
// C4347.cpp
// compile with: /W4 /EHsc
#pragma warning (default : 4347)

template <typename T>
void f(T t) { T i = t; i = 0; }

void f(int i) { i++; }

// OK
template <typename T>
void f2(T t) { T i = t; i = 0; }

template <>
void f2(int i) { i++; }

int main() {
   f(5);   // regular function call
   f<int>(5);   // C4347 calls implicit instantiation

   f2(5);
   f2<int>(5);
}
```

Compiler Warning (level 1) C4348

Error Message
'type' : redefinition of default parameter : parameter number

A template parameter was redefined.

The following sample generates C4348:

```
// C4348.cpp
// compile with: /LD /W1
template <class T=int> struct A;   // forward declaration

template <class T=int> struct A { };
// C4348, redefinition of default parameter
// try the following line instead
// template <class T> struct A { };
```

Compiler Warning (level 1) C4350

Error Message
behavior change: 'member1' called instead of 'member2'

An rvalue cannot be bound to a non-const reference. In previous versions of Visual C++, it was possible to bind an rvalue to a non-const reference in a direct initialization. This code now gives a warning.

For backward compatibility, it is still possible to bind rvalues to non-const references, but standard conversions are preferred wherever possible.

This warning represents a change of behavior from the Visual C++ .NET 2002 compiler. If enabled, this warning could possibly be given for correct code. For example, it could be given when using the **std::auto_ptr** class template.

If you get this warning, examine your code to see if it depends on binding rvalues to non-const references. Adding a const to the reference or providing a additional const-reference overload may solve the problem.

This warning is off by default. For more information, see Compiler Warnings That Are Off by Default.

The following sample generates C4350:

```
// C4350.cpp
// compile with: /W1
#pragma warning (default : 4350)
class A {};

class B
{
public:
   B(B&){}
   // try the following instead
   // B(const B&){}

   B(A){}
   operator A(){ return A();}
};

B source() { return A(); }

int main()
{
   B ap(source());   // C4350
}
```

Compiler Warning (level 1) C4351

Error Message
new behavior: elements of array 'array' will be default initialized

This warning is for a change of compiler behavior in Visual C++ 2005. When an array is in a constructor's member initialization list, the elements of the array will be default initialized. In previous versions of Visual C++, when an array was in a constructor's member initialization list, the elements of the array may not have been default initialized in some cases.

If the array's element type does not have a constructor, the elements of the array will be initialized with the corresponding zero representation for that type.

C4351 means that you should inspect your code. If you want the compiler's previous behavior, remove the array from the constructor's member initialization list.

If you want the new behavior, which is likely, because the array was explicitly added to the constructor's member initialization list, use the warning pragma to disable the warning. The new behavior should be fine for most users.

One situation where the new behavior can result in unexpected behavior is when placement **new** is used to construct the object that has the array as a member, and the program depends on the contents that the memory (for the elements of the default initialized array) had before the call to placement **new**. In this case, the older compiler would have left the contents of memory unchanged, but the new behavior will cause default initialization of the array elements, overwriting the original contents in memory.

Example
The following sample generates C4351:

```
// C4351.cpp
// compile with: /W1 /LD
#include <new>
#include <stdio.h>

extern "C" int printf_s(const char *, ...);

struct POD
{
    char m_c;
};

struct S
{
    int m_i;
    POD m_default_initialized_arr[10];
    POD m_not_initialized_arr[10];

    S() :
       m_i(), m_default_initialized_arr()
    {
    }   // C4351
};

int main()
{
    S s;
    printf_s("First byte of the default initialized array "
             "= '%c' (%d)\n",
             s.m_default_initialized_arr[0].m_c,
             s.m_default_initialized_arr[0].m_c);
    printf_s("First byte of the array not initialized = '%c' (%d)\n\n",
             s.m_not_initialized_arr[0].m_c,
             s.m_not_initialized_arr[0].m_c);

    s.m_default_initialized_arr[0].m_c = 'a';
    s.m_not_initialized_arr[0].m_c = 'a';
```

```
    s.~S();

    printf_s("First byte of the default initialized array "
             "= '%c' (%d)\n",
             s.m_default_initialized_arr[0].m_c, (int)
             s.m_default_initialized_arr[0].m_c);
    printf_s("First byte of the array not initialized = '%c' (%d)\n\n",
             s.m_not_initialized_arr[0].m_c, (int)
             s.m_not_initialized_arr[0].m_c);

    new (&s) S();

    // If your program depends on s.m_default_initialized_arr[0].m_c
    // still having the value 'a', then the new compiler behavior will
    // cause your program to behave differently

    printf_s("First byte of the default initialized array "
             "= '%c' (%d)\n",
             s.m_default_initialized_arr[0].m_c, (int)
             s.m_default_initialized_arr[0].m_c);
    printf_s("First byte of the array not initialized = '%c' (%d)\n\n",
             s.m_not_initialized_arr[0].m_c, (int)
             s.m_not_initialized_arr[0].m_c);
}
```

Compiler Warning (level 1) C4353

Error Message
nonstandard extension used: constant 0 as function expression. Use '__noop' function intrinsic instead

You cannot use the constant zero (0) as a function expression. For more information, see __noop.

The following sample generates C4353:

```
// C4353.cpp
// compile with: /W1
void MyPrintf(void){};
#define X 0
#if X
   #define DBPRINT MyPrint
#else
   #define DBPRINT 0   // C4353 expected
#endif
int main(){
DBPRINT();
}
```

Compiler Warning (levels 1 and 4) C4355

Error Message
'this' : used in base member initializer list

The **this** pointer is valid only within nonstatic member functions. It cannot be used in the initializer list for a base class.

The base-class constructors and class member constructors are called before **this** constructor. In effect, you've passed a pointer to an unconstructed object to another constructor. If those other constructors access any members or call member functions on this, the result will be undefined. You should not use the **this** pointer until all construction has completed.

This is a level-1 warning under Microsoft extensions (/Ze) and a level-4 warning otherwise.

The following sample generates C4355:

```
// C4355.cpp
// compile with: /W1 /c
#include <tchar.h>

class CDerived;
class CBase {
public:
   CBase(CDerived *derived): m_pDerived(derived) {};
   ~CBase();
   virtual void function() = 0;

   CDerived * m_pDerived;
};

class CDerived : public CBase {
public:
   CDerived() : CBase(this) {};   // C4355 "this" used in derived c'tor
   virtual void function() {};
};

CBase::~CBase() {
   m_pDerived -> function();
}

int main() {
   CDerived myDerived;
}
```

Compiler Warning (level 2) C4356

Error Message
'member' : static data member cannot be initialized via derived class

The initialization of a static data member was ill formed. The compiler accepted the initialization.

This is a breaking change in the Visual C++ .NET 2003 compiler. See Summary of Compile-Time Breaking Changes for more information.

For code that works the same in all versions of Visual C++, initialize the member through the base class.

Use the warning pragma to suppress this warning.

The following sample generates C4356:

```
// C4356.cpp
// compile with: /W2 /EHsc
#include <iostream>

template <class T>
class C {
   static int n;
};

class D : C<int> {};

int D::n = 0; // C4356
// try the following line instead
// int C<int>::n = 0;

class A {
public:
   static int n;
};

class B : public A {};

int B::n = 10;   // C4356
// try the following line instead
// int A::n = 99;

int main() {
   using namespace std;
   cout << B::n << endl;
}
```

Compiler Warning (level 3) C4357

Error Message
param array argument in formal argument list for delegate 'del' ignored when generating 'function'

The **ParamArray** attribute was ignored, and *function* cannot be called with variable arguments.

The following sample generates C4357:

```
// C4357.cpp
// compile with: /clr /W3 /c
using namespace System;
public delegate void f(int i, ... array<Object^>^ varargs);   // C4357

public delegate void g(int i, array<Object^>^ varargs);   // OK
```

Compiler Warning (level 1) C4358

Error Message
'operator': return type of combined delegates is not 'void'; returned value is undefined

Two delegates were combined and the return value is not void. If two delegates with non-void return values are combined, the compiler will not be able to do a proper assignment if the return value of the delegate is used.

The following sample generates C4358:

```
// C4358.cpp
// compile with: /clr /W1
delegate int D();
delegate void E();

ref class X {
   int i;
public:
   X(int ii) : i(ii) {}
   int f() {
      return i;
   }
};

ref class Y {
   int i;
public:
   Y() {}
   void g() {}
};

int main() {
   D^ d = gcnew D(gcnew X(1), &X::f);
   D^ d2 = gcnew D(gcnew X(2), &X::f);

   d += d2;   // C4358
   int j = d();   // return value indeterminate

   E^ e = gcnew E(gcnew Y, &Y::g);
   E^ e2 = gcnew E(gcnew Y, &Y::g);
   e += e2;   // OK
}
```

Compiler Warning (level 3) C4359

Error Message

'type': actual alignment (8) is greater than the value specified in __declspec(align())

The alignment specified for a type is less than the alignment of the type of one of its data members. For more information, see align (C++).

Example

The following sample generates C4359.

```
// C4359.cpp
// compile with: /W3 /c
struct __declspec(align(8)) C8 { __int64 i; };
struct __declspec(align(4)) C4  { C8 m8; };   // C4359
struct __declspec(align(8)) C8_b  { C8 m8; };   // OK
struct __declspec(align(16)) C16  { C8 m8; };   // OK
```

Compiler Warning (level 1) C4364

Error Message
#using for assembly 'file' previously seen at location(line_number) without as_friend attribute; as_friend not applied

A **#using** directive was repeated for a given metadata file, but the **as_friend** qualifier was not used in the first occurrence; the compiler will ignore the second **as_friend**.

For more information, see Friend Assemblies.

Example
The following sample creates a component.

```
// C4364.cpp
// compile with: /clr /LD
ref class A {};
```

The following sample generates C4364.

```
// C4364_b.cpp
// compile with: /clr /W1 /c
#using " C4364.dll"
#using " C4364.dll" as_friend   // C4364
```

Compiler Warning (level 4) C4365

Error Message
'action' : conversion from 'type_1' to 'type_2', signed/unsigned mismatch

For example, you tried to convert an unsigned value to a signed value.

C4365 is off by default. For more information, see Compiler Warnings That Are Off by Default.

Example
The following sample generates C4365.

```
// C4365.cpp
// compile with: /W4
#pragma warning(default:4365)

int f(int) { return 0; }
void Test(size_t i) {}

int main() {
   unsigned int n = 10;
   int o = 10;
   n++;
   f(n);   // C4365
   f(o);   // OK

   Test( -19 );   // C4365
}
```

Compiler Warning (level 4) C4366

Error Message
The result of the unary 'operator' operator may be unaligned

If a structure member could ever be unaligned because of packing, the compiler will warn when that member's address is assigned to an aligned pointer. By default, all pointers are aligned.

To resolve C4366, either change the alignment of the structure or declare the pointer with the __unaligned keyword.

For more information, see __unaligned and pack.

Example
The following sample generates C4366.

```
// C4366.cpp
// compile with: /W4 /c
// processor: IPF x64
#pragma pack(1)
struct X {
   short s1;
   int s2;
};

int main() {
   X x;
   short * ps1 = &x.s1;   // OK
   int * ps2 = &x.s2;   // C4366
}
```

Compiler Warning C4368

Error Message
cannot define 'member' as a member of managed 'type': mixed types are not supported

You cannot embed a native data member in a CLR type.

You can, however, declare a pointer to a native type and control its lifetime in the constructor and destructor and finalizer of your managed class (see Destructors and Finalizers in Visual C++ for more information).

This warning is always issued as an error. Use the warning pragma to disable C4368.

Example
The following sample generates C4368.

```
// C4368.cpp
// compile with: /clr /c
struct N {};
ref struct O {};
ref struct R {
    R() : m_p( new N ) {}
    ~R() { delete m_p; }

   property N prop;   // C4368
   int i[10];   // C4368

   property O ^ prop2;   // OK
   N * m_p;   // OK
};
```

Compiler Warning (level 1) C4369

Error Message
'enumerator' : enumerator value 'value' cannot be represented as 'type', value is 'new_value'

An enumerator was calculated to be greater than the greatest value for the specified underlying type. This caused an overflow and the compiler wrapped the enumerator value to the lowest possible value for the type.

Example
The following sample generates C4369.

```
// C4369.cpp
// compile with: /W1
int main() {
   enum Color: char { red = 0x7e, green, blue };   // C4369
   enum Color2: char { red2 = 0x7d, green2, blue2};   // OK
}
```

Compiler Warning (level 1) C4374

Error Message
'function1': interface method will not be implemented by non-virtual method 'function2'

The compiler expected to find the virtual keyword on a method definition.

The following sample generates C4374:

```
// C4374.cpp
// compile with: /clr /W1 /c /WX
public interface class I {
   void f();
};

public ref struct B {
   void f() {
      System::Console::WriteLine("B::f()");
   }
};

public ref struct C {
   virtual void f() {
      System::Console::WriteLine("C::f()");
   }
};

public ref struct D : B, I {};   // C4374
public ref struct E : C, I {};   // OK
```

Compiler Warning (level 1) C4375

Error Message
non-public method 'method2' does not override 'method1'

A type that implements another type defined an override method, but the override was not public. Therefore, the method does not override the base type method.

Compiler Warning (level 1) C4376

Error Message
access specifier 'old_specifier:' is no longer supported: please use 'new_specifier:' instead

For more information on specifying type and member accessibility in metadata, see Type and Member Visibility.

Example
The following sample generates C4376.

```
// C4376.cpp
// compile with: /clr /W1 /c
public ref class G {
public public:   // C4376
   void m2();
};

public ref class H {
public:   // OK
   void m2();
};
```

Compiler Warning (level 1) C4377

Error Message
native types are private by default; -d1PrivateNativeTypes is deprecated

In previous releases, native types in assemblies were public by default, and an internal, undocumented compiler option (**/d1PrivateNativeTypes**) was used to make them private.

All types, native and CLR, are now private by default in an assembly, so **/d1PrivateNativeTypes** is no longer needed.

Example
The following sample generates C4377.

```
// C4377.cpp
// compile with: /clr /d1PrivateNativeTypes /W1
// C4377 warning expected
int main() {}
```

Compiler Warning (level 1) C4378

Error Message
Must obtain function pointers to run initializers; consider System::ModuleHandle::ResolveMethodHandle

Under **/clr**, initializer symbols contain function tokens, not functions pointers. You need to convert tokens to pointers using ResolveMethodHandle.

Example
The following sample generates C4378.

```
// C4378.cpp
// compile with: /W1 /clr /c
typedef void (__cdecl *PF)(void);
int cxpf = 0;   // number of destructors to call
PF pfx[200];   // ptrs to those dtors, watch for overflow

int myexit (PF pf) {
   pfx[cxpf++] = pf;
   return 0;
}

struct A {
   A() {}
   ~A() {}
};

A aaaa;

#pragma data_seg(".mine$a")
PF InitSegStart = (PF)1;
#pragma data_seg(".mine$z")
PF InitSegEnd = (PF)1;
#pragma data_seg()

void InitializeObjects () {
   PF *x = &InitSegStart;
   for (++x ; x < &InitSegEnd ; ++x)
      if (*x)
         (*x)();
}

#pragma init_seg(".mine$m",myexit)   // C4378
A bbbb;   // crash

int main () {
   InitializeObjects();
}
```

The following sample shows how to resolve C4378.

```
// C4378_b.cpp
// compile with: /clr
#pragma warning(disable:4378)
using namespace System;
typedef void (__cdecl *PF)(void);
typedef void (__clrcall * CLRPF)(void);

int cxpf = 0;  // number of destructors we need to call
PF pfx[200];   // ptrs to those dtors. Watch out for overflow!

ref class TypeClassHolder {
public:
   static TypeClassHolder ^typeClass = gcnew TypeClassHolder();
```

```
};

CLRPF FuncTokenToFuncPtr(PF tknFunc) {
   ModuleHandle type =
      Type::GetTypeFromHandle(Type::GetTypeHandle(TypeClassHolder::typeClass))->Module->Mod
uleHandle;
   return (CLRPF)type.ResolveMethodHandle((int)(size_t)(tknFunc)).GetFunctionPointer().ToPo
inter();
}

int myexit (PF pf) {
   pfx[cxpf++] = pf;
   return 0;
}

struct A {
   A() {}
   ~A() {}
};

A aaaa;

#pragma data_seg(".mine$a")
PF InitSegStart = (PF)1;
#pragma data_seg(".mine$z")
PF InitSegEnd = (PF)1;
#pragma data_seg()

void InitializeObjects () {
   PF *x = &InitSegStart;
   for (++x ; x < &InitSegEnd ; ++x)
      if(*x) {
         CLRPF realppfunc;
         realppfunc = FuncTokenToFuncPtr(*x);
         (realppfunc)();
      }
}

#pragma init_seg(".mine$m",myexit)
A bbbb; // constructor call succeeds

int main () {
   InitializeObjects();
}
```

Compiler Warning (level 1) C4379

Error Message
Version version of the common language runtime is not supported by this compiler. Using this version may cause unexpected results.

You have a previous version of the common language runtime on your machine, but not the current version. To resolve C4379, install the version of the common language runtime that shipped with your compiler.

Compiler Warning (level 1) C4381

Error Message
'function1': interface method will not be implemented by non-public method 'function2'

A class must implement all function in an interface. A class can satisfy this condition if one of its base classes implements the function. However, the function must be implemented as a public function.

Compiler Warning (level 1) C4382

Error Message
throwing 'type' : a type with __clrcall destructor or copy constructor can only be caught in /clr:pure module

When compiled with **/clr** (not **/clr:pure**), exception handling expects the member functions in a native type to be __cdecl and not __clrcall. Native types with member functions using **__clrcall** calling convention cannot be caught in a module compiled with **/clr**.

If the exception will be caught in a module compiled with **/clr:pure**, you can ignore this warning.

For more information, see /clr (Common Language Runtime Compilation).

Example
The following sample generates C4382.

```
// C4382.cpp
// compile with: /clr /W1 /c
struct S {
   __clrcall ~S() {}
};

struct T {
   ~T() {}
};

int main() {
   S s;
   throw s;   // C4382

   S * ps = &s;
   throw ps;   // OK

   T t;
   throw t;   // OK
}
```

Compiler Warning (level 1) C4383

Error Message
'instance_dereference_operator' : the meaning of dereferencing a handle can change, when a user-defined 'operator' operator exists; write the operator as a static function to be explicit about the operand

When you add a user-defined instance override of the dereference operator in a managed type, you potentially override the ability of the type's dereference operator to return the handle's object. Consider writing a static, user-defined dereference operator.

For more information, see ^ (Handle to Object on Managed Heap) and % (Tracking Reference).

Also, an instance operator is not available to other language compilers via referenced metadata. For more information, see User-Defined Operators.

Example
The following sample generates C4383.

```
// C4383.cpp
// compile with: /clr /W1

ref struct S {
   int operator*() { return 0; }   // C4383
};

ref struct T {
   static int operator*(T%) { return 0; }
};

int main() {
   S s;
   S^ pS = %s;

   T t;
   T^ pT = %t;
   T% rT = *pT;
}
```

Compiler Warning (level 1) C4384

Error Message
#pragma 'make_public' should only be used at global scope

The make_public pragma was applied incorrectly.

Example
The following sample generates C4384.

```
// C4384.cpp
// compile with: /c /W1
namespace n {
   #pragma make_public(N::C)   // C4384
   namespace N {
      class C {};
   }
}
```

Compiler Warning (level 4) C4389

Error Message
'operator' : signed/unsigned mismatch

An operation involved signed and unsigned variables. This could result in a loss of data.

The following sample generates C4389:

```
// C4389.cpp
// compile with: /W4
#pragma warning(default: 4389)

int main()
{
   int a = 9;
   unsigned int b = 10;
   if (a == b)   // C4389
      return 0;
   else
      return 0;
};
```

Compiler Warning (level 3) C4390

Error Message
';' : empty controlled statement found; is this the intent?

A semicolon was found after a control statement that contains no instructions.

If you get C4390 because of a macro, you should use the warning pragma to disable C4390 in the module containing the macro.

The following sample generates C4390:

```
// C4390.cpp
// compile with: /W3
int main() {
   int i = 0;
   if (i);   // C4390
      i++;
}
```

Compiler Warning (level 1) C4391

Error Message
'signature' : incorrect return type for intrinsic function, expected 'type'

A function declaration for a compiler intrinsic had the wrong return type. The resulting image may not run correctly.

To fix this warning, either correct the declaration or delete the declaration and simply #include the appropriate header file.

The following sample generates C4391:

```
// C4391.cpp
// compile with: /W1
// processor: x86
// uncomment the following line and delete the line that
// generated the warning to resolve
// #include "xmmintrin.h"

#ifdef  __cplusplus
extern "C" {
#endif

extern void _mm_load_ss(float *p);   // C4391

#ifdef  __cplusplus
}
#endif

int main()
{
}
```

Compiler Warning (level 1) C4392

Error Message
'signature' : incorrect number of arguments for intrinsic function, expected 'number' arguments

A function declaration for a compiler intrinsic had the wrong number of arguments. The resulting image may not run correctly.

To fix this warning, either correct the declaration or delete the declaration and simply #include the appropriate header file.

The following sample generates C4392:

```
// C4392.cpp
// compile with: /W1
// processor: x86
// uncomment the following line and delete the line that
// generated the warning to resolve
// #include "xmmintrin.h"

#ifdef  __cplusplus
extern "C" {
#endif

extern void _mm_stream_pd(double *dp);   // C4392

#ifdef  __cplusplus
}
#endif

int main()
{
}
```

Compiler Warning (level 1) C4393

Error Message
'var' : const has no effect on literal data member; ignored

A literal data member was also specified as const. Since a literal data member implies const, you do not need to add const to the declaration.

The following sample generates C4393:

```
// C4393.cpp
// compile with: /clr /W1 /c
ref struct Y1 {
   literal const int staticConst = 10;   // C4393
   literal int staticConst2 = 10;   // OK
};
```

Compiler Warning C4394

Error Message
'function' : per-appdomain symbol should not be marked with __declspec(dllexport)

A function marked with the appdomain **__declspec** modifier is compiled to MSIL (not to native), and export tables (export **__declspec** modifier) are not supported for managed functions.

You can declare a managed function to have public accessibility, see Type and Member Visibility for more information.

C4394 is always issued as an error. You can turn off this warning with the **#pragma warning** or **/wd**; see warning or /w, /Wn, /WX, /Wall, /wln, /wdn, /wen, /won (Warning Level) for more information.

Example
The following sample generates C4394.

```
// C4394.cpp
// compile with: /clr /c
__declspec(dllexport) __declspec(appdomain) int g1 = 0;   // C4394
__declspec(dllexport) int g2 = 0;   // OK
```

Compiler Warning (level 1) C4395

Error Message
'function' : member function will be invoked on a copy of the initonly data member 'member'

A member function was called on an initonly data member. C4395 warns that the **initonly** data member cannot be modified by the function.

The following sample generates C4395:

```
// C4395.cpp
// compile with: /W1 /clr
public value class V {
public:
   V(int data) : m_data(data) {}

   void Mutate() {
      System::Console::WriteLine("Enter Mutate: m_data = {0}", m_data);
      m_data *= 2;
      System::Console::WriteLine("Leave Mutate: m_data = {0}", m_data);
   }

   int m_data;
};

public ref class R {
public:
   static void f() {
      System::Console::WriteLine("v.m_data = {0}", v.m_data);
      v.Mutate();   // C4395
      System::Console::WriteLine("v.m_data = {0}", v.m_data);
   }

private:
   initonly static V v = V(4);
};

int main() {
   R::f();
}
```

Compiler Warning (level 1) C4397

Error Message
DefaultCharSetAttribute is ignored

DefaultCharSetAttribute is ignored by the Visual C++ compiler. To specify a character set for the DLL, use the CharSet option of DllImport. For more information, see Using C++ Interop (Implicit PInvoke).

Example
The following sample generates C4397.

```
// C4397.cpp
// compile with: /W1 /c /clr
using namespace System;
using namespace System::Runtime::InteropServices;

[module:DefaultCharSetAttribute(CharSet::Unicode)];   // C4397

[DllImport("kernel32", EntryPoint="CloseHandle", CharSet=CharSet::Unicode)]   // OK
extern "C" bool ImportDefault(IntPtr hObject);

public ref class MySettingVC {
public:
   void method() {
      ImportDefault(IntPtr::Zero);
   }
};

[StructLayout(LayoutKind::Explicit)]
public ref struct StructDefault1{};

public ref class ClassDefault1{};
```

Compiler Warning (level 3) C4398

Error Message
'variable' : per-process global object might not work correctly with multiple appdomains; consider using __declspec(appdomain)

A virtual function with __clrcall calling convention in a native type causes the creation of a per application domain vtable. Such a variable may not correct correctly when used in multiple application domains.

You can resolve this warning by compiling with **/clr:pure**, which makes global variables per appdomain by default, or by explicitly marking the variable **__declspec(appdomain)**.

For more information, see appdomain and Application Domains and Visual C++.

Example
The following sample generates C4398.

```
// C4398.cpp
// compile with: /clr /W3 /c
struct S {
   virtual void f( System::String ^ );   // String^ parameter makes function __clrcall
};

S glob_s;   // C4398
__declspec(appdomain) S glob_s2;   // OK
```

Compiler Warning (level 1) C4399

Error Message
'symbol' : per-process symbol should not be marked with __declspec(dllimport) when compiled with /clr:pure

Data from a native image or an image with native and CLR constructs can not be imported into a pure image. To resolve this warning, compile with **/clr** (not **/clr:pure**) or delete **__declspec(dllimport)**.

Example
The following sample generates C4399.

```
// C4399.cpp
// compile with: /clr:pure /doc /W1 /c
__declspec(dllimport) __declspec(process) extern const int i;   // C4399
```

Compiler Warning (level 4) C4400

Error Message

'type' : const/volatile qualifiers on this type are not supported

The const (C++)and volatile (C++)qualifiers will not work with variables of common language runtime types.

Example

The following sample generates C4400.

```
// C4400.cpp
// compile with: /clr /W4
// C4401 expected
using namespace System;
#pragma warning (disable : 4101)
int main() {
   const String^ str;   // C4400
   volatile String^ str2;   // C4400
}
```

Compiler Warning (level 1) C4401

Error Message
'bitfield' : member is bit field

Inline assembly code tries to access a bit-field member. Inline assembly cannot access bit-field members, so the last packing boundary before the bit-field member is used.

To avoid this warning, cast the bit field to an appropriate type before making the reference in inline assembly code. The following sample generates C4401:

```
// C4401.cpp
// compile with: /W1
// processor: x86
typedef struct bitfield {
   signed bit : 1;
} mybitfield;

int main() {
   mybitfield bf;
   bf.bit = 0;
   __asm {
      mov bf.bit,0;   // C4401
   }

   /* use the following __asm block to resolve the warning
   int i = (int)bf.bit;
   __asm {
      mov i,0;
   }
   */
}
```

Compiler Warning (level 1) C4402

Error Message
must use PTR operator

A type is used on an operand without a PTR operator when referring to or casting to a type in inline assembly code.

The compiler assumes the PTR operator.

Compiler Warning (level 1) C4403

Error Message
illegal PTR operator

A PTR operator is used inappropriately in inline assembler code.

The compiler ignored the PTR operator.

Compiler Warning (level 3) C4404

Error Message
period on directive ignored

The optional period preceding the directive is ignored.

Compiler Warning (level 1) C4405

Error Message
'identifier' : identifier is reserved word

A word reserved for inline assembly is used as a variable name. This may cause unpredictable results. To fix this warning, avoid naming variables with words reserved for inline assembly. The following sample generates C4405:

```
// C4405.cpp
// compile with: /W1
// processor: x86
void func1() {
   int mov = 0, i = 0;
   _asm {
      mov mov, 0;   // C4405
      // instead, try ..
      // mov i, 0;
   }
}

int main() {
}
```

Compiler Warning (level 1) C4406

Error Message
operand on directive ignored

The directive does not take any operands, but an operand was specified.

The compiler ignored the given operand or operands.

Compiler Warning (level 1) C4407

Error Message
cast between different pointer to member representations, compiler may generate incorrect code

An incorrect cast was detected.

C4407 can be generated because of compiler conformance work that was done in Visual C++ 2005; pointer-to-members now require qualified name and &. For more information, see Breaking Breaking Changes in the Visual C++ 2005 Compiler.

C4407 can occur if you cast between a multiple inheritance pointer-to-member to a single inheritance pointer-to-member. Sometimes this can work, but sometimes it can't because the single inheritance pointer-to-member representation doesn't hold sufficient information. Compiling with the **/vmm** might help (see /vmm, /vms, /vmv (General Purpose Representation) for more information). You can also try rearranging your base classes; the compiler is detecting a loss of information in the conversion because a base class is at a non-zero offset from the derived.

The following sample generates C4407:

```
// C4407.cpp
// compile with: /W1 /c
struct C1 {};
struct C2 {};
struct C3 : C1, C2 {};

typedef void(C3::*PMF_C3)();
typedef void(C2::*PMF_C2)();

PMF_C2 f1(PMF_C3 pmf) {
   return (PMF_C2)pmf;   // C4407, change type of cast,
   // or reverse base class inheritance of C3 (i.e. : C2, C1)
}
```

Compiler Warning (level 4) C4408

Error Message
anonymous union did not declare any data members

An anonymous union must have at least one data member.

The following sample generates C4408:

```
// C4408.cpp
// compile with: /W4 /LD
static union
{
   // int i;
};
// a named union can have no data members
// } MyUnion;
```

Compiler Warning (level 1) C4409

Error Message
illegal instruction size

The instruction did not have a form with the specified size. The smallest legal size was used.

Compiler Warning (level 1) C4410

Error Message
illegal size for operand

One of the operands on the instruction had an incorrect size. The smallest legal size for the operand was used.

Compiler Warning (level 1) C4411

Error Message
'identifier' : symbol resolves to displacement register

The identifier is a local symbol that resolves to a displacement register and therefore may be used on an operand with another symbol.

Compiler Warning (level 2) C4412

Error Message
'function' : function signature contains type 'type'; C++ objects are unsafe to pass between pure code and mixed or native.

The compiler detected a potentially unsafe situation that could result in a runtime error: a call is being made from a **/clr:pure** compiland to a function that was imported via dllimport and the function signature contains an unsafe type. A type is unsafe if it contains a member function or has a data member that is an unsafe type or an indirection to an unsafe type.

This is unsafe because of the difference in the default calling conventions between pure and native code (or mixed native and managed). When importing (via **dllimport**) a function into a **/clr:pure** compiland, ensure that the declarations of each type in the signature are identical to those in the compiland that exports the function (being especially careful about differences in implicit calling conventions).

A virtual member function is especially prone to give unexpected results. However, even a non-virtual function should be tested to ensure that you get the correct results. If you are sure that you are getting the correct results, you can ignore this warning.

For more information on **/clr:pure**, see How to: Migrate to /clr:pure.

C4412 is off by default. See Compiler Warnings That Are Off by Default and dllexport, dllimport for more information.

To resolve this warning, remove all functions from the type.

Example
The following sample generates C4412.

```
// C4412.cpp
// compile with: /c /W2 /clr:pure
#pragma warning (default : 4412)

struct Unsafe {
   virtual void __cdecl Test();
};

struct Safe {
   int i;
};

__declspec(dllimport) Unsafe * __cdecl func();
__declspec(dllimport) Safe * __cdecl func2();

int main() {
   Unsafe *pUnsafe = func();   // C4412
   // pUnsafe->Test();

   Safe *pSafe = func2();   // OK
}
```

The following sample is a header file that declares two types. The `Unsafe` type is unsafe because it has a member function.

```
// C4412.h
struct Unsafe {
   // will be __clrcall if #included in pure compilation
   // defaults to __cdecl in native or mixed mode compilation
   virtual void Test(int * pi);

   // try the following line instead
   // virtual void __cdecl Test(int * pi);
};

struct Safe {
   int i;
};
```

This sample exports functions with the types defined in the header file.

```
// C4412_2.cpp
// compile with: /LD
#include "C4412.h"

void Unsafe::Test(int * pi) {
   *pi++;
}

__declspec(dllexport) Unsafe * __cdecl func() { return new Unsafe; }
__declspec(dllexport) Safe * __cdecl func2() { return new Safe; }
```

The default calling convention in a **/clr:pure** compilation is different from a native compilation. When C4412.h is included, `Test` will be default to **__clrcall**. If you compile and run this program (do not use **/c**), the program will throw an exception.

The following sample generates C4412.

```
// C4412_3.cpp
// compile with: /W2 /clr:pure /c /link C4412_2.lib
#pragma warning (default : 4412)
#include "C4412.h"

__declspec(dllimport) Unsafe * __cdecl func();
__declspec(dllimport) Safe * __cdecl func2();

int main() {
   int n = 7;
   Unsafe *pUnsafe = func();   // C4412
   pUnsafe->Test(&n);

   Safe *pSafe = func2();   // OK
}
```

Compiler Warning (level 3) C4414

Error Message
'function' : short jump to function converted to near

Short jumps generate compact instruction which branches to an address within a limited range from the instruction. The instruction includes a short offset that represents the distance between the jump and the target address, the function definition. During linking a function may be moved or subject to link-time optimizations that cause the function to be moved out of the range reachable from a short offset. The compiler must generate a special record for the jump, which requires the jmp instruction to be either NEAR or FAR. The compiler made the conversion.

For example, the following code generates C4414:

```
// C4414.cpp
// compile with: /W3 /c
// processor: x86
int DoParityEven();
int DoParityOdd();
unsigned char global;
int __declspec(naked) DoParityEither()
{
   __asm
   {
      test global,0
      jpe SHORT DoParityEven  // C4414
      jmp SHORT DoParityOdd   // C4414
   }
}
```

Compiler Warning (level 1) C4420

Error Message
'operator' : operator not available, using 'operator' instead; run-time checking may be compromised

This warning is generated when you use the /RTCv (vector new/delete checking) and when no vector form is found. In this case, the non-vector form is used.

In order for /RTCv to work correctly, the compiler should always call the vector form of new/delete if the vector syntax was used.

Compiler Warning (level 4) C4428

Error Message
universal-character-name encountered in source

The compiler issues C4428 when it detects at least one universal character name in a source code file. To fix this warning, use the Unicode equivalent of the universal character name.

This warning is only issued once per compiland.

The following sample generates C4428:

```
// C4428.cpp
// compile with: /W4 /c
int \u20ac = 0;  // C4428 universal character name
/// The following line is the Unicode equivalent of \u20ac:
// int  = 0;
```

Compiler Warning (level 4) C4429

Error Message
possible incomplete or improperly formed universal-character-name

The compiler detected a character sequence that may be a badly formed universal character name. A universal character name is `\u` followed by four hex digits, or `\U` followed by eight hex digits.

The following sample generates C4429:

```
// C4429.cpp
// compile with: /W4 /WX
int \ug0e4 = 0;   // C4429
// Try the following line instead:
// int \u00e4 = 0;   // OK, a well-formed universal character name.
```

Compiler Warning C4430

Error Message
missing type specifier - int assumed. Note: C++ does not support default-int

This error can be generated as a result of compiler conformance work that was done for Visual C++ 2005: all declarations must now explicitly specify the type; int is no longer assumed. See Breaking Changes in the Visual C++ 2005 Compiler for more information.

C4430 is always issued as an error. You can turn off this warning with the **#pragma warning** or **/wd**; see warning or /w, /Wn, /WX, /Wall, /wln, /wdn, /wen, /won (Warning Level) for more information.

Example
The following sample generates C4430.

```
// C4430.cpp
// compile with: /c
struct CMyClass {
   CUndeclared m_myClass;  // C4430
   int m_myClass;  // OK
};

typedef struct {
   POINT();   // C4430
   // try the following line instead
   // int POINT();
   unsigned x;
   unsigned y;
} POINT;
```

Compiler Warning (level 4) C4431

Error Message
missing type specifier - int assumed. Note: C no longer supports default-int

This error can be generated as a result of compiler conformance work that was done for Visual C++ 2005: Visual C++ no longer creates untyped identifiers as int by default. The type of an identifier must be specified explicitly. See Breaking Changes in the Visual C++ 2005 Compiler for more information.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

Example
The following sample generates C4431.

```
// C4431.c
// compile with: /c /W4
#pragma warning(default:4431)
i;   // C4431
int i;   // OK
```

Compiler Warning (level 4) C4432

Error Message
a destructor in a ref-class must have public accessibility; changing to public access

The compiler made a destructor public. To resolve this warning, explicitly put the destructor in a public section of the class.

For more information, see Destructors and Finalizers in Visual C++.

Compiler Warning (level 4) C4433

Error Message
a finalizer must have protected accessibility; changing to protected access

C4433 indicates that the compiler changed the accessibility of a finalizer. Finalizers must have protected accessibility. For more information, see Destructors and Finalizers in Visual C++.

Compiler Warning (level 4) C4434

Error Message
a class constructor must have private accessibility; changing to private access

C4434 indicates that the compiler changed the accessibility of a static constructor. Static constructors must have private accessibility, as they are only meant to be called by the common language runtime. For more information, see How to: Define Static Constructors in a Class or Struct.

Example

The following sample generates C4434.

```
// C4434.cpp
// compile with: /W4 /c /clr
public ref struct R {
   static R(){}   // C4434
};
```

Compiler Warning C4439

Error Message

'function' : function definition with a managed type in the signature must have a __clrcall calling convention

The compiler implicitly replaced a calling convention with __clrcall. To resolve this warning, remove the **__cdecl** or **__stdcall** calling convention.

C4439 is always issued as an error. You can turn off this warning with the **#pragma warning** or **/wd**; see warning or /w, /Wn, /WX, /Wall, /wln, /wdn, /wen, /won (Warning Level) for more information.

Example

The following sample generates C4439.

```
// C4439.cpp
// compile with: /clr
void __stdcall f( System::String^ arg ) {}   // C4439
void __clrcall f2( System::String^ arg ) {}   // OK
void f3( System::String^ arg ) {}   // OK
```

Compiler Warning (level 1) C4440

Error Message
calling convention redefinition from 'calling_convention1' to 'calling_convention2' ignored

An attempt to change the calling convention was ignored.

The following sample generates C4440:

```
// C4440.cpp
// compile with: /W1 /LD /clr
typedef void __clrcall F();
typedef F __cdecl *PFV;   // C4440
```

Compiler Warning (level 1) C4441

Error Message
calling convention of 'cc1' ignored; 'cc2' used instead

Member functions in managed user-defined types and global function generics must use the __clrcall calling convention. The compiler used **__clrcall**.

Example
The following sample generates C4441.

```
// C4441.cpp
// compile with: /clr /W1 /c
generic <class ItemType>
void __cdecl Test(ItemType item) {}   // C4441
// try the following line instead
// void Test(ItemType item) {}

ref struct MyStruct {
   void __cdecl Test(){}   // C4441
   void Test2(){}   // OK
};
```

Compiler Warning (level 1) C4445

Error Message
'function' : in a managed type a virtual method cannot be private

If a virtual function is private, it cannot be accessed by a derived type.

Compiler Warning (level 4) C4460

Error Message
CLR operator 'operator', has parameter passed by reference. CLR operator 'operator' has different semantics from C++ operator 'operator', did you intend to pass by value?

You passed a value by reference to a user-defined CLR operator. If the value is changed inside the function, note that the value returned after the function call will be assigned the return value of the function. In standard C++, the changed value is reflected after the function call.

Example
The following sample generates C4460.

```
// C4460.cpp
// compile with: /W4 /clr
#include <stdio.h>

public value struct V {
   static V operator ++(V& me) {   // C4460
   // try the following line instead
   // static V operator ++(V me) {

      printf_s(__FUNCSIG__ " called\n");
      V tmp = me;
      me.m_i++;
      return tmp;
   }
   int m_i;
};

int main() {
   V v;
   v.m_i = 0;

   printf_s("%d\n", v.m_i);   // Should print 0
   v++;   // Translates to "v = V::operator ++(v)"
   printf_s("%d\n", v.m_i);   // will print 0, hence the warning
}
```

Compiler Warning (level 1) C4461

Error Message
'type' : this class has a finalizer 'finalizer' but no destructor 'dtor'

The presence of a finalizer in a type implies resources to delete. Unless a finalizer is explicitly called from the type's destructor, the common language runtime determines when to run the finalizer, after your object goes out of scope.

If you define a destructor in the type and explicitly call the finalizer from the destructor, you can deterministically run your finalizer.

For more information, see Destructors and Finalizers in Visual C++.

Example
The following sample generates C4461.

```
// C4461.cpp
// compile with: /W1 /clr /c
ref class A {
protected:
   !A() {}   // C4461
};

// OK
ref struct B {
   ~B() {
      B::!B();
   }

   !B() {}
};
```

Compiler Warning (level 1) C4470

Error Message
floating-point control pragmas ignored under /clr

The float-control pragmas:

- fenv_access
- float_control
- fp_contract

have no effect under /clr.

The following sample generates C4470:

```
// C4470.cpp
// compile with: /clr /W1 /LD
#pragma float_control(except, on)   // C4470
```

Compiler Warning (level 4) C4480

Error Message
nonstandard extension used: specifying underlying type for enum 'enum'

An extension to the language under /clr was used without /clr. You can disable C4480 with the warning pragma.

For more information, see /clr (Common Language Runtime Compilation), /Za, /Ze (Disable Language Extensions), enum class, and C++ Enumeration Declarations.

Example
The following sample generates C4480.

```
// C4480.cpp
// compile with: /W4 /c
enum E : char {};   // C4480
enum F {};
```

Compiler Warning (level 4) C4481

Error Message
nonstandard extension used: override specifier 'keyword'

A keyword was used that is not in the C++ standard, for example, one of the override specifiers that also works under /clr. For more information, see,

- /clr (Common Language Runtime Compilation)
- Override Specifiers

Example
The following sample generates C4481.

```
// C4481.cpp
// compile with: /W4 /c
class B {
   virtual void f(unsigned);
};

class C : B {
   void f(unsigned) override;   // C4481
   void f2(unsigned);
};
```

Compiler Warning (level 1) C4482

Error Message
nonstandard extension used: enum 'enum' used in qualified name

When you refer to an enum inside a type, you do not need to specify the name of the enum.

For more information on CLR enums, see enum class. For more information about native enums, see C++ Enumeration Declarations.

Example
The following sample generates C4482.

```
// C4482.cpp
// compile with: /c /W1
struct S {
   enum E { a };
};

int i = S::E::a;   // C4482
int j = S::a;   // OK
```

Compiler Warning C4484

Error Message
'override_function' : matches base ref class method 'base_class_function', but is not marked 'virtual', 'new' or 'override'; 'new' (and not 'virtual') is assumed

When compiling with **/clr**, the compiler will not implicitly override a base class function, which means the function will get a new slot in the vtable. To resolve, explicitly specify whether a function is an override.

For more information, see:

- /clr (Common Language Runtime Compilation)
- override
- new (new slot in vtable)

C4484 is always issued as an error. Use the warning pragma to suppress C4484.

Example
The following sample generates C4484.

```
// C4484.cpp
// compile with: /clr
ref struct A {
   virtual void Test() {}
};

ref struct B : A {
   void Test() {}   // C4484
};

// OK
ref struct C {
   virtual void Test() {}
   virtual void Test2() {}
};

ref struct D : C {
   virtual void Test() new {}
   virtual void Test2() override {}
};
```

Compiler Warning C4485

Error Message
'override_function' : matches base ref class method 'base_class_function ', but is not marked 'new' or 'override'; 'new' (and 'virtual') is assumed

An accessor overrides, with or without the **virtual** keyword, a base class accessor function, but the **override** or **new** specifier was not part of the overriding function signature. Add the **new** or **override** specifier to resolve this warning.

See override and new (new slot in vtable) for more information.

C4485 is always issued as an error. Use the warning pragma to suppress C4485.

Example
The following sample generates C4485

```
// C4485.cpp
// compile with: /clr
delegate void Del();

ref struct A {
   virtual event Del ^E;
};

ref struct B : A {
   virtual event Del ^E;   // C4485
};

ref struct C : B {
   virtual event Del ^E {
      void raise() override {}
      void add(Del ^) override {}
      void remove(Del^) override {}
   }
};
```

Compiler Warning (level 1) C4486

Error Message
'function' : a private virtual method of a ref class or value class should be marked 'sealed'

Since a private virtual member function of a managed class or struct cannot be accessed or overridden, it should be marked sealed.

Example
The following sample generates C4486.

```
// C4486.cpp
// compile with: /clr /c /W1
ref class B {
private:
   virtual void f() {}   // C4486
   virtual void f1() sealed {}   // OK
};
```

The following sample shows one possible use of a private sealed, virtual function.

```
// C4486_b.cpp
// compile with: /clr /c
ref class B {};

ref class D : B {};

interface class I {
   B^ mf();
};

ref class E : I {
private:
   virtual B^ g() sealed = I::mf {
      return gcnew B;
   }

public:
   virtual D^ mf() {
      return gcnew D;
   }
};
```

Compiler Warning (level 4) C4487

Error Message
'derived_class_function' : matches inherited non-virtual method 'base_class_function' but is not explicitly marked 'new'

A function in a derived class has the same signature as a non-virtual base class function. C4487 reminds you that the derived class function does not override the base class function. Explicitly mark the derived class function as **new** to resolve this warning.

For more information, see new (new slot in vtable).

Example
The following sample generates C4487.

```
// C4487.cpp
// compile with: /W4 /clr
using namespace System;
public ref struct B {
   void f() { Console::WriteLine("in B::f"); }
   void g() { Console::WriteLine("in B::g"); }
};

public ref struct D : B {
   void f() { Console::WriteLine("in D::f"); }   // C4487
   void g() new { Console::WriteLine("in D::g"); }   // OK
};

int main() {
   B ^ a = gcnew D;
   // will call base class functions
   a->f();
   a->g();
}
```

Compiler Warning (level 1) C4488

Error Message
'function' : requires 'keyword' keyword to implement the interface method 'interface_method'

A class must implement all members of an interface from which it directly inherits. An implemented member must have public accessibility, and must be marked virtual.

Example

C4488 can occur if an implemented member is not public. The following sample generates C4488.

```
// C4488.cpp
// compile with: /clr /c /W1 /WX
interface struct MyI {
   void f1();
};

// implemented member not public
ref class B : MyI { virtual void f1() {} };  // C4488

// OK
ref class C : MyI {
public:
   virtual void f1() {}
};
```

C4488 can occur if an implemented member is not marked virtual. The following sample generates C4488.

```
// C4488_b.cpp
// compile with: /clr /c /W1 /WX
interface struct MyI {
   void f1();
};

ref struct B : MyI { void f1() {} };   // C4488
ref struct C : MyI { virtual void f1() {} };   // OK
```

Compiler Warning (level 1) C4489

Error Message

'specifier' : not allowed on interface method 'method'; override specifiers are only allowed on ref class and value class methods

A specifier keyword was incorrectly used on an interface method.

For more information, see Override Specifiers.

Example

The following sample generates C4489.

```
// C4489.cpp
// compile with: /clr /c /W1
public interface class I {
   void f() new;   // C4489
   virtual void b() override;   // C4489

   void g();   // OK
};
```

Compiler Warning (level 1) C4490

Error Message
'override' : incorrect use of override specifier; 'function' does not match a base ref class method

An override specifier was used incorrectly. For example, you do not override an interface function, you implement it.

For more information, see Override Specifiers.

Example
The following sample generates C4490.

```
// C4490.cpp
// compile with: /clr /c /W1

interface struct IFace {
   void Test();
};

ref struct Class1 : public IFace {
   virtual void Test() override {}   // C4490
   // try the following line instead
   // virtual void Test() {}
};
```

Compiler Warning (level 1) C4502

Error Message
'linkage specification' requires use of keyword 'extern' and must precede all other specifiers

A linkage was specified without the **extern** keyword. Linkage is not relevant to non-extern types.

The compiler assumed the **extern** keyword.

Compiler Warning (level 1) C4503

Error Message
'identifier' : decorated name length exceeded, name was truncated

The decorated name was longer than the compiler limit (4096), and was truncated. To avoid this warning and the truncation, reduce the number of arguments or name length of identifiers used.

One situation where this warning will be issued is when your code contains templates specialized on templates repeatedly. For example, a map of maps (from the Standard C++ Library). In this situation, you can make your typedefs a type (struct, for example) that contains the map.

You might, however, decide to not restructure your code. It is possible to ship an application that generates C4503, but if you get link time errors on a truncated symbol, it will be more difficult to determine the type of the symbol in the error. Debugging will also be more difficult; the debugger will also have difficultly mapping symbol name to type name. The correctness of the program, however, is unaffected by the truncated name.

The following sample generates C4503:

```
// C4503.cpp
// compile with: /W1 /EHsc /c
// C4503 expected
#include <string>
#include <map>

class Field{};

typedef std::map<std::string, Field> Screen;
typedef std::map<std::string, Screen> WebApp;
typedef std::map<std::string, WebApp> WebAppTest;
typedef std::map<std::string, WebAppTest> Hello;
Hello MyWAT;
```

The following sample shows one way to rewrite your code to resolve C4503:

```
// C4503b.cpp
// compile with: /W1 /EHsc /c
#include <string>
#include <map>

class Field{};
struct Screen2 {
   std::map<std::string, Field> Element;
};

struct WebApp2 {
   std::map<std::string, Screen2> Element;
};

struct WebAppTest2 {
   std::map<std::string, WebApp2> Element;
};

struct Hello2 {
   std::map<std::string, WebAppTest2> Element;
};

Hello2 MyWAT2;
```

Compiler Warning (level 4) C4505

Error Message
'function' : unreferenced local function has been removed

The given function is local and not referenced in the body of the module; therefore, the function is dead code.

The compiler did not generate code for this dead function.

Compiler Warning (level 1) C4506

Error Message
no definition for inline function 'function'

The given function was declared and marked for inlining but was not defined.

The compiler did not inline the function.

Make sure that external functions to be inlined are declared with the **extern** keyword.

Compiler Warning (level 1) C4508

Error Message
'function' : function should return a value; 'void' return type assumed

The function has no return type specified. In this case, C4430 should also fire and the compiler implements the fix reported by C4430 (default value is int).

To resolve this warning, explicitly declare the return type of functions.

The following sample generates C4508:

```
// C4508.cpp
// compile with: /W1 /c
#pragma warning (disable : 4430)
func() {}   // C4508
void func2() {}   // OK
```

Compiler Warning (level 3) C4509

Error Message
nonstandard extension used: 'function' uses SEH and 'object' has destructor

You should not use structured exception handling in functions that use objects with destructors. If an exception occurs, the destructor cannot be called. Use C++ exception handling instead.

Compiler Warning (level 4) C4510

Error Message
'class' : default constructor could not be generated

The compiler cannot generate a default constructor for the specified class and no user-defined constructor was created. You will not be able to create objects of this type.

There are several situations that prevent the compiler from generating a default constructor, including:

- A const data member.
- A data member that is a reference.

You need to create a user-defined default constructor for the class that initializes these members.

The following sample generates C4510:

```
// C4510.cpp
// compile with: /W4
struct A {
   const int i;
   int &j;
   A& operator=( const A& ); // C4510 expected
   // uncomment the following line to resolve this C4510
   // A(int ii, int &jj) : i(ii), j(jj) {}
};   // C4510

int main() {
}
```

Compiler Warning (level 3) C4511

Error Message
'class' : copy constructor could not be generated

The compiler could not generate a default copy-constructor for a class; a base class may have a private copy-constructor.

Compiler Warning (level 4) C4512

Error Message
'class' : assignment operator could not be generated

The compiler cannot generate an assignment operator for the given class. No assignment operator was created.

An assignment operator for the base class that is not accessible by the derived class can cause this warning.

To avoid this warning, specify a user-defined assignment operator for the class.

The compiler will also generate an assignment operator function for a class that does not define one. This assignment operator is simply a member wise copy of the data members of an object. Because const data items cannot be modified after initialization, if the class contains a const item, the default assignment operator would not work.

You can resolve the C4512 warning for this code in one of three ways:

- Explicitly define an assignment operator for the class.
- Remove **const** from the data item in the class.
- Use the #pragma warning statement to suppress the warning.

Example
The following sample generates C4512.

```
// C4512.cpp
// compile with: /EHsc /W4
// processor: x86

class base {
private:
   const int a;

public:
   base(int val = 0) : a(val) {}
   int get_a() { return a; }
};   // C4512 warning

class base2 {
private:
   const int a;

public:
   base2(int val = 0) : a(val) {}
   base2 & operator=( const base2 & ) {}
   int get_a() { return a; }
};

int main() {
   base first;
   base second;

   // OK
   base2 first2;
   base2 second2;
}
```

Compiler Warning (level 4) C4513

Error Message
'class' : destructor could not be generated

The compiler cannot generate a default destructor for the given class; no destructor was created. The destructor is in a base class that is not accessible to the derived class. If the base class has a private destructor, make it public or protected.

Compiler Warning (level 4) C4514

Error Message
'function' : unreferenced inline function has been removed

The optimizer removed an inline function that is not called.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4514:

```
// C4514.cpp
// compile with: /W4
#pragma warning(default : 4514)
class A
{
   public:
      void func()   // C4514, remove the function to resolve
      {
      }
};

int main()
{
}
```

Compiler Warning (level 4) C4515

Error Message
'namespace' : namespace uses itself

A namespace is used recursively.

The following sample generates C4515:

```
// C4515.cpp
// compile with: /W4
namespace A
{
   using namespace A; // C4515
}

int main()
{
}
```

Compiler Warning (level 4) C4516

Error Message
'class::symbol' : access-declarations are deprecated; member using-declarations provide a better alternative

The ANSI C++ committee has declared access declarations (changing the access of a member in a derived class without the using keyword) to be outdated. Access declarations may not be supported by future versions of C++.

The following sample generates C4516:

```
// C4516.cpp
// compile with: /W4
class A
{
public:
   void x(char);
};

class B : protected A
{
public:
   A::x;  // C4516 on access-declaration
   // use the following line instead
   // using A::x; // using-declaration, ok
};

int main()
{
}
```

Compiler Warning (level 4) C4517

Error Message
access-declarations are deprecated; member using-declarations provide a better alternative

The ANSI C++ committee has declared access declarations (changing the access of a member in a derived class without the using keyword) to be outdated. Access declarations may not be supported by future versions of C++.

Compiler Warning (level 1) C4518

Error Message

'specifier' : storage-class or type specifier(s) unexpected here; ignored

The following sample generates C4518:

```
// C4518.cpp
// compile with: /c /W1
_declspec(dllexport) extern "C" void MyFunction();   // C4518

extern "C" void MyFunction();   // OK
```

Compiler Warning (level 1) C4519

Error Message
default template arguments are only allowed on a class template

Default template arguments are allowed only on a class template declaration or definition. Default template arguments may not be used in a function template, or in the definition of a member of a class template.

By default, C4519 is issued as an error.

Example

The following code shows how to use default template arguments, shows some common mistakes, and shows how to issue C4519 as a warning.

```
// C4519.cpp
// compile with: /W1
// C4519 expected
#pragma warning(1 : 4519)
#include <stdio.h>

// OK : default template argument is allowed on a class template
template<class T=int> class A
{
public:
    static T f(T a, T b)
    {
        return a + b;
    };
};

class B
{
public:
    // C4519 : default template argument is not allowed here
    template<class T=int> static T f(T a, T b)
    {
        return a + b;
    };
};

// C4519 : default template argument is not allowed here
template<class T=int> T f(T a, T b)
{
    return a + b;
};

int main()
{
    double a = 3.5, b = 4.25;
    double d = 0.0;

    // Double is deduced from the types of the arguments to f().
    // Note that f's default template argument is ignored.
    d = f(a, b);   // double f(double, double)

    // Prints the value 7.75
    printf_s("%2.2lf\n", d);

    // Template argument defaults to int. Arguments are
    // coerced to ints, and an int is returned.
    d = A<>::f(a, b);        // int A<>::f(int, int)

    printf_s("%2.2lf\n", d);   // Prints the value 7.00

    // Double is deduced from the types of the
```

```
    // arguments to B::f(). Note that B::f's default
    d = B::f(a, b);

    // template argument is ignored.
    printf_s("%2.2lf\n", d);   // Prints the value 7.75
}
```

Compiler Warning (level 3) C4520

Error Message
'class' : multiple default constructors specified

The class has multiple default constructors. The first constructor is used.

Compiler Warning (level 3) C4521

Error Message
'class' : multiple copy constructors specified

The class has multiple copy constructors of a single type. This warning is informational; the constructors are callable in your program.

Use the warning pragma to suppress this warning.

Example
The following sample generates C4521.

```
// C4521.cpp
// compile with: /EHsc /W3
#include <iostream>

using namespace std;
class A {
public:
   A() { cout << "A's default constructor" << endl; }
   A( A &o ) { cout << "A&" << endl; }
   A( const A &co ) { cout << "const A&" << endl; }   // C4521
};

int main() {
   A o1;         // Calls default constructor.
   A o2( o1 );   // Calls A( A& ).
   const A o3;   // Calls default constructor.
   A o4( o3 );   // Calls A( const A& ).
}
```

Compiler Warning (level 3) C4522

Error Message
'class' : multiple assignment operators specified

The class has multiple assignment operators of a single type. This warning is informational; the constructors are callable in your program.

Use the warning pragma to suppress this warning.

Example

The following sample generates C4522.

```
// C4522.cpp
// compile with: /EHsc /W3
#include <iostream>

using namespace std;
class A {
public:
   A& operator=( A & o ) { cout << "A&" << endl; return *this; }
   A& operator=( const A &co ) { cout << "const A&" << endl; return *this; }   // C4522
};

int main() {
   A o1, o2;
   o2 = o1;
   const A o3;
   o1 = o3;
}
```

Compiler Warning (level 3) C4523

Error Message
'class' : multiple destructors specified

The class has multiple destructors.

Compiler Warning (level 1) C4526

Error Message
'function' : static member function cannot override virtual function 'virtual function'override ignored, virtual function will be hidden

The static member function meets the criteria to override the virtual function, which makes the member function both virtual and static.

The following code generates C4526:

```
// C4526.cpp
// compile with: /W1 /c
// C4526 expected
struct myStruct1 {
   virtual void __stdcall func( int ) = 0;
};

struct myStruct2: public myStruct1 {
   static void __stdcall func( int );
};
```

The following are possible fixes:

- If the function was intended to override the base class virtual function, remove the static specifier.
- If the function was intended to be a static member function, rename it so it doesn't conflict with the base class virtual function.

Compiler Warning (level 1) C4530

Error Message
C++ exception handler used, but unwind semantics are not enabled. Specify /EHsc

C++ exception handling was used but /EHsc was not selected.

When the /EHsc option has not been enabled, an object with automatic storage in the frame, between the function doing the throw and the function catching the throw, will not be destroyed. However, an object with automatic storage created in a **try** or **catch** block will be destroyed.

The following sample generates C4530:

```
// C4530.cpp
// compile with: /W1
int main() {
   try{} catch(int*) {}   // C4530
}
```

Compile the sample with /EHsc to resolve the warning.

Compiler Warning (level 1) C4532

Error Message
'continue' : jump out of __finally/finally block has undefined behavior during termination handling

The compiler encountered one of the following keywords:

- continue
- break
- goto

causing a jump out of a __finally or finally block during abnormal termination.

If an exception occurs, and while the stack is being unwound during execution of the termination handlers (the **__finally** or finally blocks), and your code jumps out of a **__finally** block before the **__finally** block ends, the behavior is undefined. Control may not return to the unwinding code, so the exception may not be handled properly.

If you must jump out of a **__finally** block, check for abnormal termination first.

The following sample generates C4532; simply comment out the jump statements to resolve the warnings.

```
// C4532.cpp
// compile with: /W1
// C4532 expected
int main() {
   int i;
   for (i = 0; i < 10; i++) {
      __try {
      } __finally {
         // Delete the following line to resolve.
         continue;
      }

      __try {
      } __finally {
         // Delete the following line to resolve.
         break;
      }
   }
}
```

Compiler Warning (level 1) C4533

Error Message
initialization of 'variable' is skipped by 'instruction'

An instruction in your program changed the flow of control, such that, an instruction that initialized a variable was not executed. The following sample generates C4533:

```
// C4533.cpp
// compile with: /W1
#include <stdio.h>

struct A
{
   int m_data;
};

int main()
{
   if (1)
   {
      goto Label;
   }

   A a = { 100 };

   Label:   // C4533
      printf("\n%d", a.m_data);   // prints an uninitialized value
}
```

Compiler Warning (level 3) C4534

Error Message
'constructor' will not be a default constructor for class 'class' due to the default argument

An unmanaged class can have a constructor with parameters that have default values and the compiler will use this as the default constructor. A class marked with the **value** keyword will not use a constructor with default values for its parameters as a default constructor.

For more information, see Classes and Structs (Managed).

The following sample generates C4534:

```
// C4534.cpp
// compile with: /W3 /clr /WX
value class MyClass {
public:
   int ii;
   MyClass(int i = 9) {   // C4534, will not be the default constructor
      i++;
   }
};

int main() {
   MyClass ^ xx = gcnew MyClass;
   xx->ii = 0;
}
```

Compiler Warning (level 3) C4535

Error Message
calling _set_se_translator() requires /EHa

The use of _set_se_translator requires the /EHa compiler option and not **/EHs**.

Example
The following sample generates C4535.

```
// C4535.cpp
// compile with: /W3 /EHsc /c
// C4535 expected
// to fix, compile with /EHa instead
#include <stdio.h>
#include <windows.h>
#include <eh.h>

void SEFunc();
void trans_func( unsigned int, EXCEPTION_POINTERS* );

class SE_Exception {
private:
   unsigned int nSE;
public:
   SE_Exception() {}
   SE_Exception( unsigned int n ) : nSE( n ) {}
   ~SE_Exception() {}
   unsigned int getSeNumber() { return nSE; }
};

int main() {
   try {
      _set_se_translator( trans_func );
      SEFunc();
   }
   catch( SE_Exception e ) {
      printf_s( "Caught a __try exception with SE_Exception.\n" );
   }
}

void SEFunc() {
   __try {
      int x, y=0;
      x = 5 / y;
   }
   __finally {
      printf_s( "In finally\n" );
   }
}

void trans_func( unsigned int u, EXCEPTION_POINTERS* pExp ) {
   printf_s( "In trans_func.\n" );
   throw SE_Exception();
}
```

Compiler Warning (level 4) C4536

Error Message
'type name' : type-name exceeds meta-data limit of 'limit' characters

A type name would be truncated in metadata if it was a managed type. See C3180 for more information.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

Compiler Warning (level 1) C4537

Error Message
'object' : 'operator' applied to non-UDT type

A reference was passed where an object (user-defined type) was expected. A reference is not an object, but inline assembler code is not able to make the distinction. The compiler generates code as though ***object*** were an instance.

The following sample generates C4537:

```
// C4537.cpp
// compile with: /W1 /c
// processor: x86
struct S {
   int member;
};

void f1(S &s) {
   __asm mov eax, s.member;   // C4537
   // try the following code instead
   // or, make the declaration "void f1(S s)"
   /*
   mov eax, s
   mov eax, [eax]s.member
   */
}
```

Compiler Warning (level 3) C4538

Error Message
'type' : const/volatile qualifiers on this type are not supported

A qualifier keyword was applied to an array incorrectly. For more information, see array.

The following sample generates C4538:

```
// C4538.cpp
// compile with: /clr /W3 /LD
const array<int> ^f1();   // C4538
array<const int> ^f2();   // OK
```

Compiler Warning (level 1) C4539

Error Message
'char': a universal-character-name specifies an invalid character

A character was specified out of Unicode range.

The character must be in the Unicode range, or be a valid surropate pair.

Example
The following sample generates C4539

```
// C4539.cpp
// compile with: /W1
int main() {
   char \U12345678 = 'x';   // C4539
   char \u2134 = 'x';   // OK 16-bit Unicode char
   char \UD808DF45 = 'x';   // OK 32-bit Unicode value
}
```

Compiler Warning (level 1) C4540

Error Message
dynamic_cast used to convert to inaccessible or ambiguous base; run-time test will fail ('type1' to 'type2')

You used **dynamic_cast** to convert from one type to another. The compiler determined that the cast would always fail (return **NULL**) because a base class is inaccessible (**private**, for instance) or ambiguous (appears more than once in the class hierarchy, for instance).

The following shows an example of this warning. Class **B** is derived from class **A**. The program uses **dynamic_cast** to cast from class **B** (the derived class) to class **A**, which will always fail because class **B** is **private** and thus inaccessible. Changing the access of **A** to **public** will resolve the warning.

```
// C4540.cpp
// compile with: /W1

struct A {
   virtual void g() {}
};

struct B : private A {
   virtual void g() {}
};

int main() {
   B b;
   A * ap = dynamic_cast<A*>(&b);   // C4540
}
```

Compiler Warning (level 1) C4541

Error Message
'identifier' used on polymorphic type 'type' with /GR-; unpredictable behavior may result

You tried to use a feature that requires run-time type information without enabling run-time type information. Recompile with /GR.

Compiler Warning (level 3) C4543

Error Message
Injected text suppressed by attribute 'no_injected_text'

The no_injected_text attribute appeared in source code, which means the compiler will prevent attributes from injecting code.

This warning is to remind you that attributes will not be able to inject code.

Compiler Warning (level 1) C4544

Error Message
Default arguments on template declaration ignored

A default template argument was specified in an incorrect location and was ignored. A default template argument for a class template can only be specified in the declaration or definition of the class template and not on a member of the class template.

The following sample generates C4545:

```
// C4544.cpp
// compile with: /W1 /LD
template <class T>
struct S
{
   template <class T1>
      struct S1;
   void f();
};

template <class T=int>
template <class T1>
struct S<T>::S1 {};   // C4544
```

In the following example, the default parameter applies to class template `S`:

```
// C4544b.cpp
// compile with: /LD
template <class T = int>
struct S
{
   template <class T1>
      struct S1;
   void f();
};

template <class T>
template <class T1>
struct S<T>::S1 {};
```

Compiler Warning (level 1) C4545

Error Message
expression before comma evaluates to a function which is missing an argument list

The compiler detected an ill-formed comma expression.

This warning is off by default. For more information, see Compiler Warnings That Are Off by Default.

The following sample generates C4545:

```
// C4545.cpp
// compile with: /W1
#pragma warning (default : 4545)

void f() { }

int main()
{
   *(&f), 10;   // C4545
   // try the following line instead
   // (*(&f))(), 10;
}
```

Compiler Warning (level 1) C4546

Error Message
function call before comma missing argument list

The compiler detected an ill-formed comma expression.

This warning is off by default. For more information, see Compiler Warnings That Are Off by Default.

Example

The following sample generates C4546:

```
// C4546.cpp
// compile with: /W1
#pragma warning (default : 4546)
void f(int i) {
   i++;
}

int main() {
   int i = 0, k = 0;

   if ( f, k )   // C4546
   // try the following line instead
   // if ( f(i), k )
      i++;
}
```

Compiler Warning (level 1) C4547

Error Message
'operator' : operator before comma has no effect; expected operator with side-effect

The compiler detected an ill-formed comma expression.

This warning is off by default. For more information, see Compiler Warnings That Are Off by Default.

The following sample generates C4547:

```
// C4547.cpp
// compile with: /W1
#pragma warning (default : 4547)
int i = 0;
int j = 1;
int main() {
   int l = (i != i,0);   // C4547
   // try the following line instead
   // int l = (i != i);
   // or
   // int l = ((void)(i != i),0);
}
```

Compiler Warning (level 1) C4548

Error Message
expression before comma has no effect; expected expression with side-effect

The compiler detected an ill-formed comma expression.

This warning is off by default. For more information, see Compiler Warnings That Are Off by Default.

The following sample generates C4548:

```
// C4548.cpp
// compile with: /W1
#pragma warning (default : 4548)
int main()
{
   int i = 0, k = 0;

   if ( i, k )   // C4548
   // try the following line instead
   // if ( i = 0, k )
      i++;
}
```

Compiler Warning (level 1) C4549

Error Message
'operator' : operator before comma has no effect; did you intend 'operator'?

The compiler detected an ill-formed comma expression.

This warning is off by default. For more information, see Compiler Warnings That Are Off by Default.

The following sample generates C4549:

```
// C4549.cpp
// compile with: /W1
#pragma warning (default : 4549)

int main() {
   int i = 0, k = 0;

   if ( i == 0, k )   // C4549
   // try the following line instead
   // if ( i == 0 )
      i++;
}
```

Compiler Warning (level 1) C4550

Error Message
expression evaluates to a function which is missing an argument list

A dereferenced pointer to a function is missing an argument list.

Example

```
// C4550.cpp
// compile with: /W1
bool f()
{
   return true;
}

typedef bool (*pf_t)();

int main()
{
   pf_t pf = f;
   if (*pf) {}  // C4550
   return 0;
}
```

Compiler Warning (level 1) C4551

Error Message
function call missing argument list

A function call must include the open and close parentheses after the function name even if the function takes no parameters.

The following sample generates C4551:

```
// C4551.cpp
// compile with: /W1
void function1() {
}

int main() {
   function1;   // C4551
   function1();   // OK
}
```

Compiler Warning (level 1) C4552

Error Message
'operator' : operator has no effect; expected operator with side-effect

If an expression statement has an operator with no side effect as the top of the expression, it's probably a mistake.

To override this warning, put the expression in parentheses.

The following sample generates C4552:

```
// C4552.cpp
// compile with: /W1
int main() {
   int i, j;
   i + j;   // C4552
   // try the following line instead
   // (i + j);
}
```

Compiler Warning (level 1) C4553

Error Message
'operator' : operator has no effect; did you intend 'operator'?

If an expression statement has an operator with no side effect as the top of the expression, it's probably a mistake.

The following sample generates C4553:

```
// C4553.cpp
// compile with: /W1
int func()
{
   return 0;
}

int main()
{
   int i;
   i == func();   // C4553
   // try the following line instead
   // i = func();
}
```

Compiler Warning (level 3) C4554

Error Message

'operator' : check operator precedence for possible error; use parentheses to clarify precedence

The following sample generates C4554:

```
// C4554.cpp
// compile with: /W3 /WX
int main() {
   int a = 0, b = 0, c = 0;
   a = a << b + c;   // C4554
   // probably intended (a << b) + c,
   // but will get a << (b + c)
}
```

Compiler Warning (level 1) C4555

Error Message
expression has no effect; expected expression with side-effect

This warning informs you when an expression has no effect.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

For example:

```
// C4555.cpp
// compile with: /W1
#pragma warning(default:4555)

void func1()
{
   1;   // C4555
}

void func2()
{
   int x;
   x;   // C4555
}

int main()
{
}
```

Compiler Warning (level 1) C4556

Error Message
value of intrinsic immediate argument 'value' is out of range 'lowerbound - upperbound'

An intrinsic matches a hardware instruction. The hardware instruction has a fixed number of bits to encode the constant. If ***value*** is out of range, it will not encode properly. The compiler truncates the extra bits.

The following sample generates C4556:

```
// C4556.cpp
// compile with: /W1
// processor: x86 IPF
#include <xmmintrin.h>
void test()
{
   __m64 m;
   _m_pextrw(m, 5);   // C4556
}
int main()
{
}
```

Compiler Warning (level 3) C4557

Error Message
'__assume' contains effect 'effect'

The value passed to an __assume statement2 was modified.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4557:

```
// C4557.cpp
// compile with: /W3
#pragma warning(default : 4557)
int main()
{
   int i;
   __assume(i++);   // C4557
}
```

Compiler Warning (level 1) C4558

Error Message
value of operand 'value' is out of range 'lowerbound - upperbound'

The value passed to an assembly language instruction is out of the range specified for the parameter. The value will be truncated.

The following sample generates C4558:

```
// C4558.cpp
// compile with: /W1
// processor: x86
void asm_test() {
   __asm pinsrw   mm1, eax, 8;   // C4558
}

int main() {
}
```

Compiler Warning (level 4) C4559

Error Message
'function' : redefinition; the function gains __declspec(modifier)

A function was redefined or redeclared and the second definition or declaration added a **__declspec** modifier (***modifier***). This warning is informational. To fix this warning, delete one of the definitions.

The following sample generates C4559:

```
// C4559.cpp
// compile with: /W4 /LD
void f();
__declspec(noalias) void f();   // C4559
```

Compiler Warning (level 1) C4561

Error Message
'__fastcall' incompatible with the '/clr' option: converting to '__stdcall'

The __fastcall function-calling convention cannot be used with the /clr compiler option. The compiler ignores the calls to **__fastcall**. To fix this warning, either remove the calls to **__fastcall** or compile without **/clr**.

The following sample generates C4561:

```
// C4561.cpp
// compile with: /clr /W1 /c
// processor: x86
void __fastcall Func(void *p);   // C4561, remove __fastcall to resolve
```

Compiler Warning (level 4) C4564

Error Message
method 'method' of class 'class' defines unsupported default parameter 'parameter'

The compiler detected a method with one or more parameters with default values. The default value(s) for the parameters will be ignored when the method is invoked; explicitly specify values for those parameters. If you do not explicitly specify values for those parameters, the C++ compiler will generate an error.

Given the following .dll created with Visual Basic, which allows default parameters on method arguments:

```
' C4564.vb
' compile with: vbc /t:library C4564.vb
Public class TestClass
   Public Sub MyMethod (a as Integer, _
                        Optional c as Integer=1)
   End Sub
End class
```

And the following C++ sample that uses the .dll created with Visual Basic,

```
// C4564.cpp
// compile with: /clr /W4 /WX
#using <C4564.dll>

int main() {
   TestClass ^ myx = gcnew TestClass();   // C4564
   myx->MyMethod(9);
   // try the following line instead, to avoid an error
   // myx->MyMethod(9, 1);
}
```

Compiler Warning (level 4) C4565

Error Message
'function' : redefinition; the symbol was previously declared with __declspec(modifier)

A symbol was redefined or redeclared and the second definition or declaration, unlike the first definition or declaration, did not have a **__declspec** modifier (***modifier***). This warning is informational. To fix this warning, delete one of the definitions.

The following sample generates C4565:

```
// C4565.cpp
// compile with: /W4 /LD
__declspec(noalias) void f();
void f();   // C4565
```

Compiler Warning (level 1) C4566

Error Message
character represented by universal-character-name 'char' cannot be represented in the current code page (page)

Not every Unicode character can be represented in your current ANSI code page.

Narrow strings (one-byte characters) are converted to multi-byte characters whereas wide strings (two-byte characters) are not.

The following sample generates C4566:

```
// C4566.cpp
// compile with: /W1
int main() {
   char c1 = '\u03a0';   // C4566
   char c2 = '\u0642';   // C4566

   wchar_t c3 = L'\u03a0';   // OK
   wchar_t c4 = L'\u0642';   // OK
}
```

Compiler Warning (level 3) C4570

Error Message

'type' : is not explicitly declared as abstract but has abstract functions

A type that contains abstract (Visual C++) functions should itself be marked as abstract.

Example

The following sample generates C4570.

```
// C4570.cpp
// compile with: /clr /W3 /c
ref struct X {   // C4570
// try the following line instead
// ref class X abstract {
   virtual void f() abstract;
};
```

Compiler Warning (level 4) C4571

Error Message
Informational: catch(...) semantics changed since Visual C++ 7.1; structured exceptions (SEH) are no longer caught

C4571 is generated for every catch(...) block when compiling with **/EHs**.

When compiling with **/EHs**, a catch(...) block will not catch a structured exception (divide by zero, null pointer, for example); a catch(...) block will only catch explicitly-thrown, C++ exceptions. For more information, see Exception Handling in Visual C++.

This warning is off by default. Turn this warning on to ensure that when you compile with **/EHs** your catch (...) blocks do not intend to catch structured exceptions. See Compiler Warnings That Are Off by Default for more information.

You can resolve C4571 in one of the following ways,

- Compile with **/EHa** if you still want your catch(...) blocks to catch structured exceptions.
- Do not enable C4571 if you do not want your catch(...) blocks to catch structured exceptions, but you still want to use catch(...) blocks. You can still catch structured exceptions using the structured exception handling keywords (**__try**, **__except**, and **__finally**). But remember, when compiled **/EHs** destructors will only be called when a C++ exception is thrown, not when an SEH exception occurs.
- Replace catch(...) block with catch blocks for specific C++ exceptions, and optionally, add structured exception handling around the C++ exception handling (**__try**, **__except**, and **__finally**). See Structured Exception Handling (C++) for more information.

See /EH (Exception Handling Model) for more information.

Example
The following sample generates C4571.

```
// C4571.cpp
// compile with: /EHs /W4 /c
#pragma warning(default : 4571)
int main() {
   try {
      int i = 0, j = 1;
      j /= i;   // this will throw a SE (divide by zero)
   }
   catch(...) {}   // C4571 warning
}
```

Compiler Warning (level 1) C4572

Error Message
[ParamArray] attribute is deprecated under /clr, use '...' instead

An obsolete style for specifying a variable argument list was used. When compiling for the CLR, use ellipsis syntax instead of ParamArrayAttribute. For more information, see How to: Accept Variable Arguments.

Example
The following sample generates C4572.

```
// C4572.cpp
// compile with: /clr /W1
void Func([System::ParamArray] array<int> ^);   // C4572
void Func2(... array<int> ^){}   // OK

int main() {
   Func2(1, 2, 3);
}
```

Compiler Warning (level 3) C4580

Error Message
[attribute] is deprecated; instead specify System::Attribute as a base class

attribute is no longer the preferred syntax for creating user-defined attributes. For more information, see User-Defined Attributes.

Example
The following sample generates C3454.

```
// C4580.cpp
// compile with: /W3 /c /clr
[attribute]   // C4580
public ref class Attr {
public:
   int m_t;
};

public ref class Attr2 : System::Attribute {
public:
   int m_t;
};
```

Compiler Warning (level 1) C4581

Error Message
deprecated behavior: '"string1"' replaced with 'string2' to process attribute

This error can be generated as a result of compiler conformance work that was done for Visual C++ 2005: parameter checking for Visual C++ attributes.

In previous versions, attribute values were accepted whether or not they were enclosed in quotation marks. If the value is an enumeration, it must not be enclosed in quotation marks.

See Breaking Changes in the Visual C++ 2005 Compiler for more information.

Example
The following sample generates C4581.

```
// C4581.cpp
// compile with: /c /W1
#include "unknwn.h"
[object, uuid("00000000-0000-0000-0000-000000000001")]
__interface IMyI : IUnknown {};

[coclass, uuid(12345678-1111-2222-3333-123456789012), threading("free")]   // C4581
// try the following line instead
// [coclass, uuid(12345678-1111-2222-3333-123456789012), threading(free)]
class CSample : public IMyI {};
```

Compiler Warning (level 1) C4584

Error Message
'class1' : base-class 'class2' is already a base-class of 'class3'

The class you defined inherits from two classes, one of which inherits from the other. For example:

```
// C4584.cpp
// compile with: /W1 /LD
class A {
};

class B : public A {
};

class C : public A, public B { // C4584
};
```

In this case, a warning would be issued on class C because it inherits both from class A and class B, which also inherits from class A. This warning serves as a reminder that you must fully qualify the use of members from these base classes or a compiler error will be generated due to the ambiguity as to which class member you refer.

Compiler Warnings C4600 Through C4999

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, type the error number in the **Look for** box in the Index, or click a link below.

Compiler Warning (level 1) C4600

Compiler Warning (level 1) C4602

Compiler Warning (level 1) C4606

Compiler Warning (level 3) C4608

Compiler Warning (level 4) C4610

Compiler Warning (level 4) C4611

Compiler Warning (level 1) C4612

Compiler Warning (level 1) C4613

Compiler Warning (level 1) C4615

Compiler Warning (level 1) C4616

Compiler Warning (level 1) C4618

Compiler Warning (level 3) C4619

Compiler Warning (level 1) C4620

Compiler Warning (level 1) C4621

Compiler Warning (level 3) C4622

Compiler Warning (level 4) C4623

Compiler Warning (level 1) C4624

Compiler Warning (level 4) C4625

Compiler Warning (level 4) C4626

Compiler Warning (level 1) C4628

Compiler Warning (level 4) C4629

Compiler Warning (level 1) C4630

Compiler Warning (level 1) C4631

Compiler Warning (level 1) C4632

Compiler Warning (level 3) C4633

Compiler Warning (level 4) C4634

Compiler Warning (level 3) C4635

Compiler Warning (level 3) C4636

Compiler Warning (level 3) C4637

Compiler Warning (level 3) C4638

Compiler Warning (level 4) C4639

Compiler Warning (level 3) C4640

Compiler Warning (level 3) C4641

Compiler Warning (level 3) C4645

Compiler Warning (level 3) C4646

Compiler Warning (level 1) C4650

Compiler Warning (level 1) C4651

Compiler Warning (level 1) C4652

Compiler Warning (level 2) C4653

Compiler Warning (level 1) C4655

Compiler Warning (level 1) C4656

Compiler Warning (level 1) C4657

Compiler Warning (level 1) C4659

Compiler Warning (level 1) C4661

Compiler Warning (level 1) C4662

Compiler Warning (level 1) C4667

Compiler Warning (level 4) C4668

Compiler Warning (level 1) C4669

Compiler Warning (level 4) C4670

Compiler Warning (level 4) C4672

Compiler Warning (level 4) C4673

Compiler Warning (level 1) C4674

Compiler Warning (level 1) C4677

Compiler Warning (level 1) C4678

Compiler Warning (level 1) C4679

Compiler Warning (level 4) C4680

Compiler Warning (level 4) C4681

Compiler Warning (level 4) C4682

Compiler Warning (level 1) C4683

Compiler Warning (level 1) C4684

Compiler Warning (level 1) C4685

Compiler Warning (level 3) C4686

Compiler Warning C4687

Compiler Warning (level 1) C4688

Compiler Warning (level 4) C4690

Compiler Warning (level 1) C4691

Compiler Warning (level 1) C4692

Compiler Warning C4693

Compiler Warning C4694

Compiler Warning (level 1 and level 4) C4700

Compiler Warning (level 4) C4701

Compiler Warning (level 4) C4702

Compiler Warning (level 4) C4706

Compiler Warning (level 4) C4709

Compiler Warning (level 4) C4710

Compiler Warning (level 1) C4711

Compiler Warning (level 4) C4714

Compiler Warning (level 1) C4715

Compiler Warning (level 1) C4716

Compiler Warning (level 1) C4717

Compiler Warning (level 4) C4718

Compiler Warning (level 1) C4722

Compiler Warning (level 3) C4723

Compiler Warning (level 3) C4724

Compiler Warning (level 4) C4725

Compiler Warning (level 1) C4727

Compiler Warning (Level 1) C4729

Compiler Warning (Level 1) C4730

Compiler Warning (Level 1) C4731

Compiler Warning (Level 1) C4733

Compiler Warning (Level 1) C4734

Compiler Warning (Level 3) C4738

Compiler Warning (Level 1) C4739

Compiler Warning (Level 4) C4740

Compiler Warning (Level 1) C4742

Compiler Warning (Level 1) C4743

Compiler Warning (Level 1) C4744

Compiler Warning (level 1) C4747

Compiler Warning (level 3) C4748

Compiler Warning C4750

Compiler Warning (level 2) C4756

Compiler Warning (level 4) C4764

Compiler Warning (level 1) C4772

Compiler Warning (Level 1) C4788

Compiler Warning (Level 1) C4789

Compiler Warning (level 3) C4792

Compiler Warning (level 1 and 3) C4793

Compiler Warning (level 1) C4794

Compiler Warning (level 1) C4799

Compiler Warning (level 3) C4800

Compiler Warning C4801

Compiler Warning (level 1) C4803

Compiler Warning (level 1) C4804

Compiler Warning (level 1) C4805

Compiler Warning (level 1) C4806

Compiler Warning (level 1) C4807

Compiler Warning (level 1) C4808

Compiler Warning (level 1) C4809

Compiler Warning (level 1) C4810

Compiler Warning (level 1) C4811

Compiler Warning (level 1) C4812

Compiler Warning (level 1) C4813

Compiler Warning (level 4) C4815

Compiler Warning (level 4) C4816

Compiler Warning (level 1) C4817

Compiler Warning (level 1) C4819

Compiler Warning (level 4) C4820

Compiler Warning (level 1) C4821

Compiler Warning (level 1) C4822

Compiler Warning (level 3) C4823

Compiler Warning (level 2) C4826

Compiler Warning (level 1) C4829

Compiler Warning (level 1) C4832

Compiler Warning (level 1) C4835

Compiler Warning (level 1) C4836

Compiler Warning C4867

Compiler Warning (level 1) C4900

Compiler Warning (level 1) C4905

Compiler Warning (level 1) C4906

Compiler Warning (level 1) C4912

Compiler Warning (level 4) C4913

Compiler Warning (level 1) C4917

Compiler Warning (level 4) C4918

Compiler Warning (level 1) C4920

Compiler Warning (level 1) C4925

Compiler Warning (level 1) C4926

Compiler Warning (level 1) C4927

Compiler Warning (level 1) C4928

Compiler Warning (level 1) C4929

Compiler Warning (level 1) C4930

Compiler Warning (level 4) C4931

Compiler Warning (level 4) C4932

Compiler Warning (level 1) C4935

Compiler Warning (level 4) C4937

Compiler Warning (level 4) C4938

Compiler Warning (level 1) C4939

Compiler Warning (level 1) C4944

Compiler Warning (level 1) C4945

Compiler Warning (level 1) C4946

Compiler Warning (level 1) C4947

Compiler Warning (level 2) C4948

Compiler Warning (level 1 and level 4) C4949

Compiler Warning (level 1) C4951

Compiler Warning (level 1) C4952

Compiler Warning (level 1) C4953

Compiler Warning (level 1) C4955

Compiler Warning (level 4) C4960

Compiler Warning (level 1) C4964

Compiler Warning (level 1) C4965

Compiler Warning (level 3) C4980

Compiler Warning (level 1) C4995

Compiler Warning (level 1) C4996

Compiler Warning (level 1) C4997

Compiler Warning (level 1) C4999

Compiler Warning (level 1) C4600

Error Message
#pragma 'macro name' : expected a valid non-empty string

You cannot specify an empty string when you push or pop a macro name with either the pop_macro or push_macro.

The following sample generates C4600:

```
// C4600.cpp
// compile with: /W1
int main()
{
   #pragma push_macro("")   // C4600 passing an empty string
}
```

Compiler Warning (level 1) C4602

Error Message
#pragma pop_macro : 'macro name' no previous #pragma push_macro for this identifier

If you use pop_macro for a particular macro, you must first have passed that macro name to push_macro. For example, the following sample generates C4602:

```
// C4602.cpp
// compile with: /W1
int main()
{
   #pragma pop_macro("x")   // C4602 x is not on the stack
}
```

Compiler Warning (level 1) C4606

Error Message
#pragma warning : 'warning_number' ignored; Code Analysis warnings are not associated with warning levels

For Code Analysis warnings, only **error**, **once**, and **default** are supported with the warning pragma.

Example
The following sample generates C4606.

```
// C4606.cpp
// compile with: /c /W1
#pragma warning(1: 6001)   // C4606
#pragma warning(once: 6001)   // OK
```

Compiler Warning (level 3) C4608

Error Message
'union_member' has already been initialized by another union member in the initializer list, 'union_member'

Two members of the same union were initialized in an initialization list. You can only access one member of the union.

The following sample generates C4608:

```
// C4608.cpp
// compile with: /W3 /c
class X {
public:
   X(char c) : m_i( c + 1), m_c(c) {}   // C4608
   // try the following line instead
   // X(char c) : m_c(c) {}


private:
   union {
      int m_i;
      char m_c;
   };
};

union Y {
public:
   Y(char * name) : m_number(0.3), m_string( name ) {} // C4608

private:
   double m_number;
   char * m_string;
};
```

Compiler Warning (level 4) C4610

Error Message
object 'class' can never be instantiated - user-defined constructor required

The class has no user-defined or default constructors. No instantiation is performed. The following sample generates C4610:

```
// C4610.cpp
// compile with: /W4
struct A {
   int &j;

   A& A::operator=( const A& );
};   // C4610

/* use this structure definition to resolve the warning
struct B {
   int &k;

   B(int i = 0) : k(i) {
   }

   B& B::operator=( const B& );
} b;
*/

int main() {
}
```

Compiler Warning (level 4) C4611

Error Message
interaction between 'function' and C++ object destruction is non-portable

On some platforms, functions that include **catch** may not support C++ object semantics of destruction when out of scope.

To avoid unexpected behavior, avoid using **catch** in functions that have constructors and destructors.

This warning is only issued once; see pragma warning.

Compiler Warning (level 1) C4612

Error Message
error in include filename

This warning occurs with **#pragma include_alias** when a filename is incorrect or missing.

The arguments to the **#pragma include_alias** statement can use the quote from ("*filename*") or angle-bracket form (**<***filename***>**), but both must use the same form.

Example

```
// C4612.cpp
// compile with: /W1 /LD
#pragma include_alias("StandardIO", <stdio.h>) // C4612
```

Compiler Warning (level 1) C4613

Error Message
'segment' : class of segment cannot be changed

You tried to create a segment with the same class name as a segment used by the compiler. No new segment class was created.

Compiler Warning (level 1) C4615

Error Message
#pragma warning : unknown user warning type

An invalid warning specifier was used with **pragma** warning. To resolve the error, use a valid warning specifier.

The following sample generates C4615:

```
// C4615.cpp
// compile with: /W1 /LD
#pragma warning(enable : 4401)   // C4615, 'enable' not valid specifier

// use the code below to resolve the error
// #pragma warning(default : 4401)
```

Compiler Warning (level 1) C4616

Error Message
#pragma warning : warning number 'number' not a valid compiler warning

The warning number specified in the warning pragma cannot be reassigned. The pragma was ignored.

The following sample generates C4616:

```
// C4616.cpp
// compile with: /W1 /c
#pragma warning( disable : 0 )   // C4616
#pragma warning( disable : 999 )   // OK
#pragma warning( disable : 4998 )   // OK
```

Compiler Warning (level 1) C4618

Error Message
pragma parameters included an empty string; pragma ignored

A null string was given as an argument to a **#pragma**.

The pragma was processed without the argument.

The following sample generates C4618:

```
// C4618.cpp
// compile with: /W1 /LD
#pragma code_seg("")   // C4618
```

Compiler Warning (level 3) C4619

Error Message
#pragma warning : there is no warning number 'number'

An attempt was made to disable a warning that does not exist.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4619:

```
// C4619.cpp
// compile with: /W3 /c
#pragma warning(default : 4619)
#pragma warning(disable : 4354)   // C4619, C4354 does not exist
```

Compiler Warning (level 1) C4620

Error Message
no postfix form of 'operator ++' found for type 'type', using prefix form

There is no postfix increment operator defined for the given type. The compiler used the overloaded prefix operator.

This warning can be avoided by defining a postfix **++** operator. Create a two-argument version of the **++** operator as shown here:

```
// C4620.cpp
// compile with: /W1
class A
{
public:
   A(int nData) : m_nData(nData)
   {
   }

   A operator++()
   {
      m_nData -= 1;
      return *this;
   }

   // A operator++(int)
   // {
   //    A tmp = *this;
   //    m_nData -= 1;
   //    return tmp;
   // }

private:
   int m_nData;
};

int main()
{
   A a(10);
   ++a;
   a++;   // C4620
}
```

Compiler Warning (level 1) C4621

Error Message
no postfix form of 'operator --' found for type 'type', using prefix form

There was no postfix decrement operator defined for the given type. The compiler used the overloaded prefix operator.

This warning can be avoided by defining a postfix **--** operator. Create a two-argument version of the **--** operator as shown below:

```
// C4621.cpp
// compile with: /W1
class A
{
public:
   A(int nData) : m_nData(nData)
   {
   }

   A operator--()
   {
      m_nData -= 1;
      return *this;
   }

   // A operator--(int)
   // {
   //    A tmp = *this;
   //    m_nData -= 1;
   //    return tmp;
   // }

private:
   int m_nData;
};

int main()
{
   A a(10);
   --a;
   a--;   // C4621
}
```

Compiler Warning (level 3) C4622

Error Message

Overwriting debug information formed during creation of the precompiled header in object file: 'file'

CodeView information in the specified file was lost when it was compiled with the /Yu (Use Precompiled Headers) option.

Rename the object file (using /Fo) when creating or using the precompiled header file, and link using the new object file.

Compiler Warning (level 4) C4623

'*derived class*' : default constructor could not be generated because a base class default constructor is inaccessible

A constructor was not accessible in a base class and was therefore not generated for a derived class. Any attempt to create an object of this type on the stack will cause a compiler error.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

Example

The following sample generates C4623.

```
// C4623.cpp
// compile with: /W4
#pragma warning(default : 4623)
class B {
   B();
};

class C {
public:
   C();
};

class D : public B {};   // C4623 make B's constructor public
class E : public C {};   // OK

int main() {
   // D d;  will cause an error
}
```

Compiler Warning (level 1) C4624

Error Message
'derived class' : destructor could not be generated because a base class destructor is inaccessible

A destructor was not accessible in a base class and was therefore not generated for a derived class. Any attempt to create an object of this type on the stack will cause a compiler error.

The following sample generates C4624:

```
// C4624.cpp
// compile with: /W1 /c
class B {
// Uncomment the following line to resolve.
// public:
   ~B();
};

class D : public B {};   // C4624 B's destructor not public
```

Compiler Warning (level 4) C4625

Error Message
'derived class' : copy constructor could not be generated because a base class copy constructor is inaccessible

A copy constructor was not accessible in a base class and was therefore not generated for a derived class. Any attempt to copy an object of this type will cause a compiler error.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

Example
The following sample generates C4625.

```
// C4625.cpp
// compile with: /W4 /c
#pragma warning(default : 4625)

struct A {
   A() {}

private:
   A(const A&) {}
};

struct C : private virtual A {};
struct B :  C {};   // C4625 no copy constructor

struct D : A {};
struct E :  D {};   // OK
```

Compiler Warning (level 4) C4626

Error Message
'derived class' : assignment operator could not be generated because a base class assignment operator is inaccessible

An assignment operator was not accessible in a base class and was therefore not generated for a derived class. Any attempt to assign objects of this type will cause a compiler error.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4626:

```
// C4626
// compile with: /W4
#pragma warning(default : 4626)
class B
{
// public:
   B& operator = (const B&)
   {
      return *this;
   }
};

class D : public B
{

}; // C4626, make B's copy constructor public

int main()
{
   D m;
   D n;
   // m = n;   // this line will cause an error
}
```

Compiler Warning (level 1) C4628

Error Message
digraphs not supported with -Ze. Character sequence 'digraph' not interpreted as alternate token for 'char'

Digraphs are not supported under /Ze. This warning will be followed by an error.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4628:

```
// C4628.cpp
// compile with: /WX
#pragma warning(default : 4628)
int main()
<%   // C4628 <% digraph for {
}
```

Compiler Warning (level 4) C4629

Error Message
digraph used, character sequence 'digraph' interpreted as token 'char' (insert a space between the two characters if this is not what you intended)

Under /Za, the compiler warns when it detects a digraph.

The following sample generates C4629:

```
// C4629.cpp
// compile with: /Za /W4
int main()
<%   // C4629 <% digraph for {
}
```

Compiler Warning (level 1) C4630

Error Message
'symbol' : 'extern' storage class specifier illegal on member definition

A data member or member function is defined as **extern**. Members cannot be external, although entire objects can. The compiler ignores the **extern** keyword. The following sample generates C4630:

```
// C4630.cpp
// compile with: /W1 /LD
class A {
   void func();
};

extern void A::func() {   // C4630, remove 'extern' to resolve
}
```

Compiler Warning (level 1) C4631

Error Message
MSXML or XPath unavailable, XML document comments will not be processed. reason

Your common language runtime installation did not have the necessary files to support processing doc comment. Reinstall the common language runtime.

Compiler Warning (level 1) C4632

Error Message
XML document comment: file - access denied: reason

The path to .xdc file (*file*) was not valid, and no .xdc file created.

The following sample generates C4632:

```
// C4632.cpp
// compile with: /clr /docv:\\falsedir /LD /W1
// C4632 expected

/// Text for class MyClass.
public ref class MyClass {};
```

Compiler Warning (level 3) C4633

Error Message
XML document comment target: error: reason

A name passed to the <param> tag was not found by the compiler.

The following sample generates C4633:

```
// C4633.cpp
// compile with: /clr /doc /LD /W3

/// Text for class MyClass.
public ref class MyClass {
   // C4633 remove line for Int3
   /// <param name="Int1">Used to indicate status.</param>
   /// <param name="Int3">Used to indicate status.</param>
   void MyMethod(int Int1) {
      Int1 = 0;
      Int1++;
   }
};
```

Compiler Warning (level 4) C4634

Error Message
XML document comment: cannot be applied: reason

XML documentation tags can not be applied to all C++ constructs. For example, you cannot add a documentation comment to a namespace or template.

For more information, see XML Documentation (Visual C++).

Example
The following sample generates C4634.

```
// C4634.cpp
// compile with: /W4 /doc /c
/// This is a namespace.   // C4634
namespace hello {
   class MyClass  {};
};
```

The following sample generates C4634.

```
// C4634_b.cpp
// compile with: /W4 /doc /c
/// This is a template.   // C4634
template <class T>
class MyClass  {};
```

Compiler Warning (level 3) C4635

Error Message
XML document comment target: badly-formed XML: reason

The compiler found some problem with the XML tags. Fix the problem and recompile

The following sample generates C4635:

```
// C4635.cpp
// compile with: /doc /clr /W3 /c
/// <summary>
/// The contents of the folder have changed.
/// <summary/>   // C4635

// try the following line instead
// /// </summary>
public ref class Test {};
```

Notice that the output for this sample says: **End tag 'member' does not match the start tag 'summary'.**

The problem with this sample is that the end tag for <summary> is poorly formed, and the compiler does not recognize it as the <summary> end tag. The <member> tag is embedded in the .xdc file by the compiler in every /doc compilation. So, the problem here is that the end tag </member>, does not match the previous start tag that the compiler processed (<summary>.

Compiler Warning (level 3) C4636

Error Message
XML document comment applied to 'construct': tag requires non-empty '' attribute.

A tag, such as **cref**, did not have a value.

Example
The following sample generates C4636.

```
// C4636.cpp
// compile with: /clr /doc /W3 /c
/// <see cref=''/>
// /// <see cref='System::Exception'/>
ref struct A {   // C4636
   void f(int);
};

// OK
/// <see cref='System::Exception'/>
ref struct B {
   void f(int);
};
```

Compiler Warning (level 3) C4637

Error Message
XML document comment target: <include> tag discarded. reason

The syntax of an <include> tag was not correct.

The following sample generates C4637:

```
// C4637.cpp
// compile with: /clr /doc /LD /W3
using namespace System;

/// Text for class MyClass.
public ref class MyClass {
public:
   /// <include file="c:\davedata\jtest\xml_include.doc"/>
   // Try the following line instead:
   // /// <include file='xml_include.doc' path='MyDocs/MyMembers/*' />
   void MyMethod() {
   }
};   // C4637
```

Compiler Warning (level 3) C4638

Error Message
XML document comment target: reference to unknown symbol 'symbol'

The compiler was unable to resolve a symbol (***symbol***). The symbol must be valid in the compilation.

The following sample generates C4638:

```
// C4638.cpp
// compile with: /clr /doc /LD /W3
using namespace System;

/// Text for class MyClass.
public ref class MyClass {
public:
   /// <summary> Text </summary>
   /// <see cref="aSymbolThatAppearsNowhereInMyProject"/>
   // Try the following line instead:
   // /// <see cref="System::Console::WriteLine"/>
   void MyMethod() {}
};   // C4638
```

Compiler Warning (level 4) C4639

Error Message
MSXML error, XML document comments will not be processed. reason

This warning can occur for any number of reasons.

To resolve this warning:

- Recompile.
- Reinstall MSXML by reinstalling the common language runtime.
- Edit or remove the documentation comment that is causing the warning and recompile.

When C4639 is issued, all further XML comment processing is disabled, and the .xdc file is not generated.

Compiler Warning (level 3) C4640

Error Message
'instance' : construction of local static object is not thread-safe

A static instance of an object is not thread safe.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4640:

```
// C4640.cpp
// compile with: /W3
#pragma warning(default:4640)

class X {
public:
   X() {
   }
};

void f() {
   static X aX;   // C4640
}

int main() {
   f();
}
```

Compiler Warning (level 3) C4641

Error Message
XML document comment has an ambiguous cross reference

The compiler was unable to unambiguously resolve a reference. To resolve this warning, specify the parameter information necessary to make the reference unambiguous.

For more information, see XML Documentation (Visual C++).

Example
The following sample generates C4641.

```
// C4641.cpp
// compile with: /W3 /doc /clr /c

/// <see cref="f" />   // C4641
// try the following line instead
// /// <see cref="f(int)" />
public ref class GR {
public:
   void f( int ) {}
   void f( char ) {}
};
```

Compiler Warning (level 3) C4645

Error Message
function declared with __declspec(noreturn) has a return statement

A return statement was found in a function that is marked with the noreturn **__declspec** modifier. The **return** statement was ignored.

The following sample generates C4645:

```
// C4645.cpp
// compile with:  /W3
void __declspec(noreturn) func() {
   return;   // C4645, delete this line to resolve
}

int main() {
}
```

Compiler Warning (level 3) C4646

Error Message
function declared with __declspec(noreturn) has non-void return type

A function marked with the noreturn **__declspec** modifier should have a void return type.

The following sample generates C4646:

```
// C4646.cpp
// compile with: /W3 /WX
int __declspec(noreturn) TestFunction()
{   // C4646  make return type void
}
```

Compiler Warning (level 1) C4650

Error Message
debugging information not in precompiled header; only global symbols from the header will be available

The precompiled header file was not compiled with Microsoft symbolic debugging information.

When linked, the resulting executable or dynamic-link library file will not include debugging information for local symbols contained in the precompiled header.

This warning can be avoided by recompiling the precompiled header file with the /Zi command-line option.

Compiler Warning (level 1) C4651

Error Message
'definition' specified for precompiled header but not for current compile

The definition was specified when the precompiled header was generated, but not in this compilation.

The definition will be in effect inside the precompiled header, but not in the rest of the code.

If a precompiled header was built with /DSYMBOL, the compiler will generate this warning if the /Yu compile doesn't have /DSYMBOL. Adding /DSYMBOL to the /Yu command line resolves this warning.

Compiler Warning (level 1) C4652

Error Message
compiler option 'option' inconsistent with precompiled header; current command-line option will override that defined in the precompiled header

The given command-line option differed from that given when the precompiled header (.pch) was created. The option specified in the current command line was used.

This warning can be avoided by regenerating the precompiled header with the given command-line option.

Compiler Warning (level 2) C4653

Error Message
compiler option 'option' inconsistent with precompiled header; current command-line option ignored

An option specified with the Use Precompiled Headers (/Yu) option was inconsistent with the options specified when the precompiled header was created. This compilation used the option specified when the precompiled header was created.

This warning can occur when a different value for the Pack Structures option (/Zp) was specified during compilation of the precompiled header.

Compiler Warning (level 1) C4655

' *symbol* ' : variable type is new since the last build, or is defined differently elsewhere

You changed or added a new data type since the last successful build. Edit and Continue does not support changes to existing data types.

This warning is followed by a Fatal Error C1092. For further information, see the Supported Code Changes.

To remove this warning without ending the current debug session

1. Change the data type back to its state prior to the error.
2. From the **Debug** menu, choose **Apply Code Changes**.

To remove this warning without changing your source code

1. From the **Debug** menu, choose **Stop Debugging**.
2. From the **Build** menu, choose **Build**.

Compiler Warning (level 1) C4656

Error Message
'symbol' : data type is new or has changed since the last build, or is defined differently elsewhere

You added or changed a data type, making it new to your source code since the last successful build. Edit and Continue does not support changes to existing data types.

This warning will always be followed by Fatal Error C1092. For further information, see the Supported Code Changes.

To remove this warning without ending the current debug session

1. Change the data type back to its state prior to the error.
2. From the **Debug** menu, choose **Apply Code Changes**.

To remove this error without changing your source code

1. From the **Debug** menu, choose **Stop Debugging**.
2. From the **Build** menu, choose **Build**.

Compiler Warning (level 1) C4657

Error Message
expression involves a data type that is new since the last build

You added or changed a data type, making it new to your source code since the last successful build. Edit and Continue does not support changes to existing data types.

This warning will always be followed by Fatal Error C1092. For further information, see the Supported Code Changes.

To remove this warning without ending the current debug session

1. Change the data type back to its state prior to the error.
2. From the **Debug** menu, choose **Apply Code Changes**.

To remove this error without changing your source code

1. From the **Debug** menu, choose **Stop Debugging**.
2. From the **Build** menu, choose **Build**.

Compiler Warning (level 1) C4659

Error Message
#pragma 'pragma' : use of reserved segment 'segment' has undefined behavior, use #pragma comment(linker, ...)

The .drectve option was used to pass an option to the linker. Instead use pragma comment for passing a linker option.

```
// C4659.cpp
// compile with: /W1 /LD
#pragma code_seg(".drectve")   // C4659
```

Compiler Warning (level 1) C4661

Error Message

'identifier' : no suitable definition provided for explicit template instantiation request

A member of the template class is not defined.

Example

```
// C4661.cpp
// compile with: /W1 /LD
template<class T> class MyClass {
public:
   void i();   // declaration but not definition
};
template MyClass< int >;  // C4661
```

Compiler Warning (level 1) C4662

Error Message
explicit instantiation; template-class 'identifier1' has no definition from which to specialize 'identifier2'

The specified template-class was declared, but not defined.

Example

```
// C4662.cpp
// compile with: /W1 /LD
template<class T, int i> class MyClass; // no definition
template MyClass< int, 1>;              // C4662
```

Compiler Warning (level 1) C4667

Error Message
'function' : no function template defined that matches forced instantiation

You cannot instantiate a function template that has not been declared.

The following sample will cause C4667:

```
// C4667a.cpp
// compile with: /LD /W1
template
void max(const int &, const int &); // C4667 expected
```

To avoid this warning, first declare the function template:

```
// C4667b.cpp
// compile with: /LD
// Declare the function template
template<typename T>
const T &max(const T &a, const T &b) {
   return (a > b) ? a : b;
}
// Then forcibly instantiate it with a desired type ... i.e. 'int'
//
template
const int &max(const int &, const int &);
```

Compiler Warning (level 4) C4668

Error Message
'symbol' is not defined as a preprocessor macro, replacing with '0' for 'directives'

A symbol that was not defined was used with a preprocessor directive. The symbol will evaluate to false. To define a symbol, you can use either the #define directive or /D compiler option.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

Example
The following sample generates C4668:

```
// C4668.cpp
// compile with: /W4
#include <stdio.h>

#pragma warning (default : 4668)   // turn warning on

int main()
{
    #if q   // C4668, q is not defined
        printf_s("defined");
    #else
        printf_s("undefined");
    #endif
}
```

Compiler Warning (level 1) C4669

Error Message
'cast' : unsafe conversion: 'class' is a managed type object

A cast contains a managed type. The compiler completes the cast by performing a bit wise copy of one pointer to the other, but provides no other checking. To resolve this warning, do not cast classes containing managed members.

The following sample generates C4669:

```
// C4669.cpp
// compile with: /clr /W1
ref struct A {
   int i;
   Object ^ pObj;   // remove the managed member to resolve the warning
};

ref struct B {
   int j;
};

int main() {
   A ^ a = gcnew A;
   B ^ b = reinterpret_cast<B ^>(a);   // C4669
}
```

Compiler Warning (level 4) C4670

Error Message
'identifier' : this base class is inaccessible

The specified base class of an object to be thrown in a **try** block is not accessible. The object cannot be instantiated if it is thrown. Check that the base class is inherited with the correct access specifier.

The following sample generates C4670:

```
// C4670.cpp
// compile with: /EHsc /W4
class A
{
};

class B : /* public */ A
{
} b;   // inherits A with private access by default

int main()
{
    try
    {
       throw b;   // C4670
    }
    catch( B )
    {
    }
}
```

Compiler Warning (level 4) C4672

Error Message
'identifier1' : ambiguous. First seen as 'identifier2'

The specified object to be thrown in a **try** block is ambiguous. The object cannot be disambiguated if it is thrown.

Compiler Warning (level 4) C4673

Error Message
throwing 'identifier' the following types will not be considered at the catch site

A throw object cannot be handled in the **catch** block. Each type that cannot be handled is listed in the error output immediately following the line containing this warning. Each unhandled type has its own warning. Read the warning for each specific type for more information.

The following sample generates C4673:

```
// C4673.cpp
// compile with: /EHsc /W4
class Base {
private:
   char * m_chr;
public:
   Base() {
      m_chr = 0;
   }

   ~Base() {
      if(m_chr)
         delete m_chr;
   }
};

class Derv : private Base {
public:
   Derv() {}
   ~Derv() {}
};

int main() {
   try {
      Derv D1;
      // delete previous line, uncomment the next line to resolve
      // Base D1;
      throw D1;   // C4673
   }

   catch(...) {}
}
```

Compiler Warning (level 1) C4674

Error Message
'method' should be declared 'static' and have exactly one parameter

The signature of a conversion operator was not correct. The method is not considered a user-defined conversion.

When using the new syntax (**/clr**), see User-Defined Operators and User-Defined Conversions for information on defining operators.

Example
The following sample generates C4674.

```
// C4674.cpp
// compile with: /clr /WX /W1 /LD
ref class G {
   int op_Implicit(int i) {   // C4674
      return 0;
   }
};
```

The following sample generates C4674.

```
// C4674_b.cpp
// compile with: /clr:oldSyntax /W1 /LD
__gc class G {
   int op_Implicit(int i) {   // C4674
   // try the following line instead
   // static int op_Implicit(int i) {
      return 0;
   }
};
```

Compiler Warning (level 1) C4677

Error Message
'function': signature of non-private member contains assembly private type 'private_type'

A type that has public accessibility outside the assembly uses a type that has private access outside the assembly. A component that references the public assembly type will not be able to use the type member or members that reference the assembly private type.

Example
The following sample generates C4677.

```
// C4677.cpp
// compile with: /clr /c /W1
delegate void TestDel();
public delegate void TestDel2();

public ref class MyClass {
public:
   static event TestDel^ MyClass_Event;   // C4677
   static event TestDel2^ MyClass_Event2;   // OK
};
```

Compiler Warning (level 1) C4678

Error Message
base class 'base_type' is less accessible than 'derived_type'

A public type derives from a private type. If the public type is instantiated in a referenced assembly, members of the private base type will not be accessible.

C4678 is only reachable using **/clr:oldSyntax**. It is an error, using **/clr**, to have a base class that is less accessible that its derived class.

The following sample generates C4678:

```
// C4678.cpp
// compile with: /clr:oldSyntax /LD /W1
#using <mscorlib.dll>
private __gc struct privateG {
// try the following line instead
// public __gc struct privateG {
public:
   int i;
};

public __gc struct V: public privateG {   // C4678
public:
   int j;
};
```

Compiler Warning (level 1) C4679

Error Message

The compiler encountered a construct that it cannot support, that cannot be imported from metadata.

' *member* ' : could not import member

Do not try to use the construct.

Compiler Warning (level 4) C4680

Error Message
'class' : coclass does not specify a default interface

A default interface was not specified for a class that was marked with the coclass attribute. In order for an object to be useful, it must implement an interface.

The following sample generates C4680:

```
// C4680.cpp
// compile with: /W4
#include <windows.h>
[module(name="MyModule")];

[ object, uuid(373a1a4c-469b-11d3-a6b0-00c04f79ae8f) ]
__interface IMyIface1
{
   HRESULT f1();
};

[ object, uuid(37331a4c-469b-11d3-a6b0-00c04f79ae8f) ]
__interface IMyIface2
{
   HRESULT f1();
};

// to resolve C4680, specify a source interface also
// for example, default(IMyIface1, IMyface2)
[ coclass, uuid(373a1a4d-469b-11d3-a6b0-00c04f79ae8f), default(IMyIface1), source(IMyIface1
) ]
class CMyClass : public IMyIface1
{   // C4680
};

int main()
{
}
```

Compiler Warning (level 4) C4681

Error Message
'class' : coclass does not specify a default interface that is an event source

A source interface was not specified for a class.

The following sample generates C4681:

```
// C4681.cpp
// compile with: /W4 /c
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>

[module(name="test")];

[dual, uuid("00000000-0000-0000-0000-000000000000")]
__interface IEvent { [id(3)] HRESULT myEvent(); };

[object, uuid("00000000-0000-0000-0000-000000000001")]
__interface ISource { HRESULT Fire(); };

[ coclass,
  source(IEvent),
  default(ISource),
  // Uncomment the following line to resolve.
  // default(IEvent),
  uuid("00000000-0000-0000-0000-000000000002")
]
struct CSource : ISource {   // C4681
   HRESULT Fire() { return 0; }
};
```

Compiler Warning (level 4) C4682

Error Message
'parameter' : no directional parameter attribute specified, defaulting to [in]

A method on a parameter in an attributed interface does not have one of the directional attributes: in or out. The parameter defaults to in.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4682:

```
// C4682.cpp
// compile with: /W4
#pragma warning(default : 4682)
#include <windows.h>
[module(name="MyModule")];

[ library_block, object, uuid("c54ad59d-d516-41dd-9acd-afda17565c2b") ]
__interface IMyIface : IUnknown
{
   HRESULT f1(int i, int *pi); // C4682
   // try the following line
   // HRESULT f1([in] int i, [in] int *pi);
};

int main()
{
}
```

Compiler Warning (level 1) C4683

' *function* ': event source has an 'out'-parameter; exercise caution when hooking multiple event handlers

If more than one event sink is listening to a COM event source, the value of an out parameter may be ignored.

Be aware that a memory leak will occur in the following situation:

1. If a method has an out parameter that is internally allocated, for example a BSTR *.
2. If the event has more than one handler (is a multicast event)

The reason for the leak is that the out parameter will be set by more than one handler, but returned to the call site only by the last handler.

The following sample generates C4683:

```
// C4683.cpp
// compile with: /W1 /LD
#define _ATL_ATTRIBUTES 1
#include "atlbase.h"
#include "atlcom.h"

[ module(name="xx") ];

[ object ]
__interface I {
   HRESULT f([out] int* pi);
   // try the following line instead
   // HRESULT f(int* pi);
};

[ coclass, event_source(com) ]
struct E {
   __event __interface I;   // C4683
};
```

Compiler Warning (level 1) C4684

Error Message
'attribute' : WARNING!! attribute may cause invalid code generation: use with caution

You used an attribute that should not commonly be used.

The following sample generates C4684:

```
// C4684.cpp
// compile with: /W1 /LD
 [module(name="xx")]; // C4684 expected
[no_injected_text];
```

Compiler Warning (level 1) C4685

Error Message
expecting '> >' found '>>' when parsing template parameters

A template definition was not terminated correctly.

Compiler Warning (level 3) C4686

' *user-defined type* ' : possible change in behavior, change in UDT return calling convention

A class template specialization was not is defined before it was used in a return type. Anything that instantiates the class will resolve C4686; declaring an instance or accessing a member (C<int>::anything) are also options.

This warning is the result of work to make the Visual C++ .NET 2003 compiler conform to the ISO C++ standard.

See Summary of Compiler-Time Breaking Changes for more information.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

Try the following instead,

```
// C4686.cpp
// compile with: /W3
#pragma warning (default : 4686)
template <class T>
class C;

template <class T>
C<T> f(T);

template <class T>
class C {};

int main() {
   f(1);   // C4686
}

template <class T>
C<T> f(T) {
   return C<int>();
}
```

Compiler Warning C4687

Error Message
'class': a sealed abstract class cannot implement an interface 'interface'

A sealed, abstract type is typically only useful to hold static member functions.

For more information, see abstract (Visual C++)and sealed.

C4687 is issued as an error by default. You can suppress C4687 with the warning pragma. If you are certain that you want to implement an interface in a sealed, abstract type, you can suppress C4687.

Example
The following sample generates C4687.

```
// C4687.cpp
// compile with: /clr /c
interface class A {};

ref struct B sealed abstract : A {};   // C4687
ref struct C sealed : A {};   // OK
ref struct D abstract : A {};   // OK
```

Compiler Warning (level 1) C4688

Error Message
'constraint': constraint list contains assembly private type 'type'

A constraint list has an assembly private type, meaning it will not be available when the type is accessed from outside the assembly. For more information, see Generics (Visual C++).

Example
The following sample generates C4688.

```
// C4688.cpp
// compile with: /clr /c /W1
ref struct A {};   // private type
public ref struct B {};

// Delete the following 3 lines to resolve.
generic <class T>
where T : A   // C4688
public ref struct M {};

generic <class T>
where T : B
public ref struct N {};
```

Compiler Warning (level 4) C4690

Error Message
[emitidl(pop)] : more pops than pushes

The emitidl attribute was popped one more time that it was pushed.

Example
The following sample generates C4690.

```
// C4690.cpp
// compile with: /c /W4
[emitidl(pop)];   // C4690
class x {};
```

Compiler Warning (level 1) C4691

Error Message
'type' : type referenced was expected in unreferenced assembly 'file', type defined in current translation unit used instead

The metadata file containing the original type definition is not referenced, and the compiler is using a local type definition.

In the case where you are rebuilding *file*, C4691 can be ignored or turned off with pragma warning. That is, if the file you are building is the same as the file where the compiler expects to find the type definition, you can ignore C4691.

However, unexpected behavior can occur if the compiler uses a definition that is not from the same assembly that is referenced in metadata; CLR types are typed not only by the name of the type, but also by the assembly. That is, a type Z from assembly z.dll is different from a type Z from assembly y.dll.

Example
This sample contains the original type definition.

```
// C4691_a.cpp
// compile with: /clr /LD /W1
public ref class Original_Type {};
```

This sample references C4691_a.dll and declares a field of type Original_Type.

```
// C4691_b.cpp
// compile with: /clr /LD
#using "C4691_a.dll"
public ref class Client {
public:
   Original_Type^ ot;
};
```

The following sample generates C4691. Notice this sample contains a definition for Original_Type and does not reference C4691a.dll.

To resolve, reference the metadata file that contains the original type definition and remove the local declaration and definition.

```
// C4691_c.cpp
// compile with: /clr /LD /W1
// C4691 expected

// Uncomment the following line to resolve.
// #using "C4691_a.dll"
#using "C4691_b.dll"

// Delete the following line to resolve.
ref class Original_Type;

public ref class MyClass : Client {};
```

Compiler Warning (level 1) C4692

Error Message
'function': signature of non-private member contains assembly private native type 'native_type'

A type that is visible outside the assembly contains a member function whose signature contains a native type that is not visible outside the assembly. Therefore, the member function should not be called if its containing type is instantiated outside the assembly.

For more information, see Type Visibility.

This warning is off by default. For more information, see Compiler Warnings That Are Off by Default.

Example
The following sample generates C4692.

```
// C4692.cpp
// compile with: /W1 /c /clr
#pragma warning(default:4692)
class Private_Native_Class {};
public class Public_Native_Class {};
public ref class Public_Ref_Class {
public:
   void Test(Private_Native_Class *) {}   // C4692
   void Test2(Public_Native_Class *) {}   // OK
};
```

Compiler Warning C4693

Error Message

'class': a sealed abstract class cannot have any instance members 'Test'

If a type is marked sealed and abstract (Visual C++), it can only have static members.

Example

The following sample generates C4693.

```
// C4693.cpp
// compile with: /clr /c
public ref class Public_Ref_Class sealed abstract {
public:
   void Test() {}   // C4693
   static void Test2() {}   // OK
};
```

Compiler Warning C4694

Error Message
'class_1': a sealed abstract class cannot have a base-class 'base_class'

An abstract and sealed class cannot inherit from a reference type; a sealed and abstract class can neither implement the base class functions nor allow itself to be used as a base class.

For more information, see abstract (Visual C++), sealed, and Classes and Structs (Managed).

Example
The following sample generates C4694.

```
// C4694.cpp
// compile with: /c /clr
ref struct A {};
ref struct B sealed abstract : A {};   // C4694
```

Compiler Warning (level 1 and level 4) C4700

Error Message
uninitialized local variable 'name' used

You used the local variable *name* without first assigning it a value, which could lead to unpredictable results.

The following sample generates C4700:

```
// C4700.cpp
// compile with: /W1
int main() {
   int i;
   return i;   // C4700
}
```

Under /clr:safe this is a level 4 warning. The following sample generates C4700:

```
// C4700b.cpp
// compile with: /W4 /clr:safe /c
using namespace System;
int main() {
   Int32^ bi;
   return *bi;   // C4700
}
```

Compiler Warning (level 4) C4701

Error Message
Potentially uninitialized local variable 'name' used

You may have used the local variable *name* without first assigning it a value, which could lead to unpredictable results.

Compiler Warning (level 4) C4702

Error Message
unreachable code

This warning is the result of compiler conformance work that was done for Visual Studio .NET 2003: unreachable code. When the compiler (back end) detects unreachable code, it will generate C4702, a level 4 warning.

For code that is valid in both the Visual Studio .NET 2003 and Visual Studio .NET versions of Visual C++, remove the unreachable code or assure that all source code is reachable by some flow of execution.

See Summary of Compiler-Time Breaking Changes for more information.

Example

The following sample generates C4702.

```
// C4702.cpp
// compile with: /W4
#include <stdio.h>

int main() {
   return 1;
   printf_s("I won't print.\n");   // C4702 unreachable
}
```

When compiling with **/GX**, **/EHc**, **/EHsc**, or **/EHac** and using extern C functions, code can become unreachable because extern C functions are assumed to not throw, thus the catch block isn't reachable. If you feel that this warning is not valid because a function can throw, compile with **/EHa** or **/EHs**, depending on the exception thrown.

For more information, see /EH (Exception Handling Model) for more information.

The following sample generates C4702.

```
// C4702b.cpp
// compile with: /W4 /EHsc
#include <iostream>

using namespace std;
extern "C" __declspec(dllexport) void Function2(){}

int main() {
   try {
      Function2();
   }
   catch (...) {
      cout << "Exp: Function2!" << endl;   // C4702
   }
}
```

Compiler Warning (level 4) C4706

Error Message
assignment within conditional expression

The test value in a conditional expression was the result of an assignment.

An assignment has a value (the value on the left side of the assignment) that can be used legally in another expression, including a test expression.

The following sample generates C4706:

```
// C4706a.cpp
// compile with: /W4
int main()
{
   int a = 0, b = 0;
   if ( a  = b ) // C4706
   {
   }
}
```

The warning will occur even if you double the parentheses around the test condition:

```
// C4706b.cpp
// compile with: /W4
int main()
{
   int a = 0, b = 0;
   if ( ( a  =  b ) ) // C4706
   {
   }
}
```

If your intention is to test a relation and not to make an assignment, use the == operator. For example, the following line tests whether a and b are equal:

```
// C4706c.cpp
// compile with: /W4
int main()
{
   int a = 0, b = 0;
   if ( a == b )
   {
   }
}
```

If you intend to make your test value the result of an assignment, test to ensure that the assignment is non-zero or not null. For example, the following code will not generate this warning:

```
// C4706d.cpp
// compile with: /W4
int main()
{
   int a = 0, b = 0;
   if ( ( a = b ) != 0 )
   {
   }
}
```

Compiler Warning (level 4) C4709

Error Message
comma operator within array index expression

When a comma occurs in an array index expression, the compiler uses the value after the last comma.

Example
The following sample generates C4709:

```
// C4709.cpp
// compile with: /W4
#include <stdio.h>

int main()
{
    int arr[2][2];
    arr[0][0] = 10;
    arr[0][1] = 11;

    // Prints 10, not 11
    printf_s("\n%d",arr[0][1,0]);   // C4709
}
```

Compiler Warning (level 4) C4710

Error Message
'function' : function not inlined

The given function was selected for inline expansion, but the compiler did not perform the inlining.

Inlining is performed at the compiler's discretion. The **inline** keyword, like the **register** keyword, is used as a hint for the compiler. The compiler uses heuristics to determine if it should inline a particular function to speed up the code when compiling for speed, or if it should inline a particular function to make the code smaller when compiling for space. The compiler will only inline very small functions when compiling for space.

In some cases, the compiler will not inline a particular function for mechanical reasons. See C4714 for a list of reasons the compiler may not inline a function.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

Compiler Warning (level 1) C4711

Error Message
function 'function' selected for inline expansion

The compiler performed inlining on the given function, although it was not marked for inlining.

C4711 is enabled if /Ob2 is specified.

Inlining is performed at the compiler's discretion. This warning is informational.

This warning is off by default. To enable a warning, use #pragma warning. See Compiler Warnings That Are Off by Default for more information.

Compiler Warning (level 4) C4714

Error Message
function 'function' marked as __forceinline not inlined

The given function was selected for inline expansion, but the compiler did not perform the inlining.

Although **__forceinline** is a stronger indication to the compiler than **__inline**, inlining is still performed at the compiler's discretion, but no heuristics are used to determine the benefits from inlining this function.

In some cases, the compiler will not inline a particular function for mechanical reasons. For example, the compiler will not inline:

- A function if it would result in mixing both SEH and C++ EH.
- Some functions with copy constructed objects passed by value when -GX/EHs/EHa is on.
- Functions returning an unwindable object by value when -GX/EHs/EHa is on.
- Functions with inline assembly when compiling without -Og/Ox/O1/O2.
- Functions with a variable argument list.
- A function with a **try** (C++ exception handling) statement.

The following sample generates C4714:

```
// C4714.cpp
// compile with: /Ob1 /GX /W4
__forceinline void func1()
{
   try
   {
   }
   catch (...)
   {
   }
}


void func2()
{
   func1();   // C4714
}

int main()
{
}
```

Compiler Warning (level 1) C4715

Error Message
'function' : not all control paths return a value

The specified function can potentially not return a value.

Example

```
// C4715a.cpp
// compile with: /W1 /LD
int func1( int i )
{
   if( i )
   return 3;  // C4715 warning, nothing returned if i == 0
}
```

To prevent this warning, modify the code so that all paths assign a return value to the function:

```
// C4715b.cpp
// compile with: /LD
int func1( int i )
{
   if( i ) return 3;
   else return 0;     // OK, always returns a value
}
```

It is possible that your code may contain a call to a function that never returns, as in the following example:

```
// C4715c.cpp
// compile with: /W1 /LD
void fatal()
{
}
int glue()
{
   if(0)
      return 1;
   else if(0)
      return 0;
   else
      fatal();   // C4715
}
```

This code also generates a warning, because the compiler does not know that `fatal` never returns. To prevent this code from generating an error message, declare `fatal` using __declspec(noreturn).

Compiler Warning (level 1) C4716

Error Message
'function' must return a value

The given function did not return a value.

Only functions with a return type of void can use the return command without an accompanying return value.

An undefined value will be returned when this function is called.

This warning is automatically promoted to an error. If you wish to modify this behavior, use #pragma warning.

The following sample generates C4716:

```
// C4716.cpp
// compile with: /c /W1
// C4716 expected
#pragma warning(default:4716)
int test() {
   // uncomment the following line to resolve
   // return 0;
}
```

Compiler Warning (level 1) C4717

Error Message
'function' : recursive on all control paths, function will cause runtime stack overflow

Every path through a function contains a call to the function. Since there is no way to exit the function without first calling itself recursively, the function will never exit.

The following sample generates C4717:

```
// C4717.cpp
// compile with: /W1 /c
// C4717 expected
int func(int x) {
   if (x > 1)
      return func(x - 1); // recursive call
   else {
      int y = func(0) + 1; // recursive call
      return y;
   }
}

int main(){
   func(1);
}
```

Compiler Warning (level 4) C4718

Error Message
'function call' : recursive call has no side effects, deleting

A function contains a recursive call, but otherwise has no side effects. A call to this function is being deleted. The correctness of the program is not affected, but the behavior is. Whereas leaving the call in could result in a runtime stack overflow exception, deleting the call removes that possibility.

Compiler Warning (level 1) C4722

Error Message
'function' : destructor never returns, potential memory leak

The flow of control terminates in a destructor. The thread or the entire program will terminate and allocated resources may not be released. Furthermore, if a destructor will be called for stack unwinding during exception processing, the behavior of executable is undefined.

To resolve, remove the function call that causes the destructor to not return.

Example
The following sample generates C4722:

```
// C4722.cpp
// compile with: /O1 /W1 /c
#include <stdlib.h>
class C {
public:
   C();
   ~C() { exit(1); };   // C4722
};

extern void func (C*);

void Test(){
   C x;
   func(&x);
   // control will not leave Test because destructor will exit
}
```

Compiler Warning (level 3) C4723

Error Message
potential divide by 0

The second operand in a divide operation evaluated to zero at compile time, giving undefined results.

This warning is issued only when using /Og or an optimization option that implies /Og.

The compiler may have generated the zero operand.

Compiler Warning (level 3) C4724

Error Message
potential mod by 0

The second operand in a remainder operation evaluated to zero at compile time, giving undefined results.

Compiler Warning (level 4) C4725

Error Message
instruction may be inaccurate on some Pentiums

Your code contains an inline assembly instruction that may not produce accurate results on some Pentium microprocessors.

The following sample generates C4725:

```
// C4725.cpp
// compile with: /W4
// processor: x86
double m32fp = 2.0003e-17;

void f() {
   __asm
   {
      FDIV m32fp   // C4725
   }
}

int main() {
}
```

Compiler Warning (level 1) C4727

Error Message
"PCH named pch_file with same timestamp found in obj_file_1 and obj_file_2. Using first PCH.

C4727 occurs when compiling multiple compilands with **/Yc**, and where the compiler was able to mark all .obj files with the same .pch timestamp.

To resolve, compile one source file with **/Yc /c** (creates pch), and the others compile separately with **/Yu /c** (uses pch), then link them together.

So, if you did the following and generates C4727:

cl /clr /GL a.cpp b.cpp c.cpp /Ycstdafx.h

You would do the following instead:

cl /clr /GL a.cpp /Ycstdafx.h /c

cl /clr /GL b.cpp c.cpp /Yustdafx.h /link a.obj

For more information, see

- /Yc (Create Precompiled Header File)
- /Yu (Use Precompiled Header File)

Compiler Warning (Level 1) C4729

Error Message
function too big for flow graph based warnings

This warning is generated when a function is too big to be compiled with reliable checking for situations that would generate a warning. This warning is only generated when the /Od compiler option used.

To resolve this warning, break the function into smaller functions.

Compiler Warning (Level 1) C4730

Error Message
'main' : mixing _m64 and floating point expressions may result in incorrect code

A function uses __m64 and **float/double** types. Because the MMX and floating-point registers share the same physical register space (cannot be used simultaneously), using **__m64** and **float/double** types in the same function can result in data corruption, possibly causing an exception.

To safely use **__m64** types and floating-point types in the same function, each instruction that uses one of the types should be separated by the **_m_empty()** (for MMX) or **_m_femms()** (for 3DNow!™) intrinsic.

The following sample generates C4730:

```
// C4730.cpp
// compile with: /W1
// processor: x86
#include "mmintrin.h"

void func(double)
{
}

int main(__m64 a, __m64 b)
{
   __m64 m;
   double f;
   f = 1.0;
   m = _m_paddb(a, b);
   // uncomment the next line to resolve C4730
   // _m_empty();
   func(f * 3.0);   // C4730
}
```

Compiler Warning (Level 1) C4731

Error Message
'pointer' : frame pointer register 'register' modified by inline assembly code

A frame pointer register was modified. You must save and restore the register in your inline assembly block or frame variable (local or parameter, depending on the register modified), or your code may not work properly.

The following sample generates C4731:

```
// C4731.cpp
// compile with: /W1 /LD
// processor: x86
// C4731 expected
void bad(int p) {
   __asm
   {
      mov ebp, 1
   }

   if (p == 1)
   {
      // ...
   }
}
```

EBP is the frame pointer (FPO is disallowed) and it is being modified. When `p` is later referenced, it is referenced relative to `EBP`. But `EBP` has been overwritten by the code, so the program will not work properly and may even fault.

Compiler Warning (Level 1) C4733

Error Message
Inline asm assigning to 'FS:0' : handler not registered as safe handler

A function modifying the value at FS:0 to add a new exception handler may not work with Safe Exceptions, because the handler may not be registered as a valid exception handler (see /SAFESEH).

To resolve this warning, either remove the FS:0 definition or turn off this warning and use .SAFESEH to specify the safe exception handlers.

The following sample generates C4733:

```
// C4733.cpp
// compile with: /W1
// processor: x86
#include "stdlib.h"
#include "stdio.h"
void my_handler()
{
   printf("Hello from my_handler\n");
   exit(1);
}

int main()
{
   _asm {
      push    my_handler
      mov     eax, DWORD PTR fs:0
      push    eax
      mov     DWORD PTR fs:0, esp   // C4733
   }

   *(int*)0 = 0;
}
```

Compiler Warning (Level 1) C4734

Error Message
More than 64k line numbers in a COFF debug info section; stop emitting COFF debug line numbers for module 'module'

COFF format does not allow storing more than 64K lines in COFF debug info.

To resolve this warning, do not emit debug info for ***module***, or decrease the amount of code in the file by putting the code into two files.

Compiler Warning (Level 3) C4738

Error Message
storing 32-bit float result in memory, possible loss of performance

C4738 warns that the result of an assignment, cast, passed argument, or other operation may need to be rounded or that the operation ran out of registers and needed to use memory (spilling). This can result in performance loss.

To resolve this warning and avoid rounding, compile with /fp:fast or use **double**s instead of **float**s.

To resolve this warning and avoid running out of registers, change the order of computation and modify your use of inlining

This warning is off by default. For more information, see Compiler Warnings That Are Off by Default.

Example
The following sample generates C4738:

```
// C4738.cpp
// compile with: /c /fp:precise /O2 /W3
// processor: x86
#include <stdio.h>

#pragma warning(default : 4738)

float func(float f)
{
    return f;
}

int main()
{
    extern float f, f1, f2;
    double d = 0.0;

    f1 = func(d);
    f2 = (float) d;
    f = f1 + f2;   // C4738
    printf_s("%f\n", f);
}
```

Compiler Warning (Level 1) C4739

Error Message
reference to variable 'var' exceeds its storage space

A value was assigned to a variable, but the value is greater than the size of the variable. Memory will be written beyond the variable's memory location, and data loss is possible.

To resolve this warning, only assign a value to a variable whose size can accommodate the value.

The following sample generates C4739:

```
// C4739.cpp
// compile with: /RTCs /Zi /W1 /c
char *pc;
int main() {
   char c;
   *(int *)&c = 1;   // C4739

   // OK
   *(char *)&c = 1;
}
```

Compiler Warning (Level 4) C4740

Error Message
flow in or out of inline asm code suppresses global optimization

When there is a jump in to or out of an **asm** block, global optimizations are disabled for that function.

The following sample generates C4740:

```
// C4740.cpp
// compile with: /O2 /W4
// processor: x86
int main() {
   __asm jmp tester
   tester:;
}
```

Compiler Warning (Level 1) C4742

Error Message
'var' has different alignment in 'file1' and 'file2': number and number

An external variable that was referenced or defined in two files has different alignment in those files. This warning is emitted when compiler finds that **__alignof** for the variable in *file1* differs from **__alignof** for the variable in *file2*. This can be caused by using incompatible types when declaring variable in different files, or by using non-matching **#pragma pack** in different files.

To resolve this warning, either use the same type definition or use different names for the variables.

For more information, see pack and __alignof Operator.

Example

This is the first file that defines the type.

```
// C4742a.c
// compile with: /c
struct X {
   char x, y, z, w;
} global;
```

The following sample generates C4742.

```
// C4742b.c
// compile with: C4742a.c /W1 /GL
// C4742 expected
extern struct X {
   int a;
} global;

int main() {
   global.a = 0;
}
```

Compiler Warning (Level 1) C4743

'*type*' has different size in '*file1*' and '*file2*': *number* and *number* bytes

An external variable referenced or defined in two files has different types in those files, and the compiler determined that the size of the variable in *file1* differs from the size of the variable in *file2*.

There is important case when this warning can be emitted for C++. If you declare the same types with the same name in two different files, if those declarations contain virtual functions, and if the declarations are not the same, then the compiler can emit warning C4744 for the virtual function tables. The warning occurs because there are two different sized virtual function tables for the same type, and linker must choose one of them to incorporate into the executable. It is possible that this can result in your program calling the wrong virtual function.

To resolve this warning, either use the same type definition or use different names for the types or variables.

Example

This sample contains one definition of the type.

```
// C4743a.cpp
// compile with: /c
class C {
public:
    virtual void f1(void);
    virtual void f2(void);
    virtual void f3(void);
};

void C::f1(void) {}
void C::f2(void) {}
void C::f3(void) {}
C q;
```

The following sample generates C4743.

```
// C4743b.cpp
// compile with: C4743a.cpp /W1 /GL /O2
// C4743 expected
class C {
public:
    virtual void f1(void);
    virtual void f2(void);
    virtual void f3(void);
    virtual void f4(void);
    virtual void f5(void);
};

void C::f4(void) {}
void C::f5(void) {}
C x;

int main() {}
```

Compiler Warning (Level 1) C4744

Error Message
'var' has different type in 'file1' and 'file2': 'type1' and 'type2'

An external variable referenced or defined in two files has different types in those files. To resolve, either make the type definitions the same, or change variable name in one of the files.

C4744 is emitted only when files are compiled with /GL. For more information, see /GL (Whole Program Optimization).

Note
C4744 usually occurs in C (not C++) files, because in C++ a variable name is decorated with type information. When the sa mple (below) is compiles as C++, you'll get linker error LNK2019.

Example

This sample contains the first definition.

```
// C4744.c
// compile with: /c /GL
int global;
```

The following sample generates C4744.

```
// C4744b.c
// compile with: C4744.c /GL /W1
// C4744 expected
#include <stdio.h>

extern unsigned global;

main()
{
    printf_s("%d\n", global);
}
```

Compiler Warning (level 1) C4747

Error Message
Calling managed 'entrypoint': Managed code may not be run under loader lock, including the DLL entrypoint and calls reached from the DLL entrypoint

The compiler found a (probable) DLL entry point compiled to MSIL. Because of potential problems with loading a DLL whose entry point has been compiled to MSIL, you are strongly discouraged from compiling a DLL entry point function to MSIL.

For more information, see Initialization of Mixed Assemblies and Linker Tools Error LNK1306.

To correct this error

1. Do not compile the module with **/clr**.
2. Mark the entry point function with **#pragma unmanaged**.

Example
The following sample generates C4747.

```
// C4747.cpp
// compile with: /clr /c /W1
// C4747 expected
#include <windows.h>

// Uncomment the following line to resolve.
// #pragma unmanaged

BOOL WINAPI DllMain(HANDLE hInstance, ULONG Command, LPVOID Reserved) {
   return TRUE;
};
```

Compiler Warning (level 3) C4748

Error Message
/GS can not protect parameters and local variables from local buffer overrun because optimizations are disabled in function

/GS (Buffer Security Check), which is enabled by default, cannot protect parameters and local variables from local buffer overrun in a function unless the function has optimizations enabled.

The compiler will disable optimizations if a function has inline assembly code containing flow of control (jmp or jcc, for example) statements.

Enable optimizations to resolve this warning, and to allow **/GS** to protect parameters and local variables from local buffer overrun.

Example
The following sample generates C4748.

```
// C4748.cpp
// compile with: /O2 /W3
#include <string.h>
#include <stdlib.h>

#pragma optimize("", off)
void f(const char *str) {  // C4748 warning
   char buf[100];
   strcpy_s(buf, _countof(buf), str);
}
#pragma optimize("", on)

int main() {
   f("aa");
}
```

Compiler Warning C4750

Error Message
'identifier': function with _alloca() inlined into a loop

The 'identifier' function forces inline expansion of the _alloca function within a loop, which might cause a stack overflow when the loop is executed.

To correct this error

1. Ensure that the 'identifier' function is not modified with the __forceinline specifier.
2. Ensure that the 'identifier' function does not contain a _alloca function that is contained in a loop.
3. Do not specify the /O1, /O2, /Ox, or /Og compilation switch.
4. Place the _alloca function in a try-except statement that will catch a stack overflow.

Example

The following code example calls `MyFunction` in a loop, and `MyFunction` calls the **_alloca** function. The **__forceinline** modifier causes the inline expansion of the **_alloca** function.

```
// c4750.cpp
// compile with: /O2 /W2 /c
#include <intrin.h>

char * volatile newstr;

__forceinline void myFunction(void) // C4750
{
// The _alloca function does not require a __try/__except
// block because the example uses compiler option /c.
    newstr = (char * volatile) _alloca(1000);
}

int main(void)
{
    for (int i=0; i<50000; i++)
       myFunction();
    return 0;
}
```

Compiler Warning (level 2) C4756

Error Message
overflow in constant arithmetic

The compiler generated an exception while doing constant arithmetic during compilation.

The following sample generates C4756:

```
// C4756.cpp
// compile with: /W2 /Od
int main()
{
   float f = 1e100+1e100;   // C4756
}
```

Compiler Warning (level 4) C4764

Error Message
Cannot align catch objects to greater than 16 bytes

An alignment greater than 16 was specified, but on some platforms, if the function throws an exception, the stack will force an alignment of not greater than 16.

Example
The following sample generates C4764:

```
// C4764.cpp
// compile with: /W4 /EHsc
// processor: x64 IPF
#include <stdio.h>

class A
{
public:
    int x;
};

typedef __declspec(align(32)) A ALIGNEDA;

int main()
{
    ALIGNEDA a;
    try
    {
        a.x = 15;
        throw a;
    }
    catch (ALIGNEDA b) // can’t align b to > 16 bytes
    {
        printf_s("%d\n", b.x);
    }
}   // C4764
```

Compiler Warning (level 1) C4772

Error Message
#import referenced a type from a missing type library; 'missing_type' used as a placeholder

A type library was referenced with the #import directive. However, the type library contained a reference to another type library that was not referenced with **#import**. This other .tlb file was not found by the compiler.

Note that the compiler will not find type libraries in different directories if you use the /I (Additional Include Directories) compiler option to specify those directories. If you want the compiler to find type libraries in different directories, add those directories to the PATH environment variable.

This warning is, by default, issued as an error. C4772 can not be suppressed with /W0.

Example

This is the first type library needed to reproduce C4772.

```
// c4772a.idl
[uuid("f87070ba-c6d9-405c-a8e4-8cd9ca25c12b")]
library C4772aLib
{
   [uuid("f87070ba-c6d9-405c-a8e4-8cd9ca25c100")]
   enum E_C4772a
   {
      one, two, three
   };
};
```

This is the second type library needed to reproduce C4772.

```
// c4772b.idl
// post-build command: del /f C4772a.tlb
// C4772a.tlb is available when c4772b.tlb is built
[uuid("f87070ba-c6d9-405c-a8e4-8cd9ca25c12d")]
library C4772bLib
{
   importlib ("c4772a.tlb");
   [uuid("f87070ba-c6d9-405c-a8e4-8cd9ca25c12e")]
   struct S_C4772b
   {
      enum E_C4772a e;
   };
};
```

The following sample generates C4772:

```
// C4772.cpp
// assumes that C4772a.tlb is not available to the compiler
// #import "C4772a.tlb"
#import "C4772b.tlb"   // C4772 uncomment previous line to resolve
                       // and make sure c4772a.tlb is on disk
```

Compiler Warning (Level 1) C4788

Error Message
'identifier' : identifier was truncated to 'number' characters

The compiler limits the maximum length allowed for a function name. When the compiler generates funclets for EH/SEH code, it forms the funclet name by prepending the function name with some text, for example "__catch", "__unwind", or another string.

The resulting funclet name can be too long, and the compiler will truncate it and generate C4788.

To resolve this warning, shorten the original function name. If the function is a C++ template function or method, use a typedef for part of the name. For example:

```
C1<x, y, z<T>>::C2<a,b,c>::f
```

can be replaced by:

```
typedef C1<x, y, z<T>>::C2<a,b,c> new_class ;
new_class::f
```

This warning only occurs in the Itanium Processor Family (IPF) and x64 compiler.

Compiler Warning (Level 1) C4789

Error Message
destination of memory copy is too small

Data whose length is known at compile time was copied into a data block whose size is also known at compile time and whose size is too small for the data, and where the copy was done by the intrinsic form of one of the following CRT functions:

- strcpy
- memset
- memcpy, wmemcpy

Compiling with /Oi- or /Od will resolve this warning

Example
The following sample generates C4789.

```
// C4789.cpp
// compile with: /Oi /W1 /c
#include <string.h>
#include <stdio.h>

int main()
{
    char a[20];
    strcpy(a, "000000000000000000000000\n");   // C4789

    char buf2[20];
    memset(buf2, 'a', 21);   // C4789

    char c;
    wchar_t w = 0;
    memcpy(&c, &w, sizeof(wchar_t));
}
```

The following sample generates C4789.

```
// C4789b.cpp
// compile with: /W1 /O2
// processor: x86
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main()
{
    // /O2 collapses this loop on x86 into a memset intrinsic and warn
    // on Itanium, only explicit memset calll will generate C4789
    char buf[98];

    memset(buf, 0, sizeof(buf)+1);
    for ( int i = 0; i < 99; i++ )
        buf[i] = 'a';   // C4789
    printf_s("%c\n", buf[0]);
}
```

Compiler Warning (level 3) C4792

Error Message
function 'function' declared using sysimport and referenced from native code; import library required to link

A native function that was imported into the program with DllImport was called from an unmanaged function. Therefore, you must link to the import library for the DLL.

This warning cannot be resolved in code or by changing the way you compile. Use the warning pragma to disable this warning.

The following sample generates C4792:

```
// C4792.cpp
// compile with: /clr /W3
// C4792 expected
using namespace System::Runtime::InteropServices;
[DllImport("msvcrt")]
extern "C" int __cdecl puts(const char *);
int main() {}

// Uncomment the following line to resolve.
// #pragma warning(disable : 4792)
#pragma unmanaged
void func(void){
   puts("test");
}
```

Compiler Warning (level 1 and 3) C4793

Error Message
'reason' : causes native code generation for function 'function'

function was compiled to unmanaged, native code, even though the file was compiled /clr. The compiler will not be able to create an MSIL version of the function if the function includes setjmp, inline assembly, or a parameter list that includes a variable number of parameters.

This is a level 1 warning when compiling with **/clr:pure**.

Example

The following sample generates C4793.

```
// C4793.cpp
// compile with: /clr /W3 /c
// processor: x86
int asmfunc(void) {   // C4793, compiled as unmanaged, native code
   __asm {
      mov eax, 0
   }
}
```

The following sample generates C4793.

```
// C4793_b.cpp
// compile with: /clr:pure /WX
#include <setjmp.h>
jmp_buf test_buf;

void f() {
   setjmp(test_buf)   // C4793 warning
}
```

Compiler Warning (level 1) C4794

Error Message
segment of thread local storage variable 'variable' changed from 'section name' to '.tls$'

You used #pragma data_seg to put a tls variable in a section not starting with .tls$.

The .tls$*x* section will exist in the object file where __declspec(thread) variables are defined. A .tls section in the EXE or DLL will result from these sections.

Example
The following sample generates C4794:

```
// C4794.cpp
// compile with: /W1 /c
#pragma data_seg(".someseg")
__declspec(thread) int i;   // C4794

// OK
#pragma data_seg(".tls$9")
__declspec(thread) int j;
```

Compiler Warning (level 1) C4799

Error Message
No EMMS at end of function 'function'

The function has at least one MMX instruction, but does not have an EMMS instruction. When a multimedia instruction is used, an EMMS instruction should also be used to clear the multimedia tag word at the end of the MMX code. For more information on EMMS instructions, see Guidelines for When to Use EMMS.

You may get C4799 when using ivec.h, indicating that the code does not use properly execute the EMMS instruction before returning. This is a false warning for these headers. You may turn these off by defining _SILENCE_IVEC_C4799 in ivec.h. However, be aware that this will also keep the compiler from giving correct warnings of this type.

For related information, see the Intel's MMX Instruction Set.

Compiler Warning (level 3) C4800

Error Message
'type' : forcing value to bool 'true' or 'false' (performance warning)

This warning is generated when a value that is not **bool** is assigned or coerced into type **bool**. Typically, this message is caused by assigning **int** variables to **bool** variables where the **int** variable contains only values **true** and **false**, and could be redeclared as type **bool**. If you cannot rewrite the expression to use type **bool**, then you can add "`!=0`" to the expression, which gives the expression type **bool**. Casting the expression to type **bool** will not disable the warning, which is by design.

The following sample generates C4800:

```
// C4800.cpp
// compile with: /W3
int main() {
   int i = 0;

   // try..
   // bool i = 0;

   bool j = i;   // C4800
   j++;
}
```

Compiler Warning C4801

Error Message
Return by reference is not verifiable: reason

You cannot store a tracking reference into a local variable and then return it verifiably.

A reference can only be verifiably returned when it can be tracked by the verifier from creation to return point and when it is a reference to an element of an array, or a member of a class.

For more information, see PEVerify Tool (Peverify.exe).

A reference must remain on the stack from creation to return in order to be verifiable.

C4801 is always issued as an error. You can turn off this warning with the **#pragma warning** or **/wd**; see warning or /w, /Wn, /WX, /Wall, /wln, /wdn, /wen, /won (Warning Level) for more information.

Example

C4801 will be generated if the verifier cannot see the address taken, so it must assume that it might be an unverifiable reference. The following sample generates C4801.

```
// C4801.cpp
// compile with: /clr:safe /c
int% f(int% p) {
   return p;   // C4801
}
```

The following sample generates C4801.

```
// C4801_b.cpp
// compile with: /clr:safe /c

int% f(int i, array<int>^ ar) {
   int x;
   int% bri = x;   // cannot return a byref to a local.
   if (i < ar->Length) {
      bri = ar[i];   // this byref is ok.
   }

   return bri;   // C4801 not returned within the basic block
}
```

C4801 can also occur if you attempt to dereference and return a reference value in a try block. This will result in code that cannot be verified because the stack is cleared at the end of a try block, destroying any return value on the stack. Instead, dereference the reference type and assign it to a variable, to ensure that no exception is thrown. Then, at the end of the function, dereference the reference type again and return it.

The following sample generates C4801.

```
// C4801_c.cpp
// compile with: /clr:safe
using namespace System;

ref class StackEmptyException : public System::Exception {};
ref class StackFullException : public System::Exception {};

template <typename T>
ref struct Stack {

   Stack() {
      topElem = -1;   // initialize stack
      stackPtr = gcnew array<T>(10);
   }
```

```
   void push(const T%);
   T% top();

private:
   int topElem;
   array<T>^ stackPtr;
};

template <typename T>
T% Stack<T>::top() {
   try {
      return stackPtr[topElem];   // C4801
      // Try the following line instead.
      // T% deadstore = stackPtr[topElem] ;
   }

   catch (System::IndexOutOfRangeException ^ e) {
      throw gcnew StackEmptyException();
   }

   catch (System::Exception ^ e) {
      throw;
   }

   return stackPtr[topElem] ;
}

int main() {
   typedef Stack<Int32> IntStack;
   IntStack ^ is = gcnew IntStack();

   int i = 1;
   while (1) {
      try {
         is->push(i++);
      }
      catch (System::Exception ^ e) {
         break;
      }
      Console::Write("{0} ", is->top());
   }
}
```

Compiler Warning (level 1) C4803

Error Message
'method' : the raise method has a different storage class from that of the event, 'event'

Event methods must have the same storage class as the event declaration. The compiler adjusts the event's methods so that the storage classes are the same.

This warning can occur if you have a class that implements an event from an interface. The compiler does not implicitly generate a raise method for an event in an interface. When you implement that interface in a class, the compiler does implicitly generate a raise method and that method will not be virtual, hence the warning.

See warning pragma for information on how to turn a warning off.

Example

C4803 can be generated using **/clr**. See event (Visual C++) for more information on using events..

The following sample generates C4803.

```
// C4803.cpp
// compile with: /clr /W1
using namespace System;

public delegate void Del();

ref struct E {
   Del ^ _pd1;
   event Del ^ E1 {
      void add (Del ^ pd1) {
         _pd1 = dynamic_cast<Del ^>(Delegate::Combine(_pd1, pd1));
      }

      void remove(Del^ pd1) {
         _pd1 = dynamic_cast<Del^> (Delegate::Remove(_pd1, pd1));
      }

      virtual void raise() {   // C4803 warning (remove virtual)
         if (_pd1)
            _pd1();
      }
   }

   void func() {
      Console::WriteLine("In E::func()");
   }
};

int main() {
   E ^ ep = gcnew E;
   ep->E1 += gcnew Del(ep, &E::func);
   ep->E1();
   ep->E1 -= gcnew Del(ep, &E::func);
   ep->E1();
}
```

C4803 can be generated using **/clr:oldSyntax**. The following sample generates C4803.

```
// C4803_b.cpp
// compile with: /clr:oldSyntax /W1
using namespace System;

public __delegate void Del();

__gc struct E {
```

```
   Del* _pd1;
   virtual __event void add_E1(Del* pd1) {
      _pd1 = dynamic_cast<Del*> (Delegate::Combine(_pd1, pd1));
   }

   virtual __event void remove_E1(Del* pd1) {
      _pd1 = dynamic_cast<Del*> (Delegate::Remove(_pd1, pd1));
   }

   __event void raise_E1 () {   // C4803, add virtual
      if (_pd1)
         _pd1->Invoke();
   }

   void func() {
      Console::WriteLine("In E::func()");
   }
};

int main() {
   E* ep = new E;
   ep->E1 += new Del(ep, &E::func);
   ep->E1();
   ep->E1 -= new Del(ep, &E::func);
   ep->E1();
}
```

Compiler Warning (level 1) C4804

Error Message
'operation' : unsafe use of type 'bool' in operation

This warning is for when you used a **bool** variable or value in an unexpected way. For example, C4804 is generated if you use operators such as the negative unary operator (**-**) or the complement operator (**~**). The compiler evaluates the expression.

Example

The following sample generates C4804:

```
// C4804.cpp
// compile with: /W1

int main()
{
   bool i = true;
   if (-i)   // C4804, remove the '-' to resolve
   {
      i = false;
   }
}
```

Compiler Warning (level 1) C4805

Error Message
'operation' : unsafe mix of type 'type' and type 'type' in operation

This warning is generated for comparison operations between bool and int. The following sample generates C4805:

```
// C4805.cpp
// compile with: /W1
int main() {
   int i = 1;
   bool b = true;

   if (i == b) {   // C4805, comparing bool and int variables
   }
}
```

Compiler Warning (level 1) C4806

Error Message
'operation' : unsafe operation: no value of type 'type' promoted to type 'type' can equal the given constant

This message warns against code such as `b == 3`, where `b` has type **bool**. The promotion rules cause **bool** to be promoted to **int**. This is legal, but it can never be **true**. The following sample generates C4806:

```
// C4806.cpp
// compile with: /W1
int main()
{
   bool b = true;
   // try..
   // int b = true;

   if (b == 3)   // C4806
   {
      b = false;
   }
}
```

Compiler Warning (level 1) C4807

Error Message
'operation' : unsafe mix of type 'type' and signed bitfield of type 'type'

This warning is generated when comparing a one-bit signed bit field to a **bool** variable. Because a one-bit, signed bit field can only contain the values -1 or 0, it is dangerous to compare it to **bool**. No warnings are generated about mixing **bool** and one-bit, unsigned bitfields since they are identical to **bool** and can only hold 0 or 1.

Example
The following sample generates C4807:

```
// C4807.cpp
// compile with: /W1
typedef struct bitfield {
   signed mybit : 1;
} mybitfield;

int main() {
   mybitfield bf;
   bool b = true;

   // try..
   // int b = true;

   bf.mybit = -1;
   if (b == bf.mybit) {   // C4807
      b = false;
   }
}
```

Compiler Warning (level 1) C4808

Error Message
case 'value' is not a valid value for switch condition of type 'bool'

This warning occurs if a **switch** condition is of type **bool** and one of the **case** values is outside the range of **bool** (that is, **true** and **false**). Only **true**, **false**, 1, 0, and default are valid **case** values if the **switch** condition is of type **bool**.

Compiler Warning (level 1) C4809

Error Message
switch statement has redundant 'default' label; all possible 'case' labels are given

This message is generated in places where you have both 'true' and 'false' cases in addition to a default label.

Compiler Warning (level 1) C4810

Error Message
value of pragma pack(show) == n

This warning is issued when you use the **show** option of the pack pragma. *n* is the current pack value.

For example, the following code shows how the C4810 warning works with the pack pragma:

```
// C4810.cpp
// compile with: /W1 /LD
// C4810 expected
#pragma pack(show)
#pragma pack(4)
#pragma pack(show)
```

Compiler Warning (level 1) C4811

Error Message
value of pragma conform(forScope, show) == value

This warning is issued when you use the **show** option of the conform pragma. *value* is the current conform value.

Compiler Warning (level 1) C4812

Error Message
obsolete declaration style: please use 'new_syntax' instead

In the current release of Visual C++, the explicit constructor specialization is still supported, but it may not be supported in a future release.

The following sample generates C4812:

```
// C4812.cpp
// compile with: /W1 /c
template <class T>
class MyClass;

template<class T>
class MyClass<T*> {
   MyClass();
};

template<class T>
MyClass<T*>::MyClass<T*>() {}   // C4812
// try the following line instead
// MyClass<T*>::MyClass() {}
```

Compiler Warning (level 1) C4813

Error Message
'function' : a friend function of a local class must have been previously declared

A friend function in an inner class was not declared in the outer class.

The following sample generates C4813:

```
// C4813.cpp
// compile with: /W1 /LD
void MyClass()
{
   // void func();
   class InnerClass
   {
      friend void func();   // C4813 uncomment declaration above
   };
}
```

Compiler Warning (level 4) C4815

Error Message
'var' : zero-sized array in stack object will have no elements (unless the object is an aggregate that has been aggregate initialized)

An array with an undefined number of elements (zero-sized array) is the last member of a type and an object of the type was created on the stack. No memory will be allocated for the array. If you need a useful constructor, you can allocate memory for the struct on the heap.

The following sample generates C4815:

```
// C4815.cpp
// compile with: /W4
#include <memory>
#pragma warning(disable : 4200)
struct S1
{
   int i;
   char cArr[];
};

S1 s1_glb;   // C4815 stack object with zero-array (not aggregate inited)
// try the following line instead
// S1 s1_glb = { 0, 0, 0 };

struct S2
{
   S2(int _i) : i(_i) {}
   int i;
   char cArr[];
};

S2 s2_glb1(10);   // C4815
// try the following line instead
// S2 s2_glb1 = { 10 };  // to work, comment the constructor

int main()
{
   S1 s1_loc1;   // C4815
   // try the following line instead
   // S1 s1_loc1 = { 0 };

   S1 s1_loc2 = { 1, { 'a', 'b', 'c' } };

   S1 s1_loc3 = s1_loc2;   // C4815 truncation when copy ctor called
   // truncation if call a compiler-generated copy constructor
   // to copy a struct with a zero sized array in it
   s1_loc1;
   s1_loc2;
   s1_loc3;

   // allocate memory for struct on heap
   int nEle = 10;   // allocate 10 chars for char array
   void * pV = malloc (sizeof(S2) + nEle * sizeof(char));
   S2 *pS1 = new (pV) S2(nEle);
   pS1;
}
```

Compiler Warning (level 4) C4816

Error Message
'param' : parameter has a zero-sized array which will be truncated (unless the object is passed by reference)

A parameter to an object with a zero-size array was not passed by reference. The array will not get copied when the object is passed.

Example
The following sample generates C4816:

```
// C4816.cpp
// compile with: /W4
#include <stdio.h>

extern "C" int printf_s(const char *, ...);

#pragma warning(disable : 4200)

struct S1
{
    int i;
    char cArr[];
};

void TestErr(S1 s1)   // C4816 param with zero-array
                      // not passed by reference
{
    printf_s("%d %c %c\n", s1.i, s1.cArr[0], s1.cArr[1]);
}

void TestOk(S1 &s1)
{
    printf_s("%d %c %c\n", s1.i, s1.cArr[0], s1.cArr[1]);
}

int main()
{
    S1 myS_1 = { 6, 'a', 'b', 'c' };
    TestErr(myS_1);
    TestOk(myS_1);
}
```

Compiler Warning (level 1) C4817

Error Message
'member' : illegal use of '.' to access this member; compiler replaced with '->'

The wrong member access operator was used.

Example
The following sample generates C4817.

```
// C4817.cpp
// compile with: /clr /W1
using namespace System;
int main() {
   array<Int32> ^ a = gcnew array<Int32>(100);
   Console::WriteLine( a.Length );   // C4817
   Console::WriteLine( a->Length );   // OK
}
```

C4817 can also be generated using **/clr:oldSyntax**. The following sample generates C4817.

```
// C4817_b.cpp
// compile with: /clr:oldSyntax /W1
using namespace System;
int main() {
   Int32 a[] = new Int32[11];
   Console::WriteLine( a.Length );
   // Console::WriteLine( a->Length );
}
```

Compiler Warning (level 1) C4819

Error Message
The file contains a character that cannot be represented in the current code page (number). Save the file in Unicode format to prevent data loss.

C4819 occurs when an ANSI source file is compiled on a system with a codepage that cannot represent all characters in the file.

To resolve C4819, save the file in Unicode format.

Compiler Warning (level 4) C4820

Error Message
'bytes' bytes padding added after construct 'member_name'

The type and order of elements caused the compiler to add padding to the end of a struct. See align for more information on padding in a struct.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4820:

```
// C4820.cpp
// compile with: /W4 /c
#pragma warning(default : 4820)

// Delete the following 4 lines to resolve.
__declspec(align(2)) struct MyStruct {
   char a;
   int i;   // C4820
};

// OK
#pragma pack(1)
__declspec(align(1)) struct MyStruct2 {
   char a;
   int i;
};
```

Compiler Warning (level 1) C4821

Error Message
Unable to determine Unicode encoding type, please save the file with signature (BOM)

The compiler could not determine the encoding type for a file. To resolve this warning, save the file with a byte order marker. See Managing Files with Encoding for more information.

Compiler Warning (level 1) C4822

Error Message
'member' : local class member function does not have a body

A local class member function was declared but not defined in class. To use a local class member function, you must define it in the class. You cannot declare it in class and define it out of class.

Any out-of-class definition for a local class member function will be an error.

The following sample generates C4822:

```
// C4822.cpp
// compile with: /W1
int main() {
   struct C {
      void func1(int);   // C4822
      // try the following line instead
      // void func1(int){}
  };
}
```

Compiler Warning (level 3) C4823

Error Message
'function' : uses pinning pointers but unwind semantics are not enabled. Consider using /EHa

To unpin an object on the managed heap pointed to by a pinning pointer declared in a block scope, the compiler simulates the behavior of destructors of local classes, "pretending" the pinning pointer has a destructor that nullifies the pointer. To enable a call to a destructor after throwing an exception, you must enable object unwinding, which you can do by using /EHsc.

You can also manually unpin the object and ignore the warning.

Example
The following sample generates C4823.

```
// C4823.cpp
// compile with: /clr /W3 /EHa-
using namespace System;

ref struct G {
   int m;
};

void f(G ^ pG) {
   try {
      pin_ptr<int> p = &pG->m;
      // manually unpin, ignore warning
      // p = nullptr;
      throw gcnew Exception;
   }
   catch(Exception ^) {}
}   // C4823 warning

int main() {
   f( gcnew G );
}
```

C4823 can also be generated using **/clr:oldSyntax**. The following sample generates C4823.

```
// C4823_b.cpp
// compile with: /clr:oldSyntax /W3 /EHa-
using namespace System;

__gc struct G {
   int m;
};

void f(G* pG) {
   try {
      int __pin* p = &pG->m;
      // manually unpin, ignore warning
      // p = 0;
      throw new Exception;
   }
   catch(Exception*) {}
}   // C4823

int main() {
   f( new G );
}
```

Compiler Warning (level 2) C4826

Error Message
Conversion from 'type1 ' to 'type_2' is sign-extended. This may cause unexpected runtime behavior.

This warning indicates that the compiler performed sign extension when a 32-bit pointer was cast to a 64-bit variable.

If the extension was performed on a windows HANDLE type, it is safe to ignore this warning. If the extension was performed on a pointer type, you should modify the cast to prevent the sign extension (see example below). You can also prevent or permit the sign extension by declaring the pointer with the __sptr, __uptr modifier.

C4826 is off by default. See Compiler Warnings That Are Off by Default for more information.

Example
The following sample generates C4826.

```
// C4826.cpp
// compile with: /W2 /c
#include <windows.h>
#pragma warning(default: 4826)

void * __ptr64 F1 (void * __ptr32 P ) {
   return (void * __ptr64)P;   // C4826
   // try the following line instead
   // return (void * __ptr64)(ULONGLONG)(ULONG)P;
}

void * __ptr64 F2 ( void * P ) {
   return (void * __ptr64)P;   // C4826
   // try the following line instead
   // return (void * __ptr64)(ULONGLONG)(ULONG)P;
}

unsigned __int64 F3r ( void * P ) {
   return (unsigned __int64)P;   // C4826
   // try the following line instead
   // return (unsigned __int64)(ULONGLONG)(ULONG)P;
}
```

Compiler Warning (level 1) C4829

Error Message
Possibly incorrect parameters to function main. Consider 'int main(array<System::String^>^ argv)'

Certain functions, such as main, cannot take reference type parameters. While compilation will succeed, the resulting image will probably not run.

The following sample generates C4829:

```
// C4829.cpp
// compile with: /clr /W1 /c
int main(System::String ^ s) {}   // C4829
```

Compiler Warning (level 1) C4832

Error Message
token 'token' is illegal after UDT 'type name'

A member of a UDT (user-defined type, such a class or struct) was qualified incorrectly. The compiler issues this warning and proceeds as if the qualification was specified correctly.

The following sample generates C4832:

```
// C4832.cpp
// compile with: /W1
struct A {
   enum { e };
};

int main() {
   return A.e;   // C4832
   // try the following line instead
   // return A::e;
}
```

Compiler Warning (level 1) C4835

Error Message
'variable' : the initializer for exported data will not be run until managed code is first executed in the host assembly

When accessing data between managed components, it is recommended that you not use native C++ import and export mechanisms. Instead, declare your data members inside a managed type and reference the metadata with **#using** in the client. For more information, see The #using Directive.

Example
The following sample generates C4835.

```
// C4835.cpp
// compile with: /W1 /clr /LD
int f() { return 1; }
int n = 9;

__declspec(dllexport) int m = f();   // C4835
__declspec(dllexport) int *p = &n;   // C4835
```

The following sample consumes the component built in the previous sample, showing that the value of the variables is not as expected.

```
// C4835_b.cpp
// compile with: /clr C4835.lib
#include <stdio.h>
__declspec(dllimport) int m;
__declspec(dllimport) int *p;

int main() {
   printf("%d\n", m);
   printf("%d\n", p);
}
```

Sample Output

```
0
268456008
```

Compiler Warning (level 1) C4836

Error Message
nonstandard extension used : 'type' : local types or unnamed types cannot be used as template arguments

The C++ standard does not allow the use of local types as template arguments, but the Visual C++ compiler does allow this under **/Ze**. C4836 is an informational warning, to let you know that you are writing non-conformant code. For more information, see /Za, /Ze (Disable Language Extensions).

C4836 is off by default. See Compiler Warnings That Are Off by Default for more information.

Example
The following sample generates C4836.

```
// C4836.cpp
// compile with: /W1
#pragma warning(default:4836)
template <class T>
struct TA {};

struct R {
public:
   void f() {
      struct S {};
      TA<S> ta;   // C4836
   }
};

int main() {
   R r;
   r.f();
}
```

Compiler Warning C4867

Error Message
'function': function call missing argument list; use 'call' to create a pointer to member

A pointer to member function was initialized incorrectly.

This warning can be generated as a result of compiler conformance work that was done for Visual C++ 2005: enhanced pointer-to-member conformance. Code that compiled prior to Visual C++ 2005 will now generate C4867. See Breaking Changes in the Visual C++ 2005 Compiler for more information.

This warning is always issued as an error. Use the warning pragma to disable this warning. For more information about C4867 and MFC/ATL, see _ATL_ENABLE_PTM_WARNING.

Example
The following sample generates C4867.

```
// C4867.cpp
// compile with: /c
class A {
public:
   void f(int) {}

   typedef void (A::*TAmtd)(int);

   struct B {
      TAmtd p;
   };

   void g() {
      B b = {f};   // C4867
      B b2 = {&A::f};   // OK
   }
};
```

Compiler Warning (level 1) C4900

Error Message
intermediate language mismatch between 'tool1' version 'version1' and 'tool2' version 'version2'

The intermediate language used in *tool1* and *tool2* did not match. Check that the most current version of each tool has been installed.

Compiler Warning (level 1) C4905

Error Message
wide string literal cast to 'LPSTR'

The compiler detected an unsafe cast. The cast did succeed, but you should use a conversion routine.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

Example
The following sample generates C4905.

```
// C4905.cpp
// compile with: /W1
#pragma warning(default : 4905)
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
    LPSTR y = (LPSTR)L"1234";   // C4905

    // try the following lines instead
    // wchar_t y[128];
    // size_t  sizeOfConverted;
    // errcode err = 0;
    //
    // err = mbstowcs_s(&sizeOfConverted, &y[0], 128, "12345", 4);
    // if (err != 0)
    // {
    //     printf_s("mbstowcs_s failed!");
    //     exit (-1);
    // }
    // wprintf(L"%s\n", y);

    return 0;
}
```

Compiler Warning (level 1) C4906

Error Message
string literal cast to 'LPWSTR'

The compiler detected an unsafe cast. The cast did succeed, but you should use a conversion routine.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

Example
The following sample generates C4906:

```
// C4906.cpp
// compile with: /W1
#pragma warning(default : 4906)
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
    LPWSTR x = (LPWSTR)"1234";   // C4906

    // try the following lines instead
    // char y[128];
    // size_t numberOfCharConverted = 0;
    // errcode err = 0;
    // err = wcstombs_s(&numberOfCharConverted , &y[0], 128,
    //                  L"12345", 4);
    // if (err != 0)
    // {
    //     printf_s("wcstombs_s failed!");
    //     return -1;
    // }
    // printf_s("%s\n", y);

    return 0;
}
```

Compiler Warning (level 1) C4912

Error Message
'attribute': attribute has undefined behavior on a nested UDT

Attributes that apply to nested UDTs (user-defined type, which could be a typedef, union, or struct) may be ignored.

The following code shows how this warning would be generated:

```
// C4912.cpp
// compile with: /W1
#include <windows.h>
[emitidl, module(name="xx")];

[object, uuid("00000000-0000-0000-0000-000000000002")]
__interface IMy
{
};

[coclass, default(IMy), appobject, uuid("00000000-0000-0000-0000-000000000001")]
class CMy : public IMy
{
   [export, v1_enum] typedef enum myEnum { k3_1 = 1, k3_2 = 2 } myEnumv;   // C4912
};
int main()
{
}
```

Compiler Warning (level 4) C4913

user defined binary operator ',' exists but no overload could convert all operands, default built-in binary operator ',' used

A call to the built-in comma operator occurred in a program that also had an overloaded comma operator; a conversion that you thought may have occurred did not.

The following code sample generates C4913:

```
// C4913.cpp
// compile with: /W4
struct A
{
};

struct S
{
};

struct B
{
   // B() { }
   // B(S &s) { s; }
};

B operator , (A a, B b)
{
   a;
   return b;
}


int main()
{
   A a;
   B b;
   S s;

   a, b;   // OK calls user defined operator
   a, s;   // C4913 uses builtin comma operator
           // uncomment the conversion code in B to resolve.
}
```

Compiler Warning (level 1) C4917

Error Message
'declarator' : a GUID can only be associated with a class, interface or namespace

A user-defined structure other than class, interface, or namespace cannot have a GUID.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following code sample generates C4917:

```
// C4917.cpp
// compile with: /W1
#pragma warning(default : 4917)
__declspec(uuid("00000000-0000-0000-0000-000000000001")) struct S
{
} s;   // C4917, don't put uuid on a struct

int main()
{
}
```

Compiler Warning (level 4) C4918

Error Message
'character' : invalid character in pragma optimization list

An unknown character was found in the optimization list of an optimize pragma statement.

For example, the following statement generates C4918:

```
// C4918.cpp
// compile with: /W4
#pragma optimize("X", on) // C4918 expected
int main()
{
}
```

Compiler Warning (level 1) C4920

Error Message
enum enum member member=value already seen in enum enum as member=value

If a .tlb that you pass to #import has the same symbol defined in two or more enums, this warning indicates that subsequent identical symbols are ignored and will be commented out in the .tlh file.

Assuming a .tlb that contains:

```
library MyLib
{
    typedef enum {
        enumMember = 512
    } AProblem;

    typedef enum {
        enumMember = 1024
    } BProblem;
};
```

the following samples generates C4920,

```
// C4920.cpp
// compile with: /W1
#import "t4920.tlb"   // C4920

int main() {
}
```

Compiler Warning (level 1) C4925

Error Message
'method': dispinterface method cannot be called from script

Scripting languages cannot create a VT_BYREF 'in' parameter, it can only create VT_BYREF 'out' parameters.

Another way to resolve this warning is not make the parameter (in the definition and implementation) a pointer type.

The following sample generates C4925:

```
// C4925.cpp
// compile with: /LD /W1
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
[ module(name="Test")];

[ dispinterface, uuid("00000000-0000-0000-0000-000000000001") ]
__interface IDisp {
   [id(9)] void f([in] int*);
};

[ coclass, uuid("00000000-0000-0000-0000-000000000002")  ]
struct CDisp : IDisp {   // C4925
   void f(int*) {}
};
```

Compiler Warning (level 1) C4926

Error Message
'identifier': symbol is already defined: attributes ignored

A forward declaration was found but an attributed construct with the same name already exists. The attributes for the forward declaration are ignored.

The following sample generates C4926:

```
// C4926.cpp
// compile with: /W1
[module(name="MyLib")];

[coclass]
struct a {
};

[coclass]
struct a;   // C4926

int main() {
}
```

Compiler Warning (level 1) C4927

Error Message
illegal conversion; more than one user-defined conversion has been implicitly applied

More than one user-defined conversion is implicitly applied to a single value -- the compiler did not find an explicit conversion but did find a conversion, which it used.

The following sample generates C4927:

```
// C4927.cpp
// compile with: /W1
struct B
{
   operator int ()
   {
      return 0;
   }
};

struct A
{
   A(int i)
   {
   }

   /*
   // uncomment this constructor to resolve
   A(B b)
   {
   }
   */
};

A f1( B& b)
{
   return A(b);
}

B& f2( B& b)
{
   return b;
}

A f()
{
   B b;
   return A(b);   // ok
   return f1(b);  // ok
   return b;      // C4927
   return B(b);   // C4927
   return f2(b);  // C4927
}

int main()
{
   B b;
   A a = b;
   A a2(b);
}
```

Compiler Warning (level 1) C4928

Error Message
illegal copy-initialization; more than one user-defined conversion has been implicitly applied

More than one user-defined conversion routine was found. The compiler executed the code in all such routines.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4928:

```
// C4928.cpp
// compile with: /W1
#pragma warning(default: 4928)

struct I
{
};

struct I1 : I
{
};

struct I2 : I
{
};

template <class T>
struct Ptr
{
   operator T*()
   {
      return 0;
   }

   Ptr()
   {
   }

   Ptr(I*)
   {
   }
};

int main()
{
   Ptr<I1> p1;
   Ptr<I2> p2 = p1;   // C4928
   // try one of the following two lines to resolve this error
   // Ptr<I2> p2(p1);
   // Ptr<I2> p2 = (I1*) p1;
}
```

Compiler Warning (level 1) C4929

Error Message
'file': typelibrary contains a union; ignoring the 'embedded_idl' qualifier

The embedded_idl attribute of #import could not be applied to the type library because a union is present in the type library. To resolve this warning, don't use embedded_idl.

Example
The following sample defines a component.

```
// C4929a.cpp
// compile with: /LD /link /TLBOUT:C4929a.tlb
#include <objbase.h>
[module(name="Test")];
[public, switch_type(short)] typedef union _TD_UNION_TYPE   {
   [case(24)]
      float fM;
   [case(25)]
      double dMN;
   [default]
      int x;
} TD_UNION_TYPE;

[export, public] typedef struct _TDW_TYPE {
   [switch_is(sU)] TD_UNION_TYPE w;
      short sU;
} TD_TYPE;

[object, uuid("00000000-0000-0000-0000-000000000001")]
__interface I {
   HRESULT f(TD_TYPE*);
};

[coclass, uuid("00000000-0000-0000-0000-000000000002")]
struct C : I {
   HRESULT f(TD_TYPE*) { return 0; }
};
```

The following sample generates C4929.

```
// C4929b.cpp
// compile with: /c /W1
#import "d:\school\bin\C4929a.tlb" embedded_idl   // C4929
```

Compiler Warning (level 1) C4930

Error Message
'prototype': prototyped function not called (was a variable definition intended?)

The compiler detected an unused function prototype. If the prototype was intended as a variable declaration, remove the open/close parentheses.

The following sample generates C4930:

```
// C4930.cpp
// compile with: /W1
class Lock {
public:
   int i;
};

void f() {
   Lock theLock();   // C4930
   // try the following line instead
   // Lock theLock;
}

int main() {
}
```

C4930 can also occur when the compiler cannot distinguish between a function prototype declaration and a function call.

The following sample generates C4930:

```
// C4930b.cpp
// compile with: /EHsc /W1

class BooleanException
{
   bool _result;

public:
   BooleanException(bool result)
      : _result(result)
   {
   }

   bool GetResult() const
   {
      return _result;
   }
};

template<class T = BooleanException>
class IfFailedThrow
{
public:
   IfFailedThrow(bool result)
   {
      if (!result)
      {
         throw T(result);
      }
   }
};

class MyClass
{
public:
```

```
   bool Foo()
   {
      try
      {
         IfFailedThrow<>(MyMethod()); // C4930

         // try one of the following lines instead
         // IfFailedThrow<> ift(MyMethod());
         // IfFailedThrow<>(this->MyMethod());
         // IfFailedThrow<>((*this).MyMethod());

         return true;
      }
      catch (BooleanException e)
      {
         return e.GetResult();
      }
   }

private:
   bool MyMethod()
   {
      return true;
   }
};

int main()
{
   MyClass myClass;
   myClass.Foo();
}
```

In the above sample, the result of a method that takes zero arguments is passed as an argument to the constructor of an unnamed local class variable. The call can be disambiguated by either naming the local variable or prefixing the method call with an object instance along with the appropriate pointer-to-member operator.

Compiler Warning (level 4) C4931

Error Message
we are assuming the type library was built for number-bit pointers

Explicit information was not supplied with the **ptrsize** attribute of the #import directive; the compiler concluded that pointer size of the type library is ***number***.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

Compiler Warning (level 4) C4932

Error Message
__identifier(identifier) and __identifier(identifier) are indistinguishable

The compiler is unable to distinguish between **_finally** and **__finally** or **__try** and **_try** as a parameter passed to __identifier. You should not attempt to use them both as identifiers in the same program, as it will result in a C2374 error.

The following sample generates C4932:

```
// C4932.cpp
// compile with: /clr /W4 /WX
int main() {
   int __identifier(_finally) = 245;   // C4932
   int __identifier(__finally) = 25;   // C4932
}
```

Compiler Warning (level 1) C4935

Error Message
assembly access specifier modified from 'access'

The assembly visibility of a type was modified. The compiler uses the last specifier that it encounters. For example, the assembly visibility of a forward declaration may be different from the assembly visibility of the class definition.

C4935 is only reachable using **/clr:oldSyntax**.

The following sample generates C4935:

```
// C4935.cpp
// compile with: /clr:oldSyntax /W1 /c
public __gc public class X {   // C4935
   int i;
};
```

Compiler Warning C4936

Error Message
this __declspec is supported only when compiled with /clr or /clr:pure

A **__declspec** modifier was used but that **__declspec** modifier is only valid when compiled with one of the /clr options.

For more information, see appdomain and process.

C4936 is always issued as an error. You can disable C4936 with the warning pragma.

The following sample generates C4936:

```
// C4936.cpp
// compile with: /c
// #pragma warning (disable : 4936)
__declspec(process) int i;   // C4936
__declspec(appdomain) int j;   // C4936
```

Compiler Warning (level 4) C4937

Error Message
'text1' and 'text2' are indistinguishable as arguments to 'directive'

Because of the way the compiler processes arguments to directives, names that have meaning to the compiler, such as keywords with multiple text representations (single and double underscore forms), cannot be distinguished.

Examples of such strings are __cdecl and __forceinline. Note, under /Za, only the double underscore forms are enabled.

The following sample generates C4937:

```
// C4937.cpp
// compile with: /openmp /W4
#include "omp.h"
int main() {
   #pragma omp critical ( __leave )   // C4937
   ;

   // OK
   #pragma omp critical ( leave )
   ;
}
```

Compiler Warning (level 4) C4938

Error Message
'var' : Floating point reduction variable may cause inconsistent results under /fp:strict or #pragma fenv_access

You should not use /fp:strict or fenv_access with OpenMP floating-point reductions, because the sum is computed in a different order. Thus, results can differ from the results without /openmp.

The following sample generates C4938:

```
// C4938.cpp
// compile with: /openmp /W4 /fp:strict /c
// #pragma fenv_access(on)
extern double *a;

double test(int first, int last) {
   double sum = 0.0;
   #pragma omp parallel for reduction(+: sum)   // C4938
   for (int i = first ; i <= last ; ++i)
      sum += a[i];
   return sum;
}
```

Without explicit parallelization, the sum is computed as follows:

```
sum = a[first] + a[first + 1] + ... + a[last];
```

With explicit parallelization (and two threads), the sum is computed as follows:

```
sum1 = a[first] + ... a[first + last / 2];
sum2 = a[(first + last / 2) + 1] + ... a[last];
sum = sum1 + sum2;
```

Compiler Warning (level 1) C4939

Error Message

#pragma vtordisp is deprecated and will be removed in a future release of Visual C++

The vtordisp pragma will be removed in a future release of Visual C++.

Example

The following sample generates C4939.

```
// C4939.cpp
// compile with: /c /W1
#pragma vtordisp(off)   // C4939
```

Compiler Warning (level 1) C4944

Error Message
'symbol' : cannot import symbol from 'assembly1': as 'symbol' already exists in the current scope

A symbol was defined in a source code file and then a #using statement referenced an assembly that also defined the symbol. The symbol in the assembly is ignored.

Example
The following sample creates a component with a type called ClassA.

```
// C4944.cs
// compile with: /target:library
// C# source code to create a dll
public class ClassA {
   public int i;
}
```

The following samples generate C4944.

```
// C4944b.cpp
// compile with: /clr /W1
class ClassA {
public:
   int u;
};

#using "C4944.dll"   // C4944 ClassA also defined C4944.dll

int main() {
   ClassA * x = new ClassA();
   x->u = 9;
   System::Console::WriteLine(x->u);
}
```

Compiler Warning (level 1) C4945

Error Message
'symbol' : cannot import symbol from 'assembly2': as 'symbol' has already been imported from another assembly 'assembly1'

A symbol was imported from a referenced assembly but that symbol was already imported from another referenced assembly. Either do not reference one of the assemblies or get the symbol name changed in one of the assemblies.

The following samples generate C4945.

```
// C4945a.cs
// compile with: /target:library
// C# source code to create a dll
public class ClassA {
   public int i;
}
```

and then,

```
// C4945b.cs
// compile with: /target:library
// C# source code to create a dll
public class ClassA {
   public int i;
}
```

and then,

```
// C4945c.cpp
// compile with: /clr /LD /W1
#using "C4945a.dll"
#using "C4945b.dll"   // C4945
```

Compiler Warning (level 1) C4946

Error Message
reinterpret_cast used between related classes: 'class1' and 'class2'

reinterpret_cast should not be used to cast between related types. Use either static_cast or a C-style cast.

This warning is off by default. See Compiler Warnings That Are Off by Default for more information.

The following sample generates C4946:

```
// C4946.cpp
// compile with: /W1
#pragma warning (default : 4946)
class a {
public:
   a() : m(0) {}
   int m;
};

class b : public virtual a {
};

class b2 : public virtual a {
};

class c : public b, public b2 {
};

int main() {
   c* pC = new c;
   a* pA = reinterpret_cast<a*>(pC);   // C4946
   // try the following line instead
   // a* pA = static_cast<a*>(pC);
}
```

Compiler Warning (level 1) C4947

Error Message
'type_or_member' : marked as obsolete

A member or type was marked as obsolete with the ObsoleteAttribute class.

The following sample generates C4947:

```
// C4947.cpp
// compile with: /clr /W1
// C4947 expected
using namespace System;

[System::ObsoleteAttribute]
ref struct S {
   [System::ObsoleteAttribute]
   int i;

   [System::ObsoleteAttribute]
   void mFunc(){}
};

// Any reference to Func1 should generate a warning
[System::ObsoleteAttribute]
Int32 Func1(Int32 a, Int32 b) {
   return (a + b);
}

// Any reference to Func2 should generate a warning with  message
[System::ObsoleteAttribute("Will be removed in next version")]
Int32 Func2(Int32 a, Int32 b) {
   return (a + b);
}

int main() {
   Int32 MyInt1 = ::Func1(2, 2);
   Int32 MyInt2 = ::Func2(2, 2);

   S^ s = gcnew S();
   s->i = 10;
   s->mFunc();
}
```

Compiler Warning (level 2) C4948

Error Message
return type of 'accessor' does not match the last parameter type of the corresponding setter

The compiler found a mismatch between what data type is being get and set for an indexed property.

C4948 is only reachable using **/clr:oldSyntax**.

The following sample generates C4948:

```
// C4948.cpp
// compile with: /clr:oldSyntax /LD /W2

__gc class MyClass
{
   int prop __nogc [2];
   public:

      __property int get_P(int i)
      // try the following line instead
      // __property char get_P(int i)
      {
         return prop[i];
      }

      __property void set_P(int i, char c)
      {
         prop[i] = c;
      }
};   // C4948
```

Compiler Warning (level 1 and level 4) C4949

Error Message
pragmas 'managed' and 'unmanaged' are meaningful only when compiled with '/clr[:option]'

The compiler ignores the managed and unmanaged pragmas if the source code is not compiled with /clr. This warning is informational.

The following sample generates C4949:

```
// C4949.cpp
// compile with: /LD /W1
#pragma managed   // C4949
```

When **#pragma unmanaged** is used without **/clr**, C4949 is a level 4 warning.

The following sample generates C4949:

```
// C4949b.cpp
// compile with: /LD /W4
#pragma unmanaged   // C4949
```

Compiler Warning C4950

Error Message
'type_or_member' : marked as obsolete

A member or type was marked as obsolete with the ObsoleteAttribute.

C4950 is always issued as an error. You can turn off this warning with the **#pragma warning** or **/wd**; see warning or /w, /Wn, /WX, /Wall, /wln, /wdn, /wen, /won (Warning Level) for more information.

The following sample generates C4950:

```
// C4950.cpp
// compile with: /clr
using namespace System;

// Any reference to Func3 should generate an error with message
[System::ObsoleteAttribute("Will be removed in next version", true)]
Int32 Func3(Int32 a, Int32 b) {
   return (a + b);
}

int main() {
   Int32 MyInt3 = ::Func3(2, 2);   // C4950
}
```

Compiler Warning (level 1) C4951

Error Message
'function' has been edited since profile data was collected, function profile data not used

A function has been edited in an input module to /LTCG:PGUPDATE, so that the profile data is now not valid. The input module was recompiled after **/LTCG:PGINSTRUMENT** and has a function (***function***) with a different flow of control than was in the module at the time of the **/LTCG:PGINSTRUMENT** operation.

This warning is informational. To resolve this warning, run **/LTCG:PGINSTRUMENT**, redo all test runs, and run **/LTCG:PGOPTIMIZE**.

This warning would be replaced with an error if **/LTCG:PGOPTIMIZE** had been used.

Compiler Warning (level 1) C4952

Error Message
'function' : no profile data found in program database 'pgd_file'

When using /LTCG:PGUPDATE, the compiler detected an input module that was recompiled after **/LTCG:PGINSTRUMENT** and has a new function (***function***) present.

This warning is informational. To resolve this warning, run **/LTCG:PGINSTRUMENT**, redo all test runs, and run **/LTCG:PGOPTIMIZE**.

This warning would be replaced with an error if **/LTCG:PGOPTIMIZE** had been used.

Compiler Warning (level 1) C4953

Error Message
Inlinee 'function' has been edited since profile data was collected, profile data not used

When using /LTCG:PGUPDATE, the compiler detected an input module that was recompiled after **/LTCG:PGINSTRUMENT** and has a function (***function***) that was edited and where existing test runs identified the function as a candidate for inlining. However, as a result of recompiling the module, the function will no longer be a candidate for inlining.

This warning is informational. To resolve this warning, run **/LTCG:PGINSTRUMENT**, redo all test runs, and run **/LTCG:PGOPTIMIZE**.

This warning would be replaced with an error if **/LTCG:PGOPTIMIZE** had been used.

Compiler Warning (level 1) C4955

Error Message
'import2' : import ignored; already imported from 'import1'

The compiler found two #using statements that reference the same assembly.

To resolve this warning, delete one **#using** statement.

Compiler Warning C4956

Error Message
'type' : this type is not verifiable

This warning is generated when /clr:safe is specified and your code contains a type that is not verifiable.

For more information, see Pure and Verifiable Code.

This warning is issued as an error and can be disabled with the warning pragma or the /wd compiler option.

The following sample generates C4956:

```
// C4956.cpp
// compile with: /clr:safe
int* p;   // C4956
```

Compiler Warning C4957

Error Message
'cast' : explicit cast from 'cast_from' to 'cast_to' is not verifiable

A cast will result in an unverifiable image.

Some casts are safe (for example, a **static_cast** that triggers user-defined conversions and a **const_cast**). A safe_cast is guaranteed to produce verifiable code.

For more information, see Pure and Verifiable Code.

This warning is issued as an error and can be disabled with the warning pragma or the /wd compiler option.

The following sample generates C4957:

```
// C4957.cpp
// compile with: /clr:safe
// #pragma warning( disable : 4957 )
using namespace System;
int main() {
   Object ^ o = "Hello, World!";
   String ^ s = static_cast<String^>(o);   // C4957
   String ^ s2 = safe_cast<String^>(o);   // OK
}
```

Compiler Warning C4958

Error Message
'operation' : pointer arithmetic is not verifiable

Using pointer arithmetic will produce an unverifiable image.

For more information, see Pure and Verifiable Code.

This warning is issued as an error and can be disabled with the warning pragma or the /wd compiler option.

The following sample generates C4958:

```
// C4958.cpp
// compile with: /clr:safe
// #pragma warning( disable : 4958 )
using namespace System;

int main( ) {
   Int32 arr[] = new Int32[10];
   Int32* p = &arr[0];
   p++;   // C4958
}
```

The compiler implements array operations with pointer arithmetic. Therefore, native arrays are not verifiable; use a CLR array instead. For more information, see array.

The following sample generates C4958:

```
// C4958b.cpp
// compile with: /clr:safe
// #pragma warning( disable : 4958 )

int main() {
   int array[5];
   array[4] = 0;   // C4958
}
```

Compiler Warning C4959

Error Message
cannot define unmanaged struct 'type' in /clr:safe because accessing its members yields unverifiable code

Accessing a member of an unmanaged type will produce an unverifiable (peverify.exe) image.

For more information, see Pure and Verifiable Code.

This warning is issued as an error and can be disabled with the warning pragma or the /wd compiler option.

The following sample generates C4959:

```
// C4959.cpp
// compile with: /clr:safe

// Uncomment the following line to resolve.
// #pragma warning( disable : 4959 )
struct X {
   int data;
};

int main() {
   X x;
   x.data = 10;   // C4959
}
```

Compiler Warning (level 4) C4960

Error Message
'function' is too big to be profiled

When using /LTCG:PGOPTIMIZE, the compiler detected an input module with a function larger than 65,535 instructions. Such a large function is not available for profile-guided optimizations.

To resolve this warning, reduce the size of the function.

Compiler Warning C4961

Error Message

No profile data was merged into '.pgd file', profile-guided optimizations disabled

No profile data (.pgc files) were available, so profile-guided optimizations cannot take place.

Compiler Warning C4962

Error Message
'function' : Profile-guided optimizations disabled because optimizations caused profile data to become inconsistent"

A function was not compiled with /LTCG:PGO, because count (profile) data for the function was unreliable. Redo profiling to regenerate the .pgc file that contains the unreliable profile data for that function.

This warning is off by default. For more information, see Compiler Warnings That Are Off by Default.

Compiler Warning (level 1) C4964

Error Message
No optimization options were specified; profile info will not be collected

/GL and /LTCG were specified, but no optimizations were requested, so no .pgc files will be generated and, therefore, no profile-guided optimizations will be possible.

If you want .pgc files to be generated when you run your application, specify one of the /O compiler options.

The following sample generates C4964:

```
// C4964.cpp
// compile with: /W1 /GL /link /ltcg:pgi
// C4964 expected
// Add /O2, for example, to the command line to resolve this warning.
int main() {
   int i;
}
```

Compiler Warning (level 1) C4965

Error Message
implicit box of integer 0; use nullptr or explicit cast

Visual C++ 2005 features implicit boxing of value types. An instruction that resulted in a null assignment using Managed Extensions for C++ now becomes an assignment to a boxed int.

For more information, see Implicit Boxing.

Example
The following sample generates C4965.

```
// C4965.cpp
// compile with: /clr /W1
int main() {
   System::Object ^o = 0;   // C4965

   // the previous line is the same as the following line
   // using Managed Extensions for C++
   // System::Object *o = __box(0);

   // OK
   System::Object ^o2 = nullptr;
   System::Object ^o3 = safe_cast<System::Object^>(0);
}
```

Compiler Warning C4972

Error Message
Directly modifying or treating the result of an unbox operation as an lvalue is unverifiable

Dereferencing a handle to a value type, also known as unboxing, and then assigning to it is not verifiable.

For more information, see Implicit Boxing.

Example
The following sample generates C4972.

```
// C4972.cpp
// compile with: /clr:safe
using namespace System;
ref struct R {
   int ^ p;   // a value type
};

int main() {
   R ^ r = gcnew R;
   *(r->p) = 10;   // C4972

   // OK
   r->p = 10;
   Console::WriteLine( r->p );
   Console::WriteLine( *(r->p) );
}
```

Compiler Warning (level 3) C4980

Error Message
'keyword' : use of this keyword requires /clr:oldSyntax command line option

A keyword from a previous version was used. Update your code for the newer syntax or use /clr:oldSyntax.

C4980 is a warning that is always issued as an error. Use **/wd** to turn the warning off. See /w, /Wn, /WX, /Wall, /wln, /wdn, /wen, /won (Warning Level) for more information.

The following sample generates C4980:

```
// C4980.cpp
// compile with: /clr
int main() {
   int x1 = 0;
   int^ px = __box(x1);   // C4980
   int^ px2 = x1;   // OK
}
```

Compiler Warning (level 1) C4995

Error Message
'function': name was marked as #pragma deprecated

The compiler encountered a function that was marked with pragma deprecated. The function may no longer be supported in a future release. You can turn this warning off with the warning pragma (example below).

Example
The following sample generates C4995:

```
// C4995.cpp
// compile with: /W1
#include <stdio.h>

// #pragma warning(disable : 4995)
void func1(void)
{
    printf("\nIn func1");
}

int main()
{
    func1();
    #pragma deprecated(func1)
    func1();   // C4995
}
```

Compiler Warning (level 1) C4996

Error Message
'function': was declared deprecated

The compiler encountered a function that was marked with deprecated. The function may no longer be supported in a future release. You can turn this warning off with the warning pragma (example below).

C4996 is generated for the line on which the function is declared and for the line on which the function is used.

You will see C4996 if you are using members of the <hash_map> and <hash_set> header files in the std namespace. See The stdext Namespace for more information.

Some CRT and Standard C++ Library functions have been deprecated in favor of new, more secure functions. For more information on deprecated functions, see Security Enhancements in the CRT and Safe Libraries: Standard C++ Library.

C4996 can also occur if you use MFC or ATL functions that were deprecated for security reasons. To suppress these warnings, see _AFX_SECURE_NO_WARNINGS and _ATL_SECURE_NO_WARNINGS.

Example

The following sample generates C4996.

```
// C4996.cpp
// compile with: /W1
// C4996 warning expected
#include <stdio.h>

// #pragma warning(disable : 4996)
void func1(void) {
   printf_s("\nIn func1");
}

__declspec(deprecated) void func1(int) {
   printf_s("\nIn func2");
}

int main() {
   func1();
   func1(1);
}
```

C4996 can also occur if you do not use a checked iterator when compiling with _SECURE_SCL 1. See Checked Iterators for more information.

The following sample generates C4996.

```
// C4996_b.cpp
// compile with: /EHsc /W1
#define _SECURE_SCL 1
#include <algorithm>
using namespace std;
using namespace stdext;
int main() {
   int a [] = {1, 2, 3};
   int b [] = {10, 11, 12};
   copy(a, a + 3, b);   // C4996
   copy(a, a + 3, checked_array_iterator<int *>(b, 3));   // OK
}
```

Compiler Warning (level 1) C4997

Error Message
'class': coclass does not implement a COM interface or pseudo-interface

A class marked with the coclass attribute did not implement an interface.

The following sample generates C4997:

```
// C4997.cpp
// compile with: /WX
// to resolve this C4997, uncomment all code
#include <objbase.h>

[ object ]
__interface I {
   HRESULT func();
};

[ coclass ]
struct C /*: I*/ {
   /*
   HRESULT func() {
      return S_OK;
   }
   */
};   // C4997
```

Compiler Warning (level 1) C4999

Error Message
UNKNOWN WARNING From the Help menu choose the Technical Support command or open the Technical Support help file for more information

Note the circumstances of the error, try to isolate the problem and create a reproducible test case, then contact Microsoft Product Support Services.

Compiler Errors when Implementing a CObject-Derived Class

When you implement a class derived from **CObject** and your code is written so that the copy constructor or assignment operator for the class needs to be called, the compiler may report errors similar to the following:

```
error C2660: 'CSample::CSample' : function does not take 1 parameters
error C2582: 'CSample' : 'operator =' function is unavailable
```

You can reproduce the problem by compiling the example in the Sample Code section below.

Note

The sample code shown in this article generates the following error messages:

```
error C2558: 'CSample::CSample' : no copy constructor available
error C2582: 'CSample' : 'operator =' function is unavailable
```

The reason for the compiler errors is that **CObject** declares a private copy constructor and assignment operator in the AFX.h file. Consequently, the compiler does not generate a default copy constructor and assignment operator for the **CObject**-derived class. Because the compiler does not find these functions declared in the class, it reports the errors.

To avoid the compiler errors, you need to implement a copy constructor and assignment operator for the **CObject**-derived class. This is illustrated in the sample code below. You can avoid the errors by uncommenting the lines indicated in the sample code.

Sample Code

```
// _error_Compiler_Errors_when_Implementing_a_CObject.2d.Derived_Class.cpp
/* Compile options needed: /c
*/

// replace with #define _CONSOLE when compiling for Windows NT
#define _DOS

#include <afx.h>

class CSample : public CObject
{
private:
    short m_nValue;
public:
// uncomment the lines below to avoid the compiler errors
//    CSample() {}
//    CSample( const CSample &s )  // copy ctor
//        { m_nValue = s.m_nValue; }
//    CSample& operator=( const CSample &s )  // assignment operator
//        { m_nValue = s.m_nValue; return *this; }
};

int main()
{
    CSample a;
    CSample b = a;
    a = b;
}
```

See Also

Other Resources

Compiler Warnings C4600 Through C4999

Expression Evaluator Errors CXX0000 Through CXX0072

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, or type the error number in the **Look for** box in the Index.

Expression Evaluator Error CXX0000

Error Message
no error condition

No error has occurred. You can continue debugging normally.

Note the circumstances and notify Microsoft Product Support Services.

Expression Evaluator Error CXX0001

Error Message
exception executing user function

The code being executed caused a general protection fault. This error is identical to CAN0001.

Expression Evaluator Error CXX0002

Error Message
error accessing user memory

The expression attempts to reference memory that is not allocated to the program being debugged.

This error is identical to CAN0002.

Expression Evaluator Error CXX0003

Error Message
internal error in expression evaluator

The expression evaluator encountered an internal error.

Note the circumstances of the error and notify Microsoft Corporation. You can report the error online at http://support.microsoft.com/support/visualc/report/ or you can call Microsoft Product Support Services. For more information, click **Technical Support** on the **Help** menu in Visual C++.

This error is identical to CAN0003.

Expression Evaluator Error CXX0004

Error Message
syntax error

The syntax of the expression is incorrect.

Retype the expression with the correct syntax.

This error is identical to CAN0004.

Expression Evaluator Error CXX0005

Error Message
operator not supported

An unsupported C operator was specified in an expression. Write an equivalent expression using the supported C operators.

This error is identical to CAN0005.

Expression Evaluator Error CXX0006

Error Message
missing left parenthesis

Unbalanced parentheses were found in the expression. Retype the expression with balanced parentheses.

This error is identical to CAN0006.

Expression Evaluator Error CXX0007

Error Message
missing right parenthesis

Unbalanced parentheses were found in the expression. Retype the expression with balanced parentheses.

This error is identical to CAN0007.

Expression Evaluator Error CXX0008

missing " at end of string

The double quote expected at the end of the string literal was missing. Retype the expression, enclosing the string literal in double quotation marks.

This error is identical to CAN0008.

Expression Evaluator Error CXX0009

Error Message
missing ' after character constant

The single quote expected at the end of the character constant was missing. Retype the expression, enclosing the character constant in single quotation marks.

This error is identical to CAN0009.

Expression Evaluator Error CXX0010

Error Message
missing left bracket

The expression contains unbalanced square brackets. Retype the expression with balanced square brackets.

This error is identical to CAN0010.

Expression Evaluator Error CXX0011

Error Message
missing right bracket

The expression contains unbalanced square brackets. Retype the expression with balanced square brackets.

This error is identical to CAN0011.

Expression Evaluator Error CXX0012

Error Message
missing left curly brace

The expression contains an unbalanced curly brace. Retype the expression with balanced curly braces.

This error is identical to CAN0012.

Expression Evaluator Error CXX0013

Error Message
missing operator

An operator was expected in the expression but was not found. Check the syntax of the expression.

This error is identical to CAN0013.

Expression Evaluator Error CXX0014

Error Message
missing operand

An operator was specified without a required operand. Check the syntax of the expression.

This error is identical to CAN0014.

Expression Evaluator Error CXX0015

Error Message
expression too complex (stack overflow)

The expression entered was too complex or nested too deeply for the amount of storage available to the C expression evaluator.

Overflow usually occurs because of too many pending calculations.

Rearrange the expression so that each component of the expression can be evaluated as it is encountered, rather than having to wait for other parts of the expression to be calculated.

Break the expression into multiple commands.

This error is identical to CAN0015.

Expression Evaluator Error CXX0016

Error Message
constant too big

The C expression evaluator cannot accept an unsigned integer constant larger than 4,294,967,295 (0FFFFFFFF hexadecimal), or a floating-point constant whose magnitude is larger than approximately 1.8E+308.

This error is identical to CAN0016.

Expression Evaluator Error CXX0017

Error Message
symbol not found

A symbol specified in an expression could not be found.

One possible cause of this error is a case mismatch in the symbol name. Because C and C++ are case-sensitive languages, a symbol name must be given in the exact case in which it is defined in the source.

This error can occur when trying to typecast a variable in order to watch the variable during debugging. The **typedef** declares a new name for a type, but it does not define a new type. The typecast attempted in the debugger requires the name of a defined type.

This error is identical to CAN0017.

To fix by using the following possible solutions

1. Make sure the symbol is already declared at the point in the program where it is being used.
2. Use an actual type name to cast variables in the debugger, rather than a **typedef**-defined name.

Expression Evaluator Error CXX0018

Error Message
bad register name

A specified register does not exist or cannot be displayed.

The Watch window can display the following registers:

AX SP DS IP

BX BP ES FL

CX SI SS GS

DX DI CS SS

EAX ESP DS EIP

EBX EBP ES EFL

ECX ESI FS

EDX EDI CS

This error is identical to CAN0018.

Expression Evaluator Error CXX0019

Error Message
bad type cast

The C expression evaluator cannot perform the type cast as written.

This error is identical to CAN0019.

To fix by checking the following possible causes

1. The specified type is unknown.
2. There were too many levels of pointer types. For example, the type cast

   ```
   (char **)h_message
   ```

   cannot be evaluated by the C expression evaluator.

Expression Evaluator Error CXX0020

Error Message
operand types bad for this operation

An operator was applied to an expression with an invalid type for that operator.

For example, it is not valid to take the address of a register or subscript an array with a floating-point expression.

This error is identical to CAN0020.

Expression Evaluator Error CXX0021

Error Message
struct or union used as scalar

A structure or union was used in an expression, but no element was specified.

When manipulating a structure or union variable, the name of the variable may appear by itself, without a field qualifier. If a structure or union is used in an expression, it must be qualified with the specific element desired.

Specify the element whose value is to be used in the expression.

This error is identical to CAN0021.

Expression Evaluator Error CXX0022

Error Message
function call before _main

The C expression evaluator cannot evaluate a function before the debugger has entered the function **_main**. The program is not properly initialized until **_main** has been called.

This error is identical to CAN0022.

Expression Evaluator Error CXX0023

Error Message
bad radix

The C expression evaluator does not recognize the radix specified. Only decimal, hexadecimal, and octal radixes are valid.

This error is identical to CAN0023.

Expression Evaluator Error CXX0024

Error Message
operation needs l-value

An expression that does not evaluate to an l-value was specified for an operation that requires an l-value.

An l-value (so called because it appears on the left side of an assignment statement) is an expression that refers to a memory location.

For example, `buffer[count]` is a valid l-value because it points to a specific memory location. The logical comparison `zed != 0` is not a valid l-value because it evaluates to TRUE or FALSE, not to a memory address.

This error is identical to CAN0024.

Expression Evaluator Error CXX0025

Error Message
operator needs struct/union

An operator that takes an expression of **struct** or **union** type was applied to an expression that is not a **struct** or **union**.

Components of class, structure, or union variables must have a fully qualified name. Components cannot be entered without full specification.

This error is identical to CAN0025.

Expression Evaluator Error CXX0026

Error Message
bad format string

A format string was improperly specified. Check the syntax of the expression.

This error is identical to CAN0026.

For more information, see Format Specifiers in C++.

Expression Evaluator Error CXX0027

Error Message
tp addr not l-value

Check the syntax of the expression.

This error is identical to CAN0027.

Expression Evaluator Error CXX0028

Error Message
not struct/union element

An expression of the form 'Struct.Member' or 'pStruct->Member' was specified, but <member> is not an element of the structure.

The expression may not be parenthesized correctly.

This error is identical to CAN0028.

Expression Evaluator Error CXX0029

Error Message
not struct pointer

The member-selection operator (**->**) was applied to an expression that is not a pointer to a structure.

Check that the entire expression is parenthesized correctly, or type cast the address expression to the appropriate structure pointer type.

This error is identical to CAN0029.

Expression Evaluator Error CXX0030

Error Message
expression not evaluatable

The debugger's expression evaluator could not obtain a value for the expression as written. One likely cause is that the expression refers to memory that is outside the program's address space (dereferencing a null pointer is one example). Windows does not allow access to memory that is outside of the program's address space.

You may want to rewrite your expression using parentheses to control the order of evaluation.

This error is identical to CAN0030.

Expression Evaluator Error CXX0031

Error Message
expression not expandable

The expression evaluator encountered an internal error.

You may be able to write an equivalent expression that can be evaluated correctly.

This error is identical to CAN0031.

Expression Evaluator Error CXX0032

Error Message
divide by 0

The expression contains a divisor of zero, which is illegal. This divisor may be the literal number zero, or it may be an expression that evaluates to zero.

This error is identical to CAN0032.

Expression Evaluator Error CXX0033

Error Message
error in OMF type information

The executable file did not have a valid object module format (OMF) for debugging.

This error is identical to CAN0033.

To fix by checking the following possible causes

1. The executable file was not created with the linker released with this version of Visual C++. Relink the object code using the current version of LINK.exe.
2. The .exe file may have been corrupted. Recompile and relink the program.

Expression Evaluator Error CXX0034

types incompatible with operator

The operand types specified are not legal for the operation.

For example, a pointer cannot be multiplied by any value.

You may need to type cast the operands to a type compatible with the operator.

Expression Evaluator Error CXX0036

Error Message
bad context {...} specification

This message can be generated by any of several errors in the use of the context operator (**{}**).

- The syntax of the context operator (**{}**) was given incorrectly.

  The syntax of the context operator is:

  {*function*,*module*,*dll*}*expression*

  This specifies the context of *expression*. The context operator has the same precedence and usage as a type cast.

  Trailing commas can be omitted. If any of *function*, *module*, or *dll* contains a literal comma, you must enclose the entire name in parentheses.

- The function name was spelled incorrectly, or does not exist in the specified module or dynamic-link library.

  Because C is a case-sensitive language, *function* must be given in the exact case as it is defined in the source.

- The module or DLL could not be found.

  Check the full path name of the specified module or DLL.

This error is identical to CAN0036.

Expression Evaluator Error CXX0037

Error Message
out of memory

The C expression evaluator ran out of memory evaluating the expression.

This error is identical to CAN0037.

Expression Evaluator Error CXX0038

Error Message
function argument count and/or type mismatch

The function call as specified does not match the prototype for the function.

Retype the call with the correct number of arguments. Type cast each argument to match the prototype, as necessary.

This error is identical to CAN0038.

Expression Evaluator Error CXX0039

Error Message
symbol is ambiguous

The C expression evaluator cannot determine which instance of a symbol to use in an expression. The symbol occurs more than once in the inheritance tree.

You must use the scope resolution operator (**::**) to explicitly specify the instance to use in the expression.

This error is identical to CAN0039.

Expression Evaluator Error CXX0040

Error Message
function requires implicit conversion

The C expression evaluator does not support implicit conversions involving constructor calls.

This error is identical to CAN0040.

Expression Evaluator Error CXX0041

Error Message
class element must be static member or member function

A nonstatic member of a class (or structure or union) was used without specifying which instantiation of the class to use.

Only static data members or member functions can be used without specifying an instantiation.

This error is identical to CAN0041.

Expression Evaluator Error CXX0042

Error Message
bad line number

This error should never occur.

Note the circumstances of the error and notify Microsoft Corporation. You can report the error online at http://support.microsoft.com/support/visualc/report/ or you can call Microsoft Product Support Services. For more information, click **Technical Support** on the **Help** menu in Visual C++.

This error is identical to CAN0042.

Expression Evaluator Error CXX0043

Error Message
this pointer used outside member function

The **this** pointer can only be used for nonstatic member functions.

This error is identical to CAN0043.

Expression Evaluator Error CXX0044

Error Message
use of _based(void) pointer requires :> operator

A pointer based on **void** cannot be used directly. You must form a complete pointer using the **:>** operator.

This error is identical to CAN0044.

Expression Evaluator Error CXX0045

Error Message
not a function

An argument list was supplied for a symbol in the program that is not the name of a function.

Example

```
queue( alpha, beta )
```

when `queue` is not a function.

This error is identical to CAN0045.

Expression Evaluator Error CXX0046

Error Message
argument list required for member function

An expression called a member function but did not specify any actual parameters.

This error is identical to CAN0046.

Expression Evaluator Error CXX0047

Error Message
argument list does not match a function

An expression called a function with an actual parameter list that did not match the formal parameter list of any function with the same name defined in the program.

Overloaded functions can be called only if there is an exact parameter match or a match that does not require the construction of an object.

This error is identical to CAN0047.

Expression Evaluator Error CXX0048

Error Message
calling sequence not supported

A function specified in the expression uses a calling sequence not supported by the C expression evaluator. You cannot call this function in a Watch window expression.

This error is identical to CAN0048.

Expression Evaluator Error CXX0049

Error Message
obsolete OMF - please relink program

The program used an old OMF (Object Module Format).

Relink the program using the current linker version.

This error is identical to CAN0049.

Expression Evaluator Error CXX0050

Error Message
left side of :: must be class/struct/union

The symbol on the left side of the scope-resolution operator (**::**) was not a class, structure, or union.

This error is identical to CAN0050.

Expression Evaluator Error CXX0051

Error Message
more than one overloaded symbol specified in breakpoint

The expression evaluator could not determine which of more than one overloaded symbol to use as a breakpoint.

This error is identical to CAN0051.

Expression Evaluator Error CXX0052

Error Message
member function not present

A member function was specified as a breakpoint but could not be found. Setting a breakpoint at a function that has been inlined can cause this error.

Recompile the file with inlining forced off (/Ob0) to set a breakpoint in this function.

An expression called a function that was not defined.

This error is identical to CAN0052.

Expression Evaluator Error CXX0053

Error Message
nonfunction symbol match while binding breakpoints

A symbol used as a breakpoint was not a function. Specifying a data member as a breakpoint can cause this error.

This error is identical to CAN0053.

Expression Evaluator Error CXX0054

Error Message
register in breakpoint expression illegal

A register cannot be used in a breakpoint expression.

This error is identical to CAN0054.

Expression Evaluator Error CXX0055

Error Message
ambiguous symbol in context operator

A symbol in the context operator (**{}**) referred to more than one symbol in the program.

The scope resolution operator (**::**) may be able to resolve the ambiguity.

This error is identical to CAN0055.

Expression Evaluator Error CXX0056

Error Message
error in line number

An invalid line number was specified.

This error is identical to CAN0056.

Expression Evaluator Error CXX0057

Error Message
no code at line number

No code was generated for the specified line number. It cannot be used as a breakpoint.

This error is identical to CAN0057.

Expression Evaluator Error CXX0058

Error Message
overloaded operator not found

A class type was specified as the left operand in an expression, but an overloaded operator was not defined for the class.

This error is identical to CAN0058.

Expression Evaluator Error CXX0059

Error Message
left operand is class not a function name

The left operand of a function call was a class name and could not be resolved to a function call. Omitting the name of a member function in an expression can cause this error.

This error is identical to CAN0059.

Expression Evaluator Error CXX0060

Error Message
register is not available

An expression specified a register that cannot be used. This error can be caused by trying to access a register that does not exist on the machine running.

This error is identical to CAN0060.

Expression Evaluator Error CXX0061

Error Message
function nesting depth exceeded

The expression contains a function nesting depth greater than the limit. Modify the expression to reduce the nesting depth.

This error is identical to CAN0061.

Expression Evaluator Error CXX0062

Error Message
constructor calls not supported

An expression made a call to a constructor. Expressions cannot make explicit calls to constructors or make conversions that require a call to a constructor.

This error is identical to CAN0062.

Expression Evaluator Error CXX0063

Error Message
overloaded operator -> not supported

The expression used an overloaded class member access operator (**->**).

This error is identical to CAN0063.

Expression Evaluator Error CXX0064

Error Message
can't set breakpoint on bound virtual member function

A breakpoint was set on a virtual member function through a pointer to an object, such as:

```
pClass->vfunc( int );
```

A breakpoint can be set on a virtual function by entering the class, such as:

```
Class::vfunc( int );
```

This error is identical to CAN0064.

Expression Evaluator Error CXX0065

Error Message
variable needs stack frame

An expression contained a variable that exists within the current scope but hasn't been created yet.

This error can occur when you have stepped into the prolog of a function but not yet set up the stack frame for the function, or if you have stepped into the exit code for the function.

Step through the prolog code until the stack frame has been set up before evaluating the expression.

This error is identical to CAN0065.

Expression Evaluator Error CXX0066

Error Message
static member not present

A static member of a class could not be found or was not defined. This error can result from a static class member that is declared but not defined, or is only defined and referenced in modules that do not contain debug information.

This error is identical to CAN0066.

Expression Evaluator Error CXX0067

Error Message
function evaluation not supported

The expression contained a function call. Some expression evaluators do not support function calls.

This error is identical to CAN0067.

Expression Evaluator Error CXX0069

Error Message
variable needs stack frame

The expression evaluator cannot evaluate the variable because it does not occur in a stack frame. This may be caused by variables declared as part of an inline function.

Expression Evaluator Error CXX0072

Error Message

Error: type information missing or unknown

The .pch file did not get linked in, or the code has a reference to a type that is in a module not compiled with /Zi.

This error is identical to CAN0067.

Command-Line Errors D8000 Through D9999

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, type the error number in the **Look for** box in the Index, or click a link below.

Command-Line Error D8000

Command-Line Error D8003

Command-Line Error D8004

Command-Line Error D8016

Command-Line Error D8018

Command-Line Error D8021

Command-Line Error D8022

Command-Line Error D8027

Command-Line Error D8029

Command-Line Error D8030

Command-Line Error D8034

Command-Line Error D8035

Command-Line Error D8036

Command-Line Error D8037

Command-Line Error D8039

Command-Line Error D8042

Command-Line Error D8045

Command-Line Error D8046

Command-Line Warning D9000

Command-Line Warning D9002

Command-Line Warning D9007

Command-Line Warning D9014

Command-Line Warning D9015

Command-Line Warning D9021

Command-Line Warning D9024

Command-Line Warning D9025

Command-Line Warning D9026

Command-Line Warning D9027

Command-Line Warning D9028

Command-Line Warning D9029

Command-Line Warning D9032

Command-Line Warning D9035

Command-Line Warning D9036

Command-Line Warning D9038

Command-Line Warning D9040

Command-Line Warning D9041

Command-Line Warning D9042

Command-Line Warning D9043

Command-Line Warning D9044

See Also

Other Resources

C/C++ Build Errors

Command-Line Error D8000

Error Message
unknown error; consult documentation for technical support options

Note the circumstances of the error, try to isolate the problem and create a reproducible test case, then contact Microsoft Product Support Services.

Command-Line Error D8003

Error Message
missing source filename

No file name was specified when compiling from the command line. To remedy this error, add at least one file name to the command.

Example

The following command line example generates error D8003:

```
cl /c
```

To fix the error in the above code example, try `cl /c` *filename*`.cpp`.

See Also

Reference
Compiler Options
Other Resources
Command-Line Errors D8000 Through D9999

Command-Line Error D8004

Error Message
'/option' requires an argument

A command line option that requires an argument was not passed one. To remedy the error, specify an appropriate argument for the option.

Example
The following command line example generates error D8004:

```
cl /FU /LD filename.cpp
```

To fix the error in the above code example, try `cl /FU System.Data.dll /LD` *filename*`.cpp`.

See Also
Reference
Compiler Options
Other Resources
Command-Line Errors D8000 Through D9999

Command-Line Error D8016

Error Message
'option1' and 'option2' command-line options are incompatible

The command-line options cannot be specified together.

Check environment variables, such as CL, for option specifications.

In Visual C++ 2005, **/clr** implies **/EHa**, and you cannot specify any other **/EH** compiler option with **/clr**. For more information, see /clr (Common Language Runtime Compilation).

You may get D8016 after updating a Visual C++ 6.0 project: the project update wizard process may enable **/RTC** for each source code file in the project, which overrides the **/RTC** setting for the project. To resolve, change the **/RTC** setting for each source code file in the project to the default setting, which means the project setting for **/RTC** will be in effect for each file.

See /RTC (Run-Time Error Checks) for information on changing the **/RTC** property setting.

Command-Line Error D8018

Error Message
cannot create linker response file

The compiler could not create a response file for passing arguments to the linker.

This error can occur when an existing read-only file has the same name as the filename the compiler gives to the response file.

Command-Line Error D8021

Error Message
invalid numeric argument 'number'

A number greater than 65,534 was specified as a numeric argument.

Command-Line Error D8022

Error Message
cannot open 'messagefile'

The given file was not in the current directory or in a directory specified in the PATH environment variable. The message file contains a brief summary of compiler command-line syntax and options.

Move this file to the current directory or a directory in the current path. If this file cannot be found, run the SETUP program to copy it from the distribution disks.

Command-Line Error D8027

Error Message
cannot execute 'component'

The compiler could not run the given compiler component or linker.

To fix by checking the following possible causes

1. Not enough memory to load the component. If NMAKE invoked the compiler, run the compiler outside of the makefile.
2. The current operating system could not run the component. Make sure the path points to the executable files appropriate to your operating system.
3. The component was corrupted. Recopy the component from the distribution disks, using the SETUP program.
4. An option was specified incorrectly. For example:

```
cl /B1 file1.c
```

Command-Line Error D8029

Error Message
cannot execute 'file' out of environment memory

D8029 occurs when the compiler fails to set environment variables, normally due to lack of environment space.

This can happen if some limit being approached by you project, such as a very long huge command line or many large environment variables.

Command-Line Error D8030

Error Message
INTERNAL COMPILER ERROR in 'component'

Note the circumstances of the error, try to isolate the problem and create a reproducible test case, then notify Microsoft Corporation. You can report the error online at http://support.microsoft.com/support/visualc/report/ or you can call Microsoft Technical Support. For more information, click **Technical Support** on the **Help** menu in Visual C++.

Command-Line Error D8034

Error Message
error reading 'filename'

The specified file was opened but could not be read.

This error may be caused by a hardware failure.

Command-Line Error D8035

Error Message
response files nested too deeply

Move the directives in a nested response file into the calling response file.

A recursively nested response file can cause this error.

Command-Line Error D8036

Error Message
'/option' not allowed with multiple source files

These compiler options cannot be used with multiple source files:

- Name assembly file listing (/Fa)
- Rename object file (/Fo)
- Create source browser information without local variables (/Fr)
- Create source browser information that includes local variables (/FR)

Command-Line Error D8037

Error Message
cannot create temporary il file; clean temp directory of old il files

There is not enough space to create temporary compiler intermediate files. To remedy this error, remove any old MSIL files in the directory specified by the **TMP** environment variable. These files will be of the form _CL_hhhhhhhh.ss, where h represents a random hexadecimal digit and ss represents the type of IL file. Also, be sure to update your machine with the latest operating system patches.

See Also

Reference
Compiler Options
Other Resources
Command-Line Errors D8000 Through D9999

Command-Line Error D8039

Error Message
cannot find entry point 'entry' in file 'file', make sure the correct version of the compiler is used

The version of the Visual C++ compiler that you are using is incorrect.

To correct this error

- Reinstall Visual Studio.

Command-Line Error D8042

Error Message
out of memory

D2042 indicates a general, out-of-memory condition.

To resolve this error, ensure there is enough physical and swap memory available prior to invoking the compiler.

Command-Line Error D8045

Error Message
cannot compile C file 'file' with the /clr option

Only C++ source code files can be passed to a compilation that uses **/clr**. Use **/TP** to compile a .c file as a .cpp file; see /Tc, /Tp, /TC, /TP (Specify Source File Type) for more information.

For more information, see /clr (Common Language Runtime Compilation).

D8045 can also occur if you compile an ATL application using Visual C++ 2005. See How To: Migrate to /clr for more information.

Command-Line Error D8046

Error Message
cannot compile C++ file 'file' with the /Zg option; /Zg is only valid on C files.

/Zg can only be used with C files; see /Zg (Generate Function Prototypes) for more information on **/Zg**.

Command-Line Warning D9000

Error Message
unknown warning; consult documentation for technical support options

Note the circumstances of the error, try to isolate the problem and create a reproducible test case, then get support as described in Getting Help from Microsoft Product Support Services (Visual Studio).

Command-Line Warning D9002

Error Message
ignoring unknown option 'option'

The compiler did not recognize the given command-line option; the option was ignored.

Check the CL or FL environment variable for option specifications.

Command-Line Warning D9007

Error Message
'option1' requires 'option2'; option ignored

An option was specified without a required related option. The compiler ignored *option1*.

Example

```
cl /C program.c
```

In this example, the /C option was given instead of /c. The /C option must be used with /E, /P, or /EP.

Check the CL or FL environment variable for option specifications.

Command-Line Warning D9014

Error Message
invalid value 'number1' for 'option'; assuming 'number2'

The given option was specified with an invalid numeric argument. The compiler assumed the value *number2*.

Example

```
cl /Zp3 program.c
```

In this example, 3 is an invalid value. Valid arguments for the /Zp option are 1, 2, and 4.

Command-Line Warning D9015

Error Message
invalid assembly listing configuration 'option'

The List Assembly option (/FA) can specify combined source and assembly code (/FAs) or combined source and machine code (/FAc). Other specifiers for the List Assembly option are invalid.

Command-Line Warning D9021

Error Message
no action performed

A contradictory set of filenames and switches caused the compiler to perform no operation.

This warning can be generated by giving the /c Compile-Only command-line option and specifying no .c, .cpp, or .cxx files to compile.

Command-Line Warning D9024

Error Message
unrecognized source file type 'filename', object file assumed

The extension of the specified file was not recognized. The file was assumed to be an object file and was passed to the linker.

The following extensions are recognized:

- .c (C source file)
- .cxx (C++ source file)
- .cpp (C++ source file)
- .obj (Object file)
- .lib (Library file)
- .def (Module definition file)
- .exp (Linker exports file, created by LINK /LIB)

Command-Line Warning D9025

Error Message
overriding 'option1' with 'option2'

The *option1* option was specified but was then overridden by *option2*. The *option2* option was used.

If two options specify contradictory or incompatible directives, the directive specified or implied in the option farthest to the right on the command line is used.

If you get this warning when compiling from the development environment, and are not sure where the conflicting options are coming from, consider the following:

- An option can be specified either in code or in the project's project settings. If you look at the compiler's Command Line Property Pages and if you see the conflicting options in the **All Options** field then the options are set in the project's property pages, otherwise, the options are set in source code.

  If the options are set in project's property pages, look on the compiler's Preprocessor property page (with the project node selected in the Solution Explorer). If you do not see the option set there, check the Preprocessor property page settings for each source code file (in Solution Explorer) to make sure it's not added there.

  If the options are set in code it could be set either in code or in the windows headers. You might try creating a preprocessed file (/P) and search it for the symbol.

Command-Line Warning D9026

Error Message
options apply to entire command line

An option was specified on a command after a filename was specified. The option was applied to the file that preceded it.

For example, in the command

```
CL verdi.c /G5 puccini.c
```

the file VERDI.c will be compiled using the /G5 option, not the /G4 default.

This behavior is different from that of some previous versions, which applied only the options specified before the filename, resulting in VERDI.c being compiled using /G4 and PUCCINI.c being compiled using /G5.

Command-Line Warning D9027

Error Message
source file '<filename>' ignored

CL.exe ignored the input source file.

This warning can be caused by a space between the /Fo option and an output filename on a command line with the /c option. For example:

```
cl /c /Fo output.obj input.c
```

Because there is a space between /Fo and `output.obj`, CL.exe takes `output.obj` as the name of the input file. To fix the problem, remove the space:

```
cl /c /Fooutput.obj input.c
```

Command-Line Warning D9028

Error Message
minimal rebuild failure, reverting to normal build

The project .idb file is corrupt. Delete the file and rebuild.

For more information on minimal rebuilds and the .idb file, see the /Gm compiler option.

Command-Line Warning D9029

Error Message
optimization is not available in the standard edition compiler

The standard edition compiler ignores options /Ox, /O1, or /O2.

Command-Line Warning D9032

Error Message
'-doc' : XML documentation comments are not supported by the C compiler

XML documentation is supported for C++ files but not for C files. To remedy this warning, remove the **/doc** command line option or compile the file as a C++ file.

Example
The following command line example generates warning D9032:

```
cl /doc filename.c
```

To fix the warning in the above code example, either remove the **/doc** command line option, rename the file to *filename*.cpp and compile it as a C++ file, or include the /TP or /Tp command line option to force the file to be compiled as C++.

See Also
Reference
Compiler Options
Other Resources
Command-Line Errors D8000 Through D9999

Command-Line Warning D9035

Error Message

An option was passed to the compiler that will be removed in future version of the compiler. option will continue to work in this release. But you may want to update your build configuration now, so that when you update the next version of Visual C++, your projects continue to compile.

option ' *option* ' has been deprecated and will be removed in a future release

This warning will be followed by another warning, which will give the suggested replacement for ***option***.

Command-Line Warning D9036

Error Message

D9036 is a continuation of Command-Line Warning D90356254f933-e37a-45ba-b860-1a870d1bc8e8.

' *option_2* ' instead of ' *option_1* '

Command-Line Warning D9038

Error Message
/ZI is not supported on this platform; use /Zi instead

/ZI is only supported for x86; see /Z7, /Zi, /ZI (Debug Information Format) for more information.

Command-Line Warning D9040

Error Message
ignoring option '/analyze'; Code Analysis warnings are not available in this edition of the compiler

The **/analyze** command line option is supported only on x86 versions of Visual Studio Team System. It is not available for x64 or Itanium compilers, or for other versions of Visual Studio besides Visual Studio Team System. To remedy this warning, either switch to the x86 version of Visual Studio Team System, or remove the command line option.

See Also

Reference
Compiler Options
Other Resources
Command-Line Errors D8000 Through D9999

Command-Line Warning D9041

Error Message
invalid value 'value' for '/option'; assuming 'value'; add '/analyze' to command-line options when specifying this warning

A Code Analysis warning number was added to the **/wd**, **/we**, **/wo**, or **/wl** command line option without also specifying the **/analyze** command line option. To remedy this error, either add the **/analyze** command line option, or remove the invalid warning number from the appropriate **/w** command line option.

Example
The following command line example generates the warning D9041:

```
cl /EHsc /LD /wd6001 filename.cpp
```

To fix the warning, add the **/analyze** command line option. If **/analyze** is not supported on your version of the compiler, remove the invalid warning number from the **/wd** option.

See Also
Reference
Compiler Options
Other Resources
Command-Line Errors D8000 Through D9999

Command-Line Warning D9042

Error Message
invalid value 'value' for '/option'; assuming 'value'; Code Analysis warnings are not available in this edition of the compiler

A Code Analysis warning number was added to the **/wd**, **/we**, **/wo**, or **/wl** command line option and the **/analyze** command line option was specified on a platform that does not support Code Analysis. To remedy this warning, either switch to the x86 version of Visual Studio Team System, or remove the **/analyze** command line option.

Example
The following command line example generates the warning D9042 on an x64 or Itanium system:

```
cl /EHsc /LD /wd6001 /analyze filename.cpp
```

To remedy this warning, either switch to the x86 version of Visual Studio Team System, or remove the **/analyze** and **/wd6001** command line options.

See Also
Reference
Compiler Options
Other Resources
Command-Line Errors D8000 Through D9999

Command-Line Warning D9043

Error Message

invalid value 'warning_level' for 'compiler_option'; assuming '4999'; Code Analysis warnings are not associated with warning levels

Example

The following sample generates C9043.

```
// D9043.cpp
// compile with: /analyze /w16001
// D9043 warning expected
int main() {}
```

Command-Line Warning D9044

Error Message
Before using /errorReport:send, you must read and agree to the Data Collection Policy at "url".

Before you can use **/errorReport:send** to send information on an internal compiler error you must agree to Microsoft's policy on data collection.

The text of the D9044 diagnostic will refer you to a website where you can agree to Microsoft's policy on data collection.

For more information on error reporting, see /errorReport (Report Internal Compiler Errors).

Linker Tools Errors and Warnings

LINK, LIB, DUMPBIN, and EDITBIN generate these errors and warnings. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, type the error number in the **Look for** box in the Index, or click a link below.

Linker Tools Error LNK1000

Linker Tools Error LNK1101

Linker Tools Error LNK1102

Linker Tools Error LNK1103

Linker Tools Error LNK1104

Linker Tools Error LNK1105

Linker Tools Error LNK1106

Linker Tools Error LNK1107

Linker Tools Error LNK1108

Linker Tools Error LNK1109

Linker Tools Error LNK1110

Linker Tools Error LNK1111

Linker Tools Error LNK1112

Linker Tools Error LNK1113

Linker Tools Error LNK1115

Linker Tools Error LNK1117

Linker Tools Error LNK1118

Linker Tools Error LNK1119

Linker Tools Error LNK1120

Linker Tools Error LNK1121

Linker Tools Error LNK1123

Linker Tools Error LNK1127

Linker Tools Error LNK1129

Linker Tools Error LNK1130

Linker Tools Error LNK1131

Linker Tools Error LNK1132

Linker Tools Error LNK1136

Linker Tools Error LNK1137

Linker Tools Error LNK1140

Linker Tools Error LNK1141

Linker Tools Error LNK1143

Linker Tools Error LNK1145

Linker Tools Error LNK1146

Linker Tools Error LNK1147

Linker Tools Error LNK1148

Linker Tools Error LNK1149

Linker Tools Error LNK1152

Linker Tools Error LNK1154

Linker Tools Error LNK1155

Linker Tools Error LNK1156

Linker Tools Error LNK1158

Linker Tools Error LNK1159

Linker Tools Error LNK1160

Linker Tools Error LNK1161

Linker Tools Error LNK1162

Linker Tools Error LNK1163

Linker Tools Error LNK1164

Linker Tools Error LNK1165

Linker Tools Error LNK1166

Linker Tools Error LNK1167

Linker Tools Error LNK1168

Linker Tools Error LNK1169

Linker Tools Error LNK1170

Linker Tools Error LNK1171

Linker Tools Error LNK1172

Linker Tools Error LNK1173

Linker Tools Error LNK1174

Linker Tools Error LNK1175

Linker Tools Error LNK1178

Linker Tools Error LNK1179

Linker Tools Error LNK1180

Linker Tools Error LNK1181

Linker Tools Error LNK1182

Linker Tools Error LNK1183

Linker Tools Error LNK1184

Linker Tools Error LNK1185

Linker Tools Error LNK1186

Linker Tools Error LNK1187

Linker Tools Error LNK1188

Linker Tools Error LNK1189

Linker Tools Error LNK1190

Linker Tools Error LNK1194

Linker Tools Error LNK1195

Linker Tools Error LNK1196

Linker Tools Error LNK1197

Linker Tools Error LNK1198
Linker Tools Error LNK1199
Linker Tools Error LNK1200
Linker Tools Error LNK1201
Linker Tools Error LNK1207
Linker Tools Error LNK1209
Linker Tools Error LNK1210
Linker Tools Error LNK1211
Linker Tools Error LNK1212
Linker Tools Error LNK1213
Linker Tools Error LNK1214
Linker Tools Error LNK1215
Linker Tools Error LNK1218
Linker Tools Error LNK1219
Linker Tools Error LNK1220
Linker Tools Error LNK1221
Linker Tools Error LNK1223
Linker Tools Error LNK1224
Linker Tools Error LNK1227
Linker Tools Error LNK1229
Linker Tools Error LNK1230
Linker Tools Error LNK1232
Linker Tools Error LNK1233
Linker Tools Error LNK1234
Linker Tools Error LNK1235
Linker Tools Error LNK1236
Linker Tools Error LNK1237
Linker Tools Error LNK1240
Linker Tools Error LNK1241
Linker Tools Error LNK1242
Linker Tools Error LNK1243
Linker Tools Error LNK1244
Linker Tools Error LNK1245
Linker Tools Error LNK1246
Linker Tools Error LNK1248
Linker Tools Error LNK1249
Linker Tools Error LNK1250
Linker Tools Error LNK1252
Linker Tools Error LNK1253
Linker Tools Error LNK1255

Linker Tools Error LNK1256
Linker Tools Error LNK1257
Linker Tools Error LNK1258
Linker Tools Error LNK1260
Linker Tools Error LNK1261
Linker Tools Error LNK1262
Linker Tools Error LNK1263
Linker Tools Error LNK1264
Linker Tools Error LNK1265
Linker Tools Error LNK1266
Linker Tools Error LNK1267
Linker Tools Error LNK1268
Linker Tools Error LNK1269
Linker Tools Error LNK1270
Linker Tools Error LNK1272
Linker Tools Error LNK1274
Linker Tools Error LNK1276
Linker Tools Error LNK1277
Linker Tools Error LNK1278
Linker Tools Error LNK1279
Linker Tools Error LNK1280
Linker Tools Error LNK1281
Linker Tools Error LNK1282
Linker Tools Error LNK1283
Linker Tools Error LNK1285
Linker Tools Error LNK1286
Linker Tools Error LNK1287
Linker Tools Error LNK1290
Linker Tools Error LNK1291
Linker Tools Error LNK1292
Linker Tools Error LNK1293
Linker Tools Error LNK1294
Linker Tools Error LNK1295
Linker Tools Error LNK1296
Linker Tools Error LNK1299
Linker Tools Error LNK1300
Linker Tools Error LNK1301
Linker Tools Error LNK1302
Linker Tools Error LNK1303
Linker Tools Error LNK1304

Linker Tools Error LNK1306

Linker Tools Error LNK1307

Linker Tools Error LNK1308

Linker Tools Error LNK1309

Linker Tools Error LNK1310

Linker Tools Error LNK1311

Linker Tools Error LNK1312

Linker Tools Error LNK1313

Linker Tools Error LNK1314

Linker Tools Error LNK1315

Linker Tools Error LNK1316

Linker Tools Error LNK1561

Linker Tools Error LNK2001

Linker Tools Error LNK2003

Linker Tools Error LNK2004

Linker Tools Error LNK2005

Linker Tools Error LNK2008

Linker Tools Error LNK2009

Linker Tools Error LNK2011

Linker Tools Error LNK2013

Linker Tools Error LNK2014

Linker Tools Error LNK2016

Linker Tools Error LNK2017

Linker Tools Error LNK2019

Linker Tools Error LNK2020

Linker Tools Error LNK2021

Linker Tools Error LNK2022

Linker Tools Error LNK2023

Linker Tools Error LNK2024

Linker Tools Error LNK2026

Linker Tools Error LNK2027

Linker Tools Error LNK2028

Linker Tools Error LNK2029

Linker Tools Error LNK2030

Linker Tools Error LNK2031

Linker Tools Error LNK2032

Linker Tools Error LNK2033

Linker Tools Error LNK2034

Linker Tools Error LNK2035

Linker Tools Error LNK4196

Linker Tools Warning LNK4001

Linker Tools Warning LNK4002

Linker Tools Warning LNK4003

Linker Tools Warning LNK4006

Linker Tools Warning LNK4010

Linker Tools Warning LNK4012

Linker Tools Warning LNK4013

Linker Tools Warning LNK4014

Linker Tools Warning LNK4017

Linker Tools Warning LNK4018

Linker Tools Warning LNK4019

Linker Tools Warning LNK4020

Linker Tools Warning LNK4022

Linker Tools Warning LNK4030

Linker Tools Warning LNK4031

Linker Tools Warning LNK4037

Linker Tools Warning LNK4038

Linker Tools Warning LNK4039

Linker Tools Warning LNK4040

Linker Tools Warning LNK4041

Linker Tools Warning LNK4042

Linker Tools Warning LNK4043

Linker Tools Warning LNK4044

Linker Tools Warning LNK4047

Linker Tools Warning LNK4048

Linker Tools Warning LNK4049

Linker Tools Warning LNK4051

Linker Tools Warning LNK4052

Linker Tools Warning LNK4056

Linker Tools Warning LNK4060

Linker Tools Warning LNK4061

Linker Tools Warning LNK4062

Linker Tools Warning LNK4065

Linker Tools Warning LNK4066

Linker Tools Warning LNK4067

Linker Tools Warning LNK4068

Linker Tools Warning LNK4069

Linker Tools Warning LNK4070

Linker Tools Warning LNK4071

Linker Tools Warning LNK4072

Linker Tools Warning LNK4073

Linker Tools Warning LNK4075

Linker Tools Warning LNK4076

Linker Tools Warning LNK4077

Linker Tools Warning LNK4078

Linker Tools Warning LNK4079

Linker Tools Warning LNK4081

Linker Tools Warning LNK4085

Linker Tools Warning LNK4086

Linker Tools Warning LNK4087

Linker Tools Warning LNK4088

Linker Tools Warning LNK4092

Linker Tools Warning LNK4093

Linker Tools Warning LNK4094

Linker Tools Warning LNK4096

Linker Tools Warning LNK4097

Linker Tools Warning LNK4098

Linker Tools Warning LNK4099

Linker Tools Warning LNK4102

Linker Tools Warning LNK4103

Linker Tools Warning LNK4104

Linker Tools Warning LNK4105

Linker Tools Warning LNK4108

Linker Tools Warning LNK4194

Linker Tools Warning LNK4195

Linker Tools Warning LNK4197

Linker Tools Warning LNK4198

Linker Tools Warning LNK4199

Linker Tools Warning LNK4200

Linker Tools Warning LNK4202

Linker Tools Warning LNK4203

Linker Tools Warning LNK4204

Linker Tools Warning LNK4205

Linker Tools Warning LNK4206

Linker Tools Warning LNK4207

Linker Tools Warning LNK4208

Linker Tools Warning LNK4209

Linker Tools Warning LNK4210

Linker Tools Warning LNK4216

Linker Tools Warning LNK4217

Linker Tools Warning LNK4219

Linker Tools Warning LNK4220

Linker Tools Warning LNK4221

Linker Tools Warning LNK4222

Linker Tools Warning LNK4223

Linker Tools Warning LNK4224

Linker Tools Warning LNK4225

Linker Tools Warning LNK4226

Linker Tools Warning LNK4227

Linker Tools Warning LNK4228

Linker Tools Warning LNK4229

Linker Tools Warning LNK4232

Linker Tools Warning LNK4233

Linker Tools Warning LNK4236

Linker Tools Warning LNK4237

Linker Tools Warning LNK4238

Linker Tools Warning LNK4239

Linker Tools Warning LNK4240

Linker Tools Warning LNK4241

Linker Tools Warning LNK4242

Linker Tools Warning LNK4245

Linker Tools Warning LNK4246

Linker Tools Warning LNK4247

Linker Tools Warning LNK4248

Linker Tools Warning LNK4249

Linker Tools Warning LNK4251

Linker Tools Warning LNK4252

Linker Tools Warning LNK4253

Linker Tools Warning LNK4254

Linker Tools Error LNK1000

Error Message
unknown error; consult documentation for technical support options

Note the circumstances of the error, try to isolate the problem and create a reproducible test case, then contact Microsoft Product Support Services. For information on how to investigate and report these errors, see http://support.microsoft.com/default.aspx?scid=kb;en-us;134650.

You may get this error if you mix standard header files (for example, dos.h) and your own files. **#include** the standard headers first, followed by your own header files.

Linker Tools Error LNK1101

Error Message
incorrect MSPDB80.DLL version; recheck installation of this product

The version of MSPDB80.dll available on your system does not match the version required by this tool.

Linker Tools Error LNK1102

Error Message
out of memory

There was not enough memory for the tool to run.

This error can be caused by the paging file exceeding available disk space.

If a shortage of disk space is not the cause, note the circumstances of the error, try to isolate the problem, and create a reproducible test case. Then consult the technical-support help file, the technical-support section in one of your manuals, or Microsoft Product Support Services.

Linker Tools Error LNK1103

Error Message
debugging information corrupt; recompile module

This error can be caused because the compilation was terminated before a valid object file was created.

Recompile the given object file. If recompiling does not correct the problem, note the circumstances of the error, try to isolate the problem, and create a reproducible test case. Then consult the technical-support help file, the technical-support section in one of your manuals, or Microsoft Product Support Services.

See Corrupt Object File for more information.

Linker Tools Error LNK1104

Error Message
cannot open file 'filename'

The tool could not open the given file.

To fix by checking the following possible causes:

- Not enough disk space.
- File does not exist.
- When specifying libraries in a project's property pages dialog box, library names should be separated by spaces (and not commas).
- Incorrect filename or path.
- Invalid drive specification.
- Insufficient file permissions.
- Path for *filename* expands to more than 260 characters.
- If the given file is named `LNKn`, which is a filename generated by the linker for a temporary file, the directory specified in the TMP environment variable may not exist, or more than one directory is specified for the TMP environment variable. (Only one directory path should be specified for the TMP environment variable.)
- If the error message occurs for a library name, and you recently ported the .mak file from a previous Microsoft Visual C++ development system, the library may no longer be valid. Ensure that the library still exists in this circumstance.
- Another program may have the file open and the linker cannot write to it.
- Incorrect LIB environment variable. For information on how to update the LIB environment variable, see VC++ Directories, Projects and Solutions, Options Dialog Box . Make sure any directories with libraries you need are listed here.

The linker uses temporary files in several cases. Even if you have sufficient disk space, a very large link can deplete or fragment the address space.

To fix by using the following possible solutions

- Use /OPT (Optimizations); doing transitive comdat elimination reads all the object files multiple times.
- Upgrade to Windows XP.

Linker Tools Error LNK1105

Error Message
cannot close file 'filename'

The tool could not close the given file.

This error can be caused if there was insufficient disk space available.

Linker Tools Error LNK1106

Error Message
invalid file or disk full: cannot seek to location

The tool could not read or write to *location* in a memory-mapped file.

To fix by checking the following possible causes

1. Disk full.

   Free up some space and link again.

2. Trying to link over a network.

   Some networks do not fully support the memory-mapped files used by the linker. Try linking on your local disk.

3. Bad block on your disk.

   Although the operating system and disk hardware should have detected such an error, you may want to run a disk-checking program.

4. Out of heap space.

   See C1060 for more information.

Linker Tools Error LNK1107

Error Message
invalid or corrupt file: cannot read at location

The tool could not read the file. Recreate the file.

LNK1107 could also occur if you attempt to pass a module (.dll or .netmodule extension created with /clr:noAssembly or /NOASSEMBLY) to the linker; pass the .obj file instead.

If you compile the following sample:

```
// LNK1107.cpp
// compile with: /clr /LD
public ref class MyClass {
public:
   void Test(){}
};
```

and then specify **link LNK1107.dll** on the command line, you will get LNK1107. To resolve the error, specify **link LNK1107.obj** instead.

Linker Tools Error LNK1108

Error Message
cannot write file at location

The tool could not write to the file.

To fix by checking the following possible causes

1. There was not enough disk space to create the file.
2. The drive being written to was not available, possibly due to a network problem.

Linker Tools Error LNK1109

Error Message
cannot remove file 'filename'

LIB could not delete the given file.

Before LIB writes the new version of a library, it removes the existing library file.

To fix by checking the following possible causes

1. The given file does not have the appropriate permissions.
2. The drive containing the file was not available, possibly due to a network problem.

Linker Tools Error LNK1110

Error Message
cannot rename file 'filename'

LIB could not rename the given file.

When LIB builds a new version of a library, it creates a temporary file, then renames the file.

To fix by checking the following possible causes

1. The given file does not have the appropriate permissions.
2. The drive containing the file was not available, possibly due to a network problem.

Linker Tools Error LNK1111

Error Message
invalid /BASE argument 'argument'

The /BASE option was incorrectly specified. Either no argument was specified or *argument* is not a number.

Linker Tools Error LNK1112

Error Message
module machine type 'type1' conflicts with target machine type 'type2'

The object files specified as input were compiled for different machine types.

For example, if you try to link an object file compiled with **/clr** and an object file compiled with **/clr:pure** (machine type CEE), the linker will generate LNK1112.

Similarly, if you create a module with the x64 compiler and a module with the x86 compiler, and try to link them, the linker will generate LNK1112.

Linker Tools Error LNK1113

Error Message
invalid machine type type

The machine type specified in the object header is not valid.

This error can be caused if the file is corrupt. Rebuild the object.

For more information, see Corrupt Object File.

Linker Tools Error LNK1115

Error Message
/MACHINE option required

LINK or LIB cannot determine the machine type for objects that are provided entirely from libraries. Either specify the /MACHINE option with the appropriate machine type or specify at least one object file in the input.

Linker Tools Error LNK1117

Error Message
syntax error in option 'option'

The option was not correctly specified.

Make sure you are not using a linker option that is no longer valid in the current version of Visual C++.

Linker Tools Error LNK1118

Error Message
syntax error in 'keyword' statement

A module-definition statement is specified incorrectly.

Linker Tools Error LNK1119

Error Message
invalid ordinal number 'argument'

The argument following the at sign (@) in an ordinal specification was not a valid number.

An ordinal number is an optional argument in either an /EXPORT option in a LINK or LIB command, or an EXPORTS statement in a module-definition file. It is an index into the exports table. The number must be an integer in the range 1–65535.

Linker Tools Error LNK1120

Error Message
number unresolved externals

Error LNK1120 gives you a count (*number*) of unresolved externals for this link. The conditions that cause unresolved externals are described with error LNK2001, which precedes this error message (once for each unresolved external).

For more information, see http://support.microsoft.com/default.aspx?scid=kb;en-us;815650.

Linker Tools Error LNK1121

Error Message
duplicate ordinal number 'number'

The given ordinal number was specified more than once in either an /EXPORT option in a LINK or LIB command, or in an EXPORTS statement in a module-definition file. Ordinal numbers must be unique integers in the range 1–65535.

Linker Tools Error LNK1123

Error Message
failure during conversion to COFF: file invalid or corrupt

The object or resource could not be converted to Common Object File Format (COFF).

This tool requires the format of all input files to be COFF. If an input file is not COFF, LINK automatically converts 32-bit OMF objects to COFF, or the tool runs CVTRES.EXE to convert resource files.

To fix by checking the following possible causes

1. The file is corrupt.
2. The file is not a valid file type. An example of an invalid type is a 16-bit OMF object.

See Also

Reference
EDITBIN Reference
DUMPBIN Reference

Linker Tools Error LNK1127

Error Message
library is corrupt

The library file is corrupt. Rebuild the library.

Linker Tools Error LNK1129

Error Message
cannot find resolution for weak extern symbol

The given weak external symbol does not have a default resolution.

This error can be caused if the symbol table is corrupt.

Rebuild the object file.

See Corrupt Object File for more information.

Linker Tools Error LNK1130

Error Message

Linker miscalc (base relocations off by number)

There was not enough space in the image to write base relocations.

Linker Tools Error LNK1131

Error Message
no library file specified

The LIB /EXTRACT command required a library as input, but a library filename was not specified.

Linker Tools Error LNK1132

Error Message
invalid format for MS-DOS stub file 'filename'

The *filename* specified with the /STUB option was not a valid real-mode MS-DOS executable (.exe) file.

Linker Tools Error LNK1136

Error Message
invalid or corrupt file

The input file either has a corrupt header or is zero size or abnormally small.

LNK1136 may also be caused if the file was incorrectly specified as an additional dependency.

See Corrupt Object File for more information.

Linker Tools Error LNK1137

Error Message
invalid argument 'arg' specified with /SECTION

Either the name or the attributes argument to the /SECTION option is specified incorrectly.

Linker Tools Error LNK1140

Error Message
too many modules for program database; link with /PDB:NONE

The project contains more than 4096 modules.

To fix by using the following possible solutions

1. Relink using /PDB:NONE.
2. Compile some modules without debugging information.
3. Reduce the number of modules.

Linker Tools Error LNK1141

Error Message
failure during build of exports file

LINK could not build the exports (.exp) file.

Causes of this error include system problems such as insufficient memory and syntax errors in options or module-definition statements.

This error is preceded by another error that gives additional information.

See Also

Reference
.exp Files as Linker Input

Linker Tools Error LNK1143

Error Message
invalid or corrupt file: no symbol for COMDAT section number

This error can be caused if the object file is corrupt.

Rebuild the file.

See Corrupt Object File for more information.

Linker Tools Error LNK1145

Error Message

/MERGE created circular link for section 'section'

You attempted to merge a section into itself. Check the /MERGE options.

Linker Tools Error LNK1146

Error Message
no argument specified with option 'option'

The given linker option requires an argument. For example, when you specify /IDLOUT, you must also pass a name:

/IDLOUT:MyIDL.idl.

Linker Tools Error LNK1147

Error Message
invalid number specified with option 'option'

The argument to the given option was specified incorrectly.

Linker Tools Error LNK1148

Error Message
failure during conversion to COFF: cannot copy temp file 'filename'

EDITBIN could not convert the input file to COFF.

When EDITBIN.EXE converts a file, it creates a temporary file, then copies the file.

To fix by checking the following possible causes

1. The given file does not have the appropriate permissions.
2. There was not enough disk space to create the file.
3. The drive being written to was not available, possibly due to a network problem.

Linker Tools Error LNK1149

Error Message

output filename matches input filename 'filename'

The output filename specified with the /OUT or /IMPLIB option was the same as an input file.

Linker Tools Error LNK1152

Error Message
cannot resolve one or more undecorated symbols

This error is preceded by one warning LNK4022 for each undecorated symbol that could not be resolved and by at least two warnings LNK4006 for the duplicate symbols found for the undecorated symbol.

Linker Tools Error LNK1154

Error Message

specified import library filename matches exports file 'filename'

The filename specified with `LINK /IMPLIB` or with `LIB /DEF /OUT` conflicted with the *filename* given by LINK or LIB to the exports file. The *filename* is formed from the base name of the main output file and the extension .exp.

See Also

Reference

/IMPLIB (Name Import Library)
/DEF (Specify Module-Definition File)
/OUT (Output File Name)

Linker Tools Error LNK1155

Error Message
special symbol 'symbol' already defined

The given symbol is reserved for use by LINK.

Linker Tools Error LNK1156

Error Message

.sbss section not supported

An object file contained an .sbss section.

Linker Tools Error LNK1158

Error Message
cannot run 'filename'

The given executable file called by LINK is not in the directory that contains LINK nor in a directory specified in the PATH environment variable.

For example, you will get this error if you try to use the PGOPTIMIZE parameter to the /LTCG linker option on a machine with a 32-bit operating system.

Linker Tools Error LNK1159

Error Message
no output file specified

No name was specified for the main output file (executable file or DLL).

LINK derives the default name of the output file from the base name of the first object file. If no object files are specified, and if the /OUT option is not used, this error occurs.

Linker Tools Error LNK1160

Error Message
library with zero objects not allowed

An attempt was made to remove an object from a library that contained only that object. The object was not removed.

Linker Tools Error LNK1161

Error Message
invalid export specification

Either the /EXPORT option or the EXPORTS module-definition statement incorrectly specified an export.

This error can be caused by a typing error.

Linker Tools Error LNK1162

Error Message
expected aux symbol for COMDAT section number

The linker expected to find an auxiliary symbol table for the indicated COMDAT but could not.

This error can be caused if the object file is corrupt.

For more information, see Corrupt Object File.

Linker Tools Error LNK1163

Error Message
invalid selection for COMDAT section number

The byte in the object file indicating the type of COMDAT section is invalid.

This error can be caused if the object file is corrupt.

For more information, see Corrupt Object File.

Linker Tools Error LNK1164

Error Message
section section alignment (number) greater than /ALIGN value

The alignment size for the given section in the object file exceeds the value specified with the /ALIGN option. The **/ALIGN** value must be a power of 2 and must equal or exceed the section alignment given in the object file.

Either recompile with a smaller section alignment or increase the **/ALIGN** value.

Linker Tools Error LNK1165

Error Message
link failed because of fixup errors

The build failed due to fixup errors.

The /FORCE or **/FORCE:UNRESOLVED** option overrides this error.

Linker Tools Error LNK1166

Error Message
cannot adjust code at offset=offset, va=value

LINK was unable to pad the code as required.

Certain instructions are not allowed to cross page boundaries on some processors. LINK attempts to add pads to correct this situation. In this case, LINK could not work around the problem.

Linker Tools Error LNK1167

Error Message
file contains relocs but header has no machine type

A converted COFF object did not have a machine type specified in its header.

This error can be caused by omitting a machine type when converting a .res file in a separate step before linking.

Linker Tools Error LNK1168

Error Message
cannot open filename for writing

The given file does not have write permission.

Linker Tools Error LNK1169

Error Message
one or more multiply defined symbols found

The build failed due to multiple definitions of one or more symbols. This error is preceded by error LNK2005.

The /FORCE or **/FORCE:MULTIPLE** option overrides this error.

Linker Tools Error LNK1170

Error Message
line in command file contains limit or more characters

The length of a line in a command file must be less than the given *limit*.

Linker Tools Error LNK1171

Error Message
unable to load filename

The given DLL was unavailable.

The possible locations for the DLL are the current directory, the system directory, the Windows directory, and the directories specified in the PATH environment variable.

LNK1171 can also indicate that your machine is low on memory, or that the linker cannot load a file required by *filename*.

It is also possible that *filename* can only load at a given address, but that the address is already used.

Linker Tools Error LNK1172

Error Message
more than one object with the same name found in the library; rename object(s)

A library contained two or more objects with the same name.

To fix by using the following possible solutions

1. Rename the objects using unique names and rebuild the library.
2. Link using the /PDB:NONE option.

Linker Tools Error LNK1173

Error Message
unable to find entrypoint 'function' in filename

The function does not exist in the given DLL.

Linker Tools Error LNK1174

Error Message
unable to /REBASE filename; not a valid Win32 image

The file format is invalid.

Linker Tools Error LNK1175

Error Message
failed to /REBASE filename

The rebase operation failed on the given file.

Linker Tools Error LNK1178

Error Message
missing MODEND record; file is invalid or corrupt

The linker tried to convert an OMF object module to COFF format but could not find an expected MODEND (module end) record in the OMF object. The OMF object module is corrupt and needs to be recreated or recopied.

See Corrupt Object File for more information.

Linker Tools Error LNK1179

Error Message
invalid or corrupt file: duplicate COMDAT 'filename'

An object module contains two or more COMDATs with the same name.

This error can be caused by using /H, which limits the length of external names, and /Gy, which packages functions in COMDATs.

Example

In the following code, `function1` and `function2` are identical in the first eight characters. Compiling with **/Gy** and **/H8** produces a link error.

```
void function1(void);
void function2(void);

int main() {
    function1();
    function2();
}

void function1(void) {}
void function2(void) {}
```

Linker Tools Error LNK1180

Error Message
insufficient disk space to complete link

The linker could not complete a file operation because it ran out of disk space.

Free up space on your local or network drive.

Linker Tools Error LNK1181

Error Message
cannot open input file 'filename'

The linker could not find *filename* because it does not exist or the path was not found.

Some common causes for error LNK1181 include:

- *filename* is referenced as an additional dependency on the linker line, but the file does not exist.
- A **/LIBPATH** statement that specifies the directory containing *filename* is missing.

To resolve the above issues, ensure any files referenced on the linker line are present on the system. Also ensure there is a **/LIBPATH** statement for each directory containing a linker-dependent file.

Another possible cause for LNK1181 is that a long file name with embedded spaces was not enclosed in quotation marks. In that case, the linker will only recognize a file name up to the first space, and then assume a file extension of .obj. The solution to this situation is to enclose the long file name (path plus file name) in quotation marks.

For more information, see .lib Files as Linker Input.

See Also

Reference
/LIBPATH (Additional Libpath)

Linker Tools Error LNK1182

Error Message
cannot have more than 64K exports

You have reached the linker limit of 65,536 exports.

Linker Tools Error LNK1183

Error Message
invalid or corrupt file: extended relocation count number less than 65535

An extended relocation count that is less than 0xFFFF indicates that the COFF object file is corrupt.

See Corrupt Object File for more information.

Linker Tools Error LNK1184

Error Message
invalid section name 'section' specified in option or directive 'option'

You gave the linker an invalid section name in *option*.

This error can be caused by a `$` symbol or blank in name.

Linker Tools Error LNK1185

Error Message
invalid section name 'section' specified

A .def file gave the linker an invalid section name.

This error can be caused by a `$` symbol or blank in name.

Linker Tools Error LNK1186

Error Message
invalid or corrupt COFF object; reloc to undefined static symbol 'symbol'

The COFF object module is corrupt. It contains a relocation entry for an undefined static symbol.

See Corrupt Object File for more information.

Linker Tools Error LNK1187

Error Message

Corrupt object - unmatched name relocation; ignored

The object file contains bad relocation information. Recopy or recreate the object file.

See Corrupt Object File for more information.

Linker Tools Error LNK1188

Error Message
BADFIXUPSECTION:: invalid fixup target 'symbol'; possible zero length section

During a VxD link, the target of a relocation did not have a section. With LINK386 (an older version), an OMF GROUP record (generated by a MASM GROUP directive) may have been used to combine the zero length section with another non-zero length section. COFF format does not support the GROUP directive and zero-length sections. When LINK automatically converts this type of OMF objects to COFF, this error may occur.

Linker Tools Error LNK1189

Error Message
LIBTOOMANYMEMBERS:: library limit of number objects exceeded

The limit of 65535 objects or members in a library has been exceeded.

Linker Tools Error LNK1190

Error Message
invalid fixup found, type type

The object file is corrupt. Recompile.

Linker Tools Error LNK1194

Error Message

cannot delay-load 'dll name' due to import of data symbol 'symbol name'; link without /DELAYLOAD:dll name

You cannot delay load a DLL if data is imported from it.

Linker Tools Error LNK1195

Error Message
target machine 'machine' requires 'option'

Add the required option.

Linker Tools Error LNK1196

Error Message
invalid or corrupt import object: unknown version

The import library is corrupted. Rebuild the library.

Linker Tools Error LNK1197

Error Message
invalid or corrupt import object: unknown type

The import library is corrupt. Rebuild the library.

Linker Tools Error LNK1198

Error Message
invalid or corrupt import object: unknown name type

The import library is corrupt. Rebuild the library.

Linker Tools Error LNK1199

Error Message
invalid or corrupt import object: non-zero reserved fields

The import library is corrupt. Rebuild the library.

Linker Tools Error LNK1200

Error Message
error reading program database 'filename'

The program database (PDB) could not be read.

This error can be caused by file corruption.

If *filename* is the PDB for an object file, recompile the object file using /Zi.

If *filename* is the PDB for the main output file, and this error occurred during an incremental link, delete the PDB and relink.

Linker Tools Error LNK1201

Error Message
error writing to program database 'filename'; check for insufficient disk space, invalid path, or insufficient privilege

LINK could not write to the program database (PDB) for the output file.

To fix by checking the following possible causes

1. File is corrupt. Delete the PDB file and relink.
2. Not enough disk space to write the file.
3. Drive is not available, possibly due to a network problem.
4. The debugger is active on the program you are trying to link.
5. Out of heap space. See C1060 for more information.

Linker Tools Error LNK1207

Error Message
incompatible PDB format in 'filename'; delete and rebuild

This version of LINK cannot write to the existing program database (PDB). Delete the PDB and rebuild.

Linker Tools Error LNK1209

Error Message
program database 'filename' differs from previous link; relink or rebuild

The given program database (PDB) is invalid and possibly corrupt. Relink. If *filename* is also the PDB for an object file, recompile to recreate the PDB.

Linker Tools Error LNK1210

Error Message
exceeded internal ILK size limit; link with /INCREMENTAL:NO

There was not enough memory available for LINK to create the incremental status (.ilk) file.

Linker Tools Error LNK1211

Error Message
precompiled type information not found; 'filename' not linked or overwritten

The given object file, compiled with /Yc, was not specified in the LINK command or was overwritten.

If you are creating a debug library that uses precompiled headers and if you specify /Yc and /Z7, Visual C++ generates a precompiled object file that contains CodeView debugging information. The error occurs only when you store the precompiled object file in a library, use the library to build an executable image, and the object files that are referenced have no transitive references to any of the functions the precompiled object file defines.

There are two methods to work around this situation:

- Specify the /Yd compiler option to add the CodeView information from the precompiled header to each object module. This method is less desirable because it generally produces large object modules that can increase the time required to link the application.
- Specify /Yl and pass the name of any arbitrary string, when you create a precompiled header file that does not contain any function definitions. This directs the compiler to create a symbol in the precompiled object file and to emit a reference to that symbol in each object file that used the precompiled header file that is associated with the precompiled object file.

When you compile a module with **/Yc** and **/Yl**, the compiler creates a symbol similar to `__@@_PchSym_@00@...@symbol_name`, where the ellipsis (...) represents a compiler-generated character string, and stores it in the object module. Any source file that you compile with this precompiled header refers to the specified symbol, which causes the linker to include the object module and its debugging information from the library.

For more information on this error, see Knowledge Base article Q102697 PRB: Build Errors Using Precompiled Header in Debugging Lib.

Linker Tools Error LNK1212

Error Message
error opening program database; file is in use

The PDB is in use by another application.

Linker Tools Error LNK1213

Error Message
unexpected import object encountered

The import library is corrupt. Rebuild the library.

Linker Tools Error LNK1214

Error Message
initialization failed (hresult) : 'reason'

A function in the common language run-time (CLR) library failed. *reason* is a description of the failure from the CLR and may be blank.

This error may be caused by a partial, or corrupt installation of the Visual C++ toolset. Reinstalling Visual C++ may resolve this error.

Linker Tools Error LNK1215

Error Message
metadata operation failed (HRESULT) : error

The linker received an error from the .NET runtime while attempting to do a metadata update through the .NET runtime.

HRESULT is the HRESULT from the .NET runtime method. *error* is the .NET-supplied text.

You probably have a mismatched linker and .NET runtime; reinstall Visual C++.

Linker Tools Error LNK1218

Error Message
warning treated as error; no output file generated

When you link with **/WX**, any linker warnings will be treated as errors, and no output file will be created.

See /WX (Treat Linker Warnings as Errors) for more information.

Linker Tools Error LNK1219

Error Message

ILK corrupt or not found; link with full command line

In an edit and continue session, the image could not automatically be relinked. Rebuild your project.

Linker Tools Error LNK1220

Error Message
'linker option1' requires 'linker option2' specification

The use of *linker option1* requires that you also specify *linker option2*.

See Linker Options for more information.

Linker Tools Error LNK1221

Error Message
a subsystem can't be inferred and must be defined

The linker does not have enough information to infer which subsystem you want to target.

To fix this error, use /SUBSYSTEM.

Linker Tools Error LNK1223

Error Message
invalid or corrupt file: file contains invalid .pdata contributions

For RISC platforms that use pdata, this error will occur if the compiler emitted a .pdata section with unsorted entries.

Linker Tools Error LNK1224

Error Message
invalid image base address

You specified an invalid base address for the image. Base addresses must be 64KB aligned (the last four hex digits must be zero) and image base must fit within a 32-bit signed or unsigned value.

Linker Tools Error LNK1227

Error Message

conflicting weak extern definition for 'weak_symbol_name'. New default 'default_symbol_name' conflicts with old default 'other_default_symbol_name' in filename_for_this_other_default_symbol

The linker encountered a weak, external-symbol definition with multiple, varying default definitions.

Linker Tools Error LNK1229

Error Message
Commit size greater than reserve size in 'linker_option'

If you specify both *commit* and *reserve* values when using the /HEAP or /STACK linker options, *commit* size has to be less than or equal to the *reserve* size.

Linker Tools Error LNK1230

Error Message
exceeded incremental image size limit; link with /INCREMENTAL:NO

On RISC platforms the linker can incrementally link images up to approximately 1GB in size: after that, /INCREMENTAL:NO must be specified.

Linker Tools Error LNK1232

Error Message
invalid selection for comdat symbol 'symbol'

The compiler produced a comdat selection type unknown by the linker. The section number for the offending comdat is unknown, so the symbol name is reported.

Linker Tools Error LNK1233

Error Message
conversion to COFF object failed

Code generation failed. Examine any compiler errors to determine the source of the failure.

Linker Tools Error LNK1234

Error Message
corrupt or invalid public symbols

The public symbol table for the given object appears to be corrupt.

This error may represent a failure of the compiler; contact Microsoft Product Support Services.

Linker Tools Error LNK1235

Error Message
corrupt or invalid COFF symbol table

The compiler-generated content for a section in the given object appears to be corrupt.

This error may represent a failure of the compiler. For information on getting help for this issue, see Getting Help from Microsoft Product Support Services (Visual Studio).

Linker Tools Error LNK1236

Error Message
corrupt or invalid COFF sections

The compiler-generated content for a section in the given object appears to be corrupt.

This error may represent a failure of the compiler. For information on getting help for this issue, see Getting Help from Microsoft Product Support Services (Visual Studio).

Linker Tools Error LNK1237

Error Message
during code generation, compiler introduced reference to symbol 'symbol' defined in module 'module' compiled with /GL

During code generation, the compiler should not introduce symbols that are later resolved to definitions compiled **/GL**. *symbol* is a symbol that was introduced and later resolved to a definition compiled with **/GL**.

For more information, see /GL (Whole Program Optimization).

To resolve LNK1237, do not compile the symbol with **/GL** or use /INCLUDE (Force Symbol References) to force a reference to the symbol.

Example

The following sample generates LNK1237. To resolve this error, do not initialize the array in LNK1237_a.cpp and add **/include:__chkstk** to the link command.

```
// LNK1237_a.cpp
int main() {
   char c[5000] = {0};
}
```

```
// LNK1237_b.cpp
// compile with: /GS- /GL /c LNK1237_a.cpp
// processor: x86
// post-build command: lib LNK1237_b.obj /LTCG & link LNK1237_a.obj LNK1237_b.lib /nodefaul
tlib /entry:main /LTCG
// LNK1237 expected
extern "C" void _chkstk(size_t s) {}
```

Linker Tools Error LNK1240

Error Message
failed to compile IDL content

The linker spawned MIDL to compile embedded IDL but there was a problem. Check the errors specified by MIDL.

Linker Tools Error LNK1241

Error Message
resource file 'resource file' already specified

This error is generated if you run **cvtres** manually from the command line and if you then pass the resulting .obj file to the linker in addition to other .res files.

To specify multiple .res files, pass them all to the linker as .res files, not from within .obj files created by **cvtres**.

Linker Tools Error LNK1242

Error Message
'export' is an invalid export symbol name

An exported name contains an invalid character, such as a period (.) character.

Valid characters in an export name are anything allowed in a C/C++ identifier plus @ and ?.

Linker Tools Error LNK1243

Error Message

invalid or corrupt file: COMDAT section 'section number' associated with following section 'section number'

The linker has detected a corrupt .obj file. A section must be associated with another section that precedes it.

Linker Tools Error LNK1244

Error Message
invalid image type 'image'; link without 'linker option'

The specified linker option (*linker option*) is not valid for the specified image type (*image*).

Linker Tools Error LNK1245

Error Message
invalid subsystem 'subsystem' specified; /SUBSYSTEM must be WINDOWS, WINDOWSCE, or CONSOLE

/clr was used to compile the object and one of the following conditions was true:

- A custom entry point was defined (/ENTRY), such that, the linker could not infer a subsystem.
- A value was passed to the /SUBSYSTEM linker option that is not valid for /clr objects.

For both situations, the resolution is to specify a valid value to the /SUBSYSTEM linker option.

Linker Tools Error LNK1246

Error Message

'linker option' not compatible with 'machine type' target machine; link without 'linker option'

Certain linker options are not compatible with certain machine types.

Linker Tools Error LNK1248

Error Message
image size ('size') exceeds maximum allowable size (80000000)

The linker determined that the size of the output file will exceed the largest possible size for a program image. You may want to make your program into multiple DLLs. For more information on the program executable (PE), see Peering Inside the PE: A Tour of the Win32 Portable Executable File Format.

Linker Tools Error LNK1249

Error Message
image exceeds maximum extent with base address address and size size

The size of the program is too great, given the base address. Try to specify a lower base address with the /BASE linker option.

Linker Tools Error LNK1250

Error Message
failed to merge IDL content

The linker's call to the compiler to merge IDL content in .obj files failed. To resolve this error, refer to the compiler's error messages to see what the problem is.

Linker Tools Error LNK1252

Error Message

Exports not supported for 'machine' target machine

Exports, for example with __declspec(dllexport), are not allowed on some /MACHINE types, for example CEE.

Linker Tools Error LNK1253

Error Message
illegal export of managed symbol "symbol"

An attempt was made to export a managed symbol either by using /EXPORT or by using a module definition file and the decorated name of the object.

Try using dllexport.

Linker Tools Error LNK1255

Error Message
link failed because of metadata errors

The link failed while merging metadata. The metadata issues must be resolved before the link can be successful.

Linker Tools Error LNK1256

Error Message
ALINK operation failed : reason

One reason for LNK1256 is a mismatch between the linker and Alink.dll. Compare the versions of these files on the product CD to the files on your computer.

Another reason for LNK1256 is an incorrect version number for an assembly: 65535 is not allowed for any part of the assembly version number. The valid range for assembly versions is 0 – 65534.

The following sample generates LNK1256:

```
// LNK1256.cpp
// compile with: /clr /LD
// LNK1256 expected
[assembly:System::Reflection::AssemblyVersionAttribute("1.0.65535")];
public class CMyClass {
public:
   int value;
};
```

Linker Tools Error LNK1257

Error Message
code generation failed

When compiling with /GL, the linker failed to perform code generation. Investigate and resolve any compiler diagnostics.

Linker Tools Error LNK1258

Error Message

cannot merge 'section1' with 'section2' due to differing section attributes

For help on this issue, see Getting Help from Microsoft Product Support Services (Visual Studio).

Linker Tools Error LNK1260

Error Message
invalid or corrupt file: 'relocation' relocation target symbol 'symbol' must be defined with external linkage

An .obj file is corrupt. Contact Microsoft Product Support Services.

A way to work around this is to change the definition of the symbol to use external linkage.

Linker Tools Error LNK1261

Error Message
multiple FUNCINFO relocations with target symbol 'symbol'

An .obj file is corrupt. Contact Getting Help from Microsoft Product Support Services (Visual Studio).

Two FUNCINFO relocations were emitted targeting the same symbol, which is illegal. This is an error in the tool that generated the relocation.

Linker Tools Error LNK1262

Error Message
link failed due to mismatched versions of 'compiler' and LINK.EXE

Your computer has a mismatched compiler/linker. For example, LINK.EXE and C2.DLL communicate via an interface, and if the interface versions don't match, the linker emits this error.

compiler is the full path and name of the compiler back end that conflicts with the linker.

Linker Tools Error LNK1263

Error Message
/LTCG:PGOPTIMIZE specified but no code generation required; optimization failed

/LTCG:PGOPTIMIZE was specified but no .obj files were found that were compiled with /GL. To resolve this error, provide the identical set of .obj files to the linker that was used when **/LTCG:PGINSTRUMENT** was specified.

Profile guided optimization (PGO) is only available in 64-bit compilers.

Linker Tools Error LNK1264

Error Message
/LTCG:PGINSTRUMENT specified but no code generation required; instrumentation failed

/LTCG:PGINSTRUMENT was specified but no .obj files were found that were compiled with /GL. Instrumentation cannot take place and the link failed. There must be at least one .obj file on the command line that is compiled with **/GL** so that the instrumentation can occur.

Profile guided optimization (PGO) is only available in 64-bit compilers.

Linker Tools Error LNK1265

Error Message
error writing to instrumentation file 'file'; reason

The linker attempted to update information in the instrumentation (PGD) *file*, but failed. The solution depends on the error (*reason*) that the runtime reports.

For more information, see Profile-Guided Optimizations.

Linker Tools Error LNK1266

Error Message
error reading instrumentation file 'file'; reason

The linker was passed /LTCG:PGOPTIMIZE, but the database file could not be read.

Linker Tools Error LNK1267

Error Message
error reading instrumentation file; 'file' not created with compatible version of LINK.EXE

A file, *file*, created with /LTCG:PGINSTRUMENT used a linker version different from the one being used to invoke **/LTCG:PGOPTIMIZE**. The linker versions must be the same.

Linker Tools Error LNK1268

Error Message
inconsistent option 'option' specified with /LTCG:PGOPTIMIZE but not with /LTCG:PGINSTRUMENT

The same linker options used when specifying /LTCG:PGINSTRUMENT must also be used when specifying **/LTCG:PGOPTIMIZE**.

Linker Tools Error LNK1269

Error Message
inconsistent file 'file' specified with /LTCG:PGOPTIMIZE but not with /LTCG:PGINSTRUMENT

The same files passed to the linker when specifying /LTCG:PGINSTRUMENT must also be passed when specifying **/LTCG:PGOPTIMIZE**.

To resolve this error, remove the obj file from the **/LTCG:PGOPTIMIZE** link line.

If *file* is needed to build the image and if you do not have time to build an instrumented image (**/LTCG:PGINSTRUMENT**) and rerun the test scenarios, you can resolve this error by using **/LTCG:PGUPDATE** instead of **/LTCG:PGOPTIMIZE**.

Linker Tools Error LNK1270

Error Message
invalid file; timestamp does not match file linked with /LTCG:PGINSTRUMENT

The same files passed to the linker when specifying /LTCG:PGINSTRUMENT must also be passed when specifying **/LTCG:PGOPTIMIZE**.

/LTCG:PGOPTIMIZE requires that the inputs be identical to the /LTCG:PGINSTRUMENT phase.

To resolve this warning, run /LTCG:PGINSTRUMENT, redo all test runs, and run /LTCG:PGOPTIMIZE, or, create the optimized image with /LTCG:UPDATE instead of /LTCG:PGOPTIMIZE.

Linker Tools Error LNK1272

Error Message
cannot merge 'section' with any section

This error occurs when the /MERGE option is used and one or more of the following sections are specified:

- .idata
- .didat
- .rsrc
- .reloc
- .pdata

Linker Tools Error LNK1274

Error Message
failure during pgomgr

This linker error follows any errors in pgomgr.exe. Resolve the pgomgr errors to resolve this linker error.

Linker Tools Error LNK1276

Error Message
invalid directive 'directive' found; does not start with '/'

When specifying a linker option with the comment pragma, the option must begin with a forward slash.

The following sample generates LNK1276:

```
// LNK1276.cpp
// LNK1276 expected
#pragma data_seg(".MySeg")
int i = 7;
#pragma comment(linker, "section:.MySeg,E")
// try the following line instead
// #pragma comment(linker, "/section:.MySeg,E")

int main() {
}
```

Linker Tools Error LNK1277

Error Message
object record not found in pgd (filename)

When using /LTCG:PGOPTIMZE, the path of one of the input .lib, def, or .obj files was different from the path on which they were found during /LTCG:PGINSTRUMENT. This may be explained by a change in the LIB environment variable after /LTCG:PGINSTRUMENT. The full path to the input files is stored in the .pgd file.

/LTCG:PGOPTIMIZE requires that the inputs be identical to the /LTCG:PGINSTRUMENT phase.

To resolve this warning, do one of the following:

- Run /LTCG:PGINSTRUMENT, redo all test runs, and run /LTCG:PGOPTIMIZE.
- Change the LIB environment variable to what it was when you ran /LTCG:PGINSTRUMENT.

It is not recommended that you work around LNK1277 by using /LTCG:PGUPDATE.

Linker Tools Error LNK1278

Error Message
obj relocation applied to non-BRL instruction; object file is corrupt

The object file *obj* is corrupt; rebuild the object.

If after rebuilding, the file is still corrupt, it implies a tool bug; contact the developer of the tool that generated the object file.

Linker Tools Error LNK1279

Error Message
invalid or corrupt file: file contains invalid .sxdata contributions

One of the input files to the linker had a .sxdata section, which contains exception handler information, but the content of the .sxdata section was invalid.

Try recompiling the module. If it still produces a file with invalid .sxdata section, then contact Microsoft Product Support Services.

Linker Tools Error LNK1280

Error Message
invalid load config structure for SAFESEH image

You supplied an old IMAGE_LOAD_CONFIG_DIRECTORY (_load_config_used) structure and/or without specifying the symbols __safe_se_handler_table or __safe_se_handler_count in the structure.

To resolve this error, use the latest CRT libraries, which contain the new definition, or modify the user-supplied custom load config struct to conform to the new definition.

Linker Tools Error LNK1281

Error Message
Unable to generate SAFESEH image

/SAFESEH was specified and at least one of the modules in the compilation was not compatible for safe exception handlers. The linker could not produce an image with a table of safe exception handlers.

To resolve this error, use a Visual C++ .NET 2003 compiler (or later) to recompile the module or modules that are not compatible with safe exception handlers and relink.

Linker Tools Error LNK1282

Error Message
unable to /REBASE file; it has been signed

You attempted to change the base address of a signed assembly with the /REBASE option for editbin. To do this, first change the base address and then sign the assembly.

Linker Tools Error LNK1283

Error Message
unable to /REBASE file; it has been signed with a strong signature

You attempted to change the base address of a signed assembly with the /REBASE option for editbin. To do this, first change the base address and then sign the assembly.

Linker Tools Error LNK1285

Error Message
corrupt PDB file 'pdb'; delete and rebuild

A .pdb file is corrupt. Delete the .pdb file and rebuild.

Linker Tools Error LNK1286

Error Message
pure MSIL DLL cannot have entry point

When linking a DLL with objects built using /clr:safe, the image cannot have an entry point. Remove /entry, or use /noentry.

Linker Tools Error LNK1287

Error Message
invalid managed entry point function

The entry point is not valid for a managed image. The return type of a managed entry point function can only be **void** or **int**. The type of the parameter of a managed entry point function can only be **void** or **String []**.

Linker Tools Error LNK1290

Error Message
invalid or corrupt file: type MSIL object contains invalid relocations

The linker has detected bad relocation type in the object compiled with **/clr**. For more information, see /clr (Common Language Runtime Compilation).

This error implies a bug in the compiler. Contact Microsoft Product Support Services.

Linker Tools Error LNK1292

Error Message
compiler 'version' does not support this input file format

An /LTCG object was passed to the linker, but the object was not in a format supported by the current compiler.

Recompile your /LTCG objects with a compiler that is matched to the linker.

Linker Tools Error LNK1293

Error Message

Can't get handler from 'entry_point' (reason)

occurs when the linker calls the backend LTCG entry point and it fails with an error not specifically handled by the linker.

Linker Tools Error LNK1291

Error Message
cannot insert .netmodule into a library

An MSIL module is not allowed as input to lib.exe.

An MSIL module can be created with /clr:noAssembly or with /target:module in one of the other Visual Studio compilers.

Linker Tools Error LNK1294

Error Message
exports and import libraries are not supported with machine

This error is caused when you try to generate a DLL and/or exports for machine types (such as CEE and EFI) or subsystems (such as EFI_ROM) that do not support exports.

Linker Tools Error LNK1295

Error Message

'/option1' not compatible with '/option2' specification; link without '/option1'

Two mutually exclusive linker options were specified; */option1* cannot be used with */option2*.

Linker Tools Error LNK1296

Error Message
unable to load filename

The given DLL was unavailable.

The possible locations for the DLL are the current directory, the system directory, the Windows directory, and the directories specified in the PATH environment variable.

Linker Tools Error LNK1299

Error Message
no response file specified with '@'

The linker option to invoke a response file was specified, but no file name was passed to the option.

The following sample generates LNK1299:

```
// LNK1299.cpp
// compile with: /c
int main() {}
```

and then type `link @ lnk1299.obj` on the command line, the linker will generate LNK1299.

Linker Tools Error LNK1300

Error Message

cannot connect to or launch MSPDBSRV.EXE; recheck installation of this product

The version of MSPDBSRV.EXE available on your system does not match the version required by this tool, or is not found.

Reinstall Visual Studio to resolve this error.

Linker Tools Error LNK1301

Error Message
LTCG clr modules found, incompatible with /LTCG:parameter

A module compiled with /clr and /GL was passed to the linker along with one of the profile guided optimizations (PGO) parameters of /LTCG.

Profile guided optimizations are not supported for /clr modules.

For more information, see:

- /GL (Whole Program Optimization)
- /LTCG (Link-time Code Generation)
- /clr (Common Language Runtime Compilation)
- Profile-Guided Optimizations

To correct this error

- Do not compile with /clr or do not link with one of the PGO parameters to /LTCG.

Example
The following sample generates LNK1301:

```
// LNK1301.cpp
// compile with: /clr /GL /link /LTCG:PGI LNK1301.obj
// LNK1301 expected
class MyClass {
public:
   int i;
};
```

Linker Tools Error LNK1302

Error Message
only support linking safe .netmodules; unable to link file .netmodule

A .netmodule (compiled with **/LN**) was passed to the linker in a user attempt to invoke MSIL linking. A C++ module is valid for MSIL linking if it is compiled with **/clr:safe**.

To correct this error, compile with **/clr:safe** to enable MSIL linking, or pass the **/clr** or **/clr:pure** .obj file to the linker instead of the module.

For more information, see

- /LN (Create MSIL Module)
- .netmodule Files as Linker Input

Linker Tools Error LNK1303

Error Message

ALINK operation failed (code) : reason : Unable to import assembly module file

The linker uses al.exe for assembly generation. LNK1303 means that a linker call into al.exe failed. For more information on al.exe diagnostics, see Al.exe Tool Errors and Warnings.

Linker Tools Error LNK1304

Error Message
COFF section name exceeds string table limit

A module was too large.

To correct this error

- Break the large module into smaller modules.

Linker Tools Error LNK1306

Error Message
DLL entry point function cannot be managed; compile to native

Beginning in Visual C++ 2005, DllMain cannot be compiled to MSIL; it must be compiled to native.

To resolve,

- Compile the file that contains the entry point without **/clr**.
- Put the entry point in a **#pragma unmanaged** section.
- For more information, see
- /clr (Common Language Runtime Compilation)
- managed, unmanaged
- Initialization of Mixed Assemblies
- Run-Time Library Behavior

Example
The following sample generates LNK1306.

```
// LNK1306.cpp
// compile with: /clr /link /dll /entry:NewDllMain
// LNK1306 error expected
#include <windows.h>
int __stdcall NewDllMain( HINSTANCE h, ULONG ulReason, PVOID pvReserved ) {
   return 1;
}
```

Linker Tools Error LNK1307

Error Message
cannot import from DLL 'file_name'; unable to convert name to ANSI codepage code_page

The linker was unable to convert a Unicode DLL name to ANSI because one or more characters had no equivalent in the ANSI code page.

To resolve LNK1307, change the DLL name so it is locale independent, for example using 7- or 8-bit ASCII.

Linker Tools Error LNK1308

Error Message
type module detected; unable to infer clr image type; use /CLRIMAGETYPE:type to link

Commonly, the linker will be able to infer the kind of CLR image it should create, based on the CLR types of the input modules. LNK1308 indicates that the linker was not able to do that, and that you must explicitly specify the kind of CLR image that the linker should create.

See /CLRIMAGETYPE (Specify Type of CLR Image) for more information.

Linker Tools Error LNK1309

Error Message
type1 module detected; invalid with switch /CLRIMAGETYPE:type2

A CLR image type was requested with **/CLRIMAGETYPE** but the linker could not produce an image of that type because one or more modules were incompatible with that type.

For example, you will see LNK1309 if you specify **/CLRIMAGETYPE:safe** and you pass a module built with **/clr:pure**.

You will also see LNK1309 if you attempt to build a partially trusted CLR pure application using ptrustu[d].lib. For information on how to create a partially trusted application, see
How to: Create a Partially Trusted Application by Removing Dependency on msvcm80.dll.

For more information, see /clr (Common Language Runtime Compilation) and /CLRIMAGETYPE (Specify Type of CLR Image).

Linker Tools Error LNK1310

Error Message
Exports not supported for type MSIL image

When compiling with **/clr:pure** or **/clr:safe**, you cannot also use the **/EXPORTS** linker option.

For more information, see /clr (Common Language Runtime Compilation) and /EXPORTS.

Linker Tools Error LNK1311

Error Message
number unresolved module references

LNK1311 always follows LNK2027.

Linker Tools Error LNK1312

Error Message
invalid or corrupt file: unable to import assembly

When building an assembly, a file other than a module or assembly compiled with **/clr** was passed to the **/ASSEMBLYMODULE** linker option. If you passed an object file to **/ASSEMBLYMODULE**, just pass the object directly to the linker, instead of to **/ASSEMBLYMODULE**.

Example

The following sample created the .obj file.

```
// LNK1312.cpp
// compile with: /clr /LD
public ref class A {
public:
   int i;
};
```

The following sample generates LNK1312.

```
// LNK1312_b.cpp
// compile with: /clr /LD /link /assemblymodule:LNK1312.obj
// LNK1312 error expected
public ref class M {};
```

Linker Tools Error LNK1313

Error Message
ijw/native module detected; cannot link with pure modules

The current version of Visual C++ does not support linking native or mixed managed/native (also called IJW) .obj files with .obj files compiled with **/clr:pure**.

Example

```
// LNK1313.cpp
// compile with: /c /clr:pure
// a pure module
int main() {}
```

```
// LNK1313_b.cpp
// compile with: /c /clr
// an IJW module
void test(){}
```

The following sample will generate LNK1313.

```
// LNK1313_c.cpp
// compile with: /link LNK1313.obj LNK1313_b.obj
// LNK1313 warning expected
```

Linker Tools Error LNK1314

Error Message
corrupt or invalid COFF symbol table (undefined static or label symbol)

The compiler-generated content for a section in the given object appears to be corrupt.

This error may represent a failure of the compiler; contact
Getting Help from Microsoft Product Support Services (Visual Studio).

Linker Tools Error LNK1315

Error Message
assembly reference to mscorlib.dll not found in any input file

The linker requires at least one input file to contain a reference in metadata to mscorlib.dll. By default, all Visual Studio compilers create modules and assemblies with this reference.

Linker Tools Error LNK1316

Error Message
duplicate managed resource name 'name'

A file was specified in more than one invocation of **/ASSEMBLYRESOURCE** or multiple invocations of **/ASSEMBLYRESOURCE** specified the same name inside the assembly.

A resource name inside an assembly must be unique. Modify your **/ASSEMBLYRESOURCE** calls.

For more information, see /ASSEMBLYRESOURCE (Embed a Managed Resource).

Linker Tools Error LNK1317

Error Message
during code generation, compiler introduced reference to managed symbol 'symbol' defined in module 'module' compiled with /GL; try compiling the function as native with '#pragma unmanaged'

During code generation, the compiler should not introduce symbols that are later resolved to definitions compiled **/GL**. *symbol* is a symbol that was introduced and later resolved to a definition compiled with **/GL**.

For more information, see /GL (Whole Program Optimization).

To resolve LNK1317, do not compile the symbol with **/GL** or use /INCLUDE (Force Symbol References) to force a reference to the symbol. You could also try compiling the function (symbol) with the **unmanaged** pragma. See managed, unmanaged for more information.

Linker Tools Error LNK1561

Error Message
entry point must be defined

The linker did not find an entry point. You may have intended to link as a DLL, in which case you should link with the /DLL option. You may have also forgotten to specify the name of the entry point; link with the /ENTRY option.

Otherwise, you should include a main, wmain, WinMain, or wMain function in your code.

If you are using LIB and intend to build a .dll, one reason for this error is that you supplied a .def file. If so, remove the .def file from the build.

The following sample generates LNK1561:

```
// LNK1561.cpp
// LNK1561 expected
int i;
// add a main function to resolve this error
```

Linker Tools Error LNK2001

Error Message
unresolved external symbol "symbol"

Code references something (such as a function, variable, or label) that the linker can't find in the libraries and object files.

This error message is followed by fatal error LNK1120.

Possible Causes

- When upgrading a managed library or web service project from Visual C++ 2003 to Visual C++ 2005, the **/Zl** compiler option will added to the **Command Line** property page. This will cause LNK2001. Remove **/Zl** from the **Command Line** property page to resolve. See /Zl (Omit Default Library Name) and How to: Open Project Property Pages for more information. Or, add msvcrt.lib and msvcmrt.lib to the linker's Additional Dependencies property.
- What the code asks for doesn't exist (the symbol is spelled incorrectly or uses the wrong case, for example).
- The code asks for the wrong thing (you are using mixed versions of the libraries, some from one version of the product, others from another version).

Specific Causes

Coding Problems

- If the LNK2001 diagnostic text reports that `__check_commonlanguageruntime_version` is an unresolved external symbol, see LNK2019 for information on how to resolve.
- The definition of member template is outside the class. Visual C++ has a limitation in which member templates must be fully defined within the enclosing class. See KB article Q239436 for more information about LNK2001 and member templates.
- Mismatched case in your code or module-definition (.def) file can cause LNK2001. For example, if you named a variable `var1` in one C++ source file and tried to access it as `VAR1` in another.
- A project that uses function inlining yet defines the functions in a .cpp file rather than in the header file can cause LNK2001.
- Calling a C function from a C++ program without using **extern** "C" (which causes the compiler to use the C naming convention) can cause LNK2001. Compiler options /Tp and /Tc cause the compiler to compile files as C++ or C, respectively, regardless of the filename extension. These options can cause function names different from what you expect.
- Attempting to reference functions or data that don't have external linkage can cause LNK2001. In C++, inline functions and **const** data have internal linkage unless explicitly specified as **extern**.
- A missing function body or variable can cause LNK2001. With just a function prototype or **extern** declaration the compiler can continue without error, but the linker cannot resolve a call to an address or reference to a variable because there is no function code or variable space reserved.
- Calling a function with parameter types that do not match those in the function declaration can cause LNK2001. Name decoration incorporates the parameters of a function into the final decorated function name.
- Incorrectly included prototypes, which cause the compiler to expect a function body that is not provided can cause LNK2001. If you have both a class and non-class implementation of a function `F`, beware of C++ scope-resolution rules.
- When using C++, including a function prototype in a class definition and failing to include the implementation of the function for that class can cause LNK2001.
- Attempting to call a pure virtual function from the constructor or destructor of an abstract base class can cause LNK2001. A pure virtual function has no base class implementation.
- Trying to use a variable declared within a function (a local variable) outside the scope of that function can cause LNK2001.
- When building a Release version of an ATL project, indicates that CRT startup code is required. To fix, do one of the

following,

- Remove `_ATL_MIN_CRT` from the list of preprocessor defines to allow CRT startup code to be included. See General Configuration Settings Property Page for more information.
- If possible, remove calls to CRT functions that require CRT startup code. Instead, use their Win32 equivalents. For example, use `lstrcmp` instead of **strcmp**. Known functions that require CRT startup code are some of the string and floating point functions.

Compiling and Linking Problems

- The project is missing a reference to a library (.LIB) or object (.OBJ) file. See .lib Files as Linker Input for more information.
- If you use /NODEFAULTLIB or /Zl, libraries containing required code will not be linked into the project unless you have explicitly included them. (When compiling with **/clr** or **/clr:pure**, you will see a reference to .cctor; see Initialization of Mixed Assemblies for more information.)
- If you are using Unicode and MFC, you will get an unresolved external on `_WinMain@16` if you don't create an entrypoint to `wWinMainCRTStartup`; use the /ENTRY. See Unicode Programming Summary.

  See the following Knowledge Base articles, located in the MSDN Library, for more information. In the MSDN Library, click the **Search** tab, paste the article number or article title into the text box, and then click **List Topics**. If you search on the article number, make sure the **Search titles only** option is clear.

  - Q125750 "PRB: Error LNK2001: '_WinMain@16': Unresolved External Symbol"
  - Q131204 "PRB: Wrong Project Selection Causes LNK2001 on _WinMain@16"
  - Q100639 "Unicode Support in the Microsoft Foundation Class Library"
  - Q291952 "PRB: Link Error LNK2001: Unresolved External Symbol _main"
- Linking code compiled with /MT with the library LIBC.lib causes LNK2001 on **_beginthread**, **_beginthreadex**, **_endthread**, and **_endthreadex**.
- Linking code requiring the multithreaded libraries (any MFC code or code compiled with /MT) causes LNK2001 on _beginthread, **_beginthreadex**, _endthread, and **_endthreadex**. See the following Knowledge Base article for more information:

  - Q126646 "PRB: Error Msg: LNK2001 on __beginthreadex and __endthreadex"
  - Q128641 "INFO: /Mx Compiler Options and the LIBC, LIBCMT, MSVCRT Libs"
  - Q166504 "PRB: MFC and CRT Must Match in debug/release and static/dynamic"
- When compiling with **/MD**, a reference to "func" in your source becomes a reference "`__imp__func`" in the object since all the run-time is now held within a DLL. If you try to link with the static libraries LIBC.lib or LIBCMT.lib, you will get LNK2001 on `__imp__func`. If you try to link with MSVCxx.lib when compiling without /MD you will not always get LNK2001, but you will likely have other problems.
- Linking with the release mode libraries when building a debug version of an application can cause LNK2001. Similarly, using an **/Mxd** option (**/MTd**, or **/MDd**) and/or defining `_DEBUG` and then linking with the release libraries will give you potential unresolved externals (among other problems). Linking a release mode build with the debug libraries will also cause similar problems.
- Mixing versions of Microsoft libraries and compiler products can be problematic. A new compiler version's libraries may contain new symbols that cannot be found in the libraries included with previous versions. You may want to change the order of the directories in the search path, or changing them to point to the current version.

  The Tools | Options | Projects | VC++ Directories dialog, under the Library files selection, allows you to change the search order. The Linker folder in the project's Property Pages dialog box may also contain paths that could be out of date.

  This problem may appear when a new SDK is installed (perhaps to a different location), and the search order is not updated to point to the new location. Normally, you should put the path to new SDKs' include and lib directories in front of the default Visual C++ location. Also, a project containing embedded paths may still point to old paths that are valid, but out of date for new functionality added by the new version that is installed to a different location.
- There is currently no standard for C++ naming between compiler vendors or even between different versions of a

compiler. Therefore, linking object files compiled with other compilers may not produce the same naming scheme and thus cause error LNK2001.

- Mixing inline and non-inline compile options on different modules can cause LNK2001. If a C++ library is created with function inlining turned on (**/Ob1** or **/Ob2**) but the corresponding header file describing the functions has inlining turned off (no **inline** keyword), you will get this error. To prevent this problem, have the inline functions defined with **inline** in the header file you are going to include in other files.
- If you are using the **#pragma inline_depth** compiler directive, make sure you have a value of 2 or greater set, and make sure you are using the /Ob1 or /Ob2 compiler option.
- Omitting the LINK option /NOENTRY when creating a resource-only DLL will cause LNK2001.
- Using incorrect /SUBSYSTEM or /ENTRY settings can cause LNK2001. For example, if you write a character-based application (a console application) and specify /SUBSYSTEM:WINDOWS, you will get an unresolved external for **WinMain**. For more information on these options and entry points, see the /SUBSYSTEM and /ENTRY linker options.

Export Problems

- When you are porting an application from 16 to 32 or 64 bits, LNK2001 can occur. The current module-definition (.def) file syntax requires that **__cdecl**, **__stdcall**, and **__fastcall** functions be listed in the EXPORTS section without underscores (undecorated). This differs from the 16-bit syntax, where they must be listed with underscores (decorated). For more information, see the description of the EXPORTS section of module-definition files.
- Any export listed in the .def file and not found will cause LNK2001. This could be because it does not exist, is spelled incorrectly, or uses C++ decorated names (.def files do not take decorated names)

Interpreting the Output

When a symbol is unresolved, you can get information about the function by the following guidelines:

On x86 platforms, the calling convention decoration for names compiled in C, or for extern "C" names in C++, is:

__cdecl

Function has an underscore (_) prefix.

__stdcall

Function has an underscore (_) prefix and an @ suffix followed by the dword aligned size of parameters on the stack.

__fastcall

Function has an @ prefix and an @ suffix followed by the dword aligned size of parameters on the stack.

Use undname.exe to get the undecorated form of a decorated name.

For more information on some of the causes listed above, see Name decoration.

Name Decoration

Name decoration usually refers to C++ naming conventions, but can apply to a number of C cases as well. By default, C++ uses the function name, parameters, and return type to create a linker name for the function. Consider the following function:

```
void CALLTYPE test(void)
```

The following table shows the linker name for various calling conventions.

Calling convention	extern "C" or .c file	.cpp, .cxx or /TP
C naming convention (**__cdecl**)	_test	?test@@ZAXXZ
Fastcall naming convention (**__fastcall**)	@test@0	?test@@YIXXZ
Standard Call naming convention (**__stdcall**)	_test@0	?test@@YGXXZ

Use extern "C" to call a C function from C++. Extern "C" forces use of the C naming convention for non-class C++ functions. Be aware of compiler switches **/Tc** or **/TP**, which tell the compiler to ignore the filename extension and compile the file as C or C++, respectively. These options may cause names you do not expect.

Having function prototypes that have mismatched parameters can also cause this error. Name decoration incorporates the parameters of a function into the final decorated function name. Calling a function with the parameter types that do not match those in the function declaration may also cause LNK2001.

There is currently no standard for C++ naming between compiler vendors or even between different versions of a compiler. Therefore linking object files compiled with other compilers may not produce the same naming scheme and thus causes unresolved externals.

See Also

Reference

Linker Tools Error LNK2001

Linker Tools Error LNK2003

Error Message

gp relative fixup to symbol not in .sdata 'module'

The compiler emitted an ill-formed object; contact Microsoft Product Support Services.

This error is followed by fatal error LNK1165.

Linker Tools Error LNK2004

Error Message
gp relative fixup overflow to 'target'; short section 'section' is too large or out of range.

The section was too large.

To resolve this error, reduce the size of the short section, either by explicitly putting data in the long sections via #pragma section(".sectionname", read, write, long) and using `__declspec(allocate(".sectionname"))` on data definitions and declarations. For example,

```
#pragma section(".data$mylong", read, write, long)
__declspec(allocate(".data$mylong"))
char    rg0[1] = { 1 };
char    rg1[2] = { 1 };
char    rg2[4] = { 1 };
char    rg3[8] = { 1 };
char    rg4[16] = { 1 };
char    rg5[32] = { 1 };
```

You can also move logically-grouped data into its own structure that will be a collection of data greater than 8 bytes, which the compiler will allocate in a long data section. For example,

```
// from this...
int     w1  = 23;
int     w2 = 46;
int     w3 = 23*3;
int     w4 = 23*4;

// to this...
struct X {
    int     w1;
    int     w2;
    int     w3;
    int     w4;
} x  = { 23, 23*2, 23*3, 23*4 };
```

This error is followed by fatal error LNK1165.

Linker Tools Error LNK2005

Error Message
symbol already defined in object

The given *symbol*, displayed in its decorated form, was multiply defined.

For more information, see the Knowledge Base articles:

- "LNK2005 Errors When Link C Run-Time Libraries Are Linked Before MFC Libraries" (Q148652)
- "Global Overloaded Delete Operator Causes LNK2005" (Q140440)
- "LNK2005 Errors on New and Delete When Defining _ATL_MIN_CRT" (Q184235).

You can find Knowledge Base articles on the MSDN Library CD-ROM or at http://support.microsoft.com/support.

This error is followed by fatal error LNK1169.

To fix by checking the following possible causes

1. Mixing static and dynamic libraries when also using /clr.
2. The symbol is a packaged function (created by compiling with /Gy) and was included in more than one file but was changed between compilations. Recompile all files that include *symbol*.
3. The symbol is defined differently in two member objects in different libraries, and both member objects were used.
4. An absolute is defined twice, with a different value in each definition.
5. A header file declared and defined a variable. Possible solutions include:
   - Declare the variable in .h: `extern BOOL MyBool;` and then assign to it in a .c or .cpp file: `BOOL MyBool = FALSE;`.
   - Declare the variable static.
   - Declare the variable selectany.
6. If you use uuid.lib in combination with other .lib files that define GUIDs (for example, oledb.lib and adsiid.lib). For example:

   ```
   oledb.lib(oledb_i.obj) : error LNK2005: _IID_ITransactionObject
   already defined in uuid.lib(go7.obj)
   ```

   To fix, add /FORCE:MULTIPLE to the linker command line options, and make sure that uuid.lib is the first library referenced.

Linker Tools Error LNK2008

Error Message
Fixup target is not aligned 'symbol_name'

LINK found a fixup target in your object file that was not aligned properly.

This error can be caused by custom secton alignment (for example, #pragma pack), align modifier, or by using assembly language code that modifies secton alignment.

If your code does not use any of the above, this may be caused by the compiler. Note the circumstances of the error, try to isolate the problem and create a reproducible test case, then contact Microsoft Product Support Services (if you are using Visual C++ compiler).

Linker Tools Error LNK2009

Error Message

Fixup target must be absolute 'name' w/o -FIXED; fixup ignored

LINK found an invalid fixup and ignored it.

Linker Tools Error LNK2011

Error Message
precompiled object not linked in; image may not run

If you use precompiled headers, LINK requires that all of the object files created with precompiled headers must be linked in. If you have a source file that you use to generate a precompiled header for use with other source files, you now must include the object file created along with the precompiled header.

For example, if you compile a file called STUB.cpp to create a precompiled header for use with other source files, you must link with STUB.obj or you will get this error. In the following command lines, line one is used to create a precompiled header, COMMON.pch, which is used with PROG1.cpp and PROG2.cpp in lines two and three. The file STUB.cpp contains only **#include** lines (the same **#include** lines as in PROG1.cpp and PROG2.cpp) and is used only to generate precompiled headers. In the last line, STUB.obj must be linked in to avoid LNK2011.

```
cl /c /Yccommon.h stub.cpp
cl /c /Yucommon.h prog1.cpp
cl /c /Yucommon.h prog2.cpp
link /out:prog.exe stub.obj prog1.obj prog2.obj
```

Linker Tools Error LNK2013

Error Message
fixup type fixup overflow. Target 'symbol name' is out of range

The linker cannot fit the necessary address or offset into the given instruction because the target symbol is too far away from the instruction's location.

You can resolve this problem by creating multiple images or by using the /ORDER option so the instruction and target are closer together.

You may also get this error when using the Itanium Architecture compiler and when *symbol name* is a static function. In this case, you can use /incremental:NO to resolve the error.

When the symbol name is a user-defined symbol (and not a compiler-generated symbol), you can also try the following actions to resolve the error:

- Change the static function to be non-static.
- Rename the code section containing the static function to be the same as the caller.

Use `DUMPBIN /SYMBOLS`, to see if a function is static.

Linker Tools Error LNK2014

Error Message
TLS relative fixup overflow; .tls section ('section') is too large

The section created for thread-local storage (.tls) is too large. TLS data must fit in 32 KB. This data is created using the **__declspec(thread)** storage class modifier in the declaration and definition of that data.

To fix by using the following possible solutions

1. Reduce the amount of thread-local data in the code.
2. Use dynamic TLS by calling functions such as TlsAlloc and TlsFree.

Linker Tools Error LNK2016

Error Message

Absolute symbol 'symbol' used as target of target relocation

An absolute symbol is not a legal target of a relocation. You may want to contact your compiler vendor.

Linker Tools Error LNK2017

Error Message
'symbol' relocation to 'segment' invalid without /LARGEADDRESSAWARE:NO

You are trying to build a 64-bit image with 32-bit addresses. To do this, you must:

- Use a fixed load address.
- Restrict the image to 3 GB.
- Specify /largeaddressaware:no.

Linker Tools Error LNK2019

Error Message
unresolved external symbol 'symbol' referenced in function 'function'

An undefined external symbol (*symbol*) was found in *function*. To resolve this error, provide a definition for symbol or remove the code that references it. For more information, see,

- Function inlining problems
- Automatic (function scope) variables
- Missing function body or variable

The following sample generates LNK2019:

```
// LNK2019.cpp
// LNK2019 expected
extern char B[100];   // B is not in avilable to the linker
int main() {
   B[0] = ' ';
}
```

LNK2019 can also occur when you declare but do not define a static data member. The following sample generates LNK2019:

```
// LNK2019b.cpp
// LNK2019 expected
struct C {
   static int s;
};

// Uncomment the following line to resolve.
// int C::s;

int main() {
   C c;
   C::s = 1;
}
```

Consider the following sample:

```
// LNK2019c.cpp
// LNK2019 expected
extern int i;
extern void g();
void f() {
   i++;
   g();
}
int main() {}
```

If `i` and `g` are not defined in one of the files included in the build, the linker will generate LNK2019. These definitions can be added by including the source code file that contains the definitions as part of the compilation. Alternatively, you can pass .obj or .lib files that contain the definitions to the linker.

For C++ projects from previous releases that were upgraded to the current version, if __UNICODE was defined and the entry point was WinMain, you need to change the name of the entry point function to either _tWinMain or _tmain.

Common problems that cause LNK2019 include:

- The declaration of the symbol contains a spelling mistake, such that, it is not the same name as the definition of the symbol.

- A function was used but the type or number of the parameters did not match the function definition.
- The calling convention (__cdecl, __stdcall, or __fastcall) differs on the use of the function declaration and the function definition.
- Symbol definitions are in a file that was compiled as a C program and symbols are declared in a C++ file without an extern "C" modifier. In that case, modify the declaration, for example, instead of:

  ```
  extern int i;
  extern void g();
  ```

  use:

  ```
  extern "C" int i;
  extern "C" void g();
  ```

  Similarly, if you define a symbol in a C++ file that will be used by a C program, use `extern "C"` in the definition.
- A symbol is defined as static and then later referenced outside the file. In C++, unlike C, global constants have **static** linkage. To get around this limitation, you can include the **const** initializations in a header file and include that header in your .cpp files, or you can make the variable non-constant and use a constant reference to access it.
- A static member of a class is not defined. For example, member variable `si` in the class declaration below should be defined separately:

  ```
  // LNK2019d.cpp
  #include <stdio.h>
  struct X {
     static int si;
  };

  // int X::si = 0;   // uncomment this line to resolve

  int main() {
     X *px = new X[2];
     printf_s("\n%d",px[0].si);   // LNK2019
  }
  ```

The following sample generates LNK2019 on a user-defined operator.

```
// LNK2019e.cpp
// compile with: /EHsc
// LNK2019 expected
#include <iostream>
using namespace std;

template<class T> class
Test {
   friend ostream& operator<<(ostream&, Test&);
   // Uncomment the following line to resolve.
   // template<typename T> friend ostream& operator << (ostream&, Test<T>&);
};

template<typename T>
ostream& operator<<(ostream& os, Test<T>& tt) {
   return os;
}

int main() {
   Test<int> t;
   cout << "Test: " << t << endl;   // unresolved external
}
```

The /VERBOSE linker option will help you see which files the linker is referencing. The /EXPORTS and /SYMBOLS options of the DUMPBIN utility can also help you see which symbols are defined in your dll and object/library files.

The Microsoft Knowledge Base has more information about LNK2019; see http://support.microsoft.com.

LNK2019 can also be generated as a result of conformance work that was done for Visual Studio .NET 2003: template friends and specialization. In Visual Studio .NET 2003, a declaration of a friend function with the same name as a function template does not refer to that function template unless template arguments are explicitly specified in the friend declaration.

If you do not specify template arguments, the friend declaration declares a non-template function.

See Summary of Compile-Time Breaking Changes for more information.

For code that is valid in both the Visual Studio .NET 2003 and Visual Studio .NET versions of Visual C++, explicitly specify the friend function's template argument list.

```
// LNK2019f.cpp
// LNK2019 expected
template<class T>
void f(T) {}

template<class T>
struct S {
   friend void f(T);
   // try the folowing line instead
   // friend void f<T>(T);
};

int main() {
   S<int> s;
   f(1);   // unresolved external
}
```

LNK2019 can also occur as a result of conformance work that was done in Visual C++ 2005, **/Zc:wchar_t** is now on by default. All modules may not have been compiled with the same **/Zc:wchar_t** settings, causing type references to not resolve to compatible types. To resolve, ensure that types in all modules are compatible, either by compiling with the appropriate **/Zc:wchar_t** settings (for example, using **/Zc:wchar_t-** when building modules with the Visual C++ 2005 toolset that will be linked with modules from previous versions) or, if possible, by updating your types so they are compatible.

Explicit references to comsupp.lib, either from the comment pragma or via the command line, should be changed to now use either comsuppw.lib or comsuppwd.lib, as **/Zc:wchar_t** is now on by default. comsupp.lib should still be used when compiling with **/Zc:wchar_t-**.

For more information, see Breaking Changes in the Visual C++ 2005 Compiler and /Zc:wchar_t (wchar_t Is Native Type).

The following sample creates an export that uses **WCHAR**, which resolves to **wchar_t**.

```
// LNK2019g.cpp
// compile with: /LD
#include "windows.h"
// WCHAR resolves to wchar_t
__declspec(dllexport) void func(WCHAR*) {}
```

The following sample generates LNK2019:

```
// LNK2019h.cpp
// compile with: LNK2019g.lib
// LNK2019 expected
__declspec(dllimport) void func(unsigned short*);

int main() {
   func(0);
}
```

To resolve this error, either change unsigned short to wchar_t or WCHAR, or compile LNK2019g.cpp with **/Zc:wchar_t-**.

You can also get LNK2019 if you are building a console application with **/SUBSYSTEM:WINDOWS**. The unresolved symbol will be _WinMain@16. In this case, just link with **/SUBSYSTEM:CONSOLE**.

Global Constants in C++

C++ global constants have static linkage. This is different than C. If you try to use a global constant in C++ in multiple files you get an unresolved external error. The compiler optimizes global constants out, leaving no space reserved for the variable.

One way to resolve this error is to include the const initializations in a header file and include that header in your CPP files when necessary, just as if it was function prototype. Another possibility is to make the variable non-constant and use a constant reference when assessing it.

The following sample generates C2019:

```
// global_constants.cpp
// LNK2019 expected
void test(void);
const int lnktest1 = 0;

int main() {
   test();
}
```

And then,

```
// global_constants_2.cpp
// compile with: global_constants.cpp
extern int lnktest1;

void test() {
  int i = lnktest1;   // LNK2019
}
```

See Also

Reference

Linker Tools Error LNK2019

Function Inlining Problems

If you are using function inlining, you must:

- Have the inline functions implemented in the header file you include.
- Have inlining turned ON in the header file.

```
// LNK2019_function_inline.cpp
// compile with: /c
// post-build command: lib LNK2019_function_inline.obj
#include <stdio.h>
struct _load_config_used {
   void Test();
   void Test2() { printf("in Test2\n"); }
};

void _load_config_used::Test() { printf("in Test\n"); }
```

And then,

```
// LNK2019_function_inline_2.cpp
// compile with: LNK2019_function_inline.lib
struct _load_config_used {
   void Test();
   void Test2();
};

int main() {
   _load_config_used x;
   x.Test();
   x.Test2();   // LNK2019
}
```

If you are using the **#pragma inline_depth** compiler directive, make sure you have a value of 2 or greater set. A value of zero will turn off inlining. Also make sure you are using the **/Ob1** or **/Ob2** compiler options.

Mixing inline and non-inline compile options on different modules can sometimes cause problems. If a C++ library is created with function inlining turned on (/Ob1 or /Ob2) but the corresponding header file describing the functions has inlining turned off (no option), you will get error LNK2001. The functions do not get inlined into the code from the header file, but since they are not in the library file there is no address to resolve the reference.

Similarly, a project that uses function inlining yet defines the functions in a .cpp file rather than in the header file will also get LNK2019. The header file is included everywhere deemed appropriate, but the functions are only inlined when the .cpp file passes through the compiler; therefore, the linker sees the functions as unresolved externals when used in other modules.

```
// LNK2019_FIP.h
struct testclass {
   void PublicStatMemFunc1(void);
};
```

and then,

```
// LNK2019_FIP.cpp
// compile with: /c
#include "LNK2019_FIP.h"
inline void testclass::PublicStatMemFunc1(void) {}
```

and then,

```
// LNK2019_FIP_2.cpp
```

```
// compile with: LNK2019_FIP.cpp
// LNK2019 expected
#include "LNK2019_FIP.h"
int main() {
   testclass testclsObject;

   // module cannot see the implementation of PublicStatMemFunc1
   testclsObject.PublicStatMemFunc1();
}
```

See Also

Reference

Linker Tools Error LNK2019

Automatic (Function Scope) Variables

A variable declared within a function can only be used within the scope of that function.

```
// LNK2019_AV.cpp
// compile with: /c
void test(void);

static int lnktest3 = 3;
int lnktest4 = 4;

int main() {
   static int lnktest1 = 1;
   int lnktest2 = 2;
   test();
}
```

and then,

```
// LNK2019_AV_2.cpp
// compile with: LNK2019_AV.cpp
// LNK2019 expected
extern int lnktest1;
extern int lnktest2;
extern int lnktest3;
extern int lnktest4;

void test(void) {
   int i = 0;
   i = lnktest1;
   i = lnktest2;
   i = lnktest3;
   i = lnktest4;   // OK
}
```

See Also

Reference

Linker Tools Error LNK2019

Missing Function Body or Variable

With just a function prototype, the compiler can continue without error, but the linker cannot resolve a call to an address because there is no function code or variable space reserved. You will not see this error until you create a call to the function that the linker must resolve.

Example

The function call in main will cause LNK2019 because the prototype allows the compiler to think the function exists. The linker finds that it doesn't.

```
// LNK2019_MFBV.cpp
// LNK2019 expected
void DoSomething(void);
int main() {
   DoSomething();
}
```

In C++, make sure that you include the implementation of a specific function for a class and not just a prototype in the class definition. If you are defining the class outside of the header file, be sure to include the class name before the function (*Classname*::*memberfunction*).

```
// LNK2019_MFBV_2.cpp
// LNK2019 expected
struct A {
   static void Test();
};

// Should be void A::Test() {}
void Test() {}

int main() {
   A AObject;
   AObject.Test();
}
```

See Also

Reference

Linker Tools Error LNK2019

Linker Tools Error LNK2020

Error Message
unresolved token 'token'

Similar to an undefined external error, except that the reference is via metadata. In metadata, all functions and data must be defined.

To resolve:

- Define the missing function or data, or
- Include the object file or library in which the missing function or data is already defined.

Example

The following sample generates LNK2020.

```
// LNK2020.cpp
// compile with: /clr /LD
ref struct A {
   A(int x);   // LNK2020
   static int f();   // LNK2020
};

// OK
ref struct B {
   B(int x) {}
   static int f() { return 0; }
};
```

LNK2020 will also occur if you create a variable of a managed template type, but do not also instantiate the type.

The following sample generates LNK2020.

```
// LNK2020_b.cpp
// compile with: /clr

template <typename T>
ref struct Base {
   virtual void f1() {};
};

template <typename T>
ref struct Base2 {
   virtual void f1() {};
};

int main() {
   Base<int>^ p;   // LNK2020
   Base2<int>^ p2 = gcnew Base2<int>();   // OK
};
```

Linker Tools Error LNK2021

Error Message
Invalid .pdata fixup detected ('invalid fixup type'); must be type 'valid fixup types'

An invalid fixup type was specified for pointing into the .pdata section.

This error may represent a failure of the compiler; contact Microsoft Product Support Services.

Linker Tools Error LNK2022

Error Message
metadata operation failed (HRESULT) : error_message

The linker detected an error while merging metadata. The metadata errors must be resolved to link successfully.

One way to diagnose this problem is to run **ildasm –tokens** on the object files to find which types have the tokens listed in *error_message*, and look for differences. In metadata, two different types with the same name is not valid, even if the just LayoutType attribute is different.

One reason for LNK2022 is when a type (such as a struct) exists in multiple compilands with the same name, but with conflicting definitions, and when you compile with /clr. In this case, make sure that the type has an identical definition in all compilands. The type name is listed in *error_message*.

Another possible cause for LNK2022 is when the linker finds a metadata file in a different location than was specified to the compiler (with #using). Make sure that the metadata file (.dll or .netmodule) is in the same location when passed to the linker, as it was when it was passed to the compiler.

When building an ATL application, the use of _ATL_MIXED is required in all compilands, if it is used in at least one.

Example

The following sample defines an empty type.

```
// LNK2022_a.cpp
// compile with: /clr /c
public ref class Test {};
```

This sample shows that you cannot link two source code files that contain types of the same name but different definitions.

The following sample generates LNK2022.

```
// LNK2022_b.cpp
// compile with: LNK2022_a.cpp /clr /LD
// LNK2022 expected
public ref class Test {
   void func() {}
   int var;
};
```

Linker Tools Error LNK2023

Error Message
bad dll or entry point <dll or entry point>

The linker is loading an incorrect version of msobj80.dll. Ensure that link.exe and msobj80.dll in your path have the same version.

A dependency of msobj80.dll may not be present. The dependency list for msobj80.dll is:

- Msvcr80.dll
- Kernel32.dll

Check your machine for any other copies of msobj80.dll that may be out of date.

Linker Tools Error LNK2024

Error Message
Invalid fixup to .pdata

This linker error implies a bug in the compiler that created the object. Contact the vendor whose compiler created the object with the invalid fixup.

If the compiler is Visual C++, contact Microsoft Product Support Services.

Linker Tools Error LNK2026

Error Message
module unsafe for SAFESEH image

/SAFESEH was specified, but a module was not compatible with the safe exception handling feature. If you want to use this module with **/SAFESEH**, then you will need to recompile the module using the Visual C++ .NET 2003 (or later) compiler.

Linker Tools Error LNK2027

Error Message
unresolved module reference 'module'

A file passed to the linker has a dependency on a module that was neither specified with **/ASSEMBLYMODULE** nor passed directly to the linker.

To resolve LNK2027, do one of the following:

- Do not pass to the linker the file that has the module dependency.
- Specify the module with **/ASSEMBLYMODULE**.
- If the module is a safe .netmodule, pass the module directly to the linker.

For more information, see /ASSEMBLYMODULE (Add a MSIL Module to the Assembly) and .netmodule Files as Linker Input.

Linker Tools Error LNK2028

Error Message
"exported_function" (decorated_name) referenced in function "function_containing_function_call" (decorated_name)

When attempting to import a native function into a pure image, remember that the implicit calling conventions differ between native and pure compilations.

Example

This code sample generates a component with an exported, native, function whose calling convention is implicitly __cdecl.

```
// LNK2028.cpp
// compile with: /LD
__declspec(dllexport) int func() {
   return 3;
}
```

The following sample creates a pure client that consumes the native function. However, the calling convention under **/clr:pure** is __clrcall. The following sample generates LNK2028.

```
// LNK2028_b.cpp
// compile with: /clr:pure lnk2028.lib
// LNK2028 expected
int func();

int main() {
   return func();
}
```

Linker Tools Error LNK2029

Error Message
metadata for symbol 'symbol' inconsistent with COFF symbol table

The compiler did not generate correct code; contact Microsoft Product Support Services.

Linker Tools Error LNK2030

Error Message
metadata inconsistent with COFF symbol table: symbol 'symbol' (value) is different from 'symbol' in metadata

The compiler did not generate correct code; contact Microsoft Product Support Services.

Linker Tools Error LNK2031

Error Message
unable to generate p/invoke for "function_declaration" decorated_name; calling convention missing in metadata

When attempting to import a native function into a pure image, remember that the implicit calling conventions differ between native and pure compilations.

Example

This code sample generates a component with an exported, native, function whose calling convention is implicitly __cdecl.

```
// LNK2031.cpp
// compile with: /LD
extern "C" {
   __declspec(dllexport) int func() { return 3; }
};
```

The following sample creates a pure client that consumes the native function. However, the calling convention under **/clr:pure** is __clrcall. The following sample generates LNK2031.

```
// LNK2031_b.cpp
// compile with: /clr:pure LNK2031.lib
// LNK2031 expected
extern "C" int func();

int main() {
   return func();
}
```

The following sample shows how to consume the native function from a pure image. Note the explicit **__cdecl** calling convention specifier.

```
// LNK2031_c.cpp
// compile with: /clr:pure LNK2031.lib
extern "C" int __cdecl func();

int main() {
   return func();
}
```

Linker Tools Error LNK2032

Error Message
metadata inconsistent with COFF symbol table: symbol 'symbol' (token) mapped to 'symbol' (token) in module

The linker found symbols with the same name but different definitions.

This error may also be caused by a compiler error. In that case, note the circumstances of the error, try to isolate the problem and create a reproducible test case, and then contact Microsoft Product Support Services.

Linker Tools Error LNK2033

Error Message
unresolved typeref token (token) for 'type'

A type doesn't have a definition in MSIL metadata.

LNK2033 can occur when compiling with **/clr:safe** and where there is only a forward declaration for a type in an MSIL module, where the type is referenced in the MSIL module.

The type needs to be defined under **/clr:safe**.

For more information, see /clr (Common Language Runtime Compilation).

Example
The following sample generates LNK2033.

```
// LNK2033.cpp
// compile with: /clr:safe
// LNK2033 expected
ref class A;
ref class B {};

int main() {
   A ^ aa = nullptr;
   B ^ bb = nullptr;   // OK
};
```

Linker Tools Error LNK2034

Error Message
unresolved typeref token (token) for 'symbol'

The linker found an unresolved token for the indicated symbol.

This error may also be caused by a compiler error. In that case, note the circumstances of the error, try to isolate the problem and create a reproducible test case, and then contact Microsoft Product Support Services.

Linker Tools Error LNK2035

Error Message
metadata inconsistent with COFF symbol table: symbol 'symbol' (token) has inconsistent metadata with (token) in module

The linker found symbols with the same name but different definitions.

This error may also be caused by a compiler error. In that case, note the circumstances of the error, try to isolate the problem and create a reproducible test case, and then contact Microsoft Product Support Services.

Linker Tools Warning LNK4001

Error Message
no object files specified; libraries used

The linker was passed one or more .lib files, but no .obj files.

Because the linker is not able to access information in a .lib file that it is able to access in an .obj file, this warning indicates that you will have to explicitly specify other linker options. For example, you may have to specify the /MACHINE, /OUT, or /ENTRY options.

Linker Tools Warning LNK4002

Error Message
symbol defined in object

The symbol, displayed in its decorated form, was specified in its undecorated form in *object*, but a unique match to a decorated symbol could not be found. This warning is always preceded by warning LNK4022 and followed by fatal error LNK1152.

Linker Tools Warning LNK4003

Error Message
invalid library format; library ignored

The file header for the input library was invalid. The tool did not use the library.

Linker Tools Warning LNK4006

Error Message
symbol already defined in object; second definition ignored

The given *symbol*, displayed in its decorated form, was multiply defined. When this warning is encountered, *symbol* will be added twice, but only its first form will be used.

You can get this warning if you try to merge two import libs into one.

If you are rebuilding the C run-time library, you can ignore this message.

To fix by using the following possible solutions

1. The given *symbol* may be a packaged function, created by compiling with /Gy. This symbol was included in more than one file but was changed between compilations. Recompile all files that include the *symbol*.
2. The given *symbol* may have been defined differently in two member objects in different libraries.
3. An absolute may have been defined twice, with a different value in each definition.
4. If the error message is received when combining libraries, *symbol* already exists in the library being added to.

Linker Tools Warning LNK4010

Error Message
invalid subsystem version number number; default subsystem version assumed

You can specify a version for the image's subsystem (/SUBSYSTEM). Each subsystem has a minimum version requirement. If the specified version is lower than the minimum, this warning will occur and the linker will just use the default subsystem.

Linker Tools Warning LNK4012

Error Message
invalid value 'argument', must be 'keyword'; option ignored

The given argument was not a valid keyword for the option it was specified with. The valid arguments are shown in the given list of keywords.

Linker Tools Warning LNK4013

Error Message
image size imagesize exceeds specified maximum maxsize

The base address file specified with the /BASE option listed the maximum size of the program as *maxsize*, but *maxsize* was less than the resulting image size of the program.

Linker Tools Warning LNK4014

Error Message
cannot find member object "objectname"

LIB could not find *objectname* in the library.

The **/REMOVE** and **/EXTRACT** options require the full name of the member object that is to be deleted or copied to a file. The full name includes the path of the original object file. To see the full names of member objects in a library, use DUMPBIN /ARCHIVEMEMBERS or LIB /LIST.

Linker Tools Warning LNK4017

Error Message
keyword statement not supported for the target platform; ignored

The given module-definition statement is not supported by this version of the linker tools. The tool ignored the statement in the .def file.

Linker Tools Warning LNK4018

Error Message
too many type indexes in PDB 'filename', discarding subsequent type information

There are more than 64K type indexes in the program database (PDB). The PDB might contain unneeded type information. Delete the existing PDB and rebuild. If the warning persists, then compile some modules without debugging information.

See the discussion of MinTypeInfo for a possible way to avoid this warning

Linker Tools Warning LNK4019

Error Message
corrupt string table (table end); new end assumed

The object file is corrupt. The tool attempted to correct the problem.

To prevent this warning, rebuild the object file.

See Corrupt Object File for more information.

Linker Tools Warning LNK4020

Error Message
a type record in 'file' is corrupted; some symbols and types may not be accessible from the debugger

The Visual C++ compiler may have produced debugging information that is incorrect, and the resulting debugging information for the executable may be incomplete or incorrect. Try recompiling the .pdb application with the Visual C++ 2003 (or later) compiler.

If the problem persists, contact Microsoft Product Support Services.

Linker Tools Warning LNK4022

Error Message
cannot find unique match for symbol 'symbol'

LINK or LIB found multiple matches for the given undecorated symbol and it could not resolve the ambiguity. No output file (.exe, .dll, .exp, or .lib) is produced. This warning is followed by one warning LNK4006 for each duplicate symbol and is finally followed by fatal error LNK1152.

To prevent this warning, specify the symbol in its decorated form. Run DUMPBIN on the object to see decorated names.

Linker Tools Warning LNK4030

Error Message
invalid file offset number for category; category not dumped

DUMPBIN could not display information for the given category.

This error can be caused if the file is invalid or corrupt.

For more information, see Corrupt Object File.

Linker Tools Warning LNK4031

Error Message
no subsystem specified; CONSOLE assumed

The /SUBSYSTEM option was not specified and LINK was not able to determine a subsystem from the input files.

Linker Tools Warning LNK4037

Error Message
'symbol' does not exist; ignored

The *symbol* could not be ordered using the /ORDER option because it could not be found in the program. Check the specification of *symbol* in the ordering file.

Note
LINK cannot order static functions because static function names are not public symbol names. When **/ORDER** is specified, th is linker warning is generated for each symbol, in the order file, that is either static or not found.

Linker Tools Warning LNK4038

Error Message
displaying raw data in number units per line

The /RAWDATA option was specified without a formatting argument. DUMPBIN assumed a format value of *number* units per line of the specified or assumed data type, as follows:

Data Type	***number***
none	16
BYTES	16
SHORTS	8
LONGS	4

Linker Tools Warning LNK4039

Error Message

section 'name' specified with /SECTION option does not exist

DUMPBIN or EDITBIN could not find a section called *name* in the input file.

To see the sections in a file, run DUMPBIN with the /HEADERS option.

Linker Tools Warning LNK4040

Error Message
corrupt string table (size); string table ignored

The string table in the object file is corrupt. The program may not run as expected. Rebuild the object file.

See Corrupt Object File for more information.

Linker Tools Warning LNK4041

Error Message
no edit options specified

EDITBIN was run on the input file, but no options were used.

If the file format was not COFF, EDITBIN attempted to convert it. Otherwise, EDITBIN took no action on the file.

Linker Tools Warning LNK4042

Error Message
object specified more than once; extras ignored

The same object file was specified more than once in the LINK or LIB command. The result is the same as if the file were specified only the first time.

Linker Tools Warning LNK4043

Error Message
invalid /ALIGN value number (must be power of 2); assumed default

The argument specified with the /ALIGN option was not a power of 2.

Linker Tools Warning LNK4044

Error Message
unrecognized option 'option'; ignored

The given option is not a valid option for this tool. The tool ignored the option.

To see a list of valid options, run the tool with no arguments or options.

Linker Tools Warning LNK4047

Error Message
cannot edit library file 'filename'; ignored

EDITBIN cannot process a library file.

Linker Tools Warning LNK4048

Error Message
Invalid format file; ignored

EDITBIN could not process the file because it was not a valid object file, executable file, or dynamic-link library (DLL).

Linker Tools Warning LNK4049

Error Message
locally defined symbol 'symbol' imported

The symbol was both exported from and imported to the program.

This warning is generated by the linker when you declare a symbol by using the **__declspec(dllexport)** storage-class attribute in one object file and reference it by using the **__declspec(dllimport)** attribute in another.

Warning LNK4049 is a more general version of Linker Tools Warning LNK4217. The linker generates Warning LNK4049 when it cannot determine from which function the imported symbol was referenced.

The common cases where LNK4049 is generated instead of LNK4217 are:

- Performing incremental linking by using the /INCREMENTAL option.
- Performing whole-program optimization by using the /LTCG option.

To resolve LNK4049, try one of the following:

- Remove the **__declspec(dllimport)** name declaration from the forward declaration of the symbol which triggered LNK4049. You can search for symbols within a binary image by using the **DUMPBIN** utility. The **DUMPBIN /SYMBOLS** switch displays the COFF symbol table of the image. For more information on the **DUMPBIN** utility, see DUMPBIN Reference.
- Temporarily disable incremental linking and whole-program optimization. Recompiling the application will generate Warning LNK4217, which will include the name of the function from which the imported symbol was referenced. Remove the **__declspec(dllimport)** declaration from the imported symbol and enable incremental linking or whole-program optimization as required.

Although the final generated code will behave correctly, the code generated to call the imported function is less efficient than calling the function directly. This warning will not appear when you compile by using the option /clr.

For more information on import and export data declarations, see dllexport, dllimport.

Example

Linking the following two modules will generate LNK4049. The first module generates an object file containing a single exported function.

```
// LNK4049a.cpp
// compile with: /c

__declspec(dllexport) int func()
{
   return 3;
}
```

The second module generates an object file containing a forward declaration to the function exported in the first module, along with a call to this function inside the `main` function. Linking this module with the first module will generate LNK4049. Removing the **__declspec(dllimport)** declaration will resolve the warning.

```
// LNK4049b.cpp
// compile with: /link /WX /LTCG LNK4049a.obj
// LNK4049 expected

__declspec(dllimport) int func();
// try the following line instead
// int func();

int main()
{
   return func();
}
```

See Also

Reference

Linker Tools Warning LNK4217

dllexport, dllimport

Linker Tools Warning LNK4051

Error Message
extra delimiter in /EXPORT specification

The specification of the **/EXPORT** option ended with a comma (,).

Linker Tools Warning LNK4052

Error Message
using .EXP file; ignoring .DEF file 'filename'

The .def file specified in the LINK command was ignored because an .exp file was also specified. The .def file was already processed when the .exp file was created and is not necessary in this step of the build process.

Linker Tools Warning LNK4056

Error Message
extra arguments ignored for option 'option'

The option is specified with too many arguments. The additional arguments were ignored.

Linker Tools Warning LNK4060

Error Message
stub file missing full MS-DOS header; rebuild stub with /KNOWEAS 16-bit LINK option

The MS-DOS application specified with the /STUB option does not have a full 40-byte header. The stub program may not run as expected.

To rebuild the 16-bit stub with a full 40-byte header, use the undocumented **/KNOWEAS** option with the 16-bit LINK version 5.5*x* or higher.

Linker Tools Warning LNK4061

Error Message

/NOSTUB ignored because no DOS stub is present

EDITBIN could not remove the stub file because it could not find one in your executable file.

Linker Tools Warning LNK4062

Error Message

'option' not compatible with 'platform' target machine; option ignored

The given option is not supported for the target platform of this build. LINK ignored the option and continued the build.

Linker Tools Warning LNK4065

Error Message
'function' cannot be ordered; ignored

The given function was not compiled as a packaged function. Recompile using /Gy.

Linker Tools Warning LNK4066

Error Message
DLL contains .sdata section

One or more objects were compiled using **/Gt**. Objects compiled with **/Gt** cannot be used to create a DLL.

Linker Tools Warning LNK4067

Error Message
ambiguous entry point; selected 'entry'

LINK found multiple entry points for the program.

Linker Tools Warning LNK4068

Error Message
/MACHINE not specified; defaulting to machinetype

LINK did not find a machine specification. If the default is incorrect, resulting in fatal error LNK1112, relink using the /MACHINE option.

Linker Tools Warning LNK4069

Error Message

cannot create map and/or checksum image; checksum set to 0

This warning can occur if IMAGEHLP.dll is not available.

Linker Tools Warning LNK4070

Error Message
/OUT:filename directive in .EXP differs from output filename 'filename'; ignoring directive

The *filename* specified in the NAME or LIBRARY statement when the .exp file was created differs from the output *filename* that was either assumed by default or specified with the /OUT option.

You will see this warning if you change the name of an output file in the development environment and where the project's .def file was not updated. Manually update the .def file to resolve this warning.

A client program that uses the resulting DLL might encounter problems.

Linker Tools Warning LNK4071

Error Message
cannot be incrementally linked on subsequent links

LINK found multiple definitions for one or more symbols, but /FORCE or **/FORCE:MULTIPLE** was used to create an output file regardless of errors. LINK deleted the incremental status (.ilk) file.

Linker Tools Warning LNK4072

Error Message
section count count exceeds max (number); image may not run

The number of sections (a COFF .exe or .dll file is composed of sections, similar to segments) in the output file is greater than the limit that can be handled by the loader.

You can define your own sections using the **data_seg**, **code_seg**, and **alloc_text** pragmas. For example, the following program defines three sections called .a1, .a2, and .a3.

```
#pragma data_seg(".a1")
int a1 = 1;
#pragma data_seg(".a2")
int a2 = 2;
#pragma data_seg(".a3")
int a3 = 3;

int main() {
   ;
}
```

You can view the names of the sections in your executable file by using the DUMPBIN program with the default **/SUMMARY** option. See the DUMPBIN Reference for more information.

To reduce the number of sections in your executable file:

- Remove unneeded **data_seg**, **code_seg** or **alloc_text** pragmas. See data_seg, code_seg, and alloc_text.
- Use the **/MERGE** option with LINK to combine sections. See /MERGE.

Linker Tools Warning LNK4073

Error Message
cannot create map for .ilk file; linking nonincrementally

There was not a large enough contiguous space in shared memory for LINK to create the incremental status (.ilk) file. LINK performed a nonincremental build.

Linker Tools Warning LNK4075

Error Message
ignoring "option1" due to "option2" specification

The second option overrides the first.

Mutually exclusive linker options are being specified. Examine your linker options. Where linker options are specified depends on how you are building your project.

- If you are building in the development environment, look in the linker property pages for your project, and see where both linker options are being specified. See How to: Specify Project Properties with Property Pages for more information.
- If you build at the command line, look at the linker options specified there.
- If you build with build scripts, look through your scripts to see where these linker options are being specified.

When you find where the mutually exclusive linker options are specified, remove one of the linker options.

Some specific examples:

- If you link a module that was compiled with **/ZI**, which implies an internal linker option called /EDITANDCONTINUE, and a module that was compiled with /OPT:REF, /OPT:ICF, or /INCREMENTAL:NO, which imply no /EDITANDCONTINUE, you will get LNK4075. See /Z7, /Zi, /ZI (Debug Information Format) for more information.

Linker Tools Warning LNK4076

Error Message
invalid incremental status file 'filename'; linking nonincrementally

LINK cannot write to the incremental status (.ilk) file. Either *filename* is corrupt or it is not an incremental linking database. Remove the file and relink.

Linker Tools Warning LNK4077

Error Message
exports file 'filename' used; ignoring other export specifications

The given exports file provides the program's exports. Exports specified by the **/EXPORT** option or the EXPORTS statement in a .def file were ignored.

Linker Tools Warning LNK4078

Error Message
multiple 'section name' sections found with different attributes

LINK found two or more sections that have the same name but different attributes.

This warning can be caused by an import library or exports file that was created by a previous version of LINK or LIB.

Recreate the file and relink.

Example

LNK4078 can also be caused by a breaking change: the section named by init_seg on x86 was read/write, it is now read only.

The following sample generates LNK4078.

```
// LNK4078.cpp
// compile with: /W1
// LNK4078 expected
#include <stdio.h>
#pragma warning(disable : 4075)
typedef void (__cdecl *PF)(void);
int cxpf = 0;   // number of destructors to call
PF pfx[200];   // pointers to destructors.

struct A { A() {} };

int myexit (PF pf) { return 0; }

#pragma section(".mine$a", read, write)
// try the following line instead
// #pragma section(".mine$a", read)
__declspec(allocate(".mine$a")) int ii = 1;

#pragma section(".mine$z", read, write)
// try the following line instead
// #pragma section(".mine$z", read)
__declspec(allocate(".mine$z")) int i = 1;

#pragma data_seg()
#pragma init_seg(".mine$m", myexit)
A bbbb;
A cccc;
int main() {}
```

Linker Tools Warning LNK4079

Error Message

no write permission for incremental status file 'filename'; linking nonincrementally

LINK cannot write to the incremental status (.ilk) file because *filename* does not have the correct permissions.

Linker Tools Warning LNK4081

Error Message
insufficient disk space; linking nonincrementally

There is not enough space on disk for LINK to create or write to the incremental status (.ilk) file.

Linker Tools Warning LNK4085

Error Message
more than number comments; skipping exestr

This warning occurs when the linker tool is converting an OMF object to a COFF object. It indicates there are more than 256 comments in the OMF module. For more information about comment strings, see #pragma comment in the C/C++ Preprocessor Reference.

Linker Tools Warning LNK4086

Error Message
entrypoint 'function' is not __stdcall with 'number' bytes of arguments; image may not run

The entry point for a DLL must be **__stdcall**. Either recompile the function with the /Gz option or specify **__stdcall** or WINAPI when you define the function.

Linker Tools Warning LNK4087

Error Message
CONSTANT keyword is obsolete; use DATA

This warning occurs if you use **CONSTANT** with your exports in a module definition file or on the command line. Use **DATA** instead.

Linker Tools Warning LNK4088

Error Message
image being generated due to /FORCE option; image may not run

The linker encountered an error. Normally it deletes the image it created because the image may not run due to the error. Since you used the /FORCE option, however, the linker did not remove the image.

Linker Tools Warning LNK4092

Error Message
shared writable section 'section' contains relocations; image may not run correctly

The linker emits this warning whenever you have a shared section to warn you of a potentially serious problem.

One way to share data between multiple processes is to mark a section as "shared." However, marking a section as shared can cause problems. For example, you have a DLL that contains declarations like this in a shared data section:

```
int var = 1;
int *pvar = &var;
```

The linker cannot resolve `pvar` because its value depends on where the DLL is loaded in memory, so it puts a relocation record in the DLL. When the DLL is loaded into memory, the address of `var` can be resolved and `pvar` assigned. If another process loads the same DLL but cannot load it at the same address, the relocation for the address of `var` will be updated for the second process and the first process's address space will point to the wrong address.

Linker Tools Warning LNK4093

Error Message
Drive/Directory component ignored in 'module-definition' statement

Your module-definition (.def) file used a statement incorrectly. The LIBRARY or NAME statements cannot contain directory or drive information. See Module-Definition Files for more information.

Linker Tools Warning LNK4094

Error Message

'filename' is an MS-DOS executable; use EXEHDR to dump it

The DUMPBIN Reference program only works with COFF-format executable files. Use EXEHDR on MS-DOS executables. EXEHDR is included with Visual C++ 1.5.

Linker Tools Warning LNK4096

Error Message
/BASE value "number" is invalid for Windows 95 and Windows 98; image may not run

The base address you specified is invalid. Windows 95 and Windows 98 executable files must have a base address greater than 0x400000. For more information on base addresses, see the /BASE linker option.

Linker Tools Warning LNK4097

Error Message
duplicate name 'name' in order file; using last one

Order files specify the order you want the linker to load your functions. In your order file, a function appears more than once, so the linker ignored all references to it except the last.

For more information on order files, see the /ORDER linker option.

Linker Tools Warning LNK4098

Error Message
defaultlib 'library' conflicts with use of other libs; use /NODEFAULTLIB:library

You are trying to link with incompatible libraries.

Note
The run-time libraries now contain directives to prevent mixing different types. You will receive this warning if you try to use different types or debug and non-debug versions of the run-time library in the same program. For example, if you compiled one file to use one kind of run-time library and another file to use another kind (for example, single-threaded versus multithreaded) and tried to link them, you will get this warning. You should compile all source files to use the same run-time library. See the Use Run-Time Library (**/MD**, **/MT**, **/LD**) compiler options for more information.

You can use the linker's /VERBOSE:LIB switch to determine which libraries the linker is searching. If you receive LNK4098 and want to create an executable file that uses, for example, the single-threaded, non-debug run-time libraries, use the **/VERBOSE:LIB** option to find out which libraries the linker is searching. The linker should print LIBC.lib and not LIBCMT.lib, MSVCRT.lib, LIBCD.lib, LIBCMTD.lib, or MSVCRTD.lib as the libraries searched. You can tell the linker to ignore the incorrect run-time libraries by using /NODEFAULTLIB for each library you want to ignore.

The table below shows which libraries should be ignored depending on which run-time library you want to use.

To use this run-time library	Ignore these libraries
Single-threaded (libc.lib)	libcmt.lib, msvcrt.lib, libcd.lib, libcmtd.lib, msvcrtd.lib
Multithreaded (libcmt.lib)	libc.lib, msvcrt.lib, libcd.lib, libcmtd.lib, msvcrtd.lib
Multithreaded using DLL (msvcrt.lib)	libc.lib, libcmt.lib, libcd.lib, libcmtd.lib, msvcrtd.lib
Debug Single-threaded (libcd.lib)	libc.lib, libcmt.lib, msvcrt.lib, libcmtd.lib, msvcrtd.lib
Debug Multithreaded (libcmtd.lib)	libc.lib, libcmt.lib, msvcrt.lib, libcd.lib, msvcrtd.lib
Debug Multithreaded using DLL (msvcrtd.lib)	libc.lib, libcmt.lib, msvcrt.lib, libcd.lib, libcmtd.lib

For example, if you received this warning and you want to create an executable file that uses the non-debug, single-threaded version of the run-time libraries, you could use the following options with the linker:

```
/NODEFAULTLIB:libcmt.lib /NODEFAULTLIB:msvcrt.lib /NODEFAULTLIB:libcd.lib /NODEFAULTLIB:lib
cmtd.lib /NODEFAULTLIB:msvcrtd.lib
```

Linker Tools Warning LNK4099

Error Message
PDB 'filename' was not found with 'object/library' or at 'path'; linking object as if no debug info

The linker was unable to find your .pdb file. Copy it into the directory that contains *object/library*.

To find the name of the .pdb file associated with the object file:

1. Extract an object file from the library with lib **/extract:***objectname***.obj** *xyz***.lib**.
2. Check the path to the .pdb file with **dumpbin /section:.debug$T /rawdata** *objectname***.obj**.

You could also compile with /Z7, so the pdb doesn't need to be used, or remove the /DEBUG linker option if you do not have .pdb files for the objects you are linking.

Linker Tools Warning LNK4102

Error Message
export of deleting destructor 'name'; image may not run correctly

The program has attempted to export a deleting destructor. The resulting delete may occur across a DLL boundary such that a process can free memory that it does not own. Make sure that the given symbol is not listed in your .def file, and that the symbol is not listed as an argument of the **/IMPORT** or **/EXPORT** option in the linker command line.

If you are rebuilding the C run-time library, you can ignore this message.

Linker Tools Warning LNK4103

Error Message
import of deleting destructor 'name'; image may not run correctly

The program has attempted to import a deleting destructor. The resulting delete may occur across a DLL boundary such that a process can free memory that it does not own. Make sure that the given symbol is not listed in your .def file, and that the symbol is not listed as an argument of the **/IMPORT** or **/EXPORT** option in the linker command line.

Linker Tools Warning LNK4104

Error Message
export of symbol 'symbol' should be PRIVATE

The *symbol* can be one of the following:

- **DllRegisterServer**
- **DllRegisterServerEx**
- **DllUnregisterServer**
- **DllGetClassObject**
- **DllCanUnloadNow**

This warning is emitted when you are building an import library for a DLL and export one of the above functions without specifying it as PRIVATE in the module-definition file. In general, these functions are exported for use only by OLE. Placing them in the import library can lead to unusual behavior when a program linked to the library incorrectly makes calls to them. For more information about the PRIVATE keyword, see EXPORTS.

Linker Tools Warning LNK4105

Error Message
no argument specified with option 'option'; ignoring option

This warning occurs only when the /LIBPATH option is set. If no directory is specified with this option, then the linker ignores the option and generates this warning message.

If you do not need to override the existing environmental library settings, remove the /LIBPATH option from the linker command line. If you want to use an alternate search path for libraries, specify the alternate path following the /LIBPATH option.

Example

```
link /libpath:c:\filepath\lib bar.obj
```

would direct the linker to search for the required libraries in `c:\filepath\lib` before searching in the default locations.

Linker Tools Warning LNK4108

Error Message
/ALIGN specified without /DRIVER; image may not run

The /ALIGN option has been specified without also specifying /DRIVER.

Bad alignment can prevent the operating system from loading the final image. Do not use **/ALIGN** unless building a driver.

Linker Tools Warning LNK4194

Error Message
/DELAYLOAD:dll name ignored

The linker cannot delay load the requested DLL.

Linker Tools Warning LNK4195

Error Message
unable to load dll name

The linker could not load the requested DLL.

Linker Tools Error LNK4196

Error Message
unable to find entrypoint 'name' in .dll

This error is emitted when the linker loads a .dll file but cannot find the expected entry point.

Linker Tools Warning LNK4197

Error Message
export 'exportname' specified multiple times; using first specification

An export is specified in multiple and different ways. The linker uses the first specification and ignores the rest.

If you are rebuilding the C run-time library, you can ignore this message.

If an export is specified exactly the same way multiple times, the linker will not issue a warning.

For example, the following contents of a .def file would cause this warning:

```
EXPORTS
   functioname      NONAME
   functioname      @10
```

To fix by checking the following possible causes

1. The same export is specified both on the command line (through export:) and in the .def file
2. The same export is listed twice in the .def file with different attributes.

Linker Tools Warning LNK4198

Error Message
base key 'keyname' not found - using default

The *keyname* specified in a /BASE option does not appear in the base address file.

Linker Tools Warning LNK4199

Error Message
/DELAYLOAD:dllname ignored; no imports found from dllname

The linker ignores *dllname* because it does not need any of the functions that *dllname* exports.

Linker Tools Warning LNK4200

Error Message
corrupt line number information in object file; ignored

The line number information in the object file is corrupt. Rebuild.

Linker Tools Warning LNK4202

Error Message
spawning full build due to /directive directive

Emitted when the linker encounters a directive specifying a full build is to be performed.

A directive is added to source code with the comment pragma.

Linker Tools Warning LNK4203

Error Message

error reading program database 'filename'; linking object as if no debug info

The PDB for the given file couldn't be read; the linker will continue to link the object without debug information. See Linker Tools Error LNK1200 for more information.

Linker Tools Warning LNK4204

Error Message
'filename' is missing debugging information for referencing module; linking object as if no debug info

The .pdb file has an erroneous signature. The linker will continue to link the object without debug information. You may want to recompile the object file using the /Zi option.

LNK4204 can occur if some of the objects in the library refer to a file that no longer exists. This could happen when rebuilding the solution, for example; an object file might be deleted and not rebuilt because of a compilation error. In this case, either compile with **/Z7**, or **/Fd**, to update the objects to refer to a single file per-library (that is not the default .pdb file name). For more information, see /Fd (Program Database File Name). Ensure that the .pdb is saved with the library every time it is updated in the source control system.

Linker Tools Warning LNK4205

Error Message

'filename' is missing current debugging information for referencing module; linking object as if no debug info

The .pdb file has out-of-date information. The linker will continue to link object without debug information. You may want to recompile the object file using the /Zi option.

Linker Tools Warning LNK4206

Error Message
precompiled type information not found; 'filename' not linked or overwritten; linking object as if no debug info

The given object file, compiled with /Yc, was either not specified in the LINK command or was overwritten.

A common scenario for this warning is when the .obj that was compiled with /Yc is in a library, and where there are no symbol references to that .obj from your code. In that case, the linker will not use (or even see) the .obj file. In this situation, you should recompile your code and use /Yl for the remaining objects (the objects that are not compiled with /Yc).

Linker Tools Warning LNK4207

Error Message

'filename' compiled /Yc /Yu /Z7; cannot create PDB; recompile with /Zi; linking object as if no debug info

Cannot create a .pdb file with the given object as it is currently compiled; recompile specifying the /Zi option and relink.

Linker Tools Warning LNK4208

Error Message
incompatible PDB format in 'filename'; delete and rebuild; linking object as if no debug info

The .pdb format for the file is incompatible; delete the .pdb and rebuild.

If you have a .obj or .lib that was built with an earlier version Visual C++ and if that file has .pdb files associated with it and you link with /DEBUG (Generate Debug Info), you will get LNK4208. In this case, either do not use **/DEBUG** or delete and rebuild your older .pdb files.

You might see this error if you have a mismatched compiler and linker. For example, if your compiler is from the current version of Visual C++ and your linker is from a previous version.

Linker Tools Warning LNK4209

Error Message

debugging information corrupt; recompile module; linking object as if no debug info

The file contains bad CV information; recompile the module and relink.

Linker Tools Warning LNK4210

Error Message
section section exists; there may be unhandled static initializers or terminators

Some code introduced static initializers or terminators, but the CRT or its equivalent (which needs to run the static initializers or terminators) isn't run when the application starts. Examples of code that would cause this:

- Global class variable with a constructor, destructor, or virtual function table.
- Global variable initialized with a non-compile-time constant.

To fix this problem, try one of the following:

- Remove all code with static initializers.
- Do not use /NOENTRY. After you remove /NOENTRY, you may also have to add msvcrt.lib, libcmt.lib, or libcmtd.lib to your linker command line.
- Add msvcrt.lib, libcmt.lib, or libcmtd.lib to your linker command line.
- When moving from /clr:pure compilation to /clr, remove the /ENTRY option from the linker-line. This will enable CRT initialization allowing static initializers to be executed at application startup.
- If your project is built with /ENTRY, and if /ENTRY is passed a function other than `_DllMainCRTStartup`, the function must call CRT_INIT. See Run-Time Library Behavior and Knowledge Base article Q94248, http://support.microsoft.com/default.aspx?scid=kb;en-us;94248 for more information.

The /GS compiler option requires CRT startup code.

See Also
Reference
Setting Linker Options

Linker Tools Warning LNK4216

Error Message
Exported entry point entry

An entry point was exported from a DLL. The entry point of a DLL does not need to be exported.

Linker Tools Warning LNK4217

Error Message
locally defined symbol 'symbol' imported in function 'function'

__declspec(dllimport) was specified for a symbol even though the symbol is defined locally. Remove the **__declspec** modifier to resolve this warning.

symbol is the symbol name that's defined within the image. *function* is the function that is importing the symbol.

This warning will not appear when you compile by using the option /clr./clr (Common Language Runtime Compilation)

LNK4217 can also occur if you attempt to link two modules together, when instead you should compile the second module with the import library of the first module.

```
// LNK4217.cpp
// compile with: /LD
#include "windows.h"
__declspec(dllexport) void func(unsigned short*) {}
```

And then,

```
// LNK4217b.cpp
// compile with: /c
#include "windows.h"
__declspec(dllexport) void func(unsigned short*) {}
```

Attempting to link these two modules will result in LNK4217, compile the second sample with the import library of the first sample to resolve.

Linker Tools Warning LNK4219

Error Message
fixup name fixup overflow. Target 'target symbol name' is out of range, inserting thunk

The linker inserted a thunk in a situation where the address or offset was unable to fit in the given instruction because the target symbol is too far away from the instruction's location.

You may want to reorder the image (using the /ORDER option, for example) to avoid the extra level of indirection.

Linker Tools Warning LNK4220

Error Message
invalid 'linker option' value 'value'; assumed default

An out-of-range value was specified with the /TLBID option. The default value for **/TLBID** is 1.

Linker Tools Warning LNK4221

Error Message
no public symbols found; archive member will be inaccessible

A .obj file with no public symbols was added to a library. The symbols in the object will not be accessible in subsequent linker operations that include the .lib.

To resolve this warning, do not link a .obj file with no public symbols into a library.

Linker Tools Warning LNK4222

Error Message
exported symbol 'symbol' should not be assigned an ordinal

The following symbols should not be exported by ordinal:

- **DllCanUnloadNow**
- **DllGetClassObject**
- **DllGetClassFactoryFromClassString**
- **DllInstall**
- **DllRegisterServer**
- **DllRegisterServerEx**
- **DllUnregisterServer**

These functions are always located by name, using `GetProcAddress`. The linker warns about this kind of export is because it could result in a larger image. This could happen if the range of your ordinal exports is large with relatively few exports. For example,

```
EXPORTS
   DllGetClassObject   @1
   MyOtherAPI      @100
```

will require 100 slots in the export address table with 98 of them (2-99) just filler. On the other hand,

```
EXPORTS
   DllGetClassObject
   MyOtherAPI      @100
```

will require two slots. (Be aware that you can also export with the /EXPORT linker option.)

Linker Tools Warning LNK4223

Error Message
the attribute1 attribute specified for section 'section' is ineffective without the attribute2 attribute

The shared (S) attribute was specified on a section that isn't writable (W). The loader won't make it shared unless it's also writable.

attribute1 is the attribute specified in the /SECTION linker option, in this case "SHARED (S)". *section* is the section setting the attribute via the **/SECTION** option. *attribute2* is the attribute that is missing for *attribute1* to succeed (in this case, "WRITE (W)").

Linker Tools Warning LNK4224

Error Message
option is no longer supported; ignored

An invalid, obsolete linker option was specified and ignored.

For example, LNK4224 can occur if a /comment directive appears in .obj. The /comment directive would have been added via the comment (C/C++) pragma, using the deprecated exestr option. Use dumpbin /ALL to view the linker directives in an .obj file.

If possible, modify the source for the .obj and remove the pragma. If you do ignore this warning, it is possible that an .executable compiled with **/clr:pure** will not run as expected.

Example

The following sample generates LNK4224.

```
// LNK4224.cpp
// compile with: /W1 /LD
// LNK4224 expected
#pragma comment(exestr, "aabbcc")
```

Linker Tools Warning LNK4225

Error Message
'file' has an uneditable file format

An attempt was made to use EDITBIN on a binary file that was not in the correct format. For example, if a .obj file was produced with the /GL compiler option, it is not a format that can be accessed by EDITBIN.

Linker Tools Warning LNK4226

Error Message

alignment specified (align value) exceeds target machine page size (default page size); image may not run

/DRIVER and /ALIGN (or the align parameter of /SECTION) were specified, but the align value exceeds the default page size; the loader may fail to load the resulting image.

Linker Tools Warning LNK4227

Error Message
metadata operation warning (HRESULT) : warning_message

The linker detected metadata differences when merging:

- One or more referenced assemblies with the assembly currently being built.
- One or more source code files in a compilation.

For example, LNK4227 may be caused if you have two global functions with the same name but parameter information declared differently (declarations are not consistent in all compilands). Use ildasm.exe /TEXT /METADATA *object_file* on each .obj file and you should see how the types differ.

LNK4227 also reports problems that originate with another tool. For example, al.exe; see Al.exe Tool Errors and Warnings.

The metadata problems must be fixed to resolve the warning.

For example, LNK4227 is generated when a referenced assembly was signed differently than the assembly that references it.

The following sample generates LNK4227:

```
// LNK4227.cpp
// compile with: /clr
using namespace System::Reflection;

[assembly:AssemblyDelaySignAttribute(false)];

int main() {}
```

and then,

```
// LNK4227b.cpp
// compile with: /clr LNK4227.cpp /W1
// LNK4227 expected
using namespace System::Reflection;
using namespace System::Runtime::CompilerServices;

[assembly:AssemblyDelaySignAttribute(true)];
// Try the following line instead
// [assembly:AssemblyDelaySignAttribute(false)];

ref class MyClass
{
};
```

The following sample generates LNK4227:

```
// LNK4227c.cpp
// compile with: /clr:oldSyntax
#using <mscorlib.dll>
using namespace System::Reflection;

[assembly:AssemblyDelaySignAttribute(false)];

int main() {}
```

and then,

```
// LNK4227d.cpp
// compile with: /clr:oldSyntax LNK4227c.cpp /W1
// LNK4227 expected
```

```
#using <mscorlib.dll>
using namespace System::Reflection;
using namespace System::Runtime::CompilerServices;

[assembly:AssemblyDelaySignAttribute(true)];

__gc class MyClass
{
};
```

LNK4227 can also be generated when version numbers in the wrong format are passed to assembly attributes. The '*' notation is specific to the AssemblyVersionAttribute. To resolve this warning, use only numbers in the version attributes other than AssemblyVersionAttribute.

The following sample generates LNK4227:

```
// LNK4227e.cpp
// compile with: /clr /LD /W1
using namespace System::Reflection;
[assembly:AssemblyVersionAttribute("2.3.*")];   // OK
[assembly:AssemblyFileVersionAttribute("2.3.*")];   // LNK4227
// try the following line instead
// [assembly:AssemblyFileVersionAttribute("2.3")];
```

Linker Tools Warning LNK4228

Error Message
invalid directive option found; ignored

Some options are invalid when editing a DLL, that is, when using EDITBIN Reference.

The linker will ignore *option*.

Linker Tools Warning LNK4229

Error Message
invalid directive /directive found; ignored

A directive passed via the comment pragma was not valid. The linker ignores */directive*.

Linker Tools Warning LNK4232

Error Message

name dll.ext contains non-ASCII characters, DLL may not load on system with ANSI codepage other than CodePage

A DLL was linked that contained a non-ASCII character. *CodePage* is the value returned by GetACP.

Linker Tools Warning LNK4233

Error Message

name dll.ext contains non-ASCII characters, DLL will not load if system does not have same ANSI codepage as the one used for linking this DLL

A DLL was linked, via import library, that contained a non-ASCII character. GetACP returns the current code page.

Linker Tools Warning LNK4236

Error Message

statement statement is obsolete. Use statement.

An obsolete keyword was used in a module definition file. For example, when working with module definition files, use the SECTIONS keyword and not SEGMENTS.

Linker Tools Warning LNK4237

Error Message
/SUBSYSTEM:NATIVE specified when importing from 'dll'; Use /SUBSYSTEM:CONSOLE or /SUBSYSTEM:WINDOWS.

/SUBSYSTEM:NATIVE was specified when building a windows (Win32) application that directly uses one or more of the following:

- kernel32.dll
- gdi32.dll
- user32.dll
- one of the msvcrt* dlls.

Resolve this warning by not specifying **/SUBSYSTEM:NATIVE**.

Linker Tools Warning LNK4238

Error Message
shared writable section 'section' contains the import address table; image may not run correctly

The linker emits this warning whenever you have a shared section to warn you of a potentially serious problem.

One way to share data between multiple processes is to mark a section as "shared." However, marking a section as shared can cause problems. For example, you have a function that is exported from a DLL that contains declarations like this: int f();

The address of f() will be written in the import table when you load the DLL. If another process loads the same DLL but can not load it at the same address, and the import table is shared, the import table entry will be updated for the second process and the first process will get the incorrect value.

Linker Tools Warning LNK4239

Error Message
invalid load config struct for SAFESEH image; generating non-SAFESEH image.

/SAFESEH was not specified and you supplied an old IMAGE_LOAD_CONFIG_DIRECTORY (_load_config_used) structure and/or without specifying the symbols __safe_se_handler_table or __safe_se_handler_count in the structure.

To resolve this error, use the latest CRT libraries, which contain the new definition, or modify the user-supplied custom load config struct to conform to the new definition.

The linker will produce an image without a table of safe exception handlers.

Linker Tools Warning LNK4240

Error Message
SAFESEH table contained in incompatible target machine type 'type'

An image that was marked for machine type *type* had a safe exception handler table. This means that the image is corrupt. Recompile the image.

Only x86 images should have a safe exception handler table.

Linker Tools Warning LNK4241

Error Message
invalid subsystem version number number

You can specify a version for the image's subsystem (/SUBSYSTEM). Each subsystem has a minimum version requirement. If the specified version is lower than the minimum, this warning will occur, but editbin will still modify the image to the specified subsystem version number.

Linker Tools Warning LNK4242

Error Message
'linker_option' overriding source setting

A linker option was specified that conflicts with a custom attribute setting in source code.

When a linker option setting conflicts with an equivalent value specified in source code, the linker option setting will override the equivalent setting specified in source code.

Linker Tools Warning LNK4245

Error Message
ignoring '/option' because of type MSIL image

A linker option was ignored because the linker is creating an MSIL image.

Linker Tools Warning LNK4246

Error Message
.netmodule entry point token (reason) ignored

A module was used as an input file to the linker, but the module contained an entry point. A module with an entrypoint may have been created with Ilasm.exe.

The linker ignored the entry point.

To resolve this linker warning, remove the (.entrypoint) specification, and recompile with ilasm.

For more information on modules as linker input, see .netmodule Files as Linker Input.

Linker Tools Warning LNK4247

Error Message
entry point 'decorated_function_name' already has a thread attribute; 'attribute' ignored

An entry point, specified with /ENTRY (Entry-Point Symbol), had a threading attribute, but /CLRTHREADATTRIBUTE (Set CLR Thread Attribute) was also specified, with a different threading model.

The linker ignored the value specified with /CLRTHREADATTRIBUTE.

To resolve this warning:

- Remove /CLRTHREADATTRIBUTE from your build.
- Remove attribute from your source code file.
- Remove both the attribute from source and /CLRTHREADATTRIBUTE from your build, and accept the default CLR threading model.
- Change the value passed to /CLRTHREADATTRIBUTE, such that, it agrees with the attribute in source.
- Change the attribute in source, such that, it agrees with the value passed to /CLRTHREADATTRIBUTE.

The following sample generates LNK4247

```
// LNK4247.cpp
// compile with: /clr /c
// post-build command: link /CLRTHREADATTRIBUTE:STA LNK4247.obj /entry:xxx /SUBSYSTEM:Conso
le
 [System::MTAThreadAttribute]
void xxx(){}
```

Linker Tools Warning LNK4248

Error Message
unresolved typeref token (token) for 'type'; image may not run

A type doesn't have a definition in MSIL metadata.

LNK4248 can occur when there is only a forward declaration for a type in an MSIL module (compiled with **/clr**), where the type is referenced in the MSIL module, and where the MSIL module is linked with a native module that has a definition for the type.

In this situation, the linker will provide the native type definition in the MSIL metadata, and this may provide for the correct behavior.

However, if a forward type declaration is a CLR type, then the linker's native type definition may not be correct

For more information, see /clr (Common Language Runtime Compilation).

To correct this error

- Provide the type definition in the MSIL module.

Example

The following sample generates LNK4248. Define struct A to resolve.

```
// LNK4248.cpp
// compile with: /clr /W1
// LNK4248 expected
struct A;
void Test(A*){}

int main() {
   Test(0);
}
```

The following sample has a forward definition of a type.

```
// LNK4248_2.cpp
// compile with: /clr /c
class A;   // provide a definition for A here to resolve
A * newA();
int valueA(A * a);

int main() {
   A * a = newA();
   return valueA(a);
}
```

The following sample generates LNK4248.

```
// LNK4248_3.cpp
// compile with: /c
// post-build command: link LNK4248_2.obj LNK4248_3.obj
class A {
public:
   int b;
};

A* newA() {
   return new A;
}

int valueA(A * a) {
   return (int)a;
```

```
}
```

Linker Tools Warning LNK4249

Error Message
directive 'threading_attribute' conflicts with command line; ignored

The threading attribute specified in source code was overridden by the threading model specified with **/CLRTHREADATTRIBUTE**.

For more information, see /CLRTHREADATTRIBUTE (Set CLR Thread Attribute).

To correct this error

- Remove the threading attribute from your source code or link without **/CLRTHREADATTRIBUTE**.

Example
The following sample generates LNK4249.

```
// LNK4249.cpp
// compile with: /clr /W1 /link /CLRTHREADATTRIBUTE:STA
// LNK4249 expected
ref class MyClass {
public:
   int i;
};
[System::MTAThreadAttribute]
int main() {
   System::Console::WriteLine("test");
}
```

Linker Tools Warning LNK4251

Error Message
AllowPartiallyTrustedCallersAttribute assembly attribute used without specifying /CLRUNMANAGEDCODECHECK

If you use AllowPartiallyTrustedCallersAttribute in your code, you should explicitly set **/CLRUNMANAGEDCODECHECK**. It is a potential security vulnerability if an image contains both the SuppressUnmanagedCodeSecurity and AllowPartiallyTrustedCallers attributes.

For more information, see /CLRUNMANAGEDCODECHECK (Add SupressUnmanagedCodeSecurityAttribute).

Example
The following sample generates LNK4251.

```
// LNK4251.cpp
// compile with: /clr /LD /W1
// LNK4251 expected
[assembly:System::Security::AllowPartiallyTrustedCallersAttribute];
```

Linker Tools Warning LNK4252

Error Message
must read and agree to the Data Collection Policy at "url" before using /ERRORREPORT:SEND

Before you can use the linker's **/errorReport:send** to send information on an internal linker error you must agree to Microsoft's policy on data collection.

The text of the LNK4252 diagnostic will refer you to a website where you can agree to Microsoft's policy on data collection.

For more information on error reporting, see /errorReport (Report Internal Compiler Errors).

Linker Tools Warning LNK4253

Error Message
section 'section1' not merged into 'section2'; already merged into 'section3'

The linker detected multiple, conflicting merge requests. The linker will ignore one of the requests.

A **/MERGE** option or directive is encountered and the *from* section has already been merged into a different section due to a previous **/MERGE** option or directive (or due to an implicit merge from the linker).

To resolve LNK4253, remove one of the merge requests.

When targeting x86 machines and Windows CE targets (ARM, MIPS, SH4, and Thumb) with Visual C++ 2005, the .CRT section is now read only. If your code depends on the previous behavior (.CRT sections are read/write), you could see unexpected behavior.

For more information, see,

- /MERGE (Combine Sections)
- comment (C/C++)

Example

In the following sample, the linker is instructed to merge the `.rdata` section twice, but into different sections. The following sample generates LNK4253.

```
// LNK4253.cpp
// compile with: /W1 /link /merge:.rdata=text2
// LNK4253 expected
#pragma comment(linker, "/merge:.rdata=.text")
int main() {}
```

Linker Tools Warning LNK4254

Error Message
section 'section1' (offset) merged into 'section2' (offset) with different attributes

The contents of one section were merged into another, but the attributes of the two sections are different. Your program may give unexpected results. For example, data you wanted to be read only may now be in a writable section.

To resolve LNK4254, modify or remove the merge request.

When targeting x86 machines and Windows CE targets (ARM, MIPS, SH4, and Thumb) with Visual C++ 2005, the .CRT section is now read only. If your code depends on the previous behavior (.CRT sections are read/write), you could see unexpected behavior.

For more information, see,

- /MERGE (Combine Sections)
- comment (C/C++)

Example
The following sample generates LNK4254.

```
// LNK4254.cpp
// compile with: /W1 /link /WX
// LNK4254 expected
#pragma comment(linker, "/merge:.data=.text")
int main() {}
```

Linker Tools Warning LNK4255

Error Message
library contain multiple objects of the same name; linking object as if no debug info

LNK4255 can occur when merging two static libraries and where each library has an .obj file that has the same name as an .obj file in the other library and where both .obj files were built with **/Zi**. In this case, the linker will not include debug symbols to the library for one of the .obj files.

Example

The following sample creates an .obj file that will be used to create a static library.

```
// LNK4255.cpp
// compile with: /c /Zi /FoLNK4255_obj1.obj
// post-build command: lib LNK4255_obj1.obj /out:LNK4255_lib1.lib
void func1() {}
```

The following sample creates another .obj file that will be used to create a static library.

After the LNK4255_lib2.lib static library is created, you would then run lib.exe again, to merge the two static libraries into one static library. For example, `lib LNK4255_lib1.lib LNK4255_lib2.lib /out:LNK4255_lib.lib.`.

```
// LNK4255_b.cpp
// compile with: /c /FoLNK4255_obj1.obj /Zi
// post-build command: lib LNK4255_obj1.obj /out:LNK4255_lib2.lib & lib LNK4255_lib1.lib  L
NK4255_lib2.lib /out:LNK4255_lib.lib.
void func2() {}
```

The following sample generates LNK4255.

```
// LNK4255_c.cpp
// compile with: LNK4255_lib.lib /Zi
// processor: x86
// LNK4255 expected
void func1();
void func2();
int main() {
   func1();
   func2();
}
```

Corrupt Object File

Sometimes the linker is unable to process your object files because they are corrupt. Besides the usual causes of corrupt files such as network problems or exposing a floppy disk to a magnetic field, sometimes the compiler writes an incorrect object file, which you should report to Microsoft Product Support Services.

After you have ruled out a hardware, operating system, or network problem as the cause of the corrupt object file, you should try the tips below. Changing compiler options and recompiling sometimes causes the compiler to write a non-corrupt object file.

To fix by using the following possible solutions

1. Turn off optimization with the /Od (Disable) option.
2. Disable minimal rebuild with the /Gm– (Enable Minimal Rebuild) option.
3. Compile with the /Gy (Enable Function-Level Linking) option to package functions.
4. Use a different code generation option. See the /G (Optimize for Processor) options.
5. Change the order of functions and global variables.

Profile-Guided Optimization Errors and Warnings

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, type the error number in the **Look for** box in the Index, or click a link below.

Profile-Guided Optimization Error PG0001

Profile-Guided Optimization Error PG0002

Profile-Guided Optimization Error PG0003

Profile-Guided Optimization Error PG0061

Profile-Guided Optimization Error PG0062

Profile-Guided Optimization Error PG0063

Profile-Guided Optimization Error PG0065

Profile-Guided Optimization Error PG0066

Profile-Guided Optimization Error PG0067

Profile-Guided Optimization Error PG0068

Profile-Guided Optimization Error PG0069

Profile-Guided Optimization Error PG0070

Profile-Guided Optimization Error PG0071

Profile-Guided Optimization Error PG0081

Profile-Guided Optimization Error PG0087

Profile-Guided Optimization Error PG0090

Profile-Guided Optimization Error PG0091

Profile-Guided Optimization Error PG0092

Profile-Guided Optimization Error PG0094

Profile-Guided Optimization Error PG0096

Profile-Guided Optimization Error PG0168

Profile-Guided Optimization Error PG0169

Profile-Guided Optimization Error PG0181

Profile-Guided Optimization Error PG0188

Profile-Guided Optimization Error PG1000

Profile-Guided Optimization Error PG1001

Profile-Guided Optimization Error PG1032

Profile-Guided Optimization Error PG1035

Profile-Guided Optimization Error PG1060

Profile-Guided Optimization Error PG1061

Profile-Guided Optimization Warning PG1033

Profile-Guided Optimization Warning PG1036

Profile-Guided Optimization Warning PG1038

Profile-Guided Optimization Warning PG1039

Profile-Guided Optimization Warning PG1051

Profile-Guided Optimization Warning PG1052

Profile-Guided Optimization Warning PG1053

Profile-Guided Optimization Warning PG1056

Profile-Guided Optimization Warning PG1058

Profile-Guided Optimization Warning PG1065

Profile-Guided Optimization Warning PG1066

Profile-Guided Optimization Warning PG1067

Profile-Guided Optimization Warning PG1087

Profile-Guided Optimization Error PG0001

Error Message

An unexpected internal error was detected in source file 'file', line number.

PG0001 represents an internal error. Contact Microsoft Product Support Services.

Profile-Guided Optimization Error PG0002

Error Message

Out of memory. Please increase the size of the page file.

Profile guided optimization tools can not allocate memory on the heap. Increase virtual memory size.

Profile-Guided Optimization Error PG0003

Error Message
Unhandled exception: reason.

PG0003 represents an internal error. To get help, see Getting Help from Microsoft Product Support Services (Visual Studio).

Profile-Guided Optimization Error PG0061

Error Message
The option 'option' requires an argument.

One of the options for a profile-guided optimizations utility was specified, but the required parameter was not passed.

For example, passing /weight but not specifying a value to pgomgr will generate PG0061.

Profile-Guided Optimization Error PG0062

Error Message
The option 'option' must be provided.

In some cases, an option must be specified with another option.

Profile-Guided Optimization Error PG0063

Error Message
Unrecognized option 'option'

An option to a profile-guided optimizations utility was specified, but the option was not recognized.

For example, misspelling /merge to pgomgr will generate PG0063:

```
pgomgr t.pgd /merged:8
```

Try the following command line instead:

```
pgomgr t.pgd /merge:8
```

Profile-Guided Optimization Error PG0065

Error Message
The argument 'arg' provided for the option 'option' is invalid.

One of the options for a profile-guided optimizations utility was specified, but the required parameter was not in the correct format.

For more information, see Tools for Profile-Guided Optimization.

For example, the following command line will generate PG0065:

```
pgomgr t.pgd /merge:xxx
```

Try the following command line instead:

```
pgomgr t.pgd /merge:6
```

Profile-Guided Optimization Error PG0066

Error Message

The option 'option1' conflicts with the option 'option2'.

Two mutually exclusive options were specified.

Profile-Guided Optimization Error PG0067

Error Message
The option 'option' was specified more than once, last occurrence was used.

One of the options for a profile-guided optimizations utility was specified twice.

For more information, see Tools for Profile-Guided Optimization.

For example, the following command line will generate PG0067:

```
pgomgr t.pgd /merge:3 /merge:8
```

Try the following command line instead:

```
pgomgr t.pgd /merge:8
```

Profile-Guided Optimization Error PG0068

Error Message
The option 'option1' must be used with 'option2'.

One of the options for a profile-guided optimizations utility was specified, but another, required option was not also specified.

For more information, see Tools for Profile-Guided Optimization.

Profile-Guided Optimization Error PG0069

Error Message
Please specify a PGD file.

An operation required a .pgd file, but no .pgd file was specified.

For more information, see Tools for Profile-Guided Optimization.

For example, the following command line will generate PG0069:

```
pgomgr *.pgc
```

Try the following command line instead:

```
pgomgr *.pgc MyApp.pgd
```

Profile-Guided Optimization Error PG0070

Error Message
Please specify exactly one PGD file.

An operation required a .pgd file, but two .pgd files were specified.

For more information, see Tools for Profile-Guided Optimization.

For example, the following command line will generate PG0070:

```
pgomgr *.pgc MyApp.pgd MyApp.pgd
```

Try the following command line instead:

```
pgomgr *.pgc MyApp.pgd
```

Profile-Guided Optimization Error PG0071

Error Message
'option' must be combined with an output option.

Some operations require that an option be specified that generates output.

Profile-Guided Optimization Error PG0081

Error Message
Unable to open file 'file' for reading.

The specified file was not found.

Profile-Guided Optimization Error PG0087

Error Message
Unable to open file 'file' for writing.

A .pgd file was read only and an operation required write access to the file.

Profile-Guided Optimization Error PG0090

Error Message
Read I/O operation on file 'file' failed.

A file could not be read. Check for file system problems, concurrent access, for example.

Profile-Guided Optimization Error PG0091

Write I/O operation on file ' *file* ' failed.

A file could not be written to. Check for file system problems, concurrent access, for example.

Profile-Guided Optimization Error PG0092

Error Message
Unable to get full path name for '%1'.

The executable path passed to pgosweep is not correct; the drive letter is not found or the folder structure is not found.

Profile-Guided Optimization Error PG0094

Error Message

Not enough virtual memory to map file file. Please increase the pagefile size.

The image being created is too big to be mapped to memory. Close some running applications or increase virtual memory.

Profile-Guided Optimization Error PG0096

Error Message
Cannot determine windows system directory.

The compiler could not determine the location of the windows system directory.

In order for profile guided optimizations to work, the compiler needs access to the windows system directory to ensure it is loading the correct version of system DLLs.

To resolve, ensure that your windows installation is correct, and, if necessary, reinstall the operating system.

Profile-Guided Optimization Error PG0168

Error Message
Couldn't open func function in module module

PG0168 represents a bug in profile guided optimizations. Contact Microsoft Product Support Services.

Profile-Guided Optimization Error PG0169

Error Message
Couldn't open module module in PGD file file

PG0169 represents a bug in profile guided optimizations. Contact Microsoft Product Support Services.

Profile-Guided Optimization Error PG0181

Error Message
The PGC file 'pgc_file' has already been merged (added) into the profile database 'pgd_file'. Ignored.

A pgomgr operation specified a .pgc file that has already been merged into the .pgd file. **pgomgr.exe** will not merge a .pgc file if it has already been merged unless you specify /force.

Profile-Guided Optimization Error PG0188

Error Message
No PGC files matching 'pgc_file' were found.

A pgomgr operation specified a .pgc file that was not found.

Profile-Guided Optimization Error PG1000

Error Message
The PGD file 'pgd_file' does not contain any profile data.

An operation that expects profile data to have been already merged into the .pgd file found an empty .pgd file.

Profile-Guided Optimization Error PG1001

Error Message
The PGD file 'file' is already optimized; no changes made. Use -force to force reoptimization.

An attempt was made to produce an optimized output file from the same sources that were used to produce an existing output file.

Profile-Guided Optimization Error PG1032

Error Message

Can find no running instrumented instance of image 'app'.

pgosweep was invoked but the specified application was not running. Ensure that ***app*** is running before you invoke pgosweep.

Profile-Guided Optimization Warning PG1033

Error Message

Cannot access file mapping 'file'.

The image being created is too big to be mapped to memory. Close some running applications or increase virtual memory.

Profile-Guided Optimization Error PG1035

Error Message
Cannot create PGC/PGD Id.

PG1035 represents an internal error. See Getting Help from Microsoft Product Support Services (Visual Studio).

Profile-Guided Optimization Warning PG1036

Error Message
Interrupted by allocation. Retrying...

The instrumented program is busy and the size of the .PGC file grew while the pgosweep was in progress.

pgosweep will retry automatically.

Profile-Guided Optimization Warning PG1038

Error Message
Access denied while attempting to open a mutex 'mutex'

PG1038 represents an error in profile guided optimizations. For information on getting help for this issue, see Getting Help from Microsoft Product Support Services (Visual Studio).

Profile-Guided Optimization Warning PG1039

Error Message
Failed waiting for quiet time to sweep.

The application did not allow pgosweep to run. If possible, run pgosweep when the application is in an idle state.

Profile-Guided Optimization Warning PG1051

Error Message
The file 'file' is not a valid instrumentation data file (.PGC).

pgomgr encountered a file that was not a valid .pgc file.

Profile-Guided Optimization Warning PG1052

Error Message

The instrumentation data file 'pgc_file' belongs to a different database.

.pgc files were found on disk whose names are correct for the merging into the current .pgd file, but the time stamp on the .pgc file shows that it was created for a different application than the one currently being profiled.

Profile-Guided Optimization Warning PG1053

Error Message

The instrumentation data file 'file' is inconsistent because one or more threads terminated abnormally.

PG1053 can occur if some threads of the instrumented bits were terminated abnormally during the profiling scenario run.

Profile-Guided Optimization Warning PG1056

Error Message
Please specify only one of /clear or PGC, not both.

When specifying pgomgr /clear, you cannot also specify a .pcg file. You can either clear the .pgd file or you can write .pgc data to the .pgd file, not both.

Profile-Guided Optimization Warning PG1058

Error Message
The PGC file 'file' file was generated by an incompatible revision of the profile runtime.

PG1058 can occur with pgomgr trying to merge a .pgc file that was created using a profile runtime version that does not match the version of the compiler used to create the .pgd file.

Profile-Guided Optimization Error PG1060

Error Message
Please specify '/merge', '/clear', or '/summary'.

When you invoke pgomgr, you must specify an option.

Profile-Guided Optimization Error PG1061

Error Message
Couldn't read profile data for 'pgd_file'.

PG1061 represents a bug in profile guided optimizations. Contact Microsoft Product Support Services.

Profile-Guided Optimization Warning PG1065

Error Message
Reading 'file' : 'reason'.

An error occured while reading the PGD file.

Profile-Guided Optimization Warning PG1066

Error Message

Missing or invalid stream 'reason1''reason2' in 'file'.

An error occured while reading the PGD file.

Profile-Guided Optimization Warning PG1067

Error Message

There was a problem updating PGD file 'file'.

A file could not be written to. Check for file system problems, concurrent access, for example.

Profile-Guided Optimization Warning PG1087

Error Message
The PGC file 'file' is not valid.

A .pgc file is corrupt, possibly due to an abnormal end to a profiling session. Rerun the profiling scenario and terminate normally.

Math Errors M6101 through M6205

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, or type the error number in the **Look for** box in the Index.

Math Error M6101

Error Message
invalid

Invalid operation.

This error can be caused when an operand is NaN (not a number) or infinity.

Program terminates with exit code 129.

Math Error M6102

Error Message
denormal

An operation generated a very small floating-point number, which be invalid due loss of significance. Denormal floating-point exceptions are usually masked, causing them to be trapped and operated upon.

Program terminates with exit code 130.

Math Error M6103

Error Message
divide by 0

A floating-point operation attempted to divide by zero.

Program terminates with exit code 131.

Math Error M6104

Error Message
overflow

An overflow occurred in a floating-point operation.

Program terminates with exit code 132.

Math Error M6105

Error Message
underflow

An underflow occurred in a floating-point operation. Underflow floating-point exceptions are usually masked, causing the underflowing value to be replaced by 0.0.

Program terminates with exit code 133.

Math Error M6106

Error Message
inexact

Loss of precision occurred in a floating-point operation. This exception is usually masked. Many floating-point operations cause a loss of precision.

Program terminates with exit code 134.

Math Error M6107

Error Message
unemulated

An attempt was made to execute a coprocessor instruction that is invalid or is not supported by the emulator.

Program terminates with exit code 135.

Math Error M6108

Error Message
square root

The operand in a square-root operation was negative.

Program terminates with exit code 136.

Note
The **sqrt** function in the C run-time library and the FORTRAN intrinsic function **SQRT** do not generate this error. The C **sqrt** function checks the argument before performing the operation and returns an error value if the operand is negative. The FORTRAN **SQRT** function generates the DOMAIN error M6201 instead of this error.

Math Error M6110

Error Message
stack overflow

A floating-point expression caused a floating-point stack overflow.

Stack-overflow floating-point exceptions are trapped up to a limit of seven levels in addition to the eight levels usually supported by the 8087/287/387 coprocessor.

Program terminates with exit code 138.

Math Error M6111

Error Message
stack underflow

A floating-point operation resulted in a stack underflow on the 8087/287/387 coprocessor or the emulator.

This error is often caused by a call to a **long double** function that does not return a value. For example, the following generates this error when compiled and run:

```
long double ld() {};
main ()
{
  ld();
}
```

Program terminates with exit code 139.

Math Error M6201

Error Message
'function' : _DOMAIN error

An argument to the given function was outside the domain of legal input values for that function.

Example

```
result = sqrt(-1.0)   // C statement
result = SQRT(-1.0)   !  FORTRAN statement
```

This error calls the **_matherr** function with the function name, its arguments, and the error type. You can rewrite the **_matherr** function to customize the handling of certain run-time floating-point math errors.

Math Error M6202

Error Message
'function' : _SING error

An argument to the given function was a singularity value for this function. The function is not defined for that argument.

This error calls the **_matherr** function with the function name, its arguments, and the error type. You can rewrite the **_matherr** function to customize the handling of certain run-time floating-point math errors.

Math Error M6203

Error Message
'function' : _OVERFLOW error

The given function result was too large to be represented.

This error calls the **_matherr** function with the function name, its arguments, and the error type. You can rewrite the **_matherr** function to customize the handling of certain run-time floating-point math errors.

Math Error M6205

Error Message
'function' : _TLOSS error

A total loss of significance (precision) occurred.

This error may be caused by giving a very large number as the operand of sin, cos, or tan because the operand must be reduced to a number between 0 and 2*pi.

NMAKE Errors U1000 through U4011

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, or type the error number in the **Look for** box in the Index.

NMAKE Fatal Error U1000

Error Message
syntax error : ')' missing in macro invocation

A left parenthesis, **(**, appeared without a matching right parenthesis, **)**, in a macro invocation. The correct form is **$(***name***)**; **$***n* is allowed for one-character names.

NMAKE Fatal Error U1001

Error Message
syntax error : illegal character 'character' in macro

The given character appears in a macro but is not a letter, number, or underscore.

This error can be caused by a missing colon in a macro expansion:

```
syntax error : illegal character '=' in macro
```

NMAKE Fatal Error U1002

Error Message
syntax error : invalid macro invocation '$'

A single dollar sign appears without an associated macro name. The correct form is **$(***name***)**. To specify a dollar sign, use a double dollar sign (**$$**) or precede the dollar sign with a caret (**^**).

NMAKE Fatal Error U1003

Error Message
syntax error : '=' missing in macro

A macro invocation contains a colon (**:**), which begins a substitution, but does not contain an equal sign (**=**).

The correct form is:

$(*macroname* **:** *oldstring*=*newstring***)**

NMAKE Fatal Error U1004

Error Message
syntax error : macro name missing

The syntax for using a macro is:

```
$(name)
```

To fix by checking the following possible causes

1. The name of a macro being defined was itself a macro invocation that expanded to nothing. For example, if the macro named `ONE` is undefined or has a null value, the following macro definition causes this error:

   ```
   $(ONE)=TWO
   ```

2. A macro invocation did not specify a name in the parentheses. The following specification causes this error:

   ```
   $()
   ```

NMAKE Fatal Error U1005

Error Message

syntax error : text must follow ':' in macro

A string substitution was specified for a macro, but the string to be changed in the macro was not specified.

NMAKE Fatal Error U1006

Error Message

syntax error : missing closing double quotation mark

An opening double quotation mark (") appeared without a closing double quotation mark.

NMAKE Fatal Error U1007

Error Message
double quotation mark not allowed in name

The specified target name or filename contained a double quotation mark (").

Double quotation marks can surround a filename but cannot be contained within it.

NMAKE Fatal Error U1017

Error Message
unknown directive '!directive'

The specified *directive* is not one of the recognized directives.

NMAKE Fatal Error U1018

Error Message
directive and/or expression part missing

The directive was incompletely specified.

The expression part of the directive is required.

NMAKE Fatal Error U1019

Error Message
too many nested !IF blocks

The limit on nesting of **!IF** directives was exceeded.

The **!IF** preprocessing directives include **!IF**, **!IFDEF**, **!IFNDEF**, **!ELSE IF**, **!ELSE IFDEF**, and **!ELSE IFNDEF**.

NMAKE Fatal Error U1020

Error Message
end-of-file found before next directive

An expected directive was missing.

For example, an **!IF** was not followed by an **!ENDIF**.

NMAKE Fatal Error U1021

Error Message
syntax error : !ELSE unexpected

An **!ELSE** directive was found that was not preceded by an **!IF** directive, or the directive was placed in a syntactically incorrect place.

The **!IF** preprocessing directives include **!IF**, **!IFDEF**, **!IFNDEF**, **!ELSE IF**, **!ELSE IFDEF**, and **!ELSE IFNDEF**.

NMAKE Fatal Error U1022

Error Message
missing terminating character for string/program invocation : 'char'

To fix by checking the following possible causes

1. The closing double quotation mark (**"**) in a string comparison in a directive was missing.
2. The closing bracket (**]**) in a program invocation in a directive was missing.

NMAKE Fatal Error U1023

Error Message
syntax error in expression

An expression is invalid.

Check the allowed operators and operator precedence.

NMAKE Fatal Error U1024

Error Message
illegal argument to !CMDSWITCHES

An unrecognized command switch was specified.

NMAKE Fatal Error U1025

Error Message
syntax error : !ENDIF unexpected

An **!ENDIF** directive was found that was not preceded by an **!IF** directive, or the directive was placed in a syntactically incorrect place.

The **!IF** preprocessing directives include **!IF**, **!IFDEF**, **!IFNDEF**, **!ELSE IF**, **!ELSE IFDEF**, and **!ELSE IFNDEF**.

NMAKE Fatal Error U1031

Error Message
filename missing (or macro is null)

An **!INCLUDE** directive was found, but the name of the file to be included was missing or a macro representing the filename expanded to nothing.

NMAKE Fatal Error U1033

Error Message
syntax error : 'string' unexpected

The string is not part of the valid syntax for a makefile.

To fix by checking the following possible causes

1. If the closing set of angle brackets (**<<**) for an inline file are not at the beginning of a line, the following error occurs:

   ```
   syntax error : 'EOF' unexpected
   ```

2. If a macro definition in the makefile contained an equal sign (**=**) without a preceding name or if the name being defined is a macro that expands to nothing, the following error occurs:

   ```
   syntax error : '=' unexpected
   ```

3. If the semicolon (**;**) in a comment line in TOOLS.INI is not at the beginning of the line, the following error occurs:

   ```
   syntax error : ';' unexpected
   ```

4. If the makefile has been formatted by a word processor, the following error can occur:

   ```
   syntax error : ':' unexpected
   ```

NMAKE Fatal Error U1034

Error Message
syntax error : separator missing

The colon (**:**) that separates targets and dependents is missing.

NMAKE Fatal Error U1035

Error Message
syntax error : expected ':' or '=' separator

Either a colon (**:**) or an equal sign (**=**) was expected.

To fix by checking the following possible causes

1. A colon did not follow a target.
2. A colon and no space (such as a:) followed a single-letter target. NMAKE interpreted it as a drive specification.
3. A colon did not follow an inference rule.
4. A macro definition was not followed by an equal sign.
5. A character followed a backslash (****) that was used to continue a command to a new line.
6. A string appeared that did not follow any NMAKE syntax rule.
7. The makefile was formatted by a word processor.

NMAKE Fatal Error U1036

Error Message

syntax error : too many names to left of '='

Only one string is allowed to the left of a macro definition.

NMAKE Fatal Error U1037

Error Message
syntax error : target name missing

A colon (**:**) was found before a target name was found.

At least one target is required.

NMAKE Fatal Error U1038

Error Message
internal error : lexer

Note the circumstances of the error and try to isolate the problem and create a reproducible test case before Contacting Microsoft Product Support Services.

NMAKE Fatal Error U1039

Error Message
internal error : parser

Note the circumstances of the error and try to isolate the problem and create a reproducible test case before Contacting Microsoft Product Support Services.

NMAKE Fatal Error U1040

Error Message
internal error : macro expansion

Note the circumstances of the error and try to isolate the problem and create a reproducible test case before Contacting Microsoft Product Support Services.

NMAKE Fatal Error U1041

Error Message
internal error : target building

Note the circumstances of the error and try to isolate the problem and create a reproducible test case before Contacting Microsoft Product Support Services.

NMAKE Fatal Error U1042

Error Message
internal error : expression stack overflow

Note the circumstances of the error and try to isolate the problem and create a reproducible test case before Contacting Microsoft Product Support Services.

NMAKE Fatal Error U1043

Error Message
internal error : temp file limit exceeded

Note the circumstances of the error and try to isolate the problem and create a reproducible test case before Contacting Microsoft Product Support Services.

NMAKE Fatal Error U1045

Error Message
spawn failed : message

A program or command called by NMAKE failed for the given reason.

NMAKE Fatal Error U1047

Error Message
argument before ')' expands to nothing

The parentheses following the preprocessing operator **DEFINED** or **EXIST** either were empty or contained an argument that evaluated to a null string.

NMAKE Fatal Error U1048

Error Message
cannot write to file 'filename'

NMAKE could not write to the given file.

One cause of this error is a read-only file specified with **/X**.

NMAKE Fatal Error U1049

Error Message
macro or inline file too long (maximum : 64K)

An inline file or a macro exceeded the limit of 64K.

NMAKE Fatal Error U1050

Error Message
message

The message specified with the **!ERROR** directive was displayed.

NMAKE Fatal Error U1051

Error Message
out of memory

NMAKE ran out of memory, including virtual memory, because the makefile was too large or complex.

To fix by using the following possible solutions

1. Free some space on disk.
2. Increase the size of the Windows NT paging file or the Windows swap file.
3. If only part of the makefile is being used, either divide the makefile into separate files or use **!IF** preprocessing directives to limit the amount that NMAKE must process. The **!IF** directives include **!IF**, **!IFDEF**, **!IFNDEF**, **!ELSE IF**, **!ELSE IFDEF**, and **!ELSE IFNDEF**.

NMAKE Fatal Error U1052

Error Message
file 'filename' not found

NMAKE could not find the file specified with one of the following:

- **/F** option
- **!INCLUDE** preprocessing directive
- At sign (**@**) specifier for a response file

Check that the file exists and the filename is spelled correctly.

NMAKE Fatal Error U1053

Error Message
file 'filename' unreadable

Possible causes

- File in use by another process.
- Bad area on disk.
- Bad file-allocation table.

NMAKE Fatal Error U1054

Error Message
cannot create inline file 'filename'

NMAKE failed to create the given inline file.

To fix by checking the following possible causes

1. A file by that name exists with a read-only attribute.
2. The disk is full.

NMAKE Fatal Error U1055

Error Message
out of environment space

The operating system ran out of room for environment variables.

Either increase the environment space or set fewer environment variables.

NMAKE Fatal Error U1056

Error Message
cannot find command processor

The command processor was not in the path specified in the **COMSPEC** or **PATH** environment variables.

NMAKE uses COMMAND.COM or CMD.EXE as a command processor when executing commands. It looks for the command processor first in the path set in **COMSPEC**. If **COMSPEC** does not exist, NMAKE searches the directories specified in **PATH**.

NMAKE Fatal Error U1057

Error Message
cannot delete temporary file 'filename'

NMAKE failed to delete the temporary inline file.

NMAKE Fatal Error U1058

Error Message
terminated by user

NMAKE was halted by CTRL+C or CTRL+BREAK.

NMAKE Fatal Error U1059

Error Message
syntax error : '}' missing in dependent

A search path for a dependent was incorrectly specified. Either a space existed in the path or the closing brace (**}**) was omitted.

The syntax for a directory specification for a dependent is

{ *directories* }dependent

where *directories* specifies one or more paths, each separated by a semicolon (**;**). No spaces are allowed.

If part or all of a search path is replaced by a macro, make sure no spaces exist in the macro expansion.

NMAKE Fatal Error U1060

Error Message
unable to close file : 'filename'

NMAKE encountered an error while closing a file.

To fix by checking the following possible causes

1. File is read-only.
2. Locking or sharing violation.
3. Disk full.

NMAKE Fatal Error U1061

Error Message
/F option requires a filename

The /F command-line option must be followed by a makefile name or a hyphen (which represents standard input).

NMAKE Fatal Error U1062

Error Message
missing filename with /X option

The /X command-line option requires the name of the file to contain diagnostic error output or a hyphen to indicate standard output.

NMAKE Fatal Error U1063

Error Message
missing macro name before '='

A macro definition on the NMAKE command line contained an equal sign (**=**) without a preceding name.

This error can occur if the macro name being defined is itself a macro that expands to nothing.

NMAKE Fatal Error U1064

Error Message
MAKEFILE not found and no target specified

The NMAKE command line did not specify a makefile or a target, and the current directory did not contain a file named MAKEFILE.

NMAKE requires either a makefile or a command-line target (or both). To make a makefile available to NMAKE, either specify the /F option or place a file named MAKEFILE in the current directory. NMAKE can create a command-line target by using an inference rule if a makefile is not provided.

NMAKE Fatal Error U1065

Error Message
invalid option 'option'

The option is not valid for NMAKE.

NMAKE Fatal Error U1069

Error Message
no match found for wildcard 'filename'

There is no file that matches the given filename, which was specified using one or more wildcards (***** and **?**).

A target file that is specified using a wildcard must exist on disk.

NMAKE Fatal Error U1070

Error Message
cycle in macro definition 'macroname'

The given macro definition contained a macro whose definition contained the given macro. Circular macro definitions are invalid.

Example
The following macro definitions

```
ONE=$(TWO)
TWO=$(ONE)
```

cause the following error:

```
cycle in macro definition 'TWO'
```

NMAKE Fatal Error U1071

Error Message
cycle in dependency tree for target 'targetname'

A circular dependency exists in the dependency tree for the given target. The given target is a dependent of one of the dependents of the given target. Circular dependencies are invalid.

NMAKE Fatal Error U1072

Error Message
cycle in include files : 'filename'

The given file includes a file that eventually includes the given file. Inclusions (using the **!INCLUDE** preprocessing directive) cannot be circular.

NMAKE Fatal Error U1073

Error Message
don't know how to make 'targetname'

The specified target does not exist, and there is no command to execute or inference rule to apply.

To fix by using the following possible solutions

1. Check the spelling of the target name.
2. If *targetname* is a pseudotarget, specify it as a target in another description block.
3. If *targetname* is a macro invocation, be sure it does not expand to a null string.

NMAKE Fatal Error U1076

Error Message
name too long

A string exceeded one of the following limits:

- 1024 characters for a macro name.
- 256 characters for a target pathname.
- 2048 characters for a command.

NMAKE Fatal Error U1077

Error Message
'program' : return code 'value'

The given command or program called by NMAKE failed and returned the given exit code.

To suppress this error and continue the NMAKE session, use the /I option, the **.IGNORE** dot directive, or the dash (**-**) command modifier. To continue the NMAKE session for unrelated parts of the dependency tree, use the /K option.

NMAKE Fatal Error U1078

Error Message
constant overflow at 'expression'

The given expression contained a constant that exceeded the range – 2,147,483,648 to 2,147,483,647. The constant appeared in one of the following situations:

- An expression specified with a preprocessing directive
- An error level specified with the dash (**-**) command modifier

NMAKE Fatal Error U1079

Error Message
illegal expression : divide by zero

An expression tried to divide by zero.

NMAKE Fatal Error U1080

Error Message
operator and/or operand usage illegal

The expression incorrectly used an operator or operand.

Check the allowed set of operators and their order of precedence.

NMAKE Fatal Error U1081

Error Message
'filename' : program not found

NMAKE could not find the given program in order to run it.

Make sure that the program is in a directory specified in the **PATH** environment variable and is not misspelled.

NMAKE Fatal Error U1082

Error Message
'command' : cannot execute command; out of memory

There is not enough memory to execute the given command.

One solution is to use the /N option to generate a batch file, then run the batch file instead of using NMAKE. In most cases the results will be the same.

NMAKE Fatal Error U1083

Error Message
target macro 'target' expands to nothing

The given target is an invocation of a macro that has not been defined or has a null value. NMAKE cannot process a null target.

NMAKE Fatal Error U1084

Error Message
cannot create temporary file 'filename'

NMAKE was unable to create the temporary file it needs when it processes the makefile.

To fix by checking the following possible causes

1. File already exists and is read-only.
2. Disk full.
3. Directory specified in the **TMP** environment variable does not exist.

NMAKE Fatal Error U1085

Error Message
cannot mix implicit and explicit rules

A target and a pair of inference-rule extensions were specified on the same line. Targets cannot be named in inference rules.

NMAKE Fatal Error U1086

inference rule cannot have dependents

The colon (**:**) in an inference rule must be followed by one of these:

- Newline character
- Semicolon (**;**), which can be followed by a command
- Number sign (**#**), which can be followed by a comment

NMAKE Fatal Error U1087

Error Message
cannot have : and :: dependents for same target

A target cannot be specified in both a single-colon (**:**) and a double-colon (**::**) dependency.

To specify a target in multiple description blocks, use **::** in each dependency line.

NMAKE Fatal Error U1088

Error Message
invalid separator '::' on inference rule

An inference rule must be followed by a single colon (**:**).

NMAKE Fatal Error U1089

Error Message
cannot have build commands for directive 'targetname'

Dot directives cannot be followed by commands. The dot directives are **.IGNORE**, **.PRECIOUS**, **.SILENT**, and **.SUFFIXES**.

NMAKE Fatal Error U1090

Error Message
cannot have dependents for directive 'targetname'

Dot directives cannot be followed by dependents. The dot directives are **.IGNORE**, **.PRECIOUS**, **.SILENT**, and **.SUFFIXES**.

NMAKE Fatal Error U1092

Error Message
too many names in rule

An inference rule cannot specify more than two extensions.

NMAKE Fatal Error U1093

Error Message
cannot mix dot directives

Multiple dot directives cannot be specified on one line. The dot directives are **.IGNORE**, **.PRECIOUS**, **.SILENT**, and **.SUFFIXES**.

NMAKE Fatal Error U1094

Error Message
syntax error : only (NO)KEEP allowed here

Something other than **KEEP** or **NOKEEP** appeared after the closing set of angle brackets (<<) specifying an inline file. Only **KEEP**, **NOKEEP**, or a newline character may follow the angle brackets. No spaces, tabs, or other characters may appear.

KEEP preserves the inline file on disk. **NOKEEP** deletes the file after the NMAKE session. The default is **NOKEEP**.

NMAKE Fatal Error U1095

Error Message
expanded command line 'commandline' too long

After macro expansion, the given command line exceeded the limit on length of command lines for the operating system.

MS-DOS permits up to 128 characters on a command line.

If the command is for a program that can accept command-line input from a file, change the command and supply input from either a file on disk or an inline file. For example, LINK and LIB accept input from a response file.

NMAKE Fatal Error U1096

Error Message
cannot open inline file 'filename'

NMAKE could not create the given inline file.

To fix by checking the following possible causes

1. Disk full.
2. File already exists and is read-only.

NMAKE Fatal Error U1097

Error Message
filename-parts syntax requires dependent

The current dependency does not have either an explicit dependent or an implicit dependent. Filename-parts syntax, which uses the percent (**%**) specifier, represents components of the first dependent of the current target.

NMAKE Fatal Error U1098

Error Message
illegal filename-parts syntax in 'string'

The given string does not contain valid filename-parts syntax.

NMAKE Fatal Error U1099

Error Message
stack overflow

The makefile being processed was too complex for the current stack allocation in NMAKE. NMAKE has an allocation of 0x3000 (12K).

To increase NMAKE's stack allocation, run the editbin /stack utility with a larger stack option:

editbin /STACK:reserve NMAKE.EXE

where *reserve* is a number greater than the current stack allocation in NMAKE.

NMAKE Fatal Error U1100

Error Message
macro 'macroname' is illegal in the context of batch rule 'rule'

NMAKE generates this error when the command block of a batch-mode rule directly or indirectly references a special file macro that is not $<.

$< is the only allowed macro for batch-mode rules.

NMAKE Fatal Error U2001

Error Message
no more file handles (too many files open)

NMAKE could not find a free file handle.

To fix by using the following possible solutions

1. Reduce recursion in the build procedures.
2. In MS-DOS, increase the number of file handles by changing the FILES setting in CONFIG.SYS to allow a larger number of open files. FILES=50 is the recommended setting.

NMAKE Warning U4001

Error Message
command file can be invoked only from command line

A command file, which is invoked by the at sign (**@**) specifier, cannot contain a specification for another command file. Such nesting is not allowed. The specification was ignored.

NMAKE Warning U4002

Error Message
resetting value of special macro 'macroname'

The given predefined macro was redefined.

NMAKE Warning U4004

Error Message
too many rules for target 'targetname'

More than one description block was specified for the given target using single colons (**:**) as separators. NMAKE executed the commands in the first description block and ignored later blocks.

To specify the same target in multiple dependencies, use double colons (**::**) as the separator in each dependency line.

NMAKE Warning U4005

Error Message
ignoring rule 'rule' (extension not in .SUFFIXES)

The given rule contained a suffix that is not specified in the **.SUFFIXES** list. NMAKE ignored the rule.

This warning appears only when the /P option is used.

NMAKE Warning U4006

Error Message
special macro undefined : 'macroname'

The given special macro name is undefined and expands to nothing.

NMAKE Warning U4007

Error Message
filename 'filename' too long; truncating to 8.3

The base name of the given file has more than eight characters, or the extension has more than three characters. NMAKE truncated the name to an eight-character base and a three-character extension.

If long filenames are supported by your file system, enclose the name in double quotation marks (**"**).

NMAKE Warning U4008

Error Message
removed target 'target'

NMAKE was interrupted while trying to build the given target, and the target file was incomplete. Because the target was not specified in the **.PRECIOUS** list, NMAKE deleted the file.

NMAKE Warning U4010

Error Message
'target' : build failed; /K specified, continuing ...

A command in the commands block for the given target returned a nonzero exit code. The /K option told NMAKE to continue processing unrelated parts of the build and to issue an exit code 1 when the NMAKE session is finished.

If the given target is, itself, a dependent for another target, NMAKE issues warning U4011 after this warning.

NMAKE Warning U4011

Error Message
'target' : not all dependents available; target not built

A dependent of the given target either did not exist or was out-of-date, and a command for updating the dependent returned a nonzero exit code. The /K option told NMAKE to continue processing unrelated parts of the build and to issue an exit code 1 when the NMAKE session is finished.

This warning is preceded by warning U4010 for each dependent that failed to be created or updated.

NMAKE Warning U4014

Error Message
must read and agree to the Data Collection Policy at "url" before using /errorreport:send

Before you can use nmake's **/errorReport:send** to send information on an internal error you must agree to Microsoft's policy on data collection.

The text of the U4014 diagnostic will refer you to a website where you can agree to Microsoft's policy on data collection.

For more information on error reporting, see /errorReport (Report Internal Compiler Errors).

C Run-Time Errors R6002 through R6035 and Warning CRT1001

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, or type the error number in the **Look for** box in the Index.

C Run-Time Error R6002

Error Message
floating-point support not loaded

The necessary floating-point library was not linked.

To fix by checking the following possible causes

1. The program was compiled or linked with an option, such as /FPi87, that requires a coprocessor, but the program was run on a machine that did not have a coprocessor installed.
2. A format string for a **printf_s** or **scanf_s** function contained a floating-point format specification and the program did not contain any floating-point values or variables.
3. The compiler minimizes a program's size by loading floating-point support only when necessary. The compiler cannot detect floating-point format specifications in format strings, so it does not load the necessary floating-point routines.
4. Use a floating-point argument to correspond to the floating-point format specification, or perform a floating-point assignment elsewhere in the program. This causes floating-point support to be loaded.
5. In a mixed-language program, a C library was specified before a FORTRAN library when the program was linked. Relink and specify the C library last.

C Run-Time Error R6008

Error Message
not enough space for arguments

There was enough memory to load the program but not enough memory to create the **argv** array.

To fix by using the following possible solutions

1. Increase the amount of memory available to the program.
2. Reduce the number and size of command-line arguments.
3. Reduce the environment size by removing unnecessary variables.

C Run-Time Error R6009

Error Message
not enough space for environment

There was enough memory to load the program, but not enough memory to create the **envp** array.

To fix by using the following possible solutions

1. Increase the amount of memory available to the program.
2. Reduce the number and size of command-line arguments.
3. Reduce the environment size by removing unnecessary variables.

C Run-Time Error R6016

Error Message
not enough space for thread data

The program did not receive enough memory from the operating system to complete a **_beginthread** call.

When a new thread is started, the library must create an internal database for the thread. If the database cannot be expanded with memory provided by the operating system, the thread will not begin and the calling process will stop.

C Run-Time Error R6017

Error Message
unexpected multithread lock error

The process received an unexpected error while trying to access a C run-time multithread lock on a system resource.

This error usually occurs if the process inadvertently alters the run-time heap data. However, it can also be caused by an internal error in the run-time or operating-system code.

C Run-Time Error R6018

Error Message
unexpected heap error

The program encountered an unexpected error while performing a memory-management operation.

This error usually occurs if the program inadvertently alters the run-time heap data. However, it can also be caused by an internal error in the run-time or operating-system code.

If your compiler provides a library containing **_heapchk** and **_heapwalk**, you can use these functions to diagnose this error.

C Run-Time Error R6019

Error Message
unable to open console device

The program called a console function declared in CONIO.h, but the operating system did not grant access to the console.

C Run-Time Error R6024

Error Message
not enough space for _onexit/atexit table

There was no memory available for the **_onexit** or **atexit** function. This error is caused by a low-memory condition.

C Run-Time Error R6025

Error Message
pure virtual function call

No object has been instantiated to handle the pure virtual function call.

If you are seeing this from an application, contact technical support for your application.

This error is caused by calling a virtual function in an abstract base class through a pointer which is created by a cast to the type of the derived class, but is actually a pointer to the base class. This can occur when casting from a **void*** to a pointer to a class when the **void*** was created during the construction of the base class.

For more information, see the Microsoft support website.

C Run-Time Error R6026

Error Message
not enough space for stdio initialization

This error occurs when there is not enough free memory available to initialize the standard I/O support.

C Run-Time Error R6027

Error Message
not enough space for lowio initialization

This error occurs when there is not enough free memory available to initialize the low-level I/O support.

C Run-Time Error R6028

Error Message
unable to initialize heap

This error occurs when the operating system failed to create the memory pool for the application. Specifically, the CRT called the Win32 function HeapCreate, which returned NULL indicating failure.

C Run-Time Error R6030

Error Message
CRT not initialized

This error occurs if you are using the CRT, but the CRT startup code was not executed. It is possible to get this error if the linker switch /ENTRY is used to override the default starting address, usually **mainCRTStartup**, **wmainCRTStartup** for a console EXE, **WinMainCRTStartup** or **wWinMainCRTStartup** for a Windows EXE, or **_DllMainCRTStartup** for a DLL. Unless one of the above functions is called on startup, the C Runtime will not be initialized.

C Run-Time Error R6031

Error Message
Attempt to initialize the CRT more than once. This indicates a bug in your application.

This diagnostic indicates that MSIL instructions were executing during loader lock. For more information, see Initialization of Mixed Assemblies.

C Run-Time Error R6032

Error Message
Not enough space for locale information

The runtime maintains information about the locale on each thread so that it can process calls to locale-sensitive functions. If the allocation of the memory for this information fails, the runtime is unable to proceed because too many of its basic facilities depend on it.

C Run-Time Error R6033

Error Message
Attempt to use MSIL code from this assembly during native code initialization. This indicates a bug in your application. It is most likely the result of calling an MSIL-compiled (/clr) function from a native constructor or from DllMain.

This diagnostic indicates that MSIL instructions were executing during loader lock. For more information, see Initialization of Mixed Assemblies.

C Run-Time Error R6034

Error Message
An application has made an attempt to load the C runtime library without using a manifest. This is an unsupported way to load Visual C++ DLLs. You need to modify your application to build with a manifest. For more information, see the "Visual C++ Libraries as Shared Side-by-Side Assemblies" topic in the product documentation.

Applications must use a manifest to load the C runtime library. For more information, see Visual C++ Libraries as Shared Side-by-Side Assemblies and Manifest Generation in Visual Studio.

In release builds, the diagnostic message reads: "An application has made an attempt to load the C runtime library incorrectly. Please contact the application's support team for more information."

To correct this error

- Rebuild your application with a manifest. Building an application with Visual Studio automatically puts the manifest into the resulting EXE or DLL file. If you are building at the command line, use the mt.exe tool to add the manifest as a resource. Use resource ID 1 if building an EXE, 2 if building a DLL. For more information, see How to: Embed a Manifest Inside a C/C++ Application.

C Run-Time Error R6035

Error Message
Microsoft Visual C++ Runtime Library, Error R6035 - A module in this application is initializing the module's global security cookie while a function relying on that security cookie is active. Call __security_init_cookie earlier.

__security_init_cookie must be called before the first use of the global security cookie.

The global security cookie is used for buffer overrun protection in code compiled with /GS (Buffer Security Check) and in code that uses structured exception handling. Essentially, on entry to an overrun-protected function, the cookie is put on the stack, and on exit, the value on the stack is compared against the global cookie. Any difference between them indicates that a buffer overrun has occurred and results in immediate termination of the program.

Error R6035 indicates that a call to **__security_init_cookie** was made after a protected function was entered. If execution were to continue, a spurious buffer overrun would be detected because the cookie on the stack would no longer match the global cookie.

Consider the following DLL example. The DLL entry point is set to DllEntryPoint through the linker /ENTRY (Entry-Point Symbol) option. This bypasses the CRT's initialization which would ordinarily initialize the global security cookie, so the DLL itself must call **__security_init_cookie**.

```
// Wrong way to call __security_init_cookie
DllEntryPoint(...) {
   DllInitialize();
   ...
   __try {
      ...
   } __except()... {
      ...
   }
}

void DllInitialize() {
   __security_init_cookie();
}
```

This example will generate error R6035 because DllEntryPoint uses structured exception handling and therefore uses the security cookie to detect buffer overruns. By the time DllInitialize is called, the global security cookie has already been put on the stack.

The correct way is demonstrated in this example:

```
// Correct way to call __security_init_cookie
DllEntryPoint(...) {
   __security_init_cookie();
   DllEntryHelper();
}

void DllEntryHelper() {
   ...
   __try {
      ...
   } __except()... {
      ...
   }
}
```

In this case, DllEntryPoint is not protected against buffer overruns (it has no local string buffers and doesn't use structured exception handling); therefore it can safely call **__security_init_cookie**. It then calls a helper function that is protected.

Note

Error message R6035 is only generated by the x86 debug CRT, and only for structured exception handling, but the condition is an error on all platforms, and for all forms of exception handling, such as C++ EH.

See Also

Reference

Compiler Security Checks In Depth

C Run-Time Warning CRT1001

Error Message
_vcclrit.h is deprecated

The header file _vcclrit.h used to be necessary to avoid loader lock, but is no longer needed and should not be included.

To correct this error

1. Define the symbol **_CRT_VCCLRIT_NO_DEPRECATE** before including _vcclrit.h.
2. Alternatively, remove all unnecessary legacy code that was added according to Converting Projects from Mixed Mode to Pure Intermediate Language.

See Also

Concepts
Mixed (Native and Managed) Assemblies

Resource Compiler Errors RC1000 through RC4413

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, or type the error number in the **Look for** box in the Index.

Resource Compiler Fatal Error RC1000

Error Message
UNKNOWN FATAL ERROR

Note the circumstances of the error and try to isolate the problem and create a reproducible test case before Contacting Microsoft Product Support Services.

Resource Compiler Fatal Error RC1002

Error Message
out of heap space

To fix by using the following possible solutions

1. Increase the Windows swap-file space. For more information about increasing the swap-file space, see virtual memory in Windows help.
2. Split the current file into smaller files and compile them separately.
3. Remove other programs or drivers running on the system.

Resource Compiler Fatal Error RC1003

Error Message
error count exceeds number; stopping compilation

The number of errors was too great to continue compiling. Fix some errors and recompile.

Resource Compiler Fatal Error RC1004

Error Message
unexpected end of file found

This error can be caused by missing linefeed and carriage return characters on the last line of a text file.

Resource Compiler Fatal Error RC1005

Error Message
file not found: filename

The file specified in the Resource Compiler command line was not found. Check to see whether the file has been moved to another directory and whether the filename or path is typed correctly.

Files are searched for using the **INCLUDE** environment variable or the Visual C++ Directories setting.

Resource Compiler Fatal Error RC1007

Error Message
unrecognized option 'option' in 'filename'

The specified option is not valid.

Resource Compiler Fatal Error RC1008

Error Message
no input file specified

One or more source files must be specified on the Resource Compiler command line.

Resource Compiler Fatal Error RC1009

Error Message
compiler limit : macros too deeply nested 'macro'

The file exceeds the Resource Compiler limit for macro nesting. Revise the specified source file to decrease the nesting depth of its macros.

Resource Compiler Fatal Error RC1010

Error Message
no output file specified

The Resource Compiler command line did not specify a filename for the compiled resource file.

Resource Compiler Fatal Error RC1011

Error Message

compiler limit : 'identifier' : macro definition too big

Try to split the definition into shorter definitions.

Resource Compiler Fatal Error RC1012

Error Message
unmatched parenthesis : missing 'character'

The parentheses in a preprocessor directive do not match.

Resource Compiler Fatal Error RC1013

Error Message
mismatched parentheses

Make sure that every open parenthesis has a matching closing parenthesis.

Resource Compiler Fatal Error RC1014

Error Message
too many include files : depth = 'level'

The nesting depth of **#include** directives was too great.

Use nested directives to include open files. The source file containing the directive is counted as one of the files.

Resource Compiler Fatal Error RC1015

Error Message
cannot open include file 'filename'

The given include file does not exist, could not be opened, or was not found.

Make sure that the environment settings are valid and that the correct path for the file is specified. Ensure that sufficient file handles are available to the Resource Compiler. If the file is on a network drive, make sure that you have permissions to open the file.

RC1015 can occur even if the include file exists in a directory specified as an Additional Include Directory in the Configuration Properties -> Resources -> General property page; specify the complete path to the include file.

For more information, see Knowledge Base article Q326987 : RC1015 Error When Using Resource View If the Include Path is Too Long.

Resource Compiler Fatal Error RC1016

Error Message
#if[n]def expected an identifier

The conditional compilation directives **#ifdef** or **#ifndef** require an identifier.

Resource Compiler Fatal Error RC1017

Error Message
invalid integer constant expression

The expression in an **#if** directive either did not exist or does not evaluate to a constant.

Resource Compiler Fatal Error RC1018

Error Message
unexpected '#elif'

The **#elif** directive did not appear within an **#if**, **#ifdef**, or **#ifndef** construct.

Make sure that there is an **#if**, **#ifdef**, or **#ifndef** statement in effect before this statement.

Resource Compiler Fatal Error RC1019

Error Message
unexpected '#else'

The **#else** directive did not appear within an **#if**, **#ifdef**, or **#ifndef** construct.

Make sure that there is an **#if**, **#ifdef**, or **#ifndef** statement in effect before this statement.

Resource Compiler Fatal Error RC1020

Error Message
unexpected '#endif'

An **#endif** directive appeared without a matching **#if**, **#ifdef**, or **#ifndef** directive.

Make sure that there is a matching **#endif** for each **#if**, **#ifdef**, and **#ifndef** statement.

Resource Compiler Fatal Error RC1021

Error Message
invalid preprocessor command 'string'

The characters following the number sign (**#**) did not form a valid preprocessor directive.

The number sign cannot be used as the first character in an identifier.

Resource Compiler Fatal Error RC1022

Error Message
expected '#endif'

An **#if**, **#ifdef**, or **#ifndef** directive was not terminated with an **#endif** directive.

Make sure that there is an **#if**, **#ifdef**, or **#ifndef** statement in effect before this statement.

Resource Compiler Fatal Error RC1023

Error Message
cannot open source file 'filename'

The specified file either did not exist, could not be opened, or was not found.

Make sure that the environment settings are valid and that the correct path for the file is specified.

If this error appears without an error message, the Resource Compiler has run out of file handles.

Resource Compiler Fatal Error RC1047

Error Message
"too many option options, 'string'"

The given option was specified too many times. The given string is the argument to the option that caused the error.

Resource Compiler Fatal Error RC1048

Error Message
unknown option 'character' in 'option'

The given character is not a valid letter for this option.

Resource Compiler Fatal Error RC1049

Error Message
invalid numerical argument 'string'

The Resource Compiler expected a numerical argument but received the given string.

This error may be caused by giving a hexadecimal number without the necessary \x prefix or by an incorrectly formed floating-point number.

Resource Compiler Fatal Error RC1052

Error Message
compiler limit : #if or #ifdef blocks nested too deeply

The program exceeded the maximum allowable nesting levels for **#if** and **#ifdef** directives.

This error can be caused by include files that use these preprocessor directives.

Resource Compiler Fatal Error RC1067

Error Message
compiler limit : identifier overflowed internal buffer

An internal compiler limit was exceeded.

Resource Compiler Fatal Error RC1101

Error Message
no resource binary filename specified

The Rename Output (/fo) option was not followed by a filename.

Use the following syntax for the /fo option:

RC /fo *outfilename infilename*. rc

Resource Compiler Fatal Error RC1102

Error Message
internal error : too many arguments to RCPP

Too many arguments were passed to the Resource Compiler preprocessor. Reduce the number of symbols defined with the Define Symbols (/d) option by defining them in your source. This error can also be caused by specifying too many include file search paths using the Include Search Path option (/i).

Resource Compiler Fatal Error RC1103

Error Message
invalid switch, option

The option is not a valid Resource Compiler option.

Resource Compiler Fatal Error RC1105

Error Message
invalid switch, option: too many /d switches

Too many symbols were defined using the Define Symbols (/d) option. Define some symbols in your source and recompile.

Resource Compiler Fatal Error RC1106

Error Message
invalid switch: option

The option is not a valid Resource Compiler option.

Valid options are:

/r

Emit .res file

/v

Verbose (print progress messages)

/d

Define a symbol

/fo

Rename .res file

/l

Default language ID in hex

/I

Add a path for **INCLUDE** searches

/x

Ignore **INCLUDE** environment variable

/c

Define a code page used by NLS Conversion

Resource Compiler Fatal Error RC1107

Error Message
invalid usage; use RC /? for Help

An invalid Resource Compiler option was specified. Valid options are /r, /v, /d, /fo, /l, /i, /x, and /c.

Resource Compiler Fatal Error RC1109

Error Message
error creating resource-name

Could not create specified .res file. Make sure it is not being created on a read-only drive. Use the /V option to find out whether the file is being created.

Resource Compiler Fatal Error RC1110

Error Message
could not open filename

The Resource Compiler could not open the specified resource script file. Make sure that the file exists.

Resource Compiler Fatal Error RC1116

Error Message
RC terminating after preprocessor errors

The Resource Compiler halted due to other errors. Fix the other errors and recompile.

Resource Compiler Fatal Error RC1120

Error Message
out of memory, needed number bytes

The Resource Compiler ran out of storage for items that it stores in its heap. Usually this is the result of having too many symbols.

To fix by using the following possible solutions

1. Increase the Windows swap file space. For more information about increasing the swap-file space, see virtual memory in Windows help.
2. Eliminate unnecessary include files, especially unneeded **#define**s and function prototypes.
3. Split the current file into two or more files and compile them separately.
4. Remove other programs or drivers running in the system, which could be consuming significant amounts of memory.

Resource Compiler Fatal Error RC1121

Error Message
I/O error reading file

The Resource Compiler was not able to read a file. Check that the drive containing the file is available and that the file is valid.

Resource Compiler Fatal Error RC1122

Error Message
I/O error writing file

The Resource Compiler could not write to a file.

To fix by checking the following possible causes

1. Insufficient disk space. Free space must equal about twice the size of the executable file you are creating.
2. Volume is read-only.
3. Bad sector.
4. Sharing violation.

Resource Compiler Fatal Error RC1123

Error Message
I/O error seeking in file

The compiler was unable to complete an I/O operation.

Resource Compiler Fatal Error RC1201

Error Message
invalid switch - missing include path after /i

The Specify Include Path (/i) option was not followed by a path.

Resource Compiler Fatal Error RC1202

Error Message
invalid switch - missing default language ID after /l

The Specify Default Language (/l) option was not followed by a hexadecimal language ID.

Resource Compiler Fatal Error RC1203

Error Message
invalid hexadecimal default language ID specified.

The Specify Default Language (/l) option was followed by an invalid hexadecimal language ID. See Language and Country/Region Strings in the *Run-Time Library Reference* for a list of valid language IDs.

Resource Compiler Fatal Error RC1204

Error Message
Invalid switch - missing code page after /c

The Specify Code Page (/c) option was not followed by a code page.

Resource Compiler Fatal Error RC1205

Error Message
invalid code page

The Specify Code Page (/c) option was followed by an invalid code page. See Code Pages in the *Run-Time Library Reference* for more information.

Resource Compiler Fatal Error RC1206

Error Message
specified code page at cmd line does not exist in registry

The Specify Code Page (/c) option was followed by a code page whose location is not specified in the registry. Run SETUP to install the specified code page.

Resource Compiler Fatal Error RC1207

Error Message
default code page is invalid

The code page specified by the **RCCODEPAGE** environment variable is not valid. See Code Pages in the *Run-Time Library Reference* for more information.

Resource Compiler Fatal Error RC1208

Error Message
input file has .RES extension

The .RES extension is used for Resource Compiler output. The .RC extension should be used for Resource Compiler scripts.

Resource Compiler Error RC2001

Error Message
newline in constant

A string constant was continued on a second line without either a backslash (****) or closing and opening double quotation marks (**"**).

To break a string constant that is on two lines in the source file, do one of the following:

- End the first line with the line-continuation character, a backslash.
- Close the string on the first line with a double quotation mark and open the string on the next line with another quotation mark.

It is not sufficient to end the first line with \n, the escape sequence for embedding a newline character in a string constant.

Resource Compiler Error RC2003

Error Message
expected 'defined identifier'

An identifier was expected after the specified preprocessing keyword.

Resource Compiler Error RC2004

Error Message
expected 'defined(identifier)'

An identifier was expected after the left parenthesis.

Resource Compiler Error RC2005

Error Message
#line expected a line number, found 'token'

A **#line** directive lacked the required line-number specification.

Resource Compiler Error RC2006

Error Message
#include expected a filename, found 'token'

An **#include** directive lacked the required filename.

Resource Compiler Error RC2007

Error Message
#define syntax

An identifier was expected following **#define** in a preprocessing directive.

Resource Compiler Error RC2008

Error Message
'character' : unexpected in macro definition

The character was found immediately following the name of the macro.

Resource Compiler Error RC2009

Error Message
reuse of macro formal 'identifier'

The identifier is used more than once in the formal parameter list of a macro definition.

Resource Compiler Error RC2010

Error Message
'character' : unexpected in formal list

The character is used incorrectly in the formal parameter list of a macro definition.

Resource Compiler Error RC2012

Error Message
missing name following '<'

An **#include** directive lacked the required filename specification.

Resource Compiler Error RC2013

Error Message
missing '>'

The closing angle bracket (**>**) was missing from an **#include** directive.

Resource Compiler Error RC2014

Error Message
preprocessor command must start as first non-whitespace

Non-whitespace characters appeared before the number sign (**#**) of a preprocessor directive on the same line.

Resource Compiler Error RC2015

Error Message
too many chars in constant

A character constant contained more than two characters.

Character constants are limited to one character (standard character constants) or two characters (long character constants).

Note that an escape sequence (for example, \t for tab) is converted to a single character.

Resource Compiler Error RC2016

Error Message
no closing single quote

A newline character was found before the closing single quotation mark of a character constant.

Resource Compiler Error RC2017

Error Message
illegal escape sequence

An escape sequence appeared where one was not expected.

An escape sequence — a backslash (****) followed by a number or letter — may occur only in a character or string constant.

Resource Compiler Error RC2018

Error Message
unknown character 'hexnumber'

The ASCII character corresponding to the given hexadecimal number appears in the source file but is an illegal character.

This error can be caused by source-file corruption.

Resource Compiler Error RC2019

Error Message
expected preprocessor directive, found 'character'

The given character followed a number sign (**#**), but it was not the first letter of a preprocessor directive.

Resource Compiler Error RC2020

Error Message
illegal digit number for base radix

The specified *digit* is not a valid digit for the base specified by *radix*. Either the digit or the radix could be incorrect.

Octal digits must be numbers from 0 to 7, and hexadecimal digits must be numbers from 0 to 9 or letters from A through E.

Resource Compiler Error RC2021

Error Message
expected exponent value, not 'character'

The given character was used as the exponent of a floating-point constant but was not a valid number.

Resource Compiler Error RC2022

Error Message

'number' : too big for character

The octal number following a backslash (****) in a character or string constant was too large to be represented as a character.

Resource Compiler Error RC2101

Error Message
Invalid directive in preprocessed RC file

The Resource Compiler file contains a **#pragma** directive.

Use the **#ifndef** preprocessor directive with the RC_INVOKED constant that the Resource Compiler defines when it processes an include file. Place the **#pragma** directive inside a block of code that is not processed when the RC_INVOKED constant is defined. Code in the block is processed only by the C/C++ compiler and not by the Resource Compiler. The following sample code demonstrates this technique:

```
#ifndef RC_INVOKED
#pragma pack(2)  // C/C++ only, ignored by Resource Compiler
#endif
```

The **#pragma** preprocessor directive has no meaning in an .RC file. The **#include** preprocessor directive is used frequently in an .RC file to include a header file (either a project-based custom header file or a standard header file provided by Microsoft with one of its products). Some of these include files contain the **#pragma** directive. Because a header file can include one or more other header files, the file that contains the offending **#pragma** directive may not be immediately obvious.

The **#ifndef** RC_INVOKED technique can control including header files in project-based header files.

Resource Compiler Error RC2102

Error Message
string literal too long

An **RCDATA** string exceeded the maximum allowable length.

Resource Compiler Error RC2103

Error Message
unexpected end of file in string literal

An end of file was found before the end of a string. The string is probably missing a closing double quotation mark (**"**).

Resource Compiler Error RC2104

Error Message
undefined keyword or key name: key

The specified keyword or key name was not defined.

If you encounter the following error message:

```
undefined keyword or key name: MFT_STRING
```

open MCL\MFC\Include\AfxRes.h and add the following include directive:

```
#include <winresrc.h>
```

Resource Compiler Error RC2105

Error Message
BEGIN expected in string table

The **BEGIN** keyword must immediately follow the **ACCELERATORS** keyword.

Resource Compiler Error RC2106

Error Message
BEGIN expected in accelerator table

The **BEGIN** keyword must immediately follow the **ACCELERATORS** keyword.

Resource Compiler Error RC2107

expected numeric command value

The Resource Compiler was expecting a numeric *idvalue* field in the **ACCELERATORS** statement. Make sure that you have used a **#define** constant to specify the value and that the constant is spelled correctly.

Resource Compiler Error RC2108

Error Message
unbalanced parentheses

An open parenthesis was not matched with a closing parenthesis.

Resource Compiler Error RC2109

Error Message
expected numerical dialog constant

A **DIALOG** statement requires integer values for the *x, y, width*, and *height* fields. Make sure these values are included after the **DIALOG** keyword and that they are not negative.

Resource Compiler Error RC2110

Error Message
expected numerical string ID

A **STRINGTABLE** statement requires a numerical string ID.

Resource Compiler Error RC2111

Error Message
invalid control type

Each CONTROL statement in a **DIALOG** statement must be one of the following: 3STATE, AUTO3, AUTOCHECK, AUTORADIO, BEDIT, CHECKBOX, COMBOBOX, CONTROL, CTEXT, DEFPUSHBUTTON, EDITTEXT, GROUPBOX, HEDIT, ICON, IEDIT, LISTBOX, LTEXT, PUSHBOX, PUSHBUTTON, RADIOBUTTON, RTEXT, SCROLLBAR, USERBUTTON.

Make sure these CONTROL statements are spelled correctly.

Resource Compiler Error RC2112

Error Message
BEGIN expected in dialog

The **BEGIN** keyword must immediately follow the **DIALOG** keyword.

Resource Compiler Error RC2113

Error Message
END expected in dialog

The **END** keyword must occur at the end of a **DIALOG** statement. Make sure there are no open quotes left from the preceding statement.

Resource Compiler Error RC2114

Error Message
expected control class name

The *class* field of a CONTROL statement in the **DIALOG** statement must be one of the following types: BUTTON, COMBOBOX, EDIT, LISTBOX, SCROLLBAR, STATIC, or user-defined. Make sure the class is spelled correctly.

Resource Compiler Error RC2115

Error Message
text string or ordinal expected in control

The *text* field of a CONTROL statement in the **DIALOG** statement must be either a text string or an ordinal reference to the type of control. If using an ordinal, make sure that you have a **#define** statement for the control.

Resource Compiler Error RC2116

Error Message
expecting number for ID

Expecting a number for the *id* field of a control statement in the **DIALOG** statement. Make sure you have a number or **#define** statement for the control ID.

Resource Compiler Error RC2117

Error Message
expected numeric point size

The *pointsize* field of the FONT option in the **DIALOG** statement must be an integer point size value.

Resource Compiler Error RC2118

Error Message
expected font face name

The *typeface* field of the FONT option in the **DIALOG** statement must be an ASCII character string enclosed in double quotation marks. This field specifies the name of a font.

Resource Compiler Error RC2119

Error Message
expecting quoted string in dialog title

The *captiontext* field of the CAPTION option in the **DIALOG** statement must be an ASCII character string enclosed in double quotation marks.

Resource Compiler Error RC2120

Error Message
expecting quoted string in dialog class

The *class* field of the CLASS option in the **DIALOG** statement must be an integer or a string, enclosed in double quotation marks.

Resource Compiler Error RC2121

Error Message
BEGIN expected in menu

The **BEGIN** keyword must immediately follow the **MENU** keyword.

Resource Compiler Error RC2122

Error Message
unknown menu subtype

The *item-definition* field of the **MENU** statement can contain only **MENUITEM** and **POPUP** statements.

Resource Compiler Error RC2123

Error Message
END expected in menu

The **END** keyword must come at the end of a **MENU** statement. Make sure you do not have any open quotation marks or a mismatched pair of **BEGIN** and **END** statements.

Resource Compiler Error RC2124

Error Message
empty menus not allowed

An **END** keyword appears before any menu items are defined in the **MENU** statement. The Resource Compiler does not permit empty menus. Make sure you do not have any open quotation marks within the **MENU** statement.

Resource Compiler Error RC2125

Error Message
expected ID value for menuitem

The **MENU** statement must contain a *menuID* field, which specifies the name or number that identifies the menu resource.

Resource Compiler Error RC2126

Error Message
expected menu string

Each **MENUITEM** and **POPUP** statement must contain a *text* field, which is a string enclosed in double quotation marks that specifies the name of the menu item or pop-up menu. A **MENUITEM SEPARATOR** statement requires no quoted string.

Resource Compiler Error RC2127

Error Message
version WORDs separated by commas expected

Version numbers in a version resource should be of type **WORD**, and separated by commas.

Resource Compiler Error RC2128

Error Message
DWORD expected

A valid version number was not found. It must be a **DWORD** type.

Resource Compiler Error RC2129

Error Message
BEGIN expected in VERSIONINFO resource

The **BEGIN** keyword must immediately follow the **VERSIONINFO** keyword.

Resource Compiler Error RC2130

Error Message
#line expected a string containing the file name, found 'token'

A **#line** directive was missing a required filename.

Resource Compiler Error RC2131

Error Message
expecting quoted string for key

The field for this statement must be a character string enclosed in double quotation marks.

Resource Compiler Error RC2132

Error Message
expected VALUE, BLOCK, or, END keyword

A block in a **VERSION** resource doesn't end properly, or a new block doesn't begin properly. This results in a **VERSION** statement that is not a valid block.

Resource Compiler Error RC2133

Error Message
unexpected value in value data

The *raw-data* values in the **RCDATA** statement must be integers or strings, each separated by a comma. Make sure you did not leave out a comma or leave out a quotation mark around a string.

Resource Compiler Error RC2134

Error Message
BEGIN expected in RCDATA

The **BEGIN** keyword must immediately follow the **RCDATA** keyword.

Resource Compiler Error RC2135

Error Message
file not found: filename

The file specified in the Resource Compiler command line was not found. Check to see whether the file has been moved to another directory and whether the file name and path are typed correctly.

Files are searched for using the **INCLUDE** environment variable or the Visual C++ **INCLUDE** setting.

Resource Compiler Error RC2136

Error Message
missing '=' in EXSTYLE=<flags>

An equal sign (**=**) was missing from an **EXSTYLE** (Extended Style Flags) statement. When the **EXSTYLE** is embedded in the **DIALOG** or **MENU** statement it must have the following form:

```
EXSTYLE=FLAGS
```

Resource Compiler Error RC2137

Error Message
empty character constant

A pair of single quotes was found with no character specified between them.

Resource Compiler Error RC2138

Error Message
unmatched close comment '/*'

An open comment sequence (`/*`) was not matched with a close comment sequence (`*/`) This error can be caused by nesting comments.

Resource Compiler Error RC2139

Error Message
VERSION not a number

A **VERSION** resource must be a number.

Resource Compiler Error RC2140

Error Message
CHARACTERISTICS not a number

A **CHARACTERISTICS** resource must be a number.

Resource Compiler Error RC2141

Error Message
invalid type

A different type was expected.

Resource Compiler Error RC2142

Error Message
ellipsis requires three periods

An incorrect number of periods (**.**) was used in an ellipsis.

Resource Compiler Error RC2143

Error Message
font names must be ordinals

The *pointsize* field in the **FONT** statement must be an integer, not a string.

Resource Compiler Error RC2144

Error Message
PRIMARY LANGUAGE ID not a number

The PRIMARY LANGUAGE ID must be a hexadecimal language ID. See Language and Country/Region Strings in the *Run-Time Library Reference* for a list of valid Language IDs.

Resource Compiler Error RC2145

Error Message
PRIMARY LANGUAGE ID too large

The PRIMARY LANGUAGE ID must be a valid hexadecimal language ID. See Language and Country/Region Strings in the *Run-Time Library Reference* for a list of valid Language IDs.

Resource Compiler Error RC2146

Error Message
missing COMMA in LANGUAGE statement

The comma separating the primary language and secondary language is missing.

The **LANGUAGE** statement must use the following syntax:

LANGUAGE *primary_language_ID,secondary_language_ID*

Resource Compiler Error RC2147

Error Message
SUBLANGUAGE ID not a number

The SUBLANGUAGE ID value must be a number.

The **LANGUAGE** statement must use the following syntax:

LANGUAGE *primary_language_ID,secondary_language_ID*

Valid SUBLANGUAGE IDs are defined as **SUBLANG_** constants in the WINNT.h file.

Resource Compiler Error RC2148

Error Message
SUBLANGUAGE ID too large

The SUBLANGUAGE ID value was out of range.

The **LANGUAGE** statement must use the following syntax:

LANGUAGE *primary_language_ID,secondary_language_ID*

Valid SUBLANGUAGE IDs are defined as **SUBLANG_** constants in the WINNT.h file.

Resource Compiler Error RC2149

Error Message
expected numeric constant in string table

A numeric constant, defined in a **#define** statement, must immediately follow the **BEGIN** keyword in a **STRINGTABLE** statement.

Resource Compiler Error RC2150

Error Message
expected string in STRINGTABLE

A string is expected after each *stringid* value in a **STRINGTABLE** statement.

Resource Compiler Error RC2151

Error Message
cannot re-use string constants

You are using the same value twice in a **STRINGTABLE** statement. Make sure you are not mixing overlapping decimal and hexadecimal values.

Each ID in a **STRINGTABLE** must be unique. For maximum efficiency use contiguous constants that start on a multiple of 16.

Resource Compiler Error RC2152

Error Message
invalid control character

A control character in the **ACCELERATORS** statement is invalid. A valid control character consists of one letter (only) following a caret (**^**).

Resource Compiler Error RC2153

Error Message
hex constants must have at least 1 hex digit

An empty hexadecimal constant was found.

Resource Compiler Error RC2154

Error Message
control character out of range [^A - ^Z]

A control character in the **ACCELERATORS** statement is invalid. The character following the caret (**^**) must be between A and Z, inclusive.

Resource Compiler Error RC2155

invalid accelerator

An *event* field in the **ACCELERATORS** statement was not recognized or was more than two characters in length.

Resource Compiler Error RC2156

Error Message
expected string or constant accelerator command

The Resource Compiler was not able to determine what kind of key is being set up for the accelerator. The *event* field in the **ACCELERATORS** statement might be invalid.

Resource Compiler Error RC2157

Error Message
expected comma in accelerator table

The Resource Compiler requires a comma between the *event* and *idvalue* fields in the **ACCELERATORS** statement.

Resource Compiler Error RC2159

Error Message
invalid accelerator type [ASCII or VIRTKEY]

The *type* field in the **ACCELERATORS** statement must contain either the ASCII or VIRTKEY value.

Resource Compiler Error RC2160

Error Message

cannot occur at the beginning of a macro definition

A macro definition began with a token-pasting operator (**##**).

Resource Compiler Error RC2161

Error Message
cannot occur at the end of a macro definition

A macro definition ended with a token-pasting operator (**##**).

Resource Compiler Error RC2162

Error Message
expected macro formal parameter

The token following a stringizing operator (**#**) was not a formal parameter name.

Resource Compiler Error RC2163

Error Message
accelerator type required [ASCII or VIRTKEY]

The *type* field in the **ACCELERATORS** statement must contain either the ASCII or VIRTKEY value.

Resource Compiler Error RC2164

Error Message
unexpected value in RCDATA

The *raw-data* values in the **RCDATA** statement must be integers or strings, each separated by a comma. Make sure you did not leave out a comma or leave out a quotation mark around a string.

Resource Compiler Error RC2165

Error Message
string not found in DLGINCLUDE statement

The statement did not specify a valid include file.

The **DLGINCLUDE** statement must use the following syntax:

DLGINCLUDE "*filename*.h"

Resource Compiler Error RC2166

Error Message
numeric value expected at line

The resource on the specified line must be a numeric value.

Resource Compiler Error RC2167

Error Message
unrecognized VERSIONINFO field; BEGIN or comma expected

An unrecognized field was found in the **FIXED** part of a **VERSIONINFO** structure declaration. A **VERSIONINFO** field must be DWORDS separated by a comma.

Resource Compiler Error RC2168

Error Message
resource too large

The size limitation of a resource was exceeded. This limit does not apply to cursors, icons, bitmaps, or other file-based resources.

Resource Compiler Error RC2169

Error Message
resource file filename is not in 2.03 format

The specified resource used a format earlier than version 2.03. The resource file must be converted or recreated using the format for version 3.00 or later.

Resource Compiler Error RC2170

Error Message
bitmap file filename is not in 3.00 format

Bitmaps using the Windows version 2.x format cannot be used in version 3.x resource files. The bitmap must be redrawn or converted to 3.x format.

Resource Compiler Error RC2171

Error Message
unknown DIB header format

The bitmap header is not a **BITMAPCOREHEADER** or **BITMAPINFOHEADER** structure.

Resource Compiler Error RC2174

Error Message
bitmap file filename is not in 2.03 format

A bitmap used a format earlier than version 2.03. The bitmap must be converted or redrawn using the format for version 3.00 or later.

Resource Compiler Error RC2175

Error Message
resource file filename is not in 3.00 format

The specified resource used a format earlier than version 3.00. The resource file must be converted or recreated using the format for version 3.00 or later.

Resource Compiler Error RC2176

old DIB in filename. Pass it through SDKPAINT.

An old format Device Independent Bitmap was found in the specified file. It should be converted to the current format. The SDKPAINT application provided in the Windows 3.0 SDK, or an equivalent application, can be used to do this.

Resource Compiler Error RC2177

Error Message
constant too big

A constant value was too large to be represented in the type to which it was assigned.

Resource Compiler Error RC2180

Error Message
unable to open temporary file

The Resource Compiler/Visual C++ was unable to open a temporary file. The probable cause is either that you do not have write permissions for the directory, or that the directory does not exist. The Resource Compiler/Visual C++ attempts to use these files in the directory specified by the **TMP** environment variable or the current directory if none is specified.

Resource Compiler Error RC2181

Error Message
duplicate font ID fontID

The specified font ID was already defined.

Resource Compiler Error RC2189

Error Message
#error : error

This error is used to display other error messages.

Resource Compiler Error RC2235

Error Message
too many arguments supplied

An expression contained more formal parameters than expected.

Resource Compiler Error RC2236

Error Message
required parameter missing

An expression contained fewer formal parameters than expected.

Resource Compiler Warning RC4000

Error Message
UNKNOWN WARNING

Note the circumstances of the warning and try to isolate the problem and create a reproducible test case before Contacting Microsoft Product Support Services.

Resource Compiler Warning RC4002

Error Message
too many actual parameters for macro 'identifier'

The number of actual parameters specified with the given identifier was greater than the number of formal parameters given in the macro definition of the identifier.

The additional actual parameters were collected but ignored during expansion of the macro.

Resource Compiler Warning RC4003

Error Message
not enough actual parameters for macro 'identifier'

The number of actual parameters specified with the given identifier was less than the number of formal parameters given in the macro definition of the identifier.

When a formal parameter is referenced in the definition and the corresponding actual parameter has not been provided, empty text is substituted in the macro expansion.

Resource Compiler Warning RC4004

Error Message
missing close parenthesis after 'defined'

An opening parenthesis was not matched with a closing parenthesis.

Resource Compiler Warning RC4005

Error Message
'identifier' : macro redefinition

The identifier is defined twice. The compiler used the second macro definition.

This warning can be caused by defining a macro on the command line and in the code with a **#define** directive. It also can be caused by macros imported from include files.

To eliminate the warning, either remove one of the definitions or use an **#undef** directive before the second definition.

Resource Compiler Warning RC4006

Error Message
#undef expected an identifier

The name of the identifier whose definition was to be removed was not given with the **#undef** directive. The **#undef** directive was ignored.

Resource Compiler Warning RC4009

Error Message
string too big, trailing chars truncated

A string was too large to fit in a buffer. Trailing characters were truncated and the remaining string was used.

Resource Compiler Warning RC4011

Error Message
identifier truncated to 'identifier'

An identifier was too long and was truncated to the name shown in the warning.

Resource Compiler Warning RC4012

Error Message
float constant in a cross compilation

A **float** constant may not have the same value on different target platforms.

Resource Compiler Warning RC4067

Error Message
unexpected characters following 'token' directive - newline expected

A newline character should follow the specified token.

Resource Compiler Warning RC4079

Error Message
unexpected token 'token'

Check the syntax of the line containing the specified token.

Resource Compiler Warning RC4093

Error Message
unescaped newline in character constant in inactive code

The constant expression of an **#if**, **#elif**, **#ifdef**, or **#ifndef** preprocessor directive evaluated to zero, making the code that follows inactive. Within that inactive code, a newline character appeared within a set of single or double quotation marks.

All text until the next double quotation mark was considered to be within a character constant.

Resource Compiler Warning RC4111

Error Message
unexpected token 'string'

Check the syntax of the line containing the specified token.

Resource Compiler Warning RC4117

Error Message
macro name 'macro' is reserved, macro ignored

The specified macro was given a reserved name.

Resource Compiler Warning RC4203

Error Message
SHIFT or CONTROL used without VIRTKEY

In an accelerator table resource, SHIFT or CONTROL require VIRTKEY. Because SHIFT and CONTROL are indicated as flag bits in a VIRTKEY type accelerator, they cannot exist independent from a VIRTKEY.

Resource Compiler Warning RC4204

Error Message
ASCII character not equivalent to virtual key code

A string literal was used for the virtual key code in a VIRTKEY type accelerator.

This warning allows you to continue, but be aware that the accelerator keys generated may not match the string you indicated. (VIRTKEYs use different key codes than ASCII accelerators.)

While string literals are syntactically valid, you can only ensure that you get the accelerator you want by using the **VK_* #define** values in WINDOWS.h.

Resource Compiler Warning RC4205

Error Message
string literal longer than 256 - stored anyway

A literal string exceeded 256 characters. It was stored intact.

Resource Compiler Warning RC4206

Error Message
name string too long - truncated at 256

A name string exceeded the limit of 256 characters. It was truncated to 256 characters.

Resource Compiler Warning RC4207

Error Message
type string too long - truncated at 256

A type string exceeded the limit of 256 characters. It was truncated to 256 characters.

Resource Compiler Warning RC4208

Error Message
title string too long - truncated at 256

A title string exceeded the limit of 256 characters. It was truncated to 256 characters.

Resource Compiler Warning RC4214

Error Message
Codepage not valid : ignored

The .rc file contained a codepage argument and the codepage specified is invalid. See IsValidCodePage for more information.

Resource Compiler Warning RC4413

Error Message

cannot determine file type for 'filename' : assuming 8-bit ASCII

The Resource Compiler does not know the type of specified file. It was compiled as an 8-bit ASCII file.

Resource Compiler Errors RW1004 through RW4004

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, or type the error number in the **Look for** box in the Index.

Resource Compiler Fatal Error RW1004

Error Message
Unexpected end of file

This error can be caused by missing linefeed and carriage-return characters on the last line of a text file.

Resource Compiler Fatal Error RW1009

Error Message
Error creating resource-name

Could not create specified .res file. Make sure it is not being created on a read-only drive. Use /V to find out whether the file is being created.

Resource Compiler Fatal Error RW1016

Error Message
RC terminating after preprocessor errors

The Resource Compiler halted due to other errors. Fix the other errors and recompile.

Resource Compiler Fatal Error RW1021

Error Message
I/O error reading file

The Resource Compiler was not able to read a file. Check that the drive containing the file is available and that the file is valid.

Resource Compiler Fatal Error RW1022

I/O error writing file

The Resource Compiler could not write to a file.

To fix by checking the following possible causes

1. Insufficient disk space. Free space must equal at least twice the size of the executable file you are creating.
2. Volume is read-only.
3. Bad sector.
4. Sharing violation.

Resource Compiler Fatal Error RW1023

Error Message
I/O error writing file, drive full

Free space must equal at least twice the size of the executable file you are creating.

Resource Compiler Fatal Error RW1025

Error Message
Out of far heap memory

Check for memory-resident software that might be taking up too much space. Use the CHKDSK program to find out how much memory you have.

If you are creating a large resource file, split the resource script into two files. After creating two .res files, use the MS-DOS command line to join them together:

```
copy first.res /b + second.res /b full.res
```

Resource Compiler Fatal Error RW1030

Error Message
Output Error

This error can be caused if the Resource Compiler was not able to create the destination file. Make sure that there is enough disk space and that you have write permission on the volume.

Resource Compiler Error RW2001

Error Message
Invalid directive in preprocessed RC file

The RC file contains a **#pragma** directive.

Use the **#ifndef** preprocessor directive with the **RC_INVOKED** constant that the Resource Compiler defines when it processes an include file. Place the **#pragma** directive inside a block of code that is not processed when the **RC_INVOKED** constant is defined. Code in the block is processed only by the C/C++ compiler and not by the Resource Compiler. The following sample code demonstrates this technique:

```
#ifndef RC_INVOKED
#pragma pack(2)  // C/C++ only, ignored by Resource Compiler
#endif
```

The **#pragma** preprocessor directive has no meaning in an .RC file. The **#include** preprocessor directive is used frequently in an .RC file to include a header file (either a project-based custom header file or a standard header file provided by Microsoft with one of its products). Some of these include files contain the **#pragma** directive. Because a header file can include one or more other header files, the file that contains the offending **#pragma** directive may not be immediately obvious.

The **#ifndef RC_INVOKED** technique can control including header files in project-based header files.

Resource Compiler Error RW2002

Error Message
Parsing error

To fix by checking the following possible causes

1. **Accelerator type required (ASCII or VIRTKEY)**

   The *type* field in the **ACCELERATORS** statement must contain either the ASCII or VIRTKEY value.

2. **BEGIN expected in accelerator table**

   The **BEGIN** keyword must immediately follow the **ACCELERATORS** keyword.

3. **BEGIN expected in dialog**

   The **BEGIN** keyword must immediately follow the **DIALOG** keyword.

4. **BEGIN expected in menu**

   The **BEGIN** keyword must immediately follow the **MENU** keyword.

5. **BEGIN expected in RCData**

   The **BEGIN** keyword must immediately follow the **RCDATA** keyword.

6. **BEGIN keyword expected in string table**

   The **BEGIN** keyword must immediately follow the **STRINGTABLE** keyword.

7. **Cannot re-use string constants**

   You are using the same value twice in a **STRINGTABLE** statement. Make sure you are not mixing overlapping decimal and hexadecimal values. Each ID in a **STRINGTABLE** must be unique. For maximum efficiency, use contiguous constants that start on a multiple of 16.

8. **Control character out of range [^A - ^Z]**

   A control character in the **ACCELERATORS** statement is invalid. The character following the caret (**^**) must be between A and Z, inclusive.

9. **Empty menus not allowed**

   An **END** keyword appears before any menu items are defined in the **MENU** statement. The Resource Compiler does not permit empty menus . Make sure you do not have any open quotation marks within the **MENU** statement.

10. **END expected in dialog**

    The **END** keyword must occur at the end of a **DIALOG** statement. Make sure there are no open quotes left from the preceding statement.

11. **END expected in menu**

    The **END** keyword must come at the end of a **MENU** statement. Make sure you do not have any open quotation marks or a mismatched pair of **BEGIN** and **END** statements.

12. **Expected comma in accelerator Table**

    The Resource Compiler requires a comma between the *event* and *idvalue* fields in the **ACCELERATORS** statement.

13. **Expected control class name**

    The *class* field of a **CONTROL** statement in the **DIALOG** statement must be one of the following types: BUTTON, COMBOBOX, EDIT, LISTBOX, SCROLLBAR, STATIC, or user-defined. Make sure the class is spelled correctly.

14. **Expected font face name**

    The *typeface* field of the FONT option in the **DIALOG** statement must be an ASCII character string enclosed in double quotation marks. This field specifies the name of a font.

15. **Expected ID value for menuitem**

    The **MENU** statement must contain a *menuID* field, which specifies the name or number that identifies the menu resource.

16. **Expected menu string**

    Each **MENUITEM** and **POPUP** statement must contain a *text* field, which is a string enclosed in double quotation marks that specifies the name of the menu item or pop-up menu. A **MENUITEM SEPARATOR** statement requires no quoted string.

17. **Expected numeric command value**

    The Resource Compiler was expecting a numeric *idvalue* field in the **ACCELERATORS** statement. Make sure that you have used a **#define** constant to specify the value and that the constant is spelled correctly.

18. **Expected numeric constant in string table**

    A numeric constant, defined in a **#define** statement, must immediately follow the **BEGIN** keyword in a **STRINGTABLE** statement.

19. **Expected numeric point size**

    The *pointsize* field of the FONT option in the **DIALOG** statement must be an integer point size value.

20. **Expected numerical dialog constant**

    A **DIALOG** statement requires integer values for the *x, y, width*, and *height* fields. Make sure that these values are included after the **DIALOG** keyword and that they are not negative.

21. **Expected string in STRINGTABLE**

    A string is expected after each *stringid* value in a **STRINGTABLE** statement.

22. **Expected string or constant accelerator command**

    The Resource Compiler was not able to determine what kind of key is being set up for the accelerator. The *event* field in the **ACCELERATORS** statement might be invalid.

23. **Expecting number for ID**

    Expecting a number for the *id* field of a control statement in the **DIALOG** statement. Make sure you have a number or **#define** statement for the control ID.

24. **Expecting quoted string in dialog class**

    The *class* field of the CLASS option in the **DIALOG** statement must be an integer or a string, enclosed in double quotation marks.

25. **Expecting quoted string in dialog title**

    The *captiontext* field of the CAPTION option in the **DIALOG** statement must be an ASCII character string enclosed in double quotation marks.

26. **File not found: filename**

    The file specified in the Resource Compiler command line was not found. Check to see whether the file has been moved to another directory and whether the filename or path is typed correctly. Files are searched for using the **INCLUDE** environment variable or the Visual Workbench setting, if available.

27. **Font names must be ordinals**

    The *pointsize* field in the FONT statement must be an integer, not a string.

28. **Invalid accelerator**

    An *event* field in the **ACCELERATORS** statement was not recognized or was more than two characters in length.

29. **Invalid accelerator type (ASCII or VIRTKEY)**

    The *type* field in the **ACCELERATORS** statement must contain either the ASCII or VIRTKEY value.

30. **Invalid control character**

A control character in the **ACCELERATORS** statement is invalid. A valid control character consists of one letter (only) following a caret (^).

31. **Invalid control type**

    Each control statement in a **DIALOG** statement must be one of the following: CHECKBOX, COMBOBOX, CONTROL, CTEXT, DEFPUSHBUTTON, EDITTEXT, GROUPBOX, ICON, LISTBOX, LTEXT, PUSHBUTTON, RADIOBUTTON, RTEXT, SCROLLBAR. Make sure these control statements are spelled correctly.

32. **Invalid type**

    The resource type was not among the types defined in the WINDOWS.h file.

33. **Text string or ordinal expected in control**

    The *text* field of a **CONTROL** statement in the **DIALOG** statement must be either a text string or an ordinal reference to the type of control. If using an ordinal, make sure that you have a **#define** statement for the control.

34. **Mismatched parentheses**

    Make sure you have closed every open parenthesis in the **DIALOG** statement.

35. **Unexpected value in RCData**

    The *raw-data* values in the **RCDATA** statement must be integers or strings, each separated by a comma. Make sure you did not leave out a comma or leave out a quotation mark around a string.

36. **Unknown menu subtype**

    The *item-definition* field of the **MENU** statement can contain only **MENUITEM** and **POPUP** statements.

Resource Compiler Error RW2003

Error Message
Generation Error

To fix by checking the following possible causes

1. **Error: Bitmap file resource-file is not in 3.00 format**

   Bitmaps using the Windows version 2.x format cannot be used in version 3.x resource files. The bitmap must be redrawn or converted to 3.x format.

2. **Error: Old DIB in resource-name. Pass it through SDKPAINT**

   A Device Independent Bitmap (DIB) in the specified resource is not compatible with the Windows 3.0 format. The bitmap must be redrawn or converted to the 3.x format.

3. **Error: Resource file resource-name is not in 3.00 format**

   An icon or cursor in the specified resource used a format from a previous version of Windows. The icon or cursor must be redrawn or converted to the 3.x format.

4. **Unknown DIB header format**

   The bitmap header is not a BITMAPCOREHEADER or BITMAPINFOHEADER structure.

5. **Unable to initialize symbol information**

   This error occurs only in Visual C++. The probable cause is that you have too many open files or you cannot open or write to the data files needed for Visual C++ to import the symbols in your script. Visual C++ attempts to create these files in the directory specified by the **TMP** environment variable or the current directory if none is specified.

6. **Unable to save symbol information**

   This error occurs only in Visual C++. The probable cause is that you have too many open files or you cannot close or write to the data files needed for Visual C++ to import the symbols in your script. Visual C++ attempts to use these files in the directory specified by the **TMP** environment variable or the current directory if none is specified.

7. **Bitmap file resource file is not in 2.03 format**

   A bitmap used a format earlier than version 2.03. The bitmap must be converted or redrawn using the format for version 3.00 or later.

8. **Resource too large**

   For Windows 3.1 a resource cannot exceed approximately 65000 bytes. If your resource does, then you will not be able to compile it with Visual C++ or the command-line resource compiler. This limit does not apply to cursors, icons, bitmaps, or other file-based resources.

9. **Resource file is not in 3.00 format**

   A cursor or icon used a format earlier than version 3.00. The resource must be converted or redrawn using the format for version 3.00 or later.

10. **Unable to open temporary file**

    The Resource Compiler/Visual C++ was unable to open a temporary file. The probable cause is either that you do not have write permissions for the directory or that the directory does not exist. The Resource Compiler/Visual C++ attempts to use these files in the directory specified by the **TMP** environment variable or the current directory if none is specified.

Resource Compiler Warning RW4001

Error Message

.EXE processing options (/L /M /P /T /K /E /31 or /30)

You specified EXE processing options, but there was no executable file to process. Don't use these options with a .res file.

Resource Compiler Warning RW4003

Error Message
SHIFT or CONTROL used without VIRTKEY

In an accelerator table resource, SHIFT or CONTROL requires VIRTKEY. Because SHIFT and CONTROL are indicated as flag bits in a VIRTKEY type accelerator, they cannot exist independent from a VIRTKEY.

Resource Compiler Warning RW4004

Error Message
ASCII character not equivalent to virtual key code

A string literal was used for the virtual key code in a VIRTKEY type accelerator.

This warning allows you to continue, but be aware that the accelerator keys generated may not match the string you indicated. (VIRTKEYs use different key codes than ASCII accelerators.)

While string literals are syntactically valid, you can only ensure that you get the accelerator you want by using the **VK_* #define** values in WINDOWS.h.

CVTRES Errors CVT1100 Through CVT4001

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, or type the error number in the **Look for** box in the Index.

CVTRES Fatal Error CVT1100

Error Message
duplicate resource — type:type, name:name, language:language, flags:flags, size:size

The given resource was specified more than once.

You can get this error if the linker is creating a type library and you did not specify /TLBID and a resource in your project already uses 1. In this case, specify /TLBID and specify another number up to 65535.

CVTRES Fatal Error CVT1101

Error Message
cannot open filename for reading

CVTRES could not open and read the file.

To fix by checking the following possible causes

1. The file does not exist. Check the spelling of the filename and path.
2. The file was opened or deleted by another process.
3. No read permission.
4. Not enough space on disk.

CVTRES Fatal Error CVT1102

Error Message
out of memory; size bytes required

There was not enough memory for CVTRES to complete the operation.

CVTRES Fatal Error CVT1103

Error Message
cannot read filename

An unrecoverable error occurred when CVTRES attempted to read the given file.

This error can be caused by file corruption.

CVTRES Fatal Error CVT1104

Error Message
cannot get location in file

CVTRES could not determine the current location in the file.

This error can be caused by file corruption.

CVTRES Fatal Error CVT1105

Error Message
cannot seek in file

CVTRES could not go to a location in the file.

This error can be caused by file corruption.

CVTRES Fatal Error CVT1106

Error Message
cannot write to file

An unrecoverable error occurred when CVTRES attempted to write to the given file.

CVTRES Fatal Error CVT1107

Error Message
filename is corrupt

The given file is not a valid resource file.

CVTRES Fatal Error CVT1108

Error Message
cannot open filename for writing

CVTRES could not open and write to the given file.

To fix by checking the following possible causes

1. The file does not exist. Check the spelling of the filename and path.
2. The file was opened or deleted by another process.
3. No write permission.
4. Not enough space on disk.

CVTRES Warning CVT4001

machine type not specified; assumed *type*

CVTRES did not find a machine specification. It assumed the given machine type. If the default is incorrect, rerun CVTRES using the /MACHINE option.

Project Build Errors and Warnings (PRJxxxx)

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, or type the error number in the **Look for** box in the Index.

Project Build Error PRJ0002

Error Message
error result returned from 'command line'.

A command, ***command line***, which was formed from user input in the **Property Pages** dialog box, returned an error code but no information will appear in the Output window.

The resolution to this error depends on which tool generated the error. For MIDL, you will get an idea of what went wrong if /o (Redirect Output) is defined.

A batch file, such as a custom build step or build event, that is not informative about failure conditions could also be the reason for this error.

To resolve this error, be more informative on error conditions.

Project Build Error PRJ0003

Error Message
Error spawning 'command line'.

A command, ***command line***, which was formed from input in the **Property Pages** dialog box, returned an error code but no information will appear in the Output window.

Possible reasons for this error:

- Low system resources. Close some applications to resolve this.
- Insufficient security privileges. Verify that you have sufficient security privileges.
- The executable paths specified in VC++ Directories do not include the path for the tool that you are attempting to run.
- For makefile projects, you are missing a command to run on either **Build Command Line** or **Rebuild Command Line**.

Project Build Error PRJ0004

Error Message
Could not generate command line for the 'tool' tool.

One or more properties were specified in such a way as to make the syntax of the call to ***tool*** illegal:

- You may have specified badly formed or unknown macros.
- Your computer may be low on free disk space.

You may want to review the settings for the tool by looking at the Command Line property page.

Project Build Error PRJ0005

Error Message
Unable to create a temporary file in directory 'directory'.

The call to create a temporary file failed. Reasons for failure include:

- Ran out of temporary file names.
- The temp directory is read-only.
- There is no temp directory or TMP environment variable.
- Your computer is out of free disk space.

Project Build Error PRJ0006

Error Message
Could not open the temporary file 'file'. Make sure the file exists and that the directory is not write-protected.

Visual C++ could not create a temporary file during the build process. Reasons for this include:

- No temp directory.
- Read-only temp directory.
- Out of disk space.
- The $(IntDir) folder is either read-only or contains temporary files that are read-only.

This error will also occur following error PRJ0007: Could not create output directory 'directory'. Error PRJ0007 means that the $(IntDir) directory could not be created, implying the creation of temporarily files will also fail.

Temp files are created whenever you specify:

- A response file.
- A custom build step.
- A build event.

Project Build Error PRJ0007

Error Message
Could not create output directory 'directory'.

Visual C++ failed to create an output directory. Possible reasons include:

- The directory in which the output directory was to be created is read-only.
- Computer is out of disk space.
- You specified an invalid directory path.
- User permissions are not sufficient to create the directory.

Project Build Error PRJ0008

Error Message
Could not delete file 'file'.

Make sure that the file is not open by another process and is not write-protected.

During a rebuild or clean, Visual C++ deletes all known intermediate and output files for the build, as well as any files that meet the wildcard specifications in the **Extensions to Delete on Clean** property in the General Configuration Settings Property Page.

You will see this error if Visual C++ is not able to delete a file. To resolve the error, make the file and its directory writeable for the user doing the build.

Project Build Error PRJ0009

Error Message
Build log could not be opened for writing.

Make sure that the file is not open by another process and is not write-protected.

After setting the **Build Logging** property to **Yes** and performing a build or rebuild, Visual C++ was unable to open the build log in exclusive mode.

Inspect the **Build Logging** setting by opening the **Options** dialog box (on the **Tools** menu, click **Options** command) and then selecting **VC++ Build** in the **Projects** folder. The build file is called BuildLog.htm and is written to the intermediate directory $(IntDir).

Possible reasons for this error:

- You are running two processes of Visual C++ and trying to build the same configuration of the same project in both simultaneously.
- The build log file is opened in a process that locks the file.
- The user does not have permission to create a file.
- The current user does not have write access to the file BuildLog.htm.

Project Build Error PRJ0010

Error Message
Unable to find the required dependency: file

A file has a specified dependency, but the dependent file cannot be found.

Project Build Error PRJ0011

Error Message
System resource could be critically low. Unable to create an event required to launch a build.

If system resources are low:

- Close some of your running applications.
- Restart Visual Studio to free resources.
- Reboot.

Project Build Error PRJ0012

Error Message
System resource could be critically low. Unable to create a thread required to launch a build.

If system resources are low:

- Close some of your running applications.
- Restart Visual Studio to free resources.
- Reboot.

Project Build Error PRJ0013

Error Message
System resource could be critically low. Unable to create a pipe required to launch a build.

This error indicates that system resources are low. To resolve this error, decrease system resource usage by other processes/applications.

This error can also occur if your security level is insufficient to create pipes (see CreatePipe).

Project Build Error PRJ0014

Error Message
The job object used to control the spawned processes has failed. The build cannot continue.

An error occurred in the development environment.

To resolve this error, close and reload the project. If necessary, exit and restart Visual Studio.

Project Build Error PRJ0015

Error Message
The NULL device is missing from your system. We are unable to launch a build.

This error can be caused by low system resources. Close some applications or reboot.

It can also be caused if you do not have sufficient privileges to the NULL device on the computer.

Project Build Error PRJ0016

Error Message
The user's security settings prevent the process from being created. These settings are required for building.

You are logged in as a user who does not have permissions to create processes using a process. This is a security feature in Windows NT and Windows 2000. You must change the permission levels for this user account, or contact your account administrator.

This error can also occur if the following registry key is set:

\\HKCU\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\RestrictRun

To resolve this error, delete the RestrictRun key. If this registry key is needed, append **vcspawn.exe** to the list of entries in the key.

Another cause for this error is that your Policy Setting does not include VCSpawn.exe under the registry key HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\RestrictRun as an allowed Window program for this user account.

For additional information, see:

- Knowledge Base article 324153, which is available on http://support.microsoft.com/default.aspx?scid=kb;en-us;324153.
- Adhering to System Policy Settings, the section on "Run only allowed Windows applications".

Project Build Error PRJ0017

Error Message
The current working directory is invalid.

The path to the current working directory is longer than _MAXPATH. To resolve this error, do not create your project at such a deep level.

Project Build Warning PRJ0018

Error Message
The following environment variables were not found:

An environment variable is not defined. This error lists the environment variables that were not defined.

See Macros for Custom Build Commands for information on macros in the build process.

Project Build Error PRJ0019

Error Message
A tool returned an error code from

An error level was nonzero for a custom build step or build event.

You will also see PRJ0019 when a tool returned an error code but no error message. This can happen, for example, if you redirect the output of MIDL to NUL.

See Troubleshooting Custom Build Steps and Build Events for more information.

This error can also occur when you are running as a member of the Users group and Administrative access is needed. For more information, see Running as a Member of the Users Group.

Project Build Error PRJ0020

Error Message
Tool 'tool', Property 'property' contains invalid file name 'file'.

The file name, ***file***, specified in the property, ***property***, for the tool, ***tool***, was invalid.

You may have used an unknown or invalid macro.

See Setting Visual C++ Project Properties for information on how to access your project's properties.

Project Build Error PRJ0021

Error Message
Tool 'tool', Property 'property' contains invalid file name.

The file name specified in the property, ***property***, for the tool, ***tool***, was invalid.

You may have used an unknown or invalid macro.

See Setting Visual C++ Project Properties for information on how to access your project's properties.

Project Build Error PRJ0022

Error Message
Unknown Tool, Property 'property' contains invalid file name 'file'.

The file name, ***file***, specified in the property, ***property***, was invalid.

You may have used an unknown or invalid macro.

See Setting Visual C++ Project Properties for information on how to access your project's properties.

Project Build Error PRJ0023

Error Message
Tool 'tool', Unknown Property contains invalid file name 'file'.

The file name, ***file***, specified for the tool, ***tool***, was invalid.

You may have used an unknown or invalid macro.

See Setting Visual C++ Project Properties for information on how to access your project's properties.

Project Build Error PRJ0024

Error Message
Unicode path 'path' could not be translated to user's ANSI code page.

path is the original Unicode version of the path string. This error can occur in cases where there is a Unicode path that cannot be directly translated to ANSI for the current system code page.

This error may occur if you are working with a project that was developed on a system using a code page that is not on your computer.

The resolution for this error is to update the path to use ANSI text or to install the code page on your computer and set it as the system default.

Project Build Error PRJ0025

Error Message
Batch file 'file' contains Unicode contents that could not be translated to user's ANSI code page.

UNICODE contents of file

The project system found Unicode contents in a custom build rule or build event that cannot be translated properly to the user's current ANSI code page.

The resolution for this error is to update the contents of the build rule or build event to use ANSI or to install the code page on your computer and set it as the system default.

For more information on custom build steps and build events, see Understanding Custom Build Steps and Build Events.

Project Build Error PRJ0026

Error Message
Response file 'file' contains Unicode contents that could not be translated to user's ANSI code page.

UNICODE contents of file

The project system found Unicode contents in a response file that cannot be translated properly to the user's current ANSI code page.

The resolution for this error is to update the contents of the response file to use ANSI or to install the code page on your computer and set it as the system default.

Project Build Error PRJ0027

Error Message
Unicode log message 'contents' contains content that could not be translated to the user's ANSI code page.

You will typically only see this warning in conjunction with errors in creating batch and/or response files.

The resolution for this error is to update the contents of the build log to use ANSI or to install the code page on your computer and set it as the system default.

Project Build Error PRJ0028

Error Message
Temporary file 'file' contains Unicode contents that could not be translated to user's ANSI code page.

A value was specified with the /MIDL (Specify MIDL Command Line Options) linker option that could not be resolved by the system code page.

The code page used when you specify the MIDL command (the input code page) must be the same as the system code page.

Project Build Warning PRJ0029

Error Message
The 'Outputs' property for the project-level custom build step is not set. The custom build step will be skipped.

A custom build step was not executed because no output was specified.

To resolve this error, do one the following:

- Exclude the custom build step from the build.
- Add an output.
- Delete the contents of the custom build step's command.

Project Build Error PRJ0030

Error Message
Macro expansion error. Evaluate recursion exceeded 32 levels for $(macro).

This error is caused by recursion in your macros. For example, if you set the **Intermediate Directory** property (see General Property Page (Project)) to $(IntDir), you will have recursion.

To resolve this error, do not define macros or properties in terms of macros they are used to define.

Project Build Error PRJ0031

Error Message
The 'Outputs' property for the custom build step for file 'file' contained 'macro' which evaluates out to 'macro_expansion'.

A custom build step on a file had bad output probably due to a macro evaluation problem. This error could also mean that the path is badly formed, containing characters or combinations of characters that are illegal in a file path.

To resolve this error, fix the macro or fix the path specification. The evaluated path is an absolute path from the project directory.

Project Build Error PRJ0032

Error Message
The 'Outputs' property for the project-level custom build step contained 'macro' which evaluates out to 'macro_expansion'.

A custom build step on a project had bad output probably due to a macro evaluation problem. This error could also mean that the path is badly formed, containing characters or combinations of characters that are illegal in a file path.

To resolve this error, fix the macro or fix the path specification. The evaluated path is an absolute path from the project directory.

Project Build Error PRJ0033

Error Message
The 'Additional Dependencies' property for the custom build step for file 'file' contained 'macro' which evaluates out to 'macro_expansion'.

A custom build step on a file contained an error in its additional dependency probably due to a macro evaluation problem. This error could also mean that the path is badly formed, containing characters or combinations of characters that are illegal in a file path.

To resolve this error, fix the macro or fix the path specification. The evaluated path is an absolute path from the project directory.

Project Build Error PRJ0034

Error Message
The 'Additional Dependencies' property for the project-level custom build step contained 'macro' which evaluates out to 'macro_expansion'.

A custom build step on a project contained an error in its additional dependency probably due to a macro evaluation problem. This error could also mean that the path is badly formed, containing characters or combinations of characters that are illegal in a file path.

To resolve this error, fix the macro or fix the path specification. The evaluated path is an absolute path from the project directory.

Project Build Error PRJ0035

Error Message
XML file 'file' contains Unicode contents that could not be translated to user's ANSI code page.

UNICODE contents of file

file is the XML file created as the command line to the Web Deployment tool.

The project system found Unicode characters in some property on the Web Deployment property page that can't properly be translated to ANSI.

The resolution for this error is to update the contents of the property to use ANSI or to install the code page on your computer and set it as the system default.

Project Build Error PRJ0036

Error Message
The 'Additional Files' property for the Web Deployment Tool contained an invalid entry.

The Additional Files property on the Web Deployment property page contained an error, possibly due to a macro evaluation problem. This error could also mean that the path is badly formed, containing characters or combinations of characters that are illegal in a file path.

To resolve this error, fix the macro or fix the path specification. The evaluated path is an absolute path from the project directory.

This error could also mean that one of the files referenced doesn't exist.

Project Build Error PRJ0038

Error Message
Failed to find the IIS root directory. IIS must be installed in order for the Web Deployment Tool to work.

Web deployment requires IIS on the machine.

Project Build Error PRJ0039

Error Message
Unable to create a temporary file. The ability to do this is required in order for the NMake tool to be able to run.

When building a makefile project (see Creating a Makefile Project), the Visual C++ project system needs to create some temporary files. This error indicates that the project system was unable to create one or more of those files.

The TMP environment variable contains the name of the temp directory.

Possible causes for this error, and suggested resolutions, are listed below:

User doesn't have write access to temp directory

Make sure you have write access to the temp directory.

Too many temp files in temp directory

Other processes may have created temp files, but not deleted them. Delete some or all of these temp files.

No free disk space

Delete some files on your disk or change your temp directory to a disk with free space.

TMP environment variable is bad

It is possible that the TMP environment variable points to a directory that does not exist.

Project Build Error PRJ0040

Error Message
Internal error on build. Cannot continue. Please reload project and try again.

The project system was unable to process your project.

Try reloading the project.

See Getting Help from Microsoft Product Support Services (Visual Studio).

Project Build Warning PRJ0041

Error Message
Cannot find missing dependency 'dependency' for file 'file'. Your project may still build, but may continue to appear out of date until this file is found.

A file (resource file or .idl/.odl file, for example, contained an include statement that the project system could not resolve.

Because the project system does not process preprocessor statements (#if, for example), the offending statement may not actually be part of the build.

To resolve this warning, delete all unnecessary code in .rc files or add placeholder files of the appropriate name.

Project Build Warning PRJ0042

The 'Outputs' property for the custom build step for file ' *file* ' is not set. The custom build step will be skipped.

A custom build step was not executed because no output was specified.

To resolve this error, do one the following:

- Exclude the custom build step from the build.
- Add an output.
- Delete the contents of the custom build step's command.

Project Build Error PRJ0043

Error Message
The 'Outputs' property for custom build rule 'rule' assigned to file 'file' is invalid. The property contained 'string' which evaluates to 'value'.

The Outputs property evaluated to an empty string. Custom build rules need the output of the build action.

Project Build Error PRJ0044

Error Message

The 'Additional Dependencies' property for custom build rule 'rule' assigned to file 'file' is invalid. The property contained 'string' which evaluates to 'value'.

The Outputs property evaluated to an empty string, or to a string that contained invalid characters (any character that could not be in a file or directory name). Custom build rules need the output of the build action.

Project Build Warning PRJ0045

Error Message

The 'Outputs' property for custom build rule 'rule' assigned to file 'file' is not set. The execution of the rule will be skipped.

The Outputs property is not defined for a custom build rule. Custom build rules need the output of the build action.

Project Build Error PRJ0046

Error Message
Could not spawn command line because the one specified was empty.

An empty command line was specified for a makefile configuration; the command line is required.

Project Build Error PRJ0047

Error Message
Could not resume the suspended process. The build has failed.

An error occurred in the development environment.

To resolve this error, close and reload the project. If necessary, exit and restart Visual Studio.

Project Build Error PRJ0048

Error Message
Could not start a build because a build is already in progress.

You cannot begin a build if another build is in progress. Wait for the current build to complete, or cancel the build,

Project Build Error PRJ0050

Error Message
Failed to register output. Please ensure you have the appropriate permissions to modify the registry.

The Visual C++ build system was not able to register the output of the build (dll or .exe). You need to be logged on as an administrator to modify the registry.

If you are building a .dll, you can try to register the .dll manually using regsvr32.exe, this should display information about why the build failed.

If you are not building a .dll, look at the build log for the command that causes an error.

Project Build Error PRJ0051

Error Message
Failed to copy 'file' to the target directory. Please make sure the destination file is not in use.

The build system failed to copy a file.

Ensure that the file is not open on your machine, or in use by another program.

ATL Provider Errors and Warnings

This section is a reference to the errors generated by the ATL provider. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, or type the error number in the **Look for** box in the Index.

ATL Provider Error ATL2004

Error Message
Out of memory processing attributes.

The provider ran out of memory.

ATL Provider Error ATL2007

Error Message
Requires inclusion of header header.h.

You must include the header file named *header*.h.

ATL Provider Error ATL2010

Error Message
Undeclared identifier "id".

The attribute references an undeclared identifier. To resolve the error, ensure that *id* is declared.

The following sample generates ATL2010.

```
// ATL2010.cpp
// compile with: /c
// ATL2010 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atldbcli.h>

int main()
{
   [ db_source("<Connection string to connect to Northwind database>")];
   // Uncomment the next line to correct ATL2010:
   // char szFirstName[21];
   [ db_command("([bindto] szFirstName) select FirstName from Employees")];
}
```

ATL Provider Error ATL2011

Error Message
Only 1 inline unnamed attribute allowed per scope.

The provider has encountered multiple uses of an attribute of which only one is allowed to be unnamed.

Example
The following sample generates ATL2011 because it contains two unnamed db_source attributes and two unnamed db_command attributes.

```
// ATL2011.cpp
// compile with: /c
// ATL2011 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atldbcli.h>

int main()
{
 [ db_source("<Connection string to connect to Northwind database>")];
// to resolve error, change preceding line to
// [ db_source("<Connection string to connect to Northwind database>",name="nwnd")];
 [ db_source("<Connection string to connect to some other database>")];

 char szFirstName[21];
 [ db_command(db_command="([bindto] szFirstName) select FirstName from Employees")];
 char szCustomerID[6];
 [ db_command(db_command="([bindto] szCustomerID) select CustomerID from Customers")];
// to resolve error, change preceding line to
// [ db_command(db_command="([bindto] szCustomerID) select CustomerID from Customers",name=
"custcmd")];
}
```

ATL Provider Error ATL2012

Error Message
Missing attribute db_accessor(num).

The provider encountered an out-of-sequence db_accessor attribute.

Ensure that the arguments to successive **db_accessor** attributes start at 0 and increase by 1.

The following sample generates ATL2012.

```
// ATL2012.cpp
// compile with: /c
// ATL2012 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atldbcli.h>

[ db_command("Select FirstName, LastName from Employees") ]
class CEmployees {
public :
   [ db_accessor(0, TRUE) ];
   [ db_column("1") ]TCHAR m_szFirstName[21];
   [ db_accessor(2, TRUE)];   // to resolve, change to 1
   [db_column("2") ] TCHAR m_szLastName[21];
};
```

ATL Provider Error ATL2013

Error Message
Class class contains a COLUMN_MAP. Attribute cannot be used on the class.

The attribute is incompatible with a **COLUMN_MAP**. Either remove the attribute or remove the **COLUMN_MAP**.

The following sample generates ATL2013.

```
// ATL2013.cpp
// compile with: /c
// ATL2013 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atldbcli.h>

[ db_command("Select FirstName, LastName from Employees") ]
class CEmployees
{
public :
   BEGIN_COLUMN_MAP(CEmployees)
   END_COLUMN_MAP()
// to resolve, remove the next two lines
   [ db_column(1) ] TCHAR m_szFirstName[21];
   [ db_column(2) ] TCHAR m_szLastName[21];
};
```

ATL Provider Error ATL2014

Error Message
Class class contains a PARAM_MAP. Attribute cannot be used on the class.

The attribute is incompatible with a **PARAM_MAP**. Either remove the attribute or remove the **PARAM_MAP**.

The following sample generates ATL2014.

```
// ATL2014.cpp
// compile with: /c
// ATL2014 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atldbcli.h>

[ db_command("Select FirstName, LastName from Employees") ]
class CEmployees
{
public :
   BEGIN_PARAM_MAP(CEmployees)
   END_PARAM_MAP()
// to resolve, remove the next two lines
   [ db_column(1) ] TCHAR m_szFirstName[21];
   [ db_column(2) ] TCHAR m_szLastName[21];
};
```

ATL Provider Error ATL2015

Error Message

Attribute cannot be used on a class that has constructor and/or destructor.

ATL Provider Error ATL2020

Error Message
Unknown root key format, no acceptable root key found in key.

key contains an invalid root key. Valid root key values are

- HKCR
- HKCU
- HKLM
- HKU
- HKPD
- HKDD
- HKCC
- HKEY_CLASSES_ROOT
- HKEY_CURRENT_USER
- HKEY_LOCAL_MACHINE
- HKEY_USERS
- HKEY_PERFORMANCE_DATA
- HKEY_DYN_DATA
- HKEY_CURRENT_CONFIG

The following sample generates ATL2020.

```
// ATL2020.cpp
// compile with: /c
// ATL2020 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atlcom.h>

class CRDXTest
{
   public :
   [ rdx ("HKLMM", "test", "text") ] TCHAR m_szValue[10];
// to resolve, change HKLMM to HKLM
};
```

ATL Provider Error ATL2021

Error Message
Subkey syntax - (\) not allowed in values.

The provider encountered a value containing a backslash character (\).

The backslash character, which indicates a subkey, is not allowed in values. If you want to specify a subkey, move the text preceding the backslash to the key. For example,

```
[ rdx ("HKLM", "test\\test1", "text") ] TCHAR m_szValue[10];
```

would become

```
[ rdx ("HKLM\\test", "test1", "text") ] TCHAR m_szValue[10];
```

The following sample generates ATL2021.

```
// ATL2021.cpp
// compile with: /c
// ATL2021 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atlcom.h>

class CRDXTest
{
   public :

   [ rdx ("HKLM", "test\\test1", "text") ] TCHAR m_szValue[10];
// change above to
// [ rdx ("HKLM\\test", "test1", "text") ] TCHAR m_szValue[10];
};
```

ATL Provider Error ATL2023

Error Message
Attribute should be used with a real array with storage backing it and not just a simple pointer.

The following sample generates ATL2023.

```
// ATL2023.cpp
// compile with: /c
// ATL2023 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atlcom.h>

class CATL2023
{
   public :
   [ rdx ("HKLM", "test", "text") ] TCHAR *m_szValue;
// to resolve, change the preceding line to
// [ rdx ("HKLM", "test", "text") ] TCHAR m_szValue[100];
};
```

ATL Provider Error ATL2024

Error Message
Attribute should be used with a 4 byte storage data type for a DWORD key.

The attribute was used with a data type too small to hold a DWORD. To resolve, change the data type to a four-byte type (such as DWORD, int, or long).

The following sample generates ATL2024.

```
// ATL2024.cpp
// compile with: /c
// ATL2024 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atlcom.h>

class CRDXTest
{
   public :
   [ rdx ("HKLM", "testdw", "DWORD") ] short m_dwValue;
// to resolve, change short to DWORD
};
```

ATL Provider Error ATL2025

Error Message
Failed to compile RGS. Unknown root key (key) specified.

The root key *key* specified in an RGS file is unknown. Change it to one of the following:

- HKCR
- HKCU
- HKLM
- HKU
- HKPD
- HKDD
- HKCC
- HKEY_CLASSES_ROOT
- HKEY_CURRENT_USER
- HKEY_LOCAL_MACHINE
- HKEY_USERS
- HKEY_PERFORMANCE_DATA
- HKEY_DYN_DATA
- HKEY_CURRENT_CONFIG

The following RGS file generates ATL2025.

```
HKLMM
{
}
```

To resolve the error, change `HKLMM` to `HKLM`.

ATL Provider Error ATL2026

Error Message
Failed to compile RGS. Syntax error: expected a {, found a string.

The RGS file is missing a left brace after the root key.

The following sample RGS file generates ATL2026.

```
HKLM
 subkey = 'test'
}
```

To resolve the error, add a line containing a left brace:

```
HKLM
{
 subkey = 'test'
}
```

ATL Provider Error ATL2027

Error Message
Failed to compile RGS. Out of memory.

The provider ran out of memory while attempting to compile an RGS file.

ATL Provider Error ATL2028

Error Message
Failed to compile RGS. RGS tree contains unparseable constructs.

The provider encountered a syntax error in the RGS file.

The following sample RGS file generates ATL2028.

```
HKLM
{
 subkey = 'test'
}
```

To resolve the error, change the third line to

```
subkey = s 'test'
```

ATL Provider Error ATL2029

Error Message
Failed to compile RGS. Syntax error: [!ifexist] not followed by [!endif]

The RGS file contains a **[!ifexist]** conditional compilation directive that does not have a matching **[!endif]**. Add a **[!endif]** directive at the end of the code you want conditionally compiled.

The following sample RGS file generates ATL2029.

```
HKLM
{
 [!ifexist(noncreatable)]
 val testval = s 'testval'
}
```

To resolve the error, add **[!endif]**:

```
HKLM
{
 [!ifexist(noncreatable)]
 val testval = s 'testval'
 [!endif]
}
```

ATL Provider Error ATL2030

Error Message
Failed to compile RGS. Syntax error: closing ')' not found to match '(' in ifexist block.

The RGS file contains a **[!ifexist]** directive that lacks a closing parenthesis.

The following sample RGS file generates ATL2030.

```
HKLM
{
 [!ifexist(noncreatable]
 val testval = s 'testval'
 [!endif]
}
```

To resolve the error, add a closing parenthesis:

```
 [!ifexist(noncreatable)]
```

ATL Provider Error ATL2031

Error Message
Failed to compile RGS. Syntax error: illegal else without matching if.

The RGS file contains a **[!else]** directive without a corresponding **[!if]**.

The following sample RGS file generates ATL2031.

```
HKLM
{
 [!else]
 val testval = s 'testval'
 [!endif]
}
```

To resolve the error, add a **[!if]** directive:

```
HKLM
{
 [!ifexist(noncreatable]
 [!else]
 val testval = s 'testval'
 [!endif]
}
```

ATL Provider Error ATL2032

Failed to compile RGS. Attribute *attr* specified in the script not found in attribute block.

The RGS file references an attribute that is not in the attribute block.

The following sample files generate ATL2032. First the C++ file:

```
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atlcom.h>

[
 coclass,
 registration_script("atl2032.rgs")
]
class CATL2032
{
};
```

Then the RGS file:

```
HKLM
{
 val testval = s '[!noncreatable]'
}
```

Note that the RGS file references the **noncreatable** attribute, which does not exist in the attribute block. To resolve the error, change **noncreatable** to **coclass**.

ATL Provider Error ATL2033

Error Message
Failed to compile RGS. Value value specified in script not found in attribute block.

The RGS file references a value that is not in the attribute block.

The following sample files generate ATL2033. First the C++ file:

```
// ATL2033.cpp
// compile with: /c
// ATL2033 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atlcom.h>

[ coclass, registration_script("ATL2033.rgs") ]
// to resolve, change the previous line to
// [ coclass, registration_script("ATL2033.rgs"), threading(both) ]
class CATL2033
{
};
```

Then the RGS file:

```
HKLM
{
 val ATL2033 = s '[!threading(threading)]'
}
```

ATL Provider Error ATL2034

Failed to compile RGS. Script file *filename* not found.

The file specified in the registration_script attribute does not exist.

The following sample generates ATL2034. To resolve the error, change `nonexistent.rgs` to the name of an RGS file.

```
// ATL2034.cpp
// compile with: /c
// ATL2034 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atlcom.h>

[
 coclass,
 registration_script("nonexistent.rgs")
]
class CATL2034
{
};
```

ATL Provider Error ATL2050

Error Message
Attribute cannot be specified on a class that indirectly derives from CComCoClass.

The provider has encountered an attribute that is incompatible with a class that indirectly derives from **CComCoClass**.

The following sample generates ATL2050.

```
// ATL2050.cpp
// compile with: /c
// ATL2050 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atlcom.h>

class CATL2050Base : public CComCoClass<CATL2050Base>
{
};

[ coclass ]
class CATL2050 : public CATL2050Base
// to resolve, change the previous line to
// class CATL2050 : public CComCoClass<CATL2050>
{
};
```

ATL Provider Error ATL2054

Error Message
Unknown interface "interface" used with attribute.

The provider has encountered an unknown interface.

The following sample generates ATL2054.

```
// ATL2054.cpp
// compile with: /c
// ATL2054 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atlcom.h>

[ dual ]
__interface IATL2054
{
};

[
   coclass,
   support_error_info(IAtl2054) // should be IATL2054
]
class CATL2054 : public IATL2054
{
};
```

ATL Provider Error ATL2055

Error Message
Interface "interface" does not have a UUID associated with it.

An interface cannot be specified with support_error_info unless it has a UUID associated with it. Some previous versions of MIDL generated header files that define interfaces without associated UUIDs. In those cases, use **__declspec(uuid)** to associate a UUID with an interface. Embedded IDL interfaces will always associate a UUID with the interface.

```
// ATL2055.cpp
// compile with: /c
// ATL2055 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atlcom.h>

struct IATL2055 : public IUnknown
{
};

// to resolve, uncomment the following line
// struct __declspec(uuid("1CA5C7A3-8375-44CA-9605-9DCC1CB8B051")) IATL2055;

[
   coclass,
   support_error_info(IATL2055)
]
class CATL2055 : public IATL2055
{
};
```

ATL Provider Error ATL2056

Error Message
Derive from either class1 or class2, not both.

The class is derived from two incompatible classes. To resolve, remove the derivation from either *class1* or *class2*.

The following sample generates ATL2056.

```
// ATL2056.cp
// compile with: /c
// ATL2056 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atlcom.h>


[ dual ]
__interface IATL2056 : IDispatch
{
   HRESULT Method1();
};

[ coclass ]
class CATL2056 : public IDispatchImpl<IATL2056>
   public IATL2056 // to resolve, comment this line out
{
   public :
   HRESULT Method1()
   {
      return S_OK;
   }
};
```

ATL Provider Error ATL2057

Error Message
Derive from either class or template<class>, not both.

The class is derived from a class and a template that are incompatible. To resolve, remove the derivation from either the class or the template.

The following sample generates ATL2057.

```
// ATL2057.cpp
// compile with: /c
// ATL2057 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atlcom.h>


[ dual ]
__interface IATL2057 : IDispatch
{
   HRESULT Method1();
};

[ coclass ]
class CATL2057 : public IDispatchImpl<IATL2057>
   public IATL2057 // to resolve, comment this line out

{
   public :
   HRESULT Method1()
   {
      return S_OK;
   }
};
```

ATL Provider Error ATL2070

" *function* " cannot be used in attributed project

The global function *function* will be injected automatically. Applies to **DllMain**, **DllRegisterServer**, **DllUnregisterServer**, **DllCanUnloadNow**, **DllGetClassObject**, and **WinMain**.

The following sample generates ATL2070.

```
// ATL2070.cpp
// compile with: /c
// ATL2070 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atlcom.h>

// to resolve, remove the definition of DllMain
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
   return TRUE;
}

[ module (dll, name="ATL2070Lib") ];
```

ATL Provider Error ATL2100

Error Message

Cannot have more than one default counter per perf_object: object already has a default counter

ATL Provider Error ATL2102

Must define either a name_res/help_res or a namestring/helpstring.

ATL Provider Error ATL2103

Error Message

Cannot have a perf_counter without a perf_object class.

ATL Provider Error ATL2104

Error Message

Must define one perf counter type. Use either countertype or countertype_string.

ATL Provider Error ATL2105

Error Message

Invalid Counter Type: "type". Only the following counters are supported through countertype_string: counter, timer, bulk_count, text, rawcount, value, rate, fraction, base, elapsed, queuelen, histogram, precision. For other counters, use the countertype parameter with the perf counter type value.

ATL Provider Error ATL2107

Error Message

Replacement methods that have parameters must use simple types or specify the parse_func parameter to tag_name.

ATL Provider Error ATL2108

Error Message

"sdl" parameter requires "soap_handler" attribute

ATL Provider Error ATL2109

Error Message

Invalid number of indirections. For default parameter handling, one indirection is required for the replacement method.

ATL Provider Error ATL2112

Error Message

Using the tag_name attribute with the parse_func parameter requires that the function take the parsed arguments as a parameter.

ATL Provider Error ATL2200

Error Message
Invalid struct field.

ATL Provider Error ATL2201

Error Message

Indirections not supported on soap headers.

ATL Provider Error ATL2202

Error Message

Out parameters must be pointers.

ATL Provider Error ATL2203

Error Message

soap_handler class must expose at least one method using soap_method.

ATL Provider Error ATL2204

Error Message

Parameters with more than one indirection must have a size_is attribute.

ATL Provider Error ATL2205

Error Message

"method" is not an interface method. SOAP methods must be introduced by an interface.

ATL Provider Error ATL2206

Error Message

"string" is not valid. The string cannot contain '<' or '>'.

ATL Provider Error ATL2207

Error Message

"param" parameter of method "method" has too many indirections. Fixed size arrays cannot have indirections.

ATL Provider Error ATL2209

Error Message

"param" parameter of method "method" has an illegal size_is attribute. The size_is attribute is only allowed on variable-length out arrays.

ATL Provider Error ATL2210

" *param* " parameter of method " *method* " has more than one size_is parameter. Only one-dimensional variable-length arrays are supported.

ATL Provider Error ATL2211

Error Message

"param" parameter of method "method" is missing a size_is attribute. Variable-length out arrays require the size_is attribute.

ATL Provider Error ATL2212

Error Message

The "size" size_is parameter for the "param" parameter of method "method" must be an out parameter.

ATL Provider Error ATL2213

Error Message

"param" parameter of method "method" has too many indirections. In parameters cannot have more than 1 indirection.

ATL Provider Error ATL2214

Error Message

"param" parameter of method "method" has too many indirections. Out parameters cannot have more than 2 indirections.

ATL Provider Error ATL2215

Error Message

"param" parameter of method "method" has an illegal size_is attribute. The size_is attribute is only allowed on variable-length arrays.

ATL Provider Error ATL2216

Error Message

SOAP methods cannot have parameters that are templatized.

ATL Provider Error ATL2217

Error Message

Cannot process "name": SOAP methods that have parameters with base classes are not supported in this version.

ATL Provider Error ATL2218

Error Message

Cannot process "name": SOAP methods cannot have parameters that are templatized.

ATL Provider Error ATL2219

Error Message

Cannot process "struct", failure at field "field": SOAP structs cannot have fields that are templatized.

ATL Provider Error ATL2220

Error Message

Failed processing "string".

ATL Provider Error ATL2221

Error Message

SOAP headers can only be simple types.

ATL Provider Error ATL2222

Error Message

Cannot process "field": struct/class fields cannot have indirections.

ATL Provider Error ATL2223

Error Message

Cannot process: "name": SOAP methods with SAFEARRAY parameters are not supported in this version.

ATL Provider Error ATL2225

Error Message

Cannot process: "union": unions are not supported.

ATL Provider Error ATL2227

Error Message

Cannot process "name" of method "method": soap_header value does not match any member variable.

ATL Provider Error ATL2228

Error Message

Cannot process "name" of method "method": soap_header values must reference public member variables.

ATL Provider Error ATL2229

Error Message

Cannot process: "struct": cannot marshal structs with private members.

ATL Provider Error ATL2230

Error Message

soap_header attribute requires soap_method attribute.

ATL Provider Error ATL2231

Error Message

Cannot have multiple soap_headers with the same name: "name".

ATL Provider Error ATL2232

Error Message

Cannot have multiple "in" soap_headers referencing the same value: "value".

ATL Provider Error ATL2234

Error Message

Ignoring soap_header "header", with in=false and out=false.

ATL Provider Error ATL2235

Error Message

Cannot process soap_header "header" of method "method": fixed-size arrays cannot be used in a SOAP header directly; please wrap the array in a struct.

ATL Provider Error ATL2236

Error Message

Invalid use of retval on method "method": a retval parameter must be the last parameter on the method.

ATL Provider Error ATL2237

Error Message

ATL Server does not support style="style" with use="use".

ATL Provider Error ATL2238

Error Message

Unrecognized style: "style". Legal values are "document" and "rpc".

ATL Provider Error ATL2239

Error Message

Unrecognized use: "use". Legal values are "literal" and "encoded".

ATL Provider Error ATL2240

Error Message

"field" field of struct "struct" has too many indirections. Fixed-size arrays cannot have indirections.

ATL Provider Error ATL2241

Error Message

Cannot process "name": size_is field "size" has invalid type "type".

ATL Provider Error ATL2242

Error Message

Cannot process "name": field "field" has too many indirections.

ATL Provider Error ATL2243

Error Message

Cannot process "name": size_is field "field" does not exist.

ATL Provider Error ATL2244

Error Message

Cannot process "name": field "field" requires size_is attribute.

ATL Provider Error ATL2245

Error Message

Cannot process "name": multidimensional arrays are not supported with style="document" and use="literal"; please use style="rpc" and use="encoded".

ATL Provider Error ATL2246

Error Message

The "size" size_is parameter for the "param" parameter of method "method" must be an in parameter.

ATL Provider Error ATL2247

Error Message

Cannot process "name": size_is parameter "size" does not exist.

ATL Provider Error ATL2248

Error Message

Cannot process "name": SOAP methods cannot have unnamed parameters.

ATL Provider Warning ATL4001

Error Message

Cannot inject full attribute code due to duplicate member "member".

ATL Provider Warning ATL4002

Error Message

Cannot inject full attribute code due to duplicate base class "base".

ATL Provider Warning ATL4003

Cannot inject full attribute code due to duplicate global " *global* ".

ATL Provider Warning ATL4005

Error Message

Attributes are not supported on nested classes.

ATL Provider Warning ATL4051

Error Message
Class "class" derives from CComObjectRoot(Ex). Not injecting base class. Ignoring threading attribute.

The provider would ordinarily inject a base class, but class *class* already derives from the base class. In this situation, the threading attribute, if present, is ignored.

The following sample generates ATL4051.

```
// ATL4051.cpp
// compile with: /c /W3
// ATL4051 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atlcom.h>

[ coclass, uuid("00000000-0000-0000-0000-000000000001") ]
class CATL4051 : public CComObjectRoot
{
};
```

ATL Provider Warning ATL4052

Error Message

Attribute cannot be specified on a class that indirectly derives from CComObjectRoot(Ex).

ATL Provider Warning ATL4053

Error Message
Class "class" derives from CComCoClass. Not injecting base class.

The provider would ordinarily inject a base class, but class *class* already derives from the base class.

The following sample generates ATL4053.

```
/* ATL4053.cpp */
// compile with: /c /WX
// ATL4053 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atlcom.h>

[ coclass, uuid("00000000-0000-0000-0000-000000000001") ]
class CATL4053 : public CComCoClass
{
};
```

ATL Provider Warning ATL4058

Error Message
"interface" is not an interface defined in embedded IDL. It will not be replaced with the corresponding IDispatchImpl.

You may receive this warning in conjunction with a C4199 compiler warning when you build attributed Active Template Library (ATL) code. The error occurs because the interface is not defined in embedded interface definitions in your code and the attribute provider cannot replace the interface with the corresponding **IDispatchImpl**.

If your interface is a dispinterface or a dual interface and you are implementing the **IDispatch** methods, you can ignore the warning.

If however your interface is a dispinterface or dual interface and you expect the attribute provider to implement the **IDispatch** methods (either through the use of an **IDispatchImpl** or the compiler to generate the **IDispatch** methods), the warning is letting you know that the attribute provider cannot implement the **IDispatch** methods for you. To fix this, derive your class from **IDispatchImpl** instead of the interface.

The following sample generates ATL4058.

```
// ATL4058.cpp
// compile with: /c /W4 /WX
// ATL4058 expected
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atlcom.h>
#include <mshtml.h>

[ coclass ]
class ATL4058 : public IHTMLFiltersCollection
{
public :
    virtual HRESULT STDMETHODCALLTYPE get_length(long *p)
    {
       return E_NOTIMPL;
    }
    virtual HRESULT STDMETHODCALLTYPE get__newEnum(IUnknown **p)
    {
       return E_NOTIMPL;
    }

    virtual HRESULT STDMETHODCALLTYPE item(
      VARIANT *pvarIndex,
      VARIANT *pvarResult)
    {
       return E_NOTIMPL;
    }
};
```

ATL Provider Warning ATL4101

Error Message
Creating multiple handler entries for request_handler class: class.

The provider has encountered multiple request_handler attributes on class *class* and will create multiple handler entries.

The following sample generates ATL4101.

```
// ATL4101.cpp
// compile with: /c /W3
// ATL4101 expected
#define _ATL_ATTRIBUTES
#define _WIN32_WINNT 0x0500

#include <atlstencil.h>

[ request_handler("Default"), request_handler("Default2") ]
class CTest
{
};
```

ATL Provider Warning ATL4105

Error Message

Class "class" already derives from IRequestHandler. Not injecting base class.

ATL Provider Warning ATL4106

Error Message

Class "class" already has a tag named "tag".

ATL Provider Warning ATL4110

Error Message

Creating default handler name "name".

ATL Provider Warning ATL4111

Error Message

Using the tag_name attribute on a class that does not have the request_handler attribute or does not derive from CRequestHandlerT will add extra code to your class instance.

SPROXY Errors and Warnings

This section is a reference of error and warning messages generated by SPROXY.EXE.

SPROXY Error SDL0000

Error Message
Internal SPROXY Error (most likely out of memory). Please contact technical support.

This error is triggered by some kind of fatal situation internal to SPROXY.

See Also
Reference
SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1000

Error Message
message

A general error for reporting unexpected failures and out-of-memory conditions. The text in *message* describes the situation.

See Also

Reference
SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1002

Error Message
failure in generating output file: "outputfile"

This error is always emitted when SPROXY fails while generating the output file (for example, when the file named *outputfile* cannot be opened for writing). It is always the last error emitted and is for general reporting purposes.

See Also

Reference
SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1001

Error Message
failure in processing WSDL file: "wsdl_location"

This error is always emitted when SPROXY fails while processing the WSDL (that is, when the WSDL is invalid). It is always the last error emitted and is for general reporting purposes.

See Also

Reference
SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1003

Error Message
ISE : Internal Sproxy Error. Please contact technical support.

Like SDL0000, except this only fires in unexpected situations, not on crashes.

See Also
Reference
SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1004

Error Message

parse error SDL1004 : message

This error is triggered on invalid XML. *message* describes the invalid XML.

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1005

Error Message

unrecognized tag, with

[

namespace = " *namespace* "

tag = " *tag_name* "

]

SPROXY has encountered a tag that it does not recognize.

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1006

Error Message
missing required attribute, with

[

namespace = " *namespace* "

attribute = " *tag_name* "

]

SPROXY has encountered an element missing a required attribute.

See Also

Reference
SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1007

Error Message

MSXML error. Please check your installation of MSXML.

SPROXY has encountered an unexpected MSXML error.

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1008

Error Message

unrecognized document type: "file_or_url". Unknown root node, with

[

namespace = " *namespace* "

name = " *name* "

]

SPROXY has encountered an unknown document type while parsing an imported file.

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1009

Error Message
could not resolve element, with

[

namespace = " *namespace* "

name = " *name* "

]

SPROXY has encountered a reference to an element it cannot find in the WSDL document.

See Also

Reference
SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1010

Error Message
could not resolve namespace : "namespace"

SPROXY has encountered a namespace qualifier that is not associated with any namespace in the document.

See Also
Reference
SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1012

invalid " *attribute* " value : " *value* "

SPROXY has encountered an invalid value for an attribute.

Example

The following example generates SDL1012 because the attribute `xsi:nillable` requires a **boolean** value but is given a non-**boolean** value:

```
<element name="ElementName" xsi:nillable="74"/>
```

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1013

Error Message

could not resolve element_name, with

[

namespace = " *namespace* "

name = " *name* "

]

SPROXY has encountered a reference to an element it cannot find in the WSDL document.

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1014

recursively defined types are not supported, with

[

namespace = " *namespace* "

name = " *name* "

]

SPROXY has encountered a reference to an element that is recursively defined (that is, it contains an instance of itself as a child element).

Example

In the following example, `Buddy` contains an instance of `Buddy` as an element. This triggers error SDL1014.

```
<s:complexType name="Buddy">
   <s:sequence>
      <s:element minOccurs="1" maxOccurs="1" name="Name" type="s:string" />
      <s:element minOccurs="1" maxOccurs="1" name="BestFriend" type="s0:Buddy" />
   </s:sequence>
</s:complexType>
```

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error D1015

Error Message
missing required parameter: parameter

This error is triggered when a required command line parameter is omitted from the SPROXY command line.

See Also

Reference
SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1016

Error Message
could not open WSDL location: "wsdl_location"

SPROXY was unable to open the WSDL location specified on the command line.

See Also
Reference
SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1017

could not find message part " *message_part* " in message with

[

namespace = " *namespace* "

name = " *name* "

]

SPROXY has encountered a reference to a message part that does not exist.

Example

In the following example, message `RetValFrom1HeaderHeader1` contains message part `Header1`, but the code contains a reference to `Header7`, which does not exist. This triggers error SDL1017.

```
      <message name="RetValFrom1HeaderHeader1">
    <part name="Header1" type="s0:Header1" />
  </message>
  <portType name="ValFrom1HeaderSoap">
    <operation name="RetValFrom1Header">
      <input message="tns:RetValFrom1HeaderSoapIn" />
      <output message="tns:RetValFrom1HeaderSoapOut" />
    </operation>
  </portType>
<binding name="ValFrom1HeaderSoap" type="tns:ValFrom1HeaderSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc" />
    <operation name="RetValFrom1Header">
      <soap:operation soapAction="http://tempuri.org/RetValFrom1Header" style="rpc" />
      <input>
        <soap:body use="encoded" namespace="http://tempuri.org/" encodingStyle="http://sche
mas.xmlsoap.org/soap/encoding/" />
        <soap:header n1:required="true" message="tns:RetValFrom1HeaderHeader1" part="Header
7" use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:n1="http:/
/schemas.xmlsoap.org/wsdl/" />
      </input>
      <output>
        <soap:body use="encoded" namespace="http://tempuri.org/" encodingStyle="http://sche
mas.xmlsoap.org/soap/encoding/" />
      </output>
    </operation>
  </binding>
```

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1018

cannot have multiple "in" headers with the same name, with

[

namespace = " *namespace* "

name = " *name* "

]

SPROXY has encountered an operation with multiple `input` headers that have the same name.

Example

In the following example, the operation `RetValFrom1HeaderHeader1` has two `input` headers named `Header1` (shown in bold). This triggers SDL1018.

```
<binding name="ValFrom1HeaderSoap" type="tns:ValFrom1HeaderSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc" />
    <operation name="RetValFrom1Header">
      <soap:operation soapAction="http://tempuri.org/RetValFrom1Header" style="rpc" />
      <input>
        <soap:body use="encoded" namespace="http://tempuri.org/" encodingStyle="http://sche
mas.xmlsoap.org/soap/encoding/" />
        <soap:header n1:required="true" message="tns:RetValFrom1HeaderHeader1" part="Header
1" use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:n1="http:/
/schemas.xmlsoap.org/wsdl/" />
      <soap:header n1:required="true" message="tns:RetValFrom1HeaderHeader1" part="Header1"
use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:n1="http://sc
hemas.xmlsoap.org/wsdl/" />
      </input>
      <output>
        <soap:body use="encoded" namespace="http://tempuri.org/" encodingStyle="http://sche
mas.xmlsoap.org/soap/encoding/" />
      </output>
    </operation>
  </binding>
```

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1019

" *schema_element* " tags at the schema level must have a name attribute.

SPROXY has encountered an unnamed `complexType`, `simpleType`, or `element` element at the schema level. Such elements must have a `name` attribute.

Example

In the following example, the `complexType` element at the schema level does not have a name. This triggers SDL1019.

```
<s:schema targetNamespace="http://tempuri.org/encodedTypes">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="1" maxOccurs="1" name="StrVal" type="s:string" />
      <s:element minOccurs="1" maxOccurs="1" name="iVal" type="s:int" />
    </s:sequence>
  </s:complexType>
</s:schema>
```

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1020

Error Message
document/literal can only reference one message part that has the 'type' attribute.

SPROXY has encountered an operation that meets these conditions:

- `style="document"`
- `use="literal"`
- has more than one message part in a message that uses the `type` attribute

See Also
Reference
SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1021

document/literal cannot reference message parts with the 'type' attribute where the referenced type is not a complexType, it results in an invalid SOAP message.

A message part has a `type` that results in an invalid SOAP message.

SPROXY has encountered an operation that meets these conditions:

- `style="document"`
- `use="literal"`
- has a message part with a `type` attribute that is not a `complexType`

In this situation, the resulting SOAP `<Body>` element lacks required wrapper elements and is therefore invalid.

Example

The following example generates SDL1021. Because the type `xsd:int` is not a `complexType`, the resulting SOAP message would have the form

`<Envelope><Body>` *someData* `</Body></Envelope>`

which is invalid according the SOAP Envelope schema. (There should be wrapper elements around *someData*.)

```
<message name="XSDTypes.Mint">
      <part name="v" type="xsd:int"/>
   </message>
   <message name="XSDTypes.MintResponse">
      <part name="Result" type="xsd:int"/>
   </message>
   <portType name="XSDTypesSoapPort">
      <operation name="Mint" parameterOrder="v">
         <input message="wsdlns:XSDTypes.Mint"/>
         <output message="wsdlns:XSDTypes.MintResponse"/>
      </operation>
   </portType>
   <binding name="XSDTypesSoapBinding" type="wsdlns:XSDTypesSoapPort">
      <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="Mint">
         <soap:operation soapAction="http://tempuri.org/action/XSDTypes.Mint"/>
         <input>
            <soap:body use="literal" namespace="http://tempuri.org/message/" encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"/>
         </input>
         <output>
            <soap:body use="literal" namespace="http://tempuri.org/message/" encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"/>
         </output>
      </operation>
   </binding>
```

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1022

Error Message
rpc/encoded messages require the 'type' attribute.

SPROXY has encountered an operation that meets these conditions:

- `style="rpc"`
- `use="encoded"`
- references a message part without a `type` attribute

According to the WSDL specification, rpc/encoded operations require the `type` attribute on any referenced message parts.

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1023

Error Message
document/encoded is not supported by sproxy.exe.

SPROXY has encountered an operation that meets these conditions:

- `style="document"`
- `use="encoded"`

This combination is not supported by SPROXY.

See Also

Reference
SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1024

unsupported encodingStyle : " *encoding_style* ".

SPROXY has encountered an operation that meets these conditions:

- `style="rpc"`
- `use="encoded"`
- has an `input` or `output` element that has a `<soap:body>` or `<soap:header>` element with an invalid `encodingStyle` attribute

In this situation, the only valid `encodingStyle` attribute value is `"http://schemas.xmlsoap.org/soap/encoding/"`.

Example

In the following example, the `<soap:body>` in bold has an invalid `encodingStyle` ("…`encodingx/`" instead of "…`encoding/`"). This will trigger SDL1024.

```
<binding name="ValFrom1HeaderSoap" type="tns:ValFrom1HeaderSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc" />
    <operation name="RetValFrom1Header">
      <soap:operation soapAction="http://tempuri.org/RetValFrom1Header" style="rpc" />
      <input>
        <soap:body use="encoded" namespace="http://tempuri.org/" encodingStyle="http://sche
mas.xmlsoap.org/soap/encodingx/" />
        <soap:header n1:required="true" message="tns:RetValFrom1HeaderHeader1" part="Header
1" use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:n1="http:/
/schemas.xmlsoap.org/wsdl/" />
      </input>
      <output>
        <soap:body use="encoded" namespace="http://tempuri.org/" encodingStyle="http://sche
mas.xmlsoap.org/soap/encoding/" />
      </output>
    </operation>
  </binding>
```

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1025

input/output operations with different namespaces are not supported.

SPROXY has encountered an `input` operation and an `output` operation that specify different values for the `namespace` attribute. The operations must have the same namespace.

Example

In the following example, the `input` operation specifies the namespace `"http://tempuri.org"` while the `output` specifies the namespace `"http://tempuri.org2"`. This will trigger SDL1025.

```
<binding name="ValFrom1HeaderSoap" type="tns:ValFrom1HeaderSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc" />
    <operation name="RetValFrom1Header">
      <soap:operation soapAction="http://tempuri.org/RetValFrom1Header" style="rpc" />
      <input>
        <soap:body use="encoded" namespace="http://tempuri.org/" encodingStyle="http://sche
mas.xmlsoap.org/soap/encoding/" />
        <soap:header n1:required="true" message="tns:RetValFrom1HeaderHeader1" part="Header
1" use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:n1="http:/
/schemas.xmlsoap.org/wsdl/" />
      </input>
      <output>
        <soap:body use="encoded" namespace="http://tempuri.org2/" encodingStyle="http://sch
emas.xmlsoap.org/soap/encoding/" />
      </output>
    </operation>
  </binding>
```

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1026

Error Message
rpc/literal is not supported by sproxy.exe.

SPROXY has encountered an operation that meets these conditions:

- `style="rpc"`
- `use="literal"`

This combination is not supported by SPROXY.

See Also

Reference
SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1027

headers with different namespaces than their parent operations are not supported.

SPROXY has encountered a `<soap:header>` with a different namespace than that specified in its parent operation. The namespaces must be the same.

Example

In the following example, the `<soap:header>` and its parent `input` operation specify different namespaces (in bold). This will trigger SDL1027.

```
<binding name="ValFrom1HeaderSoap" type="tns:ValFrom1HeaderSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc" />
    <operation name="RetValFrom1Header">
      <soap:operation soapAction="http://tempuri.org/RetValFrom1Header" style="rpc" />
      <input>
        <soap:body use="encoded" namespace="http://tempuri.org/" encodingStyle="http://sche
mas.xmlsoap.org/soap/encoding/" />
        <soap:header n1:required="true" message="tns:RetValFrom1HeaderHeader1" part="Header
1" use="encoded" namespace="http://tempuri.org2/" encodingStyle="http://schemas.xmlsoap.org
/soap/encoding/" xmlns:n1="http://schemas.xmlsoap.org/wsdl/" />
      </input>
      <output>
        <soap:body use="encoded" namespace="http://tempuri.org/" encodingStyle="http://sche
mas.xmlsoap.org/soap/encoding/" />
      </output>
    </operation>
  </binding>
```

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1028

invalid array description.

SPROXY has encountered an array description that is invalid or unsupported.

Example

The following example will trigger SDL1028 because `"s0:TestStruct[,]"` is an unsupported array format.

```
<s:complexType name="MDArrayInTest3_arr_Array">
<s:complexContent>
      <s:restriction base="soapenc:Array">
         <s:attribute ref="soapenc:arrayType" wsdl:arrayType="s0:TestStruct[,]"/>
      </s:restriction>
   </s:complexContent>
</s:complexType>
```

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1029

Error Message
no web service methods to generate.

There are no web service methods for SPROXY to generate. This can occur when the WSDL does not contain any SOAP ports or SOAP bindings (for example, when there are only httppost or httpget bindings).

See Also

Reference
SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Error SDL1030

sproxy.exe does not support extension of complexType

SPROXY does not support `complexType`s with extension--that is, complex types that are *derived* and extend their base type.

The following XML will trigger the error because of `<s:extension base="s0:BaseObject">` which is unsupported.

```
<s:complexType name="Vehicle">
        <s:complexContent mixed="false">
          <s:extension base="s0:BaseObject">
            <s:sequence>
              <s:element minOccurs="0" maxOccurs="1" name="Make" type="s:string" />
              <s:element minOccurs="0" maxOccurs="1" name="Model" type="s:string" />
            </s:sequence>
          </s:extension>
        </s:complexContent>
      </s:complexType>
  <s:complexType name="BaseObject" />
```

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Warning SDL4000

Error Message

parse warning SDL4000 : message

The XML source triggered an XML parse warning. *message* describes the problem.

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Warning SDL4001

only one SOAP port will be processed.

SPROXY has encountered a WSDL `"service"` tag that contains multiple `port` elements.

Example

The following example WSDL will trigger SDL4001. The second and third ports (`"AStructureHttpGet"` and `"AStructureHttpPost"`) will not be processed.

```
<service name="AStructure">
  <port name="AStructureSoap" binding="tns:AStructureSoap">
    <soap:address location="http://someurl" />
  </port>
  <port name="AStructureHttpGet" binding="tns:AStructureHttpGet">
    <http:address location="http://someurl" />
  </port>
  <port name="AStructureHttpPost" binding="tns:AStructureHttpPost">
    <http:address location="http://someurl" />
  </port>
</service>
```

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Warning SDL4002

Error Message
only SOAP ports are supported.

SPROXY has encountered a WSDL `service` tag that does not begin with or contain a SOAP `port` element.

Example
The following example WSDL will trigger SDL4002 because the http port is not supported.

```
<service name="AStructure">
  <port name="AStructureHttpGet" binding="tns:AStructureHttpGet">
    <http:address location="http://someurl" />
  </port>
</service>
```

See Also
Reference
SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Warning SDL4003

Error Message
only SOAP bindings are supported.

SPROXY has encountered a WSDL `port` element under the `service` tag that references a binding that is not a SOAP binding.

Example
The following example WSDL will trigger SDL4003 because the `HttpPost` binding `"AStructureHttpPost"` is not supported.

```
<binding name="AStructureHttpPost" type="tns:AStructureHttpPost">
  <http:binding verb="POST" />
</binding>
<service name="AStructure">
  <port name="AStructureSoap" binding="tns:AStructureHttpGet">
    <soap:address location="http://someurl" />
  </port>
</service>
```

See Also
Reference
SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Warning SDL4004

skipping currently unsupported element, with

[

namespace = " *namespace* "

name = " *element_name* "

]

SPROXY has encountered a known unsupported XML element.

Example

The following example WSDL will trigger multiple SDL4004 warnings indicating that the `urlEncoded`, `mimeXml`, and `content` elements are not supported:

```
<binding name="BinaryTestHttpPost" xmlns:q16="http://tempuri.org/" type="q16:BinaryTestHttp
Post">
  <binding verb="POST" xmlns="http://schemas.xmlsoap.org/wsdl/http/" />
  <operation name="RetBinary">
    <operation location="/RetBinary" xmlns="http://schemas.xmlsoap.org/wsdl/http/" />
    <input>
      <content type="application/x-www-form-urlencoded" xmlns="http://schemas.xmlsoap.org/w
sdl/mime/" />
    </input>
    <output>
      <mimeXml part="Body" xmlns="http://schemas.xmlsoap.org/wsdl/mime/" />
    </output>
  </operation>
</binding>
```

SPROXY Output

```
warn_4004.wsdl(104,69) : warning SDL4004 : skipping currently unsupported element, with
  [
    namespace = "http://schemas.xmlsoap.org/wsdl/http/"
    name = "urlEncoded"
  ]

warn_4004.wsdl(107,78) : warning SDL4004 : skipping currently unsupported element, with
  [
    namespace = "http://schemas.xmlsoap.org/wsdl/mime/"
    name = "mimeXml"
  ]

warn_4004.wsdl(116,107) : warning SDL4004 : skipping currently unsupported element, with
  [
    namespace = "http://schemas.xmlsoap.org/wsdl/mime/"
    name = "content"
  ]

warn_4004.wsdl(119,78) : warning SDL4004 : skipping currently unsupported element, with
  [
    namespace = "http://schemas.xmlsoap.org/wsdl/mime/"
    name = "mimeXml"
  ]
```

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Warning SDL4005

treating as string, with

[

namespace = " *namespace* "

tag = " *tag_name* "

]

SPROXY has encountered an XML Schema element that it does not support. SPROXY will treat the unsupported element as a string.

Example

The following schema fragment will trigger SDL4005 because the `choice` element is not supported:

```
<xsd:complexType name="MyName">
  <xsd:choice>
  </xsd:choice>
</xsd:complexType>
```

SPROXY Output

```
warn_4005.wsdl(6,19) : warning SDL4005 : treating as string, with
  [
    namespace = "http://www.w3.org/2000/10/XMLSchema"
    tag = "choice"
  ]
```

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Warning D4006

ignoring unknown parameter: / *x*

This command-line warning is triggered when SPROXY encounters a command-line parameter */x* that it does not recognize. For example:

```
C:\VC7Libs\Dev\Nonship\Src\SPROXY>sproxy /test
Microsoft (R) Native Web Service Proxy Generator 7.00.9128
Copyright (C) Microsoft Corporation 1999-2001. All rights reserved.

sproxy : command line warning D4006 : ignoring unknown parameter: /test
```

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Warning SDL4007

Error Message
invalid array description, treating as string.

SPROXY has encountered an unsupported array description, such as that for a .NET Framework jagged array. SPROXY treats the array description as a string.

See Also

Reference
SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Warning SDL4008

Error Message

skipping unrecognized extensibility element, with

[

namespace = " *namespace* "

tag = " *tag_name* "

]

SPROXY has encountered an unrecognized WSDL extensibility element. SPROXY ignores the element.

See Also

Reference

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY Warning SDL4009

Error Message
treating custom type as string, with

[

namespace = " *namespace* "

tag = " *tag_name* "

]

SPROXY has encountered a .NET Framework-specific type, such as **Guid** or **Char**. SPROXY will treat those types as strings.

See Also

Reference
SPROXY.EXE: XML Web Service Proxy Generator

Web Deployment Errors and Warnings VCD0001 Through VCD0048

Error Message

This section is a reference to the errors generated by the build tools. To get help on a particular error message, either click an error number in the Output window and press F1, or type the error number in the Look for box in the Index.

Web Deployment Error VCD0001

Error Message
Failed to initialize COM.

See Also

Tasks
Deploying ATL Server Applications
Reference
Web Deployment Property Page

Web Deployment Error VCD0005

Error Message

An unexpected error occurred.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0006

Error Message

Failed to load MSXML. Make sure MSXML is installed on this machine.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0007

Error Message

MSXML could not load the specified settings file. Check that the XML syntax in the settings file is correct.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0008

Error Message

The deployment settings are corrupt.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0009

Error Message

The web server name was not specified. This information is required

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0010

Error Message

Virtual Directory Name was not specified. This information is required.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0011

Error Message

Virtual Directory Name was empty. This information is required.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Warning VCD0012

Error Message

Register Output was empty. Must be the string 'true' or 'false'.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Warning VCD0013

Error Message

Register Output was invalid. Must be the string 'true' or 'false'.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Warning VCD0014

Unload Before Copy was empty. Must be the string 'true' or 'false'.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Warning VCD0015

Error Message

Unload Before Copy was invalid. Must be the string 'true' or 'false'.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Warning VCD0016

Error Message

Application Protection was empty. Must be 0, 1, or 2.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Warning VCD0017

Error Message

Application Protection was invalid. Must be 0, 1, or 2.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0018

Error Message

Virtual Directory Path was not specified. This information is required.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0019

Error Message

Virtual Directory Path was empty. This information is required.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0020

Error Message

Failed to connect to the IIS admin objects. Make sure that IIS is on the local and target machines.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0021

Error Message

Failed to create new virtual directory in the IIS metabase.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0022

Error Message

Failed to set one or more metabase properties for the specified virtual directory.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0023

Error Message

Application Mappings did not include a file extension.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0024

Error Message

Failed to obtain file extension from Application Mappings.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0025

Error Message

Too many verbs listed for Application Mappings.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0026

Error Message

Too many Application Mappings.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0028

Error Message

The W3SVC failed to stop in a timely manner. You may want to run IISRESET.exe on the target machine.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0029

Error Message

The W3SVC failed to restart in a timely manner. You may want to run IISRESET.exe on the target machine.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0030

Error Message

Failed to open the service control manager. You may not have privileges to control services on the target machine.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0031

Error Message

Failed to open the W3SVC. You may not have privileges to control services on the target machine.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0032

Error Message

Failed to send stop command to W3SVC. The service may already be stopped or you may not have privileges to control services on the target machine.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0033

Error Message

Failed to query for W3SVC status. The service might not be running or you might not have privileges to query service status on the target machine.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0034

Error Message

Failed to send the W3SVC a start command. The service may already be started or you may not have privileges to start services on the target machine.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0035

Error Message

Failed to create the file system directory for the virtual directory.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0036

Error Message

Failed to create the file system directory for a directory relative to the virtual directory.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Warning VCD0037

Error Message

Failed to copy a file to the virtual directory.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0038

Error Message

An error occurred registering the ISAPI extension. Make sure the extension is located at the correct path in the virtual directory.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Warning VCD0039

Error Message

Tried to register an ISAPI extension but no virtual directory path was specified.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Warning VCD0040

Error Message

Register Output is only valid for ISAPI extensions. Registration skipped.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0041

Error Message

IIS must be installed on this machine in order for this program to function correctly.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0042

Error Message

An out of memory exception was unexpectedly thrown.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Warning VCD0043

Error Message

The source file file_name does not exist.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Warning VCD0044

Error Message

Error accessing source file. Error number: error_number.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0045

Error Message

Failed to save IIS metabase data. Error number: error_number.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Warning VCD0046

Error Message

An empty WEBHOSTNAME tag was found in the settings document. Host will be ignored.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0047

Error Message

This program requires the caller to be a member of the local Administrators group.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

Web Deployment Error VCD0048

Error Message

An error occurred trying to determine the security context of the user creating this process. Error number: error_number.

See Also

Tasks

Deploying ATL Server Applications

Reference

Web Deployment Property Page

XDCMake Errors and Warnings

This section contains topics describing diagnostics for XDCMake. Click on a link below for more information on a specific error or warning.

XDCMake Error XDC0025

XDCMake Error XDC0026

XDCMake Error XDC0027

XDCMake Error XDC0028

XDCMake Warning XDC0001

XDCMake Warning XDC0002

XDCMake Warning XDC0003

XDCMake Warning XDC0004

XDCMake Warning XDC0006

XDCMake Warning XDC0007

XDCMake Warning XDC0008

XDCMake Warning XDC0009

XDCMake Warning XDC0011

XDCMake Warning XDC0012

XDCMake Warning XDC0014

XDCMake Warning XDC0017

XDCMake Warning XDC0018

XDCMake Warning XDC0020

XDCMake Warning XDC0021

XDCMake Warning XDC0022

XDCMake Warning XDC0023

XDCMake Warning XDC0024

XDCMake Warning XDC0029

XDCMake Warning XDC0030

XDCMake Warning XDC0031

XDCMake Warning XDC0032

XDCMake Warning XDC0001

Error Message
path path does not exist

XDCMake was passed a directory path that was not found.

No .xml file is generated when this diagnostic occurs.

For more information, see XDCMake.

The following sample generates XDC0001:

```
// xdc0001.xdc
// compile with: c:\baddir1\baddir2\xdc0001.xdc
// XDC0001 expected
<?xml version="1.0"?>
<doc>
<members>
<member name="T:MyClass" decl="true" source="c:\test1.cpp" line="6">
   <returns>return value from TEST1.CPP</returns>
</member>
</members>
</doc>
```

XDCMake Warning XDC0002

Error Message
cannot write to file: reason

XDCMake was not able to write to a .xml file. For example, the .xml file may exist, but is read only.

No .xml file is generated when this diagnostic occurs.

For more information, see XDCMake.

The following sample generates XDC0002:

```
// xdc0002.xdc
// compile with: xdc0002.xdc
// post-build command: attrib +R xdc0002.xml
<?xml version="1.0"?>
<doc>
<members>
<member name="T:MyClass" decl="true" source="c:\test1.cpp" line="6">
   <returns>return value from TEST1.CPP</returns>
</member>
</members>
</doc>
```

and then,

```
// xdc0002_b.xdc
// compile with: xdc0002.xdc /out:xdc0002.xml
// XDC0002 expected
<?xml version="1.0"?>
<doc>
<members>
<member name="T:MyClass" decl="true" source="c:\test1.cpp" line="6">
   <returns>return value from TEST1.CPP</returns>
</member>
</members>
</doc>
```

XDCMake Warning XDC0003

Error Message
<include> tags not supported in .XDC files

An <include> tag can be used in a source code file. The <include> tag is processed and expanded by the compiler when generating .xdc files. However, it is not allowed to insert an <include> tag into an .xdc file.

No .xml file is generated when this diagnostic occurs.

For more information, see XDCMake.

The following sample generates XDC0003:

```
// xdc0003.xdc
// compile with: xdc0003.xdc
// XDC0003 expected
<?xml version="1.0"?>
<doc>
<members>
    <member name="T:A" decl="true" source="c:\test.cpp" line="3">
        <summary>Sample XML doc comment</summary>
                <include file="include.me" xpath="*"/>
    </member>
</members>
</doc>
```

XDCMake Warning XDC0004

Error Message
Unable to load XML fragment: reason

A line in an .xdc file was ill formed, possibly due to a manual edit that resulted in a truncated line. If you are unable to manually restore the line, regenerate the .xdc file.

For more information, see XDCMake.

The following sample generates XDC0044

```
// xdc0004.xdc
// compile with: xdc0004.xdc
// XDC0004 expected
<?xml version="1.0"?>
<doc>
<members>
        <member name="T:A" decl="true" source="c:
```

XDCMake Warning XDC0006

Error Message
file(s) not found

xdcmake did not find any .xdc files in the current, or specified, directory.

.xdc files are required input to xdcmake.exe.

For more information, see XDCMake.

XDCMake Warning XDC0007

Error Message
No or multiple output files specified (list), specify exactly one output file

More than one output file was passed to xdcmake. xdcmake.exe requires one and only one output file name, per invocation.

No .xml file is generated when this diagnostic occurs.

For more information, see XDCMake.

The following sample generates XDC0007:

```
// xdc0007.xdc
// compile with: /xdc0007.xdc MyFile1 MyFile2
// XDC0007 expected
<?xml version="1.0"?>
<doc>
<members>
<member name="T:MyClass" decl="true" source="c:\test1.cpp" line="6">
   <returns>return value from TEST1.CPP</returns>
</member>
</members>
</doc>
```

XDCMake Warning XDC0008

Error Message
Replacing existing XML tag content with new: text

Two .xdc file had the information for the same member. However, the first .xdc file processed by xdcmake was for documentation comments on the member declaration (`decl="true"`) and second .xdc file was for documentation comments on the member definition (`decl="false"`). xdcmake will use documentation comments for the definition.

For more information, see XDCMake.

The following sample generates XDC0009:

```
// xdc0008_a.xdc
<?xml version="1.0"?>
<doc>
<members>
<member name="T:MyClass" decl="true" source="c:\test1.cpp" line="6">
   <returns>return value from TEST1.CPP</returns>
</member>
</members>
</doc>
```

and then,

```
// xdc0008_b.xdc
// compile with: xdc0008_b.xdc xdc0008_a.xdc
// XDC0008 expected
<?xml version="1.0"?>
<doc>
<members>
<member name="T:MyClass" decl="false" source="c:\test2.cpp" line="6">
   <returns>return value from TEST2.CPP</returns>
</member>
</members>
</doc>
```

XDCMake Warning XDC0009

Error Message
Replacing existing XML tag attributes with new: text

Two .xdc files had document comments for the same member, but there were different attributes values. xdcmake.exe processed the attribute for the definition (`decl="false"`) of the member, and discarded the attribute for the declaration (`decl="true"`) of the member.

For more information, see XDCMake.

The following sample generates XDC0009:

```
// xdc0009_a.xdc
<doc>
<members>
<member name="T:MyClass" decl="true" source="c:\test1.cpp" line="6">
   <returns bogusattribute="old value">return value from TEST1.CPP</returns>
</member>
</members>
</doc>
```

and then,

```
// xdc0009_b.xdc
// compile with: xdc0009_a.xdc xdc0009_b.xdc
// XDC0009 expected
<?xml version="1.0"?>
<doc>
<members>
<member name="T:MyClass" decl="false" source="c:\test2.cpp" line="6">
   <returns bogusattribute="new value">return value from TEST1.CPP</returns>
</member>
</members>
</doc>
```

XDCMake Warning XDC0011

Error Message
Appending new XML tag content to existing: text

Documentation comments were placed both on the definition (`decl="false"`) and declaration (`decl="true"`) of a member. For some tags, xdcmake.exe will combine the text if multiple tags are found. This warning indicates that the text of the two comments is being merged.

The document comment on the definition is processed into the .xml file before the document comment on the declaration.

For more information, see XDCMake.

The following sample generates XDC0011:

```
// xdc0011_a.xdc
<doc>
<members>
<member name="T:MyClass" decl="true" source="c:\test1.cpp" line="6">
        <summary>This is text in TEST1.CPP</summary>
</member>
</members>
</doc>
```

and then,

```
// xdc0011_b.xdc
// compile with: xdc0011_a.xdc xdc0011_b.xdc
// XDC0011 expected
<?xml version="1.0"?>
<doc>
<members>
<member name="T:MyClass" decl="false" source="c:\test2.cpp" line="6">
        <summary>This is text in TEST2.CPP</summary>
</member>
</members>
</doc>
```

XDCMake Warning XDC0012

Error Message
Creating new tag with same attributes but different content: text

This warning reports that comments for multiple <exception> tags for the same member (in source code) resulted in multiple <exception> tags in the .xml file. If <exception> tags have identical attributes and identical content, no warning is given and no duplicate tags are created.

For more information, see XDCMake.

The following sample generates XDC0012:

```
// xdc0012_a.xdc
<doc>
<members>
<member name="T:MyClass" decl="true" source="c:\test1.cpp" line="6">
   <exception cref="MyException">User-defined exception</exception>
</member>
</members>
</doc>
```

and then,

```
// xdc0012_b.xdc
// compile with: xdc0012_a.xdc xdc0012_b.xdc
// XDC0012 expected
<?xml version="1.0"?>
<doc>
<members>
<member name="T:MyClass" decl="false" source="c:\test2.cpp" line="6">
   <exception cref="MyException">Another exception</exception>
</member>
</members>
</doc>
```

XDCMake Warning XDC0014

Error Message
Multiple definitions for construct, creating new tag with same attributes but different content: content

C++ allows you to declare a symbol in multiple locations, but it must be defined only once. However, multiple definitions can exist in code by the use of conditional statements and symbol definitions.

If different source code files contain definitions for the same member and if those members have document comments, xdcmake will create multiple tags in the generated .xml file, for document comments on all definitions.

XDC0014 is for tags that xdcmake can generate multiple entries for, in the .xml file.

For more information, see XDCMake.

The following sample generates XDC0014:

```
// xdc0014_a.xdc
<?xml version="1.0"?>
<doc>
<members>
<member name="T:MyClass" decl="false" source="c:\test1.cpp" line="6">
   <permission cref='MyPermission'>explains the permission</permission>
</member>
</members>
</doc>
```

and then,

```
// xdc0014_b.xdc
// compile with: xdc0014_a.xdc xdc0014_b.xdc
// XDC0014 expected
<?xml version="1.0"?>
<doc>
<members>
<member name="T:MyClass" decl="false" source="c:\test2.cpp" line="6">
   <permission cref='MyPermission'>explains the permission in another way</permission>
</member>
</members>
</doc>
```

XDCMake Warning XDC0017

Error Message
Unrecognized XDCMake option: option

An option was passed to xdcmake.exe that was not recognized by xdcmake.

No .xml file is generated when this diagnostic occurs.

For more information, see XDCMake.

XDCMake Warning XDC0018

Error Message
tag tag must have content: end_tag

Certain tag types are meaningless unless they contain content.

No .xml file is generated when this diagnostic occurs.

For more information, see XDCMake.

The following sample generates XDC0018:

```
// xdc0018.xdc
// compile with: xdc0018.xdc
// XDC0018 expected
<?xml version="1.0"?>
<doc>
<members>
<member name="T:MyClass" decl="false" source="c:\test1.cpp" line="6">
   <summary/>
</member>
</members>
</doc>
```

XDCMake Warning XDC0020

Error Message
tag tag must have cref attribute: end_tag

A tag (***tag***) was specified that requires a cref attribute. Refer to the documentation for ***tag***, to see how to specify the cref attribute.

No .xml file is generated when this diagnostic occurs.

For more information, see XDCMake.

The following sample generates XDC0020:

```
// xdc0020.xdc
// compile with: xdc0020.xdc
// XDC0020 expected
<?xml version="1.0"?>
<doc>
<members>
<member name="T:MyClass" decl="false" source="c:\test1.cpp" line="6">
   <see/>
</member>
</members>
</doc>
```

XDCMake Warning XDC0021

Error Message
Output xml_file contains no XML document comments

An .xdc file was manually edited to remove all <member> tags. Therefore, the xdcmake-generated .xml file has no content.

For more information, see XDCMake.

The following sample generates XDC0021:

```
// xdc0021.xdc
// compile with: xdc0021.xdc
// XDC0021 expected
<?xml version="1.0"?>
<doc>
<members>
</members>
</doc>
```

XDCMake Warning XDC0022

Error Message
File file contains no <doc><members> tag

The <doc> and <members> tags were not present in a .xdc file.

For example, if a .xdc file is manually edited and the <members> tag is changed to <Memebers>, xdcmake will not recognize the tag.

No .xml file is generated when this diagnostic occurs.

For more information, see XDCMake.

The following sample generates XDC0022:

```
// xdc0022.xdc
// compile with: xdc0022.xdc
// XDC0022 expected
<?xml version="1.0"?>
<doc>
<Members>
<Member name="T:MyClass" decl="false" source="c:\test1.cpp" line="6">
   <value attribute='MyAttr'>explains the property value</value>
</Member>
</Members>
</doc>
```

XDCMake Warning XDC0023

Error Message
discarded <member> tag with missing or empty 'name', 'source', 'line' attribute

A <member> tag was missing information for one of its attributes, so xdcmake discarded the <member> tag.

For more information, see XDCMake.

The following sample generates XDC0023:

```
// xdc0023.xdc
// compile with: xdc0023.xdc
// XDC0023 expected
<?xml version="1.0"?>
<doc>
<members>
<member name="T:MyClass" decl="" source="c:\test1.cpp" line="6">
   <value attribute='MyAttr'>explains the property value</value>
</member>
</members>
</doc>
```

XDCMake Warning XDC0024

Error Message
<member> tag, line attribute not a decimal integer, assuming default value 0

An .xdc file was manually edited and the value of the line attribute was changed to be something other that a base 10 integer.

For more information, see XDCMake.

The following sample generates XDC0024:

```
// xdc0024.xdc
// compile with: xdc0024.xdc
// XDC0024 expected
<?xml version="1.0"?>
<doc>
<members>
   <member name="T:MyClass" decl="true" source="c:\test1.cpp" line="VIII">
   <value attribute='MyAttr'>explains the property value</value>
</member>
</members>
</doc>
```

XDCMake Error XDC0025

Error Message
can't switch to culture culture

For more information, see Comparing and Sorting Data for a Specific Culture and XDCMake Reference.

XDCMake Error XDC0026

Error Message
Too many response files. Is there a recursive reference?

The limit on the number of response files with xdcmake.exe is 40.

XDCMake Error XDC0027

Error Message
Cannot read from file file

XDC0027 is a general read error for a response file. Ensure that you can read the response file.

XDCMake Error XDC0028

Error Message
Paths cannot have wildcards

Wildcard characters are not allowed in path specifications. For example,

```
xdcmake c:\test*\dir*\xdc0028.xdc
```

For more information, see XDCMake Reference.

XDCMake Warning XDC0029

Error Message
No input (*.xdc) files specified

xdcmake.exe was invoked at the command line, no .xdc files were specified, and no .xdc files were present in the current directory.

For more information on how to use xdcmake.exe, see XDCMake Reference.

XDCMake Warning XDC0030

Error Message
/out and /assembly require a value

xdcmake.exe was invoked at the command line with **/out** or **/assembly**, but no value was passed to **/out** or **/assembly**.

For more information on how to use xdcmake.exe, see XDCMake Reference.

XDCMake Warning XDC0031

Error Message
/out and /assembly can be specified at most once.

xdcmake.exe was invoked at the command line with more than one **/out** or **/assembly**. Only one **/out** or one **/assembly** option is allowed.

For more information on how to use xdcmake.exe, see XDCMake Reference.

XDCMake Warning XDC0032

Error Message
the '.xdc' extension must be specified explicitly for input files.

xdcmake.exe only accepts .xdc files as input. If you do not specify any .xdc files when invoking xdcmake.exe, all .xdc files in the current directory are used.

For more information on how to use xdcmake.exe, see XDCMake Reference.

Visual C++ Reference

The following links take you to topics providing reference material for Visual C++.

In This Section

Visual C++ Libraries

Provides links to the various libraries provided with Visual C++, including ATL, ATL Server, MFC, OLE DB, the C run-time library, and the Standard C++ Library.

C++ Attributes Reference

Provides links to topics discussing attributes, which simplify COM programming and .NET Framework common language runtime development.

C/C++ Languages

Provides links to the C language reference, the C++ language reference, and C and C++ preprocessor directives reference.

Microsoft Macro Assembler Reference

Provides links to topics describing the MASM directives, operators, and error messages.

Related Sections

Porting and Upgrading Programs

Provides links to topics describing how to create your projects for portability to other languages and platforms and how to upgrade your projects to conform to the current release of Visual C++.

Creating and Managing Visual C++ Projects

Provides links to topics describing how to use application and code wizards to create your own projects and how to employ property pages to manage your projects.

Building a C/C++ Program

Provides links to topics describing building your program from the command line or from the integrated development environment of Visual Studio.

General Concepts

Provides links to topics describing conceptual information about the Visual C++ libraries and topics discussing various coding technologies and techniques.

Samples

Provides links to sample code showing the capabilities of Visual C++ and the libraries and technologies it supports.

Visual C++

Provides links to different areas of the Visual Studio and Visual C++ documentation set.

Integrated Development Environment for Visual Studio

Provides links to topics describing how you can use Visual Studio to accomplish each step in the development process.

Debugging Native Code

Discusses some common debugging problems and techniques for C and C++ applications.

Visual C++ Extensibility Object Model

Provides links to topics discussing COM objects, interfaces, and associated managed wrappers that can be used with any COM- or .NET-compliant language to automate actions occurring in the Visual Studio development environment.

Visual C++ Libraries Reference

The Visual C++ libraries are documented in the following references.

In This Section

ATL Reference

Provides reference material for the ATL Library, a set of template-based C++ classes that simplify the programming of COM objects.

ATL Server Reference

Provides reference material for the ATL Server Library, a set of native C++ classes that allows you to create Web applications, XML Web services, and other server applications.

MFC Reference

Provides reference material for the MFC Library, a set of classes that constitute an application framework, which is the framework of an application written for the Windows API.

ATL/MFC Shared Classes

Provides reference material for classes shared between ATL and MFC.

OLE DB Templates

Provides reference material for the OLE DB consumer and provider templates, a set of template classes that implement many commonly used OLE DB interfaces.

Run-Time Library Reference

Provides reference material for the C Run-Time Library, a set of routines that automate many common programming tasks that are not provided by the C and C++ languages.

Standard C++ Library Reference

Provides reference material for the Standard C++ Library implementation, a set of header files that provide functions to perform essential services such as input and output and provide efficient implementations of frequently used operations.

Related Sections

.NET Framework General Reference

Provides links to reference documentation for the .NET Framework.

Visual C++ Samples

Provides links to sample code showing the capabilities of Visual C++ and the libraries and technologies it supports.

Debugging in Visual Studio

Provides links to using the Visual Studio debugger to correct logic errors in your application or stored procedures.

Building a C/C++ Program

Provides links to topics describing building your program from the command line or from the integrated development environment of Visual Studio.

Visual C++

Provides links to different areas of the Visual Studio and Visual C++ documentation set.

Visual C++ Libraries as Shared Side-by-Side Assemblies

In Visual C++ 2005, the ATL, MFC, Standard C++, and CRT libraries support the new deployment model available on Windows XP, Windows Server 2003, and Windows Vista. The DLLs corresponding to all Visual C++ libraries have been grouped into several shared side-by-side assemblies and are installed into the native assembly cache, also called the WinSxS folder, under the operating system root directory. Similarly, while building a C++ application using Visual C++ 2005, by default the compiler and the linker generate a manifest file that describes runtime dependencies of this application on Visual C++ libraries.

Visual C++ libraries cannot be used by a C/C++ application without a manifest binding the application to these libraries. If a C/C++ application that depends on a Visual C++ library does not use a manifest, then an attempt to load the Visual C++ library as a dependent DLL from the application-local folder will result in an error message indicating that this is an unsupported way of loading a Visual C++ library.

Note
On versions of Windows that do not support deployment of shared side-by-side assemblies, such as Windows 98 and Windows 2000 Server, the Visual C++ libraries are installed in the System32 folder and WinSxS folder under the operating system root directory. This setup enables running Visual C++ applications on these operating system versions because they do not support manifest-based binding of applications to dependent DLLs. On these operating systems, when an application is loaded, the corresponding manifest file is ignored and the operating systems searches for dependent DLLs using paths set in the current running environment. However, on upgrading the operating system to a version that support manifest-based binding, such as Windows XP, Windows Server 2003, or Windows Vista, applications built with manifests start using the DLLs installed in the WinSxS folder.

This change to the deployment model of Visual C++ libraries prevents the problem of version conflicts between DLLs that occur when you add updates or configurations to a machine, and will allow support of side-by-side installation of two different Visual C++ toolsets. It will also allow you to produce reliable, self-describing applications and components that will not conflict with existing components. For more information on the advantages of new deployment model, please see Concepts of Isolated Applications and Side-by-side Assemblies. To learn about how this may impact deployment of existing native C++ applications, please refer to Redistributing Visual C++ Files.

Visual C++ libraries have been packaged in several shared side-by-side assemblies with corresponding manifest files.

Assembly Name	DLLs included in the assembly	Visual C++ Library
Microsoft.VC80.ATL	atl80.dll	Active Template Library
Microsoft.VC80.CRT	msvcr80.dll msvcp80.dll msvcm80.dll	C Runtime Library, release DLLs
Microsoft.VC80.DebugCRT	msvcr80d.dll msvcp80d.dll msvcm80d.dll	C Runtime Library, debug DLLs
Microsoft.VC80.MFC	mfc80.dll mfcm80.dll mfc80u.dll mfcm80u.dll	Microsoft Foundation Classes, release DLLs

Microsoft.VC80.DebugMFC	mfc80d.dll mfcm80d.dll mfc80ud.dll mfcm80ud.dll	Microsoft Foundation Classes, debug DLLs
Microsoft.VC80.MFCLOC	mfc80chs.dll mfc80deu.dll mfc80esp.dll mfc80ita.dll mfc80kor.dll mfc80cht.dll mfc80enu.dll mfc80fra.dll mfc80jpn.dll	Microsoft Foundation Classes, localized resources
Microsoft.VC80.OpenMP	vcomp.dll	OpenMP Library, release DLLs
Microsoft.VC80.DebugOpenMP	Vcompd.dll	OpenMP Library, debug DLLs

When you build applications with Visual C++ 2005, you may need to redistribute Visual C++ libraries and deploy the corresponding DLLs to customers' desktops. For more information please see Deployment (C++).

ATL Reference

The ATL Reference documents the Active Template Library (ATL), a set of template-based C++ classes that simplify the programming of Component Object Model (COM) objects. To fully take advantage of ATL, a working familiarity with COM is highly recommended.

In This Section

ATL Class Overview

Provides links to and brief descriptions of the ATL classes organized by category.

ATL Classes

Provides reference material on the classes organized alphabetically.

ATL Functions

Provides reference material on the global functions organized alphabetically. Includes topics organizing the functions into categories.

ATL Global Variables

Provides reference material on the global variables organized alphabetically.

ATL Macros

Provides reference material on the macros organized alphabetically. Includes topics organizing the macros into categories.

ATL Structures

Provides reference material on the structures organized alphabetically.

ATL Typedefs

Provides reference material on the typedefs organized alphabetically

Obsolete ATL Topics

Provides reference material on the classes, macros, and functions that are obsolete or deprecated.

Related Sections

ATL

Provides topics on how to program using the Active Template Library (ATL).

ATL Tutorial

Leads you through the creation of a control and demonstrates some ATL fundamentals in the process.

ATL Samples

Sample code that shows how to use ATL to write COM objects.

ATL Server Reference

Provides reference material for the ATL Server Library, a set of template-based C++ classes for creating Web applications, XML Web services, and other server applications.

OLE DB Templates

Provides reference material for the OLE DB consumer and provider templates, a set of template classes that implement many commonly used OLE DB interfaces.

Visual C++ Libraries

Provides links to the various libraries provided with Visual C++, including ATL, MFC, OLE DB Templates, the C run-time library, and the Standard C++ Library.

Breaking Changes (ATL)

Breaking changes in ATL.

Breaking changes

- COleDateTime::operator = breaking change

  In the overload **COleDateTime& operator=(const FILETIME&** *filetimeSrc***)**, the FILETIME value is correctly converted and copied into this **COleDateTime** object. If the conversion is successful, the status of this object is set to valid; if unsuccessful, it is set to invalid. **FILETIME** uses Universal Coordinated Time (UTC), so if you pass a local time in the structure, your results will be converted from local file time to system time to variant time. Note that this corrected behavior is the same as in Visual C++ 6.0 and Visual C++.NET 2003 SP2. See File Times in the *Platform SDK* for more information.

- CComVariant::operator == now does not consider **VT_EMPTY** to be equal to a variant of another type whose value is zero.

See Also

Concepts

What's New in the Visual C++ Libraries

Other Resources

ATL Reference

ATL Concepts

The Active Template Library (ATL) is a set of template-based C++ classes that let you create small, fast Component Object Model (COM) objects. It has special support for key COM features, including stock implementations, dual interfaces, standard COM enumerator interfaces, connection points, tear-off interfaces, and ActiveX controls.

If you do a lot of ATL programming, you will want to learn more about attributes, a new feature in Visual C++ .NET that is designed to simplify COM programming. For more information, see Attributed Programming.

In This Section

ATL Tutorial

Leads you through the creation of a control and demonstrates some ATL fundamentals in the process.

Attributes Tutorial

Leads you through the creation of a client and a server application using attributes and events.

Introduction to COM and ATL

Introduces the major concepts behind the Component Object Model (COM). This article also briefly explains what ATL is and when you should use it.

Fundamentals of ATL COM Objects

Discusses the relationship among various ATL classes and how those classes are implemented.

Dual Interfaces and ATL

Describes dual interfaces from an ATL perspective.

ATL Collections and Enumerators

Describes the implementation and creation of collections and enumerators in ATL.

Composite Control Fundamentals

Provides step-by-step instructions for creating a composite control. A composite control is a type of ActiveX control that can contain other ActiveX controls or Windows controls.

ATL Control Containment FAQ

Covers the fundamental questions related to hosting controls with ATL.

ATL COM Property Pages

Shows you how to specify and implement COM property pages.

ATL Support for DHTML Controls

Provides step-by-step instructions for creating a DHTML control.

ATL Connection Points

Explains what connection points are and how ATL implements them.

Event Handling and ATL

Describes the steps that you need to take to handle COM events using ATL's IDispEventImpl and IDispEventSimpleImpl classes.

ATL and the Free Threaded Marshaler

Provides details on the ATL Simple Object Wizard's option that allows your class to aggregate the free threaded marshaler (FTM).

Specifying the Project's Threading Model

Describes the macros that are available to control run-time performance related to threading in your project.

ATL Module Classes

Discusses the module classes new for ATL 7.0. Module classes implement the basic functionality required by ATL.

ATL Services

Covers the series of events that occur when a service is implemented. Also talks about some of the concepts related to developing a service.

ATL Window Classes

Describes how to create, superclass, and subclass windows in ATL. The ATL window classes are not COM classes.

ATL Collection Classes

Describes how to use arrays and maps in ATL.

The ATL Registry Component (Registrar)

Discusses ATL scripting syntax and replaceable parameters. It also explains how to set up a static link to the Registrar.

Programming with ATL and C Run-Time Code

Discusses the benefits of using the C Run-Time Library (CRT) or alternatively using the **_ATL_MIN_CRT** macro to reduce dependency on CRT code.

Programming with CComBSTR

Discusses several situations that require caution when programming with **CComBSTR**.

Related Sections

ATL Samples

Provides descriptions of and links to the ATL sample programs.

Creating an ATL Project

Contains information on the ATL Project Wizard.

ATL Control Wizard

Discusses how to add classes.

ATL Debugging Techniques

Tells you how to use ATL's built-in support for debugging **QueryInterface**, **AddRef**, and **Release** calls.

Breaking Changes in ATL 7.0 and MFC 7.0 Since Visual C++ 6.0

Lists items to take into account when updating code from previous versions to ATL 7.0.

Attributed Programming

Provides an overview on using attributes to simplify COM programming plus a list of links to more detailed topics.

ATL Class Overview

Provides reference information and links to the ATL classes.

Adding Functionality

Provides links to topics describing conceptual information about Visual C++ libraries and topics discussing various coding technologies and techniques.

ATL Tutorial

ATL is designed to simplify the process of creating efficient, flexible, lightweight controls. This tutorial leads you through the creation of an ActiveX control, demonstrating many ATL and COM fundamentals.

This tutorial does not use attributes. For a step-by-step guide to using attributes, see Attributes Tutorial.

By following this tutorial, you will learn how to add a control to an ATL project that draws a circle and a filled polygon. You will then add a property to indicate how many sides the polygon will have and create drawing code for updating the control when the property changes. The control will then be displayed on a Web page using some VBScript to make it respond to events.

The tutorial is divided into seven steps. You should perform each step in order as later steps depend on previously completed tasks. Before you begin, you should confirm that you have privileges required to register an ActiveX component on your particular computer. This is usually only a concern if you are running Visual Studio .NET over a Terminal Services connection.

- Step 1: Creating the Project
- Step 2: Adding a Control to Your Project
- Step 3. Adding a Property to Your Control
- Step 4: Changing Your Control's Drawing Code
- Step 5: Adding an Event
- Step 6: Adding a Property Page
- Step 7: Putting Your Control on a Web Page

See Also

Reference

Attributes Tutorial

Other Resources

ATL Concepts

Step 1: Creating the Project

This tutorial walks you step-by-step through a nonattributed ATL project that creates an ActiveX object that displays a polygon. The object includes options for allowing the user to change the number of sides making up the polygon, and code to refresh the display.

Note
This tutorial creates the same source code as the Polygon sample. If you want to avoid entering the source code manually, you can download it from the Polygon sample abstract. You can then refer to the Polygon source code as you work through the tutorial, or use it to check for errors in your own project.

To create the initial ATL project using the ATL Project Wizard

1. In the Visual Studio development environment, click **New** on the **File** menu, and then click **Project**.
2. Click the **Visual C++ Projects** folder and select **ATL Project**.
3. Type `Polygon` as the project name:

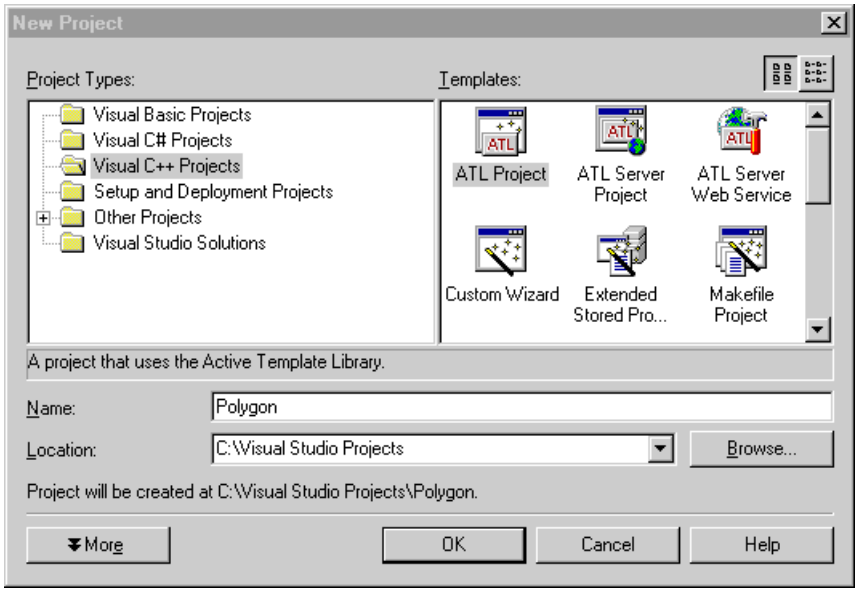

   The location for the source code will usually default to My Documents\Visual Studio Projects, and a new folder will be created automatically.

4. Click **OK** and the ATL Project Wizard opens.
5. Click **Application Settings** to see the options available:

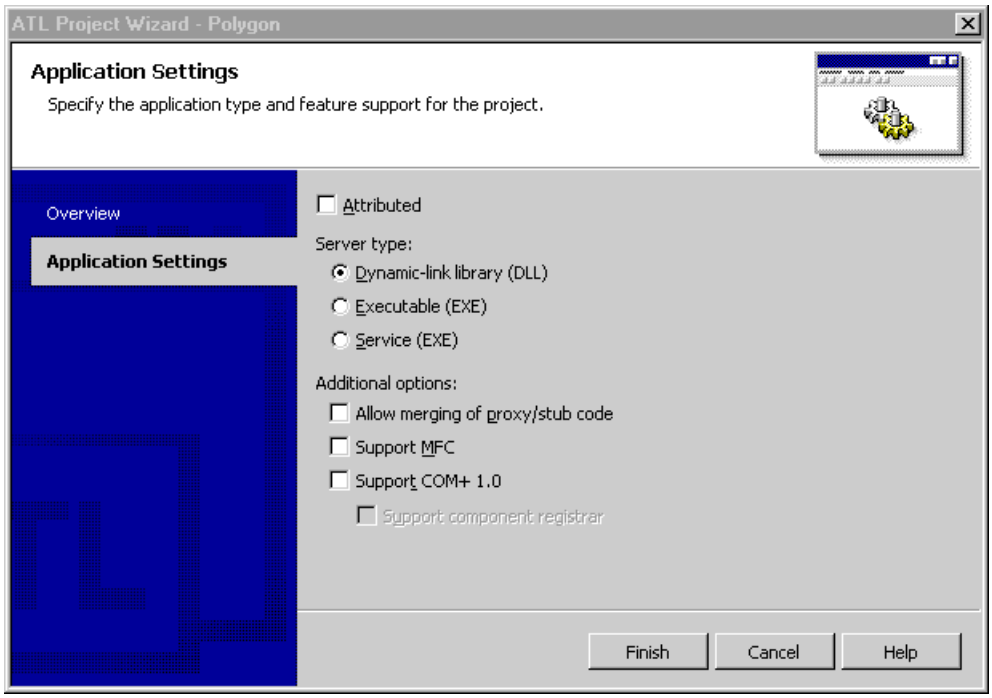

6. As you are creating a control, and a control must be an in-process server, leave the **Server type** as a DLL.
7. In this tutorial, you will not be using attributes, so ensure that the **Attributed** check box is not selected.
8. Leave the other options at their default values, and click **Finish**.

The ATL Project Wizard will create the project by generating several files. You can view these files in Solution Explorer by expanding the Polygon object. The files are listed below.

File	Description
Polygon.cpp	Contains the implementation of **DllMain**, **DllCanUnloadNow**, **DllGetClassObject**, **DllRegisterServer**, and **DllUnregisterServer**. Also contains the object map, which is a list of the ATL objects in your project. This is initially blank.
Polygon.def	This module-definition file provides the linker with information about the exports required by your DLL.
Polygon.idl	The interface definition language file, which describes the interfaces specific to your objects.
Polygon.rgs	This registry script contains information for registering your program's DLL.
Polygon.rc	The resource file, which initially contains the version information and a string containing the project name.
Resource.h	The header file for the resource file.
Polygonps.def	This module definition file provides the linker with information about the exports required by the proxy and stub code that support calls across apartments.
stdafx.cpp	The file that will **#include** the ATL implementation files.
stdafx.h	The file that will **#include** the ATL header files.

In the next step, you will add a control to your project.

On to Step 2

See Also

Reference

ATL Tutorial

Step 2: Adding a Control

In this step, you will add a control to your project, build it, and test it on a Web page.

Procedures

To add an object to an ATL project

1. In Class View, right-click the Polygon project.
2. Point to **Add** on the shortcut menu, and click **Add Class**.

   The **Add Class** dialog box appears. The different object categories are listed in the tree structure on the left:

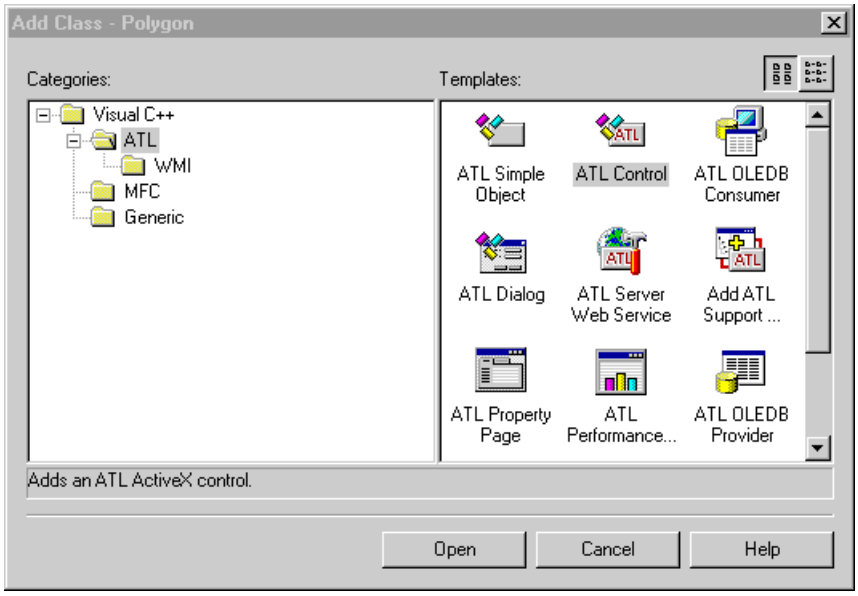

3. Expand the tree structure and click the **ATL** folder.
4. From the list of templates on the right, select **ATL Control**. Click **Open**. The ATL Control Wizard will open, and you can configure the control:

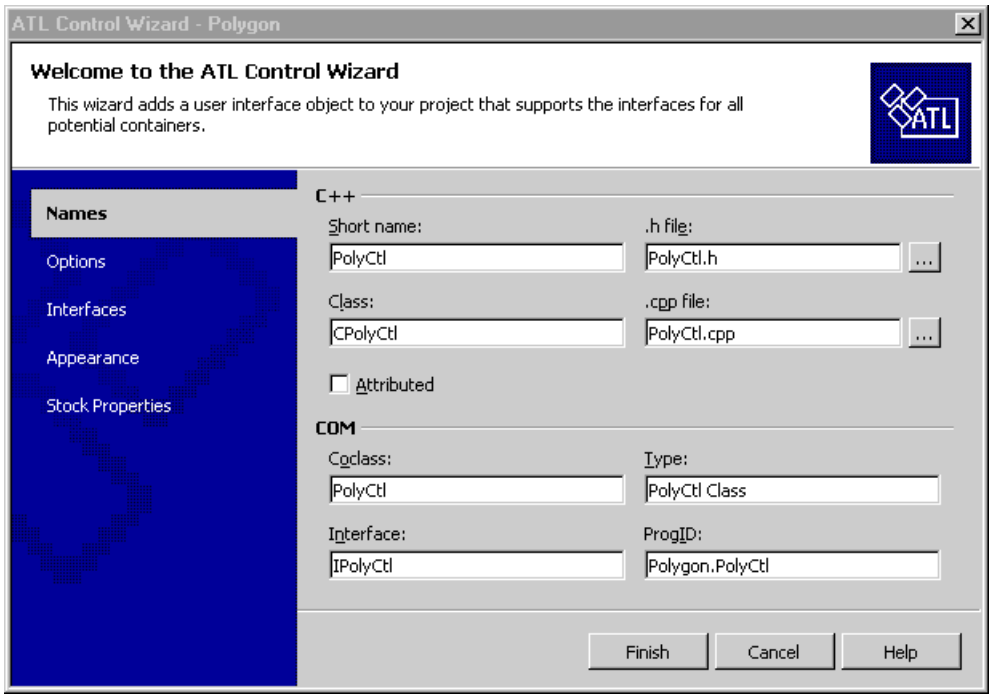

5. Type `PolyCtl` as the short name and note that the other fields are automatically completed. Do not click **Finish** yet, because you need to make some changes.

The ATL Control Wizard's **Names** page contains the following fields:

Field	Contents
Short name	The name you entered for the control.
Class	The C++ class name created to implement the control.
.h file	The file created to contain the definition of the C++ class.
.cpp file	The file created to contain the implementation of the C++ class.
CoClass	The name of the component class for this control.
Interface	The name of the interface on which the control will implement its custom methods and properties.
Type	A description for the control.
ProgID	The readable name that can be used to look up the CLSID of the control.

You need to make several additional settings in the ATL Control Wizard.

To enable support for rich error information and connection points

1. Click **Options** to open the **Options** page.
2. Select the **Connection points** check box. This will create support for an outgoing interface in the IDL file.

You can also make the control insertable, which means it can be embedded into applications that support embedded objects, such as Excel or Word.

To make the control insertable

1. Click **Appearance** to open the **Appearance** page.
2. Select the **Insertable** check box, which by default will be cleared.

The polygon displayed by the object will have a solid fill color, so you need to add a Fill Color stock property.

To add a Fill Color stock property and create the control

1. Click **Stock Properties** to open the **Stock Properties** page.
2. Under **Not supported**, scroll down the list of possible stock properties. Double-click **Fill Color** to move it to the **Supported** list:

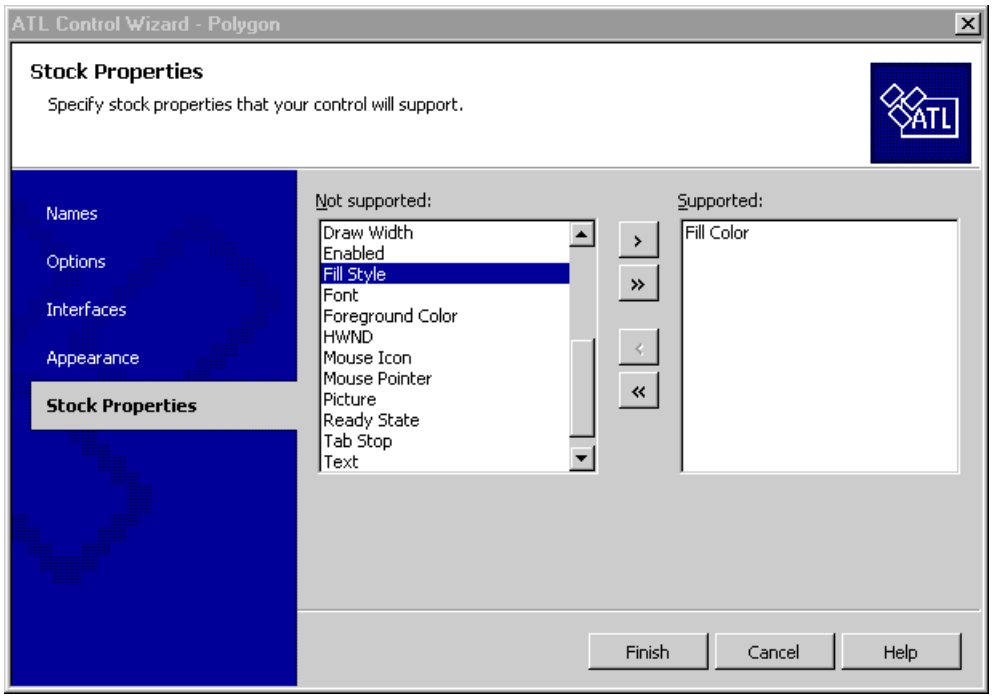

3. This completes the options for the control. Click **Finish**.

As the wizard created the control, several code changes and file additions occurred. The following files were created:

File	Description
PolyCtl.h	Contains most of the implementation of the C++ class `CPolyCtl`.
PolyCtl.cpp	Contains the remaining parts of `CPolyCtl`.
PolyCtl.rgs	A text file that contains the registry script used to register the control.
PolyCtl.htm	A Web page containing a reference to the newly created control.

The wizard also performed the following code changes:

- Added an **#include** statement to the stdafx.h and stdafx.cpp files to include the ATL files necessary for supporting controls.
- Changed Polygon.idl to include details of the new control.
- Added the new control to the object map in Polygon.cpp.

Now you can build the control to see it in action.

Building and Testing the Control

To build and test the control

1. On the **Build** menu, click **Build Polygon**.
2. Once the control finishes building, double-click PolyCtl.htm in Solution Explorer. The HTML Web page containing the control will be displayed. You should see a rectangle and the text `ATL 8.0 : PolyCtl`. This is your control.

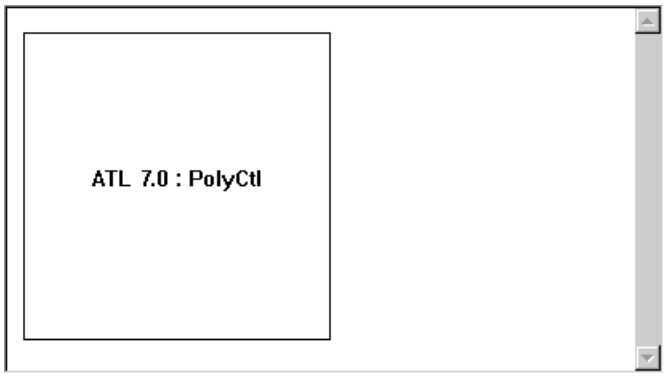

Note

When completing this tutorial, if you receive an error message where the DLL file cannot be created, close the PolyCtl.htm file and the ActiveX Control Test container and build the solution again. If you still cannot create the DLL, reboot the computer or log off (if you are using Terminal Services).

Next, you will add a custom property to the control.

Back to Step 1 | On to Step 3

See Also

Reference

ATL Tutorial

Step 3: Adding a Property to the Control

`IPolyCtl` is the interface that contains the control's custom methods and properties, and you will add a property to it.

To add a property using the Add Property Wizard

1. Right-click IPolyCtl in Class View (expand the Polygon branch to find it).
2. On the shortcut menu, click **Add**, and then click **Add Property**.

   The Add Property Wizard will appear.
3. In the drop-down list of property types, select **SHORT**.
4. Type `Sides` as the **Property name**:

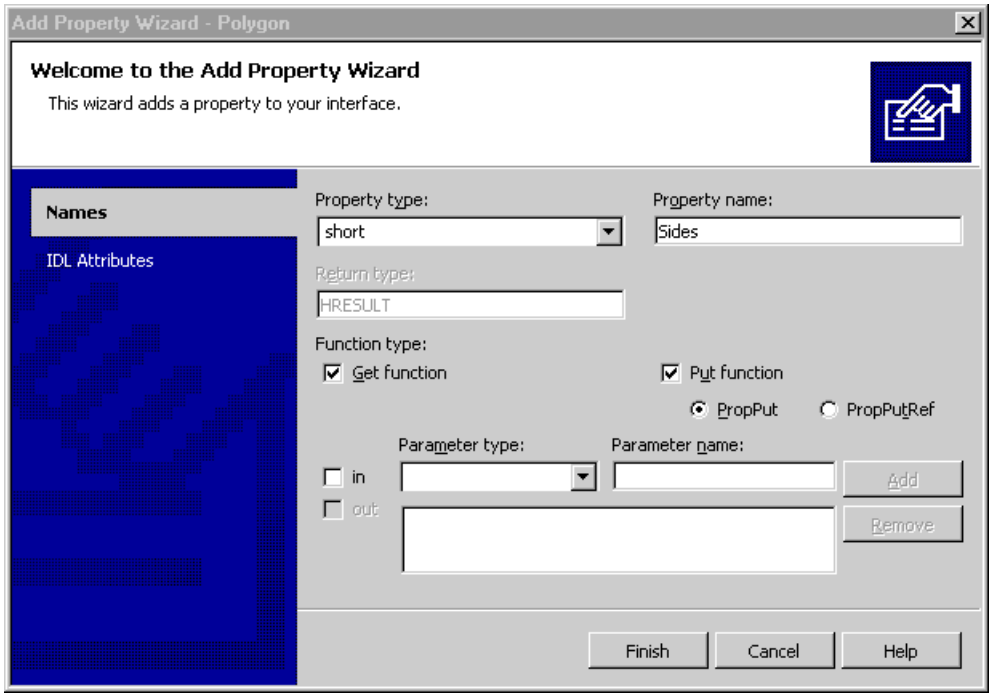

5. Click **Finish** to finish adding the property.

When you add the property to the interface, MIDL (the program that compiles .idl files) defines a `Get` method for retrieving its value and a `Put` method for setting a new value. The methods are named by prepending `put_` and `get_` to the property name.

The Add Property Wizard adds the necessary lines to the .idl file. It also adds the `Get` and `Put` function prototypes to the class definition in PolyCtl.h and adds an empty implementation to PolyCtl.cpp. You can check this by opening PolyCtl.cpp and looking for the functions `get_Sides` and `put_Sides`.

Although you now have skeleton functions to set and retrieve the property, it needs a place to be stored. You will create a variable to store the property and update the functions accordingly.

To create a variable to store the property, and update the put and get methods

1. From Solution Explorer, open PolyCtl.h and add the following line at the end of the class definition, after the definition of `m_clrFillColor`:

```
short m_nSides;
```

2. Set the default value of `m_nSides`. Make the default shape a triangle by adding a line to the constructor in PolyCtl.h:

```
CPolyCtl()
{
   m_nSides = 3;
}
```

3. Implement the `Get` and `Put` methods. The `get_Sides` and `put_Sides` function declarations have been added to PolyCtl.h. Add the following code to PolyCtl.cpp to complete both methods:

```
STDMETHODIMP CPolyCtl::get_Sides(short *pVal)
{
   *pVal = m_nSides;
   return S_OK;
}
STDMETHODIMP CPolyCtl::put_Sides(short newVal)
{
   if (newVal > 2 && newVal < 101)
   {
      m_nSides = newVal;
      return S_OK;
   }
   else
     return Error(_T("Shape must have between 3 and 100 sides"));
}
```

The `get_Sides` method returns the current value of the `Sides` property through the `pVal` pointer. In the `put_Sides` method, the code ensures the user is setting the `Sides` property to an acceptable value. The minimum must be 2, and because an array of points will be used for each side, 100 is a reasonable limit for a maximum value.

You now have a property called `Sides`. In the next step, you will change the drawing code to use it.

Back to Step 2 | On to Step 4

See Also
Reference
ATL Tutorial

Step 4: Changing the Drawing Code

By default, the control's drawing code displays a square and the text `ATL 8.0 : PolyCtl`. In this step, you will change the code to display something more interesting. The following tasks are involved:

- Modifying the Header File
- Modifying the **OnDraw** Function
- Adding a Method to Calculate the Polygon Points
- Initializing the Fill Color

Modifying the Header File

Start by adding support for the math functions `sin` and `cos`, which will be used calculate the polygon points, and by creating an array to store positions.

To modify the header file

1. Add the line `#include <math.h>` to the top of PolyCtl.h:

```
#include <math.h>
#include "resource.h"       // main symbols
```

2. Once the polygon points are calculated, they will be stored in an array of type `POINT`, so add the array to the end of the class definition in PolyCtl.h:

```
OLE_COLOR m_clrFillColor;
short m_nSides;
POINT m_arrPoint[100];
```

Modifying the OnDraw Method

Now you should modify the `OnDraw` method in PolyCtl.h. The code you will add creates a new pen and brush with which to draw your polygon, and then calls the **Ellipse** and **Polygon** Win32 API functions to perform the actual drawing.

To modify the OnDraw function

- Replace the existing `OnDraw` method in PolyCtl.h with the following code:

```
HRESULT CPolyCtl::OnDraw(ATL_DRAWINFO& di)
{
   RECT& rc = *(RECT*)di.prcBounds;
   HDC hdc  = di.hdcDraw;

   COLORREF    colFore;
   HBRUSH      hOldBrush, hBrush;
   HPEN        hOldPen, hPen;

   // Translate m_colFore into a COLORREF type
   OleTranslateColor(m_clrFillColor, NULL, &colFore);

   // Create and select the colors to draw the circle
   hPen = (HPEN)GetStockObject(BLACK_PEN);
   hOldPen = (HPEN)SelectObject(hdc, hPen);
   hBrush = (HBRUSH)GetStockObject(WHITE_BRUSH);
   hOldBrush = (HBRUSH)SelectObject(hdc, hBrush);
```

```
   Ellipse(hdc, rc.left, rc.top, rc.right, rc.bottom);

   // Create and select the brush that will be used to fill the polygon
   hBrush    = CreateSolidBrush(colFore);
   SelectObject(hdc, hBrush);

   CalcPoints(rc);
   Polygon(hdc, &m_arrPoint[0], m_nSides);

   // Select back the old pen and brush and delete the brush we created
   SelectObject(hdc, hOldPen);
   SelectObject(hdc, hOldBrush);
   DeleteObject(hBrush);

   return S_OK;
}
```

Adding a Method to Calculate the Polygon Points

Add a method, called `CalcPoints`, that will calculate the coordinates of the points that make up the perimeter of the polygon. These calculations will be based on the RECT variable that is passed into the function.

To add the CalcPoints method

1. Add the declaration of `CalcPoints` to the `IPolyCtl` public section of the `CPolyCtl` class in PolyCtl.h:

   ```
   void CalcPoints(const RECT& rc);
   ```

   The last part of the public section of the `CPolyCtl` class will look like this:

   ```
   void FinalRelease()
      {
      }
   STDMETHOD(get_Sides)(short* pVal);
   STDMETHOD(put_Sides)(short newVal);
   void CalcPoints(const RECT& rc);
   ```

2. Add this implementation of the `CalcPoints` function to the end of PolyCtl.cpp:

   ```
   void CPolyCtl::CalcPoints(const RECT& rc)
   {
      const double pi = 3.14159265358979;
      POINT   ptCenter;
      double  dblRadiusx = (rc.right - rc.left) / 2;
      double  dblRadiusy = (rc.bottom - rc.top) / 2;
      double  dblAngle = 3 * pi / 2;          // Start at the top
      double  dblDiff  = 2 * pi / m_nSides;   // Angle each side will make
      ptCenter.x = (rc.left + rc.right) / 2;
      ptCenter.y = (rc.top + rc.bottom) / 2;

      // Calculate the points for each side
      for (int i = 0; i < m_nSides; i++)
      {
         m_arrPoint[i].x = (long)(dblRadiusx * cos(dblAngle) + ptCenter.x + 0.5);
         m_arrPoint[i].y = (long)(dblRadiusy * sin(dblAngle) + ptCenter.y + 0.5);
         dblAngle += dblDiff;
      }
   }
   ```

Initializing the Fill Color

Initialize `m_clrFillColor` with a default color.

To initialize the fill color

- Use green as the default color by adding this line to the `CPolyCtl` constructor in PolyCtl.h:

```
m_clrFillColor = RGB(0, 0xFF, 0);
```

The constructor now looks like this:

```
CPolyCtl()
{
   m_nSides = 3;
   m_clrFillColor = RGB(0, 0xFF, 0);
}
```

Building and Testing the Control

Rebuild the control. Make sure the PolyCtl.htm file is closed if it is still open, and then click **Build Polygon** on the **Build** menu. You could view the control once again from the PolyCtl.htm page, but this time use the ActiveX Control Test Container.

To use the ActiveX Control Test Container

1. On the **Tools** menu, click **ActiveX Control Test Container.**
2. In Test Container, on the **Edit** menu, click **Insert New Control**.
3. Locate your control, which will be called `PolyCtl Class`, and click **OK**. You will see a green triangle within a circle.

Try changing the number of sides by following the next procedure. To modify properties on a dual interface from within Test Container, use **Invoke Methods**.

To modify a control's property from within the Test Container

1. In Test Container, click **Invoke Methods** on the **Control** menu.

   The **Invoke Method** dialog box is displayed:

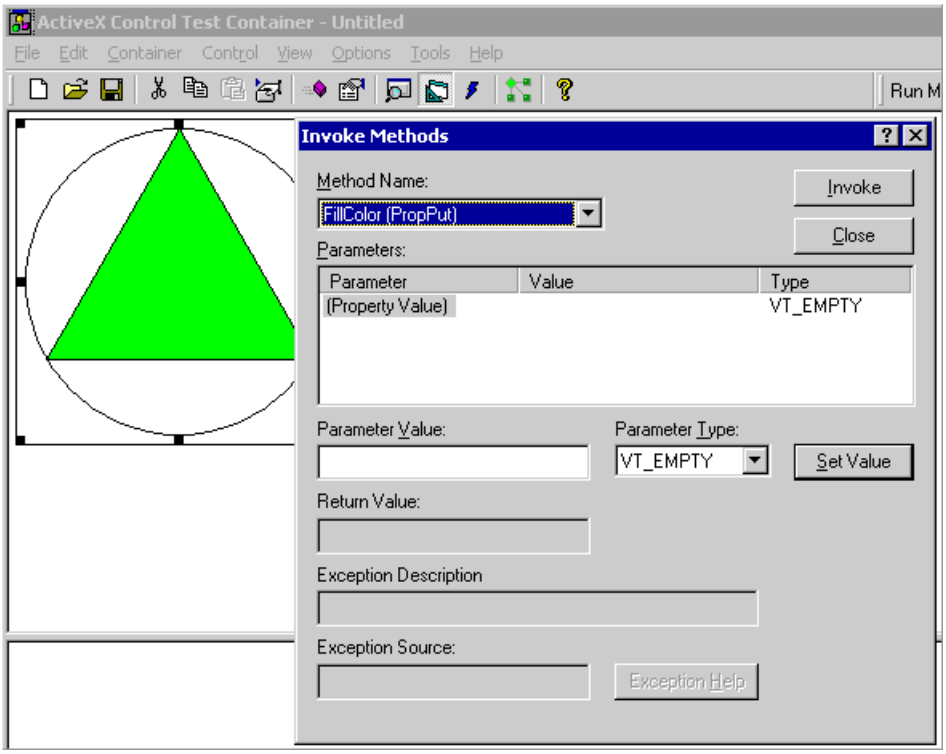

2. Select the **PropPut** version of the **Sides** property from the **Method Name** drop-down list box.
3. Type `5` in the **Parameter Value** box, click **Set Value**, and click **Invoke**.

Note that the control does not change. Although you changed the number of sides internally by setting the `m_nSides` variable, this did not cause the control to repaint. If you switch to another application and then switch back to Test Container, you will find that the control has repainted and has the correct number of sides.

To correct this problem, add a call to the `FireViewChange` function, defined in **IViewObjectExImpl**, after you set the number of sides. If the control is running in its own window, `FireViewChange` will call the `InvalidateRect` method directly. If the control is running windowless, the `InvalidateRect` method will be called on the container's site interface. This forces the control to repaint itself.

To add a call to FireViewChange

- Update PolyCtl.cpp by adding the call to `FireViewChange` to the `put_Sides` method. When you have finished, the `put_Sides` method should look like this:

```
STDMETHODIMP CPolyCtl::put_Sides(short newVal)
{
   if (newVal > 2 && newVal < 101)
   {
      m_nSides = newVal;
      FireViewChange();
      return S_OK;
   }
   else
      return Error(_T("Shape must have between 3 and 100 sides"));
}
```

After adding `FireViewChange`, rebuild and try the control again in the ActiveX Control Test Container. This time when you change the number of sides and click **Invoke**, you should see the control change immediately.

In the next step, you will add an event.

Back to Step 3 | On to Step 5

See Also
Tasks
Testing Properties and Events with Test Container
Reference
ATL Tutorial

Step 5: Adding an Event

In this step, you will add a `ClickIn` and a `ClickOut` event to your ATL control. You will fire the `ClickIn` event if the user clicks within the polygon and fire `ClickOut` if the user clicks outside. The tasks to add an event are as follows:

- Adding the `ClickIn` and `ClickOut` Methods
- Generating the Type Library
- Implementing the Connection Point Interfaces

Adding the ClickIn and ClickOut Methods

When you created the ATL control in step 2, you selected the **Connection points** check box. This created the `_IPolyCtlEvents` interface in the Polygon.idl file. Note that the interface name starts with an underscore. This is a convention to indicate that the interface is an internal interface. Thus, programs that allow you to browse COM objects can choose not to display the interface to the user. Also note that selecting **Connection points** added the following line in the Polygon.idl file to indicate that `_IPolyCtlEvents` is the default source interface:

```
[default, source] dispinterface _IPolyCtlEvents;
```

The source attribute indicates that the control is the source of the notifications, so it will call this interface on the container.

Now add the `ClickIn` and `ClickOut` methods to the `_IPolyCtlEvents` interface.

To add the ClickIn and ClickOut methods

1. In Class View, expand Polygon and PolygonLib to display _IPolyCtlEvents.
2. Right-click _IPolyCtlEvents. On the shortcut menu, click **Add**, and then click **Add Method**.
3. Select a **Return Type** of `void`.
4. Enter `ClickIn` in the **Method name** box.
5. Under **Parameter attributes**, select the **in** box.
6. Select a **Parameter type** of **LONG**.
7. Type `x` as the **Parameter name**, and click **Add**.
8. Repeat steps 5 through 7, this time for a **Parameter name** of `y`.
9. Click **Finish**.
10. Repeat the steps above to define a `ClickOut` method with the same **LONG** parameters `x` and `y`, the same **Parameter attributes** and the same `void` return type.

Check the Polygon.idl file to see that the code was added to the `_IPolyCtlEvents` dispinterface.

The `_IPolyCtlEvents` dispinterface in your Polygon.idl file should now look like this:

```
dispinterface _IPolyCtlEvents
{
   properties:
   methods:
   [id(1), helpstring("method ClickIn")] void ClickIn([in]LONG x, [in] LONG y);
   [id(2), helpstring("method ClickOut")] void ClickOut([in] LONG x, [in] LONG y);
};
```

The `ClickIn` and `ClickOut` methods take the x and y coordinates of the clicked point as parameters.

Generating the Type Library

Generate the type library at this point, because the Connection Point Wizard will use it to obtain the information it needs to construct a connection point interface and a connection point container interface for your control.

To generate the type library

1. Rebuild your project.

   -or-

2. Right-click the Polygon.idl file in Solution Explorer and click **Compile** on the shortcut menu.

This will create the Polygon.tlb file, which is your type library. The Polygon.tlb file is not visible from Solution Explorer, because it is a binary file and cannot be viewed or edited directly.

Implementing the Connection Point Interfaces

Implement a connection point interface and a connection point container interface for your control. In COM, events are implemented through the mechanism of connection points. To receive events from a COM object, a container establishes an advisory connection to the connection point that the COM object implements. Because a COM object can have multiple connection points, the COM object also implements a connection point container interface. Through this interface, the container can determine which connection points are supported.

The interface that implements a connection point is called **IConnectionPoint**, and the interface that implements a connection point container is called **IConnectionPointContainer**.

To help implement **IConnectionPoint**, you will use the Implement Connection Point Wizard. This wizard generates the **IConnectionPoint** interface by reading your type library and implementing a function for each event that can be fired.

To use the Implement Connection Point Wizard

1. In Class View, right-click your control's implementation class `CPolyCtl`.
2. On the shortcut menu, click **Add**, and then click **Add Connection Point**.
3. Select **_IPolyCtlEvents** from the **Source Interfaces** list and double-click it to add it to the **Implement connection points** column. Click **Finish**. A proxy class for the connection point will be generated, in this case, `CProxy_IPolyCtlEvents`.

If you look at the generated _IPolyCtlEvents_CP.h file in Solution Explorer, you will see that it has a class called `CProxy_IPolyCtlEvents` that derives from `IConnectionPointImpl`. _IPolyCtlEvents_CP.h also defines the two methods `Fire_ClickIn` and `Fire_ClickOut`, which take the two coordinate parameters. You call these methods when you want to fire an event from your control.

The wizard also added `CProxy_PolyEvents` and `IConnectionPointContainerImpl` to your control's multiple inheritance list. The wizard also exposed `IConnectionPointContainer` for you by adding appropriate entries to the COM map.

You are finished implementing the code to support events. Now, add some code to fire the events at the appropriate moment. Remember, you are going to fire a `ClickIn` or `ClickOut` event when the user clicks the left mouse button in the control. To find out when the user clicks the button, add a handler for the **WM_LBUTTONDOWN** message.

To add a handler for the WM_LBUTTONDOWN message

1. In Class View, right-click the CPolyCtl class and click **Properties** on the shortcut menu.
2. In the **Properties** window, click the **Messages** icon and then click **WM_LBUTTONDOWN** from the list on the left.
3. From the drop-down list that appears, click <**Add**> **OnLButtonDown**. The `OnLButtonDown` handler declaration will be added to PolyCtl.h, and the handler implementation will be added to PolyCtl.cpp.

Next, modify the handler.

To modify the OnLButtonDown method

- Change the code which comprises the `OnLButtonDown` method in PolyCtl.cpp (deleting any code placed by the wizard) so that it looks like this:

```
LRESULT CPolyCtl::OnLButtonDown(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandle
d)
{
   HRGN hRgn;
```

```
    WORD xPos = LOWORD(lParam);  // horizontal position of cursor
    WORD yPos = HIWORD(lParam);  // vertical position of cursor

    CalcPoints(m_rcPos);

    // Create a region from our list of points
    hRgn = CreatePolygonRgn(&m_arrPoint[0], m_nSides, WINDING);

    // If the clicked point is in our polygon then fire the ClickIn
    //  event otherwise we fire the ClickOut event
    if (PtInRegion(hRgn, xPos, yPos))
       Fire_ClickIn(xPos, yPos);
    else
       Fire_ClickOut(xPos, yPos);

    // Delete the region that we created
    DeleteObject(hRgn);
    return 0;
 }
```

This code makes use of the points calculated in the `OnDraw` function to create a region that detects the user's mouse clicks with the call to `PtInRegion`.

The `uMsg` parameter is the ID of the Windows message being handled. This allows you to have one function that handles a range of messages. The `wParam` and the `lParam` parameters are the standard values for the message being handled. The parameter `bHandled` allows you to specify whether the function handled the message or not. By default, the value is set to `TRUE` to indicate that the function handled the message, but you can set it to `FALSE`. This will cause ATL to continue looking for another message handler function to send the message to.

Building and Testing the Control

Now try out your events. Build the control and start the ActiveX Control Test Container again. This time, view the event log window. To route events to the output window, click **Logging** from the **Options** menu and select **Log to output window**. Insert the control and try clicking in the window. Note that `ClickIn` is fired if you click within the filled polygon, and `ClickOut` is fired when you click outside of it.

Next, you will add a property page.

Back to Step 4 | On to Step 6

See Also

Reference

ATL Tutorial

Step 6: Adding a Property Page

Property pages are implemented as separate COM objects, which allow them to be shared if required. In this step, you will do the following tasks to add a property page to the control:

- Creating the Property Page Resource
- Adding Code to Create and Manage the Property Page
- Adding the Property Page to the Control

Creating the Property Page Resource

To add a property page to your control, use the ATL Add Class Wizard.

To add a Property Page

1. In Solution Explorer, right-click Polygon.
2. On the shortcut menu, click **Add**, and then click **Add Class**.
3. From the list of templates, select **ATL Property Page** and click **Open**.
4. When the ATL Property Page Wizard appears, enter `PolyProp` as the **Short** name:

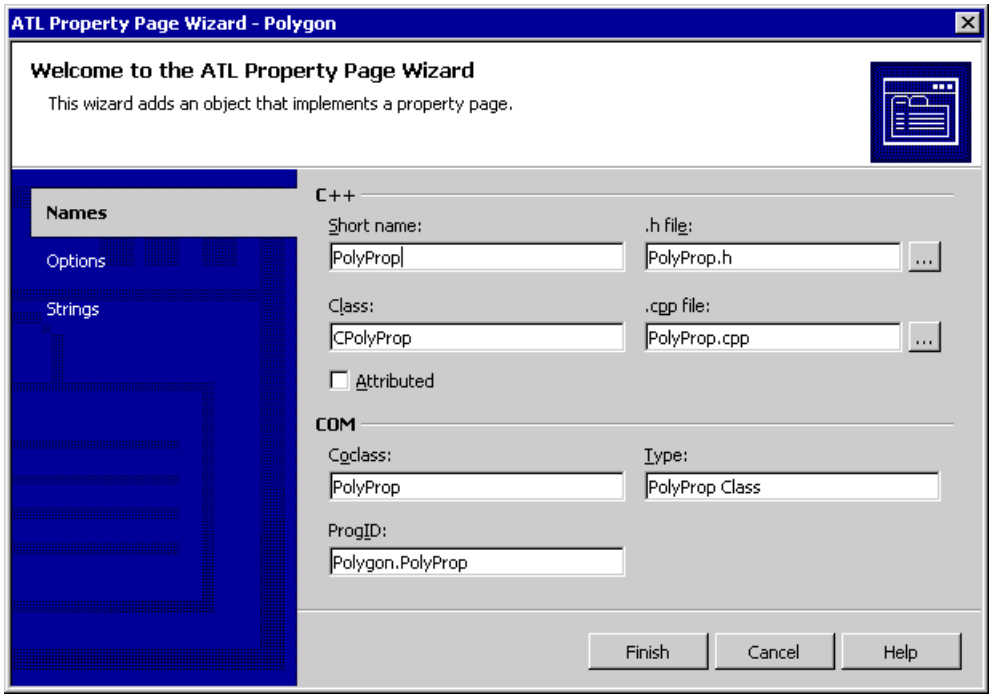

5. Click **Strings** to open the **Strings** page and enter `&Polygon` as the **Title**:

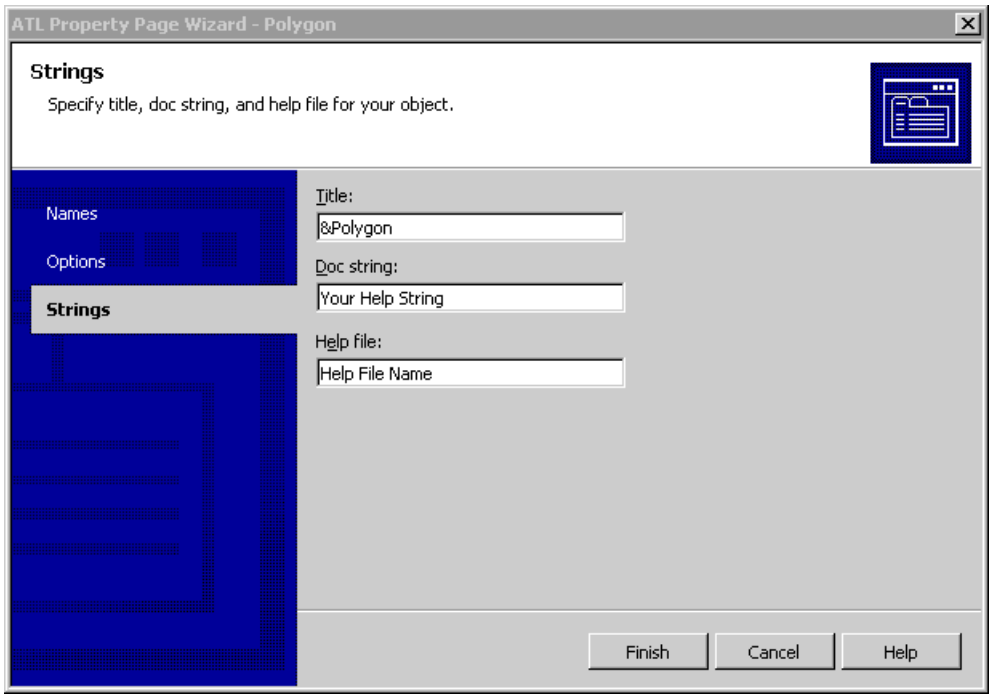

The **Title** of the property page is the string that appears in the tab for that page. The **Doc string** is a description that a property frame uses to put in a status line or tool tip. Note that the standard property frame currently does not use this string, so you can leave it with the default contents. You will not going to generate a **Help file** at the moment, so delete the entry in that text box.

6. Click **Finish**, and the property page object will be created.

The following three files are created:

File	Description
PolyProp.h	Contains the C++ class `CPolyProp`, which implements the property page.
PolyProp.cpp	Includes the PolyProp.h file.
PolyProp.rgs	The registry script that registers the property page object.

The following code changes are also made:

- The new property page is added to the object entry map in Polygon.cpp.
- The `PolyProp` class is added to the Polygon.idl file.
- The new registry script file PolyProp.rgs is added to the project resource.
- A dialog box template is added to the project resource for the property page.
- The property strings that you specified are added to the resource string table.

Now add the fields that you want to appear on the property page.

To add fields to the Property Page

1. In Solution Explorer, double-click the Polygon.rc resource file. This will open Resource View.
2. In Resource View, expand the Dialog node and double-click IDD_POLYPROP. Note that the dialog box that appears is empty except for a label that tells you to insert your controls here.
3. Select that label and change it to read `Sides:` by altering the **Caption** text in the **Properties** window and resizing the label box:

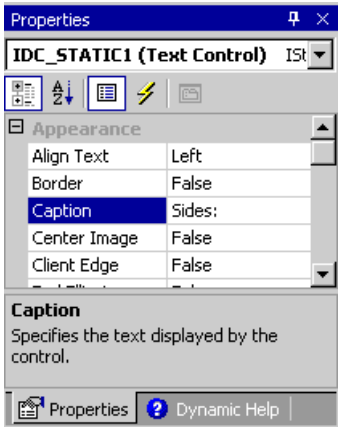

4. Drag an Edit control from the Toolbox to the right of the label:

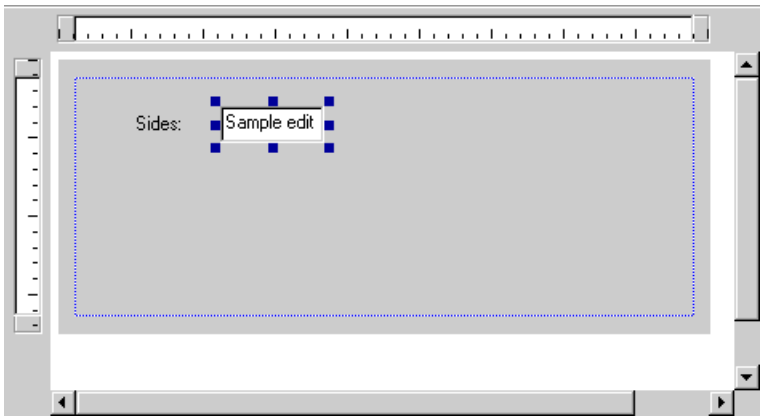

5. Finally, change the **ID** of the Edit control to `IDC_SIDES` using the Properties window.

This completes the process of creating the property page resource.

Adding Code to Create and Manage the Property Page

Now that you have created the property page resource, you need to write the implementation code.

First, enable the `CPolyProp` class to set the number of sides in your object when the **Apply** button is pressed.

To modify the Apply function to set the number of sides

- Change the `Apply` function in PolyProp.h as follows:

```
STDMETHOD(Apply)(void)
{
   USES_CONVERSION;
   ATLTRACE(_T("CPolyProp::Apply\n"));
   for (UINT i = 0; i < m_nObjects; i++)
   {
      CComQIPtr<IPolyCtl, &IID_IPolyCtl> pPoly(m_ppUnk[i]);
      short nSides = (short)GetDlgItemInt(IDC_SIDES);
      if FAILED(pPoly->put_Sides(nSides))
      {
         CComPtr<IErrorInfo> pError;
         CComBSTR strError;
         GetErrorInfo(0, &pError);
         pError->GetDescription(&strError);
         MessageBox(OLE2T(strError), _T("Error"), MB_ICONEXCLAMATION);
         return E_FAIL;
      }
   }
   m_bDirty = FALSE;
   return S_OK;
```

```
}
```

A property page can have more than one client attached to it at a time, so the `Apply` function loops around and calls `put_Sides` on each client with the value retrieved from the edit box. You are using the CComQIPtr class, which performs the **QueryInterface** on each object to obtain the `IPolyCtl` interface from the **IUnknown** interface (stored in the `m_ppUnk` array).

The code now checks that setting the `Sides` property actually worked. If it fails, the code displays a message box displaying error details from the **IErrorInfo** interface. Typically, a container asks an object for the **ISupportErrorInfo** interface and calls **InterfaceSupportsErrorInfo** first, to determine whether the object supports setting error information. You can skip this task.

CComPtr helps you by automatically handling the reference counting, so you do not need to call **Release** on the interface. **CComBSTR** helps you with **BSTR** processing, so you do not have to perform the final **SysFreeString** call. You also use one of the various string conversion classes, so you can convert the **BSTR** if necessary (this is why the **USES_CONVERSION** macro is at the start of the function).

You also need to set the property page's dirty flag to indicate that the **Apply** button should be enabled. This occurs when the user changes the value in the **Sides** edit box.

To handle the Apply button

1. In Class View, right-click CPolyProp and click **Properties** on the shortcut menu.
2. In the Properties window, click the **Events** icon.
3. Expand the **IDC_SIDES** node in the event list.
4. Select **EN_CHANGE**, and from the drop-down menu to the right, click **<Add> OnEnChangeSides**. The `OnEnChangeSides` handler declaration will be added to Polyprop.h, and the handler implementation to Polyprop.cpp.

Next, you will modify the handler.

To modify the OnEnChangeSides method

- Add the following code in Polyprop.cpp to the `OnEnChangeSides` method (deleting any code that the wizard put there):

```
LRESULT CPolyProp::OnEnChangeSides(WORD /*wNotifyCode*/,
WORD /*wID*/, HWND /*hWndCtl*/, BOOL& /*bHandled*/)
{
   SetDirty(TRUE);
   return 0;
}
```

`OnEnChangeSides` will be called when a **WM_COMMAND** message is sent with the **EN_CHANGE** notification for the `IDC_SIDES` control. `OnEnChangeSides` then calls `SetDirty` and passes `TRUE` to indicate the property page is now dirty and the **Apply** button should be enabled.

Adding the Property Page to the Control

The ATL Add Class Wizard and the ATL Property Page Wizard do not add the property page to your control for you automatically, because there could be multiple controls in your project. You will need to add an entry to the control's property map.

To add the property page

- Open PolyCtl.h and add this line to the property map:

```
PROP_ENTRY("Sides", 1, CLSID_PolyProp)
```

The control's property map now looks like this:

```
BEGIN_PROP_MAP(CPolyCtl)
   PROP_DATA_ENTRY("_cx", m_sizeExtent.cx, VT_UI4)
   PROP_DATA_ENTRY("_cy", m_sizeExtent.cy, VT_UI4)
```

```
    PROP_ENTRY("FillColor", DISPID_FILLCOLOR, CLSID_StockColorPage)
    PROP_ENTRY("Sides", 1, CLSID_PolyProp)
    // Example entries
    // PROP_ENTRY("Property Description", dispid, clsid)
    // PROP_PAGE(CLSID_StockColorPage)
END_PROP_MAP()
```

You could have added a `PROP_PAGE` macro with the CLSID of your property page, but if you use the `PROP_ENTRY` macro as shown, the `Sides` property value is also saved when the control is saved.

The three parameters to the macro are the property description, the DISPID of the property, and the CLSID of the property page that has the property on it. This is useful if, for example, you load the control into Visual Basic and set the number of Sides at design time. Because the number of Sides is saved, when you reload your Visual Basic project, the number of Sides will be restored.

Building and Testing the Control

Now build that control and insert it into ActiveX Control Test Container. In Test Container, on the **Edit** menu, click **PolyCtl Class Object**. The property page appears; click the **Polygon** tab.

The **Apply** button is initially disabled. Start typing a value in the **Sides** box and the **Apply** button will become enabled. After you have finished entering the value, click the **Apply** button. The control display changes, and the **Apply** button is again disabled. Try entering an invalid value. You will see a message box containing the error description that you set from the `put_Sides` function.

Next, you will put your control on a Web page.

Back to Step 5 | On to Step 7

See Also

Reference

ATL Tutorial

Step 7: Putting the Control on a Web Page

Your control is now finished. To see your control work in a real-world situation, put it on a Web page. An HTML file that contains the control was created when you defined your control. Open the PolyCtl.htm file from Solution Explorer, and you can see your control on a Web page.

In this step, you will script the Web page to respond to events. You will also modify the control to let Internet Explorer know that the control is safe for scripting.

Scripting the Web Page

The control does not do anything yet, so change the Web page to respond to the events that you send.

To script the Web page

- Open PolyCtl.htm and select HTML view. Add the lines in **bold** to the HTML code that makes up the page.

```
<HTML>
<HEAD>
<TITLE>ATL 8.0 test page for object PolyCtl</TITLE>
</HEAD>
<BODY>
<OBJECT ID="PolyCtl" <
 CLASSID="CLSID:4CBBC676-507F-11D0-B98B-000000000000">
>
</OBJECT>
<SCRIPT LANGUAGE="VBScript">
<!--
Sub PolyCtl_ClickIn(x, y)
   PolyCtl.Sides = PolyCtl.Sides + 1
End Sub
Sub PolyCtl_ClickOut(x, y)
   PolyCtl.Sides = PolyCtl.Sides - 1
End Sub
-->
</SCRIPT>
</BODY>
</HTML>
```

You have added some VBScript code that gets the Sides property from the control and increases the number of sides by one if you click inside the control. If you click outside the control, you reduce the number of sides by one.

Indicating that the Control Is Safe for Scripting

You can view the Web page with the control in Internet Explorer or, more conveniently, use the Web browser view built into Visual C++ .NET. To see your control in the Web browser view, right-click PolyCtl.htm, and click **View in Browser.**

Based on your current Internet Explorer security settings, you may receive a Security Alert dialog box stating that the control may not be safe to script and could potentially do damage. For example, if you had a control that displayed a file but also had a **Delete** method that deleted a file, it would be safe if you just viewed it on a page. It would be not safe to script, however, because someone could call the **Delete** method.

Security Note For this tutorial, you can change your security settings in Internet Explorer to run ActiveX controls that are not marked as safe. In Control Panel, click **Internet Properties** and click **Security** to change the appropriate settings. When you have completed the tutorial, change your security settings back to their original state.

You can programmatically alert Internet Explorer that it does not need to display the Security Alert dialog box for this particular control. You can do this with the **IObjectSafety** interface, and ATL supplies an implementation of this interface in the class IObjectSafetyImpl. To add the interface to your control, add **IObjectSafetyImpl** to your list of inherited classes and add an entry for it in your COM map.

To add IObjectSafetyImpl to the control

1. Add the following line to the end of the list of inherited classes in PolyCtl.h and add a comma to the previous line:

```
public IObjectSafetyImpl<CPolyCtl, INTERFACESAFE_FOR_UNTRUSTED_CALLER>
```

2. Add the following line to the COM map in PolyCtl.h:

```
COM_INTERFACE_ENTRY(IObjectSafety)
```

Building and Testing the Control

Build the control. Once the build has finished, open PolyCtl.htm in browser view again. This time, the Web page should be displayed directly without the Safety Alert dialog box. Click inside the polygon; the number of sides increases by one. Click outside the polygon to reduce the number of sides. If you try to reduce the number of sides below three, you will see the error message that you set. Here is the control after one click:

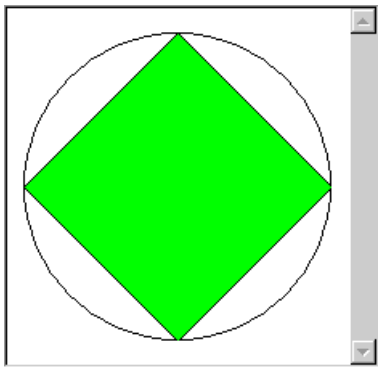

Back to Step 6

Next Steps

This concludes the ATL tutorial. For links to more information about ATL, see the ATL start page.

See Also

Reference

ATL Tutorial

Attributes Tutorial

Using Visual C++ and attributes, you can speed up and simplify the process of COM programming. This tutorial uses attributes to implement both a client and a server application. During the course of this tutorial, you will use attributes and events.

The tutorial develops a singleton server object (an object that can have only one instance) that has its own dual interface and a dispatch interface used for firing off events. The server object takes data, given to it through the **Send** method of its dual interface, and transmits it to all connected components through the **Transfer** event on its dispatch interface.

In addition, the tutorial implements a client (an ActiveX control) that contains a server object. The control responds to the **Transfer** event fired by the server object and has its own dual interface that implements several methods: **Connect**, **Send**, and **Disconnect**. If the **Transfer** event is fired with a variant containing a **BSTR**, the string is displayed in the center of the control.

The tutorial is divided into seven steps, each building on the application developed in the previous step.

- Step 1: Creating the Projects
- Step 2: Adding the Server Object
- Step 3: Implementing the Server
- Step 4: Adding the Client Object
- Step 5: Adding the Client Interfaces
- Step 6: Implementing the Client
- Step 7: Using the Client Control

Note This tutorial creates similar source code as the source code for the DispSink sample. You can download the DispSink source code from the DispSink sample abstract. If you prefer to read the tutorial without creating your project, you can refer to the DispSink source code. You can also compare your code to the DispSink code if you run into any build errors during the tutorial.

See Also

Other Resources

Attributed Programming Concepts
ATL Concepts
ATL Class Overview
ATL Samples
ATL Attributes Samples

Step 1: Creating the Projects

In this step, you will create an initial solution containing two ATL projects. The two projects will implement the server and client objects of the tutorial.

To create a new solution and project

1. In the Visual Studio environment, click **New** from the **File** menu and then click **Project**.
2. From the **Visual C++ Projects** folder, select **ATL Project**.
3. In the **Name** box, type **DispClient**.
4. In the **Solution Name** box, type **DispSink**.
5. Click **OK** to create the DispSink solution and start the ATL Project Wizard.
6. Click the **Application Settings** tab to display the current options for the initial project.
7. Check the **Attributed** checkbox.
8. Click **Finish** to generate the DispClient project.

The files generated by the wizard are listed in the following table.

File	Description
DispClient.cpp	Contains the `module` attribute, which implements `DLLMain`, `DLLRegisterServer`, and `DLLUnregisterServer`. The module type also defines the GUID for the type library. Notice that the GUID and helpstring have been automatically generated.
DispClient.rc	The resource file, which initially contains the version information and a string containing the project name.
DispClient.rgs	Contains entries that are added to the registry, which will register your COM object.
DispClient.vcproj	A file containing the project settings.
DispClientps.def	The module definition file for the proxy/stub DLL. For purposes of this tutorial, this is unnecessary.
ReadMe.txt	A file containing an explanation of the files generated by the application wizard.
Resource.h	The header file for the resource file.
StdAfx.cpp	The file that will **#include** the ATL implementation files.
StdAfx.h	The file that will **#include** the ATL header files.

You will also notice a DispClientPS project. This project creates the proxy and stub that allow your object to be accessed from outside of its COM apartment.

Now add the server project to the solution.

To create a new server project

1. In Solution Explorer, right-click the DispSink solution node.
2. On the shortcut menu, click **Add** and then click **New Project**.

   The **Add New Project** dialog box appears.
3. From the **Visual C++ Projects** folder, select **ATL Project**.
4. In the **Name** box, type **DispServer**.
5. Click **OK** to start the ATL Project Wizard.

   The ATL Project Wizard appears.
6. Click the **Application Settings** tab to display the current options for the initial project.
7. Check the **Attributed** checkbox.
8. Select the **Executable** server type.
9. Click **Finish** to generate the DispServer project.

For the DispServer project, the application wizard creates a similar set of files compared to the DispClient project. The only difference is in the file DispServer.cpp, where the module type is **exe** instead of **dll**.

The next step focuses on the implementation of the server object.

On to Step 2 | Back to the Overview

See Also
Reference
Attributes Tutorial
Other Resources
ATL Concepts
ATL Class Overview

Step 2: Adding the Server Object

In this step, you will use Class View to add objects to the project. You need to add a single ATL object (named `CDispServ`) to the server. This object also serves as an event source.

To add a class to the project

1. In Class View, right-click the DispServer project.
2. On the shortcut menu, click **Add** and then click **Class**.

   The **Add Class** dialog box appears.
3. Select **ATL Simple Object** and click **Add**.

   The ATL Simple Object Wizard appears.
4. In the **Short name** field, type **DispServ**.

   The remaining fields are automatically completed.

   The additional fields contain information on the name of the class as well as the names of the files that should be created. The **Type** field is a description of the object, while the **ProgID** field is the readable name that can be used to look up the CLSID of the object. Note that the **Attributed** box is selected and unavailable. An ATL object created by the wizard in an attributed project is always attributed.
5. Click the **Options** tab in the wizard.
6. On the **Options** tab, select **Connection points** support. This allows the object to act as an event source.
7. Click **Finish** to generate the `DispServ` class.

Note in Class View that the `CDispServ` class, as well as the `IDispServ` and `_IDispServEvents` interfaces, have been created and are now visible.

To implement the new class, the wizard added two new files to the project:

- DispServ.h contains most of the implementation of the `CDispServ` class, as well as the interfaces. The `CDispServ` class has automatically been made a COM event source through the `event_source` attribute with the `_IDispServEvents` interface automatically specified as the event interface for `CDispServ`. UUIDs, progids, help strings, and version numbers have been automatically generated for the class and the interfaces.
- DispServ.cpp contains the remainder of the `CDispServ` class. At this point, it includes a few necessary files.

You can build the application by clicking **Build DispServer** from the **Build** menu, though DispServer does not actually do anything interesting yet. However, it does have the capability to register itself. This is done automatically when the project is built.

The next step implements the functionality of the `DispServer` object.

On to Step 3 | Back to Step 1

See Also

Reference

Attributes Tutorial

Other Resources

ATL Concepts

ATL Class Overview

Step 3: Implementing the Server

In this step, you will add the functionality to make the class do something interesting. In the last two steps, you have created the `CDispServ` class, which now exposes a custom interface (`IDispServ`), and an event source (the `_IDispServEvents` event interface). Beyond this implementation, however, it does not actually do anything.

The main purpose of the `CDispServ` class is to receive data using the `Send` method and transmit the information using the `Transfer` event. Therefore, a `DispClient` object connected to a `DispServer` object can call `Send` through the `IDispServ` interface, passing it data. The `Send` method then fires the `Transfer` event, propagating the data to all connected `DispClient` objects. Previously, this required creating a connection point proxy class to fire the events. With the new events feature, the situation is simplified; you will add the `Transfer` event to the `_IDispServEvents` interface.

To add and define the Transfer event

1. In Class View, click the DispServer project to expand the node.
2. Double-click the `_IDispServEvents` interface.

   This opens the interface definition in the Code editor.
3. Define the `Transfer` event for the `_IDispServEvents` interface by adding the following code to the `_IDispServEvents` interface definition:

   ```
   [id(1), helpstring("method Transfer")] HRESULT Transfer(VARIANT data);
   ```
4. Make `IDispatch` a base class of the interface.

Your interface definition should now match the following definition:

```
__interface _IDispServEvents : public IDispatch
{
   [id(1), helpstring("method Transfer")] HRESULT Transfer(VARIANT data);
};
```

To fire the `Transfer` event, make a call to `_IDispServEvents_Transfer` from your event source. The form for firing an event is always *InterfaceName_EventName*.

To add and define the Send method

1. In Class View, double-click the `IDispServ` interface.

   This opens the interface definition in the Code editor.
2. Define the `Send` method for the `IDispServ` interface by adding the following code to the `IDispServ` interface definition:

   ```
   [id(1), helpstring("method Send")] HRESULT Send(VARIANT data);
   ```

Your interface definition should now match the following definition:

```
__interface IDispServ : IDispatch
{
   [id(1), helpstring("method Send")] HRESULT Send(VARIANT data);
};
```

To implement the `Send` method for the `CDispServ` class, scroll down to the bottom of DispServ.h and add the following code directly below the last `public:` section of the class:

```
STDMETHOD(Send)(VARIANT data)
{
   _IDispServEvents_Transfer(data);
```

```
    return (S_OK);
}
```

The full `CDispServ` class should now look like this:

```
class ATL_NO_VTABLE CDispServ :
   public IDispServ
{
public:
   CDispServ()
   {
   }

   __event __interface _IDispServEvents;

   DECLARE_PROTECT_FINAL_CONSTRUCT()

   HRESULT FinalConstruct()
   {
      return S_OK;
   }

   void FinalRelease()
   {
   }

public:
   STDMETHOD(Send)(VARIANT data)
   {
      _IDispServEvents_Transfer(data);
      return S_OK;
   }
};
```

The final task implements the `CDispServ` class as a singleton server, which means that all clients will connect to the same server.

To define CDispServ as a singleton server

1. In Class View, double-click the CDispServ node.
2. Add the following lines directly below the `DECLARE_PROTECT_FINAL_CONSTRUCT()` in class `CDispServ`:

   ```
   DECLARE_CLASSFACTORY_SINGLETON(CDispServ);
   ```

The server is now complete. You can build the server by selecting **Build DispServer** from the **Build** menu. Once the server is successfully built, it will register itself.

The Visual Studio development environment also provides wizards that let you add properties and methods. Just right click on the interface node in Class View and select a wizard from the context menu.

The next step adds a simple client object to the DispClient project.

On to Step 4 | Back to Step 2

See Also

Reference

Attributes Tutorial

Other Resources

ATL Concepts

ATL Class Overview

Step 4: Adding the Client Object

In this step, you will add a client object (named `CDispCtl`) to the client project. This client object will be implemented by a simple, lightweight control added by the **Add Class** dialog box.

Adding the client object to the project

1. In Class View, right-click the DispClient project.
2. On the shortcut menu, click **Add** and then click **Class**.

   The **Add Class** dialog box appears.
3. From the **ATL** node, double-click **ATL Control**.

   The ATL Control Wizard appears.
4. In the **Short name** field, type `DispCtl`. The remaining fields are automatically completed.
5. Click the **Options** tab.
6. Select **Standard control**.
7. Change the **Threading Model** to **Single**.
8. Click **Finish** to generate the `CDispCtl` class.

`CDispCtl` inherits from all the ATL template classes required for a light control. Additionally, the class inherits from `IDataObjectImpl`, providing methods to support Uniform Data Transfer, and `IProvideClassInfo2Impl`, allowing the access of type information for the object's coclass from other COM objects.

Registration of the control is achieved using the registration_script attribute. The parameter on this attribute is the name of the registration script, in this case Control.rgs, which is the default registration script for controls.

Also included is a default implementation of the `OnDraw` method.

The final step is to add the event_receiver attribute to the new class.

Adding an attribute to an existing class

1. In Class View, double-click the DispClient node and click the `CDispCtl` class.

   This will display the class definition for `CDispCtl`.
2. Just above the class definition is the attribute block for the class. Add `event_receiver(com)` to the attribute block.

The resulting attribute list should resemble the following:

```
[
   coclass,
   control,
   default(IDispCtl),
   threading(single),
   vi_progid("DispClient.DispCtl"),
   progid("DispClient.DispCtl.1"),
   version(1.0),
   uuid("0378A7CB-1492-4D2D-B0A0-C186484F7588"),
   helpstring("DispCtl Class"),
   support_error_info(IDispCtl),
   registration_script("control.rgs")
   event_receiver(com)
]
```

In the next step, you will add the needed interfaces to the control object.

On to Step 5 | Back to Step 3

See Also

Reference

Attributes Tutorial

Other Resources

ATL Concepts

ATL Class Overview

Step 5: Adding the Client Interfaces

In this step, you will implement the various methods of the `IDispCtl` interface.

To allow the client object to handle events fired from the `_IDispServEvents` interface you need to allow access to the interface exposed by the `CDispServ` class.

To allow the client object to handle events

1. Open DispCtl.h in your source editor.
2. Add the following line below the `#include <atlctl.h>` line:

   ```
   #import "progid:DispServer.DispServ.1" embedded_idl, no_namespace
   ```

   This assumes that your class name is DispServer.

In the last step, we added a simple client object (`CDispCtl`) with an empty interface to the project. At this point, you need to add the following methods to the `IDispCtl` interface:

- `Connect`

  Causes the client to hook to the server, enabling it to receive events.

- `Disconnect`

  Unhooks the event source of the client.

- `Send`

  Sends the specified data to the server.

To add the methods to your interface

1. In Class View, double-click the IDispCtl node under the DispClient project.
2. In the Source editor, add the following lines to the `IDispCtl` interface:

   ```
   [id(1), helpstring("method Connect")] HRESULT Connect();
   [id(2), helpstring("method Disconnect")] HRESULT Disconnect();
   [id(3), helpstring("method Send")] HRESULT Send(VARIANT data);
   ```

The interface should now resemble the following code:

```
[
   object,
   uuid(...),
   dual,
   helpstring("IDispCtl Interface"),
   pointer_default(unique)
]
__interface IDispCtl : IDispatch
{
   [id(1), helpstring("method Connect")] HRESULT Connect();
   [id(2), helpstring("method Disconnect")] HRESULT Disconnect();
   [id(3), helpstring("method Send")] HRESULT Send(VARIANT data);
};
```

In the next step, you will add the implementation of the `IDispCtl` methods and modify other sections of code.

On to Step 6 | Back to Step 4

See Also

Reference

Attributes Tutorial

Other Resources

ATL Concepts

ATL Class Overview

Step 6: Implementing the Client

In this step, you will implement `IDispCtl`'s methods in `CDispCtl`, add an event handler, and modify the `OnDraw` function.

Reminder: You can download completed source for this tutorial from the DispSink sample abstract.

Implementing the IDispCtl Interface

`CDispCtl` is where you will implement methods declared in `IDispCtl`.

The implementation begins with adding the following three data members, used by the new methods:

- `m_bConnected (bool)`

  Indicates the connected state of the server.

- `m_pIServ (_IDispServEvents*)`

  A pointer to the `IDispServ` interface the client will connect to.

- `m_OutText (variant)`

  Holds the data received from the server.

To add the data members

1. In Class View, double-click the IDispCtl node under the DispClient project.
2. In the Source editor, add the following lines at the end of the `CDispCtl` class, in a **private:** section:

```
private:
   // Data
   bool m_bConnected;
   CComPtr<IDispServ> m_spIServ;
   CComVariant m_OutText;
```

3. To initialize the new data members, modify the default constructor to match the following:

```
CDispCtl()
{
   m_bConnected = false;
   m_OutText = L"Not connected";
}
```

4. To ensure that you disconnect from the server upon exiting, add a destructor for the `CDispCtl` class. Add the following directly below the default constructor declaration:

```
~CDispCtl()
{
   Disconnect();
}
```

The first method you will implement is the `Connect` method. This method creates an instance of `CDispServ` using `CoCreateInstance` and connects the `Transfer` event from the newly created instance of `CDispServ` to the event handler method, `OnTransmit` (not yet implemented). The connection is achieved by the __hook keyword.

To implement the Connect method

- Below the data members created earlier, add the following code:

```
HRESULT Connect()
{
   HRESULT hr;
   hr = m_spIServ.CoCreateInstance(__uuidof(CDispServ));
   if (SUCCEEDED(hr))
   {
      hr = __hook(&_IDispServEvents::Transfer, m_spIServ,
      &CDispCtl::OnTransmit);
   }
   if (hr == S_OK)
   {
      m_bConnected = true;
      m_OutText = L"Connected";
   }
   FireViewChange();
   return(hr);
}
```

The implementation of the `Disconnect` method uses **__unhook** to disconnect the event and `Release` to release the instance of `CDispServ` created earlier.

To implement the Disconnect method

- Below the `Connect` method, add the following code directly:

```
HRESULT Disconnect()
{
   if (m_bConnected)
   {
      HRESULT hr = __unhook(&_IDispServEvents::Transfer, m_spIServ,
      &CDispCtl::OnTransmit);
      if (SUCCEEDED(hr))
      {
         m_spIServ.Release();
         m_bConnected = false;
      }
      return(hr);
   }
   return(S_OK);
}
```

Adding the Event Handler

For the control to respond to events fired from the server, you need to implement a handler (called `OnTransmit`) for the `Transmit` event. The `OnTransmit` event handler takes the data passed in from the `Transfer` event and places it in the `m_OutText` data member. It then calls `FireViewChange` (not yet implemented), which updates the control by displaying the contents of the `m_OutText` data member.

To implement a handler for the Transmit event

- Add the following code to the source file, below `Disconnect`:

```
HRESULT OnTransmit(VARIANT data)
{
   if (data.vt == VT_BSTR)
      m_OutText = data;
   FireViewChange();
   return(S_OK);
```

```
}
```

The last method you will implement is the `Send` method. This method sends data to the server object.

To implement the Send method

- Add the following code to the source file, below `OnTransmit`:

```
HRESULT Send(VARIANT data)
{
   if (m_bConnected)
      m_spIServ->Send(data);
   return(S_OK);
}
```

Modifying the OnDraw Method

The final modification you will make is to the `CDispCtl::OnDraw` method. The `OnDraw` method needs to output the contents of the data member `m_OutText` to the screen.

To modify the OnDraw method

- Replace the body of the existing `OnDraw` method with the following:

```
USES_CONVERSION;
LPCTSTR text = OLE2CT(m_OutText.bstrVal);
RECT& rc = *(RECT*)di.prcBounds;
Rectangle(di.hdcDraw, rc.left, rc.top, rc.right, rc.bottom);
SetTextAlign(di.hdcDraw, TA_CENTER|TA_BASELINE);
TextOut(di.hdcDraw,
   (rc.left + rc.right) / 2,
   (rc.top + rc.bottom) / 2,
   text,
   lstrlen(text));

return S_OK;
```

The DispClient control is now complete. Build the control by selecting **Build DispClient** from the **Build** menu.

On to Step 7 | Back to Step 5

See Also

Reference

Attributes Tutorial

Other Resources

ATL Concepts

ATL Class Overview

Step 7: Using the Client Control

In this step, you will use the client control to invoke the methods.

To use the client control

1. On the **Tools** menu, click **ActiveX Control Test Container**.

   This starts Tstcon32.exe.

2. Click the **New Control** button on the toolbar to insert a client control (CDispCtl).

   The control appears, displaying "Not Connected" as its string.

3. Right-click the control and click **Invoke Methods** on the shortcut menu.

   The **Invoke Methods** dialog box appears.

4. Ensure that **Method Name** is set to **Connect** and click **Invoke**.

   The control now displays "Connected" as its string.

5. Set **Method Name** to **Send** in the **Invoke Methods** dialog.

6. Change the **Parameter Type** to **VT_BSTR**.

7. Enter any string in the **Parameter Value** box.

8. Click the **Invoke** button.

   The string is displayed in all connected controls.

Back to Step 6 | Back to the Overview

See Also
Reference
Attributes Tutorial
Other Resources
ATL Concepts
ATL Class Overview

Introduction to COM and ATL

This section provides a brief introduction to COM and ATL.

In This Section

Introduction to COM

Provides an overview of the Component Object Model's (COM) fundamental concepts, including interfaces, **IUnknown**, reference counting, **QueryInterface**, marshaling, and aggregation.

Introduction to ATL

Discusses, briefly, what the Active Template Library (ATL) was designed for, template libraries, and ATL version numbers. Includes recommendations for choosing between ATL and MFC.

Related Sections

The Component Object Model

The *Platform SDK* material on COM.

ATL

Provides links to conceptual topics on how to program using the Active Template Library.

ATL Class Overview

Provides reference information and links to the ATL classes.

Introduction to COM

COM is the fundamental "object model" on which ActiveX Controls and OLE are built. COM allows an object to expose its functionality to other components and to host applications. It defines both how the object exposes itself and how this exposure works across processes and across networks. COM also defines the object's life cycle.

Fundamental to COM are these concepts:

- Interfaces — the mechanism through which an object exposes its functionality.
- IUnknown — the basic interface on which all others are based. It implements the reference counting and interface querying mechanisms running through COM.
- Reference counting — the technique by which an object (or, strictly, an interface) decides when it is no longer being used and is therefore free to remove itself.
- QueryInterface — the method used to query an object for a given interface.
- Marshaling — the mechanism that enables objects to be used across thread, process, and network boundaries, allowing for location independence.
- Aggregation — a way in which one object can make use of another.

See Also

Reference

The Component Object Model

Other Resources

Introduction to COM and ATL

Interfaces (ATL)

An interface is the way in which an object exposes its functionality to the outside world. In COM, an interface is a table of pointers (like a C++ vtable) to functions implemented by the object. The table represents the interface, and the functions to which it points are the methods of that interface. An object can expose as many interfaces as it chooses.

Each interface is based on the fundamental COM interface, IUnknown. The methods of **IUnknown** allow navigation to other interfaces exposed by the object.

Also, each interface is given a unique interface ID (IID). This uniqueness makes it easy to support interface versioning. A new version of an interface is simply a new interface, with a new IID.

Note
IIDs for the standard COM and OLE interfaces are predefined.

See Also

Reference

Introduction to COM

COM Objects and Interfaces

IUnknown

IUnknown is the base interface of every other COM interface. **IUnknown** defines three methods: **QueryInterface**, **AddRef**, and **Release**. QueryInterface allows an interface user to ask the object for a pointer to another of its interfaces. AddRef and Release implement reference counting on the interface.

See Also

Reference

Introduction to COM

IUnknown and Interface Inheritance

Reference Counting

COM itself does not automatically try to remove an object from memory when it thinks the object is no longer being used. Instead, the object programmer must remove the unused object. The programmer determines whether an object can be removed based on a reference count.

COM uses the **IUnknown** methods, AddRef and Release, to manage the reference count of interfaces on an object. The general rules for calling these methods are:

- Whenever a client receives an interface pointer, **AddRef** must be called on the interface.
- Whenever the client has finished using the interface pointer, it must call **Release**.

In a simple implementation, each **AddRef** call increments and each **Release** call decrements a counter variable inside the object. When the count returns to zero, the interface no longer has any users and is free to remove itself from memory.

Reference counting can also be implemented so that each reference to the object (not to an individual interface) is counted. In this case, each **AddRef** and **Release** call delegates to a central implementation on the object, and **Release** frees the entire object when its reference count reaches zero.

Note
When a **CComObject**-derived object is constructed using the **new** operator, the reference count is 0. Therefore, a call to **Add Ref** must be made after successfully creating the **CComObject**-derived object.

See Also

Reference

Introduction to COM

Managing Object Lifetimes through Reference Counting

QueryInterface

Although there are mechanisms by which an object can express the functionality it provides statically (before it is instantiated), the fundamental COM mechanism is to use the **IUnknown** method called QueryInterface.

Every interface is derived from **IUnknown**, so every interface has an implementation of **QueryInterface**. Regardless of implementation, this method queries an object using the IID of the interface to which the caller wants a pointer. If the object supports that interface, **QueryInterface** retrieves a pointer to the interface, while also calling **AddRef**. Otherwise, it returns the **E_NOINTERFACE** error code.

Note that you must obey Reference Counting rules at all times. If you call **Release** on an interface pointer to decrement the reference count to zero, you should not use that pointer again. Occasionally you may need to obtain a weak reference to an object (that is, you may wish to obtain a pointer to one of its interfaces without incrementing the reference count), but it is not acceptable to do this by calling **QueryInterface** followed by **Release**. The pointer obtained in such a manner is invalid and should not be used. This more readily becomes apparent when _ATL_DEBUG_INTERFACES is defined, so defining this macro is a useful way of finding reference counting bugs.

See Also

Reference

Introduction to COM

QueryInterface: Navigating in an Object

Marshaling

The COM technique of marshaling allows interfaces exposed by an object in one process to be used in another process. In marshaling, COM provides code (or uses code provided by the interface implementor) both to pack a method's parameters into a format that can be moved across processes (as well as, across the wire to processes running on other machines) and to unpack those parameters at the other end. Likewise, COM must perform these same steps on the return from the call.

Note
Marshaling is typically not necessary when an interface provided by an object is being used in the same process as the object. However, marshaling may be needed between threads.

See Also

Reference

Introduction to COM

Marshaling Details

Aggregation

There are times when an object's implementor would like to take advantage of the services offered by another, prebuilt object. Furthermore, it would like this second object to appear as a natural part of the first. COM achieves both of these goals through containment and aggregation.

Aggregation means that the containing (outer) object creates the contained (inner) object as part of its creation process and the interfaces of the inner object are exposed by the outer. An object allows itself to be aggregatable or not. If it is, then it must follow certain rules for aggregation to work properly.

Primarily, all **IUnknown** method calls on the contained object must delegate to the containing object.

See Also

Reference

Introduction to COM

Reusing Objects

Introduction to ATL

ATL is the Active Template Library, a set of template-based C++ classes with which you can easily create small, fast Component Object Model (COM) objects. It has special support for key COM features including: stock implementations of IUnknown, IClassFactory, IClassFactory2, and **IDispatch**; dual interfaces; standard COM enumerator interfaces; connection points; tear-off interfaces; and ActiveX controls. For more information, see Implementing the IDispatch Interface.

ATL code can be used to create single-threaded objects, apartment-model objects, free-threaded model objects, or both free-threaded and apartment-model objects.

Topics covered in this section include:

- How a template library differs from a standard C++ library.
- What you can and cannot do with ATL.
- Recommendations for choosing between ATL and MFC.
- ATL and Visual C++ version numbers.

See Also

Other Resources

Introduction to COM and ATL

Using a Template Library

A template is somewhat like a macro. As with a macro, invoking a template causes it to expand (with appropriate parameter substitution) to code you have written. However, a template goes further than this to allow the creation of new classes based on types that you pass as parameters. These new classes implement type-safe ways of performing the operation expressed in your template code.

Template libraries such as ATL differ from traditional C++ class libraries in that they are typically supplied only as source code (or as source code with a little, supporting run time) and are not inherently or necessarily hierarchical in nature. Rather than deriving from a class to get the functionality you desire, you instantiate a class from a template.

See Also

Reference

Introduction to ATL

The Scope of ATL

ATL allows you to easily create COM objects, Automation servers, and ActiveX controls. ATL provides built-in support for many of the fundamental COM interfaces.

ATL is shipped as source code which you include in your application. ATL also makes a DLL available (atl80.dll), which contains code that can be shared across components. However, this DLL is not necessary.

See Also

Reference

Introduction to ATL

ATL Project Wizard

Recommendations for Choosing Between ATL and MFC

When developing components and applications, you can choose between two approaches — ATL and MFC (the Microsoft Foundation Class Library).

Using ATL

ATL is a fast, easy way to both create a COM component in C++ and maintain a small footprint. Use ATL to create a control if you don't need all of the built-in functionality that MFC automatically provides.

Using MFC

MFC allows you to create full applications, ActiveX controls, and active documents. If you have already created a control with MFC, you may want to continue development in MFC. When creating a new control, consider using ATL if you don't need all of MFC's built-in functionality.

Using ATL in an MFC Project

You can add support for using ATL in an existing MFC project by running a wizard. For details, see Adding ATL Support to Your MFC Project.

See Also

Reference

Introduction to ATL

Fundamentals of ATL COM Objects

The following illustration depicts the relationship among the various classes and interfaces used in defining an ATL COM object.

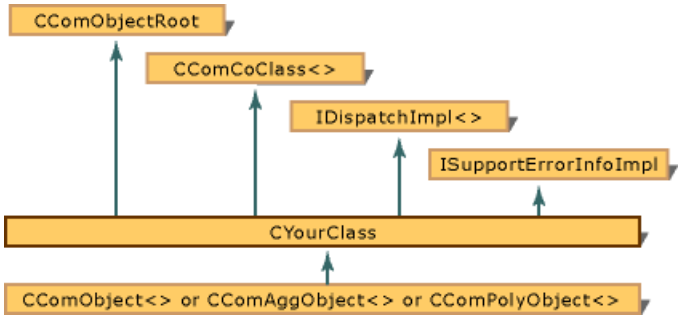

ATL implements IUnknown in two phases:

- CComObject, CComAggObject, or CComPolyObject implements the **IUnknown** methods.
- CComObjectRoot or CComObjectRootEx manages the reference count and outer pointers of **IUnknown**.

Other aspects of your ATL COM object are handled by other classes:

- CComCoClass defines the object's default class factory and aggregation model.
- IDispatchImpl provides a default implementation of the IDispatch Interface portion of any dual interfaces on the object.
- ISupportErrorInfoImpl implements the **ISupportErrorInfo** interface that ensures error information can be propagated up the call chain correctly.

In This Section

Implementing CComObjectRootEx

Show example COM map entries for implementing **CComObjectRootEx**.

Implementing CComObject, CComAggObject, and CComPolyObject

Discusses how the **DECLARE_*_AGGREGATABLE** macros affect the use of **CComObject**, **CComAggObject**, and **CComPolyObject**.

Supporting IDispatch and IErrorInfo

Lists the ATL implementation classes to use for supporting the **IDispatch** and **IErrorInfo** interfaces.

Supporting IDispEventImpl

Discusses the steps to implement a connection point for your class.

Changing the Default Class Factory and Aggregation Model

Show what macros to use to change the default class factory and aggregation model.

Creating an Aggregated Object

Lists the steps for creating an aggregated object.

Related Sections

Creating an ATL Project

Provides information about creating an ATL COM object.

ATL

Provides links to conceptual topics on how to program using the Active Template Library.

See Also

Other Resources

ATL Concepts

Implementing CComObjectRootEx

CComObjectRootEx is essential; all ATL objects must have one instance of **CComObjectRootEx** or CComObjectRoot in their inheritance. **CComObjectRootEx** provides the default **QueryInterface** mechanism based on COM map entries.

Through its COM map, an object's interfaces are exposed to a client when the client queries for an interface. The query is performed through **CComObjectRootEx::InternalQueryInterface**. **InternalQueryInterface** only handles interfaces in the COM map table.

You can enter interfaces into the COM map table with the COM_INTERFACE_ENTRY macro or one of its variants. For example, the following code from the BEEPER sample enters the interfaces `IDispatch`, `IBeeper`, and `ISupportErrorInfo` into the COM map table:

```
BEGIN_COM_MAP(CBeeper)
   COM_INTERFACE_ENTRY(IDispatch)
   COM_INTERFACE_ENTRY(IBeeper)
   COM_INTERFACE_ENTRY_TEAR_OFF(IID_ISupportErrorInfo, CBeeper2)
END_COM_MAP( )
```

See Also

Reference

Fundamentals of ATL COM Objects

Other Resources

COM Map Macros

Implementing CComObject, CComAggObject, and CComPolyObject

The template classes CComObject, CComAggObject, and CComPolyObject are always the most derived classes in the inheritance chain. It is their responsibility to handle all of the methods in **IUnknown**: **QueryInterface**, **AddRef**, and **Release**. In addition, **CComAggObject** and **CComPolyObject** (when used for aggregated objects) provide the special reference counting and **QueryInterface** semantics required for the inner unknown.

Whether **CComObject**, **CComAggObject**, or **CComPolyObject** is used depends on whether you declare one (or none) of the following macros:

Macro	Effect
DECLARE_NOT_AGGREGATABLE	Always uses **CComObject**.
DECLARE_AGGREGATABLE	Uses **CComAggObject** if the object is aggregated and **CComObject** if it is not. **CComCoClass** contains this macro so if none of the **DECLARE_*_AGGREGATABLE** macros are declared in your class, this will be the default.
DECLARE_ONLY_AGGREGATABLE	Always uses **CComAggObject**. Returns an error if the object is not aggregated.
DECLARE_POLY_AGGREGATABLE	ATL creates an instance of **CComPolyObject<CYourClass>** when **IClassFactory::CreateInstance** is called. During creation, the value of the outer unknown is checked. If it is **NULL**, **IUnknown** is implemented for a nonaggregated object. If the outer unknown is not **NULL**, **IUnknown** is implemented for an aggregated object.

The advantage of using **CComAggObject** and **CComObject** is that the implementation of **IUnknown** is optimized for the kind of object being created. For instance, a nonaggregated object only needs a reference count, while an aggregated object needs both a reference count for the inner unknown and a pointer to the outer unknown.

The advantage of using **CComPolyObject** is that you avoid having both **CComAggObject** and **CComObject** in your module to handle the aggregated and nonaggregated cases. A single **CComPolyObject** object handles both cases. This means only one copy of the vtable and one copy of the functions exist in your module. If your vtable is large, this can substantially decrease your module size. However, if your vtable is small, using **CComPolyObject** can result in a slightly larger module size because it is not optimized for an aggregated or nonaggregated object, as are **CComAggObject** and **CComObject**.

See Also

Reference

Fundamentals of ATL COM Objects

Other Resources

Aggregation and Class Factory Macros

Supporting IDispatch and IErrorInfo

You can use the template class IDispatchImpl to provide a default implementation of the IDispatch Interface portion of any dual interfaces on your object. For more information, see Implementing the IDispatch Interface.

If your object uses the IErrorInfo interface to report errors back to the client, then your object must support the ISupportErrorInfo Interface interface. The template class ISupportErrorInfoImpl provides an easy way to implement this if you only have a single interface that generates errors on your object.

See Also

Reference

Fundamentals of ATL COM Objects

Supporting IDispEventImpl

The template class IDispEventImpl can be used to provide support for connection point sinks in your ATL class. A connection point sink allows your class to handle events fired from external COM objects. These connection point sinks are mapped with an event sink map, provided by your class.

To properly implement a connection point sink for your class, the following steps must be completed:

- Import the type libraries for each external object
- Declare the **IDispEventImpl** interfaces
- Declare an event sink map
- Advise and unadvise the connection points

The steps involved in implementing a connection point sink are all accomplished by modifying only the header file (.h) of your class.

Importing the Type Libraries

For each external object whose events you want to handle, you must import the type library. This step defines the events that can be handled and provides information that is used when declaring the event sink map. The #import directive can be used to accomplish this. Add the necessary **#import** directive lines for each dispatch interface you will support to the header file (.h) of your class.

The following example imports the type library of an external COM server (`MyServer`):

```
#import "D:\MyServer.dll" raw_interfaces_only, no_namespace, named_guids
```

Note
You must have a separate **#import** statement for each external type library you will support.

Declaring the IDispEventImpl Interfaces

Now that you have imported the type libraries of each dispatch interface, you need to declare separate **IDispEventImpl** interfaces for each external dispatch interface. Modify the declaration of your class by adding an **IDispEventImpl** interface declaration for each external object. For more information on the parameters, see IDispEventImpl.

The following code declares two connection point sinks, for the custom `IExtEvents1` and `IExtEvents2` interfaces, for the COM object implemented by class `CMyObj`:

```
public IDispEventImpl<0, CMyObj, &DIID__IExtEvents1,
    &LIBID_EXTEVENTS1Lib, 1, 0>,
public IDispEventImpl<1, CMyObj, &DIID__IExtEvents2,
    &LIBID_EXTEVENTS2Lib, 1, 0>
```

Declaring an Event Sink Map

In order for the event notifications to be handled by the proper function, your class must route each event to its correct handler. This is achieved by declaring an event sink map.

ATL provides several macros, BEGIN_SINK_MAP, END_SINK_MAP, and SINK_ENTRY, that make this mapping easier. The standard format is as follows:

```
BEGIN_SINK_MAP(comClass)
   SINK_ENTRY(id, dispid, func)
   . . . //additional external event entries
END_SINK_MAP()
```

The following example declares an event sink map with two event handlers:

```
BEGIN_SINK_MAP(CMyObj)
   SINK_ENTRY(0, Events1, OnClick1)
   SINK_ENTRY(0, Events2, OnClick2)
END_SINK_MAP()
```

The implementation is nearly complete. The last step concerns the advising and unadvising of the external interfaces.

Advising and Unadvising the IDispEventImpl Interfaces

The final step is to implement a method that will advise (or unadvise) all connection points at the proper times. This advising must be done before communication between the external clients and your object can take place. Before your object becomes visible, each external dispatch interface supported by your object is queried for outgoing interfaces. A connection is established and a reference to the outgoing interface is used to handle events from the object. This procedure is referred to as "advising."

After your object is finished with the external interfaces, the outgoing interfaces should be notified that they are no longer used by your class. This process is referred to as "unadvising."

Because of the unique nature of COM objects, this procedure varies, in detail and execution, between implementations. These details are beyond the scope of this topic and are not addressed.

See Also

Reference

Fundamentals of ATL COM Objects

Changing the Default Class Factory and Aggregation Model

ATL uses CComCoClass to define the default class factory and aggregation model for your object. **CComCoClass** specifies the following two macros:

- DECLARE_CLASSFACTORY Declares the class factory to be CComClassFactory.
- DECLARE_AGGREGATABLE Declares that your object can be aggregated.

You can override either of these defaults by specifying another macro in your class definition. For example, to use CComClassFactory2 instead of **CComClassFactory**, specify the DECLARE_CLASSFACTORY2 macro:

```
class CMyClass : ...,
   public CComCoClass<CMyClass, &CLSID_CMyClass>
{
public:
   DECLARE_CLASSFACTORY2(CMyLicense)

   ...
};
```

Two other macros that define a class factory are DECLARE_CLASSFACTORY_AUTO_THREAD and DECLARE_CLASSFACTORY_SINGLETON.

ATL also uses the **typedef** mechanism to implement default behavior. For example, the **DECLARE_AGGREGATABLE** macro uses **typedef** to define a type called **_CreatorClass**, which is then referenced throughout ATL. Note that in a derived class, a **typedef** using the same name as the base class's **typedef** results in ATL using your definition and overriding the default behavior.

See Also

Reference

Fundamentals of ATL COM Objects

Other Resources

Aggregation and Class Factory Macros

Creating an Aggregated Object

Aggregation delegates **IUnknown** calls, providing a pointer to the outer object's **IUnknown** to the inner object.

To create an aggregated object

1. Add an **IUnknown** pointer to your class object and initialize it to **NULL** in the constructor.
2. Override FinalConstruct to create the aggregate.
3. Use the **IUnknown** pointer, defined in Step 1, as the second parameter for the COM_INTERFACE_ENTRY_AGGREGATE macros.
4. Override FinalRelease to release the **IUnknown** pointer.

Note
If you use and release an interface from the aggregated object during **FinalConstruct**, you should add the DECLARE_PROTECT_FINAL_CONSTRUCT macro to the definition of your class object.

See Also

Reference

Fundamentals of ATL COM Objects

Dual Interfaces and ATL

A dual interface allows its methods to be accessed as dispinterface methods or as vtable methods. This section covers some of the features of dual interfaces from an ATL perspective.

In This Section

Implementing a Dual Interface

Discusses the classes and wizards involved in implementing a dual interface.

Multiple Dual Interfaces

Discusses how to expose multiple dual interfaces on a single object.

The nonextensible Attribute

Discusses when to use the **nonextensible** attribute on your interface definition.

Dual Interfaces and Events

Discusses design reasons for not making an event interface a dual interface.

Related Sections

ATL

Provides links to conceptual topics on how to program using the Active Template Library.

See Also

Other Resources

ATL Concepts

Implementing a Dual Interface

You can implement a dual interface using the IDispatchImpl class, which provides a default implementation of the **IDispatch** methods in a dual interface. For more information, see Implementing the IDispatch Interface.

To use this class:

- Define your dual interface in a type library.
- Derive your class from a specialization of **IDispatchImpl** (pass information about the interface and type library as the template arguments).
- Add an entry (or entries) to the COM map to expose the dual interface through **QueryInterface**.
- Implement the vtable part of the interface in your class.
- Ensure that the type library containing the interface definition is available to your objects at run time.

ATL Simple Object Wizard

If you want to create a new interface and a new class to implement it, you can use the ATL Add Class dialog box, and then the ATL Simple Object Wizard.

Implement Interface Wizard

If you have an existing interface, you can use the Implement Interface Wizard to add the necessary base class, COM map entries, and skeleton method implementations to an existing class.

Note
You may need to adjust the generated base class so that the major and minor version numbers of the type library are passed as template arguments to your **IDispatchImpl** base class. The Implement Interface Wizard doesn't check the type library version number for you.

Implementing IDispatch

You can use an **IDispatchImpl** base class to provide an implementation of a dispinterface just by specifying the appropriate entry in the COM map (using the COM_INTERFACE_ENTRY2 or COM_INTERFACE_ENTRY_IID macro) as long as you have a type library describing a corresponding dual interface. It is quite common to implement the **IDispatch** interface this way, for example. The ATL Simple Object Wizard and Implement Interface Wizard both assume that you intend to implement **IDispatch** in this way, so they will add the appropriate entry to the map.

Note
ATL offers the IDispEventImpl and IDispEventSimpleImpl classes to help you implement dispinterfaces without requiring a type library containing the definition of a compatible dual interface.

See Also

Concepts

Dual Interfaces and ATL

Multiple Dual Interfaces

You may want to combine the advantages of a dual interface (that is, the flexibility of both vtable and late binding, thus making the class available to scripting languages as well as C++) with the techniques of multiple inheritance.

Although it is possible to expose multiple dual interfaces on a single COM object, it is not recommended. If there are multiple dual interfaces, there must be only one **IDispatch** interface exposed. The techniques available to ensure that this is the case carry penalties such as loss of function or increased code complexity. The developer considering this approach should carefully weigh the advantages and disadvantages.

Exposing a Single IDispatch Interface

It is possible to expose multiple dual interfaces on a single object by deriving from two or more specializations of **IDispatchImpl**. However, if you allow clients to query for the **IDispatch** interface, you will need to use the COM_INTERFACE_ENTRY2 macro (or COM_INTERFACE_ENTRY_IID) to specify which base class to use for the implementation of **IDispatch**.

```
COM_INTERFACE_ENTRY2(IDispatch, IMyDualInterface)
```

Because only one **IDispatch** interface is exposed, clients that can only access your objects through the **IDispatch** interface will not be able to access the methods or properties in any other interface.

Combining Multiple Dual Interfaces into a Single Implementation of IDispatch

ATL does not provide any support for combining multiple dual interfaces into a single implementation of **IDispatch**. However, there are several known approaches to manually combining the interfaces, such as creating a templated class that contains a union of the separate **IDispatch** interfaces, creating a new object to perform the **QueryInterface** function, or using a typeinfo-based implementation of nested objects to create the **IDispatch** interface.

These approaches have problems with potential namespace collisions, as well as code complexity and maintainability. It is not recommended that you create multiple dual interfaces.

See Also

Concepts

Dual Interfaces and ATL

The nonextensible Attribute

If a dual interface will not be extended at run time (that is, you won't provide methods or properties via **IDispatch::Invoke** that are not available via the vtable), you should apply the **nonextensible** attribute to your interface definition. This attribute provides information to client languages (such as Visual Basic) that can be used to enable full code verification at compile time. If this attribute is not supplied, bugs may remain hidden in the client code until run time.

For more information on the **nonextensible** attribute and an example, see nonextensible.

See Also

Concepts

Dual Interfaces and ATL

Dual Interfaces and Events

While it is possible to design an event interface as a dual, there are a number of good design reasons not to do so. The fundamental reason is that the source of the event will only fire the event via the vtable or via **Invoke**, not both. If the event source fires the event as a direct vtable method call, the **IDispatch** methods will never be used and it's clear that the interface should have been a pure vtable interface. If the event source fires the event as a call to **Invoke**, the vtable methods will never be used and it's clear that the interface should have been a dispinterface. If you define your event interfaces as duals, you'll be requiring clients to implement part of an interface that will never be used.

Note
This argument does not apply to dual interfaces, in general. From an implementation perspective, duals are a quick, convenient, and well-supported way of implementing interfaces that are accessible to a wide range of clients.

There are further reasons to avoid dual event interfaces; neither Visual Basic nor Internet Explorer support them.

See Also

Concepts

Dual Interfaces and ATL

ATL Collections and Enumerators

A *collection* is a COM object that provides an interface that allows access to a group of data items (raw data or other objects). An interface that follows the standards for providing access to a group of objects is known as a *collection interface*.

At a minimum, collection interfaces must provide a **Count** property that returns the number of items in the collection, an **Item** property that returns an item from the collection based on an index, and a **_NewEnum** property that returns an enumerator for the collection. Optionally, collection interfaces can provide **Add** and **Remove** methods to allow items to be inserted into or deleted from the collection, and a **Clear** method to remove all items.

An *enumerator* is a COM object that provides an interface for iterating through items in a collection. Enumerator interfaces provide serial access to the elements of a collection via four required methods: **Next**, **Skip**, **Reset**, and **Clone**.

You can learn more about enumerator interfaces by reading about the archetypal (but entirely imaginary) IEnumXXXX interface.

In This Section

ATL Collection and Enumerator Classes

Briefly describes and provides links to the ATL classes that will help you implement collections and enumerators.

Design Principles for Collection and Enumerator Interfaces

Discusses the different design principles behind each type of interface.

Implementing an STL-Based Collection

An extended example that walks you through the implementation of a Standard Template Library (STL)-based collection.

Related Sections

ATL

Provides links to conceptual topics on how to program using the Active Template Library.

ATLCollections Sample

A sample that demonstrates the use of **ICollectionOnSTLImpl** and **CComEnumOnSTL**, and the implementation of custom copy policy classes.

See Also

Other Resources

ATL Concepts

ATL Collection and Enumerator Classes

ATL provides the following classes to help you implement collections and enumerators.

Class	Description
ICollectionOnSTLImpl	Collection interface implementation
IEnumOnSTLImpl	Enumerator interface implementation (assumes data stored in an STL-compatible container)
CComEnumImpl	Enumerator interface implementation (assumes data stored in an array)
CComEnumOnSTL	Enumerator object implementation (uses **IEnumOnSTLImpl**)
CComEnum	Enumerator object implementation (uses **CComEnumImpl**)
_Copy	Copy policy class
_CopyInterface	Copy policy class
CAdapt	Adapter class (hides **operator &** allowing **CComPtr**, **CComQIPtr**, and **CComBSTR** to be stored in STL c ontainers)

See Also

Concepts

ATL Collections and Enumerators

Design Principles for Collection and Enumerator Interfaces

There are different design principles behind each type of interface:

- A collection interface provides *random* access to a *single* item in the collection via the **Item** method, it lets clients discover how many items are in the collection via the **Count** property, and often allows clients to add and remove items.
- An enumerator interface provides *serial* access to *multiple* items in a collection, it doesn't allow the client to discover how many items are in the collection (until the enumerator stops returning items), and it doesn't provide any way of adding or removing items.

Each type of interface plays a different role in providing access to the elements in a collection.

See Also

Concepts

ATL Collections and Enumerators

Implementing an STL-Based Collection

ATL provides the **ICollectionOnSTLImpl** interface to enable you to quickly implement Standard Template Library (STL)-based collection interfaces on your objects. To understand how this class works, you will work through a simple example (below) that uses this class to implement a read-only collection aimed at Automation clients.

The sample code is from the ATLCollections sample.

To complete this procedure, you will:

- Generate a new Simple Object.
- Edit the IDL file for the generated interface.
- Create five typedefs describing how the collection items are stored and how they will be exposed to clients via COM interfaces.
- Create two typedefs for copy policy classes.
- Create typedefs for the enumerator and collection implementations.
- Edit the wizard-generated C++ code to use the collection typedef.
- Add code to populate the collection.

Generating a New Simple Object

Create a new project, ensuring that the Attributes box under Application Settings is cleared. Use the ATL Add Class dialog box and Add Simple Object Wizard to generate a Simple Object called `Words`. Make sure that a dual interface called `IWords` is generated. Objects of the generated class will be used to represent a collection of words (that is, strings).

Editing the IDL File

Now, open the IDL file and add the three properties necessary to turn `IWords` into a read-only collection interface, as shown below:

```
[
    object,
    uuid(0D44F689-B373-11D2-9A7F-50F653C10000),
    dual,                                           // (1)
    pointer_default(unique),
    nonextensible                                   // (2)
]
interface IWords : IDispatch
{
    [id(DISPID_NEWENUM), propget]                   // (3)
    HRESULT _NewEnum([out, retval] IUnknown** ppUnk);

    [id(DISPID_VALUE), propget]                     // (4)
    HRESULT Item(
                    [in] long Index,
                    [out, retval] BSTR* pVal);      // (5)

    [id(0x00000001), propget]                       // (6)
    HRESULT Count([out, retval] long* pVal);
};
```

This is the standard form for a read-only collection interface designed with Automation clients in mind. The numbered comments in this interface definition correspond to the comments below:

1. Collection interfaces are usually dual because Automation clients accesses the **_NewEnum** property via **IDispatch::Invoke**. However, Automation clients can access the remaining methods via the vtable, so dual interfaces are preferable to dispinterfaces.
2. If a dual interface or dispinterface will not be extended at run time (that is, you won't provide extra methods or properties via **IDispatch::Invoke**), you should apply the **nonextensible** attribute to your definition. This attribute enables

Automation clients to perform full code verification at compile time. In this case, the interface should not be extended.

3. The correct DISPID is important if you want Automation clients to be able to use this property. (Note that there is only one underscore in **DISPID_NEWENUM**.)

4. You can supply any value as the DISPID of the **Item** property. However, **Item** typically uses **DISPID_VALUE** to make it the default property of the collection. This allows Automation clients to refer to the property without naming it explicitly.

5. The data type used for the return value of the **Item** property is the type of the item stored in the collection as far as COM clients are concerned. The interface returns strings, so you should use the standard COM string type, **BSTR**. You can store the data in a different format internally as you'll see shortly.

6. The value used for the DISPID of the **Count** property is completely arbitrary. There's no standard DISPID for this property.

Creating Typedefs for Storage and Exposure

Once the collection interface is defined, you need to decide how the data will be stored, and how the data will be exposed via the enumerator.

The answers to these questions can be provided in the form of a number of typedefs, which you can add near the top of the header file for your newly created class:

```
    // Store the data in a vector of std::strings
    typedef std::vector< std::string >         ContainerType;

    // The collection interface exposes the data as BSTRs
    typedef BSTR                               CollectionExposedType;
    typedef IWords                             CollectionInterface;

    // Use IEnumVARIANT as the enumerator for VB compatibility
    typedef VARIANT                            EnumeratorExposedType;
    typedef IEnumVARIANT                       EnumeratorInterface;
```

In this case, you will store the data as a **std::vector** of **std::string**s. **std::vector** is an STL container class that behaves like a managed array. **std::string** is the Standard C++ Library's string class. These classes make it easy to work with a collection of strings.

Since Visual Basic support is vital to the success of this interface, the enumerator returned by the **_NewEnum** property must support the **IEnumVARIANT** interface. This is the only enumerator interface understood by Visual Basic.

Creating Typedefs for Copy Policy Classes

The typedefs you have created so far provide all the information you need to create further typedefs for the copy classes that will be used by the enumerator and collection:

```
    // Typedef the copy classes using existing typedefs
    typedef VCUE::GenericCopy<EnumeratorExposedType, ContainerType::value_type>    Enumerat
orCopyType;
    typedef VCUE::GenericCopy<CollectionExposedType, ContainerType::value_type>    Collecti
onCopyType;
```

In this example, you can use the custom `GenericCopy` class defined in VCUE_Copy.h and VCUE_CopyString.h from the ATLCollections sample. You can use this class in other code, but you may need to define further specializations of `GenericCopy` to support data types used in your own collections. For more information, see ATL Copy Policy Classes.

Creating Typedefs for Enumeration and Collection

Now all the template parameters necessary to specialize the **CComEnumOnSTL** and **ICollectionOnSTLImpl** classes for this situation have been provided in the form of typedefs. To simplify the use of the specializations, create two more typedefs as shown below:

```
    typedef CComEnumOnSTL< EnumeratorInterface, &__uuidof(EnumeratorInterface), EnumeratorE
xposedType, EnumeratorCopyType, ContainerType >    EnumeratorType;

    typedef ICollectionOnSTLImpl< CollectionInterface, ContainerType, CollectionExposedType
```

```
, CollectionCopyType, EnumeratorType >    CollectionType;
```

Now `CollectionType` is a synonym for a specialization of **ICollectionOnSTLImpl** that implements the `IWords` interface defined earlier and provides an enumerator that supports **IEnumVARIANT**.

Editing the Wizard-Generated Code

Now you must derive `CWords` from the interface implementation represented by the `CollectionType` typedef rather than `IWords`, as shown below:

```
class ATL_NO_VTABLE CWords :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CWords, &CLSID_Words>,
    public IDispatchImpl<CollectionType, &IID_IWords, &LIBID_ATLCOLLECTIONSLib>
{
public:
    DECLARE_REGISTRY_RESOURCEID(IDR_STRINGCOLLECTION)
    DECLARE_PROTECT_FINAL_CONSTRUCT()

BEGIN_COM_MAP(CWords)
    COM_INTERFACE_ENTRY(IWords)
    COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()

// IWords
public:
};
```

Adding Code to Populate the Collection

The only thing that remains is to populate the vector with data. In this simple example, you can add a few words to the collection in the constructor for the class:

```
CWords()
{
    m_coll.push_back("this");
    m_coll.push_back("is");
    m_coll.push_back("a");
    m_coll.push_back("test");
}
```

Now, you can test the code with the client of your choice.

See Also

Tasks

ATLCollections Sample: Demonstrates ICollectionOnSTLImpl, CComEnumOnSTL, and Custom Copy Policy Classes

Concepts

ATL Collections and Enumerators

ATL Copy Policy Classes

ATL Copy Policy Classes

Copy policy classes are utility classes used to initialize, copy, and delete data. Copy policy classes allow you to define copy semantics for any type of data, and to define conversions between different data types.

ATL uses copy policy classes in its implementations of the following templates:

- CComEnumImpl
- IEnumOnSTLImpl
- ICollectionOnSTLImpl

By encapsulating the information needed to copy or convert data in a copy policy class that can be passed as a template argument, the ATL developers have provided for extreme reusability of these classes. For example, if you need to implement a collection using any arbitrary data type, all you need to provide is the appropriate copy policy; you never have to touch the code that implements the collection.

Definition

By definition, a class that provides the following static functions is a copy policy class:

```
static void init(DestinationType* p);
static HRESULT copy(DestinationType* pTo, const SourceType* pFrom);
static void destroy(DestinationType* p);
```

You can replace the types *DestinationType* and *SourceType* with arbitrary data types for each copy policy.

Note
Although you can define copy policy classes for any arbitrary data types, use of the classes in ATL code should limit the types that make sense. For example, when using a copy policy class with ATL's collection or enumerator implementations, *DestinationType* must be a type that can be used as a parameter in a COM interface method.

Use **init** to initialize data, **copy** to copy data, and **destroy** to free the data. The precise meaning of initialization, copying, and destruction are the domain of the copy policy class and will vary depending on the data types involved.

There are two requirements on the use and implementation of a copy policy class:

- The first parameter to **copy** must only receive a pointer to data that you have previously initialized using **init**.
- **destroy** must only ever receive a pointer to data that you have previously initialized using **init** or copied via **copy**.

Standard Implementations

ATL provides two copy policy classes in the form of the **_Copy** and **_CopyInterface** template classes:

- The **_Copy** class allows homogeneous copying only (not conversion between data types) since it only offers a single template parameter to specify both *DestinationType* and *SourceType*. The generic implementation of this template contains no initialization or destruction code and uses **memcpy** to copy the data. ATL also provides specializations of **_Copy** for **VARIANT**, **LPOLESTR**, **OLEVERB**, and **CONNECTDATA** data types.
- The **_CopyInterface** class provides an implementation for copying interface pointers following standard COM rules. Once again this class allows only homogeneous copying, so it uses simple assignment and a call to **AddRef** to perform the copy.

Custom Implementations

Typically, you'll need to define your own copy policy classes for heterogeneous copying (that is, conversion between data types). For some examples of custom copy policy classes, look at the files VCUE_Copy.h and VCUE_CopyString.h in the ATLCollections sample. These files contain two template copy policy classes, `GenericCopy` and `MapCopy`, plus a number of specializations of `GenericCopy` for different data types.

GenericCopy

`GenericCopy` allows you to specify the *SourceType* and *DestinationType* as template arguments. Here's the most general form of the `GenericCopy` class from VCUE_Copy.h:

```
template <class DestinationType, class SourceType = DestinationType>
class GenericCopy
{
public :
    typedef DestinationType   destination_type;
    typedef SourceType        source_type;

    static void init(destination_type* p)
    {
        _Copy<destination_type>::init(p);
    }
    static void destroy(destination_type* p)
    {
        _Copy<destination_type>::destroy(p);
    }
    static HRESULT copy(destination_type* pTo, const source_type* pFrom)
    {
        return _Copy<destination_type>::copy(pTo, const_cast<source_type*>(pFrom));
    }

}; // class GenericCopy
```

VCUE_Copy.h also contains the following specializations of this class: `GenericCopy<BSTR>`, `GenericCopy<VARIANT, BSTR>`, `GenericCopy<BSTR, VARIANT>`. VCUE_CopyString.h contains specializations for copying from **std::string**s: `GenericCopy<std::string>`, `GenericCopy<VARIANT, std::string>`, and `GenericCopy<BSTR, std::string>`. You could enhance `GenericCopy` by providing further specializations of your own.

MapCopy

`MapCopy` assumes that the data being copied will be stored in an STL-style map, so it allows you to specify the type of map in which the data is stored and the destination type. The implementation of the class just uses the typedefs supplied by the *MapType* class to determine the type of the source data and to call the appropriate `GenericCopy` class. No specializations of this class are needed.

```
template <class MapType, class DestinationType = MapType::referent_type>
class MapCopy
{
public :
    typedef DestinationType           destination_type;
    typedef MapType::value_type       source_type;

    typedef MapType                   map_type;
    typedef MapType::referent_type    pseudosource_type;

    static void init(destination_type* p)
    {
        GenericCopy<destination_type, pseudosource_type>::init(p);
    }
    static void destroy(destination_type* p)
    {
        GenericCopy<destination_type, pseudosource_type>::destroy(p);
    }
    static HRESULT copy(destination_type* pTo, const source_type* pFrom)
    {
        return GenericCopy<destination_type, pseudosource_type>::copy(pTo, &(pFrom->second)
);
    }

}; // class MapCopy
```

See Also

Tasks

ATLCollections Sample: Demonstrates ICollectionOnSTLImpl, CComEnumOnSTL, and Custom Copy Policy Classes

Composite Control Fundamentals

A composite control is a type of ActiveX control that can contain (similar to a dialog box) other ActiveX controls or Windows controls. Once the composite control is built, it can be inserted anywhere an ActiveX control can be hosted.

The ATL Project Wizard and **Add Class** dialog box automate the process of creating and implementing a composite control project, similar to the result of running the Application Wizard to create an MFC application framework. The development process consists of five steps:

- Creating an ATL project
- Inserting a composite control
- Modifying the ATL project
- Adding functionality to the composite control
- Building and testing the ATL project

See Also

Other Resources

ATL Concepts
Composite Control Global Functions
Composite Control Macros

Inserting a Composite Control

The **Add Class** dialog box allows you to insert an ATL object into a project. Access this dialog box by right-clicking the project name in Solution Explorer, pointing to **Add**, and then clicking **Add Class**.

In the **Add Class** dialog box, choose **ATL Control**. This will start the ATL Control Wizard. To create a composite control, select the **Options** tab, and click the **Composite control** check box.

A default HTML page will be created for viewing the control.

See Also

Concepts

Composite Control Fundamentals

Modifying the ATL Project

At this point, your composite control project implements the necessary objects for your composite control. The next step is to add any controls that the composite control will contain and handle any necessary events.

To add additional ActiveX or Windows controls, add a new resource script and then use the Dialog editor. For more information on adding controls (and related tasks), see Dialog Editor.

To handle any necessary events from the ActiveX controls, see Adding Functionality to the Composite Control.

See Also

Tasks

Creating a New Resource Script File

Concepts

Composite Control Fundamentals

Adding Functionality to the Composite Control

Once you have inserted any necessary controls into the composite control, the next step involves adding new functionality. This new functionality usually falls into two categories:

- Supporting additional interfaces and customizing the behavior of your composite control with additional, specific features.
- Handling events from the contained ActiveX control (or controls).

For the purpose and scope of this article, the remainder of this section focuses solely on handling events from ActiveX controls.

Note
If you need to handle messages from Windows controls, see Implementing a Window for more information on message handling in ATL.

After inserting an ActiveX control in the dialog resource, right-click the control and click **Add Event Handler**. Select the event you want to handle and click **Add and Edit**. The event handler code will be added to the control's .h file.

Connection points for ActiveX controls on the composite control are automatically connected and disconnected via calls to CComCompositeControl::AdviseSinkMap.

See Also

Concepts

Composite Control Fundamentals

Building and Testing the ATL Project

As mentioned in Inserting a Composite Control, one of the initial components of the project is a default HTML page that hosts your new composite control. After you finish modifying the composite control, click **Build Solution** or **Rebuild Solution** from the **Build** menu. Once the project successfully builds, load the HTML page, located in the root directory of your composite control project, into Internet Explorer or another browser and test the functionality of your control.

You can also test your composite control using the Test Container tool, or any other application that can host an ActiveX control. See Testing Properties and Events with Test Container for information on how to access the test container.

See Also

Concepts

Composite Control Fundamentals

ATL Control Containment FAQ

This section covers questions related to hosting controls with ATL.

Note
In Visual C++ .NET, the window class **"AtlAxWin"** is renamed to **"AtlAxWin80"**. You can still use **"AtlAxWin"** if you use the hosting code in atl.dll (shipped with Visual C++ 6.0 and now a system .dll).

Classes and APIs

- Which ATL Classes Facilitate ActiveX Control Containment?
- What Is the ATL Control-Hosting API?
- What Is "AtlAxWin80"?

Creating and Loading Controls

- How Do I Create a Control Instance?
- How Do I Load Controls Specified at Run Time?
- How Do I Load Controls Specified on a Dialog Resource?
- How Do I Load a Licensed Control?
- How Do I Deal with Licensed Controls?
- When Do I Need to Call AtlAxWinInit?

Working with Controls

- What Is a Host Object?
- Can I Host More Than One Control in a Single Window?
- Can I Reuse a Host Window?
- How Do I Resize a Control?
- How Do I Get an Interface Pointer to a Control?
- How Do I Set Ambient Properties for My Controls?
- Hosting ActiveX Controls Using ATL AXHost

Destroying Controls

- How Do I Destroy a Control?
- When Do I Need to Call AtlAxWinTerm?

See Also

Other Resources

ATL Concepts

Which ATL Classes Facilitate ActiveX Control Containment?

ATL's control-hosting code doesn't require you to use any ATL classes; you can simply create an **"AtlAxWin80"** window and use the control-hosting API if necessary (for more information, see What Is the ATL Control-Hosting API?). However, the following classes make the containment features easier to use.

Class	Description
CAxWindow	Wraps an **"AtlAxWin80"** window, providing methods for creating the window, creating a control and/or attaching a control to the window, and retrieving interface pointers on the host object.
CAxWindow2T	Wraps an **"AtlAxWinLic80"** window, providing methods for creating the window, creating a control and/or attaching a licensed control to the window, and retrieving interface pointers on the host object.
CComCompositeControl	Acts as a base class for ActiveX control classes based on a dialog resource. Such controls can contain other ActiveX controls.
CAxDialogImpl	Acts as a base class for dialog classes based on a dialog resource. Such dialogs can contain ActiveX controls.
CWindow	Provides a method, GetDlgControl, that will return an interface pointer on a control, given the ID of its host window. In addition, the Windows API wrappers exposed by **CWindow** generally make window management easier.

See Also

Concepts

ATL Control Containment FAQ

What Is the ATL Control-Hosting API?

ATL's control-hosting API is the set of functions that allows any window to act as an ActiveX control container. These functions can be statically or dynamically linked into your project since they are available as source code and exposed by ATL80.dll. The control-hosting functions are listed in the table below.

Function	Description
AtlAxAttachControl	Creates a host object, connects it to the supplied window, then attaches an existing control.
AtlAxCreateControl	Creates a host object, connects it to the supplied window, then loads a control.
AtlAxCreateControlLic	Creates a licensed ActiveX control, initializes it, and hosts it in the specified window, similar to AtlAxCreateControl.
AtlAxCreateControlEx	Creates a host object, connects it to the supplied window, then loads a control (also allows event sinks to be set up).
AtlAxCreateControlLicEx	Creates a licensed ActiveX control, initializes it, and hosts it in the specified window, similar to AtlAxCreateControlLic.
AtlAxCreateDialog	Creates a modeless dialog box from a dialog resource and returns the window handle.
AtlAxDialogBox	Creates a modal dialog box from a dialog resource.
AtlAxGetControl	Returns the **IUnknown** interface pointer of the control hosted in a window.
AtlAxGetHost	Returns the **IUnknown** interface pointer of the host object connected to a window.
AtlAxWinInit	Initializes the control-hosting code.
AtlAxWinTerm	Uninitializes the control-hosting code.

The **HWND** parameters in the first three functions must be an existing window of (almost) any type. If you call any of these three functions explicitly (typically, you won't have to), do not pass a handle to a window that's already acting as a host (if you do, the existing host object won't be freed).

The first seven functions call AtlAxWinInit implicitly.

Note
The control-hosting API forms the foundation of ATL's support for ActiveX control containment. However, there is usually littl e need to call these functions directly if you take advantage of or make full use of ATL's wrapper classes. For more informatio n, see How Do I Create a Control Instance? and Which ATL Classes Facilitate ActiveX Control Containment?.

See Also

Concepts

ATL Control Containment FAQ

What Is AtlAxWin80?

"AtlAxWin80" is the name of a window class that helps provide ATL's control-hosting functionality. When you create an instance of this class, the window procedure will automatically use the control-hosting API to create a host object associated with the window and load it with the control that you specify as the title of the window. For more information, see How Do I Load Controls Specified at Run Time?.

See Also

Concepts

ATL Control Containment FAQ

How Do I Create a Control Instance?

The ATL control-hosting code provides two main ways of creating controls. For details on each of these techniques, see How Do I Load Controls Specified at Run Time? and How Do I Load Controls Specified on a Dialog Resource?.

See Also

Concepts

ATL Control Containment FAQ

How Do I Load Controls Specified at Run Time?

To load controls dynamically, you need to create an **"AtlAxWin80"** hosting window and specify the control that it should host. There are two main ways of doing this:

1. Use the standard window creation API and the ATL hosting API. This technique is described in the Knowledge Base article "Adding ATL Control Containment Support to Any Window" (Q192560). You can find Knowledge Base articles in the MSDN Library or at http://support.microsoft.com.

2. Use the CAxWindow class as described below:

   - Make sure that AtlAxWinInit has been called.

     ```
     AtlAxWinInit();
     ```

     AtlAxWinInit initializes the control-hosting code.

   - Create a CAxWindow object:

     ```
     CAxWindow wnd;
     ```

     CAxWindow is a **CWindow**-derived wrapper for creating and manipulating **"AtlAxWin80"** windows.

   - Create a host window and control by calling **Create**.

     ```
     RECT rect = { 0, 0, 100, 100 };
     wnd.Create(m_hWnd, rect, _T("MSCAL.Calendar"), WS_CHILD |
        WS_VISIBLE | WS_CLIPSIBLINGS | WS_CLIPCHILDREN, WS_EX_CLIENTEDGE);
     ```

     The window title (the third parameter) passed to the **Create** function is a string identifying the control to create. This string can be a CLSID (with braces), a ProgID, a URL, or raw HTML (prefixed with **MSHTML:**). If either a URL or raw HTML is supplied, the Web browser will be loaded with this information.

For information on control lifetimes, see How Do I Destroy a Control?.

See Also

Concepts

ATL Control Containment FAQ

How Do I Load Controls Specified on a Dialog Resource?

To load controls specified as part of a dialog resource, you can create a class based on the Dialog, Composite Control, or Light Composite Control object types provided by the **ATL Control Wizard**, or you can call the AtlAxCreateDialog/AtlAxDialogBox APIs. Each control on the dialog box will be automatically hosted in its own **"AtlAxWin80"** window.

See Also

Concepts

Composite Control Fundamentals

ATL Control Containment FAQ

How Do I Load a Licensed Control?

"AtlAxWinLic80" is the name of a window class that helps provide ATL's control-hosting functionality for licensed controls. When you create an instance of this class, the window procedure will automatically use the control-hosting API to create a host object associated with the window and load it with the control that you specify as the title of the window. For more information, see How Do I Load Controls Specified at Run Time?.

CAxWindow2 is a **CWindow**-derived wrapper for creating and manipulating **"AtlAxWinLic80"** windows.

The procedures in the ATL Control Containment FAQ, while explicitly discussing the hosting of nonlicensed ActiveX controls, can be applied to loading licensed controls by just substituting **CAxWindow2** for **CAxWindow** and **"AtlAxWinLic80"** for **"AtlAxWin80"**, and by providing a valid license.

See Also

Concepts

ATL Control Containment FAQ

How Do I Deal with Licensed Controls?

If you need to host licensed controls, or controls that have other specialized initialization needs, you will need to create the control yourself before attaching it to a host window. You can do this using the following steps:

1. Create the control (for example, use **IClassFactory2**).
2. Initialize the control (for example, call **IPersistStreamInit::New** or **IPersistStreamInit::Load**).
3. Create a CAxWindow2T Class object.
4. Call CAxWindow::Create with an empty string as the title (this will create an **"AtlAxWin80"** window and host object without loading a control).
5. Call CAxWindow::QueryHost to get the IAxWinHostWindow interface of the host object.
6. Call IAxWinHostWindow::AttachControl to attach the control you just created to its container.

See Also

Concepts

ATL Control Containment FAQ

When Do I Need to Call AtlAxWinInit?

AtlAxWinInit registers the **"AtlAxWin80"** window class (plus a couple of custom window messages) so this function must be called before you try to create a host window. However, you don't always need to call this function explicitly, since the hosting APIs (and the classes that use them) often call this function for you. There is no harm in calling this function more than once. For more information, see What Is the ATL Control-Hosting API?.

See Also

Concepts

When Do I Need to Call AtlAxWinTerm?
ATL Control Containment FAQ

What Is a Host Object?

A host object is a COM object that represents the ActiveX control container supplied by ATL for a particular window. The host object subclasses the container window so that it can reflect messages to the control, it provides the necessary container interfaces to be used by the control, and it exposes the IAxWinHostWindow and IAxWinAmbientDispatch interfaces to allow you to configure the environment of the control.

You can use the host object to set the ambient properties of the container. For more information, see How Do I Set Ambient Properties for My Controls?.

See Also

Concepts

ATL Control Containment FAQ

Can I Host More Than One Control in a Single Window?

It is not possible to host more than one control in a single ATL host window. Each host window is designed to hold exactly one control at a time (this allows for a simple mechanism for handling message reflection and per-control ambient properties). However, if you need the user to see multiple controls in a single window, it's a simple matter to create multiple host windows as children of that window.

See Also

Concepts

ATL Control Containment FAQ

Can I Reuse a Host Window?

It is not recommended that you reuse host windows. To ensure the robustness of your code, you should tie the lifetime of your host window to the lifetime of a single control.

For information on creating controls, see How Do I Create a Control Instance?.

For information on destroying controls, see How Do I Destroy a Control?.

See Also

Concepts

ATL Control Containment FAQ

How Do I Resize a Control?

To resize a control, just resize the host window using the standard MoveWindow or SetWindowPos APIs (or their **CWindow**-wrappers). The host object resizes the control automatically to fill the host window in response to the window messages that it receives.

See Also

Concepts

ATL Control Containment FAQ

How Do I Get an Interface Pointer to a Control?

Given only the handle of a host window, you can get the **IUnknown** interface pointer of the hosted control by calling AtlAxGetControl. You can query that pointer for any interface supported by that control.

If you have the control ID of the control's host window and a **CWindow**-derived wrapper for its parent window (for example, you are using the CAxDialogImpl or CComCompositeControl classes), you can get an interface on the ActiveX control by calling CWindow::GetDlgControl.

If you have a CAxWindow object, you can get an interface on the ActiveX control by calling CAxWindow::QueryControl.

See Also

Concepts

ATL Control Containment FAQ

How Do I Set Ambient Properties for My Controls?

Given only the handle of an **"AtlAxWin80"** window, you can get the **IUnknown** interface pointer of the host object by calling AtlAxGetHost. You can query that pointer for either the IAxWinHostWindow or **IAxWinAmbientDispatch** interface. The IAxWinAmbientDispatch interface allows you to set the ambient properties available to the hosted control.

If you have a CAxWindow object, you can get an interface on the host object by calling CAxWindow::QueryHost.

See Also

Concepts

ATL Control Containment FAQ

How Do I Destroy a Control?

The best way to destroy a control is to destroy the host window. You can do this explicitly by calling either CWindow::DestroyWindow or the DestroyWindow API, or implicitly by letting the destruction of the host window's parent cause its death. Any of these methods will effectively destroy the hosted control. Note that the destruction of a CAxWindow object does *not* cause the destruction of the underlying **"AtlAxWin80"** window.

See Also

Concepts

ATL Control Containment FAQ

When Do I Need to Call AtlAxWinTerm?

AtlAxWinTerm unregisters the **"AtlAxWin80"** window class. You should call this function (if you no longer need to create host windows) after all existing host windows have been destroyed. If you don't call this function, the window class will be unregistered automatically when the process terminates.

See Also

Concepts

When Do I Need to Call AtlAxWinInit?
ATL Control Containment FAQ

Hosting ActiveX Controls Using ATL AXHost

The sample in this topic shows how to create AXHost and how to host an ActiveX control using various ATL functions. It also shows how to access the control and sink events (using IDispEventImpl) from the control that is hosted. The sample hosts the Calendar control in a main window or in a child window.

Notice the definition of the `USE_METHOD` symbol. You can change the value of this symbol to vary between 1 and 8. The value of the symbol determines how the control will be created:

- For even-numbered values of `USE_METHOD`, the call to create the host subclasses a window and converts it into a control host. For odd-numbered values, the code creates a child window that acts as a host.
- For values of `USE_METHOD` between 1 and 4, access to the control and sinking of events are accomplished in the call that also creates the host. Values between 5 and 8 query the host for interfaces and hook the sink.

Here's a summary:

USE_METHOD	Host	Control access and event sinking	Function demonstrated
1	Child window	One step	CreateControlLicEx
2	Main window	One step	AtlAxCreateControlLicEx
3	Child window	One step	CreateControlEx
4	Main window	One step	AtlAxCreateControlEx
5	Child window	Multiple steps	CreateControlLic
6	Main window	Multiple steps	AtlAxCreateControlLic
7	Child window	Multiple steps	CreateControl
8	Main window	Multiple steps	AtlAxCreateControl

```
// save this code in a .cpp file, test.cpp, then compile with
// cl test.cpp gdi32.lib
#include <atlbase.h>
#include <atlwin.h>
#include <atlhost.h>

// Value of USE_METHOD determines the function used to create the control.
#define USE_METHOD   1 // 1 to 8 are valid values

#import "PROGID:MSCAL.Calendar.7" no_namespace, raw_interfaces_only

// Child window class that will be subclassed for hosting Active X control
class CChildWindow : public CWindowImpl<CChildWindow>
{
public:
   BEGIN_MSG_MAP(CChildWindow)
   END_MSG_MAP()
};

// Helper Macro for Main Frame window class
#define DECLARE_MAINFRAME_WND_CLASS(WndClassName, style, bkgnd, menuid) \
static ATL::CWndClassInfo& GetWndClassInfo() \
{ \
   static ATL::CWndClassInfo wc = \
   { \
      { sizeof(WNDCLASSEX), style, StartWindowProc, \
        0, 0, NULL, NULL, NULL, (HBRUSH)(bkgnd + 1), menuid, WndClassName, NULL }, \
```

```
      NULL, NULL, IDC_ARROW, TRUE, 0, _T("") \
   }; \
   return wc; \
}

class CMainWindow : public CWindowImpl<CMainWindow, CWindow, CFrameWinTraits>,
   public IDispEventImpl<1, CMainWindow, &__uuidof(DCalendarEvents), &__uuidof(__MSACAL), 7
, 0>
{
public :

   CChildWindow m_wndChild;

   DECLARE_MAINFRAME_WND_CLASS("MainWindow", CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS, COLOR_WI
NDOW, 0)

   BEGIN_MSG_MAP(CMainWindow)
      MESSAGE_HANDLER(WM_CREATE, OnCreate)
      MESSAGE_HANDLER(WM_DESTROY, OnDestroy)
   END_MSG_MAP()

   BEGIN_SINK_MAP(CMainWindow)
      SINK_ENTRY_EX(1, __uuidof(DCalendarEvents), 1, OnClick)
   END_SINK_MAP()

   // Helper to display events
   void DisplayNotification(TCHAR* pszMessage)
   {
      CWindow wnd;
      wnd.Attach(GetDlgItem(2));

      wnd.SendMessage(EM_SETSEL, (WPARAM)-1, -1);
      wnd.SendMessage(EM_REPLACESEL, 0, (LPARAM)pszMessage);
   }

   // Event Handler for Click
   STDMETHOD(OnClick)()
   {
      DisplayNotification(_T("OnClick\r\n"));
      return S_OK;
   }

   LRESULT OnCreate(UINT, WPARAM, LPARAM, BOOL&)
   {
      _pAtlModule->Lock();

      RECT rect;
      GetClientRect(&rect);

      RECT rect2;
      rect2 = rect;

      rect2.bottom -=200;

      CAxWindow2 axwnd;

#if USE_METHOD | 0x2 // if USE_METHOD is a multiple of 2 then the AtlAx
                     // version is invoked.

      // Create a child window.
      // AtlAx functions will subclass this window.
      m_wndChild.Create(m_hWnd, rect2, NULL, WS_CHILD | WS_VISIBLE | WS_BORDER, 0, 1);
      // Attach the child window to the CAxWindow so we can access the
      // host that subclasses the child window.
      axwnd.Attach(m_wndChild);

#else
```

```
      // Create a Axhost directly as the child of the main window
      axwnd.Create(m_hWnd, rect2, NULL, WS_CHILD | WS_VISIBLE | WS_BORDER, 0, 1);

#endif

      if (axwnd.m_hWnd != NULL)
      {
         CComPtr<IUnknown> spControl;

         // The calls to (AtlAx)CreateControl(Lic)(Ex) do the following:
         // Create Calendar control. (Passing in NULL for license key.
         //   Pass in valid license key to the Lic functions if the
         //   control requires one.)
         // Get the IUnknown pointer for the control.
         // Sink events from the control.

         // The AtlAx versions subclass the hWnd that is passed in to them
         //   to implement the host functionality.

         // The first 4 calls accomplish it in one call.
         // The last 4 calls accomplish it using multiple steps.

#if USE_METHOD == 1
         HRESULT hr = axwnd.CreateControlLicEx(
            OLESTR("MSCAL.Calendar.7"),
            NULL,
            NULL,
            &spControl,
            __uuidof(DCalendarEvents),
            (IUnknown*)(IDispEventImpl<1, CMainWindow, &__uuidof(DCalendarEvents), &__uuido
f(__MSACAL), 7, 0>*)this
         );
#endif

#if USE_METHOD == 2
         HRESULT hr = AtlAxCreateControlLicEx(
            OLESTR("MSCAL.Calendar.7"),
            m_wndChild.m_hWnd,
            NULL,
            NULL,
            &spControl,
            __uuidof(DCalendarEvents),
            (IUnknown*)(IDispEventImpl<1, CMainWindow, &__uuidof(DCalendarEvents), &__uuido
f(__MSACAL), 7, 0>*)this,
            NULL
         );
#endif

#if USE_METHOD == 3
         HRESULT hr = axwnd.CreateControlEx(
            OLESTR("MSCAL.Calendar.7"),
            NULL,
            NULL,
            &spControl,
            __uuidof(DCalendarEvents),
            (IUnknown*)(IDispEventImpl<1, CMainWindow, &__uuidof(DCalendarEvents), &__uuido
f(__MSACAL), 7, 0>*)this
         );
#endif

#if USE_METHOD == 4
         HRESULT hr = AtlAxCreateControlEx(
            OLESTR("MSCAL.Calendar.7"),
            m_wndChild.m_hWnd,
            NULL,
            NULL,
            &spControl,
            __uuidof(DCalendarEvents),
```

```
            (IUnknown*)(IDispEventImpl<1, CMainWindow, &__uuidof(DCalendarEvents), &__uuido
f(__MSACAL), 7, 0>*)this
         );
#endif

      // The following calls create the control, obtain an interface to
      // the control, and set up the sink in multiple steps.
#if USE_METHOD == 5
         HRESULT hr = axwnd.CreateControlLic(
            OLESTR("MSCAL.Calendar.7")
         );
#endif

#if USE_METHOD == 6
         HRESULT hr = AtlAxCreateControlLic(
            OLESTR("MSCAL.Calendar.7"),
            m_wndChild.m_hWnd,
            NULL,
            NULL
         );
#endif

#if USE_METHOD == 7
         HRESULT hr = axwnd.CreateControl(
            OLESTR("MSCAL.Calendar.7")
         );
#endif

#if USE_METHOD == 8
         HRESULT hr = AtlAxCreateControl(
            OLESTR("MSCAL.Calendar.7"),
            m_wndChild.m_hWnd ,
            NULL,
            NULL
         );
#endif

#if USE_METHOD > 4 // have to obtain an interface to the control and set
                   // up the sink
         if (SUCCEEDED(hr))
         {
            hr = axwnd.QueryControl(&spControl);
            if (SUCCEEDED(hr))
            {
               // Sink events form the control
               DispEventAdvise(spControl, &__uuidof(DCalendarEvents));
            }
         }
#endif

         if (SUCCEEDED(hr))
         {
            // Use the returned IUnknown pointer.
            CComPtr<ICalendar> spCalendar;
            hr = spControl.QueryInterface(&spCalendar);
            if (SUCCEEDED(hr))
            {
               spCalendar->put_ShowDateSelectors(VARIANT_FALSE);
            }
         }
      }

      rect2 = rect;
      rect2.top = rect.bottom - 200 + 1;
      CWindow wnd;
      wnd.Create(_T("Edit"), m_hWnd, rect2, NULL, WS_CHILD | WS_VISIBLE | ES_AUTOHSCROLL |
ES_AUTOVSCROLL | ES_MULTILINE, 0, 2);
      return 0;
```

```
    }

    LRESULT OnDestroy(UINT, WPARAM, LPARAM, BOOL&)
    {
        _pAtlModule->Unlock();
        return 0;
    }
};

class CHostActiveXModule : public CAtlExeModuleT<CHostActiveXModule>
{
public :

    CMainWindow m_wndMain;

    // Create the Main window
    HRESULT PreMessageLoop(int nCmdShow)
    {
        HRESULT hr = CAtlExeModuleT<CHostActiveXModule>::PreMessageLoop(nCmdShow);
        if (SUCCEEDED(hr))
        {
            AtlAxWinInit();
            hr = S_OK;
            RECT rc;
            rc.top = rc.left = 100;
            rc.bottom = rc.right = 500;
            m_wndMain.Create(NULL, rc, _T("Host Calendar") );
            m_wndMain.ShowWindow(nCmdShow);
        }
        return hr;
    }

    // Clean up. App is exiting.
    HRESULT PostMessageLoop()
    {
        AtlAxWinTerm();
        return CAtlExeModuleT<CHostActiveXModule>::PostMessageLoop();
    }
};

CHostActiveXModule _AtlModule;

int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int nShowCmd)
{
    return _AtlModule.WinMain(nShowCmd);
}
```

See Also

Reference

AtlAxCreateControl
AtlAxCreateControlEx
AtlAxCreateControlLic
AtlAxCreateControlLicEx
CAxWindow2T Class
IAxWinHostWindowLic Interface

Concepts

ATL Control Containment FAQ

ATL COM Property Pages

COM property pages provide a user interface for setting the properties (or calling the methods) of one or more COM objects. Property pages are used extensively by ActiveX controls for providing rich user interfaces that allow control properties to be set at design time.

Property pages are COM objects that implement the IPropertyPage or IPropertyPage2 interface. These interfaces provide methods that allow the page to be associated with a *site* (a COM object representing the container of the page) and a number of *objects* (COM objects whose methods will be called in response to changes made by the user of the property page). The property page container is responsible for calling methods on the property page interface to tell the page when to show or hide its user interface, and when to apply the changes made by the user to the underlying objects.

Each property page can be built completely independently of the objects whose properties can be set. All that a property page needs is to understand a particular interface (or set of interfaces) and to provide a user interface for calling methods on that interface.

For more information, see Property Sheets and Property Pages in the *Platform SDK*.

In This Section

Specifying Property Pages

Lists the steps for specifying property pages for you control and shows an example class.

Implementing Property Pages

Lists the steps for implementing property pages, including methods to override. Walks you through a complete example based on the ATLPages sample program.

Related Sections

ATLPages Sample

The sample abstract for the ATLPages sample, which implements a property page using **IPropertyPageImpl**.

ATL

Provides links to conceptual topics on how to program using the Active Template Library.

See Also

Other Resources

ATL Concepts

Specifying Property Pages

When you create an ActiveX control, you will often want to associate it with property pages that can be used to set the properties of your control. Control containers use the **ISpecifyPropertyPages** interface to find out which property pages can be used to set your control's properties. You will need to implement this interface on your control.

To implement **ISpecifyPropertyPages** using ATL, take the following steps:

1. Derive your class from ISpecifyPropertyPagesImpl.
2. Add an entry for **ISpecifyPropertyPages** to your class's COM map.
3. Add a PROP_PAGE entry to the property map for each page associated with your control.

Note

When generating a standard control using the ATL Control Wizard, you will only have to add the **PROP_PAGE** entries to the property map. The wizard generates the necessary code for the other steps.

Well-behaved containers will display the specified property pages in the same order as the **PROP_PAGE** entries in the property map. Generally, you should put standard property page entries after the entries for your custom pages in the property map, so that users see the pages specific to your control first.

Example

The following class for a calendar control uses the **ISpecifyPropertyPages** interface to tell containers that its properties can be set using a custom date page and the stock color page.

```
class ATL_NO_VTABLE CCalendar :
    // ...
    public ISpecifyPropertyPagesImpl<CCalendar>,
    // ...
{
BEGIN_COM_MAP(CCalendar)
    // ...
    COM_INTERFACE_ENTRY(ISpecifyPropertyPages)
    // ...
END_COM_MAP()
// ...
BEGIN_PROP_MAP(CCalendar)
    // ...
    PROP_PAGE(CLSID_DatePage)
    PROP_PAGE(CLSID_StockColorPage)
    // ...
END_PROP_MAP()

};
```

See Also

Tasks

ATLPages Sample: Implements a Property Page Using IPropertyPageImpl

Concepts

ATL COM Property Pages

Implementing Property Pages

Property pages are COM objects that implement the **IPropertyPage** or **IPropertyPage2** interface. ATL provides support for implementing property pages through the ATL Property Page Wizard in the Add Class dialog box.

To create a property page using ATL:

- Create or open an ATL Dynamic-link library (DLL) server project.
- Open the Add Class dialog box and select **ATL Property Page**.
- Make sure your property page is apartment threaded (since it has a user interface).
- Set the title, description (Doc String), and help file to be associated with your page.
- Add controls to the generated dialog resource to act as the user interface of your property page.
- Respond to changes in your page's user interface to perform validation, update the page site, or update the objects associated with your page. In particular, call IPropertyPageImpl::SetDirty when the user makes changes to the property page.
- Optionally override the **IPropertyPageImpl** methods using the guidelines below.

IPropertyPageImpl method	Override when you want to...	Notes
SetObjects	Perform basic sanity checks on the number of objects being passed to your page and the interfaces that they support.	Execute your own code before calling the base class implementation. If the objects being set don't conform to your expectations, you should fail the call as soon as possible.
Activate	Initialize your page's user interface (for example, set dialog controls with current property values from objects, create controls dynamically, or perform other initializations).	Call the base class implementation before your code so that the base class has a chance to create the dialog window and all the controls before you try to update them.
Apply	Validate the property settings and update the objects.	There is no need to call the base class implementation since it doesn't do anything apart from trace the call.
Deactivate	Clean up window-related items.	The base class implementation destroys the dialog box representing the property page. If you need to clean up before the dialog box is destroyed, you should add your code before calling the base class.

For an example property page implementation, see Example: Implementing a Property Page.

Note
If you want to host ActiveX controls in your property page, you will need to change the derivation of your wizard-generated class. Replace **CDialogImpl<CYourClass>** with **CAxDialogImpl<CYourClass>** in the list of base classes.

See Also

Tasks

ATLPages Sample: Implements a Property Page Using IPropertyPageImpl

Concepts

ATL COM Property Pages

Example: Implementing a Property Page

This example shows how to build a property page that displays (and allows you to change) properties of the Document Classes interface. This interface is exposed by documents in Visual Studio's Common Environment Object Model Examples (although the property page that you'll create won't care where the objects it manipulates come from as long as they support the correct interface).

The example is based on the ATLPages sample.

To complete this example, you will:

- Add the ATL property page class using the Add Class dialog box and the ATL Property Page Wizard.
- Edit the dialog resource by adding new controls for the interesting properties of the **Document** interface.
- Add message handlers to keep the property page site informed of changes made by the user.
- Add some **#import** statements and a typedef in the Housekeeping section.
- Override IPropertyPageImpl::SetObjects to validate the objects being passed to the property page.
- Override IPropertyPageImpl::Activate to initialize the property page's interface.
- Override IPropertyPageImpl::Apply to update the object with the latest property values.
- Display the property page by creating a simple helper object.
- Create a macro that will test the property page.

Adding the ATL Property Page Class

First, create a new ATL project for a DLL server called `ATLPages7`. Now use the ATL Property Page Wizard to generate a property page. Give the property page a **Short Name** of **DocumentProperties** then switch to the **Strings** page to set property-page-specific items as shown in the table below.

Item	Value
Title	TextDocument
Doc String	VCUE TextDocument Properties
Helpfile	*<blank>*

The values that you set on this page of the wizard will be returned to the property page container when it calls **IPropertyPage::GetPageInfo**. What happens to the strings after that is dependent on the container, but typically they will be used to identify your page to the user. The Title will usually appear in a tab above your page and the Doc String may be displayed in a status bar or ToolTip (although the standard property frame doesn't use this string at all).

Note
The strings that you set here are stored as string resources in your project by the wizard. You can easily edit these strings using the resource editor if you need to change this information after the code for your page has been generated.

Click **OK** to have the wizard generate your property page.

Editing the Dialog Resource

Now that your property page has been generated, you'll need to add a few controls to the dialog resource representing your page. Add an edit box, a static text control, and a check box and set their IDs as shown below:

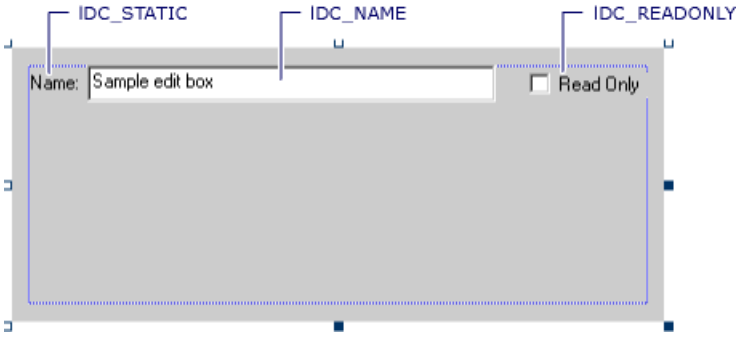

These controls will be used to display the file name of the document and its read-only status.

Note
The dialog resource does not include a frame or command buttons, nor does it have the tabbed look that you might have expected. These features are provided by a property page frame such as the one created by calling OleCreatePropertyFrame.

Adding Message Handlers

With the controls in place, you can add message handlers to update the dirty status of the page when the value of either of the controls changes:

```
BEGIN_MSG_MAP(CDocumentProperties)
   COMMAND_HANDLER(IDC_NAME, EN_CHANGE, OnUIChange)
   COMMAND_HANDLER(IDC_READONLY, BN_CLICKED, OnUIChange)
   CHAIN_MSG_MAP(IPropertyPageImpl<CDocumentProperties>)
END_MSG_MAP()

// Respond to changes in the UI to update the dirty status of the page
LRESULT OnUIChange(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled)
{
   wNotifyCode; wID; hWndCtl; bHandled;
   SetDirty(true);
   return 0;
}
```

This code responds to changes made to the edit control or check box by calling IPropertyPageImpl::SetDirty, which informs the page site that the page has changed. Typically the page site will respond by enabling or disabling an **Apply** button on the property page frame.

Note
In your own property pages, you might need to keep track of precisely which properties have been altered by the user so that you can avoid updating properties that haven't been changed. This example implements that code by keeping track of the original property values and comparing them with the current values from the UI when it's time to apply the changes.

Housekeeping

Now add a couple of **#import** statements to DocumentProperties.h so that the compiler knows about the **Document** interface:

```
// MSO.dll
#import <libid:2DF8D04C-5BFA-101B-BDE5-00AA0044DE52> version("2.2") \
   rename("RGB", "Rgb")   \
   rename("DocumentProperties", "documentProperties")   \
   rename("ReplaceText", "replaceText")   \
   rename("FindText", "findText")   \
   rename("GetObject", "getObject")   \
   raw_interfaces_only

// dte.olb
#import <libid:80CC9F66-E7D8-4DDD-85B6-D9E6CD0E93E2> \
   inject_statement("using namespace Office;")   \
```

```
    rename("ReplaceText", "replaceText")   \
    rename("FindText", "findText")   \
    rename("GetObject", "getObject")   \
    rename("SearchPath", "searchPath")   \
    raw_interfaces_only
```

You'll also need to refer to the **IPropertyPageImpl** base class; add the following **typedef** to the **CDocumentProperties** class:

```
typedef IPropertyPageImpl<CDocumentProperties> PPGBaseClass;
```

Overriding IPropertyPageImpl::SetObjects

The first **IPropertyPageImpl** method that you need to override is SetObjects. Here you'll add code to check that only a single object has been passed and that it supports the **Document** interface that you're expecting:

```
STDMETHOD(SetObjects)(ULONG nObjects, IUnknown** ppUnk)
{
   HRESULT hr = E_INVALIDARG;
   if (nObjects == 1)
   {
      CComQIPtr<EnvDTE::Document> pDoc(ppUnk[0]);
      if (pDoc)
         hr = PPGBaseClass::SetObjects(nObjects, ppUnk);
   }
   return hr;
}
```

Note

It makes sense to support only a single object for this page because you will allow the user to set the file name of the object — only one file can exist at any one location.

Overriding IPropertyPageImpl::Activate

The next step is to initialize the property page with the property values of the underlying object when the page is first created.

In this case you should add the following members to the class since you'll also use the initial property values for comparison when users of the page apply their changes:

```
CComBSTR m_bstrFullName;  // The original name
VARIANT_BOOL m_bReadOnly; // The original read-only state
```

The base class implementation of the Activate method is responsible for creating the dialog box and its controls, so you can override this method and add your own initialization after calling the base class:

```
STDMETHOD(Activate)(HWND hWndParent, LPCRECT prc, BOOL bModal)
{
   // If we don't have any objects, this method should not be called
   // Note that OleCreatePropertyFrame will call Activate even if
   // a call to SetObjects fails, so this check is required
   if (!m_ppUnk)
      return E_UNEXPECTED;

   // Use Activate to update the property page's UI with information
   // obtained from the objects in the m_ppUnk array

   // We update the page to display the Name and ReadOnly properties
   // of the document

   // Call the base class
   HRESULT hr = PPGBaseClass::Activate(hWndParent, prc, bModal);
   if (FAILED(hr))
      return hr;
```

```
    // Get the EnvDTE::Document pointer
    CComQIPtr<EnvDTE::Document> pDoc(m_ppUnk[0]);
    if (!pDoc)
       return E_UNEXPECTED;

    // Get the FullName property
    hr = pDoc->get_FullName(&m_bstrFullName);
    if (FAILED(hr))
       return hr;

    // Set the text box so that the user can see the document name
    USES_CONVERSION;
    SetDlgItemText(IDC_NAME, CW2CT(m_bstrFullName));

    // Get the ReadOnly property
    m_bReadOnly = VARIANT_FALSE;
    hr = pDoc->get_ReadOnly(&m_bReadOnly);
    if (FAILED(hr))
       return hr;

    // Set the check box so that the user can see the document's read-only status
    CheckDlgButton(IDC_READONLY, m_bReadOnly ? BST_CHECKED : BST_UNCHECKED);

    return hr;
 }
```

This code uses the COM methods of the **Document** interface to get the properties that you're interested in. It then uses the Win32 API wrappers provided by CDialogImpl and its base classes to display the property values to the user.

Overriding IPropertyPageImpl::Apply

When users want to apply their changes to the objects, the property page site will call the Apply method. This is the place to do the reverse of the code in **Activate** — whereas **Activate** took values from the object and pushed them into the controls on the property page, **Apply** takes values from the controls on the property page and pushes them into the object.

```
 STDMETHOD(Apply)(void)
 {
    // If we don't have any objects, this method should not be called
    if (!m_ppUnk)
       return E_UNEXPECTED;

    // Use Apply to validate the user's settings and update the objects'
    // properties

    // Check whether we need to update the object
    // Quite important since standard property frame calls Apply
    // when it doesn't need to
    if (!m_bDirty)
       return S_OK;

    HRESULT hr = E_UNEXPECTED;

    // Get a pointer to the document
    CComQIPtr<EnvDTE::Document> pDoc(m_ppUnk[0]);
    if (!pDoc)
       return hr;

    // Get the read-only setting
    VARIANT_BOOL bReadOnly = IsDlgButtonChecked(IDC_READONLY) ? VARIANT_TRUE : VARIANT_FALSE
 ;

    // Get the file name
    CComBSTR bstrName;
    if (!GetDlgItemText(IDC_NAME, bstrName.m_str))
       return E_FAIL;
```

```
    // Set the read-only property
    if (bReadOnly != m_bReadOnly)
    {
       hr = pDoc->put_ReadOnly(bReadOnly);
       if (FAILED(hr))
       return hr;
    }

    // Save the document
    if (bstrName != m_bstrFullName)
    {
       EnvDTE::vsSaveStatus status;
       hr = pDoc->Save(bstrName, &status);
       if (FAILED(hr))
       return hr;
    }

    // Clear the dirty status of the property page
    SetDirty(false);

    return S_OK;
 }
```

Note

The check against m_bDirty at the beginning of this implementation is an initial check to avoid unnecessary updates of the objects if **Apply** is called more than once. There are also checks against each of the property values to ensure that only changes result in a method call to the **Document**.

Note

Document exposes **FullName** as a read-only property. To update the file name of the document based on changes made to the property page, you have to use the **Save** method to save the file with a different name. Thus, the code in a property page doesn't have to limit itself to getting or setting properties.

Displaying the Property Page

To display this page, you need to create a simple helper object. The helper object will provide a method that simplifies the **OleCreatePropertyFrame** API for displaying a single page connected to a single object. This helper will be designed so that it can be used from Visual Basic.

Use the Add Class dialog box and the ATL Simple Object Wizard to generate a new class and use `Helper` as its short name. Once created, add a method as shown in the table below.

Item	Value
Method Name	`ShowPage`
Parameters	`[in] BSTR bstrCaption, [in] BSTR bstrID, [in] IUnknown* pUnk`

The `bstrCaption` parameter is the caption to be displayed as the title of the dialog box. The `bstrID` parameter is a string representing either a CLSID or a ProgID of the property page to display. The `pUnk` parameter will be the `IUnknown` pointer of the object whose properties will be configured by the property page.

Implement the method as shown below:

```
STDMETHODIMP CHelper::ShowPage(BSTR bstrCaption, BSTR bstrID,
                               IUnknown* pUnk)
{
    if (!pUnk)
        return E_INVALIDARG;

    // First, assume bstrID is a string representing the CLSID
    CLSID theCLSID = {0};
```

```
    HRESULT hr = CLSIDFromString(bstrID, &theCLSID);
    if (FAILED(hr))
    {
        // Now assume bstrID is a ProgID
        hr = CLSIDFromProgID(bstrID, &theCLSID);
        if (FAILED(hr))
            return hr;
    }

    // Use the system-supplied property frame
    return OleCreatePropertyFrame(
        GetActiveWindow(),   // Parent window of the property frame
                0,           // Horizontal position of the property frame
                0,           // Vertical position of the property frame
                bstrCaption, // Property frame caption
                1,           // Number of objects
                &pUnk,       // Array of IUnknown pointers for objects
                1,           // Number of property pages
                &theCLSID,   // Array of CLSIDs for property pages
                NULL,        // Locale identifier
                0,           // Reserved - 0
                NULL         // Reserved - 0
                );
}
```

Creating a Macro

Once you've built the project, you can test the property page and the helper object using a simple macro that you can create and run in the Visual Studio development environment. This macro will create a helper object, then call its **ShowPage** method using the ProgID of the **DocumentProperties** property page and the **IUnknown** pointer of the document currently active in the Visual Studio editor. The code you need for this macro is shown below:

```
Imports EnvDTE
Imports System.Diagnostics

Public Module AtlPages

    Public Sub Test()
        Dim Helper
        Helper = CreateObject("ATLPages7.Helper.1")

        On Error Resume Next
        Helper.ShowPage( _
            ActiveDocument.Name, _
            "ATLPages7.DocumentProperties.1", _
            DTE.ActiveDocument _
            )
    End Sub

End Module
```

When you run this macro, the property page will be displayed showing the file name and read-only status of the currently active text document. The read-only state of the document only reflects the ability to write to the document in the development environment; it doesn't affect the read-only attribute of the file on disk.

See Also

Tasks

ATLPages Sample: Implements a Property Page Using IPropertyPageImpl

Concepts

ATL COM Property Pages

ATL Support for DHTML Controls

Using ATL, you can create a control with Dynamic HTML (DHTML) capability. An ATL DHTML control:

- Hosts the WebBrowser control.
- Specifies, using HTML, the user interface (UI) of the DHTML control.
- Accesses the WebBrowser object and its methods through its interface, IWebBrowser2.
- Manages communication between C++ code and HTML.

A DHTML control is similar to any other ATL control, except the DHTML control includes an additional dispatch interface. See the figure in Identifying the Elements of the DHTML Control Project for an illustration of the interfaces provided in the default DHTML project.

You can view the ATL DHTML control in a Web browser or other container, such as the ActiveX Control Test Container.

In This Section

Identifying the Elements of the DHTML Control Project

Describes the elements of a DHTML control project.

Calling C++ Code from DHTML

Provides an example of calling C++ code from a DHTML control.

Creating an ATL DHTML Control

Lists the steps for creating a DHTML control.

Testing the ATL DHTML Control

Shows how to build and test the initial DHTML control project.

Modifying the ATL DHTML Control

Shows how to add some functionality to the control.

Testing the Altered ATL DHTML Control

Shows how to build and test the control's added functionality.

Related Sections

ATL

Provides links to conceptual topics on how to program using the Active Template Library.

Identifying the Elements of the DHTML Control Project

Most DHTML control code is exactly like that created for any ATL control. For a basic understanding of the generic code, work through the ATL tutorial, and read the sections Creating an ATL Project and Fundamentals of ATL COM Objects.

A DHTML control is similar to any ATL control, except:

- In addition to the regular interfaces a control implements, it implements an additional interface that is used to communicate between the C++ code and the HTML user interface (UI). The HTML UI calls into C++ code using this interface.
- It creates an HTML resource for the control UI.
- It allows access to the DHTML object model through the member variable `m_spBrowser`, which is a smart pointer of type IWebBrowser2. Use this pointer to access any part of the DHTML object model.

The following graphic illustrates the relationship between your DLL, the DHTML control, the Web browser, and the HTML resource.

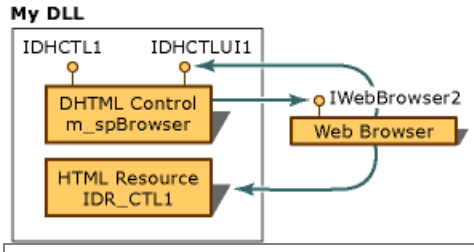

Note
The names on this graphic are placeholders. The names of your HTML resource and the interfaces exposed on your control are based on the names you assign them in the ATL Control Wizard.

In this graphic, the elements are:

- **My DLL** The DLL created using the ATL Project Wizard.
- **DHTML Control** (`m_spBrowser`) The DHTML control, created using the ATL Object Wizard. This control accesses the Web browser object and its methods through the Web browser object's interface, **IWebBrowser2**. The control itself exposes the following two interfaces, in addition to the other standard interfaces required for a control.
  - **IDHCTL1** The interface exposed by the control for use only by the container.
  - **IDHCTLUI1** The dispatch interface for communicating between the C++ code and the HTML user interface. The Web browser uses the control's dispatch interface to display the control. You can call various methods of this dispatch interface from the control's user interface by invoking `window.external`, followed by the method name on this dispatch interface that you want to invoke. You would access `window.external` from a SCRIPT tag within the HTML that makes up the UI for this control. For more information about invoking external methods in the resource file, see Calling C++ Code from DHTML.
- **IDR_CTL1** The resource ID of the HTML resource. Its file name, in this case, is DHCTL1UI.htm. The DHTML control uses an HTML resource that contains standard HTML tags and external window dispatch commands that you can edit using the Text editor.
- **Web Browser** The Web browser displays the control's UI, based on the HTML in the HTML resource. A pointer to the Web browser's **IWebBrowser2** interface is available in the DHTML control to allow access to the DHTML object model.

The ATL Control Wizard generates a control with default code in both the HTML resource and the .cpp file. You can compile and run the control as generated by the wizard, and then view the control in either the Web browser or the ActiveX Control Test Container. The picture below shows the default ATL DHTML control with three buttons displayed in Test Container:

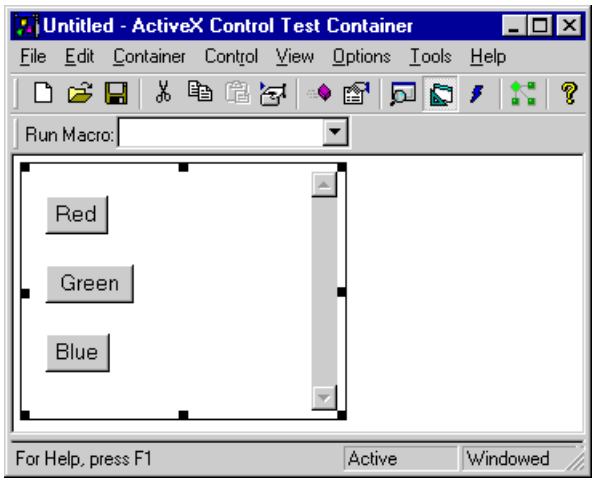

See Creating an ATL DHTML Control to get started building a DHTML control. See Testing Properties and Events with Test Container for information on how to access Test Container.

See Also

Other Resources

ATL Support for DHTML Controls

Calling C++ Code from DHTML

A DHTML control can be hosted in a container, such as Test Container or Internet Explorer. See Testing Properties and Events with Test Container for information on how to access Test Container.

The container hosting the control communicates with the control using the normal control interfaces. DHTML uses the dispatch interface that ends with "UI" to communicate with your C++ code and your HTML resource. In Modifying the ATL DHTML Control, you can practice adding the methods to be called by these different interfaces.

To see an example of calling C++ code from DHTML, create a DHTML control using the ATL Control Wizard and examine the code in the header file and in the HTML file.

Declaring WebBrowser Methods in the Header File

To invoke C++ methods from the DHTML UI, you must add methods to your control's UI interface. For example, the header file created by the ATL Control Wizard contains the C++ method `OnClick`, which is a member of the UI interface of the wizard-generated control.

Examine `OnClick` in the control's .h file:

```
STDMETHOD(OnClick)(IDispatch* pdispBody, VARIANT varColor)
```

The first parameter, `pdispBody`, is a pointer to the body object's dispatch interface. The second parameter, `varColor`, identifies the color to apply to the control.

Calling C++ Code in the HTML File

Once you have declared the WebBrowser methods in the header file, you can invoke the methods from the HTML file. Notice in the HTML file that the ATL Control Wizard inserts three Windows dispatch methods: three `OnClick` methods that dispatch messages to change the background color of the control.

Examine one of the methods in the HTML file:

```
<BUTTON onclick='window.external.OnClick(theBody, "red");'>Red</BUTTON>
```

In the HTML code above, the window external method, `OnClick`, is called as part of the button tag. The method has two parameters: `theBody`, which references the body of the HTML document, and `"red"`, which indicates that the control's background color will be changed to red when the button is clicked. The `Red` following the tag is the button's label.

See Modifying the ATL DHTML Control for more information about providing your own methods. See Identifying the Elements of the DHTML Control Project for more information about the HTML file.

See Also

Other Resources

ATL Support for DHTML Controls

Creating an ATL DHTML Control

The ATL Control Wizard automates the process of creating a DHTML control. It generates the necessary resource files, including an HTML file containing sample code.

To create an ATL DHTML control

1. Follow the steps in Creating an ATL Project.
2. In Class View, right-click the project node, point to **Add**, and click **Add Class** from the shortcut menu. In the **Add Class** dialog box, double-click the ATL Control Wizard. In the ATL Control Wizard, click the **Options** tab and select **DHTML control**.

You can now test the default control.

See Also

Other Resources

ATL Support for DHTML Controls

Testing the ATL DHTML Control

Once you have created your project, you can build and test the sample control. Before you do this, use Class View and Solution Explorer to examine the project. The elements of your project are described in greater detail in Identifying the Elements of the DHTML Control Project.

To build and test the ATL DHTML control

1. Build the project. From the **Build** menu, click **Build Solution**.
2. When the build is completed, open Test Container. See Testing Properties and Events with Test Container for information on how to access Test Container.
3. In Test Container, from the **Edit** menu, click **Insert New Control**.
4. In the **Insert Control** dialog box, select your control from the list box. Remember, its name is based on the short name you indicated in the ATL Control Wizard. Click **OK**.
5. Examine the control. Note that it has a scroll bar. Use the control's handles to resize the control to activate the scrollbar.
6. Test the control's buttons. The background color changes to the color indicated by the button.
7. Close Test Container.

Next, try modifying the DHTML control.

See Also

Other Resources

ATL Support for DHTML Controls

Modifying the ATL DHTML Control

The ATL Control Wizard provides starter code so you can build and run the control, and so you can see how the methods are written in the project files and how the DHTML calls into the control's C++ code using the dispatch methods. You can add any dispatch method to the interface. Then, you can call the methods in the HTML resource.

To modify the ATL DHTML control

1. In Class View, expand the control project.

   Note that the interface that ends in "UI" has one method, `OnClick`. The interface that does not end in "UI" does not have any methods.

2. Add a method called `MethodInvoked` to the interface that does not end in "UI."

   This method will be added to the interface that is used in the control container for container interaction, not to the interface used by DHTML to interact with the control. Only the container can invoke this method.

3. Find the stubbed-out method in the .cpp file and add code to display a message box, for example:

   ```
   ::MessageBox(NULL, "I'm invoked", "Your Container Message", MB_OK);
   ```

4. Add another method called `HelloHTML`, only this time, add it to the interface that ends in "UI." Find the stubbed-out `HelloHTML` method in the .cpp file and add code to display a message box, for example:

   ```
   ::MessageBox(NULL, "Here's your message", "HelloHTML", MB_OK);
   ```

5. Add a third method, `GoToURL`, also to the interface ending in "UI." Implement this method by calling IWebBrowser2::Navigate, as follows:

   ```
   m_spBrowser->Navigate(CComBSTR("www.microsoft.com"), NULL, NULL, NULL, NULL);
   ```

   You can use the **IWebBrowser2** methods because ATL provides a pointer to that interface for you in your .h file.

Next, modify the HTML resource to invoke the methods you created. You will add three buttons for invoking these methods.

To modify the HTML resource

1. In Solution Explorer, double-click the .htm file to display the HTML resource.

   Examine the HTML, especially the calls to the external Windows dispatch methods. The HTML calls the project's `OnClick` method, and the parameters indicate the body of the control (`theBody`) and the color to assign ("`red`"). The text following the method call is the label that appears on the button.

2. Add another `OnClick` method, only change the color. For example:

   ```
   <br>
   <br>
   <BUTTON onclick='window.external.OnClick(theBody, "white");'>Refresh</BUTTON>
   ```

   This method will create a button, labeled **Refresh**, that the user can click to return the control to the original, white background.

3. Add the call to the `HelloHTML` method you created. For example:

   ```
   <br>
   <br>
   <BUTTON onclick='window.external.HelloHTML();'>HelloHTML</BUTTON>
   ```

This method will create a button, labeled **HelloHTML**, that the user can click to display the `HelloHTML` message box.

4. Finally, add the call to the `GoToURL` method you created. For example:

```
<br>
<br>
<BUTTON onclick='window.external.GoToURL();'>WWW</BUTTON>
```

This method will create a button, labeled **WWW**, that the user can click to display the Microsoft Web page in the control.

You can now build and test the modified DHTML control.

See Also

Other Resources

ATL Support for DHTML Controls

Testing the Modified ATL DHTML Control

Try out your new control to see how it works now.

To build and test the modified control

1. Rebuild the project and open it in Test Container. See Testing Properties and Events with Test Container for information on how to access Test Container.

   Resize the control to show all of the buttons you added. Before you try them, test the method that is not part of the UI.

2. Highlight the control, so the border is activated.

3. On the **Control** menu, click **Invoke Methods**.

   The only method on the list labeled **Method Name** is the method that the container can call: `MethodInvoked`. All other methods are controlled by the UI.

4. Click **Invoke** to display the method's message box.

5. Click **OK** and then, in the **Invoke Methods** dialog box, click **Close**.

6. Examine the three buttons that you inserted by altering the HTML. Each button bears the label you identified in Modifying the ATL DHTML Control: **Refresh**, **HelloHTML**, and **WWW**.

7. Test the three new buttons to see how they work.

To learn more about the various elements and files that make up an ATL DHTML control, see Identifying the Elements of the DHTML Control Project.

See Also

Other Resources

ATL Support for DHTML Controls

ATL Connection Points

A connectable object is one that supports outgoing interfaces. An outgoing interface allows the object to communicate with a client. For each outgoing interface, the connectable object exposes a connection point. Each outgoing interface is implemented by a client on an object called a sink.

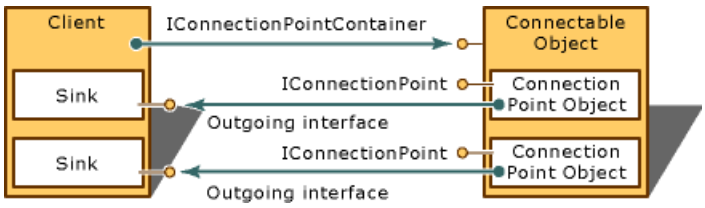

Each connection point supports the IConnectionPoint interface. The connectable object exposes its connection points to the client through the IConnectionPointContainer interface.

In This Section

ATL Connection Point Classes

Briefly describes the ATL classes that support connection points.

Adding Connection Points to an Object

Outlines the steps used to add connection points to an object.

ATL Connection Point Example

Provides an example of declaring a connection point.

ATL Connection Point Samples

Provides links to ATL samples that demonstrate connection points being used in real projects.

Related Sections

ATL

Provides links to conceptual topics on how to program using the Active Template Library.

See Also

Other Resources

ATL Concepts

ATL Connection Point Classes

ATL uses the following classes to support connection points:

- IConnectionPointImpl implements a connection point. The IID of the outgoing interface it represents is passed as a template parameter.
- IConnectionPointContainerImpl implements the connection point container and manages the list of **IConnectionPointImpl** objects.
- IPropertyNotifySinkCP implements a connection point representing the IPropertyNotifySink interface.
- CComDynamicUnkArray manages an arbitrary number of connections between the connection point and its sinks.
- CComUnkArray manages a predefined number of connections as specified by the template parameter.
- CFirePropNotifyEvent notifies a client's sink that an object's property has changed or is about to change.
- IDispEventImpl provides support for connection points for an ATL COM object. These connection points are mapped with an event sink map, which is provided by your COM object.
- IDispEventSimpleImpl works in conjunction with the event sink map in your class to route events to the appropriate handler function.

See Also
Reference
ATL Connection Points

Adding Connection Points to an Object

The ATL Tutorial demonstrates how to create a control with support for connection points, how to add events, and then how to implement the connection point. ATL implements connection points with the IConnectionPointImpl class.

To implement a connection point, you have two choices:

- Implement your own outgoing event source, by adding a connection point to the control or object.
- Reuse a connection point interface defined in another type library.

In either case, the Implement Connection Point Wizard uses a type library to do its work.

To add a connection point to a control or object

1. Define a dispinterface in the library block of the .idl file. If you enabled support for connection points when you created the control with the ATL Control Wizard, the dispinterface will already be created. If you did not enable support for connection points when you created the control, you must manually add a dispinterface to the .idl file. The following is an example of a dispinterface. Outgoing interfaces are not required to be dispatch interfaces but many scripting languages such as VBScript and JScript require this, so this example uses two dispinterfaces:

   ```
   library DProjLib
   {
      importlib("stdole32.tlb");
      importlib("stdole2.tlb");
      [
         uuid(57BC50F0-780B-11d1-8C44-0060083E866C),
         helpstring("Buddy Events")
      ]
   dispinterface DBuddyEvents
      properties:
      methods:
   };
   ```

   Use either the uuidgen.exe or guidgen.exe utility to generate a GUID.

2. Add the dispinterface as the `[default,source]` interface in the coclass for the object in the project's .idl file. Again, if you enabled support for connection points when you created the control, the ATL Control Wizard will create the `[default,source]` entry. To manually add this entry, add the line in bold:

   ```
   coclass Buddy
   {
      [default] interface IBuddy;
      [default,source] dispinterface DBuddyEvents;
   };
   ```

   See the .idl file in the Circ ATL sample for an example.

3. Use Class View to add methods and properties to the event interface. Right-click the class in Class View, point to **Add** on the shortcut menu, and click **Add Connection Point**.

4. In the **Source Interfaces** list box of the Implement Connection Point Wizard, select **Project's interfaces**. If you choose an interface for your control and press **OK**, you will:

   - Generate a header file with an event proxy class that implements the code that will make the outgoing calls for the event.
   - Add an entry to the connection point map.

   You will also see a list of all of the type libraries on your computer. You should only use one of these other type libraries

to define your connection point if you want to implement the exact same outgoing interface found in another type library.

To reuse a connection point interface defined in another type library

1. In Class View, right-click a class that implements a **BEGIN_COM_MAP** macro, point to **Add** on the shortcut menu, and click **Add Connection Point**.
2. In the Implement Connection Point Wizard, select a type library and an interface in the type library and click **Add**.
3. Edit the .idl file to either:
   - Copy the dispinterface from the .idl file for the object whose event-source is being used.
   - Use the **importlib** instruction on that type library, as shown below. This example is from an .idl file that is using the connection point from the Circ control.

```
[
   uuid(38BBFD91-7575-11D1-8C3C-0060083E866C),
   version(1.0),
   helpstring("16190 1.0 Type Library")
]
library MY16190Lib
{
   importlib("stdole32.tlb");
   importlib("stdole2.tlb");
   importlib("D:\\ATL25\\samples\\circ\\circ.tlb");   //**** add this line

   [
      uuid(38BBFD9E-7575-11D1-8C3C-0060083E866C),
      helpstring("Atl16190Ctl Class")
   ]
   coclass Atl16190Ctl
   {
      [default] interface IAtl16190Ctl;
      [default,source] dispinterface _CircEvents;      //**** and this one
   };
};
```

See Also

Reference

ATL Connection Points

ATL Connection Point Example

This example shows an object that supports IPropertyNotifySink as an outgoing interface:

```
class CConnect :
   public CComObjectRootEx<CComObjectThreadModel>,
   public CComCoClass<CConnect, &CLSID_CConnect>,
   public IConnectionPointContainerImpl<CConnect>,
   public IConnectionPointImpl<CConnect,
             &IID_IPropertyNotifySink>
{
public:
   ...
   BEGIN_COM_MAP(CConnect)
      COM_INTERFACE_ENTRY_IMPL(IConnectionPointContainer)
   END_COM_MAP()

   BEGIN_CONNECTION_POINT_MAP(CConnect)
      CONNECTION_POINT_ENTRY(IID_IPropertyNotifySink)
   END_CONNECTION_POINT_MAP()
   ...
};
```

When specifying **IPropertyNotifySink** as an outgoing interface, you can use class IPropertyNotifySinkCP instead of **IConnectionPointImpl**. For example:

```
class CConnect :
   public CComObjectRootEx<CComObjectThreadModel>,
   public CComCoClass<CConnect, &CLSID_CConnect>,
   public IConnectionPointContainerImpl<CConnect>,
   public IPropertyNotifySinkCP<CConnect>
{
   ...
};
```

See Also

Reference

ATL Connection Points

ATL Connection Point Samples

The following sample demonstrate the use of connection points:

- ATLDuck Sample: Uses Connection Points with ATL

See Also

Reference

ATL Connection Points

Event Handling and ATL

This section shows how to sink events using ATL. It covers the principles of COM event handling and the specifics of sinking events using the support provided by ATL.

For information on how to fire events and implement connection points, read ATL Connection Points.

In This Section

Event Handling Principles

Discusses the steps common to all event handing.

Implementing the Event Handling Interface

Discusses the classes to use for implementing the event interface.

Using IDispEventImpl

Lists the steps for using **IDispEventImpl** and shows a code sample.

Using IDispEventSimpleImpl

Lists the steps for using **IDispEventSimpleImpl** and shows a code sample.

ATL Event Handling Summary

Summarizes, using tables, the main ways for implementing an event interface and for advising and unadvising the event source using ATL.

Related Sections

ATL

Provides links to conceptual topics on how to program using the Active Template Library.

See Also

Other Resources

ATL Concepts

Event Handling Principles

There are three steps common to all event handling. You will need to:

- Implement the event interface on your object.
- Advise the event source that your object wants to receive events.
- Unadvise the event source when your object no longer needs to receive events.

The way that you'll implement the event interface will depend on its type. An event interface can be vtable, dual, or a dispinterface. It's up to the designer of the event source to define the interface; it's up to you to implement that interface.

Note
Although there are no technical reasons that an event interface can't be dual, there are a number of good design reasons to a void the use of duals. However, this is a decision made by the designer/implementer of the event *source*. Since you're workin g from the perspective of the event *sink*, you need to allow for the possibility that you might not have any choice but to imple ment a dual event interface. For more information on dual interfaces, see Dual Interfaces and ATL.

Advising the event source can be broken down into three steps:

- Query the source object for IConnectionPointContainer.
- Call IConnectionPointContainer::FindConnectionPoint passing the IID of the event interface that interests you. If successful, this will return the IConnectionPoint interface on a connection point object.
- Call IConnectionPoint::Advise passing the **IUnknown** of the event sink. If successful, this will return a **DWORD** cookie representing the connection.

Once you have successfully registered your interest in receiving events, methods on your object's event interface will be called according to the events fired by the source object. When you no longer need to receive events, you can pass the cookie back to the connection point via IConnectionPoint::Unadvise. This will break the connection between source and sink.

Be careful to avoid reference cycles when handling events.

See Also

Concepts

Event Handling and ATL

Implementing the Event Handling Interface

ATL helps you with all three elements required for handling events: implementing the event interface, advising the event source, and unadvising the event source. The precise steps you'll need to take depend on the type of the event interface and the performance requirements of your application.

The most common ways of implementing an interface using ATL are:

- Deriving from a custom interface directly.
- Deriving from IDispatchImpl for dual interfaces described in a type library.
- Deriving from IDispEventImpl for dispinterfaces described in a type library.
- Deriving from IDispEventSimpleImpl for dispinterfaces not described in a type library or when you want to improve efficiency by not loading the type information at run time.

If you are implementing a custom or dual interface, you should advise the event source by calling AtlAdvise or CComPtrBase::Advise. You will need to keep track of the cookie returned by the call yourself. Call AtlUnadvise to break the connection.

If you are implementing a dispinterface using **IDispEventImpl** or **IDispEventSimpleImpl**, you should advise the event source by calling IDispEventSimpleImpl::DispEventAdvise. Call IDispEventSimpleImpl::DispEventUnadvise to break the connection.

If you are using **IDispEventImpl** as a base class of a composite control, the event sources listed in the sink map will be advised and unadvised automatically using CComCompositeControl::AdviseSinkMap.

The **IDispEventImpl** and **IDispEventSimpleImpl** classes manage the cookie for you.

See Also

Concepts

Event Handling and ATL

Using IDispEventImpl

When using **IDispEventImpl** to handle events, you will need to:

- Derive your class from IDispEventImpl.
- Add an event sink map to your class.
- Add entries to the event sink map using the SINK_ENTRY or SINK_ENTRY_EX macro.
- Implement the methods that you're interested in handling.
- Advise and unadvise the event source.

Example

The example below shows how to handle the **DocumentChange** event fired by Word's **Application** object. This event is defined as a method on the **ApplicationEvents** dispinterface.

The example is from the ATLEventHandling sample.

```
[
  uuid(000209F7-0000-0000-C000-000000000046),
  hidden
]
dispinterface ApplicationEvents {
    properties:
    methods:
        [id(0x00000001), restricted, hidden]
        void Startup();
        [id(0x00000002)]
        void Quit();
        [id(0x00000003)]
        void DocumentChange();
};
```

The example uses **#import** to generate the required header files from Word's type library. If you want to use this example with other versions of Word, you must specify the correct mso dll file. For example, Office 2000 provides mso9.dll and OfficeXP provides mso.dll. This code is from stdafx.h.

```
#pragma warning (disable : 4146)
#import "C:\PROGRAM FILES\MICROSOFT OFFICE\OFFICE\MSO97.DLL" raw_interfaces_only
#import "C:\PROGRAM FILES\COMMON FILES\MICROSOFT SHARED\VBA\VBEEXT1.OLB" raw_interfaces_onl
y
#import "C:\PROGRAM FILES\MICROSOFT OFFICE\OFFICE\MSWORD8.OLB" rename("ExitWindows", "WordE
xitWindows") raw_interfaces_only
#pragma warning (default : 4146)
```

The following code appears in NotSoSimple.h. The relevant code is in bold:

```
        // #import doesn't generate a LIBID (since we don't use 'named_guids'),
// so we have to do it manually
namespace Word
{
   struct __declspec(uuid("00020905-0000-0000-C000-000000000046"))
      /* library */ Library;
};

class ATL_NO_VTABLE CNotSoSimple :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<CNotSoSimple, &CLSID_NotSoSimple>,
   public IDispatchImpl<ISwitch, &IID_ISwitch, &LIBID_DISPEVENTLib>,
   public IDispEventImpl</*nID*/ 1, CNotSoSimple,
```

```
            &__uuidof(Word::ApplicationEvents),
            &__uuidof(Word::Library), /*wMajor*/ 8, /*wMinor*/ 0>
{
public:
   CNotSoSimple()
   {
   }

DECLARE_REGISTRY_RESOURCEID(IDR_NOTSOSIMPLE)

DECLARE_PROTECT_FINAL_CONSTRUCT()

BEGIN_COM_MAP(CNotSoSimple)
   COM_INTERFACE_ENTRY(ISwitch)
   COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()

   CComPtr<Word::_Application> m_pApp;

   // Event handler
   // Note the __stdcall calling convention and
   // dispinterface-style signature
   void __stdcall OnDocChange()
   {
      // Get a pointer to the _Document interface on the active document
      CComPtr<Word::_Document> pDoc;
      m_pApp->get_ActiveDocument(&pDoc);

      // Get the name from the active document
      CComBSTR bstrName;
      pDoc->get_Name(&bstrName);

      // Create a display string
      CComBSTR bstrDisplay(_T("New document title:\n"));
      bstrDisplay += bstrName;

      // Display the name to the user
      USES_CONVERSION;
      MessageBox(NULL, W2CT(bstrDisplay), _T("Active Document Changed"), MB_OK);
   }

BEGIN_SINK_MAP(CNotSoSimple)
   SINK_ENTRY_EX(/*nID =*/ 1, __uuidof(Word::ApplicationEvents), /*dispid =*/ 3, OnDocChang
e)
END_SINK_MAP()

// ISwitch
public:
   STDMETHOD(Start)()
   {
      // If you already have an object, just return
      if (m_pApp)
         return S_OK;

      // Create an instance of Word's Application object
      m_pApp.CoCreateInstance(__uuidof(Word::Application), NULL, CLSCTX_SERVER);

      // Make the Word user interface visible
      m_pApp->put_Visible(true);

      // Forge a connection to enable you to receive events
      DispEventAdvise(m_pApp);

      return S_OK;
   }

   STDMETHOD(Stop)()
   {
```

```
        // Check you have an object to unadvise on
        if (!m_pApp)
           return S_OK;

        // Break the connection with the event source
        DispEventUnadvise(m_pApp);

        // Release the Word application
        m_pApp.Release();

        return S_OK;
    }
};
```

See Also

Tasks

ATLEventHandling Sample: Handles Word Events Using IDispEventImpl and IDispEventSimpleImpl

Concepts

Event Handling and ATL

Using IDispEventSimpleImpl

When using **IDispEventSimpleImpl** to handle events, you will need to:

- Derive your class from IDispEventSimpleImpl.
- Add an event sink map to your class.
- Define _ATL_FUNC_INFO structures describing the events.
- Add entries to the event sink map using the SINK_ENTRY_INFO macro.
- Implement the methods that you're interested in handling.
- Advise and unadvise the event source.

Example

The example below shows you how to handle the **DocumentChange** event fired by Word's **Application** object. This event is defined as a method on the **ApplicationEvents** dispinterface.

The example is from the ATLEventHandling sample.

```
[
  uuid(000209F7-0000-0000-C000-000000000046),
  hidden
]
dispinterface ApplicationEvents {
    properties:
    methods:
        [id(0x00000001), restricted, hidden]
        void Startup();
        [id(0x00000002)]
        void Quit();
        [id(0x00000003)]
        void DocumentChange();
};
```

The example uses **#import** to generate the required header files from Word's type library. If you want to use this example with other versions of Word, you must specify the correct mso dll file. For example, Office 2000 provides mso9.dll and OfficeXP provides mso.dll. This code is from stdafx.h.:

```
#pragma warning (disable : 4146)
#import "C:\PROGRAM FILES\MICROSOFT OFFICE\OFFICE\MSO97.DLL" raw_interfaces_only
#import "C:\PROGRAM FILES\COMMON FILES\MICROSOFT SHARED\VBA\VBEEXT1.OLB" raw_interfaces_onl
y
#import "C:\PROGRAM FILES\MICROSOFT OFFICE\OFFICE\MSWORD8.OLB" rename("ExitWindows", "WordE
xitWindows") raw_interfaces_only
#pragma warning (default : 4146)
```

The only information from the type library actually used in this example is the CLSID of the Word **Application** object and the IID of the **ApplicationEvents** interface. This information is only used at compile time.

The following code appears in Simple.h. The relevant code is in bold:

```

        // Declare structure for type information
extern _ATL_FUNC_INFO OnDocChangeInfo;

class ATL_NO_VTABLE CSimple :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<CSimple, &CLSID_Simple>,
   public IDispatchImpl<ISwitch, &IID_ISwitch, &LIBID_DISPEVENTLib>,
   public IDispEventSimpleImpl</*nID =*/ 1, CSimple, &__uuidof(Word::ApplicationEvents)>
{
```

```
public:
   CSimple()
   {
   }

DECLARE_REGISTRY_RESOURCEID(IDR_SIMPLE)

DECLARE_PROTECT_FINAL_CONSTRUCT()

BEGIN_COM_MAP(CSimple)
   COM_INTERFACE_ENTRY(ISwitch)
   COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()

   CComPtr<Word::_Application> m_pApp;

   // Event handler
   // Note the __stdcall calling convention and
   // dispinterface-style signature
   void __stdcall OnDocChange()
   {
      // Get a pointer to the _Document interface on the active document
      CComPtr<Word::_Document> pDoc;
      m_pApp->get_ActiveDocument(&pDoc);

      // Get the name from the active document
      CComBSTR bstrName;
      pDoc->get_Name(&bstrName);

      // Create a display string
      CComBSTR bstrDisplay(_T("New document title:\n"));
      bstrDisplay += bstrName;

      // Display the name to the user
      USES_CONVERSION;
      MessageBox(NULL, W2CT(bstrDisplay), _T("Active Document Changed"), MB_OK);
   }

BEGIN_SINK_MAP(CSimple)
   SINK_ENTRY_INFO(/*nID =*/ 1, __uuidof(Word::ApplicationEvents), /*dispid =*/ 3, OnDocCha
nge, &OnDocChangeInfo)
END_SINK_MAP()

// ISwitch
public:
   STDMETHOD(Start)()
   {
      // If you already have an object, just return
      if (m_pApp)
         return S_OK;

      // Create an instance of Word's Application object
      m_pApp.CoCreateInstance(__uuidof(Word::Application), NULL, CLSCTX_SERVER);

      // Make the Word user interface visible
      m_pApp->put_Visible(true);

      // Forge a connection to enable you to receive events
      DispEventAdvise(m_pApp);

      return S_OK;
   }

   STDMETHOD(Stop)()
   {
      // Check you have an object to unadvise on
      if (!m_pApp)
         return S_OK;
```

```
        // Break the connection with the event source
        DispEventUnadvise(m_pApp);

        // Release the Word application
        m_pApp.Release();

        return S_OK;
    }
};
```

The following code is from Simple.cpp:

```
        // Define type info structure
_ATL_FUNC_INFO OnDocChangeInfo = {CC_STDCALL, VT_EMPTY, 0};
```

See Also

Tasks

ATLEventHandling Sample: Handles Word Events Using IDispEventImpl and IDispEventSimpleImpl

Concepts

Event Handling and ATL

ATL Event Handling Summary

In general, handling COM events is a relatively simple process. There are three main steps:

- Implement the event interface on your object.
- Advise the event source that your object wants to receive events.
- Unadvise the event source when your object no longer needs to receive events.

Implementing the Interface

There are four main ways of implementing an interface using ATL.

Derive from	Suitable for Interface type	Requires you to implement all methods *	Requires a type library at run time
The interface	Vtable	Yes	No
IDispatchImpl	Dual	Yes	Yes
IDispEventImpl	Dispinterface	No	Yes
IDispEventSimpleImpl	Dispinterface	No	No

* When using ATL support classes, you are never required to implement the **IUnknown** or **IDispatch** methods manually.

Advising and Unadvising the Event Source

There are three main ways of advising and unadvising an event source using ATL.

Advise function	Unadvise function	Most suitable for use with	Requires you to keep track of a cookie?	Comments

AtlAdvise, CComPtrBase::Advise	AtlUnadvise	Vtable or dual interfaces	Yes	**AtlAdvise** is a global ATL function. **CComPtrBase::Advise** is used by CComPtr and CComQIPtr.
IDispEventSimpleImpl::DispEventAdvise	IDispEventSimpleImpl::DispEventUnadvise	IDispEventImpl or IDispEventSimpleImpl	No	Fewer parameters than **AtlAdvise** since the base class does more work.
CComCompositeControl::AdviseSinkMap(TRUE)	CComCompositeControl::AdviseSinkMap(FALSE)	ActiveX controls in Composite controls	No	**CComCompositeControl::AdviseSinkMap** advises all entries in the event sink map. The same function unadvises the entries. This method is called automatically by the **CComCompositeControl** class.
CAxDialogImpl::AdviseSinkMap(TRUE)	CAxDialogImpl::AdviseSinkMap(FALSE)	ActiveX controls in a dialog box	No	**CAxDialogImpl::AdviseSinkMap** advises and unadvises all ActiveX controls in the dialog resource. This is done automatically for you.

See Also

Concepts

Event Handling and ATL

Supporting IDispEventImpl

ATL and the Free Threaded Marshaler

The ATL Simple Object Wizard's Attributes page provides an option that allows your class to aggregate the free threaded marshaler (FTM).

The wizard generates code to create an instance of the free threaded marshaler in **FinalConstruct** and release that instance in **FinalRelease**. A **COM_INTERFACE_ENTRY_AGGREGATE** macro is automatically added to the COM map to ensure that **QueryInterface** requests for IMarshal are handled by the free threaded marshaler.

The free threaded marshaler allows direct access to interfaces on your object from any thread in the same process, speeding up cross-apartment calls. This option is intended for classes that use the Both threading model.

When using this option, classes must take responsibility for the thread-safety of their data. In addition, objects that aggregate the free threaded marshaler and need to use interface pointers obtained from other objects must take extra steps to ensure that the interfaces are correctly marshaled. Typically this involves storing the interface pointers in the global interface table (GIT) and getting the pointer from the GIT each time it is used. ATL provides the class CComGITPtr to help you use interface pointers stored in the GIT.

See Also

Reference

CoCreateFreeThreadedMarshaler
IMarshal
When to Use the Global Interface Table
In-Process Server Threading Issues

Other Resources

ATL Concepts

Specifying the Project's Threading Model

The following macros are available to specify the threading model of an ATL project:

Macro	Guidelines for using
_ATL_SINGLE_THREADED	Define if all of your objects use the single threading model.
_ATL_APARTMENT_THREADED	Define if one or more of your objects use apartment threading.
_ATL_FREE_THREADED	Define if one or more of your objects use free or neutral threading. Existing code may contain references to the equivalent macro _ATL_MULTI_THREADED.

If you do not define any of these macros for your project, _ATL_FREE_THREADED will be in effect.

The macros affect run-time performance as follows:

- Specifying the macro that corresponds to the objects in your project can improve run-time performance.
- Specifying a higher level of macro, for example if you specify _ATL_APARTMENT_THREADED when all of your objects are single threaded, will slightly degrade run-time performance.
- Specifying a lower level of macro, for example, if you specify _ATL_SINGLE_THREADED when one or more of your objects use apartment threading or free threading, can cause your application to fail at run time.

See Options, ATL Simple Object Wizard for a description of the threading models available for an ATL object.

See Also

Other Resources

ATL Concepts

ATL Module Classes

This topic discusses the module classes that were new in ATL 7.0.

CComModule Replacement Classes

Earlier versions of ATL used **CComModule**. In ATL 7.0, **CComModule** functionality is replaced by several classes:

- CAtlBaseModule Contains information required by most applications that use ATL. Contains the HINSTANCE of the module and the resource instance.
- CAtlComModule Contains information required by the COM classes in ATL.
- CAtlWinModule Contains information required by the windowing classes in ATL.
- CAtlDebugInterfacesModule Contains support for interface debugging.
- CAtlModule The following **CAtlModule**-derived classes are customized to contain information required in a particular application type. Most members in these classes can be overridden:
    - CAtlDllModuleT Used in DLL applications. Provides code for the standard exports.
    - CAtlExeModuleT Used in EXE applications. Provides code required in an EXE.
    - CAtlServiceModuleT Provides support to create Windows NT and Windows 2000 Services.

CComModule is still available for backward compatibility.

Reasons for Distributing CComModule Functionality

The functionality of **CComModule** was distributed into several new classes for the following reasons:

- Make the functionality in **CComModule** granular.

  Support for COM, windowing, interface debugging, and application-specific (DLL or EXE) features is now in separate classes.

- Automatically declare global instance of each of these modules.

  A global instance of the required module classes is linked into the project.

- Remove the necessity of calling Init and Term methods.

  Init and Term methods have moved into the constructors and destructors for the module classes; there is no longer a need to call Init and Term.

See Also

Other Resources

ATL Concepts

ATL Class Overview

ATL Services

To create your ATL COM object so that it runs in a service, simply select Service (EXE) from the list of server options in the ATL Project Wizard. The wizard will then create a class derived from **CAtlServiceModuleT** to implement the service.

When the ATL COM object is built as a service, it will only be registered as a local server, and it will not appear in the list of services in Control Panel. This is because it is easier to debug the service as a local server than as a service. To install it as a service, run the following at the command prompt:

```
YourEXE.exe /Service
```

To uninstall it, run the following:

```
YourEXE.exe /UnregServer
```

The first four topics in this section discuss the actions that occur during execution of **CAtlServiceModuleT** member functions. These topics appear in the same sequence as the functions are typically called. To improve your understanding of these topics, it is a good idea to use the source code generated by the ATL Project Wizard as a reference. These first four topics are:

- The CAtlServiceModuleT::Start Function
- The CAtlServiceModuleT::ServiceMain Function
- The CAtlServiceModuleT::Run Function
- The CAtlServiceModuleT::Handler Function

The last three topics discuss concepts related to developing a service:

- Registry Entries for ATL services
- DCOMCNFG
- Debugging Tips for ATL services

See Also
Other Resources
ATL Concepts

The CAtlServiceModuleT::Start Function

When the service is run, **_tWinMain** calls **CAtlServiceModuleT::WinMain**, which in turn calls **CAtlServiceModuleT::Start**.

CAtlServiceModuleT::Start sets up an array of **SERVICE_TABLE_ENTRY** structures that map each service to its startup function. This array is then passed to the Win32 API function, StartServiceCtrlDispatcher. In theory, one EXE could handle multiple services and the array could have multiple **SERVICE_TABLE_ENTRY** structures. Currently, however, an ATL-generated service supports only one service per EXE. Therefore, the array has a single entry that contains the service name and **_ServiceMain** as the startup function. **_ServiceMain** is a static member function of **CAtlServiceModuleT** that calls the non-static member function, **ServiceMain**.

Note
Failure of **StartServiceCtrlDispatcher** to connect to the service control manager (SCM) probably means that the program is not running as a service. In this case, the program calls **CAtlServiceModuleT::Run** directly so that the program can run as a local server. For more information about running the program as a local server, see Debugging Tips.

See Also

Reference

ATL Services

CAtlServiceModuleT::Start

The CAtlServiceModuleT::ServiceMain Function

The service control manager (SCM) calls **ServiceMain** when you open the Services Control Panel application, select the service, and click **Start**.

After the SCM calls **ServiceMain**, a service must give the SCM a handler function. This function lets the SCM obtain the service's status and pass specific instructions (such as pausing or stopping). The SCM gets this function when the service passes **_Handler** to the Win32 API function, RegisterServiceCtrlHandler. (**_Handler** is a static member function that calls the non-static member function Handler.)

At startup, a service should also inform the SCM of its current status. It does this by passing **SERVICE_START_PENDING** to the Win32 API function, SetServiceStatus.

ServiceMain then calls **CAtlExeModuleT::InitializeCom**, which calls the Win32 API function CoInitializeEx. By default, **InitializeCom** passes the **COINIT_MULTITHREADED** flag to the function. This flag indicates that the program is to be a free-threaded server.

Now, **CAtlServiceModuleT::Run** is called to perform the main work of the service. **Run** continues to execute until the service is stopped.

See Also

Reference

ATL Services

CAtlServiceModuleT::ServiceMain

The CAtlServiceModuleT::Run Function

Run contains calls to **PreMessageLoop**, **RunMessageLoop**, and **PostMessageLoop**. After being called, **PreMessageLoop** first stores the service's thread ID. The service will use this ID to close itself by sending a **WM_QUIT** message using the Win32 API function, PostThreadMessage.

PreMessageLoop then calls **InitializeSecurity**. By default, **InitializeSecurity** calls CoInitializeSecurity with the security descriptor set to NULL, which means that any user has access to your object.

If you do not want the service to specify its own security, override **PreMessageLoop** and don't call **InitializeSecurity**, and COM will then determine the security settings from the registry. A convenient way to configure registry settings is with the DCOMCNFG utility discussed later in this section.

Once security is specified, the object is registered with COM so that new clients can connect to the program. Finally, the program tells the service control manager (SCM) that it is running and the program enters a message loop. The program remains running until it posts a quit message upon service shutdown.

For more information about Windows NT security, see the MSDN article, Windows NT Security in Theory and Practice.

See Also

Reference

ATL Services
CSecurityDesc Class
CSid Class
CDacl Class
CAtlServiceModuleT::Run

The CAtlServiceModuleT::Handler Function

CAtlServiceModuleT::Handler is the routine that the service control manager (SCM) calls to retrieve the status of the service and give it various instructions (such as stopping or pausing). The SCM passes an operation code to **Handler** to indicate what the service should do. A default ATL-generated service only handles the stop instruction. If the SCM passes the stop instruction, the service tells the SCM that the program is about to stop. The service then calls **PostThreadMessage** to post a quit message to itself. This terminates the message loop and the service will ultimately close.

To handle more instructions, you need to change the **m_status** data member initialized in the **CAtlServiceModuleT** constructor. This data member tells the SCM which buttons to enable when the service is selected in the Services Control Panel application.

See Also

Reference

ATL Services

CAtlServiceModuleT::Handler

Registry Entries

DCOM introduced the concept of Application IDs (AppIDs), which group configuration options for one or more DCOM objects into a centralized location in the registry. You specify an AppID by indicating its value in the AppID named value under the object's CLSID.

By default, an ATL-generated service uses its CLSID as the GUID for its AppID. Under `HKEY_CLASSES_ROOT\AppID`, you can specify DCOM-specific entries. Initially, two entries exist:

- `LocalService`, with a value equal to the name of the service. If this value exists, it is used instead of the `LocalServer32` key under the CLSID.
- `ServiceParameters`, with a value equal to `-Service`. This value specifies parameters that will be passed to the service when it is started. Note that these parameters are passed to the service's **ServiceMain** function, not **WinMain**.

Any DCOM service also needs to create another key under `HKEY_CLASSES_ROOT\AppID`. This key is equal to the name of the EXE and acts as a cross-reference, as it contains an AppID value pointing back to the AppID entries.

See Also
Reference
ATL Services

DCOMCNFG

DCOMCNFG is a Windows NT 4.0 utility that allows you to configure various DCOM-specific settings in the registry. The **DCOMCNFG** window has three pages: Default Security, Default Properties, and Applications. Under Windows 2000 a fourth page, Default Protocols, is present.

Default Security Page

You can use the Default Security page to specify default permissions for objects on the system. The Default Security page has three sections: Access, Launch, and Configuration. To change a section's defaults, click the corresponding **Edit Default** button. These Default Security settings are stored in the registry under `HKEY_LOCAL_MACHINE\Software\Microsoft\OLE`.

Default Protocols Page

This page lists the set of network protocols available to DCOM on this machine. The order reflects the priority in which they will be used; the first in the list has the highest priority. Protocols can be added or deleted from this page.

Default Properties Page

On the Default Properties page, you must select the **Enable Distributed COM on this computer** check box if you want clients on other machines to access COM objects running on this machine. Selecting this option sets the `HKEY_LOCAL_MACHINE\Software\Microsoft\OLE\EnableDCOM` value to `Y`.

Applications Page

You change the settings for a particular object with the Applications page. Simply select the application from the list and click the **Properties** button. The Properties window has five pages:

- The General page confirms the application you are working with.
- The Location page allows you to specify where the application should run when a client calls **CoCreateInstance** on the relevant CLSID. If you select the **Run application on the following computer** check box and enter a computer name, then a `RemoteServerName` value is added under the AppID for that application. Clearing the **Run application on this computer** check box renames the `LocalService` value to `_LocalService` and, thereby, disables it.
- The Security page is similar to the Default Security page found in the **DCOMCNFG** window, except that these settings apply only to the current application. Again, the settings are stored under the AppID for that object.
- The Identify page identifies which user is used to run the application.
- The Endpoints page lists the set of protocols and endpoints available for use by clients of the selected DCOM server.

See Also
Reference
ATL Services

Debugging Tips

The following topics outline some useful steps for debugging your service:

- Using Task Manager
- Displaying Assertions
- Running the Program as a Local Server

See Also

Reference

ATL Services

Using Task Manager

One of the simplest ways to debug a service is through the use of the Task Manager in Windows NT 4.0 or Windows 2000. While the service is running, start the Task Manager and click the **Processes** tab. Right-click the name of the EXE and then click **Debug**. This launches Visual C++ attached to that running process. Now, click **Break** on the **Debug** menu to allow you to set breakpoints in your code. Click **Run** to run to your selected breakpoints.

See Also

Reference

Debugging Tips

Displaying Assertions

If the client connected to your service appears to hang, the service may have asserted and displayed a message box that you are not able to see. You can confirm this by using Visual C++'s debugger to debug your code (see Using Task Manager earlier in this section).

If you determine that your service is displaying a message box that you cannot see, you may want to set the **Allow Service to Interact with Desktop** option before using the service again. This option is a startup parameter that permits any message boxes displayed by the service to appear on the desktop. To set this option, open the Services Control Panel application, select the service, click **Startup**, and then select the **Allow Service to Interact with Desktop** option.

See Also

Reference

Debugging Tips

Running the Program as a Local Server

If running the program as a service is inconvenient, you can temporarily change the registry so that the program is run as a normal local server. Simply rename the `LocalService` value under your AppID to `_LocalService` and ensure the `LocalServer32` key under your CLSID is set correctly. (Note that using DCOMCNFG to specify that your application should be run on a different computer renames your `LocalServer32` key to `_LocalServer32`.) Running your program as a local server takes a few more seconds on startup because the call to **StartServiceCtrlDispatcher** in **CAtlServiceModuleT::Start** takes a few seconds before it fails.

See Also

Reference

Debugging Tips

ATL Window Classes

ATL includes several classes that allow you to use and implement windows. These classes, like other ATL classes, provide an efficient implementation that does not impose an overhead on your code.

This section describes the ATL window classes and explains how to use them.

In This Section

Introduction to ATL Window Classes

Briefly describes each ATL window class and provides links to the reference material on them.

Using a Window

Discusses how to use **CWindow** to manipulate a window.

Implementing a Window

Discusses message handlers, message maps, and using **CWindowImpl**. Includes details on superclassing and subclassing.

Implementing a Dialog Box

Discusses the two methods for adding a dialog box class and shows a code sample.

Using Contained Windows

Discusses contained windows in ATL, which are windows that delegate their messages to a container object instead of handling them in their own class.

Understanding Window Traits

Discusses window traits classes in ATL. These classes provide a simple method for standardizing the styles used for the creation of a window object.

Related Sections

ATL

Provides links to conceptual topics on how to program using the Active Template Library.

Windows Support Classes

Lists additional ATL classes that support windows and message maps in ATL.

Introduction to ATL Window Classes

The following ATL classes are designed to implement and manipulate windows:

- CWindow allows you to attach a window handle to the **CWindow** object. You then call **CWindow** methods to manipulate the window.
- CWindowImpl allows you to implement a new window and process messages with a message map. You can create a window based on a new Windows class, superclass an existing class, or subclass an existing window.
- CDialogImpl allows you to implement a modal or a modeless dialog box and process messages with a message map.
- CContainedWindowT is a prebuilt class that implements a window whose message map is contained in another class. Using **CContainedWindowT** allows you to centralize message processing in one class.
- CAxDialogImpl allows you to implement a dialog box (modal or modeless) that hosts ActiveX controls.
- CSimpleDialog allows you to implement a modal dialog box with basic functionality.
- CAxWindow allows you to implement a window that hosts an ActiveX control.
- CAxWindow2T allows you to implement a window that hosts a licensed ActiveX control.

In addition to specific window classes, ATL provides several classes designed to make the implementation of an ATL window object easier. They are as follows:

- CWndClassInfo manages the information of a new window class.
- CWinTraits and CWinTraitsOR provide a simple method of standardizing the traits of an ATL window object.

See Also

Other Resources

ATL Window Classes

Using a Window

Class CWindow allows you to use a window. Once you attach a window to a **CWindow** object, you can then call **CWindow** methods to manipulate the window. **CWindow** also contains an **HWND** operator to convert a **CWindow** object to an **HWND**. Thus you can pass a **CWindow** object to any function that requires a handle to a window. You can easily mix **CWindow** method calls and Win32 function calls, without creating any temporary objects.

Because **CWindow** has only two data member (a window handle and the default dimensions), it does not impose an overhead on your code. In addition, many of the **CWindow** methods simply wrap corresponding Win32 API functions. By using **CWindow**, the **HWND** member is automatically passed to the Win32 function.

In addition to using **CWindow** directly, you can also derive from it to add data or code to your class. ATL itself derives three classes from **CWindow**: CWindowImpl, CDialogImpl, and CContainedWindowT.

See Also

Other Resources

ATL Window Classes

Implementing a Window

Class CWindowImpl allows you to implement a window and handle its messages. Message handing in ATL is based on a message map. This section explains:

- How to add a message handler to a control.
- What message maps are and how to use them.
- The syntax for message handler functions.
- How to implement a window with CWindowImpl.

See Also

Other Resources

ATL Window Classes

Adding an ATL Message Handler

To add a message handler (a member function that handles Windows messages) to a control, first select the control in the Class View. Then open the **Properties** window, select the **Messages** icon, and click the drop-down control in the box opposite the required message. This will add a declaration for the message handler in the control's header file and a skeleton implementation of the handler in the control's .cpp file. It will also add the message map and add an entry for the handler.

Adding a message handler in ATL is similar to adding a message handler to an MFC class. See Adding an MFC Message Handler for more information.

The following conditions apply only to adding an ATL message handler:

- The message handlers follow the same naming convention as MFC.
- The new message map entries are added into the main message map. The wizard does not recognize alternate message maps and chaining.

See Also
Reference
Implementing a Window

Message Maps (ATL)

A message map associates a handler function with a particular message, command, or notification. By using ATL's message map macros, you can specify a message map for a window. The window procedures in **CWindowImpl**, **CDialogImpl**, and **CContainedWindowT** direct a window's messages to its message map.

The message handler functions accept an additional argument of type `BOOL&`. This argument indicates whether a message has been processed, and it is set to `TRUE` by default. A handler function can then set the argument to `FALSE` to indicate that it has not handled a message. In this case, ATL will continue to look for a handler function further in the message map. By setting this argument to `FALSE`, you can first perform some action in response to a message and then allow the default processing or another handler function to finish handling the message.

Chained Message Maps

ATL also allows you to chain message maps, which directs the message handling to a message map defined in another class. For example, you can implement common message handling in a separate class to provide uniform behavior for all windows chaining to that class. You can chain to a base class or to a data member of your class.

ATL also supports dynamic chaining, which allows you to chain to another object's message map at run time. To implement dynamic chaining, you must derive your class from CDynamicChain. Then declare the CHAIN_MSG_MAP_DYNAMIC macro in your message map. **CHAIN_MSG_MAP_DYNAMIC** requires a unique number that identifies the object and the message map to which you are chaining. You must define this unique value through a call to **CDynamicChain::SetChainEntry**.

You can chain to any class that declares a message map, provided the class derives from CMessageMap. **CMessageMap** allows an object to expose its message maps to other objects. Note that **CWindowImpl** already derives from **CMessageMap**.

Alternate Message Maps

Finally, ATL supports alternate message maps, declared with the ALT_MSG_MAP macro. Each alternate message map is identified by a unique number, which you pass to **ALT_MSG_MAP**. Using alternate message maps, you can handle the messages of multiple windows in one map. Note that by default, **CWindowImpl** does not use alternate message maps. To add this support, override the **WindowProc** method in your **CWindowImpl**-derived class and call **ProcessWindowMessage** with the message map identifier.

See Also

Reference

Implementing a Window

Message Handler Functions

ATL provides three types of message handler functions:

Type of message handler	Corresponding message macro
MessageHandler	MESSAGE_HANDLER
CommandHandler	COMMAND_HANDLER
NotifyHandler	NOTIFY_HANDLER

See Also

Reference

Implementing a Window
Message Maps (ATL)
WM_NOTIFY

CommandHandler

CommandHandler is the function identified by the third parameter of the **COMMAND_HANDLER** macro in your message map.

```
LRESULT CommandHandler(
   WORD wNotifyCode,
   WORD wID,
   HWND hWndCtl,
   BOOL& bHandled
);
```

Parameters

wNotifyCode

The notification code.

wID

The identifier of the menu item, control, or accelerator.

hWndCtl

A handle to a window control.

bHandled

The message map sets *bHandled* to **TRUE** before *CommandHandler* is called. If *CommandHandler* does not fully handle the message, it should set *bHandled* to **FALSE** to indicate the message needs further processing.

Return Value

The result of message processing. 0 if successful.

Remarks

For an example of using this message handler in a message map, see COMMAND_HANDLER.

See Also

Reference

Implementing a Window
Message Maps (ATL)
WM_NOTIFY

MessageHandler

MessageHandler is the name of the function identified by the second parameter of the **MESSAGE_HANDLER** macro in your message map.

```
LRESULT MessageHandler(
   UINT uMsg,
   WPARAM wParam,
   LPARAM lParam,
   BOOL& bHandled
);
```

Parameters

uMsg

Specifies the message.

wParam

Additional message-specific information.

lParam

Additional message-specific information.

bHandled

The message map sets *bHandled* to **TRUE** before *MessageHandler* is called. If *MessageHandler* does not fully handle the message, it should set *bHandled* to **FALSE** to indicate the message needs further processing.

Return Value

The result of message processing. 0 if successful.

Remarks

For an example of using this message handler in a message map, see MESSAGE_HANDLER.

See Also

Reference

Implementing a Window
Message Maps (ATL)
WM_NOTIFY

NotifyHandler

The name of the function identified by the third parameter of the **NOTIFY_HANDLER** macro in your message map.

```
LRESULT NotifyHandler(
   int idCtrl,
   LPNMHDR pnmh,
   BOOL& bHandled
);
```

Parameters

idCtrl

The identifier of the control sending the message.

pnmh

Address of an NMHDR structure that contains the notification code and additional information. For some notification messages, this parameter points to a larger structure that has the **NMHDR** structure as its first member.

bHandled

The message map sets *bHandled* to **TRUE** before *NotifyHandler* is called. If *NotifyHandler* does not fully handle the message, it should set *bHandled* to **FALSE** to indicate the message needs further processing.

Return Value

The result of message processing. 0 if successful.

Remarks

For an example of using this message handler in a message map, see NOTIFY_HANDLER.

See Also

Reference

Implementing a Window
Message Maps (ATL)
WM_NOTIFY

Implementing a Window with CWindowImpl

To implement a window, derive a class from **CWindowImpl**. In your derived class, declare a message map and the message handler functions. You can now use your class in three different ways:

- Create a window based on a new Windows class
- Superclass an existing Windows class
- Subclass an existing window

Creating a Window Based on a New Windows Class

CWindowImpl contains the DECLARE_WND_CLASS macro to declare Windows class information. This macro implements the **GetWndClassInfo** function, which uses CWndClassInfo to define the information of a new Windows class. When **CWindowImpl::Create** is called, this Windows class is registered and a new window is created.

Note
CWindowImpl passes **NULL** to the **DECLARE_WND_CLASS** macro, which means ATL will generate a Windows class name. To specify your own name, pass a string to **DECLARE_WND_CLASS** in your **CWindowImpl**-derived class.

Example

Following is an example of a class that implements a window based on a new Windows class:

```
class CMyWindow : public CWindowImpl<CMyWindow>, ...
{
public:
   // Optionally specify name of the new Windows class
   DECLARE_WND_CLASS("MyName")
              // If this macro is not specified in your
              // class, ATL will generate a class name
   ...

   BEGIN_MSG_MAP(CMyWindow)
      MESSAGE_HANDLER(WM_PAINT, OnPaint)
   END_MSG_MAP()

   LRESULT OnPaint(UINT nMsg, WPARAM wParam,
                   LPARAM lParam, BOOL& bHandled)
   {
      // Do some painting code
      return 0;
   }
};
```

To create a window, create an instance of `CMyWindow` and then call the **Create** method.

Note
To override the default Windows class information, implement the **GetWndClassInfo** method in your derived class by settin g the **CWndClassInfo** members to the appropriate values.

Following is an example of a class that superclasses the standard Edit class:

```
class CMyEdit : public CWindowImpl<CMyEdit>, ...
{
public:
   // "Edit" is the name of the standard Windows class.
   // "MyEdit" is the name of the new Windows class
   // that will be based on the Edit class.
   DECLARE_WND_SUPERCLASS("MyEdit", "Edit")
```

```
    ...
    BEGIN_MSG_MAP(CMyEdit)
       MESSAGE_HANDLER(WM_CHAR, OnChar)
    END_MSG_MAP()

    LRESULT OnChar(UINT nMsg, WPARAM wParam,
                   LPARAM lParam, BOOL& bHandled)
    {
       // Do some character handling code
    }
 };
```

To create the superclassed Edit window, create an instance of `CMyEdit` and then call the **Create** method.

Superclassing an Existing Windows Class

The DECLARE_WND_SUPERCLASS macro allows you to create a window that superclasses an existing Windows class. Specify this macro in your **CWindowImpl**-derived class. Like any other ATL window, messages are handled by a message map.

When you use **DECLARE_WND_SUPERCLASS**, a new Windows class will be registered. This new class will be the same as the existing class you specify, but will replace the window procedure with **CWindowImpl::WindowProc** (or with your function that overrides this method).

Subclassing an Existing Window

To subclass an existing window, derive a class from **CWindowImpl** and declare a message map, as in the two previous cases. Note, however, that you do not specify any Windows class information, since you will subclass an already existing window.

Instead of calling **Create**, call **SubclassWindow** and pass it the handle to the existing window you want to subclass. Once the window is subclassed, it will use **CWindowImpl::WindowProc** (or your function that overrides this method) to direct messages to the message map. To detach a subclassed window from your object, call **UnsubclassWindow**. The window's original window procedure will then be restored.

See Also

Reference

Implementing a Window

Implementing a Dialog Box

There are two ways to add a dialog box to your ATL project: use the ATL Dialog Wizard or add it manually.

Adding a Dialog Box with the ATL Dialog Wizard

In the Add Class dialog box, select the ATL Dialog object to add a dialog box to your ATL project. Fill in the ATL Dialog Wizard as appropriate and click **Finish**. The wizard adds a class derived from CAxDialogImpl to your project. Open the Resource View from the **View** menu, locate your dialog, and double-click it to open it in the resource editor.

Note
If your dialog box is derived from **CAxDialogImpl**, it can host both ActiveX and Windows controls. If you don't want the overhead of ActiveX control support in your dialog box class, use CSimpleDialog or CDialogImpl instead.

Message and event handlers can be added to your dialog class from Class View. For more information, see Adding an ATL Message Handler.

Adding a Dialog Box Manually

Implementing a dialog box is similar to implementing a window. You derive a class from either CAxDialogImpl, CDialogImpl, or CSimpleDialog and declare a message map to handle messages. However, you must also specify a dialog template resource ID in your derived class. Your class must have a data member called `IDD` to hold this value.

Note
When you create a dialog box using the ATL Dialog Wizard, the wizard automatically adds the `IDD` member as an **enum** type.

CDialogImpl allows you to implement a modal or a modeless dialog box that hosts Windows controls. **CAxDialogImpl** allows you to implement a modal or a modeless dialog box that hosts both ActiveX and Windows controls.

To create a modal dialog box, create an instance of your **CDialogImpl**-derived (or **CAxDialogImpl**-derived) class and then call the DoModal method. To close a modal dialog box, call the EndDialog method from a message handler. To create a modeless dialog box, call the Create method instead of **DoModal**. To destroy a modeless dialog box, call DestroyWindow.

Sinking events is automatically done in CAxDialogImpl. Implement the dialog box's message handlers as you would the handlers in a **CWindowImpl**-derived class. If there is a message-specific return value, return it as an `LRESULT`. The returned `LRESULT` values are mapped by ATL for proper handling by the Windows dialog manager. For details, see the source code for CDialogImplBaseT::DialogProc in atlwin.h.

Example

The following class implements a dialog box:

```
class CMyDialog : public CDialogImpl<CMyDialog>, ...
{
public:
   enum { IDD = IDD_MYDIALOG };

   BEGIN_MSG_MAP(CMyDialog)
      MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
   END_MSG_MAP()

   LRESULT OnInitDialog(UINT uMsg, WPARAM wParam,
                        LPARAM lParam, BOOL& bHandled)
   {
      // Do some initialization code
      return 1;
   }
};
```

See Also

Other Resources

ATL Window Classes

Using Contained Windows

ATL implements contained windows with CContainedWindowT. A contained window represents a window that delegates its messages to a container object instead of handling them in its own class.

Note
You do not need to derive a class from **CContainedWindowT** in order to use contained windows.

With contained windows, you can either superclass an existing Windows class or subclass an existing window. To create a window that superclasses an existing Windows class, first specify the existing class name in the constructor for the **CContainedWindowT** object. Then call **CContainedWindowT::Create**. To subclass an existing window, you don't need to specify a Windows class name (pass **NULL** to the constructor). Simply call the **CContainedWindowT::SubclassWindow** method with the handle to the window being subclassed.

You typically use contained windows as data members of a container class. The container does not need to be a window; however, it must derive from CMessageMap.

A contained window can use alternate message maps to handle its messages. If you have more than one contained window, you should declare several alternate message maps, each corresponding to a separate contained window.

Example

Following is an example of a container class with two contained windows:

```
class CMyContainer : public CMessageMap, ...
{
public:
   CContainedWindow m_wndEdit;
   CContainedWindow m_wndList;

   CMyContainer() : m_wndEdit("Edit", this, 1),
                    m_wndList("List", this, 2)
   {
   }

   ...

   BEGIN_MSG_MAP(CMyContainer)
   ALT_MSG_MAP(1)
      // handlers for the Edit window go here
   ALT_MSG_MAP(2)
      // handlers for the List window go here
   END_MSG_MAP()

};
```

For more information about contained windows, see the SUBEDIT sample.

See Also

Other Resources

ATL Window Classes

Understanding Window Traits

Window traits classes provide a simple method for standardizing the styles used for the creation of an ATL window object. Window traits are accepted as template parameters by CWindowImpl and other ATL window classes as a way of providing default window styles at the class level.

If the creator of a window instance doesn't provide styles explicitly in the call to Create, you can use a traits class to ensure that the window is still created with the correct styles. You can even ensure that certain styles are set for all instances of that window class while permitting other styles to be set on a per-instance basis.

ATL Window Traits Templates

ATL provides two window traits templates that allow you to set default styles at compile time using their template parameters.

Class	Description
CWinTraits	Use this template when you want to provide default window styles that will be used only when no other styles a re specified in the call to **Create**. The styles provided at run time take precedence over the styles set at compile t ime.
CWinTraitsOR	Use this class when you want to specify styles that must always be set for the window class. The styles provided at run time are combined with the styles set at compile time using the bitwise OR operator.

In addition to these templates, ATL provides a number of predefined specializations of the **CWinTraits** template for commonly used combinations of window styles. See the CWinTraits reference documentation for full details.

Custom Window Traits

In the unlikely situation that specializing one of the templates provided by ATL isn't sufficient and you need to create your own traits class, you just need to create a class that implements two static functions: **GetWndStyle** and **GetWndStyleEx**:

```
static DWORD GetWndStyle(DWORD dwStyle);
static DWORD GetWndExStyle(DWORD dwExStyle);
```

Each of these functions will be passed some style value at run time which it can use to produce a new style value. If your window traits class is being used as the template argument to an ATL window class, the style values passed to these static functions will be whatever was passed as the style arguments to Create.

See Also

Other Resources

ATL Window Classes

ATL Collection Classes

ATL provides many classes for storing and accessing data. Which class you decide to use depends on several factors, including:

- The amount of data to be stored
- Efficiency versus performance in accessing the data
- The ability to access the data by index or by key
- How the data is ordered
- Personal preference

Small Collection Classes

ATL provides the following array classes for dealing with small numbers of objects. However, these classes are limited and designed for use internally by ATL. It is not recommended that you use them in your programs.

Class	Type of data storage
CSimpleArray	Implements an array class for dealing with small numbers of objects.
CSimpleMap	Implements a mapping class for dealing with small numbers of objects.

General Purpose Collection Classes

The follow classes implement arrays, lists, and maps and are provided as general purpose collection classes:

Class	Type of data storage
CAtlArray	Implements an array.
CAtlList	Implements a list.
CAtlMap	Implements a mapping structure, whereby data can be referenced by key or value.
CRBMap	Implements a mapping structure using the Red-Black algorithm.
CRBMultiMap	Implements a Red-Black multimapping structure.

These classes will trap many programming errors when used in debug builds, but for sake of performance, these checks will not be performed in retail builds.

Specialized Collection Classes

More specialized collection classes are also provided for managing memory pointers and interface pointers:

Class	Purpose
CAutoPtrArray	Provides methods useful when constructing an array of smart pointers.
CAutoPtrList	Provides methods useful when constructing a list of smart pointers.
CComUnkArray	Stores **IUnknown** pointers and is designed to be used as a parameter to the IConnectionPointImpl template c lass.
CHeapPtrList	Provides methods useful when constructing a list of heap pointers.
CInterfaceArray	Provides methods useful when constructing an array of COM interface pointers.

CInterfaceList	Provides methods useful when constructing a list of COM interface pointers.

Choosing a Collection Class

Each of the available collection classes offers different performance characteristics, as shown in the table below.

- Columns 2 and 3 describe each class's ordering and access characteristics. In the table, the term "ordered" means that the order in which items are inserted and deleted determines their order in the collection; it does not mean the items are sorted on their contents. The term "indexed" means that the items in the collection can be retrieved by an integer index, much like items in a typical array.
- Columns 4 and 5 describe each class's performance. In applications that require many insertions into the collection, insertion speed might be especially important; for other applications, lookup speed may be more important.
- Column 6 describes whether each shape allows duplicate elements.
- The performance of a given collection class operation is expressed in terms of the relationship between the time required to complete the operation and the number of elements in the collection. An operation taking an amount of time that increases linearly as the number of elements increases is described as an O(n) algorithm. By contrast, an operation taking a period of time that increases less and less as the number of elements increases is described as an O(log n) algorithm. Therefore, in terms of performance, O(log n) algorithms outperform O(n) algorithms more and more as the number of elements increases.

Collection Shape Features

Shape	Ordered?	Indexed?	Insert an element	Search for specified element	Duplicate elements?
List	Yes	No	Fast (constant time)	Slow O(n)	Yes
Array	Yes	By int (constant time)	Slow O(n) except if inserting at end, in which case constant time	Slow O(n)	Yes
Map	No	By key (constant time)	Fast (constant time)	Fast (constant time)	No (keys) Yes (values)
Red-Black Map	Yes (by key)	By key O(log n)	Fast O(log n)	Fast O(log n)	No
Red-Black Multimap	Yes (by key)	By key O(log n) (multiple values per key)	Fast O(log n)	Fast O(log n)	Yes (multiple values per key)

Using CTraits Objects

As the ATL collection classes can be used to store a wide range of user-defined data types, it can be useful to be able to override important functions such as comparisons. This is achieved using the CTraits classes.

CTraits classes are similar to, but more flexible than, the MFC collection class helper functions; see Collection Class Helpers for more information.

When constructing your collection class, you have the option of specifying a CTraits class. This class will contain the code that will perform operations such as comparisons when called by the other methods that make up the collection class. For example, if your list object contains your own user-defined structures, you may want to redefine the equality test to only compare certain member variables. In this way, the list object's Find method will operate in a more useful manner.

Example

```
// alt_collection_classes.cpp
// Collection class / traits class example.
// This program demonstrates using a CTraits class
// to create a new comparison operator.
#include <string.h>
#include <atlsimpstr.h>
#include <atlstr.h>
```

```
#include "atlcoll.h"

#define MAX_STRING 80

// Define our own data type to store in the list.

struct MyData
{
    int ID;
    TCHAR name[MAX_STRING];
    TCHAR address[MAX_STRING];
};

// Define our own traits class, making use of the
// existing traits and overriding only the comparison
// we need.

class MyTraits : public CElementTraits< MyData >
{
public:
    // Override the comparison to only compare
    // the ID value.

    static bool CompareElements( const MyData& element1, const MyData& element2)
    {
        if ( element1.ID == element2.ID )
            return true;
        else
            return false;
    };
};

// The main function

int _tmain(int argc, _TCHAR* argv[])
{
    // Declare the array, with our data type and traits class

    CAtlList < MyData, MyTraits > MyList;

    // Create some variables of our data type

    MyData add_item, search_item;

    // Add some elements to the list.

    add_item.ID = 1;
    wsprintf( add_item.name, "Adam" );
    wsprintf( add_item.address, "One Garden Way" );

    MyList.AddHead( add_item );

    add_item.ID = 2;
    wsprintf( add_item.name, "Eve" );
    wsprintf( add_item.address, "One Garden Way" );

    MyList.AddHead( add_item );

    add_item.ID = 3;
    wsprintf( add_item.name, "Cain" );
    wsprintf( add_item.address, "Two Garden Way" );

    MyList.AddHead( add_item );

    // Create an element which will be used
    // to search the list for a match.

    search_item.ID = 2;
```

```
     wsprintf( search_item.name, "Don't care" );
     wsprintf( search_item.address, "Don't care" );

    // Perform a comparison by searching for a match
    // between any element in the list, and our
    // search item. This operation will use the
    // (overridden) comparison operator and will
    // find a match when the IDs are the same.

     POSITION i;

     i = MyList.Find( search_item );

     if ( i != NULL )
         printf_s("Item found!\n");
     else
         printf("Item not found.\n");

    return 0;
}
```

Comments

For a list of the CTraits classes, see Collection Classes.

The following diagram shows the class hierarchy for the CTraits classes.

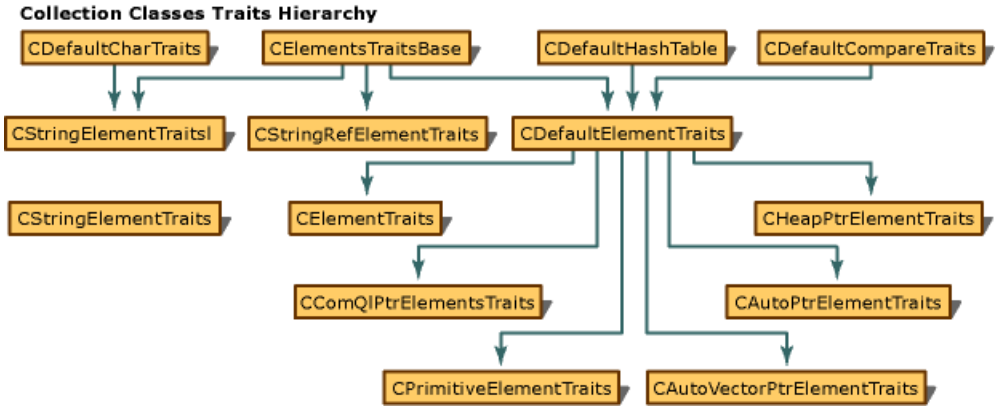

Collection Classes Samples

The following samples demonstrate the collection classes:

- MMXSwarm Sample
- DynamicConsumer Sample
- UpdatePV Sample
- Marquee Sample

See Also

Concepts

Collection Classes

Other Resources

ATL Concepts

The ATL Registry Component (Registrar)

The ATL Registrar provides optimized access to the system registry through a custom interface. The Registrar is free-threaded and allows static linking of code for C++ clients.

Note
The source code for the ATL Registrar can be found in atlmfc\include\atliface.h.

In This Section

Creating Registrar Scripts

A guide to creating registrar scripts. Includes topics on BNF syntax, parse trees, registry scripting examples, using replaceable parameters, and invoking scripts.

Setting Up a Static Link to the Registrar Code (C++ only)

Lists the steps to set up static linking to the Registrar.

Related Sections

ATL

Provides links to conceptual topics on how to program using the Active Template Library.

Creating Registrar Scripts

A registrar script provides data-driven, rather than API-driven, access to the system registry. Data-driven access is typically more efficient since it takes only one or two lines in a script to add a key to the registry.

The ATL Control Wizard automatically generates a registrar script for your COM server. You can find this script in the .rgs file associated with your object.

The ATL Registrar's Script Engine processes your registrar script at run time. ATL automatically invokes the Script Engine during server setup.

This article covers the following topics related to the registrar scripts:

- Understanding Backus Nauer Form (BNF) Syntax
- Understanding Parse Trees
- Registry Scripting Examples
- Using Replaceable Parameters (The Registrar's Preprocessor)
- Invoking Scripts

See Also

Other Resources

The ATL Registry Component (Registrar)

Understanding Backus Nauer Form (BNF) Syntax

The scripts used by the ATL Registrar are described in this topic using BNF syntax, which uses the notation shown in the following table.

Convention/symbol	Meaning
`::=`	Equivalent
`|`	OR
`X+`	One or more `X`s.
`[X]`	`X` is optional. Optional delimiters are denoted by `[]`.
Any **bold** text	A string literal.
Any *italicized* text	How to construct the string literal.

As indicated in the preceding table, registrar scripts use string literals. These values are actual text that must appear in your script. The following table describes the string literals used in an ATL Registrar script.

String literal	Action
ForceRemove	Completely removes the next key (if it exists) and then re-creates it.
NoRemove	Does not remove the next key during Unregister.
val	Specifies that `<Key Name>` is actually a named value.
Delete	Deletes the next key during Register.
s	Specifies that the next value is a string (**REG_SZ**).
d	Specifies that the next value is a **DWORD** (**REG_DWORD**).
m	Specifies that the next value is a multistring (**REG_MULTI_SZ**).
b	Specifies that the next value is a binary value (**REG_BINARY**).

BNF Syntax Examples

Here are a few syntax examples to help you understand how the notation and string literals work in an ATL Registrar script.

Syntax Example 1

```
<registry expression> ::= <Add Key>
```

specifies that `registry expression` is equivalent to `Add Key`.

Syntax Example 2

```
<registry expression> ::= <Add Key> | <Delete Key>
```

specifies that `registry expression` is equivalent to either `Add Key` or `Delete Key`.

Syntax Example 3

```
<Key Name> ::= '<AlphaNumeric>+'
```

specifies that `Key Name` is equivalent to one or more `AlphaNumerics`.

Syntax Example 4

```
<Add Key> ::= [ForceRemove | NoRemove | val]<Key Name>
```

specifies that `Add Key` is equivalent to `Key Name`, and that the string literals, `ForceRemove`, `NoRemove`, and `val`, are optional.

Syntax Example 5

```
<AlphaNumeric> ::= any character not NULL, that is, ASCII 0
```

specifies that `AlphaNumeric` is equivalent to any non-NULL character.

Syntax Example 6

```
val 'testmulti' = m 'String 1\0String 2\0'
```

specifies that the key name `testmulti` is a multistring value composed of `String 1` and `String 2`.

Syntax Example 7

```
val 'testhex' = d '&H55'
```

specifies that the key name `testhex` is a **DWORD** value set to hexadecimal 55 (decimal 85). Note this format adheres to the **&H** notation as found in the Visual Basic specification. For more information on this format, refer to Hex Function (Visual Basic).

See Also

Reference

Creating Registrar Scripts

Understanding Parse Trees

You can define one or more parse trees in your registrar script, where each parse tree has the following form:

```
<root key>{<registry expression>}+
```

where:

```
<root key> ::= HKEY_CLASSES_ROOT | HKEY_CURRENT_USER |
               HKEY_LOCAL_MACHINE | HKEY_USERS |
               HKEY_PERFORMANCE_DATA | HKEY_DYN_DATA |
               HKEY_CURRENT_CONFIG | HKCR | HKCU |
               HKLM | HKU | HKPD | HKDD | HKCC
<registry expression> ::= <Add Key> | <Delete Key>
<Add Key> ::= [ForceRemove | NoRemove | val]<Key Name>
              [<Key Value>][{< Add Key>}]
<Delete Key> ::= Delete<Key Name>
<Key Name> ::= '<AlphaNumeric>+'
<AlphaNumeric> ::= any character not NULL, i.e. ASCII 0
<Key Value> ::== <Key Type><Key Name>
<Key Type> ::= s | d
<Key Value> ::= '<AlphaNumeric>'
```

Note

`HKEY_CLASSES_ROOT` and `HKCR` are equivalent; `HKEY_CURRENT_USER` and `HKCU` are equivalent; and so on.

A parse tree can add multiple keys and subkeys to the <root key>. In doing so, it keeps a subkey's handle open until the parser has completed parsing all of its subkeys. This approach is more efficient than operating on a single key at a time, as seen in the following example:

```
HKEY_CLASSES_ROOT
{
   'MyVeryOwnKey'
   {
      'HasASubKey'
      {
         'PrettyCool?'
      }
   }
}
```

Here, the Registrar initially opens (creates) `HKEY_CLASSES_ROOT\MyVeryOwnKey`. It then sees that `MyVeryOwnKey` has a subkey. Rather than close the key to `MyVeryOwnKey`, the Registrar retains the handle and opens (creates) `HasASubKey` using this parent handle. (The system registry can be slower when no parent handle is open.) Thus, opening `HKEY_CLASSES_ROOT\MyVeryOwnKey` and then opening `HasASubKey` with `MyVeryOwnKey` as the parent is faster than opening `MyVeryOwnKey`, closing `MyVeryOwnKey`, and then opening `MyVeryOwnKey\HasASubKey`.

See Also

Reference

Creating Registrar Scripts

Registry Scripting Examples

The scripting examples in this topic demonstrate how to add a key to the system registry, register the Registrar COM server, and specify multiple parse trees.

Add a Key to HKEY_CURRENT_USER

The following parse tree illustrates a simple script that adds a single key to the system registry. In particular, the script adds the key, `MyVeryOwnKey`, to `HKEY_CURRENT_USER`. It also assigns the default string value of `HowGoesIt?` to the new key:

```
HKEY_CURRENT_USER
{
   'MyVeryOwnKey' = s 'HowGoesIt?'
}
```

This script can easily be extended to define multiple subkeys as follows:

```
HKCU
{
   'MyVeryOwnKey' = s 'HowGoesIt?'
   {
      'HasASubkey'
      {
         'PrettyCool?' = d '55'
         val 'ANameValue' = s 'WithANamedValue'
      }
   }
}
```

Now, the script adds a subkey, `HasASubkey`, to `MyVeryOwnKey`. To this subkey, it adds both the `PrettyCool?` subkey (with a default **DWORD** value of 55) and the `ANameValue` named value (with a string value of `WithANamedValue`).

Register the Registrar COM Server

The following script registers the Registrar COM server itself.

```
HKCR
{
   ATL.Registrar = s 'ATL Registrar Class'
   {
      CLSID = s '{44EC053A-400F-11D0-9DCD-00A0C90391D3}'
   }
   NoRemove CLSID
   {
      ForceRemove {44EC053A-400F-11D0-9DCD-00A0C90391D3} =
                   s 'ATL Registrar Class'
      {
         ProgID = s 'ATL.Registrar'
         InprocServer32 = s '%MODULE%'
         {
            val ThreadingModel = s 'Apartment'
         }
      }
   }
}
```

At run time, this parse tree adds the `ATL.Registrar` key to `HKEY_CLASSES_ROOT`. To this new key, it then:

- Specifies `ATL Registrar Class` as the key's default string value.
- Adds `CLSID` as a subkey.
- Specifies `{44EC053A-400F-11D0-9DCD-00A0C90391D3}` for `CLSID`. (This value is the Registrar's CLSID for use with

CoCreateInstance.)

Since `CLSID` is shared, it should not be removed in Unregister mode. The statement, `NoRemove CLSID`, does this by indicating that `CLSID` should be opened in Register mode and ignored in Unregister mode.

The `ForceRemove` statement provides a housekeeping function by removing a key and all of its subkeys before re-creating the key. This can be useful if the names of the subkeys have changed. In this scripting example, `ForceRemove` checks to see if `{44EC053A-400F-11D0-9DCD-00A0C90391D3}` already exists. If it does, `ForceRemove`:

- Recursively deletes `{44EC053A-400F-11D0-9DCD-00A0C90391D3}` and all of its subkeys.
- Re-creates `{44EC053A-400F-11D0-9DCD-00A0C90391D3}`.
- Adds `ATL Registrar Class` as the default string value for `{44EC053A-400F-11D0-9DCD-00A0C90391D3}`.

The parse tree now adds two new subkeys to `{44EC053A-400F-11D0-9DCD-00A0C90391D3}`. The first key, `ProgID`, gets a default string value that is the ProgID. The second key, `InprocServer32`, gets a default string value, `%MODULE%`, that is a preprocessor value explained in the section, Using Replaceable Parameters (The Registrar's Preprocessor), of this article. `InprocServer32` also gets a named value, `ThreadingModel`, with a string value of `Apartment`.

Specify Multiple Parse Trees

To specify more than one parse tree in a script, simply place one tree at the end of another. For example, the following script adds the key, `MyVeryOwnKey`, to the parse trees for both `HKEY_CLASSES_ROOT` and `HKEY_CURRENT_USER`:

```
HKCR
{
   'MyVeryOwnKey' = s 'HowGoesIt?'
}
HKEY_CURRENT_USER
{
   'MyVeryOwnKey' = s 'HowGoesIt?'
}
```

Note

In a Registrar script, 4K is the maximum token size. (A token is any recognizable element in the syntax.) In the previous scripting example, `HKCR`, `HKEY_CURRENT_USER`, `'MyVeryOwnKey'`, and `'HowGoesIt?'` are all tokens.

See Also

Reference

Creating Registrar Scripts

Using Replaceable Parameters (The Registrar's Preprocessor)

Replaceable parameters allow a Registrar's client to specify run-time data. To do this, the Registrar maintains a replacement map into which it enters the values associated with the replaceable parameters in your script. The Registrar makes these entries at run time.

Using %MODULE%

The ATL Control Wizard automatically generates a script that uses `%MODULE%`. ATL uses this replaceable parameter for the actual location of your server's DLL or EXE.

Concatenating Run-Time Data with Script Data

Another use of the preprocessor is to concatenate run-time data with script data. For example, suppose an entry is needed that contains a full path to a module with the string "`, 1`" appended at the end. First, define the following expansion:

```
'MySampleKey' = s '%MODULE%, 1'
```

Then, before calling one of the script processing methods listed in Invoking Scripts, add a replacement to the map:

```
TCHAR szModule[_MAX_PATH]
GetModuleFileName(pM->m_hInst, szModule, _MAX_PATH);
p->AddReplacement(OLESTR("Module"), T2OLE(szModule));
```

During the parsing of the script, the Registrar expands `'%MODULE%, 1'` to `c:\mycode\mydll.dll, 1`.

Note

In a Registrar script, 4K is the maximum token size. (A token is any recognizable element in the syntax.) This includes tokens t hat were created or expanded by the preprocessor.

Note

To substitute replacement values at run time, remove the call in the script to the DECLARE_REGISTRY_RESOURCE or DECLARE_REGISTRY_RESOURCEID macro. Instead, replace it with your own **UpdateRegistry** method that calls CAtlModule::UpdateRegistryFromResourceD or CAtlModule::UpdateRegistryFromResourceS, and pass your array of **_ATL_RE GMAP_ENTRY** structures. Your array of **_ATL_REGMAP_ENTRY** must have at least one entry that is set to {**NULL**,**NULL**}, an d this entry should always be the last entry. Otherwise, an access violation error will be generated when **UpdateRegistryFro mResource** is called.

Note

When building a project that outputs an executable, ATL automatically adds quotation marks around the path name created a t run time with the **%MODULE%** registrar script parameter. If you do not want the path name to include the quotation marks , use the new **%MODULE_RAW%** parameter instead.

When building a project that outputs a DLL, ATL will not add quotation marks to the path name if **%MODULE%** or **%MODU LE_RAW%** is used.

See Also

Reference

Creating Registrar Scripts

Invoking Scripts

Using Replaceable Parameters (The Registrar's Preprocessor) discusses replacement maps and mentions the Registrar method **AddReplacement**. The Registrar has eight other methods specific to scripting, and all are described in the following table.

Method	Syntax/Description
ResourceRegister	**HRESULT ResourceRegister(LPCOLESTR** *resFileName* **, UINT** *nID* **, LPCOLESTR** *szType* **);** Registers the script contained in a module's resource. *resFileName* indicates the UNC path to the module itself. *nID* and *szType* contain the resource's ID and type, respectively.
ResourceUnregister	**HRESULT ResourceUnregister(LPCOLESTR** *resFileName* **, UINT** *nID* **, LPCOLESTR** *szType* **);** Unregisters the script contained in a module's resource. *resFileName* indicates the UNC path to the module itself. *nID* and *szType* contain the resource's ID and type, respectively.
ResourceRegisterSz	**HRESULT ResourceRegisterSz(LPCOLESTR** *resFileName* **, LPCOLESTR** *szID* **, LPCOLESTR** *szType* **);** Registers the script contained in a module's resource. *resFileName* indicates the UNC path to the module itself. *szID* and *szType* contain the resource's string identifier and type, respectively.
ResourceUnregisterSz	**HRESULT ResourceUnregisterSz(LPCOLESTR** *resFileName* **, LPCOLESTR** *szID* **, LPCOLESTR** *szType* **);** Unregisters the script contained in a module's resource. *resFileName* indicates the UNC path to the module itself. *szID* and *szType* contain the resource's string identifier and type, respectively.
FileRegister	**HRESULT FileRegister(LPCOLESTR** *fileName* **);** Registers the script in a file. *fileName* is a UNC path to a file that contains (or is) a resource script.
FileUnregister	**HRESULT FileUnregister(LPCOLESTR** *fileName* **);** Unregisters the script in a file. *fileName* is a UNC path to a file that contains (or is) a resource script.
StringRegister	**HRESULT StringRegister(LPCOLESTR** *data* **);** Registers the script in a string. *data* contains the script itself.
StringUnregister	**HRESULT StringUnregister(LPCOLESTR** *data* **);** Unregisters the script in a string. *data* contains the script itself.

ResourceRegisterSz and **ResourceUnregisterSz**, are similar to **ResourceRegister** and **ResourceUnregister**, but allow you to specify a string identifier.

The methods **FileRegister** and **FileUnregister** are useful if you do not want the script in a resource or if you want the script in its own file. The methods **StringRegister** and **StringUnregister** allow the .rgs file to be stored in a dynamically allocated string.

See Also

Reference

Creating Registrar Scripts

Setting Up a Static Link to the Registrar Code (C++ Only)

C++ clients can create a static link to the Registrar's code. Static linking of the Registrar's parser adds approximately 5K to a release build.

The simplest way to set up static linking assumes you have specified DECLARE_REGISTRY_RESOURCEID in your object's declaration. (This is the default specification used by the ATL.)

To create a static link using DECLARE_REGISTRY_RESOURCEID

1. Specify /D_**ATL_STATIC_REGISTRY** instead of /D_**ATL_DLL**.
2. Recompile.

If the project has been created using the ATL Project Wizard, you can set the Property Page option **Minimize CRT Use in ATL** to **Yes.** This Property Page setting adds **_ATL_MIN_CRT** to the compiler command line. See General Property Page (Project) for details on accessing and changing General project settings using the Property Page dialog box.

See Also

Other Resources

The ATL Registry Component (Registrar)

Programming with ATL and C Run-Time Code

This section discusses the benefits of using the C Run-Time Library (CRT) with either static or dynamic linking, or alternatively using the **_ATL_MIN_CRT** macro to reduce your dependency on CRT code.

In This Section

Benefits and Tradeoffs of Using _ATL_MIN_CRT

Summarizes the benefits and tradeoffs involved in linking statically to the CRT, linking dynamically, or avoiding the CRT DLL altogether.

Linking to the CRT in Your ATL Project

Discusses the project settings and linker options for linking to the CRT; also provides details on how linking to the CRT affects your program image.

Using _ATL_MIN_CRT in Your ATL Project

Discusses the project settings for using **_ATL_MIN_CRT** and how this affects your code.

Recognizing Functions that Rely on CRT Startup Code

Lists functions, compiler options, and libraries that require the CRT startup code.

Identifying CRT Startup Code Dependencies by Building

Shows how to identify your CRT dependencies by using various build options and examining the output.

Exception Handling and _ATL_MIN_CRT

Describes how using **_ATL_MIN_CRT** affects exception handling in ATL projects.

Related Sections

ATL

Provides links to conceptual topics on how to program using the Active Template Library.

Run-Time Library Behavior

Provides details on how the CRT startup code works.

C Run-Time Libraries

Discusses the various .lib files that comprise the C run-time libraries and lists their associated compiler options and preprocessor directives.

See Also

Other Resources

ATL Concepts

Benefits and Tradeoffs of Using _ATL_MIN_CRT

Your project can link with the CRT either dynamically or statically, or avoid the CRT DLL altogether. The table below outlines the benefits and tradeoffs involved in choosing which method to use.

Method	Benefit	Tradeoff
Statically linking to the CRT (**Runtime Library** set to **Single-threaded**)	The CRT DLL is not required on the system where the image will run.	About 25K of startup code is added to your image, substantially increasing its size.
Dynamically linking to the CRT (**Runtime Library** set to **Multi-threaded**)	Your image does not require the CRT startup code, so it is much smaller.	The CRT DLL must be on the system running the image.
Using **_ATL_MIN_CRT** (**Minimize CRT Use in ATL** set to **Yes)**	Keeps the image size small, and does not require that the CRT DLL be on the system.	Provides only a limited set of CRT functions. You must monitor your output to ensure your image does not depend on CRT startup code.

The topic Linking to the CRT in Your ATL Project discusses how to select the manner in which to link to the CRT.

See Also

Reference

C Run-Time Libraries

Concepts

Programming with ATL and C Run-Time Code

Run-Time Library Behavior

Linking to the CRT in Your ATL Project

The C Run-Time Libraries (CRT) provide many useful functions that can make programming much easier during ATL development. By default, ATL projects link to the CRT library. You can see the advantages and disadvantages of linking to the CRT in Benefits and Tradeoffs of Using _ATL_MIN_CRT.

Project Settings

There are two settings for the **Minimize CRT Use in ATL** project property. See General Property Page (Project) for details on accessing and changing General project settings using the Property Page dialog box.

- **Yes** If you don't want your project to use the CRT, or even use the CRT startup code, set the Property Page option **Minimize CRT Use in ATL** to **Yes.** This Property Page setting adds **_ATL_MIN_CRT** to the compiler command line
- **No** If your project needs to use CRT startup code, set the **Minimize CRT Use in ATL** option to **No** and link to the CRT either statically or dynamically.

For more information about using **Minimize CRT Use in ATL**, see Using _ATL_MIN_CRT in Your ATL Project.

Affects of Linking to the CRT on Your Program Image

If you statically link to the CRT, code from the CRT is placed in your executable image and you do not need to have the CRT DLL present on a system to run your image. If you dynamically link to the CRT, references to the code in the CRT DLL are placed in your image, but not the code itself. In order for your image to run on a given system, the CRT DLL must be present on that system. Even when you dynamically link to the CRT, you may find that some code can be statically linked (for example, **DllMainCRTStartup**).

When you link your image, you either explicitly or implicitly specify an entry point that the operating system will call into after loading the image. For a DLL, the default entry point is **DllMainCRTStartup**. For an EXE, it is **WinMainCRTStartup**. You can override the default with the /ENTRY linker option. The CRT provides an implementation for **DllMainCRTStartup**, **WinMainCRTStartup**, and **wWinMainCRTStartup** (the Unicode entry point for an EXE). These CRT-provided entry points call constructors on global objects and initialize other data structures that are used by some CRT functions. This startup code adds about 25K to your image if it is linked statically. If it is linked dynamically, most of the code is in the DLL, so your image size stays small.

For more information, see the linker topic /ENTRY (Entry-Point Symbol).

Optimization Options

Using the linker option /OPT:NOWIN98 can further reduce a default ATL control by 10K, at the expense of increased loading time on Windows 98 systems. For more information on linking options, see /OPT (Optimizations).

See Also

Concepts

Programming with ATL and C Run-Time Code
Run-Time Library Behavior

Using _ATL_MIN_CRT in Your ATL Project

Two advantages of ATL are that it allows you to:

- Minimize your image size
- Minimize your reliance on run-time DLLs

However, you might want to take advantage of some functions provided by the CRT. Because statically linking increases your image size, and dynamically linking ties you to the run-time DLL, using the CRT may deprive you of these key goals for using ATL.

To help solve this dilemma, ATL provides a solution: the **_ATL_MIN_CRT** macro. The **_ATL_MIN_CRT** macro, activated by setting the General Property option **Minimize CRT Use in ATL** to **Yes**, provides alternative implementations for many of the common CRT functions that would otherwise require the CRT startup code.

Note
Console applications do not support **_ATL_MIN_CRT**. If you use **_ATL_MIN_CRT** in an ATL console application, you will receive link error "LNK2005: __osplatform already defined in atlmincrt.lib(atlinit.obj)."

Remember that using the **_ATL_MIN_CRT** macro does not guarantee that no functions from the CRT will be required.

If you use a function that requires the CRT startup code to operate properly, you will get the following linker error:

```
LIBCMT.LIB(crt0.obj) : error LNK2001: unresolved external symbol _main
```

Providing your own implementation of **_main** does not solve this problem: you must either remove reliance on the functions that require the CRT startup code, or you must either statically link the startup code in your image or dynamically link to the CRT.

When **_ATL_MIN_CRT** is used without the **/GS-** compiler option, ATL headers will require the static CRT library. For more information, see /GS (Buffer Security Check).

For more information on linking to the CRT, see Linking to the CRT in Your ATL Project.

To identify the CRT startup code, see the following topics:

- Recognizing Functions that Rely on CRT Startup Code
- Identifying CRT Startup Code Dependencies by Building

See Also

Concepts

Programming with ATL and C Run-Time Code

Recognizing Functions that Rely on CRT Startup Code

If you want your project to be free of dependencies on the CRT startup code, you must be careful with your choice of functions. Functions that rely on the CRT startup code are listed below. It is also worth noting that not all CRT functions require the CRT startup code, and these are listed as well. Finally, it is often possible to substitute Win32 equivalents for code that requires the CRT startup code: this is discussed in the final section.

See Identifying CRT Startup Code Dependencies by Building regarding techniques for discovering any remaining CRT-dependent code in your project.

Functions that Rely on CRT Startup Code

Below are some functions, compiler options, and libraries that require the CRT startup code:

- Memory allocation functions
- Floating-point functions
- Compiler option /GX

  If you enable C++ exceptions by using the compiler option /GX, you must have the CRT startup code. Because MFC requires code to be compiled with /GX, you cannot use **_ATL_MIN_CRT** in conjunction with MFC.

  If you use **_ATL_MIN_CRT** and need to handle exceptions, you can use structured exception handling (SEH). You will need to cannibalize some of the CRT static library code that ships with Visual C++ and replace some code with system calls. See C++ Exception Handling for more information.

CRT Functions that Do Not Rely on CRT Startup Code

Some of the CRT functions that can be used without requiring the CRT startup code include:

- CStringT Class

  Although by default, ATL's **CStringT** will use the CRT instead of the Win32 string functions, it is possible to modify this behavior by #defining either **_ATL_MIN_CRT** or **_ATL_CSTRING_NO_CRT**.

  The Win32 string functions have several functionality limitations compared with the CRT especially for Unicode strings on Windows 95, Windows 98, and Windows Me, but choosing not to link in the CRT may benefit developers who are looking for the smallest size for their executables. **CStringT**'s full functionality is enabled by default. Programmers can explicitly request not to use the CRT if they are willing to give up some functionality in exchange for some improvement in executable size.

- the **mem*** functions
- wcslen
- wcscmp

  Most string manipulation functions require CRT startup code. The string comparison function **wcscmp** does not require the CRT startup code, because Unicode string comparisons are performed through memcmp. Other CRT string comparisons require the startup code because the CRT initializes some tables used for comparing. Likewise, **wcsicmp** requires the startup code, because the CRT uses tables to change case.

Win32 Equivalents to Using CRT Startup Code

Many functions that require CRT startup code have Win32 equivalents, which you can substitute in your code. For example, you can use lstrcmp instead of **strcmp**. For a list of the substitute functions, see KB article Q99456, "Win32 Equivalents for C Run-Time Functions."

See Also

Reference

C Run-Time Libraries

Run-Time Routines by Category

Concepts

Programming with ATL and C Run-Time Code

Identifying CRT Startup Code Dependencies by Building

One way to tell if you have used functions that require the CRT startup code is to build the project and check for the linker error, described in Using _ATL_MIN_CRT in Your ATL Project. You can further identify the functions by specifying the /NODEFAULTLIB:libcmt.lib linker option. You will get several unresolved externals. This list is everything that you are using from the CRT. Examine this list for functions that may be trying to pull in the startup code.

An alternative method for identifying functions that need startup code is to specify /VERBOSE. From the build output window, search for **_main** to determine what functions need the CRT startup code.

After you analyze these results, remove some functions, and replace as many CRT functions as possible with their Win32 equivalents, you still may decide that you need the CRT startup code. In this case, you set the General Property option **Minimize CRT Use in ATL** to **No**, and therefore remove the **_ATL_MIN_CRT** define from the project settings. At this point, you must decide whether to statically link to the CRT and increase your image size, or dynamically link to the CRT and supply the CRT DLL.

See Also

Reference

C Run-Time Libraries

Concepts

Programming with ATL and C Run-Time Code

Recognizing Functions that Rely on CRT Startup Code

Exception Handling and _ATL_MIN_CRT

By default, ATL projects will have exception handling enabled, unless at least one of the following is true:

- **_ATL_MIN_CRT** is defined.
- **_ATL_NO_EXCEPTIONS** is defined.
- The project is not compiled with /GX (Enable Exception Handling).

See Also

Reference

CStringT Class

Concepts

Programming with ATL and C Run-Time Code

Programming with CComBSTR

The ATL class CComBSTR provides a wrapper around the **BSTR** data type. While **CComBSTR** is a useful tool, there are several situations that require caution.

- Conversion Issues
- Scope Issues
- Explicitly Freeing the CComBSTR Object
- Using CComBSTR Objects in Loops
- Memory Leak Issues

Conversion Issues

Although several **CComBSTR** methods will automatically convert an ANSI string argument into Unicode, the methods will always return Unicode format strings. To convert the output string back to ANSI, use an ATL conversion class. For more information on the ATL conversion classes, see ATL and MFC String Conversion Macros.

Example

```
// Declare a CComBSTR object. Although the argument is ANSI,
// the constructor converts it into UNICODE.
CComBSTR bstrMyString( "Hello World" );
// Convert the string into an ANSI string
CW2CT szMyString( bstrMyString );
// Display the ANSI string
MessageBox( NULL, szMyString, _T("String Test"), MB_OK );
```

If you are using a string literal to modify a **CComBSTR** object, use wide character strings. This will reduce unnecessary conversions.

Example

```
// The following converts the ANSI string to Unicode
CComBSTR bstr("Test");
// The following uses a Unicode string at compile time
CComBSTR bstr(L"Test");
```

Scope Issues

As with any well-behaved class, **CComBSTR** will free its resources when it goes out of scope. If a function returns a pointer to the **CComBSTR** string, this can cause problems, as the pointer will reference memory that has already been freed. In these cases, use the **Copy** method, as shown below.

Example

```
// The wrong way to do it
BSTR * MyBadFunction()
{
   // Define a pointer to a BSTR
   BSTR * bstrStringPtr;
   // Create the CComBSTR object
   CComBSTR bstrString("Hello World");
   // Convert the string to uppercase
   bstrString.ToUpper();
   // Assign the pointer
   * bstrStringPtr = bstrString;
   // Return the pointer. ** Bad thing to do **
   return bstrStringPtr;
}
// The correct way to do it
HRESULT MyGoodFunction(/*[out]*/ BSTR* bstrStringPtr)
{
```

```
    // Create the CComBSTR object
    CComBSTR bstrString("Hello World");
    // Convert the string to uppercase
    bstrString.ToUpper();
    // Return a copy of the string.
    return bstrString.CopyTo(bstrStringPtr);
}
```

Explicitly Freeing the CComBSTR Object

It is possible to explicitly free the string contained in the **CComBSTR** object before the object goes out scope. If the string is freed, the **CComBSTR** object is invalid.

Example

```
// Declare a CComBSTR object
CComBSTR bstrMyString( "Hello World" );
// Free the string explicitly
::SysFreeString(bstrMyString);
// The string will be freed a second time
// when the CComBSTR object goes out of scope,
// which is invalid.
```

Using CComBSTR Objects in Loops

As the **CComBSTR** class allocates a buffer to perform certain operations, such as the **+=** operator or **Append** method, it is not recommended that you perform string manipulation inside a tight loop. In these situations, **CStringT** provides better performance.

Example

```
// This is not an efficient way
// to use a CComBSTR object.
CComBSTR bstrMyString;
while (bstrMyString.Length()<1000)
   bstrMyString.Append(L"*");
```

Memory Leak Issues

Passing the address of an initialized **CComBSTR** to a function as an **[out]** parameter causes a memory leak.

In the example below, the string allocated to hold the string `"Initialized"` is leaked when the function `OutString` replaces the string.

```
CComBSTR bstrLeak(L"Initialized");
HRESULT hr = OutString(&bstrLeak);
```

To avoid the leak, call the **Empty** method on existing **CComBSTR** objects before passing the address as an **[out]** parameter.

Note that the same code would not cause a leak if the function's parameter was **[in, out]**.

See Also

Reference

CStringT Class
wstring

Other Resources

ATL Concepts
String Conversion Macros

ATL Class Overview

Classes in the Active Template Library (ATL) can be categorized as follows:

Class Factories	Memory Management
Class Information	MMC Snap-In
Collection	Object Safety
COM Modules	Persistence
Composite Controls	Properties and Property Pages
Connection Points	Registry Support
Control Containment	Running Objects
Controls: General Support	Security
Data Transfer	Service Provider Support
Data Types	Site Information
Debugging and Exception	String and Text
Dual Interfaces	Tear-Off Interfaces
Enumerators and Collections	Thread Pooling
Error Information	Threading Models and Critical Sections
File Handling	UI Support
Interface Pointers	Windows Support
IUnknown Implementation	Utility

For additional classes that can be used in ATL projects, see Shared Classes.

See Also

Other Resources

ATL Classes
ATL Reference
ATL Functions
ATL Global Variables
ATL Macros
ATL Structures
ATL Typedefs

Class Factories Classes

The following classes implement or support a class factory:

- CComClassFactory Provides a default class factory for object creation.
- CComClassFactory2 Controls object creation through a license.
- CComClassFactoryAutoThread Allows objects to be created in multiple thread-pooled apartments.
- CComClassFactorySingleton Creates a single object.
- CComCoClass Defines the class factory for the object.

See Also

Other Resources

ATL Class Overview
Aggregation and Class Factory Macros

Class Information Classes

The following class provides support for retrieving class information:

- IProvideClassInfo2Impl Provides access to type information. Retrieves the outgoing IID for the object's default event set.

See Also

Other Resources

ATL Class Overview

Collection Classes

The following classes provide support for arrays, lists, maps, and also traits methods for helping with comparisons and element access.

- CAtlArray This class implements an array object.
- CAtlList This class provides methods for creating and managing a list object.
- CAtlMap This class provides methods for creating and managing a map object.
- CAutoPtrArray This class provides methods useful when constructing an array of smart pointers.
- CAutoPtrElementTraits This class provides methods, static functions, and typedefs useful when creating collections of smart pointers.
- CAutoPtrList This class provides methods useful when constructing a list of smart pointers.
- CAutoVectorPtrElementTraits This class provides methods, static functions, and typedefs useful when creating collections of smart pointers using vector new and delete operators.
- CComQIPtrElementTraits This class provides methods, static functions, and typedefs useful when creating collections of COM interface pointers.
- CComSafeArray This class is a wrapper for the SAFEARRAY Data Type structure.
- CComSafeArrayBound This class is a wrapper for a SAFEARRAYBOUND structure.
- CComUnkArray This class stores **IUnknown** pointers and is designed to be used as a parameter to the IConnectionPointImpl template class.
- CDefaultCharTraits This class provides two static functions for converting characters between uppercase and lowercase.
- CDefaultCompareTraits This class provides default element comparison functions.
- CDefaultElementTraits This class provides default methods and functions for a collection class.
- CDefaultHashTraits This class provides a static function for calculating hash values.
- CElementTraits This class is used by collection classes to provide methods and functions for moving, copying, comparison, and hashing operations.
- CElementTraitsBase This class provides default copy and move methods for a collection class.
- CHeapPtrElementTraits This class provides methods, static functions, and typedefs useful when creating collections of heap pointers.
- CHeapPtrList This class provides methods useful when constructing a list of heap pointers.
- CInterfaceArray This class provides methods useful when constructing an array of COM interface pointers.
- CInterfaceList This class provides methods useful when constructing a list of COM interface pointers.
- CPrimitiveElementTraits This class provides default methods and functions for a collection class composed of primitive data types.
- CRBMap This class represents a mapping structure, using a Red-Black binary tree.
- CRBMultiMap This class represents a mapping structure that allows each key to be associated with more than one value, using a Red-Black binary tree.
- CRBTree This class provides methods for creating and utilizing a Red-Black tree.
- CSimpleArray This class provides methods for managing a simple array.
- CSimpleArrayEqualHelper This class is a helper for the CSimpleArray class.
- CSimpleArrayEqualHelperFalse This class is a helper for the CSimpleArray class.
- CSimpleMap This class provides support for a simple mapping array.

- CSimpleMapEqualHelper This class is a helper for the CSimpleMap class.
- CSimpleMapEqualHelperFalse This class is a helper for the CSimpleMap class.
- CStringElementTraits This class provides static functions used by collection classes storing **CString** objects.
- CStringElementTraitsI This class provides static functions related to strings stored in collection class objects. It is similar to CStringElementTraits, but performs case-insensitive comparisons.
- CStringRefElementTraits This class provides static functions related to strings stored in collection class objects. The string objects are dealt with as references.

Related Articles

ATL Collection Classes

See Also

Concepts

ATL Collection Classes

Other Resources

ATL Class Overview

COM Modules Classes

The following classes provide support for a COM module:

- CAtlBaseModule This class is instantiated in every ATL project.
- CAtlComModule This class implements a COM server module.
- CAtlModule This class provides methods used by several ATL module classes.
- CAtlModuleT This class implements an ATL module.
- CAtlExeModuleT This class represents the module for an application.
- CAtlServiceModuleT This class implements a service.
- CAtlWinModule This class provides support for ATL windowing components.
- CComModule This class implements a DLL or EXE module. Obsolete in ATL 7.0.
- CComAutoThreadModule This class implements an EXE module, with support for multiple thread-pooled apartments. Obsolete in ATL 7.0.

Related Articles

ATL Module Classes

See Also

Reference

ATL Module Classes

Other Resources

ATL Class Overview

Composite Controls Classes

The following class provides support for creating composite controls

- CComCompositeControl ActiveX controls derived from **CComCompositeControl** are hosted by a standard dialog box. These types of controls are called composite controls because they are able to host other controls (native Windows controls and ActiveX controls).

Related Articles

Composite Control Fundamentals

See Also

Other Resources

ATL Class Overview
Composite Control Macros
Composite Control Global Functions

Connection Points Classes

The following classes provide support for connection points:

- IConnectionPointContainerImpl Implements a connection point container.
- IConnectionPointImpl Implements a connection point.
- IPropertyNotifySinkCP Implements a connection point representing the IPropertyNotifySink interface.
- CComDynamicUnkArray Manages unlimited connections between a connection point and its sinks.
- CComUnkArray Manages a fixed number of connections between a connection point and its sinks.
- CFirePropNotifyEvent Notifies a client's sink that an object's property has changed or is about to change.
- IDispEventImpl Provides support for connection points for an ATL COM object. These connection points are mapped with an event sink map, which is provided by your COM object.
- IDispEventSimpleImpl Works in conjunction with the event sink map in your class to route events to the appropriate handler function.

Related Articles

Connection Points

Event Handling and ATL

See Also

Other Resources

ATL Class Overview
Connection Point Macros
Connection Point Global Functions

Control Containment Classes

The following classes provide containment support for hosting controls:

- CAxWindow Provides methods for manipulating a window that hosts an ActiveX control.
- CAxWindow2T Provides methods for manipulating a window that hosts an ActiveX control and also has support for hosting licensed ActiveX controls.
- IAxWinAmbientDispatch Call the methods on this interface to set the ambient properties available to a hosted control.
- IAxWinHostWindow Call the methods on this interface to create and/or attach a control to a host object, or to get an interface from a hosted control.

Related Articles

ATL Control Containment FAQ

See Also

Other Resources

ATL Class Overview

Controls: General Support Classes

The following classes provide general support for ATL controls:

- CComControl Consists of helper functions and data members that are essential to ATL controls.
- IOleControlImpl Provides methods necessary for controls.
- IOleObjectImpl Provides the principal methods through which a container communicates with a control. Manages the activation and deactivation of in-place controls.
- IQuickActivateImpl Combines initialization into a single call to help containers avoid delays when loading controls.
- IPointerInactiveImpl Provides minimal mouse interaction for an otherwise inactive control.

Sample Program

ATLFire

Related Articles

ATL Tutorial

See Also

Other Resources

ATL Class Overview

Data Transfer Classes

The following classes support various types of data transfer:

- IDataObjectImpl Supports Uniform Data Transfer by using standard formats to retrieve and set data. Handles data change notifications by managing connections to advise sinks.
- CBindStatusCallback Allows an asynchronous moniker to send and receive information about the asynchronous data transfer to and from your object.

See Also

Other Resources

ATL Class Overview

Data Types Classes

The following classes wrap C++ data types:

- CComBSTR Wraps the **BSTR** data type.
- CComVariant Wraps the **VARIANT** data type.
- CComCurrency Includes methods and operators for creating and managing a CURRENCY object.

See Also

Other Resources

ATL Class Overview

Debugging and Exceptions Classes

These classes provide support for exception handling and debugging.

- CAtlDebugInterfacesModule This class provides support for debugging interfaces.
- CAtlException This class defines an ATL exception.

See Also

Other Resources

ATL Class Overview
Debugging and Error Reporting Global Functions
Debugging and Error Reporting Macros

Dual Interfaces Classes

The following class provides support for dual interfaces:

- IDispatchImpl Implements the **IDispatch** portion of a dual interface. For more information, see Implementing the IDispatch Interface.

See Also

Other Resources

ATL Class Overview

Enumerators and Collections Classes

The following classes provide support for COM collections and enumerations:

- CComEnum Defines a COM enumerator object based on an array.
- CComEnumImpl Provides the implementation for a COM enumerator interface where the items being enumerated are stored in an array.
- CComEnumOnSTL Defines a COM enumerator object based on an STL collection.
- IEnumOnSTLImpl Provides the implementation for a COM enumerator interface where the items being enumerated are stored in an STL-compatible container.
- ICollectionOnSTLImpl Provides the implementation for the **Count**, **Item**, and **_NewEnum** properties of a collection interface.

Related Articles

ATL Collections and Enumerators

See Also

Other Resources

ATL Class Overview

Error Information Classes

The following class indicates how error information is handled:

- ISupportErrorInfoImpl Determines whether the object supports the IErrorInfo interface. **IErrorInfo** allows error information to be propagated back to the client.

See Also

Other Resources

ATL Class Overview

File Handling Classes

These classes provide methods for handling files, temporary files, and memory-mapped files.

- CAtlFile This class provides a thin wrapper around the Windows file-handling API.
- CAtlFileMapping This class represents a memory-mapped file, adding a cast operator to the methods of CAtlFileMappingBase.
- CAtlFileMappingBase This class represents a memory-mapped file.
- CAtlTemporaryFile This class provides methods for the creation and use of a temporary file.

See Also

Other Resources

ATL Class Overview

Interface Pointers Classes

The following classes manage a given interface pointer:

- CComPtr Performs automatic reference counting.
- CComQIPtr Similar to **CComPtr**, but also performs automatic querying of interfaces.
- CInterfaceArray Provides methods useful when constructing an array of COM interface pointers.
- CInterfaceList Provides methods useful when constructing a list of COM interface pointers.
- CComGITPtr Provides methods for dealing with interface pointers and the global interface table (GIT).

See Also

Other Resources

ATL Class Overview

IUnknown Implementation Classes

The following classes implement **IUnknown** and related methods:

- CComObjectRootEx Manages reference counting for both aggregated and nonaggregated objects. Allows you to specify a threading model.
- CComObjectRoot Manages reference counting for both aggregated and nonaggregated objects. Uses the default threading model of the server.
- CComAggObject Implements **IUnknown** for an aggregated object.
- CComObject Implements **IUnknown** for a nonaggregated object.
- CComPolyObject Implements **IUnknown** for aggregated and nonaggregated objects. Using **CComPolyObject** avoids having both **CComAggObject** and **CComObject** in your module. A single **CComPolyObject** object handles both aggregated and nonaggregated cases.
- CComObjectNoLock Implements **IUnknown** for a nonaggregated object, without modifying the module lock count.
- CComTearOffObject Implements **IUnknown** for a tear-off interface.
- CComCachedTearOffObject Implements **IUnknown** for a "cached" tear-off interface.
- CComContainedObject Implements **IUnknown** for the inner object of an aggregation or a tear-off interface.
- CComObjectGlobal Manages a reference count on the module to ensure your object won't be deleted.
- CComObjectStack Creates a temporary COM object, using a skeletal implementation of **IUnknown**.

Related Articles

Fundamentals of ATL COM Objects

See Also

Other Resources

ATL Class Overview
Aggregation and Class Factory Macros
COM Map Macros
COM Map Global Functions

Memory Management Classes

These classes provide support for heap pointers, smart pointers, and other memory allocation routines.

- CAutoPtr This class represents a smart pointer object.
- CAutoVectorPtr This class represents a smart pointer object using vector new and delete operators.
- CAutoPtrArray This class provides methods useful when constructing an array of smart pointers.
- CAutoPtrList This class provides methods useful when constructing a list of smart pointers.
- CAutoVectorPtr This class represents a smart pointer object using vector new and delete operators.
- CComAllocator This class provides methods for managing memory using COM memory routines.
- CComGITPtr This class provides methods for dealing with interface pointers and the global interface table (GIT).
- CComHeap This class implements IAtlMemMgr using the COM memory allocation functions.
- CComHeapPtr A smart pointer class for managing heap pointers.
- CComPtr A smart pointer class for managing COM interface pointers.
- CComPtrBase This class provides a basis for smart pointer classes using COM-based memory routines.
- CComQIPtr A smart pointer class for managing COM interface pointers.
- CCRTAllocator This class provides methods for managing memory using CRT memory routines.
- CCRTHeap This class implements IAtlMemMgr using the CRT heap functions.
- CGlobalHeap This class implements IAtlMemMgr using the Win32 global heap functions.
- CHandle This class provides methods for creating and using a handle object.
- CHeapPtr A smart pointer class for managing heap pointers.
- CHeapPtrBase This class forms the basis for several smart heap pointer classes.
- CHeapPtrList This class provides methods useful when constructing a list of heap pointers.
- CLocalHeap This class implements IAtlMemMgr using the Win32 local heap functions.
- CWin32Heap This class implements IAtlMemMgr using the Win32 heap allocation functions.
- IAtlMemMgr This class represents the interface to a memory manager.

See Also

Other Resources

ATL Class Overview

MMC Snap-In Classes

The following classes provide support for developing Microsoft Management Console (MMC) snap-in components:

- CSnapInItemImpl Implements a snap-in node object, such as adding menu items and toolbars, and forwarding commands for the snap-in node to the appropriate handler function.
- CSnapInPropertyPageImpl Implements a snap-in property page object.

See Also

Other Resources

ATL Class Overview
Snap-In Object Macros

Object Safety Classes

The following class provides support for object safety:

- IObjectSafetyImpl Allows an object to be marked as safe for initialization or safe for scripting.

Related Articles

ATL Tutorial

See Also

Other Resources

ATL Class Overview

Persistence Classes

The following classes implement object persistence:

- IPersistPropertyBagImpl Allows a client to load and save an object's properties to a property bag.
- IPersistStreamInitImpl Allows a client to load and save an object's persistent data to a stream.
- IPersistStorageImpl Allows a client to load and save an object's persistent data to a storage.

Related Articles

ATL Tutorial

See Also

Other Resources

ATL Class Overview
Property Map Macros

Properties and Property Pages Classes

The following classes support properties and property pages:

- CComDispatchDriver Retrieves or sets an object's properties through an **IDispatch** pointer. For more information, see Implementing the IDispatch Interface.
- CStockPropImpl Implements the stock properties supported by ATL.
- IPerPropertyBrowsingImpl Accesses the information in an object's property pages.
- IPersistPropertyBagImpl Stores an object's properties in a client-supplied property bag.
- IPropertyPageImpl Manages a particular property page within a property sheet.
- IPropertyPage2Impl Similar to **IPropertyPageImpl**, but also allows a client to select a specific property in a property page.
- ISpecifyPropertyPagesImpl Obtains the CLSIDs for the property pages supported by an object.

Related Articles

ATL Tutorial

ATL COM Property Pages

See Also

Other Resources

ATL Class Overview
Property Map Macros

Registry Support Classes

The following class provides registry support:

- CRegKey Contains methods for manipulating values in the system registry.

Related Articles

The ATL Registry Component (Registrar)

See Also

Other Resources

ATL Class Overview
Registry Macros

Running Objects Classes

The following class provides support for running objects:

- IRunnableObjectImpl Determines if an object is running, forces it to run, or locks it into the running state.

Related Articles

ATL Tutorial

See Also

Other Resources

ATL Class Overview

Security Classes

These classes are wrappers for common Win32 security classes and objects.

- CAccessToken This class is a wrapper for an access token.
- CAcl This class is a wrapper for an **ACL** (access-control list) structure.
- CDacl This class is a wrapper for a **DACL** (discretionary access-control list) structure.
- CPrivateObjectSecurityDesc This class represents a private object security descriptor object.
- CSacl This class is a wrapper for a **SACL** (system access-control list) structure.
- CSecurityAttributes This class is a thin wrapper for the **SECURITY_ATTRIBUTES** structure.
- CSecurityDesc This class is a wrapper for the **SECURITY_DESCRIPTOR** structure.
- CSid This class is a wrapper for a **SID** (security identifier) structure.
- CTokenGroups This class is a wrapper for the **TOKEN_GROUPS** structure.
- CTokenPrivileges This class is a wrapper for the **TOKEN_PRIVILEGES** structure.

See Also

Other Resources

ATL Class Overview
Security Global Functions

Service Provider Support Classes

The following class provides support for service providers:

- IServiceProviderImpl Locates a service specified by its GUID and returns the interface pointer for the requested interface on the service.

See Also

Other Resources

ATL Class Overview

Site Information Classes

The following classes allow an object to communicate with its site:

- IObjectWithSiteImpl Retrieves and sets a pointer to an object's site. Used for objects that are not controls.
- IOleObjectImpl Retrieves and sets a pointer to an object's site. Used for controls.

See Also

Other Resources

ATL Class Overview

String and Text Classes

These classes provide support for strings and text string conversions.

- CA2AEX This class is used by the string conversion macros **CA2TEX** and **CT2AEX**, and the typedef **CA2A**.
- CA2CAEX This class is used by string conversion macros **CA2CTEX** and **CT2CAEX**, and the typedef **CA2CA**.
- CA2WEX This class is used by the string conversion macros **CA2TEX**, **CA2CTEX**, **CT2WEX**, and **CT2CWEX**, and the typedef **CA2W**.
- CW2AEX This class is used by the string conversion macros **CT2AEX**, **CW2TEX**, **CW2CTEX**, and **CT2CAEX**, and the typedef **CW2A**.
- CW2CWEX This class is used by the string conversion macros **CW2CTEX** and **CT2CWEX**, and the typedef **CW2CW**.
- CW2WEX This class is used by the string conversion macros **CW2TEX** and **CT2WEX**, and the typedef **CW2W**.
- CComBSTR This class is a wrapper for **BSTR**s.
- _U_STRINGorID This argument adapter class allows either resource names (**LPCTSTR**s) or resource IDs (**UINT**s) to be passed to a function without requiring the caller to convert the ID to a string using the **MAKEINTRESOURCE** macro.

See Also

Concepts

ATL and MFC String Conversion Macros

Other Resources

ATL Class Overview

Tear-Off Interfaces Classes

The following classes provide support for tear-off interfaces:

- CComTearOffObject Implements **IUnknown** for a tear-off interface.
- CComCachedTearOffObject Implements **IUnknown** for a "cached" tear-off interface.

See Also

Other Resources

ATL Class Overview

Thread Pooling Classes

The following classes support thread pooling:

- CComAutoThreadModule Implements an EXE module, with support for multiple thread-pooled apartments.
- CComApartment Manages an apartment in a thread-pooled EXE module.
- CComSimpleThreadAllocator Manages thread selection for an EXE module.

See Also

Other Resources

ATL Class Overview

Threading Models and Critical Sections Classes

The following classes define a threading model and critical section:

- CAtlAutoThreadModule Implements a thread-pooled, apartment-model COM server.
- CAtlAutoThreadModuleT Provides methods for implementing a thread-pooled, apartment-model COM server.
- CComMultiThreadModel Provides thread-safe methods for incrementing and decrementing a variable. Provides a critical section.
- CComMultiThreadModelNoCS Provides thread-safe methods for incrementing and decrementing a variable. Does not provide a critical section.
- CComSingleThreadModel Provides methods for incrementing and decrementing a variable. Does not provide a critical section.
- CComObjectThreadModel Determines the appropriate threading-model class for a single object class.
- CComGlobalsThreadModel Determines the appropriate threading-model class for an object that is globally available.
- CComAutoCriticalSection Contains methods for obtaining and releasing a critical section. The critical section is automatically initialized.
- CComCriticalSection Contains methods for obtaining and releasing a critical section. The critical section must be explicitly initialized.
- CComFakeCriticalSection Mirrors the methods in **CComCriticalSection** without providing a critical section. The methods in **CComFakeCriticalSection** do nothing.
- CRTThreadTraits Provides the creation function for a CRT thread. Use this class if the thread will use CRT functions.
- Win32ThreadTraits Provides the creation function for a Windows thread. Use this class if the thread will not use CRT functions.

See Also

Other Resources

ATL Class Overview

UI Support Classes

The following classes provide general UI support:

- IDocHostUIHandlerDispatch An interface to the Microsoft HTML parsing and rendering engine.
- IOleObjectImpl Provides the principal methods through which a container communicates with a control. Manages the activation and deactivation of in-place controls.
- IOleInPlaceObjectWindowlessImpl Manages the reactivation of in-place controls. Enables a windowless control to receive messages, as well as to participate in drag-and-drop operations.
- IOleInPlaceActiveObjectImpl Assists communication between an in-place control and its container.
- IViewObjectExImpl Enables a control to display itself directly and to notify the container of changes in its display. Provides support for flicker-free drawing, non-rectangular and transparent controls, and hit testing.

Related Articles

ATL Tutorial

See Also

Other Resources

ATL Class Overview

Utility Classes

The following MFC-independent utility classes are provided:

- CImage Provides enhanced bitmap support, including the ability to load and save images in JPEG, GIF, BMP, and Portable Network Graphics (PNG) formats.
- CPoint Provides an implementation for storing coordinate (x, y) pairs.
- CRect Provides an implementation for storing coordinates of rectangular areas.
- CSize Provides an implementation for storing distance, relative positions, or paired values.
- CString Provides an implementation for storing character strings.
- CAdapt A simple template used to wrap classes that redefine the address-of operator.
- _U_RECT An argument adapter class that allows either **RECT** pointers or references to be passed to a function that is implemented in terms of pointers.

See Also

Other Resources

ATL Class Overview

Windows Support Classes

The following classes provide support for windows:

- _U_MENUorID Provides wrappers for **CreateWindow** and **CreateWindowEx**.
- CWindow Contains methods for manipulating a window. **CWindow** is the base class for **CWindowImpl**, **CDialogImpl**, and **CContainedWindow**.
- CWindowImpl Implements a window based on a new window class. Also allows you to subclass or superclass the window.
- CDialogImpl Implements a dialog box.
- CAxDialogImpl Implements a dialog box (modal or modeless) that hosts ActiveX controls.
- CSimpleDialog Implements a dialog box (modal or modeless) with basic functionality.
- CAxWindow Manipulates a window that hosts an ActiveX control.
- CAxWindow2T Provides methods for manipulating a window that hosts an ActiveX control and also has support for hosting licensed ActiveX controls.
- CContainedWindowT Implements a window contained within another object.
- CWndClassInfo Manages the information of a new window class.
- CDynamicChain Supports dynamic chaining of message maps.
- CMessageMap Allows an object to expose its message maps to other objects.
- CWinTraits Provides a simple method of standardizing the traits of an ATL window object.
- CWinTraitsOR Provides default values for window styles and extended styles used to create a window. These values are added, using the logical-OR operator, to values provided during the creation of a window.

Related Articles

ATL Window Classes

ATL Tutorial

See Also

Other Resources

ATL Class Overview
Message Map Macros (ATL)
Window Class Macros

ATL Classes

The Active Template Library (ATL) includes the following classes. To find a particular class by category, see the ATL Class Overview.

Class	Description	Header file
CA2AEX	This class is used by the string conversion macros **CA2TEX** and **CT2AEX**, and the typedef **CA2A**.	atlconv.h
CA2CAEX	This class is used by string conversion macros **CA2CTEX** and **CT2CAEX**, and the typedef **CA2CA**.	atlconv.h
CA2WEX	This class is used by the string conversion macros **CA2TEX**, **CA2CTEX**, **CT2WEX**, and **CT2CWEX**, and the typedef **CA2W**.	atlconv.h
CAccessToken	This class is a wrapper for an access token.	atlsecurity.h
CAcl	This class is a wrapper for an **ACL** (access-control list) structure.	atlsecurity.h
CAdapt	This template is used to wrap classes that redefine the address-of operator to return something other than the address of the object.	atlcomcli.h
CAtlArray	This class implements an array object.	atlcoll.h
CAtlAutoThreadModule	This class implements a thread-pooled, apartment-model COM server.	atlbase.h
CAtlAutoThreadModuleT	This class provides methods for implementing a thread-pooled, apartment-model COM server.	atlbase.h
CAtlBaseModule	This class is instantiated in every ATL project.	atlcore.h
CAtlComModule	This class implements a COM server module.	atlbase.h
CAtlDebugInterfacesModule	This class provides support for debugging interfaces.	atlbase.h
CAtlDllModuleT	This class represents the module for a DLL.	atlbase.h
CAtlException	This class defines an ATL exception.	atlexcept.h
CAtlExeModuleT	This class represents the module for an application.	atlbase.h
CAtlFile	This class provides a thin wrapper around the Windows file-handling API.	atlfile.h

CAtlFileMapping	This class represents a memory-mapped file, adding a cast operator to the methods of CAtlFileMappingBase.	atlfile.h
CAtlFileMappingBase	This class represents a memory-mapped file.	atlfile.h
CAtlList	This class provides methods for creating and managing a list object.	atlcoll.h
CAtlMap	This class provides methods for creating and managing a map object.	atlcoll.h
CAtlModule	This class provides methods used by several ATL module classes.	atlbase.h
CAtlModuleT	This class implements an ATL module.	atlbase.h
CAtlServiceModuleT	This class implements a service.	atlbase.h
CAtlTemporaryFile	This class provides methods for the creation and use of a temporary file.	atlfile.h
CAtlWinModule	This class provides support for ATL windowing components.	atlbase.h
CAutoPtr	This class represents a smart pointer object.	atlbase.h
CAutoPtrArray	This class provides methods useful when constructing an array of smart pointers.	atlbase.h
CAutoPtrElementTraits	This class provides methods, static functions, and typedefs useful when creating collections of smart pointers.	atlcoll.h
CAutoPtrList	This class provides methods useful when constructing a list of smart pointers.	atlcoll.h
CAutoVectorPtr	This class represents a smart pointer object using vector new and delete operators.	atlbase.h
CAutoVectorPtrElementTraits	This class provides methods, static functions, and typedefs useful when creating collections of smart pointers using vector new and delete operators.	atlcoll.h
CAxDialogImpl	This class implements a dialog box (modal or modeless) that hosts ActiveX controls.	atlwin.h
CAxWindow	This class provides methods for manipulating a window hosting an ActiveX control.	atlwin.h
CAxWindow2T	This class provides methods for manipulating a window that hosts an ActiveX control and also has support for hosting licensed ActiveX controls.	atlwin.h
CBindStatusCallback	This class implements the **IBindStatusCallback** interface.	atlctl.h
CComAggObject	This class implements IUnknown for an aggregated object.	atlcom.h

CComAllocator	This class provides methods for managing memory using COM memory routines.	atlbase.h
CComApartment	This class provides support for managing an apartment in a thread-pooled EXE module.	atlbase.h
CComAutoCriticalSection	This class provides methods for obtaining and releasing ownership of a critical section object.	atlcore.h
CComAutoThreadModule	As of ATL 7.0, **CComAutoThreadModule** is obsolete: see ATL Modules for more details.	atlbase.h
CComBSTR	This class is a wrapper for **BSTR**s.	atlbase.h
CComCachedTearOffObject	This class implements IUnknown for a tear-off interface.	atlcom.h
CComClassFactory	This class implements the IClassFactory interface.	atlcom.h
CComClassFactory2	This class implements the IClassFactory2 interface.	atlcom.h
CComClassFactoryAutoThread	This class implements the IClassFactory interface and allows objects to be created in multiple apartments.	atlcom.h
CComClassFactorySingleton	This class derives from CComClassFactory and uses CComObjectGlobal to construct a single object.	atlcom.h
CComCoClass	This class provides methods for creating instances of a class and obtaining its properties.	atlcom.h
CComCompositeControl	This class provides the methods required to implement a composite control.	atlctl.h
CComContainedObject	This class implements IUnknown by delegating to the owner object's **IUnknown**.	atlcom.h
CComControl	This class provides methods for creating and managing ATL controls.	atlctl.h
CComControlBase	This class provides methods for creating and managing ATL controls.	atlctl.h
CComCriticalSection	This class provides methods for obtaining and releasing ownership of a critical section object.	atlcore.h
CComCritSecLock	This class provides methods for locking and unlocking a critical section object.	atlbase.h
CComCurrency	This class has methods and operators for creating and managing a CURRENCY object.	atlcur.h
CComDynamicUnkArray	This class stores an array of **IUnknown** pointers.	atlcom.h

CComEnum	This class defines a COM enumerator object based on an array.	atlcom.h
CComEnumImpl	This class provides the implementation for a COM enumerator interface where the items being enumerated are stored in an array.	atlcom.h
CComEnumOnSTL	This class defines a COM enumerator object based on an STL collection.	atlcom.h
CComFakeCriticalSection	This class provides the same methods as CComCriticalSection but does not provide a critical section.	atlcore.h
CComGITPtr	This class provides methods for dealing with interface pointers and the global interface table (GIT).	atlbase.h
CComHeap	This class implements IAtlMemMgr using the COM memory allocation functions.	ATLComMem.h
CComHeapPtr	A smart pointer class for managing heap pointers.	atlbase.h
CComModule	As of ATL 7.0, **CComModule** is obsolete: see ATL Modules for more details.	atlbase.h
CComMultiThreadModel	This class provides thread-safe methods for incrementing and decrementing the value of a variable.	atlbase.h
CComMultiThreadModelNoCS	This class provides thread-safe methods for incrementing and decrementing the value of a variable, without critical section locking or unlocking functionality.	atlbase.h
CComObject	This class implements **IUnknown** for a nonaggregated object.	atlcom.h
CComObjectGlobal	This class manages a reference count on the module containing your *Base* object.	atlcom.h
CComObjectNoLock	This class implements **IUnknown** for a nonaggregated object, but does not increment the module lock count in the constructor.	atlcom.h
CComObjectRoot	This typedef of CComObjectRootEx is templatized on the default threading model of the server.	atlcom.h
CComObjectRootEx	This class provides methods to handle object reference count management for both nonaggregated and aggregated objects.	atlcom.h
CComObjectStack	This class creates a temporary COM object and provides it with a skeletal implementation of **IUnknown**.	atlcom.h
CComPolyObject	This class implements **IUnknown** for an aggregated or nonaggregated object.	atlcom.h
CComPtr	A smart pointer class for managing COM interface pointers.	atlcomcli.h

CComPtrBase	This class provides a basis for smart pointer classes using COM-based memory routines.	atlcomcli.h
CComQIPtr	A smart pointer class for managing COM interface pointers.	atlcomcli.h
CComQIPtrElementTraits	This class provides methods, static functions, and typedefs useful when creating collections of COM interface pointers.	atlcoll.h
CComSafeArray	This class is a wrapper for the SAFEARRAY Data Type structure.	atlsafe.h
CComSafeArrayBound	This class is a wrapper for a SAFEARRAYBOUND structure.	atlsafe.h
CComSimpleThreadAllocator	This class manages thread selection for the class CComAutoThreadModule.	atlbase.h
CComSingleThreadModel	This class provides methods for incrementing and decrementing the value of a variable.	atlbase.h
CComTearOffObject	This class implements a tear-off interface.	atlcom.h
CComUnkArray	This class stores **IUnknown** pointers and is designed to be used as a parameter to the IConnectionPointImpl template class.	atlcom.h
CComVariant	This class wraps the VARIANT type, providing a member indicating the type of data stored.	atlcomcli.h
CContainedWindowT	This class implements a window contained within another object.	atlwin.h
CCRTAllocator	This class provides methods for managing memory using CRT memory routines.	atlcore.h
CCRTHeap	This class implements IAtlMemMgr using the CRT heap functions.	atlmem.h
CDacl	This class is a wrapper for a DACL (discretionary access-control list) structure.	atlsecurity.h
CDefaultCharTraits	This class provides two static functions for converting characters between uppercase and lowercase.	atlcoll.h
CDefaultCompareTraits	This class provides default element comparison functions.	atlcoll.h
CDefaultElementTraits	This class provides default methods and functions for a collection class.	atlcoll.h
CDefaultHashTraits	This class provides a static function for calculating hash values.	atlcoll.h
CDialogImpl	This class provides methods for creating a modal or modeless dialog box.	atlwin.h

CDynamicChain	This class provides methods supporting the dynamic chaining of message maps.	atlwin.h
CElementTraits	This class is used by collection classes to provide methods and functions for moving, copying, comparison, and hashing operations.	atlcoll.h
CElementTraitsBase	This class provides default copy and move methods for a collection class.	atlcoll.h
CFirePropNotifyEvent	This class provides methods for notifying the container's sink regarding control property changes.	atlctl.h
CGlobalHeap	This class implements IAtlMemMgr using the Win32 global heap functions.	atlmem.h
CHandle	This class provides methods for creating and using a handle object.	atlbase.h
CHeapPtr	A smart pointer class for managing heap pointers.	atlcore.h
CHeapPtrBase	This class forms the basis for several smart heap pointer classes.	atlcore.h
CHeapPtrElementTraits Class	This class provides methods, static functions, and typedefs useful when creating collections of heap pointers.	atlcoll.h
CHeapPtrList	This class provides methods useful when constructing a list of heap pointers.	atlcoll.h
CImage	Provides enhanced bitmap support, including the ability to load and save images in JPEG, GIF, BMP, and Portable Network Graphics (PNG) formats.	atlimage.h
CInterfaceArray	This class provides methods useful when constructing an array of COM interface pointers.	atlcoll.h
CInterfaceList	This class provides methods useful when constructing a list of COM interface pointers.	atlcoll.h
CLocalHeap	This class implements IAtlMemMgr using the Win32 local heap functions.	atlmem.h
CMessageMap	This class allows an object's message maps to be accessed by another object.	atlwin.h
CPrimitiveElementTraits	This class provides default methods and functions for a collection class composed of primitive data types.	atlcoll.h
CPrivateObjectSecurityDesc	This class represents a private object security descriptor object.	atlsecurity.h
CRBMap	This class represents a mapping structure, using a Red-Black binary tree.	atlcoll.h
CRBMultiMap	This class represents a mapping structure that allows each key to be associated with more than one value, using a Red-Black binary tree.	atlcoll.h
CRBTree	This class provides methods for creating and utilizing a Red-Black tree.	atlcoll.h

CRegKey	This class provides methods for manipulating entries in the system registry.	atlbase.h
CRTThreadTraits	This class provides the creation function for a CRT thread. Use this class if the thread will use CRT functions.	atlbase.h
CSacl	This class is a wrapper for a SACL (system access-control list) structure.	atlsecurity.h
CSecurityAttributes	This class is a thin wrapper for the **SECURITY_ATTRIBUTES** structure.	atlsecurity.h
CSecurityDesc	This class is a wrapper for the **SECURITY_DESCRIPTOR** structure.	atlsecurity.h
CSid	This class is a wrapper for a **SID** (security identifier) structure.	atlsecurity.h
CSimpleArray	This class provides methods for managing a simple array.	atlsimpcoll.h
CSimpleArrayEqualHelper	This class is a helper for the CSimpleArray class.	atlsimpcoll.h
CSimpleArrayEqualHelperFalse	This class is a helper for the CSimpleArray class.	atlsimpcoll.h
CSimpleDialog	This class implements a basic modal dialog box.	atlsimpcoll.h
CSimpleMap	This class provides support for a simple mapping array.	atlsimpcoll.h
CSimpleMapEqualHelper	This class is a helper for the CSimpleMap class.	atlsimpcoll.h
CSimpleMapEqualHelperFalse	This class is a helper for the CSimpleMap class.	atlsimpcoll.h
CSnapInItemImpl	This class provides methods for implementing a snap-in node object.	atlsnap.h
CSnapInPropertyPageImpl	This class provides methods for implementing a snap-in property page object.	atlsnap.h
CStockPropImpl	This class provides methods for supporting stock property values.	atlctl.h
CStringElementTraits	This class provides static functions used by collection classes storing **CString** objects.	cstringt.h
CStringElementTraitsI	This class provides static functions related to strings stored in collection class objects. It is similar to CStringElementTraits, but performs case-insensitive comparisons.	atlcoll.h
CStringRefElementTraits	This class provides static functions related to strings stored in collection class objects. The string objects are dealt with as references.	atlcoll.h

CTokenGroups	This class is a wrapper for the **TOKEN_GROUPS** structure.	atlsecurity.h
CTokenPrivileges	This class is a wrapper for the **TOKEN_PRIVILEGES** structure.	atlsecurity.h
CW2AEX	This class is used by the string conversion macros **CT2AEX**, **CW2TEX**, **CW2CTEX**, and **CT2CAEX**, and the typedef **CW2A**.	atlconv.h
CW2CWEX	This class is used by the string conversion macros **CW2CTEX** and **CT2CWEX**, and the typedef **CW2CW**.	atlconv.h
CW2WEX	This class is used by the string conversion macros **CW2TEX** and **CT2WEX**, and the typedef **CW2W**.	atlconv.h
CWin32Heap	This class implements IAtlMemMgr using the Win32 heap allocation functions.	atlmem.h
CWindow	This class provides methods for manipulating a window.	atlwin.h
CWindowImpl	This class provides methods for creating or subclassing a window.	atlwin.h
CWinTraits	This class provides a method for standardizing the styles used when creating a window object.	atlwin.h
CWinTraitsOR	This class provides a method for standardizing the styles used when creating a window object.	atlwin.h
CWndClassInfo	This class provides methods for registering information for a window class.	atlwin.h
IAtlAutoThreadModule	This class represents an interface to a **CreateInstance** method.	atlbase.h
IAtlMemMgr	This class represents the interface to a memory manager.	atlmem.h
IAxWinAmbientDispatch	This interface provides methods for specifying characteristics of the hosted control or container.	atlbase.h, ATLIface.h
IAxWinAmbientDispatchEx	This interface implements supplemental ambient properties for a hosted control.	atlbase.h, ATLIface.h
IAxWinHostWindow	This interface provides methods for manipulating a control and its host object.	atlbase.h, ATLIface.h
IAxWinHostWindowLic	This interface provides methods for manipulating a licensed control and its host object.	atlbase.h, ATLIface.h

ICollectionOnSTLImpl	This class provides methods used by a collection class.	atlcom.h
IConnectionPointContainerImpl	This class implements a connection point container to manage a collection of IConnectionPointImpl objects.	atlcom.h
IConnectionPointImpl	This class implements a connection point.	atlcom.h
IDataObjectImpl	This class provides methods for supporting Uniform Data Transfer and managing connections.	atlctl.h
IDispatchImpl	This class provides a default implementation for the **IDispatch** portion of a dual interface.	atlctl.h
IDispEventImpl	This class provides implementations of the **IDispatch** methods.	atlcom.h
IDispEventSimpleImpl	This class provides implementations of the **IDispatch** methods, without getting type information from a type library.	atlcom.h
IDocHostUIHandlerDispatch	An interface to the Microsoft HTML parsing and rendering engine.	atlbase.h, ATLIface.h
IEnumOnSTLImpl	This class defines an enumerator interface based on an STL collection.	atlcom.h
IObjectSafetyImpl	This class provides a default implementation of the **IObjectSafety** interface to allow a client to retrieve and set an object's safety levels.	atlctl.h
IObjectWithSiteImpl	This class provides methods allowing an object to communicate with its site.	atlcom.h
IOleControlImpl	This class provides a default implementation of the **IOleControl** interface and implements **IUnknown**.	atlctl.h
IOleInPlaceActiveObjectImpl	This class provides methods for assisting communication between an in-place control and its container.	atlctl.h
IOleInPlaceObjectWindowlessImpl	This class implements **IUnknown** and provides methods that enable a windowless control to receive window messages and to participate in drag-and-drop operations.	atlctl.h
IOleObjectImpl	This class implements **IUnknown** and is the principal interface through which a container communicates with a control.	atlctl.h
IPerPropertyBrowsingImpl	This class implements **IUnknown** and allows a client to access the information in an object's property pages.	atlctl.h
IPersistPropertyBagImpl	This class implements **IUnknown** and allows an object to save its properties to a client-supplied property bag.	atlcom.h
IPersistStorageImpl	This class implements the IPersistStorage interface.	atlcom.h

IPersistStreamInitImpl	This class implements **IUnknown** and provides a default implementation of the IPersistStreamInit interface.	atlcom.h
IPointerInactiveImpl	This class implements **IUnknown** and the IPointerInactive interface methods.	atlctl.h
IPropertyNotifySinkCP	This class exposes the IPropertyNotifySink interface as an outgoing interface on a connectable object.	atlctl.h
IPropertyPage2Impl	This class implements **IUnknown** and inherits the default implementation of IPropertyPageImpl.	atlctl.h
IPropertyPageImpl	This class implements **IUnknown** and provides a default implementation of the IPropertyPage interface.	atlctl.h
IProvideClassInfo2Impl	This class provides a default implementation of the IProvideClassInfo and IProvideClassInfo2 methods.	atlcom.h
IQuickActivateImpl	This class combines containers' control initialization into a single call.	atlctl.h
IRunnableObjectImpl	This class implements **IUnknown** and provides a default implementation of the IRunnableObject interface.	atlctl.h
IServiceProviderImpl	This class provides a default implementation of the IServiceProvider interface.	atlcom.h
ISpecifyPropertyPagesImpl	This class implements **IUnknown** and provides a default implementation of the ISpecifyPropertyPages interface.	atlcom.h
ISupportErrorInfoImpl	This class provides a default implementation of the ISupportErrorInfo Interface interface and can be used when only a single interface generates errors on an object.	atlcom.h
IViewObjectExImpl	This class implements **IUnknown** and provides default implementations of the IViewObject, IViewObject2, and IViewObjectEx interfaces.	atlctl.h
_U_MENUorID	This class provides wrappers for **CreateWindow** and **CreateWindowEx**.	atlwin.h
_U_RECT	This argument adapter class allows either **RECT** pointers or references to be passed to a function that is implemented in terms of pointers.	atlwin.h
_U_STRINGorID	This argument adapter class allows either resource names (**LPCTSTR**s) or resource IDs (**UINT**s) to be passed to a function without requiring the caller to convert the ID to a string using the **MAKEINTRESOURCE** macro.	atlwin.h
Win32ThreadTraits	This class provides the creation function for a Windows thread. Use this class if the thread will not use CRT functions.	atlbase.h

See Also

Other Resources

ATL Reference
ATL Functions
ATL Global Variables
ATL Structures
ATL Typedefs
ATL Class Overview

CA2AEX Class

This class is used by the string conversion macros **CA2TEX** and **CT2AEX**, and the typedef **CA2A**.

```
template<
   int t_nBufferLength = 128
>
class CA2AEX
```

Parameters

t_nBufferLength

The size of the buffer used in the translation process. The default length is 128 bytes.

Remarks

Unless extra functionality is required, use **CA2TEX**, **CT2AEX**, or **CA2A** in your own code.

This class contains a fixed-size static buffer which is used to store the result of the conversion. If the result is too large to fit into the static buffer, the class allocates memory using **malloc**, freeing the memory when the object goes out of scope. This ensures that, unlike text conversion macros available in previous versions of ATL, this class is safe to use in loops and that it won't overflow the stack.

If the class tries to allocate memory on the heap and fails, it will call **AtlThrow** with an argument of **E_OUTOFMEMORY**.

By default, the ATL conversion classes and macros use the current thread's ANSI code page for the conversion.

The following macros are based on this class:

- **CA2TEX**
- **CT2AEX**

The following typedef is based on this class:

- **CA2A**

For a discussion of these text conversion macros, see ATL and MFC String Conversion Macros.

Example

See ATL and MFC String Conversion Macros for an example of using these string conversion macros.

Requirements

Header: atlconv.h

See Also

Reference

CA2CAEX Class
CA2WEX Class
CW2AEX Class
CW2CWEX Class
CW2WEX Class

Other Resources

CA2AEX Members
ATL Class Overview

CA2AEX Members

Methods

CA2AEX	The constructor.
~CA2AEX	The destructor.

Operators

operator LPSTR	Conversion operator.

Data Members

m_psz	The data member that stores the source string.
m_szBuffer	The static buffer, used to store the converted string.

See Also

Reference

CA2AEX Class

CA2AEX Methods

For information about the methods in **CA2AEX**, see CA2AEX Members.

CA2AEX::CA2AEX

The constructor.

```
CA2AEX(
   LPCSTR psz,
   UINT nCodePage
) throw(...);
CA2AEX(
   LPCSTR psz
) throw(...);
```

Parameters

psz

The text string to be converted.

nCodePage

Unused in this class.

Remarks

Creates the buffer required for the translation.

See Also

Reference

CA2AEX Class

Other Resources

CA2AEX Members

CA2AEX::~CA2AEX

The destructor.

```
~CA2AEX( ) throw( );
```

Remarks

Frees the allocated buffer.

See Also

Reference

CA2AEX Class

Other Resources

CA2AEX Members

CA2AEX Operators

For information about the operators in **CA2AEX**, see CA2AEX Members.

CA2AEX::operator LPSTR

Conversion operator.

```
operator LPSTR( ) const throw( );
```

Return Value

Returns the text string as type **LPSTR**.

See Also

Reference

CA2AEX Class

Other Resources

CA2AEX Members

CA2AEX Data Members

For information about the data members in **CA2AEX**, see CA2AEX Members.

CA2AEX::m_psz

The data member that stores the source string.

```
LPSTR m_psz;
```

See Also

Reference

CA2AEX Class

Other Resources

CA2AEX Members

CA2AEX::m_szBuffer

The static buffer, used to store the converted string.

```
char m_szBuffer[t_nBufferLength];
```

See Also

Reference

CA2AEX Class

Other Resources

CA2AEX Members

CA2CAEX Class

This class is used by string conversion macros **CA2CTEX** and **CT2CAEX**, and the typedef **CA2CA**.

```
template<
   int t_nBufferLength = 128
>
class CA2CAEX
```

Parameters

t_nBufferLength

The size of the buffer used in the translation process. The default length is 128 bytes.

Remarks

Unless extra functionality is required, use **CA2CTEX**, **CT2CAEX**, or **CA2CA** in your own code.

This class is safe to use in loops and won't overflow the stack. By default, the ATL conversion classes and macros will use the current thread's ANSI code page for the conversion.

The following macros are based on this class:

- **CA2CTEX**
- **CT2CAEX**

The following typedef is based on this class:

- **CA2CA**

For a discussion of these text conversion macros, see ATL and MFC String Conversion Macros.

Example

See ATL and MFC String Conversion Macros for an example of using these string conversion macros.

Requirements

Header: atlconv.h

See Also

Reference

CA2AEX Class
CA2WEX Class
CW2AEX Class
CW2CWEX Class
CW2WEX Class

Other Resources

CA2CAEX Members
ATL Class Overview

CA2CAEX Members

Methods

CA2CAEX	The constructor.
~CA2CAEX	The destructor.

Operators

operator LPCSTR	Conversion operator.

Data Members

m_psz	The data member that stores the source string.

See Also

Reference

CA2CAEX Class

CA2CAEX Methods

For information about the methods in **CA2CAEX**, see CA2CAEX Members.

CA2CAEX::CA2CAEX

The constructor.

```
CA2CAEX(
   LPCSTR psz,
   UINT nCodePage
) throw(...);
CA2CAEX(
   LPCSTR psz
) throw(...);
```

Parameters

psz

The text string to be converted.

nCodePage

Unused in this class.

Remarks

Creates the buffer required for the translation.

See Also

Reference

CA2CAEX Class

Other Resources

CA2CAEX Members

CA2CAEX::~CA2CAEX

The destructor.

```
~CA2CAEX( ) throw( );
```

Remarks

Frees the allocated buffer.

See Also

Reference

CA2CAEX Class

Other Resources

CA2CAEX Members

CA2CAEX Operators

For information about the operators in **CA2CAEX**, see CA2CAEX Members.

CA2CAEX::operator LPCSTR

Conversion operator.

```
operator LPCSTR( ) const throw( );
```

Return Value

Returns the text string as type **LPCSTR**.

See Also

Reference

CA2CAEX Class

Other Resources

CA2CAEX Members

CA2CAEX Data Members

For information about the data members in **CA2CAEX**, see CA2CAEX Members

CA2CAEX::m_psz

The data member that stores the source string.

```
LPCSTR m_psz;
```

See Also

Reference

CA2CAEX Class

Other Resources

CA2CAEX Members

CA2WEX Class

This class is used by the string conversion macros **CA2TEX**, **CA2CTEX**, **CT2WEX**, and **CT2CWEX**, and the typedef **CA2W**.

```
template<
   int t_nBufferLength = 128
>
class CA2WEX
```

Parameters

t_nBufferLength

The size of the buffer used in the translation process. The default length is 128 bytes.

Remarks

Unless extra functionality is required, use **CA2TEX**, **CA2CTEX**, **CT2WEX**, **CT2CWEX**, or **CA2W** in your code.

This class contains a fixed-size static buffer which is used to store the result of the conversion. If the result is too large to fit into the static buffer, the class allocates memory using **malloc**, freeing the memory when the object goes out of scope. This ensures that, unlike text conversion macros available in previous versions of ATL, this class is safe to use in loops and that it won't overflow the stack.

If the class tries to allocate memory on the heap and fails, it will call **AtlThrow** with an argument of **E_OUTOFMEMORY**.

By default, the ATL conversion classes and macros use the current thread's ANSI code page for the conversion. If you want to override that behavior for a specific conversion, specify the code page as the second parameter to the constructor for the class.

The following macros are based on this class:

- **CA2TEX**
- **CA2CTEX**
- **CT2WEX**
- **CT2CWEX**

The following typedef is based on this class:

- **CA2W**

For a discussion of these text conversion macros, see ATL and MFC String Conversion Macros.

Example

See ATL and MFC String Conversion Macros for an example of using these string conversion macros.

Requirements

Header: atlconv.h

See Also

Reference

CA2AEX Class
CA2CAEX Class
CW2AEX Class
CW2CWEX Class
CW2WEX Class

Other Resources

CA2WEX Members
ATL Class Overview

CA2WEX Members

Methods

CA2WEX	The constructor.
~CA2WEX	The destructor.

Operators

operator LPWSTR	Conversion operator.

Data Members

m_psz	The data member that stores the source string.
m_szBuffer	The static buffer, used to store the converted string.

See Also

Reference

CA2WEX Class

CA2WEX Methods

For information about the methods in **CA2WEX**, see CA2WEX Members.

CA2WEX::CA2WEX

The constructor.

```
CA2WEX(
   LPCSTR psz,
   UINT nCodePage
) throw(...);
CA2WEX(
   LPCSTR psz
) throw(...);
```

Parameters

psz

The text string to be converted.

nCodePage

The code page used to perform the conversion. See the code page parameter discussion for the Platform SDK function MultiByteToWideChar for more details.

Remarks

Allocates the buffer used in the translation process.

See Also

Reference

CA2WEX Class

Other Resources

CA2WEX Members

CA2WEX::~CA2WEX

The destructor.

```
~CA2WEX( ) throw( );
```

Remarks

Frees the allocated buffer.

See Also

Reference

CA2WEX Class

Other Resources

CA2WEX Members

CA2WEX Operators

For information about the operators in **CA2WEX**, see CA2WEX Members.

CA2WEX::operator LPWSTR

Conversion operator.

```
operator LPWSTR( ) const throw( );
```

Return Value

Returns the text string as type **LPWSTR.**

See Also

Reference

CA2WEX Class

Other Resources

CA2WEX Members

CA2WEX Data Members

For information about the data members in **CA2WEX**, see CA2WEX Members.

CA2WEX::m_psz

The data member that stores the source string.

```
LPWSTR m_psz;
```

See Also

Reference

CA2WEX Class

Other Resources

CA2WEX Members

CA2WEX::m_szBuffer

The static buffer, used to store the converted string.

```
wchar_t m_szBuffer[t_nBufferLength];
```

See Also

Reference

CA2WEX Class

Other Resources

CA2WEX Members

CAccessToken Class

This class is a wrapper for an access token.

```
class CAccessToken
```

Remarks

An access token is an object that describes the security context of a process or thread and is allocated to each user logged onto a Windows NT or Windows 2000 system.

For an introduction to the access control model in Windows, see Access Control in the *Platform SDK*.

Requirements

Header: atlsecurity.h

See Also

Tasks

ATLSecurity Sample: Demonstrates ATL Security Classes

Reference

Access Tokens

Other Resources

CAccessToken Members

ATL Class Overview

CAccessToken Members

Methods

Attach	Call this method to take ownership of the given access token handle.
~CAccessToken	The destructor.
CheckTokenMembership	Call this method to determine if a specified SID is enabled in the **CAccessToken** object.
CreateImpersonationToken	Call this method to create a new impersonation access token.
CreatePrimaryToken	Call this method to create a new primary token.
CreateProcessAsUser	Call this method to create a new process running in the security context of the user represented by the **CAccessToken** object.
CreateRestrictedToken	Call this method to create a new, restricted **CAccessToken** object.
Detach	Call this method to revoke ownership of the access token.
DisablePrivilege	Call this method to disable a privilege in the **CAccessToken** object.
DisablePrivileges	Call this method to disable one or more privileges in the **CAccessToken** object.
EnablePrivilege	Call this method to enable a privilege in the **CAccessToken** object.
EnablePrivileges	Call this method to enable one or more privileges in the **CAccessToken** object.
GetDefaultDacl	Call this method to return the **CAccessToken** object's default DACL.
GetEffectiveToken	Call this method to get the **CAccessToken** object equal to the access token in effect for the curr ent thread.
GetGroups	Call this method to return the **CAccessToken** object's token groups.
GetHandle	Call this method to retrieve a handle to the access token.
GetImpersonationLevel	Call this method to get the impersonation level from the access token.
GetLogonSessionId	Call this method to get the Logon Session ID associated with the **CAccessToken** object.
GetLogonSid	Call this method to get the Logon SID associated with the **CAccessToken** object.
GetOwner	Call this method to get the owner associated with the **CAccessToken** object.
GetPrimaryGroup	Call this method to get the primary group associated with the **CAccessToken** object.
GetPrivileges	Call this method to get the privileges associated with the **CAccessToken** object.
GetProcessToken	Call this method to initialize the **CAccessToken** with the access token from the given process.

GetProfile	Call this method to get the handle pointing to the user profile associated with the **CAccessToken** object.
GetSource	Call this method to get the source of the **CAccessToken** object.
GetStatistics	Call this method to get information associated with the **CAccessToken** object.
GetTerminalServicesSessionId	Call this method to get the Terminal Services Session ID associated with the **CAccessToken** object.
GetThreadToken	Call this method to initialize the **CAccessToken** with the token from the given thread.
GetTokenId	Call this method to get the Token ID associated with the **CAccessToken** object.
GetType	Call this method to get the token type of the **CAccessToken** object.
GetUser	Call this method to identify the user associated with the **CAccessToken** object.
HKeyCurrentUser	Call this method to get the handle pointing to the user profile associated with the **CAccessToken** object.
Impersonate	Call this method to assign an impersonation **CAccessToken** to a thread.
ImpersonateLoggedOnUser	Call this method to allow the calling thread to impersonate the security context of a logged-on user.
IsTokenRestricted	Call this method to test if the **CAccessToken** object contains a list of restricted SIDs.
LoadUserProfile	Call this method to load the user profile associated with the **CAccessToken** object.
LogonUser	Call this method to create a logon session for the user associated with the given credentials.
OpenCOMClientToken	Call this method from within a COM server handling a call from a client to initialize the **CAccessToken** with the access token from the COM client.
OpenNamedPipeClientToken	Call this method from within a server taking requests over a named pipe to initialize the **CAccessToken** with the access token from the client.
OpenRPCClientToken	Call this method from within a server handling a call from an RPC client to initialize the **CAccessToken** with the access token from the client.
OpenThreadToken	Call this method to set the impersonation level and then initialize the **CAccessToken** with the token from the given thread.
PrivilegeCheck	Call this method to determine whether a specified set of privileges are enabled in the **CAccessToken** object.
Revert	Call this method to stop a thread that is using an impersonation token.
SetDefaultDacl	Call this method to set the default DACL of the **CAccessToken** object.
SetOwner	Call this method to set the owner of the **CAccessToken** object.

SetPrimaryGroup	Call this method to set the primary group of the **CAccessToken** object.

See Also

Reference

CAccessToken Class

CAccess Methods

For information about the methods in **CAccessToken**, see CAccessToken Members.

CAccessToken::Attach

Call this method to take ownership of the given access token handle.

```
void Attach(
   HANDLE hToken
) throw( );
```

Parameters

hToken

A handle to the access token.

Remarks

In debug builds, an assertion error will occur if the **CAccessToken** object already has ownership of an access token.

See Also

Reference

CAccessToken Class

CAccessToken::Detach

Other Resources

CAccessToken Members

CAccessToken::~CAccessToken

The destructor.

```
virtual ~CAccessToken( ) throw( );
```

Remarks

Frees all allocated resources.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::CheckTokenMembership

Call this method to determine if a specified SID is enabled in the **CAccessToken** object.

```
bool CheckTokenMembership(
   const CSid& rSid,
   bool* pbIsMember
) const throw(...);
```

Parameters

rSid

Reference to a CSid Class object.

pbIsMember

Pointer to a variable that receives the results of the check.

Return Value

Returns true on success, false on failure.

Remarks

The **CheckTokenMembership** method checks for the presence of the SID in the user and group SIDs of the access token. If the SID is present and has the SE_GROUP_ENABLED attribute, *pbIsMember* is set to true; otherwise, it is set to false.

In debug builds, an assertion error will occur if *pbIsMember* is not a valid pointer.

Note
The **CAccessToken** object must be an impersonation token and not a primary token.

See Also

Reference

CAccessToken Class

CheckTokenMembership

Other Resources

CAccessToken Members

CAccessToken::CreateImpersonationToken

Call this method to create a new impersonation access token.

```
bool CreateImpersonationToken(
   CAccessToken* pImp,
   SECURITY_IMPERSONATION_LEVEL sil = SecurityImpersonation
) const throw(...);
```

Parameters

pImp

Pointer to the new **CAccessToken** object.

sil

Specifies a SECURITY_IMPERSONATION_LEVEL enumerated type that supplies the impersonation level of the new token.

Return Value

Returns true on success, false on failure.

Remarks

CreateImpersonationToken calls DuplicateToken to create a new impersonation token.

See Also

Reference

CAccessToken Class
CAccessToken::CreatePrimaryToken
CAccessToken::CreateRestrictedToken
CAccessToken::Revert

Other Resources

CAccessToken Members

CAccessToken::CreatePrimaryToken

Call this method to create a new primary token.

```
bool CreatePrimaryToken(
   CAccessToken* pPri,
   DWORD dwDesiredAccess = MAXIMUM_ALLOWED,
   const CSecurityAttributes* pTokenAttributes = NULL
) const throw(...);
```

Parameters

pPri

Pointer to the new **CAccessToken** object.

dwDesiredAccess

Specifies the requested access rights for the new token. The default, MAXIMUM_ALLOWED, requests all access rights that are valid for the caller. See Access Rights and Access Masks for more on access rights.

pTokenAttributes

Pointer to a SECURITY_ATTRIBUTES structure that specifies a security descriptor for the new token and determines whether child processes can inherit the token. If *pTokenAttributes* is NULL, the token gets a default security descriptor and the handle cannot be inherited.

Return Value

Returns true on success, false on failure.

Remarks

CreatePrimaryToken calls DuplicateTokenEx to create a new primary token.

See Also

Reference

CAccessToken Class
CAccessToken::CreateImpersonationToken
CAccessToken::CreateRestrictedToken

Other Resources

CAccessToken Members

CAccessToken::CreateProcessAsUser

Call this method to create a new process running in the security context of the user represented by the **CAccessToken** object.

```
bool CreateProcessAsUser(
   LPCTSTR pApplicationName,
   LPTSTR pCommandLine,
   LPPROCESS_INFORMATION pProcessInformation,
   LPSTARTUPINFO pStartupInfo,
   DWORD dwCreationFlags = NORMAL_PRIORITY_CLASS,
   bool bLoadProfile = false,
   const CSecurityAttributes* pProcessAttributes = NULL,
   const CSecurityAttributes* pThreadAttributes = NULL,
   bool bInherit = false,
   LPCTSTR pCurrentDirectory = NULL
) throw( );
```

Parameters

pApplicationName

Pointer to a null-terminated string that specifies the module to execute. This parameter may not be NULL.

pCommandLine

Pointer to a null-terminated string that specifies the command line to execute.

pProcessInformation

Pointer to a PROCESS_INFORMATION structure that receives identification information about the new process.

pStartupInfo

Pointer to a STARTUPINFO structure that specifies how the main window for the new process should appear.

dwCreationFlags

Specifies additional flags that control the priority class and the creation of the process. See the Win32 function CreateProcessAsUser for a list of flags.

bLoadProfile

If true, the user's profile is loaded with LoadUserProfile.

pProcessAttributes

Pointer to a SECURITY_ATTRIBUTES structure that specifies a security descriptor for the new process and determines whether child processes can inherit the returned handle. If *pProcessAttributes* is NULL, the process gets a default security descriptor and the handle cannot be inherited.

pThreadAttributes

Pointer to a SECURITY_ATTRIBUTES structure that specifies a security descriptor for the new thread and determines whether child processes can inherit the returned handle. If *pThreadAttributes* is NULL, the thread gets a default security descriptor and the handle cannot be inherited.

bInherit

Indicates whether the new process inherits handles from the calling process. If true, each inheritable open handle in the calling process is inherited by the new process. Inherited handles have the same value and access privileges as the original handles.

pCurrentDirectory

Pointer to a null-terminated string that specifies the current drive and directory for the new process. The string must be a full path that includes a drive letter. If this parameter is NULL, the new process will have the same current drive and directory as the calling process.

Return Value

Returns true on success, false on failure.

Remarks

CreateProcessAsUser uses the **CreateProcessAsUser** Win32 function to create a new process that runs in the security context of the user represented by the **CAccessToken** object. See the description of the CreateProcessAsUser function for a full discussion of the parameters required.

For this method to succeed, the **CAccessToken** object must hold AssignPrimaryToken (unless it is a restricted token) and IncreaseQuota privileges.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::CreateRestrictedToken

Call this method to create a new, restricted **CAccessToken** object.

```
bool CreateRestrictedToken(
   CAccessToken* pRestrictedToken,
   const CTokenGroups& SidsToDisable,
   const CTokenGroups& SidsToRestrict,
   const CTokenPrivileges& PrivilegesToDelete = CTokenPrivileges( )
) const throw(...);
```

Parameters

pRestrictedToken

The new, restricted **CAccessToken** object.

SidsToDisable

A **CTokenGroups** object that specifies the deny-only SIDs.

SidsToRestrict

A **CTokenGroups** object that specifies the restricting SIDs.

PrivilegesToDelete

A **CTokenPrivileges** object that specifies the privileges to delete in the restricted token. The default creates an empty object.

Return Value

Returns true on success, false on failure.

Remarks

CreateRestrictedToken uses the CreateRestrictedToken Win32 function to create a new **CAccessToken** object, with restrictions.

Note
This method is only available on Windows 2000 or later.

Security Note
When using **CreateRestrictedToken**, ensure the following: the existing token is valid (and not entered by the user) and *Sids ToDisable* and *PrivilegesToDelete* are both valid (and not entered by the user). If the method returns false, deny functionality.

See Also

Reference

CAccessToken Class
CAccessToken::CreatePrimaryToken
CAccessToken::CreateImpersonationToken

Other Resources

CAccessToken Members

CAccessToken::Detach

Call this method to revoke ownership of the access token.

```
HANDLE Detach( ) throw( );
```

Return Value

Returns the handle to the **CAccessToken** which has been detached.

Remarks

This method revokes the **CAccessToken**'s ownership of the access token.

See Also

Reference

CAccessToken Class

CAccessToken::Attach

Other Resources

CAccessToken Members

CAccessToken::DisablePrivilege

Call this method to disable a privilege in the **CAccessToken** object.

```
bool DisablePrivilege(
   LPCTSTR pszPrivilege,
   CTokenPrivileges* pPreviousState = NULL
) throw(...);
```

Parameters

pszPrivilege

Pointer to a string containing the privilege to disable in the **CAccessToken** object.

pPreviousState

Pointer to a **CTokenPrivileges** object which will contain the previous state of the privileges.

Return Value

Returns true on success, false on failure.

See Also

Reference

CAccessToken Class
CAccessToken::DisablePrivileges
CAccessToken::EnablePrivilege

Other Resources

CAccessToken Members

CAccessToken::DisablePrivileges

Call this method to disable one or more privileges in the **CAccessToken** object.

```
bool DisablePrivileges(
   const CAtlArray< LPCTSTR >& rPrivileges,
   CTokenPrivileges* pPreviousState = NULL
) throw(...);
```

Parameters

rPrivileges

Pointer to an array of strings containing the privileges to disable in the **CAccessToken** object.

pPreviousState

Pointer to a **CTokenPrivileges** object which will contain the previous state of the privileges.

Return Value

Returns true on success, false on failure.

See Also

Reference

CAccessToken Class
CAccessToken::DisablePrivilege
CAccessToken::EnablePrivileges

Other Resources

CAccessToken Members

CAccessToken::EnablePrivilege

Call this method to enable a privilege in the **CAccessToken** object.

```
bool EnablePrivilege(
   LPCTSTR pszPrivilege,
   CTokenPrivileges* pPreviousState = NULL
) throw(...);
```

Parameters

pszPrivilege

Pointer to a string containing the privilege to enable in the **CAccessToken** object.

pPreviousState

Pointer to a **CTokenPrivileges** object which will contain the previous state of the privileges.

Return Value

Returns true on success, false on failure.

See Also

Reference

CAccessToken Class
CAccessToken::EnablePrivileges
CAccessToken::DisablePrivilege

Other Resources

CAccessToken Members

CAccessToken::EnablePrivileges

Call this method to enable one or more privileges in the **CAccessToken** object.

```
bool EnablePrivileges(
   const CAtlArray< LPCTSTR >& rPrivileges,
   CTokenPrivileges* pPreviousState = NULL
) throw(...);
```

Parameters

rPrivileges

Pointer to an array of strings containing the privileges to enable in the **CAccessToken** object.

pPreviousState

Pointer to a **CTokenPrivileges** object which will contain the previous state of the privileges.

Return Value

Returns true on success, false on failure.

See Also

Reference

CAccessToken Class
CAccessToken::EnablePrivilege
CAccessToken::DisablePrivileges

Other Resources

CAccessToken Members

CAccessToken::GetDefaultDacl

Call this method to return the **CAccessToken** object's default DACL.

```
bool GetDefaultDacl(
   CDacl* pDacl
) const throw(...);
```

Parameters

pDacl

Pointer to the CDacl Class object which will receive the **CAccessToken** object's default DACL.

Return Value

Returns true if the default DACL has been recovered, false otherwise.

See Also

Reference

CAccessToken Class

CAccessToken::SetDefaultDacl

Other Resources

CAccessToken Members

CAccessToken::GetEffectiveToken

Call this method to get the **CAccessToken** object equal to the access token in effect for the current thread.

```
bool GetEffectiveToken(
   DWORD dwDesiredAccess
) throw( );
```

Parameters

dwDesiredAccess

Specifies an access mask that specifies the requested types of access to the access token. These requested access types are compared with the token's DACL to determine which accesses are granted or denied.

Return Value

Returns true on success, false on failure.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::GetGroups

Call this method to return the **CAccessToken** object's token groups.

```
bool GetGroups(
   CTokenGroups* pGroups
) const throw(...);
```

Parameters

pGroups

Pointer to the CTokenGroups Class object which will receive the group information.

Return Value

Returns true on success, false on failure.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::GetHandle

Call this method to retrieve a handle to the access token.

```
HANDLE GetHandle( ) const throw( );
```

Return Value

Returns a handle to the **CAccessToken** object's access token.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::GetImpersonationLevel

Call this method to get the impersonation level from the access token.

```
bool GetImpersonationLevel(
   SECURITY_IMPERSONATION_LEVEL* pImpersonationLevel
) const throw(...);
```

Parameters

pImpersonationLevel

Pointer to a SECURITY_IMPERSONATION_LEVEL enumeration type which will receive the impersonation level information.

Return Value

Returns true on success, false on failure.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::GetLogonSessionId

Call this method to get the Logon Session ID associated with the **CAccessToken** object.

```
bool GetLogonSessionId(
   LUID* pluid
) const throw(...);
```

Parameters

pluid

Pointer to a LUID which will receive the Logon Session ID.

Return Value

Returns true on success, false on failure.

Remarks

In debug builds, an assertion error will occur if *pluid* is an invalid value.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::GetLogonSid

Call this method to get the Logon SID associated with the **CAccessToken** object.

```
bool GetLogonSid(
   CSid* pSid
) const throw(...);
```

Parameters

pSid

Pointer to a CSid Class object.

Return Value

Returns true on success, false on failure.

Remarks

In debug builds, an assertion error will occur if *pSid* is an invalid value.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::GetOwner

Call this method to get the owner associated with the **CAccessToken** object.

```
bool GetOwner(
   CSid* pSid
) const throw(...);
```

Parameters

pSid

Pointer to a CSid Class object.

Return Value

Returns true on success, false on failure.

Remarks

The owner is set by default on any objects created while this access token is in effect.

See Also

Reference

CAccessToken Class

CAccessToken::SetOwner

Other Resources

CAccessToken Members

CAccessToken::GetPrimaryGroup

Call this method to get the primary group associated with the **CAccessToken** object.

```
bool GetPrimaryGroup(
   CSid* pSid
) const throw(...);
```

Parameters

pSid

Pointer to a CSid Class object.

Return Value

Returns true on success, false on failure.

Remarks

The group is set by default on any objects created while this access token is in effect.

See Also

Reference

CAccessToken Class

CAccessToken::SetPrimaryGroup

Other Resources

CAccessToken Members

CAccessToken::GetPrivileges

Call this method to get the privileges associated with the **CAccessToken** object.

```
bool GetPrivileges(
   CTokenPrivileges* pPrivileges
) const throw(...);
```

Parameters

pPrivileges

Pointer to a CTokenPrivileges Class object which will receive the privileges.

Return Value

Returns true on success, false on failure.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::GetProcessToken

Call this method to initialize the **CAccessToken** with the access token from the given process.

```
bool GetProcessToken(
   DWORD dwDesiredAccess,
   HANDLE hProcess = NULL
) throw( );
```

Parameters

dwDesiredAccess

Specifies an access mask that specifies the requested types of access to the access token. These requested access types are compared with the token's DACL to determine which accesses are granted or denied.

hProcess

Handle to the process whose access token is opened. If the default value of NULL is used, the current process is used.

Return Value

Returns true on success, false on failure.

Remarks

Calls the OpenProcessToken Win32 function.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::GetProfile

Call this method to get the handle pointing to the user profile associated with the **CAccessToken** object.

```
HANDLE GetProfile( ) const throw( );
```

Return Value

Returns a handle pointing to the user profile, or NULL if no profile exists.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::GetSource

Call this method to get the source of the **CAccessToken** object.

```
bool GetSource(
   TOKEN_SOURCE* pSource
) const throw(...);
```

Parameters

pSource

Pointer to a TOKEN_SOURCE structure.

Return Value

Returns true on success, false on failure.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::GetStatistics

Call this method to get information associated with the **CAccessToken** object.

```
bool GetStatistics(
   TOKEN_STATISTICS* pStatistics
) const throw(...);
```

Parameters

pStatistics

Pointer to a TOKEN_STATISTICS structure.

Return Value

Returns true on success, false on failure.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::GetTerminalServicesSessionId

Call this method to get the Terminal Services Session ID associated with the **CAccessToken** object.

```
bool GetTerminalServicesSessionId(
   DWORD* pdwSessionId
) const throw(...);
```

Parameters

pdwSessionId

The Terminal Services Session ID.

Return Value

Returns true on success, false on failure.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::GetThreadToken

Call this method to initialize the **CAccessToken** with the token from the given thread.

```
bool GetThreadToken(
   DWORD dwDesiredAccess,
   HANDLE hThread = NULL,
   bool bOpenAsSelf = true
) throw( );
```

Parameters

dwDesiredAccess

Specifies an access mask that specifies the requested types of access to the access token. These requested access types are compared with the token's DACL to determine which accesses are granted or denied.

hThread

Handle to the thread whose access token is opened.

bOpenAsSelf

Indicates whether the access check is to be made against the security context of the thread calling the **GetThreadToken** method or against the security context of the process for the calling thread.

If this parameter is false, the access check is performed using the security context for the calling thread. If the thread is impersonating a client, this security context can be that of a client process. If this parameter is true, the access check is made using the security context of the process for the calling thread.

Return Value

Returns true on success, false on failure.

See Also

Reference

CAccessToken Class

OpenThreadToken

Other Resources

CAccessToken Members

CAccessToken::GetTokenId

Call this method to get the Token ID associated with the **CAccessToken** object.

```
bool GetTokenId(
   LUID* pluid
) const throw(...);
```

Parameters

pluid

Pointer to a LUID which will receive the Token ID.

Return Value

Returns true on success, false on failure.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::GetType

Call this method to get the token type of the **CAccessToken** object.

```
bool GetType(
   TOKEN_TYPE* pType
) const throw(...);
```

Parameters

pType

Address of the TOKEN_TYPE variable that, on success, receives the type of the token.

Return Value

Returns true on success, false on failure.

Remarks

The **TOKEN_TYPE** enumeration type contains values that differentiate between a primary token and an impersonation token.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::GetUser

Call this method to identify the user associated with the **CAccessToken** object.

```
bool GetUser(
   CSid* pSid
) const throw(...);
```

Parameters

pSid

Pointer to a CSid Class object.

Return Value

Returns true on success, false on failure.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::HKeyCurrentUser

Call this method to get the handle pointing to the user profile associated with the **CAccessToken** object.

```
HKEY HKeyCurrentUser( ) const throw( );
```

Return Value

Returns a handle pointing to the user profile, or NULL if no profile exists.

See Also

Reference

CAccessToken Class

CAccessToken::LoadUserProfile

Other Resources

CAccessToken Members

CAccessToken::Impersonate

Call this method to assign an impersonation **CAccessToken** to a thread.

```
bool Impersonate(
   HANDLE hThread = NULL
) const throw(...);
```

Parameters

hThread

Handle to the impersonation token to assign to the thread. This handle must have been opened with TOKEN_IMPERSONATE access rights. If *hThread* is NULL, the method causes the thread to stop using an impersonation token.

Return Value

Returns true on success, false on failure.

Remarks

In debug builds, an assertion error will occur if **CAccessToken** does not have a valid pointer to a token.

The CAutoRevertImpersonation class can be used to automatically revert impersonated access tokens.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::ImpersonateLoggedOnUser

Call this method to allow the calling thread to impersonate the security context of a logged-on user.

```
bool ImpersonateLoggedOnUser( ) const throw(...);
```

Return Value

Returns true on success, false on failure.

Remarks

Security Note
If a call to an impersonation function fails for any reason, the client is not impersonated and the client request is made in the security context of the process from which the call was made. If the process is running as a highly privileged account, or as a member of an administrative group, the user might be able to perform actions he or she would otherwise be disallowed. Therefore, the return value for this function should always be confirmed.

See Also

Reference

CAccessToken Class
CAccessToken::LoadUserProfile
CAccessToken::LogonUser

Other Resources

CAccessToken Members

CAccessToken::IsTokenRestricted

Call this method to test if the **CAccessToken** object contains a list of restricted SIDs.

```
bool IsTokenRestricted( ) const throw( );
```

Return Value

Returns true if the object contains a list of restricting SIDs, false if there are no restricting SIDs or if the method fails.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::LoadUserProfile

Call this method to load the user profile associated with the **CAccessToken** object.

```
bool LoadUserProfile( ) throw(...);
```

Return Value

Returns true on success, false on failure.

Remarks

In debug builds, an assertion error will occur if the **CAccessToken** does not contain a valid token, or if a user profile already exists.

See Also

Reference

CAccessToken Class
CAccessToken::ImpersonateLoggedOnUser
CAccessToken::HKeyCurrentUser

Other Resources

CAccessToken Members

CAccessToken::LogonUser

Call this method to create a logon session for the user associated with the given credentials.

```
bool LogonUser(
   LPCTSTR pszUserName,
   LPCTSTR pszDomain,
   LPCTSTR pszPassword,
   DWORD dwLogonType = LOGON32_LOGON_INTERACTIVE,
   DWORD dwLogonProvider = LOGON32_PROVIDER_DEFAULT
) throw( );
```

Parameters

pszUserName

Pointer to a null-terminated string that specifies the user name. This is the name of the user account to log on to.

pszDomain

Pointer to a null-terminated string that specifies the name of the domain or server whose account database contains the *pszUserName* account.

pszPassword

Pointer to a null-terminated string that specifies the clear-text password for the user account specified by *pszUserName*.

dwLogonType

Specifies the type of logon operation to perform. See LogonUser for more details.

dwLogonProvider

Specifies the logon provider. See LogonUser for more details.

Return Value

Returns true on success, false on failure.

Remarks

The access token resulting from the logon will be associated with the **CAccessToken**. For this method to succeed, the **CAccessToken** object must hold SE_TCB_NAME privileges, identifying the holder as part of the trusted computer base. See LogonUser for more information regarding the privileges required.

See Also

Reference

CAccessToken Class

CAccessToken::ImpersonateLoggedOnUser

Other Resources

CAccessToken Members

CAccessToken::OpenCOMClientToken

Call this method from within a COM server handling a call from a client to initialize the **CAccessToken** with the access token from the COM client.

```
bool OpenCOMClientToken(
   DWORD dwDesiredAccess,
   bool bImpersonate = false,
   bool bOpenAsSelf = true
) throw(...);
```

Parameters

dwDesiredAccess

Specifies an access mask that specifies the requested types of access to the access token. These requested access types are compared with the token's DACL to determine which accesses are granted or denied.

bImpersonate

If true, the current thread will impersonate the calling COM client if this call completes successfully. If false, the access token will be opened, but the thread will not have an impersonation token when this call completes.

bOpenAsSelf

Indicates whether the access check is to be made against the security context of the thread calling the GetThreadToken method or against the security context of the process for the calling thread.

If this parameter is false, the access check is performed using the security context for the calling thread. If the thread is impersonating a client, this security context can be that of a client process. If this parameter is true, the access check is made using the security context of the process for the calling thread.

Return Value

Returns true on success, false on failure.

Remarks

The CAutoRevertImpersonation Class can be used to automatically revert impersonated access tokens created by setting the *bImpersonate* flag to *true*.

See Also

Reference

CAccessToken Class
CAccessToken::OpenNamedPipeClientToken
CAccessToken::OpenRPCClientToken

Other Resources

CAccessToken Members

CAccessToken::OpenNamedPipeClientToken

Call this method from within a server taking requests over a named pipe to initialize the **CAccessToken** with the access token from the client.

```
bool OpenNamedPipeClientToken(
   HANDLE hPipe,
   DWORD dwDesiredAccess,
   bool bImpersonate = false,
   bool bOpenAsSelf = true
) throw(...);
```

Parameters

hPipe

Handle to a named pipe.

dwDesiredAccess

Specifies an access mask that specifies the requested types of access to the access token. These requested access types are compared with the token's DACL to determine which accesses are granted or denied.

bImpersonate

If true, the current thread will impersonate the calling pipe client if this call completes successfully. If false, the access token will be opened, but the thread will not have an impersonation token when this call completes.

bOpenAsSelf

Indicates whether the access check is to be made against the security context of the thread calling the GetThreadToken method or against the security context of the process for the calling thread.

If this parameter is false, the access check is performed using the security context for the calling thread. If the thread is impersonating a client, this security context can be that of a client process. If this parameter is true, the access check is made using the security context of the process for the calling thread.

Return Value

Returns true on success, false on failure.

Remarks

The CAutoRevertImpersonation Class can be used to automatically revert impersonated access tokens created by setting the *bImpersonate* flag to *true*.

See Also

Reference

CAccessToken Class
CAccessToken::OpenCOMClientToken
CAccessToken::OpenRPCClientToken

Other Resources

CAccessToken Members

CAccessToken::OpenRPCClientToken

Call this method from within a server handling a call from an RPC client to initialize the **CAccessToken** with the access token from the client.

```
bool OpenRPCClientToken(
   RPC_BINDING_HANDLE BindingHandle,
   DWORD dwDesiredAccess,
   bool bImpersonate = false,
   bool bOpenAsSelf = true
) throw(...);
```

Parameters

BindingHandle

Binding handle on the server that represents a binding to a client.

dwDesiredAccess

Specifies an access mask that specifies the requested types of access to the access token. These requested access types are compared with the token's DACL to determine which accesses are granted or denied.

bImpersonate

If true, the current thread will impersonate the calling RPC client if this call completes successfully. If false, the access token will be opened, but the thread will not have an impersonation token when this call completes.

bOpenAsSelf

Indicates whether the access check is to be made against the security context of the thread calling the GetThreadToken method or against the security context of the process for the calling thread.

If this parameter is false, the access check is performed using the security context for the calling thread. If the thread is impersonating a client, this security context can be that of a client process. If this parameter is true, the access check is made using the security context of the process for the calling thread.

Return Value

Returns true on success, false on failure.

Remarks

The CAutoRevertImpersonation Class can be used to automatically revert impersonated access tokens created by setting the *bImpersonate* flag to *true*.

Requirements

Library: rpcrt4.lib

See Also

Reference

CAccessToken Class
CAccessToken::OpenCOMClientToken
CAccessToken::OpenNamedPipeClientToken

Other Resources

CAccessToken Members

CAccessToken::OpenThreadToken

Call this method to set the impersonation level and then initialize the **CAccessToken** with the token from the given thread.

```
bool OpenThreadToken(
   DWORD dwDesiredAccess,
   bool bImpersonate = false,
   bool bOpenAsSelf = true,
   SECURITY_IMPERSONATION_LEVEL sil = SecurityImpersonation
) throw(...);
```

Parameters

dwDesiredAccess

Specifies an access mask that specifies the requested types of access to the access token. These requested access types are compared with the token's DACL to determine which accesses are granted or denied.

bImpersonate

If true, the thread will be left at the requested impersonation level after this method completes. If false, the thread will revert to its original impersonation level.

bOpenAsSelf

Indicates whether the access check is to be made against the security context of the thread calling the GetThreadToken method or against the security context of the process for the calling thread.

If this parameter is false, the access check is performed using the security context for the calling thread. If the thread is impersonating a client, this security context can be that of a client process. If this parameter is true, the access check is made using the security context of the process for the calling thread.

sil

Specifies a SECURITY_IMPERSONATION_LEVEL enumerated type that supplies the impersonation level of the token.

Return Value

Returns true on success, false on failure.

Remarks

OpenThreadToken is similar to CAccessToken::GetThreadToken, but sets the impersonation level before initializing the **CAccessToken** from the thread's access token.

The CAutoRevertImpersonation Class can be used to automatically revert impersonated access tokens created by setting the *bImpersonate* flag to *true*.

See Also

Reference

CAccessToken Class

CAccessToken::GetThreadToken

Other Resources

CAccessToken Members

CAccessToken::PrivilegeCheck

Call this method to determine whether a specified set of privileges are enabled in the **CAccessToken** object.

```
bool PrivilegeCheck(
   PPRIVILEGE_SET RequiredPrivileges,
   bool* pbResult
) const throw( );
```

Parameters

RequiredPrivileges

Pointer to a PRIVILEGE_SET structure.

pbResult

Pointer to a value the method sets to indicate whether any or all of the specified privilege are enabled in the **CAccessToken** object.

Return Value

Returns true on success, false on failure.

Remarks

When **PrivilegeCheck** returns, the **Attributes** member of each LUID_AND_ATTRIBUTES structure is set to SE_PRIVILEGE_USED_FOR_ACCESS if the corresponding privilege is enabled. This method calls the PrivilegeCheck Win32 function.

See Also

Reference

CAccessToken Class

Other Resources

CAccessToken Members

CAccessToken::Revert

Call this method to stop a thread from using an impersonation token.

```
bool Revert(
   HANDLE hThread = NULL
) const throw( );
```

Parameters

hThread

Handle to the thread to revert from impersonation. If *hThread* is NULL, the current thread is assumed.

Return Value

Returns true on success, false on failure.

Remarks

The reversion of impersonation tokens can be performed automatically with the CAutoRevertImpersonation Class.

See Also

Reference

CAccessToken Class

CAccessToken::CreateImpersonationToken

Other Resources

CAccessToken Members

CAccessToken::SetDefaultDacl

Call this method to set the default DACL of the **CAccessToken** object.

```
bool SetDefaultDacl(
   const CDacl& rDacl
) throw(...);
```

Parameters

rDacl

The new default CDacl Class information.

Return Value

Returns true on success, false on failure.

Remarks

The default DACL is the DACL that is used by default when new objects are created with this access token in effect.

See Also

Reference

CAccessToken Class

CAccessToken::GetDefaultDacl

Other Resources

CAccessToken Members

CAccessToken::SetOwner

Call this method to set the owner of the **CAccessToken** object.

```
bool SetOwner(
   const CSid& rSid
) throw(...);
```

Parameters

rSid

The CSid Class object containing the owner information.

Return Value

Returns true on success, false on failure.

Remarks

The owner is the default owner that is used for new objects created while this access token is in effect.

See Also

Reference

CAccessToken Class

CAccessToken::GetOwner

Other Resources

CAccessToken Members

CAccessToken::SetPrimaryGroup

Call this method to set the primary group of the **CAccessToken** object.

```
bool SetPrimaryGroup(
   const CSid& rSid
) throw(...);
```

Parameters

rSid

The CSid Class object containing the primary group information.

Return Value

Returns true on success, false on failure.

Remarks

The primary group is the default group for new objects created while this access token is in effect.

See Also

Reference

CAccessToken Class

CAccessToken::GetPrimaryGroup

Other Resources

CAccessToken Members

CAcl Class

This class is a wrapper for an **ACL** (access-control list) structure.

```
class CAcl
```

Remarks

The **ACL** structure is the header of an ACL (access-control list). An ACL includes a sequential list of zero or more ACEs (access-control entries). The individual ACEs in an ACL are numbered from 0 to *n-1*, where *n* is the number of ACEs in the ACL. When editing an ACL, an application refers to an access-control entry (ACE) within the ACL by its index.

There are two ACL types:

- Discretionary
- System

A discretionary ACL is controlled by the owner of an object or anyone granted **WRITE_DAC** access to the object. It specifies the access particular users and groups can have to an object. For example, the owner of a file can use a discretionary ACL to control which users and groups can and cannot have access to the file.

An object can also have system-level security information associated with it, in the form of a system ACL controlled by a system administrator. A system ACL can allow the system administrator to audit any attempts to gain access to an object.

For more details, see the ACL discussion in the *Platform SDK*.

For an introduction to the access control model in Windows, see Access Control in the *Platform SDK*.

Requirements

Header: atlsecurity.h

See Also

Other Resources

CAcl Members
ATL Class Overview
Security Global Functions

CAcl Members

Methods

CAcl	The constructor.
~CAcl	The destructor.
GetAceCount	Returns the number of access-control entry (ACE) objects.
GetAclEntries	Retrieves the access-control list (ACL) entries from the **CAcl** object.
GetAclEntry	Retrieves all of the information about an entry in a **CAcl** object.
GetLength	Returns the length of the ACL.
GetPACL	Returns a PACL (pointer to an ACL).
IsEmpty	Tests the **CAcl** object for entries.
IsNull	Returns the status of the **CAcl** object.
RemoveAce	Removes a specific ACE (access-control entry) from the **CAcl** object.
RemoveAces	Removes alls ACEs (access-control entries) from the **CAcl** that apply to the given **CSid**.
SetEmpty	Marks the **CAcl** object as empty.
SetNull	Marks the **CAcl** object as NULL.

Operators

operator =	Assignment operator.
operator const ACL *	Casts a **CAcl** object to an **ACL** structure.

Typedefs

CAccessMaskArray	An array of ACCESS_MASKs.
CAceFlagArray	An array of BYTEs.
CAceTypeArray	An array of BYTEs.

See Also

Reference

CAcl Class

CAcl Methods

For information about the methods in **CAcl**, see CAcl Members.

CAcl::CAcl

The constructor.

```
CAcl( ) throw( );
CAcl(
   const CAcl & rhs
) throw(...);
```

Parameters

rhs

An existing **CAcl** object.

Remarks

The **CAcl** object can be optionally created using an existing **CAcl** object.

See Also

Reference

CAcl Class

CAcl::~CAcl

Other Resources

CAcl Members

CAcl::~CAcl

The destructor.

```
virtual ~CAcl( ) throw( );
```

Remarks

The destructor frees any resources acquired by the object.

See Also

Reference

CAcl Class

CAcl::CAcl

Other Resources

CAcl Members

CAcl::GetAceCount

Returns the number of access-control entry (ACE) objects.

```
virtual UINT GetAceCount( ) const throw( ) = 0;
```

Return Value

Returns the number of ACE entries in the **CAcl** object.

See Also

Reference

CAcl Class

Other Resources

CAcl Members

CAcl::GetAclEntries

Retrieves the access-control list (ACL) entries from the **CAcl** object.

```
void GetAclEntries(
   CSid::CSidArray * pSids,
   CAccessMaskArray * pAccessMasks = NULL,
   CAceTypeArray * pAceTypes = NULL,
   CAceFlagArray * pAceFlags = NULL
) const throw(...);
```

Parameters

pSids

A pointer to an array of CSid objects.

pAccessMasks

The access masks.

pAceTypes

The access-control entry (**ACE**) types.

pAceFlags

The **ACE** flags.

Remarks

This method fills the array parameters with the details of every **ACE** object contained in the **CAcl** object. Use NULL when the details for that particular array are not required.

The contents of each array correspond to each other, that is, the first element of the **CAccessMaskArray** array corresponds to the first element in the **CSidArray** array, and so on.

See ACE_HEADER for more details on ACE types and flags.

See Also

Reference

CAcl Class
CAcl::CAceFlagArray
CAcl::CAceTypeArray
CAcl::CAccessMaskArray
CAcl::GetAclEntry

Other Resources

CAcl Members

CAcl::GetAclEntry

Retrieves all of the information about an entry in an access-control list (ACL).

```
void GetAclEntry(
   UINT nIndex,
   CSid * pSid,
   ACCESS_MASK * pMask = NULL,
   BYTE * pType = NULL,
   BYTE * pFlags = NULL,
   GUID * pObjectType = NULL,
   GUID * pInheritedObjectType = NULL
) const throw(...);
```

Parameters

nIndex

Index to the ACL entry to retrieve.

pSid

The CSid object to which the ACL entry applies.

pMask

The mask specifying permissions to grant or deny access.

pType

The ACE type.

pFlags

The ACE flags.

pObjectType

The object type. This will be set to GUID_NULL if the object type is not specified in the ACE, or if the ACE is not an OBJECT ACE.

pInheritedObjectType

The inherited object type. This will be set to GUID_NULL if the inherited object type is not specified in the ACE, or if the ACE is not an OBJECT ACE.

Remarks

This method will retrieve all of the information about an individual ACE, providing more information than CAcl::GetAclEntries alone makes available.

See ACE_HEADER for more details on ACE types and flags.

See Also

Reference

CAcl Class
CAcl::CAceFlagArray
CAcl::CAceTypeArray
ACCESS_MASK
CAcl::GetAclEntries

Other Resources

CAcl Members

CAcl::GetLength

Returns the length of the access-control list (ACL).

```
UINT GetLength( ) const throw( );
```

Return Value

Returns the required length in bytes necessary to hold the **ACL** structure.

See Also

Reference

CAcl Class

Other Resources

CAcl Members

CAcl::GetPACL

Returns a pointer to an access-control list (ACL).

```
const ACL * GetPACL( ) const throw(...);
```

Return Value

Returns a pointer to the **ACL** structure.

See Also

Reference

CAcl Class

Other Resources

CAcl Members

CAcl::IsEmpty

Tests the **CAcl** object for entries.

```
bool IsEmpty( ) const throw( );
```

Remarks

Returns **true** if the **CAcl** object is not NULL, and contains no entries. Returns **false** if the **CAcl** object is either NULL, or contains at least one entry.

See Also

Reference

CAcl Class

CAcl::SetEmpty

Other Resources

CAcl Members

CAcl::IsNull

Returns the status of the **CAcl** object.

```
bool IsNull( ) const throw( );
```

Return Value

Returns **true** if the **CAcl** object is NULL, **false** otherwise.

See Also

Reference

CAcl Class

CAcl::SetNull

Other Resources

CAcl Members

CAcl::RemoveAce

Removes a specific ACE (access-control entry) from the **CAcl** object.

```
void RemoveAce(
   UINT nIndex
) throw( );
```

Parameters

nIndex

Index to the ACE entry to remove.

Remarks

This method is derived from CAtlArray::RemoveAt.

See Also

Reference

CAcl Class

CAcl::RemoveAces

Other Resources

CAcl Members

CAcl::RemoveAces

Removes alls ACEs (access-control entries) from the **CAcl** that apply to the given **CSid**.

```
bool RemoveAces(
   const CSid & rSid
) throw(...)
```

Parameters

rSid

A reference to a **CSid** object.

Returns

Returns true if successful, false if no ACEs are removed or the **CAcl** or **CSid** objects are invalid.

See Also

Reference

CAcl Class

CAcl::RemoveAce

Other Resources

CAcl Members

CAcl::SetEmpty

Marks the **CAcl** object as empty.

```
void SetEmpty( ) throw( );
```

Remarks

The **CAcl** can be set to empty or to NULL: the two states are distinct.

See Also

Reference

CAcl Class

CAcl::SetNull

Other Resources

CAcl Members

CAcl::SetNull

Marks the **CAcl** object as NULL.

```
void SetNull( ) throw( );
```

Remarks

The **CAcl** can be set to empty or to NULL: the two states are distinct.

See Also

Reference

CAcl Class

CAcl::IsNull

Other Resources

CAcl Members

CAcl Operators

For information about the operators in **CAcl**, see CAcl Members.

CAcl::operator =

Assignment operator.

```
CAcl & operator =(
   const CAcl & rhs
) throw(...);
```

Parameters

rhs

The **CAcl** to assign to the existing object.

Return Value

Returns a reference to the updated **CAcl** object.

See Also

Reference

CAcl Class

Other Resources

CAcl Members

CAcl::operator const ACL *

Casts a **CAcl** object to an **ACL** (access-control list) structure.

```
operator const ACL *( ) const throw(...);
```

Remarks

Returns the address of the **ACL** structure.

See Also

Reference

CAcl Class

Other Resources

CAcl Members

CAcl Typedefs

For information about the typedefs in **CAcl**, see CAcl Members.

CAcl::CAccessMaskArray

An array of ACCESS_MASK objects.

```
typedef CAtlArray<ACCESS_MASK> CAccessMaskArray;
```

Remarks

This typedef specifies the array type that can be used to store access rights used in access-control entries (ACEs).

See Also

Reference

CAcl Class

ACCESS_MASK

Other Resources

CAcl Members

CAcl::CAceFlagArray

An array of BYTEs.

```
typedef CAtlArray<BYTE> CAceFlagArray;
```

Remarks

This typedef specifies the array type used to define the access-control entry (ACE) type-specific control flags. See the ACE_HEADER definition for the complete list of possible flags.

See Also

Reference

CAcl Class

ACE_HEADER

Other Resources

CAcl Members

CAcl::CAceTypeArray

An array of BYTEs.

```
typedef CAtlArray<BYTE> CAceTypeArray;
```

Remarks

This typedef specifies the array type used to define the nature of the access-control entry (ACE) objects, such as ACCESS_ALLOWED_ACE_TYPE or ACCESS_DENIED_ACE_TYPE. See the ACE_HEADER definition for the complete list of possible types.

See Also

Reference

CAcl Class

ACE_HEADER

Other Resources

CAcl Members

CAdapt Class

This template is used to wrap classes that redefine the address-of operator to return something other than the address of the object.

```
template <
   class T
>
class CAdapt
```

Parameters

T

The adapted type.

Remarks

CAdapt is a simple template used to wrap classes that redefine the address-of operator (**operator &**) to return something other than the address of the object. Examples of such classes include ATL's **CComBSTR**, **CComPtr**, and **CComQIPtr** classes, and the compiler COM support class, **_com_ptr_t**. These classes all redefine the address-of operator to return the address of one of their data members (a **BSTR** in the case of **CComBSTR**, and an interface pointer in the case of the other classes).

CAdapt's primary role is to hide the address-of operator defined by class *T*, yet still retain the characteristics of the adapted class. **CAdapt** fulfils this role by holding a public member, m_T, of type *T*, and by defining conversion operators, comparison operators, and a copy constructor to allow specializations of **CAdapt** to be treated as if they are objects of type *T*.

The adapter class **CAdapt** is useful because many container classes (such as the STL container classes) expect to be able to obtain the addresses of their contained objects using the address-of operator. The redefinition of the address-of operator can confound this requirement, typically causing compilation errors and preventing the use of the unadapted type with that container. **CAdapt** provides a way around those problems.

Typically, you will use **CAdapt** when you want to store **CComBSTR**, **CComPtr**, **CComQIPtr**, or **_com_ptr_t** objects in an STL container such as a **list**. You can't store objects of these types like this:

```
std::list< CComBSTR > m_List;
```

Instead, you should store adapter objects like this:

```
std::list< CAdapt< CComBSTR > > m_List;
```

Requirements

Header: atlcomcli.h

See Also

Other Resources

CAdapt Members

ATL Class Overview

CAdapt Members

Constructor

CAdapt	Constructor.

Operators

operator const T&	Returns a **const** reference to **m_T**.
operator T&	Returns a reference to **m_T**.
operator<	Compares an object of the adapted type with **m_T**.
operator=	Assigns an object of the adapted type to **m_T**.
operator==	Compares an object of the adapted type with **m_T**.

Data Members

m_T	The data being adapted.

See Also

Reference

CAdapt Class

CAdapt Methods

For information about the methods in **CAdapt**, see CAdapt Members.

CAdapt::CAdapt

The constructors allow an adapter object to be default constructed, copied from an object of the adapted type, or copied from another adapter object.

```
CAdapt( );
CAdapt(
   const T& rSrc
);
CAdapt(
   const CAdapt& rSrCA
);
```

Parameters

rSrc

A variable of the type being adapted to be copied into the newly constructed adapter object.

rSrCA

An adapter object whose contained data should be copied into the newly constructed adapter object.

See Also

Reference

CAdapt Class

CAdapt::operator =

Other Resources

CAdapt Members

CAdapt Operators

For information about the operators in **CAdapt**, see CAdapt Members.

CAdapt::operator =

The assignment operator assigns the argument, *rSrc*, to the data member m_T and returns the current adapter object.

```
CAdapt& operator=(
   const T& rSrc
);
```

Parameters

rSrc

A reference to an object of the adapted type to be copied.

Return Value

A reference to the current object.

See Also

Reference

CAdapt Class

CAdapt::CAdapt

Other Resources

CAdapt Members

CAdapt::operator <

Compares an object of the adapted type with m_T.

```
bool operator<(
   const T& rSrc
) const;
```

Parameters

rSrc

A reference to the object to be compared.

Return Value

The result of the comparison between **m_T** and *rSrc*.

See Also

Reference

CAdapt Class

CAdapt::operator ==

Other Resources

CAdapt Members

CAdapt::operator ==

Compares an object of the adapted type with m_T.

```
bool operator==(
   const T& rSrc
) const;
```

Parameters

rSrc

A reference to the object to be compared.

Return Value

The result of the comparison between **m_T** and *rSrc*.

See Also

Reference

CAdapt Class

CAdapt::operator <

Other Resources

CAdapt Members

CAdapt::operator T&

Returns a reference to the m_T member, allowing the adapter object to be treated as if it were an object of type *T*.

```
operator T&( );
```

Return Value

A reference to **m_T**.

See Also

Reference

CAdapt Class

operator const T&

Other Resources

CAdapt Members

CAdapt::operator const T&

Returns a **const** reference to the m_T member, allowing the adapter object to be treated as if it were an object of type *T*.

```
operator const T&( ) const;
```

Return Value

A **const** reference to **m_T**.

See Also

Reference

CAdapt Class

CAdapt::operator T&

Other Resources

CAdapt Members

CAdapt Data Members

For information about the data members in **CAdapt**, see CAdapt Members.

CAdapt::m_T

Holds the data being adapted.

```
T m_T;
```

Remarks

This **public** data member can be accessed directly or indirectly with operator const T& and operator T&.

See Also

Reference

CAdapt Class

Other Resources

CAdapt Members

CAtlArray Class

This class implements an array object.

```
template<
   typename E,
   class ETraits = CElementTraits< E >
>
class CAtlArray
```

Parameters

E

The type of data to be stored in the array.

ETraits

The code used to copy or move elements.

Remarks

CAtlArray provides methods for creating and managing an array of elements of a user-defined type. Although similar to standard C arrays, the **CAtlArray** object can dynamically shrink and grow as necessary. The array index always starts at position 0, and the upper bound can be fixed, or allowed to expand as new elements are added.

For arrays with a small number of elements, the ATL class CSimpleArray can be used.

CAtlArray is closely related to MFC's **CArray** class and will work in an MFC project, albeit without serialization support.

For more information, see ATL Collection Classes.

Requirements

Header: atlcoll.h

See Also

Tasks

MMXSwarm Sample: Demonstrates CImage and Visual C++ MMX Support
DynamicConsumer Sample: Uses Dynamic Accessor and Schema Rowset Classes to Read Metadata from a Database
UpdatePV Sample: Implements an Updatable OLE DB Provider
Marquee Sample: Provides Performance Monitoring Data Through Scripting

Reference

CArray Class

Other Resources

CAtlArray Members
ATL Class Overview

CAtlArray Members

Methods

Add	Call this method to add an element to the array object.
Append	Call this method to add the contents of one array to the end of another.
AssertValid	Call this method to confirm that the array object is valid.
CAtlArray	The constructor.
~CAtlArray	The destructor.
Copy	Call this method to copy the elements of one array to another.
FreeExtra	Call this method to remove any empty elements from the array.
GetAt	Call this method to retrieve a single element from the array object.
GetCount	Call this method to return the number of elements stored in the array.
GetData	Call this method to return a pointer to the first element in the array.
InsertArrayAt	Call this method to insert one array into another.
InsertAt	Call this method to insert a new element (or multiple copies of an element) into the array object.
IsEmpty	Call this method to test if the array is empty.
RemoveAll	Call this method to remove all elements from the array object.
RemoveAt	Call this method to remove one or more elements from the array.
SetAt	Call this method to set the value of an element in the array object.
SetAtGrow	Call this method to set the value of an element in the array object, expanding the array as required.
SetCount	Call this method to set the size of the array object.

Operators

operator []	Call this operator to return a reference to an element in the array.

Typedefs

INARGTYPE	The data type to use for adding elements to the array.
OUTARGTYPE	The data type to use for retrieving elements from the array.

See Also

Reference

CAtlArray Class

CAtlArray Methods

For information about the methods in **CAtlArray**, see CAtlArray Members.

CAtlArray::Add

Call this method to add an element to the array object.

```
size_t Add(
   INARGTYPE element
);
size_t Add( );
```

Parameters

element

The element to be added to the array.

Return Value

Returns the index of the added element.

Remarks

The new element is added to the end of the array. If no element is provided, an empty element is added; that is, the array is increased in size as though a real element has been added. If the operation fails, AtlThrow is called with the argument E_OUTOFMEMORY.

Example

```
// Declare an array of integers
CAtlArray<int> iArray;

iArray.Add(1);   // element 0
iArray.Add(2);   // element 1
iArray.Add();    // element 2

ATLASSERT(iArray.GetCount() == 3);
```

See Also

Reference

CAtlArray Class

CAtlArray::InsertAt

Other Resources

CAtlArray Members

CAtlArray::Append

Call this method to add the contents of one array to the end of another.

```
size_t Append(
   const CAtlArray< E, ETraits >& aSrc
);
```

Parameters

aSrc

The array to append.

Return Value

Returns the index of the first appended element.

Remarks

The elements in the supplied array are added to the end of the existing array. If necessary, memory will be allocated to accommodate the new elements.

The arrays must be of the same type, and it is not possible to append an array to itself.

In debug builds, an ATLASSERT will be raised if the **CAtlArray** argument is not a valid array or if *aSrc* refers to the same object. In release builds, invalid arguments may lead to unpredictable behavior.

Example

```
// Declare two integer arrays
CAtlArray<int> iArray1,iArray2;

iArray1.Add(1);   // element 0
iArray1.Add(2);   // element 1

iArray2.Add(3);   // element 0
iArray2.Add(4);   // element 1

// Append iArray2 to iArray1
iArray1.Append(iArray2);

ATLASSERT(iArray1.GetCount() == 4);
```

See Also

Reference

CAtlArray Class

CAtlArray::InsertArrayAt

Other Resources

CAtlArray Members

CAtlArray::AssertValid

Call this method to confirm that the array object is valid.

```
void AssertValid( ) const;
```

Remarks

If the array object is not valid, **ATLASSERT** will throw an assertion. This method is available only if _DEBUG is defined.

Example

```
CAtlArray<float> fArray;
fArray.AssertValid();
```

See Also

Reference

CAtlArray Class

CAtlArray::IsEmpty

Other Resources

CAtlArray Members

CAtlArray::CAtlArray

The constructor.

```
CAtlArray( ) throw( );
```

Remarks

Initializes the array object.

Example

```
CAtlArray<int> iArray;
```

See Also

Reference

CAtlArray Class

Other Resources

CAtlArray Members

CAtlArray::~CAtlArray

The destructor.

```
~CAtlArray( ) throw( );
```

Remarks

Frees up any resources used by the array object.

See Also

Reference

CAtlArray Class

Other Resources

CAtlArray Members

CAtlArray::Copy

Call this method to copy the elements of one array to another.

```
void Copy(
   const CAtlArray< E, ETraits >& aSrc
);
```

Parameters

aSrc

The source of the elements to copy to an array.

Remarks

Call this method to overwrite elements of one array with the elements of another array. If necessary, memory will be allocated to accommodate the new elements. It is not possible to copy elements of an array to itself.

If the existing contents of the array are to be retained, use CAtlArray::Append instead.

In debug builds, an ATLASSERT will be raised if the existing **CAtlArray** object is not valid, or if *aSrc* refers to the same object. In release builds, invalid arguments may lead to unpredictable behavior.

Note
CAtlArray::Copy does not support arrays consisting of elements created with the CAutoPtr class.

Example

```
// Copy demonstration
CAtlArray<int> iArrayS, iArrayT;

iArrayS.Add(1);
iArrayS.Add(2);

iArrayT.Add(3);
iArrayT.Add(4);

iArrayT.Copy(iArrayS);

ATLASSERT(iArrayT.GetCount() == 2);
ATLASSERT(iArrayT[0] == 1);
ATLASSERT(iArrayT[1] == 2);
```

See Also

Reference

CAtlArray Class

Other Resources

CAtlArray Members

CAtlArray::FreeExtra

Call this method to remove any empty elements from the array.

```
void FreeExtra( ) throw( );
```

Remarks

Any empty elements are removed, but the size and upper bound of the array remain unchanged.

In debug builds, an ATLASSERT will be raised if the CAtlArray object is not valid, or if the array would exceed its maximum size.

See Also

Reference

CAtlArray Class

CAtlArray::RemoveAll

Other Resources

CAtlArray Members

CAtlArray::GetAt

Call this method to retrieves a single element from the array object.

```
const E& GetAt(
   size_t iElement
) const throw( );
E& GetAt(
   size_t iElement
) throw( );
```

Parameters

iElement

The index value of the array element to return.

Return Value

Returns a reference to the required array element.

Remarks

In debug builds, an ATLASSERT will be raised if *iElement* exceeds the number of elements in the array. In release builds, an invalid argument may lead to unpredictable behavior.

Example

```
// Declare an array of integers

CAtlArray<int> iMyArray;
int element;

// Add ten elements to the array
for (int i=0; i < 10; i++)
   iMyArray.Add(i);

// Use GetAt and SetAt to modify
// every element in the array

for (i=0; i < iMyArray.GetCount(); i++)
{
   element = iMyArray.GetAt(i);
   element *= 10;
   iMyArray.SetAt(i, element);
}
```

See Also

Reference

CAtlArray Class

CAtlArray::SetAt

Other Resources

CAtlArray Members

CAtlArray::GetCount

Call this method to return the number of elements stored in the array.

```
size_t GetCount( ) const throw( );
```

Return Value

Returns the number of elements stored in the array.

Remarks

As the first element in the array is at position 0, the value returned by **GetCount** is always 1 greater than the largest index.

Example

See the example for CAtlArray::GetAt.

See Also

Reference

CAtlArray Class

CAtlArray::GetData

Other Resources

CAtlArray Members

CAtlArray::GetData

Call this method to return a pointer to the first element in the array.

```
E* GetData( ) throw( );
const E* GetData( ) const throw( );
```

Return Value

Returns a pointer to the memory location storing the first element in the array. If no elements are available, NULL is returned.

Example

```
// Define an array of integers
CAtlArray<int> MyArray;

// Define a pointer
int * pData;

// Allocate enough space for 32 elements
// with buffer increase to be calculated
// automatically
MyArray.SetCount(32, -1);

// Set the pointer to the first element
pData = MyArray.GetData();

// Set array values directly
for (int j=0; j < 32; j++, pData++)
   *pData = j*10;
```

See Also

Reference

CAtlArray Class

CAtlArray::GetCount

Other Resources

CAtlArray Members

CAtlArray::InsertArrayAt

Call this method to insert one array into another.

```
void InsertArrayAt(
   size_t iStart,
   const CAtlArray< E, ETraits >* paNew
);
```

Parameters

iStart

The index at which the array is to be inserted.

paNew

The array to be inserted.

Remarks

Elements from the array *paNew* are copied into the array object, beginning at element *iStart*. The existing array elements are moved to avoid being overwritten.

In debug builds, an ATLASSERT will be raised if the **CAtlArray** object is not valid, or if the *paNew* pointer is NULL or invalid.

Note

CAtlArray::InsertArrayAt does not support arrays consisting of elements created with the CAutoPtr class.

Example

```
// Define two integer arrays
CAtlArray<int> iTargetArray, iSourceArray;

// Add elements to first array
for (int x=0; x<10; x++)
   iTargetArray.Add(x);

// Add elements to the second array
for (x=0; x<10; x++)
   iSourceArray.Add(x*10);

// Insert the Source array into the Target
// array, starting at the 5th element.
iTargetArray.InsertArrayAt(5,&iSourceArray);
```

See Also

Reference

CAtlArray Class

CAtlArray::Append

Other Resources

CAtlArray Members

CAtlArray::InsertAt

Call this method to insert a new element (or multiple copies of an element) into the array object.

```
void InsertAt(
   size_t iElement,
   INARGTYPE element,
   size_t nCount = 1
);
```

Parameters

iElement

The index where the element or elements are to be inserted.

element

The value of the element or elements to be inserted.

nCount

The number of elements to add.

Remarks

Inserts one or more elements into the array, starting at index *iElement*. Existing elements are moved to avoid being overwritten.

In debug builds, an ATLASSERT will be raised if the **CAtlArray** object is invalid, the number of elements to be added is zero, or the combined number of elements is too large for the array to contain. In retail builds, passing invalid parameters may cause unpredictable results.

Example

```
// Declare an array of integers
CAtlArray<int> iBuffer;

// Add elements to the array
for (int b=0; b<10; b++)
   iBuffer.Add(0);

// Instert ten 1's into the array
// at position 5
iBuffer.InsertAt(5,1,10);
```

See Also

Reference

CAtlArray Class

CAtlArray::Add

Other Resources

CAtlArray Members

CAtlArray::IsEmpty

Call this method to test if the array is empty.

```
bool IsEmpty( ) const throw( );
```

Return Value

Returns true if the array is empty, false otherwise.

Remarks

The array is said to be empty if it contains no elements. Therefore, even if the array contains empty elements, it is not empty.

Example

```
// Define an array of chars
   CAtlArray<char> cArray;

// Add an element
cArray.Add('a');

// Confirm array is not empty
ATLASSERT(!cArray.IsEmpty());

// Remove all elements
cArray.RemoveAll();

// Confirm array is empty
ATLASSERT(cArray.IsEmpty());
```

See Also

Reference

CAtlArray Class

CAtlArray::AssertValid

Other Resources

CAtlArray Members

CAtlArray::RemoveAll

Call this method to remove all elements from the array object.

```
void RemoveAll( ) throw( );
```

Remarks

Removes all of the elements from the array object.

This method calls CAtlArray::SetCount to resize the array and subsequently frees any allocated memory.

Example

See the example for CAtlArray::IsEmpty.

See Also

Reference

CAtlArray Class
CAtlArray::RemoveAt
CAtlArray::FreeExtra

Other Resources

CAtlArray Members

CAtlArray::RemoveAt

Call this method to remove one or more elements from the array.

```
void RemoveAt(
   size_t iElement,
   size_t nCount = 1
);
```

Parameters

iElement

The index of the first element to remove.

nCount

The number of elements to remove.

Remarks

Removes one or more elements from the array. Any remaining elements are shifted down. The upper bound is decremented, but memory is not freed until a call to CAtlArray::FreeExtra is made.

In debug builds, an ATLASSERT will be raised if the **CAtlArray** object is not valid, or if the combined total of *iElement* and *nCount* exceeds the total number of elements in the array. In retail builds, invalid parameters may cause unpredictable results.

Example

```
// Declare an array of chars
CAtlArray<char> cMyArray;

// Add ten elements to the array
for (int a=0; a < 10; a++)
   cMyArray.Add('*');

// Remove five elements starting with
// the element at position 1
cMyArray.RemoveAt(1,5);

// Free memory
cMyArray.FreeExtra();

// Confirm size of array
ATLASSERT(cMyArray.GetCount() == 5);
```

See Also

Reference

CAtlArray Class

CAtlArray::RemoveAll

Other Resources

CAtlArray Members

CAtlArray::SetAt

Call this method to set the value of an element in the array object.

```
void SetAt(
   size_t iElement,
   INARGTYPE element
);
```

Parameters

iElement

The index pointing to the array element to set.

element

The new value of the specified element.

Remarks

In debug builds, an ATLASSERT will be raised if *iElement* exceeds the number of elements in the array. In retail builds, an invalid parameter may result in unpredictable results.

Example

See the example for CAtlArray::GetAt.

See Also

Reference

CAtlArray Class
CAtlArray::GetAt
CAtlArray::SetAtGrow

Other Resources

CAtlArray Members

CAtlArray::SetCount

Call this method to set the size of the array object.

```
bool SetCount(
   size_t nNewSize,
   int nGrowBy = - 1
);
```

Parameters

nNewSize

The required size of the array.

nGrowBy

A value used to determine how large to make the buffer. A value of -1 causes an internally calculated value to be used.

Return Value

Returns true if the array is successfully resized, false otherwise.

Remarks

The array can be increased or decreased in size. If increased, extra empty elements are added to the array. If decreased, the elements with the largest indices will be deleted and memory freed.

Use this method to set the size of the array before using it. If **SetCount** is not used, the process of adding elements — and the subsequent memory allocation performed — will reduce performance and fragment memory.

Example

See the example for CAtlArray::GetData.

See Also

Reference

CAtlArray Class

Other Resources

CAtlArray Members

CAtlArray::SetAtGrow

Call this method to set the value of an element in the array object, expanding the array as required.

```
void SetAtGrow(
   size_t iElement,
   INARGTYPE element
);
```

Parameters

iElement

The index pointing to the array element to set.

element

The new value of the specified element.

Remarks

Replaces the value of the element pointed to by the index. If *iElement* is larger than the current size of the array, the array is automatically increased using a call to CAtlArray::SetCount. In debug builds, an ATLASSERT will be raised if the **CAtlArray** object is not valid. In retail builds, invalid parameters may cause unpredictable results.

Example

```
// Declare an array of integers
CAtlArray<int> iGrowArray;

// Add an element
iGrowArray.Add(0);

// Add an extra element at position 19.
// This will grow the array to accommodate.
iGrowArray.SetAtGrow(19,0);

// Confirm size of new array
ATLASSERT(iGrowArray.GetCount() == 20);

// Note: the values at position 1 to 18
// are undefined.
```

See Also

Reference

CAtlArray Class

CAtlArray::SetAt

Other Resources

CAtlArray Members

CAtlArray Typedefs

For information about the typedefs in **CAtlArray**, see CAtlArray Members.

CAtlArray Operators

For information about the operators in **CAtlArray**, see CAtlArray Members.

CAtlArray::operator []

Call this operator to return a reference to an element in the array.

```
E& operator[](
   size_t iElement
) throw( );
const E& operator[](
   size_t iElement
) const throw( );
```

Parameters

iElement

The index value of the array element to return.

Return Value

Returns a reference to the required array element.

Remarks

Performs a similar function to CAtlArray::GetAt. Unlike the MFC class CArray, this operator cannot be used as a substitute for CAtlArray::SetAt.

In debug builds, an ATLASSERT will be raised if *iElement* exceeds the total number of elements in the array. In retail builds, an invalid parameter may cause unpredictable results.

See Also

Reference

CAtlArray Class

Other Resources

CAtlArray Members

CAtlArray Typedefs

For information about the typedefs in **CAtlArray**, see CAtlArray Members.

CAtlArray::INARGTYPE

The data type to use for adding elements to the array.

```
typedef ETraits::INARGTYPE INARGTYPE;
```

See Also

Reference

CAtlArray Class

CAtlArray::OUTARGTYPE

Other Resources

CAtlArray Members

CAtlArray::OUTARGTYPE

The data type to use for retrieving elements from the array.

```
typedef ETraits::OUTARGTYPE OUTARGTYPE;
```

See Also

Reference

CAtlArray Class

CAtlArray::INARGTYPE

Other Resources

CAtlArray Members

CAtlAutoThreadModule Class

This class implements a thread-pooled, apartment-model COM server.

```
class CAtlAutoThreadModule :
   public CAtlAutoThreadModuleT< CAtlAutoThreadModule >
```

Remarks

CAtlAutoThreadModule derives from CAtlAutoThreadModuleT and implements a thread-pooled, apartment-model COM server. **CAtlAutoThreadModule** uses CComApartment to manage an apartment for each thread in the module.

You must use the DECLARE_CLASSFACTORY_AUTO_THREAD macro in your object's class definition to specify CComClassFactoryAutoThread as the class factory. You should then add a single instance of a class derived from **CAtlAutoThreadModuleT** such as **CAtlAutoThreadModule**. For example:

```
CAtlAutoThreadModule _AtlAutoModule; // name is immaterial.
```

Note
This class replaces the obsolete CComAutoThreadModule class.

Requirements

Header: atlbase.h

See Also

Reference

CAtlAutoThreadModuleT Class
IAtlAutoThreadModule Class
ATL Module Classes

Other Resources

ATL Class Overview

CAtlAutoThreadModuleT Class

This class provides methods for implementing a thread-pooled, apartment-model COM server.

```
template <
   class T,
   class ThreadAllocator = CComSimpleThreadAllocator,
   DWORD dwWait = INFINITE
>
class ATL_NO_VTABLE CAtlAutoThreadModuleT :
   public IAtlAutoThreadModule
```

Parameters

T

The class which will implement the COM server.

ThreadAllocator

The class managing thread selection. The default value is CComSimpleThreadAllocator.

dwWait

Specifies the time-out interval, in milliseconds. The default is INFINITE, which means the method's time-out interval never elapses.

Remarks

The class CAtlAutoThreadModule derives from **CAtlAutoThreadModuleT** in order to implement a thread-pooled, apartment-model COM server. It replaces the obsolete class CComAutoThreadModule.

Note
This class should not be used in a DLL, as the default *dwWait* value of INFINITE will cause a deadlock when the DLL is unloaded.

Requirements

Header: atlbase.h

See Also

Reference

IAtlAutoThreadModule Class

IAtlAutoThreadModule Class

ATL Module Classes

Other Resources

CAtlAutoThreadModuleT Members

ATL Class Overview

CAtlAutoThreadModuleT Members

Static Functions

GetDefaultThreads	This static function dynamically calculates and returns the maximum number of threads for the EXE module, based on the number of processors.

See Also

Reference

CAtlAutoThreadModuleT Class

CAtlAutoThreadModuleT Static Functions

For information about the static functions in **CAtlAutoThreadModuleT**, see CAtlAutoThreadModuleT Members.

CAtlAutoThreadModuleT::GetDefaultThreads

This static function dynamically calculates and returns the maximum number of threads for the EXE module, based on the number of processors.

```
static int GetDefaultThreads( );
```

Return Value

The number of threads to be created in the EXE module.

Remarks

Override this method if you want to use a different method for calculating the number of threads. By default, the number of threads is based on the number of processors.

See Also

Reference

CAtlAutoThreadModuleT Class

Other Resources

CAtlAutoThreadModuleT Members

CAtlBaseModule Class

This class is instantiated in every ATL project.

```
class CAtlBaseModule :
   public _ATL_BASE_MODULE
```

Remarks

An instance of **CAtlBaseModule** named _AtlBaseModule is present in every ATL project, containing a handle to the module instance, a handle to the module containing resources (which by default, are one and the same), and an array of handles to modules providing primary resources. **CAtlBaseModule** can be safely accessed from multiple threads.

This class replaces the obsolete CComModule class used in earlier versions of ATL.

Requirements

Header: atlcore.h

See Also

Reference

ATL Module Classes

Other Resources

CAtlBaseModule Members

ATL Class Overview

CAtlBaseModule Members

Methods

AddResourceInstance	Adds a resource instance to the list of stored handles.
CAtlBaseModule	The constructor.
GetHInstanceAt	Returns a handle to a specified resource instance.
GetModuleInstance	Returns the module instance from a **CAtlBaseModule** object.
GetResourceInstance	Returns the resource instance from a **CAtlBaseModule** object.
RemoveResourceInstance	Removes a resource instance from the list of stored handles.
SetResourceInstance	Sets the resource instance of a **CAtlBaseModule** object.

Data Members

m_bInitFailed	A variable that indicates if the module initialization has failed.

See Also

Reference

CAtlBaseModule Class

CAtlBaseModule Methods

For information about the methods in **CAtlBaseModule**, see CAtlBaseModule Members.

CAtlBaseModule::AddResourceInstance

Adds a resource instance to the list of stored handles.

```
bool AddResourceInstance(
   HINSTANCE hInst
) throw( );
```

Parameters

hInst

The resource instance to add.

Return Value

Returns true if the resource was successfully added, false otherwise.

See Also

Reference

CAtlBaseModule Class

CAtlBaseModule::RemoveResourceInstance

Other Resources

CAtlBaseModule Members

CAtlBaseModule::CAtlBaseModule

The constructor.

```
CAtlBaseModule( ) throw( );
```

Remarks

Creates the **CAtlBaseModule**.

See Also

Reference

CAtlBaseModule Class

Other Resources

CAtlBaseModule Members

CAtlBaseModule::GetHInstanceAt

Returns a handle to a specified resource instance.

```
HINSTANCE GetHInstanceAt(
   int i
) throw( );
```

Parameters

i

The number of the resource instance.

Return Value

Returns the handle to the resource instance, or NULL if no corresponding resource instance exists.

See Also

Reference

CAtlBaseModule Class

CAtlBaseModule::AddResourceInstance

Other Resources

CAtlBaseModule Members

CAtlBaseModule::GetModuleInstance

Returns the module instance from a **CAtlBaseModule** object.

```
HINSTANCE GetModuleInstance( ) throw( );
```

Return Value

Returns the module instance.

See Also

Reference

CAtlBaseModule Class
CAtlBaseModule::GetResourceInstance
CAtlBaseModule::GetHInstanceAt

Other Resources

CAtlBaseModule Members

CAtlBaseModule::GetResourceInstance

Returns the resource instance.

```
HINSTANCE GetResourceInstance( ) throw( );
```

Return Value

Returns the resource instance.

See Also

Reference

CAtlBaseModule Class
CAtlBaseModule::GetModuleInstance
CAtlBaseModule::SetResourceInstance

Other Resources

CAtlBaseModule Members

CAtlBaseModule::RemoveResourceInstance

Removes a resource instance from the list of stored handles.

```
bool RemoveResourceInstance(
   HINSTANCE hInst
) throw( );
```

Parameters

hInst

The resource instance to remove.

Return Value

Returns true if the resource was successfully removed, false otherwise.

See Also

Reference

CAtlBaseModule Class

CAtlBaseModule::AddResourceInstance

Other Resources

CAtlBaseModule Members

CAtlBaseModule::SetResourceInstance

Sets the resource instance of a **CAtlBaseModule** object.

```
HINSTANCE SetResourceInstance(
   HINSTANCE hInst
) throw( );
```

Parameters

hInst

The new resource instance.

Return Value

Returns the updated resource instance.

See Also

Reference

CAtlBaseModule Class

CAtlBaseModule::GetResourceInstance

Other Resources

CAtlBaseModule Members

CAtlBaseModule Data Members

For information about the data members in **CAtlBaseModule**, see CAtlBaseModule Members.

CAtlBaseModule::m_bInitFailed

A variable that indicates if the module initialization has failed.

```
static bool m_bInitFailed;
```

Remarks

True if the module initialized, false if it failed to initialize.

See Also

Reference

CAtlBaseModule Class

Other Resources

CAtlBaseModule Members

CAtlComModule Class

This class implements a COM server module.

```
class CAtlComModule :
   public _ATL_COM_MODULE
```

Remarks

CAtlComModule implements a COM server module, allowing a client to access the module's components.

This class replaces the obsolete CComModule class used in earlier versions of ATL. See ATL Module Classes for more details.

Requirements

Header: atlbase.h

See Also

Reference

_ATL_COM_MODULE

Other Resources

CAtlComModule Members

ATL Class Overview

CAtlComModule Members

Methods

CAtlComModule	The constructor.
~CAtlComModule	The destructor.
RegisterServer	Call this method to update the system registry for each object in the object map.
RegisterTypeLib	Call this method to register a type library.
UnregisterServer	Call this method to unregister each object in the object map.
UnRegisterTypeLib	Call this method to unregister a type library.

See Also

Reference

CAtlComModule Class

CAtlComModule Methods

For information about the methods in **CAtlComModule**, see CAtlComModule Members.

CAtlComModule::CAtlComModule

The constructor.

```
CAtlComModule( ) throw( );
```

Remarks

Initializes the module.

See Also

Reference

CAtlComModule Class

CAtlComModule::~CAtlComModule

Other Resources

CAtlComModule Members

CAtlComModule::~CAtlComModule

The destructor.

```
~CAtlComModule( );
```

Remarks

Frees all class factories.

See Also

Reference

CAtlComModule Class

CAtlComModule::CAtlComModule

Other Resources

CAtlComModule Members

CAtlComModule::RegisterServer

Call this method to update the system registry for each object in the object map.

```
HRESULT RegisterServer(
   BOOL bRegTypeLib = FALSE,
   const CLSID* pCLSID = NULL
);
```

Parameters

bRegTypeLib

TRUE if the type library is to be registered. The default value is FALSE.

pCLSID

Points to the CLSID of the object to be registered. If NULL (the default value), all objects in the object map will be registered.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls the global function AtlComModuleRegisterServer.

See Also

Reference

CAtlComModule Class

CAtlComModule::UnregisterServer

Other Resources

CAtlComModule Members

CAtlComModule::RegisterTypeLib

Call this method to register a type library.

```
HRESULT RegisterTypeLib(
   LPCTSTR lpszIndex
);
HRESULT RegisterTypeLib( );
```

Parameters

lpszIndex

String in the format "\\N", where N is the integer index of the TYPELIB resource.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Adds information about a type library to the system registry. If the module instance contains multiple type libraries, use the first version of this method to specify which type library should be used.

See Also

Reference

CAtlComModule Class

CAtlComModule::UnRegisterTypeLib

Other Resources

CAtlComModule Members

CAtlComModule::UnregisterServer

Call this method to unregister each object in the object map.

```
HRESULT UnregisterServer(
   BOOL bRegTypeLib = FALSE,
   const CLSID* pCLSID = NULL
);
```

Parameters

bRegTypeLib

TRUE if the type library is to be unregistered. The default value is FALSE.

pCLSID

Points to the CLSID of the object to be unregistered. If NULL (the default value), all objects in the object map will be unregistered.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls the global function AtlComModuleUnregisterServer.

See Also

Reference

CAtlComModule Class

CAtlComModule::RegisterServer

Other Resources

CAtlComModule Members

CAtlComModule::UnRegisterTypeLib

Call this method to unregister a type library.

```
HRESULT UnRegisterTypeLib(
   LPCTSTR lpszIndex
);
HRESULT UnRegisterTypeLib( );
```

Parameters

lpszIndex

String in the format "\\N", where N is the integer index of the TYPELIB resource.

Remarks

Removes information about a type library from the system registry. If the module instance contains multiple type libraries, use the first version of this method to specify which type library should be used.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CAtlComModule Class

CAtlComModule::RegisterTypeLib

Other Resources

CAtlComModule Members

CAtlDebugInterfacesModule Class

This class provides support for debugging interfaces.

```
class CAtlDebugInterfacesModule
```

Remarks

CAtlDebugInterfacesModule provides the support required for debugging interfaces. It is included in any project that defines the symbol _ATL_DEBUG_QI.

Requirements

Header: atlbase.h

See Also

Tasks

How to: Debug QueryInterface Calls

Reference

ATL Module Classes

Other Resources

ATL Debugging Techniques

ATL Class Overview

CAtlDllModuleT Class

This class represents the module for a DLL.

```
template <
   class T
>
class ATL_NO_VTABLE CAtlDllModuleT :
   public CAtlModuleT< T >
```

Parameters

T

Your class derived from **CAtlDllModuleT**.

Remarks

CAtlDllModuleT represents the module for a dynamic-link library (DLL) and provides functions used by all DLL projects. This specialization of CAtlModuleT class includes support for registration.

For more information on modules in ATL, see ATL Module Classes.

Requirements

Header: atlbase.h

See Also

Reference

CAtlModuleT Class
CAtlExeModuleT Class
ATL Module Classes

Other Resources

CAtlDllModuleT Members
ATL Class Overview

CAtlDllModuleT Members

Methods

CAtlDllModuleT	The constructor.
~CAtlDllModuleT	The destructor.
DllCanUnloadNow	Tests if the DLL can be unloaded.
DllGetClassObject	Returns a class factory.
DllMain	The optional entry point into a dynamic-link library (DLL).
DllRegisterServer	Adds entries to the system registry for objects in the DLL.
DllUnregisterServer	Removes entries in the system registry for objects in the DLL.
GetClassObject	Returns a class factory. Invoked by DllGetClassObject.

See Also

Reference

CAtlDllModuleT Class

CAtlDllModuleT Methods

For information about the methods in **CAtlDllModuleT**, see CAtlDllModuleT Members.

CAtlDllModuleT::CAtlDllModuleT

The constructor.

```
CAtlDllModuleT( ) throw( );
```

See Also

Reference

CAtlDllModuleT Class

CAtlDllModuleT::~CAtlDllModuleT

Other Resources

CAtlDllModuleT Members

CAtlDllModuleT::~CAtlDllModuleT

The destructor.

```
~CAtlDllModuleT( ) throw( );
```

See Also

Reference

CAtlDllModuleT Class

CAtlDllModuleT::CAtlDllModuleT

Other Resources

CAtlDllModuleT Members

CAtlDllModuleT::DllCanUnloadNow

Tests if the DLL can be unloaded.

```
HRESULT DllCanUnloadNow( ) throw( );
```

Return Value

Returns S_OK if the DLL can be unloaded, or S_FALSE if it cannot.

See Also

Reference

CAtlDllModuleT Class

Other Resources

CAtlDllModuleT Members

CAtlDllModuleT::DllGetClassObject

Returns the class factory.

```
HRESULT DllGetClassObject(
   REFCLSID rclsid,
   REFIID riid,
   LPVOID* ppv
) throw( );
```

Parameters

rclsid

The CLSID of the object to be created.

riid

The IID of the requested interface.

ppv

A pointer to the interface pointer identified by *riid*. If the object does not support this interface, *ppv* is set to NULL.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CAtlDllModuleT Class

Other Resources

CAtlDllModuleT Members

CAtlDllModuleT::DllMain

The optional entry point into a dynamic-link library (DLL).

```
BOOL WINAPI DllMain(
   DWORD dwReason,
   LPVOID /*lpReserved*/
) throw( );
```

Parameters

dwReason

If set to DLL_PROCESS_ATTACH, the DLL_THREAD_ATTACH and DLL_THREAD_DETACH notification calls are disabled.

lpReserved

Reserved.

Return Value

Always returns TRUE.

Remarks

Disabling the DLL_THREAD_ATTACH and DLL_THREAD_DETACH notification calls can be a useful optimization for multithreaded applications that have many DLLs, that frequently create and delete threads, and whose DLLs do not need these thread-level notifications of attachment/detachment.

See Also

Reference

CAtlDllModuleT Class
DisableThreadLibraryCalls
DllMain

Other Resources

CAtlDllModuleT Members

CAtlDllModuleT::DllRegisterServer

Adds entries to the system registry for objects in the DLL.

```
HRESULT DllRegisterServer(
   BOOL bRegTypeLib = TRUE
) throw( );
```

Parameters

bRegTypeLib

TRUE if the type library is to be registered. The default value is TRUE.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CAtlDllModuleT Class

CAtlDllModuleT::DllUnregisterServer

Other Resources

CAtlDllModuleT Members

CAtlDllModuleT::DllUnregisterServer

Removes entries in the system registry for objects in the DLL.

```
HRESULT DllUnregisterServer(
   BOOL bUnRegTypeLib = TRUE
) throw( );
```

Parameters

bUnRegTypeLib

TRUE if the type library is to be removed from the registry. The default value is TRUE.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CAtlDllModuleT Class

CAtlDllModuleT::DllRegisterServer

Other Resources

CAtlDllModuleT Members

CAtlDllModuleT::GetClassObject

Creates an object of the specified CLSID.

```
HRESULT GetClassObject(
   REFCLSID rclsid,
   REFIID riid,
   LPVOID* ppv
) throw( );
```

Parameters

rclsid

The CLSID of the object to be created.

riid

The IID of the requested interface.

ppv

A pointer to the interface pointer identified by *riid*. If the object does not support this interface, *ppv* is set to NULL.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method is called by CAtlDllModuleT::DllGetClassObject and is included for backward compatibility.

See Also

Reference

CAtlDllModuleT Class

Other Resources

CAtlDllModuleT Members

CAtlException Class

This class defines an ATL exception.

```
class CAtlException
```

Remarks

A **CAtlException** object represents an exception condition related to an ATL operation. The **CAtlException** class includes a public data member that stores the status code indicating the reason for the exception and a cast operator that allows you to treat the exception as if it were an HRESULT.

In general, you will call **AtlThrow** rather than creating a **CAtlException** object directly.

Requirements

Header: atlexcept.h

See Also

Reference

AtlThrow

Other Resources

CAtlException Members

ATL Class Overview

CAtlException Members

Methods

CAtlException	The constructor.

Operators

operator HRESULT	Casts the current object to an HRESULT value.

Data Members

m_hr	The variable of type HRESULT created by the object and used to store the error condition.

See Also

Reference

CAtlException Class

CAtlException Methods

For information about the methods in **CAtlException**, see CAtlException Members.

CAtlException::CAtlException

The constructor.

```
CAtlException(
   HRESULT hr
) throw( );
CAtlException( ) throw( );
```

Parameters

hr

The **HRESULT** error code.

See Also

Reference

CAtlException Class

AtlThrow

Other Resources

CAtlException Members

CAtlException Operators

For information about the operators in **CAtlException**, see CAtlException Members.

CAtlException::operator HRESULT

Casts the current object to an HRESULT value.

```
operator HRESULT( ) const throw ( );
```

See Also

Reference

CAtlException Class

AtlThrow

Other Resources

CAtlException Members

CAtlException Data Members

For information about the data members in **CAtlException**, see CAtlException Members.

CAtlException::m_hr

The **HRESULT** data member.

```
HRESULT m_hr;
```

Remarks

The data member that stores the error condition. The HRESULT value is set by the constructor, CAtlException::CAtlException.

See Also

Reference

CAtlException Class

AtlThrow

Other Resources

CAtlException Members

CAtlExeModuleT Class

This class represents the module for an application.

```
template <
   class T
>
class ATL_NO_VTABLE CAtlExeModuleT :
   public CAtlModuleT< T >
```

Parameters

T

Your class derived from **CAtlExeModuleT**.

Remarks

CAtlExeModuleT represents the module for an application (EXE) and contains code that supports creating an EXE, processing the command line, registering class objects, running the message loop, and cleaning up on exit.

This class is designed to improve performance when COM objects in the EXE server are continually created and destroyed. After the last COM object is released, the EXE waits for a duration specified by the CAtlExeModuleT::m_dwTimeOut data member. If there is no activity during this period (that is, no COM objects are created), the shutdown process is initiated.

The CAtlExeModuleT::m_bDelayShutdown data member is a flag used to determine if the EXE should use the mechanism defined above. If it is set to false, then the module will terminate immediately.

For more information on modules in ATL, see ATL Module Classes.

Requirements

Header: atlbase.h

See Also

Tasks

ATLDuck Sample: Uses Connection Points with ATL

Reference

CAtlModuleT Class

CAtlDllModuleT Class

Other Resources

CAtlExeModuleT Members

ATL Class Overview

CAtlExeModuleT Members

Methods

CAtlExeModuleT	The constructor.
~CAtlExeModuleT	The destructor.
ParseCommandLine	Parses the command line and performs registration if necessary.
PostMessageLoop	This method is called immediately after the message loop exits.
PreMessageLoop	This method is called immediately before entering the message loop.
RegisterClassObjects	Registers the class object.
RevokeClassObjects	Revokes the class object.
Run	This method executes code in the EXE module to initialize, run the message loop, and clean up.
RunMessageLoop	This method executes the message loop.
Unlock	Decrements the module's lock count.
WinMain	This method implements the code required to run an EXE.

Static Functions

InitializeCom	Initializes COM.
UninitializeCom	Uninitializes COM.

Data Members

m_bDelayShutdown	A flag indicating that there should be a delay shutting down the module.
m_dwPause	A pause value used to ensure all objects are released before shutdown.
m_dwTimeOut	A time-out value used to delay the unloading of the module.

See Also

Reference

CAtlExeModuleT Class

Other Resources

CAtlModuleT Members

CAtlExeModuleT Methods

For information about the methods in **CAtlExeModuleT**, see CAtlExeModuleT Members.

CAtlExeModuleT::CAtlExeModuleT

The constructor.

```
CAtlExeModuleT( ) throw( );
```

Remarks

If the EXE module could not be initialized, WinMain will immediately return without further processing.

See Also

Reference

CAtlExeModuleT Class

CAtlExeModuleT::~CAtlExeModuleT

Other Resources

CAtlExeModuleT Members

CAtlExeModuleT::~CAtlExeModuleT

The destructor.

```
~CAtlExeModuleT( ) throw( );
```

Remarks

Frees all allocated resources.

See Also

Reference

CAtlExeModuleT Class

CAtlExeModuleT::CAtlExeModuleT

Other Resources

CAtlExeModuleT Members

CAtlExeModuleT::ParseCommandLine

Parses the command line and performs registration if necessary.

```
bool ParseCommandLine(
   LPCTSTR lpCmdLine,
   HRESULT* pnRetCode
) throw( );
```

Parameters

lpCmdLine

The command line passed to the application.

pnRetCode

The HRESULT corresponding to the registration (if it took place).

Return Value

Return true if the application should continue to run, otherwise false.

Remarks

This method is called from CAtlExeModuleT::WinMain and can be overridden to handle command-line switches. The default implementation checks for **/RegServer** and **/UnRegServer** command-line arguments and performs registration or unregistration.

See Also

Reference

CAtlExeModuleT Class

Other Resources

CAtlExeModuleT Members

CAtlExeModuleT::PostMessageLoop

This method is called immediately after the message loop exits.

```
HRESULT PostMessageLoop( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Override this method to perform custom application cleanup. The default implementation calls CAtlExeModuleT::RevokeClassObjects.

See Also

Reference

CAtlExeModuleT Class

CAtlExeModuleT::PreMessageLoop

Other Resources

CAtlExeModuleT Members

CAtlExeModuleT::PreMessageLoop

This method is called immediately before entering the message loop.

```
HRESULT PreMessageLoop(
   int nShowCmd
) throw( );
```

Parameters

nShowCmd

The value passed as the *nShowCmd* parameter in WinMain.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Override this method to add custom initialization code for the application. The default implementation registers the class objects.

See Also

Reference

CAtlExeModuleT Class
CAtlExeModuleT::PostMessageLoop
CAtlExeModuleT::WinMain

Other Resources

CAtlExeModuleT Members

CAtlExeModuleT::RegisterClassObjects

Registers the class object with OLE so other applications can connect to it.

```
HRESULT RegisterClassObjects(
   DWORD dwClsContext,
   DWORD dwFlags
) throw( );
```

Parameters

dwClsContext

Specifies the context in which the class object is to be run. Possible values are CLSCTX_INPROC_SERVER, CLSCTX_INPROC_HANDLER, or CLSCTX_LOCAL_SERVER.

dwFlags

Determines the connection types to the class object. Possible values are REGCLS_SINGLEUSE, REGCLS_MULTIPLEUSE, or REGCLS_MULTI_SEPARATE.

Return Value

Returns S_OK on success, S_FALSE if there were no classes to register, or an error HRESULT on failure.

See Also

Reference

CAtlExeModuleT Class
CLSCTX
REGCLS
CAtlExeModuleT::RevokeClassObjects

Other Resources

CAtlExeModuleT Members

CAtlExeModuleT::RevokeClassObjects

Removes the class object.

```
HRESULT RevokeClassObjects( ) throw( );
```

Return Value

Returns S_OK on success, S_FALSE if there were no classes to register, or an error HRESULT on failure.

See Also

Reference

CAtlExeModuleT Class

CAtlExeModuleT::RegisterClassObjects

Other Resources

CAtlExeModuleT Members

CAtlExeModuleT::Run

This method executes code in the EXE module to initialize, run the message loop, and clean up.

```
HRESULT Run(
   int nShowCmd = SW_HIDE
) throw( );
```

Parameters

nShowCmd

Specifies how the window is to be shown. This parameter can be one of the values discussed in the WinMain section. Defaults to SW_HIDE.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method can be overridden. However, in practice is it better to override CAtlExeModuleT::PreMessageLoop, CAtlExeModuleT::RunMessageLoop, or CAtlExeModuleT::PostMessageLoop instead.

See Also

Reference

CAtlExeModuleT Class

Other Resources

CAtlExeModuleT Members

CAtlExeModuleT::RunMessageLoop

This method executes the message loop.

```
void RunMessageLoop( ) throw( );
```

Remarks

This method can be overridden to change the behavior of the message loop.

See Also

Reference

CAtlExeModuleT Class

Other Resources

CAtlExeModuleT Members

CAtlExeModuleT::Unlock

Decrements the module's lock count.

```
LONG Unlock( ) throw( );
```

Return Value

Returns a value which may be useful for diagnostics or testing.

See Also

Reference

CAtlExeModuleT Class

Other Resources

CAtlExeModuleT Members

CAtlExeModuleT::WinMain

This method implements the code required to run an EXE.

```
int WinMain(
   int nShowCmd
) throw( );
```

Parameters

nShowCmd

Specifies how the window is to be shown. This parameter can be one of the values discussed in the WinMain section.

Return Value

Returns the executable's return value.

Remarks

This method can be overridden. If overriding CAtlExeModuleT::PreMessageLoop, CAtlExeModuleT::PostMessageLoop, or CAtlExeModuleT::RunMessageLoop doesn't provide enough flexibility, it's possible to override the **WinMain** function using this method.

See Also

Reference

CAtlExeModuleT Class

WinMain

Other Resources

CAtlExeModuleT Members

CAtlExeModuleT Data Members

For information about the data members in **CAtlExeModuleT**, see CAtlExeModuleT Members.

CAtlExeModuleT::m_bDelayShutdown

A flag indicating that there should be a delay shutting down the module.

```
bool m_bDelayShutdown;
```

Remarks

See the CAtlExeModuleT Overview for details.

See Also

Reference

CAtlExeModuleT Class

Other Resources

CAtlExeModuleT Members

CAtlExeModuleT::m_dwPause

A pause value used to ensure all objects are gone before shutdown.

```
DWORD m_dwPause;
```

Remarks

Change this value after calling CAtlExeModuleT::InitializeCom to set the number of milliseconds used as the pause value for shutting down the server. The default value is 1000 milliseconds.

See Also

Reference

CAtlExeModuleT Class

Other Resources

CAtlExeModuleT Members

CAtlExeModuleT::m_dwTimeOut

A time-out value used to delay the unloading of the module.

```
DWORD m_dwTimeOut;
```

Remarks

Change this value after calling CAtlExeModuleT::InitializeCom to define the number of milliseconds used as the time-out value for shutting down the server. The default value is 5000 milliseconds. See the CAtlExeModuleT Overview for more details.

See Also

Reference

CAtlExeModuleT Class

Other Resources

CAtlExeModuleT Members

CAtlExeModuleT Static Functions

For information about the static functions in **CAtlExeModuleT**, see CAtlExeModuleT Members.

CAtlExeModuleT::InitializeCom

Initializes COM.

```
static HRESULT InitializeCom( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method is called from the constructor and can be overridden to initialize COM in a manner different from the default implementation. The default implementation either calls **CoInitializeEx(NULL, COINIT_MULTITHREADED)** or **CoInitialize(NULL)** depending on the project configuration.

Overriding this method normally requires overriding CAtlExeModuleT::UninitializeCom.

See Also

Reference

CAtlExeModuleT Class
CoInitialize
CoInitializeEx

Other Resources

CAtlExeModuleT Members

CAtlExeModuleT::UninitializeCom

Uninitializes COM.

```
static void UninitializeCom( ) throw( );
```

Remarks

By default this method simply calls CoUninitialize and is called from the destructor. Override this method if you override CAtlExeModuleT::InitializeCom.

See Also

Reference

CAtlExeModuleT Class

Other Resources

CAtlExeModuleT Members

CAtlFile Class

This class provides a thin wrapper around the Windows file-handling API.

```
class CAtlFile : public CHandle
```

Remarks

Use this class when file-handling needs are relatively simple, but more abstraction than the Windows API provides is required, without including MFC dependencies.

Requirements

Header: atlfile.h

Example

See the following samples:

- ISAPIFilter Sample
- MantaWeb Sample
- RegExp Sample
- ShowImage Sample
- SOAPDebugApp Sample

See Also

Tasks

Marquee Sample: Provides Performance Monitoring Data Through Scripting

MantaWeb Sample: Online Personal Information Management

Reference

CHandle Class

Other Resources

CAtlFile Members

ATL Class Overview

CAtlFile Members

Methods

CAtlFile	The constructor.
Create	Call this method to create or open a file.
Flush	Call this method to clear the buffers for the file and cause all buffered data to be written to the file.
GetOverlappedResult	Call this method to get the results of an overlapped operation on the file.
GetPosition	Call this method to get the current file pointer position from the file.
GetSize	Call this method to get the size in bytes of the file.
LockRange	Call this method to lock a region in the file to prevent other processes from accessing it.
Read	Call this method to read data from a file starting at the position indicated by the file pointer.
Seek	Call this method to move the file pointer of the file.
SetSize	Call this method to set the size of the file.
UnlockRange	Call this method to unlock a region of the file.
Write	Call this method to write data to the file starting at the position indicated by the file pointer.

See Also

Reference

CAtlFile Class

CAtlFile Methods

For information about the methods in **CAtlFile**, see CAtlFile Members.

CAtlFile::CAtlFile

The constructor.

```
CAtlFile( ) throw( );
CAtlFile(
   CAtlFile& file
) throw( );
explicit CAtlFile(
   HANDLE hFile
) throw( );
```

Parameters

file

The file object.

hFile

The file handle.

Remarks

The copy constructor transfers ownership of the file handle from the original **CAtlFile** object to the newly constructed object.

See Also

Reference

CAtlFile Class

Other Resources

CAtlFile Members

CAtlFile::Create

Call this method to create or open a file.

```
HRESULT Create(
   LPCTSTR szFilename,
   DWORD dwDesiredAccess,
   DWORD dwShareMode,
   DWORD dwCreationDisposition,
   DWORD dwFlagsAndAttributes = FILE_ATTRIBUTE_NORMAL,
   LPSECURITY_ATTRIBUTES lpsa = NULL,
   HANDLE hTemplateFile = NULL
) throw( );
```

Parameters

szFilename

The file name.

dwDesiredAccess

The desired access. See *dwDesiredAccess* in CreateFile in the *Platform SDK*.

dwShareMode

The share mode. See *dwShareMode* in **CreateFile**.

dwCreationDisposition

The creation disposition. See *dwCreationDisposition* in **CreateFile**.

dwFlagsAndAttributes

The flags and attributes. See *dwFlagsAndAttributes* in **CreateFile**.

lpsa

The security attributes. See *lpSecurityAttributes* in **CreateFile**.

hTemplateFile

The template file. See *hTemplateFile* in **CreateFile**.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Calls CreateFile to create or open the file.

Example

See the following samples:

- ISAPIFilter Sample
- MantaWeb Sample
- RegExp Sample
- ShowImage Sample

See Also

Reference

CAtlFile Class

Other Resources

CAtlFile Members

CAtlFile::Flush

Call this method to clear the buffers for the file and cause all buffered data to be written to the file.

```
HRESULT Flush( ) throw( );
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Calls FlushFileBuffers to flush buffered data to the file.

See Also

Reference

CAtlFile Class

Other Resources

CAtlFile Members

CAtlFile::GetOverlappedResult

Call this method to get the results of an overlapped operation on the file.

```
HRESULT GetOverlappedResult(
   LPOVERLAPPED pOverlapped,
   DWORD& dwBytesTransferred,
   BOOL bWait
) throw( );
```

Parameters

pOverlapped

The overlapped structure. See *lpOverlapped* in GetOverlappedResult in the *Platform SDK*.

dwBytesTransferred

The bytes transferred. See *lpNumberOfBytesTransferred* in **GetOverlappedResult**.

bWait

The wait option. See *bWait* in **GetOverlappedResult**.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Calls GetOverlappedResult to get the results of an overlapped operation on the file.

See Also

Reference

CAtlFile Class

Other Resources

CAtlFile Members

CAtlFile::GetPosition

Call this method to get the current file pointer position.

```
HRESULT GetPosition(
   ULONGLONG& nPos
) const throw( );
```

Parameters

nPos

The position in bytes.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Calls SetFilePointer to get the current file pointer position.

See Also

Reference

CAtlFile Class

CAtlFile::GetSize

Other Resources

CAtlFile Members

CAtlFile::GetSize

Call this method to get the size in bytes of the file.

```
HRESULT GetSize(
   ULONGLONG& nLen
) const throw( );
```

Parameters

nLen

The number of bytes in the file.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Calls GetFileSize to get the size in bytes of the file.

Example

See the ShowImage Sample and the ISAPIFilter Sample.

See Also

Reference

CAtlFile Class
CAtlFile::GetPosition
CAtlFile::SetSize

Other Resources

CAtlFile Members

CAtlFile::LockRange

Call this method to lock a region in the file to prevent other processes from accessing it.

```
HRESULT LockRange(
   ULONGLONG nPos,
   ULONGLONG nCount
) throw( );
```

Parameters

nPos

The position in the file where the lock should begin.

nCount

The length of the byte range to be locked.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Calls LockFile to lock a region in the file. Locking bytes in a file prevents access to those bytes by other processes. You can lock more than one region of a file, but no overlapping regions are allowed. When you unlock a region, using CAtlFile::UnlockRange, the byte range must correspond exactly to the region that was previously locked. **LockRange** does not merge adjacent regions; if two locked regions are adjacent, you must unlock each separately.

See Also

Reference

CAtlFile Class

CAtlFile::UnlockRange

Other Resources

CAtlFile Members

CAtlFile::Read

Call this method to read data from a file starting at the position indicated by the file pointer.

```
HRESULT Read(
   LPVOID pBuffer,
   DWORD nBufSize
) throw( );
HRESULT Read(
   LPVOID pBuffer,
   DWORD nBufSize,
   DWORD& nBytesRead
) throw( );
HRESULT Read(
   LPVOID pBuffer,
   DWORD nBufSize,
   LPOVERLAPPED pOverlapped
) throw( );
HRESULT Read(
   LPVOID pBuffer,
   DWORD nBufSize,
   LPOVERLAPPED pOverlapped,
   LPOVERLAPPED_COMPLETION_ROUTINE pfnCompletionRoutine
) throw( );
```

Parameters

pBuffer

Pointer to the buffer that will receive the data read from the file.

nBufSize

The buffer size in bytes.

nBytesRead

The number of bytes read.

pOverlapped

The overlapped structure. See *lpOverlapped* in ReadFile in the *Platform SDK*.

pfnCompletionRoutine

The completion routine. See *lpCompletionRoutine* in ReadFileEx in the *Platform SDK*.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The first three forms call ReadFile, the last ReadFileEx to read data from the file. Use CAtlFile::Seek to move the file pointer.

Example

See the following samples:

- ISAPIFilter Sample
- MantaWeb Sample
- RegExp Sample

See Also

Reference

CAtlFile Class

CAtlFile::Write

Other Resources

CAtlFile Members

CAtlFile::Seek

Call this method to move the file pointer of the file.

```
HRESULT Seek(
   LONGLONG nOffset,
   DWORD dwFrom = FILE_CURRENT
) throw( );
```

Parameters

nOffset

The offset from the starting point given by *dwFrom*.

dwFrom

The starting point (FILE_BEGIN, FILE_CURRENT, or FILE_END).

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Calls SetFilePointer to move the file pointer.

See Also

Reference

CAtlFile Class

Other Resources

CAtlFile Members

CAtlFile::SetSize

Call this method to set the size of the file.

```
HRESULT SetSize(
   ULONGLONG nNewLen
) throw( );
```

Parameters

nNewLen

The new length of the file in bytes.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Calls SetFilePointer and SetEndOfFile to set the size of the file. On return, the file pointer is positioned at the end of the file.

See Also

Reference

CAtlFile Class

CAtlFile::GetSize

Other Resources

CAtlFile Members

CAtlFile::UnlockRange

Call this method to unlock a region of the file.

```
HRESULT UnlockRange(
   ULONGLONG nPos,
   ULONGLONG nCount
) throw( );
```

Parameters

nPos

The position in the file where the unlock should begin.

nCount

The length of the byte range to be unlocked.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Calls UnlockFile to unlock a region of the file.

See Also

Reference

CAtlFile Class

CAtlFile::LockRange

Other Resources

CAtlFile Members

CAtlFile::Write

Call this method to write data to the file starting at the position indicated by the file pointer.

```
HRESULT Write(
   LPCVOID pBuffer,
   DWORD nBufSize,
   LPOVERLAPPED pOverlapped,
   LPOVERLAPPED_COMPLETION_ROUTINE pfnCompletionRoutine
) throw( );
HRESULT Write(
   LPCVOID pBuffer,
   DWORD nBufSize,
   DWORD* pnBytesWritten = NULL
) throw( );
HRESULT Write(
   LPCVOID pBuffer,
   DWORD nBufSize,
   LPOVERLAPPED pOverlapped
) throw( );
```

Parameters

pBuffer

The buffer containing the data to be written to the file.

nBufSize

The number of bytes to be transferred from the buffer.

pOverlapped

The overlapped structure. See *lpOverlapped* in WriteFile in the *Platform SDK*.

pfnCompletionRoutine

The completion routine. See *lpCompletionRoutine* in WriteFileEx in the *Platform SDK*.

pnBytesWritten

The bytes written.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The first three forms call WriteFile, the last calls WriteFileEx to write data to the file. Use CAtlFile::Seek to move the file pointer.

See Also

Reference

CAtlFile Class

CAtlFile::Read

Other Resources

CAtlFile Members

CAtlFileMapping Class

This class represents a memory-mapped file, adding a cast operator to the methods of CAtlFileMappingBase.

```
template <
   typename T = char
>
class CAtlFileMapping :
   public CAtlFileMappingBase
```

Parameters

T

The type of data used for the cast operator.

Remarks

This class adds a single cast operator to allow implicit conversion of **CAtlFileMapping** objects to *T**. Other members are supplied by the base class, CAtlFileMappingBase.

Requirements

Header: atlfile.h

See Also

Reference

CAtlFileMappingBase Class

Other Resources

CAtlFileMapping Members

ATL Class Overview

CAtlFileMapping Members

Operators

operator T*	Allows implicit conversion of **CAtlFileMapping** objects to *T***.

See Also

Reference

CAtlFileMapping Class

Other Resources

CAtlFileMappingBase Members

CAtlFileMapping Operators

For information about the operators in **CAtlFileMapping**, see CAtlFileMapping Members.

CAtlFileMapping::operator T*

Allows implicit conversion of **CAtlFileMapping** objects to *T**.

```
operator T*() const throw( );
```

Return Value

Returns a *T** pointer to the start of the memory-mapped file.

Remarks

Calls CAtlFileMappingBase::GetData and reinterprets the returned pointer as a *T** where *T* is the type used as the template parameter of this class.

See Also

Reference

CAtlFileMappingBase Class

CAtlFileMappingBase::GetData

Other Resources

CAtlFileMappingBase Members

CAtlFileMappingBase Class

This class represents a memory-mapped file.

```
class CAtlFileMappingBase
```

Remarks

File mapping is the association of a file's contents with a portion of the virtual address space of a process. This class provides methods for creating file-mapping objects that permit programs to easily access and share data.

For more information, see File Mapping in the *Platform SDK*.

Requirements

Header: atlfile.h

See Also

Reference

CAtlFileMapping Class

Other Resources

CAtlFileMappingBase Members

ATL Class Overview

CAtlFileMappingBase Members

Methods

CAtlFileMappingBase	The constructor.
~CAtlFileMappingBase	The destructor.
CopyFrom	Call this method to copy from a file-mapping object.
GetData	Call this method to get the data from a file-mapping object.
GetHandle	Call this method to return the file handle.
GetMappingSize	Call this method to get the mapping size from a file-mapping object.
MapFile	Call this method to create a file-mapping object.
MapSharedMem	Call this method to create a file-mapping object that permits full access to all processes.
OpenMapping	Call this method to return a handle to the file-mapping object.
Unmap	Call this method to unmap a file-mapping object.

Operators

operator =	Sets the current file-mapping object to another file-mapping object.

See Also

Reference

CAtlFileMappingBase Class

Other Resources

CAtlFileMapping Members

CAtlFileMappingBase Methods

For information about the methods in **CAtlFileMappingBase**, see CAtlFileMappingBase Members.

CAtlFileMappingBase::CAtlFileMappingBase

The constructor.

```
CAtlFileMappingBase(
   CAtlFileMappingBase& orig
);
CAtlFileMappingBase( ) throw( );
```

Parameters

orig

The original file-mapping object to copy to create the new object.

Remarks

Creates a new file-mapping object, optionally using an existing object. It is still necessary to call CAtlFileMappingBase::MapFile to open or create the file-mapping object for a particular file.

Example

```
// FileMapSample
#include "stdafx.h"
#include "atlfile.h"

int main(int argc, char* argv[])
{
   // Create the file-mapping object.
   CAtlFileMappingBase myFileMap;

   // Create a file.
   CAtlFile myFile;
myFile.Create("myMapTestFile",
      GENERIC_READ|GENERIC_WRITE|STANDARD_RIGHTS_ALL,
      FILE_SHARE_READ|FILE_SHARE_WRITE,
      OPEN_ALWAYS);

// The file handle.
   HANDLE hFile = (HANDLE)myFile;

   // Test the file has opened successfully.
   ATLASSERT(hFile != INVALID_HANDLE_VALUE);

   // Open the file for file-mapping.
   // Must give a size as the file is zero by default.
   if (myFileMap.MapFile(hFile,
      1024,
      0,
      PAGE_READWRITE,
      FILE_MAP_READ) != S_OK)
   {
      CloseHandle(hFile);
      return 0;
   }

   // Confirm the size of the mapping file.
   ATLASSERT(myFileMap.GetMappingSize() == 1024);

   // Now the file-mapping object is open, a second
   // process could access the filemap object to exchange
   // data.

   return 0;
}
```

See Also

Reference

CAtlFileMappingBase Class

Other Resources

CAtlFileMappingBase Members

CAtlFileMappingBase::~CAtlFileMappingBase

The destructor.

```
~CAtlFileMappingBase( ) throw( );
```

Remarks

Frees any resources allocated by the class and calls the CAtlFileMappingBase::Unmap method.

See Also

Reference

CAtlFileMappingBase Class

Other Resources

CAtlFileMappingBase Members

CAtlFileMappingBase::CopyFrom

Call this method to copy from a file-mapping object.

```
HRESULT CopyFrom(
   CAtlFileMappingBase& orig
) throw( );
```

Parameters

orig

The original file-mapping object to copy from.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

See Also

Reference

CAtlFileMappingBase Class

Other Resources

CAtlFileMappingBase Members

CAtlFileMappingBase::GetData

Call this method to get the data from a file-mapping object.

```
void* GetData( ) const throw( );
```

Return Value

Returns a pointer to the data.

See Also

Reference

CAtlFileMappingBase Class

Other Resources

CAtlFileMappingBase Members

CAtlFileMappingBase::GetHandle

Call this method to return a handle to the file-mapping object.

```
HANDLE GetHandle( ) throw ( );
```

Return Value

Returns a handle to the file-mapping object.

See Also

Reference

CAtlFileMappingBase Class

Other Resources

CAtlFileMappingBase Members

CAtlFileMappingBase::GetMappingSize

Call this method to get the mapping size from a file-mapping object.

```
SIZE_T GetMappingSize( ) throw( );
```

Return Value

Returns the mapping size.

Example

See the example for CAtlFileMappingBase::CAtlFileMappingBase.

See Also

Reference

CAtlFileMappingBase Class

Other Resources

CAtlFileMappingBase Members

CAtlFileMappingBase::MapFile

Call this method to open or create a file-mapping object for the specified file.

```
HRESULT MapFile(
   HANDLE hFile,
   SIZE_T nMappingSize = 0,
   ULONGLONG nOffset = 0,
   DWORD dwMappingProtection = PAGE_READONLY,
   DWORD dwViewDesiredAccess = FILE_MAP_READ
) throw( );
```

Parameters

hFile

Handle to the file from which to create a mapping object. *hFile* must be valid and cannot be set to INVALID_HANDLE_VALUE.

nMappingSize

The mapping size. If 0, the maximum size of the file-mapping object is equal to the current size of the file identified by *hFile*.

nOffset

The file offset where mapping is to begin. The offset value must be a multiple of the system's memory allocation granularity.

dwMappingProtection

The protection desired for the file view when the file is mapped. See *flProtect* in CreateFileMapping in the *Platform SDK*.

dwViewDesiredAccess

Specifies the type of access to the file view and, therefore, the protection of the pages mapped by the file. See *dwDesiredAccess* in MapViewOfFileEx in the *Platform SDK*.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

After a file-mapping object has been created, the size of the file must not exceed the size of the file-mapping object; if it does, not all of the file's contents will be available for sharing. For more details, see CreateFileMapping and MapViewOfFileEx in the *Platform SDK*.

Example

See the example for CAtlFileMappingBase::CAtlFileMappingBase.

Smart Device Developer Notes

The hFile parameter must be a handle returned from CreateFileForMapping, and cannot be a handle returned from CreateFile. This restriction also means that this method cannot be used in conjunction with AtlFile.

See Also

Reference

CAtlFileMappingBase Class

Other Resources

CAtlFileMappingBase Members

CAtlFileMappingBase::MapSharedMem

Call this method to create a file-mapping object that permits full access to all processes.

```
HRESULT MapSharedMem(
   SIZE_T nMappingSize,
   LPCTSTR szName,
   BOOL* pbAlreadyExisted = NULL,
   LPSECURITY_ATTRIBUTES lpsa = NULL,
   DWORD dwMappingProtection = PAGE_READWRITE,
   DWORD dwViewDesiredAccess = FILE_MAP_ALL_ACCESS
) throw( );
```

Parameters

nMappingSize

The mapping size. If 0, the maximum size of the file-mapping object is equal to the current size of the file-mapping object identified by *szName*.

szName

The name of the mapping object.

pbAlreadyExisted

Points to a BOOL value that is set to TRUE if the mapping object already existed.

lpsa

The pointer to a **SECURITY_ATTRIBUTES** structure that determines whether the returned handle can be inherited by child processes. See *lpAttributes* in CreateFileMapping in the *Platform SDK*.

dwMappingProtection

The protection desired for the file view, when the file is mapped. See *flProtect* in **CreateFileMapping** in the *Platform SDK*.

dwViewDesiredAccess

Specifies the type of access to the file view and, therefore, the protection of the pages mapped by the file. See *dwDesiredAccess* in MapViewOfFileEx in the *Platform SDK*.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

MapShareMem allows an existing file-mapping object, created by CreateFileMapping, to be shared between processes.

See Also

Reference

CAtlFileMappingBase Class

Other Resources

CAtlFileMappingBase Members

CAtlFileMappingBase::OpenMapping

Call this method to open a named file-mapping object for the specified file.

```
HRESULT OpenMapping(
   LPCTSTR szName,
   SIZE_T nMappingSize,
   ULONGLONG nOffset = 0,
   DWORD dwViewDesiredAccess = FILE_MAP_ALL_ACCESS
) throw( );
```

Parameters

szName

The name of the mapping object. If there is an open handle to a file-mapping object by this name and the security descriptor on the mapping object does not conflict with the *dwViewDesiredAccess* parameter, the open operation succeeds.

nMappingSize

The mapping size. If 0, the maximum size of the file-mapping object is equal to the current size of the file-mapping object identified by *szName*.

nOffset

The file offset where mapping is to begin. The offset value must be a multiple of the system's memory allocation granularity.

dwViewDesiredAccess

Specifies the type of access to the file view and, therefore, the protection of the pages mapped by the file. See *dwDesiredAccess* in MapViewOfFileEx in the *Platform SDK*.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

In debug builds, an assertion error will occur if the input parameters are invalid.

See Also

Reference

CAtlFileMappingBase Class
CAtlFileMappingBase::MapFile

Other Resources

CAtlFileMappingBase Members

CAtlFileMappingBase::Unmap

Call this method to unmap a file-mapping object.

```
HRESULT Unmap( ) throw( );
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

See UnmapViewOfFile in the *Platform SDK* for more details.

See Also

Reference

CAtlFileMappingBase Class

Other Resources

CAtlFileMappingBase Members

CAtlFileMappingBase Operators

For information about the operators in **CAtlFileMappingBase**, see CAtlFileMappingBase Members.

CAtlFileMappingBase::operator =

Sets the current file-mapping object to another file-mapping object.

```
CAtlFileMappingBase& operator =(
   CAtlFileMappingBase& orig
);
```

Parameters

orig

The current file-mapping object.

Return Value

Returns a reference to the current object.

See Also

Reference

CAtlFileMappingBase Class

Other Resources

CAtlFileMappingBase Members

CAtlList Class

This class provides methods for creating and managing a list object.

```
template<
   typename E,
   class ETraits = CElementTraits< E >
>
class CAtlList
```

Parameters

E

The element type.

ETraits

The code used to copy or move elements. See CElementTraits Class for more details.

Remarks

The **CAtlList** class supports ordered lists of nonunique objects accessible sequentially or by value. **CAtlList** lists behave like doubly linked lists. Each list has a head and a tail, and new elements (or lists in some cases) can be added to either end of the list, or inserted before or after specific elements.

Most of the **CAtlList** methods make use of a position value. This value is used by the methods to reference the actual memory location where the elements are stored, and should not be calculated or predicted directly. If it is necessary to access the *n*th element in the list, the method CAtlList::FindIndex will return the corresponding position value for a given index. The methods CAtlList::GetNext and CAtlList::GetPrev can be used to iterate through the objects in the list.

For more information regarding the collection classes available with ATL, see ATL Collection Classes.

Requirements

Header: atlcoll.h

See Also

Reference

CList Class

Other Resources

CAtlList Members

ATL Class Overview

CAtlList Members

Methods

AddHead	Call this method to add an element to the head of the list.
AddHeadList	Call this method to add an existing list to the head of the list.
AddTail	Call this method to add an element to the tail of this list.
AddTailList	Call this method to add an existing list to the tail of this list.
AssertValid	Call this method to confirm the list is valid.
CAtlList	The constructor.
~CAtlList	The destructor.
Find	Call this method to search the list for the specified element.
FindIndex	Call this method to obtain the position of an element, given an index value.
GetAt	Call this method to return the element at a specified position in the list.
GetCount	Call this method to return the number of objects in the list.
GetHead	Call this method to return the element at the head of the list.
GetHeadPosition	Call this method to obtain the position of the head of the list.
GetNext	Call this method to return the next element from the list.
GetPrev	Call this method to return the previous element from the list.
GetTail	Call this method to return the element at the tail of the list.
GetTailPosition	Call this method to obtain the position of the tail of the list.
InsertAfter	Call this method to insert a new element into the list after the specified position.
InsertBefore	Call this method to insert a new element into the list before the specified position.
IsEmpty	Call this method to determine if the list is empty.
MoveToHead	Call this method to move the specified element to the head of the list.
MoveToTail	Call this method to move the specified element to the tail of the list.
RemoveAll	Call this method to remove all of the elements from the list.
RemoveAt	Call this method to remove a single element from the list.

RemoveHead	Call this method to remove the element at the head of the list.
RemoveHeadNoReturn	Call this method to remove the element at the head of the list without returning a value.
RemoveTail	Call this method to remove the element at the tail of the list.
RemoveTailNoReturn	Call this method to remove the element at the tail of the list without returning a value.
SetAt	Call this method to set the value of the element at a given position in the list.
SwapElements	Call this method to swap elements in the list.

Typedefs

INARGTYPE	Type used when an element is passed as an input argument.

See Also

Reference

CAtlList Class

CAtList Methods

For information about the methods in **CAtlList**, see CAtlList Members.

CAtlList::AddHead

Call this method to add an element to the head of the list.

```
POSITION AddHead( );
POSITION AddHead(
   INARGTYPE element
);
```

Parameters

element

The new element.

Return Value

Returns the position of the newly added element.

Remarks

If the first version is used, an empty element is created using its default constructor, rather than its copy constructor.

Example

```
// AddHead Example

// Declare a list of integers
CAtlList<int> myList;

// Add some elements, each to the head of the list.
// As each new element is added, the previous head is
// pushed down the list.
myList.AddHead(42);
myList.AddHead(49);

// Confirm the value currently at the head of the list
ATLASSERT(myList.GetHead() == 49);

// Confirm the value currently at the tail of the list
ATLASSERT(myList.GetTail() == 42);
```

See Also

Reference

CAtlList Class

CAtlList::AddHeadList

CAtlList::AddTail

Other Resources

CAtlList Members

CAtlList::AddHeadList

Call this method to add an existing list to the head of the list.

```
void AddHeadList(
   const CAtlList< E, ETraits >* plNew
);
```

Parameters

plNew

The list to be added.

Remarks

The list pointed to by *plNew* is inserted at the start of the existing list. In debug builds, an assertion failure will occur if *plNew* is equal to NULL.

Example

```
// AddHeadList Example

// Define two lists of integers
CAtlList<int> myList1;
CAtlList<int> myList2;

// Fill up the first list
myList1.AddTail(1);
myList1.AddTail(2);
myList1.AddTail(3);

// Add an element to the second list
myList2.AddTail(4);

// Insert the first list into the second
myList2.AddHeadList(&myList1);

// The second list now contains:
// 1, 2, 3, 4
```

See Also

Reference

CAtlList Class
CAtlList::AddHead
CAtlList::AddTailList

Other Resources

CAtlList Members

CAtlList::AddTail

Call this method to add an element to the tail of this list.

```
POSITION AddTail( );
POSITION AddTail(
   INARGTYPE element
);
```

Parameters

element

The element to add.

Return Value

Returns the POSITION of the newly added element.

Remarks

If the first version is used, an empty element is created using its default constructor, rather than its copy constructor. The element is added to the end of the list, and so it now becomes the tail. This method can be used with an empty list.

Example

```
// AddTail Example

// Define the list
CAtlList<int> myList;

// Add elements to the tail
myList.AddTail(1);
myList.AddTail(2);
myList.AddTail(3);

// Confirm the current head of the list
ATLASSERT(myList.GetHead() == 1);

// Confirm the current tail of the list
ATLASSERT(myList.GetTail() == 3);
```

See Also

Reference

CAtlList Class
CAtlList::AddTailList
CAtlList::AddHead

Other Resources

CAtlList Members

CAtlList::AddTailList

Call this method to add an existing list to the tail of this list.

```
void AddTailList(
   const CAtlList< E, ETraits >* plNew
);
```

Parameters

plNew

The list to be added.

Remarks

The list pointed to by *plNew* is inserted after the last element (if any) in the list object. The last element in the *plNew* list therefore becomes the tail. In debug builds, an assertion failure will occur if *plNew* is equal to NULL.

Example

```
// AddTailList Example

// Define two integer lists
CAtlList<int> myList1;
CAtlList<int> myList2;

// Fill up the first list
myList1.AddTail(1);
myList1.AddTail(2);
myList1.AddTail(3);

// Add an element to the second list
myList2.AddTail(4);

// Insert the first list into the second
myList2.AddTailList(&myList1);

// The second list now contains:
// 4, 1, 2, 3
```

See Also

Reference

CAtlList Class
CAtlList::AddTail
CAtlList::AddHead

Other Resources

CAtlList Members

CAtlList::AssertValid

Call this method to confirm the list is valid.

```
void AssertValid( ) const;
```

Remarks

In debug builds, an assertion failure will occur if the list object is not valid. To be valid, an empty list must have both the head and tail pointing to NULL, and a list that is not empty must have both the head and tail pointing to valid addresses.

Example

```
// AssertValid Example

// Define the list
CAtlList<int> myList;

// AssertValid only exists in debug builds
#ifdef _DEBUG
myList.AssertValid();
#endif
```

See Also

Reference

CAtlList Class

CAtlList::IsEmpty

Other Resources

CAtlList Members

CAtlList::CAtlList

The constructor.

```
CAtlList(
   UINT nBlockSize = 10
) throw( );
```

Parameters

nBlockSize

The block size.

Remarks

The constructor for the **CAtlList** object. The block size is a measure of the amount of memory allocated when a new element is required. Larger block sizes reduce calls to memory allocation routines, but use more resources.

Example

```
// CAtlList Example

// Define two lists
CAtlList<int> myList1;
CAtlList<double> myList2;
```

See Also

Reference

CAtlList Class

CAtlList::~CAtlList

Other Resources

CAtlList Members

CAtlList::~CAtlList

The destructor.

```
~CAtlList( ) throw( );
```

Remarks

Frees all allocated resources, including a call to CAtlList::RemoveAll to remove all elements from the list.

In debug builds, an assertion failure will occur if the list still contains some elements after the call to **RemoveAll**.

See Also

Reference

CAtlList Class

CAtlList::CAtlList

Other Resources

CAtlList Members

CAtlList::Find

Call this method to search the list for the specified element.

```
POSITION Find(
   INARGTYPE element,
   POSITION posStartAfter = NULL
) const throw( );
```

Parameters

element

The element to be found in the list.

posStartAfter

The start position for the search. If no value is specified, the search begins with the head element.

Return Value

Returns the POSITION value of the element if found, otherwise returns NULL.

Remarks

In debug builds, an assertion failure will occur if the list object is not valid, or if the *posStartAfter* value is out of range.

Example

```
// Find example

// Define the integer list
CAtlList<int> myList;

// Populate the list
myList.AddTail(100);
myList.AddTail(200);
myList.AddTail(300);
myList.AddTail(400);

// Find the '300' element in the list,
// starting from the list head.
POSITION myPos = myList.Find(300);

// Confirm that the element was found
ATLASSERT(myList.GetAt(myPos) == 300);
```

See Also

Reference

CAtlList Class

CAtlList::FindIndex

Other Resources

CAtlList Members

CAtlList::FindIndex

Call this method to obtain the position of an element, given an index value.

```
POSITION FindIndex(
   size_t iElement
) const throw( );
```

Parameters

iElement

The zero-based index of the required list element.

Return Value

Returns the corresponding POSITION value, or NULL if *iElement* is out of range.

Remarks

This method returns the POSITION corresponding to a given index value, allowing access to the *n*th element in the list.

In debug builds, an assertion failure will occur if the list object is not valid.

Example

```
// FindIndex Example

// Define the integer list
CAtlList<int> myList;

// Populate the list
for (int i=0; i<100; i++)
   myList.AddTail(i);

// Iterate through the entire list
for (size_t j=0; j<myList.GetCount(); j++)
{
   i=myList.GetAt(myList.FindIndex(j));
   ATLASSERT(i == j);
}
```

See Also

Reference

CAtlList Class

CAtlList::Find

Other Resources

CAtlList Members

CAtlList::GetAt

Call this method to return the element at a specified position in the list.

```
E& GetAt(
   POSITION pos
) throw( );
const E& GetAt(
   POSITION pos
) const throw( );
```

Parameters

pos

The POSITION value specifying a particular element.

Return Value

A reference to, or copy of, the element.

Remarks

If the list is **const**, **GetAt** returns a copy of the element. This allows the method to be used only on the right side of an assignment statement and protects the list from modification.

If the list is not **const**, **GetAt** returns a reference to the element. This allows the method to be used on either side of an assignment statement and thus allows the list entries to be modified.

In debug builds, an assertion failure will occur if *pos* is equal to NULL.

Example

See the example for CAtlList::FindIndex.

See Also

Reference

CAtlList Class

CAtlList::FindIndex

Other Resources

CAtlList Members

CAtlList::GetCount

Call this method to return the number of objects in the list.

```
size_t GetCount( ) const throw( );
```

Return Value

Returns the number of elements in the list.

Example

See the example for CAtlList::Find.

See Also

Reference

CAtlList Class

CAtlList::IsEmpty

Other Resources

CAtlList Members

CAtlList::GetHead

Call this method to return the element at the head of the list.

```
E& GetHead( ) throw( );
const E& GetHead( ) const throw( );
```

Return Value

Returns a reference to, or a copy of, the element at the head of the list.

Remarks

If the list is **const**, **GetHead** returns a copy of the element at the head of the list. This allows the method to be used only on the right side of an assignment statement and protects the list from modification.

If the list is not **const**, **GetHead** returns a reference to the element at the head of the list. This allows the method to be used on either side of an assignment statement and thus allows the list entries to be modified.

In debug builds, an assertion failure will occur if the head of the list points to NULL.

Example

See the example for CAtlList::AddHead.

See Also

Reference

CAtlList Class
CAtlList::GetHeadPosition
CAtlList::GetTail

Other Resources

CAtlList Members

CAtlList::GetHeadPosition

Call this method to obtain the position of the head of the list.

```
POSITION GetHeadPosition( ) const throw( );
```

Return Value

Returns the POSITION value corresponding to the element at the head of the list.

Remarks

If the list is empty, the value returned is NULL.

Example

```
// GetHeadPosition Example

// Define the integer list
CAtlList<int> myList;

// Populate the list
for (int i=0; i<100; i++)
   myList.AddTail(i);

// Get the starting position value
POSITION myPos = myList.GetHeadPosition();

// Iterate through the entire list
i = 0;
int j;

do {
   j = myList.GetNext(myPos);
   ATLASSERT(i == j);
   i++;
} while (myPos != NULL);
```

See Also

Reference

CAtlList Class
CAtlList::GetHead
CAtlList::GetTailPosition

Other Resources

CAtlList Members

CAtlList::GetNext

Call this method to return the next element from the list.

```
E& GetNext(
   POSITION& pos
) throw( );
const E& GetNext(
   POSITION& pos
) const throw( );
```

Parameters

pos

A POSITION value, returned by a previous call to **GetNext**, CAtlList::GetHeadPosition, or other **CAtlList** method.

Return Value

If the list is **const**, **GetNext** returns a copy of the next element of the list. This allows the method to be used only on the right side of an assignment statement and protects the list from modification.

If the list is not **const**, **GetNext** returns a reference to the next element of the list. This allows the method to be used on either side of an assignment statement and thus allows the list entries to be modified.

Remarks

The POSITION counter, *pos*, is updated to point to the next element in the list, or NULL if there are no more elements. In debug builds, an assertion failure will occur if *pos* is equal to NULL.

Example

See the example for CAtlList::GetHeadPosition.

See Also

Reference

CAtlList Class

CAtlList::GetPrev

Other Resources

CAtlList Members

CAtlList::GetPrev

Call this method to return the previous element from the list.

```
E& GetPrev(
   POSITION& pos
) throw( );
const E& GetPrev(
   POSITION& pos
) const throw( );
```

Parameters

pos

A POSITION value, returned by a previous call to **GetPrev**, CAtlList::GetTailPosition, or other **CAtlList** method.

Return Value

If the list is **const**, **GetPrev** returns a copy of an element of the list. This allows the method to be used only on the right side of an assignment statement and protects the list from modification.

If the list is not **const**, **GetPrev** returns a reference to an element of the list. This allows the method to be used on either side of an assignment statement and thus allows the list entries to be modified.

Remarks

The POSITION counter, *pos*, is updated to point to the previous element in the list, or NULL if there are no more elements. In debug builds, an assertion failure will occur if *pos* is equal to NULL.

Example

See the example for CAtlList::GetTailPosition.

See Also

Reference

CAtlList Class

CAtlList::GetNext

Other Resources

CAtlList Members

CAtlList::GetTail

Call this method to return the element at the tail of the list.

```
E& GetTail( ) throw( );
const E& GetTail( ) const throw( );
```

Return Value

Returns a reference to, or a copy of, the element at the tail of the list.

Remarks

If the list is **const**, **GetTail** returns a copy of the element at the head of the list. This allows the method to be used only on the right side of an assignment statement and protects the list from modification.

If the list is not **const**, **GetTail** returns a reference to the element at the head of the list. This allows the method to be used on either side of an assignment statement and thus allows the list entries to be modified.

In debug builds, an assertion failure will occur if the tail of the list points to NULL.

Example

See the example for CAtlList::AddTail.

See Also

Reference

CAtlList Class
CAtlList::GetTailPosition
CAtlList::GetHead

Other Resources

CAtlList Members

CAtlList::GetTailPosition

Call this method to obtain the position of the tail of the list.

```
POSITION GetTailPosition( ) const throw( );
```

Return Value

Returns the POSITION value corresponding to the element at the tail of the list.

Remarks

If the list is empty, the value returned is NULL.

Example

```
// GetTailPosition Example

// Define the integer list
CAtlList<int> myList;

// Populate the list
for (int i=0; i<100; i++)
   myList.AddHead(i);

// Get the starting position value
POSITION myP = myList.GetTailPosition();

// Iterate through the entire list
i = 0;
int j;

do {
   j = myList.GetPrev(myP);
   ATLASSERT(i == j);
   i++;
} while (myP != NULL);
```

See Also

Reference

CAtlList Class
CAtlList::GetTail
CAtlList::GetHeadPosition

Other Resources

CAtlList Members

CAtlList::InsertAfter

Call this method to insert a new element into the list after the specified position.

```
POSITION InsertAfter(
   POSITION pos,
   INARGTYPE element
);
```

Parameters

pos

The POSITION value after which the new element will be inserted.

element

The element to be inserted.

Return Value

Returns the POSITION value of the new element.

Remarks

In debug builds, an assertion failure will occur if the list isn't valid, if the insert fails, or if an attempt is made to insert the element after the tail.

Example

```
// InsertAfter Example

// Define the integer list
CAtlList<int> myList;

// Populate the list
POSITION myPos = myList.AddHead(1);
myPos = myList.InsertAfter(myPos,2);
myPos = myList.InsertAfter(myPos,3);

// Confirm the tail value is as expected
ATLASSERT(myList.GetTail() == 3);
```

See Also

Reference

CAtlList Class

CAtlList::InsertBefore

Other Resources

CAtlList Members

CAtlList::InsertBefore

Call this method to insert a new element into the list before the specified position.

```
POSITION InsertBefore(
   POSITION pos,
   INARGTYPE element
);
```

Parameters

pos

The new element will be inserted into the list before this POSITION value.

element

The element to be inserted.

Return Value

Returns the POSITION value of the new element.

Remarks

In debug builds, an assertion failure will occur if the list isn't valid, if the insert fails, or if an attempt is made to insert the element before the head.

Example

```
// InsertBefore Example

// Define the integer list
CAtlList<int> myList;

// Populate the list
POSITION myPos = myList.AddHead(1);
myPos = myList.InsertBefore(myPos, 2);
myPos = myList.InsertBefore(myPos, 3);

// Confirm the head value is as expected
ATLASSERT(myList.GetHead() == 3);
```

See Also

Reference

CAtlList Class

CAtlList::InsertAfter

Other Resources

CAtlList Members

CAtlList::IsEmpty

Call this method to determine if the list is empty.

```
bool IsEmpty( ) const throw( );
```

Return Value

Returns true if the list contains no objects, otherwise false.

Example

```
// IsEmpty Example

// Define the integer list
CAtlList<int> myList;

// Populate the list
myList.AddTail(1);
myList.AddTail(2);
myList.AddTail(3);
myList.AddTail(4);

// Confirm not empty
ATLASSERT(myList.IsEmpty() == false);

// Remove the tail element
myList.RemoveTailNoReturn();

// Confirm not empty
ATLASSERT(myList.IsEmpty() == false);

// Remove the head element
myList.RemoveHeadNoReturn();

// Confirm not empty
ATLASSERT(myList.IsEmpty() == false);

// Remove all remaining elements
myList.RemoveAll();

// Confirm empty
ATLASSERT(myList.IsEmpty() == true);
```

See Also

Reference

CAtlList Class

CAtlList::GetCount

Other Resources

CAtlList Members

CAtlList::MoveToHead

Call this method to move the specified element to the head of the list.

```
void MoveToHead(
   POSITION pos
) throw( );
```

Parameters

pos

The POSITION value of the element to move.

Remarks

The specified element is moved from its current position to the head of the list. In debug builds, an assertion failure will occur if *pos* is equal to NULL.

Example

```
// MoveToHead Example

// Define the integer list
CAtlList<int> myList;

// Populate the list
myList.AddTail(1);
myList.AddTail(2);
myList.AddTail(3);
myList.AddTail(4);

// Move the tail element to the head
myList.MoveToHead(myList.GetTailPosition());

// Confirm the head is as expected
ATLASSERT(myList.GetHead() == 4);

// Move the head element to the tail
myList.MoveToTail(myList.GetHeadPosition());

// Confirm the tail is as expected
ATLASSERT(myList.GetTail() == 4);
```

See Also

Reference

CAtlList Class

CAtlList::MoveToTail

CAtlList::SwapElements

Other Resources

CAtlList Members

CAtlList::MoveToTail

Call this method to move the specified element to the tail of the list.

```
void MoveToTail(
   POSITION pos
) throw( );
```

Parameters

pos

The POSITION value of the element to move.

Remarks

The specified element is moved from its current position to the tail of the list. In debug builds, an assertion failure will occur if *pos* is equal to NULL.

Example

See the example for CAtlList::MoveToHead.

See Also

Reference

CAtlList Class
CAtlList::MoveToHead
CAtlList::SwapElements

Other Resources

CAtlList Members

CAtlList::RemoveAll

Call this method to remove all of the elements from the list.

```
void RemoveAll( ) throw( );
```

Remarks

This method removes all of the elements from the list and frees the allocated memory. In debugs builds, an ATLASSERT will be raised if all elements aren't deleted or if the list structure has become corrupted.

Example

See the example for CAtlList::IsEmpty.

See Also

Reference

CAtlList Class

CAtlList::RemoveAt

Other Resources

CAtlList Members

CAtlList::RemoveAt

Call this method to remove a single element from the list.

```
void RemoveAt(
   POSITION pos
) throw( );
```

Parameters

pos

The POSITION value of the element to remove.

Remarks

The element referenced by *pos* is removed, and memory is freed. It is acceptable to use **RemoveAt** to remove the head or tail of the list.

In debug builds, an assertion failure will occur if the list is not valid or if removing the element causes the list to access memory which isn't part of the list structure.

Example

```
// RemoveAt Example

// Define the integer list
CAtlList<int> myList;

// Populate the list
myList.AddTail(100);
myList.AddTail(200);
myList.AddTail(300);

// Use RemoveAt to remove elements one by one
myList.RemoveAt(myList.Find(100));
myList.RemoveAt(myList.Find(200));
myList.RemoveAt(myList.Find(300));

// Confirm all have been deleted
ATLASSERT(myList.IsEmpty() == true);
```

See Also

Reference

CAtlList Class
CAtlList::RemoveAll
CAtlList::SetAt

Other Resources

CAtlList Members

CAtlList::RemoveHead

Call this method to remove the element at the head of the list.

```
E RemoveHead( );
```

Return Value

Returns the element at the head of the list.

Remarks

The head element is deleted from the list, and memory is freed. A copy of the element is returned. In debug builds, an assertion failure will occur if the list is empty.

Example

```
// RemoveHead Example

// Define the integer list
CAtlList<int> myList;

// Populate the list
myList.AddTail(100);
myList.AddTail(200);
myList.AddTail(300);

// Confirm the head of the list
ATLASSERT(myList.GetHead() == 100);

// Remove the head of the list
ATLASSERT(myList.RemoveHead() == 100);

// Confirm the new head of the list
ATLASSERT(myList.GetHead() == 200);
```

See Also

Reference

CAtlList Class

CAtlList::RemoveHeadNoReturn

Other Resources

CAtlList Members

CAtlList::RemoveHeadNoReturn

Call this method to remove the element at the head of the list without returning a value.

```
void RemoveHeadNoReturn( ) throw( );
```

Remarks

The head element is deleted from the list, and memory is freed. In debug builds, an assertion failure will occur if the list is empty.

Example

See the example for CAtlList::IsEmpty.

See Also

Reference

CAtlList Class
CAtlList::RemoveHead
CAtlList::RemoveTailNoReturn
CAtlList::RemoveTail

Other Resources

CAtlList Members

CAtlList::RemoveTail

Call this method to remove the element at the tail of the list.

```
E RemoveTail( );
```

Return Value

Returns the element at the tail of the list.

Remarks

The tail element is deleted from the list, and memory is freed. A copy of the element is returned. In debug builds, an assertion failure will occur if the list is empty.

Example

```
// RemoveTail Example

// Define the integer list
CAtlList<int> myList;

// Populate the list
myList.AddTail(100);
myList.AddTail(200);
myList.AddTail(300);

// Confirm the tail of the list
ATLASSERT(myList.GetTail() == 300);

// Remove the tail of the list
ATLASSERT(myList.RemoveTail() == 300);

// Confirm the new tail of the list
ATLASSERT(myList.GetTail() == 200);
```

See Also

Reference

CAtlList Class
CAtlList::RemoveTailNoReturn
CAtlList::RemoveHead
CAtlList::RemoveHeadNoReturn

Other Resources

CAtlList Members

CAtlList::RemoveTailNoReturn

Call this method to remove the element at the tail of the list without returning a value.

```
void RemoveTailNoReturn( ) throw( );
```

Remarks

The tail element is deleted from the list, and memory is freed. In debug builds, an assertion failure will occur if the list is empty.

Example

See the example for CAtlList::IsEmpty.

See Also

Reference

CAtlList Class
CAtlList::RemoveTail
CAtlList::RemoveHead
CAtlList::RemoveHeadNoReturn

Other Resources

CAtlList Members

CAtlList::SetAt

Call this method to set the value of the element at a given position in the list.

```
void SetAt(
   POSITION pos,
   INARGTYPE element
);
```

Parameters

pos

The POSITION value corresponding to the element to change.

element

The new element value.

Remarks

Replaces the existing value with *element*. In debug builds, an assertion failure will occur if *pos* is equal to NULL.

Example

```
// SetAt Example

// Define the integer list
CAtlList<int> myList;

// Populate the list
myList.AddTail(100);
myList.AddTail(200);

// Use SetAt to change the values stored in the head and
// tail of the list
myList.SetAt(myList.GetHeadPosition(), myList.GetHead() * 10);
myList.SetAt(myList.GetTailPosition(), myList.GetTail() * 10);

// Confirm the values
ATLASSERT(myList.GetHead() == 1000);
ATLASSERT(myList.GetTail() == 2000);
```

See Also

Reference

CAtlList Class

CAtlList::RemoveAt

Other Resources

CAtlList Members

CAtlList::SwapElements

Call this method to swap elements in the list.

```
void SwapElements(
   POSITION pos1,
   POSITION pos2
) throw( );
```

Parameters

pos1

The first POSITION value.

pos2

The second POSITION value.

Remarks

Swaps the elements at the two positions specified. In debug builds, an assertion failure will occur if either position value is equal to NULL.

Example

```
// SwapElements Example

// Define the integer list
CAtlList<int> myList;

// Populate the list
for (int i=0; i<100; i++)
   myList.AddHead(i);

// Order is: 99, 98, 97, 96...
ATLASSERT(myList.GetHead() == 99);
ATLASSERT(myList.GetTail() == 0);

// Perform a crude bubble sort
for (int j=0; j<100; j++)
   for(i=0; i<99; i++)
      if (myList.GetAt(myList.FindIndex(i)) >
         myList.GetAt(myList.FindIndex(i+1)))
            myList.SwapElements(myList.FindIndex(i),
                                myList.FindIndex(i+1));

// Order is: 0, 1, 2, 3...
ATLASSERT(myList.GetHead() == 0);
ATLASSERT(myList.GetTail() == 99);
```

See Also

Reference

CAtlList Class

CAtlList::MoveToHead

CAtlList::MoveToTail

Other Resources

CAtlList Members

CAtList Typedefs

For information about the typedefs in **CAtlList**, see CAtlList Members.

CAtlList::INARGTYPE

Type used when an element is passed as an input argument.

```
typedef ETraits::INARGTYPE INARGTYPE;
```

See Also

Reference

CAtlList Class

Other Resources

CAtlList Members

CAtlMap Class

This class provides methods for creating and managing a map object.

```
template<
      typename K,
      typename V,
      class KTraits = CElementTraits< K >,
      class VTraits = CElementTraits< V >
>
class CAtlMap
```

Parameters

K

The key element type.

V

The value element type.

KTraits

The code used to copy or move key elements. See CElementTraits Class for more details.

VTraits

The code used to copy or move value elements.

Remarks

CAtlMap provides support for a mapping array of any given type, managing an unordered array of key elements and their associated values. Elements (consisting of a key and a value) are stored using a hashing algorithm, allowing a large amount of data to be efficiently stored and retrieved.

The *KTraits* and *VTraits* parameters are traits classes that contain any supplemental code needed to copy or move elements.

An alternative to **CAtlMap** is offered by the CRBMap class. **CRBMap** also stores key/value pairs, but exhibits different performance characteristics. The time taken to insert an item, look up a key, or delete a key from a **CRBMap** object is of order *log(n)*, where *n* is the number of elements. For **CAtlMap**, all of these operations typically take a constant time, although worst-case scenarios might be of order *n*. Therefore, in a typical case, **CAtlMap** is faster.

The other difference between **CRBMap** and **CAtlMap** becomes apparent when iterating through the stored elements. In a **CRBMap**, the elements are visited in a sorted order. In a **CAtlMap**, the elements are not ordered, and no order can be inferred.

When a small number of elements need to be stored, consider using the CSimpleMap class instead.

For more information, see ATL Collection Classes.

Example

See the PooledAsync Sample.

Requirements

Header: atlcoll.h

See Also

Tasks

Marquee Sample: Provides Performance Monitoring Data Through Scripting

UpdatePV Sample: Implements an Updatable OLE DB Provider

Other Resources

CAtlMap Members

ATL Class Overview

CAtlMap Members

Methods

AssertValid	Call this method to cause an ASSERT if the **CAtlMap** is not valid.
CAtlMap	The constructor.
~CAtlMap	The destructor.
DisableAutoRehash	Call this method to disable automatic rehashing of the **CAtlMap** object.
EnableAutoRehash	Call this method to enable automatic rehashing of the **CAtlMap** object.
GetAt	Call this method to return the element at a specified position in the map.
GetCount	Call this method to retrieve the number of elements in the map.
GetHashTableSize	Call this method to determine the number of bins in the map's hash table.
GetKeyAt	Call this method to retrieve the key stored at the given position in the **CAtlMap** object.
GetNext	Call this method to obtain a pointer to the next element pair stored in the **CAtlMap** object.
GetNextAssoc	Gets the next element for iterating.
GetNextKey	Call this method to retrieve the next key from the **CAtlMap** object.
GetNextValue	Call this method to get the next value from the **CAtlMap** object.
GetStartPosition	Call this method to start a map iteration.
GetValueAt	Call this method to retrieve the value stored at a given position in the **CAtlMap** object.
InitHashTable	Call this method to initialize the hash table.
IsEmpty	Call this method to test for an empty map object.
Lookup	Call this method to look up keys or values in the **CAtlMap** object.
Rehash	Call this method to rehash the **CAtlMap** object.
RemoveAll	Call this method to remove all elements from the **CAtlMap** object.
RemoveAtPos	Call this method to remove the element at the given position in the **CAtlMap** object.
RemoveKey	Call this method to remove an element from the **CAtlMap** object, given the key.
SetAt	Call this method to insert an element pair into the map.
SetOptimalLoad	Call this method to set the optimal load of the **CAtlMap** object.

SetValueAt	Call this method to change the value stored at a given position in the **CAtlMap** object.

Operators

operator []	Replaces or adds a new element to the **CAtlMap**.

Data Members

m_key	The data member storing the key element.
m_value	The data member storing the value element.

Typedefs

KINARGTYPE	Type used when a key is passed as an input argument
KOUTARGTYPE	Type used when a key is returned as an output argument.
VINARGTYPE	Type used when a value is passed as an input argument.
VOUTARGTYPE	Type used when a value is passed as an output argument.

Classes

CPair	A class containing the key and value elements.

See Also

Reference

CAtlMap Class

CAtlMap Methods

For information about the methods in **CAtlMap**, see CAtlMap Members.

CAtlMap::AssertValid

Call this method to cause an ASSERT if the **CAtlMap** object is not valid.

```
void AssertValid( ) const;
```

Remarks

In debug builds, this method will cause an ASSERT if the **CAtlMap** object is not valid.

Example

See the example for CAtlMap::CAtlMap.

See Also

Reference

CAtlMap Class

CAtlMap::IsEmpty

Other Resources

CAtlMap Members

CAtlMap::CAtlMap

The constructor.

```
CAtlMap(
   UINT nBins = 17,
   float fOptimalLoad = 0.75f,
   float fLoThreshold = 0.25f,
   float fHiThreshold = 2.25f,
   UINT nBlockSize = 10
) throw ( );
```

Parameters

nBins

The number of bins providing pointers to the stored elements. See Remarks later in this topic for an explanation of bins.

fOptimalLoad

The optimal load ratio.

fLoThreshold

The lower threshold for the load ratio.

fHiThreshold

The upper threshold for the load ratio.

nBlockSize

The block size.

Remarks

CAtlMap references all of its stored elements by first creating an index using a hashing algorithm on the key. This index references a "bin" which contains a pointer to the stored elements. If the bin is already in use, a linked-list is created to access the subsequent elements. Traversing a list is slower than directly accessing the correct element, and so the map structure needs to balance storage requirements against performance. The default parameters have been chosen to give good results in most cases.

The load ratio is the ratio of the number of bins to the number of elements stored in the map object. When the map structure is recalculated, the *fOptimalLoad* parameter value will be used to calculate the number of bins required. This value can be changed using the CAtlMap::SetOptimalLoad method.

The *fLoThreshold* parameter is the lower value that the load ratio can reach before **CAtlMap** will recalculate the optimal size of the map.

The *fHiThreshold* parameter is the upper value that the load ratio can reach before the **CAtlMap** object will recalculate the optimal size of the map.

This recalculation process (known as rehashing) is enabled by default. If you want to disable this process, perhaps when entering a lot of data at one time, call the CAtlMap::DisableAutoRehash method. Reactivate it with the CAtlMap::EnableAutoRehash method.

The *nBlockSize* parameter is a measure of the amount of memory allocated when a new element is required. Larger block sizes reduce calls to memory allocation routines, but use more resources.

Before any data can be stored, it is necessary to initialize the hash table with a call to CAtlMap::InitHashTable.

Example

```
// Example program for CAtlMap class

#include "stdafx.h"
#include "atlcoll.h"
#include "math.h"
```

```
int main(int argc, char* argv[])
{
   // Create a map which stores a double
   // value using an integer key

   CAtlMap<int, double> mySinTable;
   int i;

   // Initialize the Hash Table
   mySinTable.InitHashTable(257);

   // Add items to the map
   for (i=0; i<90; i++)
      mySinTable[i]=sin(i);

   // Confirm the map is valid
   mySinTable.AssertValid();

   // Confirm the number of elements in the map
   ATLASSERT(mySinTable.GetCount()==90);

   // Remove elements with even key values
   for (i=0; i<90; i+=2)
      mySinTable.RemoveKey(i);

   // Confirm the number of elements in the map
   ATLASSERT(mySinTable.GetCount()==45);

   // Walk through all the elements in the map.
   // First, get start position.
   POSITION pos;
   int key;
   double value;
   pos = mySinTable.GetStartPosition();

   // Now iterate the map, element by element
   while (pos != NULL) {
      key = mySinTable.GetKeyAt(pos);
      value = mySinTable.GetNextValue(pos);
   }
   return 0;
}
```

See Also

Reference

CAtlMap Class

CAtlMap::~CAtlMap

Other Resources

CAtlMap Members

CAtlMap::~CAtlMap

The destructor.

```
~CAtlMap( ) throw( );
```

Remarks

Frees any allocated resources.

See Also

Reference

CAtlMap Class

CAtlMap::CAtlMap

Other Resources

CAtlMap Members

CAtlMap::DisableAutoRehash

Call this method to disable automatic rehashing of the **CAtlMap** object.

```
void DisableAutoRehash( ) throw( );
```

Remarks

When automatic rehashing is enabled (which it is by default), the number of bins in the hash table will automatically be recalculated if the load value (the ratio of the number of bins to the number of elements stored in the array) exceeds the maximum or minimum values specified at the time the map was created.

DisableAutoRehash is most useful when a large number of elements will be added to the map at once. Instead of triggering the rehashing process every time the limits are exceeded, it is more efficient to call **DisableAutoRehash**, add the elements, and finally call CAtlMap::EnableAutoRehash.

See Also

Reference

CAtlMap Class

CAtlMap::EnableAutoRehash

Other Resources

CAtlMap Members

CAtlMap::EnableAutoRehash

Call this method to enable automatic rehashing of the **CAtlMap** object.

```
void EnableAutoRehash( ) throw( );
```

Remarks

When automatic rehashing is enabled (which it is by default), the number of bins in the hash table will automatically be recalculated if the load value (the ratio of the number of bins to the number of elements stored in the array) exceeds the maximum or minimum values specified at the time the map is created.

EnableAutoRefresh is most often used after a call to CAtlMap::DisableAutoRehash.

See Also

Reference

CAtlMap Class

CAtlMap::DisableAutoRehash

Other Resources

CAtlMap Members

CAtlMap::GetAt

Call this method to return the element at a specified position in the map.

```
void GetAt(
   POSITION pos,
   KOUTARGTYPE key,
   VOUTARGTYPE value
) const;
CPair* GetAt(
   POSITION& pos
) throw( );
```

Parameters

pos

The position counter, returned by a previous call to CAtlMap::GetNextAssoc or CAtlMap::GetStartPosition.

key

Template parameter specifying the type of the map's key.

value

Template parameter specifying the type of the map's value.

Return Value

Returns a pointer to the current pair of key/value elements stored in the map.

Remarks

In debug builds, an assertion error will occur if *pos* is equal to NULL.

See Also

Reference

CAtlMap Class
CAtlMap::SetAt
CAtlMap::GetKeyAt
CAtlMap::GetValueAt
CAtlMap::CPair Class

Other Resources

CAtlMap Members

CAtlMap::GetCount

Call this method to retrieve the number of elements in the map.

```
size_t GetCount( ) const throw( );
```

Return Value

Returns the number of elements in the map object. A single element is a key/value pair.

Example

See the example for CAtlMap::CAtlMap.

See Also

Reference

CAtlMap Class

CAtlMap::GetHashTableSize

Other Resources

CAtlMap Members

CAtlMap::GetHashTableSize

Call this method to determine the number of bins in the map's hash table.

```
UINT GetHashTableSize( ) const throw( );
```

Return Value

Returns the number of bins in the hash table. See CAtlMap::CAtlMap for an explanation.

See Also

Reference

CAtlMap Class

CAtlMap::GetCount

Other Resources

CAtlMap Members

CAtlMap::GetKeyAt

Call this method to retrieve the key stored at the given position in the **CAtlMap** object.

```
const K& GetKeyAt(
   POSITION pos
) const throw( );
```

Parameters

pos

The position counter, returned by a previous call to CAtlMap::GetNextAssoc or CAtlMap::GetStartPosition.

Return Value

Returns a reference to the key stored at the given position in the **CAtlMap** object.

Example

See the example for CAtlMap::CAtlMap.

See Also

Reference

CAtlMap Class

CAtlMap::SetValueAt

Other Resources

CAtlMap Members

CAtlMap::GetNext

Call this method to obtain a pointer to the next element pair stored in the **CAtlMap** object.

```
CPair* GetNext(
   POSITION& pos
) throw( );
const CPair* GetNext(
   POSITION& pos
) const throw( );
```

Parameters

pos

The position counter, returned by a previous call to CAtlMap::GetNextAssoc or CAtlMap::GetStartPosition.

Return Value

Returns a pointer to the next pair of key/value elements stored in the map. The *pos* position counter is updated after each call. If the retrieved element is the last in the map, *pos* is set to NULL.

See Also

Reference

CAtlMap Class
CAtlMap::GetNextAssoc
CAtlMap::GetNextKey
CAtlMap::GetNextValue
CAtlMap::CPair Class

Other Resources

CAtlMap Members

CAtlMap::GetNextAssoc

Gets the next element for iterating.

```
void GetNextAssoc(
   POSITION& pos,
   KOUTARGTYPE key,
   VOUTARGTYPE value
) const;
```

Parameters

pos

The position counter, returned by a previous call to CAtlMap::GetNextAssoc or CAtlMap::GetStartPosition.

key

Template parameter specifying the type of the map's key.

value

Template parameter specifying the type of the map's value.

Remarks

The *pos* position counter is updated after each call. If the retrieved element is the last in the map, *pos* is set to NULL.

See Also

Reference

CAtlMap Class
CAtlMap::GetNext
CAtlMap::GetNextKey
CAtlMap::GetNextValue
CAtlMap::GetStartPosition

Other Resources

CAtlMap Members

CAtlMap::GetNextKey

Call this method to retrieve the next key from the **CAtlMap** object.

```
const K& GetNextKey(
   POSITION& pos
) const throw( );
```

Parameters

pos

The position counter, returned by a previous call to CAtlMap::GetNextAssoc or CAtlMap::GetStartPosition.

Return Value

Returns a reference to the next key in the map.

Remarks

Updates the current position counter, *pos*. If there are no more entries in the map, the position counter is set to NULL.

See Also

Reference

CAtlMap Class
CAtlMap::GetNextValue
CAtlMap::GetNext
CAtlMap::GetNextAssoc

Other Resources

CAtlMap Members

CAtlMap::GetNextValue

Call this method to get the next value from the **CAtlMap** object.

```
V& GetNextValue(
   POSITION& pos
) throw( );
const V& GetNextValue(
   POSITION& pos
) const throw( );
```

Parameters

pos

The position counter, returned by a previous call to CAtlMap::GetNextAssoc or CAtlMap::GetStartPosition.

Return Value

Returns a reference to the next value in the map.

Remarks

Updates the current position counter, *pos*. If there are no more entries in the map, the position counter is set to NULL.

Example

See the example for CAtlMap::CAtlMap.

See Also

Reference

CAtlMap Class
CAtlMap::GetNext
CAtlMap::GetNextKey
CAtlMap::GetNextAssoc

Other Resources

CAtlMap Members

CAtlMap::GetStartPosition

Call this method to start a map iteration.

```
POSITION GetStartPosition( ) const throw( );
```

Return Value

Returns the start position, or NULL is returned if the map is empty.

Remarks

Call this method to start a map iteration by returning a **POSITION** value that can be passed to the **GetNextAssoc** method.

Note
The iteration sequence is not predictable

Example

See the example for CAtlMap::CAtlMap.

See Also

Reference

CAtlMap Class

CAtlMap::GetNextAssoc

Other Resources

CAtlMap Members

CAtlMap::GetValueAt

Call this method to retrieve the value stored at a given position in the **CAtlMap** object.

```
V& GetValueAt(
   POSITION pos
) throw( );
const V& GetValueAt(
   POSITION pos
) const throw( );
```

Parameters

pos

The position counter, returned by a previous call to CAtlMap::GetNextAssoc or CAtlMap::GetStartPosition.

Return Value

Returns a reference to the value stored at the given position in the **CAtlMap** object.

See Also

Reference

CAtlMap Class

CAtlMap::SetValueAt

Other Resources

CAtlMap Members

CAtlMap::InitHashTable

Call this method to initialize the hash table.

```
bool InitHashTable(
   UINT nBins,
   bool bAllocNow = true
);
```

Parameters

nBins

The number of bins used by the hash table. See CAtlMap::CAtlMap for an explanation.

bAllocNow

A flag indication when memory should be allocated.

Return Value

Returns **true** on successful initialization, **false** on failure.

Remarks

InitHashTable must be called before any elements are stored in the hash table. If this method is not called explicitly, it will be called automatically the first time an element is added using the bin count specified by the **CAtlMap** constructor. Otherwise, the map will be initialized using the new bin count specified by the *nBins* parameter.

If the *bAllocNow* parameter is false, the memory required by the hash table will not be allocated until it is first required. This can be useful if it is uncertain if the map will be used.

Example

See the example for CAtlMap::CAtlMap.

See Also

Reference

CAtlMap Class

Other Resources

CAtlMap Members

CAtlMap::IsEmpty

Call this method to test for an empty map object.

```
bool IsEmpty( ) const throw( );
```

Return Value

Returns **true** if the map is empty, **false** otherwise.

See Also

Reference

CAtlMap Class

CAtlMap::AssertValid

Other Resources

CAtlMap Members

CAtlMap::Lookup

Call this method to look up keys or values in the **CAtlMap** object.

```
bool Lookup(
   KINARGTYPE key,
   VOUTARGTYPE value
) const;
const CPair* Lookup(
   KINARGTYPE key
) const throw( );
CPair* Lookup(
   KINARGTYPE key
) throw( );
```

Parameters

key

Specifies the key that identifies the element to be looked up.

value

Variable that receives the looked-up value.

Return Value

The first form of the method returns true if the key is found, otherwise false. The second and third forms return a pointer to a CPair which can be used as a position for calls to CAtlMap::GetNext and so on.

Remarks

Lookup uses a hashing algorithm to quickly find the map element containing a key that exactly matches the given key parameter.

See Also

Reference

CAtlMap Class

Other Resources

CAtlMap Members

CAtlMap::Rehash

Call this method to rehash the **CAtlMap** object.

```
void Rehash(
   UINT nBins = 0
);
```

Parameters

nBins

The new number of bins to use in the hash table. See CAtlMap::CAtlMap for an explanation.

Remarks

If *nBins* is 0, **CAtlMap** calculates a reasonable number based on the number of elements in the map and the optimal load setting. Normally the rehashing process is automatic, but if CAtlMap::DisableAutoRehash has been called, this method will perform the necessary resizing.

See Also

Reference

CAtlMap Class

Other Resources

CAtlMap Members

CAtlMap::RemoveAll

Call this method to remove all elements from the **CAtlMap** object.

```
void RemoveAll( ) throw( );
```

Remarks

Clears out the **CAtlMap** object, freeing the memory used to store the elements.

See Also

Reference

CAtlMap Class
CAtlMap::RemoveKey
CAtlMap::RemoveAtPos

Other Resources

CAtlMap Members

CAtlMap::RemoveAtPos

Call this method to remove the element at the given position in the **CAtlMap** object.

```
void RemoveAtPos(
   POSITION pos
) throw( );
```

Parameters

pos

The position counter, returned by a previous call to CAtlMap::GetNextAssoc or CAtlMap::GetStartPosition.

Remarks

Removes the key/value pair stored at the specified position. The memory used to store the element is freed. The POSITION referenced by *pos* becomes invalid, and while the POSITION of any other elements in the map remains valid, they do not necessarily retain the same order.

See Also

Reference

CAtlMap Class
CAtlMap::RemoveKey
CAtlMap::RemoveAll

Other Resources

CAtlMap Members

CAtlMap::RemoveKey

Call this method to remove an element from the **CAtlMap** object, given the key.

```
bool RemoveKey(
   KINARGTYPE key
) throw( );
```

Parameters

key

The key corresponding to the element pair you want to remove.

Return Value

Returns **true** if the key is found and removed, **false** on failure.

Example

See the example for CAtlMap::CAtlMap.

See Also

Reference

CAtlMap Class
CAtlMap::RemoveAll
CAtlMap::RemoveAtPos

Other Resources

CAtlMap Members

CAtlMap::SetAt

Call this method to insert an element pair into the map.

```
POSITION SetAt(
   KINARGTYPE key,
   VINARGTYPE value
);
```

Parameters

key

The key value to add to the **CAtlMap** object.

value

The value to add to the **CAtlMap** object.

Return Value

Returns the position of the key/value element pair in the **CAtlMap** object.

Remarks

SetAt replaces an existing element if a matching key is found. If the key is not found, a new key/value pair is created.

See Also

Reference

CAtlMap Class

CAtlMap::SetValueAt

Other Resources

CAtlMap Members

CAtlMap::SetOptimalLoad

Call this method to set the optimal load of the **CAtlMap** object.

```
void SetOptimalLoad(
   float fOptimalLoad,
   float fLoThreshold,
   float fHiThreshold,
   bool bRehashNow = false
);
```

Parameters

fOptimalLoad

The optimal load ratio.

fLoThreshold

The lower threshold for the load ratio.

fHiThreshold

The upper threshold for the load ratio.

bRehashNow

Flag indicating if the hash table should be recalculated.

Remarks

This method redefines the optimal load value for the **CAtlMap** object. See CAtlMap::CAtlMap for a discussion of the various parameters. If *bRehashNow* is true, and the number of elements is outside the minimum and maximum values, the hash table is recalculated.

See Also

Reference

CAtlMap Class

Other Resources

CAtlMap Members

CAtlMap::SetValueAt

Call this method to change the value stored at a given position in the **CAtlMap** object.

```
void SetValueAt(
   POSITION pos,
   VINARGTYPE value
);
```

Parameters

pos

The position counter, returned by a previous call to CAtlMap::GetNextAssoc or CAtlMap::GetStartPosition.

value

The value to add to the **CAtlMap** object.

Remarks

Changes the value element stored at the given position in the **CAtlMap** object.

See Also

Reference

CAtlMap Class
CAtlMap::SetAt
CAtlMap::GetKeyAt

Other Resources

CAtlMap Members

CAtlMap Operators

For information about the operators in **CAtlMap**, see CAtlMap Members.

CAtlMap::operator []

Replaces or adds a new element to the **CAtlMap**.

```
V& operator[](
   KINARGTYPE key
) throw( );
```

Parameters

key

The key of the element to add or replace.

Return Value

Returns a reference to the value associated with the given key.

Example

If the key already exists, the element is replaced. If the key does not exist, a new element is added. See the example for CAtlMap::CAtlMap.

See Also

Reference

CAtlMap Class

CAtlMap::SetAt

Other Resources

CAtlMap Members

CAtlMap Typedefs

For information about the typedefs in **CAtlMap**, see CAtlMap Members.

CAtlMap::KINARGTYPE

Type used when a key is passed as an input argument.

```
typedef KTraits::INARGTYPE KINARGTYPE;
```

See Also

Reference

CAtlMap Class

CAtlMap::KOUTARGTYPE

Other Resources

CAtlMap Members

CAtlMap::KOUTARGTYPE

Type used when a key is returned as an output argument.

```
typedef KTraits::OUTARGTYPE KOUTARGTYPE;
```

See Also

Reference

CAtlMap Class

CAtlMap::KINARGTYPE

Other Resources

CAtlMap Members

CAtlMap::VINARGTYPE

Type used when a value is passed as an input argument.

```
typedef VTraits::INARGTYPE VINARGTYPE;
```

See Also

Reference

CAtlMap Class

CAtlMap::VOUTARGTYPE

Other Resources

CAtlMap Members

CAtlMap::VOUTARGTYPE

Type used when a value is passed as an output argument.

```
typedef VTraits::OUTARGTYPE VOUTARGTYPE;
```

See Also

Reference

CAtlMap Class

CAtlMap::VINARGTYPE

Other Resources

CAtlMap Members

CAtlMap Classes

For information about the classes in **CAtlMap**, see CAtlMap Members.

CAtlMap::CPair Class

A class containing the key and value elements.

```
class CPair : public __POSITION
```

Remarks

This class is used by the methods CAtlMap::GetNext and CAtlMap::Lookup to access the key and value elements stored in the mapping structure.

Requirements

Header: atlcoll.h

See Also

Reference

CAtlMap Class

Other Resources

CAtlMap Members

CAtlMap::CPair Data Members

For information about the **CPair** data members, see CAtlMap::CPair Class. For information about the data members in **CAtlMap**, see CAtlMap Members.

CAtlMap::CPair::m_key

The data member storing the key element.

```
const K m_key;
```

Parameters

K

The key element type.

See Also

Reference

CAtlMap Class

Other Resources

CAtlMap Members

CAtlMap::CPair::m_value

The data member storing the value element.

```
V m_value;
```

Parameters

V

The value element type.

See Also

Reference

CAtlMap Class

Other Resources

CAtlMap Members

CAtlModule Class

This class provides methods used by several ATL module classes.

```
class ATL_NO_VTABLE CAtlModule :
   public _ATL_MODULE
```

Remarks

This class is used by CAtlDllModuleT Class, CAtlExeModuleT Class, and CAtlServiceModuleT Class to provide support for DLL applications, EXE applications, and Windows services, respectively.

For more information on modules in ATL, see ATL Module Classes.

This class replaces the obsolete CComModule Class used in earlier versions of ATL.

Requirements

Header: atlbase.h

See Also

Reference

_ATL_MODULE

ATL Module Classes

Other Resources

CAtlModule Members

ATL Class Overview

The ATL Registry Component (Registrar)

CAtlModule Members

Methods

CAtlModule	The constructor.
~CAtlModule	The destructor.
AddCommonRGSReplacements	Override this method to add parameters to the ATL Registry Component (Registrar) replacement map.
AddTermFunc	Adds a new function to be called when the module terminates.
GetGITPtr	Returns the Global Interface Pointer.
GetLockCount	Returns the lock count.
Lock	Increments the lock count.
Term	Releases all data members.
Unlock	Decrements the lock count.
UpdateRegistryFromResourceD	Runs the script contained in a specified resource to register or unregister an object.
UpdateRegistryFromResourceDHelper	This method is called by **UpdateRegistryFromResourceD** to perform the registry update.
UpdateRegistryFromResourceS	Runs the script contained in a specified resource to register or unregister an object. This method statically links to the ATL Registry Component.

Data Members

m_libid	Contains the GUID of the current module.
m_pGIT	Pointer to the Global Interface Table.

See Also

Reference

CAtlModule Class

CAtlModule Methods

For information about the methods in **CAtlModule**, see CAtlModule Members.

CAtlModule::AddCommonRGSReplacements

Override this method to add parameters to the ATL Registry Component (Registrar) replacement map.

```
virtual HRESULT AddCommonRGSReplacements(
   IRegistrarBase* /*pRegistrar*/
) throw( ) = 0;
```

Parameters

pRegistrar

Reserved.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Replaceable parameters allow a Registrar's client to specify run-time data. To do this, the Registrar maintains a replacement map into which it enters the values associated with the replaceable parameters in your script. The Registrar makes these entries at run time.

See the topic Using Replaceable Parameters (The Registrar's Preprocessor) for more details.

See Also

Reference

CAtlModule Class

Other Resources

CAtlModule Members

The ATL Registry Component (Registrar)

CAtlModule::AddTermFunc

Adds a new function to be called when the module terminates.

```
HRESULT AddTermFunc(
   _ATL_TERMFUNC* pFunc,
   DWORD_PTR dw
) throw( );
```

Parameters

pFunc

Pointer to the function to add.

dw

User-defined data, passed to the function.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CAtlModule Class

CAtlModule::Term

Other Resources

CAtlModule Members

CAtlModule::CAtlModule

The constructor.

```
CAtlModule( ) throw( );
```

Remarks

Initializes data members and initiates a critical section around the module's thread.

See Also

Reference

CAtlModule Class

Other Resources

CAtlModule Members

CAtlModule::~CAtlModule

The destructor.

```
~CAtlModule( ) throw( );
```

Remarks

Releases all data members.

See Also

Reference

CAtlModule Class

Other Resources

CAtlModule Members

CAtlModule::GetGITPtr

Retrieves a pointer to the Global Interface Table.

```
virtual HRESULT GetGITPtr(
   IGlobalInterfaceTable** ppGIT
) throw( );
```

Parameters

ppGIT

Pointer to the variable which will receive the pointer to the Global Interface Table.

Return Value

Returns S_OK on success, or an error code on failure. E_POINTER is returned if *ppGIT* is equal to NULL.

Remarks

If the Global Interface Table object does not exist, it is created, and its address is stored in the member variable CAtlModule::m_pGIT.

In debug builds, an assertion error will occur if *ppGIT* is equal to NULL, or if the Global Interface Table pointer cannot be obtained.

See IGlobalInterfaceTable for information on the Global Interface Table.

See Also

Reference

CAtlModule Class

Other Resources

CAtlModule Members

CAtlModule::GetLockCount

Returns the lock count.

```
virtual LONG GetLockCount( ) throw( );
```

Return Value

Returns the lock count. This value may be useful for diagnostics and debugging.

See Also

Reference

CAtlModule Class

Other Resources

CAtlModule Members

CAtlModule::Lock

Increments the lock count.

```
virtual LONG Lock( ) throw( );
```

Return Value

Increments the lock count and returns the updated value. This value may be useful for diagnostics and debugging.

See Also

Reference

CAtlModule Class

Other Resources

CAtlModule Members

CAtlModule::Term

Releases all data members.

```
void Term( ) throw( );
```

Remarks

Releases all data members. This method is called by the destructor.

See Also

Reference

CAtlModule Class
CAtlModule::AddTermFunc
CAtlModule::~CAtlModule

Other Resources

CAtlModule Members

CAtlModule::Unlock

Decrements the lock count.

```
virtual LONG Unlock( ) throw( );
```

Return Value

Decrements the lock count and returns the updated value. This value may be useful for diagnostics and debugging.

See Also

Reference

CAtlModule Class

Other Resources

CAtlModule Members

CAtlModule::UpdateRegistryFromResourceD

Runs the script contained in a specified resource to register or unregister an object.

```
HRESULT WINAPI UpdateRegistryFromResourceD(
   UINT nResID,
   BOOL bRegister,
   struct _ATL_REGMAP_ENTRY* pMapEntries = NULL
) throw( );
HRESULT WINAPI UpdateRegistryFromResourceD(
   LPCTSTR lpszRes,
   BOOL bRegister,
   struct _ATL_REGMAP_ENTRY* pMapEntries = NULL
) throw( );
```

Parameters

lpszRes

A resource name.

nResID

A resource ID.

bRegister

TRUE if the object should be registered; **FALSE** otherwise.

pMapEntries

A pointer to the replacement map storing values associated with the script's replaceable parameters. ATL automatically uses %MODULE%. To use additional replaceable parameters, see CAtlModule::AddCommonRGSReplacements. Otherwise, use the **NULL** default value.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Runs the script contained in the resource specified by *lpszRes or nResID*. If *bRegister* is **TRUE**, this method registers the object in the system registry; otherwise it removes the object from the registry.

To statically link to the ATL Registry Component (Registrar), see CAtlModule::UpdateRegistryFromResourceS.

This method calls CAtlModule::UpdateRegistryFromResourceDHelper.

See Also

Reference

CAtlModule Class

Other Resources

CAtlModule Members

CAtlModule::UpdateRegistryFromResourceDHelper

This method is called by **UpdateRegistryFromResourceD** to perform the registry update.

```
inline HRESULT WINAPI UpdateRegistryFromResourceDHelper(
   LPCOLESTR lpszRes,
   BOOL bRegister,
   struct _ATL_REGMAP_ENTRY* pMapEntries = NULL
) throw( );
```

Parameters

lpszRes

A resource name.

bRegister

Indicates whether the object should be registered.

pMapEntries

A pointer to the replacement map storing values associated with the script's replaceable parameters. ATL automatically uses %MODULE%. To use additional replaceable parameters, see CAtlModule::AddCommonRGSReplacements. Otherwise, use the **NULL** default value.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method provides the implementation of CAtlModule::UpdateRegistryFromResourceD.

See Also

Reference

CAtlModule Class

Other Resources

CAtlModule Members

CAtlModule::UpdateRegistryFromResourceS

Runs the script contained in a specified resource to register or unregister an object. This method statically links to the ATL Registry Component.

```
HRESULT WINAPI UpdateRegistryFromResourceS(
   UINT nResID,
   BOOL bRegister,
   struct _ATL_REGMAP_ENTRY* pMapEntries = NULL
) throw( );
HRESULT WINAPI UpdateRegistryFromResourceS(
   LPCTSTR lpszRes,
   BOOL bRegister,
   struct _ATL_REGMAP_ENTRY* pMapEntries = NULL
) throw( );
```

Parameters

nResID

A resource ID.

lpszRes

A resource name.

bRegister

Indicates whether the resource script should be registered.

pMapEntries

A pointer to the replacement map storing values associated with the script's replaceable parameters. ATL automatically uses %MODULE%. To use additional replaceable parameters, see CAtlModule::AddCommonRGSReplacements. Otherwise, use the **NULL** default value.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Similar to CAtlModule::UpdateRegistryFromResourceD except **CAtlModule::UpdateRegistryFromResourceS** creates a static link to the ATL Registry Component (Registrar).

See Also

Reference

CAtlModule Class

Other Resources

CAtlModule Members

The ATL Registry Component (Registrar)

CAtlModule Data Members

For information about the data members in **CAtlModule**, see CAtlModule Members.

CAtlModule::m_libid

Contains the GUID of the current module.

```
static GUID m_libid;
```

See Also

Reference

CAtlModule Class

Other Resources

CAtlModule Members

CAtlModule::m_pGIT

Pointer to the Global Interface Table.

```
IGlobalInterfaceTable * m_pGIT;
```

See Also

Reference

CAtlModule Class

Other Resources

CAtlModule Members

CAtlModuleT Class

This class implements an ATL module.

```
template <
   class T
>
class ATL_NO_VTABLE CAtlModuleT :
   public CAtlModule
```

Parameters

T

Your class derived from **CAtlModuleT**.

Remarks

CAtlModuleT, derived from CAtlModule, implements an Executable (EXE) or a Service (EXE) ATL module. An Executable module is a local, out-of-process server, whereas a Service module is a Windows application that runs in the background when Windows starts.

CAtlModuleT provides support for initializing, registering, and unregistering of the module.

Requirements

Header: atlbase.h

See Also

Reference

CAtlModule Class

ATL Module Classes

Other Resources

CAtlModuleT Members

ATL Class Overview

CAtlModuleT Members

Methods

CAtlModuleT	The constructor.
RegisterAppId	Adds the EXE to the registry.
RegisterServer	Adds the service to the registry.
UnregisterAppId	Removes the EXE from the registry.
UnregisterServer	Removes the service from the registry.

Static Functions

InitLibId	Initializes the data member containing the GUID of the current module.
UpdateRegistryAppId	Updates the EXE information in the registry.

See Also

Reference

CAtlModuleT Class

Other Resources

CAtlModule Members

CAtlModuleT Methods

For information about the methods in **CAtlModuleT**, see CAtlModuleT Members.

CAtlModuleT::CAtlModuleT

The constructor.

```
CAtlModuleT( ) throw( );
```

Remarks

Calls CAtlModuleT::InitLibId.

See Also

Reference

CAtlModuleT Class

Other Resources

CAtlModuleT Members

CAtlModuleT::RegisterAppId

Adds the EXE to the registry.

```
HRESULT RegisterAppId( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CAtlModuleT Class

CAtlModuleT::UnregisterAppId

Other Resources

CAtlModuleT Members

CAtlModuleT::RegisterServer

Adds the service to the registry.

```
HRESULT RegisterServer(
   BOOL bRegTypeLib = FALSE,
   const CLSID* pCLSID = NULL
) throw( );
```

Parameters

bRegTypeLib

TRUE if the type library is to be registered. The default value is FALSE.

pCLSID

Points to the CLSID of the object to be registered. If NULL (the default value), all objects in the object map will be registered.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CAtlModuleT Class

CAtlModuleT::UnregisterServer

Other Resources

CAtlModuleT Members

CAtlModuleT::UnregisterAppId

Removes the EXE from the registry.

```
HRESULT UnregisterAppId( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CAtlModuleT Class

CAtlModuleT::RegisterAppId

Other Resources

CAtlModuleT Members

CAtlModuleT::UnregisterServer

Removes the service from the registry.

```
HRESULT UnregisterServer(
   BOOL bUnRegTypeLib,
   const CLSID* pCLSID = NULL
) throw( );
```

Parameters

bUnRegTypeLib

TRUE if the type library is also to be unregistered.

pCLSID

Points to the CLSID of the object to be unregistered. If NULL (the default value), all objects in the object map will be unregistered.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CAtlModuleT Class

CAtlModuleT::RegisterServer

Other Resources

CAtlModuleT Members

CAtlModuleT Static Functions

For information about the static functions in **CAtlModuleT**, see CAtlModuleT Members.

CAtlModuleT::InitLibId

Initializes the data member containing the GUID of the current module.

```
static void InitLibId( ) throw( );
```

Remarks

Called by the constructor CAtlModuleT::CAtlModuleT.

See Also

Reference

CAtlModuleT Class

Other Resources

CAtlModuleT Members

CAtlModuleT::UpdateRegistryAppId

Updates the EXE information in the registry.

```
static HRESULT WINAPI UpdateRegistryAppId(
   BOOL /*bRegister*/
) throw( );
```

Parameters

bRegister

Reserved.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CAtlModuleT Class

Other Resources

CAtlModuleT Members

CAtlServiceModuleT Class

This class implements a service.

```
template <
   class T,
   UINT nServiceNameID
>
class ATL_NO_VTABLE CAtlServiceModuleT :
   public CAtlExeModuleT< T >
```

Parameters

T

Your class derived from **CAtlServiceModuleT**.

nServiceNameID

The resource identifier of the service.

Remarks

CAtlServiceModuleT, derived from CAtlExeModuleT, implements a ATL Service module. **CAtlServiceModuleT** provides methods for command-line processing, installation, registering, and removal. If extra functionality is required, these and other methods can be overridden.

This class replaces the obsolete CComModule Class used in earlier versions of ATL. See ATL Module Classes for more details.

Requirements

Header: atlbase.h

See Also

Reference

CAtlExeModuleT Class

Other Resources

CAtlServiceModuleT Members

ATL Class Overview

CAtlServiceModuleT Members

Methods

CAtlServiceModuleT	The constructor.
Handler	The handler routine for the service.
InitializeSecurity	Provides the default security settings for the service.
Install	Installs and creates the service.
IsInstalled	Confirms that the service has been installed.
LogEvent	Writes to the event log.
ParseCommandLine	Parses the command line and performs registration if necessary.
PreMessageLoop	This method is called immediately before entering the message loop.
OnContinue	Override this method to continue the service.
OnInterrogate	Override this method to interrogate the service.
OnPause	Override this method to pause the service.
OnShutdown	Override this method to shut down the service
OnStop	Override this method to stop the service
OnUnknownRequest	Override this method to handle unknown requests to the service
RegisterAppId	Registers the service in the registry.
Run	Runs the service.
ServiceMain	The method called by the Service Control Manager.
SetServiceStatus	Updates the service status.
Start	Called by **CAtlServiceModuleT::WinMain** when the service starts.
Uninstall	Stops and removes the service.
Unlock	Decrements the service's lock count.
UnregisterAppId	Removes the service from the registry.
WinMain	This method implements the code required to run the service.

Data Members

m_bService	Flag indicating the program is running as a service.

m_dwThreadID	Member variable storing the thread identifier.
m_hServiceStatus	Member variable storing a handle to the status information structure for the current service.
m_status	Member variable storing the status information structure for the current service.
m_szServiceName	The name of the service being registered.

See Also

Reference

CAtlServiceModuleT Class

CAtlServiceModuleT Methods

For information about the methods in **CAtlServiceModuleT**, see CAtlServiceModuleT Members.

CAtlServiceModuleT::CAtlServiceModuleT

The constructor.

```
CAtlServiceModuleT( ) throw( );
```

Remarks

Initializes the data members and sets the initial service status.

See Also

Reference

CAtlServiceModuleT Class

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::Handler

The handler routine for the service.

```
void Handler(
   DWORD dwOpcode
) throw( );
```

Parameters

dwOpcode

A switch that defines the handler operation. For details, see the Remarks.

Remarks

This is the code that the Service Control Manager (SCM) calls to retrieve the status of the service and issue instructions such as stop or pause. The SCM passes an operation code, shown below, to **Handler** to indicate what the service should do.

Operation code	Meaning
SERVICE_CONTROL_STOP	Stops the service. Override the method CAtlServiceModuleT::OnStop in atlbase.h to change the behavior.
SERVICE_CONTROL_PAUSE	User implemented. Override the empty method CAtlServiceModuleT::OnPause in atlbase.h to pause the service.
SERVICE_CONTROL_CONTINUE	User implemented. Override the empty method CAtlServiceModuleT::OnContinue in atlbase.h to continue the service.
SERVICE_CONTROL_INTERROGATE	User implemented. Override the empty method CAtlServiceModuleT::OnInterrogate in atlbase.h to interrogate the service.
SERVICE_CONTROL_SHUTDOWN	User implemented. Override the empty method CAtlServiceModuleT::OnShutdown in atlbase.h to shutdown the service.

If the operation code isn't recognized, the method CAtlServiceModuleT::OnUnknownRequest is called.

A default ATL-generated service only handles the stop instruction. If the SCM passes the stop instruction, the service tells the SCM that the program is about to stop. The service then calls **PostThreadMessage** to post a quit message to itself. This terminates the message loop and the service will ultimately close.

See Also

Reference

CAtlServiceModuleT Class

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::InitializeSecurity

Provides the default security settings for the service.

```
HRESULT InitializeSecurity( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

In Visual Studio .NET 2003, this method is not implemented in the base class. The Visual Studio project wizard includes this method in the generated code, but a compilation error will occur if a project created in an earlier version of Visual C++ is compiled using ATL 7.1. Any class that derives from **CAtlServiceModuleT** must implement this method in the derived class. The method has to call CoInitializeSecurity and provide the appropriate security settings for the service.

Use PKT-level authentication, impersonation level of RPC_C_IMP_LEVEL_IDENTIFY and an appropriate non-null security descriptor in the call to **CoInitializeSecurity**.

For wizard-generated nonattributed service projects, this would be in

```
class CXXXModule : public CAtlServiceModuleT< CXXXModule, IDS_SERVICENAME > // XXX is the P
roject Name
{
public :
   DECLARE_LIBID(...)
   DECLARE_REGISTRY_APPID_RESOURCEID(...)
   HRESULT InitializeSecurity() throw()
   {
      // TODO : Call CoInitializeSecurity and provide the
      // appropriate security settings for your service.
      // Suggested - PKT Level Authentication,
      // Impersonation Level of RPC_C_IMP_LEVEL_IDENTIFY
      // and an appropiate Non NULL Security Descriptor.
      return S_OK;
   }
};
```

For attributed service projects, this would be in

```
[module(service, ...)] // Comment out any ; after the ]
class CXXXModule // XXX is the Project Name
{
public:
   HRESULT InitializeSecurity() throw()
   {
      // TODO : Call CoInitializeSecurity and provide the
      // appropriate security settings for your service.
      // Suggested - PKT Level Authentication,
      // Impersonation Level of RPC_C_IMP_LEVEL_IDENTIFY
      // and an appropiate Non NULL Security Descriptor.

      return S_OK;
   }
};
```

See Also

Reference

CAtlServiceModuleT Class

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::Install

Installs and creates the service.

```
BOOL Install( ) throw( );
```

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Installs the service into the Service Control Manager (SCM) database and then creates the service object. If the service could not be created, a message box is displayed and the method returns FALSE.

See Also

Reference

CAtlServiceModuleT Class
CreateService
CAtlServiceModuleT::Uninstall

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::IsInstalled

Confirms that the service has been installed.

```
BOOL IsInstalled( ) throw( );
```

Return Value

Returns TRUE if the service is installed, FALSE otherwise.

See Also

Reference

CAtlServiceModuleT Class

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::LogEvent

Writes to the event log.

```
void __cdecl LogEvent(
   LPCTSTR pszFormat,
   ...
) throw( );
```

Parameters

pszFormat

The string to write to the event log.

...

Optional extra strings to be written to the event log.

Remarks

This method writes details out to an event log, using the function ReportEvent. If no service is running, the string is sent to the console.

See Also

Reference

CAtlServiceModuleT Class

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::OnContinue

Override this method to continue the service.

```
void OnContinue( ) throw( );
```

See Also

Reference

CAtlServiceModuleT Class

CAtlServiceModuleT::Handler

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::OnInterrogate

Override this method to interrogate the service.

```
void OnInterrogate( ) throw( );
```

See Also

Reference

CAtlServiceModuleT Class

CAtlServiceModuleT::Handler

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::OnPause

Override this method to pause the service.

```
void OnPause( ) throw( );
```

See Also

Reference

CAtlServiceModuleT Class

CAtlServiceModuleT::Handler

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::OnShutdown

Override this method to shut down the service.

```
void OnShutdown( ) throw( );
```

See Also

Reference

CAtlServiceModuleT Class

CAtlServiceModuleT::Handler

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::OnStop

Override this method to stop the service.

```
void OnStop( ) throw( );
```

See Also

Reference

CAtlServiceModuleT Class

CAtlServiceModuleT::Handler

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::OnUnknownRequest

Override this method to handle unknown requests to the service.

```
void OnUnknownRequest(
   DWORD /* dwOpcode */
) throw( );
```

Parameters

/ dwOpcode */*

Reserved.

See Also

Reference

CAtlServiceModuleT Class

CAtlServiceModuleT::Handler

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::ParseCommandLine

Parses the command line and performs registration if necessary.

```
bool ParseCommandLine(
   LPCTSTR lpCmdLine,
   HRESULT* pnRetCode
) throw( );
```

Parameters

lpCmdLine

The command line.

pnRetCode

The HRESULT corresponding to the registration (if it took place).

Return Value

Returns true on success, or false if the RGS file supplied in the command line could not be registered.

Remarks

Parses the command line and registers or unregisters the supplied RGS file if necessary. This method calls CAtlExeModuleT::ParseCommandLine to check for **/RegServer** and **/UnregServer**. Adding the argument **-/Service** will register the service.

See Also

Reference

CAtlServiceModuleT Class

CAtlExeModuleT::ParseCommandLine

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::PreMessageLoop

This method is called immediately before entering the message loop.

```
HRESULT PreMessageLoop(
   int nShowCmd
) throw( );
```

Parameters

nShowCmd

This parameter is passed to CAtlExeModuleT::PreMessageLoop.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Override this method to add custom initialization code for the Service.

See Also

Reference

CAtlServiceModuleT Class

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::RegisterAppId

Registers the service in the registry.

```
inline HRESULT RegisterAppId(
   bool bService = false
) throw( );
```

Parameters

bService

Must be true to register as a service.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CAtlServiceModuleT Class

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::Run

Runs the service.

```
HRESULT Run(
   int nShowCmd = SW_HIDE
) throw( );
```

Parameters

nShowCmd

Specifies how the window is to be shown. This parameter can be one of the values discussed in the WinMain section. The default value is SW_HIDE.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

After being called, **Run** calls CAtlServiceModuleT::PreMessageLoop, CAtlExeModuleT::RunMessageLoop, and CAtlExeModuleT::PostMessageLoop.

See Also

Reference

CAtlServiceModuleT Class

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::ServiceMain

This method is called by the Service Control Manager.

```
void ServiceMain(
   DWORD dwArgc,
   LPTSTR* lpszArgv
) throw( );
```

Parameters

dwArgc

The argc argument.

lpszArgv

The argv argument.

Remarks

The Service Control Manager (SCM) calls **ServiceMain** when you open the Services application in the Control Panel, select the service, and click Start.

After the SCM calls **ServiceMain**, a service must give the SCM a handler function. This function lets the SCM obtain the service's status and pass specific instructions (such as pausing or stopping). Subsequently, CAtlServiceModuleT::Run is called to perform the main work of the service. **Run** continues to execute until the service is stopped.

See Also

Reference

CAtlServiceModuleT Class

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::SetServiceStatus

This method updates the service status.

```
void SetServiceStatus(
   DWORD dwState
) throw( );
```

Parameters

dwState

The new status. See SetServiceStatus for possible values.

Remarks

Updates the Service Control Manager's status information for the service. It is called by CAtlServiceModuleT::Run, CAtlServiceModuleT::ServiceMain and other handler methods. The status is also stored in the member variable CAtlServiceModuleT::m_status.

See Also

Reference

CAtlServiceModuleT Class

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::Start

Called by **CAtlServiceModuleT::WinMain** when the service starts.

```
HRESULT Start(
   int nShowCmd
) throw( );
```

Parameters

nShowCmd

Specifies how the window is to be shown. This parameter can be one of the values discussed in the WinMain section.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The CAtlServiceModuleT::WinMain method handles both registration and installation, as well as tasks involved in removing registry entries and uninstalling the module. When the service is run, **WinMain** calls **Start**.

See Also

Reference

CAtlServiceModuleT Class

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::Uninstall

Stops and removes the service.

```
BOOL Uninstall( ) throw( );
```

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Stops the service from running and removes it from the Service Control Manager database.

See Also

Reference

CAtlServiceModuleT Class

CAtlServiceModuleT::Install

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::Unlock

Decrements the service's lock count.

```
LONG Unlock( ) throw( );
```

Return Value

Returns the lock count, which may be useful for diagnostics and debugging.

See Also

Reference

CAtlServiceModuleT Class

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::UnregisterAppId

Removes the service from the registry.

```
HRESULT UnregisterAppId( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CAtlServiceModuleT Class

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::WinMain

This method implements the code required to start the service.

```
int WinMain(
   int nShowCmd
) throw( );
```

Parameters

nShowCmd

Specifies how the window is to be shown. This parameter can be one of the values discussed in the WinMain section.

Return Value

Returns the service's return value.

Remarks

This method processes the command line (with CAtlServiceModuleT::ParseCommandLine) and then starts the service (using CAtlServiceModuleT::Start).

See Also

Reference

CAtlServiceModuleT Class

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT Data Members

For information about the data members in **CAtlServiceModuleT**, see CAtlServiceModuleT Members.

CAtlServiceModuleT::m_bService

Flag indicating the program is running as a service.

```
BOOL m_bService;
```

Remarks

Used to distinguish a Service EXE from an Application EXE.

See Also

Reference

CAtlServiceModuleT Class

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::m_dwThreadID

Member variable storing the thread identifier of the Service.

```
DWORD m_dwThreadID;
```

Remarks

This variable stores the thread identifier of the current thread.

See Also

Reference

CAtlServiceModuleT Class

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::m_hServiceStatus

Member variable storing a handle to the status information structure for the current service.

```
SERVICE_STATUS_HANDLE m_hServiceStatus;
```

Remarks

The SERVICE_STATUS structure contains information about a service.

See Also

Reference

CAtlServiceModuleT Class

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::m_status

Member variable storing the status information structure for the current service.

```
SERVICE_STATUS m_status;
```

Remarks

The SERVICE_STATUS structure contains information about a service.

See Also

Reference

CAtlServiceModuleT Class

Other Resources

CAtlServiceModuleT Members

CAtlServiceModuleT::m_szServiceName

The name of the service being registered.

```
TCHAR [256] m_szServiceName;
```

Remarks

A null-terminated string which stores the name of the service.

See Also

Reference

CAtlServiceModuleT Class

Other Resources

CAtlServiceModuleT Members

CAtlTemporaryFile Class

This class provides methods for the creation and use of a temporary file.

```
class CAtlTemporaryFile
```

Remarks

CAtlTemporaryFile makes it easy to create and use a temporary file. The file is automatically named, opened, closed, and deleted. If the file contents are required after the file is closed, they can be saved to a new file with a specified name.

Requirements

Header: atlfile.h

Example

See the example for CAtlTemporaryFile::CAtlTemporaryFile.

See Also

Reference

CAtlFile Class

Other Resources

CAtlTemporaryFile Members

ATL Class Overview

CAtlTemporaryFile Members

Methods

CAtlTemporaryFile	The constructor.
~CAtlTemporaryFile	The destructor.
Close	Call this method to close a temporary file and either delete its contents or store them under the specified file name.
Create	Call this method to create a temporary file.
Flush	Call this method to force any data remaining in the file buffer to be written to the temporary file.
GetPosition	Call this method to get the current file pointer position.
GetSize	Call this method to get the size in bytes of the temporary file.
HandsOff	Call this method to disassociate the file from the **CAtlTemporaryFile** object.
HandsOn	Call this method to open an existing temporary file and position the pointer at the end of the file.
LockRange	Call this method to lock a region in the file to prevent other processes from accessing it.
Read	Call this method to read data from the temporary file starting at the position indicated by the file pointer.
Seek	Call this method to move the file pointer of the temporary file.
SetSize	Call this method to set the size of the temporary file.
TempFileName	Call this method to return the name of the temporary file.
UnlockRange	Call this method to unlock a region of the temporary file.
Write	Call this method to write data to the temporary file starting at the position indicated by the file pointer.

Operators

operator HANDLE	Returns a handle to the temporary file.

See Also

Reference

CAtlTemporaryFile Class

Other Resources

CAtlFile Members

CAtlTemporaryFile Methods

For information about the methods in **CAtlTemporaryFile**, see CAtlTemporaryFile Members.

CAtlTemporaryFile::CAtlTemporaryFile

The constructor.

```
CAtlTemporaryFile( ) throw( );
```

Remarks

A file is not actually opened until a call is made to CAtlTemporaryFile::Create.

Example

```
// CAtlTemporaryFile Example

#include "stdafx.h"
#include "atlfile.h"

int main(int argc, char* argv[])
{
   // Declare the temporary file object
   CAtlTemporaryFile myTempFile;

   // Create the temporary file, without caring where it
   // will be created, but with both read and write access.
   ATLVERIFY (myTempFile.Create(NULL,
      GENERIC_READ|GENERIC_WRITE) == S_OK);

   // Create some data to write to the file

   int nBuffer[100];
   DWORD bytes_written = 0, bytes_read = 0;
   int i;

   for (i = 0; i<100; i++)
      nBuffer[i] = i;

   // Write some data to the file
   myTempFile.Write(&nBuffer, sizeof(nBuffer), &bytes_written);

   // Confirm it was written ok
   ATLASSERT(bytes_written == sizeof(nBuffer));

   // Flush the data to disk
   ATLVERIFY (myTempFile.Flush() == S_OK);

   // Reset the file pointer to the beginning of the file
   ATLVERIFY (myTempFile.Seek(0, FILE_BEGIN) == S_OK);

   // Read in the data
   myTempFile.Read(&nBuffer, sizeof(nBuffer), bytes_read);

   // Confirm it was read ok
   ATLASSERT(bytes_read == sizeof(nBuffer));

   // Close the file, making a copy of it at another location
   ATLVERIFY (myTempFile.Close("c:/mydata.tmp")==S_OK);

   return 0;
}
```

See Also

Reference

CAtlTemporaryFile Class

CAtlTemporaryFile::~CAtlTemporaryFile

Other Resources

CAtlTemporaryFile Members

CAtlTemporaryFile::~CAtlTemporaryFile

The destructor.

```
~CAtlTemporaryFile( ) throw( );
```

Remarks

The destructor calls CAtlTemporaryFile::Close.

See Also

Reference

CAtlTemporaryFile Class

CAtlTemporaryFile::CAtlTemporaryFile

Other Resources

CAtlTemporaryFile Members

CAtlTemporaryFile::Close

Call this method to close a temporary file and either delete its contents or store them under the specified file name.

```
HRESULT Close(
   LPCTSTR szNewName = NULL
) throw( );
```

Parameters

szNewName

The name for the new file to store the contents of the temporary file in. If this argument is NULL, the contents of the temporary file are deleted.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

See the example for CAtlTemporaryFile::CAtlTemporaryFile.

See Also

Reference

CAtlTemporaryFile Class
CAtlTemporaryFile::HandsOff
CAtlTemporaryFile::Create

Other Resources

CAtlTemporaryFile Members

CAtlTemporaryFile::Create

Call this method to create a temporary file.

```
HRESULT Create(
   LPCTSTR pszDir = NULL,
   DWORD dwDesiredAccess = GENERIC_WRITE
) throw( );
```

Parameters

pszDir

The path for the temporary file. If this is NULL, GetTempPath will be called to assign a path.

dwDesiredAccess

The desired access. See *dwDesiredAccess* in CreateFile in the *Platform SDK*.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

See the example for CAtlTemporaryFile::CAtlTemporaryFile.

See Also

Reference

CAtlTemporaryFile Class

Other Resources

CAtlTemporaryFile Members

CAtlTemporaryFile::Flush

Call this method to force any data remaining in the file buffer to be written to the temporary file.

```
HRESULT Flush( ) throw( );
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Similar to CAtlTemporaryFile::HandsOff, except that the file is not closed.

Example

See the example for CAtlTemporaryFile::CAtlTemporaryFile.

See Also

Reference

CAtlTemporaryFile Class

Other Resources

CAtlTemporaryFile Members

CAtlTemporaryFile::GetPosition

Call this method to get the current file pointer position.

```
HRESULT GetPosition(
   ULONGLONG& nPos
) const throw( );
```

Parameters

nPos

The position in bytes.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

To change the file pointer position, use CAtlTemporaryFile::Seek.

See Also

Reference

CAtlTemporaryFile Class

CAtlFile::GetPosition

Other Resources

CAtlTemporaryFile Members

CAtlTemporaryFile::GetSize

Call this method to get the size in bytes of the temporary file.

```
HRESULT GetSize(
   ULONGLONG& nLen
) const throw( );
```

Parameters

nLen

The number of bytes in the file.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

See Also

Reference

CAtlTemporaryFile Class
CAtlFile::GetSize
CAtlTemporaryFile::SetSize

Other Resources

CAtlTemporaryFile Members

CAtlTemporaryFile::HandsOff

Call this method to disassociate the file from the **CAtlTemporaryFile** object.

```
HRESULT HandsOff( ) throw( );
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

HandsOff and CAtlTemporaryFile::HandsOn are used to disassociate the file from the object, and reattach it if needed. **HandsOff** will force any data remaining in the file buffer to be written to the temporary file, and then close the file. If you want to close and delete the file permanently, or if you want to close and retain the contents of the file with a given name, use CAtlTemporaryFile::Close.

See Also

Reference

CAtlTemporaryFile Class

CAtlTemporaryFile::HandsOn

Other Resources

CAtlTemporaryFile Members

CAtlTemporaryFile::HandsOn

Call this method to open an existing temporary file and position the pointer at the end of the file.

```
HRESULT HandsOn( ) throw( );
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

CAtlTemporaryFile::HandsOff and **HandsOn** are used to disassociate the file from the object, and reattach it if needed.

See Also

Reference

CAtlTemporaryFile Class

CAtlTemporaryFile::HandsOff

Other Resources

CAtlTemporaryFile Members

CAtlTemporaryFile::LockRange

Call this method to lock a region in the temporary file to prevent other processes from accessing it.

```
HRESULT LockRange(
   ULONGLONG nPos,
   ULONGLONG nCount
) throw( );
```

Parameters

nPos

The position in the file where the lock should begin.

nCount

The length of the byte range to be locked.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Locking bytes in a file prevents access to those bytes by other processes. You can lock more than one region of a file, but no overlapping regions are allowed. To successfully unlock a region, use CAtlTemporaryFile::UnlockRange, ensuring the byte range corresponds exactly to the region that was previously locked. **LockRange** does not merge adjacent regions; if two locked regions are adjacent, you must unlock each separately.

See Also

Reference

CAtlTemporaryFile Class
CAtlFile::LockRange
CAtlTemporaryFile::UnlockRange

Other Resources

CAtlTemporaryFile Members

CAtlTemporaryFile::Read

Call this method to read data from the temporary file starting at the position indicated by the file pointer.

```
HRESULT Read(
   LPVOID pBuffer,
   DWORD nBufSize,
   DWORD& nBytesRead
) throw( );
```

Parameters

pBuffer

Pointer to the buffer that will receive the data read from the file.

nBufSize

The buffer size in bytes.

nBytesRead

The number of bytes read.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Calls CAtlFile::Read. To change the position of the file pointer, call CAtlTemporaryFile::Seek.

Example

See the example for CAtlTemporaryFile::CAtlTemporaryFile.

See Also

Reference

CAtlTemporaryFile Class
CAtlTemporaryFile::Write

Other Resources

CAtlTemporaryFile Members

CAtlTemporaryFile::Seek

Call this method to move the file pointer of the temporary file.

```
HRESULT Seek(
   LONGLONG nOffset,
   DWORD dwFrom = FILE_CURRENT
) throw( );
```

Parameters

nOffset

The offset, in bytes, from the starting point given by *dwFrom*.

dwFrom

The starting point (FILE_BEGIN, FILE_CURRENT, or FILE_END).

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Calls CAtlFile::Seek. To obtain the current file pointer position, call CAtlTemporaryFile::GetPosition.

Example

See the example for CAtlTemporaryFile::CAtlTemporaryFile.

See Also

Reference

CAtlTemporaryFile Class
CAtlTemporaryFile::Read

Other Resources

CAtlTemporaryFile Members

CAtlTemporaryFile::SetSize

Call this method to set the size of the temporary file.

```
HRESULT SetSize(
   ULONGLONG nNewLen
) throw( );
```

Parameters

nNewLen

The new length of the file in bytes.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Calls CAtlFile::SetSize. On return, the file pointer is positioned at the end of the file.

See Also

Reference

CAtlTemporaryFile Class

CAtlTemporaryFile::GetSize

Other Resources

CAtlTemporaryFile Members

CAtlTemporaryFile::TempFileName

Call this method to return the name of temporary file.

```
LPCTSTR TempFileName( ) throw( );
```

Return Value

Returns the **LPCTSTR** pointing to the file name.

Remarks

The file name is generated in CAtlTemporaryFile::CAtlTemporaryFile with a call to the GetTempFile *Platform SDK* function. The file extension will always be "TFR" for the temporary file.

See Also

Reference

CAtlTemporaryFile Class

Other Resources

CAtlTemporaryFile Members

CAtlTemporaryFile::UnlockRange

Call this method to unlock a region of the temporary file.

```
HRESULT UnlockRange(
   ULONGLONG nPos,
   ULONGLONG nCount
) throw( );
```

Parameters

nPos

The position in the file where the unlock should begin.

nCount

The length of the byte range to be unlocked.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Calls CAtlFile::UnlockRange.

See Also

Reference

CAtlTemporaryFile Class

CAtlTemporaryFile::LockRange

Other Resources

CAtlTemporaryFile Members

CAtlTemporaryFile::Write

Call this method to write data to the temporary file starting at the position indicated by the file pointer.

```
HRESULT Write(
   LPCVOID pBuffer,
   DWORD nBufSize,
   DWORD* pnBytesWritten = NULL
) throw( );
```

Parameters

pBuffer

The buffer containing the data to be written to the file.

nBufSize

The number of bytes to be transferred from the buffer.

pnBytesWritten

The number of bytes written.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Calls CAtlFile::Write.

Example

See the example for CAtlTemporaryFile::CAtlTemporaryFile.

See Also

Reference

CAtlTemporaryFile Class
CAtlTemporaryFile::Read

Other Resources

CAtlTemporaryFile Members

CAtlTemporaryFile Operators

For information about the operators in **CAtlTemporaryFile**, see CAtlTemporaryFile Members.

CAtlTemporaryFile::operator HANDLE

Returns a handle to the temporary file.

```
operator HANDLE( ) throw( );
```

See Also

Reference

CAtlTemporaryFile Class

Other Resources

CAtlTemporaryFile Members

CAtlWinModule Class

This class provides support for ATL windowing components.

```
class CAtlWinModule : public _ATL_WIN_MODULE
```

Remarks

This class provides support for all ATL classes which require windowing features.

Requirements

Header: atlbase.h

See Also

Reference

_ATL_WIN_MODULE

ATL Module Classes

Other Resources

CAtlWinModule Members

ATL Class Overview

CAtlWinModule Members

Methods

AddCreateWndData	Adds a data object.
CAtlWinModule	The constructor.
~CAtlWinModule	The destructor.
ExtractCreateWndData	Returns a pointer to the window module data object.

See Also

Reference

CAtlWinModule Class

CAtlWinModule Methods

For information about the methods in **CAtlWinModule**, see CAtlWinModule Members.

CAtlWinModule::AddCreateWndData

This method initializes and adds an **_AtlCreateWndData** structure.

```
void AddCreateWndData(
   _AtlCreateWndData* pData,
   void* pObject
);
```

Parameters

pData

Pointer to the **_AtlCreateWndData** structure to be initialized and added to the current module.

pObject

Pointer to an object's **this** pointer.

Remarks

This method calls AtlWinModuleAddCreateWndData which initializes an _AtlCreateWndData structure. This structure will store the **this** pointer, used to obtain the class instance in window procedures.

See Also

Reference

CAtlWinModule Class

CAtlWinModule::ExtractCreateWndData

Other Resources

CAtlWinModule Members

CAtlWinModule::CAtlWinModule

The constructor.

```
CAtlWinModule( );
```

Remarks

If initialization fails, an **EXCEPTION_NONCONTINUABLE** exception is raised.

See Also

Reference

CAtlWinModule Class

CAtlWinModule::~CAtlWinModule

Other Resources

CAtlWinModule Members

CAtlWinModule::~CAtlWinModule

The destructor.

```
~CAtlWinModule( );
```

Remarks

Frees all allocated resources.

See Also

Reference

CAtlWinModule Class

CAtlWinModule::CAtlWinModule

Other Resources

CAtlWinModule Members

CAtlWinModule::ExtractCreateWndData

This method returns a pointer to an **_AtlCreateWndData** structure.

```
void* ExtractCreateWndData( );
```

Return Value

Returns a pointer to the **_AtlCreateWndData** structure previously added with CAtlWinModule::AddCreateWndData, or NULL if no object is available.

See Also

Reference

CAtlWinModule Class

CAtlWinModule::AddCreateWndData

Other Resources

CAtlWinModule Members

CAutoPtr Class

This class represents a smart pointer object.

```
template<
   typename T
>
class CAutoPtr
```

Parameters

T

The pointer type.

Remarks

This class provides methods for creating and managing a smart pointer, which will help protect against memory leaks by automatically freeing resources when it falls out of scope.

Further, **CAutoPtr**'s copy constructor and assignment operator transfer ownership of the pointer, copying the source pointer to the destination pointer and setting the source pointer to NULL. It is therefore impossible to have two **CAutoPtr** objects each storing the same pointer, and this reduces the possibility of deleting the same pointer twice.

CAutoPtr also simplifies the creation of collections of pointers. Instead of deriving a collection class and overriding the destructor, it's simpler to make a collection of **CAutoPtr** objects. When the collection is deleted, the **CAutoPtr** objects will go out of scope and automatically delete themselves.

CHeapPtr and variants work in the same way as **CAutoPtr**, except that they allocate and free memory using different heap functions instead of the C++ **new** and **delete** operators. CAutoVectorPtr is similar to **CAutoPtr**, the only difference being that it uses **vector new[]** and **vector delete[]** to allocate and free memory.

See also CAutoPtrArray and CAutoPtrList when arrays or lists of smart pointers are required.

Requirements

Header: atlbase.h

Example

```
// CAutoPtr

#include "stdafx.h"
#include "atlbase.h"

// A simple class for demonstration purposes

class MyClass
{
   int iA;
   int iB;
public:
   MyClass(int a, int b);
   void Test();
};

MyClass::MyClass(int a, int b)
{
   iA = a;
   iB = b;
}

void MyClass::Test()
{
   ATLASSERT(iA == iB);
}
```

```
// A simple function

void MyFunction(MyClass * c)
{
   c->Test();
}

int main()
{
   // Create an object of MyClass.
   MyClass *pMyC = new MyClass(1, 1);

   // Create a CAutoPtr object and have it take
   // over the pMyC pointer by calling Attach.
   CAutoPtr <MyClass> apMyC;
   apMyC.Attach(pMyC);

   // The overloaded -> operator allows the
   // CAutoPtr object to be used in place of the pointer.
   apMyC->Test();

   // Assign a second CAutoPtr, using the = operator.
   CAutoPtr <MyClass> apMyC2;
   apMyC2 = apMyC;   // apMyC releases ownership of pointer

   // The casting operator allows the CAutoPtr
   // object to be used in place of the pointer.
   MyFunction(pMyC);
   MyFunction(apMyC2);

   return 0;
}  // CAutoPtr destroys any object it controls when it
   // goes out of scope, so apMyC2 destroys the object
   // pointed to by pMyC here.
```

For another example, see the ISAPIFilter Sample.

See Also

Reference

CHeapPtr Class

CAutoVectorPtr Class

Other Resources

CAutoPtr Members

ATL Class Overview

CAutoPtr Members

Methods

Attach	Call this method to take ownership of an existing pointer.
CAutoPtr	The constructor.
~CAutoPtr	The destructor.
Detach	Call this method to release ownership of a pointer.
Free	Call this method to delete an object pointed to by a **CAutoPtr**.

Operators

operator ->	The pointer-to-member operator.
operator =	The assignment operator.
operator T*	The cast operator.

Data Members

m_p	The pointer data member variable.

See Also

Reference

CAutoPtr Class

CAutoPtr Methods

For information about the methods in **CAutoPtr**, see CAutoPtr Members.

CAutoPtr::Attach

Call this method to take ownership of an existing pointer.

```
void Attach(
   T* p
) throw( );
```

Parameters

p

The **CAutoPtr** object will take ownership of this pointer.

Remarks

When a **CAutoPtr** object takes ownership of a pointer, it will automatically delete the pointer and any allocated data when it goes out of scope. If CAutoPtr::Detach is called, the programmer is again given responsibility for freeing any allocated resources.

In debug builds, an assertion failure will occur if the CAutoPtr::m_p data member currently points to an existing value; that is, it is not equal to NULL.

Example

See the example in the CAutoPtr Overview and see the ISAPIFilter Sample.

See Also

Reference

CAutoPtr Class
CAutoPtr::Detach
CAutoPtr::operator =

Other Resources

CAutoPtr Members

CAutoPtr::CAutoPtr

The constructor.

```
CAutoPtr( ) throw( );
explicit CAutoPtr(
   T* p
) throw( );
template< typename TSrc > CAutoPtr(
   CAutoPtr< TSrc >& p
) throw( );
template< > CAutoPtr(
   CAutoPtr< T >& p
) throw( );
```

Parameters

p

An existing pointer.

TSrc

The type being managed by another **CAutoPtr**, used to initialize the current object.

Remarks

The **CAutoPtr** object can be created using an existing pointer, in which case it transfers ownership of the pointer.

Example

See the example in the CAutoPtr Overview.

See Also

Reference

CAutoPtr Class

CAutoPtr::~CAutoPtr

Other Resources

CAutoPtr Members

CAutoPtr::~CAutoPtr

The destructor.

```
~CAutoPtr( ) throw( );
```

Remarks

Frees any allocated resources. Calls CAutoPtr::Free.

See Also

Reference

CAutoPtr Class

CAutoPtr::CAutoPtr

Other Resources

CAutoPtr Members

CAutoPtr::Detach

Call this method to release ownership of a pointer.

```
T* Detach( ) throw( );
```

Return Value

Returns a copy of the pointer.

Remarks

Releases ownership of a pointer, sets the CAutoPtr::m_p data member variable to NULL, and returns a copy of the pointer. After calling **Detach**, it is up to the programmer to free any allocated resources over which the **CAutoPtr** object may have previously assumed reponsibility.

Example

See the example in the CAutoPtr Overview.

See Also

Reference

CAutoPtr Class

CAutoPtr::Attach

Other Resources

CAutoPtr Members

CAutoPtr::Free

Call this method to delete an object pointed to by a **CAutoPtr**.

```
void Free( ) throw( );
```

Remarks

The object pointed to by the **CAutoPtr** is freed, and the CAutoPtr::m_p data member variable is set to NULL.

See Also

Reference

CAutoPtr Class

Other Resources

CAutoPtr Members

CAutoPtr Operators

For information about the operators in **CAutoPtr**, see CAutoPtr Members.

CAutoPtr::operator =

The assignment operator.

```
template< >
CAutoPtr< T > & operator =(
   CAutoPtr< T > & p
);
template< typename TSrc >
CAutoPtr< T > & operator =(
   CAutoPtr< TSrc > & p
);
```

Parameters

p

A pointer.

TSrc

A class type.

Return Value

Returns a reference to a **CAutoPtr< T >**.

Remarks

The assignment operator detaches the **CAutoPtr** object from any current pointer and attaches the new pointer, *p*, in its place.

Example

See the example in the CAutoPtr Overview.

See Also

Reference

CAutoPtr Class
CAutoPtr::Attach
CAutoPtr::Detach

Other Resources

CAutoPtr Members

CAutoPtr::operator ->

The pointer-to-member operator.

```
T * operator ->( ) const throw( );
```

Return Value

Returns the value of the CAutoPtr::m_p data member variable.

Remarks

Use this operator to call a method in a class pointed to by the **CAutoPtr** object. In debug builds, an assertion failure will occur if the **CAutoPtr** points to NULL.

Example

See the example in the CAutoPtr Overview.

See Also

Reference

CAutoPtr Class

Other Resources

CAutoPtr Members

CAutoPtr::operator T*

The cast operator.

```
operator T* ( ) const throw( );
```

Return Value

Returns a pointer to the object data type defined in the class template.

Example

See the example in the CAutoPtr Overview.

See Also

Reference

CAutoPtr Class

Other Resources

CAutoPtr Members

CAutoPtr Data Members

For information about the data members in **CAutoPtr**, see CAutoPtr Members.

CAutoPtr::m_p

The pointer data member variable.

```
T * m_p;
```

Remarks

This member variable holds the pointer information.

See Also

Reference

CAutoPtr Class

Other Resources

CAutoPtr Members

CAutoPtrArray Class

This class provides methods useful when constructing an array of smart pointers.

```
template<
   typename E
>
class CAutoPtrArray : public CAtlArray<
   ATL::CAutoPtr< E >,
   CAutoPtrElementTraits< E >
>
```

Parameters

E

The pointer type.

Remarks

This class provides a constructor and derives methods from CAtlArray and CAutoPtrElementTraits to aid the creation of a collection class object storing smart pointers.

For more information, see ATL Collection Classes.

Requirements

Header: atlcoll.h

Example

See the ISAPIFilter Sample.

See Also

Reference

CAtlArray Class
CAutoPtrElementTraits Class
CAutoPtrList Class

Other Resources

CAutoPtrArray Members
ATL Class Overview

CAutoPtrArray Members

Methods

CAutoPtrArray	The constructor.

See Also

Reference

CAutoPtrArray Class

Other Resources

CAtlArray Members

CAutoPtrElementTraits Members

CAutoPtrArray Methods

For information about the methods in **CAutoPtrArray**, see CAutoPtrArray Members.

CAutoPtrArray::CAutoPtrArray

The constructor.

```
CAutoPtrArray( ) throw( );
```

Remarks

Initializes the smart pointer array.

See Also

Reference

CAutoPtrArray Class

Other Resources

CAutoPtrArray Members

CAutoPtrElementTraits Class

This class provides methods, static functions, and typedefs useful when creating collections of smart pointers.

```
template<
   typename T
>
class CAutoPtrElementTraits : public CDefaultElementTraits<
   ATL::CAutoPtr< T >
>
```

Parameters

T

The pointer type.

Remarks

This class provides methods, static functions, and typedefs for aiding the creation of collection class objects containing smart pointers. The classes CAutoPtrArray and CAutoPtrList derive from **CAutoPtrElementTraits**. If building a collection of smart pointers that requires vector new and delete operators, use CAutoVectorPtrElementTraits instead.

Requirements

Header: atlcoll.h

See Also

Reference

CDefaultElementTraits Class

Other Resources

CAutoPtrElementTraits Members

ATL Class Overview

CAutoPtrElementTraits Members

Typedefs

INARGTYPE	The data type to use for adding elements to the collection class object.
OUTARGTYPE	The data type to use for retrieving elements from the collection class object.

See Also

Reference

CAutoPtrElementTraits Class

CAutoPtrElementTraits Typedefs

For information about the typedefs in **CAutoPtrElementTraits**, see CAutoPtrElementTraits Members.

CAutoPtrElementTraits::INARGTYPE

The data type to use for adding elements to the collection class object.

```
typedef CAutoPtr<T> & INARGTYPE;
```

See Also

Reference

CAutoPtrElementTraits Class

CAutoPtrElementTraits::OUTARGTYPE

Other Resources

CAutoPtrElementTraits Members

CAutoPtrElementTraits::OUTARGTYPE

The data type to use for retrieving elements from the collection class object.

```
typedef T *& OUTARGTYPE;
```

See Also

Reference

CAutoPtrElementTraits Class

CAutoPtrElementTraits::INARGTYPE

Other Resources

CAutoPtrElementTraits Members

CAutoPtrList Class

This class provides methods useful when constructing a list of smart pointers.

```
template<
   typename E
>
class CAutoPtrList : public CAtlList<
   ATL::CAutoPtr< E >,
   CAutoPtrElementTraits< E >
>
```

Parameters

E

The pointer type.

Remarks

This class provides a constructor and derives methods from CAtlList and CAutoPtrElementTraits to aid the creation of a list object storing smart pointers. The class CAutoPtrArray provides a similar function for an array object.

For more information, see ATL Collection Classes.

Requirements

Header: atlcoll.h

See Also

Reference

CAtlList Class

CAutoPtrElementTraits Class

Other Resources

CAutoPtrList Members

ATL Class Overview

CAutoPtrList Members

Methods

CAutoPtrList	The constructor.

See Also

Reference

CAutoPtrList Class

Other Resources

CAtlList Members

CAutoPtrElementTraits Members

CAutoPtrList Methods

For information about the methods in **CAutoPtrList**, see CAutoPtrList Members.

CAutoPtrList::CAutoPtrList

The constructor.

```
CAutoPtrList(
   UINT nBlockSize = 10
) throw( );
```

Parameters

nBlockSize

The block size, with a default of 10.

Remarks

The block size is a measure of the amount of memory allocated when a new element is required. Larger block sizes reduce calls to memory allocation routines, but use more resources.

See Also

Reference

CAutoPtrList Class

Other Resources

CAutoPtrList Members

CAutoRevertImpersonation Class

This class reverts CAccessToken objects to a nonimpersonating state when it goes out of scope.

```
class CAutoRevertImpersonation
```

Remarks

An access token is an object that describes the security context of a process or thread and is allocated to each user logged onto a Windows NT or Windows 2000 system. These access tokens can be represented with the **CAccessToken** class.

It is sometimes necessary to impersonate access tokens. This class is provided as a convenience, but it does not perform the impersonation of access tokens; it only performs the automatic reversion to a nonimpersonated state. This is because token access impersonation can be performed several different ways.

For an introduction to the access control model in Windows, see Access Control in the *Platform SDK*.

Requirements

Header: atlsecurity.h

See Also

Tasks

ATLSecurity Sample: Demonstrates ATL Security Classes

Reference

Access Tokens

Other Resources

CAutoRevertImpersonation Members

ATL Class Overview

CAutoRevertImpersonation Members

Methods

Attach	Automates the impersonation reversion of an access token.
CAutoRevertImpersonation	Constructs an **CAutoRevertImpersonation** object
~CAutoRevertImpersonation	Destroys the object and reverts access token impersonation.
Detach	Cancels the automatic impersonation reversion.
GetAccessToken	Retrieves the access token current associated with this object.

See Also

Reference

CAutoRevertImpersonation Class
CAccessToken Class

CAutoRevertImpersonation Methods

For information about the methods in **CAutoRevertImpersonation**, see CAutoRevertImpersonation Members

CAutoRevertImpersonation::Attach

Automates the impersonation reversion of an access token.

```
void Attach(
   const CAccessToken* pAT
) throw( );
```

Parameters

pAT

The address of the CAccessToken object to be reverted automatically

Remarks

This method should only be used if the CAutoRevertImpersonation object was created with a NULL **CAccessToken** pointer, or if Detach was called previously. For simple cases, it is not necessary to use this method.

See Also

Reference

CAutoRevertImpersonation Class

CAccessToken Class

Other Resources

CAutoRevertImpersonation Members

CAutoRevertImpersonation::CAutoRevertImpersonation

Constructs a **CAutoRevertImpersonation** object.

```
CAutoRevertImpersonation(
   const CAccessToken* pAT
) throw( );
```

Parameters

pAT

The address of the CAccessToken object to be reverted automatically.

Remarks

The actual impersonation of the access token should be performed separately from and preferably before the creation of a **CAutoRevertImpersonation** object. This impersonation will be reverted automatically when the **CAutoRevertImpersonation** object goes out of scope.

See Also

Reference

CAutoRevertImpersonation Class

CAccessToken Class

Other Resources

CAutoRevertImpersonation Members

CAutoRevertImpersonation::~CAutoRevertImpersonation

Destroys the object and reverts access token impersonation.

```
~CAutoRevertImpersonation() throw( );
```

Remarks

Reverts any impersonation currently in effect for the CAccessToken object provided either at construction or through the Attach method. If no **CAccessToken** is associated, the destructor has no effect.

See Also

Reference

CAutoRevertImpersonation Class

CAccessToken Class

Other Resources

CAutoRevertImpersonation Members

CAutoRevertImpersonation::Detach

Cancels the automatic impersonation reversion.

```
const CAccessToken* Detach() throw( );
```

Return Value

The address of the previously associated CAccessToken, or NULL if no association existed.

Remarks

Calling **Detach** prevents the **CAutoRevertImpersonation** object from reverting any impersonation currently in effect for the CAccessToken object associated with this object. **CAutoRevertImpersonation** can then be destroyed with no effect or reassociated to the same or another **CAccessToken** object using Attach.

See Also

Reference

CAutoRevertImpersonation Class

CAccessToken Class

Other Resources

CAutoRevertImpersonation Members

CAutoRevertImpersonation::GetAccessToken

Retrieves the access token current associated with this object.

```
const CAccessToken* GetAccessToken() throw( );
```

Return Value

The address of the previously associated CAccessToken, or NULL if no association existed.

Remarks

If this method is called for the purposes that include the reversion of an impersonation of the **CAccessToken** object, the Detach method should be used instead.

See Also

Reference

CAutoRevertImpersonation Class

CAccessToken Class

Other Resources

CAutoRevertImpersonation Members

CAutoVectorPtr Class

This class represents a smart pointer object using vector new and delete operators.

```
template<
   typename T
> class CAutoVectorPtr
```

Parameters

T

The pointer type.

Remarks

This class provides methods for creating and managing a smart pointer, which will help protect against memory leaks by automatically freeing resources when it falls out of scope. **CAutoVectorPtr** is similar to **CAutoPtr**, the only difference being that **CAutoVectorPtr** uses vector new[] and vector delete[] to allocate and free memory instead of the C++ **new** and **delete** operators. See CAutoVectorPtrElementTraits if collection classes of **CAutoVectorPtr** are required.

See CAutoPtr for an example of using a smart pointer class.

Requirements

Header: atlbase.h

See Also

Reference

CAutoPtr Class

Other Resources

CAutoVectorPtr Members

ATL Class Overview

CAutoVectorPtr Members

Methods

Allocate	Call this method to allocate the memory required by the array of objects pointed to by **CAutoVectorPtr**.
Attach	Call this method to take ownership of an existing pointer.
CAutoVectorPtr	The constructor.
~CAutoVectorPtr	The destructor.
Detach	Call this method to release ownership of a pointer.
Free	Call this method to delete an object pointed to by a **CAutoVectorPtr**.

Operators

operator =	The assignment operator.
operator T *	The cast operator.

Data Members

m_p	The pointer data member variable.

See Also

Reference

CAutoVectorPtr Class

CAutoVectorPtr Methods

For information about the methods in **CAutoVectorPtr**, see CAutoVectorPtr Members.

CAutoVectorPtr::Allocate

Call this method to allocate the memory required by the array of objects pointed to by **CAutoVectorPtr**.

```
bool Allocate(
   size_t nElements
) throw( );
```

Parameters

nElements

The number of elements in the array.

Return Value

Returns true if the memory is successfully allocated, false on failure.

Remarks

In debug builds, an assertion failure will occur if the CAutoVectorPtr::m_p member variable currently points to an existing value; that is, it is not equal to NULL.

See Also

Reference

CAutoVectorPtr Class

Other Resources

CAutoVectorPtr Members

CAutoVectorPtr::Attach

Call this method to take ownership of an existing pointer.

```
void Attach(
   T* p
) throw( );
```

Parameters

p

The **CAutoVectorPtr** object will take ownership of this pointer.

Remarks

When a **CAutoVectorPtr** object takes ownership of a pointer, it will automatically delete the pointer and any allocated data when it goes out of scope. If CAutoVectorPtr::Detach is called, the programmer is again given responsibility for freeing any allocated resources.

In debug builds, an assertion failure will occur if the CAutoVectorPtr::m_p member variable currently points to an existing value; that is, it is not equal to NULL.

See Also

Reference

CAutoVectorPtr Class
CAutoVectorPtr::Detach
CAutoVectorPtr::operator =

Other Resources

CAutoVectorPtr Members

CAutoVectorPtr::CAutoVectorPtr

The constructor.

```
CAutoVectorPtr( ) throw( );
explicit CAutoVectorPtr(
   T* p
) throw( );
CAutoVectorPtr(
   CAutoVectorPtr< T >& p
) throw( );
```

Parameters

p

An existing pointer.

Remarks

The **CAutoVectorPtr** object can be created using an existing pointer, in which case it transfers ownership of the pointer.

See Also

Reference

CAutoVectorPtr Class

CAutoVectorPtr::~CAutoVectorPtr

Other Resources

CAutoVectorPtr Members

CAutoVectorPtr::~CAutoVectorPtr

The destructor.

```
~CAutoVectorPtr( ) throw( );
```

Remarks

Frees any allocated resources. Calls CAutoVectorPtr::Free.

See Also

Reference

CAutoVectorPtr Class

CAutoVectorPtr::CAutoVectorPtr

Other Resources

CAutoVectorPtr Members

CAutoVectorPtr::Detach

Call this method to release ownership of a pointer.

```
T* Detach( ) throw( );
```

Return Value

Returns a copy of the pointer.

Remarks

Releases ownership of a pointer, sets the CAutoVectorPtr::m_p member variable to NULL, and returns a copy of the pointer. After calling **Detach**, it is up to the programmer to free any allocated resources over which the **CAutoVectorPtr** object may have previously assumed responsibility.

See Also

Reference

CAutoVectorPtr Class

CAutoVectorPtr::Attach

Other Resources

CAutoVectorPtr Members

CAutoVectorPtr::Free

Call this method to delete an object pointed to by a **CAutoVectorPtr**.

```
void Free( ) throw( );
```

Remarks

The object pointed to by the **CAutoVectorPtr** is freed, and the CAutoVectorPtr::m_p member variable is set to NULL.

See Also

Reference

CAutoVectorPtr Class

Other Resources

CAutoVectorPtr Members

CAutoVectorPtr Operators

For information about the operators in **CAutoVectorPtr**, see CAutoVectorPtr Members.

CAutoVectorPtr::operator =

The assignment operator.

```
CAutoVectorPtr< T >& operator =(
   CAutoVectorPtr< T >& p
) throw( );
```

Parameters

p

A pointer.

Return Value

Returns a reference to a **CAutoVectorPtr< T >**.

Remarks

The assignment operator detaches the **CAutoVectorPtr** object from any current pointer and attaches the new pointer, *p*, in its place.

See Also

Reference

CAutoVectorPtr Class
CAutoVectorPtr::Attach
CAutoVectorPtr::Detach

Other Resources

CAutoVectorPtr Members

CAutoVectorPtr::operator T *

The cast operator.

```
operator T*( ) const throw( );
```

Remarks

Returns a pointer to the object data type defined in the class template.

See Also

Reference

CAutoVectorPtr Class

Other Resources

CAutoVectorPtr Members

CAutoVectorPtr Data Members

For information about the data members in **CAutoVectorPtr**, see CAutoVectorPtr Members.

CAutoVectorPtr::m_p

The pointer data member variable.

```
T* m_p;
```

Remarks

This member variable holds the pointer information.

See Also

Reference

CAutoVectorPtr Class

Other Resources

CAutoVectorPtr Members

CAutoVectorPtrElementTraits Class

This class provides methods, static functions, and typedefs useful when creating collections of smart pointers using vector new and delete operators.

```
template<
   typename T
>
class CAutoVectorPtrElementTraits : public CDefaultElementTraits<
   ATL::CAutoVectorPtr< T >
>
```

Parameters

T

The pointer type.

Remarks

This class provides methods, static functions, and typedefs for aiding the creation of collection class objects containing smart pointers. Unlike CAutoPtrElementTraits, this class uses vector new and delete operators.

Requirements

Header: atlcoll.h

See Also

Reference

CDefaultElementTraits Class

CAutoVectorPtr Class

Other Resources

CAutoVectorPtrElementTraits Members

ATL Class Overview

CAutoVectorPtrElementTraits Members

Typedefs

INARGTYPE	The data type to use for adding elements to the collection class object.
OUTARGTYPE	The data type to use for retrieving elements from the collection class object.

See Also

Reference

CAutoVectorPtrElementTraits Class

CAutoVectorPtrElementTraits Typedefs

For information about the typedefs in **CAutoVectorPtrElementTraits**, see CAutoVectorPtrElementTraits Members.

CAutoVectorPtrElementTraits::INARGTYPE

The data type to use for adding elements to the collection class object.

```
typedef CAutoVectorPtr<T> & INARGTYPE;
```

See Also

Reference

CAutoVectorPtrElementTraits Class

CAutoVectorPtrElementTraits::OUTARGTYPE

Other Resources

CAutoVectorPtrElementTraits Members

CAutoVectorPtrElementTraits::OUTARGTYPE

The data type to use for retrieving elements from the collection class object.

```
typedef T *& OUTARGTYPE;
```

See Also

Reference

CAutoVectorPtrElementTraits Class

CAutoVectorPtrElementTraits::INARGTYPE

Other Resources

CAutoVectorPtrElementTraits Members

CAxDialogImpl Class

This class implements a dialog box (modal or modeless) that hosts ActiveX controls.

```
template <
   class T,
   class TBase = CWindow
>
class ATL_NO_VTABLE CAxDialogImpl :
   public CDialogImplBaseT< TBase >
```

Parameters

T

Your class, derived from **CAxDialogImpl**.

TBase

The base window class for **CDialogImplBaseT**.

Remarks

CAxDialogImpl allows you to create a modal or modeless dialog box. **CAxDialogImpl** provides the dialog box procedure, which uses the default message map to direct messages to the appropriate handlers.

CAxDialogImpl derives from **CDialogImplBaseT**, which in turn derives from *TBase* (by default, **CWindow**) and **CMessageMap**.

Your class must define an IDD member that specifies the dialog template resource ID. For example, adding an ATL Dialog object using the **Add Class** dialog box automatically adds the following line to your class:

```
enum { IDD = IDD_MYDIALOG };
```

where `MyDialog` is the **Short name** entered in the ATL Dialog Wizard.

See Implementing a Dialog Box for more information.

Note that an ActiveX control on a modal dialog box created with **CAxDialogImpl** will not support accelerator keys. To support accelerator keys on a dialog box created with **CAxDialogImpl**, create a modeless dialog box and, using your own message loop, use CAxDialogImpl::IsDialogMessage after getting a message from the queue to handle an accelerator key.

For more information on **CAxDialogImpl**, see ATL Control Containment FAQ.

Requirements

Header: atlwin.h

See Also

Reference

CDialogImpl Class

Other Resources

CAxDialogImpl Members

ATL Class Overview

CAxDialogImpl Members

Methods

AdviseSinkMap	Call this method to advise or unadvise all entries in the object's sink map event map.
Create	Call this method to create a modeless dialog box.
DestroyWindow	Call this method to destroy a modeless dialog box.
DoModal	Call this method to create a modal dialog box.
EndDialog	Call this method to destroy a modal dialog box.
GetDialogProc	Call this method to get a pointer to the **DialogProc** callback function.
GetIDD	Call this method to get the dialog template resource ID
IsDialogMessage	Call this method to determine whether a message is intended for this dialog box and, if it is, process the message.

Data Members

m_bModal	A variable that exists only in debug builds and is set to true if the dialog box is modal.

See Also

Reference

CAxDialogImpl Class

CAxDialogImpl Methods

For information about the methods in **CAxDialogImpl**, see CAxDialogImpl Members.

CAxDialogImpl::AdviseSinkMap

Call this method to advise or unadvise all entries in the object's sink map event map.

```
HRESULT AdviseSinkMap(
   bool bAdvise
);
```

Parameters

bAdvise

Set to true if all sink entries are to be advised; false if all sink entries are to be unadvised.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CAxDialogImpl Class

Other Resources

CAxDialogImpl Members

CAxDialogImpl::Create

Call this method to create a modeless dialog box.

```
HWND Create(
   HWND hWndParent,
   LPARAM dwInitParam = NULL
);
HWND Create(
   HWND hWndParent,
   RECT&,
   LPARAM dwInitParam = NULL
);
```

Parameters

hWndParent

[in] The handle to the owner window.

dwInitParam

[in] Specifies the value to pass to the dialog box in the *lParam* parameter of the **WM_INITDIALOG** message.

RECT&

This parameter is not used. This parameter is passed in by **CComControl**.

Return Value

The handle to the newly created dialog box.

Remarks

This dialog box is automatically attached to the **CAxDialogImpl** object. To create a modal dialog box, call DoModal.

The second override is provided only so dialog boxes can be used with CComControl.

See Also

Reference

CAxDialogImpl Class
CAxDialogImpl::DestroyWindow
CAxDialogImpl::DoModal

Other Resources

CAxDialogImpl Members

CAxDialogImpl::DestroyWindow

Call this method to destroy a modeless dialog box.

```
BOOL DestroyWindow( );
```

Return Value

TRUE if the window is successfully destroyed; otherwise FALSE.

Remarks

Do not call **DestroyWindow** to destroy a modal dialog box. Call EndDialog instead.

See Also

Reference

CAxDialogImpl Class

DestroyWindow

Other Resources

CAxDialogImpl Members

CAxDialogImpl::DoModal

Call this method to create a modal dialog box.

```
INT_PTR DoModal(
   HWND hWndParent = ::GetActiveWindow( ),
   LPARAM dwInitParam = NULL
);
```

Parameters

hWndParent

[in] The handle to the owner window. The default value is the return value of the GetActiveWindow Win32 function.

dwInitParam

[in] Specifies the value to pass to the dialog box in the *lParam* parameter of the **WM_INITDIALOG** message.

Return Value

If successful, the value of the *nRetCode* parameter specified in the call to EndDialog; otherwise, -1.

Remarks

This dialog box is automatically attached to the **CAxDialogImpl** object.

To create a modeless dialog box, call Create.

See Also

Reference

CAxDialogImpl Class
CAxDialogImpl::EndDialog
CAxDialogImpl::Create

Other Resources

CAxDialogImpl Members

CAxDialogImpl::EndDialog

Call this method to destroy a modal dialog box.

```
BOOL EndDialog(
   int nRetCode
);
```

Parameters

nRetCode

[in] The value to be returned by DoModal.

Return Value

TRUE if the dialog box is destroyed; otherwise, FALSE.

Remarks

EndDialog must be called through the dialog box procedure. After the dialog box is destroyed, Windows uses the value of *nRetCode* as the return value for **DoModal**, which created the dialog box.

Note
Do not call **EndDialog** to destroy a modeless dialog box. Call DestroyWindow instead.

See Also

Reference

CAxDialogImpl Class

CAxDialogImpl::DoModal

Other Resources

CAxDialogImpl Members

CAxDialogImpl::GetDialogProc

Call this method to get a pointer to the **DialogProc** callback function.

```
virtual DLGPROC GetDialogProc( );
```

Return Value

Returns a pointer to the **DialogProc** callback function.

Remarks

The **DialogProc** function is an application-defined callback function.

See Also

Reference

CAxDialogImpl Class

Other Resources

CAxDialogImpl Members

CAxDialogImpl::GetIDD

Call this method to get the dialog template resource ID.

```
int GetIDD( );
```

Return Value

Returns the dialog template resource ID.

See Also

Reference

CAxDialogImpl Class

Other Resources

CAxDialogImpl Members

CAxDialogImpl::IsDialogMessage

Call this method to determine whether a message is intended for this dialog box and, if it is, process the message.

```
BOOL IsDialogMessage(
   LPMSG pMsg
);
```

Parameters

pMsg

Pointer to a MSG structure that contains the message to be checked.

Return Value

Returns TRUE if the message has been processed, FALSE otherwise.

Remarks

This method is intended to be called from within a message loop.

See Also

Reference

CAxDialogImpl Class

Other Resources

CAxDialogImpl Members

CAxDialogImpl Data Members

For information about the data members in **CAxDialogImpl**, see CAxDialogImpl Members.

CAxDialogImpl::m_bModal

A variable that exists only in debug builds and is set to true if the dialog box is modal.

```
bool m_bModal;
```

See Also

Reference

CAxDialogImpl Class

Other Resources

CAxDialogImpl Members

CAxWindow Class

This class provides methods for manipulating a window hosting an ActiveX control.

```
class CAxWindow : public CWindow
```

Remarks

This class provides methods for manipulating a window that hosts an ActiveX control. The hosting is provided by "**AtlAxWin80**", which is wrapped by **CAxWindow**.

Class **CAxWindow** is implemented as a specialization of the **CAxWindowT** class. This specialization is declared as:

```
typedef CAxWindowT<CWindow> CAxWindow;
```

If you need to change the base class, you can use **CAxWindowT** and specify the new base class as a template argument.

Requirements

Header: atlwin.h

See Also

Tasks

ATLCON Sample: Demonstrates Creating a Simple Container

Reference

CWindow Class

Concepts

Composite Control Fundamentals

ATL Control Containment FAQ

Other Resources

CAxWindow Members

ATL Class Overview

CAxWindow Members

Methods

AttachControl	Attaches an existing ActiveX control to the **CAxWindow** object.
CAxWindow	Constructs a **CAxWindow** object.
CreateControl	Creates an ActiveX control, initializes it, and hosts it in the **CAxWindow** window.
CreateControlEx	Creates an ActiveX control and retrieves an interface pointer (or pointers) from the control.
QueryControl	Retrieves the **IUnknown** of the hosted ActiveX control.
QueryHost	Retrieves the **IUnknown** pointer of the **CAxWindow** object.
SetExternalDispatch	Sets the external dispatch interface used by the **CAxWindow** object.
SetExternalUIHandler	Sets the external **IDocHostUIHandler** interface used by the **CAxWindow** object.

Operators

operator =	Assigns an **HWND** to an existing **CAxWindow** object.

Static Functions

GetWndClassName	Retrieves the predefined class name of the **CAxWindow** object.

See Also

Reference

CAxWindow Class

Other Resources

CWindow Members

CAxWindow Methods

For information about the methods in **CAxWindow**, see CAxWindow Members.

CAxWindow::AttachControl

Creates a new host object if one isn't already present and attaches the specified control to the host.

```
HRESULT AttachControl(
   IUnknown* pControl,
   IUnknown** ppUnkContainer
);
```

Parameters

pControl

[in] A pointer to the **IUnknown** of the control.

ppUnkContainer

[out] A pointer to the **IUnknown** of the host (the **AxWin** object).

Return Value

A standard **HRESULT** value.

Remarks

The control object being attached must be correctly initialized before calling **AttachControl**.

See Also

Reference

CAxWindow Class

AtlAxAttachControl

Other Resources

CAxWindow Members

CAxWindow::CAxWindow

Constructs a **CAxWindow** object using an existing window object handle.

```
CAxWindow(
   HWND hWnd = NULL
);
```

Parameters

hWnd

A handle to an existing window object.

See Also

Reference

CAxWindow Class

Other Resources

CAxWindow Members

CAxWindow::CreateControl

Creates an ActiveX control, initializes it, and hosts it in the specified window.

```
HRESULT CreateControl(
   LPCOLESTR lpszName,
   IStream* pStream = NULL,
   IUnknown** ppUnkContainer = NULL
);
HRESULT CreateControl(
   DWORD dwResID,
   IStream* pStream = NULL,
   IUnknown** ppUnkContainer = NULL
);
```

Parameters

lpszName

A pointer to a string to create the control. Must be formatted in one of the following ways:

- A ProgID such as "MSCAL.Calendar.7"
- A CLSID such as "{8E27C92B-1264-101C-8A2F-040224009C02}"
- A URL such as "http://www.microsoft.com"
- A reference to an Active document such as "file://\\Documents\MyDoc.doc"
- A fragment of HTML such as "MSHTML:<HTML><BODY>This is a line of text</BODY></HTML>"

> **Note**
>
> "MSHTML:" must precede the HTML fragment so that it is designated as being an MSHTML stream. Only the ProgID and CLSID are supported in Windows Mobile platforms. Windows CE embedded platforms, other than Windows Mobile with support for CE IE support all types including ProgID, CLSID, URL, reference to active document, and fragment of HTML.

pStream

[in] A pointer to a stream that is used to initialize the properties of the control. Can be **NULL**.

ppUnkContainer

[out] The address of a pointer that will receive the **IUnknown** of the container. Can be **NULL**.

dwResID

The resource ID of an HTML resource. The WebBrowser control will be created and loaded with the specified resource.

Return Value

A standard **HRESULT** value.

Remarks

If the second version of this method is used, an HTML control is created and bound to the resource identified by *dwResID*.

This method gives you the same result as calling:

```
CreateControlEx( lpszName, hWnd, pStream, NULL, NULL, NULL, NULL );
```

See CAxWindow2T::CreateControlLic to create, initialize, and host a licensed ActiveX control.

Example

See Hosting ActiveX Controls Using ATL AXHost for a sample that uses **CreateControl**.

See Also

Reference

CAxWindow Class

AtlAxCreateControl

Other Resources

CAxWindow Members

CAxWindow::CreateControlEx

Creates an ActiveX control, initializes it, and hosts it in the specified window.

```
HRESULT CreateControlEx(
   LPCOLESTR lpszName,
   IStream* pStream = NULL,
   IUnknown** ppUnkContainer = NULL,
   IUnknown** ppUnkControl = NULL,
   REFIID iidSink = IID_NULL,
   IUnknown* punkSink = NULL
);
HRESULT CreateControlEx(
   DWORD dwResID,
   IStream* pStream = NULL,
   IUnknown** ppUnkContainer = NULL,
   IUnknown** ppUnkControl = NULL,
   REFIID iidSink = IID_NULL,
   IUnknown* punkSink = NULL
);
```

Parameters

lpszName

A pointer to a string to create the control. Must be formatted in one of the following ways:

- A ProgID such as "MSCAL.Calendar.7"
- A CLSID such as "{8E27C92B-1264-101C-8A2F-040224009C02}"
- A URL such as "http://www.microsoft.com"
- A reference to an Active document such as "file://\\Documents\MyDoc.doc"
- A fragment of HTML such as "MSHTML:<HTML><BODY>This is a line of text</BODY></HTML>"

Note
"MSHTML:" must precede the HTML fragment so that it is designated as being an MSHTML stream. Only the ProgID and CLSID are supported in Windows Mobile platforms. Windows CE embedded platforms, other than Windows Mobile with support for CE IE support all types including ProgID, CLSID, URL, reference to active document, and fragment of HTML.

pStream

[in] A pointer to a stream that is used to initialize the properties of the control. Can be **NULL**.

ppUnkContainer

[out] The address of a pointer that will receive the **IUnknown** of the container. Can be **NULL**.

ppUnkControl

[out] The address of a pointer that will receive the **IUnknown** of the control. Can be **NULL**.

iidSink

[in] The interface identifier of an outgoing interface on the contained object. Can be **IID_NULL**.

punkSink

[in] A pointer to the **IUnknown** interface of the sink object to be connected to the connection point on the contained object specified by *iidSink*.

dwResID

[in] The resource ID of an HTML resource. The WebBrowser control will be created and loaded with the specified resource.

Return Value

A standard **HRESULT** value.

Remarks

This method is similar to CAxWindow::CreateControl, but unlike that method, **CreateControlEx** also allows you to receive an interface pointer to the newly created control and set up an event sink to receive events fired by the control.

See CAxWindow2T::CreateControlLicEx to create, initialize, and host a licensed ActiveX control.

Example

See Hosting ActiveX Controls Using ATL AXHost for a sample that uses **CreateControlEx**.

See Also

Reference

CAxWindow Class

AtlAxCreateControlEx

Other Resources

CAxWindow Members

CAxWindow::QueryControl

Retrieves the specified interface of the hosted control.

```
HRESULT QueryControl(
   REFIID iid,
   void** ppUnk
);
template <class Q>
HRESULT QueryControl(
   Q** ppUnk
);
```

Parameters

iid

[in] Specifies the IID of the control's interface.

ppUnk

[out] A pointer to the interface of the control. In the template version of this method, there is no need for a reference ID as long as a typed interface with an associated UUID is passed.

Q

[in] The interface that is being queried for.

Return Value

A standard **HRESULT** value.

See Also

Reference

CAxWindow Class

CAxWindow::QueryHost

Other Resources

CAxWindow Members

CAxWindow::QueryHost

Returns the specified interface of the host.

```
HRESULT QueryHost(
   REFIID iid,
   void** ppUnk
);
template <class Q>
HRESULT QueryHost(
   Q** ppUnk
);
```

Parameters

iid

[in] Specifies the IID of the control's interface.

ppUnk

[out] A pointer to the interface on the host. In the template version of this method, there is no need for a reference ID as long as a typed interface with an associated UUID is passed.

Q

[in] The interface that is being queried for.

Return Value

A standard **HRESULT** value.

Remarks

The interface of the host allows access to the underlying functionality of the window-hosting code, implemented by **AxWin**.

See Also

Reference

CAxWindow Class

CAxWindow::QueryControl

Other Resources

CAxWindow Members

CAxWindow::SetExternalDispatch

Sets the external dispatch interface for the **CAxWindow** object.

```
HRESULT SetExternalDispatch(
   IDispatch* pDisp
);
```

Parameters

pDisp

[in] A pointer to an **IDispatch** interface.

Return Value

A standard **HRESULT** value.

See Also

Reference

CAxWindow Class

CAxWindow::SetExternalUIHandler

Other Resources

CAxWindow Members

CAxWindow::SetExternalUIHandler

Sets the external IDocHostUIHandlerDispatch interface for the **CAxWindow** object.

```
HRESULT SetExternalUIHandler(
   IDocHostUIHandlerDispatch* pUIHandler
);
```

Parameters

pUIHandler

[in] A pointer to an **IDocHostUIHandlerDispatch** interface.

Return Value

A standard **HRESULT** value.

Remarks

The external **IDocHostUIHandlerDispatch** interface is used by controls that query the host's site for the **IDocHostUIHandlerDispatch** interface. The WebBrowser control is one control that does this.

See Also

Reference

CAxWindow Class

CAxWindow::SetExternalDispatch

Other Resources

CAxWindow Members

CAxWindow Operators

For information about the operators in **CAxWindow**, see CAxWindow Members.

CAxWindow::operator =

Assigns an **HWND** to an existing **CAxWindow** object.

```
CAxWindow< TBase >& operator=(
   HWND hWnd
);
```

Parameters

hWnd

A handle to an existing window.

Return Value

Returns a reference to the current **CAxWindow** object.

See Also

Reference

CAxWindow Class

Other Resources

CAxWindow Members

CAxWindow Static Functions

For information about the static functions in **CAxWindow**, see CAxWindow Members.

CAxWindow::GetWndClassName

Retrieves the name of the window class.

```
static LPCTSTR GetWndClassName( );
```

Return Value

A pointer to a string containing the name of the window class that can host nonlicensed ActiveX controls.

See Also

Reference

CAxWindow Class

Other Resources

CAxWindow Members

CAxWindow2T Class

This class provides methods for manipulating a window that hosts an ActiveX control, and also has support for hosting licensed ActiveX controls.

```
template <
   class TBase = CWindow
>
class CAxWindow2T :
   public CAxWindowT< TBase >
```

Parameters

TBase

The class from which **CAxWindowT** derives.

Remarks

CAxWindow2T provides methods for manipulating a window that hosts an ActiveX control. **CAxWindow2T** also has support for hosting licensed ActiveX controls. The hosting is provided by "**AtlAxWinLic80**", which is wrapped by **CAxWindow2T**.

Class **CAxWindow2** is implemented as a specialization of the **CAxWindow2T** class. This specialization is declared as:

```
typedef CAxWindow2T <CWindow> CAxWindow2;
```

Note
CAxWindowT members are documented under CAxWindow.

See Hosting ActiveX Controls Using ATL AXHost for a sample that uses the members of this class.

Requirements

Header: atlwin.h

See Also

Concepts

ATL Control Containment FAQ

Other Resources

CAxWindow2T Members

ATL Class Overview

CAxWindow2T Members

Methods

CAxWindow2T	Constructs a **CAxWindow2T** object.
Create	Creates a host window.
CreateControlLic	Creates a licensed ActiveX control, initializes it, and hosts it in the specified window.
CreateControlLicEx	Creates a licensed ActiveX control, initializes it, hosts it in the specified window, and retrieves an interface pointer (or pointers) from the control.

Operators

operator =	Assigns an **HWND** to an existing **CAxWindow2T** object.

Static Functions

GetWndClassName	Retrieves the name of the window class.

See Also

Reference

CAxWindow2T Class

CAxWindows2T Methods

For information about the methods in **CAxWindow2T**, see CAxWindow2T Members.

CAxWindow2T::CAxWindow2T

Constructs a **CAxWindow2T** object.

```
CAxWindow2T( HWND hWnd = NULL ) : CAxWindowT< TBase >( hWnd )
```

Parameters

hWnd

A handle of an existing window.

See Also

Reference

CAxWindow2T Class

Other Resources

CAxWindow2T Members

CAxWindow2T::Create

Creates a host window.

```
HWND Create(
   HWND hWndParent,
   _U_RECT rect = NULL,
   LPCTSTR szWindowName = NULL,
   DWORD dwStyle = 0,
   DWORD dwExStyle = 0,
   _U_MENUorID MenuOrID = 0U,
   LPVOID lpCreateParam = NULL
);
```

Remarks

CAxWindow2T::Create calls CWindow::Create with the **LPCTSTR** *lpstrWndClass* parameter set to the window class that provides control hosting (**AtlAxWinLic80**).

See **CWindow::Create** for a description of the parameters and return value.

Note If 0 is used as the value for the *MenuOrID* parameter, it must be specified as 0U (the default value) to avoid a compiler error.

Example

See Hosting ActiveX Controls Using ATL AXHost for a sample that uses **CAxWindow2T::Create**.

See Also

Reference

CAxWindow2T Class

Other Resources

CAxWindow2T Members

CAxWindow2T::CreateControlLic

Creates a licensed ActiveX control, initializes it, and hosts it in the specified window.

```
HRESULT CreateControlLic(
   DWORD dwResID,
   IStream* pStream = NULL,
   IUnknown** ppUnkContainer = NULL,
   BSTR bstrLicKey = NULL
);
HRESULT CreateControlLic(
   LPCOLESTR lpszName,
   IStream* pStream = NULL,
   IUnknown** ppUnkContainer = NULL,
   BSTR bstrLicKey = NULL
);
```

Parameters

bstrLicKey

The license key for the control; NULL if creating a nonlicensed control.

Remarks

See CAxWindow::CreateControl for a description of the remaining parameters and return value.

Example

See Hosting ActiveX Controls Using ATL AXHost for a sample that uses **CAxWindow2T::CreateControlLic**.

See Also

Reference

CAxWindow2T Class

Other Resources

CAxWindow2T Members

CAxWindow2T::CreateControlLicEx

Creates a licensed ActiveX control, initializes it, hosts it in the specified window, and retrieves an interface pointer (or pointers) from the control.

```
HRESULT CreateControlLicEx(
   LPCOLESTR lpszName,
   IStream* pStream = NULL,
   IUnknown** ppUnkContainer = NULL,
   IUnknown** ppUnkControl = NULL,
   REFIID iidSink = IID_NULL,
   IUnknown* punkSink = NULL,
   BSTR bstrLicKey = NULL
);
HRESULT CreateControlLicEx(
   DWORD dwResID,
   IStream* pStream = NULL,
   IUnknown** ppUnkContainer = NULL,
   IUnknown** ppUnkControl = NULL,
   REFIID iidSink = IID_NULL,
   IUnknown* punkSink = NULL,
   BSTR bstrLickey = NULL
);
```

Parameters

bstrLicKey

The license key for the control; NULL if creating a nonlicensed control.

Remarks

See CAxWindow::CreateControlEx for a description of the remaining parameters and return value.

Example

See Hosting ActiveX Controls Using ATL AXHost for a sample that uses **CAxWindow2T::CreateControlLicEx**.

See Also

Reference

CAxWindow2T Class

Other Resources

CAxWindow2T Members

CAxWindows2T Operators

For information about the operators in **CAxWindow2T**, see CAxWindow2T Members.

CAxWindow2T::operator =

Assigns an **HWND** to an existing **CAxWindow2T** object.

```
CAxWindow2T< TBase >& operator= (
      HWND hWnd
);
```

Parameters

hWnd

A handle of an existing window.

See Also

Reference

CAxWindow2T Class

Other Resources

CAxWindow2T Members

CAxWindows2T Static Functions

For information about the static functions in **CAxWindow2T**, see CAxWindow2T Members.

CAxWindow2T::GetWndClassName

Retrieves the name of the window class.

```
static LPCTSTR GetWndClassName( );
```

Return Value

A pointer to a string containing the name of the window class (**AtlAxWinLic80**) that can host licensed and nonlicensed ActiveX controls.

See Also

Reference

CAxWindow2T Class

Other Resources

CAxWindow2T Members

CBindStatusCallback Class

This class implements the **IBindStatusCallback** interface.

```
template <class T, int nBindFlags = BINDF_ASYNCHRONOUS |
   BINDF_ASYNCSTORAGE | BINDF_GETNEWESTVERSION | BINDF_NOWRITECACHE>
class ATL_NO_VTABLE CBindStatusCallback : public CComObjectRootEx
   <T::_ThreadModel::ThreadModelNoCS>, public IBindStatusCallbackImpl<T>
```

Parameters

T

Your class containing the function that will be called as the data is received.

nBindFlags

Specifies the bind flags that are returned by GetBindInfo. The default implementation sets the binding to be asynchronous, retrieves the newest version of the data/object, and does not store retrieved data in the disk cache.

Remarks

The **CBindStatusCallback** class implements the **IBindStatusCallback** interface. **IBindStatusCallback** must be implemented by your application so it can receive notifications from an asynchronous data transfer. The asynchronous moniker provided by the system uses **IBindStatusCallback** methods to send and receive information about the asynchronous data transfer to and from your object.

Typically, the **CBindStatusCallback** object is associated with a specific bind operation. For example, in the ASYNC sample, when you set the URL property, it creates a **CBindStatusCallback** object in the call to **Download**:

```
STDMETHOD(put_URL)(BSTR strURL)
{
   ...
   m_bstrURL = strURL;
   CBindStatusCallback<CATLAsync>::Download(this,
             OnData, m_bstrURL, m_spClientSite, FALSE);
   return S_OK;
}
```

The asynchronous moniker uses the callback function `OnData` to call your application when it has data. The asynchronous moniker is provided by the system.

Requirements

Header: atlctl.h

See Also

Other Resources

CBindStatusCallback Members

ATL Class Overview

CBindStatusCallback Members

IBindStatusCallback Methods

GetBindInfo	Called by the asynchronous moniker to request information on the type of bind to be created.
GetPriority	Called by the asynchronous moniker to get the priority of the bind operation. The ATL implementation returns **E_NOTIMPL**.
OnLowResource	Called when resources are low. The ATL implementation returns **S_OK**.
OnObjectAvailable	Called by the asynchronous moniker to pass an object interface pointer to your application. The ATL implementation returns **S_OK**.
OnProgress	Called to indicate the progress of a data downloading process. The ATL implementation returns **S_OK**.
OnStartBinding	Called when binding is started.
OnStopBinding	Called when the asynchronous data transfer is stopped.

Class Methods

CBindStatusCallback	The constructor.
~CBindStatusCallback	The destructor.
Download	Starts the download process, creates a **CBindStatusCallback** object, and calls **StartAsyncDownload**.
OnDataAvailable	Called to provide data to your application as it becomes available. Reads the data, then calls the function passed to it to use the data.
StartAsyncDownload	Initializes the bytes available and bytes read to zero, creates a push-type stream object from a URL, and calls **OnDataAvailable** every time data is available.

Data Members

m_dwAvailableToRead	Number of bytes available to read.
m_dwTotalRead	Total number of bytes read.
m_pFunc	Pointer to the function called when data is available.
m_pT	Pointer to the object requesting the asynchronous data transfer.
m_spBindCtx	Pointer to the IBindCtx interface for the current bind operation.
m_spBinding	Pointer to the **IBinding** interface for the current bind operation.
m_spMoniker	Pointer to the IMoniker interface for the URL to use.
m_spStream	Pointer to the IStream interface for the data transfer.

Static Functions

Download	Creates a **CBindStatusCallback** object and calls **StartAsyncDownload** to start downloading data asynchronously from the specified URL.

See Also

Reference

CBindStatusCallback Class

CBindStatusCallback Methods

For information about the methods in **CBindStatusCallback**, see CBindStatusCallback Members.

CBindStatusCallback::CBindStatusCallback

The constructor.

```
CBindStatusCallback( );
```

Remarks

Creates an object to receive notifications concerning the asynchronous data transfer. Typically, one object is created for each bind operation.

The constructor also initializes m_pT and m_pFunc to **NULL**.

See Also

Reference

CBindStatusCallback Class

CBindStatusCallback::StartAsyncDownload

Other Resources

CBindStatusCallback Members

CBindStatusCallback::~CBindStatusCallback

The destructor.

```
~CBindStatusCallback( );
```

Remarks

Frees all allocated resources.

See Also

Reference

CComPolyObject Class

CComPolyObject::FinalRelease

Other Resources

CComPolyObject Members

CBindStatusCallback::GetBindInfo

Called to tell the moniker how to bind.

```
STDMETHOD(GetBindInfo)(
   DWORD* pgrfBSCF,
      BINDINFO* pbindinfo
);
```

Parameters

pgrfBSCF

[out] A pointer to **BINDF** enumeration values indicating how the bind operation should occur. By default, set with the following enumeration values:

BINDF_ASYNCHRONOUS Asynchronous download.

BINDF_ASYNCSTORAGE **OnDataAvailable** returns **E_PENDING** when data is not yet available rather than blocking until data is available.

BINDF_GETNEWESTVERSION The bind operation should retrieve the newest version of the data.

BINDF_NOWRITECACHE The bind operation should not store retrieved data in the disk cache.

pbindinfo

[in, out] A pointer to the **BINDINFO** structure giving more information about how the object wants binding to occur.

Return Value

One of the standard **HRESULT** values.

Remarks

The default implementation sets the binding to be asynchronous and to use the data-push model. In the data-push model, the moniker drives the asynchronous bind operation and continuously notifies the client whenever new data is available.

See Also

Reference

CBindStatusCallback Class

Other Resources

CBindStatusCallback Members

CBindStatusCallback::GetPriority

Called by the asynchronous moniker to get the priority of the bind operation.

```
STDMETHOD(GetPriority)(
   LONG* pnPriority
);
```

Parameters

pnPriority

[out] Address of the **LONG** variable that, on success, receives the priority.

Return Value

Returns **E_NOTIMPL**.

See Also

Reference

CBindStatusCallback Class

Other Resources

CBindStatusCallback Members

CBindStatusCallback::OnDataAvailable

The system-supplied asynchronous moniker calls **OnDataAvailable** to provide data to the object as it becomes available.

```
STDMETHOD(OnDataAvailable)(
   DWORD grfBSCF,
      DWORD dwSize,
      FORMATETC* /* pformatetc */,
      STGMEDIUM* pstgmed
);
```

Parameters

grfBSCF

[in] A **BSCF** enumeration value. One or more of the following: **BSCF_FIRSTDATANOTIFICATION**, **BSCF_INTERMEDIARYDATANOTIFICATION**, or **BSCF_LASTDATANOTIFICATION**.

dwSize

[in] The cumulative amount (in bytes) of data available since the beginning of the binding. Can be zero, indicating that the amount of data is not relevant or that no specific amount became available.

pformatetc

[in] Pointer to the FORMATETC structure that contains the format of the available data. If there is no format, can be **CF_NULL**.

pstgmed

[in] Pointer to the STGMEDIUM structure that holds the actual data now available.

Return Value

One of the standard **HRESULT** values.

Remarks

OnDataAvailable reads the data, then calls a method of your object's class (for example, to store the data or print it to the screen). See CBindStatusCallback::StartAsyncDownload for details.

See Also

Reference

CBindStatusCallback Class

CBindStatusCallback::StartAsyncDownload

Other Resources

CBindStatusCallback Members

CBindStatusCallback::OnLowResource

Called when resources are low.

```
STDMETHOD(OnLowResource)(
   DWORD /* dwReserved */
);
```

Parameters

dwReserved

Reserved.

Return Value

Returns **S_OK**.

See Also

Reference

CBindStatusCallback Class

Other Resources

CBindStatusCallback Members

CBindStatusCallback::OnObjectAvailable

Called by the asynchronous moniker to pass an object interface pointer to your application.

```
STDMETHOD(OnObjectAvailable)(
   REFID /* riid */,
      IUnknown* /* punk */
);
```

Parameters

riid

Interface identifier of the requested interface. Unused.

punk

Address of the IUnknown interface. Unused.

Return Value

Returns **S_OK**.

See Also

Reference

CBindStatusCallback Class

Other Resources

CBindStatusCallback Members

CBindStatusCallback::OnProgress

Called to indicate the progress of a data downloading process.

```
STDMETHOD(OnProgress)(
   ULONG /* ulProgress */,
      ULONG /* ulProgressMax */,
      ULONG /* ulStatusCode */,
      LPCWSTRONG /* szStatusText */
);
```

Parameters

ulProgress

Unsigned long integer. Unused.

ulProgressMax

Unsigned long integer Unused.

ulStatusCode

Unsigned long integer. Unused.

szStatusText

Address of a string value. Unused.

Return Value

Returns **S_OK**.

See Also

Reference

CBindStatusCallback Class

Other Resources

CBindStatusCallback Members

CBindStatusCallback::OnStartBinding

Sets the data member m_spBinding to the **IBinding** pointer in *pBinding*.

```
STDMETHOD(OnStartBinding)(
   DWORD /* dwReserved */,
      IBinding* pBinding
);
```

Parameters

dwReserved

Reserved for future use.

pBinding

[in] Address of the IBinding interface of the current bind operation. This cannot be NULL. The client should call AddRef on this pointer to keep a reference to the binding object.

See Also

Reference

CBindStatusCallback Class

CBindStatusCallback::OnStopBinding

Other Resources

CBindStatusCallback Members

CBindStatusCallback::OnStopBinding

Releases the **IBinding** pointer in the data member m_spBinding.

```
STDMETHOD(OnStopBinding)(
   HRESULT hresult,
      LPCWSTR /* szError */
);
```

Parameters

hresult

Status code returned from the bind operation.

szStatusText

Address of a string value Unused.

Remarks

Called by the system-supplied asynchronous moniker to indicate the end of the bind operation.

See Also

Reference

CBindStatusCallback Class

CBindStatusCallback::OnStartBinding

Other Resources

CBindStatusCallback Members

CBindStatusCallback::StartAsyncDownload

Starts downloading data asynchronously from the specified URL.

```
HRESULT StartAsyncDownload(
      T* pT,
      ATL_PDATAAVAILABLE pFunc,
      BSTR bstrURL,
   IUnknown* pUnkContainer = NULL,
      BOOL bRelative = FALSE
);
```

Parameters

pT

[in] A pointer to the object requesting the asynchronous data transfer. The **CBindStatusCallback** object is templatized on this object's class.

pFunc

[in] A pointer to the function that receives the data being read. The function is a member of your object's class of type *T*. See **Remarks** for syntax and an example.

bstrURL

[in] The URL to obtain data from. Can be any valid URL or file name. Cannot be **NULL**. For example:

```
CComBSTR mybstr =_T("http://somesite/data.htm")
```

pUnkContainer

[in] The **IUnknown** of the container. **NULL** by default.

bRelative

[in] A flag indicating whether the URL is relative or absolute. **FALSE** by default, meaning the URL is absolute.

Return Value

One of the standard **HRESULT** values.

Remarks

Every time data is available it is sent to the object through **OnDataAvailable**. **OnDataAvailable** reads the data and calls the function pointed to by *pFunc* (for example, to store the data or print it to the screen).

The function pointed to by *pFunc* is a member of your object's class and has the following syntax:

```
        void Function_Name(
   CBindStatusCallback<T>* pbsc,
   BYTE* pBytes,
   DWORD dwSize
);
```

In the following example (taken from the ASYNC sample), the function `OnData` writes the received data into a text box.

Example

```
void OnData(CBindStatusCallback<CATLAsync>* pbsc,
            BYTE* pBytes, DWORD dwSize)
{
   m_bstrText.Append= (LPCSTR)pBytes;
   if (::IsWindow(m_EditCtrl.m_hWnd))
   {
      USES_CONVERSION;
      ::SendMessage(m_EditCtrl.m_hWnd, WM_SETTEXT, 0,
```

```
                        (LPARAM)OLE2CT((BSTR)m_bstrText));
    }
}
```

See Also

Reference

CBindStatusCallback Class

CBindStatusCallback::OnDataAvailable

Other Resources

CBindStatusCallback Members

CBindStatusCallback Data Members

For information about the data members in **CBindStatusCallback**, see CBindStatusCallback Members.

CBindStatusCallback::m_dwAvailableToRead

Can be used to store the number of bytes available to be read.

```
DWORD m_dwAvailableToRead;
```

Remarks

Initialized to zero in **StartAsyncDownload**.

See Also

Reference

CBindStatusCallback Class

CBindStatusCallback::StartAsyncDownload

Other Resources

CBindStatusCallback Members

CBindStatusCallback::m_dwTotalRead

The cumulative total of bytes read in the asynchronous data transfer.

```
DWORD m_dwTotalRead;
```

Remarks

Incremented every time **OnDataAvailable** is called by the number of bytes actually read. Initialized to zero in **StartAsyncDownload**.

See Also

Reference

CBindStatusCallback Class
CBindStatusCallback::StartAsyncDownload
CBindStatusCallback::OnDataAvailable

Other Resources

CBindStatusCallback Members

CBindStatusCallback::m_pFunc

The function pointed to by **m_pFunc** is called by **OnDataAvailable** after it reads the available data (for example, to store the data or print it to the screen).

```
ATL_PDATAAVAILABLE m_pFunc;
```

Remarks

The function pointed to by **m_pFunc** is a member of your object's class and has the following syntax:

```
        void Function_Name(
   CBindStatusCallback<T>* pbsc,
   BYTE* pBytes,
   DWORD dwSize
);
```

See Also

Reference

CBindStatusCallback Class

CBindStatusCallback::StartAsyncDownload

CBindStatusCallback::OnDataAvailable

Other Resources

CBindStatusCallback Members

CBindStatusCallback::m_pT

A pointer to the object requesting the asynchronous data transfer.

```
T* m_pT;
```

Remarks

The **CBindStatusCallback** object is templatized on this object's class.

See Also

Reference

CBindStatusCallback Class

CBindStatusCallback::StartAsyncDownload

Other Resources

CBindStatusCallback Members

CBindStatusCallback::m_spBindCtx

A pointer to an IBindCtx interface that provides access to the bind context (an object that stores information about a particular moniker binding operation).

```
CComPtr<IBindCtx> m_spBindCtx;
```

Remarks

Initialized in **StartAsyncDownload**.

See Also

Reference

CBindStatusCallback Class
CBindStatusCallback::StartAsyncDownload
CComPtr Class

Other Resources

CBindStatusCallback Members

CBindStatusCallback::m_spBinding

A pointer to the **IBinding** interface of the current bind operation.

```
CComPtr<IBinding> m_spBinding;
```

Remarks

Initialized in **OnStartBinding** and released in **OnStopBinding**.

See Also

Reference

CBindStatusCallback Class
CBindStatusCallback::OnStartBinding
CBindStatusCallback::OnStopBinding
CComPtr Class

Other Resources

CBindStatusCallback Members

CBindStatusCallback::m_spMoniker

A pointer to the IMoniker interface for the URL to use.

```
CComPtr<IMoniker> m_spMoniker;
```

Remarks

Initialized in **StartAsyncDownload**.

See Also

Reference

CBindStatusCallback Class
CBindStatusCallback::StartAsyncDownload
CComPtr Class

Other Resources

CBindStatusCallback Members

CBindStatusCallback::m_spStream

A pointer to the IStream interface of the current bind operation.

```
CComPtr<IStream> m_spStream;
```

Remarks

Initialized in **OnDataAvailable** from the **STGMEDIUM** structure when the **BCSF** flag is **BCSF_FIRSTDATANOTIFICATION** and released when the **BCSF** flag is **BCSF_LASTDATANOTIFICATION**.

See Also

Reference

CBindStatusCallback Class
CBindStatusCallback::OnDataAvailable
CComPtr Class

Other Resources

CBindStatusCallback Members

CBindStatusCallback Static Functions

For information about the static functions in **CBindStatusCallback**, see CBindStatusCallback Members.

CBindStatusCallback::Download

Creates a **CBindStatusCallback** object and calls **StartAsyncDownload** to start downloading data asynchronously from the specified URL.

```
static HRESULT Download(
      T* pT,
      ATL_PDATAAVAILABLE pFunc,
      BSTR bstrURL,
      IUnknown* pUnkContainer = NULL,
      BOOL bRelative = FALSE
);
```

Parameters

pT

[in] A pointer to the object requesting the asynchronous data transfer. The **CBindStatusCallback** object is templatized on this object's class.

pFunc

[in] A pointer to the function that receives the data that is read. The function is a member of your object's class of type *T*. See StartAsyncDownload for syntax and an example.

bstrURL

[in] The URL to obtain data from. Can be any valid URL or file name. Cannot be **NULL**. For example:

```
CComBSTR mybstr =_T("http://somesite/data.htm")
```

pUnkContainer

[in] The **IUnknown** of the container. **NULL** by default.

bRelative

[in] A flag indicating whether the URL is relative or absolute. **FALSE** by default, meaning the URL is absolute.

Return Value

One of the standard **HRESULT** values.

Remarks

Every time data is available it is sent to the object through **OnDataAvailable**. **OnDataAvailable** reads the data and calls the function pointed to by *pFunc* (for example, to store the data or print it to the screen).

See Also

Reference

CBindStatusCallback Class

CBindStatusCallback::StartAsyncDownload

Other Resources

CBindStatusCallback Members

CComAggObject Class

This class implements IUnknown for an aggregated object.

```
template<
   class contained
>
class CComAggObject :
   public IUnknown, public CComObjectRootEx
   < contained::_ThreadModel::ThreadModelNoCS >
```

Parameters

contained

Your class, derived from CComObjectRoot or CComObjectRootEx, as well as from any other interfaces you want to support on the object.

Remarks

CComAggObject implements IUnknown for an aggregated object. **CComAggObject** has its own **IUnknown**, separate from the outer object's **IUnknown**, and maintains its own reference count.

CComAggObject uses CComContainedObject to delegate to the outer unknown.

For more information about aggregation, see the article Fundamentals of ATL COM Objects.

Requirements

Header: atlcom.h

See Also

Reference

CComObject Class
CComPolyObject Class
DECLARE_AGGREGATABLE
DECLARE_ONLY_AGGREGATABLE
DECLARE_NOT_AGGREGATABLE

Other Resources

CComAggObject Members
ATL Class Overview

CComAggObject Members

Class Methods

CComAggObject	The constructor.
~CComAggObject	The destructor.
FinalConstruct	Performs final initialization of **m_contained**.
FinalRelease	Performs final destruction of **m_contained**.

IUnknown Methods

AddRef	Increments the reference count on the aggregated object.
QueryInterface	Retrieves a pointer to the requested interface.
Release	Decrements the reference count on the aggregated object.

Data Members

m_contained	Delegates **IUnknown** calls to the outer unknown.

Static Functions

CreateInstance	This static function allows you to create a new **CComAggObject<** *contained* **>** object without the overhead of CoCreateInstance.

See Also

Reference

CComAggObject Class

CComAggObject Methods

For information about the methods in **CComAggObject**, see CComAggObject Members.

CComAggObject::AddRef

Increments the reference count on the aggregated object.

```
STDMETHOD_(ULONG, AddRef)( );
```

Return Value

A value that may be useful for diagnostics or testing.

See Also

Reference

CComAggObject Class

CComAggObject::Release

Other Resources

CComAggObject Members

CComAggObject::CComAggObject

The constructor.

```
CComAggObject(
   void* pv
);
```

Parameters

pv

[in] The outer unknown.

Remarks

Initializes the **CComContainedObject** member, m_contained, and increments the module lock count.

The destructor decrements the module lock count.

See Also

Reference

CComAggObject Class
CComAggObject::FinalConstruct
CComAggObject::FinalRelease

Other Resources

CComAggObject Members

CComAggObject::~CComAggObject

The destructor.

```
~CComAggObject( );
```

Remarks

Frees all allocated resources, calls FinalRelease, and decrements the module lock count.

See Also

Reference

CComAggObject Class

Other Resources

CComAggObject Members

CComAggObject::FinalConstruct

Called during the final stages of object construction, this method performs any final initialization on the m_contained member.

```
HRESULT FinalConstruct( );
```

Return Value

A standard **HRESULT** value.

See Also

Reference

CComAggObject Class
CComObjectRootEx::FinalConstruct
CComAggObject::FinalRelease

Other Resources

CComAggObject Members

CComAggObject::FinalRelease

Called during object destruction, this method frees the m_contained member.

```
void FinalRelease( );
```

See Also

Reference

CComAggObject Class
CComObjectRootEx::FinalRelease
CComAggObject::FinalConstruct

Other Resources

CComAggObject Members

CComAggObject::QueryInterface

Retrieves a pointer to the requested interface.

```
STDMETHOD(QueryInterface)(
   REFIID iid,
   void ** ppvObject
);
template <class Q>
HRESULT STDMETHODCALLTYPE QueryInterface(
   Q** pp
);
```

Parameters

iid

[in] The identifier of the interface being requested.

ppvObject

[out] A pointer to the interface pointer identified by *iid*. If the object does not support this interface, *ppvObject* is set to **NULL**.

pp

[out] A pointer to the interface pointer identified by type *Q*. If the object does not support this interface, *pp* is set to **NULL**.

Return Value

A standard **HRESULT** value.

Remarks

If the requested interface is **IUnknown**, **QueryInterface** returns a pointer to the aggregated object's own **IUnknown** and increments the reference count. Otherwise, this method queries for the interface through the **CComContainedObject** member, m_contained.

See Also

Reference

CComAggObject Class

Other Resources

CComAggObject Members

CComAggObject::Release

Decrements the reference count on the aggregated object.

```
STDMETHOD_(ULONG, Release)( );
```

Return Value

In debug builds, **Release** returns a value that may be useful for diagnostics or testing. In non-debug builds, **Release** always returns 0.

See Also

Reference

CComAggObject Class

CComAggObject::AddRef

Other Resources

CComAggObject Members

CComAggObject Data Members

For information about the data members in **CComAggObject**, see CComAggObject Members.

CComAggObject::m_contained

A CComContainedObject object derived from your class.

```
CComContainedObject< contained > m_contained;
```

Parameters

contained

[in] Your class, derived from CComObjectRoot or CComObjectRootEx, as well as from any other interfaces you want to support on the object.

Remarks

All **IUnknown** calls through **m_contained** are delegated to the outer unknown.

See Also

Reference

CComAggObject Class

Other Resources

CComAggObject Members

CComAggObject Static Functions

For information about the static functions in **CComAggObject**, see CComAggObject Members.

CComAggObject::CreateInstance

This static function allows you to create a new **CComAggObject<** *contained* **>** object without the overhead of CoCreateInstance.

```
static HRESULT WINAPI CreateInstance(
   LPUNKNOWN pUnkOuter,
   CComAggObject< contained >** pp
);
```

Parameters

pp

[out] A pointer to a **CComAggObject<** *contained* **>** pointer. If **CreateInstance** is unsuccessful, *pp* is set to **NULL**.

Return Value

A standard **HRESULT** value.

Remarks

The object returned has a reference count of zero, so call **AddRef** immediately, then use **Release** to free the reference on the object pointer when you're done.

If you do not need direct access to the object, but still want to create a new object without the overhead of **CoCreateInstance**, use CComCoClass::CreateInstance instead.

See Also

Reference

CComAggObject Class

Other Resources

CComAggObject Members

CComAllocator Class

This class provides methods for managing memory using COM memory routines.

```
class CComAllocator
```

Remarks

This class is used by CComHeapPtr to provide the COM memory allocation routines. The counterpart class, CCRTAllocator, provides the same methods using CRT routines.

Requirements

Header: atlbase.h

See Also

Reference

CComHeapPtr Class

CCRTAllocator Class

Other Resources

CComAllocator Members

ATL Class Overview

CComAllocator Members

Static Functions

Allocate	Call this method to allocate memory.
Free	Call this method to free allocated memory.
Reallocate	Call this method to reallocate memory.

See Also

Reference

CComAllocator Class

CComAllocator Static Functions

For information about the static functions in **CComAllocator**, see CComAllocator Members.

CComAllocator::Allocate

Call this static function to allocate memory.

```
static void* Allocate(
   size_t nBytes
) throw( );
```

Parameters

nBytes

The number of bytes to allocate.

Return Value

Returns a void pointer to the allocated space, or NULL if there is insufficient memory available.

Remarks

Allocates memory. See CoTaskMemAlloc for more details.

See Also

Reference

CComAllocator Class
CComAllocator::Free
CComAllocator::Reallocate

Other Resources

CComAllocator Members

CComAllocator::Free

Call this static function to free allocated memory.

```
static void Free(
   void* p
) throw( );
```

Parameters

p

Pointer to the allocated memory.

Remarks

Frees the allocated memory. See CoTaskMemFree for more details.

See Also

Reference

CComAllocator Class
CComAllocator::Allocate
CComAllocator::Reallocate

Other Resources

CComAllocator Members

CComAllocator::Reallocate

Call this static function to reallocate memory.

```
static void* Reallocate(
   void* p,
   size_t nBytes
) throw( );
```

Parameters

p

Pointer to the allocated memory.

nBytes

The number of bytes to reallocate.

Return Value

Returns a void pointer to the allocated space, or NULL if there is insufficient memory

Remarks

Resizes the amount of allocated memory. See CoTaskMemRealloc for more details.

See Also

Reference

CComAllocator Class
CComAllocator::Allocate
CComAllocator::Free

Other Resources

CComAllocator Members

CComApartment Class

This class provides support for managing an appartment in a thread-pooled EXE module.

```
class CComApartment
```

Remarks

CComApartment is used by CComAutoThreadModule to manage an apartment in a thread-pooled EXE module. **CComApartment** provides methods for incrementing and decrementing the lock count on a thread.

Requirements

Header: atlbase.h

See Also

Other Resources

CComApartment Members

ATL Class Overview

CComApartment Members

Methods

Apartment	Marks the thread's starting address.
CComApartment	The constructor.
GetLockCount	Returns the thread's current lock count.
Lock	Increments the thread's lock count.
Unlock	Decrements the thread's lock count.

Data Members

m_dwThreadID	Contains the thread's identifier.
m_hThread	Contains the thread's handle.
m_nLockCnt	Contains the thread's current lock count.

See Also

Reference

CComApartment Class

CComApartment Methods

For information about the methods in **CComApartment**, see CComApartment Members.

CComApartment::Apartment

Marks the thread's starting address.

```
DWORD Apartment( );
```

Return Value

Always 0.

Remarks

Automatically set during CComAutoThreadModule::Init.

See Also

Reference

CComApartment Class

Other Resources

CComApartment Members

CComApartment::CComApartment

The constructor.

```
CComApartment( );
```

Remarks

Initializes the **CComApartment** data members m_nLockCnt and m_hThread.

See Also

Reference

CComApartment Class

Other Resources

CComApartment Members

CComApartment::GetLockCount

Returns the thread's current lock count.

```
LONG GetLockCount( );
```

Return Value

The lock count on the thread.

See Also

Reference

CComApartment Class
CComApartment::Lock
CComApartment::Unlock
CComApartment::m_nLockCnt

Other Resources

CComApartment Members

CComApartment::Lock

Increments the thread's lock count.

```
LONG Lock( );
```

Return Value

A value that may be useful for diagnostics or testing.

Remarks

Called by CComAutoThreadModule::Lock.

The lock count on the thread is used for statistical purposes.

See Also

Reference

CComApartment Class
CComApartment::Unlock
CComApartment::GetLockCount
CComApartment::m_nLockCnt

Other Resources

CComApartment Members

CComApartment::Unlock

Decrements the thread's lock count.

```
LONG Unlock( );
```

Return Value

A value that may be useful for diagnostics or testing.

Remarks

Called by CComAutoThreadModule::Unlock.

The lock count on the thread is used for statistical purposes.

See Also

Reference

CComApartment Class
CComApartment::Lock
CComApartment::GetLockCount
CComApartment::m_nLockCnt

Other Resources

CComApartment Members

CComApartment Data Members

For information about the data members in **CComApartment**, see CComApartment Members.

CComApartment::m_dwThreadID

Contains the thread's identifier.

```
DWORD m_dwThreadID;
```

See Also

Reference

CComApartment Class

Other Resources

CComApartment Members

CComApartment::m_hThread

Contains the thread's handle.

```
HANDLE m_hThread;
```

See Also

Reference

CComApartment Class

Other Resources

CComApartment Members

CComApartment::m_nLockCnt

Contains the thread's current lock count.

```
LONG m_nLockCnt;
```

See Also

Reference

CComApartment Class
CComApartment::Lock
CComApartment::Unlock

Other Resources

CComApartment Members

CComAutoCriticalSection Class

CComAutoCriticalSection provides methods for obtaining and releasing ownership of a critical section object.

```
class CComAutoCriticalSection : public CComCriticalSection
```

Remarks

CComAutoCriticalSection is similar to class CComCriticalSection, except **CComAutoCriticalSection** automatically initializes the critical section object in the constructor.

Typically, you use **CComAutoCriticalSection** through the **typedef** name AutoCriticalSection. This name references **CComAutoCriticalSection** when CComMultiThreadModel is being used.

The **Init** and **Term** methods from CComCriticalSection are not available when using this class.

Requirements

Header: atlcore.h

See Also

Reference

CComFakeCriticalSection Class

CComCriticalSection Class

Other Resources

CComAutoCriticalSection Members

ATL Class Overview

CComAutoCriticalSection Members

Methods

CComAutoCriticalSection	The constructor.
~CComAutoCriticalSection	The destructor.

See Also

Reference

CComAutoCriticalSection Class

Other Resources

CComCriticalSection Members

CComAutoCriticalSection Methods

For information about the methods in **CComAutoCriticalSection**, see CComAutoCriticalSection Members.

CComAutoCriticalSection::CComAutoCriticalSection

The constructor.

```
CComAutoCriticalSection( );
```

Remarks

Calls the Win32 function InitializeCriticalSection, which initializes the critical section object.

See Also

Reference

CComAutoCriticalSection Class

CComAutoCriticalSection::~CComAutoCriticalSection

Other Resources

CComAutoCriticalSection Members

CComAutoCriticalSection::~CComAutoCriticalSection

The destructor.

```
~CComAutoCriticalSection( ) throw( );
```

Remarks

The destructor calls DeleteCriticalSection, which releases all system resources used by the critical section object.

See Also

Reference

CComAutoCriticalSection Class

CComAutoCriticalSection::CComAutoCriticalSection

Other Resources

CComAutoCriticalSection Members

CComAutoDeleteCriticalSection Class

This class provides methods for obtaining and releasing ownership of a critical section object.

```
class CComAutoDeleteCriticalSection : public CComSafeDeleteCriticalSection
```

Remarks

CComAutoDeleteCriticalSection derives from the class CComSafeDeleteCriticalSection. However, **CComAutoDeleteCriticalSection** overrides the Term method to **private** access, which forces internal memory cleanup to occur only when instances of this class go out of scope or are explicitly deleted from memory.

This class introduces no additional methods over its base class. See CComSafeDeleteCriticalSection and CComCriticalSection for more information on critical section helper classes.

Requirements

Header: atlcore.h

See Also

Reference

CComSafeDeleteCriticalSection Class

CComCriticalSection Class

Other Resources

ATL Class Overview

CComAutoThreadModule Class

As of ATL 7.0, **CComAutoThreadModule** is obsolete: see ATL Module Classes for more details.

```
template<
   class ThreadAllocator = CComSimpleThreadAllocator
>
class CComAutoThreadModule :
   public CComModule
```

Parameters

ThreadAllocator

[in] The class managing thread selection. The default value is CComSimpleThreadAllocator.

Remarks

Note

This class is obsolete, having been replaced by the CAtlAutoThreadModule and CAtlModule derived classes. The information that follows is for use with older releases of ATL.

CComAutoThreadModule derives from CComModule to implement a thread-pooled, apartment-model COM server for EXEs and Windows services. **CComAutoThreadModule** uses CComApartment to manage an apartment for each thread in the module.

Derive your module from **CComAutoThreadModule** when you want to create objects in multiple apartments. You must also include the DECLARE_CLASSFACTORY_AUTO_THREAD macro in your object's class definition to specify CComClassFactoryAutoThread as the class factory.

By default, the ATL COM AppWizard (the ATL Project Wizard in Visual Studio .NET) will derive your module from **CComModule**. To use **CComAutoThreadModule**, modify the class definition. For example:

```
class CMyModule :
public CComAutoThreadModule<CComSimpleThreadAllocator>
{
public:
   LONG Unlock( )
   {
      LONG l = CComAutoThreadModule<ComSimpleThreadAllocator>::Unlock( );
      if (l == 0)
         PostThreadMessage(dwThreadID, WM_QUIT, 0, 0);
      return l;
   }

   DWORD dwThreadID;
};
```

Requirements

Header: atlbase.h

See Also

Reference

ATL Module Classes

Other Resources

CComAutoThreadModule Members

ATL Class Overview

CComAutoThreadModule Members

Methods

CreateInstance	Selects a thread and then creates an object in the associated apartment.
Init	Creates the module's threads.
Lock	Increments the lock count on the module and on the current thread.
Unlock	Decrements the lock count on the module and on the current thread.

Data Members

dwThreadID	Contains the identifier of the current thread.
m_Allocator	Manages thread selection.
m_nThreads	Contains the number of threads in the module.
m_pApartments	Manages the module's apartments.

Static Functions

GetDefaultThreads	Dynamically calculates the number of threads for the module based on the number of processors.

See Also

Reference

CComAutoThreadModule Class

Other Resources

CComModule Members

CComAutoThreadModule Methods

For information about the methods in **CComAutoThreadModule**, see CComAutoThreadModule Members.

CComAutoThreadModule::CreateInstance

As of ATL 7.0, **CComAutoThreadModule** is obsolete: see ATL Module Classes for more details.

```
HRESULT CreateInstance(
   void* pfnCreateInstance,
   REFIID riid,
   void** ppvObj
);
```

Parameters

pfnCreateInstance

[in] A pointer to a creator function.

riid

[in] The IID of the requested interface.

ppvObj

[out] A pointer to the interface pointer identified by *riid*. If the object does not support this interface, *ppvObj* is set to NULL.

Return Value

A standard HRESULT value.

Remarks

Selects a thread and then creates an object in the associated apartment.

See Also

Reference

CComAutoThreadModule Class

Other Resources

CComAutoThreadModule Members

CComAutoThreadModule::Init

As of ATL 7.0, **CComAutoThreadModule** is obsolete: see ATL Module Classes for more details.

```
HRESULT Init(
   _ATL_OBJMAP_ENTRY* p,
   HINSTANCE h,
   const GUID* plibid = NULL,
   int nThreads = GetDefaultThreads( )
);
```

Parameters

p

[in] A pointer to an array of object map entries.

h

[in] The **HINSTANCE** passed to **DLLMain** or **WinMain**.

plibid

[in] A pointer to the LIBID of the type library associated with the project.

nThreads

[in] The number of threads to be created. By default, *nThreads* is the value returned by GetDefaultThreads.

Remarks

Initializes data members and creates the number of threads specified by *nThreads*.

See Also

Reference

CComAutoThreadModule Class

CComAutoThreadModule::m_nThreads

Other Resources

CComAutoThreadModule Members

CComAutoThreadModule::Lock

As of ATL 7.0, **CComAutoThreadModule** is obsolete: see ATL Module Classes for more details.

```
LONG Lock( );
```

Return Value

A value that may be useful for diagnostics or testing.

Remarks

Performs an atomic increment on the lock count for the module and for the current thread. **CComAutoThreadModule** uses the module lock count to determine whether any clients are accessing the module. The lock count on the current thread is used for statistical purposes.

See Also

Reference

CComAutoThreadModule Class

CComAutoThreadModule::Unlock

Other Resources

CComAutoThreadModule Members

CComAutoThreadModule::Unlock

As of ATL 7.0, **CComAutoThreadModule** is obsolete: see ATL Module Classes for more details.

```
LONG Unlock( );
```

Return Value

A value that may be useful for diagnostics or testing.

Remarks

Performs an atomic decrement on the lock count for the module and for the current thread. **CComAutoThreadModule** uses the module lock count to determine whether any clients are accessing the module. The lock count on the current thread is used for statistical purposes.

When the module lock count reaches zero, the module can be unloaded.

See Also

Reference

CComAutoThreadModule Class

CComAutoThreadModule::Lock

Other Resources

CComAutoThreadModule Members

CComAutoThreadModule Data Members

For information about the data members in **CComAutoThreadModule**, see CComAutoThreadModule Members.

CComAutoThreadModule::dwThreadID

As of ATL 7.0, **CComAutoThreadModule** is obsolete: see ATL Module Classes for more details.

```
DWORD dwThreadID;
```

Remarks

Contains the identifier of the current thread.

See Also

Reference

CComAutoThreadModule Class

Other Resources

CComAutoThreadModule Members

CComAutoThreadModule::m_Allocator

As of ATL 7.0, **CComAutoThreadModule** is obsolete: see ATL Module Classes for more details.

```
ThreadAllocator m_Allocator;
```

Remarks

The object managing thread selection. By default, the *ThreadAllocator* class template parameter is CComSimpleThreadAllocator.

See Also

Reference

CComAutoThreadModule Class

Other Resources

CComAutoThreadModule Members

CComAutoThreadModule::m_nThreads

As of ATL 7.0, **CComAutoThreadModule** is obsolete: see ATL Module Classes for more details.

```
int m_nThreads;
```

Remarks

Contains the number of threads in the EXE module. When Init is called, **m_nThreads** is set to the *nThreads* parameter value. Each thread's associated apartment is managed by a CComApartment object.

See Also

Reference

CComAutoThreadModule Class

CComAutoThreadModule::m_pApartments

Other Resources

CComAutoThreadModule Members

CComAutoThreadModule::m_pApartments

As of ATL 7.0, **CComAutoThreadModule** is obsolete: see ATL Module Classes for more details.

```
CComApartment* m_pApartments;
```

Remarks

Points to an array of CComApartment objects, each of which manages an apartment in the module. The number of elements in the array is based on the m_nThreads member.

See Also

Reference

CComAutoThreadModule Class

Other Resources

CComAutoThreadModule Members

CComAutoThreadModule Static Functions

For information about the static functions in **CComAutoThreadModule**, see CComAutoThreadModule Members.

CComAutoThreadModule::GetDefaultThreads

As of ATL 7.0, **CComAutoThreadModule** is obsolete: see ATL Module Classes for more details.

```
static int GetDefaultThreads( );
```

Return Value

The number of threads to be created in the EXE module.

Remarks

This static function dynamically calculates the maximum number of threads for the EXE module, based on the number of processors. By default, this return value is passed to the Init method to create the threads.

See Also

Reference

CComAutoThreadModule Class

Other Resources

CComAutoThreadModule Members

CComBSTR Class

This class is a wrapper for **BSTR**s.

```
class CComBSTR
```

Remarks

The **CComBSTR** class is a wrapper for **BSTR**s, which are length-prefixed strings. The length is stored as an integer at the memory location preceding the data in the string.

A BSTR is null-terminated after the last counted character but may also contain null characters embedded within the string. The string length is determined by the character count, not the first null character.

Note

The **CComBSTR** class provides a number of members (constructors, assignment operators, and comparison operators) that take either ANSI or Unicode strings as arguments. The ANSI versions of these functions are less efficient than their Unicode counterparts because temporary Unicode strings are often created internally. For efficiency, use the Unicode versions where possible.

Note

Because of the improved lookup behavior implemented in Visual Studio .NET, code such as `bstr = L"String2" + bstr;`, which may have compiled in previous releases, should instead be implemented as `bstr = CStringW(L"String2") + bstr`.

For a list of cautions when using **CComBSTR**, see Programming with CComBSTR.

Requirements

Header: atlbase.h

See Also

Concepts

ATL and MFC String Conversion Macros

Other Resources

CComBSTR Members

ATL Class Overview

CComBSTR Members

Methods

Append	Appends a string to **m_str**.
AppendBSTR	Appends a **BSTR** to **m_str**.
AppendBytes	Appends a specified number of bytes to **m_str**.
ArrayToBSTR	Creates a **BSTR** from the first character of each element in the safearray and attaches it to the **CComBSTR** object.
AssignBSTR	Assigns a **BSTR** to **m_str**.
Attach	Attaches a **BSTR** to the **CComBSTR** object.
BSTRToArray	Creates a zero-based one-dimensional safearray, where each element of the array is a character from the **CComBSTR** object.
ByteLength	Returns the length of **m_str** in bytes.
CComBstr	The constructor.
~CComBstr	The destructor.
Copy	Returns a copy of **m_str**.
CopyTo	Returns a copy of **m_str** via an **[out]** parameter
Detach	Detaches **m_str** from the **CComBSTR** object.
Empty	Frees **m_str**.
Length	Returns the length of **m_str**.
LoadString	Loads a string resource.
ReadFromStream	Loads a **BSTR** object from a stream.
ToLower	Converts the string to lowercase.
ToUpper	Converts the string to uppercase.
WriteToStream	Saves **m_str** to a stream.

Operators

operator !	Returns **true** or **false**, depending on whether **m_str** is **NULL**.
operator !=	Compares a **CComBSTR** with a string.
operator &	Returns the address of **m_str**.

operator +=	Appends a **CComBSTR** to the object.
operator <	Compares a **CComBSTR** with a string.
operator =	Assigns a value to **m_str**.
operator ==	Compares a **CComBSTR** with a string.
operator >	Compares a **CComBSTR** with a string.
operator BSTR	Casts a **CComBSTR** object to a **BSTR**.

Data Members

m_str	Contains the **BSTR** associated with the **CComBSTR** object.

See Also

Reference

CComBSTR Class

CComBSTR Methods

For information about the methods in **CComBSTR**, see CComBSTR Members.

CComBSTR::Append

Appends either *lpsz* or the **BSTR** member of *bstrSrc* to m_str.

```
HRESULT Append(
   const CComBSTR& bstrSrc
) throw( );
HRESULT Append(
   wchar_t ch
) throw( );
HRESULT Append(
   char ch
) throw( );
HRESULT Append(
   LPCOLESTR lpsz
) throw( );
HRESULT Append(
   LPCSTR lpsz
) throw( );
HRESULT Append(
   LPCOLESTR lpsz,
   int nLen
) throw( );
```

Parameters

bstrSrc

[in] A **CComBSTR** object to append.

ch

[in] A character to append.

lpsz

[in] A zero-terminated character string to append. You can pass a Unicode string via the **LPCOLESTR** overload or an ANSI string via the **LPCSTR** version.

nLen

[in] The number of characters from *lpsz* to append.

Return Value

S_OK on success, or any standard **HRESULT** error value.

Remarks

An ANSI string will be converted to Unicode before being appended.

Example

```
// CComBSTR example
BOOL bASP, bHTM, bISAPI;

CComBSTR bstrURL = "http://SomeSite/";
CComBSTR bstrDEF = "/OtherSite";

CComBSTR bstrASP = "default.asp";

if (bASP)
    // bstrURL is 'http://SomeSite/default.asp'
    bstrURL.Append(bstrASP);
else if (bHTM)
    // bstrURL is 'http://SomeSite/default.htm'
    bstrURL.Append("default.htm");
else if (bISAPI)
```

```
    // bstrURL is 'http://SomeSite/default.dll?func'
    bstrURL.Append(OLESTR("default.dll?func"));
else
{
    CComBSTR bstrTemp;
    // bstrTemp is 'http://'
    bstrTemp.Append(bstrURL, 7);
    // bstrURL is 'http://OtherSite'
    bstrURL = bstrTemp.Append(bstrDEF);
}
```

See Also

Reference

CComBSTR Class

CComBSTR::AppendBSTR

CComBSTR::operator +=

Other Resources

CComBSTR Members

CComBSTR::AppendBSTR

Appends the specified **BSTR** to m_str.

```
HRESULT AppendBSTR(
   BSTR p
) throw( );
```

Parameters

p

[in] A **BSTR** to append.

Return Value

S_OK on success, or any standard **HRESULT** error value.

Remarks

Do not pass an ordinary wide-character string to this method. The compiler cannot catch the error and run time errors will occur.

Example

```
// Append Example

CComBSTR bstrPre("Hello ");
CComBSTR bstrSuf("World!");

// Appends "World!" to "Hello "
bstrPre.AppendBSTR( bstrSuf );

// Displays a message box with text "Hello World!"
::MessageBox(NULL, CW2CT(bstrPre), NULL, MB_OK);
```

See Also

Reference

CComBSTR Class
CComBSTR::Append
CComBSTR::operator +=

Concepts

ATL and MFC String Conversion Macros

Other Resources

CComBSTR Members

CComBSTR::AppendBytes

Appends the specified number of bytes to m_str without conversion.

```
HRESULT AppendBytes(
   const char* lpsz,
   int nLen
) throw( );
```

Parameters

lpsz

[in] A pointer to an array of bytes to append.

p

[in] The number of bytes to append.

Return Value

S_OK on success, or any standard **HRESULT** error value.

Example

```
// AppendBytes Example

CComBSTR bstrPre("Hello ");

// Appends "Wo" to "Hello " (4 bytes == 2 characters)
bstrPre.AppendBytes( reinterpret_cast<char*>(L"World!"), 4 );

// Displays a message box with text "Hello Wo"
::MessageBox(NULL, CW2CT(bstrPre), NULL, MB_OK);
```

See Also

Reference

CComBSTR Class
CComBSTR::Append
CComBSTR::operator +=
CComBSTR::operator +=

Other Resources

CComBSTR Members

CComBSTR::ArrayToBSTR

Frees any existing string held in the **CComBSTR** object, then creates a **BSTR** from the first character of each element in the safearray and attaches it to the **CComBSTR** object.

```
HRESULT ArrayToBSTR(
   const SAFEARRAY* pSrc
) throw( );
```

Parameters

pSrc

[in] The safearray containing the elements used to create the string.

Return Value

S_OK on success, or any standard **HRESULT** error value.

See Also

Reference

CComBSTR Class

Other Resources

CComBSTR Members

CComBSTR::AssignBSTR

Assigns a **BSTR** to m_str.

```
HRESULT AssignBSTR(
   const BSTR bstrSrc
) throw( );
```

Parameters

bstrSrc

[in] A **BSTR** to assign to the current **CComBSTR** object.

Return Value

S_OK on success, or any standard **HRESULT** error value.

See Also

Reference

CComBSTR Class

Other Resources

CComBSTR Members

CComBSTR::Attach

Attaches a **BSTR** to the **CComBSTR** object by setting the m_str member to *src*.

```
void Attach(
   BSTR src
) throw( );
```

Parameters

src

[in] The **BSTR** to attach to the object.

Remarks

Do not pass an ordinary wide-character string to this method. The compiler cannot catch the error and run time errors will occur.

Note
This method will assert if **m_str** is non-**NULL**.

Example

```
// STDMETHOD(BSTRToUpper)(/*[in, out]*/ BSTR bstrConv);
STDMETHODIMP CPolyCtl::BSTRToUpper(BSTR bstrConv)
{
    if (bstrConv == NULL)
        return E_POINTER;

    // Assign bstrConv to m_str member of CComBSTR
    CComBSTR bstrTemp;
    bstrTemp.Attach(bstrConv);

    // Make string uppercase
    bstrTemp.ToUpper();

    // Set m_str to NULL, so the BSTR is not freed
    bstrTemp.Detach();

    return S_OK;
}
```

See Also

Reference

CComBSTR Class
CComBSTR::Detach
CComBSTR::operator =

Other Resources

CComBSTR Members

CComBSTR::BSTRToArray

Creates a zero-based one-dimensional safearray, where each element of the array is a character from the **CComBSTR** object.

```
HRESULT BSTRToArray(
   LPSAFEARRAY* ppArray
) throw( );
```

Parameters

ppArray

[out] The pointer to the safearray used to hold the results of the function.

Return Value

S_OK on success, or any standard **HRESULT** error value.

See Also

Reference

CComBSTR Class

Other Resources

CComBSTR Members

CComBSTR::ByteLength

Returns the number of bytes in **m_str**, excluding the terminating null character.

```
unsigned int ByteLength( ) const throw( );
```

Return Value

The length of the m_str member in bytes.

Remarks

Returns 0 if **m_str** is **NULL**.

Example

```
// string with 11 chars (22 bytes)
CComBSTR bstrTemp("Hello World");

unsigned int len = bstrTemp.ByteLength();

_ASSERTE(len == 22);
```

See Also

Reference

CComBSTR Class

CComBSTR::Length

Other Resources

CComBSTR Members

CComBSTR::CComBSTR

The constructor. The default constructor sets the m_str member to **NULL**.

```
CComBSTR( ) throw( );
CComBSTR(
   const CComBSTR& src
);
CComBSTR(
   REFGUID guid
);
CComBSTR(
   int nSize
);
CComBSTR(
   int nSize,
   LPCOLESTR sz
);
CComBSTR(
   int nSize,
   LPCSTR sz
);
CComBSTR(
   LPCOLESTR pSrc
);
CComBSTR(
   LPCSTR pSrc
);
```

Parameters

nSize

[in] The number of characters to copy from *sz* or the initial size in characters for the **CComBSTR**.

sz

[in] A string to copy. The Unicode version specifies an **LPCOLESTR**; the ANSI version specifies an **LPCSTR**.

pSrc

[in] A string to copy. The Unicode version specifies an **LPCOLESTR**; the ANSI version specifies an **LPCSTR**.

src

[in] A **CComBSTR** object.

guid

[in] A reference to a **GUID** structure.

Remarks

The copy constructor sets **m_str** to a copy of the **BSTR** member of *src*. The **REFGUID** constructor converts the **GUID** to a string using **StringFromGUID2** and stores the result.

The other constructors set **m_str** to a copy of the specified string. If you pass a value for *nSize*, then only *nSize* characters will be copied, followed by a terminating null character.

The destructor frees the string pointed to by **m_str**.

Example

```
CComBSTR bstr1;   // BSTR points to NULL
bstr1 = "Bye";    // initialize with assignment operator

OLECHAR* str = OLESTR("ta ta");  // wide char string of length 5
CComBSTR bstr2(wcslen(str));     // unintialized BSTR of length 5
```

```
wcscpy(bstr2.m_str, str);        // copy wide char string to BSTR

CComBSTR bstr3(5, OLESTR("Hello World")); // BSTR containing 'Hello',
                                          // input string is wide char
CComBSTR bstr4(5, "Hello World");         // same as above, input string
                                          // is ANSI

CComBSTR bstr5(OLESTR("Hey there")); // BSTR containing 'Hey there',
                                     // input string is wide char
CComBSTR bstr6("Hey there");         // same as above, input string
                                     // is ANSI

CComBSTR bstr7(bstr6);     // copy constructor, bstr7 contains 'Hey there'
```

See Also

Reference

CComBSTR Class

Other Resources

CComBSTR Members

CComBSTR Members

CComBSTR::~CComBSTR

The destructor.

```
~CComBSTR( );
```

Remarks

The destructor frees the string pointed to by **m_str**.

See Also

Reference

CComBSTR Class

Other Resources

CComBSTR Members

CComBSTR::Copy

Allocates and returns a copy of **m_str**.

```
BSTR Copy( ) const throw();
```

Return Value

A copy of the m_str member. If **m_str** is **NULL**, returns **NULL**.

Example

```
CComBSTR m_bstrURL;    // BSTR representing a URL

// put_URL is the put method for the URL property.
STDMETHOD(put_URL)(BSTR strURL)
{
    ATLTRACE(_T("put_URL\n"));

    // free existing string in m_bstrURL & make a copy
    // of strURL pointed to by m_bstrURL
    m_bstrURL = strURL;
    return S_OK;
}

// get_URL is the get method for the URL property.
STDMETHOD(get_URL)(BSTR* pstrURL)
{
    ATLTRACE(_T("get_URL\n"));

    // make a copy of m_bstrURL pointed to by pstrURL
    *pstrURL = m_bstrURL.Copy(); // See CComBSTR::CopyTo
    return S_OK;
}
```

See Also

Reference

CComBSTR Class

CComBSTR::CopyTo

CComBSTR::operator =

Other Resources

CComBSTR Members

CComBSTR::CopyTo

Allocates and returns a copy of m_str via the parameter.

```
HRESULT CopyTo(
   BSTR* pbstr
) throw( );
HRESULT CopyTo(
   VARIANT* pvarDest
) throw( );
```

Parameters

pbstr

[out] The address of a **BSTR** in which to return the string allocated by this method.

pvarDest

[out] The address of a **VARIANT** in which to return the string allocated by this method.

Return Value

A standard **HRESULT** value indicating the success or failure of the copy.

Remarks

After calling this method, the **VARIANT** pointed to by *pvarDest* will be of type **VT_BSTR**.

Example

```
CComBSTR m_bstrURL; // BSTR representing a URL

// get_URL is the get method for the URL property.
STDMETHOD(get_URL)(BSTR* pstrURL)
{
    // Make a copy of m_bstrURL and return it via pstrURL
    return m_bstrURL.CopyTo(pstrURL);
}
```

See Also

Reference

CComBSTR Class
CComBSTR::Copy
CComBSTR::operator =

Other Resources

CComBSTR Members

CComBSTR::Detach

Detaches m_str from the **CComBSTR** object and sets **m_str** to **NULL**.

```
BSTR Detach( ) throw( );
```

Return Value

The **BSTR** associated with the **CComBSTR** object.

Example

```
// Method which converts bstrIn to uppercase
STDMETHODIMP CPolyCtl::BSTRToUpper(BSTR bstrIn, BSTR* pbstrOut)
{
    if (bstrIn == NULL || pbstrOut == NULL)
        return E_POINTER;

    // Create a temporary copy of bstrIn
    CComBSTR bstrTemp(bstrIn);

    if (!bstrTemp)
        return E_OUTOFMEMORY;

    // Make string uppercase
    bstrTemp.ToUpper();

    // Return m_str member of bstrTemp
    *pbstrOut = bstrTemp.Detach();

    return S_OK;
}
```

See Also

Reference

CComBSTR Class

CComBSTR::Attach

Other Resources

CComBSTR Members

CComBSTR::Empty

Frees the m_str member.

```
void Empty( ) throw( );
```

Example

```
// Example for CComBSTR::Empty
CComBSTR bstr( "abc" );

// Calls SysFreeString to free the BSTR
bstr.Empty();
_ASSERTE( bstr.Length( ) == 0 );
```

See Also

Reference

CComBSTR Class

Other Resources

CComBSTR Members

CComBSTR::Length

Returns the number of characters in **m_str**, excluding the terminating null character.

```
unsigned int Length( ) const throw( );
```

Return Value

The length of the m_str member.

Example

```
// string with 11 chars
CComBSTR bstrTemp("Hello World");

unsigned int len = bstrTemp.Length();

_ASSERTE(len == 11);
```

See Also

Reference

CComBSTR Class

Other Resources

CComBSTR Members

CComBSTR::LoadString

Loads a string resource specified by *nID* and stores it in this object.

```
bool LoadString(
   HINSTANCE hInst,
   UINT nID
) throw();
bool LoadString(
   UINT nID
) throw();
```

Parameters

See LoadString in the *Platform SDK*.

Return Value

Returns **true** if the string is successfully loaded; otherwise, returns **false**.

Remarks

The first function loads the resource from the module identified by you via the *hInst* parameter. The second function loads the resource from the resource module associated with the CComModule-derived object used in this project.

Example

```
// Loadstring Example

CComBSTR bstrTemp;

// IDS_PROJNAME proj name stored as resource in string table
bstrTemp.LoadString(IDS_PROJNAME);

// the above is equivalent to:
// bstrTemp.LoadString(_Module.m_hInstResource, IDS_PROJNAME);

// display message box w/ proj name as title & text
::MessageBox(NULL, CW2CT(bstrTemp), CW2CT(bstrTemp), MB_OK);
```

See Also

Reference

CComBSTR Class

Other Resources

CComBSTR Members

CComBSTR Members

CComBSTR::ReadFromStream

Sets the m_str member to the **BSTR** contained in the specified stream.

```
HRESULT ReadFromStream(
   IStream* pStream
) throw( );
```

Parameters

pStream

[in] A pointer to the IStream interface on the stream containing the data.

Return Value

A standard **HRESULT** value.

Remarks

ReadToStream requires the contents of the stream at the current position to be compatible with the data format written out by a call to WriteToStream.

Example

```
void CMyView::OnEditCopy()
{
    IDataObject* pDataObj;

    // Fill in the FORMATETC struct to retrieve desired format
    // from clipboard
    FORMATETC formatetcIn = {CF_TEXT, NULL, DVASPECT_CONTENT, -1,
        TYMED_ISTREAM};
    STGMEDIUM medium;
    ZeroMemory(&medium, sizeof(STGMEDIUM));

    // Get IDataObject from clipboard
    HRESULT hr = ::OleGetClipboard(&pDataObj);

    // Retrieve data from clipboard
    hr = pDataObj->GetData(&formatetcIn, &medium);

    if ( SUCCEEDED(hr) && medium.tymed == TYMED_ISTREAM)
    {
        CComBSTR bstrStr;
        // Get BSTR out of the stream
        hr = bstrStr.ReadFromStream(medium.pstm);

        //release the stream
        ::ReleaseStgMedium(&medium);
    }

}
```

See Also

Reference

CComBSTR Class

Other Resources

CComBSTR Members

CComBSTR::ToLower

Converts the contained string to lowercase.

```
HRESULT ToLower( ) throw( );
```

Return Value

A standard **HRESULT** value.

Remarks

See **CharLowerBuff** for more information on how the conversion is performed.

See Also

Reference

CComBSTR Class

CComBSTR::ToUpper

Other Resources

CComBSTR Members

CComBSTR::ToUpper

Converts the contained string to uppercase.

```
HRESULT ToUpper( ) throw( );
```

Return Value

A standard **HRESULT** value.

Remarks

See **CharUpperBuff** for more information on how the conversion is performed.

See Also

Reference

CComBSTR Class

CComBSTR::ToLower

Other Resources

CComBSTR Members

CComBSTR::WriteToStream

Saves the m_str member to a stream.

```
HRESULT WriteToStream(
   IStream* pStream
) throw( );
```

Parameters

pStream

[in] A pointer to the IStream interface on a stream.

Return Value

A standard **HRESULT** value.

Remarks

You can recreate a BSTR from the contents of the stream using the ReadFromStream function.

Example

```
//implementation of IDataObject::GetData()
STDMETHODIMP CPolyCtl::GetData(FORMATETC *pformatetcIn, STGMEDIUM *pmedium)
{
    HRESULT hr = S_OK;
    if (pformatetcIn->cfFormat == CF_TEXT && pformatetcIn->tymed == TYMED_ISTREAM)
    {
        IStream *pStm;
        // Create an IStream from global memory
        HRESULT hr = CreateStreamOnHGlobal(NULL, TRUE, &pStm);
        if (FAILED(hr))
            return hr;

        // Initialize CComBSTR
        CComBSTR bstrStr = "Hello World";

        // Serialize string into stream
        // the length followed by actual string is serialized into stream
        hr = bstrStr.WriteToStream(pStm);

        // Pass the IStream pointer back through STGMEDIUM struct
        pmedium->tymed = TYMED_ISTREAM;
        pmedium->pstm = pStm;
        pmedium->pUnkForRelease = NULL;
    }

    return hr;
}
```

See Also

Reference

CComBSTR Class

CComBSTR::ReadFromStream

Other Resources

CComBSTR Members

CComBSTR Operators

For information about the operators in **CComBSTR**, see CComBSTR Members.

CComBSTR::operator !

Checks whether **BSTR** string is NULL.

```
bool operator !( ) const throw( );
```

Return Value

Returns **true** if the m_str member is **NULL**; otherwise, **false**.

Remarks

This operator only checks for a NULL value, not for an empty string.

Example

```
STDMETHODIMP CPolyCtl::BSTRToUpper(BSTR bstrConv)
{
    // Assign bstrConv to m_str member of CComBSTR
    CComBSTR bstrTemp;
    bstrTemp.Attach(bstrConv);

    // Make sure BSTR is not NULL string
    if (!bstrTemp)
        return E_POINTER;

    // Make string uppercase
    bstrTemp.ToUpper();

    // Set m_str to NULL, so the BSTR is not freed
    bstrTemp.Detach();

    return S_OK;
}
```

See Also

Reference

CComBSTR Class

Other Resources

CComBSTR Members

CComBSTR::operator !=

Returns the logical opposite of operator ==.

```
bool operator!=(
   const CComBSTR& bstrSrc
) const throw( );
bool operator!=(
   LPCOLESTR pszSrc
) const;
bool operator!=(
   LPCSTR pszSrc
) const;
bool operator!=(
   int nNull
) const throw( );
```

Parameters

bstrSrc

[in] A **CComBSTR** object.

pszSrc

[in] A zero-terminated string.

nNull

[in] Must be **NULL**.

Return Value

Returns **true** if the item being compared is not equal to the **CComBSTR** object; otherwise, returns **false**.

Remarks

CComBSTRs are compared textually in the context of the user's default locale. The final comparison operator just compares the contained string against **NULL**.

See Also

Reference

CComBSTR Class

Other Resources

CComBSTR Members

CComBSTR::operator &

Returns the address of the **BSTR** stored in the m_str member.

```
BSTR* operator &( ) throw( );
```

Remarks

A special assertion was inserted to CComBstr operator &. It asserts when the m_str member initialized. The purpose of that assertion is to prevent memory leak. The problematic scenario is when the user uses that & operator to realloc the m_str member with out freeing the first previous allocation of m_str.

In case that m_str wasn't allocated yet, there is no assert

The user can define ATL_NO_CCOMBSTR_ADDRESS_OF_ASSERT to avoid that assertion.

Example

```
void MyInitFunction(BSTR* pbsr)
{
*pbsr = SysReAllocString(L"Hello World");
return;
}
CComBSTR bstrStr ;
MyInitFunction(&bstrStr);// in that case bstrStr not initialzed
   // so there is no assertion

...

CComBSTR bstrStr2 = "Hello World";
::SysReAllocString(&bstrStr2, OLESTR("Bye"));// in that case
   // bstrStr2 initialized there is assertion.
```

See Also

Reference

CComBSTR Class

Other Resources

CComBSTR Members

CComBSTR::operator +=

Appends a string to the **CComBSTR** object.

```
CComBSTR& operator +=(
   const CComBSTR& bstrSrc
);
CComBSTR& operator +=(
   const LPCOLESTR pszSrc
);
```

Parameters

bstrSrc

[in] A **CComBSTR** object to append.

pszSrc

[in] A zero-terminated string to append.

Remarks

CComBSTRs are compared textually in the context of the user's default locale. The **LPCOLESTR** comparison is done using **memcmp** on the raw data in each string. The **LPCSTR** comparison is carried out in the same way once a temporary Unicode copy of *pszSrc* has been created. The final comparison operator just compares the contained string against **NULL**.

Example

```
// += Example

CComBSTR bstrPre("Hello ");
CComBSTR bstrSuf("World!");

// Appends "World!" to "Hello "
bstrPre += bstrSuf;

// Displays a message box with text "Hello World!"
::MessageBox(NULL, CW2CT(bstrPre), NULL, MB_OK);
```

See Also

Reference

CComBSTR Class
CComBSTR::Append
CComBSTR::AppendBSTR

Concepts

ATL and MFC String Conversion Macros

Other Resources

CComBSTR Members

CComBSTR::operator <

Compares a **CComBSTR** with a string.

```
bool operator <(
   const CComBSTR& bstrSrc
) const throw( );
bool operator <(
   LPCOLESTR pszSrc
) const throw( );
bool operator <(
   LPCSTR pszSrc
) const throw( );
```

Return Value

Returns **true** if the item being compared is less than the **CComBSTR** object; otherwise, returns **false**.

Remarks

The comparison is performed using the user's default locale.

See Also

Reference

CComBSTR Class

Other Resources

CComBSTR Members

CComBSTR::operator =

Sets the m_str member to a copy of *pSrc* or to a copy of the **BSTR** member of *src*.

```
CComBSTR& operator =(
   const CComBSTR& src
);
CComBSTR& operator =(
   LPCOLESTR pSrc
);
CComBSTR& operator =(
   LPCSTR pSrc
);
```

Remarks

The *pSrc* parameter specifies either an **LPCOLESTR** for Unicode versions or **LPCSTR** for ANSI versions.

Example

See the example for CComBSTR::Copy.

See Also

Reference

CComBSTR Class

CComBSTR::Copy

Other Resources

CComBSTR Members

CComBSTR::operator ==

Compares a **CComBSTR** with a string. **CComBSTR**s are compared textually in the context of the user's default locale.

```
bool operator ==(
   const CComBSTR& bstrSrc
) const throw( );
bool operator ==(
   LPCOLESTR pszSrc
) const;
bool operator ==(
   LPCSTR pszSrc
) const;
bool operator ==(
   int nNull
) const throw( );
```

Parameters

bstrSrc

[in] A **CComBSTR** object.

pszSrc

[in] A zero-terminated string.

nNull

[in] Must be **NULL**.

Return Value

Returns **true** if the item being compared is equal to the **CComBSTR** object; otherwise, returns **false**.

Remarks

The final comparison operator just compares the contained string against **NULL**.

See Also

Reference

CComBSTR Class

Other Resources

CComBSTR Members

CComBSTR::operator >

Compares a **CComBSTR** with a string.

```
bool operator >(
   const CComBSTR& bstrSrc
) const throw( );
```

Return Value

Returns **true** if the item being compared is greater than the **CComBSTR** object; otherwise, returns **false**.

Remarks

The comparison is performed using the user's default locale.

See Also

Reference

CComBSTR Class

Other Resources

CComBSTR Members

CComBSTR::operator BSTR

Casts a **CComBSTR** object to a **BSTR**.

```
operator BSTR( ) const throw( );
```

Remarks

Allows you to pass **CComBSTR** objects to functions that have **[in] BSTR** parameters.

Example

See the example for CComBSTR::m_str.

See Also

Reference

CComBSTR Class

Other Resources

CComBSTR Members

CComBSTR Data Members

For information about the data members in **CComBSTR**, see CComBSTR Members.

CComBSTR::m_str

Contains the **BSTR** associated with the **CComBSTR** object.

```
BSTR m_str;
```

Example

```
CComBSTR GuidToBSTR(REFGUID guid)
{
    // 39 - length of string representation of GUID + 1
    CComBSTR b(39, L"");

    // Convert GUID to BSTR
    // m_str member of CComBSTR is of type BSTR. When BSTR param
    // is required, pass the m_str member explicitly or use implicit
    // BSTR cast operator.
    int nRet = StringFromGUID2(guid, b.m_str, 39);

    // Above equivalent to:
    // int nRet = StringFromGUID2(guid, b, 39);
    // implicit BSTR cast operator used for 2nd param

    // Both lines are equivalent to:
    // CComBSTR b(guid);
    // CComBSTR constructor can convert GUIDs

    _ASSERTE(nRet);
    return b;
}
```

See Also

Reference

CComBSTR Class

Other Resources

CComBSTR Members

CComCachedTearOffObject Class

This class implements IUnknown for a tear-off interface.

```
template <
   class contained
>
class CComCachedTearOffObject : public IUnknown,
      public CComObjectRootEx< contained::_ThreadModel::ThreadModelNoCS >
```

Parameters

contained

Your tear-off class, derived from **CComTearOffObjectBase** and the interfaces you want your tear-off object to support.

Remarks

CComCachedTearOffObject implements IUnknown for a tear-off interface. This class differs from **CComTearOffObject** in that **CComCachedTearOffObject** has its own **IUnknown**, separate from the owner object's **IUnknown** (the owner is the object for which the tear-off is being created). **CComCachedTearOffObject** maintains its own reference count on its **IUnknown** and deletes itself once its reference count is zero. However, if you query for any of its tear-off interfaces, the reference count of the owner object's **IUnknown** will be incremented.

If the **CComCachedTearOffObject** object implementing the tear-off is already instantiated, and the tear-off interface is queried for again, the same **CComCachedTearOffObject** object is reused. In contrast, if a tear-off interface implemented by a **CComTearOffObject** is again queried for through the owner object, another **CComTearOffObject** will be instantiated.

The owner class must implement **FinalRelease** and call **Release** on the cached **IUnknown** for the **CComCachedTearOffObject**, which will decrement its reference count. This will cause **CComCachedTearOffObject**'s **FinalRelease** to be called and delete the tear-off.

Requirements

Header: atlcom.h

See Also

Reference

CComTearOffObject Class

CComObjectRootEx Class

Other Resources

CComCachedTearOffObject Members

ATL Class Overview

CComCachedTearOffObject Members

Methods

AddRef	Increments the reference count for a **CComCachedTearOffObject** object.
CComCachedTearOffObject	The constructor.
~CComCachedTearOffObject	The destructor.
FinalConstruct	Calls the **m_contained::FinalConstruct** (the tear-off class' method).
FinalRelease	Calls the **m_contained::FinalRelease** (the tear-off class' method).
QueryInterface	Returns a pointer to the **IUnknown** of the **CComCachedTearOffObject** object, or to the requested interface on your tear-off class (the class *contained*).
Release	Decrements the reference count for a **CComCachedTearOffObject** object and destroys it if the reference count is 0.

Data Members

m_contained	A **CComContainedObject** object derived from your tear-off class (the class *contained*).

See Also

Reference

CComCachedTearOffObject Class

CComCachedTearOffObject Methods

For information about the methods in **CComCachedTearOffObject**, see CComCachedTearOffObject Members.

CComCachedTearOffObject::AddRef

Increments the reference count of the **CComCachedTearOffObject** object by 1.

```
STDMETHOD_(ULONG, AddRef)( );
```

Return Value

A value that may be useful for diagnostics and testing.

See Also

Reference

CComCachedTearOffObject Class

CComCachedTearOffObject::Release

Other Resources

CComCachedTearOffObject Members

CComCachedTearOffObject::CComCachedTearOffObject

The constructor.

```
CComCachedTearOffObject(
   void* pv
);
```

Parameters

pv

[in] Pointer to the **IUnknown** of the **CComCachedTearOffObject**.

Remarks

Initializes the **CComContainedObject** member, m_contained.

See Also

Reference

CComCachedTearOffObject Class

CComTearOffObject Class

Other Resources

CComCachedTearOffObject Members

CComCachedTearOffObject::~CComCachedTearOffObject

The destructor.

```
~CComCachedTearOffObject( );
```

Remarks

Frees all allocated resources and calls FinalRelease.

See Also

Reference

CComCachedTearOffObject Class

CComTearOffObject Class

Other Resources

CComCachedTearOffObject Members

CComCachedTearOffObject::FinalConstruct

Calls **m_contained::FinalConstruct** to create **m_contained**, the **CComContainedObject**<*contained*> object used to access the interface implemented by your tear-off class.

```
HRESULT FinalConstruct( );
```

Return Value

A standard **HRESULT** value.

See Also

Reference

CComCachedTearOffObject Class

CComCachedTearOffObject::FinalRelease

Other Resources

CComCachedTearOffObject Members

CComCachedTearOffObject::FinalRelease

Calls **m_contained::FinalRelease** to free **m_contained**, the **CComContainedObject**<*contained*> object.

```
void FinalRelease( );
```

See Also

Reference

CComCachedTearOffObject Class

CComCachedTearOffObject::FinalConstruct

Other Resources

CComCachedTearOffObject Members

CComCachedTearOffObject::QueryInterface

Retrieves a pointer to the requested interface.

```
STDMETHOD(QueryInterface)(
   REFIID iid ,
   void** ppvObject
);
```

Parameters

iid

[in] The GUID of the interface being requested.

ppvObject

[out] A pointer to the interface pointer identified by *iid*, or **NULL** if the interface is not found.

Return Value

A standard **HRESULT** value.

Remarks

If the requested interface is **IUnknown**, returns a pointer to the **CComCachedTearOffObject**'s own **IUnknown** and increments the reference count. Otherwise, queries for the interface on your tear-off class using the InternalQueryInterface method inherited from **CComObjectRootEx**.

See Also

Reference

CComCachedTearOffObject Class
CComCachedTearOffObject::AddRef
CComCachedTearOffObject::Release

Other Resources

CComCachedTearOffObject Members

CComCachedTearOffObject::Release

Decrements the reference count by 1 and, if the reference count is 0, deletes the **CComCachedTearOffObject** object.

```
STDMETHOD_(ULONG, Release)( );
```

Return Value

In non-debug builds, always returns 0. In debug builds, returns a value that may be useful for diagnostics or testing.

See Also

Reference

CComCachedTearOffObject Class

CComCachedTearOffObject::AddRef

Other Resources

CComCachedTearOffObject Members

CComCachedTearOffObject Data Members

For information about the data members in **CComCachedTearOffObject**, see CComCachedTearOffObject Members.

CComCachedTearOffObject::m_contained

A CComContainedObject object derived from your tear-off class.

```
CcomContainedObject <contained> m_contained;
```

Parameters

contained

[in] Your tear-off class, derived from **CComTearOffObjectBase** and the interfaces you want your tear-off object to support.

Remarks

The methods **m_contained** inherits are used to access the tear-off interface in your tear-off class through the cached tear-off object's **QueryInterface**, **FinalConstruct**, and **FinalRelease**.

See Also

Reference

CComCachedTearOffObject Class

CComTearOffObject Class

Other Resources

CComCachedTearOffObject Members

CComClassFactory Class

This class implements the IClassFactory interface.

```
class CComClassFactory : public IClassFactory,
      public CComObjectRootEx< CComGlobalsThreadModel >
```

Remarks

CComClassFactory implements the IClassFactory interface, which contains methods for creating an object of a particular CLSID, as well as locking the class factory in memory to allow new objects to be created more quickly. **IClassFactory** must be implemented for every class that you register in the system registry and to which you assign a CLSID.

ATL objects normally acquire a class factory by deriving from CComCoClass. This class includes the macro DECLARE_CLASSFACTORY, which declares **CComClassFactory** as the default class factory. To override this default, specify one of the **DECLARE_CLASSFACTORY***XXX* macros in your class definition. For example, the DECLARE_CLASSFACTORY_EX macro uses the specified class for the class factory:

```
class CMyClass : ..., public CComCoClass< ... >
{
public:
   DECLARE_CLASSFACTORY_EX(CMyClassFactory)

   ...
};
```

The above class definition specifies that **CMyClassFactory** will be used as the object's default class factory. **CMyClassFactory** must derive from **CComClassFactory** and override **CreateInstance**.

ATL provides three other macros that declare a class factory:

- DECLARE_CLASSFACTORY2 Uses CComClassFactory2, which controls creation through a license.
- DECLARE_CLASSFACTORY_AUTO_THREAD Uses CComClassFactoryAutoThread, which creates objects in multiple apartments.
- DECLARE_CLASSFACTORY_SINGLETON Uses CComClassFactorySingleton, which constructs a single CComObjectGlobal object.

Requirements

Header: atlcom.h

See Also

Reference

CComObjectRootEx Class

CComGlobalsThreadModel

Other Resources

CComClassFactory Members

ATL Class Overview

CComClassFactory Members

IClassFactory Methods

CreateInstance	Creates an object of the specified CLSID.
LockServer	Locks the class factory in memory.

See Also

Reference

CComClassFactory Class

CComClassFactory Methods

For information about the methods in **CComClassFactory**, see CComClassFactory Members.

CComClassFactory::CreateInstance

Creates an object of the specified CLSID and retrieves an interface pointer to this object.

```
STDMETHOD(CreateInstance)(
   LPUNKNOWN pUnkOuter,
   REFIID riid,
   void** ppvObj
);
```

Parameters

pUnkOuter

[in] If the object is being created as part of an aggregate, then *pUnkOuter* must be the outer unknown. Otherwise, *pUnkOuter* must be **NULL**.

riid

[in] The IID of the requested interface. If *pUnkOuter* is non-**NULL**, *riid* must be **IID_IUnknown**.

ppvObj

[out] A pointer to the interface pointer identified by *riid*. If the object does not support this interface, *ppvObj* is set to **NULL**.

Return Value

A standard **HRESULT** value.

See Also

Reference

CComClassFactory Class
CoCreateInstance
CoGetClassObject

Other Resources

CComClassFactory Members

CComClassFactory::LockServer

Increments and decrements the module lock count by calling **_Module::Lock** and **_Module::Unlock**, respectively.

```
STDMETHOD(LockServer)(
   BOOL fLock
);
```

Parameters

fLock

[in] If **TRUE**, the lock count is incremented; otherwise, the lock count is decremented.

Return Value

A standard **HRESULT** value.

Remarks

_Module refers to the global instance of CComModule or a class derived from it.

Calling **LockServer** allows a client to hold onto a class factory so that multiple objects can be created quickly.

See Also

Reference

CComClassFactory Class

Other Resources

CComClassFactory Members

CComClassFactory2 Class

This class implements the IClassFactory2 interface.

```
template <
   class license
>
class CComClassFactory2 : public IClassFactory2,
   public CComObjectRootEx<CComGlobalsThreadModel>,
   public license
```

Parameters

license

A class that implements the following static functions:

- **static BOOL VerifyLicenseKey(BSTR** *bstr* **);**
- **static BOOL GetLicenseKey(DWORD** *dwReserved* **, BSTR*** *pBstr* **);**
- **static BOOL IsLicenseValid( );**

Remarks

CComClassFactory2 implements the IClassFactory2 interface, which is an extension of IClassFactory. **IClassFactory2** controls object creation through a license. A class factory executing on a licensed machine can provide a run-time license key. This license key allows an application to instantiate objects when a full machine license does not exist.

ATL objects normally acquire a class factory by deriving from CComCoClass. This class includes the macro DECLARE_CLASSFACTORY, which declares CComClassFactory as the default class factory. To use **CComClassFactory2**, specify the DECLARE_CLASSFACTORY2 macro in your object's class definition. For example:

```
class CMyClass : ..., public CComCoClass< ... >
{
public:
   DECLARE_CLASSFACTORY2(CMyLicense)

   ...
};
```

CMyLicense, the template parameter to **CComClassFactory2**, must implement the static functions **VerifyLicenseKey**, **GetLicenseKey**, and **IsLicenseValid**. The following is an example of a simple license class:

```
class CMyLicense
{
protected:
   static BOOL VerifyLicenseKey(BSTR bstr)
   {
      USES_CONVERSION;
      return !lstrcmp(OLE2T(bstr), _T("My run-time license key"));
   }

   static BOOL GetLicenseKey(DWORD dwReserved, BSTR* pBstr)
   {
      USES_CONVERSION;
      *pBstr = SysAllocString( T2OLE(_T("My run-time license key")));
      return TRUE;
   }

   static BOOL IsLicenseValid() {  return TRUE; }
};
```

CComClassFactory2 derives from both **CComClassFactory2Base** and *license*. **CComClassFactory2Base**, in turn, derives from **IClassFactory2** and **CComObjectRootEx< CComGlobalsThreadModel >**.

Requirements

Header: atlcom.h

See Also

Reference

CComClassFactoryAutoThread Class
CComClassFactorySingleton Class
CComObjectRootEx Class
CComGlobalsThreadModel

Other Resources

CComClassFactory2 Members
ATL Class Overview

CComClassFactory2 Members

IClassFactory Methods

CreateInstance	Creates an object of the specified CLSID.
LockServer	Locks the class factory in memory.

IClassFactory2 Methods

CreateInstanceLic	Given a license key, creates an object of the specified CLSID.
GetLicInfo	Retrieves information describing the licensing capabilities of the class factory.
RequestLicKey	Creates and returns a license key.

See Also

Reference

CComClassFactory2 Class

CComClassFactory2 Methods

For information about the methods in **CComClassFactory2**, see CComClassFactory2 Members.

CComClassFactory2::CreateInstance

Creates an object of the specified CLSID and retrieves an interface pointer to this object.

```
STDMETHOD(CreateInstance)(
   LPUNKNOWN pUnkOuter,
   REFIID riid,
   void** ppvObj
);
```

Parameters

pUnkOuter

[in] If the object is being created as part of an aggregate, then *pUnkOuter* must be the outer unknown. Otherwise, *pUnkOuter* must be **NULL**.

riid

[in] The IID of the requested interface. If *pUnkOuter* is non-**NULL**, *riid* must be **IID_IUnknown**.

ppvObj

[out] A pointer to the interface pointer identified by *riid*. If the object does not support this interface, *ppvObj* is set to **NULL**.

Return Value

A standard **HRESULT** value.

Remarks

Requires the machine to be fully licensed. If a full machine license does not exist, call CreateInstanceLic.

See Also

Reference

CComClassFactory2 Class
CoCreateInstance
CoGetClassObject

Other Resources

CComClassFactory2 Members

CComClassFactory2::CreateInstanceLic

Similar to CreateInstance, except that **CreateInstanceLic** requires a license key.

```
STDMETHOD(CreateInstanceLic)(
   IUnknown* pUnkOuter,
   IUnknown* /* pUnkReserved */,
   REFIID riid,
   BSTR bstrKey,
   void** ppvObject
);
```

Parameters

pUnkOuter

[in] If the object is being created as part of an aggregate, then *pUnkOuter* must be the outer unknown. Otherwise, *pUnkOuter* must be **NULL**.

pUnkReserved

[in] Not used. Must be **NULL**.

riid

[in] The IID of the requested interface. If *pUnkOuter* is non-**NULL**, *riid* must be **IID_IUnknown**.

bstrKey

[in] The run-time license key previously obtained from a call to **RequestLicKey**. This key is required to create the object.

ppvObject

[out] A pointer to the interface pointer specified by *riid*. If the object does not support this interface, *ppvObject* is set to **NULL**.

Return Value

A standard **HRESULT** value.

Remarks

You can obtain a license key using RequestLicKey. In order to create an object on an unlicensed machine, you must call **CreateInstanceLic**.

See Also

Reference

CComClassFactory2 Class
CoCreateInstance
CoGetClassObject

Other Resources

CComClassFactory2 Members

CComClassFactory2::GetLicInfo

Fills a LICINFO structure with information that describes the class factory's licensing capabilities.

```
STDMETHOD(GetLicInfo)(
   LICINFO* pLicInfo
);
```

Parameters

pLicInfo

[out] Pointer to a **LICINFO** structure.

Return Value

A standard **HRESULT** value.

Remarks

The *fRuntimeKeyAvail* member of this structure indicates whether, given a license key, the class factory allows objects to be created on an unlicensed machine. The *fLicVerified* member indicates whether a full machine license exists.

See Also

Reference

CComClassFactory2 Class
CComClassFactory2::RequestLicKey
CComClassFactory2::CreateInstanceLic

Other Resources

CComClassFactory2 Members

CComClassFactory2::LockServer

Increments and decrements the module lock count by calling **_Module::Lock** and **_Module::Unlock**, respectively.

```
STDMETHOD(LockServer)(
   BOOL fLock
);
```

Parameters

fLock

[in] If **TRUE**, the lock count is incremented; otherwise, the lock count is decremented.

Return Value

A standard **HRESULT** value.

Remarks

_Module refers to the global instance of CComModule or a class derived from it.

Calling **LockServer** allows a client to hold onto a class factory so that multiple objects can be quickly created.

See Also

Reference

CComClassFactory2 Class

Other Resources

CComClassFactory2 Members

CComClassFactory2::RequestLicKey

Creates and returns a license key, provided that the *fRuntimeKeyAvail* member of the LICINFO structure is **TRUE**.

```
STDMETHOD(RequestLicKey)(
   DWORD dwReserved,
   BSTR* pbstrKey
);
```

Parameters

dwReserved

[in] Not used. Must be zero.

pbstrKey

[out] Pointer to the license key.

Return Value

A standard **HRESULT** value.

Remarks

A license key is required for calling CreateInstanceLic to create an object on an unlicensed machine. If *fRuntimeKeyAvail* is **FALSE**, then objects can only be created on a fully licensed machine.

Call GetLicInfo to retrieve the value of *fRuntimeKeyAvail*.

See Also

Reference

CComClassFactory2 Class

Other Resources

CComClassFactory2 Members

CComClassFactoryAutoThread Class

This class implements the IClassFactory interface, and allows objects to be created in multiple apartments.

```
class CComClassFactoryAutoThread : public IClassFactory,
public CComObjectRootEx< CComGlobalsThreadModel >
```

Remarks

CComClassFactoryAutoThread is similar to CComClassFactory, but allows objects to be created in multiple apartments. To take advantage of this support, derive your EXE module from CComAutoThreadModule.

ATL objects normally acquire a class factory by deriving from CComCoClass. This class includes the macro DECLARE_CLASSFACTORY, which declares CComClassFactory as the default class factory. To use **CComClassFactoryAutoThread**, specify the DECLARE_CLASSFACTORY_AUTO_THREAD macro in your object's class definition. For example:

```
class CMyClass : ..., public CComCoClass< ... >
{
public:
   DECLARE_CLASSFACTORY_AUTO_THREAD( )

   ...
};
```

Requirements

Header: atlcom.h

See Also

Reference

IClassFactory
CComClassFactory2 Class
CComClassFactorySingleton Class
CComObjectRootEx Class
CComGlobalsThreadModel

Other Resources

CComClassFactoryAutoThread Members
ATL Class Overview

CComClassFactoryAutoThread Members

IClassFactory Methods

CreateInstance	Creates an object of the specified CLSID.
LockServer	Locks the class factory in memory.

See Also

Reference

CComClassFactoryAutoThread Class

CComClassFactoryAutoThread Methods

For information about the methods in **CComClassFactoryAutoThread**, see CComClassFactoryAutoThread Members.

CComClassFactoryAutoThread::CreateInstance

Creates an object of the specified CLSID and retrieves an interface pointer to this object.

```
STDMETHODIMP CreateInstance(
   LPUNKNOWN pUnkOuter,
   REFIID riid,
   void** ppvObj
);
```

Parameters

pUnkOuter

[in] If the object is being created as part of an aggregate, then *pUnkOuter* must be the outer unknown. Otherwise, *pUnkOuter* must be **NULL**.

riid

[in] The IID of the requested interface. If *pUnkOuter* is non-**NULL**, *riid* must be **IID_IUnknown**.

ppvObj

[out] A pointer to the interface pointer identified by *riid*. If the object does not support this interface, *ppvObj* is set to **NULL**.

Return Value

A standard **HRESULT** value.

Remarks

If your module derives from CComAutoThreadModule, **CreateInstance** first selects a thread to create the object in the associated apartment.

See Also

Reference

CComClassFactoryAutoThread Class
CoCreateInstance
CoGetClassObject

Other Resources

CComClassFactoryAutoThread Members

CComClassFactoryAutoThread::LockServer

Increments and decrements the module lock count by calling **_Module::Lock** and **_Module::Unlock**, respectively.

```
STDMETHODIMP LockServer(
   BOOL fLock
);
```

Parameters

fLock

[in] If **TRUE**, the lock count is incremented; otherwise, the lock count is decremented.

Return Value

A standard **HRESULT** value.

Remarks

When using **CComClassFactoryAutoThread**, **_Module** typically refers to the global instance of CComAutoThreadModule.

Calling **LockServer** allows a client to hold onto a class factory so that multiple objects can be quickly created.

See Also

Reference

CComClassFactoryAutoThread Class
CComAutoThreadModule::Lock
CComAutoThreadModule::Unlock

Other Resources

CComClassFactoryAutoThread Members

CComClassFactorySingleton Class

This class derives from CComClassFactory and uses CComObjectGlobal to construct a single object.

```
template<
   class T
>
class CComClassFactorySingleton :
   public CComClassFactory
```

Parameters

T

Your class.

CComClassFactorySingleton derives from CComClassFactory and uses CComObjectGlobal to construct a single object. Each call to the **CreateInstance** method simply queries this object for an interface pointer.

Remarks

ATL objects normally acquire a class factory by deriving from CComCoClass. This class includes the macro DECLARE_CLASSFACTORY, which declares **CComClassFactory** as the default class factory. To use **CComClassFactorySingleton**, specify the DECLARE_CLASSFACTORY_SINGLETON macro in your object's class definition. For example:

```
class CMyClass : ..., public CComCoClass< ... >
{
public:
   DECLARE_CLASSFACTORY_SINGLETON(CMyClass)

   ...
};
```

Requirements

Header: atlcom.h

See Also

Reference

IClassFactory
CComClassFactory2 Class
CComClassFactoryAutoThread Class
CComObjectRootEx Class
CComGlobalsThreadModel

Other Resources

CComClassFactorySingleton Members
ATL Class Overview

CComClassFactorySingleton Members

IClassFactory Methods

CreateInstance	Queries **m_spObj** for an interface pointer.

Data Members

m_spObj	The CComObjectGlobal object constructed by **CComClassFactorySingleton**.

See Also

Reference

CComClassFactorySingleton Class

CComClassFactorySingleton Methods

For information about the methods in **CComClassFactorySingleton**, see CComClassFactorySingleton Members.

CComClassFactorySingleton::CreateInstance

Calls **QueryInterface** through m_spObj to retrieve an interface pointer.

```
STDMETHOD(CreateInstance)(
   LPUNKNOWN pUnkOuter,
   REFIID riid,
   void** ppvObj
);
```

Parameters

pUnkOuter

[in] If the object is being created as part of an aggregate, then *pUnkOuter* must be the outer unknown. Otherwise, *pUnkOuter* must be **NULL**.

riid

[in] The IID of the requested interface. If *pUnkOuter* is non-**NULL**, *riid* must be **IID_IUnknown**.

ppvObj

[out] A pointer to the interface pointer identified by *riid*. If the object does not support this interface, *ppvObj* is set to **NULL**.

Return Value

A standard **HRESULT** value.

See Also

Reference

CComClassFactorySingleton Class
CoCreateInstance
CoGetClassObject

Other Resources

CComClassFactorySingleton Members

CComClassFactorySingleton Data Members

For information about the data members in **CComClassFactorySingleton**, see CComClassFactorySingleton Members.

CComClassFactorySingleton::m_spObj

The CComObjectGlobal object constructed by **CComClassFactorySingleton**.

```
CComPtr<IUnknown> m_spObj;
```

Remarks

Each call to the CreateInstance method simply queries this object for an interface pointer.

Note that the current form of **m_spObj** presents a breaking change from the way that **CComClassFactorySingleton** worked in previous versions of ATL. In previous versions the **CComClassFactorySingleton** object was created at the same time as the class factory, during server initialization. In Visual C++.NET 2003, the object is created lazily, on the first request. This change could cause errors in programs that rely on early initialization.

See Also

Reference

CComClassFactorySingleton Class

Other Resources

CComClassFactorySingleton Members

CComCoClass Class

This class provides methods for creating instances of a class, and obtaining its properties.

```
template<
   class T,
   const CLSID* pclsid = &CLSID_NULL
>
class CComCoClass
```

Parameters

T

Your class, derived from **CComCoClass**.

pclsid

A pointer to the CLSID of the object.

Remarks

CComCoClass provides methods for retrieving an object's CLSID, setting error information, and creating instances of the class. Any class registered in the object map should be derived from **CComCoClass**.

CComCoClass also defines the default class factory and aggregation model for your object. **CComCoClass** uses the following two macros:

- DECLARE_CLASSFACTORY Declares the class factory to be CComClassFactory.
- DECLARE_AGGREGATABLE Declares that your object can be aggregated.

You can override either of these defaults by specifying another macro in your class definition. For example, to use CComClassFactory2 instead of **CComClassFactory**, specify the DECLARE_CLASSFACTORY2 macro:

```
class CMyClass : ...,
   public CComCoClass<CMyClass, &CLSID_CMyClass>
{
public:
   DECLARE_CLASSFACTORY2(CMyLicense)

   ...
};
```

Requirements

Header: atlcom.h

See Also

Other Resources

CComCoClass Members

ATL Class Overview

CComCoClass Members

Static Functions

CreateInstance	Creates an instance of the class and queries for an interface.
Error	Returns rich error information to the client.
GetObjectCLSID	Returns the object's class identifier.
GetObjectDescription	Override to return the object's description.

See Also

Reference

CComCoClass Class

CComCoClass Static Functions

For information about the static functions in **CComCoClass**, see CComCoClass Members.

CComCoClass::CreateInstance

Use these **CreateInstance** functions to create an instance of a COM object and retrieve an interface pointer without using the COM API.

```
template <class Q>
static HRESULT CreateInstance(
   Q** pp
);
template <class Q>
static HRESULT CreateInstance(
   IUnknown* punkOuter,
   Q** pp
);
```

Parameters

Q

The COM interface that should be returned via *pp*.

punkOuter

[in] The outer unknown or controlling unknown of the aggregate.

pp

[out] The address of a pointer variable that receives the requested interface pointer if creation succeeds.

Return Value

A standard **HRESULT** value. See CoCreateInstance in the *Platform SDK* for a description of possible return values.

Remarks

Use the first overload of this function for typical object creation; use the second overload when you need to aggregate the object being created.

The ATL class implementing the required COM object (that is, the class used as the first template parameter to CComCoClass) must be in the same project as the calling code. The creation of the COM object is carried out by the class factory registered for this ATL class.

These functions are useful for creating objects that you have prevented from being externally creatable by using the OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO macro. They are also useful in situations where you want to avoid the COM API for reasons of efficiency.

Note that the interface *Q* must have an IID associated with it that can be retrieved using the __uuidof operator.

Example

In the following example, `CDocument` is a wizard-generated ATL class derived from **CComCoClass** that implements the **IDocument** interface. The class is registered in the object map with the **OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO** macro so clients can't create instances of the document using CoCreateInstance. `CApplication` is a CoClass that provides a method on one of its own COM interfaces to create instances of the document class. The code below shows how easy it to create instances of the document class using the **CreateInstance** member inherited from the **CComCoClass** base class.

```
STDMETHODIMP CApplication::CreateDocument( /* [out, retval] */ IDocument** ppDoc)
{
   *ppDoc = NULL;
   return CDocument::CreateInstance(ppDoc);
}
```

See Also

Reference

CComCoClass Class

Other Resources

CComCoClass::Error

This static function sets up the IErrorInfo interface to provide error information to the client.

```
static HRESULT WINAPI Error(
   LPCOLESTR lpszDesc,
   const IID& iid = GUID_NULL,
   HRESULT hRes = 0
);
static HRESULT WINAPI Error(
   LPCOLESTR lpszDesc,
   DWORD dwHelpID,
   LPCOLESTR lpszHelpFile,
   const IID& iid = GUID_NULL,
   HRESULT hRes = 0
);
static HRESULT WINAPI Error(
   LPCSTR lpszDesc,
   const IID& iid = GUID_NULL,
   HRESULT hRes = 0
);
static HRESULT WINAPI Error(
   LPCSTR lpszDesc,
   DWORD dwHelpID,
   LPCSTR lpszHelpFile,
   const IID& iid = GUID_NULL,
   HRESULT hRes = 0
);
static HRESULT WINAPI Error(
   UINT nID,
   const IID& iid = GUID_NULL,
   HRESULT hRes = 0,
   HINSTANCE hInst = _AtlBaseModule.GetResourceInstance ()
);
static HRESULT Error(
   UINT nID,
   DWORD dwHelpID,
   LPCOLESTR lpszHelpFile,
   const IID& iid = GUID_NULL,
   HRESULT hRes = 0,
   HINSTANCE hInst = _AtlBaseModule.GetResourceInstance()
);
```

Parameters

lpszDesc

[in] The string describing the error. The Unicode version of **Error** specifies that *lpszDesc* is of type **LPCOLESTR**; the ANSI version specifies a type of **LPCSTR**.

iid

[in] The IID of the interface defining the error or **GUID_NULL** (the default value) if the error is defined by the operating system.

hRes

[in] The **HRESULT** you want returned to the caller. The default value is 0. For more details about *hRes*, see Remarks.

nID

[in] The resource identifier where the error description string is stored. This value should lie between 0x0200 and 0xFFFF, inclusively. In debug builds, an **ASSERT** will result if *nID* does not index a valid string. In release builds, the error description string will be set to "Unknown Error."

dwHelpID

[in] The help context identifier for the error.

lpszHelpFile

[in] The path and name of the help file describing the error.

hInst

[in] The handle to the resource. By default, this parameter is **_AtlModule::GetResourceInstance**, where **_AtlModule** is the global instance of CAtlModule.

Return Value

A standard **HRESULT** value. For details, see Remarks.

Remarks

To call **Error**, your object must implement the ISupportErrorInfo Interface interface.

If the *hRes* parameter is nonzero, then **Error** returns the value of *hRes*. If *hRes* is zero, then the first four versions of **Error** return **DISP_E_EXCEPTION**. The last two versions return the result of the macro **MAKE_HRESULT(1, FACILITY_ITF,** *nID* **)**.

See Also

Reference

CComCoClass Class
ISupportErrorInfoImpl Class
MAKE_HRESULT

Other Resources

CComCoClass Members

CComCoClass::GetObjectCLSID

Provides a consistent way of retrieving the object's CLSID.

```
static const CLSID& WINAPI GetObjectCLSID( );
```

Return Value

The object's class identifier.

See Also

Reference

CComCoClass Class

Other Resources

CComCoClass Members

CComCoClass::GetObjectDescription

This static function retrieves the text description for your class object.

```
static LPCTSTR WINAPI GetObjectDescription( );
```

Return Value

The class object's description.

Remarks

The default implementation returns **NULL**. You can override this method with the DECLARE_OBJECT_DESCRIPTION macro. For example:

```
class CMyClass : public CComCoClass< ... >, ...
{
public:
   DECLARE_OBJECT_DESCRIPTION("Account Transfer Object 1.0")

   ...
};
```

GetObjectDescription is called by **IComponentRegistrar::GetComponents**. **IComponentRegistrar** is an Automation interface that allows you to register and unregister individual components in a DLL. When you create a Component Registrar object with the ATL Project Wizard, the wizard will automatically implement the **IComponentRegistrar** interface. **IComponentRegistrar** is typically used by Microsoft Transaction Server.

For more information about the ATL Project Wizard, see the article Creating an ATL Project.

See Also

Reference

CComCoClass Class

Other Resources

CComCoClass Members

CComCompositeControl Class

This class provides the methods required to implement a composite control.

```
template <
   class T
>
class CComCompositeControl :
   public CComControl< T, CAxDialogImpl< T > >
```

Parameters

T

Your class, derived from CComObjectRoot or CComObjectRootEx, as well as from any other interfaces you want to support for your composite control.

Remarks

Classes derived from class **CComCompositeControl** inherit the functionality of an ActiveX composite control. ActiveX controls derived from **CComCompositeControl** are hosted by a standard dialog box. These types of controls are called composite controls because they are able to host other controls (native Windows controls and ActiveX controls).

CComCompositeControl identifies the dialog resource to use in creating the composite control by looking for an enumerated data member in the child class. The member IDD of this child class is set to the resource ID of the dialog resource that will be used as the control's window. The following is an example of the data member that the class derived from **CComCompositeControl** should contain to identify the dialog resource to be used for the control's window:

```
enum { IDD = IDD_MYCOMPOSITE };
```

Note

Composite controls are always windowed controls, although they can contain windowless controls.

A control implemented by a **CComCompositeControl**-derived class has default tabbing behavior built in. When the control receives focus by being tabbed to in a containing application, successively pressing the TAB key will cause the focus to be cycled through all of the composite control's contained controls, then out of the composite control and on to the next item in the tab order of the container. The tab order of the hosted controls is determined by the dialog resource and determines the order in which tabbing will occur.

Note

In order for accelerators to work properly with a **CComCompositeControl**, it is necessary to load an accelerator table as the control is created, pass the handle and number of accelerators back into IOleControlImpl::GetControlInfo, and finally destroy the table when the control is released.

Example

```
// Example for overriding IOleControlImpl::GetControlInfo()
// This example uses the accelerator table from the project resources
// with the identifier IDR_ACCELTABLE
// Define GetControlInfo() in the header of your composite
// control class as follows:

STDMETHOD(GetControlInfo)(CONTROLINFO* pCI)
{
    // Load the accelerator table from the resource
    pCI->hAccel = LoadAccelerators(_Module.GetModuleInstance(), MAKEINTRESOURCE(IDR_ACCELTA
BLE));
    if (pCI->hAccel == NULL)
        return E_FAIL;
    // Get the number of accelerators in the table
```

```
    pCI->cAccel = CopyAcceleratorTable(pCI->hAccel, NULL, 0);
    // The following is optional if you want your control
    // to process the return and/or escape keys
    // pCI.dwFlags = CTRLINFO_EATS_RETURN | CTRLINFO_EATS_ESCAPE;
    pCI->dwFlags = 0;
    return S_OK;
}
```

Requirements

Header: atlctl.h

See Also

Reference

CComControl Class

Concepts

Composite Control Fundamentals

Other Resources

CComCompositeControl Members

ATL Class Overview

CComCompositeControl Members

Methods

AdviseSinkMap	Call this method to advise or unadvise all controls hosted by the composite control.
CalcExtent	Call this method to calculate the size in **HIMETRIC** units of the dialog resource used to host the composite control.
CComCompositeControl	The constructor.
~CComCompositeControl	The destructor.
Create	This method is called to create the control window for the composite control.
CreateControlWindow	Call this method to create the control window and advise any hosted control.
SetBackgroundColorFromAmbient	Call this method to set the background color of the composite control using the container's background color.

Data Members

m_hbrBackground	The background brush.
m_hWndFocus	The handle of the window that currently has focus.

See Also

Reference

CComCompositeControl Class

Other Resources

CComControl Members

CComCompositeControl Methods

For information about the methods in **CComCompositeControl**, see CComCompositeControl Members.

CComCompositeControl::AdviseSinkMap

Call this method to advise or unadvise all controls hosted by the composite control.

```
HRESULT AdviseSinkMap(
   bool bAdvise
);
```

Parameters

bAdvise

True if all controls are to be advised; otherwise false.

Return Value

S_OK

All controls in the event sink map were connected or disconnected from their event source successfully.

E_FAIL

Not all controls in the event sink map could be connected or disconnected from their event source successfully.

E_POINTER

This error usually indicates a problem with an entry in the control's event sink map or a problem with a template argument used in an **IDispEventImpl** or **IDispEventSimpleImpl** base class.

CONNECT_E_ADVISELIMIT

The connection point has already reached its limit of connections and cannot accept any more.

CONNECT_E_CANNOTCONNECT

The sink does not support the interface required by this connection point.

CONNECT_E_NOCONNECTION

The cookie value does not represent a valid connection. This error usually indicates a problem with an entry in the control's event sink map or a problem with a template argument used in an **IDispEventImpl** or **IDispEventSimpleImpl** base class.

Remarks

The base implementation of this method searches through the entries in the event sink map. It then advises or unadvises the connection points to the COM objects described by the event sink map's sink entries. This member method also relies on the fact that the derived class inherits from one instance of **IDispEventImpl** for every control in the sink map that is to be advised or unadvised.

See Also

Reference

CComCompositeControl Class
IDispEventImpl Class
BEGIN_SINK_MAP
CComCompositeControl::CreateControlWindow

Other Resources

CComCompositeControl Members

CComCompositeControl::CalcExtent

Call this method to calculate the size in **HIMETRIC** units of the dialog resource used to host the composite control.

```
BOOL CalcExtent(
   SIZE& size
);
```

Parameters

size

A reference to a **SIZE** structure to be filled by this method.

Return Value

TRUE if the control is hosted by a dialog box; otherwise FALSE.

Remarks

The size is returned in the *size* parameter.

See Also

Reference

CComCompositeControl Class

AtlPixelToHiMetric

Other Resources

CComCompositeControl Members

CComCompositeControl::Create

This method is called to create the control window for the composite control.

```
HWND Create(
   HWND hWndParent,
   RECT& /* rcPos */,
   LPARAM dwInitParam = NULL
);
```

Parameters

hWndParent

A handle to the parent window of the control.

rcPos

Reserved.

dwInitParam

Data to be passed to the control during control creation. The data passed as *dwInitParam* will show up as the **LPARAM** parameter of the WM_INITDIALOG message, which will be sent to the composite control when it gets created.

Return Value

A handle to the newly created composite control dialog box.

Remarks

This method is usually called during in-place activation of the control.

See Also

Reference

CComCompositeControl Class
AtlAxCreateDialog
CComCompositeControl::CreateControlWindow

Other Resources

CComCompositeControl Members

CComCompositeControl::~CComCompositeControl

The destructor.

```
~CComCompositeControl( );
```

Remarks

Deletes the background object, if it exists.

See Also

Reference

CComCompositeControl Class

CComCompositeControl::CComCompositeControl

Other Resources

CComCompositeControl Members

CComCompositeControl::CComCompositeControl

The constructor.

```
CComCompositeControl( );
```

Remarks

Initializes the CComCompositeControl::m_hbrBackground and CComCompositeControl::m_hWndFocus data members to NULL.

See Also

Reference

CComCompositeControl Class

CComCompositeControl::~CComCompositeControl

Other Resources

CComCompositeControl Members

CComCompositeControl::CreateControlWindow

Call this method to create the control window and advise any hosted controls.

```
virtual HWND CreateControlWindow(
   HWND hWndParent,
   RECT& rcPos
);
```

Parameters

hWndParent

A handle to the parent window of the control.

rcPos

The position rectangle of the composite control in client coordinates relative to *hWndParent*.

Return Value

Returns a handle to the newly created composite control dialog box.

Remarks

This method calls CComCompositeControl::Create and CComCompositeControl::AdviseSinkMap.

See Also

Reference

CComCompositeControl Class

Other Resources

CComCompositeControl Members

CComCompositeControl::SetBackgroundColorFromAmbient

Call this method to set the background color of the composite control using the container's background color.

```
HRESULT SetBackgroundColorFromAmbient( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CComCompositeControl Class

Other Resources

CComCompositeControl Members

CComCompositeControl Data Members

For information about the data members in **CComCompositeControl**, see CComCompositeControl Members.

CComCompositeControl::m_hbrBackground

The background brush.

```
HBRUSH m_hbrBackground;
```

See Also

Reference

CComCompositeControl Class

Other Resources

CComCompositeControl Members

CComCompositeControl::m_hWndFocus

The handle of the window that currently has focus.

```
HWND m_hWndFocus;
```

See Also

Reference

CComCompositeControl Class

Other Resources

CComCompositeControl Members

CComContainedObject Class

This class implements IUnknown by delegating to the owner object's **IUnknown**.

```
template<
   class Base
>
class CComContainedObject :
   public Base
```

Parameters

Base

Your class, derived from CComObjectRoot or CComObjectRootEx.

Remarks

ATL uses **CComContainedObject** in classes CComAggObject, CComPolyObject, and CComCachedTearOffObject. **CComContainedObject** implements IUnknown by delegating to the owner object's **IUnknown**. (The owner is either the outer object of an aggregation, or the object for which a tear-off interface is being created.) **CComContainedObject** calls **CComObjectRootEx**'s **OuterQueryInterface**, **OuterAddRef**, and **OuterRelease**, all inherited through *Base*.

Requirements

Header: atlcom.h

See Also

Other Resources

CComContainedObject Members

ATL Class Overview

CComContainedObject Members

Class Methods

CComContainedObject	The constructor. Initializes the member pointer to the owner object's **IUnknown**.
~CComContainedObject	The destructor.
GetControllingUnknown	Retrieves the owner object's **Iunknown**.

IUnknown Methods

AddRef	Increments the reference count on the owner object.
QueryInterface	Retrieves a pointer to the interface requested on the owner object.
Release	Decrements the reference count on the owner object.

See Also

Reference

CComContainedObject Class

CComContainedObject Methods

For information about the methods in **CComContainedObject**, see CComContainedObject Members.

CComContainedObject::AddRef

Increments the reference count on the owner object.

```
STDMETHOD_(ULONG, AddRef)( );
```

Return Value

A value that may be useful for diagnostics or testing.

See Also

Reference

CComContainedObject Class

CComContainedObject::Release

Other Resources

CComContainedObject Members

CComContainedObject::CComContainedObject

The constructor.

```
CComContainedObject(
   void* pv
);
```

Parameters

pv

[in] The owner object's **IUnknown**.

Remarks

Sets the **m_pOuterUnknown** member pointer (inherited through the *Base* class) to *pv*.

See Also

Reference

CComContainedObject Class

CComObjectRootEx::m_pOuterUnknown

Other Resources

CComContainedObject Members

CComContainedObject::~CComContainedObject

The destructor.

```
~CComContainedObject( );
```

Remarks

Frees all allocated resources.

See Also

Reference

CComContainedObject Class

Other Resources

CComContainedObject Members

CComContainedObject::GetControllingUnknown

Returns the **m_pOuterUnknown** member pointer (inherited through the *Base* class) that holds the owner object's **IUnknown**.

```
IUnknown* GetControllingUnknown( );
```

Return Value

The owner object's **IUnknown**.

Remarks

This method may be virtual if *Base* has declared the DECLARE_GET_CONTROLLING_UNKNOWN macro.

See Also

Reference

CComContainedObject Class

CComObjectRootEx::m_pOuterUnknown

Other Resources

CComContainedObject Members

CComContainedObject::QueryInterface

Retrieves a pointer to the interface requested on the owner object.

```
STDMETHOD(QueryInterface)(
   REFIID iid,
   void** ppvObject
);
template <class Q>
HRESULT STDMETHODCALLTYPE QueryInterface(
   Q** pp
);
```

Parameters

iid

[in] The identifier of the interface being requested.

ppvObject

[out] A pointer to the interface pointer identified by *iid*. If the object does not support this interface, *ppvObject* is set to **NULL**.

pp

[out] A pointer to the interface pointer identified by type *Q*. If the object does not support this interface, *pp* is set to **NULL**.

Return Value

A standard **HRESULT** value.

See Also

Reference

CComContainedObject Class

Other Resources

CComContainedObject Members

CComContainedObject::Release

Decrements the reference count on the owner object.

```
STDMETHOD_(ULONG, Release)( );
```

Return Value

In debug builds, **Release** returns a value that may be useful for diagnostics or testing. In non-debug builds, **Release** always returns 0.

See Also

Reference

CComContainedObject Class

CComContainedObject::AddRef

Other Resources

CComContainedObject Members

CComControl Class

This class provides methods for creating and managing ATL controls.

```
template <
   class T,
   class WinBase = CWindowImpl< T >
>
class ATL_NO_VTABLE CComControl :
   public CComControlBase, public WinBase;
```

Parameters

T

The class implementing the control.

WinBase

The base class that implements windowing functions. Defaults to CWindowImpl.

Remarks

CComControl is a set of useful control helper functions and essential data members for ATL controls. When you create a standard control or a DHTML control using the ATL Control Wizard, the wizard will automatically derive your class from **CComControl**. **CComControl** derives most of its methods from CComControlBase.

For more information about creating a control, see the ATL Tutorial. For more information about the ATL Project Wizard, see the article Creating an ATL Project.

For a demonstration of **CComControl** methods and data members, see the CIRC sample.

Requirements

Header: atlctl.h

See Also

Reference

CWindowImpl Class
CComControlBase Class
CComCompositeControl Class

Other Resources

CComControl Members
ATL Class Overview

CComControl Members

Methods

CComControl	Constructor.
ControlQueryInterface	Retrieves a pointer to the requested interface.
CreateControlWindow	Creates a window for the control.
FireOnChanged	Notifies the container's sink that a control property has changed.
FireOnRequestEdit	Notifies the container's sink that a control property is about to change and that the object is asking the sink how to proceed.
MessageBox	Call this method to create, display, and operate a message box.

See Also

Reference

CComControl Class

Other Resources

CComControlBase Members

CComControl Methods

For information about the methods in **CComControl**, see CComControl Members.

CComControl::CComControl

The constructor.

```
CComControl( );
```

Remarks

Calls the CComControlBase constructor, passing the **m_hWnd** data member inherited through CWindowImpl.

See Also

Reference

CComControl Class

CWindow::m_hWnd

Other Resources

CComControl Members

CComControl::ControlQueryInterface

Retrieves a pointer to the requested interface.

```
virtual HRESULT ControlQueryInterface(
   const IID& iid,
      void** ppv
);
```

Parameters

iid

[in] The GUID of the interface being requested.

ppv

[out] A pointer to the interface pointer identified by *iid*, or **NULL** if the interface is not found.

Remarks

Only handles interfaces in the COM map table.

Example

```
// Retrieve the control's IOleObject interface. Note interface
// is automatically released when pOleObject goes out of scope

CComPtr<IOleObject> pOleObject;
ControlQueryInterface(IID_IOleObject, (void**)&pOleObject);
```

See Also

Reference

CComControl Class

CComObjectRootEx::InternalQueryInterface

Other Resources

CComControl Members

CComControl::CreateControlWindow

By default, creates a window for the control by calling **CWindowImpl::Create**.

```
virtual HWND CreateControlWindow(
   HWND hWndParent,
      RECT& rcPos
);
```

Parameters

hWndParent

[in] Handle to the parent or owner window. A valid window handle must be supplied. The control window is confined to the area of its parent window.

rcPos

[in] The initial size and position of the window to be created.

Remarks

Override this method if you want to do something other than create a single window, for example, to create two windows, one of which becomes a toolbar for your control.

Example

```
RECT rc = {10,10,210,110};
HWND hwndParent, hwndControl;

// get HWND of control's parent window from IOleInPlaceSite interface
m_spInPlaceSite->GetWindow(&hwndParent);
hwndControl = CreateControlWindow(hwndParent, rc);
```

See Also

Reference

CComControl Class

CWindowImpl::Create

Other Resources

CComControl Members

CComControl::FireOnChanged

Notifies the container's sink that a control property has changed.

```
HRESULT FireOnChanged(
   DISPID dispID
);
```

Parameters

dispID

[in] Identifier of the property that has changed.

Return Value

One of the standard HRESULT values.

Remarks

If your control class derives from IPropertyNotifySink, this method calls CFirePropNotifyEvent::FireOnChanged to notify all connected **IPropertyNotifySink** interfaces that the specified control property has changed. If your control class does not derive from **IPropertyNotifySink**, this method returns **S_OK**.

This method is safe to call even if your control doesn't support connection points.

Example

```
STDMETHODIMP CMyCtrl::put_MyText(BSTR newVal)
{
   // store newVal in CComBstr member
   m_bstrMyText = newVal;

   // note the DISPID for the MyText property is 1 in this example
   FireOnChanged(1);

   return S_OK;
}
```

See Also

Reference

CComControl Class

CComControl::FireOnRequestEdit

Other Resources

CComControl Members

CComControl::FireOnRequestEdit

Notifies the container's sink that a control property is about to change and that the object is asking the sink how to proceed.

```
HRESULT FireOnRequestEdit(
   DISPID dispID
);
```

Parameters

dispID

[in] Identifier of the property about to change.

Return Value

One of the standard HRESULT values.

Remarks

If your control class derives from IPropertyNotifySink, this method calls CFirePropNotifyEvent::FireOnRequestEdit to notify all connected **IPropertyNotifySink** interfaces that the specified control property is about to change. If your control class does not derive from **IPropertyNotifySink**, this method returns **S_OK**.

This method is safe to call even if your control doesn't support connection points.

Example

```
STDMETHODIMP CMyCtrl::put_MyText(BSTR newVal)
{
   // the DISPID for MyText in this example is 1
   DISPID dispID = 1;

   // make sure we can change the property
   if (FireOnRequestEdit(dispID) == S_FALSE)
      return S_FALSE;

   // store newVal in CComBstr member
   m_bstrMyText = newVal;

   // signal that the property has been changed
   FireOnChanged(dispID);

   return S_OK;
}
```

See Also

Reference

CComControl Class

CComControl::FireOnChanged

Other Resources

CComControl Members

CComControl::MessageBox

Call this method to create, display, and operate a message box.

```
int MessageBox(
   LPCTSTR lpszText,
   LPCTSTR lpszCaption = _T(""),
   UINT nType = MB_OK
);
```

Parameters

lpszText

The text to be displayed in the message box.

lpszCaption

The dialog box title. If NULL (the default), the title "Error" is used.

nType

Specifies the contents and behavior of the dialog box. See the MessageBox entry in the *Platform SDK* documentation for a list of the different message boxes available. The default provides a simple **OK** button.

Return Value

Returns an integer value specifying one of the menu-item values listed under MessageBox in the *Platform SDK* documentation.

Remarks

MessageBox is useful both during development and as an easy way to display an error or warning message to the user.

See Also

Reference

CComControl Class

Other Resources

CComControl Members

CComControlBase Class

This class provides methods for creating and managing ATL controls.

```
class ATL_NO_VTABLE CComControlBase
```

Remarks

This class provides methods for creating and managing ATL controls. CComControl Class derives from **CComControlBase**. When you create a Standard Control or DHTML control using the ATL Control Wizard, the wizard will automatically derive your class from **CComControlBase**.

For more information about creating a control, see the ATL Tutorial. For more information about the ATL Project Wizard, see the article Creating an ATL Project.

Requirements

Header: atlctl.h

See Also

Reference

CComControl Class

Other Resources

CComControlBase Members

ATL Class Overview

CComControlBase Members

CComControlBase class members are divided into the following categories:

- Methods
- GetAmbient Property Methods
- Data Members
- Typedefs

Methods

CComControlBase	The constructor.
~CComControlBase	The destructor.
ControlQueryInterface	Retrieves a pointer to the requested interface.
DoesVerbActivate	Checks that the *iVerb* parameter used by **IOleObjectImpl::DoVerb** either activates the control's user interface (*iVerb* equals **OLEIVERB_UIACTIVATE**), defines the action taken when the user double-clicks the control (*iVerb* equals **OLEIVERB_PRIMARY**), displays the control (*iVerb* equals **OLEIVERB_SHOW**), or activates the control (*iVerb* equals **OLEIVERB_INPLACEACTIVATE**).
DoesVerbUIActivate	Checks that the *iVerb* parameter used by **IOleObjectImpl::DoVerb** causes the control's user interface to activate and returns **TRUE**.
DoVerbProperties	Displays the control's property pages.
FireViewChange	Call this method to tell the container to redraw the control, or notify the registered advise sinks that the control's view has changed.
GetDirty	Returns the value of data member **m_bRequiresSave**.
GetZoomInfo	Retrieves the x and y values of the numerator and denominator of the zoom factor for a control activated for in-place editing.
InPlaceActivate	Causes the control to transition from the inactive state to whatever state the verb in *iVerb* indicates.
InternalGetSite	Call this method to query the control site for a pointer to the identified interface.
OnDraw	Override this method to draw your control.
OnDrawAdvanced	The default **OnDrawAdvanced** prepares a normalized device context for drawing, then calls your control class's **OnDraw** method.
OnKillFocus	Checks that the control is in-place active and has a valid control site, then informs the container that the control has lost focus.
OnMouseActivate	Checks that the UI is in user mode, then activates the control.
OnPaint	Prepares the container for painting, gets the control's client area, then calls the control class's **OnDraw** method.

OnSetFocus	Checks that the control is in-place active and has a valid control site, then informs the container the control has gained focus.
PreTranslateAccelerator	Override this method to provide your own keyboard accelerator handlers.
SendOnClose	Notifies all advisory sinks registered with the advise holder that the control has been closed.
SendOnDataChange	Notifies all advisory sinks registered with the advise holder that the control data has changed.
SendOnRename	Notifies all advisory sinks registered with the advise holder that the control has a new moniker.
SendOnSave	Notifies all advisory sinks registered with the advise holder that the control has been saved.
SendOnViewChange	Notifies all registered advisory sinks that the control's view has changed.
SetControlFocus	Sets or removes the keyboard focus to or from the control.
SetDirty	Sets the data member **m_bRequiresSave** to the value in *bDirty*.

GetAmbient Property Methods

GetAmbientAppearance	Retrieves **DISPID_AMBIENT_APPEARANCE**, the current appearance setting for the control: 0 for flat and 1 for 3D.
GetAmbientAutoClip	Retrieves **DISPID_AMBIENT_AUTOCLIP**, a flag indicating whether the container supports automatic clipping of the control display area.
GetAmbientBackColor	Retrieves **DISPID_AMBIENT_BACKCOLOR**, the ambient background color for all controls, defined by the container.
GetAmbientCharSet	Retrieves **DISPID_AMBIENT_CHARSET**, the ambient character set for all controls, defined by the container.
GetAmbientCodePage	Retrieves **DISPID_AMBIENT_CODEPAGE**, the ambient character set for all controls, defined by the container.
GetAmbientDisplayAsDefault	Retrieves **DISPID_AMBIENT_DISPLAYASDEFAULT**, a flag that is **TRUE** if the container has marked the control in this site to be a default button, and therefore a button control should draw itself with a thicker frame.
GetAmbientDisplayName	Retrieves **DISPID_AMBIENT_DISPLAYNAME**, the name the container has supplied to the control.
GetAmbientFont	Retrieves a pointer to the container's ambient **IFont** interface.
GetAmbientFontDisp	Retrieves a pointer to the container's ambient **IFontDisp** dispatch interface.
GetAmbientForeColor	Retrieves **DISPID_AMBIENT_FORECOLOR**, the ambient foreground color for all controls, defined by the container.
GetAmbientLocaleID	Retrieves **DISPID_AMBIENT_LOCALEID**, the identifier of the language used by the container.
GetAmbientMessageReflect	Retrieves **DISPID_AMBIENT_MESSAGEREFLECT**, a flag indicating whether the container wants to receive window messages (such as **WM_DRAWITEM**) as events.

GetAmbientPalette	Retrieves **DISPID_AMBIENT_PALETTE**, used to access the container's **HPALETTE**.
GetAmbientProperty	Retrieves the container property specified by *id*.
GetAmbientRightToLeft	Retrieves **DISPID_AMBIENT_RIGHTTOLEFT**, the direction in which content is displayed by the container.
GetAmbientScaleUnits	Retrieves **DISPID_AMBIENT_SCALEUNITS**, the container's ambient units (such as inches or centimeters) for labeling displays.
GetAmbientShowGrabHandles	Retrieves **DISPID_AMBIENT_SHOWGRABHANDLES**, a flag indicating whether the container allows the control to display grab handles for itself when active.
GetAmbientShowHatching	Retrieves **DISPID_AMBIENT_SHOWHATCHING**, a flag indicating whether the container allows the control to display itself with a hatched pattern when the UI is active.
GetAmbientSupportsMnemonics	Retrieves **DISPID_AMBIENT_SUPPORTSMNEMONICS**, a flag indicating whether the container supports keyboard mnemonics.
GetAmbientTextAlign	Retrieves **DISPID_AMBIENT_TEXTALIGN**, the text alignment preferred by the container: 0 for general alignment (numbers right, text left), 1 for left alignment, 2 for center alignment, and 3 for right alignment.
GetAmbientTopToBottom	Retrieves **DISPID_AMBIENT_TOPTOBOTTOM**, the direction in which content is displayed by the container.
GetAmbientUIDead	Retrieves **DISPID_AMBIENT_UIDEAD**, a flag indicating whether the container wants the control to respond to user-interface actions.
GetAmbientUserMode	Retrieves **DISPID_AMBIENT_USERMODE**, a flag indicating whether the container is in run-mode (**TRUE**) or design-mode (**FALSE**).

Data Members

m_bAutoSize	Flag indicating the control cannot be any other size.
m_bDrawFromNatural	Flag indicating that **IDataObjectImpl::GetData** and **CComControlBase::GetZoomInfo** should set the control size from **m_sizeNatural** rather than from **m_sizeExtent**.
m_bDrawGetDataInHimetric	Flag indicating that **IDataObjectImpl::GetData** should use HIMETRIC units and not pixels when drawing.
m_bInPlaceActive	Flag indicating the control is in-place active.
m_bInPlaceSiteEx	Flag indicating the container supports the **IOleInPlaceSiteEx** interface and OCX96 control features, such as windowless and flicker-free controls.
m_bNegotiatedWnd	Flag indicating whether or not the control has negotiated with the container about support for OCX96 control features (such as flicker-free and windowless controls), and whether the control is windowed or windowless.
m_bRecomposeOnResize	Flag indicating the control wants to recompose its presentation when the container changes the control's display size.
m_bRequiresSave	Flag indicating the control has changed since it was last saved.

m_bResizeNatural	Flag indicating the control wants to resize its natural extent (its unscaled physical size) when the c ontainer changes the control's display size.
m_bUIActive	Flag indicating the control's user interface, such as menus and toolbars, is active.
m_bUsingWindowRgn	Flag indicating the control is using the container-supplied window region.
m_bWasOnceWindowless	Flag indicating the control has been windowless, but may or may not be windowless now.
m_bWindowOnly	Flag indicating the control should be windowed, even if the container supports windowless contr ols.
m_bWndLess	Flag indicating the control is windowless.
m_hWndCD	Contains a reference to the window handle associated with the control.
m_nFreezeEvents	A count of the number of times the container has frozen events (refused to accept events) without an intervening thaw of events (acceptance of events).
m_rcPos	The position in pixels of the control, expressed in the coordinates of the container.
m_sizeExtent	The extent of the control in HIMETRIC units (each unit is 0.01 millimeters) for a particular display.
m_sizeNatural	The physical size of the control in HIMETRIC units (each unit is 0.01 millimeters).
m_spAdviseSink	A direct pointer to the advisory connection on the container (the container's IAdviseSink).
m_spAmbientDispatch	A **CComDispatchDriver** object that lets you retrieve and set the container's properties through a n **IDispatch** pointer. For more information, see Implementing the IDispatch Interface.
m_spClientSite	A pointer to the control's client site within the container.
m_spDataAdviseHolder	Provides a standard means to hold advisory connections between data objects and advise sinks.
m_spInPlaceSite	A pointer to the container's IOleInPlaceSite, IOleInPlaceSiteEx, or IOleInPlaceSiteWindowless interf ace pointer.
m_spOleAdviseHolder	Provides a standard implementation of a way to hold advisory connections.

Typedefs

AppearanceType	Override if your **m_nAppearance** stock property isn't of type **short**.

See Also

Reference

CComControlBase Class

CComControlBase Methods

For information about the methods in **CComControlBase**, see CComControlBase Members.

CComControlBase::CComControlBase

The constructor.

```
CComControlBase(
   HWND& h
);
```

Parameters

h

The handle to the window associated with the control.

Remarks

Initializes the control size to 5080X5080 HIMETRIC units (2"X2") and initializes the **CComControlBase** data member values to **NULL** or **FALSE**.

See Also

Reference

CComControlBase Class

CComControlBase::~CComControlBase

Other Resources

CComControlBase Members

CComControlBase::~CComControlBase

The destructor.

```
~CComControlBase( );
```

Remarks

If the control is windowed, **~CComControlBase** destroys it by calling DestroyWindow.

See Also

Reference

CComControlBase Class

CComControlBase::CComControlBase

Other Resources

CComControlBase Members

CComControlBase::ControlQueryInterface

Retrieves a pointer to the requested interface.

```
virtual HRESULT ControlQueryInterface(
   const IID& iid,
      void** ppv
);
```

Parameters

iid

The GUID of the interface being requested.

ppv

A pointer to the interface pointer identified by *iid*, or **NULL** if the interface is not found.

Remarks

Only handles interfaces in the COM map table.

Example

```
// Retrieve the control's IOleObject interface. Note interface
// is automatically released when pOleObject goes out of scope

CComPtr<IOleObject> pOleObject;
ControlQueryInterface(IID_IOleObject, (void**)&pOleObject);
```

See Also

Reference

CComControlBase Class

CComObjectRootEx::InternalQueryInterface

Other Resources

CComControlBase Members

CComControlBase::DoesVerbActivate

Checks that the *iVerb* parameter used by **IOleObjectImpl::DoVerb** either activates the control's user interface (*iVerb* equals **OLEIVERB_UIACTIVATE**), defines the action taken when the user double-clicks the control (*iVerb* equals **OLEIVERB_PRIMARY**), displays the control (*iVerb* equals **OLEIVERB_SHOW**), or activates the control (*iVerb* equals **OLEIVERB_INPLACEACTIVATE**).

```
BOOL DoesVerbActivate(
   LONG iVerb
);
```

Parameters

iVerb

Value indicating the action to be performed by **DoVerb**.

Return Value

Returns **TRUE** if *iVerb* equals **OLEIVERB_UIACTIVATE**, **OLEIVERB_PRIMARY**, **OLEIVERB_SHOW**, or **OLEIVERB_INPLACEACTIVATE**; otherwise, returns **FALSE**.

Remarks

You can override this method to define your own activation verb.

See Also

Reference

CComControlBase Class
IOleObjectImpl::DoVerb
CComControlBase::DoesVerbUIActivate

Other Resources

CComControlBase Members

CComControlBase::DoesVerbUIActivate

Checks that the *iVerb* parameter used by **IOleObjectImpl::DoVerb** causes the control's user interface to activate and returns **TRUE**.

```
BOOL DoesVerbUIActivate(
   LONG iVerb
);
```

Parameters

iVerb

Value indicating the action to be performed by **DoVerb**.

Return Value

Returns **TRUE** if *iVerb* equals **OLEIVERB_UIACTIVATE**, **OLEIVERB_PRIMARY**, **OLEIVERB_SHOW**, or **OLEIVERB_INPLACEACTIVATE**. Otherwise, the method returns **FALSE**.

See Also

Reference

CComControlBase Class
IOleObjectImpl::DoVerb
CComControlBase::DoesVerbActivate

Other Resources

CComControlBase Members

CComControlBase::DoVerbProperties

Displays the control's property pages.

```
HRESULT DoVerbProperties(
   LPCRECT /* prcPosRect */,
   HWND hwndParent
);
```

Parameters

prcPosRec

Reserved.

hwndParent

Handle of the window containing the control.

Return Value

One of the standard HRESULT values.

Example

```
// The following implementation of the WM_RBUTTONDOWN message handler
// will pop up the ActiveX Control's PropertyPages

LRESULT CMyControl::OnRBtnDown(UINT nMsg, WPARAM wParam, LPARAM lParam,
   BOOL& bHandled)
{
   DoVerbProperties(NULL, ::GetActiveWindow());
   return 0L;
}

// The control's message map has the corresponding entry:
BEGIN_MSG_MAP(CMyControl)
   ...
   ...
   MESSAGE_HANDLER(WM_RBUTTONDOWN, OnRBtnDown)
END_MSG_MAP()
```

See Also

Reference

CComControlBase Class

IOleObjectImpl::DoVerbPrimary

Other Resources

CComControlBase Members

CComControlBase::FireViewChange

Call this method to tell the container to redraw the control, or notify the registered advise sinks that the control's view has changed.

```
HRESULT FireViewChange( );
```

Return Value

One of the standard HRESULT values.

Remarks

If the control is active (the control class data member CComControlBase::m_bInPlaceActive is **TRUE**), notifies the container that you want to redraw the entire control. If the control is inactive, notifies the control's registered advise sinks (through the control class data member CComControlBase::m_spAdviseSink) that the control's view has changed.

Example

```
STDMETHODIMP CSomeCtrl::put_Shape(short newVal)
{
   // store newVal in m_nShape user-defined member
   m_nShape = newVal;

   // notify container to redraw control
   FireViewChange();
   return S_OK;
}
```

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase::GetDirty

Returns the value of data member **m_bRequiresSave**.

```
BOOL GetDirty( );
```

Return Value

Returns the value of data member m_bRequiresSave.

Remarks

This value is set using CComControlBase::SetDirty.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase::GetZoomInfo

Retrieves the x and y values of the numerator and denominator of the zoom factor for a control activated for in-place editing.

```
void GetZoomInfo(
   ATL_DRAWINFO& di
);
```

Parameters

di

The structure that will hold the zoom factor's numerator and denominator. For more information, see ATL_DRAWINFO.

Remarks

The zoom factor is the proportion of the control's natural size to its current extent.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase::InPlaceActivate

Causes the control to transition from the inactive state to whatever state the verb in *iVerb* indicates.

```
HRESULT InPlaceActivate(
   LONG iVerb,
   const RECT* prcPosRect = NULL
);
```

Parameters

iVerb

Value indicating the action to be performed by IOleObjectImpl::DoVerb.

prcPosRect

Pointer to the position of the in-place control.

Return Value

One of the standard HRESULT values.

Remarks

Before activation, this method checks that the control has a client site, checks how much of the control is visible, and gets the control's location in the parent window. After the control is activated, this method activates the control's user interface and tells the container to make the control visible.

This method also retrieves an **IOleInPlaceSite**, **IOleInPlaceSiteEx**, or **IOleInPlaceSiteWindowless** interface pointer for the control and stores it in the control class's data member CComControlBase::m_spInPlaceSite. The control class data members CComControlBase::m_bInPlaceSiteEx, CComControlBase::m_bWndLess, CComControlBase::m_bWasOnceWindowless, and CComControlBase::m_bNegotiatedWnd are set to true as appropriate.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase::InternalGetSite

Call this method to query the control site for a pointer to the identified interface.

```
HRESULT InternalGetSite(
   REFIID riid,
   void** ppUnkSite
);
```

Parameters

riid

The IID of the interface pointer that should be returned in *ppUnkSite*.

ppUnkSite

Address of the pointer variable that receives the interface pointer requested in *riid*.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If the site supports the interface requested in *riid*, the pointer is returned by means of *ppUnkSite*. Otherwise, *ppUnkSite* is set to NULL.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase::OnDraw

Override this method to draw your control.

```
virtual HRESULT OnDraw(
   ATL_DRAWINFO& di
);
```

Parameters

di

A reference to the ATL_DRAWINFO structure that contains drawing information such as the draw aspect, the control bounds, and whether the drawing is optimized or not.

Return Value

A standard **HRESULT** value.

Remarks

The default **OnDraw** deletes or restores the device context or does nothing, depending on flags set in CComControlBase::OnDrawAdvanced.

An **OnDraw** method is automatically added to your control class when you create your control with the ATL Control Wizard. The wizard's default **OnDraw** draws a rectangle with the label "ATL 8.0".

Example

See the example for CComControlBase::GetAmbientAppearance.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase::OnDrawAdvanced

The default **OnDrawAdvanced** prepares a normalized device context for drawing, then calls your control class's **OnDraw** method.

```
virtual HRESULT OnDrawAdvanced(
   ATL_DRAWINFO& di
);
```

Parameters

di

A reference to the ATL_DRAWINFO structure that contains drawing information such as the draw aspect, the control bounds, and whether the drawing is optimized or not.

Return Value

A standard **HRESULT** value.

Remarks

Override this method if you want to accept the device context passed by the container without normalizing it.

See CComControlBase::OnDraw for more details.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase::OnKillFocus

Checks that the control is in-place active and has a valid control site, then informs the container that the control has lost focus.

```
LRESULT OnKillFocus(
   UINT /* nMsg */,
   WPARAM /* wParam */,
   LPARAM /* lParam */,
   BOOL& bHandled
);
```

Parameters

nMsg

Reserved.

wParam

Reserved.

lParam

Reserved.

bHandled

Flag that indicates whether the window message was successfully handled. The default is **TRUE**.

Return Value

Always returns 1.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase::OnMouseActivate

Checks that the UI is in user mode, then activates the control.

```
LRESULT OnMouseActivate(
   UINT /* nMsg */,
   WPARAM /* wParam */,
   LPARAM /* lParam */,
   BOOL& bHandled
);
```

Parameters

nMsg

Reserved.

wParam

Reserved.

lParam

Reserved.

bHandled

Flag that indicates whether the window message was successfully handled. The default is **TRUE**.

Return Value

Always returns 1.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase::OnPaint

Prepares the container for painting, gets the control's client area, then calls the control class's **OnDrawAdvanced** method.

```
LRESULT OnPaint(
   UINT /* nMsg */,
      WPARAM wParam,
      LPARAM /* lParam */,
      BOOL& /* lResult */
);
```

Parameters

nMsg

Reserved.

wParam

An existing HDC.

lParam

Reserved.

lResult

Reserved.

Return Value

Always returns zero.

Remarks

If *wParam* is not NULL, **OnPaint** assumes it contains a valid HDC and uses it instead of CComControlBase::m_hWndCD.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase::OnSetFocus

Checks that the control is in-place active and has a valid control site, then informs the container the control has gained focus.

```
LRESULT OnMouseActivate(
   UINT /* nMsg */,
   WPARAM /* wParam */,
   LPARAM /* lParam */,
   BOOL& bHandled
);
```

Parameters

nMsg

Reserved.

wParam

Reserved.

lParam

Reserved.

bHandled

Flag that indicates whether the window message was successfully handled. The default is **TRUE**.

Return Value

Always returns 1.

Remarks

Sends a notification to the container that the control has received focus.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase::PreTranslateAccelerator

Override this method to provide your own keyboard accelerator handlers.

```
BOOL PreTranslateAccelerator(
   LPMSG /* pMsg */,
   HRESULT& /* hRet */
);
```

Parameters

pMsg

Reserved.

hRet

Reserved.

Return Value

By default returns **FALSE**.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase::SendOnClose

Notifies all advisory sinks registered with the advise holder that the control has been closed.

```
HRESULT SendOnClose( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Sends a notification that the control has closed its advisory sinks.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase::SendOnDataChange

Notifies all advisory sinks registered with the advise holder that the control data has changed.

```
HRESULT SendOnDataChange(
   DWORD advf = 0
);
```

Parameters

advf

Advise flags that specify how the call to IAdviseSink::OnDataChange is made. Values are from the ADVF enumeration.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CComControlBase Class

CComControlBase::m_spDataAdviseHolder

Other Resources

CComControlBase Members

CComControlBase::SendOnRename

Notifies all advisory sinks registered with the advise holder that the control has a new moniker.

```
HRESULT SendOnRename(
   IMoniker* pmk
);
```

Parameters

pmk

Pointer to the new moniker of the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Sends a notification that the moniker for the control has changed.

See Also

Reference

CComControlBase Class

CComControlBase::m_spOleAdviseHolder

Other Resources

CComControlBase Members

CComControlBase::SendOnSave

Notifies all advisory sinks registered with the advise holder that the control has been saved.

```
HRESULT SendOnSave( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Sends a notification that the control has just saved its data.

See Also

Reference

CComControlBase Class

CComControlBase::m_spOleAdviseHolder

Other Resources

CComControlBase Members

CComControlBase::SendOnViewChange

Notifies all registered advisory sinks that the control's view has changed.

```
HRESULT SendOnViewChange(
   DWORD dwAspect,
   LONG lindex = -1
);
```

Parameters

dwAspect

The aspect or view of the control.

lindex

The portion of the view that has changed. Only -1 is valid.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

SendOnViewChange calls IAdviseSink::OnViewChange. The only value of *lindex* currently supported is -1, which indicates that the entire view is of interest.

See Also

Reference

CComControlBase Class

CComControlBase::m_spAdviseSink

Other Resources

CComControlBase Members

CComControlBase::SetControlFocus

Sets or removes the keyboard focus to or from the control.

```
BOOL SetControlFocus(
   BOOL bGrab
);
```

Parameters

bGrab

If **TRUE**, sets the keyboard focus to the calling control. If **FALSE**, removes the keyboard focus from the calling control, provided it has the focus.

Return Value

Returns **TRUE** if the control successfully receives focus; otherwise, **FALSE**.

Remarks

For a windowed control, the Windows API function SetFocus is called. For a windowless control, IOleInPlaceSiteWindowless::SetFocus is called. Through this call, a windowless control obtains the keyboard focus and can respond to window messages.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase::SetDirty

Sets the data member **m_bRequiresSave** to the value in *bDirty*.

```
void SetDirty(
   BOOL bDirty
);
```

Parameters

bDirty

Value of the data member CComControlBase::m_bRequiresSave.

Remarks

SetDirty(TRUE) should be called to flag that the control has changed since it was last saved. The value of **m_bRequiresSave** is retrieved with CComControlBase::GetDirty.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

GetAmbient Property Methods

For information about the GetAmbient Property Methods in **CComControlBase**, see CComControlBase Members.

CComControlBase::GetAmbientAppearance

Retrieves **DISPID_AMBIENT_APPEARANCE**, the current appearance setting for the control: 0 for flat and 1 for 3D.

```
HRESULT GetAmbientAppearance(
   short& nAppearance
);
```

Parameters

nAppearance

The property **DISPID_AMBIENT_APPEARANCE**.

Return Value

One of the standard HRESULT values.

Example

```
HRESULT OnDraw(ATL_DRAWINFO& di)
{
   short nAppearance;
   RECT& rc = *(RECT*)di.prcBounds;

   // draw 3D border if AmbientAppearance is not supported or is set to 1
   HRESULT hr = GetAmbientAppearance(nAppearance);
   if (hr!=S_OK || nAppearance==1)
   {
      DrawEdge(di.hdcDraw, &rc, EDGE_SUNKEN, BF_RECT);
   }
   else
   {
      Rectangle(di.hdcDraw, rc.left, rc.top, rc.right, rc.bottom);
   }

   SetTextAlign(di.hdcDraw, TA_CENTER|TA_BASELINE);
   LPCTSTR pszText = _T("ATL 8.0 : MyCtrl");
   TextOut(di.hdcDraw,
      (rc.left + rc.right) / 2,
      (rc.top + rc.bottom) / 2,
      pszText,
      lstrlen(pszText));

   return S_OK;
}
```

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientAutoClip

Retrieves **DISPID_AMBIENT_AUTOCLIP**, a flag indicating whether the container supports automatic clipping of the control display area.

```
HRESULT GetAmbientAutoClip(
   BOOL& bAutoClip
);
```

Parameters

bAutoClip

The property **DISPID_AMBIENT_AUTOCLIP**.

Return Value

One of the standard HRESULT values.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientBackColor

Retrieves **DISPID_AMBIENT_BACKCOLOR**, the ambient background color for all controls, defined by the container.

```
HRESULT GetAmbientBackColor(
   OLE_COLOR& BackColor
);
```

Parameters

BackColor

The property **DISPID_AMBIENT_BACKCOLOR**.

Return Value

One of the standard HRESULT values.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientCharSet

Retrieves **DISPID_AMBIENT_CHARSET**, the ambient character set for all controls, defined by the container.

```
HRESULT GetAmbientCharSet(
   BSTR& bstrCharSet
);
```

Parameters

bstrCharSet

The property **DISPID_AMBIENT_CHARSET**.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientCodePage

Retrieves **DISPID_AMBIENT_CODEPAGE**, the ambient code page for all controls, defined by the container.

```
HRESULT GetAmbientCodePage(
   ULONG& ulCodePage
);
```

Parameters

ulCodePage

The property **DISPID_AMBIENT_CODEPAGE**.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientDisplayAsDefault

Retrieves **DISPID_AMBIENT_DISPLAYASDEFAULT**, a flag that is **TRUE** if the container has marked the control in this site to be a default button, and therefore a button control should draw itself with a thicker frame.

```
HRESULT GetAmbientDisplayAsDefault(
   BOOL& bDisplayAsDefault
);
```

Parameters

bDisplayAsDefault

The property **DISPID_AMBIENT_DISPLAYASDEFAULT**.

Return Value

One of the standard HRESULT values.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientDisplayName

Retrieves **DISPID_AMBIENT_DISPLAYNAME**, the name the container has supplied to the control.

```
HRESULT GetAmbientDisplayName(
   BSTR& bstrDisplayName
);
```

Parameters

bstrDisplayName

The property **DISPID_AMBIENT_DISPLAYNAME**.

Return Value

One of the standard HRESULT values.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientFont

Retrieves a pointer to the container's ambient **IFont** interface.

```
HRESULT GetAmbientFont(
   IFont** ppFont
);
```

Parameters

ppFont

A pointer to the container's ambient IFont interface.

Return Value

One of the standard HRESULT values.

Remarks

If the property is **NULL**, the pointer is **NULL**. If the pointer is not **NULL**, the caller must release the pointer.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientFontDisp

Retrieves a pointer to the container's ambient **IFontDisp** dispatch interface.

```
HRESULT GetAmbientFontDisp(
   IFontDisp** ppFont
);
```

Parameters

ppFont

A pointer to the container's ambient IFontDisp dispatch interface.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If the property is **NULL**, the pointer is **NULL**. If the pointer is not **NULL**, the caller must release the pointer.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientForeColor

Retrieves **DISPID_AMBIENT_FORECOLOR**, the ambient foreground color for all controls, defined by the container.

```
HRESULT GetAmbientForeColor(
   OLE_COLOR& ForeColor
);
```

Parameters

ForeColor

The property **DISPID_AMBIENT_FORECOLOR**.

Return Value

One of the standard HRESULT values.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientLocaleID

Retrieves **DISPID_AMBIENT_LOCALEID**, the identifier of the language used by the container.

```
HRESULT GetAmbientLocaleID(
   LCID& lcid
);
```

Parameters

lcid

The property **DISPID_AMBIENT_LOCALEID**.

Return Value

One of the standard HRESULT values.

Remarks

The control can use this identifier to adapt its user interface to different languages.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientMessageReflect

Retrieves **DISPID_AMBIENT_MESSAGEREFLECT**, a flag indicating whether the container wants to receive window messages (such as **WM_DRAWITEM**) as events.

```
HRESULT GetAmbientMessageReflect(
   BOOL& bMessageReflect
);
```

Parameters

bMessageReflect

The property **DISPID_AMBIENT_MESSAGEREFLECT**.

Return Value

One of the standard HRESULT values.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientPalette

Retrieves **DISPID_AMBIENT_PALETTE**, used to access the container's **HPALETTE**.

```
HRESULT GetAmbientPalette(
   HPALETTE& hPalette
);
```

Parameters

hPalette

The property **DISPID_AMBIENT_PALETTE**.

Return Value

One of the standard HRESULT values.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientProperty

Retrieves the container property specified by *dispid*.

```
HRESULT GetAmbientProperty(
   DISPID dispid,
   VARIANT& var
);
```

Parameters

dispid

Identifier of the container property to be retrieved.

var

Variable to receive the property.

Return Value

One of the standard HRESULT values.

Remarks

ATL has provided a set of helper functions to retrieve specific properties, for example, CComControlBase::GetAmbientBackColor. If there is no suitable method available, use **GetAmbientProperty**.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientRightToLeft

Retrieves **DISPID_AMBIENT_RIGHTTOLEFT**, the direction in which content is displayed by the container.

```
HRESULT GetAmbientRightToLeft(
   BOOL& bRightToLeft
);
```

Parameters

bRightToLeft

The property **DISPID_AMBIENT_RIGHTTOLEFT**. Set to **TRUE** if content is displayed right to left, **FALSE** if it is displayed left to right.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientScaleUnits

Retrieves **DISPID_AMBIENT_SCALEUNITS**, the container's ambient units (such as inches or centimeters) for labeling displays.

```
HRESULT GetAmbientScaleUnits(
   BSTR& bstrScaleUnits
);
```

Parameters

bstrScaleUnits

The property **DISPID_AMBIENT_SCALEUNITS**.

Return Value

One of the standard HRESULT values.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientShowGrabHandles

Retrieves **DISPID_AMBIENT_SHOWGRABHANDLES**, a flag indicating whether the container allows the control to display grab handles for itself when active.

```
HRESULT GetAmbientShowGrabHandles(
   BOOL& bShowGrabHandles
);
```

Parameters

bShowGrabHandles

The property **DISPID_AMBIENT_SHOWGRABHANDLES**.

Return Value

One of the standard HRESULT values.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientShowHatching

Retrieves **DISPID_AMBIENT_SHOWHATCHING**, a flag indicating whether the container allows the control to display itself with a hatched pattern when the control's user interface is active.

```
HRESULT GetAmbientShowHatching(
   BOOL& bShowHatching
);
```

Parameters

bShowHatching

The property **DISPID_AMBIENT_SHOWHATCHING**.

Return Value

One of the standard HRESULT values.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientSupportsMnemonics

Retrieves **DISPID_AMBIENT_SUPPORTSMNEMONICS**, a flag indicating whether the container supports keyboard mnemonics.

```
HRESULT GetAmbientSupportsMnemonics(
   BOOL& bSupportsMnemonics
);
```

Parameters

bSupportsMnemonics

The property **DISPID_AMBIENT_SUPPORTSMNEMONICS**.

Return Value

One of the standard HRESULT values.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientTextAlign

Retrieves **DISPID_AMBIENT_TEXTALIGN**, the text alignment preferred by the container: 0 for general alignment (numbers right, text left), 1 for left alignment, 2 for center alignment, and 3 for right alignment.

```
HRESULT GetAmbientTextAlign(
   short& nTextAlign
);
```

Parameters

nTextAlign

The property **DISPID_AMBIENT_TEXTALIGN**.

Return Value

One of the standard HRESULT values.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientTopToBottom

Retrieves **DISPID_AMBIENT_TOPTOBOTTOM**, the direction in which content is displayed by the container.

```
HRESULT GetAmbientTopToBottom(
   BOOL& bTopToBottom
);
```

Parameters

bTopToBottom

The property **DISPID_AMBIENT_TOPTOBOTTOM**. Set to **TRUE** if text is displayed top to bottom, **FALSE** if it is displayed bottom to top.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientUIDead

Retrieves **DISPID_AMBIENT_UIDEAD**, a flag indicating whether the container wants the control to respond to user-interface actions.

```
HRESULT GetAmbientUIDead(
   BOOL& bUIDead
);
```

Parameters

bUIDead

The property **DISPID_AMBIENT_UIDEAD**.

Return Value

One of the standard HRESULT values.

Remarks

If **TRUE**, the control should not respond. This flag applies regardless of the **DISPID_AMBIENT_USERMODE** flag. See CComControlBase::GetAmbientUserMode.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase::GetAmbientUserMode

Retrieves **DISPID_AMBIENT_USERMODE**, a flag indicating whether the container is in run-mode (**TRUE**) or design-mode (**FALSE**).

```
HRESULT GetAmbientUserMode(
   BOOL& bUserMode
);
```

Parameters

bUserMode

The property **DISPID_AMBIENT_USERMODE**.

Return Value

One of the standard HRESULT values.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase Members

CComControlBase Data Members

For information about the data members in **CComControlBase**, see CComControlBase Members.

CComControlBase::m_bAutoSize

Flag indicating the control cannot be any other size.

```
unsigned m_bAutoSize:1;
```

Remarks

This flag is checked by **IOleObjectImpl::SetExtent** and, if **TRUE**, causes the function to return **E_FAIL**.

Note
To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

If you add the **Auto Size** option on the Stock Properties tab of the ATL Control Wizard, the wizard automatically creates this data member in your control class, creates put and get methods for the property, and supports IPropertyNotifySink to automatically notify the container when the property changes.

See Also

Reference

CComControlBase Class

IOleObjectImpl::SetExtent

Other Resources

CComControlBase Members

CComControlBase::m_bDrawFromNatural

Flag indicating that **IDataObjectImpl::GetData** and **CComControlBase::GetZoomInfo** should set the control size from **m_sizeNatural** rather than from **m_sizeExtent**.

```
unsigned m_bDrawFromNatural:1;
```

Remarks

Note
To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

See Also

Reference

CComControlBase Class
IDataObjectImpl::GetData
CComControlBase::m_sizeNatural
CComControlBase::m_sizeExtent

Other Resources

CComControlBase Members

CComControlBase::m_bDrawGetDataInHimetric

Flag indicating that **IDataObjectImpl::GetData** should use HIMETRIC units and not pixels when drawing.

```
unsigned m_bDrawGetDataInHimetric:1;
```

Remarks

Each logical HIMETRIC unit is 0.01 millimeter.

Note
To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

See Also

Reference

CComControlBase Class

IDataObjectImpl::GetData

Other Resources

CComControlBase Members

CComControlBase::m_bInPlaceActive

Flag indicating the control is in-place active.

```
unsigned m_bInPlaceActive:1;
```

Remarks

This means the control is visible and its window, if any, is visible, but its menus and toolbars may not be active. The **m_bUIActive** flag indicates the control's user interface, such as menus, is also active.

Note

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

See Also

Reference

CComControlBase Class

CComControlBase::m_bUIActive

Other Resources

CComControlBase Members

CComControlBase::m_bInPlaceSiteEx

Flag indicating the container supports the **IOleInPlaceSiteEx** interface and OCX96 control features, such as windowless and flicker-free controls.

```
unsigned m_bInPlaceSiteEx:1;
```

Remarks

Note
To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

The data member **m_spInPlaceSite** points to an IOleInPlaceSite, IOleInPlaceSiteEx, or IOleInPlaceSiteWindowless interface, depending on the value of the **m_bWndLess** and **m_bInPlaceSiteEx** flags. (The data member **m_bNegotiatedWnd** must be **TRUE** for the **m_spInPlaceSite** pointer to be valid.)

If **m_bWndLess** is **FALSE** and **m_bInPlaceSiteEx** is **TRUE**, **m_spInPlaceSite** is an **IOleInPlaceSiteEx** interface pointer. See m_spInPlaceSite for a table showing the relationship among these three data members.

See Also

Reference

CComControlBase Class
CComControlBase::m_bWndLess
CComControlBase::m_bNegotiatedWnd
CComControlBase::m_spInPlaceSite

Other Resources

CComControlBase Members

CComControlBase::m_bNegotiatedWnd

Flag indicating whether or not the control has negotiated with the container about support for OCX96 control features (such as flicker-free and windowless controls), and whether the control is windowed or windowless.

```
unsigned m_bNegotiatedWnd:1;
```

Remarks

Note
To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

The **m_bNegotiatedWnd** flag must be **TRUE** for the **m_spInPlaceSite** pointer to be valid.

See Also

Reference

CComControlBase Class
CComControlBase::m_bWndLess
CComControlBase::m_spInPlaceSite

Other Resources

CComControlBase Members

CComControlBase::m_bRecomposeOnResize

Flag indicating the control wants to recompose its presentation when the container changes the control's display size.

```
unsigned m_bRecomposeOnResize:1;
```

Remarks

Note
To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

This flag is checked by IOleObjectImpl::SetExtent and, if **TRUE**, **SetExtent** notifies the container of view changes. if this flag is set, the **OLEMISC_RECOMPOSEONRESIZE** bit in the OLEMISC enumeration should also be set.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase::m_bRequiresSave

Flag indicating the control has changed since it was last saved.

```
unsigned m_bRequiresSave:1;
```

Remarks

The value of **m_bRequiresSave** can be set with CComControlBase::SetDirty and retrieved with CComControlBase::GetDirty.

Note
To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase::m_bResizeNatural

Flag indicating the control wants to resize its natural extent (its unscaled physical size) when the container changes the control's display size.

```
unsigned m_bResizeNatural:1;
```

Remarks

This flag is checked by **IOleObjectImpl::SetExtent** and, if **TRUE**, the size passed into **SetExtent** is assigned to **m_sizeNatural**.

The size passed into **SetExtent** is always assigned to **m_sizeExtent**, regardless of the value of **m_bResizeNatural**.

Note
To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

See Also

Reference

CComControlBase Class
IOleObjectImpl::SetExtent
CComControlBase::m_sizeNatural
CComControlBase::m_sizeExtent

Other Resources

CComControlBase Members

CComControlBase::m_bUIActive

Flag indicating the control's user interface, such as menus and toolbars, is active.

```
unsigned m_bUIActive:1;
```

Remarks

The **m_bInPlaceActive** flag indicates that the control is active, but not that its user interface is active.

Note
To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

See Also

Reference

CComControlBase Class

CComControlBase::m_bInPlaceActive

Other Resources

CComControlBase Members

CComControlBase::m_bUsingWindowRgn

Flag indicating the control is using the container-supplied window region.

```
unsigned m_bUsingWindowRgn:1;
```

Remarks

Note

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase::m_bWasOnceWindowless

Flag indicating the control has been windowless, but may or may not be windowless now.

```
unsigned m_bWasOnceWindowless:1;
```

Remarks

Note

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

See Also

Reference

CComControlBase Class

CComControlBase::m_bWndLess

Other Resources

CComControlBase Members

CComControlBase::m_bWindowOnly

Flag indicating the control should be windowed, even if the container supports windowless controls.

```
unsigned m_bWindowOnly:1;
```

Remarks

Note

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

See Also

Reference

CComControlBase Class

CComControlBase::m_bWndLess

Other Resources

CComControlBase Members

CComControlBase::m_bWndLess

Flag indicating the control is windowless.

```
unsigned m_bWndLess:1;
```

Remarks

Note

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

The data member **m_spInPlaceSite** points to an IOleInPlaceSite, IOleInPlaceSiteEx, or IOleInPlaceSiteWindowless interface, depending on the value of the **m_bWndLess** and CComControlBase::m_bInPlaceSiteEx flags. (The data member CComControlBase::m_bNegotiatedWnd must be **TRUE** for the CComControlBase::m_spInPlaceSite pointer to be valid.)

If **m_bWndLess** is **TRUE**, **m_spInPlaceSite** is an **IOleInPlaceSiteWindowless** interface pointer. See CComControlBase::m_spInPlaceSite for a table showing the complete relationship between these data members.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase::m_hWndCD

Contains a reference to the window handle associated with the control.

```
HWND& m_hWndCD;
```

Remarks

Note

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

See Also

Reference

CComControlBase Class

CComControlBase::CComControlBase

Other Resources

CComControlBase Members

CComControlBase::m_nFreezeEvents

A count of the number of times the container has frozen events (refused to accept events) without an intervening thaw of events (acceptance of events).

```
short m_nFreezeEvents;
```

Remarks

Note

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

See Also

Reference

CComControlBase Class

IOleControl::FreezeEvents

Other Resources

CComControlBase Members

CComControlBase::m_rcPos

The position in pixels of the control, expressed in the coordinates of the container.

```
RECT m_rcPos;
```

Remarks

Note

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

See Also

Reference

CComControlBase Class
CComControlBase::m_sizeExtent
CComControlBase::m_sizeNatural
RECT

Other Resources

CComControlBase Members

CComControlBase::m_sizeExtent

The extent of the control in HIMETRIC units (each unit is 0.01 millimeters) for a particular display.

```
SIZE m_sizeExtent;
```

Remarks

Note

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

This size is scaled by the display. The control's physical size is specified in the **m_sizeNatural** data member and is fixed.

You can convert the size to pixels with the global function AtlHiMetricToPixel.

See Also

Reference

CComControlBase Class
SIZE
CComControlBase::m_sizeNatural

Other Resources

CComControlBase Members

CComControlBase::m_sizeNatural

The physical size of the control in HIMETRIC units (each unit is 0.01 millimeters).

```
SIZE m_sizeNatural;
```

Remarks

Note

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

This size is fixed, while the size in **m_sizeExtent** is scaled by the display.

You can convert the size to pixels with the global function AtlHiMetricToPixel.

See Also

Reference

CComControlBase Class
SIZE
CComControlBase::m_sizeExtent

Other Resources

CComControlBase Members

CComControlBase::m_spAdviseSink

A direct pointer to the advisory connection on the container (the container's IAdviseSink).

```
CComPtr<IAdviseSink> m_spAdviseSink;
```

Remarks

Note

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComControlBase::m_spAmbientDispatch

A **CComDispatchDriver** object that lets you retrieve and set an object's properties through an **IDispatch** pointer. For more information, see Implementing the IDispatch Interface

```
CComDispatchDriver m_spAmbientDispatch;
```

Remarks

Note

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

See Also

Reference

CComControlBase Class

CComDispatchDriver

Other Resources

CComControlBase Members

CComControlBase::m_spClientSite

A pointer to the control's client site within the container.

```
CComPtr<IOleClientSite> m_spClientSite;
```

Remarks

Note

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

See Also

Reference

CComControlBase Class
CComPtr Class
IOleClientSite

Other Resources

CComControlBase Members

CComControlBase::m_spDataAdviseHolder

Provides a standard means to hold advisory connections between data objects and advise sinks.

```
CComPtr<IDataAdviseHolder> m_spDataAdviseHolder;
```

Remarks

Note
To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

A data object is a control that can transfer data and that implements IDataObject, whose methods specify the format and transfer medium of the data.

The interface **m_spDataAdviseHolder** implements the IDataObject::DAdvise and IDataObject::DUnadvise methods to establish and delete advisory connections to the container. The control's container must implement an advise sink by supporting the IAdviseSink interface.

See Also

Reference

CComControlBase Class

CComPtr Class

Other Resources

CComControlBase Members

CComControlBase::m_spInPlaceSite

A pointer to the container's IOleInPlaceSite, IOleInPlaceSiteEx, or IOleInPlaceSiteWindowless interface pointer.

```
CComPtr<IOleInPlaceSiteWindowless> m_spInPlaceSite;
```

Remarks

Note
To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

The **m_spInPlaceSite** pointer is valid only if the m_bNegotiatedWnd flag is **TRUE**.

The following table shows how the **m_spInPlaceSite** pointer type depends on the m_bWndLess and m_bInPlaceSiteEx data member flags:

m_spInPlaceSite Type	m_bWndLess Value	m_bInPlaceSiteEx Value
IOleInPlaceSiteWindowless	**TRUE**	**TRUE** or **FALSE**
IOleInPlaceSiteEx	**FALSE**	**TRUE**
IOleInPlaceSite	**FALSE**	**FALSE**

See Also

Reference

CComControlBase Class

CComPtr Class

Other Resources

CComControlBase Members

CComControlBase::m_spOleAdviseHolder

Provides a standard implementation of a way to hold advisory connections.

```
CComPtr<IOleAdviseHolder> m_spOleAdviseHolder;
```

Remarks

Note

To use this data member within your control class, you must declare it as a data member in your control class. Your control class will not inherit this data member from the base class because it is declared within a union in the base class.

The interface **m_spOleAdviseHolder** implements the IOleObject::Advise and IOleObject::Unadvise methods to establish and delete advisory connections to the container. The control's container must implement an advise sink by supporting the IAdviseSink interface.

See Also

Reference

CComControlBase Class

CComPtr Class

Other Resources

CComControlBase Members

CComControlBase Typedefs

For information about the typedefs in **CComControlBase**, see CComControlBase Members.

CComControlBase::AppearanceType

Override if your **m_nAppearance** stock property isn't of type **short**.

```
typedef short AppearanceType;
```

Remarks

The ATL Control Wizard adds **m_nAppearance** stock property of type short. Override **AppearanceType** if you use a different data type.

See Also

Reference

CComControlBase Class

Other Resources

CComControlBase Members

CComCriticalSection Class

This class provides methods for obtaining and releasing ownership of a critical section object.

```
class CComCriticalSection
```

Remarks

CComCriticalSection is similar to class CComAutoCriticalSection, except that you must explicitly initialize and release the critical section.

Typically, you use **CComCriticalSection** through the **typedef** name CriticalSection. This name references **CComCriticalSection** when CComMultiThreadModel is being used.

See CComCritSecLock Class for a safer way to use this class than calling **Lock** and **Unlock** directly.

Requirements

Header: atlcore.h

Example

See the PooledAsync Sample.

See Also

Tasks

PooledAsync Sample: Increases Performance with an Additional Thread Pool

Reference

CComFakeCriticalSection Class

CComCritSecLock Class

Other Resources

CComCriticalSection Members

ATL Class Overview

CComCriticalSection Members

Methods

CComCriticalSection	The constructor.
Init	Creates and initializes a critical section object.
Lock	Obtains ownership of the critical section object.
Term	Releases system resources used by the critical section object.
Unlock	Releases ownership of the critical section object.

Data Members

m_sec	A **CRITICAL_SECTION** object.

See Also

Reference

CComCriticalSection Class

CComCriticalSection Methods

For information about the methods in **CComCriticalSection**, see CComCriticalSection Members.

CComCriticalSection::CComCriticalSection

The constructor.

```
CComCriticalSection( ) throw( );
```

Remarks

Sets the m_sec data member to NULL.

See Also

Reference

CComCriticalSection Class

Other Resources

CComCriticalSection Members

CComCriticalSection::Init

Calls the Win32 function InitializeCriticalSection, which initializes the critical section object contained in the m_sec data member.

```
HRESULT Init( ) throw( );
```

Return Value

Returns **S_OK** on success, **E_OUTOFMEMORY** or **E_FAIL** on failure.

See Also

Reference

CComCriticalSection Class

Other Resources

CComCriticalSection Members

CComCriticalSection::Lock

Calls the Win32 function EnterCriticalSection, which waits until the thread can take ownership of the critical section object contained in the m_sec data member.

```
HRESULT Lock( ) throw( );
```

Return Value

Returns **S_OK** on success, **E_OUTOFMEMORY** or **E_FAIL** on failure.

Remarks

The critical section object must first be initialized with a call to the Init method. When the protected code has finished executing, the thread must call Unlock to release ownership of the critical section.

See Also

Reference

CComCriticalSection Class

Other Resources

CComCriticalSection Members

CComCriticalSection::Term

Calls the Win32 function DeleteCriticalSection, which releases all resources used by the critical section object contained in the m_sec data member.

```
HRESULT Term( ) throw( );
```

Return Value

Returns **S_OK**.

Remarks

Once **Term** has been called, the critical section can no longer be used for synchronization.

See Also

Reference

CComCriticalSection Class

Other Resources

CComCriticalSection Members

CComCriticalSection::Unlock

Calls the Win32 function LeaveCriticalSection, which releases ownership of the critical section object contained in the m_sec data member.

```
HRESULT Unlock( ) throw( );
```

Return Value

Returns **S_OK**.

Remarks

To first obtain ownership, the thread must call the Lock method. Each call to **Lock** requires a corresponding call to **Unlock** to release ownership of the critical section.

See Also

Reference

CComCriticalSection Class

Other Resources

CComCriticalSection Members

CComCriticalSection Data Members

For information about the data members in **CComCriticalSection**, see CComCriticalSection Members.

CComCriticalSection::m_sec

Contains a critical section object that is used by all **CComCriticalSection** methods.

```
CRITICAL_SECTION m_sec;
```

See Also

Reference

CComCriticalSection Class
CComCriticalSection::Lock
CComCriticalSection::Unlock
CComCriticalSection::Init
CComCriticalSection::Term

Other Resources

CComCriticalSection Members

CComCritSecLock Class

This class provides methods for locking and unlocking a critical section object.

```
template<
   class TLock
> class CComCritSecLock
```

Parameters

TLock

The object to be locked and unlocked.

Remarks

Use this class to lock and unlock objects in a safer way than with the CComCriticalSection Class or CComAutoCriticalSection Class.

Requirements

Header: atlbase.h

See Also

Reference

CComCriticalSection Class

CComAutoCriticalSection Class

Other Resources

CComCritSecLock Members

CComCritSecLock Members

Methods

CComCritSecLock	The constructor.
~CComCritSecLock	The destructor.
Lock	Call this method to lock the critical section object.
Unlock	Call this method to unlock the critical section object.

See Also

Reference

CComCritSecLock Class

CComCritSecLock Methods

For information about the methods in **CComCritSecLock**, see CComCritSecLock Members.

CComCritSecLock::CComCritSecLock

The constructor.

```
CComCritSecLock(
   TLock& cs,
   bool bInitialLock = true
);
```

Parameters

cs

The critical section object.

bInitialLock

The initial lock state: **true** means locked.

Remarks

Initializes the critical section object.

See Also

Reference

CComCritSecLock Class

CComCritSecLock::~CComCritSecLock

Other Resources

CComCritSecLock Members

CComCritSecLock::~CComCritSecLock

The destructor.

```
~CComCritSecLock( ) throw( );
```

Remarks

Unlocks the critical section object.

See Also

Reference

CComCritSecLock Class

CComCritSecLock::CComCritSecLock

Other Resources

CComCritSecLock Members

CComCritSecLock::Lock

Call this method to lock the critical section object.

```
HRESULT Lock( ) throw( );
```

Return Value

Returns S_OK if the object has successfully been locked, or an error HRESULT on failure.

Remarks

If the object is already locked, an ASSERT error will occur in debug builds.

See Also

Reference

CComCritSecLock Class

CComCritSecLock::Unlock

Other Resources

CComCritSecLock Members

CComCritSecLock::Unlock

Call this method to unlock the critical section object.

```
void Unlock( ) throw( );
```

Remarks

If the object is already unlocked, an ASSERT error will occur in debug builds.

See Also

Reference

CComCritSecLock Class

CComCritSecLock::Lock

Other Resources

CComCritSecLock Members

CComCurrency Class

CComCurrency has methods and operators for creating and managing a CURRENCY object.

```
class CComCurrency
```

Remarks

CComCurrency is a wrapper for the **CURRENCY** data type. **CURRENCY** is implemented as an 8-byte two's-complement integer value scaled by 10,000. This gives a fixed-point number with 15 digits to the left of the decimal point and 4 digits to the right. The **CURRENCY** data type is extremely useful for calculations involving money, or for any fixed-point calculations where accuracy is important.

The **CComCurrency** wrapper implements arithmetic, assignment, and comparison operations for this fixed-point type. The supported applications have been selected to control the rounding errors that can occur during fixed-point calculations.

The **CComCurrency** object provides access to the numbers on either side of the decimal point in the form of two components: an integer component which stores the value to the left of the decimal point, and a fractional component which stores the value to the right of the decimal point. The fractional component is stored internally as an integer value between -9999 (**CY_MIN_FRACTION**) and +9999 (**CY_MAX_FRACTION**). The method CComCurrency::GetFraction returns a value scaled by a factor of 10000 (**CY_SCALE**).

When specifying the integer and fractional components of a **CComCurrency** object, remember that the fractional component is a number in the range 0 to 9999. This is important when dealing with a currency such as the US dollar that expresses amounts using only two significant digits after the decimal point. Even though the last two digits are not displayed, they must be taken into account.

Value	Possible CComCurrency assignments
$10.50	CComCurrency(10,5000) *or* CComCurrency(10.50)
$10.05	CComCurrency(10,500) *or* CComCurrency(10.05)

The values **CY_MIN_FRACTION**, **CY_MAX_FRACTION**, and **CY_SCALE** are defined in atlcur.h.

Requirements

Header: atlcur.h

See Also

Reference

COleCurrency Class

CURRENCY

Other Resources

CComCurrency Members

ATL Class Overview

CComCurrency Members

Methods

CComCurrency	The constructor for a **CComCurrency** object.
GetCurrencyPtr	Returns the address of an **m_currency** data member.
GetFraction	Call this method to return the fractional component of a **CComCurrency** object.
GetInteger	Call this method to return the integer component of a **CComCurrency** object.
Round	Call this method to round a **CComCurrency** object to the nearest integer value.
SetFraction	Call this method to set the fractional component of a **CComCurrency** object.
SetInteger	Call this method to set the integer component of a **CComCurrency** object.

Operators

operator !=	Compares two **CComCurrency** objects for inequality.
operator *	This operator is used to perform multiplication on a **CComCurrency** object.
operator *=	This operator is used to perform multiplication on a **CComCurrency** object and assign it the result.
operator +	This operator is used to perform addition on a **CComCurrency** object.
operator +=	This operator is used to perform addition on a **CComCurrency** object and assign the result to the curren t object.
operator -	This operator is used to perform subtraction on a **CComCurrency** object.
operator -=	This operator is used to perform subtraction on a **CComCurrency** object and assign it the result.
operator /	This operator is used to perform division on a **CComCurrency** object.
operator /=	This operator is used to perform division on a **CComCurrency** object and assign it the result.
operator =	This operator assigns the **CComCurrency** object to a new value.
operator ==	This operator compares two **CComCurrency** objects for equality.
operator >	This operator compares two **CComCurrency** objects to determine the larger.
operator >=	This operator compares two **CComCurrency** objects to determine equality or the larger.
operator <	This operator compares two **CComCurrency** objects to determine the lesser.
operator <=	This operator compares two **CComCurrency** objects to determine equality or the lesser.
operator CURRENCY	Casts a **CURRENCY** object.

Data Members

m_currency	The **CURRENCY** variable created by your class instance.

See Also

Reference

CComCurrency Class

CComCurrency Methods

For information about the methods in **CComCurrency**, see CComCurrency Members.

CComCurrency::CComCurrency

The constructor.

```
CComCurrency( ) throw( );
CComCurrency(
   const CComCurrency& curSrc
) throw( );
CComCurrency(
   CURRENCY cySrc
) throw( );
CComCurrency(
   DECIMAL dSrc
);
CComCurrency(
   ULONG ulSrc
);
CComCurrency(
   USHORT usSrc
);
CComCurrency(
   CHAR cSrc
);
CComCurrency(
   DOUBLE dSrc
);
CComCurrency(
   FLOAT fSrc
);
CComCurrency(
   LONG lSrc
);
CComCurrency(
   SHORT sSrc
);
CComCurrency(
   BYTE bSrc
);
CComCurrency(
   LONGLONG nInteger,
   SHORT nFraction
);
explicit CComCurrency(
   LPDISPATCH pDispSrc
);
explicit CComCurrency(
   const VARIANT& varSrc
);
explicit CComCurrency(
   LPCWSTR szSrc
);
explicit CComCurrency(
   LPCSTR szSrc
);
```

Parameters

curSrc

An existing **CComCurrency** object.

cySrc

A variable of type **CURRENCY**.

bSrc, *dSrc*, *fSrc*, *lSrc*, *sSrc*, *ulSrc*, *usSrc*

The initial value given to the member variable **m_currency**.

cSrc

A character containing the initial value given to the member variable **m_currency**.

nInteger, *nFraction*

The integer and fractional components of the initial monetary value. See the CComCurrency overview for more information.

pDispSrc

An **IDispatch** pointer.

varSrc

A variable of type **VARIANT**. The locale of the current thread is used to perform the conversion.

szSrc

A Unicode or ANSI string containing the initial value. The locale of the current thread is used to perform the conversion.

Remarks

The constructor sets the initial value of CComCurrency::m_currency, and accepts a wide range of data types, including integers, strings, floating-point numbers, **CURRENCY** variables, and other **CComCurrency** objects. If no value is provided, **m_currency** is set to 0.

In the event of an error, such as an overflow, the constructors lacking an empty exception specification (**throw()**) call **AtlThrow** with an HRESULT describing the error.

When using floating-point or double values to assign a value, note that **CComCurrency(10.50)** is equivalent to **CComCurrency(10,5000)** and not **CComCurrency(10,50)**.

See Also

Reference

CComCurrency Class

Other Resources

CComCurrency Members

CComCurrency::GetCurrencyPtr

Returns the address of an **m_currency** data member.

```
CURRENCY * GetCurrencyPtr( ) throw();
```

Return Value

Returns the address of an **m_currency** data member

See Also

Reference

CComCurrency Class

CComCurrency::operator CURRENCY

Other Resources

CComCurrency Members

CComCurrency::GetFraction

Call this method to return the fractional component of the **CComCurrency** object.

```
SHORT GetFraction( ) const;
```

Return Value

Returns the fractional component of the **m_currency** data member.

Remarks

The fractional component is a 4-digit integer value between -9999 (**CY_MIN_FRACTION**) and +9999 (**CY_MAX_FRACTION**). **GetFraction** returns this value scaled by 10000 (**CY_SCALE**). The values of **CY_MIN_FRACTION**, **CY_MAX_FRACTION**, and **CY_SCALE** are defined in atlcur.h.

Example

```
// CComCurrency::GetFraction Example
   CComCurrency cur(10,5000);
   int nFract;
   nFract = cur.GetFraction();
   ATLASSERT(nFract == 5000);
```

See Also

Reference

CComCurrency Class
CComCurrency::SetFraction
CComCurrency::GetInteger
CComCurrency::SetInteger

Other Resources

CComCurrency Members

CComCurrency::GetInteger

Call this method to get the integer component of a **CComCurrency** object.

```
LONGLONG GetInteger( ) const;
```

Return Value

Returns the integer component of the **m_currency** data member.

Example

```
// CComCurrency::GetInteger Example
   CComCurrency cur(10,5000);
   LONGLONG nInteger;
   nInteger = cur.GetInteger();
   ATLASSERT(nInteger == 10);
```

See Also

Reference

CComCurrency Class
CComCurrency::SetInteger
CComCurrency::GetFraction
CComCurrency::SetFraction

Other Resources

CComCurrency Members

CComCurrency::Round

Call this method to round the currency to a specified number of decimal places.

```
HRESULT Round(
   int nDecimals
);
```

Parameters

nDecimals

The number of digits to which **m_currency** will be rounded, in the range 0 to 4.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

```
// CComCurrency::Round Example
   CComCurrency cur(10,1234);
   cur.Round(3);
   ATLASSERT(cur.GetFraction() == 1230);
```

See Also

Reference

CComCurrency Class

Other Resources

CComCurrency Members

CComCurrency::SetFraction

Call this method to set the fractional component of a **CComCurrency** object.

```
HRESULT SetFraction(
   SHORT nFraction
);
```

Parameters

nFraction

The value to be assigned to the fractional component of the **m_currency** data member. The sign of the fractional component must the same as the integer component, and the value must be in range -9999 (**CY_MIN_FRACTION**) to +9999 (**CY_MAX_FRACTION**).

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

```
// CComCurrency::SetFraction Example
   CComCurrency cur(10,0);
   cur.SetFraction(5000);
   ATLASSERT(CComCurrency(10,5000) == cur);
```

See Also

Reference

CComCurrency Class
CComCurrency::GetFraction
CComCurrency::SetInteger
CComCurrency::SetInteger

Other Resources

CComCurrency Members

CComCurrency::SetInteger

Call this method to set the integer component of a **CComCurrency** object.

```
HRESULT SetInteger(
   LONGLONG nInteger
);
```

Parameters

nInteger

The value to be assigned to the integer component of the **m_currency** data member. The sign of the integer component must match the sign of the existing fractional component.

nInteger must be in the range **CY_MIN_INTEGER** to **CY_MAX_INTEGER** inclusive. These values are defined in atlcur.h.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

```
// CComCurrency::SetInteger Example
   CComCurrency cur(0,5000);
   cur.SetInteger(10);
   ATLASSERT(CComCurrency(10,5000) == cur);
```

See Also

Reference

CComCurrency Class
CComCurrency::GetInteger
CComCurrency::GetFraction
CComCurrency::SetFraction

Other Resources

CComCurrency Members

CComCurrency Operators

For information about the operators in **CComCurrency**, see CComCurrency Members.

CComCurrency::operator -

This operator is used to perform subtraction on a **CComCurrency** object.

```
CComCurrency operator -( ) const;
CComCurrency operator -(
   const CComCurrency & cur
) const;
```

Parameters

cur

A **CComCurrency** object.

Return Value

Returns a **CComCurrency** object representing the result of the subtraction. In the event of an error, such as an overflow, this operator calls **AtlThrow** with an HRESULT describing the error.

Example

```
// CComCurrency::operator - Example
   CComCurrency cur1(10,5000),cur2;
   cur2=cur1-CComCurrency(4,5000);
   ATLASSERT(cur2 == CComCurrency(6,0));
```

See Also

Reference

CComCurrency Class
CComCurrency::operator -=
CComCurrency::operator +
CComCurrency::operator *
CComCurrency::operator /

Other Resources

CComCurrency Members

CComCurrency::operator !=

This operator compares two objects for inequality.

```
bool operator !=(
   const CComCurrency & cur
) const;
```

Parameters

cur

The **CComCurrency** object to be compared.

Return Value

Returns **true** if the item being compared is not equal to the **CComCurrency** object; otherwise, **false**.

Example

```
// CComCurrency::operator != Example
   CComCurrency cur1(10,5000), cur2(10,5001);
   ATLASSERT(cur1 != cur2);
```

See Also

Reference

CComCurrency Class

CComCurrency::operator ==

Other Resources

CComCurrency Members

CComCurrency::operator *

This operator is used to perform multiplication on a **CComCurrency** object.

```
CComCurrency operator *(
   long nOperand
) const;
CComCurrency operator *(
   const CComCurrency & cur
) const;
```

Parameters

nOperand

The multiplier.

cur

The **CComCurrency** object used as the multiplier.

Return Value

Returns a **CComCurrency** object representing the result of the multiplication. In the event of an error, such as an overflow, this operator calls **AtlThrow** with an HRESULT describing the error.

Example

```
// CComCurrency::operator * Example
   CComCurrency cur1(10,5000), cur2;
   cur2 = cur1 * 2;
   ATLASSERT(cur2 == CComCurrency(21,0));
```

See Also

Reference

CComCurrency Class
CComCurrency::operator *=
CComCurrency::operator +
CComCurrency::operator -
CComCurrency::operator /

Other Resources

CComCurrency Members

CComCurrency::operator *=

This operator is used to perform multiplication on a **CComCurrency** object and assign it the result.

```
const CComCurrency & operator *=(
   long nOperand
);
const CComCurrency & operator *=(
   const CComCurrency & cur
);
```

Parameters

nOperand

The multiplier.

cur

The **CComCurrency** object used as the multiplier.

Return Value

Returns the updated **CComCurrency** object. In the event of an error, such as an overflow, this operator calls **AtlThrow** with an HRESULT describing the error.

Example

```
// CComCurrency::operator *= Example
   CComCurrency cur(10,5000);
   cur*=2;
   ATLASSERT(cur == CComCurrency(21,0));
```

See Also

Reference

CComCurrency Class
CComCurrency::operator *
CComCurrency::operator +
CComCurrency::operator -
CComCurrency::operator /

Other Resources

CComCurrency Members

CComCurrency::operator /

This operator is used to perform division on a **CComCurrency** object.

```
CComCurrency operator /(
   long nOperand
) const;
```

Parameters

nOperand

The divisor.

Return Value

Returns a **CComCurrency** object representing the result of the division. If the divisor is 0, an assert failure will occur.

Example

```
// CComCurrency::operator / Example
   CComCurrency cur1(10,5000),cur2;
   cur2=cur1/10;
   ATLASSERT(cur2 == CComCurrency(1,500));
```

See Also

Reference

CComCurrency Class
CComCurrency::operator /=
CComCurrency::operator +
CComCurrency::operator -
CComCurrency::operator *

Other Resources

CComCurrency Members

CComCurrency::operator /=

This operator is used to perform division on a **CComCurrency** object and assign it the result.

```
const CComCurrency & operator /=(
   long nOperand
);
```

Parameters

nOperand

The divisor.

Return Value

Returns the updated **CComCurrency** object. If the divisor is 0, an assert failure will occur.

Example

```
// CComCurrency::operator /= Example
   CComCurrency cur(10,5000);
   cur /= 10;
   ATLASSERT(cur == CComCurrency(1,500));
```

See Also

Reference

CComCurrency Class
CComCurrency::operator /
CComCurrency::operator +
CComCurrency::operator -
CComCurrency::operator *

Other Resources

CComCurrency Members

CComCurrency::operator +

This operator is used to perform addition on a **CComCurrency** object.

```
CComCurrency operator +(
   const CComCurrency & cur
) const;
```

Parameters

cur

The **CComCurrency** object to be added to the original object.

Return Value

Returns a **CComCurrency** object representing the result of the addition. In the event of an error, such as an overflow, this operator calls **AtlThrow** with an HRESULT describing the error.

Example

```
// CComCurrency::operator + Example
   CComCurrency cur1(10,5000),cur2;
   cur2=cur1+CComCurrency(4,5000);
   ATLASSERT(cur2 == CComCurrency(15,0));
```

See Also

Reference

CComCurrency Class
CComCurrency::operator *=
CComCurrency::operator +
CComCurrency::operator -
CComCurrency::operator /

Other Resources

CComCurrency Members

CComCurrency::operator +=

This operator is used to perform addition on a **CComCurrency** object and assign the result to the current object.

```
const CComCurrency & operator +=(
   const CComCurrency & cur
);
```

Parameters

cur

The **CComCurrency** object.

Return Value

Returns the updated **CComCurrency** object. In the event of an error, such as an overflow, this operator calls **AtlThrow** with an HRESULT describing the error.

Example

```
// CComCurrency::operator += Example
   CComCurrency cur(10,2500);
   cur+=CComCurrency(4,2500);
   ATLASSERT(cur == CComCurrency(14,5000));
```

See Also

Reference

CComCurrency Class
CComCurrency::operator *=
CComCurrency::operator +
CComCurrency::operator -
CComCurrency::operator /

Other Resources

CComCurrency Members

CComCurrency::operator <

This operator compares two **CComCurrency** objects to determine the lesser.

```
bool operator <(
   const CComCurrency & cur
) const;
```

Parameters

cur

A **CComCurrency** object.

Return Value

Returns **true** if the first object is less than the second, **false** otherwise.

Example

```
// CComCurrency::operator < Example
   CComCurrency cur1(10,4900);
   CComCurrency cur2(10,5000);
   ATLASSERT(cur1 < cur2);
```

See Also

Reference

CComCurrency Class
CComCurrency::operator >
CComCurrency::operator >=
CComCurrency::operator <=

Other Resources

CComCurrency Members

CComCurrency::operator <=

This operator compares two **CComCurrency** objects to determine equality or the lesser.

```
bool operator <=(
   const CComCurrency & cur
) const;
```

Parameters

cur

A **CComCurrency** object.

Return Value

Returns **true** if the first object is less than or equal to the second, **false** otherwise.

Example

```
// CComCurrency::operator <= Example
   CComCurrency cur1(10,4900);
   CComCurrency cur2(10,5000);
   ATLASSERT(cur1 <= cur2);
```

See Also

Reference

CComCurrency Class
CComCurrency::operator >=
CComCurrency::operator >
CComCurrency::operator <

Other Resources

CComCurrency Members

CComCurrency::operator =

This operator assigns the **CComCurrency** object to a new value.

```
const CComCurrency & operator =(
   const CComCurrency & curSrc
) throw( );
const CComCurrency & operator =(
   CURRENCY cySrc
) throw( );
const CComCurrency & operator =(
   FLOAT fSrc
);
const CComCurrency & operator =(
   SHORT sSrc
);
const CComCurrency & operator =(
   LONG lSrc
);
const CComCurrency & operator =(
   BYTE bSrc
);
const CComCurrency & operator =(
   USHORT usSrc
);
const CComCurrency & operator =(
   DOUBLE dSrc
);
const CComCurrency & operator =(
   CHAR cSrc
);
const CComCurrency & operator =(
   ULONG ulSrc
);
const CComCurrency & operator =(
   DECIMAL dSrc
);
```

Parameters

curSrc

A **CComCurrency** object.

cySrc

A variable of type **CURRENCY**.

sSrc, *fSrc*, *lSrc*, *bSrc*, *usSrc*, *dSrc*, *cSrc*, *ulSrc*, *dSrc*

The numeric value to assign to the **CComCurrency** object.

Return Value

Returns the updated **CComCurrency** object. In the event of an error, such as an overflow, this operator calls **AtlThrow** with an HRESULT describing the error.

Example

```
// CComCurrency::operator = Example
   CComCurrency cur1, cur2(10,5000);
   CURRENCY cy;

   // Copying one object to another
   cur1 = cur2;

   // Using the CURRENCY data type
```

```
    cy.int64 = 105000;
    cur1 = cy;

    ATLASSERT(cur1 == cur2);
```

See Also

Reference

CComCurrency Class

Other Resources

CComCurrency Members

CComCurrency::operator -=

This operator is used to perform subtraction on a **CComCurrency** object and assign it the result.

```
const CComCurrency & operator -=(
   const CComCurrency & cur
);
```

Parameters

cur

A **CComCurrency** object.

Return Value

Returns the updated **CComCurrency** object. In the event of an error, such as an overflow, this operator calls **AtlThrow** with an HRESULT describing the error.

Example

```
// CComCurrency::operator -= Example
   CComCurrency cur(10,5000);
   cur-=CComCurrency(4,5000);
   ATLASSERT(cur == CComCurrency(6,0));
```

See Also

Reference

CComCurrency Class
CComCurrency::operator -
CComCurrency::operator +
CComCurrency::operator *
CComCurrency::operator /

Other Resources

CComCurrency Members

CComCurrency::operator ==

This operator compares two **CComCurrency** objects for equality.

```
bool operator ==(
   const CComCurrency & cur
) const;
```

Parameters

cur

The **CComCurrency** object to compare.

Return Value

Returns **true** if the objects are equal (that is, the **m_currency** data members, both integer and fractional, in both objects have the same value), **false** otherwise.

Example

```
// CComCurrency::operator == Example
   CComCurrency cur1(10,5000), cur2;
   cur2 = cur1;
   ATLASSERT(cur1 == cur2);
```

See Also

Reference

CComCurrency Class

CComCurrency::operator !=

Other Resources

CComCurrency Members

CComCurrency::operator >

This operator compares two **CComCurrency** objects to determine the larger.

```
bool operator >(
   const CComCurrency & cur
) const;
```

Parameters

cur

A **CComCurrency** object.

Return Value

Returns **true** if the first object is greater than the second, **false** otherwise.

Example

```
// CComCurrency::operator > Example
   CComCurrency cur1(10,5100);
   CComCurrency cur2(10,5000);
   ATLASSERT(cur1 > cur2);
```

See Also

Reference

CComCurrency Class
CComCurrency::operator >=
CComCurrency::operator <
CComCurrency::operator <=

Other Resources

CComCurrency Members

CComCurrency::operator >=

This operator compares two **CComCurrency** objects to determine equality or the larger.

```
bool operator >=(
   const CComCurrency & cur
) const;
```

Parameters

cur

A **CComCurrency** object.

Return Value

Returns **true** if the first object is greater than or equal to the second, **false** otherwise.

Example

```
// CComCurrency::operator >= Example
   CComCurrency cur1(10,5100);
   CComCurrency cur2(10,5000);
   ATLASSERT(cur1 >= cur2);
```

See Also

Reference

CComCurrency Class
CComCurrency::operator >
CComCurrency::operator <
CComCurrency::operator <=

Other Resources

CComCurrency Members

CComCurrency::operator CURRENCY

These operators are used to cast a **CComCurrency** object to a **CURRENCY** data type.

```
operator CURRENCY&( ) throw( );
operator const CURRENCY&( ) const throw( );
```

Return Value

Returns a reference to a **CURRENCY** object.

Example

```
// CComCurrency::operator CURRENCY Example
   CComCurrency cur(10,5000);
   CURRENCY cy = static_cast<CURRENCY>(cur); // Note that explicit cast is not necessary
   ATLASSERT(cy.int64 == 105000);
```

See Also

Reference

CComCurrency Class

Other Resources

CComCurrency Members

CComCurrency Data Members

For information about the data members in **CComCurrency**, see CComCurrency Members.

CComCurrency::m_currency

The **CURRENCY** data member.

```
CURRENCY m_currency;
```

Remarks

This member holds the currency accessed and manipulated by the methods of this class.

See Also

Reference

CComCurrency Class

CComCurrency::GetCurrencyPtr

Other Resources

CComCurrency Members

CComDynamicUnkArray Class

This class stores an array of **IUnknown** pointers.

```
class CComDynamicUnkArray
```

Remarks

CComDynamicUnkArray holds a dynamically allocated array of **IUnknown** pointers, each an interface on a connection point. **CComDynamicUnkArray** can be used as a parameter to the IConnectionPointImpl template class.

The **CComDynamicUnkArray** methods begin and end can be used to loop through all connection points (for example, when an event is fired).

See Adding Connection Points to an Object for details on automating creation of connection point proxies.

Note The class **CComDynamicUnkArray** is used by the **Add Class** wizard when creating a control which has Connection Points. If you wish to specify the number of Connection Points manually, change the reference from **CComDynamicUnkArray** to `CComUnkArray< n >`, where *n* is the number of connection points required.

Requirements

Header: atlcom.h

See Also

Reference

CComUnkArray Class

Other Resources

CComDynamicUnkArray Members

ATL Class Overview

CComDynamicUnkArray Members

Methods

Add	Call this method to add an **IUnknown** pointer to the array.
begin	Returns a pointer to the first **IUnknown** pointer in the collection.
clear	Empties the array.
CComDynamicUnkArray	Constructor. Initializes the collection values to **NULL** and the collection size to zero.
~CComDynamicUnkArray	The destructor.
end	Returns a pointer to one past the last **IUnknown** pointer in the collection.
GetAt	Retrieves the element at the specified index.
GetCookie	Call this method to get the cookie associated with a given **IUnknown** pointer.
GetSize	Returns the length of an array.
GetUnknown	Call this method to get the **IUnknown** pointer associated with a given cookie.
Remove	Call this method to remove an **IUnknown** pointer from the array.

See Also

Reference

CComDynamicUnkArray Class

CComDynamicUnkArray Methods

For information about the methods in **CComDynamicUnkArray**, see CComDynamicUnkArray Members.

CComDynamicUnkArray::Add

Call this method to add an **IUnknown** pointer to the array.

```
DWORD Add(
   IUnknown* pUnk
);
```

Parameters

pUnk

The **IUnknown** pointer to add to the array.

Return Value

Returns the cookie associated with the newly added pointer.

See Also

Reference

CComDynamicUnkArray Class

CComDynamicUnkArray::Remove

Other Resources

CComDynamicUnkArray Methods

CComDynamicUnkArray::begin

Returns a pointer to the beginning of the collection of **IUnknown** interface pointers.

```
IUnknown** begin( );
```

Return Value

A pointer to an **IUnknown** interface pointer.

Remarks

The collection contains pointers to interfaces stored locally as **IUnknown**. You cast each **IUnknown** interface to the real interface type and then call through it. You do not need to query for the interface first.

Before using the **IUnknown** interface, you should check that it is not **NULL**.

See Also

Reference

CComDynamicUnkArray Class
CComDynamicUnkArray::end
CComUnkArray::begin

Other Resources

CComDynamicUnkArray Members

CComDynamicUnkArray::clear

Empties the array.

```
void clear( );
```

See Also

Reference

CComDynamicUnkArray Class

Other Resources

CComDynamicUnkArray Members

CComDynamicUnkArray::CComDynamicUnkArray

The constructor.

```
CComDynamicUnkArray( );
```

Remarks

Sets the collection size to zero and initializes the values to **NULL**. The destructor frees the collection, if necessary.

See Also

Reference

CComDynamicUnkArray Class

Other Resources

CComDynamicUnkArray Members

CComDynamicUnkArray::~CComDynamicUnkArray

The destructor.

```
~CComDynamicUnkArray( );
```

Remarks

Frees resources allocated by the class constructor.

See Also

Reference

CComDynamicUnkArray Class

Other Resources

CComDynamicUnkArray Methods

CComDynamicUnkArray::end

Returns a pointer to one past the last **IUnknown** pointer in the collection.

```
IUnknown** end( );
```

Return Value

A pointer to an **IUnknown** interface pointer.

See Also

Reference

CComDynamicUnkArray Class
CComDynamicUnkArray::begin
CComUnkArray::end

Other Resources

CComDynamicUnkArray Members

CComDynamicUnkArray::GetAt

Retrieves the element at the specified index.

```
IUnknown* GetAt(
   int nIndex
);
```

Parameters

nIndex

The index of the element to retrieve.

Return Value

A pointer to an IUnknown interface.

See Also

Reference

CComDynamicUnkArray Class

Other Resources

CComDynamicUnkArray Members

CComDynamicUnkArray::GetCookie

Call this method to get the cookie associated with a given **IUnknown** pointer.

```
DWORD WINAPI GetCookie(
   IUnknown** ppFind
);
```

Parameters

ppFind

The **IUnknown** pointer for which the associated cookie is required.

Return Value

Returns the cookie associated with the **IUnknown** pointer, or zero if no matching **IUnknown** pointer is found.

Remarks

If there is more than one instance of the same **IUnknown** pointer, this function returns the cookie for the first one.

See Also

Reference

CComDynamicUnkArray Class
CComDynamicUnkArray::Add
CComDynamicUnkArray::Remove
CComDynamicUnkArray::GetUnknown

Other Resources

CComDynamicUnkArray Methods

CComDynamicUnkArray::GetSize

Returns the length of an array.

```
int GetSize( ) const;
```

Return Value

The length of the array.

See Also

Reference

CComDynamicUnkArray Class

Other Resources

CComDynamicUnkArray Members

CComDynamicUnkArray::GetUnknown

Call this method to get the **IUnknown** pointer associated with a given cookie.

```
IUnknown* WINAPI GetUnknown(
   DWORD dwCookie
);
```

Parameters

dwCookie

The cookie for which the associated **IUnknown** pointer is required.

Return Value

Returns the **IUnknown** pointer, or NULL if no matching cookie is found.

See Also

Reference

CComDynamicUnkArray Class

CComDynamicUnkArray::GetCookie

Other Resources

CComDynamicUnkArray Methods

CComDynamicUnkArray::Remove

Call this method to remove an **IUnknown** pointer from the array.

```
BOOL Remove(
   DWORD dwCookie
);
```

Parameters

dwCookie

The cookie referencing the **IUnknown** pointer to be removed from the array.

Return Value

Returns TRUE if the pointer is removed; otherwise FALSE.

See Also

Reference

CComDynamicUnkArray Class
CComDynamicUnkArray::Add
CComDynamicUnkArray::GetCookie

Other Resources

CComDynamicUnkArray Methods

CComEnum Class

This class defines a COM enumerator object based on an array.

```
template <
   class Base,
   const IID* piid,
   class T,
   class Copy,
   class ThreadModel = CcomObjectThreadModel
>
class ATL_NO_VTABLE CComEnum :
      public CComEnumImpl<Base, piid, T, Copy>,
      public CComObjectRootEx< ThreadModel >
```

Parameters

Base

A COM enumerator (IEnumXXXX) interface.

piid

A pointer to the interface ID of the enumerator interface.

T

The type of item exposed by the enumerator interface.

Copy

A homogeneous copy policy class.

ThreadModel

The threading model of the class. This parameter defaults to the global object thread model used in your project.

Remarks

CComEnum defines a COM enumerator object based on an array. This class is analogous to CComEnumOnSTL which implements an enumerator based on an STL container. Typical steps for using this class are outlined below. For more information, see ATL Collections and Enumerators.

To use this class:

- **typedef** a specialization of this class.
- Use the **typedef** as the template argument in a specialization of **CComObject**.
- Create an instance of the **CComObject** specialization.
- Initialize the enumerator object by calling CComEnumImpl::Init.
- Return the enumerator interface to the client.

Requirements

Header: atlcom.h

Example

The code shown below provides a reusable function for creating and initializing an enumerator object.

```
     template <class EnumType, class ElementType>
     HRESULT CreateEnumerator(IUnknown** ppUnk,
                              ElementType* begin, ElementType* end,
                              IUnknown* pUnk,
                              CComEnumFlags flags)
     {
```

```
        if (ppUnk == NULL)
            return E_POINTER;
        *ppUnk = NULL;

        CComObject<EnumType>* pEnum = NULL;
        HRESULT hr = CComObject<EnumType>::CreateInstance(&pEnum);

        if (FAILED(hr))
            return hr;

        hr = pEnum->Init(begin, end, pUnk, flags);

        if (SUCCEEDED(hr))
            hr = pEnum->QueryInterface(ppUnk);

        if (FAILED(hr))
            delete pEnum;

        return hr;

    } // CreateEnumerator
```

This template function can be used to implement the **_NewEnum** property of a collection interface as shown below:

```
typedef CComEnum<IEnumVARIANT, &IID_IEnumVARIANT, VARIANT,
                             _Copy<VARIANT> > VarArrEnum;

class ATL_NO_VTABLE CVariantArrayCollection :
    // ... code omitted
{
    // ... code omitted

    VARIANT m_arr[3];
public:
    STDMETHOD(get__NewEnum)(IUnknown** ppUnk)
    {
        return CreateEnumerator<VarArrEnum>(ppUnk, &m_arr[0], &m_arr[3],
                                            this, AtlFlagNoCopy);
    }
};
```

This code creates a **typedef** for **CComEnum** that exposes a vector of **VARIANT**s through the **IEnumVariant** interface. The **CVariantArrayCollection** class simply specializes **CreateEnumerator** to work with enumerator objects of this type and passes the necessary arguments.

See Also

Reference

CComObjectThreadModel
CComEnumImpl Class
CComObjectRootEx Class

Other Resources

CComEnum Members
ATL Class Overview

CComEnum Members

These members are inherited from the CComEnumImpl base class.

See Also

Reference

CComEnum Class

CComEnumImpl Class

This class provides the implementation for a COM enumerator interface where the items being enumerated are stored in an array.

```
template <
   class Base,
   const IID* piid,
   class T,
   class Copy
>
class ATL_NO_VTABLE CComEnumImpl :
   public Base
```

Parameters

Base

A COM enumerator (IEnumXXXX) interface.

piid

A pointer to the interface ID of the enumerator interface.

T

The type of item exposed by the enumerator interface.

Copy

A homogeneous copy policy class.

Remarks

CComEnumImpl provides the implementation for a COM enumerator interface where the items being enumerated are stored in an array. This class is analogous to the **IEnumOnSTLImpl** class, which provides an implementation of an enumerator interface based on an STL container.

Note
For details on further differences between **CComEnumImpl** and **IEnumOnSTLImpl**, see CComEnumImpl::Init.

Typically, you will *not* need to create your own enumerator class by deriving from this interface implementation. If you want to use an ATL-supplied enumerator based on an array, it is more common to create an instance of CComEnum.

However, if you do need to provide a custom enumerator (for example, one that exposes interfaces in addition to the enumerator interface), you can derive from this class. In this situation, it is likely that you'll need to override the CComEnumImpl::Clone method to provide your own implementation.

For more information, see ATL Collections and Enumerators.

Requirements

Header: atlcom.h

See Also

Reference

IEnumOnSTLImpl Class

CComEnum Class

Other Resources

CComEnumImpl Members

ATL Class Overview

CComEnumImpl Members

Methods

CComEnumImpl	The constructor.
~CComEnumImpl	The destructor.
Clone	The implementation of IEnumXXXX::Clone.
Init	Initializes the enumerator.
Next	The implementation of IEnumXXXX::Next.
Reset	The implementation of IEnumXXXX::Reset.
Skip	The implementation of IEnumXXXX::Skip.

Data Members

m_begin	A pointer to the first item in the array.
m_dwFlags	Copy flags passed through **Init**.
m_end	A pointer to the location just beyond the last item in the array.
m_iter	A pointer to the current item in the array.
m_spUnk	The **IUnknown** pointer of the object supplying the collection being enumerated.

See Also

Reference

CComEnumImpl Class

CComEnumImpl Methods

For information about the methods in **CComEnumImpl**, see CComEnumImpl Members.

CComEnumImpl::CComEnumImpl

The constructor.

```
CComEnumImpl( );
```

See Also

Reference

CComEnumImpl Class

Other Resources

CComEnumImpl Members

CComEnumImpl::~CComEnumImpl

The destructor.

```
~CComEnumImpl( );
```

See Also

Reference

CComEnumImpl Class

Other Resources

CComEnumImpl Members

CComEnumImpl::Init

You must call this method before passing a pointer to the enumerator interface back to any clients.

```
HRESULT Init(
   T* begin,
   T* end,
   IUnknown* pUnk,
   CComEnumFlags flags = AtlFlagNoCopy
);
```

Parameters

begin

A pointer to the first element of the array containing the items to be enumerated.

end

A pointer to the location just beyond the last element of the array containing the items to be enumerated.

pUnk

[in] The **IUnknown** pointer of an object that must be kept alive during the lifetime of the enumerator. Pass **NULL** if no such object exists.

flags

Flags specifying whether or not the enumerator should take ownership of the array or make a copy of it. Possible values are described below.

Return Value

A standard **HRESULT** value.

Remarks

Only call this method once — initialize the enumerator, use it, then throw it away.

If you pass pointers to items in an array held in another object (and you don't ask the enumerator to copy the data), you can use the *pUnk* parameter to ensure that the object and the array it holds are available for as long as the enumerator needs them. The enumerator simply holds a COM reference on the object to keep it alive. The COM reference is automatically released when the enumerator is destroyed.

The *flags* parameter allows you to specify how the enumerator should treat the array elements passed to it. *flags* can take one of the values from the **CComEnumFlags** enumeration shown below:

```
enum CComEnumFlags
{
    AtlFlagNoCopy = 0,
    AtlFlagTakeOwnership = 2, // BitOwn
    AtlFlagCopy = 3           // BitOwn | BitCopy
};
```

AtlFlagNoCopy means that the array's lifetime is not controlled by the enumerator. In this case, either the array will be static or the object identified by *pUnk* will be responsible for freeing the array when it's no longer needed.

AtlFlagTakeOwnership means that the destruction of the array is to be controlled by the enumerator. In this case, the array must have been dynamically allocated using **new**. The enumerator will delete the array in its destructor. Typically, you would pass **NULL** for *pUnk*, although you can still pass a valid pointer if you need to be notified of the destruction of the enumerator for some reason.

AtlFlagCopy means that a new array is to be created by copying the array passed to **Init**. The new array's lifetime is to be controlled by the enumerator. The enumerator will delete the array in its destructor. Typically, you would pass **NULL** for *pUnk*, although you can still pass a valid pointer if you need to be notified of the destruction of the enumerator for some reason.

Note

The prototype of this method specifies the array elements as being of type **T**, where **T** was defined as a template parameter to the class. This is the same type that is exposed by means of the COM interface method CComEnumImpl::Next. The implication of this is that, unlike IEnumOnSTLImpl, this class does not support different storage and exposed data types. The data type of elements in the array must be the same as the data type exposed by means of the COM interface.

See Also

Reference

CComEnumImpl Class

IEnumOnSTLImpl::Init

Other Resources

CComEnumImpl Members

CComEnumImpl::Clone

This method provides the implementation of the IEnumXXXX::Clone method by creating an object of type **CComEnum**, initializing it with the same array and iterator used by the current object, and returning the interface on the newly created object.

```
STDMETHOD(Clone)(
   Base** ppEnum
);
```

Parameters

ppEnum

[out] The enumerator interface on a newly created object cloned from the current enumerator.

Return Value

A standard **HRESULT** value.

Remarks

Note that cloned enumerators never make their own copy (or take ownership) of the data used by the original enumerator. If necessary, cloned enumerators will keep the original enumerator alive (using a COM reference) to ensure that the data is available for as long as they need it.

See Also

Reference

CComEnumImpl Class

CComEnumImpl::Init

Other Resources

CComEnumImpl Members

CComEnumImpl::Next

This method provides the implementation of the IEnumXXXX::Next method.

```
STDMETHOD(Next)(
   ULONG celt,
   T* rgelt,
   ULONG* pceltFetched
);
```

Parameters

celt

[in] The number of elements requested.

rgelt

[out] The array to be filled with the elements.

pceltFetched

[out] The number of elements actually returned in *rgelt*. This can be less than *celt* if fewer than *celt* elements remained in the list.

Return Value

A standard **HRESULT** value.

See Also

Reference

CComEnumImpl Class

Other Resources

CComEnumImpl Members

CComEnumImpl::Reset

This method provides the implementation of the IEnumXXXX::Reset method.

```
STDMETHOD(Reset)(
   void
);
```

Return Value

A standard **HRESULT** value.

See Also

Reference

CComEnumImpl Class

Other Resources

CComEnumImpl Members

CComEnumImpl::Skip

This method provides the implementation of the IEnumXXXX::Skip method.

```
STDMETHOD(Skip)(
   ULONG celt
);
```

Parameters

celt

[in] The number of elements to skip.

Return Value

A standard **HRESULT** value.

Remarks

Returns E_INVALIDARG if *celt* is zero, returns S_FALSE if less than *celt* elements are returned, returns S_OK otherwise.

See Also

Reference

CComEnumImpl Class

Other Resources

CComEnumImpl Members

CComEnumImpl Data Members

For information about the data members in **CComEnumImpl**, see CComEnumImpl Members.

CComEnumImpl::m_spUnk

This smart pointer maintains a reference on the object passed to CComEnumImpl::Init, ensuring that it remains alive during the lifetime of the enumerator.

```
CComPtr<IUnknown> m_spUnk;
```

See Also

Reference

CComEnumImpl Class

CComEnumImpl::Init

Other Resources

CComEnumImpl Members

CComEnumImpl::m_begin

A pointer to the location just beyond the last element of the array containing the items to be enumerated.

```
T* m_begin;
```

See Also

Reference

CComEnumImpl Class

CComEnumImpl::Init

Other Resources

CComEnumImpl Members

CComEnumImpl::m_end

A pointer to the first element of the array containing the items to be enumerated.

```
T* m_end;
```

See Also

Reference

CComEnumImpl Class

CComEnumImpl::Init

Other Resources

CComEnumImpl Members

CComEnumImpl::m_iter

A pointer to the current element of the array containing the items to be enumerated.

```
T* m_iter;
```

See Also

Reference

CComEnumImpl Class

Other Resources

CComEnumImpl Members

CComEnumImpl::m_dwFlags

The flags passed to CComEnumImpl::Init.

```
DWORD m_dwFlags;
```

See Also

Reference

CComEnumImpl Class

CComEnumImpl::Init

Other Resources

CComEnumImpl Members

CComEnumOnSTL Class

This class defines a COM enumerator object based on an STL collection.

```
template <
   class Base,
   const IID* piid,
   class T,
   class Copy,
   class CollType,
   class ThreadModel = CComObjectThreadModel
>
class ATL_NO_VTABLE CComEnumOnSTL :
   public IEnumOnSTLImpl<Base, piid, T, Copy, CollType>,
   public CComObjectRootEx< ThreadModel >
```

Parameters

Base

A COM enumerator (IEnumXXXX) interface.

piid

A pointer to the interface ID of the enumerator interface.

T

The type of item exposed by the enumerator interface.

Copy

A copy policy class.

CollType

An STL container class.

Remarks

CComEnumOnSTL defines a COM enumerator object based on an STL collection. This class can be used on its own or in conjunction with ICollectionOnSTLImpl. Typical steps for using this class are outlined below. For more information, see ATL Collections and Enumerators.

To use this class with ICollectionOnSTLImpl:

- **typedef** a specialization of this class.
- Use the **typedef** as the final template argument in a specialization of **ICollectionOnSTLImpl**.

See ATL Collections and Enumerators for an example.

To use this class independently of ICollectionOnSTLImpl:

- **typedef** a specialization of this class.
- Use the **typedef** as the template argument in a specialization of **CComObject**.
- Create an instance of the **CComObject** specialization.
- Initialize the enumerator object by calling IEnumOnSTLImpl::Init.
- Return the enumerator interface to the client.

Requirements

Header: atlcom.h

Example

The code shown below provides a generic function to handle the creation and initialization of an enumerator object:

```
    template <class EnumType, class CollType>
    HRESULT CreateSTLEnumerator(IUnknown** ppUnk, IUnknown* pUnkForRelease, CollType& colle
ction)
    {
        if (ppUnk == NULL)
            return E_POINTER;
        *ppUnk = NULL;

        CComObject<EnumType>* pEnum = NULL;
        HRESULT hr = CComObject<EnumType>::CreateInstance(&pEnum);

        if (FAILED(hr))
            return hr;

        hr = pEnum->Init(pUnkForRelease, collection);

        if (SUCCEEDED(hr))
            hr = pEnum->QueryInterface(ppUnk);

        if (FAILED(hr))
            delete pEnum;

        return hr;

    } // CreateSTLEnumerator
```

This template function can be used to implement the **_NewEnum** property of a collection interface as shown below:

```
typedef CComEnumOnSTL<IEnumVARIANT, &IID_IEnumVARIANT, VARIANT,
                  _Copy<VARIANT>, std::vector<CComVariant> > VarVarEnum;

class ATL_NO_VTABLE CVariantCollection :
    // ... code omitted
{
    // ... code omitted
    std::vector<CComVariant> m_vec;

    STDMETHOD(get__NewEnum)(IUnknown** ppUnk)
    {
        return CreateSTLEnumerator<VarVarEnum>(ppUnk, this, m_vec);
   }
};
```

This code creates a **typedef** for **CComEnumOnSTL** that exposes a vector of **CComVariant**s by means of the **IEnumVariant** interface. The **CVariantCollection** class simply specializes **CreateSTLEnumerator** to work with enumerator objects of this type.

See Also

Tasks

ATLCollections Sample: Demonstrates ICollectionOnSTLImpl, CComEnumOnSTL, and Custom Copy Policy Classes

Reference

CComObjectRootEx Class

CComObjectThreadModel

IEnumOnSTLImpl Class

Other Resources

CComEnumOnSTL Members

ATL Class Overview

CComEnumOnSTL Members

These members are inherited from the IEnumOnSTLImpl base class.

See Also

Reference

CComEnumOnSTL Class

CComFakeCriticalSection Class

This class provides the same methods as CComCriticalSection but does not provide a critical section.

```
class CComFakeCriticalSection
```

Remarks

CComFakeCriticalSection mirrors the methods found in CComCriticalSection. However, **CComFakeCriticalSection** does not provide a critical section; therefore, its methods do nothing.

Typically, you use **CComFakeCriticalSection** through a **typedef** name, either **AutoCriticalSection** or **CriticalSection**. When using CComSingleThreadModel or CComMultiThreadModelNoCS, both of these **typedef** names reference **CComFakeCriticalSection**. When using CComMultiThreadModel, they reference CComAutoCriticalSection and **CComCriticalSection**, respectively.

Requirements

Header: atlcore.h

See Also

Other Resources

CComFakeCriticalSection Members

ATL Class Overview

CComFakeCriticalSection Members

Methods

Init	Does nothing since there is no critical section.
Lock	Does nothing since there is no critical section.
Term	Does nothing since there is no critical section.
Unlock	Does nothing since there is no critical section.

See Also

Reference

CComFakeCriticalSection Class

CComFakeCriticalSection Methods

For information about the methods in **CComFakeCriticalSection**, see CComFakeCriticalSection Members.

CComFakeCriticalSection::Init

Does nothing since there is no critical section.

```
HRESULT Init( ) throw( );
```

Return Value

Returns S_OK.

See Also

Reference

CComFakeCriticalSection Class

CComCriticalSection::Init

Other Resources

CComFakeCriticalSection Members

CComFakeCriticalSection::Lock

Does nothing since there is no critical section.

```
HRESULT Lock( ) throw( );
```

Return Value

Returns S_OK.

See Also

Reference

CComFakeCriticalSection Class

CComCriticalSection::Lock

Other Resources

CComFakeCriticalSection Members

CComFakeCriticalSection::Term

Does nothing since there is no critical section.

```
HRESULT Term( ) throw( );
```

Return Value

Returns S_OK.

See Also

Reference

CComFakeCriticalSection Class

CComCriticalSection::Term

Other Resources

CComFakeCriticalSection Members

CComFakeCriticalSection::Unlock

Does nothing since there is no critical section.

```
HRESULT Unlock( ) throw( );
```

Return Value

Returns S_OK.

See Also

Reference

CComFakeCriticalSection Class

CComCriticalSection::Unlock

Other Resources

CComFakeCriticalSection Members

CComGITPtr Class

This class provides methods for dealing with interface pointers and the global interface table (GIT).

```
template <
   class T
>
class CComGITPtr
```

Parameters

T

The type of the interface pointer to be stored in the GIT.

Remarks

Objects that aggregate the free threaded marshaler and need to use interface pointers obtained from other objects must take extra steps to ensure that the interfaces are correctly marshaled. Typically this involves storing the interface pointers in the GIT and getting the pointer from the GIT each time it is used. The class **CComGITPtr** is provided to help you use interface pointers stored in the GIT.

Note
The global interface table facility is only available on Windows 95 with DCOM version 1.1 and later, Windows 98, Windows NT 4.0 with Service Pack 3 and later, and Windows 2000.

Requirements

Header: atlbase.h

See Also

Reference

Accessing Interfaces Across Apartments

When to Use the Global Interface Table

Concepts

ATL and the Free Threaded Marshaler

Other Resources

CComGITPtr Members

ATL Class Overview

CComGITPtr Members

Methods

Attach	Call this method to register the interface pointer in the global interface table (GIT).
CComGITPtr	The constructor.
~CComGITPtr	The destructor.
CopyTo	Call this method to copy the interface from the global interface table (GIT) to the passed pointer.
Detach	Call this method to disassociate the interface from the **CComGITPtr** object.
GetCookie	Call this method to return the cookie from the **CComGITPtr** object.
Revoke	Call this method to remove the interface from the global interface table (GIT).

Operators

operator =	Assignment operator.
operator DWORD	Returns the cookie from the **CComGITPtr** object.

Data Members

m_dwCookie	The cookie.

See Also

Reference

CComGITPtr Class

CComGITPtr Methods

For information about the methods in **CComGITPtr**, see CComGITPtr Members.

CComGITPtr::Attach

Call this method to register the interface pointer in the global interface table (GIT).

```
HRESULT Attach(
   T* p
) throw( );
HRESULT Attach(
   DWORD dwCookie
) throw( );
```

Parameters

p

The interface pointer to be added to the GIT.

dwCookie

The cookie used to identify the interface pointer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

In debug builds, an assertion error will occur if the GIT is not valid, or if the cookie is equal to NULL.

See Also

Reference

CComGITPtr Class

CComGITPtr::Detach

Other Resources

CComGITPtr Members

CComGITPtr::CComGITPtr

The constructor.

```
CComGITPtr( ) throw( );
CComGITPtr(
   T* p
);
CComGITPtr(
   const CComGITPtr& git
);
explicit CComGITPtr(
   DWORD dwCookie
) throw( );
```

Parameters

p

An interface pointer to be stored in the global interface table (GIT).

git

A reference to an existing **CComGITPtr** object.

dwCookie

A cookie used to identify the interface pointer.

Remarks

Creates a new **CComGITPtr** object, optionally using an existing **CComGITPtr** object.

See Also

Reference

CComGITPtr Class

Other Resources

CComGITPtr Members

CComGITPtr::~CComGITPtr

The destructor.

```
~CComGITPtr( ) throw( );
```

Remarks

Removes the interface from the global interface table (GIT), using CComGITPtr::Revoke.

See Also

Reference

CComGITPtr Class

Other Resources

CComGITPtr Members

CComGITPtr::CopyTo

Call this method to copy the interface from the global interface table (GIT) to the passed pointer.

```
HRESULT CopyTo(
   T** pp
) const throw( );
```

Parameters

pp

The pointer which is to receive the interface.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The interface from the GIT is copied to the passed pointer. The pointer must be released by the caller when it is no longer required.

See Also

Reference

CComGITPtr Class

Other Resources

CComGITPtr Members

CComGITPtr::Detach

Call this method to disassociate the interface from the **CComGITPtr** object.

```
DWORD Detach( ) throw( );
```

Return Value

Returns the cookie from the **CComGITPtr** object.

Remarks

It is up to the caller to remove the interface from the GIT, using CComGITPtr::Revoke.

See Also

Reference

CComGITPtr Class

CComGITPtr::Attach

Other Resources

CComGITPtr Members

CComGITPtr::GetCookie

Call this method to return the cookie from the **CComGITPtr** object.

```
DWORD GetCookie( ) const;
```

Return Value

Returns the cookie.

Remarks

The cookie is a variable used to identify an interface and its location.

See Also

Reference

CComGITPtr Class

Other Resources

CComGITPtr Members

CComGITPtr::Revoke

Call this method to remove the current interface from the global interface table (GIT).

```
HRESULT Revoke( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Removes the interface from the GIT.

See Also

Reference

CComGITPtr Class

Other Resources

CComGITPtr Members

CComGITPtr Operators

For information about the operators in **CComGITPtr**, see CComGITPtr Members.

CComGITPtr::operator =

The assignment operator.

```
CComGITPtr< T >& operator =(
   T* p
);
CComGITPtr< T >& operator =(
   const CComGITPtr< T >& git
);
CComGITPtr< T >& operator =(
   DWORD dwCookie
);
```

Parameters

p

A pointer to an interface.

git

A reference to a **CComGITPtr** object.

dwCookie

A cookie used to identify the interface pointer.

Return Value

Returns the updated **CComGITPtr** object.

Remarks

Assigns a new value to a **CComGITPtr** object, either from an existing object or from a reference to a global interface table.

See Also

Reference

CComGITPtr Class

Other Resources

CComGITPtr Members

CComGITPtr::operator DWORD

Returns the cookie associated with the **CComGITPtr** object.

```
operator DWORD( ) const;
```

Remarks

The cookie is a variable used to identify an interface and its location.

See Also

Reference

CComGITPtr Class

Other Resources

CComGITPtr Members

CComGITPtr Data Members

For information about the data members in **CComGITPtr**, see CComGITPtr Members.

CComGITPtr::m_dwCookie

The cookie.

```
DWORD m_dwCookie;
```

Remarks

The cookie is a member variable used to identify an interface and its location.

See Also

Reference

CComGITPtr Class

Other Resources

CComGITPtr Members

CComHeap Class

This class implements IAtlMemMgr using the COM memory allocation functions.

```
class CComHeap : public IAtlMemMgr
```

Remarks

CComHeap implements memory allocation functions using the COM allocation functions, including CoTaskMemAlloc, CoTaskMemFree, IMalloc::GetSize, and CoTaskMemRealloc. The maximum amount of memory that can be allocated is equal to **INT_MAX** (2147483647) bytes.

Example

See the example for IAtlMemMgr.

Requirements

Header: ATLComMem.h

See Also

Tasks

DynamicConsumer Sample: Uses Dynamic Accessor and Schema Rowset Classes to Read Metadata from a Database

Reference

CWin32Heap Class
CLocalHeap Class
CGlobalHeap Class
CCRTHeap Class
IAtlMemMgr Class

Other Resources

CComHeap Members
ATL Class Overview

CComHeap Members

Methods

Allocate	Call this method to allocate a block of memory.
Free	Call this method to free a block of memory allocated by this memory manager.
GetSize	Call this method to get the allocated size of a memory block allocated by this memory manager.
Reallocate	Call this method to reallocate memory allocated by this memory manager.

See Also

Reference

CComHeap Class

Other Resources

IAtlMemMgr Members

CComHeap Methods

For information about the methods in **CComHeap**, see CComHeap Members.

CComHeap::Allocate

Call this method to allocate a block of memory.

```
virtual void* Allocate(
   size_t nBytes
) throw( );
```

Parameters

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call CComHeap::Free or CComHeap::Reallocate to free the memory allocated by this method.

Implemented using CoTaskMemAlloc.

See Also

Reference

CComHeap Class
CComHeap::Free
CComHeap::Reallocate

Other Resources

CComHeap Members

CComHeap::Free

Call this method to free a block of memory allocated by this memory manager.

```
virtual void Free(
   void* p
) throw( );
```

Parameters

p

Pointer to memory previously allocated by this memory manager. NULL is a valid value and does nothing.

Remarks

Implemented using CoTaskMemFree.

See Also

Reference

CComHeap Class
CComHeap::Allocate
CComHeap::Reallocate

Other Resources

CComHeap Members

CComHeap::GetSize

Call this method to get the allocated size of a memory block allocated by this memory manager.

```
virtual size_t GetSize(
   void* p
) throw( );
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

Return Value

Returns the size of the allocated memory block in bytes.

Remarks

Implemented using IMalloc::GetSize.

See Also

Reference

CComHeap Class

Other Resources

CComHeap Members

CComHeap::Reallocate

Call this method to reallocate memory allocated by this memory manager.

```
virtual void* Reallocate(
   void* p,
   size_t nBytes
) throw( );
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call CComHeap::Free to free the memory allocated by this method.

Implemented using CoTaskMemRealloc.

See Also

Reference

CComHeap Class

CComHeap::Free

Other Resources

CComHeap Members

CComHeapPtr Class

A smart pointer class for managing heap pointers.

```
template<
   typename T
> class CComHeapPtr :
   public CHeapPtr< T, CComAllocator >
```

Parameters

T

The object type to be stored on the heap.

Remarks

CComHeapPtr derives from **CHeapPtr**, but uses CComAllocator to allocate memory using COM routines. See CHeapPtr and CHeapPtrBase for the methods available.

Requirements

Header: atlbase.h

See Also

Reference

CHeapPtr Class
CHeapPtrBase Class
CComAllocator Class

Other Resources

CComHeapPtr Members
ATL Class Overview

CComHeapPtr Members

Methods

CComHeapPtr	The constructor.

See Also

Reference

CComHeapPtr Class

Other Resources

CHeapPtr Members

CHeapPtrBase Members

CComHeapPtr Methods

For information about the methods in **CComHeapPtr**, see CComHeapPtr Members.

CComHeapPtr::CComHeapPtr

The constructor.

```
CComHeapPtr( ) throw( );
explicit CComHeapPtr(
   T* pData
) throw( );
```

Parameters

pData

An existing **CComHeapPtr** object.

Remarks

The heap pointer can optionally be created using an existing **CComHeapPtr** object. If so, the new **CComHeapPtr** object assumes responsibility for managing the new pointer and resources.

See Also

Reference

CComHeapPtr Class

Other Resources

CComHeapPtr Members

CComModule Class

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
class CComModule : public _ATL_MODULE
```

Remarks

Note

This class is obsolete, and the ATL code generation wizards now use the CAtlAutoThreadModule and CAtlModule derived classes. See ATL Module Classes for more information. The information that follows is for use with applications created with older releases of ATL.

CComModule implements a COM server module, allowing a client to access the module's components. **CComModule** supports both DLL (in-process) and EXE (local) modules.

A **CComModule** instance uses an object map to maintain a set of class object definitions. This object map is implemented as an array of **_ATL_OBJMAP_ENTRY** structures, and contains information for:

- Entering and removing object descriptions in the system registry.
- Instantiating objects through a class factory.
- Establishing communication between a client and the root object in the component.
- Performing lifetime management of class objects.

When you run the ATL COM AppWizard, the wizard automatically generates **_Module**, a global instance of **CComModule** or a class derived from it. For more information about the ATL Project Wizard, see the article Creating an ATL Project.

In addition to **CComModule**, ATL provides CComAutoThreadModule, which implements an apartment-model module for EXEs and Windows services. Derive your module from **CComAutoThreadModule** when you want to create objects in multiple apartments.

Requirements

Header: atlbase.h

See Also

Other Resources

CComModule Members

ATL Class Overview

CComModule Members

Methods

GetClassObject	Creates an object of a specified CLSID. For DLLs only.
GetModuleInstance	Returns **m_hInst**.
GetResourceInstance	Returns **m_hInstResource**.
GetTypeLibInstance	Returns **m_hInstTypeLib**.
Init	Initializes data members.
RegisterClassHelper	Enters an object's standard class registration in the system registry.
RegisterClassObjects	Registers the class object. For EXEs only.
RegisterServer	Updates the system registry for each object in the object map.
RegisterTypeLib	Registers a type library.
RevokeClassObjects	Revokes the class object. For EXEs only.
Term	Releases data members.
UnregisterClassHelper	Removes an object's standard class registration from the system registry.
UnregisterServer	Unregisters each object in the object map.
UpdateRegistryClass	Registers or unregisters an object's standard class registration.
UpdateRegistryFromResourceD	Runs the script contained in a specified resource to register or unregister an object.
UpdateRegistryFromResourceS	Statically links to the ATL Registry Component. Runs the script contained in a specified resour ce to register or unregister an object.

Data Members

m_csObjMap	Ensures synchronized access to the object map information.
m_csTypeInfoHolder	Ensures synchronized access to the type library information.
m_csWindowCreate	Ensures synchronized access to window class information and static data used during window creation.
m_hInst	Contains the handle to the module instance.
m_hInstResource	By default, contains the handle to the module instance.
m_hInstTypeLib	By default, contains the handle to the module instance.
m_pObjMap	Points to the object map maintained by the module instance.

See Also

Reference

CComModule Class

CComModule Methods

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

For information about the methods in **CComModule**, see CComModule Members.

See Also
Reference
CComModule Class

CComModule::GetClassObject

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
HRESULT GetClassObject(
   REFCLSID rclsid,
   REFIID riid,
   LPVOID* ppv
) throw( );
```

Parameters

rclsid

[in] The CLSID of the object to be created.

riid

[in] The IID of the requested interface.

ppv

[out] A pointer to the interface pointer identified by *riid*. If the object does not support this interface, *ppv* is set to **NULL**.

Return Value

A standard HRESULT value.

Remarks

Creates an object of the specified CLSID and retrieves an interface pointer to this object.

GetClassObject is only available to DLLs.

See Also

Reference

CComModule Class

Other Resources

CComModule Members

CComModule::GetModuleInstance

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
HINSTANCE GetModuleInstance( ) throw( );
```

Return Value

The **HINSTANCE** identifying this module.

Remarks

Returns the m_hInst data member.

See Also

Reference

CComModule Class
CComModule::GetResourceInstance
CComModule::GetTypeLibInstance

Other Resources

CComModule Members

CComModule::GetResourceInstance

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
HINSTANCE GetResourceInstance( ) throw( );
```

Return Value

An **HINSTANCE**.

Remarks

Returns the m_hInstResource data member.

See Also

Reference

CComModule Class
CComModule::GetModuleInstance
CComModule::GetTypeLibInstance

Other Resources

CComModule Members

CComModule::GetTypeLibInstance

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
HINSTANCE GetTypeLibInstance( ) const throw( );
```

Return Value

An **HINSTANCE**.

Remarks

Returns the m_hInstTypeLib data member.

See Also

Reference

CComModule Class
CComModule::GetModuleInstance
CComModule::GetResourceInstance

Other Resources

CComModule Members

CComModule::Init

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
HRESULT Init(
   _ATL_OBJMAP_ENTRY* p,
   HINSTANCE h,
   const GUID* plibid = NULL
) throw( );
```

Parameters

p

[in] A pointer to an array of object map entries.

h

[in] The **HINSTANCE** passed to **DLLMain** or **WinMain**.

plibid

[in] A pointer to the LIBID of the type library associated with the project.

Return Value

A standard HRESULT value.

Remarks

Initializes all data members.

See Also

Reference

CComModule Class

CComModule::Term

Other Resources

CComModule Members

CComModule::RegisterClassHelper

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
ATL_DEPRECATED HRESULT RegisterClassHelper(
   const CLSID& clsid,
   LPCTSTR lpszProgID,
   LPCTSTR lpszVerIndProgID,
   UINT nDescID,
   DWORD dwFlags
);
```

Parameters

clsid

[in] The CLSID of the object to be registered.

lpszProgID

[in] The ProgID associated with the object.

lpszVerIndProgID

[in] The version-independent ProgID associated with the object.

nDescID

[in] The identifier of a string resource for the object's description.

dwFlags

[in] Specifies the threading model to enter in the registry. Possible values are **THREADFLAGS_APARTMENT**, **THREADFLAGS_BOTH**, or **AUTPRXFLAG**.

Return Value

A standard HRESULT value.

Remarks

Enters an object's standard class registration in the system registry.

The UpdateRegistryClass method calls **RegisterClassHelper**.

See Also

Reference

CComModule Class

CComModule::UnregisterClassHelper

Other Resources

CComModule Members

CComModule::RegisterClassObjects

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
HRESULT RegisterClassObjects(
   DWORD dwClsContext,
   DWORD dwFlags
) throw( );
```

Parameters

dwClsContext

[in] Specifies the context in which the class object is to be run. Possible values are **CLSCTX_INPROC_SERVER**, **CLSCTX_INPROC_HANDLER**, or **CLSCTX_LOCAL_SERVER**. For a description of these values, see CLSCTX in the *Platform SDK*.

dwFlags

[in] Determines the connection types to the class object. Possible values are **REGCLS_SINGLEUSE**, **REGCLS_MULTIPLEUSE**, or **REGCLS_MULTI_SEPARATE**. For a description of these values, see REGCLS in the *Platform SDK*.

Return Value

A standard HRESULT value.

Remarks

Registers an EXE class object with OLE so other applications can connect to it. This method is only available to EXEs.

See Also

Reference

CComModule Class

CComModule::RevokeClassObjects

Other Resources

CComModule Members

CComModule::RegisterServer

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
HRESULT RegisterServer(
   BOOL bRegTypeLib = FALSE,
   const CLSID* pCLSID = NULL
) throw( );
```

Parameters

bRegTypeLib

[in] Indicates whether the type library will be registered. The default value is **FALSE**.

pCLSID

[in] Points to the CLSID of the object to be registered. If **NULL** (the default value), all objects in the object map will be registered.

Return Value

A standard HRESULT value.

Remarks

Depending on the *pCLSID* parameter, updates the system registry for a single class object or for all objects in the object map.

If *bRegTypeLib* is **TRUE**, the type library information will also be updated.

See OBJECT_ENTRY_AUTO for information on how to add an entry to the object map.

RegisterServer will be called automatically by **DLLRegisterServer** for a DLL or by **WinMain** for an EXE run with the **/RegServer** command line option.

See Also

Reference

CComModule Class

CComModule::UnregisterServer

Other Resources

CComModule Members

CComModule::RegisterTypeLib

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
HRESULT RegisterTypeLib( ) throw( );
HRESULT RegisterTypeLib(
   LPCTSTR lpszIndex
) throw( );
```

Parameters

lpszIndex

[in] String in the format `"\\N"`, where `N` is the integer index of the TYPELIB resource.

Return Value

A standard HRESULT value.

Remarks

Adds information about a type library to the system registry.

If the module instance contains multiple type libraries, use the second version of this method to specify which type library should be used.

See Also

Reference

CComModule Class

Other Resources

CComModule Members

CComModule::RevokeClassObjects

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
HRESULT RevokeClassObjects( ) throw( );
```

Return Value

A standard HRESULT value.

Remarks

Removes the class object. This method is only available to EXEs.

See Also

Reference

CComModule Class

CComModule::RegisterClassObjects

Other Resources

CComModule Members

CComModule::Term

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
void Term( ) throw( );
```

Remarks

Releases all data members.

See Also

Reference

CComModule Class

CComModule::Init

Other Resources

CComModule Members

CComModule::UnregisterClassHelper

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
ATL_DEPRECATED HRESULT UnregisterClassHelper(
   const CLSID& clsid,
   LPCTSTR lpszProgID,
   LPCTSTR lpszVerIndProgID
);
```

Parameters

clsid

[in] The CLSID of the object to be unregistered.

lpszProgID

[in] The ProgID associated with the object.

lpszVerIndProgID

[in] The version-independent ProgID associated with the object.

Return Value

A standard HRESULT value.

Remarks

Removes an object's standard class registration from the system registry.

The UpdateRegistryClass method calls **UnregisterClassHelper**.

See Also

Reference

CComModule Class

CComModule::RegisterClassHelper

Other Resources

CComModule Members

CComModule::UnregisterServer

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
HRESULT UnregisterServer(
   const CLSID* pCLSID = NULL
) throw ( );
inline HRESULT UnregisterServer(
   BOOL bUnRegTypeLib,
   const CLSID* pCLSID = NULL
) throw ( );
```

Parameters

bUnRegTypeLib

If **TRUE**, the type library is also unregistered.

pCLSID

Points to the CLSID of the object to be unregistered. If **NULL** (the default value), all objects in the object map will be unregistered.

Return Value

A standard HRESULT value.

Remarks

Depending on the *pCLSID* parameter, unregisters either a single class object or all objects in the object map.

UnregisterServer will be called automatically by **DLLUnregisterServer** for a DLL or by **WinMain** for an EXE run with the **/UnregServer** command line option.

See OBJECT_ENTRY_AUTO for information on how to add an entry to the object map.

See Also

Reference

CComModule Class

CComModule::RegisterServer

Other Resources

CComModule Members

CComModule::UpdateRegistryClass

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
ATL_DEPRECATED HRESULT UpdateRegistryClass(
   const CLSID& clsid,
   LPCTSTR lpszProgID,
   LPCTSTR lpszVerIndProgID,
   UINT nDescID,
   DWORD dwFlags,
   BOOL bRegister
);
ATL_DEPRECATED HRESULT UpdateRegistryClass(
   const CLSID& clsid,
   LPCTSTR lpszProgID,
   LPCTSTR lpszVerIndProgID,
   LPCTSTR szDesc,
   DWORD dwFlags,
   BOOL bRegister
);
```

Parameters

clsid

The CLSID of the object to be registered or unregistered.

lpszProgID

The ProgID associated with the object.

lpszVerIndProgID

The version-independent ProgID associated with the object.

nDescID

The identifier of the string resource for the object's description.

szDesc

A string containing the object's description.

dwFlags

Specifies the threading model to enter in the registry. Possible values are **THREADFLAGS_APARTMENT**, **THREADFLAGS_BOTH**, or **AUTPRXFLAG**.

bRegister

Indicates whether the object should be registered.

Return Value

A standard HRESULT value.

Remarks

If *bRegister* is **TRUE**, this method enters the object's standard class registration in the system registry.

If *bRegister* is **FALSE**, it removes the object's registration.

Depending on the value of *bRegister*, **UpdateRegistryClass** calls either RegisterClassHelper or UnregisterClassHelper.

By specifying the DECLARE_REGISTRY macro, **UpdateRegistryClass** will be invoked automatically when your object map is processed.

See Also

Reference

CComModule Class

Other Resources

CComModule Members

CComModule::UpdateRegistryFromResourceD

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
virtual HRESULT UpdateRegistryFromResourceD(
   LPCTSTR lpszRes,
   BOOL bRegister,
   struct _ATL_REGMAP_ENTRY* pMapEntries = NULL
) throw( );
virtual HRESULT UpdateRegistryFromResourceD(
   UINT nResID,
   BOOL bRegister,
   struct _ATL_REGMAP_ENTRY* pMapEntries = NULL
) throw ( );
```

Parameters

lpszRes

[in] A resource name.

nResID

[in] A resource ID.

bRegister

[in] Indicates whether the object should be registered.

pMapEntries

[in] A pointer to the replacement map storing values associated with the script's replaceable parameters. ATL automatically uses `%MODULE%`. To use additional replaceable parameters, see the Remarks for details. Otherwise, use the **NULL** default value.

Return Value

A standard HRESULT value.

Remarks

Runs the script contained in the resource specified by *lpszRes* or *nResID*.

If *bRegister* is **TRUE**, this method registers the object in the system registry; otherwise, it unregisters the object.

By specifying the DECLARE_REGISTRY_RESOURCE or DECLARE_REGISTRY_RESOURCEID macro, **UpdateRegistryFromResourceD** will be invoked automatically when your object map is processed.

Note

To substitute replacement values at run time, do not specify the **DECLARE_REGISTRY_RESOURCE** or **DECLARE_REGISTRY_RESOURCEID** macro. Instead, create an array of **_ATL_REGMAP_ENTRIES** structures, where each entry contains a variable placeholder paired with a value to replace the placeholder at run time. Then call **UpdateRegistryFromResourceD**, passing the array for the *pMapEntries* parameter. This adds all the replacement values in the **_ATL_REGMAP_ENTRIES** structures to the Registrar's replacement map.

Note

To statically link to the ATL Registry Component (Registrar), see UpdateRegistryFromResourceS.

For more information about replaceable parameters and scripting, see the article The ATL Registry Component (Registrar).

See Also

Reference

CComModule Class

Other Resources

CComModule Members

CComModule::UpdateRegistryFromResourceS

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
virtual HRESULT UpdateRegistryFromResourceS(
   LPCTSTR lpszRes,
   BOOL bRegister,
   struct _ATL_REGMAP_ENTRY* pMapEntries = NULL
) throw( );
virtual HRESULT UpdateRegistryFromResourceS(
   UINT nResID,
   BOOL bRegister,
   struct _ATL_REGMAP_ENTRY* pMapEntries = NULL
) throw( );
```

Parameters

lpszRes

[in] A resource name.

nResID

[in] A resource ID.

bRegister

[in] Indicates whether the resource script should be registered.

pMapEntries

[in] A pointer to the replacement map storing values associated with the script's replaceable parameters. ATL automatically uses `%MODULE%`. To use additional replaceable parameters, see the Remarks for details. Otherwise, use the **NULL** default value.

Return Value

A standard HRESULT value.

Remarks

Similar to UpdateRegistryFromResourceD except **UpdateRegistryFromResourceS** creates a static link to the ATL Registry Component (Registrar).

UpdateRegistryFromResourceS will be invoked automatically when your object map is processed, provided you add `#define _ATL_STATIC_REGISTRY` to your stdafx.h.

Note

To substitute replacement values at run time, do not specify the DECLARE_REGISTRY_RESOURCE or DECLARE_REGISTRY_RESOURCEID macro. Instead, create an array of **_ATL_REGMAP_ENTRIES** structures, where each entry contains a variable placeholder paired with a value to replace the placeholder at run time. Then call **UpdateRegistryFromResourceS**, passing the array for the *pMapEntries* parameter. This adds all the replacement values in the **_ATL_REGMAP_ENTRIES** structures to the Registrar's replacement map.

For more information about replaceable parameters and scripting, see the article The ATL Registry Component (Registrar).

See Also

Reference

CComModule Class

Other Resources

CComModule Members

CComModule Data Members

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

For information about the data members in **CComModule**, see CComModule Members.

See Also
Reference
CComModule Class

CComModule::m_csObjMap

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
CRITICAL_SECTION m_csObjMap;
```

Remarks

Ensures synchronized access to the object map.

See Also

Reference

CComModule Class

Other Resources

CComModule Members

CComModule::m_csTypeInfoHolder

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
CRITICAL_SECTION m_csTypeInfoHolder;
```

Remarks

Ensures synchronized access to the type library.

See Also

Reference

CComModule Class

Other Resources

CComModule Members

CComModule::m_csWindowCreate

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
CRITICAL_SECTION m_csWindowCreate;
```

Remarks

Ensures synchronized access to window class information and to static data used during window creation.

See Also

Reference

CComModule Class

Other Resources

CComModule Members

CComModule::m_hInst

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
HINSTANCE m_hInst;
```

Remarks

Contains the handle to the module instance.

The Init method sets **m_hInst** to the handle passed to **DLLMain** or **WinMain**.

See Also

Reference

CComModule Class

Other Resources

CComModule Members

CComModule::m_hInstResource

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
HINSTANCE m_hInstResource;
```

Remarks

By default, contains the handle to the module instance.

The Init method sets **m_hInstResource** to the handle passed to **DLLMain** or **WinMain**. You can explicitly set **m_hInstResource** to the handle to a resource.

The GetResourceInstance method returns the handle stored in **m_hInstResource**.

See Also

Reference

CComModule Class

Other Resources

CComModule Members

CComModule::m_hInstTypeLib

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
HINSTANCE m_hInstTypeLib;
```

Remarks

By default, contains the handle to the module instance.

The Init method sets **m_hInstTypeLib** to the handle passed to **DLLMain** or **WinMain**. You can explicitly set **m_hInstTypeLib** to the handle to a type library.

The GetTypeLibInstance method returns the handle stored in **m_hInstTypeLib**.

See Also

Reference

CComModule Class

Other Resources

CComModule Members

CComModule::m_pObjMap

As of ATL 7.0, **CComModule** is obsolete: see ATL Module Classes for more details.

```
_ATL_OBJMAP_ENTRY* m_pObjMap;
```

Remarks

Points to the object map maintained by the module instance.

See Also

Reference

CComModule Class

Other Resources

CComModule Members

CComMultiThreadModel Class

CComMultiThreadModel provides thread-safe methods for incrementing and decrementing the value of a variable.

```
class CComMultiThreadModel
```

Remarks

Typically, you use **CComMultiThreadModel** through one of two **typedef** names, either CComObjectThreadModel or CComGlobalsThreadModel. The class referenced by each **typedef** depends on the threading model used, as shown in the following table:

typedef	Single threading	Apartment threading	Free threading
CComObjectThreadModel	S	S	M
CComGlobalsThreadModel	S	M	M

S=**CComSingleThreadModel**; M=**CComMultiThreadModel**

CComMultiThreadModel itself defines three **typedef** names. **AutoCriticalSection** and **CriticalSection** reference classes that provide methods for obtaining and releasing ownership of a critical section. **ThreadModelNoCS** references class CComMultiThreadModelNoCS.

Requirements

Header: atlbase.h

See Also

Reference

CComSingleThreadModel Class
CComAutoCriticalSection Class
CComCriticalSection Class

Other Resources

CComMultiThreadModel Members
ATL Class Overview

CComMultiThreadModel Members

Static Functions

Decrement	Decrements the value of the specified variable in a thread-safe manner.
Increment	Increments the value of the specified variable in a thread-safe manner.

Typedefs

AutoCriticalSection	References class CComAutoCriticalSection.
CriticalSection	References class CComCriticalSection.
ThreadModelNoCS	References class CComMultiThreadModelNoCS.

See Also

Reference

CComMultiThreadModel Class

CComMultiThreadModel Static Functions

For information about the static functions in **CComMultiThreadModel**, see CComMultiThreadModel Members.

CComMultiThreadModel::Decrement

This static function calls the Win32 function InterlockedDecrement, which decrements the value of the variable pointed to by *p*.

```
static ULONG WINAPI Decrement(
   LPLONG p
) throw ( );
```

Parameters

p

[in] Pointer to the variable to be decremented.

Return Value

If the result of the decrement is 0, then **Decrement** returns 0. If the result of the decrement is nonzero, the return value is also nonzero but may not equal the result of the decrement.

Remarks

InterlockedDecrement prevents more than one thread from simultaneously using this variable.

See Also

Reference

CComMultiThreadModel Class

CComMultiThreadModel::Increment

Other Resources

CComMultiThreadModel Members

CComMultiThreadModel::Increment

This static function calls the Win32 function InterlockedIncrement, which increments the value of the variable pointed to by *p*.

```
static ULONG WINAPI Increment(
   LPLONG p
) throw ( );
```

Parameters

p

[in] Pointer to the variable to be incremented.

Return Value

If the result of the increment is 0, then **Increment** returns 0. If the result of the increment is nonzero, the return value is also nonzero but may not equal the result of the increment.

Remarks

InterlockedIncrement prevents more than one thread from simultaneously using this variable.

See Also

Reference

CComMultiThreadModel Class

CComMultiThreadModel::Decrement

Other Resources

CComMultiThreadModel Members

CComMultiThreadModel Typedefs

For information about the typedefs in **CComMultiThreadModel**, see CComMultiThreadModel Members.

CComMultiThreadModel::AutoCriticalSection

When using **CComMultiThreadModel**, the **typedef** name **AutoCriticalSection** references class CComAutoCriticalSection, which provides methods for obtaining and releasing ownership of a critical section object.

```
typedef CComAutoCriticalSection AutoCriticalSection;
```

Remarks

CComSingleThreadModel and CComMultiThreadModelNoCS also contain definitions for **AutoCriticalSection**. The following table shows the relationship between the threading model class and the critical section class referenced by **AutoCriticalSection**:

Class defined in	Class referenced
CComMultiThreadModel	**CComCriticalSection**
CComSingleThreadModel	**CComFakeCriticalSection**
CComMultiThreadModelNoCS	**CComFakeCriticalSection**

In addition to **AutoCriticalSection**, you can use the **typedef** name CriticalSection. You should not specify **AutoCriticalSection** in global objects or static class members if you want to eliminate the CRT startup code.

Example

The following code is taken from CComObjectRootEx, and demonstrates **AutoCriticalSection** being used in a threading environment.

```
template< class ThreadModel >
class CComObjectRootEx : public CComObjectRootBase
{
public:
   typedef ThreadModel _ThreadModel;
   typedef _ThreadModel::AutoCriticalSection _CritSec;

   ULONG InternalAddRef( )
   {
      ...
      return _ThreadModel::Increment(&m_dwRef);
   }
   ...
   void Lock( ) { m_critsec.Lock( ); }
   ...

private:
   _CritSec m_critsec;

};
```

The following tables show the results of the **InternalAddRef** and **Lock** methods, depending on the **ThreadModel** template parameter and the threading model used by the application:

ThreadModel = CComObjectThreadModel

Method	Single or Apartment Threading	Free Threading
InternalAddRef	The increment is not thread-safe.	The increment is thread-safe.
Lock	Does nothing; there is no critical section to lock.	The critical section is locked.

ThreadModel = CComObjectThreadModel::ThreadModelNoCS

Method	Single or Apartment Threading	Free Threading
InternalAddRef	The increment is not thread-safe.	The increment is thread-safe.
Lock	Does nothing; there is no critical section to lock.	Does nothing; there is no critical section to lock.

See Also

Reference

CComMultiThreadModel Class
CComObjectThreadModel
CComGlobalsThreadModel
CComMultiThreadModel::ThreadModelNoCS

Other Resources

CComMultiThreadModel Members

CComMultiThreadModel::CriticalSection

When using **CComMultiThreadModel**, the **typedef** name **CriticalSection** references class CComCriticalSection, which provides methods for obtaining and releasing ownership of a critical section object.

```
typedef CComCriticalSection CriticalSection;
```

Remarks

CComSingleThreadModel and CComMultiThreadModelNoCS also contain definitions for **CriticalSection**. The following table shows the relationship between the threading model class and the critical section class referenced by **CriticalSection**:

Class defined in	Class referenced
CComMultiThreadModel	**CComCriticalSection**
CComSingleThreadModel	**CComFakeCriticalSection**
CComMultiThreadModelNoCS	**CComFakeCriticalSection**

In addition to **CriticalSection**, you can use the **typedef** name AutoCriticalSection. You should not specify **AutoCriticalSection** in global objects or static class members if you want to eliminate the CRT startup code.

Example

See CComMultiThreadModel::AutoCriticalSection.

See Also

Reference

CComMultiThreadModel Class
CComObjectThreadModel
CComGlobalsThreadModel
CComMultiThreadModel::ThreadModelNoCS

Other Resources

CComMultiThreadModel Members

CComMultiThreadModel::ThreadModelNoCS

When using **CComMultiThreadModel**, the **typedef** name **ThreadModelNoCS** references class CComMultiThreadModelNoCS.

```
typedef CComMultiThreadModelNoCS ThreadModelNoCS;
```

Remarks

CComMultiThreadModelNoCS provides thread-safe methods for incrementing and decrementing a variable; however, it does not provide a critical section.

CComSingleThreadModel and **CComMultiThreadModelNoCS** also contain definitions for **ThreadModelNoCS**. The following table shows the relationship between the threading model class and the class referenced by **ThreadModelNoCS**:

Class defined in	Class referenced
CComMultiThreadModel	**CComMultiThreadModelNoCS**
CComSingleThreadModel	**CComSingleThreadModel**
CComMultiThreadModelNoCS	**CComMultiThreadModelNoCS**

Example

See CComMultiThreadModel::AutoCriticalSection.

See Also

Reference

CComMultiThreadModel Class
CComObjectThreadModel
CComGlobalsThreadModel

Other Resources

CComMultiThreadModel Members

CComMultiThreadModelNoCS Class

CComMultiThreadModelNoCS provides thread-safe methods for incrementing and decrementing the value of a variable, without critical section locking or unlocking functionality.

```
class CComMultiThreadModelNoCS
```

Remarks

CComMultiThreadModelNoCS is similar to CComMultiThreadModel in that it provides thread-safe methods for incrementing and decrementing a variable. However, when you reference a critical section class through **CComMultiThreadModelNoCS**, methods such as **Lock** and **Unlock** will do nothing.

Typically, you use **CComMultiThreadModelNoCS** through the **ThreadModelNoCS typedef** name. This **typedef** is defined in **CComMultiThreadModelNoCS**, **CComMultiThreadModel**, and CComSingleThreadModel.

Note
The global **typedef** names CComObjectThreadModel and CComGlobalsThreadModel do not reference **CComMultiThread ModelNoCS**.

In addition to **ThreadModelNoCS**, **CComMultiThreadModelNoCS** defines **AutoCriticalSection** and **CriticalSection**. These latter two **typedef** names reference CComFakeCriticalSection, which provides empty methods associated with obtaining and releasing a critical section.

Requirements

Header: atlbase.h

See Also

Other Resources

CComMultiThreadModelNoCS Members

ATL Class Overview

CComMultiThreadModelNoCS Members

Static Functions

Decrement	Decrements the value of the specified variable in a thread-safe manner.
Increment	Increments the value of the specified variable in a thread-safe manner.

Typedefs

AutoCriticalSection	References class CComFakeCriticalSection.
CriticalSection	References class **CComFakeCriticalSection**.
ThreadModelNoCS	References class **CComMultiThreadModelNoCS**.

See Also

Reference

CComMultiThreadModelNoCS Class

CComMultiThreadModelNoCS Static Functions

For information about the static functions in **CComMultiThreadModelNoCS**, see CComMultiThreadModelNoCS Members.

CComMultiThreadModelNoCS::Decrement

This static function calls the Win32 function InterlockedDecrement, which decrements the value of the variable pointed to by *p*.

```
static ULONG WINAPI Decrement(
   LPLONG p
) throw( );
```

Parameters

p

[in] Pointer to the variable to be decremented.

Return Value

If the result of the decrement is 0, then **Decrement** returns 0. If the result of the decrement is nonzero, the return value is also nonzero but may not equal the result of the decrement.

Remarks

InterlockedDecrement prevents more than one thread from simultaneously using this variable.

See Also

Reference

CComMultiThreadModelNoCS Class

CComMultiThreadModelNoCS::Increment

Other Resources

CComMultiThreadModelNoCS Members

CComMultiThreadModelNoCS::Increment

This static function calls the Win32 function InterlockedIncrement, which increments the value of the variable pointed to by *p*.

```
static ULONG WINAPI Increment(
   LPLONG p
) throw( );
```

Parameters

p

[in] Pointer to the variable to be incremented.

Return Value

If the result of the increment is 0, then **Increment** returns 0. If the result of the increment is nonzero, the return value is also nonzero but may not equal the result of the increment.

Remarks

InterlockedIncrement prevents more than one thread from simultaneously using this variable.

See Also

Reference

CComMultiThreadModelNoCS Class

CComMultiThreadModelNoCS::Decrement

Other Resources

CComMultiThreadModelNoCS Members

CComMultiThreadModelNoCS Typedefs

For information about the typedefs in **CComMultiThreadModelNoCS**, see CComMultiThreadModelNoCS Members.

CComMultiThreadModelNoCS::AutoCriticalSection

When using **CComMultiThreadModelNoCS**, the **typedef** name **AutoCriticalSection** references class CComFakeCriticalSection.

```
typedef CComFakeCriticalSection AutoCriticalSection;
```

Remarks

Because **CComFakeCriticalSection** does not provide a critical section, its methods do nothing.

CComMultiThreadModel and CComSingleThreadModel also contain definitions for **AutoCriticalSection**. The following table shows the relationship between the threading model class and the critical section class referenced by **AutoCriticalSection**:

Class defined in	Class referenced
CComMultiThreadModelNoCS	**CComFakeCriticalSection**
CComMultiThreadModel	**CComAutoCriticalSection**
CComSingleThreadModel	**CComFakeCriticalSection**

In addition to **AutoCriticalSection**, you can use the **typedef** name CriticalSection. You should not specify **AutoCriticalSection** in global objects or static class members if you want to eliminate the CRT startup code.

Example

See CComMultiThreadModel::AutoCriticalSection.

See Also

Reference

CComMultiThreadModelNoCS Class
CComObjectThreadModel
CComGlobalsThreadModel
CComMultiThreadModelNoCS::ThreadModelNoCS

Other Resources

CComMultiThreadModelNoCS Members

CComMultiThreadModelNoCS::CriticalSection

When using **CComMultiThreadModelNoCS**, the **typedef** name **CriticalSection** references class CComFakeCriticalSection.

```
typedef CComFakeCriticalSection CriticalSection;
```

Remarks

Because **CComFakeCriticalSection** does not provide a critical section, its methods do nothing.

CComMultiThreadModel and CComSingleThreadModel also contain definitions for **CriticalSection**. The following table shows the relationship between the threading model class and the critical section class referenced by **CriticalSection**:

Class defined in	Class referenced
CComMultiThreadModelNoCS	**CComFakeCriticalSection**
CComMultiThreadModel	**CComCriticalSection**
CComSingleThreadModel	**CComFakeCriticalSection**

In addition to **CriticalSection**, you can use the **typedef** name **AutoCriticalSection**. You should not specify **AutoCriticalSection** in global objects or static class members if you want to eliminate the CRT startup code.

Example

See CComMultiThreadModel::AutoCriticalSection.

See Also

Reference

CComMultiThreadModelNoCS Class
CComObjectThreadModel
CComGlobalsThreadModel
CComMultiThreadModelNoCS::ThreadModelNoCS

Other Resources

CComMultiThreadModelNoCS Members

CComMultiThreadModelNoCS::ThreadModelNoCS

When using **CComMultiThreadModelNoCS**, the **typedef** name **ThreadModelNoCS** simply references **CComMultiThreadModelNoCS**.

```
typedef CComMultiThreadModelNoCS ThreadModelNoCS;
```

Remarks

CComMultiThreadModel and CComSingleThreadModel also contain definitions for **ThreadModelNoCS**. The following table shows the relationship between the threading model class and the class referenced by **ThreadModelNoCS**:

Class defined in	Class referenced
CComMultiThreadModelNoCS	**CComMultiThreadModelNoCS**
CComMultiThreadModel	**CComMultiThreadModelNoCS**
CComSingleThreadModel	**CComSingleThreadModel**

Note that the definition of **ThreadModelNoCS** in **CComMultiThreadModelNoCS** provides symmetry with **CComMultiThreadModel** and **CComSingleThreadModel**. For example, suppose the sample code in **CComMultiThreadModel::AutoCriticalSection** declared the following **typedef**:

```
typedef ThreadModel::ThreadModelNoCS _ThreadModel;
```

Regardless of the class specified for `ThreadModel` (such as **CComMultiThreadModelNoCS**), `_ThreadModel` resolves accordingly.

Example

See CComMultiThreadModel::AutoCriticalSection.

See Also

Reference

CComMultiThreadModelNoCS Class
CComObjectThreadModel
CComGlobalsThreadModel

Other Resources

CComMultiThreadModelNoCS Members

CComObject Class

This class implements **IUnknown** for a nonaggregated object.

```
template<
   class Base
>
class CComObject :
   public Base
```

Parameters

Base

Your class, derived from CComObjectRoot or CComObjectRootEx, as well as from any other interfaces you want to support on the object.

Remarks

CComObject implements IUnknown for a nonaggregated object. However, calls to **QueryInterface**, **AddRef**, and **Release** are delegated to **CComObjectRootEx**.

For more information about using **CComObject**, see the article Fundamentals of ATL COM Objects.

Requirements

Header: atlcom.h

See Also

Reference

CComAggObject Class
CComPolyObject Class
DECLARE_AGGREGATABLE
DECLARE_NOT_AGGREGATABLE

Other Resources

CComObject Members
ATL Class Overview

CComObject Members

Methods

CComObject	The constructor.
~CComObject	The destructor.

IUnknown Methods

AddRef	Increments the reference count on the object.
QueryInterface	Retrieves a pointer to the requested interface.
Release	Decrements the reference count on the object.

Static Functions

CreateInstance	Creates a new **CComObject** object.

See Also

Reference

CComObject Class

CComObject Methods

For information about the methods in **CComObject**, see CComObject Members.

CComObject::AddRef

Increments the reference count on the object.

```
STDMETHOD_(ULONG, AddRef)( );
```

Return Value

This function returns the new incremented reference count on the object. This value may be useful for diagnostics or testing.

See Also

Reference

CComObject Class

CComObject::Release

Other Resources

CComObject Members

CComObject::CComObject

The constructor increments the module lock count.

```
CComObject(
   void* = NULL
);
```

Parameters

void*

[in] This unnamed parameter is not used. It exists for symmetry with other **CCom***XXX***Object***XXX* constructors.

Remarks

The destructor decrements it.

If a **CComObject**-derived object is successfully constructed using the **new** operator, the initial reference count is 0. To set the reference count to the proper value (1), make a call to the AddRef function.

See Also

Reference

CComObject Class

Other Resources

CComObject Members

CComObject::~CComObject

The destructor.

```
CComObject( );
```

Remarks

Frees all allocated resources, calls FinalRelease, and decrements the module lock count.

See Also

Reference

CComObject Class

Other Resources

CComObject Members

CComObject::QueryInterface

Retrieves a pointer to the requested interface.

```
STDMETHOD(QueryInterface)(
   REFIID iid,
   void** ppvObject
);
template <class Q>
HRESULT STDMETHODCALLTYPE QueryInterface(
   Q** pp
);
```

Parameters

iid

[in] The identifier of the interface being requested.

ppvObject

[out] A pointer to the interface pointer identified by *iid*. If the object does not support this interface, *ppvObject* is set to **NULL**.

pp

[out] A pointer to the interface pointer identified by type *Q*. If the object does not support this interface, *pp* is set to **NULL**.

Return Value

A standard **HRESULT** value.

See Also

Reference

CComObject Class

Other Resources

CComObject Members

CComObject::Release

Decrements the reference count on the object.

```
STDMETHOD_(ULONG, Release)( );
```

Return Value

This function returns the new decremented reference count on the object. In debug builds, the return value may be useful for diagnostics or testing. In non-debug builds, **Release** always returns 0.

See Also

Reference

CComObject Class

CComObject::AddRef

Other Resources

CComObject Members

CComObject Static Functions

For information about the static functions in **CComObject**, see CComObject Members.

CComObject::CreateInstance

This static function allows you to create a new **CComObject<***Base***>** object, without the overhead of CoCreateInstance.

```
static HRESULT WINAPI CreateInstance(
   CComObject< Base >** pp
);
```

Parameters

pp

[out] A pointer to a **CComObject<***Base***>** pointer. If **CreateInstance** is unsuccessful, *pp* is set to **NULL**.

Return Value

A standard **HRESULT** value.

Remarks

The object returned has a reference count of zero, so call **AddRef** immediately, then use **Release** to free the reference on the object pointer when you're done.

If you do not need direct access to the object, but still want to create a new object without the overhead of **CoCreateInstance**, use CComCoClass::CreateInstance instead.

Example

```
// Definition of CMyCircle.
class CMyCircle :
   public IDispatchImpl<IMyCircle, &IID_IMyCircle, &LIBID_CIRCCOLLLib>,
   public ISupportErrorInfo,
   public CComObjectRoot
{
public:
   CMyCircle();
BEGIN_COM_MAP(CMyCircle)
   COM_INTERFACE_ENTRY(IDispatch)
   COM_INTERFACE_ENTRY(IMyCircle)
   COM_INTERFACE_ENTRY(ISupportErrorInfo)
END_COM_MAP()
DECLARE_NOT_AGGREGATABLE(CMyCircle)

// IMyCircle
   STDMETHOD(get_XCenter)(double* pXCenter);
...
...
}

// Create a local instance of COM object CMyCircle.
double x;
CComObject<CMyCircle>* pCircle;
HRESULT hRes = CComObject<CMyCircle>::CreateInstance(&pCircle);
_ASSERTE(SUCCEEDED(hRes));

// Increment reference count immediately
pCircle->AddRef();

// Access method of COM object
hRes = pCircle->get_XCenter(&x);

// Decrement reference count when done
pCircle->Release();
pCircle = NULL;
```

See Also

Reference

CComObject Class

Other Resources

CComObject Members

CComObjectGlobal Class

This class manages a reference count on the module containing your *Base* object.

```
template<
   class Base
>
class CComObjectGlobal :
      public Base
```

Parameters

Base

Your class, derived from CComObjectRoot or CComObjectRootEx, as well as from any other interface you want to support on the object.

Remarks

CComObjectGlobal manages a reference count on the module containing your *Base* object. **CComObjectGlobal** ensures your object will not be deleted as long as the module is not released. Your object will only be removed when the reference count on the entire module goes to zero.

For example, using **CComObjectGlobal**, a class factory can hold a common global object that is shared by all its clients.

Requirements

Header: atlcom.h

See Also

Reference

CComObjectStack Class
CComAggObject Class
CComObject Class

Other Resources

CComObjectGlobal Members
ATL Class Overview

CComObjectGlobal Members

Class Methods

CComObjectGlobal	The constructor.
~CComObjectGlobal	The destructor.

IUnknown Methods

AddRef	Implements a global **AddRef**.
QueryInterface	Implements a global **QueryInterface**.
Release	Implements a global **Release**.

Data Members

m_hResFinalConstruct	Contains the **HRESULT** returned during construction of the **CComObjectGlobal** object.

See Also

Reference

CComObjectGlobal Class

CComObjectGlobal Methods

For information about the methods in **CComObjectGlobal**, see CComObjectGlobal Members.

CComObjectGlobal::AddRef

Increments the reference count of the object by 1.

```
STDMETHOD_(ULONG, AddRef)( );
```

Return Value

A value that may be useful for diagnostics and testing.

Remarks

By default, **AddRef** calls **_Module::Lock**, where **_Module** is the global instance of CComModule or a class derived from it.

See Also

Reference

CComObjectGlobal Class

CComObjectGlobal::Release

Other Resources

CComObjectGlobal Members

CComObjectGlobal::CComObjectGlobal

The constructor. Calls **FinalConstruct** and then sets m_hResFinalConstruct to the **HRESULT** returned by **FinalConstruct**.

```
CComObjectGlobal(
   void* = NULL
) );
```

Remarks

If you have not derived your base class from CComObjectRoot, you must supply your own **FinalConstruct** method. The destructor calls **FinalRelease**.

See Also

Reference

CComObjectGlobal Class

CComObjectRootEx::FinalConstruct

Other Resources

CComObjectGlobal Members

CComObjectGlobal::~CComObjectGlobal

The destructor.

```
CComObjectGlobal( );
```

Remarks

Frees all allocated resources and calls FinalRelease.

See Also

Reference

CComObjectGlobal Class

Other Resources

CComObjectGlobal Members

CComObjectGlobal::QueryInterface

Retrieves a pointer to the requested interface pointer.

```
STDMETHOD(QueryInterface)(
   REFIID iid,
      void** ppvObject
) ;
```

Parameters

iid

[in] The GUID of the interface being requested.

ppvObject

[out] A pointer to the interface pointer identified by iid, or **NULL** if the interface is not found.

Return Value

A standard **HRESULT** value.

Remarks

QueryInterface only handles interfaces in the COM map table.

See Also

Reference

CComObjectGlobal Class
CComObjectRootEx::InternalQueryInterface
BEGIN_COM_MAP

Other Resources

CComObjectGlobal Members

CComObjectGlobal::Release

Decrements the reference count of the object by 1.

```
STDMETHOD_(ULONG, Release)( );
```

Return Value

In debug builds, **Release** returns a value that may be useful for diagnostics and testing. In non-debug builds, **Release** always returns 0.

Remarks

By default, **Release** calls **_Module::Unlock**, where **_Module** is the global instance of CComModule or a class derived from it.

See Also

Reference

CComObjectGlobal Class

CComObjectGlobal::AddRef

Other Resources

CComObjectGlobal Members

CComObjectGlobal Data Members

For information about the data members in **CComObjectGlobal**, see CComObjectGlobal Members.

CComObjectGlobal::m_hResFinalConstruct

Contains the **HRESULT** from calling **FinalConstruct** during construction of the **CComObjectGlobal** object.

```
HRESULT m_hResFinalConstruct;
```

See Also

Reference

CComObjectGlobal Class

CComObjectRootEx::FinalConstruct

Other Resources

CComObjectGlobal Members

CComObjectNoLock Class

This class implements **IUnknown** for a nonaggregated object, but does not increment the module lock count in the constructor.

```
template<
   class Base
>
class CComObjectNoLock :
   public Base
```

Parameters

Base

Your class, derived from CComObjectRoot or CComObjectRootEx, as well as from any other interface you want to support on the object.

Remarks

CComObjectNoLock is similar to CComObject in that it implements IUnknown for a nonaggregated object; however, **CComObjectNoLock** does not increment the module lock count in the constructor.

ATL uses **CComObjectNoLock** internally for class factories. In general, you will not use this class directly.

Requirements

Header: atlcom.h

See Also

Other Resources

CComObjectNoLock Members

ATL Class Overview

CComObjectNoLock Members

Class Methods

CComObjectNoLock	Constructor.
~CComObjectNoLock	The destructor.

IUnknown Methods

AddRef	Increments the reference count on the object.
QueryInterface	Returns a pointer to the requested interface.
Release	Decrements the reference count on the object.

See Also

Reference

CComObjectNoLock Class

CComObjectNoLock Methods

For information about the methods in **CComObjectNoLock**, see CComObjectNoLock Members.

CComObjectNoLock::AddRef

Increments the reference count on the object.

```
STDMETHOD_(ULONG, AddRef)( );
```

Return Value

A value that may be useful for diagnostics or testing.

See Also

Reference

CComObjectNoLock Class

CComObjectNoLock::Release

Other Resources

CComObjectNoLock Members

CComObjectNoLock::CComObjectNoLock

The constructor. Unlike CComObject, does not increment the module lock count.

```
CComObjectNoLock(
   void* = NULL
);
```

Parameters

void*

[in] This unnamed parameter is not used. It exists for symmetry with other **CCom***XXX***Object***XXX* constructors.

See Also

Reference

CComObjectNoLock Class

Other Resources

CComObjectNoLock Members

CComObjectNoLock::~CComObjectNoLock

The destructor.

```
~CComObjectNoLock( );
```

Remarks

Frees all allocated resources and calls FinalRelease.

See Also

Reference

CComObjectNoLock Class

Other Resources

CComObjectNoLock Members

CComObjectNoLock::QueryInterface

Retrieves a pointer to the requested interface.

```
STDMETHOD(QueryInterface)(
   REFIID iid,
   void** ppvObject
);
```

Parameters

iid

[in] The identifier of the interface being requested.

ppvObject

[out] A pointer to the interface pointer identified by *iid*. If the object does not support this interface, *ppvObject* is set to **NULL**.

Return Value

A standard **HRESULT** value.

See Also

Reference

CComObjectNoLock Class

Other Resources

CComObjectNoLock Members

CComObjectNoLock::Release

Decrements the reference count on the object.

```
STDMETHOD_(ULONG, Release)( );
```

Return Value

In debug builds, **Release** returns a value that may be useful for diagnostics or testing. In non-debug builds, **Release** always returns 0.

See Also

Reference

CComObjectNoLock Class

CComObjectNoLock::AddRef

Other Resources

CComObjectNoLock Members

CComObjectRoot Class

This typedef of CComObjectRootEx is templatized on the default threading model of the server.

```
typedef CComObjectRootEx<CComObjectThreadModel> CComObjectRoot;
```

Remarks

CComObjectRoot is a **typedef** of CComObjectRootEx templatized on the default threading model of the server. Thus CComObjectThreadModel will reference either CComSingleThreadModel or CComMultiThreadModel.

CComObjectRootEx handles object reference count management for both nonaggregated and aggregated objects. It holds the object reference count if your object is not being aggregated, and holds the pointer to the outer unknown if your object is being aggregated. For aggregated objects, **CComObjectRootEx** methods can be used to handle the failure of the inner object to construct, and to protect the outer object from deletion when inner interfaces are released or the inner object is deleted.

Requirements

Header: atlcom.h

See Also

Reference

CComObjectRootEx Class
CComAggObject Class
CComObject Class
CComPolyObject Class

Other Resources

CComObjectRootEx Members
ATL Class Overview

CComObjectRootEx Class

This class provides methods to handle object reference count management for both nonaggregated and aggregated objects.

```
template<
   class ThreadModel
>
class CComObjectRootEx : public CComObjectRootBase
```

Parameters

ThreadModel

The class whose methods implement the desired threading model. You can explicitly choose the threading model by setting *ThreadModel* to CComSingleThreadModel, CComMultiThreadModel, or CComMultiThreadModelNoCS. You can accept the server's default thread model by setting *ThreadModel* to CComObjectThreadModel or CComGlobalsThreadModel.

Remarks

CComObjectRootEx handles object reference count management for both nonaggregated and aggregated objects. It holds the object reference count if your object is not being aggregated, and holds the pointer to the outer unknown if your object is being aggregated. For aggregated objects, **CComObjectRootEx** methods can be used to handle the failure of the inner object to construct, and to protect the outer object from deletion when inner interfaces are released or the inner object is deleted.

A class that implements a COM server must inherit from **CComObjectRootEx** or CComObjectRoot.

If your class definition specifies the DECLARE_POLY_AGGREGATABLE macro, ATL creates an instance of **CComPolyObject<CYourClass>** when **IClassFactory::CreateInstance** is called. During creation, the value of the outer unknown is checked. If it is **NULL**, **IUnknown** is implemented for a nonaggregated object. If the outer unknown is not **NULL**, **IUnknown** is implemented for an aggregated object.

If your class does not specify the **DECLARE_POLY_AGGREGATABLE** macro, ATL creates an instance of **CAggComObject<CYourClass>** for aggregated objects or an instance of **CComObject<CYourClass>** for nonaggregated objects.

The advantage of using **CComPolyObject** is that you avoid having both **CComAggObject** and **CComObject** in your module to handle the aggregated and nonaggregated cases. A single **CComPolyObject** object handles both cases. Therefore, only one copy of the vtable and one copy of the functions exist in your module. If your vtable is large, this can substantially decrease your module size. However, if your vtable is small, using **CComPolyObject** can result in a slightly larger module size because it is not optimized for an aggregated or nonaggregated object, as are **CComAggObject** and **CComObject**.

If your object is aggregated, IUnknown is implemented by **CComAggObject** or **CComPolyObject**. These classes delegate **QueryInterface**, **AddRef**, and **Release** calls to **CComObjectRootEx**'s **OuterQueryInterface**, **OuterAddRef**, and **OuterRelease** to forward to the outer unknown. Typically, you override **CComObjectRootEx::FinalConstruct** in your class to create any aggregated objects, and override **CComObjectRootEx::FinalRelease** to free any aggregated objects.

If your object is not aggregated, **IUnknown** is implemented by **CComObject** or **CComPolyObject**. In this case, calls to **QueryInterface**, **AddRef**, and **Release** are delegated to **CComObjectRootEx**'s **InternalQueryInterface**, **InternalAddRef**, and **InternalRelease** to perform the actual operations.

Requirements

Header: atlcom.h

See Also

Reference

CComAggObject Class
CComObject Class
CComPolyObject Class

Other Resources

CComObjectRootEx Members
ATL Class Overview

CComObjectRootEx Members

Methods

CComObjectRootEx	Constructor.
InternalAddRef	Increments the reference count for a nonaggregated object.
InternalRelease	Decrements the reference count for a nonaggregated object.
Lock	If the thread model is multithreaded, obtains ownership of a critical section object.
Unlock	If the thread model is multithreaded, releases ownership of a critical section object.

CComObjectRootBase Methods

FinalConstruct	Override in your class to perform any initialization required by your object.
FinalRelease	Override in your class to perform any cleanup required by your object.
OuterAddRef	Increments the reference count for an aggregated object.
OuterQueryInterface	Delegates to the outer **IUnknown** of an aggregated object.
OuterRelease	Decrements the reference count for an aggregated object.

Static Functions

InternalQueryInterface	Delegates to the **IUnknown** of a nonaggregated object.
ObjectMain	Called during module initialization and termination for derived classes listed in the object map.

Data Members

m_dwRef	With **m_pOuterUnknown**, part of a union. Used when the object is not aggregated to hold the reference count of **AddRef** and **Release**.
m_pOuterUnknown	With **m_dwRef**, part of a union. Used when the object is aggregated to hold a pointer to the outer unknowns.

See Also

Reference

CComObjectRootEx Class

CComObjectRootEx Methods

For information about the methods in **CComObjectRootEx**, see CComObjectRootEx Members.

CComObjectRootEx::CComObjectRootEx

The constructor initializes the reference count to 0.

```
CComObjectRootEx( );
```

See Also

Reference

CComObjectRootEx Class

Other Resources

CComObjectRootEx Members

CComObjectRootEx::FinalConstruct

You can override this method in your derived class to perform any initialization required for your object.

```
HRESULT FinalConstruct( );
```

Return Value

Return **S_OK** on success or one of the standard error **HRESULT** values.

Remarks

By default, **CComObjectRootEx::FinalConstruct** simply returns **S_OK**.

There are advantages to performing initialization in **FinalConstruct** rather than the constructor of your class:

- You cannot return a status code from a constructor, but you can return an **HRESULT** by means of **FinalConstruct**'s return value. When objects of your class are being created using the standard class factory provided by ATL, this return value is propagated back to the COM client allowing you to provide them with detailed error information.
- You cannot call virtual functions through the virtual function mechanism from the constructor of a class. Calling a virtual function from the constructor of a class results in a statically resolved call to the function as it is defined at that point in the inheritance hierarchy. Calls to pure virtual functions result in linker errors.

  Your class is not the most derived class in the inheritance hierarchy — it relies on a derived class supplied by ATL to provide some of its functionality. There is a good chance that your initialization will need to use the features provided by that class (this is certainly true when objects of your class need to aggregate other objects), but the constructor in your class has no way to access those features. The construction code for your class is executed before the most derived class is fully constructed.

  However, **FinalConstruct** is called immediately after the most derived class is fully constructed allowing you to call virtual functions and use the reference-counting implementation provided by ATL.

Example

Typically, override this method in the class derived from **CComObjectRootEx** to create any aggregated objects. For example:

```
class CMyAggObject : public CComObjectRootEx< ... >
{
   DECLARE_GET_CONTROLLING_UNKNOWN( )
   HRESULT FinalConstruct( )
   {
      return CoCreateInstance(CLSID_SomeServer,
               GetControllingUnknown(), CLSCTX_ALL,
               IID_ISomeServer, &m_pSomeServer);
   }
   ...
};
```

If the construction fails, you can return an error. You can also use the macro DECLARE_PROTECT_FINAL_CONSTRUCT to protect your outer object from being deleted if, during creation, the internal aggregated object increments the reference count then decrements the count to 0.

Here is a typical way to create an aggregate:

- Add an **IUnknown** pointer to your class object and initialize it to **NULL** in the constructor.
- Override **FinalConstruct** to create the aggregate.
- Use the **IUnknown** pointer you defined as the parameter to the COM_INTERFACE_ENTRY_AGGREGATE macro.
- Override **FinalRelease** to release the **IUnknown** pointer.

See Also

Reference

CComObjectRootEx Class

CComObjectRootEx::FinalRelease

DECLARE_GET_CONTROLLING_UNKNOWN

Other Resources

CComObjectRootEx Members

CComObjectRootEx::FinalRelease

You can override this method in your derived class to perform any cleanup required for your object.

```
void FinalRelease( );
```

Remarks

By default, **CComObjectRootEx::FinalRelease** does nothing.

Performing cleanup in **FinalRelease** is preferable to adding code to the destructor of your class since the object is still fully constructed at the point at which **FinalRelease** is called. This enables you to safely access the methods provided by the most derived class. This is particularly important for freeing any aggregated objects before deletion.

See Also

Reference

CComObjectRootEx Class

CComObjectRootEx::FinalConstruct

Other Resources

CComObjectRootEx Members

CComObjectRootEx::InternalAddRef

Increments the reference count of a nonaggregated object by 1.

```
ULONG InternalAddRef( );
```

Return Value

A value that may be useful for diagnostics and testing.

Remarks

If the thread model is multithreaded, **InterlockedIncrement** is used to prevent more than one thread from changing the reference count at the same time.

See Also

Reference

CComObjectRootEx Class
CComObjectRootEx::InternalRelease
InterlockedIncrement

Other Resources

CComObjectRootEx Members

CComObjectRootEx::InternalRelease

Decrements the reference count of a nonaggregated object by 1.

```
ULONG InternalRelease( );
```

Return Value

In both non-debug and debug builds, this function returns a value which may be useful for diagnostics or testing. The exact value returned depends on many factors such as the operating system used, and may, or may not, be the reference count.

Remarks

If the thread model is multithreaded, **InterlockedDecrement** is used to prevent more than one thread from changing the reference count at the same time.

See Also

Reference

CComObjectRootEx Class
CComObjectRootEx::InternalAddRef
InterlockedDecrement

Other Resources

CComObjectRootEx Members

CComObjectRootEx::Lock

If the thread model is multithreaded, this method calls the Win32 API function EnterCriticalSection, which waits until the thread can take ownership of the critical section object obtained through a private data member.

```
void Lock( );
```

Remarks

When the protected code finishes executing, the thread must call **Unlock** to release ownership of the critical section.

If the thread model is single-threaded, this method does nothing.

See Also

Reference

CComObjectRootEx Class

CComObjectRootEx::Unlock

Other Resources

CComObjectRootEx Members

CComObjectRootEx::OuterAddRef

Increments the reference count of the outer unknown of an aggregation.

```
ULONG OuterAddRef( );
```

Return Value

A value that may be useful for diagnostics and testing.

See Also

Reference

CComObjectRootEx Class
CComObjectRootEx::OuterRelease
CComObjectRootEx::OuterQueryInterface

Other Resources

CComObjectRootEx Members

CComObjectRootEx::OuterQueryInterface

Retrieves an indirect pointer to the requested interface.

```
HRESULT OuterQueryInterface(
   REFIID iid,
      void** ppvObject
);
```

Parameters

iid

[in] The GUID of the interface being requested.

ppvObject

[out] A pointer to the interface pointer specified in *iid*, or **NULL** if the aggregation does not support the interface.

Return Value

One of the standard **HRESULT** values.

See Also

Reference

CComObjectRootEx Class
CComObjectRootEx::OuterAddRef
CComObjectRootEx::OuterRelease

Other Resources

CComObjectRootEx Members

CComObjectRootEx::OuterRelease

Decrements the reference count of the outer unknown of an aggregation.

```
ULONG OuterRelease( );
```

Return Value

In non-debug builds, always returns 0. In debug builds, returns a value that may be useful for diagnostics or testing.

See Also

Reference

CComObjectRootEx Class
CComObjectRootEx::OuterAddRef
CComObjectRootEx::OuterQueryInterface

Other Resources

CComObjectRootEx Members

CComObjectRootEx::Unlock

If the thread model is multithreaded, this method calls the Win32 API function LeaveCriticalSection, which releases ownership of the critical section object obtained through a private data member.

```
void Unlock( );
```

Remarks

To obtain ownership, the thread must call **Lock**. Each call to **Lock** requires a corresponding call to **Unlock** to release ownership of the critical section.

If the thread model is single-threaded, this method does nothing.

See Also

Reference

CComObjectRootEx Class

CComObjectRootEx::Lock

Other Resources

CComObjectRootEx Members

CComObjectRootEx Data Members

For information about the data members in **CComObjectRootEx**, see CComObjectRootEx Members.

CComObjectRootEx::m_dwRef

Part of a union that accesses four bytes of memory.

```
long m_dwRef;
```

Remarks

With **m_pOuterUnknown**, part of a union:

```
union
{
   long m_dwRef;
   IUnknown* m_pOuterUnknown;
};
```

If the object is not aggregated, the reference count accessed by **AddRef** and **Release** is stored in **m_dwRef**. If the object is aggregated, the pointer to the outer unknown is stored in m_pOuterUnknown.

See Also

Reference

CComObjectRootEx Class

Other Resources

CComObjectRootEx Members

CComObjectRootEx::m_pOuterUnknown

Part of a union that accesses four bytes of memory.

```
IUnknown* m_pOuterUnknown;
```

Remarks

With **m_dwRef**, part of a union:

```
union
{
   long m_dwRef;
   IUnknown* m_pOuterUnknown;
};
```

If the object is aggregated, the pointer to the outer unknown is stored in **m_pOuterUnknown**. If the object is not aggregated, the reference count accessed by **AddRef** and **Release** is stored in m_dwRef.

See Also

Reference

CComObjectRootEx Class

Other Resources

CComObjectRootEx Members

CComObjectRootEx Static Functions

For information about the static functions in **CComObjectRootEx**, see CComObjectRootEx Members.

CComObjectRootEx::InternalQueryInterface

Retrieves a pointer to the requested interface.

```
static HRESULT InternalQueryInterface(
   void* pThis,
      const _ATL_INTMAP_ENTRY* pEntries,
      REFIID iid,
      void** ppvObject
);
```

Parameters

pThis

[in] A pointer to the object that contains the COM map of interfaces exposed to **QueryInterface**.

pEntries

[in] A pointer to the **_ATL_INTMAP_ENTRY** structure that accesses a map of available interfaces.

iid

[in] The GUID of the interface being requested.

ppvObject

[out] A pointer to the interface pointer specified in *iid*, or **NULL** if the interface is not found.

Return Value

One of the standard **HRESULT** values.

Remarks

InternalQueryInterface only handles interfaces in the COM map table. If your object is aggregated, **InternalQueryInterface** does not delegate to the outer unknown. You can enter interfaces into the COM map table with the macro COM_INTERFACE_ENTRY or one of its variants.

See Also

Reference

CComObjectRootEx Class
CComObjectRootEx::InternalAddRef
CComObjectRootEx::InternalRelease

Other Resources

CComObjectRootEx Members

CComObjectRootEx::ObjectMain

For each class listed in the object map, this function is called once when the module is initialized, and again when it is terminated.

```
static void WINAPI ObjectMain(
   bool bStarting
);
```

Parameters

bStarting

[out] The value is **true** if the class is being initialized; otherwise **false**.

Remarks

The value of the *bStarting* parameter indicates whether the module is being initialized or terminated. The default implementation of **ObjectMain** does nothing, but you can override this function in your class to initialize or clean up resources that you want to allocate for the class. Note that **ObjectMain** is called before any instances of the class are requested.

ObjectMain is called from the entry point of the DLL, so the type of operation that the entry-point function can perform is restricted. For more information on these restrictions, see Run-Time Library Behavior and DllMain.

Example

```
class ATL_NO_VTABLE CMyObj :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<CMyObj, &CLSID_MyObj>,
   public IDispatchImpl<IMyObj, &IID_IMyObj, &LIBID_TESTLib>
{
public:
   CMyObj()
   {
   }

static void WINAPI ObjectMain(bool bStarting)
{
   if (bStarting)
      // Perform custom initialization routines
   else
      // Perform custom termination routines
}

. . .
// IMyObj
public:
};
```

See Also

Reference

CComObjectRootEx Class

Other Resources

CComObjectRootEx Members

CComObjectStack Class

This class creates a temporary COM object and provides it with a skeletal implementation of **IUnknown**.

```
template<
   class Base
>
class CComObjectStack :
      public Base
```

Parameters

Base

Your class, derived from CComObjectRoot or CComObjectRootEx, as well as from any other interface you want to support on the object.

Remarks

CComObjectStack is used to create a temporary COM object and provide the object a skeletal implementation of **IUnknown**. Typically, the object is used as a local variable within one function (that is, pushed onto the stack). Since the object is destroyed when the function finishes, reference counting is not performed to increase efficiency.

The following example shows how to create a COM object used inside a function:

```
void MyFunc( )
{
   CComObjectStack<CMyObject> Tempobj;
   ...
}
```

The temporary object `Tempobj` is pushed onto the stack and automatically disappears when the function finishes.

Requirements

Header: atlcom.h

See Also

Reference

CComAggObject Class
CComObject Class
CComObjectGlobal Class

Other Resources

CComObjectStack Members
ATL Class Overview

CComObjectStack Members

Class Methods

CComObjectStack	The constructor.
~CComObjectStack	The destructor.

IUnknown Methods

AddRef	Returns zero. In debug mode, calls **_ASSERTE**.
QueryInterface	Returns **E_NOINTERFACE**. In debug mode, calls **_ASSERTE**.
Release	Returns zero. In debug mode, calls **_ASSERTE**. ~

Data Members

m_hResFinalConstruct	Contains the **HRESULT** returned during construction of the **CComObjectStack** object.

See Also

Reference

CComObjectStack Class

CComObjectStack Methods

For information about the methods in **CComObjectStack**, see CComObjectStack Members.

CComObjectStack::AddRef

Returns zero.

```
STDMETHOD_(ULONG, AddRef)( );
```

Return Value

Returns zero.

Remarks

In debug mode, calls **_ASSERTE**.

See Also

Reference

CComObjectStack Class

Other Resources

CComObjectStack Members

CComObjectStack::CComObjectStack

The constructor.

```
CComObjectStack(
   void* = NULL
);
```

Remarks

Calls **FinalConstruct** and then sets m_hResFinalConstruct to the **HRESULT** returned by **FinalConstruct**. If you have not derived your base class from CComObjectRoot, you must supply your own **FinalConstruct** method. The destructor calls **FinalRelease**.

See Also

Reference

CComObjectStack Class

CComObjectRootEx::FinalConstruct

Other Resources

CComObjectStack Members

CComObjectStack::~CComObjectStack

The destructor.

```
CComObjectStack( );
```

Remarks

Frees all allocated resources and calls FinalRelease.

See Also

Reference

CComObjectStack Class

Other Resources

CComObjectStack Members

CComObjectStack::QueryInterface

Returns **E_NOINTERFACE**.

```
HRESULT QueryInterface(
   REFIID,
      void**
) ;
```

Return Value

Returns **E_NOINTERFACE**.

Remarks

In debug mode, calls **_ASSERTE**.

See Also

Reference

CComObjectStack Class

Other Resources

CComObjectStack Members

CComObjectStack::Release

Returns zero.

```
STDMETHOD_(ULONG, Release)( );
```

Return Value

Returns zero.

Remarks

In debug mode, calls **_ASSERTE**.

See Also

Reference

CComObjectStack Class

Other Resources

CComObjectStack Members

CComObjectStack Data Members

For information about the data members in **CComObjectStack**, see CComObjectStack Members.

CComObjectStack::m_hResFinalConstruct

Contains the **HRESULT** returned from calling **FinalConstruct** during construction of the **CComObjectStack** object.

```
HRESULT m_hResFinalConstruct;
```

See Also

Reference

CComObjectStack Class

CComObjectRootEx::FinalConstruct

Other Resources

CComObjectStack Members

CComPolyObject Class

This class implements **IUnknown** for an aggregated or nonaggregated object.

```
template<
   class contained
>
class CComPolyObject : public IUnknown, public CComObjectRootEx
   < contained::_ThreadModel::ThreadModelNoCS >
```

Parameters

contained

Your class, derived from CComObjectRoot or CComObjectRootEx, as well as from any other interfaces you want to support on the object.

Remarks

CComPolyObject implements IUnknown for an aggregated or nonaggregated object.

When an instance of **CComPolyObject** is created, the value of the outer unknown is checked. If it is **NULL**, **IUnknown** is implemented for a nonaggregated object. If the outer unknown is not **NULL**, **IUnknown** is implemented for an aggregated object.

The advantage of using **CComPolyObject** is that you avoid having both CComAggObject and CComObject in your module to handle the aggregated and nonaggregated cases. A single **CComPolyObject** object handles both cases. This means only one copy of the vtable and one copy of the functions exist in your module. If your vtable is large, this can substantially decrease your module size. However, if your vtable is small, using **CComPolyObject** can result in a slightly larger module size because it is not optimized for an aggregated or nonaggregated object, as are **CComAggObject** and **CComObject**.

If the **DECLARE_POLY_AGGREGATABLE** macro is specified in your object's class definition, **CComPolyObject** will be used to create your object. **DECLARE_POLY_AGGREGATABLE** will automatically be declared if you use the ATL Project Wizard to create a full control or Internet Explorer control.

If aggregated, the **CComPolyObject** object has its own **IUnknown**, separate from the outer object's **IUnknown**, and maintains its own reference count. **CComPolyObject** uses CComContainedObject to delegate to the outer unknown.

For more information about aggregation, see the article Fundamentals of ATL COM Objects.

Requirements

Header: atlcom.h

See Also

Reference

CComObjectRootEx Class

DECLARE_POLY_AGGREGATABLE

Other Resources

CComPolyObject Members

ATL Class Overview

CComPolyObject Members

Class Methods

CComPolyObject	The constructor.
~CComPolyObject	The destructor.
FinalConstruct	Performs final initialization of **m_contained**.
FinalRelease	Performs final destruction of **m_contained**.

IUnknown Methods

AddRef	Increments the object's reference count.
QueryInterface	Retrieves a pointer to the requested interface.
Release	Decrements the object's reference count.

Data Members

m_contained	Delegates **IUnknown** calls to the outer unknown if the object is aggregated or to the **IUnknown** of the object if the object is not aggregated.

Static Functions

CreateInstance	Allows you to create a new **CComPolyObject<** *contained* **>** object without the overhead of CoCreateInstance.

See Also

Reference

CComPolyObject Class

CComPolyObject Methods

For information about the methods in **CComPolyObject**, see CComPolyObject Members.

CComPolyObject::AddRef

Increments the reference count on the object.

```
STDMETHOD_(ULONG, AddRef)( );
```

Return Value

A value that may be useful for diagnostics or testing.

See Also

Reference

CComPolyObject Class

CComPolyObject::Release

Other Resources

CComPolyObject Members

CComPolyObject::CComPolyObject

The constructor.

```
CComPolyObject(
   void* pv
);
```

Parameters

pv

[in] A pointer to the outer unknown if the object is to be aggregated, or **NULL** if the object if the object is not aggregated.

Remarks

Initializes the **CComContainedObject** data member, m_contained, and increments the module lock count.

The destructor decrements the module lock count.

See Also

Reference

CComPolyObject Class
CComPolyObject::FinalConstruct
CComPolyObject::FinalRelease

Other Resources

CComPolyObject Members

CComPolyObject::~CComPolyObject

The destructor.

```
~CComPolyObject( );
```

Remarks

Frees all allocated resources, calls FinalRelease, and decrements the module lock count.

See Also

Reference

CComPolyObject Class

CComPolyObject::FinalRelease

Other Resources

CComPolyObject Members

CComPolyObject::FinalConstruct

Called during the final stages of object construction, this method performs any final initialization on the m_contained data member.

```
HRESULT FinalConstruct( );
```

Return Value

A standard **HRESULT** value.

See Also

Reference

CComPolyObject Class
CComObjectRootEx::FinalConstruct
CComPolyObject::FinalRelease

Other Resources

CComPolyObject Members

CComPolyObject::FinalRelease

Called during object destruction, this method frees the m_contained data member.

```
void FinalRelease( );
```

See Also

Reference

CComPolyObject Class
CComObjectRootEx::FinalRelease
CComPolyObject::FinalConstruct

Other Resources

CComPolyObject Members

CComPolyObject::QueryInterface

Retrieves a pointer to the requested interface.

```
STDMETHOD(QueryInterface)(
   REFIID iid,
   void** ppvObject
);
template <class Q>
HRESULT QueryInterface(Q ** pp);
```

Parameters

Q

The COM interface.

iid

[in] The identifier of the interface being requested.

ppvObject

[out] A pointer to the interface pointer identified by *iid*. If the object does not support this interface, *ppvObject* is set to **NULL**.

pp

[out] A pointer to the interface identified by **__uuidof(Q)**.

Return Value

A standard **HRESULT** value.

Remarks

For an aggregated object, if the requested interface is **IUnknown**, **QueryInterface** returns a pointer to the aggregated object's own **IUnknown** and increments the reference count. Otherwise, this method queries for the interface through the **CComContainedObject** data member, m_contained.

See Also

Reference

CComPolyObject Class

Other Resources

CComPolyObject Members

CComPolyObject::Release

Decrements the reference count on the object.

```
STDMETHOD_(ULONG, Release)( );
```

Return Value

In debug builds, **Release** returns a value that may be useful for diagnostics or testing. In nondebug builds, **Release** always returns 0.

See Also

Reference

CComPolyObject Class

CComPolyObject::AddRef

Other Resources

CComPolyObject Members

CComPolyObject Data Members

For information about the data members in **CComPolyObject**, see CComPolyObject Members.

CComPolyObject::m_contained

A CComContainedObject object derived from your class.

```
CComContainedObject< contained > m_contained;
```

Parameters

contained

[in] Your class, derived from CComObjectRoot or CComObjectRootEx, as well as from any other interfaces you want to support on the object.

Remarks

IUnknown calls through **m_contained** are delegated to the outer unknown if the object is aggregated, or to the **IUnknown** of this object if the object is not aggregated.

See Also

Reference

CComPolyObject Class

Other Resources

CComPolyObject Members

CComPolyObject Static Functions

For information about the static functions in **CComPolyObject**, see CComPolyObject Members.

CComPolyObject::CreateInstance

Allows you to create a new **CComPolyObject<** *contained* **>** object without the overhead of CoCreateInstance.

```
static HRESULT WINAPI CreateInstance(
   LPUNKNOWN pUnkOuter,
   CComPolyObject< contained >** pp
);
```

Parameters

pp

[out] A pointer to a **CComPolyObject<** *contained* **>** pointer. If **CreateInstance** is unsuccessful, *pp* is set to **NULL**.

Return Value

A standard **HRESULT** value.

Remarks

The object returned has a reference count of zero, so call **AddRef** immediately, then use **Release** to free the reference on the object pointer when you're done.

If you don't need direct access to the object, but still want to create a new object without the overhead of **CoCreateInstance**, use CComCoClass::CreateInstance instead.

See Also

Reference

CComPolyObject Class

Other Resources

CComPolyObject Members

CComPtr Class

A smart pointer class for managing COM interface pointers.

```
template<
   class T
>
class CComPtr
```

Parameters

T

A COM interface specifying the type of pointer to be stored.

Remarks

ATL uses **CComPtr** and CComQIPtr to manage COM interface pointers. Both are derived from CComPtrBase, and both perform automatic reference counting.

The **CComPtr** and CComQIPtr classes can help eliminate memory leaks by performing automatic reference counting. The following functions both perform the same logical operations; however, note how the second version may be less error-prone by using the **CComPtr** class:

```
// Error-checking routine that performs manual lifetime management
// of a COM IErrorInfo object
HRESULT CheckComError_Manual()
{
   HRESULT hr;
   CComBSTR bstrDescription;
   CComBSTR bstrSource;
   CComBSTR bstrHelpFile;

   IErrorInfo* pErrInfo = NULL; // naked COM interface pointer
   hr = ::GetErrorInfo( 0, &pErrInfo );
   if( hr != S_OK )
      return hr;

   hr = pErrInfo->GetDescription( &bstrDescription );
   if( FAILED( hr ) )
   {
      pErrInfo->Release();   // must release interface pointer before returning
      return hr;
   }
   hr = pErrInfo->GetSource( &bstrSource );
   if( FAILED( hr ) )
   {
      pErrInfo->Release();   // must release interface pointer before returning
      return hr;
   }
   hr = pErrInfo->GetHelpFile( &bstrHelpFile );
   if( FAILED( hr ) )
   {
      pErrInfo->Release();   // must release interface pointer before returning
      return hr;
   }

   pErrInfo->Release();      // must release interface pointer before returning
   return S_OK;
}
```

```
// Error-checking routine that performs automatic lifetime management
// of a COM IErrorInfo object through a CComPtr smart pointer object
HRESULT CheckComError_SmartPtr()
```

```
{
   HRESULT hr;
   CComBSTR bstrDescription;
   CComBSTR bstrSource;
   CComBSTR bstrHelpFile;

   CComPtr<IErrorInfo> pErrInfo;
   hr = ::GetErrorInfo( 0, &pErrInfo );
   if( hr != S_OK )
      return hr;

   hr = pErrInfo->GetDescription( &bstrDescription );
   if( FAILED( hr ) )
      return hr;
   hr = pErrInfo->GetSource( &bstrSource );
   if( FAILED( hr ) )
      return hr;
   hr = pErrInfo->GetHelpFile( &bstrHelpFile );
   if( FAILED( hr ) )
      return hr;

   return S_OK;
}   // CComPtr will auto-release underlying IErrorInfo interface pointer as needed
```

In Debug builds, link atlsd.lib for code tracing.

Requirements

Header: atlbase.h

See Also

Reference

CComPtr::CComPtr

CComQIPtr::CComQIPtr

Other Resources

CComPtr Members

ATL Class Overview

CComPtr Members

Methods

CComPtr	The constructor.

Operators

operator =	Assigns a pointer to the member pointer.

See Also

Reference

CComPtr Class

CComPtr Methods

For information about the methods in **CComPtr**, see CComPtr Members.

CComPtr::CComPtr

The constructor.

```
CComPtr( ) throw ( );
CComPtr(
   T* lp
) throw ( );
CComPtr (
   const CComPtr< T >& lp
) throw ( );
```

Parameters

lp

Used to initialize the interface pointer.

T

A COM interface.

See Also

Reference

CComPtr Class

Other Resources

CComPtr Members

CComPtr Operators

For information about the operators in **CComPtr**, see CComPtr Members.

CComPtr::operator =

Assignment operator.

```
T* operator =(
      T* lp
) throw ( );
T* operator =(
   const CComPtr< T >& lp
) throw ( );
```

Return Value

Returns a pointer to the updated **CComPtr** object

See Also

Reference

CComPtr Class

CComPtrBase Class

Other Resources

CComPtr Members

CComPtrBase Class

This class provides a basis for smart pointer classes using COM-based memory routines.

```
template <
   class T
> class CComPtrBase
```

Parameters

T

The object type to be referenced by the smart pointer.

Remarks

This class provides the basis for other smart pointers which use COM memory management routines, such as CComQIPtr and CComPtr. The derived classes add their own constructors and operators, but rely on the methods provided by **CComPtrBase**.

Requirements

Header: atlcomcli.h

See Also

Other Resources

CComPtrBase Members

ATL Class Overview

CComPtrBase Members

Methods

Advise	Call this method to create a connection between the **CComPtrBase**'s connection point and a client's sink.
Attach	Call this method to take ownership of an existing pointer.
~CComPtrBase	The destructor.
CoCreateInstance	Call this method to create an object of the class associated with a specified Class ID or Program ID.
CopyTo	Call this method to copy the **CComPtrBase** pointer to another pointer variable.
Detach	Call this method to release ownership of a pointer.
IsEqualObject	Call this method to check if the specified **IUnknown** points to the same object associated with the **CComPtrBase** object.
QueryInterface	Call this method to return a pointer to a specified interface.
Release	Call this method to release the interface.
SetSite	Call this method to set the site of the **CComPtrBase** object to the **IUnknown** of the parent object.

Operators

operator T*	The cast operator.
operator <	The less-than operator.
operator !	The NOT operator.
operator &	The & operator.
operator *	The * operator.
operator ->	The pointer-to-members operator.
operator ==	The equality operator.

Data Members

p	The pointer data member variable.

See Also

Reference

CComPtrBase Class

CComPtrBase Methods

For information about the methods in **CComPtrBase**, see CComPtrBase Members.

CComPtrBase::Advise

Call this method to create a connection between the **CComPtrBase**'s connection point and a client's sink.

```
HRESULT Advise(
   IUnknown* pUnk,
   const IID& iid,
   LPDWORD pdw
) throw( );
```

Parameters

pUnk

A pointer to the client's **IUnknown**.

iid

The GUID of the connection point. Typically, this is the same as the outgoing interface managed by the connection point.

pdw

A pointer to the cookie that uniquely identifies the connection.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

See AtlAdvise for more information.

See Also

Reference

CComPtrBase Class

Other Resources

CComPtrBase Members

CComPtrBase::Attach

Call this method to take ownership of an existing pointer.

```
void Attach(
   T* p2
) throw( );
```

Parameters

p2

The **CComPtrBase** object will take ownership of this pointer.

Remarks

Attach calls CComPtrBase::Release on the existing CComPtrBase::p member variable and then assigns *p2* to **CComPtrBase::p**. When a **CComPtrBase** object takes ownership of a pointer, it will automatically delete the pointer and any allocated data when it goes out of scope.

See Also

Reference

CComPtrBase Class

CComPtrBase::Detach

Other Resources

CComPtrBase Members

CComPtrBase::~CComPtrBase

The destructor.

```
~CComPtrBase( ) throw( );
```

Remarks

Releases the interface pointed to by **CComPtrBase**.

See Also

Reference

CComPtrBase Class

Other Resources

CComPtrBase Members

CComPtrBase::CoCreateInstance

Call this method to create an object of the class associated with a specified Class ID or Program ID.

```
HRESULT CoCreateInstance(
   LPCOLESTR szProgID,
   LPUNKNOWN pUnkOuter = NULL,
   DWORD dwClsContext = CLSCTX_ALL
) throw( );
HRESULT CoCreateInstance(
   REFCLSID rclsid,
   LPUNKNOWN pUnkOuter = NULL,
   DWORD dwClsContext = CLSCTX_ALL
) throw( );
```

Parameters

szProgID

Pointer to a ProgID, used to recover the CLSID.

pUnkOuter

If **NULL**, indicates that the object is not being created as part of an aggregate. If non-**NULL**, is a pointer to the aggregate object's **IUnknown** interface (the controlling **IUnknown**).

dwClsContext

Context in which the code that manages the newly created object will run.

rclsid

CLSID associated with the data and code that will be used to create the object.

Return Value

Returns S_OK on success, or REGDB_E_CLASSNOTREG, CLASS_E_NOAGGREGATION, or E_NOINTERFACE on failure. See CoCreateClassInstance for a description of these errors.

Remarks

If the first form of the method is called, CLSIDFromProgID is used to recover the CLSID. Both forms then call CoCreateClassInstance.

In debug builds, an assertion error will occur if CComPtrBase::p is not equal to NULL.

See Also

Reference

CComPtrBase Class

Other Resources

CComPtrBase Members

CComPtrBase::CopyTo

Call this method to copy the **CComPtrBase** pointer to another pointer variable.

```
HRESULT CopyTo(
   T** ppT
) throw( );
```

Parameters

ppT

Address of the variable which will receive the **CComPtrBase** pointer.

Return Value

Returns S_OK on success, E_POINTER on failure.

Remarks

Copies the **CComPtrBase** pointer to *ppT*. The reference count on the CComPtrBase::p member variable is incremented.

An error HRESULT will be returned if *ppT* is equal to NULL. In debug builds, an assertion error will occur if *ppT* is equal to NULL.

See Also

Reference

CComPtrBase Class

Other Resources

CComPtrBase Members

CComPtrBase::Detach

Call this method to release ownership of a pointer.

```
T* Detach( ) throw( );
```

Return Value

Returns a copy of the pointer.

Remarks

Releases ownership of a pointer, sets the CComPtrBase::p data member variable to NULL, and returns a copy of the pointer.

See Also

Reference

CComPtrBase Class

CComPtrBase::Attach

Other Resources

CComPtrBase Members

CComPtrBase::IsEqualObject

Call this method to check if the specified **IUnknown** points to the same object associated with the **CComPtrBase** object.

```
bool IsEqualObject(
   IUnknown* pOther
) throw( );
```

Parameters

pOther

The **IUnknown** * to compare.

Return Value

Returns true if the objects are identical, false otherwise.

See Also

Reference

CComPtrBase Class

Other Resources

CComPtrBase Members

CComPtrBase::QueryInterface

Call this method to return a pointer to a specified interface.

```
template <
   class Q
> HRESULT QueryInterface(Q** pp ) const throw( );
```

Parameters

Q

The object type whose interface pointer is required.

pp

Address of output variable that receives the requested interface pointer.

Return Value

Returns S_OK on success, or E_NOINTERFACE on failure.

Remarks

This method calls IUnknown::QueryInterface.

In debug builds, an assertion error will occur if *pp* is not equal to NULL.

See Also

Reference

CComPtrBase Class

CComPtrBase::SetSite

Other Resources

CComPtrBase Members

CComPtrBase::Release

Call this method to release the interface.

```
void Release( ) throw( );
```

Remarks

The interface is released, and CComPtrBase::p is set to NULL.

See Also

Reference

CComPtrBase Class

Other Resources

CComPtrBase Members

CComPtrBase::SetSite

Call this method to set the site of the **CComPtrBase** object to the **IUnknown** of the parent object.

```
HRESULT SetSite(
   IUnknown* punkParent
) throw( );
```

Parameters

punkParent

A pointer to the **IUnknown** interface of the parent.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method calls AtlSetChildSite.

See Also

Reference

CComPtrBase Class

CComPtrBase::QueryInterface

Other Resources

CComPtrBase Members

CComPtrBase Operators

For information about the operators in **CComPtrBase**, see CComPtrBase Members.

CComPtrBase::operator !

The NOT operator.

```
bool operator !( ) const throw( );
```

Return Value

Returns true if the **CComHeapPtr** pointer is equal to NULL, false otherwise.

See Also

Reference

CComPtrBase Class

Other Resources

CComPtrBase Members

CComPtrBase::operator &

The & operator.

```
T ** operator &( ) throw( );
```

Return Value

Returns the address of the object pointed to by the **CComPtrBase** object.

See Also

Reference

CComPtrBase Class

Other Resources

CComPtrBase Members

CComPtrBase::operator *

The * operator.

```
T & operator *( ) const throw( );
```

Return Value

Returns the value of CComPtrBase::p; that is, a pointer to the object referenced by the **CComPtrBase** object.

If debug builds, an assertion error will occur if CComPtrBase::p is not equal to NULL.

See Also

Reference

CComPtrBase Class

Other Resources

CComPtrBase Members

CComPtrBase::operator ==

The equality operator.

```
bool operator ==(
   T * pT
) const throw( );
```

Parameters

pT

A pointer to an object.

Return Value

Returns true if **CComPtrBase** and *pT* point to the same object, false otherwise.

See Also

Reference

CComPtrBase Class

Other Resources

CComPtrBase Members

CComPtrBase::operator ->

The pointer-to-member operator.

```
_NoAddRefReleaseOnCComPtr< T > * operator ->( ) const throw( );
```

Return Value

Returns the value of the CComPtrBase::p data member variable.

Remarks

Use this operator to call a method in a class pointed to by the **CComPtrBase** object. In debug builds, an assertion failure will occur if the **CComPtrBase** data member points to NULL.

See Also

Reference

CComPtrBase Class

Other Resources

CComPtrBase Members

CComPtrBase::operator <

The less-than operator.

```
bool operator <(
   T * pT
) const throw( );
```

Parameters

pT

A pointer to an object.

Return Value

Returns true if the pointer managed by current object is less than the pointer to which it is being compared.

See Also

Reference

CComPtrBase Class

Other Resources

CComPtrBase Members

CComPtrBase::operator T*

The cast operator.

```
operator T*( ) const throw( );
```

Remarks

Returns a pointer to the object data type defined in the class template.

See Also

Reference

CComPtrBase Class

Other Resources

CComPtrBase Members

CComPtrBase Data Members

For information about the data members in **CComPtrBase**, see CComPtrBase Members.

CComPtrBase::p

The pointer data member variable.

```
T * p;
```

Remarks

This member variable holds the pointer information.

See Also

Reference

CComPtrBase Class

Other Resources

CComPtrBase Members

CComQIPtr Class

A smart pointer class for managing COM interface pointers.

```
template<
   class T,
   const IID* piid = &__uuidof(T)
>
class CComQIPtr: public CComPtr<T>
```

Parameters

T

A COM interface specifying the type of pointer to be stored.

piid

A pointer to the IID of *T*.

Remarks

ATL uses **CComQIPtr** and CComPtr to manage COM interface pointers, both of which derive from CComPtrBase. Both classes perform automatic reference counting through calls to **AddRef** and **Release**. Overloaded operators handle pointer operations.

Requirements

Header: atlcomcli.h

See Also

Reference

CComPtr::CComPtr
CComQIPtr::CComQIPtr
CComPtrBase Class
CComQIPtrElementTraits Class

Other Resources

CComQIPtr Members
ATL Class Overview

CComQIPtr Members

Methods

CComQIPtr	Constructor.

Operators

operator =	Assigns a pointer to the member pointer.

See Also

Reference

CComQIPtr Class

CComQIPtr Methods

For information about the methods in **CComQIPtr**, see CComQIPtr Members.

CComQIPtr::CComQIPtr

The constructor.

```
CComQIPtr( ) throw( );
CComQIPtr(
      T* lp
) throw( );
CComQIPtr(
   IUnknown* lp
) throw( );
CComQIPtr(
   const CComQIPtr< T,
   piid >& lp
) throw( );
```

Parameters

lp

Used to initialize the interface pointer.

T

A COM interface.

piid

A pointer to the IID of *T*.

See Also

Reference

CComQIPtr Class

Other Resources

CComQIPtr Members

CComQIPtr Operators

For information about the operators in **CComQIPtr**, see CComQIPtr Members.

CComQIPtr::operator =

The assignment operator.

```
T* operator =(
      T* lp
) throw( );
T* operator =(
   const CComQIPtr< T,
      piid >& lp
) throw( );
T* operator =(
   IUnknown* lp
) throw( );
```

Parameters

lp

Used to initialize the interface pointer.

T

A COM interface.

piid

A pointer to the IID of *T*.

Return Value

Returns a pointer to the updated **CComQIPtr** object.

See Also

Reference

CComQIPtr Class

CComPtrBase Class

Other Resources

CComQIPtr Members

CComQIPtrElementTraits Class

This class provides methods, static functions, and typedefs useful when creating collections of COM interface pointers.

```
template<
   typename I,
   const IID* piid = & __uuidof( I )
>
class CComQIPtrElementTraits : public CDefaultElementTraits<
   ATL::CComQIPtr< I, piid >
>
```

Parameters

I

A COM interface specifying the type of pointer to be stored.

piid

A pointer to the IID of *I*.

Remarks

This class derives methods and provides a typedef useful when creating a collection class of CComQIPtr COM interface pointer objects. This class is utilized by both the CInterfaceArray and CInterfaceList classes.

For more information, see ATL Collection Classes.

Requirements

Header: atlcoll.h

See Also

Reference

CDefaultElementTraits Class

Other Resources

CComQIPtrElementTraits Members

ATL Class Overview

CComQIPtrElementTraits Members

Typedefs

INARGTYPE	The data type to use for adding elements to the collection class object.

See Also

Reference

CComQIPtrElementTraits Class

CComQIPtrElementTraits Typedefs

For information about the typedefs in **CComQIPtrElementTraits**, see CComQIPtrElementTraits Members.

CComQIPtrElementTraits::INARGTYPE

The data type to use for adding elements to the collection class object.

```
typedef I * INARGTYPE;
```

See Also

Reference

CComQIPtrElementTraits Class

Other Resources

CComQIPtrElementTraits Members

CComSafeArray Class

This class is a wrapper for the **SAFEARRAY** structure.

```
template <
   typename T,
   VARTYPE _vartype = _ATL_AutomationType< T >::type
>
class CComSafeArray
```

Parameters

T

The type of data to be stored in the array.

Remarks

CComSafeArray provides a wrapper for the SAFEARRAY Data Type class, making it a simple matter to create and manage single- and multidimensional arrays of almost any of the VARIANT-supported types.

CComSafeArray simplifies passing arrays between processes, and in addition provides extra security by checking array index values against upper and lower bounds.

The lower bound of a **CComSafeArray** can start at any user-defined value; however, arrays that are accessed through C++ should use a lower bound of 0. Other languages such as Visual Basic may use other bounding values (for example, -10 to 10).

Use CComSafeArray::Create to create a **CComSafeArray** object, and CComSafeArray::Destroy to delete it.

A **CComSafeArray** can contain the following subset of VARIANT data types:

VARTYPE	Description
VT_I1	char
VT_I2	short
VT_I4	int
VT_I4	long
VT_I8	longlong
VT_UI1	byte
VT_UI2	ushort
VT_UI4	uint
VT_UI4	ulong
VT_UI8	ulonglong
VT_R4	float
VT_R8	double
VT_DECIMAL	decimal pointer

VT_VARIANT	variant pointer
VT_CY	Currency data type

Requirements

Header: atlsafe.h

Example

```
// Create a multidimensional array,
// then write and read elements

#include "stdafx.h"
#include "atlsafe.h"

int main(int argc, char* argv[])
{
   // Define an array of character pointers
   CComSafeArray<char> *pSar;

   char cElement;
   char cTable[2][3] = {'A','B','C','D','E','F'};

   // Declare the variable used to store the
   // array indexes
   LONG aIndex[2];

   // Define the array bound structure
   CComSafeArrayBound bound[2];
   bound[0].SetCount(3);
   bound[0].SetLowerBound(0);
   bound[1].SetCount(3);
   bound[1].SetLowerBound(0);

   // Create a new 2 dimensional array
   // each dimension size is 3
   pSar = new CComSafeArray<char>(bound,2);

   // Use MultiDimSetAt to store characters in the array
   for (int x = 0; x < 2; x++)
   {
      for (int y = 0; y < 3; y++)
      {
         aIndex[0] = x;
         aIndex[1] = y;
         HRESULT hr = pSar->MultiDimSetAt(aIndex,cTable[x][y]);
         ATLASSERT(hr == S_OK);
      }
   }
   // Use MultiDimGetAt to retrieve characters in the array
   for (int x = 0; x < 2; x++)
   {
      for (int y = 0; y < 3; y++)
      {
         aIndex[0]=x;
         aIndex[1]=y;
         HRESULT hr = pSar->MultiDimGetAt(aIndex,cElement);
         ATLASSERT(hr == S_OK);
         ATLASSERT(cElement == cTable[x][y]);
      }
   }
   return 0;
}
```

Smart Device Developer Notes

Only the following VARTYPEs are fully supported: VT_BSTR, VT_VARIANT, VT_UNKNOWN, and VT_DISPATCH.

See Also

Reference

SAFEARRAY Data Type

CComSafeArray::Create

CComSafeArray::Destroy

Other Resources

CComSafeArray Members

ATL Class Overview

CComSafeArray Members

Methods

~CComSafeArray	The destructor.
Add	Adds one or more elements, or a **SAFEARRAY** structure, to a **CComSafeArray**.
Attach	Attaches a **SAFEARRAY** structure to a **CComSafeArray** object.
CComSafeArray	The constructor.
CopyFrom	Copies the contents of a **SAFEARRAY** structure into the **CComSafeArray** object.
CopyTo	Creates a copy of the **CComSafeArray** object.
Create	Creates a **CComSafeArray** object.
Destroy	Destroys a **CComSafeArray** object.
Detach	Detaches a **SAFEARRAY** from a **CComSafeArray** object.
GetAt	Retrieves a single element from a single-dimensional array.
GetCount	Returns the number of elements in the array.
GetDimensions	Returns the number of dimensions in the array.
GetLowerBound	Returns the lower bound for a given dimension of the array.
GetSafeArrayPtr	Returns the address of the **m_psa** data member.
GetType	Returns the type of data stored in the array.
GetUpperBound	Returns the upper bound for any dimension of the array.
IsSizable	Tests if a **CComSafeArray** object can be resized.
MultiDimGetAt	Retrieves a single element from a multidimensional array.
MultiDimSetAt	Sets the value of an element in a multidimensional array.
Resize	Resizes a **CComSafeArray** object.
SetAt	Sets the value of an element in a single-dimensional array.

Operators

operator []	Retrieves an element from the array.
operator =	Assignment operator.
operator LPSAFEARRAY	Casts a value to a **SAFEARRAY** pointer.

Data Members

m_psa	This data member holds the address of the **SAFEARRAY** structure.

See Also

Reference

CComSafeArray Class

CComSafeArray Methods

For information about the methods in **CComSafeArray**, see CComSafeArray Members.

CComSafeArray::Add

Adds one or more elements, or a **SAFEARRAY** structure, to a **CComSafeArray**.

```
HRESULT Add(
   const SAFEARRAY * psaSrc
);
HRESULT Add(
   ULONG ulCount,
   const T * pT,
      BOOL bCopy = TRUE
);
HRESULT Add(
   const T& t,
      BOOL bCopy = TRUE
);
```

Parameters

psaSrc

A pointer to a **SAFEARRAY** object.

ulCount

The number of objects to add to the array.

pT

A pointer to one or more objects to be added to the array.

t

A reference to the object to be added to the array.

bCopy

Indicates whether a copy of the data should be created. The default value is **TRUE**.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The new objects are appended to the end of the existing **SAFEARRAY** object. Adding an object to a multidimensional **SAFEARRAY** object is not supported. When adding an existing array of objects, both arrays must contain elements of the same type.

The *bCopy* flag is taken into account when elements of type **BSTR** or **VARIANT** are added to an array. The default value of **TRUE** ensures that a new copy is made of the data when the element is added to the array.

Smart Device Developer Notes

CComSafeArray::Add(SAFEARRAY *) is supported for only VT_BSTR, VT_VARIANT, VT_UNKNOWN, and VT_DISPATCH VARTYPEs.

See Also

Reference

CComSafeArray Class

Other Resources

CComSafeArray Members

CComSafeArray::Attach

Attaches a **SAFEARRAY** structure to a **CComSafeArray** object.

```
HRESULT Attach(
   const SAFEARRAY * psaSrc
);
```

Parameters

psaSrc

A pointer to the **SAFEARRAY** structure.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Attaches a **SAFEARRAY** structure to a **CComSafeArray** object, making the existing **CComSafeArray** methods available.

Smart Device Developer Notes

CComSafeArray::Attach(SAFEARRAY *) is supported for only VT_BSTR, VT_VARIANT, VT_UNKNOWN, and VT_DISPATCH VARTYPEs.

See Also

Reference

CComSafeArray Class

CComSafeArray::Detach

Other Resources

CComSafeArray Members

CComSafeArray::CComSafeArray

The constructor.

```
CComSafeArray( );
CComSafeArray(
   const SAFEARRAYBOUND& bound
);
CComSafeArray(
   ULONG ulCount,
   LONG lLBound = 0
);
CComSafeArray(
   const SAFEARRAYBOUND * pBound,
   UINT uDims = 1
);
CComSafeArray(
   const CComSafeArray& saSrc
);
CComSafeArray(
   const SAFEARRAY& saSrc
);
CComSafeArray(
   const SAFEARRAY * psaSrc
);
```

Parameters

bound

A **SAFEARRAYBOUND** structure.

ulCount

The number of elements in the array.

lLBound

The lower bound value; that is, the index of the first element in the array.

pBound

A pointer to a **SAFEARRAYBOUND** structure.

uDims

The count of dimensions in the array.

saSrc

A reference to a **SAFEARRAY** structure or **CComSafeArray** object. In either case the constructor uses this reference to make a copy of the array, so the array is not referenced after construction.

psaSrc

A pointer to a **SAFEARRAY** structure. The constructor uses this address to make a copy of the array, so the array is not referenced after construction.

Remarks

Creates a **CComSafeArray** object.

Smart Device Developer Notes

CComSafeArray::CComSafeArray (SAFEARRAY *) and CComSafeArray::CComSafeArray (CComSafeArray) are supported for only VT_BSTR, VT_VARIANT, VT_UNKNOWN, and VT_DISPATCH VARTYPEs.

See Also

Reference

CComSafeArray Class
CComSafeArray::~CComSafeArray
Other Resources
CComSafeArray Members

CComSafeArray::~CComSafeArray

The destructor.

```
~CComSafeArray( );
```

Remarks

Frees all allocated resources. In the event of an error, this method calls **AtlThrow** with an HRESULT describing the error.

See Also

Reference

CComSafeArray Class

CComSafeArray::CComSafeArray

Other Resources

CComSafeArray Members

CComSafeArray::CopyFrom

Copies the contents of a **SAFEARRAY** structure into the **CComSafeArray** object.

```
HRESULT CopyFrom(
   LPSAFEARRAY * ppArray
);
```

Parameters

ppArray

Pointer to the **SAFEARRAY** to copy.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method copies the contents of a **SAFEARRAY** into the current **CComSafeArray** object. The existing contents of the array are replaced.

Smart Device Developer Notes

CComSafeArray::CopyFrom(SAFEARRAY *) is supported for only VT_BSTR, VT_VARIANT, VT_UNKNOWN, and VT_DISPATCH VARTYPEs.

See Also

Reference

CComSafeArray Class

CComSafeArray::CopyTo

Other Resources

CComSafeArray Members

CComSafeArray::CopyTo

Creates a copy of the **CComSafeArray** object.

```
HRESULT CopyTo(
   LPSAFEARRAY * ppArray
);
```

Parameters

ppArray

A pointer to a location in which to create the new **SAFEARRAY**.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method copies the contents of a **CComSafeArray** object into a **SAFEARRAY** structure.

See Also

Reference

CComSafeArray Class

CComSafeArray::CopyFrom

Other Resources

CComSafeArray Members

CComSafeArray::Create

Creates a **CComSafeArray**.

```
HRESULT Create(
   const SAFEARRAYBOUND * pBound,
   UINT uDims = 1
);
HRESULT Create(
   ULONG ulCount = 0,
   LONG lLBound = 0
);
```

Parameters

pBound

A pointer to a **SAFEARRAYBOUND** object.

uDims

The number of dimensions in the array.

ulCount

The number of elements in the array.

lLBound

The lower bound value; that is, the index of the first element in the array.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

A **CComSafeArray** object can be created from an existing **SAFEARRAYBOUND** structure and the number of dimensions, or by specifying the number of elements in the array and the lower bound. If the array is to be accessed from Visual C++, the lower bound should be 0. Other languages may allow other values for the lower bound (for example, Visual Basic supports arrays with elements with a range such as -10 to 10).

See Also

Reference

CComSafeArray Class
CComSafeArray::Destroy

Other Resources

CComSafeArray Members

CComSafeArray::Destroy

Destroys a **CComSafeArray** object.

```
HRESULT Destroy( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Destroys an existing **CComSafeArray** object and all of the data it contains.

See Also

Reference

CComSafeArray Class

CComSafeArray::Create

Other Resources

CComSafeArray Members

CComSafeArray::Detach

Detaches a **SAFEARRAY** from a **CComSafeArray** object.

```
LPSAFEARRAY Detach( );
```

Return Value

Returns a pointer to a **SAFEARRAY** object.

Remarks

This method detaches the **SAFEARRAY** object from the **CComSafeArray** object.

See Also

Reference

CComSafeArray Class

CComSafeArray::Attach

Other Resources

CComSafeArray Members

CComSafeArray::GetAt

Retrieves a single element from a single-dimensional array.

```
T& GetAt(
   LONG lIndex
) const;
```

Parameters

lIndex

The index number of the value in the array to return.

Return Value

Returns a reference to the required array element.

See Also

Reference

CComSafeArray Class
CComSafeArray::MultiDimGetAt
CComSafeArray::SetAt

Other Resources

CComSafeArray Members

CComSafeArray::GetCount

Returns the number of elements in the array.

```
ULONG GetCount(
   UINT uDim = 0
) const;
```

Parameters

uDim

The array dimension.

Return Value

Returns the number of elements in the array.

Remarks

When used with a multidimensional array, this method will return the number of elements in a specific dimension only.

See Also

Reference

CComSafeArray Class
CComSafeArray::GetLowerBound
CComSafeArray::GetUpperBound
CComSafeArray::GetDimensions

Other Resources

CComSafeArray Members

CComSafeArray::GetDimensions

Returns the number of dimensions in the array.

```
UINT GetDimensions( ) const;
```

Return Value

Returns the number of dimensions in the array.

See Also

Reference

CComSafeArray Class
CComSafeArray::GetLowerBound
CComSafeArray::GetUpperBound
CComSafeArray::GetCount

Other Resources

CComSafeArray Members

CComSafeArray::GetLowerBound

Returns the lower bound for a given dimension of the array.

```
LONG GetLowerBound(
   UINT uDim = 0
) const;
```

Parameters

uDim

The array dimension for which to get the lower bound. If omitted, the default is 0.

Return Value

Returns the lower bound.

Remarks

If the lower bound is 0, this indicates a C-like array whose first element is element number 0. In the event of an error, for example, an invalid dimension argument, this method calls **AtlThrow** with an HRESULT describing the error.

See Also

Reference

CComSafeArray Class
CComSafeArray::GetCount
CComSafeArray::GetUpperBound
CComSafeArray::GetDimensions

Other Resources

CComSafeArray Members

CComSafeArray::GetSafeArrayPtr

Returns the address of the **m_psa** data member.

```
LPSAFEARRAY* GetSafeArrayPtr( ) throw( );
```

Return Value

Returns a pointer to the CComSafeArray::m_psa data member.

See Also

Reference

CComSafeArray Class

CComSafeArray::m_psa

Other Resources

CComSafeArray Members

CComSafeArray::GetType

Returns the type of data stored in the array.

```
VARTYPE GetType( ) const;
```

Return Value

Returns the type of data stored in the array, which could be any of the following types:

VARTYPE	Description
VT_I1	char
VT_I2	short
VT_I4	int
VT_I4	long
VT_I8	longlong
VT_UI1	byte
VT_UI2	ushort
VT_UI4	uint
VT_UI4	ulong
VT_UI8	ulonglong
VT_R4	float
VT_R8	double
VT_DECIMAL	decimal pointer
VT_VARIANT	variant pointer
VT_CY	Currency data type

See Also

Reference

CComSafeArray Class

Other Resources

CComSafeArray Members

CComSafeArray::GetUpperBound

Returns the upper bound for any dimension of the array.

```
LONG GetUpperBound(
   UINT uDim = 0
) const;
```

Parameters

uDim

The array dimension for which to get the upper bound. If omitted, the default is 0.

Return Value

Returns the upper bound. This value is inclusive, the maximum valid index for this dimension.

Remarks

In the event of an error, for example, an invalid dimension argument, this method calls **AtlThrow** with an HRESULT describing the error.

See Also

Reference

CComSafeArray Class
CComSafeArray::GetLowerBound
CComSafeArray::GetCount

Other Resources

CComSafeArray Members

CComSafeArray::IsSizable

Tests if a **CComSafeArray** object can be resized.

```
bool IsSizable( ) const;
```

Return Value

Returns **true** if the **CComSafeArray** can be resized, **false** if it cannot.

See Also

Reference

CComSafeArray Class

CComSafeArray::Resize

Other Resources

CComSafeArray Members

CComSafeArray::MultiDimGetAt

Retrieves a single element from a multidimensional array.

```
HRESULT MultiDimGetAt(
   const LONG * alIndex,
   T& t
);
```

Parameters

alIndex

Pointer to a vector of indexes for each dimension in the array. The leftmost (most significant) dimension is *alIndex*[0].

t

A reference to the data returned.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CComSafeArray Class
CComSafeArray::GetAt
CComSafeArray::SetAt
CComSafeArray::MultiDimSetAt

Other Resources

CComSafeArray Members

CComSafeArray::MultiDimSetAt

Sets the value of an element in a multidimensional array.

```
HRESULT MultiDimSetAt(
   const LONG * alIndex,
   const T& t
);
```

Parameters

alIndex

Pointer to a vector of indexes for each dimension in the array. The rightmost (least significant) dimension is *alIndex*[0].

T

Specifies the value of the new element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This is a multidimensional version of CComSafeArray::SetAt.

See Also

Reference

CComSafeArray Class
CComSafeArray::SetAt
CComSafeArray::GetAt
CComSafeArray::MultiDimGetAt

Other Resources

CComSafeArray Members

CComSafeArray::Resize

Resizes a **CComSafeArray** object.

```
HRESULT Resize(
   const SAFEARRAYBOUND * pBound
);
HRESULT Resize(
   ULONG ulCount,
   LONG lLBound = 0
);
```

Parameters

pBound

A pointer to a **SAFEARRAYBOUND** structure that contains information on the number of elements and the lower bound of an array.

ulCount

The requested number of objects in the resized array.

lLBound

The lower bound.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method only resizes the rightmost dimension. It will not resize arrays that return **IsResizable** as **false**.

See Also

Reference

CComSafeArray Class

CComSafeArray::IsSizable

Other Resources

CComSafeArray Members

CComSafeArray::SetAt

Sets the value of an element in a single-dimensional array.

```
HRESULT SetAt(
   LONG lIndex,
   const T& t,
      BOOL bCopy = TRUE
);
```

Parameters

lIndex

The index number of the array element to set.

t

The new value of the specified element.

bCopy

Indicates whether a copy of the data should be created. The default value is **TRUE**.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The *bCopy* flag is taken into account when elements of type **BSTR** or **VARIANT** are added to an array. The default value of **TRUE** ensures that a new copy is made of the data when the element is added to the array.

See Also

Reference

CComSafeArray Class
CComSafeArray::GetAt
CComSafeArray::MultiDimSetAt

Other Resources

CComSafeArray Members

CComSafeArray Operators

For information about the operators in **CComSafeArray**, see CComSafeArray Members.

CComSafeArray::operator []

Retrieves an element from the array.

```
T & operator [](
   LONG lIndex
) const;
T & operator [](
   int nIndex
) const;
```

Parameters

lIndex, nIndex

The index number of the required element in the array.

Return Value

Returns the appropriate array element.

Remarks

Performs a similar function to CComSafeArray::GetAt, however this operator only works with single-dimensional arrays.

See Also

Reference

CComSafeArray Class

CComSafeArray::GetAt

Other Resources

CComSafeArray Members

CComSafeArray::operator =

Assignment operator.

```
ATL::CComSafeArray< T > & operator =(
   const ATL::CComSafeArray & saSrc
);
ATL::CComSafeArray< T > & operator =(
   const SAFEARRAY * psaSrc
);
```

Parameters

saSrc

A reference to a **CComSafeArray** object.

psaSrc

A pointer to a **SAFEARRAY** object.

Return Value

Returns the type of data stored in the array.

Smart Device Developer Notes

CComSafeArray<T>& operator=(const SAFEARRAY *psaSrc) is supported for only VT_BSTR, VT_VARIANT, VT_UNKNOWN, and VT_DISPATCH VARTYPEs.

See Also

Reference

CComSafeArray Class

Other Resources

CComSafeArray Members

CComSafeArray::operator LPSAFEARRAY

Casts a value to a **SAFEARRAY** pointer.

```
operator LPSAFEARRAY( ) const;
```

Return Value

Casts a value to a **SAFEARRAY** pointer.

See Also

Reference

CComSafeArray Class

Other Resources

CComSafeArray Members

CComSafeArray Data Members

For information about the data members in **CComSafeArray**, see CComSafeArray Members.

CComSafeArray::m_psa

Holds the address of the **SAFEARRAY** structure accessed.

```
LPSAFEARRAY m_psa;
```

See Also

Reference

CComSafeArray Class

Other Resources

CComSafeArray Members

CComSafeArrayBound Class

This class is a wrapper for a SAFEARRAYBOUND structure.

```
class CComSafeArrayBound : public SAFEARRAYBOUND
```

Remarks

This class is a wrapper for the **SAFEARRAYBOUND** structure used by CComSafeArray. It provides methods for querying and setting the upper and lower bounds of a single dimension of a **CComSafeArray** object and the number of elements it contains. A multidimensional **CComSafeArray** object uses an array of **CComSafeArrayBound** objects, one for each dimension. Therefore, when using methods such as GetCount, be aware that this method will not return the total number of elements in a multidimensional array.

Header: atlsafe.h

See Also

Other Resources

CComSafeArrayBound Members

ATL Class Overview

CComSafeArrayBound Members

Methods

CComSafeArrayBound	The constructor.
GetCount	Call this method to return the number of elements.
GetLowerBound	Call this method to return the lower bound.
GetUpperBound	Call this method to return the upper bound.
SetCount	Call this method to set the number of elements.
SetLowerBound	Call this method to set the lower bound.

Operators

operator =	Sets the **CComSafeArrayBound** to a new value.

See Also

Reference

CComSafeArrayBound Class

CComSafeArrayBound Methods

For information about the methods in **CComSafeArrayBound**, see CComSafeArrayBound Members.

CComSafeArrayBound::CComSafeArrayBound

The constructor.

```
CComSafeArrayBound(
   ULONG ulCount = 0,
   LONG lLowerBound = 0
) throw( );
```

Parameters

ulCount

The number of elements in the array.

lLowerBound

The lower bound from which the array is numbered.

Remarks

If the array is to be accessed from a Visual C++ program, it is recommended that the lower bound be defined as 0. It may be preferable to use a different lower bound value if the array is to be used with other languages, such as Visual Basic.

See Also

Reference

CComSafeArrayBound Class

Other Resources

CComSafeArrayBound Members

CComSafeArrayBound::GetCount

Call this method to return the number of elements.

```
ULONG GetCount( ) const throw( );
```

Return Value

Returns the number of elements.

Remarks

If the associated **CComSafeArray** object represents a multidimensional array, this method will only return the total number of elements in the rightmost dimension. Use CComSafeArray::GetCount to obtain the total number of elements.

See Also

Reference

CComSafeArrayBound Class

CComSafeArray::GetCount

Other Resources

CComSafeArrayBound Members

CComSafeArrayBound::GetLowerBound

Call this method to return the lower bound.

```
LONG GetLowerBound( ) const throw( );
```

Return Value

Returns the lower bound of the **CComSafeArrayBound** object.

See Also

Reference

CComSafeArrayBound Class
CComSafeArrayBound::GetUpperBound
CComSafeArrayBound::GetCount
CComSafeArrayBound::SetLowerBound

Other Resources

CComSafeArrayBound Members

CComSafeArrayBound::GetUpperBound

Call this method to return the upper bound.

```
LONG GetUpperBound( ) const throw( );
```

Return Value

Returns the upper bound of the **CComSafeArrayBound** object.

Remarks

The upper bound depends on the number of elements and the lower bound value. For example, if the lower bound is 0 and the number of elements is 10, the upper bound will automatically be set to 9.

See Also

Reference

CComSafeArrayBound Class
CComSafeArrayBound::GetCount
CComSafeArrayBound::GetLowerBound

Other Resources

CComSafeArrayBound Members

CComSafeArrayBound::SetCount

Call this method to set the number of elements.

```
ULONG SetCount(
   ULONG ulCount
) throw( );
```

Parameters

ulCount

The number of elements.

Return Value

Returns the number of elements in the **CComSafeArrayBound** object.

See Also

Reference

CComSafeArrayBound Class
CComSafeArrayBound::GetCount
CComSafeArrayBound::SetLowerBound

Other Resources

CComSafeArrayBound Members

CComSafeArrayBound::SetLowerBound

Call this method to set the lower bound.

```
LONG SetLowerBound(
   LONG lLowerBound
) throw( );
```

Parameters

lLowerBound

The lower bound.

Return Value

Returns the new lower bound of the **CComSafeArrayBound** object.

Remarks

If the array is to be accessed from a Visual C++ program, it is recommended that the lower bound be defined as 0. It may be preferable to use a different lower bound value if the array is to be used with other languages, such as Visual Basic.

The upper bound depends on the number of elements and the lower bound value. For example, if the lower bound is 0 and the number of elements is 10, the upper bound will automatically be set to 9.

See Also

Reference

CComSafeArrayBound Class
CComSafeArrayBound::GetUpperBound
CComSafeArrayBound::GetLowerBound
CComSafeArrayBound::SetCount

Other Resources

CComSafeArrayBound Members

CComSafeArrayBound Operators

For information about the operators in **CComSafeArrayBound**, see CComSafeArrayBound Members.

CComSafeArrayBound::operator =

Sets the **CComSafeArrayBound** to a new value.

```
CComSafeArrayBound & operator =(
   const CComSafeArrayBound & bound
) throw( );
CComSafeArrayBound & operator =(
   ULONG ulCount
) throw( );
```

Parameters

bound

A **CComSafeArrayBound** object.

ulCount

The number of elements.

Return Value

Returns a pointer to the **CComSafeArrayBound** object.

Remarks

The **CComSafeArrayBound** object can be assigned using an existing **CComSafeArrayBound**, or by supplying the number of elements, in which case the lower bound is set to 0 by default.

See Also

Reference

CComSafeArrayBound Class

Other Resources

CComSafeArrayBound Members

CComSafeDeleteCriticalSection Class

This class provides methods for obtaining and releasing ownership of a critical section object.

```
class CComSafeDeleteCriticalSection : public CComCriticalSection
```

Remarks

CComSafeDeleteCriticalSection derives from the class CComCriticalSection. However, **CComSafeDeleteCriticalSection** provides additional safety mechanisms over CComCriticalSection.

When an instance of **CComSafeDeleteCriticalSection** goes out of scope or is explicitly deleted from memory, the underlying critical section object will automatically be cleaned up if it is still valid. In addition, the CComSafeDeleteCriticalSection::Term method will exit gracefully if the underlying critical section object has not yet been allocated or has already been released from memory.

See CComCriticalSection for more information on critical section helper classes.

Requirements

Header: atlcore.h

See Also

Reference

CComCriticalSection Class

Other Resources

CComSafeDeleteCriticalSection Members

ATL Class Overview

CComSafeDeleteCriticalSection Members

Methods

CComSafeDeleteCriticalSection	The constructor.
~CComSafeDeleteCriticalSection	The destructor.
Init	Creates and initializes a critical section object.
Lock	Obtains ownership of the critical section object.
Term	Releases system resources used by the critical section object.

Data Members

m_bInitialized	Flags whether the internal **CRITICAL_SECTION** object has been initialized.

See Also

Reference

CComSafeDeleteCriticalSection Class

CComSafeDeleteCriticalSection Methods

For information about the methods in **CComSafeDeleteCriticalSection**, see CComSafeDeleteCriticalSection Members.

CComSafeDeleteCriticalSection::CComSafeDeleteCriticalSection

The constructor.

```
CComSafeDeleteCriticalSection();
```

Remarks

Sets the m_bInitialized data member to **false**.

See Also

Reference

CComSafeDeleteCriticalSection Class

Other Resources

CComSafeDeleteCriticalSection Members

CComSafeDeleteCriticalSection::~CComSafeDeleteCriticalSection

The destructor.

```
~CComSafeDeleteCriticalSection() throw();
```

Remarks

Releases the internal **CRITICAL_SECTION** object from memory if the m_bInitialized data member is set to **true**.

See Also

Reference

CComSafeDeleteCriticalSection Class

Other Resources

CComSafeDeleteCriticalSection Members

CComSafeDeleteCriticalSection::Init

Calls the base class implementation of Init and sets m_bInitialized to **true** if successful.

```
HRESULT Init() throw();
```

Return Value

Returns the result of CComCriticalSection::Init.

See Also

Reference

CComSafeDeleteCriticalSection Class

Other Resources

CComSafeDeleteCriticalSection Members

CComSafeDeleteCriticalSection::Term

Calls the base class implementation of CComCriticalSection::Term if the internal **CRITICAL_SECTION** object is valid.

```
HRESULT Term() throw();
```

Return Value

Returns the result of CComCriticalSection::Term, or **S_OK** if m_bInitialized was set to **false** upon entry.

Remarks

It is safe to call this method even if the internal **CRITICAL_SECTION** object is not valid. The destructor of this class calls this method if the m_bInitialized data member is set to **true**.

See Also

Reference

CComSafeDeleteCriticalSection Class

Other Resources

CComSafeDeleteCriticalSection Members

CComSafeDeleteCriticalSection::Lock

Calls the base class implementation of Lock.

```
HRESULT Lock();
```

Return Value

Returns the result of CComCriticalSection::Lock.

Remarks

This method assumes the m_bInitialized data member is set to **true** upon entry. An assertion is generated in Debug builds if this condidtion is not met.

For more information on the behavior of the function, refer to CComCriticalSection::Lock.

See Also

Reference

CComSafeDeleteCriticalSection Class

Other Resources

CComSafeDeleteCriticalSection Members

CComSafeDeleteCriticalSection Data Members

For information about the data members in **CComSafeDeleteCriticalSection**, see CComSafeDeleteCriticalSection Members.

CComSafeDeleteCriticalSection::m_bInitialized

Flags whether the internal **CRITICAL_SECTION** object has been initialized.

```
bool m_bInitialized;
```

Remarks

The **m_bInitialized** data member is used to track validity of the underlying **CRITICAL_SECTION** object associated with the CComSafeDeleteCriticalSection class. The underlying **CRITICAL_SECTION** object will not be attempted to be released from memory if this flag is not set to **true**.

See Also

Reference

CComSafeDeleteCriticalSection Class
CComSafeDeleteCriticalSection::Lock
CComSafeDeleteCriticalSection::Init
CComSafeDeleteCriticalSection::Term

Other Resources

CComSafeDeleteCriticalSection Members

CComSimpleThreadAllocator Class

This class manages thread selection for the class **CComAutoThreadModule**.

```
class CComSimpleThreadAllocator
```

Remarks

CComSimpleThreadAllocator manages thread selection for CComAutoThreadModule.
CComSimpleThreadAllocator::GetThread simply cycles through each thread and returns the next one in the sequence.

Requirements

Header: atlbase.h

See Also
Reference
CComApartment Class
Other Resources
CComSimpleThreadAllocator Members
ATL Class Overview

CComSimpleThreadAllocator Members

Methods

GetThread	Selects a thread.

See Also

Reference

CComSimpleThreadAllocator Class

CComSimpleThreadAllocator Methods

For information about the methods in **CComSimpleThreadAllocator**, see CComSimpleThreadAllocator Members.

CComSimpleThreadAllocator::GetThread

Selects a thread by specifying the next thread in the sequence.

```
int GetThread(
   CComApartment* /* pApt */,
   int nThreads
);
```

Parameters

pApt

Not used in ATL's default implementation.

nThreads

The maximum number of threads in the EXE module.

Return Value

An integer between zero and (*nThreads* – 1). Identifies one of the threads in the EXE module.

Remarks

You can override **GetThread** to provide a different method of selection or to make use of the *pApt* parameter.

GetThread is called by CComAutoThreadModule::CreateInstance.

See Also

Reference

CComSimpleThreadAllocator Class

CComApartment Class

Other Resources

CComSimpleThreadAllocator Members

CComSingleThreadModel Class

This class provides methods for incrementing and decrementing the value of a variable.

```
class CComSingleThreadModel
```

Remarks

CComSingleThreadModel provides methods for incrementing and decrementing the value of a variable. Unlike CComMultiThreadModel and CComMultiThreadModelNoCS, these methods are not thread-safe.

Typically, you use **CComSingleThreadModel** through one of two **typedef** names, either CComObjectThreadModel or CComGlobalsThreadModel. The class referenced by each **typedef** depends on the threading model used, as shown in the following table:

typedef	Single threading model	Apartment threading model	Free threading model
CComObjectThreadModel	S	S	M
CComGlobalsThreadModel	S	M	M

S=**CComSingleThreadModel**; M=**CComMultiThreadModel**

CComSingleThreadModel itself defines three **typedef** names. **ThreadModelNoCS** references **CComSingleThreadModel**. **AutoCriticalSection** and **CriticalSection** reference class CComFakeCriticalSection, which provides empty methods associated with obtaining and releasing ownership of a critical section.

Requirements

Header: atlbase.h

See Also

Other Resources

CComSingleThreadModel Members

ATL Class Overview

CComSingleThreadModel Members

Static Functions

Decrement	Decrements the value of the specified variable. This implementation is not thread-safe.
Increment	Increments the value of the specified variable. This implementation is not thread-safe.

Typedefs

AutoCriticalSection	References class CComFakeCriticalSection.
CriticalSection	References class **CComFakeCriticalSection**.
ThreadModelNoCS	References **CComSingleThreadModel**.

See Also

Reference

CComSingleThreadModel Class

CComSingleThreadModel Static Functions

For information about the static functions in **CComSingleThreadModel**, see CComSingleThreadModel Members.

CComSingleThreadModel::Decrement

This static function decrements the value of the variable pointed to by *p*.

```
static ULONG WINAPI Decrement(
   LPLONG p
) throw();
```

Parameters

p

[in] Pointer to the variable to be decremented.

Return Value

The result of the decrement.

See Also

Reference

CComSingleThreadModel Class

CComSingleThreadModel::Increment

Other Resources

CComSingleThreadModel Members

CComSingleThreadModel::Increment

This static function decrements the value of the variable pointed to by *p*.

```
static ULONG WINAPI Increment(
   LPLONG p
) throw();
```

Parameters

p

[in] Pointer to the variable to be incremented.

Return Value

The result of the increment.

See Also

Reference

CComSingleThreadModel Class

CComSingleThreadModel::Decrement

Other Resources

CComSingleThreadModel Members

CComSingleThreadModel Typedefs

For information about the typedefs in **CComSingleThreadModel**, see CComSingleThreadModel Members.

CComSingleThreadModel::AutoCriticalSection

When using **CComSingleThreadModel**, the **typedef** name **AutoCriticalSection** references class CComFakeCriticalSection.

```
typedef CComFakeCriticalSection AutoCriticalSection;
```

Remarks

Because **CComFakeCriticalSection** does not provide a critical section, its methods do nothing.

CComMultiThreadModel and CComMultiThreadModelNoCS contain definitions for **AutoCriticalSection**. The following table shows the relationship between the threading model class and the critical section class referenced by **AutoCriticalSection**:

Class defined in	Class referenced
CComSingleThreadModel	**CComFakeCriticalSection**
CComMultiThreadModel	**CComAutoCriticalSection**
CComMultiThreadModelNoCS	**CComFakeCriticalSection**

In addition to **AutoCriticalSection**, you can use the **typedef** name CriticalSection. You should not specify **AutoCriticalSection** in global objects or static class members if you want to eliminate the CRT startup code.

Example

See CComMultiThreadModel::AutoCriticalSection.

See Also

Reference

CComSingleThreadModel Class
CComObjectThreadModel
CComGlobalsThreadModel
CComSingleThreadModel::ThreadModelNoCS

Other Resources

CComSingleThreadModel Members

CComSingleThreadModel::CriticalSection

When using **CComSingleThreadModel**, the **typedef** name **CriticalSection** references class CComFakeCriticalSection.

```
typedef CComFakeCriticalSection CriticalSection;
```

Remarks

Because **CComFakeCriticalSection** does not provide a critical section, its methods do nothing.

CComMultiThreadModel and CComMultiThreadModelNoCS contain definitions for **CriticalSection**. The following table shows the relationship between the threading model class and the critical section class referenced by **CriticalSection**:

Class defined in	Class referenced
CComSingleThreadModel	**CComFakeCriticalSection**
CComMultiThreadModel	**CComCriticalSection**
CComMultiThreadModelNoCS	**CComFakeCriticalSection**

In addition to **CriticalSection**, you can use the **typedef** name AutoCriticalSection. You should not specify **AutoCriticalSection** in global objects or static class members if you want to eliminate the CRT startup code.

Example

See CComMultiThreadModel::AutoCriticalSection.

See Also

Reference

CComSingleThreadModel Class
CComObjectThreadModel
CComGlobalsThreadModel
CComSingleThreadModel::ThreadModelNoCS

Other Resources

CComSingleThreadModel Members

CComSingleThreadModel::ThreadModelNoCS

When using **CComSingleThreadModel**, the **typedef** name **ThreadModelNoCS** simply references **CComSingleThreadModel**.

```
typedef CComSingleThreadModel ThreadModelNoCS;
```

Remarks

CComMultiThreadModel and CComMultiThreadModelNoCS contain definitions for **ThreadModelNoCS**. The following table shows the relationship between the threading model class and the class referenced by **ThreadModelNoCS**:

Class defined in	Class referenced
CComSingleThreadModel	**CComSingleThreadModel**
CComMultiThreadModel	**CComMultiThreadModelNoCS**
CComMultiThreadModelNoCS	**CComMultiThreadModelNoCS**

Example

See CComMultiThreadModel::AutoCriticalSection.

See Also

Reference

CComSingleThreadModel Class
CComObjectThreadModel
CComGlobalsThreadModel

Other Resources

CComSingleThreadModel Members

CComTearOffObject Class

This class implements a tear-off interface.

```
template<
   class Base
>
class CComTearOffObject :
      public Base
```

Parameters

Base

Your tear-off class, derived from **CComTearOffObjectBase** and the interfaces you want your tear-off object to support.

ATL implements its tear-off interfaces in two phases — the **CComTearOffObjectBase** methods handle the reference count and **QueryInterface**, while **CComTearOffObject** implements IUnknown.

Remarks

CComTearOffObject implements a tear-off interface as a separate object that is instantiated only when that interface is queried for. The tear-off is deleted when its reference count becomes zero. Typically, you build a tear-off interface for an interface that is rarely used, since using a tear-off saves a vtable pointer in all the instances of your main object.

You should derive the class implementing the tear-off from **CComTearOffObjectBase** and from whichever interfaces you want your tear-off object to support. **CComTearOffObjectBase** is templatized on the owner class and the thread model. The owner class is the class of the object for which a tear-off is being implemented. If you do not specify a thread model, the default thread model is used.

You should create a COM map for your tear-off class. When ATL instantiates the tear-off, it will create **CComTearOffObject<CYourTearOffClass>** or **CComCachedTearOffObject<CYourTearOffClass>**.

For example, in the BEEPER sample, the `CBeeper2` class is the tear-off class and the `CBeeper` class is the owner class:

```
class CBeeper2 : public ISupportErrorInfo,
                 public CComTearOffObjectBase<CBeeper>
{
public:
   CBeeper2() {}
   STDMETHOD(InterfaceSupportsErrorInfo)(REFIID riid)
   {
      return (InlineIsEqualGUID(IID_IBeeper,riid)) ?
              S_OK : S_FALSE;
   }

   BEGIN_COM_MAP(CBeeper2)
      COM_INTERFACE_ENTRY(ISupportErrorInfo)
   END_COM_MAP()
};

class CBeeper :
   public IDispatchImpl<IBeeper, &IID_IBeeper,
                        &LIBID_BeeperLib>,
   public CComObjectRoot,
   public CComCoClass<CBeeper, &CLSID_Beeper>
{
public:
   CBeeper();
   BEGIN_COM_MAP(CBeeper)
      COM_INTERFACE_ENTRY(IDispatch)
      COM_INTERFACE_ENTRY(IBeeper)
      COM_INTERFACE_ENTRY_TEAR_OFF(IID_ISupportErrorInfo,
                              CBeeper2)
   END_COM_MAP()
```

```
...
};
```

Requirements

Header: atlcom.h

See Also

Reference

CComCachedTearOffObject Class

Other Resources

CComTearOffObject Members

ATL Class Overview

CComTearOffObject Members

Methods

AddRef	Increments the reference count for a **CComTearOffObject** object.
CComTearOffObject	The constructor.
~CComTearOffObject	The destructor.
QueryInterface	Returns a pointer to the requested interface on either your tear-off class or the owner class.
Release	Decrements the reference count for a **CComTearOffObject** object and destroys it.

CComTearOffObjectBase Methods

CComTearOffObjectBase	Constructor.

CComTearOffObjectBase Data Members

m_pOwner	A pointer to a **CComObject** derived from the owner class.

See Also

Reference

CComTearOffObject Class

CComTearOffObject Methods

For information about the methods in **CComTearOffObject**, see CComTearOffObject Members.

CComTearOffObject::AddRef

Increments the reference count of the **CComTearOffObject** object by one.

```
STDMETHOD_(ULONG, AddRef)( );
```

Return Value

A value that may be useful for diagnostics and testing.

See Also

Reference

CComTearOffObject Class

CComTearOffObject::Release

Other Resources

CComTearOffObject Members

CComTearOffObject::CComTearOffObject

The constructor.

```
CComTearOffObject(
   void* pv
);
```

Parameters

pv

[in] Pointer that will be converted to a pointer to a **CComObject<Owner>** object.

Remarks

Increments the owner's reference count by one.

See Also

Reference

CComTearOffObject Class

CComCachedTearOffObject::CComCachedTearOffObject

Other Resources

CComTearOffObject Members

CComTearOffObject::~CComTearOffObject

The destructor.

```
~CComTearOffObject( );
```

Remarks

Frees all allocated resources, calls FinalRelease, and decrements the module lock count.

See Also

Reference

CComTearOffObject Class

Other Resources

CComTearOffObject Members

CComTearOffObject::CComTearOffObjectBase

The constructor.

```
CComTearOffObjectBase( );
```

Remarks

Initializes the m_pOwner member to **NULL**.

See Also

Reference

CComTearOffObject Class

CComCachedTearOffObject::CComCachedTearOffObject

Other Resources

CComTearOffObject Members

CComTearOffObject::QueryInterface

Retrieves a pointer to the requested interface.

```
STDMETHOD(QueryInterface)(
   REFIID iid ,
      void** ppvObject
);
```

Parameters

iid

[in] The IID of the interface being requested.

ppvObject

[out] A pointer to the interface pointer identified by *iid*, or **NULL** if the interface is not found.

Return Value

A standard **HRESULT** value.

Remarks

Queries first for interfaces on your tear-off class. If the interface is not there, queries for the interface on the owner object. If the requested interface is **IUnknown**, returns the **IUnknown** of the owner.

See Also

Reference

CComTearOffObject Class
CComTearOffObject::AddRef
CComTearOffObject::Release

Other Resources

CComTearOffObject Members

CComTearOffObject::Release

Decrements the reference count by one and, if the reference count is zero, deletes the **CComTearOffObject**.

```
STDMETHOD_ULONG Release( );
```

Return Value

In non-debug builds, always returns zero. In debug builds, returns a value that may be useful for diagnostics or testing.

See Also

Reference

CComTearOffObject Class

CComTearOffObject::AddRef

Other Resources

CComTearOffObject Members

CComTearOffObject Data Members

For information about the data members in **CComTearOffObject**, see CComTearOffObject Members.

CComTearOffObject::m_pOwner

A pointer to a CComObject object derived from *Owner*.

```
CComObject<Owner>* m_pOwner;
```

Parameters

Owner

[in] The class for which a tear-off is being implemented.

Remarks

The pointer is initialized to **NULL** during construction.

See Also

Reference

CComTearOffObject Class

CComTearOffObject::CComTearOffObjectBase

Other Resources

CComTearOffObject Members

CComUnkArray Class

This class stores **IUnknown** pointers, and is designed to be used as a parameter to the IConnectionPointImpl template class.

```
template<
   unsigned int nMaxSize
>
class CComUnkArray
```

Parameters

nMaxSize

The maximum number of **IUnknown** pointers that can be held in the static array.

Remarks

CComUnkArray holds a fixed number of **IUnknown** pointers, each an interface on a connection point. **CComUnkArray** can be used as a parameter to the IConnectionPointImpl template class. **CComUnkArray<1>** is a template specialization of **CComUnkArray** that has been optimized for one connection point.

The **CComUnkArray** methods begin and end can be used to loop through all connection points (for example, when an event is fired).

See Adding Connection Points to an Object for details on automating creation of connection point proxies.

Note The class CComDynamicUnkArray is used by the **Add Class** wizard when creating a control which has Connection Points. If you wish to specify the number of Connection Points manually, change the reference from **CComDynamicUnkArray** to `CComUnkArray< n >`, where *n* is the number of connection points required.

Requirements

Header: atlcom.h

See Also

Reference

CComDynamicUnkArray Class

Other Resources

CComUnkArray Members

ATL Class Overview

CComUnkArray Members

Methods

Add	Call this method to add an **IUnknown** pointer to the array.
begin	Returns a pointer to the first **IUnknown** pointer in the collection.
CComUnkArray	Constructor.
end	Returns a pointer to one past the last **IUnknown** pointer in the collection.
GetCookie	Call this method to get the cookie associated with a given **IUnknown** pointer.
GetUnknown	Call this method to get the **IUnknown** pointer associated with a given cookie.
Remove	Call this method to remove an **IUnknown** pointer from the array.

See Also

Reference

CComUnkArray Class

CComUnkArray Methods

For information about the methods in **CComUnkArray**, see CComUnkArray Members.

CComUnkArray::Add

Call this method to add an **IUnknown** pointer to the array.

```
DWORD Add(
   IUnknown* pUnk
);
```

Parameters

pUnk

Call this method to add an **IUnknown** pointer to the array.

Return Value

Returns the cookie associated with the newly added pointer, or 0 if the array is not large enough to contain the new pointer.

See Also

Reference

CComUnkArray Class

CComUnkArray::Remove

Other Resources

CComUnkArray Members

CComUnkArray::begin

Returns a pointer to the beginning of the collection of **IUnknown** interface pointers.

```
IUnknown** begin( );
```

Return Value

A pointer to an **IUnknown** interface pointer.

Remarks

The collection contains pointers to interfaces stored locally as **IUnknown**. You cast each **IUnknown** interface to the real interface type and then call through it. You do not need to query for the interface first.

Before using the **IUnknown** interface, you should check that it is not **NULL**.

See Also

Reference

CComUnkArray Class
CComUnkArray::end
CComDynamicUnkArray::begin

Other Resources

CComUnkArray Members

CComUnkArray::CComUnkArray

The constructor.

```
CComUnkArray( );
```

Remarks

Sets the collection to hold *nMaxSize* **IUnknown** pointers, and initializes the pointers to **NULL**.

See Also

Reference

CComUnkArray Class

Other Resources

CComUnkArray Members

CComUnkArray::end

Returns a pointer to one past the last **IUnknown** pointer in the collection.

```
IUnknown** end( );
```

Return Value

A pointer to an **IUnknown** interface pointer.

Remarks

The **CComUnkArray** methods **begin** and **end** can be used to loop through all connection points, for example, when an event is fired.

```
IUnknown** p = m_vec.begin();
while(p != m_vec.end())
{
   // Do something with *p
   p++;
}
```

See Also

Reference

CComUnkArray Class
CComUnkArray::begin
CComDynamicUnkArray::end

Other Resources

CComUnkArray Members

CComUnkArray::GetCookie

Call this method to get the cookie associated with a given **IUnknown** pointer.

```
DWORD WINAPI GetCookie(
   IUnknown** ppFind
);
```

Parameters

ppFind

The **IUnknown** pointer for which the associated cookie is required.

Return Value

Returns the cookie associated with the **IUnknown** pointer, or 0 if no matching **IUnknown** pointer is found.

Remarks

If there is more than one instance of the same **IUnknown** pointer, this function returns the cookie for the first one.

See Also

Reference

CComUnkArray Class

CComUnkArray::GetUnknown

Other Resources

CComUnkArray Members

CComUnkArray::GetUnknown

Call this method to get the **IUnknown** pointer associated with a given cookie.

```
IUnknown* WINAPI GetUnknown(
   DWORD dwCookie
);
```

Parameters

dwCookie

The cookie for which the associated **IUnknown** pointer is required.

Return Value

Returns the **IUnknown** pointer, or NULL if no matching cookie is found.

See Also

Reference

CComUnkArray Class

CComUnkArray::GetCookie

Other Resources

CComUnkArray Members

CComUnkArray::Remove

Call this method to remove an **IUnknown** pointer from the array.

```
BOOL Remove(
   DWORD dwCookie
);
```

Parameters

dwCookie

The cookie referencing the **IUnknown** pointer to be removed from the array.

Return Value

Returns **TRUE** if the pointer is removed, **FALSE** otherwise.

See Also

Reference

CComUnkArray Class

CComUnkArray::Add

Other Resources

CComUnkArray Members

CComVariant Class

This class wraps the VARIANT type, providing a member indicating the type of data stored.

```
class CComVariant : public tagVARIANT
```

Remarks

CComVariant wraps the VARIANT and VARIANTARG type, which consists of a union and a member indicating the type of the data stored in the union. **VARIANT**s are typically used in Automation.

CComVariant derives from the **VARIANT** type so it can be used wherever a **VARIANT** can be used. You can, for example, use the **V_VT** macro to extract the type of a **CComVariant** or you can access the **vt** member directly just as you can with a **VARIANT**.

Requirements

Header: atlcomcli.h

See Also

Other Resources

CComVariant Members

ATL Class Overview

CComVariant Members

Methods

Attach	Attaches a **VARIANT** to the **CComVariant** object.
CComVariant	The constructor.
~CComVariant	The destructor.
ChangeType	Converts the **CComVariant** object to a new type.
Clear	Clears the **CComVariant** object.
Copy	Copies a **VARIANT** to the **CComVariant** object.
CopyTo	Copies the contents of the **CComVariant** object.
Detach	Detaches the underlying **VARIANT** from the **CComVariant** object.
GetSize	Returns the size in number of bytes of the contents of the **CComVariant** object.
ReadFromStream	Loads a **VARIANT** from a stream.
SetByRef	Initializes the **CComVariant** object and sets the **vt** member to **VT_BYREF**.
WriteToStream	Saves the underlying **VARIANT** to a stream.

Operators

CComVariant::operator <	Indicates whether the **CComVariant** object is less than the specified **VARIANT**.
CComVariant::operator >	Indicates whether the **CComVariant** object is greater than the specified **VARIANT**.
operator !=	Indicates whether the **CComVariant** object does not equal the specified **VARIANT**.
operator =	Assigns a value to the **CComVariant** object.
operator ==	Indicates whether the **CComVariant** object equals the specified **VARIANT**.

See Also

Reference

CComVariant Class

CComVariant Methods

For information about the methods in **CComVariant**, see CComVariant Members.

CComVariant::Attach

Safely clears the current contents of the **CComVariant** object, copies the contents of *pSrc* into this object, then sets the variant type of *pSrc* to **VT_EMPTY**.

```
HRESULT Attach(
   VARIANT* pSrc
);
```

Parameters

pSrc

[in] Points to the VARIANT to be attached to the object.

Return Value

A standard **HRESULT** value.

Remarks

Ownership of the data held by *pSrc* is transferred to the **CComVariant** object.

See Also

Reference

CComVariant Class

CComVariant::Detach

Other Resources

CComVariant Members

CComVariant::CComVariant

Each constructor handles the safe initialization of the **CComVariant** object by calling the VariantInit Win32 function or by setting the object's value and type according to the parameters passed.

```
CComVariant( ) throw();
CComVariant(
   const CComVariant& varSrc
);
CComVariant(
   const VARIANT& varSrc
);
CComVariant(
   LPCOLESTR lpszSrc
);
CComVariant(
   LPCSTR lpszSrc
);
CComVariant(
   bool bSrc
);
CComVariant(
   BYTE nSrc
) throw();
CComVariant(
   int nSrc,
   VARTYPE vtSrc = VT_I4
) throw();
CComVariant(
   unsigned int nSrc,
   VARTYPE vtSrc = VT_UI4
) throw();
CComVariant(
   short nSrc
) throw();
CComVariant(
   unsigned short nSrc
) throw();
CComVariant(
   long nSrc,
   VARTYPE vtSrc = VT_I4
) throw();
CComVariant(
   unsigned long nSrc
) throw();
CComVariant(
   LONGLONG nSrc
) throw();
CComVariant(
   ULONGLONG nSrc
) throw();
CComVariant(
   float fltSrc
) throw();
CComVariant(
   double dblSrc,
   VARTYPE vtSrc = VT_R8
) throw();
CComVariant(
   CY cySrc
) throw();
CComVariant(
   IDispatch* pSrc
) throw();
```

```
CComVariant(
   IUnknown* pSrc
) throw();
CComVariant(
   const SAFEARRAY *pSrc
);
CComVariant(
   char cSrc
) throw();
CComVariant(
   const CComBSTR& bstrSrc
);
```

Parameters

varSrc

[in] The **CComVariant** or VARIANT used to initialize the **CComVariant** object. The contents of the source variant are copied to the destination without conversion.

lpszSrc

[in] The character string used to initialize the **CComVariant** object. You can pass a zero-terminated wide (Unicode) character string to the **LPCOLESTR** version of the constructor or an ANSI string to the **LPCSTR** version. In either case the string is converted to a Unicode **BSTR** allocated using **SysAllocString**. The type of the **CComVariant** object will be **VT_BSTR**.

bSrc

[in] The **bool** used to initialize the **CComVariant** object. The **bool** argument is converted to a **VARIANT_BOOL** before being stored. The type of the **CComVariant** object will be **VT_BOOL**.

nSrc

[in] The **int**, **BYTE**, **short**, **long**, **LONGLONG**, **ULONGLONG**, **unsigned short**, **unsigned long**, or **unsigned int** used to initialize the **CComVariant** object. The type of the **CComVariant** object will be **VT_I4**, **VT_UI1**, **VT_I2**, **VT_I4**, **VT_I8**, **VT_UI8**, **VT_UI2**, **VT_UI4**, or **VT_UI4**, respectively.

vtSrc

[in] The type of the variant. When the first parameter is **int**, valid types are **VT_I4** and **VT_INT**. When the first parameter is **long**, valid types are **VT_I4** and **VT_ERROR**. When the first parameter is **double**, valid types are **VT_R8** and **VT_DATE**. When the first parameter is **unsigned int**, valid types are **VT_UI4** and **VT_UINT**.

fltSrc

[in] The **float** used to initialize the **CComVariant** object. The type of the **CComVariant** object will be **VT_R4**.

dblSrc

[in] The **double** used to initialize the **CComVariant** object. The type of the **CComVariant** object will be **VT_R8**.

cySrc

[in] The **CY** used to initialize the **CComVariant** object. The type of the **CComVariant** object will be **VT_CY**.

pSrc

[in] The **IDispatch** or **IUnknown** pointer used to initialize the **CComVariant** object. **AddRef** will be called on the interface pointer. The type of the **CComVariant** object will be **VT_DISPATCH** or **VT_UNKNOWN**, respectively.

Or, the **SAFERRAY** pointer used to initialize the **CComVariant** object. A copy of the **SAFEARRAY** is stored in the **CComVariant** object. The type of the **CComVariant** object will be a combination of the original type of the **SAFEARRAY** and **VT_ARRAY**.

cSrc

[in] The **char** used to initialize the **CComVariant** object. The type of the **CComVariant** object will be **VT_I1**.

bstrSrc

[in] The BSTR used to initialize the **CComVariant** object. The type of the **CComVariant** object will be **VT_BSTR**.

Remarks

The destructor manages cleanup by calling CComVariant::Clear.

See Also

Reference

CComVariant Class

Other Resources

CComVariant Members

CComVariant::~CComVariant

The destructor.

```
~CComVariant( ) throw();
```

Remarks

This method manages cleanup by calling CComVariant::Clear.

See Also

Reference

CComVariant Class

Other Resources

CComVariant Members

CComVariant::ChangeType

Converts the **CComVariant** object to a new type.

```
HRESULT ChangeType(
   VARTYPE vtNew,
   const VARIANT* pSrc = NULL
);
```

Parameters

vtNew

[in] The new type for the **CComVariant** object.

pSrc

[in] A pointer to the VARIANT whose value will be converted to the new type. The default value is **NULL**, meaning the **CComVariant** object will be converted in place.

Return Value

A standard **HRESULT** value.

Remarks

If you pass a value for *pSrc*, **ChangeType** will use this **VARIANT** as the source for the conversion. Otherwise, the **CComVariant** object will be the source.

See Also

Reference

CComVariant Class

Other Resources

CComVariant Members

CComVariant::Clear

Clears the **CComVariant** object by calling the VariantClear Win32 function.

```
HRESULT Clear( );
```

Return Value

A standard **HRESULT** value.

Remarks

The destructor automatically calls **Clear**.

See Also

Reference

CComVariant Class

Other Resources

CComVariant Members

CComVariant::Copy

Frees the **CComVariant** object and then assigns it a copy of the specified **VARIANT**.

```
HRESULT Copy(
   const VARIANT* pSrc
);
```

Parameters

pSrc

[in] A pointer to the VARIANT to be copied.

Return Value

A standard **HRESULT** value.

See Also

Reference

CComVariant Class
CComVariant::operator =
VariantCopy
CComVariant::CopyTo

Other Resources

CComVariant Members

CComVariant::CopyTo

Copies the contents of the **CComVariant** object.

```
HRESULT CopyTo(
   BSTR *pstrDest
);
```

Parameters

pstrDest

Points to a **BSTR** that will receive a copy of the contents of the **CComVariant** object.

Return Value

A standard **HRESULT** value.

Remarks

The **CComVariant** object must be of type **VT_BSTR**.

See Also

Reference

CComVariant Class

CComVariant::Copy

Other Resources

CComVariant Members

CComVariant::Detach

Detaches the underlying **VARIANT** from the **CComVariant** object and sets the object's type to **VT_EMPTY**.

```
HRESULT Detach(
   VARIANT* pDest
);
```

Parameters

pDest

[out] Returns the underlying VARIANT value of the object.

Return Value

A standard **HRESULT** value.

Remarks

Note that the contents of the **VARIANT** referenced by *pDest* will automatically be cleared before being assigned the value and type of the calling **CComVariant** object.

See Also

Reference

CComVariant Class

CComVariant::Attach

Other Resources

CComVariant Members

CComVariant::GetSize

For simple-fixed size **VARIANT**s, this method returns the **sizeof** the underlying data type plus **sizeof(VARTYPE)**.

```
ULONG GetSize() const;
```

Return Value

The size in bytes of the current contents of the **CComVariant** object.

Remarks

If the **VARIANT** contains an interface pointer, **GetSize** queries for **IPersistStream** or **IPersistStreamInit**. If successful, the return value is the low-order 32 bits of the value returned by **GetSizeMax** plus **sizeof(VARTYPE)**. If the interface pointer is **NULL**, **GetSize** returns the **sizeof** a **CLSID** plus **sizeof(VARTYPE).**

In all other cases, a temporary **VARIANT** of type **VT_BSTR** is coerced from the current **VARIANT**. The length of this **BSTR** is calculated as the size of the length of the string plus the length of the string itself plus the size of the null character plus **sizeof(VARTYPE)**. If the **VARIANT** cannot be coerced to a **VARIANT** of type **VT_BSTR**, **GetSize** returns **sizeof(VARTYPE).**

The size returned by this method matches the number of bytes used by CComVariant::WriteToStream under successful conditions.

See Also

Reference

CComVariant Class

Other Resources

CComVariant Members

CComVariant::ReadFromStream

Sets the underlying **VARIANT** to the **VARIANT** contained in the specified stream.

```
HRESULT ReadFromStream(
   IStream* pStream
);
```

Parameters

pStream

[in] A pointer to the IStream interface on the stream containing the data.

Return Value

A standard **HRESULT** value.

Remarks

ReadToStream requires a previous call to WriteToStream.

See Also

Reference

CComVariant Class

Other Resources

CComVariant Members

CComVariant::SetByRef

Initializes the **CComVariant** object and sets the **vt** member to **VT_BYREF**.

```
template< typename T >
void SetByRef(
   T* pT
) throw();
```

Parameters

T

The type of **VARIANT**, for example, **BSTR**, **int**, or **char**.

pT

The pointer used to initialize the **CComVariant** object.

Remarks

SetByRef is a function template that initializes the **CComVariant** object to the pointer *pT* and sets the **vt** member to **VT_BYREF**. For example:

```
CComVariant var;
int nData = 10;
var.SetByRef(&nData);
```

See Also

Reference

CComVariant Class

Other Resources

CComVariant Members

CComVariant::WriteToStream

Saves the underlying **VARIANT** to a stream.

```
HRESULT WriteToStream(
   IStream* pStream
);
```

Parameters

pStream

[in] A pointer to the IStream interface on a stream.

Return Value

A standard **HRESULT** value.

See Also

Reference

CComVariant Class

CComVariant::ReadFromStream

Other Resources

CComVariant Members

CComVariant Operators

For information about the operators in **CComVariant**, see CComVariant Members.

CComVariant::operator =

Assigns a value and corresponding type to the **CComVariant** object.

```
CComVariant& operator =(
   const CComVariant& varSrc
);
CComVariant& operator =(
   const VARIANT& varSrc
);
CComVariant& operator =(
   const CComBSTR& bstrSrc
);
CComVariant& operator =(
   LPCOLESTR lpszSrc
);
CComVariant& operator =(
   LPCSTR lpszSrc
);
CComVariant& operator =(
   bool bSrc
);
CComVariant& operator =(
   BYTE nSrc
) throw();
CComVariant& operator =(
   int nSrc
) throw();
CComVariant& operator =(
   unsigned int nSrc
) throw();
CComVariant& operator =(
   short nSrc
) throw();
CComVariant& operator =(
   unsigned short nSrc
) throw();
CComVariant& operator =(
   long nSrc
) throw();
CComVariant& operator =(
   unsigned long nSrc
) throw();
CComVariant& operator =(
   LONGLONG nSrc
) throw();
CComVariant& operator =(
   ULONGLONG nSrc
) throw();
CComVariant& operator =(
   float fltSrc
) throw();
CComVariant& operator =(
   double dblSrc
) throw();
CComVariant& operator =(
   CY cySrc
) throw();
CComVariant& operator =(
   IDispatch* pSrc
) throw();
CComVariant& operator =(
   IUnknown* pSrc
) throw();
```

```
CComVariant& operator =(
   const SAFEARRAY *pSrc
);
CComVariant& operator =(
   char cSrc
) throw();
```

Parameters

varSrc

[in] The **CComVariant** or VARIANT to be assigned to the **CComVariant** object. The contents of the source variant are copied to the destination without conversion.

bstrSrc

[in] The BSTR to be assigned to the **CComVariant** object. The type of the **CComVariant** object will be **VT_BSTR**.

lpszSrc

[in] The character string to be assigned to the **CComVariant** object. You can pass a zero-terminated wide (Unicode) character string to the **LPCOLESTR** version of the operator or an ANSI string to the **LPCSTR** version. In either case, the string is converted to a Unicode **BSTR** allocated using **SysAllocString**. The type of the **CComVariant** object will be **VT_BSTR**.

bSrc

[in] The **bool** to be assigned to the **CComVariant** object. The **bool** argument is converted to a **VARIANT_BOOL** before being stored. The type of the **CComVariant** object will be **VT_BOOL**.

nSrc

[in] The **int**, **BYTE**, **short**, **long**, **LONGLONG**, **ULONGLONG**, **unsigned short**, **unsigned long**, or **unsigned int** to be assigned to the **CComVariant** object. The type of the **CComVariant** object will be **VT_I4**, **VT_UI1**, **VT_I2**, **VT_I4**, **VT_I8**, **VT_UI8**, **VT_UI2**, **VT_UI4**, or **VT_UI4**, respectively.

fltSrc

[in] The **float** to be assigned to the **CComVariant** object. The type of the **CComVariant** object will be **VT_R4**.

dblSrc

[in] The **double** to be assigned to the **CComVariant** object. The type of the **CComVariant** object will be **VT_R8**.

cySrc

[in] The **CY** to be assigned to the **CComVariant** object. The type of the **CComVariant** object will be **VT_CY**.

pSrc

[in] The **IDispatch** or **IUnknown** pointer to be assigned to the **CComVariant** object. **AddRef** will be called on the interface pointer. The type of the **CComVariant** object will be **VT_DISPATCH** or **VT_UNKNOWN**, respectively.

Or, a **SAFEARRAY** pointer to be assigned to the **CComVariant** object. A copy of the **SAFEARRAY** is stored in the **CComVariant** object. The type of the **CComVariant** object will be a combination of the original type of the **SAFEARRAY** and **VT_ARRAY**.

cSrc

[in] The char to be assigned to the **CComVariant** object. The type of the **CComVariant** object will be **VT_I1**.

See Also

Reference

CComVariant Class

CComVariant::Copy

Concepts

VARIANT

Other Resources

CComVariant Members

CComVariant::operator ==

Indicates whether the **CComVariant** object equals the specified **VARIANT**.

```
bool operator ==(
   const VARIANT& varSrc
) const throw();
```

Remarks

Returns **true** if the value and type of *varSrc* are equal to the value and type, respectively, of the **CComVariant** object. Otherwise, **false**. The operator uses the user's default locale to perform the comparison.

See Also

Reference

CComVariant Class

CComVariant::operator !=

Concepts

VARIANT and VARIANTARG

Other Resources

CComVariant Members

CComVariant::operator !=

Indicates whether the **CComVariant** object does not equal the specified **VARIANT**.

```
bool operator !=(
   const VARIANT& varSrc
) const throw();
```

Remarks

Returns **true** if either the value or type of *varSrc* is not equal to the value or type, respectively, of the **CComVariant** object. Otherwise, **false**. The operator uses the user's default locale to perform the comparison.

See Also

Reference

CComVariant Class

CComVariant::operator ==

Concepts

VARIANT and VARIANTARG

Other Resources

CComVariant Members

CComVariant::operator <

Indicates whether the **CComVariant** object is less than the specified **VARIANT**.

```
bool operator <(
   const VARIANT& varSrc
) const throw();
```

Remarks

Returns **true** if the value of the **CComVariant** object is less than the value of *varSrc*. Otherwise, **false**. The operator uses the user's default locale to perform the comparison.

See Also

Reference

CComVariant Class

CComVariant::operator >

Concepts

VARIANT and VARIANTARG

Other Resources

CComVariant Members

CComVariant::operator >

Indicates whether the **CComVariant** object is greater than the specified **VARIANT**.

```
bool operator >(
   const VARIANT& varSrc
) const throw();
```

Remarks

Returns **true** if the value of the **CComVariant** object is greater than the value of *varSrc*. Otherwise, **false**. The operator uses the user's default locale to perform the comparison.

See Also

Reference

CComVariant Class

CComVariant::operator <

Concepts

VARIANT and VARIANTARG

Other Resources

CComVariant Members

CContainedWindowT Class

This class implements a window contained within another object.

```
template <
   class TBase = CWindow,
   class TWinTraits = CControlWinTraits
>
class CContainedWindowT :
   public TBase
```

Parameters

TBase

The base class of your new class. The default base class is **CWindow**.

TWinTraits

A traits class that defines styles for your window. The default is **CControlWinTraits**.

Note

CContainedWindow is a specialization of **CContainedWindowT**. If you want to change the base class or traits, use **CContainedWindowT** directly.

Remarks

CContainedWindowT implements a window contained within another object. **CContainedWindowT**'s window procedure uses a message map in the containing object to direct messages to the appropriate handlers. When constructing a **CContainedWindowT** object, you specify which message map should be used.

CContainedWindowT allows you to create a new window by superclassing an existing window class. The **Create** method first registers a window class that is based on an existing class but uses **CContainedWindowT::WindowProc**. **Create** then creates a window based on this new window class. Each instance of **CContainedWindowT** can superclass a different window class.

CContainedWindowT also supports window subclassing. The **SubclassWindow** method attaches an existing window to the **CContainedWindowT** object and changes the window procedure to **CContainedWindowT::WindowProc**. Each instance of **CContainedWindowT** can subclass a different window.

Note

For any given **CContainedWindowT** object, call either **Create** or **SubclassWindow**. You should not invoke both methods on the same object.

When you use the **Add control based on** option in the ATL Project Wizard, the wizard will automatically add a **CContainedWindowT** data member to the class implementing the control. The following example is taken from the SUBEDIT sample and shows how the contained window is declared:

```
class CAtlEdit : ...
{
public:
   // Declare a contained window data member
   CContainedWindow m_EditCtrl;

   // Initialize the contained window:
   // 1. Pass "EDIT" to specify that the contained
   //    window should be based on the standard
   //    Windows Edit box
   // 2. Pass 'this' pointer to specify that CAtlEdit
   //    contains the message map to be used for the
   //    contained window's message processing
   // 3. Pass the identifier of the message map. '1'
   //    identifies the alternate message map declared
```

```
    //    with ALT_MSG_MAP(1)
    CAtlEdit() : m_EditCtrl(_T("EDIT"), this, 1)
    {
       m_bWindowOnly = TRUE;
    }

    // Declare the default message map,
    // identified by '0'
    BEGIN_MSG_MAP(CAtlEdit)
       ...
       MESSAGE_HANDLER(WM_CREATE, OnCreate)
       ...
    // Declare an alternate message map,
    // identified by '1'
    ALT_MSG_MAP(1)
       MESSAGE_HANDLER(WM_CHAR, OnChar)
    END_MSG_MAP()

    // Define OnCreate handler
    // When the containing window receives a WM_CREATE
    // message, create the contained window by calling
    // CContainedWindow::Create
    LRESULT OnCreate(UINT uMsg, WPARAM wParam,
                     LPARAM lParam, BOOL& bHandled)
    {
       ...
       m_EditCtrl.Create(m_hWnd, rc, _T("hello"),
                         WS_CHILD | WS_VISIBLE |
                         ES_MULTILINE | ES_AUTOVSCROLL);
       return 0;
    }

    ...
};
```

For more information about	See
Creating controls	ATL Tutorial
Using windows in ATL	ATL Window Classes
ATL Project Wizard	Creating an ATL Project
Windows	Windows and subsequent topics in the *Platform SDK*

Requirements

Header: atlwin.h

See Also

Reference

CWindow Class
CWindowImpl Class
CMessageMap Class
BEGIN_MSG_MAP
ALT_MSG_MAP

Other Resources

CContainedWindowT Members
ATL Class Overview

CContainedWindowT Members

Methods

CContainedWindowT	Constructor. Initializes data members to specify which message map will process the contained window's messages.
Create	Creates a window.
DefWindowProc	Provides default message processing.
GetCurrentMessage	Returns the current message.
RegisterWndSuperclass	Registers the window class of the contained window.
SubclassWindow	Subclasses a window.
SwitchMessageMap	Changes which message map is used to process the contained window's messages.
UnsubclassWindow	Restores a previously subclassed window.

Data Members

m_dwMsgMapID	Identifies which message map will process the contained window's messages.
m_lpszClassName	Specifies the name of an existing window class on which a new window class will be based.
m_pfnSuperWindowProc	Points to the window class's original window procedure.
m_pObject	Points to the containing object.

Static Functions

WindowProc	Processes messages sent to the contained window.

See Also

Reference

CContainedWindowT Class

CContainedWindowT Methods

For information about the methods in **CContainedWindowT**, see CContainedWindowT Members.

CContainedWindowT::CContainedWindowT

The constructor initializes data members.

```
CContainedWindowT(
   LPTSTR lpszClassName,
   CMessageMap* pObject,
   DWORD dwMsgMapID = 0
);
CContainedWindowT( CMessageMap* pObject, DWORD dwMsgMapID = 0 )
CContainedWindowT( );
```

Parameters

lpszClassName

[in] The name of an existing window class on which the contained window will be based.

pObject

[in] A pointer to the containing object that declares the message map. This object's class must derive from CMessageMap.

dwMsgMapID

[in] Identifies the message map that will process the contained window's messages. The default value, 0, specifies the default message map declared with BEGIN_MSG_MAP. To use an alternate message map declared with ALT_MSG_MAP(msgMapID), pass *msgMapID*.

Remarks

If you want to create a new window through Create, you must pass the name of an existing window class for the *lpszClassName* parameter. For an example, see the CContainedWindow overview.

There are three constructors:

- The constructor with three arguments is the one typically called.
- The constructor with two arguments uses the class name from **TBase::GetWndClassName**.
- The constructor with no arguments is used if you want to supply the arguments later. You must supply the window class name, message map object, and message map ID when you later call **Create**.

If you subclass an existing window through SubclassWindow, the *lpszClassName* value will not be used; therefore, you can pass **NULL** for this parameter.

See Also

Reference

CContainedWindowT Class
CContainedWindowT::m_lpszClassName
CContainedWindowT::m_pObject
CContainedWindowT::m_pfnSuperWindowProc
CContainedWindowT::SwitchMessageMap

Other Resources

CContainedWindowT Members

CContainedWindowT::Create

Calls RegisterWndSuperclass to register a window class that is based on an existing class but uses CContainedWindowT::WindowProc.

```
HWND Create(
   HWND hWndParent,
   _U_RECT rect,
   LPCTSTR szWindowName = NULL,
   DWORD dwStyle = 0,
   DWORD dwExStyle = 0,
   _U_MENUorID MenuOrID = 0U,
   LPVOID lpCreateParam = NULL
);
HWND Create(
   CMessageMap* pObject,
   DWORD dwMsgMapID,
   HWND hWndParent,
   _U_RECT rect,
   LPCTSTR szWindowName = NULL,
   DWORD dwStyle = 0,
   DWORD dwExStyle = 0,
   _U_MENUorID MenuOrID = 0U,
   LPVOID lpCreateParam = NULL
);
HWND Create(
   LPCTSTR lpszClassName,
   CMessageMap* pObject,
   DWORD dwMsgMapID,
   HWND hWndParent,
   _U_RECT rect,
   LPCTSTR szWindowName = NULL,
   DWORD dwStyle = 0,
   DWORD dwExStyle = 0,
   _U_MENUorID MenuOrID = 0U,
   LPVOID lpCreateParam = NULL
);
```

Parameters

lpszClassName

[in] The name of an existing window class on which the contained window will be based.

pObject

[in] A pointer to the containing object that declares the message map. This object's class must derive from CMessageMap.

dwMsgMapID

[in] Identifies the message map that will process the contained window's messages. The default value, 0, specifies the default message map declared with BEGIN_MSG_MAP. To use an alternate message map declared with ALT_MSG_MAP(msgMapID), pass *msgMapID*.

hWndParent

[in] The handle to the parent or owner window.

rect

[in] A RECT structure specifying the position of the window. The **RECT** can be passed by pointer or by reference.

szWindowName

[in] Specifies the name of the window. The default value is **NULL**.

dwStyle

[in] The style of the window. The default value is **WS_CHILD | WS_VISIBLE**. For a list of possible values, see CreateWindow in the *Platform SDK*.

dwExStyle

[in] The extended window style. The default value is 0, meaning no extended style. For a list of possible values, see CreateWindowEx in the *Platform SDK*.

MenuOrID

[in] For a child window, the window identifier. For a top-level window, a menu handle for the window. The default value is **0U**.

lpCreateParam

[in] A pointer to window-creation data. For a full description, see the description for the final parameter to CreateWindowEx.

Return Value

If successful, the handle to the newly created window; otherwise, **NULL**.

Remarks

The existing window class name is saved in m_lpszClassName. **Create** then creates a window based on this new class. The newly created window is automatically attached to the **CContainedWindowT** object.

Note
Do not call **Create** if you have already called SubclassWindow.

Note
If 0 is used as the value for the *MenuOrID* parameter, it must be specified as 0U (the default value) to avoid a compiler error.

See Also

Reference

CContainedWindowT Class

CWindow::m_hWnd

Other Resources

CContainedWindowT Members

CContainedWindowT::DefWindowProc

Called by WindowProc to process messages not handled by the message map.

```
LRESULT DefWindowProc( )
LRESULT DefWindowProc(
   UINT uMsg,
   WPARAM wParam,
   LPARAM lParam
);
```

Parameters

uMsg

[in] The message sent to the window.

wParam

[in] Additional message-specific information.

lParam

[in] Additional message-specific information.

Return Value

The result of the message processing.

Remarks

By default, **DefWindowProc** calls the CallWindowProc Win32 function to send the message information to the window procedure specified in m_pfnSuperWindowProc.

See Also

Reference

CContainedWindowT Class

Other Resources

CContainedWindowT Members

CContainedWindowT::GetCurrentMessage

Returns the current message (**m_pCurrentMsg**).

```
const _ATL_MSG* GetCurrentMessage( );
```

Return Value

The current message, packaged in the **MSG** structure.

See Also

Reference

CContainedWindowT Class

Other Resources

CContainedWindowT Members

CContainedWindowT::RegisterWndSuperclass

Called by Create to register the window class of the contained window.

```
ATOM RegisterWndSuperClass( );
```

Return Value

If successful, an atom that uniquely identifies the window class being registered; otherwise, zero.

Remarks

This window class is based on an existing class but uses CContainedWindowT::WindowProc. The existing window class's name and window procedure are saved in m_lpszClassName and m_pfnSuperWindowProc, respectively.

See Also

Reference

CContainedWindowT Class

CContainedWindowT::CContainedWindowT

Other Resources

CContainedWindowT Members

CContainedWindowT::SubclassWindow

Subclasses the window identified by *hWnd* and attaches it to the **CContainedWindowT** object.

```
BOOL SubclassWindow(
   HWND hWnd
);
```

Parameters

hWnd

[in] The handle to the window being subclassed.

Return Value

TRUE if the window is successfully subclassed; otherwise, **FALSE**.

Remarks

The subclassed window now uses CContainedWindowT::WindowProc. The original window procedure is saved in m_pfnSuperWindowProc.

Note
Do not call **SubclassWindow** if you have already called Create.

See Also

Reference

CContainedWindowT Class

CContainedWindowT::UnsubclassWindow

Other Resources

CContainedWindowT Members

CContainedWindowT::SwitchMessageMap

Changes which message map will be used to process the contained window's messages.

```
void SwitchMessageMap(
   DWORD dwMsgMapID
);
```

Parameters

dwMsgMapID

[in] The message map identifier. To use the default message map declared with BEGIN_MSG_MAP, pass zero. To use an alternate message map declared with ALT_MSG_MAP(msgMapID), pass *msgMapID*.

Remarks

The message map must be defined in the containing object.

You initially specify the message map identifier in the constructor.

See Also

Reference

CContainedWindowT Class
CContainedWindowT::CContainedWindowT
CContainedWindowT::m_dwMsgMapID

Other Resources

CContainedWindowT Members

CContainedWindowT::UnsubclassWindow

Detaches the subclassed window from the **CContainedWindowT** object and restores the original window procedure, saved in m_pfnSuperWindowProc.

```
HWND UnsubclassWindow(
   BOOL bForce = FALSE
);
```

Parameters

bForce

[in] Set to **TRUE** to force the original window procedure to be restored even if the window procedure for this **CContainedWindowT** object is not currently active. If *bForce* is set to **FALSE** and the window procedure for this **CContainedWindowT** object is not currently active, the original window procedure will not be restored.

Return Value

The handle to the window previously subclassed. If *bForce* is set to **FALSE** and the window procedure for this **CContainedWindowT** object is not currently active, returns **NULL**.

Remarks

Use this method only if you want to restore the original window procedure before the window is destroyed. Otherwise, WindowProc will automatically do this when the window is destroyed.

See Also

Reference

CContainedWindowT Class

CContainedWindowT::SubclassWindow

Other Resources

CContainedWindowT Members

CContainedWindowT Data Members

For information about the data members in **CContainedWindowT**, see CContainedWindowT Members.

CContainedWindowT::m_dwMsgMapID

Holds the identifier of the message map currently being used for the contained window.

```
DWORD m_dwMsgMapID;
```

Remarks

This message map must be declared in the containing object.

The default message map, declared with BEGIN_MSG_MAP, is always identified by zero. An alternate message map, declared with ALT_MSG_MAP(msgMapID), is identified by *msgMapID*.

m_dwMsgMapID is first initialized by the constructor and can be changed by calling SwitchMessageMap. For an example, see the CContainedWindowT Overview.

See Also

Reference

CContainedWindowT Class
CContainedWindowT::m_pObject

Other Resources

CContainedWindowT Members

CContainedWindowT::m_lpszClassName

Specifies the name of an existing window class.

```
LPTSTR m_lpszClassName;
```

Remarks

When you create a window, Create registers a new window class that is based on this existing class but uses CContainedWindowT::WindowProc.

m_lpszClassName is initialized by the constructor. For an example, see the CContainedWindowT overview.

See Also

Reference

CContainedWindowT Class

Other Resources

CContainedWindowT Members

CContainedWindowT::m_pfnSuperWindowProc

If the contained window is subclassed, **m_pfnSuperWindowProc** points to the original window procedure of the window class.

```
WNDPROC m_pfnSuperWindowProc;
```

Remarks

If the contained window is superclassed, meaning it is based on a window class that modifies an existing class, **m_pfnSuperWindowProc** points to the existing window class's window procedure.

The DefWindowProc method sends message information to the window procedure saved in **m_pfnSuperWindowProc**.

See Also

Reference

CContainedWindowT Class
CContainedWindowT::Create
CContainedWindowT::SubclassWindow

Other Resources

CContainedWindowT Members

CContainedWindowT::m_pObject

Points to the object containing the **CContainedWindowT** object.

```
CMessageMap* m_pObject;
```

Remarks

This container, whose class must derive from CMessageMap, declares the message map used by the contained window.

m_pObject is initialized by the constructor. For an example, see the CContainedWindowT overview.

See Also

Reference

CContainedWindowT Class

CContainedWindowT::m_dwMsgMapID

Other Resources

CContainedWindowT Members

CContainedWindowT Static Functions

For information about the static functions in **CContainedWindowT**, see CContainedWindowT Members.

CContainedWindowT::WindowProc

This static method implements the window procedure.

```
static LRESULT CALLBACK WindowProc(
   HWND hWnd,
   UINT uMsg,
   WPARAM wParam,
   LPARAM lParam
);
```

Parameters

hWnd

[in] The handle to the window.

uMsg

[in] The message sent to the window.

wParam

[in] Additional message-specific information.

lParam

[in] Additional message-specific information.

Return Value

The result of the message processing.

Remarks

WindowProc directs messages to the message map identified by m_dwMsgMapID. If necessary, **WindowProc** calls DefWindowProc for additional message processing.

See Also

Reference

CContainedWindowT Class
BEGIN_MSG_MAP
ALT_MSG_MAP

Other Resources

CContainedWindowT Members

CCRTAllocator Class

This class provides methods for managing memory using CRT memory routines.

```
class ATL::CCRTAllocator
```

Remarks

This class is used by CHeapPtr to provide the CRT memory allocation routines. The counterpart class, CComAllocator, provides the same methods using COM routines.

Requirements

Header: atlcore.h

See Also

Reference

CHeapPtr Class

CComAllocator Class

Other Resources

CCRTAllocator Members

ATL Class Overview

CCRTAllocator Members

Static Functions

Allocate	Call this method to allocate memory.
Free	Call this method to free memory.
Reallocate	Call this method to reallocate memory.

See Also

Reference

CCRTAllocator Class

CCRTAllocator Static Functions

For information about the static functions in **CCRTAllocator**, see CCRTAllocator Members.

CCRTAllocator::Allocate

Call this static function to allocate memory.

```
static void* Allocate(
   size_t nBytes
) throw( );
```

Parameters

nBytes

The number of bytes to allocate.

Return Value

Returns a void pointer to the allocated space, or NULL if there is insufficient memory available.

Remarks

Allocates memory. See malloc for more details.

See Also

Reference

CCRTAllocator Class
CCRTAllocator::Reallocate
CCRTAllocator::Free

Other Resources

CCRTAllocator Members

CCRTAllocator::Free

Call this static function to free memory.

```
static void Free(
   void* p
) throw( );
```

Parameters

p

Pointer to the allocated memory.

Remarks

Frees the allocated memory. See free for more details.

See Also

Reference

CCRTAllocator Class
CCRTAllocator::Allocate
CCRTAllocator::Reallocate

Other Resources

CCRTAllocator Members

CCRTAllocator::Reallocate

Call this static function to reallocate memory.

```
static void* Reallocate(
   void* p,
   size_t nBytes
) throw( );
```

Parameters

p

Pointer to the allocated memory.

nBytes

The number of bytes to reallocate.

Return Value

Returns a void pointer to the allocated space, or NULL if there is insufficient memory.

Remarks

Resizes the amount of allocated memory. See realloc for more details.

See Also

Reference

CCRTAllocator Class
CCRTAllocator::Allocate
CCRTAllocator::Free

Other Resources

CCRTAllocator Members

CCRTHeap Class

This class implements IAtlMemMgr using the CRT heap functions.

```
class CCRTHeap : public IAtlMemMgr
```

Remarks

CCRTHeap implements memory allocation functions using the CRT heap functions, including malloc, free, realloc, and _msize.

Example

See the example for IAtlMemMgr.

Requirements

Header: atlmem.h

See Also

Reference

CComHeap Class
CWin32Heap Class
CLocalHeap Class
CGlobalHeap Class
IAtlMemMgr Class

Other Resources

CCRTHeap Members
ATL Class Overview

CCRTHeap Members

Methods

Allocate	Call this method to allocate a block of memory.
Free	Call this method to free a block of memory allocated by this memory manager.
GetSize	Call this method to get the allocated size of a memory block allocated by this memory manager.
Reallocate	Call this method to reallocate memory allocated by this memory manager.

See Also

Reference

CCRTHeap Class

Other Resources

IAtlMemMgr Members

CCRTHeap Methods

For information about the methods in **CCRTHeap**, see CCRTHeap Members.

CCRTHeap::Allocate

Call this method to allocate a block of memory.

```
virtual void* Allocate(
   size_t nBytes
) throw( );
```

Parameters

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call CCRTHeap::Free or CCRTHeap::Reallocate to free the memory allocated by this method.

Implemented using malloc.

See Also

Reference

CCRTHeap Class
CCRTHeap::Reallocate
CCRTHeap::Free

Other Resources

CCRTHeap Members

CCRTHeap::Free

Call this method to free a block of memory allocated by this memory manager.

```
virtual void Free(
   void* p
) throw( );
```

Parameters

p

Pointer to memory previously allocated by this memory manager. NULL is a valid value and does nothing.

Remarks

Implemented using free.

See Also

Reference

CCRTHeap Class
CCRTHeap::Allocate
CCRTHeap::Reallocate

Other Resources

CCRTHeap Members

CCRTHeap::GetSize

Call this method to get the allocated size of a memory block allocated by this memory manager.

```
virtual size_t GetSize(
   void* p
) throw( );
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

Return Value

Returns the size of the allocated memory block in bytes.

Remarks

Implemented using _msize.

See Also

Reference

CCRTHeap Class

Other Resources

CCRTHeap Members

CCRTHeap::Reallocate

Call this method to reallocate memory allocated by this memory manager.

```
virtual void* Reallocate(
   void* p,
   size_t nBytes
) throw( );
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call CCRTHeap::Free to free the memory allocated by this method. Implemented using realloc.

See Also

Reference

CCRTHeap Class
CCRTHeap::Allocate
CCRTHeap::Free

Other Resources

CCRTHeap Members

CDacl Class

This class is a wrapper for a DACL (discretionary access-control list) structure.

```
class CDacl : public CAcl
```

Remarks

An object's security descriptor can contain a DACL. A DACL contains zero or more ACEs (access-control entries) that identify the users and groups who can access the object. If a DACL is empty (that is, it contains zero ACEs), no access is explicitly granted, so access is implicitly denied. However, if an object's security descriptor does not have a DACL, the object is unprotected and everyone has complete access.

To retrieve an object's DACL, you must be the object's owner or have READ_CONTROL access to the object. To change an object's DACL, you must have WRITE_DAC access to the object.

Use the class methods provided to create, add, remove, and delete ACEs from the **CDacl** object. See also AtlGetDacl and AtlSetDacl.

For an introduction to the access control model in Windows, see Access Control in the *Platform SDK*.

Requirements

Header: atlsecurity.h

See Also

Tasks

ATLSecurity Sample: Demonstrates ATL Security Classes

Reference

CAcl Class

ACLs

ACEs

Other Resources

CDacl Members

ATL Class Overview

Security Global Functions

CDacl Members

Methods

CDacl	The constructor.
~CDacl	The destructor.
AddAllowedAce	Adds an allowed ACE (access-control entry) to the **CDacl** object.
AddDeniedAce	Adds a denied ACE to the **CDacl** object.
GetAceCount	Returns the number of ACEs (access-control entries) in the **CDacl** object.
RemoveAce	Removes a specific ACE (access-control entry) from the **CDacl** object.
RemoveAllAces	Removes all of the ACEs contained in the **CDacl** object.

Operators

operator =	Assignment operator.

See Also

Reference

CDacl Class

CDacl Methods

For information about the methods in **CDacl**, see CDacl Members.

CDacl::AddAllowedAce

Adds an allowed ACE (access-control entry) to the **CDacl** object.

```
bool AddAllowedAce(
   const CSid & rSid,
   ACCESS_MASK AccessMask,
   BYTE AceFlags = 0
) throw(...);
bool AddAllowedAce(
   const CSid & rSid,
   ACCESS_MASK AccessMask,
   BYTE AceFlags,
   const GUID * pObjectType,
   const GUID * pInheritedObjectType
) throw(...);
```

Parameters

rSid

A CSid object.

AccessMask

Specifies the mask of access rights to be allowed for the specified **CSid** object.

AceFlags

A set of bit flags that control ACE inheritance.

pObjectType

The object type.

pInheritedObjectType

The inherited object type.

Return Value

Returns **true** if the ACE is added to the **CDacl** object, **false** on failure.

Remarks

A **CDacl** object contains zero or more ACEs (access-control entries) that identify the users and groups who can access the object. This method adds an ACE that allows access to the **CDacl** object.

Note
The second form of **AddAllowedAce** is only available on Windows 2000 and later.

See ACE_HEADER for a description of the various flags which can be set in the *AceFlags* parameter.

See Also

Reference

CDacl Class
CDacl::AddDeniedAce
ACCESS_MASK

Other Resources

CDacl Members

CDacl::AddDeniedAce

Adds a denied ACE (access-control entry) to the **CDacl** object.

```
bool AddDeniedAce(
   const CSid & rSid,
   ACCESS_MASK AccessMask,
   BYTE AceFlags = 0
) throw(...);
bool AddDeniedAce(
   const CSid & rSid,
   ACCESS_MASK AccessMask,
   BYTE AceFlags,
   const GUID * pObjectType,
   const GUID * pInheritedObjectType
) throw(...);
```

Parameters

rSid

A **CSid** object.

AccessMask

Specifies the mask of access rights to be denied for the specified **CSid** object.

AceFlags

A set of bit flags that control ACE inheritance. Defaults to 0 in the first form of the method.

pObjectType

The object type.

pInheritedObjectType

The inherited object type.

Return Value

Returns **true** if the ACE is added to the **CDacl** object, **false** on failure.

Remarks

A **CDacl** object contains zero or more ACEs (access-control entries) that identify the users and groups who can access the object. This method adds an ACE that denies access to the **CDacl** object.

Note
The second form of **AddDeniedAce** is only available on Windows 2000 and later.

See ACE_HEADER for a description of the various flags which can be set in the *AceFlags* parameter.

See Also

Reference

CDacl Class
CDacl::AddAllowedAce
CDacl::RemoveAllAces
ACCESS_MASK

Other Resources

CDacl Members

CDacl::CDacl

The constructor.

```
CDacl (
   const ACL & rhs
) throw(...);
CDacl ( ) throw( );
```

Parameters

rhs

An existing **ACL** (access-control list) structure.

Remarks

The **CDacl** object can be optionally created using an existing **ACL** structure. It is important to note that only a DACL (discretionary access-control list), and not a SACL (system access-control list), should be passed as this parameter. In debug builds, passing a SACL will cause an ASSERT. In release builds, passing a SACL will cause the ACEs (access-control entries) in the ACL to be ignored, and no error will occur.

See Also

Reference

CDacl Class

Other Resources

CDacl Members

CDacl::~CDacl

The destructor.

```
~CDacl ( ) throw( );
```

Remarks

The destructor frees any resources acquired by the object, including all ACEs (access-control entries) using CDacl::RemoveAllAces.

See Also

Reference

CDacl Class

CDacl::CDacl

Other Resources

CDacl Members

CDacl::GetAceCount

Returns the number of ACEs (access-control entries) in the **CDacl** object.

```
UINT GetAceCount( ) const throw( );
```

Return Value

Returns the number of ACEs contained in the **CDacl** object.

See Also

Reference

CDacl Class

Other Resources

CDacl Members

CDacl::RemoveAce

Removes a specific ACE (access-control entry) from the **CDacl** object.

```
void RemoveAce(
   UINT nIndex
) throw( );
```

Parameters

nIndex

Index to the ACE entry to remove.

Remarks

This method is derived from CAtlArray::RemoveAt.

See Also

Reference

CDacl Class

CDacl::RemoveAllAces

Other Resources

CDacl Members

CDacl::RemoveAllAces

Removes all of the ACEs (access-control entries) contained in the **CDacl** object.

```
void RemoveAllAces( ) throw( );
```

Remarks

Removes every **ACE** (access-control entry) structure (if any) in the **CDacl** object.

See Also

Reference

CDacl Class

CDacl::RemoveAce

Other Resources

CDacl Members

CDacl Operators

For information about the operators in **CDacl**, see CDacl Members.

CDacl::operator =

Assignment operator.

```
CDacl & operator =(
   const ACL & rhs
) throw(...);
```

Parameters

rhs

The ACL (access-control list) to assign to the existing object.

Return Value

Returns a reference to the updated **CDacl** object.

Remarks

You should ensure that you only pass a DACL (discretionary access-control list) to this function. Passing a SACL (system access-control list) to this function will cause an ASSERT in debug builds but will cause no error in release builds.

See Also

Reference

CDacl Class

Other Resources

CDacl Members

CDefaultCharTraits Class

This class provides two static functions for converting characters between uppercase and lowercase.

```
template <
   typename T
>
class CDefaultCharTraits
```

Parameters

T

The type of data to be stored in the collection.

Remarks

This class provides functions that are utilized by the class CStringElementTraitsI.

Requirements

Header: atlcoll.h

See Also

Other Resources

CDefaultCharTraits Members

ATL Class Overview

CDefaultCharTraits Members

Static Functions

CharToUpper	Call this function to convert a character to uppercase.
CharToLower	Call this function to convert a character to lowercase.

See Also

Reference

CDefaultCharTraits Class

CDefaultCharTraits Static Functions

For information about the static functions in **CDefaultCharTraits**, see CDefaultCharTraits Members.

CDefaultCharTraits::CharToLower

Call this function to convert a character to lowercase.

```
static wchar_t CharToLower(
   wchar_t x
);
static char CharToLower(
   char x
);
```

Parameters

x

The character to convert to lowercase.

Example

```
// CDefaultCharTraits_CharToLower.cpp
#include <atlcoll.h>
#include <stdio.h>
int main() {
   printf("%c\n", CDefaultCharTraits<char>::CharToLower('A'));
}
```

Output

```
a
```

See Also

Reference

CDefaultCharTraits::CharToUpper

Other Resources

CDefaultCharTraits Members

CDefaultCharTraits::CharToUpper

Call this function to convert a character to uppercase.

```
static wchar_t CharToUpper(
   wchar_t x
);
static char CharToUpper(
   char x
);
```

Parameters

x

The character to convert to uppercase.

See Also

Reference

CDefaultCharTraits::CharToLower

Other Resources

CDefaultCharTraits Members

CDefaultCompareTraits Class

This class provides default element comparison functions.

```
template<
   typename T
>
class CDefaultCompareTraits
```

Parameters

T

The type of data to be stored in the collection.

Remarks

This class contains two static functions for comparing elements stored in a collection class object. This class is utilized by the CDefaultElementTraits Class.

For more information, see ATL Collection Classes.

Requirements

Header: atlcoll.h

See Also

Other Resources

CDefaultCompareTraits Members

ATL Class Overview

CDefaultCompareTraits Members

Static Functions

CompareElements	Call this function to compare two elements for equality.
CompareElementsOrdered	Call this function to determine the greater and lesser element.

See Also

Reference

CDefaultCompareTraits Class

CDefaultCompareTraits Static Functions

For information about the static functions in **CDefaultCompareTraits**, see CDefaultCompareTraits Members.

CDefaultCompareTraits::CompareElements

Call this function to compare two elements for equality.

```
static bool CompareElements(
   const T& element1,
   const T& element2
);
```

Parameters

element1

The first element.

element2

The second element.

Return Value

Returns true if the elements are equal, false otherwise.

Remarks

The default implementation of this function is the equality (**==**) operator. For objects other than simple data types, this function may need to be overridden.

See Also

Reference

CDefaultCompareTraits Class

CDefaultCompareTraits::CompareElementsOrdered

Other Resources

CDefaultCompareTraits Members

CDefaultCompareTraits::CompareElementsOrdered

Call this function to determine the greater and lesser element.

```
static int CompareElementsOrdered(
   const T& element1,
   const T& element2
);
```

Parameters

element1

The first element.

element2

The second element.

Return Value

Returns an integer based on the following table:

Condition	Return value
element1 < *element2*	<0
element1 == *element2*	0
element1 > *element2*	>0

Remarks

The default implementation of this function uses the **==**, **<**, and **>** operators. For objects other than simple data types, this function may need to be overridden.

See Also

Reference

CDefaultCompareTraits Class

CDefaultCompareTraits::CompareElementsOrdered

Other Resources

CDefaultCompareTraits Members

CDefaultElementTraits Class

This class provides default methods and functions for a collection class.

```
template<
   typename T
>
class CDefaultElementTraits : public CElementTraitsBase< T >,
   public CDefaultHashTraits< T >,
   public CDefaultCompareTraits< T >
```

Parameters

T

The type of data to be stored in the collection.

Remarks

This class provides default static functions and methods for moving, copying, comparing, and hashing elements stored in a collection class object. This class derives its functions and methods from CElementTraitsBase, CDefaultHashTraits, and CDefaultCompareTraits, and is utilized by CElementTraits, CPrimitiveElementTraits, and CHeapPtrElementTraits.

For more information, see ATL Collection Classes.

Requirements

Header: atlcoll.h

See Also

Other Resources

ATL Class Overview

CDefaultHashTraits Class

This class provides a static function for calculating hash values.

```
template<
   typename T
>
class CDefaultHashTraits
```

Parameters

T

The type of data to be stored in the collection.

Remarks

This class contains a single static function that returns a hash value for a given element. This class is utilized by the CDefaultElementTraits Class.

For more information, see ATL Collection Classes.

Requirements

Header: atlcoll.h

See Also

Other Resources

CDefaultHashTraits Members

ATL Class Overview

CDefaultHashTraits Members

Static Functions

Hash	Call this function to calculate a hash value for a given element.

See Also

Reference

CDefaultHashTraits Class

CDefaultHashTraits Static Functions

For information about the static functions in **CDefaultHashTraits**, see CDefaultHashTraits Members.

CDefaultHashTraits::Hash

Call this function to calculate a hash value for a given element.

```
static ULONG Hash(
   const T& element
) throw( );
```

Parameters

element

The element.

Return Value

Returns the hash value.

Remarks

The default hashing algorithm is very simple: the return value is the element number. Override this function if a more complicated algorithm is required.

See Also

Reference

CDefaultHashTraits Class

Other Resources

CDefaultHashTraits Members

CDialogImpl Class

This class provides methods for creating a modal or modeless dialog box.

```
template <
   class T,
   class TBase = CWindow
>
class ATL_NO_VTABLE CDialogImpl :
   public CDialogImplBaseT< TBase >
```

Parameters

T

Your class, derived from **CDialogImpl**.

TBase

The base class of your new class. The default base class is CWindow.

Remarks

With **CDialogImpl** you can create a modal or modeless dialog box. **CDialogImpl** provides the dialog box procedure, which uses the default message map to direct messages to the appropriate handlers.

The base class destructor **~CWindowImplRoot** ensures that the window is gone before destroying the object.

CDialogImpl derives from **CDialogImplBaseT**, which in turn derives from **CWindowImplRoot**.

Note
Your class must define an **IDD** member that specifies the dialog template resource ID. For example, the ATL Project Wizard a utomatically adds the following line to your class:

```
enum { IDD = IDD_MYDIALOG };
```

where `MyDialog` is the **Short name** entered in the wizard's **Names** page.

For more information about	See
Creating controls	ATL Tutorial
Using dialog boxes in ATL	ATL Window Classes
ATL Project Wizard	Creating an ATL Project
Dialog boxes	Dialog Boxes and subsequent topics in the *Platform SDK*

Requirements

Header: atlwin.h

See Also

Reference

BEGIN_MSG_MAP

Other Resources

CDialogImpl Members

ATL Class Overview

CDialogImpl Members

Methods

Create	Creates a modeless dialog box.
DestroyWindow	Destroys a modeless dialog box.
DoModal	Creates a modal dialog box.
EndDialog	Destroys a modal dialog box.

CDialogImplBaseT Methods

GetDialogProc	Returns the current dialog box procedure.
MapDialogRect	Maps the dialog-box units of the specified rectangle to screen units (pixels).
OnFinalMessage	Called after receiving the last message, typically **WM_NCDESTROY**.

Static Functions

DialogProc	Processes messages sent to the dialog box.
StartDialogProc	Called when the first message is received to process messages sent to the dialog box.

See Also

Reference

CDialogImpl Class

CDialogImpl Methods

For information about the methods in **CDialogImpl**, see CDialogImpl Members.

CDialogImpl::Create

Creates a modeless dialog box.

```
HWND Create(
   HWND hWndParent,
   LPARAM dwInitParam = NULL
);
HWND Create(
   HWND hWndParent,
   RECT&,
   LPARAM dwInitParam = NULL
);
```

Parameters

hWndParent

[in] The handle to the owner window.

RECT& *rect*

[in] A RECT structure specifying the dialog's size and position.

dwInitParam

[in] Specifies the value to pass to the dialog box in the **lParam** parameter of the **WM_INITDIALOG** message.

Return Value

The handle to the newly created dialog box.

Remarks

This dialog box is automatically attached to the **CDialogImpl** object. To create a modal dialog box, call DoModal. The second override above is used only with CComControl.

See Also

Reference

CDialogImpl Class
CWindow::m_hWnd

Other Resources

CDialogImpl Members

CDialogImpl::DestroyWindow

Destroys a modeless dialog box.

```
BOOL DestroyWindow( );
```

Return Value

TRUE if the dialog box was successfully destroyed; otherwise **FALSE**.

Remarks

Returns **TRUE** if the dialog box was successfully destroyed; otherwise **FALSE**.

See Also

Reference

CDialogImpl Class

CDialogImpl::Create

Other Resources

CDialogImpl Members

CDialogImpl::DoModal

Creates a modal dialog box.

```
INT_PTR DoModal(
   HWND hWndParent = ::GetActiveWindow( ),
   LPARAM dwInitParam = NULL
);
```

Parameters

hWndParent

[in] The handle to the owner window. The default value is the return value of the GetActiveWindow Win32 function.

dwInitParam

[in] Specifies the value to pass to the dialog box in the **lParam** parameter of the **WM_INITDIALOG** message.

Return Value

If successful, the value of the *nRetCode* parameter specified in the call to EndDialog. Otherwise, -1.

Remarks

This dialog box is automatically attached to the **CDialogImpl** object.

To create a modeless dialog box, call Create.

See Also

Reference

CDialogImpl Class
CWindow::m_hWnd
CDialogImpl::EndDialog

Other Resources

CDialogImpl Members

CDialogImpl::EndDialog

Destroys a modal dialog box.

```
BOOL EndDialog(
   int nRetCode
);
```

Parameters

nRetCode

[in] The value to be returned by CDialogImpl::DoModal.

Return Value

TRUE if the dialog box is destroyed; otherwise, **FALSE**.

Remarks

EndDialog must be called through the dialog procedure. After the dialog box is destroyed, Windows uses the value of *nRetCode* as the return value for **DoModal**, which created the dialog box.

Note
Do not call **EndDialog** to destroy a modeless dialog box. Call CWindow::DestroyWindow instead.

See Also

Reference

CDialogImpl Class

CDialogImpl::DialogProc

Other Resources

CDialogImpl Members

CDialogImpl::GetDialogProc

Returns **DialogProc**, the current dialog box procedure.

```
virtual WNDPROC GetDialogProc( );
```

Return Value

The current dialog box procedure.

Remarks

Override this method to replace the dialog procedure with your own.

See Also

Reference

CDialogImpl Class
CDialogImpl::DialogProc
CDialogImpl::OnFinalMessage

Other Resources

CDialogImpl Members

CDialogImpl::MapDialogRect

Converts (maps) the dialog-box units of the specified rectangle to screen units (pixels).

```
BOOL MapDialogRect(
   LPRECT lpRect
);
```

Parameters

lpRect

Points to a **CRect** object or RECT structure that is to receive the client coordinates of the update that encloses the update region.

Return Value

Nonzero if the update succeeds; 0 if the update fails. To get extended error information, call **GetLastError**.

Remarks

The function replaces the coordinates in the specified **RECT** structure with the converted coordinates, which allows the structure to be used to create a dialog box or position a control within a dialog box.

See Also

Reference

CDialogImpl Class

MapDialogRect

Other Resources

CDialogImpl Members

CDialogImpl::OnFinalMessage

Called after receiving the last message (typically **WM_NCDESTROY**).

```
virtual void OnFinalMessage(
   HWND hWnd
);
```

Parameters

hWnd

[in] A handle to the window being destroyed.

Remarks

Note that if you want to automatically delete your object upon the window destruction, you can call `delete this;` here.

See Also

Reference

CDialogImpl Class

CDialogImpl::GetDialogProc

Other Resources

CDialogImpl Members

CDialogImpl Static Functions

For information about the static functions in **CDialogImpl**, see CDialogImpl Members.

CDialogImpl::DialogProc

This static function implements the dialog box procedure.

```
static LRESULT CALLBACK DialogProc(
   HWND hWnd,
   UINT uMsg,
   WPARAM wParam,
   LPARAM lParam
);
```

Parameters

hWnd

[in] The handle to the dialog box.

uMsg

[in] The message sent to the dialog box.

wParam

[in] Additional message-specific information.

lParam

[in] Additional message-specific information.

Return Value

TRUE if the message is processed; otherwise, **FALSE**.

Remarks

DialogProc uses the default message map to direct messages to the appropriate handlers.

You can override **DialogProc** to provide a different mechanism for handling messages.

See Also

Reference

CDialogImpl Class

BEGIN_MSG_MAP

Other Resources

CDialogImpl Members

CDialogImpl::StartDialogProc

Called only once, when the first message is received, to process messages sent to the dialog box.

```
static LRESULT CALLBACK StartDialogProc(
   HWND hWnd,
   UINT uMsg,
   WPARAM wParam,
   LPARAM lParam
);
```

Parameters

hWnd

[in] The handle to the dialog box.

uMsg

[in] The message sent to the dialog box.

wParam

[in] Additional message-specific information.

lParam

[in] Additional message-specific information.

Return Value

The window procedure.

Remarks

After the initial call to **StartDialogProc**, **DialogProc** is set as a dialog procedure, and further calls go there.

See Also

Reference

CDialogImpl Class

CDialogImpl::DialogProc

Other Resources

CDialogImpl Members

CDynamicChain Class

This class provides methods supporting the dynamic chaining of message maps.

```
class CDynamicChain
```

Remarks

CDynamicChain manages a collection of message maps, enabling a Windows message to be directed, at run time, to another object's message map.

To add support for dynamic chaining of message maps, do the following:

- Derive your class from **CDynamicChain**. In the message map, specify the CHAIN_MSG_MAP_DYNAMIC macro to chain to another object's default message map.
- Derive every class you want to chain to from CMessageMap. **CMessageMap** allows an object to expose its message maps to other objects.
- Call **CDynamicChain::SetChainEntry** to identify which object and which message map you want to chain to.

For example, suppose your class is defined as follows:

```
class CMyWindow : public CDynamicChain, ...
{
public:
   ...

   BEGIN_MSG_MAP(CMyWindow)
      MESSAGE_HANDLER(WM_PAINT, OnPaint)
      MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
      // dynamically chain to the default
      // message map in another object
      CHAIN_MSG_MAP_DYNAMIC(1313)
                // '1313' identifies the object
                // and the message map that will be
                // chained to. '1313' is defined
                // through the SetChainEntry method
   END_MSG_MAP()

   LRESULT OnPaint(UINT uMsg, WPARAM wParam,
                   LPARAM lParam, BOOL& bHandled)
   { ... }

   LRESULT OnSetFocus(UINT uMsg, WPARAM wParam,
                      LPARAM lParam, BOOL& bHandled)
   { ... }

};
```

The client then calls `CMyWindow::SetChainEntry`:

```
// myCtl is a CMyWindow object
myCtl.SetChainEntry(1313, &chainedObj);
```

where `chainedObj` is the chained object and is an instance of a class derived from **CMessageMap**. Now, if `myCtl` receives a message that is not handled by `OnPaint` or `OnSetFocus`, the window procedure directs the message to `chainedObj`'s default message map.

For more information about message map chaining, see Message Maps in the article "ATL Window Classes."

Requirements

Header: atlwin.h

See Also

Reference

CWindowImpl Class

Other Resources

CDynamicChain Members

ATL Class Overview

CDynamicChain Members

Methods

~CDynamicChain	The destructor.
CallChain	Directs a Windows message to another object's message map.
CDynamicChain	The constructor.
RemoveChainEntry	Removes a message map entry from the collection.
SetChainEntry	Adds a message map entry to the collection or modifies an existing entry.

See Also

Reference

CDynamicChain Class

CDynamicChain Methods

For information about the methods in **CDynamicChain**, see CDynamicChain Members.

CDynamicChain::CallChain

Directs the Windows message to another object's message map.

```
BOOL CallChain(
   DWORD dwChainID,
   HWND hWnd,
   UINT uMsg,
   WPARAM wParam,
   LPARAM lParam,
   LRESULT& lResult
);
```

Parameters

dwChainID

[in] The unique identifier associated with the chained object and its message map.

hWnd

[in] The handle to the window receiving the message.

uMsg

[in] The message sent to the window.

wParam

[in] Additional message-specific information.

lParam

[in] Additional message-specific information.

lResult

[out] The result of the message processing.

Return Value

TRUE if the message is fully processed; otherwise, **FALSE**.

Remarks

For the window procedure to invoke **CallChain**, you must specify the CHAIN_MSG_MAP_DYNAMIC macro in your message map. For an example, see the CDynamicChain overview.

CallChain requires a previous call to SetChainEntry to associate the *dwChainID* value with an object and its message map.

See Also

Reference

CDynamicChain Class

Other Resources

CDynamicChain Members

CDynamicChain::CDynamicChain

The constructor.

```
CDynamicChain( );
```

See Also

Reference

CDynamicChain Class

Other Resources

CDynamicChain Members

CDynamicChain::~CDynamicChain

The destructor.

```
~CDynamicChain( );
```

Remarks

Frees all allocated resources.

See Also

Reference

CDynamicChain Class

Other Resources

CDynamicChain Members

CDynamicChain::RemoveChainEntry

Removes the specified message map from the collection.

```
BOOL RemoveChainEntry(
   DWORD dwChainID
);
```

Parameters

dwChainID

[in] The unique identifier associated with the chained object and its message map. You originally define this value through a call to SetChainEntry.

Return Value

TRUE if the message map is successfully removed from the collection. Otherwise, **FALSE**.

See Also

Reference

CDynamicChain Class

Other Resources

CDynamicChain Members

CDynamicChain::SetChainEntry

Adds the specified message map to the collection.

```
BOOL SetChainEntry(
   DWORD dwChainID,
   CMessageMap* pObject,
   DWORD dwMsgMapID = 0
);
```

Parameters

dwChainID

[in] The unique identifier associated with the chained object and its message map.

pObject

[in] A pointer to the chained object declaring the message map. This object must derive from CMessageMap.

dwMsgMapID

[in] The identifier of the message map in the chained object. The default value is 0, which identifies the default message map declared with BEGIN_MSG_MAP. To specify an alternate message map declared with ALT_MSG_MAP(msgMapID), pass *msgMapID*.

Return Value

TRUE if the message map is successfully added to the collection. Otherwise, **FALSE**.

Remarks

If the *dwChainID* value already exists in the collection, its associated object and message map are replaced by *pObject* and *dwMsgMapID*, respectively. Otherwise, a new entry is added.

See Also

Reference

CDynamicChain Class
CDynamicChain::CallChain
CDynamicChain::RemoveChainEntry
CHAIN_MSG_MAP_DYNAMIC

Other Resources

CDynamicChain Members

CElementTraits Class

This class is used by collection classes to provide methods and functions for moving, copying, comparison, and hashing operations.

```
template<
   typename T
>
class CElementTraits : public CDefaultElementTraits< T >
```

Parameters

T

The type of data to be stored in the collection.

Remarks

This class provides default static functions and methods for moving, copying, comparing, and hashing elements stored in a collection class object. **CElementTraits** is specified as the default provider of these operations by the collection classes CAtlArray, CAtlList, CRBMap, CRBMultiMap, and CRBTree.

The default implementations will suffice for simple data types, but if the collection classes are used to store more complex objects, the functions and methods must be overridden by user-supplied implementations.

For more information, see ATL Collection Classes.

Requirements

Header: atlcoll.h

See Also

Reference

CDefaultElementTraits Class

Other Resources

ATL Class Overview

CElementTraitsBase Class

This class provides default copy and move methods for a collection class.

```
template<
   typename T
>
class CElementTraitsBase
```

Parameters

T

The type of data to be stored in the collection.

Remarks

This base class defines methods for copying and relocating elements in a collection class. It is utilized by the classes CDefaultElementTraits, CStringRefElementTraits, and CStringElementTraitsI.

For more information, see ATL Collection Classes.

Requirements

Header: atlcoll.h

See Also

Other Resources

CElementTraitsBase Members

ATL Class Overview

CElementTraitsBase Members

Methods

CopyElements	Call this method to copy elements stored in a collection class object.
RelocateElements	Call this method to relocate elements stored in a collection class object.

Typedefs

INARGTYPE	The data type to use for adding elements to the collection class object.
OUTARGTYPE	The data type to use for retrieving elements from the collection class object.

See Also

Reference

CElementTraitsBase Class

CElementTraitsBase Methods

For information about the methods in **CElementTraitsBase**, see CElementTraitsBase Members.

CElementTraitsBase::CopyElements

Call this method to copy elements stored in a collection class object.

```
static void CopyElements(
   T* pDest,
   const T* pSrc,
   size_t nElements
);
```

Parameters

pDest

Pointer to the first element that will receive the copied data.

pSrc

Pointer to the first element to copy.

nElements

The number of elements to copy.

Remarks

The source and destination elements should not overlap.

See Also

Reference

CElementTraitsBase Class

CElementTraitsBase::RelocateElements

Other Resources

CElementTraitsBase Members

CElementTraitsBase::RelocateElements

Call this method to relocate elements stored in a collection class object.

```
static void RelocateElements(
   T* pDest,
   T* pSrc,
   size_t nElements
);
```

Parameters

pDest

Pointer to the first element that will receive the relocated data.

pSrc

Pointer to the first element to relocate.

nElements

The number of elements to relocate.

Remarks

This method calls memmove, which is sufficient for most data types. If the objects being moved contain pointers to their own members, this method will need to be overridden.

See Also

Reference

CElementTraitsBase Class

CElementTraitsBase::CopyElements

Other Resources

CElementTraitsBase Members

CElementTraitsBase Typedefs

For information about the typedefs in **CElementTraitsBase**, see CElementTraitsBase Members.

CElementTraitsBase::INARGTYPE

The data type to use for adding elements to the collection.

```
typedef const T& INARGTYPE;
```

See Also

Reference

CElementTraitsBase Class

CElementTraitsBase::OUTARGTYPE

Other Resources

CElementTraitsBase Members

CElementTraitsBase::OUTARGTYPE

The data type to use for retrieving elements from the collection.

```
typedef T& OUTARGTYPE;
```

See Also

Reference

CElementTraitsBase Class

CElementTraitsBase::INARGTYPE

Other Resources

CElementTraitsBase Members

CFirePropNotifyEvent Class

This class provides methods for notifying the container's sink regarding control property changes.

```
class CFirePropNotifyEvent
```

Remarks

CFirePropNotifyEvent has two methods that notify the container's sink that a control property has changed or is about to change.

If the class implementing your control is derived from **IPropertyNotifySink**, the **CFirePropNotifyEvent** methods are invoked when you call **FireOnRequestEdit** or **FireOnChanged**. If your control class is not derived from **IPropertyNotifySink**, calls to these functions return **S_OK**.

For more information about creating controls, see the ATL Tutorial.

Requirements

Header: atlctl.h

See Also

Other Resources

CFirePropNotifyEvent Members

ATL Class Overview

CFirePropNotifyEvent Members

Static Functions

FireOnChanged	Notifies the container's sink that a control property has changed.
FireOnRequestEdit	Notifies the container's sink that a control property is about to change.

See Also

Reference

CFirePropNotifyEvent Class

CFirePropNotifyEvent Static Functions

For information about the static functions in **CFirePropNotifyEvent**, see CFirePropNotifyEvent Members.

CFirePropNotifyEvent::FireOnChanged

Notifies all connected IPropertyNotifySink interfaces (on every connection point of the object) that the specified object property has changed.

```
static HRESULT FireOnChanged(
   IUnknown* pUnk,
      DISPID dispID
);
```

Parameters

pUnk

[in] Pointer to the **IUnknown** of the object sending the notification.

dispID

[in] Identifier of the property that has changed.

Return Value

One of the standard **HRESULT** values.

Remarks

This function is safe to call even if your control does not support connection points.

See Also

Reference

CFirePropNotifyEvent Class
CFirePropNotifyEvent::FireOnRequestEdit
CComControl::FireOnChanged

Other Resources

CFirePropNotifyEvent Members

CFirePropNotifyEvent::FireOnRequestEdit

Notifies all connected IPropertyNotifySink interfaces (on every connection point of the object) that the specified object property is about to change.

```
static HRESULT FireOnRequestEdit(
   IUnknown* pUnk,
      DISPID dispID
);
```

Parameters

pUnk

[in] Pointer to the **IUnknown** of the object sending the notification.

dispID

[in] Identifier of the property about to change.

Return Value

One of the standard **HRESULT** values.

Remarks

This function is safe to call even if your control does not support connection points.

See Also

Reference

CFirePropNotifyEvent Class
CFirePropNotifyEvent::FireOnChanged
CComControl::FireOnRequestEdit

Other Resources

CFirePropNotifyEvent Members

CGlobalHeap Class

This class implements IAtlMemMgr using the Win32 global heap functions.

```
class CGlobalHeap : public IAtlMemMgr
```

Remarks

CGlobalHeap implements memory allocation functions using the Win32 global heap functions.

Note
The global heap functions are slower than other memory management functions and do not provide as many features. Therefore, new applications should use the heap functions. These are available in the CWin32Heap class. Global functions are still used by DDE and the clipboard functions.

Example

See the example for IAtlMemMgr.

Requirements

Header: atlmem.h

See Also

Reference

CComHeap Class
CWin32Heap Class
CLocalHeap Class
CCRTHeap Class
IAtlMemMgr Class

Other Resources

CGlobalHeap Members
ATL Class Overview

CGlobalHeap Members

Methods

Allocate	Call this method to allocate a block of memory.
Free	Call this method to free a block of memory allocated by this memory manager.
GetSize	Call this method to get the allocated size of a memory block allocated by this memory manager.
Reallocate	Call this method to reallocate memory allocated by this memory manager.

See Also

Reference

CGlobalHeap Class

Other Resources

IAtlMemMgr Members

CGlobalHeap Methods

For information about the methods in **CGlobalHeap**, see CGlobalHeap Members.

CGlobalHeap::Allocate

Call this method to allocate a block of memory.

```
virtual void* Allocate(
   size_t nBytes
) throw( );
```

Parameters

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call CGlobalHeap::Free or CGlobalHeap::Reallocate to free the memory allocated by this method.

Implemented using GlobalAlloc with a flag parameter of **GMEM_FIXED**.

See Also

Reference

CGlobalHeap Class
CGlobalHeap::Free
CGlobalHeap::Reallocate

Other Resources

CGlobalHeap Members

CGlobalHeap::Free

Call this method to free a block of memory allocated by this memory manager.

```
virtual void Free(
   void* p
) throw( );
```

Parameters

p

Pointer to memory previously allocated by this memory manager. NULL is a valid value and does nothing.

Remarks

Implemented using GlobalFree.

See Also

Reference

CGlobalHeap Class

CGlobalHeap::Allocate

Other Resources

CGlobalHeap Members

CGlobalHeap::GetSize

Call this method to get the allocated size of a memory block allocated by this memory manager.

```
virtual size_t GetSize(
   void* p
) throw( );
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

Return Value

Returns the size of the allocated memory block in bytes.

Remarks

Implemented using GlobalSize.

See Also

Reference

CGlobalHeap Class

Other Resources

CGlobalHeap Members

CGlobalHeap::Reallocate

Call this method to reallocate memory allocated by this memory manager.

```
virtual void* Reallocate(
   void* p,
   size_t nBytes
) throw( );
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call CGlobalHeap::Free to free the memory allocated by this method.

Implemented using GlobalReAlloc.

See Also

Reference

CGlobalHeap Class

CGlobalHeap::Allocate

Other Resources

CGlobalHeap Members

CHandle Class

This class provides methods for creating and using a handle object.

```
class CHandle
```

Remarks

A **CHandle** object can be used whenever a handle is required: the main difference is that the **CHandle** object will automatically be deleted.

Note
Some API functions will use NULL as an empty or invalid handle, while others use INVALID_HANDLE_VALUE. **CHandle** only uses NULL and will treat INVALID_HANDLE_VALUE as a real handle. If you call an API which can return INVALID_HANDLE_VALUE, you should check for this value before calling CHandle::Attach or passing it to the **CHandle** constructor, and instead pass NULL.

Requirements

Header: atlbase.h

See Also

Other Resources

CHandle Members

ATL Class Overview

CHandle Members

Methods

Attach	Call this method to attach the **CHandle** object to an existing handle.
CHandle	The constructor.
~CHandle	The destructor.
Close	Call this method to close a **CHandle** object.
Detach	Call this method to detach a handle from a **CHandle** object.

Operators

operator HANDLE	Returns the value of the stored handle.
operator =	Assignment operator.

Data Members

m_h	The member variable that stores the handle.

See Also

Reference

CHandle Class

CHandle Methods

For information about the methods in **CHandle**, see CHandle Members.

CHandle::Attach

Call this method to attach the **CHandle** object to an existing handle.

```
void Attach(
   HANDLE h
) throw( );
```

Parameters

h

CHandle will take ownership of the handle *h*.

Remarks

Assigns the **CHandle** object to the *h* handle. In debugs builds, an ATLASSERT will be raised if *h* is NULL. No other check as to the validity of the handle is made.

See Also

Reference

CHandle Class

CHandle::Detach

Other Resources

CHandle Members

CHandle::CHandle

The constructor.

```
CHandle( ) throw( );
CHandle(
   CHandle& h
) throw( );
explicit CHandle(
   HANDLE h
) throw( );
```

Parameters

h

An existing handle or **CHandle**.

Remarks

Creates a new **CHandle** object, optionally using an existing handle or **CHandle** object.

See Also

Reference

CHandle Class

CHandle::Attach

Other Resources

CHandle Members

CHandle::~CHandle

The destructor.

```
~CHandle( ) throw( );
```

Remarks

Frees the **CHandle** object by calling CHandle::Close.

See Also

Reference

CHandle Class

CHandle::CHandle

Other Resources

CHandle Members

CHandle::Close

Call this method to close a **CHandle** object.

```
void Close( ) throw( );
```

Remarks

Closes an open object handle. If the handle is NULL, which will be the case if **Close** has already been called, an ATLASSERT will be raised in debug builds.

See Also

Reference

CHandle Class

CHandle::~CHandle

Other Resources

CHandle Members

CHandle::Detach

Call this method to detach a handle from a **CHandle** object.

```
HANDLE Detach( ) throw( );
```

Return Value

Returns the handle being detached.

Remarks

Releases ownership of the handle.

See Also

Reference

CHandle Class

CHandle::Attach

Other Resources

CHandle Members

CHandle Operators

For information about the operators in **CHandle**, see CHandle Members.

CHandle::operator =

The assignment operator.

```
CHandle& operator =(
   CHandle& h
) throw( );
```

Parameters

h

CHandle will take ownership of the handle *h*.

Return Value

Returns a reference to the new **CHandle** object.

Remarks

If the **CHandle** object currently contains a handle, it will be closed. The **CHandle** object being passed in will have its handle reference set to NULL. This ensures that two **CHandle** objects will never contain the same active handle.

See Also

Reference

CHandle Class

CHandle::Attach

Other Resources

CHandle Members

CHandle::operator HANDLE

Returns the value of the stored handle.

```
operator HANDLE( ) const throw( );
```

Remarks

Returns the value stored in CHandle::m_h.

See Also

Reference

CHandle Class

Other Resources

CHandle Members

CHandle Data Members

For information about the data members in **CHandle**, see CHandle Members.

CHandle::m_h

The member variable that stores the handle.

```
HANDLE m_h;
```

See Also

Reference

CHandle Class

Other Resources

CHandle Members

CHeapPtr Class

A smart pointer class for managing heap pointers.

```
template<
   typename T,
   class Allocator = CCRTAllocator
> class CHeapPtr :
   public CHeapPtrBase< T, Allocator >
```

Parameters

T

The object type to be stored on the heap.

Allocator

The memory allocation class to use.

Remarks

CHeapPtr is derived from CHeapPtrBase and by default uses the CRT routines (in CCRTAllocator) to allocate and free memory. The class CHeapPtrList may be used to construct a list of heap pointers. See also CComHeapPtr, which uses COM memory allocation routines.

Requirements

Header: atlcore.h

Example

See the ISAPIFilter Sample.

See Also

Reference

CHeapPtrBase Class

CCRTAllocator Class

Other Resources

CHeapPtr Members

ATL Class Overview

CHeapPtr Members

Methods

Allocate	Call this method to allocate memory on the heap to store objects.
CHeapPtr	The constructor.
Reallocate	Call this method to reallocate the memory on the heap.

Operators

operator =	The assignment operator.

See Also

Reference

CHeapPtr Class

Other Resources

CHeapPtrBase Members

CHeapPtr Methods

For information about the methods in **CHeapPtr**, see CHeapPtr Members.

CHeapPtr::Allocate

Call this method to allocate memory on the heap to store objects.

```
bool Allocate(
   size_t nElements = 1
) throw( );
```

Parameters

nElements

The number of elements used to calculate the amount of memory to allocate. The default value is 1.

Return Value

Returns true if the memory was successfully allocated, false on failure.

Remarks

The allocator routines are used to reserve enough memory on the heap to store *nElement* objects of a type defined in the constructor.

Example

```
// Create a new CHeapPtr object
CHeapPtr <int> myHP;
// Allocate space for 10 integers on the heap
myHP.Allocate(10);
```

For another example, see the ISAPIFilter Sample.

See Also

Reference

CHeapPtr Class

CHeapPtr::Reallocate

Other Resources

CHeapPtr Members

CHeapPtr::CHeapPtr

The constructor.

```
CHeapPtr( ) throw( );
explicit CHeapPtr(
   T* p
) throw( );
CHeapPtr(
   CHeapPtr< T, Allocator >& p
) throw( );
```

Parameters

p

An existing heap pointer or **CHeapPtr**.

Remarks

The heap pointer can optionally be created using an existing pointer, or a **CHeapPtr** object. If so, the new **CHeapPtr** object assumes responsibility for managing the new pointer and resources.

Example

```
// Create a new CHeapPtr object
CHeapPtr <int> myHP;
// Create a new CHeapPtr from the first
CHeapPtr <int> myHP2 (myHP);
```

See Also

Reference

CHeapPtr Class

Other Resources

CHeapPtr Members

CHeapPtr::Reallocate

Call this method to reallocate the memory on the heap.

```
bool Reallocate(
   size_t nElements
) throw( );
```

Parameters

nElements

The new number of elements used to calculate the amount of memory to allocate.

Return Value

Returns true if the memory was successfully allocated, false on failure.

Example

```
// Create a new CHeapPtr object
CHeapPtr <int> myHP;
// Allocate space for 10 integers on the heap
myHP.Allocate(10);
// Resize the allocated memory for 20 integers
myHP.Reallocate(20);
```

See Also

Reference

CHeapPtr Class

CHeapPtr::Allocate

Other Resources

CHeapPtr Members

CHeapPtr Operators

For information about the operators in **CHeapPtr**, see CHeapPtr Members.

CHeapPtr::operator =

Assignment operator.

```
CHeapPtr< T, Allocator >& operator =(
   CHeapPtr< T, Allocator >& p
) throw( );
```

Parameters

p

An existing **CHeapPtr** object.

Return Value

Returns a reference to the updated **CHeapPtr**.

Example

```
// Create a new CHeapPtr object
CHeapPtr <int> myHP;
// Allocate space for 10 integers on the heap
myHP.Allocate(10);
// Create a second heap pointer
// and assign it to the first pointer.
CHeapPtr <int> myHP2;
myHP2 = myHP;
```

See Also

Reference

CHeapPtr Class

Other Resources

CHeapPtr Members

CHeapPtrBase Class

This class forms the basis for several smart heap pointer classes.

```
template <
   class T,
   class Allocator = CCRTAllocator
> class CHeapPtrBase
```

Parameters

T

The object type to be stored on the heap.

Allocator

The memory allocation class to use. By default CRT routines are used to allocate and free memory.

Remarks

This class forms the basis for several smart heap pointer classes. The derived classes, for example, CHeapPtr and CComHeapPtr, add their own constructors and operators. See these classes for implementation examples.

Requirements

Header: atlcore.h

See Also

Reference

CHeapPtr Class

CComHeapPtr Class

Other Resources

CHeapPtrBase Members

ATL Class Overview

CHeapPtrBase Members

Methods

AllocateBytes	Call this method to allocate memory.
Attach	Call this method to take ownership of an existing pointer.
~CHeapPtrBase	The destructor.
Detach	Call this method to release ownership of a pointer.
Free	Call this method to delete an object pointed to by a **CHeapPtrBase**.
ReallocateBytes	Call this method to reallocate memory.

Operators

operator &	The & operator.
operator T*	The cast operator.
operator ->	The pointer-to-member operator.

Data Members

m_pData	The pointer data member variable.

See Also

Reference

CHeapPtrBase Class

CHeapPtrBase Methods

For information about the methods in **CHeapPtrBase**, see CHeapPtrBase Members.

CHeapPtrBase::AllocateBytes

Call this method to allocate memory.

```
bool AllocateBytes(
   size_t nBytes
) throw( );
```

Parameters

nBytes

The number of bytes of memory to allocate.

Return Value

Returns true if the memory is successfully allocated, false otherwise.

Remarks

In debug builds, an assertion failure will occur if the CHeapPtrBase::m_pData member variable currently points to an existing value; that is, it is not equal to NULL.

See Also

Reference

CHeapPtrBase Class
CHeapPtrBase::Free
CHeapPtrBase::ReallocateBytes

Other Resources

CHeapPtrBase Members

CHeapPtrBase::Attach

Call this method to take ownership of an existing pointer.

```
void Attach(
   T* pData
) throw( );
```

Parameters

pData

The **CHeapPtrBase** object will take ownership of this pointer.

Remarks

When a **CHeapPtrBase** object takes ownership of a pointer, it will automatically delete the pointer and any allocated data when it goes out of scope.

In debug builds, an assertion failure will occur if the CHeapPtrBase::m_pData member variable currently points to an existing value; that is, it is not equal to NULL.

See Also

Reference

CHeapPtrBase Class

CHeapPtrBase::Detach

Other Resources

CHeapPtrBase Members

CHeapPtrBase::~CHeapPtrBase

The destructor.

```
~CHeapPtrBase( ) throw( );
```

Remarks

Frees all allocated resources.

See Also

Reference

CHeapPtrBase Class

Other Resources

CHeapPtrBase Members

CHeapPtrBase::Detach

Call this method to release ownership of a pointer.

```
T* Detach( ) throw( );
```

Return Value

Returns a copy of the pointer.

Remarks

Releases ownership of a pointer, sets the CHeapPtrBase::m_pData member variable to NULL, and returns a copy of the pointer.

See Also

Reference

CHeapPtrBase Class

CHeapPtrBase::Attach

Other Resources

CHeapPtrBase Members

CHeapPtrBase::Free

Call this method to delete an object pointed to by a **CHeapPtrBase**.

```
void Free( ) throw( );
```

Remarks

The object pointed to by the **CHeapPtrBase** is freed, and the CHeapPtrBase::m_pData member variable is set to NULL.

See Also

Reference

CHeapPtrBase Class
CHeapPtrBase::AllocateBytes
CHeapPtrBase::ReallocateBytes

Other Resources

CHeapPtrBase Members

CHeapPtrBase::ReallocateBytes

Call this method to reallocate memory.

```
bool ReallocateBytes(
   size_t nBytes
) throw( );
```

Parameters

nBytes

The new amount of memory to allocate, in bytes.

Return Value

Returns true if the memory is successfully allocated, false otherwise.

See Also

Reference

CHeapPtrBase Class
CHeapPtrBase::AllocateBytes
CHeapPtrBase::Free

Other Resources

CHeapPtrBase Members

CHeapPtrBase Operators

For information about the operators in **CHeapPtrBase**, see CHeapPtrBase Members.

CHeapPtrBase::operator &

The & operator.

```
T** operator&( ) throw( );
```

Return Value

Returns the address of the object pointed to by the **CHeapPtrBase** object.

See Also

Reference

CHeapPtrBase Class

Other Resources

CHeapPtrBase Members

CHeapPtrBase::operator ->

The pointer-to-member operator.

```
T* operator ->( ) const throw( );
```

Return Value

Returns the value of the CHeapPtrBase::m_pData member variable.

Remarks

Use this operator to call a method in a class pointed to by the **CHeapPtrBase** object. In debug builds, an assertion failure will occur if the **CHeapPtrBase** points to NULL.

See Also

Reference

CHeapPtrBase Class

Other Resources

CHeapPtrBase Members

CHeapPtrBase::operator T*

The cast operator.

```
operator T*( ) const throw( );
```

Remarks

Returns CHeapPtrBase::m_pData.

See Also

Reference

CHeapPtrBase Class

Other Resources

CHeapPtrBase Members

CHeapPtrBase Data Members

For information about the data members in **CHeapPtrBase**, see CHeapPtrBase Members.

CHeapPtrBase::m_pData

The pointer data member variable.

```
T* m_pData;
```

Remarks

This member variable holds the pointer information.

See Also

Reference

CHeapPtrBase Class

Other Resources

CHeapPtrBase Members

CHeapPtrElementTraits Class

This class provides methods, static functions, and typedefs useful when creating collections of heap pointers.

```
template<
   typename T,
   class Allocator = ATL::CCRTAllocator
>
class CHeapPtrElementTraits : public CDefaultElementTraits<
   ATL::CHeapPtr< T, Allocator >
>
```

Parameters

T

The object type to be stored in the collection class.

Allocator

The memory allocation class to use. The default is CCRTAllocator.

Remarks

This class provides methods, static functions, and typedefs for aiding the creation of collection class objects containing heap pointers. The class **CHeapPtrList** derives from **CHeapPtrElementTraits**.

For more information, see ATL Collection Classes.

Requirements

Header: atlcoll.h

See Also

Reference

CDefaultElementTraits Class

CComHeapPtr Class

Other Resources

CHeapPtrElementTraits Members

ATL Class Overview

CHeapPtrElementTraits Members

Typedefs

INARGTYPE	The data type to use for adding elements to the collection class object.
OUTARGTYPE	The data type to use for retrieving elements from the collection class object.

See Also

Reference

CHeapPtrElementTraits Class

CHeapPtrElementTraits Typedefs

For information about the typedefs in **CHeapPtrElementTraits**, see CHeapPtrElementTraits Members.

CHeapPtrElementTraits::INARGTYPE

The data type to use for adding elements to the collection class object.

```
typedef CHeapPtr<T, Allocator> & INARGTYPE;
```

See Also

Reference

CHeapPtrElementTraits Class

CHeapPtrElementTraits::OUTARGTYPE

Other Resources

CHeapPtrElementTraits Members

CHeapPtrElementTraits::OUTARGTYPE

The data type to use for retrieving elements from the collection class object.

```
typedef T *& OUTARGTYPE;
```

See Also

Reference

CHeapPtrElementTraits Class

CHeapPtrElementTraits::INARGTYPE

Other Resources

CHeapPtrElementTraits Members

CHeapPtrList Class

This class provides methods useful when constructing a list of heap pointers.

```
template<
   typename E,
   class Allocator = ATL::CCRTAllocator
>
class CHeapPtrList : public CAtlList<
   ATL::CHeapPtr< E, Allocator >,
   CHeapPtrElementTraits< E, Allocator >
>
```

Parameters

E

The object type to be stored in the collection class.

Allocator

The memory allocation class to use. The default is CCRTAllocator.

Remarks

This class provides a constructor and derives methods from CAtlList and CHeapPtrElementTraits to aid the creation of a collection class object storing heap pointers.

Requirements

Header: atlcoll.h

See Also

Reference

CAtlList Class
CHeapPtr Class
CHeapPtrElementTraits Class

Other Resources

CHeapPtrList Members
ATL Class Overview

CHeapPtrList Members

Methods

CHeapPtrList	The constructor.

See Also

Reference

CHeapPtrList Class

Other Resources

CAtlList Members

CHeapPtrElementTraits Members

CHeapPtrList Methods

For information about the methods in **CHeapPtrList**, see CHeapPtrList Members.

CHeapPtrList::CHeapPtrList

The constructor.

```
CHeapPtrList(
   UINT nBlockSize = 10
) throw( );
```

Parameters

nBlockSize

The block size.

Remarks

The block size is a measure of the amount of memory allocated when a new element is required. Larger block sizes reduce calls to memory allocation routines, but use more resources.

See Also

Reference

CHeapPtrList Class

Other Resources

CHeapPtrList Members

CInterfaceArray Class

This class provides methods useful when constructing an array of COM interface pointers.

```
template<
   class I,
   const IID* piid = & __uuidof( I )
>
class CInterfaceArray : public CAtlArray<
   ATL::CComQIPtr< I, piid >,
   CComQIPtrElementTraits< I, piid >
>
```

Parameters

I

A COM interface specifying the type of pointer to be stored.

piid

A pointer to the IID of *I*.

Remarks

This class provides a constructor and derived methods for creating an array of COM interface pointers. Use CInterfaceList when a list is required.

For more information, see ATL Collection Classes.

Requirements

Header: atlcoll.h

See Also

Reference

CAtlArray Class
CComQIPtr Class
CComQIPtrElementTraits Class

Other Resources

CInterfaceArray Members
ATL Class Overview

CInterfaceArray Members

Methods

CInterfaceArray	The constructor for the interface array.

See Also

Reference

CInterfaceArray Class

Other Resources

CAtlArray Members

CInterfaceArray Methods

For information about the methods in **CInterfaceArray**, see CInterfaceArray Members.

CInterfaceArray::CInterfaceArray

The constructor.

```
CInterfaceArray( ) throw( );
```

Remarks

Initializes the smart pointer array.

See Also

Reference

CInterfaceArray Class

Other Resources

CInterfaceArray Members

CInterfaceList Class

This class provides methods useful when constructing a list of COM interface pointers.

```
template<
   class I,
   const IID* piid = & __uuidof( I )
>
class CInterfaceList : public CAtlList<
   ATL::CComQIPtr< I, piid >,
   CComQIPtrElementTraits< I, piid >
>
```

Parameters

I

A COM interface specifying the type of pointer to be stored.

piid

A pointer to the IID of *I*.

Remarks

This class provides a constructor and derived methods for creating a list of COM interface pointers. Use CInterfaceArray when an array is required.

For more information, see ATL Collection Classes.

Requirements

Header: atlcoll.h

See Also

Reference

CAtlList Class
CComQIPtr Class
CComQIPtrElementTraits Class

Other Resources

CInterfaceList Members
ATL Class Overview

CInterfaceList Members

Methods

CInterfaceList	The constructor for the interface list.

See Also

Reference

CInterfaceList Class

Other Resources

CAtlArray Members

CInterfaceList Methods

For information about the methods in **CInterfaceList**, see CInterfaceList Members.

CInterfaceList::CInterfaceList

The constructor for the interface list.

```
CInterfaceList(
   UINT nBlockSize = 10
) throw( );
```

Parameters

nBlockSize

The block size, with a default of 10.

Remarks

The block size is a measure of the amount of memory allocated when a new element is required. Larger block sizes reduce calls to memory allocation routines, but use more resources.

See Also

Reference

CInterfaceList Class

Other Resources

CInterfaceList Members

CLocalHeap Class

This class implements IAtlMemMgr using the Win32 local heap functions.

```
class CLocalHeap : public IAtlMemMgr
```

Remarks

CLocalHeap implements memory allocation functions using the Win32 local heap functions.

Note
The local heap functions are slower than other memory management functions and do not provide as many features. Therefore, new applications should use the heap functions. These are available in the CWin32Heap class.

Example

See the example for IAtlMemMgr.

Requirements

Header: atlmem.h

See Also

Reference

CComHeap Class
CWin32Heap Class
CGlobalHeap Class
CCRTHeap Class
IAtlMemMgr Class

Other Resources

CLocalHeap Members
ATL Class Overview

CLocalHeap Members

Methods

Allocate	Call this method to allocate a block of memory.
Free	Call this method to free a block of memory allocated by this memory manager.
GetSize	Call this method to get the allocated size of a memory block allocated by this memory manager.
Reallocate	Call this method to reallocate memory allocated by this memory manager.

See Also

Reference

CLocalHeap Class

Other Resources

IAtlMemMgr Members

CLocalHeap Methods

For information about the methods in **CLocalHeap**, see CLocalHeap Members.

CLocalHeap::Allocate

Call this method to allocate a block of memory.

```
virtual void* Allocate(
   size_t nBytes
) throw( );
```

Parameters

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call CLocalHeap::Free or CLocalHeap::Reallocate to free the memory allocated by this method.

Implemented using LocalAlloc with a flag parameter of **LMEM_FIXED**.

See Also

Reference

CLocalHeap Class
CLocalHeap::Free
CLocalHeap::Reallocate

Other Resources

CLocalHeap Members

CLocalHeap::Free

Call this method to free a block of memory allocated by this memory manager.

```
virtual void Free(
   void* p
) throw( );
```

Parameters

p

Pointer to memory previously allocated by this memory manager. NULL is a valid value and does nothing.

Remarks

Implemented using LocalFree.

See Also

Reference

CLocalHeap Class

CLocalHeap::Allocate

Other Resources

CLocalHeap Members

CLocalHeap::GetSize

Call this method to get the allocated size of a memory block allocated by this memory manager.

```
virtual size_t GetSize(
   void* p
) throw( );
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

Return Value

Returns the size of the allocated memory block in bytes.

Remarks

Implemented using LocalSize.

See Also

Reference

CLocalHeap Class

Other Resources

CLocalHeap Members

CLocalHeap::Reallocate

Call this method to reallocate memory allocated by this memory manager.

```
virtual void* Reallocate(
   void* p,
   size_t nBytes
) throw( );
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call CLocalHeap::Free to free the memory allocated by this method.

Implemented using LocalReAlloc.

See Also

Reference

CLocalHeap Class
CLocalHeap::Free
CLocalHeap::Allocate

Other Resources

CLocalHeap Members

CMessageMap Class

This class allows an object's message maps to be access by another object.

```
class ATL_NO_VTABLE CMessageMap
```

Remarks

CMessageMap is an abstract base class that allows an object's message maps to be accessed by another object. In order for an object to expose its message maps, its class must derive from **CMessageMap**.

ATL uses **CMessageMap** to support contained windows and dynamic message map chaining. For example, any class containing a CContainedWindow object must derive from **CMessageMap**. The following code is taken from the SUBEDIT sample. Through CComControl, the `CAtlEdit` class automatically derives from **CMessageMap**.

```
class CAtlEdit : public CComControl<CAtlEdit>, ...
               // CComControl derives from CWindowImpl,
               // which derives from CMessageMap
{
public:
   // Declare a contained window data member
   CContainedWindow m_EditCtrl;

   // Initialize the contained window:
   // 1. Pass "EDIT" to specify that the contained
   //    window should be based on the standard
   //    Windows Edit box
   // 2. Pass 'this' pointer to specify that CAtlEdit
   //    contains the message map to be used for the
   //    contained window's message processing
   // 3. Pass the identifier of the message map. In
   //    this case, '1' identifies the message map
   //    declared with ALT_MSG_MAP(1)
   CAtlEdit() : m_EditCtrl(_T("EDIT"), this, 1)
   {
      m_bWindowOnly = TRUE;
   }

   // Declare the default message map
   BEGIN_MSG_MAP(CAtlEdit)
      MESSAGE_HANDLER(WM_PAINT, OnPaint)
      ...
   // Declare an alternate message map,
   // identified by '1'
   ALT_MSG_MAP(1)
      MESSAGE_HANDLER(WM_CHAR, OnChar)
   END_MSG_MAP()

   ...
};
```

Because the contained window, `m_EditCtrl`, will use a message map in the containing class, `CAtlEdit` derives from **CMessageMap**.

For more information about message maps, see Message Maps in the article "ATL Window Classes."

Requirements

Header: atlwin.h

See Also

Reference

CDynamicChain Class

BEGIN_MSG_MAP

ALT_MSG_MAP

Other Resources

CMessageMap Members

ATL Class Overview

CMessageMap Members

Methods

ProcessWindowMessage	Accesses a message map in the **CMessageMap**-derived class.

See Also

Reference

CMessageMap Class

CMessageMap Methods

For information about the methods in **CMessageMap**, see CMessageMap Members.

CMessageMap::ProcessWindowMessage

Accesses the message map identified by *dwMsgMapID* in a **CMessageMap**-derived class.

```
virtual BOOL ProcessWindowMessage(
   HWND hWnd,
   UINT uMsg,
   WPARAM wParam,
   LPARAM lParam,
   LRESULT& lResult,
   DWORD dwMsgMapID
) = 0;
```

Parameters

hWnd

[in] The handle to the window receiving the message.

uMsg

[in] The message sent to the window.

wParam

[in] Additional message-specific information.

lParam

[in] Additional message-specific information.

lResult

[out] The result of the message processing.

dwMsgMapID

[in] The identifier of the message map that will process the message. The default message map, declared with BEGIN_MSG_MAP, is identified by 0. An alternate message map, declared with ALT_MSG_MAP(msgMapID), is identified by *msgMapID*.

Return Value

TRUE if the message is fully handled; otherwise, **FALSE**.

Remarks

Called by the window procedure of a CContainedWindow object or of an object that is dynamically chaining to the message map.

See Also

Reference

CMessageMap Class
CHAIN_MSG_MAP_DYNAMIC

Other Resources

CMessageMap Members

CPrimitiveElementTraits Class

This class provides default methods and functions for a collection class composed of primitive data types.

```
template<
   typename T
> class CPrimitiveElementTraits :
   public CDefaultElementTraits< T >
```

Parameters

T

The type of data to be stored in the collection class object.

Remarks

This class provides default static functions and methods for moving, copying, comparing, and hashing primitive data type elements stored in a collection class object.

For more information, see ATL Collection Classes.

Requirements

Header: atlcoll.h

See Also

Reference

CDefaultElementTraits Class

Other Resources

CPrimitiveElementTraits Members

ATL Class Overview

CPrimitiveElementTraits Members

Typedefs

INARGTYPE	The data type to use for adding elements to the collection class object.
OUTARGTYPE	The data type to use for retrieving elements from the collection class object.

See Also

Reference

CPrimitiveElementTraits Class

CPrimitiveElementTraits Typedefs

For information about the typedefs in **CPrimitiveElementTraits**, see CPrimitiveElementTraits Members.

CPrimitiveElementTraits::INARGTYPE

The data type to use for adding elements to the collection class object.

```
typedef T INARGTYPE;
```

See Also

Reference

CPrimitiveElementTraits Class

CPrimitiveElementTraits::OUTARGTYPE

Other Resources

CPrimitiveElementTraits Members

CPrimitiveElementTraits::OUTARGTYPE

The data type to use for retrieving elements from the collection class object.

```
typedef T & OUTARGTYPE;
```

See Also

Reference

CPrimitiveElementTraits Class

CPrimitiveElementTraits::INARGTYPE

Other Resources

CPrimitiveElementTraits Members

CPrivateObjectSecurityDesc Class

This class represents a private object security descriptor object.

```
class CPrivateObjectSecurityDesc : public CSecurityDesc
```

Remarks

This class, derived from CSecurityDesc, provides methods for creating and managing the security descriptor of a private object.

For an introduction to the access control model in Windows, see Access Control in the *Platform SDK*.

Requirements

Header: atlsecurity.h

See Also

Reference

SECURITY_DESCRIPTOR

CSecurityDesc Class

Other Resources

CPrivateObjectSecurityDesc Members

ATL Class Overview

Security Global Functions

CPrivateObjectSecurityDesc Members

Methods

ConvertToAutoInherit	Call this method to convert a security descriptor and its access-control lists (ACLs) to a format th at supports automatic propagation of inheritable access-control entries (ACEs).
CPrivateObjectSecurityDesc	The constructor.
~CPrivateObjectSecurityDesc	The destructor.
Create	Call this method to allocate and initialize a self-relative security descriptor for the private object created by the calling resource manager.
Get	Call this method to retrieve information from a private object's security descriptor.
Set	Call this method to modify a private object's security descriptor.

Operators

operator =	Assignment operator.

See Also

Reference

CPrivateObjectSecurityDesc Class

CPrivateObjectSecurityDesc Methods

For information about the methods in **CPrivateObjectSecurityDesc**, see CPrivateObjectSecurityDesc Members.

CPrivateObjectSecurityDesc::ConvertToAutoInherit

Call this method to convert a security descriptor and its access-control lists (ACLs) to a format that supports automatic propagation of inheritable access-control entries (ACEs).

```
bool ConvertToAutoInherit(
   const CSecurityDesc* pParent,
   GUID* ObjectType,
   bool bIsDirectoryObject,
   PGENERIC_MAPPING GenericMapping
) throw( );
```

Parameters

pParent

Pointer to a CSecurityDesc object referencing the parent container of the object. If there is no parent container, this parameter is NULL.

ObjectType

Pointer to a **GUID** structure that identifies the type of object associated with the current object. Set *ObjectType* to NULL if the object does not have a GUID.

bIsDirectoryObject

Specifies whether the new object can contain other objects. A value of true indicates that the new object is a container. A value of false indicates that the new object is not a container.

GenericMapping

Pointer to a GENERIC_MAPPING structure that specifies the mapping from each generic right to specific rights for the object.

Return Value

Returns true on success, false on failure.

Remarks

This method attempts to determine whether the ACEs in the discretionary access-control list (DACL) and system access-control list (SACL) of the current security descriptor were inherited from the parent security descriptor. It calls the ConvertToAutoInheritPrivateObjectSecurity function.

See Also

Reference

CPrivateObjectSecurityDesc Class

Other Resources

CPrivateObjectSecurityDesc Members

CPrivateObjectSecurityDesc::CPrivateObjectSecurityDesc

The constructor.

```
CPrivateObjectSecurityDesc( ) throw( );
```

Remarks

Initializes the **CPrivateObjectSecurityDesc** object.

See Also

Reference

CPrivateObjectSecurityDesc Class

CPrivateObjectSecurityDesc::~CPrivateObjectSecurityDesc

Other Resources

CPrivateObjectSecurityDesc Members

CPrivateObjectSecurityDesc::~CPrivateObjectSecurityDesc

The destructor.

```
~CPrivateObjectSecurityDesc( ) throw( );
```

Remarks

The destructor frees all allocated resources and deletes the private object's security descriptor.

See Also

Reference

CPrivateObjectSecurityDesc Class

CPrivateObjectSecurityDesc::CPrivateObjectSecurityDesc

Other Resources

CPrivateObjectSecurityDesc Members

CPrivateObjectSecurityDesc::Create

Call this method to allocate and initialize a self-relative security descriptor for the private object created by the calling resource manager.

```
bool Create(
   const CSecurityDesc* pParent,
   const CSecurityDesc* pCreator,
   bool bIsDirectoryObject,
   const CAccessToken& Token,
   PGENERIC_MAPPING GenericMapping
) throw( );
bool Create(
   const CSecurityDesc* pParent,
   const CSecurityDesc* pCreator,
   GUID* ObjectType,
   bool bIsContainerObject,
   ULONG AutoInheritFlags,
   const CAccessToken& Token,
   PGENERIC_MAPPING GenericMapping
) throw( );
```

Parameters

pParent

Pointer to a CSecurityDesc object referencing the parent directory in which a new object is being created. Set to NULL if there is no parent directory.

pCreator

Pointer to a security descriptor provided by the creator of the object. If the object's creator does not explicitly pass security information for the new object, set this parameter to NULL.

bIsDirectoryObject

Specifies whether the new object can contain other objects. A value of true indicates that the new object is a container. A value of false indicates that the new object is not a container.

Token

Reference to the CAccessToken object for the client process on whose behalf the object is being created.

GenericMapping

Pointer to a GENERIC_MAPPING structure that specifies the mapping from each generic right to specific rights for the object.

ObjectType

Pointer to a **GUID** structure that identifies the type of object associated with the current object. Set *ObjectType* to NULL if the object does not have a GUID.

bIsContainerObject

Specifies whether the new object can contain other objects. A value of true indicates that the new object is a container. A value of false indicates that the new object is not a container.

AutoInheritFlags

A set of bit flags that control how access-control entries (ACEs) are inherited from *pParent*. See CreatePrivateObjectSecurityEx for more details.

Return Value

Returns true on success, false on failure.

Remarks

This method calls CreatePrivateObjectSercurity or CreatePrivateObjectSecurityEx.

The second method, which permits specifying the object type GUID of the new object or controlling how ACEs are inherited, is only available on systems running Windows 2000 and later.

Note
A self-relative security descriptor is a security descriptor that stores all of its security information in a contiguous block of memory.

See Also

Reference

CPrivateObjectSecurityDesc Class

Other Resources

CPrivateObjectSecurityDesc Members

CPrivateObjectSecurityDesc::Get

Call this method to retrieve information from a private object's security descriptor.

```
bool Get(
   SECURITY_INFORMATION si,
   CSecurityDesc* pResult
) const throw( );
```

Parameters

si

A set of bit flags that indicate the parts of the security descriptor to retrieve. This value can be a combination of the SECURITY_INFORMATION bit flags.

pResult

Pointer to a CSecurityDesc object that receives a copy of the requested information from the specified security descriptor.

Return Value

Returns true on success, false on failure.

Remarks

The security descriptor is a structure and associated data that contains the security information for a securable object.

See Also

Reference

CPrivateObjectSecurityDesc Class
GetPrivateObjectSecurity
CPrivateObjectSecurityDesc::Set

Other Resources

CPrivateObjectSecurityDesc Members

CPrivateObjectSecurityDesc::Set

Call this method to modify a private object's security descriptor.

```
bool Set(
   SECURITY_INFORMATION si,
   const CSecurityDesc& Modification,
   PGENERIC_MAPPING GenericMapping,
   const CAccessToken& Token
) throw( );
bool Set(
   SECURITY_INFORMATION si,
   const CSecurityDesc& Modification,
   ULONG AutoInheritFlags,
   PGENERIC_MAPPING GenericMapping,
   const CAccessToken& Token
) throw( );
```

Parameters

si

A set of bit flags that indicate the parts of the security descriptor to set. This value can be a combination of the SECURITY_INFORMATION bit flags.

Modification

Pointer to a CSecurityDesc object. The parts of this security descriptor indicated by the *si* parameter are applied to the object's security descriptor.

GenericMapping

Pointer to a GENERIC_MAPPING structure that specifies the mapping from each generic right to specific rights for the object.

Token

Reference to the CAccessToken object for the client process on whose behalf the object is being created.

AutoInheritFlags

A set of bit flags that control how access-control entries (ACEs) are inherited from *pParent*. See CreatePrivateObjectSecurityEx for more details.

Return Value

Returns true on success, false on failure.

Remarks

The second method, which permits specifying the object type GUID of the object or controlling how ACEs are inherited, is only available on systems running Windows 2000 and later.

See Also

Reference

CPrivateObjectSecurityDesc Class
SetPrivateObjectSecurity
CPrivateObjectSecurityDesc::Get

Other Resources

CPrivateObjectSecurityDesc Members

CPrivateObjectSecurityDesc Operators

For information about the operators in **CPrivateObjectSecurityDesc**, see CPrivateObjectSecurityDesc Members.

CPrivateObjectSecurityDesc::operator =

Assignment operator.

```
CPrivateObjectSecurityDesc& operator =(
   const CPrivateObjectSecurityDesc& rhs
) throw(...);
```

Parameters

rhs

The **CPrivateObjectSecurityDesc** object to assign to the current object.

Return Value

Returns the updated **CPrivateObjectSecurityDesc** object.

See Also

Reference

CPrivateObjectSecurityDesc Class

Other Resources

CPrivateObjectSecurityDesc Members

CRBMap Class

This class represents a mapping structure, using a Red-Black binary tree.

```
template<
   typename K,
   typename V,
   class KTraits = CElementTraits< K >,
   class VTraits = CElementTraits< V >
> class CRBMap : public CRBTree< K, V, KTraits, VTraits >
```

Parameters

K

The key element type.

V

The value element type.

KTraits

The code used to copy or move key elements. See CElementTraits Class for more details.

VTraits

The code used to copy or move value elements.

Remarks

CRBMap provides support for a mapping array of any given type, managing an ordered array of key elements and their associated values. Each key can have only one associated value. Elements (consisting of a key and a value) are stored in a binary tree structure, using the CRBMap::SetAt method. Elements can be removed using the CRBMap::RemoveKey method, which deletes the element with the given key value.

Traversing the tree is made possible with methods such as CRBTree::GetHeadPosition, CRBTree::GetNext, and CRBTree::GetNextValue.

The *KTraits* and *VTraits* parameters are traits classes that contain any supplemental code needed to copy or move elements.

CRBMap is derived from CRBTree, which implements a binary tree using the Red-Black algorithm. CRBMultiMap is a variation that allows multiple values for each key. It too is derived from **CRBTree**, and so shares many features with **CRBMap**.

An alternative to both **CRBMap** and **CRBMultiMap** is offered by the CAtlMap class. When only a small number of elements needs to be stored, consider using the CSimpleMap class instead.

For a more complete discussion of the various collection classes and their features and performance characteristics, see ATL Collection Classes.

Requirements

Header: atlcoll.h

See Also

Reference

CRBTree Class
CAtlMap Class
CRBMultiMap Class

Other Resources

CRBMap Members
ATL Class Overview

CRBMap Members

Methods

CRBMap	The constructor.
~CRBMap	The destructor.
Lookup	Call this method to look up keys or values in the **CRBMap** object.
RemoveKey	Call this method to remove an element from the **CRBMap** object, given the key.
SetAt	Call this method to insert an element pair into the map.

See the documentation for the base class CRBTree for information on the other methods available.

See Also

Reference

CRBMap Class

Other Resources

CRBTree Members

CRBMap Methods

For information about the methods in **CRBMap**, see CRBMap Members.

CRBMap::CRBMap

The constructor.

```
explicit CRBMap(
   size_t nBlockSize = 10
) throw( );
```

Parameters

nBlockSize

The block size.

Remarks

The *nBlockSize* parameter is a measure of the amount of memory allocated when a new element is required. Larger block sizes reduce calls to memory allocation routines, but use more resources. The default will allocate space for 10 elements at a time.

See the documentation for the base class CRBTree for information on the other methods available.

Example

```
// Define a map object which has an
// integer key, a double value, and a
// block size of 5
CRBMap<int, double> myMap(5);
```

See Also

Reference

CRBMap Class

CRBMap::~CRBMap

Other Resources

CRBMap Members

CRBMap::~CRBMap

The destructor.

```
~CRBMap( ) throw( );
```

Remarks

Frees any allocated resources.

See the documentation for the base class CRBTree for information on the other methods available.

See Also

Reference

CRBMap Class

CRBMap::CRBMap

Other Resources

CRBMap Members

CRBMap::Lookup

Call this method to look up keys or values in the **CRBMap** object.

```
bool Lookup(
   KINARGTYPE key,
   VOUTARGTYPE value
) const throw(...);
const CPair* Lookup(
   KINARGTYPE key
) const throw( );
CPair* Lookup(
   KINARGTYPE key
) throw( );
```

Parameters

key

Specifies the key that identifies the element to be looked up.

value

Variable that receives the looked-up value.

Return Value

The first form of the method returns true if the key is found, otherwise false. The second and third forms return a pointer to a CPair.

Remarks

See the documentation for the base class CRBTree for information on the other methods available.

Example

```
// Look up the value for a key of 0
double v;
myMap.Lookup(0,v);
```

See Also

Reference

CRBMap Class

Other Resources

CRBMap Members

CRBMap::RemoveKey

Call this method to remove an element from the **CRBMap** object, given the key.

```
bool RemoveKey(
   KINARGTYPE key
) throw( );
```

Parameters

key

The key corresponding to the element pair you want to remove.

Return Value

Returns true if the key is found and removed, false on failure.

Remarks

See the documentation for the base class CRBTree for information on the other methods available.

Example

```
// Remove an element, based on the key of 0
ATLVERIFY(myMap.RemoveKey(0) == true);
```

See Also

Reference

CRBMap Class

CRBMap::SetAt

Other Resources

CRBMap Members

CRBMap::SetAt

Call this method to insert an element pair into the map.

```
POSITION SetAt(
   KINARGTYPE key,
   VINARGTYPE value
) throw(...);
```

Parameters

key

The key value to add to the **CRBMap** object.

value

The value to add to the **CRBMap** object.

Return Value

Returns the position of the key/value element pair in the **CRBMap** object.

Remarks

SetAt replaces an existing element if a matching key is found. If the key is not found, a new key/value pair is created.

See the documentation for the base class CRBTree for information on the other methods available.

Example

```
// Add an element to the map, with a key of 0
myMap.SetAt(0,1.1);
```

See Also

Reference

CRBMap Class

CRBMap::RemoveKey

Other Resources

CRBMap Members

CRBMultiMap Class

This class represents a mapping structure that allows each key can be associated with more than one value, using a Red-Black binary tree.

```
template<
   typename K,
   typename V,
   class KTraits = CElementTraits< K >,
   class VTraits = CElementTraits< V >
> class CRBMultiMap : public CRBTree< K, V, KTraits, VTraits >
```

Parameters

K

The key element type.

V

The value element type.

KTraits

The code used to copy or move key elements. See CElementTraits Class for more details.

VTraits

The code used to copy or move value elements.

Remarks

CRBMultiMap provides support for a mapping array of any given type, managing an ordered array of key elements and values. Unlike the CRBMap class, each key can be associated with more than one value.

Elements (consisting of a key and a value) are stored in a binary tree structure, using the CRBMultiMap::Insert method. Elements can be removed using the CRBMultiMap::RemoveKey method, which deletes all elements which match the given key.

Traversing the tree is made possible with methods such as CRBTree::GetHeadPosition, CRBTree::GetNext, and CRBTree::GetNextValue. Accessing the potentially multiple values per key is possible using the CRBMultiMap::FindFirstWithKey, CRBMultiMap::GetNextValueWithKey, and CRBMultiMap::GetNextWithKey methods. See the example for CRBMultiMap::CRBMultiMap for an illustration of this in practice.

The *KTraits* and *VTraits* parameters are traits classes that contain any supplemental code needed to copy or move elements.

CRBMultiMap is derived from CRBTree, which implements a binary tree using the Red-Black algorithm. An alternative to **CRBMultiMap** and **CRBMap** is offered by the CAtlMap class. When only a small number of elements needs to be stored, consider using the CSimpleMap class instead.

For a more complete discussion of the various collection classes and their features and performance characteristics, see ATL Collection Classes.

Requirements

Header: atlcoll.h

See Also

Reference

CRBTree Class
CAtlMap Class
CRBMap Class

Other Resources

CRBMultiMap Members
ATL Class Overview

CRBMultiMap Members

Methods

CRBMultiMap	The constructor.
~CRBMultiMap	The destructor.
FindFirstWithKey	Call this method to find the position of the first element with a given key.
GetNextValueWithKey	Call this method to get the value associated with a given key, and update the position value.
GetNextWithKey	Call this method to get the element associated with a given key, and update the position value.
Insert	Call this method to insert an element pair into the map.
RemoveKey	Call this method to remove all of the key/value elements for a given key.

See the documentation for the base class CRBTree for information on the other methods available.

See Also

Reference

CRBMultiMap Class

Other Resources

CRBTree Members

CRBMultiMap Methods

For information about the methods in **CRBMultiMap**, see CRBMultiMap Members.

CRBMultiMap::CRBMultiMap

The constructor.

```
explicit CRBMultiMap(
   size_t nBlockSize = 10
) throw( );
```

Parameters

nBlockSize

The block size.

Remarks

The *nBlockSize* parameter is a measure of the amount of memory allocated when a new element is required. Larger block sizes reduce calls to memory allocation routines, but use more resources. The default will allocate space for 10 elements at a time.

See the documentation for the base class CRBTree for information on the other methods available.

Example

```
// CRBMultiMap Example

// Define a multimap object which has an integer
// key, a double value, and a block size of 5
CRBMultiMap<int, double> myMap(5);

// Add some key/values. Notice how three
// different values are associated with
// one key. In a CRBMap object, the values
// would simply overwrite each other.
myMap.Insert(0,1.1);
myMap.Insert(0,1.2);
myMap.Insert(0,1.3);
myMap.Insert(1,2.1);

// Look up a key and iterate through
// all associated values

double v;
POSITION myPos = myMap.FindFirstWithKey(0);

while (myPos != NULL)
{
   v = myMap.GetNextValueWithKey(myPos,0);
   // As the loop iterates, v
   // contains the values 0.3, 0.2, 0.1
}

// Remove all of the values associated with that key
int i = myMap.RemoveKey(0);

// Confirm all three values were deleted
ATLASSERT(i == 3);
```

See Also

Reference

CRBMultiMap Class

CRBMultiMap::~CRBMultiMap

Other Resources

CRBMultiMap Members

CRBMultiMap::~CRBMultiMap

The destructor.

```
~CRBMultiMap( ) throw( );
```

Remarks

Frees any allocated resources.

See the documentation for the base class CRBTree for information on the other methods available.

See Also

Reference

CRBMultiMap Class

CRBMultiMap::CRBMultiMap

Other Resources

CRBMultiMap Members

CRBMultiMap::FindFirstWithKey

Call this method to find the position of the first element with a given key.

```
POSITION FindFirstWithKey(
   KINARGTYPE key
) const throw( );
```

Parameters

key

Specifies the key that identifies the element to be found.

Return Value

Returns the POSITION of the first key/value element if the key is found, NULL otherwise.

Remarks

A key in the **CRBMultiMap** can have one or more associated values. This method will provide the position value of the first value (which may, in fact, be the only value) associated with that particular key. The position value returned can then be used with CRBMultiMap::GetNextValueWithKey or CRBMultiMap::GetNextWithKey to obtain the value and update the position.

See the documentation for the base class CRBTree for information on the other methods available.

Example

See the example for CRBMultiMap::CRBMultiMap.

See Also

Reference

CRBMultiMap Class
CRBMultiMap::GetNextValueWithKey
CRBMultiMap::GetNextWithKey

Other Resources

CRBMultiMap Members

CRBMultiMap::GetNextValueWithKey

Call this method to get the value associated with a given key and update the position value.

```
const V& GetNextValueWithKey(
   POSITION& pos,
   KINARGTYPE key
) const throw( );
V& GetNextValueWithKey(
   POSITION& pos,
   KINARGTYPE key
) throw( );
```

Parameters

pos

The position value, obtained with either a call to CRBMultiMap::FindFirstWithKey or CRBMultiMap::GetNextWithKey, or a previous call to **GetNextValueWithKey**.

key

Specifies the key that identifies the element to be found.

Return Value

Returns the element pair associated with the given key.

Remarks

The position value is updated to point to the next value associated with the key. If no more values exist, the position value is set to NULL.

See the documentation for the base class CRBTree for information on the other methods available.

Example

See the example for CRBMultiMap::CRBMultiMap.

See Also

Reference

CRBMultiMap Class
CRBMultiMap::FindFirstWithKey
CRBMultiMap::GetNextWithKey

Other Resources

CRBMultiMap Members

CRBMultiMap::GetNextWithKey

Call this method to get the element associated with a given key and update the position value.

```
const CPair* GetNextWithKey(
   POSITION& pos,
   KINARGTYPE key
) const throw( );
CPair* GetNextWithKey(
   POSITION& pos,
   KINARGTYPE key
) throw( );
```

Parameters

pos

The position value, obtained with either a call to CRBMultiMap::FindFirstWithKey or CRBMultiMap::GetNextValueWithKey, or a previous call to **GetNextWithKey**.

key

Specifies the key that identifies the element to be found.

Return Value

Returns the next CRBTree::CPair Class element associated with the given key.

Remarks

The position value is updated to point to the next value associated with the key. If no more values exist, the position value is set to NULL.

See the documentation for the base class CRBTree for information on the other methods available.

See Also

Reference

CRBMultiMap Class
CRBMultiMap::FindFirstWithKey
CRBMultiMap::GetNextValueWithKey

Other Resources

CRBMultiMap Members

CRBMultiMap::Insert

Call this method to insert an element pair into the map.

```
POSITION Insert(
   KINARGTYPE key,
   VINARGTYPE value
) throw(...);
```

Parameters

key

The key value to add to the **CRBMultiMap** object.

value

The value to add to the **CRBMultiMap** object, associated with *key*.

Return Value

Returns the position of the key/value element pair in the **CRBMultiMap** object.

Remarks

See the documentation for the base class CRBTree for information on the other methods available.

Example

See the example for CRBMultiMap::CRBMultiMap.

See Also

Reference

CRBMultiMap Class

CRBMultiMap::RemoveKey

Other Resources

CRBMultiMap Members

CRBMultiMap::RemoveKey

Call this method to remove all of the key/value elements for a given key.

```
size_t RemoveKey(
   KINARGTYPE key
) throw( );
```

Parameters

key

Specifies the key that identifies the element(s) to be deleted.

Return Value

Returns the number of values associated with the given key.

Remarks

RemoveKey deletes all of the key/value elements that have a key that matches *key*.

See the documentation for the base class CRBTree for information on the other methods available.

Example

See the example for CRBMultiMap::CRBMultiMap.

See Also

Reference

CRBMultiMap Class

CRBMultiMap::Insert

Other Resources

CRBMultiMap Members

CRBTree Class

This class provides methods for creating and utilizing a Red-Black tree.

```
template<
   typename K,
   typename V,
   class KTraits = CElementTraits< K >,
   class VTraits = CElementTraits< V >
> class CRBTree
```

Parameters

K

The key element type.

V

The value element type.

KTraits

The code used to copy or move key elements. See CElementTraits Class for more details.

VTraits

The code used to copy or move value elements.

Remarks

A Red-Black tree is a binary search tree that uses an extra bit of information per node to ensure that it remains "balanced," that is, the tree height doesn't grow disproportionately large and affect performance.

This template class is designed to be used by CRBMap and CRBMultiMap. The bulk of the methods that make up these derived classes are provided by **CRBTree**.

For a more complete discussion of the various collection classes and their features and performance characteristics, see ATL Collection Classes.

Requirements

Header: atlcoll.h

See Also

Other Resources

CRBTree Members

ATL Class Overview

CRBTree Members

Methods

~CRBTree	The destructor.
FindFirstKeyAfter	Call this method to find the position of the element that uses the next available key.
GetAt	Call this method to get the element at a given position in the tree.
GetCount	Call this method to get the number of elements in the tree.
GetHeadPosition	Call this method to get the position value for the element at the head of the tree.
GetKeyAt	Call this method to get the key from a given position in the tree.
GetNext	Call this method to obtain a pointer to an element stored in the **CRBTree** object, and advance the position to the next element.
GetNextAssoc	Call this method to get the key and value of an element stored in the map and advance the position to the next element.
GetNextKey	Call this method to get the key of an element stored in the tree and advance the position to the next element.
GetNextValue	Call this method to get the value of an element stored in the tree and advance the position to the next element.
GetPrev	Call this method to obtain a pointer to an element stored in the **CRBTree** object, and then update the position to the previous element.
GetTailPosition	Call this method to get the position value for the element at the tail of the tree.
GetValueAt	Call this method to retrieve the value stored at a given position in the **CRBTree** object.
IsEmpty	Call this method to test for an empty tree object.
RemoveAll	Call this method to remove all elements from the **CRBTree** object.
RemoveAt	Call this method to remove the element at the given position in the **CRBTree** object.
SetValueAt	Call this method to change the value stored at a given position in the **CRBTree** object.

Typedefs

KINARGTYPE	Type used when a key is passed as an input argument.
KOUTARGTYPE	Type used when a key is returned as an output argument.
VINARGTYPE	Type used when a value is passed as an input argument.
VOUTARGTYPE	Type used when a value is passed as an output argument.

Classes

CPair	A class containing the key and value elements.

See Also

Reference

CRBTree Class

CRBTree Methods

For information about the methods in **CRBTree**, see CRBTree Members.

CRBTree::~CRBTree

The destructor.

```
~CRBTree( ) throw( );
```

Remarks

Frees any allocated resources. Calls CRBTree::RemoveAll to delete all elements.

See Also

Reference

CRBTree Class

Other Resources

CRBTree Members

CRBTree::FindFirstKeyAfter

Call this method to find the position of the element that uses the next available key.

```
POSITION FindFirstKeyAfter(
   KINARGTYPE key
) const throw( );
```

Parameters

key

A key value.

Return Value

Returns the position value of the element that uses the next available key. If there are no more elements, NULL is returned.

Remarks

This method makes it easy to traverse the tree without having to calculate position values beforehand.

See Also

Reference

CRBTree Class

Other Resources

CRBTree Members

CRBTree::GetAt

Call this method to get the element at a given position in the tree.

```
CPair* GetAt(
   POSITION pos
) throw( );
const CPair* GetAt(
   POSITION pos
) const throw( );
void GetAt(
   POSITION pos,
   KOUTARGTYPE key,
   VOUTARGTYPE value
) const;
```

Parameters

pos

The position value.

key

The variable that receives the key.

value

The variable that receives the value.

Return Value

The first two forms return a pointer to a CPair. The third form obtains a key and a value for the given position.

Remarks

The position value can be previously determined with a call to a method such as CRBTree::GetHeadPosition or CRBTree::GetTailPosition.

In debug builds, an assertion failure will occur if *pos* is equal to NULL.

See Also

Reference

CRBTree Class
CRBTree::GetValueAt
CRBTree::GetKeyAt
CRBTree::SetValueAt

Other Resources

CRBTree Members

CRBTree::GetCount

Call this method to get the number of elements in the tree.

```
size_t GetCount( ) const throw( );
```

Return Value

Returns the number of elements (each key/value pair is one element) stored in the tree.

See Also

Reference

CRBTree Class

CRBTree::IsEmpty

Other Resources

CRBTree Members

CRBTree::GetHeadPosition

Call this method to get the position value for the element at the head of the tree.

```
POSITION GetHeadPosition( ) const throw( );
```

Return Value

Returns the position value for the element at the head of the tree.

Remarks

The value returned by **GetHeadPosition** can be used with methods such as CRBTree::GetKeyAt or CRBTree::GetNext to traverse the tree and retrieve values.

See Also

Reference

CRBTree Class

CRBTree::GetTailPosition

Other Resources

CRBTree Members

CRBTree::GetKeyAt

Call this method to get the key from a given position in the tree.

```
const K& GetKeyAt(
   POSITION pos
) const throw( );
```

Parameters

pos

The position value.

Return Value

Returns the key stored at position *pos* in the tree.

Remarks

If *pos* is not a valid position value, results are unpredictable. In debug builds, an assertion failure will occur if *pos* is equal to NULL.

See Also

Reference

CRBTree Class
CRBTree::GetAt
CRBTree::SetValueAt
CRBTree::GetValueAt

Other Resources

CRBTree Members

CRBTree::GetNext

Call this method to obtain a pointer to an element stored in the **CRBTree** object, and advance the position to the next element.

```
const CPair* GetNext(
   POSITION& pos
) const throw( );
CPair* GetNext(
   POSITION& pos
) throw( );
```

Parameters

pos

The position counter, returned by a previous call to methods such as CRBTree::GetHeadPosition or CRBTree::FindFirstKeyAfter.

Return Value

Returns a pointer to the next CPair value in the tree.

Remarks

The *pos* position counter is updated after each call. If the retrieved element is the last in the tree, *pos* is set to NULL.

See Also

Reference

CRBTree Class

CRBTree::GetPrev

Other Resources

CRBTree Members

CRBTree::GetNextAssoc

Call this method to get the key and value of an element stored in the map and advance the position to the next element.

```
void GetNextAssoc(
   POSITION& pos,
   KOUTARGTYPE key,
   VOUTARGTYPE value
) const;
```

Parameters

pos

The position counter, returned by a previous call to methods such as CRBTree::GetHeadPosition or CRBTree::FindFirstKeyAfter.

key

Template parameter specifying the type of the tree's key.

value

Template parameter specifying the type of the tree's value.

Remarks

The *pos* position counter is updated after each call. If the retrieved element is the last in the tree, *pos* is set to NULL.

See Also

Reference

CRBTree Class
CRBTree::GetPrev
CRBTree::GetNext
CRBTree::GetAt

Other Resources

CRBTree Members

CRBTree::GetNextKey

Call this method to get the key of an element stored in the tree and advance the position to the next element.

```
const K& GetNextKey(
   POSITION& pos
) const throw( );
```

Parameters

pos

The position counter, returned by a previous call to methods such as CRBTree::GetHeadPosition or CRBTree::FindFirstKeyAfter.

Return Value

Returns a reference to the next key in the tree.

Remarks

Updates the current position counter, *pos*. If there are no more entries in the tree, the position counter is set to NULL.

See Also

Reference

CRBTree Class
CRBTree::FindFirstKeyAfter
CRBTree::GetNextValue

Other Resources

CRBTree Members

CRBTree::GetNextValue

Call this method to get the value of an element stored in the tree and advance the position to the next element.

```
const V& GetNextValue(
   POSITION& pos
) const throw( );
V& GetNextValue(
   POSITION& pos
) throw( );
```

Parameters

pos

The position counter, returned by a previous call to methods such as CRBTree::GetHeadPosition or CRBTree::FindFirstKeyAfter.

Return Value

Returns a reference to the next value in the tree.

Remarks

Updates the current position counter, *pos*. If there are no more entries in the tree, the position counter is set to NULL.

See Also

Reference

CRBTree Class

CRBTree::GetNextKey

Other Resources

CRBTree Members

CRBTree::GetPrev

Call this method to obtain a pointer to an element stored in the **CRBTree** object, and then update the position to the previous element.

```
const CPair* GetPrev(
   POSITION& pos
) const throw( );
CPair* GetPrev(
   POSITION& pos
) throw( );
```

Parameters

pos

The position counter, returned by a previous call to methods such as CRBTree::GetHeadPosition or CRBTree::FindFirstKeyAfter.

Return Value

Returns a pointer to the previous CPair value stored in the tree.

Remarks

Updates the current position counter, *pos*. If there are no more entries in the tree, the position counter is set to NULL.

See Also

Reference

CRBTree Class

CRBTree::GetNext

Other Resources

CRBTree Members

CRBTree::GetTailPosition

Call this method to get the position value for the element at the tail of the tree.

```
POSITION GetTailPosition( ) const throw( );
```

Return Value

Returns the position value for the element at the tail of the tree.

Remarks

The value returned by **GetTailPosition** can be used with methods such as CRBTree::GetKeyAt or CRBTree::GetPrev to traverse the tree and retrieve values.

See Also

Reference

CRBTree Class

CRBTree::GetHeadPosition

Other Resources

CRBTree Members

CRBTree::GetValueAt

Call this method to retrieve the value stored at a given position in the **CRBTree** object.

```
const V& GetValueAt(
   POSITION pos
) const throw( );
V& GetValueAt(
   POSITION pos
) throw( );
```

Parameters

pos

The position counter, returned by a previous call to methods such as CRBTree::GetHeadPosition or CRBTree::FindFirstKeyAfter.

Return Value

Returns a reference to the value stored at the given position in the **CRBTree** object.

See Also

Reference

CRBTree Class
CRBTree::GetAt
CRBTree::GetKeyAt

Other Resources

CRBTree Members

CRBTree::IsEmpty

Call this method to test for an empty tree object.

```
bool IsEmpty( ) const throw( );
```

Return Value

Returns **true** if the tree is empty, **false** otherwise.

See Also

Reference

CRBTree Class

CRBTree::GetCount

Other Resources

CRBTree Members

CRBTree::RemoveAll

Call this method to remove all elements from the **CRBTree** object.

```
void RemoveAll( ) throw( );
```

Remarks

Clears out the **CRBTree** object, freeing the memory used to store the elements.

See Also

Reference

CRBTree Class

CRBTree::RemoveAt

Other Resources

CRBTree Members

CRBTree::RemoveAt

Call this method to remove the element at the given position in the **CRBTree** object.

```
void RemoveAt(
   POSITION pos
) throw( );
```

Parameters

pos

The position counter, returned by a previous call to methods such as CRBTree::GetHeadPosition or CRBTree::FindFirstKeyAfter.

Remarks

Removes the key/value pair stored at the specified position. The memory used to store the element is freed. The POSITION referenced by *pos* becomes invalid, and while the POSITION of any other elements in the tree remains valid, they do not necessarily retain the same order.

See Also

Reference

CRBTree Class

CRBTree::RemoveAll

Other Resources

CRBTree Members

CRBTree::SetValueAt

Call this method to change the value stored at a given position in the **CRBTree** object.

```
void SetValueAt(
   POSITION pos,
   VINARGTYPE value
);
```

Parameters

pos

The position counter, returned by a previous call to methods such as CRBTree::GetHeadPosition or CRBTree::FindFirstKeyAfter.

value

The value to add to the **CRBTree** object.

Remarks

Changes the value element stored at the given position in the **CRBTree** object.

See Also

Reference

CRBTree Class

Other Resources

CRBTree Members

CRBTree Typedefs

For information about the typedefs in **CRBTree**, see CRBTree Members.

CRBTree::KINARGTYPE

Type used when a key is passed as an input argument.

```
typedef KTraits::INARGTYPE KINARGTYPE;
```

See Also

Reference

CRBTree Class

CRBTree::KOUTARGTYPE

Other Resources

CRBTree Members

CRBTree::KOUTARGTYPE

Type used when a key is returned as an output argument.

```
typedef KTraits::OUTARGTYPE KOUTARGTYPE;
```

See Also

Reference

CRBTree Class

CRBTree::KINARGTYPE

Other Resources

CRBTree Members

CRBTree::VINARGTYPE

Type used when a value is passed as an input argument.

```
typedef VTraits::INARGTYPE VINARGTYPE;
```

See Also

Reference

CRBTree Class

CRBTree::VOUTARGTYPE

Other Resources

CRBTree Members

CRBTree::VOUTARGTYPE

Type used when a value is passed as an output argument.

```
typedef VTraits::OUTARGTYPE VOUTARGTYPE;
```

See Also

Reference

CRBTree Class

CRBTree::VINARGTYPE

Other Resources

CRBTree Members

CRBTree Classes

For information about the classes in **CRBTree**, see CRBTree Members.

CRBTree::CPair Class

A class containing the key and value elements.

```
class CPair : public __POSITION
```

Remarks

This class is used by the methods CRBTree::GetAt, CRBTree::GetNext, and CRBTree::GetPrev to access the key and value elements stored in the tree structure.

The members are as follows:

m_key	The data member storing the key element.
m_value	The data member storing the value element.

Requirements

Header: atlcoll.h

See Also

Reference

CRBTree Class

Other Resources

CRBTree Members

CRBTree::CPair Data Members

For information about the **CPair** data members, see CRBTree::CPair Class. For information about the data members in **CRBTree**, see CRBTree Members.

CRBTree::CPair::m_key

The data member storing the key element.

```
const K m_key;
```

Parameters

K

The key element type.

See Also

Reference

CRBTree::CPair Class

CRBTree::CPair::m_value

The data member storing the value element.

```
V m_value;
```

Parameters

V

The value element type.

See Also

Reference

CRBTree::CPair Class

CRegKey Class

This class provides methods for manipulating entries in the system registry.

```
class CRegKey
```

Remarks

CRegKey provides methods for creating and deleting keys and values in the system registry. The registry contains an installation-specific set of definitions for system components, such as software version numbers, logical-to-physical mappings of installed hardware, and COM objects.

CRegKey provides a programming interface to the system registry for a given machine. For example, to open a particular registry key, call **CRegKey::Open**. To retrieve or modify a data value, call **CRegKey::QueryValue** or **CRegKey::SetValue**, respectively. To close a key, call **CRegKey::Close**.

When you close a key, its registry data is written (flushed) to the hard disk. This process may take several seconds. If your application must explicitly write registry data to the hard disk, you can call the RegFlushKey Win32 function. However, **RegFlushKey** uses many system resources and should be called only when absolutely necessary.

Security Note
Any methods that allow the caller to specify a registry location have the potential to read data that cannot be trusted. Methods that make use of RegQueryValueEx should take into consideration that this function does not explicitly handle strings which are NULL terminated. Both conditions should be checked for by the calling code.

Requirements

Header: atlbase.h

See Also

Tasks

DCOM Sample: Demonstrates Remotely Calling a COM Object

Reference

Registry Overview

Platform SDK Registry Functions

Registry Value Types

Other Resources

CRegKey Members

ATL Class Overview

CRegKey Members

Methods

Attach	Call this method to attach an HKEY to the **CRegKey** object by setting the m_hKey member handle to *hKey*.
Close	Call this method to release the m_hKey member handle and set it to NULL.
Create	Call this method to create the specified key, if it does not exist as a subkey of *hKeyParent*.
CRegKey	The constructor.
~CRegKey	The destructor.
DeleteSubKey	Call this method to remove the specified key from the registry.
DeleteValue	Call this method to remove a value field from m_hKey.
Detach	Call this method to detach the m_hKey member handle from the **CRegKey** object and set **m_hKey** to NULL.
EnumKey	Call this method to enumerate the subkeys of the open registry key.
Flush	Call this method to write all of the attributes of the open registry key into the registry.
GetKeySecurity	Call this method to retrieve a copy of the security descriptor protecting the open registry key.
NotifyChangeKeyValue	This method notifies the caller about changes to the attributes or contents of the open registry key.
Open	Call this method to open the specified key and set m_hKey to the handle of this key.
QueryBinaryValue	Call this method to retrieve the binary data for a specified value name.
QueryDWORDValue	Call this method to retrieve the DWORD data for a specified value name.
QueryGUIDValue	Call this method to retrieve the GUID data for a specified value name.
QueryMultiStringValue	Call this method to retrieve the multistring data for a specified value name.
QueryQWORDValue	Call this method to retrieve the QWORD data for a specified value name.
QueryStringValue	Call this method to retrieve the string data for a specified value name.
QueryValue	Call this method to retrieve the data for the specified value field of m_hKey. Earlier versions of this method are no longer supported and are marked as **ATL_DEPRECATED**.
RecurseDeleteKey	Call this method to remove the specified key from the registry and explicitly remove any subkeys.
SetBinaryValue	Call this method to set the binary value of the registry key.
SetDWORDValue	Call this method to set the DWORD value of the registry key.

SetGUIDValue	Call this method to set the GUID value of the registry key.
SetKeySecurity	Call this method to set the security of the registry key.
SetKeyValue	Call this method to store data in a specified value field of a specified key.
SetMultiStringValue	Call this method to set the multistring value of the registry key.
SetQWORDValue	Call this method to set the QWORD value of the registry key.
SetStringValue	Call this method to set the string value of the registry key.
SetValue	Call this method to store data in the specified value field of m_hKey. Earlier versions of this method are no longer supported and are marked as **ATL_DEPRECATED**.

Operators

operator HKEY	Converts a **CRegKey** object to an HKEY.
operator =	Assignment operator.

Data Members

m_hKey	Contains a handle of the registry key associated with the **CRegKey** object.

Static Functions

SetValue	Call this method to store data in the specified value field of m_hKey. Earlier versions of this method are no longer supported and are marked as **ATL_DEPRECATED**.

See Also

Reference

CRegKey Class

CRegKey Methods

For information about the methods in **CRegKey**, see CRegKey Members.

CRegKey::Attach

Call this method to attach an HKEY to the **CRegKey** object by setting the m_hKey member handle to *hKey*.

```
void Attach(
   HKEY hKey
) throw( );
```

Parameters

hKey

The handle of a registry key.

Remarks

Attach will assert if **m_hKey** is non-NULL.

See Also

Reference

CRegKey Class

CRegKey::Detach

Other Resources

CRegKey Members

CRegKey::Close

Call this method to release the m_hKey member handle and set it to NULL.

```
LONG Close( ) throw( );
```

Return Value

If successful, returns ERROR_SUCCESS; otherwise returns an error value.

See Also

Reference

CRegKey Class

CRegKey::Open

Other Resources

CRegKey Members

CRegKey::Create

Call this method to create the specified key, if it does not exist as a subkey of *hKeyParent*.

```
LONG Create(
   HKEY hKeyParent,
   LPCTSTR lpszKeyName,
   LPTSTR lpszClass = REG_NONE,
   DWORD dwOptions = REG_OPTION_NON_VOLATILE,
   REGSAM samDesired = KEY_READ | KEY_WRITE,
   LPSECURITY_ATTRIBUTES lpSecAttr = NULL,
   LPDWORD lpdwDisposition = NULL
) throw( );
```

Parameters

hKeyParent

The handle of an open key.

lpszKeyName

Specifies the name of a key to be created or opened. This name must be a subkey of *hKeyParent*.

lpszClass

Specifies the class of the key to be created or opened. The default value is REG_NONE.

dwOptions

Options for the key. The default value is REG_OPTION_NON_VOLATILE. For a list of possible values and descriptions, see RegCreateKeyEx in the *Platform SDK*.

samDesired

The security access for the key. The default value is KEY_READ | KEY_WRITE. For a list of possible values and descriptions, see **RegCreateKeyEx**.

lpSecAttr

A pointer to a SECURITY_ATTRIBUTES structure that indicates whether the handle of the key can be inherited by a child process. By default, this parameter is NULL (meaning the handle cannot be inherited).

lpdwDisposition

[out] If non-NULL, retrieves either REG_CREATED_NEW_KEY (if the key did not exist and was created) or REG_OPENED_EXISTING_KEY (if the key existed and was opened).

Return Value

If successful, returns ERROR_SUCCESS and opens the key. If the method fails, the return value is a nonzero error code defined in WINERROR.H.

Remarks

Create sets the m_hKey member to the handle of this key.

See Also

Reference

CRegKey Class
CRegKey::Open
CRegKey::Close

Other Resources

CRegKey Members

CRegKey::CRegKey

The constructor.

```
CRegKey( ) throw( );
CRegKey(
   CRegKey& key
) throw( );
explicit CRegKey(
   HKEY hKey
) throw( );
```

Parameters

key

A reference to a **CRegKey** object.

hKey

A handle to a registry key.

Remarks

Creates a new **CRegKey** object. The object can be created from an existing **CRegKey** object, or from a handle to a registry key.

See Also

Reference

CRegKey Class

Other Resources

CRegKey Members

CRegKey::~CRegKey

The destructor.

```
~CRegKey( ) throw( );
```

Remarks

The destructor releases **m_hKey**.

See Also

Reference

CRegKey Class

Other Resources

CRegKey Members

CRegKey::DeleteSubKey

Call this method to remove the specified key from the registry.

```
LONG DeleteSubKey(
   LPCTSTR lpszSubKey
) throw( );
```

Parameters

lpszSubKey

Specifies the name of the key to delete. This name must be a subkey of m_hKey.

Return Value

If successful, returns ERROR_SUCCESS. If the method fails, the return value is a nonzero error code defined in WINERROR.H.

Remarks

Under Windows 95 and Windows 98, **DeleteSubKey** deletes the key and all of its subkeys. Under Windows NT/2000, **DeleteSubKey** can only delete a key that has no subkeys. If the key has subkeys, call RecurseDeleteKey instead.

See Also

Reference

CRegKey Class

CRegKey::DeleteValue

Other Resources

CRegKey Members

CRegKey::DeleteValue

Call this method to remove a value field from m_hKey.

```
LONG DeleteValue(
   LPCTSTR lpszValue
) throw( );
```

Parameters

lpszValue

Specifies the value field to remove.

Return Value

If successful, returns ERROR_SUCCESS. If the method fails, the return value is a nonzero error code defined in WINERROR.H.

See Also

Reference

CRegKey Class

CRegKey::DeleteSubKey

Other Resources

CRegKey Members

CRegKey::Detach

Call this method to detach the m_hKey member handle from the **CRegKey** object and set **m_hKey** to NULL.

```
HKEY Detach( ) throw( );
```

Return Value

The HKEY associated with the **CRegKey** object.

See Also

Reference

CRegKey Class

CRegKey::Attach

Other Resources

CRegKey Members

CRegKey::EnumKey

Call this method to enumerate the subkeys of the open registry key.

```
LONG EnumKey(
   DWORD iIndex,
   LPTSTR pszName,
   LPDWORD pnNameLength,
   FILETIME* pftLastWriteTime = NULL
) throw( );
```

Parameters

iIndex

The subkey index. This parameter should be zero for the first call and then incremented for subsequent calls

pszName

Pointer to a buffer that receives the name of the subkey, including the terminating null character. Only the name of the subkey is copied to the buffer, not the full key hierarchy.

pnNameLength

Pointer to a variable that specifies the size, in TCHARs, of the buffer specified by the *pszName* parameter. This size should include the terminating null character. When the method returns, the variable pointed to by *pnNameLength* contains the number of characters stored in the buffer. The count returned does not include the terminating null character.

pftLastWriteTime

Pointer to a variable that receives the time the enumerated subkey was last written to.

Return Value

If the method succeeds, the return value is ERROR_SUCCESS. If the method fails, the return value is a nonzero error code defined in WINERROR.H.

Remarks

To enumerate the subkeys, call **CRegKey::EnumKey** with an index of zero. Increment the index value and repeat until the method returns ERROR_NO_MORE_ITEMS. For more information, see RegEnumKeyEx in the *Platform SDK*.

See Also

Reference

CRegKey Class

Other Resources

CRegKey Members

CRegKey::Flush

Call this method to write all of the attributes of the open registry key into the registry.

```
LONG Flush( ) throw( );
```

Return Value

If the method succeeds, the return value is ERROR_SUCCESS. If the method fails, the return value is a nonzero error code defined in WINERROR.H.

Remarks

For more information, see RegEnumFlush in the *Platform SDK*.

See Also

Reference

CRegKey Class

Other Resources

CRegKey Members

CRegKey::GetKeySecurity

Call this method to retrieve a copy of the security descriptor protecting the open registry key.

```
LONG GetKeySecurity(
   SECURITY_INFORMATION si,
   PSECURITY_DESCRIPTOR psd,
   LPDWORD pnBytes
) throw( );
```

Parameters

si

The SECURITY_INFORMATION value that indicates the requested security information.

psd

A pointer to a buffer that receives a copy of the requested security descriptor.

pnBytes

The size, in bytes, of the buffer pointed to by *psd*.

Return Value

If the method succeeds, the return value is ERROR_SUCCESS. If the method fails, the return value is a nonzero error code is defined in WINERROR.H.

Remarks

For more information, see RegGetKeySecurity.

See Also

Reference

CRegKey Class

CRegKey::SetKeySecurity

Other Resources

CRegKey Members

CRegKey::NotifyChangeKeyValue

This method notifies the caller about changes to the attributes or contents of the open registry key.

```
LONG NotifyChangeKeyValue(
   BOOL bWatchSubtree,
   DWORD dwNotifyFilter,
   HANDLE hEvent,
   BOOL bAsync = TRUE
) throw( );
```

Parameters

bWatchSubtree

Specifies a flag that indicates whether to report changes in the specified key and all of its subkeys or only in the specified key. If this parameter is TRUE, the method reports changes in the key and its subkeys. If the parameter is FALSE, the method reports changes only in the key.

dwNotifyFilter

Specifies a set of flags that control which changes should be reported. This parameter can be a combination of the following values:

Value	Meaning
REG_NOTIFY_CHANGE_NAME	Notify the caller if a subkey is added or deleted.
REG_NOTIFY_CHANGE_ATTRIBUTES	Notify the caller of changes to the attributes of the key, such as the security descriptor information.
REG_NOTIFY_CHANGE_LAST_SET	Notify the caller of changes to a value of the key. This can include adding or deleting a value, or changing an existing value.
REG_NOTIFY_CHANGE_SECURITY	Notify the caller of changes to the security descriptor of the key.

hEvent

Handle to an event. If the *bAsync* parameter is TRUE, the method returns immediately and changes are reported by signaling this event. If *bAsync* is FALSE, *hEvent* is ignored.

bAsync

Specifies a flag that indicates how the method reports changes. If this parameter is TRUE, the method returns immediately and reports changes by signaling the specified event. When this parameter is FALSE, the method does not return until a change has occurred. If *hEvent* does not specify a valid event, the *bAsync* parameter cannot be TRUE.

Return Value

If the method succeeds, the return value is ERROR_SUCCESS. If the method fails, the return value is a nonzero error code defined in WINERROR.H.

Remarks

Note
This method does not notify the caller if the specified key is deleted.

For more details and a sample program, see RegNotifyChangeKeyValue.

See Also

Reference

CRegKey Class

Other Resources

CRegKey Members

CRegKey::Open

Call this method to open the specified key and set m_hKey to the handle of this key.

```
LONG Open(
   HKEY hKeyParent,
   LPCTSTR lpszKeyName,
   REGSAM samDesired = KEY_READ | KEY_WRITE
) throw( );
```

Parameters

hKeyParent

The handle of an open key.

lpszKeyName

Specifies the name of a key to be created or opened. This name must be a subkey of *hKeyParent*.

samDesired

The security access for the key. The default value is KEY_ALL_ACCESS. For a list of possible values and descriptions, see RegCreateKeyEx in the *Platform SDK*.

Return Value

If successful, returns ERROR_SUCCESS; otherwise, a non-zero error value defined in WINERROR.H.

Remarks

If the *lpszKeyName* parameter is NULL or points to an empty string, **Open** opens a new handle of the key identified by *hKeyParent*, but does not close any previously opened handle.

Unlike CRegKey::Create, **Open** will not create the specified key if it does not exist.

See Also

Reference

CRegKey Class

CRegKey::Close

Other Resources

CRegKey Members

CRegKey::QueryBinaryValue

Call this method to retrieve the binary data for a specified value name.

```
LONG QueryBinaryValue(
   LPCTSTR pszValueName,
   void* pValue,
   ULONG* pnBytes
) throw( );
```

Parameters

pszValueName

Pointer to a null-terminated string containing the name of the value to query.

pValue

Pointer to a buffer that receives the value's data.

pnBytes

Pointer to a variable that specifies the size, in bytes, of the buffer pointed to by the *pValue* parameter. When the method returns, this variable contains the size of the data copied to the buffer.

Return Value

If the method succeeds, ERROR_SUCCESS is returned. If the method fails to read a value, it returns a nonzero error code defined in WINERROR.H. If the data referenced is not of type REG_BINARY, ERROR_INVALID_DATA is returned.

Remarks

This method makes use of **RegQueryValueEx** and confirms that the correct type of data is returned. See RegQueryValueEx for more details.

Security Note
This method allows the caller to specify any registry location, potentially reading data which cannot be trusted. Also, the RegQueryValueEx function used by this method does not explicitly handle strings which are NULL terminated. Both condition s should be checked for by the calling code.

See Also

Reference

CRegKey Class
CRegKey::SetBinaryValue
CRegKey::QueryDWORDValue
CRegKey::QueryGUIDValue
CRegKey::QueryMultiStringValue
CRegKey::QueryQWORDValue
CRegKey::QueryStringValue

Other Resources

CRegKey Members

CRegKey::QueryDWORDValue

Call this method to retrieve the DWORD data for a specified value name.

```
LONG QueryDWORDValue(
   LPCTSTR pszValueName,
   DWORD& dwValue
) throw( );
```

Parameters

pszValueName

Pointer to a null-terminated string containing the name of the value to query.

dwValue

Pointer to a buffer that receives the DWORD.

Return Value

If the method succeeds, ERROR_SUCCESS is returned. If the method fails to read a value, it returns a nonzero error code defined in WINERROR.H. If the data referenced is not of type REG_DWORD, ERROR_INVALID_DATA is returned.

Remarks

This method makes use of **RegQueryValueEx** and confirms that the correct type of data is returned. See RegQueryValueEx for more details.

Security Note
This method allows the caller to specify any registry location, potentially reading data which cannot be trusted. Also, the RegQueryValueEx function used by this method does not explicitly handle strings which are NULL terminated. Both condition s should be checked for by the calling code.

See Also

Reference

CRegKey Class
CRegKey::SetDWORDValue
CRegKey::QueryBinaryValue
CRegKey::QueryGUIDValue
CRegKey::QueryMultiStringValue
CRegKey::QueryQWORDValue
CRegKey::QueryStringValue

Other Resources

CRegKey Members

CRegKey::QueryGUIDValue

Call this method to retrieve the GUID data for a specified value name.

```
LONG QueryGUIDValue(
   LPCTSTR pszValueName,
   GUID& guidValue
) throw( );
```

Parameters

pszValueName

Pointer to a null-terminated string containing the name of the value to query.

guidValue

Pointer to a variable that receives the GUID.

Return Value

If the method succeeds, ERROR_SUCCESS is returned. If the method fails to read a value, it returns a nonzero error code defined in WINERROR.H. If the data referenced is not a valid GUID, ERROR_INVALID_DATA is returned.

Remarks

This method makes use of **CRegKey::QueryStringValue** and converts the string into a GUID using CLSIDFromString.

Security Note
This method allows the caller to specify any registry location, potentially reading data which cannot be trusted.

See Also

Reference

CRegKey Class
CRegKey::SetGUIDValue
CRegKey::QueryBinaryValue
CRegKey::QueryDWORDValue
CRegKey::QueryMultiStringValue
CRegKey::QueryQWORDValue
CRegKey::QueryStringValue

Other Resources

CRegKey Members

CRegKey::QueryMultiStringValue

Call this method to retrieve the multistring data for a specified value name.

```
LONG QueryMultiStringValue(
   LPCTSTR pszValueName,
   LPTSTR pszValue,
   ULONG* pnChars
) throw( );
```

Parameters

pszValueName

Pointer to a null-terminated string containing the name of the value to query.

pszValue

Pointer to a buffer that receives the multistring data. A multistring is an array of null-terminated strings, terminated by two null characters.

pnChars

The size, in TCHARs, of the buffer pointed to by *pszValue*. When the method returns, *pnChars* contains the size, in TCHARs, of the multistring retrieved, including a terminating null character.

Return Value

If the method succeeds, ERROR_SUCCESS is returned. If the method fails to read a value, it returns a nonzero error code defined in WINERROR.H. If the data referenced is not of type REG_MULTI_SZ, ERROR_INVALID_DATA is returned.

Remarks

This method makes use of **RegQueryValueEx** and confirms that the correct type of data is returned. See RegQueryValueEx for more details.

Security Note
This method allows the caller to specify any registry location, potentially reading data which cannot be trusted. Also, the RegQueryValueEx function used by this method does not explicitly handle strings which are NULL terminated. Both condition s should be checked for by the calling code.

See Also

Reference

CRegKey Class
CRegKey::SetMultiStringValue
CRegKey::QueryBinaryValue
CRegKey::QueryDWORDValue
CRegKey::QueryGUIDValue
CRegKey::QueryQWORDValue
CRegKey::QueryStringValue

Other Resources

CRegKey Members

CRegKey::QueryQWORDValue

Call this method to retrieve the QWORD data for a specified value name.

```
LONG QueryQWORDValue(
   LPCTSTR pszValueName,
   ULONGLONG& qwValue
) throw( );
```

Parameters

pszValueName

Pointer to a null-terminated string containing the name of the value to query.

qwValue

Pointer to a buffer that receives the QWORD.

Return Value

If the method succeeds, ERROR_SUCCESS is returned. If the method fails to read a value, it returns a nonzero error code defined in WINERROR.H. If the data referenced is not of type REG_QWORD, ERROR_INVALID_DATA is returned.

Remarks

This method makes use of **RegQueryValueEx** and confirms that the correct type of data is returned. See RegQueryValueEx for more details.

Security Note
This method allows the caller to specify any registry location, potentially reading data which cannot be trusted. Also, the RegQueryValueEx function used by this method does not explicitly handle strings which are NULL terminated. Both condition s should be checked for by the calling code.

See Also

Reference

CRegKey Class
CRegKey::SetQWORDValue
CRegKey::QueryBinaryValue
CRegKey::QueryDWORDValue
CRegKey::QueryGUIDValue
CRegKey::QueryMultiStringValue
CRegKey::QueryStringValue

Other Resources

CRegKey Members

CRegKey::QueryStringValue

Call this method to retrieve the string data for a specified value name.

```
LONG QueryStringValue(
   LPCTSTR pszValueName,
   LPTSTR pszValue,
   ULONG* pnChars
) throw( );
```

Parameters

pszValueName

Pointer to a null-terminated string containing the name of the value to query.

pszValue

Pointer to a buffer that receives the string data.

pnChars

The size, in TCHARs, of the buffer pointed to by *pszValue*. When the method returns, *pnChars* contains the size, in TCHARs, of the string retrieved, including a terminating null character.

Return Value

If the method succeeds, ERROR_SUCCESS is returned. If the method fails to read a value, it returns a nonzero error code defined in WINERROR.H. If the data referenced is not of type REG_SZ, ERROR_INVALID_DATA is returned. If the method returns ERROR_MORE_DATA, *pnChars* equals zero, not the required buffer size in bytes.

Remarks

This method makes use of **RegQueryValueEx** and confirms that the correct type of data is returned. See RegQueryValueEx for more details.

Security Note
This method allows the caller to specify any registry location, potentially reading data which cannot be trusted. Also, the RegQueryValueEx function used by this method does not explicitly handle strings which are NULL terminated. Both condition s should be checked for by the calling code.

See Also

Reference

CRegKey Class
CRegKey::SetStringValue
CRegKey::QueryBinaryValue
CRegKey::QueryDWORDValue
CRegKey::QueryGUIDValue
CRegKey::QueryMultiStringValue
CRegKey::QueryQWORDValue

Other Resources

CRegKey Members

CRegKey::QueryValue

Call this method to retrieve the data for the specified value field of m_hKey. Earlier versions of this method are no longer supported and are marked as **ATL_DEPRECATED**.

```
LONG QueryValue(
   LPCTSTR pszValueName,
   DWORD* pdwType,
   void* pData,
      ULONG* pnBytes
) throw( );
ATL_DEPRECATED LONG QueryValue(
   DWORD& dwValue,
   LPCTSTR lpszValueName
);
ATL_DEPRECATED LONG QueryValue(
   LPTSTR szValue,
   LPCTSTR lpszValueName,
   DWORD* pdwCount
);
```

Parameters

pszValueName

Pointer to a null-terminated string containing the name of the value to query. If *pszValueName* is NULL or an empty string, "", the method retrieves the type and data for the key's unnamed or default value, if any.

pdwType

Pointer to a variable that receives a code indicating the type of data stored in the specified value. The *pdwType* parameter can be NULL if the type code is not required.

pData

Pointer to a buffer that receives the value's data. This parameter can be NULL if the data is not required.

pnBytes

Pointer to a variable that specifies the size, in bytes, of the buffer pointed to by the *pData* parameter. When the method returns, this variable contains the size of the data copied to *pData*.

dwValue

The value field's numerical data.

lpszValueName

Specifies the value field to be queried.

szValue

The value field's string data.

pdwCount

The size of the string data. Its value is initially set to the size of the *szValue* buffer.

Return Value

If successful, returns ERROR_SUCCESS; otherwise, a nonzero error code defined in WINERROR.H.

Remarks

The two original versions of **QueryValue** are no longer supported and are marked as **ATL_DEPRECATED**. The compiler will issue a warning if these forms are used.

The remaining method calls RegQueryValueEx.

Security Note

This method allows the caller to specify any registry location, potentially reading data which cannot be trusted. Also, the RegQueryValueEx function used by this method does not explicitly handle strings which are NULL terminated. Both conditions should be checked for by the calling code.

See Also

Reference

CRegKey Class

CRegKey::SetValue

Registry Value Types

Other Resources

CRegKey Members

CRegKey::RecurseDeleteKey

Call this method to remove the specified key from the registry and explicitly remove any subkeys.

```
LONG RecurseDeleteKey(
   LPCTSTR lpszKey
) throw( );
```

Parameters

lpszKey

Specifies the name of the key to delete. This name must be a subkey of m_hKey.

Return Value

If successful, returns ERROR_SUCCESS; otherwise, a non-zero error value defined in WINERROR.H.

Remarks

If the key has subkeys, under Windows NT and Windows 2000 you must call this method to delete the key. Under Windows 95 and Windows 98, you can call DeleteSubKey to remove the key and any subkeys.

See Also

Reference

CRegKey Class

Other Resources

CRegKey Members

CRegKey::SetBinaryValue

Call this method to set the binary value of the registry key.

```
LONG SetBinaryValue(
   LPCTSTR pszValueName,
   const void* pValue,
   ULONG nBytes
) throw( );
```

Parameters

pszValueName

Pointer to a string containing the name of the value to set. If a value with this name is not already present, the method adds it to the key.

pValue

Pointer to a buffer containing the data to be stored with the specified value name.

nBytes

Specifies the size, in bytes, of the information pointed to by the *pValue* parameter.

Return Value

If the method succeeds, the return value is ERROR_SUCCESS. If the method fails, the return value is a nonzero error code defined in WINERROR.H.

Remarks

This method uses RegSetValueEx to write the value to the registry.

See Also

Reference

CRegKey Class
CRegKey::QueryBinaryValue
CRegKey::SetDWORDValue
CRegKey::SetQWORDValue
CRegKey::SetGUIDValue
CRegKey::SetStringValue
CRegKey::SetMultiStringValue

Other Resources

CRegKey Members

CRegKey::SetDWORDValue

Call this method to set the DWORD value of the registry key.

```
LONG SetDWORDValue(
   LPCTSTR pszValueName,
   DWORD dwValue
) throw( );
```

Parameters

pszValueName

Pointer to a string containing the name of the value to set. If a value with this name is not already present, the method adds it to the key.

dwValue

The DWORD data to be stored with the specified value name.

Return Value

If the method succeeds, the return value is ERROR_SUCCESS. If the method fails, the return value is a nonzero error code defined in WINERROR.H.

Remarks

This method uses RegSetValueEx to write the value to the registry.

See Also

Reference

CRegKey Class
CRegKey::QueryDWORDValue
CRegKey::SetQWORDValue
CRegKey::SetBinaryValue
CRegKey::SetGUIDValue
CRegKey::SetStringValue
CRegKey::SetMultiStringValue

Other Resources

CRegKey Members

CRegKey::SetGUIDValue

Call this method to set the GUID value of the registry key.

```
LONG SetGUIDValue(
   LPCTSTR pszValueName,
   REFGUID guidValue
) throw( );
```

Parameters

pszValueName

Pointer to a string containing the name of the value to set. If a value with this name is not already present, the method adds it to the key.

guidValue

Reference to the GUID to be stored with the specified value name.

Return Value

If the method succeeds, the return value is ERROR_SUCCESS. If the method fails, the return value is a nonzero error code defined in WINERROR.H.

Remarks

This method makes use of **CRegKey::SetStringValue** and converts the GUID into a string using StringFromGUID2.

See Also

Reference

CRegKey Class
CRegKey::QueryGUIDValue
CRegKey::SetBinaryValue
CRegKey::SetDWORDValue
CRegKey::SetQWORDValue
CRegKey::SetStringValue
CRegKey::SetMultiStringValue

Other Resources

CRegKey Members

CRegKey::SetKeyValue

Call this method to store data in a specified value field of a specified key.

```
LONG SetKeyValue(
   LPCTSTR lpszKeyName,
   LPCTSTR lpszValue,
   LPCTSTR lpszValueName = NULL
) throw( );
```

Parameters

lpszKeyName

Specifies the name of the key to be created or opened. This name must be a subkey of m_hKey.

lpszValue

Specifies the data to be stored. This parameter must be non-NULL.

lpszValueName

Specifies the value field to be set. If a value field with this name does not already exist in the key, it is added.

Return Value

If successful, returns ERROR_SUCCESS; otherwise, a nonzero error code defined in WINERROR.H.

Remarks

Call this method to create or open the *lpszKeyName* key and store the *lpszValue* data in the *lpszValueName* value field.

See Also

Reference

CRegKey Class

CRegKey::SetValue

Other Resources

CRegKey Members

CRegKey::SetKeySecurity

Call this method to set the security of the registry key.

```
LONG SetKeySecurity(
   SECURITY_INFORMATION si,
   PSECURITY_DESCRIPTOR psd
) throw( );
```

Parameters

si

Specifies the components of the security descriptor to set. The value can be a combination of the following values:

Value	Meaning
DACL_SECURITY_INFORMATION	Sets the key's discretionary access-control list (DACL). The key must have WRITE_DAC access, or the calling process must be the object's owner.
GROUP_SECURITY_INFORMATION	Sets the key's primary group security identifier (SID). The key must have WRITE_OWNER access, or the calling process must be the object's owner.
OWNER_SECURITY_INFORMATION	Sets the key's owner SID. The key must have WRITE_OWNER access, or the calling process must be the object's owner or have the SE_TAKE_OWNERSHIP_NAME privilege enabled.
SACL_SECURITY_INFORMATION	Sets the key's system access-control list (SACL). The key must have ACCESS_SYSTEM_SECURITY access. The proper way to get this access is to enable the SE_SECURITY_NAME privilege in the caller's current access token, open the handle for ACCESS_SYSTEM_SECURITY access, and then disable the privilege.

psd

Pointer to a SECURITY_DESCRIPTOR structure that specifies the security attributes to set for the specified key.

Return Value

If the method succeeds, the return value is ERROR_SUCCESS. If the method fails, the return value is a nonzero error code defined in WINERROR.H.

Remarks

Sets the key's security attributes. See RegSetKeySecurity for more details.

See Also

Reference

CRegKey Class

CRegKey::GetKeySecurity

Other Resources

CRegKey Members

CRegKey::SetMultiStringValue

Call this method to set the multistring value of the registry key.

```
LONG SetMultiStringValue(
   LPCTSTR pszValueName,
   LPCTSTR pszValue
) throw( );
```

Parameters

pszValueName

Pointer to a string containing the name of the value to set. If a value with this name is not already present, the method adds it to the key.

pszValue

Pointer to the multistring data to be stored with the specified value name. A multistring is an array of null-terminated strings, terminated by two null characters.

Return Value

If the method succeeds, the return value is ERROR_SUCCESS. If the method fails, the return value is a nonzero error code defined in WINERROR.H.

Remarks

This method uses RegSetValueEx to write the value to the registry.

See Also

Reference

CRegKey Class
CRegKey::QueryMultiStringValue
CRegKey::SetDWORDValue
CRegKey::SetQWORDValue
CRegKey::SetGUIDValue
CRegKey::SetStringValue
CRegKey::SetBinaryValue

Other Resources

CRegKey Members

CRegKey::SetQWORDValue

Call this method to set the QWORD value of the registry key.

```
LONG SetQWORDValue(
   LPCTSTR pszValueName,
   ULONGLONG qwValue
) throw( );
```

Parameters

pszValueName

Pointer to a string containing the name of the value to set. If a value with this name is not already present, the method adds it to the key.

qwValue

The QWORD data to be stored with the specified value name.

Return Value

If the method succeeds, the return value is ERROR_SUCCESS. If the method fails, the return value is a nonzero error code defined in WINERROR.H.

Remarks

This method uses RegSetValueEx to write the value to the registry.

See Also

Reference

CRegKey Class
CRegKey::QueryQWORDValue
CRegKey::SetDWORDValue
CRegKey::SetGUIDValue
CRegKey::SetStringValue
CRegKey::SetMultiStringValue
CRegKey::SetBinaryValue

Other Resources

CRegKey Members

CRegKey::SetStringValue

Call this method to set the string value of the registry key.

```
LONG SetStringValue(
   LPCTSTR pszValueName,
   LPCTSTR pszValue,
   DWORD dwType = REG_SZ
) throw( );
```

Parameters

pszValueName

Pointer to a string containing the name of the value to set. If a value with this name is not already present, the method adds it to the key.

pszValue

Pointer to the string data to be stored with the specified value name.

dwType

The type of the string to write to the registry: either REG_SZ (the default) or REG_EXPAND_SZ (for multistrings).

Return Value

If the method succeeds, the return value is ERROR_SUCCESS. If the method fails, the return value is a nonzero error code defined in WINERROR.H.

Remarks

This method uses RegSetValueEx to write the value to the registry.

See Also

Reference

CRegKey Class
CRegKey::QueryStringValue
CRegKey::SetDWORDValue
CRegKey::SetQWORDValue
CRegKey::SetGUIDValue
CRegKey::SetMultiStringValue
CRegKey::SetBinaryValue

Other Resources

CRegKey Members

CRegKey::SetValue

Call this method to store data in the specified value field of m_hKey. Earlier versions of this method are no longer supported and are marked as **ATL_DEPRECATED**.

```
LONG SetValue(
   LPCTSTR pszValueName,
   DWORD dwType,
   const void* pValue,
   ULONG nBytes
) throw( );
static LONG WINAPI SetValue(
   HKEY hKeyParent,
   LPCTSTR lpszKeyName,
   LPCTSTR lpszValue,
   LPCTSTR lpszValueName = NULL);
ATL_DEPRECATED LONG SetValue(
   DWORD dwValue,
   LPCTSTR lpszValueName
);
ATL_DEPRECATED LONG SetValue(
   LPCTSTR lpszValue,
   LPCTSTR lpszValueName = NULL,
   bool bMulti = false,
   int nValueLen = -1
);
```

Parameters

pszValueName

Pointer to a string containing the name of the value to set. If a value with this name is not already present in the key, the method adds it to the key. If *pszValueName* is NULL or an empty string, "", the method sets the type and data for the key's unnamed or default value.

dwType

Specifies a code indicating the type of data pointed to by the *pValue* parameter.

pValue

Pointer to a buffer containing the data to be stored with the specified value name.

nBytes

Specifies the size, in bytes, of the information pointed to by the *pValue* parameter. If the data is of type REG_SZ, REG_EXPAND_SZ, or REG_MULTI_SZ, *nBytes* must include the size of the terminating null character.

hKeyParent

The handle of an open key.

lpszKeyName

Specifies the name of a key to be created or opened. This name must be a subkey of *hKeyParent*.

lpszValue

Specifies the data to be stored. This parameter must be non-NULL.

lpszValueName

Specifies the value field to be set. If a value field with this name does not already exist in the key, it is added.

dwValue

Specifies the data to be stored.

bMulti

If false, indicates the string is of type REG_SZ. If true, indicates the string is a multistring of type REG_MULTI_SZ.

nValueLen

If *bMulti* is true, *nValueLen* is the length of the *lpszValue* string in characters. If *bMulti* is false, a value of -1 indicates that the method will calculate the length automatically.

Return Value

If successful, returns ERROR_SUCCESS; otherwise, a nonzero error code defined in WINERROR.H.

Remarks

The two original versions of **SetValue** are marked as **ATL_DEPRECATED** and should no longer be used. The compiler will issue a warning if these forms are used.

The third method calls RegSetValueEx.

See Also

Reference

CRegKey Class
CRegKey::SetKeyValue
CRegKey::QueryValue
Registry Value Types

Other Resources

CRegKey Members

CRegKey Operators

For information about the operators in **CRegKey**, see CRegKey Members.

CRegKey::operator HKEY

Converts a **CRegKey** object to an HKEY.

```
operator HKEY( ) const throw( );
```

See Also

Reference

CRegKey Class

Other Resources

CRegKey Members

CRegKey::operator =

Assignment operator.

```
CRegKey& operator =(
   CRegKey& key
) throw( );
```

Parameters

key

The key to copy.

Return Value

Returns a reference to the new key.

Remarks

This operator detaches *key* from its current object and assigns it to the **CRegKey** object instead.

See Also

Reference

CRegKey Class

Other Resources

CRegKey Members

CRegKey Data Members

For information about the data members in **CRegKey**, see CRegKey Members.

CRegKey::m_hKey

Contains a handle of the registry key associated with the **CRegKey** object.

```
HKEY m_hKey;
```

See Also

Reference

CRegKey Class

Other Resources

CRegKey Members

CRTThreadTraits Class

This class provides the creation function for a CRT thread. Use this class if the thread will use CRT functions.

```
class CRTThreadTraits
```

Remarks

Thread traits are classes that provide a creation function for a particular type of thread. The creation function has the same signature and semantics as the Windows CreateThread function.

Thread traits are used by the following classes:

- CThreadPool
- CWorkerThread

If the thread will not be using CRT functions, use Win32ThreadTraits instead.

Requirements

Header: atlbase.h

See Also

Other Resources

CRTThreadTraits Members

ATL Class Overview

CRTThreadTraits Members

Static Functions

CreateThread	Call this function to create a thread that can use CRT functions.

See Also

Reference

CRTThreadTraits Class

CRTThreadTraits Static Functions

For information about the static functions in **CRTThreadTraits**, see CRTThreadTraits Members.

CRTThreadTraits::CreateThread

Call this function to create a thread that can use CRT functions.

```
static HANDLE CreateThread(
   LPSECURITY_ATTRIBUTES lpsa,
   DWORD dwStackSize,
   LPTHREAD_START_ROUTINE pfnThreadProc,
   void* pvParam,
   DWORD dwCreationFlags,
   DWORD* pdwThreadId
) throw( );
```

Parameters

lpsa

The security attributes for the new thread.

dwStackSize

The stack size for the new thread.

pfnThreadProc

The thread procedure of the new thread.

pvParam

The parameter to be passed to the thread procedure.

dwCreationFlags

The creation flags (0 or CREATE_SUSPENDED).

pdwThreadId

[out] Address of the DWORD variable that, on success, receives the thread ID of the newly created thread.

Return Value

Returns the handle to the newly created thread or NULL on failure. Call GetLastError to get extended error information.

Remarks

See CreateThread for further information on the parameters to this function.

This function calls _beginthreadex to create the thread.

See Also

Reference

CRTThreadTraits Class

Other Resources

CRTThreadTraits Members

CSacl Class

This class is a wrapper for a SACL (system access-control list) structure.

```
class CSacl : public CAcl
```

Remarks

A SACL contains access-control entries (ACEs) that specify the types of access attempts that generate audit records in the security event log of a domain controller. Note that a SACL generates log entries only on the domain controller where the access attempt occurred, not on every domain controller that contains a replica of the object.

To set or retrieve the SACL in an object's security descriptor, the SE_SECURITY_NAME privilege must be enabled in the access token of the requesting thread. The administrators group has this privilege granted by default, and it can be granted to other users or groups. Having the privilege granted is not all that is required: before the operation defined by the privilege can be performed, the privilege must be enabled in the security access token in order to take effect. The model allows privileges to be enabled only for specific system operations, and then disabled when they are no longer needed. See AtlGetSacl and AtlSetSacl for examples of enabling SE_SECURITY_NAME.

Use the class methods provided to add, remove, create, and delete ACEs from the **SACL** object. See also AtlGetSacl and AtlSetSacl.

For an introduction to the access control model in Windows, see Access Control in the *Platform SDK*.

Requirements

Header: atlsecurity.h

See Also

Reference

CAcl Class
ACLs
ACEs

Other Resources

CSacl Members
ATL Class Overview
Security Global Functions

CSacl Members

Methods

CSacl	The constructor.
~CSacl	The destructor.
AddAuditAce	Adds an audit access-control entry (ACE) to the **CSacl** object.
GetAceCount	Returns the number of access-control entries (ACEs) in the **CSacl** object.
RemoveAce	Removes a specific ACE (access-control entry) from the **CSacl** object.
RemoveAllAces	Removes all of the ACEs contained in the **CSacl** object.

Operators

operator =	Assignment operator.

See Also

Reference

CSacl Class

CSacl Methods

For information about the methods in **CSacl**, see CSacl Members.

CSacl::AddAuditAce

Adds an audit access-control entry (ACE) to the **CSacl** object.

```
bool AddAuditAce(
   const CSid & rSid,
   ACCESS_MASK AccessMask,
   bool bSuccess,
   bool bFailure,
   BYTE AceFlags = 0
) throw(...);
bool AddAuditAce(
   const CSid & rSid,
   ACCESS_MASK AccessMask,
   bool bSuccess,
   bool bFailure,
   BYTE AceFlags,
   const GUID * pObjectType,
   const GUID * pInheritedObjectType
) throw(...);
```

Parameters

rSid

The CSid object.

AccessMask

Specifies the mask of access rights to be audited for the specified **CSid** object.

bSuccess

Specifies whether allowed access attempts are to be audited. Set this flag to true to enable auditing; otherwise, set it to false.

bFailure

Specifies whether denied access attempts are to be audited. Set this flag to true to enable auditing; otherwise, set it to false.

AceFlags

A set of bit flags that control ACE inheritance.

pObjectType

The object type.

pInheritedObjectType

The inherited object type.

Return Value

Returns **true** if the ACE is added to the **CSacl** object, **false** on failure.

Remarks

A **CSacl** object contains access-control entries (ACEs) that specify the types of access attempts that generate audit records in the security event log. This method adds such an ACE to the **CSacl** object. The second form of **AddAuditAce** is only available on Windows 2000 and later.

See ACE_HEADER for a description of the various flags which can be set in the *AceFlags* parameter.

See Also

Reference

CSacl Class
CSid Class
ACCESS_MASK
CSacl::RemoveAllAces

Other Resources

CSacl Members

CSacl::CSacl

The constructor.

```
CSacl( ) throw( );
CSacl(
   const ACL & rhs
) throw(...);
```

Parameters

rhs

An existing **ACL** (access-control list) structure.

Remarks

The **CSacl** object can be optionally created using an existing **ACL** structure. Ensure that this parameter is a system access-control list (SACL) and not a discretionary access-control list (DACL). In debug builds, if a DACL is supplied an assertion will occur. In release builds any entries from a DACL are ignored.

See Also

Reference

CSacl Class

CSacl::~CSacl

Other Resources

CSacl Members

CSacl::~CSacl

The destructor.

```
~CSacl( ) throw( );
```

Remarks

The destructor frees any resources acquired by the object, including all access-control entries (ACEs).

See Also

Reference

CSacl Class

CSacl::CSacl

Other Resources

CSacl Members

CSacl::AddAuditAce

Adds an audit access-control entry (ACE) to the **CSacl** object.

```
bool AddAuditAce(
   const CSid & rSid,
   ACCESS_MASK AccessMask,
   bool bSuccess,
   bool bFailure,
   BYTE AceFlags = 0
) throw(...);
bool AddAuditAce(
   const CSid & rSid,
   ACCESS_MASK AccessMask,
   bool bSuccess,
   bool bFailure,
   BYTE AceFlags,
   const GUID * pObjectType,
   const GUID * pInheritedObjectType
) throw(...);
```

Parameters

rSid

The CSid object.

AccessMask

Specifies the mask of access rights to be audited for the specified **CSid** object.

bSuccess

Specifies whether allowed access attempts are to be audited. Set this flag to true to enable auditing; otherwise, set it to false.

bFailure

Specifies whether denied access attempts are to be audited. Set this flag to true to enable auditing; otherwise, set it to false.

AceFlags

A set of bit flags that control ACE inheritance.

pObjectType

The object type.

pInheritedObjectType

The inherited object type.

Return Value

Returns **true** if the ACE is added to the **CSacl** object, **false** on failure.

Remarks

A **CSacl** object contains access-control entries (ACEs) that specify the types of access attempts that generate audit records in the security event log. This method adds such an ACE to the **CSacl** object. The second form of **AddAuditAce** is only available on Windows 2000 and later.

See ACE_HEADER for a description of the various flags which can be set in the *AceFlags* parameter.

See Also

Reference

CSacl Class
CSid Class
ACCESS_MASK
CSacl::RemoveAllAces

Other Resources

CSacl Members

CSacl::GetAceCount

Returns the number of access-control entries (ACEs) in the **CSacl** object.

```
UINT GetAceCount( ) const throw( );
```

Return Value

Returns the number of ACEs contained in the **CSacl** object.

See Also

Reference

CSacl Class

Other Resources

CSacl Members

CSacl::RemoveAce

Removes a specific ACE (access-control entry) from the **CSacl** object.

```
void RemoveAce(
   UINT nIndex
) throw( );
```

Parameters

nIndex

Index to the ACE entry to remove.

Remarks

This method is derived from CAtlArray::RemoveAt.

See Also

Reference

CSacl Class

CSacl::RemoveAllAces

Other Resources

CSacl Members

CSacl::RemoveAllAces

Removes all of the access-control entries (ACEs) contained in the **CSacl** object.

```
void RemoveAllAces( ) throw( );
```

Remarks

Removes every **ACE** structure (if any) in the **CSacl** object.

See Also

Reference

CSacl Class

CSacl::RemoveAce

Other Resources

CSacl Members

CSacl Operators

For information about the operators in **CSacl**, see CSacl Members.

CSacl::operator =

Assignment operator.

```
CSacl & operator =(
   const ACL & rhs
) throw(...);
```

Parameters

rhs

The **ACL** (access-control list) to assign to the existing object.

Return Value

Returns a reference to the updated **CSacl** object. Ensure that the **ACL** parameter is actually a system access-control list (SACL) and not a discretionary access-control list (DACL). In debug builds an assertion will occur, and in release builds the **ACL** parameter will be ignored.

See Also

Reference

CSacl Class

Other Resources

CSacl Members

CSecurityAttributes Class

This class is a thin wrapper for the security attributes structure.

```
class CSecurityAttributes : public SECURITY_ATTRIBUTES
```

Remarks

The **SECURITY_ATTRIBUTES** structure contains a security descriptor used for the creation of an object and specifies whether the handle retrieved by specifying this structure is inheritable.

For an introduction to the access control model in Windows, see Access Control in the *Platform SDK*.

Requirements

Header: atlsecurity.h

See Also

Tasks

ATLSecurity Sample: Demonstrates ATL Security Classes

Reference

SECURITY_ATTRIBUTES

security descriptor

Other Resources

CSecurityAttributes Members

ATL Class Overview

Security Global Functions

CSecurityAttributes Members

Methods

CSecurityAttributes	The constructor.
Set	Call this method to set the attributes of the **CSecurityAttributes** object.

See Also

Reference

CSecurityAttributes Class

CSecurityAttributes Methods

For information about the methods in **CSecurityAttributes**, see CSecurityAttributes Members.

CSecurityAttributes::CSecurityAttributes

The constructor.

```
CSecurityAttributes( ) throw( );
explicit CSecurityAttributes(
   const CSecurityDesc & rSecurityDescriptor,
   bool bInheritHandle = false
) throw(...);
```

Parameters

rSecurityDescriptor

Reference to a security descriptor.

bInheritHandle

Specifies whether the returned handle is inherited when a new process is created. If this member is true, the new process inherits the handle.

See Also

Reference

CSecurityAttributes Class

CSecurityAttributes::Set

Other Resources

CSecurityAttributes Members

CSecurityAttributes::Set

Call this method to set the attributes of the **CSecurityAttributes** object.

```
void Set(
   const CSecurityDesc & rSecurityDescriptor,
   bool bInheritHandle = false
) throw(...);
```

Parameters

rSecurityDescriptor

Reference to a security descriptor.

bInheritHandle

Specifies whether the returned handle is inherited when a new process is created. If this member is true, the new process inherits the handle.

Remarks

This method is used by the constructor to initialize the **CSecurityAttributes** object.

See Also

Reference

CSecurityAttributes Class

Other Resources

CSecurityAttributes Members

CSecurityDesc Class

This class is a wrapper for the **SECURITY_DESCRIPTOR** structure.

```
class CSecurityDesc
```

Remarks

The **SECURITY_DESCRIPTOR** structure contains the security information associated with an object. Applications use this structure to set and query an object's security status. See also AtlGetSecurityDescriptor.

Applications should not modify the **SECURITY_DESCRIPTOR** structure directly, and instead should use the class methods provided.

For an introduction to the access control model in Windows, see Access Control in the *Platform SDK*.

Requirements

Header: atlsecurity.h

See Also

Tasks

ATLSecurity Sample: Demonstrates ATL Security Classes

Reference

SECURITY_DESCRIPTOR

Other Resources

CSecurityDesc Members

ATL Class Overview

Security Global Functions

CSecurityDesc Members

Methods

CSecurityDesc	The constructor.
~CSecurityDesc	The destructor.
FromString	Converts a string-format security descriptor into a valid, functional security descriptor.
GetControl	Retrieves control information from the security descriptor.
GetDacl	Retrieves discretionary access-control list (DACL) information from the security descriptor.
GetGroup	Retrieves the primary group information from the security descriptor.
GetOwner	Retrieves owner informaton from the security descriptor.
GetPSECURITY_DESCRIPTOR	Returns a pointer to the **SECURITY_DESCRIPTOR** structure.
GetSacl	Retrieves system access-control list (SACL) information from the security descriptor.
IsDaclAutoInherited	Determines if the DACL is configured to support automatic propagation.
IsDaclDefaulted	Determines if the security descriptor is configured with a default DACL.
IsDaclPresent	Determines if the security descriptor contains a DACL.
IsDaclProtected	Determines if the DACL is configured to prevent modifications.
IsGroupDefaulted	Determines if the security descriptor's group security identifier (SID) was set by default.
IsOwnerDefaulted	Determines if the security descriptor's owner SID was set by default.
IsSaclAutoInherited	Determines if the SACL is configured to support automatic propagation.
IsSaclDefaulted	Determines if the security descriptor is configured with a default SACL.
IsSaclPresent	Determines if the security descriptor contains a SACL.
IsSaclProtected	Determines if the SACL is configured to prevent modifications.
IsSelfRelative	Determines if the security descriptor is in self-relative format.
MakeAbsolute	Call this method to convert the security descriptor to absolute format.
MakeSelfRelative	Call this method to convert the security descriptor to self-relative format.
SetControl	Sets the control bits of a security descriptor.
SetDacl	Sets information in a DACL. If a DACL is already present in the security descriptor, it is replaced.

SetGroup	Sets the primary group information of an absolute format security descriptor, replacing any primary group information already present.
SetOwner	Sets the owner information of an absolute format security descriptor, replacing any owner information already present.
SetSacl	Sets information in a SACL. If a SACL is already present in the security descriptor, it is replaced.
ToString	Converts a security descriptor to a string format.

Operators

operator =	Assignment operator.
operator const SECURITY_DESCRIPTOR *	Returns a pointer to the **SECURITY_DESCRIPTOR** structure.

See Also

Reference

CSecurityDesc Class

CSecurityDesc Methods

For information about the methods in **CSecurityDesc**, see CSecurityDesc Members.

CSecurityDesc::CSecurityDesc

The constructor.

```
CSecurityDesc( ) throw( );
CSecurityDesc(
   const CSecurityDesc & rhs
) throw(...);
CSecurityDesc(
   const SECURITY_DESCRIPTOR & rhs
) throw(...);
```

Parameters

rhs

The **CSecurityDesc** object or **SECURITY_DESCRIPTOR** structure to assign to the new **CSecurityDesc** object.

Remarks

The **CSecurityDesc** object can optionally be created using a **SECURITY_DESCRIPTOR** structure or a previously defined **CSecurityDesc** object.

See Also

Reference

CSecurityDesc Class
CSecurityDesc::~CSecurityDesc
SECURITY_DESCRIPTOR

Other Resources

CSecurityDesc Members

CSecurityDesc::~CSecurityDesc

The destructor.

```
virtual ~CSecurityDesc( ) throw( );
```

Remarks

The destructor frees all allocated resources.

See Also

Reference

CSecurityDesc Class

CSecurityDesc::CSecurityDesc

Other Resources

CSecurityDesc Members

CSecurityDesc::FromString

Converts a string-format security descriptor into a valid, functional security descriptor.

```
bool FromString(
   LPCTSTR pstr
) throw(...);
```

Parameters

pstr

Pointer to a null-terminated string containing the string-format security descriptor to convert.

Return Value

Returns true on success, false on failure.

Remarks

The string can be created using CSecurityDesc::ToString. Converting the security descriptor into a string makes it easier to store and transmit.

This method is only available with Windows 2000 and later as it calls ConvertStringSecurityDescriptorToSecurityDescriptor.

See Also

Reference

CSecurityDesc Class
ConvertSecurityDescriptorToStringSecurityDescriptor
SECURITY_DESCRIPTOR
CSecurityDesc::ToString

Other Resources

CSecurityDesc Members

CSecurityDesc::GetControl

Retrieves control information from the security descriptor.

```
bool GetControl(
   SECURITY_DESCRIPTOR_CONTROL * psdc
) const throw( );
```

Parameters

psdc

Pointer to a **SECURITY_DESCRIPTOR_CONTROL** structure that receives the security descriptor's control information.

Return Value

Returns true if the method succeeds, false if it fails.

Remarks

This method is only meaningful when using Windows 2000 or later, as it calls GetSecurityDescriptorControl.

See Also

Reference

CSecurityDesc Class
CSecurityDesc::GetDacl
CSecurityDesc::GetGroup
CSecurityDesc::GetOwner
CSecurityDesc::GetSacl
CSecurityDesc::SetControl

Other Resources

CSecurityDesc Members

CSecurityDesc::GetDacl

Retrieves discretionary access-control list (DACL) information from the security descriptor.

```
bool GetDacl(
   CDacl * pDacl,
   bool * pbPresent = NULL,
   bool * pbDefaulted = NULL
) const throw(...);
```

Parameters

pDacl

Pointer to an **CDacl** structure in which to store a copy of the security descriptor's DACL. If a discretionary **ACL** exists, the method sets *pDacl* to the address of the security descriptor's discretionary **ACL**. If a discretionary **ACL** does not exist, no value is stored.

pbPresent

Pointer to a value that indicates the presence of a discretionary **ACL** in the specified security descriptor. If the security descriptor contains a discretionary **ACL**, this parameter is set to true. If the security descriptor does not contain a discretionary **ACL**, this parameter is set to false.

pbDefaulted

Pointer to a flag set to the value of the SE_DACL_DEFAULTED flag in the **SECURITY_DESCRIPTOR_CONTROL** structure if a discretionary **ACL** exists for the security descriptor. If this flag is true, the discretionary **ACL** was retrieved by a default mechanism; if false, the discretionary **ACL** was explicitly specified by a user.

Return Value

Returns true if the method succeeds, false if it fails.

See Also

Reference

CSecurityDesc Class
GetSecurityDescriptorDacl
CSecurityDesc::GetControl
CSecurityDesc::GetGroup
CSecurityDesc::GetOwner
CSecurityDesc::GetSacl
CSecurityDesc::SetDacl

Other Resources

CSecurityDesc Members

CSecurityDesc::GetGroup

Retrieves the primary group information from the security descriptor.

```
bool GetGroup(
   CSid * pSid,
   bool * pbDefaulted = NULL
) const throw(...);
```

Parameters

pSid

Pointer to a CSid (security identifier) that receives a copy of the group stored in the CDacl.

pbDefaulted

Pointer to a flag set to the value of the SE_GROUP_DEFAULTED flag in the **SECURITY_DESCRIPTOR_CONTROL** structure when the method returns.

Return Value

Returns true if the method succeeds, false if it fails.

See Also

Reference

CSecurityDesc Class
GetSecurityDescriptorGroup
CSecurityDesc::GetControl
CSecurityDesc::GetDacl
CSecurityDesc::GetOwner
CSecurityDesc::GetSacl
CSecurityDesc::SetGroup

Other Resources

CSecurityDesc Members

CSecurityDesc::GetOwner

Retrieves owner informaton from the security descriptor.

```
bool GetOwner(
   CSid * pSid,
   bool * pbDefaulted = NULL
) const throw(...);
```

Parameters

pSid

Pointer to a CSid (security identifier) that receives a copy of the group stored in the CDacl.

pbDefaulted

Pointer to a flag set to the value of the SE_OWNER_DEFAULTED flag in the **SECURITY_DESCRIPTOR_CONTROL** structure when the method returns.

Return Value

Returns true if the method succeeds, false if it fails.

See Also

Reference

CSecurityDesc Class
GetSecurityDescriptorOwner
CSecurityDesc::GetControl
CSecurityDesc::GetDacl
CSecurityDesc::GetGroup
CSecurityDesc::GetSacl
CSecurityDesc::SetOwner

Other Resources

CSecurityDesc Members

CSecurityDesc::GetPSECURITY_DESCRIPTOR

Returns a pointer to the **SECURITY_DESCRIPTOR** structure.

```
const SECURITY_DESCRIPTOR * GetPSECURITY_DESCRIPTOR( ) const throw( );
```

Return Value

Returns a pointer to the SECURITY_DESCRIPTOR structure.

See Also

Reference

CSecurityDesc Class

SECURITY_DESCRIPTOR

Other Resources

CSecurityDesc Members

CSecurityDesc::GetSacl

Retrieves system access-control list (SACL) information from the security descriptor.

```
bool GetSacl(
   CSacl * pSacl,
   bool * pbPresent = NULL,
   bool * pbDefaulted = NULL
) const throw(...);
```

Parameters

pSacl

Pointer to an **CSacl** structure in which to store a copy of the security descriptor's SACL. If a system **ACL** exists, the method sets *pSacl* to the address of the security descriptor's system **ACL**. If a system **ACL** does not exist, no value is stored.

pbPresent

Pointer to a flag the method sets to indicate the presence of a system **ACL** in the specified security descriptor. If the security descriptor contains a system **ACL**, this parameter is set to true. If the security descriptor does not contain a system **ACL**, this parameter is set to false.

pbDefaulted

Pointer to a flag set to the value of the SE_SACL_DEFAULTED flag in the **SECURITY_DESCRIPTOR_CONTROL** structure if a system **ACL** exists for the security descriptor.

Return Value

Returns true if the method succeeds, false if it fails.

See Also

Reference

CSecurityDesc Class
GetSecurityDescriptorSacl
CSecurityDesc::GetControl
CSecurityDesc::GetDacl
CSecurityDesc::GetGroup
CSecurityDesc::GetOwner
CSecurityDesc::SetSacl

Other Resources

CSecurityDesc Members

CSecurityDesc::IsDaclAutoInherited

Determines if the discretionary access-control list (DACL) is configured to support automatic propagation.

```
bool IsDaclAutoInherited( ) const throw( );
```

Return Value

Returns true if the security descriptor contains a DACL which is set up to support automatic propagation of inheritable access-control entries (ACEs) to existing child objects. Returns false otherwise.

Remarks

The system sets this bit when it performs the automatic inheritance algorithm for the object and its existing child objects.

This bit is not set in security descriptors for Windows NT versions 4.0 and earlier, which do not support automatic propagation of inheritable ACEs. As a result, this method is only meaningful for Windows 2000 or later.

See Also

Reference

CSecurityDesc Class
SECURITY_DESCRIPTOR_CONTROL
CSecurityDesc::IsDaclDefaulted
CSecurityDesc::IsDaclPresent
CSecurityDesc::IsDaclProtected

Other Resources

CSecurityDesc Members

CSecurityDesc::IsDaclDefaulted

Determines if the security descriptor is configured with a default discretionary access-control list (DACL).

```
bool IsDaclDefaulted( ) const throw( );
```

Return Value

Returns true if the security descriptor contains a default DACL, false otherwise.

Remarks

This flag can affect how the system treats the DACL, with respect to access-control entry (ACE) inheritance. For example, if an object's creator does not specify a DACL, the object receives the default DACL from the creator's access token. The system ignores this flag if the SE_DACL_PRESENT flag is not set.

This flag is used to determine how the final DACL on the object is to be computed and is not stored physically in the security descriptor control of the securable object.

To set this flag, use the CSecurityDesc::SetDacl method.

See Also

Reference

CSecurityDesc Class
SECURITY_DESCRIPTOR_CONTROL
CSecurityDesc::IsDaclAutoInherited
CSecurityDesc::IsDaclPresent
CSecurityDesc::IsDaclProtected

Other Resources

CSecurityDesc Members

CSecurityDesc::IsDaclPresent

Determines if the security descriptor contains a discretionary access-control list (DACL).

```
bool IsDaclPresent( ) const throw( );
```

Return Value

Returns true if the security descriptor contains a DACL, false otherwise.

Remarks

If this flag is not set, or if this flag is set and the DACL is NULL, the security descriptor allows full access to everyone.

This flag is used to hold the security information specified by a caller until the security descriptor is associated with a securable object. Once the security descriptor is associated with a securable object, the SE_DACL_PRESENT flag is always set in the security descriptor control.

To set this flag, use the CSecurityDesc::SetDacl method.

See Also

Reference

CSecurityDesc Class
SECURITY_DESCRIPTOR_CONTROL
CSecurityDesc::IsDaclAutoInherited
CSecurityDesc::IsDaclDefaulted
CSecurityDesc::IsDaclProtected

Other Resources

CSecurityDesc Members

CSecurityDesc::IsDaclProtected

Determines if the discretionary access-control list (DACL) is configured to prevent modifications.

```
bool IsDaclProtected( ) const throw( );
```

Return Value

Returns true if the DACL is configured to prevent the security descriptor from being modified by inheritable access-control entries (ACEs). Returns false otherwise.

Remarks

To set this flag, use the CSecurityDesc::SetDacl method.

This method is only meaningful for Windows 2000 or later, as only Windows 2000 supports automatic propagation of inheritable ACEs.

See Also

Reference

CSecurityDesc Class
SECURITY_DESCRIPTOR_CONTROL
CSecurityDesc::IsDaclAutoInherited
CSecurityDesc::IsDaclDefaulted
CSecurityDesc::IsDaclPresent

Other Resources

CSecurityDesc Members

CSecurityDesc::IsGroupDefaulted

Determines if the security descriptor's group security identifier (SID) was set by default.

```
bool IsGroupDefaulted( ) const throw( );
```

Return Value

Returns true if a default mechanism, rather than the original provider of the security descriptor, provided the security descriptor's group SID. Returns false otherwise.

Remarks

To set this flag, use the CSecurityDesc::SetGroup method.

See Also

Reference

CSecurityDesc Class
SECURITY_DESCRIPTOR_CONTROL
CSecurityDesc::SetGroup
CSecurityDesc::GetGroup

Other Resources

CSecurityDesc Members

CSecurityDesc::IsOwnerDefaulted

Determines if the security descriptor's owner security identifier (SID) was set by default.

```
bool IsOwnerDefaulted( ) const throw( );
```

Return Value

Returns true if a default mechanism, rather than the original provider of the security descriptor, provided the security descriptor's owner SID. Returns false otherwise.

Remarks

To set this flag, use the CSecurityDesc::SetOwner method.

See Also

Reference

CSecurityDesc Class
SECURITY_DESCRIPTOR_CONTROL
CSecurityDesc::SetOwner
CSecurityDesc::GetOwner

Other Resources

CSecurityDesc Members

CSecurityDesc::IsSaclAutoInherited

Determines if the system access-control list (SACL) is configured to support automatic propagation.

```
bool IsSaclAutoInherited( ) const throw( );
```

Return Value

Returns true if the security descriptor contains a SACL which is set up to support automatic propagation of inheritable access-control entries (ACEs) to existing child objects. Returns false otherwise.

Remarks

The system sets this bit when it performs the automatic inheritance algorithm for the object and its existing child objects.

This bit is not set in security descriptors for Windows NT versions 4.0 and earlier, which do not support automatic propagation of inheritable ACEs. This method is therefore only meaningful for Windows 2000 or later

See Also

Reference

CSecurityDesc Class
SECURITY_DESCRIPTOR_CONTROL
CSecurityDesc::IsSaclDefaulted
CSecurityDesc::IsSaclPresent
CSecurityDesc::IsSaclProtected

Other Resources

CSecurityDesc Members

CSecurityDesc::IsSaclDefaulted

Determines if the security descriptor is configured with a default system access-control list (SACL).

```
bool IsSaclDefaulted( ) const throw( );
```

Return Value

Returns true if the security descriptor contains a default SACL, false otherwise.

Remarks

This flag can affect how the system treats the SACL, with respect to access-control entry (ACE) inheritance. The system ignores this flag if the SE_SACL_PRESENT flag is not set.

To set this flag, use the CSecurityDesc::SetSacl method.

See Also

Reference

CSecurityDesc Class
SECURITY_DESCRIPTOR_CONTROL
CSecurityDesc::IsSaclAutoInherited
CSecurityDesc::IsSaclPresent
CSecurityDesc::IsSaclProtected

Other Resources

CSecurityDesc Members

CSecurityDesc::IsSaclPresent

Determines if the security descriptor contains a system access-control list (SACL).

```
bool IsSaclPresent( ) const throw( );
```

Return Value

Returns true if the security descriptor contains a SACL, false otherwise.

Remarks

To set this flag, use the CSecurityDesc::SetSacl method.

See Also

Reference

CSecurityDesc Class
SECURITY_DESCRIPTOR_CONTROL
CSecurityDesc::IsSaclAutoInherited
CSecurityDesc::IsSaclDefaulted
CSecurityDesc::IsSaclProtected

Other Resources

CSecurityDesc Members

CSecurityDesc::IsSaclProtected

Determines if the system access-control list (SACL) is configured to prevent modifications.

```
bool IsSaclProtected( ) const throw( );
```

Return Value

Returns true if the SACL is configured to prevent the security descriptor from being modified by inheritable access-control entries (ACEs). Returns false otherwise.

Remarks

To set this flag, use the CSecurityDesc::SetSacl method.

This method is only meaningful for Windows 2000 or later, as only Windows 2000 supports automatic propagation of inheritable ACEs.

See Also

Reference

CSecurityDesc Class
SECURITY_DESCRIPTOR_CONTROL
CSecurityDesc::IsSaclAutoInherited
CSecurityDesc::IsSaclDefaulted
CSecurityDesc::IsSaclPresent

Other Resources

CSecurityDesc Members

CSecurityDesc::IsSelfRelative

Determines if the security descriptor is in self-relative format.

```
bool IsSelfRelative( ) const throw( );
```

Return Value

Returns true if the security descriptor is in self-relative format with all the security information in a contiguous block of memory. Returns false if the security descriptor is in absolute format. For more information, see Absolute and Self-Relative Security Descriptors.

See Also

Reference

CSecurityDesc Class
SECURITY_DESCRIPTOR_CONTROL
Absolute and Self-Relative Security Descriptors
CSecurityDesc::MakeSelfRelative
CSecurityDesc::MakeAbsolute

Other Resources

CSecurityDesc Members

CSecurityDesc::MakeAbsolute

Call this method to convert the security descriptor to absolute format.

```
bool MakeAbsolute( ) throw(...);
```

Return Value

Returns true if the method succeeds, false otherwise.

Remarks

A security descriptor in absolute format contains pointers to the information it contains, rather than the information itself. A security descriptor in self-relative format contains the information in a contiguous block of memory. In a self-relative security descriptor, a **SECURITY_DESCRIPTOR** structure always starts the information, but the security descriptor's other components can follow the structure in any order. Instead of using memory addresses, the components of the self-relative security descriptor are identified by offsets from the beginning of the security descriptor. This format is useful when a security descriptor must be stored on a disk or transmitted by means of a communications protocol. For more information, see Absolute and Self-Relative Security Descriptors.

See Also

Reference

CSecurityDesc Class
SECURITY_DESCRIPTOR_CONTROL
Absolute and Self-Relative Security Descriptors
CSecurityDesc::MakeSelfRelative
CSecurityDesc::IsSelfRelative
MakeAbsoluteSD

Other Resources

CSecurityDesc Members

CSecurityDesc::MakeSelfRelative

Call this method to convert the security descriptor to self-relative format.

```
bool MakeSelfRelative( ) throw(...);
```

Return Value

Returns true if the method succeeds, false otherwise.

Remarks

A security descriptor in absolute format contains pointers to the information it contains, rather than containing the information itself. A security descriptor in self-relative format contains the information in a contiguous block of memory. In a self-relative security descriptor, a **SECURITY_DESCRIPTOR** structure always starts the information, but the security descriptor's other components can follow the structure in any order. Instead of using memory addresses, the components of the security descriptor are identified by offsets from the beginning of the security descriptor. This format is useful when a security descriptor must be stored on a disk or transmitted by means of a communications protocol. For more information, see Absolute and Self-Relative Security Descriptors.

See Also

Reference

CSecurityDesc Class
SECURITY_DESCRIPTOR_CONTROL
Absolute and Self-Relative Security Descriptors
CSecurityDesc::MakeAbsolute
CSecurityDesc::IsSelfRelative
MakeSelfRelativeSD

Other Resources

CSecurityDesc Members

CSecurityDesc::SetControl

Sets the control bits of a security descriptor.

```
bool SetControl(
   SECURITY_DESCRIPTOR_CONTROL ControlBitsOfInterest,
   SECURITY_DESCRIPTOR_CONTROL ControlBitsToSet
) throw( );
```

Parameters

ControlBitsOfInterest

A **SECURITY_DESCRIPTOR_CONTROL** mask that indicates the control bits to set. For a list of the flags which can be set, see SetSecurityDescriptorControl.

ControlBitsToSet

A **SECURITY_DESCRIPTOR_CONTROL** mask that indicates the new values for the control bits specified by the *ControlBitsOfInterest* mask. This parameter can be a combination of the flags listed for the *ControlBitsOfInterest* parameter.

Return Value

Returns true on success, false on failure.

Remarks

This method is only available on Windows 2000 and later, as it calls SetSecurityDescriptorControl.

See Also

Reference

CSecurityDesc Class
CSecurityDesc::SetDacl
CSecurityDesc::SetGroup
CSecurityDesc::SetOwner
CSecurityDesc::SetSacl
CSecurityDesc::GetControl

Other Resources

CSecurityDesc Members

CSecurityDesc::SetDacl

Sets information in a discretionary access-control list (DACL). If a DACL is already present in the security descriptor, it is replaced.

```
inline void SetDacl(
   bool bPresent = true,
   bool bDefaulted = false
) throw(...);
inline void SetDacl(
   const CDacl & Dacl,
   bool bDefaulted = false
) throw(...);
```

Parameters

Dacl

Reference to a **CDacl** object specifying the DACL for the security descriptor. This parameter must not be NULL. To set a NULL DACL in the security descriptor, the first form of the method should be used with *bPresent* set to false.

bPresent

Specifies a flag indicating the presence of a DACL in the security descriptor. If this parameter is true, the method sets the SE_DACL_PRESENT flag in the **SECURITY_DESCRIPTOR_CONTROL** structure and uses the values in the *Dacl* and *bDefaulted* parameters. If it is false, the method clears the SE_DACL_PRESENT flag, and *bDefaulted* is ignored.

bDefaulted

Specifies a flag indicating the source of the DACL. If this flag is true, the DACL has been retrieved by some default mechanism. If false, the DACL has been explicitly specified by a user. The method stores this value in the SE_DACL_DEFAULTED flag of the **SECURITY_DESCRIPTOR_CONTROL** structure. If this parameter is not specified, the SE_DACL_DEFAULTED flag is cleared.

Return Value

Returns true on success, false on failure.

Remarks

There is an important difference between an empty and a nonexistent DACL. When a DACL is empty, it contains no access-control entries and no access rights have been explicitly granted. As a result, access to the object is implicitly denied. When an object has no DACL, on the other hand, no protection is assigned to the object, and any access request is granted.

See Also

Reference

CSecurityDesc Class
SetSecurityDescriptorDacl
CSecurityDesc::SetControl
CSecurityDesc::SetGroup
CSecurityDesc::SetOwner
CSecurityDesc::SetSacl
CSecurityDesc::GetDacl

Other Resources

CSecurityDesc Members

CSecurityDesc::SetGroup

Sets the primary group information of an absolute format security descriptor, replacing any primary group information already present.

```
bool SetGroup(
   const CSid & Sid,
   bool bDefaulted = false
) throw(...);
```

Parameters

Sid

Reference to a CSid object for the security descriptor's new primary group. This parameter must not be NULL. A security descriptor can be marked as not having a DACL or a SACL, but it must have a group and an owner, even it these are the NULL SID (which is a built-in SID with a special meaning).

bDefaulted

Indicates whether the primary group information was derived from a default mechanism. If this value is true, it is default information, and the method stores this value as the SE_GROUP_DEFAULTED flag in the **SECURITY_DESCRIPTOR_CONTROL** structure. If this parameter is zero, the SE_GROUP_DEFAULTED flag is cleared.

Return Value

Returns true on success, false on failure.

See Also

Reference

CSecurityDesc Class
SetSecurityDescriptorGroup
CSecurityDesc::SetControl
CSecurityDesc::SetDacl
CSecurityDesc::SetOwner
CSecurityDesc::SetSacl
CSecurityDesc::GetGroup

Other Resources

CSecurityDesc Members

CSecurityDesc::SetOwner

Sets the owner information of an absolute format security descriptor. It replaces any owner information already present.

```
bool SetOwner(
   const CSid & Sid,
   bool bDefaulted = false
) throw(...);
```

Parameters

Sid

The CSid object for the security descriptor's new primary owner. This parameter must not be NULL.

bDefaulted

Indicates whether the owner information is derived from a default mechanism. If this value is true, it is default information. The method stores this value as the SE_OWNER_DEFAULTED flag in the **SECURITY_DESCRIPTOR_CONTROL** structure. If this parameter is zero, the SE_OWNER_DEFAULTED flag is cleared.

Return Value

Returns true on success, false on failure.

See Also

Reference

CSecurityDesc Class
CSecurityDesc::SetControl
CSecurityDesc::SetGroup
CSecurityDesc::SetSacl
CSecurityDesc::GetOwner

Other Resources

CSecurityDesc Members

CSecurityDesc::SetSacl

Sets information in a system access-control list (SACL). If a SACL is already present in the security descriptor, it is replaced.

```
bool SetSacl(
   const CSacl & Sacl,
   bool bDefaulted = false
) throw(...);
```

Parameters

Sacl

Pointer to an **CSacl** object specifying the SACL for the security descriptor. This parameter must not be NULL, and must be a CSacl object. Unlike DACLs, there is no difference between NULL and an empty SACL, as SACL objects do not specify access rights, only auditing information.

bDefaulted

Specifies a flag indicating the source of the SACL. If this flag is true, the SACL has been retrieved by some default mechanism. If false, the SACL has been explicitly specified by a user. The method stores this value in the SE_SACL_DEFAULTED flag of the **SECURITY_DESCRIPTOR_CONTROL** structure. If this parameter is not specified, the SE_SACL_DEFAULTED flag is cleared.

Return Value

Returns true on success, false on failure.

See Also

Reference

CSecurityDesc Class
CSecurityDesc::SetControl
CSecurityDesc::SetGroup
CSecurityDesc::SetDacl
CSecurityDesc::SetOwner
CSecurityDesc::GetSacl

Other Resources

CSecurityDesc Members

CSecurityDesc::ToString

Converts a security descriptor to a string format.

```
bool ToString(
   CString * pstr,
   SECURITY_INFORMATION si = OWNER_SECURITY_INFORMATION |
      GROUP_SECURITY_INFORMATION | DACL_SECURITY_INFORMATION |
      SACL_SECURITY_INFORMATION
) const throw(...);
```

Parameters

pstr

Pointer to a null-terminated string which will receive the string-format security descriptor.

si

Specifies a combination of SECURITY_INFORMATION bit flags to indicate the components of the security descriptor to include in the output string.

Return Value

Returns true on success, false on failure.

Remarks

Once the security descriptor is in string format, it can more easily be stored or transmitted. Use the **CSecurityDesc::FromString** method to convert the string back into a security descriptor.

The *si* parameter can contain the following SECURITY_INFORMATION flags:

Value	Meaning
OWNER_SECURITY_INFORMATION	Include the owner.
GROUP_SECURITY_INFORMATION	Include the primary group.
DACL_SECURITY_INFORMATION	Include the DACL.
SACL_SECURITY_INFORMATION	Include the SACL.

If the DACL is NULL and the SE_DACL_PRESENT control bit is set in the input security descriptor, the method fails.

If the DACL is NULL and the SE_DACL_PRESENT control bit is not set in the input security descriptor, the resulting security descriptor string does not have a D: component. See Security Descriptor String Format for more details.

This method is only available with Windows 2000 and later, as it calls ConvertStringSecurityDescriptorToSecurityDescriptor.

See Also

Reference

CSecurityDesc Class
SECURITY_DESCRIPTOR
CSecurityDesc::FromString

Other Resources

CSecurityDesc Members

CSecurityDesc Operators

For information about the operators in **CSecurityDesc**, see CSecurityDesc Members.

CSecurityDesc::operator =

Assignment operator.

```
CSecurityDesc & operator =(
   const SECURITY_DESCRIPTOR & rhs
) throw(...);
CSecurityDesc & operator =(
   const CSecurityDesc & rhs
) throw(...);
```

Parameters

rhs

The **SECURITY_DESCRIPTOR** structure or **CSecurityDesc** object to assign to the **CSecurityDesc** object.

Return Value

Returns the updated **CSecurityDesc** object.

See Also

Reference

CSecurityDesc Class

SECURITY_DESCRIPTOR

Other Resources

CSecurityDesc Members

CSecurityDesc::operator const SECURITY_DESCRIPTOR *

Casts a value to a pointer to the **SECURITY_DESCRIPTOR** structure.

```
operator const SECURITY_DESCRIPTOR *( ) const throw( );
```

See Also

Reference

CSecurityDesc Class

SECURITY_DESCRIPTOR

Other Resources

CSecurityDesc Members

CSid Class

This class is a wrapper for a **SID** (security identifier) structure.

```
class CSid
```

Remarks

The **SID** structure is a variable-length structure used to uniquely identify users or groups.

Applications should not modify the **SID** structure directly, but instead use the methods provided in this wrapper class. See also AtlGetOwnerSid, AtlSetGroupSid, AtlGetGroupSid, and AtlSetOwnerSid.

For an introduction to the access control model in Windows, see Access Control in the *Platform SDK*.

Requirements

Header: atlsecurity.h

See Also

Tasks

ATLSecurity Sample: Demonstrates ATL Security Classes

Other Resources

CSid Members

ATL Class Overview

Security Global Functions

ATL Operators Alphabetical Reference

CSid Members

Methods

AccountName	Returns the name of the account associated with the **CSid** object.
CSid	The constructor.
~CSid	The destructor.
Domain	Returns the name of the domain associated with the **CSid** object.
EqualPrefix	Tests **SID** (security identifier) prefixes for equality.
GetLength	Returns the length of the **CSid** object.
GetPSID	Returns a pointer to a **SID** structure.
GetPSID_IDENTIFIER_AUTHORITY	Returns a pointer to the **SID_IDENTIFIER_AUTHORITY** structure.
GetSubAuthority	Returns a specified subauthority in a **SID** structure.
GetSubAuthorityCount	Returns the subauthority count.
IsValid	Tests the **CSid** object for validity.
LoadAccount	Updates the **CSid** object given the account name and domain, or an existing **SID** structure.
Sid	Returns the ID string.
SidNameUse	Returns a description of the state of the **CSid** object.

Operators

operator =	Assignment operator.
operator const SID *	Casts a **CSid** object to a pointer to a **SID** structure.

Global Operators

operator ==	Tests two security descriptor objects for equality
operator !=	Tests two security descriptor objects for inequality
operator <	Compares relative value of two security descriptor objects.
operator >	Compares relative value of two security descriptor objects.
operator <=	Compares relative value of two security descriptor objects.
operator >=	Compares relative value of two security descriptor objects.

Typedefs

CSidArray	An array of **CSid** objects.

See Also

Reference

CSid Class

CSid Methods

For information about the methods in **CSid**, see CSid Members.

CSid::AccountName

Returns the name of the account associated with the **CSid** object.

```
LPCTSTR AccountName( ) const throw(...);
```

Return Value

Returns the **LPCTSTR** pointing to the name of the account.

Remarks

This method attempts to find a name for the specified **SID** (security identifier). For full details, see LookupAccountSid.

If no account name for the **SID** can be found, **AccountName** returns an empty string. This can occur if a network timeout prevents this method from finding the name. It also occurs for security identifiers with no corresponding account name, such as a logon **SID** that identifies a logon session.

See Also

Reference

CSid Class
CSid::Domain
LookupAccountSid

Other Resources

CSid Members

CSid::CSid

The constructor.

```
CSid( ) throw( );
CSid(
   const SID & rhs
) throw(...);
CSid(
   const CSid & rhs
) throw(...);
CSid(
   const SID_IDENTIFIER_AUTHORITY & IdentifierAuthority,
   BYTE nSubAuthorityCount,
   ...
) throw(...);
explicit CSid(
   LPCTSTR pszAccountName,
   LPCTSTR pszSystem = NULL
) throw(...);
explicit CSid(
   const SID * pSid,
   LPCTSTR pszSystem = NULL
) throw(...);
```

Parameters

rhs

An existing **CSid** object or **SID** (security identifier) structure.

IdentifierAuthority

The authority.

nSubAuthorityCount

The subauthority count.

pszAccountName

The account name.

pszSystem

The system name. This string can be the name of a remote computer. If this string is NULL, the local system is used instead.

pSid

A pointer to a **SID** structure.

Remarks

The constructor initializes the **CSid** object, setting an internal data member to *SidTypeInvalid*, or by copying the settings from an existing **CSid**, **SID**, or existing account.

If initialization fails, the constructor will throw a CAtlException Class.

See Also

Reference

CSid Class

Other Resources

CSid Members

CSid::~CSid

The destructor.

```
virtual ~CSid( ) throw( );
```

Remarks

The destructor frees any resources acquired by the object.

See Also

Reference

CSid Class

Other Resources

CSid Members

CSid::Domain

Returns the name of the domain associated with the **CSid** object.

```
LPCTSTR Domain( ) const throw(...);
```

Return Value

Returns the **LPCTSTR** pointing to the domain.

Remarks

This method attempts to find a name for the specified **SID** (security identifier). For full details, see LookupAccountSid.

If no account name for the **SID** can be found, **Domain** returns the domain as an empty string. This can occur if a network timeout prevents this method from finding the name. It also occurs for security identifiers with no corresponding account name, such as a logon **SID** that identifies a logon session.

See Also

Reference

CSid Class
CSid::AccountName
LookupAccountSid

Other Resources

CSid Members

CSid::EqualPrefix

Tests **SID** (security identifier) prefixes for equality.

```
bool EqualPrefix(
   const SID & rhs
) const throw();
bool EqualPrefix(
   const CSid & rhs
) const throw();
```

Parameters

rhs

The **SID** (security identifier) structure or **CSid** object to compare.

Return Value

Returns **true** on success, **false** on failure.

Remarks

See EqualPrefixSid in the *Platform SDK* for more details.

See Also

Reference

CSid Class

Other Resources

CSid Members

CSid::GetLength

Returns the length of the **CSid** object.

```
UINT GetLength( ) const throw( );
```

Return Value

Returns the length in bytes of the **CSid** object.

Remarks

If the **CSid** structure is not valid, the return value is undefined. Before calling **GetLength**, use the CSid::IsValid member function to verify that **CSid** is valid.

Note
Under debug builds the function will cause an ASSERT if the **CSid** object is not valid.

See Also

Reference

CSid Class

Other Resources

CSid Members

CSid::GetPSID

Returns a pointer to a **SID** (security identifier) structure.

```
const SID * GetPSID( ) const throw(...);
```

Return Value

Returns the address of the **CSid** object's underlying **SID** structure.

See Also

Reference

CSid Class

CSid::GetPSID_IDENTIFIER_AUTHORITY

Other Resources

CSid Members

CSid::GetPSID_IDENTIFIER_AUTHORITY

Returns a pointer to the **SID_IDENTIFIER_AUTHORITY** structure.

```
const SID_IDENTIFIER_AUTHORITY * GetPSID_IDENTIFIER_AUTHORITY( ) const throw();
```

Return Value

If the method succeeds, it returns the address of the **SID_IDENTIFIER_AUTHORITY** structure. If it fails, the return value is undefined. Failure may occur if the **CSid** object is not valid, in which case the CSid::IsValid method returns **false**. The function **GetLastError** can be called for extended error information.

Note
Under debug builds the function will cause an ASSERT if the **CSid** object is not valid.

See Also

Reference

CSid Class

CSid::GetSubAuthority

Other Resources

CSid Members

CSid::GetSubAuthority

Returns a specified subauthority in a **SID** (security identifier) structure.

```
DWORD GetSubAuthority(
   DWORD nSubAuthority
) const throw( );
```

Parameters

nSubAuthority

The subauthority.

Return Value

Returns the subauthority referenced by *nSubAuthority*. The subauthority value is a relative identifier (RID).

Remarks

The *nSubAuthority* parameter specifies an index value identifying the subauthority array element the method will return. The method performs no validation tests on this value. An application can call CSid::GetSubAuthorityCount to discover the range of acceptable values.

Note
Under debug builds the function will cause an ASSERT if the **CSid** object is not valid.

See Also

Reference

CSid Class

CSid::GetPSID_IDENTIFIER_AUTHORITY

Other Resources

CSid Members

CSid::GetSubAuthorityCount

Returns the subauthority count.

```
UCHAR GetSubAuthorityCount( ) const throw( );
```

Return Value

If the method succeeds, the return value is the subauthority count.

If the method fails, the return value is undefined. The method fails if the **CSid** object is invalid. To get extended error information, call **GetLastError**.

Note
Under debug builds the function will cause an ASSERT if the **CSid** object is not valid.

See Also

Reference

CSid Class

CSid::GetSubAuthority

Other Resources

CSid Members

CSid::IsValid

Tests the **CSid** object for validity.

```
bool IsValid( ) const throw();
```

Return Value

Returns **true** if the **CSid** object is valid, **false** if not. There is no extended error information for this method; do not call **GetLastError**.

Remarks

The **IsValid** method validates the **CSid** object by verifying that the revision number is within a known range and that the number of subauthorities is less than the maximum.

See Also

Reference

CSid Class

Other Resources

CSid Members

CSid::LoadAccount

Updates the **CSid** object given the account name and domain, or an existing **SID** (security identifier) structure.

```
bool LoadAccount(
   LPCTSTR pszAccountName,
   LPCTSTR pszSystem = NULL
) throw(...);
bool LoadAccount(
   const SID * pSid,
   LPCTSTR pszSystem = NULL
) throw(...);
```

Parameters

pszAccountName

The account name.

pszSystem

The system name. This string can be the name of a remote computer. If this string is NULL, the local system is used instead.

pSid

A pointer to a **SID** structure.

Return Value

Returns **true** on success, **false** on failure. To get extended error information, call **GetLastError**.

Remarks

LoadAccount attempts to find a security identifier for the specified name. See LookupAccountSid for more details.

See Also

Reference

CSid Class

Other Resources

CSid Members

CSid::Sid

Returns the **SID** (security identifier) structure as a string.

```
LPCTSTR Sid( ) const throw(...);
```

Return Value

Returns the **SID** structure as a string in a format suitable for display, storage, or transmission. Equivalent to ConvertSidToStringSid, although this function is only available on Windows 2000 or later and so is emulated for earlier operating systems.

See Also

Reference

CSid Class

Other Resources

CSid Members

CSid::SidNameUse

Returns a description of the state of the **CSid** object.

```
SID_NAME_USE SidNameUse( ) const throw( );
```

Return Value

Returns the value of the data member that stores a value describing the state of the **CSid** object.

Value	Description
SidTypeUser	Indicates a user **SID** (security identifier).
SidTypeGroup	Indicates a group **SID**.
SidTypeDomain	Indicates a domain **SID**.
SidTypeAlias	Indicates an alias **SID**.
SidTypeWellKnownGroup	Indicates a **SID** for a well-known group.
SidTypeDeletedAccount	Indicates a **SID** for a deleted account.
SidTypeInvalid	Indicates an invalid **SID**.
SidTypeUnknown	Indicates an unknown **SID** type.
SidTypeComputer	Indicates a **SID** for a computer.

Remarks

Call CSid::LoadAccount to update the **CSid** object before calling **SidNameUse** to return its state. **SidNameUse** does not change the state of the object (by calling to **LookupAccountName** or **LookupAccountSid**), but only returns the current state.

See Also

Reference

CSid Class

Other Resources

CSid Members

CSid Operators

For information about the operators in **CSid**, see CSid Members.

CSid::operator =

Assignment operator.

```
CSid & operator =(
   const CSid & rhs
) throw(...);
CSid & operator =(
   const SID & rhs
) throw(...);
```

Parameters

rhs

The **SID** (security identifier) or **CSid** to assign to the **CSid** object.

Return Value

Returns a reference to the updated **CSid** object.

See Also

Reference

CSid Class

Other Resources

CSid Members

CSid::operator ==

Tests two security descriptor objects for equality.

```
bool operator==(
   const CSid &lhs,
   const CSid &rhs
) throw(   );
```

Parameters

lhs

The **SID** (security identifier) or **CSid** that appears on the left side of the == operator.

rhs

The **SID** (security identifier) or **CSid** that appears on the right side of the == operator.

Return Value

true if the security descriptors are equal, otherwise **false**.

See Also

Reference

CSid Class

Other Resources

CSid Members

CSid::operator !=

Tests two security descriptor objects for inequality.

```
bool operator!=(
   const CSid &lhs,
   const CSid &rhs
) throw(   );
```

Parameters

lhs

The **SID** (security identifier) or **CSid** that appears on the left side of the != operator.

rhs

The **SID** (security identifier) or **CSid** that appears on the right side of the != operator.

Return Value

true if the security descriptors are not equal, otherwise **false**.

See Also

Reference

CSid Class

Other Resources

CSid Members

CSid::operator <

Compares relative value of two security descriptor objects.

```
bool operator<(
   const CSid &lhs,
   const CSid &rhs
) throw(   );
```

Parameters

lhs

The **SID** (security identifier) or **CSid** that appears on the left side of the != operator.

rhs

The **SID** (security identifier) or **CSid** that appears on the right side of the != operator.

Return Value

true if *lhs* is less than *rhs*, otherwise **false**.

See Also

Reference

CSid Class

Other Resources

CSid Members

CSid::operator <=

Compares relative value of two security descriptor objects.

```
bool operator<(
   const CSid &lhs,
   const CSid &rhs
) throw( );
```

Parameters

lhs

The **SID** (security identifier) or **CSid** that appears on the left side of the != operator.

rhs

The **SID** (security identifier) or **CSid** that appears on the right side of the != operator.

Return Value

true if *lhs* is less than or equal to *rhs*, otherwise **false**.

See Also

Reference

CSid Class

Other Resources

CSid Members

CSid::operator >

Compares relative value of two security descriptor objects.

```
bool operator<(
   const CSid &lhs,
   const CSid &rhs
) throw(   );
```

Parameters

lhs

The **SID** (security identifier) or **CSid** that appears on the left side of the != operator.

rhs

The **SID** (security identifier) or **CSid** that appears on the right side of the != operator.

Return Value

true if *lhs* is greater than *rhs*, otherwise **false**.

See Also

Reference

CSid Class

Other Resources

CSid Members

CSid::operator >=

Compares relative value of two security descriptor objects.

```
bool operator<(
   const CSid &lhs,
   const CSid &rhs
) throw( )
);
```

Parameters

lhs

The **SID** (security identifier) or **CSid** that appears on the left side of the != operator.

rhs

The **SID** (security identifier) or **CSid** that appears on the right side of the != operator.

Return Value

true if *lhs* is greater than or equal to *rhs*, otherwise **false**.

See Also

Reference

CSid Class

Other Resources

CSid Members

CSid::operator const SID *

Casts a **CSid** object to a pointer to a **SID** (security identifier) structure.

```
operator const SID *( ) const throw(...);
```

Remarks

Returns the address of the **SID** structure.

See Also

Reference

CSid Class

CSid::GetPSID

Other Resources

CSid Members

CSid Typedefs

For information about the typedefs in **CSid**, see CSid Members.

CSid::CSidArray

An array of CSid objects.

```
typedef CAtlArray<CSid> CSidArray;
```

Remarks

This typedef specifies the array type that can be used to retrieve security identifiers from an ACL (access-control list). See CAcl::GetAclEntries.

See Also

Reference

CSid Class

Other Resources

CSid Members

CSimpleArray Class

This class provides methods for managing a simple array.

```
template <
   class T,
   class TEqual = CSimpleArrayEqualHelper< T >
>
class CSimpleArray
```

Parameters

T

The type of data to store in the array.

TEqual

A trait object, defining the equality test for elements of type *T*.

Remarks

CSimpleArray provides methods for creating and managing a simple array, of any given type *T*.

The parameter *TEqual* provides a means of defining an equality function for two elements of type *T*. By creating a class similar to CSimpleArrayEqualHelper, it is possible to alter the behavior of the equality test for any given array. For example, when dealing with an array of pointers, it may be useful to define the equality as depending on the values the pointers reference. The default implementation utilizes **operator=()**.

Both **CSimpleArray** and CSimpleMap are designed for a small number of elements. CAtlArray and CAtlMap should be used when the array contains a large number of elements.

Requirements

Header: atlsimpcoll.h

Example

```
// Create an array of integers
   CSimpleArray<int> iArray;

// Create an array of char pointers
// and use a new equality function
   CSimpleArray<char *, MyEqualityEqualHelper<char *> > cMyArray;
```

See Also

Other Resources

CSimpleArray Members

ATL Class Overview

CSimpleArray Members

Methods

~CSimpleArray	The destructor for the simple array.
Add	Adds a new element to the array.
CSimpleArray	The constructor for the simple array.
Find	Finds an element in the array.
GetData	Returns a pointer to the data stored in the array.
GetSize	Returns the number of elements stored in the array.
Remove	Removes a given element from the array.
RemoveAll	Removes all elements from the array.
RemoveAt	Removes the specified element from the array.
SetAtIndex	Sets the specified element in the array.

Operators

operator []	Retrieves an element from the array.
operator =	Assignment operator.

See Also

Reference

CSimpleArray Class

CSimpleArray Methods

For information about the methods in **CSimpleArray**, see CSimpleArray Members.

CSimpleArray::Add

Adds a new element to the array.

```
BOOL Add(
   const T& t
);
```

Parameters

t

The element to add to the array.

Return Value

Returns TRUE if the element is successfully added to the array, FALSE otherwise.

Example

```
// Create an array of integers
// and add some elements
   CSimpleArray<int> iMyArray;
   for (int i=0; i<10; i++)
      iMyArray.Add(i);
```

See Also

Reference

CSimpleArray Class

CSimpleArray::SetAtIndex

Other Resources

CSimpleArray Members

CSimpleArray::CSimpleArray

The constructor for the array object.

```
CSimpleArray(
   const CSimpleArray< T, TEqual >& src
);
CSimpleArray( );
```

Parameters

src

An existing **CSimpleArray** object.

Remarks

Initializes the data members, creating a new empty **CSimpleArray** object, or a copy of an existing **CSimpleArray** object.

See Also

Reference

CSimpleArray Class

CSimpleArray::~CSimpleArray

Other Resources

CSimpleArray Members

CSimpleArray::~CSimpleArray

The destructor.

```
~CSimpleArray( );
```

Remarks

Frees all allocated resources.

See Also

Reference

CSimpleArray Class

CSimpleArray::CSimpleArray

Other Resources

CSimpleArray Members

CSimpleArray::Find

Finds an element in the array.

```
int Find(
   const T& t
) const;
```

Parameters

t

The element for which to search.

Return Value

Returns the index of the found element, or -1 if the element is not found.

Example

```
// Create an array of floats
// and search for a particular element

   CSimpleArray<float> fMyArray;
   for (int i=0; i<10; i++)
      fMyArray.Add((float)i*100);
   int e = fMyArray.Find(200);
   if (e == -1)
      printf("Could not find element\n");
   else
      printf("Found the element at location %d\n",e);
```

See Also

Reference

CSimpleArray Class

Other Resources

CSimpleArray Members

CSimpleArray::GetData

Returns a pointer to the data stored in the array.

```
T* GetData( ) const;
```

Return Value

Returns a pointer to the data in the array.

See Also

Reference

CSimpleArray Class

Other Resources

CSimpleArray Members

CSimpleArray::GetSize

Returns the number of elements stored in the array.

```
int GetSize( ) const;
```

Return Value

Returns the number of elements stored in the array.

See Also

Reference

CSimpleArray Class

Other Resources

CSimpleArray Members

CSimpleArray::Remove

Removes a given element from the array.

```
BOOL Remove(
   const T& t
);
```

Parameters

t

The element to remove from the array.

Return Value

Returns TRUE if the element is found and removed, FALSE otherwise.

Remarks

When an element is removed, the remaining elements in the array are renumbered to fill the empty space.

See Also

Reference

CSimpleArray Class
CSimpleArray::RemoveAll
CSimpleArray::RemoveAt

Other Resources

CSimpleArray Members

CSimpleArray::RemoveAll

Removes all elements from the array.

```
void RemoveAll( );
```

Remarks

Removes all elements currently stored in the array.

See Also

Reference

CSimpleArray Class
CSimpleArray::Remove
CSimpleArray::RemoveAt

Other Resources

CSimpleArray Members

CSimpleArray::RemoveAt

Removes the specified element from the array.

```
BOOL RemoveAt(
   int nIndex
);
```

Parameters

nIndex

Index pointing to the element to remove.

Return Value

Returns TRUE if the element was removed, FALSE if the index was invalid.

Remarks

When an element is removed, the remaining elements in the array are renumbered to fill the empty space.

See Also

Reference

CSimpleArray Class
CSimpleArray::RemoveAll
CSimpleArray::Remove

Other Resources

CSimpleArray Members

CSimpleArray::SetAtIndex

Set the specified element in the array.

```
BOOL SetAtIndex(
   int nIndex,
   const T& t
);
```

Parameters

nIndex

The index of the element to change.

t

The value to assign to the specified element.

Return Value

Returns TRUE if successful, FALSE if the index was not valid.

See Also

Reference

CSimpleArray Class
CSimpleArray::RemoveAt
CSimpleArray::Add

Other Resources

CSimpleArray Members

CSimpleArray Operators

For information about the operators in **CSimpleArray**, see CSimpleArray Members.

CSimpleArray::operator []

Retrieves an element from the array.

```
T& operator[](
   int nIndex
);
```

Parameters

nIndex

The element index.

Return Value

Returns the element of the array referenced by *nIndex*.

Example

```
// csimplearray_bracketoperator.cpp
#include <string.h>
#include <atlsimpstr.h>
#include <atlstr.h>

int main ()
{
    // Create an array and display its contents
    CSimpleArray<int> iMySampleArray;

    for (int i=0; i<10; i++)
        iMySampleArray.Add(i);
    for (i=0; i<iMySampleArray.GetSize(); i++)
        printf_s("Array index %d contains %d\n",i,iMySampleArray[i]);
}
```

Output

```
Array index 0 contains 0
Array index 1 contains 1
Array index 2 contains 2
Array index 3 contains 3
Array index 4 contains 4
Array index 5 contains 5
Array index 6 contains 6
Array index 7 contains 7
Array index 8 contains 8
Array index 9 contains 9
```

See Also

Reference

CSimpleArray Class

Other Resources

CSimpleArray Members

CSimpleArray::operator =

Assignment operator.

```
CSimpleArray< T, TEqual >
& operator =(
   const CSimpleArray< T, TEqual >& src
);
```

Parameters

src

The array to copy.

Return Value

Returns a pointer to the updated **CSimpleArray** object.

Remarks

Copies all elements from the **CSimpleArray** object referenced by *src* into the current array object, replacing all existing data.

Example

```
// Create an array of chars
// and copy it to a second array
   CSimpleArray<char> cMyArray1;
   cMyArray1.Add('a');
   CSimpleArray<char> cMyArray2;
   cMyArray2 = cMyArray1;
   ATLASSERT(cMyArray2[0] == 'a');
```

See Also

Reference

CSimpleArray Class

Other Resources

CSimpleArray Members

CSimpleArrayEqualHelper Class

This class is a helper for the CSimpleArray class.

```
template <
   class T
>
class CSimpleArrayEqualHelper
```

Parameters

T

A derived class.

Remarks

This traits class is a supplement to the **CSimpleArray** class. It provides a method for comparing two elements stored in a **CSimpleArray** object. By default, the elements are compared using **operator=()**, but if the array contains complex data types that lack their own equality operator, you will need to override this class.

Requirements

Header: atlsimpcoll.h

See Also

Reference

CSimpleArray Class

CSimpleArrayEqualHelperFalse Class

Other Resources

CSimpleArrayEqualHelper Members

ATL Class Overview

CSimpleArrayEqualHelper Members

Static Functions

IsEqual	Tests two **CSimpleArray** object elements for equality.

See Also

Reference

CSimpleArrayEqualHelper Class

CSimpleArrayEqualHelper Static Functions

For information about the static functions in **CSimpleArrayEqualHelper**, see CSimpleArrayEqualHelper Members.

CSimpleArrayEqualHelper::IsEqual

Tests two **CSimpleArray** object elements for equality.

```
static bool IsEqual(
   const T& t1,
   const T& t2
);
```

Parameters

t1

An object of type T.

t2

An object of type T.

Return Value

Returns true if the elements are equal, false otherwise.

See Also

Reference

CSimpleArrayEqualHelper Class

Other Resources

CSimpleArrayEqualHelper Members

CSimpleArrayEqualHelperFalse Class

This class is a helper for the CSimpleArray class.

```
template <
   class T
>
class CSimpleArrayEqualHelperFalse
```

Parameters

T

A derived class.

Remarks

This traits class is a complement to the **CSimpleArray** class. It always returns false, and in addition, will call **ATLASSERT** with an argument of false if it is ever referenced. In situations where the equality test is not sufficiently defined, this class allows an array containing elements to operate correctly for most methods but fail in a well-defined manner for methods that depend on comparisons such as CSimpleArray::Find.

Requirements

Header: atlsimpcoll.h

See Also

Reference

CSimpleArrayEqualHelper Class

Other Resources

CSimpleArrayEqualHelperFalse Members

ATL Class Overview

CSimpleArrayEqualHelperFalse Members

Static Functions

IsEqual	Returns false.

See Also

Reference

CSimpleArrayEqualHelperFalse Class

CSimpleArrayEqualHelperFalse Static Functions

For information about the static functions in **CSimpleArrayEqualHelperFalse**, see CSimpleArrayEqualHelperFalse Members.

CSimpleArrayEqualHelperFalse::IsEqual

Returns false.

```
static bool IsEqual(
   const T& ,
   const T&
);
```

Return Value

Returns false.

Remarks

This method always returns false, and will call **ATLASSERT** with an argument of false if referenced. The purpose of **CSimpleArrayEqualHelperFalse::IsEqual** is to force methods using comparisons to fail in a well-defined manner when equality tests have not been adequately defined.

See Also

Reference

CSimpleArrayEqualHelperFalse Class

Other Resources

CSimpleArrayEqualHelperFalse Members

CSimpleDialog Class

This class implements a basic modal dialog box.

```
template <
   WORD t_wDlgTemplateID,
   BOOL t_bCenter = TRUE
>
class CSimpleDialog :
   public CDialogImplBase
```

Parameters

t_wDlgTemplateID

The resource ID of the dialog template resource.

t_bCenter

TRUE if the dialog object is to be centered on the owner window; otherwise **FALSE**.

Remarks

Implements a modal dialog box with basic functionality. **CSimpleDialog** provides support for Windows common controls only. To create and display a modal dialog box, create an instance of this class, providing the name of an existing resource template for the dialog box. The dialog box object closes when the user clicks any control with a pre-defined value (such as IDOK or IDCANCEL).

CSimpleDialog allows you to create only modal dialog boxes. **CSimpleDialog** provides the dialog box procedure, which uses the default message map to direct messages to the appropriate handlers.

See Implementing a Dialog Box for more information.

Requirements

Header: atlwin.h

See Also

Other Resources

CSimpleDialog Members

ATL Class Overview

CSimpleDialog Members

Methods

DoModal	Creates a modal dialog box.

See Also

Reference

CSimpleDialog Class

CSimpleDialog Methods

For information about the methods in **CSimpleDialog**, see CSimpleDialog Members.

CSimpleDialog::DoModal

Invokes a modal dialog box and returns the dialog-box result when done.

```
INT_PTR DoModal(
   HWND hWndParent = ::GetActiveWindow()
);
```

Parameters

hWndParent

A handle to the parent of the dialog box. If no value is provided, the parent is set to the current active window.

Return Value

If successful, the return value is the resource ID of the control that dismissed the dialog box.

If the function fails, the return value is –1. To get extended error information, call **GetLastError**.

Remarks

This method handles all interaction with the user while the dialog box is active. This is what makes the dialog box modal; that is, the user cannot interact with other windows until the dialog box is closed.

See Also

Reference

CSimpleDialog Class

Other Resources

CSimpleDialog Members

CSimpleMap Class

This class provides support for a simple mapping array.

```
template <
   class TKey,
   class TVal,
   class TEqual = CSimpleMapEqualHelper< TKey, TVal >
>
class CSimpleMap
```

Parameters

TKey

The key element type.

TVal

The value element type.

TEqual

A trait object, defining the equality test for elements of type *T*.

Remarks

CSimpleMap provides support for a simple mapping array of any given type *T*, managing an unordered array of key elements and their associated values.

The parameter *TEqual* provides a means of defining an equality function for two elements of type *T*. By creating a class similar to CSimpleMapEqualHelper, it is possible to alter the behavior of the equality test for any given array. For example, when dealing with an array of pointers, it may be useful to define the equality as depending on the values the pointers reference. The default implementation utilizes **operator==()**.

Both **CSimpleMap** and CSimpleArray are provided for compatibility with previous ATL releases, and more complete and efficient collection implementations are provided by CAtlArray and CAtlMap.

Unlike other map collections in ATL and MFC, this class is implemented with a simple array, and lookup searches require a linear search. **CAtlMap** should be used when the array contains a large number of elements.

Requirements

Header: atlsimpcoll.h

Example

```
// Create a map with an integer key
// and character pointer value
      CSimpleArray<int, char *> iArray;
```

See Also

Other Resources

CSimpleMap Members

ATL Class Overview

CSimpleMap Members

Methods

~CSimpleMap	The destructor.
Add	Adds a key and associated value to the map array.
CSimpleMap	The constructor.
FindKey	Finds a specific key.
FindVal	Finds a specific value.
GetKeyAt	Retrieves the specified key.
GetSize	Returns the number of entries in the mapping array.
GetValueAt	Retrieves the specified value.
Lookup	Returns the value associated with the given key.
Remove	Removes a key and matching value.
RemoveAll	Removes all keys and values.
RemoveAt	Removes a specific key and matching value.
ReverseLookup	Returns the key associated with the given value.
SetAt	Sets the value associated with the given key.
SetAtIndex	Sets the specific key and value.

Typedefs

_ArrayElementType	Typedef for the key type.
_ArrayKeyType	Typedef for the value type.

See Also

Reference

CSimpleMap Class

CSimpleMap Methods

For information about the methods in **CSimpleMap**, see CSimpleMap Members.

CSimpleMap::Add

Adds a key and associated value to the map array.

```
BOOL Add(
   const TKey& key,
   const TVal& val
);
```

Parameters

key

The key.

val

The associated value.

Return Value

Returns TRUE if the key and value were successfully added, FALSE otherwise.

Remarks

Each key and value pair added causes the mapping array memory to be freed and reallocated, in order to ensure the data for each is always stored contiguously. That is, the second key element always directly follows the first key element in memory and so on.

See Also

Reference

CSimpleMap Class

Other Resources

CSimpleMap Members

CSimpleMap::CSimpleMap

The constructor.

```
CSimpleMap( );
```

Remarks

Initializes the data members.

See Also

Reference

CSimpleMap Class

CSimpleMap::~CSimpleMap

Other Resources

CSimpleMap Members

CSimpleMap::~CSimpleMap

The destructor.

```
~CSimpleMap( );
```

Remarks

Frees all allocated resources.

See Also

Reference

CSimpleMap Class

CSimpleMap::CSimpleMap

Other Resources

CSimpleMap Members

CSimpleMap::FindKey

Finds a specific key.

```
int FindKey(
   const TKey& key
) const;
```

Parameters

key

The key to search for.

Return Value

Returns the index of the key if found, otherwise returns -1.

See Also

Reference

CSimpleMap Class

CSimpleMap::FindVal

Other Resources

CSimpleMap Members

CSimpleMap::FindVal

Finds a specific value.

```
int FindVal(
   const TVal& val
) const;
```

Parameters

val

The value for which to search.

Return Value

Returns the index of the value if it is found, otherwise returns -1.

See Also

Reference

CSimpleMap Class

CSimpleMap::FindKey

Other Resources

CSimpleMap Members

CSimpleMap::GetKeyAt

Retrieves the key at the specified index.

```
TKey& GetKeyAt(
   int nIndex
) const;
```

Parameters

nIndex

The index of the key to return.

Return Value

Returns the key referenced by *nIndex*.

Remarks

The index passed by *nIndex* must be valid for the return value to be meaningful.

See Also

Reference

CSimpleMap Class

CSimpleMap::GetValueAt

Other Resources

CSimpleMap Members

CSimpleMap::GetSize

Returns the number of entries in the mapping array.

```
int GetSize( ) const;
```

Return Value

Returns the number of entries (a key and value is one entry) in the mapping array.

See Also

Reference

CSimpleMap Class

Other Resources

CSimpleMap Members

CSimpleMap::GetValueAt

Retrieves the value at the specific index.

```
TVal& GetValueAt(
   int nIndex
) const;
```

Parameters

nIndex

The index of the value to return.

Return Value

Returns the value referenced by *nIndex*.

Remarks

The index passed by *nIndex* must be valid for the return value to be meaningful.

See Also

Reference

CSimpleMap Class

CSimpleMap::GetKeyAt

Other Resources

CSimpleMap Members

CSimpleMap::Lookup

Returns the value associated with the given key.

```
TVal Lookup(
   const TKey& key
) const;
```

Parameters

key

The key.

Return Value

Returns the associated value. If no matching key is found, NULL is returned.

See Also

Reference

CSimpleMap Class

CSimpleMap::ReverseLookup

Other Resources

CSimpleMap Members

CSimpleMap::Remove

Removes a key and matching value.

```
BOOL Remove(
   const TKey& key
);
```

Parameters

key

The key.

Return Value

Returns TRUE if the key, and matching value, were successfully removed, FALSE otherwise.

See Also

Reference

CSimpleMap Class
CSimpleMap::RemoveAll
CSimpleMap::RemoveAt

Other Resources

CSimpleMap Members

CSimpleMap::RemoveAll

Removes all keys and values.

```
void RemoveAll( );
```

Remarks

Removes all keys and values from the mapping array object.

See Also

Reference

CSimpleMap Class
CSimpleMap::Remove
CSimpleMap::RemoveAt

Other Resources

CSimpleMap Members

CSimpleMap::RemoveAt

Removes a key and associated value at the specified index.

```
BOOL RemoveAt(
   int nIndex
);
```

Parameters

nIndex

The index of the key and associated value to remove.

Return Value

Returns TRUE on success, FALSE if the index specified is an invalid index.

See Also

Reference

CSimpleMap Class
CSimpleMap::RemoveAll
CSimpleMap::Remove

Other Resources

CSimpleMap Members

CSimpleMap::ReverseLookup

Returns the key associated with the given value.

```
TKey ReverseLookup(
   const TVal& val
) const;
```

Parameters

val

The value.

Return Value

Returns the associated key. If no matching key is found, NULL is returned.

See Also

Reference

CSimpleMap Class

CSimpleMap::Lookup

Other Resources

CSimpleMap Members

CSimpleMap::SetAt

Sets the value associated with the given key.

```
BOOL SetAt(
   const TKey& key,
   const TVal& val
);
```

Parameters

key

The key.

val

The new value to assign.

Return Value

Returns TRUE if the key was found, and the value was successfully changed, FALSE otherwise.

See Also

Reference

CSimpleMap Class

CSimpleMap::SetAtIndex

Other Resources

CSimpleMap Members

CSimpleMap::SetAtIndex

Sets the key and value at a specified index.

```
BOOL SetAtIndex(
   int nIndex,
   const TKey& key,
   const TVal& val
);
```

Parameters

nIndex

The index, referencing the key and value pairing to change.

key

The new key.

val

The new value.

Return Value

Returns TRUE if successful, FALSE if the index was not valid.

Remarks

Updates both the key and value pointed to by *nIndex*.

See Also

Reference

CSimpleMap Class

CSimpleMap::SetAt

Other Resources

CSimpleMap Members

CSimpleMap Typedefs

For information about the typedefs in **CSimpleMap**, see CSimpleMap Members.

CSimpleMap::_ArrayElementType

A typedef for the key type.

```
typedef TVal _ArrayElementType;
```

See Also

Reference

CSimpleMap Class

Other Resources

CSimpleMap Members

CSimpleMap::_ArrayKeyType

A typedef for the value type.

```
typedef TKey _ArrayKeyType;
```

See Also

Reference

CSimpleMap Class

Other Resources

CSimpleMap Members

CSimpleMapEqualHelper Class

This class is a helper for the CSimpleMap class.

```
template <
   class TKey,
   class TVal
>
class CSimpleMapEqualHelper
```

Parameters

TKey

The key element.

TVal

The value element.

Remarks

This traits class is a supplement to the **CSimpleMap** class. It provides methods for comparing two **CSimpleMap** object elements (specifically, the key and value components) for equality. By default, the keys and values are compared using **operator=()**, but if the map contains complex data types that lack their own equality operator, this class can be overridden to provide the extra required functionality.

Requirements

Header: atlsimpcoll.h

See Also

Reference

CSimpleMapEqualHelperFalse Class

Other Resources

CSimpleMapEqualHelper Members

ATL Class Overview

CSimpleMapEqualHelper Members

Static Functions

IsEqualKey	Tests two keys for equality.
IsEqualValue	Tests two values for equality.

See Also

Reference

CSimpleMapEqualHelper Class

CSimpleMapEqualHelper Static Functions

For information about the static functions in **CSimpleMapEqualHelper**, see CSimpleMapEqualHelper Members.

CSimpleMapEqualHelper::IsEqualKey

Tests two keys for equality.

```
static bool IsEqualKey(
   const TKey& k1,
   const TKey& k2
);
```

Parameters

k1

The first key.

k2

The second key.

Return Value

Returns true if the keys are equal, false otherwise.

See Also

Reference

CSimpleMapEqualHelper Class

CSimpleMapEqualHelper::IsEqualValue

Other Resources

CSimpleMapEqualHelper Members

CSimpleMapEqualHelper::IsEqualValue

Tests two values for equality.

```
static bool IsEqualValue(
   const TVal& v1,
   const TVal& v2
);
```

Parameters

v1

The first value.

v2

The second value.

Return Value

Returns true if the values are equal, false otherwise.

See Also

Reference

CSimpleMapEqualHelper Class

CSimpleMapEqualHelper::IsEqualKey

Other Resources

CSimpleMapEqualHelper Members

CSimpleMapEqualHelperFalse Class

This class is a helper for the CSimpleMap class.

```
template < class TKey, class TVal > class CSimpleMapEqualHelperFalse
```

Remarks

This traits class is a supplement to the **CSimpleMap** class. It provides a method for comparing two elements contained in the **CSimpleMap** object, specifically two value elements or two key elements.

The value comparison will always return false, and in addition, will call **ATLASSERT** with an argument of false if it is ever referenced. In situations where the equality test is not sufficiently defined, this class allows a map containing key/value pairs to operate correctly for most methods but fail in a well-defined manner for methods that depend on comparisons such as CSimpleMap::FindVal.

Requirements

Header: atlsimpcoll.h

See Also

Reference

CSimpleMapEqualHelper Class

Other Resources

CSimpleMapEqualHelperFalse Members

ATL Class Overview

CSimpleMapEqualHelperFalse Members

Static Functions

IsEqualKey	Tests two keys for equality.
IsEqualValue	Returns false.

See Also

Reference

CSimpleMapEqualHelperFalse Class

CSimpleMapEqualHelperFalse Methods

For information about the methods in **CSimpleMapEqualHelperFalse**, see CSimpleMapEqualHelperFalse Members.

CSimpleMapEqualHelperFalse::IsEqualKey

Tests two keys for equality.

```
static bool IsEqualKey(
   const TKey& k1,
   const TKey& k2
);
```

Parameters

k1

The first key.

k2

The second key.

Return Value

Returns true if the keys are equal, false otherwise.

Remarks

This method calls CSimpleArrayEqualHelper.

See Also

Reference

CSimpleMapEqualHelperFalse Class

CSimpleMapEqualHelperFalse::IsEqualValue

Other Resources

CSimpleMapEqualHelperFalse Members

CSimpleMapEqualHelperFalse::IsEqualValue

Returns false.

```
static bool IsEqualValue(
   const TVal& ,
   const TVal&
);
```

Return Value

Returns false.

Remarks

This method always returns false, and will call **ATLASSERT** with an argument of false if it is ever referenced. The purpose of **CSimpleMapEqualHelperFalse::IsEqualValue** is to force methods using comparisons to fail in a well-defined manner when equality tests have not been adequately defined.

See Also

Reference

CSimpleMapEqualHelperFalse Class

CSimpleMapEqualHelperFalse::IsEqualKey

Other Resources

CSimpleMapEqualHelperFalse Members

CSnapInItemImpl Class

This class provides methods for implementing a snap-in node object.

```
template <
   class T,
   BOOL bIsExtension = FALSE
>
class ATL_NO_VTABLE CSnapInItemImpl :
   public CSnapInItem
```

Parameters

T

Your class, derived from **CSnapInItemImpl**.

bIsExtension

TRUE if the object is a snap-in extension; otherwise **FALSE**.

Remarks

CSnapInItemImpl provides a basic implementation for a snap-in node object, such as adding menu items and toolbars, and forwarding commands for the snap-in node to the appropriate handler function. These features are implemented using several different interfaces and map types. The default implementation handles notifications sent to the node object by determining the correct instance of the derived class and then forwarding the message to the correct instance.

Requirements

Header: atlsnap.h

See Also

Other Resources

CSnapInItemImpl Members

ATL Class Overview

CSnapInItemImpl Members

Methods

AddMenuItems	Adds menu items to a context menu.
Command	Called by the console when a custom menu item is selected.
CreatePropertyPages	Adds pages to the property sheet of the snap-in.
CSnapInItemImpl	Constructor.
FillData	Copies information on the snap-in object into a specified stream.
GetResultPaneInfo	Retrieves the **RESULTDATAITEM** structure of the snap-in.
GetResultViewType	Determines the type of view used by the result pane.
GetScopePaneInfo	Retrieves the **SCOPEDATAITEM** structure of the snap-in.
Notify	Called by the console to notify the snap-in of actions taken by the user.
SetMenuInsertionFlags	Modifies the menu insertion flags for a snap-in object.
SetToolbarButtonInfo	Sets the information of the specified toolbar button.
UpdateMenuState	Updates the state of a context menu item.
UpdateToolbarButton	Updates the state of the specified toolbar button.

Data Members

m_bstrDisplayName	The name of the snap-in object.
m_resultDataItem	The Windows **RESULTDATAITEM** structure used by the **CsnapInItemImpl** object.
m_scopeDataItem	The Windows **SCOPEDATAITEM** structure used by the **CSnapInItemImpl** object.

See Also

Reference

CSnapInItemImpl Class

CSnapInItemImpl Methods

For information about the methods in **CSnapInItemImpl**, see CSnapInItemImpl Members.

CSnapInItemImpl::AddMenuItems

This method implements the Win32 function IExtendContextMenu::AddMenuItems.

```
AddMenuItems(
   LPCONTEXTMENUCALLBACK piCallback,
   long *pInsertionAllowed,
      DATA_OBJECT_TYPES type
);
```

Parameters

piCallback

[in] Pointer to the **IContextMenuCallback** that can add items to the context menu.

pInsertionAllowed

[in, out] Identifies Microsoft Management Console (MMC)-defined, menu-item insertion points that can be used. This can be a combination of the following flags:

- **CCM_INSERTIONALLOWED_TOP** Items can be inserted at the top of a context menu.
- **CCM_INSERTIONALLOWED_NEW** Items can be inserted in the Create New submenu.
- **CCM_INSERTIONALLOWED_TASK** Items can be inserted in the Task submenu.
- **CCM_INSERTIONALLOWED_VIEW** Items can be inserted in the toolbar view menu or in the View submenu of the result pane context menu.

type

[in] Specifies the type of object. It can have one of the following values:

- **CCT_SCOPE** Data object for scope pane context.
- **CCT_RESULT** Data object for result pane context.
- **CCT_SNAPIN_MANAGER** Data object for snap-in manager context.
- **CCT_UNINITIALIZED** Data object has an invalid type.

See Also

Reference

CSnapInItemImpl Class

Other Resources

CSnapInItemImpl Members

CSnapInItemImpl::Command

This method implements the Win32 function IExtendContextMenu::Command.

```
Command(
   long lCommandID,
      DATA_OBJECT_TYPES type
);
```

Parameters

lCommandID

[in] Specifies the command identifier of the menu item.

type

[in] Specifies the type of object. It can have one of the following values:

- **CCT_SCOPE** Data object for scope pane context.
- **CCT_RESULT** Data object for result pane context.
- **CCT_SNAPIN_MANAGER** Data object for snap-in manager context.
- **CCT_UNINITIALIZED** Data object has an invalid type.

See Also

Reference

CSnapInItemImpl Class

Other Resources

CSnapInItemImpl Members

CSnapInItemImpl::CreatePropertyPages

This method implements the Win32 function IExtendPropertySheet::CreatePropertyPages.

```
CreatePropertyPages(
   LPPROPERTYSHEETCALLBACK lpProvider,
   long handle,
   IUnknown* pUnk,
      DATA_OBJECT_TYPES type
);
```

Parameters

lpProvider

[in] Pointer to the **IPropertySheetCallback** interface.

handle

[in] Specifies the handle used to route the **MMCN_PROPERTY_CHANGE** notification message to the appropriate data class.

pUnk

[in] Pointer to the **IExtendPropertySheet** interface on the object that contains context information about the node.

type

[in] Specifies the type of object. It can have one of the following values:

- **CCT_SCOPE** Data object for scope pane context.
- **CCT_RESULT** Data object for result pane context.
- **CCT_SNAPIN_MANAGER** Data object for snap-in manager context.
- **CCT_UNINITIALIZED** Data object has an invalid type.

See Also

Reference

CSnapInItemImpl Class

Other Resources

CSnapInItemImpl Members

CSnapInItemImpl::CSnapInItemImpl

Constructs a **CSnapInItemImpl** object.

```
CSnapInItemImpl( );
```

See Also

Reference

CSnapInItemImpl Class

Other Resources

CSnapInItemImpl Members

CSnapInItemImpl::FillData

This function is called to retrieve information about the item.

```
FillData(
   CLIPFORMAT cf,
   LPSTREAM pStream
);
```

Parameters

cf

[in] The format (text, rich text, or rich text with OLE items) of the Clipboard.

pStream

[in] A pointer to the stream containing the object data.

Remarks

To properly implement this function, copy the correct information into the stream (*pStream*), depending on the Clipboard format indicated by *cf*.

See Also

Reference

CSnapInItemImpl Class

Other Resources

CSnapInItemImpl Members

CSnapInItemImpl::GetResultViewType

Call this function to retrieve the type of view for the result pane of the snap-in object.

```
GetResultViewType(
   LPOLESTR *ppViewType,
   long *pViewOptions
);
```

Parameters

ppViewType

[out] Pointer to the address of the returned view type.

pViewOptions

[out] Pointer to the **MMC_VIEW_OPTIONS** enumeration, which provides the console with options specified by the owning snap-in. This value can be one of the following:

- **MMC_VIEW_OPTIONS_NOLISTVIEWS** = 0x00000001 Tells the console to refrain from presenting standard list view choices in the **View** menu. Allows the snap-in to display its own custom views only in the result view pane. This is the only option flag defined at this time.
- **MMC_VIEW_OPTIONS_NONE** = 0 Allows the default view options.

See Also

Reference

CSnapInItemImpl Class

Other Resources

CSnapInItemImpl Members

CSnapInItemImpl::GetScopePaneInfo

Call this function to retrieve the **SCOPEDATAITEM** structure of the snap-in.

```
GetScopePaneInfo (
   SCOPEDATAITEM *pScopeDataItem
);
```

Parameters

pScopeDataItem

[out] A pointer to the **SCOPEDATAITEM** structure of the **CSnapInItemImpl** object.

See Also

Reference

CSnapInItemImpl Class

Other Resources

CSnapInItemImpl Members

CSnapInItemImpl::GetResultPaneInfo

Call this function to retrieve the **RESULTDATAITEM** structure of the snap-in.

```
GetResultPaneInfo (
   RESULTDATAITEM *pResultDataItem
);
```

Parameters

pResultDataItem

[out] A pointer to the **RESULTDATAITEM** structure of the **CSnapInItemImpl** object.

See Also

Reference

CSnapInItemImpl Class

Other Resources

CSnapInItemImpl Members

CSnapInItemImpl::Notify

Called when the snap-in object is acted upon by the user.

```
STDMETHOD(
   Notify
)(
   MMC_NOTIFY_TYPE event,
   long arg,
   long param,
   IComponentData* pComponentData,
   IComponent* pComponent,
   DATA_OBJECT_TYPES type
) = 0;
```

Parameters

event

[in] Identifies an action taken by a user. The following notifications are possible:

- **MMCN_ACTIVATE** Sent when a window is being activated and deactivated.
- **MMCN_ADD_IMAGES** Sent to add images to the result pane.
- **MMCN_BTN_CLICK** Sent when the user clicks one of the toolbar buttons.
- **MMCN_CLICK** Sent when a user clicks a mouse button on a list view item.
- **MMCN_DBLCLICK** Sent when a user double clicks a mouse button on a list view item.
- **MMCN_DELETE** Sent to inform the snap-in that the object should be deleted.
- **MMCN_EXPAND** Sent when a folder needs to be expanded or contracted.
- **MMCN_MINIMIZED** Sent when a window is being minimized or maximized.
- **MMCN_PROPERTY_CHANGE** Sent to notify a snap-in object that the snap-in object's view is about to change.
- **MMCN_REMOVE_CHILDREN** Sent when the snap-in must delete the entire subtree it has added below the specified node.
- **MMCN_RENAME** Sent the first time to query for a rename and the second time to do the rename.
- **MMCN_SELECT** Sent when an item in the scope or result view pane is selected.
- **MMCN_SHOW** Sent when a scope item is selected or deselected for the first time.
- **MMCN_VIEW_CHANGE** Sent when the snap-in can update all views when a change occurs.

arg

[in] Depends on the notification type.

param

[in] Depends on the notification type.

pComponentData

[out] A pointer to the object implementing **IComponentData**. This parameter is **NULL** if the notification is not being forwarded from **IComponentData::Notify**.

pComponent

[out] A pointer to the object that implements **IComponent**. This parameter is **NULL** if the notification is not being forwarded from **IComponent::Notify**.

type

[in] Specifies the type of object. It can have one of the following values:

- **CCT_SCOPE** Data object for scope pane context.
- **CCT_RESULT** Data object for result pane context.
- **CCT_SNAPIN_MANAGER** Data object for snap-in manager context.
- **CCT_UNINITIALIZED** Data object has an invalid type.

See Also

Reference

CSnapInItemImpl Class

Other Resources

CSnapInItemImpl Members

CSnapInItemImpl::QueryPagesFor

Called to see if the snap-in node supports property pages.

```
QueryPagesFor(
   DATA_OBJECT_TYPES type
);
```

See Also

Reference

CSnapInItemImpl Class

Other Resources

CSnapInItemImpl Members

CSnapInItemImpl::SetMenuInsertionFlags

Call this function to modify the menu insertion flags, specified by *pInsertionAllowed*, for the snap-in object.

```
void SetMenuInsertionFlags(
   bool bBeforeInsertion,
   long* pInsertionAllowed
);
```

Parameters

bBeforeInsertion

[in] Nonzero if the function should be called before items are added to the context menu; otherwise 0.

pInsertionAllowed

[in, out] Identifies Microsoft Management Console (MMC)-defined, menu-item insertion points that can be used. This can be a combination of the following flags:

- **CCM_INSERTIONALLOWED_TOP** Items can be inserted at the top of a context menu.
- **CCM_INSERTIONALLOWED_NEW** Items can be inserted in the Create New submenu.
- **CCM_INSERTIONALLOWED_TASK** Items can be inserted in the Task submenu.
- **CCM_INSERTIONALLOWED_VIEW** Items can be inserted in the toolbar view menu or in the View submenu of the result pane context menu.

Remarks

If you are developing a primary snap-in, you can reset any of the insertion flags as a way of restricting the kind of menu items that a third-party extension can add. For example, the primary snap-in can clear the **CCM_INSERTIONALLOWED_NEW** flag to prevent extensions from adding their own Create New menu items.

You should not attempt to set bits in *pInsertionAllowed* that were originally cleared. Future versions of MMC may use bits not currently defined so you should not change bits that are currently not defined.

See Also

Reference

CSnapInItemImpl Class

Other Resources

CSnapInItemImpl Members

CSnapInItemImpl::SetToolbarButtonInfo

Call this function to modify any toolbar button styles, of the snap-in object, before the toolbar is created.

```
void SetToolbarButtonInfo(
   UINT id,
   BYTE *fsState,
   BYTE *fsType
);
```

Parameters

id

[in] The ID of the toolbar button to be set.

fsState

[in] The state flags of the button. Can be one or more of the following:

- **TBSTATE_CHECKED** The button has the **TBSTYLE_CHECKED** style and is being pressed.
- **TBSTATE_ENABLED** The button accepts user input. A button that does not have this state does not accept user input and is grayed.
- **TBSTATE_HIDDEN** The button is not visible and cannot receive user input.
- **TBSTATE_INDETERMINATE** The button is grayed.
- **TBSTATE_PRESSED** The button is being pressed.
- **TBSTATE_WRAP** A line break follows the button. The button must also have the **TBSTATE_ENABLED**.

fsType

[in] The state flags of the button. Can be one or more of the following:

- **TBSTYLE_BUTTON** Creates a standard push button.
- **TBSTYLE_CHECK** Creates a button that toggles between the pressed and not-pressed states each time the user clicks it. The button has a different background color when it is in the pressed state.
- **TBSTYLE_CHECKGROUP** Creates a check button that stays pressed until another button in the group is pressed.
- **TBSTYLE_GROUP** Creates a button that stays pressed until another button in the group is pressed.
- **TBSTYLE_SEP** Creates a separator, providing a small gap between button groups. A button that has this style does not receive user input.

See Also

Reference

CSnapInItemImpl Class

Other Resources

CSnapInItemImpl Members

CSnapInItemImpl::UpdateMenuState

Call this function to modify a menu item before it is inserted into the context menu of the snap-in object.

```
void UpdateMenuState(
   UINT id,
   LPTSTR pBuf,
   UINT *flags
);
```

Parameters

id

[in] The ID of the menu item to be set.

pBuf

[in] A pointer to the string for the menu item to be updated.

flags

[in] Specifies the new state flags. This can be a combination of the following flags:

- **MF_POPUP** Specifies that this is a submenu within the context menu. Menu items, insertion points, and further submenus may be added to this submenu using its **ICommandID** as their **IInsertionPointID**.
- **MF_BITMAP** and **MF_OWNERDRAW** These flags are not permitted and will result in a return value of **E_INVALIDARG**.
- **MF_SEPARATOR** Draws a horizontal dividing line. Only **IContextMenuProvider** is allowed to add menu items with **MF_SEPARATOR** set.
- **MF_CHECKED** Places a check mark next to the menu item.
- **MF_DISABLED** Disables the menu item so it cannot be selected, but the flag does not gray it.
- **MF_ENABLED** Enables the menu item so it can be selected, restoring it from its grayed state.
- **MF_GRAYED** Disables the menu item, graying it so it cannot be selected.
- **MF_MENUBARBREAK** Functions the same as the **MF_MENUBREAK** flag for a menu bar. For a drop-down menu, submenu, or shortcut menu, the new column is separated from the old column by a vertical line.
- **MF_MENUBREAK** Places the item on a new line (for a menu bar) or in a new column (for a drop-down menu, submenu, or shortcut menu) without separating columns.
- **MF_UNCHECKED** Does not place a check mark next to the item (default).

The following groups of flags cannot be used together:

- **MF_DISABLED**, **MF_ENABLED**, and **MF_GRAYED**.
- **MF_MENUBARBREAK** and **MF_MENUBREAK**.
- **MF_CHECKED** and **MF_UNCHECKED**.

See Also

Reference

CSnapInItemImpl Class

Other Resources

CSnapInItemImpl Members

CSnapInItemImpl::UpdateToolbarButton

Call this function to modify a toolbar button, of the snap-in object, before it is displayed.

```
BOOL UpdateToolbarButton(
   UINT id,
   BYTE fsState
);
```

Parameters

id

Specifies the button ID of the toolbar button to be updated.

fsState

Specifies a toolbar button state. If this state is to be set, return **TRUE**. This can be a combination of the following flags:

- **ENABLED** The button accepts user input. A button that does not have this state does not accept user input and is grayed.
- **CHECKED** The button has the **CHECKED** style and is being pressed.
- **HIDDEN** The button is not visible and cannot receive user input.
- **INDETERMINATE** The button is grayed.
- **BUTTONPRESSED** The button is being pressed.

See Also

Reference

CSnapInItemImpl Class

Other Resources

CSnapInItemImpl Members

CSnapInItemImpl Data Members

For information about the data members in **CSnapInItemImpl**, see CSnapInItemImpl Members.

CSnapInItemImpl::m_bstrDisplayName

Contains the string displayed for the node item.

```
CComBSTR m_bstrDisplayName;
```

See Also

Reference

CSnapInItemImpl Class

Other Resources

CSnapInItemImpl Members

CSnapInItemImpl::m_scopeDataItem

The SCOPEDATAITEM structure of the snap-in data object.

```
SCOPEDATAITEM m_scopeDataItem;
```

See Also

Reference

CSnapInItemImpl Class

Other Resources

CSnapInItemImpl Members

CSnapInItemImpl::m_resultDataItem

The RESULTDATAITEM structure of the snap-in data object.

```
RESULTDATAITEM m_resultDataItem;
```

See Also

Reference

CSnapInItemImpl Class

Other Resources

CSnapInItemImpl Members

CSnapInPropertyPageImpl Class

This class provides methods for implementing a snap-in property page object.

```
CSnapInPropertyPageImpl : public CDialogImplBase
```

Remarks

CSnapInPropertyPageImpl provides a basic implementation for a snap-in property page object. The basic features of a snap-in property page are implemented using several different interfaces and map types.

Requirements

Header: atlsnap.h

See Also

Other Resources

CSnapInPropertyPageImpl Members

ATL Class Overview

CSnapInPropertyPageImpl Members

Methods

CancelToClose	Changes the status of the **OK** and **Cancel** buttons.
Create	Initializes a newly created **CSnapInPropertyPageImpl** object.
CSnapInPropertyPageImpl	Constructor.
OnApply	Called by the framework when the user clicks the **Apply Now** button while using a wizard-type property sheet.
OnHelp	Called by the framework when the user clicks the **Help** button while using a wizard-type property sheet.
OnKillActive	Called by the framework when the current page is no longer active.
OnQueryCancel	Called by the framework when the user clicks the **Cancel** button and before the cancel has taken place.
OnReset	Called by the framework when the user clicks the **Reset** button while using a wizard-type property sheet.
OnSetActive	Called by the framework when the current page becomes active.
OnWizardBack	Called by the framework when the user clicks the **Back** button while using a wizard-type property sheet.
OnWizardFinish	Called by the framework when the user clicks the **Finish** button while using a wizard-type property sheet.
OnWizardNext	Called by the framework when the user clicks the **Next** button while using a wizard-type property sheet.
QuerySiblings	Forwards the current message to all pages of the property sheet.
SetModified	Call to activate or deactivate the **Apply Now** button.

Data Members

m_psp	The Windows **PROPSHEETPAGE** structure used by the **CSnapInPropertyPageImpl** object.

See Also

Reference

CSnapInPropertyPageImpl Class

CSnapInPropertyPageImpl Methods

For information about the methods in **CSnapInPropertyPageImpl**, see CSnapInPropertyPageImpl Members.

CSnapInPropertyPageImpl::CancelToClose

Call this function after an unrecoverable change has been made to the data in a page of a modal property sheet.

```
void CancelToClose( );
```

Remarks

This function will change the **OK** button to **Close** and disable the **Cancel** button. This change alerts the user that a change is permanent and the modifications cannot be cancelled.

The **CancelToClose** member function does nothing in a modeless property sheet, because a modeless property sheet does not have a **Cancel** button by default.

See Also

Reference

CSnapInPropertyPageImpl Class

Other Resources

CSnapInPropertyPageImpl Members

CSnapInPropertyPageImpl::CSnapInPropertyPageImpl

Constructs a **CSnapInPropertyPageImpl** object.

```
CSnapInPropertyPageImpl(
   LPCTSTR lpszTitle = NULL
);
```

Parameters

lpszTitle

[in] The title of the property page.

Remarks

To initialize the underlying structure, call CSnapInPropertyPageImpl::Create.

See Also

Reference

CSnapInPropertyPageImpl Class

Other Resources

CSnapInPropertyPageImpl Members

CSnapInPropertyPageImpl::Create

Call this function to initialize the underlying structure of the property page.

```
HPROPSHEETPAGE Create( );
```

Return Value

A handle to a **PROPSHEETPAGE** structure containing the attributes of the newly created property sheet.

Remarks

You should first call CSnapInPropertyPageImpl::CSnapInPropertyPageImpl before calling this function.

See Also

Reference

CSnapInPropertyPageImpl Class

Other Resources

CSnapInPropertyPageImpl Members

CSnapInPropertyPageImpl::OnApply

This member function is called when the user clicks the **OK** or the **Apply Now** button.

```
BOOL OnApply( );
```

Return Value

Nonzero if the changes are accepted; otherwise 0.

Remarks

Before **OnApply** can be called by the framework, you must have called **SetModified** and set its parameter to **TRUE**. This will activate the **Apply Now** button as soon as the user makes a change on the property page.

Override this member function to specify what action your program takes when the user clicks the **Apply Now** button. When overriding, the function should return **TRUE** to accept changes and **FALSE** to prevent changes from taking effect.

The default implementation of **OnApply** returns **TRUE**.

See Also

Reference

CSnapInPropertyPageImpl Class

Other Resources

CSnapInPropertyPageImpl Members

CSnapInPropertyPageImpl::OnHelp

This member function is called when the user clicks the **Help** button for the property page.

```
void OnHelp( );
```

Remarks

Override this member function to display help for the property page.

See Also

Reference

CSnapInPropertyPageImpl Class

Other Resources

CSnapInPropertyPageImpl Members

CSnapInPropertyPageImpl::OnKillActive

This member function is called when the page is no longer the active page.

```
BOOL OnKillActive( );
```

Return Value

Nonzero if data was updated successfully; otherwise 0.

Remarks

Override this member function to perform special data validation tasks.

See Also

Reference

CSnapInPropertyPageImpl Class

Other Resources

CSnapInPropertyPageImpl Members

CSnapInPropertyPageImpl::OnQueryCancel

This member function is called when the user clicks the **Cancel** button and before the cancel action has taken place.

```
BOOL OnQueryCancel( );
```

Return Value

Nonzero to allow the cancel operation; otherwise 0.

Remarks

Override this member function to specify an action the program takes when the user clicks the **Cancel** button.

The default implementation of **OnQueryCancel** returns **TRUE**.

See Also

Reference

CSnapInPropertyPageImpl Class

Other Resources

CSnapInPropertyPageImpl Members

CSnapInPropertyPageImpl::OnReset

This member function is called when the user clicks the **Cancel** button.

```
void OnReset( );
```

Remarks

When this function is called, changes to all property pages that were made by the user previously clicking the **Apply Now** button are discarded, and the property sheet retains focus.

Override this member function to specify what action the program takes when the user clicks the **Cancel** button.

See Also

Reference

CSnapInPropertyPageImpl Class

Other Resources

CSnapInPropertyPageImpl Members

CSnapInPropertyPageImpl::OnSetActive

This member function is called when the page is chosen by the user and becomes the active page.

```
BOOL OnSetActive( );
```

Return Value

Nonzero if the page was successfully set active; otherwise 0.

Remarks

Override this member function to perform tasks when a page is activated. Your override of this member function should call the default version before any other processing is done.

The default implementation returns **TRUE**.

See Also

Reference

CSnapInPropertyPageImpl Class

Other Resources

CSnapInPropertyPageImpl Members

CSnapInPropertyPageImpl::OnWizardBack

This member function is called when the user clicks the **Back** button in a wizard.

```
BOOL OnWizardBack( );
```

Return Value

- 0 to automatically advance to the previous page.
- –1 to prevent the page from changing.

To jump to a page other than the next one, return the identifier of the dialog box to be displayed.

Remarks

Override this member function to specify some action the user must take when the **Back** button is clicked.

See Also

Reference

CSnapInPropertyPageImpl Class

Other Resources

CSnapInPropertyPageImpl Members

CSnapInPropertyPageImpl::OnWizardFinish

This member function is called when the user clicks the **Finish** button in a wizard.

```
BOOL OnWizardFinish( );
```

Return Value

Nonzero if the property sheet is destroyed when the wizard finishes; otherwise zero.

Remarks

Override this member function to specify some action the user must take when the **Finish** button is clicked.

See Also

Reference

CSnapInPropertyPageImpl Class

Other Resources

CSnapInPropertyPageImpl Members

CSnapInPropertyPageImpl::OnWizardNext

This member function is called when the user clicks the **Next** button in a wizard.

```
BOOL OnWizardNext( );
```

Return Value

- 0 to automatically advance to the next page.
- –1 to prevent the page from changing.

To jump to a page other than the next one, return the identifier of the dialog box to be displayed.

Remarks

Override this member function to specify some action the user must take when the **Next** button is clicked.

See Also

Reference

CSnapInPropertyPageImpl Class

Other Resources

CSnapInPropertyPageImpl Members

CSnapInPropertyPageImpl::QuerySiblings

Call this member function to forward a message to each page in the property sheet.

```
LRESULT QuerySiblings(
   WPARAM wParam,
   LPARAM lParam
);
```

Parameters

wParam

[in] Specifies additional message-dependent information.

lParam

[in] Specifies additional message-dependent information.

Return Value

Nonzero if the message should not be forwarded to the next property page; otherwise zero.

Remarks

If a page returns a nonzero value, the property sheet does not send the message to subsequent pages.

See Also

Reference

CSnapInPropertyPageImpl Class

Other Resources

CSnapInPropertyPageImpl Members

CSnapInPropertyPageImpl::SetModified

Call this member function to enable or disable the **Apply Now** button, based on whether the settings in the property page should be applied to the appropriate external object.

```
void SetModified(
   BOOL bChanged = TRUE
);
```

Parameters

bChanged

[in] **TRUE** to indicate that the property page settings have been modified since the last time they were applied; **FALSE** to indicate that the property page settings have been applied, or should be ignored.

Remarks

The property sheet keeps track of which pages are "dirty," that is, property pages for which you have called **SetModified(TRUE)**. The **Apply Now** button will always be enabled if you call **SetModified(TRUE)** for one of the pages. The **Apply Now** button will be disabled when you call **SetModified(FALSE)** for one of the pages, but only if none of the other pages is "dirty."

See Also

Reference

CSnapInPropertyPageImpl Class

Other Resources

CSnapInPropertyPageImpl Members

CSnapInPropertyPageImpl Data Members

For information about the data members in **CSnapInPropertyPageImpl**, see CSnapInPropertyPageImpl Members.

CSnapInPropertyPageImpl::m_psp

m_psp is a structure whose members store the characteristics of **PROPSHEETPAGE**.

```
PROPSHEETPAGE m_psp;
```

Remarks

Use this structure to initialize the appearance of a property page after it is constructed.

For more information on this structure, including a listing of its members, see PROPSHEETPAGE in the *Platform SDK*.

See Also

Reference

CSnapInPropertyPageImpl Class

Other Resources

CSnapInPropertyPageImpl Members

CSocketAddr Class

This class provides methods for converting host names to host addresses, supporting both IPv4 and IPV6 formats.

```
class CSocketAddr
```

Remarks

This class provides an IP version agnostic approach for looking up network addresses for use with Windows sockets API functions and socket wrappers in libraries.

The members of this class that are used to look up network addresses use the Win32 API function getaddrinfo.

This class supports both IPv4 andIPv6 network addresses.

Requirements

Header: atlsocket.h

See Also

Other Resources

ATL Class Overview

CSocketAddr Members

CSocketAddr Members

Methods

CSocketrAddr	The constructor.
FindAddr	Call this method to convert the provided host name to the host address.
FindINET4Addr	Call this method to convert the IPv4 host name to the host address.
FindINET6Addr	Call this method to convert the IPv6 host name to the host address.
GetAddrInfoList	Call this method to return a pointer to the **addrinfo** list.
GetAddrInfo	Call this method to return a pointer to a specific element in the **addrinfo** list.

See Also

Reference

CSocketAddr Class

CSocketAddr Methods

For information about the methods in **CSocketAddr**, see CSocketAddr Members.

CSocketAddr::CSocketAddr

The constructor.

```
CSocketAddr( );
```

Remarks

Creates a new **CSocketAddr** object and initializes the linked list containing response information about the host.

See Also

Reference

CSocketAddr Class

Other Resources

CSocketAddr Members

CSocketAddr::FindAddr

Call this method to convert the provided host name to the host address.

```
int FindAddr(
   const char *szHost,
   const char *szPortOrServiceName,
   int flags,
   int addr_family,
   int sock_type,
   int ai_proto
   );
int FindAddr(
   const char *szHost,
   int nPortNo,
   int flags,
   int addr_family,
   int sock_type,
   int ai_proto
   );
```

Parameters

szHost

The host name or dotted IP address.

szPortOrServiceName

The port number or name of service on host.

nPortNo

The port number.

flags

0 or combination of AI_PASSIVE, AI_CANONNAME or AI_NUMERICHOST.

addr_family

Address family (such as PF_INET).

sock_type

Socket type (such as SOCK_STREAM).

ai_proto

Protocol (such as IPPROTO_IP or IPPROTO_IPV6).

Return Value

Returns zero if the address is calculated successfully. Returns a nonzero Windows Socket error code on failure. If successful, the calculated address is stored in a linked list that may be referenced using **CSocketAddr::GetAddrInfoList** and **CSocketAddr::GetAddrInfo**.

Remarks

The host name parameter may be in either IPv4 or IPv6 format. This method calls the Win32 API function getaddrinfo to perform the conversion.

See Also

Reference

CSocketAddr Class
getaddrinfo
CSocketAddr::GetAddrInfoList
CSocketAddr::GetAddrInfo

Other Resources

CSocketAddr Members

CSocketAddr::FindINET4Addr

Call this method to convert the IPv4 host name to the host address.

```
int FindINET4Addr(
   const char *szHost,
   int nPortNo,
   int flags,
   int sock_type,
);
```

Parameters

szHost

The host name or dotted IP address.

nPortNo

The port number.

flags

0 or combination of AI_PASSIVE, AI_CANONNAME or AI_NUMERICHOST.

sock_type

Socket type (such as SOCK_STREAM).

Return Value

Returns zero if the address is calculated successfully. Returns a nonzero Windows Socket error code on failure. If successful, the calculated address is stored in a linked list that may be referenced using **CSocketAddr::GetAddrInfoList** and **CSocketAddr::GetAddrInfo**.

Remarks

This method calls the Win32 API function getaddrinfo to perform the conversion.

See Also

Reference

CSocketAddr Class

CSocketAddr::FindINET6Addr

Other Resources

CSocketAddr Members

CSocketAddr::FindINET6Addr

Call this method to convert the IPv6 host name to the host address.

```
int FindINET6Addr(
   const char *szHost,
   int nPortNo,
   int flags,
   int sock_type,
);
```

Parameters

szHost

The host name or dotted IP address.

nPortNo

The port number.

flags

0 or combination of AI_PASSIVE, AI_CANONNAME or AI_NUMERICHOST.

sock_type

Socket type (such as SOCK_STREAM).

Return Value

Returns zero if the address is calculated successfully. Returns a nonzero Windows Socket error code on failure. If successful, the calculated address is stored in a linked list that may be referenced using **CSocketAddr::GetAddrInfoList** and **CSocketAddr::GetAddrInfo**.

Remarks

This method calls the Win32 API function getaddrinfo to perform the conversion.

See Also

Reference

CSocketAddr Class

CSocketAddr::FindINET4Addr

Other Resources

CSocketAddr Members

CSocketAddr::GetAddrInfo

Call this method to return a pointer to a specific element in the **addrinfo** list.

```
addrinfo* const GetAddrInfo(
   int nIndex = 0
) const;
```

Parameters

nIndex

A reference to a specific element in the addrinfo list.

Return Value

Returns a pointer to the **addrinfo** structure referenced by *nIndex* in the linked list containing response information about the host.

See Also

Reference

CSocketAddr Class

CSocketAddr::GetAddrInfoList

Other Resources

CSocketAddr Members

CSocketAddr::GetAddrInfoList

Call this method to return a pointer to the **addrinfo** list.

```
addrinfo* const GetAddrInfoList( ) const;
```

Return Value

Pointer to a linked list of one or more addrinfo structures containing response information about the host.

See Also

Reference

CSocketAddr Class

CSocketAddr::GetAddrInfo

Other Resources

CSocketAddr Members

CStringElementTraits Class

This class provides static functions used by collection classes storing **CString** objects.

```
template<
   typename T
>
class CStringElementTraits
```

Parameters

T

The type of data to be stored in the collection.

Remarks

This class provides static functions for copying, moving, and comparing strings and for creating a hash value. These functions are useful when using a collection class to store string-based data. Use CStringElementTraitsI when case-insensitive comparisons are required. Use CStringRefElementTraits when the string objects are to be dealt with as references.

For more information, see ATL Collection Classes.

Requirements

Header: cstringt.h

See Also

Reference

CElementTraitsBase Class
CStringElementTraitsI Class

Other Resources

CStringElementTraits Members
ATL Class Overview

CStringElementTraits Members

Static Functions

CompareElements	Call this static function to compare two string elements for equality.
CompareElementsOrdered	Call this static function to compare two string elements.
CopyElements	Call this static function to copy **CString** elements stored in a collection class object.
Hash	Call this static function to calculate a hash value for the given string element.
RelocateElements	Call this static function to relocate **CString** elements stored in a collection class object.

Typedefs

INARGTYPE	The data type to use for adding elements to the collection class object.
OUTARGTYPE	The data type to use for retrieving elements from the collection class object.

See Also

Reference

CStringElementTraits Class

CStringElementTraits Static Functions

For information about the static functions in **CStringElementTraits**, see CStringElementTraits Members.

CStringElementTraits::CompareElements

Call this static function to compare two string elements for equality.

```
static bool CompareElements(
   INARGTYPE str1,
   INARGTYPE str2
);
```

Parameters

str1

The first string element.

str2

The second string element.

Return Value

Returns true if the elements are equal, false otherwise.

See Also

Reference

CStringElementTraits Class
CStringElementTraits::CompareElementsOrdered
CStringElementTraitsI::CompareElements

Other Resources

CStringElementTraits Members

CStringElementTraits::CompareElementsOrdered

Call this static function to compare two string elements.

```
static int CompareElementsOrdered(
   INARGTYPE str1,
   INARGTYPE str2
);
```

Parameters

str1

The first string element.

str2

The second string element.

Return Value

Zero if the strings are identical, < 0 if *str1* is less than *str2*, or > 0 if *str1* is greater than *str2*. The CStringT::Compare method is used to perform the comparisons.

See Also

Reference

CStringElementTraits Class
CStringElementTraits::CompareElements
CStringElementTraitsI::CompareElementsOrdered

Other Resources

CStringElementTraits Members

CStringElementTraits::CopyElements

Call this static function to copy **CString** elements stored in a collection class object.

```
static void CopyElements(
   T* pDest,
   const T* pSrc,
   size_t nElements
);
```

Parameters

pDest

Pointer to the first element that will receive the copied data.

pSrc

Pointer to the first element to copy.

nElements

The number of elements to copy.

Remarks

The source and destination elements should not overlap.

See Also

Reference

CStringElementTraits Class

CStringElementTraits::RelocateElements

Other Resources

CStringElementTraits Members

CStringElementTraits::Hash

Call this static function to calculate a hash value for the given string element.

```
static ULONG Hash(
   INARGTYPE str
);
```

Parameters

str

The string element.

Return Value

Returns a hash value, calculated using the string's contents.

See Also

Reference

CStringElementTraits Class

CStringElementTraitsI::Hash

Other Resources

CStringElementTraits Members

CStringElementTraits::RelocateElements

Call this static function to relocate **CString** elements stored in a collection class object.

```
static void RelocateElements(
   T* pDest,
   T* pSrc,
   size_t nElements
);
```

Parameters

pDest

Pointer to the first element that will receive the relocated data.

pSrc

Pointer to the first element to relocate.

nElements

The number of elements to relocate.

Remarks

This static function calls memmove, which is sufficient for most data types. If the objects being moved contain pointers to their own members, this static function will need to be overridden.

See Also

Reference

CStringElementTraits Class

CStringElementTraits::CopyElements

Other Resources

CStringElementTraits Members

CStringElementTraitsTypedefs

For information about the typedefs in **CStringElementTraits**, see CStringElementTraits Members.

CStringElementTraits::INARGTYPE

The data type to use for adding elements to the collection class object.

```
typedef T::PCXSTR INARGTYPE;
```

See Also

Reference

CStringElementTraits Class

CStringElementTraits::OUTARGTYPE

Other Resources

CStringElementTraits Members

CStringElementTraits::OUTARGTYPE

The data type to use for retrieving elements from the collection class object.

```
typedef T & OUTARGTYPE;
```

See Also

Reference

CStringElementTraits Class

CStringElementTraits::INARGTYPE

Other Resources

CStringElementTraits Members

CStringElementTraitsI Class

This class provides static functions related to strings stored in collection class objects. It is similar to CStringElementTraits, but performs case-insensitive comparisons.

```
template<
   typename T,
   class CharTraits = CDefaultCharTraits< T::XCHAR >
>
class CStringElementTraitsI : public CElementTraitsBase< T >
```

Parameters

T

The type of data to be stored in the collection.

Remarks

This class provides static functions for comparing strings and for creating a hash value. These functions are useful when using a collection class to store string-based data. Use CStringRefElementTraits when the string objects are to be with dealt with as references.

For more information, see ATL Collection Classes.

Requirements

Header: atlcoll.h

See Also

Reference

CElementTraitsBase Class

CStringElementTraits Class

Other Resources

CStringElementTraitsI Members

ATL Class Overview

CStringElementTraitsI Members

Static Functions

CompareElements	Call this static function to compare two string elements for equality, ignoring differences in case.
CompareElementsOrdered	Call this static function to compare two string elements, ignoring differences in case.
Hash	Call this static function to calculate a hash value for the given string element.

Typedefs

INARGTYPE	The data type to use for adding elements to the collection class object.
OUTARGTYPE	The data type to use for retrieving elements from the collection class object.

See Also

Reference

CStringElementTraitsI Class

Other Resources

CElementTraitsBase Members

CStringElementTraitsI Static Functions

For information about the static functions in **CStringElementTraitsI**, see CStringElementTraitsI Members.

CStringElementTraitsI::CompareElements

Call this static function to compare two string elements for equality, ignoring differences in case.

```
static bool CompareElements(
   INARGTYPE str1,
   INARGTYPE str2
) throw( );
```

Parameters

str1

The first string element.

str2

The second string element.

Return Value

Returns true if the elements are equal, false otherwise.

Remarks

Comparisons are case insensitive.

See Also

Reference

CStringElementTraitsI Class
CStringElementTraitsI::CompareElementsOrdered
CStringElementTraits::CompareElements

Other Resources

CStringElementTraitsI Members

CStringElementTraitsI::CompareElementsOrdered

Call this static function to compare two string elements, ignoring differences in case.

```
static int CompareElementsOrdered(
   INARGTYPE str1,
   INARGTYPE str2
) throw( );
```

Parameters

str1

The first string element.

str2

The second string element.

Return Value

Zero if the strings are identical, < 0 if *str1* is less than *str2*, or > 0 if *str1* is greater than *str2*. The CStringT::Compare method is used to perform the comparisons.

Remarks

Comparisons are case insensitive.

See Also

Reference

CStringElementTraitsI Class
CStringElementTraitsI::CompareElements
CStringElementTraits::CompareElementsOrdered

Other Resources

CStringElementTraitsI Members

CStringElementTraitsI::Hash

Call this static function to calculate a hash value for the given string element.

```
static ULONG Hash(
   INARGTYPE str
) throw( );
```

Parameters

str

The string element.

Return Value

Returns a hash value, calculated using the string's contents.

See Also

Reference

CStringElementTraitsI Class

CStringElementTraits::Hash

Other Resources

CStringElementTraitsI Members

CStringElementTraitsI Typedefs

For information about the typedefs in **CStringElementTraitsI**, see CStringElementTraitsI Members.

CStringElementTraitsI::INARGTYPE

The data type to use for adding elements to the collection class object.

```
typedef T::PCXSTR INARGTYPE;
```

See Also

Reference

CStringElementTraitsI Class

CStringElementTraitsI::OUTARGTYPE

Other Resources

CStringElementTraitsI Members

CStringElementTraitsI::OUTARGTYPE

The data type to use for retrieving elements from the collection class object.

```
typedef T & OUTARGTYPE;
```

See Also

Reference

CStringElementTraitsI Class

CStringElementTraitsI::INARGTYPE

Other Resources

CStringElementTraitsI Members

CStringRefElementTraits Class

This class provides static functions related to strings stored in collection class objects. The string objects are dealt with as references.

```
template<
   typename T
>
class CStringRefElementTraits : public CElementTraitsBase< T >
```

Parameters

T

The type of data to be stored in the collection.

Remarks

This class provides static functions for comparing strings and for creating a hash value. These functions are useful when using a collection class to store string-based data. Unlike CStringElementTraits and CStringElementTraitsI, **CStringRefElementTraits** causes the **CString** arguments to be passed as **const CString&** references.

For more information, see ATL Collection Classes.

Requirements

Header: atlcoll.h

See Also

Reference

CElementTraitsBase Class

Other Resources

CStringRefElementTraits Members

ATL Class Overview

CStringRefElementTraits Members

Static Functions

CompareElements	Call this static function to compare two string elements for equality.
CompareElementsOrdered	Call this static function to compare two string elements.
Hash	Call this static function to calculate a hash value for the given string element.

See Also

Reference

CStringRefElementTraits Class

Other Resources

CElementTraitsBase Members

CStringRefElementTraits Static Functions

For information about the static functions in **CStringRefElementTraits**, see CStringRefElementTraits Members.

CStringRefElementTraits::CompareElements

Call this static function to compare two string elements for equality.

```
static bool CompareElements(
   INARGTYPE element1,
   INARGTYPE element2
) throw( );
```

Parameters

element1

The first string element.

element2

The second string element.

Return Value

Returns true if the elements are equal, false otherwise.

See Also

Reference

CStringRefElementTraits Class

CStringRefElementTraits::CompareElementsOrdered

Other Resources

CStringRefElementTraits Members

CStringRefElementTraits::CompareElementsOrdered

Call this static function to compare two string elements.

```
static int CompareElementsOrdered(
   INARGTYPE str1,
   INARGTYPE str2
) throw( );
```

Parameters

str1

The first string element.

str2

The second string element.

Return Value

Zero if the strings are identical, < 0 if *str1* is less than *str2*, or > 0 if *str1* is greater than *str2*. The CStringT::Compare method is used to perform the comparisons.

See Also

Reference

CStringRefElementTraits Class

CStringRefElementTraits::CompareElements

Other Resources

CStringRefElementTraits Members

CStringRefElementTraits::Hash

Call this static function to calculate a hash value for the given string element.

```
static ULONG Hash(
   INARGTYPE str
) throw( );
```

Parameters

str

The string element.

Return Value

Returns a hash value, calculated using the string's contents.

See Also

Reference

CStringRefElementTraits Class

Other Resources

CStringRefElementTraits Members

CStockPropImpl Class

This class provides methods for supporting stock property values.

```
template <
   class T,
   class InterfaceName,
   const IID* piid = &_ATL_IIDOF(InterfaceName),
   const GUID* plibid = &CComModule::m_libid,
   WORD wMajor = 1,
   WORD wMinor = 0,
   class tihclass = CcomTypeInfoHolder
>
class ATL_NO_VTABLE CStockPropImpl :
   public IDispatchImpl< InterfaceName, piid, plibid, wMajor,
      wMinor, tihclass >
```

Parameters

T

The class implementing the control and deriving from **CStockPropImpl**.

InterfaceName

A dual interface exposing the stock properties.

piid

A pointer to the IID of *InterfaceName*.

plibid

A pointer to the LIBID of the type library containing the definition of *InterfaceName*.

wMajor

The major version of the type library. The default value is 1.

wMinor

The minor version of the type library. The default value is 0.

tihclass

The class used to manage the type information for *T*. The default value is **CComTypeInfoHolder**.

Remarks

CStockPropImpl provides **put** and **get** methods for each stock property. These methods provide the code necessary to set or get the data member associated with each property and to notify and synchronize with the container when any property changes.

Visual C++ provides support for stock properties through its wizards. For more information about adding stock properties to a control, see the ATL Tutorial.

For backward compatibility, **CStockPropImpl** also exposes **get_Window** and **put_Window** methods that simply call **get_HWND** and **put_HWND**, respectively. The default implementation of **put_HWND** returns **E_FAIL** since **HWND** should be a read-only property.

The following properties also have a **putref** implementation:

- Font
- MouseIcon
- Picture

The same three stock properties require their corresponding data member to be of type **CComPtr** or some other class that

provides correct interface reference counting by means of the assignment operator.

Requirements

Header: atlctl.h

See Also

Reference

IDispatchImpl Class

Other Resources

ATL Class Overview

CStockPropImpl Members

Methods

get_Appearance	Call this method to get the paint style used by the control, for example, flat or 3D.
get_AutoSize	Call this method to get the status of the flag that indicates if the control cannot be any other size.
get_BackColor	Call this method to get the control's background color.
get_BackStyle	Call this method to get the control's background style, either transparent or opaque.
get_BorderColor	Call this method to get the control's border color.
get_BorderStyle	Call this method to get the control's border style.
get_BorderVisible	Call this method to get the status of the flag that indicates if the control's border is visible or not.
get_BorderWidth	Call this method to get the width (in pixels) of the control's border.
get_Caption	Call this method to get the text specified in an object's caption.
get_DrawMode	Call this method to get the control's drawing mode, for example, XOR Pen or Invert Colors.
get_DrawStyle	Call this method to get the control's drawing style, for example, solid, dashed, or dotted.
get_DrawWidth	Call this method to get the drawing width (in pixels) used by the control's drawing methods.
get_Enabled	Call this method to get the status of the flag that indicates if the control is enabled.
get_FillColor	Call this method to get the control's fill color.
get_FillStyle	Call this method to get the control's fill style, for example, solid, transparent, or cross-hatched.
get_Font	Call this method to get a pointer to the control's font properties.
get_ForeColor	Call this method to get the control's foreground color.
get_HWND	Call this method to get the window handle associated with the control.
get_MouseIcon	Call this method to get the picture properties of the graphic (icon, bitmap, or metafile) to be displayed when the mouse is over the control.
get_MousePointer	Call this method to get the type of mouse pointer displayed when the mouse is over the control, for exampl e, arrow, cross, or hourglass.
get_Picture	Call this method to get a pointer to the picture properties of a graphic (icon, bitmap, or metafile) to be displ ayed.
get_ReadyState	Call this method to get the control's ready state, for example, loading or loaded.
get_TabStop	Call this method to get the flag that indicates if the control is a tab stop or not.

get_Text	Call this method to get the text that is displayed with the control.
get_Valid	Call this method to get the status of the flag that indicates if the control is valid or not.
get_Window	Call this method to get the window handle associated with the control. Identical to CStockPropImpl::get_HWND.
put_Appearance	Call this method to set the paint style used by the control, for example, flat or 3D.
put_AutoSize	Call this method to set the value of the flag that indicates if the control cannot be any other size.
put_BackColor	Call this method to set the control's background color.
put_BackStyle	Call this method to set the control's background style.
put_BorderColor	Call this method to set the control's border color.
put_BorderStyle	Call this method to set the control's border style.
put_BorderVisible	Call this method to set the value of the flag that indicates if the control's border is visible or not.
put_BorderWidth	Call this method to set the width of the control's border.
put_Caption	Call this method to set the text to be displayed with the control.
put_DrawMode	Call this method to set the control's drawing mode, for example, XOR Pen or Invert Colors.
put_DrawStyle	Call this method to set the control's drawing style, for example, solid, dashed, or dotted.
put_DrawWidth	Call this method to set the width (in pixels) used by the control's drawing methods.
put_Enabled	Call this method to set the flag that indicates if the control is enabled.
put_FillColor	Call this method to set the control's fill color.
put_FillStyle	Call this method to set the control's fill style, for example, solid, transparent, or cross-hatched.
put_Font	Call this method to set the control's font properties.
put_ForeColor	Call this method to set the control's foreground color.
put_HWND	This method returns E_FAIL.
put_MouseIcon	Call this method to set the picture properties of the graphic (icon, bitmap, or metafile) to be displayed when the mouse is over the control.
put_MousePointer	Call this method to set the type of mouse pointer displayed when the mouse is over the control, for example, arrow, cross, or hourglass.
put_Picture	Call this method to set the picture properties of a graphic (icon, bitmap, or metafile) to be displayed.
put_ReadyState	Call this method to set the control's ready state, for example, loading or loaded.

put_TabStop	Call this method to set the value of the flag that indicates if the control is a tab stop or not.
put_Text	Call this method to set the text that is displayed with the control.
put_Valid	Call this method to set the flag that indicates if the control is valid or not.
put_Window	This method calls CStockPropImpl::put_HWND, which returns E_FAIL.
putref_Font	Call this method to set the control's font properties, with a reference count.
putref_MouseIcon	Call this method to set the picture properties of the graphic (icon, bitmap, or metafile) to be displayed when the mouse is over the control, with a reference count.
putref_Picture	Call this method to set the picture properties of a graphic (icon, bitmap, or metafile) to be displayed, with a reference count.

See Also

Reference

CStockPropImpl Class

CStockPropImpl Methods

For information about the methods in **CStockPropImpl**, see CStockPropImpl Members.

CStockPropImpl::get_Appearance

Call this method to get the paint style used by the control, for example, flat or 3D.

```
HRESULT STDMETHODCALLTYPE get_Appearance(
   SHORT pnAppearance
);
```

Parameters

pnAppearance

Variable that receives the control's paint style.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_Appearance

Other Resources

CStockPropImpl Members

CStockPropImpl::get_AutoSize

Call this method to get the status of the flag that indicates if the control cannot be any other size.

```
HRESULT STDMETHODCALLTYPE get_Autosize(
   VARIANT_BOOL * pbAutoSize
);
```

Parameters

pbAutoSize

Variable that receives the flag status. TRUE indicates that the control cannot be any other size.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_AutoSize

Other Resources

CStockPropImpl Members

CStockPropImpl::get_BackColor

Call this method to get the control's background color.

```
HRESULT STDMETHODCALLTYPE get_BackColor(
   OLE_COLOR * pclrBackColor
);
```

Parameters

pclrBackColor

Variable that receives the control's background color.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_BackColor

Other Resources

CStockPropImpl Members

CStockPropImpl::get_BackStyle

Call this method to get the control's background style, either transparent or opaque.

```
HRESULT STDMETHODCALLTYPE get_BackStyle(
   LONG * pnBackStyle
);
```

Parameters

pnBackStyle

Variable that receives the control's background style.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_BackStyle

Other Resources

CStockPropImpl Members

CStockPropImpl::get_BorderColor

Call this method to get the control's border color.

```
HRESULT STDMETHODCALLTYPE get_BorderColor(
   OLE_COLOR * pclrBorderColor
);
```

Parameters

pclrBorderColor

Variable that receives the control's border color.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_BorderColor

Other Resources

CStockPropImpl Members

CStockPropImpl::get_BorderStyle

Call this method to get the control's border style.

```
HRESULT STDMETHODCALLTYPE get_BorderStyle(
   LONG * pnBorderStyle
);
```

Parameters

pnBorderStyle

Variable that receives the control's border style.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_BorderStyle

Other Resources

CStockPropImpl Members

CStockPropImpl::get_BorderVisible

Call this method to get the status of the flag that indicates if the control's border is visible or not.

```
HRESULT STDMETHODCALLTYPE get_BorderVisible(
   VARIANT_BOOL * pbBorderVisible
);
```

Parameters

pbBorderVisible

Variable that receives the flag status. TRUE indicates that the control's border is visible.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_BorderVisible

Other Resources

CStockPropImpl Members

CStockPropImpl::get_BorderWidth

Call this method to get the width of the control's border.

```
HRESULT STDMETHODCALLTYPE get_BorderWidth(
   LONG * pnBorderWidth
);
```

Parameters

pnBorderWidth

Variable that receives the control's border width.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_BorderWidth

Other Resources

CStockPropImpl Members

CStockPropImpl::get_Caption

Call this method to get the text specified in an object's caption.

```
HRESULT STDMETHODCALLTYPE get_Caption(
   BSTR * pbstrCaption
);
```

Parameters

pbstrCaption

The text to be displayed with the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_Caption

Other Resources

CStockPropImpl Members

CStockPropImpl::get_DrawMode

Call this method to get the control's drawing mode, for example, XOR Pen or Invert Colors.

```
HRESULT STDMETHODCALLTYPE get_DrawMode(
   LONG * pnDrawMode
);
```

Parameters

pnDrawMode

Variable that receives the control's drawing mode.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_DrawMode

Other Resources

CStockPropImpl Members

CStockPropImpl::get_DrawStyle

Call this method to get the control's drawing style, for example, solid, dashed, or dotted.

```
HRESULT STDMETHODCALLTYPE get_DrawStyle(
   LONG * pnDrawStyle
);
```

Parameters

pnDrawStyle

Variable that receives the control's drawing style.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_DrawStyle

Other Resources

CStockPropImpl Members

CStockPropImpl::get_DrawWidth

Call this method to get the drawing width (in pixels) used by the control's drawing methods.

```
HRESULT STDMETHODCALLTYPE get_DrawWidth(
   LONG * pnDrawWidth
);
```

Parameters

pnDrawWidth

Variable that receives the control's width value, in pixels.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_DrawWidth

Other Resources

CStockPropImpl Members

CStockPropImpl::get_Enabled

Call this method to get the status of the flag that indicates if the control is enabled.

```
HRESULT STDMETHODCALLTYPE get_Enabled(
   VARIANT_BOOL * pbEnabled
);
```

Parameters

pbEnabled

Variable that receives the flag status. TRUE indicates that the control is enabled.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_Enabled

Other Resources

CStockPropImpl Members

CStockPropImpl::get_FillColor

Call this method to get the control's fill color.

```
HRESULT STDMETHODCALLTYPE get_FillColor(
   OLE_COLOR * pclrFillColor
);
```

Parameters

pclrFillColor

Variable that receives the control's fill color.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_FillColor

Other Resources

CStockPropImpl Members

CStockPropImpl::get_FillStyle

Call this method to get the control's fill style, for example, solid, transparent, or crosshatched.

```
HRESULT STDMETHODCALLTYPE get_FillStyle(
   LONG * pnFillStyle
);
```

Parameters

pnFillStyle

Variable that receives the control's fill style.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_FillStyle

Other Resources

CStockPropImpl Members

CStockPropImpl::get_Font

Call this method to get a pointer to the control's font properties.

```
HRESULT STDMETHODCALLTYPE get_Font(
   IFontDisp** ppFont
);
```

Parameters

ppFont

Variable that receives a pointer to the control's font properties.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_Font

Other Resources

CStockPropImpl Members

CStockPropImpl::get_ForeColor

Call this method to get the control's foreground color.

```
HRESULT STDMETHODCALLTYPE get_ForeColor(
   OLE_COLOR * pclrForeColor
);
```

Parameters

pclrForeColor

Variable that receives the controls foreground color.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_ForeColor

Other Resources

CStockPropImpl Members

CStockPropImpl::get_HWND

Call this method to get the window handle associated with the control.

```
HRESULT STDMETHODCALLTYPE get_HWND(
   LONG_PTR* phWnd
);
```

Parameters

phWnd

The window handle associated with the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_HWND

Other Resources

CStockPropImpl Members

CStockPropImpl::get_MouseIcon

Call this method to get the picture properties of the graphic (icon, bitmap, or metafile) to be displayed when the mouse is over the control.

```
HRESULT STDMETHODCALLTYPE get_MouseIcon(
   IPictureDisp** ppPicture
);
```

Parameters

ppPicture

Variable that receives a pointer to the picture properties of the graphic.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_MouseIcon

Other Resources

CStockPropImpl Members

CStockPropImpl::get_MousePointer

Call this method to get the type of mouse pointer displayed when the mouse is over the control, for example, arrow, cross, or hourglass.

```
HRESULT STDMETHODCALLTYPE get_MousePointer(
   LONG * pnMousePointer
);
```

Parameters

pnMousePointer

Variable that receives the type of mouse pointer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_MousePointer

Other Resources

CStockPropImpl Members

CStockPropImpl::get_Picture

Call this method to get a pointer to the picture properties of a graphic (icon, bitmap, or metafile) to be displayed.

```
HRESULT STDMETHODCALLTYPE get_Picture(
   IPictureDisp** ppPicture
);
```

Parameters

ppPicture

Variable that receives a pointer to the picture's properties. See IPictureDisp for more details.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_Picture

Other Resources

CStockPropImpl Members

CStockPropImpl::get_ReadyState

Call this method to get the control's ready state, for example, loading or loaded.

```
HRESULT STDMETHODCALLTYPE get_ReadyState(
   LONG * pnReadyState
);
```

Parameters

pnReadyState

Variable that receives the control's ready state.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_ReadyState

Other Resources

CStockPropImpl Members

CStockPropImpl::get_TabStop

Call this method to get the status of the flag that indicates if the control is a tab stop or not.

```
HRESULT STDMETHODCALLTYPE get_TabStop(
   VARIANT_BOOL * pbTabStop
);
```

Parameters

pbTabStop

Variable that receives the flag status. TRUE indicates that the control is a tab stop.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_TabStop

Other Resources

CStockPropImpl Members

CStockPropImpl::get_Text

Call this method to get the text that is displayed with the control.

```
HRESULT STDMETHODCALLTYPE get_Text(
   BSTR * pbstrText
);
```

Parameters

pbstrText

The text that is displayed with the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_Text

Other Resources

CStockPropImpl Members

CStockPropImpl::get_Valid

Call this method to get the status of the flag that indicates if the control is valid or not.

```
HRESULT STDMETHODCALLTYPE get_Valid(
   VARIANT_BOOL * pbValid
);
```

Parameters

pbValid

Variable that receives the flag status. TRUE indicates that the control is valid.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_Valid

Other Resources

CStockPropImpl Members

CStockPropImpl::get_Window

Call this method to get the window handle associated with the control. Identical to CStockPropImpl::get_HWND.

```
HRESULT STDMETHODCALLTYPE get_Window(
   LONG_PTR* phWnd
);
```

Parameters

phWnd

The window handle associated with the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_Window

Other Resources

CStockPropImpl Members

CStockPropImpl::put_Appearance

Call this method to set the paint style used by the control, for example, flat or 3D.

```
HRESULT STDMETHODCALLTYPE put_Appearance(
   SHORT nAppearance
);
```

Parameters

nAppearance

The new paint style to be used by the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_Appearance

Other Resources

CStockPropImpl Members

CStockPropImpl::put_AutoSize

Call this method to set the value of flag that indicates if the control cannot be any other size.

```
HRESULT STDMETHODCALLTYPE put_AutoSize(
   VARIANT_BOOL bAutoSize,
);
```

Parameters

bAutoSize

TRUE if the control cannot be any other size.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_AutoSize

Other Resources

CStockPropImpl Members

CStockPropImpl::put_BackColor

Call this method to set the control's background color.

```
HRESULT STDMETHODCALLTYPE put_BackColor(
   OLE_COLOR clrBackColor
);
```

Parameters

clrBackColor

The new control background color.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_BackColor

Other Resources

CStockPropImpl Members

CStockPropImpl::put_BackStyle

Call this method to set the control's background style.

```
HRESULT STDMETHODCALLTYPE put_BackStyle(
   LONG nBackStyle
);
```

Parameters

nBackStyle

The new control background style.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_BackStyle

Other Resources

CStockPropImpl Members

CStockPropImpl::put_BorderColor

Call this method to set the control's border color.

```
HRESULT STDMETHODCALLTYPE put_BorderColor(
   OLE_COLOR clrBorderColor
);
```

Parameters

clrBorderColor

The new border color. The OLE_COLOR data type is internally represented as a 32-bit long integer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_BorderColor

Other Resources

CStockPropImpl Members

CStockPropImpl::put_BorderStyle

Call this method to set the control's border style.

```
HRESULT STDMETHODCALLTYPE put_BorderStyle(
   LONG nBorderStyle
);
```

Parameters

nBorderStyle

The new border style.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_BorderStyle

Other Resources

CStockPropImpl Members

CStockPropImpl::put_BorderVisible

Call this method to set the value of the flag that indicates if the control's border is visible or not.

```
HRESULT STDMETHODCALLTYPE put_BorderVisible(
   VARIANT_BOOL bBorderVisible
);
```

Parameters

bBorderVisible

TRUE if the border is to be visible.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_BorderVisible

Other Resources

CStockPropImpl Members

CStockPropImpl::put_BorderWidth

Call this method to set the width of the control's border.

```
HRESULT STDMETHODCALLTYPE put_BorderWidth(
   LONG nBorderWidth
);
```

Parameters

nBorderWidth

The new width of the control's border.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_BorderWidth

Other Resources

CStockPropImpl Members

CStockPropImpl::put_Caption

Call this method to set the text to be displayed with the control.

```
HRESULT STDMETHODCALLTYPE put_Caption(
   BSTR bstrCaption
);
```

Parameters

bstrCaption

The text to be displayed with the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_Caption

Other Resources

CStockPropImpl Members

CStockPropImpl::put_DrawMode

Call this method to set the control's drawing mode, for example, XOR Pen or Invert Colors.

```
HRESULT STDMETHODCALLTYPE put_DrawMode(
   LONG nDrawMode
);
```

Parameters

nDrawMode

The new drawing mode for the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_DrawMode

Other Resources

CStockPropImpl Members

CStockPropImpl::put_DrawStyle

Call this method to set the control's drawing style, for example, solid, dashed, or dotted.

```
HRESULT STDMETHODCALLTYPE put_DrawStyle(
   LONG pnDrawStyle
);
```

Parameters

nDrawStyle

The new drawing style for the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_DrawStyle

Other Resources

CStockPropImpl Members

CStockPropImpl::put_DrawWidth

Call this method to set the width (in pixels) used by the control's drawing methods.

```
HRESULT STDMETHODCALLTYPE put_DrawWidth(
   LONG nDrawWidth
);
```

Parameters

nDrawWidth

The new width to be used by the control's drawing methods.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_DrawWidth

Other Resources

CStockPropImpl Members

CStockPropImpl::put_Enabled

Call this method to set the value of the flag that indicates if the control is enabled.

```
HRESULT STDMETHODCALLTYPE put_Enabled(
   VARIANT_BOOL bEnabled
);
```

Parameters

bEnabled

TRUE if the control is enabled.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_Enabled

Other Resources

CStockPropImpl Members

CStockPropImpl::put_FillColor

Call this method to set the control's fill color.

```
HRESULT STDMETHODCALLTYPE put_FillColor(
   OLE_COLOR clrFillColor
);
```

Parameters

clrFillColor

The new fill color for the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_FillColor

Other Resources

CStockPropImpl Members

CStockPropImpl::put_FillStyle

Call this method to set the control's fill style, for example, solid, transparent, or cross-hatched.

```
HRESULT STDMETHODCALLTYPE put_FillStyle(
   LONG nFillStyle
);
```

Parameters

nFillStyle

The new fill style for the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_FillStyle

Other Resources

CStockPropImpl Members

CStockPropImpl::put_Font

Call this method to set the control's font properties.

```
HRESULT STDMETHODCALLTYPE put_Font(
   IFontDisp* pFont
);
```

Parameters

pFont

A pointer to the control's font properties.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class
CStockPropImpl::get_Font
CStockPropImpl::putref_Font

Other Resources

CStockPropImpl Members

CStockPropImpl::put_ForeColor

Call this method to set the control's foreground color.

```
HRESULT STDMETHODCALLTYPE put_ForeColor(
   OLE_COLOR clrForeColor
);
```

Parameters

clrForeColor

The new foreground color of the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_ForeColor

Other Resources

CStockPropImpl Members

CStockPropImpl::put_HWND

This method returns E_FAIL.

```
HRESULT STDMETHODCALLTYPE put_HWND(
   LONG_PTR /* hWnd */
);
```

Parameters

/ hWnd */*

Reserved.

Return Value

Returns E_FAIL.

Remarks

The window handle is a read-only value.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_HWND

Other Resources

CStockPropImpl Members

CStockPropImpl::put_MouseIcon

Call this method to set the picture properties of the graphic (icon, bitmap, or metafile) to be displayed when the mouse is over the control.

```
HRESULT STDMETHODCALLTYPE put_MouseIcon(
   IPictureDisp* pPicture
);
```

Parameters

pPicture

A pointer to the picture properties of the graphic.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class
CStockPropImpl::get_MouseIcon
CStockPropImpl::putref_MouseIcon

Other Resources

CStockPropImpl Members

CStockPropImpl::put_MousePointer

Call this method to set the type of mouse pointer displayed when the mouse is over the control, for example, arrow, cross, or hourglass.

```
HRESULT STDMETHODCALLTYPE put_MousePointer(
   LONG nMousePointer
);
```

Parameters

nMousePointer

The type of mouse pointer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_MousePointer

Other Resources

CStockPropImpl Members

CStockPropImpl::put_Picture

Call this method to set the picture properties of a graphic (icon, bitmap, or metafile) to be displayed.

```
HRESULT STDMETHODCALLTYPE put_Picture(
   IPictureDisp* pPicture
);
```

Parameters

pPicture

A pointer to the picture's properties. See IPictureDisp for more details.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class
CStockPropImpl::get_Picture
CStockPropImpl::putref_Picture

Other Resources

CStockPropImpl Members

CStockPropImpl::put_ReadyState

Call this method to set the control's ready state, for example, loading or loaded.

```
HRESULT STDMETHODCALLTYPE put_ReadyState(
   LONG nReadyState
);
```

Parameters

nReadyState

The control's ready state.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_ReadyState

Other Resources

CStockPropImpl Members

CStockPropImpl::put_TabStop

Call this method to set the flag that indicates if the control is a tab stop or not.

```
HRESULT STDMETHODCALLTYPE put_TabStop(
   VARIANT_BOOL bTabStop
);
```

Parameters

bTabStop

TRUE if the control is a tab stop.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_TabStop

Other Resources

CStockPropImpl Members

CStockPropImpl::put_Text

Call this method to set the text that is displayed with the control.

```
HRESULT STDMETHODCALLTYPE put_Text(
   BSTR bstrText
);
```

Parameters

bstrText

The text that is displayed with the control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_Text

Other Resources

CStockPropImpl Members

CStockPropImpl::put_Valid

Call this method to set the flag that indicates if the control is valid or not.

```
HRESULT STDMETHODCALLTYPE get_Valid(
   VARIANT_BOOL bValid
);
```

Parameters

bValid

TRUE if the control is valid.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_Valid

Other Resources

CStockPropImpl Members

CStockPropImpl::put_Window

This method calls CStockPropImpl::put_HWND, which returns E_FAIL.

```
HRESULT STDMETHODCALLTYPE put_Window(
   LONG_PTR hWnd
);
```

Parameters

hWnd

The window handle.

Return Value

Returns E_FAIL.

Remarks

The window handle is a read-only value.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::get_Window

Other Resources

CStockPropImpl Members

CStockPropImpl::putref_Font

Call this method to set the control's font properties, with a reference count.

```
HRESULT STDMETHODCALLTYPE putref_Font(
   IFontDisp* pFont
);
```

Parameters

pFont

A pointer to the control's font properties.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The same as CStockPropImpl::put_Font, but with a reference count.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_Font

Other Resources

CStockPropImpl Members

CStockPropImpl::putref_MouseIcon

Call this method to set the picture properties of the graphic (icon, bitmap, or metafile) to be displayed when the mouse is over the control, with a reference count.

```
HRESULT STDMETHODCALLTYPE putref_MouseIcon(
   IPictureDisp* pPicture
);
```

Parameters

pPicture

A pointer to the picture properties of the graphic.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The same as CStockPropImpl::put_MouseIcon, but with a reference count.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_MouseIcon

Other Resources

CStockPropImpl Members

CStockPropImpl::putref_Picture

Call this method to set the picture properties of a graphic (icon, bitmap, or metafile) to be displayed, with a reference count.

```
HRESULT STDMETHODCALLTYPE putref_Picture(
   IPictureDisp* pPicture
);
```

Parameters

pPicture

A pointer to the picture's properties. See IPictureDisp for more details.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The same as CStockPropImpl::put_Picture, but with a reference count.

See Also

Reference

CStockPropImpl Class

CStockPropImpl::put_Picture

Other Resources

CStockPropImpl Members

CTokenGroups Class

This class is a wrapper for the **TOKEN_GROUPS** structure.

```
class CTokenGroups
```

Remarks

An access token is an object that describes the security context of a process or thread and is allocated to each user logged onto a Windows NT or Windows 2000 system.

The **CTokenGroups** class is a wrapper for the TOKEN_GROUPS structure, containing information about the group security identifiers (SIDs) in an access token.

For an introduction to the access control model in Windows, see Access Control in the *Platform SDK*.

Requirements

Header: atlsecurity.h

See Also

Tasks

ATLSecurity Sample: Demonstrates ATL Security Classes

Reference

CSid Class

Other Resources

CTokenGroups Members

ATL Class Overview

Security Global Functions

CTokenGroups Members

Methods

Add	Adds a **CSid** or existing **TOKEN_GROUPS** structure to the **CTokenGroups** object.
CTokenGroups	The constructor.
~CTokenGroups	The destructor.
Delete	Deletes a **CSid** and its associated attributes from the **CTokenGroups** object.
DeleteAll	Deletes all **CSid** objects and their associated attributes from the **CTokenGroups** object.
GetCount	Returns the number of **CSid** objects and associated attributes contained in the **CTokenGroups** object.
GetLength	Returns the size of the **CTokenGroups** object.
GetPTOKEN_GROUPS	Retrieves a pointer to the **TOKEN_GROUPS** structure.
GetSidsAndAttributes	Retrieves the **CSid** objects and attributes belonging to the **CTokenGroups** object.
LookupSid	Retrieves the attributes associated with a **CSid** object.

Operators

operator =	Assignment operator.
operator const TOKEN_GROUPS *	Casts the **CTokenGroups** object to a pointer to the **TOKEN_GROUPS** structure.

See Also

Reference

CTokenGroups Class

CTokenGroups Methods

For information about the methods in **CTokenGroups**, see CTokenGroups Members.

CTokenGroups::Add

Adds a **CSid** or existing **TOKEN_GROUPS** structure to the **CTokenGroups** object.

```
void Add(
   const CSid & rSid,
   DWORD dwAttributes
) throw(...);
void Add(
   const TOKEN_GROUPS & rTokenGroups
) throw(...);
```

Parameters

rSid

A CSid object.

dwAttributes

The attributes to associate with the **CSid** object.

rTokenGroups

A TOKEN_GROUPS structure.

Remarks

These methods add one or more **CSid** objects and their associated attributes to the **CTokenGroups** object.

See Also

Reference

CTokenGroups Class

Other Resources

CTokenGroups Members

CTokenGroups::CTokenGroups

The constructor.

```
CTokenGroups( ) throw( );
CTokenGroups(
   const CTokenGroups & rhs
) throw(...);
CTokenGroups(
   const TOKEN_GROUPS & rhs
) throw(...);
```

Parameters

rhs

The **CTokenGroups** object or TOKEN_GROUPS structure with which to construct the **CTokenGroups** object.

Remarks

The **CTokenGroups** object can optionally be created using a **TOKEN_GROUPS** structure or a previously defined **CTokenGroups** object.

See Also

Reference

CTokenGroups Class
CTokenGroups::~CTokenGroups
TOKEN_GROUPS

Other Resources

CTokenGroups Members

CTokenGroups::~CTokenGroups

The destructor.

```
virtual ~CTokenGroups( ) throw( );
```

Remarks

The destructor frees all allocated resources.

See Also

Reference

CTokenGroups Class

CTokenGroups::CTokenGroups

Other Resources

CTokenGroups Members

CTokenGroups::Delete

Deletes a **CSid** and its associated attributes from the **CTokenGroups** object.

```
bool Delete(
   const CSid & rSid
) throw( );
```

Parameters

rSid

The CSid object for which the security identifier (SID) and attributes should be removed.

Return Value

Returns true if the **CSid** is removed, false otherwise.

See Also

Reference

CTokenGroups Class
CTokenGroups::DeleteAll
CTokenGroups::Add

Other Resources

CTokenGroups Members

CTokenGroups::DeleteAll

Deletes all **CSid** objects and their associated attributes from the **CTokenGroups** object.

```
void DeleteAll( ) throw( );
```

See Also

Reference

CTokenGroups Class
CTokenGroups::Delete
CTokenGroups::Add

Other Resources

CTokenGroups Members

CTokenGroups::GetCount

Returns the number of **CSid** objects contained in **CTokenGroups**.

```
UINT GetCount( ) const throw( );
```

Return Value

Returns the number of CSid objects and their associated attributes contained in the **CTokenGroups** object.

See Also

Reference

CTokenGroups Class

CTokenGroups::GetLength

Other Resources

CTokenGroups Members

CTokenGroups::GetLength

Returns the size of the **CTokenGroup** object.

```
UINT GetLength( ) const throw( );
```

Remarks

Returns the total size of the **CTokenGroup** object, in bytes.

See Also

Reference

CTokenGroups Class

CTokenGroups::GetCount

Other Resources

CTokenGroups Members

CTokenGroups::GetPTOKEN_GROUPS

Retrieves a pointer to the **TOKEN_GROUPS** structure.

```
const TOKEN_GROUPS * GetPTOKEN_GROUPS( ) const throw(...);
```

Return Value

Retrieves a pointer to the TOKEN_GROUPS structure belonging to the **CTokenGroups** access token object.

See Also

Reference

CTokenGroups Class

Other Resources

CTokenGroups Members

CTokenGroups::GetSidsAndAttributes

Retrieves the **CSid** objects and (optionally) the attributes belonging to the **CTokenGroups** object.

```
void GetSidsAndAttributes(
   CSid::CSidArray * pSids,
   CAtlArray< DWORD > * pAttributes = NULL
) const throw(...);
```

Parameters

pSids

Pointer to an array of CSid objects.

pAttributes

Pointer to an array of DWORDs. If this parameter is omitted or NULL, the attributes are not retrieved.

Remarks

This method will enumerate all of the **CSid** objects contained in the **CTokenGroups** object and place them and (optionally) the attribute flags into array objects.

See Also

Reference

CTokenGroups Class

CTokenGroups::LookupSid

Other Resources

CTokenGroups Members

CTokenGroups::LookupSid

Retrieves the attributes associated with a **CSid** object.

```
bool LookupSid(
   const CSid & rSid,
   DWORD * pdwAttributes = NULL
) const throw( );
```

Parameters

rSid

The CSid object.

pdwAttributes

Pointer to a DWORD which will accept the **CSid** object's attribute. If omitted or NULL, the attribute will not be retrieved.

Return Value

Returns true if the **CSid** is found, false otherwise.

Remarks

Setting *pdwAttributes* to NULL provides a way of confirming the existence of the **CSid** without accessing the attribute. Note that this method should not be used to check access rights as incorrect results may occur under Windows 2000. Applications should instead use the CAccessToken::CheckTokenMembership method.

See Also

Reference

CTokenGroups Class

CTokenGroups::GetSidsAndAttributes

Other Resources

CTokenGroups Members

CTokenGroups Operators

For information about the operators in **CTokenGroups**, see CTokenGroups Members.

CTokenGroups::operator =

Assignment operator.

```
CTokenGroups & operator =(
   const TOKEN_GROUPS & rhs
) throw(...);
CTokenGroups & operator =(
   const CTokenGroups & rhs
) throw(...);
```

Parameters

rhs

The **CTokenGroups** object or TOKEN_GROUPS structure to assign to the **CTokenGroups** object.

Return Value

Returns the updated **CTokenGroups** object.

See Also

Reference

CTokenGroups Class

Other Resources

CTokenGroups Members

CTokenGroups::operator const TOKEN_GROUPS *

Casts a value to a pointer to the **TOKEN_GROUPS** structure.

```
operator const TOKEN_GROUPS *( ) const throw(...);
```

Remarks

Casts a value to a pointer to the TOKEN_GROUPS structure.

See Also

Reference

CTokenGroups Class

Other Resources

CTokenGroups Members

CTokenPrivileges Class

This class is a wrapper for the **TOKEN_PRIVILEGES** structure.

```
class CTokenPrivileges
```

Remarks

An access token is an object that describes the security context of a process or thread and is allocated to each user logged onto a Windows NT or Windows 2000 system.

The access token is used to describe the various security privileges granted to each user. A privilege consists of a 64-bit number called a locally unique identifier (LUID) and a descriptor string.

The **CTokenPrivileges** class is a wrapper for the TOKEN_PRIVILEGES structure and contains 0 or more privileges. Privileges can be added, deleted, or queried using the supplied class methods.

For an introduction to the access control model in Windows, see Access Control in the *Platform SDK*.

Requirements

Header: atlsecurity.h

See Also

Tasks

ATLSecurity Sample: Demonstrates ATL Security Classes

Reference

TOKEN_PRIVILEGES

LUID

LUID_AND_ATTRIBUTES

Other Resources

CTokenPrivileges Members

ATL Class Overview

Security Global Functions

CTokenPrivileges Members

Methods

Add	Adds one or more privileges to the **CTokenPrivileges** object.
CTokenPrivileges	The constructor.
~CTokenPrivileges	The destructor.
Delete	Deletes a privilege from the **CTokenPrivileges** object.
DeleteAll	Deletes all privileges from the **CTokenPrivileges** object.
GetCount	Returns the number of privilege entries in the **CTokenPrivileges** object.
GetDisplayNames	Retrieves display names for the privileges contained in the **CTokenPrivileges** object.
GetLength	Returns the buffer size in bytes required to hold the **TOKEN_PRIVILEGES** structure represented by the **CTokenPrivileges** object.
GetLuidsAndAttributes	Retrieves the locally unique identifiers (LUIDs) and attribute flags from the **CTokenPrivileges** object.
GetNamesAndAttributes	Retrieves the privilege names and attribute flags from the **CTokenPrivileges** object.
GetPTOKEN_PRIVILEGES	Returns a pointer to the **TOKEN_PRIVILEGES** structure.
LookupPrivilege	Retrieves the attribute associated with a given privilege name.

Operators

operator =	Assignment operator.
operator const TOKEN_PRIVILEGES *	Casts a value to a pointer to the **TOKEN_PRIVILEGES** structure.

See Also

Reference

CTokenPrivileges Class

CTokenPrivileges Methods

For information about the methods in **CTokenPrivileges**, see CTokenPrivileges Members.

CTokenPrivileges::Add

Adds one or more privileges to the **CTokenPrivileges** access token object.

```
bool Add(
   LPCTSTR pszPrivilege,
   bool bEnable
) throw(...);
void Add(
   const TOKEN_PRIVILEGES & rPrivileges
) throw(...);
```

Parameters

pszPrivilege

Pointer to a null-terminated string that specifies the name of the privilege, as defined in the WINNT.H header file.

bEnable

If true, the privilege is enabled. If false, the privilege is disabled.

rPrivileges

Reference to a TOKEN_PRIVILEGES structure. The privileges and attributes are copied from this structure and added to the **CTokenPrivileges** object.

Return Value

The first form of this method returns true if the privileges are successfully added, false otherwise.

See Also

Reference

CTokenPrivileges Class
CTokenPrivileges::Delete
CTokenPrivileges::DeleteAll

Other Resources

CTokenPrivileges Members

CTokenPrivileges::CTokenPrivileges

The constructor.

```
CTokenPrivileges( ) throw( );
CTokenPrivileges(
   const CTokenPrivileges & rhs
) throw(...);
CTokenPrivileges(
   const TOKEN_PRIVILEGES & rPrivileges
) throw(...);
```

Parameters

rhs

The **CTokenPrivileges** object to assign to the new object.

rPrivileges

The TOKEN_PRIVILEGES structure to assign to the new **CTokenPrivileges** object.

Remarks

The **CTokenPrivileges** object can optionally be created using a **TOKEN_PRIVILEGES** structure or a previously defined **CTokenPrivileges** object.

See Also

Reference

CTokenPrivileges Class

CTokenPrivileges::~CTokenPrivileges

Other Resources

CTokenPrivileges Members

CTokenPrivileges::~CTokenPrivileges

The destructor.

```
virtual ~CTokenPrivileges( ) throw( );
```

Remarks

The destructor frees all allocated resources.

See Also

Reference

CTokenPrivileges Class

CTokenPrivileges::CTokenPrivileges

Other Resources

CTokenPrivileges Members

CTokenPrivileges::Delete

Deletes a privilege from the **CTokenPrivileges** access token object.

```
bool Delete(
   LPCTSTR pszPrivilege
) throw( );
```

Parameters

pszPrivilege

Pointer to a null-terminated string that specifies the name of the privilege, as defined in the WINNT.H header file. For example, this parameter could specify the constant SE_SECURITY_NAME, or its corresponding string, "SeSecurityPrivilege."

Return Value

Returns true if the privilege was successfully deleted, false otherwise.

Remarks

This method is useful as a tool for creating restricted tokens under Windows 2000.

See Also

Reference

CTokenPrivileges Class
CTokenPrivileges::DeleteAll
CTokenPrivileges::Add
CAccessToken::CreateRestrictedToken

Other Resources

CTokenPrivileges Members

CTokenPrivileges::DeleteAll

Deletes all privileges from the **CTokenPrivileges** access token object.

```
void DeleteAll( ) throw( );
```

Remarks

Deletes all privileges contained in the **CTokenPrivileges** access token object.

See Also

Reference

CTokenPrivileges Class
CTokenPrivileges::Delete
CTokenPrivileges::Add

Other Resources

CTokenPrivileges Members

CTokenPrivileges::GetDisplayNames

Retrieves display names for the privileges contained in the **CTokenPrivileges** access token object.

```
void GetDisplayNames(
   CNames* pDisplayNames
) const throw(...);
```

Parameters

pDisplayNames

A pointer to an array of **CString** objects. **CNames** is defined as a typedef: **CTokenPrivileges::CAtlArray<CString>**.

Remarks

The parameter *pDisplayNames* is a pointer to an array of **CString** objects which will receive the display names corresponding to the privileges contained in the **CTokenPrivileges** object. This method retrieves display names only for the privileges specified in the Defined Privileges section of WINNT.H.

This method retrieves a displayable name: for example, if the attribute name is SE_REMOTE_SHUTDOWN_NAME, the displayable name is "Force shutdown from a remote system." To obtain the system name, use CTokenPrivileges::GetNamesAndAttributes.

See Also

Reference

CTokenPrivileges Class
CTokenPrivileges::GetNamesAndAttributes
CTokenPrivileges::LookupPrivilege
CTokenPrivileges::GetLuidsAndAttributes

Other Resources

CTokenPrivileges Members

CTokenPrivileges::GetCount

Returns the number of privilege entries in the **CTokenPrivileges** object.

```
UINT GetCount( ) const throw( );
```

Return Value

Returns the number of privileges contained in the **CTokenPrivileges** object.

See Also

Reference

CTokenPrivileges Class

CTokenPrivileges::GetLength

Other Resources

CTokenPrivileges Members

CTokenPrivileges::GetLength

Returns the length of the **CTokenPrivileges** object.

```
UINT GetLength( ) const throw( );
```

Return Value

Returns the number of bytes required to hold a **TOKEN_PRIVILEGES** structure represented by the **CTokenPrivileges** object, including all of the privilege entries it contains.

See Also

Reference

CTokenPrivileges Class

CTokenPrivileges::GetCount

Other Resources

CTokenPrivileges Members

CTokenPrivileges::GetLuidsAndAttributes

Retrieves the locally unique identifiers (LUIDs) and attribute flags from the **CTokenPrivileges** object.

```
void GetLuidsAndAttributes(
   CLUIDArray * pPrivileges,
   CAttributes * pAttributes = NULL
) const throw(...);
```

Parameters

pPrivileges

Pointer to an array of LUID objects. **CLUIDArray** is a typedef defined as **CAtlArray<LUID> CLUIDArray**.

pAttributes

Pointer to an array of DWORD objects. If this parameter is omitted or NULL, the attributes are not retrieved. **CAttributes** is a typedef defined as **CAtlArray <DWORD> CAttributes**.

Remarks

This method will enumerate all of the privileges contained in the **CTokenPrivileges** access token object and place the individual LUIDs and (optionally) the attribute flags into array objects.

See Also

Reference

CTokenPrivileges Class
CTokenPrivileges::GetNamesAndAttributes
CTokenPrivileges::LookupPrivilege
CTokenPrivileges::GetDisplayNames
LUID_AND_ATTRIBUTES

Other Resources

CTokenPrivileges Members

CTokenPrivileges::GetNamesAndAttributes

Retrieves the name and attribute flags from the **CTokenPrivileges** object.

```
void GetNamesAndAttributes(
   CNames * pNames,
   CAttributes * pAttributes = NULL
) const throw(...);
```

Parameters

pNames

Pointer to an array of **CString** objects. **CNames** is a typedef defined as **CAtlArray <CString> CNames**.

pAttributes

Pointer to an array of DWORD objects. If this parameter is omitted or NULL, the attributes are not retrieved. **CAttributes** is a typedef defined as **CAtlArray <DWORD> CAttributes**.

Remarks

This method will enumerate all of the privileges contained in the **CTokenPrivileges** object, placing the name and (optionally) the attribute flags into array objects.

This method retrieves the attribute name, rather than the displayable name: for example, if the attribute name is SE_REMOTE_SHUTDOWN_NAME, the system name is "SeRemoteShutdownPrivilege." To obtain the displayable name, use the method CTokenPrivileges::GetDisplayNames.

See Also

Reference

CTokenPrivileges Class
CTokenPrivileges::GetDisplayNames
CTokenPrivileges::LookupPrivilege
CTokenPrivileges::GetLuidsAndAttributes

Other Resources

CTokenPrivileges Members

CTokenPrivileges::GetPTOKEN_PRIVILEGES

Returns a pointer to the **TOKEN_PRIVILEGES** structure.

```
const TOKEN_PRIVILEGES * GetPTOKEN_PRIVILEGES( ) const throw(...);
```

Return Value

Returns a pointer to the TOKEN_PRIVILEGES structure.

See Also

Reference

CTokenPrivileges Class

TOKEN_PRIVILEGES

Other Resources

CTokenPrivileges Members

CTokenPrivileges::LookupPrivilege

Retrieves the attribute associated with a given privilege name.

```
bool LookupPrivilege(
   LPCTSTR pszPrivilege,
   DWORD * pdwAttributes = NULL
) const throw(...);
```

Parameters

pszPrivilege

Pointer to a null-terminated string that specifies the name of the privilege, as defined in the WINNT.H header file. For example, this parameter could specify the constant SE_SECURITY_NAME, or its corresponding string, "SeSecurityPrivilege."

pdwAttributes

Pointer to a variable that receives the attributes.

Return Value

Returns true if the attribute is successfully retrieved, false otherwise.

See Also

Reference

CTokenPrivileges Class
CTokenPrivileges::GetNamesAndAttributes
CTokenPrivileges::GetDisplayNames
CTokenPrivileges::GetLuidsAndAttributes

Other Resources

CTokenPrivileges Members

CTokenPrivileges Operators

For information about the operators in **CTokenPrivileges**, see CTokenPrivileges Members.

CTokenPrivileges::operator =

Assignment operator.

```
CTokenPrivileges & operator =(
   const TOKEN_PRIVILEGES & rPrivileges
) throw(...);
CTokenPrivileges & operator =(
   const CTokenPrivileges & rhs
) throw(...);
```

Parameters

rPrivileges

The TOKEN_PRIVILEGES structure to assign to the **CTokenPrivileges** object.

rhs

The **CTokenPrivileges** object to assign to the object.

Return Value

Returns the updated **CTokenPrivileges** object.

See Also

Reference

CTokenPrivileges Class

Other Resources

CTokenPrivileges Members

CTokenPrivileges::operator const TOKEN_PRIVILEGES *

Casts a value to a pointer to the **TOKEN_PRIVILEGES** structure.

```
operator const TOKEN_PRIVILEGES *( ) const throw(...);
```

Remarks

Casts a value to a pointer to the TOKEN_PRIVILEGES structure.

See Also

Reference

CTokenPrivileges Class

Other Resources

CTokenPrivileges Members

CW2AEX Class

This class is used by the string conversion macros **CT2AEX**, **CW2TEX**, **CW2CTEX**, and **CT2CAEX**, and the typedef **CW2A**.

```
template<
   int t_nBufferLength = 128
>
class CW2AEX
```

Parameters

t_nBufferLength

The size of the buffer used in the translation process. The default length is 128 bytes.

Remarks

Unless extra functionality is required, use **CT2AEX**, **CW2TEX**, **CW2CTEX**, **CT2CAEX**, or **CW2A** in your code.

This class contains a fixed-size static buffer which is used to store the result of the conversion. If the result is too large to fit into the static buffer, the class allocates memory using **malloc**, freeing the memory when the object goes out of scope. This ensures that, unlike text conversion macros available in previous versions of ATL, this class is safe to use in loops and that it won't overflow the stack.

If the class tries to allocate memory on the heap and fails, it will call **AtlThrow** with an argument of **E_OUTOFMEMORY**.

By default, the ATL conversion classes and macros use the current thread's ANSI code page for the conversion. If you want to override that behavior for a specific conversion, specify the code page as the second parameter to the constructor for the class.

The following macros are based on this class:

- **CT2AEX**
- **CW2TEX**
- **CW2CTEX**
- **CT2CAEX**

The following typedef is based on this class:

- **CW2A**

For a discussion of these text conversion macros, see ATL and MFC String Conversion Macros.

Example

See ATL and MFC String Conversion Macros for an example of using these string conversion macros.

Requirements

Header: atlconv.h

See Also

Reference

CA2AEX Class
CA2CAEX Class
CA2WEX Class
CW2CWEX Class
CW2WEX Class

Other Resources

CW2AEX Members
ATL Class Overview

CW2AEX Members

Methods

CW2AEX	The constructor.
~CW2AEX	The destructor.

Operators

operator LPSTR	Conversion operator.

Data Members

m_psz	The data member that stores the source string.
m_szBuffer	The static buffer, used to store the converted string.

See Also

Reference

CW2AEX Class

CW2AEX Methods

For information about the methods in **CW2AEX**, see CW2AEX Members.

CW2AEX::CW2AEX

The constructor.

```
CW2AEX(
   LPCWSTR psz,
   UINT nCodePage
) throw(...);
CW2AEX(
   LPCWSTR psz
) throw(...);
```

Parameters

psz

The text string to be converted.

nCodePage

The code page used to perform the conversion. See the code page parameter discussion for the Platform SDK function MultiByteToWideChar for more details.

Remarks

Allocates the buffer used in the translation process.

See Also

Reference

CW2AEX Class

Other Resources

CW2AEX Members

CW2AEX::~CW2AEX

The destructor.

```
~CW2AEX( ) throw( );
```

Remarks

Frees the allocated buffer.

See Also

Reference

CW2AEX Class

Other Resources

CW2AEX Members

CW2AEX Operators

For information about the operators in **CW2AEX**, see CW2AEX Members.

CW2AEX::operator LPSTR

Conversion operator.

```
operator LPSTR( ) const throw( );
```

Return Value

Returns the text string as type **LPSTR.**

See Also

Reference

CW2AEX Class

Other Resources

CW2AEX Members

CW2AEX Data Members

For information about the data members in **CW2AEX**, see CW2AEX Members.

CW2AEX::m_psz

The data member that stores the source string.

```
LPSTR m_psz;
```

See Also

Reference

CW2AEX Class

Other Resources

CW2AEX Members

CW2AEX::m_szBuffer

The static buffer, used to store the converted string.

```
char m_szBuffer[t_nBufferLength];
```

See Also

Reference

CW2AEX Class

Other Resources

CW2AEX Members

CW2CWEX Class

This class is used by the string conversion macros **CW2CTEX** and **CT2CWEX**, and the typedef **CW2W**.

```
template<
   int t_nBufferLength = 128
>
class CW2CWEX
```

Parameters

t_nBufferLength

The size of the buffer used in the translation process. The default length is 128 bytes.

Remarks

Unless extra functionality is required, use **CW2CTEX**, **CT2CWEX**, or **CW2W** in your code.

This class is safe to use in loops and won't overflow the stack. By default, the ATL conversion classes and macros use the current thread's ANSI code page for the conversion.

The following macros are based on this class:

- **CW2CTEX**
- **CT2CWEX**

The following typedef is based on this class:

- **CW2W**

For a discussion of these text conversion macros, see ATL and MFC String Conversion Macros.

Example

See ATL and MFC String Conversion Macros for an example of using these string conversion macros.

Requirements

Header: atlconv.h

See Also

Reference

CA2AEX Class
CA2CAEX Class
CA2WEX Class
CW2AEX Class
CW2WEX Class

Other Resources

CW2CWEX Members
ATL Class Overview

CW2CWEX Members

Methods

CW2CWEX	The constructor.
~CW2CWEX	The destructor.

Operators

operator LPCWSTR	Conversion operator.

Data Members

m_psz	The data member that stores the source string.

See Also

Reference

CW2CWEX Class

CW2CWEX Methods

For information about the methods in **CW2CWEX**, see CW2CWEX Members.

CW2CWEX::CW2CWEX

The constructor.

```
CW2CWEX(
   LPCWSTR psz,
   UINT nCodePage
) throw(...);
CW2CWEX(
   LPCWSTR psz
) throw(...);
```

Parameters

psz

The text string to be converted.

nCodePage

The code page. Not used in this class.

Remarks

Allocates the buffer used in the translation process.

See Also

Reference

CW2CWEX Class

Other Resources

CW2CWEX Members

CW2CWEX::~CW2CWEX

The destructor.

```
~CW2CWEX( ) throw( );
```

Remarks

Frees the allocated buffer.

See Also

Reference

CW2CWEX Class

Other Resources

CW2CWEX Members

CW2CWEX Operators

For information about the operators in **CW2CWEX**, see CW2CWEX Members.

CW2CWEX::operator LPCWSTR

Conversion operator.

```
operator LPCWSTR( ) const throw( );
```

Return Value

Returns the text string as type **LPCWSTR.**

See Also

Reference

CW2CWEX Class

Other Resources

CW2CWEX Members

CW2CWEX Data Members

For information about the data members in **CW2CWEX**, see CW2CWEX Members.

CW2CWEX::m_psz

The data member that stores the source string.

```
LPCWSTR m_psz;
```

See Also

Reference

CW2CWEX Class

Other Resources

CW2CWEX Members

CW2WEX Class

This class is used by the string conversion macros **CW2TEX** and **CT2WEX**, and the typedef **CW2W**.

```
template<
   int t_nBufferLength = 128
>
class CW2WEX
```

Parameters

t_nBufferLength

The size of the buffer used in the translation process. The default length is 128 bytes.

Remarks

Unless extra functionality is required, use **CW2TEX**, **CT2WEX**, or **CW2W** in your code.

This class contains a fixed-size static buffer which is used to store the result of the conversion. If the result is too large to fit into the static buffer, the class allocates memory using **malloc**, freeing the memory when the object goes out of scope. This ensures that, unlike text conversion macros available in previous versions of ATL, this class is safe to use in loops and that it won't overflow the stack.

If the class tries to allocate memory on the heap and fails, it will call **AtlThrow** with an argument of **E_OUTOFMEMORY**.

By default, the ATL conversion classes and macros use the current thread's ANSI code page for the conversion.

The following macros are based on this class:

- **CW2TEX**
- **CT2WEX**

The following typedef is based on this class:

- **CW2W**

For a discussion of these text conversion macros, see ATL and MFC String Conversion Macros.

Example

See ATL and MFC String Conversion Macros for an example of using these string conversion macros.

Requirements

Header: atlconv.h

See Also

Reference

CA2AEX Class
CA2CAEX Class
CA2WEX Class
CW2AEX Class
CW2CWEX Class

Other Resources

CW2WEX Members
ATL Class Overview

CW2WEX Members

Methods

CW2WEX	The constructor.
~CW2WEX	The destructor.

Operators

operator LPWSTR	Conversion operator.

Data Members

m_psz	The data member that stores the source string.
m_szBuffer	The static buffer, used to store the converted string.

See Also

Reference

CW2WEX Class

CW2WEX Methods

For information about the methods in **CW2WEX**, see CW2WEX Members.

CW2WEX::CW2WEX

The constructor.

```
CW2WEX(
   LPCWSTR psz,
   UINT nCodePage
) throw(...);
CW2WEX(
   LPCWSTR psz
 ) throw(...);
```

Parameters

psz

The text string to be converted.

nCodePage

The code page. Not used in this class.

Remarks

Creates the buffer required for the translation.

See Also

Reference

CW2WEX Class

Other Resources

CW2WEX Members

CW2WEX::~CW2WEX

The destructor..

```
~CW2WEX( ) throw( );
```

Remarks

Frees the allocated buffer.

See Also

Reference

CW2WEX Class

Other Resources

CW2WEX Members

CW2WEX Operators

For information about the operators in **CW2WEX**, see CW2WEX Members.

CW2WEX::operator LPWSTR

Cast operator.

```
operator LPWSTR( ) const throw( );
```

Return Value

Returns the text string as type **LPWSTR**.

See Also

Reference

CW2WEX Class

Other Resources

CW2WEX Members

CW2WEX Data Members

For information about the data members in **CW2WEX**, see CW2WEX Members.

CW2WEX::m_psz

The data member that stores the source string.

```
LPWSTR m_psz;
```

See Also

Reference

CW2WEX Class

Other Resources

CW2WEX Members

CW2WEX::m_szBuffer

The static buffer, used to store the converted string.

```
wchar_t m_szBuffer[t_nBufferLength];
```

See Also

Reference

CW2WEX Class

Other Resources

CW2WEX Members

CWin32Heap Class

This class implements IAtlMemMgr using the Win32 heap allocation functions.

```
class CWin32Heap : public IAtlMemMgr
```

Remarks

CWin32Heap implements memory allocation methods using the Win32 heap allocation functions, including HeapAlloc and HeapFree. Unlike other Heap classes, **CWin32Heap** requires a valid heap handle to be provided before memory is allocated: the other classes default to using the process heap. The handle can be supplied to the constructor or to the CWin32Heap::Attach method. See the CWin32Heap::CWin32Heap method for more details.

Example

See the example for IAtlMemMgr.

Requirements

Header: atlmem.h

See Also

Reference

IAtlMemMgr Class
CLocalHeap Class
CGlobalHeap Class
CCRTHeap Class
CComHeap Class

Other Resources

CWin32Heap Members
ATL Class Overview

CWin32Heap Members

Methods

Allocate	Allocates a block of memory from the heap object.
Attach	Attaches the heap object to an existing heap.
CWin32Heap	The constructor.
~CWin32Heap	The destructor.
Detach	Detaches the heap object from an existing heap.
Free	Frees memory previously allocated from the heap.
GetSize	Returns the size of a memory block allocated from the heap object.
Reallocate	Reallocates a block of memory from the heap object.

Data Members

m_bOwnHeap	A flag used to determine current ownership of the heap handle.
m_hHeap	Handle to the heap object.

See Also

Reference

CWin32Heap Class

Other Resources

IAtlMemMgr Members

CWin32Heap Methods

For information about the methods in **CWin32Heap**, see CWin32Heap Members.

CWin32Heap::Allocate

Allocates a block of memory from the heap object.

```
virtual void* Allocate(
   size_t nBytes
) throw( );
```

Parameters

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the newly allocated memory block.

Remarks

Call CWin32Heap::Free or CWin32Heap::Reallocate to free the memory allocated by this method.

Implemented using HeapAlloc.

See Also

Reference

CWin32Heap Class

Other Resources

CWin32Heap Members

CWin32Heap::Attach

Attaches the heap object to an existing heap.

```
void Attach(
   HANDLE hHeap,
   bool bTakeOwnership
) throw( );
```

Parameters

hHeap

An existing heap handle.

bTakeOwnership

A flag indicating if the **CWin32Heap** object is to take ownership over the resources of the heap.

Remarks

If *bTakeOwnership* is TRUE, the **CWin32Heap** object is responsible for deleting the heap handle.

See Also

Reference

CWin32Heap Class

CWin32Heap::Detach

Other Resources

CWin32Heap Members

CWin32Heap::CWin32Heap

The constructor.

```
CWin32Heap( ) throw( );
CWin32Heap(
   HANDLE hHeap
) throw( );
CWin32Heap(
   DWORD dwFlags,
   size_t nInitialSize,
   size_t nMaxSize = 0
);
```

Parameters

hHeap

An existing heap object.

dwFlags

Flags used in creating the heap.

nInitialSize

The initial size of the heap.

nMaxSize

The maximum size of the heap.

Remarks

Before allocating memory, it is necessary to provide the **CWin32Heap** object with a valid heap handle. The simplest way to achieve this is to use the process heap:

```
CWin32Heap MyHeap(GetProcessHeap());
```

It is also possible to supply an existing heap handle to the constructor, in which case the new object does not take over ownership of the heap. The original heap handle will still be valid when the **CWin32Heap** object is deleted.

An existing heap can also be attached to the new object, using CWin32Heap::Attach.

If a heap is required where operations are all performed from a single thread, the best way is to create the object as follows:

```
CWin32Heap MyHeap(HEAP_NO_SERIALIZE, SomeInitialSize);
```

The parameter **HEAP_NO_SERIALIZE** specifies that mutual exclusion will not be used when the heap functions allocate and free memory, with an according increase in performance.

The third parameter defaults to 0, which allows the heap to grow as required. See HeapCreate for an explanation of the memory sizes and flags.

See Also

Reference

CWin32Heap Class

HeapCreate

Other Resources

CWin32Heap Members

CWin32Heap::~CWin32Heap

The destructor.

```
~CWin32Heap( ) throw( );
```

Remarks

Destroys the heap handle if the **CWin32Heap** object has ownership of the heap.

See Also

Reference

CWin32Heap Class

Other Resources

CWin32Heap Members

CWin32Heap::Detach

Detaches the heap object from an existing heap.

```
HANDLE Detach( ) throw( );
```

Return Value

Returns the handle to the heap to which the object was previously attached.

See Also

Reference

CWin32Heap Class

CWin32Heap::Attach

Other Resources

CWin32Heap Members

CWin32Heap::Free

Frees memory previously allocated from the heap by CWin32Heap::Allocate or CWin32Heap::Reallocate.

```
virtual void Free(
   void* p
) throw( );
```

Parameters

p

Pointer to the block of memory to free. NULL is a valid value and does nothing.

See Also

Reference

CWin32Heap Class
CWin32Heap::Allocate
CWin32Heap::Reallocate
HeapFree

Other Resources

CWin32Heap Members

CWin32Heap::GetSize

Returns the size of a memory block allocated from the heap object.

```
virtual size_t GetSize(
   void* p
) throw( );
```

Parameters

p

Pointer to the memory block whose size the method will obtain. This is a pointer returned by CWin32Heap::Allocate or CWin32Heap::Reallocate.

Return Value

Returns the size, in bytes, of the allocated memory block.

See Also

Reference

CWin32Heap Class

HeapSize

Other Resources

CWin32Heap Members

CWin32Heap::Reallocate

Reallocates a block of memory from the heap object.

```
virtual void* Reallocate(
   void* p,
   size_t nBytes
) throw( );
```

Parameters

p

Pointer to the block of memory to reallocate.

nBytes

The new size in bytes of the allocated block. The block can be made larger or smaller.

Return Value

Returns a pointer to the newly allocated memory block.

Remarks

If *p* is NULL, it's assumed that the memory block has not yet been allocated and CWin32Heap::Allocate is called, with an argument of *nBytes*.

See Also

Reference

CWin32Heap Class
HeapReAlloc
CWin32Heap::Allocate
CWin32Heap::Free

Other Resources

CWin32Heap Members

CWin32Heap Data Members

For information about the data members in **CWin32Heap**, see CWin32Heap Members.

CWin32Heap::m_bOwnHeap

A flag used to determine current ownership of the heap handle stored in m_hHeap.

```
bool m_bOwnHeap;
```

See Also

Reference

CWin32Heap Class

Other Resources

CWin32Heap Members

CWin32Heap::m_hHeap

Handle to the heap object.

```
HANDLE m_hHeap;
```

Remarks

A variable used to store a handle to the heap object.

See Also

Reference

CWin32Heap Class

Other Resources

CWin32Heap Members

CWindow Class

This class provides methods for manipulating a window.

```
class CWindow
```

Remarks

CWindow provides the base functionality for manipulating a window in ATL. Many of the **CWindow** methods simply wrap one of the Win32 API functions. For example, compare the prototypes for **CWindow::ShowWindow** and **ShowWindow**:

CWindow method	Win32 function
BOOL ShowWindow(int *nCmdShow* **);**	**BOOL ShowWindow(HWND** *hWnd* **, int** *nCmdShow* **);**

CWindow::ShowWindow calls the Win32 function **ShowWindow** by passing **CWindow::m_hWnd** as the first parameter. Every **CWindow** method that directly wraps a Win32 function passes the **m_hWnd** member; therefore, much of the **CWindow** documentation will refer you to the *Platform SDK*.

Note
Not every window-related Win32 function is wrapped by **CWindow**, and not every **CWindow** method wraps a Win32 function.

CWindow::m_hWnd stores the **HWND** that identifies a window. An **HWND** is attached to your object when you:

- Specify an **HWND** in **CWindow**'s constructor.
- Call **CWindow::Attach**.
- Use **CWindow**'s **operator =**.
- Create or subclass a window using one of the following classes derived from **CWindow**:

  CWindowImpl Allows you to create a new window or subclass an existing window.

  CContainedWindow Implements a window contained within another object. You can create a new window or subclass an existing window.

  CDialogImpl Allows you to create a modal or modeless dialog box.

For more information about windows, see Windows and subsequent topics in the *Platform SDK*. For more information about using windows in ATL, see the article ATL Window Classes.

Requirements

Header: atlwin.h

See Also

Other Resources

CWindow Members

ATL Class Overview

CWindow Members

The following categories of **CWindow** members are available:

Alert Methods	Data Members
Attribute Methods	Icon Methods
Caret Methods	Menu Methods
Clipboard Methods	Message Methods
Construction, Destruction, and Initialization	Process and Thread Methods
Coordinate Mapping Methods	Timer Methods
Dialog Box Item Methods	Update and Painting Methods
Drag-Drop Methods	Window Access Methods
Font Methods	Window Size and Position Methods
Help Methods	Window State Methods
Window Tree Access Methods	Scrolling Methods
Hot Key Methods	Window Text Methods
Operators	

Alert Methods

FlashWindow	Flashes the window once.
MessageBox	Displays a message box.

Attribute Methods

GetExStyle	Retrieves the extended window styles.
GetStyle	Retrieves the window styles.
GetWindowLong	Retrieves a 32-bit value at a specified offset into the extra window memory.
GetWindowLongPtr	Retrieves information about the specified window, including a value at a specified offset into the extra window memory.
GetWindowWord	Retrieves a 16-bit value at a specified offset into the extra window memory.
ModifyStyle	Modifies the window styles.
ModifyStyleEx	Modifies the extended window styles.
SetWindowLong	Sets a 32-bit value at a specified offset into the extra window memory.

SetWindowLongPtr	Changes an attribute of the specified window, and also sets a value at the specified offset in the extra window memory.
SetWindowWord	Sets a 16-bit value at a specified offset into the extra window memory.

Caret Methods

CreateCaret	Creates a new shape for the system caret.
CreateGrayCaret	Creates a gray rectangle for the system caret.
CreateSolidCaret	Creates a solid rectangle for the system caret.
HideCaret	Hides the system caret.
ShowCaret	Displays the system caret.

Clipboard Methods

ChangeClipboardChain	Removes the window from the chain of Clipboard viewers.
OpenClipboard	Opens the Clipboard.
SetClipboardViewer	Adds the window to the Clipboard viewer chain.

Construction, Destruction, and Initialization

Attach	Attaches a window to the **CWindow** object.
Create	Creates a window.
CWindow	Constructor.
DestroyWindow	Destroys the window associated with the **CWindow** object.
Detach	Detaches the window from the **CWindow** object.

Coordinate Mapping Methods

ClientToScreen	Converts client coordinates to screen coordinates.
MapWindowPoints	Converts a set of points from the window's coordinate space to the coordinate space of another window.
ScreenToClient	Converts screen coordinates to client coordinates.

Dialog Box Item Methods

CheckDlgButton	Changes the check state of the specified button.
CheckRadioButton	Checks the specified radio button.
DlgDirList	Fills a list box with the names of all files matching a specified path or file name.
DlgDirListComboBox	Fills a combo box with the names of all files matching a specified path or file name.
DlgDirSelect	Retrieves the current selection from a list box.
DlgDirSelectComboBox	Retrieves the current selection from a combo box.

GetDlgControl	Retrieves an interface on the specified control.
GetDlgHost	Retrieves a pointer to an interface to the ATL Control hosting container.
GetDlgItemInt	Translates a control's text to an integer.
GetDlgItemText	Retrieves a control's text.
GetNextDlgGroupItem	Retrieves the previous or next control within a group of controls.
GetNextDlgTabItem	Retrieves the previous or next control having the **WS_TABSTOP** style.
GotoDlgCtrl	Sets the keyboard focus to a control in the dialog box.
IsDialogMessage	Determines whether a message is intended for the specified dialog box.
IsDlgButtonChecked	Determines the check state of the button.
NextDlgCtrl	Sets the keyboard focus to the next control in the dialog box.
PrevDlgCtrl	Sets the keyboard focus to the previous control in the dialog box.
SendDlgItemMessage	Sends a message to a control.
SetDlgItemInt	Changes a control's text to the string representation of an integer value.
SetDlgItemText	Changes a control's text.

Drag-Drop Methods

DragAcceptFiles	Registers whether the window accepts dragged files.

Font Methods

GetFont	Retrieves the window's current font.
SetFont	Changes the window's current font.

Help Methods

GetWindowContextHelpId	Retrieves the window's help context identifier.
SetWindowContextHelpId	Sets the window's help context identifier.
WinHelp	Starts Windows Help.

Hot Key Methods

GetHotKey	Determines the hot key associated with the window.
SetHotKey	Associates a hot key with the window.

Icon Methods

GetIcon	Retrieves the window's large or small icon.
SetIcon	Changes the window's large or small icon.

Menu Methods

DrawMenuBar	Redraws the window's menu bar.
GetMenu	Retrieves the window's menu.
GetSystemMenu	Creates a copy of the system menu for modification.
HiliteMenuItem	Highlights or removes the highlight from a top-level menu item.
SetMenu	Changes the window's current menu.

Message Methods

PostMessage	Places a message in the message queue associated with the thread that created the window. Returns without waiting for the thread to process the message.
SendMessage	Sends a message to the window and does not return until the window procedure has processed the message.
SendNotifyMessage	Sends a message to the window. If the window was created by the calling thread, **SendNotifyMessage** does not return until the window procedure has processed the message. Otherwise, it returns immediately.

Process and Thread Methods

GetWindowProcessID	Retrieves the identifier of the process that created the window.
GetWindowThreadID	Retrieves the identifier of the thread that created the specified window.

Scrolling Methods

EnableScrollBar	Enables or disables the scroll bar arrows.
GetScrollInfo	Retrieves the parameters of a scroll bar.
GetScrollPos	Retrieves the position of the scroll box.
GetScrollRange	Retrieves the scroll bar range.
ScrollWindow	Scrolls the specified client area.
ScrollWindowEx	Scrolls the specified client area with additional features.
SetScrollInfo	Sets the parameters of a scroll bar.
SetScrollPos	Changes the position of the scroll box.
SetScrollRange	Changes the scroll bar range.
ShowScrollBar	Shows or hides a scroll bar.

Timer Methods

KillTimer	Destroys a timer event.
SetTimer	Creates a timer event.

Update and Painting Methods

BeginPaint	Prepares the window for painting.

EndPaint	Marks the end of painting.
GetDC	Retrieves a device context for the client area.
GetDCEx	Retrieves a device context for the client area and allows clipping options.
GetUpdateRect	Retrieves the coordinates of the smallest rectangle that completely encloses the update region.
GetUpdateRgn	Retrieves the update region and copies it into a specified region.
GetWindowDC	Retrieves a device context for the entire window.
Invalidate	Invalidates the entire client area.
InvalidateRect	Invalidates the client area within the specified rectangle.
InvalidateRgn	Invalidates the client area within the specified region.
IsWindowVisible	Determines the window's visibility state.
LockWindowUpdate	Disables or enables drawing in the window.
Print	Requests that the window be drawn in a specified device context.
PrintClient	Requests that the window's client area be drawn in a specified device context.
RedrawWindow	Updates a specified rectangle or region in the client area.
ReleaseDC	Releases a device context.
SetRedraw	Sets or clears the redraw flag.
ShowOwnedPopups	Shows or hides the pop-up windows owned by the window.
ShowWindow	Sets the window's show state.
ShowWindowAsync	Sets the show state of a window created by a different thread.
UpdateWindow	Updates the client area.
ValidateRect	Validates the client area within the specified rectangle.
ValidateRgn	Validates the client area within the specified region.

Window Access Methods

ChildWindowFromPoint	Retrieves the child window containing the specified point.
ChildWindowFromPointEx	Retrieves a particular type of child window containing the specified point.
GetLastActivePopup	Retrieves the most recently active pop-up window.
GetParent	Retrieves the immediate parent window.

GetTopLevelParent	Retrieves the top-level parent or owner window.
GetTopLevelWindow	Retrieves the top-level owner window.
GetTopWindow	Retrieves the top-level child window.
GetWindow	Retrieves the specified window.
IsChild	Determines whether the specified window is a child window.
SetParent	Changes the parent window.

Window Size and Position Methods

ArrangeIconicWindows	Arranges all minimized child windows.
BringWindowToTop	Brings the window to the top of the Z order.
CenterWindow	Centers the window against a given window.
DeferWindowPos	Updates the specified multiple-window-position structure for the specified window.
GetClientRect	Retrieves the coordinates of the client area.
GetWindowPlacement	Retrieves the show state and positions.
GetWindowRect	Retrieves the window's bounding dimensions.
GetWindowRgn	Obtains a copy of the window region of a window.
IsIconic	Determines whether the window is minimized.
IsZoomed	Determines whether the window is maximized.
MoveWindow	Changes the window's size and position.
ResizeClient	Resizes the window.
SetWindowPlacement	Sets the show state and positions.
SetWindowPos	Sets the size, position, and Z order.
SetWindowRgn	Sets the window region of a window.

Window State Methods

EnableWindow	Enables or disables input.
IsParentDialog	Determines if the parent window of a control is a dialog window.
IsWindowEnabled	Determines whether the window is enabled for input.
IsWindowUnicode	Determines whether the specified window is a native Unicode window.
IsWindow	Determines whether the specified window handle identifies an existing window.

SetActiveWindow	Activates the window.
SetCapture	Sends all subsequent mouse input to the window.
SetFocus	Sets the input focus to the window.

Window Text Methods

GetWindowText	Retrieves the window's text.
GetWindowTextLength	Retrieves the length of the window's text.
SetWindowText	Changes the window's text.

Window Tree Access Methods

GetDescendantWindow	Retrieves the specified descendant window.
GetDlgCtrlID	Retrieves the window's identifier (for child windows only).
GetDlgItem	Retrieves the specified child window.
SendMessageToDescendants	Sends a message to the specified descendant windows.
SetDlgCtrlID	Changes the window's identifier.

Operators

operator =	Assigns an **HWND** to the **CWindow** object.
operator HWND	Converts the **CWindow** object to an **HWND**.

Data Members

m_hWnd	The handle to the window associated with the **CWindow** object.
rcDefault	Contains default window dimensions.

See Also

Reference

CWindow Class

CWindow Methods

For information about the methods in **CWindow**, see CWindow Members.

CWindow::ArrangeIconicWindows

Arranges all minimized child windows.

```
UINT ArrangeIconicWindows( ) throw();
```

Remarks

See ArrangeIconicWindows in the *Platform SDK*.

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::Attach

Attaches the window identified by *hWndNew* to the **CWindow** object.

```
void Attach(
   HWND hWndNew
) throw();
```

Parameters

hWndNew

[in] The handle to a window.

Example

```
//The following example attaches an HWND to the CWindow object
HWND hWnd  = ::GetDesktopWindow();
CWindow myWindow;
myWindow.Attach(hWnd);
```

See Also

Reference

CWindow Class

CWindow::Detach

Other Resources

CWindow Members

CWindow::BeginPaint

Prepares the window for painting.

```
HDC BeginPaint(
   LPPAINTSTRUCT lpPaint
) throw();
```

Remarks

See BeginPaint in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object
//and calls CWindow::BeginPaint() in the WM_PAINT handler
//of a CWindowImpl-derived class

LRESULT OnPaint(UINT nMsg, WPARAM wParam,LPARAM lParam, BOOL& bHandled)
{
   CWindow myWindow;
   myWindow.Attach(m_hWnd);
   PAINTSTRUCT ps;
   HDC hDC  = myWindow.BeginPaint(&ps);
   //Use the hDC as much as you want
   ...
   ...
   myWindow.EndPaint(&ps);
   return 0L;
}
```

See Also

Reference

CWindow Class

CWindow::EndPaint

Other Resources

CWindow Members

CWindow::BringWindowToTop

Brings the window to the top of the Z order.

```
BOOL BringWindowToTop( ) throw();
```

Remarks

See BringWindowToTop in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::BringWindowToTop() to bring the window to the top
//of the z-order.

CWindow myWindow;
myWindow.Attach(hWnd);
Bool bOnTop = myWindow.BringWindowToTop();

//check if we could bring the window on top
if(bOnTop)
{
//Do something
...
}
```

See Also

Reference

CWindow Class
CWindow::MoveWindow
CWindow::SetWindowPos

Other Resources

CWindow Members

CWindow::CenterWindow

Centers the window against a given window.

```
BOOL CenterWindow(
   HWND hWndCenter = NULL
) throw();
```

Parameters

hWndCenter

[in] The handle to the window against which to center. If this parameter is **NULL** (the default value), the method will set *hWndCenter* to the window's parent window if it is a child window. Otherwise, it will set *hWndCenter* to the window's owner window.

Return Value

TRUE if the window is successfully centered; otherwise, **FALSE**.

Example

```
//The following example attaches various HWNDs to the CWindow objects
//and calls CWindow::CenterWindow() for each of them

CWindow childWindow, popupWindow, overlappedWindow;

childWindow.Attach(hWndChild); //a window created with WS_CHILD style
childWindow.CenterWindow();    //This will center the child
                               //window against its Parent window

popupWindow.Attach(hWndPopup); //a window created with WS_POPUP style
popupWindow.CenterWindow();    //This will center the popup window
                               //against its Owner window

overlappedWindow.Attach(hWndOverlapped); //a window created with
                                         //WS_OVERLAPPED style
overlappedWindow.CenterWindow(::GetDesktopWindow()); //This will center
           //the overlapped window against the DeskTop window
```

See Also

Reference

CWindow Class

CWindow::MoveWindow

CWindow::SetWindowPos

Other Resources

CWindow Members

CWindow::ChangeClipboardChain

Removes the window from the chain of Clipboard viewers.

```
BOOL ChangeClipboardChain(
   HWND hWndNewNext
) throw();
```

Remarks

See ChangeClipboardChain in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::SetClipboardViewer

Other Resources

CWindow Members

CWindow::CheckDlgButton

Changes the check state of the specified button.

```
BOOL CheckDlgButton(
   int nIDButton,
   UINT nCheck
) throw();
```

Remarks

See CheckDlgButton in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::CheckRadioButton
CWindow::IsDlgButtonChecked

Other Resources

CWindow Members

CWindow::CheckRadioButton

Checks the specified radio button.

```
BOOL CheckRadioButton(
   int nIDFirstButton,
   int nIDLastButton,
   int nIDCheckButton
) throw();
```

Remarks

See CheckRadioButton in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::CheckDlgButton

Other Resources

CWindow Members

CWindow::ChildWindowFromPoint

Retrieves the child window containing the specified point.

```
HWND ChildWindowFromPoint(
   POINT point
) const throw();
```

Remarks

See ChildWindowFromPoint in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::ChildWindowFromPointEx
POINT

Other Resources

CWindow Members

CWindow::ChildWindowFromPointEx

Retrieves a particular type of child window containing the specified point.

```
HWND ChildWindowFromPoint(
   POINT point,
   UINT uFlags
) const throw();
```

Remarks

See ChildWindowFromPointEx in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::ChildWindowFromPoint
POINT

Other Resources

CWindow Members

CWindow::ClientToScreen

Converts client coordinates to screen coordinates.

```
BOOL ClientToScreen(
   LPPOINT lpPoint
) const throw();
BOOL ClientToScreen(
   LPRECT lpRect
) const throw();
```

Remarks

See ClientToScreen in the *Platform SDK*.

The second version of this method allows you to convert the coordinates of a RECT structure.

See Also

Reference

CWindow Class
CWindow::ScreenToClient
POINT

Other Resources

CWindow Members

CWindow::Create

Creates a window.

```
HWND Create(
   LPCTSTR lpstrWndClass,
   HWND hWndParent,
   _U_RECT rect = NULL,
   LPCTSTR szWindowName = NULL,
   DWORD dwStyle = 0,
   DWORD dwExStyle = 0,
   _U_MENUorID MenuOrID = 0U,
   LPVOID lpCreateParam = NULL
) throw();
```

Parameters

lpstrWndClass

[in] A pointer to the window's class.

hWndParent

[in] The handle to the parent or owner window.

rect

[in] A variable of type _U_RECT specifying the position of the window. The default value is **NULL**. When this parameter is **NULL**, the value of **CWindow::rcDefault** is used.

szWindowName

[in] Specifies the name of the window. The default value is **NULL**.

dwStyle

[in] The style of the window. The default value is 0, meaning no style is specified. For a list of possible values, see CreateWindow in the *Platform SDK*.

dwExStyle

[in] The extended window style. The default value is 0, meaning no extended style is specified. For a list of possible values, see CreateWindowEx in the *Platform SDK*.

MenuOrID

[in] A variable of type _U_MENUorID specifying a handle to a menu or a window identifier. The default value is 0U.

lpCreateParam

A pointer to the window-creation data contained in a CREATESTRUCT structure.

Return Value

If successful, the handle to the newly created window, specified by m_hWnd. Otherwise, **NULL**.

Remarks

CWindow::rcDefault is defined as `__declspec(selectany) RECT CWindow::rcDefault = {CW_USEDEFAULT, CW_USEDEFAULT, 0, 0};`.

See CreateWindow in the *Platform SDK* for more information.

Note If 0 is used as the value for the *MenuOrID* parameter, it must be specified as 0U (the default value) to avoid a compiler error.

See Also

Reference

CWindow Class

CWindow::m_hWnd

Other Resources

CWindow Members

CWindow::CreateCaret

Creates a new shape for the system caret.

```
BOOL CreateCaret(
   HBITMAP pBitmap
) throw();
```

Remarks

See CreateCaret in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::CreateGrayCaret
CWindow::CreateSolidCaret

Other Resources

CWindow Members

CWindow::CreateGrayCaret

Creates a gray rectangle for the system caret.

```
BOOL CreateGrayCaret(
   int nWidth,
   int nHeight
) throw();
```

Remarks

See CreateCaret in the *Platform SDK*.

Passes (HBITMAP) 1 for the bitmap handle parameter to the Win32 function.

See Also

Reference

CWindow Class
CWindow::CreateCaret
CWindow::CreateSolidCaret

Other Resources

CWindow Members

CWindow::CreateSolidCaret

Creates a solid rectangle for the system caret.

```
BOOL CreateSolidCaret(
   int nWidth,
   int nHeight
) throw();
```

Remarks

See CreateCaret in the *Platform SDK*.

Passes (HBITMAP) 0 for the bitmap handle parameter to the Win32 function.

See Also

Reference

CWindow Class
CWindow::CreateCaret
CWindow::CreateGrayCaret

Other Resources

CWindow Members

CWindow::CWindow

The constructor.

```
CWindow(
   HWND hWnd = NULL
) throw();
```

Parameters

hWnd

[in] The handle to a window.

Remarks

Initializes the m_hWnd member to *hWnd*, which by default is **NULL**.

Note

CWindow::CWindow does not create a window. Classes CWindowImpl, CContainedWindow, and CDialogImpl (all of which derive from **CWindow**) provide a method to create a window or dialog box, which is then assigned to **CWindow::m_hWnd**. You can also use the CreateWindow Win32 function.

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::DeferWindowPos

Updates the specified multiple-window-position structure for the specified window.

```
HDWP DeferWindowPos(
   HDWP hWinPosInfo,
   HWND hWndInsertAfter,
   int x,
   int y,
   int cx,
   int cy,
   UINT uFlags
) throw();
```

Remarks

See DeferWindowPos in the *Platform SDK*.

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::DestroyWindow

Destroys the window associated with the **CWindow** object and sets m_hWnd to **NULL**.

```
BOOL DestroyWindow( ) throw();
```

Remarks

See DestroyWindow in the *Platform SDK*.

It does not destroy the **CWindow** object itself.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::DestroyWindow() to destroy the window

CWindow myWindow;
myWindow.Attach(hWndChild);
//call the CWindow wrappers
...
...
...
myWindow.DestroyWindow();
hWndChild = NULL;
```

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::Detach

Detaches m_hWnd from the **CWindow** object and sets **m_hWnd** to **NULL**.

```
HWND Detach( ) throw();
```

Return Value

The **HWND** associated with the **CWindow** object.

Example

```
//The following example attaches an HWND to the CWindow object and
//later detaches the CWindow object from the HWND when no longer needed

CWindow myWindow;
myWindow.Attach(hWnd);
//call CWindow wrappers
...
...
...

//We don't need the C++ object any more, so detach it from the HWND.
myWindow.Detach();
```

See Also

Reference

CWindow Class

CWindow::Attach

Other Resources

CWindow Members

CWindow::DlgDirList

Fills a list box with the names of all files matching a specified path or file name.

```
int DlgDirList(
   LPTSTR lpPathSpec,
   int nIDListBox,
   int nIDStaticPath,
   UINT nFileType
) throw();
```

Remarks

See DlgDirList in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::DlgDirListComboBox
CWindow::DlgDirSelect
CWindow::DlgDirSelectComboBox

Other Resources

CWindow Members

CWindow::DlgDirListComboBox

Fills a combo box with the names of all files matching a specified path or file name.

```
int DlgDirListComboBox(
   LPTSTR lpPathSpec,
   int nIDComboBox,
   int nIDStaticPath,
   UINT nFileType
) throw();
```

Remarks

See DlgDirListComboBox in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::DlgDirList
CWindow::DlgDirSelect
CWindow::DlgDirSelectComboBox

Other Resources

CWindow Members

CWindow::DlgDirSelect

Retrieves the current selection from a list box.

```
BOOL DlgDirSelect(
   LPTSTR lpString,
   int nCount,
   int nIDListBox
) throw();
```

Remarks

See DlgDirSelectEx in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::DlgDirSelectComboBox
CWindow::DlgDirList

Other Resources

CWindow Members

CWindow::DlgDirSelectComboBox

Retrieves the current selection from a combo box.

```
BOOL DlgDirSelectComboBox(
   LPTSTR lpString,
   int nCount,
   int nIDComboBox
) throw();
```

Remarks

See DlgDirSelectComboBoxEx in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::DlgDirSelect
CWindow::DlgDirListComboBox

Other Resources

CWindow Members

CWindow::DragAcceptFiles

Registers whether the window accepts dragged files.

```
void DragAcceptFiles(
   BOOL bAccept = TRUE
);
```

Remarks

See DragAcceptFiles in the *Platform SDK*.

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::DrawMenuBar

Redraws the window's menu bar.

```
BOOL DrawMenuBar( ) throw();
```

Remarks

See DrawMenuBar in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::GetMenu
CWindow::SetMenu

Other Resources

CWindow Members

CWindow::EnableScrollBar

Enables or disables the scroll bar arrows.

```
BOOL EnableScrollBar(
   UINT uSBFlags,
   UINT uArrowFlags = ESB_ENABLE_BOTH
) throw();
```

Remarks

See EnableScrollBar in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::ShowScrollBar

Other Resources

CWindow Members

CWindow::EnableWindow

Enables or disables input.

```
BOOL EnableWindow(
   BOOL bEnable = TRUE
) throw();
```

Remarks

See EnableWindow in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::EnableWindow() to enable and disable the window
//wrapped by the CWindow object

CWindow myWindow;
myWindow.Attach(hWnd);

//The following call enables the window
//CWindow::EnableWindow() takes TRUE as the default parameter

BOOL bEnable  = myWindow.EnableWindow();

if(bEnable)
{
//Do something now that the window is enabled
...
...
//Now it's time to disable the window again
bEnable = myWindow.EnableWindow(FALSE);
}
```

See Also

Reference

CWindow Class

CWindow::IsWindowEnabled

Other Resources

CWindow Members

CWindow::EndPaint

Marks the end of painting.

```
void EndPaint(
   LPPAINTSTRUCT lpPaint
) throw();
```

Remarks

See EndPaint in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::EndPaint() in the WM_PAINT handler of a
//CWindowImpl-derived class

LRESULT OnPaint(UINT nMsg, WPARAM wParam,LPARAM lParam, BOOL& bHandled)
{
   ...
   CWindow myWindow;
   myWindow.Attach(m_hWnd);
   PAINTSTRUCT ps;
   HDC hDC  = myWindow.BeginPaint(&ps);
   //Use hDC as much as you want
   ...
   ...
   myWindow.EndPaint(&ps);
   return 0L;
}
```

See Also

Reference

CWindow Class

CWindow::BeginPaint

Other Resources

CWindow Members

CWindow::FlashWindow

Flashes the window once.

```
BOOL FlashWindow(
   BOOL bInvert
) throw();
```

Remarks

See FlashWindow in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::SetTimer

Other Resources

CWindow Members

CWindow::GetClientRect

Retrieves the coordinates of the client area.

```
BOOL GetClientRect(
   LPRECT lpRect
) const throw();
```

Remarks

See GetClientRect in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::GetClientRect() to get the client area rectangle
//of the window

CWindow myWindow;
myWindow.Attach(hWnd);
RECT rc;
myWindow.GetClientRect(&rc);
```

See Also

Reference

CWindow Class

CWindow::GetWindowRect

RECT

Other Resources

CWindow Members

CWindow::GetDC

Retrieves a device context for the client area.

```
HDC GetDC( ) throw();
```

Remarks

See GetDC in the *Platform SDK*.

Example

```
// The following example attaches a HWND to the CWindow object
// and calls CWindow::GetDC to retrieve the DC of the client
// area of the window wrapped by CWindow Object.

CWindow myWindow;
myWindow.Attach(hWnd);
HDC hDC = myWindow.GetDC();
```

See Also

Reference

CWindow Class
CWindow::GetDCEx
CWindow::GetWindowDC
CWindow::ReleaseDC

Other Resources

CWindow Members

CWindow::GetDCEx

Retrieves a device context for the client area and allows clipping options.

```
HDC GetDCEx(
   HRGN hRgnClip,
   DWORD flags
) throw();
```

Remarks

See GetDCEx in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::GetDC
CWindow::GetWindowDC
CWindow::ReleaseDC

Other Resources

CWindow Members

CWindow::GetDescendantWindow

Finds the descendant window specified by the given identifier.

```
HWND GetDescendantWindow(
   int nID
) const throw();
```

Parameters

nID

[in] The identifier of the descendant window to be retrieved.

Return Value

The handle to a descendant window.

Remarks

GetDescendantWindow searches the entire tree of child windows, not only the windows that are immediate children.

See Also

Reference

CWindow Class

CWindow::GetDlgItem

Other Resources

CWindow Members

CWindow::GetDlgControl

Call this function to get a pointer to an interface of an ActiveX control that is hosted by a composite control or a control-hosting dialog.

```
HRESULT GetDlgControl(
   int nID,
   REFIID iid,
   void** ppCtrl
) throw();
```

Parameters

nID

[in] The resource ID of the control being retrieved.

iid

[in] The ID of the interface you would like to get from the control.

ppCtrl

[out] The pointer to the interface.

Return Value

Returns **S_OK** on success or any valid error **HRESULT**. For example, the function returns **E_FAIL** if the control specified by *nID* cannot be found and it returns **E_NOINTERFACE** if the control can be found, but it doesn't support the interface specified by *iid*.

Remarks

Using this pointer, you can call methods on the interface.

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::GetDlgCtrlID

Retrieves the window's identifier (for child windows only).

```
int GetDlgCtrlID( ) const throw();
```

Remarks

See GetDlgCtrlID in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::SetDlgCtrlID

Other Resources

CWindow Members

CWindow::GetDlgHost

Retrieves a pointer to an interface to the ATL Control hosting container.

```
HRESULT GetDlgHost(
   int nID,
   REFIID iid,
   void** ppHost
) throw();
```

Parameters

nID

[in] The resource ID of the control being retrieved.

iid

[in] The ID of the interface you would like to get from the control.

ppHost

[out] The pointer to the interface.

Return Value

Returns **S_OK** if the window specified by *iid* is a Control Container, and the requested interface could be retrieved. Returns **E_FAIL** if the window is not a Control Container, or if the interface requested could not be retrieved. If a window with the specified ID could not be found, then the return value is equal to HRESULT_FROM_WIN32(ERROR_CONTROL_ID_NOT_FOUND).

Remarks

Using this pointer, you can call methods on the interface.

See Also

Reference

CWindow Class

AtlAxGetHost

Other Resources

CWindow Members

CWindow::GetDlgItem

Retrieves the specified child window.

```
HWND GetDlgItem(
   int nID
) const throw();
```

Remarks

See GetDlgItem in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::GetDescendantWindow

Other Resources

CWindow Members

CWindow::GetDlgItemInt

Translates a control's text to an integer.

```
UINT GetDlgItemInt(
   int nID,
   BOOL* lpTrans = NULL,
   BOOL bSigned = TRUE
) const throw();
```

Remarks

See GetDlgItemInt in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::SetDlgItemInt
CWindow::GetDlgItemText

Other Resources

CWindow Members

CWindow::GetDlgItemText

Retrieves a control's text.

```
UINT GetDlgItemText(
   int nID,
   LPTSTR lpStr,
   int nMaxCount
) const throw();
BOOL GetDlgItemText(
   int nID,
   BSTR& bstrText
) const throw();
```

Remarks

See GetDlgItemText in the *Platform SDK*.

Remarks

The second version of this method allows you to copy the control's text to a **BSTR**. This version returns **TRUE** if the text is successfully copied; otherwise, **FALSE**.

See Also

Reference

CWindow Class

CWindow::SetDlgItemText

CWindow::GetDlgItem

Other Resources

CWindow Members

CWindow::GetExStyle

Retrieves the extended window styles of the window.

```
DWORD GetExStyle( ) const throw();
```

Return Value

The window's extended styles.

Remarks

To retrieve the regular window styles, call GetStyle.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::GetExStyle() to retrieve the extended styles of
//the window

CWindow myWindow;
myWindow.Attach(hWnd);
DWORd dwExStyles = myWindow.GetExStyle();
```

See Also

Reference

CWindow Class

CWindow::ModifyStyleEx

Other Resources

CWindow Members

CWindow::GetFont

Retrieves the window's current font by sending a WM_GETFONT message to the window.

```
HFONT GetFont( ) const throw();
```

Return Value

A font handle.

See Also

Reference

CWindow Class

CWindow::SetFont

Other Resources

CWindow Members

CWindow::GetHotKey

Determines the hot key associated with the window by sending a **WM_GETHOTKEY** message.

```
DWORD GetHotKey( ) const throw();
```

Return Value

The virtual key code and modifiers for the hot key associated with the window. For a list of possible modifiers, see WM_GETHOTKEY in the *Platform SDK*. For a list of of standard virtual key codes, see Winuser.h.

See Also

Reference

CWindow Class

CWindow::SetHotKey

Other Resources

CWindow Members

CWindow::GetIcon

Retrieves the handle to the window's large or small icon.

```
HICON GetIcon( BOOL bBigIcon = TRUE ) const;
```

Parameters

bBigIcon

[in] If **TRUE** (the default value) the method returns the large icon. Otherwise, it returns the small icon.

Return Value

An icon handle.

Remarks

GetIcon sends a WM_GETICON message to the window.

See Also

Reference

CWindow Class

CWindow::SetIcon

Other Resources

CWindow Members

CWindow::GetLastActivePopup

Retrieves the most recently active pop-up window.

```
HWND GetLastActivePopup( ) const throw();
```

Remarks

See GetLastActivePopup in the *Platform SDK*.

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::GetMenu

Retrieves the window's menu.

```
HMENU GetMenu( ) const throw();
```

Remarks

See GetMenu in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::SetMenu

Other Resources

CWindow Members

CWindow::GetNextDlgGroupItem

Retrieves the previous or next control within a group of controls.

```
HWND GetNextDlgGroupItem(
   HWND hWndCtl,
   BOOL bPrevious = FALSE
) const throw();
```

Remarks

See GetNextDlgGroupItem in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::GetNextDlgTabItem

Other Resources

CWindow Members

CWindow::GetNextDlgTabItem

Retrieves the previous or next control having the **WS_TABSTOP** style.

```
HWND GetNextDlgTabItem(
   HWND hWndCtl,
   BOOL bPrevious = FALSE
) const throw();
```

Remarks

See GetNextDlgTabItem in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::GetNextDlgGroupItem

Other Resources

CWindow Members

CWindow::GetParent

Retrieves the immediate parent window.

```
HWND GetParent( ) const throw();
```

Remarks

See GetParent in the *Platform SDK*.

Example

```
// The following example attaches a HWND to the CWindow object
// and calls CWindow::GetParent to find out the parent
// window of the window wrapped by CWindow object.

CWindow myWindow;
myWindow.Attach(hWnd);
HWND hWndParent = myWindow.GetParent();
```

See Also

Reference

CWindow Class

CWindow::SetParent

Other Resources

CWindow Members

CWindow::GetScrollInfo

Retrieves the parameters of a scroll bar.

```
BOOL GetScrollInfo(
   int nBar,
   LPSCROLLINFO lpScrollInfo
) throw();
```

Remarks

See GetScrollInfo in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::SetScrollInfo

Other Resources

CWindow Members

CWindow::GetScrollPos

Retrieves the position of the scroll box.

```
int GetScrollPos(
   int nBar
) const throw();
```

Remarks

See GetScrollPos in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::SetScrollPos

Other Resources

CWindow Members

CWindow::GetScrollRange

Retrieves the scroll bar range.

```
BOOL GetScrollRange(
   int nBar,
   LPINT lpMinPos,
   LPINT lpMaxPos
) const throw();
```

Remarks

See GetScrollRange in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::SetScrollRange

Other Resources

CWindow Members

CWindow::GetStyle

Retrieves the window styles of the window.

```
DWORD GetStyle( ) const throw();
```

Return Value

The window's styles.

Remarks

To retrieve the extended window styles, call GetExStyle.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::GetStyle() to retrieve the styles of the window

CWindow myWindow;
myWindow.Attach(hWnd);
DWORd dwStyles = myWindow.GetStyle();
```

See Also

Reference

CWindow Class
CWindow::ModifyStyle
Window Styles

Other Resources

CWindow Members

CWindow::GetSystemMenu

Creates a copy of the system menu for modification.

```
HMENU GetSystemMenu(
   BOOL bRevert
) const throw();
```

Remarks

See GetSystemMenu in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::GetMenu

Other Resources

CWindow Members

CWindow::GetTopLevelParent

Retrieves the window's top-level parent window.

```
HWND GetTopLevelParent( ) const throw();
```

Return Value

The handle to the top-level parent window.

See Also

Reference

CWindow Class
CWindow::GetParent
CWindow::GetTopLevelWindow
CWindow::GetWindow

Other Resources

CWindow Members

CWindow::GetTopLevelWindow

Retrieves the window's top-level parent or owner window.

```
HWND GetTopLevelWindow( ) const throw();
```

Return Value

The handle to the top-level owner window.

See Also

Reference

CWindow Class
CWindow::GetTopLevelParent
CWindow::GetWindow

Other Resources

CWindow Members

CWindow::GetTopWindow

Retrieves the top-level child window.

```
HWND GetTopWindow( ) const throw();
```

Remarks

See GetTopWindow in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::GetTopWindow() to get the top-level child window

CWindow myWindow;
myWindow.Attach(hWnd);
HWND hWndFavoriteChild = myWindow.GetTopWindow();
```

See Also

Reference

CWindow Class

CWindow::GetWindow

Other Resources

CWindow Members

CWindow::GetUpdateRect

Retrieves the coordinates of the smallest rectangle that completely encloses the update region.

```
BOOL GetUpdateRect(
   LPRECT lpRect,
   BOOL bErase = FALSE
) throw();
```

Remarks

See GetUpdateRect in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::GetUpdateRgn
RECT

Other Resources

CWindow Members

CWindow::GetUpdateRgn

Retrieves the update region and copies it into a specified region.

```
int GetUpdateRgn(
   HRGN hRgn,
   BOOL bErase = FALSE
) throw();
```

Remarks

See GetUpdateRgn in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::GetUpdateRect

Other Resources

CWindow Members

CWindow::GetWindow

Retrieves the specified window.

```
HWND GetWindow(
   UINT nCmd
) const throw();
```

Remarks

See GetWindow in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::GetTopWindow
CWindow::GetTopLevelParent
CWindow::GetTopLevelWindow

Other Resources

CWindow Members

CWindow::GetWindowContextHelpId

Retrieves the window's help context identifier.

```
DWORD GetWindowContextHelpId( ) const throw();
```

Remarks

See GetWindowContextHelpId in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::SetWindowContextHelpId

Other Resources

CWindow Members

CWindow::GetWindowDC

Retrieves a device context for the entire window.

```
HDC GetWindowDC( ) throw();
```

Remarks

See GetWindowDC in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::GetWindowDC() to retrieve the DC of the entire window

CWindow myWindow;
myWindow.Attach(hWnd);
HDC hDC = myWindow.GetWindowDC();
```

See Also

Reference

CWindow Class
CWindow::GetDC
CWindow::GetDCEx
CWindow::ReleaseDC

Other Resources

CWindow Members

CWindow::GetWindowLong

Retrieves a 32-bit value at a specified offset into the extra window memory.

```
LONG GetWindowLong(
   int nIndex
) const throw();
```

Remarks

See GetWindowLong in the *Platform SDK*.

Note
To write code that is compatible with both 32-bit and 64-bit versions of Windows, use CWindow::GetWindowLongPtr.

See Also

Reference

CWindow Class
CWindow::SetWindowLong
CWindow::GetWindowWord

Other Resources

CWindow Members

CWindow::GetWindowLongPtr

Retrieves information about the specified window, including a value at a specified offset into the extra window memory.

```
LONG_PTR GetWindowLongPtr(
   int nIndex
) const throw( );
```

Remarks

See GetWindowLongPtr in the *Platform SDK*.

Remarks

If you are retrieving a pointer or a handle, this function supersedes the **CWindow::GetWindowLong** method.

Note
Pointers and handles are 32 bits on 32-bit Windows and 64 bits on 64-bit Windows.

To write code that is compatible with both 32-bit and 64-bit versions of Windows, use **CWindow::GetWindowLongPtr**.

See Also

Reference

CWindow Class
CWindow::SetWindowLongPtr
CWindow::GetWindowLong

Other Resources

CWindow Members

CWindow::GetWindowPlacement

Retrieves the show state and positions.

```
BOOL GetWindowPlacement(
   WINDOWPLACEMENT FAR* lpwndpl
) const throw();
```

Remarks

See GetWindowPlacement in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::SetWindowPlacement
WINDOWPLACEMENT

Other Resources

CWindow Members

CWindow::GetWindowProcessID

Retrieves the identifier of the process that created the window.

```
DWORD GetWindowProcessID( ) throw();
```

Remarks

See GetWindowThreadProcessID in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::GetWindowProcessID() to retrieve the id of the
//process that created the window

CWindow myWindow;
myWindow.Attach(hWnd);
DWORD dwID = myWindow.GetWindowProcessID();
```

See Also

Reference

CWindow Class

CWindow::GetWindowThreadID

Other Resources

CWindow Members

CWindow::GetWindowRect

Retrieves the window's bounding dimensions.

```
BOOL GetWindowRect(
   LPRECT lpRect
) const throw();
```

Remarks

See GetWindowRect in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::GetClientRect
RECT

Other Resources

CWindow Members

CWindow::GetWindowRgn

Obtains a copy of the window region of a window.

```
int GetWindowRgn(
   HRGN hRgn
) throw();
```

Remarks

See GetWindowRgn in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::SetWindowRgn

Other Resources

CWindow Members

CWindow::GetWindowText

Retrieves the window's text.

```
int GetWindowText(
   LPTSTR lpszStringBuf,
   int nMaxCount
) const throw();
BOOL GetWindowText(
   BSTR& bstrText
) throw();
```

Remarks

See GetWindowText in the *Platform SDK*.

The second version of this method allows you to store the text in a **BSTR**. If the text is successfully copied, the return value is **TRUE**; otherwise, the return value is **FALSE**.

See Also

Reference

CWindow Class

CWindow::GetWindowTextLength

CWindow::SetWindowText

Other Resources

CWindow Members

CWindow::GetWindowTextLength

Retrieves the length of the window's text.

```
int GetWindowTextLength( ) const throw();
```

Remarks

See GetWindowTextLength in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::GetWindowText

Other Resources

CWindow Members

CWindow::GetWindowThreadID

Retrieves the identifier of the thread that created the specified window.

```
DWORD GetWindowThreadID( ) throw();
```

Remarks

See GetWindowThreadProcessID in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::GetWindowThreadID() to retrieve the id of the thread
//that created the window

CWindow myWindow;
myWindow.Attach(hWnd);
DWORD dwID = myWindow.GetWindowThreadID();
```

See Also

Reference

CWindow Class

CWindow::GetWindowProcessID

Other Resources

CWindow Members

CWindow::GetWindowWord

Retrieves a 16-bit value at a specified offset into the extra window memory.

```
WORD GetWindowWord(
   int nIndex
) const throw();
```

Remarks

See GetWindowLong in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::SetWindowWord
CWindow::GetWindowLong

Other Resources

CWindow Members

CWindow::GotoDlgCtrl

Sets the keyboard focus to a control in the dialog box.

```
void GotoDlgCtrl(
   HWND hWndCtrl
) const throw();
```

Remarks

See WM_NEXTDLGCTL in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::NextDlgCtrl
CWindow::PrevDlgCtrl

Other Resources

CWindow Members

CWindow::HideCaret

Hides the system caret.

```
BOOL HideCaret( ) throw();
```

Remarks

See HideCaret in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::HideCaret() to hide the caret of the window owning
//the caret

CWindow myWindow;
myWindow.Attach(hWndEdit);
myWindow.HideCaret();
```

See Also

Reference

CWindow Class

CWindow::ShowCaret

Other Resources

CWindow Members

CWindow::HiliteMenuItem

Highlights or removes the highlight from a top-level menu item.

```
BOOL HiliteMenuItem(
   HMENU hMenu,
   UINT uHiliteItem,
   UINT uHilite
) throw();
```

Remarks

See HiliteMenuItem in the *Platform SDK*.

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::Invalidate

Invalidates the entire client area.

```
BOOL Invalidate(
   BOOL bErase = TRUE
) throw();
```

Remarks

See InvalidateRect in the *Platform SDK*.

Passes **NULL** for the **RECT** parameter to the **InvalidateRect** Win32 function.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::Invalidate() to invalidate the entire client area

CWindow myWindow;
myWindow.Attach(hWnd);
myWindow.Invalidate();
```

See Also

Reference

CWindow Class
CWindow::InvalidateRect
CWindow::InvalidateRgn
CWindow::ValidateRect
CWindow::ValidateRgn

Other Resources

CWindow Members

CWindow::InvalidateRect

Invalidates the client area within the specified rectangle.

```
BOOL InvalidateRect(
   LPCRECT lpRect,
   BOOL bErase = TRUE
) throw();
```

Remarks

See InvalidateRect in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::Invalidate
CWindow::InvalidateRgn
CWindow::ValidateRect
RECT

Other Resources

CWindow Members

CWindow::InvalidateRgn

Invalidates the client area within the specified region.

```
void InvalidateRgn(
   HRGN hRgn,
   BOOL bErase = TRUE
) throw();
```

Remarks

See InvalidateRgn in the *Platform SDK*.

Remarks

Specifies a **void** return type, while the **InvalidateRgn** Win32 function always returns **TRUE**.

See Also

Reference

CWindow Class
CWindow::Invalidate
CWindow::InvalidateRect
CWindow::ValidateRgn

Other Resources

CWindow Members

CWindow::IsChild

Determines whether the specified window is a child window.

```
BOOL IsChild(
   const HWND hWnd
) const throw();
```

Remarks

See IsChild in the *Platform SDK*.

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::IsDialogMessage

Determines whether a message is intended for the specified dialog box.

```
BOOL IsDialogMessage(
   LPMSG lpMsg
) throw();
```

Remarks

See IsDialogMessage in the *Platform SDK*.

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::IsDlgButtonChecked

Determines the check state of the button.

```
UINT IsDlgButtonChecked(
   int nIDButton
) const throw();
```

Remarks

See IsDlgButtonChecked in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::CheckDlgButton

Other Resources

CWindow Members

CWindow::IsIconic

Determines whether the window is minimized.

```
BOOL IsIconic( ) const throw();
```

Remarks

See IsIconic in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::IsIconic() to determine if the window is minimized

CWindow myWindow;
myWindow.Attach(hWnd);
BOOL bIconic = myWindow.IsIconic();
```

See Also

Reference

CWindow Class

CWindow::IsZoomed

Other Resources

CWindow Members

CWindow::IsParentDialog

Determines if the parent window of the control is a dialog window.

```
BOOL IsParentDialog( ) throw( );
```

Return Value

Returns TRUE if the parent window is a dialog, FALSE otherwise.

See Also

Reference

CWindow Class

CWindow::IsWindow

Other Resources

CWindow Members

CWindow::IsWindow

Determines whether the specified window handle identifies an existing window.

```
BOOL IsWindow( ) throw();
```

Remarks

See IsWindow in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::IsWindow() to verify if the HWND corresponds
//to an existing window

CWindow myWindow;
myWindow.Attach(hWnd);
BOOL bWindow = myWindow.IsWindow();
```

See Also

Reference

CWindow Class
CWindow::IsWindowEnabled
CWindow::IsWindowVisible
CWindow::IsWindowUnicode
CWindow::IsParentDialog

Other Resources

CWindow Members

CWindow::IsWindowEnabled

Determines whether the window is enabled for input.

```
BOOL IsWindowEnabled( ) const throw();
```

Remarks

See IsWindowEnabled in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::IsWindowEnabled() to verify if the window is enabled
//for receiving input

CWindow myWindow;
myWindow.Attach(hWnd);
BOOL bEnabled = myWindow.IsWindowEnabled();
```

See Also

Reference

CWindow Class
CWindow::EnableWindow
CWindow::IsWindowVisible
CWindow::IsWindowUnicode

Other Resources

CWindow Members

CWindow::IsWindowVisible

Determines the window's visibility state.

```
BOOL IsWindowVisible( ) const throw();
```

Remarks

See IsWindowVisible in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::IsWindowVisible() to determine the visibility state
//of the window

CWindow myWindow;
myWindow.Attach(hWnd);
BOOL bVisible = myWindow.IsWindowVisible();
```

See Also

Reference

CWindow Class
CWindow::IsWindow
CWindow::IsWindowUnicode

Other Resources

CWindow Members

CWindow::IsWindowUnicode

Determines whether the specified window is a native Unicode window.

```
BOOL IsWindowUnicode( ) throw();
```

Remarks

See IsWindowUnicode in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::IsWindowUnicode() to determine if the window is a
//UNICODE window or an ANSI one.

CWindow myWindow;
myWindow.Attach(hWnd);
BOOL bUnicode = myWindow.IsWindowUnicode();
```

See Also

Reference

CWindow Class
CWindow::IsWindow
CWindow::IsWindowEnabled
CWindow::IsWindowVisible

Other Resources

CWindow Members

CWindow::IsZoomed

Determines whether the window is maximized.

```
BOOL IsZoomed( ) const throw();
```

Remarks

See IsZoomed in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::IsIconic

Other Resources

CWindow Members

CWindow::KillTimer

Destroys a timer event created by **CWindow::SetTimer**.

```
BOOL KillTimer(
   UINT nIDEvent
) throw();
```

Remarks

See KillTimer in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::SetTimer

Other Resources

CWindow Members

CWindow::LockWindowUpdate

Disables or enables drawing in the window by calling the LockWindowUpdate Win32 function.

```
BOOL LockWindowUpdate(
   BOOL bLock = TRUE
) throw();
```

Parameters

bLock

[in] If **TRUE** (the default value), the window will be locked. Otherwise, it will be unlocked.

Return Value

TRUE if the window is successfully locked; otherwise, **FALSE**.

Remarks

If *bLock* is **TRUE**, this method passes m_hWnd to the Win32 function; otherwise, it passes **NULL**.

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::MapWindowPoints

Converts a set of points from the window's coordinate space to the coordinate space of another window.

```
int MapWindowPoints(
   HWND hWndTo,
   LPPOINT lpPoint,
   UINT nCount
) const throw();
int MapWindowPoints(
   HWND hWndTo,
   LPRECT lpRect
) const throw();
```

Remarks

See MapWindowPoints in the *Platform SDK*.

The second version of this method allows you to convert the coordinates of a RECT structure.

See Also

Reference

CWindow Class

POINT

Other Resources

CWindow Members

CWindow::MessageBox

Displays a message box.

```
int MessageBox(
   LPCTSTR lpszText,
   LPCTSTR lpszCaption = NULL,
   UINT nType = MB_OK
) throw();
```

Remarks

See MessageBox in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::MessageBox() to pop up a Windows message box

CWindow myWindow;
myWindow.Attach(hWnd);
myWindow.MessageBox(_T("Hello World"));
```

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::ModifyStyle

Modifies the window styles of the **CWindow** object.

```
BOOL ModifyStyle(
   DWORD dwRemove,
   DWORD dwAdd,
   UINT nFlags = 0
) throw();
```

Parameters

dwRemove

[in] Specifies the window styles to be removed during style modification.

dwAdd

[in] Specifies the window styles to be added during style modification.

nFlags

[in] Window-positioning flags. For a list of possible values, see the SetWindowPos function in the *Platform SDK*.

Return Value

TRUE if the window styles are modified; otherwise, **FALSE**.

Remarks

Styles to be added or removed can be combined by using the bitwise OR (|) operator. See the CreateWindow function in the *Platform SDK* for information about the available window styles.

If *nFlags* is nonzero, **ModifyStyle** calls the Win32 function **SetWindowPos**, and redraws the window by combining *nFlags* with the following four flags:

- **SWP_NOSIZE** Retains the current size.
- **SWP_NOMOVE** Retains the current position.
- **SWP_NOZORDER** Retains the current Z order.
- **SWP_NOACTIVATE** Does not activate the window.

To modify a window's extended styles, call ModifyStyleEx.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::ModifyStyle() to add and remove the window styles

CWindow myWindow;
myWindow.Attach(hWnd);

//The following line removes the WS_CLIPCHILDREN style from the
//window and adds the WS_CAPTION style to the window
myWindow.ModifyStyle(WS_CLIPCHILDREN,WS_CAPTION);
```

See Also

Reference

CWindow Class

CWindow::GetStyle

Other Resources

CWindow Members

CWindow::ModifyStyleEx

Modifies the extended window styles of the **CWindow** object.

```
BOOL ModifyStyleEx(
   DWORD dwRemove,
   DWORD dwAdd,
   UINT nFlags = 0
) throw();
```

Parameters

dwRemove

[in] Specifies the extended styles to be removed during style modification.

dwAdd

[in] Specifies the extended styles to be added during style modification.

nFlags

[in] Window-positioning flags. For a list of possible values, see the SetWindowPos function in the *Platform SDK*.

Return Value

TRUE if the extended window styles are modified; otherwise, **FALSE**.

Remarks

Styles to be added or removed can be combined by using the bitwise OR (|) operator. See the CreateWindowEx function in the *Platform SDK* for information about the available extended styles.

If *nFlags* is nonzero, **ModifyStyleEx** calls the Win32 function **SetWindowPos**, and redraws the window by combining *nFlags* with the following four flags:

- **SWP_NOSIZE** Retains the current size.
- **SWP_NOMOVE** Retains the current position.
- **SWP_NOZORDER** Retains the current Z order.
- **SWP_NOACTIVATE** Does not activate the window.

To modify windows using regular window styles, call ModifyStyle.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::ModifyStyleEx() to add and remove the extended
//window styles

CWindow myWindow;
myWindow.Attach(hWnd);

//The following line removes WS_EX_CONTEXTHELP extended style from
//the window and adds WS_EX_TOOLWINDOW extended style to the window

myWindow.ModifyStyleEx(WS_EX_CONTEXTHELP,WS_EX_TOOLWINDOW);
```

See Also

Reference

CWindow Class

CWindow::GetExStyle

Other Resources

CWindow Members

CWindow::MoveWindow

Changes the window's size and position.

```
BOOL MoveWindow(
   int x,
   int y,
   int nWidth,
   int nHeight,
   BOOL bRepaint = TRUE
) throw();
BOOL MoveWindow(
   LPCRECT lpRect,
   BOOL bRepaint = TRUE
) throw();
```

Remarks

For a top-level window object, the x and y parameters are relative to the upper-left corner of the screen. For a child window object, they are relative to the upper-left corner of the parent window's client area.

The second version of this method uses a RECT structure to determine the window's new position, width, and height.

See Also

Reference

CWindow Class

CWindow::SetWindowPos

Other Resources

CWindow Members

CWindow::NextDlgCtrl

Sets the keyboard focus to the next control in the dialog box.

```
void NextDlgCtrl( ) const throw();
```

Remarks

See WM_NEXTDLGCTL in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::PrevDlgCtrl
CWindow::GotoDlgCtrl

Other Resources

CWindow Members

CWindow::OpenClipboard

Opens the Clipboard.

```
BOOL OpenClipboard( ) throw();
```

Remarks

See OpenClipboard in the *Platform SDK*.

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::PostMessage

Places a message in the message queue associated with the thread that created the window.

```
BOOL PostMessage(
   UINT message,
   WPARAM wParam = 0,
   LPARAM lParam = 0
) throw();
```

Remarks

See PostMessage in the *Platform SDK*.

Returns without waiting for the thread to process the message.

Example

```
//The following example attaches an HWND to the CWindow object and
//posts a WM_PAINT message to the Window wrapped by the CWindow object
//using CWindow::PostMessage() with the default values of WPARAM and
//LPARAM

CWindow myWindow;
myWindow.Attach(hWnd);
myWindow.PostMessage(WM_PAINT);
```

See Also

Reference

CWindow Class
CWindow::SendMessage
CWindow::SendNotifyMessage

Other Resources

CWindow Members

CWindow::PrevDlgCtrl

Sets the keyboard focus to the previous control in the dialog box.

```
void PrevDlgCtrl( ) const throw();
```

Remarks

See WM_NEXTDLGCTL in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::NextDlgCtrl
CWindow::GotoDlgCtrl

Other Resources

CWindow Members

CWindow::Print

Sends a WM_PRINT message to the window to request that it draw itself in the specified device context.

```
void Print(
   HDC hDC,
   DWORD dwFlags
) const throw();
```

Parameters

hDC

[in] The handle to a device context.

dwFlags

[in] Specifies the drawing options. You can combine one or more of the following flags:

- **PRF_CHECKVISIBLE** Draw the window only if it is visible.
- **PRF_CHILDREN** Draw all visible child windows.
- **PRF_CLIENT** Draw the client area of the window.
- **PRF_ERASEBKGND** Erase the background before drawing the window.
- **PRF_NONCLIENT** Draw the non-client area of the window.
- **PRF_OWNED** Draw all owned windows.

See Also

Reference

CWindow Class

CWindow::PrintClient

Other Resources

CWindow Members

CWindow::PrintClient

Sends a WM_PRINTCLIENT message to the window to request that it draw its client area in the specified device context.

```
void PrintClient(
   HDC hDC,
   DWORD dwFlags
) const throw();
```

Parameters

hDC

[in] The handle to a device context.

dwFlags

[in] Specifies drawing options. You can combine one or more of the following flags:

- **PRF_CHECKVISIBLE** Draw the window only if it is visible.
- **PRF_CHILDREN** Draw all visible child windows.
- **PRF_CLIENT** Draw the client area of the window.
- **PRF_ERASEBKGND** Erase the background before drawing the window.
- **PRF_NONCLIENT** Draw the non-client area of the window.
- **PRF_OWNED** Draw all owned windows.

See Also

Reference

CWindow Class

CWindow::Print

Other Resources

CWindow Members

CWindow::RedrawWindow

Updates a specified rectangle or region in the client area.

```
BOOL RedrawWindow(
   LPCRECT lpRectUpdate = NULL,
   HRGN hRgnUpdate = NULL,
   UINT flags = RDW_INVALIDATE | RDW_UPDATENOW | RDW_ERASE
); throw()
```

Remarks

See RedrawWindow in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::RedrawWindow() to update the entire window using the
//default arguments

CWindow myWindow;
myWindow.Attach(hWnd);
BOOL bRedrawn = myWindow.RedrawWindow();
```

See Also

Reference

CWindow Class

CWindow::UpdateWindow

RECT

Other Resources

CWindow Members

CWindow::ReleaseDC

Releases a device context.

```
int ReleaseDC(
   HDC hDC
);
```

Remarks

See ReleaseDC in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::ReleaseDC() to release the DC

CWindow myWindow;
myWindow.Attach(hWnd);
HDC hDC = myWindow.GetDC();
//Use the DC
...
...
...
myWindow.ReleaseDC(hDC);
hDC = NULL;
```

See Also

Reference

CWindow Class
CWindow::GetDC
CWindow::GetDCEx
CWindow::GetWindowDC

Other Resources

CWindow Members

CWindow::ResizeClient

Resizes the window to the specified client area size.

```
BOOL ResizeClient(
   int nWidth,
   int nHeight,
   BOOL bRedraw = FALSE
) throw();
```

Parameters

nWidth

New width of the window in pixels.

nHeight

New height of the window in pixels.

bRedraw

A flag indicating whether to redraw changes. Default is **FALSE**, indicating the window does not redraw changes.

See Also

Reference

CWindow Class
CWindow::SetWindowPos
SetWindowPos

Other Resources

CWindow Members

CWindow::ScreenToClient

Converts screen coordinates to client coordinates.

```
BOOL ScreenToClient(
   LPPOINT lpPoint
) const throw();
BOOL ScreenToClient(
   LPRECT lpRect
) const throw();
```

Remarks

See ScreenToClient in the *Platform SDK*.

The second version of this method allows you to convert the coordinates of a RECT structure.

See Also

Reference

CWindow Class
CWindow::ClientToScreen
POINT

Other Resources

CWindow Members

CWindow::ScrollWindow

Scrolls the specified client area.

```
BOOL ScrollWindow(
   int xAmount,
   int yAmount,
   LPCRECT lpRect = NULL,
   LPCRECT lpClipRect = NULL
) throw();
```

Remarks

See ScrollWindow in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::ScrollWindowEx
RECT

Other Resources

CWindow Members

CWindow::ScrollWindowEx

Scrolls the specified client area with additional features.

```
int ScrollWindowEx(
   int dx,
   int dy,
   LPCRECT lpRectScroll,
   LPCRECT lpRectClip,
   HRGN hRgnUpdate,
   LPRECT lpRectUpdate,
   UINT flags
) throw();
```

Remarks

See ScrollWindowEx in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::ScrollWindow
RECT

Other Resources

CWindow Members

CWindow::SendDlgItemMessage

Sends a message to a control.

```
LRESULT SendDlgItemMessage(
   int nID,
   UINT message,
   WPARAM wParam = 0,
   LPARAM lParam = 0
) throw();
```

Remarks

See SendDlgItemMessage in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::SendMessage

Other Resources

CWindow Members

CWindow::SendMessage

Sends a message to the window and does not return until the window procedure has processed the message.

```
LRESULT SendMessage(
   UINT message,
   WPARAM wParam = 0,
   LPARAM lParam = 0
) throw();
static LRESULT SendMessage(
   HWND hWnd,
   UINT message,
   WPARAM wParam,
   LPARAM lParam
) throw();
```

Remarks

See SendMessage in the *Platform SDK*.

Example

```
// The following example attaches a HWND to the CWindow
// object and sends a WM_PAINT message to the window
// wrapped by CWindow object using CWindow::SendMessage.

CWindow myWindow;
myWindow.Attach(hWnd);
myWindow.SendMessage(WM_PAINT,0L,0L);
```

See Also

Reference

CWindow Class
CWindow::PostMessage
CWindow::SendNotifyMessage
CWindow::SendMessageToDescendants

Other Resources

CWindow Members

CWindow::SendMessageToDescendants

Sends the specified message to all immediate children of the **CWindow** object.

```
void SendMessageToDescendants(
   UINT message,
   WPARAM wParam = 0,
   LPARAM lParam = 0,
   BOOL bDeep = TRUE
) throw();
```

Parameters

message

[in] The message to be sent.

wParam

[in] Additional message-specific information.

lParam

[in] Additional message-specific information.

bDeep

[in] If **TRUE** (the default value), the message will be sent to all descendant windows; otherwise, it will be sent only to the immediate child windows.

Remarks

If *bDeep* is **TRUE**, the message is additionally sent to all other descendant windows.

See Also

Reference

CWindow Class
CWindow::SendMessage
CWindow::SendNotifyMessage
CWindow::PostMessage

Other Resources

CWindow Members

CWindow::SendNotifyMessage

Sends a message to the window.

```
BOOL SendNotifyMessage(
   UINT message,
   WPARAM wParam = 0,
   LPARAM lParam = 0
) throw();
```

Remarks

See SendNotifyMessage in the *Platform SDK*.

If the window was created by the calling thread, **SendNotifyMessage** does not return until the window procedure has processed the message. Otherwise, it returns immediately.

See Also

Reference

CWindow Class
CWindow::SendMessage
CWindow::SendMessageToDescendants
CWindow::PostMessage

Other Resources

CWindow Members

CWindow::SetActiveWindow

Activates the window.

```
HWND SetActiveWindow( ) throw();
```

Remarks

See SetActiveWindow in the *Platform SDK*.

Example

```
// The following example attaches a HWND to the CWindow object
// and sets the window as an active window by calling
// CWindow::SetActiveWindow which returns the HWND of the
// previously active window.

CWindow myWindow;
myWindow.Attach(hWnd);
HWND hWndPrev = myWindow.SetActiveWindow();
```

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::SetCapture

Sends all subsequent mouse input to the window.

```
HWND SetCapture( ) throw();
```

Remarks

See SetCapture in the *Platform SDK*.

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::SetClipboardViewer

Adds the window to the Clipboard viewer chain.

```
HWND SetClipboardViewer( ) throw();
```

Remarks

See SetClipboardViewer in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::ChangeClipboardChain

Other Resources

CWindow Members

CWindow::SetDlgCtrlID

Sets the identifier of the window to the specified value.

```
int SetDlgCtrlID(
   int nID
) throw();
```

Parameters

nID

[in] The new value to set for the window's identifier.

Return Value

If successful, the previous identifier of the window; otherwise 0.

See Also

Reference

CWindow Class

CWindow::GetDlgCtrlID

Other Resources

CWindow Members

CWindow::SetDlgItemInt

Changes a control's text to the string representation of an integer value.

```
BOOL SetDlgItemInt(
   int nID,
   UINT nValue,
   BOOL bSigned = TRUE
) throw();
```

Remarks

See SetDlgItemInt in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::GetDlgItemInt
CWindow::SetDlgItemText

Other Resources

CWindow Members

CWindow::SetDlgItemText

Changes a control's text.

```
BOOL SetDlgItemText(
   int nID,
   LPCTSTR lpszString
) throw();
```

Remarks

See SetDlgItemText in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::GetDlgItemText
CWindow::SetDlgItemInt

Other Resources

CWindow Members

CWindow::SetFocus

Sets the input focus to the window.

```
HWND SetFocus( ) throw();
```

Remarks

See SetFocus in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::SetFocus() to set the input focus

CWindow myWindow;
myWindow.Attach(hWnd);
HWND hWndLeftFocus = myWindow.SetFocus();
```

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::SetFont

Changes the window's current font by sending a WM_SETFONT message to the window.

```
void SetFont(
   HFONT hFont,
   BOOL bRedraw = TRUE
) throw();
```

Parameters

hFont

[in] The handle to the new font.

bRedraw

[in] If **TRUE** (the default value), the window is redrawn. Otherwise, it is not.

See Also

Reference

CWindow Class

CWindow::GetFont

Other Resources

CWindow Members

CWindow::SetHotKey

Associates a hot key with the window by sending a **WM_SETHOTKEY** message.

```
int SetHotKey(
   WORD wVirtualKeyCode,
   WORD wModifiers
) throw();
```

Parameters

wVirtualKeyCode

[in] The virtual key code of the hot key. For a list of of standard virtual key codes, see Winuser.h.

wModifiers

[in] The modifiers of the hot key. For a list of possible values, see **WM_SETHOTKEY** in the *Platform SDK*.

Return Value

For a list of possible return values, see WM_SETHOTKEY in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::GetHotKey

Other Resources

CWindow Members

CWindow::SetIcon

Sets the window's large or small icon to the icon identified by *hIcon*.

```
HICON SetIcon(
   HICON hIcon,
   BOOL bBigIcon = TRUE
) throw();
```

Parameters

hIcon

[in] The handle to a new icon.

bBigIcon

[in] If **TRUE** (the default value), the method sets a large icon. Otherwise, it sets a small icon.

Return Value

The handle to the previous icon.

Remarks

SetIcon sends a WM_SETICON message to the window.

See Also

Reference

CWindow Class

CWindow::GetIcon

Other Resources

CWindow Members

CWindow::SetMenu

Changes the window's current menu.

```
BOOL SetMenu(
   HMENU hMenu
) throw();
```

Remarks

See SetMenu in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::GetMenu

Other Resources

CWindow Members

CWindow::SetParent

Changes the parent window.

```
HWND SetParent(
   HWND hWndNewParent
) throw();
```

Remarks

See SetParent in the *Platform SDK*.

Example

```
// The following example attaches a HWND to the CWindow object
// and sets the hWndParent as the parent window of the
// window wrapped by CWindow object using CWindow::SetParent.

CWindow myWindow;
myWindow.Attach(hWndChild);
HWND hWndPrevParent = myWindow.SetParent(hWndParent);
```

See Also

Reference

CWindow Class

CWindow::GetParent

Other Resources

CWindow Members

CWindow::SetRedraw

Sets or clears the redraw flag by sending a WM_SETREDRAW message to the window.

```
void SetRedraw(
   BOOL bRedraw = TRUE
) throw();
```

Parameters

bRedraw

[in] Specifies the state of the redraw flag. If **TRUE** (the default value), the redraw flag is set; if **FALSE**, the flag is cleared.

Remarks

Call **SetRedraw** to allow changes to be redrawn or to prevent changes from being redrawn.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::SetRedraw() to set and reset the redraw flag

CWindow myWindow;
myWindow.Attach(hWnd);
myWindow.SetRedraw();      //sets the redraw flag to TRUE
...
...
myWindow.SetRedraw(FALSE); //sets the redraw flag to FALSE
```

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::SetScrollInfo

Sets the parameters of a scroll bar.

```
int SetScrollInfo(
   int nBar,
   LPSCROLLINFO lpScrollInfo,
   BOOL bRedraw = TRUE
) throw();
```

Remarks

See SetScrollInfo in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::GetScrollInfo

Other Resources

CWindow Members

CWindow::SetScrollPos

Changes the position of the scroll box.

```
int SetScrollPos(
   int nBar,
   int nPos,
   BOOL bRedraw = TRUE
) throw();
```

Remarks

See SetScrollPos in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::GetScrollPos

Other Resources

CWindow Members

CWindow::SetScrollRange

Changes the scroll bar range.

```
BOOL SetScrollRange(
   int nBar,
   int nMinPos,
   int nMaxPos,
   BOOL bRedraw = TRUE
) throw();
```

Remarks

See SetScrollRange in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::GetScrollRange

Other Resources

CWindow Members

CWindow::SetTimer

Creates a timer event.

```
UINT SetTimer(
   UINT nIDEvent,
   UINT nElapse,
   void ( CALLBACK* lpfnTimer )(HWND, UINT, UINT, DWORD) = NULL
) throw();
```

Remarks

See SetTimer in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::KillTimer

Other Resources

CWindow Members

CWindow::SetWindowContextHelpId

Sets the window's help context identifier.

```
BOOL SetWindowContextHelpId(
   DWORD dwContextHelpId
) throw();
```

Remarks

See SetWindowContextHelpId in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::GetWindowContextHelpId

Other Resources

CWindow Members

CWindow::SetWindowLong

Sets a 32-bit value at a specified offset into the extra window memory.

```
LONG SetWindowLong(
   int nIndex,
   LONG dwNewLong
) throw();
```

Remarks

See SetWindowLong in the *Platform SDK*.

Note
To write code that is compatible with both 32-bit and 64-bit versions of Windows, use CWindow::SetWindowLongPtr.

See Also

Reference

CWindow Class
CWindow::GetWindowLong
CWindow::SetWindowWord

Other Resources

CWindow Members

CWindow::SetWindowLongPtr

Changes an attribute of the specified window, and also sets a value at the specified offset in the extra window memory.

```
LONG_PTR SetWindowLongPtr(
   int nIndex,
   LONG_PTR dwNewLong
) throw( );
```

Remarks

See SetWindowLongPtr in the *Platform SDK*.

This function supersedes the **CWindow::SetWindowLong** method. To write code that is compatible with both 32-bit and 64-bit versions of Windows, use **CWindow::SetWindowLongPtr**.

See Also

Reference

CWindow Class
CWindow::GetWindowLongPtr
CWindow::SetWindowLong

Other Resources

CWindow Members

CWindow::SetWindowPlacement

Sets the show state and positions.

```
BOOL SetWindowPlacement(
   const WINDOWPLACEMENT FAR*lpwndpl
);
```

Remarks

See SetWindowPlacement in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::GetWindowPlacement
WINDOWPLACEMENT

Other Resources

CWindow Members

CWindow::SetWindowPos

Sets the size, position, and Z order.

```
BOOL SetWindowPos(
   HWND hWndInsertAfter,
   int x,
   int y,
   int cx,
   int cy,
   UINT nFlags
) throw();
BOOL SetWindowPos(
   HWND hWndInsertAfter,
   LPCRECT lpRect,
   UINT nFlags
) throw();
```

Remarks

See SetWindowPos in the *Platform SDK*.

The second version of this method uses a RECT structure to set the window's new position, width, and height.

See Also

Reference

CWindow Class
CWindow::BringWindowToTop
CWindow::MoveWindow
RECT

Other Resources

CWindow Members

CWindow::SetWindowRgn

Sets the window region of a window.

```
int SetWindowRgn(
   HRGN hRgn,
   BOOL bRedraw = FALSE
) throw();
```

Remarks

See SetWindowRgn in the *Platform SDK*.

See Also

Reference

CWindow Class

CWindow::GetWindowRgn

Other Resources

CWindow Members

CWindow::SetWindowText

Changes the window's text.

```
BOOL SetWindowText(
   LPCTSTR lpszString
) throw();
```

Remarks

See SetWindowText in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::SetWindowText() to set the new title-text of the
//window

CWindow myWindow;
myWindow.Attach(hWnd);
myWindow.SetWindowText(_T("Hello ATL"));
```

See Also

Reference

CWindow Class

CWindow::GetWindowText

Other Resources

CWindow Members

CWindow::SetWindowWord

Sets a 16-bit value at a specified offset into the extra window memory.

```
WORD SetWindowWord(
   int nIndex,
   WORD wNewWord
) throw();
```

Remarks

See SetWindowLong in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::GetWindowWord
CWindow::SetWindowLong

Other Resources

CWindow Members

CWindow::ShowCaret

Displays the system caret.

```
BOOL ShowCaret( ) throw();
```

Remarks

See ShowCaret in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::ShowCaret() to show the caret

CWindow myWindow;
myWindow.Attach(hWnd);
myWindow.ShowCaret();
```

See Also

Reference

CWindow Class

CWindow::HideCaret

Other Resources

CWindow Members

CWindow::ShowOwnedPopups

Shows or hides the pop-up windows owned by the window.

```
BOOL ShowOwnedPopups(
   BOOL bShow = TRUE
) throw();
```

Remarks

See ShowOwnedPopups in the *Platform SDK*.

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::ShowScrollBar

Shows or hides a scroll bar.

```
BOOL ShowScrollBar(
   UINT nBar,
   BOOL bShow = TRUE
) throw();
```

Remarks

See ShowScrollBar in the *Platform SDK*.

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::ShowWindow

Sets the window's show state.

```
BOOL ShowWindow(
   int nCmdShow
) throw();
```

Remarks

See ShowWindow in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::ShowWindow() to show the window in its maximized state

CWindow myWindow;
myWindow.Attach(hWnd);
myWindow.ShowWindow(SW_SHOWMAXIMIZED);
```

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::ShowWindowAsync

Sets the show state of a window created by a different thread.

```
BOOL ShowWindowAsync(
   int nCmdShow
) throw();
```

Remarks

See ShowWindowAsync in the *Platform SDK*.

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::UpdateWindow

Updates the client area.

```
BOOL UpdateWindow( ) throw();
```

Remarks

See UpdateWindow in the *Platform SDK*.

Example

```
//The following example attaches an HWND to the CWindow object and
//calls CWindow::UpdateWindow() to update the window

CWindow myWindow;
myWindow.Attach(hWnd);
BOOL bUpdated = myWindow.UpdateWindow();
```

See Also

Reference

CWindow Class

CWindow::RedrawWindow

Other Resources

CWindow Members

CWindow::ValidateRect

Validates the client area within the specified rectangle.

```
BOOL ValidateRect(
   LPCRECT lpRect
) throw();
```

Remarks

See ValidateRect in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::ValidateRgn
CWindow::InvalidateRect

Other Resources

CWindow Members

CWindow::ValidateRgn

Validates the client area within the specified region.

```
BOOL ValidateRgn(
   HRGN hRgn
) throw();
```

Remarks

See ValidateRgn in the *Platform SDK*.

See Also

Reference

CWindow Class
CWindow::ValidateRect
CWindow::InvalidateRgn

Other Resources

CWindow Members

CWindow::WinHelp

Starts Windows Help.

```
BOOL WinHelp(
   LPCTSTR lpszHelp,
   UINT nCmd = HELP_CONTEXT,
   DWORD dwData = 0
) throw();
```

Remarks

See WinHelp in the *Platform SDK*.

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow Operators

For information about the operators in **CWindow**, see CWindow Members.

CWindow::operator HWND

Converts a **CWindow** object to an **HWND**.

```
operator HWND( ) const throw();
```

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow::operator =

Assigns an **HWND** to the **CWindow** object by setting the m_hWnd member to *hWnd*.

```
CWindow& operator =(
   HWND hWnd
) throw();
```

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindow Data Members

For information about the data members in **CWindow**, see CWindow Members.

CWindow::m_hWnd

Contains a handle to the window associated with the **CWindow** object.

```
HWND m_hWnd throw() throw();
```

See Also

Reference

CWindow Class

CWindow::CWindow

Other Resources

CWindow Members

CWindow::rcDefault

Contains default window dimensions.

```
static RECT rcDefault;
```

See Also

Reference

CWindow Class

Other Resources

CWindow Members

CWindowImpl Class

This class provides methods for creating or subclassing a window.

```
template <
   class T,
   class TBase = CWindow,
   class TWinTraits = CControlWinTraits
>
class ATL_NO_VTABLE CWindowImpl :
   public CWindowImplBaseT< TBase, TWinTraits >
```

Parameters

T

Your class, derived from **CWindowImpl**.

TBase

The base class of your new class. The default base class is CWindow.

TWinTraits

A traits class that defines styles for your window. The default is **CControlWinTraits**.

Remarks

CWindowImpl allows you to create a new window or subclass an existing window. **CWindowImpl**'s window procedure uses a message map to direct messages to the appropriate handlers.

CWindowImpl::Create creates a new window based on the window class information managed by CWndClassInfo. **CWindowImpl** contains the DECLARE_WND_CLASS macro, which means **CWndClassInfo** will register a new window class. If you want to superclass an existing window class, derive your class from **CWindowImpl** and include the DECLARE_WND_SUPERCLASS macro. In this case, **CWndClassInfo** will register a window class that is based on an existing class but uses **CWindowImpl::WindowProc**. For example:

```
class CMyWindow : CComControl<CMyWindow>, ...
                // CComControl derives from CWindowImpl
{
public:
   // 1. The NULL parameter means ATL will generate a
   //    name for the superclass
   // 2. The "EDIT" parameter means the superclass is
   //    based on the standard Windows Edit box
   DECLARE_WND_SUPERCLASS(NULL, "EDIT")
   ...
};
```

Note

Because **CWndClassInfo** manages the information for a single window class, each window created through an instance of **CWindowImpl** will be based on the same window class.

CWindowImpl also supports window subclassing. The **SubclassWindow** method attaches an existing window to the **CWindowImpl** object and changes the window procedure to **CWindowImpl::WindowProc**. Each instance of **CWindowImpl** can subclass a different window.

Note

For any given **CWindowImpl** object, call either **Create** or **SubclassWindow**. You should not invoke both methods on the same object.

In addition to **CWindowImpl**, ATL provides CContainedWindow to create a window contained within another object.

The base class destructor (~**CWindowImplRoot**) ensures that the window is gone before the object is destroyed.

CWindowImpl derives from **CWindowImplBaseT**, which derives from **CWindowImplRoot**, which in turn derives from **TBase** and CMessageMap.

For more information about	See
Creating controls	ATL Tutorial
Using windows in ATL	ATL Window Classes
ATL Project Wizard	Creating an ATL Project
Windows	About Windows Procedures

Requirements

Header: atlwin.h

See Also

Reference

BEGIN_MSG_MAP

CComControl Class

Other Resources

CWindowImpl Members

ATL Class Overview

CWindowImpl Members

Methods

Create	Creates a window.

CWindowImplBaseT Methods

DefWindowProc	Provides default message processing.
GetCurrentMessage	Returns the current message.
GetWindowProc	Returns the current window procedure.
OnFinalMessage	Called after receiving the last message (typically **WM_NCDESTROY**).
SubclassWindow	Subclasses a window.
UnsubclassWindow	Restores a previously subclassed window.

Static Functions

GetWndClassInfo	Returns a static instance of CWndClassInfo, which manages the window class information.
WindowProc	Processes messages sent to the window.

Data Members

m_pfnSuperWindowProc	Points to the window class's original window procedure.

See Also

Reference

CWindowImpl Class

CWindowImpl Methods

For information about the methods in **CWindowImpl**, see CWindowImpl Members.

CWindowImpl::Create

Creates a window based on a new window class.

```
HWND Create(
   HWND hWndParent,
   _U_RECT rect = NULL,
   LPCTSTR szWindowName = NULL,
   DWORD dwStyle = 0,
   DWORD dwExStyle = 0,
   _U_MENUorID MenuOrID = 0U,
   LPVOID lpCreateParam = NULL
);
```

Parameters

hWndParent

[in] The handle to the parent or owner window.

rect

[in] A RECT structure specifying the position of the window. The **RECT** can be passed by pointer or by reference.

szWindowName

[in] Specifies the name of the window. The default value is **NULL**.

dwStyle

[in] The style of the window. This value is combined with the style provided by the traits class for the window. The default value gives the traits class full control over the style. For a list of possible values, see CreateWindow in the *Platform SDK*.

dwExStyle

[in] The extended window style. This value is combined with the style provided by the traits class for the window. The default value gives the traits class full control over the style. For a list of possible values, see CreateWindowEx in the *Platform SDK*.

MenuOrID

[in] For a child window, the window identifier. For a top-level window, a menu handle for the window. The default value is **0U**.

lpCreateParam

[in] A pointer to window-creation data. For a full description, see the description for the final parameter to CreateWindowEx.

Return Value

If successful, the handle to the newly created window. Otherwise, **NULL**.

Remarks

Create first registers the window class if it has not yet been registered. The newly created window is automatically attached to the **CWindowImpl** object.

Note
Do not call **Create** if you have already called SubclassWindow.

To use a window class that is based on an existing window class, derive your class from **CWindowImpl** and include the DECLARE_WND_SUPERCLASS macro. The existing window class's window procedure is saved in m_pfnSuperWindowProc. For more information, see the CWindowImpl overview.

Note
If 0 is used as the value for the *MenuOrID* parameter, it must be specified as 0U (the default value) to avoid a compiler error.

See Also

Reference

CWindowImpl Class

CWindowImpl::GetWndClassInfo

CWndClassInfo::Register

CWindow::m_hWnd

Other Resources

CWindowImpl Members

CWindowImpl::DefWindowProc

Called by WindowProc to process messages not handled by the message map.

```
LRESULT DefWindowProc(
   UINT uMsg,
   WPARAM wParam,
   LPARAM lParam
);
LRESULT DefWindowProc( );
```

Parameters

uMsg

[in] The message sent to the window.

wParam

[in] Additional message-specific information.

lParam

[in] Additional message-specific information.

Return Value

The result of the message processing.

Remarks

By default, **DefWindowProc** calls the CallWindowProc Win32 function to send the message information to the window procedure specified in m_pfnSuperWindowProc.

The function with no parameters automatically retrieves the needed parameters from the current message.

See Also

Reference

CWindowImpl Class

Other Resources

CWindowImpl Members

CWindowImpl::GetCurrentMessage

Returns the current message, packaged in the **MSG** structure.

```
const MSG* GetCurrentMessage( );
```

Return Value

The current message.

See Also

Reference

CWindowImpl Class
CWindowImpl::OnFinalMessage
CWindowImpl::UnsubclassWindow
CWindowImpl::WindowProc

Other Resources

CWindowImpl Members

CWindowImpl::GetWindowProc

Returns **WindowProc**, the current window procedure.

```
virtual WNDPROC GetWindowProc( );
```

Return Value

The current window procedure.

Remarks

Override this method to replace the window procedure with your own.

See Also

Reference

CWindowImpl Class
CWindowImpl::OnFinalMessage
CWindowImpl::UnsubclassWindow
CWindowImpl::WindowProc

Other Resources

CWindowImpl Members

CWindowImpl::OnFinalMessage

Called after receiving the last message (typically **WM_NCDESTROY**).

```
virtual void OnFinalMessage(
   HWND hWnd
);
```

Parameters

hWnd

[in] A handle to the window being destroyed.

Remarks

The default implementation of **OnFinalMessage** does nothing, but you can override this function to handle cleanup before destroying a window. If you want to automatically delete your object upon the window destruction, you can call `delete this;` in this function.

See Also

Reference

CWindowImpl Class
CWindowImpl::GetWindowProc
CWindowImpl::UnsubclassWindow

Other Resources

CWindowImpl Members

CWindowImpl::SubclassWindow

Subclasses the window identified by *hWnd* and attaches it to the **CWindowImpl** object.

```
BOOL SubclassWindow(
   HWND hWnd
);
```

Parameters

hWnd

[in] The handle to the window being subclassed.

Return Value

TRUE if the window is successfully subclassed; otherwise, **FALSE**.

Remarks

The subclassed window now uses CWindowImpl::WindowProc. The original window procedure is saved in m_pfnSuperWindowProc.

Note
Do not call **SubclassWindow** if you have already called Create.

See Also

Reference

CWindowImpl Class

CWindowImpl::UnsubclassWindow

Other Resources

CWindowImpl Members

CWindowImpl::UnsubclassWindow

Detaches the subclassed window from the **CWindowImpl** object and restores the original window procedure, saved in m_pfnSuperWindowProc.

```
HWND UnsubclassWindow( );
```

Return Value

The handle to the window previously subclassed.

See Also

Reference

CWindowImpl Class

CWindowImpl::SubclassWindow

Other Resources

CWindowImpl Members

CWindowImpl Data Members

For information about the data members in **CWindowImpl**, see CWindowImpl Members.

CWindowImpl::m_pfnSuperWindowProc

Depending on the window, points to one of the following window procedures.

```
WNDPROC m_pfnSuperWindowProc;
```

Remarks

Type of window	Window procedure
A window based on a new window class, specified through the DECLARE_WND_CLASS macro.	The DefWindowProc Win32 function.
A window based on a window class that modifies an existing class, specified through the DECLARE_WND_SUPERCLASS macro.	The existing window class's window procedure.
A subclassed window.	The subclassed window's original window procedure.

CWindowImpl::DefWindowProc sends message information to the window procedure saved in **m_pfnSuperWindowProc**.

See Also

Reference

CWindowImpl Class
CWindowImpl::Create
CWindowImpl::SubclassWindow

Other Resources

CWindowImpl Members

CWindowImpl Static Functions

For information about the static functions in **CWindowImpl**, see CWindowImpl Members.

CWindowImpl::GetWndClassInfo

Called by Create to access the window class information.

```
static CWndClassInfo& GetWndClassInfo( );
```

Return Value

A static instance of CWndClassInfo.

Remarks

By default, **CWindowImpl** obtains this method through the DECLARE_WND_CLASS macro, which specifies a new window class.

To superclass an existing window class, derive your class from **CWindowImpl** and include the DECLARE_WND_SUPERCLASS macro to override **GetWndClassInfo**. For more information, see the CWindowImpl overview.

Besides using the **DECLARE_WND_CLASS** and **DECLARE_WND_SUPERCLASS** macros, you can override **GetWndClassInfo** with your own implementation.

See Also

Reference

CWindowImpl Class

Other Resources

CWindowImpl Members

CWindowImpl::WindowProc

This static function implements the window procedure.

```
static LRESULT CALLBACK WindowProc(
   HWND hWnd,
   UINT uMsg,
   WPARAM wParam,
   LPARAM lParam
);
```

Parameters

hWnd

[in] The handle to the window.

uMsg

[in] The message sent to the window.

wParam

[in] Additional message-specific information.

lParam

[in] Additional message-specific information.

Return Value

The result of the message processing.

Remarks

WindowProc uses the default message map (declared with BEGIN_MSG_MAP) to direct messages to the appropriate handlers. If necessary, **WindowProc** calls DefWindowProc for additional message processing. If the final message is not handled, **WindowProc** does the following:

- Performs unsubclassing if the window was unsubclassed.
- Clears **m_hWnd**.
- Calls OnFinalMessage before the window is destroyed.

You can override **WindowProc** to provide a different mechanism for handling messages.

See Also

Reference

CWindowImpl Class

Other Resources

CWindowImpl Members

CWinTraits Class

This class provides a method for standardizing the styles used when creating a window object.

```
template <
   DWORD t_dwStyle = 0,
   DWORD t_dwExStyle = 0
>
class CWinTraits
```

Parameters

t_dwStyle

Default standard window styles.

t_dwExStyle

Default extended window styles.

Remarks

This window traits class provides a simple method for standardizing the styles used for the creation of an ATL window object. Use a specialization of this class as a template parameter to CWindowImpl or another of ATL's window classes to specify the default standard and extended styles used for instances of that window class.

Use this template when you want to provide default window styles that will be used only when no other styles are specified in the call to CWindowImpl::Create.

ATL provides three predefined specializations of this template for commonly used combinations of window styles:

CControlWinTraits

Designed for a standard control window. The following standard styles are used: **WS_CHILD**, **WS_VISIBLE**, **WS_CLIPCHILDREN**, and **WS_CLIPSIBLINGS**. There are no extended styles.

CFrameWinTraits

Designed for a standard frame window. The standard styles used include: **WS_OVERLAPPEDWINDOW**, **WS_CLIPCHILDREN**, and **WS_CLIPSIBLINGS**. The extended styles used include: **WS_EX_APPWINDOW** and **WS_EX_WINDOWEDGE**.

CMDIChildWinTraits

Designed for a standard MDI child window. The standard styles used include: **WS_OVERLAPPEDWINDOW**, **WS_CHILD**, **WS_VISIBLE**, **WS_CLIPCHILDREN**, and **WS_CLIPSIBLINGS**. The extended styles used include: **WS_EX_MDICHILD**.

If you want to ensure that certain styles are set for all instances of the window class while permitting other styles to be set on a per-instance basis, use CWinTraitsOR instead.

Requirements

Header: atlwin.h

See Also

Concepts

Understanding Window Traits

Other Resources

CWinTraits Members

ATL Class Overview

CWinTraits Members

Static Functions

GetWndExStyle	Retrieves the extended styles for the **CWinTraits** object.
GetWndStyle	Retrieves the standard styles for the **CWinTraits** object.

See Also

Reference

CWinTraits Class

CWinTraits Static Functions

For information about the static functions in **CWinTraits**, see CWinTraits Members.

CWinTraits::GetWndStyle

Call this function to retrieve the standard styles of the **CWinTraits** object.

```
static DWORD GetWndStyle(
   DWORD dwStyle
);
```

Parameters

dwStyle

Standard styles used for creation of a window. If *dwStyle* is 0, the template style values (*t_dwStyle*) are returned. If *dwStyle* is nonzero, *dwStyle* is returned.

Return Value

The standard window styles of the object.

See Also

Reference

CWinTraits Class

CWinTraits::GetWndExStyle

Other Resources

CWinTraits Members

CWinTraits::GetWndExStyle

Call this function to retrieve the extended styles of the **CWinTraits** object.

```
static DWORD GetWndExStyle(
   DWORD dwExStyle
);
```

Parameters

dwExStyle

Extended styles used for creation of a window. If *dwExStyle* is 0, the template style values (*t_dwExStyle*) are returned. If *dwExStyle* is nonzero, *dwExStyle* is returned.

Return Value

The extended window styles of the object.

See Also

Reference

CWinTraits Class

CWinTraits::GetWndStyle

Other Resources

CWinTraits Members

CWinTraitsOR Class

This class provides a method for standardizing the styles used when creating a window object.

```
template <
   DWORD t_dwStyle = 0,
   DWORD t_dwExStyle = 0,
   class TWinTraits = CControlWinTraits
>
class CWinTraitsOR
```

Parameters

t_dwStyle

Default window styles.

t_dwExStyle

Default extended window styles.

Remarks

This window traits class provides a simple method for standardizing the styles used for the creation of an ATL window object. Use a specialization of this class as a template parameter to CWindowImpl or another of ATL's window classes to specify the minimum set of standard and extended styles to be used for instances of that window class.

Use a specialization of this template if you want to ensure that certain styles are set for all instances of the window class while permitting other styles to be set on a per-instance basis in the call to CWindowImpl::Create.

If you want to provide default window styles that will be used only when no other styles are specified in the call to **CWindowImpl::Create**, use CWinTraits instead.

Requirements

Header: atlwin.h

See Also

Concepts

Understanding Window Traits

Other Resources

CWinTraitsOR Members

ATL Class Overview

CWinTraitsOR Members

Static Functions

GetWndExStyle	Retrieves the extended styles for the **CWinTraitsOR** object.
GetWndStyle	Retrieves the standard styles for the **CWinTraitsOR** object.

See Also

Reference

CWinTraitsOR Class

CWinTraitsOR Static Functions

For information about the static functions in **CWinTraitsOR**, see CWinTraitsOR Members.

CWinTraitsOR::GetWndStyle

Call this function to retrieve a combination (using the logical OR operator) of the standard styles of the **CWinTraits** object and the default styles specified by *t_dwStyle*.

```
static DWORD GetWndStyle(
   DWORD dwStyle
);
```

Parameters

dwStyle

Styles used for creation of a window.

Return Value

A combination of styles that are passed in *dwStyle* and the default ones specified by *t_dwStyle*, using the logical OR operator.

See Also

Reference

CWinTraitsOR Class

CWinTraitsOR::GetWndExStyle

Other Resources

CWinTraitsOR Members

CWinTraitsOR::GetWndExStyle

Call this function to retrieve a combination (using the logical OR operator) of the extended styles of the **CWinTraits** object and the default styles specified by *t_dwStyle*.

```
static DWORD GetWndExStyle(
   DWORD dwExStyle
);
```

Parameters

dwExStyle

Extended styles used for creation of a window.

Return Value

A combination of extended styles that are passed in *dwExStyle* and default ones specified by *t_dwExStyle*, using the logical OR operator

See Also

Reference

CWinTraitsOR Class

CWinTraitsOR::GetWndStyle

Other Resources

CWinTraitsOR Members

CWndClassInfo Class

This class provides methods for registering information for a window class.

```
class CWndClassInfo
```

Remarks

CWndClassInfo manages the information of a window class. You typically use **CWndClassInfo** through one of three macros, **DECLARE_WND_CLASS**, **DECLARE_WND_CLASS_EX**, or **DECLARE_WND_SUPERCLASS**, as described in the following table:

Macro	Description
DECLARE_WND_CLASS	**CWndClassInfo** registers information for a new window class.
DECLARE_WND_CLASS_EX	**CWndClassInfo** registers information for a new window class, including the class parameters.
DECLARE_WND_SUPERCLASS	**CWndClassInfo** registers information for a window class that is based on an existing class but uses a different window procedure. This technique is called superclassing.

By default, CWindowImpl includes the **DECLARE_WND_CLASS** macro to create a window based on a new window class. DECLARE_WND_CLASS provides default styles and background color for the control. If you want to specify the style and background color yourself, derive your class from **CWindowImpl** and include the **DECLARE_WND_CLASS_EX** macro in your class definition.

If you want to create a window based on an existing window class, derive your class from **CWindowImpl** and include the **DECLARE_WND_SUPERCLASS** macro in your class definition. For example:

```
class CMyWindow : CComControl<CMyWindow>, ...
                // CComControl derives from CWindowImpl
{
public:
   // 1. The NULL parameter means ATL will generate a
   //    name for the superclass
   // 2. The "EDIT" parameter means the superclass is
   //    based on the standard Windows Edit box
   DECLARE_WND_SUPERCLASS(NULL, "EDIT")
   ...
};
```

For more information about window classes, see Window Classes in the *Platform SDK*.

For more information about using windows in ATL, see the article ATL Window Classes.

Requirements

Header: atlwin.h

See Also

Reference

CComControl Class

Other Resources

CWndClassInfo Members

ATL Class Overview

CWndClassInfo Members

Methods

Register	Registers the window class.

Data Members

m_atom	Uniquely identifies the registered window class.
m_bSystemCursor	Specifies whether the cursor resource refers to a system cursor or to a cursor contained in a module resource.
m_lpszCursorID	Specifies the name of the cursor resource.
m_lpszOrigName	Contains the name of an existing window class.
m_szAutoName	Holds an ATL-generated name of the window class.
m_wc	Maintains window class information in a **WNDCLASSEX** structure.
pWndProc	Points to the window procedure of an existing window class.

See Also

Reference

CWndClassInfo Class

CWndClassInfo Methods

For information about the methods in **CWndClassInfo**, see CWndClassInfo Members.

CWndClassInfo::Register

Called by CWindowImpl::Create to register the window class if it has not yet been registered.

```
ATOM Register(
   WNDPROC* pProc
);
```

Parameters

pProc

[out] Specifies the original window procedure of an existing window class.

Return Value

If successful, an atom that uniquely identifies the window class being registered. Otherwise, 0.

Remarks

If you have specified the DECLARE_WND_CLASS (the default in CWindowImpl) or the DECLARE_WND_CLASS_EX macro, **Register** registers a new window class. In this case, the *pProc* parameter is not used.

If you have specified the DECLARE_WND_SUPERCLASS macro, **Register** registers a superclass — a window class that is based on an existing class but uses a different window procedure. The existing window class's window procedure is returned in *pProc*.

See Also

Reference

CWndClassInfo Class
CWndClassInfo::m_atom
CWndClassInfo::m_wc
CWndClassInfo::pWndProc

Other Resources

CWndClassInfo Members

CWndClassInfo Data Members

For information about the data members in **CWndClassInfo**, see CWndClassInfo Members.

CWndClassInfo::m_atom

Contains the unique identifier for the registered window class.

```
ATOM m_atom;
```

See Also

Reference

CWndClassInfo Class

CWndClassInfo::Register

Other Resources

CWndClassInfo Members

CWndClassInfo::m_bSystemCursor

If **TRUE**, the system cursor resource will be loaded when the window class is registered.

```
BOOL m_bSystemCursor;
```

Remarks

Otherwise, the cursor resource contained in your module will be loaded.

CWndClassInfo uses **m_bSystemCursor** only when the DECLARE_WND_CLASS (the default in CWindowImpl) or the DECLARE_WND_CLASS_EX macro is specified. In this case, **m_bSystemCursor** is initialized to **TRUE**. For more information, see the CWndClassInfo overview.

See Also

Reference

CWndClassInfo Class

CWndClassInfo::m_lpszCursorID

Other Resources

CWndClassInfo Members

CWndClassInfo::m_lpszCursorID

Specifies either the name of the cursor resource or the resource identifier in the low-order word and zero in the high-order word.

```
LPCTSTR m_lpszCursorID;
```

Remarks

When the window class is registered, the handle to the cursor identified by **m_lpszCursorID** is retrieved and stored by m_wc.

CWndClassInfo uses **m_lpszCursorID** only when the DECLARE_WND_CLASS (the default in CWindowImpl) or the DECLARE_WND_CLASS_EX macro is specified. In this case, **m_lpszCursorID** is initialized to **IDC_ARROW**. For more information, see the CWndClassInfo overview.

See Also

Reference

CWndClassInfo Class

CWndClassInfo::m_bSystemCursor

Other Resources

CWndClassInfo Members

CWndClassInfo::m_lpszOrigName

Contains the name of an existing window class.

```
LPCTSTR m_lpszOrigName;
```

Remarks

CWndClassInfo uses **m_lpszOrigName** only when you include the DECLARE_WND_SUPERCLASS macro in your class definition. In this case, **CWndClassInfo** registers a window class based on the class named by **m_lpszOrigName**. For more information, see the CWndClassInfo overview.

See Also

Reference

CWndClassInfo Class
CWndClassInfo::m_wc
CWndClassInfo::pWndProc

Other Resources

CWndClassInfo Members

CWndClassInfo::m_szAutoName

Holds the name of the window class.

```
TCHAR m_szAutoName[13];
```

Remarks

CWndClassInfo uses **m_szAutoName** only if **NULL** is passed for the *WndClassName* parameter to DECLARE_WND_CLASS, the DECLARE_WND_CLASS_EX or DECLARE_WND_SUPERCLASS. ATL will construct a name when the window class is registered.

See Also

Reference

CWndClassInfo Class

Other Resources

CWndClassInfo Members

CWndClassInfo::m_wc

Maintains the window class information in a WNDCLASSEX structure.

```
WNDCLASSEX m_wc;
```

Remarks

If you have specified the DECLARE_WND_CLASS (the default in CWindowImpl) or the DECLARE_WND_CLASS_EX macro, **m_wc** contains information about a new window class.

If you have specified the DECLARE_WND_SUPERCLASS macro, **m_wc** contains information about a superclass — a window class that is based on an existing class but uses a different window procedure. m_lpszOrigName and pWndProc save the existing window class's name and window procedure, respectively.

See Also

Reference

CWndClassInfo Class

Other Resources

CWndClassInfo Members

CWndClassInfo::pWndProc

Points to the window procedure of an existing window class.

```
WNDPROC pWndProc;
```

Remarks

CWndClassInfo uses **pWndProc** only when you include the DECLARE_WND_SUPERCLASS macro in your class definition. In this case, **CWndClassInfo** registers a window class that is based on an existing class but uses a different window procedure. The existing window class's window procedure is saved in **pWndProc**. For more information, see the CWndClassInfo overview.

See Also

Reference

CWndClassInfo Class
CWndClassInfo::m_wc
CWndClassInfo::m_lpszOrigName

Other Resources

CWndClassInfo Members

IAtlAutoThreadModule Class

This class represents an interface to a **CreateInstance** method.

```
__interface IAtlAutoThreadModule
```

Remarks

The class CAtlAutoThreadModuleT derives from **IAtlAutoThreadModule**, using it to provide code for creating an object and retrieving an interface pointer.

Requirements

Header: atlbase.h

See Also

Other Resources

ATL Class Overview

IAtlMemMgr Class

This class represents the interface to a memory manager.

```
__interface __declspec( uuid( "654F7EF5-CFDF-4df9-A450-6C6A13C622C0" )) IAtlMemMgr
```

Remarks

This interface is implemented by CComHeap, CCRTHeap, CLocalHeap, CGlobalHeap, or CWin32Heap.

Note

The local and global heap functions are slower than other memory management functions, and do not provide as many features. Therefore, new applications should use the heap functions. These are available in the CWin32Heap class.

Example

```
// Demonstrate IAtlMemMgr using the five possible
// memory function implementation classes.

#include "stdafx.h"
#include "atlmem.h"

HRESULT MemoryManagerDemonstration(IAtlMemMgr& MemoryManager) throw()
{
   // The IAtlMemMgr interface guarantees not to throw exceptions
   // so we can make the same guarantee for this function
   // without adding exception handling code.

   // A variable which will point to some allocated memory.
   void* pMemory = NULL;

   const size_t BytesInChunk = 1024;

   // Allocate a chunk of memory
   pMemory = MemoryManager.Allocate(BytesInChunk);

   // Confirm the validity of the allocated memory
   if (pMemory == NULL)
      return E_OUTOFMEMORY;

   // Confirm the size of the allocated memory
   ATLASSERT(MemoryManager.GetSize(pMemory) == BytesInChunk);

   // Increase the size of the allocated memory
   pMemory = MemoryManager.Reallocate(pMemory, BytesInChunk * 2);

   // Confirm the validity of the allocated memory
   if (pMemory == NULL)
      return E_OUTOFMEMORY;

   // Confirm the size of the reallocated  memory
   ATLASSERT(MemoryManager.GetSize(pMemory) == BytesInChunk * 2);

   // Free the allocated memory
   MemoryManager.Free(pMemory);

   return S_OK;
}

int main()
{
   CComHeap heapCom;
   CCRTHeap heapCrt;
```

```
    CLocalHeap heapLocal;
    CGlobalHeap heapGlobal;
    // It is necessary to provide extra information
    // to the constructor when using CWin32Heap
    CWin32Heap heapWin32(NULL, 4096);

    ATLASSERT(S_OK==MemoryManagerDemonstration(heapCom));
    ATLASSERT(S_OK==MemoryManagerDemonstration(heapCrt));
    ATLASSERT(S_OK==MemoryManagerDemonstration(heapLocal));
    ATLASSERT(S_OK==MemoryManagerDemonstration(heapGlobal));
    ATLASSERT(S_OK==MemoryManagerDemonstration(heapWin32));

    return 0;
}
```

For another example, see the SRFSyntax Sample.

Requirements

Header: atlmem.h

See Also

Tasks

SRFSyntax Sample: Server Response File Demonstration

Other Resources

IAtlMemMgr Members

ATL Class Overview

IAtlMemMgr Members

Methods

Allocate	Call this method to allocate a block of memory.
Free	Call this method to free a block of memory.
GetSize	Call this method to retrieve the size of an allocated memory block.
Reallocate	Call this method to reallocate a block of memory.

See Also

Reference

IAtlMemMgr Class

IAtlMemMgr Methods

For information about the methods in **IAtlMemMgr**, see IAtlMemMgr Members.

IAtlMemMgr::Allocate

Call this method to allocate a block of memory.

```
void* Allocate(
   size_t nBytes
) throw( );
```

Parameters

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call IAtlMemMgr::Free or IAtlMemMgr::Reallocate to free the memory allocated by this method.

Example

For an example, see the IAtlMemMgr Overview and see the SRFSyntax Sample.

See Also

Reference

IAtlMemMgr Class

Other Resources

IAtlMemMgr Members

IAtlMemMgr::Free

Call this method to free a block of memory.

```
void Free(
   void* p
) throw( );
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

Remarks

Use this method to free memory obtained by IAtlMemMgr::Allocate or IAtlMemMgr::Reallocate.

Example

For an example, see the IAtlMemMgr Overview.

See Also

Reference

IAtlMemMgr Class

Other Resources

IAtlMemMgr Members

IAtlMemMgr::GetSize

Call this method to retrieve the size of an allocated memory block.

```
size_t GetSize(
   void* p
) throw( );
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

Return Value

Returns the size of the memory block in bytes.

Example

For an example, see the IAtlMemMgr Overview.

See Also

Reference

IAtlMemMgr Class

Other Resources

IAtlMemMgr Members

IAtlMemMgr::Reallocate

Call this method to reallocate memory allocated by this memory manager.

```
void* Reallocate(
   void* p,
   size_t nBytes
) throw( );
```

Parameters

p

Pointer to memory previously allocated by this memory manager.

nBytes

The requested number of bytes in the new memory block.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call IAtlMemMgr::Free or IAtlMemMgr::Reallocate to free the memory allocated by this method.

Conceptually this method frees the existing memory and allocates a new memory block. In reality, the existing memory may be extended or otherwise reused.

Example

For an example, see the IAtlMemMgr Overview.

See Also

Reference

IAtlMemMgr Class

Other Resources

IAtlMemMgr Members

IAxWinAmbientDispatch Interface

This interface provides methods for specifying characteristics of the hosted control or container.

```
interface IAxWinAmbientDispatch : IDispatch
```

Remarks

This interface is exposed by ATL's ActiveX control hosting objects. Call the methods on this interface to set the ambient properties available to the hosted control or to specify other aspects of the container's behavior. To supplement the properties provided by **IAxWinAmbientDispatch**, use IAxWinAmbientDispatchEx.

AXHost will try to load type information about **IAxWinAmbientDispatch** and **IAxWinAmbientDispatchEx** from the typelib that contains the code.

If you are linking to ATL80.dll, **AXHost** will load the type information from the typelib in the DLL.

If you are linking the code into the project, the IDL file for the project should reference **IAxWinAmbientDispatch** and **IAxWinAmbientDispatchEx** in order to use them correctly. To make the reference, edit the project IDL file and add the following line after the other imports:

```
import "atliface.idl"
```

and the following line inside the library block:

```
interface IAxWinAmbientDispatchEx;
```

See Hosting ActiveX Controls Using ATL AXHost for more details.

Requirements

The definition of this interface is available in a number of forms, as shown in the table below.

Definition Type	File
IDL	atliface.idl
Type Library	ATL.dll
C++	atliface.h (also included in ATLBase.h)

See Also

Reference

IAxWinAmbientDispatchEx Interface
IAxWinHostWindow Interface
CAxWindow::QueryHost
AtlAxGetHost

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch Members

Methods

get_AllowContextMenu	The **AllowContextMenu** property specifies whether the hosted control is allowed to display its own context menu.
get_AllowShowUI	The **AllowShowUI** property specifies whether the hosted control is allowed to display its own user interface.
get_AllowWindowlessActivation	The **AllowWindowlessActivation** property specifies whether the container will allow windowless activation.
get_BackColor	The **BackColor** property specifies the ambient background color of the container.
get_DisplayAsDefault	**DisplayAsDefault** is an ambient property that allows a control to find out if it is the default control.
get_DocHostDoubleClickFlags	The **DocHostDoubleClickFlags** property specifies the operation that should take place in response to a double-click.
get_DocHostFlags	The **DocHostFlags** property specifies the user interface capabilities of the host object.
get_Font	The **Font** property specifies the ambient font of the container.
get_ForeColor	The **ForeColor** property specifies the ambient foreground color of the container.
get_LocaleID	The **LocaleID** property specifies the ambient locale ID of the container.
get_MessageReflect	The **MessageReflect** ambient property specifies whether the container will reflect messages to the hosted control.
get_OptionKeyPath	The **OptionKeyPath** property specifies the registry key path to user settings.
get_ShowGrabHandles	The **ShowGrabHandles** ambient property allows the control to find out if it should draw itself with grab handles.
get_ShowHatching	The **ShowHatching** ambient property allows the control to find out if it should draw itself hatched.
get_UserMode	The **UserMode** property specifies the ambient user mode of the container.
put_AllowContextMenu	The **AllowContextMenu** property specifies whether the hosted control is allowed to display its own context menu.
put_AllowShowUI	The **AllowShowUI** property specifies whether the hosted control is allowed to display its own user interface.
put_AllowWindowlessActivation	The **AllowWindowlessActivation** property specifies whether the container will allow windowless activation.
put_BackColor	The **BackColor** property specifies the ambient background color of the container.

put_DisplayAsDefault	**DisplayAsDefault** is an ambient property that allows a control to find out if it is the default c ontrol.
put_DocHostDoubleClickFlags	The **DocHostDoubleClickFlags** property specifies the operation that should take place in res ponse to a double-click.
put_DocHostFlags	The **DocHostFlags** property specifies the user interface capabilities of the host object.
put_Font	The **Font** property specifies the ambient font of the container.
put_ForeColor	The **ForeColor** property specifies the ambient foreground color of the container.
put_LocaleID	The **LocaleID** property specifies the ambient locale ID of the container.
put_MessageReflect	The **MessageReflect** ambient property specifies whether the container will reflect messages to the hosted control.
put_OptionKeyPath	The **OptionKeyPath** property specifies the registry key path to user settings.
put_UserMode	The **UserMode** property specifies the ambient user mode of the container.

See Also

Reference

IAxWinAmbientDispatch Interface

IAxWinAmbientDispatch Methods

For information about the methods in **IAxWinAmbientDispatch**, see IAxWinAmbientDispatch Members.

IAxWinAmbientDispatch::get_AllowContextMenu

The **AllowContextMenu** property specifies whether the hosted control is allowed to display its own context menu.

```
STDMETHOD( get_AllowContextMenu )(
   VARIANT_BOOL* pbAllowContextMenu
);
```

Parameters

pbAllowContextMenu

[out] The address of a variable to receive the current value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses **VARIANT_TRUE** as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

IDocHostUIHandler::ShowContextMenu

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::get_AllowShowUI

The **AllowShowUI** property specifies whether the hosted control is allowed to display its own user interface.

```
STDMETHOD( get_AllowShowUI )(
   VARIANT_BOOL* pbAllowShowUI
);
```

Parameters

pbAllowShowUI

[out] The address of a variable to receive the current value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses **VARIANT_FALSE** as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

IDocHostUIHandler::ShowUI

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::get_AllowWindowlessActivation

The **AllowWindowlessActivation** property specifies whether the container will allow windowless activation.

```
STDMETHOD( get_AllowWindowlessActivation )(
   VARIANT_BOOL* pbAllowWindowless
);
```

Parameters

pbAllowWindowless

[out] The address of a variable to receive the current value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses **VARIANT_TRUE** as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

IOleInPlaceSiteWindowless::CanWindowlessActivate

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::get_BackColor

The **BackColor** property specifies the ambient background color of the container.

```
STDMETHOD( get_BackColor )(
   OLE_COLOR* pclrBackground
);
```

Parameters

pclrBackground

[out] The address of a variable to receive the current value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses **COLOR_BTNFACE** or **COLOR_WINDOW** as the default value of this property (depending on whether the parent of the host window is a dialog or not).

See Also

Reference

IAxWinAmbientDispatch Interface

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::get_DisplayAsDefault

DisplayAsDefault is an ambient property that allows a control to find out if it is the default control.

```
STDMETHOD( get_DisplayAsDefault )(
   VARIANT_BOOL* pbDisplayAsDefault
);
```

Parameters

pbDisplayAsDefault

[out] The address of a variable to receive the current value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses **VARIANT_FALSE** as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::get_DocHostDoubleClickFlags

The **DocHostDoubleClickFlags** property specifies the operation that should take place in response to a double-click.

```
STDMETHOD( get_DocHostDoubleClickFlags )(
   DWORD* pdwDocHostDoubleClickFlags
);
```

Parameters

pdwDocHostDoubleClickFlags

[out] The address of a variable to receive the current value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses **DOCHOSTUIDBLCLK_DEFAULT** as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

IDocHostUIHandler::GetHostInfo

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::get_DocHostFlags

The **DocHostFlags** property specifies the user interface capabilities of the host object.

```
STDMETHOD( get_DocHostFlags )(
   DWORD* pdwDocHostFlags
);
```

Parameters

pdwDocHostFlags

[out] The address of a variable to receive the current value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses **DOCHOSTUIFLAG_NO3DBORDER** as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

IDocHostUIHandler::GetHostInfo

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::get_Font

The **Font** property specifies the ambient font of the container.

```
STDMETHOD( get_Font )(
   IFontDisp** pFont
);
```

Parameters

pFont

[out] The address of an **IFontDisp** interface pointer used to receive the current value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses the default GUI font or the system font as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::get_ForeColor

The **ForeColor** property specifies the ambient foreground color of the container.

```
STDMETHOD( get_ForeColor )(
   OLE_COLOR* pclrForeground
);
```

Parameters

pclrForeground

[out] The address of a variable to receive the current value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses the system window text color as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::get_LocaleID

The **LocaleID** property specifies the ambient locale ID of the container.

```
STDMETHOD( get_LocaleID )(
   LCID* plcidLocaleID
);
```

Parameters
plcidLocaleID

[out] The address of a variable to receive the current value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses the user's default locale as the default value of this property.

With this method you can discover the Ambient LocalID, that is, the LocaleID of the program your control is being used in. Once you know the LocaleID, you can call code to load locale-specific captions, error message text, and so forth from a resource file or satellite DLL.

See Also
Reference
IAxWinAmbientDispatch Interface
Other Resources
IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::get_MessageReflect

The **MessageReflect** ambient property specifies whether the container will reflect messages to the hosted control.

```
STDMETHOD( get_MessageReflect )(
   VARIANT_BOOL* pbMessageReflect
);
```

Parameters

pbMessageReflect

[out] The address of a variable to receive the current value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses **VARIANT_TRUE** as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::get_OptionKeyPath

The **OptionKeyPath** property specifies the registry key path to user settings.

```
STDMETHOD( get_OptionKeyPath )(
   BSTR* pbstrOptionKeyPath
);
```

Parameters

pbstrOptionKeyPath

[out] The address of a variable to receive the current value of this property.

Return Value

A standard **HRESULT** value.

See Also

Reference

IAxWinAmbientDispatch Interface

IDocHostUIHandler::GetOptionKeyPath

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::get_ShowGrabHandles

The **ShowGrabHandles** ambient property allows the control to find out if it should draw itself with grab handles.

```
STDMETHOD( get_ShowGrabHandles )(
   VARIANT_BOOL* pbShowGrabHandles
);
```

Parameters

pbShowGrabHandles

[out] The address of a variable to receive the current value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation always returns **VARIANT_FALSE** as the value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::get_ShowHatching

The **ShowHatching** ambient property allows the control to find out if it should draw itself hatched.

```
STDMETHOD( get_ShowHatching )(
   VARIANT_BOOL* pbShowHatching
);
```

Parameters

pbShowHatching

[out] The address of a variable to receive the current value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation always returns **VARIANT_FALSE** as the value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::get_UserMode

The **UserMode** property specifies the ambient user mode of the container.

```
STDMETHOD( get_UserMode )(
   VARIANT_BOOL* pbUserMode
);
```

Parameters

pbUserMode

[out] The address of a variable to receive the current value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses **VARIANT_TRUE** as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::put_AllowContextMenu

The **AllowContextMenu** property specifies whether the hosted control is allowed to display its own context menu.

```
STDMETHOD( put_AllowContextMenu )(
   VARIANT_BOOL bAllowContextMenu
);
```

Parameters

bAllowContextMenu

[in] The new value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses **VARIANT_TRUE** as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

IDocHostUIHandler::ShowContextMenu

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::put_AllowShowUI

The **AllowShowUI** property specifies whether the hosted control is allowed to display its own user interface.

```
STDMETHOD( put_AllowShowUI )(
   VARIANT_BOOL bAllowShowUI
);
```

Parameters

bAllowShowUI

[in] The new value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses **VARIANT_FALSE** as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::put_AllowWindowlessActivation

The **AllowWindowlessActivation** property specifies whether the container will allow windowless activation.

```
STDMETHOD( put_AllowWindowlessActivation )(
   VARIANT_BOOL bAllowWindowless
);
```

Parameters

bAllowWindowless

[in] The new value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses **VARIANT_TRUE** as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::put_BackColor

The **BackColor** property specifies the ambient background color of the container.

```
STDMETHOD( put_BackColor )(
   OLE_COLOR clrBackground
);
```

Parameters

clrBackground

[in] The new value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses **COLOR_BTNFACE** or **COLOR_WINDOW** as the default value of this property (depending on whether the parent of the host window is a dialog or not).

See Also

Reference

IAxWinAmbientDispatch Interface

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::put_DisplayAsDefault

DisplayAsDefault is an ambient property that allows a control to find out if it is the default control.

```
STDMETHOD( put_DisplayAsDefault )(
   VARIANT_BOOL bDisplayAsDefault
);
```

Parameters

bDisplayAsDefault

[in] The new value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses **VARIANT_FALSE** as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::put_DocHostDoubleClickFlags

The **DocHostDoubleClickFlags** property specifies the operation that should take place in response to a double-click.

```
STDMETHOD( put_DocHostDoubleClickFlags )(
   DWORD dwDocHostDoubleClickFlags
);
```

Parameters

dwDocHostDoubleClickFlags

[in] The new value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses **DOCHOSTUIDBLCLK_DEFAULT** as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

IDocHostUIHandler::GetHostInfo

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::put_DocHostFlags

The **DocHostFlags** property specifies the user interface capabilities of the host object.

```
STDMETHOD( put_DocHostFlags )(
   DWORD dwDocHostFlags
);
```

Parameters

dwDocHostFlags

[in] The new value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses **DOCHOSTUIFLAG_NO3DBORDER** as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

IDocHostUIHandler::GetHostInfo

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::put_Font

The **Font** property specifies the ambient font of the container.

```
STDMETHOD( put_Font )(
   IFontDisp* pFont
);
```

Parameters

pFont

[in] The new value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses the default GUI font or the system font as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::put_ForeColor

The **ForeColor** property specifies the ambient foreground color of the container.

```
STDMETHOD( put_ForeColor )(
   OLE_COLOR clrForeground
);
```

Parameters

clrForeground

[in] The new value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses the system window text color as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::put_LocaleID

The **LocaleID** property specifies the ambient locale ID of the container.

```
STDMETHOD( put_LocaleID )(
   LCID lcidLocaleID
);
```

Parameters
lcidLocaleID

[in] The new value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses the user's default locale as the default value of this property.

See Also
Reference
IAxWinAmbientDispatch Interface
Other Resources
IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::put_MessageReflect

The **MessageReflect** ambient property specifies whether the container will reflect messages to the hosted control.

```
STDMETHOD( put_MessageReflect )(
   VARIANT_BOOL bMessageReflect
);
```

Parameters

bMessageReflect

[in] The new value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses **VARIANT_TRUE** as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::put_OptionKeyPath

The **OptionKeyPath** property specifies the registry key path to user settings.

```
STDMETHOD( put_OptionKeyPath )(
   BSTR bstrOptionKeyPath
);
```

Parameters

bstrOptionKeyPath

[in] The new value of this property.

Return Value

A standard **HRESULT** value.

See Also

Reference

IAxWinAmbientDispatch Interface

IDocHostUIHandler::GetOptionKeyPath

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatch::put_UserMode

The **UserMode** property specifies the ambient user mode of the container.

```
STDMETHOD( put_UserMode )(
   VARIANT_BOOL bUserMode
);
```

Parameters

bUserMode

[in] The new value of this property.

Return Value

A standard **HRESULT** value.

Remarks

The ATL host object implementation uses **VARIANT_TRUE** as the default value of this property.

See Also

Reference

IAxWinAmbientDispatch Interface

Other Resources

IAxWinAmbientDispatch Members

IAxWinAmbientDispatchEx Interface

This interface implements supplemental ambient properties for a hosted control.

```
MIDL_INTERFACE( "B2D0778B - AC99 - 4c58 - A5C8 - E7724E5316B5" )
IAxWinAmbientDispatchEx : public IAxWinAmbientDispatch
```

Remarks

Include this interface in ATL applications that are statically linked to ATL and host ActiveX Controls, especially ActiveX Controls that have Ambient Properties. Not including this interface will generate this assertion: "Did you forget to pass the LIBID to CComModule::Init?"

This interface is exposed by ATL's ActiveX control hosting objects. Derived from IAxWinAmbientDispatch, **IAxWinAmbientDispatchEx** adds a method that allows you to supplement the ambient property interface provided by ATL with one of your own.

AXHost will try to load type information about **IAxWinAmbientDispatch** and **IAxWinAmbientDispatchEx** from the type library that contains the code.

If you are linking to ATL80.dll, **AXHost** will load the type information from the type library in the DLL.

If you are linking the code into the project, the IDL file for the project should reference **IAxWinAmbientDispatch** and **IAxWinAmbientDispatchEx** in order to use them correctly. To make the reference, edit the project IDL file and add the following line after the other imports:

```
import "atliface.idl"
```

and the following line inside the library block:

```
interface IAxWinAmbientDispatchEx;
```

See Hosting ActiveX Controls Using ATL AXHost for more details.

Requirements

The definition of this interface is available in a number of forms, as shown in the following table.

Definition Type	File
IDL	atliface.idl
Type Library	ATL.dll
C++	atliface.h (also included in ATLBase.h)

See Also

Reference

IAxWinAmbientDispatch Interface

Other Resources

IAxWinAmbientDispatchEx Members

IAxWinAmbientDispatchEx Members

Methods

SetAmbientDispatch	This method is called to supplement the default ambient property interface with a user-defined interface.

See Also

Reference

IAxWinAmbientDispatchEx Interface

IAxWinAmbientDispatchEx Methods

For information about the methods in **IAxWinAmbientDispatchEx**, see IAxWinAmbientDispatchEx Members.

IAxWinAmbientDispatchEx::SetAmbientDispatch

This method is called to supplement the default ambient property interface with a user-defined interface.

```
virtual HRESULT STDMETHODCALLTYPE SetAmbientDispatch(
   IDispatch* pDispatch
) = 0;
```

Parameters

pDispatch

Pointer to the new interface.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

When **SetAmbientDispatch** is called with a pointer to a new interface, this new interface will be used to invoke any properties or methods asked for by the hosted control — if those properties are not already provided by IAxWinAmbientDispatch.

See Also

Reference

IAxWinAmbientDispatchEx Interface

Other Resources

IAxWinAmbientDispatchEx Members

IAxWinHostWindow Interface

This interface provides methods for manipulating a control and its host object.

```
interface IAxWinHostWindow : IUnknown
```

Remarks

This interface is exposed by ATL's ActiveX control hosting objects. Call the methods on this interface to create and/or attach a control to the host object, to get an interface from a hosted control, or to set the external dispinterface or UI handler for use when hosting the Web browser.

Requirements

The definition of this interface is available as IDL or C++, as shown below.

Definition type	File
IDL	ATLIFace.idl
C++	ATLIFace.h (also included in ATLBase.h)

See Also

Reference

IAxWinAmbientDispatch Interface
CAxWindow::QueryHost
AtlAxGetHost

Other Resources

IAxWinHostWindow Members

IAxWinHostWindow Members

Methods

AttachControl	Attaches an existing control to the host object.
CreateControl	Creates a control and attaches it to the host object.
CreateControlEx	Creates a control, attaches it to the host object, and optionally sets up an event handler.
QueryControl	Returns an interface pointer to the hosted control.
SetExternalDispatch	Sets the external **IDispatch** interface.
SetExternalUIHandler	Sets the external **IDocHostUIHandlerDispatch** interface.

See Also

Reference

IAxWinHostWindow Interface

IAxWinHostWindow Methods

For information about the methods in **IAxWinHostWindow**, see IAxWinHostWindow Members.

IAxWinHostWindow::AttachControl

Attaches an existing (and previously initialized) control to the host object using the window identified by *hWnd*.

```
STDMETHOD( AttachControl )(
   IUnknown* pUnkControl,
   HWND hWnd
);
```

Parameters

pUnkControl

[in] A pointer to the **IUnknown** interface of the control to be attached to the host object.

hWnd

[in] A handle to the window to be used for hosting.

Return Value

A standard **HRESULT** value.

See Also

Reference

IAxWinHostWindow Interface
IAxWinHostWindow::CreateControl
IAxWinHostWindow::CreateControlEx
CAxWindow::AttachControl
AtlAxAttachControl

Other Resources

IAxWinHostWindow Members

IAxWinHostWindow::CreateControl

Creates a control, initializes it, and hosts it in the window identified by *hWnd*.

```
STDMETHOD( CreateControl )(
   LPCOLESTR lpTricsData,
   HWND hWnd,
   IStream* pStream
);
```

Parameters

lpTricsData

[in] A string identifying the control to create. Can be a CLSID (must include the braces), ProgID, URL, or raw HTML (prefixed by **MSHTML:**).

hWnd

[in] A handle to the window to be used for hosting.

pStream

[in] An interface pointer for a stream containing initialization data for the control. Can be **NULL**.

Return Value

A standard **HRESULT** value.

Remarks

This window will be subclassed by the host object exposing this interface so that messages can be reflected to the control and other container features will work.

Calling this method is equivalent to calling IAxWinHostWindow::CreateControlEx.

To create a licensed ActiveX control, see IAxWinHostWindowLic::CreateControlLic.

See Also

Reference

IAxWinHostWindow Interface
IAxWinHostWindow::CreateControlEx
IAxWinHostWindow::AttachControl
CAxWindow::CreateControl
AtlAxCreateControl

Other Resources

IAxWinHostWindow Members

IAxWinHostWindow::CreateControlEx

Creates an ActiveX control, initializes it, and hosts it in the specified window, similar to IAxWinHostWindow::CreateControl.

```
STDMETHOD( CreateControlEx )(
   LPCOLESTR lpszTricsData,
   HWND hWnd,
   IStream* pStream,
   IUnknown** ppUnk,
   REFIID riidAdvise,
   IUnknown* punkAdvise
);
```

Parameters

lpTricsData

[in] A string identifying the control to create. Can be a CLSID (must include the braces), ProgID, URL, or raw HTML (prefixed with **MSHTML:**).

hWnd

[in] A handle to the window to be used for hosting.

pStream

[in] An interface pointer for a stream containing initialization data for the control. Can be **NULL**.

ppUnk

[out] The address of a pointer that will receive the **IUnknown** interface of the created control. Can be **NULL**.

riidAdvise

[in] The interface identifier of an outgoing interface on the contained object. Can be **IID_NULL**.

punkAdvise

[in] A pointer to the **IUnknown** interface of the sink object to be connected to the connection point on the contained object specified by *iidSink*.

Return Value

A standard **HRESULT** value.

Remarks

Unlike the **CreateControl** method, **CreateControlEx** also allows you to receive an interface pointer to the newly created control and set up an event sink to receive events fired by the control.

To create a licensed ActiveX control, see IAxWinHostWindowLic::CreateControlLicEx.

See Also

Reference

IAxWinHostWindow Interface
IAxWinHostWindow::CreateControl
IAxWinHostWindow::AttachControl
CAxWindow::CreateControlEx
AtlAxCreateControlEx

Other Resources

IAxWinHostWindow Members

IAxWinHostWindow::QueryControl

Returns the specified interface pointer provided by the hosted control.

```
STDMETHOD( QueryControl )(
   REFIID riid,
   void** ppvObject
);
```

Parameters

riid

[in] The ID of an interface on the control being requested.

ppvObject

[out] The address of a pointer that will receive the specified interface of the created control.

Return Value

A standard **HRESULT** value.

See Also

Reference

IAxWinHostWindow Interface
CAxWindow::QueryControl
AtlAxGetControl

Other Resources

IAxWinHostWindow Members

IAxWinHostWindow::SetExternalDispatch

Sets the external dispinterface, which is available to contained controls through the IDocHostUIHandlerDispatch::GetExternal method.

```
STDMETHOD( SetExternalDispatch )(
   IDispatch* pDisp
);
```

Parameters

pDisp

[in] A pointer to an **IDispatch** interface.

Return Value

A standard **HRESULT** value.

See Also

Reference

IAxWinHostWindow Interface

CAxWindow::SetExternalDispatch

Other Resources

IAxWinHostWindow Members

IAxWinHostWindow::SetExternalUIHandler

Call this function to set the external IDocHostUIHandlerDispatch interface for the **CAxWindow** object.

```
STDMETHOD( SetExternalUIHandler )(
   IDocHostUIHandlerDispatch* pDisp
);
```

Parameters

pDisp

[in] A pointer to an **IDocHostUIHandlerDispatch** interface.

Return Value

A standard **HRESULT** value.

Remarks

This function is used by controls (such as the Web browser control) that query the host's site for the **IDocHostUIHandlerDispatch** interface.

See Also

Reference

IAxWinHostWindow Interface

CAxWindow::SetExternalUIHandler

Other Resources

IAxWinHostWindow Members

IAxWinHostWindowLic Interface

This interface provides methods for manipulating a licensed control and its host object.

```
interface IAxWinHostWindowLic : IAxWinHostWindow
```

Remarks

IAxWinHostWindowLic inherits from IAxWinHostWindow and adds methods that support the creation of licensed controls.

See Hosting ActiveX Controls Using ATL AXHost for a sample that uses the members of this interface.

Requirements

The definition of this interface is available as IDL or C++, as shown below.

Definition type	File
IDL	ATLIFace.idl
C++	ATLIFace.h (also included in ATLBase.h)

See Also

Other Resources

IAxWinHostWindowLic Members

IAxWinHostWindowLic Members

Methods

CreateControlLic	Creates a licensed control and attaches it to the host object.
CreateControlLicEx	Creates a licensed control, attaches it to the host object, and optionally sets up an event handler.

See Also

Reference

IAxWinHostWindowLic Interface

IAxWinHostWindowLic Methods

For information about the methods in **IAxWinHostWindow**, see IAxWinHostWindowLic Members.

IAxWinHostWindowLic::CreateControlLic

Creates a licensed control, initializes it, and hosts it in the window identified by *hWnd*.

```
STDMETHOD( CreateControlLic )(
   LPCOLESTR lpTricsData,
   HWND hWnd,
   IStream* pStream,
   BSTR bstrLic
);
```

Parameters

bstrLic

[in] The BSTR that contains the license key for the control.

Remarks

See IAxWinHostWindow::CreateControl for a description of the remaining parameters and return value.

Calling this method is equivalent to calling IAxWinHostWindowLic::CreateControlLicEx

Example

See Hosting ActiveX Controls Using ATL AXHost for a sample that uses **IAxWinHostWindowLic::CreateControlLic**.

See Also

Reference

IAxWinHostWindowLic Interface

Other Resources

IAxWinHostWindowLic Members

IAxWinHostWindowLic::CreateControlLicEx

Creates a licensed ActiveX control, initializes it, and hosts it in the specified window, similar to IAxWinHostWindow::CreateControl.

```
STDMETHOD( CreateControlLicEx )(
   LPCOLESTR lpszTricsData,
   HWND hWnd,
   IStream* pStream,
   IUnknown** ppUnk,
   REFIID riidAdvise,
   IUnknown* punkAdvise,
      BSTR bstrLic
);
```

Parameters

bstrLic

[in] The BSTR that contains the license key for the control.

Remarks

See IAxWinHostWindow::CreateControlEx for a description of the remaining parameters and return value.

Example

See Hosting ActiveX Controls Using ATL AXHost for a sample that uses **IAxWinHostWindowLic::CreateControlLicEx**.

See Also

Reference

IAxWinHostWindowLic Interface

Other Resources

IAxWinHostWindowLic Members

ICollectionOnSTLImpl Class

This class provides methods used by a collection class.

```
template <
   class T,
   class CollType,
   class ItemType,
   class CopyItem,
   class EnumType
>
class ICollectionOnSTLImpl :
   public T
```

Parameters

T

A COM collection interface.

CollType

An STL container class.

ItemType

The type of item exposed by the container interface.

CopyItem

A copy policy class.

EnumType

A CComEnumOnSTL-compatible enumerator class.

Remarks

This class provides the implementation for three methods of a collection interface: get_Count, get_Item, and get__NewEnum.

To use this class:

- Define (or borrow) a collection interface that you wish to implement.
- Derive your class from a specialization of **ICollectionOnSTLImpl** based on this collection interface.
- Use your derived class to implement any methods from the collection interface not handled by **ICollectionOnSTLImpl**.

Note
If the collection interface is a dual interface, derive your class from IDispatchImpl, passing the **ICollectionOnSTLImpl** specialization as the first template parameter if you want ATL to provide the implementation of the **IDispatch** methods.

- Add items to the m_coll member to populate the collection.

For more information and examples, see ATL Collections and Enumerators.

Requirements

Header: atlcom.h

See Also

Tasks

ATLCollections Sample: Demonstrates ICollectionOnSTLImpl, CComEnumOnSTL, and Custom Copy Policy Classes

Other Resources

ICollectionOnSTLImpl Members

ATL Class Overview

ICollectionOnSTLImpl Members

Methods

get_Count	Returns the number of elements in the collection.
get_Item	Returns the requested item from the collection.
get__NewEnum	Returns an enumerator object for the collection.

Data Members

m_coll	The collection.

See Also

Reference

ICollectionOnSTLImpl Class

ICollectionOnSTLImpl Methods

For information about the methods in **ICollectionOnSTLImpl**, see ICollectionOnSTLImpl Members.

ICollectionOnSTLImpl::get_Count

This method returns the number of items in the collection.

```
STDMETHOD(get_Count)(
   long* pcount
);
```

Parameters

pcount

[out] The number of elements in the collection.

Return Value

A standard **HRESULT** value.

See Also

Reference

ICollectionOnSTLImpl Class

Other Resources

ICollectionOnSTLImpl Members

ICollectionOnSTLImpl::get_Item

This method returns the specified item from the collection.

```
STDMETHOD(get_Item)(
   long Index,
   ItemType* pvar
);
```

Parameters

Index

[in] The 1-based index of an item in the collection.

pvar

[out] The item corresponding to *Index*.

Return Value

A standard **HRESULT** value.

Remarks

The item is obtained by copying the data at the specified position in m_coll using the copy method of the copy policy class passed as a template argument in the **ICollectionOnSTLImpl** specialization.

See Also

Reference

ICollectionOnSTLImpl Class

Other Resources

ICollectionOnSTLImpl Members

ICollectionOnSTLImpl::get__NewEnum

Returns an enumerator object for the collection.

```
STDMETHOD(get__NewEnum)(
   IUnknown** ppUnk
);
```

Parameters

ppUnk

[out] The **IUnknown** pointer of a newly created enumerator object.

Return Value

A standard **HRESULT** value.

Remarks

The newly created enumerator maintains an iterator on the original collection, **m_coll**, (so no copy is made) and holds a COM reference on the collection object to ensure that the collection remains alive while there are outstanding enumerators.

See Also

Reference

ICollectionOnSTLImpl Class

Concepts

ATL Collections and Enumerators

Other Resources

ICollectionOnSTLImpl Members

ICollectionOnSTLImpl Data Members

For information about the data members in **ICollectionOnSTLImpl**, see ICollectionOnSTLImpl Members.

ICollectionOnSTLImpl::m_coll

This member holds the items represented by the collection.

```
CollType m_coll;
```

See Also

Reference

ICollectionOnSTLImpl Class

Other Resources

ICollectionOnSTLImpl Members

IConnectionPointContainerImpl Class

This class implements a connection point container to manage a collection of IConnectionPointImpl objects.

```
template<
   class T
>
class ATL_NO_VTABLE IConnectionPointContainerImpl :
   public IConnectionPointContainer
```

Parameters

T

Your class, derived from **IConnectionPointContainerImpl**.

Remarks

IConnectionPointContainerImpl implements a connection point container to manage a collection of IConnectionPointImpl objects. **IConnectionPointContainerImpl** provides two methods that a client can call to retrieve more information about a connectable object:

- **EnumConnectionPoints** allows the client to determine which outgoing interfaces the object supports.
- **FindConnectionPoint** allows the client to determine whether the object supports a specific outgoing interface.

For information about using connection points in ATL, see the article Connection Points.

Requirements

Header: atlcom.h

See Also

Reference

IConnectionPointContainer

Other Resources

IConnectionPointContainerImpl Members

ATL Class Overview

IConnectionPointContainerImpl Members

Methods

EnumConnectionPoints	Creates an enumerator to iterate through the connection points supported in the connectable object.
FindConnectionPoint	Retrieves an interface pointer to the connection point that supports the specified IID.

See Also

Reference

IConnectionPointContainerImpl Class

IConnectionPointContainerImpl Methods

For information about the methods in **IConnectionPointContainerImpl**, see IConnectionPointContainerImpl Members.

IConnectionPointContainerImpl::EnumConnectionPoints

Creates an enumerator to iterate through the connection points supported in the connectable object.

```
STDMETHOD(EnumConnectionPoints)(
   IEnumConnectionPoints **ppEnum
);
```

Remarks

See IConnectionPointContainer::EnumConnectionPoints in the *Platform SDK*.

See Also

Reference

IConnectionPointContainerImpl Class
IConnectionPointImpl Class
IEnumConnectionPoints

Other Resources

IConnectionPointContainerImpl Members

IConnectionPointContainerImpl::FindConnectionPoint

Retrieves an interface pointer to the connection point that supports the specified IID.

```
STDMETHOD(FindConnectionPoint)(
   REFIID riid,
   IConnectionPoint** ppCP
);
```

Remarks

See IConnectionPointContainer::FindConnectionPoint in the *Platform SDK*.

See Also

Reference

IConnectionPointContainerImpl Class

IConnectionPointImpl Class

Other Resources

IConnectionPointContainerImpl Members

IConnectionPointImpl Class

This class implements a connection point.

```
template<
   class T,
   const IID* piid,
   class CDV = CComDynamicUnkArray
>
class ATL_NO_VTABLE IConnectionPointImpl :
   public _ICPLocator< piid >
```

Parameters

T

Your class, derived from **IConnectionPointImpl**.

piid

A pointer to the IID of the interface represented by the connection point object.

CDV

A class that manages the connections. The default value is CComDynamicUnkArray, which allows unlimited connections. You can also use CComUnkArray, which specifies a fixed number of connections.

Remarks

IConnectionPointImpl implements a connection point, which allows an object to expose an outgoing interface to the client. The client implements this interface on an object called a sink.

ATL uses IConnectionPointContainerImpl to implement the connectable object. Each connection point within the connectable object represents an outgoing interface, identified by *piid*. Class *CDV* manages the connections between the connection point and a sink. Each connection is uniquely identified by a "cookie."

For more information about using connection points in ATL, see the article Connection Points.

Requirements

Header: atlcom.h

See Also

Reference

IConnectionPoint

Other Resources

IConnectionPointImpl Members

ATL Class Overview

IConnectionPointImpl Members

Methods

Advise	Establishes a connection between the connection point and a sink.
EnumConnections	Creates an enumerator to iterate through the connections for the connection point.
GetConnectionInterface	Retrieves the IID of the interface represented by the connection point.
GetConnectionPointContainer	Retrieves an interface pointer to the connectable object.
Unadvise	Terminates a connection previously established through **Advise**.

Data Members

m_vec	Manages the connections for the connection point.

See Also

Reference

IConnectionPointImpl Class

IConnectionPointImpl Methods

For information about the methods in **IConnectionPointImpl**, see IConnectionPointImpl Members.

IConnectionPointImpl::Advise

Establishes a connection between the connection point and a sink.

```
STDMETHOD(Advise)(
   IUnknown* pUnkSink,
   DWORD* pdwCookie
);
```

Remarks

Use Unadvise to terminate the connection call.

See IConnectionPoint::Advise in the *Platform SDK*.

See Also

Reference

IConnectionPointImpl Class

Other Resources

IConnectionPointImpl Members

IConnectionPointImpl::EnumConnections

Creates an enumerator to iterate through the connections for the connection point.

```
STDMETHOD(EnumConnections)(
   IEnumConnections** ppEnum
);
```

Remarks

See IConnectionPoint::EnumConnections in the *Platform SDK*.

See Also

Reference

IConnectionPointImpl Class

IEnumConnections

Other Resources

IConnectionPointImpl Members

IConnectionPointImpl::GetConnectionInterface

Retrieves the IID of the interface represented by the connection point.

```
STDMETHOD(GetConnectionInterface)(
   IID* piid2
);
```

Remarks

See IConnectionPoint::GetConnectionInterface in the *Platform SDK*.

See Also

Reference

IConnectionPointImpl Class

Other Resources

IConnectionPointImpl Members

IConnectionPointImpl::GetConnectionPointContainer

Retrieves an interface pointer to the connectable object.

```
STDMETHOD(GetConnectionPointContainer)(
   IConnectionPointContainer** ppCPC
);
```

Remarks

See IConnectionPoint::GetConnectionPointContainer in the *Platform SDK*.

See Also

Reference

IConnectionPointImpl Class

IConnectionPointContainerImpl Class

Other Resources

IConnectionPointImpl Members

IConnectionPointImpl::Unadvise

Terminates a connection previously established through Advise.

```
STDMETHOD(Unadvise)(
   DWORD dwCookie
);
```

Remarks

See IConnectionPoint::Unadvise in the *Platform SDK*.

See Also

Reference

IConnectionPointImpl Class

Other Resources

IConnectionPointImpl Members

IConnectionPointImpl Data Members

For information about the data members in **IConnectionPointImpl**, see IConnectionPointImpl Members.

IConnectionPointImpl::m_vec

Manages the connections between the connection point object and a sink.

```
CDV m_vec;
```

Remarks

By default, **m_vec** is of type CComDynamicUnkArray.

See Also

Reference

IConnectionPointImpl Class

Other Resources

IConnectionPointImpl Members

IDataObjectImpl Class

This class provides methods for supporting Uniform Data Transfer and managing connections.

```
template< class T >
class IDataObjectImpl
```

Parameters

T

Your class, derived from **IDataObjectImpl**.

Remarks

The IDataObject interface provides methods to support Uniform Data Transfer. **IDataObject** uses the standard format structures FORMATETC and STGMEDIUM to retrieve and store data.

IDataObject also manages connections to advise sinks to handle data change notifications. In order for the client to receive data change notifications from the data object, the client must implement the IAdviseSink interface on an object called an advise sink. When the client then calls **IDataObject::DAdvise**, a connection is established between the data object and the advise sink.

Class **IDataObjectImpl** provides a default implementation of **IDataObject** and implements **IUnknown** by sending information to the dump device in debug builds.

Related Articles ATL Tutorial, Creating an ATL Project

Requirements

Header: atlctl.h

See Also

Other Resources

IDataObjectImpl Members

ATL Class Overview

IDataObjectImpl Members

Class Methods

FireDataChange	Sends a change notification back to each advise sink.

IDataObject Methods

DAdvise	Establishes a connection between the data object and an advise sink. This enables the advise sink to receive notifications of changes in the object.
DUnadvise	Terminates a connection previously established through **DAdvise**.
EnumDAdvise	Creates an enumerator to iterate through the current advisory connections.
EnumFormatEtc	Creates an enumerator to iterate through the **FORMATETC** structures supported by the data object. The ATL implementation returns **E_NOTIMPL**.
GetCanonicalFormatEtc	Retrieves a logically equivalent **FORMATETC** structure to one that is more complex. The ATL implementation returns **E_NOTIMPL**.
GetData	Transfers data from the data object to the client. The data is described in a **FORMATETC** structure and is transferred through a **STGMEDIUM** structure.
GetDataHere	Similar to **GetData**, except the client must allocate the **STGMEDIUM** structure. The ATL implementation returns **E_NOTIMPL**.
QueryGetData	Determines whether the data object supports a particular **FORMATETC** structure for transferring data. The ATL implementation returns **E_NOTIMPL**.
SetData	Transfers data from the client to the data object. The ATL implementation returns **E_NOTIMPL**.

See Also

Reference

IDataObjectImpl Class

IDataObjectImpl Methods

For information about the methods in **IDataObjectImpl**, see IDataObjectImpl Members.

IDataObjectImpl::DAdvise

Establishes a connection between the data object and an advise sink.

```
HRESULT DAdvise(
   FORMATETC* pformatetc,
   DWORD advf,
   IAdviseSink* pAdvSink,
   DWORD* pdwConnection
);
```

Remarks

This enables the advise sink to receive notifications of changes in the object.

To terminate the connection, call DUnadvise.

See IDataObject::DAdvise in the *Platform SDK*.

See Also

Reference

IDataObjectImpl Class
FORMATETC
IAdviseSink

Other Resources

IDataObjectImpl Members

IDataObjectImpl::DUnadvise

Terminates a connection previously established through DAdvise.

```
HRESULT DUnadvise(
   DWORD dwConnection
);
```

Remarks

See IDataObject::DUnadvise in the *Platform SDK*.

See Also

Reference

IDataObjectImpl Class
FORMATETC
IAdviseSink

Other Resources

IDataObjectImpl Members

IDataObjectImpl::EnumDAdvise

Creates an enumerator to iterate through the current advisory connections.

```
HRESULT DAdvise(
   FORMATETC* pformatetc,
   DWORD advf,
   IAdviseSink* pAdvSink,
   DWORD* pdwConnection
);
```

Remarks

See IDataObject::EnumDAdvise in the *Platform SDK*.

See Also

Reference

IDataObjectImpl Class
FORMATETC
IAdviseSink

Other Resources

IDataObjectImpl Members

IDataObjectImpl::EnumFormatEtc

Creates an enumerator to iterate through the **FORMATETC** structures supported by the data object.

```
HRESULT EnumFormatEtc(
   DWORD dwDirection,
   IEnumFORMATETC** ppenumFormatEtc
);
```

Remarks

See IDataObject::EnumFormatEtc in the *Platform SDK*.

Return Value

Returns **E_NOTIMPL**.

See Also

Reference

IDataObjectImpl Class

IEnumFORMATETC

Other Resources

IDataObjectImpl Members

IDataObjectImpl::FireDataChange

Sends a change notification back to each advise sink that is currently being managed.

```
HRESULT FireDataChange( );
```

Return Value

A standard **HRESULT** value.

See Also

Reference

IDataObjectImpl Class

Other Resources

IDataObjectImpl Members

IDataObjectImpl::GetCanonicalFormatEtc

Retrieves a logically equivalent **FORMATETC** structure to one that is more complex.

```
HRESULT GetCanonicalFormatEtc(
   FORMATETC* pformatetcIn,
   FORMATETC* pformatetcOut
);
```

Return Value

Returns **E_NOTIMPL**.

Remarks

See IDataObject::GetCanonicalFormatEtc in the *Platform SDK*.

See Also

Reference

IDataObjectImpl Class

FORMATETC

Other Resources

IDataObjectImpl Members

IDataObjectImpl::GetData

Transfers data from the data object to the client.

```
HRESULT GetData(
   FORMATETC* pformatetcIn,
   STGMEDIUM* pmedium
);
```

Remarks

The *pformatetcIn* parameter must specify a storage medium type of **TYMED_MFPICT**.

See IDataObject::GetData in the *Platform SDK*.

See Also

Reference

IDataObjectImpl Class
IDataObjectImpl::GetDataHere
IDataObjectImpl::QueryGetData
IDataObjectImpl::SetData
FORMATETC
STGMEDIUM
TYMED

Other Resources

IDataObjectImpl Members

IDataObjectImpl::GetDataHere

Similar to **GetData**, except the client must allocate the **STGMEDIUM** structure.

```
HRESULT GetDataHere(
   FORMATETC* pformatetc,
   STGMEDIUM* pmedium
);
```

Return Value

Returns **E_NOTIMPL**.

Remarks

See IDataObject::GetDataHere in the *Platform SDK*.

See Also

Reference

IDataObjectImpl Class
IDataObjectImpl::GetData
IDataObjectImpl::QueryGetData
IDataObjectImpl::SetData
FORMATETC
STGMEDIUM

Other Resources

IDataObjectImpl Members

IDataObjectImpl::QueryGetData

Determines whether the data object supports a particular **FORMATETC** structure for transferring data.

```
HRESULT QueryGetData(
   FORMATETC* pformatetc
);
```

Return Value

Returns **E_NOTIMPL**.

Remarks

See IDataObject::QueryGetData in the *Platform SDK*.

See Also

Reference

IDataObjectImpl Class
IDataObjectImpl::GetData
IDataObjectImpl::GetDataHere
IDataObjectImpl::SetData
FORMATETC

Other Resources

IDataObjectImpl Members

IDataObjectImpl::SetData

Transfers data from the client to the data object.

```
HRESULT SetData(
   FORMATETC* pformatetc,
   STGMEDIUM* pmedium,
   BOOL fRelease
);
```

Return Value

Returns **E_NOTIMPL**.

Remarks

See IDataObject::SetData in the *Platform SDK*.

See Also

Reference

IDataObjectImpl Class
IDataObjectImpl::GetData
IDataObjectImpl::GetDataHere
IDataObjectImpl::QueryGetData
FORMATETC
STGMEDIUM

Other Resources

IDataObjectImpl Members

IDispatchImpl Class

This class provides a default implementation for **IDispatch** portion of a dual interface.

```
template<
   class T,
   const IID* piid= &__uuidof(T),
   const GUID* plibid = &CAtlModule::m_libid,
   WORD wMajor = 1,
   WORD wMinor = 0,
   class tihclass = CComTypeInfoHolder
>
class ATL_NO_VTABLE IDispatchImpl :
   public T
```

Parameters

T

A dual interface.

piid

A pointer to the IID of *T*.

plibid

A pointer to the LIBID of the type library that contains information about the interface. By default, the server-level type library is passed.

wMajor

The major version of the type library. The default value is 1.

wMinor

The minor version of the type library. The default value is 0.

tihclass

The class used to manage the type information for *T*. The default value is **CComTypeInfoHolder**.

Remarks

IDispatchImpl provides a default implementation for the **IDispatch** portion of any dual interface on your object. A dual interface derives from **IDispatch** and uses only Automation-compatible types. Like a dispinterface, a dual interface supports early and late binding; however, a dual interface differs in that it also supports vtable binding. For more information, see Implementing the IDispatch Interface.

The following example shows a typical implementation of **IDispatchImpl**:

```
class CBeeper :
      public IDispatchImpl< IBeeper, &IID_IBeeper,
         &LIBID_BeeperLib >,
      public CComObjectRoot,
      public CComCoClass< CBeeper, &CLSID_Beeper >
{
   ...
};
```

IDispatchImpl contains a static member of type **CComTypeInfoHolder** that manages the type information for the dual interface. If you have multiple objects implementing the same dual interface, only a single instance of **CComTypeInfoHolder** will be used.

Requirements

Header: atlcom.h

See Also

Other Resources

IDispatchImpl Members

ATL Class Overview

IDispatchImpl Members

Class Methods

IDispatchImpl	The constructor. Calls **AddRef** on the protected member variable that manages the type information for the dual interface. The destructor calls **Release**.

IDispatch Methods

GetIDsOfNames	Maps a set of names to a corresponding set of dispatch identifiers.
GetTypeInfo	Retrieves the type information for the dual interface.
GetTypeInfoCount	Determines whether there is type information available for the dual interface.
Invoke	Provides access to the methods and properties exposed by the dual interface.

See Also

Reference

IDispatchImpl Class

IDispatchImpl Methods

For information about the methods in **IDispatchImpl**, see IDispatchImpl Members.

IDispatchImpl::GetIDsOfNames

Maps a set of names to a corresponding set of dispatch identifiers.

```
STDMETHOD(GetIDsOfNames)(
   REFIID riid,
   LPOLESTR* rgszNames,
   UINT cNames,
   LCID lcid,
   DISPID* rgdispid
);
```

Remarks

See IDispatch::GetIDsOfNames in the *Platform SDK*.

See Also

Reference

IDispatchImpl Class

Other Resources

IDispatchImpl Members

IDispatchImpl::GetTypeInfo

Retrieves the type information for the dual interface.

```
STDMETHOD(GetTypeInfo)(
   UINT itinfo,
   LCID lcid,
   ITypeInfo** pptinfo
);
```

Remarks

See IDispatch::GetTypeInfo in the *Platform SDK*.

See Also

Reference

IDispatchImpl Class

IDispatchImpl::GetTypeInfoCount

Other Resources

IDispatchImpl Members

IDispatchImpl::GetTypeInfoCount

Determines whether there is type information available for the dual interface.

```
STDMETHOD(GetTypeInfoCount)(
   UINT* pctinfo
);
```

Remarks

See IDispatch::GetTypeInfoCount in the *Platform SDK*.

See Also

Reference

IDispatchImpl Class

IDispatchImpl::GetTypeInfo

Other Resources

IDispatchImpl Members

IDispatchImpl::IDispatchImpl

The constructor. Calls **AddRef** on the protected member variable that manages the type information for the dual interface. The destructor calls **Release**.

```
IDispatchImpl( );
```

See Also

Reference

IDispatchImpl Class

Other Resources

IDispatchImpl Members

IDispatchImpl::Invoke

Provides access to the methods and properties exposed by the dual interface.

```
STDMETHOD(Invoke)(
   DISPID dispidMember,
   REFIID riid,
   LCID lcid,
   WORD wFlags,
   DISPPARAMS* pdispparams,
   VARIANT* pvarResult,
   EXCEPINFO* pexcepinfo,
   UINT* puArgErr
);
```

Remarks

See IDispatch::Invoke in the *Platform SDK*.

See Also

Reference

IDispatchImpl Class

Other Resources

IDispatchImpl Members

IDispEventImpl Class

This class provides implementations of the **IDispatch** methods.

```
template <
   UINT nID,
   class T,
   const IID* pdiid = &IID_NULL,
   const GUID* plibid = &GUID_NULL,
   WORD wMajor = 0,
   WORD wMinor = 0,
   class tihclass = CcomTypeInfoHolder
>
class ATL_NO_VTABLE IDispEventImpl :
   public IDispEventSimpleImpl<nID, T, pdiid>
```

Parameters

nID

A unique identifier for the source object. When **IDispEventImpl** is the base class for a composite control, use the resource ID of the desired contained control for this parameter. In other cases, use an arbitrary positive integer.

T

The user's class, which is derived from **IDispEventImpl**.

pdiid

The pointer to the IID of the event dispinterface implemented by this class. This interface must be defined in the type library denoted by *plibid*, *wMajor*, and *wMinor*.

plibid

A pointer to the type library that defines the dispatch interface pointed to by *pdiid*. If **&GUID_NULL**, the type library will be loaded from the object sourcing the events.

wMajor

The major version of the type library. The default value is 0.

wMinor

The minor version of the type library. The default value is 0.

tihclass

The class used to manage the type information for *T*. The default value is a class of type **CComTypeInfoHolder**; however, you can override this template parameter by providing a class of a type other than **CComTypeInfoHolder**.

Remarks

IDispEventImpl provides a way of implementing an event dispinterface without requiring you to supply implementation code for every method/event on that interface. **IDispEventImpl** provides implementations of the **IDispatch** methods. You only need to supply implementations for the events that you are interested in handling.

IDispEventImpl works in conjunction with the event sink map in your class to route events to the appropriate handler function. To use this class:

Add a SINK_ENTRY or SINK_ENTRY_EX macro to the event sink map for each event on each object that you want to handle. When using **IDispEventImpl** as a base class of a composite control, you can call AtlAdviseSinkMap to establish and break the connection with the event sources for all entries in the event sink map. In other cases, or for greater control, call DispEventAdvise to establish the connection between the source object and the base class. Call DispEventUnadvise to break the connection.

You must derive from **IDispEventImpl** (using a unique value for *nID*) for each object for which you need to handle events. You can reuse the base class by unadvising against one source object then advising against a different source object, but the maximum number of source objects that can be handled by a single object at one time is limited by the number of

IDispEventImpl base classes.

IDispEventImpl provides the same functionality as IDispEventSimpleImpl, except it gets type information about the interface from a type library rather than having it supplied as a pointer to an _ATL_FUNC_INFO structure. Use **IDispEventSimpleImpl** when you do not have a type library describing the event interface or want to avoid the overhead associated with using the type library.

Note
IDispEventImpl and **IDispEventSimpleImpl** provide their own implementation of **IUnknown::QueryInterface** enabling each **IDispEventImpl** and **IDispEventSimpleImpl** base class to act as a separate COM identity while still allowing direct access to class members in your main COM object.

CE ATL implementation of ActiveX event sinks only supports return values of type HRESULT or void from your event handler methods; any other return value is unsupported and its behavior is undefined.

For more information, see Supporting IDispEventImpl.

Requirements

Header: atlcom.h

See Also

Reference

_ATL_FUNC_INFO Structure
IDispatchImpl Class
IDispEventSimpleImpl Class
SINK_ENTRY
SINK_ENTRY_EX
SINK_ENTRY_INFO

Other Resources

IDispEventImpl Members
ATL Class Overview

IDispEventImpl Members

Methods

GetFuncInfoFromId	Locates the function index for the specified dispatch identifier.
GetIDsOfNames	Maps a single member and an optional set of argument names to a corresponding set of integer DISPIDs.
GetTypeInfo	Retrieves the type information for an object.
GetTypeInfoCount	Retrieves the number of type information interfaces.
GetUserDefinedType	Retrieves the basic type of a user-defined type.
IDispEventImpl	The constructor.

Typedefs

tihclass	The class used to manage the type information. By default, **CComTypeInfoHolder**.

See Also

Reference

IDispEventImpl Class

IDispEventImpl Methods

For information about the methods in **IDispEventImpl**, see IDispEventImpl Members.

IDispEventImpl::GetFuncInfoFromId

Locates the function index for the specified dispatch identifier.

```
HRESULT GetFuncInfoFromId(
   const IID& iid,
   DISPID dispidMember,
   LCID lcid,
   _ATL_FUNC_INFO& info
);
```

Parameters

iid

[in] A reference to the ID of the function.

dispidMember

[in] The dispatch ID of the function.

lcid

[in] The locale context of the function ID.

info

[in] The structure indicating how the function is called.

Return Value

A standard **HRESULT** value.

See Also

Reference

IDispEventImpl Class

Concepts

Supporting IDispEventImpl

Other Resources

IDispEventImpl Members

IDispEventImpl::GetIDsOfNames

Maps a single member and an optional set of argument names to a corresponding set of integer DISPIDs, which can be used on subsequent calls to IDispatch::Invoke.

```
STDMETHOD(GetIDsOfNames)(
   REFIID riid,
   LPOLESTR* rgszNames,
   UINT cNames,
   LCID lcid,
   DISPID* rgdispid
);
```

Remarks

See IDispatch::GetIDsOfNames in the *Platform SDK*.

See Also

Reference

IDispEventImpl Class

IDispEventImpl::GetFuncInfoFromId

Concepts

Supporting IDispEventImpl

Other Resources

IDispEventImpl Members

IDispEventImpl::GetTypeInfo

Retrieves the type information for an object, which can then be used to get the type information for an interface.

```
STDMETHOD(GetTypeInfo)(
   UINT itinfo,
   LCID lcid,
   ITypeInfo** pptinfo
);
```

Remarks

See IDispatch::GetTypeInfo in the *Platform SDK*.

See Also

Reference

IDispEventImpl Class

IDispEventImpl::GetTypeInfoCount

Concepts

Supporting IDispEventImpl

Other Resources

IDispEventImpl Members

IDispEventImpl::GetTypeInfoCount

Retrieves the number of type information interfaces that an object provides (either 0 or 1).

```
STDMETHOD(GetTypeInfoCount)(
   UINT* pctinfo
);
```

Remarks

See IDispatch::GetTypeInfoCount in the *Platform SDK*.

See Also

Reference

IDispEventImpl Class

IDispEventImpl::GetTypeInfo

Concepts

Supporting IDispEventImpl

Other Resources

IDispEventImpl Members

IDispEventImpl::GetUserDefinedType

Retrieves the basic type of a user-defined type.

```
VARTYPE GetUserDefinedType(
   ITypeInfo *pTI,
      HREFTYPE hrt
);
```

Parameters

pTI

[in] A pointer to the ITypeInfo interface containing the user-defined type.

hrt

[in] A handle to the type description to be retrieved.

Return Value

The type of variant.

Remarks

See ITypeInfo::GetRefTypeInfo.

See Also

Reference

IDispEventImpl Class

Concepts

Supporting IDispEventImpl

Other Resources

IDispEventImpl Members

IDispEventImpl::IDispEventImpl

The constructor. Stores the values of the class template parameters *plibid*, *pdiid*, *wMajor*, and *wMinor*.

```
IDispEventImpl( );
```

See Also

Reference

IDispEventImpl Class

Concepts

Supporting IDispEventImpl

Other Resources

IDispEventImpl Members

IDispEventImpl Typedefs

For information about the typedefs in **IDispEventImpl**, see IDispEventImpl Members.

IDispEventImpl::tihclass

This typedef is an instance of the class template parameter *tihclass*.

```
typedef tihclass _tihclass;
```

Remarks

By default, the class is **CComTypeInfoHolder**. **CComTypeInfoHolder** manages the type information for the class.

See Also

Reference

IDispEventImpl Class

Concepts

Supporting IDispEventImpl

Other Resources

IDispEventImpl Members

IDispEventSimpleImpl Class

This class provides implementations of the **IDispatch** methods, without getting type information from a type library.

```
template <
   UINT nID,
   class T,
   const IID* pdiid
>
class ATL_NO_VTABLE IDispEventSimpleImpl :
   public _IDispEventLocator<nID, pdiid>
```

Parameters

nID

A unique identifier for the source object. When **IDispEventSimpleImpl** is the base class for a composite control, use the resource ID of the desired contained control for this parameter. In other cases, use an arbitrary positive integer.

T

The user's class, which is derived from **IDispEventSimpleImpl**.

pdiid

The pointer to the IID of the event dispinterface implemented by this class.

Remarks

IDispEventSimpleImpl provides a way of implementing an event dispinterface without requiring you to supply implementation code for every method/event on that interface. **IDispEventSimpleImpl** provides implementations of the **IDispatch** methods. You only need to supply implementations for the events that you are interested in handling.

IDispEventSimpleImpl works in conjunction with the event sink map in your class to route events to the appropriate handler function. To use this class:

- Add a SINK_ENTRY_INFO macro to the event sink map for each event on each object that you want to handle.
- Supply type information for each event by passing a pointer to an _ATL_FUNC_INFO structure as a parameter to each entry. On the x86 platform, the `_ATL_FUNC_INFO.cc` value must be CC_CDECL with the callback function calling method of __stdcall. (Smart Device Developers, see Smart Device Developer Notes below.)
- Call DispEventAdvise to establish the connection between the source object and the base class.
- Call DispEventUnadvise to break the connection.

You must derive from **IDispEventSimpleImpl** (using a unique value for *nID*) for each object for which you need to handle events. You can reuse the base class by unadvising against one source object then advising against a different source object, but the maximum number of source objects that can be handled by a single object at one time is limited by the number of **IDispEventSimpleImpl** base classes.

IDispEventSimplImpl provides the same functionality as IDispEventImpl, except it does not get type information about the interface from a type library. The wizards generate code based only on **IDispEventImpl**, but you can use **IDispEventSimpleImpl** by adding the code by hand. Use **IDispEventSimpleImpl** when you don't have a type library describing the event interface or want to avoid the overhead associated with using the type library.

Note
IDispEventImpl and **IDispEventSimpleImpl** provide their own implementation of **IUnknown::QueryInterface** enabling each **IDispEventImpl** or **IDispEventSimpleImpl** base class to act as a separate COM identity while still allowing direct access to class members in your main COM object.

CE ATL implementation of ActiveX event sinks only supports return values of type HRESULT or void from your event handler methods; any other return value is unsupported and its behavior is undefined.

For more information, see Supporting IDispEventImpl.

Requirements

Header: atlcom.h

Smart Device Developer Notes

IDispEventSimpleImpl requires the `_ATL_FUNC_INFO.cc` member to be CC_STDCALL, but CE only supports CC_CDECL. Although this is an apparent contradiction, for Pocket PC, the callback works regardless of the `_ATL_FUNC_INFO.cc` value or the callback method's calling convention. The following example demonstrates a sample event handler for Pocket PC:

```
_ATL_FUNC_INFO Event1Info1 = { CC_CDECL, VT_EMPTY, 1, { VT_I4 } };

class CEventHandler :
public IDispEventSimpleImpl<1234, CEventHandler, &__uuidof(_ISdeLibObj1Events)>
{
public:
BEGIN_SINK_MAP(CEventHandler)
SINK_ENTRY_INFO(1234, __uuidof(_ISdeLibObj1Events), 1, OnEvent1, &Event1Info1)
END_SINK_MAP()
void __stdcall  OnEvent1(LONG l)
{
//ATLASSERT(l == 445533);
if (l != 445533)
OutputDebugString(L"l is not 445533\n");
}
HRESULT Advise1234(IUnknown * punk) {
return IDispEventSimpleImpl<1234, CEventHandler, &__uuidof(_ISdeLibObj1Events)>::DispEventA
dvise(punk);
}
};
```

See Also

Reference

_ATL_FUNC_INFO Structure
IDispatchImpl Class
IDispEventImpl Class
SINK_ENTRY_INFO

Other Resources

IDispEventSimpleImpl Members
ATL Class Overview

IDispEventSimpleImpl Members

Class Methods

Advise	Establishes a connection with the default event source.
DispEventAdvise	Establishes a connection with the event source.
DispEventUnadvise	Breaks the connection with the event source.
Unadvise	Breaks the connection with the default event source.

IDispatch Methods

GetIDsOfNames	Returns **E_NOTIMPL**.
GetTypeInfo	Returns **E_NOTIMPL**.
GetTypeInfoCount	Returns **E_NOTIMPL**.
Invoke	Calls the event handlers listed in the event sink map.

See Also

Reference

IDispEventSimpleImpl Class

IDispEventSimpleImpl Methods

For information about the methods in **IDispEventSimpleImpl**, see IDispEventSimpleImpl Members.

IDispEventSimpleImpl::Advise

Call this method to establish a connection with the event source represented by *pUnk*.

```
HRESULT Advise(
   IUnknown* pUnk
);
```

Parameters

pUnk

[in] A pointer to the **IUnknown** interface of the event source object.

Return Value

S_OK or any failure **HRESULT** value.

Remarks

Once the connection is established, events fired from *pUnk* will be routed to handlers in your class by way of the event sink map.

Note
If your class derives from multiple **IDispEventSimpleImpl** classes, you will need to disambiguate calls to this method by scoping the call with the particular base class you are interested in.

Advise establishes a connection with the default event source, it gets the IID of the default event source of the object as determined by AtlGetObjectSourceInterface.

See Also

Reference

IDispEventSimpleImpl Class

IDispEventSimpleImpl::DispEventAdvise

Concepts

Supporting IDispEventImpl

Other Resources

IDispEventSimpleImpl Members

IDispEventSimpleImpl::DispEventAdvise

Call this method to establish a connection with the event source represented by *pUnk*.

```
HRESULT DispEventAdvise(
   IUnknown* pUnk
      const IID* piid
);
```

Parameters

pUnk

[in] A pointer to the **IUnknown** interface of the event source object.

piid

A pointer to the IID of the event source object.

Return Value

S_OK or any failure **HRESULT** value.

Remarks

Subsequently, events fired from *pUnk* will be routed to handlers in your class by way of the event sink map.

Note
If your class derives from multiple **IDispEventSimpleImpl** classes, you will need to disambiguate calls to this method by sc oping the call with the particular base class you are interested in.

DispEventAdvise establishes a connection with the event source specified in *pdiid*.

See Also

Reference

IDispEventSimpleImpl Class

Concepts

Supporting IDispEventImpl

Other Resources

IDispEventSimpleImpl Members

IDispEventSimpleImpl::DispEventUnadvise

Breaks the connection with the event source represented by *pUnk*.

```
HRESULT DispEventUnadvise(
   IUnknown* pUnk
      const IID* piid
);
```

Parameters

pUnk

[in] A pointer to the **IUnknown** interface of the event source object.

piid

A pointer to the IID of the event source object.

Return Value

S_OK or any failure **HRESULT** value.

Remarks

Once the connection is broken, events will no longer be routed to the handler functions listed in the event sink map.

Note
If your class derives from multiple **IDispEventSimpleImpl** classes, you will need to disambiguate calls to this method by sc oping the call with the particular base class you are interested in.

DispEventAdvise breaks a connection that was established with the event source specified in *pdiid*.

See Also

Reference

IDispEventSimpleImpl Class

Concepts

Supporting IDispEventImpl

Other Resources

IDispEventSimpleImpl Members

IDispEventSimpleImpl::GetIDsOfNames

This implementation of **IDispatch::GetIDsOfNames** returns **E_NOTIMPL**.

```
STDMETHOD(GetIDsOfNames)(
   REFIID /* riid */,
   LPOLESTR* /* rgszNames */,
   UINT /* cNames */,
   LCID /* lcid */,
   DISPID* /* rgdispid */
);
```

Remarks

See IDispatch::GetIDsOfNames in the *Platform SDK*.

See Also

Reference

IDispEventSimpleImpl Class

Other Resources

IDispEventSimpleImpl Members

IDispEventSimpleImpl::GetTypeInfo

This implementation of **IDispatch::GetTypeInfo** returns **E_NOTIMPL**.

```
STDMETHOD(GetTypeInfo)(
   UINT /* itinfo */,
   LCID /* lcid */,
   ITypeInfo** /* pptinfo */
);
```

Remarks

See IDispatch::GetTypeInfo in the *Platform SDK*.

See Also

Reference

IDispEventSimpleImpl Class

Other Resources

IDispEventSimpleImpl Members

IDispEventSimpleImpl::GetTypeInfoCount

This implementation of **IDispatch::GetTypeInfoCount** returns **E_NOTIMPL**.

```
STDMETHOD(GetTypeInfoCount)(
   UINT* /* pctinfo */
);
```

Remarks

See IDispatch::GetTypeInfoCount in the *Platform SDK*.

See Also

Reference

IDispEventSimpleImpl Class

Other Resources

IDispEventSimpleImpl Members

IDispEventSimpleImpl::Invoke

This implementation of **IDispatch::Invoke** calls the event handlers listed in the event sink map.

```
STDMETHOD(Invoke)(
   DISPID dispidMember,
   REFIID /* riid */,
   LCID lcid,
   WORD /* wFlags */,
   DISPPARMS* pdispparams,
   VARIANT* pvarResult,
   EXCEPINFO* /* pexcepinfo */,
   UINT* /* puArgErr */
);
```

Remarks

See IDispatch::Invoke.

See Also

Reference

IDispEventSimpleImpl Class

Other Resources

IDispEventSimpleImpl Members

IDispEventSimpleImpl::Unadvise

Breaks the connection with the event source represented by *pUnk*.

```
HRESULT Unadvise(
   IUnknown* pUnk
);
```

Parameters

pUnk

[in] A pointer to the **IUnknown** interface of the event source object.

Return Value

S_OK or any failure **HRESULT** value.

Remarks

Once the connection is broken, events will no longer be routed to the handler functions listed in the event sink map.

Note
If your class derives from multiple **IDispEventSimpleImpl** classes, you will need to disambiguate calls to this method by sc oping the call with the particular base class you are interested in.

Unadvise breaks a connection that was established with the default event source specified in *pdiid*.

Unavise breaks a connection with the default event source, it gets the IID of the default event source of the object as determined by AtlGetObjectSourceInterface.

See Also

Reference

IDispEventSimpleImpl Class

IDispEventSimpleImpl::DispEventAdvise

Concepts

Supporting IDispEventImpl

Other Resources

IDispEventSimpleImpl Members

IDocHostUIHandlerDispatch Interface

An interface to the Microsoft HTML parsing and rendering engine.

```
interface IDocHostUIHandlerDispatch : IDispatch
```

Remarks

A host can replace the menus, toolbars, and context menus used by Microsoft's HTML parsing and rendering engine (MSHTML) by implementing this interface.

Requirements

The definition of this interface is available as IDL or C++, as shown below.

Definition type	File
IDL	ATLIFace.idl
C++	ATLIFace.h (also included in ATLBase.h)

See Also

Reference

IDocUIHostHandler

Other Resources

IDocHostUIHandlerDispatch Members

IDocHostUIHandlerDispatch Members

Note
The links in the following table are to the INet SDK reference topics for the members of the IDocUIHostHandler interface. **IDocHostUIHandlerDispatch** has the same functionality as **IDocUIHostHandler**, with the difference being that **IDocHostUIHandlerDispatch** is a dispinterface whereas **IDocUIHostHandler** is a custom interface.

Methods

EnableModeless	Called from MSHTML implementation of IOleInPlaceActiveObject::EnableModeless. Also called when MSHTML displays modal UI.
FilterDataObject	Called on the host by MSHTML to allow the host to replace MSHTML's data object.
GetDropTarget	Called by MSHTML when it is being used as a drop target to allow the host to supply an alternative IDropTarget.
GetExternal	Called by MSHTML to obtain the host's IDispatch interface.
GetHostInfo	Retrieves the UI capabilities of MSHTML host.
GetOptionKeyPath	Returns the registry key under which MSHTML stores user preferences.
HideUI	Called when MSHTML removes its menus and toolbars.
OnDocWindowActivate	Called from MSHTML implementation of IOleInPlaceActiveObject::OnDocWindowActivate.
OnFrameWindowActivate	Called from MSHTML implementation of IOleInPlaceActiveObject::OnFrameWindowActivate.
ResizeBorder	Called from MSHTML implementation of IOleInPlaceActiveObject::ResizeBorder.
ShowContextMenu	Called from MSHTML to display a context menu.
ShowUI	Allows the host to replace MSHTML menus and toolbars.
TranslateAccelerator	Called by MSHTML when IOleInPlaceActiveObject::TranslateAccelerator or IOleControlSite::TranslateAccelerator is called.
TranslateUrl	Called by MSHTML to allow the host an opportunity to modify the URL to be loaded.
UpdateUI	Notifies the host that the command state has changed.

See Also

Reference

IDocHostUIHandlerDispatch Interface

IEnumOnSTLImpl Class

This class defines an enumerator interface based on an STL collection.

```
template <
   class Base,
   const IID* piid,
   class T,
   class Copy,
   class CollType
>
class ATL_NO_VTABLE IEnumOnSTLImpl :
   public Base
```

Parameters

Base

A COM enumerator (IEnumXXXX) interface.

piid

A pointer to the interface ID of the enumerator interface.

T

The type of item exposed by the enumerator interface.

Copy

A copy policy class.

CollType

An STL container class.

Remarks

IEnumOnSTLImpl provides the implementation for a COM enumerator interface where the items being enumerated are stored in an STL-compatible container. This class is analogous to the CComEnumImpl class, which provides an implementation for an enumerator interface based on an array.

Note
See CComEnumImpl::Init for details on further differences between **CComEnumImpl** and **IEnumOnSTLImpl**.

Typically, you will *not* need to create your own enumerator class by deriving from this interface implementation. If you want to use an ATL-supplied enumerator based on an STL container, it is more common to create an instance of CComEnumOnSTL, or to create a collection class that returns an enumerator by deriving from ICollectionOnSTLImpl.

However, if you do need to provide a custom enumerator (for example, one that exposes interfaces in addition to the enumerator interface), you can derive from this class. In this situation it is likely that you'll need to override the Clone method to provide your own implementation.

Requirements

Header: atlcom.h

See Also

Other Resources

IEnumOnSTLImpl Members

ATL Class Overview

IEnumOnSTLImpl Members

Methods

Clone	The implementation of IEnumXXXX::Clone.
Init	Initializes the enumerator.
Next	The implementation of IEnumXXXX::Next.
Reset	The implementation of IEnumXXXX::Reset.
Skip	The implementation of IEnumXXXX::Skip.

Data Members

m_iter	The iterator that represents the enumerator's current position within the collection.
m_pcollection	A pointer to the STL container holding the items to be enumerated.
m_spUnk	The **IUnknown** pointer of the object supplying the collection.

See Also

Reference

IEnumOnSTLImpl Class

IEnumOnSTLImpl Methods

For information about the methods in **IEnumOnSTLImpl**, see IEnumOnSTLImpl Members.

IEnumOnSTLImpl::Init

Initializes the enumerator.

```
HRESULT Init(
   IUnknown* pUnkForRelease,
   CollType& collection
);
```

Parameters

pUnkForRelease

[in] The **IUnknown** pointer of an object that must be kept alive during the lifetime of the enumerator. Pass **NULL** if no such object exists.

collection

A reference to the STL container that holds the items to be enumerated.

Return Value

A standard **HRESULT** value.

Remarks

If you pass **Init** a reference to a collection held in another object, you can use the *pUnkForRelease* parameter to ensure that the object, and the collection it holds, is available for as long as the enumerator needs it.

You must call this method before passing a pointer to the enumerator interface back to any clients.

See Also

Reference

IEnumOnSTLImpl Class

Other Resources

IEnumOnSTLImpl Members

IEnumOnSTLImpl::Clone

This method provides the implementation of the IEnumXXXX::Clone method by creating an object of type **CComEnumOnSTL**, initializing it with the same collection and iterator used by the current object, and returning the interface on the newly created object.

```
STDMETHOD(Clone)(
   Base** ppEnum
);
```

Parameters

ppEnum

[out] The enumerator interface on a newly created object cloned from the current enumerator.

Return Value

A standard **HRESULT** value.

See Also

Reference

IEnumOnSTLImpl Class

IEnumOnSTLImpl::Init

Other Resources

IEnumOnSTLImpl Members

IEnumOnSTLImpl::Next

This method provides the implementation of the IEnumXXXX::Next method.

```
STDMETHOD(Next)(
   ULONG celt,
   T* rgelt,
   ULONG* pceltFetched
);
```

Parameters

celt

[in] The number of elements requested.

rgelt

[out] The array to be filled in with the elements.

pceltFetched

[out] The number of elements actually returned in *rgelt*. This can be less than *celt* if fewer than *celt* elements remain in the list.

Return Value

A standard **HRESULT** value.

See Also

Reference

IEnumOnSTLImpl Class

Other Resources

IEnumOnSTLImpl Members

IEnumOnSTLImpl::Reset

This method provides the implementation of the IEnumXXXX::Reset method.

```
STDMETHOD(Reset)(
   void
);
```

Return Value

A standard **HRESULT** value.

See Also

Reference

IEnumOnSTLImpl Class

Other Resources

IEnumOnSTLImpl Members

IEnumOnSTLImpl::Skip

This method provides the implementation of the IEnumXXXX::Skip method.

```
STDMETHOD(Skip)(
   ULONG celt
);
```

Parameters

celt

[in] The number of elements to skip.

Return Value

A standard **HRESULT** value.

See Also

Reference

IEnumOnSTLImpl Class

Other Resources

IEnumOnSTLImpl Members

IEnumOnSTLImpl Data Members

For information about the data members in **IEnumOnSTLImpl**, see IEnumOnSTLImpl Members.

IEnumOnSTLImpl::m_spUnk

The **IUnknown** pointer of the object supplying the collection.

```
CComPtr<IUnknown> m_spUnk;
```

Remarks

This smart pointer maintains a reference on the object passed to IEnumOnSTLImpl::Init, ensuring that it remains alive during the lifetime of the enumerator.

See Also

Reference

IEnumOnSTLImpl Class

IEnumOnSTLImpl::Init

Other Resources

IEnumOnSTLImpl Members

IEnumOnSTLImpl::m_pcollection

This member points to the collection that provides the data driving the implementation of the enumerator interface.

```
CollType* m_pcollection;
```

Remarks

This member is initialized by a call to IEnumOnSTLImpl::Init.

See Also

Reference

IEnumOnSTLImpl Class

IEnumOnSTLImpl::Init

Other Resources

IEnumOnSTLImpl Members

IEnumOnSTLImpl::m_iter

This member holds the iterator used to mark the current position within the collection and navigate to subsequent elements.

```
CollType::iterator m_iter;
```

See Also

Reference

IEnumOnSTLImpl Class

Other Resources

IEnumOnSTLImpl Members

IObjectSafetyImpl Class

This class provides a default implementation of the **IObjectSafety** interface to allow a client to retrieve and set an object's safety levels.

```
template <class T, DWORD dwSupportedSafety>
class IObjectSafetyImpl
```

Parameters

T

Your class, derived from **IObjectSafetyImpl**.

dwSupportedSafety

Specifies the supported safety options for the control. Can be one of the following values:

- **INTERFACESAFE_FOR_UNTRUSTED_CALLER** The interface identified by the SetInterfaceSafetyOptions parameter *riid* should be made safe for scripting.
- **INTERFACESAFE_FOR_UNTRUSTED_DATA** The interface identified by the **SetInterfaceSafetyOptions** parameter *riid* should be made safe for untrusted data during initialization.

Remarks

Class **IObjectSafetyImpl** provides a default implementation of **IObjectSafety**. The **IObjectSafety** interface allows a client to retrieve and set an object's safety levels. For example, a web browser can call **IObjectSafety::SetInterfaceSafetyOptions** to make a control safe for initialization or safe for scripting.

Note that using the IMPLEMENTED_CATEGORY macro with the **CATID_SafeForScripting** and **CATID_SafeForInitializing** component categories provides an alternative way of specifying that a component is safe.

Related Articles ATL Tutorial, Creating an ATL Project

Requirements

Header: atlctl.h

See Also

Reference

IObjectSafety Interface

Other Resources

IObjectSafetyImpl Members

ATL Class Overview

IObjectSafetyImpl Members

IObjectSafety Methods

GetInterfaceSafetyOptions	Retrieves the safety options supported by the object, as well as the safety options currently set for the object.
SetInterfaceSafetyOptions	Makes the object safe for initialization or scripting.

Data Members

m_dwCurrentSafety	Stores the object's current safety level.

See Also

Reference

IObjectSafetyImpl Class

IObjectSafetyImpl Methods

For information about the methods in **IObjectSafetyImpl**, see IObjectSafetyImpl Members.

IObjectSafetyImpl::GetInterfaceSafetyOptions

Retrieves the safety options supported by the object, as well as the safety options currently set for the object.

```
HRESULT GetInterfaceSafetyOptions(
   REFIID riid,
   DWORD* pdwSupportedOptions,
   DWORD* pdwEnabledOptions
);
```

Remarks

The implementation returns the appropriate values for any interface supported by the object's implementation of **IUnknown::QueryInterface**.

Security Note
Any object that supports **IObjectSafety** is responsible for its own security, and that of any object it delegates. The programmer must take into account issues arising from running code in the user's context, cross-site scripting and perform suitable zone checking.

See IObjectSafety::GetInterfaceSafetyOptions in the *Platform SDK*.

See Also

Reference

IObjectSafetyImpl Class

IObjectSafetyImpl::SetInterfaceSafetyOptions

Other Resources

IObjectSafetyImpl Members

IObjectSafetyImpl::SetInterfaceSafetyOptions

Makes the object safe for initialization or scripting by setting the m_dwCurrentSafety member to the appropriate value.

```
HRESULT SetInterfaceSafetyOptions(
   REFIID riid,
   DWORD dwOptionsSetMask,
   DWORD dwEnabledOptions
);
```

Remarks

The implementation returns **E_NOINTERFACE** for any interface not supported by the object's implementation of **IUnknown::QueryInterface**.

Security Note
Any object that supports **IObjectSafety** is responsible for its own security, and that of any object it delegates. The programmer must take into account issues arising from running code in the user's context, cross-site scripting and perform suitable zone checking.

See IObjectSafety::SetInterfaceSafetyOptions in the *Platform SDK*.

See Also

Reference

IObjectSafetyImpl Class

IObjectSafetyImpl::GetInterfaceSafetyOptions

Other Resources

IObjectSafetyImpl Members

IObjectSafetyImpl Data Members

For information about the data members in **IObjectSafetyImpl**, see IObjectSafetyImpl Members.

IObjectSafetyImpl::m_dwCurrentSafety

Stores the object's current safety level.

```
DWORD m_dwCurrentSafety;
```

See Also

Reference

IObjectSafetyImpl Class

Other Resources

IObjectSafetyImpl Members

IObjectWithSiteImpl Class

This class provides methods allowing an object to communicate with its site.

```
template<
   class T
>
class ATL_NO_VTABLE IObjectWithSiteImpl :
   public IObjectWithSite
```

Parameters

T

Your class, derived from **IObjectWithSiteImpl**.

Remarks

The IObjectWithSite interface allows an object to communicate with its site. Class **IObjectWithSiteImpl** provides a default implementation of this interface and implements **IUnknown** by sending information to the dump device in debug builds.

IObjectWithSiteImpl specifies two methods. The client first calls **SetSite**, passing the site's **IUnknown** pointer. This pointer is stored within the object, and can later be retrieved through a call to **GetSite**.

Typically, you derive your class from **IObjectWithSiteImpl** when you are creating an object that is not a control. For controls, derive your class from IOleObjectImpl, which also provides a site pointer. Do not derive your class from both **IObjectWithSiteImpl** and **IOleObjectImpl**.

Requirements

Header: atlcom.h

See Also

Other Resources

IObjectWithSiteImpl Members

ATL Class Overview

IObjectWithSiteImpl Members

IObjectWithSite Methods

GetSite	Queries the site for an interface pointer.
SetChildSite	Provides the object with the site's **IUnknown** pointer.
SetSite	Provides the object with the site's **IUnknown** pointer.

Data Members

m_spUnkSite	Manages the site's **IUnknown** pointer.

See Also

Reference

IObjectWithSiteImpl Class

IObjectWithSiteImpl Methods

For information about the methods in **IObjectWithSiteImpl**, see IObjectWithSiteImpl Members.

IObjectWithSiteImpl::GetSite

Queries the site for a pointer to the interface identified by *riid*.

```
STDMETHOD(GetSite)(
   REFIID riid,
   void **ppvSite
);
```

Remarks

If the site supports this interface, the pointer is returned via *ppvSite*. Otherwise, *ppvSite* is set to **NULL**.

See IObjectWithSite::GetSite in the *Platform SDK*.

See Also

Reference

IObjectWithSiteImpl Class

IObjectWithSiteImpl::SetSite

Other Resources

IObjectWithSiteImpl Members

IObjectWithSiteImpl::SetChildSite

Provides the object with the site's **IUnknown** pointer.

```
HRESULT SetChildSite(
   IUnknown* pUnkSite
);
```

Parameters

pUnkSite

[in] Pointer to the **IUnknown** interface pointer of the site managing this object. If NULL, the object should call **IUnknown::Release** on any existing site at which point the object no longer knows its site.

Return Value

Returns **S_OK**.

See Also

Reference

IObjectWithSiteImpl Class

IObjectWithSiteImpl::GetSite

Other Resources

IObjectWithSiteImpl Members

IObjectWithSiteImpl::SetSite

Provides the object with the site's **IUnknown** pointer.

```
STDMETHOD(SetSite)(
   IUnknown* pUnkSite
);
```

Remarks

See IObjectWithSite::SetSite in the *Platform SDK*.

See Also

Reference

IObjectWithSiteImpl Class

IObjectWithSiteImpl::GetSite

Other Resources

IObjectWithSiteImpl Members

IObjectWithSiteImpl Data Members

For information about the data members in **IObjectWithSiteImpl**, see IObjectWithSiteImpl Members.

IObjectWithSiteImpl::m_spUnkSite

Manages the site's **IUnknown** pointer.

```
CComPtr< IUnknown > m_spUnkSite;
```

Remarks

m_spUnkSite initially receives this pointer through a call to SetSite.

See Also

Reference

IObjectWithSiteImpl Class

CComPtr Class

Other Resources

IObjectWithSiteImpl Members

IOleControlImpl Class

This class provides a default implementation of the **IOleControl** interface and implements **IUnknown**.

```
template< class T >
class IOleControlImpl
```

Parameters

T

Your class, derived from **IOleControlImpl**.

Remarks

Class **IOleControlImpl** provides a default implementation of the IOleControl interface and implements **IUnknown** by sending information to the dump device in debug builds.

Related Articles ATL Tutorial, Creating an ATL Project

Requirements

Header: atlctl.h

See Also

Reference

IOleObjectImpl Class

ActiveX Controls Interfaces

Other Resources

IOleControlImpl Members

ATL Class Overview

IOleControlImpl Members

IOleControl Methods

FreezeEvents	Indicates whether or not the container ignores or accepts events from the control.
GetControlInfo	Fills in information about the control's keyboard behavior. The ATL implementation returns **E_NOTIMPL**.
OnAmbientPropertyChange	Informs a control that one or more of the container's ambient properties has changed. The ATL implementation returns **S_OK**.
OnMnemonic	Informs the control that a user has pressed a specified keystroke. The ATL implementation returns **E_NOTIMPL**.

See Also

Reference

IOleControlImpl Class

IOleControlImpl Methods

For information about the methods in **IOleControlImpl**, see IOleControlImpl Members.

IOleControlImpl::FreezeEvents

In ATL's implementation, **FreezeEvents** increments the control class's **m_nFreezeEvents** data member if *bFreeze* is **TRUE**, and decrements **m_nFreezeEvents** if *bFreeze* is **FALSE**.

```
HRESULT FreezeEvents(
   BOOL bFreeze
);
```

Remarks

FreezeEvents then returns **S_OK**.

See IOleControl::FreezeEvents in the *Platform SDK*.

See Also

Reference

IOleControlImpl Class

CComControlBase::m_nFreezeEvents

Other Resources

IOleControlImpl Members

IOleControlImpl::GetControlInfo

Fills in information about the control's keyboard behavior.

```
HRESULT GetControlInfo(
   LPCONTROLINFO pCI
);
```

Remarks

See IOleControl:GetControlInfo in the *Platform SDK*.

Return Value

Returns **E_NOTIMPL**.

See Also

Reference

IOleControlImpl Class

Other Resources

IOleControlImpl Members

IOleControlImpl::OnAmbientPropertyChange

Informs a control that one or more of the container's ambient properties has changed.

```
HRESULT OnAmbientPropertyChange(
   DISPID dispid
);
```

Return Value

Returns **S_OK**.

Remarks

See IOleControl::OnAmbientPropertyChange in the *Platform SDK*.

See Also

Reference

IOleControlImpl Class

Other Resources

IOleControlImpl Members

IOleControlImpl::OnMnemonic

Informs the control that a user has pressed a specified keystroke.

```
HRESULT OnMnemonic(
   LPMSG pMsg
);
```

Return Value

Returns **E_NOTIMPL**.

Remarks

See IOleControl::OnMnemonic in the *Platform SDK*.

See Also

Reference

IOleControlImpl Class

Other Resources

IOleControlImpl Members

IOleInPlaceActiveObjectImpl Class

This class provides methods for assisting communication between an in-place control and its container.

```
template< class T >
class IOleInPlaceActiveObjectImpl
```

Parameters

T

Your class, derived from **IOleInPlaceActiveObjectImpl**.

Remarks

The IOleInPlaceActiveObject interface assists communication between an in-place control and its container; for example, communicating the active state of the control and container, and informing the control it needs to resize itself. Class **IOleInPlaceActiveObjectImpl** provides a default implementation of **IOleInPlaceActiveObject** and supports **IUnknown** by sending information to the dump device in debug builds.

Related Articles ATL Tutorial, Creating an ATL Project

Requirements

Header: atlctl.h

See Also

Reference

CComControl Class

ActiveX Controls Interfaces

Other Resources

IOleInPlaceActiveObjectImpl Members

ATL Class Overview

IOleInPlaceActiveObjectImpl Members

IOleWindow Methods

ContextSensitiveHelp	Enables context-sensitive help. The ATL implementation returns **E_NOTIMPL**.
GetWindow	Gets a window handle.

IOleInPlaceActiveObject Methods

EnableModeless	Enables modeless dialog boxes. The ATL implementation returns **S_OK**.
OnDocWindowActivate	Notifies the control when the container's document window is activated or deactivated. The ATL implementation returns **S_OK**.
OnFrameWindowActivate	Notifies the control when the container's top-level frame window is activated or deactivated. The ATL implementation returns
ResizeBorder	Informs the control it needs to resize its borders. The ATL implementation returns **S_OK**.
TranslateAccelerator	Processes menu accelerator-key messages from the container. The ATL implementation returns **E_NOTIMPL**.

See Also

Reference

IOleInPlaceActiveObjectImpl Class

IOleInPlaceActiveObjectImpl Methods

For information about the methods in **IOleInPlaceActiveObjectImpl**, see IOleInPlaceActiveObjectImpl Members.

IOleInPlaceActiveObjectImpl::ContextSensitiveHelp

Enables context-sensitive help.

```
HRESULT ContextSensitiveHelp(
   BOOL fEnterMode
);
```

Return Value

Returns **E_NOTIMPL**.

Remarks

See IOleWindow::ContextSensitiveHelp in the *Platform SDK*.

See Also

Reference

IOleInPlaceActiveObjectImpl Class

Other Resources

IOleInPlaceActiveObjectImpl Members

IOleInPlaceActiveObjectImpl::EnableModeless

Enables modeless dialog boxes.

```
HRESULT EnableModeless(
   BOOL fEnable
);
```

Return Value

Returns **S_OK**.

Remarks

See IOleInPlaceActiveObject::EnableModeless in the *Platform SDK*.

See Also

Reference

IOleInPlaceActiveObjectImpl Class

Other Resources

IOleInPlaceActiveObjectImpl Members

IOleInPlaceActiveObjectImpl::GetWindow

The container calls this function to get the window handle of the control.

```
HRESULT GetWindow(
   HWND* phwnd
);
```

Remarks

Some containers will not work with a control that has been windowless, even if it is currently windowed. In ATL's implementation, if the **CComControl::m_bWasOnceWindowless** data member is **TRUE**, the function returns **E_FAIL**. Otherwise, if **phwnd* is not **NULL**, **GetWindow** assigns *phwnd* to the control class's data member **m_hWnd** and returns **S_OK**.

See IOleWindow::GetWindow in the *Platform SDK*.

See Also

Reference

IOleInPlaceActiveObjectImpl Class

CComControlBase::m_bWasOnceWindowless

Other Resources

IOleInPlaceActiveObjectImpl Members

IOleInPlaceActiveObjectImpl::OnDocWindowActivate

Notifies the control when the container's document window is activated or deactivated.

```
HRESULT OnDocWindowActivate(
   BOOL fActivate
);
```

Return Value

Returns **S_OK**.

Remarks

See IOleInPlaceActiveObject::OnDocWindowActivate in the *Platform SDK*.

See Also

Reference

IOleInPlaceActiveObjectImpl Class

Other Resources

IOleInPlaceActiveObjectImpl Members

IOleInPlaceActiveObjectImpl::OnFrameWindowActivate

Notifies the control when the container's top-level frame window is activated or deactivated.

```
HRESULT OnFrameWindowActivate(
   BOOL fActivate
);
```

Return Value

Returns **S_OK**.

Remarks

See IOleInPlaceActiveObject::OnFrameWindowActivate in the *Platform SDK*.

See Also

Reference

IOleInPlaceActiveObjectImpl Class

Other Resources

IOleInPlaceActiveObjectImpl Members

IOleInPlaceActiveObjectImpl::ResizeBorder

Informs the control it needs to resize its borders.

```
HRESULT ResizeBorder(
   LPRECT prcBorder,
   IOleInPlaceUIWindow* pUIWindow,
   BOOL fFrameWindow
);
```

Return Value

Returns **S_OK**.

Remarks

See IOleInPlaceActiveObject::ResizeBorder in the *Platform SDK*.

See Also

Reference

IOleInPlaceActiveObjectImpl Class

Other Resources

IOleInPlaceActiveObjectImpl Members

IOleInPlaceActiveObjectImpl::TranslateAccelerator

Processes menu accelerator-key messages from the container.

```
HRESULT TranslateAccelerator(
   LPMSG lpmsg
);
```

Return Value

This method supports the following return values:

S_OK if the message was translated successfully.

S_FALSE if the message was not translated.

Remarks

See IOleInPlaceActiveObject::TranslateAccelerator in the *Platform SDK*.

See Also

Reference

IOleInPlaceActiveObjectImpl Class

Other Resources

IOleInPlaceActiveObjectImpl Members

IOleInPlaceObjectWindowlessImpl Class

This class implements **IUnknown** and provides methods that enable a windowless control to receive window messages and to participate in drag-and-drop operations.

```
template< class T >
class IOleInPlaceObjectWindowlessImpl
```

Parameters

T

Your class, derived from **IOleInPlaceObjectWindowlessImpl**.

Remarks

The IOleInPlaceObject interface manages the reactivation and deactivation of in-place controls and determines how much of the control should be visible. The IOleInPlaceObjectWindowless interface enables a windowless control to receive window messages and to participate in drag-and-drop operations. Class **IOleInPlaceObjectWindowlessImpl** provides a default implementation of **IOleInPlaceObject** and **IOleInPlaceObjectWindowless** and implements **IUnknown** by sending information to the dump device in debug builds.

Related Articles ATL Tutorial, Creating an ATL Project

Requirements

Header: atlctl.h

See Also

Reference

CComControl Class

Other Resources

IOleInPlaceObjectWindowlessImpl Members

ATL Class Overview

IOleInPlaceObjectWindowlessImpl Members

IOleWindow Methods

ContextSensitiveHelp	Enables context-sensitive help. The ATL implementation returns **E_NOTIMPL**.
GetWindow	Gets a window handle.

IOleInPlaceObject Methods

InPlaceDeactivate	Deactivates an active in-place control.
ReactivateAndUndo	Reactivates a previously deactivated control. The ATL implementation returns **E_NOTIMPL**.
SetObjectRects	Indicates what part of the in-place control is visible.
UIDeactivate	Deactivates and removes the user interface that supports in-place activation.

IOleInPlaceObjectWindowless Methods

GetDropTarget	Supplies the **IDropTarget** interface for an in-place active, windowless object that supports drag and drop. The ATL implementation returns **E_NOTIMPL**.
OnWindowMessage	Dispatches a message from the container to a windowless control that is in-place active.

See Also

Reference

IOleInPlaceObjectWindowlessImpl Class

IOleInPlaceObjectWindowlessImpl Methods

For information about the methods in **IOleInPlaceObjectWindowlessImpl**, see IOleInPlaceObjectWindowlessImpl Members.

IOleInPlaceObjectWindowlessImpl::ContextSensitiveHelp

Returns **E_NOTIMPL**.

```
HRESULT ContextSensitiveHelp(
   BOOL fEnterMode
);
```

Remarks

See IOleWindow::ContextSensitiveHelp in the *Platform SDK*.

See Also

Reference

IOleInPlaceObjectWindowlessImpl Class

Other Resources

IOleInPlaceObjectWindowlessImpl Members

IOleInPlaceObjectWindowlessImpl::GetDropTarget

Returns **E_NOTIMPL**.

```
HRESULT GetDropTarget(
   IDropTarget** ppDropTarget
);
```

Remarks

See IOleInPlaceObjectWindowless::GetDropTarget in the *Platform SDK*.

See Also

Reference

IOleInPlaceObjectWindowlessImpl Class

Other Resources

IOleInPlaceObjectWindowlessImpl Members

IOleInPlaceObjectWindowlessImpl::GetWindow

The container calls this function to get the window handle of the control.

```
HRESULT GetWindow(
   HWND* phwnd
);
```

Remarks

Some containers will not work with a control that has been windowless, even if it is currently windowed. In ATL's implementation, if the control class's data member **m_bWasOnceWindowless** is **TRUE**, the function returns **E_FAIL**. Otherwise, if *phwnd* is not **NULL**, **GetWindow** sets **phwnd* to the control class's data member **m_hWnd** and returns **S_OK**.

See IOleWindow::GetWindow in the *Platform SDK*.

See Also

Reference

IOleInPlaceObjectWindowlessImpl Class

CComControlBase::m_bWasOnceWindowless

Other Resources

IOleInPlaceObjectWindowlessImpl Members

IOleInPlaceObjectWindowlessImpl::InPlaceDeactivate

Called by the container to deactivate an in-place active control.

```
HRESULT InPlaceDeactivate(
   HWND* phwnd
);
```

Remarks

This method performs a full or partial deactivation depending on the state of the control. If necessary, the control's user interface is deactivated, and the control's window, if any, is destroyed. The container is notified that the control is no longer active in place. The **IOleInPlaceUIWindow** interface used by the container to negotiate menus and border space is released.

See IOleInPlaceObject::InPlaceDeactivate in the *Platform SDK*.

See Also

Reference

IOleInPlaceObjectWindowlessImpl Class

CComControlBase::InPlaceActivate

Other Resources

IOleInPlaceObjectWindowlessImpl Members

IOleInPlaceObjectWindowlessImpl::OnWindowMessage

Dispatches a message from a container to a windowless control that is in-place active.

```
HRESULT OnWindowMessage(
   UINT msg,
   WPARAM WParam,
   LPARAM LParam,
   LRESULT plResultParam
);
```

Remarks

See IOleInPlaceObjectWindowless::OnWindowMessage in the *Platform SDK*.

See Also

Reference

IOleInPlaceObjectWindowlessImpl Class

Other Resources

IOleInPlaceObjectWindowlessImpl Members

IOleInPlaceObjectWindowlessImpl::ReactivateAndUndo

Returns **E_NOTIMPL**.

```
HRESULT ReactivateAndUndo( );
```

Remarks

See IOleInPlaceObject::ReactivateAndUndo in the *Platform SDK*.

See Also

Reference

IOleInPlaceObjectWindowlessImpl Class

Other Resources

IOleInPlaceObjectWindowlessImpl Members

IOleInPlaceObjectWindowlessImpl::SetObjectRects

Called by the container to inform the control that its size and/or position has changed.

```
HRESULT SetObjectRects(
   LPCRECT prcPos,
   LPCRECT prcClip
);
```

Remarks

Updates the control's **m_rcPos** data member and the control display. Only the part of the control that intersects the clip region is displayed. If a control's display was previously clipped but the clipping has been removed, this function can be called to redraw a full view of the control.

See IOleInPlaceObject::SetObjectRects in the *Platform SDK*.

See Also

Reference

IOleInPlaceObjectWindowlessImpl Class

CComControlBase::m_rcPos

Other Resources

IOleInPlaceObjectWindowlessImpl Members

IOleInPlaceObjectWindowlessImpl::UIDeactivate

Deactivates and removes the control's user interface that supports in-place activation.

```
HRESULT UIDeactivate( );
```

Remarks

Sets the control class's data member **m_bUIActive** to **FALSE**. The ATL implementation of this function always returns **S_OK**.

See IOleInPlaceObject::UIDeactivate in the *Platform SDK*.

See Also

Reference

IOleInPlaceObjectWindowlessImpl Class

CComControlBase::m_bUIActive

Other Resources

IOleInPlaceObjectWindowlessImpl Members

IOleObjectImpl Class

This class implements **IUnknown** and is the principal interface through which a container communicates with a control.

```
template<
   class T
>
class ATL_NO_VTABLE IOleObjectImpl :
   public IOleObject
```

Parameters

T

Your class, derived from **IOleObjectImpl**.

Remarks

The IOleObject interface is the principal interface through which a container communicates with a control. Class **IOleObjectImpl** provides a default implementation of this interface and implements **IUnknown** by sending information to the dump device in debug builds.

Related Articles ATL Tutorial, Creating an ATL Project

Requirements

Header: atlctl.h

See Also

Reference

CComControl Class

ActiveX Controls Interfaces

Other Resources

IOleObjectImpl Members

ATL Class Overview

IOleObjectImpl Members

IOleObject Methods

Advise	Establishes an advisory connection with the control.
Close	Changes the control state from running to loaded.
DoVerb	Tells the control to perform one of its enumerated actions.
EnumAdvise	Enumerates the control's advisory connections.
EnumVerbs	Enumerates actions for the control.
GetClientSite	Retrieves the control's client site.
GetClipboardData	Retrieves data from the Clipboard. The ATL implementation returns **E_NOTIMPL**.
GetExtent	Retrieves the extent of the control's display area.
GetMiscStatus	Retrieves the status of the control.
GetMoniker	Retrieves the control's moniker. The ATL implementation returns **E_NOTIMPL**.
GetUserClassID	Retrieves the control's class identifier.
GetUserType	Retrieves the control's user-type name.
InitFromData	Initializes the control from selected data. The ATL implementation returns **E_NOTIMPL**.
IsUpToDate	Checks if the control is up to date. The ATL implementation returns **S_OK**.
SetClientSite	Tells the control about its client site in the container.
SetColorScheme	Recommends a color scheme to the control's application, if any. The ATL implementation returns **E_NOTIMPL**.
SetExtent	Sets the extent of the control's display area.
SetHostNames	Tells the control the names of the container application and container document.
SetMoniker	Tells the control what its moniker is. The ATL implementation returns **E_NOTIMPL**.
Unadvise	Deletes an advisory connection with the control.
Update	Updates the control. The ATL implementation returns **S_OK**.

DoVerb Helper Methods

DoVerbDiscardUndo	Tells the control to discard any undo state it is maintaining.
DoVerbHide	Tells the control to remove its user interface from view.

DoVerbInPlaceActivate	Runs the control and installs its window, but does not install the control's user interface.
DoVerbOpen	Causes the control to be open-edited in a separate window.
DoVerbPrimary	Performs the specified action when the user double-clicks the control. The control defines the actio n, usually to activate the control in-place.
DoVerbShow	Shows a newly inserted control to the user.
DoVerbUIActivate	Activates the control in-place and shows the control's user interface, such as menus and toolbars.
OnPostVerbDiscardUndo	Called by DoVerbDiscardUndo after the undo state is discarded.
OnPostVerbHide	Called by DoVerbHide after the control is hidden.
OnPostVerbInPlaceActivate	Called by DoVerbInPlaceActivate after the control is activated in place.
OnPostVerbOpen	Called by DoVerbOpen after the control has been opened for editing in a separate window.
OnPostVerbShow	Called by DoVerbShow after the control has been made visible.
OnPostVerbUIActivate	Called by DoVerbUIActivate after the control's user interface has been activated.
OnPreVerbDiscardUndo	Called by DoVerbDiscardUndo before the undo state is discarded.
OnPreVerbHide	Called by DoVerbHide before the control is hidden.
OnPreVerbInPlaceActivate	Called by DoVerbInPlaceActivate before the control is activated in place.
OnPreVerbOpen	Called by DoVerbOpen before the control has been opened for editing in a separate window.
OnPreVerbShow	Called by DoVerbShow before the control has been made visible.
OnPreVerbUIActivate	Called by DoVerbUIActivate before the control's user interface has been activated.

See Also

Reference

IOleObjectImpl Class

IOleObjectImpl Methods

For information about the methods in **IOleObjectImpl**, see IOleObjectImpl Members.

IOleObjectImpl::Advise

Establishes an advisory connection with the control.

```
STDMETHOD(Advise)(
   IAdviseSink* pAdvSink,
   DWORD* pdwConnection
);
```

Remarks

See IOleObject::Advise in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class
CComControlBase::m_spOleAdviseHolder
IOleObjectImpl::Unadvise

Other Resources

IOleObjectImpl Members

IOleObjectImpl::Close

Changes the control state from running to loaded.

```
STDMETHOD(Close)(
   DWORD dwSaveOption
);
```

Remarks

Deactivates the control and destroys the control window if it exists. If the control class data member CComControlBase::m_bRequiresSave is **TRUE** and the *dwSaveOption* parameter is either **OLECLOSE_SAVEIFDIRTY** or **OLECLOSE_PROMPTSAVE**, the control properties are saved before closing.

The pointers held in the control class data members CComControlBase::m_spInPlaceSite and CComControlBase::m_spAdviseSink are released, and the data members CComControlBase::m_bNegotiatedWnd, CComControlBase::m_bWndless, and CComControlBase::m_bInPlaceSiteEx are set to **FALSE**.

See IOleObject::Close in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class

Other Resources

IOleObjectImpl Members

IOleObjectImpl::DoVerb

Tells the control to perform one of its enumerated actions.

```
STDMETHOD(DoVerb)(
   LONG iVerb,
   LPMSG /* pMsg */,
   IOleClientSite* pActiveSite,
   LONG /* lindex */,
   HWND hwndParent,
   LPCRECT lprcPosRect
);
```

Remarks

Depending on the value of *iVerb*, one of the ATL **DoVerb** helper functions is called as follows:

***iVerb* Value**	**DoVerb helper function called**
OLEIVERB_DISCARDUNDOSTATE	DoVerbDiscardUndo
OLEIVERB_HIDE	DoVerbHide
OLEIVERB_INPLACEACTIVATE	DoVerbInPlaceActivate
OLEIVERB_OPEN	DoVerbOpen
OLEIVERB_PRIMARY	DoVerbPrimary
OLEIVERB_PROPERTIES	CComControlBase::DoVerbProperties
OLEIVERB_SHOW	DoVerbShow
OLEIVERB_UIACTIVATE	DoVerbUIActivate

See IOleObject::DoVerb in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class

IOleObject::EnumVerbs

Other Resources

IOleObjectImpl Members

IOleObjectImpl::DoVerbDiscardUndo

Tells the control to discard any undo state it is maintaining.

```
HRESULT DoVerbDiscardUndo(
   LPCRECT /* prcPosRect */,
   HWND /* hwndParent */
);
```

Parameters

prcPosRec

[in] Pointer to the rectangle the container wants the control to draw into.

hwndParent

[in] Handle of the window containing the control.

Return Value

Returns **S_OK**.

See Also

Reference

IOleObjectImpl Class

Other Resources

IOleObjectImpl Members

IOleObjectImpl::DoVerbHide

Deactivates and removes the control's user interface, and hides the control.

```
HRESULT DoVerbHide(
   LPCRECT /* prcPosRect */,
   HWND /* hwndParent */
);
```

Parameters

prcPosRec

[in] Pointer to the rectangle the container wants the control to draw into.

hwndParent

[in] Handle of the window containing the control. Not used in the ATL implementation.

Return Value

Returns **S_OK**.

See Also

Reference

IOleObjectImpl Class

IOleObjectImpl::DoVerbShow

Other Resources

IOleObjectImpl Members

IOleObjectImpl::DoVerbInPlaceActivate

Runs the control and installs its window, but does not install the control's user interface.

```
HRESULT DoVerbInPlaceActivate(
   LPCRECT prcPosRect,
   HWND /* hwndParent */
);
```

Parameters

prcPosRec

[in] Pointer to the rectangle the container wants the control to draw into.

hwndParent

[in] Handle of the window containing the control. Not used in the ATL implementation.

Return Value

One of the standard **HRESULT** values.

Remarks

Activates the control in place by calling CComControlBase::InPlaceActivate. Unless the control class's data member **m_bWindowOnly** is **TRUE**, **DoVerbInPlaceActivate** first attempts to activate the control as a windowless control (possible only if the container supports IOleInPlaceSiteWindowless). If that fails, the function attempts to activate the control with extended features (possible only if the container supports IOleInPlaceSiteEx). If that fails, the function attempts to activate the control with no extended features (possible only if the container supports IOleInPlaceSite). If activation succeeds, the function notifies the container the control has been activated.

See Also

Reference

IOleObjectImpl Class
CComControlBase::InPlaceActivate
CComControlBase::m_bWindowOnly

Other Resources

IOleObjectImpl Members

IOleObjectImpl::DoVerbOpen

Causes the control to be open-edited in a separate window.

```
HRESULT DoVerbOpen(
   LPCRECT /* prcPosRect */,
   HWND /* hwndParent */
);
```

Parameters

prcPosRec

[in] Pointer to the rectangle the container wants the control to draw into.

hwndParent

[in] Handle of the window containing the control.

Return Value

Returns **S_OK**.

See Also

Reference

IOleObjectImpl Class

Other Resources

IOleObjectImpl Members

IOleObjectImpl::DoVerbPrimary

Defines the action taken when the user double-clicks the control.

```
HRESULT DoVerbPrimary(
   LPCRECT prcPosRect,
   HWND hwndParent
);
```

Parameters

prcPosRec

[in] Pointer to the rectangle the container wants the control to draw into.

hwndParent

[in] Handle of the window containing the control.

Return Value

One of the standard **HRESULT** values.

Remarks

By default, set to display the property pages. You can override this in your control class to invoke a different behavior on double-click; for example, play a video or go in-place active.

See Also

Reference

IOleObjectImpl Class

CComControlBase::DoVerbProperties

Other Resources

IOleObjectImpl Members

IOleObjectImpl::DoVerbShow

Tells the container to make the control visible.

```
HRESULT DoVerbShow(
   LPCRECT prcPosRect,
   HWND /* hwndParent */
);
```

Parameters

prcPosRec

[in] Pointer to the rectangle the container wants the control to draw into.

hwndParent

[in] Handle of the window containing the control. Not used in the ATL implementation.

Return Value

One of the standard **HRESULT** values.

See Also

Reference

IOleObjectImpl Class

IOleObjectImpl::DoVerbHide

Other Resources

IOleObjectImpl Members

IOleObjectImpl::DoVerbUIActivate

Activates the control's user interface and notifies the container that its menus are being replaced by composite menus.

```
HRESULT DoVerbUIActivate(
   LPCRECT prcPosRect,
   HWND /* hwndParent */
);
```

Parameters

prcPosRec

[in] Pointer to the rectangle the container wants the control to draw into.

hwndParent

[in] Handle of the window containing the control. Not used in the ATL implementation.

Return Value

One of the standard **HRESULT** values.

See Also

Reference

IOleObjectImpl Class
CComControlBase::InPlaceActivate
IOleObjectImpl::DoVerbInPlaceActivate

Other Resources

IOleObjectImpl Members

IOleObjectImpl::EnumAdvise

Supplies an enumeration of registered advisory connections for this control.

```
STDMETHOD(EnumAdvise)(
   IEnumSTATDATA** ppenumAdvise
);
```

Remarks

See IOleObject::EnumAdvise in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class

IOleObjectImpl::EnumVerbs

Other Resources

IOleObjectImpl Members

IOleObjectImpl::EnumVerbs

Supplies an enumeration of registered actions (verbs) for this control by calling **OleRegEnumVerbs**.

```
STDMETHOD(EnumVerbs)(
   IEnumOLEVERB** ppEnumOleVerb
);
```

Remarks

You can add verbs to your project's .rgs file. For example, see CIRCCTL.RGS in the CIRC sample.

See IOleObject::EnumVerbs in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class
OleRegEnumVerbs
IOleObjectImpl::EnumAdvise

Other Resources

IOleObjectImpl Members

IOleObjectImpl::GetClientSite

Puts the pointer in the control class data member CComControlBase::m_spClientSite into *ppClientSite* and increments the reference count on the pointer.

```
STDMETHOD(GetClientSite)(
   IOleClientSite** ppClientSite
);
```

Remarks

See IOleObject::GetClientSite in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class

IOleObjectImpl::SetClientSite

Other Resources

IOleObjectImpl Members

IOleObjectImpl::GetClipboardData

Retrieves data from the Clipboard.

```
STDMETHOD(GetClipboardData)(
   DWORD /* dwReserved */,
   IDataObject** /* ppDataObject */
);
```

Return Value

Returns **E_NOTIMPL**.

Remarks

See IOleObject::GetClipboardData in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class

Other Resources

IOleObjectImpl Members

IOleObjectImpl::GetExtent

Retrieves a running control's display size in HIMETRIC units (0.01 millimeter per unit).

```
STDMETHOD(GetExtent)(
   DWORD dwDrawAspect,
   SIZEL* psizel
);
```

Remarks

The size is stored in the control class data member CComControlBase::m_sizeExtent.

See IOleObject::GetExtent in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class

IOleObjectImpl::SetExtent

Other Resources

IOleObjectImpl Members

IOleObjectImpl::GetMiscStatus

Returns a pointer to registered status information for the control by calling **OleRegGetMiscStatus**.

```
STDMETHOD(GetMiscStatus)(
   DWORD dwAspect,
   DWORD* pdwStatus
);
```

Remarks

The status information includes behaviors supported by the control and presentation data. You can add status information to your project's .rgs file.

See IOleObject::GetMiscStatus in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class
OleRegGetMiscStatus
IOleObjectImpl::EnumVerbs
IOleObjectImpl::GetUserType

Other Resources

IOleObjectImpl Members

IOleObjectImpl::GetMoniker

Retrieves the control's moniker.

```
STDMETHOD(GetMoniker)(
   DWORD /* dwAssign */,
   DWORD /* dwWhichMoniker */,
   IMoniker** /* ppmk  */
);
```

Return Value

Returns **E_NOTIMPL**.

Remarks

See IOleObject::GetMoniker in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class

Other Resources

IOleObjectImpl Members

IOleObjectImpl::GetUserClassID

Returns the control's class identifier.

```
STDMETHOD(GetUserClassID)(
   CLSID* pClsid
);
```

Remarks

See IOleObject::GetUserClassID in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class

IOleObjectImpl::GetUserType

Other Resources

IOleObjectImpl Members

IOleObjectImpl::GetUserType

Returns the control's user-type name by calling **OleRegGetUserType**.

```
STDMETHOD(GetUserType)(
   DWORD dwFormOfType,
   LPOLESTR* pszUserType
);
```

Remarks

The user-type name is used for display in user-interfaces elements such as menus and dialog boxes. You can change the user-type name in your project's .rgs file.

See IOleObject::GetUserType in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class
OleRegGetUserType
IOleObjectImpl::GetUserClassID
IOleObjectImpl::GetMiscStatus
IOleObjectImpl::EnumVerbs

Other Resources

IOleObjectImpl Members

IOleObjectImpl::InitFromData

Initializes the control from selected data.

```
STDMETHOD(InitFromData)(
   IDataObject* /* pDataObject */,
   BOOL /* fCreation */,
   DWORD /* dwReserved */
);
```

Return Value

Returns **E_NOTIMPL**.

Remarks

See IOleObject::InitFromData in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class

Other Resources

IOleObjectImpl Members

IOleObjectImpl::IsUpToDate

Checks if the control is up to date.

```
STDMETHOD(IsUpToDate)(void);
```

Return Value

Returns **S_OK**.

Remarks

See IOleObject::IsUpToDate in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class

Other Resources

IOleObjectImpl Members

IOleObjectImpl::OnPostVerbDiscardUndo

Called by DoVerbDiscardUndo after the undo state is discarded.

```
HRESULT OnPostVerbDiscardUndo( );
```

Return Value

Returns **S_OK**.

Remarks

Override this method with code you want executed after the undo state is discarded.

See Also

Reference

IOleObjectImpl Class
IOleObjectImpl::DoVerbDiscardUndo
IOleObjectImpl::OnPreVerbDiscardUndo

Other Resources

IOleObjectImpl Members

IOleObjectImpl::OnPostVerbHide

Called by DoVerbHide after the control is hidden.

```
HRESULT OnPostVerbHide( );
```

Return Value

Returns **S_OK**.

Remarks

Override this method with code you want executed after the control is hidden.

See Also

Reference

IOleObjectImpl Class
IOleObjectImpl::DoVerbHide
IOleObjectImpl::OnPreVerbHide

Other Resources

IOleObjectImpl Members

IOleObjectImpl::OnPostVerbInPlaceActivate

Called by DoVerbInPlaceActivate after the control is activated in place.

```
HRESULT OnPostVerbInPlaceActivate( );
```

Return Value

Returns **S_OK**.

Remarks

Override this method with code you want executed after the control is activated in place.

See Also

Reference

IOleObjectImpl Class
IOleObjectImpl::DoVerbInPlaceActivate
IOleObjectImpl::OnPreVerbInPlaceActivate

Other Resources

IOleObjectImpl Members

IOleObjectImpl::OnPostVerbOpen

Called by DoVerbOpen after the control has been opened for editing in a separate window.

```
HRESULT OnPostVerbOpen( );
```

Return Value

Returns **S_OK**.

Remarks

Override this method with code you want executed after the control has been opened for editing in a separate window.

See Also

Reference

IOleObjectImpl Class
IOleObjectImpl::DoVerbOpen
IOleObjectImpl::OnPreVerbOpen

Other Resources

IOleObjectImpl Members

IOleObjectImpl::OnPostVerbShow

Called by DoVerbShow after the control has been made visible.

```
HRESULT OnPostVerbShow( );
```

Return Value

Returns **S_OK**.

Remarks

Override this method with code you want executed after the control has been made visible.

See Also

Reference

IOleObjectImpl Class
IOleObjectImpl::DoVerbShow
IOleObjectImpl::OnPreVerbShow

Other Resources

IOleObjectImpl Members

IOleObjectImpl::OnPostVerbUIActivate

Called by DoVerbUIActivate after the control's user interface has been activated.

```
HRESULT OnPostVerbUIActivate( );
```

Return Value

Returns **S_OK**.

Remarks

Override this method with code you want executed after the control's user interface has been activated.

See Also

Reference

IOleObjectImpl Class
IOleObjectImpl::DoVerbUIActivate
IOleObjectImpl::OnPreVerbUIActivate

Other Resources

IOleObjectImpl Members

IOleObjectImpl::OnPreVerbDiscardUndo

Called by DoVerbDiscardUndo before the undo state is discarded.

```
HRESULT OnPreVerbDiscardUndo( );
```

Return Value

Returns **S_OK**.

Remarks

To prevent the undo state from being discarded, override this method to return an error HRESULT.

See Also

Reference

IOleObjectImpl Class
IOleObjectImpl::DoVerbDiscardUndo
IOleObjectImpl::OnPostVerbDiscardUndo

Other Resources

IOleObjectImpl Members

IOleObjectImpl::OnPreVerbHide

Called by DoVerbHide before the control is hidden.

```
HRESULT OnPreVerbHide( );
```

Return Value

Returns **S_OK**.

Remarks

To prevent the control from being hidden, override this method to return an error HRESULT.

See Also

Reference

IOleObjectImpl Class
IOleObjectImpl::DoVerbHide
IOleObjectImpl::OnPostVerbHide

Other Resources

IOleObjectImpl Members

IOleObjectImpl::OnPreVerbInPlaceActivate

Called by DoVerbInPlaceActivate before the control is activated in place.

```
HRESULT OnPreVerbInPlaceActivate( );
```

Return Value

Returns **S_OK**.

Remarks

To prevent the control from being activated in place, override this method to return an error HRESULT.

See Also

Reference

IOleObjectImpl Class
IOleObjectImpl::DoVerbInPlaceActivate
IOleObjectImpl::OnPostVerbInPlaceActivate

Other Resources

IOleObjectImpl Members

IOleObjectImpl::OnPreVerbOpen

Called by DoVerbOpen before the control has been opened for editing in a separate window.

```
HRESULT OnPreVerbOpen( );
```

Return Value

Returns **S_OK**.

Remarks

To prevent the control from being opened for editing in a separate window, override this method to return an error HRESULT.

See Also

Reference

IOleObjectImpl Class
IOleObjectImpl::DoVerbOpen
IOleObjectImpl::OnPostVerbOpen

Other Resources

IOleObjectImpl Members

IOleObjectImpl::OnPreVerbShow

Called by DoVerbShow before the control has been made visible.

```
HRESULT OnPreVerbShow( );
```

Return Value

Returns **S_OK**.

Remarks

To prevent the control from being made visible, override this method to return an error HRESULT.

See Also

Reference

IOleObjectImpl Class
IOleObjectImpl::DoVerbShow
IOleObjectImpl::OnPostVerbShow

Other Resources

IOleObjectImpl Members

IOleObjectImpl::OnPreVerbUIActivate

Called by DoVerbUIActivate before the control's user interface has been activated.

```
HRESULT OnPreVerbUIActivate( );
```

Return Value

Returns **S_OK**.

Remarks

To prevent the control's user interface from being activated, override this method to return an error HRESULT.

See Also

Reference

IOleObjectImpl Class
IOleObjectImpl::DoVerbUIActivate
IOleObjectImpl::OnPostVerbUIActivate

Other Resources

IOleObjectImpl Members

IOleObjectImpl::SetClientSite

Tells the control about its client site in the container.

```
STDMETHOD(SetClientSite)(
   IOleClientSite* pClientSite
);
```

Remarks

The method then returns **S_OK**.

See IOleObject::SetClientSite in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class

IOleObjectImpl::GetClientSite

Other Resources

IOleObjectImpl Members

IOleObjectImpl::SetColorScheme

Recommends a color scheme to the control's application, if any.

```
STDMETHOD(SetColorScheme)(
   LOGPALETTE* /* pLogPal */
);
```

Return Value

Returns **E_NOTIMPL**.

Remarks

See IOleObject::SetColorScheme in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class

Other Resources

IOleObjectImpl Members

IOleObjectImpl::SetExtent

Sets the extent of the control's display area.

```
STDMETHOD(SetExtent)(
   DWORD dwDrawAspect,
   SIZEL* psizel
);
```

Remarks

Otherwise, **SetExtent** stores the value pointed to by *psizel* in the control class data member CComControlBase::m_sizeExtent. This value is in HIMETRIC units (0.01 millimeter per unit).

If the control class data member CComControlBase::m_bResizeNatural is **TRUE**, **SetExtent** also stores the value pointed to by *psizel* in the control class data member CComControlBase::m_sizeNatural.

If the control class data member CComControlBase::m_bRecomposeOnResize is **TRUE**, **SetExtent** calls **SendOnDataChange** and **SendOnViewChange** to notify all advisory sinks registered with the advise holder that the control size has changed.

See IOleObject::SetExtent in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class
IOleObjectImpl::GetExtent
CComControlBase::SendOnDataChange
CComControlBase::SendOnViewChange

Other Resources

IOleObjectImpl Members

IOleObjectImpl::SetHostNames

Tells the control the names of the container application and container document.

```
STDMETHOD(SetHostNames)(
   LPCOLESTR /* szContainerApp */,
   LPCOLESTR /* szContainerObj */
);
```

Return Value

Returns **S_OK**.

Remarks

See IOleObject::SetHostNames in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class

Other Resources

IOleObjectImpl Members

IOleObjectImpl::SetMoniker

Tells the control what its moniker is.

```
STDMETHOD(SetMoniker)(
   DWORD /* dwWhichMoniker */,
   IMoniker** /* pmk */
);
```

Return Value

Returns **E_NOTIMPL**.

Remarks

See IOleObject::SetMoniker in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class

Other Resources

IOleObjectImpl Members

IOleObjectImpl::Unadvise

Deletes the advisory connection stored in the control class's **m_spOleAdviseHolder** data member.

```
STDMETHOD(Unadvise)(
   DWORD dwConnection
);
```

Remarks

See IOleObject::Unadvise in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class
CComControlBase::m_spOleAdviseHolder
IOleObjectImpl::Advise

Other Resources

IOleObjectImpl Members

IOleObjectImpl::Update

Updates the control.

```
STDMETHOD(Update)(void);
```

Return Value

Returns **S_OK**.

Remarks

See IOleObject::Update in the *Platform SDK*.

See Also

Reference

IOleObjectImpl Class

Other Resources

IOleObjectImpl Members

IPerPropertyBrowsingImpl Class

This class implements **IUnknown** and allows a client to access the information in an object's property pages.

```
template<
   class T
>
class ATL_NO_VTABLE IPerPropertyBrowsingImpl :
   public IPerPropertyBrowsing
```

Parameters

T

Your class, derived from **IPerPropertyBrowsingImpl**.

Remarks

The IPerPropertyBrowsing interface allows a client to access the information in an object's property pages. Class **IPerPropertyBrowsingImpl** provides a default implementation of this interface and implements **IUnknown** by sending information to the dump device in debug builds.

Note
If you are using Microsoft Access as the container application, you must derive your class from **IPerPropertyBrowsingImpl**. Otherwise, Access will not load your control.

Related Articles ATL Tutorial, Creating an ATL Project

Requirements

Header: atlctl.h

See Also

Reference

IPropertyPageImpl Class

ISpecifyPropertyPagesImpl Class

Other Resources

IPerPropertyBrowsingImpl Members

ATL Class Overview

IPerPropertyBrowsingImpl Members

IPerPropertyBrowsing Methods

GetDisplayString	Retrieves a string describing a given property.
GetPredefinedStrings	Retrieves an array of strings corresponding to the values that a given property can accept.
GetPredefinedValue	Retrieves a **VARIANT** containing the value of a property identified by a given DISPID. The DISPID is associated with the string name retrieved from **GetPredefinedStrings**. The ATL implementation returns **E_NOTIMPL**.
MapPropertyToPage	Retrieves the CLSID of the property page associated with a given property.

See Also

Reference

IPerPropertyBrowsingImpl Class

IPerPropertyBrowsingImpl Methods

For information about the methods in **IPerPropertyBrowsingImpl**, see IPerPropertyBrowsingImpl Members.

IPerPropertyBrowsingImpl::GetDisplayString

Retrieves a string describing a given property.

```
STDMETHOD(GetDisplayString)(
   DISPID dispID,
   BSTR* pBstr
);
```

Remarks

See IPerPropertyBrowsing::GetDisplayString in the *Platform SDK*.

See Also

Reference

IPerPropertyBrowsingImpl Class

Other Resources

IPerPropertyBrowsingImpl Members

IPerPropertyBrowsingImpl::GetPredefinedStrings

Fills each array with zero items.

```
STDMETHOD(GetPredefinedStrings)(
   DISPID dispID,
   CALPOLESTR* pCaStringsOut,
   CADWORD* pCaCookiesOut
);
```

Return Value

ATL's implementation of GetPredefinedValue returns **E_NOTIMPL**.

Remarks

See IPerPropertyBrowsing::GetPredefinedStrings in the *Platform SDK*.

See Also

Reference

IPerPropertyBrowsingImpl Class
CADWORD
CALPOLESTR

Other Resources

IPerPropertyBrowsingImpl Members

IPerPropertyBrowsingImpl::GetPredefinedValue

Retrieves a **VARIANT** containing the value of a property identified by a given DISPID. The DISPID is associated with the string name retrieved from **GetPredefinedStrings**.

```
STDMETHOD(GetPredefinedValue)(
   DISPID dispID,
   DWORD dwCookie,
   VARIANT* pVarOut
);
```

Return Value

Returns **E_NOTIMPL**.

Remarks

ATL's implementation of GetPredefinedStrings retrieves no corresponding strings.

See IPerPropertyBrowsing::GetPredefinedValue in the *Platform SDK*.

See Also

Reference

IPerPropertyBrowsingImpl Class

Other Resources

IPerPropertyBrowsingImpl Members

IPerPropertyBrowsingImpl::MapPropertyToPage

Retrieves the CLSID of the property page associated with the specified property.

```
STDMETHOD(MapPropertyToPage)(
   DISPID dispID,
   CLSID* pClsid
);
```

Remarks

ATL uses the object's property map to obtain this information.

See IPerPropertyBrowsing::MapPropertyToPage in the *Platform SDK*.

See Also

Reference

IPerPropertyBrowsingImpl Class

BEGIN_PROP_MAP

Other Resources

IPerPropertyBrowsingImpl Members

IPersistPropertyBagImpl Class

This class implements **IUnknown** and allows an object to save its properties to a client-supplied property bag.

```
template <
   class T
>
class ATL_NO_VTABLE IPersistPropertyBagImpl :
   public IPersistPropertyBag
```

Parameters

T

Your class, derived from **IPersistPropertyBagImpl**.

Remarks

The IPersistPropertyBag interface allows an object to save its properties to a client-supplied property bag. Class **IPersistPropertyBagImpl** provides a default implementation of this interface and implements **IUnknown** by sending information to the dump device in debug builds.

IPersistPropertyBag works in conjunction with IPropertyBag and IErrorLog. These latter two interfaces must be implemented by the client. Through **IPropertyBag**, the client saves and loads the object's individual properties. Through **IErrorLog**, both the object and the client can report any errors encountered.

Related Articles ATL Tutorial, Creating an ATL Project

Requirements

Header: atlcom.h

See Also

Reference

BEGIN_PROP_MAP

Other Resources

IPersistPropertyBagImpl Members

ATL Class Overview

IPersistPropertyBagImpl Members

IPersist Methods

GetClassID	Retrieves the object's CLSID.

IPersistPropertyBag Methods

InitNew	Initializes a newly created object. The ATL implementation returns **S_OK**.
Load	Loads the object's properties from a client-supplied property bag.
Save	Saves the object's properties into a client-supplied property bag.

See Also

Reference

IPersistPropertyBagImpl Class

IPersistPropertyBagImpl Methods

For information about the methods in **IPersistPropertyBagImpl**, see IPersistPropertyBagImpl Members.

IPersistPropertyBagImpl::GetClassID

Retrieves the object's CLSID.

```
STDMETHOD(GetClassID)(
   CLSID *pClassID
);
```

Remarks

See IPersist::GetClassID in the *Platform SDK*.

See Also

Reference

IPersistPropertyBagImpl Class

Other Resources

IPersistPropertyBagImpl Members

IPersistPropertyBagImpl::InitNew

Initializes a newly created object.

```
STDMETHOD(InitNew)( );
```

Return Value

Returns **S_OK**.

Remarks

See IPersistPropertyBag::InitNew in the *Platform SDK*.

See Also

Reference

IPersistPropertyBagImpl Class

Other Resources

IPersistPropertyBagImpl Members

IPersistPropertyBagImpl::Load

Loads the object's properties from a client-supplied property bag.

```
STDMETHOD(Load)(
   LPPROPERTYBAG pPropBag,
   LPERRORLOG pErrorLog
);
```

Remarks

ATL uses the object's property map to retrieve this information.

See IPersistPropertyBag::Load in the *Platform SDK*.

See Also

Reference

IPersistPropertyBagImpl Class
BEGIN_PROP_MAP
IPersistPropertyBagImpl::Save
IPropertyBag
IErrorLog

Other Resources

IPersistPropertyBagImpl Members

IPersistPropertyBagImpl::Save

Saves the object's properties into a client-supplied property bag.

```
STDMETHOD(Save)(
   LPPROPERTYBAG pPropBag,
   BOOL fClearDirty,
   BOOL fSaveAllProperties
);
```

Remarks

ATL uses the object's property map to store this information. By default, this method saves all properties, regardless of the value of *fSaveAllProperties*.

See IPersistPropertyBag::Save in the *Platform SDK*.

See Also

Reference

IPersistPropertyBagImpl Class
BEGIN_PROP_MAP
IPersistPropertyBagImpl::Load
IPropertyBag

Other Resources

IPersistPropertyBagImpl Members

IPersistStorageImpl Class

This class implements the IPersistStorage interface.

```
template <
   class T
>
class ATL_NO_VTABLE IPersistStorageImpl :
   public IPersistStorage
```

Parameters

T

Your class, derived from **IPersistStorageImpl**.

Remarks

IPersistStorageImpl implements the IPersistStorage interface, which allows a client to request that your object load and save its persistent data using a storage.

The implementation of this class requires class *T* to make an implementation of the **IPersistStreamInit** interface available via **QueryInterface**. Typically this means that class *T* should derive from IPersistStreamInitImpl, provide an entry for **IPersistStreamInit** in the COM map, and use a property map to describe the class's persistent data.

Related Articles ATL Tutorial, Creating an ATL Project

Requirements

Header: atlcom.h

See Also

Reference

Storages and Streams
IPersistStreamInitImpl Class
IPersistPropertyBagImpl Class

Other Resources

IPersistStorageImpl Members
ATL Class Overview

IPersistStorageImpl Members

IPersist Methods

GetClassID	Retrieves the object's CLSID.

IPersistStorage Methods

HandsOffStorage	Instructs the object to release all storage objects and enter HandsOff mode. The ATL implementation returns **S_OK**.
InitNew	Initializes a new storage.
IsDirty	Checks whether the object's data has changed since it was last saved.
Load	Loads the object's properties from the specified storage.
Save	Saves the object's properties to the specified storage.
SaveCompleted	Notifies an object that it can return to Normal mode to write to its storage object. The ATL implementation returns **S_OK**.

See Also

Reference

IPersistStorageImpl Class

IPersistStorageImpl Methods

For information about the methods in **IPersistStorageImpl**, see IPersistStorageImpl Members.

IPersistStorageImpl::GetClassID

Retrieves the object's CLSID.

```
STDMETHOD(GetClassID)(
   CLSID *pClassID
);
```

Remarks

See IPersist::GetClassID in the *Platform SDK*.

See Also

Reference

IPersistStorageImpl Class

Other Resources

IPersistStorageImpl Members

IPersistStorageImpl::HandsOffStorage

Instructs the object to release all storage objects and enter HandsOff mode.

```
STDMETHOD(HandsOffStorage)(void);
```

Return Value

Returns **S_OK**.

Remarks

See IPersistStorage::HandsOffStorage in the *Platform SDK*.

See Also

Reference

IPersistStorageImpl Class
IPersistStorageImpl::SaveCompleted
IPersistStorageImpl::Save

Other Resources

IPersistStorageImpl Members

IPersistStorageImpl::InitNew

Initializes a new storage.

```
STDMETHOD(InitNew)(
   IStorage*
);
```

Remarks

The ATL implementation delegates to the IPersistStreamInit interface.

See IPersistStorage:InitNew in the *Platform SDK*.

See Also

Reference

IPersistStorageImpl Class

IStorage

Other Resources

IPersistStorageImpl Members

IPersistStorageImpl::IsDirty

Checks whether the object's data has changed since it was last saved.

```
STDMETHOD(IsDirty)(void);
```

Remarks

The ATL implementation delegates to the IPersistStreamInit interface.

See IPersistStorage:IsDirty in the *Platform SDK*.

See Also

Reference

IPersistStorageImpl Class

Other Resources

IPersistStorageImpl Members

IPersistStorageImpl::Load

Loads the object's properties from the specified storage.

```
STDMETHOD(Load)(
   IStorage* pStorage
);
```

Remarks

The ATL implementation delegates to the IPersistStreamInit interface. **Load** uses a stream named "Contents" to retrieve the object's data. The Save method originally creates this stream.

See IPersistStorage:Load in the *Platform SDK*.

See Also

Reference

IPersistStorageImpl Class

IStorage

Other Resources

IPersistStorageImpl Members

IPersistStorageImpl::Save

Saves the object's properties to the specified storage.

```
STDMETHOD(Save)(
   IStorage* pStorage,
   BOOL fSameAsLoad
);
```

Remarks

The ATL implementation delegates to the IPersistStreamInit interface. When **Save** is first called, it creates a stream named "Contents" on the specified storage. This stream is then used in later calls to **Save** and in calls to Load.

See IPersistStorage:Save in the *Platform SDK*.

See Also

Reference

IPersistStorageImpl Class
IPersistStorageImpl::SaveCompleted
IStorage

Other Resources

IPersistStorageImpl Members

IPersistStorageImpl::SaveCompleted

Notifies an object that it can return to Normal mode to write to its storage object.

```
STDMETHOD(SaveCompleted)(
   IStorage*
);
```

Return Value

Returns **S_OK**.

Remarks

See IPersistStorage:SaveCompleted in the *Platform SDK*.

See Also

Reference

IPersistStorageImpl Class
IPersistStorageImpl::HandsOffStorage
IPersistStorageImpl::Save
IStorage

Other Resources

IPersistStorageImpl Members

IPersistStreamInitImpl Class

This class implements **IUnknown** and provides a default implementation of the IPersistStreamInit interface.

```
template<
   class T
>
class ATL_NO_VTABLE IPersistStreamInitImpl :
   public IPersistStreamInit
```

Parameters

T

Your class, derived from **IPersistStreamInitImpl**.

Remarks

The IPersistStreamInit interface allows a client to request that your object loads and saves its persistent data to a single stream. Class **IPersistStreamInitImpl** provides a default implementation of this interface and implements **IUnknown** by sending information to the dump device in debug builds.

Related Articles ATL Tutorial, Creating an ATL Project

Requirements

Header: atlcom.h

See Also

Reference

Storages and Streams

Other Resources

IPersistStreamInitImpl Members

ATL Class Overview

IPersistStreamInitImpl Members

IPersist Methods

GetClassID	Retrieves the object's CLSID.

IPersistStreamInit Methods

GetSizeMax	Retrieves the size of the stream needed to save the object's data. The ATL implementation returns **E_NOTIMPL**.
InitNew	Initializes a newly created object.
IsDirty	Checks whether the object's data has changed since it was last saved.
Load	Loads the object's properties from the specified stream.
Save	Saves the object's properties to the specified stream.

See Also

Reference

IPersistStreamInitImpl Class

IPersistStreamInitImpl Methods

For information about the methods in **IPersistStreamInitImpl**, see IPersistStreamInitImpl Members.

IPersistStreamInitImpl::GetClassID

Retrieves the object's CLSID.

```
STDMETHOD(GetClassID)(
   CLSID *pClassID
);
```

Remarks

See IPersist::GetClassID in the *Platform SDK*.

See Also

Reference

IPersistStreamInitImpl Class

Other Resources

IPersistStreamInitImpl Members

IPersistStreamInitImpl::GetSizeMax

Retrieves the size of the stream needed to save the object's data.

```
STDMETHOD(GetSizeMax)(
   ULARGE_INTEGER FAR* pcbSize
);
```

Return Value

Returns **E_NOTIMPL**.

Remarks

See IPersistStreamInit::GetSizeMax in the *Platform SDK*.

See Also

Reference

IPersistStreamInitImpl Class

Other Resources

IPersistStreamInitImpl Members

IPersistStreamInitImpl::InitNew

Initializes a newly created object.

```
STDMETHOD(InitNew)( );
```

Remarks

See IPersistStreamInit::InitNew in the *Platform SDK*.

See Also

Reference

IPersistStreamInitImpl Class

Other Resources

IPersistStreamInitImpl Members

IPersistStreamInitImpl::IsDirty

Checks whether the object's data has changed since it was last saved.

```
STDMETHOD(IsDirty)( );
```

Remarks

See IPersistStreamInit::IsDirty in the *Platform SDK*.

See Also

Reference

IPersistStreamInitImpl Class

Other Resources

IPersistStreamInitImpl Members

IPersistStreamInitImpl::Load

Loads the object's properties from the specified stream.

```
STDMETHOD(Load)(
   LPSTREAM pStm
);
```

Remarks

ATL uses the object's property map to retrieve this information.

See IPersistStreamInit::Load in the *Platform SDK*.

See Also

Reference

IPersistStreamInitImpl Class
BEGIN_PROP_MAP
IPersistStreamInitImpl::Save
IStream

Other Resources

IPersistStreamInitImpl Members

IPersistStreamInitImpl::Save

Saves the object's properties to the specified stream.

```
STDMETHOD(Save)(
   LPSTREAM pStm,
   BOOL fClearDirty
);
```

Remarks

ATL uses the object's property map to store this information.

See IPersistStreamInit::Save in the *Platform SDK*.

See Also

Reference

IPersistStreamInitImpl Class
BEGIN_PROP_MAP
IPersistStreamInitImpl::Load
IStream

Other Resources

IPersistStreamInitImpl Members

IPointerInactiveImpl Class

This class implements **IUnknown** and the IPointerInactive interface methods.

```
template< class T >
class IPointerInactiveImpl
```

Parameters

T

Your class, derived from **IPointerInactiveImpl**.

Remarks

An inactive object is one that is simply loaded or running. Unlike an active object, an inactive object cannot receive Windows mouse and keyboard messages. Thus, inactive objects use fewer resources and are typically more efficient.

The IPointerInactive interface allows an object to support a minimal level of mouse interaction while remaining inactive. This functionality is particularly useful for controls.

Class **IPointerInactiveImpl** implements the **IPointerInactive** methods by simply returning **E_NOTIMPL**. However, it implements **IUnknown** by sending information to the dump device in debug builds.

Related Articles ATL Tutorial, Creating an ATL Project

Requirements

Header: atlctl.h

See Also

Other Resources

IPointerInactiveImpl Members

ATL Class Overview

IPointerInactiveImpl Members

IPointerInactive Methods

GetActivationPolicy	Retrieves the current activation policy for the object. The ATL implementation returns **E_NOTIMPL**.
OnInactiveMouseMove	Notifies the object that the mouse pointer has moved over it, indicating the object can fire mouse even ts. The ATL implementation returns **E_NOTIMPL**.
OnInactiveSetCursor	Sets the mouse pointer for the inactive object. The ATL implementation returns **E_NOTIMPL**.

See Also
Reference
IPointerInactiveImpl Class

IPointerInactiveImpl Methods

For information about the methods in **IPointerInactiveImpl**, see IPointerInactiveImpl Members.

IPointerInactiveImpl::GetActivationPolicy

Retrieves the current activation policy for the object.

```
HRESULT GetActivationPolicy(
   DWORD* pdwPolicy
);
```

Return Value

Returns **E_NOTIMPL**.

Remarks

See IPointerInactive::GetActivationPolicy in the *Platform SDK*.

See Also

Reference

IPointerInactiveImpl Class

Other Resources

IPointerInactiveImpl Members

IPointerInactiveImpl::OnInactiveMouseMove

Notifies the object that the mouse pointer has moved over it, indicating the object can fire mouse events.

```
HRESULT OnInactiveMouseMove(
   LPCRECT pRectBounds,
   long x,
   long y,
   DWORD dwMouseMsg
);
```

Return Value

Returns **E_NOTIMPL**.

Remarks

See IPointerInactive::OnInactiveMouseMove in the *Platform SDK*.

See Also

Reference

IPointerInactiveImpl Class

Other Resources

IPointerInactiveImpl Members

IPointerInactiveImpl::OnInactiveSetCursor

Sets the mouse pointer for the inactive object.

```
HRESULT OnInactiveSetCursor(
   LPCRECT pRectBounds,
   long x,
   long y,
   DWORD dwMouseMsg,
   BOOL fSetAlways
);
```

Return Value

Returns **E_NOTIMPL**.

Remarks

See IPointerInactive::OnInactiveSetCursor in the *Platform SDK*.

See Also

Reference

IPointerInactiveImpl Class

Other Resources

IPointerInactiveImpl Members

IPropertyNotifySinkCP Class

This class exposes IPropertyNotifySink interface as an outgoing interface on a connectable object.

```
template< class T, class CDV = CComDynamicUnkArray >
class IPropertyNotifySinkCP :
public IConnectionPointImpl< T, &IID_IPropertyNotifySink, CDV >
```

Parameters

T

Your class, derived from **IPropertyNotifySinkCP**.

CDV

A class that manages the connections between a connection point and its sinks. The default value is CComDynamicUnkArray, which allows unlimited connections. You can also use CComUnkArray, which specifies a fixed number of connections.

Remarks

The IPropertyNotifySink interface allows a sink object to receive notifications about property changes. Class **IPropertyNotifySinkCP** exposes this interface as an outgoing interface on a connectable object. The client must implement the **IPropertyNotifySink** methods on the sink.

Derive your class from **IPropertyNotifySinkCP** when you want to create a connection point that represents the **IPropertyNotifySink** interface.

For more information about using connection points in ATL, see the article Connection Points.

Requirements

Header: atlctl.h

See Also

Reference

IConnectionPointImpl Class

IConnectionPointContainerImpl Class

Other Resources

IPropertyNotifySinkCP Members

ATL Class Overview

IPropertyNotifySinkCP Members

IPropertyNotifySinkCP inherits all methods through its base class, IConnectionPointImpl.

See Also

Reference

IPropertyNotifySinkCP Class

IPropertyPageImpl Class

This class implements **IUnknown** and provides a default implementation of the IPropertyPage interface.

```
template< class T >
class IPropertyPageImpl
```

Parameters

T

Your class, derived from **IPropertyPageImpl**.

Remarks

The IPropertyPage interface allows an object to manage a particular property page within a property sheet. Class **IPropertyPageImpl** provides a default implementation of this interface and implements **IUnknown** by sending information to the dump device in debug builds.

Related Articles ATL Tutorial, Creating an ATL Project

Requirements

Header: atlctl.h

See Also

Reference

IPropertyPage2Impl Class
IPerPropertyBrowsingImpl Class
ISpecifyPropertyPagesImpl Class

Other Resources

IPropertyPageImpl Members
ATL Class Overview

IPropertyPageImpl Members

Class Methods

IPropertyPageImpl	Constructor.
SetDirty	Flags the property page's state as changed or unchanged.

IPropertyPage Methods

Activate	Creates the dialog box window for the property page.
Apply	Applies current property page values to the underlying objects specified through **SetObjects**. The ATL implementation returns **S_OK**.
Deactivate	Destroys the window created with **Activate**.
GetPageInfo	Retrieves information about the property page.
Help	Invokes Windows help for the property page.
IsPageDirty	Indicates whether the property page has changed since it was activated.
Move	Positions and resizes the property page dialog box.
SetObjects	Provides an array of **IUnknown** pointers for the objects associated with the property page. These objects receive the current property page values through a call to **Apply**.
SetPageSite	Provides the property page with an **IPropertyPageSite** pointer, through which the property page communicates with the property frame.
Show	Makes the property page dialog box visible or invisible.
TranslateAccelerator	Processes a specified keystroke.

Data Members

m_bDirty	Specifies whether the property page's state has changed.
m_dwDocString	Stores the resource identifier associated with the text string describing the property page.
m_dwHelpContext	Stores the context identifier for the help topic associated with the property page.
m_dwHelpFile	Stores the resource identifier associated with the name of the help file describing the property page.
m_dwTitle	Stores the resource identifier associated with the text string that appears in the tab for the property page.
m_nObjects	Stores the number of objects associated with the property page.
m_pPageSite	Points to the **IPropertyPageSite** interface through which the property page communicates with the property frame.
m_ppUnk	Points to an array of **IUnknown** pointers to the objects associated with the property page.
m_size	Stores the height and width of the property page's dialog box, in pixels.

See Also

Reference

IPropertyPageImpl Class

IPropertyPageImpl Methods

For information about the methods in **IPropertyPageImpl**, see IPropertyPageImpl Members.

IPropertyPageImpl::Activate

Creates the dialog box window for the property page.

```
HRESULT Activate(
   HWND hWndParent,
   LPCRECT pRect,
   BOOL bModal
);
```

Remarks

By default, the dialog box is always modeless, regardless of the value of the *bModal* parameter.

See IPropertyPage::Activate in the *Platform SDK*.

See Also

Reference

IPropertyPageImpl Class

IPropertyPageImpl::Deactivate

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::Apply

Applies current property page values to the underlying objects specified through **SetObjects**.

```
HRESULT Apply( );
```

Return Value

Returns **S_OK**.

Remarks

See IPropertyPage::Apply in the *Platform SDK*.

See Also

Reference

IPropertyPageImpl Class

IPropertyPageImpl::SetObjects

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::Deactivate

Destroys the dialog box window created with Activate.

```
HRESULT Deactivate( );
```

Remarks

See IPropertyPage::Deactivate in the *Platform SDK*.

See Also

Reference

IPropertyPageImpl Class

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::GetPageInfo

Fills the *pPageInfo* structure with information contained in the data members.

```
HRESULT GetPageInfo(
   PROPPAGEINFO* pPageInfo
);
```

Remarks

GetPageInfo loads the string resources associated with m_dwDocString, m_dwHelpFile, and m_dwTitle.

See IPropertyPage::GetPageInfo in the *Platform SDK*.

See Also

Reference

IPropertyPageImpl Class
IPropertyPageImpl::m_dwHelpContext
IPropertyPageImpl::m_size
PROPPAGEINFO

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::Help

Invokes Windows help for the property page.

```
HRESULT Help(
   PROPPAGEINFO* pPageInfo
);
```

Remarks

See IPropertyPage::Help in the *Platform SDK*.

See Also

Reference

IPropertyPageImpl Class

PROPPAGEINFO

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::IPropertyPageImpl

The constructor.

```
IPropertyPageImpl( );
```

Remarks

Initializes all data members.

See Also

Reference

IPropertyPageImpl Class

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::IsPageDirty

Indicates whether the property page has changed since it was activated.

```
HRESULT Help(
   PROPPAGEINFO* pPageInfo
);
```

Remarks

See IPropertyPage::Help in the *Platform SDK*.

See Also

Reference

IPropertyPageImpl Class
IPropertyPageImpl::SetDirty
IPropertyPageImpl::m_bDirty

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::Move

Positions and resizes the property page dialog box.

```
HRESULT Move(
   LPCRECT pRect
);
```

Remarks

See IPropertyPage::Move in the *Platform SDK*.

See Also

Reference

IPropertyPageImpl Class

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::SetDirty

Flags the property page's state as changed or unchanged, depending on the value of *bDirty*.

```
void SetDirty(
   BOOL bDirty
);
```

Parameters

bDirty

[in] If **TRUE**, the property page's state is marked as changed. Otherwise, it is marked as unchanged.

Remarks

If necessary, **SetDirty** informs the frame that the property page has changed.

See Also

Reference

IPropertyPageImpl Class
IPropertyPageImpl::IsPageDirty
IPropertyPageImpl::SetPageSite
IPropertyPageImpl::m_bDirty

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::SetObjects

Provides an array of **IUnknown** pointers for the objects associated with the property page.

```
HRESULT SetObjects(
   ULONG nObjects,
   IUnknown** ppUnk
);
```

Remarks

See IPropertyPage::SetObjects in the *Platform SDK*.

See Also

Reference

IPropertyPageImpl Class
IPropertyPageImpl::Apply
IPropertyPageImpl::m_nObjects
IPropertyPageImpl::m_ppUnk

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::SetPageSite

Provides the property page with an IPropertyPageSite pointer, through which the property page communicates with the property frame.

```
HRESULT SetPageSite(
   IPropertyPageSite* pPageSite
);
```

Remarks

See IPropertyPage::SetPageSite in the *Platform SDK*.

See Also

Reference

IPropertyPageImpl Class
IPropertyPageImpl::m_pPageSite
IPropertyPageSite

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::Show

Makes the property page dialog box visible or invisible.

```
HRESULT Show(
   UINT nCmdShow
);
```

Remarks

See IPropertyPage::Show in the *Platform SDK*.

See Also

Reference

IPropertyPageImpl Class

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::TranslateAccelerator

Processes the keystroke specified in *pMsg*.

```
HRESULT TranslateAccelerator(
   MSG* pMsg
);
```

Remarks

See IPropertyPage::TranslateAccelerator in the *Platform SDK*.

See Also

Reference

IPropertyPageImpl Class

MSG

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl Data Members

For information about the data members in **IPropertyPageImpl**, see IPropertyPageImpl Members.

IPropertyPageImpl::m_bDirty

Specifies whether the property page's state has changed.

```
BOOL m_bDirty;
```

See Also

Reference

IPropertyPageImpl Class
IPropertyPageImpl::IsPageDirty
IPropertyPageImpl::SetDirty

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::m_nObjects

Stores the number of objects associated with the property page.

```
ULONG m_nObjects;
```

See Also

Reference

IPropertyPageImpl Class

IPropertyPageImpl::SetObjects

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::m_dwHelpContext

Stores the context identifier for the help topic associated with the property page.

```
DWORD m_dwHelpContext;
```

See Also

Reference

IPropertyPageImpl Class

IPropertyPageImpl::GetPageInfo

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::m_dwDocString

Stores the resource identifier associated with the text string describing the property page.

```
UINT m_dwDocString;
```

See Also

Reference

IPropertyPageImpl Class

IPropertyPageImpl::GetPageInfo

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::m_dwHelpFile

Stores the resource identifier associated with the name of the help file describing the property page.

```
UINT m_dwHelpFile;
```

See Also

Reference

IPropertyPageImpl Class

IPropertyPageImpl::GetPageInfo

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::m_dwTitle

Stores the resource identifier associated with the text string that appears in the tab for the property page.

```
UINT m_dwTitle;
```

See Also

Reference

IPropertyPageImpl Class

IPropertyPageImpl::GetPageInfo

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::m_pPageSite

Points to the IPropertyPageSite interface through which the property page communicates with the property frame.

```
IPropertyPageSite* m_pPageSite;
```

See Also

Reference

IPropertyPageImpl Class

IPropertyPageImpl::SetPageSite

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::m_ppUnk

Points to an array of **IUnknown** pointers to the objects associated with the property page.

```
IUnknown** m_ppUnk;
```

See Also

Reference

IPropertyPageImpl Class

IPropertyPageImpl::SetObjects

Other Resources

IPropertyPageImpl Members

IPropertyPageImpl::m_size

Stores the height and width of the property page's dialog box, in pixels.

```
SIZE m_size;
```

See Also

Reference

IPropertyPageImpl Class
IPropertyPageImpl::GetPageInfo
SIZE

Other Resources

IPropertyPageImpl Members

IPropertyPage2Impl Class

This class implements **IUnknown** and inherits the default implementation of IPropertyPageImpl.

```
template< class T >
class IPropertyPage2Impl : public IPropertyPageImpl< T >
```

Parameters

T

Your class, derived from **IPropertyPage2Impl**.

Remarks

The IPropertyPage2 interface extends IPropertyPage by adding the **EditProperty** method. This method allows a client to select a specific property in a property page object.

Class **IPropertyPage2Impl** simply returns **E_NOTIMPL** for **IPropertyPage2::EditProperty**. However, it inherits the default implementation of IPropertyPageImpl and implements **IUnknown** by sending information to the dump device in debug builds.

When you create a property page, your class is typically derived from **IPropertyPageImpl**. To provide the extra support of **IPropertyPage2**, modify your class definition and override the **EditProperty** method.

Related Articles ATL Tutorial, Creating an ATL Project

Requirements

Header: atlctl.h

See Also

Reference

IPerPropertyBrowsingImpl Class
ISpecifyPropertyPagesImpl Class

Other Resources

IPropertyPage2Impl Members
ATL Class Overview

IPropertyPage2Impl Members

IPropertyPage2 Methods

EditProperty	Specifies which property control will receive the focus when the property page is activated. The ATL implementation returns **E_NOTIMPL**.

See Also

Reference

IPropertyPage2Impl Class

IPropertyPage2Impl Methods

For information about the methods in **IPropertyPage2Impl**, see IPropertyPage2Impl Members.

IPropertyPage2Impl::EditProperty

Specifies which property control will receive the focus when the property page is activated.

```
HRESULT EditProperty(
   DISPID dispID
);
```

Return Value

Returns **E_NOTIMPL**.

Remarks

See IPropertyPage2::EditProperty in the *Platform SDK*.

See Also

Reference

IPropertyPage2Impl Class

Other Resources

IPropertyPage2Impl Members

IProvideClassInfo2Impl Class

This class provides a default implementation of the IProvideClassInfo and IProvideClassInfo2 methods.

```
template <
   const CLSID* pcoclsid,
   const IID* psrcid,
   const GUID* plibid = &CAtlModule::m_libid,
   WORD wMajor = 1,
   WORD wMinor = 0,
   class tihclass = CComTypeInfoHolder
>
class ATL_NO_VTABLE IProvideClassInfo2Impl :
   public IProvideClassInfo2
```

Parameters

pcoclsid

A pointer to the coclass' identifier.

psrcid

A pointer to the identifier for the coclass' default outgoing dispinterface.

plibid

A pointer to the LIBID of the type library that contains information about the interface. By default, the server-level type library is passed.

wMajor

The major version of the type library. The default value is 1.

wMinor

The minor version of the type library. The default value is 0.

tihclass

The class used to manage the coclass' type information. The default value is **CComTypeInfoHolder**.

Remarks

The IProvideClassInfo2 interface extends IProvideClassInfo by adding the **GetGUID** method. This method allows a client to retrieve an object's outgoing interface IID for its default event set. Class **IProvideClassInfo2Impl** provides a default implementation of the **IProvideClassInfo** and **IProvideClassInfo2** methods.

IProvideClassInfo2Impl contains a static member of type **CComTypeInfoHolder** that manages the type information for the coclass.

Requirements

Header: atlcom.h

See Also

Other Resources

IProvideClassInfo2Impl Members

ATL Class Overview

IProvideClassInfo2Impl Members

Class Methods

IProvideClassInfo2Impl	Constructor.

IProvideClassInfo Methods

GetClassInfo	Retrieves an **ITypeInfo** pointer to the coclass' type information.

IProvideClassInfo2 Methods

GetGUID	Retrieves the GUID for the object's outgoing dispinterface.

Data Members

_tih	Manages the type information for the coclass.

See Also

Reference

IProvideClassInfo2Impl Class

IProvideClassInfo2Impl Methods

For information about the methods in **IProvideClassInfo2Impl**, see IProvideClassInfo2Impl Members.

IProvideClassInfo2Impl::GetClassInfo

Retrieves an ITypeInfo pointer to the coclass' type information.

```
STDMETHOD(GetClassInfo)(
   ITypeInfo** pptinfo
);
```

Remarks

See IProvideClassInfo::GetClassInfo in the *Platform SDK*.

See Also

Reference

IProvideClassInfo2Impl Class

Other Resources

IProvideClassInfo2Impl Members

IProvideClassInfo2Impl::GetGUID

Retrieves the GUID for the object's outgoing dispinterface.

```
STDMETHOD(GetGUID)(
   DWORD dwGuidKind,
   GUID* pGUID
);
```

Remarks

See IProvideClassInfo2::GetGUID in the *Platform SDK*.

See Also

Reference

IProvideClassInfo2Impl Class

Other Resources

IProvideClassInfo2Impl Members

IProvideClassInfo2Impl::IProvideClassInfo2Impl

The constructor.

```
IProvideClassInfo2Impl( );
```

Remarks

Calls **AddRef** on the _tih member. The destructor calls **Release**.

See Also

Reference

IProvideClassInfo2Impl Class

Other Resources

IProvideClassInfo2Impl Members

IProvideClassInfo2Impl Data Members

For information about the data members in **IProvideClassInfo2Impl**, see IProvideClassInfo2Impl Members.

IProvideClassInfo2Impl::_tih

This static data member is an instance of the class template parameter, *tihclass*, which by default is **CComTypeInfoHolder**.

```
static tihclass _tih;
```

Remarks

_tih manages the type information for the coclass.

See Also

Reference

IProvideClassInfo2Impl Class

Other Resources

IProvideClassInfo2Impl Members

IQuickActivateImpl Class

This class combines containers' control initialization into a single call.

```
template<
   class T
>
class ATL_NO_VTABLE IQuickActivateImpl :
   public IQuickActivate
```

Parameters

T

Your class, derived from **IQuickActivateImpl**.

Remarks

The IQuickActivate interface helps containers avoid delays when loading controls by combining initialization in a single call. The **QuickActivate** method allows the container to pass a pointer to a QACONTAINER structure that holds pointers to all the interfaces the control needs. On return, the control passes back a pointer to a QACONTROL structure that holds pointers to its own interfaces, which are used by the container. Class **IQuickActivateImpl** provides a default implementation of **IQuickActivate** and implements **IUnknown** by sending information to the dump device in debug builds.

Related Articles ATL Tutorial, Creating an ATL Project

Requirements

Header: atlctl.h

See Also

Reference

CComControl Class

Other Resources

IQuickActivateImpl Members

ATL Class Overview

IQuickActivateImpl Members

IQuickActivate Methods

GetContentExtent	Retrieves the current display size for a running control.
QuickActivate	Performs quick initialization of controls being loaded.
SetContentExtent	Informs the control of how much display space the container has assigned to it.

See Also

Reference

IQuickActivateImpl Class

IQuickActivateImpl Methods

For information about the methods in **IQuickActivateImpl**, see IQuickActivateImpl Members.

IQuickActivateImpl::GetContentExtent

Retrieves the current display size for a running control.

```
STDMETHOD(GetContentExtent)(
   LPSIZEL pSize
);
```

Remarks

The size is for a full rendering of the control and is specified in HIMETRIC units.

See IQuickActivate::GetContentExtent in the *Platform SDK*.

See Also

Reference

IQuickActivateImpl Class

IQuickActivateImpl::SetContentExtent

Other Resources

IQuickActivateImpl Members

IQuickActivateImpl::QuickActivate

Performs quick initialization of controls being loaded.

```
STDMETHOD(QuickActivate)(
   QACONTAINER* pQACont,
      QACONTROL* pQACtrl
);
```

Remarks

The structure contains pointers to interfaces needed by the control and the values of some ambient properties. Upon return, the control passes a pointer to a QACONTROL structure that contains pointers to its own interfaces that the container requires, and additional status information.

See IQuickActivate::QuickActivate in the *Platform SDK*.

See Also

Reference

IQuickActivateImpl Class

Other Resources

IQuickActivateImpl Members

IQuickActivateImpl::SetContentExtent

Informs the control of how much display space the container has assigned to it.

```
STDMETHOD(SetContentExtent)(
   LPSIZEL pSize
);
```

Remarks

The size is specified in HIMETRIC units.

See IQuickActivate::SetContentExtent in the *Platform SDK*.

See Also

Reference

IQuickActivateImpl Class

IQuickActivateImpl::GetContentExtent

Other Resources

IQuickActivateImpl Members

IRunnableObjectImpl Class

This class implements **IUnknown** and provides a default implementation of the IRunnableObject interface.

```
template< class T >
class IRunnableObjectImpl
```

Parameters

T

Your class, derived from **IRunnableObjectImpl**.

Remarks

The IRunnableObject interface enables a container to determine if a control is running, force it to run, or lock it into the running state. Class **IRunnableObjectImpl** provides a default implementation of this interface and implements **IUnknown** by sending information to the dump device in debug builds.

Related Articles ATL Tutorial, Creating an ATL Project

Requirements

Header: atlctl.h

See Also

Reference

CComControl Class

Other Resources

IRunnableObjectImpl Members

ATL Class Overview

IRunnableObjectImpl Members

IRunnable Object Methods

GetRunningClass	Returns the CLSID of the running control. The ATL implementation sets the CLSID to **GUID_NULL** and returns **E_UNEXPECTED**.
IsRunning	Determines if the control is running. The ATL implementation returns **TRUE**.
LockRunning	Locks the control into the running state. The ATL implementation returns **S_OK**.
Run	Forces the control to run. The ATL implementation returns **S_OK**.
SetContainedObject	Indicates that the control is embedded. The ATL implementation returns **S_OK**.

See Also

Reference

IRunnableObjectImpl Class

IRunnableObjectImpl Methods

For information about the methods in **IRunnableObjectImpl**, see IRunnableObjectImpl Members.

IRunnableObjectImpl::GetRunningClass

Returns the CLSID of the running control.

```
HRESULT GetRunningClass(
   LPCLSID lpClsid
);
```

Return Value

The ATL implementation sets **lpClsid* to **GUID_NULL** and returns **E_UNEXPECTED**.

Remarks

See IRunnableObject::GetRunningClass in the *Platform SDK*.

See Also

Reference

IRunnableObjectImpl Class

Other Resources

IRunnableObjectImpl Members

IRunnableObjectImpl::IsRunning

Determines if the control is running.

```
virtual BOOL IsRunning( );
```

Return Value

The ATL implementation returns **TRUE**.

Remarks

See IRunnableObject::IsRunning in the *Platform SDK*.

See Also

Reference

IRunnableObjectImpl Class

Other Resources

IRunnableObjectImpl Members

IRunnableObjectImpl::LockRunning

Locks the control into the running state.

```
HRESULT LockRunning(
   BOOL fLock,
      BOOL fLastUnlockCloses
);
```

Return Value

The ATL implementation returns **S_OK**.

Remarks

See IRunnableObject::LockRunning in the *Platform SDK*.

See Also

Reference

IRunnableObjectImpl Class

Other Resources

IRunnableObjectImpl Members

IRunnableObjectImpl::Run

Forces the control to run.

```
HRESULT Run(
   LPBINDCTX lpbc
);
```

Return Value

The ATL implementation returns **S_OK**.

Remarks

See IRunnableObject::Run in the *Platform SDK*.

See Also

Reference

IRunnableObjectImpl Class

Other Resources

IRunnableObjectImpl Members

IRunnableObjectImpl::SetContainedObject

Indicates that the control is embedded.

```
HRESULT SetContainedObject(
   BOOL fContained
);
```

Return Value

The ATL implementation returns **S_OK**.

Remarks

See IRunnableObject::SetContainedObject in the *Platform SDK*.

See Also

Reference

IRunnableObjectImpl Class

Other Resources

IRunnableObjectImpl Members

IServiceProviderImpl Class

This class provides a default implementation of the IServiceProvider interface.

```
template <
   class T
>
class ATL_NO_VTABLE IServiceProviderImpl :
   public IServiceProvider
```

Parameters

T

Your class, derived from **IServiceProviderImpl**.

Remarks

The IServiceProvider interface locates a service specified by its GUID and returns the interface pointer for the requested interface on the service. Class **IServiceProviderImpl** provides a default implementation of this interface.

IServiceProviderImpl specifies one method: QueryService, which creates or accesses the specified service and returns an interface pointer to the specified interface for the service.

IServiceProviderImpl uses a service map, starting with BEGIN_SERVICE_MAP and ending with END_SERVICE_MAP.

The service map contains two entries: SERVICE_ENTRY, which indicates a specified service id (SID) supported by the object, and SERVICE_ENTRY_CHAIN, which calls **QueryService** to chain to another object.

Requirements

Header: atlcom.h

Example

See the PerfPersist Sample and the StencilCache Sample.

See Also

Other Resources

IServiceProviderImpl Members

ATL Class Overview

IServiceProviderImpl Members

Methods

QueryService	Creates or accesses the specified service and returns an interface pointer to the specified interface for the service .

See Also

Reference

IServiceProviderImpl Class

IServiceProviderImpl Methods

For information about the methods in **IServiceProviderImpl**, see IServiceProviderImpl Members.

IServiceProviderImpl::QueryService

Creates or accesses the specified service and returns an interface pointer to the specified interface for the service.

```
STDMETHOD(QueryService)(
   REFGUID guidService,
   REFIID riid,
   void** ppvObject
);
```

Remarks

See IServiceProvider::QueryService in the *Platform SDK*.

Example

See the StencilCache Sample.

See Also

Reference

IServiceProviderImpl Class

BEGIN_SERVICE_MAP

Other Resources

IServiceProviderImpl Members

ISpecifyPropertyPagesImpl Class

This class implements **IUnknown** and provides a default implementation of the ISpecifyPropertyPages interface.

```
template<
   class T
>
class ATL_NO_VTABLE ISpecifyPropertyPagesImpl :
   public ISpecifyPropertyPages
```

Parameters

T

Your class, derived from **ISpecifyPropertyPagesImpl**.

Remarks

The ISpecifyPropertyPages interface allows a client to obtain a list of CLSIDs for the property pages supported by an object. Class **ISpecifyPropertyPagesImpl** provides a default implementation of this interface and implements **IUnknown** by sending information to the dump device in debug builds.

Note
Do not expose the **ISpecifyPropertyPages** interface if your object does not support property pages.

Related Articles ATL Tutorial, Creating an ATL Project

Requirements

Header: atlcom.h

See Also

Reference

IPropertyPageImpl Class

IPerPropertyBrowsingImpl Class

Other Resources

ISpecifyPropertyPagesImpl Members

ATL Class Overview

ISpecifyPropertyPagesImpl Members

ISpecifyPropertyPages Methods

GetPages	Fills a Counted Array of UUID values. Each UUID corresponds to the CLSID for one of the property pages that can be displayed in the object's property sheet.

See Also

Reference

ISpecifyPropertyPagesImpl Class

ISpecifyPropertyPagesImpl Methods

For information about the methods in **ISpecifyPropertyPagesImpl**, see ISpecifyPropertyPagesImpl Members.

ISpecifyPropertyPagesImpl::GetPages

Fills the array in the CAUUID structure with the CLSIDs for the property pages that can be displayed in the object's property sheet.

```
STDMETHOD(GetPages)(
   CAUUID* pPages
);
```

Remarks

ATL uses the object's property map to retrieve each CLSID.

See ISpecifyPropertyPages::GetPages in the *Platform SDK*.

See Also

Reference

ISpecifyPropertyPagesImpl Class

BEGIN_PROP_MAP

Other Resources

ISpecifyPropertyPagesImpl Members

ISupportErrorInfoImpl Class

This class provides a default implementation of the ISupportErrorInfo Interface and can be used when only a single interface generates errors on an object.

```
template<
   const IID* piid
>
class ATL_NO_VTABLE ISupportErrorInfoImpl :
   public ISupportErrorInfo
```

Parameters

piid

A pointer to the IID of an interface that supports IErrorInfo.

Remarks

The ISupportErrorInfo Interface ensures that error information can be returned to the client. Objects that use **IErrorInfo** must implement **ISupportErrorInfo**.

Class **ISupportErrorInfoImpl** provides a default implementation of **ISupportErrorInfo** and can be used when only a single interface generates errors on an object. For example:

```
class CMyClass :
   public IDispatchImpl< ... >,
   public CComObjectRoot,
   public CComCoClass< ... >
   public ISupportErrorInfoImpl< &IID_IMyClass >
{
   ...
};
```

Requirements

Header: atlcom.h

See Also

Other Resources

ISupportErrorInfoImpl Members

ATL Class Overview

ISupportErrorInfoImpl Members

ISupportErrorInfo Methods

InterfaceSupportsErrorInfo	Indicates whether the interface identified by *riid* supports the IErrorInfo interface.

See Also

Reference

ISupportErrorInfoImpl Class

ISupportErrorInfoImpl Methods

For information about the methods in **ISupportErrorInfoImpl**, see ISupportErrorInfoImpl Members.

ISupportErrorInfoImpl::InterfaceSupportsErrorInfo

Indicates whether the interface identified by *riid* supports the IErrorInfo interface.

```
STDMETHOD(InterfaceSupportsErrorInfo)(
   REFIID riid
);
```

Remarks

See ISupportErrorInfo::InterfaceSupportsErrorInfo in the *Platform SDK*.

See Also

Reference

ISupportErrorInfoImpl Class

Other Resources

ISupportErrorInfoImpl Members

IViewObjectExImpl Class

This class implements **IUnknown** and provides default implementations of the IViewObject, IViewObject2, and IViewObjectEx interfaces.

```
template<
   class T
>
class ATL_NO_VTABLE IViewObjectExImpl :
   public IViewObjectEx
```

Parameters

T

Your class, derived from **IViewObjectExImpl**.

Remarks

The IViewObject, IViewObject2, and IViewObjectEx interfaces enable a control to display itself directly, and to create and manage an advise sink to notify the container of changes in the control display. The **IViewObjectEx** interface provides support for extended control features such as flicker-free drawing, non-rectangular and transparent controls, and hit-testing (for example, how close a mouse click must be to be considered on the control). Class **IViewObjectExImpl** provides a default implementation of these interfaces and implements **IUnknown** by sending information to the dump device in debug builds.

Requirements

Header: atlctl.h

See Also

Tasks

Creating an ATL Project

Reference

CComControl Class

ActiveX Controls Interfaces

ATL Tutorial

Other Resources

IViewObjectExImpl Members

ATL Class Overview

IViewObjectExImpl Members

IViewObject Methods

Draw	Draws a representation of the control onto a device context.
Freeze	Freezes the drawn representation of a control so it won't change until an **Unfreeze**. The ATL implementation returns **E_NOTIMPL**.
GetAdvise	Retrieves an existing advisory sink connection on the control, if there is one.
GetColorSet	Returns the logical palette used by the control for drawing. The ATL implementation returns **E_NOTIMPL**.
GetExtent	Retrieves the control's display size in HIMETRIC units (0.01 millimeter per unit) from the control class data member CComControlBase::m_sizeExtent.
GetNaturalExtent	Provides sizing hints from the container for the object to use as the user resizes it.
GetRect	Returns a rectangle describing a requested drawing aspect. The ATL implementation returns **E_NOTIMPL**.
GetViewStatus	Returns information about the opacity of the object and what drawing aspects are supported.
QueryHitPoint	Checks if the specified point is in the specified rectangle and returns a HITRESULT value in *pHitResult*.
QueryHitRect	Checks whether the control's display rectangle overlaps any point in the specified location rectangle and returns a **HITRESULT** value in *pHitResult*.
SetAdvise	Sets up a connection between the control and an advise sink so the sink can be notified about changes in the control's view.
Unfreeze	Unfreezes the drawn representation of the control. The ATL implementation returns **E_NOTIMPL**.

See Also

Reference

IViewObjectExImpl Class

IViewObjectExImpl Methods

For information about the methods in **IViewObjectExImpl**, see IViewObjectExImpl Members.

IViewObjectExImpl::Draw

Draws a representation of the control onto a device context.

```
STDMETHOD(Draw)(
   DWORD dwDrawAspect,
      LONG lindex,
      void* pvAspect,
      DVTARGETDEVICE* ptd,
      HDC hicTargetDev,
      LPCRECTL prcBounds,
      LPCRECTL prcWBounds,
      BOOL(_stdcall * /* pfnContinue /) (DWORD_PTR dwContinue),
   DWORD_PTR /* dwContinue */
);
```

Remarks

This method calls **CComControl::OnDrawAdvanced** which in turn calls your control class's **OnDraw** method. An **OnDraw** method is automatically added to your control class when you create your control with the ATL Control Wizard. The Wizard's default **OnDraw** draws a rectangle with the label "ATL 3.0".

See IViewObject::Draw in the *Platform SDK*.

See Also

Reference

IViewObjectExImpl Class
CComControlBase::OnDrawAdvanced
CComControlBase::OnDraw

Other Resources

IViewObjectExImpl Members

IViewObjectExImpl::Freeze

Freezes the drawn representation of a control so it won't change until an **Unfreeze**. The ATL implementation returns **E_NOTIMPL**.

```
STDMETHOD(Freeze)(
   DWORD /* dwAspect */,
      LONG /* lindex */,
      void* /* pvAspect */,
      DWORD* /* pdwFreeze */
);
```

Remarks

See IViewObject::Freeze in the *Platform SDK*.

See Also

Reference

IViewObjectExImpl Class

Other Resources

IViewObjectExImpl Members

IViewObjectExImpl::GetAdvise

Retrieves an existing advisory sink connection on the control, if there is one.

```
STDMETHOD(GetAdvise)(
   DWORD* /* pAspects */,
      DWORD* /* pAdvf */,
      IAdviseSink** /* ppAdvSink */
);
```

Remarks

The advisory sink is stored in the control class data member CComControlBase::m_spAdviseSink.

See IViewObject::GetAdvise in the *Platform SDK*.

See Also

Reference

IViewObjectExImpl Class

IViewObjectExImpl::SetAdvise

Other Resources

IViewObjectExImpl Members

IViewObjectExImpl::GetColorSet

Returns the logical palette used by the control for drawing. The ATL implementation returns **E_NOTIMPL**.

```
STDMETHOD(GetColorSet)(
   DWORD /* dwAspect */,
      LONG /* lindex */,
      void* /* pvAspect */,
      DVTARGETDEVICE* /* ptd */,
      HDC /* hicTargetDevice */,
      LOGPALETTE** /* ppColorSet */
);
```

Remarks

See IViewObject::GetColorSet in the *Platform SDK*.

See Also

Reference

IViewObjectExImpl Class

Other Resources

IViewObjectExImpl Members

IViewObjectExImpl::GetExtent

Retrieves the control's display size in HIMETRIC units (0.01 millimeter per unit) from the control class data member CComControlBase::m_sizeExtent.

```
STDMETHOD(GetExtent)(
   DWORD /* dwDrawAspect */,
      LONG /* lindex */,
      DVTARGETDEVICE* /* ptd */,
      LPSIZEL* lpsizel
);
```

Remarks

See IViewObject2::GetExtent in the *Platform SDK*.

See Also

Reference

IViewObjectExImpl Class

Other Resources

IViewObjectExImpl Members

IViewObjectExImpl::GetNaturalExtent

Provides sizing hints from the container for the object to use as the user resizes it.

```
STDMETHOD(GetNaturalExtent)(
   DWORD dwAspect,
      LONG /* lindex */,
      DVTARGETDEVICE* /* ptd */,
      HDC /* hicTargetDevice */,
      DVEXTENTINFO* pExtentInfo,
      LPSIZEL psizel
);
```

Remarks

If *dwAspect* is **DVASPECT_CONTENT** and *pExtentInfo->dwExtentMode* is **DVEXTENT_CONTENT**, sets **psizel* to the control class's data member CComControlBase::m_sizeNatural. Otherwise, returns an error **HRESULT**.

See IViewObjectEx::GetNaturalExtent in the *Platform SDK*.

See Also

Reference

IViewObjectExImpl Class

Other Resources

IViewObjectExImpl Members

IViewObjectExImpl::GetRect

Returns a rectangle describing a requested drawing aspect. The ATL implementation returns **E_NOTIMPL**.

```
STDMETHOD(GetRect)(
   DWORD /* dwAspect */,
      LPRECTL /* pRect */
);
```

Remarks

See IViewObjectEx::GetRect in the *Platform SDK*.

See Also

Reference

IViewObjectExImpl Class

Other Resources

IViewObjectExImpl Members

IViewObjectExImpl::GetViewStatus

Returns information about the opacity of the object and what drawing aspects are supported.

```
STDMETHOD(GetViewStatus)(
   DWORD* pdwStatus
);
```

Remarks

By default, ATL sets *pdwStatus* to indicate that the control supports **VIEWSTATUS_OPAQUE** (possible values are in the VIEWSTATUS enumeration).

See IViewObjectEx::GetViewStatus in the *Platform SDK*.

See Also

Reference

IViewObjectExImpl Class

Other Resources

IViewObjectExImpl Members

IViewObjectExImpl::QueryHitPoint

Checks if the specified point is in the specified rectangle and returns a HITRESULT value in *pHitResult*.

```
STDMETHOD(QueryHitPoint)(
   DWORD dwAspect,
      LPCRECT pRectBounds,
      POINT ptlLoc,
      LONG /* lCloseHit */,
      DWORD* /* pHitResult */
);
```

Remarks

The value can be either **HITRESULT_HIT** or **HITRESULT_OUTSIDE**.

If *dwAspect* equals DVASPECT_CONTENT, the method returns **S_OK**. Otherwise, the method returns **E_FAIL**.

See IViewObjectEx::QueryHitPoint in the *Platform SDK*.

See Also

Reference

IViewObjectExImpl Class

IViewObjectExImpl::QueryHitRect

Other Resources

IViewObjectExImpl Members

IViewObjectExImpl::QueryHitRect

Checks whether the control's display rectangle overlaps any point in the specified location rectangle and returns a HITRESULT value in *pHitResult*.

```
STDMETHOD(QueryHitRect)(
   DWORD dwAspect,
      LPCRECT pRectBounds,
      LPRECT prcLoc,
      LONG /* lCloseHit */,
      DWORD* /* pHitResult */
);
```

Remarks

The value can be either **HITRESULT_HIT** or **HITRESULT_OUTSIDE**.

If *dwAspect* equals DVASPECT_CONTENT, the method returns **S_OK**. Otherwise, the method returns **E_FAIL**.

See IViewObjectEx::QueryHitRect in the *Platform SDK*.

See Also

Reference

IViewObjectExImpl Class

IViewObjectExImpl::QueryHitPoint

Other Resources

IViewObjectExImpl Members

IViewObjectExImpl::SetAdvise

Sets up a connection between the control and an advise sink so the sink can be notified about changes in the control's view.

```
STDMETHOD(SetAdvise)(
   DWORD /* aspects */,
      DWORD /* advf */,
      IAdviseSink* pAdvSink
);
```

Remarks

The pointer to the IAdviseSink interface on the advise sink is stored in the control class data member CComControlBase::m_spAdviseSink.

See IViewObject::SetAdvise in the *Platform SDK*.

See Also

Reference

IViewObjectExImpl Class

IViewObjectExImpl::GetAdvise

Other Resources

IViewObjectExImpl Members

IViewObjectExImpl::Unfreeze

Unfreezes the drawn representation of the control. The ATL implementation returns **E_NOTIMPL**.

```
STDMETHOD(Unfreeze)(
   DWORD /* dwFreeze */
);
```

Remarks

See IViewObject::Unfreeze in the *Platform SDK*.

See Also

Reference

IViewObjectExImpl Class

Other Resources

IViewObjectExImpl Members

_U_MENUorID Class

This class provides wrappers for **CreateWindow** and **CreateWindowEx**.

```
class _U_MENUorID
```

Remarks

This argument adapter class allows either IDs (**UINT**s) or menu handles (**HMENU**s) to be passed to a function without requiring an explicit cast on the part of the caller.

This class is designed for implementing wrappers to the Windows API, particularly the CreateWindow and CreateWindowEx functions, both of which accept an **HMENU** argument that may be a child window identifier (**UINT**) rather than a menu handle. For example, you can see this class in use as a parameter to CWindowImpl::Create.

The class defines two constructor overloads: one accepts a **UINT** argument and the other accepts an **HMENU** argument. The **UINT** argument is just cast to an **HMENU** in the constructor and the result stored in the class's single data member, m_hMenu. The argument to the **HMENU** constructor is stored directly without conversion.

Requirements

Header: atlwin.h

See Also

Other Resources

_U_MENUorID Members

ATL Class Overview

_U_MENUorID Members

Methods

_U_MENUorID	The constructor.

Data Members

m_hMenu	A handle to a menu.

See Also

Reference

_U_MENUorID Class

_U_MENUorID Methods

For information about the methods in **_U_MENUorID**, see _U_MENUorID Members.

_U_MENUorID::_U_MENUorID

The **UINT** argument is just cast to an **HMENU** in the constructor and the result stored in the class's single data member, m_hMenu.

```
_U_MENUorID(
   UINT nID
);
_U_MENUorID(
   HMENU hMenu
);
```

Parameters

nID

A child window identifier.

hMenu

A menu handle.

Remarks

The argument to the **HMENU** constructor is stored directly without conversion.

See Also

Reference

_U_MENUorID Class

_U_MENUorID::m_hMenu

Other Resources

_U_MENUorID Members

_U_MENUorID Data Members

For information about the data members in **_U_MENUorID**, see _U_MENUorID Members.

_U_MENUorID::m_hMenu

The class holds the value passed to either of its constructors as a public **HMENU** data member.

```
HMENU m_hMenu;
```

See Also

Reference

_U_MENUorID Class

_U_MENUorID::_U_MENUorID

Other Resources

_U_MENUorID Members

_U_RECT Class

This argument adapter class allows either **RECT** pointers or references to be passed to a function that is implemented in terms of pointers.

```
class _U_RECT
```

Remarks

The class defines two constructor overloads: one accepts a **RECT&** argument and the other accepts an **LPRECT** argument. The first constructor stores the address of the reference argument in the class's single data member, m_lpRect. The argument to the pointer constructor is stored directly without conversion.

Requirements

Header: atlwin.h

See Also

Other Resources

_U_RECT Members

ATL Class Overview

_U_RECT Members

Methods

_U_RECT	The constructor.

Data Members

m_lpRect	Pointer to a **RECT**.

See Also

Reference

_U_RECT Class

_U_RECT Class Methods

For information about the methods in **_U_RECT**, see _U_RECT Members.

_U_RECT::_U_RECT

The address of the reference argument is stored in the class's single data member, m_lpRect.

```
_U_RECT(
   RECT& rc
);
_U_RECT(
   LPRECT lpRect
);
```

Parameters

rc

A **RECT** reference.

lpRect

A **RECT** pointer.

Remarks

The argument to the pointer constructor is stored directly without conversion.

See Also

Reference

_U_RECT Class

_U_RECT::m_lpRect

Other Resources

_U_RECT Members

_U_RECT Class Data Members

For information about the data members in **_U_RECT**, see _U_RECT Members.

_U_RECT::m_lpRect

The class holds the value passed to either of its constructors as a public **LPRECT** data member.

```
LPRECT m_lpRect;
```

See Also

Reference

_U_RECT Class

_U_RECT::_U_RECT

Other Resources

_U_RECT Members

_U_STRINGorID Class

This argument adapter class allows either resource names (**LPCTSTR**s) or resource IDs (**UINT**s) to be passed to a function without requiring the caller to convert the ID to a string using the **MAKEINTRESOURCE** macro.

```
class _U_STRINGorID
```

Remarks

This class is designed for implementing wrappers to the Windows resource management API such as the FindResource, LoadIcon, and LoadMenu functions, which accept an **LPCTSTR** argument that may be either the name of a resource or its ID.

The class defines two constructor overloads: one accepts a **LPCTSTR** argument and the other accepts a **UINT** argument. The **UINT** argument is converted to a resource type compatible with Windows resource-management functions using the **MAKEINTRESOURCE** macro and the result stored in the class's single data member, m_lpstr. The argument to the **LPCTSTR** constructor is stored directly without conversion.

Requirements

Header: atlwin.h

See Also

Other Resources

_U_STRINGorID Members

ATL Class Overview

_U_STRINGorID Members

Methods

_U_STRINGorID	The constructor.

Data Members

m_lpstr	The resource identifier.

See Also

Reference

_U_STRINGorID Class

_U_STRINGorID Methods

For information about the methods in **_U_STRINGorID**, see _U_STRINGorID Members.

_U_STRINGorID::_U_STRINGorID

The **UINT** constructor converts its argument to a resource type compatible with Windows resource-management functions using the **MAKEINTRESOURCE** macro and the result is stored in the class's single data member, m_lpstr.

```
_U_STRINGorID(
   UINT nID
);
_U_STRINGorID(
   LPCTSTR lpString
);
```

Parameters

nID

A resource ID.

lpString

A resource name.

Remarks

The argument to the **LPCTSTR** constructor is stored directly without conversion.

See Also

Reference

_U_STRINGorID Class

_U_STRINGorID::m_lpstr

Other Resources

_U_STRINGorID Members

_U_STRINGorID Data Members

For information about the data members in **_U_STRINGorID**, see _U_STRINGorID Members.

_U_STRINGorID::m_lpstr

The class holds the value passed to either of its constructors as a public **LPCTSTR** data member.

```
LPCTSTR m_lpstr;
```

See Also

Reference

_U_STRINGorID Class

_U_STRINGorID::_U_STRINGorID

Other Resources

_U_STRINGorID Members

Win32ThreadTraits Class

This class provides the creation function for a Windows thread. Use this class if the thread will not use CRT functions.

```
class Win32ThreadTraits
```

Remarks

Thread traits are classes that provide a creation function for a particular type of thread. The creation function has the same signature and semantics as the Windows CreateThread function.

Thread traits are used by the following classes:

- CThreadPool
- CWorkerThread

If the thread will be using CRT functions, use CRTThreadTraits instead.

Requirements

Header: atlbase.h

See Also

Other Resources

Win32ThreadTraits Members
ATL Class Overview

Win32ThreadTraits Members

Static Functions

CreateThread	Call this function to create a thread that should not use CRT functions.

See Also

Reference

Win32ThreadTraits Class

Win32ThreadTraits Static Functions

For information about the static functions in **Win32ThreadTraits**, see Win32ThreadTraits Members.

Win32ThreadTraits::CreateThread

Call this function to create a thread that should not use CRT functions.

```
static HANDLE CreateThread(
   LPSECURITY_ATTRIBUTES lpsa,
   DWORD dwStackSize,
   LPTHREAD_START_ROUTINE pfnThreadProc,
   void* pvParam,
   DWORD dwCreationFlags,
   DWORD* pdwThreadId
) throw( );
```

Parameters

lpsa

The security attributes for the new thread.

dwStackSize

The stack size for the new thread.

pfnThreadProc

The thread procedure of the new thread.

pvParam

The parameter to be passed to the thread procedure.

dwCreationFlags

The creation flags (0 or CREATE_SUSPENDED).

pdwThreadId

[out] Address of the DWORD variable that, on success, receives the thread ID of the newly created thread.

Return Value

Returns the handle to the newly created thread or NULL on failure. Call GetLastError to get extended error information.

Remarks

See CreateThread for further information on the parameters to this function.

This function calls **CreateThread** to create the thread.

See Also

Reference

Win32ThreadTraits Class

Other Resources

Win32ThreadTraits Members

ATL Functions

To find an ATL function by category, see the following topics.

COM Map Global Functions

Provide support for COM map **IUnknown** implementations.

Composite Control Global Functions

Provide support for creating dialog boxes, and for creating, hosting and licensing ActiveX controls.

Connection Point Global Functions

Provide support for connection points and sink maps.

Debugging and Error Reporting Global Functions

Provide useful debugging and trace facilities.

Device Context Global Functions

Creates a device context for a given device.

Event Handling Global Functions

Provides an event handler.

Marshaling Global Functions

Provide support for marshaling and converting marshaling data into interface pointers.

Pixel/HIMETRIC Conversion Global Functions

Provide support for converting to and from pixel and HIMETRIC units.

Registry and TypeLib Global Functions

Provide support for loading and registering a type library.

Security Global Functions

Provide support for modifying SID and ACL objects.

Security Identifier Global Functions

Return common well-known SID objects.

Server Registration Global Functions

Provide support for registering and unregistering server objects in the object map.

WinModule Global Functions

Provide support for **_AtlCreateWndData** structure operations.

See Also

Other Resources

ATL Functions Alphabetical Reference
ATL Reference
ATL Macros
ATL Global Variables
ATL Structures
ATL Typedefs
ATL Classes

COM Map Global Functions

These functions provide support for COM Map **IUnknown** implementations.

AtlInternalQueryInterface	Delegates to the **IUnknown** of a nonaggregated object.
InlineIsEqualIUnknown	Generates efficient code for comparing interfaces against **IUnknown**.

See Also

Other Resources

ATL Functions

COM Map Macros

Composite Control Global Functions

These functions provide support for creating dialog boxes, and for creating, hosting and licensing ActiveX controls.

AtlAxDialogBox	Creates a modal dialog box from a dialog template provided by the user. The resulting dialog box can contain ActiveX controls.
AtlAxCreateDialog	Creates a modeless dialog box from a dialog template provided by the user. The resulting dialog box can contain ActiveX controls.
AtlAxCreateControl	Creates an ActiveX control, initializes it, and hosts it in the specified window.
AtlAxCreateControlEx	Creates an ActiveX control, initializes it, hosts it in the specified window, and retrieves an interface pointer (or pointers) from the control.
AtlAxCreateControlLic	Creates a licensed ActiveX control, initializes it, and hosts it in the specified window.
AtlAxCreateControlLicEx	Creates a licensed ActiveX control, initializes it, hosts it in the specified window, and retrieves an i nterface pointer (or pointers) from the control.
AtlAxAttachControl	Attaches a previously created control to the specified window.
AtlAxGetHost	Used to obtain a direct interface pointer to the container for a specified window (if any), given its handle.
AtlAxGetControl	Used to obtain a direct interface pointer to the control contained inside a specified window (if any), given its handle.
AtlSetChildSite	Initializes the **IUnknown** of the child site.
AtlAxWinInit	Initializes the hosting code for AxWin objects.
AtlAxWinTerm	Uninitializes the hosting code for AxWin objects.
AtlGetObjectSourceInterface	Returns information about the default source interface of an object.

See Also

Other Resources

ATL Functions

Composite Control Macros

Connection Point Global Functions

These functions provide support for connection points and sink maps.

AtlAdvise	Creates a connection between an object's connection point and a client's sink.
AtlUnadvise	Terminates the connection established through **AtlAdvise**.
AtlAdviseSinkMap	Advises or unadvises entries in an event sink map.

See Also

Other Resources

ATL Functions

Connection Point Macros

Device Context Global Functions

This function creates a device context for a given device.

AtlCreateTargetDC	Creates a device context.

See Also

Other Resources

ATL Functions

Debugging and Error Reporting Global Functions

These functions provide useful debugging and trace facilities.

AtlHresultFromLastError	Returns a **GetLastError** error code in the form of an HRESULT.
AtlHresultFromWin32	Converts a Win32 error code into an HRESULT.
AtlReportError	Sets up **IErrorInfo** to provide error details to a client.
AtlThrow	Throws a **CAtlException**.
AtlThrowLastWin32	Call this function to signal an error based on the result of the Windows function **GetLastError**.
AtlTraceLoadSettings	Call this function to load trace settings from a file.
AtlTraceSaveSettings	Call this function to save the current trace settings to a file.

See Also

Other Resources

ATL Functions

Debugging and Error Reporting Macros

Event Handling Global Functions

This function provides an event handler.

AtlWaitWithMessageLoop	Waits for an object to be signaled, meanwhile dispatching window messages as needed.

See Also

Other Resources

ATL Functions

Marshaling Global Functions

These functions provide support for marshaling and converting marshaling data into interface pointers.

AtlFreeMarshalStream	Releases the marshal data and the **IStream** pointer.
AtlMarshalPtrInProc	Creates a new stream object and marshals the specified interface pointer.
AtlUnmarshalPtr	Converts a stream's marshaling data into an interface pointer.

See Also

Other Resources

ATL Functions

Pixel/HIMETRIC Conversion Global Functions

These functions provide support for converting to and from pixel and HIMETRIC units.

AtlHiMetricToPixel	Converts HIMETRIC units (each unit is 0.01 millimeter) to pixels.
AtlPixelToHiMetric	Converts pixels to HIMETRIC units (each unit is 0.01 millimeter).

See Also

Other Resources

ATL Functions

Registry and TypeLib Global Functions

These functions provide support for loading and registering a type library.

AtlRegisterTypeLib	This function is called to register a type library.
AtlUnRegisterTypeLib	This function is called to unregister a type library
AtlLoadTypeLib	This function is called to load a type library.
AtlUpdateRegistryFromResourceD	This function is called to update the registry from the supplied resource.
RegistryDataExchange	This function is called to read from, or write to, the system registry. Called by the Registry Data Exchange Macros.

See Also

Other Resources

ATL Functions

Security Global Functions

These functions provide support for modifying SID and ACL objects.

AtlGetDacl	Call this function to retrieve the discretionary access-control list (DACL) information of a specified object.
AtlSetDacl	Call this function to set the discretionary access-control list (DACL) information of a specified object.
AtlGetGroupSid	Call this function to retrieve the group security identifier (SID) of an object.
AtlSetGroupSid	Call this function to set the group security identifier (SID) of an object.
AtlGetOwnerSid	Call this function to set the owner security identifier (SID) of an object.
AtlSetOwnerSid	Call this function to retrieve the owner security identifier (SID) of an object.
AtlGetSacl	Call this function to retrieve the system access-control list (SACL) information of a specified object.
AtlSetSacl	Call this function to set the system access-control list (SACL) information of a specified object.
AtlGetSecurityDescriptor	Call this function to retrieve the security descriptor of a given object.

See Also

Other Resources

ATL Functions

Security Identifier Global Functions

These functions return common well-known SID objects.

Sids::AccountOps	Returns the DOMAIN_ALIAS_RID_ACCOUNT_OPS SID.
Sids::Admins	Returns the DOMAIN_ALIAS_RID_ADMINS SID.
Sids::AnonymousLogon	Returns the SECURITY_ANONYMOUS_LOGON_RID SID.
Sids::AuthenticatedUser	Returns the SECURITY_AUTHENTICATED_USER_RID SID.
Sids::BackupOps	Returns the DOMAIN_ALIAS_RID_BACKUP_OPS SID.
Sids::Batch	Returns the SECURITY_BATCH_RID SID.
Sids::CreatorGroup	Returns the SECURITY_CREATOR_GROUP_RID SID.
Sids::CreatorGroupServer	Returns the SECURITY_CREATOR_GROUP_SERVER_RID SID.
Sids::CreatorOwner	Returns the SECURITY_CREATOR_OWNER_RID SID.
Sids::CreatorOwnerServer	Returns the SECURITY_CREATOR_OWNER_SERVER_RID SID.
Sids::Dialup	Returns the SECURITY_DIALUP_RID SID.
Sids::Guests	Returns the DOMAIN_ALIAS_RID_GUESTS SID.
Sids::Interactive	Returns the SECURITY_INTERACTIVE_RID SID.
Sids::Local	Returns the SECURITY_LOCAL_RID SID.
Sids::Network	Returns the SECURITY_NETWORK_RID SID.
Sids::NetworkService	Returns the SECURITY_NETWORK_SERVICE_RID SID.
Sids::Null	Returns the SECURITY_NULL_RID SID.
Sids::PreW2KAccess	Returns the DOMAIN_ALIAS_RID_PREW2KCOMPACCESS SID.
Sids::PowerUsers	Returns the DOMAIN_ALIAS_RID_POWER_USERS SID.
Sids::PrintOps	Returns the DOMAIN_ALIAS_RID_PRINT_OPS SID.
Sids::Proxy	Returns the SECURITY_PROXY_RID SID.
Sids::RasServers	Returns the DOMAIN_ALIAS_RID_RAS_SERVERS SID.
Sids::Replicator	Returns the DOMAIN_ALIAS_RID_REPLICATOR SID.
Sids::RestrictedCode	Returns the SECURITY_RESTRICTED_CODE_RID SID.

Sids::Self	Returns the SECURITY_PRINCIPAL_SELF_RID SID.
Sids::ServerLogon	Returns the SECURITY_SERVER_LOGON_RID SID.
Sids::Service	Returns the SECURITY_SERVICE_RID SID.
Sids::System	Returns the SECURITY_LOCAL_SYSTEM_RID SID.
Sids::SystemOps	Returns the DOMAIN_ALIAS_RID_SYSTEM_OPS SID.
Sids::TerminalServer	Returns the SECURITY_TERMINAL_SERVER_RID SID.
Sids::Users	Returns the DOMAIN_ALIAS_RID_USERS SID.
Sids::World	Returns the SECURITY_WORLD_RID SID.

See Also

Other Resources

ATL Functions

Server Registration Global Functions

These functions provide support for registering and unregistering server objects in the object map.

AtlComModuleRegisterServer	This function is called to register every object in the object map.
AtlComModuleUnregisterServer	This function is called to unregister every object in the object map.
AtlComModuleRegisterClassObjects	This function is called to register class objects.
AtlComModuleRevokeClassObjects	This function is called to revoke class objects from a COM module.
AtlComModuleGetClassObject	This function is called to get the class object.

See Also

Other Resources

ATL Functions

WinModule Global Functions

These functions provide support for **_AtlCreateWndData** structure operations.

AtlWinModuleAddCreateWndData	This function is used to initialize and add an **_AtlCreateWndData** structure.
AtlWinModuleExtractCreateWndData	Call this function to extract an existing **_AtlCreateWndData** structure.

See Also

Other Resources

ATL Functions

ATL Functions Alphabetical Reference

In this section, reference topics for the ATL global functions are organized alphabetically. To find a particular function by category, see ATL Functions.

Function	Description
AtlAdvise	Creates a connection between an object's connection point and a client's sink.
AtlAdviseSinkMap	Call this function to advise or unadvise all entries in the object's sink event map.
AtlAxAttachControl	Attaches a previously created control to the specified window.
AtlAxCreateControlEx	Creates an ActiveX control, initializes it, and hosts it in the specified window. An interface pointer and event sink for the new control can also be created.
AtlAxCreateControlLicEx	Creates a licensed ActiveX control, initializes it, and hosts it in the specified window. An interface pointer and event sink for the new control can also be created.
AtlAxCreateControlLic	Creates a licensed ActiveX control, initializes it, and hosts it in the specified window.
AtlAxCreateControl	Creates an ActiveX control, initializes it, and hosts it in the specified window.
AtlAxCreateDialog	Creates a modeless dialog box from a dialog template provided by the user.
AtlAxDialogBox	Creates a modal dialog box from a dialog template provided by the user.
AtlAxGetControl	Obtains a direct interface pointer to the control contained inside a specified window given its handle.
AtlAxGetHost	Obtains a direct interface pointer to the container for a specified window (if any), given its handle.
AtlAxWinInit	This function initializes ATL's control hosting code by registering the **"AtlAxWin80"** and **"AtlAxWinLic80"** window classes plus a couple of custom window messages.
AtlAxWinTerm	This function uninitializes ATL's control hosting code by unregistering the **"AtlAxWin80"** and **"AtlAxWinLic80"** window classes.
AtlComModuleGetClassObject	This function is called to return the class factory.
AtlComModuleRegisterClassObjects	This function is called to register class objects.
AtlComModuleRegisterServer	This function is called to register every object in the object map.
AtlComModuleRevokeClassObjects	This function is called to remove the class factory/factories from the Running Object Table.
AtlComModuleUnregisterServer	This function is called to unregister every object in the object map.
AtlComPtrAssign	Assigns an interface pointer to another interface pointer of the same type.
AtlComQIPtrAssign	Assigns an interface pointer to another interface pointer of a different type.

AtlCreateTargetDC	Creates a device context for the device specified in the DVTARGETDEVICE structure.
AtlFreeMarshalStream	Releases the marshal data in the stream, then releases the stream pointer.
AtlGetDacl	Call this function to retrieve the discretionary access-control list (DACL) information of a specified object.
AtlGetGroupSid	Call this function to retrieve the group security identifier (SID) of an object.
AtlGetObjectSourceInterface	Call this function to retrieve information about the default source interface of an object.
AtlGetOwnerSid	Call this function to retrieve the owner security identifier (SID) of an object.
AtlGetSacl	Call this function to retrieve the system access-control list (SACL) information of a specifi ed object.
AtlGetSecurityDescriptor	Call this function to retrieve the security descriptor of a given object.
AtlHiMetricToPixel	Converts an object's size in HIMETRIC units (each unit is 0.01 millimeter) to a size in pixel s on the screen device.
AtlHresultFromLastError	Returns the calling thread's last-error code value in the form of an HRESULT.
AtlHresultFromWin32	Converts a Win32 error code into an HRESULT.
AtlInternalQueryInterface	Retrieves a pointer to the requested interface.
AtlLoadTypeLib	This function is called to load a type library.
AtlMarshalPtrInProc	Creates a new stream object, writes the CLSID of the proxy to the stream, and marshals t he specified interface pointer by writing the data needed to initialize the proxy into the st ream.
AtlModuleRegisterServer	Registers every object in the object map.
AtlModuleRegisterTypeLib	Registers a type library.
AtlModuleUnregisterServerEx	Unregisters every object in the object map.
AtlModuleUnregisterServer	Unregisters every object in the object map. It is similar to **AtlModuleUnregisterServer Ex** except that it cannot unregister the type library.
AtlModuleUnregisterTypeLib	Unregisters a type library.
AtlPixelToHiMetric	Converts an object's size in pixels on the screen device to a size in HIMETRIC units (each unit is 0.01 millimeter).
AtlRegisterTypeLib	This function is called to register a type library.
AtlReportError	Sets up the IErrorInfo interface to provide error information to clients of the object.
AtlSetChildSite	Call this function to set the site of the child object to the **IUnknown** of the parent object.

AtlSetDacl	Call this function to set the discretionary access-control list (DACL) information of a speci fied object.
AtlSetGroupSid	Call this function to set the group security identifier (SID) of an object.
AtlSetOwnerSid	Call this function to set the owner security identifier (SID) of an object.
AtlSetSacl	Call this function to set the system access-control list (SACL) information of a specified o bject.
AtlThrowLastWin32	Call this function to signal an error based on the result of the Windows function **GetLast Error**.
AtlThrow	Call this function to signal an error based on a **HRESULT** status code.
AtlTraceLoadSettings	Call this function to load trace settings from a file.
AtlTraceSaveSettings	Call this function to save the current trace settings to a file.
AtlUnadvise	Terminates the connection established through AtlAdvise.
AtlUnmarshalPtr	Converts the stream's marshaling data into an interface pointer that can be used by the c lient.
AtlUnRegisterTypeLib	This function is called to unregister a type library.
AtlUpdateRegistryFromResourceD	This function is called to update the registry from the supplied resource.
AtlWaitWithMessageLoop	Waits for the object to be signaled, meanwhile dispatching window messages as needed.
AtlWinModuleAddCreateWndData	This function is used to initialize and add an **_AtlCreateWndData** structure.
AtlWinModuleExtractCreateWndData	Call this function to extract an existing **_AtlCreateWndData** structure.
InlineIsEqualIUnknown	Call this function, for the special case of testing for **IUnknown**.
RegistryDataExchange	This function is called to read from, or write to, the system registry.
Sids::AccountOps	Returns the DOMAIN_ALIAS_RID_ACCOUNT_OPS SID.
Sids::Admins	Returns the DOMAIN_ALIAS_RID_ADMINS SID.
Sids::AnonymousLogon	Returns the SECURITY_ANONYMOUS_LOGON_RID SID.
Sids::AuthenticatedUser	Returns the SECURITY_AUTHENTICATED_USER_RID SID.
Sids::BackupOps	Returns the DOMAIN_ALIAS_RID_BACKUP_OPS SID.
Sids::Batch	Returns the SECURITY_BATCH_RID SID.
Sids::CreatorGroupServer	Returns the SECURITY_CREATOR_GROUP_SERVER_RID SID.
Sids::CreatorGroup	Returns the SECURITY_CREATOR_GROUP_RID SID.

Sids::CreatorOwnerServer	Returns the SECURITY_CREATOR_OWNER_SERVER_RID SID.
Sids::CreatorOwner	Returns the SECURITY_CREATOR_OWNER_RID SID.
Sids::Dialup	Returns the SECURITY_DIALUP_RID SID.
Sids::Guests	Returns the DOMAIN_ALIAS_RID_GUESTS SID.
Sids::Interactive	Returns the SECURITY_INTERACTIVE_RID SID.
Sids::Local	Returns the SECURITY_LOCAL_RID SID.
Sids::Network	Returns the SECURITY_NETWORK_RID SID.
Sids::NetworkService	Returns the SECURITY_NETWORK_SERVICE_RID SID.
Sids::Null	Returns the SECURITY_NULL_RID SID.
Sids::PowerUsers	Returns the DOMAIN_ALIAS_RID_POWER_USERS SID.
Sids::PreW2KAccess	Returns the DOMAIN_ALIAS_RID_PREW2KCOMPACCESS SID.
Sids::PrintOps	Returns the DOMAIN_ALIAS_RID_PRINT_OPS SID.
Sids::Proxy	Returns the SECURITY_PROXY_RID SID.
Sids::RasServers	Returns the DOMAIN_ALIAS_RID_RAS_SERVERS SID.
Sids::Replicator	Returns the DOMAIN_ALIAS_RID_REPLICATOR SID.
Sids::RestrictedCode	Returns the SECURITY_RESTRICTED_CODE_RID SID.
Sids::Self	Returns the SECURITY_PRINCIPAL_SELF_RID SID.
Sids::ServerLogon	Returns the SECURITY_SERVER_LOGON_RID SID.
Sids::Service	Returns the SECURITY_SERVICE_RID SID.
Sids::SystemOps	Returns the DOMAIN_ALIAS_RID_SYSTEM_OPS SID.
Sids::System	Returns the SECURITY_LOCAL_SYSTEM_RID SID.
Sids::TerminalServer	Returns the SECURITY_TERMINAL_SERVER_RID SID.
Sids::Users	Returns the DOMAIN_ALIAS_RID_USERS SID.
Sids::World	Returns the SECURITY_WORLD_RID SID.

AtlAdvise

Creates a connection between an object's connection point and a client's sink.

```
HRESULT AtlAdvise(
   IUnknown* pUnkCP,
   IUnknown* pUnk,
   const IID& iid,
   LPDWORD pdw
);
```

Parameters

pUnkCP

[in] A pointer to the **IUnknown** of the object the client wants to connect with.

pUnk

[in] A pointer to the client's **IUnknown**.

iid

[in] The GUID of the connection point. Typically, this is the same as the outgoing interface managed by the connection point.

pdw

[out] A pointer to the cookie that uniquely identifies the connection.

Return Value

A standard HRESULT value.

Remarks

The sink implements the outgoing interface supported by the connection point. The client uses the *pdw* cookie to remove the connection by passing it to AtlUnadvise.

Example

```
LPUNKNOWN m_pSourceUnk;
LPUNKNOWN m_pSinkUnk;
DWORD m_dwCustCookie;

// create source object
HRESULT hr = CoCreateInstance (CLSID_MyComponent, NULL, CLSCTX_ALL,
         IID_IUnknown, (LPVOID*)&m_pSourceUnk);
_ASSERT (SUCCEEDED (hr));

// Create sink object.  CMySink is a CComObjectRootEx-derived class
// that implements the event interface methods.
CComObject<CMySink> *pSinkClass;
CComObject<CMySink>::CreateInstance (&pSinkClass);
hr = pSinkClass->QueryInterface (IID_IUnknown, (LPVOID*)&m_pSinkUnk);
_ASSERT (SUCCEEDED (hr));

hr = AtlAdvise (m_pSourceUnk, m_pSinkUnk, IID__IMyEvent, &m_dwCustCookie);
_ASSERT (SUCCEEDED (hr));
```

See Also

Other Resources

Connection Point Global Functions

AtlAdviseSinkMap

Call this function to advise or unadvise all entries in the object's sink event map.

```
HRESULT AtlAdviseSinkMap(
   T* pT,
   bool bAdvise
);
```

Parameters

pT

[in] A pointer to the object containing the sink map.

bAdvise

[in] **true** if all sink entries are to be advised; **false** if all sink entries are to be unadvised.

Return Value

A standard HRESULT value.

Example

```
class CMyDlg :
   public CAxDialogImpl<CMyDlg>,
   public IDispEventImpl<IDC_MASKEDBOX1, CMyDlg>
{
BEGIN_MSG_MAP(CMyDlg)
   MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
END_MSG_MAP()

   LRESULT OnInitDialog(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
   {
      // hook up connection point
      AtlAdviseSinkMap (this, TRUE);
      return 1;
   }
...
}
```

See Also

Other Resources

Connection Point Global Functions

AtlAxAttachControl

Attaches a previously created control to the specified window.

```
ATLAPI AtlAxAttachControl(
   IUnknown* pControl,
   HWND hWnd,
   IUnknown** ppUnkContainer
);
```

Parameters

pControl

[in] A pointer to the **IUnknown** of the control.

hWnd

[in] Handle to the window that will host the control.

ppUnkContainer

[out] A pointer to a pointer to the **IUnknown** of the container object.

Return Value

One of the standard HRESULT values.

Remarks

Use AtlAxCreateControlEx and AtlAxCreateControl to simultaneously create and attach a control.

Note
The control object being attached must be correctly initialized before calling **AtlAxAttachControl**.

See Also

Concepts

Composite Control Fundamentals

AtlAxCreateControl

Creates an ActiveX control, initializes it, and hosts it in the specified window.

```
ATLAPI AtlAxCreateControl(
   LPCOLESTR lpszName,
   HWND hWnd,
   IStream* pStream,
   IUnknown** ppUnkContainer
);
```

Parameters

lpszName

A pointer to a string to be passed to the control. Must be formatted in one of the following ways:

- A ProgID such as "MSCAL.Calendar.7"
- A CLSID such as "{8E27C92B-1264-101C-8A2F-040224009C02}"
- A URL such as "http://www.microsoft.com"
- A reference to an Active document such as "file://\\Documents\MyDoc.doc"
- A fragment of HTML such as "MSHTML:<HTML><BODY>This is a line of text</BODY></HTML>"

Note
"MSHTML:" must precede the HTML fragment so that it is designated as being an MSHTML stream.

hWnd

[in] Handle to the window that the control will be attached to.

pStream

[in] A pointer to a stream that is used to initialize the properties of the control. Can be **NULL**.

ppUnkContainer

[out] The address of a pointer that will receive the **IUnknown** of the container. Can be **NULL**.

Return Value

One of the standard HRESULT values.

Remarks

This global function gives you the same result as calling AtlAxCreateControlEx(*lpszName*, *hWnd*, *pStream*, **NULL, NULL, NULL, NULL**);.

To create a licensed ActiveX control, see AtlAxCreateControlLic.

See Also

Reference

CAxWindow::CreateControl

Concepts

Composite Control Fundamentals

Other Resources

Composite Control Global Functions

AtlAxCreateControlEx

Creates an ActiveX control, initializes it, and hosts it in the specified window. An interface pointer and event sink for the new control can also be created.

```
ATLAPI AtlAxCreateControlEx(
   LPCOLESTR lpszName,
   HWND hWnd,
   IStream* pStream,
   IUnknown** ppUnkContainer,
   IUnknown** ppUnkControl,
   REFIID iidSink = IID_NULL,
   IUnknown* punkSink = NULL
);
```

Parameters

lpszName

A pointer to a string to be passed to the control. Must be formatted in one of the following ways:

- A ProgID such as "MSCAL.Calendar.7"
- A CLSID such as "{8E27C92B-1264-101C-8A2F-040224009C02}"
- A URL such as "http://www.microsoft.com"
- A reference to an Active document such as "file://\\Documents\MyDoc.doc"
- A fragment of HTML such as "MSHTML:<HTML><BODY>This is a line of text</BODY></HTML>"

Note
"MSHTML:" must precede the HTML fragment so that it is designated as being an MSHTML stream.

hWnd

[in] Handle to the window that the control will be attached to.

pStream

[in] A pointer to a stream that is used to initialize the properties of the control. Can be **NULL**.

ppUnkContainer

[out] The address of a pointer that will receive the **IUnknown** of the container. Can be **NULL**.

ppUnkControl

[out] The address of a pointer that will receive the **IUnknown** of the created control. Can be **NULL**.

iidSink

The interface identifier of an outgoing interface on the contained object.

punkSink

A pointer to the **IUnknown** interface of the sink object to be connected to the connection point specified by *iidSink* on the contained object after the contained object has been successfully created.

Return Value

One of the standard HRESULT values.

Remarks

AtlAxCreateControlEx is similar to AtlAxCreateControl but also allows you to receive an interface pointer to the newly created control and set up an event sink to receive events fired by the control.

To create a licensed ActiveX control, see AtlAxCreateControlLicEx.

See Also

Reference

CAxWindow::CreateControlEx

Concepts

Composite Control Fundamentals

Other Resources

Composite Control Global Functions

AtlAxCreateControlLic

Creates a licensed ActiveX control, initializes it, and hosts it in the specified window.

```
ATLAPI AtlAxCreateControlLic(
   LPCOLESTR lpszName,
   HWND hWnd,
   IStream* pStream,
   IUnknown** ppUnkContainer,
   BSTR bstrLic = NULL
);
```

Parameters

lpszName

A pointer to a string to be passed to the control. Must be formatted in one of the following ways:

- A ProgID such as "MSCAL.Calendar.7"
- A CLSID such as "{8E27C92B-1264-101C-8A2F-040224009C02}"
- A URL such as "http://www.microsoft.com"
- A reference to an Active document such as "file://\\Documents\MyDoc.doc"
- A fragment of HTML such as "MSHTML:<HTML><BODY>This is a line of text</BODY></HTML>"

Note
"MSHTML:" must precede the HTML fragment so that it is designated as being an MSHTML stream.

hWnd

Handle to the window that the control will be attached to.

pStream

A pointer to a stream that is used to initialize the properties of the control. Can be **NULL**.

ppUnkContainer

The address of a pointer that will receive the **IUnknown** of the container. Can be **NULL**.

bstrLic

The BSTR containing the license for the control.

Return Value

One of the standard HRESULT values.

Example

See Hosting ActiveX Controls Using ATL AXHost for a sample of how to use **AtlAxCreateControlLic**.

See Also

Reference

AtlAxCreateControl

CAxWindow2T::CreateControlLic

Concepts

Composite Control Fundamentals

Other Resources

Composite Control Global Functions

AtlAxCreateControlLicEx

Creates a licensed ActiveX control, initializes it, and hosts it in the specified window. An interface pointer and event sink for the new control can also be created.

```
ATLAPI AtlAxCreateControlLicEx(
   LPCOLESTR lpszName,
   HWND hWnd,
   IStream* pStream,
   IUnknown** ppUnkContainer,
   IUnknown** ppUnkControl,
   REFIID iidSink = IID_NULL,
   IUnknown* punkSink = NULL,
   BSTR bstrLic = NULL
);
```

Parameters

lpszName

A pointer to a string to be passed to the control. Must be formatted in one of the following ways:

- A ProgID such as "MSCAL.Calendar.7"
- A CLSID such as "{8E27C92B-1264-101C-8A2F-040224009C02}"
- A URL such as "http://www.microsoft.com"
- A reference to an Active document such as "file://\\Documents\MyDoc.doc"
- A fragment of HTML such as "MSHTML:<HTML><BODY>This is a line of text</BODY></HTML>"

Note
"MSHTML:" must precede the HTML fragment so that it is designated as being an MSHTML stream.

hWnd

Handle to the window that the control will be attached to.

pStream

A pointer to a stream that is used to initialize the properties of the control. Can be **NULL**.

ppUnkContainer

The address of a pointer that will receive the **IUnknown** of the container. Can be **NULL**.

ppUnkControl

[out] The address of a pointer that will receive the **IUnknown** of the created control. Can be **NULL**.

iidSink

The interface identifier of an outgoing interface on the contained object.

punkSink

A pointer to the **IUnknown** interface of the sink object to be connected to the connection point specified by *iidSink* on the contained object after the contained object has been successfully created.

bstrLic

The BSTR containing the license for the control.

Return Value

One of the standard HRESULT values.

Remarks

AtlAxCreateControlLicEx is similar to AtlAxCreateControlLic but also allows you to receive an interface pointer to the newly created control and set up an event sink to receive events fired by the control.

Example

See Hosting ActiveX Controls Using ATL AXHost for a sample of how to use **AtlAxCreateControlLicEx**.

See Also

Reference

AtlAxCreateControl

CAxWindow2T::CreateControlLicEx

Concepts

Composite Control Fundamentals

Other Resources

Composite Control Global Functions

AtlAxCreateDialog

Creates a modeless dialog box from a dialog template provided by the user.

```
ATLAPI_(HWND) AtlAxCreateDialog(
   HINSTANCE hInstance,
   LPCWSTR lpTemplateName,
   HWND hWndParent,
   DLGPROC lpDialogProc,
   LPARAM dwInitParam
);
```

Parameters

hInstance

[in] Identifies an instance of the module whose executable file contains the dialog box template.

lpTemplateName

[in] Identifies the dialog box template. This parameter is either the pointer to a null-terminated character string that specifies the name of the dialog box template or an integer value that specifies the resource identifier of the dialog box template. If the parameter specifies a resource identifier, its high-order word must be zero and its low-order word must contain the identifier. You can use the MAKEINTRESOURCE macro to create this value.

hWndParent

[in] Identifies the window that owns the dialog box.

lpDialogProc

[in] Points to the dialog box procedure. For more information about the dialog box procedure, see DialogProc.

dwInitParam

[in] Specifies the value to pass to the dialog box in the **lParam** parameter of the **WM_INITDIALOG** message.

Return Value

One of the standard HRESULT values.

Remarks

The resulting dialog box can contain ActiveX controls.

See CreateDialog and CreateDialogParam in the *Platform SDK*.

See Also

Reference

AtlAxDialogBox

Concepts

Composite Control Fundamentals

Other Resources

Composite Control Global Functions

AtlAxDialogBox

Creates a modal dialog box from a dialog template provided by the user.

```
ATLAPI_(int) AtlAxDialogBox(
   HINSTANCE hInstance,
   LPCWSTR lpTemplateName,
   HWND hWndParent,
   DLGPROC lpDialogProc,
   LPARAM dwInitParam
);
```

Parameters

hInstance

[in] Identifies an instance of the module whose executable file contains the dialog box template.

lpTemplateName

[in] Identifies the dialog box template. This parameter is either the pointer to a null-terminated character string that specifies the name of the dialog box template or an integer value that specifies the resource identifier of the dialog box template. If the parameter specifies a resource identifier, its high-order word must be zero and its low-order word must contain the identifier. You can use the MAKEINTRESOURCE macro to create this value.

hWndParent

[in] Identifies the window that owns the dialog box.

lpDialogProc

[in] Points to the dialog box procedure. For more information about the dialog box procedure, see DialogProc.

dwInitParam

[in] Specifies the value to pass to the dialog box in the **lParam** parameter of the **WM_INITDIALOG** message.

Return Value

One of the standard HRESULT values.

Remarks

To use **AtlAxDialogBox** with a dialog template that contains an ActiveX control, specify a valid **CLSID**, **APPID** or URL string as the *text* field of the **CONTROL** section of the dialog resource, along with "AtlAxWin80" as the *class name* field under the same section. The following demonstrates what a valid **CONTROL** section might look like:

```
CONTROL    "{04FE35E9-ADBC-4f1d-83FE-8FA4D1F71C7F}", IDC_TEST,
           "AtlAxWin80", WS_GROUP | WS_TABSTOP, 0, 0, 100, 100
```

For more information on editing resource scripts, see Opening a Resource Script File in Text Format. For more information on control resource-definition statements, see Common Control Parameters under *Platform SDK: SDK Tools*.

For more information on dialog boxes in general, refer to DialogBox and CreateDialogParam in the *Platform SDK*.

See Also

Reference

AtlAxCreateDialog

Concepts

Composite Control Fundamentals

Other Resources

Composite Control Global Functions

AtlAxGetControl

Obtains a direct interface pointer to the control contained inside a specified window given its handle.

```
ATLAPI AtlAxGetControl(
   HWND h,
   IUnknown** pp
);
```

Parameters

h

[in] A handle to the window that is hosting the control.

pp

[out] The **IUnknown** of the control being hosted.

Return Value

One of the standard HRESULT values.

See Also

Concepts

Composite Control Fundamentals

Other Resources

Composite Control Global Functions

AtlAxGetHost

Obtains a direct interface pointer to the container for a specified window (if any), given its handle.

```
ATLAPI AtlAxGetHost(
   HWND h,
   IUnknown** pp
);
```

Parameters

h

[in] A handle to the window that is hosting the control.

pp

[out] The **IUnknown** of the container of the control.

Return Value

One of the standard HRESULT values.

See Also

Concepts

Composite Control Fundamentals

Other Resources

Composite Control Global Functions

AtlAxWinInit

This function initializes ATL's control hosting code by registering the **"AtlAxWin80"** and **"AtlAxWinLic80"** window classes plus a couple of custom window messages.

```
ATLAPI_(BOOL) AtlAxWinInit( );
```

Return Value

Nonzero if the initialization of the control hosting code was successful; otherwise **FALSE**.

Remarks

This function must be called before using the ATL control hosting API. Following a call to this function, the **"AtlAxWin"** window class can be used in calls to CreateWindow or CreateWindowEx, as described in the *Platform SDK*.

For more details on when to call this function, see When Do I Need To Call AtlAxWinInit?

See Also

Reference

AtlAxWinTerm

Concepts

Composite Control Fundamentals

Other Resources

Composite Control Global Functions

AtlAxWinTerm

This function uninitializes ATL's control hosting code by unregistering the **"AtlAxWin80"** and **"AtlAxWinLic80"** window classes.

```
inline BOOL AtlAxWinTerm( );
```

Return Value

Always returns **TRUE**.

Remarks

This function simply calls UnregisterClass as described in the *Platform SDK*.

Call this function to clean up after all existing host windows have been destroyed if you called AtlAxWinInit and you no longer need to create host windows. If you don't call this function, the window class will be unregistered automatically when the process terminates.

See Also

Reference

AtlAxWinInit

Concepts

Composite Control Fundamentals

Other Resources

Composite Control Global Functions

AtlComModuleGetClassObject

This function is called to return the class factory.

```
ATLINLINE ATLAPI AtlComModuleGetClassObject(
   _ATL_COM_MODULE * pComModule,
   REFCLSID rclsid,
   REFIID riid,
   LPVOID* ppv
);
```

Parameters

pComModule

Pointer to the COM module.

rclsid

The CLSID of the object to be created.

riid

The IID of the requested interface.

ppv

A pointer to the interface pointer identified by *riid*. If the object does not support this interface, *ppv* is set to NULL.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This helper function is utilized by CComModule::GetClassObject (obsolete in ATL 7.0) and CAtlDllModuleT::GetClassObject.

See Also

Other Resources

Server Registration Global Functions

AtlComModuleRegisterClassObjects

This function is called to register class objects.

```
ATLINLINE ATLAPI AtlComModuleRegisterClassObjects(
   _ATL_COM_MODULE * pComModule,
   DWORD dwClsContext,
   DWORD dwFlags
);
```

Parameters

pComModule

Pointer to the COM module.

dwClsContext

Specifies the context in which the class object is to be run. Possible values are CLSCTX_INPROC_SERVER, CLSCTX_INPROC_HANDLER, or CLSCTX_LOCAL_SERVER. See CLSCTX for more details.

dwFlags

Determines the connection types to the class object. Possible values are REGCLS_SINGLEUSE, REGCLS_MULTIPLEUSE, or REGCLS_MULTI_SEPARATE. See REGCLS for more details.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This helper function is utilized by CComModule::RegisterClassObjects (obsolete in ATL 7.0) and CAtlExeModuleT::RegisterClassObjects.

See Also

Other Resources

Server Registration Global Functions

AtlComModuleRegisterServer

This function is called to register every object in the object map.

```
ATLINLINE ATLAPI AtlComModuleRegisterServer(
   _ATL_COM_MODULE* pComModule,
   BOOL bRegTypeLib,
   const CLSID* pCLSID
);
```

Parameters

pComModule

Pointer to the COM module.

bRegTypeLib

TRUE if the type library is to be registered.

pCLSID

Points to the CLSID of the object to be registered. If NULL, all objects in the object map will be registered.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

AtlComModuleRegisterServer walks the ATL autogenerated object map and registers each object in the map. If *pCLSID* is not NULL, then only the object referred to by *pCLSID* is registered; otherwise all of the objects are registered.

This function is called by CAtlComModule::RegisterServer.

See Also

Reference

CAtlComModule::RegisterServer

Other Resources

Server Registration Global Functions

AtlComModuleRevokeClassObjects

This function is called to remove the class factory/factories from the Running Object Table.

```
ATLINLINE ATLAPI AtlComModuleRevokeClassObjects(
   _ATL_COM_MODULE * pComModule
);
```

Parameters

pComModule

Pointer to the COM module.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This helper function is utilized by CComModule::RevokeClassObjects (obsolete in ATL 7.0) and CAtlExeModuleT::RevokeClassObjects.

See Also

Other Resources

Server Registration Global Functions

AtlComModuleUnregisterServer

This function is called to unregister every object in the object map.

```
ATLINLINE ATLAPI AtlComModuleUnregisterServer(
   _ATL_COM_MODULE* pComModule,
   BOOL bUnRegTypeLib,
   const CLSID* pCLSID
);
```

Parameters

pComModule

Pointer to the COM module.

bUnRegTypeLib

TRUE if the type library is to be registered.

pCLSID

Points to the CLSID of the object to be unregistered. If NULL all objects in the object map will be unregistered.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

AtlComModuleUnregisterServer walks the ATL object map and unregisters each object in the map. If *pCLSID* is not NULL, then only the object referred to by *pCLSID* is unregistered; otherwise all of the objects are unregistered.

This function is called by CAtlComModule::UnregisterServer.

See Also

Reference

CAtlComModule::UnregisterServer

Other Resources

Server Registration Global Functions

AtlComPtrAssign

Assigns an interface pointer to another interface pointer of the same type.

```
ATLINLINE ATLAPI_(IUnknown*) AtlComPtrAssign(
   IUnknown** pp,
   IUnknown* lp
);
```

Parameters

pp

Address of an **IUnknown** pointer to which to assign another pointer.

lp

An **IUnknown** pointer of the same type as the one in *pp*. This is assigned to the pointer in *pp*.

Return Value

Returns NULL on error; otherwise returns *lp*.

Remarks

This function copies an interface pointer using COM reference counting rules. It calls **Release** on the interface pointed to by *pp* and then assigns *lp* to it after incrementing the reference count.

This smart pointer helper function is used by CComPtr Class and CComQIPtr Class.

See Also

Reference

CComPtr Class
CComQIPtr Class

AtlComQIPtrAssign

Assigns an interface pointer to another interface pointer of a different type.

```
ATLINLINE ATLAPI_(IUnknown*) AtlComQIPtrAssign(
   IUnknown** pp,
   IUnknown* lp,
   REFIID riid
);
```

Parameters

pp

Address of an **IUnknown** pointer to which to assign another pointer.

lp

An **IUnknown** pointer of a type different from the one in *pp*. This is assigned to the pointer in *pp*.

riid

The IID of the requested interface.

Return Value

Returns NULL on error; otherwise returns *lp*.

Remarks

This function copies an interface pointer using COM reference counting rules. It calls **QueryInterface** on *lp*, specifying *riid* to obtain the required interface. It calls **Release** on the interface pointed to by *pp* and then assigns the pointer returned by **QueryInterface**.

This smart pointer helper function is used by CComPtr Class and CComQIPtr Class.

See Also

Reference

CComPtr Class
CComQIPtr Class

AtlCreateTargetDC

Creates a device context for the device specified in the DVTARGETDEVICE structure.

```
HDC AtlCreateTargetDC(
   HDC hdc,
   DVTARGETDEVICE* ptd
);
```

Parameters

hdc

[in] The existing handle of a device context, or **NULL**.

ptd

[in] A pointer to the **DVTARGETDEVICE** structure that contains information about the target device.

Return Value

Returns the handle to a device context for the device specified in the **DVTARGETDEVICE**. If no device is specified, returns the handle to the default display device.

Remarks

If the structure is **NULL** and *hdc* is **NULL**, creates a device context for the default display device.

If *hdc* is not **NULL** and *ptd* is **NULL**, the function returns the existing *hdc*.

See Also

Other Resources

Device Context Global Functions

AtlFreeMarshalStream

Releases the marshal data in the stream, then releases the stream pointer.

```
HRESULT AtlFreeMarshalStream(
   IStream* pStream
);
```

Parameters

pStream

[in] A pointer to the **IStream** interface on the stream used for marshaling.

Example

See the example for AtlMarshalPtrInProc.

See Also

Reference

AtlMarshalPtrInProc

Other Resources

Marshaling Global Functions

AtlGetDacl

Call this function to retrieve the discretionary access-control list (DACL) information of a specified object.

```
inline bool AtlGetDacl(
   HANDLE hObject,
   SE_OBJECT_TYPE ObjectType,
   CDacl* pDacl
) throw(   );
```

Parameters

hObject

Handle to the object for which to retrieve the security information.

ObjectType

Specifies a value from the SE_OBJECT_TYPE enumeration that indicates the type of object identified by the *hObject* parameter.

pDacl

Pointer to a DACL object which will contain the retrieved security information.

Return Value

Returns true on success, false on failure.

Remarks

In debug builds, an assertion error will occur if either *hObject* or *pDacl* is invalid.

See Also

Reference

AtlSetDacl

CDacl Class

Other Resources

Security Global Functions

AtlGetGroupSid

Call this function to retrieve the group security identifier (SID) of an object.

```
inline bool AtlGetGroupSid(
   HANDLE hObject,
   SE_OBJECT_TYPE ObjectType,
   CSid* pSid
) throw(...);
```

Parameters

hObject

Handle to the object from which to retrieve security information.

ObjectType

Specifies a value from the SE_OBJECT_TYPE enumeration that indicates the type of object identified by the *hObject* parameter.

pSid

Pointer to a **CSid** object which will contain the new security information.

Return Value

Returns true on success, false on failure.

See Also

Reference

AtlSetGroupSid

CSid Class

Other Resources

Security Global Functions

AtlGetObjectSourceInterface

Call this function to retrieve information about the default source interface of an object.

```
ATLAPI AtlGetObjectSourceInterface(
   IUnknown* punkObj,
   GUID* plibid,
   IID* piid,
   unsigned short* pdwMajor,
   unsigned short* pdwMinor
);
```

Parameters

punkObj

[in] A pointer to the object for which information is to be returned.

plibid

[out] A pointer to the LIBID of the type library containing the definition of the source interface.

piid

[out] A pointer to the interface ID of the object's default source interface.

pdwMajor

[out] A pointer to the major version number of the type library containing the definition of the source interface.

pdwMinor

[out] A pointer to the minor version number of the type library containing the definition of the source interface.

Return Value

A standard HRESULT value.

Remarks

AtlGetObjectSourceInterface can provide you with the interface ID of the default source interface, along with the LIBID and major and minor version numbers of the type library describing that interface.

Note

For this function to successfully retrieve the requested information, the object represented by *punkObj* must implement **IDispatch** (and return type information through **IDispatch::GetTypeInfo**) plus it must also implement either **IProvideClassInfo2** or **IPersist**. The type information for the source interface must be in the same type library as the type information for **IDispatch**.

Example

The example below shows how you might define an event sink class, `CEasySink`, that reduces the number of template arguments that you can pass to **IDispEventImpl** to the bare essentials. `EasyAdvise` and `EasyUnadvise` use **AtlGetObjectSourceInterface** to initialize the IDispEventImpl members before calling DispEventAdvise or DispEventUnadvise.

```
template <UINT nID, class T>
class CEasySink :
    public IDispEventImpl<nID, T>
{
public:
    HRESULT EasyAdvise(IUnknown* pUnk)
    {
        AtlGetObjectSourceInterface(pUnk,
           &m_libid, &m_iid, &m_wMajorVerNum, &m_wMinorVerNum);
        return DispEventAdvise(pUnk, &m_iid);
```

```
    }
    HRESULT EasyUnadvise(IUnknown* pUnk)
    {
        AtlGetObjectSourceInterface(pUnk,
            &m_libid, &m_iid, &m_wMajorVerNum, &m_wMinorVerNum);
        return DispEventUnadvise(pUnk, &m_iid);
    }
};
```

See Also

Reference

IDispEventImpl Class

IDispEventSimpleImpl Class

Other Resources

Composite Control Global Functions

AtlGetOwnerSid

Call this function to retrieve the owner security identifier (SID) of an object.

```
inline bool AtlGetOwnerSid(
   HANDLE hObject,
   SE_OBJECT_TYPE ObjectType,
   CSid* pSid
) throw(...);
```

Parameters

hObject

Handle to the object from which to retrieve security information.

ObjectType

Specifies a value from the SE_OBJECT_TYPE enumeration that indicates the type of object identified by the *hObject* parameter.

pSid

Pointer to a **CSid** object which will contain the new security information.

Return Value

Returns true on success, false on failure.

See Also

Reference

AtlSetOwnerSid

CSid Class

Other Resources

Security Global Functions

AtlGetSacl

Call this function to retrieve the system access-control list (SACL) information of a specified object.

```
inline bool AtlGetSacl(
   HANDLE hObject,
   SE_OBJECT_TYPE ObjectType,
   CSacl* pSacl,
   bool bRequestNeededPrivileges = true
) throw(...);
```

Parameters

hObject

Handle to the object from which to retrieve the security information.

ObjectType

Specifies a value from the SE_OBJECT_TYPE enumeration that indicates the type of object identified by the *hObject* parameter.

pSacl

Pointer to a SACL object which will contain the retrieved security information.

bRequestNeededPrivileges

If true, the function will attempt to enable the SE_SECURITY_NAME privilege, and restore it on completion.

Return Value

Returns true on success, false on failure.

Remarks

If **AtlGetSacl** is to be called many times on many different objects, it will be more efficient to enable the SE_SECURITY_NAME privilege once before calling the function, with *bRequestNeededPrivileges* set to false.

See Also

Reference

AtlSetSacl

CSacl Class

Other Resources

Security Global Functions

AtlGetSecurityDescriptor

Call this function to retrieve the security descriptor of a given object.

```
inline bool AtlGetSecurityDescriptor(
   LPCTSTR pszObjectName,
   SE_OBJECT_TYPE ObjectType,
   CSecurityDesc * pSecurityDescriptor,
   SECURITY_INFORMATION requestedInfo = OWNER_SECURITY_INFORMATION |
      GROUP_SECURITY_INFORMATION | DACL_SECURITY_INFORMATION |
      SACL_SECURITY_INFORMATION,
   bool bRequestNeededPrivileges = true
) throw(...);
```

Parameters

pszObjectName

Pointer to a null-terminated string that specifies the name of the object from which to retrieve security information.

ObjectType

Specifies a value from the SE_OBJECT_TYPE enumeration that indicates the type of object identified by the *pszObjectName* parameter.

pSecurityDescriptor

The object which receives the requested security descriptor.

requestedInfo

A set of SECURITY_INFORMATION bit flags that indicate the type of security information to retrieve. This parameter can be a combination of the following values.

bRequestNeededPrivileges

If true, the function will attempt to enable the SE_SECURITY_NAME privilege, and restore it on completion.

Return Value

Returns true on success, false on failure.

Remarks

If **AtlGetSecurityDescriptor** is to be called many times on many different objects, it will be more efficient to enable the SE_SECURITY_NAME privilege once before calling the function, with *bRequestNeededPrivileges* set to false.

See Also

Other Resources

Security Global Functions

AtlGetVersion

Call this function to get the version of the ATL library that you are using.

```
ATLAPI_(DWORD) AtlGetVersion(void* pReserved);
```

Parameters

pReserved

A reserved pointer.

Return Value

Returns a **DWORD** integer value of the version of the ATL library that you are compiling or running.

Example

The function should be called as follows.

```
DWORD ver;
...
ver = AtlGetVersion( NULL );
```

Requirements

Header: atlbase.h

See Also

Other Resources

ATL Functions Alphabetical Reference

AtlHiMetricToPixel

Converts an object's size in HIMETRIC units (each unit is 0.01 millimeter) to a size in pixels on the screen device.

```
extern void AtlHiMetricToPixel(
   const SIZEL* lpSizeInHiMetric,
   LPSIZEL lpSizeInPix
);
```

Parameters

lpSizeInHiMetric

[in] Pointer to the size of the object in HIMETRIC units.

lpSizeInPix

[out] Pointer to where the object's size in pixels is to be returned.

Example

```
// m_sizeExtent is a member of CComControlBase that holds the
// control's extents in HIMETRIC units.
// Use AtlHiMetricToPixel to find the extent of the control in pixels.
AtlHiMetricToPixel (&m_sizeExtent, &sz);
ATLTRACE ("Width = %d, Height = %d\n", sz.cx, sz.cy);
```

See Also

Reference

AtlPixelToHiMetric

Other Resources

Pixel/HIMETRIC Conversion Global Functions

AtlHresultFromLastError

Returns the calling thread's last-error code value in the form of an HRESULT.

```
HRESULT AtlHresultFromLastError( );
```

Remarks

AtlHresultFromLastError calls **GetLastError** to obtain the last error and returns the error after converting it to an HRESULT using the **HRESULT_FROM_WIN32** macro.

Example

See the ShowImage Sample.

See Also

Reference

GetLastError
HRESULT_FROM_WIN32
AtlHresultFromWin32

Other Resources

Debugging and Error Reporting Global Functions

AtlHresultFromWin32

Converts a Win32 error code into an HRESULT.

```
AtlHresultFromWin32{
      DWORD error
};
```

Parameters

error

The error value to convert.

Remarks

Converts a Win32 error code into an HRESULT, using the macro **HRESULT_FROM_WIN32**.

Note
Instead of using **HRESULT_FROM_WIN32(GetLastError())**, use the function AtlHresultFromLastError.

See Also

Reference

HRESULT_FROM_WIN32

AtlHresultFromLastError

Other Resources

Debugging and Error Reporting Global Functions

AtlInternalQueryInterface

Retrieves a pointer to the requested interface.

```
HRESULT AtlInternalQueryInterface(
   void* pThis,
   const _ATL_INTMAP_ENTRY* pEntries,
   REFIID iid,
   void** ppvObject
);
```

Parameters

pThis

[in] A pointer to the object that contains the COM map of interfaces exposed to **QueryInterface**.

pEntries

[in] An array of **_ATL_INTMAP_ENTRY** structures that access a map of available interfaces.

iid

[in] The GUID of the interface being requested.

ppvObject

[out] A pointer to the interface pointer specified in *iid*, or **NULL** if the interface is not found.

Return Value

One of the standard HRESULT values.

Remarks

AtlInternalQueryInterface only handles interfaces in the COM map table. If your object is aggregated, **AtlInternalQueryInterface** does not delegate to the outer unknown. You can enter interfaces into the COM map table with the macro COM_INTERFACE_ENTRY or one of its variants.

Example

```
// TimerProc is a callback function passed to SetTimer()
VOID CALLBACK TimerProc(HWND hwnd, UINT uMsg, UINT idEvent, DWORD dwTime)
{
   LPDISPATCH pDisp = NULL;
   // gpMyCtrl is a global variable of type CMyCtrl*
   // _GetEntries() is a static function you get with BEGIN_COM_MAP()
   HRESULT hr = AtlInternalQueryInterface (gpMyCtrl,
   CMyCtrl::_GetEntries(), IID_IDispatch, (LPVOID*)&pDisp);
   ...
   pDisp->Release ();
}
```

See Also

Reference

CComObjectRootEx::InternalAddRef

CComObjectRootEx::InternalRelease

Other Resources

COM Map Global Functions

AtlLoadTypeLib

This function is called to load a type library.

```
ATLINLINE ATLAPI AtlLoadTypeLib(
   HINSTANCE hInstTypeLib,
   LPCOLESTR lpszIndex,
   BSTR* pbstrPath,
   ITypeLib** ppTypeLib
);
```

Parameters

hInstTypeLib

Handle to the module associated with the type library.

lpszIndex

String in the format "\\N", where N is the integer index of the type library resource. Can be NULL if no index is required.

pbstrPath

On successful return, contains the full path of the module associated with the type library.

ppTypeLib

On successful return, contains a pointer to a pointer to the loaded type library.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This helper function is utilized by AtlRegisterTypeLib and AtlUnRegisterTypeLib.

See Also

Reference

AtlUnRegisterTypeLib
AtlRegisterTypeLib

Other Resources

Server Registration Global Functions

AtlMarshalPtrInProc

Creates a new stream object, writes the CLSID of the proxy to the stream, and marshals the specified interface pointer by writing the data needed to initialize the proxy into the stream.

```
HRESULT AtlMarshalPtrInProc(
   IUnknown* pUnk,
   const IID& iid,
   IStream** ppStream
);
```

Parameters

pUnk

[in] A pointer to the interface to be marshaled.

iid

[in] The GUID of the interface being marshaled.

ppStream

[out] A pointer to the **IStream** interface on the new stream object used for marshaling.

Return Value

A standard HRESULT value.

Remarks

The **MSHLFLAGS_TABLESTRONG** flag is set so the pointer can be marshaled to multiple streams. The pointer can also be unmarshaled multiple times.

If marshaling fails, the stream pointer is released.

AtlMarshalPtrInProc can only be used on a pointer to an in-process object.

Example

```
//marshaling interface from one thread to another

//IStream ptr to hold serialized presentation of interface ptr
IStream* g_pStm;

//forward declaration
DWORD WINAPI ThreadProc(LPVOID lpParameter);

HRESULT WriteInterfacePtrToStream(IFoo *pFoo)
{
   //marshall the interface ptr to another thread
   //pFoo has to be pointer to actual object & not a proxy
   HRESULT hr = AtlMarshalPtrInProc(pFoo, IID_IFoo, &g_pStm);

   //m_dwThreadID is a DWORD holding thread ID of thread being created.
   CreateThread(NULL, 0, ThreadProc, NULL, 0, &m_dwThreadID);
   return hr;
}

DWORD WINAPI ThreadProc(LPVOID lpParameter)
{
   IFoo* rpFoo;

   //unmarshal IFoo ptr from the stream
   HRESULT hr = AtlUnmarshalPtr(g_pStm, IID_IFoo, &rpFoo);

   // use IFoo ptr to call its methods
   //...
```

```
    //...

    //release the stream if no other thread requires it
    //to unmarshal the IFoo interface pointer
    hr = AtlFreeMarshalStream(g_pStm);

    return hr;
}
```

See Also

Reference

AtlUnmarshalPtr

AtlFreeMarshalStream

MSHLFLAGS

Other Resources

Marshaling Global Functions

AtlModuleRegisterServer

Registers every object in the object map.

Note This function is obsolete in Visual C++ .NET 2002 and later but is available for backward compatibility with projects created with previous versions of Visual C++.

```
ATLAPI AtlModuleRegisterServer(
   _ATL_MODULE* pM,
   BOOL bUnRegTypeLib,
   const CLSID* pCLSID
);
```

Parameters

pM

Pointer to a **CComModule** class or derived class.

bUnRegTypeLib

TRUE if the type library is to be registered.

pCLSID

Points to the CLSID of the object to be registered. If NULL, all objects in the object map will be registered.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This function walks the ATL object map and registers each object in the map. If *pCLSID* is not NULL, then only the object referred to by *pCLSID* is registered; otherwise, all objects are registered.

This function is obsolete. Instead, use its replacement, AtlComModuleRegisterServer.

This helper function is used by CComModule::RegisterServer.

See Also

Reference

CComModule::RegisterServer

AtlComModuleRegisterServer

Other Resources

Server Registration Global Functions

AtlModuleRegisterTypeLib

Registers a type library.

Note This function is obsolete in Visual C++ .NET 2002 and later but is available for backward compatibility with projects created with previous versions of Visual C++.

```
ATLAPI AtlModuleRegisterTypeLib(
   _ATL_MODULE* pM,
   LPCOLESTR lpszIndex
);
```

Parameters

pM

Pointer to a **CComModule** class or derived class.

lpszIndex

String in the format "\\N", where N is the integer index of the type library resource. Can be NULL if no index is required.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This function is obsolete. Instead, use its replacement, AtlRegisterTypeLib.

This helper function is used by CComModule::RegisterTypeLib.

See Also

Reference

CComModule::RegisterTypeLib

Other Resources

Server Registration Global Functions

AtlModuleUnregisterServer

Unregisters every object in the object map. It is similar to **AtlModuleUnregisterServerEx** except that it cannot unregister the type library.

Note This function is obsolete in Visual C++ .NET 2002 and later but is available for backward compatibility with projects created with previous versions of Visual C++.

```
ATLAPI AtlModuleUnregisterServerEx(
   _ATL_MODULE* pM,
   const CLSID* pCLSID
);
```

Parameters

pM

Pointer to a **CComModule** class or derived class.

pCLSID

Points to the CLSID of the object to be unregistered. If NULL, all objects in the object map will be unregistered.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This function walks the ATL object map and unregisters each object in the map. If *pCLSID* is not NULL, then only the object referred to by *pCLSID* is unregistered; otherwise all of the objects are unregistered.

This function is obsolete. Instead, use its replacement, AtlComModuleUnregisterServer.

This helper function is used by CComModule::UnregisterServer.

See Also

Reference

CComModule::UnregisterServer

AtlComModuleUnregisterServer

Other Resources

Server Registration Global Functions

AtlModuleUnregisterServerEx

Unregisters every object in the object map.

Note This function is obsolete in Visual C++ .NET 2002 and later but is available for backward compatibility with projects created with previous versions of Visual C++.

```
ATLAPI AtlModuleUnregisterServerEx(
   _ATL_MODULE* pM,
   BOOL bUnRegTypeLib,
   const CLSID* pCLSID
);
```

Parameters

pM

Pointer to a **CComModule** class or derived class.

bUnRegTypeLib

TRUE if the type library is to be registered.

pCLSID

Points to the CLSID of the object to be unregistered. If NULL, all objects in the object map will be unregistered.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This function walks the ATL object map and unregisters each object in the map. If *pCLSID* is not NULL, then only the object referred to by *pCLSID* is unregistered; otherwise all of the objects are unregistered.

This function is obsolete. Instead, use its replacement, AtlComModuleUnregisterServer.

This helper function is used by CComModule::UnregisterServer.

See Also

Reference

CComModule::UnregisterServer

AtlComModuleUnregisterServer

Other Resources

Server Registration Global Functions

AtlModuleUnregisterTypeLib

Unregisters a type library.

Note This function is obsolete in Visual C++ .NET 2002 and later but is available for backward compatibility with projects created with previous versions of Visual C++.

```
ATLAPI AtlModuleUnregisterTypeLib(
   _ATL_MODULE* pM,
   LPCOLESTR lpszIndex
);
```

Parameters

pM

Pointer to a **CComModule** class or derived class.

lpszIndex

String in the format "\\N", where N is the integer index of the type library resource. Can be NULL if no index is required.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This function is obsolete. Instead, use its replacement, AtlUnRegisterTypeLib.

This helper function is used by CAtlComModule::UnRegisterTypeLib.

See Also

Reference

CAtlComModule::UnRegisterTypeLib

Other Resources

Server Registration Global Functions

AtlPixelToHiMetric

Converts an object's size in pixels on the screen device to a size in HIMETRIC units (each unit is 0.01 millimeter).

```
extern void AtlPixelToHiMetric(
   const SIZEL* lpSizeInPix,
   LPSIZEL lpSizeInHiMetric
);
```

Parameters

lpSizeInPix

[in] Pointer to the object's size in pixels.

lpSizeInHiMetric

[out] Pointer to where the object's size in HIMETRIC units is to be returned.

Example

```
// Initialize our control's default size to 100 by 25 pixels
CMyControl::CMyControl()
{
    // width = 100 pixels, height = 25 pixels
    SIZE sz = { 100, 25 };
    // convert pixels to himetric
    AtlPixelToHiMetric (&sz, &m_sizeExtent);
    // store natural extent
    m_sizeNatural = m_sizeExtent;
}
```

See Also

Reference

AtlHiMetricToPixel

Other Resources

Pixel/HIMETRIC Conversion Global Functions

AtlRegisterTypeLib

This function is called to register a type library.

```
ATLAPI AtlRegisterTypeLib(
   HINSTANCE hInstTypeLib,
   LPCOLESTR lpszIndex
);
```

Parameters

hInstTypeLib

The handle to the module instance.

lpszIndex

String in the format "\\N", where N is the integer index of the type library resource. Can be NULL if no index is required.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This helper function is utilized by AtlComModuleUnregisterServer and CAtlComModule::RegisterTypeLib.

See Also

Reference

AtlLoadTypeLib

AtlUnRegisterTypeLib

Other Resources

Server Registration Global Functions

AtlReportError

Sets up the IErrorInfo interface to provide error information to clients of the object.

```
HRESULT WINAPI AtlReportError(
   const CLSID& clsid,
   LPCOLESTR lpszDesc,
   const IID& iid = GUID_NULL,
   HRESULT hRes = 0
);
HRESULT WINAPI AtlReportError(
   const CLSID& clsid,
   LPCOLESTR lpszDesc,
   DWORD dwHelpID,
   LPCOLESTR lpszHelpFile,
   const IID& iid = GUID_NULL,
   HRESULT hRes = 0
);
HRESULT WINAPI AtlReportError(
   const CLSID& clsid,
   LPCSTR lpszDesc,
   const IID& iid = GUID_NULL,
   HRESULT hRes = 0
);
HRESULT WINAPI AtlReportError(
   const CLSID& clsid,
   LPCSTR lpszDesc,
   DWORD dwHelpID,
   LPCSTR lpszHelpFile,
   const IID& iid = GUID_NULL,
   HRESULT hRes = 0
);
HRESULT WINAPI AtlReportError(
   const CLSID& clsid,
   UINT nID,
   const IID& iid = GUID_NULL,
   HRESULT hRes = 0,
   HINSTANCE hInst = _AtlBaseModule.GetResourceInstance( )
);
HRESULT WINAPI AtlReportError(
   const CLSID& clsid,
   UINT nID,
   DWORD dwHelpID,
   LPCOLESTR lpszHelpFile,
   const IID& iid = GUID_NULL,
   HRESULT hRes = 0,
   HINSTANCE hInst = _AtlBaseModule.GetResourceInstance( )
);
```

Parameters

clsid

[in] The CLSID of the object reporting the error.

lpszDesc

[in] The string describing the error. The Unicode versions specify that *lpszDesc* is of type **LPCOLESTR**; the ANSI version specifies a type of **LPCSTR**.

iid

[in] The IID of the interface defining the error or **GUID_NULL** if the error is defined by the operating system.

hRes

[in] The **HRESULT** you want returned to the caller.

nID

[in] The resource identifier where the error description string is stored. This value should lie between 0x0200 and 0xFFFF, inclusively. In debug builds, an **ASSERT** will result if *nID* does not index a valid string. In release builds, the error description string will be set to "Unknown Error."

dwHelpID

[in] The help context identifier for the error.

lpszHelpFile

[in] The path and name of the help file describing the error.

hInst

[in] The handle to the resource. By default, this parameter is **__AtlBaseModuleModule::GetResourceInstance**, where **__AtlBaseModuleModule** is the global instance of CAtlBaseModule or a class derived from it.

Return Value

If the *hRes* parameter is nonzero, returns the value of *hRes*. If *hRes* is zero, then the first four versions of **AtlReportError** return **DISP_E_EXCEPTION**. The last two versions return the result of the macro **MAKE_HRESULT(1, FACILITY_ITF,** *nID* **)**.

Remarks

The string *lpszDesc* is used as the text description of the error. When the client receives the *hRes* you return from **AtlReportError**, the client can access the **IErrorInfo** structure for details about the error.

Example

```
STDMETHODIMP CMyCtrl::MyMethod()
{
...
   if (bSucceeded)
      return S_OK;
   else
      // hRes is set to DISP_E_EXCEPTION
      return AtlReportError (GetObjectCLSID(), "My error message");
}
```

Caution

Do not use **AtlReportError** in C++ catch handlers. Some overrides of these functions use the ATL string conversion macros internally, which in turn use the **_alloca** function internally. Using **AtlReportError** in a C++ catch handler can cause exceptions in C++ catch handlers.

See Also

Reference

MAKE_HRESULT

Other Resources

Debugging and Error Reporting Global Functions

AtlSetChildSite

Call this function to set the site of the child object to the **IUnknown** of the parent object.

```
HRESULT AtlSetChildSite(
   IUnknown* punkChild,
   IUnknown* punkParent
);
```

Parameters

punkChild

[in] A pointer to the **IUnknown** interface of the child.

punkParent

[in] A pointer to the **IUnknown** interface of the parent.

Return Value

A standard HRESULT value.

See Also

Other Resources

Composite Control Global Functions

AtlSetDacl

Call this function to set the discretionary access-control list (DACL) information of a specified object.

```
inline bool AtlSetDacl(
   HANDLE hObject,
   SE_OBJECT_TYPE ObjectType,
   const CDacl& rDacl,
   DWORD dwInheritanceFlowControl = 0
) throw(...);
```

Parameters

hObject

Handle to the object for which to set security information.

ObjectType

Specifies a value from the SE_OBJECT_TYPE enumeration that indicates the type of object identified by the *hObject* parameter.

rDacl

The DACL containing the new security information.

dwInheritanceFlowControl

The inheritance flow control. This value can be 0 (the default), PROTECTED_DACL_SECURITY_INFORMATION or UNPROTECTED_DACL_SECURITY_INFORMATION.

Return Value

Returns true on success, false on failure.

Remarks

In debug builds, an assertion error will occur if *hObject* is invalid, or if *dwInheritanceFlowControl* is not one of the three permitted values.

See Also

Reference

AtlGetDacl

CDacl Class

Other Resources

Security Global Functions

AtlSetGroupSid

Call this function to set the group security identifier (SID) of an object.

```
inline bool AtlSetGroupSid(
   HANDLE hObject,
   SE_OBJECT_TYPE ObjectType,
   const CSid& rSid
) throw(...);
```

Parameters

hObject

Handle to the object for which to set security information.

ObjectType

Specifies a value from the SE_OBJECT_TYPE enumeration that indicates the type of object identified by the *hObject* parameter.

rSid

The **CSid** object containing the new security information.

Return Value

Returns true on success, false on failure.

See Also

Reference

AtlGetGroupSid

CSid Class

Other Resources

Security Global Functions

AtlSetOwnerSid

Call this function to set the owner security identifier (SID) of an object.

```
inline bool AtlSetOwnerSid(
   HANDLE hObject,
   SE_OBJECT_TYPE ObjectType,
   const CSid& rSid
) throw(...);
```

Parameters

hObject

Handle to the object for which to set security information.

ObjectType

Specifies a value from the SE_OBJECT_TYPE enumeration that indicates the type of object identified by the *hObject* parameter.

rSid

The **CSid** object containing the new security information.

Return Value

Returns true on success, false on failure.

See Also

Reference

AtlGetOwnerSid

CSid Class

Other Resources

Security Global Functions

AtlSetSacl

Call this function to set the system access-control list (SACL) information of a specified object.

```
inline bool AtlSetSacl(
   HANDLE hObject,
   SE_OBJECT_TYPE ObjectType,
   const CSacl& rSacl,
   DWORD dwInheritanceFlowControl = 0,
   bool bRequestNeededPrivileges = true
) throw(...);
```

Parameters

hObject

Handle to the object for which to set security information.

ObjectType

Specifies a value from the SE_OBJECT_TYPE enumeration that indicates the type of object identified by the *hObject* parameter.

rSacl

The SACL containing the new security information.

dwInheritanceFlowControl

The inheritance flow control. This value can be 0 (the default), PROTECTED_SACL_SECURITY_INFORMATION or UNPROTECTED_SACL_SECURITY_INFORMATION.

bRequestNeededPrivileges

If true, the function will attempt to enable the SE_SECURITY_NAME privilege, and restore it on completion.

Return Value

Returns true on success, false on failure.

Remarks

In debug builds, an assertion error will occur if *hObject* is invalid, or if *dwInheritanceFlowControl* is not one of the three permitted values.

If **AtlSetSacl** is to be called many times on many different objects, it will be more efficient to enable the SE_SECURITY_NAME privilege once before calling the function, with *bRequestNeededPrivileges* set to false.

See Also

Reference

AtlGetSacl

CSacl Class

Other Resources

Security Global Functions

AtlTraceLoadSettings

Call this function to load trace settings from a file.

```
BOOL AtlTraceLoadSettings(
   LPCTSTR* pszFilename,
   DWORD_PTR dwProcess = 0
);
```

Parameters

pszFilename

The full path and file name of a .TRC file. This parameter can be NULL, in which case the .TRC file should have the same file name as the EXE, and reside in the same directory.

dwProcess

Reserved.

Return Value

TRUE if the file was successfully loaded, FALSE otherwise.

Remarks

This function will load trace settings from a .TRC file. The .TRC file can be created using AtlTraceTool.exe. The trace settings control the type and amount of trace messages that appear in the Output window.

See Also

Tasks

ATLTraceTool Sample: Displays Output of ATLTRACE2

Other Resources

Debugging and Error Reporting Global Functions

AtlTraceSaveSettings

Call this function to save the current trace settings to a file.

```
BOOL AtlTraceSaveSettings(
   LPCTSTR* pszFilename,
   DWORD_PTR dwProcess = 0
);
```

Parameters

pszFilename

The full path and file name of the .TRC file to create.

dwProcess

Reserved.

Return Value

TRUE if the file was successfully saved, FALSE otherwise.

Remarks

This function will save the current trace settings to a .TRC file. The trace settings control the type and amount of trace messages that appear in the Output window.

AtlTraceSaveSettings makes use of WritePrivateProfileSection to store settings in the .TRC file. If a section already exists it will be replaced, otherwise it will be added to the file.

See Also

Tasks

ATLTraceTool Sample: Displays Output of ATLTRACE2

Other Resources

Debugging and Error Reporting Global Functions

AtlThrow

Call this function to signal an error based on a **HRESULT** status code.

```
inline void AtlThrow(
   HRESULT hr
);
```

Parameters

hr

Standard HRESULT value.

Remarks

This function is used by ATL and MFC code in the event of an error condition. It can also be called from your own code. The default implementation of this function depends on the definition of the symbol **_ATL_NO_EXCEPTIONS** and on the type of project, MFC or ATL.

In all cases, this function traces the HRESULT to the debugger.

If **_ATL_NO_EXCEPTIONS** is not defined in an MFC project, this function throws a CMemoryException or a COleException based on the value of the HRESULT.

If **_ATL_NO_EXCEPTIONS** is not defined in an ATL project, the function throws a CAtlException.

If **_ATL_NO_EXCEPTIONS** is defined, the function causes an assertion failure instead of throwing an exception.

For ATL projects, it is possible to provide your own implementation of this function to be used by ATL in the event of a failure. To do this, define your own function with the same signature as **AtlThrow** and #define **AtlThrow** to be the name of your function. This must be done before including atlexcept.h (which means that it must be done prior to including any ATL headers since atlbase.h includes atlexcept.h).

Example

```
// Example for AtlThrow
// Constructors and operators cannot return error codes, and
// so they are the place where exceptions are generally used.
class CMyClass
{
private:
   CComPtr<ITheirInterface> m_spTheirInterface;
public:
   CMyClass()
   {
      HRESULT hr = m_spTheirInterface.CoCreateInstance(__uuidof(CTheirCLSID));
      if (FAILED(hr))
         AtlThrow(hr);
   }
   //   methods ..
};
```

See Also

Reference

CAtlException Class
ATLTRACE2
CMemoryException Class
COleException Class
AtlThrowLastWin32

Other Resources

Debugging and Error Reporting Global Functions

AtlThrowLastWin32

Call this function to signal an error based on the result of the Windows function **GetLastError**.

```
inline void AtlThrowLastWin32( );
```

Remarks

This function traces the result of **GetLastError** to the debugger.

If **_ATL_NO_EXCEPTIONS** is not defined in an MFC project, this function throws a CMemoryException or a COleException based on the value returned by **GetLastError**.

If **_ATL_NO_EXCEPTIONS** is not defined in an ATL project, the function throws a CAtlException.

If **_ATL_NO_EXCEPTIONS** is defined, the function causes an assertion failure instead of throwing an exception.

See Also

Reference

ATLTRACE2

AtlThrow

GetLastError

Other Resources

Debugging and Error Reporting Global Functions

AtlUnadvise

Terminates the connection established through AtlAdvise.

```
HRESULT AtlUnadvise(
   IUnknown* pUnkCP,
   const IID& iid,
   DWORD dw
);
```

Parameters

pUnkCP

[in] A pointer to the **IUnknown** of the object that the client is connected with.

iid

[in] The GUID of the connection point. Typically, this is the same as the outgoing interface managed by the connection point.

dw

[in] The cookie that uniquely identifies the connection.

Return Value

A standard HRESULT value.

Example

```
LPUNKNOWN m_pSourceUnk;
LPUNKNOWN m_pSinkUnk;
DWORD m_dwCustCookie;

// create source object
HRESULT hr = CoCreateInstance (CLSID_MyComponent, NULL, CLSCTX_ALL,
         IID_IUnknown, (LPVOID*)&m_pSourceUnk);
_ASSERT (SUCCEEDED (hr));

// Create sink object.  CMySink is a CComObjectRootEx-derived class
// that implements the event interface methods.
CComObject<CMySink> *pSinkClass;
CComObject<CMySink>::CreateInstance (&pSinkClass);
hr = pSinkClass->QueryInterface (IID_IUnknown, (LPVOID*)&m_pSinkUnk);
_ASSERT (SUCCEEDED (hr));

hr = AtlAdvise (m_pSourceUnk, m_pSinkUnk, IID__IMyEvent, &m_dwCustCookie);
_ASSERT (SUCCEEDED (hr));

// do something

HRESULT hr = AtlUnadvise (m_pSourceUnk, IID__IMyEvent, m_dwCustCookie);
_ASSERT (SUCCEEDED (hr));
```

See Also

Other Resources

Connection Point Global Functions

AtlUnmarshalPtr

Converts the stream's marshaling data into an interface pointer that can be used by the client.

```
HRESULT AtlUnmarshalPtr(
   IStream* pStream,
   const IID& iid,
   IUnknown** ppUnk
);
```

Parameters

pStream

[in] A pointer to the stream being unmarshaled.

iid

[in] The GUID of the interface being unmarshaled.

ppUnk

[out] A pointer to the unmarshaled interface.

Return Value

A standard HRESULT value.

Example

See the example for AtlMarshalPtrInProc.

See Also

Reference

AtlMarshalPtrInProc

Other Resources

Marshaling Global Functions

AtlUnRegisterTypeLib

This function is called to unregister a type library.

```
ATLAPI AtlUnRegisterTypeLib(
   HINSTANCE hInstTypeLib,
   LPCOLESTR lpszIndex
);
```

Parameters

hInstTypeLib

The handle to the module instance.

lpszIndex

String in the format "\\N", where N is the integer index of the type library resource. Can be NULL if no index is required.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This helper function is utilized by CAtlComModule::UnRegisterTypeLib and AtlComModuleUnregisterServer.

See Also

Reference

AtlRegisterTypeLib

AtlLoadTypeLib

Other Resources

Server Registration Global Functions

AtlUpdateRegistryFromResourceD

This function is called to update the registry from the supplied resource.

```
STDAPI AtlUpdateRegistryFromResourceD(
   HINSTANCE hInst,
   LPCOLESTR lpszRes,
   BOOL bRegister,
   struct _ATL_REGMAP_ENTRY* pMapEntries,
   IRegistrar * pReg
);
```

Parameters

hInst

Handle to the current process instance.

lpszRes

A resource name.

bRegister

Indicates whether the object should be registered.

pMapEntries

A pointer to the replacement map storing values associated with the script's replaceable parameters.

pReg

Pointer to the ATL Registry Component (Registrar) replacement map.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Utilized by CAtlModule::UpdateRegistryFromResourceDHelper.

See Also

Other Resources

Registry and TypeLib Global Functions

AtlWaitWithMessageLoop

Waits for the object to be signaled, meanwhile dispatching window messages as needed.

```
BOOL AtlWaitWithMessageLoop(
   HANDLE hEvent
);
```

Parameters

hEvent

[in] The handle of the object to wait for.

Return Value

Returns **TRUE** if the object has been signaled.

Remarks

This is useful if you want to wait for an object's event to happen and be notified of it happening, but allow window messages to be dispatched while waiting.

See Also

Other Resources

Event Handling Global Functions

AtlWinModuleAddCreateWndData

This function is used to initialize and add an **_AtlCreateWndData** structure.

```
ATLINLINE ATLAPI_(void) AtlWinModuleAddCreateWndData(
   _ATL_WIN_MODULE * pWinModule,
   _AtlCreateWndData* pData,
   void * pObject
);
```

Parameters

pWinModule

Pointer to a module's _ATL_WIN_MODULE70 structure.

pData

Pointer to the _AtlCreateWndData structure to be initialized and added to the current module.

pObject

Pointer to an object's **this** pointer.

Remarks

Initializes an **_AtlCreateWndData** structure, which is used to store the **this** pointer used to refer to class instances, and adds it to the list referenced by a module's **_ATL_WIN_MODULE70** structure. Called by CAtlWinModule::AddCreateWndData.

See Also

Reference

AtlWinModuleExtractCreateWndData

Other Resources

WinModule Global Functions

AtlWinModuleExtractCreateWndData

Call this function to extract an existing **_AtlCreateWndData** structure.

```
ATLINLINE ATLAPI_(void*) AtlWinModuleExtractCreateWndData(
   _ATL_WIN_MODULE* pWinModule
);
```

Parameters

pWinModule

Pointer to a module's _ATL_WIN_MODULE70 structure.

Return Value

Returns a pointer to the _AtlCreateWndData structure.

Remarks

This function will extract an existing **_AtlCreateWndData** structure from the list referenced by a module's **_ATL_WIN_MODULE70** structure.

See Also

Reference

AtlWinModuleAddCreateWndData

Other Resources

WinModule Global Functions

InlineIsEqualIUnknown

Call this function, for the special case of testing for **IUnknown**.

```
BOOL InlineIsEqualUnknown(
   REFGUID rguid1
);
```

Parameters

rguid1

[in] The GUID to compare to **IID_IUnknown**.

See Also

Other Resources

COM Map Global Functions

RegistryDataExchange

This function is called to read from, or write to, the system registry.

```
HRESULT RegistryDataExchange(
   T* pT,
   enum RDXOperations rdxOp,
   void* pItem = NULL
);
```

Parameters

pT

A pointer to the current object.

rdxOp

An enum value that indicates which operation the function should perform. See the table in the Remarks section for the allowed values.

pItem

Pointer to the data that is to be read from, or written to, the registry. The data can also represent a key to be deleted from the registry. The default value is NULL.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The macros BEGIN_RDX_MAP and END_RDX_MAP expand to a function that calls **RegistryDataExchange**.

The possible enum values that indicate the operation the function should perform are shown in the following table:

Enum value	Operation
eReadFromReg	Read data from the registry.
eWriteToReg	Write data to the registry.
eDeleteFromReg	Delete the key from the registry.

See Also

Other Resources

Registry and TypeLib Global Functions

Registry Data Exchange Macros

Sids::AccountOps

Returns the DOMAIN_ALIAS_RID_ACCOUNT_OPS SID.

```
CSid AccountOps( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::Admins

Returns the DOMAIN_ALIAS_RID_ADMINS SID.

```
CSid Admins( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::AnonymousLogon

Returns the SECURITY_ANONYMOUS_LOGON_RID SID.

```
CSid AnonymousLogon( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::AuthenticatedUser

Returns the SECURITY_AUTHENTICATED_USER_RID SID.

```
CSid AuthenticatedUser( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::BackupOps

Returns the DOMAIN_ALIAS_RID_BACKUP_OPS SID.

```
CSid BackupOps( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::Batch

Returns the SECURITY_BATCH_RID SID.

```
CSid Batch( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::CreatorGroup

Returns the SECURITY_CREATOR_GROUP_RID SID.

```
CSid CreatorGroup( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::CreatorGroupServer

Returns the SECURITY_CREATOR_GROUP_SERVER_RID SID.

```
CSid CreatorGroupServer( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::CreatorOwner

Returns the SECURITY_CREATOR_OWNER_RID SID.

```
CSid CreatorOwner( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::CreatorOwnerServer

Returns the SECURITY_CREATOR_OWNER_SERVER_RID SID.

```
CSid CreatorOwnerServer( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::Dialup

Returns the SECURITY_DIALUP_RID SID.

```
CSid Dialup( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::Guests

Returns the DOMAIN_ALIAS_RID_GUESTS SID.

```
CSid Guests( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::Interactive

Returns the SECURITY_INTERACTIVE_RID SID.

```
CSid Interactive( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::Local

Returns the SECURITY_LOCAL_RID SID.

```
CSid Local( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::Network

Returns the SECURITY_NETWORK_RID SID.

```
CSid Network( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::NetworkService

Returns the SECURITY_NETWORK_SERVICE_RID SID.

```
CSid NetworkService() throw(...);
```

Remarks

Use NetworkService to enable the NT AUTHORITY\NetworkService user to read a CPerfMon security object. NetworkService adds a SecurityAttribute to the ATLServer code which will allow the DLL to login under the NetworkService account on Windows XP Home Edition, Windows XP Professional, Windows Server 2003 and greater operating system.

When custom log counters are created with ATLServer CPerfMon class in the Perfmon MMC, the counters may not appear when viewing the log file although they will appear correctly in the realtime view. CPerfMon custom performance counters don't have the necessary permissions to run under the "Performance Logs and Alerts" service (smlogsvc.exe) on Windows XP Home Edition, Windows XP Professional, Windows Server 2003 (or greater) operating systems. This service runs under the "NT AUTHORITY\NetworkService" account.

See Also

Other Resources

Security Identifier Global Functions

Sids::Null

Returns the SECURITY_NULL_RID SID.

```
CSid Null( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::PowerUsers

Returns the DOMAIN_ALIAS_RID_POWER_USERS SID.

```
CSid PowerUsers( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::PreW2KAccess

Returns the DOMAIN_ALIAS_RID_PREW2KCOMPACCESS SID.

```
CSid PreW2KAccess( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::PrintOps

Returns the DOMAIN_ALIAS_RID_PRINT_OPS SID.

```
CSid PrintOps( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::Proxy

Returns the SECURITY_PROXY_RID SID.

```
CSid Proxy( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::RasServers

Returns the DOMAIN_ALIAS_RID_RAS_SERVERS SID.

```
CSid RasServers( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::Replicator

Returns the DOMAIN_ALIAS_RID_REPLICATOR SID.

```
CSid Replicator( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::RestrictedCode

Returns the SECURITY_RESTRICTED_CODE_RID SID.

```
CSid RestrictedCode( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::Self

Returns the SECURITY_PRINCIPAL_SELF_RID SID.

```
CSid Self( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::ServerLogon

Returns the SECURITY_SERVER_LOGON_RID SID.

```
CSid ServerLogon( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::Service

Returns the SECURITY_SERVICE_RID SID.

```
CSid Service( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::System

Returns the SECURITY_LOCAL_SYSTEM_RID SID.

```
CSid System( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::SystemOps

Returns the DOMAIN_ALIAS_RID_SYSTEM_OPS SID.

```
CSid SystemOps( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::TerminalServer

Returns the SECURITY_TERMINAL_SERVER_RID SID.

```
CSid TerminalServer( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::Users

Returns the DOMAIN_ALIAS_RID_USERS SID.

```
CSid Users( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

Sids::World

Returns the SECURITY_WORLD_RID SID.

```
CSid World( ) throw(...);
```

See Also

Other Resources

Security Identifier Global Functions

ATL Global Variables

The following global variables are provided as part of the ATL classes:

_pAtlModule	A variable storing a pointer to the current module.

See Also

Other Resources

ATL Reference
ATL Functions
ATL Macros
ATL Structures
ATL Typedefs
ATL Classes

_pAtlModule

A global variable storing a pointer to the current module.

```
__declspec(selectany) CAtlModule * _pAtlModule
```

Remarks

Methods on this global variable can be used to provide the functionality that the (now obsolete) class CComModule provided in Visual C++ 6.0.

Example

```
// Example - obtaining the module's resource handle
HINSTANCE hInstRes = _pAtlModule->GetResourceInstance();
```

Requirements

Header: atlbase.h

See Also

Other Resources

ATL Global Variables

ATL Macros

To find an ATL macro by category, see the following topics.

Aggregation and Class Factory Macros

Provide ways of controlling aggregation and of declaring class factories.

Category Macros

Define category maps.

COM Map Macros

Define COM interface maps.

Compiler Options Macros

Control specific compiler features.

Composite Control Macros

Define event sink maps and entries.

Connection Point Macros

Define connection point maps and entries.

Debugging and Error Reporting Macros

Provide useful debugging and trace facilities.

Exception Handling Macros

Provide support for exception handling.

Message Map Macros

Define message maps and entries.

Object Map Macros

Define object maps and entries.

Object Status Macros

Sets flags belonging to ActiveX controls.

Property Map Macros

Define property maps and entries.

Registry Data Exchange Macros

Perform Registry Data Exchange operations.

Registry Macros

Define useful type library and registry facilities.

Service Map Macros

Define service maps and entries.

Snap-In Object Macros

Provide support for snap-in extensions.

String Conversion Macros

Provide string conversion features.

Window Class Macros

Define window class utilities.

Windows Messages Macros

Forward window messages.

See Also

Other Resources

ATL Macros Alphabetical Reference
ATL Reference
ATL Functions
ATL Global Variables
ATL Structures
ATL Typedefs
ATL Classes

Aggregation and Class Factory Macros

These macros provide ways of controlling aggregation and of declaring class factories.

DECLARE_AGGREGATABLE	Declares that your object can be aggregated (the default).
DECLARE_CLASSFACTORY	Declares the class factory to be CComClassFactory, the ATL default class factory.
DECLARE_CLASSFACTORY_EX	Declares your class factory object to be the class factory.
DECLARE_CLASSFACTORY2	Declares CComClassFactory2 to be the class factory.
DECLARE_CLASSFACTORY_AUTO_THREAD	Declares CComClassFactoryAutoThread to be the class factory.
DECLARE_CLASSFACTORY_SINGLETON	Declares CComClassFactorySingleton to be the class factory.
DECLARE_GET_CONTROLLING_UNKNOWN	Declares a virtual **GetControllingUnknown** function.
DECLARE_NOT_AGGREGATABLE	Declares that your object cannot be aggregated.
DECLARE_ONLY_AGGREGATABLE	Declares that your object must be aggregated.
DECLARE_POLY_AGGREGATABLE	Checks the value of the outer unknown and declares your object aggregatable or not aggregatable, as appropriate.
DECLARE_PROTECT_FINAL_CONSTRUCT	Protects the outer object from deletion during construction of an inner object.
DECLARE_VIEW_STATUS	Specifies the **VIEWSTATUS** flags to the container.

See Also

Other Resources

ATL Macros

Category Macros

These macros define category maps.

BEGIN_CATEGORY_MAP	Marks the beginning of the category map.
END_CATEGORY_MAP	Marks the end of the category map.
IMPLEMENTED_CATEGORY	Indicates categories that are implemented by the COM object.
REQUIRED_CATEGORY	Indicates categories that are required of the container by the COM object.

See Also

Other Resources

ATL Macros

COM Map Macros

These macros define COM interface maps.

BEGIN_COM_MAP	Marks the beginning of the COM interface map entries.
COM_INTERFACE_ENTRY	Enters interfaces into the COM interface map.
COM_INTERFACE_ENTRY2	Use this macro to disambiguate two branches of inheritance.
COM_INTERFACE_ENTRY_IID	Use this macro to enter the interface into the COM map and specify its IID .
COM_INTERFACE_ENTRY2_IID	Same as COM_INTERFACE_ENTRY2, except you can specify a different IID.
COM_INTERFACE_ENTRY_AGGREGATE	When the interface identified by *iid* is queried for, **COM_INTERFACE_ENTRY_AGGREGATE** forwards to *punk*.
COM_INTERFACE_ENTRY_AGGREGATE_BLIND	Same as COM_INTERFACE_ENTRY_AGGREGATE, except that querying for any IID results in forwarding the query to *punk*.
COM_INTERFACE_ENTRY_AUTOAGGREGATE	Same as COM_INTERFACE_ENTRY_AGGREGATE, except if *punk* is **NULL**, it automatically creates the aggregate described by the *clsid*.
COM_INTERFACE_ENTRY_AUTOAGGREGATE_BLIND	Same as COM_INTERFACE_ENTRY_AUTOAGGREGATE, except that querying for any IID results in forwarding the query to *punk*, and if *punk* is **NULL**, automatically creating the aggregate described by the *clsid*.
COM_INTERFACE_ENTRY_BREAK	Causes your program to call DebugBreak when the specified interface is queried for.
COM_INTERFACE_ENTRY_CACHED_TEAR_OFF	Saves the interface-specific data for every instance.
COM_INTERFACE_ENTRY_TEAR_OFF	Exposes your tear-off interfaces.
COM_INTERFACE_ENTRY_CHAIN	Processes the COM map of the base class when the processing reaches this entry in the COM map.
COM_INTERFACE_ENTRY_FUNC	A general mechanism for hooking into ATL's **QueryInterface** logic.
COM_INTERFACE_ENTRY_FUNC_BLIND	Same as COM_INTERFACE_ENTRY_FUNC, except that querying for any IID results in a call to *func*.
COM_INTERFACE_ENTRY_NOINTERFACE	Returns **E_NOINTERFACE** and terminates COM map processing when the specified interface is queried for.
END_COM_MAP	Marks the end of the COM interface map entries.

See Also

Other Resources

ATL Macros

COM Map Global Functions

Compiler Options Macros

These macros control specific compiler features.

_ATL_ALL_WARNINGS	A symbol which enables errors in projects converted from previous versions of ATL.
_ATL_APARTMENT_THREADED	Define if one or more of your objects use apartment threading.
_ATL_CSTRING_EXPLICIT_CONSTRUCTORS	Makes certain **CString** constructors explicit, preventing any unintentional conversions.
_ATL_ENABLE_PTM_WARNING	Define this macro in order to use C++ standard compliant syntax, which generates the C4867 compiler error when a non standard syntax is used to initialize a pointer to a member function.
_ATL_FREE_THREADED	Define if one or more of your objects use free or neutral threading.
_ATL_MULTI_THREADED	A symbol that indicates the project will have objects that are marked as Both, Free or Neutral. The macro _ATL_FREE_THREADED should be used instead.
_ATL_NO_AUTOMATIC_NAMESPACE	A symbol which prevents the default use of namespace as ATL.
_ATL_NO_COM_SUPPORT	A symbol which prevents COM-related code from being compiled with your project.
ATL_NO_VTABLE	A symbol that prevents the vtable pointer from being initialized in the class's constructor and destructor.
ATL_NOINLINE	A symbol that indicates a function should not be inlined.
_ATL_SINGLE_THREADED	Define if all of your objects use the single threading model.

See Also

Other Resources

ATL Macros

Composite Control Macros

These macros define event sink maps and entries.

BEGIN_SINK_MAP	Marks the beginning of the event sink map for the composite control.
END_SINK_MAP	Marks the end of the event sink map for the composite control.
SINK_ENTRY	Entry to the event sink map.
SINK_ENTRY_EX	Entry to the event sink map with an additional parameter.
SINK_ENTRY_INFO	Entry to the event sink map with manually supplied type information for use with IDispEventSimpleImpl.

See Also

Other Resources

ATL Macros

Composite Control Global Functions

Connection Point Macros

These macros define connection point maps and entries.

BEGIN_CONNECTION_POINT_MAP	Marks the beginning of the connection point map entries.
CONNECTION_POINT_ENTRY	Enters connection points into the map.
END_CONNECTION_POINT_MAP	Marks the end of the connection point map entries.

See Also

Other Resources

ATL Macros

Connection Point Global Functions

Debugging and Error Reporting Macros

These macros provide useful debugging and trace facilities.

_ATL_DEBUG_INTERFACES	Writes, to the output window, any interface leaks that are detected when `_Module.Term` is called.
_ATL_DEBUG_QI	Writes all calls to **QueryInterface** to the output window.
ATLASSERT	Performs the same functionality as the _ASSERTE macro found in the C run-time library.
ATLENSURE	Performs parameters validation. Call **AtlThrow** if needed
ATLTRACENOTIMPL	Sends a message to the dump device that the specified function is not implemented.
ATLTRACE	Reports warnings to an output device, such as the debugger window, according to the indicated flags and levels. Included for backward compatibility.
ATLTRACE2	Reports warnings to an output device, such as the debugger window, according to the indicated flags and levels.

See Also

Other Resources

ATL Macros

Debugging and Error Reporting Global Functions

Exception Handling Macros

These macros provide support for exception handling.

_ATLCATCH	Statement(s) to handle errors occurring in the associated **_ATLTRY**.
_ATLCATCHALL	Statement(s) to handle errors occurring in the associated **_ATLTRY**.
_ATLTRY	Marks a guarded code section where an error could possibly occur.

See Also

Other Resources

ATL Macros

Message Map Macros (ATL)

These macros define message maps and entries.

ALT_MSG_MAP	Marks the beginning of an alternate message map.
BEGIN_MSG_MAP	Marks the beginning of the default message map.
CHAIN_MSG_MAP_ALT	Chains to an alternate message map in the base class.
CHAIN_MSG_MAP_ALT_MEMBER	Chains to an alternate message map in a data member of the class.
CHAIN_MSG_MAP	Chains to the default message map in the base class.
CHAIN_MSG_MAP_DYNAMIC	Chains to the message map in another class at run time.
CHAIN_MSG_MAP_MEMBER	Chains to the default message map in a data member of the class.
COMMAND_CODE_HANDLER	Maps a **WM_COMMAND** message to a handler function, based on the notification code.
COMMAND_HANDLER	Maps a **WM_COMMAND** message to a handler function, based on the notification code and the identifier of the menu item, control, or accelerator.
COMMAND_ID_HANDLER	Maps a **WM_COMMAND** message to a handler function, based on the identifier of the menu item, control, or accelerator.
COMMAND_RANGE_CODE_HANDLER	Maps a **WM_COMMAND** message to a handler function, based on the notification code and a contiguous range of control identifiers.
COMMAND_RANGE_HANDLER	Maps a **WM_COMMAND** message to a handler function, based on a contiguous range of control identifiers.
DECLARE_EMPTY_MSG_MAP	Implements an empty message map.
DEFAULT_REFLECTION_HANDLER	Provides a default handler for reflected messages that are not handled otherwise.
END_MSG_MAP	Marks the end of a message map.
FORWARD_NOTIFICATIONS	Forwards notification messages to the parent window.
MESSAGE_HANDLER	Maps a Windows message to a handler function.
MESSAGE_RANGE_HANDLER	Maps a contiguous range of Windows messages to a handler function.
NOTIFY_CODE_HANDLER	Maps a **WM_NOTIFY** message to a handler function, based on the notification code.
NOTIFY_HANDLER	Maps a **WM_NOTIFY** message to a handler function, based on the notification code and the control identifier.

NOTIFY_ID_HANDLER	Maps a **WM_NOTIFY** message to a handler function, based on the control identifier.
NOTIFY_RANGE_CODE_HANDLER	Maps a **WM_NOTIFY** message to a handler function, based on the notification code and a contiguous range of control identifiers.
NOTIFY_RANGE_HANDLER	Maps a **WM_NOTIFY** message to a handler function, based on a contiguous range of control identifiers.
REFLECT_NOTIFICATIONS	Reflects notification messages back to the window that sent them.
REFLECTED_COMMAND_CODE_HANDLER	Maps a reflected **WM_COMMAND** message to a handler function, based on the notification code.
REFLECTED_COMMAND_HANDLER	Maps a reflected **WM_COMMAND** message to a handler function, based on the notification code and the identifier of the menu item, control, or accelerator.
REFLECTED_COMMAND_ID_HANDLER	Maps a reflected **WM_COMMAND** message to a handler function, based on the identifier of the menu item, control, or accelerator.
REFLECTED_COMMAND_RANGE_CODE_HANDLER	Maps a reflected **WM_COMMAND** message to a handler function, based on the notification code and a contiguous range of control identifiers.
REFLECTED_COMMAND_RANGE_HANDLER	Maps a reflected **WM_COMMAND** message to a handler function, based on a contiguous range of control identifiers.
REFLECTED_NOTIFY_CODE_HANDLER	Maps a reflected **WM_NOTIFY** message to a handler function, based on the notification code.
REFLECTED_NOTIFY_HANDLER	Maps a reflected **WM_NOTIFY** message to a handler function, based on the notification code and the control identifier.
REFLECTED_NOTIFY_ID_HANDLER	Maps a reflected **WM_NOTIFY** message to a handler function, based on the control identifier.
REFLECTED_NOTIFY_RANGE_CODE_HANDLER	Maps a reflected **WM_NOTIFY** message to a handler function, based on the notification code and a contiguous range of control identifiers.
REFLECTED_NOTIFY_RANGE_HANDLER	Maps a reflected **WM_NOTIFY** message to a handler function, based on a contiguous range of control identifiers.

See Also

Other Resources

ATL Macros

Object Map Macros

These macros define object maps and entries.

DECLARE_OBJECT_DESCRIPTION	Allows you to specify a class object's text description, which will be entered into the object map.
OBJECT_ENTRY_AUTO	Enters an ATL object into the object map, updates the registry, and creates an instance of the object.
OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO	Allows you to specify that the object should be registered and initialized, but it should not be externally creatable via **CoCreateInstance**.

See Also

Other Resources

ATL Macros

Object Status Macros

This macro sets flags belonging to ActiveX controls.

DECLARE_OLEMISC_STATUS	Used in ATL ActiveX controls to set the **OLEMISC** flags.

See Also

Other Resources

ATL Macros

Property Map Macros

These macros define property maps and entries.

BEGIN_PROP_MAP	Marks the beginning of the ATL property map.
PROP_DATA_ENTRY	Indicates the extent, or dimensions, of an ActiveX control.
PROP_ENTRY	Enters a property description, property DISPID, and property page CLSID into the property map.
PROP_ENTRY_EX	Enters a property description, property DISPID, property page CLSID, and **IDispatch** IID into the property map.
PROP_PAGE	Enters a property page CLSID into the property map.
END_PROP_MAP	Marks the end of the ATL property map.

See Also

Other Resources

ATL Macros

Registry Data Exchange Macros

These macros perform Registry Data Exchange operations.

BEGIN_RDX_MAP	Marks the beginning of the Registry Data Exchange map.
END_RDX_MAP	Marks the end of the Registry Data Exchange map.
RDX_BINARY	Associates the specified registry entry with a specified member variable of type BYTE.
RDX_CSTRING_TEXT	Associates the specified registry entry with a specified member variable of type CString.
RDX_DWORD	Associates the specified registry entry with a specified member variable of type DWORD.
RDX_TEXT	Associates the specified registry entry with a specified member variable of type TCHAR.

See Also

Reference

RegistryDataExchange

Other Resources

ATL Macros

Registry Macros

These macros define useful type library and registry facilities.

_ATL_STATIC_REGISTRY	Indicates that you want the registration code for your object to be in the object to a void a dependency on ATL.DLL.
DECLARE_LIBID	Provides a way for ATL to obtain the *libid* of the type library.
DECLARE_NO_REGISTRY	Avoids default ATL registration.
DECLARE_REGISTRY	Enters or removes the main object's entry in the system registry.
DECLARE_REGISTRY_APPID_RESOURCEID	Specifies the information required to automatically register the *appid*.
DECLARE_REGISTRY_RESOURCE	Finds the named resource and runs the registry script within it.
DECLARE_REGISTRY_RESOURCEID	Finds the resource identified by an ID number and runs the registry script within it.

See Also

Other Resources

ATL Macros

Service Map Macros

These macros define service maps and entries.

BEGIN_SERVICE_MAP	Marks the beginning of an ATL service map.
END_SERVICE_MAP	Marks the end of an ATL service map.
SERVICE_ENTRY	Indicates that the object supports a specific service ID.
SERVICE_ENTRY_CHAIN	Instructs IServiceProviderImpl::QueryService to chain to the specified object.

See Also

Other Resources

ATL Macros

Snap-In Object Macros

These macros provide support for snap-in extensions.

BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP	Marks the beginning of the snap-in extension data class map for a Snap-In object.
BEGIN_SNAPINTOOLBARID_MAP	Marks the beginning of the toolbar map for a Snap-In object.
END_EXTENSION_SNAPIN_NODEINFO_MAP	Marks the end of the snap-in extension data class map for a Snap-In object.
END_SNAPINTOOLBARID_MAP	Marks the end of the toolbar map for a Snap-In object.
EXTENSION_SNAPIN_DATACLASS	Creates a data member for the data class of the snap-in extension.
EXTENSION_SNAPIN_NODEINFO_ENTRY	Enters a snap-in extension data class into the snap-in extension data class map of the Snap-In object.
SNAPINMENUID	Declares the ID of the context menu used by the Snap-In object.
SNAPINTOOLBARID_ENTRY	Enters a toolbar into the toolbar map of the Snap-In object.

See Also

Other Resources

ATL Macros

String Conversion Macros

These macros provide string conversion features.

ATL and MFC String Conversion Macros	Set of macros that convert between string types.
DEVMODE and TEXTMETRIC String Conversion Macros	Set of macros that convert the strings within **DEVMODE** and **TEXTMETRIC** structures.

See Also

Other Resources

ATL Macros

Window Class Macros

These macros define window class utilities.

DECLARE_WND_CLASS	Allows you to specify the name of a new window class.
DECLARE_WND_SUPERCLASS	Allows you to specify the name of an existing window class on which a new window class will be based.
DECLARE_WND_CLASS_EX	Allows you to specify the parameters of a class.

See Also

Other Resources

ATL Macros

Windows Messages Macros

This macro forwards window messages.

WM_FORWARDMSG	Use to forward a message received by a window to another window for processing.

See Also

Other Resources

ATL Macros

ATL Macros Alphabetical Reference

In this section, reference topics for the ATL macros are organized alphabetically. To find a particular macro by category, see ATL Macros.

Macro	Description
_ATL_ALL_WARNINGS	A symbol which enables errors in projects converted from previous versions of ATL.
_ATL_APARTMENT_THREADED	Define if one or more of your objects use apartment threading.
_ATL_CSTRING_EXPLICIT_CONSTRUCTORS	Makes certain **CString** constructors explicit, preventing any unintentional conversions.
_ATL_DEBUG_INTERFACES	Define this macro before including any ATL header files to trace all **AddRef** and **Release** calls on your components' interfaces to the output window.
_ATL_DEBUG_QI	Writes all calls to **QueryInterface** to the output window.
_ATL_FREE_THREADED	Define if one or more of your objects use free or neutral threading.
_ATL_MIXED	Tells the compiler that your ATL application is comprised of native and **/clr** compilands.
_ATL_MULTI_THREADED	A symbol that indicates the project will have objects that are marked as Both, Free or Neutral. The macro _ATL_FREE_THREADED should be used in new code.
_ATL_NO_COM_SUPPORT	A symbol which prevents COM-related code from being compiled with your project.
_ATL_NO_AUTOMATIC_NAMESPACE	A symbol which prevents the default use of namespace as ATL.
_ATL_SECURE_NO_WARNINGS	Suppresses compiler warnings for the use of deprecated ATL functions.
_ATL_STATIC_REGISTRY	A symbol that indicates you want the registration code for your object to be in the object to avoid a dependency on ATL.DLL.
_ATLCATCH	Statement(s) to handle errors occurring in the associated **_ATLTRY**
_ATLCATCHALL	Statement(s) to handle errors occurring in the associated **_ATLTRY**.
_ATLTRY	Marks a guarded code section where an error could possibly occur.
ALT_MSG_MAP	Marks the beginning of an alternate message map.
ATL and MFC String Conversion Macros	String conversion macros valid for both ATL and MFC.
ATLASSERT	The **ATLASSERT** macro performs the same functionality as the _ASSERTE macro found in the C run-time library.
ATL_NOINLINE	A symbol that indicates a function should not be inlined.

ATL_NO_VTABLE	A symbol that prevents the vtable pointer from being initialized in the class's constructor and destructor.
ATLTRACE	Reports warnings to an output device, such as the debugger window, a ccording to the indicated flags and levels. Included for backward comp atibility.
ATLTRACE2	Reports warnings to an output device, such as the debugger window, a ccording to the indicated flags and levels.
ATLTRACENOTIMPL	In debug builds of ATL, sends the string "*funcname* is not implemented " to the dump device and returns **E_NOTIMPL**.
BEGIN_CATEGORY_MAP	Marks the beginning of the category map.
BEGIN_COM_MAP	The COM map is the mechanism that exposes interfaces on an object t o a client through **QueryInterface**.
BEGIN_CONNECTION_POINT_MAP	Marks the beginning of the connection point map entries.
BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP	Marks the beginning of the snap-in extension data class map.
BEGIN_MSG_MAP	Marks the beginning of the default message map.
BEGIN_PROP_MAP	Marks the beginning of the object's property map.
BEGIN_RDX_MAP	Marks the beginning of the Registry Data Exchange map.
BEGIN_SERVICE_MAP	Marks the beginning of the service map.
BEGIN_SINK_MAP	Declares the beginning of the event sink map for the composite contro l.
BEGIN_SNAPINTOOLBARID_MAP	Declares the beginning of the toolbar ID map for the Snap-In object.
CHAIN_MSG_MAP	Defines an entry in a message map.
CHAIN_MSG_MAP_ALT	Defines an entry in a message map.
CHAIN_MSG_MAP_ALT_MEMBER	Defines an entry in a message map.
CHAIN_MSG_MAP_DYNAMIC	Defines an entry in a message map.
CHAIN_MSG_MAP_MEMBER	Defines an entry in a message map.
COM_INTERFACE_ENTRY Macros	These macros enter an object's interfaces into its COM map so that the y can be accessed by **QueryInterface**.
COM_INTERFACE_ENTRY	Enters interfaces into the COM interface map.
COM_INTERFACE_ENTRY2	Use this macro to disambiguate two branches of inheritance.

COM_INTERFACE_ENTRY2_IID	Same as COM_INTERFACE_ENTRY2, except you can specify a different I ID.
COM_INTERFACE_ENTRY_AGGREGATE	When the interface identified by *iid* is queried for, **COM_INTERFACE_E NTRY_AGGREGATE** forwards to *punk*.
COM_INTERFACE_ENTRY_AGGREGATE_BLIND	Same as COM_INTERFACE_ENTRY_AGGREGATE, except that querying f or any IID results in forwarding the query to *punk*.
COM_INTERFACE_ENTRY_AUTOAGGREGATE	Same as COM_INTERFACE_ENTRY_AGGREGATE, except if *punk* is **NUL L**, it automatically creates the aggregate described by the *clsid*.
COM_INTERFACE_ENTRY_AUTOAGGREGATE_BLIND	Same as COM_INTERFACE_ENTRY_AUTOAGGREGATE, except that quer ying for any IID results in forwarding the query to *punk*, and if *punk* is **NULL**, automatically creating the aggregate described by the *clsid*.
COM_INTERFACE_ENTRY_BREAK	Causes your program to call DebugBreak when the specified interface i s queried for.
COM_INTERFACE_ENTRY_CACHED_TEAR_OFF	Saves the interface-specific data for every instance.
COM_INTERFACE_ENTRY_CHAIN	Processes the COM map of the base class when the processing reaches this entry in the COM map.
COM_INTERFACE_ENTRY_FUNC	A general mechanism for hooking into ATL's **QueryInterface** logic.
COM_INTERFACE_ENTRY_FUNC_BLIND	Same as COM_INTERFACE_ENTRY_FUNC, except that querying for any IID results in a call to *func*.
COM_INTERFACE_ENTRY_IID	Use this macro to enter the interface into the COM map and specify its IID.
COM_INTERFACE_ENTRY_NOINTERFACE	Returns **E_NOINTERFACE** and terminates COM map processing when the specified interface is queried for.
COM_INTERFACE_ENTRY_TEAR_OFF	Exposes your tear-off interfaces.
COMMAND_CODE_HANDLER	Similar to COMMAND_HANDLER, but maps a WM_COMMAND messa ge based only on the notification code.
COMMAND_HANDLER	Defines an entry in a message map.
COMMAND_ID_HANDLER	Similar to COMMAND_HANDLER, but maps a WM_COMMAND messa ge based only on the identifier of the menu item, control, or accelerato r.
COMMAND_RANGE_CODE_HANDLER	Similar to COMMAND_RANGE_HANDLER, but maps WM_COMMAND messages with a specific notification code from a range of controls to a single handler function.
COMMAND_RANGE_HANDLER	Similar to COMMAND_HANDLER, but maps WM_COMMAND messag es from a range of controls to a single handler function.
CONNECTION_POINT_ENTRY	Enters a connection point for the specified interface into the connectio n point map so that it can be accessed.

DECLARE_AGGREGATABLE	Specifies that your object can be aggregated.
DECLARE_CLASSFACTORY	Declares CComClassFactory to be the class factory.
DECLARE_CLASSFACTORY2	Declares CComClassFactory2 to be the class factory.
DECLARE_CLASSFACTORY_AUTO_THREAD	Declares CComClassFactoryAutoThread to be the class factory.
DECLARE_CLASSFACTORY_EX	Declares *cf* to be the class factory.
DECLARE_CLASSFACTORY_SINGLETON	Declares CComClassFactorySingleton to be the class factory.
DECLARE_EMPTY_MSG_MAP	Declares an empty message map.
DECLARE_GET_CONTROLLING_UNKNOWN	Declares a virtual function **GetControllingUnknown**.
DECLARE_LIBID	Provides a way for ATL to obtain the *libid* of the type library.
DECLARE_NO_REGISTRY	Use **DECLARE_NO_REGISTRY** if you want to avoid any default ATL registration for the class in which this macro appears.
DECLARE_NOT_AGGREGATABLE	Specifies that your object cannot be aggregated.
DECLARE_OBJECT_DESCRIPTION	Allows you to specify a text description for your class object.
DECLARE_OLEMISC_STATUS	Used in ATL ActiveX controls to set the OLEMISC flags.
DECLARE_ONLY_AGGREGATABLE	Specifies that your object must be aggregated.
DECLARE_POLY_AGGREGATABLE	Specifies that an instance of **CComPolyObject** < *x* > is created when your object is created.
DECLARE_PROTECT_FINAL_CONSTRUCT	Protects your object from being deleted if (during FinalConstruct) the internal aggregated object increments the reference count then decrements the count to 0.
DECLARE_REGISTRY	Enters the standard class registration into the system registry or removes it from the system registry.
DECLARE_REGISTRY_APPID_RESOURCEID	Specifies the information required to automatically register the *appid*.
DECLARE_REGISTRY_RESOURCE	Gets the named resource containing the registry file and runs the script to either enter objects into the system registry or remove them from the system registry.
DECLARE_REGISTRY_RESOURCEID	Same as DECLARE_REGISTRY_RESOURCE except that it uses a wizard-generated **UINT** to identify the resource, rather than a string name.
DECLARE_VIEW_STATUS	Place this macro in an ATL ActiveX control's control class to specify the **VIEWSTATUS** flags to the container.
DECLARE_WND_CLASS	Allows you to specify the name of a new window class. Place this macro in an ATL ActiveX control's control class.

DECLARE_WND_CLASS_EX	Allows you to specify the name of an existing window class on which a new window class will be based. Place this macro in an ATL ActiveX control's control class.
DECLARE_WND_SUPERCLASS	Allows you to specify the parameters of a class. Place this macro in an ATL ActiveX control's control class.
DEFAULT_REFLECTION_HANDLER	Provides a default handler for the child window (control) that will receive reflected messages; the handler will properly pass unhandled messages to **DefWindowProc**.
DEVMODE and TEXTMETRIC String Conversion Macros	These macros create a copy of a DEVMODE or TEXTMETRIC structure and convert the strings within the new structure to a new string type.
END_CATEGORY_MAP	Marks the end of the category map.
END_COM_MAP	Ends the definition of your COM interface map.
END_CONNECTION_POINT_MAP	Marks the end of the connection point map entries.
END_EXTENSION_SNAPIN_NODEINFO_MAP	Marks the end of the snap-in extension data class map.
END_MSG_MAP	Marks the end of a message map.
END_PROP_MAP	Marks the end of the object's property map.
END_RDX_MAP	Marks the end of the Registry Data Exchange map.
END_SERVICE_MAP	Marks the end of the service map.
END_SINK_MAP	Declares the end of the event sink map for the composite control.
END_SNAPINTOOLBARID_MAP	Declares the end of the toolbar ID map for the Snap-In object.
EXTENSION_SNAPIN_DATACLASS	Adds a data member to the snap-in extension data class for an **ISnapInItemImpl**-derived class.
EXTENSION_SNAPIN_NODEINFO_ENTRY	Adds a snap-in extension data class to the snap-in extension data class map.
FORWARD_NOTIFICATIONS	Forwards notification messages to the parent window.
IMPLEMENTED_CATEGORY	Add an **IMPLEMENTED_CATEGORY** macro to your component's category map to specify that it should be registered as implementing the category identified by the *catID* parameter.
MESSAGE_HANDLER	Defines an entry in a message map.
MESSAGE_RANGE_HANDLER	Similar to MESSAGE_HANDLER, but maps a range of Windows messages to a single handler function.
NOTIFY_CODE_HANDLER	Similar to NOTIFY_HANDLER, but maps a WM_NOTIFY message based only on the notification code.

NOTIFY_HANDLER	Defines an entry in a message map.
NOTIFY_ID_HANDLER	Similar to NOTIFY_HANDLER, but maps a WM_NOTIFY message based only on the control identifier.
NOTIFY_RANGE_CODE_HANDLER	Similar to NOTIFY_RANGE_HANDLER, but maps WM_NOTIFY messages with a specific notification code from a range of controls to a single handler function.
NOTIFY_RANGE_HANDLER	Similar to NOTIFY_HANDLER, but maps WM_NOTIFY messages from a range of controls to a single handler function.
OBJECT_ENTRY_AUTO	Enters an ATL object into the object map, updates the registry, and creates an instance of the object.
OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO	Allows you to specify that the object should be registered and initialized, but it should not be externally creatable via **CoCreateInstance**.
PROP_DATA_ENTRY	Indicates the extent, or dimensions, of an ActiveX control.
PROP_ENTRY	Use this macro to enter a property description, property DISPID, and property page CLSID into the object's property map.
PROP_ENTRY_EX	Similar to PROP_ENTRY, but allows you specify a particular IID if your object supports multiple dual interfaces.
PROP_PAGE	Use this macro to enter a property page CLSID into the object's property map.
RDX_BINARY	Associates the specified registry entry with a specified member variable of type BYTE.
RDX_CSTRING_TEXT	Associates the specified registry entry with a specified member variable of type CString.
RDX_DWORD	Associates the specified registry entry with a specified member variable of type DWORD.
RDX_TEXT	Associates the specified registry entry with a specified member variable of type TCHAR.
REFLECT_NOTIFICATIONS	Reflects notification messages back to the child window (control) that sent them.
REFLECTED_COMMAND_CODE_HANDLER	Similar to COMMAND_CODE_HANDLER, but maps commands reflected from the parent window.
REFLECTED_COMMAND_HANDLER	Similar to COMMAND_HANDLER, but maps commands reflected from the parent window.
REFLECTED_COMMAND_ID_HANDLER	Similar to COMMAND_ID_HANDLER, but maps commands reflected from the parent window.
REFLECTED_COMMAND_RANGE_CODE_HANDLER	Similar to COMMAND_RANGE_CODE_HANDLER, but maps commands reflected from the parent window.

REFLECTED_COMMAND_RANGE_HANDLER	Similar to COMMAND_RANGE_HANDLER, but maps commands reflected from the parent window.
REFLECTED_NOTIFY_CODE_HANDLER	Similar to NOTIFY_CODE_HANDLER, but maps notifications reflected from the parent window.
REFLECTED_NOTIFY_HANDLER	Similar to NOTIFY_HANDLER, but maps notifications reflected from the parent window.
REFLECTED_NOTIFY_ID_HANDLER	Similar to NOTIFY_ID_HANDLER, but maps notifications reflected from the parent window.
REFLECTED_NOTIFY_RANGE_CODE_HANDLER	Similar to NOTIFY_RANGE_CODE_HANDLER, but maps notifications reflected from the parent window.
REFLECTED_NOTIFY_RANGE_HANDLER	Similar to NOTIFY_RANGE_HANDLER, but maps notifications reflected from the parent window.
REQUIRED_CATEGORY	Add a **REQUIRED_CATEGORY** macro to your component's category map to specify that it should be registered as requiring the category identified by the *catID* parameter.
SERVICE_ENTRY	Indicates that the object supports the service id specified by *SID*.
SERVICE_ENTRY_CHAIN	Instructs IServiceProviderImpl::QueryService to chain to the object specified by *punk*.
SINK_ENTRY	Declares the handler function (*fn*) for the specified event (*dispid*), of the control identified by *id*.
SINK_ENTRY_EX	Declares the handler function (*fn*) for the specified event (*dispid*), of the dispatch interface (*iid)*, for the control identified by *id*.
SINK_ENTRY_INFO	Use the **SINK_ENTRY_INFO** macro within an event sink map to provide the information needed by IDispEventSimpleImpl to route events to the relevant handler function.
SNAPINMENUID	Use this macro to declare the context menu resource of the Snap-In object.
SNAPINTOOLBARID_ENTRY	Use this macro to enter a toolbar ID into the Snap-In object's toolbar ID map.
WM_FORWARDMSG	This macro forwards a message received by a window to another window for processing.

See Also

Other Resources

ATL Macros
ATL Reference
ATL Functions
ATL Global Variables
ATL Structures
ATL Typedefs
ATL Class Overview

_ATL_ALL_WARNINGS

A symbol which enables errors in projects converted from previous versions of ATL.

```
#define _ATL_ALL_WARNINGS
```

Remarks

Before version 7.0, ATL disabled a lot of warnings and left them disabled so that they never showed up in user code. Specifically:

- C4127 conditional expression is constant
- C4786 'identifier' : identifier was truncated to 'number' characters in the debug information
- C4201 nonstandard extension used : nameless struct/union
- C4103 'filename' : used #pragma pack to change alignment
- C4291 'declaration' : no matching operator delete found; memory will not be freed if initialization throws an exception
- C4268 'identifier' : 'const' static/global data initialized with compiler generated default constructor fills the object with zeros
- C4702 unreachable code

In projects converted from previous versions, these warnings are still disabled by the libraries headers.

By adding the following line to the stdafx.h file before including libraries headers, this behavior can be changed.

```
#define _ATL_ALL_WARNINGS
```

If this `#define` is added, the ATL headers are careful to preserve the state of these warnings so that they are not disabled globally (or if the user explicitly disables individual warnings, not to enable them).

New projects generated with version 7.0 will have this `#define` set in stdafx.h by default.

See Also

Other Resources

Compiler Options Macros

_ATL_APARTMENT_THREADED

Define if one or more of your objects use apartment threading.

```
_ATL_APARTMENT_THREADED
```

Remarks

Specifies apartment threading. See Specifying the Project's Threading Model for other threading options, and Options, ATL Simple Object Wizard for a description of the threading models available for an ATL object.

See Also

Other Resources

Compiler Options Macros

ATL Macros

_ATL_CSTRING_EXPLICIT_CONSTRUCTORS

Makes certain **CString** constructors explicit, preventing any unintentional conversions.

```
_ATL_CSTRING_EXPLICIT_CONSTRUCTORS
```

See Also

Reference

CStringT Class

Other Resources

Compiler Options Macros

_ATL_DEBUG_INTERFACES

Define this macro before including any ATL header files to trace all **AddRef** and **Release** calls on your components' interfaces to the output window.

```
#define _ATL_DEBUG_INTERFACES
```

Remarks

The trace output will appear as shown below:

```
ATL: QIThunk - 2008         AddRef  :   Object = 0x00d81ba0   Refcount = 1   CBug - IBug
```

The first part of each trace will always be `ATL: QIThunk`. Next is a value identifying the particular *interface thunk* being used. An interface thunk is an object used to maintain a reference count and provide the tracing capability used here. A new interface thunk is created on every call to **QueryInterface** except for requests for the **IUnknown** interface (in this case, the same thunk is returned every time to comply with COM's identity rules).

Next you'll see **AddRef** or **Release** indicating which method was called. Following that, you'll see a value identifying the object whose interface reference count was changed. The value traced is the **this** pointer of the object.

The reference count that is traced is the reference count on that thunk after **AddRef** or **Release** was called. Note that this reference count may not match the reference count for the object. Each thunk maintains its own reference count to help you fully comply with COM's reference-counting rules.

The final piece of information traced is the name of the object and the interface being affected by the **AddRef** or **Release** call.

Any interface leaks that are detected when the server shuts down and `_Module.Term` is called will be logged like this:

```
ATL: QIThunk - 2005         LEAK    :   Object = 0x00d81ca0   Refcount = 1   MaxRefCount =
1   CBug - IBug
```

The information provided here maps directly to the information provided in the previous trace statements, so you can examine the reference counts throughout the whole lifetime of an interface thunk. In addition, you get an indication of the maximum reference count on that interface thunk.

Note
_ATL_DEBUG_INTERFACES can be used in retail builds.

Tips

- You can see when a new interface thunk is created by looking for `AddRef` calls with a `Refcount` of 1.
- You can see when an interface thunk is destroyed by looking for `Release` calls with a `Refcount` of 0.
- The trace statements are output in a tab-separated format so you can easily cut and paste the information into a spreadsheet such as Microsoft Excel to provide advanced filtering, searching, and sorting capabilities.
- If your code seems to work fine when **_ATL_DEBUG_INTERFACES** is not defined, but causes access violations when the macro is defined, you almost certainly have a mismatched reference-counting bug in your client code, similar to the one shown below:

  ```
  IBug* pBug = NULL;
  hr = p->QueryInterface(&pBug);
  _ASSERTE(SUCCEEDED(hr));
  IBug* pBug2 = NULL;
  hr = p->QueryInterface(&pBug2);
  _ASSERTE(SUCCEEDED(hr));
  pBug->Release();
  ```

```
pBug->Release();    // Mismatched - should be pBug2->Release();
```

This code may appear to work under some common circumstances, but it is clearly buggy. For this code to work at all, it relies on implementation details of the COM object providing the `IBug` interface (`IBug` cannot be implemented as a tear-off interface) and it is location-dependent (the client must be in the same apartment as the `IBug` implementation). Use of the **_ATL_DEBUG_INTERFACES** macro can bring such bugs to light.

See Also

Other Resources

Debugging and Error Reporting Macros

ATL Macros

_ATL_DEBUG_QI

Writes all calls to **QueryInterface** to the output window.

```
#define _ATL_DEBUG_QI
```

Remarks

If a call to **QueryInterface** failed, the output window will display:

```
        interface name - failed
```

See Also

Other Resources

Debugging and Error Reporting Macros

ATL Macros

_ATL_ENABLE_PTM_WARNING

Define this macro in order to force the use of ANSI C++ standard-compliant syntax for pointer to member functions. Using this macro will cause the C4867 compiler error to be generated when non-standard syntax is used to initialize a pointer to a member function.

```
#define _ATL_ENABLE_PTM_WARNING
```

Remarks

The ATL and MFC libraries have been changed to match the Visual C++ compiler's improved standard C++ compliance. According to the ANSI C++ standard, the syntax of a pointer to a class member function should be `&CMyClass::MyFunc`.

When _ATL_ENABLE_PTM_WARNING is not defined (the default case), ATL/MFC disables the C4867 error in macro maps (notably message maps) to allow legacy code to continue building as before. If you define **_ATL_ENABLE_PTM_WARNING**, your code should be C++ standard compliant.

However, the non-standard form has been deprecated, so you need to move existing code to C++ standard compliant syntax. For example, the following:

```
ON_COMMAND(ID_FUNC1, OnFunc1)
```

Should be changed to:

```
ON_COMMAND(ID_FUNC1, &MyClass::OnFunc1)
```

Note that for map macros that add the '&' character, you should not add it again in your code.

See Also

Reference

Compiler Warning C4867

Other Resources

Compiler Options Macros

_ATL_FREE_THREADED

Define if one or more of your objects use free or neutral threading.

```
_ATL_FREE_THREADED
```

Remarks

Specifies free threading. Free threading is equivalent to a multithread apartment model. See Specifying the Project's Threading Model for other threading options, and Options, ATL Simple Object Wizard for a description of the threading models available for an ATL object.

See Also

Other Resources

Compiler Options Macros

ATL Macros

_ATL_MIXED

_ATL_MIXED specifies that your ATL application is comprised of native and **/clr** compilands.

```
_ATL_MIXED
```

Remarks

When building an application that uses ATL, and where the compilands are not all native or all **/clr**, then you must add this macro to each compiland.

```
#define _ATL_MIXED
```

This will cause all startup code to run as native. You may need to ensure that all startup code is native. See Initialization of Mixed Assemblies for more information.

See Also

Other Resources

Compiler Options Macros

ATL Macros

_ATL_MULTI_THREADED

A symbol that indicates the project will have objects that are marked as Both, Free or Neutral.

```
_ATL_MULTI_THREADED
```

Remarks

If this symbol is defined, ATL will pull in code that will correctly synchronize access to global data. New code should use the equivalent macro _ATL_FREE_THREADED instead.

See Also

Other Resources

Compiler Options Macros

ATL Macros

_ATL_NO_AUTOMATIC_NAMESPACE

A symbol which prevents the default use of namespace as ATL.

```
_ATL_NO_AUTOMATIC_NAMESPACE
```

Remarks

If this symbol is not defined, including atlbase.h will perform **using namespace ATL** by default, which may lead to naming conflicts. To prevent this, define this symbol.

See Also

Other Resources

Compiler Options Macros

ATL Macros

_ATL_NO_COM_SUPPORT

A symbol which prevents COM-related code from being compiled with your project.

```
_ATL_NO_COM_SUPPORT
```

See Also

Other Resources

Compiler Options Macros

ATL Macros

_ATL_SECURE_NO_WARNINGS

Suppresses compiler warnings for the use of deprecated ATL functions.

```
#define _ATL_SECURE_NO_WARNINGS
```

Example

This code sample would cause a compiler warning if _ATL_SECURE_NO_WARNINGS were not defined.

```
// ATL_SECURE_NO_WARNINGS.cpp
#define _ATL_SECURE_NO_WARNINGS
#include<atlbase.h>
#include<atlsimpstr.h>

int main() {
   IAtlStringMgr *pMgr = NULL;

   CSimpleString str( "xxxxxxxxxxxxxxxxxxx", 20, pMgr );

   char* pszSrc= NULL;
   pszSrc= new char(12);
   str.CopyChars(str.GetBuffer(), pszSrc, 12);
}
```

See Also

Other Resources

Compiler Options Macros

ATL Macros

_ATL_SINGLE_THREADED

Define if all of your objects use the single threading model

```
_ATL_SINGLE_THREADED
```

Remarks

Specifies that the object always runs in the primary COM thread. See Specifying the Project's Threading Model for other threading options, and Options, ATL Simple Object Wizard for a description of the threading models available for an ATL object.

See Also

Other Resources

Compiler Options Macros

ATL Macros

_ATL_STATIC_REGISTRY

A symbol that indicates you want the registration code for your object to be in the object to avoid a dependency on ATL.DLL.

```
#define _ATL_STATIC_REGISTRY
```

Remarks

When you define **ATL_STATIC_REGISTRY**, you should use the following code:

```
#ifdef _ATL_STATIC_REGISTRY
#include <statreg.h>
#endif
```

See Also

Reference

DECLARE_STATIC_REGISTRY_RESOURCEID
DECLARE_REGISTRY_RESOURCE
DECLARE_STATIC_REGISTRY_RESOURCE
DECLARE_REGISTRY_RESOURCEID

Other Resources

Registry Macros
ATL Macros

_ATLCATCH

Statement(s) to handle errors occurring in the associated **_ATLTRY**.

```
_ATLCATCH( e )
```

Parameters

e

The exception to catch.

Remarks

Used in conjunction with **_ATLTRY**. Resolves to C++ catch(CAtlException e) for handling a given type of C++ exceptions.

See Also

Reference

_ATLTRY

_ATLCATCHALL

Other Resources

Exception Handling Macros

_ATLCATCHALL

Statement(s) to handle errors occurring in the associated **_ATLTRY**.

```
_ATLCATCHALL
```

Remarks

Used in conjunction with **_ATLTRY**. Resolves to C++ catch(...) for handling all types of C++ exceptions.

See Also

Reference

_ATLTRY

_ATLCATCH

Other Resources

Exception Handling Macros

_ATLTRY

Marks a guarded code section where an error could possibly occur.

```
_ATLTRY
```

Remarks

Used in conjunction with **_ATL_CATCH** or **_ATL_CATCHALL**. Resolves to the C++ symbol try.

See Also

Reference

_ATLCATCH

_ATLCATCHALL

Other Resources

Exception Handling Macros

ALT_MSG_MAP

Marks the beginning of an alternate message map.

```
ALT_MSG_MAP( msgMapID )
```

Parameters

msgMapID

[in] The message map identifier.

Remarks

ATL identifies each message map by a number. The default message map (declared with the **BEGIN_MSG_MAP** macro) is identified by 0. An alternate message map is identified by *msgMapID*.

Message maps are used to process messages sent to a window. For example, CContainedWindow allows you to specify the identifier of a message map in the containing object. CContainedWindow::WindowProc then uses this message map to direct the contained window's messages either to the appropriate handler function or to another message map. For a list of macros that declare handler functions, see BEGIN_MSG_MAP.

Always begin a message map with **BEGIN_MSG_MAP**. You can then declare subsequent alternate message maps.

The END_MSG_MAP macro marks the end of the message map. Note that there is always exactly one instance of **BEGIN_MSG_MAP** and **END_MSG_MAP**.

For more information about using message maps in ATL, see Message Maps.

Example

The following example shows the default message map and one alternate message map, each containing one handler function:

```
BEGIN_MSG_MAP(CMyClass)
   MESSAGE_HANDLER(WM_PAINT, OnPaint)
ALT_MSG_MAP(1)
   MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
END_MSG_MAP()
```

The next example shows two alternate message maps. The default message map is empty.

```
BEGIN_MSG_MAP(CMyClass)
ALT_MSG_MAP(1)
   MESSAGE_HANDLER(WM_PAINT, OnPaint)
   MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
ALT_MSG_MAP(2)
   MESSAGE_HANDLER(WM_CREATE, OnCreate)
END_MSG_MAP()
```

See Also

Reference

MESSAGE_HANDLER
CMessageMap Class
CDynamicChain Class

Other Resources

Message Map Macros (ATL)
ATL Macros

ATL and MFC String Conversion Macros

The string conversion macros discussed here are valid for both ATL and MFC. For more information on MFC string conversion, see TN059: Using MFC MBCS/Unicode Conversion Macros and MFC Macros and Globals.

- ATL 7.0 String Conversion Classes and Macros
- ATL 3.0 String Conversion Macros

ATL 7.0 String Conversion Classes and Macros

ATL 7.0 introduces several new conversion classes and macros, providing significant improvements over the existing macros.

The names of the new string conversion classes and macros take the form:

```
CSourceType2[C]DestinationType[EX]
```

where:

- *SourceType* and *DestinationType* are described in the table below.
- [**C**] is present when the destination type must be constant.
- [**EX**] is present when the initial size of the buffer must be specified as a template argument.

SourceType/DestinationType	Description
A	ANSI character string.
W	Unicode character string.
T	Generic character string (equivalent to W when _UNICODE is defined, equivalent to A otherwise).
OLE	OLE character string (equivalent to W).

For example, to convert from a Unicode string to a generic string without changing the converted string, use **CW2CT**.

If it is known that the converted string is unlikely to be more than 64 characters, the **EX** version, such as **CW2CTEX<64>**, can be used to save space on the stack.

Note

The recommended way of converting to and from BSTR strings is to use the CComBSTR class. To convert to a BSTR, pass the existing string to the constructor of **CComBSTR.** To convert from a BSTR, use **COLE2**[**C**]*DestinationType*[**EX**], such as **COLE2T**.

The new conversion classes which require a buffer (**CA2AEX**, **CA2WEX**, **CW2AEX**, and **CW2WEX**) use a fixed-size static buffer to store the result of the conversion. If the result is too large to fit into the static buffer, the class allocates memory using **malloc**, freeing the memory when the object goes out of scope. This ensures that, unlike the older text conversion macros, these classes are safe to use in loops and won't overflow the stack.

The conversion macros introduced in ATL 7.0 are optimized to be aware of input **NULL** strings. These macros will return **NULL** if the input parameter is **NULL** without allocating any memory.

By default, the ATL conversion classes and macros will use the current thread's ANSI code page for the conversion. If you want to override that behavior for a specific conversion using macros based on the classes **CA2WEX** or **CW2AEX**, specify the code page as the second parameter to the constructor for the class.

Security Note

Check the length of the strings before passing them to these macros to avoid potential buffer overrun problems. Stack overflows are exceptions that could also be caught with try/except.

There are several important differences between the older string conversion macros and the new string conversion classes:

Old ATL 3.0 Conversion Macros	**New ATL 7.0 Conversion Classes**
Allocates memory on the stack.	Uses stack memory for small strings. Uses the heap if the stack is not large enough.
The string is freed when the function is exited.	The string is freed when the variable goes out of scope.
Cannot be used in exception handlers.	Can be used in exception handlers.
Not suitable for use in loops. Memory use grows until the function is exited.	Supports use in loops. Loop scope ensures that memory is freed on each iteration.
Not good for large strings. Stack space is limited.	No problems with large strings. Strings will be allocated on the heap.
Usually require USES_CONVERSION to be defined.	Never require USES_CONVERSION to be defined.
Meaning of OLE depends on definition of OLE2ANSI.	OLE is always equivalent to W.

Example

```
//Example 1
// Convert LPCWSTR to LPCSTR.
void ExampleFunctionW( LPCWSTR pszW )
{
   // Create an instance of CW2A, called pszA,
   // and initialize it with pszW.
   CW2A pszA( pszW );
   // pszA works like an LPCSTR, and can be used thus:
   ExampleFunctionA( pszA );
   // Note: pszA will become invalid when it goes out of scope.
}

// Example 2
// Use a temporary instance of CW2A.
void ExampleFunctionW( LPCWSTR pszW )
{
   // Create a temporary instance of CW2A,
   // and initialize it with pszW.
   ExampleFunctionA( CW2A( pszW ) );
   // Note: the temporary instance becomes invalid
   // after the execution of the statement above.
}

// Example 3
// Incorrect use of conversion macros.
void ExampleFunctionW( LPCWSTR pszW )
{
   // Create a temporary instance of CW2A,
   // save a pointer to it and then delete
   // the temportary instance.
   LPCSTR pszA = CW2A( pszW );
   // The pszA in the following line is an invalid pointer,
   // as the instance of CW2A has gone out of scope.
   ExampleFunctionA( pszA );
}
```

A Warning Regarding Temporary Class Instances

It should be stressed that the following is not good code:

```
LPCTSTR szr = CA2T( szReplaceFile );
```

Using ATL 3.0 macros, it was acceptable to use:

```
LPCTSTR szr = A2T( szReplaceFile );
```

as the memory allocated by the conversion functions would not be freed until the current function was exited. The same code does not work with the new classes.

This code:

```
LPCTSTR szr = CA2T( szReplaceFile );
```

is equivalent to this:

```
LPCTSTR szr;
{
   CA2T temp( szReplaceFile );
   szr = temp.operator LPTSTR();
}
```

As the memory allocated by the temporary object and returned from the cast operator is destroyed when the temporary object is destroyed, using the value in `szr` will have undesirable results.

Instead, use this code:

```
CA2T szr( szReplaceFile );
```

The cast operator makes the CA2T object look like a **LPCTSTR**.

Advanced Usage

The default static buffer size is 128 characters. If the buffer size must be changed for a specific conversion, use the EX version of a macro, and specify the buffer size as a template argument.

```
// Example 4
// Changing the size of the buffer.
void ExampleFunctionW( LPCWSTR pszW )
{
   // Use a 16-character buffer.
   ExampleFunctionA( CW2CAEX< 16 >( pszW ) );
}
```

Here is an example of specifying the code page as the second parameter to the constructor for the class.

```
// Example 5
// Specifying the code page.
void ExampleFunctionW( LPCWSTR pszW )
{
   // Convert to the Macintosh code page
   ExampleFunctionA( CW2A( pszW, CP_MACCP ) );
}
```

ATL 3.0 String Conversion Macros

The original text conversion macros are still available and are listed in the table below:

ATL 3.0 String Conversion Macros

A2BSTR	**OLE2A**	**T2A**	**W2A**
A2COLE	**OLE2BSTR**	**T2BSTR**	**W2BSTR**
A2CT	**OLE2CA**	**T2CA**	**W2CA**
A2CW	**OLE2CT**	**T2COLE**	**W2COLE**
A2OLE	**OLE2CW**	**T2CW**	**W2CT**
A2T	**OLE2T**	**T2OLE**	**W2OLE**
A2W	**OLE2W**	**T2W**	**W2T**

The syntax for using these macros is as follows:

```
MACRONAME( string_address )
```

For example:

```
A2W( lpa )
```

In the macro names, the source string type is on the left (for example, **A**) and the destination string type is on the right (for example, **W**). **A** stands for **LPSTR**, **OLE** stands for **LPOLESTR**, **T** stands for **LPTSTR**, and **W** stands for **LPWSTR**.

If there is a **C** in the macro name, the macro converts to a **const** string. For example, **W2CA** converts an **LPWSTR** to an **LPCSTR**.

Thus, **A2W** converts an **LPSTR** to an **LPWSTR**, **OLE2T** converts an **LPOLESTR** to an **LPTSTR**, and so on.

The behavior of the ATL string conversion macros depends on the compiler directive in effect, if any. If the source and destination types are the same, no conversion takes place. Compiler directives change **T** and **OLE** as follows:

Compiler directive in effect	T becomes	OLE becomes
none	**A**	**W**
_UNICODE	**W**	**W**
OLE2ANSI	**A**	**A**
_UNICODE and **OLE2ANSI**	**W**	**A**

The destination string is created using **_alloca**, except when the destination type is **BSTR**. Using **_alloca** allocates memory off the stack, so that when your function returns, it is automatically cleaned up. By default this macro will only convert up to 500KB at one time.

When using an ATL string conversion macro, specify the **USES_CONVERSION** macro at the beginning of your function in order to avoid compiler errors. For example:

```
void func( LPSTR lpsz )
{
   USES_CONVERSION;
   ...
   LPWSTR x = A2W(lpsz)
   // Do something with x
   ...
}
```

Requirements

Header file: AtlBase.h, AtlConv.h (declared in AtlConv.h)

See Also

Reference

DEVMODE and TEXTMETRIC String Conversion Macros

Other Resources

ATL Macros

ATL_NO_VTABLE

A symbol that prevents the vtable pointer from being initialized in the class's constructor and destructor.

```
ATL_NO_VTABLE
```

Remarks

If the vtable pointer is prevented from being initialized in the class's constructor and destructor, the linker can eliminate the vtable and all of the functions to which it points. Expands to **__declspec(novtable)**.

Example

```
class ATL_NO_VTABLE MyClass   // MyClass will not have a vtable
{
   // ...
};
```

See Also

Concepts

Specifying Compiler Optimization for an ATL Project

Other Resources

Compiler Options Macros

ATL Macros

ATL_NOINLINE

A symbol that indicates a function should not be inlined.

```
ATL_NOINLINE inline myfunction
{
...
}
```

Parameters

myfunction

The function that should not be inlined.

Remarks

Use this symbol if you want to ensure a function does not get inlined by the compiler, even though it must be declared as inline so that it can be placed in a header file. Expands to **__declspec(noinline)**.

See Also

Other Resources

Compiler Options Macros

ATL Macros

ATLASSERT

The **ATLASSERT** macro performs the same functionality as the _ASSERTE macro found in the C run-time library.

```
ATLASSERT( booleanExpression );
```

Parameters

booleanExpression

Expression (including pointers) that evaluates to nonzero or 0.

Remarks

In debug builds, **ATLASSERT** evaluates *booleanExpression* and generates a debug report when the result is false.

See Also

Other Resources

Debugging and Error Reporting Macros

ATL Macros

ATLENSURE

This macro is used to validate parameters passed to a function.

```
ATLENSURE(booleanExpression);
ATLENSURE_THROW(booleanExpression, hr);
```

Parameters

booleanExpression

Specifies a boolean expression to be tested.

hr

Specifies an error code to return.

Remarks

These macros provide a mechanism to detect and notify the user of incorrect parameter usage.

The macro calls **ATLASSERT** and if the condition fails calls **AtlThrow**.

In the **ATLENSURE** case, **AtlThrow** is called with E_FAIL.

In the **ATLENSURE_THROW** case, **AtlThrow** is called with the specified HRESULT.

The difference between **ATLENSURE** and **ATLASSERT** is that **ATLENSURE** throws an exception in Release builds as well as in Debug builds.

Example

```
// example for ATLENSURE
void CMyClass::GetKeyAt( POSITION pos ){
   ATLENSURE(pos != NULL);
   ...
}
```

See Also

Reference

ATLASSERT

ENSURE (MFC)

Other Resources

Debugging and Error Reporting Macros

ATL Macros

ATLTRACE (ATL)

Reports warnings to an output device, such as the debugger window, according to the indicated flags and levels. Included for backward compatibility.

```
ATLTRACE( exp );
ATLTRACE(
   DWORD category,
   UINT level,
   LPCSTR lpszFormat,
...
);
```

Parameters

exp

[in] The string and variables to send to the Visual C++ output window or any application that traps these messages.

category

[in] Type of event or method on which to report. See the Remarks for a list of categories.

level

[in] The level of tracing to report. See the Remarks for details.

lpszFormat

[in] The formatted string to send to the dump device.

Remarks

See ATLTRACE2 for a description of **ATLTRACE**. **ATLTRACE** and **ATLTRACE2** have the same behavior, **ATLTRACE** is included for backward compatibility.

See Also

Reference

ATLTRACE2

Other Resources

Debugging and Error Reporting Macros

ATLTRACE2

Reports warnings to an output device, such as the debugger window, according to the indicated flags and levels.

```
ATLTRACE2( exp );
ATLTRACE2(
   DWORD category,
   UINT level,
   LPCSTR lpszFormat,
   ...
);
```

Parameters

exp

[in] The string and variables to send to the Visual C++ output window or any application that traps these messages.

category

[in] Type of event or method on which to report. See the Remarks for a list of categories.

level

[in] The level of tracing to report. See the Remarks for details.

lpszFormat

[in] The formatted string to send to the dump device.

Remarks

The short form of **ATLTRACE2** writes output to the debugger's output window. The second form of **ATLTRACE2** also writes output to the debugger's output window, but is subject to the settings of the ATL/MFC Trace Tool (see ATLTraceTool Sample). For example, if you set *level* to 4 and the ATL/MFC Trace Tool to level 0, you will not see the message. *level* can be 0, 1, 2, 3, or 4. The default, 0, reports only the most serious problems.

The *category* parameter lists the trace flags to set. These flags correspond to the types of methods for which you want to report. The tables below list the valid trace flags you can use for the *category* parameter.

ATL Trace Flags

ATL Category	Description
atlTraceGeneral	Reports on all ATL applications. The default.
atlTraceCOM	Reports on COM methods.
atlTraceQI	Reports on QueryInterface calls.
atlTraceRegistrar	Reports on the registration of objects.
atlTraceRefcount	Reports on changing reference count.
atlTraceWindowing	Reports on windows methods; for example, reports an invalid message map ID.
atlTraceControls	Reports on controls; for example, reports when a control or its window is destroyed.
atlTraceHosting	Reports hosting messages; for example, reports when a client in a container is activated.
atlTraceDBClient	Reports on OLE DB Consumer Template; for example, when a call to GetData fails, the output can contain the HRESULT.

atlTraceDBProvider	Reports on OLE DB Provider Template; for example, reports if the creation of a column failed.
atlTraceSnapin	Reports for MMC SnapIn application.
atlTraceNotImpl	Reports that the indicated function is not implemented.
atlTraceAllocation	Reports messages printed by the memory debugging tools in atldbgmem.h.

MFC Trace Flags

MFC Category	Description
traceAppMsg	General purpose, MFC messages. Always recommended.
traceDumpContext	Messages from CDumpContext.
traceWinMsg	Messages from MFC's message handling code.
traceMemory	Messages from MFC's memory management code.
traceCmdRouting	Messages from MFC's Windows command routing code.
traceHtml	Messages from MFC's DHTML dialog support.
traceSocket	Messages from MFC's socket support.
traceOle	Messages from MFC's OLE support.
traceDatabase	Messages from MFC's database support.
traceInternet	Messages from MFC's Internet support.

To declare a custom trace category, declare a global instance of the **CTraceCategory** class as follows:

```
CTraceCategory MY_CATEGORY( "MyCategoryName", 1 );
```

The category name, `MY_CATEGORY` in this example, is the name you specify to the *category* parameter. The first parameter is the category name that will appear in the ATL/MFC Trace Tool. The second parameter is the default trace level. This parameter is optional, and the default trace level is 0.

To use a user-defined category:

```
ATLTRACE2( MY_CATEGORY, 2, "a message in a custom category" );
```

To specify that you want to filter the trace messages, insert definitions for these macros into Stdafx.h before the `#include <atlbase.h>` statement.

Alternatively, you can set the filter in the preprocessor directives in the **Property Pages** dialog box. Click the **Preprocessor** tab and then insert the global into the **Preprocessor Definitions** edit box.

Atlbase.h contains default definitions of the **ATLTRACE2** macros and these definitions will be used if you don't define these symbols before atlbase.h is processed.

In release builds, **ATLTRACE2** compiles to `(void) 0`.

ATLTRACE2 limits the contents of the string to be sent to the dump device to no more than 1023 characters, after formatting.

ATLTRACE and **ATLTRACE2** have the same behavior, **ATLTRACE** is included for backward compatibility.

Example

```
// example for ATLTRACE2
int i = 1;
ATLTRACE2( atlTraceGeneral, 4, "Integer = %d\n", i );
// Output: 'Integer = 1'
```

See Also

Reference

ATLTRACE (ATL)

Other Resources

Debugging and Error Reporting Macros

ATLTRACENOTIMPL

In debug builds of ATL, sends the string "*funcname* is not implemented" to the dump device and returns **E_NOTIMPL**.

```
ATLTRACENOTIMPL( funcname );
```

Parameters

funcname

[in] A string containing the name of the function that is not implemented.

Remarks

In release builds, simply returns **E_NOTIMPL**.

Example

```
ATLTRACENOTIMPL(_T("IOleControl::GetControlInfo"));
```

See Also

Reference

ATLTRACE2

Other Resources

Debugging and Error Reporting Macros

ATL Macros

BEGIN_CATEGORY_MAP

Marks the beginning of the category map.

```
BEGIN_CATEGORY_MAP( theClass )
```

Parameters

theClass

[in] The name of the class containing the category map.

Remarks

The category map is used to specify which component categories the COM class will implement and which categories it requires from its container.

Add an IMPLEMENTED_CATEGORY entry to the map for each category implemented by the COM class. Add a REQUIRED_CATEGORY entry to the map for each category that the class requires its clients to implement. Mark the end of the map with the END_CATEGORY_MAP macro.

The component categories listed in the map will be registered automatically when the module is registered if the class has an associated OBJECT_ENTRY_AUTO or OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO.

Note

ATL uses the standard component categories manager to register component categories. If the manager is not present on th e system when the module is registered, registration succeeds, but the component categories will not be registered for that cl ass.

For more information about component categories, see What are Component Categories and how do they work? in the *Platform SDK*.

Example

```
BEGIN_CATEGORY_MAP(CMyControl)
    IMPLEMENTED_CATEGORY(CATID_Insertable)
END_CATEGORY_MAP()
```

Smart Device Developer Notes

Not supported for smart device projects.

See Also

Reference

IMPLEMENTED_CATEGORY
REQUIRED_CATEGORY
END_CATEGORY_MAP

Other Resources

Category Macros
ATL Macros

BEGIN_COM_MAP

The COM map is the mechanism that exposes interfaces on an object to a client through **QueryInterface**.

```
BEGIN_COM_MAP( x )
```

Parameters

x

[in] The name of the class object you are exposing interfaces on.

Remarks

CComObjectRootEx::InternalQueryInterface only returns pointers for interfaces in the COM map. Start your interface map with the **BEGIN_COM_MAP** macro, add entries for each of your interfaces with the COM_INTERFACE_ENTRY macro or one of its variants, and complete the map with the END_COM_MAP macro.

Example

From the ATL BEEPER sample:

```
BEGIN_COM_MAP(CBeeper)
   COM_INTERFACE_ENTRY(IDispatch)
   COM_INTERFACE_ENTRY(IBeeper)
   COM_INTERFACE_ENTRY_TEAR_OFF(IID_ISupportErrorInfo, CBeeper2)
END_COM_MAP( )
```

See Also

Other Resources

COM Map Macros

ATL Macros

BEGIN_CONNECTION_POINT_MAP

Marks the beginning of the connection point map entries.

```
BEGIN_CONNECTION_POINT_MAP( x )
```

Parameters

x

[in] The name of the class containing the connection points.

Remarks

Start your connection point map with the **BEGIN_CONNECTION_POINT_MAP** macro, add entries for each of your connection points with the CONNECTION_POINT_ENTRY macro, and complete the map with the END_CONNECTION_POINT_MAP macro.

For more information about connection points in ATL, see the article Connection Points.

Example

```
BEGIN_CONNECTION_POINT_MAP(CConnect)
   CONNECTION_POINT_ENTRY(m_cpInterfaceBeingExposed)
END_CONNECTION_POINT_MAP( )
```

See Also

Other Resources

Connection Point Macros

ATL Macros

BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP

Marks the beginning of the snap-in extension data class map.

```
BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP( classname )
```

Parameters

classname

[in] The name of the snap-in extension data class.

Remarks

Start your snap-in extension map with the **BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP** macro, add entries for each of your snap-in extension data types with the EXTENSION_SNAPIN_NODEINFO_ENTRY macro, and complete the map with the END_EXTENSION_SNAPIN_NODEINFO_MAP macro.

Example

```
class CMyExtSnapinExtData : public CSnapInItemImpl<CMyExtSnapinExtData>
{
...
};

...

class CMyExtSnapin :
   public CSnapInObjectRoot<1, CMyExtSnapin>,
   public IComponentDataImpl<CMyExtSnapin, CMyExtSnapinComponent>,
   ...
{
...
BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP(CMyExtSnapin)
   EXTENSION_SNAPIN_NODEINFO_ENTRY(CMyExtSnapinExtData)
END_EXTENSION_SNAPIN_NODEINFO_MAP()
...
};
```

See Also

Reference

END_EXTENSION_SNAPIN_NODEINFO_MAP
EXTENSION_SNAPIN_NODEINFO_ENTRY
EXTENSION_SNAPIN_DATACLASS

Other Resources

Snap-In Object Macros
ATL Macros

BEGIN_MSG_MAP

Marks the beginning of the default message map.

```
BEGIN_MSG_MAP( theClass )
```

Parameters

theClass

[in] The name of the class containing the message map.

Remarks

CWindowImpl::WindowProc uses the default message map to process messages sent to the window. The message map directs messages either to the appropriate handler function or to another message map.

The following macros map a message to a handler function. This function must be defined in *theClass*.

Macro	Description
MESSAGE_HANDLER	Maps a Windows message to a handler function.
MESSAGE_RANGE_HANDLER	Maps a contiguous range of Windows messages to a handler function.
COMMAND_HANDLER	Maps a **WM_COMMAND** message to a handler function, based on the notification code and the identifier of the menu item, control, or accelerator.
COMMAND_ID_HANDLER	Maps a **WM_COMMAND** message to a handler function, based on the identifier of the menu item, control, or accelerator.
COMMAND_CODE_HANDLER	Maps a **WM_COMMAND** message to a handler function, based on the notification code.
COMMAND_RANGE_HANDLER	Maps a contiguous range of **WM_COMMAND** messages to a handler function, based on the identifier of the menu item, control, or accelerator.
NOTIFY_HANDLER	Maps a **WM_NOTIFY** message to a handler function, based on the notification code and the control identifier.
NOTIFY_ID_HANDLER	Maps a **WM_NOTIFY** message to a handler function, based on the control identifier.
NOTIFY_CODE_HANDLER	Maps a **WM_NOTIFY** message to a handler function, based on the notification code.
NOTIFY_RANGE_HANDLER	Maps a contiguous range of **WM_NOTIFY** messages to a handler function, based on the control identifier.

The following macros direct messages to another message map. This process is called "chaining."

Macro	Description
CHAIN_MSG_MAP	Chains to the default message map in the base class.
CHAIN_MSG_MAP_MEMBER	Chains to the default message map in a data member of the class.
CHAIN_MSG_MAP_ALT	Chains to an alternate message map in the base class.
CHAIN_MSG_MAP_ALT_MEMBER	Chains to an alternate message map in a data member of the class.

CHAIN_MSG_MAP_DYNAMIC	Chains to the default message map in another class at run time.

The following macros direct "reflected" messages from the parent window. For example, a control normally sends notification messages to its parent window for processing, but the parent window can reflect the message back to the control.

Macro	Description
REFLECTED_COMMAND_HANDLER	Maps a reflected **WM_COMMAND** message to a handler function, based on the notification code and the identifier of the menu item, control, or acc elerator.
REFLECTED_COMMAND_ID_HANDLER	Maps a reflected **WM_COMMAND** message to a handler function, based on the identifier of the menu item, control, or accelerator.
REFLECTED_COMMAND_CODE_HANDLER	Maps a reflected **WM_COMMAND** message to a handler function, based on the notification code.
REFLECTED_COMMAND_RANGE_HANDLER	Maps a reflected **WM_COMMAND** message to a handler function, based on a contiguous range of control identifiers.
REFLECTED_COMMAND_RANGE_CODE_HANDLER	Maps a reflected **WM_COMMAND** message to a handler function, based on the notification code and a contiguous range of control identifiers.
REFLECTED_NOTIFY_HANDLER	Maps a reflected **WM_NOTIFY** message to a handler function, based on th e notification code and the control identifier.
REFLECTED_NOTIFY_ID_HANDLER	Maps a reflected **WM_NOTIFY** message to a handler function, based on th e control identifier.
REFLECTED_NOTIFY_CODE_HANDLER	Maps a reflected **WM_NOTIFY** message to a handler function, based on th e notification code.
REFLECTED_NOTIFY_RANGE_HANDLER	Maps a reflected **WM_NOTIFY** message to a handler function, based on a contiguous range of control identifiers.
REFLECTED_NOTIFY_RANGE_CODE_HANDLER	Maps a reflected **WM_NOTIFY** message to a handler function, based on th e notification code and a contiguous range of control identifiers.

Example

```
class CMyWindow : ...
{
public:
   ...

   BEGIN_MSG_MAP(CMyWindow)
      MESSAGE_HANDLER(WM_PAINT, OnPaint)
      MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
      CHAIN_MSG_MAP(CMyBaseWindow)
   END_MSG_MAP()

   LRESULT OnPaint(UINT uMsg, WPARAM wParam,
                   LPARAM lParam, BOOL& bHandled)
   { ... }

   LRESULT OnSetFocus(UINT uMsg, WPARAM wParam,
                      LPARAM lParam, BOOL& bHandled)
   { ... }
};
```

When a `CMyWindow` object receives a `WM_PAINT` message, the message is directed to `CMyWindow::OnPaint` for the actual

processing. If `OnPaint` indicates the message requires further processing, the message will then be directed to the default message map in `CMyBaseWindow`.

In addition to the default message map, you can define an alternate message map with ALT_MSG_MAP. Always begin a message map with **BEGIN_MSG_MAP**. You can then declare subsequent alternate message maps. The following example shows the default message map and one alternate message map, each containing one handler function:

```
BEGIN_MSG_MAP(CMyClass)
   MESSAGE_HANDLER(WM_PAINT, OnPaint)
ALT_MSG_MAP(1)
   MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
END_MSG_MAP()
```

The next example shows two alternate message maps. The default message map is empty.

```
BEGIN_MSG_MAP(CMyClass)
ALT_MSG_MAP(1)
   MESSAGE_HANDLER(WM_PAINT, OnPaint)
   MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
ALT_MSG_MAP(2)
   MESSAGE_HANDLER(WM_CREATE, OnCreate)
END_MSG_MAP()
```

The END_MSG_MAP macro marks the end of the message map. Note that there is always exactly one instance of **BEGIN_MSG_MAP** and **END_MSG_MAP**.

For more information about using message maps in ATL, see Message Maps.

See Also

Reference

CMessageMap Class

CDynamicChain Class

Other Resources

Message Map Macros (ATL)

ATL Macros

BEGIN_PROP_MAP

Marks the beginning of the object's property map.

```
BEGIN_PROP_MAP( theClass )
```

Parameters

theClass

[in] Specifies the class containing the property map.

Remarks

The property map stores property descriptions, property DISPIDs, property page CLSIDs, and **IDispatch** IIDs. Classes IPerPropertyBrowsingImpl, IPersistPropertyBagImpl, IPersistStreamInitImpl, and ISpecifyPropertyPagesImpl use the property map to retrieve and set this information.

When you create an object with the ATL Project Wizard, the wizard will create an empty property map by specifying **BEGIN_PROP_MAP** followed by END_PROP_MAP.

BEGIN_PROP_MAP does not save out the extent (that is, the dimensions) of a property map, because an object using a property map may not have a user interface, so it would have no extent. If the object is an ActiveX control with a user interface, it has an extent. In this case, you must specify PROP_DATA_ENTRY in your property map to supply the extent.

Example

```
BEGIN_PROP_MAP( CMyClass )
   PROP_DATA_ENTRY( "Width", m_nWidth, VT_UI4 )
   PROP_DATA_ENTRY( "Height", m_nHeight, VT_UI4 )
   PROP_ENTRY( "Property1", 1, CLSID_MyClassPropPage1 )
   PROP_ENTRY_EX( "Caption", DISPID_CAPTION, CLSID_MyClassPropPage2, IID_IMyDual1 )
   PROP_PAGE( CLSID_CMyClassPropPage3 )
END_PROP_MAP( )
```

See Also

Other Resources

Property Map Macros

ATL Macros

BEGIN_RDX_MAP

Marks the beginning of the Registry Data Exchange map.

```
BEGIN_RDX_MAP
```

Remarks

The following macros are used within the Registry Data Exchange map to read and write entries in the system registry:

Macro	Description
RDX_BINARY	Associates the specified registry entry with a specified member variable of type BYTE.
RDX_DWORD	Associates the specified registry entry with a specified member variable of type DWORD.
RDX_CSTRING_TEXT	Associates the specified registry entry with a specified member variable of type CString.
RDX_TEXT	Associates the specified registry entry with a specified member variable of type TCHAR.

The global function RegistryDataExchange, or the member function of the same name created by the **BEGIN_RDX_MAP** and **END_RDX_MAP** macros, should be used whenever your code needs to exchange data between the system registry and the variables specified in the RDX map.

See Also

Other Resources

Registry Data Exchange Macros

BEGIN_SERVICE_MAP

Marks the beginning of the service map.

```
BEGIN_SERVICE_MAP( theClass )
```

Parameters

theClass

[in] Specifies the class containing the service map.

Remarks

Use the service map to implement service provider functionality on your COM object. First, you must derive your class from IServiceProviderImpl. There are two types of entries:

- SERVICE_ENTRY Indicates support for the specified service ID (SID).
- SERVICE_ENTRY_CHAIN Instructs IServiceProviderImpl::QueryService to chain to another, specified object.

Example

```
BEGIN_SERVICE_MAP( CMyService )
   SERVICE_ENTRY( SID_SBindHost )  // This object supports the SBindHost service
   SERVICE_ENTRY_CHAIN( m_spClientSite ) // Everything else, just ask the container
END_SERVICE_MAP( )
```

See Also

Reference

END_SERVICE_MAP

Other Resources

Service Map Macros

ATL Macros

BEGIN_SINK_MAP

Declares the beginning of the event sink map for the composite control.

```
BEGIN_SINK_MAP( _class )
```

Parameters

_class

[in] Specifies the control.

Example

```
BEGIN_SINK_MAP(CMyObj)
   SINK_ENTRY(IDC_CIRCCTL1, DISPID_CLICK, OnClick_CircCtrl1)
   SINK_ENTRY(IDC_CIRCCTL2, DISPID_CLICK, OnClick_CircCtrl2)
END_SINK_MAP()
```

Remarks

CE ATL implementation of ActiveX event sinks only supports return values of type HRESULT or void from your event handler methods; any other return value is unsupported and its behavior is undefined.

See Also

Reference

SINK_ENTRY

END_SINK_MAP

Concepts

Composite Control Fundamentals

Other Resources

Composite Control Macros

ATL Macros

BEGIN_SNAPINTOOLBARID_MAP

Declares the beginning of the toolbar ID map for the Snap-In object.

```
BEGIN_SNAPINTOOLBARID_MAP( _class )
```

Parameters

_class

[in] Specifies the Snap-In object class.

Example

```
class CMySnapinData : public CSnapInItemImpl<CMySnapinData>
{
...
   BEGIN_SNAPINTOOLBARID_MAP(CMySnapinData)
      // IDR_MYSNAPINTOOLBAR is the resource ID of a toolbar resource.
      SNAPINTOOLBARID_ENTRY(IDR_MYSNAPINTOOLBAR)
   END_SNAPINTOOLBARID_MAP()
...
};
...
```

See Also

Reference

END_SNAPINTOOLBARID_MAP

SNAPINTOOLBARID_ENTRY

Other Resources

Snap-In Object Macros

ATL Macros

CHAIN_MSG_MAP

Defines an entry in a message map.

```
CHAIN_MSG_MAP( theChainClass )
```

Parameters

theChainClass

[in] The name of the base class containing the message map.

Remarks

CHAIN_MSG_MAP directs messages to a base class's default message map (declared with BEGIN_MSG_MAP). To direct messages to a base class's alternate message map (declared with ALT_MSG_MAP), use CHAIN_MSG_MAP_ALT.

Note

Always begin a message map with **BEGIN_MSG_MAP**. You can then declare subsequent alternate message maps with **ALT_MSG_MAP**. The END_MSG_MAP macro marks the end of the message map. Every message map must have exactly one instance of **BEGIN_MSG_MAP** and **END_MSG_MAP**.

For more information about using message maps in ATL, see Message Maps.

Example

```
class CMyClass : public CMyBaseClass, ...
{
public:
  ...

  BEGIN_MSG_MAP(CMyClass)
     MESSAGE_HANDLER(WM_PAINT, OnPaint)
     // chain to default message map in CMyBaseClass
     CHAIN_MSG_MAP(CMyBaseClass)
  ALT_MSG_MAP(1)
     // chain to first alternative message map in CMyBaseClass
     CHAIN_MSG_MAP(CMyBaseClass)
  ALT_MSG_MAP(2)
     MESSAGE_HANDLER(WM_CHAR, OnChar)
     // chain to alternate message map in CMyBaseClass
     CHAIN_MSG_MAP_ALT(CMyBaseClass, 1)
  END_MSG_MAP()

  ...
};
```

This example illustrates the following:

- If a window procedure is using `CMyClass`'s default message map and `OnPaint` does not handle a message, the message is directed to `CMyBaseClass`'s default message map for processing.
- If a window procedure is using the first alternate message map in `CMyClass`, all messages are directed to `CMyBaseClass`'s default message map.
- If a window procedure is using `CMyClass`'s second alternate message map and `OnChar` does not handle a message, the message is directed to the specified alternate message map in `CMyBaseClass`. `CMyBaseClass` must have declared this message map with `ALT_MSG_MAP(1)`.

See Also

Reference

CHAIN_MSG_MAP_MEMBER

CHAIN_MSG_MAP_DYNAMIC
MESSAGE_HANDLER

Other Resources

Message Map Macros (ATL)
ATL Macros

CHAIN_MSG_MAP_ALT

Defines an entry in a message map.

```
CHAIN_MSG_MAP_ALT( theChainClass, msgMapID )
```

Parameters

theChainClass

[in] The name of the base class containing the message map.

msgMapID

[in] The message map identifier.

Remarks

CHAIN_MSG_MAP_ALT directs messages to an alternate message map in a base class. You must have declared this alternate message map with ALT_MSG_MAP(msgMapID). To direct messages to a base class's default message map (declared with BEGIN_MSG_MAP), use **CHAIN_MSG_MAP**. For an example, see CHAIN_MSG_MAP.

Note
Always begin a message map with **BEGIN_MSG_MAP**. You can then declare subsequent alternate message maps with **ALT_MSG_MAP**. The END_MSG_MAP macro marks the end of the message map. Every message map must have exactly one instance of **BEGIN_MSG_MAP** and **END_MSG_MAP**.

For more information about using message maps in ATL, see Message Maps.

See Also

Reference

CHAIN_MSG_MAP_ALT_MEMBER

Other Resources

Message Map Macros (ATL)

ATL Macros

CHAIN_MSG_MAP_ALT_MEMBER

Defines an entry in a message map.

```
CHAIN_MSG_MAP_ALT_MEMBER( theChainMember, msgMapID )
```

Parameters

theChainMember

[in] The name of the data member containing the message map.

msgMapID

[in] The message map identifier.

Remarks

CHAIN_MSG_MAP_ALT_MEMBER directs messages to an alternate message map in a data member. You must have declared this alternate message map with ALT_MSG_MAP(msgMapID). To direct messages to a data member's default message map (declared with BEGIN_MSG_MAP), use **CHAIN_MSG_MAP_MEMBER**. For an example, see CHAIN_MSG_MAP_MEMBER.

Note
Always begin a message map with **BEGIN_MSG_MAP**. You can then declare subsequent alternate message maps with **ALT_MSG_MAP**. The END_MSG_MAP macro marks the end of the message map. Every message map must have exactly one instance of **BEGIN_MSG_MAP** and **END_MSG_MAP**.

For more information about using message maps in ATL, see Message Maps.

See Also

Reference

CHAIN_MSG_MAP_ALT

Other Resources

Message Map Macros (ATL)

ATL Macros

CHAIN_MSG_MAP_DYNAMIC

Defines an entry in a message map.

```
CHAIN_MSG_MAP_DYNAMIC( dynaChainID )
```

Parameters

dynaChainID

[in] The unique identifier for an object's message map.

Remarks

CHAIN_MSG_MAP_DYNAMIC directs messages, at run time, to the default message map in another object. The object and its message map are associated with *dynaChainID*, which you define through CDynamicChain::SetChainEntry. You must derive your class from **CDynamicChain** in order to use **CHAIN_MSG_MAP_DYNAMIC**. For an example, see the CDynamicChain overview.

Note

Always begin a message map with BEGIN_MSG_MAP. You can then declare subsequent alternate message maps with **ALT_MSG_MAP**. The END_MSG_MAP macro marks the end of the message map. Every message map must have exactly one instance of **BEGIN_MSG_MAP** and **END_MSG_MAP**.

For more information about using message maps in ATL, see Message Maps.

See Also

Reference

CHAIN_MSG_MAP

CHAIN_MSG_MAP_MEMBER

Other Resources

Message Map Macros (ATL)

ATL Macros

CHAIN_MSG_MAP_MEMBER

Defines an entry in a message map.

```
CHAIN_MSG_MAP_MEMBER( theChainMember )
```

Parameters

theChainMember

[in] The name of the data member containing the message map.

Remarks

CHAIN_MSG_MAP_MEMBER directs messages to a data member's default message map (declared with BEGIN_MSG_MAP). To direct messages to a data member's alternate message map (declared with ALT_MSG_MAP), use CHAIN_MSG_MAP_ALT_MEMBER.

Note

Always begin a message map with **BEGIN_MSG_MAP**. You can then declare subsequent alternate message maps with **ALT_MSG_MAP**. The END_MSG_MAP macro marks the end of the message map. Every message map must have exactly one instance of **BEGIN_MSG_MAP** and **END_MSG_MAP**.

For more information about using message maps in ATL, see Message Maps.

Example

```
class CMyClass : ...
{
public:
   CMyContainedClass m_obj;
   ...

   BEGIN_MSG_MAP(CMyClass)
      MESSAGE_HANDLER(WM_PAINT, OnPaint)
      // chain to default message map of m_obj
      CHAIN_MSG_MAP_MEMBER(m_obj)
   ALT_MSG_MAP(1)
      // chain to default message map of m_obj
      CHAIN_MSG_MAP(m_obj)
   ALT_MSG_MAP(2)
      MESSAGE_HANDLER(WM_CHAR, OnChar)
      // chain to alternate message map of m_obj
      CHAIN_MSG_MAP_ALT(m_obj, 1)
   END_MSG_MAP()

   ...
};
```

This example illustrates the following:

- If a window procedure is using `CMyClass`'s default message map and `OnPaint` does not handle a message, the message is directed to `m_obj`'s default message map for processing.
- If a window procedure is using the first alternate message map in `CMyClass`, all messages are directed to `m_obj`'s default message map.
- If a window procedure is using `CMyClass`'s second alternate message map and `OnChar` does not handle a message, the message is directed to the specified alternate message map of `m_obj`. Class `CMyContainedClass` must have declared this message map with `ALT_MSG_MAP(1)`.

See Also

Reference

CHAIN_MSG_MAP

CHAIN_MSG_MAP_DYNAMIC

MESSAGE_HANDLER

Other Resources

Message Map Macros (ATL)

ATL Macros

COM_INTERFACE_ENTRY (ATL)

Enters interfaces into the COM interface map.

```
COM_INTERFACE_ENTRY( x )
```

Parameters

x

[in] The name of an interface your class object derives from directly.

Remarks

Typically, this is the entry type you use most often.

See COM_INTERFACE_ENTRY Macros for remarks about COM map entries.

Example

```
BEGIN_COM_MAP(CThisExample)
   COM_INTERFACE_ENTRY(IDispatch)
   COM_INTERFACE_ENTRY(IBaseThisExample)
   COM_INTERFACE_ENTRY(ISupportErrorInfo)
END_COM_MAP()
```

See Also

Other Resources

COM Map Macros

ATL Macros

COM_INTERFACE_ENTRY Macros

These macros enter an object's interfaces into its COM map so that they can be accessed by **QueryInterface**. The order of entries in the COM map is the order interfaces will be checked for a matching **IID** during **QueryInterface**.

Each object that wants to expose its interfaces via **QueryInterface** must have its own COM map. The COM map starts with the macro BEGIN_COM_MAP. Interface entries are added with one or more of the **COM_INTERFACE_ENTRY** macros, and the map is completed with the END_COM_MAP macro. For example:

```
BEGIN_COM_MAP(CMyObject)
    COM_INTERFACE_ENTRY(IDispatch)
    COM_INTERFACE_ENTRY(IMyObject)
END_COM_MAP( )
```

Note that the first entry in the COM map must be an interface on the object containing the COM map. Thus, you cannot start your COM map entries with **COM_INTERFACE_ENTRY_CHAIN**, which causes the COM map of a different object to be searched at the point where **COM_INTERFACE_ENTRY_CHAIN(***COtherObject***)** appears in your object's COM map. If you want to search the COM map of another object first, add an interface entry for **IUnknown** to your COM map, then chain the other object's COM map. For example:

```
BEGIN_COM_MAP(CThisObject)
   COM_INTERFACE_ENTRY(IUnknown)
   COM_INTERFACE_ENTRY_CHAIN(COtherObject)
END_COM_MAP( )
```

Caution

As of version 3.0, ATL uses the compiler keyword **__uuidof(** *class* **)** to obtain the corresponding IID for a given class. Because of changes in the **COM_INTERFACE_ENTRY** macros effective in version 3.0, now you simply include the header for the interf ace to use, instead of also linking to a library that defines the matching IIDs for that interface. This change can cause problem s if the header was previously generated by an older version of MIDL, or if it was hand-coded and not marked appropriately. If the declaration for the interface in the header has not been marked with an associated **__declspec(** *uuid* **)**, then any attemp t to use the **__uuidof** keyword for that interface will fail. You can revert to the older (ATL 2.x) **COM_INTERFACE_ENTRY** mac ros by defining **_ATL_NO_UUIDOF** in your build settings to work around any problems with this new behavior.

COM Map Entry Macros

The following are the available entry macros:

- COM_INTERFACE_ENTRY
- COM_INTERFACE_ENTRY_IID
- COM_INTERFACE_ENTRY2
- COM_INTERFACE_ENTRY2_IID
- COM_INTERFACE_ENTRY_FUNC
- COM_INTERFACE_ENTRY_FUNC_BLIND
- COM_INTERFACE_ENTRY_TEAR_OFF
- COM_INTERFACE_ENTRY_CACHED_TEAR_OFF
- COM_INTERFACE_ENTRY_AGGREGATE
- COM_INTERFACE_ENTRY_AGGREGATE_BLIND
- COM_INTERFACE_ENTRY_AUTOAGGREGATE
- COM_INTERFACE_ENTRY_AUTOAGGREGATE_BLIND
- COM_INTERFACE_ENTRY_CHAIN

- COM_INTERFACE_ENTRY_BREAK
- COM_INTERFACE_ENTRY_NOINTERFACE

See Also

Other Resources

COM Map Macros

ATL Macros

COM_INTERFACE_ENTRY_AGGREGATE

When the interface identified by *iid* is queried for, **COM_INTERFACE_ENTRY_AGGREGATE** forwards to *punk*.

```
COM_INTERFACE_ENTRY_AGGREGATE( iid, punk )
```

Parameters

iid

[in] The GUID of the interface queried for.

punk

[in] The name of an **IUnknown** pointer.

Remarks

The *punk* parameter is assumed to point to the inner unknown of an aggregate or to **NULL**, in which case the entry is ignored. Typically, you would **CoCreate** the aggregate in **FinalConstruct**.

See COM_INTERFACE_ENTRY Macros for remarks about COM map entries.

Example

```
BEGIN_COM_MAP(COuter)
   ...
   COM_INTERFACE_ENTRY_AGGREGATE(IID_IAgg, m_pUnkAgg.p)
   ...
END_COM_MAP()
```

See Also

Other Resources

COM Map Macros

ATL Macros

COM_INTERFACE_ENTRY_AGGREGATE_BLIND

Same as COM_INTERFACE_ENTRY_AGGREGATE, except that querying for any IID results in forwarding the query to *punk*.

```
COM_INTERFACE_ENTRY_AGGREGATE_BLIND( punk )
```

Parameters

punk

[in] The name of an **IUnknown** pointer.

Remarks

If the interface query fails, processing of the COM map continues.

See COM_INTERFACE_ENTRY Macros for remarks about COM map entries.

Example

```
BEGIN_COM_MAP(COuter)
   ...
   COM_INTERFACE_ENTRY_AGGREGATE_BLIND(m_pUnkAggBlind.p)
   ...
END_COM_MAP()
```

See Also

Other Resources

COM Map Macros

ATL Macros

COM_INTERFACE_ENTRY_AUTOAGGREGATE

Same as COM_INTERFACE_ENTRY_AGGREGATE, except if *punk* is **NULL**, it automatically creates the aggregate described by the *clsid*.

```
COM_INTERFACE_ENTRY_AUTOAGGREGATE( iid, punk, clsid )
```

Parameters

iid

[in] The GUID of the interface queried for.

punk

[in] The name of an **IUnknown** pointer. Must be a member of the class containing the COM map.

clsid

[in] The identifier of the aggregate that will be created if *punk* is **NULL**.

Remarks

See COM_INTERFACE_ENTRY Macros for remarks about COM map entries.

Example

```
BEGIN_COM_MAP(COuter)
...
   COM_INTERFACE_ENTRY_AUTOAGGREGATE(IID_IAutoAgg, m_pUnkAutoAgg.p, CLSID_CAutoAgg)
...
END_COM_MAP()
```

See Also

Other Resources

COM Map Macros

ATL Macros

COM_INTERFACE_ENTRY_AUTOAGGREGATE_BLIND

Same as COM_INTERFACE_ENTRY_AUTOAGGREGATE, except that querying for any IID results in forwarding the query to *punk*, and if *punk* is **NULL**, automatically creating the aggregate described by the *clsid*.

```
COM_INTERFACE_ENTRY_AUTOAGGREGATE_BLIND( punk, clsid )
```

Parameters

punk

[in] The name of an **IUnknown** pointer. Must be a member of the class containing the COM map.

clsid

[in] The identifier of the aggregate that will be created if *punk* is **NULL**.

Remarks

If the interface query fails, processing of the COM map continues.

See COM_INTERFACE_ENTRY Macros for remarks about COM map entries.

Example

```
BEGIN_COM_MAP(COuter)
   ...
   COM_INTERFACE_ENTRY_AUTOAGGREGATE_BLIND(m_pUnkAutoAggB.p, CLSID_CAutoAggB)
   ...
END_COM_MAP()
```

See Also

Other Resources

COM Map Macros

ATL Macros

COM_INTERFACE_ENTRY_BREAK

Causes your program to call DebugBreak when the specified interface is queried for.

```
COM_INTERFACE_ENTRY_BREAK( x )
```

Parameters

x

[in] Text used to construct the interface identifier.

Remarks

The interface IID will be constructed by appending *x* to `IID_`. For example, if *x* is `IPersistStorage`, the IID will be `IID_IPersistStorage`.

See COM_INTERFACE_ENTRY Macros for remarks about COM map entries.

See Also

Other Resources

COM Map Macros

ATL Macros

COM_INTERFACE_ENTRY_CACHED_TEAR_OFF

Saves the interface-specific data for every instance.

```
COM_INTERFACE_ENTRY_CACHED_TEAR_OFF( iid, x, punk )
```

Parameters

iid

[in] The GUID of the tear-off interface.

x

[in] The name of the class implementing the interface.

punk

[in] The name of an **IUnknown** pointer. Must be a member of the class containing the COM map. Should be initialized to **NULL** in the class object's constructor.

Remarks

If the interface is not used, this lowers the overall instance size of your object.

See COM_INTERFACE_ENTRY Macros for remarks about COM map entries.

Example

```
BEGIN_COM_MAP(COuter)
   ...
   COM_INTERFACE_ENTRY_CACHED_TEAR_OFF(IID_ITearOff2,
      CTearOff2, m_pUnkTearOff2.p)
   ...
END_COM_MAP()
```

See Also

Other Resources

COM Map Macros

ATL Macros

COM_INTERFACE_ENTRY_CHAIN

Processes the COM map of the base class when the processing reaches this entry in the COM map.

```
COM_INTERFACE_ENTRY_CHAIN( classname )
```

Parameters

classname

[in] A base class of the current object.

Remarks

For example, in the following code:

```
BEGIN_COM_MAP(COuter)
...COM_INTERFACE_ENTRY2(IDispatch, IOuter)
   ...
   COM_INTERFACE_ENTRY_CHAIN(CChainBase)
   ...
END_COM_MAP()
```

Note that the first entry in the COM map must be an interface on the object containing the COM map. Thus, you cannot start your COM map entries with **COM_INTERFACE_ENTRY_CHAIN**, which causes the COM map of a different object to be searched at the point where **COM_INTERFACE_ENTRY_CHAIN(***COtherObject***)** appears in your object's COM map. If you want to search the COM map of another object first, add an interface entry for **IUnknown** to your COM map, then chain the other object's COM map. For example:

```
BEGIN_COM_MAP(CThisObject)
   COM_INTERFACE_ENTRY(IUnknown)
   COM_INTERFACE_ENTRY_CHAIN(COtherObject)
END_COM_MAP( )
```

See COM_INTERFACE_ENTRY Macros for remarks about COM map entries.

See Also

Other Resources

COM Map Macros

ATL Macros

COM_INTERFACE_ENTRY_FUNC

A general mechanism for hooking into ATL's **QueryInterface** logic.

```
COM_INTERFACE_ENTRY_FUNC( iid, dw, func )
```

Parameters

iid

[in] The GUID of the interface exposed.

dw

[in] A parameter passed through to the *func*.

func

[in] The function pointer that will return *iid*.

Remarks

If *iid* matches the IID of the interface queried for, then the function specified by *func* is called. The declaration for the function should be:

```
HRESULT WINAPI func(void* pv, REFIID riid, LPVOID* ppv, DWORD dw);
```

When your function is called, *pv* points to your class object. The *riid* parameter refers to the interface being queried for, *ppv* is the pointer to the location where the function should store the pointer to the interface, and *dw* is the parameter you specified in the entry. The function should set **ppv* to **NULL** and return **E_NOINTERFACE** or **S_FALSE** if it chooses not to return an interface. With **E_NOINTERFACE**, COM map processing terminates. With **S_FALSE**, COM map processing continues, even though no interface pointer was returned. If the function returns an interface pointer, it should return **S_OK**.

See COM_INTERFACE_ENTRY Macros for remarks about COM map entries.

See Also

Other Resources

COM Map Macros

ATL Macros

COM_INTERFACE_ENTRY_FUNC_BLIND

Same as COM_INTERFACE_ENTRY_FUNC, except that querying for any IID results in a call to *func*.

```
COM_INTERFACE_ENTRY_FUNC_BLIND( dw, func )
```

Parameters

dw

[in] A parameter passed through to the *func*.

func

[in] The function that gets called when this entry in the COM map is processed.

Remarks

Any failure will cause processing to continue on the COM map. If the function returns an interface pointer, it should return **S_OK**.

See COM_INTERFACE_ENTRY Macros for remarks about COM map entries.

See Also

Other Resources

COM Map Macros

ATL Macros

COM_INTERFACE_ENTRY_IID

Use this macro to enter the interface into the COM map and specify its IID.

```
COM_INTERFACE_ENTRY_IID( iid, x )
```

Parameters

iid

[in] The GUID of the interface exposed.

x

[in] The name of the class whose vtable will be exposed as the interface identified by *iid*.

Remarks

See COM_INTERFACE_ENTRY Macros for remarks about COM map entries.

Example

```
BEGIN_COM_MAP(CThisExample)
   COM_INTERFACE_ENTRY(CThisExample)
   COM_INTERFACE_ENTRY_IID(IID_IDispatch, CThisExampleDispatch)
   COM_INTERFACE_ENTRY(IBaseThisExample)
   COM_INTERFACE_ENTRY(ISupportErrorInfo)
END_COM_MAP()
```

See Also

Other Resources

COM Map Macros

ATL Macros

COM_INTERFACE_ENTRY_NOINTERFACE

Returns **E_NOINTERFACE** and terminates COM map processing when the specified interface is queried for.

```
COM_INTERFACE_ENTRY_NOINTERFACE( x )
```

Parameters

x

[in] Text used to construct the interface identifier.

Remarks

You can use this macro to prevent an interface from being used in a particular case. For example, you can insert this macro into your COM map right before **COM_INTERFACE_ENTRY_AGGREGATE_BLIND** to prevent a query for the interface from being forwarded to the aggregate's inner unknown.

The interface IID will be constructed by appending *x* to `IID_`. For example, if *x* is `IPersistStorage`, the IID will be `IID_IPersistStorage`.

See COM_INTERFACE_ENTRY Macros for remarks about COM map entries.

See Also

Other Resources

COM Map Macros

ATL Macros

COM_INTERFACE_ENTRY_TEAR_OFF

Exposes your tear-off interfaces.

```
COM_INTERFACE_ENTRY_TEAR_OFF( iid, x )
```

Parameters

iid

[in] The GUID of the tear-off interface.

x

[in] The name of the class implementing the interface.

Remarks

A tear-off interface is implemented as a separate object that is instantiated every time the interface it represents is queried for. Typically, you build your interface as a tear-off if the interface is rarely used, since this saves a vtable pointer in every instance of your main object. The tear-off is deleted when its reference count becomes zero. The class implementing the tear-off should be derived from **CComTearOffObjectBase** and have its own COM map.

See COM_INTERFACE_ENTRY Macros for remarks about COM map entries.

Example

```
BEGIN_COM_MAP(COuter)
   ...
   COM_INTERFACE_ENTRY_TEAR_OFF(IID_ITearOff1, CTearOff1)
   ...
END_COM_MAP()
```

See Also

Other Resources

COM Map Macros

ATL Macros

COM_INTERFACE_ENTRY2

Use this macro to disambiguate two branches of inheritance.

```
COM_INTERFACE_ENTRY2( x, x2 )
```

Parameters

x

[in] The name of an interface you want to expose from your object.

x2

[in] The name of the inheritance branch from which *x* is exposed.

Remarks

For example, if you derive your class object from two dual interfaces, you expose **IDispatch** using **COM_INTERFACE_ENTRY2** since **IDispatch** can be obtained from either one of the interfaces. For more information, see Implementing the IDispatch Interface.

See COM_INTERFACE_ENTRY Macros for remarks about COM map entries.

Example

```
class COuter :
   public CChainBase, // CChainBase derives from
                      // IDispatch
   public IDispatchImpl<IOuter, &IID_IOuter,
                        &LIBID_COMMAPLib>,
   public CComCoClass<COuter, &CLSID_COuter>
{
public:
   COuter(){}
   ...

   BEGIN_COM_MAP(COuter)
      COM_INTERFACE_ENTRY2(IDispatch, IOuter)
   ...
   END_COM_MAP

};
```

See Also

Other Resources

COM Map Macros

ATL Macros

COM_INTERFACE_ENTRY2_IID

Same as COM_INTERFACE_ENTRY2, except you can specify a different IID.

```
COM_INTERFACE_ENTRY2_IID( iid, x, x2 )
```

Parameters

iid

[in] The GUID you are specifying for the interface.

x

[in] The name of an interface that your class object derives from directly.

x2

[in] The name of a second interface that your class object derives from directly.

Remarks

See COM_INTERFACE_ENTRY Macros for remarks about COM map entries.

See Also

Other Resources

COM Map Macros

ATL Macros

COMMAND_CODE_HANDLER

Similar to COMMAND_HANDLER, but maps a WM_COMMAND message based only on the notification code.

```
COMMAND_CODE_HANDLER( code, func )
```

Parameters

code

[in] The notification code.

func

[in] The name of the message-handler function.

See Also

Reference

COMMAND_ID_HANDLER
COMMAND_RANGE_HANDLER
MESSAGE_HANDLER
NOTIFY_CODE_HANDLER

Other Resources

Message Map Macros (ATL)
ATL Macros

COMMAND_HANDLER

Defines an entry in a message map.

```
COMMAND_HANDLER( id, code, func )
```

Parameters

id

[in] The identifier of the menu item, control, or accelerator.

code

[in] The notification code.

func

[in] The name of the message-handler function.

Remarks

COMMAND_HANDLER maps a WM_COMMAND message to the specified handler function, based on the notification code and the control identifier. For example:

```
class CMyClass : ...
{
public:
   ...

   BEGIN_MSG_MAP(CMyClass)
      COMMAND_HANDLER(IDC_MYCTL, EN_CHANGE, OnChange)
      ...
   END_MSG_MAP()

   // When a CMyClass object receives a WM_COMMAND
   // message identified by IDC_MYCTL and EN_CHANGE,
   // the message is directed to CMyClass::OnChange
   // for the actual processing.
   LRESULT OnChange( ... )
   { ... }

};
```

Any function specified in a **COMMAND_HANDLER** macro must be defined as follows:

```
LRESULT CommandHandler(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled);
```

The message map sets `bHandled` to **TRUE** before `CommandHandler` is called. If `CommandHandler` does not fully handle the message, it should set `bHandled` to **FALSE** to indicate the message needs further processing.

Note

Always begin a message map with BEGIN_MSG_MAP. You can then declare subsequent alternate message maps with ALT_MSG_MAP. The END_MSG_MAP macro marks the end of the message map. Every message map must have exactly one i nstance of **BEGIN_MSG_MAP** and **END_MSG_MAP**.

In addition to **COMMAND_HANDLER**, you can use MESSAGE_HANDLER to map a **WM_COMMAND** message without regard to an identifier or code. In this case, `MESSAGE_HANDLER(WM_COMMAND, OnHandlerFunction)` will direct all **WM_COMMAND** messages to `OnHandlerFunction`.

For more information about using message maps in ATL, see Message Maps.

See Also

Reference

COMMAND_ID_HANDLER
COMMAND_CODE_HANDLER
COMMAND_RANGE_HANDLER
NOTIFY_HANDLER

Other Resources

Message Map Macros (ATL)
ATL Macros

COMMAND_ID_HANDLER

Similar to COMMAND_HANDLER, but maps a WM_COMMAND message based only on the identifier of the menu item, control, or accelerator.

```
COMMAND_ID_HANDLER( id, func )
```

Parameters

id

[in] The identifier of the menu item, control, or accelerator sending the message.

func

[in] The name of the message-handler function.

See Also

Reference

COMMAND_CODE_HANDLER
COMMAND_RANGE_HANDLER
MESSAGE_HANDLER
NOTIFY_ID_HANDLER

Other Resources

Message Map Macros (ATL)
ATL Macros

COMMAND_RANGE_CODE_HANDLER

Similar to COMMAND_RANGE_HANDLER, but maps WM_COMMAND messages with a specific notification code from a range of controls to a single handler function.

```
COMMAND_RANGE_CODE_HANDLER( idFirst, idLast, code, func )
```

Parameters

idFirst

[in] Marks the beginning of a contiguous range of control identifiers.

idLast

[in] Marks the end of a contiguous range of control identifiers.

code

[in] The notification code.

func

[in] The name of the message-handler function.

Remarks

This range is based on the identifier of the menu item, control, or accelerator sending the message.

See Also

Reference

COMMAND_ID_HANDLER
COMMAND_CODE_HANDLER
MESSAGE_RANGE_HANDLER
NOTIFY_RANGE_HANDLER

Other Resources

Message Map Macros (ATL)
ATL Macros

COMMAND_RANGE_HANDLER

Similar to COMMAND_HANDLER, but maps WM_COMMAND messages from a range of controls to a single handler function.

```
COMMAND_RANGE_HANDLER( idFirst, idLast, func )
```

Parameters

idFirst

[in] Marks the beginning of a contiguous range of control identifiers.

idLast

[in] Marks the end of a contiguous range of control identifiers.

func

[in] The name of the message-handler function.

Remarks

This range is based on the identifier of the menu item, control, or accelerator sending the message.

See Also

Reference

COMMAND_ID_HANDLER
COMMAND_CODE_HANDLER
MESSAGE_RANGE_HANDLER
NOTIFY_RANGE_HANDLER

Other Resources

Message Map Macros (ATL)
ATL Macros

CONNECTION_POINT_ENTRY

Enters a connection point for the specified interface into the connection point map so that it can be accessed.

```
CONNECTION_POINT_ENTRY( iid )
```

Parameters

iid

[in] The GUID of the interface being added to the connection point map.

Remarks

Connection point entries in the map are used by IConnectionPointContainerImpl. The class containing the connection point map must inherit from **IConnectionPointContainerImpl**.

Start your connection point map with the BEGIN_CONNECTION_POINT_MAP macro, add entries for each of your connection points with the **CONNECTION_POINT_ENTRY** macro, and complete the map with the END_CONNECTION_POINT_MAP macro.

For more information about connection points in ATL, see the article Connection Points.

Example

```
class CMyCPClass :
   public IConnectionPointContainerImpl<CMyCPClass>,
   public IPropertyNotifySinkCP<CMyCPClass>
{
public:
   ...
   BEGIN_CONNECTION_POINT_MAP(CMyCPClass)
      CONNECTION_POINT_ENTRY(IID_IPropertyNotifySink)
   END_CONNECTION_POINT_MAP( )
   ...
};
```

See Also

Other Resources

Connection Point Macros

ATL Macros

DECLARE_AGGREGATABLE

Specifies that your object can be aggregated.

```
DECLARE_AGGREGATABLE(
      x
)
```

Parameters

x

[in] The name of the class you are defining as aggregatable.

Remarks

CComCoClass contains this macro to specify the default aggregation model. To override this default, specify either the DECLARE_NOT_AGGREGATABLE or DECLARE_ONLY_AGGREGATABLE macro in your class definition.

Example

```
class CMyClass : public CComCoClass< .. >, ...
{
public:
   DECLARE_NOT_AGGREGATABLE(CMyClass)
   ...
};
```

See Also

Other Resources

Aggregation and Class Factory Macros

ATL Macros

DECLARE_CLASSFACTORY

Declares CComClassFactory to be the class factory.

```
DECLARE_CLASSFACTORY( )
```

Remarks

CComCoClass uses this macro to declare the default class factory for your object.

Example

```
class CMyClass : ..., public CComCoClass< ... >
{
   ...
   DECLARE_CLASSFACTORY()
   ...
};
```

See Also

Reference

DECLARE_CLASSFACTORY_EX
DECLARE_CLASSFACTORY2
DECLARE_CLASSFACTORY_AUTO_THREAD
DECLARE_CLASSFACTORY_SINGLETON

Other Resources

Aggregation and Class Factory Macros
ATL Macros

DECLARE_CLASSFACTORY_AUTO_THREAD

Declares CComClassFactoryAutoThread to be the class factory.

```
DECLARE_CLASSFACTORY_AUTO_THREAD( )
```

Remarks

CComCoClass includes the DECLARE_CLASSFACTORY macro, which specifies CComClassFactory as the default class factory. However, by including the **DECLARE_CLASSFACTORY_AUTO_THREAD** macro in your object's class definition, you override this default.

When you create objects in multiple apartments (in an out-of-proc server), add **DECLARE_CLASSFACTORY_AUTO_THREAD** to your class.

Example

```
class CMyClass : ..., public CComCoClass< ... >
{
   ...
   DECLARE_CLASSFACTORY_AUTO_THREAD( )
   ...
};
```

See Also

Reference

DECLARE_CLASSFACTORY_EX
DECLARE_CLASSFACTORY2
DECLARE_CLASSFACTORY_SINGLETON

Other Resources

Aggregation and Class Factory Macros
ATL Macros

DECLARE_CLASSFACTORY_EX

Declares *cf* to be the class factory.

```
DECLARE_CLASSFACTORY_EX(
      cf
)
```

Parameters

cf

[in] The name of the class that implements your class factory object.

Remarks

The *cf* parameter must derive from CComClassFactory and override the **CreateInstance** method.

CComCoClass includes the DECLARE_CLASSFACTORY macro, which specifies **CComClassFactory** as the default class factory. However, by including the **DECLARE_CLASSFACTORY_EX** macro in your object's class definition, you override this default.

Example

```
class CMyClass : ..., public CComCoClass< ... >
{
   ...
   DECLARE_CLASSFACTORY_EX(CMyClassFactory)
   ...
};
```

See Also

Reference

DECLARE_CLASSFACTORY2
DECLARE_CLASSFACTORY_AUTO_THREAD
DECLARE_CLASSFACTORY_SINGLETON

Other Resources

Aggregation and Class Factory Macros
ATL Macros

DECLARE_CLASSFACTORY_SINGLETON

Declares CComClassFactorySingleton to be the class factory.

```
DECLARE_CLASSFACTORY_SINGLETON(
      obj
)
```

Parameters

obj

[in] The name of your class object.

Remarks

CComCoClass includes the DECLARE_CLASSFACTORY macro, which specifies CComClassFactory as the default class factory. However, by including the **DECLARE_CLASSFACTORY_SINGLETON** macro in your object's class definition, you override this default.

Example

```
class CMyClass : ..., public CComCoClass< ... >
{
   ...
   DECLARE_CLASSFACTORY_SINGLETON(CMyClass)
   ...
};
```

See Also

Reference

DECLARE_CLASSFACTORY2
DECLARE_CLASSFACTORY_EX
DECLARE_CLASSFACTORY_AUTO_THREAD

Other Resources

Aggregation and Class Factory Macros
ATL Macros

DECLARE_CLASSFACTORY2

Declares CComClassFactory2 to be the class factory.

```
DECLARE_CLASSFACTORY2(
      lic
)
```

Parameters

lic

[in] A class that implements **VerifyLicenseKey**, **GetLicenseKey**, and **IsLicenseValid**.

Remarks

CComCoClass includes the DECLARE_CLASSFACTORY macro, which specifies CComClassFactory as the default class factory. However, by including the **DECLARE_CLASSFACTORY2** macro in your object's class definition, you override this default.

Example

```
class CMyClass : ..., public CComCoClass< ... >
{
   ...
   DECLARE_CLASSFACTORY2(CMyLicense)
   ...
};
```

See Also

Reference

DECLARE_CLASSFACTORY_EX
DECLARE_CLASSFACTORY_AUTO_THREAD
DECLARE_CLASSFACTORY_SINGLETON

Other Resources

Aggregation and Class Factory Macros
ATL Macros

DECLARE_EMPTY_MSG_MAP

Declares an empty message map.

```
DECLARE_EMPTY_MSG_MAP( )
```

Remarks

DECLARE_EMPTY_MSG_MAP is a convenience macro that calls the macros BEGIN_MSG_MAP and END_MSG_MAP to create an empty message map:

```
BEGIN_MSG_MAP(...)
END_MSG_MAP
```

See Also

Other Resources

Message Map Macros (ATL)

ATL Macros

DECLARE_GET_CONTROLLING_UNKNOWN

Declares a virtual function **GetControllingUnknown**.

```
DECLARE_GET_CONTROLLING_UNKNOWN( )
```

Remarks

Add this macro to your object if you get the compiler error message that **GetControllingUnknown** is undefined (for example, in **CComAggregateCreator**).

See Also

Reference

DECLARE_AGGREGATABLE

Other Resources

Aggregation and Class Factory Macros

ATL Macros

DECLARE_LIBID

Provides a way for ATL to obtain the *libid* of the type library.

```
DECLARE_LIBID(
      libid
)
```

Parameters

libid

The GUID of the type library.

Remarks

Use **DECLARE_LIBID** in a **CAtlModuleT**-derived class.

Example

Non-attributed wizard-generated ATL projects will have a sample of using this macro.

See Also

Reference

CAtlModuleT Class

Other Resources

Registry Macros

ATL Macros

DECLARE_NO_REGISTRY

Use **DECLARE_NO_REGISTRY** if you want to avoid any default ATL registration for the class in which this macro appears.

```
DECLARE_NO_REGISTRY( )
```

See Also

Reference

DECLARE_REGISTRY
DECLARE_REGISTRY_RESOURCE
DECLARE_REGISTRY_RESOURCEID

Other Resources

Registry Macros
ATL Macros

DECLARE_NOT_AGGREGATABLE

Specifies that your object cannot be aggregated.

```
DECLARE_NOT_AGGREGATABLE(
      x
)
```

Parameters

x

[in] The name of the class object you are defining as not aggregatable.

Remarks

DECLARE_NOT_AGGREGATABLE causes **CreateInstance** to return an error (**CLASS_E_NOAGGREGATION**) if an attempt is made to aggregate onto your object.

By default, CComCoClass contains the DECLARE_AGGREGATABLE macro, which specifies that your object can be aggregated. To override this default behavior, include **DECLARE_NOT_AGGREGATABLE** in your class definition.

Example

```
class CMyClass : public CComCoClass< .. >, ...
{
public:
   DECLARE_NOT_AGGREGATABLE(CMyClass)
   ...
};
```

See Also

Reference

DECLARE_ONLY_AGGREGATABLE

Other Resources

Aggregation and Class Factory Macros

ATL Macros

DECLARE_OBJECT_DESCRIPTION

Allows you to specify a text description for your class object.

```
DECLARE_OBJECT_DESCRIPTION(
      x
)
```

Parameters

x

[in] The class object's description.

Remarks

ATL enters this description into the object map through the OBJECT_ENTRY macro.

DECLARE_OBJECT_DESCRIPTION implements a **GetObjectDescription** function, which you can use to override the CComCoClass::GetObjectDescription method.

The **GetObjectDescription** function is called by **IComponentRegistrar::GetComponents**. **IComponentRegistrar** is an Automation interface that allows you to register and unregister individual components in a DLL. When you create a Component Registrar object with the ATL Project Wizard, the wizard will automatically implement the **IComponentRegistrar** interface. **IComponentRegistrar** is typically used by Microsoft Transaction Server.

For more information about the ATL Project Wizard, see the article Creating an ATL Project.

Example

```
class CMyClass : public CComCoClass< ... >, ...
{
public:
   // Override CComCoClass::GetObjectDescription
   DECLARE_OBJECT_DESCRIPTION("Account Transfer Object 1.0")

   ...
};
```

See Also

Other Resources

Object Map Macros

ATL Macros

DECLARE_OLEMISC_STATUS

Used in ATL ActiveX controls to set the OLEMISC flags.

```
DECLARE_OLEMISC_STATUS(
      miscstatus
)
```

Parameters

miscstatus

All applicable OLEMISC flags.

Remarks

This macro is used to set the OLEMISC flags for an ActiveX control. Refer to IOleObject::GetMiscStatus for more details.

Example

```
DECLARE_OLEMISC_STATUS(OLEMISC_ACTSLIKEBUTTON | OLEMISC_ACTIVATEWHENVISIBLE)
```

See Also

Reference

OLEMISC

Other Resources

Object Status Macros

DECLARE_ONLY_AGGREGATABLE

Specifies that your object must be aggregated.

```
DECLARE_ONLY_AGGREGATABLE(
      x
)
```

Parameters

x

[in] The name of the class object you are defining as only aggregatable.

Remarks

DECLARE_ONLY_AGGREGATABLE causes an error (**E_FAIL**) if an attempt is made to **CoCreate** your object as nonaggregated object.

By default, CComCoClass contains the DECLARE_AGGREGATABLE macro, which specifies that your object can be aggregated. To override this default behavior, include **DECLARE_ONLY_AGGREGATABLE** in your class definition.

Example

```
class CMyClass : public CComCoClass< .. >, ...
{
public:
   DECLARE_ONLY_AGGREGATABLE(CMyClass)
   ...
};
```

See Also

Reference

DECLARE_NOT_AGGREGATABLE

Other Resources

Aggregation and Class Factory Macros

ATL Macros

DECLARE_POLY_AGGREGATABLE

Specifies that an instance of **CComPolyObject** < *x* > is created when your object is created.

```
DECLARE_POLY_AGGREGATABLE(
      x
)
```

Parameters

x

[in] The name of the class object you are defining as aggregatable or not aggregatable.

Remarks

During creation, the value of the outer unknown is checked. If it is **NULL**, **IUnknown** is implemented for a nonaggregated object. If the outer unknown is not **NULL**, **IUnknown** is implemented for an aggregated object.

The advantage of using **DECLARE_POLY_AGGREGATABLE** is that you avoid having both **CComAggObject** and **CComObject** in your module to handle the aggregated and nonaggregated cases. A single **CComPolyObject** object handles both cases. This means only one copy of the vtable and one copy of the functions exist in your module. If your vtable is large, this can substantially decrease your module size. However, if your vtable is small, using **CComPolyObject** can result in a slightly larger module size because it is not optimized for an aggregated or nonaggregated object, as are **CComAggObject** and **CComObject**.

The **DECLARE_POLY_AGGREGATABLE** macro is automatically declared in your object if you use the ATL Control Wizard to create a full control.

See Also

Reference

CComPolyObject Class
CComAggObject Class
CComObject Class

Other Resources

Aggregation and Class Factory Macros
ATL Macros

DECLARE_PROTECT_FINAL_CONSTRUCT

Protects your object from being deleted if (during FinalConstruct) the internal aggregated object increments the reference count then decrements the count to 0.

```
DECLARE_PROTECT_FINAL_CONSTRUCT( )
```

See Also

Other Resources

Aggregation and Class Factory Macros

ATL Macros

DECLARE_REGISTRY

Enters the standard class registration into the system registry or removes it from the system registry.

```
DECLARE_REGISTRY(
      class,
      pid,
      vpid,
      nid,
      flags
)
```

Parameters

class

[in] Included for backward compatibility.

pid

[in] An **LPCTSTR** that is a version-specific program identifier.

vpid

[in] An **LPCTSTR** that is a version-independent program identifier.

nid

[in] A **UINT** that is an index of the resource string in the registry to use as the description of the program.

flags

[in] A **DWORD** containing the program's threading model in the registry. Must be one of the following values: **THREADFLAGS_APARTMENT**, **THREADFLAGS_BOTH**, or **AUTPRXFLAG**.

Remarks

The standard registration consists of the CLSID, program ID, version-independent program ID, description string, and thread model.

When you create an object or control using the ATL Add Class Wizard, the wizard automatically implements script-based registry support and adds the DECLARE_REGISTRY_RESOURCEID macro to your files. If you do not want script-based registry support, you need to replace this macro with **DECLARE_REGISTRY**. **DECLARE_REGISTRY** only inserts the five basic keys described above into the registry. You must manually write code to insert other keys into the registry.

See Also

Reference

DECLARE_REGISTRY_RESOURCE

Other Resources

Registry Macros

ATL Macros

DECLARE_REGISTRY_APPID_RESOURCEID

Specifies the information required to automatically register the *appid*.

```
DECLARE_REGISTRY_APPID_RESOURCEID(
      resid,
      appid
)
```

Parameters

resid

The resource id of the .rgs file that contains information about the *appid*.

appid

A GUID.

Remarks

Use **DECLARE_REGISTRY_APPID_RESOURCEID** in a **CAtlModuleT**-derived class.

Example

Classes added to ATL projects with the Add Class code wizard will have a sample of using this macro.

See Also

Reference

CAtlModuleT Class

Other Resources

Registry Macros

ATL Macros

DECLARE_REGISTRY_RESOURCE

Gets the named resource containing the registry file and runs the script to either enter objects into the system registry or remove them from the system registry.

```
DECLARE_REGISTRY_RESOURCE(
      x
)
```

Parameters

x

[in] String identifier of your resource.

Remarks

When you create an object or control using the ATL Project Wizard, the wizard will automatically implement script-based registry support and add the DECLARE_REGISTRY_RESOURCEID macro, which is similar to **DECLARE_REGISTRY_RESOURCE**, to your files.

You can statically link with the ATL Registry Component (Registrar) for optimized registry access. To statically link to the Registrar code, add the following line to your stdafx.h file:

```
#define _ATL_STATIC_REGISTRY
```

If you want ATL to substitute replacement values at run time, do not specify the **DECLARE_REGISTRY_RESOURCE** or **DECLARE_REGISTRY_RESOURCEID** macro. Instead, create an array of **_ATL_REGMAP_ENTRIES** structures, where each entry contains a variable placeholder paired with a value to replace the placeholder at run time. Then call CAtlModule::UpdateRegistryFromResourceD or CAtlModule::UpdateRegistryFromResourceS, passing the array. This adds all of the replacement values in the **_ATL_REGMAP_ENTRIES** structures to the Registrar's replacement map.

For more information about replaceable parameters and scripting, see the article The ATL Registry Component (Registrar).

See Also

Reference

DECLARE_REGISTRY

_ATL_STATIC_REGISTRY

Other Resources

Registry Macros

ATL Macros

DECLARE_REGISTRY_RESOURCEID

Same as DECLARE_REGISTRY_RESOURCE except that it uses a wizard-generated **UINT** to identify the resource, rather than a string name.

```
DECLARE_REGISTRY_RESOURCEID(
      x
)
```

Parameters

x

[in] Wizard-generated identifier of your resource.

Remarks

When you create an object or control using the ATL Project Wizard, the wizard will automatically implement script-based registry support and add the **DECLARE_REGISTRY_RESOURCEID** macro to your files.

You can statically link with the ATL Registry Component (Registrar) for optimized registry access. To statically link to the Registrar code, add the following line to your stdafx.h file:

```
#define _ATL_STATIC_REGISTRY
```

If you want ATL to substitute replacement values at run time, do not specify the **DECLARE_REGISTRY_RESOURCE** or **DECLARE_REGISTRY_RESOURCEID** macro. Instead, create an array of **_ATL_REGMAP_ENTRIES** structures, where each entry contains a variable placeholder paired with a value to replace the placeholder at run time. Then call CAtlModule::UpdateRegistryFromResourceD or CAtlModule::UpdateRegistryFromResourceS, passing the array. This adds all of the replacement values in the **_ATL_REGMAP_ENTRIES** structures to the Registrar's replacement map.

For more information about replaceable parameters and scripting, see the article The ATL Registry Component (Registrar).

See Also

Reference

DECLARE_REGISTRY
DECLARE_REGISTRY_RESOURCE

Other Resources

Registry Macros
ATL Macros

DECLARE_VIEW_STATUS

Place this macro in an ATL ActiveX control's control class to specify the **VIEWSTATUS** flags to the container.

```
DECLARE_VIEW_STATUS(
      statusFlags
)
```

Parameters

statusFlags

[in] The **VIEWSTATUS** flags. See VIEWSTATUS for a list of flags.

Example

```
DECLARE_VIEW_STATUS(VIEWSTATUS_OPAQUE)
```

See Also

Other Resources

Aggregation and Class Factory Macros

ATL Macros

DECLARE_WND_CLASS

Allows you to specify the name of a new window class. Place this macro in an ATL ActiveX control's control class.

```
DECLARE_WND_CLASS(
      WndClassName
)
```

Parameters

WndClassName

[in] The name of the new window class. If **NULL**, ATL will generate a window class name.

Remarks

This macro allows you to specify the name of a new window class whose information will be managed by CWndClassInfo. **DECLARE_WND_CLASS** defines the new window class by implementing the following static function:

```
static CWndClassInfo& GetWndClassInfo();
```

DECLARE_WND_CLASS specifies the following styles for the new window:

- CS_HREDRAW
- CS_VREDRAW
- CS_DBLCLKS

DECLARE_WND_CLASS also specifies the default window's background color. Use the DECLARE_WND_CLASS_EX macro to provide your own styles and background color.

CWindowImpl uses the **DECLARE_WND_CLASS** macro to create a window based on a new window class. To override this behavior, use the DECLARE_WND_SUPERCLASS macro, or provide your own implementation of the GetWndClassInfo function.

For more information about using windows in ATL, see the article ATL Window Classes.

See Also

Other Resources

Window Class Macros

ATL Macros

DECLARE_WND_CLASS_EX

Allows you to specify the name of an existing window class on which a new window class will be based. Place this macro in an ATL ActiveX control's control class.

```
DECLARE_WND_CLASS_EX(
      WndClassName,
      style,
      bkgnd
)
```

Parameters

WndClassName

[in] The name of the new window class. If **NULL**, ATL will generate a window class name.

style

[in] The style of the window.

bkgnd

[in] The background color of the window.

Remarks

This macro allows you to specify the class parameters of a new window class, whose information will be managed by CWndClassInfo. **DECLARE_WND_CLASS_EX** defines the new window class by implementing the following static function:

```
static CWndClassInfo& GetWndClassInfo();
```

If you want to use the default styles and background color, use the DECLARE_WND_CLASS macro. For more information about using windows in ATL, see the article ATL Window Classes.

See Also

Reference

DECLARE_WND_SUPERCLASS

Other Resources

Window Class Macros

ATL Macros

DECLARE_WND_SUPERCLASS

Allows you to specify the parameters of a class. Place this macro in an ATL ActiveX control's control class.

```
DECLARE_WND_SUPERCLASS(
      WndClassName,
      OrigWndClassName
)
```

Parameters

WndClassName

[in] The name of the window class that will superclass *OrigWndClassName*. If **NULL**, ATL will generate a window class name.

OrigWndClassName

[in] The name of an existing window class.

Remarks

This macro allows you to specify the name of a window class that will superclass an existing window class. CWndClassInfo manages the information of the superclass.

DECLARE_WND_SUPERCLASS implements the following static function:

```
static CWndClassInfo& GetWndClassInfo();
```

By default, CWindowImpl uses the DECLARE_WND_CLASS macro to create a window based on a new window class. By specifying the **DECLARE_WND_SUPERCLASS** macro in a **CWindowImpl**-derived class, the window class will be based on an existing class but will use your window procedure. This technique is called superclassing.

Besides using the **DECLARE_WND_CLASS** and **DECLARE_WND_SUPERCLASS** macros, you can override the GetWndClassInfo function with your own implementation.

For more information about using windows in ATL, see the article ATL Window Classes.

See Also

Other Resources

Window Class Macros

ATL Macros

DEFAULT_REFLECTION_HANDLER

Provides a default handler for the child window (control) that will receive reflected messages; the handler will properly pass unhandled messages to **DefWindowProc**.

```
DEFAULT_REFLECTION_HANDLER( )
```

See Also

Reference

REFLECT_NOTIFICATIONS

BEGIN_MSG_MAP

Other Resources

Message Map Macros (ATL)

ATL Macros

DEVMODE and TEXTMETRIC String Conversion Macros

These macros create a copy of a DEVMODE or TEXTMETRIC structure and convert the strings within the new structure to a new string type. The macros allocate memory on the stack for the new structure and return a pointer to the new structure.

```
MACRONAME(
      address_of_structure
)
```

Remarks

For example:

```
DEVMODEA2W( lpa )
```

and:

```
TEXTMETRICA2W( lptma )
```

In the macro names, the string type in the source structure is on the left (for example, **A**) and the string type in the destination structure is on the right (for example, **W**). **A** stands for **LPSTR**, **OLE** stands for **LPOLESTR**, **T** stands for **LPTSTR**, and **W** stands for **LPWSTR**.

Thus, **DEVMODEA2W** copies a **DEVMODE** structure with **LPSTR** strings into a **DEVMODE** structure with **LPWSTR** strings, **TEXTMETRICOLE2T** copies a **TEXTMETRIC** structure with **LPOLESTR** strings into a **TEXTMETRIC** structure with **LPTSTR** strings, and so on.

The two strings converted in the **DEVMODE** structure are the device name (**dmDeviceName**) and the form name (**dmFormName**). The **DEVMODE** string conversion macros also update the structure size (**dmSize**).

The four strings converted in the **TEXTMETRIC** structure are the first character (**tmFirstChar**), the last character (**tmLastChar**), the default character (**tmDefaultChar**), and the break character (**tmBreakChar**).

The behavior of the **DEVMODE** and **TEXTMETRIC** string conversion macros depends on the compiler directive in effect, if any. If the source and destination types are the same, no conversion takes place. Compiler directives change **T** and **OLE** as follows:

Compiler directive in effect	T becomes	OLE becomes
none	**A**	**W**
_UNICODE	**W**	**W**
OLE2ANSI	**A**	**A**
_UNICODE and **OLE2ANSI**	**W**	**A**

The following table lists the **DEVMODE** and **TEXTMETRIC** string conversion macros.

DEVMODE and TEXTMETRIC String Conversion Macros

DEVMODEA2W	**TEXTMETRICA2W**
DEVMODEOLE2T	**TEXTMETRICOLE2T**
DEVMODET2OLE	**TEXTMETRICT2OLE**
DEVMODEW2A	**TEXTMETRICW2A**

See Also

Concepts

ATL and MFC String Conversion Macros

Other Resources

ATL Macros

END_CATEGORY_MAP

Marks the end of the category map.

```
END_CATEGORY_MAP( )
```

Example

See the example for BEGIN_CATEGORY_MAP.

Smart Device Developer Notes

Not supported for smart device projects.

See Also

Reference

BEGIN_CATEGORY_MAP
IMPLEMENTED_CATEGORY
REQUIRED_CATEGORY

Other Resources

Category Macros
ATL Macros

END_COM_MAP

Ends the definition of your COM interface map.

```
END_COM_MAP( )
```

See Also

Reference

BEGIN_COM_MAP

COM_INTERFACE_ENTRY Macros

Other Resources

COM Map Macros

ATL Macros

END_CONNECTION_POINT_MAP

Marks the end of the connection point map entries.

```
END_CONNECTION_POINT_MAP( )
```

Remarks

Start your connection point map with the BEGIN_CONNECTION_POINT_MAP macro, add entries for each of your connection points with the CONNECTION_POINT_ENTRY macro, and complete the map with the **END_CONNECTION_POINT_MAP** macro.

For more information about connection points in ATL, see the article Connection Points.

Example

```
BEGIN_CONNECTION_POINT_MAP(CMyCPClass)
   CONNECTION_POINT_ENTRY(m_cpInterfaceBeingExposed)
END_CONNECTION_POINT_MAP( )
```

See Also

Other Resources

Connection Point Macros

ATL Macros

END_EXTENSION_SNAPIN_NODEINFO_MAP

Marks the end of the snap-in extension data class map.

```
END_EXTENSION_SNAPIN_NODEINFO_MAP( )
```

Remarks

Start your snap-in extension map with the BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP macro, add entries for each of your extension snap-in data types with the EXTENSION_SNAPIN_NODEINFO_ENTRY macro, and complete the map with the **END_EXTENSION_SNAPIN_NODEINFO_MAP** macro.

Example

See the example for BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP.

See Also

Reference

BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP
EXTENSION_SNAPIN_NODEINFO_ENTRY
EXTENSION_SNAPIN_DATACLASS

Other Resources

Snap-In Object Macros
ATL Macros

END_MSG_MAP

Marks the end of a message map.

```
END_MSG_MAP( )
```

Remarks

Always use the BEGIN_MSG_MAP macro to mark the beginning of a message map. Use ALT_MSG_MAP to declare subsequent alternate message maps.

Note that there is always exactly one instance of **BEGIN_MSG_MAP** and **END_MSG_MAP**.

For more information about using message maps in ATL, see Message Maps.

Example

The following example shows the default message map and one alternate message map, each containing one handler function:

```
BEGIN_MSG_MAP(CMyClass)
   MESSAGE_HANDLER(WM_PAINT, OnPaint)
ALT_MSG_MAP(1)
   MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
END_MSG_MAP()
```

The next example shows two alternate message maps. The default message map is empty.

```
BEGIN_MSG_MAP(CMyClass)
ALT_MSG_MAP(1)
   MESSAGE_HANDLER(WM_PAINT, OnPaint)
   MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
ALT_MSG_MAP(2)
   MESSAGE_HANDLER(WM_CREATE, OnCreate)
END_MSG_MAP()
```

See Also

Other Resources

Message Map Macros (ATL)

ATL Macros

END_PROP_MAP

Marks the end of the object's property map.

```
END_PROP_MAP( )
```

Remarks

When you create an object with the ATL Project Wizard, the wizard will create an empty property map by specifying BEGIN_PROP_MAP followed by **END_PROP_MAP**.

Example

See the example for BEGIN_PROP_MAP.

See Also

Reference

PROP_ENTRY
PROP_DATA_ENTRY
PROP_ENTRY_EX
PROP_PAGE

Other Resources

Property Map Macros
ATL Macros

END_RDX_MAP

Marks the end of the Registry Data Exchange map.

```
END_RDX_MAP
```

See Also

Reference

BEGIN_RDX_MAP

Other Resources

Registry Data Exchange Macros

END_SERVICE_MAP

Marks the end of the service map.

```
END_SERVICE_MAP( )
```

Example

See the example for BEGIN_SERVICE_MAP.

See Also

Reference

BEGIN_SERVICE_MAP
SERVICE_ENTRY
SERVICE_ENTRY_CHAIN

Other Resources

Service Map Macros
ATL Macros

END_SINK_MAP

Declares the end of the event sink map for the composite control.

```
END_SINK_MAP( )
```

Example

```
BEGIN_SINK_MAP(CMyObj)
   SINK_ENTRY(IDC_CIRCCTL1, DISPID_CLICK, OnClick_CircCtrl1)
   SINK_ENTRY(IDC_CIRCCTL2, DISPID_CLICK, OnClick_CircCtrl2)
END_SINK_MAP()
```

Remarks

CE ATL implementation of ActiveX event sinks only supports return values of type HRESULT or void from your event handler methods; any other return value is unsupported and its behavior is undefined.

See Also

Reference

BEGIN_SINK_MAP

SINK_ENTRY

Concepts

Composite Control Fundamentals

Other Resources

Composite Control Macros

ATL Macros

END_SNAPINTOOLBARID_MAP

Declares the end of the toolbar ID map for the Snap-In object.

```
END_SNAPINTOOLBARID_MAP(
      _class
)
```

Parameters

_class

[in] Specifies the Snap-In object class.

Example

See the example for BEGIN_SNAPINTOOLBARID_MAP.

See Also

Reference

BEGIN_SNAPINTOOLBARID_MAP
SNAPINTOOLBARID_ENTRY

Other Resources

Snap-In Object Macros
ATL Macros

EXTENSION_SNAPIN_DATACLASS

Adds a data member to the snap-in extension data class for an **ISnapInItemImpl**-derived class.

```
EXTENSION_SNAPIN_DATACLASS(
      dataClass
)
```

Parameters

dataClass

[in] The data class of the snap-in extension.

Remarks

This class should also be entered into a snap-in extension data class map. Start your snap-in extension data class map with the BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP macro, add entries for each of your snap-in extension data types with the EXTENSION_SNAPIN_NODEINFO_ENTRY macro, and complete the map with the END_EXTENSION_SNAPIN_NODEINFO_MAP macro.

Example

```
EXTENSION_SNAPIN_DATACLASS(CMySnapInExtData)
...
BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP(CMySnapInExtData)
    EXTENSION_SNAPIN_NODEINFO_ENTRY(CMySnapInExtData)
END_EXTENSION_SNAPIN_NODEINFO_MAP()

class CMyExtSnapInExtData : public CSnapInItemImpl<CMyExtSnapInExtData>
{
...
};

...

class CMyExtSnapIn :
   public CSnapInObjectRoot<1, CMyExtSnapIn>,
   public IComponentDataImpl<CMyExtSnapIn, CMyExtSnapInComponent>,
   ...
{
...
EXTENSION_SNAPIN_DATACLASS(CMyExtSnapInExtData)
...

};
```

See Also

Reference

BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP
EXTENSION_SNAPIN_NODEINFO_ENTRY
END_EXTENSION_SNAPIN_NODEINFO_MAP

Other Resources

Snap-In Object Macros
ATL Macros

EXTENSION_SNAPIN_NODEINFO_ENTRY

Adds a snap-in extension data class to the snap-in extension data class map.

```
EXTENSION_SNAPIN_NODEINFO_ENTRY(
      dataClass
)
```

Parameters

dataClass

[in] The data class of the snap-in extension.

Remarks

Start your snap-in extension data class map with the BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP macro, add entries for each of your snap-in extension data types with the **EXTENSION_SNAPIN_NODEINFO_ENTRY** macro, and complete the map with the END_EXTENSION_SNAPIN_NODEINFO_MAP macro.

Example

See the example for BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP.

See Also

Reference

BEGIN_EXTENSION_SNAPIN_NODEINFO_MAP
END_EXTENSION_SNAPIN_NODEINFO_MAP
EXTENSION_SNAPIN_DATACLASS

Other Resources

Snap-In Object Macros
ATL Macros

FORWARD_NOTIFICATIONS

Forwards notification messages to the parent window.

```
FORWARD_NOTIFICATIONS( )
```

Remarks

Specify this macro as part of your message map.

See Also

Reference

REFLECT_NOTIFICATIONS

Other Resources

Message Map Macros (ATL)

ATL Macros

IMPLEMENTED_CATEGORY

Add an **IMPLEMENTED_CATEGORY** macro to your component's category map to specify that it should be registered as implementing the category identified by the *catID* parameter.

```
IMPLEMENTED_CATEGORY(
      catID
)
```

Parameters

catID

[in] A **CATID** constant or variable holding the globally unique identifier (GUID) for the implemented category. The address of *catID* will be taken and added to the map. See the table below for a selection of stock categories.

Remarks

The component categories listed in the map will be registered automatically when the module is registered if the class has an associated OBJECT_ENTRY_AUTO or OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO macro.

Clients can use the category information registered for the class to determine its capabilities and requirements without having to create an instance of it.

For more information about component categories, see What are Component Categories and how do they work? in the *Platform SDK*.

A Selection of Stock Categories

Description	Symbol	Registry GUID
Safe For Scripting	CATID_SafeForScripting	{7DD95801-9882-11CF-9FA9-00AA006C42C4}
Safe For Initialization	CATID_SafeForInitializing	{7DD95802-9882-11CF-9FA9-00AA006C42C4}
Simple Frame Site Containment	CATID_SimpleFrameControl	{157083E0-2368-11cf-87B9-00AA006C8166}
Simple Data Binding	CATID_PropertyNotifyControl	{157083E1-2368-11cf-87B9-00AA006C8166}
Advanced Data Binding	CATID_VBDataBound	{157083E2-2368-11cf-87B9-00AA006C8166}
Windowless Controls	CATID_WindowlessObject	{1D06B600-3AE3-11cf-87B9-00AA006C8166}
Internet-Aware Objects	See Internet Aware Objects in the *Platform SDK* for a sample list.	

Example

```
BEGIN_CATEGORY_MAP(CMyControl)
    IMPLEMENTED_CATEGORY(CATID_Insertable)
    IMPLEMENTED_CATEGORY(CATID_SafeForScripting)
END_CATEGORY_MAP()
```

Smart Device Developer Notes

Not supported for smart device projects.

See Also

Reference

BEGIN_CATEGORY_MAP

REQUIRED_CATEGORY

END_CATEGORY_MAP

Other Resources

Category Macros

ATL Macros

MESSAGE_HANDLER

Defines an entry in a message map.

```
MESSAGE_HANDLER(
      msg,
      func
)
```

Parameters

msg

[in] The Windows message.

func

[in] The name of the message-handler function.

Remarks

MESSAGE_HANDLER maps a Windows message to the specified handler function.

Any function specified in a **MESSAGE_HANDLER** macro must be defined as follows:

```
LRESULT MessageHandler(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled);
```

The message map sets `bHandled` to **TRUE** before `MessageHandler` is called. If `MessageHandler` does not fully handle the message, it should set `bHandled` to **FALSE** to indicate the message needs further processing.

Note

Always begin a message map with BEGIN_MSG_MAP. You can then declare subsequent alternate message maps with ALT_MSG_MAP. The END_MSG_MAP macro marks the end of the message map. Every message map must have exactly one i nstance of **BEGIN_MSG_MAP** and **END_MSG_MAP**.

In addition to **MESSAGE_HANDLER**, you can use COMMAND_HANDLER and NOTIFY_HANDLER to map WM_COMMAND and WM_NOTIFY messages, respectively.

For more information about using message maps in ATL, see Message Maps.

Example

```
class CMyClass : ...
{
public:
   ...

   BEGIN_MSG_MAP(CMyClass)
      MESSAGE_HANDLER(WM_PAINT, OnPaint)
      ...
   END_MSG_MAP()

   // When a CMyClass object receives a WM_PAINT
   // message, the message is directed to
   // CMyClass::OnPaint for the actual processing.
   LRESULT OnPaint( ... )
   { ... }

};
```

See Also

Reference

MESSAGE_RANGE_HANDLER

Other Resources

Message Map Macros (ATL)

ATL Macros

MESSAGE_RANGE_HANDLER

Similar to MESSAGE_HANDLER, but maps a range of Windows messages to a single handler function.

```
MESSAGE_RANGE_HANDLER(
      msgFirst,
      msgLast,
      func
)
```

Parameters

msgFirst

[in] Marks the beginning of a contiguous range of messages.

msgLast

[in] Marks the end of a contiguous range of messages.

func

[in] The name of the message-handler function.

See Also

Reference

COMMAND_RANGE_HANDLER

NOTIFY_RANGE_HANDLER

Other Resources

Message Map Macros (ATL)

ATL Macros

NOTIFY_CODE_HANDLER

Similar to NOTIFY_HANDLER, but maps a WM_NOTIFY message based only on the notification code.

```
NOTIFY_CODE_HANDLER(
      cd,
      func
)
```

Parameters

cd

[in] The notification code.

func

[in] The name of the message-handler function.

See Also

Reference

NOTIFY_ID_HANDLER
REFLECTED_NOTIFY_CODE_HANDLER
NOTIFY_RANGE_HANDLER
COMMAND_CODE_HANDLER
MESSAGE_HANDLER

Other Resources

Message Map Macros (ATL)
ATL Macros

NOTIFY_HANDLER

Defines an entry in a message map.

```
NOTIFY_HANDLER(
      id,
      cd,
      func
)
```

Parameters

id

[in] The identifier of the control sending the message.

cd

[in] The notification code.

func

[in] The name of the message-handler function.

Remarks

NOTIFY_HANDLER maps a WM_NOTIFY message to the specified handler function, based on the notification code and the control identifier.

Any function specified in a **NOTIFY_HANDLER** macro must be defined as follows:

```
LRESULT NotifyHandler(int idCtrl, LPNMHDR pnmh, BOOL& bHandled);
```

The message map sets `bHandled` to **TRUE** before `NotifyHandler` is called. If `NotifyHandler` does not fully handle the message, it should set `bHandled` to **FALSE** to indicate the message needs further processing.

Note

Always begin a message map with BEGIN_MSG_MAP. You can then declare subsequent alternate message maps with ALT_MSG_MAP. The END_MSG_MAP macro marks the end of the message map. Every message map must have exactly one i nstance of **BEGIN_MSG_MAP** and **END_MSG_MAP**.

In addition to **NOTIFY_HANDLER**, you can use MESSAGE_HANDLER to map a **WM_NOTIFY** message without regard to an identifier or code. In this case, `MESSAGE_HANDLER(WM_NOTIFY, OnHandlerFunction)` will direct all **WM_NOTIFY** messages to `OnHandlerFunction`.

For more information about using message maps in ATL, see Message Maps.

Example

```
class CMyClass : ...
{
public:
   ...

   BEGIN_MSG_MAP(CMyClass)
      NOTIFY_HANDLER(IDC_MYCTL, NM_CLICK, OnClick)
      ...
   END_MSG_MAP()

   // When a CMyClass object receives a WM_NOTIFY
   // message identified by IDC_MYCTL and NM_CLICK,
   // the message is directed to CMyClass::OnClick
   // for the actual processing.
   LRESULT OnClick( ... )
```

```
    { ... }

};
```

See Also

Reference

NOTIFY_ID_HANDLER
NOTIFY_CODE_HANDLER
REFLECTED_NOTIFY_CODE_HANDLER
NOTIFY_RANGE_HANDLER
COMMAND_HANDLER

Other Resources

Message Map Macros (ATL)
ATL Macros

NOTIFY_ID_HANDLER

Similar to NOTIFY_HANDLER, but maps a WM_NOTIFY message based only on the control identifier.

```
NOTIFY_ID_HANDLER(
      id,
      func
)
```

Parameters

id

[in] The identifier of the control sending the message.

func

[in] The name of the message-handler function.

See Also

Reference

NOTIFY_CODE_HANDLER
NOTIFY_RANGE_HANDLER
COMMAND_ID_HANDLER
MESSAGE_HANDLER

Other Resources

Message Map Macros (ATL)
ATL Macros

NOTIFY_RANGE_CODE_HANDLER

Similar to NOTIFY_RANGE_HANDLER, but maps WM_NOTIFY messages with a specific notification code from a range of controls to a single handler function.

```
NOTIFY_RANGE_CODE_HANDLER(
      idFirst,
      idLast,
      cd,
      func
)
```

Parameters

idFirst

[in] Marks the beginning of a contiguous range of control identifiers.

idLast

[in] Marks the end of a contiguous range of control identifiers.

cd

[in] The notification code.

func

[in] The name of the message-handler function.

Remarks

This range is based on the identifier of the control sending the message.

See Also

Reference

NOTIFY_ID_HANDLER
NOTIFY_CODE_HANDLER
REFLECTED_NOTIFY_CODE_HANDLER
COMMAND_RANGE_HANDLER
MESSAGE_RANGE_HANDLER

Other Resources

Message Map Macros (ATL)
ATL Macros

NOTIFY_RANGE_HANDLER

Similar to NOTIFY_HANDLER, but maps WM_NOTIFY messages from a range of controls to a single handler function.

```
NOTIFY_RANGE_HANDLER(
      idFirst,
      idLast,
      func
)
```

Parameters

idFirst

[in] Marks the beginning of a contiguous range of control identifiers.

idLast

[in] Marks the end of a contiguous range of control identifiers.

func

[in] The name of the message-handler function.

Remarks

This range is based on the identifier of the control sending the message.

See Also

Reference

NOTIFY_ID_HANDLER
NOTIFY_CODE_HANDLER
REFLECTED_NOTIFY_CODE_HANDLER
COMMAND_RANGE_CODE_HANDLER
MESSAGE_RANGE_HANDLER

Other Resources

Message Map Macros (ATL)
ATL Macros

OBJECT_ENTRY_AUTO

Enters an ATL object into the object map, updates the registry, and creates an instance of the object.

```
OBJECT_ENTRY_AUTO(
      clsid,
      class
)
```

Parameters

clsid

[in] The CLSID of a COM class implemented by the C++ class named *class*.

class

[in] The name of the C++ class implementing the COM class represented by *clsid*.

Remarks

Object entry macros are placed at global scope in the project to provide support for the registration, initialization, and creation of a class.

OBJECT_ENTRY_AUTO enters the function pointers of the creator class and class-factory creator class **CreateInstance** functions for this object into the auto-generated ATL object map. When CAtlComModule::RegisterServer is called, it updates the system registry for each object in the object map.

The table below describes how the information added to the object map is obtained from the class given as the second parameter to this macro.

Information for	Obtained from
COM registration	Registry Macros
Class factory creation	Class Factory Macros
Instance creation	Aggregation Macros
Component category registration	Category Macros
Class-level initialization and cleanup	ObjectMain

See Also

Reference

OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO
DECLARE_OBJECT_DESCRIPTION
Obsolete ATL Topics
OBJECT_ENTRY
OBJECT_ENTRY_NON_CREATEABLE

Other Resources

Object Map Macros
ATL Macros

OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO

Allows you to specify that the object should be registered and initialized, but it should not be externally creatable via **CoCreateInstance**.

```
OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO(
      clsid,
      class
)
```

Parameters

clsid

[in] The CLSID of a COM class implemented by the C++ class named *class*.

class

[in] The name of the C++ class implementing the COM class represented by *clsid*.

Remarks

Object entry macros are placed at global scope in the project to provide support for the registration, initialization, and creation of a class.

OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO allows you to specify that an object should be registered and initialized (see OBJECT_ENTRY_AUTO for more information), but it should not be creatable via **CoCreateInstance**.

See Also

Reference

OBJECT_ENTRY_AUTO
Obsolete ATL Topics
OBJECT_ENTRY
OBJECT_ENTRY_NON_CREATEABLE

Other Resources

Object Map Macros
ATL Macros

PROP_DATA_ENTRY

Indicates the extent, or dimensions, of an ActiveX control.

```
PROP_DATA_ENTRY(
      szDesc,
      member,
      vt
)
```

Parameters

szDesc

[in] The property description.

member

[in] The data member containing the extent; for example, `m_sizeExtent`.

vt

[in] Specifies the VARIANT type of the property.

Remarks

This macro causes the specified data member to be persisted.

When you create an ActiveX control, the wizard inserts this macro after the property map macro BEGIN_PROP_MAP and before the property map macro END_PROP_MAP.

Example

In the following example, the extent of the object (`m_sizeExtent`) is being persisted.

```
BEGIN_PROP_MAP(CMyControl)
   PROP_DATA_ENTRY("_cx", m_sizeExtent.cx, VT_UI4)
   PROP_DATA_ENTRY("_cy", m_sizeExtent.cy, VT_UI4)
END_PROP_MAP()

// another example
BEGIN_PROP_MAP(CMyClass)
   PROP_DATA_ENTRY("Width", m_nWidth, VT_UI4)
   PROP_DATA_ENTRY("Height", m_nHeight, VT_UI4)
END_PROP_MAP()
```

See Also

Other Resources

Property Map Macros

ATL Macros

PROP_ENTRY

Use this macro to enter a property description, property DISPID, and property page CLSID into the object's property map.

```
PROP_ENTRY(
      szDesc,
      dispid,
      clsid
)
```

Parameters

szDesc

[in] The property description.

dispid

[in] The property's DISPID.

clsid

[in] The CLSID of the associated property page. Use the special value **CLSID_NULL** for a property that does not have an associated property page.

Remarks

The BEGIN_PROP_MAP macro marks the beginning of the property map; the END_PROP_MAP macro marks the end.

Example

See the example for BEGIN_PROP_MAP.

See Also

Reference

PROP_ENTRY_EX

PROP_PAGE

Other Resources

Property Map Macros

ATL Macros

PROP_ENTRY_EX

Similar to PROP_ENTRY, but allows you specify a particular IID if your object supports multiple dual interfaces.

```
PROP_ENTRY_EX(
      szDesc,
      dispid,
      clsid,
      iidDispatch
)
```

Parameters

szDesc

[in] The property description.

dispid

[in] The property's DISPID.

clsid

[in] The CLSID of the associated property page. Use the special value **CLSID_NULL** for a property that does not have an associated property page.

iidDispatch

[in] The IID of the dual interface defining the property.

Remarks

The BEGIN_PROP_MAP macro marks the beginning of the property map; the END_PROP_MAP macro marks the end.

Example

The following example groups entries for `IMyDual1` followed by an entry for `IMyDual2`. Grouping by dual interface will improve performance.

```
BEGIN_PROP_MAP( CMyClass )
   PROP_ENTRY_EX( "Caption", DISPID_CAPTION,
                  CLSID_CMyProps, IID_IMyDual1 )
   PROP_ENTRY_EX( "Enabled", DISPID_ENABLED,
                  CLSID_CMyProps, IID_IMyDual1 )
   PROP_ENTRY_EX( "Width", DISPID_WIDTH,
                  CLSID_CMyProps, IID_IMyDual2 )
END_PROP_MAP( )
```

See Also

Reference

PROP_PAGE

Other Resources

Property Map Macros

ATL Macros

PROP_PAGE

Use this macro to enter a property page CLSID into the object's property map.

```
PROP_PAGE(
      clsid
)
```

Parameters

clsid

[in] The CLSID of a property page.

Remarks

PROP_PAGE is similar to PROP_ENTRY, but does not require a property description or DISPID.

Note
If you have already entered a CLSID with **PROP_ENTRY** or PROP_ENTRY_EX, you do not need to make an additional entry wi th **PROP_PAGE**.

The BEGIN_PROP_MAP macro marks the beginning of the property map; the END_PROP_MAP macro marks the end.

Example

```
BEGIN_PROP_MAP( CMyClass )
   PROP_PAGE( CLSID_CMyClassPropPage1 )
   PROP_PAGE( CLSID_CMyClassPropPage2 )
END_PROP_MAP( )
```

See Also

Other Resources

Property Map Macros

ATL Macros

RDX_BINARY

Associates the specified registry entry with a specified member variable of type BYTE.

```
RDX_BINARY(
      rootkey,
   subkey,
   valuename,
   member,
   member_size
)
```

Parameters

rootkey

The registry key root.

subkey

The registry subkey.

valuename

The registry key.

member

The member variable to associate with the specified registry entry.

member_size

The size, in bytes, of the member variable.

Remarks

This macro is used in conjunction with the **BEGIN_RDX_MAP** and **END_RDX_MAP** macros to associate a member variable with a given registry entry. The global function RegistryDataExchange, or the member function of the same name created by the **BEGIN_RDX_MAP** and **END_RDX_MAP** macros, should be used to perform exchange of data between the system registry and the member variables in the RDX map.

See Also

Other Resources

Registry Data Exchange Macros

RDX_CSTRING_TEXT

Associates the specified registry entry with a specified member variable of type CString.

```
RDX_CSTRING_TEXT(
      rootkey,
   subkey,
   valuename,
   member,
   member_size
)
```

Parameters

rootkey

The registry key root.

subkey

The registry subkey.

valuename

The registry key.

member

The member variable to associate with the specified registry entry.

member_size

The size, in bytes, of the member variable.

Remarks

This macro is used in conjunction with the **BEGIN_RDX_MAP** and **END_RDX_MAP** macros to associate a member variable with a given registry entry. The global function RegistryDataExchange, or the member function of the same name created by the **BEGIN_RDX_MAP** and **END_RDX_MAP** macros, should be used to perform exchange of data between the system registry and the member variables in the RDX map.

See Also

Other Resources

Registry Data Exchange Macros

RDX_DWORD

Associates the specified registry entry with a specified member variable of type DWORD.

```
RDX_DWORD(
      rootkey,
   subkey,
   valuename,
   member,
   member_size
)
```

Parameters

rootkey

The registry key root.

subkey

The registry subkey.

valuename

The registry key.

member

The member variable to associate with the specified registry entry.

member_size

The size, in bytes, of the member variable.

Remarks

This macro is used in conjunction with the **BEGIN_RDX_MAP** and **END_RDX_MAP** macros to associate a member variable with a given registry entry. The global function RegistryDataExchange, or the member function of the same name created by the **BEGIN_RDX_MAP** and **END_RDX_MAP** macros, should be used to perform exchange of data between the system registry and the member variables in the RDX map.

See Also

Other Resources

Registry Data Exchange Macros

RDX_TEXT

Associates the specified registry entry with a specified member variable of type TCHAR.

```
RDX_TEXT(
      rootkey,
   subkey,
   valuename,
   member,
   member_size
)
```

Parameters

rootkey

The registry key root.

subkey

The registry subkey.

valuename

The registry key.

member

The member variable to associate with the specified registry entry.

member_size

The size, in bytes, of the member variable.

Remarks

This macro is used in conjunction with the **BEGIN_RDX_MAP** and **END_RDX_MAP** macros to associate a member variable with a given registry entry. The global function RegistryDataExchange, or the member function of the same name created by the **BEGIN_RDX_MAP** and **END_RDX_MAP** macros, should be used to perform exchange of data between the system registry and the member variables in the RDX map.

See Also

Other Resources

Registry Data Exchange Macros

REFLECT_NOTIFICATIONS

Reflects notification messages back to the child window (control) that sent them.

```
REFLECT_NOTIFICATIONS( )
```

Remarks

Specify this macro as part of the parent window's message map.

See Also

Reference

DEFAULT_REFLECTION_HANDLER

BEGIN_MSG_MAP

Other Resources

Message Map Macros (ATL)

ATL Macros

REFLECTED_COMMAND_CODE_HANDLER

Similar to COMMAND_CODE_HANDLER, but maps commands reflected from the parent window.

```
REFLECTED_COMMAND_CODE_HANDLER(
      code,
      func
)
```

Parameters

code

[in] The notification code.

func

[in] The name of the message-handler function.

See Also

Reference

REFLECT_NOTIFICATIONS

Other Resources

Message Map Macros (ATL)

ATL Macros

REFLECTED_COMMAND_HANDLER

Similar to COMMAND_HANDLER, but maps commands reflected from the parent window.

```
REFLECTED_COMMAND_HANDLER(
      id,
      code,
      func
)
```

Parameters

id

[in] The identifier of the menu item, control, or accelerator.

code

[in] The notification code.

func

[in] The name of the message-handler function.

See Also

Reference

REFLECT_NOTIFICATIONS

Other Resources

Message Map Macros (ATL)

ATL Macros

REFLECTED_COMMAND_ID_HANDLER

Similar to COMMAND_ID_HANDLER, but maps commands reflected from the parent window.

```
REFLECTED_COMMAND_ID_HANDLER(
      id,
      func
)
```

Parameters

id

[in] The identifier of the menu item, control, or accelerator.

func

[in] The name of the message-handler function.

See Also

Reference

REFLECT_NOTIFICATIONS

Other Resources

Message Map Macros (ATL)

ATL Macros

REFLECTED_COMMAND_RANGE_CODE_HANDLER

Similar to COMMAND_RANGE_CODE_HANDLER, but maps commands reflected from the parent window.

```
REFLECTED_COMMAND_RANGE_CODE_HANDLER(
      idFirst,
      idLast,
      code,
      func
)
```

Parameters

idFirst

[in] Marks the beginning of a contiguous range of control identifiers.

idLast

[in] Marks the end of a contiguous range of control identifiers.

code

[in] The notification code.

func

[in] The name of the message-handler function.

See Also

Reference

REFLECT_NOTIFICATIONS

Other Resources

Message Map Macros (ATL)

ATL Macros

REFLECTED_COMMAND_RANGE_HANDLER

Similar to COMMAND_RANGE_HANDLER, but maps commands reflected from the parent window.

```
REFLECTED_COMMAND_RANGE_HANDLER(
      idFirst,
      idLast,
      func
)
```

Parameters

idFirst

[in] Marks the beginning of a contiguous range of control identifiers.

idLast

[in] Marks the end of a contiguous range of control identifiers.

func

[in] The name of the message-handler function.

See Also

Reference

REFLECT_NOTIFICATIONS

Other Resources

Message Map Macros (ATL)

ATL Macros

REFLECTED_NOTIFY_CODE_HANDLER

Similar to NOTIFY_CODE_HANDLER, but maps notifications reflected from the parent window.

```
REFLECTED_NOTIFY_CODE_HANDLER_EX(
      cd,
      func
)
```

Parameters

cd

[in] The notification code.

func

[in] The name of the message-handler function.

See Also

Reference

REFLECT_NOTIFICATIONS

Other Resources

Message Map Macros (ATL)

ATL Macros

REFLECTED_NOTIFY_HANDLER

Similar to NOTIFY_HANDLER, but maps notifications reflected from the parent window.

```
REFLECTED_NOTIFY_HANDLER(
      id,
      cd,
      func
)
```

Parameters

id

[in] The identifier of the menu item, control, or accelerator.

cd

[in] The notification code.

func

[in] The name of the message-handler function.

See Also

Reference

REFLECT_NOTIFICATIONS

Other Resources

Message Map Macros (ATL)

ATL Macros

REFLECTED_NOTIFY_ID_HANDLER

Similar to NOTIFY_ID_HANDLER, but maps notifications reflected from the parent window.

```
REFLECTED_NOTIFY_ID_HANDLER(
      id,
      func
)
```

Parameters

id

[in] The identifier of the menu item, control, or accelerator.

func

[in] The name of the message-handler function.

See Also

Reference

REFLECT_NOTIFICATIONS

Other Resources

Message Map Macros (ATL)

ATL Macros

REFLECTED_NOTIFY_RANGE_CODE_HANDLER

Similar to NOTIFY_RANGE_CODE_HANDLER, but maps notifications reflected from the parent window.

```
REFLECTED_NOTIFY_RANGE_CODE_HANDLER(
      idFirst,
      idLast,
      cd,
      func
)
```

Parameters

idFirst

[in] Marks the beginning of a contiguous range of control identifiers.

idLast

[in] Marks the end of a contiguous range of control identifiers.

cd

[in] The notification code.

func

[in] The name of the message-handler function.

See Also

Reference

REFLECT_NOTIFICATIONS

Other Resources

Message Map Macros (ATL)

ATL Macros

REFLECTED_NOTIFY_RANGE_HANDLER

Similar to NOTIFY_RANGE_HANDLER, but maps notifications reflected from the parent window.

```
REFLECTED_NOTIFY_RANGE_HANDLER(
      idFirst,
      idLast,
      func
)
```

Parameters

idFirst

[in] Marks the beginning of a contiguous range of control identifiers.

idLast

[in] Marks the end of a contiguous range of control identifiers.

func

[in] The name of the message-handler function.

See Also

Reference

REFLECT_NOTIFICATIONS

Other Resources

Message Map Macros (ATL)

ATL Macros

REQUIRED_CATEGORY

Add a **REQUIRED_CATEGORY** macro to your component's category map to specify that it should be registered as requiring the category identified by the *catID* parameter.

```
REQUIRED_CATEGORY(
      catID
)
```

Parameters

catID

[in] A **CATID** constant or variable holding the globally unique identifier (GUID) for the required category. The address of *catID* will be taken and added to the map. See the table below for a selection of stock categories.

Remarks

The component categories listed in the map will be registered automatically when the module is registered if the class has an associated OBJECT_ENTRY_AUTO or OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO macro.

Clients can use the category information registered for the class to determine its capabilities and requirements without having to create an instance of it. For example, a control may require that a container support data binding. The container can find out if it has the capabilities necessary to host the control by querying the category manager for the categories required by that control. If the container does not support a required feature, it can refuse to host the COM object.

For more information about component categories, including a sample list, see What are Component Categories and how do they work? in the *Platform SDK*.

A Selection of Stock Categories

Description	Symbol	Registry GUID
Safe For Scripting	CATID_SafeForScripting	{7DD95801-9882-11CF-9FA9-00AA006C42C4}
Safe For Initialization	CATID_SafeForInitializing	{7DD95802-9882-11CF-9FA9-00AA006C42C4}
Simple Frame Site Containment	CATID_SimpleFrameControl	{157083E0-2368-11cf-87B9-00AA006C8166}
Simple Data Binding	CATID_PropertyNotifyControl	{157083E1-2368-11cf-87B9-00AA006C8166}
Advanced Data Binding	CATID_VBDataBound	{157083E2-2368-11cf-87B9-00AA006C8166}
Windowless Controls	CATID_WindowlessObject	{1D06B600-3AE3-11cf-87B9-00AA006C8166}
Internet-Aware Objects	See Internet Aware Objects in the *Platform SDK* for a sample list.	

Example

```
BEGIN_CATEGORY_MAP(CMyControl)
    REQUIRED_CATEGORY(CATID_InternetAware)
END_CATEGORY_MAP()
```

Smart Device Developer Notes

Not supported for smart device projects.

See Also

Reference

BEGIN_CATEGORY_MAP

IMPLEMENTED_CATEGORY

END_CATEGORY_MAP

Other Resources

Category Macros

ATL Macros

SERVICE_ENTRY

Indicates that the object supports the service id specified by *SID*.

```
SERVICE_ENTRY(
      SID
)
```

Parameters

SID

The service ID.

Example

See the example for BEGIN_SERVICE_MAP.

See Also

Reference

SERVICE_ENTRY_CHAIN
BEGIN_SERVICE_MAP
END_SERVICE_MAP

Other Resources

Service Map Macros
ATL Macros

SERVICE_ENTRY_CHAIN

Instructs IServiceProviderImpl::QueryService to chain to the object specified by *punk*.

```
SERVICE_ENTRY_CHAIN(
      punk
)
```

Parameters

punk

A pointer to the **IUnknown** interface to which to chain.

Example

See the example for BEGIN_SERVICE_MAP.

See Also

Reference

SERVICE_ENTRY
BEGIN_SERVICE_MAP
END_SERVICE_MAP

Other Resources

Service Map Macros
ATL Macros

SINK_ENTRY

Declares the handler function (*fn*) for the specified event (*dispid*), of the control identified by *id*.

```
SINK_ENTRY(
      id,
      dispid,
      fn
)
```

Parameters

id

[in] Identifies the control.

dispid

[in] Identifies the specified event.

fn

[in] Name of the event handler function. This function must use the **_stdcall** calling convention and have the appropriate dispinterface-style signature.

Example

```
BEGIN_SINK_MAP(CMyObj)
   SINK_ENTRY(IDC_CIRCCTL1, DISPID_CLICK, OnClick_CircCtrl1)
   SINK_ENTRY(IDC_CIRCCTL2, DISPID_CLICK, OnClick_CircCtrl2)
END_SINK_MAP()
```

Remarks

CE ATL implementation of ActiveX event sinks only supports return values of type HRESULT or void from your event handler methods; any other return value is unsupported and its behavior is undefined.

See Also

Reference

BEGIN_SINK_MAP
SINK_ENTRY_EX

Concepts

Composite Control Fundamentals

Other Resources

Composite Control Macros
ATL Macros

SINK_ENTRY_EX

Declares the handler function (*fn*) for the specified event (*dispid*), of the dispatch interface (*iid)*, for the control identified by *id*.

```
SINK_ENTRY_EX(
      id,
      iid,
      dispid,
      fn
)
```

Parameters

id

[in] Identifies the control.

iid

[in] Identifies the dispatch interface.

dispid

[in] Identifies the specified event.

fn

[in] Name of the event handler function. This function must use the **_stdcall** calling convention and have the appropriate dispinterface-style signature.

Example

```
BEGIN_SINK_MAP(CMyObj)
   SINK_ENTRY_EX(IDC_CIRCCTL1, CIRCLib::DIID__CircEvents, DISPID_CLICK,
      OnClick_CircCtrl1)
   SINK_ENTRY_EX(IDC_CIRCCTL2, CIRCLib::DIID__CircEvents, DISPID_CLICK,
      OnClick_CircCtrl2)
END_SINK_MAP()
```

Remarks

CE ATL implementation of ActiveX event sinks only supports return values of type HRESULT or void from your event handler methods; any other return value is unsupported and its behavior is undefined.

See Also

Reference

BEGIN_SINK_MAP
SINK_ENTRY

Concepts

Composite Control Fundamentals

Other Resources

Composite Control Macros
ATL Macros

SINK_ENTRY_INFO

Use the **SINK_ENTRY_INFO** macro within an event sink map to provide the information needed by IDispEventSimpleImpl to route events to the relevant handler function.

```
SINK_ENTRY_INFO(
      id,
      iid,
      dispid,
      fn,
      info
)
```

Parameters

id

[in] Unsigned integer identifying the event source. This value must match the *nID* template parameter used in the related IDispEventSimpleImpl base class.

iid

[in] IID identifying the dispatch interface.

dispid

[in] DISPID identifying the specified event.

fn

[in] Name of the event handler function. This function must use the **_stdcall** calling convention and have the appropriate dispinterface-style signature.

info

[in] Type information for the event handler function. This type information is provided in the form of a pointer to an **_ATL_FUNC_INFO** structure. **CC_CDECL** is the only option supported in Windows CE for the *CALLCONV* field of the **_ATL_FUNC_INFO** structure. Any other value is unsupported thus its behavior undefined.

Remarks

The first four macro parameters are the same as those for the SINK_ENTRY_EX macro. The final parameter provides type information for the event. CE ATL implementation of ActiveX event sinks only supports return values of type HRESULT or void from your event handler methods; any other return value is unsupported and its behavior is undefined.

See Also

Reference

BEGIN_SINK_MAP
SINK_ENTRY
SINK_ENTRY_EX
IDispEventSimpleImpl Class

Concepts

Supporting IDispEventImpl

Other Resources

Composite Control Macros
ATL Macros

SNAPINMENUID

Use this macro to declare the context menu resource of the Snap-In object.

```
SNAPINMENUID(
      id
)
```

Parameters

id

[in] Identifies the context menu of the Snap-In object.

See Also

Other Resources

Snap-In Object Macros

ATL Macros

SNAPINTOOLBARID_ENTRY

Use this macro to enter a toolbar ID into the Snap-In object's toolbar ID map.

```
SNAPINTOOLBARID_ENTRY(
      id
)
```

Parameters

id

[in] Identifies the toolbar control.

Remarks

The BEGIN_SNAPINTOOLBARID_MAP macro marks the beginning of the toolbar ID map; the END_SNAPINTOOLBARID_MAP macro marks the end.

Example

See the example for BEGIN_SNAPINTOOLBARID_MAP.

See Also

Other Resources

Snap-In Object Macros

ATL Macros

WM_FORWARDMSG

This macro forwards a message received by a window to another window for processing.

```
WM_FORWARDMSG
```

Return Value

Nonzero if the message was processed, zero if not.

Remarks

Use **WM_FORWARDMSG** to forward a message received by a window to another window for processing. The LPARAM and WPARAM parameters are used as follows:

Parameter	Usage
WPARAM	Data defined by user
LPARAM	A pointer to a **MSG** structure that contains information about a message

Example

In the following example, `hWndOther` represents the other window that receives this message.

```
LRESULT CMyWindow::OnMessage(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
{
   MSG msg = { m_hWnd, uMsg, wParam, lParam, 0, { 0, 0 } };
   LRESULT lRet = SendMessage(hWndOther, 0, (LPARAM)&msg);
   if(lRet == 0)   // not handled
      bHandled = FALSE;
   return 0;
}
```

See Also

Other Resources

Windows Messages Macros

ATL Macros

ATL Operators Alphabetical Reference

This section contains the reference topics for the ATL global operators.

Operator	Description
operator ==	Compares two **CSid** objects or **SID** structures for equality.
operator !=	Compares two **CSid** objects or **SID** structures for inequality.
operator <	Tests if the **CSid** object or **SID** structure on the left side of the operator is less than the **CSid** object or **SID** structure on the right side (for STL compatibility).
operator >	Tests if the **CSid** object or **SID** structure on the left side of the operator is greater than the **CSid** object or **SID** structure on the right side (for STL compatibility).
operator <=	Tests if the **CSid** object or **SID** structure on the left side of the operator is less than or equal to the **CSid** object or **SID** structure on the right side (for STL compatibility).
operator >=	Tests if the **CSid** object or **SID** structure on the left side of the operator is greater than or equal to the **CSid** object or **SID** structure on the right side (for STL compatibility).

These operators are all defined in the file atlsecurity.h.

operator != (ATL)

Compares **CSid** objects or **SID** (security identifier) structures for inequality.

```
bool operator ==(
   const CSid & lhs,
   const CSid & rhs,
) throw();
```

Parameters

lhs

The first **CSid** object or **SID** structure to compare.

rhs

The second **CSid** object or **SID** structure to compare.

Return Value

Returns **true** if the objects are not equal, **false** if they are equal.

See Also

Reference

operator == (ATL)

Other Resources

ATL Operators Alphabetical Reference

operator < (ATL)

Tests if the **CSid** object or **SID** structure on the left side of the operator is less than the **CSid** object or **SID** structure on the right side (for STL compatibility).

```
bool operator <(
   const CSid & lhs,
   const CSid & rhs,
) throw();
```

Parameters

lhs

The first **CSid** object or **SID** structure to compare.

rhs

The second **CSid** object or **SID** structure to compare.

Return Value

Returns **true** if the address of the *lhs* object is less than the address of the *rhs* object, **false** otherwise.

Remarks

This operator acts on the address of the **CSid** object or **SID** structure, and is implemented to provide compatibility with STL collection classes.

See Also

Reference

operator > (ATL)

Other Resources

ATL Operators Alphabetical Reference

operator <= (ATL)

Tests if the **CSid** object or **SID** structure on the left side of the operator is less than or equal to the **CSid** object or **SID** structure on the right side (for STL compatibility).

```
bool operator <(
   const CSid & lhs,
   const CSid & rhs,
) throw();
```

Parameters

lhs

The first **CSid** object or **SID** structure to compare.

rhs

The second **CSid** object or **SID** structure to compare.

Return Value

Returns **true** if the address of the *lhs* is less than or equal to the address of the *rhs*, **false** otherwise.

Remarks

This operator acts on the address of the **CSid** object or **SID** structure, and is implemented to provide compatibility with STL collection classes.

See Also

Reference

operator >= (ATL)

Other Resources

ATL Operators Alphabetical Reference

operator == (ATL)

Compares **CSid** objects or **SID** (security identifier) structures for equality.

```
bool operator ==(
   const CSid & lhs,
   const CSid & rhs,
) throw();
```

Parameters

lhs

The first **CSid** object or **SID** structure to compare.

rhs

The second **CSid** object or **SID** structure to compare.

Return Value

Returns **true** if the objects are equal, **false** if they are not equal.

See Also

Reference

operator != (ATL)

Other Resources

ATL Operators Alphabetical Reference

operator > (ATL)

Tests if the **CSid** object or **SID** structure on the left side of the operator is greater than the **CSid** object or **SID** structure on the right side (for STL compatibility).

```
bool operator <(
   const CSid & lhs,
   const CSid & rhs,
) throw();
```

Parameters

lhs

The first **CSid** object or **SID** structure to compare.

rhs

The second **CSid** object or **SID** structure to compare.

Return Value

Returns **true** if the address of the *lhs* is greater than the address of the *rhs*, **false** otherwise.

Remarks

This operator acts on the address of the **CSid** object or **SID** structure, and is implemented to provide compatibility with STL collection classes.

See Also

Reference

operator < (ATL)

Other Resources

ATL Operators Alphabetical Reference

operator >= (ATL)

Tests if the **CSid** object or **SID** structure on the left side of the operator is greater than or equal to the **CSid** object or **SID** structure on the right side (for STL compatibility).

```
bool operator <(
   const CSid & lhs,
   const CSid & rhs,
) throw();
```

Parameters

lhs

The first **CSid** object or **SID** structure to compare.

rhs

The second **CSid** object or **SID** structure to compare.

Return Value

Returns **true** if the address of the *lhs* is greater than or equal to the address of the *rhs*, **false** otherwise.

Remarks

This operator acts on the address of the **CSid** object or **SID** structure, and is implemented to provide compatibility with STL collection classes.

See Also

Reference

operator <= (ATL)

Other Resources

ATL Operators Alphabetical Reference

ATL Structures

The Active Template Library includes the following structures.

In This Section

ATL_DRAWINFO

Contains information used for rendering to various targets, such as a printer, metafile, or ActiveX control.

_AtlCreateWndData

Contains class instance data in windowing code in ATL.

_ATL_BASE_MODULE70

Used by any project that uses ATL.

_ATL_COM_MODULE70

Used by COM-related code in ATL.

_ATL_FUNC_INFO

Contains type information used to describe a method or property on a dispinterface.

_ATL_MODULE70

Contains data used by every ATL module.

_ATL_WIN_MODULE70

Used by windowing code in ATL.

Related Sections

ATL Reference

Provides reference material for the ATL Library, a set of template-based C++ classes that simplify the programming of COM objects.

ATL Functions

Provides reference material on the global functions organized alphabetically. Includes topics organizing the functions into categories.

ATL Global Variables

Provides reference material on the global variables organized alphabetically.

ATL Macros

Provides reference material on the macros organized alphabetically. Includes topics organizing the macros into categories.

ATL Typedefs

Provides reference material on the typedefs organized alphabetically

ATL Classes

Provides reference material on the classes organized alphabetically.

ATL_DRAWINFO Structure

Contains information used for rendering to various targets, such as a printer, metafile, or ActiveX control.

```
struct ATL_DRAWINFO{
   UINT cbSize;
   DWORD dwDrawAspect;
   LONG lindex;
   DVTARGETDEVICE* ptd;
   HDC hicTargetDev;
   HDC hdcDraw;
   LPCRECTL prcBounds;
   LPCRECTL prcWBounds;
   BOOL bOptimize;
   BOOL bZoomed;
   BOOL bRectInHimetric;
   SIZEL ZoomNum;
   SIZEL ZoomDen;
};
```

Members

cbSize

The size of the structure, in bytes.

dwDrawAspect

Specifies how the target is to be represented. Representations can include content, an icon, a thumbnail, or a printed document. For a list of possible values, see DVASPECT and DVASPECT2.

lindex

Portion of the target that is of interest for the draw operation. Its interpretation varies depending on the value in the **dwDrawAspect** member.

ptd

Pointer to a DVTARGETDEVICE structure that enables drawing optimizations depending on the aspect specified. Note that newer objects and containers that support optimized drawing interfaces support this member as well. Older objects and containers that do not support optimized drawing interfaces always specify **NULL** for this member.

hicTargetDev

Information context for the target device pointed to by **ptd** from which the object can extract device metrics and test the device's capabilities. If **ptd** is **NULL**, the object should ignore the value in the **hicTargetDev** member.

hdcDraw

The device context on which to draw. For a windowless object, the **hdcDraw** member is in the **MM_TEXT** mapping mode with its logical coordinates matching the client coordinates of the containing window. In addition, the device context should be in the same state as the one normally passed by a **WM_PAINT** message.

prcBounds

Pointer to a RECTL structure specifying the rectangle on **hdcDraw** and in which the object should be drawn. This member controls the positioning and stretching of the object. This member should be **NULL** to draw a windowless in-place active object. In every other situation, **NULL** is not a legal value and should result in an **E_INVALIDARG** error code. If the container passes a non-**NULL** value to a windowless object, the object should render the requested aspect into the specified device context and rectangle. A container can request this from a windowless object to render a second, non-active view of the object or to print the object.

prcWBounds

If **hdcDraw** is a metafile device context (see GetDeviceCaps in the *Platform SDK*), this is a pointer to a **RECTL** structure specifying the bounding rectangle in the underlying metafile. The rectangle structure contains the window extent and window origin. These values are useful for drawing metafiles. The rectangle indicated by **prcBounds** is nested inside this

prcWBounds rectangle; they are in the same coordinate space.

bOptimize

Nonzero if the drawing of the control is to be optimized, otherwise 0. If the drawing is optimized, the state of the device context is automatically restored when you are finished rendering.

bZoomed

Nonzero if the target has a zoom factor, otherwise 0. The zoom factor is stored in **ZoomNum**.

bRectInHimetric

Nonzero if the dimensions of **prcBounds** are in **HIMETRIC**, otherwise 0.

ZoomNum

The width and height of the rectangle into which the object is rendered. The zoom factor along the x-axis (the proportion of the object's natural size to its current extent) of the target is the value of **ZoomNum.cx** divided by the value of **ZoomDen.cx**. The zoom factor along the y-axis is achieved in a similar fashion.

ZoomDen

The actual width and height of the target.

Remarks

Typical usage of this structure would be the retrieval of information during the rendering of the target object. For example, you could retrieve values from **ATL_DRAWINFO** inside your overload of CComControlBase::OnDrawAdvanced.

This structure stores pertinent information used to render the appearance of an object for the target device. The information provided can be used in drawing to the screen, a printer, or even a metafile.

Requirements

Header: atlctl.h

See Also

Reference

IViewObject::Draw

CComControlBase::OnDrawAdvanced

Other Resources

ATL Structures

_AtlCreateWndData Structure

This structure contains class instance data in windowing code in ATL.

```
struct _AtlCreateWndData{
   void* m_pThis;
   DWORD m_dwThreadID;
   _AtlCreateWndData* m_pNext;
};
```

Members

m_pThis

The **this** pointer used to get access to the class instance in window procedures.

m_dwThreadID

The thread ID of the current class instance.

m_pNext

Pointer to the next **_AtlCreateWndData** object.

Requirements

Header: atlbase.h

See Also

Other Resources

ATL Structures

_ATL_BASE_MODULE70 Structure

Used by any project that uses ATL.

```
struct _ATL_BASE_MODULE70{
   UINT cbSize;
   HINSTANCE m_hInst;
   HINSTANCE m_hInstResource;
   bool m_bNT5orWin98;
   DWORD dwAtlBuildVer;
   GUID* pguidVer;
   CRITICAL_SECTION m_csResource;
   CSimpleArray<HINSTANCE> m_rgResourceInstance;
};
```

Members

cbSize

The size of the structure, used for versioning.

m_hInst

The **hInstance** for this module (either exe or dll).

m_hInstResource

Default instance resource handle.

m_bNT5orWin98

Operating system version information. Used internally by ATL.

dwAtlBuildVer

Stores the version of ATL. Currently 0x0700.

pguidVer

ATL's internal GUID.

m_csResource

Used to synchronize access to the **m_rgResourceInstance** array. Used internally by ATL.

m_rgResourceInstance

Array used to search for resources in all the resource instances of which ATL is aware. Used internally by ATL.

Remarks

_ATL_BASE_MODULE is defined as a typedef of **_ATL_BASE_MODULE70**.

Requirements

Header: atlcore.h

See Also

Other Resources

ATL Structures

_ATL_COM_MODULE70 Structure

Used by COM-related code in ATL.

```
struct _ATL_COM_MODULE70{
   UINT cbSize;
   HINSTANCE m_hInstTypeLib;
   _ATL_OBJMAP_ENTRY** m_ppAutoObjMapFirst;
   _ATL_OBJMAP_ENTRY** m_ppAutoObjMapLast;
   CRITICAL_SECTION m_csObjMap;
};
```

Members

cbSize

The size of the structure, used for versioning.

m_hInstTypeLib

The handle instance to the type library for this module.

m_ppAutoObjMapFirst

Address of the array element indicating the beginning of the object map entries for this module.

m_ppAutoObjMapLast

Address of the array element indicating the end of the object map entries for this module.

m_csObjMap

Critical section to serialize access to the object map entries. Used internally by ATL.

Remarks

_ATL_COM_MODULE is defined as a typedef of **_ATL_COM_MODULE70**.

Requirements

Header: atlbase.h

See Also

Other Resources

ATL Structures

_ATL_FUNC_INFO Structure

Contains type information used to describe a method or property on a dispinterface.

```
struct _ATL_FUNC_INFO{
   CALLCONV cc;
   VARTYPE vtReturn;
   SHORT nParams;
   VARTYPE pVarTypes[_ATL_MAX_VARTYPES];
};
```

Members

cc

The calling convention. When using this structure with the IDispEventSimpleImpl class, this member must be **CC_STDCALL**. **CC_CDECL** is the only option supported in Windows CE for the *CALLCONV* field of the **_ATL_FUNC_INFO** structure. Any other value is unsupported thus its behavior undefined.

vtReturn

The variant type of the function return value.

nParams

The number of function parameters.

pVarTypes

An array of variant types of the function parameters.

Remarks

Internally, ATL uses this structure to hold information obtained from a type library. You may need to manipulate this structure directly if you provide type information for an event handler used with the IDispEventSimpleImpl class and SINK_ENTRY_INFO macro.

Example

Given a dispinterface method defined in IDL:

```
void SomeFunction([in] long Number, [in] BSTR String);
```

you would define an **_ATL_FUNC_INFO** structure:

```
_ATL_FUNC_INFO info = {CC_STDCALL, VT_EMPTY, 2, {VT_I4, VT_BSTR} };
```

Requirements

Header: atlcom.h

See Also

Reference

IDispEventSimpleImpl Class
SINK_ENTRY_INFO
VARTYPE

Other Resources

ATL Structures

_ATL_MODULE70 Structure

Contains data used by every ATL module.

```
struct _ATL_MODULE70{
   UINT cbSize;
   LONG m_nLockCnt;
   _ATL_TERMFUNC_ELEM* m_pTermFuncs;
   CComCriticalSection m_csStaticDataInitAndTypeInfo;
};
```

Members

cbSize

The size of the structure, used for versioning.

m_nLockCnt

Reference count to determine how long the module should stay alive.

m_pTermFuncs

Tracks functions that have been registered to be called when ATL shuts down.

m_csStaticDataInitAndTypeInfo

Used to coordinate access to internal data in multithreaded situations.

Remarks

_ATL_MODULE is defined as a typedef of **_ATL_MODULE70**.

Requirements

Header: atlbase.h

See Also

Other Resources

ATL Structures

_ATL_WIN_MODULE70 Structure

Used by windowing code in ATL.

```
struct _ATL_WIN_MODULE70{
   UNIT cbSize;
   CRITICAL_SECTION m_csWindowCreate;
   _AtlCreateWndData* m_pCreateWndList;
   CSimpleArray<ATOM> m_rgWindowClassAtoms;
};
```

Members

cbSize

The size of the structure, used for versioning.

m_csWindowCreate

Used to serialize access to window registration code. Used internally by ATL.

m_pCreateWndList

Used to bind windows to their objects. Used internally by ATL.

m_rgWindowClassAtoms

Used to track window class registrations so that they can be properly unregistered at termination. Used internally by ATL.

Remarks

_ATL_WIN_MODULE is defined as a typedef of **_ATL_WIN_MODULE70**.

Requirements

Header: atlbase.h

See Also

Other Resources

ATL Structures

ATL Typedefs

The Active Template Library includes the following typedefs.

_ATL_BASE_MODULE	Defined as a typedef based on _ATL_BASE_MODULE70.
_ATL_COM_MODULE	Defined as a typedef based on _ATL_COM_MODULE70.
_ATL_MODULE	Defined as a typedef based on _ATL_MODULE70.
_ATL_WIN_MODULE	Defined as a typedef based on _ATL_WIN_MODULE70
CComDispatchDriver	This class manages COM interface pointers.
CComGlobalsThreadModel	Calls the appropriate thread model methods, regardless of the threading model being used.
CComObjectThreadModel	Calls the appropriate thread model methods, regardless of the threading model being used.
CContainedWindow	This class is a specialization of **CContainedWindowT.**
CSimpleValArray	Represents an array for storing simple types.

See Also

Other Resources

ATL Reference
ATL Functions
ATL Global Variables
ATL Structures
ATL Macros
ATL Classes

_ATL_BASE_MODULE

Defined as a typedef based on _ATL_BASE_MODULE70.

```
typedef ATL::_ATL_BASE_MODULE70 _ATL_BASE_MODULE;
```

Remarks

Used in every ATL project. Based on _ATL_BASE_MODULE70.

Classes that are part of the ATL 7.0 Module Classes derive from the _ATL_BASE_MODULE structure. For more information on ATL Module Classes, refer to COM Modules Classes.

Requirements

Header: atlcore.h

See Also

Concepts

COM Modules Classes

Other Resources

ATL Typedefs

_ATL_COM_MODULE

Defined as a typedef based on _ATL_COM_MODULE70.

```
typedef ATL::_ATL_COM_MODULE70 _ATL_COM_MODULE;
```

Remarks

Used by ATL projects which use COM features. Based on _ATL_COM_MODULE70.

Requirements

Header: atlbase.h

See Also

Other Resources

ATL Typedefs

_ATL_MODULE

Defined as a typedef based on _ATL_MODULE70.

```
typedef ATL::_ATL_MODULE70 _ATL_MODULE;
```

Remarks

Based on _ATL_MODULE70.

Requirements

Header: atlbase.h

See Also

Other Resources

ATL Typedefs

_ATL_WIN_MODULE

Defined as a typedef based on _ATL_WIN_MODULE70.

```
typedef ATL::_ATL_WIN_MODULE70 _ATL_WIN_MODULE;
```

Remarks

Used by any ATL projects which use windowing features. Based on _ATL_WIN_MODULE70.

Requirements

Header: atlbase.h

See Also

Other Resources

ATL Typedefs

CComDispatchDriver

This class manages COM interface pointers.

```
typedef CComQIPtr<IDispatch, &__uuidof(IDispatch)> CComDispatchDriver;
```

Requirements

Header: atlbase.h

See Also

Reference

CComQIPtr Class

Other Resources

ATL Typedefs

CComGlobalsThreadModel

Calls the appropriate thread model methods, regardless of the threading model being used.

```
#if defined( _ATL_SINGLE_THREADED )
      typedef CComSingleThreadModel CComGlobalsThreadModel;
#elif defined( _ATL_APARTMENT_THREADED )
      typedef CComMultiThreadModel CComGlobalsThreadModel;
#elif defined( _ATL_FREE_THREADED )
      typedef CComMultiThreadModel CComGlobalsThreadModel;
#else
      #pragma message ("No global threading model defined")
#endif
```

Remarks

Depending on the threading model used by your application, the **typedef** name **CComGlobalsThreadModel** references either CComSingleThreadModel or CComMultiThreadModel. These classes provide additional **typedef** names to reference a critical section class.

Note
CComGlobalsThreadModel does not reference class CComMultiThreadModelNoCS.

Using **CComGlobalsThreadModel** frees you from specifying a particular threading model class. Regardless of the threading model being used, the appropriate methods will be called.

In addition to **CComGlobalsThreadModel**, ATL provides the **typedef** name CComObjectThreadModel. The class referenced by each **typedef** depends on the threading model used, as shown in the following table:

typedef	Single threading	Apartment threading	Free threading
CComObjectThreadModel	S	S	M
CComGlobalsThreadModel	S	M	M

S=**CComSingleThreadModel**; M=**CComMultiThreadModel**

Use **CComObjectThreadModel** within a single object class. Use **CComGlobalsThreadModel** in an object that is globally available to your program, or when you want to protect module resources across multiple threads.

Requirements

Header: atlbase.h

See Also

Reference

CComObjectRootEx Class

Other Resources

ATL Class Overview

ATL Typedefs

CComObjectThreadModel

Calls the appropriate thread model methods, regardless of the threading model being used.

```
#if defined( _ATL_SINGLE_THREADED )
      typedef CComSingleThreadModel CComObjectThreadModel;
#elif defined( _ATL_APARTMENT_THREADED )
      typedef CComSingleThreadModel CComObjectThreadModel;
#elif defined( _ATL_FREE_THREADED )
      typedef CComMultiThreadModel CComObjectThreadModel;
#else
      #pragma message ("No global threading model defined")
#endif
```

Remarks

Depending on the threading model used by your application, the **typedef** name **CComObjectThreadModel** references either CComSingleThreadModel or CComMultiThreadModel. These classes provide additional **typedef** names to reference a critical section class.

Note
CComObjectThreadModel does not reference class CComMultiThreadModelNoCS.

Using **CComObjectThreadModel** frees you from specifying a particular threading model class. Regardless of the threading model being used, the appropriate methods will be called.

In addition to **CComObjectThreadModel**, ATL provides the **typedef** name CComGlobalsThreadModel. The class referenced by each **typedef** depends on the threading model used, as shown in the following table:

typedef	Single threading	Apartment threading	Free threading
CComObjectThreadModel	S	S	M
CComGlobalsThreadModel	S	M	M

S=**CComSingleThreadModel**; M=**CComMultiThreadModel**

Use **CComObjectThreadModel** within a single object class. Use **CComGlobalsThreadModel** in an object that is either globally available to your program, or when you want to protect module resources across multiple threads.

Requirements

Header: atlbase.h

See Also

Reference

CComObjectRootEx Class

Other Resources

ATL Class Overview

ATL Typedefs

CContainedWindow

This class is a specialization of **CContainedWindowT.**

```
typedef CContainedWindowT<CWindow> CContainedWindow;
```

Remarks

CContainedWindow is a specialization of CContainedWindowT. If you want to change the base class or traits, use **CContainedWindowT** directly.

Requirements

Header: atlwin.h

See Also

Reference

CContainedWindowT Class

Other Resources

ATL Typedefs

CSimpleValArray

Represents an array for storing simple types.

```
#define CSimpleValArray CSimpleArray
```

Remarks

CSimpleValArray is provided for creating and managing arrays containing simple data types. It is a simple #define of CSimpleArray.

Requirements

Header: atlsimpcoll.h

See Also

Reference

CSimpleArray Class

Other Resources

ATL Class Overview

ATL Typedefs

Obsolete ATL Topics

The following items are obsolete in the Active Template Library (ATL).

Classes

- CComConnectionPoint
- CComConnectionPointContainerImpl
- CComDualImpl
- CComDynamicArrayCONNECTDATA
- CComISupportErrorInfoImpl
- CComProvideClassInfo2Impl
- CComStaticArrayCONNECTDATA

Methods, Macros, and Global Functions

- AtlTrace
- atlTraceFlags
- BEGIN_OBJECT_MAP
- CComModule::UpdateRegistryFromResource
- BEGIN_PROPERTY_MAP
- COM_INTERFACE_ENTRY_IMPL
- COM_INTERFACE_ENTRY_IMPL_IID
- DECLARE_STATIC_REGISTRY_RESOURCE
- DECLARE_STATIC_REGISTRY_RESOURCEID
- END_OBJECT_MAP
- END_PROPERTY_MAP
- OBJECT_ENTRY
- OBJECT_ENTRY_NON_CREATEABLE
- RELEASE_AND_DESTROY

Deprecated ATL Functions

The following table lists deprecated functions and the functions that should be used in their place.

ATL_DEPRECATED Function	Substitute Function
AtlModuleRegisterClassObjects	AtlComModuleRegisterClassObjects
AtlModuleRevokeClassObjects	AtlComModuleRevokeClassObjects
AtlModuleGetClassObject	AtlComModuleGetClassObject
AtlModuleRegisterServer	AtlComModuleRegisterServer
AtlModuleUnregisterServer	AtlComModuleUnregisterServer
AtlModuleUnregisterServerEx	AtlComModuleUnregisterServer

AtlModuleUpdateRegistryFromResourceD	AtlUpdateRegistryFromResourceD
AtlModuleRegisterTypeLib	AtlRegisterTypeLib
AtlModuleUnRegisterTypeLib	AtlUnRegisterTypeLib
AtlModuleLoadTypeLib	AtlLoadTypeLib
AtlModuleInit	No longer required: CAtlModule::CAtlModule constructor performs the functionality.
AtlModuleTerm	No longer required: CAtlModule::~CAtlModule destructor performs the functionality.
AtlModuleAddCreateWndData	AtlWinModuleAddCreateWndData
AtlModuleExtractCreateWndData	AtlWinModuleExtractCreateWndData
RegisterProgID	Obsolete method of registration. Use RGS files instead. See Creating Registrar Scripts for details.
UpdateRegistryClass	Obsolete method of registration. Use RGS files instead. See Creating Registrar Scripts for details.
RegisterClassHelper	Obsolete method of registration. Use RGS files instead. See Creating Registrar Scripts for details.
UnregisterClassHelper	Obsolete method of registration. Use RGS files instead. See Creating Registrar Scripts for details.
CRegKey::SetValue(DWORD dwValue, LPCTSTR lpszValueName);	CRegKey::SetDWORDValue
CRegKey::SetValue(LPCTSTR lpszValue, LPCTSTR lpszValueName = NULL, bool bMulti = false, int nValueLen = -1);	CRegKey::SetStringValue *-or-* CRegKey::SetMultiStringValue
CRegKey::QueryValue(DWORD& dwValue, LPCTSTR lpszValueName);	CRegKey::QueryDWORDValue
CRegKey::QueryValue(LPTSTR szValue, LPCTSTR lpszValueName, DWORD* pdwCount);	CRegKey::QueryStringValue *-or-* CRegKey::QueryMultiStringValue
CDynamicAccessor::GetColumnInfo(IRowset* pRowset, DBORDINAL* pColumns, DBCOLUMNINFO** ppColumnInfo);	CDynamicAccessor::GetColumnInfo

See Also

Reference

ATL and MFC Version Numbers

Other Resources

ATL Reference

AtlTrace (Obsolete) (ATL)

Beginning in ATL 7.0, **AtlTrace** is obsolete. Use ATLTRACE2 instead.

```
inline void _cdecl AtlTrace(
   LPCSTR lpszFormat,
   ...
);
inline void _cdecl AtlTrace(
   LPCWSTR lpszFormat,
   ...
);
```

Parameters

lpszFormat

[in] The string and variables to send to the Visual C++ output window or any application that traps these messages.

Remarks

Sends the specified string to the debugger of the current application. For example:

```
AtlTrace(_T("The value of x is %d.\n"), x);
```

Note

AtlTrace can be used in both debug and release builds, but it has no effect in release builds. The call to the function is evaluated, but the function does nothing. In debug builds, the function sends the specified string, after formatting, to the dump device.

AtlTrace limits the contents of the string to be sent to the dump device to no more than 511 characters, after formatting.

See Also

Reference

ATLTRACE (ATL)

Obsolete ATL Topics

Other Resources

Debugging and Error Reporting Global Functions

atlTraceFlags

Beginning in ATL 7.0, **atlTraceFlags** is obsolete. Use ATLTRACE2 instead.

```
int atlTraceFlags;
```

Remarks

Used to turn on ATL's built-in reporting features. To use these flags, call **ATLTRACE2** and use the category you want to trace as the first parameter. See ATLTRACE2 for a list of the categories and their descriptions.

Use **ATLTRACE2** only for debug builds of your application.

You can filter for specific trace flags in your project by using the global functions **ATL_TRACE_CATEGORY** and **ATL_TRACE_LEVEL**. See **ATLTRACE2** for a description of their use.

See Also

Reference

ATLTRACE (ATL)

Obsolete ATL Topics

Other Resources

Debugging and Error Reporting Global Functions

BEGIN_OBJECT_MAP

The **BEGIN_OBJECT_MAP** macro is obsolete. It is provided for backward compatibility. See OBJECT_ENTRY_AUTO for the current functionality.

Marks the beginning of the object map that provides support for the registration, initialization, and creation of instances of ATL COM classes.

```
BEGIN_OBJECT_MAP( x )
```

Parameters

x

[in] The name used for an array of object map entries.

Remarks

The parameter *x* is the name of an array holding **_ATL_OBJMAP_ENTRY** structures. Each element of the array describes a COM class and its corresponding C++ implementation in terms of:

- The CLSID of the COM class
- COM registration code for the class
- Class factory creation code
- Instance creation code
- Component category registration code
- Class-level initialization and cleanup code

Start your object map with the **BEGIN_OBJECT_MAP** macro, add entries for each class with the OBJECT_ENTRY macro, and complete the map with the END_OBJECT_MAP macro. Typically, wizards create the object map and add the appropriate entries automatically, but you may occasionally need to modify this code by hand.

The object map needs to be passed to CComModule::Init to enable the module's registration, initialization, and creation code to hook into the information provided.

Example

The example below, taken from the CIRCCOLL sample, shows a simple object map and the call to CComModule::Init from a DLL server:

```
BEGIN_OBJECT_MAP(ObjectMap)
   OBJECT_ENTRY(CLSID_MyCircleCollectionCreator, CMyCircleCollectionCreator)
END_OBJECT_MAP( )

//DLL Entry Point
extern "C"
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID /*lpReserved*/)
{
   if (dwReason == DLL_PROCESS_ATTACH)
   {
      _Module.Init(ObjectMap, hInstance);
      DisableThreadLibraryCalls(hInstance);
   }
   else if (dwReason == DLL_PROCESS_DETACH)
      _Module.Term();
   return TRUE;
}
```

See Also

Reference

Obsolete ATL Topics

BEGIN_PROPERTY_MAP

Beginning with ATL 3.0, **BEGIN_PROPERTY_MAP** is replaced by BEGIN_PROP_MAP. **BEGIN_PROPERTY_MAP** could be used only with controls, and it automatically saved out and read in the extent, or dimensions, of the map. This limitation no longer exists in ATL 3.0, and no need exists for reading in the property map's extent.

For backward compatibility, ATL defines the following:

```
#define BEGIN_PROPERTY_MAP(theClass) \
   typedef _ATL_PROP_NOTIFY_EVENT_CLASS __ATL_PROP_NOTIFY_EVENT_CLASS; \
   typedef theClass _PropMapClass; \
   static ATL_PROPMAP_ENTRY* GetPropertyMap()\
   {\
      static ATL_PROPMAP_ENTRY pPropMap[] = \
      { \
         {OLESTR("_cx"), 0, &CLSID_NULL, NULL, offsetof(_PropMapClass, m_sizeExtent.cx), si
zeof(long), VT_UI4}, \
         {OLESTR("_cy"), 0, &CLSID_NULL, NULL, offsetof(_PropMapClass, m_sizeExtent.cy), si
zeof(long), VT_UI4},
```

See Also

Reference

Obsolete ATL Topics

CComConnectionPoint

Beginning with ATL 2.0, the **CComConnectionPoint** class is obsolete and is replaced by IConnectionPointImpl.

See Also
Reference
Obsolete ATL Topics

CComConnectionPointContainerImpl

Beginning with ATL 2.0, the **CComConnectionPointContainerImpl** class is obsolete and is replaced by IConnectionPointContainerImpl. For backward compatibility, ATL defines the following:

```
#define CComConnectionPointContainerImpl
        IConnectionPointContainerImpl
```

See Also

Reference

Obsolete ATL Topics

CComDualImpl

Beginning with ATL 2.0, the **CComDualImpl** class name is obsolete and is replaced by IDispatchImpl. For backward compatibility, ATL defines the following:

```
#define CComDualImpl IDispatchImpl
```

See Also

Reference

Obsolete ATL Topics

CComDynamicArrayCONNECTDATA

Beginning with ATL 2.0, the **CComDynamicArrayCONNECTDATA** class is obsolete and is replaced by CComDynamicUnkArray.

See Also

Reference

Obsolete ATL Topics

COM_INTERFACE_ENTRY_IMPL

Beginning with ATL 3.0, the **COM_INTERFACE_ENTRY_IMPL** macro is obsolete and is replaced by COM_INTERFACE_ENTRY.

See Also
Reference
Obsolete ATL Topics

COM_INTERFACE_ENTRY_IMPL_IID

Beginning with ATL 3.0, the **COM_INTERFACE_ENTRY_IMPL_IID** macro is obsolete and is replaced by COM_INTERFACE_ENTRY_IID.

See Also

Reference

Obsolete ATL Topics

CComISupportErrorInfoImpl

Beginning with ATL 2.0, the **CComISupportErrorInfoImpl** class name is obsolete and is replaced by ISupportErrorInfoImpl. For backward compatibility, ATL defines the following:

```
#define CComISupportErrorInfoImpl
        ISupportErrorInfoImpl
```

See Also

Reference

Obsolete ATL Topics

CComModule::UpdateRegistryFromResource

Beginning with ATL 2.0, the **CComModule::UpdateRegistryFromResource** method is obsolete and is replaced by CComModule::UpdateRegistryFromResourceD. For backward compatibility, ATL defines the following:

```
#define UpdateRegistryFromResource
        UpdateRegistryFromResourceD
```

See Also

Reference

Obsolete ATL Topics

CComProvideClassInfo2Impl

Beginning with ATL 2.0, the **CComProvideClassInfo2Impl** class name is obsolete and is replaced by IProvideClassInfo2Impl. For backward compatibility, ATL defines the following:

```
#define CComProvideClassInfo2Impl
        IProvideClassInfo2Impl
```

See Also

Reference

Obsolete ATL Topics

CComStaticArrayCONNECTDATA

Beginning with ATL 2.0, the **CComStaticArrayCONNECTDATA** class is obsolete and is replaced by CComUnkArray.

See Also
Reference
Obsolete ATL Topics

DECLARE_STATIC_REGISTRY_RESOURCE

Beginning with ATL 2.0, the **DECLARE_STATIC_REGISTRY_RESOURCE** macro is obsolete. To statically link to the Registrar, you specify the `#define _ATL_STATIC_REGISTRY` statement in stdafx.h and use the DECLARE_REGISTRY_RESOURCE macro.

For backward compatibility, ATL defines the following:

```
#define DECLARE_STATIC_REGISTRY_RESOURCE(x)
        DECLARE_REGISTRY_RESOURCE(x)
```

See Also

Reference

Obsolete ATL Topics

DECLARE_STATIC_REGISTRY_RESOURCEID

Beginning with ATL 2.0, the **DECLARE_STATIC_REGISTRY_RESOURCEID** macro is obsolete. To statically link to the Registrar, you specify the `#define _ATL_STATIC_REGISTRY` statement in stdafx.h and use the DECLARE_REGISTRY_RESOURCEID macro.

For backward compatibility, ATL defines the following:

```
#define DECLARE_STATIC_REGISTRY_RESOURCEID(x)
        DECLARE_REGISTRY_RESOURCEID(x)
```

See Also

Reference

Obsolete ATL Topics

END_OBJECT_MAP

The **END_OBJECT_ENTRY** macro is obsolete. It is provided for backward compatibility. See OBJECT_ENTRY_AUTO for the current functionality.

This macro marks the end of the object map that provides support for the registration, initialization, and creation of instances of ATL COM classes.

Remarks

See BEGIN_OBJECT_MAP for more details.

See Also

Reference

Obsolete ATL Topics

END_PROPERTY_MAP

Beginning with ATL 3.0, **END_PROPERTY_MAP** is replaced by END_PROP_MAP. See BEGIN_PROPERTY_MAP for more information about the update.

For backward compatibility, ATL defines the following:

```
#define END_PROPERTY_MAP() \
         {NULL, 0, NULL, &IID_NULL, 0, 0, 0} \
      }; \
      return pPropMap; \
   }
```

See Also

Reference

Obsolete ATL Topics

OBJECT_ENTRY

The **OBJECT_ENTRY** macro is obsolete. It is provided for backward compatibility. Use OBJECT_ENTRY_AUTO.

Object entry macros can be placed in the object map to provide support for the registration, initialization, and creation of a class.

```
OBJECT_ENTRY( clsid, class )
```

Parameters

clsid

[in] The CLSID of a COM class implemented by the C++ class named *class*.

class

[in] The name of the C++ class implementing the COM class represented by *clsid*.

Remarks

Start your object map with the BEGIN_OBJECT_MAP macro, add entries for each object with the **OBJECT_ENTRY** or OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO macro, and complete the map with the END_OBJECT_MAP macro.

The table below describes how the information added to the object map is obtained from the class given as the second parameter to this macro.

Information for	Obtained from
COM registration	Registry Macros
Class factory creation	Aggregation and Class Factory Macros
Instance creation	Aggregation and Class Factory Macros
Component category registration	Category Macros
Class-level initialization and cleanup	ObjectMain

See Also

Reference

OBJECT_ENTRY_AUTO

Other Resources

ATL Macros

OBJECT_ENTRY_NON_CREATEABLE

Object entry macros can be placed in the object map to provide support for the registration, initialization, and creation of a class.

Beginning with ATL 4.0, the **OBJECT_ENTRY_NON_CREATEABLE** macro is deprecated in favor of the OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO macro, which provides full support for registering component categories using ATL's category map.

All new code should use the new macro. Existing code that used the **OBJECT_ENTRY_NON_CREATEABLE** macro and ATL's component category registration macros for the same class should also be updated to use the new macro.

For backward compatibility, the **OBJECT_ENTRY_NON_CREATEABLE** macro remains in the ATL headers so existing code will continue to work. Documentation for the original macro is provided below:

```
OBJECT_ENTRY_NON_CREATEABLE( class )
```

Parameters

class

[in] The class of the object to be registered and initialized.

Remarks

This macro allows you to specify that a class should be registered and initialized, but that instances of this class should not be externally creatable via standard COM functions such as CoCreateInstance. You can still create instances of such classes from within the same project using standard C++ techniques.

See Also

Reference

OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO

Other Resources

ATL Macros

RELEASE_AND_DESTROY

Beginning with ATL 2.0, the **RELEASE_AND_DESTROY** macro is obsolete.

See Also

Reference

Obsolete ATL Topics

ATL/MFC Shared Classes

Beginning with Visual C++ .NET 2002, several existing MFC utility classes were rewritten or revised to reduce their dependencies on other MFC classes. These utility classes can now be used in any native C++ project. This section only includes classes that were previously available to MFC projects and have now been shared, plus a few new classes related to the changes in **CString**. It does not include the ATL and ATL Server classes, which can be used in any native C++ project type by inclusion of the appropriate header.

In This Section

Classes Shared Between MFC and ATL

Provides links to the classes shared between MFC and ATL.

Related Sections

Active Template Library (ATL) Reference

Provides reference material for the ATL Library, a set of template-based C++ classes that simplify the programming of COM objects.

Microsoft Foundation Class Library (MFC) Reference

Provides reference material for the MFC Library, a set of classes in that constitute an application framework, which is the framework of an application written for the Windows API.

Visual C++ Samples

Provides links to sample code showing the capabilities of Visual C++ and the libraries and technologies it supports.

Visual C++ Libraries

Provides links to the various libraries provided with Visual C++, including ATL, MFC, OLE DB Templates, the C run-time library, and the Standard C++ Library.

Debugging

Provides links to using the Visual Studio debugger to correct logic errors in your application or stored procedures.

Breaking Changes (ATL/MFC)

Breaking changes in ATL/MFC shared classes

Breaking changes

- COleDateTime::operator = breaking change

  In the overload **COleDateTime& operator=(const FILETIME&** *filetimeSrc***)**, the FILETIME value is correctly converted and copied into this **COleDateTime** object. If the conversion is successful, the status of this object is set to valid; if unsuccessful, it is set to invalid. **FILETIME** uses Universal Coordinated Time (UTC), so if you pass a local time in the structure, your results will be converted from local file time to system time to variant time. Note that this corrected behavior is the same as in Visual C++ 6.0 and Visual C++.NET 2003. See File Times in the *Platform SDK* for more information.

- CComVariant::operator == now does not consider **VT_EMPTY** to be equal to a variant of another type whose value is zero.

See Also

Concepts

What's New in the Visual C++ Libraries

Other Resources

ATL/MFC Shared Classes

ATL/MFC Concepts

This section provides conceptual and task-based topics to help you program using the classes shared between Active Template Library (ATL) and Microsoft Foundation Class (MFC) Library.

In This Section

Strings (ATL/MFC)

Describes how to manage string data in applications.

See Also

Other Resources

ATL/MFC Shared Classes

Date and Time

MFC supports several different ways of working with dates and times. These include:

- General-purpose time classes. The CTime and CTimeSpan classes encapsulate most of the functionality associated with the ANSI-standard time library, which is declared in TIME.H.
- Support for system clock. With MFC version 3.0, support was added to **CTime** for the Win32 **SYSTEMTIME** and **FILETIME** data types.
- Support for the Automation DATE data type. **DATE** supports date, time, and date/time values. The COleDateTime and COleDateTimeSpan classes encapsulate this functionality. They work with the COleVariant class using Automation support.

What do you want to know more about?

- Date and Time: General-Purpose Classes
- Date and Time: SYSTEMTIME Support
- Date and Time: Automation Support
- Date and Time: Database Support

See Also

Concepts

General MFC Topics

Other Resources

MFC Concepts

Date and Time: General-Purpose Classes

This article describes how to take advantage of the class library general-purpose services related to date and time management. Procedures described include:

- Getting the current time
- Calculating elapsed time
- Formatting a string representation of a date/time

The **CTime** class provides a way to represent date and time information easily. The **CTimeSpan** class represents elapsed time, such as the difference between two **CTime** objects.

Note
CTime objects can be used to represent dates between January 1, 1970, and January 18, 2038. **CTime** objects have a resolution of 1 second. **CTime** is based on the **time_t** data type, defined in the *Run-Time Library Reference*.

See Also

Concepts

Date and Time

Current Time: General Purpose Classes

The following procedure shows how to create a **CTime** object and initialize it with the current time.

To get the current time

1. Allocate a **CTime** object, as follows:

```
CTime theTime;
```

Note
Uninitialized **CTime** objects are not initialized to a valid time.

2. Call the **CTime::GetCurrentTime** function to get the current time from the operating system. This function returns a **CTime** object that can be used to set the value of **CTime**, as follows:

```
theTime = CTime::GetCurrentTime();
```

Since **GetCurrentTime** is a static member function from the **CTime** class, you must qualify its name with the name of the class and the scope resolution operator (**::**), `CTime::GetCurrentTime()`.

Of course, the two steps outlined previously could be combined into a single program statement as follows:

```
CTime theTime = CTime::GetCurrentTime();
```

See Also

Concepts

Date and Time: General-Purpose Classes

Elapsed Time: General-Purpose Classes

The following procedure shows how to calculate the difference between two **CTime** objects and get a **CTimeSpan** result.

To calculate elapsed time

- Use the **CTime** and **CTimeSpan** objects to calculate the elapsed time, as follows:

```
CTime startTime = CTime::GetCurrentTime();

// ... perform time-consuming task ...

CTime endTime = CTime::GetCurrentTime();

CTimeSpan elapsedTime = endTime - startTime;
```

  Once you have calculated `elapsedTime`, you can use the member functions of **CTimeSpan** to extract the components of the elapsed-time value.

See Also

Concepts

Date and Time: General-Purpose Classes

Formatting Time Values: General-Purpose Classes

The following procedure shows how to format time values.

To format a string representation of a time or elapsed time

- Use the **Format** member function from either the CTime or CTimeSpan classes to create a character string representation of the time or elapsed time, as shown by the following example.

```
CTime t( 1991, 3, 19, 22, 15, 0 );
// 10:15PM March 19, 1991

CString s = t.Format( "%A, %B %d, %Y" );
// s == "Tuesday, March 19, 1991"
```

What do you want to know more about?

- General date and time programming in MFC
- Working with SYSTEMTIME
- Automation support of date and time programming

See Also

Concepts

Date and Time: General-Purpose Classes

Date and Time: SYSTEMTIME Support

The CTime class has constructors that accept system and file times from Win32. If you use **CTime** objects for these purposes, you must modify their initialization accordingly, as described in this article.

MFC still provides **CTime** constructors that take time arguments of the MS-DOS style, but, with MFC version 3.0, the **CTime** class also supports a constructor that takes a Win32 **SYSTEMTIME** structure and another that takes a Win32 **FILETIME** structure.

The new **CTime** constructors are:

- **CTime(const SYSTEMTIME&** *sysTime* **);**
- **CTime(const FILETIME&** *fileTime* **);**

The *fileTime* parameter is a reference to a Win32 **FILETIME** structure, which represents time as a 64-bit value, a more convenient format for internal storage than a **SYSTEMTIME** structure and the format used by Win32 to represent the time of file creation.

If your code contains a **CTime** object initialized with the system time, you should use the **SYSTEMTIME** constructor in Win32.

You most likely will not use **CTime FILETIME** initialization directly. If you use a **CFile** object to manipulate a file, CFile::GetStatus retrieves the file timestamp for you via a **CTime** object initialized with a **FILETIME** structure.

What do you want to know more about?

- General date and time programming in MFC
- Automation support of date and time programming
- General-purpose classes for date and time programming

See Also

Concepts

Date and Time

Date and Time: Automation Support

This article describes how to take advantage of the class library services related to date and time management. Procedures described include:

- Getting the current time
- Calculating elapsed time
- Formatting a string representation of a date/time

The COleDateTime class provides a way to represent date and time information. It provides finer granularity and a greater range than the CTime class. The COleDateTimeSpan class represents elapsed time, such as the difference between two **COleDateTime** objects.

The **COleDateTime** and **COleDateTimeSpan** classes are designed to be used with the **COleVariant** class used in Automation. **COleDateTime** and **COleDateTimeSpan** are also useful in MFC database programming, but they can be used whenever you want to manipulate date and time values. Although the **COleDateTime** class has a greater range of values and finer granularity than the **CTime** class, it requires more storage per object than **CTime**. There are also some special considerations when working with the underlying **DATE** type. See The DATE Type for more details on the implementation of **DATE**.

COleDateTime objects can be used to represent dates between January 1, 100, and December 31, 9999. **COleDateTime** objects are floating point values, with an approximate resolution of 1 millisecond. **COleDateTime** is based on the **DATE** data type, defined in the MFC documentation under COleDateTime::operator DATE. The actual implementation of **DATE** extends beyond these bounds. The **COleDateTime** implementation imposes these bounds to facilitate working with the class.

COleDateTime does not support Julian dates. The Gregorian calendar is assumed to extend back in time to January 1, 100.

COleDateTime ignores Daylight Saving Time (DST). The following code example compares two methods of calculating a time span that crosses the DST switchover date: one using the CRT, and the other using **COleDateTime**. DST switches over, in most locales, in the second week in April and the third in October.

The first method sets two **CTime** objects, *time1* and *time2*, to April 5 and April 6 respectively, using the standard C type structures **tm** and **time_t**. The code displays *time1* and *time2* and the time span between them.

The second method creates two **COleDateTime** objects, *oletime1* and *oletime2*, and sets them to the same dates as *time1* and *time2*. It displays *oletime1* and *oletime2* and the time span between them.

The CRT correctly calculates a difference of 23 hours. **COleDateTimeSpan** calculates a difference of 24 hours.

Note that a workaround is used near the end of the example to display the date properly using **COleDateTime::Format**. See the Knowledge Base article "BUG: Format("%D") Fails for **COleDateTime** and **COleDateTimeSpan**" (Q167338).

```
#include <afxdisp.h>

void CTimetestView::OnDraw(CDC* pDC)
{
   CTimetestDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   time_t date1_t, date2_t;
   tm date_tm;

   date_tm.tm_hour   =12;
   date_tm.tm_min   =0;
   date_tm.tm_mon   =3;
   date_tm.tm_sec   =0;
   date_tm.tm_wday   =0; //Day of week (0-6; Sunday = 0)
   date_tm.tm_yday =0;
   date_tm.tm_year   =97;
   date_tm.tm_isdst =-1; //Positive if Daylight Saving Time is in effect;
                         //0 if Daylight Saving Time is not in effect;
                         //Negative if status of DST is unknown.

   date_tm.tm_mday   =6;
```

```
    date2_t = mktime(&date_tm);

    date_tm.tm_mday   =5;
    date_tm.tm_isdst =0;
    date1_t = mktime(&date_tm);

    CTime time1(date1_t), time2(date2_t);
    CTimeSpan ts = time2 - time1;

    pDC->TextOut(0,0, CString("CTime"));
    pDC->TextOut(0,20, time1.Format("%H:%M:%S %A, %B %d, %Y"));
    pDC->TextOut(0,40, time2.Format("%H:%M:%S %A, %B %d, %Y"));
    pDC->TextOut(0,60, ts.Format("%H:%M:%S and %D days"));


    COleDateTime oletime1(date1_t), oletime2(date2_t);
    COleDateTimeSpan olets = oletime2 - oletime1;

    pDC->TextOut(0,120, CString("COleDateTime"));
    pDC->TextOut(0,140, oletime1.Format("%H:%M:%S %A, %B %d, %Y"));
    pDC->TextOut(0,160, oletime2.Format("%H:%M:%S %A, %B %d, %Y"));

    //Work-around bug in COleDateTime::Format("%D")
    CString str;
     str.Format("%s and %d days",
       (LPCTSTR)olets.Format("%H:%M:%S"), olets.GetDays());
    pDC->TextOut(0,180, str);
  }
```

See Also

Concepts

Date and Time

Current Time: Automation Classes

The following procedure shows how to create a **COleDateTime** object and initialize it with the current time.

To get the current time

1. Create a **COleDateTime** object.
2. Call **GetCurrentTime**.

```
COleDateTime timeNow;
timeNow = COleDateTime::GetCurrentTime();
```

See Also

Concepts

Date and Time: Automation Support

Elapsed Time: Automation Classes

This procedure shows how to calculate the difference between two **CTime** objects and get a **CTimeSpan** result.

To calculate elapsed time

1. Create two **COleDateTime** objects.
2. Set one of the **COleDateTime** objects to the current time.
3. Perform some time-consuming task.
4. Set the other **COleDateTime** object to the current time.
5. Take the difference between the two times.

```
COleDateTime timeStart, timeEnd;
timeStart = COleDateTime::GetCurrentTime();
// ... perform time-consuming task
timeEnd = COleDateTime::GetCurrentTime();
COleDateTimeSpan spanElapsed = timeEnd - timeStart;
```

See Also

Concepts

Date and Time: Automation Support

Formatting Time: Automation Classes

To format a time

- Use the **Format** member function of either COleDateTime or COleDateTimeSpan to create a character string representing the time or elapsed time.

```
COleDateTime time(1970, 12, 18, 17, 30, 0);
// 18 December 1970, 5:30 PM
CString s = time.Format(VAR_DATEVALUEONLY);
// s contains the date formatted based on
// the current national language specifications
// (locale ID). The time portion is ignored for
// formatting purposes in this case.
```

For more information, see class COleVariant.

What do you want to know more about?

- General date and time programming in MFC
- General-purpose classes for date and time programming
- Working with SYSTEMTIME

See Also

Concepts

Date and Time: Automation Support

Date and Time: Database Support

As of version 4.0, MFC database programming uses the COleDateTime and COleDateTimeSpan classes to represent date and time data. These classes, also used in Automation, are derived from class COleVariant. They supply better support for managing date and time data than do CTime and CTimeSpan.

What do you want to know more about?

- Automation support of date and time programming

See Also

Concepts

Date and Time

The DATE Type

The **DATE** type is implemented using an 8-byte floating-point number. Days are represented by whole number increments starting with 30 December 1899, midnight as time zero. Hour values are expressed as the absolute value of the fractional part of the number. The following table illustrates several dates along with their **DATE** type numeric equivalent:

Date and time	Representation
30 December 1899, midnight	0.00
1 January 1900, midnight	2.00
4 January 1900, midnight	5.00
4 January 1900, 6 A.M.	5.25
4 January 1900, noon	5.50
4 January 1900, 9 P.M.	5.875

The **DATE** date type, as well as the **COleDateTime** class, represents dates and times as a classic number line. The **COleDateTime** class contains several methods for manipulating DATE values, including conversion to and from other common date formats.

The following points should be noted when working with these date and time formats in Automation:

- Dates are specified in local time; synchronization must be performed manually when working with dates in different time zones.
- The date types do not account for Daylight Savings Time.
- The date timeline becomes discontinuous for date values less than 0 (before 30 December 1899). This is because the whole-number portion of the date value is treated as signed, while the fractional part is treated as unsigned. In other words, the whole-number part of the date value may be positive or negative, while the fractional part of the date value is always added to the overall logical date. The following table illustrates a few examples:

Date and time	Representation
27 December 1899, midnight	-3.00
28 December 1899, noon	-2.50
28 December 1899, midnight	-2.00
29 December 1899, midnight	-1.00
30 December 1899, 6 P.M.	-0.75
30 December 1899, noon	-0.50
30 December 1899, 6 A.M.	-0.25
30 December 1899, midnight	0.00
30 December 1899, 6 A.M.	0.25
30 December 1899, noon	0.50

30 December 1899, 6 P.M.	0.75
31 December 1899, midnight	1.00
1 January 1900, midnight	2.00
1 January 1900, noon	2.50
2 January 1900, midnight	3.00

Notice that date values between -1.0 and 0.0 represent the same logical dates as their absolute values. This is because the negative whole-number portion of the date value did not yet subtract back a full day before the fractional portion was added.

More information on issues related to the **DATE** and **COleDateTime** types can be found under COleDateTime Class and Date and Time: Automation Support.

See Also

Reference

COleDateTime Class

Concepts

Date and Time

Strings (ATL/MFC)

Nearly all programs work with string data. Visual C++ provides several ways to manage this string data.

In This Section

Using CStringT

Describes programming using the template class CStringT.

Using CString

Describes programming using CString, the default implementation of CStringT.

Related Sections

CStringT Overview

Provides reference information about the shared **CStringT** class.

MFC Concepts

Provides conceptual and task-based topics to help you program using the MFC Library.

Native Programming

Provides links to topics describing conceptual information about the Visual C++ libraries and topics discussing various coding technologies and techniques.

See Also

Other Resources

ATL/MFC Shared Classes

Using CStringT

The topics in this section describe programming using the template class CStringT.

In This Section

Memory Management with CStringT

Discusses memory management with **CStringT**, a template class used to manipulate variable-length character strings.

Exporting String Classes Using CStringT

Explains how to export your own string class from a DLL using the CStringT<> template class.

Reference

CStringT Class.

Provides reference information about the shared **CStringT** class.

See Also

Other Resources

Strings (ATL/MFC)

Memory Management with CStringT

Class CStringT is a template class used to manipulate variable-length character strings. The memory to hold these strings is allocated and released through a string manager object, associated with each instance of **CStringT**. MFC and ATL provide default instantiations of **CStringT**, called **CString**, **CStringA**, and **CStringW**, which manipulate strings of different character types. These character types are of type **TCHAR**, **char**, and **wchar_t**, respectively. These default string types use a string manager that allocates memory from the process heap (in ATL) or the CRT heap (in MFC). For typical applications, this memory allocation scheme is sufficient. However, for code making intensive use of strings (or multithreaded code) the default memory managers may not perform optimally. This topic describes how to override the default memory management behavior of **CStringT**, creating allocators specifically optimized for the task at hand.

- Implementation of a Custom String Manager (Basic Method)
- Avoidance of Heap Contention
- Implementation of a Custom String Manager (Advanced Method)
- CFixedStringT: An Example of a Custom String Manager

See Also

Tasks

CustomString Sample: Demonstrates Custom Memory Allocators for CStringT

Implementation of a Custom String Manager (Basic Method)

The easiest way to customize the memory allocation scheme for string data is to use the ATL-provided **CAtlStringMgr** class but provide your own memory allocation routines. The constructor for **CAtlStringMgr** takes a single parameter: a pointer to an **IAtlMemMgr** object. **IAtlMemMgr** is an abstract base class that provides a generic interface to a heap. Using the **IAtlMemMgr** interface, the **CAtlStringMgr** allocates, reallocates, and frees the memory used to store string data. You can either implement the **IAtlMemMgr** interface yourself, or use one of the five ATL-provided memory manager classes. The ATL-provided memory managers simply wrap existing memory allocation facilities:

- CCRTHeap Wraps the standard CRT heap functions (malloc, free, and realloc)
- CWin32Heap Wraps a Win32 heap handle, using HeapAlloc, HeapFree, and HeapRealloc
- CLocalHeap Wraps the Win32 APIs: LocalAlloc, LocalFree, and LocalRealloc
- CGlobalHeap Wraps the Win32 APIs: GlobalAlloc, GlobalFree, and GlobalRealloc.
- CComHeap Wraps the COM Task Allocator APIs: CoTaskMemAlloc, CoTaskMemFree, and CoTaskMemRealloc

For the purpose of string memory management, the most useful class is **CWin32Heap** because it allows you to create multiple independent heaps. For example, if you wanted to use a separate heap just for strings, you could do the following:

```
// Declare a thread-safe, growable, private heap with initial size 0:
CWin32Heap g_stringHeap( 0, 0, 0 );

// Declare a string manager that uses the private heap:
CAtlStringMgr g_stringMgr( &g_stringHeap );
```

To use this private string manager to manage memory for a **CString** variable, pass a pointer to the manager as a parameter to the **CString** variable's constructor:

```
void PrintPowers( int nBase )
{
    int n = 1;
    for( int nPower = 0; nPower < 10; nPower++ )
    {
        // Use the private string manager, instead of the default:
        CString strPower( &g_stringMgr );

        strPower.Format( "%d", n );
        printf_s( "%s\n", LPCSTR( strPower ) );
        n *= nBase;
    }
}
```

See Also

Concepts

Memory Management with CStringT

Avoidance of Heap Contention

The default string managers provided by MFC and ATL are simple wrappers on top of a global heap. This global heap is fully thread-safe, meaning that multiple threads can allocate and free memory from it simultaneously without corrupting the heap. To help provide thread safety, the heap has to serialize access to itself. This is usually accomplished with a critical section or similar locking mechanism. Whenever two threads try to access the heap simultaneously, one thread is blocked until the other thread's request is finished. For many applications, this situation rarely occurs and the performance impact of the heap's locking mechanism is negligible. However, for applications that frequently access the heap from multiple threads contention for the heap's lock can cause the application to run slower than if it were single-threaded (even on machines with multiple CPUs).

Applications that use CStringT are especially susceptible to heap contention because operations on **CStringT** objects frequently require reallocation of the string buffer.

One way to alleviate heap contention between threads is to have each thread allocate strings from a private, thread-local heap. As long as the strings allocated with a particular thread's allocator are used only in that thread, the allocator need not be thread-safe.

Example

The example below illustrates a thread procedure that allocates its own private non-thread-safe heap to use for strings on that thread:

```
DWORD_PTR WINAPI WorkerThreadProc( void* pContext )
{
    // Declare a non-thread-safe heap just for this thread:
    CWin32Heap stringHeap( HEAP_NO_SERIALIZE, 0, 0 );

    // Declare a string manager that uses the thread's heap:
    CAtlStringMgr stringMgr( &stringHeap );

    int n = 1;
    for( int nPower = 0; nPower < 10; nPower++ )
    {
        // Use the thread's string manager, instead of the default:
        CString strPower( &stringMgr );

        strPower.Format( "%d", n );
        printf_s( "%s\n", LPCSTR( strPower ) );
        n *= nBase;
    }

   return( 0 );
}
```

Comments

Multiple threads could be running using this same thread procedure but since each thread has its own heap there is no contention between threads. In addition, the fact that each heap is not thread-safe gives a measurable increase in performance even if just one copy of the thread is running. This is the result of the heap not using expensive interlocked operations to protect against concurrent access.

For a more complicated thread procedure, it may be convenient to store a pointer to the thread's string manager in a thread local storage (TLS) slot. This allows other functions called by the thread procedure to access the thread's string manager.

See Also

Concepts

Memory Management with CStringT

Implementation of a Custom String Manager (Advanced Method)

In specialized situations, you might want to implement a custom string manager that does more than just change which heap is used to allocate memory. In this situation, you must manually implement the IAtlStringMgr interface as your custom string manager.

In order to do this, it is important to first understand how CStringT uses that interface to manage its string data. Every instance of **CStringT** has a pointer to a CStringData structure. This variable-length structure contains important information about the string (such as length), as well as the actual character data for the string. Every custom string manager is responsible for allocating and freeing these structures at the request of **CStringT**.

The **CStringData** structure comprises four fields:

- pStringMgr This field points to the **IAtlStringMgr** interface used to manage this string data. When **CStringT** needs to reallocate or free the string buffer it calls the Reallocate or Free methods of this interface, passing the **CStringData** structure as a parameter. When allocating a **CStringData** structure in your string manager, you must set this field to point to your custom string manager.

- nDataLength This field contains the current logical length of the string stored in the buffer excluding the terminating null. **CStringT** updates this field when the length of the string changes. When allocating a **CStringData** structure, your string manager must set this field to zero. When reallocating a **CStringData** structure, your custom string manager must leave this field unchanged.

- nAllocLength This field contains the maximum number of characters (excluding the terminating null) that can be stored in this string buffer without reallocating it. Whenever **CStringT** needs to increase the logical length of the string, it first checks this field to make sure there is enough space in the buffer. If the check fails, **CStringT** calls into your custom string manager to reallocate the buffer. When allocating or reallocating a **CStringData** structure, you must set this field to at least the number of characters requested in the **nChars** parameter to IAtlStringMgr::Allocate or IAtlStringMgr::Reallocate. If there is more space in the buffer than requested, you can set this value to reflect the actual amount of space available. This allows **CStringT** to grow the string to fill the entire allocated space before it has to call back into the string manager to reallocate the buffer.

- nRefs This field contains the current reference count of the string buffer. If the value is one, then a single instance of **CStringT** is using the buffer. In addition, the instance is allowed to both read and modify the contents of the buffer. If the value is greater than one, multiple instances of **CStringT** can use the buffer. Because the character buffer is shared, **CStringT** instances can only read the contents of the buffer. To modify the contents, **CStringT** first makes a copy of the buffer. If the value is negative, only one instance of **CStringT** is using the buffer. In this case, the buffer is considered locked. When a **CStringT** instance is using a locked buffer no other instances of **CStringT** may share the buffer. Instead, these instances create a copy of the buffer before manipulating the contents. In addition, the **CStringT** instance using the locked buffer does not attempt to share the buffer of any other **CStringT** instance assigned to it. In this case, the **CStringT** instance copies the other string into the locked buffer.

  When allocating a **CStringData** structure, you must set this field to reflect the type of sharing that is allowed for the buffer. For most implementations, set this value to one. This allows the usual copy-on-write sharing behavior. However, if your string manager does not support sharing the string buffer, set this field to a locked state. This forces **CStringT** to only use this buffer for the instance of **CStringT** that allocated it.

See Also

Concepts

Memory Management with CStringT

CFixedStringT: Example of a Custom String Manager

The ATL library implements one example of a custom string manager used by class CFixedStringT, called **CFixedStringMgr**. **CFixedStringT** is derived from CStringT and implements a string that allocates its character data as part of the **CFixedStringT** object itself as long as the string is less than the length specified by the **t_nChars** template parameter of **CFixedStringT**. With this approach, the string does not need the heap at all, unless the length of the string grows beyond the size of the fixed buffer. Because **CFixedStringT** does not always use a heap to allocate its string data, it cannot use **CAtlStringMgr** as its string manager. It uses a custom string manager (**CFixedStringMgr**), implementing the IAtlStringMgr interface. This interface is discussed in Implementation of a Custom String Manager (Advanced Method).

The constructor for **CFixedStringMgr** takes three parameters:

- **pData:** A pointer to the fixed **CStringData** structure to be used.
- **nChars:** The maximum number of characters the **CStringData** structure can hold.
- **pMgr:** A pointer to the **IAtlStringMgr** interface of a "backup string manager."

The constructor stores the values of **pData** and **pMgr** in their respective member variables (**m_pData** and **m_pMgr**). It then sets the length of the buffer to zero, the available length equal to the maximum size of the fixed buffer, and the reference count to –1. The reference count value indicates the buffer is locked and to use this instance of **CFixedStringMgr** as the string manager.

Marking the buffer as locked prevents other **CStringT** instances from holding a shared reference to the buffer. If other **CStringT** instances were allowed to share the buffer it would be possible for the buffer contained by **CFixedStringT** to be deleted while other strings were still using the buffer.

CFixedStringMgr is a full implementation of the **IAtlStringMgr** interface. The implementation of each method is discussed separately.

Implementation of CFixedStringMgr::Allocate

The implementation of **CFixedStringMgr::Allocate** first checks to see if the requested size of the string is less than or equal to the size of the fixed buffer (stored in the **m_pData** member). If the fixed buffer is large enough, **CFixedStringMgr** locks the fixed buffer with a length of zero. As long as the string length does not grow beyond the size of the fixed buffer, **CStringT** will not have to reallocate the buffer.

If the requested size of the string is greater than the fixed buffer **CFixedStringMgr** forwards the request to the backup string manager. The backup string manager is presumed to allocate the buffer from the heap. However, before returning this buffer **CFixedStringMgr** locks the buffer and replaces the buffer's string manager pointer with a pointer to the **CFixedStringMgr** object. This ensures that attempts to reallocate or free the buffer by **CStringT** will call into **CFixedStringMgr**.

Implementation of CFixedStringMgr::ReAllocate

The implementation of **CFixedStringMgr::ReAllocate** is very similar to its implementation of **Allocate**.

If the buffer being reallocated is the fixed buffer and the requested buffer size is smaller than the fixed buffer, no allocation is done. However, if the buffer being reallocated is not the fixed buffer, it must be a buffer allocated with the backup manager. In this case the backup manager is used to reallocate the buffer.

If the buffer being reallocated is the fixed buffer and the new buffer size is too large to fit within the fixed buffer, **CFixedStringMgr** allocates a new buffer using the backup manager. The contents of the fixed buffer are then copied into the new buffer.

Implementation of CFixedStringMgr::Free

The implementation of **CFixedStringMgr::Free** follows the same pattern as **Allocate** and **ReAllocate**. If the buffer being freed is the fixed buffer, the method sets it to a zero-length locked buffer. If the buffer being freed was allocated with the backup manager, **CFixedStringMgr** uses the backup manager to free it.

Implementation of CFixedStringMgr::Clone

The implementation of **CFixedStringMgr::Clone** always returns a pointer to the backup manager rather than the **CFixedStringMgr** itself. This happens because every instance of **CFixedStringMgr** can only be associated with a single instance of **CStringT**. Any other instances of **CStringT** trying to clone the manager should get the backup manager instead. This is because the backup manager supports being shared.

Implementation of CFixedStringMgr::GetNilString

The implementation of **CFixedStringMgr::GetNilString** returns the fixed buffer. Because of the one-on-one correspondence of **CFixedStringMgr** and **CStringT**, a given instance of **CStringT** never uses more than one buffer at a time. Therefore, a nil string and a real string buffer are never needed at the same time.

Whenever the fixed buffer is not in use, **CFixedStringMgr** ensures that it is initialized with a zero length. This allows it to be used as the nil string. As an added bonus, the **nAllocLength** member of the fixed buffer is always set to the full size of the fixed buffer. This means that **CStringT** can grow the string without calling IAtlStringMgr::Reallocate, even for the nil string.

See Also

Concepts

Memory Management with CStringT

Exporting String Classes Using CStringT

In the past, MFC developers have derived from **CString** to specialize their own string classes. In Microsoft Visual C++.NET (MFC 8.0), the CString class was superseded by a template class called CStringT. This provided several benefits:

- It allowed the MFC **CString** class to be used in ATL projects without linking in the larger MFC static library or DLL.
- With the new **CStringT** template class, you can customize **CString** behavior using template parameters that specify character traits, similar to the templates in the Standard Template Library (STL).
- When you export your own string class from a DLL using **CStringT**, the compiler also automatically exports the **CString** base class. Since **CString** is itself a template class, it may be instantiated by the compiler when used, unless the compiler is aware that **CString** is imported from a DLL. If you have migrated projects from Visual C++ 6.0 to Visual C++.NET, you might have seen linker symbol errors for a multiply-defined **CString** because of the collision of the **CString** imported from a DLL and the locally instantiated version. The proper way to do this is described below. For more information on this issue, see the Knowledge Base article, "Linking Errors When you Import CString-derived Classes" (Q309801) on the MSDN Library CD-ROM or at http://support.microsoft.com/default.aspx.

The following scenario will cause the linker to produce symbol errors for multiply defined classes. Assume that you are exporting a **CString**-derived class (`CMyString`) from an MFC extension DLL:

```
// MyString.h
class AFX_EXT_CLASS CMyString : public CString
{
   // Your implementation code
};
```

The consumer code uses a mixture of **CString** and `CMyString`. "MyString.h" is not included in the precompiled header, and some usage of **CString** does not have `CMyString` visible.

Assume that you use the **CString** and `CMyString` classes in separate source files, Source1.cpp and Source2.cpp. In Source1.cpp, you use `CMyString` and #include MyString.h. In Source2.cpp, you use **CString**, but do not #include MyString.h. In this case, the linker will complain about **CStringT** being multiply defined. This is caused by **CString** being both imported from the DLL that exports `CMyString`, and instantiated locally by the compiler through the **CStringT** template.

To resolve this problem, do the following:

Export **CStringA** and **CStringW** (and the necessary base classes) from MFC80.DLL. Projects that include MFC will always use the MFC DLL exported **CStringA** and **CStringW**, as in previous MFC implementations.

Then create a exportable derived class using the **CStringT** template, as `CStringT_Exported` is below, for example:

```
#ifdef _AFXDLL
   #define AFX_EXT_CSTRING AFX_EXT_CLASS
#else
   #define AFX_EXT_CSTRING
#endif

// ...
template< typename BaseType, class StringTraits >
class AFX_EXT_CSTRING CStringT_Exported
   : public CStringT< BaseType, StringTraits >
{
   // Reimplement all CStringT<> constructors and
   // forward to the base class implementation
};
```

In AfxStr.h, replace the previous **CString**, **CStringA**, and **CStringW** typedefs as follows:

```
typedef ATL::StringT_Exported< wchar_t,
      StrTraitMFC< wchar_t> > CStringW;

typedef ATL::StringT_Exported< char,
```

```
        StrTraitMFC< char > > CStringA;

 typedef ATL::StringT_Exported< TCHAR,
        StrTraitMFC< TCHAR > > CString;
```

There are several caveats:

- You should not export **CStringT** itself because this will cause ATL-only projects to export a specialized **CStringT** class.
- Using an exportable derived class from **CStringT** minimizes having to re-implement **CStringT** functionality. Additional code is limited to forwarding constructors to the **CStringT** base class.
- **CString**, **CStringA**, and **CStringW** should only be marked **__declspec(dllexport/dllimport)** when you are building with an MFC shared DLL. If linking with an MFC static library, you should not mark these classes as exported; otherwise, internal use of **CString**, **CStringA**, and **CStringW** inside user DLLs will mark **CString** as exported as well.

Related Topics

CStringT Class

See Also

Other Resources

Using CStringT
Using CString

Using CString

The topics in this section describe programming using **CString**, which is the default implementation of the template class CStringT.

In This Section

Basic CString Operations

Describes basic **CString** operations, including creating objects from C literal strings, accessing individual characters in a **CString**, concatenating two objects, and comparing **CString** objects.

String Data Management

Discusses using Unicode and MBCS with **CString**.

CString Semantics

Explains how **CString** objects are used.

CString Operations Relating to C-Style Strings

Describes manipulating the contents of a **CString** object like a C-style null-terminated string.

Allocating and Releasing Memory for a BSTR

Discusses using memory for a **BSTR** and COM objects.

CString Exception Cleanup

Explains that explicit cleanup in MFC 3.0 and later is no longer necessary.

CString Argument Passing

Explains how to pass CString objects to functions and how to return **CString** objects from functions.

Unicode and Multibyte Character Set (MBCS) Support

Discusses how MFC is enabled for Unicode and MBCS support.

Reference

CStringT Class.

Provides reference information about the shared **CStringT** class.

See Also

Other Resources

Strings (ATL/MFC)

Basic CString Operations

This article explains basic CString operations, including:

- Creating CString objects from standard C literal strings
- Accessing individual characters in a CString
- Concatenating two CString objects
- Comparing CString objects

The **CString** class provides member functions and overloaded operators that duplicate and, in some cases, surpass the string services of the C run-time libraries (for example, **strcat**).

Creating CString Objects from Standard C Literal Strings

You can assign C-style literal strings to a **CString** just as you can assign one **CString** object to another:

- Assign the value of a C literal string to a **CString** object:

```
CString myString = "This is a test";
```

- Assign the value of one **CString** to another **CString** object:

```
CString oldString = "This is a test";
CString newString = oldString;
```

The contents of a **CString** object are copied when one **CString** object is assigned to another. Thus, the two strings do not share a reference to the actual characters that make up the string. For more information on using **CString** objects as values, see the article CString Semantics.

Note
To write your application so that it can be compiled for Unicode or for ANSI, code literal strings using the _T macro. For more information, see the article Unicode and Multibyte Character Set (MBCS) Support.

Accessing Individual Characters in a CString

You can access individual characters within a **CString** object with the **GetAt** and **SetAt** member functions. You can also use the array element, or subscript, operator (**[ ]**) instead of **GetAt** to get individual characters (this is similar to accessing array elements by index, as in standard C-style strings). Index values for **CString** characters are zero based.

Concatenating Two CString Objects

To concatenate two **CString** objects, use the concatenation operators (**+** or **+=**) as follows:

```
CString s1 = "This ";        // Cascading concatenation
s1 += "is a ";
CString s2 = "test";
CString message = s1 + "big " + s2;
// Message contains "This is a big test".
```

At least one argument to the concatenation operators (**+** or **+=**) must be a **CString** object, but you can use a constant character string (such as `"big"`) or a **char** (such as 'x') for the other argument.

Comparing CString Objects

The **Compare** member function and the **==** operator for **CString** are equivalent. **Compare**, **operator==**, and **CompareNoCase** are MBCS and Unicode aware; **CompareNoCase** is also case insensitive. The **Collate** member function of **CString** is locale sensitive and is often slower than **Compare**. **Collate** should be used only where it is necessary to abide by

the sorting rules as specified by the current locale.

The following table shows the available CString comparison functions and their equivalent Unicode/MBCS-portable functions in the C run-time library:

CString function	MBCS function	Unicode function
Compare	**_mbscmp**	**wcscmp**
CompareNoCase	**_mbsicmp**	**_wcsicmp**
Collate	**_mbscoll**	**wcscoll**

The **CString** class overrides the relational operators (**<**, **<=**, **>=**, **>**, **==**, and **!=**).You can compare two **CStrings** using these operators, as shown here:

```
CString s1( "Tom" );
CString s2( "Jerry" );
if( s1 < s2 )
    ...
```

See Also

Other Resources

Strings (ATL/MFC)

String Data Management

Visual C++ provides several ways to manage string data:

- Run-time library functions for working with C-style null-terminated strings
- Win32 API functions for managing strings
- MFC's class CString, which provides flexible, resizable string objects
- Class CStringT Class, which provides an MFC-independent string object with the same functionality as **CString**

Nearly all programs work with string data. MFC's **CString** class is often the best solution for flexible string handling. As of version 7.0, **CString** can be used in MFC or MFC-independent programs. Both the run-time library and **CString** support strings containing multibyte (wide) characters, as in Unicode or MBCS programming.

This article describes the general-purpose services that the class library provides related to string manipulation. Topics covered in this article include:

- Unicode and MBCS Provide portability
- CStrings and const char Pointers
- CString Reference Counting

The CString class provides support for manipulating strings. It is intended to replace and extend the functionality normally provided by the C run-time library string package. The **CString** class supplies member functions and operators for simplified string handling, similar to those found in Basic. The class also provides constructors and operators for constructing, assigning, and comparing **CStrings** and standard C++ string data types. Because **CString** is not derived from **CObject**, you can use **CString** objects independently of most of the Microsoft Foundation Class Library (MFC).

CString objects follow "value semantics." A **CString** object represents a unique value. Think of a **CString** as an actual string, not as a pointer to a string.

A **CString** object represents a sequence of a variable number of characters. **CString** objects can be thought of as arrays of characters.

Unicode and MBCS Provide Portability

With MFC version 3.0 and later, MFC, including **CString**, is enabled for both Unicode and multibyte character sets (MBCS). This support makes it easier for you to write portable applications that you can build for either Unicode or ANSI characters. To enable this portability, each character in a **CString** object is of type **TCHAR**, which is defined as **wchar_t** if you define the symbol **_UNICODE** when you build your application, or as **char** if not. A **wchar_t** character is 16 bits wide. MBCS is enabled if you build with the symbol **_MBCS** defined. MFC itself is built with either the **_MBCS** symbol (for the NAFX libraries) or the **_UNICODE** symbol (for the UAFX libraries) defined.

Note

The **CString** examples in this and the accompanying articles on strings show literal strings properly formatted for Unicode p ortability, using the **_T** macro, which translates the literal string to the form:

```
L"literal string"
```

Note

which the compiler treats as a Unicode string. For example, the following code:

```
CString strName = _T("Name");
```

Note

is translated as a Unicode string if **_UNICODE** is defined or as an ANSI string if not. For more information, see the article Unicode and Multibyte Character Set (MBCS) Support.

A **CString** object can store up to **INT_MAX** (2,147,483,647) characters. The **TCHAR** data type is used to get or set individual characters inside a **CString** object. Unlike character arrays, the **CString** class has a built-in memory allocation capability. This allows **CString** objects to automatically grow as needed (that is, you do not have to worry about growing a **CString** object to fit longer strings).

CStrings and const char Pointers

A **CString** object also can act like a literal C-style string (an **PCXSTR**, which is the same as **const char*** if not under Unicode). The CSimpleStringT::operator PCXSTR conversion operator allows **CString** objects to be freely substituted for character pointers in function calls. The **CString(LPCWSTR** *pszSrc* **)** constructor allows character pointers to be substituted for **CString** objects.

No attempt is made to fold **CString** objects. If you make two **CString** objects containing `Chicago`, for example, the characters in `Chicago` are stored in two places. (This may not be true of future versions of MFC, so you should not depend on it.)

Note

Use the GetBuffer and ReleaseBuffer member functions when you need to directly access a **CString** as a nonconstant pointer to a character.

Note

Use the CStringT::AllocSysString and CStringT::SetSysString member functions to allocate and set **BSTR** objects used in Auto mation (formerly known as OLE Automation).

Note

Where possible, allocate **CString** objects on the frame rather than on the heap. This saves memory and simplifies parameter passing.

The **CString** class is not implemented as a Microsoft Foundation Class Library collection class, though **CString** objects can certainly be stored as elements in collections.

CString Reference Counting

As of MFC version 4.0, when CString objects are copied, MFC increments a reference count rather than copying the data. This makes passing parameters by value and returning **CString** objects by value more efficient. These operations cause the copy constructor to be called, sometimes more than once. Incrementing a reference count reduces that overhead for these common operations and makes using **CString** a more attractive option.

As each copy is destroyed, the reference count in the original object is decremented. The original **CString** object is not destroyed until its reference count is reduced to zero.

You can use the **CString** member functions LockBuffer and UnlockBuffer to disable or enable reference counting.

See Also

Concepts

General MFC Topics

CString Semantics

Even though CString objects are dynamic objects that can grow, they act like built-in primitive types and simple classes. Each **CString** object represents a unique value. **CString** objects should be thought of as the actual strings rather than as pointers to strings.

You can assign one **CString** object to another. However, when you modify one of the two **CString** objects, the other **CString** object is not modified, as shown by the following example:

```
CString s1, s2;
s1 = s2 = "hi there";

if( s1 == s2 )            // TRUE - they are equal
    ...

s1.MakeUpper();        // Does not modify s2
if( s2[0] == 'h' )        // TRUE - s2 is still "hi there"
```

Note in the example that the two **CString** objects are considered "equal" because they represent the same character string. The **CString** class overloads the equality operator (**==**) to compare two **CString** objects based on their value (contents) rather than their identity (address).

See Also

Other Resources

Strings (ATL/MFC)

CString Operations Relating to C-Style Strings

It is often useful to manipulate the contents of a CString object as if it were a C-style null-terminated string. This article covers the following topics:

- Converting to C-style null-terminated strings
- Working with standard run-time library string functions
- Modifying CString contents directly
- Using CString objects with variable argument functions
- Specifying CString formal parameters

Converting to C-Style Null-Terminated Strings

Consider the following two cases:

- In the simplest case, you can cast a **CString** object to be an **LPCTSTR**. The LPCTSTR type conversion operator returns a pointer to a read-only C-style null-terminated string from a **CString** object.

  The pointer returned by **LPCTSTR** points into the data area used by the **CString**. If the **CString** goes out of scope and is automatically deleted or something else changes the contents of the **CString**, the **LPCTSTR** pointer will no longer be valid. Treat the string to which the pointer points as temporary.

- You can use **CString** functions, such as **SetAt**, to modify individual characters in the string object. However, if you need a copy of a **CString** object's characters that you can modify directly, use the more secured function **strcpy_s** (or the Unicode/MBCS-portable **_tcscpy_s**) to copy the **CString** object into a separate buffer where the characters can be safely modified, as shown by the following example:

```
CString theString( "This is a test" );
int     sizeOfString = (theString.GetLength() + 1);
LPTSTR  lpsz = new TCHAR[ sizeOfString ];
_tcscpy_s(lpsz, sizeOfString, theString);
//... modify lpsz as much as you want
```

Note

The third argument to **strcpy_s** (or the Unicode/MBCS-portable **_tcscpy_s**) is either a **const wchar_t*** (Unicode) or a **const char*** (ANSI). The example above passes a **CString** for this argument. The C++ compiler automatically applies the conversion function defined for the **CString** class that converts a **CString** to an **LPCTSTR**. The ability to define casting operations from one type to another is one of the most useful features of C++.

Working with Standard Run-Time Library String Functions

In most situations, you should be able to find **CString** member functions to perform any string operation for which you might consider using the standard C run-time library string functions, such as **strcmp** (or the Unicode/MBCS-portable **_tcscmp**).

If you need to use the C run-time string functions, you can use the techniques described in Converting to C-Style Null-Terminated Strings to copy the **CString** object to an equivalent C-style string buffer, perform your operations on the buffer, and then assign the resulting C-style string back to a **CString** object.

Modifying CString Contents Directly

In most situations, you should use **CString** member functions to modify the contents of a **CString** object or to convert the **CString** to a C-style character string.

However, there are certain situations, such as working with operating-system functions that require a character buffer, where it is advantageous to directly modify the **CString** contents.

The **GetBuffer** and **ReleaseBuffer** member functions allow you to gain access to the internal character buffer of a **CString** object and modify it directly. The following steps show how to use these functions for this purpose:

1. Call **GetBuffer** for a **CString** object and specify the length of the buffer you require.
2. Use the pointer returned by **GetBuffer** to write characters directly into the **CString** object.
3. Call **ReleaseBuffer** for the **CString** object to update all the internal **CString** state information, such as the length of the string. After modifying a **CString** object's contents directly, you must call **ReleaseBuffer** before calling any other **CString** member functions.

Using CString Objects with Variable Argument Functions

Some C functions take a variable number of arguments. A notable example is **printf_s**. Because of the way this kind of function is declared, the compiler cannot be sure of the type of the arguments and cannot determine which conversion operation to perform on each argument. Therefore, it is essential that you use an explicit type cast when passing a **CString** object to a function that takes a variable number of arguments.

To use a CString object in a variable argument function

- Explicitly cast the **CString** to an **LPCTSTR** string, as shown here:

```
CString  kindOfFruit = "bananas";
int      howmany = 25;
printf_s( "You have %d %s\n", howmany, (LPCTSTR)kindOfFruit );
```

Specifying CString Formal Parameters

For most functions that need a string argument, it is best to specify the formal parameter in the function prototype as a **const** pointer to a character (**LPCTSTR**) instead of a **CString**. When a formal parameter is specified as a **const** pointer to a character, you can pass either a pointer to a **TCHAR** array, a literal string ["`hi there`"], or a **CString** object. The **CString** object will be automatically converted to an **LPCTSTR**. Any place you can use an **LPCTSTR**, you can also use a **CString** object.

You can also specify a formal parameter as a constant string reference (that is, **const CString&**) if the argument will not be modified. Drop the **const** modifier if the string will be modified by the function. If a default null value is desired, initialize it to the null string ["`""`"], as shown below:

```
void AddCustomer( const CString& name,
                  const CString& address,
                  const CString& comment = "" );
```

For most function results, you can simply return a **CString** object by value.

See Also

Concepts

CString Argument Passing

Other Resources

Strings (ATL/MFC)

Allocating and Releasing Memory for a BSTR

When you create **BSTR**s and pass them between COM objects, you must take care in treating the memory they use in order to avoid memory leaks. When a **BSTR** stays within an interface, you must free its memory when you are done with it. However, when a **BSTR** passes out of an interface, the receiving object takes responsibility for its memory management.

In general, the rules for allocating and releasing memory allocated for **BSTR**s are as follows:

- When you call into a function that expects a **BSTR** argument, you must allocate the memory for the **BSTR** before the call and release it afterwards. For example:

```
HRESULT IWebBrowser2::put_StatusText( BSTR bstr );

// shows using the Win32 function
// to allocate memory for the string:
BSTR bstrStatus = ::SysAllocString( L"Some text" );
if (bstrStatus == NULL)
   return E_OUTOFMEMORY;

pBrowser->put_StatusText( bstrStatus );
// Free the string:
::SysFreeString( bstrStatus );
//...
```

- When you call into a function that returns a **BSTR**, you must free the string yourself. For example:

```
HRESULT IWebBrowser2::get_StatusText( BSTR FAR* pbstr );
//...
BSTR bstrStatus;
pBrowser->get_StatusText( &bstrStatus );

// shows using the Win32 function
// to freee the memory for the string:
::SysFreeString( bstrStatus );
```

- When you implement a function that returns a **BSTR**, allocate the string but do not free it. The receiving the function releases the memory. For example:

```
// Example shows using MFC's
// CString::AllocSysString

//...
HRESULT CMyClass::get_StatusText( BSTR * pbstr )
{

   try
   {
      //m_str is a CString in your class
      *pbstr = m_str.AllocSysString( );
      }
   catch (...)
   {
      return E_OUTOFMEMORY;
   }

// The client is now responsible for freeing pbstr.
return( S_OK );
```

```
}
//...
```

See Also

Reference

CStringT::AllocSysString
SysAllocString
SysFreeString

Other Resources

Strings (ATL/MFC)

CString Exception Cleanup

In previous versions of MFC, it was important that you clean up CString objects after use. With MFC version 3.0 and later, explicit cleanup is no longer necessary.

Under the C++ exception handling mechanism that MFC now uses, you do not have to worry about cleanup after an exception. For a description of how C++ "unwinds" the stack after an exception is caught, see the try, catch, and throw statements. Even if you use the MFC **TRY**/**CATCH** macros instead of the C++ keywords **try** and **catch**, MFC uses the C++ exception mechanism underneath, so you still do not need to clean up explicitly.

See Also

Concepts

Exception Handling in MFC

Other Resources

Strings (ATL/MFC)

CString Argument Passing

This article explains how to pass CString objects to functions and how to return **CString** objects from functions.

CString Argument-Passing Conventions

When you define a class interface, you must determine the argument-passing convention for your member functions. There are some standard rules for passing and returning **CString** objects. If you follow the rules described in Strings as Function Inputs and Strings as Function Outputs, you will have efficient, correct code.

Strings as Function Inputs

If a string is an input to a function, in most cases it is best to declare the string function parameter as **LPCTSTR**. Convert to a **CString** object as necessary within the function using constructors and assignment operators. If the string contents are to be changed by a function, declare the parameter as a nonconstant **CString** reference (**CString&**).

Strings as Function Outputs

Normally you can return **CString** objects from functions because **CString** objects follow value semantics like primitive types. To return a read-only string, use a constant **CString** reference (**const CString&**). The following example illustrates the use of **CString** parameters and return types:

```
class CName : public CObject
{
private:
    CString m_firstName;
    char m_middleInit;
    CString m_lastName;
public:
    CName() {}
    void SetData( LPCTSTR fn, const char mi, LPCTSTR ln )
    {
        m_firstName = fn;
        m_middleInit = mi;
        m_lastName = ln;
    }
    void GetData( CString& cfn, char& mi, CString& cln )
    {
        cfn = m_firstName;
        mi = m_middleInit;
        cln = m_lastName;
    }
    CString GetLastName()
    {
        return m_lastName;
    }
};
...
CName name;
CString last, first;
TCHAR middle;
name.SetData( "John", 'Q', "Public" );
ASSERT( name.GetLastName() == "Public" );
name.GetData( first, middle, last );
ASSERT( ( first == "John" ) && ( last == "Public" ) );
...
```

See Also

Other Resources

Strings (ATL/MFC)

Unicode and Multibyte Character Set (MBCS) Support

Some international markets use languages, such as Japanese and Chinese, with large character sets. To support programming for these markets, the Microsoft Foundation Class Library (MFC) is enabled for two different approaches to handling large character sets:

- Unicode
- Multibyte Character Sets (MBCS)

MFC Support for Unicode Strings

The entire class library is conditionally enabled for Unicode characters and strings. In particular, class CString is Unicode-enabled.

Note
The Unicode versions of the MFC libraries are not copied to your hard disk unless you select them during a Custom installation. They are not copied during other types of installation. If you attempt to build or run an MFC Unicode application without the MFC Unicode files, you may get errors.

To copy the files to your hard disk, rerun Setup and click **Add/Remove Features**. Click **Language Tools**, click **Visual C++**, and click **Visual C++ Class & Template Libraries**, and select both **ATL MFC Shared Libraries Unicode** and **ATL MFC Static Libraries Unicode**. This will copy the following files to your hard drive:

UAFXCW.LIB	UAFXCW.PDB	UAFXCWD.LIB	UAFXCWD.PDB
MFCxxU.LIB	MFCxxU.PDB	MFCxxU.DLL	MFCxxUD.LIB
MFCxxUD.PDB	MFCxxUD.DLL	MFCSxxU.LIB	MFCSxxU.PDB
MFCSxxUD.LIB	MFCSxxUD.PDB	MFCMxxU.LIB	MFCMxxU.PDB
MFCMxxU.DLL	MFCMxxUD.LIB	MFCMxxUD.PDB	MFCMxxUD.DLL

where *xx* represents the version number of the file; for example, '80' represents version 8.0.

CString is based on the **TCHAR** data type. If the symbol **_UNICODE** is defined for a build of your program, **TCHAR** is defined as type **wchar_t**, a 16-bit character encoding type; otherwise, it is defined as **char**, the normal 8-bit character encoding. Under Unicode, then, **CString**s are composed of 16-bit characters. Without Unicode, they are composed of characters of type **char**.

To complete Unicode programming of your application, you must also:

- Use the **_T** macro to conditionally code literal strings to be portable to Unicode.
- When you pass strings, pay attention to whether function arguments require a length in characters or a length in bytes. The difference is important if you're using Unicode strings.
- Use portable versions of the C run-time string-handling functions.
- Use the following data types for characters and character pointers:
    - **TCHAR** Where you would use **char**.
    - **LPTSTR** Where you would use **char***.
    - **LPCTSTR** Where you would use **const char***. **CString** provides the **operator LPCTSTR** to convert between **CString** and **LPCTSTR**.

CString also supplies Unicode-aware constructors, assignment operators, and comparison operators.

For related information on Unicode programming, see Unicode and MBCS and Unicode Topics. The *Run-Time Library Reference* defines portable versions of all its string-handling functions. See the category Internationalization.

MFC Support for MBCS Strings

The class library is also enabled for multibyte character sets — specifically for double-byte character sets (DBCS).

Under this scheme, a character can be either one or two bytes wide. If it is two bytes wide, its first byte is a special "lead byte," chosen from a particular range depending on which code page is in use. Taken together, the lead and "trail bytes" specify a unique character encoding.

If the symbol **_MBCS** is defined for a build of your program, type **TCHAR**, on which **CString** is based, maps to **char**. It's up to you to determine which bytes in a **CString** are lead bytes and which are trail bytes. The C run-time library supplies functions to help you determine this.

Under DBCS, a given string can contain all single-byte ANSI characters, all double-byte characters, or a combination of the two. These possibilities require special care in parsing strings, including **CString** objects.

Note
Unicode string serialization in MFC can read both Unicode and MBCS strings regardless of which version of the application y ou are running. Because of this, your data files are portable between Unicode and MBCS versions of your program.

CString member functions use special "generic text" versions of the C run-time functions they call, or they use Unicode-aware functions such as **lstrlen** or **lstrcpy**. Thus, for example, if a **CString** function would normally call **strcmp**, it calls the corresponding generic-text function **_tcscmp** instead. Depending on how the symbols **_MBCS** and **_UNICODE** are defined, **_tcscmp** maps as follows:

_MBCS defined	**_mbscmp**
_UNICODE defined	**wcscmp**
Neither symbol defined	**strcmp**

Note
The symbols **_MBCS** and **_UNICODE** are mutually exclusive.

Generic-text function mappings for all of the run-time string-handling routines are detailed in the *Run-Time Library Reference*. See the category Internationalization.

Similarly, **CString** member functions are implemented using "generic" data type mappings. To enable both MBCS and Unicode, MFC uses **TCHAR** for **char**, **LPTSTR** for **char***, and **LPCTSTR** for **const char***. These result in the correct mappings for either MBCS or Unicode.

See Also

Other Resources

Strings (ATL/MFC)

Classes Shared by MFC and ATL

The following table lists the classes shared between MFC and ATL.

Class	Description	Header file
CFileTime	Provides methods for managing the date and time values associated with a file.	atltime.h
CFileTimeSpan	Provides methods for managing relative date and time values associated with a file.	atltime.h
CFixedStringT	Represents a string object with a fixed character buffer.	cstringt.h
CImage	Provides enhanced bitmap support, including the ability to load and save images in JPEG, GIF, BMP, and Portable Network Graphics (PNG) formats.	atlimage.h
COleDateTime	Encapsulates the **DATE** data type used in OLE automation.	atlcomtime.h
COleDateTimeSpan	Represents a relative time, a time span.	atlcomtime.h
CPoint	A class similar to the Windows POINT structure that also includes member functions to manipulate **CPoint** and **POINT** structures.	atltypes.h
CRect	A class similar to a Windows RECT structure that also includes member functions to manipulate **CRect** objects and Windows **RECT** structures.	atltypes.h
CSimpleStringT	Represents a **CSimpleStringT** object.	atlsimpstr.h
CSize	A class similar to the Windows SIZE structure, which implements a relative coordinate or position.	atltypes.h
CStrBufT	Provides automatic resource cleanup for **GetBuffer** and **ReleaseBuffer** calls on a existing **CStringT** object.	atlsimpstr.h
CStringData	Represents the data of a string object.	atlsimpstr.h
CStringT	Represents a **CStringT** object.	cstringt.h (MFC dependent) atlstr.h (MFC independent)
CTime	Represents an absolute time and date.	atltime.h
CTimeSpan	An amount of time, which is internally stored as the number of seconds in the time span.	atltime.h
IAtlStringMgr	Represents the interface to a **CStringT** memory manager.	atlsimpstr.h

See Also

Other Resources

ATL/MFC Shared Classes

CFileTime Class

This class provides methods for managing the date and time values associated with a file.

```
class CFileTime :
   public FILETIME
```

Remarks

This class provides methods for managing the date and time values associated with the creation, access and modification of files. The methods and data of this class are frequently used in conjunction with **CFileTimeSpan** objects, which deal with relative time values.

The date and time value is stored as a 64-bit value representing the number of 100-nanosecond intervals since January 1, 1601. This is the Coordinated Universal Time (UTC) format.

The following static const member variables are provided to simplify calculations:

Member variable	Number of 100-nanosecond intervals
Millisecond	10,000
Second	Millisecond * 1,000
Minute	Second * 60
Hour	Minute * 60
Day	Hour * 24
Week	Day * 7

Note Not all file systems can record creation and last access time and not all file systems record them in the same manner. For example, on the Windows NT FAT file system, create time has a resolution of 10 milliseconds, write time has a resolution of 2 seconds, and access time has a resolution of 1 day (the access date). On NTFS, access time has a resolution of 1 hour. Furthermore, FAT records times on disk in local time, but NTFS records times on disk in UTC. For more information, see File Times.

Requirements

Header: atltime.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

FILETIME

CFileTimeSpan Class

Hierarchy Chart

Other Resources

CFileTime Members

ATL/MFC Shared Classes

CFileTime Members

Static Functions

GetCurrentTime	Call this static function to retrieve a **CFileTime** object that represents the current system date and time.

Methods

CFileTime	The constructor.
GetTime	Call this method to retrieve the time from the **CFileTime** object.
LocalToUTC	Call this method to convert a local file time to a file time based on the Coordinated Universal Time (UTC).
SetTime	Call this method to set the date and time stored by the **CFileTime** object.
UTCToLocal	Call this method to convert time based on the Coordinated Universal Time (UTC) to local file time.

Operators

operator -	This operator is used to perform subtraction on a **CFileTime** or **CFileTimeSpan** object.
operator !=	This operator compares two **CFileTime** objects for inequality.
operator +	This operator is used to perform addition on a **CFileTimeSpan** object.
operator +=	This operator is used to perform addition on a **CFileTimeSpan** object and assign the result to the current object.
operator <	This operator compares two **CFileTime** objects to determine the lesser.
operator <=	This operator compares two **CFileTime** objects to determine equality or the lesser.
operator =	The assignment operator.
operator -=	This operator is used to perform subtraction on a **CFileTimeSpan** object and assign the result to the current obje ct.
operator ==	This operator compares two **CFileTime** objects for equality.
operator >	This operator compares two **CFileTime** objects to determine the larger.
operator >=	This operator compares two **CFileTime** objects to determine equality or the larger.

Static Data Members

Day	A static data member storing the number of 100-nanosecond intervals that make up one day.
Hour	A static data member storing the number of 100-nanosecond intervals that make up one hour.
Millisecond	A static data member storing the number of 100-nanosecond intervals that make up one millisecond.
Minute	A static data member storing the number of 100-nanosecond intervals that make up one minute.
Second	A static data member storing the number of 100-nanosecond intervals that make up one second.

Week	A static data member storing the number of 100-nanosecond intervals that make up one week.

See Also

Reference

CFileTime Class

CFileTimeSpan Class

CFileTime Static Functions

For information about the static functions in **CFileTime**, see CFileTime Members.

CFileTime::GetCurrentTime

Call this static function to retrieve a **CFileTime** object that represents the current system date and time.

```
static CFileTime GetCurrentTime( ) throw( );
```

Return Value

Returns the current system date and time in Coordinated Universal Time (UTC) format.

Example

```
// Retrieve the current time
CFileTime myFT;
myFT = CFileTime::GetCurrentTime();
```

See Also

Reference

CFileTime Class

time, _time32, _time64

Other Resources

CFileTime Members

CFileTime Methods

For information about the methods in **CFileTime**, see CFileTime Members.

CFileTime::CFileTime

The constructor.

```
CFileTime( ) throw( );
CFileTime(
   const FILETIME& ft
) throw( );
CFileTime(
   ULONGLONG nTime
) throw( );
```

Parameters

ft

A FILETIME structure.

nTime

The date and time expressed as a 64-bit value.

Remarks

The **CFileTime** object can be created using an existing date and time from a **FILETIME** structure, or expressed as a 64-bit value (in local or Coordinated Universal Time (UTC) time formats). The default constructor sets the time to 0.

See Also

Reference

CFileTime Class

Other Resources

CFileTime Members

CFileTime::GetTime

Call this method to retrieve the time from the **CFileTime** object.

```
ULONGLONG GetTime( ) const throw( );
```

Return Value

Returns the date and time as a 64-bit number, which may be in either local or Coordinated Universal Time (UTC) format.

See Also

Reference

CFileTime Class

CFileTime::SetTime

Other Resources

CFileTime Members

CFileTime::LocalToUTC

Call this method to convert a local file time to a file time based on the Coordinated Universal Time (UTC).

```
CFileTime LocalToUTC( ) const throw( );
```

Return Value

Returns a **CFileTime** object containing the time in UTC format.

Example

See the example for CFileTime::UTCToLocal.

See Also

Reference

CFileTime Class

CFileTime::UTCToLocal

Other Resources

CFileTime Members

CFileTime::SetTime

Call this method to set the date and time stored by the **CFileTime** object.

```
void SetTime(
   ULONGLONG nTime
) throw( );
```

Parameters

nTime

The 64-bit value representing the date and time, in either local or Coordinated Universal Time (UTC) format.

See Also

Reference

CFileTime Class

CFileTime::GetTime

Other Resources

CFileTime Members

CFileTime::UTCToLocal

Call this method to convert time based on the Coordinated Universal Time (UTC) to local file time.

```
CFileTime UTCToLocal( ) const throw( );
```

Return Value

Returns a **CFileTime** object containing the time in local file time format.

Example

```
// Convert a UTC time to local file time format
CFileTime myUTC_FT, myL_FT;
// Get system time (in UTC format)
myUTC_FT = CFileTime::GetCurrentTime();
// Convert to local file time
myL_FT = myUTC_FT.UTCToLocal();
```

See Also

Reference

CFileTime Class

CFileTime::LocalToUTC

Other Resources

CFileTime Members

CFileTime Operators

For information about the operators in **CFileTime**, see CFileTime Members.

CFileTime::operator -

This operator is used to perform subtraction on a **CFileTime** or **CFileTimeSpan** object.

```
CFileTime operator -(
   CFileTimeSpan span
) const throw( );
CFileTimeSpan operator -(
   CFileTime ft
) const throw( );
```

Parameters

span

A **CFileTimeSpan** object.

ft

A **CFileTime** object.

Return Value

Returns a **CFileTime** object or a **CFileTimeSpan** object representing the result of the time difference between the two objects.

See Also

Reference

CFileTime Class

CFileTimeSpan Class

Other Resources

CFileTime Members

CFileTime::operator !=

This operator compares two **CFileTime** objects for inequality.

```
bool operator!=(
   CFileTime ft
) const throw( );
```

Parameters

ft

The **CFileTime** object to be compared.

Return Value

Returns **true** if the item being compared is not equal to the **CFileTime** object, otherwise **false**.

See Also

Reference

CFileTime Class

Other Resources

CFileTime Members

CFileTime::operator +

This operator is used to perform addition on a **CFileTimeSpan** object.

```
CFileTime operator +(
   CFileTimeSpan span
) const throw( );
```

Parameters

span

A **CFileTimeSpan** object.

Return Value

Returns a **CFileTime** object representing the result of the original time plus a relative time.

See Also

Reference

CFileTime Class

CFileTimeSpan Class

Other Resources

CFileTime Members

CFileTime::operator +=

This operator is used to perform addition on a **CFileTimeSpan** object and assign the result to the current object.

```
CFileTime& operator +=(
   CFileTimeSpan span
) throw( );
```

Parameters

span

A **CFileTimeSpan** object.

Return Value

Returns the updated **CFileTime** object, representing the result of the original time plus a relative time.

See Also

Reference

CFileTime Class

CFileTimeSpan Class

Other Resources

CFileTime Members

CFileTime::operator <

This operator compares two **CFileTime** objects to determine the lesser.

```
bool operator<(
   CFileTime ft
) const throw( );
```

Parameters

ft

The **CFileTime** object to be compared.

Return Value

Returns **true** if the first object is less (earlier in time) than the second, **false** otherwise.

Example

```
{
    // ...

    // Test for one time less than another
    // Declare the CFileType objects
    CFileType myFT1, myFT2;

    // Obtain the first time value
    myFT1 = CFileTime::GetCurrentTime();

    // Pause for a moment...
    Sleep(1000);

    // Obtain the second time value
    myFT2 = CFileTime::GetCurrentTime();

    // Perform the comparison
    if (myFT1 < myFT2)
        printf_s("Time is going in the correct direction.\n");
    else
        printf("Oh dear. Time is going backwards.\n");
}
```

See Also

Reference

CFileTime Class

Other Resources

CFileTime Members

CFileTime::operator <=

This operator compares two **CFileTime** objects to determine equality or the lesser.

```
bool operator<=(
   CFileTime ft
) const throw( );
```

Parameters

ft

The **CFileTime** object to be compared.

Return Value

Returns **true** if the first object is less than (earlier in time) or equal to the second, otherwise **false**.

See Also

Reference

CFileTime Class

Other Resources

CFileTime Members

CFileTime::operator =

The assignment operator.

```
CFileTime& operator =(
   const FILETIME& ft
) throw( );
```

Parameters

ft

A **CFileTime** object containing the new time and date.

Return Value

Returns the updated **CFileTime** object.

See Also

Reference

CFileTime Class

Other Resources

CFileTime Members

CFileTime::operator -=

This operator is used to perform subtraction on a **CFileTimeSpan** object and assign the result to the current object.

```
CFileTime& operator -=(
   CFileTimeSpan span
) throw( );
```

Parameters

span

A **CFileTimeSpan** object containing the relative time to subtract.

Return Value

Returns the updated **CFileTime** object.

See Also

Reference

CFileTime Class

CFileTimeSpan Class

Other Resources

CFileTime Members

CFileTime::operator ==

This operator compares two **CFileTime** objects for equality.

```
bool operator ==(
   CFileTime ft
) const throw( );
```

Parameters

ft

The **CFileTime** object to compare.

Return Value

Returns **true** if the objects are equal, otherwise **false**.

See Also

Reference

CFileTime Class

Other Resources

CFileTime Members

CFileTime::operator >

This operator compares two **CFileTime** objects to determine the larger.

```
bool operator >(
   CFileTime ft
) const throw( );
```

Parameters

ft

The **CFileTime** object to be compared.

Return Value

Returns **true** if the first object is greater than (later in time) than the second, otherwise **false**.

See Also

Reference

CFileTime Class

Other Resources

CFileTime Members

CFileTime::operator >=

This operator compares two **CFileTime** objects to determine equality or the larger.

```
bool operator >=(
   CFileTime ft
) const throw( );
```

Parameters

ft

The **CFileTime** object to be compared.

Return Value

Returns **true** if the first object is greater than (later in time) or equal to the second, otherwise **false**.

See Also

Reference

CFileTime Class

Other Resources

CFileTime Members

CFileTime Static Data Members

For information about the static data in **CFileTime**, see CFileTime Members.

CFileTime::Day

A static data member storing the number of 100-nanosecond intervals that make up one day.

```
static const ULONGLONG Day = Hour* 24;
```

Example

See the example for CFileTime::Millisecond.

See Also

Reference

CFileTime Class

Other Resources

CFileTime Members

CFileTime::Hour

A static data member storing the number of 100-nanosecond intervals that make up one hour.

```
static const ULONGLONG Hour = Minute* 60;
```

Example

See the example for CFileTime::Millisecond.

See Also

Reference

CFileTime Class

Other Resources

CFileTime Members

CFileTime::Millisecond

A static data member storing the number of 100-nanosecond intervals that make up one millisecond.

```
static const ULONGLONG Millisecond = 10000;
```

Example

```
// cfiletime_millisecond.cpp
// compile with: /W0
// Calculate the difference between two times
#include <string.h>
#include <afx.h>

int main(int argc, char* argv[])
{
    CFileTime myFT1, myFT2;
    CFileTimeSpan myFTS;

    // Get the first time
    myFT1=CFileTime::GetCurrentTime();

    // Pause for a moment
    Sleep(500);

    // Get the second time
    myFT2=CFileTime::GetCurrentTime();

    // Calculate the time difference
    myFTS = myFT2 - myFT1;

    // Measure the difference
    if (myFTS.GetTimeSpan() < CFileTime::Minute)
        printf_s("Less than a minute passed\n");
    else
        printf_s("A minute or more passed\n");

    if (myFTS.GetTimeSpan() < CFileTime::Second)
        printf_s("Less than a second passed\n");
    else
        printf_s("A second or more passed\n");

    if (myFTS.GetTimeSpan() < CFileTime::Millisecond)
        printf_s("Less than a millisecond passed\n");
    else
        printf_s("A millisecond or more passed\n");

   return 0;
}
```

Output

```
Less than a minute passed
Less than a second passed
A millisecond or more passed
```

See Also

Reference

CFileTime Class

Other Resources

CFileTime Members

CFileTime::Minute

A static data member storing the number of 100-nanosecond intervals that make up one minute.

```
static const ULONGLONG Minute = Second* 60;
```

Example

See the example for CFileTime::Millisecond.

See Also

Reference

CFileTime Class

Other Resources

CFileTime Members

CFileTime::Second

A static data member storing the number of 100-nanosecond intervals that make up one day.

```
static const ULONGLONG Second = Millisecond* 1000;
```

Example

See the example for CFileTime::Millisecond.

See Also

Reference

CFileTime Class

Other Resources

CFileTime Members

CFileTime::Week

A static data member storing the number of 100-nanosecond intervals that make up one week.

```
static const ULONGLONG Week = Day* 7;
```

Example

See the example for CFileTime::Millisecond.

See Also

Reference

CFileTime Class

Other Resources

CFileTime Members

CFileTimeSpan Class

This class provides methods for managing relative date and time values associated with a file.

```
class CFileTimeSpan
```

Remarks

This class provides methods for managing relative periods of time often encountered when performing operations concerning when a file was created, last accessed or last modified. The methods of this class are frequently used in conjunction with CFileTime class objects.

Example

See the example for CFileTime::Millisecond.

Requirements

Header: atltime.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

FILETIME

CFileTime Class

Hierarchy Chart

Other Resources

CFileTimeSpan Members

ATL/MFC Shared Classes

CFileTimeSpan Members

Methods

CFileTimeSpan	The constructor.
GetTimeSpan	Call this method to retrieve the time span from the **CFileTimeSpan** object.
SetTimeSpan	Call this method to set the time span of the **CFileTimeSpan** object.

Operators

operator -	Performs subtraction on a **CFileTimeSpan** object.
operator !=	Compares two **CFileTimeSpan** objects for inequality.
operator +	Performs addition on a **CFileTimeSpan** object.
operator +=	Performs addition on a **CFileTimeSpan** object and assign the result to the current object.
operator <	Compares two **CFileTimeSpan** objects to determine the lesser.
operator <=	Compares two **CFileTimeSpan** objects to determine equality or the lesser.
operator =	The assignment operator.
operator -=	Performs subtraction on a **CFileTimeSpan** object and assign the result to the current object.
operator ==	Compares two **CFileTimeSpan** objects for equality.
operator >	Compares two **CFileTimeSpan** objects to determine the larger.
operator >=	Compares two **CFileTimeSpan** objects to determine equality or the larger.

See Also

Reference

CFileTimeSpan Class

CFileTimeSpan Methods

For information about the methods in **CFileTimeSpan**, see CFileTimeSpan Members.

CFileTimeSpan::CFileTimeSpan

The constructor.

```
CFileTimeSpan( ) throw( );
CFileTimeSpan(
   const CFileTimeSpan& span
) throw( );
CFileTimeSpan(
   LONGLONG nSpan
) throw( );
```

Parameters

span

An existing **CFileTimeSpan** object.

nSpan

A period of time.

Remarks

The **CFileTimeSpan** object can be created using an existing **CFileTimeSpan** object, or expressed as a 64-bit value. The default constructor sets the time span to 0.

See Also

Reference

CFileTimeSpan Class

Other Resources

CFileTimeSpan Members

CFileTimeSpan::GetTimeSpan

Call this method to retrieve the time span from the **CFileTimeSpan** object.

```
LONGLONG GetTimeSpan( ) const throw( );
```

Return Value

Returns the time span.

See Also

Reference

CFileTimeSpan Class

CFileTimeSpan::SetTimeSpan

Other Resources

CFileTimeSpan Members

CFileTimeSpan::SetTimeSpan

Call this method to set the time span of the **CFileTimeSpan** object.

```
void SetTimeSpan(
   LONGLONG nSpan
) throw( );
```

Parameters

nSpan

The new value for the time span.

See Also

Reference

CFileTimeSpan Class

CFileTimeSpan::GetTimeSpan

Other Resources

CFileTimeSpan Members

CFileTimeSpan Operators

For information about the methods in **CFileTimeSpan**, see CFileTimeSpan Members.

CFileTimeSpan::operator -

Performs subtraction on a **CFileTimeSpan** object.

```
CFileTimeSpan operator -(
   CFileTimeSpan span
) const throw( );
```

Parameters

span

A **CFileTimeSpan** object.

Return Value

Returns a **CFileTimeSpan** object representing the result of the difference between two time spans.

See Also

Reference

CFileTimeSpan Class

Other Resources

CFileTimeSpan Members

CFileTimeSpan::operator !=

Compares two **CFileTimeSpan** objects for inequality.

```
bool operator!=(
   CFileTimeSpan span
) const throw( );
```

Parameters

span

The **CFileTimeSpan** object to be compared.

Return Value

Returns **true** if the item being compared is not equal to the **CFileTimeSpan** object; otherwise **false**.

See Also

Reference

CFileTimeSpan Class

Other Resources

CFileTimeSpan Members

CFileTimeSpan::operator +

Performs addition on a **CFileTimeSpan** object.

```
CFileTimeSpan operator +(
   CFileTimeSpan span
) const throw( );
```

Parameters

span

A **CFileTimeSpan** object.

Return Value

Returns a **CFileTimeSpan** object containing the sum of the two time spans.

See Also

Reference

CFileTimeSpan Class

Other Resources

CFileTimeSpan Members

CFileTimeSpan::operator +=

Performs addition on a **CFileTimeSpan** object and assigns the result to the current object.

```
CFileTimeSpan& operator +=(
   CFileTimeSpan span
) throw( );
```

Parameters

span

A **CFileTimeSpan** object.

Return Value

Returns the updated **CFileTimeSpan** object containing the sum of the two time spans.

See Also

Reference

CFileTimeSpan Class

Other Resources

CFileTimeSpan Members

CFileTimeSpan::operator <

Compares two **CFileTimeSpan** objects to determine the lesser.

```
bool operator<(
   CFileTimeSpan span
) const throw( );
```

Parameters

span

The **CFileTimeSpan** object to be compared.

Return Value

Returns **true** if the first object is less (that is, represents a shorter time period) than the second, otherwise **false**.

See Also

Reference

CFileTimeSpan Class

Other Resources

CFileTimeSpan Members

CFileTimeSpan::operator <=

Compares two **CFileTimeSpan** objects to determine equality or the lesser.

```
bool operator<=(
   CFileTimeSpan span
) const throw( );
```

Parameters

span

The **CFileTimeSpan** object to be compared.

Return Value

Returns **true** if the first object is less than (that is, represents a shorter time period) or equal to the second, otherwise **false**.

See Also

Reference

CFileTimeSpan Class

Other Resources

CFileTimeSpan Members

CFileTimeSpan::operator =

The assignment operator.

```
CFileTimeSpan& operator =(
   const CFileTimeSpan& span
) throw( );
```

Parameters

span

A **CFileTimeSpan** object.

Return Value

Returns the updated **CFileTimeSpan** object.

See Also

Reference

CFileTimeSpan Class

Other Resources

CFileTimeSpan Members

CFileTimeSpan::operator -=

Performs subtraction on a **CFileTimeSpan** object and assigns the result to the current object.

```
CFileTimeSpan& operator -=(
   CFileTimeSpan span
) throw( );
```

Parameters

span

A **CFileTimeSpan** object.

Return Value

Returns the updated **CFileTimeSpan** object.

See Also

Reference

CFileTimeSpan Class

Other Resources

CFileTimeSpan Members

CFileTimeSpan::operator ==

Compares two **CFileTimeSpan** objects for equality.

```
bool operator ==(
   CFileTimeSpan span
) const throw( );
```

Parameters

span

The **CFileTimeSpan** object to be compared.

Return Value

Returns **true** if the objects are equal, otherwise **false**.

See Also

Reference

CFileTimeSpan Class

Other Resources

CFileTimeSpan Members

CFileTimeSpan::operator >

Compares two **CFileTimeSpan** objects to determine the larger.

```
bool operator >(
   CFileTimeSpan span
) const throw( );
```

Parameters

span

The **CFileTimeSpan** object to be compared.

Return Value

Returns **true** if the first object is greater than (that is, represents a longer time period) than the second, otherwise **false**.

See Also

Reference

CFileTimeSpan Class

Other Resources

CFileTimeSpan Members

CFileTimeSpan::operator >=

Compares two **CFileTimeSpan** objects to determine equality or the larger.

```
bool operator >=(
   CFileTimeSpan span
) const throw( );
```

Parameters

span

The **CFileTimeSpan** object to be compared.

Return Value

Returns **true** if the first object is greater than (that is, represents a longer time period) or equal to the second, otherwise **false**.

See Also

Reference

CFileTimeSpan Class

Other Resources

CFileTimeSpan Members

CFixedStringT Class

This class represents a string object with a fixed character buffer.

```
template< class StringType, int t_nChars >
class CFixedStringT : private CFixedStringMgr, public StringType
```

Parameters

StringType

Used as the base class for the fixed string object and can be any **CStringT**-based type. Some examples include **CString**, **CStringA**, and **CStringW**.

t_nChars

The number of characters stored in the buffer.

Remarks

This class is an example of a custom string class based on **CStringT**. Although quite similar, the two classes differ in implementation. The major differences between **CFixedStringT** and **CStringT** are:

- The initial character buffer is allocated as part of the object and has size *t_nChars*. This allows the **CFixedString** object to occupy a contiguous memory chunk for performance purposes. However, if the contents of a **CFixedStringT** object grows beyond *t_nChars*, the buffer is allocated dynamically.
- The character buffer for a **CFixedStringT** object is always the same length (*t_nChars*). There is no limitation on buffer size for **CStringT** objects.
- The memory manager for **CFixedStringT** is customized such that sharing of a CStringData object between two or more **CFixedStringT** objectsis not allowed. **CStringT** objects do not have this limitation.

For more information on the customization of **CFixedStringT** and memory management for string objects in general, see Memory Management and CStringT.

Requirements

Header: cstringt.h

See Also

Reference

CStringT Class

Hierarchy Chart

Other Resources

CFixedStringT Members

ATL/MFC Shared Classes

CFixedStringT Members

Methods

CFixedStringT	The constructor for the string object.

Operators

operator =	Assigns a new value to a **CFixedStringT** object.

See Also

Reference

CFixedStringT Class

CFixedStringT Methods

For information about the methods in **CFixedStringT**, see CFixedStringT Members

CFixedStringT::CFixedStringT

Constructs a **CFixedStringT** object.

```
CFixedStringT( ) throw( );
explicit CFixedStringT(
   IAtlStringMgr* pStringMgr
) throw( );
CFixedStringT(
   const CFixedStringT< StringType, t_nChars >& str
);
CFixedStringT(
   const StringType& str
);
CFixedStringT(
   const StringType::XCHAR* psz
);
explicit CFixedStringT(
   const StringType::YCHAR* psz
);
explicit CFixedStringT(
   const unsigned char* psz
);
```

Parameters

psz

A null-terminated string to be copied into this **CFixedStringT** object.

str

An existing **CFixedStringT** object to be copied into this **CFixedStringT** object.

pStringMgr

A pointer to the memory manager of the **CFixedStringT** object. For more information on **IAtlStringMgr** and memory management for **CFixedStringT**, see Memory Management and CStringT.

Remarks

Because the constructors copy the input data into new allocated storage, you should be aware that memory exceptions may result. Note that some of these constructors act as conversion functions.

See Also

Reference

CFixedStringT Class

Other Resources

CFixedStringT Members

CFixedStringT Operators

For information about the operators in **CFixedStringT**, see CFixedStringT Members

CFixedStringT::operator =

Reinitializes an existing **CFixedStringT** object with new data.

```
CFixedStringT< StringType, t_nChars >& operator =(
   const CFixedStringT< StringType, t_nChars >& str
);
CFixedStringT< StringType, t_nChars >& operator =(
   const char* psz
);
CFixedStringT< StringType, t_nChars >& operator =(
   const wchar_t* psz
);
CFixedStringT< StringType, t_nChars >& operator =(
   const unsigned char* psz
);
CFixedStringT< StringType, t_nChars >& operator =(
   const StringType& str
);
```

Parameters

str

A null-terminated string to be copied into this **CFixedStringT** object.

psz

An existing **CFixedStringT** to be copied into this **CFixedStringT** object.

Remarks

You should be aware that memory exceptions may occur whenever you use the assignment operator because new storage is often allocated to hold the resulting **CFixedStringT** object.

See Also

Reference

CFixedStringT Class

Other Resources

CFixedStringT Members

CImage Class

CImage provides enhanced bitmap support, including the ability to load and save images in JPEG, GIF, BMP, and Portable Network Graphics (PNG) formats.

```
class CImage
```

Remarks

CImage takes bitmaps that are either device-independent bitmap (DIB) sections or not; however, you can use Create or CImage::Load with only DIB sections. You can attach a non-DIB section bitmap to a **CImage** object using Attach, but then you cannot use the following **CImage** methods, which support only DIB section bitmaps:

- GetBits
- GetColorTable
- GetMaxColorTableEntries
- GetPitch
- GetPixelAddress
- IsIndexed
- SetColorTable

To determine if an attached bitmap is a DIB section, call IsDibSection.

Note

Note In Visual Studio .NET 2003, this class keeps a count of the number of **CImage** objects created. Whenever the count goes to 0, the function GdiplusShutdown is automatically called to release resources used by GDI+. This ensures that any **CImage** objects created directly or indirectly by DLLs are always destroyed properly and that **GdiplusShutdown** is not called from **DllMain**.

Note

Using global **CImage** objects in a DLL is not recommended. If you need to use a global **CImage** object in a DLL, call CImage::ReleaseGDIPlus to explicitly release resources used by GDI+.

CImage cannot be selected into a new CDC. **CImage** creates its own **HDC** for the image. Because an **HBITMAP** can only be selected into one **HDC** at a time, the **HBITMAP** associated with the **CImage** cannot be selected into another **HDC**. If you need a **CDC**, retrieve the **HDC** from the **CImage** and give it to CDC::FromHandle.

Example

```
CImage image;
// Code to load/create image goes here

// Get a CDC for the image
CDC* pDC = CDC::FromHandle(image.GetDC());

// Use pDC here

image.ReleaseDC();
```

When you use **CImage** in an MFC project, note which member functions in your project expect a pointer to a CBitmap object. If you want to use **CImage** with such a function, like CMenu::AppendMenu, use CBitmap::FromHandle, pass it your **CImage HBITMAP**, and use the returned **CBitmap***.

```
CMyWnd::OnRButtonDown(int nFlags, CPoint mouse)
```

```
{
   CMenu menu;
   CImage image;

   // Code to create menu and load/create image goes here

   CBitmap* pBitmap = CBitmap::FromHandle(image.m_hBitmap);
   menu.AppendMenu(0, ID_SOMECOMMAND, pBitmap)
   menu.TrackPopupMenu(TPM_RIGHTBUTTON | TPM_LEFTALIGN, mouse.x, mouse.y, this);
}
```

For more examples, see the HttpClient Sample and the ShowImage Sample.

Through **CImage**, you have access to the actual bits of a DIB section. You can use a **CImage** object anywhere you previously used a Win32 HBITMAP or DIB section.

Note

The following **CImage** methods have limitations on their use:

Method	Limitation
PlgBlt	Works with only Windows NT 4.0 or later. Will not work on applications running on Windows 95/98 or later.
MaskBlt	Works with only Windows NT 4.0 or later. Will not work on applications running on Windows 95/98 or later.
AlphaBlend	Works with only Windows 2000, Windows 98, and later systems.
TransparentBlt	Works with only Windows 2000, Windows 98, and later systems.
Draw	Supports transparency with only Windows 2000, Windows 98, and later systems.

See CImage Limitations with Earlier Operating Systems for more detailed information about the limitations on these methods.

You can use **CImage** from either MFC or ATL.

Note

When you create a project using **CImage**, you must define **CString** before you include `atlimage.h`. If your project uses ATL without MFC, include `atlstr.h` before you include `atlimage.h`. If your project uses MFC (or if it is an ATL project with MFC s upport), include `afxstr.h` before you include `atlimage.h`.

Likewise, you must include `atlimage.h` before you include `atlimpl.cpp`. To accomplish this easily, include `atlimage.h` in you r `stdafx.h`.

Requirements

Header: atlimage.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

MMXSwarm Sample: Demonstrates CImage and Visual C++ MMX Support

SimpleImage Sample: Loads, Resizes, Converts, and Saves Images

Reference

ShowImage

Device-Independent Bitmaps

CreateDIBSection

Other Resources

CImage Members

ATL Reference

CImage Members

Operators

operator HBITMAP	Returns the Windows handle attached to the **CImage** object.

Construction

CImage	The constructor.

Operations

AlphaBlend	Displays bitmaps that have transparent or semitransparent pixels.
Attach	Attaches an **HBITMAP** to a **CImage** object. Can be used with either non-DIB section bitmaps or DIB section bitmaps.
BitBlt	Copies a bitmap from the source device context to this current device context.
Create	Creates a DIB section bitmap and attaches it to the previously constructed **CImage** object.
CreateEx	Creates a DIB section bitmap (with additional parameters) and attaches it to the previously construct ed **CImage** object.
Destroy	Detaches the bitmap from the **CImage** object and destroys the bitmap.
Detach	Detaches the bitmap from a **CImage** object.
Draw	Copies a bitmap from a source rectangle into a destination rectangle. **Draw** stretches or compresses the bitmap to fit the dimensions of the destination rectangle, if necessary, and handles alpha blendin g and transparent colors.
GetBits	Retrieves a pointer to the actual pixel values of the bitmap.
GetBPP	Retrieves the bits per pixel.
GetColorTable	Retrieves red, green, blue (RGB) color values from a range of entries in the color table.
GetDC	Retrieves the device context into which the current bitmap is selected.
GetExporterFilterString	Finds the available image formats and their descriptions.
GetImporterFilterString	Finds the available image formats and their descriptions.
GetHeight	Retrieves the height of the current image in pixels.
GetMaxColorTableEntries	Retrieves the maximum number of entries in the color table.
GetPitch	Retrieves the pitch of the current image, in bytes.
GetPixelAddress	Retrieves the address of a given pixel.
GetPixel	Retrieves the color of the pixel specified by *x* and *y*.
GetTransparentColor	Retrieves the position of the transparent color in the color table.

GetWidth	Retrieves the width of the current image in pixels.
IsDibSection	Determines if the attached bitmap is a DIB section.
IsIndexed	Indicates that a bitmap's colors are mapped to an indexed palette.
IsNull	Indicates if a source bitmap is currently loaded.
IsTransparencySupported	Indicates whether the application supports transparent bitmaps and was compiled for Windows 2000 or later.
LoadFromResource	Loads an image from the specified resource.
Load	Loads an image from the specified file.
MaskBlt	Combines the color data for the source and destination bitmaps using the specified mask and raster operation.
PlgBlt	Performs a bit-block transfer from a rectangle in a source device context into a parallelogram in a destination device context.
ReleaseDC	Releases the device context that was retrieved with CImage::GetDC.
ReleaseGDIPlus	Releases resources used by GDI+. Must be called to free resources created by a global **CImage** object.
Save	Saves an image as the specified type. **Save** cannot specify image options.
SetColorTable	Sets red, green, blue RGB) color values in a range of entries in the color table of the DIB section.
SetPixelIndexed	Sets the pixel at the specified coordinates to the color at the specified index of the palette.
SetPixelRGB	Sets the pixel at the specified coordinates to the specified red, green, blue (RGB) value.
SetPixel	Sets the pixel at the specified coordinates to the specified color.
SetTransparentColor	Sets the index of the color to be treated as transparent. Only one color in a palette can be transparent.
StretchBlt	Copies a bitmap from a source rectangle into a destination rectangle, stretching or compressing the bitmap to fit the dimensions of the destination rectangle, if necessary.
TransparentBlt	Copies a bitmap with transparent color from the source device context to this current device context.

See Also

Reference

CImage Class

CImage Methods

For information about the methods in **CImage**, see CImage Members.

CImage::AlphaBlend

Displays bitmaps that have transparent or semitransparent pixels.

```
BOOL AlphaBlend(
   HDC hDestDC,
   int xDest,
   int yDest,
   BYTE bSrcAlpha = 0xff,
   BYTE bBlendOp = AC_SRC_OVER
) const throw( );
BOOL AlphaBlend(
   HDC hDestDC,
   const POINT& pointDest,
   BYTE bSrcAlpha = 0xff,
   BYTE bBlendOp = AC_SRC_OVER
) const throw( );
BOOL AlphaBlend(
   HDC hDestDC,
   int xDest,
   int yDest,
   int nDestWidth,
   int nDestHeight,
   int xSrc,
   int ySrc,
   int nSrcWidth,
   int nSrcHeight,
   BYTE bSrcAlpha = 0xff,
   BYTE bBlendOp = AC_SRC_OVER
);
BOOL AlphaBlend(
   HDC hDestDC,
   const RECT& rectDest,
   const RECT& rectSrc,
   BYTE bSrcAlpha = 0xff,
   BYTE bBlendOp = AC_SRC_OVER
);
```

Parameters

hDestDC

Handle to the destination device context.

xDest

The x-coordinate, in logical units, of the upper left corner of the destination rectangle.

yDest

The y-coordinate, in logical units, of the upper left corner of the destination rectangle.

bSrcAlpha

An alpha transparency value to be used on the entire source bitmap. The default 0xff (255) assumes that your image is opaque, and that you want to use per-pixel alpha values only.

bBlendOp

The alpha-blending function for source and destination bitmaps, a global alpha value to be applied to the entire source bitmap, and format information for the source bitmap. The source and destination blend functions are currently limited to **AC_SRC_OVER**.

pointDest

A reference to a POINT structure that identifies the upper left corner of the destination rectangle, in logical units.

nDestWidth

The width, in logical units, of the destination rectangle.

nDestHeight

The height, in logical units, of the destination rectangle.

xSrc

The logical x-coordinate of the upper left corner of the source rectangle.

ySrc

The logical y-coordinate of the upper left corner of the source rectangle.

nSrcWidth

The width, in logical units, of the source rectangle.

nSrcHeight

The height, in logical units, of the source rectangle.

rectDest

A reference to a RECT structure, identifying the destination.

rectSrc

A reference to a **RECT** structure, identifying the source.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Alpha-blend bitmaps support color blending on a per-pixel basis.

When *bBlendOp* is set to the default of **AC_SRC_OVER**, the source bitmap is placed over the destination bitmap based on the alpha values of the source pixels.

This method is applicable to Microsoft Windows 2000, Windows 98, and later systems. See AlphaBlend in the *Platform SDK* and CImage Limitations with Earlier Operating Systems for more detailed information.

See Also

Reference

CImage Class

BLENDFUNCTION

Other Resources

CImage Members

CImage::Attach

Attaches *hBitmap* to a **CImage** object.

```
void Attach(
   HBITMAP hBitmap,
   DIBOrientation eOrientation = DIBOR_DEFAULT
) throw();
```

Parameters

hBitmap

A handle to an **HBITMAP**.

eOrientation

Specifies the orientation of the bitmap. Can be one of the following:

- **DIBOR_DEFAULT** The orientation of the bitmap is determined by the operating system. However, this may not always have the intended results on all operating systems. For more information on this, see the following Knowledge Base article (**Q186586**): PRB: GetObject() Always Returns Positive Height For DIB Sections.
- **DIBOR_BOTTOMUP** The lines of the bitmap are in reverse order. This causes CImage::GetBits to return a pointer near the end of the bitmap buffer and CImage::GetPitch to return a negative number.
- **DIBOR_TOPDOWN** The lines of the bitmap are in top to bottom order. This causes CImage::GetBits to return a pointer to the first byte of the bitmap buffer and CImage::GetPitch to return a positive number.

Remarks

The bitmap can be either a non-DIB section bitmap or a DIB section bitmap. See IsDIBSection for a list of methods that you can use only with DIB section bitmaps.

See Also

Reference

CImage Class

CImage::CImage

Other Resources

CImage Members

CImage::BitBlt

Copies a bitmap from the source device context to this current device context.

```
BOOL BitBlt(
   HDC hDestDC,
   int xDest,
   int yDest,
   DWORD dwROP = SRCCOPY
) const throw( );
BOOL BitBlt(
   HDC hDestDC,
   const POINT& pointDest,
   DWORD dwROP = SRCCOPY
) const throw( );
BOOL BitBlt(
   HDC hDestDC,
   int xDest,
   int yDest,
   int nDestWidth,
   int nDestHeight,
   int xSrc,
   int ySrc,
   DWORD dwROP = SRCCOPY
) const throw( );
BOOL BitBlt(
   HDC hDestDC,
   const RECT& rectDest,
   const POINT& pointSrc,
   DWORD dwROP = SRCCOPY
) const throw( );
```

Parameters

hDestDC

The destination **HDC**.

xDest

The logical x-coordinate of the upper left corner of the destination rectangle.

yDest

The logical y-coordinate of the upper left corner of the destination rectangle.

dwROP

The raster operation to be performed. Raster-operation codes define exactly how to combine the bits of the source, the destination, and the pattern (as defined by the currently selected brush) to form the destination. See BitBlt in the *Platform SDK* for a list of other raster-operation codes and their descriptions.

pointDest

A POINT structure indicating the upper left corner of the destination rectangle.

nDestWidth

The width, in logical units, of the destination rectangle.

nDestHeight

The height, in logical units, of the destination rectangle.

xSrc

The logical x-coordinate of the upper left corner of the source rectangle.

ySrc

The logical y-coordinate of the upper left corner of the source rectangle.

rectDest

A RECT structure indicating the destination rectangle.

pointSrc

A **POINT** structure indicating the upper left corner of the source rectangle.

Return Value

Nonzero if successful; otherwise zero.

Remarks

For more information, see BitBlt in the *Platform SDK*.

Example

See the HttpClient Sample.

See Also

Reference

CImage Class
CImage::PlgBlt
CImage::StretchBlt
CImage::MaskBlt

Other Resources

CImage Members

CImage::CImage

Constructs a **CImage** object.

```
CImage( ) throw( );
```

Remarks

Once you have constructed the object, call Create, Load, LoadFromResource, or Attach to attach a bitmap to the object.

Note In Visual Studio .NET 2003, this class keeps a count of the number of **CImage** objects created. Whenever the count goes to 0, the function GdiplusShutdown is automatically called to release resources used by GDI+. This ensures that any **CImage** objects created directly or indirectly by DLLs are always destroyed properly and that **GdiplusShutdown** is not called from DllMain.

Using global **CImage** objects in a DLL is not recommended. If you need to use a global **CImage** object in a DLL, call CImage::ReleaseGDIPlus to explicitly release resources used by GDI+.

See Also

Reference

CImage Class

Other Resources

CImage Members

CImage::Create

Creates a **CImage** bitmap and attach it to the previously constructed **CImage** object.

```
BOOL Create(
   int nWidth,
   int nHeight,
   int nBPP,
   DWORD dwFlags = 0
) throw( );
```

Parameters

nWidth

The width of the **CImage** bitmap, in pixels.

nHeight

The height of the **CImage** bitmap, in pixels. If *nHeight* is positive, the bitmap is a bottom-up DIB and its origin is the lower left corner. If *nHeight* is negative, the bitmap is a top-down DIB and its origin is the upper left corner.

nBPP

The numbers of bits per pixel in the bitmap. Usually 4, 8, 16, 24, or 32. Can be 1 for monochrome bitmaps or masks.

dwFlags

Specifies if the bitmap object has an alpha channel. Can be a combination of zero or more of the following values:

- **createAlphaChannel** Can only be used if *nBPP* is 32, and *eCompression* is **BI_RGB**. If specified, the created image has an alpha (transparency) value for each pixel, stored in the 4th byte of each pixel (unused in a non-alpha 32-bit image). This alpha channel is automatically used when calling CImage::AlphaBlend.

Note
In calls to CImage::Draw, images with an alpha channel are automatically alpha blended to the destination.

Return Value

Nonzero if successful; otherwise 0.

See Also

Reference

CImage Class
CImage::CImage
CImage::AlphaBlend
CImage::CreateEx

Other Resources

CImage Members

CImage::CreateEx

Creates a **CImage** bitmap and attach it to the previously constructed **CImage** object.

```
BOOL CreateEx(
   int nWidth,
   int nHeight,
   int nBPP,
   DWORD eCompression,
   const DWORD* pdwBitmasks = NULL,
   DWORD dwFlags = 0
) throw( );
```

Parameters

nWidth

The width of the **CImage** bitmap, in pixels.

nHeight

The height of the **CImage** bitmap, in pixels. If *nHeight* is positive, the bitmap is a bottom-up DIB and its origin is the lower left corner. If *nHeight* is negative, the bitmap is a top-down DIB and its origin is the upper left corner.

nBPP

The numbers of bits per pixel in the bitmap. Usually 4, 8, 16, 24, or 32. Can be 1 for monochrome bitmaps or masks.

eCompression

Specifies the type of compression for a compressed bottom-up bitmap (top-down DIBs cannot be compressed). Can be one of the following values:

- **BI_RGB** The format is uncompressed. Specifying this value when calling **CImage::CreateEx** is equivalent to calling **CImage::Create**.
- **BI_BITFIELDS** The format is uncompressed and the color table consists of three **DWORD** color masks that specify the red, green, and blue components, respectively, of each pixel. This is valid when used with 16- and 32-bpp bitmaps.

pdwBitfields

Only used if *eCompression* is set to **BI_BITFIELDS**, otherwise it must be **NULL**. A pointer to an array of three **DWORD** bitmasks, specifying which bits of each pixel are used for the red, green, and blue components of the color, respectively. For information on restrictions for the bitfields, see BITMAPINFOHEADER in the *Platform SDK*.

dwFlags

Specifies if the bitmap object has an alpha channel. Can be a combination of zero or more of the following values:

- **createAlphaChannel** Can only be used if *nBPP* is 32, and *eCompression* is **BI_RGB**. If specified, the created image has an alpha (transparency) value for each pixel, stored in the 4th byte of each pixel (unused in a non-alpha 32-bit image). This alpha channel is automatically used when calling CImage::AlphaBlend.

Note
In calls to CImage::Draw, images with an alpha channel are automatically alpha blended to the destination.

Return Value

TRUE if successful. Otherwise **FALSE**.

Example

The following example creates a 100x100 pixel bitmap, using 16 bits to encode each pixel. In a given 16-bit pixel, bits 0-3 encode the red component, bits 4-7 encode green, and bits 8-11 encode blue. The remaining 4 bits are unused.

```
DWORD adwBitmasks[3] = { 0x0000000f, 0x000000f0, 0x00000f00 };
image.CreateEx( 100, 100, 16, BI_BITFIELDS, adwBitmasks, 0 );
```

See Also

Reference

CImage Class

CImage::CImage

CImage::Create

CImage::AlphaBlend

Other Resources

CImage Members

CImage::Destroy

Detaches the bitmap from the **CImage** object and destroys the bitmap.

```
void Destroy( ) throw( );
```

See Also

Reference

CImage Class

CImage::Detach

Other Resources

CImage Members

CImage::Detach

Detaches a bitmap from a **CImage** object.

```
HBITMAP Detach( ) throw( );
```

Return Value

A handle to the bitmap detached, or **NULL** if no bitmap is attached.

See Also

Reference

CImage Class

CImage::Destroy

Other Resources

CImage Members

CImage::Draw

Copies a bitmap from the source device context to the current device context.

```
BOOL Draw(
   HDC hDestDC,
   int xDest,
   int yDest,
   int nDestWidth,
   int nDestHeight,
   int xSrc,
   int ySrc,
   int nSrcWidth,
   int nSrcHeight
) const throw( );
BOOL Draw(
   HDC hDestDC,
   const RECT& rectDest,
   const RECT& rectSrc
) const throw( );
BOOL Draw(
   HDC hDestDC,
   int xDest,
   int yDest
) const throw( );
BOOL Draw(
   HDC hDestDC,
   const POINT& pointDest
) const throw( );
BOOL Draw(
   HDC hDestDC,
   int xDest,
   int yDest,
   int nDestWidth,
   int nDestHeight
) const throw( );
BOOL Draw(
   HDC hDestDC,
   const RECT& rectDest
) const throw( );
```

Parameters

hDestDC

A handle to the destination device context.

xDest

The x-coordinate, in logical units, of the upper left corner of the destination rectangle.

yDest

The y-coordinate, in logical units, of the upper left corner of the destination rectangle.

nDestWidth

The width, in logical units, of the destination rectangle.

nDestHeight

The height, in logical units, of the destination rectangle.

xSrc

The x-coordinate, in logical units, of the upper left corner of the source rectangle.

ySrc

The y-coordinate, in logical units, of the upper left corner of the source rectangle.

nSrcWidth

The width, in logical units, of the source rectangle.

nSrcHeight

The height, in logical units, of the source rectangle.

rectDest

A reference to a RECT structure, identifying the destination.

rectSrc

A reference to a **RECT** structure, identifying the source.

pointDest

A reference to a POINT structure that identifies the upper left corner of the destination rectangle, in logical units.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Draw performs the same operation as StretchBlt, unless the image contains a transparent color or alpha channel. In that case, **Draw** performs the same operation as either TransparentBlt or AlphaBlend as required.

For versions of **Draw** that do not specify a source rectangle, the entire source image is the default. For the version of **Draw** that does not specify a size for the destination rectangle, the size of the source image is the default and no stretching or shrinking occurs.

Draw supports transparency only in Windows 2000 or later and Windows 98 or later. See CImage Limitations with Earlier Operating Systems for more detailed information.

See Also

Reference

CImage Class

Other Resources

CImage Members

CImage::GetBits

Retrieves a pointer to the actual bit values of a given pixel in a bitmap.

```
void* GetBits( ) throw( );
```

Return Value

A pointer to the bitmap buffer. If the bitmap is a bottom-up DIB, the pointer points near the end of the buffer. If the bitmap is a top-down DIB, the pointer points to the first byte of the buffer.

Remarks

Using this pointer, along with the value returned by GetPitch, you can locate and change individual pixels in an image.

Note
This method supports only DIB section bitmaps; consequently, you access the pixels of a **CImage** object the same way you would the pixels of a DIB section. The returned pointer points to the pixel at the location (0, 0).

See Also

Reference

CImage Class
Device-Independent Bitmaps
GetDIBits
CImage::GetBPP
CImage::GetDC

Other Resources

CImage Members

CImage::GetBPP

Retrieves the bits-per-pixel value.

```
int GetBPP( ) const throw( );
```

Return Value

The number of bits per pixel.

Remarks

This value determines the number of bits that define each pixel and the maximum number of colors in the bitmap.

The bits per pixel is usually 1, 4, 8, 16, 24, or 32. See the **biBitCount** member of BITMAPINFOHEADER in the *Platform SDK* for more information about this value.

Example

See the ShowImage Sample.

See Also

Reference

CImage Class
CImage::GetPixelAddress
CImage::GetBits
CImage::GetDC

Other Resources

CImage Members

CImage::GetColorTable

Retrieves red, green, blue (RGB) color values from a range of entries in the palette of the DIB section.

```
void GetColorTable(
   UINT iFirstColor,
   UINT nColors,
   RGBQUAD* prgbColors
) const throw( );
```

Parameters

iFirstColor

The color table index of the first entry to retrieve.

nColors

The number of color table entries to retrieve.

prgbColors

A pointer to the array of RGBQUAD structures to retrieve the color table entries.

See Also

Reference

CImage Class
CImage::SetColorTable
CImage::GetMaxColorTableEntries
GetDIBColorTable

Other Resources

CImage Members

CImage::GetDC

Retrieves the device context that currently has the image selected into it.

```
HDC GetDC( ) const throw( );
```

Return Value

A handle to a device context.

Remarks

For each call to **GetDC**, you must have a subsequent call to ReleaseDC.

See Also

Reference

CImage Class
CImage::GetBits
CImage::GetBPP

Other Resources

CImage Members

CImage::GetExporterFilterString

Finds image formats available for saving images.

```
static HRESULT GetExporterFilterString(
   CSimpleString& strExporters,
   CSimpleArray< GUID >& aguidFileTypes,
   LPCTSTR pszAllFilesDescription = NULL,
   DWORD dwExclude = excludeDefaultSave,
   TCHAR chSeparator = _T( '|' )
);
```

Parameters

strExporters

A reference to a **CSimpleString** object. See **Remarks** for more information.

aguidFileTypes

An array of GUIDs, with each element corresponding to one of the file types in the string. In the example in *pszAllFilesDescription* below, *aguidFileTypes*[0] is **GUID_NULL** and the remaining array values are the image file formats supported by the current operating system.

Note
For a complete list of constants, see Image File Format Constants in the Platform SDK.

pszAllFilesDescription

If this parameter is not **NULL**, the filter string will have one additional filter at the beginning of the list. This filter will have the current value of *pszAllFilesDescription* for its description, and accepts files of any extension supported by any other exporter in the list.

For example:

```
//First filter in the list will be titled "All Image Files", and
//will accept files with any extension supported by any exporter.
GetExporterFilterString( strExporters, aguidFileTypes, _T( "All Image Files" ));
```

dwExclude

Set of bit flags specifying which file types to exclude from the list. Allowable flags are:

- **excludeGIF** = 0x01 Excludes GIF files.
- **excludeBMP** = 0x02 Excludes BMP (Windows Bitmap) files.
- **excludeEMF** = 0x04 Excludes EMF (Enhanced Metafile) files.
- **excludeWMF** = 0x08 Excludes WMF (Windows Metafile) files.
- **excludeJPEG** = 0x10 Excludes JPEG files.
- **excludePNG** = 0x20 Excludes PNG files.
- **excludeTIFF** = 0x40 Excludes TIFF files.
- **excludeIcon** = 0x80 Excludes ICO (Windows Icon) files.
- **excludeOther** = 0x80000000 Excludes any other file type not listed above.
- **excludeDefaultLoad** = 0 For load, all file types are included by default
- **excludeDefaultSave** = **excludeIcon | excludeEMF | excludeWMF** For saving, these files are excluded by default because they usually have special requirements.

chSeparator

The separator used between the image formats. See **Remarks** for more information.

Return Value

A standard **HRESULT**.

Remarks

You can pass the resulting format string to your MFC CFileDialog object to expose the file extensions of the available image formats in the File Save As dialog box.

The parameter *strExporter* has the format:

file description0|*.ext0|filedescription1|*.ext1|...file description*n*|*.ext*n*||

where '|' is the separator character specified by *chSeparator*. For example:

```
"Bitmap format|*.bmp|JPEG format|*.jpg|GIF format|*.gif|PNG format|*.png||"
```

Use the default separator '|' if you pass this string to an MFC **CFileDialog** object. Use the null separator '\0' if you pass this string to a common File Save dialog box.

See Also

Reference

CImage Class
CImage::GetImporterFilterString
CFileDialog::m_ofn
CFileDialog::GetFileExt
OPENFILENAME
CFileDialog::SetDefExt

Other Resources

CImage Members

CImage::GetHeight

Retrieves the height, in pixels, of an image.

```
int GetHeight( ) const throw( );
```

Return Value

The height, in pixels, of an image.

Example

See the HttpClient Sample and the ShowImage Sample.

See Also

Reference

CImage Class
CImage::GetPixelAddress
CImage::GetWidth
CImage::GetPitch

Other Resources

CImage Members

CImage::GetImporterFilterString

Finds image formats available for loading images.

```
static HRESULT GetImporterFilterString(
   CSimpleString& strImporters,
   CSimpleArray< GUID >& aguidFileTypes,
   LPCTSTR pszAllFilesDescription = NULL,
   DWORD dwExclude = excludeDefaultLoad,
   TCHAR chSeparator = _T( '|' )
);
```

Parameters

strImporters

A reference to a **CSimpleString** object. See **Remarks** for more information.

aguidFileTypes

An array of GUIDs, with each element corresponding to one of the file types in the string. In the example in *pszAllFilesDescription* below, *aguidFileTypes*[0] is **GUID_NULL** with the remaining array values are the image file formats supported by the current operating system.

Note
For a complete list of constants, see Image File Format Constants in the Platform SDK.

pszAllFilesDescription

If this parameter is not **NULL**, the filter string will have one additional filter at the beginning of the list. This filter will have the current value of *pszAllFilesDescription* for its description, and accepts files of any extension supported by any other exporter in the list.

For example:

```
//First filter in the list will be titled "All Image Files", and
//will accept files with any extension supported by any importer.
GetImporterFilterString( strImporters, aguidFileTypes, _T( "All Image Files" ));
```

dwExclude

Set of bit flags specifying which file types to exclude from the list. Allowable flags are:

- **excludeGIF** = 0x01 Excludes GIF files.
- **excludeBMP** = 0x02 Excludes BMP (Windows Bitmap) files.
- **excludeEMF** = 0x04 Excludes EMF (Enhanced Metafile) files.
- **excludeWMF** = 0x08 Excludes WMF (Windows Metafile) files.
- **excludeJPEG** = 0x10 Excludes JPEG files.
- **excludePNG** = 0x20 Excludes PNG files.
- **excludeTIFF** = 0x40 Excludes TIFF files.
- **excludeIcon** = 0x80 Excludes ICO (Windows Icon) files.
- **excludeOther** = 0x80000000 Excludes any other file type not listed above.
- **excludeDefaultLoad** = 0 For load, all file types are included by default
- **excludeDefaultSave** = **excludeIcon | excludeEMF | excludeWMF** For saving, these files are excluded by default because they usually have special requirements.

chSeparator

The separator used between the image formats. See **Remarks** for more information.

Remarks

You can pass the resulting format string to your MFC CFileDialog object to expose the file extensions of the available image formats in the **File Open** dialog box.

The parameter *strImporter* has the format:

file description0|*.ext0|filedescription1|*.ext1|...file description*n*|*.ext*n*||

where '|' is the separator character specified by *chSeparator*. For example:

```
"Bitmap format|*.bmp|JPEG format|*.jpg|GIF format|*.gif|PNG format|*.png||"
```

Use the default separator '|' if you pass this string to an MFC **CFileDialog** object. Use the null separator '\0' if you pass this string to a common **File Open** dialog box.

See Also

Reference

CImage Class
CImage::GetExporterFilterString
CFileDialog::m_ofn
CFileDialog::GetFileExt
OPENFILENAME
CFileDialog::SetDefExt

Other Resources

CImage Members

CImage::GetMaxColorTableEntries

Retrieves the maximum number of entries in the color table.

```
int GetMaxColorTableEntries( ) const throw( );
```

Return Value

The number of entries in the color table.

Remarks

This method supports only DIB section bitmaps.

See Also

Reference

CImage Class

CImage::GetColorTable

Other Resources

CImage Members

CImage::GetPitch

Retrieves the pitch of an image.

```
int GetPitch( ) const throw( );
```

Return Value

The pitch of the image. If the return value is negative, the bitmap is a bottom-up DIB and its origin is the lower left corner. If the return value is positive, the bitmap is a top-down DIB and its origin is the upper left corner.

Remarks

The pitch is the distance, in bytes, between two memory addresses that represent the beginning of one bitmap line and the beginning of the next bitmap line. Because pitch is measured in bytes, the pitch of an image helps you to determine the pixel format. The pitch can also include additional memory, reserved for the bitmap.

Use **GetPitch** with GetBits to find individual pixels of an image.

Note
This method supports only DIB section bitmaps.

See Also

Reference

CImage Class
CImage::GetPixelAddress
CImage::GetWidth
CImage::GetHeight

Other Resources

CImage Members

CImage::GetPixel

Retrieves the color of the pixel at the location specified by *x* and *y*.

```
COLORREF GetPixel(
   int x,
   int y
) const throw( );
```

Parameters

x

The x-coordinate of the pixel.

y

The y-coordinate of the pixel.

Return Value

The red, green, blue (RGB) value of the pixel. If the pixel is outside of the current clipping region, the return value is **CLR_INVALID**.

See Also

Reference

CImage Class
CImage::GetPixelAddress
COLORREF

Other Resources

CImage Members

CImage::GetPixelAddress

Retrieves the exact address of a pixel.

```
void* GetPixelAddress(
   int x,
   int y
) throw( );
```

Parameters

x

The x-coordinate of the pixel.

y

The y-coordinate of the pixel.

Remarks

The address is determined according to the coordinates of a pixel, the pitch of the bitmap, and the bits per pixel.

For formats that have less than 8 bits per pixel, this method returns the address of the byte containing the pixel. For example, if your image format has 4 bits per pixel, **GetPixelAddress** returns the address of the first pixel in the byte, and you must calculate for 2 pixels per byte.

Note
This method supports only DIB section bitmaps.

See Also

Reference

CImage Class

Other Resources

CImage Members

CImage::GetTransparentColor

Retrieves the indexed location of the transparent color in the color palette.

```
LONG GetTransparentColor( ) const throw( );
```

Return Value

The index of the transparent color.

See Also

Reference

CImage Class

CImage::SetTransparentColor

Other Resources

CImage Members

CImage::GetWidth

Retrieves the width, in pixels, of an image.

```
int GetWidth( ) const throw( );
```

Return Value

The width of the bitmap, in pixels.

Example

See the HttpClient Sample and the ShowImage Sample.

See Also

Reference

CImage Class

Other Resources

CImage Members

CImage::IsDIBSection

Determines if the attached bitmap is a DIB section.

```
bool IsDIBSection( ) const throw( );
```

Return Value

true if the attached bitmap is a DIB section. Otherwise **false**.

Remarks

If the bitmap is not a DIB section, you cannot use the following **CImage** methods, which support only DIB section bitmaps:

- GetBits
- GetColorTable
- GetMaxColorTableEntries
- GetPitch
- GetPixelAddress
- IsIndexed
- SetColorTable

See Also

Reference

CImage Class

CImage::Attach

Other Resources

CImage Members

CImage::IsIndexed

Determines whether a bitmap's pixels are mapped to a color palette.

```
bool IsIndexed( ) const throw( );
```

Return Value

true if indexed; otherwise **false**.

Remarks

This method returns **true** only if the bitmap is 8-bit (256 colors) or less.

Note
This method supports only DIB section bitmaps.

See Also

Reference

CImage Class

CImage::GetColorTable

Other Resources

CImage Members

CImage::IsNull

Determines if a bitmap is currently loaded.

```
bool IsNull( ) const throw( );
```

Remarks

This method returns **True** if a bitmap is not currently loaded; otherwise **False**.

See Also

Reference

CImage Class

Other Resources

CImage Members

CImage::IsTransparencySupported

Indicates whether the application supports transparent bitmaps and was compiled for Windows 2000 or later.

```
static BOOL IsTransparencySupported( ) throw( );
```

Return Value

Nonzero if the current platform supports transparency. Otherwise 0.

Remarks

If the return value is nonzero, and transparency is supported, a call to AlphaBlend, TransparentBlt, or Draw will handle transparent colors.

If the application is compiled for use with operating systems before Windows 2000 or Windows 98, this method will always return 0, even on newer operating systems.

See CImage Limitations with Earlier Operating Systems for more detailed information.

See Also

Reference

CImage Class

CImage::GetTransparentColor

Other Resources

CImage Members

CImage::Load

Loads an image.

```
HRESULT Load(
   LPCTSTR pszFileName
) throw( );
HRESULT Load(
   IStream* pStream
) throw();
```

Parameters

pszFileName

A pointer to a string containing the name of the image file to load.

pStream

A pointer to a stream containing the name of the image file to load.

Return Value

A standard **HRESULT**.

Remarks

Loads the image specified by *pszFileName* or *pStream*.

Example

See the HttpClient Sample and the ShowImage Sample.

See Also

Reference

CImage Class

CImage::LoadFromResource

Other Resources

CImage Members

CImage::LoadFromResource

Loads an image from a resource.

```
void LoadFromResource(
   HINSTANCE hInstance,
   LPCTSTR pszResourceName
) throw( );
void LoadFromResource(
   HINSTANCE hInstance,
   UINT nIDResource
) throw( );
```

Parameters

hInstance

Handle to an instance of the module that contains the image to be loaded.

pszResourceName

A pointer to the string containing the name of the resource containing the image to load.

nIDResource

The ID of the resource to load.

See Also

Reference

CImage Class

CImage::Load

Other Resources

CImage Members

CImage::MaskBlt

Combines the color data for the source and destination bitmaps using the specified mask and raster operation.

```
BOOL MaskBlt(
   HDC hDestDC,
   int xDest,
   int yDest,
   int nDestWidth,
   int nDestHeight,
   int xSrc,
   int ySrc,
   HBITMAP hbmMask,
   int xMask,
   int yMask,
   DWORD dwROP = SRCCOPY
) const throw( );
BOOL MaskBlt(
   HDC hDestDC,
   const RECT& rectDest,
   const POINT& pointSrc,
   HBITMAP hbmMask,
   const POINT& pointMask,
   DWORD dwROP = SRCCOPY
) const throw( );
BOOL MaskBlt(
   HDC hDestDC,
   int xDest,
   int yDest,
   HBITMAP hbmMask,
   DWORD dwROP = SRCCOPY
) const throw( );
BOOL MaskBlt(
   HDC hDestDC,
   const POINT& pointDest,
   HBITMAP hbmMask,
   DWORD dwROP = SRCCOPY
) const throw( );
```

Parameters

hDestDC

The handle to the module whose executable contains the resource.

xDest

The x-coordinate, in logical units, of the upper left corner of the destination rectangle.

yDest

The y-coordinate, in logical units, of the upper left corner of the destination rectangle.

nDestWidth

The width, in logical units, of the destination rectangle and source bitmap.

nDestHeight

The height, in logical units, of the destination rectangle and source bitmap.

xSrc

The logical x-coordinate of the upper left corner of the source bitmap.

ySrc

The logical y-coordinate of the upper left corner of the source bitmap.

hbmMask

Handle to the monochrome mask bitmap combined with the color bitmap in the source device context.

xMask

The horizontal pixel offset for the mask bitmap specified by the *hbmMask* parameter.

yMask

The vertical pixel offset for the mask bitmap specified by the *hbmMask* parameter.

dwROP

Specifies both foreground and background ternary raster operation codes that the method uses to control the combination of source and destination data. The background raster operation code is stored in the high-order byte of the high-order word of this value; the foreground raster operation code is stored in the low-order byte of the high-order word of this value; the low-order word of this value is ignored, and should be zero. For a discussion of foreground and background in the context of this method, see MaskBlt in the *Platform SDK*. For a list of common raster operation codes, see BitBlt in the *Platform SDK*.

rectDest

A reference to a RECT structure, identifying the destination.

pointSrc

A POINT structure indicating the upper left corner of the source rectangle.

pointMask

A **POINT** structure indicating the upper left corner of the mask bitmap.

pointDest

A reference to a **POINT** structure that identifies the upper left corner of the destination rectangle, in logical units.

Return Value

Nonzero if successful, otherwise 0.

Remarks

This method applies to Windows NT, versions 4.0 and later only.

See MaskBlt in the *Platform SDK* and CImage Limitations with Earlier Operating Systems for more detailed information.

See Also

Reference

CImage Class
CImage::BitBlt
CImage::PlgBlt
MAKEROP4

Other Resources

CImage Members

CImage::PlgBlt

Performs a bit-block transfer from a rectangle in a source device context into a parallelogram in a destination device context.

```
BOOL PlgBlt(
   HDC hDestDC,
   const POINT* pPoints,
   HBITMAP hbmMask = NULL
) const throw( );
BOOL PlgBlt(
   HDC hDestDC,
   const POINT* pPoints,
   int xSrc,
   int ySrc,
   int nSrcWidth,
   int nSrcHeight,
   HBITMAP hbmMask = NULL,
   int xMask = 0,
   int yMask = 0
) const throw( );
BOOL PlgBlt(
   HDC hDestDC,
   const POINT* pPoints,
   const RECT& rectSrc,
   HBITMAP hbmMask = NULL,
   const POINT& pointMask = CPoint(
   0, 0 )
) const throw( );
```

Parameters

hDestDC

A handle to the destination device context.

pPoints

A pointer to an array of three points in logical space that identify three corners of the destination parallelogram. The upper left corner of the source rectangle is mapped to the first point in this array, the upper-right corner to the second point in this array, and the lower left corner to the third point. The lower-right corner of the source rectangle is mapped to the implicit fourth point in the parallelogram.

hbmMask

A handle to an optional monochrome bitmap that is used to mask the colors of the source rectangle.

xSrc

The x-coordinate, in logical units, of the upper left corner of the source rectangle.

ySrc

The y-coordinate, in logical units, of the upper left corner of the source rectangle.

nSrcWidth

The width, in logical units, of the source rectangle.

nSrcHeight

The height, in logical units, of the source rectangle.

xMask

The x-coordinate of the upper left corner of the monochrome bitmap.

yMask

The y-coordinate of the upper left corner of the monochrome bitmap.

rectSrc

A reference to a RECT structure specifying the coordinates of the source rectangle.

pointMask

A POINT structure indicating the upper left corner of the mask bitmap.

Return Value

Nonzero if successful, otherwise 0.

Remarks

If *hbmMask* identifies a valid monochrome bitmap, **PlgBit** uses this bitmap to mask the bits of color data from the source rectangle.

This method applies to Windows NT, versions 4.0 and later only. See PlgBlt in the *Platform SDK* and CImage Limitations with Earlier Operating Systems for more detailed information.

See Also

Reference

CImage Class
CImage::BitBlt
CImage::MaskBlt

Other Resources

CImage Members

CImage::ReleaseDC

Releases the device context.

```
void ReleaseDC( ) const throw( );
```

Remarks

Because only one bitmap can be selected into a device context at a time, you must call **ReleaseDC** for each call to GetDC.

See Also

Reference

CImage Class

Other Resources

CImage Members

CImage::ReleaseGDIPlus

Releases resources used by GDI+.

```
void ReleaseGDIPlus() throw();
```

Remarks

This method must be called to free resources allocated by a global **CImage** object. See CImage::CImage.

See Also

Reference

CImage Class

Other Resources

CImage Members

CImage::Save

Saves an image to the specified stream or file on disk.

```
HRESULT Save(
   IStream* pStream,
   REFGUID guidFileType
) const throw();
HRESULT Save(
   LPCTSTR pszFileName,
   REFGUID guidFileType= GUID_NULL
) const throw();
```

Parameters

pStream

A pointer to a COM IStream object containing the file image data.

pszFileName

A pointer to the file name for the image.

guidFileType

The file type to save the image as. Can be one of the following:

- **ImageFormatBMP** An uncompressed bitmap image.
- **ImageFormatPNG** A Portable Network Graphic (PNG) compressed image.
- **ImageFormatJPEG** A JPEG compressed image.
- **ImageFormatGIF** A GIF compressed image.

Note

For a complete list of constants, see Image File Format Constants in the Platform SDK.

Return Value

A standard **HRESULT**.

Remarks

Call this function to save the image using a specified name and type. If the *guidFileType* parameter is not included, the file name's file extension will be used to determine the image format. If no extension is provided, the image will be saved in BMP format.

Example:

```
// Demonstrating saving various file formats
int _tmain(int argc, _TCHAR* argv[])
{
   CImage myimage;
   // load existing image
   myimage.Load("image.bmp");
      // save an image in BMP format
   myimage.Save("c:\image1.bmp");
   // save an image in BMP format
   myimage.Save("c:\image2",ImageFormatBMP);
   // save an image in JPEG format
   myimage.Save("c:\image3.jpg");
   // save an image in BMP format, even though jpg file extension is used
   myimage.Save("c:\image4.jpg",ImageFormatBMP);
   return 0;
}
```

See also the ShowImage Sample.

See Also

Reference

CImage Class

CImage::Load

Other Resources

CImage Members

CImage::SetColorTable

Sets the red, green, blue (RGB) color values for a range of entries in the palette of the DIB section.

```
void SetColorTable(
   UINT iFirstColor,
   UINT nColors,
   const RGBQUAD* prgbColors
) throw( );
```

Parameters

iFirstColor

The color table index of the first entry to set.

nColors

The number of color table entries to set.

prgbColors

A pointer to the array of RGBQUAD structures to set the color table entries.

Remarks

This method supports only DIB section bitmaps.

See Also

Reference

CImage Class

CImage::GetColorTable

Other Resources

CImage Members

CImage::SetPixel

Sets the color of a pixel at a given location in the bitmap.

```
void SetPixel(
   int x,
   int y,
   COLORREF color
) throw( );
```

Parameters

x

The horizontal location of the pixel to set.

y

The vertical location of the pixel to set.

color

The color to which you set the pixel.

Remarks

This method fails if the pixel coordinates lie outside of the selected clipping region.

See Also

Reference

CImage Class
CImage::GetPixel
CImage::SetPixelIndexed
CImage::SetPixelRGB

Other Resources

CImage Members

CImage::SetPixelIndexed

Sets the pixel color to the color located at *iIndex* in the color palette.

```
void SetPixelIndexed(
   int x,
   int y,
   int iIndex
) throw( );
```

Parameters

x

The horizontal location of the pixel to set.

y

The vertical location of the pixel to set.

iIndex

The index of a color in the color palette.

See Also

Reference

CImage Class
CImage::SetPixel
CImage::SetPixelRGB

Other Resources

CImage Members

CImage::SetPixelRGB

Sets the pixel at the locations specified by *x* and *y* to the colors indicated by *r*, *g*, and *b*, in a red, green, blue (RGB) image.

```
void SetPixelRGB(
   int x,
   int y,
   BYTE r,
   BYTE g,
   BYTE b
) throw( );
```

Parameters

x

The horizontal location of the pixel to set.

y

The vertical location of the pixel to set.

r

The intensity of the red color.

g

The intensity of the green color.

b

The intensity of the blue color.

Remarks

The red, green, and blue parameters are each represented by a number between 0 and 255. If you set all three parameters to zero, the combined resulting color is black. If you set all three parameters to 255, the combined resulting color is white.

See Also

Reference

CImage Class
CImage::SetPixel
CImage::SetPixelIndexed

Other Resources

CImage Members

CImage::SetTransparentColor

Sets a color at a given indexed location as transparent.

```
LONG SetTransparentColor(
   LONG iTransparentColor
) throw( );
```

Parameters

iTransparentColor

The index, in a color palette, of the color to set to transparent. If –1, no color is set to transparent.

Return Value

The index of the color previously set as transparent.

See Also

Reference

CImage Class

CImage::GetTransparentColor

Other Resources

CImage Members

CImage::StretchBlt

Copies a bitmap from the source device context to this current device context.

```
BOOL StretchBlt(
   HDC hDestDC,
   int xDest,
   int yDest,
   int nDestWidth,
   int nDestHeight,
   DWORD dwROP = SRCCOPY
) const throw( );
BOOL StretchBlt(
   HDC hDestDC,
   const RECT& rectDest,
   DWORD dwROP = SRCCOPY
) const throw( );
BOOL StretchBlt(
   HDC hDestDC,
   int xDest,
   int yDest,
   int nDestWidth,
   int nDestHeight,
   int xSrc,
   int ySrc,
   int nSrcWidth,
   int nSrcHeight,
   DWORD dwROP = SRCCOPY
) const throw( );
BOOL StretchBlt(
   HDC hDestDC,
   const RECT& rectDest,
   const RECT& rectSrc,
   DWORD dwROP = SRCCOPY
) const throw( );
```

Parameters

hDestDC

A handle to the destination device context.

xDest

The x-coordinate, in logical units, of the upper left corner of the destination rectangle.

yDest

The y-coordinate, in logical units, of the upper left corner of the destination rectangle.

nDestWidth

The width, in logical units, of the destination rectangle.

nDestHeight

The height, in logical units, of the destination rectangle.

dwROP

The raster operation to be performed. Raster-operation codes define exactly how to combine the bits of the source, the destination, and the pattern (as defined by the currently selected brush) to form the destination. See BitBlt in the *Platform SDK* for a list of other raster-operation codes and their descriptions.

rectDest

A reference to a RECT structure, identifying the destination.

xSrc

The x-coordinate, in logical units, of the upper left corner of the source rectangle.

ySrc

The y-coordinate, in logical units, of the upper left corner of the source rectangle.

nSrcWidth

The width, in logical units, of the source rectangle.

nSrcHeight

The height, in logical units, of the source rectangle.

rectSrc

A reference to a **RECT** structure, identifying the source.

Return Value

Nonzero if successful, otherwise 0.

Remarks

For more information, see StretchBlt in the *Platform SDK*.

See Also

Reference

CImage Class
CImage::BitBlt
CImage::MaskBlt

Other Resources

CImage Members

CImage::TransparentBlt

Copies a bitmap from the source device context to this current device context.

```
BOOL TransparentBlt(
   HDC hDestDC,
   int xDest,
   int yDest,
   int nDestWidth,
   int nDestHeight,
   UINT crTransparent = CLR_INVALID
) const throw( );
BOOL TransparentBlt(
   HDC hDestDC,
   const RECT& rectDest,
   UINT crTransparent = CLR_INVALID
) const throw( );
BOOL TransparentBlt(
   HDC hDestDC,
   int xDest,
   int yDest,
   int nDestWidth,
   int nDestHeight,
   int xSrc,
   int ySrc,
   int nSrcWidth,
   int nSrcHeight,
   UINT crTransparent = CLR_INVALID
) const throw( );
BOOL TransparentBlt(
   HDC hDestDC,
   const RECT& rectDest,
   const RECT& rectSrc,
   UINT crTransparent = CLR_INVALID
) const throw( );
```

Parameters

hDestDC

A handle to the destination device context.

xDest

The x-coordinate, in logical units, of the upper left corner of the destination rectangle.

yDest

The y-coordinate, in logical units, of the upper left corner of the destination rectangle.

nDestWidth

The width, in logical units, of the destination rectangle.

nDestHeight

The height, in logical units, of the destination rectangle.

crTransparent

The color in the source bitmap to treat as transparent. By default, **CLR_INVALID**, indicating that the color currently set as the transparent color of the image should be used.

rectDest

A reference to a RECT structure, identifying the destination.

xSrc

The x-coordinate, in logical units, of the upper left corner of the source rectangle.

ySrc

The y-coordinate, in logical units, of the upper left corner of the source rectangle.

nSrcWidth

The width, in logical units, of the source rectangle.

nSrcHeight

The height, in logical units, of the source rectangle.

rectSrc

A reference to a **RECT** structure, identifying the source.

Return Value

TRUE if successful, otherwise **FALSE**.

Remarks

TransparentBlt is supported for source bitmaps of 4 bits per pixel and 8 bits per pixel. Use CImage::AlphaBlend to specify 32 bits-per-pixel bitmaps with transparency.

This method is applicable to Microsoft Windows 2000, Windows 98, and later systems. See TransparentBlt in the *Platform SDK* and CImage Limitations with Earlier Operating Systems for more detailed information.

Example

```
// Performs a transparent blit from the source image to the destination
// image using the images' current transparency settings
BOOL TransparentBlt( CImage* pSrcImage, CImage* pDstImage,
                     int xDest, int yDest,
                     int nDestWidth, int nDestHeight )
{
   HDC hDstDC = NULL;
   BOOL bResult;

   if( pSrcImage == NULL || pDstImage == NULL )
   {
      // Invalid parameter
      return FALSE;
   }

   // Obtain a DC to the destination image
   hDstDC = pDstImage->GetDC();

   // Perform the blit
   bResult = pSrcImage->TransparentBlt( hDstDC, xDest, yDest, nDestWidth, nDestHeight );

   // Release the destination DC
   pDstImage->ReleaseDC();

   return bResult;
}
```

See Also

Reference

CImage Class
CImage::StretchBlt
CImage::MaskBlt
CImage::BitBlt

Other Resources

CImage Members

CImage Operators

For information about the operators in **CImage**, see CImage Members.

CImage::operator HBITMAP

Use this operator to get the attached Windows GDI handle of the **CImage** object. This operator is a casting operator, which supports direct use of an **HBITMAP** object.

See Also

Reference

CImage Class

Other Resources

CImage Members

COleDateTime Class

Encapsulates the **DATE** data type used in OLE automation.

```
class COleDateTime
```

Remarks

COleDateTime does not have a base class.

It is one of the possible types for the VARIANT data type of OLE automation. A **COleDateTime** value represents an absolute date and time value.

The **DATE** type is implemented as a floating-point value, measuring days from midnight, 30 December 1899. So, midnight, 31 December 1899 is represented by 1.0. Similarly, 6 AM, 1 January 1900 is represented by 2.25, and midnight, 29 December 1899 is – 1.0. However, 6 AM, 29 December 1899 is – 1.25.

Note
To interpret the time portion, take the absolute value of the fractional part of the number.

The **COleDateTime** class handles dates from 1 January 100 – 31 December 9999. **COleDateTime** does not support Julian dates. The Gregorian calendar is assumed to extend back in time to 1 January 100.

Note
COleDateTime ignores Daylight Saving Time. See Date and Time: Automation Support.

This type is also used to represent date-only or time-only values. By convention, the date 0 (30 December 1899) is used for time-only values. Similarly, the time 0:00 (midnight) is used for date-only values.

If you create a **COleDateTime** object with a date less than 100, the date will be accepted, but subsequent calls to **GetYear**, **GetMonth**, **GetDay**, **GetHour**, **GetMinute**, and **GetSecond** will fail and return -1. Previously, you could use two-digit dates, but dates must be 100 or greater in MFC 4.2 and later.

To avoid problems, specify a four-digit date. For example:

```
COleDateTime mytime(1996,1,1,0,0,0);
```

Basic arithmetic operations for the **COleDateTime** values use the companion class COleDateTimeSpan. **COleDateTimeSpan** values represent relative time, an interval. The relation between these classes is analogous to the one between CTime and CTimeSpan.

For more information on the **COleDateTime** and **COleDateTimeSpan** classes, see the article Date and Time: Automation Support.

Requirements

Header: ATLComTime.h

Example

See the following samples:

- MantaWeb Sample
- NotModified Sample
- ShowImage Sample

See Also

Reference

COleVariant Class

Hierarchy Chart

Other Resources

COleDateTime Members

ATL/MFC Shared Classes

COleDateTime Members

Construction

COleDateTime	Constructs a **COleDateTime** object.
GetCurrentTime	Creates a **COleDateTime** object that represents the current time (static member function).

Attributes

GetAsSystemTime	Call this method to obtain the time in the **COleDateTime** object as a SYSTEMTIME data structure.
GetAsDBTIMESTAMP	Call this method to obtain the time in the **COleDateTime** object as a **DBTIMESTAMP** data structure.
GetAsUDATE	Call this method to obtain the time in the **COleDateTime** as a **UDATE** data structure.
GetDay	Returns the day this **COleDateTime** object represents (1 – 31).
GetDayOfWeek	Returns the day of the week this **COleDateTime** object represents (Sunday = 1).
GetDayOfYear	Returns the day of the year this **COleDateTime** object represents (Jan 1 = 1).
GetHour	Returns the hour this **COleDateTime** object represents (0 – 23).
GetMinute	Returns the minute this **COleDateTime** object represents (0 – 59).
GetMonth	Returns the month this **COleDateTime** object represents (1 – 12).
GetSecond	Returns the second this **COleDateTime** object represents (0 – 59).
GetStatus	Gets the status (validity) of this **COleDateTime** object.
GetYear	Returns the year this **COleDateTime** object represents.
SetStatus	Sets the status (validity) of this **COleDateTime** object.

Operations

Format	Generates a formatted string representation of a **COleDateTime** object.
ParseDateTime	Reads a date/time value from a string and sets the value of **COleDateTime**.
SetDate	Sets the value of this **COleDateTime** object to the specified date-only value.
SetDateTime	Sets the value of this **COleDateTime** object to the specified date/time value.
SetTime	Sets the value of this **COleDateTime** object to the specified time-only value.

Operators

operator +, -	Add and subtract **COleDateTime** values.
operator +=, -=	Add and subtract a **ColeDateTime** value from this **COleDateTime** object.
operator =	Copies a **COleDateTime** value.

operator ==, <, <=, etc.	Compare two **ColeDateTime** values.
operator DATE*	Converts a **ColeDateTime** value into a **DATE***.
operator DATE	Converts a **ColeDateTime** value into a **DATE**.

Data Members

m_dt	Contains the underlying **DATE** for this **COleDateTime** object.
m_status	Contains the status of this **COleDateTime** object.

See Also

Reference

COleDateTime Class

Hierarchy Chart

COleDateTime Member Functions

For information about the member functions in **COleDateTime**, see COleDateTime Members.

COleDateTime::COleDateTime

Constructs a **COleDateTime** object.

```
COleDateTime( ) throw( );
COleDateTime(
   const VARIANT& varSrc
) throw( );
COleDateTime(
   DATE dtSrc
) throw( );
COleDateTime(
   time_t timeSrc
) throw( );
COleDateTime(
   __time64_t timeSrc
) throw( );
COleDateTime(
   const SYSTEMTIME& systimeSrc
) throw( );
COleDateTime(
   const FILETIME& filetimeSrc
) throw( );
COleDateTime(
   int nYear,
   int nMonth,
   int nDay,
   int nHour,
   int nMin,
   int nSec
) throw( );
COleDateTime(
   WORD wDosDate,
   WORD wDosTime
) throw( );
COleDateTime(
      const DBTIMESTAMP& dbts
) throw();
```

Parameters

dateSrc

An existing **COleDateTime** object to be copied into the new **COleDateTime** object.

varSrc

An existing **VARIANT** data structure (possibly a **COleVariant** object) to be converted to a date/time value (**VT_DATE**) and copied into the new **COleDateTime** object.

dtSrc

A date/time (**DATE**) value to be copied into the new **COleDateTime** object.

timeSrc

A **time_t** or **__time64_t** value to be converted to a date/time value and copied into the new **COleDateTime** object.

systimeSrc

A **SYSTEMTIME** structure to be converted to a date/time value and copied into the new **COleDateTime** object.

filetimeSrc

A **FILETIME** structure to be converted to a date/time value and copied into the new **COleDateTime** object. Note that **FILETIME** uses Universal Coordinated Time (UTC), so if you pass a local time in the structure, your results will be incorrect. See File Times in the *Platform SDK* for more information.

nYear, *nMonth*, *nDay*, *nHour*, *nMin*, *nSec*

Indicate the date and time values to be copied into the new **COleDateTime** object.

wDosDate, *wDosTime*

MS-DOS date and time values to be converted to a date/time value and copied into the new **COleDateTime** object.

dbts

A reference to a DBTIMESTAMP structure containing the current local time.

Remarks

All these constructors create new **COleDateTime** objects initialized to the specified value. The following table shows valid ranges for each date and time component:

Date/time component	Valid range
year	100 – 9999
month	0 – 12
day	0 – 31
hour	0 – 23
minute	0 – 59
second	0 – 59

Note that the actual upper bound for the day component varies based on the month and year components. For details, see the **SetDate** or **SetDateTime** member functions.

Following is a brief description of each constructor:

- **COleDateTime( )** Constructs a **COleDateTime** object initialized to 0 (midnight, 30 December 1899).
- **COleDateTime(** *dateSrc* **)** Constructs a **COleDateTime** object from an existing **COleDateTime** object.
- **COleDateTime(** *varSrc* **)** Constructs a **COleDateTime** object. Attempts to convert a VARIANT structure or COleVariant object to a date/time (**VT_DATE**) value. If this conversion is successful, the converted value is copied into the new **COleDateTime** object. If it is not, the value of the **COleDateTime** object is set to 0 (midnight, 30 December 1899) and its status to invalid.
- **COleDateTime(** *dtSrc* **)** Constructs a **COleDateTime** object from a **DATE** value.
- **COleDateTime(** *timeSrc* **)** Constructs a **COleDateTime** object from a **time_t** value.
- **COleDateTime(** *systimeSrc* **)** Constructs a **COleDateTime** object from a **SYSTEMTIME** value.
- **COleDateTime(** *filetimeSrc* **)** Constructs a **COleDateTime** object from a **FILETIME** value. . Note that **FILETIME** uses Universal Coordinated Time (UTC), so if you pass a local time in the structure, your results will be incorrect. See File Times in the *Platform SDK* for more information.
- **COleDateTime(** *nYear*, *nMonth*, *nDay*, *nHour*, *nMin*, *nSec* **)** Constructs a **COleDateTime** object from the specified numerical values.
- **COleDateTime(** *wDosDate*, *wDosTime* **)** Constructs a **COleDateTime** object from the specified MS-DOS date and time values.

For more information, see the VARIANT entry in the *Platform SDK*.

For more information on the **time_t** data type, see the time function in the *Run-Time Library Reference*.

For more information, see the SYSTEMTIME and FILETIME structures in the *Platform SDK*.

For more information on MS-DOS date and time values, see DosDateTimeToVariantTime in the *Platform SDK*.

For more information about the bounds for **COleDateTime** values, see the article Date and Time: Automation Support.

Note
The constructor using **DBTIMESTAMP** parameter is only available when OLEDB.h is included.

Example

```
time_t osBinaryTime;   // C run-time time (defined in <time.h>)
time(&osBinaryTime);   // Get the current time from the
                      // operating system.

COleDateTime time1;   // initialized to 00:00am, 30 December 1899
                      // (and m_nStatus is valid!)

COleDateTime time2 = time1; // Copy constructor
COleDateTime time3(osBinaryTime)   // from time_t
COleDateTime time4( 1999, 3, 19, 22, 15, 0 ); // 10:15PM March 19, 1999

SYSTEMTIME sysTime;   // Win32 time information
GetSystemTime(&sysTime);

COleDateTime time5(sysTime);
```

For another example, see the ShowImage Sample.

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::SetDate
COleDateTime::SetDateTime
COleDateTime::SetTime
COleDateTime::GetStatus
COleDateTime::operator =
COleDateTime::m_dt
COleDateTime::m_status

Other Resources

COleDateTime Members

COleDateTime::Format

Creates a formatted representation of the date/time value.

```
CString Format(
   DWORD dwFlags = 0,
   LCID lcid = LANG_USER_DEFAULT
) const;
CString Format(
   LPCTSTR lpszFormat
) const;
CString Format(
   UINT nFormatID
) const;
```

Parameters

dwFlags

Indicates one of the following locale flags:

- **LOCALE_NOUSEROVERRIDE** Use the system default locale settings, rather than custom user settings.
- **VAR_TIMEVALUEONLY** Ignore the date portion during parsing.
- **VAR_DATEVALUEONLY** Ignore the time portion during parsing.

lcid

Indicates locale ID to use for the conversion.

lpszFormat

A formatting string similar to the **printf** formatting string. Formatting codes, preceded by a percent (**%**) sign, are replaced by the corresponding **COleDateTime** component. Other characters in the formatting string are copied unchanged to the returned string. See the run-time function strftime for details. The value and meaning of the formatting codes for **Format** are listed below:

- **%H** Hours in the current day
- **%M** Minutes in the current hour
- **%S** Seconds in the current minute
- **%%** Percent sign

nFormatID

The resource ID for the format-control string.

Return Value

A **CString** that contains the formatted date/time value.

Remarks

If the status of this **COleDateTime** object is null, the return value is an empty string. If the status is invalid, the return string is specified by the string resource **IDS_INVALID_DATETIME**.

A brief description of the three forms for this function follows:

Format(*dwFlags* **,** *lcid* **)**

This form formats the value using the national language specifications (locale IDs) for date and time. Using the default parameters, this form will print the date and the time, unless the time portion is 0 (midnight), in which case it will print just the date, or the date portion is 0 (30 December 1899), in which case it will print just the time. If the date/time value is 0 (30 December 1899, midnight), this form with the default parameters will print midnight.

Format(*lpszFormat* **)**

This form formats the value using the format string which contains special formatting codes that are preceded by a percent sign (%), as in **printf**. The formatting string is passed as a parameter to the function. For more information about the formatting codes, see strftime, wcsftime in the *Run-Time Library Reference*.

Format(*nFormatID* **)**

This form formats the value using the format string which contains special formatting codes that are preceded by a percent sign (%), as in **printf**. The formatting string is a resource. The ID of this string resource is passed as the parameter. For more information about the formatting codes, see strftime, wcsftime in the *Run-Time Library Reference*.

For a listing of locale ID values, see the section Supporting Multiple National Languages in the *Platform SDK*.

Example

```
COleDateTime t(1999, 3, 19, 22, 15, 0);

CString str = t.Format(_T("%A, %B %d, %Y"));
ASSERT(str == _T("Friday, March 19, 1999"));
```

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::ParseDateTime
COleDateTime::GetStatus

Other Resources

COleDateTime Members

COleDateTime::GetAsDBTIMESTAMP

Call this method to obtain the time in the **COleDateTime** object as a **DBTIMESTAMP** data structure.

```
bool GetAsDBTIMESTAMP(
      DBTIMESTAMP& dbts
) const throw();
```

Parameters

dbts

A reference to a DBTIMESTAMP structure.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Stores the resulting time in the referenced *dbts* structure. The **DBTIMESTAMP** data structure initialized by this function will have its **fraction** member set to zero.

Example

```
// GetAsDBTIMESTAMP Example
#include "oledb.h"
#include "ATLComTime.h"
int main()
{
   COleDateTime t = COleDateTime::GetCurrentTime();
   DBTIMESTAMP ts;
   t.GetAsDBTIMESTAMP( ts ); // retrieves the time in t into the ts structure
}
```

See Also

Reference

COleDateTime Class

Hierarchy Chart

Other Resources

COleDateTime Members

COleDateTime::GetAsSystemTime

Call this method to obtain the time in the **COleDateTime** object as a **SYSTEMTIME** data structure.

```
bool GetAsSystemTime(
   SYSTEMTIME& sysTime
) const throw( );
```

Parameters

sysTime

A reference to a SYSTEMTIME structure to receive the converted date/time value from the **COleDateTime** object.

Return Value

Returns **true** if successful; **false** if the conversion fails, or if the **COleDateTime** object is **NULL** or invalid.

Remarks

GetAsSystemTime stores the resulting time in the referenced *sysTime* object. The **SYSTEMTIME** data structure initialized by this function will have its **wMilliseconds** member set to zero.

See GetStatus for more information on the status information held in a **COleDateTime** object.

See Also

Reference

COleDateTime Class

Hierarchy Chart

Other Resources

COleDateTime Members

COleDateTime::GetAsUDATE

Call this method to obtain the time in the **COleDateTime** object as a **UDATE** data structure.

```
bool GetAsUDATE(
   UDATE& udate
) const throw( );
```

Parameters

udate

A reference to a **UDATE** structure to receive the converted date/time value from the **COleDateTime** object.

Return Value

Returns **true** if successful; **false** if the conversion fails, or if the **COleDateTime** object is **NULL** or invalid.

Remarks

A **UDATE** structure represents an "unpacked" date. See the function VarDateFromUdate for more details.

See Also

Reference

COleDateTime Class

Hierarchy Chart

Other Resources

COleDateTime Members

COleDateTime::GetCurrentTime

Call this static member function to return the current date/time value.

```
static COleDateTime WINAPI GetCurrentTime( ) throw( );
```

Example

```
// example for COleDateTime::GetCurrentTime
COleDateTime dateTest;
   // dateTest value = midnight 30 December 1899

dateTest = COleDateTime::GetCurrentTime();
   // dateTest value = current date and time

// a second example for COleDateTime::GetCurrentTime
// Since GetCurrentTime() is a static member, you can use it in
// a constructor:

COleDateTime t1 = COleDateTime::GetCurrentTime();
COleDateTime t2(COleDateTime::GetCurrentTime());

// Or in a normal assignment operator

COleDateTime t3;
t3 = COleDateTime::GetCurrentTime();

// or even in an expression

 if (COleDateTime::GetCurrentTime().GetDayOfWeek() == 6)
    _tprintf(_T("Thank Goodness it is Friday!\n\n"));
```

For another example, see the MantaWeb Sample.

See Also

Reference

COleDateTime Class

Hierarchy Chart

Other Resources

COleDateTime Members

COleDateTime::GetDay

Gets the day of the month represented by this date/time value.

```
int GetDay( ) const throw( );
```

Return Value

The day of the month represented by the value of this **COleDateTime** object or **COleDateTime::error** if the day could not be obtained.

Remarks

Valid return values range between 1 and 31.

For information on other member functions that query the value of this **COleDateTime** object, see the following member functions:

- GetMonth
- GetYear
- GetHour
- GetMinute
- GetSecond
- GetDayOfWeek
- GetDayOfYear

Example

```
COleDateTime t(1999, 3, 19, 22, 15, 0);  // 10:15PM March 19, 1999
ASSERT(t.GetDay() == 19);
ASSERT(t.GetMonth() == 3);
ASSERT(t.GetYear() == 1999);
```

For another example, see the MantaWeb Sample.

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::COleDateTime
COleDateTime::SetDateTime
COleDateTime::operator =
COleDateTime::GetStatus

Other Resources

COleDateTime Members

COleDateTime::GetDayOfWeek

Gets the day of the month represented by this date/time value.

```
int GetDayOfWeek( ) const throw( );
```

Return Value

The day of the week represented by the value of this **COleDateTime** object or **COleDateTime::error** if the day of the week could not be obtained.

Remarks

Valid return values range between 1 and 7, where 1=Sunday, 2=Monday, and so on.

For information on other member functions that query the value of this **COleDateTime** object, see the following member functions:

- GetDay
- GetMonth
- GetYear
- GetHour
- GetMinute
- GetSecond
- GetDayOfYear

Example

```
COleDateTime t(1999, 3, 19, 22, 15, 0); // 10:15PM March 19, 1999
ASSERT(t.GetDayOfWeek() == 6);          // it's a Friday
```

For another example, see the MantaWeb Sample.

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::COleDateTime
COleDateTime::SetDateTime
COleDateTime::operator =
COleDateTime::GetStatus

Other Resources

COleDateTime Members

COleDateTime::GetDayOfYear

Gets the day of the year represented by this date/time value.

```
int GetDayOfYear( ) const throw( );
```

Return Value

The day of the year represented by the value of this **COleDateTime** object or **COleDateTime::error** if the day of the year could not be obtained.

Remarks

Valid return values range between 1 and 366, where January 1 = 1.

For information on other member functions that query the value of this **COleDateTime** object, see the following member functions:

- GetDay
- GetMonth
- GetYear
- GetHour
- GetMinute
- GetSecond
- GetDayOfWeek

Example

```
COleDateTime t(1999, 3, 19, 22, 15, 0); // 10:15PM March 19, 1999
ASSERT(t.GetDayOfYear() == 78);         // 78th day of that year
```

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::COleDateTime
COleDateTime::SetDateTime
COleDateTime::operator =
COleDateTime::GetStatus

Other Resources

COleDateTime Members

COleDateTime::GetHour

Gets the hour represented by this date/time value.

```
int GetHour( ) const throw( );
```

Return Value

The hour represented by the value of this **COleDateTime** object or **COleDateTime::error** if the hour could not be obtained.

Remarks

Valid return values range between 0 and 23.

For information on other member functions that query the value of this **COleDateTime** object, see the following member functions:

- GetDay
- GetMonth
- GetYear
- GetMinute
- GetSecond
- GetDayOfWeek
- GetDayOfYear

Example

```
COleDateTime t(1999, 3, 19, 22, 15, 0);  // 10:15PM March 19, 1999
ASSERT(t.GetSecond() == 0);
ASSERT(t.GetMinute() == 15);
ASSERT(t.GetHour() == 22);
```

For another example, see the MantaWeb Sample.

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::COleDateTime
COleDateTime::SetDateTime
COleDateTime::operator =
COleDateTime::GetStatus

Other Resources

COleDateTime Members

COleDateTime::GetMinute

Gets the minute represented by this date/time value.

```
int GetMinute( ) const throw( );
```

Return Value

The minute represented by the value of this **COleDateTime** object or **COleDateTime::error** if the minute could not be obtained.

Remarks

Valid return values range between 0 and 59.

For information on other member functions that query the value of this **COleDateTime** object, see the following member functions:

- GetDay
- GetMonth
- GetYear
- GetHour
- GetSecond
- GetDayOfWeek
- GetDayOfYear

Example

See the example for GetHour and see the MantaWeb Sample.

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::COleDateTime
COleDateTime::SetDateTime
COleDateTime::operator =
COleDateTime::GetStatus

Other Resources

COleDateTime Members

COleDateTime::GetMonth

Gets the month represented by this date/time value.

```
int GetMonth( ) const throw( );
```

Return Value

The month represented by the value of this **COleDateTime** object or **COleDateTime::error** if the month could not be obtained.

Remarks

Valid return values range between 1 and 12.

For information on other member functions that query the value of this **COleDateTime** object, see the following member functions:

- GetDay
- GetYear
- GetHour
- GetMinute
- GetSecond
- GetDayOfWeek
- GetDayOfYear

Example

See the example for GetDay and see the MantaWeb Sample.

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::COleDateTime
COleDateTime::SetDateTime
COleDateTime::operator =
COleDateTime::GetStatus

Other Resources

COleDateTime Members

COleDateTime::GetSecond

Gets the second represented by this date/time value.

```
int GetSecond( ) const throw( );
```

Return Value

The second represented by the value of this **COleDateTime** object or **COleDateTime::error** if the second could not be obtained.

Remarks

Valid return values range between 0 and 59.

Note
The **COleDateTime** class does not support leap seconds.

For more information about the implementation for **COleDateTime**, see the article Date and Time: Automation Support.

For information on other member functions that query the value of this **COleDateTime** object, see the following member functions:

- GetDay
- GetMonth
- GetYear
- GetHour
- GetMinute
- GetDayOfWeek
- GetDayOfYear

Example

See the example for GetHour.

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::COleDateTime
COleDateTime::SetDateTime
COleDateTime::operator =
COleDateTime::GetStatus

Other Resources

COleDateTime Members

COleDateTime::GetStatus

Gets the status (validity) of a given **COleDateTime** object.

```
DateTimeStatus GetStatus( ) const throw( );
```

Return Value

Returns the status of this **COleDateTime** value. If you call **GetStatus** on a **COleDateTime** object constructed with the default, it will return valid. If you call **GetStatus** on a **COleDateTime** object initialized with the constructor set to null, **GetStatus** will return null. See **Remarks** for more information.

Remarks

The return value is defined by the **DateTimeStatus** enumerated type, which is defined within the **COleDateTime** class.

```
enum DateTimeStatus
{
   error = -1,
   valid = 0,
   invalid = 1,    // Invalid date (out of range, etc.)
   null = 2,       // Literally has no value
};
```

For a brief description of these status values, see the following list:

- **COleDateTime::error** Indicates that an error occurred while attempting to obtain part of the date/time value.
- **COleDateTime::valid** Indicates that this **COleDateTime** object is valid.
- **COleDateTime::invalid** Indicates that this **COleDateTime** object is invalid; that is, its value may be incorrect.
- **COleDateTime::null** Indicates that this **COleDateTime** object is null, that is, that no value has been supplied for this object. (This is "null" in the database sense of "having no value," as opposed to the C++ **NULL**.)

The status of a **COleDateTime** object is invalid in the following cases:

- If its value is set from a **VARIANT** or **COleVariant** value that could not be converted to a date/time value.
- If its value is set from a **time_t**, **SYSTEMTIME**, or **FILETIME** value that could not be converted to a valid date/time value.
- If its value is set by **SetDateTime** with invalid parameter values.
- If this object has experienced an overflow or underflow during an arithmetic assignment operation, namely, **+=** or **-=**.
- If an invalid value was assigned to this object.
- If the status of this object was explicitly set to invalid using **SetStatus**.

For more information about the operations that may set the status to invalid, see the following member functions:

- COleDateTime
- SetDateTime
- operator +, -
- operator +=, -=

For more information about the bounds for **COleDateTime** values, see the article Date and Time: Automation Support.

Example

```
COleDateTime t;

// this one is a leap year
```

```
t.SetDateTime(2000, 2, 29, 5, 0, 0);
ASSERT(t.GetStatus() == COleDateTime::valid);

// this date isn't valid
t.SetDateTime(1925, 2, 30, 5, 0, 0);
ASSERT(t.GetStatus() == COleDateTime::invalid);

// the only way to set null is to set null!
t.SetStatus(COleDateTime::null);
ASSERT(t.GetStatus() == COleDateTime::null);
```

For another example, see the MantaWeb Sample.

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::SetStatus
COleDateTime::m_status

Other Resources

COleDateTime Members

COleDateTime::GetYear

Gets the year represented by this date/time value.

```
int GetYear( ) const throw( );
```

Return Value

The year represented by the value of this **COleDateTime** object or **COleDateTime::error** if the year could not be obtained.

Remarks

Valid return values range between 100 and 9999, which includes the century.

For information on other member functions that query the value of this **COleDateTime** object, see the following member functions:

- GetDay
- GetMonth
- GetHour
- GetMinute
- GetSecond
- GetDayOfWeek
- GetDayOfYear

For more information about the bounds for **COleDateTime** values, see the article Date and Time: Automation Support.

Example

See the example for GetDay and see the MantaWeb Sample.

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::COleDateTime
COleDateTime::SetDateTime
COleDateTime::operator =
COleDateTime::GetStatus

Other Resources

COleDateTime Members

COleDateTime::ParseDateTime

Parses a string to read a date/time value.

```
bool ParseDateTime(
   LPCTSTR lpszDate,
   DWORD dwFlags = 0,
   LCID lcid = LANG_USER_DEFAULT
) throw( );
```

Parameters

lpszDate

A pointer to the null-terminated string which is to be parsed. For details, see Remarks.

dwFlags

Indicates flags for locale settings and parsing. One or more of the following flags:

- **LOCALE_NOUSEROVERRIDE** Use the system default locale settings, rather than custom user settings.
- **VAR_TIMEVALUEONLY** Ignore the date portion during parsing.
- **VAR_DATEVALUEONLY** Ignore the time portion during parsing.

lcid

Indicates locale ID to use for the conversion.

Return Value

Returns **true** if the string was successfully converted to a date/time value, otherwise **false**.

Remarks

If the string was successfully converted to a date/time value, the value of this **COleDateTime** object is set to that value and its status to valid.

Note
Year values must lie between 100 and 9999, inclusively.

The *lpszDate* parameter can take a variety of formats. For example, the following strings contain acceptable date/time formats:

```
"25 January 1996"
"8:30:00"
"20:30:00"
"January 25, 1996 8:30:00"
"8:30:00 Jan. 25, 1996"
"1/25/1996 8:30:00"  // always specify the full year,
                     // even in a 'short date' format
```

Note that the locale ID will also affect whether the string format is acceptable for conversion to a date/time value.

In the case of **VAR_DATEVALUEONLY**, the time value is set to time 0, or midnight. In the case of **VAR_TIMEVALUEONLY**, the date value is set to date 0, meaning 30 December 1899.

If the string could not be converted to a date/time value or if there was a numerical overflow, the status of this **COleDateTime** object is invalid.

For a listing of locale ID values, see the section Supporting Multiple National Languages in the *Platform SDK*.

For more information about the bounds and implementation for **COleDateTime** values, see the article Date and Time: Automation Support.

Example

See the MantaWeb Sample.

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::Format
COleDateTime::GetStatus

Other Resources

COleDateTime Members

COleDateTime::SetDate

Sets the date and time of this **COleDateTime** object.

```
int SetDate(
   int nYear,
   int nMonth,
   int nDay
) throw( );
```

Parameters

nYear, *nMonth*, *nDay*

Indicate the date components to be copied into this **COleDateTime** object.

Return Value

Zero if the value of this **COleDateTime** object was set successfully; otherwise, 1. This return value is based on the **DateTimeStatus** enumerated type. For more information, see the SetStatus member function.

Remarks

The date is set to the specified values. The time is set to time 0, midnight.

See the following table for bounds for the parameter values:

Parameter	Bounds
nYear	100 – 9999
nMonth	1 – 12
nDay	0 – 31

If the day of the month overflows, it is converted to the correct day of the next month and the month and/or year is incremented accordingly. A day value of zero indicates the last day of the previous month. The behavior is the same as SystemTimeToVariantTime.

If the date value specified by the parameters is not valid, the status of this object is set to **COleDateTime::invalid**. You should use GetStatus to check the validity of the **DATE** value and should not assume that the value of m_dt will remain unmodified.

Here are some examples of date values:

nYear	***nMonth***	***nDay***	**Value**
2000	2	29	29 February 2000
1776	7	4	4 July 1776
1925	4	35	35 April 1925 (invalid date)
10000	1	1	1 January 10000 (invalid date)

To set both date and time, see COleDateTime::SetDateTime.

For information on member functions that query the value of this **COleDateTime** object, see the following member functions:

- GetDay
- GetMonth
- GetYear

- GetHour
- GetMinute
- GetSecond
- GetDayOfWeek
- GetDayOfYear

For more information about the bounds for **COleDateTime** values, see the article Date and Time: Automation Support.

Example

```
// set only the date, time set to midnight
dt.SetDate(1999, 3, 19);
ASSERT(dt.GetYear() == 1999);
ASSERT(dt.GetDay() == 19);
ASSERT(dt.GetMonth() == 3);
ASSERT(dt.GetHour() == 0);
ASSERT(dt.GetMinute() == 0);
ASSERT(dt.GetSecond() == 0);

// setting the time only resets the date to 1899!
dt.SetTime(22, 15, 0);
ASSERT(dt.GetYear() == 1899);
ASSERT(dt.GetDay() == 30);
ASSERT(dt.GetMonth() == 12);
ASSERT(dt.GetHour() == 22);
ASSERT(dt.GetMinute() == 15);
ASSERT(dt.GetSecond() == 0);
```

For another example, see the MantaWeb Sample.

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::COleDateTime
COleDateTime::SetDateTime
COleDateTime::operator =
COleDateTime::GetStatus
COleDateTime::m_dt

Other Resources

COleDateTime Members

COleDateTime::SetDateTime

Sets the date and time of this **COleDateTime** object.

```
int SetDateTime(
   int nYear,
   int nMonth,
   int nDay,
   int nHour,
   int nMin,
   int nSec
) throw( );
```

Parameters

nYear, *nMonth*, *nDay*, *nHour*, *nMin*, *nSec*

Indicate the date and time components to be copied into this **COleDateTime** object.

Return Value

Zero if the value of this **COleDateTime** object was set successfully; otherwise, 1. This return value is based on the **DateTimeStatus** enumerated type. For more information, see the SetStatus member function.

Remarks

See the following table for bounds for the parameter values:

Parameter	**Bounds**
nYear	100 – 9999
nMonth	1 – 12
nDay	0 – 31
nHour	0 – 23
nMin	0 – 59
nSec	0 – 59

If the day of the month overflows, it is converted to the correct day of the next month and the month and/or year is incremented accordingly. A day value of zero indicates the last day of the previous month. The behavior is the same as SystemTimeToVariantTime.

If the date or time value specified by the parameters is not valid, the status of this object is set to invalid and the value of this object is not changed.

Here are some examples of time values:

nHour	***nMin***	***nSec***	**Value**
1	3	3	01:03:03
23	45	0	23:45:00
25	30	0	Invalid
9	60	0	Invalid

Here are some examples of date values:

nYear	*nMonth*	*nDay*	**Value**
1995	4	15	15 April 1995
1789	7	14	17 July 1789
1925	2	30	Invalid
10000	1	1	Invalid

To set the date only, see COleDateTime::SetDate. To set the time only, see COleDateTime::SetTime.

For information on member functions that query the value of this **COleDateTime** object, see the following member functions:

- GetDay
- GetMonth
- GetYear
- GetHour
- GetMinute
- GetSecond
- GetDayOfWeek
- GetDayOfYear

For more information about the bounds for **COleDateTime** values, see the article Date and Time: Automation Support.

Example

See the example for GetStatus.

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::COleDateTime
COleDateTime::SetDate
COleDateTime::SetTime
COleDateTime::operator =
COleDateTime::GetStatus
COleDateTime::m_dt

Other Resources

COleDateTime Members

COleDateTime::SetStatus

Sets the status of this **COleDateTime** object.

```
void SetStatus(
   DateTimeStatus status
) throw( );
```

Parameters

status

The new status value for this **COleDateTime** object.

Remarks

The *status* parameter value is defined by the **DateTimeStatus** enumerated type, which is defined within the **COleDateTime** class. See COleDateTime::GetStatus for details.

Caution
This function is for advanced programming situations. This function does not alter the data in this object. It will most often be used to set the status to **null** or **invalid**. Note that the assignment operator (operator =) and SetDateTime do set the status of the object based on the source value(s).

Example

See the example for GetStatus and see the MantaWeb Sample.

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::GetStatus
COleDateTime::operator =
COleDateTime::SetDateTime
COleDateTime::m_dt

Other Resources

COleDateTime Members

COleDateTime::SetTime

Sets the date and time of this **COleDateTime** object.

```
int SetTime(
   int nHour,
   int nMin,
   int nSec
) throw( );
```

Parameters

nHour, *nMin*, *nSec*

Indicate the time components to be copied into this **COleDateTime** object.

Return Value

Zero if the value of this **COleDateTime** object was set successfully; otherwise, 1. This return value is based on the **DateTimeStatus** enumerated type. For more information, see the SetStatus member function.

Remarks

The time is set to the specified values. The date is set to date 0, meaning 30 December 1899.

See the following table for bounds for the parameter values:

Parameter	Bounds
nHour	0 – 23
nMin	0 – 59
nSec	0 – 59

If the time value specified by the parameters is not valid, the status of this object is set to invalid and the value of this object is not changed.

Here are some examples of time values:

nHour	***nMin***	***nSec***	**Value**
1	3	3	01:03:03
23	45	0	23:45:00
25	30	0	Invalid
9	60	0	Invalid

To set both date and time, see COleDateTime::SetDateTime.

For information on member functions that query the value of this **COleDateTime** object, see the following member functions:

- GetDay
- GetMonth
- GetYear
- GetHour
- GetMinute
- GetSecond

- GetDayOfWeek
- GetDayOfYear

For more information about the bounds for **COleDateTime** values, see the article Date and Time: Automation Support.

Example

See the example for SetDate.

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::COleDateTime
COleDateTime::SetDateTime
COleDateTime::operator =
COleDateTime::GetStatus
COleDateTime::m_dt

Other Resources

COleDateTime Members

COleDateTime Operators

For information about the operators in **COleDateTime**, see COleDateTime Members.

COleDateTime::operator =

Copies a **COleDateTime** value.

```
COleDateTime& operator =(
   const VARIANT& varSrc
) throw( );
COleDateTime& operator =(
   DATE dtSrc
) throw( );
COleDateTime& operator =(
   const time_t& timeSrc
) throw( );
COleDateTime& operator =(
   const __time64_t& timeSrc
) throw( );
COleDateTime& operator =(
   const SYSTEMTIME& systimeSrc
) throw( );
COleDateTime& operator =(
   const FILETIME& filetimeSrc
) throw( );
COleDateTime& operator =(
   const UDATE& udate
) throw( );
```

Remarks

These overloaded assignment operators copy the source date/time value into this **COleDateTime** object. A brief description of each these overloaded assignment operators follows:

- **operator =(** *dateSrc* **)** The value and status of the operand are copied into this **COleDateTime** object.
- **operator =(** *varSrc* **)** If the conversion of the VARIANT value (or COleVariant object) to a date/time (**VT_DATE**) is successful, the converted value is copied into this **COleDateTime** object and its status is set to valid. If the conversion is not successful, the value of this object is set to zero (30 December 1899, midnight) and its status to invalid.
- **operator =(** *dtSrc* **)** The **DATE** value is copied into this **COleDateTime** object and its status is set to valid.
- **operator =(** *timeSrc* **)** The **time_t** or **__time64_t** value is converted and copied into this **COleDateTime** object. If the conversion is successful, the status of this object is set to valid; if unsuccessful, it is set to invalid.
- **operator =(** *systimeSrc* **)** The SYSTEMTIME value is converted and copied into this **COleDateTime** object. If the conversion is successful, the status of this object is set to valid; if unsuccessful, it is set to invalid.
- **operator =(** *udate* **)** The **UDATE** value is converted and copied into this **COleDateTime** object. If the conversion is successful, the status of this object is set to valid; if unsuccessful, it is set to invalid. A **UDATE** structure represents an "unpacked" date. See the function VarDateFromUdate for more details.
- **operator =(** *filetimeSrc* **)** The FILETIME value is converted and copied into this **COleDateTime** object. If the conversion is successful, the status of this object is set to valid; if unsuccessful, it is set to invalid. **FILETIME** uses Universal Coordinated Time (UTC), so if you pass a local time in the structure, your results will be converted from local file time to system time to variant time. Note that this behavior is the same as in Visual C++ 6.0 and Visual C++.NET 2003 SP2. See File Times in the *Platform SDK* for more information.

For more information, see the VARIANT entry in the *Platform SDK*.

For more information on the **time_t** data type, see the time function in the *Run-Time Library Reference*.

For more information, see the SYSTEMTIME and FILETIME structures in the *Platform SDK*.

For more information about the bounds for **COleDateTime** values, see the article Date and Time: Automation Support.

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::COleDateTime
COleDateTime::SetDateTime
COleDateTime::GetStatus

Other Resources

COleDateTime Members

COleDateTime::operator +, -

Add and subtract **ColeDateTime** values.

```
COleDateTime operator +(
   COleDateTimeSpan dateSpan
) const throw( );
COleDateTime operator -(
   COleDateTimeSpan dateSpan
) const throw( );
COleDateTimeSpan operator -(
   const COleDateTime& date
) const throw( );
```

Remarks

COleDateTime objects represent absolute times. COleDateTimeSpan objects represent relative times. The first two operators allow you to add and subtract a **COleDateTimeSpan** value from a **COleDateTime** value. The third operator allows you to subtract one **COleDateTime** value from another to yield a **COleDateTimeSpan** value.

If either of the operands is null, the status of the resulting **COleDateTime** value is null.

If the resulting **COleDateTime** value falls outside the bounds of acceptable values, the status of that **COleDateTime** value is invalid.

If either of the operands is invalid and the other is not null, the status of the resulting **COleDateTime** value is invalid.

The **+** and **-** operators will assert if the **COleDateTime** object is set to null. See COleDateTime Relational Operators for an example.

For more information on the valid, invalid, and null status values, see the m_status member variable.

For more information about the bounds for **COleDateTime** values, see the article Date and Time: Automation Support.

Example

```
COleDateTime t1( 1999, 3, 19, 22, 15, 0 ); // 10:15PM March 19, 1999
COleDateTime t2( 1999, 3, 20, 22, 15, 0 ); // 10:15PM March 20, 1999

// Subtract 2 COleDateTimes
COleDateTimeSpan ts = t2 - t1;

// one day is 24 * 60 * 60 == 86400 seconds
ASSERT( ts.GetTotalSeconds() == 86400L );

// Add a COleDateTimeSpan to a COleDateTime.
ASSERT( ( t1 + ts ) == t2 );

// Subtract a COleDateTimeSpan from a COleDateTime.
ASSERT( ( t2 - ts ) == t1 );
```

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::operator +=, -=
COleDateTime::GetStatus
COleDateTimeSpan Class

Other Resources

COleDateTime Members

COleDateTime::operator +=, -=

Add and subtract a **ColeDateTime** value from this **COleDateTime** object.

```
COleDateTime& operator +=(
   COleDateTimeSpan dateSpan
) throw( );
COleDateTime& operator -=(
   COleDateTimeSpan dateSpan
) throw( );
```

Remarks

These operators allow you to add and subtract a **COleDateTimeSpan** value to and from this **COleDateTime**. If either of the operands is null, the status of the resulting **COleDateTime** value is null.

If the resulting **COleDateTime** value falls outside the bounds of acceptable values, the status of this **COleDateTime** value is set to invalid.

If either of the operands is invalid and other is not null, the status of the resulting **COleDateTime** value is invalid.

For more information on the valid, invalid, and null status values, see the m_status member variable.

The **+=** and **-=** operators will assert if the **COleDateTime** object is set to null. See COleDateTime Relational Operators for an example.

For more information about the bounds for **COleDateTime** values, see the article Date and Time: Automation Support.

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::operator +, -
COleDateTime::GetStatus

Other Resources

COleDateTime Members

COleDateTime::operator DATE

Converts a **ColeDateTime** value into a **DATE**.

```
operator DATE( ) const throw( );
```

Remarks

This operator returns a **DATE** object whose value is copied from this **COleDateTime** object. For more information about the implementation of the **DATE** object, see the article Date and Time: Automation Support.

The **DATE** operator will assert if the **COleDateTime** object is set to null. See COleDateTime Relational Operators for an example.

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::m_dt

Other Resources

COleDateTime Members

COleDateTime Relational Operators

Comparison operators.

```
bool operator ==(
   const COleDateTime& date
) const throw( );
bool operator !=(
   const COleDateTime& date
) const throw( );
bool operator <(
   const COleDateTime& date
) const throw( );
bool operator >(
   const COleDateTime& date
) const throw( );
bool operator <=(
   const COleDateTime& date
) const throw( );
bool operator >=(
   const COleDateTime& date
) const throw( );
```

Parameters

date

The **COleDateTime** object to be compared.

Return values

These operators compare two date/time values and return **true** if the condition is true; otherwise **false**.

Remarks

Note
An ATLASSERT will occur if either of the two operands is invalid.

Example

The last four lines of this example will **ASSERT** in debug mode.

```
// COleDateTime_Relational_Operators.cpp
#include <atlcomtime.h>
int main() {
   COleDateTime dateOne(95, 3, 15, 12, 0, 0); // 15 March 1995 12 noon
   COleDateTime dateTwo(dateOne);             // 15 March 1995 12 noon
   BOOL b;
   b = dateOne == dateTwo;   // TRUE

   dateTwo.SetStatus(COleDateTime::invalid);
   b = dateOne == dateTwo;   // FALSE, different status
   b = dateOne != dateTwo;   // TRUE, different status

   // will assert when compiled with /MDd
   b = dateOne < dateTwo;   // FALSE, same value
   b = dateOne > dateTwo;   // FALSE, same value
   b = dateOne <= dateTwo;   // TRUE, same value
   b = dateOne >= dateTwo;   // TRUE, same value
}
```

The operators **>=**, **<=**, **>**, and **<**, will assert if the **COleDateTime** object is set to null.

```
// COleDateTime_Relational_Operators_2.cpp
```

```
// compile with: /D_DEBUG=1 /MDd /c
#include <ATLComTime.h>

int main() {
   VARIANT v = {};
   v.vt = VT_NULL;
   COleDateTime t1( v );
   COleDateTime t2( v );
   t1 = t1 + t2;
}
```

See Also

Reference

COleDateTime Class

Hierarchy Chart

COleDateTime::GetStatus

Other Resources

COleDateTime Members

COleDateTime Data Members

For information about the data members in **COleDateTime**, see COleDateTime Members.

COleDateTime::m_dt

The underlying **DATE** structure for this **COleDateTime** object.

```
DATE m_dt;
```

Remarks

Caution
Changing the value in the **DATE** object accessed by the pointer returned by this function will change the value of this **COleDateTime** object. It does not change the status of this **COleDateTime** object.

For more information about the implementation of the **DATE** object, see the article Date and Time: Automation Support.

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::COleDateTime
COleDateTime::SetDateTime
COleDateTime::SetDate
COleDateTime::SetTime
COleDateTime::operator DATE

Other Resources

COleDateTime Members

COleDateTime::m_status

Contains the status of this **COleDateTime** object.

```
DateTimeStatus m_status;
```

Remarks

The type of this data member is the enumerated type **DateTimeStatus**, which is defined within the **COleDateTime** class. See COleDateTime::GetStatus for details.

Caution

This data member is for advanced programming situations. You should use the inline member functions GetStatus and SetStatus. See **SetStatus** for further cautions regarding explicitly setting this data member.

See Also

Reference

COleDateTime Class
Hierarchy Chart
COleDateTime::GetStatus
COleDateTime::SetStatus

Other Resources

COleDateTime Members

COleDateTimeSpan Class

Represents a relative time, a time span.

```
class COleDateTimeSpan
```

Remarks

COleDateTimeSpan does not have a base class.

A **COleDateTimeSpan** keeps time in days.

COleDateTimeSpan is used with its companion class COleDateTime. **COleDateTime** encapsulates the **DATE** data type of OLE automation. **COleDateTime** represents absolute time values. All **COleDateTime** calculations involve **COleDateTimeSpan** values. The relation between these classes is analogous to the one between CTime and CTimeSpan.

For more information on the **COleDateTime** and **COleDateTimeSpan** classes, see the article Date and Time: Automation Support.

Requirements

Header: ATLComTime.h

Example

See the MantaWeb Sample and the ShowImage Sample.

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

COleDateTime Class

Hierarchy Chart

Other Resources

COleDateTimeSpan Members

ATL/MFC Shared Classes

COleDateTimeSpan Members

Constructor

COleDateTimeSpan	Constructs a **COleDateTimeSpan** object.

Attributes

GetDays	Returns the day portion of the span this **COleDateTimeSpan** object represents.
GetHours	Returns the hour portion of the span this **COleDateTimeSpan** object represents.
GetMinutes	Returns the minute portion of the span this **COleDateTimeSpan** object represents.
GetSeconds	Returns the second portion of the span this **COleDateTimeSpan** object represents.
GetStatus	Gets the status (validity) of this **COleDateTimeSpan** object.
GetTotalDays	Returns the number of days this **COleDateTimeSpan** object represents.
GetTotalHours	Returns the number of hours this **COleDateTimeSpan** object represents.
GetTotalMinutes	Returns the number of minutes this **COleDateTimeSpan** object represents.
GetTotalSeconds	Returns the number of seconds this **COleDateTimeSpan** object represents.
SetStatus	Sets the status (validity) of this **COleDateTimeSpan** object.

Operations

Format	Generates a formatted string representation of a **COleDateTimeSpan** object.
SetDateTimeSpan	Sets the value of this **COleDateTimeSpan** object.

Operators

operator +, -	Add, subtract, and change sign for **COleDateTimeSpan** values.
operator +=, -=	Add and subtract a **COleDateTimeSpan** value from this **COleDateTimeSpan** value.
operator =	Copies a **COleDateTimeSpan** value.
operator ==, <, <=	Compare two **COleDateTimeSpan** values.
operator double	Converts this **COleDateTimeSpan** value to a **double**.

Data Members

m_span	Contains the underlying **double** for this **COleDateTimeSpan** object.
m_status	Contains the status of this **COleDateTimeSpan** object.

See Also

Reference

COleDateTimeSpan Class

Hierarchy Chart

COleDateTimeSpan Member Functions

For information about the member functions in **COleDateTimeSpan**, see COleDateTimeSpan Members.

COleDateTimeSpan::COleDateTimeSpan

Constructs a **COleDateTimeSpan** object.

```
COleDateTimeSpan( ) throw( );
COleDateTimeSpan(
   double dblSpanSrc
) throw( );
COleDateTimeSpan(
   LONG lDays,
   int nHours,
   int nMins,
   int nSecs
) throw( );
```

Parameters

dblSpanSrc

The number of days to be copied into the new **COleDateTimeSpan** object.

lDays, *nHours*, *nMins*, *nSecs*

Indicate the day and time values to be copied into the new **COleDateTimeSpan** object.

Remarks

All of these constructors create new **COleDateTimeSpan** objects initialized to the specified value. A brief description of each of these constructors follows:

- **COleDateTimeSpan( )** Constructs a **COleDateTimeSpan** object initialized to 0.
- **COleDateTimeSpan(** *dblSpanSrc* **)** Constructs a **COleDateTimeSpan** object from a floating-point value.
- **COleDateTimeSpan(** *lDays*, *nHours*, *nMins*, *nSecs* **)** Constructs a **COleDateTimeSpan** object initialized to the specified numerical values.

The status of the new **COleDateTimeSpan** object is set to valid.

For more information about the bounds for **COleDateTimeSpan** values, see the article Date and Time: Automation Support.

Example

```
COleDateTimeSpan spanOne( 2.75 );          // 2 days and 18 hours
COleDateTimeSpan spanTwo( 2, 18, 0, 0 );   // 2 days and 18 hours
COleDateTimeSpan spanThree( 3, -6, 0, 0 ); // 2 days and 18 hours

COleDateTimeSpan ts1;               // Uninitialized time value
COleDateTimeSpan ts2a(ts1);         // Copy constructor
COleDateTimeSpan ts2b = ts1;         // Copy constructor again
COleDateTimeSpan ts3(100.0);          // 100 days
COleDateTimeSpan ts4(0, 1, 5, 12);   // 1 hour, 5 minutes, and 12 seconds
```

See Also

Reference

COleDateTimeSpan Class
Hierarchy Chart
COleDateTimeSpan::operator =
COleDateTimeSpan::GetStatus
COleDateTimeSpan::m_span
COleDateTimeSpan::m_status

Other Resources

COleDateTimeSpan Members

COleDateTimeSpan::Format

Generates a formatted string representation of a **COleDateTimeSpan** object.

```
CString Format(
   LPCTSTR pFormat
) const;
CString Format(
   UINT nID
) const;
```

Parameters

pFormat

A formatting string similar to the **printf** formatting string. Formatting codes, preceded by a percent (**%**) sign, are replaced by the corresponding **COleDateTimeSpan** component. Other characters in the formatting string are copied unchanged to the returned string. The value and meaning of the formatting codes for **Format** are listed below:

- **%H** Hours in the current day
- **%M** Minutes in the current hour
- **%S** Seconds in the current minute
- **%%** Percent sign

The four format codes listed above are the only codes that Format will accept.

-

nID

The resource ID for the format-control string.

Return Value

A **CString** that contains the formatted date/time-span value.

Remarks

Call these functions to create a formatted representation of the time-span value. If the status of this **COleDateTimeSpan** object is null, the return value is an empty string. If the status is invalid, the return string is specified by the string resource **IDS_INVALID_DATETIMESPAN**.

A brief description of the forms for this function follows:

Format(*pFormat* **)**

This form formats the value using the format string that contains special formatting codes that are preceded by a percent sign (%), as in **printf**. The formatting string is passed as a parameter to the function.

Format(*nID* **)**

This form formats the value using the format string that contains special formatting codes that are preceded by a percent sign (%), as in **printf**. The formatting string is a resource. The ID of this string resource is passed as the parameter.

For a listing of locale ID values, see the section Supporting Multiple National Languages in the *Platform SDK*.

Example

```
{
    // get the current time
    COleDateTime tmStart = COleDateTime::GetCurrentTime();

    // waste some time
    CString str;
    ::Sleep(3000);
```

```
    // get the current time again
    COleDateTime tmFinish = COleDateTime::GetCurrentTime();

    // find the difference
    COleDateTimeSpan tmSpan = tmFinish - tmStart;

    // tell the user
    str = tmSpan.Format(_T("%S seconds elapsed"));
    _tprintf_s(_T("%s\n"), (LPCTSTR) str);
}
```

See Also

Reference

COleDateTimeSpan Class

Hierarchy Chart

COleDateTimeSpan::GetStatus

Other Resources

COleDateTimeSpan Members

COleDateTimeSpan::GetDays

Retrieves the day portion of this date/time-span value.

```
LONG GetDays( ) const throw( );
```

Return Value

The day portion of this date/time-span value.

Remarks

The return values from this function range between approximately – 3,615,000 and 3,615,000.

For other functions that query the value of a **COleDateTimeSpan** object, see the following member functions:

- GetHours
- GetMinutes
- GetSeconds
- GetTotalDays
- GetTotalHours
- GetTotalMinutes
- GetTotalSeconds

Example

```
COleDateTimeSpan ts( 3, 1, 5, 12 ); // 3 days, 1 hour, 5 min, and 12 sec
ASSERT( ts.GetDays() == 3 );
```

See Also

Reference

COleDateTimeSpan Class

Hierarchy Chart

COleDateTimeSpan::SetDateTimeSpan

Other Resources

COleDateTimeSpan Members

COleDateTimeSpan::GetHours

Retrieves the hour portion of this date/time-span value.

```
LONG GetHours( ) const throw( );
```

Return Value

The hours portion of this date/time-span value.

Remarks

The return values from this function range between – 23 and 23.

For other functions that query the value of a **COleDateTimeSpan** object, see the following member functions:

- GetDays
- GetMinutes
- GetSeconds
- GetTotalDays
- GetTotalHours
- GetTotalMinutes
- GetTotalSeconds

Example

```
COleDateTimeSpan ts( 3, 1, 5, 12 ); // 3 days, 1 hour, 5 min, and 12 sec
ASSERT( ts.GetHours() == 1 );
```

See Also

Reference

COleDateTimeSpan Class
Hierarchy Chart
COleDateTimeSpan::SetDateTimeSpan

Other Resources

COleDateTimeSpan Members

COleDateTimeSpan::GetMinutes

Retrieves the minute portion of this date/time-span value.

```
LONG GetMinutes( ) const throw( );
```

Return Value

The minutes portion of this date/time-span value.

Remarks

The return values from this function range between – 59 and 59.

For other functions that query the value of a **COleDateTimeSpan** object, see the following member functions:

- GetDays
- GetHours
- GetSeconds
- GetTotalDays
- GetTotalHours
- GetTotalMinutes
- GetTotalSeconds

Example

```
COleDateTimeSpan ts( 3, 1, 5, 12 ); // 3 days, 1 hour, 5 min, and 12 sec
ASSERT( ts.GetMinutes() == 5 );
```

See Also

Reference

COleDateTimeSpan Class

Hierarchy Chart

COleDateTimeSpan::SetDateTimeSpan

Other Resources

COleDateTimeSpan Members

COleDateTimeSpan::GetSeconds

Retrieves the second portion of this date/time-span value.

```
LONG GetSeconds( ) const throw( );
```

Return Value

The seconds portion of this date/time-span value.

Remarks

The return values from this function range between – 59 and 59.

For other functions that query the value of a **COleDateTimeSpan** object, see the following member functions:

- GetDays
- GetHours
- GetMinutes
- GetTotalDays
- GetTotalHours
- GetTotalMinutes
- GetTotalSeconds

Example

```
COleDateTimeSpan ts( 3, 1, 5, 12 ); // 3 days, 1 hour, 5 min, and 12 sec
ASSERT( ts.GetSeconds() == 12 );
```

See Also

Reference

COleDateTimeSpan Class
Hierarchy Chart
COleDateTimeSpan::SetDateTimeSpan

Other Resources

COleDateTimeSpan Members

COleDateTimeSpan::GetStatus

Gets the status (validity) of this **COleDateTimeSpan** object.

```
DateTimeSpanStatus GetStatus( ) const throw( );
```

Return Value

The status of this **COleDateTimeSpan** value.

Remarks

The return value is defined by the **DateTimeSpanStatus** enumerated type, which is defined within the **COleDateTimeSpan** class.

```
enum DateTimeSpanStatus{
   valid = 0,
   invalid = 1,
   null = 2,
};
```

For a brief description of these status values, see the following list:

- **COleDateTimeSpan::valid** Indicates that this **COleDateTimeSpan** object is valid.
- **COleDateTimeSpan::invalid** Indicates that this **COleDateTimeSpan** object is invalid; that is, its value may be incorrect.
- **COleDateTimeSpan::null** Indicates that this **COleDateTimeSpan** object is null, that is, that no value has been supplied for this object. (This is "null" in the database sense of "having no value," as opposed to the C++ **NULL**.)

The status of a **COleDateTimeSpan** object is invalid in the following cases:

- If this object has experienced an overflow or underflow during an arithmetic assignment operation, namely, **+=** or **-=**.
- If an invalid value was assigned to this object.
- If the status of this object was explicitly set to invalid using **SetStatus**.

For more information about the operations that may set the status to invalid, see COleDateTimeSpan::operator +, - and COleDateTimeSpan::operator +=, -=.

For more information about the bounds for **COleDateTimeSpan** values, see the article Date and Time: Automation Support.

See Also

Reference

COleDateTimeSpan Class
Hierarchy Chart
COleDateTimeSpan::SetStatus
COleDateTimeSpan::m_status

Other Resources

COleDateTimeSpan Members

COleDateTimeSpan::GetTotalDays

Retrieves this date/time-span value expressed in days.

```
double GetTotalDays( ) const throw( );
```

Return Value

This date/time-span value expressed in days. Although this function is prototyped to return a double, it will always return an integer value.

Remarks

The return values from this function range between approximately – 3.65e6 and 3.65e6.

For other functions that query the value of a **COleDateTimeSpan** object, see the following member functions:

- GetDays
- GetHours
- GetMinutes
- GetSeconds
- GetTotalHours
- GetTotalMinutes
- GetTotalSeconds

Example

```
// example for COleDateTimeSpan::GetTotalXxx functions
COleDateTimeSpan ts( 3, 1, 5, 12 ); // 3 days, 1 hour, 5 min, and 12 sec
ASSERT( ts.GetTotalDays() == 3 );
ASSERT( ts.GetTotalHours() == 73 );
ASSERT( ts.GetTotalMinutes() == 4385 );
ASSERT( ts.GetTotalSeconds() == 263112 );
```

See Also

Reference

COleDateTimeSpan Class
Hierarchy Chart
COleDateTimeSpan::SetDateTimeSpan
COleDateTimeSpan::operator double

Other Resources

COleDateTimeSpan Members

COleDateTimeSpan::GetTotalHours

Retrieves this date/time-span value expressed in hours.

```
double GetTotalHours( ) const throw( );
```

Return Value

This date/time-span value expressed in hours. Although this function is prototyped to return a double, it will always return an integer value.

Remarks

The return values from this function range between approximately – 8.77e7 and 8.77e7.

For other functions that query the value of a **COleDateTimeSpan** object, see the following member functions:

- GetDays
- GetHours
- GetMinutes
- GetSeconds
- GetTotalDays
- GetTotalMinutes
- GetTotalSeconds

Example

See the example for GetTotalDays.

See Also

Reference

COleDateTimeSpan Class
Hierarchy Chart
COleDateTimeSpan::SetDateTimeSpan

Other Resources

COleDateTimeSpan Members

COleDateTimeSpan::GetTotalMinutes

Retrieves this date/time-span value expressed in minutes.

```
double GetTotalMinutes( ) const throw( );
```

Return Value

This date/time-span value expressed in minutes. Although this function is prototyped to return a double, it will always return an integer value.

Remarks

The return values from this function range between approximately – 5.26e9 and 5.26e9.

For other functions that query the value of a **COleDateTimeSpan** object, see the following member functions:

- GetDays
- GetHours
- GetMinutes
- GetSeconds
- GetTotalDays
- GetTotalHours
- GetTotalSeconds

Example

See the example for GetTotalDays.

See Also

Reference

COleDateTimeSpan Class
Hierarchy Chart
COleDateTimeSpan::SetDateTimeSpan

Other Resources

COleDateTimeSpan Members

COleDateTimeSpan::GetTotalSeconds

Retrieves this date/time-span value expressed in seconds.

```
double GetTotalSeconds( ) const throw( );
```

Return Value

This date/time-span value expressed in seconds. Although this function is prototyped to return a double, it will always return an integer value.

Remarks

The return values from this function range between approximately – 3.16e11 to 3.16e11.

For other functions that query the value of a **COleDateTimeSpan** object, see the following member functions:

- GetDays
- GetHours
- GetMinutes
- GetSeconds
- GetTotalDays
- GetTotalHours
- GetTotalMinutes

Example

See the example for GetTotalDays.

See Also

Reference

COleDateTimeSpan Class
Hierarchy Chart
COleDateTimeSpan::SetDateTimeSpan

Other Resources

COleDateTimeSpan Members

COleDateTimeSpan::SetDateTimeSpan

Sets the value of this date/time-span value.

```
void SetDateTimeSpan(
   LONG lDays,
   int nHours,
   int nMins,
   int nSecs
) throw( );
```

Parameters

lDays, *nHours*, *nMins*, *nSecs*

Indicate the date-span and time-span values to be copied into this **COleDateTimeSpan** object.

Remarks

For functions that query the value of a **COleDateTimeSpan** object, see the following member functions:

- GetDays
- GetHours
- GetMinutes
- GetSeconds
- GetTotalDays
- GetTotalHours
- GetTotalMinutes
- GetTotalSeconds

Example

```
COleDateTimeSpan spanOne;
COleDateTimeSpan spanTwo;
spanOne.SetDateTimeSpan(0, 2, 45, 0);  // 2 hours and 45 seconds
spanTwo.SetDateTimeSpan(0, 3, -15, 0); // 2 hours and 45 seconds
```

See Also

Reference

COleDateTimeSpan Class
Hierarchy Chart
COleDateTimeSpan::GetStatus
COleDateTimeSpan::m_span

Other Resources

COleDateTimeSpan Members

COleDateTimeSpan::SetStatus

Sets the status (validity) of this **COleDateTimeSpan** object.

```
void SetStatus(
   DateTimeSpanStatus status
) throw( );
```

Parameters

status

The new status value for this **COleDateTimeSpan** object.

Remarks

The *Status* parameter value is defined by the **DateTimeSpanStatus** enumerated type, which is defined within the **COleDateTimeSpan** class.

```
enum DateTimeSpanStatus{
   valid = 0,
   invalid = 1,
   null = 2,
};
```

For a brief description of these status values, see the following list:

- **COleDateTimeSpan::valid** Indicates that this **COleDateTimeSpan** object is valid.
- **COleDateTimeSpan::invalid** Indicates that this **COleDateTimeSpan** object is invalid; that is, its value may be incorrect.
- **COleDateTimeSpan::null** Indicates that this **COleDateTimeSpan** object is null, that is, that no value has been supplied for this object. (This is "null" in the database sense of "having no value," as opposed to the C++ **NULL**.)

Caution

This function is for advanced programming situations. This function does not alter the data in this object. It will most often be used to set the status to **null** or **invalid**. Note that the assignment operator (operator =) and SetDateTimeSpan do set the status of the object based on the source value(s).

Example

```
// if the person is still in school, set date
// of graduation to null
if (m_bStillInSchool || m_dtDateOfGraduation.GetStatus() == COleDateTime::null)
   m_dtDateOfGraduation.SetStatus(COleDateTimeSpan::null);
```

See Also

Reference

COleDateTimeSpan Class
Hierarchy Chart
COleDateTimeSpan::GetStatus
COleDateTimeSpan::m_status

Other Resources

COleDateTimeSpan Members

COleDateTimeSpan Operators

For information about the operators in **COleDateTimeSpan**, see COleDateTimeSpan Members.

COleDateTimeSpan::operator =

Copies a **COleDateTimeSpan** value.

```
COleDateTimeSpan& operator =(
   double dblSpanSrc
) throw( );
```

Remarks

This overloaded assignment operator copies the source date/time-span value into this **COleDateTimeSpan** object.

See Also

Reference

COleDateTimeSpan Class
Hierarchy Chart
COleDateTimeSpan::COleDateTimeSpan

Other Resources

COleDateTimeSpan Members

COleDateTimeSpan::operator +, -

Add, subtract, and change sign for **COleDateTimeSpan** values.

```
COleDateTimeSpan operator +(
   const COleDateTimeSpan& dateSpan
) const throw( );
COleDateTimeSpan operator -(
   const COleDateTimeSpan& dateSpan
) const throw( );
COleDateTimeSpan operator -( ) const throw( );
```

Remarks

The first two operators let you add and subtract date/time-span values. The third lets you change the sign of a date/time-span value.

If either of the operands is null, the status of the resulting **COleDateTimeSpan** value is null.

If either of the operands is invalid and the other is not null, the status of the resulting **COleDateTimeSpan** value is invalid.

For more information on the valid, invalid, and null status values, see the m_status member variable.

Example

```
COleDateTimeSpan ts1( 3, 1, 5, 12 ); // 3 days, 1 hour, 5 min, and 12 sec
COleDateTimeSpan ts2( 100.0 / (24 * 3600.0) ); // 100 seconds
COleDateTimeSpan ts3 = ts1 + ts2;
ASSERT( ts3.GetSeconds() == 52 ); // 6 mins, 52 secs
```

See Also

Reference

COleDateTimeSpan Class
Hierarchy Chart
COleDateTimeSpan::operator +=, -=

Other Resources

COleDateTimeSpan Members

COleDateTimeSpan::operator +=, -=

Add and subtract a **COleDateTimeSpan** value from this **COleDateTimeSpan** value.

```
COleDateTimeSpan& operator +=(
   const COleDateTimeSpan dateSpan
) throw( );
COleDateTimeSpan& operator -=(
   const COleDateTimeSpan dateSpan
) throw( );
```

Remarks

These operators let you add and subtract date/time-span values from this **COleDateTimeSpan** object. If either of the operands is null, the status of the resulting **COleDateTimeSpan** value is null.

If either of the operands is invalid and the other is not null, the status of the resulting **COleDateTimeSpan** value is invalid.

For more information on the valid, invalid, and null status values, see the m_status member variable.

Example

```
// example for COleDateTimeSpan::operator +=, -=
COleDateTimeSpan ts1( 10.0 ); // 10 days
COleDateTimeSpan ts2( 100.0 ); // 100 days
ts2 -= ts1;
ASSERT( ts2.GetTotalDays() == 90 );
```

See Also

Reference

COleDateTimeSpan Class

Hierarchy Chart

COleDateTimeSpan::operator +, -

Other Resources

COleDateTimeSpan Members

COleDateTimeSpan::operator double

Converts this **COleDateTimeSpan** value to a **double**.

```
operator double( ) const throw( );
```

Remarks

This operator returns the value of this **COleDateTimeSpan** value as a floating-point number of days.

See Also

Reference

COleDateTimeSpan Class
Hierarchy Chart
COleDateTimeSpan::GetTotalDays
COleDateTimeSpan::SetDateTimeSpan
COleDateTimeSpan::m_span

Other Resources

COleDateTimeSpan Members

COleDateTimeSpan Relational Operators

Comparison operators.

```
bool operator ==(
   const COleDateTimeSpan& dateSpan
) const throw( );
bool operator !=(
   const COleDateTimeSpan& dateSpan
) const throw( );
bool operator <(
   const COleDateTimeSpan& dateSpan
) const throw( );
bool operator >(
   const COleDateTimeSpan& dateSpan
) const throw( );
bool operator <=(
   const COleDateTimeSpan& dateSpan
) const throw( );
bool operator >=(
   const COleDateTimeSpan& dateSpan
) const throw( );
```

Parameters

dateSpan

The **COleDateTimeSpan** to compare.

Return Value

These operators compare two date/time-span values and return **true** if the condition is true; otherwise **false**.

Remarks

Note

An ATLASSERT will occur if either operand is invalid.

Example

```
COleDateTimeSpan spanOne(3, 12, 0, 0); // 3 days and 12 hours
COleDateTimeSpan spanTwo(spanOne);     // 3 days and 12 hours
BOOL b;
b = spanOne == spanTwo;                // TRUE

spanTwo.SetStatus(COleDateTimeSpan::invalid);
b = spanOne == spanTwo;                // FALSE, different status
b = spanOne != spanTwo;                // TRUE, different status
b = spanOne < spanTwo;                 // FALSE, same value
b = spanOne > spanTwo;                 // FALSE, same value
b = spanOne <= spanTwo;                // TRUE, same value
b = spanOne >= spanTwo;                // TRUE, same value
```

Note

The last four lines of the preceding example will **ASSERT** in debug mode.

```
COleDateTimeSpan ts1(100.0);   // one hundred days
COleDateTimeSpan ts2(110.0);   // ten more days

ASSERT( (ts1 != ts2) && (ts1 < ts2) && (ts1 <= ts2) );
```

See Also

Reference

COleDateTimeSpan Class

Hierarchy Chart

Other Resources

COleDateTimeSpan Members

COleDateTimeSpan Data Members

For information about the data members in **COleDateTimeSpan**, see COleDateTimeSpan Members.

COleDateTimeSpan::m_span

The underlying **double** value for this **COleDateTime** object.

```
double m_span;
```

Remarks

This value expresses the date/time-span in days.

Caution
Changing the value in the **double** data member will change the value of this **COleDateTimeSpan** object. It does not change the status of this **COleDateTimeSpan** object.

See Also

Reference

COleDateTimeSpan Class
Hierarchy Chart
COleDateTimeSpan::COleDateTimeSpan
COleDateTimeSpan::SetDateTimeSpan
COleDateTimeSpan::operator double

Other Resources

COleDateTimeSpan Members

COleDateTimeSpan::m_status

The type for this data member is the enumerated type **DateTimeSpanStatus**, which is defined within the **COleDateTimeSpan** class.

```
DateTimeSpanStatus m_status;
```

Remarks

```
enum DateTimeSpanStatus{
   valid = 0,
   invalid = 1,
   null = 2,
};
```

For a brief description of these status values, see the following list:

- **COleDateTimeSpan::valid** Indicates that this **COleDateTimeSpan** object is valid.
- **COleDateTimeSpan::invalid** Indicates that this **COleDateTimeSpan** object is invalid; that is, its value may be incorrect.
- **COleDateTimeSpan::null** Indicates that this **COleDateTimeSpan** object is null, that is, that no value has been supplied for this object. (This is "null" in the database sense of "having no value," as opposed to the C++ **NULL**.)

The status of a **COleDateTimeSpan** object is invalid in the following cases:

- If this object has experienced an overflow or underflow during an arithmetic assignment operation, namely, **+=** or **-=**.
- If an invalid value was assigned to this object.
- If the status of this object was explicitly set to invalid using SetStatus.

For more information about the operations that may set the status to invalid, see COleDateTimeSpan::operator +, - and COleDateTimeSpan::operator +=, -=.

Caution
This data member is for advanced programming situations. You should use the inline member functions GetStatus and SetStatus. See **SetStatus** for further cautions regarding explicitly setting this data member.

For more information about the bounds for **COleDateTimeSpan** values, see the article Date and Time: Automation Support.

See Also

Reference

COleDateTimeSpan Class
Hierarchy Chart
COleDateTimeSpan::GetStatus
COleDateTimeSpan::SetStatus

Other Resources

COleDateTimeSpan Members

CPoint Class

Similar to the Windows **POINT** structure.

```
class CPoint : public tagPOINT
```

Remarks

It also includes member functions to manipulate **CPoint** and POINT structures.

A **CPoint** object can be used wherever a **POINT** structure is used. The operators of this class that interact with a "size" accept either CSize objects or SIZE structures, since the two are interchangeable.

Note

This class is derived from the **tagPOINT** structure. (The name **tagPOINT** is a less commonly used name for the **POINT** structure.) This means that the data members of the **POINT** structure, **x** and **y**, are accessible data members of **CPoint**.

Note

For more information on shared utility classes (like **CPoint**), see Shared Classes.

Requirements

Header: atltypes.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

MDI Sample: Demonstrates MDI Without Using Doc/View Architecture

Reference

Hierarchy Chart

CRect Class

CSize Class

Other Resources

CPoint Members

CPoint Members

Construction

CPoint	Constructs a **CPoint**.

Operations

Offset	Adds values to the **x** and **y** members of the **CPoint**.
operator !=	Checks for inequality between two points.
operator ==	Checks for equality between two points.

Operators Returning CPoint Values

operator –	Returns the difference of a **CPoint** and a size, or the negation of a point.
operator +	Returns the sum of a **CPoint** and a size or point.
operator +=	Offsets **CPoint** by adding a size or point.
operator –=	Offsets **CPoint** by subtracting a size or point.

Operators Returning CSize Values

operator –	Returns the size difference between two points.

Operators Returning CRect Values

operator –	Returns a **CRect** offset by a negative size.
operator +	Returns a **CRect** offset by a size.

See Also

Reference

CPoint Class

Hierarchy Chart

CPoint Member Functions

For information about the member functions in **CPoint**, see CPoint Members.

CPoint::CPoint

Constructs a **CPoint** object.

```
CPoint( ) throw( );
CPoint(
   int initX,
   int initY
) throw( );
CPoint(
   POINT initPt
) throw( );
CPoint(
   SIZE initSize
) throw( );
CPoint(
   LPARAM dwPoint
) throw( );
```

Parameters

initX

Specifies the value of the **x** member of **CPoint**.

initY

Specifies the value of the **y** member of **CPoint**.

initPt

POINT structure or **CPoint** that specifies the values used to initialize **CPoint**.

initSize

SIZE structure or CSize that specifies the values used to initialize **CPoint**.

dwPoint

Sets the **x** member to the low-order word of *dwPoint* and the **y** member to the high-order word of *dwPoint*.

Remarks

If no arguments are given, **x** and **y** members are not initialized.

Example

```
CPoint   ptUndefined;
CPoint   ptTopLeft(0,0);

// works from a POINT, too

POINT   ptHere;
ptHere.x = 35;
ptHere.y = 95;

CPoint   ptMFCHere(ptHere);

// works from A SIZE

SIZE   sHowBig;
sHowBig.cx = 300;
sHowBig.cy = 10;

CPoint ptMFCBig(sHowBig);

// or from a DWORD
```

```
DWORD   dwSize;
dwSize = MAKELONG(35, 95);

CPoint ptFromDouble(dwSize);
ASSERT(ptFromDouble == ptMFCHere);
```

See Also

Reference

CPoint Class

Hierarchy Chart

Other Resources

CPoint Members

CPoint::Offset

Adds values to the **x** and **y** members of the **CPoint**.

```
void Offset(
   int xOffset,
   int yOffset
) throw( );
void Offset(
   POINT point
) throw( );
void Offset(
   SIZE size
) throw( );
```

Parameters

xOffset

Specifies the amount to offset the **x** member of the **CPoint**.

yOffset

Specifies the amount to offset the **y** member of the **CPoint**.

point

Specifies the amount (POINT or **CPoint**) to offset the **CPoint**.

size

Specifies the amount (SIZE or CSize) to offset the **CPoint**.

Example

```
CPoint   ptStart(100, 100);
ptStart.Offset(35, 35);

CPoint   ptResult(135, 135);
ASSERT(ptStart == ptResult);

// works with POINT, too

ptStart = CPoint(100, 100);
POINT pt;

pt.x = 35;
pt.y = 35;

ptStart.Offset(pt);

ASSERT(ptStart == ptResult);

// works with SIZE, too

ptStart = CPoint(100, 100);
SIZE size;

size.cx = 35;
size.cy = 35;

ptStart.Offset(size);

ASSERT(ptStart == ptResult);
```

See Also

Reference

CPoint Class

Hierarchy Chart

CPoint::operator +=

CPoint::operator -=

Other Resources

CPoint Members

CPoint Operators

For information about the operators in **CPoint**, see CPoint Members.

CPoint::operator ==

Checks for equality between two points.

```
BOOL operator ==(
   POINT point
) const throw( );
```

Parameters

point

Contains a POINT structure or **CPoint** object.

Return Value

Nonzero if the points are equal; otherwise 0.

Example

```
CPoint ptFirst(256, 128);
CPoint ptTest(256, 128);

ASSERT(ptFirst == ptTest);

// works with POINTs, too

POINT pt;
pt.x = 256;
pt.y = 128;

ASSERT(ptTest == pt);

// note that pt == ptTest isn't correct!
```

See Also

Reference

CPoint Class

Hierarchy Chart

CPoint::operator !=

Other Resources

CPoint Members

CPoint::operator !=

Checks for inequality between two points.

```
BOOL operator!=(
   POINT point
) const throw( );
```

Parameters

point

Contains a POINT structure or **CPoint** object.

Return Value

Nonzero if the points are not equal; otherwise 0.

Example

```
CPoint ptFirst(256, 128);
CPoint ptTest(111, 333);

ASSERT(ptFirst != ptTest);

// works with POINTs, too

POINT pt;
pt.x = 333;
pt.y = 111;

ASSERT(ptTest != pt);

// note that pt != ptTest isn't correct!
```

See Also

Reference

CPoint Class

Hierarchy Chart

CPoint::operator ==

Other Resources

CPoint Members

CPoint::operator +=

The first overload adds a size to the **CPoint**.

```
void operator +=(
   SIZE size
) throw( );
void operator +=(
   POINT point
) throw( );
```

Parameters

size

Contains a SIZE structure or CSize object.

point

Contains a POINT structure or CPoint object.

Remarks

The second overload adds a point to the **CPoint**.

In both cases, addition is done by adding the **x** (or **cx**) member of the right-hand operand to the **x** member of the **CPoint** and adding the **y** (or **cy**) member of the right-hand operand to the **y** member of the **CPoint**.

For example, adding `CPoint(5, -7)` to a variable which contains `CPoint(30, 40)` changes the variable to `CPoint(35, 33)`.

Example

```
CPoint   ptStart(100, 100);
CSize   szOffset(35, 35);

ptStart += szOffset;

CPoint   ptResult(135, 135);

ASSERT(ptResult == ptStart);

// also works on SIZE

ptStart = CPoint(100, 100);

SIZE   sz;
sz.cx = 35;
sz.cy = 35;

ptStart += sz;

ASSERT(ptResult == ptStart);
```

See Also

Reference

CPoint Class
Hierarchy Chart
CPoint::operator -=
CPoint::operator +
CPoint::Offset

Other Resources

CPoint Members

CPoint::operator -=

The first overload subtracts a size from the **CPoint**.

```
void operator -=(
   SIZE size
) throw( );
void operator -=(
   POINT point
) throw( );
```

Parameters

size

Contains a SIZE structure or CSize object.

point

Contains a POINT structure or CPoint object.

Remarks

The second overload subtracts a point from the **CPoint**.

In both cases, subtraction is done by subtracting the **x** (or **cx**) member of the right-hand operand from the **x** member of the **CPoint** and subtracting the **y** (or **cy**) member of the right-hand operand from the **y** member of the **CPoint**.

For example, subtracting `CPoint(5, -7)` from a variable which contains `CPoint(30, 40)` changes the variable to `CPoint(25, 47)`.

Example

```
CPoint   ptStart(100, 100);
CSize   szOffset(35, 35);

ptStart -= szOffset;

CPoint   ptResult(65, 65);

ASSERT(ptResult == ptStart);

// also works on SIZE

ptStart = CPoint(100, 100);

SIZE   sz;
sz.cx = 35;
sz.cy = 35;

ptStart -= sz;

ASSERT(ptResult == ptStart);
```

See Also

Reference

CPoint Class
Hierarchy Chart
CPoint::operator -
CPoint::operator +=
CPoint::Offset

Other Resources

CPoint Members

CPoint::operator +

Use this operator to offset **CPoint** by a **CPoint** or **CSize** object, or to offset a **CRect** by a **CPoint**.

```
CPoint operator +(
   SIZE size
) const throw( );
CPoint operator +(
   POINT point
) const throw( );
CRect operator +(
   const RECT* lpRect
) const throw( );
```

Parameters

size

Contains a SIZE structure or CSize object.

point

Contains a POINT structure or CPoint object.

lpRect

Contains a pointer to a RECT structure or CRect object.

Return Value

A **CPoint** that is offset by a size, a **CPoint** that is offset by a point, or a **CRect** offset by a point.

Remarks

For example, using one of the first two overloads to offset the point `CPoint(25, -19)` by a point `CPoint(15, 5)` or size `CSize(15, 5)` returns the value `CPoint(40, -14)`.

Adding a rectangle to a point returns the rectangle after being offset by the **x** and **y** values specified in the point. For example, using the last overload to offset a rectangle `CRect(125, 219, 325, 419)` by a point `CPoint(25, -19)` returns `CRect(150, 200, 350, 400)`.

Example

```
CPoint   ptStart(100, 100);
CSize   szOffset(35, 35);
CPoint   ptEnd;

ptEnd = ptStart + szOffset;

CPoint   ptResult(135, 135);

ASSERT(ptResult == ptEnd);

// also works on SIZE

ptStart = CPoint(100, 100);

SIZE   sz;
sz.cx = 35;
sz.cy = 35;

ptEnd = ptStart + sz;

ASSERT(ptResult == ptEnd);
```

See Also

Reference

CPoint Class
Hierarchy Chart
CPoint::operator -=
CPoint::operator -
CPoint::operator +=
CSize::operator +
CRect::operator +
CPoint::Offset
CRect::OffsetRect

Other Resources

CPoint Members

CPoint::operator -

Use one of the first two overloads to subtract a **CPoint** or **CSize** object from **CPoint**.

```
CSize operator -(
   POINT point
) const throw( );
CPoint operator -(
   SIZE size
) const throw( );
CRect operator -(
   const RECT* lpRect
) const throw( );
CPoint operator -( ) const throw( );
```

Parameters

point

A POINT structure or CPoint object.

size

A SIZE structure or CSize object.

lpRect

A pointer to a RECT structure or a CRect object.

Return Value

A **CSize** that is the difference between two points, a **CPoint** that is offset by the negation of a size, a **CRect** that is offset by the negation of a point, or a **CPoint** that is the negation of a point.

Remarks

The third overload offsets a **CRect** by the negation of **CPoint**. Finally, use the unary operator to negate **CPoint**.

For example, using the first overload to find the difference between two points `CPoint(25, -19)` and `CPoint(15, 5)` returns `CSize(10, -24)`.

Subtracting a **CSize** from **CPoint** does the same calculation as above but returns a **CPoint** object, not a **CSize** object. For example, using the second overload to find the difference between the point `CPoint(25, -19)` and the size `CSize(15, 5)` returns `CPoint(10, -24)`.

Subtracting a rectangle from a point returns the rectangle offset by the negatives of the **x** and **y** values specified in the point. For example, using the last overload to offset the rectangle `CRect(125, 200, 325, 400)` by the point `CPoint(25, -19)` returns `CRect(100, 219, 300, 419)`.

Use the unary operator to negate a point. For example, using the unary operator with the point `CPoint(25, -19)` returns `CPoint(-25, 19)`.

Example

```
// example for CPoint subtraction
CPoint   ptStart(100, 100);
CSize   szOffset(35, 35);
CPoint   ptEnd;

ptEnd = ptStart - szOffset;

CPoint   ptResult(65, 65);

ASSERT(ptResult == ptEnd);

// also works on SIZE

ptStart = CPoint(100, 100);
```

```
SIZE   sz;
sz.cx = 35;
sz.cy = 35;

ptEnd = ptStart - sz;

ASSERT(ptResult == ptEnd);

// example for CPoint unary operator
CPoint   pt(35, 35);
pt = -pt;

CPoint ptResult(-35, -35);
ASSERT(pt == ptResult);
```

See Also

Reference

CPoint Class
Hierarchy Chart
CPoint::operator -=
CPoint::operator +=
CPoint::operator +
CSize::operator -
CRect::operator -
CPoint::Offset
CRect::OffsetRect

Other Resources

CPoint Members

CRect Class

Similar to a Windows RECT structure.

```
class CRect : public tagRECT
```

Remarks

CRect also includes member functions to manipulate **CRect** objects and Windows **RECT** structures.

A **CRect** object can be passed as a function parameter wherever a **RECT** structure, **LPCRECT**, or **LPRECT** can be passed.

Note
This class is derived from the **tagRECT** structure. (The name **tagRECT** is a less-commonly-used name for the **RECT** structure.) This means that the data members (**left**, **top**, **right**, and **bottom**) of the **RECT** structure are accessible data members of **CRect**.

A **CRect** contains member variables that define the top-left and bottom-right points of a rectangle.

When specifying a **CRect**, you must be careful to construct it so that it is normalized — in other words, such that the value of the left coordinate is less than the right and the top is less than the bottom. For example, a top left of (10,10) and bottom right of (20,20) defines a normalized rectangle but a top left of (20,20) and bottom right of (10,10) defines a non-normalized rectangle. If the rectangle is not normalized, many **CRect** member functions may return incorrect results. (See CRect::NormalizeRect for a list of these functions.) Before you call a function that requires normalized rectangles, you can normalize non-normalized rectangles by calling the **NormalizeRect** function.

Use caution when manipulating a **CRect** with the CDC::DPtoLP and CDC::LPtoDP member functions. If the mapping mode of a display context is such that the y-extent is negative, as in **MM_LOENGLISH**, then **CDC::DPtoLP** will transform the **CRect** so that its top is greater than the bottom. Functions such as **Height** and **Size** will then return negative values for the height of the transformed **CRect**, and the rectangle will be non-normalized.

When using overloaded **CRect** operators, the first operand must be a **CRect**; the second can be either a RECT structure or a **CRect** object.

Note
For more information on shared utility classes (like **CRect**), see Shared Classes.

Requirements

Header: atltypes.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart
CPoint Class
CSize Class
RECT Structure

Other Resources

CRect Members

CRect Members

Construction/Destruction

Operations

Operators

Construction

CRect	Constructs a **CRect** object.

Operations

BottomRight	Returns the bottom-right point of **CRect**.
CenterPoint	Returns the centerpoint of **CRect**.
CopyRect	Copies the dimensions of a source rectangle to **CRect**.
DeflateRect	Decreases the width and height of **CRect**.
EqualRect	Determines whether **CRect** is equal to the given rectangle.
Height	Calculates the height of **CRect**.
InflateRect	Increases the width and height of **CRect**.
IntersectRect	Sets **CRect** equal to the intersection of two rectangles.
IsRectEmpty	Determines whether **CRect** is empty. **CRect** is empty if the width and/or height are 0.
IsRectNull	Determines whether the **top**, **bottom**, **left**, and **right** member variables are all equal to 0.
MoveToX	Moves **CRect** to the specified x-coordinate.
MoveToXY	Moves **CRect** to the specified x- and y-coordinates.
MoveToY	Moves **CRect** to the specified y-coordinate.
NormalizeRect	Standardizes the height and width of **CRect**.
OffsetRect	Moves **CRect** by the specified offsets.
PtInRect	Determines whether the specified point lies within **CRect**.
SetRect	Sets the dimensions of **CRect**.
SetRectEmpty	Sets **CRect** to an empty rectangle (all coordinates equal to 0).
Size	Calculates the size of **CRect**.
SubtractRect	Subtracts one rectangle from another.
TopLeft	Returns the top-left point of **CRect**.

UnionRect	Sets **CRect** equal to the union of two rectangles.
Width	Calculates the width of **CRect**.

Operators

operator !=	Determines whether **CRect** is not equal to a rectangle.
operator &	Creates the intersection of **CRect** and a rectangle and returns the resulting **CRect**.
operator &=	Sets **CRect** equal to the intersection of **CRect** and a rectangle.
operator –	Subtracts the given offsets from **CRect** or deflates **CRect** and returns the resulting **CRect**.
operator |	Creates the union of **CRect** and a rectangle and returns the resulting **CRect**.
operator |=	Sets **CRect** equal to the union of **CRect** and a rectangle.
operator +	Adds the given offsets to **CRect** or inflates **CRect** and returns the resulting **CRect**.
operator +=	Adds the specified offsets to **CRect** or inflates **CRect**.
operator =	Copies the dimensions of a rectangle to **CRect**.
operator –=	Subtracts the specified offsets from **CRect** or deflates **CRect**.
operator ==	Determines whether **CRect** is equal to a rectangle.
operator LPCRECT	Converts a **CRect** to an **LPCRECT**.
operator LPRECT	Converts a **CRect** to an **LPRECT**.

See Also

Reference

CRect Class

Hierarchy Chart

CRect Member Functions

For information about the member functions in **CRect**, see CRect Members.

CRect::BottomRight

The coordinates are returned as a reference to a CPoint object that is contained in **CRect**.

```
CPoint& BottomRight( ) throw( );
const CPoint& BottomRight( ) const throw( );
```

Return Value

The coordinates of the bottom-right corner of the rectangle.

Remarks

You can use this function to either get or set the bottom-right corner of the rectangle. Set the corner by using this function on the left side of the assignment operator.

Example

```
// use BottomRight() to retreive the bottom
// right point

CRect rect(210, 150, 350, 900);
CPoint ptDown;

ptDown = rect.BottomRight();

// ptDown is now set to (350, 900)
ASSERT(ptDown == CPoint(350, 900));

// or, use BottomRight() to set the bottom
// right point

CRect rect2(10, 10, 350, 350);
CPoint ptLow(180, 180);

rect2.BottomRight() = ptLow;

// rect2 is now (10, 10, 180, 180)
ASSERT(rect2 == CRect(10, 10, 180, 180));
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::TopLeft
CPoint Class
CRect::CenterPoint

Other Resources

CRect Members

CRect::CenterPoint

Calculates the centerpoint of **CRect** by adding the left and right values and dividing by two, and adding the top and bottom values and dividing by two.

```
CPoint CenterPoint( ) const throw( );
```

Return Value

A **CPoint** object that is the centerpoint of **CRect**.

Example

```
// Code from this OnDraw() implementation can be pasted into your own application
// to draw lines that would look like a letter "Y" within your view.

void CYourView::OnDraw(CDC* pDC)
{
   // get the size and position of the client area of
   // your window

   CRect rect;
   GetWindowRect(&rect);

   // Move the current pen to the top left of the window. We call the
   // TopLeft() member of CRect here and it returns a CPoint object we
   // pass to the override of CDC::MoveTo() that accepts a CPoint.

   pDC->MoveTo(rect.TopLeft());

   // Draw a line from the top left to the center of the window.
   // CenterPoint() gives us the middle point of the window as a
   // CPoint, and since CDC::LineTo() has an override that accepts a
   // CPoint, we can just pass it along.

   pDC->LineTo(rect.CenterPoint());

   // Now, draw a line to the top right of the window. There's no
   // CRect member which returns a CPoint for the top right of the
   // window, so we'll reference the CPoint members directly and call
   // the CDC::LineTo() override which takes two integers.

   pDC->LineTo(rect.right, rect.top);

   // The top part of the "Y" is drawn. Now, we'll draw the stem. We
   // start from the center point.

   pDC->MoveTo(rect.CenterPoint());

   // and then draw to the middle of the bottom edge of the window.
   // We'll get the x-coordinate from the x member of the CPoint
   // returned by CenterPoint(), and the y value comes directly from
   // the rect.

   pDC->LineTo(rect.CenterPoint().x, rect.bottom);
}
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::Width
CRect::Height

CRect::Size
CRect::TopLeft
CRect::BottomRight
CRect::IsRectNull
CPoint Class

Other Resources

CRect Members

CRect::CopyRect

Copies the *lpSrcRect* rectangle into **CRect**.

```
void CopyRect(
   LPCRECT lpSrcRect
) throw( );
```

Parameters

lpSrcRect

Points to the RECT structure or **CRect** object that is to be copied.

Example

```
CRect rectSource(35, 10, 125, 10);
CRect rectDest;

rectDest.CopyRect(&rectSource);

// rectDest is now set to (35, 10, 125, 10)

RECT rectSource2;
rectSource2.left = 0;
rectSource2.top = 0;
rectSource2.bottom = 480;
rectSource2.right = 640;

rectDest.CopyRect(&rectSource2);

// works against RECT structures, too!
// rectDest is now set to (0, 0, 640, 480)
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::CRect
CRect::operator =
CRect::SetRect
CRect::SetRectEmpty

Other Resources

CRect Members

CRect::CRect

Constructs a **CRect** object.

```
CRect( ) throw( );
CRect(
   int l,
   int t,
   int r,
   int b
) throw( );
CRect(
   const RECT& srcRect
) throw( );
CRect(
   LPCRECT lpSrcRect
) throw( );
CRect(
   POINT point,
   SIZE size
) throw( );
CRect(
   POINT topLeft,
   POINT bottomRight
) throw( );
```

Parameters

l

Specifies the left position of **CRect**.

t

Specifies the top of **CRect**.

r

Specifies the right position of **CRect**.

b

Specifies the bottom of **CRect**.

srcRect

Refers to the RECT structure with the coordinates for **CRect**.

lpSrcRect

Points to the **RECT** structure with the coordinates for **CRect**.

point

Specifies the origin point for the rectangle to be constructed. Corresponds to the top-left corner.

size

Specifies the displacement from the top-left corner to the bottom-right corner of the rectangle to be constructed.

topLeft

Specifies the top-left position of **CRect**.

bottomRight

Specifies the bottom-right position of **CRect**.

Remarks

If no arguments are given, **left**, **top**, **right**, and **bottom** members are not initialized.

The **CRect**(**const RECT&**) and **CRect**(**LPCRECT**) constructors perform a CopyRect. The other constructors initialize the member variables of the object directly.

Example

```
// default constructor doesn't initialize!
CRect rectUnknown;

// four-integers are left, top, right, and bottom
CRect rect(0, 0, 100, 50);
ASSERT(rect.Width() == 100);
ASSERT(rect.Height() == 50);

// Initialize from RECT stucture
RECT sdkRect;
sdkRect.left = 0;
sdkRect.top = 0;
sdkRect.right = 100;
sdkRect.bottom = 50;

CRect rect2(sdkRect);   // by reference
CRect rect3(&sdkRect);  // by address
ASSERT(rect2 == rect);
ASSERT(rect3 == rect);

// from a point and a size
CPoint pt(0, 0);
CSize sz(100, 50);
CRect rect4(pt, sz);
ASSERT(rect4 == rect2);

// from two points
CSize szTopLeft(0, 0);
CRect rect5(szTopLeft, sz);
ASSERT(rect5 == rect4);
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::SetRect
CRect::CopyRect
CRect::operator =
CRect::SetRectEmpty

Other Resources

CRect Members

CRect::DeflateRect

DeflateRect deflates **CRect** by moving its sides toward its center.

```
void DeflateRect(
   int x,
   int y
) throw( );
void DeflateRect(
   SIZE size
) throw( );
void DeflateRect(
   LPCRECT lpRect
) throw( );
void DeflateRect(
   int l,
   int t,
   int r,
   int b
) throw( );
```

Parameters

x

Specifies the number of units to deflate the left and right sides of **CRect**.

y

Specifies the number of units to deflate the top and bottom of **CRect**.

size

A SIZE or CSize that specifies the number of units to deflate **CRect**. The *cx* value specifies the number of units to deflate the left and right sides and the *cy* value specifies the number of units to deflate the top and bottom.

lpRect

Points to a RECT structure or **CRect** that specifies the number of units to deflate each side.

l

Specifies the number of units to deflate the left side of **CRect**.

t

Specifies the number of units to deflate the top of **CRect**.

r

Specifies the number of units to deflate the right side of **CRect**.

b

Specifies the number of units to deflate the bottom of **CRect**.

Remarks

To do this, **DeflateRect** adds units to the left and top and subtracts units from the right and bottom. The parameters of **DeflateRect** are signed values; positive values deflate **CRect** and negative values inflate it.

The first two overloads deflate both pairs of opposite sides of **CRect** so that its total width is decreased by two times *x* (or *cx*) and its total height is decreased by two times *y* (or *cy*). The other two overloads deflate each side of **CRect** independently of the others.

Example

```
CRect rect(10, 10, 50, 50);
```

```
rect.DeflateRect(1, 2);

ASSERT(rect.left == 11 && rect.right == 49);
ASSERT(rect.top == 12 && rect.bottom == 48);

CRect rect2(10, 10, 50, 50);
CRect rectDeflate(1, 2, 3, 4);

rect2.DeflateRect(&rectDeflate);
ASSERT(rect2.left == 11 && rect2.right == 47);
ASSERT(rect2.top == 12 && rect2.bottom == 46);
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::InflateRect
CRect::operator -
CRect::operator -=
InflateRect

Other Resources

CRect Members

CRect::EqualRect

Determines whether **CRect** is equal to the given rectangle.

```
BOOL EqualRect(
   LPCRECT lpRect
) const throw( );
```

Parameters

lpRect

Points to a RECT structure or **CRect** object that contains the upper-left and lower-right corner coordinates of a rectangle.

Return Value

Nonzero if the two rectangles have the same top, left, bottom, and right values; otherwise 0.

Note

Both of the rectangles must be normalized or this function may fail. You can call NormalizeRect to normalize the rectangles before calling this function.

Example

```
CRect rect1(35, 150, 10, 25);
CRect rect2(35, 150, 10, 25);
CRect rect3(98, 999,  6,  3);

ASSERT(rect1.EqualRect(rect2));
ASSERT(!rect1.EqualRect(rect3));

// works just fine against RECTs, as well

RECT test;
test.left = 35;
test.top = 150;
test.right = 10;
test.bottom = 25;

ASSERT(rect1.EqualRect(&test));
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::operator ==
CRect::operator !=
CRect::NormalizeRect
EqualRect

Other Resources

CRect Members

CRect::Height

Calculates the height of **CRect** by subtracting the top value from the bottom value.

```
int Height( ) const throw( );
```

Return Value

The height of **CRect**.

Remarks

The resulting value can be negative.

Note

The rectangle must be normalized or this function may fail. You can call NormalizeRect to normalize the rectangle before call ing this function.

Example

```
CRect rect(20, 30, 80, 70);

int nHt = rect.Height();

// nHt is now 40
ASSERT(nHt == 40);
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::Width
CRect::Size
CRect::CenterPoint
CRect::IsRectEmpty
CRect::IsRectNull
CRect::NormalizeRect

Other Resources

CRect Members

CRect::InflateRect

InflateRect inflates **CRect** by moving its sides away from its center.

```
void InflateRect(
   int x,
   int y
) throw( );
void InflateRect(
   SIZE size
) throw( );
void InflateRect(
   LPCRECT lpRect
) throw( );
void InflateRect(
   int l,
   int t,
   int r,
   int b
) throw( );
```

Parameters

x

Specifies the number of units to inflate the left and right sides of **CRect**.

y

Specifies the number of units to inflate the top and bottom of **CRect**.

size

A SIZE or CSize that specifies the number of units to inflate **CRect**. The *cx* value specifies the number of units to inflate the left and right sides and the *cy* value specifies the number of units to inflate the top and bottom.

lpRect

Points to a RECT structure or **CRect** that specifies the number of units to inflate each side.

l

Specifies the number of units to inflate the left side of **CRect**.

t

Specifies the number of units to inflate the top of **CRect**.

r

Specifies the number of units to inflate the right side of **CRect**.

b

Specifies the number of units to inflate the bottom of **CRect**.

Remarks

To do this, **InflateRect** subtracts units from the left and top and adds units to the right and bottom. The parameters of **InflateRect** are signed values; positive values inflate **CRect** and negative values deflate it.

The first two overloads inflate both pairs of opposite sides of **CRect** so that its total width is increased by two times *x* (or *cx*) and its total height is increased by two times *y* (or *cy*). The other two overloads inflate each side of **CRect** independently of the others.

Example

```
CRect rect(0, 0, 300, 300);
rect.InflateRect(50, 200);
```

```
// rect is now (-50, -200, 350, 500)
ASSERT(rect == CRect(-50, -200, 350, 500));
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::DeflateRect
CRect::operator +
CRect::operator +=
InflateRect

Other Resources

CRect Members

CRect::IntersectRect

Makes a **CRect** equal to the intersection of two existing rectangles.

```
BOOL IntersectRect(
   LPCRECT lpRect1,
   LPCRECT lpRect2
) throw( );
```

Parameters

lpRect1

Points to a RECT structure or **CRect** object that contains a source rectangle.

lpRect2

Points to a **RECT** structure or **CRect** object that contains a source rectangle.

Return Value

Nonzero if the intersection is not empty; 0 if the intersection is empty.

Remarks

The intersection is the largest rectangle contained in both existing rectangles.

Note

Both of the rectangles must be normalized or this function may fail. You can call NormalizeRect to normalize the rectangles before calling this function.

Example

```
CRect rectOne(125,   0, 150, 200);
CRect rectTwo(  0,  75, 350,  95);
CRect rectInter;

rectInter.IntersectRect(rectOne, rectTwo);

// rectInter is now (125, 75, 150, 95)

ASSERT(rectInter == CRect(125, 75, 150, 95));

// operator &= can do the same task:

CRect rectInter2 = rectOne;
rectInter2 &= rectTwo;
ASSERT(rectInter2 == CRect(125, 75, 150, 95));
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::operator &=
CRect::operator &
CRect::UnionRect
CRect::SubtractRect
CRect::NormalizeRect
IntersectRect

Other Resources

CRect Members

CRect::IsRectEmpty

Determines whether **CRect** is empty.

```
BOOL IsRectEmpty( ) const throw( );
```

Return Value

Nonzero if **CRect** is empty; 0 if **CRect** is not empty.

Remarks

A rectangle is empty if the width and/or height are 0 or negative. Differs from **IsRectNull**, which determines whether all coordinates of the rectangle are zero.

Note

The rectangle must be normalized or this function may fail. You can call NormalizeRect to normalize the rectangle before call ing this function.

Example

```
CRect rectNone(0, 0, 0, 0);
CRect rectSome(35, 50, 135, 150);

ASSERT(rectNone.IsRectEmpty());
ASSERT(!rectSome.IsRectEmpty());

CRect rectEmpty(35, 35, 35, 35);
ASSERT(rectEmpty.IsRectEmpty());
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::IsRectNull
CRect::SetRectEmpty
CRect::NormalizeRect
IsRectEmpty

Other Resources

CRect Members

CRect::IsRectNull

Determines whether the top, left, bottom, and right values of **CRect** are all equal to 0.

```
BOOL IsRectNull( ) const throw( );
```

Return Value

Nonzero if **CRect**'s top, left, bottom, and right values are all equal to 0; otherwise 0.

Remarks

Differs from **IsRectEmpty**, which determines whether the rectangle is empty.

Example

```
CRect rectNone(0, 0, 0, 0);
CRect rectSome(35, 50, 135, 150);

ASSERT(rectNone.IsRectNull());
ASSERT(!rectSome.IsRectNull());

// note that null means _all_ zeros

CRect rectNotNull(0, 0, 35, 50);
ASSERT(!rectNotNull.IsRectNull());
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::IsRectEmpty
CRect::SetRectEmpty

Other Resources

CRect Members

CRect::MoveToX

Call this function to move the rectangle to the absolute x-coordinate specified by *x*.

```
void MoveToX(
   int x
) throw( );
```

Parameters

x

The absolute x-coordinate for the upper-left corner of the rectangle.

Example

```
CRect rect(0, 0, 100, 100);

rect.MoveToX(10);

// rect is now (10, 0, 110, 100);
ASSERT(rect == CRect(10, 0, 110, 100));
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::MoveToXY
CRect::MoveToY

Other Resources

CRect Members

CRect::MoveToXY

Call this function to move the rectangle to the absolute x- and y-coordinates specified.

```
void MoveToXY(
   int x,
   int y
) throw( );
void MoveToXY(
   POINT point
) throw( );
```

Parameters

x

The absolute x-coordinate for the upper-left corner of the rectangle.

y

The absolute y-coordinate for the upper-left corner of the rectangle.

point

A **POINT** structure specifying the absolute upper-left corner of the rectangle.

Example

```
CRect rect(0, 0, 100, 100);

rect.MoveToXY(10, 10);

// rect is now (10, 10, 110, 110);
ASSERT(rect == CRect(10, 10, 110, 110));
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::MoveToX
CRect::MoveToY

Other Resources

CRect Members

CRect::MoveToY

Call this function to move the rectangle to the absolute y-coordinate specified by *y*.

```
void MoveToY(
   int y
) throw( );
```

Parameters

y

The absolute y-coordinate for the upper-left corner of the rectangle.

Example

```
CRect rect(0, 0, 100, 100);

rect.MoveToY(10);

// rect is now (0, 10, 100, 110);
ASSERT(rect == CRect(0, 10, 100, 110));
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::MoveToXY
CRect::MoveToX

Other Resources

CRect Members

CRect::NormalizeRect

Normalizes **CRect** so that both the height and width are positive.

```
void NormalizeRect( ) throw( );
```

Remarks

The rectangle is normalized for fourth-quadrant positioning, which Windows typically uses for coordinates. **NormalizeRect** compares the top and bottom values, and swaps them if the top is greater than the bottom. Similarly, it swaps the left and right values if the left is greater than the right. This function is useful when dealing with different mapping modes and inverted rectangles.

Note

The following **CRect** member functions require normalized rectangles in order to work properly: Height, Width, Size, IsRectEmpty, PtInRect, EqualRect, UnionRect, IntersectRect, SubtractRect, operator ==, operator !=, operator |, operator |=, operator &, and operator &=.

Example

```
CRect rect1(110, 100, 250, 310);
CRect rect2(250, 310, 110, 100);

rect1.NormalizeRect();
rect2.NormalizeRect();

// rect1 should be unchanged
// rect2 becomes (110, 100, 250, 310)

ASSERT(rect1 == rect2);
```

See Also

Reference

CRect Class

Hierarchy Chart

Other Resources

CRect Members

CRect::OffsetRect

Moves **CRect** by the specified offsets.

```
void OffsetRect(
   int x,
   int y
) throw( );
void OffsetRect(
   POINT point
) throw( );
void OffsetRect(
   SIZE size
) throw( );
```

Parameters

x

Specifies the amount to move left or right. It must be negative to move left.

y

Specifies the amount to move up or down. It must be negative to move up.

point

Contains a POINT structure or CPoint object specifying both dimensions by which to move.

size

Contains a SIZE structure or CSize object specifying both dimensions by which to move.

Remarks

Moves **CRect** *x* units along the x-axis and *y* units along the y-axis. The *x* and *y* parameters are signed values, so **CRect** can be moved left or right and up or down.

Example

```
CRect rect(0, 0, 35, 35);

rect.OffsetRect(230, 230);

// rect is now (230, 230, 265, 265)
ASSERT(rect == CRect(230, 230, 265, 265));
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::operator +
CRect::operator +=
CRect::operator -
CRect::operator -=

Other Resources

CRect Members

CRect::PtInRect

Determines whether the specified point lies within **CRect**.

```
BOOL PtInRect(
   POINT point
) const throw( );
```

Parameters

point

Contains a POINT structure or CPoint object.

Return Value

Nonzero if the point lies within **CRect**; otherwise 0.

Remarks

A point is within **CRect** if it lies on the left or top side or is within all four sides. A point on the right or bottom side is outside **CRect**.

Note

The rectangle must be normalized or this function may fail. You can call NormalizeRect to normalize the rectangle before call ing this function.

Example

```
CRect rect(5, 5, 100, 100);
CPoint pt1(35, 50);
CPoint pt2(125, 298);

// this is true, because pt1 is inside the rectangle
ASSERT(rect.PtInRect(pt1));

// this is NOT true, because pt2 is outside the rectangle
ASSERT(!rect.PtInRect(pt2));

// note that the right and the bottom aren't inside
ASSERT(!rect.PtInRect(CPoint(35, 100)));
ASSERT(!rect.PtInRect(CPoint(100, 98)));

// but the top and the
ASSERT(rect.PtInRect(CPoint(5, 65)));
ASSERT(rect.PtInRect(CPoint(88, 5)));

// and that PtInRect() works against a POINT, too
POINT pt;
pt.x = 35;
pt.y = 50;
ASSERT(rect.PtInRect(pt));
```

See Also

Reference

CRect Class

Hierarchy Chart

CRect::NormalizeRect

PtInRect

Other Resources

CRect Members

CRect::SetRect

Sets the dimensions of **CRect** to the specified coordinates.

```
void SetRect(
   int x1,
   int y1,
   int x2,
   int y2
) throw( );
```

Parameters

x1

Specifies the x-coordinate of the upper-left corner.

y1

Specifies the y-coordinate of the upper-left corner.

x2

Specifies the x-coordinate of the lower-right corner.

y2

Specifies the y-coordinate of the lower-right corner.

Example

```
CRect rect;
rect.SetRect(256, 256, 512, 512);

ASSERT(rect == CRect(256, 256, 512, 512));
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::CRect
CRect::operator =
CRect::CopyRect
CRect::SetRectEmpty
SetRect

Other Resources

CRect Members

CRect::SetRectEmpty

Makes **CRect** a null rectangle by setting all coordinates to zero.

```
void SetRectEmpty( ) throw( );
```

Example

```
CRect rect;
rect.SetRectEmpty();

// rect is now (0, 0, 0, 0)
ASSERT(rect.IsRectEmpty());
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::CRect
CRect::SetRect
CRect::CopyRect
CRect::operator =
CRect::IsRectEmpty
CRect::IsRectNull
SetRectEmpty

Other Resources

CRect Members

CRect::Size

The *cx* and *cy* members of the return value contain the height and width of **CRect**.

```
CSize Size( ) const throw( );
```

Return Value

A CSize object that contains the size of **CRect**.

Remarks

Either the height or width can be negative.

Note

The rectangle must be normalized or this function may fail. You can call NormalizeRect to normalize the rectangle before call ing this function.

Example

```
CRect rect(10, 10, 50, 50);
CSize sz = rect.Size();
ASSERT(sz.cx == 40 && sz.cy == 40);
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::Height
CRect::Width
CRect::IsRectEmpty
CRect::IsRectNull
CRect::NormalizeRect

Other Resources

CRect Members

CRect::SubtractRect

Makes the dimensions of the **CRect** equal to the subtraction of *lpRectSrc2* from *lpRectSrc1*.

```
BOOL SubtractRect(
   LPCRECT lpRectSrc1,
   LPCRECT lpRectSrc2
) throw( );
```

Parameters

lpRectSrc1

Points to the RECT structure or **CRect** object from which a rectangle is to be subtracted.

lpRectSrc2

Points to the **RECT** structure or **CRect** object that is to be subtracted from the rectangle pointed to by the *lpRectSrc1* parameter.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The subtraction is the smallest rectangle that contains all of the points in *lpRectScr1* that are not in the intersection of *lpRectScr1* and *lpRectScr2*.

The rectangle specified by *lpRectSrc1* will be unchanged if the rectangle specified by *lpRectSrc2* doesn't completely overlap the rectangle specified by *lpRectSrc1* in at least one of the x- or y-directions.

For example, if *lpRectSrc1* were (10,10, 100,100) and *lpRectSrc2* were (50,50, 150,150), the rectangle pointed to by *lpRectSrc1* would be unchanged when the function returned. If *lpRectSrc1* were (10,10, 100,100) and *lpRectSrc2* were (50,10, 150,150), however, the rectangle pointed to by *lpRectSrc1* would contain the coordinates (10,10, 50,100) when the function returned.

SubtractRect is not the same as operator - nor operator -=. Neither of these operators ever calls **SubtractRect**.

Note

Both of the rectangles must be normalized or this function may fail. You can call NormalizeRect to normalize the rectangles before calling this function.

Example

```
RECT   rectOne;
RECT   rectTwo;

rectOne.left = 10;
rectOne.top = 10;
rectOne.bottom = 100;
rectOne.right = 100;

rectTwo.left = 50;
rectTwo.top = 10;
rectTwo.bottom = 150;
rectTwo.right = 150;

CRect   rectDiff;

rectDiff.SubtractRect(&rectOne, &rectTwo);

CRect   rectResult(10, 10, 50, 100);

ASSERT(rectDiff == rectResult);

// works for CRect, too, since there is
```

```
// implicit CRect -> LPCRECT conversion

CRect rect1(10, 10, 100, 100);
CRect rect2(50, 10, 150, 150);
CRect rectOut;

rectOut.SubtractRect(rect1, rect2);
ASSERT(rectResult == rectOut);
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::operator -
CRect::operator -=
CRect::IntersectRect
CRect::UnionRect
CRect::NormalizeRect
SubtractRect

Other Resources

CRect Members

CRect::TopLeft

The coordinates are returned as a reference to a CPoint object that is contained in **CRect**.

```
CPoint& TopLeft( ) throw( );
const CPoint& TopLeft( ) const throw( );
```

Return Value

The coordinates of the top-left corner of the rectangle.

Remarks

You can use this function to either get or set the top-left corner of the rectangle. Set the corner by using this function on the left side of the assignment operator.

Example

See the example for CRect::CenterPoint.

See Also

Reference

CRect Class
Hierarchy Chart
CRect::BottomRight
CPoint Class
CRect::CenterPoint

Other Resources

CRect Members

CRect::UnionRect

Makes the dimensions of **CRect** equal to the union of the two source rectangles.

```
BOOL UnionRect(
   LPCRECT lpRect1,
   LPCRECT lpRect2
) throw( );
```

Parameters

lpRect1

Points to a RECT or **CRect** that contains a source rectangle.

lpRect2

Points to a **RECT** or **CRect** that contains a source rectangle.

Return Value

Nonzero if the union is not empty; 0 if the union is empty.

Remarks

The union is the smallest rectangle that contains both source rectangles.

Windows ignores the dimensions of an empty rectangle; that is, a rectangle that has no height or has no width.

Note

Both of the rectangles must be normalized or this function may fail. You can call NormalizeRect to normalize the rectangles before calling this function.

Example

```
CRect   rect1(100,   0, 200, 300);
CRect   rect2(  0, 100, 300, 200);
CRect   rect3;

rect3.UnionRect(&rect1, &rect2);

CRect   rectResult(0, 0, 300, 300);
ASSERT(rectResult == rect3);
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::operator |=
CRect::operator |
CRect::IntersectRect
CRect::SubtractRect
CRect::NormalizeRect
UnionRect

Other Resources

CRect Members

CRect::Width

Calculates the width of **CRect** by subtracting the left value from the right value.

```
int Width( ) const throw( );
```

Return Value

The width of **CRect**.

Remarks

The width can be negative.

Note

The rectangle must be normalized or this function may fail. You can call NormalizeRect to normalize the rectangle before call ing this function.

Example

```
CRect rect(20, 30, 80, 70);

int nWid = rect.Width();

// nWid is now 60
ASSERT(nWid == 60);
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::Height
CRect::Size
CRect::CenterPoint
CRect::IsRectEmpty
CRect::IsRectNull
CRect::NormalizeRect

Other Resources

CRect Members

CRect Operators

For information about the operators in **CRect**, see CRect Members.

CRect::operator LPCRECT

Converts a **CRect** to an LPCRECT.

```
operator LPCRECT( ) const throw( );
```

Remarks

When you use this function, you don't need the address-of (**&**) operator. This operator will be automatically used when you pass a **CRect** object to a function that expects an **LPCRECT**.

Example

```
void CYourView::OnInitialUpdate()
{
   // CWnd::GetWindowRect() takes a LPRECT, but we can
   // simply pass our CRect object because of the LPRECT
   // cast operator in the CRect class.

   CRect rect;
   GetWindowRect(rect);

   // Similarly, CWnd::MoveWindow() takes a LPCRECT but
   // the LPCRECT cast operator is used implicitly:

   MoveWindow(rect, FALSE);

   // ... more code here ...
}
```

See Also

Reference

CRect Class

Hierarchy Chart

CRect::operator LPRECT

Other Resources

CRect Members

CRect::operator LPRECT

Converts a **CRect** to an LPRECT.

```
operator LPRECT( ) throw( );
```

Remarks

When you use this function, you don't need the address-of (**&**) operator. This operator will be automatically used when you pass a **CRect** object to a function that expects an **LPRECT**.

Example

See the example for CRect::operator LPCRECT.

See Also

Reference

CRect Class
Hierarchy Chart
CRect::operator LPCRECT

Other Resources

CRect Members

CRect::operator =

Assigns *srcRect* to **CRect**.

```
void operator =(
   const RECT& srcRect
) throw( );
```

Parameters

srcRect

Refers to a source rectangle. Can be a RECT or **CRect**.

Example

```
CRect rect(0, 0, 127, 168);
CRect rect2;

rect2 = rect;

ASSERT(rect2 == CRect(0, 0, 127, 168));
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::CRect
CRect::SetRect
CRect::CopyRect
CRect::SetRectEmpty
CopyRect

Other Resources

CRect Members

CRect::operator ==

Determines whether *rect* is equal to **CRect** by comparing the coordinates of their upper-left and lower-right corners.

```
BOOL operator ==(
   const RECT& rect
) const throw( );
```

Parameters

rect

Refers to a source rectangle. Can be a RECT or **CRect**.

Return Value

Nonzero if equal; otherwise 0.

Remarks

Note

Both of the rectangles must be normalized or this function may fail. You can call NormalizeRect to normalize the rectangles before calling this function.

Example

```
CRect rect1(35, 150, 10, 25);
CRect rect2(35, 150, 10, 25);
CRect rect3(98, 999,  6,  3);

ASSERT(rect1 == rect2);

// works just fine against RECTs, as well

RECT test;
test.left = 35;
test.top = 150;
test.right = 10;
test.bottom = 25;

ASSERT(rect1 == test);
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::operator !=
CRect::NormalizeRect
EqualRect

Other Resources

CRect Members

CRect::operator !=

Determines whether *rect* is not equal to **CRect** by comparing the coordinates of their upper-left and lower-right corners.

```
BOOL operator!=(
   const RECT& rect
) const throw( );
```

Parameters

rect

Refers to a source rectangle. Can be a RECT or **CRect**.

Return Value

Nonzero if not equal; otherwise 0.

Remarks

Note

Both of the rectangles must be normalized or this function may fail. You can call NormalizeRect to normalize the rectangles before calling this function.

Example

```
CRect rect1(35, 150, 10, 25);
CRect rect2(35, 150, 10, 25);
CRect rect3(98, 999,  6,  3);

ASSERT(rect1 != rect3);

// works just fine against RECTs, as well

RECT test;
test.left = 35;
test.top = 150;
test.right = 10;
test.bottom = 25;

ASSERT(rect3 != test);
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::operator ==
CRect::NormalizeRect
EqualRect

Other Resources

CRect Members

CRect::operator +=

The first two overloads move **CRect** by the specified offsets.

```
void operator +=(
   POINT point
) throw( );
void operator +=(
   SIZE size
) throw( );
void operator +=(
   LPCRECT lpRect
) throw( );
```

Parameters

point

A POINT structure or CPoint object that specifies the number of units to move the rectangle.

size

A SIZE structure or CSize object that specifies the number of units to move the rectangle.

lpRect

Points to a RECT structure or **CRect** object that contains the number of units to inflate each side of **CRect**.

Remarks

The parameter's *x* and *y* (or *cx* and *cy*) values are added to **CRect**.

The third overload inflates **CRect** by the number of units specifed in each member of the parameter.

Example

```
CRect   rect1(100, 235, 200, 335);
CPoint   pt(35, 65);
CRect   rect2(135, 300, 235, 400);

rect1 += pt;

ASSERT(rect1 == rect2);
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::OffsetRect
CRect::InflateRect
CRect::operator +
CRect::operator -=

Other Resources

CRect Members

CRect::operator -=

The first two overloads move **CRect** by the specified offsets.

```
void operator -=(
   POINT point
) throw( );
void operator -=(
   SIZE size
) throw( );
void operator -=(
   LPCRECT lpRect
) throw( );
```

Parameters

point

A POINT structure or CPoint object that specifies the number of units to move the rectangle.

size

A SIZE structure or CSize object that specifies the number of units to move the rectangle.

lpRect

Points to a RECT structure or **CRect** object that contains the number of units to deflate each side of **CRect**.

Remarks

The parameter's *x* and *y* (or *cx* and *cy*) values are subtracted from **CRect**.

The third overload deflates **CRect** by the number of units specifed in each member of the parameter. Note that this overload functions like DeflateRect.

Example

```
CRect   rect1(100, 235, 200, 335);
CPoint   pt(35, 65);

rect1 -= pt;

CRect   rectResult(65, 170, 165, 270);
ASSERT(rect1 == rectResult);
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::OffsetRect
CRect::DeflateRect
CRect::SubtractRect
CRect::operator -
CRect::operator +=

Other Resources

CRect Members

CRect::operator |=

Sets **CRect** equal to the union of **CRect** and *rect*.

```
void operator|=(
   const RECT& rect
) throw( );
```

Parameters

rect

Contains a **CRect** or RECT.

Remarks

The union is the smallest rectangle that contains both source rectangles.

Note
Both of the rectangles must be normalized or this function may fail. You can call NormalizeRect to normalize the rectangles before calling this function.

Example

```
CRect   rect1(100,   0, 200, 300);
CRect   rect2(  0, 100, 300, 200);

rect1 |= rect2;

CRect   rectResult(0, 0, 300, 300);
ASSERT(rectResult == rect1);
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::operator |
CRect::operator &=
CRect::UnionRect
CRect::NormalizeRect
UnionRect

Other Resources

CRect Members

CRect::operator +

The first two overloads return a **CRect** object that is equal to **CRect** displaced by the specified offsets.

```
CRect operator +(
   POINT point
) const throw( );
CRect operator +(
   LPCRECT lpRect
) const throw( );
CRect operator +(
   SIZE size
) const throw( );
```

Parameters

point

A POINT structure or CPoint object that specifies the number of units to move the return value.

size

A SIZE structure or CSize object that specifies the number of units to move the return value.

lpRect

Points to a RECT structure or **CRect** object that contains the number of units to inflate each side of the return value.

Return Value

The **CRect** resulting from moving or inflating **CRect** by the number of units specified in the parameter.

Remarks

The parameter's *x* and *y* (or *cx* and *cy*) parameters are added to **CRect**'s position.

The third overload returns a new **CRect** that is equal to **CRect** inflated by the number of units specifed in each member of the parameter.

Example

```
CRect   rect1(100, 235, 200, 335);
CPoint   pt(35, 65);
CRect   rect2;

rect2 = rect1 + pt;

CRect   rectResult(135, 300, 235, 400);

ASSERT(rectResult == rect2);
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::operator +=
CRect::operator -
CRect::OffsetRect
CRect::InflateRect

Other Resources

CRect Members

CRect::operator -

The first two overloads return a **CRect** object that is equal to **CRect** displaced by the specified offsets.

```
CRect operator -(
   POINT point
) const throw( );
CRect operator -(
   SIZE size
) const throw( );
CRect operator -(
   LPCRECT lpRect
) const throw( );
```

Parameters

point

A POINT structure or **CPoint** object that specifies the number of units to move the return value.

size

A SIZE structure or **CSize** object that specifies the number of units to move the return value.

lpRect

Points to a RECT structure or **CRect** object that contains the number of units to deflate each side of the return value.

Return Value

The **CRect** resulting from moving or deflating **CRect** by the number of units specified in the parameter.

Remarks

The parameter's *x* and *y* (or *cx* and *cy*) parameters are subtracted from **CRect**'s position.

The third overload returns a new **CRect** that is equal to **CRect** deflated by the number of units specifed in each member of the parameter. Note that this overload functions like DeflateRect, not SubtractRect.

Example

```
CRect   rect1(100, 235, 200, 335);
CPoint   pt(35, 65);
CRect   rect2;

rect2 = rect1 - pt;

CRect   rectResult(65, 170, 165, 270);

ASSERT(rect2 == rectResult);
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::operator -=
CRect::operator +
CRect::OffsetRect
CRect::DeflateRect
CRect::SubtractRect

Other Resources

CRect Members

CRect::operator &

Returns a **CRect** that is the intersection of **CRect** and *rect2*.

```
CRect operator&(
   const RECT& rect2
) const throw( );
```

Parameters

rect2

Contains a RECT or **CRect**.

Return Value

A **CRect** that is the intersection of **CRect** and *rect2*.

Remarks

The intersection is the largest rectangle that is contained in both rectangles.

Note

Both of the rectangles must be normalized or this function may fail. You can call NormalizeRect to normalize the rectangles before calling this function.

Example

```
CRect   rect1(100,   0, 200, 300);
CRect   rect2(  0, 100, 300, 200);
CRect   rect3;

rect3 = rect1 & rect2;

CRect   rectResult(100, 100, 200, 200);
ASSERT(rectResult == rect3);
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::IntersectRect
CRect::operator &=
CRect::operator |
CRect::NormalizeRect

Other Resources

CRect Members

CRect::operator |

Returns a **CRect** that is the union of **CRect** and *rect2*.

```
CRect operator|(
   const RECT& rect2
) const throw( );
```

Parameters

rect2

Contains a RECT or **CRect**.

Return Value

A **CRect** that is the union of **CRect** and *rect2*.

Remarks

The union is the smallest rectangle that contains both rectangles.

Note

Both of the rectangles must be normalized or this function may fail. You can call NormalizeRect to normalize the rectangles before calling this function.

Example

```
CRect   rect1(100,   0, 200, 300);
CRect   rect2(  0, 100, 300, 200);
CRect   rect3;

rect3 = rect1 | rect2;

CRect   rectResult(0, 0, 300, 300);
ASSERT(rectResult == rect3);
```

See Also

Reference

CRect Class
Hierarchy Chart
CRect::UnionRect
CRect::operator |=
CRect::operator &
CRect::NormalizeRect

Other Resources

CRect Members

CSimpleStringT Class

This class represents a **CSimpleStringT** object.

```
template <typename BaseType>
class CSimpleStringT
```

Parameters

BaseType

The character type of the string class. Can be one of the following:

- **char** (for ANSI character strings).
- **wchar_t** (for Unicode character strings).
- **TCHAR** (for both ANSI and Unicode character strings).

Remarks

CSimpleStringT is the base class for the various string classes supported by Visual C++. It provides minimal support for memory management of the string object and basic buffer manipulation. For more advanced string objects, see CStringT Class.

Requirements

Header: atlsimpstr.h

Example

See the ISAPIFilter Sample.

See Also

Reference

Hierarchy Chart

Other Resources

CSimpleStringT Members

ATL/MFC Shared Classes

CSimpleStringT Members

Methods

Append	Appends a **CSimpleStringT** object to an existing **CSimpleStringT** object.
AppendChar	Appends a character to an existing **CSimpleStringT** object.
CopyChars	Copies a character or characters to another string.
CopyCharsOverlapped	Copies a character or characters to another string in which the buffers overlap.
CSimpleStringT	Constructs **CSimpleStringT** objects in various ways.
Empty	Forces a string to have a length of zero.
FreeExtra	Frees any extra memory previously allocated by the string object.
GetAllocLength	Retrieves the allocated length of a **CSimpleStringT** object.
GetAt	Returns the character at a given position.
GetBuffer	Returns a pointer to the characters in a **CSimpleStringT**.
GetBufferSetLength	Returns a pointer to the characters in a **CSimpleStringT**, truncating to the specified length.
GetLength	Returns the number of characters in a **CSimpleStringT** object.
GetManager	Retrieves the memory manager of the **CSimpleStringT** object.
GetString	Retrieves the character string
IsEmpty	Tests whether a **CSimpleStringT** object contains no characters.
LockBuffer	Disables reference counting and protects the string in the buffer.
Preallocate	Allocates a specific amount of memory for the character buffer.
ReleaseBuffer	Releases control of the buffer returned by **GetBuffer**.
ReleaseBufferSetLength	Releases control of the buffer returned by **GetBuffer**.
SetAt	Sets a character at a given position.
SetManager	Sets the memory manager of a **CSimpleStringT** object.
SetString	Sets the string of a **CSimpleStringT** object.
StringLength	Returns the number of characters in the specified string.
Truncate	Truncates the string to a specified length.

UnlockBuffer	Enables reference counting and releases the string in the buffer.

Operators

operator +=	Concatenates a new string to the end of an existing string.
operator =	Assigns a new value to a **CSimpleStringT** object.
operator []	Returns the character at a given position — operator substitution for **GetAt**.
operator PCXSTR	Directly accesses characters stored in a **CSimpleStringT** object as a C-style string.

Typedefs

PCXSTR	A pointer to a constant string.
PXSTR	A pointer to a string.

See Also

Reference

CSimpleStringT Class
Hierarchy Chart

CSimpleStringT Methods

For information about the methods in **CSimpleStringT**, see CSimpleStringT Members.

CSimpleStringT::Append

Appends a **CSimpleStringT** object to an existing **CSimpleStringT** object.

```
void Append(
   const CSimpleStringT& strSrc
);
void Append(
   PCXSTR pszSrc,
   int nLength
);
void Append(
      PCXSTR pszSrc
);
```

Parameters

strSrc

The **CSimpleStringT** object to be appended.

pszSrc

A pointer to a string containing the characters to be appended.

nLength

The number of characters to append.

Remarks

Call this method to append an existing **CSimpleStringT** object to another **CSimpleStringT** object.

Example

The following example demonstrates the use of **CSimpleStringT::Append**.

```
CSimpleString str1(pMgr), str2(pMgr);

str1.SetString("Soccer is");
str2.SetString(" an elegant game");

str1.Append(str2);
_ASSERT(strcmp(str1, "Soccer is an elegant game") == 0);
```

For another example, see the ISAPIFilter Sample.

See Also

Reference

CSimpleStringT Class

CSimpleStringT::AppendChar

Other Resources

CSimpleStringT Members

CSimpleStringT::AppendChar

Appends a character to an existing **CSimpleStringT** object.

```
void AppendChar(
   XCHAR ch
);
```

Parameters

ch

The character to be appended

Remarks

Call this function to append the specified character to the end of an existing **CSimpleStringT** object.

See Also

Reference

CSimpleStringT Class

CSimpleStringT::Append

Other Resources

CSimpleStringT Members

CSimpleStringT::CopyChars

Copies a character or characters to a **CSimpleStringT** object.

```
static void CopyChars(
      XCHAR* pchDest,
   const XCHAR* pchSrc,
   int nChars
) throw( );
```

Parameters

pchDest

A pointer to a character string.

pchSrc

A pointer to a string containing the characters to be copied.

nChars

The number of *pchSrc* characters to be copied.

Remarks

Call this method to copy characters from *pchSrc* to the *pchDest* string.

Example

The following example demonstrates the use of **CSimpleStringT::CopyChars**.

```
{
    // ...

    CSimpleString str( "xxxxxxxxxxxxxxxxxxx", 20, pMgr );

    char* pszSrc= NULL;
    pszSrc= new char(12);
    if(pszSrc)
        pszSrc= "Hello world!";
    printf_s("%s\n", str);

    str.CopyChars(str.GetBuffer(), pszSrc, 12);
    printf_s("%s\n", str);

    // ...
}
```

Output

```
xxxxxxxxxxxxxxxxxxx
Hello world!xxxxxxx
```

See Also

Reference

CSimpleStringT Class

Other Resources

CSimpleStringT Members

CSimpleStringT::CopyCharsOverlapped

Copies a character or characters to a **CSimpleStringT** object.

```
static void CopyCharsOverlapped(
   XCHAR* pchDest,
   const XCHAR* pchSrc,
   int nChars
) throw( );
```

Parameters

pchDest

A pointer to a character string.

pchSrc

A pointer to a string containing the characters to be copied.

nChars

The number of *pchSrc* characters to be copied.

Remarks

Call this method to copy characters from *pchSrc* to the *pchDest* string. Unlike **CopyChars**, **CopyCharsOverlapped** provides a safe method for copying from character buffers that might be overlapped.

Example

See the example for CSimpleStringT::CopyChars, or the source code for **CSimpleStringT::SetString** (located in atlsimpstr.h).

See Also

Reference

CSimpleStringT Class

Other Resources

CSimpleStringT Members

CSimpleStringT::CSimpleStringT

Constructs a **CSimpleStringT** object.

```
CSimpleStringT(
   const XCHAR* pchSrc,
   int nLength,
   IAtlStringMgr* pStringMgr
);
CSimpleStringT(
   PCXSTR pszSrc,
   IAtlStringMgr* pStringMgr
);
CSimpleStringT(
   const CSimpleStringT& strSrc
);
explicit CSimpleStringT(
   IAtlStringMgr* pStringMgr
) throw( );
```

Parameters

strSrc

An existing **CSimpleStringT** object to be copied into this **CSimpleStringT** object.

pchSrc

A pointer to an array of characters of length *nLength*, not null terminated.

pszSrc

A null-terminated string to be copied into this **CSimpleStringT** object.

nLength

A count of the number of characters in *pch*.

pStringMgr

A pointer to the memory manager of the **CSimpleStringT** object. For more information on **IAtlStringMgr** and memory management for **CSimpleStringT**, see Memory Management and CStringT.

Remarks

Call this method to construct a new **CSimpleStringT** object with the specified data. Because the constructors copy the input data into new allocated storage, you should be aware that memory exceptions may result.

Example

The following example demonstrates the use of **CSimpleStringT::CSimpleStringT**.

```
CSimpleString s1(pMgr);              // Empty string
CSimpleString s2("cat", pMgr);       // From a C string literal

CSimpleString s3 = s2;               // Copy constructor
CSimpleString s4( s2 + " " + s3 );   // From a string expression

CSimpleString s5("xxxxxx", 6, pMgr); // s5 = "xxxxxx"
```

See Also

Reference

CSimpleStringT Class

Other Resources

CSimpleStringT Members

CSimpleStringT::~CSimpleStringT

Destroys a **CSimpleStringT** object.

```
~CSimpleStringT( ) throw( );
```

Remarks

Call this method to destroy the **CSimpleStringT** object.

See Also

Reference

CSimpleStringT Class

CSimpleStringT::CSimpleStringT

Other Resources

CSimpleStringT Members

CSimpleStringT::Empty

Makes this **CSimpleStringT** object an empty string and frees memory as appropriate.

```
void Empty( ) throw( );
```

Remarks

For more information, see Strings: CString Exception Cleanup.

Example

The following example demonstrates the use of **CSimpleStringT::Empty**.

```
CSimpleString s( "abc", pMgr);
s.Empty( );
_ASSERT( s.GetLength( ) == 0 );
```

See Also

Reference

CSimpleStringT Class

CSimpleStringT::IsEmpty

Other Resources

CSimpleStringT Members

CSimpleStringT::FreeExtra

Frees any extra memory previously allocated by the string but no longer needed.

```
void FreeExtra( );
```

Remarks

This should reduce the memory overhead consumed by the string object. The method reallocates the buffer to the exact length returned by GetLength.

Example

The following example prints the three lines shown under **Output**.

```
// CSimpleStringT_FreeExtra.cpp
// Compile with: /MT
#include <string.h>
#include <atlsimpstr.h>
#include <atlstr.h>

void PrintLengths( CSimpleString& str )
{
    printf_s( "Alloc length is %d, String length is %d\n",
              str.GetAllocLength( ), str.GetLength( ) );
}

int main( )
{
    CAtlString basestr;
    IAtlStringMgr* pMgr;

    pMgr= basestr.GetManager();
    _ASSERT(pMgr != NULL);

    // Create a CSimpleString with 1024 characters
    CSimpleString str( "x", 1024, pMgr );
    PrintLengths( str );

    // Assigning a smaller string won't cause CSimpleString to free its
    // memory, because it assumes the string will grow again anyway.
    str = _T( "Soccer is Best!" );
    PrintLengths( str );

    // This call forces CSimpleString to release the extra
    // memory it doesn't need.
    str.FreeExtra( );
    PrintLengths( str );
}
```

Output

```
Alloc length is 1031, String length is 1024
Alloc length is 1031, String length is 15
Alloc length is 15, String length is 15
```

See Also

Reference

CSimpleStringT Class

Other Resources

CSimpleStringT Members

CSimpleStringT::GetAllocLength

Retrieves the allocated length of a **CSimpleStringT** object.

```
int GetAllocLength( ) const throw( );
```

Return Value

The number of characters allocated for this object.

Remarks

Call this method to determine the number of characters allocated for this **CSimpleStringT** object. See FreeExtra for an example of calling this function.

See Also

Reference

CSimpleStringT Class

Other Resources

CSimpleStringT Members

CSimpleStringT::GetAt

Returns a single character from a **CSimpleStringT** object.

```
XCHAR GetAt(
   int iChar
) const;
```

Parameters

iChar

Zero-based index of the character in the **CSimpleStringT** object. The *iChar* parameter must be greater than or equal to 0 and less than the value returned by GetLength.

Return Value

An **XCHAR** containing the character at the specified position in the string.

Remarks

Call this method to return a single character specified by *iChar*. The overloaded subscript (**[]**) operator is a convenient alias for **GetAt**.

Example

The following example demonstrates the use of **CSimpleStringT::GetAt**.

```
CSimpleString s( "abcdef", pMgr );
_ASSERT( s.GetAt( 2 ) == 'c' );
```

See Also

Reference

CSimpleStringT Class
CSimpleStringT::GetLength
CSimpleStringT::operator []

Other Resources

CSimpleStringT Members

CSimpleStringT::GetBuffer

Returns a pointer to the internal character buffer for the **CSimpleStringT** object.

```
PXSTR GetBuffer(
   int nMinBufferLength
);
PXSTR GetBuffer( );
```

Parameters

nMinBufferLength

The minimum size of the character buffer in characters. This value does not include space for a null terminator.

Return Value

An **PXSTR** pointer to the object's (null-terminated) character buffer.

Remarks

Call this method to return the buffer contents of the **CSimpleStringT** object. The returned **PXSTR** is not **const** and thus allows direct modification of **CSimpleStringT** contents.

Note

If *nMinBufferLength* is greater than the length of the current buffer, the call to **GetBuffer** destroys the current buffer, replacing it with a buffer of the requested size and resetting the reference count to zero. If you have previously called LockBuffer on this buffer, you will lose the buffer's lock.

If you use the pointer returned by **GetBuffer** to change the string contents, you must call ReleaseBuffer before using any other **CSimpleStringT** member methods.

The address returned by **GetBuffer** may not be valid after the call to **ReleaseBuffer** because additional **CSimpleStringT** operations can cause the **CSimpleStringT** buffer to be reallocated. The buffer is not reallocated if you do not change the length of the **CSimpleStringT**.

The buffer memory is automatically freed when the **CSimpleStringT** object is destroyed.

If you keep track of the string length yourself, you should not append the terminating null character. You must, however, specify the final string length when you release the buffer with **ReleaseBuffer**. If you do append a terminating null character, you should pass –1 (the default) for the length to **ReleaseBuffer**, and **ReleaseBuffer** will perform a **strlen** on the buffer to determine its length.

If there is insufficient memory to satisfy the **GetBuffer** request, the method will throw an exception.

Example

```
// CSimpleStringT_GetBuffer.cpp
#include <atlmem.h>
#include <atlstr.h>

// Declare a thread-safe, growable, private heap with initial size 0:
CWin32Heap g_stringHeap( 0, 0, 0 );

// Declare a string manager that uses the private heap:
CAtlStringMgr g_stringMgr( &g_stringHeap );

int main() {
   CSimpleString s( "abcd", &g_stringMgr );
   LPTSTR pBuffer = s.GetBuffer( 10 );

   printf("%s\n", pBuffer);
   // Directly access CSimpleString buffer
   strcpy_s( pBuffer, 10, "Hello" );
   printf("%s\n", pBuffer);
```

```
    s.ReleaseBuffer( );
    _ASSERT(strcmp(s, "Hello") == 0);
}
```

See Also

Reference

CSimpleStringT Class

CSimpleStringT::GetBufferSetLength

CSimpleStringT::ReleaseBuffer

Other Resources

CSimpleStringT Members

CSimpleStringT::GetBufferSetLength

Returns a pointer to the internal character buffer for the **CSimpleStringT** object, truncating or growing its length if necessary to exactly match the length specified in *nLength*.

```
PXSTR GetBufferSetLength(
   int nLength
);
```

Parameters

nLength

The exact size of the **CSimpleStringT** character buffer in characters.

Return Value

A **PXSTR** pointer to the object's (null-terminated) character buffer.

Remarks

Call this method to retrieve a specified length of the internal buffer of the **CSimpleStringT** object. The returned **PXSTR** pointer is not **const** and thus allows direct modification of **CSimpleStringT** contents.

If you use the pointer returned by GetBufferSetLength to change the string contents, you may need to call **ReleaseBuffer** before using any other **CSimpleStringT** member methods.

The address returned by **GetBufferSetLength** may not be valid after the call to **ReleaseBuffer** because additional **CSimpleStringT** operations can cause the **CSimpleStringT** buffer to be reallocated. The buffer is not reassigned if you do not change the length of the **CSimpleStringT**.

The buffer memory is automatically freed when the **CSimpleStringT** object is destroyed.

If you keep track of the string length yourself, you should not append the terminating null character. You must, however, specify the final string length when you release the buffer with **ReleaseBuffer**. If you do append a terminating null character when you call **ReleaseBuffer**, you should pass –1 (the default) for the length to **ReleaseBuffer**, and **ReleaseBuffer** will perform a **strlen** on the buffer to determine its length.

For more information about reference counting, see the following articles:

- Managing Object Lifetimes through Reference Counting in the *Platform SDK*.
- Implementing Reference Counting in the *Platform SDK*.
- Rules for Managing Reference Counts in the *Platform SDK*.

Example

The following example demonstrates the use of **CSimpleStringT::GetBufferSetLength**.

```
// CSimpleStringT_GetBufferSetLength.cpp
#include <atlmem.h>
#include <atlstr.h>

// Declare a thread-safe, growable, private heap with initial size 0:
CWin32Heap g_stringHeap( 0, 0, 0 );

// Declare a string manager that uses the private heap:
CAtlStringMgr g_stringMgr( &g_stringHeap );

int main() {
   CSimpleString s( "abcd", &g_stringMgr );
   LPTSTR pBuffer = s.GetBuffer( 10 );

   LPTSTR pstr = s.GetBufferSetLength( 3 );
   pstr[0] = 'C';
   pstr[1] = 'u';
```

```
    pstr[2] = 'p';

    // No need for trailing zero or call to ReleaseBuffer( )
    // because GetBufferSetLength( ) set it for us.

    s += _T( " soccer is best!" );
    _ASSERT(strcmp(s, "Cup soccer is best!") == 0);
}
```

See Also

Reference

CSimpleStringT Class

CSimpleStringT::GetBuffer

CSimpleStringT::ReleaseBuffer

Other Resources

CSimpleStringT Members

CSimpleStringT::GetLength

Returns the number of characters in the **CSimpleStringT** object.

```
int GetLength( ) const throw( );
```

Return Value

A count of the characters in the string.

Remarks

Call this method to return the number of characters in the object. The count does not include a null terminator.

For multibyte character sets (MBCS), **GetLength** counts each 8-bit character; that is, a lead and trail byte in one multibyte character are counted as two bytes. See FreeExtra for an example of calling this function.

See Also

Reference

CSimpleStringT Class

CSimpleStringT::IsEmpty

Other Resources

CSimpleStringT Members

CSimpleStringT::GetManager

Retrieves the memory manager of the **CSimpleStringT** object.

```
IAtlStringMgr* GetManager( ) const throw( );
```

Return Value

A pointer to the memory manager for the **CSimpleStringT** object.

Remarks

Call this method to retrieve the memory manager used by the **CSimpleStringT** object. For more information on memory managers and string objects, see Memory Management and CStringT.

See Also

Reference

CSimpleStringT Class

CSimpleStringT::SetManager

Other Resources

CSimpleStringT Members

CSimpleStringT::GetString

Retrieves the character string.

```
PCXSTR GetString( ) const throw( );
```

Return Value

A pointer to a null-terminated character string.

Remarks

Call this method to retrieve the character string associated with the **CSimpleStringT** object.

Note
The returned **PCXSTR** pointer is **const** and does not allow direct modification of **CSimpleStringT** contents.

Example

The following example demonstrates the use of **CSimpleStringT::GetString**.

```
{
    // ...
    CSimpleString str(pMgr);

    str += _T( "Cup soccer is best!" );
    printf_s("%s", str.GetString());
    // ...
}
```

See Also

Reference

CSimpleStringT Class

CSimpleStringT::GetBuffer

Other Resources

CSimpleStringT Members

CSimpleStringT::IsEmpty

Tests a **CSimpleStringT** object for the empty condition.

```
bool IsEmpty( ) const throw( );
```

Return Value

Returns **true** if the **CSimpleStringT** object has 0 length; otherwise **false**.

Remarks

Call this method to determine if the object contains an empty string.

Example

The following example demonstrates the use of **CSimpleStringT::IsEmpty**.

```
CSimpleString s( pMgr );
_ASSERT( s.IsEmpty( ) );
```

See Also

Reference

CSimpleStringT Class

CSimpleStringT::GetLength

Other Resources

CSimpleStringT Members

CSimpleStringT::LockBuffer

Disables reference counting and protects the string in the buffer.

```
PXSTR LockBuffer( );
```

Return Value

A pointer to a **CSimpleStringT** object or a null-terminated string.

Remarks

Call this method to lock the buffer of the **CSimpleStringT** object. By calling **LockBuffer**, you create a copy of the string, with a –1 for the reference count. When the reference count value is -1, the string in the buffer is considered to be in a "locked" state. While in a locked state, the string is protected in two ways:

- No other string can get a reference to the data in the locked string, even if that string is assigned to the locked string.
- The locked string will never reference another string, even if that other string is copied to the locked string.

By locking the string in the buffer, you ensure that the string's exclusive hold on the buffer will remain intact.

After you have finished with **LockBuffer**, call UnlockBuffer to reset the reference count to 1.

Note

If you call GetBuffer on a locked buffer and you set the **GetBuffer** parameter *nMinBufferLength* to greater than the length of the current buffer, you will lose the buffer lock. Such a call to **GetBuffer** destroys the current buffer, replaces it with a buffer of the requested size, and resets the reference count to zero.

For more information about reference counting, see the following articles:

- Managing Object Lifetimes through Reference Counting in the *Platform SDK*
- Implementing Reference Counting in the *Platform SDK*
- Rules for Managing Reference Counts in the *Platform SDK*

Example

The following example demonstrates the use of **CSimpleStringT::LockBuffer**.

```
{
    ...
    CSimpleString str( "Hello", pMgr );
    char ch;

    str.LockBuffer();
    ch= str.GetAt(2);
    printf_s("%c", ch);
    str.UnlockBuffer();
    ...
}
```

See Also

Reference

CSimpleStringT Class

CSimpleStringT::ReleaseBuffer

Other Resources

CSimpleStringT Members

CSimpleStringT::Preallocate

Allocates a specific amount of bytes for the **CSimpleStringT** object.

```
void Preallocate(
   int nLength
);
```

Parameters

nLength

The exact size of the **CSimpleStringT** character buffer in characters.

Remarks

Call this method to allocate a specific buffer size for the **CSimpleStringT** object.

Example

The following example demonstrates the use of **CSimpleStringT::Preallocate**.

```
{
    ...
    CSimpleString str( pMgr );

    printf_s("Allocated length: %d\n", str.GetAllocLength());
    str.Preallocate(100);
    printf_s("Allocated length: %d\n", str.GetAllocLength());
    ...
}
```

For another example, see the ISAPIFilter Sample.

See Also

Reference

CSimpleStringT Class

Other Resources

CSimpleStringT Members

CSimpleStringT::ReleaseBuffer

Releases control of the buffer allocated by GetBuffer.

```
void ReleaseBuffer(
   int nNewLength = -1
);
```

Parameters

nNewLength

The new length of the string in characters, not counting a null terminator. If the string is null terminated, the -1 default value sets the **CSimpleStringT** size to the current length of the string.

Remarks

Call this method to reallocate or free up the buffer of the string object. If you know that the string in the buffer is null terminated, you can omit the *nNewLength* argument. If your string is not null terminated, use *nNewLength* to specify its length. The address returned by GetBuffer is invalid after the call to **ReleaseBuffer** or any other **CSimpleStringT** operation.

Example

The following example demonstrates the use of **CSimpleStringT::ReleaseBuffer**.

```
{
    // ...
    enum {
        bufferSize = 1024
    };
    CSimpleString s( "abc", pMgr );

    LPTSTR p = s.GetBuffer(bufferSize);
    strcpy_s( p, bufferSize , "abc" );   // use the buffer directly
    _ASSERT( s.GetLength( ) == 3 ); // String length = 3
    s.ReleaseBuffer( );  // Surplus memory released, p is now invalid.
    _ASSERT( s.GetLength( ) == 3 ); // Length still 3
}
```

See Also

Reference

CSimpleStringT Class

Other Resources

CSimpleStringT Members

CSimpleStringT::ReleaseBufferSetLength

Releases control of the buffer allocated by GetBuffer.

```
void ReleaseBufferSetLength(
   int nNewLength
);
```

Parameters

nNewLength

The length of the string being released

Remarks

This function is functionally similar to ReleaseBuffer except that a valid length for the string object must be passed.

See Also

Reference

CSimpleStringT Class

Other Resources

CSimpleStringT Members

CSimpleStringT::SetAt

Sets a single character from a **CSimpleStringT** object.

```
void SetAt(
   int iChar,
   XCHAR ch
);
```

Parameters

iChar

Zero-based index of the character in the **CSimpleStringT** object. The *iChar* parameter must be greater than or equal to 0 and less than the value returned by GetLength.

ch

The new character.

Remarks

Call this method to overwrite the character located at *iChar*. This method will not enlarge the string if *iChar* exceeds the bounds of the existing string.

Example

The following example demonstrates the use of **CSimpleStringT::SetAt**.

```
CSimpleString s( "abcdef", pMgr );

s.SetAt(1, 'a');
_ASSERT(strcmp(s, "aacdef") == 0);
```

See Also

Reference

CSimpleStringT Class
CSimpleStringT::GetAt
CSimpleStringT::operator []

Other Resources

CSimpleStringT Members

CSimpleStringT::SetManager

Specifies the memory manager of the **CSimpleStringT** object.

```
void SetManager(
   IAtlStringMgr* pStringMgr
);
```

Parameters

pStringMgr

A pointer to the new memory manager.

Remarks

Call this method to specify a new memory manager used by the **CSimpleStringT** object. For more information on memory managers and string objects, see Memory Management and CStringT.

Example

The following example demonstrates the use of **CSimpleStringT::SetManager**.

```
CSimpleString s( "abcdef", pMgr );
s.SetManager( pCustomMgr );
```

See Also

Reference

CSimpleStringT Class

CSimpleStringT::GetManager

Other Resources

CSimpleStringT Members

CSimpleStringT::SetString

Sets the string of a **CSimpleStringT** object.

```
void SetString(
   PCXSTR pszSrc,
   int nLength
);
void SetString(
   PCXSTR pszSrc
);
```

Parameters

pszSrc

A pointer to a null-terminated string.

nLength

A count of the number of characters in *pszSrc*.

Remarks

Call this method to reinitialize the buffer of a **CSimpleStringT** object with a new string.

Example

The following example demonstrates the use of **CSimpleStringT::SetString**.

```
CSimpleString s( "abcdef", pMgr );
_ASSERT(strcmp(s, "abcdef") == 0);

s.SetString("Soccer", 6);
_ASSERT(strcmp(s, "Soccer") == 0);
```

See Also

Reference

CSimpleStringT Class

Other Resources

CSimpleStringT Members

CSimpleStringT::StringLength

Returns the number of characters in the specified string.

```
ATL_NOINLINE static int StringLength(
   PCXSTR psz
) throw( );
```

Parameters

psz

A pointer to a null-terminated string.

Return Value

The number of characters in *psz*; not counting a null terminator.

Remarks

Call this method to retrieve the number of characters in the string pointed to by *psz*.

Example

The following example demonstrates the use of **CSimpleStringT::StringLength**.

```
_ASSERT( CSimpleStringT::StringLength("soccer") == 6 );
```

See Also

Reference

CSimpleStringT Class

Other Resources

CSimpleStringT Members

CSimpleStringT::Truncate

Truncates the string to the new length.

```
void Truncate(
   int nNewLength
);
```

Parameters

nNewLength

The new length of the string.

Remarks

Call this method to truncate the contents of the string to the new length.

Note
This does not affect the allocated length of the buffer. To decrease or increase the current buffer, see FreeExtra and Preallocate.

Example

The following example demonstrates the use of **CSimpleStringT::Truncate**.

```
// ...
CSimpleString str( "abcdefghi", pMgr );

printf_s("Allocated length: %d\n", str.GetLength());
printf_s("Contents: %s\n", str);

str.Truncate(4);
printf_s("Allocated length: %d\n", str.GetLength());
printf_s("Contents: %s\n", str);
// ...
```

See Also

Reference

CSimpleStringT Class

Other Resources

CSimpleStringT Members

CSimpleStringT::UnlockBuffer

Unlocks the buffer of the **CSimpleStringT** object.

```
void UnlockBuffer( ) throw( );
```

Remarks

Call this method to reset the reference count of the string to 1.

The **CSimpleStringT** destructor automatically calls **UnlockBuffer** to ensure that the buffer is not locked when the destructor is called. For an example of this method, see LockBuffer.

See Also

Reference

CSimpleStringT Class
CSimpleStringT::LockBuffer
CSimpleStringT::GetBuffer
CSimpleStringT::ReleaseBuffer

Other Resources

CSimpleStringT Members

CSimpleStringT Operators

For information about the operators in **CSimpleStringT**, see CSimpleStringT Members.

CSimpleStringT::operator +=

Joins a new string or character to the end of an existing string.

```
CSimpleStringT& operator +=(
   PCXSTR pszSrc
);
CSimpleStringT& operator +=(
   const CSimpleStringT& strSrc
);
template< int t_nSize >
CSimpleStringT& operator+=(
      const CStaticString< XCHAR, t_nSize >& strSrc
);
CSimpleStringT& operator +=(
   char ch
);
CSimpleStringT& operator +=(
   unsigned char ch
);
CSimpleStringT& operator +=(
   wchar_t ch
);
```

Parameters

pszSrc

A pointer to a null-terminated string.

strSrc

A pointer to an existing **CSimpleStringT** object.

ch

The character to be appended.

Remarks

The operator accepts another **CSimpleStringT** object or a character. Note that memory exceptions may occur whenever you use this concatenation operator because new storage may be allocated for characters added to this **CSimpleStringT** object.

Example

The following example demonstrates the use of **CSimpleStringT::operator +=**.

```
CSimpleString str( "abc", pMgr );
_ASSERT(strcmp(( str += "def" ), "abcdef") == 0);
```

See Also

Reference

CSimpleStringT Class

CSimpleStringT::operator +

Other Resources

CSimpleStringT Members

CSimpleStringT::operator =

Assigns a new value to a **CSimpleStringT** object.

```
CSimpleStringT& operator =(
   PCXSTR pszSrc
);
CSimpleStringT& operator =(
   const CSimpleStringT& strSrc
);
```

Parameters

pszSrc

A pointer to a null-terminated string.

strSrc

A pointer to an existing **CSimpleStringT** object.

Remarks

If the destination string (the left side) is already large enough to store the new data, no new memory allocation is performed. Note that memory exceptions may occur whenever you use the assignment operator because new storage is often allocated to hold the resulting **CSimpleStringT** object.

Example

The following example demonstrates the use of **CSimpleStringT::operator =**.

```
CSimpleString s1( pMgr ), s2( pMgr );  // Empty CSimpleStringT objects

s1 = "cat";            // s1 = "cat"
_ASSERT(strcmp(s1, "cat") == 0);

s2 = s1;               // s1 and s2 each = "cat"
_ASSERT(strcmp(s2, "cat") == 0);

s1 = "the " + s1;      // Or expressions
_ASSERT(strcmp(s1, "the cat") == 0);

s1 = "x";              // Or just individual characters
_ASSERT(strcmp(s1, "x") == 0);
```

See Also

Reference

CSimpleStringT Class

CSimpleStringT::CSimpleStringT

Other Resources

CSimpleStringT Members

CSimpleStringT::operator []

Call this function to access a single character of the character array.

```
XCHAR operator[](
   int iChar
) const;
```

Parameters

iChar

Zero-based index of a character in the string.

Remarks

The overloaded subscript (**[]**) operator returns a single character specified by the zero-based index in *iChar*. This operator is a convenient substitute for the GetAt member function.

Note

You can use the subscript (**[]**) operator to get the value of a character in a **CSimpleStringT**, but you cannot use it to change t he value of a character in a **CSimpleStringT**.

Example

The following example demonstrates the use of **CSimpleStringT::operator []**.

```
CSimpleString s( "abc", pMgr );
_ASSERT( s[1] == 'b' );
```

See Also

Reference

CSimpleStringT Class

CSimpleStringT::SetAt

Other Resources

CSimpleStringT Members

CSimpleStringT::operator PCXSTR

Directly accesses characters stored in a **CSimpleStringT** object as a C-style string.

```
operator PCXSTR( ) const throw( );
```

Return Value

A character pointer to the string's data.

Remarks

No characters are copied; only a pointer is returned. Be careful with this operator. If you change a **CString** object after you have obtained the character pointer, you may cause a reallocation of memory that invalidates the pointer.

Example

The following example demonstrates the use of **CSimpleStringT::operator PCXSTR**.

```
// If the prototype of a function is known to the compiler,
// the PCXSTR cast operator may be invoked implicitly.

CSimpleStringT strSports( _T ( "Soccer is Best!" ) );
TCHAR sz[1024];

lstrcpy( sz, strSports );

// If the prototype isn't known or is a va_arg prototype,
// you must invoke the cast operator explicitly. For example,
// the va_arg part of a call to sprintf( ) needs the cast:

sprintf( sz, "I think that %s!\n", ( PCXSTR ) strSports );

// While the format parameter is known to be an PCXSTR and
// therefore doesn't need the cast:

sprintf( sz, strSports );

// Note that some situations are ambiguous. This line will
// put the address of the strSports object to stdout:

cout << strSports;

// while this line will put the content of the string out:

cout << ( PCXSTR ) strSports;
```

See Also

Reference

CSimpleStringT Class

Other Resources

CSimpleStringT Members

CSimpleStringT Typedefs

For information about the typedefs in **CSimpleStringT**, see CSimpleStringT Members.

CSimpleStringT::PCXSTR

A pointer to a constant string.

```
typedef ChTraitsBase< BaseType >::PCXSTR PCXSTR;
```

See Also

Reference

CSimpleStringT Class

CSimpleStringT::PXSTR

Other Resources

CSimpleStringT Members

CSimpleStringT::PXSTR

A pointer to a string.

```
typedef ChTraitsBase< BaseType >::PXSTR PXSTR;
```

See Also

Reference

CSimpleStringT Class

CSimpleStringT::PCXSTR

Other Resources

CSimpleStringT Members

CSize Class

Similar to the Windows SIZE structure, which implements a relative coordinate or position.

```
class CSize : public tagSIZE
```

Remarks

This class is derived from the **SIZE** structure. This means you can pass a **CSize** in a parameter that calls for a **SIZE** and that the data members of the **SIZE** structure are accessible data members of **CSize**.

The **cx** and **cy** members of **SIZE** (and **CSize**) are public. In addition, **CSize** implements member functions to manipulate the **SIZE** structure.

Note
For more information on shared utility classes (like **CSize**), see Shared Classes.

Requirements

Header: atltypes.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

MDI Sample: Demonstrates MDI Without Using Doc/View Architecture

Reference

Hierarchy Chart

CRect Class

CPoint Class

Other Resources

CSize Members

CSize Members

Construction

CSize	Constructs a **CSize** object.

Operators

operator !=	Checks for inequality between **CSize** and a size.
operator +=	Adds a size to **CSize**.
operator –=	Subtracts a size from **CSize**.
operator ==	Checks for equality between **CSize** and a size.

Operators Returning CSize Values

operator –	Subtracts two sizes.
operator +	Adds two sizes.

See Also

Reference

CSize Class

Hierarchy Chart

CSize Member Functions

For information about the member functions in **CSize**, see CSize Members.

CSize::CSize

Constructs a **CSize** object.

```
CSize( ) throw( );
CSize(
   int initCX,
   int initCY
) throw( );
CSize(
   SIZE initSize
) throw( );
CSize(
   POINT initPt
) throw( );
CSize(
   DWORD dwSize
) throw( );
```

Parameters

initCX

Sets the **cx** member for the **CSize**.

initCY

Sets the **cy** member for the **CSize**.

initSize

SIZE structure or **CSize** object used to initialize **CSize**.

initPt

POINT structure or **CPoint** object used to initialize **CSize**.

dwSize

DWORD used to initialize **CSize**. The low-order word is the **cx** member and the high-order word is the **cy** member.

Remarks

If no arguments are given, **cx** and **cy** members are not initialized.

Example

```
CSize szEmpty();
CSize szPointA(10, 25);

SIZE sz;
sz.cx = 10;
sz.cy = 25;
CSize szPointB(sz);

POINT pt;
pt.x = 10;
pt.y = 25;
CSize szPointC(pt);

CPoint ptObject(10, 25);
CSize szPointD(ptObject);

DWORD dw = MAKELONG(10, 25);
CSize szPointE(dw);

ASSERT(szPointA == szPointB);
ASSERT(szPointB == szPointC);
```

```
ASSERT(szPointC == szPointD);
ASSERT(szPointD == szPointE);
```

See Also

Reference

CSize Class

Hierarchy Chart

Other Resources

CSize Members

CSize Operators

For information about the operators in **CSize**, see CSize Members.

CSize::operator ==

Checks for equality between two sizes.

```
BOOL operator ==(
   SIZE size
) const throw( );
```

Remarks

Returns nonzero if the sizes are equal, otherwize 0.

Example

```
CSize sz1(135, 135);
CSize sz2(135, 135);

ASSERT(sz1 == sz2);
```

See Also

Reference

CSize Class

Hierarchy Chart

CSize::operator !=

Other Resources

CSize Members

CSize::operator !=

Checks for inequality between two sizes.

```
BOOL operator!=(
   SIZE size
) const throw( );
```

Remarks

Returns nonzero if the sizes are not equal, otherwise 0.

Example

```
CSize sz1(222, 222);
CSize sz2(111, 111);

ASSERT(sz1 != sz2);
```

See Also

Reference

CSize Class

Hierarchy Chart

CSize::operator ==

Other Resources

CSize Members

CSize::operator +=

Adds a size to this **CSize**.

```
void operator +=(
   SIZE size
) throw( );
```

Example

```
CSize sz1(100, 100);
CSize sz2( 50,  25);

sz1 += sz2;

CSize szResult(150, 125);
ASSERT(sz1 == szResult);

// works with SIZE, too

sz1 = CSize(100, 100);
SIZE sz3;
sz3.cx = 50;
sz3.cy = 25;

sz1 += sz3;
ASSERT(sz1 == szResult);
```

See Also

Reference

CSize Class

Hierarchy Chart

CSize::operator +

Other Resources

CSize Members

CSize::operator -=

Subtracts a size from this **CSize**.

```
void operator -=(
   SIZE size
) throw( );
```

Example

```
CSize sz1(100, 100);
CSize sz2( 50,  25);

sz1 -= sz2;

CSize szResult(50, 75);
ASSERT(sz1 == szResult);

// works with SIZE, too

sz1 = CSize(100, 100);
SIZE sz3;
sz3.cx = 50;
sz3.cy = 25;

sz1 -= sz3;
ASSERT(sz1 == szResult);
```

See Also

Reference

CSize Class

Hierarchy Chart

CSize::operator -

Other Resources

CSize Members

CSize::operator +

These operators add this **CSize** value to the value of parameter.

```
CSize operator +(
   SIZE size
) const throw( );
CPoint operator +(
   POINT point
) const throw( );
CRect operator +(
   const RECT* lpRect
) const throw( );
```

Remarks

See the following descriptions of the individual operators:

- **operator +(** *size* **)** This operation adds two **CSize** values.
- **operator +(** *point* **)** This operation offsets (moves) a POINT (or CPoint) value by this **CSize** value. The **cx** and **cy** members of this **CSize** value are added to the **x** and **y** data members of the **POINT** value. It is analogous to the version of CPoint::operator + that takes a SIZE parameter.
- **operator +(** *lpRect* **)** This operation offsets (moves) a RECT (or CRect) value by this **CSize** value. The **cx** and **cy** members of this **CSize** value are added to the **left**, **top**, **right**, and **bottom** data members of the **RECT** value. It is analogous to the version of CRect::operator + that takes a SIZE parameter.

Example

```
CSize sz1(100, 100);
CSize sz2( 50,  25);
CSize szOut;

szOut = sz1 + sz2;

CSize szResult(150, 125);
ASSERT(szOut == szResult);

// works with SIZE, too

sz1 = CSize(100, 100);
SIZE sz3;
sz3.cx = 50;
sz3.cy = 25;

szOut = sz1 + sz3;
ASSERT(szOut == szResult);
```

See Also

Reference

CSize Class
Hierarchy Chart
CPoint::operator +
CRect::operator +

Other Resources

CSize Members

CSize::operator -

The first three of these operators subtract this **CSize** value to the value of parameter.

```
CSize operator -(
   SIZE size
) const throw( );
CPoint operator -(
   POINT point
) const throw( );
CRect operator -(
   const RECT* lpRect
) const throw( );
CSize operator -( ) const throw( );
```

Remarks

The fourth operator, the unary minus, changes the sign of the **CSize** value. See the following descriptions of the individual operators:

- **operator -(** *size* **)** This operation subtracts two **CSize** values.
- **operator -(** *point* **)** This operation offsets (moves) a POINT or CPoint value by the additive inverse of this **CSize** value. The **cx** and **cy** of this **CSize** value are subtracted from the **x** and **y** data members of the **POINT** value. It is analogous to the version of CPoint::operator - that takes a SIZE parameter.
- **operator -(** *lpRect* **)** This operation offsets (moves) a RECT or CRect value by the additive inverse of this **CSize** value. The **cx** and **cy** members of this **CSize** value are subtracted from the **left**, **top**, **right**, and **bottom** data members of the **RECT** value. It is analogous to the version of CRect::operator - that takes a SIZE parameter.
- **operator -( )** This operation returns the additive inverse of this **CSize** value.

Example

```
CSize sz1(100, 100);
CSize sz2( 50,  25);
CSize szOut;

szOut = sz1 - sz2;

CSize szResult(50, 75);
ASSERT(szOut == szResult);

// works with SIZE, too

sz1 = CSize(100, 100);
SIZE sz3;
sz3.cx = 50;
sz3.cy = 25;

szOut = sz1 - sz3;
ASSERT(szOut == szResult);
```

See Also

Reference

CSize Class
Hierarchy Chart
CPoint::operator -
CRect::operator -

Other Resources

CSize Members

CStrBufT Class

This class provides automatic resource cleanup for **GetBuffer** and **ReleaseBuffer** calls on an existing **CStringT** object.

```
template<
   typename TCharType
>
class CStrBufT
```

Parameters

TCharType

The character type of the **CStrBufT** class. Can be one of the following:

- **char** (for ANSI character strings)
- **wchar_t** (for Unicode character strings)
- **TCHAR** (for both ANSI and Unicode character strings)

Remarks

This class is used as a wrapper class for replacing calls to GetBuffer and ReleaseBuffer, or GetBufferSetLength and **ReleaseBuffer**.

Primarily designed as a helper class, **CStrBufT** provides a convenient way for a developer to work with the character buffer of a string object without worrying about how or when to call **ReleaseBuffer**. This is possible because the wrapper object goes out of scope naturally in the case of an exception or multiple exiting code paths; causing its destructor to free the string resource.

Requirements

Header: atlsimpstr.h

Example

See the ISAPIFilter Sample.

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart

Other Resources

CStrBufT Members

ATL/MFC Shared Classes

CStrBufT Members

Methods

CStrBufT	The constructor for the string buffer object.
SetLength	Sets the character buffer length of the associated string object.

Operators

operator PCXSTR	Retrieves a **const** pointer to the character buffer of the associated string object.
operator PXSTR	Retrieves a pointer to the character buffer of the associated string object.

Data Members

AUTO_LENGTH	Automatically determine the new length of the string at release.
SET_LENGTH	Set the length of the string object at GetBuffer time

Typedefs

PCXSTR	A pointer to a constant string.
PXSTR	A pointer to a string.
StringType	The string type whose buffer is to be manipulated by specializations of this class template.

See Also

Reference

CStrBufT Class

CStrBufT Methods

For information about the methods in **CStrBufT**, see CStrBufT Members.

CStrBufT::CStrBufT

Constructs a buffer object.

```
CStrBufT(
   StringType& str,
   int nMinLength,
   DWORD dwFlags = AUTO_LENGTH
) throw(...);
explicit CStrBufT(
   StringType& str
) throw(...);
```

Parameters

str

The string object associated with the buffer. Typically, the developer will use the predefined typedefs of **CStrBuf** (**TCHAR** variant), **CStrBufA** (**char** variant) and **CStrBufW** (**wchar_t** variant).

nMinLength

The minimum length of the character buffer.

dwFlags

Determines if the string length is automatically determined. Can be one of the following:

- **AUTO_LENGTH** String length is automatically determined when CSimpleStringT::Release is called. The string must be null-terminated. Default value.
- **SET_LENGTH** String length is set when CSimpleStringT::GetBuffer is called.

Remarks

Creates a string buffer for the associated string object. During construction, CSimpleStringT::GetBuffer or CSimpleStringT::GetBufferSetLength is called.

Note that the copy constructor is **private**.

See Also

Reference

CStrBufT Class

CStrBufT::StringType

Other Resources

CStrBufT Members

CStrBufT::SetLength

Sets the length of the character buffer.

```
void SetLength(
   int nLength
);
```

Parameters

nLength

The new length of the character buffer of the string object.

Note
Must be less than or equal to the minimum buffer length specified in the constructor of **CStrBufT**.

Remarks

Call this function to set the length of the string represented by the buffer object.

See Also

Reference

CStrBufT Class

Other Resources

CStrBufT Members

CStrBufT Operators

For information about the operators in **CStrBufT**, see CStrBufT Members.

CStrBufT::operator PCXSTR

Directly accesses characters stored in the associated string object as a C-style string.

```
operator PCXSTR( ) const throw( );
```

Return Value

A character pointer to the string's data.

Remarks

Call this function to return a pointer to the character buffer of a string object. The contents of the string object cannot be changed with this pointer.

See Also

Reference

CStrBufT Class
CStrBufT::operator PXSTR
CStrBufT::PCXSTR

Other Resources

CStrBufT Members

CStrBufT::operator PXSTR

Directly accesses characters stored in the associated string object as a C-style string.

```
operator PXSTR( ) throw( );
```

Return Value

A character pointer to the string's data.

Remarks

Call this function to return a pointer to the character buffer of a string object. The developer may change the contents of the string object with this pointer.

See Also

Reference

CStrBufT Class
CStrBufT::operator PCXSTR
CStrBufT::PXSTR

Other Resources

CStrBufT Members

CStrBufT Data Members

For information about the data members in **CStrBufT**, see CStrBufT Members.

CStrBufT::AUTO_LENGTH

Automatically determine the new length of the string at release.

```
static const DWORD AUTO_LENGTH = 0x01;
```

Remarks

Automatically determine the new length of the string at release. The string must be null-terminated.

See Also

Reference

CStrBufT Class

CStrBufT::CStrBufT

Other Resources

CStrBufT Members

CStrBufT::SET_LENGTH

Set the length of the string object at **GetBuffer** time.

```
static const DWORD SET_LENGTH = 0x02;
```

Remarks

Set the length of the string object at GetBuffer time.

Determines if CSimpleStringT::GetBuffer and CSimpleStringT::GetBufferSetLength are called when the string buffer object is constructed.

See Also

Reference

CStrBufT Class

CStrBufT::CStrBufT

Other Resources

CStrBufT Members

CStrBufT Typedefs

For information about the typedefs in **CStrBufT**, see CStrBufT Members.

CStrBufT::PCXSTR

A pointer to a constant string.

```
typedef CSimpleStringT< TCharType >::PCXSTR PCXSTR;
```

See Also

Reference

CStrBufT Class

CStrBufT::operator PCXSTR

CStrBufT::PXSTR

Other Resources

CStrBufT Members

CStrBufT::PXSTR

A pointer to a string.

```
typedef CSimpleStringT< TCharType >::PXSTR PXSTR;
```

See Also

Reference

CStrBufT Class

CStrBufT::operator PXSTR

CStrBufT::PCXSTR

Other Resources

CStrBufT Members

CStrBufT::StringType

The string type whose buffer is to be manipulated by specializations of this class template.

```
typedef CSimpleStringT< TCharType > StringType;
```

Remarks

TCharType is the character type used to specialize the class template.

See Also

Reference

CStrBufT Class

Other Resources

CStrBufT Members

CStringData Class

This class represents the data of a string object.

```
struct CStringData
```

Remarks

This class should only be used by developers implementing custom string managers. For more information on custom string managers, see Memory Management and CStringT

This class encapsulates various types of information and data associated with a higher string object, such as CStringT, CSimpleStringT, or CFixedStringT objects. Every higher string object contains a pointer to its associated **CStringData** object, allowing multiple string objects to point to the same string data object. This relationship is represented by the reference count (**nRefs**) of the **CStringData** object.

Note
In certain cases, a string type (such as **CFixedString**) will not share a string data object with more than one higher string object. For more information on this, see Memory Management and CStringT.

This data is composed of:

- The memory manager (of type IAtlStringMgr) of the string.
- The current length (nDataLength) of the string.
- The allocated length (nAllocLength) of the string. For performance reasons, this can differ from the current string length
- The current reference count (nRefs) of the **CStringData** object. This value is used in determining how many string objects are sharing the same **CStringData** object.
- The actual character buffer (data) of the string.

Note
The actual character buffer of the string object is allocated by the string manager and is appended to the **CStringData** object.

Requirements

Header: atlsimpstr.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart

Other Resources

CStringData Members

ATL/MFC Shared Classes

CStringData Members

Methods

AddRef	Increments the reference count of the string data object.
data	Retrieves the character data of a string object.
IsLocked	Determines if the buffer of the associated string object is locked.
IsShared	Determines if the buffer of the associated string object is currently shared.
Lock	Locks the buffer of the associated string object.
Release	Releases the specified string object.
Unlock	Unlocks the buffer of the associated string object.

Data Members

nAllocLength	Length of allocated data in **XCHAR**s (not including terminating null)
nDataLength	Length of currently used data in **XCHAR**s (not including terminating null)
nRefs	The current reference count of the object.
pStringMgr	A pointer to the string manager of this string object.

See Also

Reference

CStringData Class

CStringData Methods

For information about the methods in **CStringData**, see CStringData Members

CStringData::AddRef

Increments the reference count of the string object.

```
void AddRef( ) throw( );
```

Remarks

Increments the reference count of the string object.

Note
Do not call this method on a string with a negative reference count, since a negative count indicates that the string buffer is locked.

See Also

Reference

CStringData Class

Other Resources

CStringData Members

CStringData::data

Returns a pointer to the character buffer of a string object.

```
void* data( ) throw( );
```

Return Value

A pointer to the character buffer of the string object.

Remarks

Call this function to return the current character buffer of the associated string object.

Note
This buffer is not allocated by the **CStringData** object but by the string manager when needed. When allocated, the buffer is appended to the string data object.

See Also

Reference

CStringData Class

Other Resources

CStringData Members

CStringData::IsLocked

Determines if the character buffer is locked.

```
bool IsLocked( ) const throw( );
```

Return Value

Returns **true** if the buffer is locked; otherwise **false**.

Remarks

Call this function to determine if the character buffer of a string object is currently locked.

See Also

Reference

CStringData Class

Other Resources

CStringData Members

CStringData::IsShared

Determines if the character buffer is shared.

```
bool IsShared( ) const throw( );
```

Return Value

Returns **true** if the buffer is shared; otherwise **false**.

Remarks

Call this function to determine if the character buffer of a string data object is currently shared among multiple string objects.

See Also

Reference

CStringData Class

Other Resources

CStringData Members

CStringData::Lock

Locks the character buffer of the associated string object.

```
void Lock( ) throw( );
```

Remarks

Call this function to lock the character buffer of the string data object. Locking and unlocking is used when direct access to the character buffer is required by the developer. A good example of locking is demonstrated by the LockBuffer and UnlockBuffer methods of **CSimpleStringT**.

Note
A character buffer can only be locked if the buffer is not shared among higher string objects.

See Also

Reference

CStringData Class

CStringData::Unlock

Other Resources

CStringData Members

CStringData::Release

Decrements the reference count of the string data object.

```
void Release( ) throw( );
```

Remarks

Call this function to decrement the reference count, freeing the **CStringData** structure if the reference count hits zero. This is commonly done when a string object is deleted, and therefore no longer needs to reference the string data object.

For example, the following code would call **CStringData::Release** for the string data object associated with `str1`:

```
{
 CString str1= "Hello world";  // Allocates new CStringData
}
// str1 is deleted when it goes out of scope, so it releases its string data
```

See Also

Reference

CStringData Class

Other Resources

CStringData Members

CStringData::Unlock

Unlocks the character buffer of the associated string object.

```
void Unlock( ) throw( );
```

Remarks

Call this function to unlock the character buffer of the string data object. Once a buffer is unlocked, it is shareable and can be reference counted.

Note
Each call to **Lock** must be matched by a corresponding call to **Unlock**.

Locking and unlocking is used when the developer must ensure that the string data not be shared. A good example of locking is demonstrated by the LockBuffer and UnlockBuffer methods of **CSimpleStringT**.

See Also

Reference

CStringData Class

CStringData::Lock

Other Resources

CStringData Members

CStringData Data Members

For information about the data members in **CStringData**, see CStringData Members

CStringData::nAllocLength

Length of the allocated character buffer.

```
int nAllocLength;
```

Remarks

Stores the length of the allocated data buffer in **XCHAR**s (not including terminating null).

See Also

Reference

CStringData Class

CStringData::nDataLength

Other Resources

CStringData Members

CStringData::nDataLength

Current length of the string object.

```
int nDataLength;
```

Remarks

Stores the length of currently used data in **XCHAR**s (not including terminating null).

See Also

Reference

CStringData Class

CStringData::nAllocLength

Other Resources

CStringData Members

CStringData::nRefs

Reference count of the string data object.

```
long nRefs;
```

Remarks

Stores the reference count of the string data object. This count indicates the number of higher string objects that are associated with the string data object. A negative value indicates that the string data object is currently locked.

See Also

Reference

CStringData Class
CStringData::Lock
CStringData::Unlock

Other Resources

CStringData Members

CStringData::pStringMgr

The memory manager of the associated string object.

```
IAtlStringMgr* pStringMgr;
```

Remarks

Stores the memory manager for the associated string object. For more information on memory managers and strings, see Memory Management and CStringT.

See Also

Reference

CStringData Class

IAtlStringMgr Class

Other Resources

CStringData Members

CStringT Class

This class represents a **CStringT** object.

```
template< typename BaseType, class StringTraits >
class CStringT :
public CSimpleStringT<   BaseType,   _CSTRING_IMPL_::_MFCDLLTraitsCheck<      BaseType,
StringTraits   >   ::c_bIsMFCDLLTraits>
```

Parameters

BaseType

The character type of the string class. Can be one of the following:

- **char** (for ANSI character strings).
- **wchar_t** (for Unicode character strings).
- **TCHAR** (for both ANSI and Unicode character strings).

StringTraits

Determines if the string class needs C Run-Time (CRT) Library support and where string resources are located. Can be one of the following:

- **StrTraitATL< wchar_t | char | TCHAR, ChTraitsCRT< wchar_t | char | TCHAR > >**

  The class requires CRT support and searches for resource strings in the module specified by **m_hInstResource** (a member of the application's module class).

- **StrTraitATL< wchar_t | char | TCHAR, ChTraitsOS< wchar_t | char | TCHAR > >**

  The class does not require CRT support and searches for resource strings in the module specified by **m_hInstResource** (a member of the application's module class).

- **StrTraitMFC< wchar_t | char | TCHAR, ChTraitsCRT< wchar_t | char | TCHAR > >**

  The class requires CRT support and searches for resource strings using the standard MFC search algorithm.

- **StrTraitMFC< wchar_t | char | TCHAR, ChTraitsOS< wchar_t | char | TCHAR > >**

  The class does not require CRT support and searches for resource strings using the standard MFC search algorithm.

Remarks

CStringT inherits from CSimpleStringT Class. Advanced features, such as character manipulation, ordering, and searching, are implemented by **CStringT**.

Note
CStringT objects are capable of throwing exceptions. This occurs when a **CStringT** object runs out of memory for any reaso n.

A **CStringT** object consists of a variable-length sequence of characters. **CStringT** provides functions and operators using a syntax similar to that of Basic. Concatenation and comparison operators, together with simplified memory management, make **CStringT** objects easier to use than ordinary character arrays.

By using different combinations of the *BaseType* and *StringTraits* parameters, **CStringT** objects can come in the following types, which are have been predefined by the ATL libraries.

If using in an ATL application:

CString, **CStringA**, and **CStringW** are exported from the MFC DLL (MFC80.DLL), never from user DLLs. This is done to prevent **CStringT** from being multiply defined.

Note

If you encountered linker errors when exporting a **CString**-derived class from an MFC extension DLL in Visual C++ .NET 2002 and have applied the workaround as described in the Knowledge Base article, "Linking Errors When You Import CString-Derived Classes" (), you should remove the workaround code, because this has been fixed in Visual C++ .NET 2003. You can find Knowledge Base articles on the MSDN Library CD-ROM or at http://support.microsoft.com/support.

If using in an MFC application:

CStringT type	Declaration
CStringA	An ANSI character type string with CRT support.
CStringW	A Unicode character type string with CRT support.
CString	Both ANSI and Unicode character types with CRT support.

If **ATL_CSTRING_NO_CRT** is defined:

CStringT type	Declaration
CAtlStringA	An ANSI character type string without CRT support.
CAtlStringW	A Unicode character type string without CRT support.
CAtlString	Both ANSI and Unicode character types without CRT support.

If **ATL_CSTRING_NO_CRT** is not defined:

CStringT type	Declaration
CAtlStringA	An ANSI character type string with CRT support.
CAtlStringW	A Unicode character type string with CRT support.
CAtlString	Both ANSI and Unicode character types with CRT support.

CString objects also have the following characteristics:

- **CStringT** objects can grow as a result of concatenation operations.
- **CStringT** objects follow "value semantics." Think of a **CStringT** object as an actual string, not as a pointer to a string.
- You can freely substitute **CStringT** objects for **PCXSTR** function arguments.
- Custom memory management for string buffers. For more information, see Memory Management and CStringT.

CStringT Predefined Types

Because **CStringT** uses a template argument to define the character type (either wchar_t or char) supported, method parameter types can be complicated at times. To simplify this issue, a set of predefined types is defined and used throughout the **CStringT** class. The following table lists the various types:

Name	Description
XCHAR	A single character (either **wchar_t** or **char**) with the same character type as the **CStringT** object.
YCHAR	A single character (either **wchar_t** or **char**) with the opposite character type as the **CStringT** object.
PXSTR	A pointer to a character string (either **wchar_t** or **char**) with the same character type as the **CStringT** object.
PYSTR	A pointer to a character string (either **wchar_t** or **char**) with the opposite character type as the **CStringT** object.

PCXSTR	A pointer to a **const** character string (either **wchar_t** or **char**) with the same character type as the **CStringT** object.
PCYSTR	A pointer to a **const** character string (either **wchar_t** or **char**) with the opposite character type as the **CStringT** object.

Note
Code that previously used undocumented methods of **CString** (such as **AssignCopy**) must be replaced with code that uses the following documented methods of **CStringT** (such as **GetBuffer** or **ReleaseBuffer**).

Requirements

Header	**Use for**
cstringt.h	MFC-only string objects
atlstr.h	Non-MFC string objects

Example

See the following samples:

- ISAPIFilter Sample
- MantaWeb Sample
- ShowLocalized Sample

See Also

Reference

Hierarchy Chart

Other Resources

CStringT Members

ATL/MFC Shared Classes

CStringT Members

Constructors/Destructors

CStringT::CStringT	Constructs a **CStringT** object in various ways.
CStringT::~CStringT	Destroys a **CStringT** object.

Methods

CStringT::AllocSysString	Allocates a **BSTR** from **CStringT** data.
CStringT::AnsiToOem	Makes an in-place conversion from the ANSI character set to the OEM character set.
CStringT::AppendFormat	Appends formatted data to an existing **CStringT** object.
CStringT::Collate	Compares two strings (case sensitive, uses locale-specific information).
CStringT::CollateNoCase	Compares two strings (case insensitive, uses locale-specific information).
CStringT::Compare	Compares two strings (case sensitive).
CStringT::CompareNoCase	Compares two strings (case insensitive).
CStringT::Delete	Deletes a character or characters from a string.
CStringT::Find	Finds a character or substring inside a larger string.
CStringT::FindOneOf	Finds the first matching character from a set.
CStringT::Format	Formats the string as **sprintf** does.
CStringT::FormatMessage	Formats a message string.
CStringT::FormatMessageV	Formats a message string using a variable argument list.
CStringT::FormatV	Formats the string using a variable list of arguments.
CStringT::GetEnvironmentVariable	Sets the string to the value of the specified environment variable.
CStringT::Insert	Inserts a single character or a substring at the given index within the string.
CStringT::Left	Extracts the left part of a string.
CStringT::LoadString	Loads an existing **CStringT** object from a Windows resource.
CStringT::MakeLower	Converts all the characters in this string to lowercase characters.
CStringT::MakeReverse	Reverses the string.
CStringT::MakeUpper	Converts all the characters in this string to uppercase characters.
CStringT::Mid	Extracts the middle part of a string.

CStringT::OemToAnsi	Makes an in-place conversion from the OEM character set to the ANSI character set.
CStringT::Remove	Removes indicated characters from a string.
CStringT::Replace	Replaces indicated characters with other characters.
CStringT::ReverseFind	Finds a character inside a larger string; starts from the end.
CStringT::Right	Extracts the right part of a string.
CStringT::SetSysString	Sets an existing **BSTR** object with data from a **CStringT** object.
CStringT::SpanExcluding	Extracts characters from the string, starting with the first character, that are not in the set of characters identified by *pszCharSet*.
CStringT::SpanIncluding	Extracts a substring that contains only the characters in a set.
CStringT::Tokenize	Extracts specified tokens in a target string.
CStringT::Trim	Trims all leading and trailing whitespace characters from the string.
CStringT::TrimLeft	Trims leading whitespace characters from the string.
CStringT::TrimRight	Trims trailing whitespace characters from the string.

Operators

CStringT::operator =	Assigns a new value to a **CStringT** object.
CStringT::operator +	Concatenates two strings or a character and a string.
CStringT::operator +=	Concatenates a new string to the end of an existing string.
CStringT::operator ==	Determines if two strings are logically equal.
CStringT::operator !=	Determines if two strings are logically not equal.
CStringT::operator <	Determines if the string on the left side of the operator is less than to the string on the right side.
CStringT::operator >	Determines if the string on the left side of the operator is greater than to the string on the right side.
CStringT::operator <=	Determines if the string on the left side of the operator is less than or equal to the string on the right side.
CStringT::operator >=	Determines if the string on the left side of the operator is greater than or equal to the string on the right side.

See Also

Reference

CStringT Class

CStringT Methods

For information about the methods in **CStringT**, see CStringT Members.

CStringT::AllocSysString

Allocates an Automation-compatible string of the type **BSTR** and copies the contents of the **CStringT** object into it, including the terminating null character.

```
BSTR AllocSysString() const;
```

Return Value

The newly allocated string.

Remarks

A CMemoryException Class is thrown if insufficient memory exists. This function is normally used to return strings for Automation.

Commonly, if this string is passed to a COM function as an [in] parameter, then this requires the caller to free the string. This can be done by using SysFreeString, as described in the Platform SDK. For more information, see Allocating and Releasing Memory for a BSTR.

For more information about OLE allocation functions in Windows, see SysAllocString in the Platform SDK.

Example

The following example demonstrates the use of **CStringT::AllocSysString**.

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString str("This is a test string!");
BSTR bstr = str.AllocSysString();

// bstr now contains "This is a test string!", and can be
// passed to any OLE function requiring a BSTR.
// Normally, if you pass the BSTR, you will
// need to free the string after returning from the function call.
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::AnsiToOem

Converts all the characters in this **CStringT** object from the ANSI character set to the OEM character set.

```
void AnsiToOem();
```

Remarks

The function is not available if **_UNICODE** is defined.

Example

```
// OEM character 252 on most IBM-compatible computers in
// Western countries/regions is superscript n, as in 2^n.
// Converting it to the ANSI English charset results in a
// normal character 'n', which is the closest possible
// representation.
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString str((TCHAR)252);
str.OemToAnsi();
_ASSERT(str[0] == 'n');

// Be aware that in OEM to ANSI conversion the 'n'
// from the previous result cannot be converted back to
// a supsercript n because the system does not know what
// the character's value truly was.
str.AnsiToOem();
_ASSERT(str[0] != 252);
_ASSERT(str[0] == 'n');
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::AppendFormat

Appends formatted data to an existing **CStringT** object.

```
void __cdecl AppendFormat(
   PCXSTR pszFormat,
   [, argument]...
);
void __cdecl AppendFormat(
   UINT nFormatID,
   [, argument]...
);
```

Parameters

pszFormat

A format-control string.

nFormatID

The string resource identifier that contains the format-control string.

argument

Optional arguments.

Remarks

This function formats and appends a series of characters and values in the **CStringT**. Each optional argument (if any) is converted and appended according to the corresponding format specification in *pszFormat* or from the string resource identified by *nFormatID*.

Example

```
// cstringt_appendformat.cpp
#include <stdio.h>
#include <atlstr.h>

int main()
{
    //typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

    CAtlString str = "Some data:\t";

    str.AppendFormat(_T("X value = %.2f\n"), 12345.12345);
    _tprintf_s("%s", (LPCTSTR) str);

   return 0;
}
```

Output

```
Some data:         X value = 12345.12
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::Collate

Compares two strings using the generic-text function **_tcscoll**.

```
int Collate(
   PCXSTR psz
) const throw();
```

Parameters

psz

The other string used for comparison.

Return Value

Zero if the strings are identical, < 0 if this **CStringT** object is less than *psz*, or > 0 if this **CStringT** object is greater than *psz*.

Remarks

The generic-text function **_tcscoll**, which is defined in TCHAR.H, maps to either **strcoll**, **wcscoll**, or **_mbscoll**, depending on the character set that is defined at compile time. Each function performs a case-sensitive comparison of the strings according to the code page currently in use. For more information, see strcoll, wcscoll, _mbscoll, _strcoll_l, _wcscoll_l, _mbscoll_l.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;
setlocale(LC_ALL,"german");

CAtlString str1 = _T("strasse");
CAtlString str2 = _T("straße");

int n;

n = str1.Collate(str2);
_ASSERT(n == 0);

// Comparison is a strict ASCII comparison with no language rules.
n = str1.Compare(str2);
_ASSERT(n < 0);
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::CollateNoCase

Compares two strings using the generic-text function **_tcscoll**.

```
int CollateNoCase(
   PCXSTR psz
) const throw();
```

Parameters

psz

The other string used for comparison.

Return Value

Zero if the strings are identical (ignoring case), < 0 if this **CStringT** object is less than *psz* (ignoring case), or > 0 if this **CStringT** object is greater than *psz* (ignoring case).

Remarks

The generic-text function **_tcscoll**, which is defined in TCHAR.H, maps to either **stricoll**, **wcsicoll**, or **_mbsicoll**, depending on the character set that is defined at compile time. Each function performs a case-insensitive comparison of the strings, according to the code page currently in use. For more information, see strcoll, wcscoll, _mbscoll, _strcoll_l, _wcscoll_l, _mbscoll_l.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString str1 = _T("Co-Op");
CAtlString str2 = _T("con");

int n;

// Collation uses language rules, such as ignoring dashes.
// NoCase version ignores case.
n = str1.CollateNoCase(str2);
_ASSERT(n > 0);

// Comparison is a strict ASCII comparison with no language rules
// but still ignores case in NoCase version.
n = str1.CompareNoCase(str2);
_ASSERT(n < 0);
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::Compare

Compares two strings (case sensitive).

```
int Compare(
   PCXSTR psz
) const;
```

Parameters

psz

The other string used for comparison.

Return Value

Zero if the strings are identical, < 0 if this **CStringT** object is less than *psz*, or > 0 if this **CStringT** object is greater than *psz*.

Remarks

The generic-text function **_tcscmp**, which is defined in TCHAR.H, maps to either **strcmp**, **wcscmp**, or **_mbscmp**, depending on the character set that is defined at compile time. Each function performs a case-sensitive comparison of the strings and is not affected by locale. For more information, see strcmp, wcscmp, _mbscmp.

If the string contains embedded nulls, for purposes of comparison the string is considered to be truncated at the first embedded null character.

Example

The following example demonstrates the use of **CStringT::Compare**.

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString s1( "abc" );
CAtlString s2( "abd" );
_ASSERT( s1.Compare( s2 ) < 0 );    // Compare with another CAtlString.
_ASSERT( s1.Compare( "abe" ) < 0 ); // Compare with LPTSTR string.
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::CompareNoCase

Compares two strings (case insensitive).

```
int CompareNoCase(
   PCXSTR psz
) const throw();
```

Parameters

psz

The other string used for comparison.

Return Value

Zero if the strings are identical (ignoring case), <0 if this **CStringT** object is less than *psz* (ignoring case), or >0 if this **CStringT** object is greater than *psz* (ignoring case).

Remarks

The generic-text function **_tcsicmp**, which is defined in TCHAR.H, maps to either **_stricmp**, **_wcsicmp** or **_mbsicmp**, depending on the character set that is defined at compile time. Each function performs a case-insensitive comparison of the strings. The comparison depends on the **LC_CTYPE** aspect of the locale but not **LC_COLLATE**. For more information, see _stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString s1( "abc" );
CAtlString s2( "ABD" );
_ASSERT( s1.CompareNoCase( s2 ) < 0 ); // Compare with a CAtlString.
_ASSERT( s1.CompareNoCase( "ABE" ) < 0 ); // Compare with LPTSTR string.
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::CStringT

Constructs a **CStringT** object.

```
CStringT( ) throw() :
   CThisSimpleString(
      StringTraits::GetDefaultManager()
   );

explicit CStringT(
   IAtlStringMgr* pStringMgr
) throw() :
   CThisSimpleString(
            pStringMgr
   );

CStringT(
   const VARIANT& varSrc
);

CStringT(
   const VARIANT& varSrc,
   IAtlStringMgr* pStringMgr
);

CStringT(
  const CStringT& strSrc
) :
   CThisSimpleString(
            strSrc
   );

operator CSimpleStringT<
      BaseType,
   !_CSTRING_IMPL_::_MFCDLLTraitsCheck<
            BaseType,
            StringTraits
   >::   c_bIsMFCDLLTraits
> &()

template <
   bool bMFCDLL
>
CStringT(
   const CSimpleStringT<
            BaseType,
            bMFCDLL
   > & strSrc
) :
   CThisSimpleString(
            strSrc
);

template <
   class SystemString
>
CStringT(
   SystemString^ pString
) :
   CThisSimpleString(
      StringTraits::GetDefaultManager()
   );

CStringT(
```

```
   const XCHAR* pszSrc
) :
   CThisSimpleString(
      StringTraits::GetDefaultManager()
   );

CSTRING_EXPLICIT CStringT(
   const YCHAR* pszSrc
) :
   CThisSimpleString(
      StringTraits::GetDefaultManager()
   );

CStringT(
   LPCSTR pszSrc,
   IAtlStringMgr* pStringMgr
) :
   CThisSimpleString(
            pStringMgr
   );

CStringT(
   LPCWSTR pszSrc,
   IAtlStringMgr* pStringMgr
) :
   CThisSimpleString(
            pStringMgr
);

CSTRING_EXPLICIT CStringT(
   const unsigned char* pszSrc
) :
   CThisSimpleString(
      StringTraits::GetDefaultManager()
   );

/*CSTRING_EXPLICIT*/ CStringT(
   char* pszSrc
) :
   CThisSimpleString(
      StringTraits::GetDefaultManager()
   );

CSTRING_EXPLICIT CStringT(
   unsigned char* pszSrc
) :
   CThisSimpleString(
      StringTraits::GetDefaultManager()
   );

CSTRING_EXPLICIT CStringT(
   wchar_t* pszSrc
) :
   CThisSimpleString(
      StringTraits::GetDefaultManager()
   );

CStringT(
   const unsigned char* pszSrc,
   IAtlStringMgr* pStringMgr
) :
   CThisSimpleString(
            pStringMgr
   );

CSTRING_EXPLICIT CStringT(
   char ch,
   int nLength = 1
```

```
) :
   CThisSimpleString(
      StringTraits::GetDefaultManager()
   );

CSTRING_EXPLICIT CStringT(
   wchar_t ch,
   int nLength = 1
) :
   CThisSimpleString(
      StringTraits::GetDefaultManager()
   );

CStringT(
   const XCHAR* pch,
   int nLength
) :
   CThisSimpleString(
            pch,
            nLength,
      StringTraits::GetDefaultManager()
   );

CStringT(
   const YCHAR* pch,
   int nLength
) :
   CThisSimpleString(
      StringTraits::GetDefaultManager()
   );

CStringT(
   const XCHAR* pch,
   int nLength,
   IAtlStringMgr* pStringMgr
) :
   CThisSimpleString(
            pch,
            nLength,
            pStringMgr
);

CStringT(
   const YCHAR* pch,
   int nLength,
   IAtlStringMgr* pStringMgr
) :
   CThisSimpleString(
            pStringMgr
   );
```

Parameters

pch

A pointer to an array of characters of length *nLength*, not null-terminated.

nLength

A count of the number of characters in *pch*.

ch

A single character.

pszSrc

A null-terminated string to be copied into this **CStringT** object.

pStringMgr

A pointer to the memory manager for the **CStringT** object. For more information on **IAtlStringMgr** and memory management for **CStringT**, see Memory Management with CStringT.

strSrc

An existing **CStringT** object to be copied into this **CStringT** object. For more information on **CThisString** and **CThisSimpleString**, see the Remarks section.

varSrc

A variant object to be copied into this **CStringT** object.

BaseType

The character type of the string class. Can be one of the following:

char (for ANSI character strings).

wchar_t (for Unicode character strings).

TCHAR (for both ANSI and Unicode character strings).

bMFCDLL

Boolean that specifies whether the project is an MFC DLL (TRUE) or not (FALSE).

SystemString

Must be **System::String**, and the project must be compiled with /clr.

pString

A handle for a **CStringT** object.

Remarks

Because the constructors copy the input data into new allocated storage, you should be aware that memory exceptions may result. Note that some of these constructors act as conversion functions. This allows you to substitute, for example, an **LPTSTR** where a **CStringT** object is expected.

- **CStringT**(**LPCSTR** *lpsz*): Constructs a Unicode **CStringT** from an ANSI string. You can also use this constructor to load a string resource as shown in the example below.
- **CStringT(LPCWSTR** *lpsz*): Constructs a **CStringT** from a Unicode string.
- **CStringT**(**const unsigned char*** *psz*): Allows you to construct a **CStringT** from a pointer to **unsigned char**.

Note that the *strSrc* parameter can be either a **CStringT** or **CThisSimpleString** object. For **CStringT**, use one of its default instantiations (**CString**, **CStringA**, or **CStringW**); for **CThisSimpleString**, use a **this** pointer. **CThisSimpleString** declares an instance of the CSimpleStringT Class, which is a smaller string class with little built-in functionality than the **CStringT** class.

The overload operator **CSimpleStringT<>&()** constructs a **CStringT** object from a **CSimpleStringT** declaration.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString s1;                    // Empty string
CAtlString s2( "cat" );           // From a C string literal
CAtlString s3 = s2;               // Copy constructor
CAtlString s4( s2 + " " + s3 );   // From a string expression

CAtlString s5( 'x' );             // s5 = "x"
CAtlString s6( 'x', 6 );          // s6 = "xxxxxx"

CAtlString s7((LPCSTR)ID_FILE_NEW); // s7 = "Create a new document"

VARIANT var;
V_VT(&var) = VT_BSTR;
V_BSTR(&var) = ::SysAllocString(L"Football is a fun sport.");
CAtlString s8(var); // s8 = "Football is a fun sport."

// The following statement does not call the assignment operator.
```

```
// The compiler considers the following statement equivalent to
// CAtlString city("Paris")
CAtlString city = "Paris";
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::~CStringT

Destroys the **CStringT** object.

```
~CStringT() throw();
```

Remarks

Destroys the **CStringT** object.

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::Delete

Deletes a character or characters from a string starting with the character at the given index.

```
int Delete(
   int iIndex,
   int nCount = 1
);
```

Parameters

iIndex

The zero-based index of the first character in the **CStringT** object to delete.

nCount

The number of characters to be removed.

Return Value

The length of the changed string.

Remarks

If *nCount* is longer than the string, the remainder of the string will be removed.

Example

```
// cstringt_delete.cpp
#include <atlstr.h>
#include <stdio.h>

int main( int argc, char* argv[] )
{
    //typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

    CAtlString str( "Soccer is best, but liquor is quicker!");
    printf_s("Before: %s\n", (LPCTSTR) str);

    int n = str.Delete(6, 3);
    printf_s("After: %s\n", (LPCTSTR) str);
    _ASSERT(n == str.GetLength());

    return 0;
}
```

Output

```
Before: Soccer is best, but liquor is quicker!
After: Soccer best, but liquor is quicker!
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::Find

Searches this string for the first match of a character or substring.

```
int Find(
   PCXSTR pszSub,
   int iStart=0
) const throw( );
int Find(
   XCHAR ch,
   int iStart=0
) const throw( );
```

Parameters

pszSub

A substring to search for.

iStart

The index of the character in the string to begin the search with, or 0 to start from the beginning.

ch

A single character to search for.

Return Value

The zero-based index of the first character in this **CStringT** object that matches the requested substring or characters; -1 if the substring or character is not found.

Remarks

The function is overloaded to accept both single characters (similar to the run-time function **strchr**) and strings (similar to **strstr**).

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString s( "abcdef" );
_ASSERT( s.Find( 'c' ) == 2 );
_ASSERT( s.Find( "de" ) == 3 );

CAtlString str("The waves are still");
int n = str.Find('e', 5);
_ASSERT(n == 7);
```

For another example, see ISAPIFilter Sample: Maps URLs to Parameterized Queries.

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::FindOneOf

Searches this string for the first character that matches any character contained in *pszCharSet*.

```
int FindOneOf(
   PCXSTR pszCharSet
) const throw( );
```

Parameters

pszCharSet

String containing characters for matching.

Return Value

The zero-based index of the first character in this string that is also in *pszCharSet*; –1 if there is no match.

Remarks

Finds the first occurrence of any of the characters in *pszCharSet*.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString s( "abcdef" );
_ASSERT( s.FindOneOf( "xd" ) == 3 ); // 'd' is first match
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::Format

Writes formatted data to a **CStringT** in the same way that **sprintf** formats data into a C-style character array.

```
void __cdecl Format(
   UINT nFormatID,
   [, argument]...
);
void __cdecl Format(
   PCXSTR pszFormat,
   [, argument]...
);
```

Parameters

nFormatID

The string resource identifier that contains the format-control string.

pszFormat

A format-control string.

argument

Optional arguments.

Remarks

This function formats and stores a series of characters and values in the **CStringT**. Each optional argument (if any) is converted and output according to the corresponding format specification in *pszFormat* or from the string resource identified by *nFormatID*.

The call will fail if the string object itself is offered as a parameter to **Format**. For example, the following code will cause unpredictable results:

```
CAtlString str = "Some Data";
str.Format("%s%d", str, 123);
// Attention: str is also used in the parameter list.
```

For more information, see Format Specification Fields: printf and wprintf Functions and sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l.

Example

```
// cstringt_format.cpp
#include <string.h>
#include <atlstr.h>

int main(int argc, char* argv[])
{
    //typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;
    CAtlString str;

    str.Format(_T("Floating point: %.2f\n"), 12345.12345);
    _tprintf_s("%s", (LPCTSTR) str);

    str.Format(_T("Left-justified integer: %.6d\n"), 35);
    _tprintf_s("%s", (LPCTSTR) str);


    return 0;
}
```

Output

```
Floating point: 12345.12
Left-justified integer: 000035
```

See Also

Tasks

MantaWeb Sample: Online Personal Information Management

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::FormatMessage

Formats a message string.

```
void __cdecl FormatMessage(
   UINT nFormatID,
   [, argument]...
);
void __cdecl FormatMessage(
   PCXSTR pszFormat,
   [, argument]...
);
```

Parameters

nFormatID

The string resource identifier that contains the unformatted message text.

pszFormat

Points to the format-control string. It will be scanned for inserts and formatted accordingly. The format string is similar to run-time function **printf**-style format strings, except it allows for the parameters to be inserted in an arbitrary order.

argument

Optional arguments.

Remarks

The function requires a message definition as input. The message definition is determined by *pszFormat* or from the string resource identified by *nFormatID*. The function copies the formatted message text to the **CStringT** object, processing any embedded insert sequences if requested.

Note

FormatMessage attempts to allocate system memory for the newly formatted string. If this attempt fails, a memory exceptio n is automatically thrown.

Each insert must have a corresponding parameter following the *pszFormat* or *nFormatID* parameter. Within the message text, several escape sequences are supported for dynamically formatting the message. For more information, see the Windows FormatMessage function in the Platform SDK.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString str;
int nAsked = 5;
int nAgree = 4;

str.FormatMessage(_T("%1!d! of %2!d! writers agree: Soccer is %3%!"),
   nAgree, nAsked, _T("Best"));
_ASSERT(str == _T("4 of 5 writers agree: Soccer is Best!"));
```

See Also

Reference

CStringT Class

CStringT::FormatMessageV

Other Resources

CStringT Members

CStringT::FormatMessageV

Formats a message string using a variable argument list.

```
void FormatMessageV(
   PCXSTR pszFormat,
   va_list* pArgList
);
```

Parameters

pszFormat

Points to the format-control string. It will be scanned for inserts and formatted accordingly. The format string is similar to run-time function **printf**-style format strings, except it allows for the parameters to be inserted in an arbitrary order.

pArgList

Pointer to a list of arguments.

Remarks

The function requires a message definition as input, determined by *pszFormat*. The function copies the formatted message text and a variable list of arguments to the **CStringT** object, processing any embedded insert sequences if requested.

Note
FormatMessageV calls CStringT::FormatMessage, which attempts to allocate system memory for the newly formatted string. If this attempt fails, a memory exception is automatically thrown.

For more information, see the Windows FormatMessage function in the Platform SDK.

See Also

Reference

CStringT Class

CStringT::FormatMessage

printf, _printf_l, wprintf, _wprintf_l

Other Resources

CStringT Members

CStringT::FormatV

Formats a message string using a variable argument list.

```
void FormatV(
   PCXSTR pszFormat,
   va_list args
);
```

Parameters

pszFormat

Points to the format-control string. It will be scanned for inserts and formatted accordingly. The format string is similar to run-time function **printf**-style format strings, except it allows for the parameters to be inserted in an arbitrary order.

args

Pointer to a list of arguments.

Remarks

Writes a formatted string and a variable list of arguments to a **CStringT** string in the same way that **vsprintf** formats data into a C-style character array.

Example

```
// cstringt_formatv.cpp
#include <string.h>
#include <atlsimpstr.h>
#include <atlstr.h>

void WriteString(LPCSTR pstrFormat, ...)
{
    CString str;

    // format and write the data you were given
    va_list args;
    va_start(args, pstrFormat);

    str.FormatV(pstrFormat, args);
    va_end(args);

    printf_s(str);
    return;
}

int main( )
{
     WriteString("%d error(s) found in %d line(s)", 10, 1351);
}
```

Output

```
10 error(s) found in 1351 line(s)
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::GetEnvironmentVariable

Sets the string to the value of the specified environment variable.

```
BOOL GetEnvironmentVariable(
   PCXSTR pszVar
);
```

Parameters

pszVar

Pointer to a null-terminated string that specifies the environment variable.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Retrieves the value of the specified variable from the environment block of the calling process. The value is in the form of a null-terminated string of characters.

Example

```
// cstringt_getenvironmentvariable.cpp
#include <string.h>
#include <atlsimpstr.h>
#include <atlstr.h>

int main(int argc, char* argv[])
{
    //typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;
    CAtlString EnvStr;

    EnvStr.GetEnvironmentVariable("TEMP");
    printf_s("Current value of TEMP variable: %s\n", EnvStr);

   return 0;
}
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::Insert

Inserts a single character or a substring at the given index within the string.

```
int Insert(
   int iIndex,
   PCXSTR psz
);
int Insert(
   int iIndex,
   XCHAR ch
);
```

Parameters

iIndex

The index of the character before which the insertion will take place.

psz

A pointer to the substring to be inserted.

ch

The character to be inserted.

Return Value

The length of the changed string.

Remarks

The *iIndex* parameter identifies the first character that will be moved to make room for the character or substring. If *nIndex* is zero, the insertion will occur before the entire string. If *nIndex* is higher than the length of the string, the function will concatenate the present string and the new material provided by either *ch* or *psz*.

Example

```
// cstringt_insert.cpp
#include <stdio.h>
#include <atlstr.h>

int main()
{
    //typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

    CAtlString str("SoccerBest");
    int n = str.Insert(6, "is ");
    _ASSERT(n == str.GetLength());
    printf_s("1: %s\n", (LPCTSTR) str);

    n = str.Insert(6, ' ');
    _ASSERT(n == str.GetLength());
    printf_s("2: %s\n", (LPCTSTR) str);

    n = str.Insert(55, '!');
    _ASSERT(n == str.GetLength());
    printf_s("3: %s\n", (LPCTSTR) str);

   return 0;
}
```

Output

```
1: Socceris Best
2: Soccer is Best
```

```
3: Soccer is Best!
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::Left

Extracts the leftmost *nCount* characters from this **CStringT** object and returns a copy of the extracted substring.

```
CStringT Left(
   int nCount
) const;
```

Parameters

nCount

The number of characters to extract from this **CStringT** object.

Return Value

A **CStringT** object containing a copy of the specified range of characters. Note that the returned **CStringT** object may be empty.

Remarks

If *nCount* exceeds the string length, then the entire string is extracted. **Left** is similar to the Basic **Left** function.

For multibyte character sets (MBCS), *nCount* refers to each 8-bit character; that is, a lead and trail byte in one multibyte character are counted as two characters.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString s( _T("abcdef") );
_ASSERT( s.Left(2) == _T("ab") );
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::LoadString

Reads a Windows string resource, identified by *nID*, into an existing **CStringT** object.

```
BOOL LoadString(
   HINSTANCE hInstance,
   UINT nID,
   WORD wLanguageID
);
BOOL LoadString(
   HINSTANCE hInstance,
   UINT nID
);
BOOL LoadString(
   UINT nID
);
```

Parameters

hInstance

A handle to the instance of the module.

nID

A Windows string resource ID.

wLanguageID

The language of the string resource.

Return Value

Nonzero if resource load was successful; otherwise 0.

Remarks

Loads the string resource (*nID*) from the specified module (*hInstance*) using the specified language (*wLanguage*).

Example

```
#define IDS_FILENOTFOUND 1
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString s;
s.LoadString( IDS_FILENOTFOUND );
```

For another example, see ShowLocalized.

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::MakeLower

Converts the **CStringT** object to a lowercase string.

```
CStringT& MakeLower();
```

Return Value

The resulting lowercase string.

Remarks

Returns a copy of the string but in all lowercase characters.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString s( "ABC" );

_ASSERT( s.MakeLower() == "abc" );
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::MakeReverse

Reverses the order of the characters in the **CStringT** object.

```
CStringT& MakeReverse();
```

Return Value

The resulting reversed string.

Remarks

Returns a copy of the string but with all characters reversed.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString s( "abc" );

_ASSERT( s.MakeReverse() == "cba" );
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::MakeUpper

Converts the **CStringT** object to an uppercase string.

```
CStringT& MakeUpper();
```

Return Value

The resulting uppercase string.

Remarks

Returns a copy of the string but in all uppercase characters.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString s( "abc" );

_ASSERT( s.MakeUpper() == "ABC" );
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::Mid

Extracts a substring of length *nCount* characters from this **CStringT** object, starting at position *iFirst* (zero-based).

```
CStringT Mid(
   int iFirst,
   int nCount
) const;
CStringT Mid(
   int iFirst
) const;
```

Parameters

iFirst

The zero-based index of the first character in this **CStringT** object that is to be included in the extracted substring.

nCount

The number of characters to extract from this **CStringT** object. If this parameter is not supplied, then the remainder of the string is extracted.

Return Value

A **CStringT** object that contains a copy of the specified range of characters. Note that the returned **CStringT** object may be empty.

Remarks

The function returns a copy of the extracted substring. **Mid** is similar to the Basic Mid function (except that indexes in Basic are one-based).

For multibyte character sets (MBCS), *nCount* refers to each 8-bit character; that is, a lead and trail byte in one multibyte character are counted as two characters.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString s( _T("abcdef") );
_ASSERT( s.Mid( 2, 3 ) == _T("cde") );
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::OemToAnsi

Converts all the characters in this **CStringT** object from the OEM character set to the ANSI character set.

```
void OemToAnsi();
```

Remarks

This function is not available if **_UNICODE** is defined.

Example

See the example for CStringT::AnsiToOem.

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::Remove

Removes all instances of the specified character from the string.

```
int Remove(
   XCHAR chRemove
);
```

Parameters

chRemove

The character to be removed from a string.

Return Value

The count of characters removed from the string. Zero if the string is not changed.

Remarks

Comparisons for the character are case sensitive.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString str("This is a test.");
int n = str.Remove('t');
_ASSERT(n == 2);
_ASSERT(str == "This is a es.");
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::Replace

Replaces a character or substring with another.

```
int Replace(
   PCXSTR pszOld,
   PCXSTR pszNew
);
int Replace(
   XCHAR chOld,
   XCHAR chNew
);
```

Parameters

pszOld

A pointer to a string containing the character to be replaced by *pszNew*.

pszNew

A pointer to a string containing the character replacing *pszOld*.

chOld

The character to be replaced by *chNew*.

chNew

The character replacing *chOld*.

Return Value

The number of replaced instances of the character or substring. Zero if the string is not changed.

Remarks

This function replaces instances of the specified character or substring with instances of the new character or substring.

The string can grow or shrink as a result of the replacement; that is, *pszNew* and *pszOld* do not have to be equal in length. The function performs a case-sensitive match.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString strBang(_T("Everybody likes field hockey"));
int n = strBang.Replace(_T("field hockey"), _T("soccer"));
_ASSERT(n == 1);
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::ReverseFind

Searches this **CStringT** object for the last match of a character.

```
int ReverseFind(
   XCHAR ch
) const throw();
```

Parameters

ch

The character to search for.

Return Value

The zero-based index of the last character in this **CStringT** object that matches the requested character, or –1 if the character is not found.

Remarks

The function is similar to the run-time function **strrchr**.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString s( "abcabc" );
_ASSERT( s.ReverseFind( 'b' ) == 4 );
```

See Also

Reference

CStringT Class

strrchr, wcsrchr, _mbsrchr, _mbsrchr_l

Other Resources

CStringT Members

CStringT::Right

Extracts the last (that is, rightmost) *nCount* characters from this **CStringT** object and returns a copy of the extracted substring.

```
CStringT Right(
   int nCount
) const;
```

Parameters

nCount

The number of characters to extract from this **CStringT** object.

Return Value

A **CStringT** object that contains a copy of the specified range of characters. Note that the returned **CStringT** object can be empty.

Remarks

If *nCount* exceeds the string length, then the entire string is extracted. **Right** is similar to the Basic **Right** function (except that indexes in Basic are zero-based).

For multibyte character sets (MBCS), *nCount* refers to each 8-bit character; that is, a lead and trail byte in one multibyte character are counted as two characters.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString s( _T("abcdef") );
_ASSERT( s.Right(2) == _T("ef") );
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::SetSysString

Reallocates the **BSTR** pointed to by *pbstr* and copies the contents of the **CStringT** object into it, including the **NULL** character.

```
BSTR SetSysString(
   BSTR* pbstr
) const;
```

Parameters

pbstr

A pointer to a character string.

Return Value

The new string.

Remarks

Depending on the contents of the **CStringT** object, the value of the **BSTR** referenced by *pbstr* can change. The function throws a **CMemoryException** if insufficient memory exists.

This function is normally used to change the value of strings passed by reference for Automation.

For more information about OLE reallocation functions in Windows, see SysReAllocStringLen and SysFreeString.

Example

```
BSTR bstr = ::SysAllocString(L"Golf is fun!");

// create a CAtlString and change the OLE
// string to the contents of the BSTR
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString str("Soccer is best!");
BSTR bstr2 = str.SetSysString(&bstr);

// Now, both bstr and bstr2 reference a single instance of
// the "Soccer" string. The "Golf" string has been freed.
_ASSERT(bstr2 == bstr);
```

See Also

Reference

CStringT Class

CMemoryException Class

Other Resources

CStringT Members

CStringT::SpanExcluding

Extracts characters from the string, starting with the first character, that are not in the set of characters identified by *pszCharSet*.

```
CStringT SpanExcluding(
   PCXSTR pszCharSet
) const;
```

Parameters

pszCharSet

A string interpreted as a set of characters.

Return Value

A substring that contains characters in the string that are not in *pszCharSet*, beginning with the first character in the string and ending with the first character found in the string that is also in *pszCharSet* (that is, starting with the first character in the string and up to but excluding the first character in the string that is found *pszCharSet*). It returns the entire string if no character in *pszCharSet* is found in the string.

Remarks

SpanExcluding extracts and returns all characters preceding the first occurrence of a character from *pszCharSet* (in other words, the character from *pszCharSet* and all characters following it in the string, are not returned). If no character from *pszCharSet* is found in the string, then **SpanExcluding** returns the entire string.

Example

```
// cstringt_spanexcluding.cpp
#include <string.h>
#include <atlsimpstr.h>
#include <atlstr.h>

int main(int argc, char* argv[])
{
    // The string can be delimited by a semicolon( ; ),
    //  a comma( , ), a period( . ), a dash( - ),
    // or an apostrophe( ' ).
    //typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

    CAtlString src("World Cup '98");

    printf_s("%s",src.SpanExcluding( ";,.-'"));


   return 0;
}
```

Output

```
World Cup
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::SpanIncluding

Extracts characters from the string, starting with the first character, that are in the set of characters identified by *pszCharSet*.

```
CStringT SpanIncluding(
   PCXSTR pszCharSet
) const;
```

Parameters

pszCharSet

A string interpreted as a set of characters.

Return Value

A substring that contains characters in the string that are in *pszCharSet*, beginning with the first character in the string and ending when a character is found in the string that is not in *pszCharSet*. **SpanIncluding** returns an empty substring if the first character in the string is not in the specified set.

Remarks

If the first character of the string is not in the character set, then **SpanIncluding** returns an empty string. Otherwise, it returns a sequence of consecutive characters that are in the set.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString str( "cabbage" );
CAtlString res = str.SpanIncluding( "abc" );
_ASSERT( res == "cabba" );
res = str.SpanIncluding( "xyz" );
_ASSERT( res.IsEmpty( ) );
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::Tokenize

Finds the next token in a target string

```
CStringT Tokenize(
   PCXSTR pszTokens,
   int& iStart
) const;
```

Parameters

pszTokens

A string containing token delimiters. The order of these delimiters is not important.

iStart

The zero-based index to begin the search.

Return Value

A **CStringT** object containing the current token value.

Remarks

The `Tokenize` function finds the next token in the target string. The set of characters in *pszTokens* specifies possible delimiters of the token to be found. On each call to `Tokenize` the function starts at *iStart*, skips leading delimiters, and returns a `CStringT` object containing the current token, which is the string of characters up to the next delimiter character. The value of *iStart* is updated to be the position following the ending delimiter character, or -1 if the end of the string was reached. More tokens can be broken out of the remainder of the target string by a series of calls to `Tokenize`, using *iStart* to keep track of where in the string the next token is to be read. When there are no more tokens the function will return an empty string and *iStart* will be set to -1.

Unlike the CRT tokenize functions like strtok_s, _strtok_s_l, wcstok_s, _wcstok_s_l, _mbstok_s, _mbstok_s_l, `Tokenize` does not modify the target string.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;
CAtlString str( "%First Second#Third" );
CAtlString resToken;
int curPos= 0;

resToken= str.Tokenize("% #",curPos);
while (resToken != "")
{
   printf("Resulting token: %s\n", resToken);
   resToken= str.Tokenize("% #",curPos);
};
```

Output

```
Resulting Token: First
Resulting Token: Second
Resulting Token: Third
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::Trim

Trims all leading and trailing whitespace from the string.

```
CStringT& Trim(
   XCHAR chTarget
);
CStringT& Trim(
   PCXSTR pszTargets
);
CStringT& Trim( );
```

Parameters

chTarget

The target character to be trimmed.

pszTargets

A pointer to a string containing the target characters to be trimmed. All leading and trailing occurrences of characters in *pszTarget* will be trimmed from the **CStringT** object.

Return Value

Returns the trimmed string.

Remarks

Removes all leading and trailing occurrences of one of the following:

- The character specified by *chTarget*.
- All characters found in the string specified by *pszTargets*.
- Whitespace.

Example

```
// cstringt_trim.cpp
#include <atlstr.h>
#include <stdio.h>

int main( int argc, char* argv[] )
{
    //typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

    CAtlString str;
    str = _T("******Soccer is best, but liquor is quicker!?!?!?!?!");

    _tprintf_s(_T("Before: \"%s\"\n"), (LPCTSTR) str);
    _tprintf_s(_T("After : \"%s\"\n"), (LPCTSTR) str.Trim(_T("?!*")));

   return 0;
}
```

Output

```
Before: "******Soccer is best, but liquor is quicker!?!?!?!?!"
After : "Soccer is best, but liquor is quicker"
```

See Also

Reference

CStringT Class
CStringT::TrimLeft
CStringT::TrimRight

Other Resources

CStringT::TrimLeft

Trims leading characters from the string.

```
CStringT& TrimLeft(
   XCHAR chTarget
);
CStringT& TrimLeft(
   PCXSTR pszTargets
);
CStringT& TrimLeft( );
```

Parameters

chTarget

The target character to be trimmed.

pszTargets

A pointer to a string containing the target characters to be trimmed. All leading occurrences of characters in *pszTarget* will be trimmed from the **CStringT** object.

Return Value

The resulting trimmed string.

Example

```
// cstringt_trimleft.cpp
#include <atlstr.h>
#include <stdio.h>

int main( int argc, char* argv[] )
{
    //typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

    CAtlString str;
    str = _T("\t\t   ****Soccer is best!");

    _tprintf_s(_T("Before: \"%s\"\n"), (LPCTSTR) str);
    _tprintf_s(_T("After : \"%s\"\n"),
               (LPCTSTR) str.TrimLeft(_T("\t *")));

   return 0;
}
```

Output

```
Before: "                   ****Soccer is best!"
After : "Soccer is best!"
```

See Also

Reference

CStringT Class
CStringT::Trim
CStringT::TrimRight

Other Resources

CStringT Members

CStringT::TrimRight

Trims trailing characters from the string.

```
CStringT& TrimRight(
   XCHAR chTarget
);
CStringT& TrimRight(
   PCXSTR pszTargets
);
CStringT& TrimRight( );
```

Parameters

chTarget

The target character to be trimmed.

pszTargets

A pointer to a string containing the target characters to be trimmed. All trailing occurrences of characters in *pszTarget* will be trimmed from the **CStringT** object.

Return Value

The resulting trimmed string.

Example

```
// cstringt_trimright.cpp
#include <atlstr.h>
#include <stdio.h>

int main( int argc, char* argv[] )
{
    //typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

    CAtlString str;
    str = _T("Soccer is best, but liquor is quicker!?!?!?!?!");

    _tprintf_s(_T("Before: \"%s\"\n"), (LPCTSTR) str);
    _tprintf_s(_T("After : \"%s\"\n"),
                            (LPCTSTR) str.TrimRight(_T("?!")));

   return 0;
}
```

Output

```
Before: "Soccer is best, but liquor is quicker!?!?!?!?!"
After : "Soccer is best, but liquor is quicker"
```

See Also

Reference

CStringT Class
CStringT::Trim
CStringT::TrimLeft

Other Resources

CStringT Members

CStringT Operators

For information about the operators in **CStringT**, see CStringT Members.

CStringT::operator =

The **CStringT** assignment (**=**) operator reinitializes an existing **CStringT** object with new data.

```
CString& operator=(
   const VARIANT& var
);
CString& operator=(
   wchar_t ch
);
CString& operator=(
   char ch
);
CString& operator=(
   const unsigned char* pszSrc
);
CString& operator=(
   PCYSTR pszSrc
);
CString& operator=(
   PCXSTR pszSrc
);
template <bool bMFCDLL>
CString& operator=(
   const CSimpleStringT<BaseType, bMFCDLL>& strSrc
);
CString& operator=(
   const CStringT& strSrc
);
```

Parameters

var

A variant-type object to be copied into this **CStringT** object.

ch

A single character.

strSrc

A null-terminated string to be copied into this **CStringT** object.

pszSrc

A **CStringT** to be copied into this **CStringT** object.

Remarks

If the destination string (that is, the left side) is already large enough to store the new data, no new memory allocation is performed. You should be aware that memory exceptions can occur whenever you use the assignment operator because new storage is often allocated to hold the resulting **CStringT** object.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString s1, s2;        // Empty CAtlString objects

s1 = "cat";            // s1 = "cat"
s2 = s1;               // s1 and s2 each = "cat"
s1 = "the " + s1;      // Or expressions
s1 = 'x';              // Or just individual characters
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::operator +

Concatenates two strings or a character and a string.

```
friend CStringT operator+(
   const CStringT& str1,
   const CStringT& str2
);
friend CStringT operator+(
   const CStringT& str1,
   PCXSTR psz2
);
friend CStringT operator+(
   PCXSTR psz1
   const CStringT& str2,
);
friend CStringT operator+(
   char ch1
   const CStringT& str2,
);
friend CStringT operator+(
   const CStringT& str1,
   char ch2
);
friend CStringT operator+(
   const CStringT& str1,
   wchar_t ch2
);
friend CStringT operator+(
   wchar_t ch1
   const CStringT& str2,
);
```

Parameters

ch1

An ANSI or Unicode character to concatenate with a string.

ch2

An ANSI or Unicode character to concatenate with a string.

str1

A **CStringT** to concatenate with a string or character.

str2

A **CStringT** to concatenate with a string or character.

psz1

A pointer to a null-terminated string to concatenate with a string or character.

psz2

A pointer to a string to concatenate with a string or character.

Remarks

There are seven overload forms of the **CStringT::operator+** function. The first version concatenates two existing **CStringT** objects. The next two concatenate a **CStringT** object and a null-terminated string. The next two concatenate a **CStringT** object and an ANSI character. The last two concatenate a **CStringT** object and a Unicode character.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;
```

```
CAtlString s1("dog "), s2(" awake"), s3;  // Empty CAtlString objects

s1= "The " + s1;
s3= s1 + 'i';
s3= s3 + 's';
s3= s3 + s2;
_ASSERT(s3 == "The dog is awake");
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::operator +=

Concatenates characters to the end of the string.

```
CStringT& operator+=(
   const CThisSimpleString& str
);
template<
   bool bMFCDLL
>
CStringT& operator+=(
   const const CSimpleStringT<BaseType, bMFCDLL>& str
);
template<
   int t_nSize
>
CStringT& operator+=(
   const CStaticString< XCHAR, t_nSize >& strSrc
);
CStringT& operator+=(
   PCXSTR pszSrc
);
CStringT& operator+=(
   PCYSTR pszSrc
);
CStringT& operator+=(
   char ch
);
CStringT& operator+=(
   unsigned char ch
);
CStringT& operator+=(
   wchar_t ch
);
CStringT& operator+=(
   const VARIANT& var
);
```

Parameters

str

A reference to a **CThisSimpleString** object.

bMFCDLL

A boolean specifying whether the project is an MFC DLL or not.

BaseType

The string base type.

var

A variant object to concatenate to this string.

ch

An ANSI or Unicode character to concatenate with a string.

pszSrc

A pointer to the original string being concatenated.

strSrc

A **CStringT** to concatenate to this string.

Remarks

The operator accepts another **CStringT** object, a character pointer, or a single character. You should be aware that memory exceptions can occur whenever you use this concatenation operator because new storage can be allocated for characters added to this **CStringT** object.

For information on **CThisSimpleString**, see the Remarks section of CStringT::CStringT.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;

CAtlString s( "abc" );
_ASSERT( ( s += "def" ) == "abcdef" );
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::operator ==

Determines whether two strings are logically equal.

```
friend bool operator==(
   const CStringT& str1,
   const CStringT& str2
) throw();
friend bool operator==(
   const CStringT& str1
   PCXSTR psz2
) throw();
friend bool operator==(
   const CStringT& str1,
   PCYSTR psz2
) throw();
friend bool operator==(
   const CStringT& str1,
   XCHAR ch2
) throw();
friend bool operator==(
   PCXSTR psz1
   const CStringT& str2
) throw();
friend bool operator==(
   PCYSTR psz1
   const CStringT& str2,
) throw();
friend bool operator==(
   XCHAR ch1
   const CStringT& str2,
) throw();
```

Parameters

ch1

An ANSI or Unicode character for comparison.

ch2

An ANSI or Unicode character for comparison.

str1

A **CStringT** for comparison.

str2

A **CStringT** for comparison.

psz1

A pointer to a null-terminated string for comparison.

psz2

A pointer to a null-terminated string for comparison.

Remarks

Tests whether a string or character on the left side is equal to a string or character on the right side, and returns TRUE or FALSE accordingly.

Example

```
// typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;
CAtlString s1("dog"), s2("f"), s3("dog");
```

```
_ASSERT(s1 == "dog");
_ASSERT(s2 == 'f');
_ASSERT(s1 == s3);
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::operator !=

Determines whether two strings are logically not equal.

```
friend bool operator!=(
   const CStringT& str1,
   const CStringT& str2
) throw();
friend bool operator!=(
   const CStringT& str1
   PCXSTR psz2
) throw();
friend bool operator!=(
   const CStringT& str1,
   PCYSTR psz2
) throw();
friend bool operator!=(
   const CStringT& str1,
   XCHAR ch2
) throw();
friend bool operator!=(
   PCXSTR psz1
   const CStringT& str2
) throw();
friend bool operator!=(
   PCYSTR psz1
   const CStringT& str2,
) throw();
friend bool operator!=(
   XCHAR ch1
   const CStringT& str2,
) throw();
```

Parameters

ch1

An ANSI or Unicode character to concatenate with a string.

ch2

An ANSI or Unicode character to concatenate with a string.

str1

A **CStringT** for comparison.

str2

A **CStringT** for comparison.

psz1

A pointer to a null-terminated string for comparison.

psz2

A pointer to a null-terminated string for comparison.

Remarks

Tests if a string or character on the left side is not equal to a string or character on the right side.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;
CAtlString s1("cat"), s2("f"), s3("horse");

_ASSERT(s1 != "dog");
```

```
_ASSERT(s2 != 't');
_ASSERT(s1 != s2);
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::operator <

Determines whether the string on the left side of the operator is less than the string on the right side.

```
friend bool operator<(
   const CStringT& str1,
   const CStringT& str2
) throw();
friend bool operator<(
   const CStringT& str1
   PCXSTR psz2
) throw();
friend bool operator<(
   PCXSTR psz1
   const CStringT& str2
) throw();
```

Parameters

str1

A **CStringT** for comparison.

str2

A **CStringT** for comparison.

psz1

A pointer to a null-terminated string for comparison.

psz2

A pointer to a null-terminated string for comparison.

Remarks

A lexicographical comparison between strings, character by character until:

- It finds two corresponding characters unequal, and the result of their comparison is taken as the result of the comparison between the strings.
- It finds no inequalities, but one string has more characters than the other, and the shorter string is considered less than the longer string.
- It finds no inequalities and finds that the strings have the same number of characters, and so the strings are equal.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;
CAtlString s1("cat"), s2("cats"), s3("dogs");

_ASSERT(s1 < "dog");
_ASSERT(s1 < "cats");
_ASSERT(s2 < "cats and dogs");
_ASSERT(s2 < s3);
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::operator >

Determines whether the string on the left side of the operator is greater than the string on the right side.

```
friend bool operator>(
   const CStringT& str1,
   const CStringT& str2
) throw();
friend bool operator>(
   const CStringT& str1
   PCXSTR psz2
) throw();
friend bool operator>(
   PCXSTR psz1
   const CStringT& str2
) throw();
```

Parameters

str1

A **CStringT** for comparison.

str2

A **CStringT** for comparison.

psz1

A pointer to a null-terminated string for comparison.

psz2

A pointer to a null-terminated string for comparison.

Remarks

A lexicographical comparison between strings, character by character until:

- It finds two corresponding characters unequal, and the result of their comparison is taken as the result of the comparison between the strings.
- It finds no inequalities, but one string has more characters than the other, and the shorter string is considered less than the longer string.
- It finds no inequalities and finds that the strings have the same number of characters, so the strings are equal.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;
CAtlString s1("cat"), s2("cats"), s3("dogs");
_ASSERT("dog" > s1);
_ASSERT("cats" > s1);
_ASSERT("cats and dogs" > s2);
_ASSERT(s3 > s2);
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::operator <=

Determines whether the string on the left side of the operator is less than or equal to the string on the right side.

```
friend bool operator<=(
   const CStringT& str1,
   const CStringT& str2
) throw();
friend bool operator<=(
   const CStringT& str1
   PCXSTR psz2
) throw();
friend bool operator<=(
   PCXSTR psz1
   const CStringT& str2
) throw();
```

Parameters

str1

A **CStringT** for comparison.

str2

A **CStringT** for comparison.

psz1

A pointer to a null-terminated string for comparison.

psz2

A pointer to a null-terminated string for comparison.

Remarks

A lexicographical comparison between strings, character by character until:

- It finds two corresponding characters unequal, and the result of their comparison is taken as the result of the comparison between the strings.
- It finds no inequalities, but one string has more characters than the other, and the shorter string is considered less than the longer string.
- It finds no inequalities and finds that the strings have the same number of characters, so the strings are equal.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;
CAtlString s1("cat"), s2("cats"), s3("dogs");

_ASSERT(s1 <= "dog");
_ASSERT(s1 <= "cat");
_ASSERT(s3 <= "dogs and cats");
_ASSERT(s2 <= s3);
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CStringT::operator >=

Determines whether the string on the left side of the operator is greater than or equal to the string on the right side.

```
friend bool operator>=(
   const CStringT& str1,
   const CStringT& str2
) throw();
friend bool operator>=(
   const CStringT& str1
   PCXSTR psz2
) throw();
friend bool operator>=(
   PCXSTR psz1
   const CStringT& str2
) throw();
```

Parameters

str1

A **CStringT** for comparison.

str2

A **CStringT** for comparison.

psz1

A pointer to a string for comparison.

psz2

A pointer to a string for comparison.

Remarks

A lexicographical comparison between strings, character by character until:

- It finds two corresponding characters unequal, and the result of their comparison is taken as the result of the comparison between the strings.
- It finds no inequalities, but one string has more characters than the other, and the shorter string is considered less than the longer string.
- It finds no inequalities and finds that the strings have the same number of characters, so the strings are equal.

Example

```
//typedef CStringT< TCHAR, StrTraitATL< TCHAR > > CAtlString;
CAtlString s1("cat"), s2("cats"), s3("dogs");

_ASSERT("dog" >= s1);
_ASSERT("cats and dogs" >= s2);
_ASSERT(s3 >= s2);
```

See Also

Reference

CStringT Class

Other Resources

CStringT Members

CTime Class

Represents an absolute time and date.

```
class CTime
```

Remarks

CTime does not have a base class.

CTime values are based on coordinated universal time (UTC), which is equivalent to Greenwich mean time (GMT). The local time zone is controlled by the **TZ** environment variable.

When creating a **CTime** object, set the *nDST* parameter to 0 to indicate that standard time is in effect, or to a value greater than 0 to indicate that daylight savings time is in effect, or to a value less than zero to have the C run-time library code compute whether standard time or daylight savings time is in effect. **tm_isdst** is a required field. If not set, its value is undefined and the return value from mktime is unpredictable. If *timeptr* points to a tm structure returned by a previous call to **asctime**, **gmtime**, or **localtime**, the **tm_isdst** field contains the correct value.

A companion class, CTimeSpan, represents a time interval.

The **CTime** and **CTimeSpan** classes are not designed for derivation. Because there are no virtual functions, the size of **CTime** and **CTimeSpan** objects is exactly 8 bytes. Most member functions are inline.

Note
The upper date limit is 12/31/3000. The lower limit is 1/1/1970 12:00:00 AM GMT.

For more information on using **CTime**, see the articles Date and Time, and Time Management in the *Run-Time Library Reference*.

Requirements

Header: atltime.h

See Also

Reference

asctime, _wasctime
_ftime, _ftime32, _ftime64
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
strftime, wcsftime, _strftime_l, _wcsftime_l
time, _time32, _time64
Hierarchy Chart

Other Resources

CTime Members
ATL/MFC Shared Classes

CTime Members

Construction

CTime	Constructs **CTime** objects in various ways.
GetCurrentTime	Creates a **CTime** object that represents the current time (static member function).

Extraction

GetDay	Returns the day represent by the **CTime** object.
GetDayOfWeek	Returns the day of the week represented by the **CTime** object.
GetHour	Returns the hour represented by the **CTime** object.
GetMinute	Returns the minute represented by the **CTime** object.
GetMonth	Returns the month represented by the **CTime** object.
GetSecond	Returns the second represented by the **CTime** object.
GetTime	Returns a **__time64_t** value for the given **CTime** object.
GetYear	Returns the year represented by the **CTime** object.

Conversion

Format	Converts a **CTime** object into a formatted string — based on the local time zone.
FormatGmt	Converts a **CTime** object into a formatted string — based on UTC.
GetAsDBTIMESTAMP	Converts the time information stored in the **CTime** object to a Win32-compatible DBTIMESTAMP structure.
GetAsSystemTime	Converts the time information stored in the **CTime** object to a Win32-compatible SYSTEMTIME structure.
GetGmtTm	Breaks down a **CTime** object into components — based on UTC.
GetLocalTm	Breaks down a **CTime** object into components — based on the local time zone.

Operators

operator + −	These operators add and subtract **CTimeSpan** and **CTime** objects.
operator +=, −=	These operators add and subtract a **CTimeSpan** object to and from this **CTime** object.
operator =	The assignment operator.
operator ==, < , etc.	Comparison operators.

Archive

Serialize64	Serializes data to or from an archive.

See Also

Reference

CTime Class

Hierarchy Chart

CTime Member Functions

For information about the member functions in **CTime**, see CTime Members.

CTime::CTime

Creates a new **CTime** object initialized with the specified time.

```
CTime( ) throw( );
CTime(
   __time64_t time
) throw( );
CTime(
   int nYear,
   int nMonth,
   int nDay,
   int nHour,
   int nMin,
   int nSec,
      int nDST = -1
);
CTime(
   WORD wDosDate,
   WORD wDosTime,
   int nDST = -1
);
CTime(
   const SYSTEMTIME& st,
   int nDST = - 1
) throw( );
CTime(
   const FILETIME& ft,
   int nDST = - 1
);
CTime(
   const DBTIMESTAMP& dbts,
   int nDST = -1
) throw( );
```

Parameters

timeSrc

Indicates a **CTime** object that already exists.

time

A **__time64_t** time value, which is the number of seconds after January 1, 1970 UTC. Note that this will be adjusted to your local time. For example, if you are in New York and create a **CTime** object by passing a parameter of 0, CTime::GetMonth will return 12.

In Visual C++ versions 6.0 and earlier, *time* was a value of **time_t**. Visual C++ .NET and later converts a **time_t** parameter to **__time64_t**.

nYear, *nMonth*, *nDay*, *nHour*, *nMin*, *nSec*

Indicates the date and time values to be copied into the new **CTime** object.

nDST

Indicates whether daylight savings time is in effect. Can have one of three values:

- *nDST* set to 0 Standard time is in effect.
- *nDST* set to a value greater than 0 Daylight savings time is in effect.
- *nDST* set to a value less than 0 The default. Automatically computes whether standard time or daylight savings time is in effect.

wDosDate, *wDosTime*

MS-DOS date and time values to be converted to a date/time value and copied into the new **CTime** object.

st

A SYSTEMTIME structure to be converted to a date/time value and copied into the new **CTime** object.

ft

A FILETIME structure to be converted to a date/time value and copied into the new **CTime** object.

dbts

A reference to a DBTIMESTAMP structure containing the current local time.

Remarks

Each constructor is described below:

- **CTime( );** Constructs an uninitialized **CTime** object. This constructor allows you to define **CTime** object arrays. You should initialize such arrays with valid times before using.
- **CTime(const CTime&);** Constructs a **CTime** object from another **CTime** value.
- **CTime(__time64_t);** Constructs a **CTime** object from a **__time64_t** type. This constructor expects a UTC time and converts the result to a local time before storing the result.
- **CTime(int, int, ...);** Constructs a **CTime** object from local time components with each component constrained to the following ranges:

Component	Range
nYear	1970–3000
nMonth	1–12
nDay	1–31
nHour	0-23
nMin	0-59
nSec	0-59

  This constructor makes the appropriate conversion to UTC. The Debug version of the Microsoft Foundation Class Library asserts if one or more of the time components are out of range. You must validate the arguments before calling. This constructor expects a local time.
- **CTime**(**WORD, WORD**); Constructs a **CTime** object from the specified MS-DOS date and time values. This constructor expects a local time.
- **CTime**(**const SYSTEMTIME&**); Constructs a **CTime** object from a **SYSTEMTIME** structure. This constructor expects a local time.
- **CTime**(**const FILETIME&**); Constructs a **CTime** object from a **FILETIME** structure. You most likely will not use **CTime FILETIME** initialization directly. If you use a **CFile** object to manipulate a file, **CFile::GetStatus** retrieves the file time stamp for you through a **CTime** object initialized with a **FILETIME** structure. This constructor assumes a time based on UTC and automatically converts the value to local time before storing the result.

Note
The constructor using **DBTIMESTAMP** parameter is only available when OLEDB.h is included.

For more information, see the SYSTEMTIME and FILETIME structure in the *Platform SDK*. Also see the MS-DOS Date and Time entry in the *Platform SDK*.

Example

```
// Example for CTime::CTime
time_t osBinaryTime;  // C run-time time (defined in <time.h>)
time( &osBinaryTime ) ;  // Get the current time from the
                         // operating system.
CTime time1; // Empty CTime. (0 is illegal time value.)
CTime time2 = time1; // Copy constructor.
CTime time3( osBinaryTime );  // CTime from C run-time time
CTime time4( 1999, 3, 19, 22, 15, 0 ); // 10:15PM March 19, 1999
```

See Also

Reference

CTime Class

Hierarchy Chart

CTime::GetTime

CTime::GetCurrentTime

CTime::operator =

Other Resources

CTime Members

CTime::Format

Call this member function to create a formatted representation of the date/time value.

```
CString Format(
   LPCTSTR pszFormat
) const;
CString Format(
   UINT nFormatID
) const;
```

Parameters

pszFormat

A formatting string similar to the **printf** formatting string. Formatting codes, preceded by a percent (**%**) sign, are replaced by the corresponding **CTime** component. Other characters in the formatting string are copied unchanged to the returned string. See the run-time function strftime for a list of formatting codes.

nFormatID

The ID of the string that identifies this format.

Return Value

A CString that contains the formatted time.

Remarks

If the status of this **CTime** object is null, the return value is an empty string.

Example

```
// Example for CTime::Format and CTime::FormatGmt
CTime t( 1999, 3, 19, 22, 15, 0 );
// 10:15 PM March 19, 1999
CString s = t.Format( "%A, %B %d, %Y" );
ATLASSERT( s == "Friday, March 19, 1999" );
```

See Also

Reference

CTime Class

Hierarchy Chart

CTime::FormatGmt

Other Resources

CTime Members

CTime::FormatGmt

Generates a formatted string that corresponds to this **CTime** object.

```
CString FormatGmt(
   LPCTSTR pszFormat
) const;
CString FormatGmt(
   UINT nFormatID
) const;
```

Parameters

pszFormat

Specifies a formatting string similar to the **printf** formatting string. See the run-time function strftime for details.

nFormatID

The ID of the string that identifies this format.

Return Value

A CString that contains the formatted time.

Remarks

The time value is not converted and thus reflects UTC.

Example

See the example for CTime::Format.

See Also

Reference

CTime Class
Hierarchy Chart
CTime::Format

Other Resources

CTime Members

CTime::GetAsDBTIMESTAMP

Note
This method is only available when OLEDB.h is included.

Call this member function to convert the time information stored in the **CTime** object to a Win32–compatible DBTIMESTAMP structure.

```
bool GetAsDBTIMESTAMP(
   DBTIMESTAMP& dbts
) const throw( );
```

Parameters

dbts

A reference to a DBTIMESTAMP structure containing the current local time.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Stores the resulting time in the referenced *dbts* structure. The **DBTIMESTAMP** data structure initialized by this function will have its **fraction** member set to zero.

Example

```
#include <oledb.h>
int main( )
{
   CTime t = CTime::GetCurrentTime();
   DBTIMESTAMP ts;
   t.GetAsDBTIMESTAMP( ts ); // Retrieves the time in t into the ts structure
}
```

See Also

Reference

CTime Class

Hierarchy Chart

Other Resources

CTime Members

CTime::GetAsSystemTime

Call this member function to convert the time information stored in the **CTime** object to a Win32–compatible SYSTEMTIME structure.

```
bool GetAsSystemTime(
   SYSTEMTIME& st
) const throw( );
```

Parameters

timeDest

A reference to a SYSTEMTIME structure that will hold the converted date/time value of the **CTime** object.

Return Value

True if successful; otherwise false.

Remarks

GetAsSystemTime stores the resulting time in the referenced *timeDest* structure. The **SYSTEMTIME** data structure initialized by this function will have its **wMilliseconds** member set to zero.

Example

```
// Convert CTime to FILETIME
CTime time(CTime::GetCurrentTime());
SYSTEMTIME timeDest;
time.GetAsSystemTime(timeDest);
FILETIME fileTime;
::SystemTimeToFileTime(&timeDest, &fileTime);
```

See Also

Reference

CTime Class

Hierarchy Chart

Other Resources

CTime Members

CTime::GetCurrentTime

Returns a **CTime** object that represents the current time.

```
static CTime WINAPI GetCurrentTime( ) throw( );
```

Remarks

Returns the current system date and time in Coordinated Universal Time (UTC).

Example

```
// Example for CTime::GetCurrentTime
CTime t = CTime::GetCurrentTime();
```

See Also

Reference

CTime Class

Hierarchy Chart

Other Resources

CTime Members

CTime::GetDay

Returns the day represent by the **CTime** object.

```
int GetDay( ) const throw( );
```

Return Value

Returns the day of the month, based on local time, in the range 1 through 31.

Remarks

This function calls **GetLocalTm**, which uses an internal, statically allocated buffer. The data in this buffer is overwritten because of calls to other **CTime** member functions.

Example

```
// Example for CTime::GetDay, CTime::GetMonth, and CTime::GetYear
CTime t( 1999, 3, 19, 22, 15, 0 ); // 10:15 PM March 19, 1999
ATLASSERT( t.GetDay() == 19 );
ATLASSERT( t.GetMonth() == 3 );
ATLASSERT( t.GetYear() == 1999 );
```

See Also

Reference

CTime Class

Hierarchy Chart

CTime::GetDayOfWeek

Other Resources

CTime Members

CTime::GetDayOfWeek

Returns the day of the week represented by the **CTime** object.

```
int GetDayOfWeek( ) const throw( );
```

Return Value

Returns the day of the week based on local time; 1 = Sunday, 2 = Monday, to 7 = Saturday.

Remarks

This function calls **GetLocalTm**, which uses an internal statically allocated buffer. The data in this buffer is overwritten because of calls to other **CTime** member functions.

Example

```
// Print out the day of the week using localized day name
UINT DayOfWeek[] = {
   LOCALE_SDAYNAME7,   // Sunday
   LOCALE_SDAYNAME1,
   LOCALE_SDAYNAME2,
   LOCALE_SDAYNAME3,
   LOCALE_SDAYNAME4,
   LOCALE_SDAYNAME5,
   LOCALE_SDAYNAME6   // Saturday
};
TCHAR strWeekday[256];
CTime time(CTime::GetCurrentTime());   // Initialize CTime with current time
::GetLocaleInfo(LOCALE_USER_DEFAULT,   // Get string for day of the week from system
   DayOfWeek[time.GetDayOfWeek()-1],   // Get day of week from CTime
   strWeekday, sizeof(strWeekday));
ATLTRACE("%s\n", strWeekday);               // Print out day of the week
```

See Also

Reference

CTime Class

Hierarchy Chart

Other Resources

CTime Members

CTime::GetGmtTm

Gets a **struct tm** that contains a decomposition of the time contained in this **CTime** object.

```
struct tm* GetGmtTm(
   struct tm* ptm
) const;
```

Parameters

ptm

Points to a buffer that will receive the time data. If this pointer is **NULL**, an exception is thrown.

Return Value

A pointer to a filled-in **struct tm** as defined in the include file TIME.H. See gmtime, _gmtime32, _gmtime64 for the structure layout.

Remarks

GetGmtTm returns UTC.

ptm cannot be **NULL**. If you want to revert to the old behavior, in which *ptm* could be **NULL** to indicate that an internal, statically allocated buffer should be used, then undefine **_SECURE_ATL**.

Example

```
// Compute difference between local time and GMT
   struct tm* osTime;   // A pointer to a structure containing
                        // time elements.
   CTime time(CTime::GetCurrentTime());
   tm t1 = *(time.GetLocalTm( osTime ));
   tm t2 = *(time.GetGmtTm( osTime ));

   ATLTRACE("Difference between local time and GMT is %d hours.\n",
          t1.tm_hour - t2.tm_hour);
```

See Also

Reference

CTime Class

Hierarchy Chart

Other Resources

CTime Members

CTime::GetHour

Returns the hour represented by the **CTime** object.

```
int GetHour( ) const throw( );
```

Return Value

Returns the hour, based on local time, in the range 0 through 23.

Remarks

This function calls **GetLocalTm**, which uses an internal statically allocated buffer. The data in this buffer is overwritten because of calls to other **CTime** member functions.

Example

```
// Example for CTime::GetHour, CTime::GetMinute, and CTime::GetSecond
CTime t( 1999, 3, 19, 22, 15, 0 ); // 10:15 PM March 19, 1999
ATLASSERT( t.GetSecond() == 0 );
ATLASSERT( t.GetMinute() == 15 );
ATLASSERT( t.GetHour() == 22 );
```

See Also

Reference

CTime Class

Hierarchy Chart

Other Resources

CTime Members

CTime::GetLocalTm

Gets a **struct tm** containing a decomposition of the time contained in this **CTime** object.

```
struct tm* GetLocalTm(
   struct tm* ptm
) const;
```

Parameters

ptm

Points to a buffer that will receive the time data. If this pointer is **NULL**, an exception is thrown.

Return Value

A pointer to a filled-in **struct tm** as defined in the include file TIME.H. See gmtime, _gmtime32, _gmtime64 for the structure layout.

Remarks

GetLocalTm returns local time.

ptm cannot be **NULL**. If you want to revert to the old behavior, in which *ptm* could be **NULL** to indicate that an internal, statically allocated buffer should be used, then undefine **_SECURE_ATL**.

Example

```
// Example for CTime::GetLocalTm
CTime t( 1999, 3, 19, 22, 15, 0 );   // 10:15PM March 19, 1999
struct tm* osTime;   // A pointer to a structure containing
                     // time elements.
osTime = t.GetLocalTm( osTime );
ATLASSERT( osTime->tm_mon == 2 ); // Note zero-based month!
```

See Also

Reference

CTime Class

Hierarchy Chart

Other Resources

CTime Members

CTime::GetMinute

Returns the minute represented by the **CTime** object.

```
int GetMinute( ) const throw( );
```

Return Value

Returns the minute, based on local time, in the range 0 through 59.

Remarks

This function calls **GetLocalTm**, which uses an internal statically allocated buffer. The data in this buffer is overwritten because of calls to other **CTime** member functions.

Example

See the example for GetHour.

See Also

Reference

CTime Class

Hierarchy Chart

Other Resources

CTime Members

CTime::GetMonth

Returns the month represented by the **CTime** object.

```
int GetMonth( ) const throw( );
```

Return Value

Returns the month, based on local time, in the range 1 through 12 (1 = January).

Remarks

This function calls **GetLocalTm**, which uses an internal statically allocated buffer. The data in this buffer is overwritten because of calls to other **CTime** member functions.

Example

See the example for GetDay.

See Also

Reference

CTime Class

Hierarchy Chart

Other Resources

CTime Members

CTime::GetSecond

Returns the second represented by the **CTime** object.

```
int GetSecond( ) const throw( );
```

Return Value

Returns the second, based on local time, in the range 0 through 59.

Remarks

This function calls **GetLocalTm**, which uses an internal statically allocated buffer. The data in this buffer is overwritten because of calls to other **CTime** member functions.

Example

See the example for GetHour.

See Also

Reference

CTime Class

Hierarchy Chart

Other Resources

CTime Members

CTime::GetTime

Returns a **__time64_t** value for the given **CTime** object.

```
__time64_t GetTime( ) const throw( );
```

Return Value

GetTime will return the number of seconds between the current **CTime** object and January 1, 1970.

Example

```
// ctime_gettime.cpp
#include <afx.h>

int main(int argc, char* argv[])
{
    // Example for CTime::GetTime
    CTime t( 2005, 10, 20, 23, 50, 0 ); // 11:50 PM October 20, 2005
    time_t osBinaryTime = t.GetTime();  // time_t defined in <time.h>

    printf_s( "time_t = %ld\n", osBinaryTime );

    return 0;
}
```

Output

```
time_t = 1129877400
```

See Also

Reference

CTime Class

Hierarchy Chart

CTime::CTime

Other Resources

CTime Members

CTime::GetYear

Returns the year represented by the **CTime** object.

```
int GetYear( ) const throw( );
```

Return Value

Returns the year, based on local time, in the range January 1,1970, to January 18, 2038 (inclusive).

Remarks

This function calls **GetLocalTm**, which uses an internal statically allocated buffer. The data in this buffer is overwritten because of calls to other **CTime** member functions.

Example

See the example for GetDay.

See Also

Reference

CTime Class
Hierarchy Chart
CTime::CTime

Other Resources

CTime Members

CTime::Serialize64

Note
This method is only available in MFC projects.

Serializes the data associated with the member variable to or from an archive.

```
CArchive& Serialize64(
   CArchive& ar
);
```

Parameters

ar

The **CArchive** object that you want to update.

Return Value

The updated **CArchive** object.

See Also

Reference

CTime Class

Hierarchy Chart

CArchive Class

Concepts

Serialization in MFC

Other Resources

CTime Members

CTime Operators

For information about the operators in **CTime**, see CTime Members.

CTime::operator =

The assignment operator.

```
CTime& operator =(
   __time64_t time
) throw( );
```

Parameters

time

The new date/time value.

Return Value

The updated **CTime** object.

Remarks

This overloaded assignment operator copies the source time into this **CTime** object. The internal time storage in a **CTime** object is independent of time zone. Time zone conversion is not necessary during assignment.

See Also

Reference

CTime Class
Hierarchy Chart
CTime::CTime

Other Resources

CTime Members

CTime::operator +, -

These operators add and subtract **CTimeSpan** and **CTime** objects.

```
CTime operator +(
   CTimeSpan timeSpan
) const throw( );
CTime operator -(
   CTimeSpan timeSpan
) const throw( );
CTimeSpan operator -(
   CTime time
) const throw( );
```

Parameters

timeSpan

The **CTimeSpan** object to be added or subtracted.

time

The **CTime** object to be subtracted.

Return Value

A **CTime** or **CTimeSpan** object representing the result of the operation.

Remarks

CTime objects represent absolute time, **CTimeSpan** objects represent relative time. The first two operators allow you to add and subtract **CTimeSpan** objects to and from **CTime** objects. The third operator allows you to subtract one **CTime** object from another to yield a **CTimeSpan** object.

Example

```
// Example for CTime::operator  +, -
CTime t1( 1999, 3, 19, 22, 15, 0 ); // 10:15 PM March 19, 1999
CTime t2( 1999, 3, 20, 22, 15, 0 ); // 10:15 PM March 20, 1999
CTimeSpan ts = t2 - t1;             // Subtract 2 CTimes
ATLASSERT( ts.GetTotalSeconds() == 86400L );
ATLASSERT( ( t1 + ts ) == t2 );       // Add a CTimeSpan to a CTime.
ATLASSERT( ( t2 - ts ) == t1 );       // Subtract a CTimeSpan from a CTime.
```

See Also

Reference

CTime Class

Hierarchy Chart

Other Resources

CTime Members

CTime::operator +=, -=

These operators add and subtract a **CTimeSpan** object to and from this **CTime** object.

```
CTime& operator +=(
   CTimeSpan span
) throw( );
CTime& operator -=(
   CTimeSpan span
) throw( );
```

Parameters

span

The **CTimeSpan** object to be added or subtracted.

Return Value

The updated **CTime** object.

Remarks

These operators allow you to add and subtract a **CTimeSpan** object to and from this **CTime** object.

Example

```
// Example for CTime::operator -=
CTime t( 1999, 3, 19, 22, 15, 0 ); // 10:15 PM March 19, 1999
t += CTimeSpan( 0, 1, 0, 0 );      // 1 hour exactly
ATLASSERT( t.GetHour() == 23 );
```

See Also

Reference

CTime Class

Hierarchy Chart

Other Resources

CTime Members

CTime Comparison Operators

Comparison operators.

```
bool operator ==(
   CTime time
) const throw( );
bool operator !=(
   CTime time
) const throw( );
bool operator <(
   CTime time
) const throw( );
bool operator >(
   CTime time
) const throw( );
bool operator <=(
   CTime time
) const throw( );
bool operator >=(
   CTime time
) const throw( );
```

Parameters

time

The **CTime** object to be compared.

Return Value

These operators compare two absolute times and return **true** if the condition is true; otherwise **false**.

Example

```
// Example for CTime comparison operators
CTime t1 = CTime::GetCurrentTime();
CTime t2 = t1 + CTimeSpan( 0, 1, 0, 0 );    // 1 hour later
ATLASSERT( t1 != t2 );
ATLASSERT( t1 < t2 );
ATLASSERT( t1 <= t2 );
```

See Also

Reference

CTime Class

Hierarchy Chart

Other Resources

CTime Members

CTimeSpan Class

An amount of time, which is internally stored as the number of seconds in the time span.

```
class CTimeSpan
```

Remarks

CTimeSpan does not have a base class.

CTimeSpan functions convert seconds to various combinations of days, hours, minutes, and seconds.

The **CTimeSpan** object is stored in a **__time64_t** structure, which is 8 bytes.

A companion class, CTime, represents an absolute time.

The **CTime** and **CTimeSpan** classes are not designed for derivation. Because there are no virtual functions, the size of both **CTime** and **CTimeSpan** objects is exactly 8 bytes. Most member functions are inline.

For more information on using **CTimeSpan**, see the articles Date and Time, and Time Management in the *Run-Time Library Reference*.

Requirements

Header: atltime.h

See Also

Reference

asctime, _wasctime
_ftime, _ftime32, _ftime64
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
strftime, wcsftime, _strftime_l, _wcsftime_l
time, _time32, _time64
Hierarchy Chart

Other Resources

CTimeSpan Members
ATL/MFC Shared Classes

CTimeSpan Members

Construction

CTimeSpan	Constructs **CTimeSpan** objects in various ways.

Extraction

GetDays	Returns a value that represents the number of complete days in this **CTimeSpan**.
GetHours	Returns a value that represents the number of hours in the current day (–23 through 23).
GetTotalHours	Returns a value that represents the total number of complete hours in this **CTimeSpan**.
GetMinutes	Returns a value that represents the number of minutes in the current hour (–59 through 59).
GetTotalMinutes	Returns a value that represents the total number of complete minutes in this **CTimeSpan**.
GetSeconds	Returns a value that represents the number of seconds in the current minute (–59 through 59).
GetTotalSeconds	Returns a value that represents the total number of complete seconds in this **CTimeSpan**.
GetTimeSpan	Returns the value of the **CTimeSpan** object.

Conversion

Format	Converts a **CTimeSpan** into a formatted string.

Operators

operator + –	Adds and subtracts **CTimeSpan** objects.
operator += –=	Adds and subtracts a **CTimeSpan** object to and from this **CTimeSpan**.
operator == < etc.	Compares two relative time values.

Archive/Dump

Serialize64	Serializes data to or from an archive.

See Also

Reference

CTimeSpan Class

Hierarchy Chart

CTimeSpan Member Functions

For information about the member functions in **CTimeSpan**, see CTimeSpan Members.

CTimeSpan::CTimeSpan

Constructs **CTimeSpan** objects in various ways.

```
CTimeSpan( ) throw( );
CTimeSpan(
   __time64_t time
) throw( );
CTimeSpan(
   LONG lDays,
   int nHours,
   int nMins,
   int nSecs
) throw( );
```

Parameters

timeSpanSrc

A **CTimeSpan** object that already exists.

time

A **__time64_t** time value, which is the number of seconds in the time span. In Visual C++ versions 6.0 and earlier, *time* was a value of **time_t**. Visual C++ .NET or later silently converts a **time_t** parameter to **__time64_t**.

lDays, *nHours*, *nMins*, *nSecs*

Days, hours, minutes, and seconds, respectively.

Remarks

All these constructors create a new **CTimeSpan** object initialized with the specified relative time. Each constructor is described below:

- **CTimeSpan( );** Constructs an uninitialized **CTimeSpan** object.
- **CTimeSpan(const CTimeSpan&);** Constructs a **CTimeSpan** object from another **CTimeSpan** value.
- **CTimeSpan(__time64_t);** Constructs a **CTimeSpan** object from a **__time64_t** type.
- **CTimeSpan(LONG, int, int, int);** Constructs a **CTimeSpan** object from components with each component constrained to the following ranges:

Component	Range
lDays	0–25,000 (approximately)
nHours	0–23
nMins	0–59
nSecs	0–59

Note that the Debug version of the Microsoft Foundation Class Library asserts if one or more of the time-day components is out of range. It is your responsibility to validate the arguments prior to calling.

Example

```
// example for CTimeSpan::CTimeSpan
CTimeSpan ts1;  // Uninitialized time value
CTimeSpan ts2a( ts1 ); // Copy constructor
CTimeSpan ts2b = ts1; // Copy constructor again
CTimeSpan ts3( 100 ); // 100 seconds
CTimeSpan ts4( 0, 1, 5, 12 );    // 1 hour, 5 minutes, and 12 seconds
```

See Also

Reference

CTimeSpan Class

Hierarchy Chart

Other Resources

CTimeSpan Members

CTimeSpan::Format

Generates a formatted string that corresponds to this **CTimeSpan**.

```
CString Format(
   LPCSTR pFormat
) const;
CString Format(
   LPCTSTR pszFormat
) const;
CString Format(
   UINT nID
) const;
```

Parameters

pFormat, *pszFormat*

A formatting string similar to the **printf** formatting string. Formatting codes, preceded by a percent (**%**) sign, are replaced by the corresponding **CTimeSpan** component. Other characters in the formatting string are copied unchanged to the returned string. The value and meaning of the formatting codes for **Format** are listed below:

- **%D** Total days in this **CTimeSpan**
- **%H** Hours in the current day
- **%M** Minutes in the current hour
- **%S** Seconds in the current minute
- **%%** Percent sign

nID

The ID of the string that identifies this format.

Return Value

A **CString** object that contains the formatted time.

Remarks

The Debug version of the library checks the formatting codes and asserts if the code is not in the list above.

Example

```
// example for CTimeSpan::Format
CTimeSpan ts( 3, 1, 5, 12 ); // 3 days, 1 hour, 5 min, and 12 sec
CString s = ts.Format( "Total days: %D, hours: %H, mins: %M, secs: %S" );
ATLASSERT( s == "Total days: 3, hours: 01, mins: 05, secs: 12" );
```

See Also

Reference

CTimeSpan Class

Hierarchy Chart

Other Resources

CTimeSpan Members

CTimeSpan::GetDays

Returns a value that represents the number of complete days in this **CTimeSpan**.

```
LONGLONG GetDays( ) const throw( );
```

Return Value

Returns the number of complete 24-hour days in the time span. This value may be negative if the time span is negative.

Remarks

Note that Daylight Savings Time can cause **GetDays** to return a potentially surprising result. For example, when DST is in effect, **GetDays** reports the number of days between April 1 and May 1 as 29, not 30, because one day in April is shortened by an hour and therefore does not count as a complete day.

Example

```
// example for CTimeSpan::GetDays
CTimeSpan ts( 3, 1, 5, 12 ); // 3 days, 1 hour, 5 min, and 12 sec
ATLASSERT( ts.GetDays() == 3 );
```

See Also

Reference

CTimeSpan Class

Hierarchy Chart

Other Resources

CTimeSpan Members

CTimeSpan::GetHours

Returns a value that represents the number of hours in the current day (–23 through 23).

```
LONG GetHours( ) const throw( );
```

Return Value

Returns the number of hours in the current day. The range is –23 through 23.

Example

```
// example for CTimeSpan::GetHours
CTimeSpan ts( 3, 1, 5, 12 ); // 3 days, 1 hour, 5 min, and 12 sec
ATLASSERT( ts.GetHours() == 1 );
ATLASSERT( ts.GetMinutes() == 5 );
ATLASSERT( ts.GetSeconds() == 12 );
```

See Also

Reference

CTimeSpan Class

Hierarchy Chart

Other Resources

CTimeSpan Members

CTimeSpan::GetMinutes

Returns a value that represents the number of minutes in the current hour (–59 through 59).

```
LONG GetMinutes( ) const throw( );
```

Return Value

Returns the number of minutes in the current hour. The range is –59 through 59.

Example

See the example for GetHours.

See Also

Reference

CTimeSpan Class

Hierarchy Chart

Other Resources

CTimeSpan Members

CTimeSpan::GetSeconds

Returns a value that represents the number of seconds in the current minute (–59 through 59).

```
LONG GetSeconds( ) const throw( );
```

Return Value

Returns the number of seconds in the current minute. The range is –59 through 59.

Example

See the example for GetHours.

See Also

Reference

CTimeSpan Class

Hierarchy Chart

Other Resources

CTimeSpan Members

CTimeSpan::GetTimeSpan

Returns the value of the **CTimeSpan** object.

```
__time64_t GetTimeSpan( ) const throw( );
```

Return Value

Returns the current value of the **CTimeSpan** object.

See Also

Reference

CTimeSpan Class

Hierarchy Chart

Other Resources

CTimeSpan Members

CTimeSpan::GetTotalHours

Returns a value that represents the total number of complete hours in this **CTimeSpan**.

```
LONGLONG GetTotalHours( ) const throw( );
```

Return Value

Returns the total number of complete hours in this **CTimeSpan**.

Example

```
// example for CTimeSpan::GetTotalHours
CTimeSpan ts( 3, 1, 5, 12 ); // 3 days, 1 hour, 5 min, and 12 sec
ATLASSERT( ts.GetTotalHours() == 73 );
ATLASSERT( ts.GetTotalMinutes() == 4385 );
ATLASSERT( ts.GetTotalSeconds() == 263112 );
```

See Also

Reference

CTimeSpan Class

Hierarchy Chart

Other Resources

CTimeSpan Members

CTimeSpan::GetTotalMinutes

Returns a value that represents the total number of complete minutes in this **CTimeSpan**.

```
LONGLONG GetTotalMinutes( ) const throw( );
```

Return Value

Returns the total number of complete minutes in this **CTimeSpan**.

Example

See the example for GetTotalHours.

See Also

Reference

CTimeSpan Class

Hierarchy Chart

Other Resources

CTimeSpan Members

CTimeSpan::GetTotalSeconds

Returns a value that represents the total number of complete seconds in this **CTimeSpan**.

```
LONGLONG GetTotalSeconds( ) const throw( );
```

Return Value

Returns the total number of complete seconds in this **CTimeSpan**.

Example

See the example for GetTotalHours.

See Also

Reference

CTimeSpan Class

Hierarchy Chart

Other Resources

CTimeSpan Members

CTimeSpan::Serialize64

Note
This method is only available in MFC projects.

Serializes the data associated with the member variable to or from an archive.

```
CArchive& Serialize64(
   CArchive& ar
);
```

Parameters

ar

The **CArchive** object that you want to update.

Return Value

The updated **CArchive** object.

See Also

Reference

CTimeSpan Class

Hierarchy Chart

CArchive Class

Concepts

Serialization in MFC

Other Resources

CTimeSpan Members

CTimeSpan Operators

For information about the operators in **CTimeSpan**, see CTimeSpan Members.

CTimeSpan::operator +, -

Adds and subtracts **CTimeSpan** objects.

```
CTimeSpan operator +(
   CTimeSpan span
) const throw( );
CTimeSpan operator -(
   CTimeSpan span
) const throw( );
```

Parameters

span

The value to add to the **CTimeSpan** object.

Return Value

A **CTimeSpan** object representing the result of the operation.

Remarks

These two operators allow you to add and subtract **CTimeSpan** objects to and from each other.

Example

```
// example for CTimeSpan::operator +, -
CTimeSpan ts1( 3, 1, 5, 12 ); // 3 days, 1 hour, 5 min, and 12 sec
CTimeSpan ts2( 100 ); // 100 seconds
CTimeSpan ts3 = ts1 + ts2;
ATLASSERT( ts3.GetSeconds() == 52 ); // 6 mins, 52 secs
```

See Also

Reference

CTimeSpan Class

Hierarchy Chart

Other Resources

CTimeSpan Members

CTimeSpan::operator +=, -=

Adds and subtracts a **CTimeSpan** object to and from this **CTimeSpan**.

```
CTimeSpan& operator +=(
   CTimeSpan span
) throw( );
CTimeSpan& operator -=(
   CTimeSpan span
) throw( );
```

Parameters

span

The value to add to the **CTimeSpan** object.

Return Value

The updated **CTimeSpan** object.

Remarks

These operators allow you to add and subtract a **CTimeSpan** object to and from this **CTimeSpan**.

Example

```
// example for CTimeSpan::operator +=, -=
CTimeSpan ts1( 10 ); // 10 seconds
CTimeSpan ts2( 100 ); // 100 seconds
ts2 -= ts1;
ATLASSERT( ts2.GetTotalSeconds() == 90 );
```

See Also

Reference

CTimeSpan Class

Hierarchy Chart

Other Resources

CTimeSpan Members

CTimeSpan Comparison Operators

Comparison operators.

```
bool operator ==(
   CTimeSpan span
) const throw( );
bool operator !=(
   CTimeSpan span
) const throw( );
bool operator <(
   CTimeSpan span
) const throw( );
bool operator >(
   CTimeSpan span
) const throw( );
bool operator <=(
   CTimeSpan span
) const throw( );
bool operator >=(
   CTimeSpan span
) const throw( );
```

Parameters

span

The object to compare.

Return Value

These operators compare two relative time values. They return **true** if the condition is true; otherwise **false**.

Example

```
// example for CTimeSpan comparison operators
CTimeSpan ts1( 100 );
CTimeSpan ts2( 110 );
ATLASSERT( ( ts1 != ts2 ) && ( ts1 < ts2 ) && ( ts1 <= ts2 ) );
```

See Also

Reference

CTimeSpan Class

Hierarchy Chart

Other Resources

CTimeSpan Members

IAtlStringMgr Class

This class represents the interface to a **CStringT** memory manager.

```
__interface IAtlStringMgr
```

Remarks

This interface manages the memory used by the MFC-independent string classes; such as CSimpleStringT, CStringT, and CFixedStringT.

You can also use this class to implement a custom memory manager for your custom string class. For more information, see Memory Management and CStringT.

Requirements

Header: atlsimpstr.h

See Also

Reference

Hierarchy Chart

Other Resources

IAtlStringMgr Members

ATL/MFC Shared Classes

IAtlStringMgr Members

Methods

Allocate	Call this method to allocate a new string data structure.
Clone	Call this method to return a pointer to a new string manager for use with another instance of **CSimpleStringT**.
Free	Call this method to free a string data structure.
GetNilString	Returns a pointer to the **CStringData** object used by empty string objects.
Reallocate	Call this method to reallocate a string data structure.

See Also

Reference

IAtlStringMgr Class

IAtlStringMgr Methods

For information about the methods in **IAtlStringMgr**, see IAtlStringMgr Members

IAtlStringMgr::Allocate

Allocates a new string data structure.

```
CStringData* Allocate(
   int nAllocLength,
   int nCharSize
) throw( );
```

Parameters

nAllocLength

The number of characters in the new memory block.

nCharSize

The size (in bytes) of the character type used by the string manager.

Return Value

Returns a pointer to the newly allocated memory block.

Note
Do not signal a failed allocation by throwing an exception. Instead, a failed allocation should be signaled by returning **NULL**.

Remarks

Call IAtlStringMgr::Free or IAtlStringMgr::ReAllocate to free the memory allocated by this method.

Note
For usage examples, see Memory Management and CStringT.

See Also

Reference

IAtlStringMgr Class

Other Resources

IAtlStringMgr Members

IAtlStringMgr::Clone

Returns a pointer to a new string manager for use with another instance of **CSimpleStringT**.

```
IAtlStringMgr* Clone( ) throw( );
```

Return Value

Returns a copy of the **IAtlStringMgr** object.

Remarks

Commonly called by the framework when a string manager is needed for a new string. In most cases, the **this** pointer is returned.

However, if the memory manager does not support being used by multiple instances of **CSimpleStringT**, a pointer to a sharable string manager should be returned.

Note
For usage examples, see Memory Management and CStringT.

See Also

Reference

IAtlStringMgr Class

Other Resources

IAtlStringMgr Members

IAtlStringMgr::Free

Frees a string data structure.

```
void Free(
   CStringData* pData
) throw( );
```

Parameters

pData

A pointer to the memory block to be freed.

Remarks

Frees the specified memory block previously allocated by Allocate or Reallocate.

Note
For usage examples, see Memory Management and CStringT.

See Also

Reference

IAtlStringMgr Class

IAtlStringMgr::Free

Other Resources

IAtlStringMgr Members

IAtlStringMgr::GetNilString

Returns a pointer to a string data structure for an empty string.

```
CStringData* GetNilString( ) throw( );
```

Return Value

A pointer to the **CStringData** object used to represent an empty string.

Remarks

Call this function to return a representation of an empty string.

Note
When implementing a custom string manager, this function must never fail. You can ensure this by embedding an instance o f **CNilStringData** in the string manager class, and return a pointer to that instance.

Note
For usage examples, see Memory Management and CStringT.

See Also

Reference

IAtlStringMgr Class

Other Resources

IAtlStringMgr Members

IAtlStringMgr::Reallocate

Reallocates a string data structure.

```
CStringData* Reallocate(
   CStringData* pData,
   int nAllocLength,
   int nCharSize
) throw( );
```

Parameters

pData

Pointer to the memory previously allocated by this memory manager.

nAllocLength

The number of characters in the new memory block.

nCharSize

The size (in bytes) of the character type used by the string manager.

Return Value

Returns a pointer to the start of the newly allocated memory block.

Remarks

Call this function to resize the existing memory block specified by *pData*.

Call IAtlStringMgr::Free to free the memory allocated by this method.

Note
For usage examples, see Memory Management and CStringT.

See Also

Reference

IAtlStringMgr Class

Other Resources

IAtlStringMgr Members

ATL Server Reference

ATL Server is a set of native C++ classes that allows developers to create Web applications, XML Web services, and other server applications. Many classes may also be used in client applications or components.

To start using ATL Server, you can read the ATL Server tutorial, build and run some ATL Server samples, or learn about the ATL Server architecture. Further information about the uses and features of the library is accessible from the ATL Server portal page.

In This Section

ATL Server Categories

Provides links to library elements grouped together by functional similarity.

ATL Server Classes

Provides links to the ATL Server classes, listed alphabetically.

ATL Server Enums

Provides links to the ATL Server enumerations, listed alphabetically.

ATL Server Functions

Provides links to the ATL Server functions, listed alphabetically.

ATL Server Macros

Provides links to the ATL Server macros, listed alphabetically.

ATL Server Typedefs

Provides links to the ATL Server typedefs, listed alphabetically.

ATL Server Archetypes

Provides links to the ATL Server archetypes, which are theoretical classes that supply a collection of methods, data members, static functions, typedefs, or other features, listed alphabetically.

ATL Server Response File Reference

Provides links to the server response file elements.

Session-State Database Schema

Describes how database implementation for ATL Server session-state services works.

Related Sections

ATL Server

Provides links to conceptual topics that discuss performing various tasks with ATL Server.

ATL Server Attributes

Provides links to the ATL Server attributes, which inject code to create a Web application or XML Web service request handler or performance monitoring object.

ATL Server Samples

Provides links to samples showing how ATL Server is used.

ATL Server Tutorial

Steps you through creating a simple online store to help you learn the features of ATL Server.

Active Template Library (ATL) Reference

Provides reference material for the ATL Library, a set of template-based C++ classes that simplify the programming of COM objects.

OLE DB Templates

Provides reference material for the OLE DB consumer and provider templates, a set of template classes that implement many commonly used OLE DB interfaces.

Visual C++ Libraries

Provides links to the various libraries provided with Visual C++, including ATL, MFC, OLE DB Templates, the C run-time library, and the Standard C++ Library.

Debugging

Provides links to using the Visual Studio debugger to correct logic errors in your application or stored procedures.

Breaking Changes (ATL Server)

Breaking changes in the ATL Server library.

Breaking changes

- ATLS_NO_ERROR_LOGGING has been removed.
- sproxy.exe now emits wchar_t for s:unsigned short. Use **/nowchar_t** to continue emitting unsigned short. See SPROXY.EXE: XML Web Service Proxy Generator for more information.
- CAtlHttpClient authorization does not send the user log on credentials silently unless the security zone settings permit it. To work around this, set `ATLHttpClient::SetSilentLogonOK(true);`
- CUrl::CrackUrl input flag ATL_URL_DECODE now functions exactly like ATL_URL_ESCAPE. ATL_URL_DECODE used to convert all escape characters to their real values before parsing, but this is vulnerable to spoofing, so now it converts escape characters after parsing, just like ATL_URL_ESCAPE.

See Also

Concepts

What's New in the Visual C++ Libraries

Other Resources

ATL Server Reference

ATL Server Concepts

ATL Server is a set of native C++ classes that allows developers to create Web applications, XML Web services, and other server applications. Many of the classes may also be used in client applications or components generated as ATL, MFC, or Windows projects.

In This Section

Tutorial

Shows how to create a simple online store that accesses a database. Covers some basic and advanced features of ATL Server including handling forms, validating user input using regular expressions, creating and using cookies, exposing statistics as performance counters, creating dynamic services, using cryptography, and more.

Architecture

Describes each of the elements that makes up an ATL Server application and explains how all the pieces fit together. Includes information about server response files, ISAPI extension DLLs, Web application DLLs, request handlers, and replacement methods.

XML Web Services

Explains how to create, distribute, and consume XML Web services using ATL Server and C++. Includes information about the SOAP attributes, SPROXY.EXE, and the supported data types.

Debugging

Provides information about debugging ATL Server Web applications and XML Web services. Includes information about WEBDBG.EXE, automatically attaching to the Web server process, and links to general debugging topics of interest to ATL Server developers.

Security

Describes the security context in which your ATL Server code is executed and how you can control it administratively and programmatically.

Extension Management Services

Describes how to use the services provided by ATL Server to manage aspects of your ISAPI extension DLL including its thread pool. Shows how to create an XML Web service client that uses `CSoapSocketClientT` to handle NTLM authentication.

HTTP Client Services

Describes the ATL Server classes that you can use to make HTTP requests.

Session-State Services

Provides information about memory-backed and database-backed sessions along with detailed instructions for enabling session state in your own applications and services.

Performance Monitoring

Describes the attributes and classes for exposing performance counters from your applications.

Caching

Describes the how to use ATL Server caching support, comparing the different classes involved and providing detailed instructions for exposing a cache as a service in your ISAPI extension DLL.

Error Handling

Describes error handling strategy for Web applications.

Developing Global Applications

Provides information and links of interest to developers creating world-ready server applications.

ATL Server and COM

Describes how COM is initialized for threads running in an ATL Server application thread pool and how you can override the default behavior.

ATL Server Tasks

Provides instructions for common tasks that you may want to perform in your ATL Server Web applications.

References

Contains a list of links to items of interest to ATL Server developers. In addition to links to the most important ATL Server-specific documentation, you will find links to specifications for the Internet standards supported by ATL Server, Knowledge Base articles with useful advice, and topics of general interest to developers of Web applications and XML Web services.

Related Sections

ATL Server Samples

Provides links to samples demonstrating how to use ATL Server.

ATL Server Reference

Provides links to reference documentation for the ATL Server library.

ATL Server Attributes

Provides links to reference documentation for the ATL Server attributes.

ATL Server Categories

Provides a list of ATL Server classes, interfaces, functions, enumerations, and macros grouped in functionally related categories.

ATL Server Tutorial

In this tutorial, you will create a simple online store using ATL Server, a set of native C++ classes for developing XML Web services, Web applications, and other server-based applications. During the course of this tutorial, you will become familiar with the architecture of an ATL Server application and learn how to use many key features of ATL Server.

You will see how to use ATL Server's request and response classes to read data from and write data to the client. You will see how to create Web pages from templates known as server response files that mix static content with calls to generate dynamic content, and you will see how to generate error responses that meet the needs of ATL Server, Internet Information Services (IIS), Web caches, and clients.

The online store will verify that users have logged in with a valid user name and password. If the user attempts to access a page before they have logged in, they will be automatically redirected to the login page. A cookie is used to identify the user's session.

The user's details will be stored in a database. You will see how to load database connection settings from a file and read data from HTML forms submitted to the server. You will also learn how to verify, using ATL Server's cryptographic classes, that a user has provided the correct password without storing the password in the database.

The main page of the store will provide the user with descriptions and prices for the store's products. You will obtain this information from a database using OLE DB consumer attributes. You will learn how to add data to the user's session that will be used to store information about the items in their shopping cart. You will also see how to validate data sent by the client using ATL Server's validation and regular expression support.

When the user makes a purchase, they will receive a confirmation message by e-mail and a performance counter used to track the number of orders will be incremented. You will learn how to use ATL Server's MIME and SMTP support classes to send e-mail, you will see how to use the performance monitor attributes to expose performance counters from your Web application DLLs, and you will discover how to create dynamic services that can be used by any DLL in your application.

You will be exposed to many of the new features of the libraries and development environment provided by Visual Studio .NET, including support for automatically registering Web applications with IIS during the build process.

Before you begin this tutorial, you should read the documentation describing the architecture of an ATL Server application.

Next

Creating the Solution

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples
ATL Server Architecture

Creating the Solution

In this step, you will use the ATL Server Project Wizard to create the projects for the Web application that you will develop into a simple online store.

To create the project

1. Start Visual Studio.
2. On the **File** menu, click **New**, and then click **Project**.

   The New Project dialog box appears.
3. In the Project Types pane, select **Visual C++ Projects**.
4. In the Templates pane, select **ATL Server Project**.

Note
There are two ATL Server project types listed in the Templates pane. Both activate the same wizard but with different de fault settings. You can use **ATL Server Project** when you want to create a Web application, and **ATL Server Web Servi ce** when you want to create an XML Web service.

5. In the **Name** box, enter **Tutorial**.

Caution
The name of the project is used to name some wizard-generated classes and files. Do not choose another name here, o r you will have trouble following the remaining steps of the tutorial.

6. Change the location or accept the default.
7. Click **OK**.

   The ATL Server Project Wizard appears.

The project wizard displays the **Overview** page by default. This page provides a summary of the current settings used by the wizard.

The wizard provides access to other pages that allow you to change the projects and code that the wizard generates.

- The Project Settings page lets you specify which projects the wizard will create. By default, two projects will be created: an ISAPI extension DLL project and a Web application DLL project. Using this page, you can choose to generate only one of these projects or choose to combine the two into a single project.

  The page also lets you specify deployment options for the generated project. This allows you to specify whether you want the DLLs to be registered and installed in IIS running on your development machine as part of the build process. This simplifies testing your code by eliminating the time-consuming steps necessary to get your Web application running. You can configure the deployment settings for an existing project using the Web Deployment Property Page.
- The Server Options page allows you to configure the services provided by the ISAPI extension DLL.
- The Application Options page allows you to configure the features of the Web application DLL.
- The Developer Support Options page offers a way of controlling whether the wizard will generate comments, use attributed code or nonattributed code, or add custom assert and trace handling support to your project.

To customize your project in the wizard

1. Click the **Server Options** tab.
2. Select the check box to add support for the **Data source cache**.

   The data source cache is a service that allows OLE DB data connections to be stored in the ISAPI extension. Each thread maintains its own collection of available connections, so the cache will not cause the server to block while trying to

retrieve an open connection.

3. Select the check box to add **Session services** support and select the option for **Memory-backed session-state services**.

   Session-state services allow you to store state across multiple requests. ATL Server provides the same interfaces for accessing data stored in memory or in a database. If you are creating an application that needs to run in a Web farm, you will probably prefer to store your information in a database that is accessible to all the servers. For this tutorial, choose to store the data in memory here because it will reduce the number of steps necessary to get the tutorial running.

4. Click **Finish**.

   A solution containing two projects, Tutorial and TutorialIsapi, will be generated by the wizard.

5. In Solution Explorer, view the projects and files generated by the wizard.

   Solution Explorer provides a tree view of the files and projects in the current solution.

6. Open ReadMe.txt for each project for a description of the generated files.

To change the properties of your solution

1. In Solution Explorer, click the icon for the Tutorial project.
2. Display the Property Pages for the project in one of the following ways:

   - On the **View** menu, click **Property Pages**.
   - On the **Project** menu, click **Properties**.
   - In Solution Explorer, right-click the project and click **Properties** on the shortcut menu.

   The property pages are modeless, and the display is synchronized to the current selection in Solution Explorer.

Note
The **Property Pages** dialog box shows different properties than the Properties window, which you can display by clicking **Properties Window** on the **View** menu.

3. Click **Web Deployment** to view the Web Deployment property page.
4. Change the **Relative Path** property to **bin** for the Tutorial project.

   It is a good idea to put DLLs and other files that are not intended for direct use by a client of the Web site in a separate directory from files that will be accessed directly, to make it easier to manage the security of these files. In this tutorial, you will put the DLLs in a directory that is beneath the virtual directory used for the content files. See Deploying ATL Server Applications for more information on the benefits of separating content and binaries into different folders.

5. Repeat steps 2 through 4 for the TutorialIsapi project.
6. In Solution Explorer, double-click Tutorial.srf.
7. Click the **HTML** tab at the bottom of the page to switch to HTML view.
8. Change the handler tag so that it will refer to the correct location by adding **bin** as shown:

   ```
   {{handler bin\Tutorial.dll/Default}}
   ```

   The handler tag uses a relative path to refer to the DLL that provides the request handler that provides the functionality of the page.

9. On the **Build** menu, click **Build Solution**.

   The progress of the build is indicated by a small icon in the status bar or by text displayed in the Output window.

10. Use the Task List window or Output window to view the results of the build.

    You can also display these windows by clicking the **View** menu and then clicking **Other Windows**. From this menu, you

can click **Task List** or **Output**.

Note
The Dynamic Help window displays links to help topics on the currently selected item. Click a link in the Dynamic Help window or use F1 to get help on the currently selected item.

11. Use the browser of your choice to view the following URL and verify that the deployment has been successful and that IIS has been automatically configured:

```
http://localhost/tutorial/tutorial.srf
```

If you do not see a greeting, use the troubleshooting guide to track down the problem.

Next

Deploying the Database

See Also

Reference

ATL Server Tutorial

ATL Server Project Wizard

Other Resources

ATL Server Architecture

Deploying the Database

This tutorial uses a database that stores information about the products and users of an online store. You can obtain an .mdb file that contains the schema and the small amount of data that you will need from the ATL Server Tutorial sample, which also contains the code that you will develop by following this tutorial.

You can import this database into SQL Server or MSDE and access it using the SQL Server driver or access the .mdb file directly using the Jet driver. The driver you choose will depend on the products installed on your system. The code that you write in subsequent steps will be identical because the database is accessed using OLE DB, and connection data is loaded from a Microsoft Data Link file.

To use the Microsoft Jet 4.0 OLE DB provider

1. Get Tutorial.mdb and Tutorial.udl from the ATL Server Tutorial Sample.
2. Copy both files to the same location to which Tutorial.dll was deployed (for example, c:\inetpub\wwwroot\Tutorial\bin\).
3. Double-click the .udl file.

   The **Data Link Properties** dialog box appears.
4. Click the **Provider** tab and select **Microsoft Jet 4.0 OLE DB Provider**.
5. Click the **Connection** tab and point to the Tutorial.mdb database.
6. Click **OK**.

Note
It is not recommended that you use a Jet database from a Web site with more than a small number of users. SQL Serve r and MSDE are more scalable.

To use the Microsoft OLE DB provider for SQL Server

1. Get Tutorial.mdb and Tutorial.udl from the ATL Server Tutorial Sample.
2. Start SQL Server Enterprise Manager.
3. On the **Action** menu, click **All Tasks**, and then click **Import Data**.

   The Data Transformation Services Import/Export Wizard appears.
4. Click **Next** to go to the **Choose a Data Source** tab.
5. Select **Microsoft Access** as the data source.
6. Enter the full path to Tutorial.mdb in the **File name** box.
7. Click **Next**.
8. Select **Microsoft OLE DB Provider For SQL Server** as the destination.
9. Select <new> as the **Database** and enter **Tutorial** as the name when requested.
10. Click **Next**.
11. Ensure that **Copy table(s) and view(s) from the source database** is selected.
12. Click **Next**.
13. Click **Select All** to copy all the tables into the new database.
14. Click **Next** two times.
15. Click **Finish**.
16. View the designer for the **Order** table.
17. Set the following properties for the **Id** column:

Property	Value
Identity	Yes
Identity Seed	1
Identity Increment	1

18. View the designer for the **SiteUser** table.
19. Make the **UserName** column the table's primary key.
20. Copy Tutorial.udl to the same location to which Tutorial.dll was deployed (for example, c:\inetpub\wwwroot\Tutorial\bin\).
21. Double-click the .udl file

    The **Data Link Properties** dialog box appears.
22. Click the **Provider** tab and select **Microsoft OLE DB Provider For SQL Server**.
23. Click the **Connection** tab and point to the database created in the previous steps.
24. Click **OK**.

Security Considerations

You will need to ensure that the clients of the Web site have read access to the .udl file and read-write access to the database. There are a number of ways to do this, but to keep the code simple, you will give access to the Internet Guest Account, **IUSR**_*Machine*, administratively.

To ensure that the database connection allows access to the Internet Guest Account

1. Right-click the Tutorial.udl file and select **Properties**.

   The **Tutorial.udl Properties** dialog box appears.
2. Select the **Security** tab and click **Add**.

   The **Select User, Computers, or Groups** dialog box appears.
3. Change the following settings:

   For Windows XP:

   - Set **Object Types** to **Users**.
   - Set **Locations** to the local machine name.
   - In the **Check Names** box, enter the name of the Internet Guest Account
   - Click **OK**.

   For Windows 2000:

   - From the **Look in** drop-down menu, select the local machine name.
   - Enter the name of the Internet Guest Account and click **Check Names**.
   - Click **OK**.

Note
You can find the name of the Internet Guest Account in the Computer Management dialog box (right-click **My Computer** and select **Manage**; then open **System Tools**, then **Local Users and Groups**, and then **Users**.)

4. If the database connection uses Windows authentication, make sure that the **IUSR**_*Machine* account has permission to read and write to the database. If the database connection uses SQL Server authentication, make sure that the account

used for the connection has permission to read and write to the database and that the security information is persisted with the connection settings.

To ensure that the Internet Guest Account can read the .udl file from disk

- Set the NTFS permissions of the file so that the **IUSR**_*Machine* account has permission to read it.

Note
If you use the Microsoft Jet 4.0 OLE DB provider, you will also need to give the **IUSR**_*Machine* account permissions to write to the directory holding the .mdb file. This is because the Jet OLE DB provider needs to create a file that contains information used for locking the database so that it is not corrupted by simultaneous access.

See ATL Server Security for a discussion of the user accounts involved in an ATL Server application.

Next

Creating the Login Page

See Also

Reference

ATL Server Tutorial

Creating the Login Page

In this step, you will create a login page that will allow users to enter their name and password before entering the store. The user interface will be provided by a server response file (SRF) that generates a simple HTML form. The application logic will live in a request handler that validates the data sent to the server, checks that the password matches the user name supplied by the user, sends a cookie identifying the user back to the browser, and redirects the user to the main page of the store.

Procedures

To add and set up a server response file in your project

1. In Solution Explorer, right-click the Tutorial project.
2. On the shortcut menu, click **Add**, and then click **Add New Item**.

   The **Add New Item** dialog box appears.
3. In the Templates pane, click **SRF File (.srf)**, and enter **login.srf** in the **Name** box.
4. Click **Open**.

   Visual Studio creates the file on disk, adds it to your project, and opens it in the HTML Designer.
5. Click the **HTML** tab at the bottom of the page.

   The new server response file is a simple HTML page with a handler tag that refers to the project's default handler.
6. Change the code in this page to include an HTML form with controls for the user's name, password, and e-mail address as shown. You can experiment with writing this code manually in HTML view, dragging items from the toolbox in Design view, or copying and pasting the code from below, or from the ATL Server Tutorial Sample.

```
<html>
   {{handler bin\tutorial.dll/Login}}
   <body>
      {{Status}} Please log in:
      <p>
         <table>
            <form action="login.srf" method="post">
               <tr>
                  <td>
                     Username:
                  </td>
                  <td>
                     <input type="text" size="50" name="username">
                  </td>
               </tr>
               <tr>
                  <td>
                     Password:
                  </td>
                  <td>
                     <input type="password" size="50" name="password">
                  </td>
               </tr>
               {{ if Debug }}
               <tr>
                  <td>
                     Email:
                  </td>
                  <td>
                     <input type="text" size="100" name="email">
```

```
                                    (Only required for first login)
                      </td>
                   </tr>
                   {{ endif }}
                   <tr>
                      <td>
                         <input type="submit" value="Login">
                      </td>
                   </tr>
                </form>
             </table>
          </p>
       </body>
    </html>
```

The HTML form on this page posts back to itself, so the request handler responsible for the login page must be able to accept GET requests (when the user navigates to the page directly) and POST requests (when the user completes the form and submits it to the server).

You can see that the handler tag now includes the relative path to Tutorial.dll that is appropriate for the deployment settings that you changed earlier in the tutorial. In addition, the tag now refers to the **Login** handler, which you will create in the next step.

The first replacement tag in the file calls the **Status** replacement method provided by the Login handler. This will be used to write information about the status of a login attempt to the HTML page. If, for example, the user supplies an invalid combination of name and password, the **Status** method will write information to the response stream that informs the user of their failed attempt to login.

The next pair of tags forms an **if-endif** block around the HTML for the text box that allows the user to enter an e-mail address. If the **Debug** replacement method returns **HTTP_S_FALSE**, the user will not be able to enter their e-mail address.

Note

This code is here to simplify the tutorial by not providing a complete implementation of the logic and user interface that would allow the user to create a new account in a real store, but to allow you to add users without accessing the database directly. To that end, you write code that automatically creates a new account for the user in debug builds if the name they supply does not match the name of an existing user. The code is disabled in release builds because this way of adding users is not user friendly enough for general use. The e-mail address is only used when adding users so the HTML code that allows it to be entered is only present in debug builds. To see a more complete sample demonstrating the creation of user accounts for online stores, see the ConfirmUser section of the WebFeatures sample.

Next

Creating the Login Request Handler

See Also

Reference

ATL Server Tutorial

Other Resources

ATL Server Architecture

Creating the Login Request Handler

In the previous step, you created the server response file that will allow users to log in to the online store. In this step, you will create the request handler class used by that server response file. The request handler will be responsible for:

- Obtaining the user's name and password from the form variables posted to the server.
- Rejecting the data if it is too large or invalid.
- Accessing the user database to determine whether the name and password match a registered user.
- Redirecting the user to the main page of the store if they log in successfully.

Procedures

To create and modify the files that will hold the request handler

1. In Solution Explorer, right-click the Tutorial project.
2. On the shortcut menu, click **Add**, and then click **Add New Item**.

   The **Add New Item** dialog box appears.
3. In the Templates pane, click **C++ File (.cpp)**, and enter **Login.cpp** in the **Name** box.
4. Click **Open**.

   Visual Studio creates the file on disk and adds it to your project.
5. In Solution Explorer, right-click the Tutorial project.
6. On the shortcut men, click **Add**, and then click **Add New Item**.

   The **Add New Item** dialog box appears.
7. In the Templates pane, click **Header File (.h)**, and enter **Login.h** in the **Name** box.
8. Click **Open**.

   Visual Studio creates the file on disk and adds it to your project.
9. Add the following code to Login.h:

```
#pragma once

[ request_handler("Login") ]
class CLoginHandler
{
public:
   CStringA m_strStatus;

   HTTP_CODE ValidateAndExchange();

   [tag_name(name="Status")]
   HTTP_CODE OnStatus();

   [tag_name(name="Debug")]
   HTTP_CODE OnDebug();
};
```

   This is the definition of the **Login** request handler referred to in login.srf. You can see the use of the request_handler attribute that causes `CLoginHandler` to derive from CRequestHandlerT and ensures that the class is available as a request handler named **Login**.

   The `m_strStatus` data member will hold information about the status of a login attempt. The status information will be

set in `ValidateAndExchange` and written to the response stream in `OnStatus`.

`ValidateAndExchange` is an initialization method that overrides CRequestHandlerT::ValidateAndExchange. This method is called by ATL Server before the response is generated so it is great for doing work such as initializing data members and validating form variables.

`OnStatus` and `OnDebug` are the implementations of the **Status** and **Debug** replacement methods used in login.srf. The tag_name attribute states that these are replacement methods and includes the tag name that can be used to refer to these methods.

10. Add the following code to Login.cpp:

```
#include "StdAfx.h"
#include "Login.h"

HTTP_CODE CLoginHandler::ValidateAndExchange()
{
   // Set the content-type of the response.
   m_HttpResponse.SetContentType("text/html");

   // Create a validation context to hold the results of the data validation.
   CValidateContext validationContext;

   // Look for the username and password in the form data.
   // Ensure that the username and password are between 4 and 50 chars.
   const CHttpRequestParams& FormFields = m_HttpRequest.GetFormVars();
   FormFields.Validate("username", static_cast<LPCSTR*>(NULL), 4, 50,
                                                &validationContext);
   FormFields.Validate("password", static_cast<LPCSTR*>(NULL), 4, 50,
                                                &validationContext);

   if (validationContext.ParamsOK())
   {
      // <A HREF="vcconatlservertutorialauthenticatingtheuser.htm">TODO - Verify passw
ord and redirect to main page</A>
   }

   return HTTP_SUCCESS;
}

HTTP_CODE CLoginHandler::OnStatus()
{
   // Output the status text.
   m_HttpResponse << "<p>" << m_strStatus << "</p>";
   return HTTP_SUCCESS;
}

HTTP_CODE CLoginHandler::OnDebug()
{
   // Return true if this is a debug build.
   #ifdef DEBUG
      return HTTP_SUCCESS;
   #else
      return HTTP_S_FALSE;
   #endif
}
```

This is the implementation of the `CLoginHandler` methods:

- `OnStatus` writes the contents of `m_strStatus` to the response stream within an HTML paragraph. The stream

insertion operator (operator <<) used here is provided by CWriteStreamHelper, from which m_HttpResponse derives. This provides an easy and efficient way of building a response and handles most common data types.

- `OnDebug` uses the preprocessor to determine whether or not the current build is a debug build. If it is, `OnDebug` returns **HTTP_SUCCESS**, which is treated as **True** when used in an **if** or **while** construct in a server response file. Otherwise, `OnDebug` returns **HTTP_S_FALSE**, which is treated as **False** when used in an **if** or **while** construct in a server response file

The code in `ValidateAndExchange` is more complicated and currently incomplete. First, the code sets the content type of the response. The content type describes, using the MIME standard, the type of data being returned by the request handler. In this case, the response is HTML, so the **text/html** MIME type is used. Other common MIME types are **text/xml** for XML text, **text/plain** for plain text, and **image/png** for Portable Network Graphics (PNG) images.

Note

ATL Server provides the GetContentTypeFromFileName function as a way of obtaining the MIME type associated with a file extension using information in the registry. For more information, see RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt).

The remaining code validates the "username" and "password" form variables. Form variables are items sent to the server as part of an HTML form. The variables consist of a name (the value of the **name** attribute of the HTML input element) and a value (the value of the **value** attribute of the **input** element such as text typed in or chosen by the user or the contents of a file). A reference to the collection of form variables can be obtained by calling CHttpRequest::GetFormVars.

The form variables collection derives from CValidateObject making it very easy to check that form variables are present or comply with some simple limits by calling Validate. In this code, the "username" and "password" variables are checked to be between four and fifty characters in length.

Note

Although the length is checked, this code does not check that the user name and password match a known format. This reduces the usability of the site because it allows users to enter invalid data without catching the mistake early and giving them a chance to correct it. This could also potentially reduce the security of the site, because if the data were to be written back to a Web page exactly as entered, it could expose the site and its users to cross-site script attacks. However, this application will not write the data back to a Web page, so the site remains secure. See the Input Sample for a detailed example of input validation.

The CValidateContext object keeps track of validation failures so that all the required data is validated before handling any errors. This is particularly useful if you have a large form with many fields and you want to provide useful feedback to the user about which fields have not been correctly filled out.

Note

The `static_cast` is necessary in the call to `Validate` to ensure that `NULL` is given a type that can be used by the template method.

In this case, a success code is returned if the form variables could not be successfully validated. This handles the case where the user has requested the page directly and keeps the application simple. Most errors that need status information conveyed to the client will be handled in the code that you write in the following sections.

Next

Connecting to the Database

See Also

Reference

ATL Server Tutorial

Other Resources

ATL Server Architecture

Connecting to the Database

In this step, you write code to obtain the database connection settings from a Microsoft Data Link (.udl) file. This will allow the code to run against different databases without modification. The code that you create will be designed so that it can be used in any C++ project that needs access to database connection settings.

To create a new header file for reusable code

1. In Solution Explorer, right-click the Tutorial project.
2. On the shortcut menu, click **Add**, and then click **Add New Item**.

   The **Add New Item** dialog box appears.
3. In the Templates pane, click **Header File (.h)**, and enter **Helpers.h** in the **Name** box.
4. Click **Open**.

   Visual Studio creates the file on disk and adds it to your project.
5. Add the following code to the file:

   ```
   #pragma once

   namespace VCUE
   {
      // TODO - write code here
   } // namespace VCUE
   ```

   The reusable code that you write will live in the `VCUE` namespace so that it can be easily incorporated into your own applications without name conflicts.
6. Replace the `TODO` comment with the following code:

   ```
   // A class that retrieves an OLE DB initialization string from a file
   class COleDbInitializationString
   {
   public:
       HRESULT LoadFromFile(LPCOLESTR szFile)
       {
           ATLASSERT(szFile != NULL);

           CComPtr<IDataInitialize> spDataInitialize;
           HRESULT hr = spDataInitialize.CoCreateInstance(
                                __uuidof(MSDAINITIALIZE));
           if (SUCCEEDED(hr))
               hr = spDataInitialize->LoadStringFromStorage(
                         szFile, &m_spInitializationString);

           return hr;
       }

       operator LPCOLESTR () const
       {
           return m_spInitializationString;
       }

   private:
       CComHeapPtr<OLECHAR> m_spInitializationString;
   };
   ```

```
// Call this function to get the name of the module
//without the file extension
// Returns true on success, false on failure
inline bool GetModuleNoExtension(CStringW& strModule)
{
    bool bSuccess = false;
    wchar_t szBuffer[_MAX_PATH];
    const size_t BufferCharacters = sizeof(szBuffer) /
      sizeof(szBuffer[0]);
    DWORD dwCharacters = GetModuleFileNameW(
           _AtlBaseModule.GetModuleInstance(), szBuffer,
           BufferCharacters);
    if (dwCharacters && (dwCharacters < BufferCharacters))
    {
        PathRemoveExtensionW(szBuffer);
        strModule = szBuffer;
        bSuccess = true;
    }

    return bSuccess;
}
```

You can obtain the initialization string used to connect to the database by loading it from a .udl file using an `MSDAINITIALIZE` object and calling IDataInitialize::LoadStringFromStorage. This method returns a string that needs to be freed, so you wrap the code in the `COleDbInitializationString` class and take advantage of the CComHeapPtr template to remove the burden of managing that memory.

The next step is determining the location of the .udl file that needs to be loaded. It is certainly convenient and intuitive to load the .udl file that is in the same directory and has the same name as the current module. `GetModuleNoExtension` provides a convenient way of obtaining the path of the current module without the extension. Note the careful checks on the return value of GetModuleFileName and the use of PathRemoveExtension to ensure that path manipulations are performed correctly.

After loading the connection settings, you must consider how to handle the connections themselves. In Creating the Solution, you added support for the ATL Server database connection cache to the ISAPI extension.

The database connection cache allows request handlers to eliminate the cost of opening a database connection on every request by having the ISAPI extension manage a collection of open connections that can be obtained on subsequent requests. Each thread manages its own connections to avoid contention. The basic procedure is to check to see if the database connection is in the cache, if it is not there, create it and add it to the cache. The GetDataSource function does exactly that in a single step.

Note
Unlike the other ATL Server caches, the items that you add to the data source cache are guaranteed to be there until you remove them.

In this example, you will only run one or two queries for any request, so you can simplify the code and combine the following procedures into a single function:

- Get the connection settings.
- Obtain or open the connection.
- Execute the query.

To simplify the code and combine the procedures

- Add the following code to that already within the `VCUE` namespace in Helpers.h:

```
template <class TDatabaseCommand>
inline HRESULT OpenCommandRowset(
```

```
      IServiceProvider* pServiceProvider, TDatabaseCommand& dbCommand,
      LPCOLESTR szFile, LPCOLESTR szConnectionId = NULL
   )
{
    ATLASSERT(pServiceProvider != NULL);
    ATLASSERT(szFile != NULL);

    // If the connection ID is not specified, just use the
    // name of the file
    if (!szConnectionId)
        szConnectionId = szFile;

    HRESULT hr = E_UNEXPECTED;
    COleDbInitializationString initializationString;
    hr = initializationString.LoadFromFile(szFile);
    if (SUCCEEDED(hr))
    {
        CDataConnection connection;
        if (S_OK != GetDataSource(pServiceProvider,
            COLE2T(szConnectionId),
                                        initializationString,
                                        &connection))
            return E_FAIL;

        hr = dbCommand.OpenRowset(connection);

        #ifdef DEBUG
            if (FAILED(hr) || DB_S_ERRORSOCCURRED == hr)
                AtlTraceErrorRecords(hr);
        #endif
    }

    return hr;
}

// This overload gets the name of the UDL file
// from the name of the module
template <class TDatabaseCommand>
inline HRESULT OpenCommandRowset(
      IServiceProvider* pServiceProvider, TDatabaseCommand& dbCommand
   )
{
    ATLASSERT(pServiceProvider != NULL);

    CStringW strUdlFile;
    if (GetModuleNoExtension(strUdlFile))
    {
        strUdlFile.Append(L".udl");
        return OpenCommandRowset(pServiceProvider, dbCommand,
          strUdlFile);
    }
    return E_UNEXPECTED;
}
```

Both overloads of `OpenCommandRowset` allow you to specify the service provider interface that supports data source caching and a database command object representing the query to be executed. The first overload also allows you to specify the filename from which the connection settings should be obtained and the string used to identify the connection in the cache. The second overload uses the name of the .udl file in the same directory as the current module as both the filename and the connection identifier.

Note

`OpenCommandRowset` is a function template because the command object passed is of an unknown type. `TDatabaseCommand` is a class that has had the db_command attribute applied. This attribute injects the `OpenRowset` method used in this code, so `OpenCommandRowset` cannot accept a common base class.

Next

Security Considerations

See Also

Reference

ATL Server Tutorial

Security Considerations

Authentication

The architecture of the online store requires that you provide a mechanism for authenticating users. Providing encrypted authentication mechanisms is a problem that requires careful attention. If possible, you should use the mechanisms provided by Windows/IIS, Passport, Microsoft Commerce Server, or another Microsoft or third-party product. If, however, your business requires you to provide your own way of authenticating users, this topic discusses issues that you need to consider by working through this tutorial. A complete membership system is not provided; you will, however, see some useful and reusable code that helps protects the user's password.

The authentication system works in the following way:

- The user will supply their name and password by submitting an HTML form. This form will be obtained and submitted using HTTPS to ensure that the user's sensitive information cannot be read if the submission is intercepted.

Note
Configuring the login page to use HTTPS is an administration issue. It does not affect the code in the tutorial and is not addressed here.

- The server will validate the user's name and password against information stored in a database. The database will not store the password itself. Instead, it will store a hash value that was generated from the password and from a random number (known as a salt) that will also be stored in the database.

  The server will execute a query to obtain the hash and the salt for a given user name, and then it will add the salt to the supplied password, hash the result, and compare with the hash obtained from the database. If the hashes are the same, the user supplied the correct password.

- If the user name and password match, the user will receive a cookie that will be used to identify them for subsequent requests.

Not storing the password helps prevent an attacker being able to log in as a site's users even if the database is compromised. The attacker cannot log in using only the hash; he or she must first obtain the password.

If the attacker has a lot of computer time and power, he or she could still crack a password from the database, but adding the salt makes the attacker's job much harder because it increases the number of possible hash values and ensures that there is random data in every hash. Storing random hashes is extremely important because in a large user group passwords will not be completely random, even with a strong password policy. Mixing in random data helps prevents the attacker from decreasing the time taken to crack a password by targeting their attack against a hash that appears frequently in the database.

You can help maintain security by increasing the size of the salt value and by storing the salt in a different physical database than you use to store the hash value. Anything that makes it harder for the attacker to find the hash, salt, and hashing algorithm that go together is a bonus.

It still vital to help protect the database itself and require users to pick strong passwords that are not highly vulnerable to dictionary attacks by requiring a mix of uppercase letters, lowercase letters, numbers, and symbols and a minimum length. Periodic changes to passwords are also necessary to help maintain the security of your site and its users. All the necessary code to do this is not provided in this tutorial, but you need to apply these kinds of policies to your own applications no matter what type of authentication you use.

It is important to keep user passwords out of the hands of an attacker even if your Web site is not one whose content or features are particularly sensitive. Given that users can use the same password on multiple sites, you should help protect their password as carefully as you would treat their credit card details.

Cookies

When cookies are used to identify a user's session, there is a risk that an attacker could intercept the data passing between the client and the server, obtain the cookie, and use it to impersonate the real client. You can help prevent this possibility by using HTTPS so that the attacker cannot easily obtain the cookie. This helps protect your client's information but may impose some overhead that may not be required for your particular Web site.

If appropriate, you can reduce the possibility of impersonation attacks without requiring HTTPS by limiting the length of time for which a session cookie is valid and by requiring reauthentication at sensitive points of the Web site (such as checkout).

The tutorial code does not include the code for helping to control the lifetime of the cookie or forcing reauthentication, but you can add it fairly easily.

Database Queries

In this tutorial, all the database queries that are performed using information submitted by the client will be parameterized queries where the values of the parameters are bound to variables in our OLE DB consumer code. Not only is this the most efficient way of accessing the database, it also helps to provide greater security. Never build up a SQL query by concatenating string information provided by the user with the rest of your query; not only is it inefficient, it is easy for the user to change the meaning of your query.

Next

Securing the User's Password

See Also

Reference

ATL Server Tutorial

Securing the User's Password

In this step, you will create some helper functions to simplify the process of hashing and comparing passwords. This code will use the cryptographic classes provided by ATL Server.

To create a new header file for the cryptographic helper functions

1. In Solution Explorer, right-click the Tutorial project.
2. On the shortcut menu, click **Add**, and then click **Add New Item**.

   The **Add New Item** dialog box appears.
3. In the Templates pane, click **Header File (.h)**, and enter **Encrypt.h** in the **Name** box.
4. Click **Open.**

   Visual Studio creates the file on disk and adds it to your project.
5. Add the following code to Encrypt.h:

```
#pragma once

#include <atlcrypt.h>

namespace VCUE
{
   inline HRESULT EnsureAcquire(
         CCryptProv& prov,
         LPCTSTR pszContainer = NULL,
         LPCTSTR pszProvider = MS_DEF_PROV,
         DWORD dwProvType = PROV_RSA_FULL,
         DWORD dwFlags = CRYPT_VERIFYCONTEXT | CRYPT_SILENT
         )
   {
      HRESULT hr = prov.Initialize(dwProvType, pszContainer,
      pszProvider, dwFlags);

      if (hr == NTE_KEYSET_NOT_DEF)
            hr = prov.Initialize(dwProvType, pszContainer, pszProvider,
                                            dwFlags | CRYPT_NEWKEYSET);

      return hr;
   }
}
```

`EnsureAcquire` is a helper function that initializes a cryptographic provider (an object that provides cryptographic functions). The function provides sensible default arguments for use in server applications by setting the flag that disables any user interface that the cryptographic provider might display to the user (`CRYPT_SILENT`) and the flag that informs the provider that private keys do not need to be accessed (`CRYPT_VERIFYCONTEXT`). The function also ensures that if initialization fails because the container does not exist, an attempt is made to create it. For more information, see CCryptProv::Initialize. Add the following code to Encrypt.h:

```
   inline HRESULT CreateSaltedHash(
            CCryptProv& Provider,
            const BYTE* Secret, DWORD SecretLength,
            const BYTE* Salt, DWORD SaltLength,
            BYTE* Hash,   DWORD& HashLength
         )
```

```
{
   HRESULT hr = E_FAIL;
   HCRYPTHASH hHash = 0;
   if (CryptCreateHash(Provider.GetHandle(), CALG_MD5,
       0, 0, &hHash))
   {
      CCryptHash oHash(hHash, TRUE);
      hr = oHash.AddData(Secret, SecretLength);
      if (SUCCEEDED(hr))
      {
         hr = oHash.AddData(Salt, SaltLength);
         if (SUCCEEDED(hr))
         {
            DWORD Size = 0;
            hr = oHash.GetSize(&Size);
            if (SUCCEEDED(hr))
            {
               if (Size <= HashLength)
                  hr = oHash.GetValue(Hash, &HashLength);
               else
                  hr = E_OUTOFMEMORY;
            }
         }
      }
   }

   return hr;
}
```

`CreateSaltedHash` uses the MD5 hashing algorithm to generate a hash from a secret (such as a password) and a salt (a random number). The code creates a CCryptHash object, adds the secret and the salt to the hash, and then gets the value of the hash if the buffer provided by the caller is big enough to hold it. Add the following code to Encrypt.h:

```
inline HRESULT HashSecret(
         const BYTE* Secret, DWORD SecretLength,
         BYTE* Salt, DWORD& SaltLength,
         BYTE* Hash, DWORD& HashLength
         )
{
   CCryptProv Provider;
   HRESULT hr = EnsureAcquire(Provider);
   if (SUCCEEDED(hr))
   {
      hr = Provider.GenRandom(SaltLength, Salt);
      if (SUCCEEDED(hr))
      {
         hr = CreateSaltedHash( Provider,
               Secret, SecretLength,
               Salt, SaltLength,
               Hash, HashLength);
      }
   }
   return hr;
}
```

`HashSecret` takes a secret and returns a salt and the hash generated by combining the secret and the salt. It wraps the calls necessary to obtain the cryptographic provider, generate the random salt value, and create the hash in a single function. Add the following code to Encrypt.h:

```
    inline HRESULT CompareSecret(
             const BYTE* Secret, DWORD SecretLength,
             const BYTE* Salt, DWORD SaltLength,
             const BYTE* Hash, DWORD HashLength
             )
    {
       DWORD CalculatedHashLength = HashLength;
       BYTE* CalculatedHash = new BYTE[HashLength];
       if (!CalculatedHash)
          return E_OUTOFMEMORY;

       CCryptProv Provider;
       HRESULT hr = EnsureAcquire(Provider);
       if (SUCCEEDED(hr))
       {
          hr = CreateSaltedHash( Provider,
                Secret, SecretLength,
                Salt, SaltLength,
                CalculatedHash, CalculatedHashLength);
          if (SUCCEEDED(hr))
          {
             hr = S_FALSE;
             if (CalculatedHashLength == HashLength)
             {
                if (0 == memcmp(Hash, CalculatedHash, HashLength))
                   hr = S_OK;
             }
          }
       }
       delete [] CalculatedHash;
       return hr;
    }

 } // namespace VCUE
```

`CompareSecret` takes a secret, a salt, and a hash. If the result of hashing the secret and the salt is the same as the hash passed to the function, `CompareSecret` returns S_OK. If the hashes are different, the function returns S_FALSE. Otherwise, the function returns an error HRESULT.

These functions are all you need to validate users' passwords. `HashSecret` will be used to generate the salt and the hash that will be stored in the database from a user's password. `CompareSecret` will compare the stored salt and hash with a password and tell you whether they match.

Next

Simplifying OLE DB Consumers

See Also

Reference

ATL Server Tutorial

Simplifying OLE DB Consumers

In the following steps, you will create a number of db_command classes for accessing the database. These classes will have data members that correspond to parameters in a SQL query or columns in the result set, and these data members will often be character arrays to which you need to write a new value. Each data member will also have associated members describing the length and status of the data. To help make sure that you can write data to these members safely and keep the related members synchronized, you will provide a helper function for writing strings to character array members.

To create the db_command classes for accessing the database

- Add the following code to Helpers.h in the `VCUE` namespace:

```
template <class TCharArray>
inline HRESULT SetOleDbStringMember(TCharArray& data, DBSTATUS& status,
                                    DBLENGTH& length, LPCSTR szNewValue)
{
   // Ensure that only char arrays are allowed.
   static_cast<char[sizeof(data)]>(data);

   // Ensure that user passes valid string.
   ATLASSERT(szNewValue != NULL);

   HRESULT hr = E_FAIL;

   #pragma warning(push)
   // conversion from 'size_t' to 'DBLENGTH', possible loss of data
   #pragma warning(disable : 4267)

   length = strlen(szNewValue);

   #pragma warning(pop)

   // Check length of string.
   if (length && (length < sizeof(data)))
   {
      // Copy string.
      if (SafeStringCopy(data, szNewValue))
      {
         hr = S_OK;
         status = DBSTATUS_S_OK;
      }
   }

   // Reset data, length, status on failure
   if (FAILED(hr))
   {
      data[0] = 0;
      length = 0;
      status = DBSTATUS_S_ISNULL;
   }

   ATLASSERT(length < sizeof(data));
   ATLASSERT((FAILED(hr) && (0 == length) && (DBSTATUS_S_ISNULL == status))
       || (SUCCEEDED(hr) && (0 != length) && (DBSTATUS_S_OK == status)));

   return hr;
}
```

This code ensures that the argument passed as the `data` parameter is a fixed-size `char` array. This array is the destination for the string that is passed as the `szNewValue` parameter. The function copies the string to the `data` parameter and takes care to ensure that the `length` parameter contains the number of characters in the `data` parameter and the `status` parameter is updated even when errors occur.

Next

Getting User Information

See Also

Reference

ATL Server Tutorial

Getting User Information

In this step, you will see how to create attributed OLE DB consumer code to add new users to the database and to obtain password information from the database. You can use the ATL OLE DB Consumer Wizard to generate OLE DB consumer classes in your own code, but for this tutorial, you will manually add the code.

To create a .h file and .cpp file for the database command classes

1. In Solution Explorer, right-click the Tutorial project.
2. On the shortcut menu, click **Add**, and then click **Add New Item**.

   The **Add New Item** dialog box appears.
3. In the Templates pane, click **C++ File (.cpp)**, and enter **Commands.cpp** in the **Name** box.
4. Click **Open**.

   Visual Studio creates the file on disk and adds it to your project.
5. In Solution Explorer, right-click the Tutorial project.
6. On the shortcut men, click **Add**, and then click **Add New Item**.

   The **Add New Item** dialog box appears.
7. In the Templates pane, click **Header File (.h)**, and enter **Commands.h** in the **Name** box.
8. Click **Open**.

   Visual Studio creates the file on disk and adds it to your project.
9. Add the following code to Commands.h:

```
[
    db_command(
        L"SELECT [SiteUser].[Password], "
        L"[SiteUser].[Salt] "
        L"FROM [SiteUser] "
        L"WHERE [SiteUser].[UserName] = ?"
    )
]
class CSiteUser
{
public:
    [ db_param(1, status = m_dwUserNameStatus,
      length = m_dwUserNameLength) ]
    char m_UserName[51];

    [ db_column(1, status = m_dwPasswordStatus,
      length = m_dwPasswordLength) ]
    BYTE m_Password[51];

    [ db_column(2, status = m_dwSaltStatus,
      length = m_dwSaltLength) ]
    BYTE m_Salt[4];

    DBSTATUS m_dwUserNameStatus;
    DBSTATUS m_dwPasswordStatus;
    DBSTATUS m_dwSaltStatus;

    DBLENGTH m_dwUserNameLength;
    DBLENGTH m_dwPasswordLength;
```

```
    DBLENGTH m_dwSaltLength;
};
```

The hashed password and the salt used to generate the hash are stored in the database in the `Password` and `Salt` columns of the `SiteUser` table. The SQL query passed here to the db_command attribute obtains the hash and the salt for a particular user name. You will use this query to determine whether an attempt to login to the site should be successful.

The `CSiteUser` class has data members for each column (`Password` and `Salt`) and parameter (`UserName`) used in the query. The db_param and db_column attributes associate the members that hold data used by the query with the data members that hold length and status information. It is optional whether you supply status and length members in your own consumer classes, but it is highly recommended because it is the only way of determining whether the data members hold useful data.

To make the `db_command` classes easier to use, you will derive helper classes that ensure the correct initialization of the data members, provide methods for setting input parameters, and provide methods for obtaining information about the results of a query. Classes are derived from the `db_command` classes rather than adding helper functions directly to those classes because the `db_command` attribute injects some hidden classes in to the hierarchy that can cause unexpected behavior.

```
class CCommandSiteUser : public CSiteUser
{
public:
   CCommandSiteUser();
   void Clear();
   HRESULT SetUserName(LPCSTR szUserName);
   HRESULT ComparePassword(LPCSTR szPassword);
};
```

The class `CCommandSiteUser` has a constructor and a `Clear` method that initializes all the data members, a `SetUserName` method for providing the value of the input parameter used by the query, and a `ComparePassword` function that can be called after the query has successfully executed to compare a password with the hash and salt obtained from the database.

```
[
    db_command(
        "INSERT INTO [SiteUser] "
        "( [Password], [Salt], [UserName], [Email] ) "
        "VALUES(?, ?, ?, ?)"
    )
]
class CCreateUser
{
public:
    [ db_param(1, status = m_dwPasswordStatus,
      length = m_dwPasswordLength) ]
    BYTE m_Password[51];

    [ db_param(2, status = m_dwSaltStatus,
      length = m_dwSaltLength) ]
    BYTE m_Salt[4];

    [ db_param(3, status = m_dwUserNameStatus,
      length = m_dwUserNameLength) ]
    char m_UserName[51];

    [ db_param(4, status = m_dwEmailStatus,
      length = m_dwEmailLength) ]
```

```
    char m_Email[256];

    DBSTATUS m_dwPasswordStatus;
    DBSTATUS m_dwSaltStatus;
    DBSTATUS m_dwUserNameStatus;
    DBSTATUS m_dwEmailStatus;

    DBLENGTH m_dwPasswordLength;
    DBLENGTH m_dwSaltLength;
    DBLENGTH m_dwUserNameLength;
    DBLENGTH m_dwEmailLength;

    void GetRowsetProperties(CDBPropSet* pPropSet)
    {
        pPropSet->AddProperty(DBPROP_IRowsetChange, true);
    }
};
```

`CCreateUser` looks exactly as you might expect. It adds the user name, e-mail address, salt, and password hash to the database. The only real point of interest in this class is `GetRowsetProperties`, which is called by the attribute-injected code before the query is executed. The property that you add here is required because this query writes to the database. The previous query was read-only and did not need this property.

```
class CCommandCreateUser : public CCreateUser
{
public:
   CCommandCreateUser();
   void Clear();
   HRESULT SetUserName(LPCSTR szUserName);
   HRESULT SetPassword(LPCSTR szPassword);
   HRESULT SetEmail(LPCSTR szEmail);
};
```

The wrapper class for `CCreateUser` is extremely simple. It includes the constructor and the `Clear` method for initializing the data members and methods for setting the data members. `SetPassword` does not correspond exactly to a data member; instead, it accepts a plain text password and generates a salt that it stores in `m_Salt` and a salted hash of the password that it stores in `m_Password`.

10. Add the implementation code for these classes to Commands.cpp:

```
#include "stdafx.h"
#include "Commands.h"
#include "Encrypt.h"
#include "Helpers.h"

// Initialization.
CCommandSiteUser::CCommandSiteUser()
{
   Clear();
}

// Initialization.
void CCommandSiteUser::Clear()
{
   m_dwPasswordStatus = DBSTATUS_S_ISNULL;
   m_dwSaltStatus = DBSTATUS_S_ISNULL;
   m_dwUserNameStatus = DBSTATUS_S_ISNULL;
```

```
   m_dwPasswordLength = 0;
   m_dwSaltLength = 0;
   m_dwUserNameLength = 0;

   m_Password[0] = 0;
   m_Salt[0] = 0;
   m_UserName[0] = 0;
}

HRESULT CCommandSiteUser::SetUserName(LPCSTR szUserName)
{
   return VCUE::SetOleDbStringMember(m_UserName, m_dwUserNameStatus,
                                     m_dwUserNameLength, szUserName);
}

HRESULT CCommandSiteUser::ComparePassword(LPCSTR szPassword)
{
   ATLASSERT(szPassword != NULL);

   HRESULT hr = E_FAIL;

   if (
      (m_dwSaltStatus == DBSTATUS_S_OK) &&
      (m_dwPasswordStatus == DBSTATUS_S_OK) &&
      (m_dwSaltLength != 0) &&
      (m_dwPasswordLength != 0)
      )
   {
      #pragma warning(push)
      // conversion from 'size_t' to 'DWORD', possible loss of data
      #pragma warning(disable : 4267)

      hr = VCUE::CompareSecret(
               reinterpret_cast<const BYTE*>(szPassword), strlen(szPassword),
               m_Salt, m_dwSaltLength,
               m_Password, m_dwPasswordLength);

      #pragma warning(pop)
   }

   return hr;
}
```

The code for `CCommandSiteUser` just uses the functions developed earlier in the tutorial. `CCommandSiteUser::ComparePassword` can be called after the query has been successfully executed. Like `CompareSecret`, it returns S_OK if the password matches the salted hash found in the database, S_FALSE if the password does not match, and an error HRESULT if a failure occurs.

```
HRESULT CCommandCreateUser::SetUserName(LPCSTR szUserName)
{
   return VCUE::SetOleDbStringMember(m_UserName, m_dwUserNameStatus,
                                     m_dwUserNameLength, szUserName);
}

HRESULT CCommandCreateUser::SetEmail(LPCSTR szEmail)
{
   return VCUE::SetOleDbStringMember(m_Email, m_dwEmailStatus,
                                      m_dwEmailLength, szEmail);
```

```
}

HRESULT CCommandCreateUser::SetPassword(LPCSTR szPassword)
{
   ATLASSERT(szPassword != NULL);

   HRESULT hr = E_FAIL;
   m_dwPasswordLength = sizeof(m_Password);
   m_dwSaltLength = sizeof(m_Salt);

   #pragma warning(push)
   // conversion from 'size_t' to 'DWORD', possible loss of data
   #pragma warning(disable : 4267)

   hr = VCUE::HashSecret(reinterpret_cast<const BYTE*>(szPassword),
                         strlen(szPassword),
            m_Salt, m_dwSaltLength,
            m_Password, m_dwPasswordLength);

   #pragma warning(pop)

   if (FAILED(hr))
   {
      m_dwPasswordLength = 0;
      m_dwSaltLength = 0;

      m_dwPasswordStatus = DBSTATUS_S_ISNULL;
      m_dwSaltStatus = DBSTATUS_S_ISNULL;
   }
   else
   {
      m_dwPasswordStatus = DBSTATUS_S_OK;
      m_dwSaltStatus = DBSTATUS_S_OK;
   }

   ATLASSERT(m_dwPasswordLength <= sizeof(m_Password));
   ATLASSERT(m_dwSaltLength <= sizeof(m_Salt));
   ATLASSERT( (FAILED(hr) && (0 == m_dwPasswordLength) && (DBSTATUS_S_ISNULL == m_dwPa
sswordStatus)) || (SUCCEEDED(hr) && (0 != m_dwPasswordLength) && (DBSTATUS_S_OK == m_d
wPasswordStatus)) );
   ATLASSERT( (FAILED(hr) && (0 == m_dwSaltLength) && (DBSTATUS_S_ISNULL == m_dwSaltSt
atus)) || (SUCCEEDED(hr) && (0 != m_dwSaltLength) && (DBSTATUS_S_OK == m_dwSaltStatus)
) );

   return hr;
}

// Initialization.
CCommandCreateUser::CCommandCreateUser()
{
   Clear();
}

// Initialization.
void CCommandCreateUser::Clear()
{
   m_dwPasswordStatus = DBSTATUS_S_ISNULL;
   m_dwSaltStatus = DBSTATUS_S_ISNULL;
   m_dwUserNameStatus = DBSTATUS_S_ISNULL;
```

```
    m_dwEmailStatus = DBSTATUS_S_ISNULL;

    m_dwPasswordLength = 0;
    m_dwSaltLength = 0;
    m_dwUserNameLength = 0;
    m_dwEmailLength = 0;

    m_Password[0] = 0;
    m_Salt[0] = 0;
    m_UserName[0] = 0;
    m_Email[0] = 0;
}
```

Next

Creating an Error Response

See Also

Reference

ATL Server Tutorial

Creating an Error Response

Now that you have created the OLE DB consumer classes, you have code that is complicated enough to fail. Before adding more code to the request handler that allows the user to log in, you should deal with errors and communicate the status of a request to the client.

First, status codes need to be communicated to several different entities:

- ATL Server.

  ATL Server needs to know when an error occurs so that it can log the error (if request handling statistics are enabled), trigger the creation of an error response (optionally), and stop further processing of a request. You typically inform ATL Server that an error has occurred by returning an HTTP_CODE indicating the error from **ValidateAndExchange** or from a replacement method.

- Internet Information Services (IIS).

  IIS needs to know the status of a request so that it can log the information for later analysis and so that it can pass that information on in the header of the HTTP response. You typically inform IIS and other downstream objects that an error has occurred by calling CHttpResponse::SetStatusCode.

- Caching proxies.

  Caching proxies, routers, and other servers between the Web site and the client may make use of the status code returned in the header of the HTTP response to improve response times, spot unusual behavior, or provide other features that improve the performance of the Internet. In addition, some specific HTTP headers allow you to provide information that controls the caching of your response. Typically, you will not want error responses to be cached.

- The client's browser.

  Browsers typically provide caching features that require access to accurate status codes in the HTTP response header and may use the cache-specific HTTP headers. Some browsers, such as Internet Explorer, are also capable of displaying helpful detailed error messages based on the HTTP status code.

- The user of the Web site.

  The user of the Web site needs to know that an error has occurred and whether there is anything he or she can do about it. The information available to the user is set in the body of the HTTP response.

Second, an error can occur at any time. It is possible that you may have already written data to the response stream or added HTTP headers by the time an error occurs. Typically, this is not a problem because HTTP responses generated by ATL Server request handlers are buffered by default. This means that you can still send the error headers and response body when an error occurs without having parts of the success response that you would have liked to send mixed in with it. You just need to clear the response before adding your error information.

To communicate an error to all interested parties, you need to:

- Clear any response that may be currently buffered.
- Set cache-specific HTTP headers to disable caching of the error response.
- Call CHttpResponse::SetStatusCode so that IIS and the user's browser know that an error has occurred.
- Set the response body with a human-readable error message so that the user knows that an error has occurred.
- Return an HTTP_CODE from **ValidateAndExchange** or from a replacement method so that the ATL Server code knows how to proceed.

This code can be wrapped in a single function.

To enable error communication

- Add the following code to Helpers.h in the VCUE namespace:

```
// Call this function to return a simple error response to the user.
// The HTTP status code defaults to 500 (a generic server error).
```

```
inline HTTP_CODE SendError(CHttpResponse& response,
                           const CStringA& strError,
                           WORD wHttpStatus = 500)
{
    // Clear any buffered headers (including cookies) and content.
    response.ClearResponse();

    // Suggest that clients and proxies do not cache this response.
    response.SetCacheControl("no-cache");
    response.SetExpires(0);

    // Set the status code in the response object.
    response.SetStatusCode(wHttpStatus);

    // Build the body of the response.
    response <<
        "<html><head><title>ATL Server Tutorial</title></head><body><p>"
        << strError << "</p></body></html>";

    // Return a HTTP_CODE that tells the ATL Server code to
    // discontinue processing of the SRF file.
    return AtlsHttpError(wHttpStatus, SUBERR_NO_PROCESS);
}
```

The call to CHttpResponse::ClearResponse removes any headers and body that have been written to the response stream up to this point.

The call to CHttpResponse::SetCacheControl tells proxies not to cache the response and the call to CHttpResponse::SetExpires has a similar effect on anyone else that receives the response. The value passed is a relative expiration time in minutes. A value of zero means that the response is only valid now; it is not valid at any time in the future.

The response body is a simple HTML message using the string passed to the function to describe the error.

The value returned by this function is designed to be used as the return value of `ValidateAndExchange` or a replacement method. The function uses AtlsHttpError to build an `HTTP_CODE` value that includes the specified HTTP status code and an HTTP_CODE subcode (`SUBERR_NO_PROCESS`) that tells the ATL Server status code to discontinue processing of the response.

Next

Authenticating the User

See Also

Reference

ATL Server Tutorial

ShowErrors

Concepts

Error Handling (ATL Server)

Authenticating the User

In this step, you will write the code to check whether the user has supplied a user name and password that matches information in the database. In the following step, you will add the code that attempts to add the user to the database if no user with that name exists. In the step following that, you will see how to create a new cookie and session if the user was successfully authenticated.

To add code checking the user name and password

1. Add the following code to Login.cpp after `#include "Login.h"`:

```
#include "Commands.h"
#include "Helpers.h"
#include "Encrypt.h"
```

2. Add the following code to Login.cpp after the form variables have been validated, in place of the comment `// TODO - Verify password and redirect to main page`):

```
        using VCUE::OpenCommandRowset;
        using VCUE::SendError;

        // Get the user name and password from the form fields.
        LPCSTR szUser = FormFields.Lookup("username");
        LPCSTR szPassword = FormFields.Lookup("password");

        // Create a command object to query the database for this
        // username and password.
        CCommandSiteUser cmdSiteUser;

        HRESULT hr = E_UNEXPECTED;

        hr = cmdSiteUser.SetUserName(szUser);
        if (FAILED(hr))
            return SendError(m_HttpResponse, "An unexpected error occurred.");

        // Open or retrieve the cached database connection and execute
        // the SQL query to retrieve any users with matching username
        hr = OpenCommandRowset(m_spServiceProvider, cmdSiteUser);

        // Move to the first row of the result set
        if (SUCCEEDED(hr))
        {
            hr = cmdSiteUser.MoveFirst();
            if (S_OK == hr)
            {
                hr = cmdSiteUser.ComparePassword(szPassword);
            }
            else if (SUCCEEDED(hr))
            {
                 // <A HREF="vcconatlservertutorialaddingauser.htm">TODO - Add the use
r in debug builds</A>
            }

        }

        if (FAILED(hr))
            return SendError(m_HttpResponse,
```

```
                                    "An error occurred accessing the database.");

        if (hr != S_OK)
        {
            // The user has supplied an invalid combination of
            // user name and password.
            if (0 == m_strStatus.GetLength())
                m_strStatus = "Invalid username or password, please try again";
        }
        else
        {
            // The user has provided a valid combination of
            // user name and password.
            // <A HREF="vcconatlservertutorialcreatingausersession.htm">TODO - Create
a cookie/session and redirect to main page</A>
        }
```

The code obtains the value of the **username** and **password** form variables that were successfully validated in the earlier code, creates an instance of the database command class, `CCommandSiteUser`, that holds the query for obtaining user information from the database, and calls `OpenCommandRowset` to execute the query.

If the query executes successfully, the code attempts to move to the first record of the result set. If MoveFirst does not return S_OK, no records were returned and you can conclude that the user name supplied is not currently being used. In that situation, control is given to the code that adds the user to the database, which you will write in the next step.

If information can be obtained for the specified user, the password is compared, using `CCommandSiteUser::ComparePassword`, with the salted hash found in the database. If this method returns S_OK, the correct password was supplied and control is given to the code that you will write in Creating a User Session. That code will create the user session and cookie and redirect the user to the main page of the store. If the method returns S_FALSE, the password was not correct, and the user will see that information on the login page.

If an error occurs anywhere in this code, the function exits immediately, returning the value returned by `SendError`. The user will see a simple HTTP response describing the error.

Note
You are only interested in the first record returned by the query because the constraints in the database ensure that the re can be at most one user with a particular user name.

Next

Adding a User

See Also

Reference

ATL Server Tutorial

Adding a User

In this step, you will write the code to add a user to the database. This code will be called automatically if the user name supplied using the login page does not exist in the database. It is only enabled in debug builds to keep the code relatively simple. Mixing the login and account signup pages in this way is not the most usable design.

The code will:

- Create an instance of the `CCommandCreateUser` database command class.
- Set the user's name, password, and e-mail address to be used by the command object.
- Execute the query that adds this information to the database.

To add code to add a user to the database

- Add this code to **Login.cpp** in place of the comment `// TODO - Add the user in debug builds`:

```
#ifdef DEBUG
// This code adds the user to the database
LPCSTR szEmail = FormFields.Lookup("email");
CCommandCreateUser cmdCreateUser;
hr = cmdCreateUser.SetUserName(szUser);
if (SUCCEEDED(hr))
{
    hr = cmdCreateUser.SetPassword(szPassword);
    if (SUCCEEDED(hr))
    {
        hr = cmdCreateUser.SetEmail(szEmail);
        if (SUCCEEDED(hr))
        {
            hr = OpenCommandRowset(m_spServiceProvider,
                                   cmdCreateUser);
        }
    }
}

if (SUCCEEDED(hr))
{
    m_strStatus =
      "The user was added to the database. You can now login.";
}
else
{
    m_strStatus = "Failed to add user.";
}

hr = S_FALSE;
#endif
```

Two factors prevent duplicate users being added to the database:

- This code will not be executed if the database already has a user of the specified name.
- The database itself has a constraint on the **UserName** column of the **SiteUser** table that ensures that each **UserName** must be unique.

Note

The only validation on the e-mail address at this point is the length check done by `CCommandCreateUser::SetEmail`. This code is only intended as a simple way for you to add users to the database, not as a complete solution; you should perform more extensive validation in your own code.

Next

Creating a User Session

See Also

Reference

ATL Server Tutorial

Creating a User Session

In this step, you are going to write the code that is executed when the supplied user name and password are found to match a user in the database. In this situation, you will create a new session object in which you will store the user's name and, later, the items in the user's shopping cart. You will also write the randomly generated session identifier to a cookie that will be sent to the client as part of the response headers. On subsequent requests, the user's browser will pass the cookie back to the Web site so that it can be used to identify the session.

Note
Although the user is associated with their session information using a cookie in this example, it is also possible to embed the session ID in the URL for users that have disabled cookie support. The implementation of this type of system is not examined here.

Because other request handlers in the site will need to access the session cookie, you need to add some definitions to StdAfx.h that will allow the session cookie to be located:

To add definitions to StdAfx.h

- In Solution Explorer, double-click StdAfx.h and add the following code:

```
// Definitions for cookie support
#define COOKIE_NAME "AtlServerTutorial"
#define COOKIE_VALUE_NAME "SessionId"
#define COOKIE_VALUE_SIZE 64
```

  `COOKIE_NAME` is the name of the cookie that holds the session identifier (and may also hold other information). `COOKIE_VALUE_NAME` is the name of the session identifier within the `COOKIE_NAME` cookie. `COOKIE_VALUE_SIZE` is the maximum size of the session identifier.

Now you will add a simple function to Helpers.h that will simplify the process of obtaining services from the IServiceProvider interface. This interface is implemented by the CIsapiExtension-derived class and allows access to functionality, such as session state services, exposed by the ISAPI extension.

IServiceProvider::QueryService takes three parameters: a GUID identifying the service, an IID identifying an interface provided by that service, and the address of a pointer to receive the requested interface. A template overload reduces the parameter list by obtaining the IID from the type of pointer passed by using the __uuidof operator. Because ATL Server typically uses the same ID for the service GUID and the interface ID, you will add a function that uses the same technique to get the GUID identifying the service.

To add a simple function to Helpers.h

- In Solution Explorer, double-click Helpers.h and add the following code in the `VCUE` namespace:

```
// A function template that wraps IServiceProvider::QueryService and sets the
// GUID of the service to be the same as the GUID of the interface.
template <class T>
inline HRESULT QueryService(IServiceProvider* pServiceProvider, T** ppService)
{
    return pServiceProvider->QueryService(__uuidof(T), ppService);
}
```

To add code to Login.cpp

- In Solution Explorer, double-click Login.cpp and replace the text `// TODO - Create a cookie/session and redirect to main page`: with the following code:

```
            // The user has provided a valid combination of
            // username and password.
            using VCUE::QueryService;

            // Get a pointer to the session service.
            CComPtr<ISessionStateService> spSessionService;
            if (FAILED(QueryService(m_spServiceProvider, &spSessionService)))
                return SendError(m_HttpResponse,
                                 "Failed to obtain the session service.");

            // Create a new session.
            CHAR szSessionId[COOKIE_VALUE_SIZE + 1];
            szSessionId[0] = 0;
            DWORD dwSize = (sizeof(szSessionId) / sizeof(szSessionId[0])) - 1;

            CComPtr<ISession> spSession;
            hr = spSessionService->CreateNewSession(szSessionId, &dwSize,
                                                    &spSession);

            if (FAILED(hr))
            {
                return SendError(m_HttpResponse,
                           "An error occurred creating a new session object.");
            }

            // Put the session ID in a cookie.
            CCookie cookieStore;
            cookieStore.SetName(COOKIE_NAME);
            BOOL bSuccess = cookieStore.AddValue(COOKIE_VALUE_NAME,
                                                 szSessionId);
            if (bSuccess)
                bSuccess = m_HttpResponse.AppendCookie(&cookieStore);

            if (!bSuccess)
            {
                return SendError(m_HttpResponse,
                            "An error occurred creating the cookie.");
            }

            // Store the username in session state
            CComVariant varUserName = szUser;
            hr = spSession->SetVariable("UserName", varUserName);
            if (FAILED(hr))
            {
                return SendError(m_HttpResponse,
                        "An error occurred adding a variable to the session.");
            }

            // Redirect to the main page.
            m_HttpResponse.Redirect("tutorial.srf");

            // Discontinue processing of the SRF file.
            return HTTP_SUCCESS_NO_PROCESS;
```

The code first obtains a pointer to the ISessionStateService interface provided by the ISAPI extension. The interface is obtained from the IRequestHandlerImpl::m_spServiceProvider member.

Once the session-state service interface has been obtained, you call ISessionStateService::CreateNewSession to create a

new session and obtain its ID. The ID is a randomly generated sequence of characters whose size is specified by the caller.

Note
For another way to generate IDs used to identify a user's session, see the ForceLogin section of the WebFeatures Sample.

When the session has been successfully created, the session ID is written to a CCookie object and the user's name is added to the session object using ISession::SetVariable.

Finally, the user is redirected to the main page of the site using CHttpResponse::Redirect and further processing of the page is prevented by returning HTTP_SUCCESS_NO_PROCESS.

Next

Creating the Online Store

See Also
Reference
ATL Server Tutorial

Creating the Online Store

The main page of the store will display the products in the database, allow the user to add items to their shopping cart, and provide a link that allows the user to purchase the items in their cart. Because the number of products carried by the store is small, all products will be displayed on a single page, the number of items in the user's cart will be displayed along with the product information, and the shopping cart information will be held in the user's session object. This page, like all the remaining pages in the site, will automatically redirect the user to the login page if they have not already logged in.

Because this is the main page of the site, you will adjust the wizard-generated server response file and request handler to fit your needs.

Procedures

To change the main page of your site

- In Solution Explorer, double-click Tutorial.srf and replace the existing code with the following code:

```
<html>
   {{handler bin\tutorial.dll/Default}}
   <head>
   </head>
   <body>
      <table border="2" width="640">
         <tr>
            <td align="center">
               <b>Name</b>
            </td>
            <td align="center">
               <b>Description</b>
            </td>
            <td align="center">
               <b>Price</b>
            </td>
            <td align="center">
               <b>Quantity</b>
            </td>
            <td align="center">
               <b>Purchase</b>
            </td>
         </tr>
         {{while GetNextProduct}}
         <tr>
            <td>
               {{ProductName}}
            </td>
            <td>
               {{ProductDescription}}
            </td>
            <td>
               <center>
                  {{ProductPrice}}
               </center>
            </td>
            <form action="tutorial.srf" method="post">
               <td align="center">
                  <input type="text" size="4" name="Quantity"
                   value="{{ProductQuantity}}" align="center">
                  ({{ProductQuantity}})
```

```
                </td>
                <td align="center">
                   <input type="submit" name="AddToCart"
                    value="Set Quantity In Cart" align="center">
                </td>
                <input type="hidden" name="ProductId" value="{{ProductId}}">
             </form>
          </tr>
          {{endwhile}}
       </table>
       <form action="cart.srf" method="post">
          <input type="submit" value="Checkout" name="Checkout" align="left">
       </form>
    </body>
 </html>
```

The server response file builds an HTML table in a **while** loop. Each iteration through the loop produces the row for a different product. The product ID, name, description, and price will all be obtained from the database. The quantity (the number of items in the user's shopping cart) will be obtained from the user's session object.

Note that the product ID is written to a hidden input element. This element will be submitted as a form variable along with the quantity of a particular item that the user wants in their cart, so that you can quickly and uniquely identify the product that is being requested. However, this information will not be displayed to the user, because it does not enhance their use of the site.

Note

Hidden input elements are a useful way of maintaining state across multiple requests. If your whole site consists of forms that are posted to each other, you can even use a hidden input element to hold the session ID. You should be careful not to write large or sensitive data items as hidden input elements; you should use the same restraint that you show when writing cookie data. Take steps to prevent such pages being cached, and consider what happens if the user navigates directly to a page.

This page features one form for each product, meaning that only one product can have its quantity changed at a time. Each form will be posted back to Tutorial.srf, where the request handler will change the number of items in the user's cart based on the specified product ID and quantity then redisplay the page using the new values. A separate form is provided to allow the user to proceed to the checkout page, Cart.srf.

Next

Getting the Product Information

See Also

Reference

ATL Server Tutorial

Getting the Product Information

The next step in implementing the store's main page is to create the database command classes that allow you to retrieve product information from the database. The information about the products is held in the `Stock` table in the `Id`, `Name`, `Description` and `Price` columns. The SQL query will just request all the items in the table because the table holds only a few products.

To create the database command classes

- In Solution Explorer, double-click Commands.h and add the following code:

```
[
    db_command(
        "SELECT [Stock].[Name], "
        "[Stock].[Description], "
        "[Stock].[Price], "
        "[Stock].[Id] "
        "FROM [Stock]"
    )
]
class CGetStock
{
public:
    [ db_column(1, status = m_dwNameStatus,
      length = m_dwNameLength) ]
    char m_Name[51];

    [ db_column(2, status = m_dwDescriptionStatus,
      length = m_dwDescriptionLength) ]
    char m_Description[300];

    [db_column(3, status = m_dwPriceStatus)]
    long m_Price;

    [db_column(4, status = m_dwIdStatus)]
    long m_Id;

    DBSTATUS m_dwNameStatus;
    DBSTATUS m_dwDescriptionStatus;
    DBSTATUS m_dwPriceStatus;
    DBSTATUS m_dwIdStatus;

    DBLENGTH m_dwNameLength;
    DBLENGTH m_dwDescriptionLength;
};

class CCommandGetStock : public CGetStock
{
public:
   CCommandGetStock();
   void Clear();

   bool AllMembersOk();
};
```

There are two classes for this query: the base **db_command** class, and the derived helper class that you will use in our code. Apart from the initialization helpers, the constructor, and `Clear`, the helper class provides a method called

`AllMembersOk` that checks the status codes for all the columns.

To add the implementation code for CCommandGetStock

- In Solution Explorer, double-click Commands.cpp and add the following code:

```
// Initialization.
CCommandGetStock::CCommandGetStock()
{
   Clear();
}

// Initialization.
void CCommandGetStock::Clear()
{
   m_dwNameStatus = DBSTATUS_S_ISNULL;
   m_dwDescriptionStatus = DBSTATUS_S_ISNULL;
   m_dwPriceStatus = DBSTATUS_S_ISNULL;
   m_dwIdStatus = DBSTATUS_S_ISNULL;

   m_dwNameLength = 0;
   m_dwDescriptionLength = 0;

   m_Name[0] = _T('\0');
   m_Description[0] = _T('\0');
   m_Price = 0;
   m_Id = 0;
}

// Returns true if all status members are equal to DBSTATUS_S_OK.
// Returns false otherwise.
bool CCommandGetStock::AllMembersOk()
{
   if (m_dwNameStatus != DBSTATUS_S_OK)
      return false;

   if (m_dwDescriptionStatus != DBSTATUS_S_OK)
      return false;

   if (m_dwPriceStatus != DBSTATUS_S_OK)
      return false;

   if (m_dwIdStatus != DBSTATUS_S_OK)
      return false;

   return true;
}
```

Next

Displaying the Product Information

See Also

Reference

ATL Server Tutorial

Displaying the Product Information

The request handler for the main page of the site already exists in Tutorial.h and Tutorial.cpp. The handler needs:

- An implementation of **ValidateAndExchange**.
- Replacement methods for each tag used in Tutorial.srf.
- A method that will handle adding items to the shopping cart.
- Data members to hold items that will be used from multiple replacement methods (specifically, an ISession pointer and a `CCommandGetStock` object).

To update the code in your header files

1. In Solution Explorer, double-click Tutorial.h and replace the existing code with the following code:

```
#pragma once
#include "Commands.h"

[ request_handler("Default") ]
class CTutorialHandler
{
public :
   // The command object used to retrieve information about
   // the products from the database. The members contain
   // information about the current product.
   CCommandGetStock m_cmdGetStock;

   // The session containing information about the current user.
   CComPtr<ISession> m_spSession;

   HTTP_CODE ValidateAndExchange();

   HRESULT AddToCart();

   [tag_name(name="GetNextProduct")]
   HTTP_CODE OnGetNextProduct();

   [tag_name(name="ProductName")]
   HTTP_CODE OnProductName();

   [tag_name(name="ProductDescription")]
   HTTP_CODE OnProductDescription();

   [tag_name(name="ProductPrice")]
   HTTP_CODE OnProductPrice();

   [tag_name(name="ProductQuantity")]
   HTTP_CODE OnProductQuantity();

   [tag_name(name="ProductId")]
   HTTP_CODE OnProductId();

}; // class CTutorialHandler
```

2. Before providing the implementation of the request handler, add the following code to Helpers.h in the `VCUE` namespace:

```
inline LPCSTR GetLoginId(CHttpRequest& request)
{
    // Check for existence of cookie
    if (!request.Cookies(COOKIE_NAME).IsEmpty())
    {
        const CCookie& cookieValidate = request.Cookies(COOKIE_NAME);

        // Check for presence of cookie value
        LPCSTR szSessionId = cookieValidate.Lookup(COOKIE_VALUE_NAME);
        if (szSessionId)
        {
            // Check length of cookie value
            if (strlen(szSessionId) <= COOKIE_VALUE_SIZE)
            {
                return szSessionId;
            }
        }
    }

    return NULL;
}

inline HRESULT GetSession(IServiceProvider* pServiceProvider,
                          LPCSTR szSessionId, ISession** ppSession)
{
    HRESULT hr = E_UNEXPECTED;

    // Get the session state service.
    CComPtr<ISessionStateService> spSessionService;
    hr = QueryService(pServiceProvider, &spSessionService);

    if (SUCCEEDED(hr))
    {
        // Get the session.
        hr = spSessionService->GetSession(szSessionId, ppSession);
    }

    return hr;
}

inline bool ItemIsPresent(const CHttpRequestParams& Map, LPCSTR Item)
{
    const CHttpRequestParams::BaseMap& baseMap = Map;
    const CHttpRequestParams::CPair* pPair = baseMap.Lookup(Item);
    return pPair ? true : false;
}
```

These functions will be used from multiple request handlers in this tutorial.

The first function, `GetLoginId`, obtains the user's session ID from the cookie if present; otherwise, it returns `NULL`. If a call to this function returns `NULL`, you will redirect them to the login page ensuring that the user has logged in before any other page can be viewed.

Note

The implementation of `GetLoginId` is not sufficient for a real online store. In a more complete solution, the contents of the cookie would be checked for validity and care would be taken to expire sessions every so often to help avoid security problems.

The second function, `GetSession`, makes it possible to get an ISession pointer directly from the service provider. The ISessionStateService interface that has to be acquired first is rarely used after the session has been obtained, so this function simplifies the process of getting a pointer to the user's session object.

The third function, `ItemIsPresent`, provides a simple and efficient way of determining whether an item can be found in a CHttpRequestParams collection. This class is used to manage collections of form variables and query parameters.

3. Change the code in Tutorial.cpp to match this code:

```
#include "Helpers.h"

HTTP_CODE CTutorialHandler::ValidateAndExchange()
{
   using VCUE::OpenCommandRowset;
   using VCUE::SendError;
   using VCUE::GetSession;
   using VCUE::GetLoginId;
   using VCUE::ItemIsPresent;

   // Set the content-type of the response.
   m_HttpResponse.SetContentType("text/html");

   // Get the session ID of the current user.
   LPCSTR szSessionId = GetLoginId(m_HttpRequest);
   if (!szSessionId)
   {
      // User doesn't have valid cookie, redirect to login page.
      m_HttpResponse.Redirect("login.srf");
      return HTTP_SUCCESS_NO_PROCESS;
   }

   // Get the session corresponding to this ID.
   HRESULT hr = GetSession(m_spServiceProvider, szSessionId, &m_spSession);
   if (FAILED(hr))
   {
      return SendError(m_HttpResponse,
                  "An error occurred while obtaining the session object.");
   }

   // Get our form data
   const CHttpRequestParams& FormFields = m_HttpRequest.GetFormVars();
   if (FormFields.GetCount())
   {
      hr = E_UNEXPECTED;

      if (ItemIsPresent(FormFields, "AddToCart"))
         hr = AddToCart();
   }

   // Start to get the product information from the database.
   hr = OpenCommandRowset(m_spServiceProvider, m_cmdGetStock);
   if (FAILED(hr))
   {
      return SendError(m_HttpResponse,
                     "An error occurred accessing the database.");
   }

    return HTTP_SUCCESS;
}
```

The code in `ValidateAndExchange` checks for the presence of the cookie containing the session ID. If not present, the user is redirected to the login page and further processing of the response is prevented by returning HTTP_SUCCESS_NO_PROCESS.

If the session ID can be obtained, a pointer to the session object is stored in `m_spSession` for later use.

The form variables collection is examined for the presence of the **AddToCart** variable that indicates whether the quantity of items in the cart is being changed. If found, the quantity of items is changed.

Finally, the query that obtains the items from the database is executed in preparation for the calls to the replacement methods that will loop through the result set or write information about the current record to the output stream.

```
HTTP_CODE CTutorialHandler::OnGetNextProduct()
{
   using VCUE::SendError;

   // Move to the next row in the result set.
   HRESULT hr = m_cmdGetStock.MoveNext();

   if (FAILED(hr) || (!m_cmdGetStock.AllMembersOk()))
   {
      return SendError(m_HttpResponse,
                       "An error occurred while accessing the database.");
   }

   // HTTP_SUCCESS continues the loop
   // HTTP_S_FALSE terminates the loop
   return (hr == S_OK) ? HTTP_SUCCESS : HTTP_S_FALSE;
}
```

`OnGetNextProduct`, which is used in a **while** tag in the server response file, does little more than call MoveNext to move through the records in the result set produced by the query executed in `ValidateAndExchange`. `OnGetNextProduct` returns HTTP_SUCCESS if there is a current record or HTTP_S_FALSE if there are no more records.

```
HTTP_CODE CTutorialHandler::OnProductName()
{
   m_HttpResponse << m_cmdGetStock.m_Name;
   return HTTP_SUCCESS;
}

HTTP_CODE CTutorialHandler::OnProductDescription()
{
   m_HttpResponse << m_cmdGetStock.m_Description;
   return HTTP_SUCCESS;
}

HTTP_CODE CTutorialHandler::OnProductPrice()
{
   m_HttpResponse << "$" << m_cmdGetStock.m_Price;
   return HTTP_SUCCESS;
}

HTTP_CODE CTutorialHandler::OnProductId()
{
   m_HttpResponse << m_cmdGetStock.m_Id;
   return HTTP_SUCCESS;
}
```

`OnProductName`, `OnProductDescription`, `OnProductPrice`, and `OnProductId` write the information obtained from the

database to the response stream. Note that this means that the information in the database must be suitable for writing directly to an HTML page.

```
HTTP_CODE CTutorialHandler::OnProductQuantity()
{
   // The quantity of the product in the user's shopping cart
   // is retrieved from session state
   // using the product ID set by a prior call to OnGetNextProduct.

   long lQuantity = 0;

   // Check whether the session contains a purchase request for this product.
   // The name of the session variable is the ID of the product.
   CStringA strProductId;
   strProductId.Format("%ld", m_cmdGetStock.m_Id);

   CComVariant varQuantity;
   HRESULT hr = m_spSession->GetVariable(strProductId, &varQuantity);
   if (S_OK == hr)
   {
      // Expect variables to be stored as long (VT_I4)
      if (V_VT(&varQuantity) == VT_I4)
         lQuantity = V_I4(&varQuantity);
   }

   m_HttpResponse << lQuantity;
   return HTTP_SUCCESS;
}
```

`OnProductQuantity` obtains the quantity of the current item that is in the user's shopping cart and writes the result to the response stream. You will see how to add items to the shopping cart in the next step, but here you can see that the shopping cart is simply the collection of session variables whose name is the ID of a product and whose value is the quantity selected by the user stored as a `VARIANT` of type `long` (`VT_I4`).

Next

Adding Items to the Shopping Cart

See Also

Reference

ATL Server Tutorial

Adding Items to the Shopping Cart

In this step, you will implement the code to add items to the user's shopping cart, which you will implement as a collection of session variables whose name is the ID of a product and whose value is the quantity selected by the user stored as a `VARIANT` of type `long` (`VT_I4`). You will also see how to validate form data using regular expressions.

Because you need to use the regular expression classes from several places, you will need to change some of your code.

Procedures

To change code to add items to the user's shopping cart

1. In Solution Explorer, double-click StdAfx.h and include the header and the following typedefs:

```
#include <atlrx.h>
// Need to use single-byte regular expression classes
typedef CAtlRegExp<CAtlRECharTraitsA> CRegularExpression;
typedef CAtlREMatchContext<CAtlRECharTraitsA> CMatchContext;
```

   Here typedefs are provided for the regular expression class templates CAtlRegExp and CAtlREMatchContext specialized to use single-byte characters. The data that you will be validating will be submitted to the server as single-byte characters.

2. Double-click Tutorial.cpp and provide the implementation of `AddToCart`:

```
HRESULT CTutorialHandler::AddToCart()
{
   HRESULT hr = E_UNEXPECTED;
   const CHttpRequestParams& FormFields = m_HttpRequest.GetFormVars();

   LPCSTR szQuantity = FormFields.Lookup("Quantity");
   LPCSTR szProductId = FormFields.Lookup("ProductId");

   if (szQuantity && szProductId)
   {
      // Build a regular expression for matching decimal integers
      CRegularExpression regularExpression;
      if (regularExpression.Parse("^\\d+$") == REPARSE_ERROR_OK)
      {
         // Both product ID and quantity should be decimal integers
         CMatchContext matchContext;
         if (regularExpression.Match(szProductId, &matchContext) &&
             regularExpression.Match(szQuantity, &matchContext))
         {
            CComVariant varQuantity = szQuantity;
            // Ensure that quantities are stored as long (VT_I4) values.
            hr = varQuantity.ChangeType(VT_I4);
            if (SUCCEEDED(hr))
            {
               if (V_I4(&varQuantity))
               {
                  // If quantity is not zero, set the session variable.
                  hr = m_spSession->SetVariable(szProductId, varQuantity);
               }
               else
               {
                  // If the quantity is zero, remove the variable.
                  m_spSession->RemoveVariable(szProductId);
```

```
                }
            }
        }
    }
}

    return hr;
}
```

This method gets the product ID and quantity that have been posted to the server. The quantity indicates how many items of the specified product the user wants in their shopping cart.

The code builds a regular expression that can be used to match a string whose body consists entirely of one or more decimal integers. The translation of the regular expression `"^\\d+$"` is explained in the following table:

Element	Description
^	Start of string.
\d	Decimal integer. Note that C++ string encoding rules cause \\d to be translated to \d by the compiler.
+	One or more of the preceding item.
$	End of string.

The regular expression is used to check that the data sent to the server matches the expected format. If the data does match the expected format, the session variable named with the product ID and containing the quantity of items requested by the user is added to the user's session.

It is important to validate data at the server because nothing prevents clients from sending any random data that they care to send. Client-side validation can be used to enhance performance and usability of a site, but it is not a substitute for validation carried out on the server. See the Input Sample for a more detailed example of input validation.

Note
The regular expression syntax used by the ATL Server classes is similar but not identical to the syntax used by other regular expression-capable tools you may typically use, such as Perl or the .NET Framework classes. See CAtlRegExp Class for the regular expression syntax used by that class.

Next

Creating the Shopping Cart Page

See Also

Reference

ATL Server Tutorial

Creating the Shopping Cart Page

When the user chooses to go to the checkout from the main shopping page of the site, he or she is first taken to a page that shows the shopping cart. This page provides a summary of their order, including the total cost, and requires the user to confirm that they want to purchase the goods.

To create a server response file and add code to it

1. In Solution Explorer, right-click the Tutorial project.
2. On the shortcut menu, click **Add**, and then click **Add New Item**.

   The **Add New Item** dialog box appears.
3. In the Templates pane, click **Server Response File (.srf)**, and enter **Cart.srf** in the **Name** box.
4. Click **Open**.

   Visual Studio adds the file to disk and opens it in the HTML Designer.
5. Click the **HTML** tab at the bottom of the page.
6. Replace the existing code with the following code:

```
<html>
    {{handler bin\tutorial.dll/Cart}}
    <body>
        <b>Shopping Cart</b>
        <p>
            Your current order is:
        </p>
        <table>
            {{while GetNextPurchase}}
            <tr>
                <td>
                    {{ProductPurchaseSummary}}
                </td>
            </tr>
            {{endwhile}}
        </table>
        <br>
        Total = {{Total}}
        <form action="checkout.srf" method="post" name="Purchase">
            <input type="submit" name="{{Total}}" value="Proceed">
        </form>
    </body>
</html>
```

   The **Cart** handler will supply replacement methods for the `GetNextPurchase`, `ProductPurchaseSummary`, and `Total` tags.

Next

Displaying the Shopping Cart

See Also

Reference

ATL Server Tutorial

Displaying the Shopping Cart

In this step, you will implement the request handler that displays a summary of the products currently in the user's shopping cart. On this rare occasion, you do not need to create any database command classes because you can reuse the `CCommandGetStock` class created earlier.

To create a new .h file and add code to it

1. In Solution Explorer, right-click the Tutorial project.
2. On the shortcut menu, click **Add**, and then click **Add New Item**.

   The **Add New Item** dialog box appears.
3. In the Templates pane, click **Header File (.h)**, and enter **Cart.h** in the **Name** box.
4. Click **Open**.
5. Add the following code:

```
#pragma once
#include "Commands.h"

[ request_handler("Cart") ]
class CCartHandler
{
public:
    // Regular expression for validating product IDs
    // retrieved from session state
    CRegularExpression m_reProductIdValidator;

    // The total cost of the items in the cart.
    long m_lTotalCost;

    // The quantity of the current item.
    long m_lQuantity;

    // The ID of the current item.
    char m_szProductId[MAX_VARIABLE_NAME_LENGTH];

    // The command object used to retrieve information about
    // the products from the database. The members contain
    // information about the current product.
    CCommandGetStock m_cmdGetStock;

    // The session containing information about the current user.
    CComPtr<ISession> m_spSession;

    // The enumeration handle and current position of the purchase info
    HSESSIONENUM m_hSession;
    POSITION m_posSession;

    void Clear();

    HTTP_CODE ValidateAndExchange();

    [tag_name(name="GetNextPurchase")]
    HTTP_CODE OnGetNextPurchase();

    [tag_name(name="ProductPurchaseSummary")]
```

```
    HTTP_CODE OnProductPurchaseSummary();

    [tag_name(name="Total")]
    HTTP_CODE OnTotal();
};
```

To create a new .h file and add code to it

1. In Solution Explorer, right-click the Tutorial project.
2. On the shortcut menu, click **Add**, and then click **Add New Item**.

   The **Add New Item** dialog box appears.
3. In the Templates pane, click **CPP File (.cpp)**, and enter **Cart.cpp** in the **Name** box.
4. Click **Open**.
5. Add the following code:

```
#include "StdAfx.h"
#include "Cart.h"
#include "Helpers.h"

void CCartHandler::Clear()
{
    // Clear raw data members.
    m_lTotalCost = 0;
    m_lQuantity = 0;
    m_hSession = NULL;
    m_posSession = NULL;
    m_szProductId[0] = '\0';
}

HTTP_CODE CCartHandler::ValidateAndExchange()
{
    using VCUE::OpenCommandRowset;
    using VCUE::SendError;
    using VCUE::GetSession;
    using VCUE::GetLoginId;

    // Initialize members.
    Clear();

    // Set the content-type of the response.
    m_HttpResponse.SetContentType("text/html");

    // Get the session ID of the current user.
    LPCSTR szSessionId = GetLoginId(m_HttpRequest);
    if (!szSessionId)
    {
        // User doesn't have valid cookie, redirect to login page.
        m_HttpResponse.Redirect("login.srf");
        return HTTP_SUCCESS_NO_PROCESS;
    }

    // Get the session.
    HRESULT hr = GetSession(m_spServiceProvider, szSessionId, &m_spSession);
    if (FAILED(hr))
    {
        return SendError(m_HttpResponse,
```

```
                        "An error occurred while obtaining the session object.");
    }

    // Start to get the purchase information from session state.
    hr = m_spSession->BeginVariableEnum(&m_posSession, &m_hSession);
    if (FAILED(hr))
    {
        return SendError(m_HttpResponse,
          "An error occurred while preparing to enumerate session variables.");
    }

    // Build a regular expression for matching decimal integers
    if (m_reProductIdValidator.Parse("^\\d+$") != REPARSE_ERROR_OK)
    {
        return SendError(m_HttpResponse,
                        "An error occurred while parsing a regular expression.");
    }

    // Start to get the product information from the database.
    hr = OpenCommandRowset(m_spServiceProvider, m_cmdGetStock);
    if (FAILED(hr))
    {
        return SendError(m_HttpResponse,
                                  "An error occurred accessing the database.");
    }

    return HTTP_SUCCESS;
}
```

`ValidateAndExchange` checks for the presence of the cookie that indicates the user has logged in and redirects to the login page if not present.

Next, the code obtains the pointer to the user's session and calls ISession::BeginVariableEnum. The user's cart is stored in session state, so you need to prepare the request handler for a call to `OnGetNextPurchase`, which will loop through the items in the cart.

`m_reProductIdValidator` is a regular expression that matches decimal integers, just like the one you saw earlier. In this case, it will be used to match the names of the variables in the user's session to ensure that only items that could be purchases are considered.

Finally, the database command object `m_cmdGetStock` is executed to obtain information about the products. The shopping cart just stores the ID and quantity of the items that the user has chosen; the name and price of the item must be obtained from the database so that the information is displayed to the user.

```
HTTP_CODE CCartHandler::OnGetNextPurchase()
{
    using VCUE::SendError;

    // Iteration has finished if the POSITION is NULL.
    if (!m_posSession)
        return HTTP_S_FALSE;

    // Initialize data members.
    m_lQuantity = 0;
    m_szProductId[0] = 0;

    // Get the next variable from session state.
    CComVariant varQuantity;
    HRESULT hr = m_spSession->GetNextVariable(&m_posSession, &varQuantity,
                        m_hSession, m_szProductId, sizeof(m_szProductId));
```

```
    while (S_OK == hr)
    {
        // Check the variable name is a decimal integer using the
        // regular expression prepared earlier.
        CMatchContext matchContext;
        if (m_reProductIdValidator.Match(m_szProductId, &matchContext))
        {
            // Check that the variable value is a variant of type long (VT_I4).
            if (VT_I4 == V_VT(&varQuantity))
            {
                // Save the quantity.
                m_lQuantity = V_I4(&varQuantity);

                // Save the product ID.
                long lProductId = atol(m_szProductId);

                // Move to start of products
                hr = m_cmdGetStock.MoveFirst();

                while ((S_OK == hr) && m_cmdGetStock.AllMembersOk())
                {
                    // Check whether the current products match.
                    if (m_cmdGetStock.m_Id == lProductId)
                    {
                        // Found the product.
                        // Increase the total cost of the purchases.
                        m_lTotalCost += m_cmdGetStock.m_Price * m_lQuantity;
                        return HTTP_SUCCESS;
                    }

                    // Loop through remaining products until we find the
                    // product or the end of the result set is reached.
                    hr = m_cmdGetStock.MoveNext();
                }

                return SendError(m_HttpResponse, "An error occurred. "
            "Session variable does not correspond to product in the database.");
            }
        }

        // No more session variables.
        if (!m_posSession)
            return HTTP_S_FALSE;

        // The previous session variable wasn't a purchased item.
        // Get the next session variable and try again.
        varQuantity.Clear();
        hr = m_spSession->GetNextVariable(&m_posSession, &varQuantity,
                                m_hSession, m_szProductId, sizeof(m_szProductId));
    }

    return HTTP_S_FALSE;
}
```

`OnGetNextPurchase` calls ISession::GetNextVariable to obtain a session variable from the user's session. If the name of that session matches the regular expression for decimal integers and the value of the variable is a `long` variant, then it is an item from the shopping cart. In that case, the number of requested items is stored in `m_lQuantity`. The product ID is

converted to a numeric value for easy comparison with the IDs in the result set returned by the database query, loop through the records until it finds the one that it wants, and increments the total cost. The method returns HTTP_S_FALSE when no more session variables are found.

Note

In typical circumstances, this is not a very efficient way of obtaining information about the products. If the shopping car t was stored in the database, a single query could join the cart with the information from the products table and provid e us with all the information needed for this page. However, because an early design decision has the shopping cart sto red in session state and because the number of products in the database is small, this implementation is acceptable co mpared with making multiple queries against the database.

```
HTTP_CODE CCartHandler::OnProductPurchaseSummary()
{
    // Output the purchase summary of the current product.
    m_HttpResponse << m_lQuantity << " x " << m_cmdGetStock.m_Name <<
                                   " at $" << m_cmdGetStock.m_Price;
    return HTTP_SUCCESS;
}
```

The code for `OnProductPurchaseSummary` is extremely simple. It is known that `m_lQuantity` contains the number of items purchased and `m_cmdGetStock` is positioned at the product that was purchased allowing easy access to the name and price. All that remains is to write the data to the response stream.

```
HTTP_CODE CCartHandler::OnTotal()
{
    // Output the total cost.
    m_HttpResponse << "$" << m_lTotalCost;
    return HTTP_SUCCESS;
}
```

`OnTotal` is similarly bare. The total cost was calculated during the loop through the items in the cart, allowing the cost to be written directly to the response stream.

Next

Creating the Checkout Page

See Also

Reference

ATL Server Tutorial

Creating the Checkout Page

The final page of the store will be a confirmation page that informs the user that his or her order has been accepted. Although the page seen by the user will be extremely simple (the server response file will not even include any replacement tags), the code behind it will deal with committing the order to the database, sending the user an e-mail message confirming the order, and keeping a count of orders that can be viewed using the performance monitor.

To create a new .srf file and add code to it

1. In Solution Explorer, right-click the Tutorial project.
2. On the shortcut menu, click **Add**, and then click **Add New Item**.

   The **Add New Item** dialog box appears.
3. In the Templates pane, click **Server Response File (.srf)**, and enter **Checkout.srf** in the **Name** box.
4. Click **Open**.
5. Add the following code:

```
<html>
   {{handler bin\tutorial.dll/Checkout}}
   <body>
      <p>
         Your order has been processed and accepted. You should receive e-mail
         confirmation of your order soon.
      </p>
      <p>
         If you do not receive this confirmation please feel free to e-mail us.
      </p>
      <p>
         The ATL Server team appreciates your order. Your continuing support
         means a lot to us. Please feel free to continue your shopping at
        <a href="tutorial.srf">this location</a>.
      </p>
   </body>
</html>
```

Next

Committing the Order

See Also

Reference

ATL Server Tutorial

Committing the Order

To keep things simple, the `Order` table in the database just contains two columns, `UserName` and `Order`. You will dump the user's purchase in the `Order` column as a tab separated list of product IDs and quantities. A real online store is likely to use a more complicated system of tracking orders.

To add the database command to update the Order table

1. In Solution Explorer, double-click Commands.h and add the following code:

```
[
    db_command(
        "INSERT INTO [Order] "
        "( [UserName], [Order] ) "
        "VALUES(?, ?)"
    )
]
class CCreateOrder
{
public:
    [ db_param(1, status = m_dwUserNameStatus,
      length = m_dwUserNameLength) ]
    char m_UserName[51];

    [ db_param(2, status = m_dwOrderStatus,
      length = m_dwOrderLength) ]
    char m_Order[101];

    DBSTATUS m_dwUserNameStatus;
    DBSTATUS m_dwOrderStatus;

    DBLENGTH m_dwUserNameLength;
    DBLENGTH m_dwOrderLength;

    void GetRowsetProperties(CDBPropSet* pPropSet)
    {
        pPropSet->AddProperty(DBPROP_IRowsetChange, true);
    }
};

class CCommandCreateOrder : public CCreateOrder
{
public:
    CCommandCreateOrder();
    void Clear();
    HRESULT SetUserName(LPCSTR szUserName);
    HRESULT SetOrder(LPCSTR szOrder);
};
```

2. Double-click Commands.cpp and add the implementation of `CCommandCreateOrder`:

```
// Initialization.
CCommandCreateOrder::CCommandCreateOrder()
{
    Clear();
}
```

```
// Initialization.
void CCommandCreateOrder::Clear()
{
    m_dwUserNameStatus = DBSTATUS_S_ISNULL;
    m_dwOrderStatus = DBSTATUS_S_ISNULL;

    m_dwUserNameLength = 0;
    m_dwOrderLength = 0;

    m_UserName[0] = _T('\0');
    m_Order[0] = _T('\0');
}

HRESULT CCommandCreateOrder::SetUserName(LPCSTR szUserName)
{
    return VCUE::SetOleDbStringMember(m_UserName, m_dwUserNameStatus,
                                      m_dwUserNameLength, szUserName);
}

HRESULT CCommandCreateOrder::SetOrder(LPCSTR szOrder)
{
    return VCUE::SetOleDbStringMember(m_Order, m_dwOrderStatus,
                                      m_dwOrderLength, szOrder);
}
```

Next

Taking the Order

See Also

Reference

ATL Server Tutorial

Taking the Order

In this step, you will create a new header file and add the code for the request handler that will commit the user's order and send them a confirmation message.

Procedures

To create the new header file and add the code

1. In Solution Explorer, right-click the Tutorial project.
2. On the shortcut menu, click **Add**, and then click **Add New Item**.

   The **Add New Item** dialog box appears.
3. In the Templates pane, click **Header File (.h)**, and enter **Checkout.h** in the **Name** box.
4. Click **Open**.
5. Add the following code:

```
#pragma once

[ request_handler("Checkout") ]
class CCheckoutHandler
{
public:
    HTTP_CODE ValidateAndExchange();
    HRESULT CommitOrder(IServiceProvider* pServiceProvider, ISession* pSession,
                        const CStringA& strUserName);
};
```

   All the code will be called from `ValidateAndExchange` because there are no replacement methods. `CommitOrder` will be responsible for writing the order to the database.

To create the new source file and add the code

1. In Solution Explorer, right-click the Tutorial project.
2. On the shortcut menu, click **Add**, and then click **Add New Item**.

   The **Add New Item** dialog box appears.
3. In the Templates pane, click **CPP File (.cpp)**, and enter **Checkout.cpp** in the **Name** box.
4. Click **Open**.
5. Add the following code:

```
#include "StdAfx.h"
#include "Checkout.h"
#include "Commands.h"
#include "Helpers.h"

HTTP_CODE CCheckoutHandler::ValidateAndExchange()
{
    using VCUE::OpenCommandRowset;
    using VCUE::SendError;
    using VCUE::GetSession;
    using VCUE::GetLoginId;

    HRESULT hr = E_UNEXPECTED;
```

```
    // Set the content-type of the response.
    m_HttpResponse.SetContentType("text/html");

    // Get the session ID of the current user.
    LPCSTR szSessionId = GetLoginId(m_HttpRequest);
    if (!szSessionId)
    {
        // User doesn't have valid cookie, redirect to login page.
        m_HttpResponse.Redirect("login.srf");
        return HTTP_SUCCESS_NO_PROCESS;
    }

    // Get the session.
    CComPtr<ISession> spSession;
    hr = GetSession(m_spServiceProvider, szSessionId, &spSession);
    if (FAILED(hr))
    {
        return SendError(m_HttpResponse,
                     "An error occurred while obtaining the session object.");
    }

    // Get the user name.
    CComVariant varUserName;
    hr = spSession->GetVariable("UserName", &varUserName);
    if (FAILED(hr))
    {
        return SendError(m_HttpResponse,
                     "An error occurred while obtaining a session variable.");
    }

    // Get the user name as a string.
    CStringA strUserName = varUserName;

    // Commit the order to the database.
    hr = CommitOrder(m_spServiceProvider, spSession, strUserName);
    if (FAILED(hr))
    {
        return SendError(m_HttpResponse,
               "An error occurred while writing the order to the database.");
    }

     // <A HREF="vcconatlservertutorialconfirmingtheorder.htm">TODO - send e-mail conf
irmation</A>

    return HTTP_SUCCESS;
}
```

The code ensures that the user has logged in and redirects them to the login page if they have not. `ValidateAndExchange` then gets the UserName variable from session state and calls `CommitOrder`, which loops through the items in the user's shopping cart, building up a string of product IDs and quantities to write to the database:

```
HRESULT CCheckoutHandler::CommitOrder(IServiceProvider* pServiceProvider,
                          ISession* pSession, const CStringA& strUserName)
{
    using VCUE::OpenCommandRowset;

    // Initialize variables.
```

```
    HRESULT hr = E_UNEXPECTED;
    HSESSIONENUM hSession = NULL;
    POSITION posSession = NULL;

    // Get the session variable enumeration handle.
    hr = pSession->BeginVariableEnum(&posSession, &hSession);
    if (FAILED(hr))
        return hr;

    // Get the first variable.
    char szProductId[MAX_VARIABLE_NAME_LENGTH];
    CComVariant varQuantity;
    hr = pSession->GetNextVariable(&posSession, &varQuantity, hSession,
                                   szProductId, sizeof(szProductId));
    if (FAILED(hr))
        return hr;

    // Build a regular expression for matching decimal integers
    CRegularExpression regularExpression;
    if (regularExpression.Parse("^\\d+$") != REPARSE_ERROR_OK)
        return E_UNEXPECTED;

    CStringA strOrder;
    CMatchContext matchContext;
    while (hr == S_OK)
    {
        // Append session variables to order string if
        // name is a decimal integer string and value
        // is a variant of type long (VT_I4).
        if (regularExpression.Match(szProductId, &matchContext))
        {
            if (VT_I4 == V_VT(&varQuantity))
            {
                strOrder.Append(szProductId);
                strOrder.Append("\t");
                strOrder.Append(CStringA(varQuantity));
                strOrder.Append("\n");
            }
        }

        // Check whether all session variables have been examined.
        if (!posSession)
            break;

        // Get the next session variable.
        varQuantity.Clear();
        hr = pSession->GetNextVariable(&posSession, &varQuantity, hSession,
                                       szProductId, sizeof(szProductId));
    }

    if (SUCCEEDED(hr))
    {
        // Commit the order to the database.
        CCommandCreateOrder cmdNewOrder;
        hr = cmdNewOrder.SetUserName(strUserName);
        if (SUCCEEDED(hr))
        {
            hr = cmdNewOrder.SetOrder(strOrder);
            if (SUCCEEDED(hr))
```

```
            {
                hr = OpenCommandRowset(pServiceProvider, cmdNewOrder);
            }
        }
    }

    return hr;
}
```

Next

Confirming the Order

See Also

Reference

ATL Server Tutorial

Confirming the Order

In this step, you will build an e-mail message and send it using Simple Mail Transport Protocol (SMTP). First, you must obtain the user's e-mail address from the database.

To obtain the user's e-mail address from the database

1. In Solution Explorer, double-click Commands.h and add the following command classes:

```
[
    db_command(
        "SELECT [SiteUser].[Email] "
        "FROM [SiteUser] "
        "WHERE [SiteUser].[UserName] = ?"
    )
]
class CGetEmail
{
public:
    [ db_param(1, status = m_dwUserNameStatus,
      length = m_dwUserNameLength) ]
    char m_UserName[51];

    [ db_column(1, status = m_dwEmailStatus,
      length = m_dwEmailLength) ]
    char m_Email[256];

    DBSTATUS m_dwUserNameStatus;
    DBSTATUS m_dwEmailStatus;

    DBLENGTH m_dwUserNameLength;
    DBLENGTH m_dwEmailLength;
};

class CCommandGetEmail : public CGetEmail
{
public:
   CCommandGetEmail();
   void Clear();
   HRESULT SetUserName(LPCSTR szUserName);
};
```

The parameterized query just returns the e-mail address for the specified user.

2. In Solution Explorer, double-click Commands.cpp and add the following code:

```
// Initialization.
CCommandGetEmail::CCommandGetEmail()
{
    Clear();
}

// Initialization.
void CCommandGetEmail::Clear()
{
    m_dwUserNameStatus = DBSTATUS_S_ISNULL;
    m_dwEmailStatus = DBSTATUS_S_ISNULL;
```

```
    m_dwUserNameLength = 0;
    m_dwEmailLength = 0;

    m_UserName[0] = _T('\0');
    m_Email[0] = _T('\0');
}

HRESULT CCommandGetEmail::SetUserName(LPCSTR szUserName)
{
    return VCUE::SetOleDbStringMember(m_UserName, m_dwUserNameStatus,
                                      m_dwUserNameLength, szUserName);
}
```

You must now include the header that will allow access to the ATL Server SMTP support classes and define values for symbols that will be used to connect to your mail server.

To include the header and define the values

- In Solution Explorer, double-click StdAfx.h and add the following code, replacing the definitions of `MAIL_SERVER_NAME` and `MAIL_SENDER_ADDRESS` with values that are appropriate to your network:

```
#include <atlsmtpconnection.h>
// Definitions for mail support.
// Change these to match your own mail server and e-mail address.
#define MAIL_SERVER_NAME "mailserver"
#define MAIL_SENDER_ADDRESS "user@yourcompany.com"
```

The code that you add to Checkout.cpp in place of the comment `TODO - send e-mail confirmation` will get the user's e-mail address using `CCommandGetEmail`, create the confirmation message using CMimeMessage, and send it using CSMTPConnection.

To change the TODO comment in Checkout.cpp

- In Solution Explorer, double-click Checkout.cpp and replace `TODO - send e-mail confirmation` with the following code:

```
    // Get the e-mail address of the user.
    CCommandGetEmail cmdGetEmail;
    hr = cmdGetEmail.SetUserName(strUserName);
    if (FAILED(hr))
    {
        return SendError(m_HttpResponse,
                         "An error occurred. The user name is too long.");
    }

    hr = OpenCommandRowset(m_spServiceProvider, cmdGetEmail);
    if (FAILED(hr))
    {
        return SendError(m_HttpResponse,
                         "An error occurred while accessing the database.");
    }

    hr = cmdGetEmail.MoveFirst();
    if ((S_OK != hr) || (cmdGetEmail.m_dwEmailStatus != DBSTATUS_S_OK))
    {
        return SendError(m_HttpResponse,
                         "An error occurred. Unable to obtain e-mail address.");
    }
```

```
    // Create the confirmation message
    CMimeMessage msgConfirmation;
    BOOL bSuccess = msgConfirmation.SetSender(MAIL_SENDER_ADDRESS);
    if (bSuccess)
    {
        bSuccess = msgConfirmation.SetDisplayName("ATL Server Tutorial");
        if (bSuccess)
        {
            bSuccess = msgConfirmation.AddRecipient(cmdGetEmail.m_Email,
                                                    cmdGetEmail.m_Email);
            if (bSuccess)
            {
                bSuccess = msgConfirmation.SetSubject("Order Confirmation");
                if (bSuccess)
                {
                    bSuccess = msgConfirmation.AddText(
               "Your order has been received and will be shipped to you ASAP");
                }
            }
        }
    }

    if (!bSuccess)
    {
        return SendError(m_HttpResponse,
               "An error occurred while creating the confirmation message.");
    }

    // Open a connection to the SMTP mail server.
    CSMTPConnection smtpConnection;
    if (!smtpConnection.Connect(MAIL_SERVER_NAME))
    {
        return SendError(m_HttpResponse,
         "An error occurred connecting to the SMTP server. "
         "Make sure that SERVER_NAME (defined in stdafx.h) is set correctly.");
    }

    // Send the confirmation message.
    if (!smtpConnection.SendMessage(msgConfirmation))
    {
        return SendError(m_HttpResponse,
            "An error occurred while trying to send the confirmation e-mail.");
    }
    // <A HREF="vcconatlservertutorialgatheringstatistics.htm">TODO - Log purchase as
performance counter</A>
```

CMimeMessage provides methods for building up a MIME-format message that we can send using SMTP.

The user is provided with the e-mail address from which the message originated by calling CMimeHeader::SetSender. The user can then reply to the mail with any questions they may have about their order. (The e-mail address of the sender is also usually checked by the SMTP server to reduce e-mail spoofing, so it is important to set `MAIL_SENDER_ADDRESS` to an address that is valid for the server specified in `MAIL_SERVER_NAME`). The common name of the sender is specified with a call to CMimeMessage::SetDisplayName.

The e-mail address and display name of the user to whom the mail is sent are specified by calling CMimeHeader::AddRecipient. (A message can have multiple recipients, but only a single sender.)

The subject and text of the message are set using CMimeHeader::SetSubject and CMimeMessage::AddText, respectively.

Finally, a connection is made to the SMTP server using CSMTPConnection::Connect and the message is sent with a call to CSMTPConnection::SendMessage.

Next

Changing Identity

See Also

Reference

ATL Server Tutorial

Changing Identity

At this stage, you should be able to build and run the application. In this step, you will see how to create and use dynamic services, how to expose performance counters, and how to help take control of the security context in which your code runs.

First, you will add a class to Helpers.h that will allow you to temporarily revert the security context of the current thread to its original security context (in other words, it terminates any impersonation that is currently happening and restores it at a later stage).

To add a class to Helpers.h

- In Solution Explorer, double-click Helpers.h and add the following code:

```
class CTemporaryRevertToSelf
{
public:
    CTemporaryRevertToSelf(HRESULT& hr) throw()
        : m_hThreadToken(INVALID_HANDLE_VALUE)
        , m_hCurrentThread(GetCurrentThread())
    {
        ATLASSERT(GetCurrentThread() == m_hCurrentThread);

        hr = Open();

        if (SUCCEEDED(hr))
        {
            if (!RevertToSelf())
            {
                hr = AtlHresultFromLastError();
                Close();
            }
        }
    }

    ~CTemporaryRevertToSelf() throw()
    {
        ATLASSERT(GetCurrentThread() == m_hCurrentThread);

        // Enable this assertion
        // if you want to ensure Restore
        // is called explicitly
        // ATLASSERT(!IsOpen());

        HRESULT hr = Restore();
        ATLASSERT(SUCCEEDED(hr));
    }

    HRESULT Restore() throw()
    {
        ATLASSERT(GetCurrentThread() == m_hCurrentThread);

        HRESULT hr = S_OK;
        if (IsOpen())
        {
            if (!SetThreadToken(&m_hCurrentThread, m_hThreadToken))
            {
                hr = AtlHresultFromLastError();
            }
```

```
                Close();
            }

            ATLASSERT(!IsOpen());
            return hr;
        }

    private:
        HANDLE m_hThreadToken;
        HANDLE m_hCurrentThread;

        HRESULT Open()
        {
            ATLASSERT(!IsOpen());

            BOOL bSuccess = OpenThreadToken(
                                m_hCurrentThread,
                                TOKEN_IMPERSONATE | TOKEN_DUPLICATE,
                                FALSE,
                                &m_hThreadToken
                                );

            ATLASSERT(bSuccess ? IsOpen() : !IsOpen());

            return bSuccess ? S_OK : AtlHresultFromLastError();
        }

        HRESULT Close()
        {
            ATLASSERT(IsOpen());

            BOOL bSuccess = CloseHandle(m_hThreadToken);
            m_hThreadToken = INVALID_HANDLE_VALUE;
            ATLASSERT(bSuccess);
            return bSuccess ? S_OK : AtlHresultFromLastError();
        }

        bool IsOpen()
        {
            return INVALID_HANDLE_VALUE != m_hThreadToken;
        }
    };
```

The constructor of this class obtains the impersonation token for the current thread and stores it in `m_hThreadToken` so that impersonation using that token can be restored later. It then calls `RevertToSelf` to restore the original security context of the thread. The user of the class is able to obtain the status of the calls using the `HRESULT` parameter passed by reference to the constructor.

`Restore` determines whether `m_hThreadToken` contains a valid handle. If so, it uses it to set the impersonation token of the current thread and then closes the handle. Note that `Restore` can be called safely if the constructor failed or if `Restore` has been called previously.

The destructor helps ensure that `Restore` is called and the impersonation token is reattached to the current thread even if an exception is thrown and `Restore` is not called manually.

Next

Gathering Statistics

See Also

Reference

ATL Server Tutorial

Gathering Statistics

In this step, you will add code to allow you to gather statistics.

To create the file PerfService.h and add the code to the necessary files

1. In Solution Explorer, right-click the Tutorial project.
2. On the shortcut menu, click **Add**, and then click **Add New Item**.

   The **Add New Item** dialog box appears.
3. In the Templates pane, click **Header File (.h)**, and enter **PerfService.h** in the **Name** box.
4. Click **Open**.
5. Add the following code:

```
#pragma once
#include "Helpers.h"

// perfmon object
[ perf_object(namestring = "ATL Server Tutorial", helpstring = "Number of orders taken
by ATL Server tutorial application", detail = PERF_DETAIL_NOVICE) ]
class CTutorialStatistics
{
public:
    [ perf_counter(namestring = "Orders", helpstring = "Number of orders taken by ATL
Server tutorial application", countertype = PERF_COUNTER_RAWCOUNT, defscale = 0, detai
l = PERF_DETAIL_NOVICE) ]
    LONG m_dwOrders;
};
```

`CTutorialStatistics` is the class that defines the statistics that will be exposed by your Web application. The perf_object attribute is used to declare the class as a performance object whose name is specified using the *namestring* parameter. In this case, you will only provide a single counter for the application: the number of orders taken. The data member used to store the count is `m_dwOrders`, and you have declared that the counter will be named **Orders** using the *namestring* parameter of the perf_counter attribute.

```
// perfmon class
[ perfmon(name="Perf_TutorialPerf", register=true) ]
class CTutorialPerformanceManager
{
};

CTutorialPerformanceManager g_PerformanceManager;
```

`CTutorialPerformanceManager` is the class responsible for registering, creating, and managing the performance objects in this application. Each DLL that exposes performance counters and is responsible for registering them needs to create a single global performance manager object to make that happen. `_ATL_PERF_REGISTER` also needs to be defined before including atlperf.h.

```
__interface ATL_NO_VTABLE
__declspec(uuid("2DC4AE64-BCD6-401A-9212-3A7EAF536F89"))
ITutorialPerformanceService : public IUnknown
{
    HRESULT AddOrder();
};
```

`ITutorialPerformanceService` is the interface that will be implemented by the service that will allow you to adjust the performance counters exposed by the application. The interface needs only a single method, `AddOrder`, that will increment the number of orders when called.

```
class CTutorialPerformanceService :
    public CComObjectRootEx<CComMultiThreadModel>,
    public CComCoClass<CTutorialPerformanceService>,
    public ITutorialPerformanceService
{
public:
    BEGIN_COM_MAP(CTutorialPerformanceService)
        COM_INTERFACE_ENTRY(ITutorialPerformanceService)
    END_COM_MAP()

    CTutorialStatistics* m_pStatistics;

    CTutorialPerformanceService()
    {
        m_pStatistics = NULL;
    }

    DECLARE_PROTECT_FINAL_CONSTRUCT()

    HRESULT FinalConstruct()
    {
        HRESULT hr = S_OK;
        VCUE::CTemporaryRevertToSelf rts(hr);
        if (FAILED(hr))
            return hr;

        hr = g_PerformanceManager.Initialize();

        if (SUCCEEDED(hr))
        {
            CPerfLock lock(&g_PerformanceManager);
            if (SUCCEEDED(lock.GetStatus()))
                hr = g_PerformanceManager.CreateInstance(1, L"Tutorial",
                                                         &m_pStatistics);
            else
                hr = E_FAIL;
        }
        return hr;
    }

    void FinalRelease()
    {
        HRESULT hr = S_OK;
        VCUE::CTemporaryRevertToSelf rts(hr);
        ATLASSERT(SUCCEEDED(hr));

        CPerfLock lock(&g_PerformanceManager);
        if (SUCCEEDED(lock.GetStatus()))
            hr = g_PerformanceManager.ReleaseInstance(m_pStatistics);

        g_PerformanceManager.UnInitialize();
    }

    HRESULT AddOrder()
```

```
    {
        HRESULT hr = E_UNEXPECTED;
        if (m_pStatistics)
        {
            InterlockedIncrement(&(m_pStatistics->m_dwOrders));
            hr = S_OK;
        }
        return hr;
    }
};
```

`CTutorialPerformanceService` is the class that implements the `ITutorialPerformanceService` interface. You can see that it implements the interface in the standard ATL style, by deriving from CComObjectRootEx and CComCoClass, and using the COM map to provide the implementation of IUnknown.

`FinalConstruct` is called when an instance of the class is first created. It is a good place to do initialization because the method returns an HRESULT that can be used to communicate the status of the initialization to the client.

The code uses `CTemporaryRevertToSelf` to temporarily turn off impersonation. This allows the initialization of the performance manager to access the registry keys necessary for the correct operation of the performance counters.

Once initialized, the performance manager is locked using CPerfLock to prevent multiple simultaneous access while creating instances of the performance counters. The performance manager is automatically unlocked when `lock` goes out of scope. The instance of the performance object is named **Tutorial,** and a pointer is stored in `m_pStatistics`.

`FinalRelease` is the complement of `FinalConstruct` and is called when an instance of the class is being destroyed. The code in this method again temporarily turns off impersonation, this time to take the lock, destroy the instance of the performance object, and uninitialize the performance manager.

`AddOrder` increments the `m_dwOrders` member of the performance object. Note that it uses InterlockedIncrement to ensure that the update occurs atomically.

6. In Solution Explorer, double-click StdAfx.h and add the following definition before any `#include` statements to ensure that the performance monitor objects can be registered:

```
#define _ATL_PERF_REGISTER
```

7. In Solution Explorer, double-click Checkout.h and add the lines shown in bold:

```
#pragma once
#include "PerfService.h"

[ request_handler("Checkout") ]
class CCheckoutHandler
{
public:
   HTTP_CODE ValidateAndExchange();
   HRESULT CommitOrder(IServiceProvider* pServiceProvider, ISession* pSession,
                                                const CStringA& strUserName);
   HRESULT GetPerformanceService(ITutorialPerformanceService** ppService);
};
```

`GetPerformanceService` will create the performance service object and add it to the list of services supported by the ISAPI extension, or it will obtain the existing object from the ISAPI extension.

To add the implementation of GetPerformanceService to Checkout.cpp

1. In Solution Explorer, double-click Checkout.cpp and add the following code:

```
HRESULT CCheckoutHandler::GetPerformanceService(
```

```
                                        ITutorialPerformanceService** ppService)
{
    using VCUE::QueryService;

    HRESULT hr = E_UNEXPECTED;

    // Get the performance service.
    CComPtr<ITutorialPerformanceService> spService;
    hr = QueryService(m_spServiceProvider, &spService);

    if (FAILED(hr))
    {
        // Failed to get an existing performance service, so create
        // the service and register it with the ISAPI DLL.
        hr = CTutorialPerformanceService::CreateInstance(&spService);
        if (SUCCEEDED(hr))
        {
            hr = m_spExtension->AddService(
                __uuidof(ITutorialPerformanceService),
                __uuidof(ITutorialPerformanceService),
                spService,
                _pModule->GetModuleInstance());

            if (S_FALSE == hr)
            {
                // The service was already added, release the
                // new service and obtain the one from the ISAPI
                // extension.
                spService.Release();
                hr = QueryService(m_spServiceProvider, &spService);
            }
        }
    }

    *ppService = spService.Detach();
    return hr;
}
```

`GetPerformanceService` attempts to obtain the `ITutorialPerformanceService` pointer from the ISAPI extension through the `IServiceProvider` interface. If that fails, it means that the service has not yet been created and added to the list of services supported by the extension. In that event, the code creates an instance of `CTutorialPerformanceService` and registers it with the ISAPI extension as a dynamic service by calling IIsapiExtension::AddService. Note that `AddService` returns `S_FALSE` if the service is already present.

2. Replace `TODO - Log purchase as performance counter` with the following code:

```
    // Create a perfmon counter for number of orders
    // Remember to regsvr32 your application dll (tutorial.dll)
    CComPtr<ITutorialPerformanceService> spService;
    hr = GetPerformanceService(&spService);
    if (SUCCEEDED(hr))
    {
        hr = spService->AddOrder();
    }

    if (FAILED(hr))
    {
        return SendError(m_HttpResponse,
```

```
                        "An error occurred while creating a PerfMon counter.");
    }
```

The code calls `GetPerformanceService` to get a pointer to the dynamic service and then calls `AddOrder` to increment the number of orders tracked by the performance counter.

See Also

Reference

ATL Server Tutorial

Troubleshooting (ATL Server)

This topic includes troubleshooting steps that you can use if you have trouble getting the tutorial running after successfully building it.

To manually deploy the tutorial

1. Create a folder on a drive on the Web server that you will use. Copy the following files to the new folder:
   - Cart.srf
   - Checkout.srf
   - Login.srf
   - Tutorial.srf
2. Create a folder named **bin** beneath the one you created in the previous step. Copy the following files to the new folder:
   - Tutorial.dll
   - TutorialIsapi.dll
   - Tutorial.udl
3. Double-click Tutorial.udl and adjust the data link properties as described in Deploying the Database.
4. Start Internet Services Manager. On the **Start** menu, click **Programs**, then click **Administrative Tools**, and then click **Internet Services Manager**.
5. Expand the tree to show the **Default Web Site**.
6. Right-click the item and, on the shortcut menu, click **New**, and then click **Virtual Directory**.
7. Enter **Tutorial** in the **Alias** box.
8. Select the folder created in the first step as the **Web Site Content Directory** for this new virtual directory.
9. Give the new directory **Read**, **Run scripts**, and **Execute** permissions.
10. Click **Next** and **Finish** to end the wizard and create the virtual directory.
11. Right-click the newly created virtual directory and click **Properties** on the shortcut menu.
12. Click the **Configuration** button on the **Virtual Directory** page of the **Properties** dialog box.
13. Use the **App Mappings** page of the **Application Configuration** dialog box to register TutorialIsapi.dll as the handler for server response files as described in Associating SRF Files with Your ISAPI DLL by Hand.

If the code is failing to create or increment the performance counters

1. Ensure that you added `#define _ATL_PERF_REGISTER` to StdAfx.h before any headers were included.
2. Ensure that Tutorial.dll is registered (run `regsvr32 Tutorial.dll` from the command prompt).

   The Web deployment tool will not register a DLL unless it is an ISAPI extension, so you must manually do this step.

See Also
Reference
ATL Server Tutorial

ATL Server Architecture

This section provides information about the request handling architecture provided by ATL Server for your Web applications and XML Web services. Each main element is described in detail in its own topic.

In This Section

Major Components

Contains a high level overview of the major components in the request handling architecture: IIS, ISAPI extension DLLs, Web application DLLs, and server response files.

Internet Information Services

Describes the role of IIS in the request handling architecture and provides links to topics related to registering your DLLs with IIS.

ISAPI Extension DLL

Provides information about the important interfaces and classes used by your ISAPI extension DLL and describes its relationship to IIS and Web application DLLs.

Thread Pool

Provides details on the role of the thread pool in ATL Server applications. Describes the creation of per-thread services, resizing the pool, and obtaining information about the pool's behavior.

Web Application DLL Cache

Contains information about managing the DLL cache and replacing Web application DLLs while the application is running.

Server Response File Cache

Describes the operation of the server response file cache and provides information about obtaining cache statistics and counters.

Response Cache

Provides information about controlling the caching of the responses generated by your request handlers and examines some security related issues.

Web Application DLL

Explains the basic features of a Web application DLL.

Request Handler

Describes the features of a request handler, including the classes and interfaces of interest, and provides information about creating replacement methods.

Server Response File

Describes the role of server response files in the ATL Server request handling architecture and provides links to further reference topics.

Sequence of Events

Describes the typical sequence of events when an HTTP request for a server response file is received by an ATL Server application.

SRF Request Architecture

Displays the essential elements of the request handling architecture as a diagram.

Related Sections

ATL Server

Provides links to conceptual information about the functionality of ATL Server.

ATL Server Samples

Provides links to samples demonstrating how ATL Server is used.

ATL Server Reference

Provides links to reference documentation for the ATL Server Library.

Major Components

The ATL Server request handling architecture consists of the following elements:

Name	Description
Internet Information Services (IIS)	The Web server receives requests from the client and passes them on to the ISAPI extension DLL when the request is for a file extension handled by that DLL.
ISAPI Extension DLL	The ISAPI extension DLL receives requests from IIS and passes them on to the appropriate Web application DLL. The ISAPI extension DLL provides the request handling infrastructure and common services used by the Web application DLLs. There is typically one ISAPI extension DLL per virtual directory that is used to handle requests for files with the .dll and .srf extensions. (Each file extension can be registered to at most one DLL.)
Web Application DLLs	Web application DLLs provide application-specific functionality for handling requests and generating responses. Web application DLLs expose request handlers that communicate with the ISAPI extension DLL. There can be many Web application DLLs in a virtual directory. Each Web application DLL can provide one or more named request handlers.
Server Response files	Server response files are text files that contain the static parts of a response and special tags describing where request handler methods can be called to generate the dynamic parts of the response. Each Web application DLL can be used from multiple server response files. Each server response file can use more than one Web application DLL.

See Also

Concepts

SRF Request Architecture
Sequence of Events

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Internet Information Services

Internet Information Services (IIS) provides the means of receiving HTTP requests from clients and passing them on to your ATL Server code using the ISAPI programming interface. ATL Server applications use an ISAPI extension DLL to receive requests from and return responses to the client.

For IIS to pass on requests to your ISAPI extension DLL, it must be correctly registered. The following topics will help ensure that your extension DLL is correctly installed:

Web Deployment Property Page

Use the Web Deployment property page to enable deployment to IIS on your local machine during the development process.

Associating .srf Files with Your ISAPI DLL by Hand

If you encounter problems or you want to see exactly what you are doing, this topic describes how to manually install your ISAPI extension DLL.

Deploying ATL Server Applications

Follow the advice in this topic when you need to distribute your application to run on customers' machines.

Once installed and registered, your ATL Server ISAPI extension DLL will receive requests from IIS when one of the following conditions is met:

- The request is for the ISAPI extension DLL.
- The request is for a server response file (a file with .srf as the extension).
- The request is for a Web application DLL (a file with .dll as the extension).

See Also

Reference

ISAPI Extension Overview

Concepts

SRF Request Architecture

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Samples

ISAPI Extension DLL

ATL Server Web applications and services communicate with Internet Information Services (IIS) using the Internet Server Application Programming Interface (ISAPI). ATL Server applications use an ISAPI extension DLL to receive requests from and return responses to the client.

Although the ISAPI extension DLL can handle a request entirely by itself, it is usually responsible for passing the request to a request handler implemented in a Web application DLL.

This mechanism requires the ISAPI extension DLL to be registered with IIS as the handler for requests with the .srf and .dll extensions. The registration can be set up by using the Web Deployment property page or by doing it manually.

When the ISAPI extension DLL receives a request for a server response file, it loads the file, reads the handler tag, and uses that information to dispatch the request to the appropriate request handler in the specified Web application DLL.

Note
The ISAPI extension reads only the handler tag in order to pass the request on to the appropriate request handler. It does not read any other tags in the server response file. The request handler is responsible for parsing the server response file. See Sequence of Events for details.

When the ISAPI extension DLL receives a request for a DLL, it checks the query string for the presence of a *Handler* parameter and uses that information to dispatch the request to the handler in the specified DLL.

The ISAPI extension DLL exposes functionality for use by the Web application DLL through the IHttpServerContext, IIsapiExtension, and IServiceProvider interfaces.

IHttpServerContext provides access to information about the Web server and the particular HTTP request to be handled. This interface is essentially a wrapper for the **EXTENSION_CONTROL_BLOCK** structure familiar to ISAPI programmers.

IIsapiExtension provides the ability to initialize threads in the thread pool by overriding OnThreadAttach and OnThreadTerminate, add and remove dynamic services by calling AddService and RemoveService, and obtain the worker object for the current thread by calling GetThreadWorker.

The **IServiceProvider** interface provides access to other services provided by the ISAPI extension.

The Web application DLL exposes its functionality through objects that implement IRequestHandler. In this way, ATL Server provides the ability to create a number of binary components, each of which is responsible for a different part of the application.

A project for an ATL Server ISAPI extension DLL can be generated using the ATL Server Project Wizard. A Web application DLL project may be created at the same time. The DLLs can even be combined using the **Generate combined DLL** option on the Project Settings page of the ATL Server Project Wizard.

The majority of the functionality of the ISAPI extension DLL is provided by CIsapiExtension. The ATL Server Project Wizard generates a derived class that you can customize. For example, you can add extra features that can be made available to all Web application DLLs that use that ISAPI extension DLL or provide processing that applies to every request or response.

CIsapiExtension provides the following features:

Feature	Description
Thread Pool	The thread pool is the key to the high performance and scalability of ATL Server Web applications. The ISAPI extension DLL adds each request that it receives to a request queue that is serviced by it s own threads. This quickly frees the Web server thread to handle another incoming request and e nsures that the application scales as more clients make requests of the server.
Web Application DLL Cache	Web application DLLs and pointers to their exported functions are loaded on first use and cached t o improve performance.
Server Response File Cache	Server response files are parsed and references to request handlers are resolved on first use. The r esults are cached to improve performance.

Response Cache	The responses generated by the Web application can be cached at your discretion. Responses are cached as files that are sent back to the client asynchronously for best performance.

See Also

Concepts

Web Application DLL

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Samples

Thread Pool

The thread pool is the key to the high performance and scalability of ATL Server Web applications. The ISAPI extension DLL adds each request that it receives to a request queue that is serviced by its own threads. This quickly frees the Web server thread to handle another incoming request and ensures that the application scales as more clients make requests of the server. The requests added to the queue are serviced in the same order in which they are received.

The thread pool is specified as a class that is passed as the first template argument to CIsapiExtension. Typically, the class is a specialization of CThreadPool although another compatible class can be used.

CThreadPool uses a worker class that provides the code and manages state for each thread in the pool. This class is specified as a template argument. When used with **CIsapiExtension**, the worker class must be CIsapiWorker or a derived class.

CIsapiWorker understands the format of the requests that are queued on the thread pool and works with the ISAPI extension class to provide various features. As a result of the collaboration between **CIsapiWorker** and **CIsapiExtension**, you can be notified when a pool thread is created or destroyed by overriding IIsapiExtension::OnThreadAttach and IIsapiExtension::OnThreadTerminate. Your request handlers can obtain a pointer to the worker object for the current thread by calling IIsapiExtension::GetThreadWorker.

Note For more information on thread initialization, see ATL Server and COM.

By deriving your own class from **CIsapiWorker** and adding your own methods and data members to that class, you can provide per-thread services that can be used in your request handlers. This can be a powerful technique for ensuring scalability and keeping your code simple — data that is only accessible from a single thread does not need synchronization.

For an example of providing per-thread services, generate an ATL Server project using the wizard and choose to add support for the data source cache on the Server Options page of the ATL Server Project Wizard.

A thread pool created using **CThreadPool** does not resize itself automatically but can be resized during use by calling CThreadPool::SetSize. The current size of the pool can be obtained by calling CThreadPool::GetSize.

The thread pool can be resized by code within the ISAPI extension or a Web application DLL by calling methods on the IThreadPoolConfig interface. This interface can be obtained for the ISAPI extension's thread pool by calling **QueryService** on the extension's **IServiceProvider** interface using **__uuidof(IThreadPoolConfig)** for both the service and interface identifiers.

ATL Server provides an XML Web service and an HTML user interface for managing the thread pool remotely. For details, see Extension Management Services.

CIsapiExtension optionally provides information about the functioning of the thread pool, such as the number of threads actively handling a request and the number of requests in the queue, through the IRequestStats interface and through performance counters. For details, see the description of the *CRequestStatClass* template parameter used by CIsapiExtension.

Note CThreadPool can be used independently of the request handling architecture. See the CThreadPool Sample for an example of using **CThreadPool** directly.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Server Response File Cache

Server response files are parsed and references to request handlers are resolved on first use. The results are cached to improve performance and stored in the server response file cache or stencil cache. During this process, further server response files, request handlers, and Web application DLLs may be loaded, initialized, and stored in the ISAPI extension caches.

The stencil cache can be managed by manipulating the CIsapiExtension::m_StencilCache member directly if necessary.

ATL Server provides an XML Web service and an HTML user interface for managing the server response file cache remotely. See Extension Management Services for details.

CIsapiExtension optionally provides information about the server response file cache, such as its current size, through the IMemoryCacheStats interface and through performance counters. For details, see the description of the *CStencilCacheStats* template parameter used by CIsapiExtension.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Web Application DLL Cache

Web application DLLs and pointers to their exported functions are loaded on first use and cached to improve performance.

The Web application DLL cache can be manipulated within the ISAPI extension or a Web application DLL by calling methods on the IDllCache interface. This interface can be obtained for the ISAPI extension's DLL cache by calling **QueryService** on the extension's **IServiceProvider** interface using **__uuidof(IDllCache)** for both the service and interface identifiers.

ATL Server provides an XML Web service and an HTML user interface for managing the Web application DLL cache remotely. For details, see Extension Management Services.

You can replace Web application DLLs while your Web application is running if the DLLs are not cached. Although it is not trivial to replace a Web application DLL that is in the cache, it is possible. The basic approach is to ensure that there are no outstanding references on the DLL that you need to replace and then flush the cache so that the DLL is unloaded. References can be held on the DLL if it is referred to in a server response file or provides a dynamic service. For details, see the HotSwap sample.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Response Cache

The responses generated by the Web application can be cached at your discretion. Responses are cached as files that are sent back to the client asynchronously for best performance. The headers sent with the original response are also cached and returned on subsequent requests.

See Enabling Page Caching for details on the steps that you need to take to use ATL Server's support for response caching.

Note that redirection and asynchronous methods are not available to request handlers that have declared that their responses can be cached. If you attempt to use these features, you will see assertion failures at runtime, and the methods will fail.

Note This is true even if your request handler later decides that the current response should not be cached by using IPageCacheControl.

Other features involving logging and impersonation will generate trace messages in debug builds but will work as expected. The warnings are there to remind you that any explicit impersonation or logging that you do during the generation of the initial response will not be repeated when the response is returned from the cache.

See CCacheServerContext for details and a complete list of the affected **IHttpServerContext** methods.

CIsapiExtension optionally provides information about the response cache, such as its current size, through the IMemoryCacheStats interface and through performance counters. For details, see the description of the *CPageCacheStats* template parameter used by CIsapiExtension.

Note The reported size of this cache is the size of the files on disk; only the HTTP headers are kept in memory. They are not counted for size calculations.

Security Considerations

Because ATL Server code in **CIsapiExtension** impersonates the current client before executing any other code, cached responses that were written in the security context of one client may not be available to other clients.

When **CIsapiExtension** cannot open a file in the response cache, it generates a new noncached response. This ensures that clients that could obtain a response from the original handler are not prevented from obtaining a response because the file was mistakenly cached in a way that prevents them from gaining access. However, if many files are cached in this way, the benefits of caching may be lost.

If anonymous access is enabled for the cached resource and it does not require authentication, all clients will write to and read from the cache using the same account. If authentication is enabled, however, you could consider overriding CIsapiExtension::GetCacheServerContext and CIsapiExtension::TransmitfromCache to take more control over the accounts used to write to and read from the cache.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Web Application DLL

A Web application DLL, also known as a request handler DLL, houses request handlers that can be used to respond to HTTP requests. The Web application DLL must export a function that can be used to obtain a request handler given its name. The DLL may also export two other functions that can be used for initialization and cleanup.

Exported function	Description
GETATLHANDLERBYNAME	Required. This function is called to obtain a pointer to a request handler given its name.
INITIALIZEATLHANDLERS	Optional. This function is called before any request handlers are requested. The Web application DLL can perform custom initialization in response to this function call.
UNINITIALIZEATLHANDLERS	Optional. This function is called after all request handlers have been released. The Web applicatio n DLL can perform custom cleanup in response to this function call.

A project for an ATL Server Web application DLL can be generated using the ATL Server Project Wizard. An ISAPI extension DLL project may be created at the same time (an ISAPI extension DLL may also act as a Web application DLL).

These functions do not need to be explicitly implemented in wizard-generated ATL Server projects. They are automatically implemented from the information provided by using the HANDLER_ENTRY macro or request_handler attribute in your code.

GETATLHANDLERBYNAME creates a new instance of the request handler class on each request for a named handler by calling CreateRequestHandler on the appropriate class.

INITIALIZEATLHANDLERS calls InitRequestHandlerClass on each request handler class.

UNINITIALIZEATLHANDLERS calls UninitRequestHandlerClass on each request handler class.

See Also

Concepts

ISAPI Extension DLL

Request Handler

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Samples

Request Handler

A request handler is a class in a Web application DLL that implements the IRequestHandler interface. Typically, such classes derive from CRequestHandlerT and make use of the HANDLER_ENTRY macro or have the request_handler attribute applied to them.

The most important method in the **IRequestHandler** interface is HandleRequest, which receives IHttpServerContext and IServiceProvider pointers from the ISAPI extension DLL.

The default implementation of the **IRequestHandler** interface is provided by IRequestHandlerImpl and CRequestHandlerT. If a request was made for a server response file (SRF), this class parses that file, builds a response from the static content of the SRF and calls replacement methods that correspond to tags in the SRF provided by the application-specific class derived from **CRequestHandlerT**. If the SRF includes other SRFs or DLLs, the request handler will call in to those DLLs as well.

The request handler communicates with the ISAPI extension and ultimately the client using the interfaces provided for that purpose:

Interface	Data member
IIsapiExtension	IRequestHandlerImpl::m_spExtension
IServiceProvider	IRequestHandlerImpl::m_spServiceProvider
IHttpServerContext	IRequestHandlerImpl::m_spServerContext

CRequestHandlerT provides request and response objects that provide a higher level abstraction on top of **IHttpServerContext**. The request and response objects can be accessed using the CRequestHandlerT::m_HttpRequest and CRequestHandlerT::m_HttpResponse members.

See Initializing Request Handlers for information on the points at which you can do initialization in a request handler.

See Also

Concepts

ISAPI Extension DLL

Replacement Method

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Samples

Replacement Method

A replacement method or request handler method is a method in a request handler that is associated with a tag name so that it can be called from a server response file.

The mapping of tag names to request handler methods is typically provided by the replacement method map (the BEGIN_REPLACEMENT_METHOD_MAP, REPLACEMENT_METHOD_ENTRY, and END_REPLACEMENT_METHOD_MAP macros) found in a **CRequestHandlerT**-derived class or by use of the tag_name attribute.

Replacement methods often generate part of a response by writing to the response stream (for example, using CRequestHandlerT::m_HttpResponse). However, there is no requirement for a replacement method to write to the response stream; some methods will be used in **if** or **while** tags, some will be used to initialize some state, and others may be used purely for debugging purposes. Your particular application will determine the functionality that you add to your replacement methods.

For more information, see the following topics:

- Adding a Replacement Method
- Choosing a Return Value for a Replacement Method
- Passing an Argument to a Replacement Method
- Replacement Method Reference
- Server Response File

See Also

Concepts

ISAPI Extension DLL

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Server Response File

A server response file is a text file containing special tags, also known as a *stencil*.

A server response file (SRF) is a text file with a .srf extension containing static content plus special replacement tags that can be substituted at run time with dynamically generated content.

SRFs offer the following benefits:

- They provide a simple and efficient mechanism for integrating static and dynamic text-based content without writing code.
- They allow you to control the presentation of dynamic content without writing code through the use of intrinsic conditionals and loops and by simple ordering of the replacement tags.
- They allow you to mix dynamic content from multiple sources without writing code.

A stencil is any text file containing special tags understood by CStencil or a derived class. A SRF is a stencil containing tags understood by CHtmlStencil.

For information on the syntax supported by SRFs, see ATL Server Response File Reference.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

SRF Request Architecture

Server Response File Request Handling Architecture

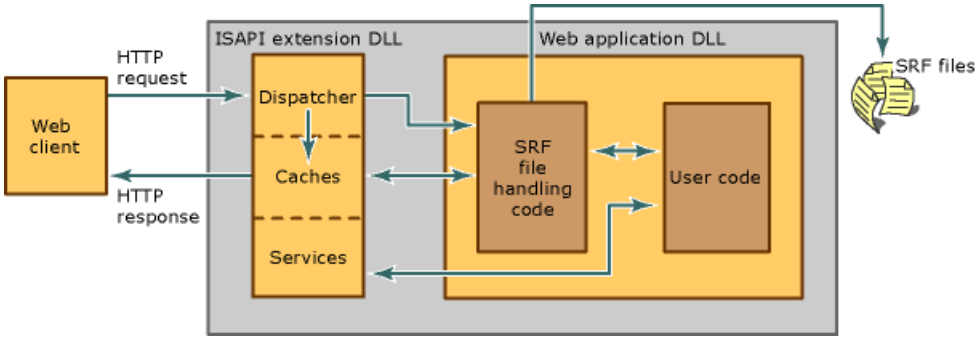

See Also

Concepts

Sequence of Events

Other Resources

ATL Server Architecture

Attributed Request Handler Code

The following code demonstrates the ATL Server request handler attributes. It shows how to write a parse function and associate it with a replacement method. You can see the non-attributed version of the same code in the topic Nonattributed Request Handler Code.

C++ Code

```
#pragma once

[ request_handler("Attributed") ]
class CAttributedHandler
{
public:
    HTTP_CODE ValidateAndExchange()
    {
        // Set the content-type of the response.
        m_HttpResponse.SetContentType("text/html");
        return HTTP_SUCCESS;
    }

protected:
    [ tag_name(name = "Hello") ]
    HTTP_CODE OnHello()
    {
        m_HttpResponse << "Hello World!";
        return HTTP_SUCCESS;
    }

    template <class T>
    HTTP_CODE VariantParse(IAtlMemMgr* pMemoryManager, LPCSTR szArgumentData, T** ppArgumen
t)
    {
        ATLASSERT(pMemoryManager != NULL);
        ATLASSERT(szArgumentData != NULL);
        ATLASSERT(ppArgument != NULL);

        CComVariant varArgument(szArgumentData);

        HRESULT hr = varArgument.ChangeType(CVarTypeInfo<T>::VT);
        if (FAILED(hr))
            return AtlsHttpError(500, ISE_SUBERR_STENCIL_PARSE_FAIL);

        *ppArgument = static_cast<T*>(pMemoryManager->Allocate(sizeof(T)));
        if (!*ppArgument)
            return AtlsHttpError(500, ISE_SUBERR_OUTOFMEM);

        **ppArgument = varArgument.*CVarTypeInfo<T>::pmField;

        return HTTP_SUCCESS;
    }

    HTTP_CODE VariantParseBool(IAtlMemMgr* pMemoryManager, LPCSTR szArgumentData, bool** pp
Argument)
    {
        ATLASSERT(pMemoryManager != NULL);
        ATLASSERT(szArgumentData != NULL);
        ATLASSERT(ppArgument != NULL);

        CComVariant varArgument(szArgumentData);

        HRESULT hr = varArgument.ChangeType(VT_BOOL);
        if (FAILED(hr))
            return AtlsHttpError(500, ISE_SUBERR_STENCIL_PARSE_FAIL);

        *ppArgument = static_cast<bool*>(pMemoryManager->Allocate(sizeof(bool)));
```

```
        if (!*ppArgument)
            return AtlsHttpError(500, ISE_SUBERR_OUTOFMEM);

        **ppArgument = !!V_BOOL(&varArgument);

        return HTTP_SUCCESS;
    }

    [ tag_name(name = "ShowCurrency", parse_func = "VariantParse<CURRENCY>") ]
    HTTP_CODE OnShowCurrency(CURRENCY* pCurrency)
    {
        if (pCurrency)
        {
            m_HttpResponse << "The currency value was: " << *pCurrency;
            // Security note: Using any user supplied value with << is
            // potentially dangerous.
        }
        return HTTP_SUCCESS;
    }

    [ tag_name(name = "ShowBoolean", parse_func = "VariantParseBool") ]
    HTTP_CODE OnShowBoolean(bool* pBoolean)
    {
        if (pBoolean)
        {
            m_HttpResponse << "The boolean value was: " << *pBoolean;
        }
        return HTTP_SUCCESS;
    }

};
```

Server Response File

```
<html>
{{handler ..\bin\Comparison.dll/Attributed}}
<head>
</head>
<body>
<h1>Attributed</h1>
<p>{{Hello}}</p>
<p>{{ShowCurrency($1234.10)}}</p>
<p>{{ShowBoolean(True)}}</p>
<p>{{ShowBoolean(0)}}</p>
</body>
</html>
```

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Nonattributed Request Handler Code

The following code demonstrates the ATL Server request handler macros. It shows how to write a parse function and associate it with a replacement method. You can see the attributed version of the same code in the topic Attributed Request Handler Code.

C++ Code

```
class CNonAttributedHandler :
   public CRequestHandlerT<CNonAttributedHandler>
{
public:
    HTTP_CODE ValidateAndExchange()
    {
        // Set the content-type of the response.
        m_HttpResponse.SetContentType("text/html");
        return HTTP_SUCCESS;
    }

protected:
    HTTP_CODE OnHello()
    {
        m_HttpResponse << "Hello World!";
        return HTTP_SUCCESS;
    }

    template <class T>
    HTTP_CODE VariantParse(IAtlMemMgr* pMemoryManager, LPCSTR szArgumentData, T** ppArgumen
t)
    {
        ATLASSERT(pMemoryManager != NULL);
        ATLASSERT(szArgumentData != NULL);
        ATLASSERT(ppArgument != NULL);

        CComVariant varArgument(szArgumentData);

        HRESULT hr = varArgument.ChangeType(CVarTypeInfo<T>::VT);
        if (FAILED(hr))
            return AtlsHttpError(500, ISE_SUBERR_STENCIL_PARSE_FAIL);

        *ppArgument = static_cast<T*>(pMemoryManager->Allocate(sizeof(T)));
        if (!*ppArgument)
            return AtlsHttpError(500, ISE_SUBERR_OUTOFMEM);

        **ppArgument = varArgument.*CVarTypeInfo<T>::pmField;

        return HTTP_SUCCESS;
    }

    HTTP_CODE VariantParseBool(IAtlMemMgr* pMemoryManager, LPCSTR szArgumentData, bool** pp
Argument)
    {
        ATLASSERT(pMemoryManager != NULL);
        ATLASSERT(szArgumentData != NULL);
        ATLASSERT(ppArgument != NULL);

        CComVariant varArgument(szArgumentData);

        HRESULT hr = varArgument.ChangeType(VT_BOOL);
        if (FAILED(hr))
            return AtlsHttpError(500, ISE_SUBERR_STENCIL_PARSE_FAIL);

        *ppArgument = static_cast<bool*>(pMemoryManager->Allocate(sizeof(bool)));
        if (!*ppArgument)
            return AtlsHttpError(500, ISE_SUBERR_OUTOFMEM);
```

```
        **ppArgument = !!V_BOOL(&varArgument);

        return HTTP_SUCCESS;
    }

    HTTP_CODE OnShowCurrency(CURRENCY* pCurrency)
    {
        if (pCurrency)
        {
            m_HttpResponse << "The currency value was: " << *pCurrency;
            // Security note: Using any user supplied value with << is
            // potentially dangerous.

        }
        return HTTP_SUCCESS;
    }

    HTTP_CODE OnShowBoolean(bool* pBoolean)
    {
        if (pBoolean)
        {
            m_HttpResponse << "The boolean value was: " << *pBoolean;
        }
        return HTTP_SUCCESS;
    }

    BEGIN_REPLACEMENT_METHOD_MAP(CNonAttributedHandler)
        REPLACEMENT_METHOD_ENTRY("Hello", OnHello)
        REPLACEMENT_METHOD_ENTRY_EX("ShowCurrency", OnShowCurrency, CURRENCY, VariantParse<
CURRENCY>)
        REPLACEMENT_METHOD_ENTRY_EX("ShowBoolean", OnShowBoolean, bool, VariantParseBool)
    END_REPLACEMENT_METHOD_MAP()
};

DECLARE_REQUEST_HANDLER("NonAttributed", CNonAttributedHandler)
```

Server Response File

```
<html>
{{handler ..\bin\Comparison.dll/NonAttributed}}
<head>
</head>
<body>
<h1>Non-Attributed</h1>
<p>{{Hello}}</p>
<p>{{ShowCurrency($1234.10)}}</p>
<p>{{ShowBoolean(True)}}</p>
<p>{{ShowBoolean(0)}}</p>
</body>
</html>
```

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Sequence of Events

This topic provides the typical sequence of events in the life of an HTTP request handled by an ATL Server Web application. For a simplified diagram of these events, see SRF Request Architecture.

Typical Sequence of Events

Client

- The client sends an HTTP request to the Web server.

IIS

- IIS looks at the requested URL.
- IIS checks the application mappings for the virtual directory specified in the URL.
- IIS uses ISAPI to pass the request to the ISAPI extension DLL associated with the requested resource.

ISAPI Extension DLL

- The ISAPI extension queues the request on the DLL's thread pool.
- A pool thread retrieves the request from the queue in the order in which it was received.
- The thread calls back into the ISAPI extension's main class.
- The ISAPI extension impersonates the client making the request.
- The ISAPI extension determines whether a response for this request is already in the response cache. If so, that response is returned to the client and request processing stops.
- Otherwise, the ISAPI DLL examines the extension of the requested resource.
- If the requested resource is a SRF:
    - The ISAPI extension checks the server response file cache. If the file is present and up to date, the request handler used to handle the request is obtained from the cache.
    - Otherwise, the ISAPI extension loads the SRF file and reads the handler tag to locate the request handler that should handle the request. The handler tag includes the location of the Web application DLL and the name of the request handler.
- If the requested resource is a DLL:
    - The ISAPI extension locates the request handler that should handle the request by examining the URL. The requested DLL is the Web application DLL. The name of the request handler is given by the *Handler* parameter in the query string.
- The ISAPI extension determines whether the DLL is in the Web application DLL cache. If it is, the cached DLL is used. Otherwise, the Web application DLL is loaded and added to the cache.
- The ISAPI extension calls IRequestHandler::GetFlags to get some initialization data from the request handler.

  The implementation of IIsapiExtension made available to the request handler is selected based on the flags returned by this method. If the request handler specifies the appropriate flags, it will use an implementation that puts the response in the response cache.
- The ISAPI extension calls IRequestHandler::InitializeHandler to initialize the request handler.
- The ISAPI extension calls IRequestHandler::HandleRequest to pass the request to the request handler for processing.

Web Application DLL

- If the requested resource is a SRF, the request handler calls in to the ISAPI extension's SRF cache to see if the SRF has already been parsed.
- If the response file has not already been parsed, the file is parsed, references to replacement methods are resolved, included handlers and subhandlers are loaded, and the parsed file is added to the cache.

During this process, further request handlers, SRFs, and Web application DLLs may be loaded, initialized, and stored in the ISAPI extension's caches.

- When the parsed version of the server response file has been obtained, the file is rendered. The static portions of the file are written to the response stream, and dynamic portions are generated by calling into the replacement methods and by following the control structures in the file.

ISAPI Extension DLL

- When the response is complete or the buffer is full, the ISAPI extension passes the response to IIS.

IIS

- IIS passes the response back to the client.

Client

- The client receives the response.

See Also

Concepts

SRF Request Architecture

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

XML Web Services Created with ATL Server

XML Web services are programmable components that expose functionality through HTTP and SOAP on an intranet or for public consumption on the Internet.

You can create and distribute XML Web services using ASP.NET and Visual Basic or C#, or ATL Server and C++. The topics in this section focus on how to create, distribute, and consume XML Web services using ATL Server and C++.

In This Section

SOAP Request Architecture

Graphically describes the SOAP request architecture.

SOAP Server Code

Provides code that demonstrates the ATL Server SOAP attributes.

SOAP Client Code

Provides code for a SOAP client that demonstrates the use of a SPROXY-generated proxy class.

Providing XML Web Services

Describes creating XML Web services using ATL Server and C++.

Consuming XML Web Services

Discusses using XML Web services from C++.

Related Sections

XML Web Services Overview

Describes what XML Web services are and how they can be used in your applications.

XML Web Services Infrastructure

Discusses the architecture of XML Web services and provides specific details on the protocols and technologies supported.

Creating XML Web Services in Managed Code

Describes creating XML Web services using ASP.NET and Visual Basic or C#.

Accessing XML Web Services in Managed Code

Discusses consuming XML Web services using ASP.NET and Visual Basic or C#.

Programming the Web with XML Web Services

Describes the XML Web services programming model and provides links to topics discussing the fundamentals of XML Web services in Visual Studio.

ATL Server

Provides a conceptual overview of the ATL Server Library, a set of native C++ classes that allows you to create Web applications, XML Web services, and other server applications.

ATL Server References

Provides links to reference documentation for the ATL Server library.

SOAP Request Architecture

SOAP Request Architecture

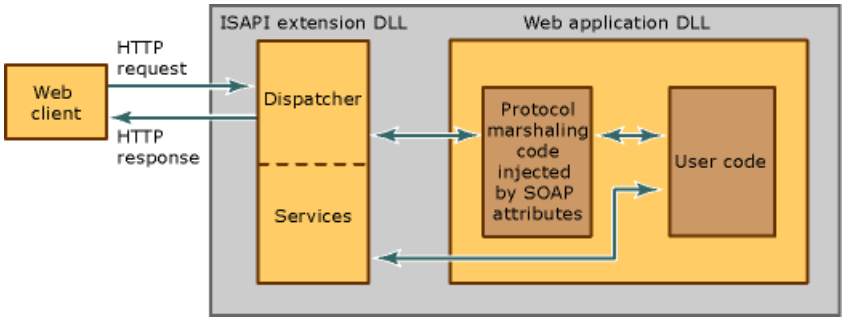

See Also

Concepts

Providing XML Web Services

Other Resources

XML Web Services Created with ATL Server

Programming the Web with XML Web Services

SOAP Server Code

The following code is for an XML Web service that demonstrates the ATL Server SOAP attributes. You can see the client code for this server in the topic SOAP Client Code.

```
namespace HelloSoapService
{
    [
        uuid("2B88FF9E-5B2E-4F75-BD24-8C202EFF8E4D"),
        object
    ]
    __interface IHelloSoapService
    {
        [id(1)] HRESULT Hello([in] BSTR bstrRequestMessage, [out, retval] BSTR* pbstrRespon
seMessage);
    };

    void SafeFreeString(BSTR& bstr)
    {
        if (bstr)
        {
            ::SysFreeString(bstr);
            bstr = 0;
        }
    }

    [
        request_handler(name="Default", sdl="GenHelloSoapServiceWSDL"),
        soap_handler(
            name="HelloSoapService",
            namespace="urn:HelloSoapService",
            protocol="soap"
        )
    ]
    class CHelloSoapService :
        public IHelloSoapService
    {
    public:
        BSTR m_bstrUser;

        CHelloSoapService() : m_bstrUser(0) { }

        ~CHelloSoapService()
        {
            SafeFreeString(m_bstrUser);
        }

        [ soap_method ]
        [ soap_header( value = "m_bstrUser", in = true, out = false ) ]
        HRESULT Hello(/*[in]*/ BSTR bstrRequestMessage, /*[out, retval]*/ BSTR* pbstrRespon
seMessage)
        {
            CComBSTR bstrMessage(L"Hello ");
            bstrMessage += m_bstrUser;
            bstrMessage += L". Your message was received:\n";
            bstrMessage += bstrRequestMessage;

            *pbstrResponseMessage = bstrMessage.Detach();

            return S_OK;
        }
    }; // class CHelloSoapService

} // namespace HelloSoapService
```

See Also

Concepts

SOAP Client Code

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Samples

SOAP Client Code

The following code is for a SOAP client that demonstrates the use of a SPROXY-generated proxy class. You can see the server code for this client in the topic SOAP Server Code.

```
#include "stdafx.h"

#include "HelloSoapService.h"

class CCom
{
private :
    HRESULT m_hr;

public:
    CCom() : m_hr(E_FAIL)
    {
    }

    HRESULT Initialize()
    {
        ATLASSERT(FAILED(m_hr));
        return m_hr = CoInitialize(NULL);
    }

    ~CCom()
    {
        if (SUCCEEDED(m_hr))
            CoUninitialize();
    }

};


int wmain(int argc, wchar_t* argv[])
{
    argc;
    argv;

    CCom COM;
    HRESULT hr = COM.Initialize();
    if (SUCCEEDED(hr))
    {
        using namespace HelloSoapService;

        CHelloSoapService service;
        service.m_bstrUser = CComBSTR(L"Mortimer").Detach();

        CComBSTR bstrResponse;
        HRESULT hr = service.Hello(CComBSTR(L"Hi!"), &bstrResponse);
        if (FAILED(hr))
        {
            std::wcout << L"0x" << std::hex << hr;
            return hr;
        }

        std::wcout << static_cast<const wchar_t*>(bstrResponse);
    }

    return 0;
}
```

See Also

Concepts

SOAP Server Code

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Samples

Providing XML Web Services

XML Web services are programmable components that expose functionality through HTTP and SOAP on an intranet or for public consumption on the Internet.

The topics in this section focus on how to create, test, and distribute XML Web services using ATL Server and C++.

In This Section

Creating XML Web Services

Describes creating XML Web service applications.

Publishing and Deploying XML Web Services

Discusses publishing and deploying XML Web services in Visual Studio.

Testing XML Web Services

Describes how to test XML Web services using Visual Studio

Using Intrinsic Services

Discusses incorporating standard XML Web services into your applications.

Supported Types in XML Web Services Created with ATL Server

Provides a table with the data types supported by the ATL Server SOAP attributes, the XML Schema data type, and C++ data type.

Related Sections

Consuming XML Web Services

Discusses using XML Web services from C++.

XML Web Services Created with ATL Server

Provides links to documentation about XML Web services and ATL Server.

See Also

Other Resources

XML Web Services Created with ATL Server

Creating XML Web Services

To create an XML Web service project

- Create an ATL Server project as described in Creating an ATL Server Project.

  If you create the project using the **ATL Server Web Service** icon, an XML Web service will be added to the Web application DLL by default.

  If you create the project using the **ATL Server Project** icon, you can add an XML Web service to the Web application DLL by selecting **Create as Web Service** on the **Application Options** page of the wizard.

To add an XML Web service to an existing project

- Follow the instructions in Adding an XML Web Service with ATL Server.

  An XML Web service can be added to a wizard-generated ATL Server ISAPI extension DLL or a Web application DLL.

To implement an XML Web service

1. Add methods to the interface in the wizard-generated header file. Use only the marshaling attributes and data types described in Supported Types in XML Web Services Created with ATL Server.
2. Ensure that structures used in the interface methods are exported by applying the export attribute to them.
3. Implement the interface methods in the soap_handler class in the wizard-generated header file. Add the soap_method attribute to each method to tell the compiler that it should be exposed through SOAP.
4. If appropriate, apply the soap_header attribute to one or more SOAP methods in the class.

   Always use the memory manager returned by CSoapRootHandler::GetMemMgr to allocate memory for **[out]** or **[in, out]** parameters and SOAP headers, except BSTRs which must always be allocated with SysAllocString, CComBSTR, or a related function or class.
5. Build the project to create the XML Web service DLL.

Example

You can find example code and further information in the following topics:

SOAP Server Code
Provides code for a simple XML Web service. Client code can be found in the SOAP Client Code topic.

Creating an XML Web Service Using ATL Server
Walks you through the creation of an XML Web service using ATL Server. For comparison, the same XML Web service is also created using Visual Basic, C#, and Managed Extensions for C++. For details, see Creating and Accessing XML Web Services Walkthroughs.

See Also

Tasks

WeatherService Sample: An XML Web Service Created with ATL Server

Concepts

Providing XML Web Services
Consuming XML Web Services

Other Resources

XML Web Services Created with ATL Server

Publishing and Deploying XML Web Services

Deploying an XML Web service created using ATL Server is similar to deploying any other ATL Server application. For deployment during the development process, you can use the Web Deployment Property Page to have your files copied and DLLs automatically registered with Internet Information Services (IIS) on the local machine when your project is built. For general deployment, you should follow the recommendations in and Deploying ATL Server Applications.

Once an XML Web service has been deployed and occasionally before it has been deployed, client applications need to become aware of its existence and the capabilities it offers. This is usually accomplished by providing them with the WSDL description of the XML Web service.

You can expose the WSDL description of your XML Web service in a number of ways:

- Provide the client with a URL from which to obtain the WSDL description directly.

  For an XML Web service created with ATL Server, the WSDL description can be obtained by accessing the WSDL request handler associated with the XML Web service. The name of this request handler is set using the *sdl* parameter of the request_handler attribute.

  For example, the description of a typical XML Web service created with ATL Server and deployed locally can be found using a URL of the following form:

  ```
  http://localhost/Project/Project.dll?Handler=GenServiceWSDL
  ```

- Provide the client with a URL of a discovery document specific to that XML Web service.

  The ATL Server wizards generate a discovery document for each XML Web service that you add to the project. You may need to change the contents of these files to match the location of the XML Web service WSDL description. For details, see XML Web Service Discovery and Enabling Discovery for an XML Web Service.

- Provide the client with a URL of a discovery document for a collection of related XML Web services, such as all the XML Web services provided by your Web site.
- Register your XML Web service in a UDDI registry.
- Send the client the description of the XML Web service by e-mail or some other mechanism of your choice.

See Also

Concepts

Providing XML Web Services

Consuming XML Web Services

Other Resources

XML Web Services Created with ATL Server

Testing XML Web Services

Testing an XML Web service is similar to testing any other C++ application. However, there are a couple of interesting techniques specific to XML Web services created with ATL Server that might help you with the testing process:

- Because of the way in which the code for parsing the XML SOAP messages is hidden by the attributes, it is easy to test the functionality of your XML Web service without having to test the parsing code. You can call the methods exposed by your handler directly.
- The SOAPDebugApp Sample demonstrates a technique for testing your XML Web services without requiring a Web server. It provides code that can be used in your client application to load the Web application DLL containing the XML Web service you are interested in testing. It also contains implementations of IHttpServerContext interface and IIsapiExtension interface for providing the facilities used by your Web application DLL. This technique allows you to run your client and server code in the same process.
- It is a relatively simple process to expose the SOAP handler classes from your Web application DLL as COM classes. This allows you to use any tools you have for automating the testing of COM components against your XML Web services. It will also help you iron out any incorrect assumptions that you've made about marshaling and memory management.
- XML Web service clients can be written in any language. It is worth testing your XML Web service against clients generated using a number of different tools so that you can ensure that clients will have a good experience regardless of the languages and tools they use to create their clients. See Generating an XML Web Service Proxy and XML Web Services Description Language Tool (Wsdl.exe) for information on generating clients for C#, Visual Basic, or JScript.

For debugging advice, see Debugging ATL Server Web Services.

See Also

Tasks

SOAPDebugApp Sample: Debugging XML Web Services in the Client Memory Space

Concepts

Providing XML Web Services

Consuming XML Web Services

Other Resources

XML Web Services Created with ATL Server

Using Intrinsic Services

ATL Server provides a number of ready-made XML Web services for use in your projects. The classes in atlextmgmt.h provide management functionality for configuring the thread pool and caches in your ISAPI extension. You can find more information on these classes in the documentation for Extension Management Services.

The classes in atlsharedsvc.h provide an XML Web service that can be used for caching string data. You can find more information on these classes in the documentation for CSharedCacheHandler.

See Also

Concepts

Providing XML Web Services

Consuming XML Web Services

Other Resources

XML Web Services Created with ATL Server

Supported Types in XML Web Services Created with ATL Server

The table in this topic shows the data types supported by the ATL Server SOAP attributes along with the corresponding XML Schema data type and the C++ data type that SPROXY will use in the SOAP clients that it generates. Types that are supported on the server that result in a slightly different type at the client are italicized:

C++ data type (SOAP Attributes)	**XML Schema data type**	**C++ data type (SPROXY)**
bool	boolean	bool
char	byte	char
__int8	byte	char
unsigned char	unsignedByte	unsigned char
unsigned __int8	unsignedByte	unsigned char
short	short	short
__int16	short	short
unsigned short	unsignedShort	unsigned short
unsigned __int16	unsignedShort	unsigned short
wchar_t	unsignedShort	unsigned short
int	int	int
__int32	int	int
long	int	int
unsigned int	unsignedInt	unsigned int
unsigned __int32	unsignedInt	unsigned int
unsigned long	unsignedInt	unsigned int
__int64	long	__int64
unsigned __int64	unsignedLong	unsigned __int64
double	double	double
float	float	float
BSTR	string	BSTR
ATLSOAP_BLOB	base64Binary	ATLSOAP_BLOB

Enumerations and arrays and structures of these types are also supported by the ATL Server SOAP attributes.

Note the following:

- **BSTR** is the only string type that is supported. Other string types (such as **LPCSTR**) will be represented as arrays or pointers of **byte** or **unsignedShort**.
- An enumeration is represented as string values in accordance with the SOAP specification.
- A member of a structure or array may be an array, a structure, or a simple type. It must not be a pointer. (In other words, pointer members can only be used for conformant arrays that use the size_is attribute).

See Types Supported By SPROXY for the full list of types that can be understood by that tool.

Marshaling Attributes Understood by SOAP Attributes

These attributes are supported by ATL Server for parameters on a method exposed by an XML Web service:

- in
- out
- retval
- size_is

Members of a structure may also use the size_is attribute.

See Also

Tasks

SOAPDataTypes Sample: Demonstrates Basic Data Types and Aggregated Types in a SOAP Server

Concepts

Providing XML Web Services

Consuming XML Web Services

Other Resources

XML Web Services Created with ATL Server

Consuming XML Web Services

XML Web services are programmable components that expose functionality through HTTP and SOAP on an intranet or for public consumption on the Internet.

The topics in this section focus on how to create and test XML Web service clients using C++.

In This Section

Creating XML Web Service Clients

Describes creating XML Web service clients.

Discovering XML Web Services

Discusses discovering XML Web services.

Testing XML Web Service Clients

Provides information about testing XML Web service clients using Visual Studio.

Related Sections

XML Web Services Created with ATL Server

Provides links to documentation about XML Web services and ATL Server.

Providing XML Web Services

Describes creating XML Web services using ATL Server and C++.

See Also

Other Resources

XML Web Services Created with ATL Server

Creating XML Web Service Clients

To create an XML Web service client using C++:

1. On the **Project** menu, click **Add Web Reference** to display the Add Web Reference dialog box.
2. Enter the URL for a discovery document or the WSDL description of the XML Web service.

   For example, the description of a typical XML Web service created with ATL Server and deployed locally can be found using a URL of the following form:

   ```
   http://localhost/Project/Project.dll?Handler=GenServiceWSDL
   ```

3. Select **Add Reference** to have SPROXY generate a header file containing a proxy class for the XML Web service. The header will be automatically added to your project so that you can use Class View to view the classes and methods available to you.
4. Include the SPROXY-generated header in the file or files where you will write the code to access the XML Web service.
5. Make sure COM is successfully initialized before using the XML Web service proxy class.
6. Create an instance of the XML Web service proxy class. Either specialize the SPROXY-generated class template using the appropriate XML Web Service Client Archetype, or use the default typedef.
7. Optionally set proxy settings or use features specific to the proxy class for controlling the connection, authentication, and other settings.
8. Write code to call the methods on the XML Web service.

   Always use the memory manager returned by CSoapRootHandler::GetMemMgr to allocate memory for **[out]** or **[in, out]** parameters and SOAP headers, except BSTRs which must always be allocated with SysAllocString, CComBSTR, or a related function or class.

Example

You can find example code and further information in the following topics:

SOAP Client Code
Provides code for a simple XML Web Service client. Server code can be found in the SOAP Server Code topic.

Extension Management XML Web Service Client
Provides code for an XML Web service client that access the thread pool management service using NTLM authentication.

Creating an XML Web Service Using ATL Server
Walks you through the creation of an XML Web service client using C++. For comparison, the same XML Web service client is also created using Visual Basic, C#, and Managed Extensions for C++. See Creating and Accessing XML Web Services Walkthroughs for details.

See Also

Tasks

WeatherService Sample: An XML Web Service Created with ATL Server

Concepts

Providing XML Web Services
Consuming XML Web Services

Other Resources

XML Web Services Created with ATL Server

Discovering XML Web Services (ATL Server)

Discovering an XML Web service means tracking down its WSDL description. You can find out where to obtain XML Web service descriptions by contacting the provider of the service. You can get some idea of the ways in which XML Web service providers can expose the descriptions for their services by reading Publishing and Deploying XML Web Services.

See Also

Concepts

Providing XML Web Services

Consuming XML Web Services

Other Resources

XML Web Services Created with ATL Server

Testing XML Web Service Clients

Testing an XML Web service client is similar to testing any other C++ application. If the interface of the XML Web service is clearly defined, you can start creating and testing your clients before the code for the service is complete or when the service is unavailable. By replacing the SPROXY-generated proxy with your own non-SOAP implementation, you can further test your clients and check that your assumptions are valid. By running your client against a service with the same interface generated using another tool or language, you can be sure that your client isn't relying on special features of one implementation. See Testing XML Web Services for details on testing the services themselves.

For debugging advice, see Debugging an ATL Server Web Services.

See ATL_SOCK_TIMEOUT for information on avoiding socket timeouts during debugging.

See Also

Tasks

SOAPTransport Sample: Communicates SOAP Messages Over Sockets, MSMQ, the File System, and HTTP Listener

Concepts

Providing XML Web Services

Consuming XML Web Services

Other Resources

XML Web Services Created with ATL Server

SPROXY.EXE: XML Web Service Proxy Generator

SPROXY.EXE is a command line tool that generates native C++ client code for accessing an XML Web service based on a WSDL description.

This tool is used by the Visual C++ project system when a Web reference is added to a native C++ project using the Add Web Reference Dialog Box or when Tool is set to **Web Service Proxy Generator** and Generated Proxy Language is set to **Native C++** in the settings for a file.

```
sproxy [ options ] [ /out:outputfile ] input_location
```

Parameters

options

(Optional) One or more of the following:

Option	Description
/? or /help	Display usage information.
/nologo	Suppress copyright message.
/nowarn	Disable all warnings.
/nopragma	Do not inject "#pragma once" into output file.
/noclobber	Do not overwrite output file if it already exists.
/nonamespace	Do not inject a C++ namespace into the generated proxy file. By default, SPROXY injects a namespace based on the WSDL service name.
/nowchar_t	Emit unsigned short instead of wchar_t for s:unsignedShort. This is a breaking change in Microsoft Visual C++ 2005. Previously, unsigned short was always emitted for s:unsignedShort.
/namespace:<*name*>	Inject a namespace named <*name*> into the generated proxy file. By default, SPROXY injects a namespace based on the WSDL service name. The namespace must be a valid C++ identifier. The /namespace:<*name*> option has the same effect as /nonamespace when <*name*> is empty.
/wsdl <*input_location*>	Process the .wsdl file at the specified path <*input_location*> rather than a .discomap file (default).

outputfile

(Optional) The name of the file to which the generated code will be written. If the file exists, it will be overwritten unless `/noclobber` is specified. If *outputfile* is not specified, SPROXY will create a file in the current directory based on the WSDL service name.

input_location

The location of the file describing the XML Web service for which to generate proxy code. *input_location* can be a URL or file system path to a .discomap or .wsdl file. Specify the `/wsdl` option when you use a .wsdl file.

SPROXY.EXE can also process a results.discomap file. Note that the .discomap file contains a link to a local copy of the .wsdl file and uses a local copy of the schema file.

Remarks

SPROXY.EXE will generate a proxy class template derived from CSoapRootHandler and its template argument. The template

argument needs to conform to the SOAP client archetype and defaults to CSoapSocketClientT<>.

Each SOAP method exposed by the service will be represented by a method in the proxy class. To access the XML Web service, create an instance of the proxy class and call the appropriate method.

SPROXY.EXE can be found in the Microsoft Visual Studio 8\VC\bin directory of your Visual C++ installation. If you are using Visual C++ on a 64-bit operating system, SPROXY.EXE can be found in the Microsoft Visual Studio 8\VC\bin\amd64 directory (x64) or Microsoft Visual Studio 8\VC\bin\ia64 directory (IPF) of your Visual C++ installation.

See Supported Types and Supported Types in XML Web Services Created with ATL Server for a list of types supported by SPROXY.

Note

The SPROXY-generated client requires MSXML3. Before running the client, you will need to install MSXML3 on the machine on which the generated client is installed.

Note

Xmlinst.exe installs MSXML3 in replace mode. Download the Xmlinst.exe Replace Mode Tool from MSDN at http://go.microsoft.com/fwlink/?LinkId=55145 For information about using Xmlinst.exe, see the Knowledge Base article "PRB: Application Errors Occur After You Run Xmlinst.exe on Production Servers" (Q278636). You can find Knowledge Base articles on the MSDN Library CD-ROM or at http://support.microsoft.com/.

Note

The SPROXY-generated header file includes atlsoap.h. This file declares `[emitidl("restricted")];`. If IDL emission is restricted when a `[module]` attribute is encountered, an error will occur. You can enable IDL emission by using `[emitidl("true")];` in your code after including atlsoap.h. (atlextmgmt.h includes the same emitidl declaration as atlsoap.h, so the same issues apply.)

Note

The SPROXY-generated methods do not check pointer parameters for `NULL` before dereferencing. Check pointers for `NULL` before passing them to methods in the XML Web service proxy class.

Sample

The following command line will generate a file, myservice.h, containing C++ proxy code for accessing the XML Web service described by the WSDL obtained from http://myserver/myservice.dll?Handler=GenMyServiceWSDL:

```
sproxy /wsdl http://myserver/myservice.dll?Handler=GenMyServiceWSDL /out:myservice.h
```

The following command line will generate an output file containing C++ proxy code for accessing the XML Web service described by the WSDL obtained from results.discomap (which contains a link to a local copy of the .wsdl file):

```
sproxy results.discomap
```

For more examples of code that uses a proxy class generated by SPROXY, see SOAP Client Code and see the DataSetConsumer Sample.

See Also

Concepts

Providing XML Web Services

Consuming XML Web Services

Other Resources

XML Web Services Created with ATL Server

Supported Types

SPROXY supports the XML Schema data types. This table shows the data types supported by SPROXY. Arrays, structures, and enumerations of these types are also supported.

XML Schema data type	C++ data type (SPROXY)
boolean	bool
byte	char
unsignedByte	unsigned char
short	short
unsignedShort	unsigned short
int	int
unsignedInt	unsigned int
long	__int64
integer	__int64
nonPositiveInteger	__int64
negativeInteger	__int64
unsignedLong	unsigned __int64
nonNegativeInteger	unsigned __int64
positiveInteger	unsigned __int64
decimal	double
double	double
float	float
string	BSTR
hexBinary	ATLSOAP_BLOB
base64Binary	ATLSOAP_BLOB
dateTime	BSTR
time	BSTR
date	BSTR

gMonth	BSTR
gYearMonth	BSTR
gYear	BSTR
gMonthDay	BSTR
gDay	BSTR
duration	BSTR
anyURI	BSTR
ENTITIES	BSTR
ENTITY	BSTR
ID	BSTR
IDREF	BSTR
IDREFS	BSTR
language	BSTR
Name	BSTR
NCName	BSTR
NMTOKEN	BSTR
NMTOKENS	BSTR
normalizedString	BSTR
NOTATION	BSTR
QName	BSTR
token	BSTR

Example

See the SOAPDataTypes Sample.

See Also

Tasks

DataSetConsumer Sample: Consumes Multiple Table Datasets

Concepts

Providing XML Web Services

Consuming XML Web Services

Supported Types in XML Web Services Created with ATL Server

Other Resources

XML Web Services Created with ATL Server

Extension Management Services

ATL Server provides a number of request handlers and XML Web services that allow you to remotely control the behavior of your ISAPI extension using an HTML user interface or by sending SOAP messages.

You can use the management services to control or obtain information about the following aspects of your ISAPI extension:

- Thread pool.
- Server response file cache.
- Web application DLL cache.

The management of each feature can be enabled independently of any other feature by defining certain symbols before including atlextmgmt.h.

You can find a complete list of code elements that support these management services along with descriptions and links to detailed reference topics in the Extension Management Reference.

To enable extension management services

1. Decide whether you want the management services to live in a Web application DLL or ISAPI extension DLL. Either type of DLL will work. The advantage of adding the management services to a Web application DLL is that you can remove the DLL to disable the management functionality while the rest of the site continues to function normally.
2. Decide which services you want to expose and add the necessary symbol definitions to your project:

To manage	Define
Thread pool	_ATL_THREADPOOL_MANAGEMENT
Server response file cache	_ATL_STENCILCACHE_MANAGEMENT
Web application DLL cache	_ATL_DLLCACHE_MANAGEMENT

To disable	Define
Thread pool HTML interface	_ATL_THREADPOOL_NOUI
Thread pool XML Web service	_ATL_THREADPOOL_NOWEBSERVICE
Server response file cache HTML interface	_ATL_STENCILCACHE_NOUI
Server response file cache XML Web service	_ATL_STENCILCACHE_NOWEBSERVICE
Web application DLL cache HTML interface	_ATL_DLLCACHE_NOUI
Web application DLL cache XML Web service	_ATL_DLLCACHE_NOWEBSERVICE

Note
If you want to obtain cache statistics for the server response file cache, you will need to change the *CStencilCacheStats* t emplate parameter used by CIsapiExtension to CStdStatClass or CPerfStatClass.

3. Decide who you want to be able to access the services. Alter the definition for ATL_DEFAULT_AUTHGRP if necessary. By default, only administrators on the local machine can access the management services. Unless you have specialized administrative groups, there should not be any real reason to change this symbol.
4. Include atlextmgmt.h in your project
5. Include atlsrv.rc in your project's resources if not already present.

Note
Wizard-generated ISAPI extension DLLs already include this resource.

6. Build your project

Example

The following code shows how to enable the HTML user interface and XML Web services for managing the thread pool, server response file cache, and web application DLL cache:

```
#define _ATL_THREADPOOL_MANAGEMENT
#define _ATL_STENCILCACHE_MANAGEMENT
#define _ATL_DLLCACHE_MANAGEMENT
#include <atlextmgmt.h>
[emitidl("true")];
```

To access extension management services using the HTML interface

- Navigate to the URL for the HTML user interface. By default, you can find the services at the following URLs if you have added the management functionality to your ISAPI extension DLL:

  Thread pool

  ```
  http://localhost/Project/ProjectIsapi.dll?Handler=ThreadMgrSrf
  ```

  Server response file cache

  ```
  http://localhost/Project/ProjectIsapi.dll?Handler=StencilMgrSrf
  ```

  Web application DLL cache

  ```
  http://localhost/Project/ProjectIsapi.dll?Handler=DllMgrSrf
  ```

To access extension management services using the XML Web services

1. Create a client application.
2. Using the Add Web Reference Dialog to add a SPROXY-generated header to your project. By default, you can find the service descriptions at the following URLs if you have added the management functionality to your ISAPI extension DLL:

   Thread pool

   ```
   http://localhost/Project/ProjectIsapi.dll?Handler=GenThreadPoolManagerWSDL
   ```

   Server response file cache

   ```
   http://localhost/Project/ProjectIsapi.dll?Handler=GenStencilCacheManagerWSDL
   ```

   Web application DLL cache

   ```
   http://localhost/Project/ProjectIsapi.dll?Handler=GenDllCacheManagerWSDL
   ```

3. Write code to create the proxy objects and call the methods on the IThreadPoolMgr, IStencilCacheMgr, and IDllCacheMgr interfaces.

Note

atlextmgmt.h declares `[emitidl("restricted")];`. If IDL emission is restricted when a `[module]` attribute is encountered, a n error will occur. You can enable IDL emission by using `[emitidl("true")];` in your code after including **atlextmgmt.h**. (**a tlsoap.h** includes the same emitidl declaration as **atlextmgmt.h**, so the same issues apply.)

Example

See Extension Management XML Web Service Client.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples
Extension Management Reference

Extension Management XML Web Service Client

The following code shows a simple XML Web service client using a SPROXY-generated proxy class to access the thread pool management service to obtain the number of threads in the ISAPI extension's pool. You can see that the code tells CAtlHttpClient to pass the current user's NTLM authentication credentials automatically by calling CAtlHttpClientT::AddAuthObj and CAtlHttpClientT::NegotiateAuth.

```
// ManagementClient.cpp : Defines the entry point for the console application.
//
// http://machine/project/project.dll?Handler=GenDllCacheManagerWSDL
// http://machine/project/project.dll?Handler=GenThreadPoolManagerWSDL
// typedef ThreadPoolManager::CThreadPoolManagerT<CSoapMSXMLInetClient> CXmlWebService;
// typedef ThreadPoolManager::CThreadPoolManagerT<CSoapWininetClient>   CXmlWebService;

#include "stdafx.h"

#define VCUE_PROXY_NAME _T("proxy_machine")
#define VCUE_SERVICE_URL _T("http://localhost/project/project.dll?Handler=ThreadPoolManager
")

#include <iostream>
#include "ThreadPoolManager.h"

class CCom
{
private :
    HRESULT m_hr;

public:
    CCom() : m_hr(E_FAIL)
    {
    }

    HRESULT Initialize()
    {
        ATLASSERT(FAILED(m_hr));
        return m_hr = CoInitialize(NULL);
    }

    ~CCom()
    {
        if (SUCCEEDED(m_hr))
            CoUninitialize();
    }

};

int _tmain(int argc, _TCHAR* argv[])
{

    CCom COM;
    HRESULT hr = COM.Initialize();
    if (SUCCEEDED(hr))
    {
        // Choose the appropriate proxy type
           typedef CSoapSocketClientT<> CXmlWebServiceClient;
        // typedef CSoapMSXMLInetClient CXmlWebServiceClient;
        // typedef CSoapWininetClient   CXmlWebServiceClient;

        using ThreadPoolManager::CThreadPoolManagerT;
        typedef CThreadPoolManagerT<CXmlWebServiceClient> CXmlWebService;

        // Create an instance of the proxy object.
        CXmlWebService service;
```

```
        // Use features that are unique to the specific proxy class.
        // Respond to 401 or 407 with credentials of current user.
        // This code is specific to CSoapSocketClientT.
        CAtlHttpClient& httpClient = service.m_socket;
        CNTLMAuthObject authCurrentUser;
        httpClient.AddAuthObj(_T("NTLM"), &authCurrentUser);
        httpClient.NegotiateAuth(true);

        // Set the proxy information
        // (Defaults to no proxy)
        hr = service.SetProxy(VCUE_PROXY_NAME, 80);

        if (SUCCEEDED(hr))
        {
            // Set the URL for accessing the service
            // (Defaults to the URL specified in the WSDL description)
            hr = service.SetUrl(VCUE_SERVICE_URL);

            if (SUCCEEDED(hr))
            {
                // Call the XML Web service methods.
                int nThreads = 0;
                hr = service.GetSize(&nThreads);

                if (SUCCEEDED(hr))
                    std::cout << "Threads: " << nThreads << "\n";
                else
                {
                    SOAPCLIENT_ERROR err = service.GetClientError();
                    std::cout << "Client Error: "
                              << err << "\n";
                    switch (err)
                    {
                    case SOAPCLIENT_SOAPFAULT:
                        {
                        const CSoapFault& soapFault = service.m_fault;
                        std::wcout << L"SoapFault Actor:  "
                                  << static_cast<const wchar_t*>(
                                        soapFault.m_strFaultActor
                                        ) << L"\n";
                        std::wcout << L"SoapFault Code:   "
                                  << static_cast<const wchar_t*>(
                                        soapFault.m_strFaultCode
                                        ) << L"\n";
                        std::wcout << L"SoapFault String: "
                                  << static_cast<const wchar_t*>(
                                        soapFault.m_strFaultString
                                        ) << L"\n";
                        std::wcout << L"SoapFault Detail: "
                                  << static_cast<const wchar_t*>(
                                        soapFault.m_strDetail
                                        ) << L"\n";
                        }
                        break;
                    default:
                        std::cout << "HTTP Status Code: "
                                  << service.GetStatusCode() << "\n";
                        std::cout << "HTTP Response -->\n\n";
                        std::cout.write(
                            reinterpret_cast<const char*>(
                                httpClient.GetResponse()
                                ),
                            httpClient.GetResponseLength()
                            );
                        std::cout << "\n\n<-- HTTP Response\n\n";
                        break;
                    }
                }
```

```
            }
        }
    }

    if (FAILED(hr))
        std::cout << "HRESULT: 0x" << std::hex << hr << "\n";

    return 0;
}
```

See Also

Tasks

Creating XML Web Service Clients

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Samples

HTTP Client Services

The primary mission of ATL Server is to provide support for responding to HTTP requests, but ATL Server also provides client-side support. This functionality allows launching of HTTP requests and handles receiving the resulting HTTP responses. With the exception of basic HTTP response header parsing, the ATL Server HTTP client support does not perform any HTTP response parsing or rendering.

In This Section

HTTP Client Overview

Describes HTTP client support in ATL Server.

HTTP Client Tasks

Provides step-by-step instructions to launch an HTTP client request.

Related Sections

HttpClient Sample: HTTP Client Services Demonstration

Demonstrates making HTTP Web request from within an MFC application using ATL Server HTTP client support.

HttpPing Sample: HTTP Client Tool

Demonstrates making HTTP Web request from within a console application using ATL Server HTTP client support.

HTTP Client Reference

Provides links to classes and interfaces that provide HTTP clients in ATL Server.

ATL Server

Provides links to conceptual information about the functionality of ATL Server.

ATL Server Samples

Provides links to samples demonstrating how ATL Server is used.

ATL Server Reference

Provides links to language reference documentation for the ATL Server Library.

See Also

Other Resources

ATL Server Concepts

HTTP Client Overview

ATL Server HTTP client support can be used with other ATL Server components to perform server-side HTTP requests or it can be used outside of an ATL Server application. Used in a server application, HTTP client services can be used to periodically retrieve data from external sites that can then be used to prepare custom content. Used externally to ATL Server, the HTTP client services can, for example, be used to provide the back end to a custom Web browser or a Web crawler.

HTTP client transactions are performed with the CAtlHttpClientT class. This class requires a template argument which is the class that is to provide the underlying network socket support through which HTTP requests are performed. For convenience, the library provides the CAtlHttpClient typedef, which specializes CAtlHttpClientT with a default network socket class.

After construction, instances of the **CAtlHttpClientT** class do not require any initialization steps. HTTP transactions can be launched using one of three Navigate overloads or the NavigateChunked method, which provides support for chunked-transfer encoding. All **Navigate** methods require an HTTP URL as arguments. Optionally, an instance of the CAtlNavigateData class, which can be used to specify options, including socket timeout values, behavior flags, and callback functions, can be provided.

The **Navigate** and **NavigateChunked** methods behave synchronously. When called, they do not return until a response has been fully received, or the request has failed altogether. Intermediate processing can be performed by using the **CAtlNavigateData** class to install callback functions that the **CAtlHttpClientT** class will periodically invoke as the HTTP response is being received. This is particularly useful for large responses such as those containing data or image files.

If a response is received successfully, the **CAtlHttpClientT** methods such as GetHeaderValue and GetRawResponseHeader can be used to inspect the HTTP headers, and the GetBody method can be used to access the response body. The GetStatus method can be used to retrieve the HTTP status code resulting from the request.

Support for HTTP authentication schemes is provided, including ready-to-use classes for the BASIC and NTLM authentication schemes (CBasicAuthObject and CNTLMAuthObject, respectively). The CAtlBaseAuthObject class is provided as a base class for implementing additional authentication schemes.

The **CAtlHttpClientT** and **CAtlNavigateData** classes do not store the password and username information required for authentication. These items are requested from the application on demand through the IAuthInfo interface. Applications must provide an **IAuthInfo** implementation to the CAtlHttpClientT::AddAuthObj method to make username and password information available. If NTLM authentication is used, CNTLMAuthObject will use the current user's credentials if an **IAuthInfo** implementation is not provided.

See Also

Concepts

HTTP Client Services

Other Resources

HTTP Client Reference

ATL Server Concepts

HTTP Client Tasks

This topic describes how you can make an HTTP request.

To make an HTTP request

1. Declare an instance of the CAtlHttpClient class:

```
CAtlHttpClient client;
```

2. If your connection requires the use of a proxy server, use the SetProxy method to specify the proxy server settings:

```
client.SetProxy( "myproxyserver", 80 );
```

3. Declare an instance of the CAtlNavigateData class:

```
CAtlNavigateData navData;
```

4. Optionally, use the AddFlags or the RemoveFlags method to override default behavior specified within the **CAtlNavigateData** constructor:

```
// This disables auxiliary processing such as
// the automatic handling of redirected URLs
// and authorization of protected URLs.
navData.RemoveFlags( ATL_HTTP_FLAG_PROCESS_RESULT );
// This restores the default behavior
navData.AddFlags( ATL_HTTP_FLAG_PROCESS_RESULT );
```

5. Optionally, a **read** callback function can be installed with the CAtlNavigateData::SetReadStatusCallback method that will be invoked periodically as a response is being received:

```
navData.SetReadStatusCallback( ReadCallbackFunc, 0 );
```

6. The actual HTTP transaction can be launched using one of three CAtlHttpClient::Navigate overloads:

```
// This version of Navigate takes a URL in a string from
// optional navigate data.
client.Navigate( "http://www.microsoft.com", &navData );
```

7. If the Navigate method returns true, the response status can be checked using the CAtlHttpClient::GetStatus method:

```
if (client.GetStatus() != ATL_INVALID_STATUS)
{
   // handle HTTP response
}
```

See Also

Concepts

HTTP Client Services

Other Resources

HTTP Client Reference

ATL Server Concepts

Session-State Services (ATL Server)

ATL Server's session-state support allows client-specific data to be stored across multiple requests. Sessions, created and used to store state variables, can be accessed during the handling of subsequent requests and used to retrieve and/or modify session variables.

In This Section

Session-State Support

Discusses using session-state support from ATL Server request handler objects.

Session-State Services

Describes the two implementations for session-state management in ATL Server.

Session-State Tasks

Covers adding session-state support to an existing ATL Server ISAPI extension DLL.

Related Sections

SessionSettings Sample

Demonstrates how to use session state to maintain user preferences across multiple client requests.

Session-State Reference

Provides links to attributes, classes, and macros that let you expose performance monitoring from your applications and services.

ATL Server

Provides links to conceptual information about the functionality of ATL Server.

ATL Server Samples

Provides links to samples demonstrating how ATL Server is used.

ATL Server Reference

Provides links to language reference documentation for the ATL Server Library.

Session-State Support

This topic addresses how to use session-state support from ATL Server request handlers. The following topics apply to applications that will be used with an ISAPI extension DLL that includes ATL session-state support. The solution files and source code necessary for session-state-enabled extension DLLs can be generated easily using the ATL Server Project Wizard.

- Session-State Interfaces
- Session-State Data Access
- Session Identification
- Session Data Storage and Retrieval
- Session Timeouts
- Implementation Differences

For instruction on how to add support to an existing ISAPI DLL created without session-state services or help to modify session service settings, see Session-State Tasks.

Session-State Interfaces

Session objects, along with the data contained by a session, are manipulated from ATL Server request handlers using these interfaces:

Interface	Description
ISessionStateControl	Can be used to set and retrieve global session timeout settings, and to determine the number of currently active sessions present on the server.
ISessionStateService	Used to create new sessions, retrieve pointers to existing sessions, and terminate sessions that are no longer needed.
ISession	Provides methods for the storage and retrieval of user-specific data and timeout settings for a single session.

Session-State Data Access

The ATL Server Project Wizard, when used to create session-state-enabled application and ISAPI DLL projects, adds code to the default request handler that performs the retrieval of an ISessionStateService interface pointer. This code includes an **ISessionStateService** pointer and, within the request handler's ValidateAndExchange method, a call to the IServiceProviderImpl::QueryService method, which attempts the initialization of the pointer. Once a valid **ISessionStateService** interface pointer has been retrieved, it can be used to create new sessions and retrieve existing ones with the CreateNewSession and GetSession methods, respectively.

See Session-State Tasks for step-by-step coverage of how to access session-state data.

Session Identification

Each session is identified by a session ID. The ISessionStateService::CreateNewSession method, in addition to initializing an **ISession** pointer, returns an ID to identify the new session. This session ID must be communicated to the client that the new session corresponds to so that subsequent client requests (which should also include the session ID) can be matched to the correct session data.

ATL Server provides support for communicating session IDs between client and server. When a new session is created and a new session ID is generated, the ID can be sent to the client as part of the server response using the CHttpResponse::AppendCookie method. Likewise, a client's session ID can be retrieved during subsequent requests via the CHttpRequest::GetSessionCookie method. Instances of the CHttpResponse and CHttpRequest classes are accessible through the CRequestHandlerT base class from which ATL Server request handlers are derived.

Session Data Storage and Retrieval

Once a valid ISession interface pointer has been retrieved using the ISessionStateControl interface (either to create a new

session or to retrieve an existing one), session variables can be stored, modified, retrieved, and removed using the **ISession** methods.

Each session variable added to the session includes a name, in the form of a string, and a value, in the form of a VARIANT. The ISession::SetVariable method can be used to create new variables and to modify existing ones. Variable values can be retrieved using the ISession::GetVariable method given the variable name, or enumerated using the ISession::BeginVariableEnum, ISession::GetNextVariable, and ISession::CloseEnum methods.

Despite the fact that session-state variables are accessed with the same API regardless of the underlying session-state implementation, there are some important differences between the database and memory-based session-state services. See Implementation Differences.

Session Timeouts

Each session has a timeout value, expressed in milliseconds. A session "times out," or expires, if an amount of time greater than the timeout value elapses. Sessions can be checked for expiration using the ISession::IsExpired method.

A session's timeout value (which defaults to 600,000 milliseconds, or 10 minutes) can be set using the ISession::SetTimeout method. Alternately, the ISessionStateControl::SetSessionTimeout method can be used to set the timeout value for all current and future sessions.

It is the responsibility of the application to call one of the **ISession** methods to maintain the life of the session and restart the timeout (just obtaining the session object does not restart the timeout). Once the session has not been accessed for the period specified by the timeout, the session object will expire.

Depending on the underlying session-state implementation, session timeout values can affect session availability because expired sessions may be removed from storage. ATL Server memory-backed session-state support, for example, removes expired sessions automatically, and the database-backed implementation does not.

Implementation Differences

The ISessionStateControl, ISessionStateService, and ISession interfaces form the interface to ATL Server session-state support regardless of the underlying implementation. There are some important differences, however, between the two implementations provided with ATL Server.

- The memory-backed session-state implementation, while providing optimal performance, cannot be used in multiple-machine server configurations.
- The database-backed implementation requires extra maintenance because expired and discarded session data is not removed from the database unless the session data is explicitly deleted using the ISession::RemoveAllVariables method.
- Before the database-backed support can be used, database tables must be configured. Use the **SessonServices.sql** file in the \vc7\bin installation directory to automatically configure these tables.

The CComVariant class is used to write session-state values in the database implementation. This has two ramifications:

- The data types that can be used as session variables are limited to the types supported by CComVariant::WriteToStream. If objects are to be persisted, they must implement the IPersistStream or IPersistStreamInit interface.
- The CComVariant class stores additional data in the database, including a data type ID and the length of the data in the case of strings. As a result, session variable names of length MAX_SESSION_KEY_LEN and session-state variables of size MAX_VARIABLE_VALUE_LENGTH will not be stored properly. To avoid this limitation, restrict the sizes of these values by at least 8 bytes to accommodate the CComVariant overhead.

See Also

Other Resources

Session-State Services (ATL Server)

Session-State Services in ATL Server

ATL Server provides two separate implementations for session-state management. One implementation uses memory to store session-state data, and the other implementation uses a database. If neither implementation meets your needs, you can design a custom implementation that works seamlessly through the provided session-state interfaces. Regardless of implementation, the ISAPI extension DLL provides session-state support.

Within ISAPI extension DLL code, session-state support takes the form of a CSessionStateService instance, which is added as a data member to a CIsapiExtension-derived class. The specific session-state implementation (memory, database, or custom) is determined by a template argument passed to the **CSessionStateService** class. For memory-based session support, use the CMemSessionServiceImpl class. For database support, use the CDBSessionServiceImpl class. The following example code declares an instance of **CSessionStateService** that uses the database-backed session-state implementation:

```
typedef CSessionStateService<CWorkerThread<>, CDBSessionServiceImpl> sessionSvcType;
CComObjectGlobal<sessionSvcType> m_SessionStateSvc;;
```

A custom implementation requires you to define the class that you provide as the template argument used to specialize **CSessionStateService**. Regardless of the implementation class used, the resulting **CSessionStateService** object is used by the **CIsapiExtension**-derived class to service session-state requests made by application DLLs. See Session-State Tasks for instruction on how to add session-state support to ISAPI extension DLLs that were not generated with built-in session-state support.

Writing Custom Implementations

To provide a custom session-state implementation, write a class that supports these methods:

- `HRESULT CreateNewSession(LPSTR szNewID, DWORD *pdwSize, ISession** ppSession);`

  The **CreateNewSession** method will be called whenever an application DLL requests a new session. The arguments are identical to those of the ISessionStateService::CreateNewSession method.

- `HRESULT GetSession(LPCSTR szID, ISession **ppSession);`

  The **GetSession** method will be called whenever an application DLL requests an existing session. The arguments are identical to those of the ISessionStateService::GetSession method.

- `HRESULT CloseSession(LPCSTR szID);`

  The **CloseSession** method will be called whenever an application DLL requests that a session be removed. The single argument is the session ID.

- `void SweepSessions();`

  ATL Server calls the **SweepSessions** method periodically to provide an opportunity for the implementation to purge expired sessions. Depending on your implementation, it may be acceptable to provide only an empty method body.

- `HRESULT SetSessionTimeout(unsigned __int64 nTimeoutMS) throw();`

  The **SetSessionTimeout** method will be called when the application DLL specifies a timeout value to be used for all sessions.

- `HRESULT GetSessionTimeout(unsigned __int64* pnTimeout);`

  The **GetSessionTimeout** method should provide the global session timeout value.

- `HRESULT GetSessionCount(DWORD *pnCount);`

  The **GetSessionCount** method should provide the number of sessions that are currently active.

- `void ReleaseAllSessions() throw();`

  The **ReleaseAllSessions** method should remove all sessions. This method is called immediately before the ISAPI extension DLL terminates.

- `HRESULT Initialize(void*, DWORD, IServiceProvider*);`

The **Initialize** method is called when the ISAPI extension DLL starts and should be used to initialize implementation-specific data structures. The **Initalize** method arguments are implementation specific. The ATL Server database session-state implementation, for example, uses these arguments to provide the implementation with database connection information.

See Also

Tasks

SessionSettings Sample: Session State Demonstration

Reference

ShowSession

Concepts

Session-State Tasks

Other Resources

Session-State Services (ATL Server)

ATL Server Samples

ATL Server Reference

Session-State Tasks

To add session-state support to an existing ATL Server ISAPI extension DLL

1. Add an instance of the CSessionStateService class to the CIsapiExtension-derived class. The following example code declares a data member called **m_SessionStateSvc**:

```
protected:
typedef CSessionStateService<CWorkerThread<>, CMemSessionServiceImpl> sessionSvcType;
CComObjectGlobal<sessionSvcType> m_SessionStateSvc;
```

   The CMemSessionServiceImpl template argument indicates that the memory-backed session-state implementation will be used. Alternately, the CDBSessionServiceImplT class can be used for database-backed support.

2. Modify the existing GetExtensionVersion method, or, if one does not exist, override the base class version. The **GetExtensionVersion** method should initialize the **CSessionStateService** object by calling its **Initialize** method, like this:

```
BOOL GetExtensionVersion(HSE_VERSION_INFO* pVer)
{
   if (!baseISAPI::GetExtensionVersion(pVer))
   {
      return FALSE;
   }
   if (S_OK != m_SessionStateSvc.Initialize(&m_WorkerThread,
                                 static_cast<IServiceProvider*>(this)))
   {
      TerminateExtension(0);
      return FALSE;
   }
   return TRUE;
}
```

3. Ensure that the TerminateExtension method is overridden, and modify it so that it terminates the session service object using the CSessionStateService::Shutdown method:

```
BOOL TerminateExtension(DWORD dwFlags)
{
   BOOL bRet = baseISAPI::TerminateExtension(dwFlags);
   m_SessionStateSvc.Shutdown();
   return bRet;
}
```

4. Override the QueryService method to support the retrieval of the ISessionStateService interface:

```
HRESULT STDMETHODCALLTYPE QueryService(REFGUID guidService,
         REFIID riid, void** ppvObject)
{
   if (InlineIsEqualGUID(guidService, __uuidof(ISessionStateService)))
         return m_SessionStateSvc.QueryInterface(riid, ppvObject);
   return baseISAPI::QueryService(guidService, riid, ppvObject);
}
```

To use session-state in an ATL Server request handler

1. Include the AtlSession.h header file in your project:

```
#include <atlsession.h>
```

2. Add two data members to the request handler class for use in manipulating the ISessionStateService and ISession interfaces:

```
CComPtr<ISessionStateService> m_spSessionService;
CComPtr<ISession> m_spSession;
```

3. Within the request handler's ValidateAndExchange method, request the ISessionStateService interface from the ISAPI extension DLL using the IServiceProviderImpl::QueryService method (this call will fail if the ISAPI extension DLL does not include session-state support):

```
HRESULT hr = m_spServiceProvider->QueryService(__uuidof(ISessionStateService), &m_spSe
ssionService);
```

(Use steps 4 and 5 if an existing session is to be retrieved.)

4. Retrieve the session cookie from the client request object. In this example, the ATL Server SESSION_COOKIE_NAME macro is used for the session cookie name:

```
m_HttpRequest.Cookies(SESSION_COOKIE_NAME).GetValue(sessionID)
```

5. If a session cookie is found, use it to attempt the retrieval of an existing session using the ISessionStateService::GetSession method. If the session cookie length is zero, no session cookie is present in the client request, so no session can be retrieved:

```
if (sessionID.GetLength())
   hr = m_spSessionService->GetSession(sessionID, &m_spSession);
```

(Use steps 6 and 7 if a new session is to be created)

6. Use the ISessionStateService::CreateNewSession method to create a new session along with a corresponding session ID:

```
const size_t nCharacters = 64;
CHAR szID[nCharacters + 1];
szID[0] = 0;
DWORD dwCharacters = nCharacters;
hr = m_spSessionService->CreateNewSession(szID, &dwCharacters, &m_spSession);
```

This code demonstrates how to create a session ID that is less than or equal to 64 characters in length.

7. If the call to CreateNewSession succeeds, attach the new session ID to the server response using the CHttpResponse::AppendCookie method:

```
if (SUCCEEDED(hr))
{
   CSessionCookie theSessionCookie( szID );
   m_HttpResponse.AppendCookie( &theSessionCookie );
}
```

8. Either by creating a new session, or retrieving an existing one, the **m_spSession** pointer is now initialized and ready for use in storing or retrieving session data. Use the ISession::SetVariable method to add or modify session data and the ISession::GetVariable method to retrieve data. See the SessionSettings Sample for an example of using session data

storage.

To convert from memory to database-backed session-state support

1. Modify the CSessionStateService declaration within your ISAPI extension DLL project to use the desired session-state implementation. Use CMemSessionServiceImpl for memory support and CDBSessionServiceImplT for database support.

2. Change the arguments passed to the CSessionStateService::Initialize method by the GetExtensionVersion method to reflect the underlying scheme.

   For memory-backed support, the **Initialize** arguments should look like this:

   ```
   m_SessionStateService.Initialize(&m_WorkerThread,NULL);
   ```

   For database backed session-state support, the data source must be specified:

   ```
   const char pszConnection[] =
            "Provider=SQLOLEDB.1;"
            "Initial Catalog=ATLServer;"
            "Data Source=YourServer"
            ;
   m_SessionStateService.Initialize(
         &m_WorkerThread,
         (IServiceProvider*)this,
         ATL_SESSION_TIMEOUT,
         const_cast<void*>(static_cast<const void*>(pszConnection)),
         lstrlen(pszConnection) + 1);
   ```

See Also

Other Resources

Session-State Services (ATL Server)
ATL Server Concepts
ATL Server Reference
ATL Server Samples

Performance Monitoring

The Windows operating system provides support for application performance monitoring. Using this feature, applications can expose critical performance values, making it possible to track application performance in real time.

Traditionally, performance monitoring has been performed using either the PDH (Performance Data Helper) API or the Registry Performance Interface. ATL Server provides a third alternative.

In This Section

Performance Monitoring Concepts

Discusses performance monitoring with ATL Server.

Performance Classes and Attributes

Describes using performance components with attributed or nonattributed code.

Server Performance Monitoring

Covers performance monitoring for Web server applications.

Predefined ISAPI Performance Counters

Describes tracking and exposing ISAPI performance data.

Enabling ISAPI Performance Monitor Support

Provides step-by-step instructions to enable ISAPI performance monitoring support in a new or existing project.

Initializing and Accessing Performance Monitor Objects

Provides step-by-step instructions on initializing, accessing, and uninitializing performance monitoring objects.

Manually Defining Performance Objects

Provides step-by-step instructions for adding performance objects to a project without using code wizards.

Related Sections

PerformanceCounter Sample

Demonstrates how to expose performance information using ATL Server performance monitoring support within a Web application.

PerformanceScribble Sample

Demonstrates how to expose performance information using ATL Server performance monitoring support within an MFC application.

Performance Monitoring Reference

Provides links to attributes, classes, and macros that let you expose performance monitoring from your applications and services.

ATL Server

Provides links to conceptual information about the functionality of ATL Server.

ATL Server Samples

Provides links to samples demonstrating how ATL Server is used.

ATL Server Reference

Provides links to language reference documentation for the ATL Server Library.

See Also

Other Resources

ATL Server Concepts

Performance Monitoring Concepts

Performance monitoring requires the cooperation of two applications:

- The application being monitored must define and expose performance values.
- The application doing the monitoring must be able to locate and retrieve the data.

Both tasks can be performed using the Win32 performance monitoring APIs, but only the first (exposing performance data) is addressed by ATL Server. The Windows Performance Console, however, can be used to monitor performance data originating from any performance-enabled application, so it is not typically necessary to develop a separate application for performance monitoring.

ATL provides three performance monitor constructs, all of which are required to expose performance data:

- Performance object manager
- Performance object
- Performance counter

The performance object manager is responsible for exposing the application's performance objects and counters, making external monitoring possible. The manager object is also responsible for granting the application access to performance objects. Performance objects and counters are implemented using shared memory, which the object manager is responsible for configuring. A manager object can manage any number of performance objects.

The performance object itself represents some or all of the application's performance data. Each performance object has a human-readable name that identifies it to performance monitoring applications such as the Windows Performance Console. Performance objects can be instanceless, meaning that only one instance exists, regardless of how many application instances are running. Conversely, each application instance can expose one or more instances of the performance object and can therefore be monitored separately. A performance object can contain any number of counters.

Performance counters represent the actual performance data being monitored. Like performance objects, each counter has a human-readable name. The meaning of each counter varies according to intent, but generally counter values represent either a value averaged over time or a simple raw value. Performance counters can also take the form of a string. For details, see Counter Data Types. Counters should always be accessed in a thread-safe manner, because performance counters exist in shared memory and are inspected by performance monitoring applications.

See Also

Tasks

PerformanceCounter Sample: Performance Monitoring in an ATL Server Application

PerformanceScribble Sample: Performance Monitoring in an MFC Application

Concepts

Performance Monitoring

Other Resources

Performance Monitoring Reference

Performance Classes and Attributes

ATL performance object managers, performance objects, and performance counters can be added to ATL applications directly or with code wizards. For details, see Adding an ATL Performance Monitor Object. In either case, these performance components can be defined using attributed or nonattributed code. The following table shows the programming constructs required for both cases:

Performance feature	Attributed	Nonattributed
Performance object manager	perfmon	CPerfMon
Performance object	perf_object	CPerfObject
Performance counter	perf_counter	DEFINE_COUNTER

See Manually Defining Performance Objects for instruction on adding nonattributed performance monitoring constructs.

Regardless of how you add performance monitor objects and whether they are attributed or nonattributed, you next add code to initialize and terminate performance monitor support within your DLL. The performance object management class, whether defined using the perfmon attribute or the CPerfMon class, provides member functions that should be called when your DLL is loaded and unloaded. In addition, instances of performance objects must be created using the performance manager object. For more information, see Performance Monitor Object Initialization and Access.

With performance monitor objects present and initialized within your ATL DLL, you can add the code that updates actual performance counters. For more information, see Performance Monitor Object Initialization and Access.

See Also

Tasks

PerformanceCounter Sample: Performance Monitoring in an ATL Server Application

PerformanceScribble Sample: Performance Monitoring in an MFC Application

Concepts

Performance Monitoring

Other Resources

Performance Monitoring Reference

Server Performance Monitoring

Performance is critical for Web server applications that are heavily used, and performance data is essential for optimizing Web server applications. Before ATL Server, this usually meant building custom performance tracking features into Web server applications.

ATL Server performance monitor support, through the Windows operating system native performance tracking functionality, allows applications to expose crucial performance data values to external applications, enabling sophisticated yet unobtrusive performance analysis.

ATL Server provides performance support at two levels:

- With predefined ISAPI performance counters.
- With custom performance object support.

ATL Server provides optional ISAPI request performance data. This feature can be activated manually or with the ATL Server Project Wizard during project creation. For more information, see Predefined ISAPI Performance Counters. See ATL Performance Monitoring to learn how to add custom performance objects to ATL Server applications.

See Also

Tasks

PerformanceCounter Sample: Performance Monitoring in an ATL Server Application

PerformanceScribble Sample: Performance Monitoring in an MFC Application

Other Resources

Performance Monitoring Reference

Predefined ISAPI Performance Counters

ATL Server applications can be instructed to track and expose ISAPI performance data. For details, see Enabling ISAPI Performance Monitor Support. This activates a predefined ATL performance monitor object hosted by the CIsapiExtension class. The result is a performance monitoring object named **ATL Server:Request** that includes the following counters:

- Active Threads
- Average Response Time
- Current Queued Requests
- Maximum Queued Requests
- Server Failed Requests
- Server Requests per second
- Server Total Requests

Like all performance object counters, these counter values can be viewed with the Windows Performance Console or programmatically with the Win32 performance monitoring APIs.

Each ATL Server ISAPI extension DLL uses two instances of the ISAPI performance objects, one for reporting global statistics and one for per-instance usage. Note that the per-instance objects use the path of the ISAPI DLL to identify each instance. The result is that multiple applications using the same ISAPI DLL will generate combined statistics. For application-specific ISAPI performance data, use separate copies of the ISAPI DLL, each with a different file name or in a different directory.

For information on defining custom performance objects, see Performance Classes and Attributes.

See Also

Tasks

PerformanceCounter Sample: Performance Monitoring in an ATL Server Application

PerformanceScribble Sample: Performance Monitoring in an MFC Application

Concepts

Performance Monitoring

Other Resources

Performance Monitoring Reference

Enabling ISAPI Performance Monitor Support

To enable optional ISAPI performance monitoring support during project creation with the ATL Server Project Wizard

1. On the **File** menu, click **New**, and then click **Project**.

   The **New Project** dialog box appears.

2. In the Project Types pane, click **Visual C++ Projects**, and in the Templates pane, click **ATL Server Project**.

3. Enter a project name and click **OK**.

   The ATL Server Project Wizard appears.

4. Click the **Server Options** tab.

5. Select **Predefined performance counters**.

6. Complete the remainder of the project options, and click **Finish**. A new ATL Server project will be created with ISAPI performance data activated.

To enable optional ISAPI performance monitor object in an existing application

1. In the ISAPI extension code, change or add the second **CIsapiExtension** class template argument to **CPerfMonRequestStats**.

2. Use the PERFREG_ENTRY macro to register the **CRequestPerfMon** class:

   ```
   PERFREG_ENTRY(CRequestPerfMon)
   ```

3. Define _ATL_PERF_REGISTER.

See the PerformanceCounter Sample: ATL Server Performance Monitoring for an example.

See Also

Concepts

Performance Monitoring

Other Resources

Performance Monitoring Reference

Initializing and Accessing Performance Monitor Objects

To initialize performance objects

1. Declare a global instance of your performance object manager class.
   - For attributed code, this is a class declared with the perfmon attribute.
   - For nonattributed code, this is a CPerfMon-derived class.)
2. In a function suited for initialization (called during or shortly after DLL loading, for example), call the CPerfMon::Initialize member on the registrar object.

To access performance objects

1. After the **Initialize** function call, lock the **CPerfMon** object using the CPerfLock class.
2. Ensure that the lock succeeded by using the CPerfLock::GetStatus function.
3. Use the CPerfMon::CreateInstanceByName function to initialize an instance of each performance object necessary.

   The **CPerfObject** is unlocked automatically when the **CPerfLock** instance goes out of scope.
4. The performance object counters, which are public data members of the performance object, can now be updated but must be accessed in a thread-safe manner, such with the InterlockedIncrement function.

To uninitialize performance objects

- In a function suited for termination (called immediately before the DLL is unloaded), call CPerfMon::UnInitialize member function on the **CPerfMon** derived object.

See Also

Tasks

PerformanceCounter Sample: Performance Monitoring in an ATL Server Application

Concepts

Performance Monitoring

Other Resources

Performance Monitoring Reference

Manually Defining Performance Objects

To add ATL performance objects to a project without code wizards

1. Define a performance object class derived from the CPerfObject class:

```
class MyPerfObject :
   public CPerfObject
{
public:
};
```

2. Use the DECLARE_PERF_OBJECT macro inside the new class body to indicate the class name, numeric ID (which is arbitrary), description string, and help string for the new performance object:

```
class MyPerfObject :
   public CPerfObject
{
public:
   DECLARE_PERF_OBJECT(MyPerfObject, 1, "MyPerfObjectName", "my object help string", -
1);
};
```

3. Add a data member (typically of type **ULONG**) to the class for each counter that the new performance object will expose. Then use the DEFINE_COUNTER macro to indicate each counter's data member name, description string, help string, counter type, and scale (normally 0). The final performance object class definition can look like this:

```
class MyPerfObject :
   public CPerfObject
{
public:
   DECLARE_PERF_OBJECT(MyPerfObject, 1, "MyPerfObjectName", "my perf object help strin
g", -1);
   BEGIN_COUNTER_MAP(MyPerfObject)
      DEFINE_COUNTER(myCounter, "myCounterName",
                  "my counter help string", PERF_COUNTER_RAWCOUNT, 0)
   END_COUNTER_MAP()
   ULONG myCounter;
};
```

   Note that the DEFINE_COUNTER macro usages must be bracketed by the BEGIN_COUNTER_MAP and END_COUNTER_MAP macros, but the data member should appear outside the counter map.

4. Use the CPerfMon class to derive a performance object manager class:

```
class MyPerfMonManager :
   public CPerfMon
{
public:
};
```

5. Use the CHAIN_PERF_OBJECT to assign each performance object to be managed by the new CPerfMon-derived class. The final performance object management class might look like this:

```
class MyPerfMonManager :
   public CPerfMon
{
public:
#define Perf_MyPerfMonManager _T("Perf_MyPerfMonManager")
   BEGIN_PERF_MAP(Perf_MyPerfMonManager)
      CHAIN_PERF_OBJECT(MyObject)
   END_PERF_MAP()
};
```

Multiple performance objects can be managed by a single performance object management class with multiple usages of the CHAIN_PERF_OBJECT macro. All such usages must be bracketed by the BEGIN_PERF_MAP and END_PERF_MAP macros.

6. Use the PERFREG_ENTRY macro to register the new performance object manager class:

```
PERFREG_ENTRY(MyPerfMonManager);
```

7. Define _ATL_PERF_REGISTER.

See Also

Tasks

PerformanceCounter Sample: Performance Monitoring in an ATL Server Application

PerformanceScribble Sample: Performance Monitoring in an MFC Application

Concepts

Performance Monitoring

Other Resources

Performance Monitoring Reference

Caching

Server performance can typically be improved with the use of caching to speed the retrieval of frequently used resources. ATL Server provides support for caching files, DLLs, stencils, data sources, and generic blocks of memory. This caching support can be customized to a high degree, allowing the configuration of automated cache culling mechanisms and memory usage restrictions. These cache implementations are thread safe and track effectiveness statistics by default.

In This Section

Caching Overview

Describes the cache implementations that can be used to improve performance.

Adding Cache Support

Provides step-by-step instructions on adding cache support to an ATL Server project.

Related Sections

BlobCache Sample

Demonstrates the use of the **CBlobCache** class from within an ATL Server application.

Caching Reference

Provides links to classes and interfaces that allow for caching in ATL Server.

ATL Server

Provides links to conceptual information about the functionality of ATL Server.

ATL Server Samples

Provides links to samples demonstrating how ATL Server is used.

ATL Server Reference

Provides links to language reference documentation for the ATL Server Library.

Caching Overview

ATL Server provides several cache implementations that can be used to boost performance by caching frequently used items. Five memory-based cache classes are provided:

CMemoryCacheBase

The **CMemoryCacheBase** class provides generic cache support and serves as the base class for the remaining four classes.

CMemoryCache

The **CMemoryCache** class extends the functionality of **CMemoryCacheBase** to include support of associating memory blocks with DLLs and allows the use of custom memory release mechanisms.

CBlobCache

The **CBlobCache** class is derived from **CMemoryCache** but specializes on the **void*** type, making it ideal for caching arbitrarily sized memory blocks, where **CMemoryCacheBase** and **CMemoryCache** both cache fixed-size memory types.

CFileCache

The **CFileCache** class caches file information but does not store file content; it stores file names and lengths. The cached files remain on disk and are managed by the cache object only in the sense that they are deleted from disk when they are removed from the cache. It is therefore very important to use disposable files when using this class. This class, like **CStencilCache**, is used internally by ATL Server.

CStencilCache

The **CStencilCache** class is the most specialized of these classes, because it caches preprocessed ATL Server stencils. This class, like **CFileCache**, is used internally by ATL Server.

Template Parameters

These classes use template parameters extensively to allow customization. Automated maintenance, cache-culling criteria, thread safety, and usage statistics are all managed through template arguments.

MonitorClass

A class that performs periodic, asynchronous cache culling, typically CWorkerThread.

DataType

The data type to be cached.

SyncClass

A class that is used for thread synchronization, typically CComCriticalSection.

FlushClass

A class that determines the order in which items are removed from the cache once a memory or item count limitation has been reached. See Cache Flusher Archetype.

CullClass

A class that determines if and when cache items expire and are therefore removed. See Cache Culler Archetype.

StatClass

A class that tracks cache effectiveness statistics, such as hit and miss count. See Cache Statistics Archetype.

These template parameters are accepted by the cache classes as outlined by this table:

Template parameters	Monitor	DataType	Sync	Flush	Culler	Stats
CMemoryCacheBase		X	X	X	X	X
CMemoryCache		X	X	X	X	X
CBlobCache	X		X	X	X	X

CFileCache	X		X	X	X	X
CStencilCache	X		X	X	X	X

This table illustrates that these caching classes share many of the same template arguments but differ when it comes to automated cache maintenance as determined by *MonitorClass*. The CMemoryCacheBase and CMemoryCache classes do not offer automated maintenance services.

The CBlobCache, CFileCache, and CStencil classes are specialized on data type (**void***, **LPSTR**, and **void***, respectively). The CMemoryCacheBase and CMemoryCache classes take the data type to be cached as the *DataType* argument.

Additional Cache Implementations

ATL Server provides two additional cache classes:

- CDllCache
- CDataSourceCache

The **CDllCache** class caches loaded DLLs, and the **CDataSourceCache** class caches open database connections. Unlike the cache implementations previously mentioned, these two class do not use CMemoryCacheBase as a base class and do use template parameters as extensively.

Template parameters	MonitorClass	SyncClass
CDllCache	X	
CDataSourceCache		X

All cache classes support multithreaded access, but by default the **CDataSourceCache** class uses CComFakeCriticalSection, which does not perform any thread synchronization. For data source caches that are used in multithreaded environments, therefore, you must override the default *SyncClass* argument with a thread synchronization class such as CComCriticalSection.

The **CDllCache** class is the only ATL Server cache class that does not take a *SyncClass* template argument, but it is thread-safe. It uses CComCriticalSection to help provide access safety.

See Also

Other Resources

Caching
ATL Server Concepts
Caching Reference

Adding Cache Support

To add cache support to an ATL Server project

1. Add an instance of the desired cache class to the ISAPI extension:

```
CBlobCache< CWorkerThread<> > m_BlobCache;
```

2. Expose the necessary cache interfaces (IMemoryCache at the minimum), by adding or modifying the ISAPI extension implementation of QueryService:

```
HRESULT STDMETHODCALLTYPE QueryService(REFGUID guidService,
         REFIID riid, void** ppvObject)
   {
      if (InlineIsEqualGUID(guidService, IID_IMemoryCache))
         return m_BlobCache.QueryInterface(riid, ppvObject);

      if (InlineIsEqualGUID(guidService, IID_IMemoryCacheStats))
         return m_BlobCache.QueryInterface(riid, ppvObject);

      if (InlineIsEqualGUID(guidService, IID_IMemoryCacheControl))
         return m_BlobCache.QueryInterface(riid, ppvObject);

      return baseISAPI::QueryService(guidService, riid, ppvObject);
   }
```

3. Obtain access to the cache from the request handler DLL:

```
CComPtr<IMemoryCache> m_spCache;
m_spServiceProvider->QueryService(__uuidof(IMemoryCache), &m_spCache);
```

4. When they are needed, attempt to retrieve items from the cache before resorting to loading or deriving them:

```
HCACHEITEM hEntry = 0;
HRESULT hr = m_spCache->LookupEntry( keyStr, &hEntry );
if (SUCCEEDED(hr))
{
   DWORD size = 0;
   void* data = 0;
   if (SUCCEEDED(m_spCache->GetData( hEntry, &data, &size )))
   {
   }
   hr = m_spCache->ReleaseEntry(hEntry);
}
```

5. Add newly loaded or derived items to the cache using the **Add** method. Adding items with keys that match existing items causes the original item to be overwritten:

```
m_spCache->Add( keyStr, szValue, len, &ft, 0, 0, m_cacheClient );
```

See Also

Tasks

BlobCache Sample: Generic Memory Caching

Other Resources

Caching
ATL Server Concepts
Caching Reference

Debugging (ATL Server)

The following topics provide information about debugging ATL Server Web applications and XML Web services.

In This Section

Viewing Trace Messages And Handling Asserts

Provides step-by-step information for viewing trace messages and assertion failures from ATL Server DLLs running in Internet Information Services (IIS).

Automatically Attaching To The Web Server Process

Describes how to automatically attach the debugger to the correct Web server process.

Related Sections

Debugging an ATL Server Web Application

Discusses how to debug a Web application on a local host, remote server, or server farm.

Debugging an ATL Server Web Service

Describes how to debug an XML Web service designed with ATL Server on a local host, remote server, or server farm.

Debugging

Provides links to using the Visual Studio debugger to correct logic errors in your application or stored procedures.

Debugging Visual C++

Provides links to topics discussing common debugging problems and techniques for C and C++ applications.

Debugging Injected Code

Describes how to debug attributes, which inject code into your project.

Detaching Programs

Provides information about dbgproxy.exe, which allows you to disconnect the debugger from a program.

ATL Server

Provides links to conceptual information about the functionality of ATL Server.

ATL Server Samples

Provides links to samples demonstrating how ATL Server is used.

ATL Server Reference

Provides links to language reference documentation for the ATL Server Library.

See Also

Other Resources

ATL Server Concepts

Viewing Trace Messages And Handling Asserts

To view trace messages and assertion failures from ATL Server DLLs running in IIS

1. Select the **Custom assert and trace handling support** check box on the **Developer Support Options** tab when you generate your ATL Server request handler or ISAPI DLL using the ATL Server Project Wizard

   -or-

   Add the following code to one of the source files in your project:

   ```
   // For custom assert and trace handling with WebDbg
   #ifdef _DEBUG
   CDebugReportHook g_ReportHook;
   #endif
   ```

2. If you intend to view the messages from a remote machine, set the name of that machine as the one which will provide the debug pipe by passing it to the CDebugReportHook constructor or SetPipeName method:

   ```
   g_ReportHook.SetPipeName("collection_machine");
   ```

3. Use standard trace and assert macros, such as ATLTRACE or _ASSERTE, in your code.

4. Build debug versions of your DLLs.

5. Install the DLLs on your Web server.

6. Run WebDbg.exe.

   WebDbg.exe is located in the \Common7\Tools folder of your Visual C++ .NET installation. This application provides a simple user interface for viewing trace messages and interacting with asserts.

   To use WebDbg.exe to view the call stack at the point that an assert is triggered, click the **View** menu and then click **Stack Trace**. This option is turned off by default.

   To set the permissions on the pipe, click the **File** menu and then click **Permissions**. If you are collecting debug information remotely or running your Web server using Medium or High application protection, make sure that the accounts of the Web processes you are interested in have permissions to access the debug pipe.

Note
The debug output is written using the security context of the process, so a typical Web application will write to the pipe using the **SYSTEM** or **IWAM**_*MachineName* accounts that are local to the Web server.

7. Test your Web application using the desired clients.

Note
If there is no instance of WebDbg.exe running against the specified pipe and if a debugger is not attached to the server process, trace messages and assertion failures will not be recorded.

See Also

Tasks

WebDbg Sample: A Client for Debugging Server Applications Using CDebugReportHook

Reference

CDebugReportHook Class

Concepts

Debugging (ATL Server)

Automatically Attaching To The Web Server Process

Internet Information Services (IIS) can load your ATL Server code into the inetinfo process or into any one of a number of dllhost processes depending on the Application Protection settings of the virtual directory used by your Web application or XML Web service.

To attach the debugger to the correct process, ATL Server provides code in CIsapiExtension that loads a COM object, whose CLSID is specified in a specially formed HTTP request, and passes it the ID of the process in which it is loaded. This allows Visual Studio to obtain the ID of the process serving the page specified by the HTTP URL property and attach the debugger without user intervention.

Note
The HTTP URL for debugging must be set to a resource whose file extension is associated with your ISAPI extension DLL. Typically, that means the HTTP URL will be set to a resource with a .srf or .dll extension. The URL must be specified completely and include the protocol prefix (http://).

To enable the code that does this in your ISAPI extension DLL, define ATLS_ENABLE_DEBUGGING before including atlisapi.h.

Caution
For security reasons, you should not define ATLS_ENABLE_DEBUGGING in code that is installed on a production server. Doing so can allow debugger users to gain access to the server.

To automatically attach to the Web server on the local machine

1. Set the following options in the Remote Settings pane of the Debugging property page for the project:
   - **Connection** = Local
   - **Remote Machine** = *<ignored>*
   - **Remote Command** = *<ignored>*
   - **HTTP URL** = *<URL for a file type associated with your ISAPI extension DLL>*
2. Ensure that your ISAPI extension DLL was built with either the **_DEBUG** or **ATLS_ENABLE_DEBUGGING** symbol defined.
3. On the **Debug** menu, click **Start**.

To automatically attach to the Web server on a remote machine

1. Ensure that remote debugging is installed on the remote machine
2. Set the following options in the Remote Settings pane of the Debugging property page for the project:
   - **Connection** = Remote via DCOM
   - **Remote Machine** = *<ignored>*
   - **Remote Command** = *<ignored>*
   - **HTTP URL** = *<URL for a file type associated with your ISAPI extension DLL>*
3. Ensure that your ISAPI extension DLL was built with either the **_DEBUG** or **ATLS_ENABLE_DEBUGGING** symbol defined.
4. On the **Debug** menu, click **Start**.

See Also

Reference

Project Settings for a C++ Debug Configuration

Concepts

Debugging (ATL Server)

Error Handling (ATL Server)

Error handling is an important part of any application. Consider your error handling strategy carefully to help you debug your application during development, troubleshoot problems once deployed, and keep your site's users informed.

Rejecting Bad Requests

One of the most important areas of error handling in a Web application is rejecting bad requests. On a high volume Web site, the number of erroneous requests can be high. To ensure that such requests have minimal impact on server performance, you should analyze and reject bad requests as early as possible. Look at each stage of your site's architecture and remove as many requests as possible before passing the request to the next stage.

The first step to rejecting bad requests is configuring Internet Information Services (IIS). Use Directory Security to reject requests from unauthorized users or to block users based on IP address if that is appropriate to your site. Decide which HTTP methods your site needs to handle and allow only those to pass through to your ISAPI DLL. Use the IIS App Mappings property page to limit the methods that are associated with your ISAPI DLL to the ones that you actually expect to use.

Once your ISAPI DLL has been invoked, you can filter out further requests. At this stage, it is a good idea to keep the processing simple and applicable to the whole site. For example, consider removing large requests at this stage. For details, see Limiting the Size of a Request. Balance the processing cost of checking every request with the cost of passing bad requests on to your request handler DLL.

Once the request reaches your request handler DLL, you can choose to reject requests before or after the request and response objects have been initialized. See Initializing Request Handlers for information about the possible points at which you can add your error checking code during initialization.

Handling Errors Early

Maintaining the performance of your Web site in the presence of bad requests is not the only reason to identify and handle errors as early as possible. In many cases, it is important to identify errors before any data has been sent to the client. In particular, if you want to send a distinguished HTTP error status code to the client, you must identify the error before you send HTTP headers to the client. ATL Server helps you in this goal by buffering the response by default. This means that you can build your responses in an intuitive way without committing to sending the data to the client until you are done.

Generating a Response

For an example of generating an error response and a discussion of the related issues, see Creating an Error Response in the ATL Server Tutorial. For a complete sample demonstrating the same principles with slightly different features, see the ShowErrors section of the WebFeatures Sample.

See Also

Tasks

Choosing a Return Value for a Replacement Method

Reference

ShowErrors

Concepts

HTTP Status Codes

Creating an Error Response

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Samples

HTTP Status Codes

While it is possible to create a functioning Web site that always returns the 200 OK HTTP status code (even for error conditions), there are a number of reasons to prefer to use distinguished status codes where possible.

The most important is that the different status codes provide features that cannot be replicated using only 200 OK. If you want to indicate success without causing a page refresh at the client, you must use 204 No Content. If you want to trigger authentication, you must use 401 Unauthorized. If you want to redirect the user to a different resource temporarily, use 307 Temporary Redirect. It pays to understand the meanings and uses of the various HTTP status codes. You can find full information in the HTTP 1.1 specification at http://ietf.org/rfc/rfc2616.txt.

Errors And 200 OK

While you should usually use the appropriate HTTP status code for error conditions, there are two valid reasons for returning 200 OK in the event of an error:

1. You returned HTTP headers to the client before the error occurred.

   Consider returning the results of a slow database query to a client in stages. After the first set of results is returned to the client through an asynchronous write, the connection with the database could be lost. At this stage, the only error that can be returned to the client is a human-readable message in the body of the response.

2. You want to prevent the user agent from displaying its own information in the event of an error and hiding the response body from the server.

   Consider Internet Explorer, which has an option to display friendly HTTP error messages. These messages are useful to users and provide helpful suggestions for working around the problem that caused the error. However, it may be that you feel you can provide more appropriate information than Internet Explorer and want to be sure that the user sees your message. If you have control over the clients, you can encourage users to disable Internet Explorer error messages. If not, you can use 200 OK responses to hide the error condition from the browser.

However, you should think carefully before deliberately hiding errors by using the 200 OK status. Employing this technique means removing programmatic access to the error code. What effect will this action have on caching servers? Do you really want to miss out on the benefits of updated user agents that react better to error codes than the current generation? Do you want to take responsibility for localizing error responses when the browser manufacturers can do it for you?

The major advantages and disadvantages of using 200 OK for error conditions are summarized in the following table:

Distinguished code	**200 OK**
Easy programmatic access to status code.	Programmatic access to error code would require nonstandard parsing of the response body.
Clients employing the HTTP HEAD method have useful information, can identify errors, and can take appropriate action.	Clients employing the HTTP HEAD method have no useful information and cannot identify errors.
Caching and proxy servers can use the status code to enhance their operation.	Caching and proxy servers have no access to the status code.
User agents can provide detailed, appropriate, and locale-specific information to the user.	The server must take responsibility for providing localized error messages.
Error must be identified before HTTP headers have been sent to the client.	Error can be identified after HTTP headers have been sent to the client.

See Also

Tasks

Choosing a Return Value for a Replacement Method

Reference

ShowErrors

Concepts

Creating an Error Response

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

ATL Server Tasks

This section provides instructions for common tasks that you may wish to perform in your ATL Server Web applications.

In This Section

Adding a Replacement Method

Explains how to add a replacement method using the HTML designer or by hand.

Choosing a Return Value for a Replacement Method

Describes how to decide on the value that you return from your replacement methods.

Passing An Argument To A Replacement Method

Shows how to allow your replacement methods to accept arguments provided in the server response file.

Initializing Request Handlers

Covers the main points at which you can carry out initialization in your request handler.

Enabling Uploaded Files

Provides information on allowing files to be uploaded to your web site and read by your request handler.

Limiting the Size of a Request

Explains the different techniques for protecting your web site against large request bodies.

Enabling Page Caching

Describes how to enable, disable, and control the caching of responses generated by your request handler.

Associating SRF Files with Your ISAPI DLL by Hand

Provides detailed instructions for registering your ISAPI extension DLL with IIS.

Related Sections

Session-State Tasks

Covers adding session-state support to an existing ATL Server ISAPI extension DLL.

HTTP Client Tasks

Provides step-by-step instructions to launch an HTTP client request.

ATL Server

Provides links to conceptual information about the functionality of ATL Server.

ATL Server Samples

Provides links to samples demonstrating how ATL Server is used.

ATL Server Reference

Provides links to language reference documentation for the ATL Server Library.

Adding a Replacement Method

To add a replacement method to a request handler using Visual Studio

1. Open the server response file in the HTML designer.
2. Switch to HTML view.
3. Add a replacement tag to the server response file:

```
{{TagName}}
```

4. Right-click the tag and click **Add Tag Handler** on the shortcut menu.
5. Implement the replacement method.

   For details, see Choosing a Return Value for a Replacement Method.

To manually add a replacement method to a request handler

1. Add the replacement method to the appropriate request handler:

```
HTTP_CODE MethodName()
{
    // TODO - add your code here
    return HTTP_SUCCESS;
}
```

2. Implement the replacement method.

   For details, see Choosing a Return Value for a Replacement Method.

3. Add an entry for the method to the replacement method map in the same class:

```
BEGIN_REPLACEMENT_METHOD_MAP(RequestHandlerName)
    // ...
    REPLACEMENT_METHOD_ENTRY("TagName", MethodName)
END_REPLACEMENT_METHOD_MAP()
```

   –or–

   Apply the tag_name attribute to the method:

```
[tag_name("TagName")]
HTTP_CODE MethodName()
```

4. Add a replacement tag to one or more server response files that use that request handler:

```
{{TagName}}
```

See Also

Concepts

Passing an Argument to a Replacement Method

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Samples
ATL Server Response File Reference

Choosing a Return Value for a Replacement Method

Return a value from a replacement method, CheckValidRequest, or ValidateAndExchange based on the following guidelines:

To continue default rendering of a response

- Return **HTTP_SUCCESS** or any HTTP success code except **HTTP_SUCCESS_NO_PROCESS** and **HTTP_S_FALSE**.

To prevent or discontinue rendering of a while or if block

1. Return **HTTP_S_FALSE** to continue rendering the response from the point after the current block.

   –or–

2. See the following procedure.

To discontinue further rendering of a response

1. Return **HTTP_SUCCESS_NO_PROCESS** to return the response generated by your request handler at this point.

   –or–

2. Return any HTTP error value to return the response provided by the ATL Server error response generation mechanism.

To trigger the ATL Server error response generation mechanism

- Return any HTTP error value. Use one of the status codes defined in atlisapi.h or define your own using AtlsHttpError.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Passing an Argument to a Replacement Method

It is possible to pass an argument to a replacement method from a server response file by enclosing the text intended to act as the data of the argument in parentheses following the replacement method name in a replacement tag.

When the server response file is parsed, the text within the parentheses is passed to the parse function connected to that replacement method. The parse function is responsible for using the data to initialize an argument that can then be passed to the replacement method whenever the method is called. Although the replacement method can be passed only a single pointer parameter, the pointer can be to any simple type, or to an array or structure of such types.

Parse Functions

Parse functions are methods of a request handler class with the following signature:

```
HTTP_CODE parseFunction(
         IAtlMemMgr* pMemoryManager,
         LPCSTR szArgumentData,
         parameterType** ppArgument
);
```

They must be implemented such that they:

- Allocate memory for the argument to be passed to the replacement method using pMemoryManager->Allocate.
- Set the value of the argument to be passed to the replacement method using the data obtained from *szArgumentData*.
- Pass a pointer to the argument to be passed to the replacement method back through *ppArgument*.

The memory allocated using `pMemoryManager` will be automatically freed by the framework when the stencil is unloaded.

Note The parameter cannot be a pointer to a complex type such as an object with a destructor or a structure whose members are pointers to dynamically allocated data because the destructor will never be called and the member pointers will never be freed.

Replacement Methods

Replacement methods that accept an argument are methods of a request handler class with the following signature:

```
HTTP_CODE replacementMethod(parameterType* pArgument);
```

The *parameterType* here must be the same as the *parameterType* of the associated parse function.

Parse functions can be associated with a replacement method in one of these ways:

- If writing attributed code, the name of the associated parse function can be specified using the *parse_func* parameter of the tag_name attribute.
- If writing attributed code and *parameterType* is one of the supported types listed in the documentation for the tag_name attribute, the name of the parse function can be omitted and the corresponding parse function will be chosen automatically.
- If writing nonattributed code, the REPLACEMENT_METHOD_ENTRY_EX macro can be used.

Performance

To get best performance, it is important to understand when the parse function and the replacement method are called.

The parse function is called only during parsing of the server response file. The SRF is tokenized, parse functions are called, and the results are stored in memory until the stencil is unloaded.

The replacement method is called on every request that causes the flow of control to pass through that method.

In typical situations, the replacement method is called many orders of magnitude more often than the parse function. For this reason, you should do as much work in the parse function as possible if it means that you can do less work in the replacement method.

Tasks

To pass an argument to a replacement method from a server response file using attributed code

1. Decide what type will be used to hold the data passed to the replacement method. If necessary define this type in your code.
2. Implement the parse function to construct an object of this type and initialize it with data obtained from the server response file.

   -or-

   Use one of the ATL Server parse functions if the data type is in the list of supported types and the implementation of the parse function is appropriate for your needs. See tag_name for the complete list.
3. Declare the replacement method with the appropriate signature.
4. Associate the parse function with the replacement method using the *parse_func* parameter of the tag_name attribute. (This value can be omitted if an ATL Server parse function is used).
5. Add the data used to initialize the argument to the server response file in parentheses following the replacement method name in a replacement tag.

To pass an argument to a replacement method from a server response file using nonattributed code

1. Decide what type will be used to hold the data passed to the replacement method. If necessary define this type in your code.
2. Implement the parse function to construct an object of this type and initialize it with data obtained from the server response file.

   -or-

   Use one of the ATL Server parse functions if the data type is in the list of supported types and the implementation of the parse function is appropriate for your needs. See the **Parse*** methods provided by ITagReplacerImpl for the complete list.
3. Declare the replacement method with the appropriate signature.
4. Associate the parse function with the replacement method using REPLACEMENT_METHOD_ENTRY_EX.
5. Add the data used to initialize the argument to the server response file in parentheses following the replacement method name in a replacement tag.

See Attributed Request Handler Code, Non-attributed Request Handler Code and the SRFSyntax sample for examples of passing an argument to a replacement method.

Limitations on Replacement Method Arguments In Server Response Files

Parameters cannot contain closing parentheses. Note that parameters are treated as ANSI text in this regard. Be careful not to use a character encoding that results in one of the bytes having the same value as an ANSI closing parenthesis.

See Also

Reference

Replacement Tag
tag_name
REPLACEMENT_METHOD_ENTRY_EX

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples
ITagReplacerImpl Members

Initializing Request Handlers

To have initialization code executed before the request and response objects have been initialized

- Add code to the constructor of your request handler.

  Use the constructor for initialization that cannot fail because there is no supported way of returning an error from the constructor of a request handler.

To have initialization code executed before the request object has been initialized and after the response object has been initialized

- Override CRequestHandlerT::CheckValidRequest in your request handler.

  You can use the server context and response object, but do not use the request object. If the initialization code that you add to **CheckValidRequest** fails, you can return an HTTP status code.

To have initialization code executed after the request and response objects have been initialized

- Override CRequestHandlerT::ValidateAndExchange in your request handler and add your code there.

  You can use any member of CRequestHandlerT at this point and can return an appropriate HTTP status code.

To uninitialize your request handler

1. Override CRequestHandlerT::Uninitialize in your request handler and add your code there.

   –or–

2. Add code to your request handler's destructor.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Handling Uploaded Files

A request handler that accepts uploaded files is a potential weak point for your server, so decide whether you really need to allow files to be uploaded. Once you have decided that it is appropriate, consider how you can limit the exposure of your server to user error or external attacks.

There are two main vulnerabilities associated with allowing files to be uploaded to your server:

- Performance can suffer.
- Your server could run out of disk space.

Performance

If you allow large files to be uploaded, your server will have to take the time to read all that data from the client. This can tie up a thread in your server and consume memory. Try to ensure that the data that you receiving is useful to you before committing those resources.

Note
Handling any large request can be a problem for the same reasons. To protect your server, limit the size of the requests that you handle wherever possible. The ATL Server default request limit of 48 KB is reasonable for most simple forms but may no t be large enough for the kinds of files that you expect. If your request handler is not designed to accept forms, you can set th e limit to zero. For details, see Limiting the Size of a Request.

Even if you limit the size of the requests that you handle, allowing file upload still has a potentially negative impact on server performance as a result of writing to the disk. File IO is an expensive operation. Consider isolating the parts of your application that need to accept uploaded files from the rest of your Web site. This would allow you to separate the more vulnerable parts of your site from the less vulnerable parts and target your hardware budget more appropriately.

Disk Space

If you do not deal correctly with the uploaded files and you run out of disk space on your system drive, you will see major problems on your server. ATL Server helps by automatically deleting files at the end of the request, but once you have enabled file upload, do not to make assumptions about those files. Monitor the disk space on your server and set policies for managing it. For example, set the temporary folder to be on a disk other than the system drive. (The temporary folder is used by ATL Server to store uploaded files until such time as you move or delete them.) Use disk quotas in Windows 2000 or later to set a hard limit on the amount of disk space that can be used. Use the features that ATL Server provides to help you delete files that you no longer need.

The decisions that you make and the code you write will be affected by the situation in which you deploy your Web site. If you expect your server to be accessed through an intranet, you may make different decisions than if you are building a Web site open to the public on the Internet. In either situation, understand and document the tradeoffs that you make. Have your test team attack your site without preconceptions.

See Enabling Uploaded Files for instructions on enabling files to be uploaded to your server.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Enabling Uploaded Files

To enable file upload

1. Read the information in Handling Uploaded Files and decide whether you really need to allow file upload. Understand the responsibilities in terms of the code you write and the long-term management of the site.
2. Limit the size of valid requests to a sensible value. See Limiting the Size of a Request for instructions.
3. Override CRequestHandlerT::FormFlags in your request handler to allow uploaded files. Include the **ATL_FORM_FLAG_IGNORE_EMPTY_FILES** flag, if possible, to avoid writing empty files to disk.
4. Use CHttpRequest::m_Files or CHttpRequest::GetFormVars to access the uploaded files from **CRequestHandlerT::m_HttpRequest**. The **m_Files** collection contains only the uploaded files and provides complete information about each of them. **GetFormVars** contains standard form fields along with the field names and temporary file names of the uploaded files.
5. Be sure to delete any files that you do not need or were not expecting. ATL Server will delete any files that you do not remove from the CHttpRequest::m_Files collection when the request is finished. To prevent files being deleted, remove them from the CHttpRequest::m_Files collection, for example by calling RemoveKey.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Limiting the Size of a Request

To limit the size of a request for all requests through a particular ISAPI extension DLL

- Override CIsapiExtension::HttpExtensionProc in your ISAPI extension DLL and compare your maximum request size with the value returned by **EXTENSION_CONTROL_BLOCK::cbTotalBytes**. Reject any requests that exceed the maximum size.

To limit the size of a request for a particular request handler

1. Override CRequestHandlerT::CheckValidRequest and compare your maximum request size with the value returned by IHttpServerContext::GetTotalBytes. Reject any requests that exceed the maximum size.

   –or–

2. Override CRequestHandlerT::ValidateAndExchange and compare your maximum request size with the value returned by CHttpRequest::GetTotalBytes. Reject any requests that exceed the maximum size.

   For more information on the differences between these two methods, see Initializing Request Handlers.

To delay form processing for a particular request handler based on the size of the request

- Override CRequestHandlerT::MaxFormSize and return the maximum number of bytes in the body of a request that should be parsed. If a request exceeds this size, default form processing is not carried out (that is, the files and form variables collections of the request object, **m_HttpRequest**, will not be populated). Note that taking this action does not cause the request to be rejected; it simply delays or prevents expensive processing on large requests.

To reject a request because it is too large

- Return **HTTP_REQUEST_ENTITY_TOO_LONG** to use the default response or build your own response and return an appropriate status code.

Note
The information in this topic applies to limiting the number of bytes in the *body* of the request. IIS limits the number of bytes in the header of a request automatically.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Enabling Page Caching

To enable page caching in a request handler

- Include **ATLSRV_INIT_USECACHE** in the flags returned from IRequestHandler::GetFlags. This is most easily achieved for IRequestHandlerImpl-derived classes (including CRequestHandlerT) by overriding IRequestHandlerImpl::CachePage to return **TRUE**. If no further action is taken, the page will be cached indefinitely.

To limit the length of time for which the page is cached

1. Query the server context for IPageCacheControl.
2. Call IPageCacheControl::SetExpiration.

To prevent page caching once enabled

1. Query the server context for IPageCacheControl.
2. Call IPageCacheControl::Cache(**FALSE**).

   –or–

   Return **HTTP_SUCCESS_NO_CACHE** from any of the following:

   - IRequestHandler::InitializeHandler
   - IRequestHandler::HandleRequest
   - CheckValidRequest
   - ValidateAndExchange
   - A replacement method

Limitations on Cached Pages

- Only responses to HTTP GET requests will be cached.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Associating SRF Files with Your ISAPI DLL by Hand

To manually associate server response files and Web application DLLs with your ATL Server ISAPI extension DLL

1. Generate an ATL Server ISAPI extension DLL using the ATL Server Project Wizard.
2. Build the ISAPI extension DLL and copy it to your Web server.
3. Create a mapping between files with the .srf and .dll extensions and the new ISAPI extension DLL.

   For example:

   - Run **Internet Services Manager**.
   - Right-click the appropriate Web site or virtual directory in the tree and click **Properties** on the shortcut menu.
   - Click the **Home Directory** or **Virtual Directory** property page.
   - Click the **Configuration** button.
   - Click the **App Mappings** property page.
   - Click **Add**.
   - Enter the path to your ISAPI DLL in the **Executable** text box.
   - Enter **.srf** in the **Extension** text box.
   - Click **OK**.
   - Click **Add**.
   - Enter the path to your ISAPI DLL in the **Executable** text box.
   - Click **.dll** in the **Extension** text box.
   - Click **OK** until all dialog boxes are accepted.

Note
You can use the Web Deployment Property Page to have these associations automatically set up on your local machine during the development process.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Writing Asynchronous Web Applications

To allow your request handler to write to the client asynchronously

- Add one of the following macros to your request handler's class definition:

Macro	Description
DECLARE_ASYNC_HANDLER	Use this macro if buffered content is to be written asynchronously.
DECLARE_ASYNC_HANDLER_EX	Use this macro if you need to make explicit calls to asynchronous methods.

Note
You do not need to write asynchronously just because you have prepared your request handler using one of these mac ros. However, if you will never write asynchronously, you should not use these macros because they impose a small pe rformance cost.

To write to the client asynchronously

1. Optionally call one of the methods for writing to the client asynchronously, such as TransmitFile.
2. Return one of the following values from your request handler's HandleRequest method:

Return value	Description
HTTP_SUCCESS_ASYNC	Writes buffered content to the client asynchronously. ATL Server code calls **HandleRequest** or a replacement method again when the buffer has bee n flushed.
HTTP_SUCCESS_ASYNC _DONE	Writes buffered content to the client asynchronously. If this value is returned from **HandleRequest** or **ValidateAndExchange**, ATL Server code clos es the connection with the client when the buffer has been flushed. If this value is returned from a replacement method, the ATL Server code moves on to the next token in the server response file.
HTTP_SUCCESS_ASYNC _NOFLUSH	User code has initiated an asynchronous call. ATL Server code calls **HandleRequest** or a replacement method again when the call is complet e.
HTTP_SUCCESS_ASYNC _NOFLUSH_DONE	User code has initiated an asynchronous call. If this value is returned from **HandleRequest** or **ValidateAndExchange**, ATL Server code clos es the connection with the client when the call is complete. If this value is returned from a replacement method, the ATL Server code moves on to the next token in the server response file.

If you write to the client asynchronously, you must have prepared your request handler by including ATLS_FLAG_ASYNC in the flags returned by GetHandlerFlags and by including **ATLSRV_INIT_USEASYNC** in the flags returned by GetFlags. Both DECLARE_ASYNC_HANDLER and DECLARE_ASYNC_HANDLER_EX will do this for you in IRequestHandlerImpl-derived classes.

In addition, if you want to enable explicit asynchronous writes, you must include **ATLSRV_INIT_USEASYNC_EX** in the flags returned by GetFlags. Only DECLARE_ASYNC_HANDLER_EX will do this for you in IRequestHandlerImpl-derived classes. (There is a separate flag for this because it requires extra synchronization to ensure that two calls to **HandleRequest** are not made at the same time.)

CRequestHandlerT enables all these values to be returned from replacement methods (except when they are used as conditional handlers called by **while** and **if** tags).

If HTTP_SUCCESS_ASYNC_DONE or HTTP_SUCCESS_ASYNC_NOFLUSH_DONE is returned from a replacement method, when processing resumes it will pick up from the next token in the stencil after the tag that called the replacement method.

If HTTP_SUCCESS_ASYNC or HTTP_SUCCESS_ASYNC_NOFLUSH is returned from a replacement method, the same method will be called again when processing resumes.

The asynchronous callback checks to see if the request is done based on whether the handler returned HTTP_SUCCESS_ASYNC_DONE or HTTP_SUCCESS_ASYNC_NOFLUSH_DONE from **HandleRequest**.

Note
Request handlers that use asynchronous functions must not store pointers to per-thread services as data members. Although request handlers are only ever accessed by a single thread at a time, a request handler that returns one of the HTTP_CODEs for asynchronous behavior can be accessed from multiple threads during the course of its life. Per-thread services must only ever be used from a single thread.

Developers should test all Web server code under stress on a multiprocessor machine. The load on the server must be high enough that multiple threads are actually being used simultaneously. This type of testing is especially important for asynchronous applications that are more sensitive to multithreading issues.

See Also

Reference

Testing XML Web Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Samples

ATL Server Security

Your ATL Server code usually runs in the security context of the client making the current request. This context is supplied by the IIS HSE_REQ_GET_IMPERSONATION_TOKEN server support function, which is wrapped by IHttpServerContext::GetImpersonationToken. The impersonation of the client is handled very early during the processing of a response (in CIsapiExtension::DispatchStencilCall) and remains in force unless you explicitly choose to impersonate another user.

This means that if the client has been authenticated, your code will run in the context of the authenticated user. If the client has not been authenticated, your code will run in the context of the anonymous user account specified for the Web site or virtual directory holding the currently requested resource. By default, the anonymous user account is the Internet Guest Account, **IUSR**_*MachineName*, but you can change this (using Internet Services Manager, for example).

By default, clients accessing your Web server will not be authenticated because anonymous access is automatically enabled for newly created virtual directories. However, you can force authentication for a particular Web site or virtual directory by disabling anonymous access (using Internet Services Manager, for example).

Even if you do not disable anonymous access, you can use administration tools or code to ask clients to authenticate themselves. You can require authentication for a particular server response file by setting its access control list so that it does not allow access to the anonymous user account. Alternately, you can request authentication more selectively by returning a 401 (Unauthorized) HTTP status code from your request handler. You can use the HTTP_UNAUTHORIZED constant for this.

If you need to change the security context in which your code executes for some reason, you can call a function such as SetThreadToken, execute your code, and then revert to the client's user account by calling AtlImpersonateClient.

You can determine whether the client has been authenticated by checking the "LOGON_USER" server variable. If the variable contains a value, the client was authenticated; otherwise, the client is accessing the resource anonymously.

You should be extremely careful to check the return values of security functions and ensure that the client's default security context has been successfully applied before executing any further code. Use destructors to ensure that your code is safe in the face of exceptions.

Example

See the ShowUser section of the WebFeatures Sample.

See Also

Other Resources

ATL Server Concepts
ATL Server Architecture

ATL Server and COM

Your ATL Server code usually runs on a thread from the ISAPI extension's thread pool on which COM has been initialized for use in a single-threaded apartment (STA). This initialization occurs in CIsapiExtension::OnThreadAttach. COM is uninitialized in CIsapiExtension::OnThreadTerminate.

If you want to change the way in which COM is initialized, you can override these methods in your ISAPI extension class derived from CIsapiExtension. You do not need to call the base class implementations of these methods; the only thing that they do is initialize and uninitialize COM.

Note that there is some code in your ISAPI extension DLL that does not run on one of the threads from the pool; the code that is executed directly from the CIsapiExtension::GetExtensionVersion, CHttpServer::HttpExtensionProc, and CIsapiExtension::TerminateExtension functions. These functions are executed on threads owned by IIS and call the corresponding methods provided by CIsapiExtension or your derived class. Because you do not own the thread(s) that call these methods, you cannot assume anything about whether or how they have been initialized for COM and you should not try to initialize COM on these threads.

If you need to create an object in your ISAPI extension DLL that requires COM, you have a few choices:

1. Create the object when the ISAPI extension DLL is initialized. Create your own thread in **GetExtensionVersion** on which you can initialize COM; however, you need and which calls your ISAPI extension's **GetExtensionVersion** method. You need to write code so that the IIS thread waits for your thread to finish its task and you'll also need to provide some way to communicate status information back to the IIS thread.

2. Create the object the first time that it is needed. For example, you could initialize the object the first time a request handler calls QueryService requesting that object. At this point, you know that the code is running on a pool thread that has been COM-initialized in CIsapiExtension::OnThreadAttach or your own override of that method.

   Be careful to synchronize access to your class data members because **QueryService** can be called simultaneously from multiple threads and make sure that you understand COM's rules for passing interface pointers between threads.

3. Create the object when the first thread in the pool is initialized. Override CIsapiExtension::OnThreadAttach and execute the initialization code there. The code will have to check whether the object has already been initialized, so that check will run every time a pool thread is created.

4. Create one object for each thread in the pool when the thread is initialized. You can store interface pointers as data members of your worker thread class derived from CIsapiWorker. Override CIsapiWorker::Initialize, call the base class implementation first to ensure that COM is initialized, and then perform any other initialization that you need to do.

5. Create multiple objects on demand. For example, create a different object every time a request comes in to **QueryService**. As before, you know that the code is running on a COM-initialized thread, but this time you do not need to worry about multiple threads accessing the object.

If the resources held by the object are not large, providing one object per thread is a great way of getting scalability without having to write any complicated code if your architecture supports that approach.

You can always create COM objects in your request handlers because this code runs on a thread from the pool on which COM has been initialized. For example, you can create a COM object in a request handler and make it available to other request handlers (or the same request handler on subsequent requests) by passing it to the ISAPI extension so that it can expose it as a dynamic service. For details, see IIsapiExtension::AddService.

See Also

Other Resources

ATL Server Concepts
ATL Server Architecture

Developing Global Applications

Developing global applications with ATL Server is similar to developing other global server applications. The same issues and opportunities present themselves. To get a good understanding of the general issues involved, you can use the following resources:

Resource	Description
International Programming	Information about C run time and C++ library features relevant to programming for international markets.
Character Set Recognition	A description of how to apply a character set to an HTML document along with a useful table of ch aracter sets, aliases, and Internet Explorer version information.

ATL Server provides some specific support for localized responses by using the codepage and locale tags in a server response file.

Multilingual responses can be generated from a server response file using the UTF-8 code page. (UCS-4, UCS-2, and UTF-16 encoded server response files are not supported).

The ShowLocalized part of the WebFeatures Sample demonstrates how to create a request handler that can match the user's language settings against languages supported by the server and display content specific to the user's locale. The sample displays a localized phrase loaded from a resource and the appropriately formatted system time.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

ATL Server References

This topic contains a list of links to items of interest to ATL Server developers. In addition to links to the most important ATL Server-specific documentation, you will find links to specifications for the Internet standards supported by ATL Server, knowledge base articles with useful advice, and topics of general interest to developers of Web applications and XML Web services.

Documentation

- ATL Server Concepts
- ATL Server Architecture
- ATL Server Tasks
- ATL Server Reference
- ATL Server Samples

Specifications

- HTTP 1.1 RFC 2616 (http://ietf.org/rfc/rfc2616.txt)
- HTTP State Management (Cookies) RFC 2109 (http://ietf.org/rfc/rfc2109.txt)
- MIME RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt).
- SOAP (http://www.w3.org/TR/SOAP)
- WSDL (http://www.w3.org/TR/wsdl)
- UDDI (http://www.uddi.org/specification.html)
- XML Web Service Discovery

Performance

- The Art and

See Also
Reference
ATL Server Tutorial

ATL Server Categories

The ATL Server reference documentation is broken down into the following categories.

Category	Description
ATL Server Framework Reference	Provides the infrastructure used by ATL Server Web applications and XML Web services for integrating with Internet Information Services (IIS), parsing HTTP requests and generating responses. These classes allow the use of Web application DLLs for partitioning functionality within an application and server response files for separating dynamic and static content. The framework classes are typically found in atlisapi.h and atlstencil.h.
Request Handling Reference	Provides attributes, classes, and macros that give you features to use directly for hooking your request handler into ATL Server's framework.
Replacement Method Reference	Provides attributes and macros for the plumbing necessary to implement a replacement method.
Request Handling Statistics Reference	Provides interfaces, classes, and structures that can be used when you need to gather request-handling statistics for your Web applications or XML Web services.
Extension Management Reference	Provides interfaces, classes, structures, and macros that can be used for management functions for your ISAPI extension that are accessible using SOAP or an HTML front end.
Shared Services Reference	Provides interfaces, classes, and macros that can be used to provide an XML Web service for caching shared data.
Session-State Reference	Provides classes and interfaces that allow data to be stored in memory or a database using a common programmatic interface.
SOAP Reference	Provides attributes, classes, structures, and enums that give you code for implementing XML Web services as SOAP request handlers and for making requests as a client to XML Web services written in any language.
Caching Reference	Provides classes, macros, structures, and interfaces cache arbitrary data, files, stencils, and DLLs in atlcache.h.
Cache Cullers Reference	Provides classes and structures use the cache culler archetype and provides code for automatic removal of cache items based on time.
Cache Flushers Reference	Provides classes and structures use the cache flusher archetype and provides code for removal of ache items based on access patterns.
Cache Statistics Reference	Provides classes, interfaces, and macros that use the cache statistics archetype and provide code for collecting cache statistics.
Cryptographic Reference	Provides classes that are a collection of thin wrappers over the CryptoAPI for accessing the cryptographic services provided by Windows and third parties in atlcrypt.h.
Encoding Reference	Provides functions and macros that support encoding in a range of common Internet standards such as Base64, uuencode, hexadecimal, and UTF8 in atlenc.h.
HTML Generation Reference	Provides classes, structures, and macros that give you a lightweight way of writing HTML text to a stream in atlhtml.h.

HTTP Client Reference	Provides code in atlhttp.h that allows generating HTTP responses and requests, allowing lightweight HTTP clients with minimal dependencies to be written in C++.
Performance Monitoring Reference	Provides attributes, classes, and macros in atlperf.h that let you expose performance monitor counters from your applications and services.
MIME & SMTP Reference	Provides classes, enums, and functions that support MIME (Multipurpose Internet Mail Extensions), the standard way of packaging files for transmission by e-mail and HTTP, and SMTP (Simple Mail Transfer Protocol),the most common way of sending e-mail, in atlmime.h and atlsmtpconnection.h.
Regular Expression Reference	Provides classes, typedefs, and enums in atlrx.h with code that can search text using patterns.
Utilities Reference	Provides code for manipulating paths and URLs in the form of CPathT and CUrl. A thread pool, CThreadPool, is used in the implementation of the ATL Server ISAPI extension class and can also be used in your own applications. This code can be found in atlpath.h and atlutil.h.
Stream Helpers Reference	Provides a number of classes that implement the IStream interface by writing the data to another stream, a WinInet handle, a socket, or a string.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

ATL Server Framework Reference

The ATL Server framework classes provide the infrastructure used by ATL Server Web applications and XML Web services for integrating with Internet Information Services (IIS), parsing HTTP requests and generating responses. These classes allow the use of Web application DLLs for partitioning functionality within an application and server response files for separating dynamic and static content. The framework classes are typically found in atlisapi.h and atlstencil.h.

Classes

CAtlIsapiBuffer Class	This class is an efficient buffer that manages its own memory and is used by the ATL Server code to buffer data in CHttpResponse and CAtlHttpClient.
CAtlValidator Class	This class provides a static function for determining whether a value lies within a particular range.
CBrowserCapsSvc Class	This class allows you to get access to objects that provide information about the capabilities of a particular user agent (Web browser).
CCacheServerContext Class	This class provides the server context for request handlers that request page-caching support.
CCookie Class	This class represents a cookie to be sent from the server to a client or a cookie that has been returned by a client to the originating server.
CDefaultAuth Class	This class allows you to check whether the user making an HTTP request is in a particular group.
CHtmlStencil Class	This class is used to parse a server response file and generate a text stream by replacing elements found within the server response file with output generated by an object implementing the ITagReplacer interface.
CHtmlTagReplacer Class	This class acts as the link between CStencil-derived classes and CRequestHandlerT and provides stencil caching.
CHttpMap Class	This class is a simple map class that allows the contents of the map to be shared.
CHttpPtrMap Class	This class represents a map of pointers.
CHttpRequest Class	This class represents an HTTP request received by the server.
CHttpRequestFile Class	This class represents an uploaded file and implements the IHttpFile interface.
CHttpRequestParams Class	This class represents a collection of name-value pairs such as query parameters or form variables.
CHttpResponse Class	This class represents an HTTP response that can be sent back to the client.
CIDServerContext Class	This class provides an implementation of IHttpServerContext for use by a subhandler.
CIncludeServerContext Class	This class provides an implementation of IHttpServerContext for use by included request handlers.
CIsapiExtension Class	This class provides the code necessary for implementing the ISAPI extension DLL that ATL Server uses to pass requests from IIS to your Web application DLLs.

CIsapiWorker Class	Receives ATL Server requests from the thread pool and passes them on to an object implementing IIsapiExtension.
CMultiPartFormParser Class	Use this class to read multipart/form-data from the associated server context and generate files as necessary from the data in the body of the request.
CPageCachePeer Class	This class is used internally by ATL Server to manage auxiliary information for cached responses returned by request handlers.
CServerContext Class	This class provides an implementation of IHttpServerContext for use within an ISAPI extension DLL or a primary request handler.
CSessionCookie Class	This class represents a cookie that can be used to identify a user's session.
CStencil Class	This class is used to parse a stencil and generate a text stream by replacing elements found within the stencil by output generated by an object implementing the ITagReplacer interface.
CTransferServerContext Class	This class provides an implementation of IHttpServerContext for use by request handlers that have had control passed to them by a call to IIsapiExtension::TransferRequest.
CValidateContext Class	This class represents a collection of validation status codes. Use this class in combination with CValidateObject to validate forms variables, cookies, or query parameters and to build up a collection of failures (and, optionally, successes).
CValidateObject Class	This class provides methods for retrieving and validating named values.
CVerifyAuth Class	This class allows you to authorize the client making a request to the application.
CWrappedServerContext Class	This class provides an implementation of IHttpServerContext that delegates all the methods to another object implementing the same interface.
CWriteStreamHelper Class	This class provides a convenient way of writing to the IWriteStream interface.
ITagReplacerImpl Class	Implementation of the ITagReplacer interface.

Structures

ATLServerDllInfo Structure	This class holds pointers to the three functions exported by an ATL Server request handler DLL along with pointers to the ISAPI extension and server context.
AtlServerRequest Structure	This structure represents an HTTP request in the ATL Server infrastructure.
CDefaultErrorProvider Structure	This class provides access to error information that can be displayed to the user or used for diagnostic messages.
CStencilState Structure	This class provides state information for use when rendering stencils using CStencil and CHtmlStencil.
CTagReplacerMethodEntry Structure	This structure holds information about a replacement method including its name and pointers to the member functions that implement it.
CTagReplacerMethods Structure	This structure holds pointers to the member functions necessary for implementing a replacement method.

CTagReplacerMethodsEx Structure	This structure provides static functions for casting type-specific replacement methods and parse methods to generic versions that can be stored in the CTagReplacerMethodEntry structure.
StencilToken Structure	Use this structure to store information about a token in CStencil or derived classes.

Typedefs

INITIALIZEATLHANDLERS	A pointer to the function exported by ATL Server Web application DLLs for initialization before any request handlers are created.
PFnAsyncComplete	A function pointer suitable for handling callbacks on completion of asynchronous IO.
PFnHandleRequest	A member function pointer compatible with IRequestHandler::HandleRequest.
UNINITIALIZEATLHANDLERS	A pointer to the function exported by ATL Server Web application DLLs for cleaning up when all request handlers have been released.
HTTP_CODE	Used throughout ATL Server to describe the success of a method call.

Enums

ATLSRV_REQUESTTYPE	The members of this enumeration describe the type of an ATL Server request.
ATLSRV_STATE	The members of this enumeration describe the state of an ATL Server request.

Functions

AtlImpersonateClient	Call this function to change the current thread's token to impersonate the client making the current request.
AtlsSecErrHandlerFunc	This function is used by ATL Server projects as the security error handler that is called in response to a buffer overrun.
CreateRequestHandlerSync	Call this function to create a request handler to handle a synchronous call.
CreateServerContext	Call this function to create a new CServerContext.
EscapeToCString	Call this function to URL-encode a string and have the result appended to a **CString** passed to the function by reference.
GetScriptFullFileName	Call this function to retrieve the full canonical physical path of a file relative to the current script.
GetStatusHeader	Call this function to obtain a status header (for example, 200 OK) that can be returned in the header of an HTTP response.
ReadClientData	Call this function to read data from the client making the current request.
RenderError	Call this function to write an error response to the client.
IsNullByType	Specialize this function to define what it means for a type to be null when used with CValidateObject.
AtlsHttpError	Call this function to create an HTTP_CODE from an HTTP status code and a subcode.
IsAsyncContinueStatus	Call this function to determine whether an ATL Server HTTP_CODE status code indicates that asynchronous processing is still in progress.

IsAsyncDoneStatus	Call this function to determine whether an ATL Server status code indicates that asynchronous processing has completed.
IsAsyncFlushStatus	Call this function to determine whether a status code indicates that asynchronous processing has occurred and ATL Server code is responsible for flushing.
IsAsyncNoFlushStatus	Call this function to determine whether a status code indicates that asynchronous processing has occurred and ATL Server code is not responsible for flushing.
IsAsyncStatus	Call this function to determine whether a status code indicates that asynchronous processing has occurred.

Macros

ATL_DEFAULT_DLL_EXTENSION	This macro defines the default file extension used to identify request handler DLLs.
ATL_DEFAULT_PRECISION	This macro defines the default precision used by CWriteStreamHelper for writing floating-point values.
ATL_DEFAULT_STENCIL_EXTENSION	This macro defines the default file extension used to identify server response files.
ATL_EPSILON	This macro defines a small value used by CAtlValidator for comparing the equality of floating point numbers.
ATL_ISAPI_BUFFER_SIZE	This macro defines the initial size in bytes of the static buffer used by CAtlIsapiBuffer.
ATL_MAX_HANDLER_NAME_LEN	This macro defines the maximum length that can be used for the alias of a subhandler in a subhandler tag.
ATLS_ASYNC_MUTEX_TIMEOUT	This macro defines the maximum timeout in milliseconds for the asynchronous guard mutex.
ATLS_WORKER_HEAP_SIZE	This macro defines the initial size of the heap used by CIsapiWorker.
DEFAULT_MAX_FORM_SIZE	This macro defines the default maximum form size in bytes.
MAX_TOKEN_LENGTH	This macro defines the maximum allowed size in bytes for the name of an HTML form element that can be handled by ATL Server.
SESSION_COOKIE_NAME	This macro is used as the name of the cookie used for tracking session-state data.
HTTP_CODE Subcodes	These macros are used as subcodes to control the actions that the ATL Server framework takes when returning from a request handler method or to distinguish one error code from another.
HTTP_CODE Status Codes	These macros provide symbols for commonly used ATL Server status codes.
ATL_MAX_BLOCK_STACK	This macro defines the maximum number of levels that can be used for nesting replacement tags.
ATL_MAX_METHOD_NAME_LEN	This macro defines the maximum length that can be used for the name of a replacement method.
Stencil Tokens	These constants identify different tokens recognized by the CStencil and CHtmlStencil classes.

STENCIL_USER_TOKEN_BASE	Use this value as the starting point for constants used to identify custom tokens in CStencil-derived classes.
Validation Results	These macros are returned as results from the Exchange or Validate methods of CValidateObject.
VALIDATION_SUCCEEDED	Use this macro to determine whether validation succeeded or failed.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Request Handling Reference

These attributes, classes, and macros provide the features that you will use directly for hooking your request handler into ATL Server's framework.

Attributes

request_handler	Apply this attribute to a class to expose it as an ATL Server request handler and enable it to handle HTTP requests.

Classes

CRequestHandlerT Class	This class acts as the base class for all user request handlers.
IRequestHandlerImpl Class	This class provides a default implementation of the IRequestHandler interface.

Enums

ATL_FORM_FLAGS	These flags describe how form processing should be carried out.

Macros

BEGIN_HANDLER_MAP	Use this macro in a request handler DLL to mark the start of a handler map.
DECLARE_ASYNC_HANDLER	Add this macro to the class definition of your request handler if you need to write to the client asynchronously.
DECLARE_ASYNC_HANDLER_EX	Add this macro to the class definition of your request handler if you need to write to the client asynchronously.
DECLARE_REQUEST_HANDLER	Add this macro to your project to map a request handler name to the class responsible for handling the corresponding requests.
END_HANDLER_MAP	Use this macro in a request handler DLL to mark the end of a handler map.
HANDLER_ENTRY	This macro is a synonym for the DECLARE_REQUEST_HANDLER macro.
HANDLER_ENTRY_EX	This macro is a synonym for the DECLARE_REQUEST_HANDLER macro.
HANDLER_ENTRY_SDL	Add this macro to your project to map a request handler name to the class responsible for handling the corresponding requests and expose the XML Web service Description Language (WSDL) for the handler.
Handler Flags	These macros are used as flags to be returned from a request handler's GetHandlerFlags method.
Request Initialization Flags	These macros are used as flags to be returned from a request handler's GetFlags method.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples
ATL Server Framework Reference

Replacement Method Reference

These attributes and macros provide the plumbing necessary to implement a replacement method.

Attributes

tag_name	Apply this attribute to a method in a request handler to expose it as a replacement method associated with a tag name.

Macros

BEGIN_REPLACEMENT_METHOD_MAP	Use this macro in a request handler class to mark the start of a replacement method map.
END_REPLACEMENT_METHOD_MAP	Use this macro in a request handler class to mark the end of a replacement method map.
REPLACEMENT_METHOD_ENTRY	Use this macro in a replacement method map to associate a replacement method with a particular tag name.
REPLACEMENT_METHOD_ENTRY_EX	Use this macro in a replacement method map to associate a replacement method with a particular tag name. Use when the replacement method needs to receive method arguments.
REPLACEMENT_METHOD_ENTRY_EX_STR	Use this macro in a replacement method map to associate a replacement method with a particular tag name. Use when the replacement method needs to receive method arguments in the form of a char* parameter.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples
ATL Server Framework Reference

Request Handling Statistics Reference

These interfaces, classes, and structures can be used when you need to gather request-handling statistics for your Web applications or services.

Interfaces

IRequestStats Interface	This interface provides methods to get statistics on how requests are being handled.

Classes

CNoRequestStats Class	Use this class as the argument for the *CRequestStatClass* template parameter to CIsapiExtension if you do not want to collect request-handling statistics.
CPerfMonRequestStats Class	Use this class as the argument for the *CRequestStatClass* template parameter to CIsapiExtension if you want to expose request-handling statistics as performance monitor counters.
CRequestPerfMon Class	This class is used in the implementation of CPerfMonRequestStats.
CStdRequestStats Class	Use this class as the argument for the *CRequestStatClass* template parameter to CIsapiExtension if you want to capture request-handling statistics.

Structures

CPerfRequestStatObject Structure	This structure is used in the implementation of CPerfMonRequestStats.
CRequestStats Class	This structure is used to collect request-handling statistics.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples
ATL Server Framework Reference

Extension Management Reference

ATL Server provides some request handlers that can be used to provide management functions for your ISAPI extension that are accessible using SOAP or an HTML front end. The code is found in atlextmgmt.h.

Interfaces

IDllCacheMgr Interface	This interface provides methods for managing a DLL cache.
IStencilCacheMgr Interface	This interface provides methods for managing a stencil cache.
IThreadPoolMgr Interface	This interface provides methods for managing a thread pool.

Classes

CDllCacheManager Class	This class implements the XML Web service that allows you to manage the DLL cache in an ATL Server ISAPI extension.
CDllCacheMgr Class	This class implements the request handler that allows you to manage the DLL cache in an ATL Server ISAPI extension using an HTML interface.
CDllMgrObject Class	This class provides methods that can be used to implement IDllCacheMgr by wrapping calls to IDllCache.
CStencilCacheManager Class	This class implements the XML Web service that allows you to manage the stencil cache in an ATL Server ISAPI extension.
CStencilCacheMgrObject Class	This class provides methods that can be used to implement IStencilCacheMgr by forwarding calls to implementations of IStencilCacheControl and IMemoryCacheStats.
CStencilMgr Class	This class implements the request handler that allows you to manage the stencil cache in an ATL Server ISAPI extension using an HTML interface.
CThreadMgrStencil Class	This class implements the request handler that allows you to manage the thread pool in an ATL Server ISAPI extension using an HTML interface.
CThreadPoolManager Class	This class implements the XML Web service that allows you to manage the thread pool in an ATL Server ISAPI extension.
CThreadPoolMgrObject Class	This class provides methods that can be used to implement IThreadPoolMgr by forwarding calls to an implementation of IThreadPoolConfig.

Structures

_DLL_CACHE_ENTRY Structure	This structure represents an entry in a DLL cache.

Macros

ATL_DEFAULT_AUTHGRP	This macro defines the group used for authorizing access to ATL Server extension management services. Only members of this group can use the Web-based management interfaces to configure the behavior of ATL Server ISAPI extension DLLs.
ID_DLLCACHEMGR_SRFHANDLER_NAME	This constant defines the name of the request handler that implements the HTML interface for managing the DLL cache.

ID_DLLCACHEMGR_WEBSERVICE_NAME	This constant defines the name of the XML Web service that exposes the SOAP interface for managing the DLL cache and the request handler that implements it.
ID_DLLCACHEMGR_WEBSERVICE_URL	This constant defines the namespace of the XML Web service that exposes the SOAP interface for managing the DLL cache.
ID_DLLCACHEMGR_WEBSERVICE_WSDL	This constant defines the name of the WSDL handler for the XML Web service that exposes the SOAP interface for managing the DLL cache.
ID_STENCILCACHEMGR_SRFHANDLER_NAME	This constant defines the name of the request handler that implements the HTML interface for managing the stencil cache.
ID_STENCILCACHEMGR_WEBSERVICE_NAME	This constant defines the name of the XML Web service that exposes the SOAP interface for managing the stencil cache and the request handler that implements it.
ID_STENCILCACHEMGR_WEBSERVICE_URL	This constant defines the namespace of the XML Web service that exposes the SOAP interface for managing the stencil cache.
ID_STENCILCACHEMGR_WEBSERVICE_WSDL	This constant defines the name of the WSDL handler for the XML Web service that exposes the SOAP interface for managing the stencil cache.
ID_THREADMGR_SRFHANDLER_NAME	This constant defines the name of the request handler that implements the HTML interface for managing the thread pool.
ID_THREADMGR_WEBSERVICE_NAME	This constant defines the name of the XML Web service that exposes the SOAP interface for managing the thread pool and the request handler that implements it.
ID_THREADMGR_WEBSERVICE_URL	This constant defines the namespace of the XML Web service that exposes the SOAP interface for managing the thread pool.
ID_THREADMGR_WEBSERVICE_WSDL	This constant defines the name of the WSDL handler for the XML Web service that exposes the SOAP interface for managing the thread pool.
IDR_DLLMGR_SRF	This constant defines the name of the stencil resource that provides the HTML interface for managing the DLL cache.
IDR_STENCILMGR_SRF	This constant defines the name of the stencil resource that provides the HTML interface for managing the stencil cache.
IDR_THREADMGR_SRF	This constant defines the name of the stencil resource that provides the HTML interface for managing the thread pool.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples
ATL Server Framework Reference

Shared Services Reference

ATL Server provides an XML Web service in atlsharedsvc.h that can be used to cache shared data.

Interfaces

ISharedBlobCache Interface	This SOAP-compatible interface provides methods for adding items to and retrieving items from a cache.

Classes

CSharedCache Class	This class implements the ISharedBlobCache interface using CBlobCache.
CSharedCacheHandler Class	This class implements an XML Web service that allows access to a cache that can be shared by SOAP clients.

Macros

ATL_SHAREDBLOBCACHE_TIMEOUT	This macro defines the default value in 100-nanosecond intervals used as a timeout by CSharedCache and CSharedCacheHandler.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples
ATL Server Framework Reference

Session-State Reference

The session-state classes and interfaces allow data to be stored in memory or a database using a common programmatic interface. This code can be found in atlsession.h.

Interfaces

ISession Interface	This interface allows the storage and retrieval of session-specific data.
ISessionStateService Interface	This interface provides methods for creating, retrieving, and closing session objects.
ISessionStateControl Interface	This interface provides methods for controlling ATL Server session-state settings.

Classes

CDefaultQueryClass Class	This class provides access to the SQL queries used for database-backed session-state.
CSessionStateService Class	Provides support for the maintenance and retrieval of session-state data.
CMemSession Class	This class provides an implementation of ISession that stores the session variables in memory .
CMemSessionServiceImpl Class	This class provides an in-memory based session-state implementation for use within ATL Server ISAPI extension DLLs.
CDBSession Class	This class template provides an implementation of ISession that stores the session variables in a database.
CDBSessionServiceImplT Class	This class provides a database-backed session-state implementation for use within ATL Server ISAPI extension DLLs.

Macros

ATL_SESSION_SWEEPER_TIMEOUT	This macro defines the frequency at which session data is maintained, in milliseconds.
ATL_SESSION_TIMEOUT	This macro defines the time, in milliseconds, after which an un-accessed session expires.
MAX_SESSION_KEY_LEN	This macro defines the maximum size allowed for a session name, in bytes.
MAX_CONNECTION_STRING_LEN	This macro defines the maximum length of the data source connection string used for session-state data storage.
MAX_VARIABLE_NAME_LENGTH	This macro is defines the maximum size of session-state data variable names.
MAX_VARIABLE_VALUE_LENGTH	This macro is defines the maximum size of session-state data values.
SESSION_COOKIE_NAME	This macro is used as the name of the cookie used for tracking session-state data.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples
ATL Server Framework Reference

SOAP Reference

ATL Server provides code for implementing XML Web services as SOAP request handlers and for making requests as a client to XML Web services written in any language. atlsoap.h contains most of the code implementing these features. The header file atlbase.h must be included before atlsoap.h is included.

Attributes

soap_handler	Apply this attribute to a class to provide the methods necessary for handling SOAP method calls and exposing information about the services offered by this class through WSDL.
soap_header	Apply this attribute to a SOAP method in an XML Web service to specify the data member used to hold the value of a SOAP header.
soap_method	Apply this attribute to a method in an XML Web service to expose the specified member as a SOAP method with a corresponding WSDL description.

Classes

CSoapFault Class	This class represents a SOAP Fault element, which is used to carry error and/or status information.
CSoapMSXMLInetClient Class	This class provides client access to XML Web services using Microsoft XML's **ServerXMLHTTP** object.
CSoapSocketClientT Class	This class provides client access to XML Web services using sockets.
CSoapWininetClient Class	This class provides client access to XML Web services using Microsoft Win32 Internet functions (WinInet).
CSoapHandler Class	This class provides functionality for handling SOAP requests in an XML Web service request handler. The class is injected as a base class by the soap_handler attribute.
CSoapRootHandler Class	This class provides the code necessary to generate and parse SOAP messages. It is used as a base by SPROXY-generated SOAP client classes and XML Web service classes using the soap_handler attribute.
ISAXContentHandlerImpl Class	This class provides a nonfunctional implementation of **ISAXContentHandler** that can be used as a base class by classes that need to provide the binary interface of **ISAXContentHandler** but don't need usable implementations for all the methods.

Structures

ATLSOAP_BLOB Structure	This class represents a binary large object (BLOB) for use with SOAP.

Enums

SOAP_ERROR_CODE	The members of this enumeration provide values corresponding to SOAP faultcodes.
SOAPCLIENT_ERROR	The members of this enumeration provide values corresponding to errors experienced by a SOAP client.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples
ATL Server Framework Reference

Caching Reference

ATL Server provides a range of classes and interfaces in atlcache.h for caching arbitrary data, files, stencils, and DLLs.

Interfaces

IDataSourceCache Interface	This interface provides methods for adding database connections to and retrieving them from a cache of open connections.
IDllCache Interface	This interface provides methods for adding DLLs to and retrieving DLLs from a cache.
IFileCache Interface	This interface provides methods for adding and retrieving files from a cache.
IMemoryCache Interface	This interface provides methods for adding, retrieving, and removing items cached in memory.
IMemoryCacheClient Interface	This interface provides a method that can be called to free an item that is being removed from a cache.
IMemoryCacheControl Interface	This interface provides methods for controlling the size of a cache.
IStencilCache Interface	This interface provides methods for adding stencils to and retrieving stencils from a cache.
IStencilCacheControl Interface	This interface provides methods for controlling a stencil cache.

Classes

CBlobCache Class	This class provides functionality for caching arbitrarily sized data blocks in memory.
CCacheDataBase Class	This is an empty class that is used as the default cache entry type by CMemoryCacheBase.
CDataSourceCache Class	This class caches open data connections and implements the IDataSourceCache interface.
CDllCache Class	This class implements a DLL cache.
CDllCachePeer Class	This class conforms to the DLL cache peer archetype and is used to cache pointers to ATL Server request handler functions.
CFileCache Class	This class implements a file cache.
CMemoryCache Class	This class represents a generic memory cache.
CMemoryCacheBase Class	This class provides features for caching generic items in memory.
CNoDllCachePeer Class	This class conforms to the DLL cache peer archetype and can be used when no extra data needs to be stored with the DLLs in the cache.
CNoFileCachePeer Class	This class conforms to the file cache peer archetype and can be used with CFileCache when no auxiliary information needs to be stored with the entries.
CStencilCache Class	This class implements a stencil cache, and is used by ATL Server to cache preprocessed stencils to speed response processing.

Structures

DLL_CACHE_ENTRY Structure	This structure describes a cached DLL.

CCacheDataEx Structure	This structure is used by CMemoryCache and CStencilCache to store auxiliary data for each item in the cache.
CCacheDataPeer Structure	This class is used by CFileCache to provide an interface through which auxiliary data can be attached to each cached file.

Typedefs

HCACHEITEM	A handle to an item stored in a CMemoryCacheBase-derived class.

Functions

GetDataSource	Call this function to retrieve a cached data source or add it to the cache if not already present.
RemoveDataSource	Call this function to close a cached connection and remove it from the data source cache.

Macros

ATL_BLOB_CACHE_TIMEOUT	This macro defines the cache maintenance frequency for the CBlobCache implementation.
ATL_CACHE_KEY_LENGTH	This macro defines the maximum length of a string used as a cache key.
ATL_DLL_CACHE_TIMEOUT	This macro defines the default value in milliseconds used as the timeout for the DLL cache. The timeout value determines the frequency with which the DLL cache is swept for stale entries.
ATL_DS_CONN_STRING_LEN	This macro defines the maximum length of data source connection names.
ATL_FILE_CACHE_TIMEOUT	This macro defines the cache maintenance frequency for the CFileCache implementation.
ATL_STENCIL_CACHE_TIMEOUT	This macro defines the default value in milliseconds used as the timeout for the stencil cache. The timeout value determines the frequency with which the stencil cache is swept for stale entries.
ATL_STENCIL_CHECK_TIMEOUT	This macro defines the frequency with which ATL Server checks cached stencils against the raw stencil file to determine if the cached version is out of date.
ATL_STENCIL_LIFESPAN	This macro defines a default filetime value used as the life span for stencils in the stencil cache. Stencils are removed from the cache when their life span has been reached whether or not they have been used recently.

See Also

Other Resources

Caching
ATL Server Concepts
ATL Server Reference
ATL Server Samples

Cache Cullers Reference

ATL Server provides classes and structures use the cache culler archetype and provides code for automatic removal of cache items based on time.

Classes

CExpireCuller Class	This class conforms to the cache culler archetype and allows the automatic removal of cache items that exceed the absolute expiration time set by users of the cache.
CFixedLifetimeCuller Class	This class conforms to the cache culler archetype and allows the automatic removal of cache items that have not been accessed in a certain period. The lifetime of items in the cache cannot be changed at runtime. The lifetime is the same for all items.
CLifetimeCuller Class	This class conforms to the cache culler archetype and allows the automatic removal of cache items that haven't been accessed in a certain period. The lifetime of an item can be changed at runtime and can be different for each of the items in the cache.
CNoExpireCuller Class	This class conforms to the cache culler archetype and can be used to disable automated removal of cache items based on expiration times.
COrCullers Class	This class combines the functionality of two cache cullers into a single class.

Structures

CCullerCacheData Structure	This structure holds information about cache entry that can be used by classes conforming to the cache culler archetype to determine whether, and in what order, items should be removed from the cache.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Cache Flushers Reference

ATL Server provides classes and structures use the cache flusher archetype and provides code for removal of ache items based on access patterns.

Classes

CLOUFlusher Class	This class conforms to the cache flusher archetype and allows the automatic removal of cache items based on their access pattern. Items that are used infrequently are presented for removal before items that are us ed more often.
CLRUFlusher Class	This class conforms to the cache flusher archetype and allows the automatic removal of cache items based on their access pattern. Items that have not been used for a long time are presented for removal before ite ms that have been used more recently.
CNoFlusher Class	This class conforms to the cache flusher archetype and can be used to disable automated removal of cache items based on access patterns.
COldFlusher Class	This class conforms to the cache flusher archetype and allows the automatic removal of cache items based on their access pattern. Items are presented for removal in the order in which they were added to the cach e.
COrFlushers Class	This class combines the functionality of two cache flushers into a single class.

Structures

CFlusherCacheData Structure	This structure holds information about an entry in a cache that can be used by classes conformin g to the cache flusher archetype to determine whether, and in what order, items should be remo ved from the cache.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Cache Statistics Reference

ATL Server provides classes, interfaces, and macros that use the cache statistics archetype and provide code for collecting cache statistics.

Interfaces

IMemoryCacheStats Interface	This interface provides methods for retrieving current and historical information about the state of a memory-based cache.

Classes

CCachePerfMon Class	This class is used in the implementation of CPerfStatClass.
CNoStatClass Class	Use this class anywhere that a class conforming to the cache statistics archetype is expected if you do not want to collect cache statistics.
CPerfStatClass Class	Use this class anywhere that a class conforming to the cache statistics archetype is expected if you want to collect cache statistics in the class's data members and expose the information as performance monitor counters.
CPerfStatObject Structure	This structure is used in the implementation of CStdStatClass and CPerfStatClass.
CStdStatClass Class	Use this class anywhere that a class conforming to the cache statistics archetype is expected if you want to collect cache statistics in the class's data members.

Macros

ATL_PERF_CACHE_OBJECT	This macro defines the ID for the cache performance monitoring object definition.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Cryptographic Reference

Atlcrypt.h contains a collection of thin wrappers over the CryptoAPI for accessing the cryptographic services provided by Windows and third parties.

Classes

CCryptDerivedKey Class	This class represents a cryptographic session key that is derived from a hashed password.
CCryptHash Class	This class represents a generic hash.
CCryptHMACHash Class	This class represents a Hashed Message Authentication Code (HMAC) keyed hash.
CCryptImportKey Class	This class represents a cryptographic key created by importing a key BLOB.
CCryptKey Class	This class represents a generic cryptographic key.
CCryptKeyedHash Class	This class represents a keyed hash, such as a Hashed Message Authentication Code (HMAC) o r a Message Authentication Code (MAC) hash.
CCryptMACHash Class	This class represents a Message Authentication Code (MAC) keyed hash.
CCryptMD2Hash Class	This class represents an MD2 hash.
CCryptMD4Hash Class	This class represents an MD4 hash.
CCryptMD5Hash Class	This class represents an MD5 hash.
CCryptProv Class	This class represents a cryptographic service provider (CSP).
CCryptRandomKey Class	This class represents a randomly generated cryptographic session key.
CCryptSHAHash Class	This class represents a Secure Hash Algorithm (SHA) hash.
CCryptSSL3SHAMD5Hash Class	This class represents an SSL3 client authentication hash.
CCryptUserExKey Class	This class represents the user's exchange key pair (used to encrypt and decrypt session keys).
CCryptUserSigKey Class	This class represents the user's signature key pair (used to create and verify digital signatures).

Typedefs

CCryptSHA1Hash	Synonym for CCryptSHAHash.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Encoding Reference

Encoding in a range of common Internet standards such as Base64, uuencode, hexadecimal, and UTF8 is supported by the code found in atlenc.h.

Functions

AtlGetHexValue	Call this function to get the numeric value of a hexadecimal digit.
AtlHexDecode	Decodes a string of data that has been encoded as hexadecimal text such as by a previous call to AtlHexEncode.
AtlHexDecodeGetRequiredLength	Call this function to get the size in bytes of a buffer that could contain data decoded from a hex-encoded string of the specified length.
AtlHexEncode	Call this function to encode some data as a string of hexadecimal text.
AtlHexEncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.
AtlUnicodeToUTF8	Call this function to convert a Unicode string to UTF-8.
Base64Decode	Decodes a string of data that has been encoded in base64 format such as by a previous call to Base64Encode.
Base64DecodeGetRequiredLength	Call this function to get the size in bytes of a buffer that could contain data decoded from a base64-encoded string of the specified length.
Base64Encode	Call this function to base64-encode some data.
Base64EncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.
BEncode	Call this function to convert some data using the "B" encoding.
BEncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.
EscapeXML	Call this function to convert characters that are unsafe for use in XML to their safe equivalents.
GetExtendedChars	Call this function to get the number of extended characters in a string.
IsExtendedChar	Call this function to find out if a given character is an extended character (less than 32, greater than 126, and not a tab, linefeed or carriage return)
QEncode	Call this function to convert some data using the "Q" encoding.
QEncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.
QPDecode	Decodes a string of data that has been encoded in quoted-printable format such as by a previous call to QPEncode.

QPDecodeGetRequiredLength	Call this function to get the size in bytes of a buffer that could contain data decoded from q uoted-printable-encoded string of the specified length.
QPEncode	Call this function to encode some data in quoted-printable format.
QPEncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.
UUDecode	Decodes a string of data that has been uuencoded such as by a previous call to UUEncode.
UUDecodeGetRequiredLength	Call this function to get the size in bytes of a buffer that could contain data decoded from a uuencoded string of the specified length.
UUEncode	Call this function to uuencode some data.
UUEncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.

Macros

ATL_BASE64 Flags	These flags describe how base64-encoding is to be performed by Base64Encode.
ATL_ESC Flags	These flags are used to control the behavior of EscapeXML.
ATLSMTP_QPENCODE Flags	These flags describe how quoted-printable encoding is to be performed by QPEncode.
ATLSMTP_UUENCODE Flags	These flags describe how uuencoding is to be performed by UUEncode.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

HTML Generation Reference

ATL Server provides a lightweight way of writing HTML text to a stream in atlhtml.h.

Classes

AtlHtmlAttrs Class	Use this class to build up strings of HTML attributes.
CHtmlGen Class	This class is used to generate HTML elements in an output stream.
CHtmlGenBase Class	This class is used to generate HTML elements in a text stream.
CSimpleStack Class	This class represents a simple stack whose elements are stored in a fixed size array.
CStreamFormatter Class	This class writes formatted text to a stream.

Structures

ATL_HTML_TAG Structure	This structure associates an HTML tag name with flags describing properties of that tag.
HTML_SCHEME Structure	This class provides HTML formatting information for use with CHtmlGenBase.

Macros

ATL_HTML_TAGS	An enumeration of HTML tags.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

HTTP Client Reference

In addition to providing the code for generating HTTP responses, ATL Server provides code in atlhttp.h for generating HTTP requests allowing lightweight HTTP clients with minimal dependencies to be written in C++.

Interfaces

IAuthInfo Class	This interface defines methods used to retrieve authentication credentials required for accessing protected URLs.

Classes

CAtlHttpClientT Class	This class provides support for making requests and receiving responses from HTTP servers.
CAtlNavigateData Class	This class represents HTTP transaction options for use with the CAtlHttpClientT class.
CAtlBaseAuthObject Class	This is an abstract base class that defines the interface for HTTP client authorization objects.
CBasicAuthObject Class	This class implements support for performing HTTP navigation using the BASIC authentication scheme.
CNTLMAuthObject Class	This class implements support for performing HTTP navigation using the NTLM authentication scheme.

Structures

ATL_NAVIGATE_DATA Structure	This class represents HTTP client settings as required by CAtlHttpClientT.

Typedefs

CAtlHttpClient	A synonym for a class that provides HTTP client support based on the CAtlHttpClientT class.
PFNATLCHUNKEDCB	A pointer to a function suitable for use as a callback used to prepare chunked-encoded HTTP requests.
PFNATLSTATUSCALLBACK	A pointer to a function suitable for use as a callback used to monitor either HTTP request or response transfer status.

Macros

ATL_HTTP_AUTHTYPE_BASIC	This macro defines a string that can be used to indicate the BASIC HTTP authorization scheme.
ATL_HTTP_AUTHTYPE_NTLM	This macro defines a string that can be used to indicate the NTLM authorization scheme.
ATL_HTTP_DEFAULT_BLOCK_SIZE	This macro represents the default memory block sizes used for both incoming and out-going HTTP transaction buffers.
ATL_HTTP_FLAG_AUTO_REDIRECT	This macro is a behavior modification flag for HTTP request transactions that specifies whether redirections should be pursued automatically.
ATL_HTTP_FLAG_INVALID_FLAGS	This macro can be used to test for an invalid state of the HTTP behavior-control flags used by the CAtlNavigateData and CAtlHttpClientT classes.
ATL_HTTP_FLAG_PROCESS_RESULT	This macro is a behavior modification flag for HTTP request transactions that specifies whether additional actions, such as redirection or authorization, should be pursued.
ATL_HTTP_FLAG_SEND_CALLBACK	This macro is a behavior modification flag that specifies whether HTTP requests should be chunk-transfer encoded.

ATL_HTTP_FLAG_SEND_BLOCKS	This macro is a behavior modification flag that indicates default HTTP request behavior.
ATL_HTTP_USERAGENT	This macro defines the user agent string that is embedded into each request.
ATL_HTTP_METHOD_GET	This macro defines a string that indicates HTTP requests based on the GET method.
ATL_HTTP_METHOD_POST	This macro defines a string that indicates HTTP requests based on the POST method.
ATL_INVALID_STATUS	This macro defines a value that represents an invalid HTTP response status code.

See Also

Concepts

HTTP Client Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Samples

Performance Monitoring Reference

Atlperf.h contains attributes and macros that let you expose Performance Monitor counters from your applications and services.

Attributes

perfmon	Apply this attribute to a class to define a performance manager object.
perf_object	Apply this attribute to a class to define a performance monitor object.
perf_counter	Apply this attribute to a data member in a perf_object class to expose it as a performance counter.

Classes

CPerfMon Class	This class represents a performance monitor object manager.
CPerfObject Structure	This structure represents a performance monitor object.
CPerfLock Class	This class manages the lock state of performance monitor objects.

Macros

ATLPERF_DEFAULT_MAXINSTNAMELENGTH	Use this macro to define the maximum name length of object instance strings.
BEGIN_COUNTER_MAP	Use this macro to begin performance counter definition maps within CPerfObject-derived classes.
BEGIN_PERF_MAP	Use this macro to preface performance object map definitions.
CHAIN_PERF_OBJECT	Use this macro to attach a performance object class to a performance object management class (a class derived from CPerfMon).
DECLARE_PERF_OBJECT	Use this macro to declare a new performance object from within a CPerfObject-derived class.
DECLARE_PERF_OBJECT_EX	Use this macro to declare a new performance object from within a CPerfObject-derived class.
DECLARE_PERF_OBJECT_NO_INSTANCES	Use this macro to declare a new instanceless performance object from within a CPerfObject-derived class.
DEFINE_COUNTER	Use this macro to define performance monitor counters within CPerfMon-derived classes.
DEFINE_COUNTER_EX	Use this macro to define performance monitor counters within CPerfMon-derived classes.
END_COUNTER_MAP	Use this macro to terminate performance counter map definitions.
END_PERF_MAP	Use this macro to end a performance map.
PERFREG_ENTRY	Use this macro to register CPerfMon-derived classes with the Windows performance monitoring subsystem.

See Also

Concepts

Performance Monitoring

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Samples

MIME & SMTP Reference

MIME (Multipurpose Internet Mail Extensions) is the standard way of packaging files for transmission by email and HTTP. SMTP (Simple Mail Transfer Protocol) is the most common way of sending email. Both of these standards are supported by the code in atlmime.h and atlsmtpconnection.h.

Classes

CMimeAttachment Class	This class represents an attachment in a MIME-encoded message.
CMimeBodyPart Class	This class represents a body part of a MIME-encoded message.
CMimeFileAttachment Class	This class represents a file attachment in a MIME-encoded message.
CMimeHeader Class	This class represents a MIME-encoded message header.
CMimeMessage Class	This class represents a MIME-encoded message.
CMimeRawAttachment Class	This class represents an attachment of raw data in a MIME-encoded message.
CMimeText Class	This class represents the body text in a MIME-encoded message.
CSMTPConnection Class	This class represents a connection to an SMTP service.

Enums

ATL_MIME_PRIORITY	The members of this enumeration provide constants for the priorities of a MIME message.

Functions

GetContentTypeFromFileName	Call this function to get the MIME content type of a file based on its name.
SetRfc822Time	Call this function to get the current system time as a string in RFC 822 format.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Regular Expression Reference

The regular expression classes in atlrx.h provide the ability to search text using patterns.

Note
Do not to allow client-side users to create regular expressions, as they have potentially unbounded processing cost.

Classes

CAtlRECharTraitsA Class	This class represents an ANSI character traits object.
CAtlRECharTraitsMB Class	This class represents a multibyte character traits object.
CAtlRECharTraitsW Class	This class represents a Unicode character traits object.
CAtlRegExp Class	This class represents a regular expression.
CAtlREMatchContext Class	This class represents a match context.

Typedefs

CAtlRECharTraits	The default character traits type to use with CAtlRegExp.

Enums

REParseError	The members of this enumeration provide values corresponding to errors returned by CAtlRegExp::Parse.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Utilities Reference

ATL Server provides code for manipulating paths and URLs in the form of CPathT and CUrl. A thread pool, CThreadPool, is used in the implementation of the ATL Server ISAPI extension class and can also be used in your own applications. This code can be found in atlpath.h and atlutil.h.

Classes

CPathT Class	This class represents a path.
CDebugReportHook Class	Use this class to send debug reports to a named pipe.
CNonStatelessWorker Class	Receives requests from a thread pool and passes them on to a worker object that is created and destroyed on each request.
CNoWorkerThread Class	Use this class as the argument for the *MonitorClass* template parameter to CBlobCache, CStencilCache, or other cache classes if you want to disable dynamic cache maintenance.
CThreadPool Class	This class provides a pool of worker threads that process a queue of work items.
CUrl Class	This class represents a URL. It allows you to manipulate each element of the URL independently of the others whether parsing an existing URL string or building a string from scratch.
CWorkerThread Class	This class creates a worker thread or uses an existing one, waits on one or more kernel object handles, and executes a specified client function when one of the handles is signaled.

Typedefs

CPath	A specialization of CPathT using **CString**.
CPathA	A specialization of CPathT using **CStringA**.
CPathW	A specialization of CPathT using **CStringW**.
ATL_URL_PORT	The type used by CUrl for specifying a port number.

Enums

ATL_URL_SCHEME	The members of this enumeration provide constants for the schemes understood by CUrl.

Functions

AtlCanonicalizeUrl	Call this function to canonicalize a URL, which includes converting unsafe characters and spaces into escape sequences.
AtlCombineUrl	Call this function to combine a base URL and a relative URL into a single, canonical URL.
AtlEscapeUrl	Call this function to convert all unsafe characters to escape sequences.
AtlGetDefaultUrlPort	Call this function to get the default port number associated with a particular internet protocol or scheme.
AtlHexValue	Call this function to get the numeric value of a hexadecimal digit.
AtlIsUnsafeUrlChar	Call this function to find out whether a character is safe for use in a URL.
AtlUnescapeUrl	Call this function to convert escaped characters back to their original values.

SafeStringCopy	Call this function to copy a string into a fixed size character buffer.
SystemTimeToHttpDate	Call this function to convert a system time to a string in a format suitable for using in HTTP header s.
RSkipSpace	Returns the pointer to the previous non-whitespace character.
SkipSpace	Returns the pointer to the previous non-whitespace character.
ATLPath::AddBackslash	This function is an overloaded wrapper for PathAddBackslash.
ATLPath::AddExtension	This function is an overloaded wrapper for PathAddExtension.
ATLPath::Append	This function is an overloaded wrapper for PathAppend.
ATLPath::BuildRoot	This function is an overloaded wrapper for PathBuildRoot.
ATLPath::Canonicalize	This function is an overloaded wrapper for PathCanonicalize.
ATLPath::Combine	This function is an overloaded wrapper for PathCombine.
ATLPath::CommonPrefix	This function is an overloaded wrapper for PathCommonPrefix.
ATLPath::CompactPath	This function is an overloaded wrapper for PathCompactPath.
ATLPath::CompactPathEx	This function is an overloaded wrapper for PathCompactPathEx.
ATLPath::FileExists	This function is an overloaded wrapper for PathFileExists.
ATLPath::FindExtension	This function is an overloaded wrapper for PathFindExtension.
ATLPath::FindFileName	This function is an overloaded wrapper for PathFindFileName.
ATLPath::GetDriveNumber	This function is an overloaded wrapper for PathGetDriveNumber.
ATLPath::IsDirectory	This function is an overloaded wrapper for PathIsDirectory.
ATLPath::IsFileSpec	This function is an overloaded wrapper for PathIsFileSpec.
ATLPath::IsPrefix	This function is an overloaded wrapper for PathIsPrefix.
ATLPath::IsRelative	This function is an overloaded wrapper for PathIsRelative.
ATLPath::IsRoot	This function is an overloaded wrapper for PathIsRoot.
ATLPath::IsSameRoot	This function is an overloaded wrapper for PathIsSameRoot.
ATLPath::IsUNC	This function is an overloaded wrapper for PathIsUNC.
ATLPath::IsUNCServer	This function is an overloaded wrapper for PathIsUNCServer.
ATLPath::IsUNCServerShare	This function is an overloaded wrapper for PathIsUNCServerShare.

ATLPath::MakePretty	This function is an overloaded wrapper for PathMakePretty.
ATLPath::MatchSpec	This function is an overloaded wrapper for PathMatchSpec.
ATLPath::QuoteSpaces	This function is an overloaded wrapper for PathQuoteSpaces.
ATLPath::RelativePathTo	This function is an overloaded wrapper for PathRelativePathTo.
ATLPath::RemoveArgs	This function is an overloaded wrapper for PathRemoveArgs.
ATLPath::RemoveBackslash	This function is an overloaded wrapper for PathRemoveBackslash.
ATLPath::RemoveBlanks	This function is an overloaded wrapper for PathRemoveBlanks.
ATLPath::RemoveExtension	This function is an overloaded wrapper for PathRemoveExtension.
ATLPath::RemoveFileSpec	This function is an overloaded wrapper for PathRemoveFileSpec.
ATLPath::RenameExtension	This function is an overloaded wrapper for PathRenameExtension.
ATLPath::SkipRoot	This function is an overloaded wrapper for PathSkipRoot.
ATLPath::StripPath	This function is an overloaded wrapper for PathStripPath.
ATLPath::StripToRoot	This function is an overloaded wrapper for PathStripToRoot.
ATLPath::UnquoteSpaces	This function is an overloaded wrapper for PathUnquoteSpaces.

Macros

ATL_URL Flags	These flags modify the behavior of AtlEscapeUrl and AtlCanonicalizeUrl .
ATL_WORKER_THREAD_WAIT	This macro defines the default value in milliseconds that CWorkerThread::Shutdown will wait for the worker thread to shut down.
ATLS_DEFAULT_THREADPOOLSHUTDOWNTIMEOUT	This macro defines the default time in milliseconds that CThreadPool will wait for a thread to shut down.
ATLS_DEFAULT_THREADSPERPROC	This macro defines the default number of threads per processor used by CThreadPool.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

Stream Helpers Reference

ATL Server provides an number of classes that implement the IStream interface by writing the data to another stream, a WinInet handle, a socket, or a string.

Classes

IStreamImpl Class	This class provides a non-functional implementation of IStream that can be used as a base class by classes that need to provide the binary interface of **IStream** but do not need usable implementations for all the methods.
CReadStreamOnInet Class	This class provides an implementation of IStream that implements Read to return data from a WinInet **HINTERNET** handle returned from a previous call to **InternetOpenUrl**, **FtpOpenFile**, **GopherOpenFile**, or **HttpOpenRequest**.
CReadStreamOnSocket Class	This class provides an implementation of IStream that implements Read to return data from the body of an HTTP response provided by CAtlHttpClientT.
CStreamOnServerContext Class	This class provides an implementation of IStream that delegates Read calls to an IHttpServerContext pointer.
CStreamOnWriteStream Class	This class provides an implementation of IStream that delegates calls to an IWriteStream pointer.
CWriteStreamOnCString Class	This class provides an implementation of IWriteStream that writes the data to a CStringA data member.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

ATL Server Classes

The following classes are included in the ATL Server Library.

Name	Description
_DLL_CACHE_ENTRY Structure	This structure represents an entry in a DLL cache.
ATL_HTML_TAG Structure	This structure associates an HTML tag name with flags describing properties of that tag.
ATL_NAVIGATE_DATA Structure	This class represents HTTP client settings as required by CAtlHttpClientT.
AtlHtmlAttrs Class	Use this class to build up strings of HTML attributes.
ATLServerDllInfo Structure	This class holds pointers to the three functions exported by an ATL Server request handle r DLL along with pointers to the ISAPI extension and server context.
AtlServerRequest Structure	This structure represents an HTTP request in the ATL Server infrastructure.
ATLSOAP_BLOB Structure	This class represents a binary large object (BLOB) for use with SOAP.
CAtlBaseAuthObject Class	This is an abstract base class that defines the interface for HTTP client authorization objec ts.
CAtlHttpClientT Class	This class provides support for making requests and receiving responses from HTTP serv ers.
CAtlIsapiBuffer Class	This class is an efficient buffer that manages its own memory and is used by the ATL Serv er code to buffer data in CHttpResponse and CAtlHttpClient.
CAtlNavigateData Class	This class represents HTTP transaction options for use with the CAtlHttpClientT class.
CAtlRECharTraitsA Class	This class represents an ANSI character traits object.
CAtlRECharTraitsMB Class	This class represents a multibyte character traits object.
CAtlRECharTraitsW Class	This class represents a Unicode character traits object.
CAtlRegExp Class	This class represents a regular expression.
CAtlREMatchContext Class	This class represents a match context.
CAtlValidator Class	This class provides a static function for determining whether a value lies within a particul ar range.
CBasicAuthObject Class	This class implements support for performing HTTP navigation using the BASIC authentic ation scheme.
CBlobCache Class	This class provides functionality for caching arbitrarily sized data blocks in memory.
CBrowserCapsSvc Class	This class allows you to get access to objects that provide information about the capabiliti es of a particular user agent (Web browser).

CCacheDataBase Class	This is an empty class that is used as the default cache entry type by CMemoryCacheBase .
CCacheDataEx Structure	This structure is used by CMemoryCache and CStencilCache to store auxiliary data for each item in the cache.
CCacheDataPeer Structure	This class is used by CFileCache to provide an interface through which auxiliary data can be attached to each cached file.
CCachePerfMon Class	This class is used in the implementation of CPerfStatClass.
CCacheServerContext Class	This class provides the server context for request handlers that request page-caching support.
CCookie Class	This class represents a cookie to be sent from the server to a client or a cookie that has been returned by a client to the originating server.
CCryptDerivedKey Class	This class represents a cryptographic session key that is derived from a hashed password .
CCryptHash Class	This class represents a generic hash.
CCryptHMACHash Class	This class represents a Hashed Message Authentication Code (HMAC) keyed hash.
CCryptImportKey Class	This class represents a cryptographic key created by importing a key BLOB.
CCryptKey Class	This class represents a generic cryptographic key.
CCryptKeyedHash Class	This class represents a keyed hash, such as a Hashed Message Authentication Code (HMAC) or a Message Authentication Code (MAC) hash.
CCryptMACHash Class	This class represents a Message Authentication Code (MAC) keyed hash.
CCryptMD2Hash Class	This class represents an MD2 hash.
CCryptMD4Hash Class	This class represents an MD4 hash.
CCryptMD5Hash Class	This class represents an MD5 hash.
CCryptProv Class	This class represents a cryptographic service provider (CSP).
CCryptRandomKey Class	This class represents a randomly generated cryptographic session key.
CCryptSHAHash Class	This class represents a Secure Hash Algorithm (SHA) hash.
CCryptSSL3SHAMD5Hash Class	This class represents an SSL3 client authentication hash.
CCryptUserExKey Class	This class represents the user's exchange key pair (used to encrypt and decrypt session keys).
CCryptUserSigKey Class	This class represents the user's signature key pair (used to create and verify digital signatures).

CCullerCacheData Structure	This structure holds information about cache entry that can be used by classes conformin g to the cache culler archetype to determine whether, and in what order, items should be removed from the cache.
CDataSourceCache Class	This class caches open data connections and implements the IDataSourceCache interface.
CDBSession Class	This class template provides an implementation of ISession that stores the session variab les in a database.
CDBSessionServiceImplT Class	This class provides a database-backed session-state implementation for use within ATL S erver ISAPI extension DLLs.
CDebugReportHook Class	Use this class to send debug reports to a named pipe.
CDefaultAuth Class	This class allows you to check whether the user making an HTTP request is in a particular group.
CDefaultErrorProvider Structure	This class provides access to error information that can be displayed to the user or used f or diagnostic messages.
CDefaultQueryClass Class	This class provides access to the SQL queries used for database-backed session state.
CDllCache Class	This class implements a DLL cache.
CDllCacheManager Class	This class implements the XML Web service that allows you to manage the DLL cache in a n ATL Server ISAPI extension.
CDllCacheMgr Class	This class implements the request handler that allows you to manage the DLL cache in an ATL Server ISAPI extension using an HTML interface.
CDllCachePeer Class	This class conforms to the DLL cache peer archetype and is used to cache pointers to ATL Server request handler functions.
CDllMgrObject Class	This class provides methods that can be used to implement IDllCacheMgr by wrapping ca lls to IDllCache.
CExpireCuller Class	This class conforms to the cache culler archetype and allows the automatic removal of ca che items that exceed the absolute expiration time set by users of the cache.
CFileCache Class	This class implements a file cache.
CFixedLifetimeCuller Class	This class conforms to the cache culler archetype and allows the automatic removal of ca che items that have not been accessed in a certain period. The lifetime of items in the cac he cannot be changed at run time. The lifetime is the same for all items.
CFlusherCacheData Structure	This structure holds information about an entry in a cache that can be used by classes co nforming to the cache flusher archetype to determine whether, and in what order, items s hould be removed from the cache.
CHtmlGen Class	This class is used to generate HTML elements in an output stream.
CHtmlGenBase Class	This class is used to generate HTML elements in a text stream.

CHtmlStencil Class	This class is used to parse a server response file and generate a text stream by replacing elements found within the server response file with output generated by an object implementing the ITagReplacer interface.
CHtmlTagReplacer Class	This class acts as the link between CStencil-derived classes and CRequestHandlerT and provides stencil caching.
CHttpMap Class	This class is a simple map class that allows the contents of the map to be shared.
CHttpPtrMap Class	This class represents a map of pointers.
CHttpRequest Class	This class represents an HTTP request received by the server.
CHttpRequestFile Class	This class represents an uploaded file and implements the IHttpFile interface.
CHttpRequestParams Class	This class represents a collection of name-value pairs such as query parameters or form variables.
CHttpResponse Class	This class represents an HTTP response that can be sent back to the client.
CIDServerContext Class	This class provides an implementation of IHttpServerContext for use by a subhandler.
CIncludeServerContext Class	This class provides an implementation of IHttpServerContext for use by included request handlers.
CIsapiExtension Class	This class provides the code necessary for implementing the ISAPI extension DLL that ATL Server uses to pass requests from IIS to your Web application DLLs.
CIsapiWorker Class	Receives ATL Server requests from the thread pool and passes them on to an object implementing IIsapiExtension.
CLifetimeCuller Class	This class conforms to the cache culler archetype and allows the automatic removal of cache items that haven't been accessed in a certain period. The lifetime of an item can be changed at runtime and can be different for each of the items in the cache.
CLOUFlusher Class	This class conforms to the cache flusher archetype and allows the automatic removal of cache items based on their access pattern. Items that are used infrequently are presented for removal before items that are used more often.
CLRUFlusher Class	This class conforms to the cache flusher archetype and allows the automatic removal of cache items based on their access pattern. Items that have not been used for a long time are presented for removal before items that have been used more recently.
CMemoryCache Class	This class represents a generic memory cache.
CMemoryCacheBase Class	This class provides features for caching generic items in memory.
CMemSession Class	This class provides an implementation of ISession that stores the session variables in memory.
CMemSessionServiceImpl Class	This class provides an in-memory based session-state implementation for use within ATL Server ISAPI extension DLLs.
CMimeAttachment Class	This class represents an attachment in a MIME-encoded message.

CMimeBodyPart Class	This class represents a body part of a MIME-encoded message.
CMimeFileAttachment Class	This class represents a file attachment in a MIME-encoded message.
CMimeHeader Class	This class represents a MIME-encoded message header.
CMimeMessage Class	This class represents a MIME-encoded message.
CMimeRawAttachment Class	This class represents an attachment of raw data in a MIME-encoded message.
CMimeText Class	This class represents the body text in a MIME-encoded message.
CMultiPartFormParser Class	Use this class to read multipart/form-data from the associated server context and genera te files as necessary from the data in the body of the request.
CNoDllCachePeer Class	This class conforms to the DLL cache peer archetype and can be used when no extra data needs to be stored with the DLLs in the cache.
CNoExpireCuller Class	This class conforms to the cache culler archetype and can be used to disable automated r emoval of cache items based on expiration times.
CNoFileCachePeer Class	This class conforms to the file cache peer archetype and can be used with CFileCache wh en no auxiliary information needs to be stored with the entries.
CNoFlusher Class	This class conforms to the cache flusher archetype and can be used to disable automated removal of cache items based on access patterns.
CNonStatelessWorker Class	Receives requests from a thread pool and passes them on to a worker object that is creat ed and destroyed on each request.
CNoRequestStats Class	Use this class as the argument for the *CRequestStatClass* template parameter to CIsapiExtension if you do not want to collect request-handling statistics.
CNoStatClass Class	Use this class anywhere that a class conforming to the cache statistics archetype is expect ed if you do not want to collect cache statistics.
CNoWorkerThread Class	Use this class as the argument for the *MonitorClass* template parameter to CBlobCache, CStencilCache, or other cache classes if you want to disable dynamic cache maintenance.
CNTLMAuthObject Class	This class implements support for performing HTTP navigation using the NTLM authentic ation scheme.
COldFlusher Class	This class conforms to the cache flusher archetype and allows the automatic removal of c ache items based on their access pattern. Items are presented for removal in the order in which they were added to the cache.
COrCullers Class	This class combines the functionality of two cache cullers into a single class.
COrFlushers Class	This class combines the functionality of two cache flushers into a single class.
CPageCachePeer Class	This class is used internally by ATL Server to manage auxiliary information for cached res ponses returned by request handlers.
CPathT Class	This class represents a path.

CPerfLock Class	This class manages the lock state of performance monitor objects.
CPerfMon Class	This class represents a performance monitor object manager.
CPerfMonRequestStats Class	Use this class as the argument for the *CRequestStatClass* template parameter to CIsapiExtension if you want to expose request-handling statistics as performance monitor counters.
CPerfObject Structure	This structure represents a performance monitor object.
CPerfRequestStatObject Structure	This structure is used in the implementation of CPerfMonRequestStats.
CPerfStatClass Class	Use this class anywhere that a class conforming to the cache statistics archetype is expected if you want to collect cache statistics in the class's data members and expose the information as performance monitor counters.
CPerfStatObject Structure	This structure is used in the implementation of CStdStatClass and CPerfStatClass.
CReadStreamOnInet Class	This class provides an implementation of IStream that implements Read to return data from a WinInet **HINTERNET** handle returned from a previous call to **InternetOpenUrl**, **FtpOpenFile**, **GopherOpenFile**, or **HttpOpenRequest**.
CReadStreamOnSocket Class	This class provides an implementation of IStream that implements Read to return data from the body of an HTTP response provided by CAtlHttpClientT.
CRequestHandlerT Class	This class acts as the base class for all user request handlers.
CRequestPerfMon Class	This class is used in the implementation of CPerfMonRequestStats.
CRequestStats Class	This structure is used to collect request-handling statistics.
CSDLGenerator Class	This class provides a request handler responsible for generating a Web Service Description Language (WSDL) description of an XML Web service.
CServerContext Class	This class provides an implementation of IHttpServerContext for use within an ISAPI extension DLL or a primary request handler.
CSessionCookie Class	This class represents a cookie that can be used to identify a user's session.
CSessionNameGenerator Class	Use this class to generate random session IDs.
CSessionStateService Class	Provides support for the maintenance and retrieval of session-state data.
CSharedCache Class	This class implements the ISharedBlobCache interface using CBlobCache.
CSharedCacheHandler Class	This class implements a XML Web service that allows access to a cache that can be shared by SOAP clients.
CSimpleStack Class	This class represents a simple stack whose elements are stored in a fixed size array.
CSMTPConnection Class	This class represents a connection to an SMTP service.
CSMTPWSAStartup Class	This class allows you to initialize a WSADATA data structure that contains details of a Windows Sockets implementation.

CSoapFault Class	This class represents a SOAP Fault element, which is used to carry error and/or status information.
CSoapHandler Class	This class provides functionality for handling SOAP requests in an XML Web service request handler. The class is injected as a base class by the soap_handler attribute.
CSoapMSXMLInetClient Class	This class provides client access to XML Web services using the Microsoft XML ServerXMLHTTP object.
CSoapRootHandler Class	This class provides the code necessary to generate and parse SOAP messages. It is used as a base by SPROXY-generated SOAP client classes and XML Web service classes using the soap_handler attribute.
CSoapSocketClientT Class	This class provides client access to XML Web services using sockets.
CSoapWininetClient Class	This class provides client access to XML Web services using the Microsoft Win32 Internet functions (WinInet).
CStdRequestStats Class	Use this class as the argument for the *CRequestStatClass* template parameter to CIsapiExtension if you want to capture request-handling statistics.
CStdStatClass Class	Use this class anywhere that a class conforming to the cache statistics archetype is expected if you want to collect cache statistics in the class's data members.
CStencil Class	This class is used to parse a stencil and generate a text stream by replacing elements found within the stencil by output generated by an object implementing the ITagReplacer interface.
CStencilCache Class	This class implements a stencil cache, and is used by ATL Server to cache pre-processed stencils to speed response processing.
CStencilCacheManager Class	This class implements the XML Web service that allows you to manage the stencil cache in an ATL Server ISAPI extension.
CStencilCacheMgrObject Class	This class provides methods that can be used to implement IStencilCacheMgr by forwarding calls to implementations of IStencilCacheControl and IMemoryCacheStats.
CStencilMgr Class	This class implements the request handler that allows you to manage the stencil cache in an ATL Server ISAPI extension using an HTML interface.
CStencilState Structure	This class provides state information for use when rendering stencils using CStencil and CHtmlStencil.
CStreamFormatter Class	This class writes formatted text to a stream.
CStreamOnServerContext Class	This class provides an implementation of IStream that delegates Read calls to an IHttpServerContext pointer.
CStreamOnWriteStream Class	This class provides an implementation of IStream that delegates calls to an IWriteStream pointer.
CTagReplacerMethodEntry Structure	This structure holds information about a replacement method including its name and pointers to the member functions that implement it.

CTagReplacerMethods Structure	This structure holds pointers to the member functions necessary for implementing a replacement method.
CTagReplacerMethodsEx Structure	This structure provides static functions for casting type-specific replacement methods and parse methods to generic versions that can be stored in the CTagReplacerMethodEntry structure.
CThreadMgrStencil Class	This class implements the request handler that allows you to manage the thread pool in an ATL Server ISAPI extension using an HTML interface.
CThreadPool Class	This class provides a pool of worker threads that process a queue of work items.
CThreadPoolManager Class	This class implements the XML Web service that allows you to manage the thread pool in an ATL Server ISAPI extension.
CThreadPoolMgrObject Class	This class provides methods that can be used to implement IThreadPoolMgr by forwarding calls to an implementation of IThreadPoolConfig.
CTransferServerContext Class	This class provides an implementation of IHttpServerContext for use by request handlers that have had control passed to them by a call to IIsapiExtension::TransferRequest.
CUrl Class	This class represents a URL. It allows you to manipulate each element of the URL independently of the others whether parsing an existing URL string or building a string from scratch.
CValidateContext Class	This class represents a collection of validation status codes. Use this class in combination with CValidateObject to validate forms variables, cookies, or query parameters and to build up a collection of failures (and, optionally, successes).
CValidateObject Class	This class provides methods for retrieving and validating named values.
CVerifyAuth Class	This class allows you to authorize the client making a request to the application.
CWorkerThread Class	This class creates a worker thread or uses an existing one, waits on one or more kernel object handles, and executes a specified client function when one of the handles is signaled.
CWrappedServerContext Class	This class provides an implementation of IHttpServerContext that delegates all the methods to another object implementing the same interface.
CWriteStreamHelper Class	This class provides a convenient way of writing to the IWriteStream interface.
CWriteStreamOnCString Class	This class provides an implementation of IWriteStream that writes the data to a CStringA data member.
DLL_CACHE_ENTRY Structure	This structure describes a cached DLL.
HTML_SCHEME Structure	This class provides HTML formatting information for use with CHtmlGenBase.
IAuthInfo Class	This interface defines methods used to retrieve authentication credentials required for accessing protected URLs.
IBrowserCaps Interface	This interface provides methods for determining the capabilities of a browser.

IBrowserCapsSvc Interface	This interface allows you to get access to objects that provide information about the capabilities of a particular user agent (Web browser).
IDataSourceCache Interface	This interface provides methods for adding database connections to and retrieving them from a cache of open connections.
IDllCache Interface	This interface provides methods for adding DLLs to and retrieving DLLs from a cache.
IDllCacheMgr Interface	This interface provides methods for managing a DLL cache.
IFileCache Interface	This interface provides methods for adding and retrieving files from a cache.
IHttpFile Interface	This interface provides read-only access to information for a file that has been uploaded to the server.
IHttpRequestLookup Interface	This interface provides access to the file, form variable, and query parameter collections associated with an HTTP request.
IHttpServerContext Interface	This interface encapsulates the capabilities of the Web server and provides information about the current request being handled.
IIsapiExtension Interface	This interface provides methods for communicating with the ISAPI extension DLL.
IMemoryCache Interface	This interface provides methods for adding, retrieving, and removing items cached in memory.
IMemoryCacheClient Interface	This interface provides a method that can be called to free an item that is being removed from a cache.
IMemoryCacheControl Interface	This interface provides methods for controlling the size of a cache.
IMemoryCacheStats Interface	This interface provides methods for retrieving current and historical information about the state of a memory-based cache.
IPageCacheControl Interface	This interface provides methods for controlling the caching of a response.
IRequestHandler Interface	This interface provides the methods necessary for a request handler.
IRequestHandlerImpl Class	This class provides a default implementation of the IRequestHandler interface.
IRequestStats Interface	This interface provides methods to get statistics on how requests are being handled.
ISAXContentHandlerImpl Class	This class provides a non-functional implementation of **ISAXContentHandler** that can be used as a base class by classes that need to provide the binary interface of **ISAXContentHandler** but do not need usable implementations for all the methods.
ISession Interface	This interface allows the storage and retrieval of session-specific data.
ISessionStateControl Interface	This interface provides methods for controlling ATL Server session-state settings.
ISessionStateService Interface	This interface provides methods for creating, retrieving, and closing session objects.
ISharedBlobCache Interface	This SOAP-compatible interface provides methods for adding items to and retrieving items from a cache.

IStencilCache Interface	This interface provides methods for adding stencils to and retrieving stencils from a cache.
IStencilCacheControl Interface	This interface provides methods for controlling a stencil cache.
IStencilCacheMgr Interface	This interface provides methods for managing a stencil cache.
IStreamImpl Class	This class provides a non-functional implementation of IStream that can be used as a base class by classes that need to provide the binary interface of **IStream** but do not need usable implementations for all the methods.
ITagReplacer Interface	This interface provides the methods necessary for server response file processing. It provides the connection between the *object.method* text of a replacement tag and the object that implements that method.
ITagReplacerImpl Class	Implementation of the ITagReplacer interface.
IThreadPoolConfig Interface	This interface provides methods for configuring a thread pool.
IThreadPoolMgr Interface	This interface provides methods for managing a thread pool.
IWorkerThreadClient Interface	**IWorkerThreadClient** is the interface implemented by clients of the CWorkerThread class.
IWriteStream Interface	This interface provides methods for writing to and flushing a stream buffer.
StencilToken Structure	Use this structure to store information about a token in CStencil or derived classes.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

ATL_HTML_TAG Structure

This structure associates an HTML tag name with flags describing properties of that tag.

```
struct ATL_HTML_TAG
```

Remarks

This data structure pairs an HTML tag name (such as "div") with some flags indicating whether the tag has an associated end tag or is a block tag. CHtmlGenBase uses an array of ATL_HTML_TAG elements internally.

The **s_tags** array in **atlhtml.h** provides information for the standard HTML tags.

Requirements

Header: atlhtml.h

See Also

Other Resources

ATL_HTML_TAG Members

ATL Server Classes

ATL_HTML_TAG Members

Data Members

szTagName	The name of the HTML element.
uFlags	The flags describing properties of the HTML element, such as whether the tag should have an end tag and whether it's a block element.

See Also

Reference

ATL_HTML_TAG Structure

ATL_HTML_TAG Data Members

For information about the data members in **ATL_HTML_TAG**, see ATL_HTML_TAG Members

ATL_HTML_TAG::szTagName

The name of the HTML element.

```
LPCTSTR szTagName;
```

See Also

Reference

ATL_HTML_TAG Structure

Other Resources

ATL_HTML_TAG Members

ATL_HTML_TAG::uFlags

The flags describing properties of the HTML element, such as whether the tag should have an end tag and whether it's a block element.

```
UINT uFlags;
```

Remarks

uFlags can be a combination of the following values

Name	Description
TAGF_NONE	No special properties.
TAGF_HASEND	The tag requires an end tag.
TAGF_BLOCK	The tag is a block tag.

See Also

Reference

ATL_HTML_TAG Structure

Other Resources

ATL_HTML_TAG Members

ATL_NAVIGATE_DATA Structure

This class represents HTTP client settings as required by CAtlHttpClientT.

```
struct ATL_NAVIGATE_DATA
```

Remarks

HTTP client requests, as launched by CAtlHttpClientT::Navigate, can be tailored by providing a pointer to an **ATL_NAVIGATE_DATA** object.

Note that it is also possible to provide a pointer to a CAtlNavigateData object, because **CAtlNavigateData** derives from **ATL_NAVIGATE_DATA**. CAtlNavigateData provides methods for the assignment and inspection of navigation settings, and a constructor that provides default values for all data members.

Requirements

Header: atlhttp.h

See Also

Concepts

HTTP Client Services

Other Resources

ATL_NAVIGATE_DATA Members

HTTP Client Reference

ATL_NAVIGATE_DATA Members

Data Members

dwDataLen	The size of the optional POST buffer indicated by the pData field.
dwFlags	The behavior modification flags for the HTTP request.
dwReadBlockSize	The size of the buffer to be used for processing HTTP responses.
dwSendBlockSize	Specifies the block size used to send request data when using ATL_HTTP_FLAG_SEND_BLOCKS.
dwTimeout	The network timeout setting, in milliseconds.
m_lParamChunkCB	The value passed to the callback function pointed to by pfnChunkCallback.
m_lParamRead	The value passed to the callback function pointed to by pfnReadStatusCallback.
m_lParamSend	The value passed to the callback function pointed to by pfnSendStatusCallback.
nPort	The port number of the HTTP server.
pData	A pointer to the request body.
pfnChunkCallback	A pointer to a callback function used for chunk-encoded requests.
pfnReadStatusCallback	A pointer to a callback function that receives status updates on arriving responses.
pfnSendStatusCallback	A pointer to a callback function that receives status updates on outgoing requests.
szDataType	Specifies the type of data being sent with the request such as "text/plain" or "text/xml".
szExtraHeaders	A buffer containing additional headers to be sent with the request.
szMethod	The HTTP method to be used for the request, such as "GET" or "POST".

See Also

Reference

ATL_NAVIGATE_DATA Structure

Concepts

HTTP Client Services

Other Resources

HTTP Client Reference

ATL_NAVIGATE_DATA Data Members

For information about the data members in **ATL_NAVIGATE_DATA**, see HTTP Client Services | ATL_NAVIGATE_DATA Members

ATL_NAVIGATE_DATA::dwDataLen

The size of the optional POST buffer indicated by the pData field.

```
DWORD dwDataLen;
```

Remarks

The **dwDataLen** data member indicates the size of an optional POST buffer. If a POST buffer is required, the pData member should point to a buffer containing the POST data; otherwise, both members should be **NULL**.

See Also

Reference

ATL_NAVIGATE_DATA Structure

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

ATL_NAVIGATE_DATA Members

HTTP Client Reference

ATL_NAVIGATE_DATA::dwFlags

The behavior modification flags for the HTTP request.

```
DWORD dwFlags;
```

Remarks

The **dwFlags** member contains flags that control how the HTTP request is processed. See ATL_HTTP Flags for more details.

See Also

Reference

ATL_NAVIGATE_DATA Structure
CAtlNavigateData Class
CAtlNavigateData::SetFlags

Concepts

HTTP Client Services

Other Resources

ATL_NAVIGATE_DATA Members
HTTP Client Reference

ATL_NAVIGATE_DATA::dwReadBlockSize

The size of the buffer to be used for processing HTTP responses.

```
DWORD dwReadBlockSize;
```

Remarks

This data member determines the size of the buffer that is used for processing normal, non-chunk-encoded responses. This member is used only for requests that use to the ATL_HTTP_FLAG_SEND_BLOCKS option.

The ATL_HTTP_DEFAULT_BLOCK_SIZE symbol can be used to provide the value for this data member.

See Also

Reference

ATL_NAVIGATE_DATA Structure
CAtlNavigateData Class
CAtlNavigateData::SetReadBlockSize

Concepts

HTTP Client Services

Other Resources

ATL_NAVIGATE_DATA Members
HTTP Client Reference

ATL_NAVIGATE_DATA::dwSendBlockSize

Specifies the block size used to send request data when using ATL_HTTP_FLAG_SEND_BLOCKS.

```
DWORD dwSendBlockSize;
```

Remarks

The **dwSendBlockSize** data member dictates the size of the buffer that CAtlHttpClientT uses to process requests.

The ATL_HTTP_DEFAULT_BLOCK_SIZE symbol can be used to provide the value for this data member.

See Also

Reference

ATL_NAVIGATE_DATA Structure
CAtlNavigateData Class
CAtlNavigateData::SetSendBlockSize

Concepts

HTTP Client Services

Other Resources

ATL_NAVIGATE_DATA Members
HTTP Client Reference

ATL_NAVIGATE_DATA::dwTimeout

The network timeout setting, in milliseconds.

```
DWORD dwTimeout;
```

Remarks

The CAtlHttpClientT class uses a socket object to send and receive network traffic. The **dwTimeout** data member dictates the time the socket object will wait for a server response before abandoning the operation.

The ATL_SOCK_TIMEOUT symbol can be used to provide a value for this data member.

See Also

Reference

ATL_NAVIGATE_DATA Structure
CAtlNavigateData Class
CAtlNavigateData::SetSocketTimeout

Concepts

HTTP Client Services

Other Resources

ATL_NAVIGATE_DATA Members
HTTP Client Reference

ATL_NAVIGATE_DATA::m_lParamChunkCB

The value passed to the callback function pointed to by pfnChunkCallback.

```
DWORD_PTR m_lParamChunkCB;
```

Remarks

If a callback function is assigned to the pfnChunkCallback member, this value is passed as the third parameter of the callback function when it is called

See Also

Reference

ATL_NAVIGATE_DATA Structure

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

ATL_NAVIGATE_DATA Members

HTTP Client Reference

ATL_NAVIGATE_DATA::m_lParamRead

The value passed to the callback function pointed to by pfnReadStatusCallback.

```
DWORD_PTR m_lParamRead;
```

Remarks

If a read status callback function is assigned to the pfnReadStatusCallback member, this value is passed as the second parameter of the callback function when it is called

See Also

Reference

ATL_NAVIGATE_DATA Structure

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

ATL_NAVIGATE_DATA Members

HTTP Client Reference

ATL_NAVIGATE_DATA::m_lParamSend

The value passed to the callback function pointed to by pfnSendStatusCallback.

```
DWORD_PTR m_lParamSend;
```

Remarks

If a send status callback function is assigned to the pfnSendStatusCallback member, this value is passed as the second parameter of the callback function when it is called

See Also

Reference

ATL_NAVIGATE_DATA Structure

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

ATL_NAVIGATE_DATA Members

HTTP Client Reference

ATL_NAVIGATE_DATA::nPort

The port number of the HTTP server.

```
short nPort;
```

Remarks

The **nPort** field stores the port number of the target HTTP server.

The ATL_URL_DEFAULT_HTTP_PORT symbol can be used to provide a value for this data member.

See Also

Reference

ATL_NAVIGATE_DATA Structure
CAtlNavigateData Class
CAtlNavigateData::SetPort

Concepts

HTTP Client Services

Other Resources

ATL_NAVIGATE_DATA Members
HTTP Client Reference

ATL_NAVIGATE_DATA::pData

A pointer to the request body.

```
BYTE* pData;
```

Remarks

The **pData** member points to an optional buffer containing a request body. The length of this buffer should be stored in the dwDataLen member. If no buffer is required, this member should be **NULL**.

See Also

Reference

ATL_NAVIGATE_DATA Structure

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

ATL_NAVIGATE_DATA Members

HTTP Client Reference

ATL_NAVIGATE_DATA::pfnChunkCallback

A pointer to a callback function used for chunk-encoded requests.

```
PFNATLCHUNKEDCB pfnChunkCallback;
```

Remarks

If chunk-encoded requests are to be used, the **pfnChunkCallback** member should be made to point to a callback function responsible for sending chunk-encoded requests. Chunk-encoded requests are launched using the CAtlHttpClientT::NavigateChunked method. If no callback is required, this member should be **NULL**.

See Also

Reference

ATL_NAVIGATE_DATA Structure
CAtlNavigateData Class
CAtlNavigateData::SetChunkCallback
PFNATLCHUNKEDCB

Concepts

HTTP Client Services

Other Resources

ATL_NAVIGATE_DATA Members
HTTP Client Reference

ATL_NAVIGATE_DATA::pfnReadStatusCallback

A pointer to a callback function that receives status updates on arriving responses.

```
PFNATLSTATUSCALLBACK pfnReadStatusCallback;
```

Remarks

The **pfnReadStatusCallback** member can optionally be assigned to point to a callback function that CAtlHttpClientT will call when HTTP response data is received. If no callback is required, this member should be **NULL**.

See Also

Reference

ATL_NAVIGATE_DATA Structure
CAtlNavigateData Class
CAtlNavigateData::SetReadStatusCallback
PFNATLSTATUSCALLBACK

Concepts

HTTP Client Services

Other Resources

ATL_NAVIGATE_DATA Members
HTTP Client Reference

ATL_NAVIGATE_DATA::pfnSendStatusCallback

A pointer to a callback function that receives status updates on outgoing requests.

```
PFNATLSTATUSCALLBACK pfnSendStatusCallback;
```

Remarks

The **pfnSendStatusCallback** member can optionally be assigned to point to a callback function that CAtlHttpClientT will call after HTTP request data has successfully been sent. If no callback is required, this member should be **NULL**.

See Also

Reference

ATL_NAVIGATE_DATA Structure
CAtlNavigateData Class
CAtlNavigateData::SetSendStatusCallback
PFNATLSTATUSCALLBACK

Concepts

HTTP Client Services

Other Resources

ATL_NAVIGATE_DATA Members
HTTP Client Reference

ATL_NAVIGATE_DATA::szDataType

Specifies the type of data being sent with the request such as "text/plain" or "text/xml".

```
LPCTSTR szDataType;
```

Remarks

The **szDataType** member stores a pointer to a string describing the data contained in the request body as pointed to by the pData member. This string is used in the HTTP request headers as the "Content-Type" header value. If no request body is to be sent, this member should be **NULL**.

See Also

Reference

ATL_NAVIGATE_DATA Structure
CAtlNavigateData Class
CAtlNavigateData::SetPostData

Concepts

HTTP Client Services

Other Resources

ATL_NAVIGATE_DATA Members
HTTP Client Reference

ATL_NAVIGATE_DATA::szExtraHeaders

A buffer containing additional headers to be sent with the request.

```
LPCTSTR szExtraHeaders;
```

Remarks

Auxiliary header data can be included in the HTTP request by assigning the **szExtraHeaders** member to point to a buffer. If no auxiliary headers are required, this member should be **NULL**.

See Also

Reference

ATL_NAVIGATE_DATA Structure
CAtlNavigateData Class
CAtlNavigateData::SetExtraHeaders

Concepts

HTTP Client Services

Other Resources

ATL_NAVIGATE_DATA Members
HTTP Client Reference

ATL_NAVIGATE_DATA::szMethod

The HTTP method to be used for the request, such as "GET" or "POST".

```
LPCTSTR szMethod;
```

Remarks

The **szMethod** member should point to a string containing an HTTP method. The most common type of HTTP method is "GET".

See Also

Reference

ATL_NAVIGATE_DATA Structure
CAtlNavigateData Class
CAtlNavigateData::SetMethod

Concepts

HTTP Client Services

Other Resources

ATL_NAVIGATE_DATA Members
HTTP Client Reference

AtlHtmlAttrs Class

Use this class to build up strings of HTML attributes.

```
class AtlHtmlAttrs
```

Remarks

This is a special-purpose class that maintains a string of name-value pairs representing HTML attributes (for example, `bgcolor="#996600"`). This class is used internally by CHtmlGenBase to collect attributes for HTML elements before writing them to the HTML stream. You can also use this class directly.

Note that this class ensures that the string it builds always starts with a space.

Requirements

Header: atlhtml.h

See Also

Other Resources

AtlHtmlAttrs Members

ATL Server Classes

AtlHtmlAttrs Members

Methods

Add	Call this method to add an HTML attribute to the string.
AddFormat	Call this method to format an entry and add it to the HTML attributes.
AtlHtmlAttrs	The constructor.
Set	Call this method to set the string managed by this object.

Operators

operator LPCTSTR	This operator is used to cast an **AtlHtmlAttrs** object to an **LPCTSTR** data type.

Data Members

m_strAttrs	The string of HTML attributes.

See Also

Reference

AtlHtmlAttrs Class

AtlHtmlAttrs Methods

For information about the methods in **AtlHtmlAttrs**, see AtlHtmlAttrs Members.

AtlHtmlAttrs::Add

Call this method to add an HTML attribute to the string.

```
BOOL Add(
   LPCTSTR szName,
   LPCTSTR szValue
);
```

Parameters

szName

The name of the attribute.

szValue

The value of the attribute.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

An entry of the form *szName*="*szValue*" is appended to the string of HTML attributes. If *szValue* is NULL, only *szName* is added to the string. This method ensures that a space is added to the string before the new attribute. No checking is done to ensure that duplicate values cannot be added.

See Also

Reference

AtlHtmlAttrs Class

AtlHtmlAttrs::AddFormat

Other Resources

AtlHtmlAttrs Members

AtlHtmlAttrs::AddFormat

Call this method to format an entry and add it to the HTML attributes.

```
void AddFormat(
   LPCTSTR szFormat,
   ...
);
```

Parameters

szFormat

The format-control string.

Remarks

This method takes a variable number of parameters after *szFormat*, formats them according to *szFormat*, and appends the result to the string of HTML attributes. This method ensures that a space is added to the string before the formatted string.

For more information, see Format Specification Fields: printf and wprintf Functions.

See Also

Reference

AtlHtmlAttrs Class

AtlHtmlAttrs::Add

Other Resources

AtlHtmlAttrs Members

AtlHtmlAttrs::AtlHtmlAttrs

The constructor.

```
AtlHtmlAttrs( );
AtlHtmlAttrs(
   LPCTSTR szFormat,
   ...
);
AtlHtmlAttrs(
   int nCount,
   ...
);
```

Parameters

szFormat

The format-control string.

nCount

The number of attribute-value pairs passed as arguments.

Remarks

The first form of the constructor constructs an **AtlHtmlAttrs** object with no HTML attributes.

The second form takes a variable number of parameters after *szFormat* and formats them according to *szFormat* to produce the string of HTML attributes. This constructor is equivalent to AtlHtmlAttrs::AddFormat.

The third form takes *nCount* pairs of LPCTSTR parameters and adds them as name-value pairs to the string of HTML attributes.

See Also

Reference

AtlHtmlAttrs Class

Other Resources

AtlHtmlAttrs Members

AtlHtmlAttrs::Set

Call this method to set the string managed by this object.

```
void Set(
   LPCTSTR szAttrs
);
```

Parameters

szAttrs

The string of HTML attributes.

Remarks

This method sets the internal HTML attributes string to *szAttrs* and ensures that it starts with a space.

See Also

Reference

AtlHtmlAttrs Class

AtlHtmlAttrs::m_strAttrs

Other Resources

AtlHtmlAttrs Members

AtlHtmlAttrs Operators

For information about the operators in **AtlHtmlAttrs**, see AtlHtmlAttrs Members.

AtlHtmlAttrs::operator LPCTSTR

This operator is used to cast an **AtlHtmlAttrs** object to an **LPCTSTR** data type.

```
operator LPCTSTR( );
```

Return Value

Returns the HTML attributes string.

See Also

Reference

AtlHtmlAttrs Class

AtlHtmlAttrs::m_strAttrs

Other Resources

AtlHtmlAttrs Members

AtlHtmlAttrs Data Members

For information about the data members in **AtlHtmlAttrs**, see AtlHtmlAttrs Members.

AtlHtmlAttrs::m_strAttrs

The string of HTML attributes.

```
CString m_strAttrs;
```

Remarks

This string accumulates the HTML attribute name-value pairs.

See Also

Reference

AtlHtmlAttrs Class

AtlHtmlAttrs::operator LPCTSTR

Other Resources

AtlHtmlAttrs Members

ATLServerDllInfo Structure

This class holds pointers to the three functions exported by an ATL Server request handler DLL along with pointers to the ISAPI extension and server context.

```
struct ATLServerDllInfo
```

Requirements

Header: atlisapi.h

See Also

Other Resources

ATLServerDllInfo Members

ATL Server Classes

ATLServerDllInfo Members

Data Members

pContext	The pointer to the server context used by this DLL.
pExtension	The pointer to the ISAPI extension used by this DLL.
pfnGetHandler	The pointer to the function used to get an interface pointer on a request handler given its name.
pfnInitHandlers	The pointer to the function used to initialize the request handlers in a DLL.
pfnUninitHandlers	The pointer to the function used to uninitialize the request handlers in a DLL.

See Also

Reference

ATLServerDllInfo Structure

ATLServerDllInfo Data Members

For information about the data members in **ATLServerDllInfo**, see ATLServerDllInfo Members.

ATLServerDllInfo::pContext

The pointer to the server context used by this DLL.

```
IHttpServerContext * pContext;
```

See Also

Reference

ATLServerDllInfo Structure

IHttpServerContext Interface

Other Resources

ATLServerDllInfo Members

ATLServerDllInfo::pExtension

The pointer to the ISAPI extension used by this DLL.

```
IIsapiExtension * pExtension;
```

See Also

Reference

ATLServerDllInfo Structure

IIsapiExtension Interface

Other Resources

ATLServerDllInfo Members

ATLServerDllInfo::pfnGetHandler

The pointer to the function used to get an interface pointer on a request handler given its name.

```
GETATLHANDLERBYNAME pfnGetHandler;
```

See Also

Reference

ATLServerDllInfo Structure

GETATLHANDLERBYNAME

Other Resources

ATLServerDllInfo Members

ATLServerDllInfo::pfnInitHandlers

The pointer to the function used to initialize the request handlers in a DLL.

```
INITIALIZEATLHANDLERS pfnInitHandlers;
```

See Also

Reference

ATLServerDllInfo Structure

INITIALIZEATLHANDLERS

Other Resources

ATLServerDllInfo Members

ATLServerDllInfo::pfnUninitHandlers

The pointer to the function used to uninitialize the request handlers in a DLL.

```
UNINITIALIZEATLHANDLERS pfnUninitHandlers;
```

See Also

Reference

ATLServerDllInfo Structure

UNINITIALIZEATLHANDLERS

Other Resources

ATLServerDllInfo Members

AtlServerRequest Structure

This structure represents an HTTP request in the ATL Server infrastructure.

```
struct AtlServerRequest
```

Remarks

This structure contains information that is used by the ATL Server request handling infrastructure. The information contained in this structure may be of interest to advanced users for debugging or other purposes. Most users, however, need not deal with this structure directly.

Requirements

Header: atlisapi.h

Example

See the CLStencil Sample and the PooledAsync Sample.

See Also

Other Resources

AtlServerRequest Members

ATL Server Classes

AtlServerRequest Members

Data Members

cbSize	The size of the structure in bytes.
dwBufferLen	The length in bytes of the buffer AtlServerRequest::pszBuffer.
dwRequestState	The state of the request.
dwRequestType	The type of the request.
dwStartTicks	The tick count when the request was received.
hEntry	The handle of an entry in the file cache.
hFile	The handle of the cached file being used to service the current request.
hInstDll	The HINSTANCE of the request handler DLL.
m_hMutex	The mutex used to synchronize calls to IRequestHandler::HandleRequest.
pDllCache	The interface pointer to the cache holding the request handler DLL.
pECB	The EXTENSION_CONTROL_BLOCK representing the current request.
pExtension	The pointer to the interface on the ISAPI extension DLL handling this request.
pFileCache	The interface pointer to the cache holding the generated response.
pfnAsyncComplete	The pointer to the function to be called on completion of asynchronous operations.
pfnHandleRequest	The pointer to the function responsible for handling this request.
pHandler	The interface pointer to the request handler responsible for this request.
pServerContext	The server context for the current request.
pszBuffer	The buffer containing data to be written to the client asynchronously.
pUserData	The pointer that can be used to pass data between different request handlers handling the same request.

See Also

Reference

AtlServerRequest Structure

AtlServerRequest Data Members

For information about the data members in **AtlServerRequest**, see AtlServerRequest Members.

AtlServerRequest::cbSize

The size of the structure in bytes.

```
DWORD cbSize;
```

Remarks

This member is present to allow for future compatibility and versioning of this structure.

See Also

Reference

AtlServerRequest Structure

Other Resources

AtlServerRequest Members

AtlServerRequest::dwBufferLen

The length in bytes of the buffer AtlServerRequest::pszBuffer.

```
DWORD dwBufferLen;
```

See Also

Reference

AtlServerRequest Structure

AtlServerRequest::pszBuffer

Other Resources

AtlServerRequest Members

AtlServerRequest::dwRequestState

The state of the request.

```
ATLSRV_STATE dwRequestState;
```

Remarks

See ATLSRV_STATE for a description of the possible values.

Example

See the PooledAsync Sample.

See Also

Reference

AtlServerRequest Structure

ATLSRV_STATE

Other Resources

AtlServerRequest Members

AtlServerRequest::dwRequestType

The type of the request.

```
ATLSRV_REQUESTTYPE dwRequestType;
```

Remarks

Indicates whether the request was for a server response file or for a request handler DLL. See ATLSRV_REQUESTTYPE for a description of the possible values.

See Also

Reference

AtlServerRequest Structure

ATLSRV_REQUESTTYPE

Other Resources

AtlServerRequest Members

AtlServerRequest::dwStartTicks

The tick count when the request was received.

```
DWORD dwStartTicks;
```

Remarks

The tick count is obtained using timeGetTime if ATL_NO_MMSYS is not defined or GetTickCount otherwise. The tick count is used to calculate average response time statistics.

See Also

Reference

AtlServerRequest Structure

Other Resources

AtlServerRequest Members

AtlServerRequest::hEntry

The handle of an entry in the file cache.

```
HCACHEITEM hEntry;
```

Remarks

This member keeps track of the file cache entry if the request is being fulfilled by an entry in the cache. This essentially represents a lock on the cache entry to ensure that the cached file will not be deleted if it is still being used to service the current request.

See Also

Reference

AtlServerRequest Structure

HCACHEITEM

Other Resources

AtlServerRequest Members

AtlServerRequest::hFile

The handle of the cached file being used to service the current request.

```
HANDLE hFile;
```

See Also

Reference

AtlServerRequest Structure

Other Resources

AtlServerRequest Members

AtlServerRequest::hInstDll

The HINSTANCE of the request handler DLL.

```
HINSTANCE hInstDll;
```

Remarks

This member is used to ensure that the request handler DLL is only released when the request has completed.

See Also

Reference

AtlServerRequest Structure

Other Resources

AtlServerRequest Members

AtlServerRequest::m_hMutex

The mutex used to synchronize calls to IRequestHandler::HandleRequest.

```
HANDLE m_hMutex;
```

Remarks

This member is only used if IRequestHandler::GetFlags includes ATLSRV_INIT_USEASYNC_EX in its list of flags.

See Also

Reference

AtlServerRequest Structure

Other Resources

AtlServerRequest Members

AtlServerRequest::pDllCache

The interface pointer to the cache holding the request handler DLL.

```
IDllCache* pDllCache;
```

See Also

Reference

AtlServerRequest Structure

IDllCache Interface

Other Resources

AtlServerRequest Members

AtlServerRequest::pECB

The **EXTENSION_CONTROL_BLOCK** representing the current request.

```
EXTENSION_CONTROL_BLOCK* pECB;
```

See Also

Reference

AtlServerRequest Structure

Other Resources

AtlServerRequest Members

AtlServerRequest::pExtension

The pointer to the interface on the ISAPI extension DLL handling this request.

```
IIsapiExtension* pExtension;
```

See Also

Reference

AtlServerRequest Structure

IIsapiExtension Interface

Other Resources

AtlServerRequest Members

AtlServerRequest::pFileCache

The interface pointer to the cache holding the generated response.

```
IFileCache* pFileCache;
```

See Also

Reference

AtlServerRequest Structure

IFileCache Interface

Other Resources

AtlServerRequest Members

AtlServerRequest::pfnAsyncComplete

The pointer to the function to be called on completion of asynchronous operations.

```
PFnAsyncComplete pfnAsyncComplete;
```

See Also

Reference

AtlServerRequest Structure

PFnAsyncComplete

Other Resources

AtlServerRequest Members

AtlServerRequest::pfnHandleRequest

The pointer to the function responsible for handling this request.

```
PFnHandleRequest pfnHandleRequest;
```

See Also

Reference

AtlServerRequest Structure

PFnHandleRequest

Other Resources

AtlServerRequest Members

AtlServerRequest::pHandler

The interface pointer to the request handler responsible for this request.

```
IRequestHandler* pHandler;
```

See Also

Reference

AtlServerRequest Structure

IRequestHandler Interface

Other Resources

AtlServerRequest Members

AtlServerRequest::pServerContext

The server context for the current request.

```
IHttpServerContext* pServerContext;
```

See Also

Reference

AtlServerRequest Structure

IHttpServerContext Interface

Other Resources

AtlServerRequest Members

AtlServerRequest::pszBuffer

The buffer containing data to be written to the client asynchronously.

```
LPCSTR pszBuffer;
```

See Also

Reference

AtlServerRequest Structure

AtlServerRequest::dwBufferLen

Other Resources

AtlServerRequest Members

AtlServerRequest::pUserData

The pointer that can be used to pass data between different request handlers handling the same request.

```
void* pUserData;
```

See Also

Reference

AtlServerRequest Structure

Other Resources

AtlServerRequest Members

ATLSOAP_BLOB Structure

This class represents a binary large object (BLOB) for use with SOAP.

```
[ export ]
typedef struct _tagATLSOAP_BLOB{
   unsigned long size;
   unsigned char* data;
} ATLSOAP_BLOB;
```

Remarks

Use this structure when you want to send binary data through SOAP.

The advantage of using **ATLSOAP_BLOB** instead of a byte array is that the ATL Server SOAP attributes recognize this type and know to represent it using XML Schema's base64Binary type. This allows the attributes to minimize the size of the SOAP packet and the time taken to parse it.

It is important to remember that C++ data types are converted to XML Schema representations that are then passed in a SOAP message to the recipient. A C++ byte array results in the creation of an XML representation in which each element of the array is represented as a decimal number and separated from the next by start and end tags describing the element in accordance with the XML Schema and SOAP specifications. Conversely, **ATLSOAP_BLOB** data is encoded as a single **base64Binary** element. The data is a single base64-encoded chunk that does not have the overhead of separating each byte from the next or representing each byte as a decimal integer.

The size of binary data represented as an XML array type is many times the size of the original data. The size of binary data represented as an XML Schema **base64Binary** element is only on the order of 1.33 times the size of the original data.

SPROXY uses the **ATLSOAP_BLOB** data type in the SOAP client classes that it generates whenever it sees the XML Schema **base64Binary** or **hexBinary** data types.

Requirements

Header: atlsoap.h

Example

See the SOAPDataTypes Sample.

See Also

Other Resources

ATLSOAP_BLOB Members

ATL Server Classes

ATLSOAP_BLOB Members

Data Members

data	An array of bytes containing binary data.
size	The size of the data in bytes.

See Also

Reference

ATLSOAP_BLOB Structure

ATLSOAP_BLOB Data Members

For information about the data members in **ATLSOAP_BLOB**, see ATLSOAP_BLOB Members

ATLSOAP_BLOB::data

An array of bytes containing binary data.

```
unsigned char* data;
```

Remarks

Use this member for storing the binary data. The size of the data is stored in ATLSOAP_BLOB::size.

See Also

Reference

ATLSOAP_BLOB Structure

Other Resources

ATLSOAP_BLOB Members

ATLSOAP_BLOB::size

The size of the data in bytes.

```
unsigned long size;
```

Remarks

The data is stored in the ATLSOAP_BLOB::data member.

See Also

Reference

ATLSOAP_BLOB Structure

Other Resources

ATLSOAP_BLOB Members

CategoryInfo Structure

This structure represents the performance monitor category information.

```
struct CategoryInfo{
   DWORD m_dwCategoryId;
   CString m_strName;
   CString m_strHelp;
   DWORD m_dwDetailLevel;
   LONG m_nDefaultCounter;
   LONG m_nInstanceLess;
   ULONG m_nStructSize;
   ULONG m_nMaxInstanceNameLen;
   ULONG m_nAllocSize;
   PERF_OBJECT_TYPE m_cache;
   ULONG m_nCounterBlockSize;
   CAtlArray<CounterInfo> m_counters;
}
```

Parameters

m_dwCategoryId

Numeric ID that uniquely represents each category.

m_strName

The name that is displayed for this object within the Performance Monitoring Console. Can be provided as a string or a resource ID.

m_strHelp

A help string that is displayed for this object within the Performance Monitoring Console. Can be provided as a string or a string resource ID.

m_dwDetailLevel

Indicates the level of detail of information provided.

m_nDefaultCounter

A zero-based index of the counter that should be selected by default when viewing objects of this type, Or -1 if there is no default counter.

m_nInstanceLess

Zero if the performance object allows instance creation, nonzero otherwise.

m_nStructSize

Size of the object class.

m_nMaxInstanceNameLen

The maximum size of instance name string.

m_nAllocSize

Allocation size.

m_cache

This structure describes object-specific performance information. This structure is followed by a list of **PERF_COUNTER_DEFINITION** structures, one for each counter defined for the type of object.

m_nCounterBlockSize

Object's counters block size.

m_counters

Array of **CounterInfo** structures, each structure contains information about category counter.

Remarks

Structure which contains a performance monitor object's data and the counters related to it.

Requirements

Header: atlperf.h

See Also

Concepts

Performance Monitoring

Other Resources

ATL Server Classes

Performance Monitoring Reference

CAtlBaseAuthObject Class

This is an abstract base class that defines the interface for HTTP client authorization objects.

```
class CAtlBaseAuthObject
```

Remarks

The **CAtlBaseAuthObject** class is used as a base class for HTTP client authorization objects that can be passed to the CAtlHttpClientT class for use in negotiating access to protected URLs. For example, the ATL Server classes CBasicAuthObject and CNTLMAuthObject, which implement support for the BASIC and NTLM authorization schemes, respectively, are both derived from the **CAtlBaseAuthObject** class.

To provide custom support for HTTP authorization schemes for use with the CAtlHttpClientT class, derive from the **CAtlBaseAuthObject** class, and override the **Init** and **Authenticate** methods.

Requirements

Header: atlhttp.h

See Also

Concepts

HTTP Client Services

Other Resources

CAtlBaseAuthObject Members

ATL Server Reference

HTTP Client Reference

CAtlBaseAuthObject Members

Methods

Authenticate	This method is called to perform the steps necessary to satisfy security requirements are specific to the au thorization scheme implemented by the derived class.
CAtlBaseAuthObject	The constructor.
Init	This method is called to initialize the authorization object.

Data Members

m_bFailed	Indicates whether this authentication object has failed in a previous attempt to satisfy security requirements.

See Also

Reference

CAtlBaseAuthObject Class

Concepts

HTTP Client Services

Other Resources

HTTP Client Reference

CAtlBaseAuthObject Methods

For information about the methods in **CAtlBaseAuthObject**, see CAtlBaseAuthObject Members

CAtlBaseAuthObject::CAtlBaseAuthObject

The constructor.

```
CAtlBaseAuthObject( );
```

Remarks

The CAtlBaseAuthObject constructor initializes the CAtlBaseAuthObject::m_bFailed data member to **false**.

See Also

Reference

CAtlBaseAuthObject Class
CBasicAuthObject Class
CNTLMAuthObject Class

Concepts

HTTP Client Services

Other Resources

CAtlBaseAuthObject Members
HTTP Client Reference

CAtlBaseAuthObject::Authenticate

This method is called to perform the steps necessary to satisfy security requirements that are specific to the authorization scheme implemented by the derived class.

```
virtual bool Authenticate(
   LPCTSTR szAuthTypes,
   bool bProxy
) = 0;
```

Parameters

szAuthTypes

A string describing the authentication scheme encountered, such as "BASIC", or "NTLM".

bProxy

true if a if proxy server is being use to access the current URL, otherwise **false**.

Return Value

Returns **true** on success, **false** on failure.

Remarks

This method must be overridden in CAtlBaseAuthObject-derived classes. The CAtlHttpClientT class calls **Authenticate** if the string used to attach this object matches one of the authentication schemes supported by the HTTP server. The **Authenticate** method is then expected to attempt to satisfy the security requirements and return **true** if successful or **false** if the authentication attempt fails.

See Also

Reference

CAtlBaseAuthObject Class
CBasicAuthObject Class
CNTLMAuthObject Class

Concepts

HTTP Client Services

Other Resources

CAtlBaseAuthObject Members
HTTP Client Reference

CAtlBaseAuthObject::Init

This method is called to initialize the authorization object.

```
virtual void Init(
   CAtlHttpClient* pSocket,
   IAuthInfo* pAuthInfo
) = 0;
```

Parameters

pSocket

A pointer to the CAtlHttpClient object to which this object has been attached and through which any necessary authorization transactions should occur.

pAuthInfo

A pointer to the IAuthInfo Class interface that can be used to acquire necessary authorization credentials such as user name and password.

Remarks

The CAtlHttpClientT class calls the **Init** method when the authorization object is added using the CAtlHttpClientT::AddAuthObj method, not when the authorization scheme is detected or when the object is used to attempt security navigation. The **Init** method is called regardless of whether the authorization scheme it implements is encountered or required, whereas CAtlBaseAuthObject::Authenticate is called only if the object is to attempt authorization.

See Also

Reference

CAtlBaseAuthObject Class
CBasicAuthObject Class
CNTLMAuthObject Class

Concepts

HTTP Client Services

Other Resources

CAtlBaseAuthObject Members
HTTP Client Reference

CAtlBaseAuthObject Data Members

For information about the data members in **CAtlBaseAuthObject**, see CAtlBaseAuthObject Members

CAtlBaseAuthObject::m_bFailed

Indicates whether this authentication object has failed in a previous attempt to satisfy security requirements.

```
bool m_bFailed;
```

Remarks

The CAtlHttpClientT Class class uses the **m_bFailed** data member to avoid using authentication objects that are known to have failed during previous attempts. The **m_bFailed** member defaults to **false** but is set to **true** automatically if the object fails.

If the same authorization objects are to be used for multiple HTTP requests, each of which should attempt all authorization methods available, this default behavior can be modified by overriding CAtlHttpClientT::NegotiateAuth or by manually resetting the **m_bFailed** data member for each authorization object between HTTP requests.

See Also

Reference

CAtlBaseAuthObject Class
CBasicAuthObject Class
CNTLMAuthObject Class

Concepts

HTTP Client Services

Other Resources

CAtlBaseAuthObject Members
HTTP Client Reference

CAtlHttpClientT Class

This class provides support for making requests and receiving responses from HTTP servers.

```
template <
   class TSocketClass
>
class CAtlHttpClientT :
   private TSocketClass
```

Parameters

TSocketClass

The class used to provide the underlying network socket support. For the current version of the library, this parameter should be set to the internal class, **ZEvtSyncSocket**. The CAtlHttpClient typedef is provided so that you do not have to specify the socket support class explicitly.

Remarks

CAtlHttpClientT supports HTTP client functionality. When using this class, requests can be sent to and responses can be received from HTTP servers. Other networking protocols, such as FTP and NNTP, are not supported. For this release, ATL Server does not support the HTTPS protocol either, but the SecureSoap Sample demonstrates how **CAtlHttpClientT** can be used with a class that provides HTTPS support.

CAtlHttpClient performs network operations synchronously. The Navigate method, which is used to make HTTP requests, does not return until the server response has been received in its entirety, or the request has failed. As a result, simultaneous HTTP requests require multiple **CAtlHttpClient** object instances.

A breaking change from previous versions, **CAtlHttpClient** no longer passes user log on credentials for authentication unless the security zone settings allow it. This change increases the security of user credentials. This may break applications that relied on the non-secure method of authentication. To restore the old behavior, call **SetSilentLogonOk** with the parameter value **true**.

Requirements

Header: atlhttp.h

See Also

Tasks

HttpClient Sample: HTTP Client Services Demonstration

HttpPing Sample: HTTP Client Tool

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT Members

Methods

AddAuthObj	Call this method to append authorization objects to an internal list used for negotiating protected server requests.
AuthProtocolFailed	Call this method to indicate that that an authorization protocol should not be used during subsequent authorization attempts.
CAtlHttpClientT	Constructs a **CAtlHttpClientT** object.
Close	Call this method to close the network connection.
FindAuthObject	Call this method to obtain access to the authorization object for a given security scheme, if present.
GetBody	Call this method to retrieve access to the HTTP response body data.
GetBodyLength	Call this method to determine the size of the response data, not including header information.
GetCurrentUrl	Call this method to gain access to the URL being used for requests by this object.
GetPostData	Call this method to gain access to outgoing request data buffer.
GetPostDataLen	Call this method to determine the size of the outgoing request data buffer.
GetPostDataType	Call this method to retrieve the type of post data associated with this object.
GetFlags	Call this method to retrieve a copy of the behavior control flags.
GetHeaderValue	Call this method to retrieve HTTP message header values.
GetLastError	Call this method to retrieve the last error code reported by the underlying network socket.
GetMethod	Call this method to retrieve the HTTP method being used.
GetProxy	Call this method to get the proxy server settings for this object.
GetProxyPort	Call this method to retrieve the proxy server port setting for this object.
GetRawResponseHeaderLength	Call this method to retrieve the size, in bytes, of the last response's header information.
GetRawResponseHeader	Call this method to retrieve a copy of the last HTTP response headers received.
GetResponse	Call this method to obtain access to the buffer that holds the entire response of the last request processed.
GetResponseLength	Call this method to retrieve the length, in bytes, of the buffer returned by GetResponse.
GetSocket	Call this method to gain access to the underlying network socket.
GetSocketTimeout	Call this method to retrieve the current socket timeout setting.

GetStatus	Call this method to retrieve the HTTP status code for the previous request.
GetResponseStatus	Call this method to retrieve the current status of the response processing mechanisms.
Navigate	Call this method to perform an HTTP request.
NavigateChunked	Call this method to perform a chunked transfer encoded HTTP request.
NegotiateAuth	Seeks authorization to secured URLs using any security protocols supported by the provided a uthorization objects.
RemoveAuthObject	Call this method to remove support for a given authorization scheme.
RemoveProxy	Call this method to remove proxy server settings from this object.
SetProxy	Call this method to assign a proxy server to be used for subsequent HTTP transactions.
SetSocketTimeout	Call this method to set the timeout value for the underlying network socket.

Enums

HTTP_RESPONSE_READ_STATUS	Constants that indicate the status of an attempt to read the HTTP response.

See Also

Concepts

HTTP Client Services

Other Resources

HTTP Client Reference

CAtlHttpClientT Methods

For information about the methods in **CAtlHttpClientT**, see CAtlHttpClientT Members

CAtlHttpClientT::AddAuthObj

Call this method to append authorization objects to an internal list used for negotiating protected server requests.

```
bool AddAuthObj(
   LPCTSTR szScheme,
   CAtlBaseAuthObject* pObject,
   IAuthInfo* pInfo = NULL
) throw( );
```

Parameters

szScheme

A string containing a description of the authorization scheme supported by the object pointed to by *pObject*, such as "BASIC" or "NTLM".

pObject

A pointer to the authorization object.

PInfo

An optional pointer to an IAuthInfo-derived class, which can be used to supply the user name, password, and domain name required for authorization.

Return Value

Returns **true** on success, **false** on failure.

Remarks

CAltHttpClientT uses the authorization objects added using this method when a 401 (Not Authorized) or 407 (Proxy Authorization Required) response is received as the result of a request. By default, authorization objects are used in the order that they were added, but CAtlHttpClientT::NegotiateAuth can be overridden to modify the way authorization negotiation occurs.

Custom authorization classes can be implemented by deriving from the CAtlBaseAuthObject class, but ATL Server provides authorization implementation classes for two popular security schemes: BASIC and NT LAN Manager (NTLM). See the CBasicAuthObject and CNTLMAuthObject classes, respectively, for more information.

Example

See the HttpClient Sample.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::AuthProtocolFailed

Call this method to indicate that that an authorization protocol should not be used during subsequent authorization attempts.

```
void AuthProtocolFailed(
   LPCTSTR szProto
) throw( );
```

Parameters

szProto

A string containing a description of the failed authorization protocol, such as "BASIC" or "NTLM".

Remarks

This method is typically called from an overridden version of the CAtlBaseAuthObject::Authenticate method, after an authentication attempt has failed. Calling **AuthProtocolFailed** marks the protocol indicated by *szProto* as having failed, preventing further use of the given security scheme, even if a request from a different URL is later requested.

To ensure that all authorization objects added to a **CAtlHttpClientT** object are used for a request, AddAuthObj must be called before the request for each desired authorization object.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::CAtlHttpClientT

Constructs a **CAtlHttpClientT** object.

```
CAtlHttpClientT( ) throw( );
```

Remarks

Constructs an instance of the **CAtlHttpClientT** class. Each instance performs HTTP transactions synchronously, so the Navigate method, which makes HTTP requests, does not return until the request has either failed or a full response has been received. Therefore, concurrent HTTP transactions require multiple instances of the **CAtlHttpClientT** class. One class instance can be used to perform multiple HTTP requests, but only serially. In practice, however, it is usually simpler to create a new instance for each transaction.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::Close

Call this method to close the network connection.

```
void Close( ) throw( );
```

Remarks

Terminates the connection. The effect of calling **Close** depends on the underlying network implementation as specified by the CAtlHttpClientT template parameter. It is not normally necessary to call **Close**, because it is called as necessary by the CAtlHttpClientT implementation.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::FindAuthObject

Call this method to obtain access to the authorization object for a given security scheme, if present.

```
const CAtlBaseAuthObject* FindAuthObject(
   LPCTSTR szScheme
) throw( );
```

Parameters

szScheme

A string containing a description of the security scheme implemented by the desired authorization object, such as "BASIC" or "NTLM".

Return Value

On success, returns a pointer to the CAtlBaseAuthObject base class, or zero if no such authorization object is present.

Remarks

Tries to find an authorization object implementing the security scheme indicated by *szScheme*. Authorization objects can be added with AddAuthObj and removed with RemoveAuthObject.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetBody

Call this method to retrieve access to the HTTP response body data.

```
const BYTE* GetBody( ) throw( );
```

Return Value

A pointer to the body data, or 0 if no body was received.

Remarks

The response body contains the data retrieved as a result of the previous call to Navigate but does not include header information. This data is not processed in any way by **CAtlHttpClientT**. No parsing is performed for HTML or script data, and binary data is returned in raw form as retrieved from the server.

The size of the response body can be retrieved with CAtlHttpClientT::GetBodyLength.

Example

See the HttpClient Sample.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetBodyLength

Call this method to determine the size of the response data, not including header information.

```
DWORD GetBodyLength( ) const throw( );
```

Return Value

The size in bytes of the HTTP response data. This value refers to the response body only and does not include the size of the response headers.

Remarks

Returns the size of the buffer returned by the CAtlHttpClientT::GetBody method.

Example

See the HttpClient Sample.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetCurrentUrl

Call this method to gain access to the URL being used for requests by this object.

```
LPCURL GetCurrentUrl( ) const throw( );
```

Return Value

Returns a const pointer to a CUrl object that represents the current URL.

Remarks

Initially, HTTP requests are launched by passing URL information to one of the CAtlHttpClientT::Navigate overloads. The URL returned by **GetCurrentUrl** may differ, however, because **CAtlHttpClientT** builds a fully canonical URL, and the initial URL may be replaced entirely as the result of redirection.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetFlags

Call this method to retrieve a copy of the behavior control flags.

```
DWORD GetFlags( ) const throw( );
```

Return Value

Returns a combination of one or more of the ATL_HTTP Flags.

Remarks

By default, the **ALT_HTTP_FLAG_AUTO_REDIRECT**, **ATL_HTTP_FLAG_PROCESS_RESULT**, and **ATL_HTTP_FLAG_SEND_BLOCKS** flags are used.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetHeaderValue

Call this method to retrieve HTTP message header values.

```
bool GetHeaderValue(
   LPCTSTR szName,
   LPTSTR szBuffer,
   DWORD* pdwLen
) const throw( );
bool GetHeaderValue(
   LPCTSTR szName,
   CString& strValue
) const throw( );
```

Parameters

szName

A string containing the name of the header key, such as "Content-Type", and "Content-Length".

szBuffer

A buffer where the header value is to be stored.

pdwLen

The length of the buffer.

strValue

A **CString** reference where the header value is to be stored.

Return Value

Returns **true** if the header key is present in the current response, and the header value is successfully stored, otherwise **false**.

Remarks

The **GetHeaderValue** method is useful for retrieving the values of known header keys. The entire response header body can be retrieved using the GetRawResponseHeader method.

Example

See the HttpClient Sample.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetLastError

Call this method to retrieve the last error code reported by the underlying network socket.

```
DWORD GetLastError( ) throw( );
```

Return Value

Returns the last error code, as reported by the socket implementation class.

Remarks

The error values returned by CAtlHttpClientT::GetLastError may vary depending the implementation of the socket class.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetMethod

Call this method to retrieve the HTTP method being used.

```
LPCTSTR GetMethod( ) throw( );
```

Return Value

A string containing the HTTP method, such as "GET", or "POST".

Remarks

The default HTTP method is "GET".

See Also

Reference

CAtlNavigateData::SetMethod

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetPostData

Call this method to gain access to outgoing request data buffer.

```
BYTE* GetPostData( ) throw( );
```

Return Value

A pointer to a buffer containing the body of the HTTP request to be made, or NULL if no request body has been assigned.

Remarks

By default, no body is associated with HTTP requests made by the **CAtlHttpClientT** class. Request body or post data can be assigned using the CAtlNavigateData::SetPostData method.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetPostDataLen

Call this method to determine the size of the outgoing request data buffer.

```
DWORD GetPostDataLen( ) throw( );
```

Return Value

The size of the request body buffer associated with this object, or 0 if no post data has been assigned.

Remarks

Post buffers can be assigned using CAtlNavigateData::SetPostData and retrieved using CAtlHttpClientT::GetPostData.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetPostDataType

Call this method to retrieve the type of post data associated with this object.

```
LPCTSTR GetPostDataType( ) throw( );
```

Return Value

A string containing the data type of the buffer as described in the "Content-Type" HTTP header, or **NULL** if no post data has been assigned.

Remarks

Post buffers can be assigned using the CAtlNavigateData::SetPostData method, and retrieved using the CAtlHttpClientT::GetPostData method.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetProxy

Call this method to get the proxy server settings for this object.

```
LPCTSTR GetProxy( ) const throw( );
```

Return Value

A const pointer to a string containing the proxy server name and port number. The proxy server name is followed by a colon and the port number. If no proxy server settings have been applied, **GetProxy** returns **NULL**.

Remarks

Retrieves the current proxy server settings, as applied with a previous call to SetProxy. The **GetProxy** method is useful if the SetProxy method was called with zero as a proxy server name, because this triggers the retrieval of proxy server settings from the registry.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetProxyPort

Call this method to retrieve the proxy server port setting for this object.

```
short GetProxyPort() const throw( );
```

Return Value

The current proxy server port setting. This value defaults to **ATL_URL_INVALID_PORT_NUMBER**.

Remarks

Retrieves the current proxy server port setting, as applied with a previous call to SetProxy. The **GetProxyPort** method is useful if the SetProxy method was called with zero as a proxy server name, because this triggers the retrieval of proxy server settings from the registry.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetRawResponseHeader

Call this method to retrieve a copy of the last HTTP response headers received.

```
bool GetRawResponseHeader(
   LPBYTE szBuffer,
   DWORD* pdwLen
) throw( );
```

Parameters

szBuffer

A pointer to a buffer that is to receive the response header data. If 0, the size of the response headers is returned by the *pdwLen* argument, and the header data is not retrieved.

pdwLen

The size of the buffer.

Return Value

true if successful, otherwise **false**.

Remarks

The size specified by *pdwLen* must be sufficient for storage of the response header data. The required size can be determined either by using the GetRawResponseHeaderLength method or by passing 0 for the *szBuffer* argument.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetRawResponseHeaderLength

Call this method to retrieve the size, in bytes, of the last response's header information.

```
DWORD GetRawResponseHeaderLength( ) throw( );
```

Return Value

Returns the response header length in bytes.

Remarks

This method returns the size of the response header retrieved as a result of the last call to the Navigate method. Once the header size is retrieved, the GetRawResponseHeader method can be used to retrieve a copy of the response header data. Alternately, direct access to the response header data can be retrieved using the GetResponse method.

Example

See the HttpClient Sample and the HttpPing Sample.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetResponse

Call this method to obtain access to the buffer that holds the entire response of the last request processed.

```
const BYTE* GetResponse( ) throw( );
```

Return Value

A pointer to the response data, or 0 if no response was retrieved.

Remarks

The response is the raw data retrieved from the server in its entirety, starting with the header information and immediately followed by the response body. For a URL indicating a binary entity, the response will include a text-based header followed by binary data. To determine the size of the response buffer returned by the **GetResponse** method, use the GetResponseLength method.

Alternately, the response header and body can be retrieved separately using the GetRawResponseHeader and GetBody methods, respectively.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetResponseLength

Call this method to retrieve the length, in bytes, of the buffer returned by GetResponse.

```
DWORD GetResponseLength( ) throw( );
```

Return Value

Returns the size of the response buffer.

Remarks

The **GetResponseLength** method returns the size of the response data retrieved as a result of the last call to the Navigate method. The response data can be accessed using the CAtlHttpClientT::GetResponse method.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetResponseStatus

Call this method to retrieve the current status of the response processing mechanisms.

```
HTTP_RESPONSE_READ_STATUS GetResponseStatus();
```

Return Value

Returns one of the values from the HTTP_RESPONSE_READ_STATUS enumeration.

Remarks

GetResponseStatus returns the **HTTP_RESPONSE_READ_STATUS** value that best describes the current state of response processing. Once the response has been retrieved fully and successfully, **RR_OK** will be returned.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetSocket

Call this method to gain access to the underlying network socket.

```
const SOCKET& GetSocket( ) throw( );
```

Return Value

A reference to the underlying SOCKET object.

Remarks

Direct access to the socket is not normally required.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetSocketTimeout

Call this method to retrieve the current socket timeout setting.

```
DWORD GetSocketTimeout( ) throw( );
```

Return Value

Returns the socket timeout, in milliseconds.

Remarks

The timeout specifies how long the underlying socket will wait for blocking network events such as send, receive, and connect to occur. This value is ATL_SOCK_TIMEOUT by default and can be adjusted with the SetSocketTimeout method.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::GetStatus

Call this method to retrieve the HTTP status code for the previous request.

```
int GetStatus( ) throw( );
```

Return Value

Returns the HTTP status code.

Remarks

HTTP status codes are defined by the HTTP specification.

Example

See the HttpClient Sample and the HttpPing Sample.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::Navigate

Call this method to perform an HTTP request.

```
bool Navigate(
   LPCTSTR szServer,
   LPCTSTR szPath,
   ATL_NAVIGATE_DATA * pNavData = NULL
) throw(...);
bool Navigate(
   const CUrl* pUrl,
   ATL_NAVIGATE_DATA * pNavData = NULL
) throw(...);
bool Navigate(
   LPCTSTR szURL,
   ATL_NAVIGATE_DATA * pNavData = NULL
) throw(...);
```

Parameters

szServer

A string containing the server name, such as "www.microsoft.com", from which the request will be made.

szPath

A string containing the path of the desired item, relative to the server.

pUrl

The address of a CUrl object containing a description of the URL from which the request will be made.

szURL

A string containing the complete URL for the desired item. This string must be prefixed with the "http://" protocol specification.

pNavData

A pointer to an ATL_NAVIGATE_DATA or CAtlNavigateData object. This argument is used to specify navigation options.

If this argument is **NULL**, **CAtlHttpClientT** uses the navigation options provided by an internal instance of CAtlNavigateData. As a result, the settings provided by the CAtlNavigateData constructor will apply.

Return Value

Returns **true** on success, **false** on failure.

Remarks

Use these methods to perform HTTP client transactions. Each overload of the **Navigate** method can be used to retrieve responses from a URL. Using the URL and optional settings specified with the ATL_NAVIGATE_DATA structure, CAtlHttpClientT performs the HTTP request and stores any corresponding response.

A breaking change from previous versions, **Navigate** no longer passes user logon credentials for authentication unless the security zone settings allow it. This change increases the security of user credentials. This may break applications that relied on this non-secure method of authentication. This change only affects applications that use authentication objects supplied with **AddAuthObj**. If an authorization error occurs, an HTTP 401 code will be returned to the user's browser.

Example

See the HttpClient Sample and the HttpPing Sample.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::NavigateChunked

Call this method to perform a chunked transfer encoded HTTP request.

```
bool NavigateChunked(
   const CUrl * pUrl,
   ATL_NAVIGATE_DATA * pData
) throw( );
bool NavigateChunked(
   LPCTSTR szURL,
   ATL_NAVIGATE_DATA * pNavData
) throw( );
bool NavigateChunked(
   LPCTSTR szServer,
   LPCTSTR szPath,
   ATL_NAVIGATE_DATA * pNavData
) throw( );
```

Parameters

pUrl

The address of a CUrl object containing a description of the URL from which the request will be made.

szURL

A string containing the complete URL for the desired item. This string must be prefixed with the "http://" protocol specification.

szServer

A string containing the server name, such as "www.microsoft.com", from which the request will be made.

szPath

A string containing the path of the desired item, relative to the server.

pNavData

A pointer to an ATL_NAVIGATE_DATA or CAtlNavigateData object. This argument is used to specify navigation options. Must not be **NULL**.

Return Value

Returns **true** on success, **false** on failure.

Remarks

The **NavigateChunked** method makes HTTP transactions using chunked-transfer encoded requests. The advantage of using chunked requests is that the size of the request does not have to be known before transmission begins. Chunked-transfer encoding is part of the HTTP 1.1 specification and is not supported by previous implementations. For nonchunked requests, use the Navigate method.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::NegotiateAuth

Seeks authorization to secured URLs using any security protocols supported by the provided authorization objects.

```
virtual bool NegotiateAuth(
   bool bProxy
) throw( );
```

Parameters

bProxy

true if a proxy authorization is required in response to a 407 HTTP response; otherwise **false**.

Return Value

Returns **true** on success, **false** on failure.

Remarks

Using any authorization objects added with the AddAuthObj method, the **NegotiateAuth** method attempts to meet the security requirements of the requested URL.

The default version of the **NegotiateAuth** method tries each authorization object present, in the order they were added with the AddAuthObj method. Each authorization object present is activated by **NegotiateAuth** with a call to the CAtlBaseAuthObject::Authenticate method, until one of the authorization objects succeeds or all of them have failed. Authorization schemes that have been flagged as having failed during previous operations are not used. This default behavior can be modified in derived classes by overriding the **NegotiateAuth** method.

The *bProxy* parameter indicates whether authorization is required as the result of a proxy requirement (HTTP code 407), or a general authorization requirement (HTTP code 401).

A breaking change from previous versions, **NegotiateAuth** no longer passes user log on credentials for authentication unless the security zone settings allow it. This change increases the security of user credentials. This may break applications that relied on this non-secure method of authentication. To restore the old behavior, call **SetSilentLogonOK** with the parameter value **true**.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::RemoveAuthObject

Call this method to remove support for a given authorization scheme.

```
bool RemoveAuthObject(
   LPCTSTR szScheme
) throw( );
```

Parameters

szScheme

A string containing a description of the security scheme implemented by the desired authorization object, such as "basic" or "ntlm".

Return Value

true if the authorization object was found and successfully removed; otherwise **false**.

Remarks

Attempts to locate and remove the authorization object correlating to the scheme described by *szScheme*. Authorization objects can be added with the AddAuthObj method and accessed with the FindAuthObject method.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::RemoveProxy

Call this method to remove proxy server settings from this object.

```
void RemoveProxy( ) throw( );
```

Remarks

Removes the current proxy settings previously assigned using the CAtlHttpClientT::SetProxy method.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::SetProxy

Call this method to assign a proxy server to be used for subsequent HTTP transactions.

```
bool SetProxy(
   LPCTSTR szProxy = NULL,
   short nProxyPort = 0
) throw( );
```

Parameters

szProxy

A string containing the name of a proxy server. The proxy server name should not include a HTTP protocol prefix. For example, using "http://myproxyserver" will cause requests to fail, but **SetProxy** will not return **false** to indicate failure. This parameter defaults to **NULL**, which causes **CAtlHttpClientT** to retrieve proxy server information from the registry. See **Remarks** for more details.

nProxyPort

The port number for the proxy server, typically 80. This argument is ignored if zero is used for *szProxy*.

Return Value

Returns **true** on success, **false** on failure.

Remarks

Calling this method causes all HTTP transactions to be routed through the given proxy server. Calling **SetProxy** with no arguments is recommended, because proxy server information will be retrieved from the registry instead of being hard-coded in the application. The registry key used to retrieve the proxy server settings is:

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Internet
```

Example

See the HttpClient Sample and the HttpPing Sample.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::SetSilentLogonOk

Call this method to control the security settings for transmitting user credentials.

```
void SetSilentLogonOk(
   BOOL bSet
);
```

Parameters

bSet

The new value of the **SilentLogonOk** flag.

Remarks

Call this method to set whether authentication is allowed to pass user credentials to sites outside of the trusted zone. The default value is false, which is the more secure option. A value of **false** means that prior to sending user credentials to an internet site when an HTTP redirect occurs, the zone of the URL is checked to determine if sending these credentials is allowed by the security policy in effect for that zone. The default settings only allow credentials to be sent to intranet sites. A value of **true** allows this security check to be bypassed.

The **SilentLogonOK** flag is only meaningful if an authentication object (**IAuthObj**) is applied to the **CAtlHttpClient** object. An authentication object may be added by calling **AddAuthObj**.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT::SetSocketTimeout

Call this method to set the timeout value for the underlying network socket.

```
DWORD SetSocketTimeout(
   DWORD dwNewTimeout
) throw( );
```

Parameters

dwNewTimeout

The new timeout in milliseconds.

Return Value

The previous timeout in milliseconds.

Remarks

Call this method to specify how long the underlying socket will wait for blocking network events such as send, receive and connect to occur. If the CAtlNavigateData class is being used with the Navigate method, the socket timeout value can also be manipulated with the CAtlNavigateData::GetSocketTimeout and CAtlNavigateData::SetSocketTimeout methods.

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlHttpClientT Enums

For information about the enums in **CAtlHttpClientT**, see CAtlHttpClientT Members

CAtlHttpClientT::HTTP_RESPONSE_READ_STATUS

Constants that indicate the status of an attempt to read the HTTP response.

```
enum HTTP_RESPONSE_READ_STATUS;
```

Remarks

The elements of this enumeration are described in the following table:

Value	Description
RR_OK	Response processed successfully.
RR_FAIL	An unknown error occurred.
RR_STATUS_INVALID	Failed to parse status line.
RR_PARSEHEADERS_FAILED	Failure during HTTP header parsing.
RR_READSOCKET_FAILED	Failed to read response from network socket.
RR_READBODY_FAILED	Failed to read response body in its entirety.
RR_READCHUNKEDBODY_FAILED	Failed to read a chunked-transfer encoded response.
RR_NOT_READ	Response has not been read yet.

See Also

Reference

CAtlHttpClientT Class

CAtlHttpClientT::GetResponseStatus

Concepts

HTTP Client Services

Other Resources

CAtlHttpClientT Members

HTTP Client Reference

CAtlIsapiBuffer Class

This class is an efficient buffer that manages its own memory and is used by the ATL Server code to buffer data in CHttpResponse and CAtlHttpClient.

```
template <
   DWORD dwSizeT = ATL_ISAPI_BUFFER_SIZE
>
class CAtlIsapiBuffer
```

Parameters

dwSizeT

The size of the statically allocated buffer in bytes.

Remarks

This class represents a dynamically resizable string buffer. Data can be written to the buffer using CAtlIsapiBuffer::Append or CAtlIsapiBuffer::operator +=. Data can be retrieved from the buffer using CAtlIsapiBuffer::operator LPCSTR.

For efficiency, each instance of this class statically allocates a buffer of *dwSizeT* bytes. Typically, this static buffer is used by calls to **CAtlIsapiBuffer::Append** until the amount of data written exceeds the size of the static buffer, at which point the class creates a new buffer and copies the existing data into it. As more data is appended, the buffer grows to accommodate it. The buffer grows in fixed size blocks or just enough to accommodate the new data, whichever is greater.

The class provides the CAtlIsapiBuffer::Alloc, CAtlIsapiBuffer::ReAlloc, and CAtlIsapiBuffer::Free methods to allow control over memory allocations and increase efficiency when use of the buffer is well understood.

Note that if the buffered data never exceeds the size of the static allocation, using this class allows you to avoid any run-time memory allocations and achieve significant speed improvements. If data is written to the static buffer initially, but the data eventually exceeds the size of the static allocation, there is a run-time cost associated with copying data from the static buffer to the dynamic buffer and a memory cost since the static buffer is no longer used.

You may want to define your own value for ATL_ISAPI_BUFFER_SIZE if you use the CHttpResponse and CAtlHttpClient classes. These classes have **CAtlIsapiBuffer** members with the static buffer initialized to the size of this macro. You can define **ATL_ISAPI_BUFFER_SIZE** to be 0 if you regularly set the size of the buffer programmatically, for example by calling CHttpResponse::SetBufferLimit, to avoid wasting memory when the size of the buffered data is larger than the statically allocated memory.

Note
A **CAtlIsapiBuffer** object cannot hold more than 4MB on Win9x operating systems.

Example

See the SOAPTransport Sample.

Requirements

Header: atlutil.h

See Also

Other Resources

CAtlIsapiBuffer Members

ATL Server Classes

CAtlIsapiBuffer Members

Methods

Alloc	Call this method to increase the size of the dynamic buffer managed by this class.
Append	Call this method to append data to the buffer.
CAtlIsapiBuffer	The constructor.
~CAtlIsapiBuffer	The destructor.
Empty	Call this method to empty the buffer of data.
Free	Call this method to free any dynamically allocated memory used by this object.
GetLength	Call this method to get the length of the data in the buffer.
ReAlloc	Call this method to reallocate memory allocated by this object.

Operators

operator +=	This operator can be used to append data to the buffer.
operator LPCSTR	This operator can be used to retrieve the data contained in the buffer. It allows you to treat instances of this class as if they were strings.

See Also

Reference

CAtlIsapiBuffer Class

CAtlIsapiBuffer Methods

For information about the methods in **CAtlIsapiBuffer**, see CAtlIsapiBuffer Members.

CAtlIsapiBuffer::Alloc

Call this method to increase the size of the dynamic buffer managed by this class.

```
BOOL Alloc(
   DWORD dwSize
) throw( );
```

Parameters

dwSize

The requested size in bytes of the buffer.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

If the memory allocated by this class is already larger than *dwSize*, no new allocation is made and the method returns TRUE. Otherwise, the method frees the existing buffer and allocates memory of the requested size.

See Also

Reference

CAtlIsapiBuffer Class

Other Resources

CAtlIsapiBuffer Members

CAtlIsapiBuffer::Append

Call this method to append data to the buffer.

```
BOOL Append(
   LPCSTR sz,
   int nLen = -1
) throw( );
```

Parameters

sz

The data to write to the buffer.

nLen

The length in bytes of the data to be written to the buffer. If *nLen* is -1, the length of the data is determined by a terminating null character.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

If the data will exceed the current size of the buffer, the buffer will grow to accommodate it. See CAtlIsapiBuffer Overview for further details.

The buffer always terminates the data added to it with a null character which is not included in the length returned by CAtlIsapiBuffer::GetLength.

See Also

Reference

CAtlIsapiBuffer Class
CAtlIsapiBuffer::GetLength

Other Resources

CAtlIsapiBuffer Members

CAtlIsapiBuffer::CAtlIsapiBuffer

The constructor.

```
CAtlIsapiBuffer( ) throw( );
CAtlIsapiBuffer(
   LPCSTR sz
);
```

Parameters

sz

A string to be copied to the buffer.

See Also

Reference

CAtlIsapiBuffer Class

Other Resources

CAtlIsapiBuffer Members

CAtlIsapiBuffer::~CAtlIsapiBuffer

The destructor.

```
~CAtlIsapiBuffer( ) throw( );
```

Remarks

The destructor frees any allocated memory.

See Also

Reference

CAtlIsapiBuffer Class

Other Resources

CAtlIsapiBuffer Members

CAtlIsapiBuffer::Empty

Call this method to empty the buffer of data.

```
void Empty( ) throw( );
```

Remarks

No memory is freed as a result of calling this method. Call CAtlIsapiBuffer::Free to release any memory allocated by this object.

See Also

Reference

CAtlIsapiBuffer Class

CAtlIsapiBuffer::Free

Other Resources

CAtlIsapiBuffer Members

CAtlIsapiBuffer::Free

Call this method to free any dynamically allocated memory used by this object.

```
void Free( ) throw( );
```

Remarks

Call CAtlIsapiBuffer::Empty to empty the buffer of data without releasing any memory allocated by this object.

See Also

Reference

CAtlIsapiBuffer Class

CAtlIsapiBuffer::Empty

Other Resources

CAtlIsapiBuffer Members

CAtlIsapiBuffer::GetLength

Call this method to get the length of the data in the buffer.

```
DWORD GetLength( ) throw( );
```

Return Value

Returns the length in bytes of the data in the buffer.

Remarks

This method returns the amount of user data in the buffer; it does not return the amount of memory allocated for the buffer.

Note that the buffer maintains a null character immediately following the data you add to the buffer. This character is not included in the length returned by this method.

See Also

Reference

CAtlIsapiBuffer Class

Other Resources

CAtlIsapiBuffer Members

CAtlIsapiBuffer::ReAlloc

Call this method to reallocate memory allocated by this object.

```
BOOL ReAlloc(
   DWORD dwNewSize
) throw( );
```

Parameters

dwNewSize

The requested number of bytes in the buffer.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Call this method to extend the buffer while keeping existing data intact. If the memory allocated by this class is already larger than *dwNewSize*, no new allocation is made and the method returns TRUE.

See Also

Reference

CAtlIsapiBuffer Class

Other Resources

CAtlIsapiBuffer Members

CAtlIsapiBuffer Operators

For information about the operators in **CAtlIsapiBuffer**, see CAtlIsapiBuffer Members.

CAtlIsapiBuffer::operator +=

This operator can be used to append data to the buffer.

```
CAtlIsapiBuffer& operator +=(
   LPCSTR sz
);
```

Parameters

sz

The data to write to the buffer.

Return Value

Returns a reference to the current object.

Remarks

This operator is equivalent to CAtlIsapiBuffer::Append.

See Also

Reference

CAtlIsapiBuffer Class

CAtlIsapiBuffer::Append

Other Resources

CAtlIsapiBuffer Members

CAtlIsapiBuffer::operator LPCSTR

This operator can be used to retrieve the data contained in the buffer. It allows you to treat instances of this class as if they were strings.

```
operator LPCSTR( ) throw( );
```

See Also

Reference

CAtlIsapiBuffer Class

Other Resources

CAtlIsapiBuffer Members

CAtlNavigateData Class

This class represents HTTP transaction options for use with the CAtlHttpClientT class.

```
class CAtlNavigateData :
   public ATL_NAVIGATE_DATA
```

Remarks

The **CAtlNavigateData** class encapsulates the ATL_NAVIGATE_DATA structure, adding initialization and access and retrieval functionality. The ATL_NAVIGATE_DATA structure is used to provide settings to the CAtlHttpClientT class. Except for the URL itself, virtually all of the settings required to perform an HTTP request are provided by the ATL_NAVIGATE_DATA structure.

The **CAtlNavigateData** class can be used with the CAtlHttpClientT::Navigate method to specify the details of HTTP transactions. These settings include auxiliary header information, the HTTP request method, and behavior modification flags. The **CAtlNavigateData** class can also be used to install send and receive callback functions for increased control during HTTP transactions.

Explicit use of the **CAtlNavigateData** class is optional. If no navigation data is specified, the CAtlHttpClientT class uses an internal instance of **CAtlNavigateData**. As such, default HTTP transaction settings are dictated by the CAtlNavigateData constructor.

Requirements

Header: atlhttp.h

Example

See the HttpClient Sample.

See Also

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData Members

Methods

AddFlags	Call this method to add HTTP transaction behavior modification flags.
CAtlNavigateData	Constructs an instance of the **CAtlNavigateData** class.
GetChunkCallback	Call this method to get the callback used for sending chunk-encoded data.
GetExtraHeaders	Call this method to access any additional headers to be included in the HTTP request.
GetFlags	Call this method to retrieve the current behavior modification flags.
GetMethod	Call this method to retrieve the currently active HTTP request method.
GetPort	Call this method to get the current port number of the HTTP server from which requests will be made.
GetReadBlockSize	Call this method to retrieve the size of the internal HTTP response buffer.
GetReadStatusCallback	Call this method to retrieve the callback function that, if assigned, provides the application with status updates on the arriving response.
GetSendBlockSize	Call this method to retrieve the block size used to send request data when using ATL_HTTP_FLAG_SEND_BLOCKS.
GetSendStatusCallback	Call this method to retrieve the callback used for request processing.
GetSocketTimeout	Call this method to get the socket timeout value.
RemoveFlags	Call this method to remove behavior modification flags.
SetChunkCallback	Call this method to assign a callback function for use with chunk-encoded requests.
SetPostData	Call this method to attach POST data to the next HTTP request.
SetExtraHeaders	Call this method to assign additional headers to be included in the HTTP request.
SetFlags	Call this method to assign behavior modification flags.
SetMethod	Call this method to set the HTTP method to be used for the next request.
SetPort	Call this method to assign the port number of the HTTP server to be used for subsequent requests.
SetReadBlockSize	Call this method to assign the size of the incoming response buffer.
SetReadStatusCallback	Call this method to assign a response processing callback function.
SetSendBlockSize	Call this method to assign the size of the outgoing request buffer.
SetSendStatusCallback	Call this method to assign a request status callback function.

SetSocketTimeout	Call this method to set the socket timeout value.

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

HTTP Client Reference

CAtlNavigateData Methods

For information about the methods in **CAtlNavigateData**, see CAtlNavigateData Members

CAtlNavigateData::AddFlags

Call this method to add HTTP transaction behavior modification flags.

```
DWORD AddFlags(
   DWORD dwFlagsToAdd
) throw( );
```

Parameters

dwFlagsToAdd

A combination of one or more of the ATL_HTTP flags.

Return Value

Returns a **DWORD** containing the previous flags.

Remarks

The **AddFlags** method is convenient for setting behavior modification flags without disturbing other flag states.

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::CAtlNavigateData

Constructs an instance of the **CAtlNavigateData** class.

```
CAtlNavigateData( ) throw( );
```

Remarks

The **CAtlNavigateData** constructor assigns default values to the data members inherited from the ATL_NAVIGATE_DATA structure. These default values are important because CAtlHttpClientT uses an internal instance of **CAtlNavigateData** if no navigation settings are provided to the CAtlHttpClientT::Navigate method. The **CAtlNavigateData** constructor assigns these values:

Data member	Default value
szExtraHeaders	**NULL**
szMethod	**ATL_HTTP_METHOD_GET**
szDataType	**NULL**
dwDataLen	**0**
dwFlags	**ATL_HTTP_FLAG_AUTO_REDIRECT** | **ATL_HTTP_FLAG_PROCESS_RESULT**| **ATL_HTTP_FLAG_SEND_BLOCKS**
dwTimeout	**ATL_SOCK_TIMEOUT**
dwSendBlockSize	**ATL_HTTP_DEFAULT_BLOCK_SIZE**
dwReadBlockSize	**ATL_HTTP_DEFAULT_BLOCK_SIZE**
m_lParamChunkCB	**NULL**
m_lParamSend	**NULL**
m_lParamRead	**NULL**
nPort	**ATL_URL_DEFAULT_HTTP_PORT**
pData	**NULL**
pfnChunkCallback	**NULL**
pfnSendStatusCallback	**NULL**
pfnReadStatusCallback	**NULL**

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::GetChunkCallback

Call this method to get the callback used for sending chunk-encoded data.

```
PFNATLCHUNKEDCB GetChunkCallback( ) throw( );
```

Return Value

A pointer to the callback used for chunk-encoded data transfer, or **NULL** if no callback function has been provided. Returns **NULL** by default.

Remarks

Chunked-data encoding, as defined by the HTTP 1.1 specification, allows outgoing data to be sent without first knowing the length of the data. The **GetChunkCallback** method retrieves a pointer to the function that will be used to a send chunked data when the CAtlHttpClientT::NavigateChunked method is called to initiate a chunked-data transfer. Chunked data callbacks can be assigned using SetChunkCallback.

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::GetExtraHeaders

Call this method to access any additional headers to be included in the HTTP request.

```
LPCTSTR GetExtraHeaders( ) throw( );
```

Return Value

A pointer to a buffer containing auxiliary headers, or **NULL** if no additional headers have been assigned. Returns **NULL** by default.

Remarks

In addition to standard HTTP header values such as "Content-Type", the CAtlHttpClientT class sends additional header information assigned using CAtlNavigateData::SetExtraHeaders.

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::GetFlags

Call this method to retrieve the current behavior modification flags.

```
DWORD GetFlags( ) throw( );
```

Return Value

A DWORD containing the ATL_HTTP flags currently in effect.

Remarks

By default, the **ATL_HTTP_FLAG_AUTO_REDIRECT**, **ATL_HTTP_FLAG_PROCESS_RESULT**, and **ATL_HTTP_FLAG_SEND_BLOCKS** flags are set. Additional flag manipulation and retrieval methods are: SetFlags, AddFlags, and RemoveFlags.

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::GetMethod

Call this method to retrieve the currently active HTTP request method.

```
LPCTSTR GetMethod( ) throw( );
```

Return Value

A string containing the method to be used for the next HTTP request, such as "GET" or "POST".

Remarks

The default HTTP method is ATL_HTTP_METHOD_GET, which, unless redefined, is defined as "GET".

See Also

Reference

CAtlNavigateData Class

ATL_HTTP_METHOD_POST

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::GetPort

Call this method to get the current port number of the HTTP server from which requests will be made.

```
short GetPort( ) throw( );
```

Return Value

Returns the port number.

Remarks

By default, the port number is set to ATL_URL_DEFAULT_HTTP_PORT. The port number can be set using SetPort.

The **CAtlNavigateData::GetPort** and CAtlNavigateData::SetPort methods refer to the port of the HTTP server and not to the proxy server. Proxy server settings can be assigned using CAtlHttpClientT::SetProxy.

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::GetReadBlockSize

Call this method to retrieve the size of the internal HTTP response buffer.

```
DWORD GetReadBlockSize( ) throw( );
```

Return Value

Returns the size of the buffer used to process HTTP responses, in bytes.

Remarks

This method retrieves the size of the buffer that is used for processing normal, non-

chunk-encoded responses. This value is used only for requests that use ATL_HTTP_FLAG_SEND_BLOCKS. By default, the block size is set to ATL_HTTP_DEFAULT_BLOCK_SIZE.

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::GetReadStatusCallback

Call this method to retrieve the callback function that, if assigned, provides the application with status updates on the arriving response.

```
PFNATLSTATUSCALLBACK GetReadStatusCallback( ) throw( );
```

Return Value

A pointer to a callback used to notify the application of new response data, or **NULL** if no read status callback has been installed. Returns **NULL** by default.

Remarks

If a read status callback is installed using the SetReadStatusCallback method, the CAtlHttpClient class will call the provided function whenever new response data arrives. This mechanism allows for the display of partially retrieved responses, such as images that are displayed incrementally as data arrives, and is also useful for updating progress bars.

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::GetSendBlockSize

Call this method to retrieve the block size used to send request data when using ATL_HTTP_FLAG_SEND_BLOCKS.

```
DWORD GetSendBlockSize( ) throw( );
```

Return Value

Returns the size of the buffer used to send HTTP requests, in bytes.

Remarks

This method returns the size of the buffer that the CAtlHttpClientT uses to process requests. By default, the block size is ATL_HTTP_DEFAULT_BLOCK_SIZE.

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::GetSendStatusCallback

Call this method to retrieve the callback used for request processing.

```
PFNATLSTATUSCALLBACK GetSendStatusCallback( ) throw( );
```

Return Value

A pointer to a callback function, or **NULL** if no send status callback has been installed. Returns **NULL** by default.

Remarks

If the SetSendStatusCallback method is used to install a send status callback function, CAtlHttpClientT will call the function after each block of request data has been sent. This mechanism is useful for displaying a progress bar to track upload progress for large requests.

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::GetSocketTimeout

Call this method to get the socket timeout value.

```
DWORD GetSocketTimeout( ) throw( );
```

Return Value

Returns the socket timeout, in milliseconds.

Remarks

The socket timeout value determines the length of time the socket will wait for responses before abandoning the operation. The socket timeout value can be assigned using the SetSocketTimeout method.

See Also

Reference

CAtlNavigateData Class

CAtlHttpClientT::GetSocketTimeout

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::RemoveFlags

Call this method to remove behavior modification flags.

```
DWORD RemoveFlags(
   DWORD dwFlagsToRemove
) throw( );
```

Parameters

dwFlagsToRemove

The ATL_HTTP flags to remove from the current behavior modification settings.

Return Value

A **DWORD** containing the previous flag states.

Remarks

The **RemoveFlags** method is useful for removing specific flags without disturbing existing flag states. Additional flag manipulation and retrieval methods are GetFlags, SetFlags, and AddFlags.

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::SetChunkCallback

Call this method to assign a callback function for use with chunk-encoded requests.

```
PFNATLCHUNKEDCB SetChunkCallback(
   PFNATLCHUNKEDCB pfn,
   DWORD_PTR dwParam
) throw( );
```

Parameters

pfn

A pointer to the chunk-encoding callback as defined by the PFNATLCHUNKEDCB typedef.

dwParam

User-defined value to be passed to the callback function.

Return Value

Returns a pointer to a previously assigned callback, or **NULL** if no callback was assigned.

Remarks

The **SetChunkCallback** method is necessary for use of chunk-encoded requests, as specified by the HTTP 1.1 RFC. Chunk-encoded requests allow for request data whose size is not known when the request is initiated and can be launched with the CAtlHttpClientT::NavigateChunked method. The GetChunkCallback method can be used to retrieve the currently installed callback.

See Also

Reference

CAtlNavigateData Class

PFNATLCHUNKEDCB

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::SetExtraHeaders

Call this method to assign additional headers to be included in the HTTP request.

```
LPCTSTR SetExtraHeaders(
   LPCTSTR szNewHeaders
) throw( );
```

Parameters

szNewHeaders

A pointer to a buffer containing the auxiliary headers.

Return Value

Returns a pointer to previously assigned extra header data, or NULL if not present.

Remarks

In addition to standard HTTP header values such as "Content-Type", the CAtlHttpClientT class sends any additional header information assigned with the **SetExtraHeaders** method. Access to existing auxiliary headers can be achieved using the GetExtraHeaders method.

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::SetFlags

Call this method to assign behavior modification flags.

```
DWORD SetFlags(
   DWORD dwNewFlags
) throw( );
```

Parameters

dwNewFlags

A **DWORD** containing the new set of behavior modification flags. See ATL_HTTP Flags for possible values.

Return Value

A **DWORD** containing the previous flag states.

Remarks

The **SetFlags** method overrides any existing flags. To add flags to the existing flag states, use the AddFlags method. The GetFlags method can be used to retrieve the current flag states.

See Also

Reference

CAtlNavigateData Class

CAtlNavigateData::RemoveFlags

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::SetMethod

Call this method to set the HTTP method to be used for the next request.

```
LPCTSTR SetMethod(
   LPCTSTR szNewMethod
) throw( );
```

Parameters

szNewMethod

A string containing the new HTTP method.

Return Value

Returns a string containing the previous HTTP method.

Remarks

By default, the HTTP method used is defined by the ATL_HTTP_METHOD_GET macro. The GetMethod method can be used to retrieve the current HTTP method.

See Also

Reference

CAtlNavigateData Class

ATL_HTTP_METHOD_POST

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::SetPort

Call this method to assign the port number of the HTTP server to be used for subsequent requests.

```
short SetPort(
   short newPort
) throw( );
```

Parameters

newPort

The desired port number.

Return Value

The previous port number. By default, the port number is set to ATL_URL_DEFAULT_HTTP_PORT.

Remarks

The HTTP server port number can be retrieved using the GetPort method. The **SetPort** and GetPort methods refer to the HTTP server and not a proxy server. See the CAtlHttpClientT::SetProxy method if you require the use of a proxy server.

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::SetPostData

Call this method to attach POST data to the next HTTP request.

```
void SetPostData(
   BYTE* pData,
   DWORD dwDataLen,
   LPCTSTR szDataType
) throw( );
```

Parameters

pData

A pointer to the POST data.

dwDataLen

The size of the POST data buffer specified by *pData*.

szDataType

A string containing the value to be used for the "Content-Type" header.

Remarks

The **SetPostData** method assigns data to be sent as the request body.

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::SetReadBlockSize

Call this method to assign the size of the incoming response buffer.

```
DWORD SetReadBlockSize(
   DWORD dwBlockSize
) throw( );
```

Parameters

dwBlockSize

The size of the response buffer, in bytes.

Return Value

Returns the previous size of the response buffer.

Remarks

The default read block size is defined by the ATL_HTTP_DEFAULT_BLOCK_SIZE macro.

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::SetReadStatusCallback

Call this method to assign a response processing callback function.

```
PFNATLSTATUSCALLBACK SetReadStatusCallback(
   PFNATLSTATUSCALLBACK pfn,
   DWORD_PTR dwData
) throw( );
```

Parameters

pfn

A pointer to the response processing callback function, as defined by the PFNATLSTATUSCALLBACK **typedef**.

dwData

Context data that is to be passed to the callback function each time it is invoked.

Return Value

Returns a pointer to a previously assigned callback function, or **NULL** if no callback was assigned.

Remarks

If a read status callback is installed, the CAtlHttpClientT class will call the provided function whenever new response data arrives. This mechanism allows for the display of partial responses (such as images that are displayed incrementally) and progress bars.

Example

See the HttpClient Sample.

See Also

Reference

CAtlNavigateData Class

CAtlNavigateData::GetReadStatusCallback

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::SetSendBlockSize

Call this method to assign the size of the outgoing request buffer.

```
DWORD SetSendBlockSize(
   DWORD dwBlockSize
) throw( );
```

Parameters

dwBlockSize

The size of the request buffer, in bytes.

Return Value

The previous size of the response buffer.

Remarks

By default, the buffer size is ATL_HTTP_DEFAULT_BLOCK_SIZE.

See Also

Reference

CAtlNavigateData Class

CAtlNavigateData::GetSendBlockSize

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::SetSendStatusCallback

Call this method to assign a request status callback function.

```
PFNATLSTATUSCALLBACK SetSendStatusCallback(
   PFNATLSTATUSCALLBACK pfn,
   DWORD_PTR dwData
) throw( );
```

Parameters

pfn

A pointer to a callback to be called after each block of request data is sent.

dwData

A context value to be passed to the callback function each time it is invoked.

Return Value

A pointer to a previously assigned callback function, or **NULL** if no callback was assigned.

Remarks

Send status callbacks are useful for processing large requests, particularly for updating a progress bar.

See Also

Reference

CAtlNavigateData Class

CAtlNavigateData::GetSendStatusCallback

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlNavigateData::SetSocketTimeout

Call this method to set the socket timeout value.

```
DWORD SetSocketTimeout(
   DWORD dwNewTimeout
) throw( );
```

Parameters

dwNewTimeout

The new timeout in milliseconds.

Return Value

The previous timeout value. The default socket timeout is determined by ATL_SOCK_TIMEOUT.

Remarks

The socket timeout value determines how long the underlying socket implementation is to wait for server responses before abandoning each operation.

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

CAtlNavigateData Members

HTTP Client Reference

CAtlRECharTraitsA Class

This class represents an ANSI character traits object.

```
class CAtlRECharTraitsA
```

Remarks

This class is passed as a template parameter to CAtlRegExp and CAtlREMatchContext.

CAtlRECharTraitsA contains the traits for ANSI characters. For multibyte characters, see CAtlRECharTraitsMB; for Unicode, CAtlRECharTraitsW. You can also define your own custom character traits classes.

Example

See the Input Sample.

Requirements

Header: atlrx.h

See Also

Other Resources

CAtlRECharTraitsA Members

ATL Server Classes

CAtlRECharTraitsA Members

Static Functions

ByteLen	This function is called to retrieve the length of a string in bytes.
GetAbbrevs	This function is called to get the abbreviations from an ANSI character traits object.
Isdigit	This function is called to determine if an ANSI character is a digit.
Next	This function is called to get the next character in an ANSI string.
Strlwr	This function is called to convert an ANSI string to lowercase.
Strncmp	This function is called to compare two ANSI strings.
Strnicmp	This function is called to make a case-insensitive comparison of two ANSI strings.
Strtol	This function is called to convert an ANSI string to a long integer.
UseBitFieldForRange	This function is called to determine the representation of character ranges.

Typedefs

RECHARTYPE	The character type.

See Also

Reference

CAtlRECharTraitsA Class

CAtlRECharTraitsA Static Functions

For information about the static functions in **CAtlRECharTraitsA**, see CAtlRECharTraitsA Members.

CAtlRECharTraitsA::ByteLen

This function is called to retrieve the length of a string in bytes.

```
static int ByteLen(
   const RECHARTYPE *sz
) throw();
```

Parameters

sz

The string.

See Also

Reference

CAtlRECharTraitsA Class

Other Resources

CAtlRECharTraitsA Members

CAtlRECharTraitsA::GetAbbrevs

This function is called to get the abbreviations from an ANSI character traits object.

```
static const RECHARTYPE** GetAbbrevs( );
```

Return Value

Returns the abbreviations.

Remarks

Abbreviations are a convenient way of writing common regular expressions; for example, you can write `\d` instead of `[0-9]`. See the table of abbreviations in CAtlRegExp.

CAtlRegExp calls this function to get a list of abbreviations and their definitions.

See Also

Reference

CAtlRECharTraitsA Class

Other Resources

CAtlRECharTraitsA Members

CAtlRECharTraitsA::Isdigit

This function is called to determine if an ANSI character is a digit.

```
static int Isdigit(
   RECHARTYPE ch
) throw( );
```

Parameters

ch

The character.

Return Value

Returns a nonzero value if the character is a digit, otherwise returns 0.

See Also

Reference

CAtlRECharTraitsA Class

Other Resources

CAtlRECharTraitsA Members

CAtlRECharTraitsA::Next

This function is called to get the next character in an ANSI string.

```
static RECHARTYPE * Next(
   const RECHARTYPE * sz
) throw( );
```

Parameters

sz

Points to the current character.

Return Value

Returns a pointer to the next character.

See Also

Reference

CAtlRECharTraitsA Class

Other Resources

CAtlRECharTraitsA Members

CAtlRECharTraitsA::Strlwr

This function is called to convert an ANSI string to lowercase.

```
static RECHARTYPE * Strlwr(
   RECHARTYPE * sz
) throw( );
```

Parameters

sz

The string.

Return Value

Returns a pointer to the string (conversion is done in place).

See Also

Reference

CAtlRECharTraitsA Class

Other Resources

CAtlRECharTraitsA Members

CAtlRECharTraitsA::Strncmp

This function is called to make a case-sensitive comparison of two ANSI strings.

```
static int Strncmp(
   const RECHARTYPE * szLeft,
   const RECHARTYPE * szRight,
   size_t nCount
) throw( );
```

Parameters

szLeft

The first string.

szRight

The second string.

nCount

The number of characters to compare.

Return Value

Returns 0 if the strings are identical; returns a negative value if the first string is less than the second, a positive value if the first string is greater than the second.

See Also

Reference

CAtlRECharTraitsA Class

Other Resources

CAtlRECharTraitsA Members

CAtlRECharTraitsA::Strnicmp

This function is called to make a case-insensitive comparison of two ANSI strings.

```
static int Strnicmp(
   const RECHARTYPE * szLeft,
   const RECHARTYPE * szRight,
   size_t nCount
) throw( );
```

Parameters

szLeft

The first string.

szRight

The second string.

nCount

The number of characters to compare.

Return Value

Returns 0 if the strings are identical; returns a negative value if the first string is less than the second, a positive value if the first string is greater than the second.

See Also

Reference

CAtlRECharTraitsA Class

Other Resources

CAtlRECharTraitsA Members

CAtlRECharTraitsA::Strtol

This function is called to convert an ANSI string to a long integer.

```
static long Strtol(
   const RECHARTYPE * sz,
   RECHARTYPE ** szEnd,
   int nBase
) throw( );
```

Parameters

sz

The string to convert.

szEnd

After calling **Strtol**, *szEnd* points to the unconverted portion of the string.

nBase

The number base to use.

Return Value

Returns the converted value.

See Also

Reference

CAtlRECharTraitsA Class

Other Resources

CAtlRECharTraitsA Members

CAtlRECharTraitsA::UseBitFieldForRange

This function is called to determine the representation of character ranges.

```
static BOOL UseBitFieldForRange( ) throw( );
```

Return Value

Returns TRUE.

Remarks

Because this function returns TRUE, CAtlRegExp will use a bit field to represent character ranges in ANSI regular expressions (such as `[a-z]`). The bit field representation is more time-efficient but uses more memory.

See Also

Reference

CAtlRECharTraitsA Class

Other Resources

CAtlRECharTraitsA Members

CAtlRECharTraitsA Typedefs

For information about the typedefs in **CAtlRECharTraitsA**, see CAtlRECharTraitsA Members.

CAtlRECharTraitsA::RECHARTYPE

The character type.

```
typedef char RECHARTYPE;
```

Remarks

This is the type of characters described by this traits class.

See Also

Reference

CAtlRECharTraitsA Class

Other Resources

CAtlRECharTraitsA Members

CAtlRECharTraitsMB Class

This class represents a multibyte character traits object.

```
class CAtlRECharTraitsMB
```

Remarks

This class is passed as a template parameter to CAtlRegExp and CAtlREMatchContext.

CAtlRECharTraitsMB contains the traits for multibyte characters (MBCS). For ANSI characters, see CAtlRECharTraitsA; for Unicode, CAtlRECharTraitsW. You can also define your own custom character traits classes.

Requirements

Header: atlrx.h

See Also

Other Resources

CAtlRECharTraitsMB Members

ATL Server Classes

CAtlRECharTraitsMB Members

Static Functions

ByteLen	This function is called to retrieve the length of a string in bytes.
GetAbbrevs	This function is called to get the abbreviations from a multibyte character traits object.
Isdigit	This function is called to determine if a multibyte character is a digit.
Next	This function is called to get the next character in an MBCS string.
Strlwr	This function is called to convert an MBCS string to lowercase.
Strncmp	This function is called to compare two MBCS strings.
Strnicmp	This function is called to make a case-insensitive comparison of two MBCS strings.
Strtol	This function is called to convert an MBCS string to a long integer.
UseBitFieldForRange	This function is called to determine the representation of character ranges.

Typedefs

RECHARTYPE	The character type.

See Also

Reference

CAtlRECharTraitsMB Class

CAtlRECharTraitsMB Static Functions

For information about the static functions in **CAtlRECharTraitsMB**, see CAtlRECharTraitsMB Members.

CAtlRECharTraitsMB::ByteLen

This function is called to retrieve the length of a string in bytes.

```
static int ByteLen(
   const RECHARTYPE *sz
) throw();
```

Parameters

sz

The string.

See Also

Reference

CAtlRECharTraitsMB Class

Other Resources

CAtlRECharTraitsMB Members

CAtlRECharTraitsMB::GetAbbrevs

This function is called to get the abbreviations from a multibyte character traits object.

```
static const RECHARTYPE** GetAbbrevs( );
```

Return Value

Returns the abbreviations.

Remarks

Abbreviations are a convenient way of writing common regular expressions; for example, you can write `\d` instead of `[0-9]`. See the table of abbreviations in CAtlRegExp.

CAtlRegExp calls this function to get a list of abbreviations and their definitions.

See Also

Reference

CAtlRECharTraitsMB Class

Other Resources

CAtlRECharTraitsMB Members

CAtlRECharTraitsMB::Isdigit

This function is called to determine if a multibyte character is a digit.

```
static int Isdigit(
   RECHARTYPE ch
) throw( );
```

Parameters

ch

The character.

Return Value

Returns a nonzero value if the character is a digit, otherwise returns 0.

See Also

Reference

CAtlRECharTraitsMB Class

Other Resources

CAtlRECharTraitsMB Members

CAtlRECharTraitsMB::Next

This function is called to get the next character in an MBCS string.

```
static RECHARTYPE * Next(
   const RECHARTYPE * sz
) throw( );
```

Parameters

sz

Points to the current character.

Return Value

Returns a pointer to the next character.

See Also

Reference

CAtlRECharTraitsMB Class

Other Resources

CAtlRECharTraitsMB Members

CAtlRECharTraitsMB::Strlwr

This function is called to convert an MBCS string to lowercase.

```
static RECHARTYPE * Strlwr(
   RECHARTYPE * sz
) throw( );
```

Parameters

sz

The string.

Return Value

Returns a pointer to the string (conversion is done in place).

See Also

Reference

CAtlRECharTraitsMB Class

Other Resources

CAtlRECharTraitsMB Members

CAtlRECharTraitsMB::Strncmp

This function is called to make a case-sensitive comparison of two MBCS strings.

```
static int Strncmp(
   const RECHARTYPE * szLeft,
   const RECHARTYPE * szRight,
   size_t nCount
) throw( );
```

Parameters

szLeft

The first string.

szRight

The second string.

nCount

The number of characters to compare.

Return Value

Returns 0 if the strings are identical; returns a negative value if the first string is less than the second, a positive value if the first string is greater than the second.

See Also

Reference

CAtlRECharTraitsMB Class

Other Resources

CAtlRECharTraitsMB Members

CAtlRECharTraitsMB::Strnicmp

This function is called to make a case-insensitive comparison of two MBSC strings.

```
static int Strnicmp(
   const RECHARTYPE * szLeft,
   const RECHARTYPE * szRight,
   size_t nCount
) throw( );
```

Parameters

szLeft

The first string.

szRight

The second string.

nCount

The number of characters to compare.

Return Value

Returns 0 if the strings are identical; returns a negative value if the first string is less than the second, a positive value if the first string is greater than the second.

See Also

Reference

CAtlRECharTraitsMB Class

Other Resources

CAtlRECharTraitsMB Members

CAtlRECharTraitsMB::Strtol

This function is called to convert an MBCS string to a long integer.

```
static long Strtol(
      const RECHARTYPE * sz,
   RECHARTYPE ** szEnd,
   int nBase
) throw( );
```

Parameters

sz

The string to convert.

szEnd

After calling **Strtol**, *szEnd* points to the unconverted portion of the string.

nBase

The number base to use.

Return Value

Returns the converted value.

See Also

Reference

CAtlRECharTraitsMB Class

Other Resources

CAtlRECharTraitsMB Members

CAtlRECharTraitsMB::UseBitFieldForRange

This function is called to determine the representation of character ranges.

```
static BOOL UseBitFieldForRange( ) throw( );
```

Return Value

Returns FALSE.

Remarks

Because this function returns FALSE, CAtlRegExp will not use a bit field to represent character ranges in MBCS regular expressions (such as `[a-z]`). The bit field representation would be more time-efficient but would use more memory.

See Also

Reference

CAtlRECharTraitsMB Class

Other Resources

CAtlRECharTraitsMB Members

CAtlRECharTraitsMB Typedefs

For information about the typedefs in **CAtlRECharTraitsMB**, see CAtlRECharTraitsMB Members.

CAtlRECharTraitsMB::RECHARTYPE

The character type.

```
typedef unsigned char RECHARTYPE;
```

Remarks

This is the type of characters described by this traits class.

See Also

Reference

CAtlRECharTraitsMB Class

Other Resources

CAtlRECharTraitsMB Members

CAtlRECharTraitsW Class

This class represents a Unicode character traits object.

```
class CAtlRECharTraitsW
```

Remarks

This class is passed as a template parameter to CAtlRegExp and CAtlREMatchContext.

CAtlRECharTraitsW contains the traits for Unicode characters. For ANSI characters, see CAtlRECharTraitsA; for MBCS, CAtlRECharTraitsMB. You can also define your own custom character traits classes.

Requirements

Header: atlrx.h

See Also

Other Resources

CAtlRECharTraitsW Members

ATL Server Classes

CAtlRECharTraitsW Members

Static Functions

ByteLen	This function is called to retrieve the length of a string in bytes.
GetAbbrevs	This function is called to get the abbreviations from a Unicode character traits object.
Isdigit	This function is called to determine if a Unicode character is a digit.
Next	This function is called to get the next character in a Unicode string.
Strlwr	This function is called to convert a Unicode string to lowercase.
Strncmp	This function is called to compare two Unicode strings.
Strnicmp	This function is called to make a case-insensitive comparison of two Unicode strings.
Strtol	This function is called to convert a Unicode string to a long integer.
UseBitFieldForRange	This function is called to determine the representation of character ranges.

Typedefs

RECHARTYPE	The character type.

See Also

Reference

CAtlRECharTraitsW Class

CAtlRECharTraitsW Static Functions

For information about the static functions in **CAtlRECharTraitsW**, see CAtlRECharTraitsW Members.

CAtlRECharTraitsW::ByteLen

This function is called to retrieve the length of a string in bytes.

```
static int ByteLen(
   const RECHARTYPE *sz
) throw();
```

Parameters

sz

The string.

See Also

Reference

CAtlRECharTraitsW Class

Other Resources

CAtlRECharTraitsW Members

CAtlRECharTraitsW::GetAbbrevs

This function is called to get the abbreviations from a Unicode character traits object.

```
static const RECHARTYPE** GetAbbrevs( );
```

Return Value

Returns the abbreviations.

Remarks

Abbreviations are a convenient way of writing common regular expressions; for example, you can write `\d` instead of `[0-9]`. See the table of abbreviations in CAtlRegExp.

CAtlRegExp calls this function to get a list of abbreviations and their definitions.

See Also

Reference

CAtlRECharTraitsW Class

Other Resources

CAtlRECharTraitsW Members

CAtlRECharTraitsW::Isdigit

This function is called to determine if a Unicode character is a digit.

```
static int Isdigit(
   RECHARTYPE ch
) throw( );
```

Parameters

ch

The character.

Return Value

Returns a nonzero value if the character is a digit, otherwise returns 0.

See Also

Reference

CAtlRECharTraitsW Class

Other Resources

CAtlRECharTraitsW Members

CAtlRECharTraitsW::Next

This function is called to get the next character in a Unicode string.

```
static RECHARTYPE* Next(
   const RECHARTYPE* sz
) throw( );
```

Parameters

sz

Points to the current character.

Return Value

Returns a pointer to the next character.

See Also

Reference

CAtlRECharTraitsW Class

Other Resources

CAtlRECharTraitsW Members

CAtlRECharTraitsW::Strlwr

This function is called to convert a Unicode string to lowercase.

```
static RECHARTYPE* Strlwr(
   RECHARTYPE* sz
) throw( );
```

Parameters

sz

The string.

Return Value

Returns a pointer to the string (conversion is done in place).

See Also

Reference

CAtlRECharTraitsW Class

Other Resources

CAtlRECharTraitsW Members

CAtlRECharTraitsW::Strncmp

This function is called to make a case-sensitive comparison of two Unicode strings.

```
static int Strncmp(
   const RECHARTYPE * szLeft,
   const RECHARTYPE * szRight,
   size_t nCount
) throw( );
```

Parameters

szLeft

The first string.

szRight

The second string.

nCount

The number of characters to compare.

Return Value

Returns 0 if the strings are identical; returns a negative value if the first string is less than the second, a positive value if the first string is greater than the second.

See Also

Reference

CAtlRECharTraitsW Class

Other Resources

CAtlRECharTraitsW Members

CAtlRECharTraitsW::Strnicmp

This function is called to make a case-insensitive comparison of two Unicode strings.

```
static int Strnicmp(
   const RECHARTYPE * szLeft,
   const RECHARTYPE * szRight,
   size_t nCount
) throw( );
```

Parameters

szLeft

The first string.

szRight

The second string.

nCount

The number of characters to compare.

Return Value

Returns 0 if the strings are identical; returns a negative value if the first string is less than the second, a positive value if the first string is greater than the second.

See Also

Reference

CAtlRECharTraitsW Class

Other Resources

CAtlRECharTraitsW Members

CAtlRECharTraitsW::Strtol

This function is called to convert a Unicode string to a long integer.

```
static long Strtol(
   const RECHARTYPE * sz,
   RECHARTYPE ** szEnd,
   int nBase
) throw( );
```

Parameters

sz

The string to convert.

szEnd

After calling **Strtol**, *szEnd* points to the unconverted portion of the string.

nBase

The number base to use.

Return Value

Returns the converted value.

See Also

Reference

CAtlRECharTraitsW Class

Other Resources

CAtlRECharTraitsW Members

CAtlRECharTraitsW::UseBitFieldForRange

This function is called to determine the representation of character ranges.

```
static BOOL UseBitFieldForRange( ) throw( );
```

Return Value

Returns FALSE.

Remarks

Because this function returns FALSE, CAtlRegExp will not use a bit field to represent character ranges in Unicode regular expressions (such as `[a-z]`). The bit field representation would be more time-efficient but would use more memory.

See Also

Reference

CAtlRECharTraitsW Class

Other Resources

CAtlRECharTraitsW Members

CAtlRECharTraitsW Typedefs

For information about the typedefs in **CAtlRECharTraitsW**, see CAtlRECharTraitsW Members.

CAtlRECharTraitsW::RECHARTYPE

The character type.

```
typedef WCHAR RECHARTYPE;
```

Remarks

This is the type of characters described by this traits class.

See Also

Reference

CAtlRECharTraitsW Class

Other Resources

CAtlRECharTraitsW Members

CAtlRegExp Class

This class represents a regular expression.

```
template <
   class CharTraits = CAtlRECharTraits
>
class CAtlRegExp
```

Parameters

CharTraits

A character traits object. For an example, see CAtlRECharTraitsA.

Remarks

To set the regular expression, call Parse:

```
CAtlRegExp<> re;
re.Parse( "{[0-9]?[0-9]}:{[0-9][0-9]}" ); // Time in h:mm or hh:mm format
```

Parse converts the regular expression into a program for the **CAtlRegExp** internal pattern-matching automaton.

To match the regular expression against a string, call Match:

```
re.Match( "1:57", &mc );  // Returns TRUE: successful match
re.Match( "01/03", &mc ); // Returns FALSE: no match
```

The arguments to **Match** are the string to match and a CAtlREMatchContext object. In the previous regular expression, there are two match groups delimited by braces. If the regular expression matches an input string, the **CAtlREMatchContext** object can be used to extract the actual match group text (in this case, the hour and the minute) from the input. For more information, see CAtlReMatchContext.

Match has an optional third parameter. If it is present, **Match** sets it to point just beyond the last character matched in the string. This allows you to continue matching in the string from that point on.

Regular Expression Syntax

This table lists the metacharacters understood by **CAtlRegExp**.

Metacharacter	Meaning
.	Matches any single character.
[ ]	Indicates a character class. Matches any character inside the brackets (for example, `[abc]` matches "a", "b", and "c").
^	If this metacharacter occurs at the start of a character class, it negates the character class. A negated character class matches any character except those inside the brackets (for example, `[^abc]` matches all characters except "a", "b", and "c"). If ^ is at the beginning of the regular expression, it matches the beginning of the input (for example, `^[abc]` will only match input that begins with "a", "b", or "c").
-	In a character class, indicates a range of characters (for example, `[0-9]` matches any of the digits "0" through "9").
?	Indicates that the preceding expression is optional: it matches once or not at all (for example, `[0-9][0-9]?` matches "2" and "12").

+	Indicates that the preceding expression matches one or more times (for example, `[0-9]+` matches "1", "13", "456", and so on).
*	Indicates that the preceding expression matches zero or more times.
??, +?, *?	Non-greedy versions of `?`, `+`, and `*`. These match as little as possible, unlike the greedy versions that match as much as possible (for example, given the input "<abc><def>", `<.*?>` matches "<abc>" while `<.*>` matches "<abc><def>").
( )	Grouping operator. Example: `(\d+,)*\d+` matches a list of numbers separated by commas (for example, "1" or "1,23,456").
{ }	Indicates a match group. The actual text in the input that matches the expression inside the braces can be retrieved through the CAtlREMatchContext object.
\	Escape character: interpret the next character literally (for example, `[0-9]+` matches one or more digits, but `[0-9]\+` matches a digit followed by a plus character). Also used for abbreviations (such as `\a` for any alphanumeric character; see the following table). If `\` is followed by a number *n*, it matches the *n*th match group (starting from 0). Example: `<{.*?}>.*?</\0>` matches "<head>Contents</head>". Note that, in C++ string literals, two backslashes must be used: `"\\+"`, `"\\a"`, `"<{.*?}>.*?</\\0>"`.
$	At the end of a regular expression, this character matches the end of the input (for example, `[0-9]$` matches a digit at the end of the input).
|	Alternation operator: separates two expressions, exactly one of which matches (for example, `T|the` matches "The" or "the").
!	Negation operator: the expression following `!` does not match the input (for example, `a!b` matches "a" not followed by "b").

Abbreviations

CAtlRegExp can handle abbreviations, such as `\d` instead of `[0-9]`. The abbreviations are provided by the character traits class passed in the *CharTraits* parameter. The predefined character traits classes provide the following abbreviations.

Abbreviation	Matches
\a	Any alphanumeric character: ([a-zA-Z0-9])
\b	White space (blank): ([\\t])
\c	Any alphabetic character: ([a-zA-Z])
\d	Any decimal digit: ([0-9])
\h	Any hexadecimal digit: ([0-9a-fA-F])
\n	Newline: (\r|(\r?\n))
\q	A quoted string: (\"[^\"]*\")|(\'[^\']*\')
\w	A simple word: ([a-zA-Z]+)
\z	An integer: ([0-9]+)

Example

The following program uses a regular expression to extract parts of a URL.

```
// catlregexp_class.cpp
#include <afx.h>
#include <atlrx.h>

int main(int argc, char* argv[])
{
    CAtlRegExp<> reUrl;
    // Five match groups: scheme, authority, path, query, fragment
    REParseError status = reUrl.Parse(
        "({[^:/?#]+}:)?(//{[^/?#]*})?{[^?#]*}(?{[^#]*})?(#{.*})?" );

    if (REPARSE_ERROR_OK != status)
    {
        // Unexpected error.
        return 0;
    }

    CAtlREMatchContext<> mcUrl;
    if (!reUrl.Match(
"http://search.microsoft.com/us/Search.asp?qu=atl&boolean=ALL#results",
        &mcUrl))
    {
        // Unexpected error.
        return 0;
    }

    for (UINT nGroupIndex = 0; nGroupIndex < mcUrl.m_uNumGroups;
         ++nGroupIndex)
    {
        const CAtlREMatchContext<>::RECHAR* szStart = 0;
        const CAtlREMatchContext<>::RECHAR* szEnd = 0;
        mcUrl.GetMatch(nGroupIndex, &szStart, &szEnd);

        ptrdiff_t nLength = szEnd - szStart;
        printf_s("%d: \"%.*s\"\n", nGroupIndex, nLength, szStart);
    }

    return 0;
}
```

Output

```
0: "http"
1: "search.microsoft.com"
2: "/us/Search.asp"
3: "qu=atl&boolean=ALL"
4: "results"
```

Requirements

Class	Required header	Compatibility
CAtlRegExp	<atlrx.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003, Windows Server 2003

See Also

Tasks

RegExp Sample: Demonstrates ATL Regular Expression Classes

Input Sample: Demonstrates User Input Validation On Client And Server

ISAPIFilter Sample: Maps URLs to Parameterized Queries

Other Resources

CAtlRegExp Members

ATL Server Classes

CAtlRegExp Members

Methods

CAtlRegExp	The constructor.
Dump	Call this method to dump a regular expression object.
Match	Call this method to match a string.
Parse	Call this method to parse a regular expression.

Typedefs

RECHAR	The regular expression character type.

See Also

Reference

CAtlRegExp Class

CAtlRegExp Class Methods

For information about the methods in **CAtlRegExp**, see CAtlRegExp Members.

CAtlRegExp::CAtlRegExp

The constructor.

```
CAtlRegExp( ) throw( );
```

See Also

Reference

CAtlRegExp Class

Other Resources

CAtlRegExp Members

CAtlRegExp::Dump

Call this method to dump a regular expression object.

```
void Dump( );
```

Remarks

This method prints out the program of the regular expression automaton for debugging.

See Also

Reference

CAtlRegExp Class

Other Resources

CAtlRegExp Members

CAtlRegExp::Match

Call this method to match a string.

```
BOOL Match(
   const RECHAR* szIn,
   CAtlREMatchContext< CharTraits >* pContext,
   const RECHAR** ppszEnd = NULL
);
```

Parameters

szIn

The input string.

pContext

The match context object.

ppszEnd

If present, is set to the beginning of the unvisited portion of the string after matching. This allows you to continue matching in the same string.

Return Value

Returns TRUE on success, FALSE on failure.

Example

See the RegExp Sample, the ISAPIFilter Sample, and the Input Sample.

See Also

Reference

CAtlRegExp Class

Other Resources

CAtlRegExp Members

CAtlRegExp::Parse

Call this method to parse a regular expression.

```
REParseError Parse(
   const RECHAR* szRE,
   BOOL bCaseSensitive = TRUE
);
```

Parameters

szRE

The regular expression.

bCaseSensitive

TRUE if the regular expression is case sensitive, FALSE otherwise.

Return Value

A member of REParseError indicating success or failure of the parse operation.

Remarks

This method converts a regular expression into a program for a pattern-matching automaton. To match a regular expression against text, call Match.

Example

See the RegExp Sample, the ISAPIFilter Sample, and the Input Sample.

See Also

Reference

CAtlRegExp Class

Other Resources

CAtlRegExp Members

CAtlRegExp Class Typedefs

For information about the typedefs in **CAtlRegExp**, see CAtlRegExp Members.

CAtlRegExp::RECHAR

The regular expression character type.

```
typedef CharTraits::RECHARTYPE RECHAR;
```

Remarks

RECHAR is a typedef for the character type of the character traits class passed as a template parameter to **CAtlRegExp**.

See Also

Reference

CAtlRegExp Class

Other Resources

CAtlRegExp Members

CAtlValidator Class

This class provides a static function for determining whether a value lies within a particular range.

```
class CAtlValidator
```

Remarks

This class is used as a default template argument by CValidateObject.

Requirements

Header: atlisapi.h

See Also

Other Resources

CAtlValidator Members

CAtlValidator Members

Static Functions

Validate	Call this function to validate a value against a minimum and maximum value or length.

See Also

Reference

CAtlValidator Class

CAtlValidator Static Functions

For information about the static functions in **CAtlValidator**, see CAtlValidator Members.

CAtlValidator::Validate

Call this function to validate a value against a minimum and maximum value or length.

```
template <class T, class TCompType>
static DWORD Validate(
      T value,
      TCompType nMinValue,
      TCompType nMaxValue
) throw( );
static DWORD Validate(
   double dblValue,
   double dblMinValue,
   double dblMaxValue
) throw( );
static DWORD Validate(
   LPCSTR pszValue,
   int nMinChars,
   int nMaxChars
) throw( );
```

Parameters

T

The type of the value being validated.

TCompType

The type of the minimum and maximum values against which the value is to be validated.

value, *dblValue*, *pszValue*

The value being validated.

nMinValue, *dblMinValue*

The minimum acceptable value.

nMaxValue, *dblMaxValue*

The maximum acceptable value.

nMinChars

The minimum acceptable length.

nMaxChars

The maximum acceptable length.

Return Value

Returns one of the validation results describing whether validation succeeded or how it failed.

Remarks

The function template casts *nMinValue* and *nMaxValue* to type *T* before comparing the values using less than (<) and greater than (>) operators.

The double specialization uses ATL_EPSILON to perform the comparison. You can redefine this value to get an acceptable tolerance on the validation.

See Also

Reference

CAtlValidator Class

Other Resources

CAtlValidator Members

CAtlREMatchContext Class

This class represents a match context.

```
template <
   class CharTraits = CAtlRECharTraits
>
class CAtlREMatchContext
```

Parameters

CharTraits

A character traits object. See, for instance, CAtlRECharTraitsA.

Remarks

This class is used in conjunction with CAtlRegExp.

A **CAtlREMatchContext** object contains information about the "match groups" in a regular expression. Match groups are indicated by braces, and allow you to extract the actual text that matches the regular expression. For example, this regular expression has two match groups:

```
{[0-9]+}-{[0-9]+}
```

This regular expression would match input such as "100-1234", and the two match groups would be "100" and "1234", respectively.

A **CAtlREMatchContext** object is passed in the call to CAtlRegExp::Match:

```
CAtlRegExp<> re;                // regular expression object
re.Parse( "{[0-9]+}-{[0-9]+}" ); // set up regular expression
CAtlREMatchContext<> mc;        // match context object
re.Match( "100-1234", &mc );    // 2nd argument is match context
```

After the call to **Match**, data member m_uNumGroups contains the number of match groups (in this case, 2) and calling GetMatch returns pointers to the beginning and end of each match group in the input:

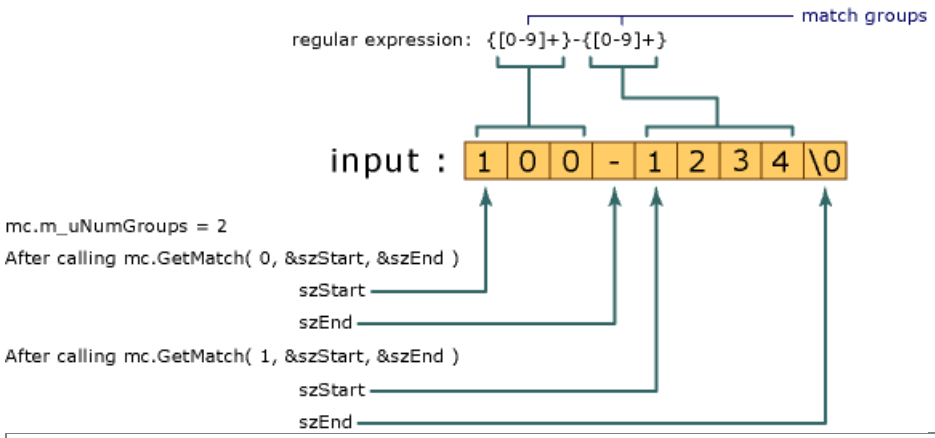

Note

The pointers point into the original input string and thus are only valid during the lifetime of the input string.

Example

See the example in CAtlRegExp.

Requirements

Header: atlrx.h

See Also

Tasks

RegExp Sample: Demonstrates ATL Regular Expression Classes

Input Sample: Demonstrates User Input Validation On Client And Server

ISAPIFilter Sample: Maps URLs to Parameterized Queries

Other Resources

CAtlREMatchContext Members

ATL Server Classes

CAtlREMatchContext Members

Methods

CAtlREMatchContext	The constructor.
GetMatch	Gets a match group.

Data Members

m_Match	Indicates the portion of the string that was matched.
m_uNumGroups	The number of match groups.

Typedefs

RECHAR	The regular expression character type.

Structures

MatchGroup	Match group information.

See Also

Reference

CAtlREMatchContext Class

CAtlREMatchContext Methods

For information about the methods in **CAtlREMatchContext**, see CAtlREMatchContext Members.

CAtlREMatchContext::CAtlREMatchContext

The constructor.

```
CAtlREMatchContext(
   size_t nInitStackSize = ATL_REGEXP_MIN_STACK
);
```

Parameters

nInitStackSize

The initial stack size.

See Also

Reference

CAtlREMatchContext Class

Other Resources

CAtlREMatchContext Members

CAtlREMatchContext::GetMatch

Gets a match group.

```
void GetMatch(
   int nIndex,
   const RECHAR **szStart,
   const RECHAR **szEnd
);
void GetMatch(
   int nIndex,
   MatchGroup *pGroup
);
```

Parameters

nIndex

Zero based index of the match group to get.

szStart

Pointer that will be set to the beginning of the match group.

szEnd

Pointer that will be set to the end of the match group.

pGroup

A MatchGroup structure containing the start and end pointers.

Remarks

After calling **GetMatch**, *szStart* points to the start of the match group specified by *nIndex*, and *szEnd* points one character beyond the end of the match group.

Note
These pointers point into the original input string and thus are only valid during the lifetime of the input string.

Example

See the RegExp Sample and the ISAPIFilter Sample.

See Also

Reference

CAtlREMatchContext Class

Other Resources

CAtlREMatchContext Members

CAtlREMatchContext Data Members

For information about the data members in **CAtlREMatchContext**, see CAtlREMatchContext Members.

CAtlREMatchContext::m_Match

Indicates the portion of the string that was matched.

```
MatchGroup m_Match;
```

Remarks

After a CAtlRegExp::Match operation, this MatchGroup indicates how much of the input was scanned. `m_Match.szStart` points to the start of the scanned portion; `m_Match.szEnd` points one character beyond the end of the scanned portion of the string.

Note
The pointers in **m_Match** point into the original input string and thus are only valid during the lifetime of the input string.

Example

See the RegExp Sample.

See Also

Reference

CAtlREMatchContext Class

Other Resources

CAtlREMatchContext Members

CAtlREMatchContext::m_uNumGroups

The number of match groups.

```
UINT m_nNumGroups;
```

Example

See the RegExp Sample.

See Also

Reference

CAtlREMatchContext Class

CAtlREMatchContext::GetMatch

Other Resources

CAtlREMatchContext Members

CAtlREMatchContext Typedefs

For information about the typedefs in **CAtlREMatchContext**, see CAtlREMatchContext Members.

CAtlREMatchContext::RECHAR

The regular expression character type.

```
typedef CharTraits::RECHARTYPE RECHAR;
```

Remarks

RECHAR is a typedef for the character type of the character traits class passed as a template parameter to **CAtlREMatchContext**.

See Also

Reference

CAtlREMatchContext Class

Other Resources

CAtlREMatchContext Members

CAtlREMatchContext Structures

For information about the structures in **CAtlREMatchContext**, see CAtlREMatchContext Members.

CAtlREMatchContext::MatchGroup Structure

This structure contains match group information.

```
struct MatchGroup;
```

Remarks

This structure contains two pointers; one points to the beginning of a match group, the other points one character beyond the end of a match group.

Example

See the RegExp Sample and the ISAPIFilter Sample.

See Also

Reference

CAtlREMatchContext::GetMatch
CAtlREMatchContext::m_Match

Other Resources

CAtlREMatchContext::MatchGroup Members
ATL Server Classes

CAtlREMatchContext::MatchGroup Members

Data Members

szEnd	The end of the match group.
szStart	The start of the match group.

See Also

Reference

CAtlREMatchContext::MatchGroup Structure

CAtlREMatchContext::MatchGroup Data Members

For information about the data members in **CAtlREMatchContext::MatchGroup**, see CAtlREMatchContext::MatchGroup Members.

CAtlREMatchContext::MatchGroup::szEnd

The end of the match group.

```
const RECHAR *szEnd;
```

Remarks

szEnd points one character beyond the end of the match group.

Example

See the RegExp Sample.

See Also

Reference

CAtlREMatchContext::MatchGroup Structure

CAtlREMatchContext::MatchGroup::szStart

Other Resources

CAtlREMatchContext::MatchGroup Members

CAtlREMatchContext::MatchGroup::szStart

The start of the match group.

```
const RECHAR *szStart;
```

Remarks

szStart points to the start of the match group.

See Also

Reference

CAtlREMatchContext::MatchGroup Structure

CAtlREMatchContext::MatchGroup::szEnd

Other Resources

CAtlREMatchContext::MatchGroup Members

CBlobCache Class

This class provides functionality for caching arbitrarily sized data blocks in memory.

```
template <
   class MonitorClass,
   class StatClass  = CStdStatClass,
   class SyncObj    = CComCriticalSection,
   class FlushClass = COldFlusher,
   class CullClass  = CExpireCuller
>
class CBlobCache :
   public CMemoryCache< void* , StatClass, FlushClass, CFixedStringKey,
         CStringElementTraits< CFixedStringKey >, SyncObj, CullClass >,
   public IMemoryCache,
   public IMemoryCacheControl,
   public IMemoryCacheStats,
   public IWorkerThreadClient
```

Parameters

MonitorClass

The class to be used for periodic cache maintenance. Normally, this will be CWorkerThread. If no cache maintenance is desired, use CNoWorkerThread.

StatClass

The class used to collect and expose statistics related to the use of the cache, such as effectiveness and memory used. This must conform to the cache statistics archetype. By default the CStdStatClass is used, which provides standard cache statistics. If no statistics are desired, use CNoStatClass.

SyncObj

The class to be used for thread synchronization. The default value of CComCriticalSection is suitable for most situations, but the CComFakeCriticalSection class can be used to avoid thread synchronization overhead in single-threaded environments.

FlushClass

The class used to determine whether, and in what order, items should be removed from the cache using the number and sequence of accesses as the basis for making the decision. This must conform to the cache flusher archetype.

This class is used to remove items from the cache to comply with the size and memory constraints specified through the IMemoryCacheControl interface. Use the CNoFlusher class if no automatic flushing is to be performed.

CullClass

The class used to determine to determine whether, and in what order, items should be removed from the cache using time-based criteria. Must conform to the cache culler archetype. Use the CNoExpireCuller class if no culling is to be performed.

Remarks

CBlobCache provides support for caching items in memory. Data can be added to the cache with the Add method and removed with the RemoveEntry method. The number of currently stored items can be retrieved with the GetCurrentEntryCount method.

CBlobCache derives from CMemoryCache, which provides most of the class's functionality. **CBlobCache** adds to this by periodically removing old, unused entries from the cache. If *MonitorClass* is set to CWorkerThread, the cache will be flushed with a frequency determined by ATL_BLOB_CACHE_TIMEOUT.

Cache statistics can be gathered using the members exposed by the IMemoryCacheStats interface, such as GetHitCount and GetMissCount.

See the BlobCache sample for a demonstration of how this class can be used in an ATL Server application.

Requirements

Header: atlcache.h

See Also

Tasks

BlobCache Sample: Generic Memory Caching

StencilCache Sample: Uses the ATL Server Stencil Cache

Reference

CMemoryCache Class

IMemoryCache Interface

IMemoryCacheControl Interface

IMemoryCacheStats Interface

IWorkerThreadClient Interface

Other Resources

Caching

CBlobCache Members

Caching Reference

CBlobCache Members

Methods

Add	Implementation of IMemoryCache::Add.
AddRef	Implementation of IUnknown::AddRef.
CBlobCache	The constructor.
~CBlobCache	The destructor.
ClearStats	Implementation of IMemoryCacheStats::ClearStats.
CloseHandle	Implementation of IWorkerThreadClient::CloseHandle.
Execute	Implementation of IWorkerThreadClient::Execute.
Flush	Implementation of IMemoryCache::Flush.
GetCurrentAllocSize	Implementation of IMemoryCacheStats::GetCurrentAllocSize.
GetCurrentEntryCount	Implementation of IMemoryCacheStats::GetCurrentEntryCount.
GetData	Implementation of IMemoryCache::GetData.
GetHitCount	Implementation of IMemoryCacheStats::GetHitCount.
GetMaxAllocSize	Implementation of IMemoryCacheStats::GetMaxAllocSize.
GetMaxAllowedEntries	Implementation of IMemoryCacheControl::GetMaxAllowedEntries.
GetMaxAllowedSize	Implementation of IMemoryCacheControl::GetMaxAllowedSize.
GetMaxEntryCount	Implementation of IMemoryCacheStats::GetMaxEntryCount.
GetMissCount	Implementation of IMemoryCacheStats::GetMissCount.
Initialize	Call this method to initialize the cache.
LookupEntry	Implementation of IMemoryCache::LookupEntry.
QueryInterface	Implementation of IUnknown::QueryInterface.
Release	Implementation of IUnknown::Release.
ReleaseEntry	Implements IMemoryCache::ReleaseEntry.
RemoveEntry	Implements IMemoryCache::RemoveEntry.
RemoveEntryByKey	Implements IMemoryCache::RemoveEntryByKey.

ResetCache	Implementation of IMemoryCacheControl::ResetCache.
SetMaxAllowedEntries	Implementation of IMemoryCacheControl::SetMaxAllowedEntries.
SetMaxAllowedSize	Implementation of IMemoryCacheControl::SetMaxAllowedSize.
Uninitialize	Call this method to uninitialize the cache.

See Also

Reference

CBlobCache Class

Other Resources

Caching

Caching Reference

CBlobCache Methods

For information about the methods in **CBlobCache**, see CBlobCache Members

CBlobCache::Add

Implementation of IMemoryCache::Add.

```
HRESULT STDMETHODCALLTYPE Add(
   LPCSTR szKey,
   void * pvData,
   DWORD dwSize,
   FILETIME * pftExpireTime,
   HINSTANCE hInstClient,
   HCACHEITEM * phEntry,
   IMemoryCacheClient * pClient
);
```

Remarks

The **Add** method calls CMemoryCache::AddEntry to perform the actual storage of the data.

Example

See the BlobCache Sample.

See Also

Reference

CBlobCache Class

IMemoryCache Interface

Other Resources

Caching

CBlobCache Members

Caching Reference

CBlobCache::AddRef

Implementation of IUnknown::AddRef.

```
ULONG STDMETHODCALLTYPE AddRef( );
```

Return Value

Always returns 1.

Remarks

This class does not implement lifetime control through reference counting.

See Also

Reference

CBlobCache Class
CBlobCache::Release
IMemoryCache Interface

Other Resources

Caching
CBlobCache Members
Caching Reference

CBlobCache::CBlobCache

The constructor.

```
CBlobCache( );
```

Remarks

The **CBlobCache** constructor performs no operations. The cache object must be initialized by calling Initialize.

See Also

Reference

CBlobCache Class

IMemoryCache Interface

Other Resources

Caching

CBlobCache Members

Caching Reference

CBlobCache::~CBlobCache

The destructor.

```
~CBlobCache( );
```

Remarks

If a timer for cache maintenance is being used, the timer is disabled.

See Also

Reference

CBlobCache Class

IMemoryCache Interface

Other Resources

Caching

CBlobCache Members

Caching Reference

CBlobCache::ClearStats

Implementation of IMemoryCacheStats::ClearStats.

```
HRESULT STDMETHODCALLTYPE ClearStats( );
```

Example

See the BlobCache Sample.

See Also

Reference

CBlobCache Class
IMemoryCacheStats Interface
IMemoryCache Interface

Other Resources

Caching
CBlobCache Members
Caching Reference

CBlobCache::CloseHandle

Implementation of IWorkerThreadClient::CloseHandle.

```
HRESULT CloseHandle(
   HANDLE hObject
);
```

Parameters

hObject

The handle of the cache monitoring timer object.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The cache monitoring class, typically CWorkerThread, calls this method to disable the monitoring timer. It is not necessary to call **CloseHandle** explicitly.

See Also

Reference

CBlobCache Class

IMemoryCache Interface

Other Resources

Caching

CBlobCache Members

Caching Reference

CBlobCache::Execute

Implementation of IWorkerThreadClient::Execute.

```
HRESULT Execute(
   DWORD_PTR dwParam,
   HANDLE /*hObject*/
);
```

Remarks

The **Execute** method performs the cache maintenance. The effects on the cache vary depending on the template parameters used to construct the **CBlobCache** object, and the settings specified through the IMemoryCacheControl interface.

If CNoWorkerThread was used as the monitor class argument during construction, the **Execute** method will not be called.

It is not normally necessary to call **Execute** explicitly.

See Also

Reference

CBlobCache Class

IMemoryCache Interface

Other Resources

Caching

CBlobCache Members

Caching Reference

CBlobCache::Flush

Implementation of IMemoryCache::Flush.

```
HRESULT STDMETHODCALLTYPE Flush( );
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

If CWorkerThread is used as the *MonitorClass* template parameter, the **Flush** method is called periodically to remove expired items that are not currently being used from the cache. The frequency with which **Flush** is called depends on the value of ATL_BLOB_CACHE_TIMEOUT. **Flush** can also be called explicitly.

Flush is implemented by a call to CMemoryCacheBase::FlushEntries. The effects of calling this method vary depending on the flushing and culling template parameters used to instantiate the **CBlobCache** template and the settings specified through the IMemoryCacheControl interface.

Example

See the BlobCache Sample.

See Also

Reference

CBlobCache Class
IMemoryCache::Flush

Other Resources

Caching
CBlobCache Members
Caching Reference

CBlobCache::GetCurrentAllocSize

Implementation of IMemoryCacheStats::GetCurrentAllocSize.

```
HRESULT STDMETHODCALLTYPE GetCurrentAllocSize(
   DWORD* pdwSize
);
```

Example

See the StencilCache Sample.

See Also

Reference

CBlobCache Class
IMemoryCacheStats Interface
IMemoryCache Interface

Other Resources

Caching
CBlobCache Members
Caching Reference

CBlobCache::GetCurrentEntryCount

Implementation of IMemoryCacheStats::GetCurrentEntryCount.

```
HRESULT STDMETHODCALLTYPE GetCurrentEntryCount(
   DWORD* pdwSize
);
```

Example

See the BlobCache Sample and the StencilCache Sample.

See Also

Reference

CBlobCache Class
IMemoryCacheStats Interface
IMemoryCache Interface

Other Resources

Caching
CBlobCache Members
Caching Reference

CBlobCache::GetData

Implementation of IMemoryCache::GetData.

```
HRESULT STDMETHODCALLTYPE GetData(
   const HCACHEITEM hKey,
   void ** ppvData,
   DWORD * pdwSize
) const;
```

See Also

Reference

CBlobCache Class

IMemoryCache Interface

Other Resources

Caching

CBlobCache Members

Caching Reference

CBlobCache::GetHitCount

Implementation of IMemoryCacheStats::GetHitCount.

```
HRESULT STDMETHODCALLTYPE GetHitCount(
   DWORD* pdwSize
);
```

Example

See the BlobCache Sample and the StencilCache Sample.

See Also

Reference

CBlobCache Class
CBlobCache::GetMissCount
IMemoryCacheStats Interface
IMemoryCache Interface

Other Resources

Caching
CBlobCache Members
Caching Reference

CBlobCache::GetMaxAllocSize

Implementation of IMemoryCacheStats::GetMaxAllocSize.

```
HRESULT STDMETHODCALLTYPE GetMaxAllocSize(
   DWORD* pdwSize
);
```

Example

See the StencilCache Sample.

See Also

Reference

CBlobCache Class
IMemoryCacheStats Interface
IMemoryCache Interface

Other Resources

Caching
CBlobCache Members
Caching Reference

CBlobCache::GetMaxAllowedEntries

Implementation of IMemoryCacheControl::GetMaxAllowedEntries.

```
HRESULT STDMETHODCALLTYPE GetMaxAllowedEntries(
   DWORD* pdwSize
);
```

Example

See the BlobCache Sample.

See Also

Reference

CBlobCache Class
CBlobCache::SetMaxAllowedEntries
IMemoryCacheControl Interface
IMemoryCache Interface

Other Resources

Caching
CBlobCache Members
Caching Reference

CBlobCache::GetMaxAllowedSize

Implementation of IMemoryCacheControl::GetMaxAllowedSize.

```
HRESULT STDMETHODCALLTYPE GetMaxAllowedSize(
   DWORD* pdwSize
);
```

See Also

Reference

CBlobCache Class
CBlobCache::SetMaxAllowedSize
IMemoryCacheControl Interface
IMemoryCache Interface

Other Resources

Caching
CBlobCache Members
Caching Reference

CBlobCache::GetMaxEntryCount

Implementation of IMemoryCacheStats::GetMaxEntryCount.

```
HRESULT STDMETHODCALLTYPE GetMaxEntryCount(
   DWORD* pdwSize
);
```

Example

See the BlobCache Sample and the StencilCache Sample.

See Also

Reference

CBlobCache Class
IMemoryCache Interface

Other Resources

Caching
CBlobCache Members
Caching Reference

CBlobCache::GetMissCount

Implementation of IMemoryCacheStats::GetMissCount.

```
HRESULT STDMETHODCALLTYPE GetMissCount(
   DWORD* pdwSize
);
```

Example

See the BlobCache Sample and the StencilCache Sample.

See Also

Reference

CBlobCache Class
CBlobCache::GetHitCount
IMemoryCache Interface

Other Resources

Caching
CBlobCache Members
Caching Reference

CBlobCache::Initialize

Call this method to initialize the cache.

```
HRESULT Initialize(
   IServiceProvider* pProv
);
template < class ThreadTraits >
HRESULT Initialize(
   IServiceProvider * pProv,
   CWorkerThread< ThreadTraits > * pWorkerThread
);
```

Parameters

pProv

The address of an object that implements the IServiceProvider interface. If provided, **CBlobCache** will attempt to obtain an IDllCache implementation for each cache entry with DLL dependencies. See the Add method for information on DLL dependencies.

pWorkerThread

The optional address of a CWorkerThread object that this **CBlobCache** object should use to perform periodic cache maintenance instead of creating a new thread. If **NULL**, the cache object will create a new worker thread to perform cache maintenance.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

This class uses CWorkerThread to perform periodic maintenance. By default, each instance creates its own thread. One instance of CWorkerThread, however, can be used to service multiple objects by providing a pointer to **Initialize.**

Example

See the BlobCache Sample.

See Also

Reference

CBlobCache Class
IMemoryCache Interface

Other Resources

Caching
CBlobCache Members
Caching Reference

CBlobCache::LookupEntry

Implementation of IMemoryCache::LookupEntry.

```
HRESULT STDMETHODCALLTYPE LookupEntry(
   LPCSTR szKey,
   HCACHEITEM * phEntry
);
```

Example

See the BlobCache Sample.

See Also

Reference

CBlobCache Class

IMemoryCache Interface

Other Resources

Caching

CBlobCache Members

Caching Reference

CBlobCache::QueryInterface

Implementation of IUnknown::QueryInterface.

```
HRESULT STDMETHODCALLTYPE QueryInterface(
   REFIID riid,
   void** ppv
);
```

Remarks

Objects of this class can be successfully queried for the **IUnknown**, IMemoryCache, IMemoryCacheStats and IMemoryCacheControl interfaces.

Example

See the BlobCache Sample.

See Also

Reference

CBlobCache Class

Other Resources

Caching

CBlobCache Members

Caching Reference

CBlobCache::Release

Implementation of IUnknown::Release.

```
ULONG STDMETHODCALLTYPE Release( );
```

Return Value

Always returns 1.

Remarks

This class does not implement lifetime control through reference counting.

See Also

Reference

CBlobCache Class
CBlobCache::AddRef

Other Resources

Caching
CBlobCache Members
Caching Reference

CBlobCache::ReleaseEntry

Implements IMemoryCache::ReleaseEntry.

```
HRESULT STDMETHODCALLTYPE ReleaseEntry(
   const HCACHEITEM hKey
);
```

Example

See the BlobCache Sample.

See Also

Reference

CBlobCache Class
IMemoryCache Interface

Other Resources

Caching
CBlobCache Members
Caching Reference

CBlobCache::RemoveEntry

Implements IMemoryCache::RemoveEntry.

```
HRESULT STDMETHODCALLTYPE RemoveEntry(
   const HCACHEITEM hKey
);
```

See Also

Reference

CBlobCache Class

IMemoryCache Interface

Other Resources

Caching

CBlobCache Members

Caching Reference

CBlobCache::RemoveEntryByKey

Implements IMemoryCache::RemoveEntryByKey.

```
HRESULT STDMETHODCALLTYPE RemoveEntryByKey(
   LPCSTR szKey
);
```

See Also

Reference

CBlobCache Class

IMemoryCache Interface

Other Resources

Caching

CBlobCache Members

Caching Reference

CBlobCache::ResetCache

Implementation of IMemoryCacheControl::ResetCache.

```
HRESULT STDMETHODCALLTYPE ResetCache( );
```

Example

See the BlobCache Sample.

See Also

Reference

CBlobCache Class
IMemoryCacheStats Interface
IMemoryCache Interface

Other Resources

Caching
CBlobCache Members
Caching Reference

CBlobCache::SetMaxAllowedEntries

Implementation of IMemoryCacheControl::SetMaxAllowedEntries.

```
HRESULT STDMETHODCALLTYPE SetMaxAllowedEntries(
   DWORD dwSize
);
```

Example

See the BlobCache Sample.

See Also

Reference

CBlobCache Class
CBlobCache::GetMaxAllowedEntries
IMemoryCacheControl Interface
IMemoryCache Interface

Other Resources

Caching
CBlobCache Members
Caching Reference

CBlobCache::SetMaxAllowedSize

Implementation of IMemoryCacheControl::SetMaxAllowedSize.

```
HRESULT STDMETHODCALLTYPE SetMaxAllowedSize(
   DWORD dwSize
);
```

See Also

Reference

CBlobCache Class
CBlobCache::GetMaxAllowedSize
IMemoryCacheControl Interface
IMemoryCache Interface

Other Resources

Caching
CBlobCache Members
Caching Reference

CBlobCache::Uninitialize

Call this method to uninitialize the cache.

```
HRESULT Uninitialize( );
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Releases cache items and resources.

Example

See the BlobCache Sample.

See Also

Reference

CBlobCache Class

IMemoryCache Interface

Other Resources

Caching

CBlobCache Members

Caching Reference

CBasicAuthObject Class

This class implements support for performing HTTP navigation using the BASIC authentication scheme.

```
class CBasicAuthObject : public CAtlBaseAuthObject
```

Remarks

This class performs BASIC authentication for protected URLs and is provided to the CAtlHttpClientT Class class through its AddAuthObj method before the request is launched. In addition to an instance of the **CBasicAuthObject** class itself, an implementation of the IAuthInfo interface must be provided to AddAuthObj so that the CAtlHttpClientT class can retrieve the required credentials when needed.

ATL Server also support NTLM authentication through the CNTLMAuthObject class.

Requirements

Header: atlhttp.h

Example

See the HttpClient Sample.

See Also

Reference

CAtlBaseAuthObject Class

Concepts

HTTP Client Services

Other Resources

CBasicAuthObject Members

HTTP Client Reference

CBasicAuthObject Members

Methods

Authenticate	This method is called to perform BASIC authentication. It is not typically necessary to call this method explicitly.
CBasicAuthObject	Constructor overloads.
GetRealm	Call this method to get the realm that was used during the authentication negotiation
Init	This method is called to initialize the basic authentication object. It is not normally necessary to call this method explicitly.
SetAuthInfo	This method is called to attach authorization information. It is not normally necessary to call this method explicitly.

See Also

Reference

CBasicAuthObject Class

Concepts

HTTP Client Services

Other Resources

HTTP Client Reference

CBasicAuthObject Methods

For information about the methods in **CBasicAuthObject**, see CBasicAuthObject Members

CBasicAuthObject::Authenticate

This method is called to perform BASIC authentication. It is not typically necessary to call this method explicitly.

```
virtual bool Authenticate(
   LPCTSTR szAuthTypes,
   bool bProxy
) throw( );
```

Parameters

szAuthTypes

A string containing the authentication scheme name. For the BASIC scheme, this string will be "BASIC", as specified by the ATL_HTTP_AUTHTYPE_BASIC macro.

bProxy

true if a proxy server is being used for HTTP transactions, otherwise **false**.

Return Value

Returns **true** on success, **false** on failure.

Remarks

See the CAtlBaseAuthObject::Authenticate method for more information.

Example

See the HttpClient Sample.

See Also

Reference

CBasicAuthObject Class

Concepts

HTTP Client Services

Other Resources

CBasicAuthObject Members

HTTP Client Reference

CBasicAuthObject::CBasicAuthObject

Constructor overloads.

```
CBasicAuthObject( ) throw( );
CBasicAuthObject(
   IAuthInfo* pAuthInfo
) throw( );
```

Parameters

pAuthInfo

The address of an object that implements the IAuthInfo interface for the purposes of providing authorization credentials.

Remarks

If the default constructor is used, an implementation of the IAuthInfo interface must be provided to the CAtlHttpClientT class when this object is attached using the AddAuthObj method. Alternately, the IAuthInfo entity can be provided to this object directly using the alternate constructor, and **NULL** can be passed to AddAuthObj in place of the IAuthInfo implementation.

See Also

Reference

CBasicAuthObject Class

Concepts

HTTP Client Services

Other Resources

CBasicAuthObject Members

HTTP Client Reference

CBasicAuthObject::GetRealm

Call this method to get the realm that was used during the authentication negotiation

```
LPCTSTR GetRealm( ) throw( );
```

Return Value

A string containing the name of the realm that is hosting the protected resource.

Remarks

It is not normally necessary to use the **GetRealm** method.

See Also

Reference

CBasicAuthObject Class

Other Resources

CBasicAuthObject Members

HTTP Client Reference

CBasicAuthObject::Init

This method is called to initialize the basic authentication object. It is not normally necessary to call this method explicitly.

```
virtual void Init(
   CAtlHttpClient* pSocket,
   IAuthInfo* pAuthInfo = NULL
) throw( );
```

Remarks

See the CAtlBaseAuthObject::Init method for more information.

Example

See the HttpClient Sample.

See Also

Reference

CBasicAuthObject Class

Concepts

HTTP Client Services

Other Resources

CBasicAuthObject Members

HTTP Client Reference

CBasicAuthObject::SetAuthInfo

This method is called to attach authorization information. It is not normally necessary to call this method explicitly.

```
void SetAuthInfo(
   IAuthInfo* pAuthInfo
) throw( );
```

Parameters

pAuthInfo

The address on an object implementing the IAuthInfo interface.

Remarks

This method is called internally by CBasicAuthObject::Init and is not normally called by the application code.

See Also

Reference

CBasicAuthObject Class

Concepts

HTTP Client Services

Other Resources

CBasicAuthObject Members

HTTP Client Reference

CBrowserCapsSvc Class

This class allows you to get access to objects that provide information about the capabilities of a particular user agent (Web browser).

```
class CBrowserCapsSvc :
   public IBrowserCapsSvc,
   public CComObjectRootEx< CComSingleThreadModel >
```

Remarks

The information about a particular user agent is exposed by the IBrowserCaps interface. Pointers to objects implementing this interface are returned by the IBrowserCapsSvc::GetCaps and IBrowserCapsSvc::GetCapsUserAgent methods.

This class implements the IBrowserCapsSvc by reading information from a file named **browscap.ini**. This file associates user agent strings (Web browser identifiers) with information about the capabilities of those user agents. The format of this file is the same format used by the Browser Capabilites component described in Browscap.ini File.

Note that **browscap.ini** will be loaded from the same location as the module whose HINSTANCE is used to initialize this object if a file of that name is present. Otherwise it will be loaded from the **inetsrv** directory beneath the system directory (the location of the standard **browscap.ini** file installed by IIS). See CBrowserCapsSvc::Initialize. This method must be called before any other method in this class.

Requirements

Header: atlutil.h

See Also

Reference

ShowBrowser
IBrowserCapsSvc Interface
IBrowserCaps Interface

Other Resources

CBrowserCapsSvc Members
ATL Server Classes

CBrowserCapsSvc Members

Methods

GetCaps	Implementation of IBrowserCapsSvc::GetCaps.
GetCapsUserAgent	Implementation of IBrowserCapsSvc::GetCapsUserAgent.
Initialize	Call this method to initialize the browser capabilities service object.
Uninitialize	Call this method to uninitialize the browser capabilities service object.

See Also

Reference

CBrowserCapsSvc Class

CBrowserCapsSvc Methods

For information about the methods in **CBrowserCapsSvc**, see CBrowserCapsSvc Members

CBrowserCapsSvc::GetCaps

Implementation of IBrowserCapsSvc::GetCaps.

```
HRESULT GetCaps(
   IHttpServerContext* pContext,
   IBrowserCaps** ppOut
);
```

See Also

Reference

CBrowserCapsSvc Class

IBrowserCapsSvc::GetCaps

Other Resources

CBrowserCapsSvc Members

CBrowserCapsSvc::GetCapsUserAgent

Implementation of IBrowserCapsSvc::GetCapsUserAgent.

```
HRESULT GetCapsUserAgent(
   BSTR bstrAgent,
   IBrowserCaps** ppOut
);
```

See Also

Reference

CBrowserCapsSvc Class

IBrowserCapsSvc::GetCapsUserAgent

Other Resources

CBrowserCapsSvc Members

CBrowserCapsSvc::Initialize

Call this method to initialize the browser capabilities service object.

```
HRESULT Initialize(
   HINSTANCE hInstance
);
```

Parameters

hInstance

The HINSTANCE of the module whose path is used to locate the **browscap.ini** file.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The information about the capabilities of each browser is kept in a file called **browscap.ini**. This file will be loaded from the same location as the module whose HINSTANCE is used to initialize this object if present or from the **inetsrv** directory beneath the system directory otherwise.

During initialization, **browscap.ini** is read and parsed. As a result, changes made to **browscap.ini** following initialization will not be reflected in the information returned by this object.

See Also

Reference

CBrowserCapsSvc Class

CBrowserCapsSvc::Uninitialize

Other Resources

CBrowserCapsSvc Members

CBrowserCapsSvc::Uninitialize

Call this method to uninitialize the browser capabilities service object.

```
HRESULT Uninitialize( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Following a call to **Uninitialize**, **browscaps.ini** may be reloaded by a call to CBrowserCapsSvc::Initialize and the object reused.

See Also

Reference

CBrowserCapsSvc Class

CBrowserCapsSvc::Initialize

Other Resources

CBrowserCapsSvc Members

CCacheDataBase Class

This is an empty class that is used as the default cache entry type by CMemoryCacheBase.

```
class CCacheDataBase
```

Remarks

The CMemoryCacheBase class allows the storage of auxiliary information for each cached item. This extra data is provided to CMemoryCacheBase as the *NodeInfo* template argument. The **CCacheDataBase** class, which provides no data members, is used by default to indicate that no auxiliary data is to be cached.

Requirements

Header: atlcache.h

See Also

Reference

CCacheDataEx Structure

Other Resources

ATL Server Classes

Caching Reference

Caching

CCacheDataEx Structure

This structure is used by CMemoryCache and CStencilCache to store auxiliary data for each item in the cache.

```
struct CCacheDataEx :
   public CCacheDataBase
```

Remarks

This structure allows DLL instance handles and cache client interface pointers to be associated with each item in the cache.

It is not normally necessary to use this class explicitly.

Requirements

Header: atlcache.h

See Also

Reference

CCacheDataBase Class

Other Resources

Caching

CCacheDataEx Members

Caching Reference

CCacheDataEx Members

Methods

CCacheDataEx	The constructor.

Data Members

hInstance	The handle of a DLL upon which the cached item relies.
pClient	The address of an IMemoryCacheClient implementation object responsible for freeing the cached memory when it is removed from the cache.

See Also

Reference

CCacheDataEx Structure

Other Resources

Caching

Caching Reference

CCacheDataEx Methods

For information about the methods in **CCacheDataEx**, see CCacheDataEx Members

CCacheDataEx::CCacheDataEx

The constructor.

```
CCacheDataEx( );
```

Remarks

Initializes the data members to **NULL**.

See Also

Reference

CCacheDataEx Structure

Other Resources

Caching

CCacheDataEx Members

Caching Reference

CCacheDataEx Data Members

For information about the data members in **CCacheDataEx**, see CCacheDataEx Members

CCacheDataEx::hInstance

The handle of a DLL upon which the cached item relies.

```
HINSTANCE hInstance;
```

Remarks

The CMemoryCache and CStencilCache classes, both of which use **CCacheDataEx** as a node type, allow DLL handles to be specified for cache entries. These handles are stored in the **hInstance** data member and used to ensure that the DLL is not unloaded before the cached memory. This functionality is provided by the cache implementation.

It is not normally necessary to access this data member explicitly.

See Also

Reference

CCacheDataEx Structure

Other Resources

Caching

CCacheDataEx Members

Caching Reference

CCacheDataEx::pClient

The address of an IMemoryCacheClient implementation object responsible for freeing the cached memory when it is removed from the cache.

```
IMemoryCacheClient* pClient;
```

Remarks

The CMemoryCache and CStencilCache classes, both of which use **CCacheDataEx** as a node type, allow memory cache clients to be specified for each cache entry. The address of each IMemoryCacheClient implementation is stored using the **pClient** data member. When the cached memory is removed from the cache, this interface is used to free the memory. This functionality is provided by the cache implementation.

It is not normally necessary to access this data member explicitly.

See Also

Reference

CCacheDataEx Structure

Other Resources

Caching

CCacheDataEx Members

Caching Reference

CCacheDataPeer Structure

This class is used by CFileCache to provide an interface through which auxiliary data can be attached to each cached file.

```
template < class Peer >
struct CCacheDataPeer :
   public CCacheDataBase
```

Parameters

Peer

A class conforming to the file cache peer archetype that defines the extra data to be associated with each file.

Remarks

To define a data cache peer data type, a class declaring a data type by the name of **PeerInfo** must be provided to **CCacheDataPeer** class as the *Peer* argument.

Requirements

Header: atlcache.h

See Also

Reference

CCacheDataBase Class

Other Resources

CCacheDataPeer Members

Caching Reference

Caching

CCacheDataPeer Members

Data Members

PeerData	An instance of the data that is associated with each file cache entry.

See Also

Reference

CCacheDataPeer Structure

CCacheDataPeer Data Members

For information about the data members in **CCacheDataPeer**, see CCacheDataPeer Members

CCacheDataPeer::PeerData

An instance of the data that is associated with each file cache entry.

```
Peer::PeerInfo PeerData;
```

Remarks

The type of the **PeerData** member is dependent on the *Peer* class provided to CCacheDataPeer as a template argument.

See Also

Reference

CCacheDataPeer Structure

Other Resources

CCacheDataPeer Members

Caching Reference

CCachePerfMon Class

This class is used in the implementation of CPerfStatClass.

```
class CCachePerfMon :
   public CPerfMon
```

Remarks

This class uses the ATL Server performance monitoring macros to manage CPerfStatObject performance objects. There is no need to use this class directly except to wrap it in a PERFREG_ENTRY macro to ensure that the performance counters are properly registered. See Enabling ISAPI Performance Monitor Support for more details.

Requirements

Header: atlcache.h

See Also

Reference

CPerfMon Class

Concepts

Predefined ISAPI Performance Counters

Other Resources

ATL Server Classes

CCacheServerContext Class

This class provides the server context for request handlers that request page-caching support.

```
class CCacheServerContext :
   public CComObjectRootEx< CComMultiThreadModel >,
   public CWrappedServerContext,
   public IPageCacheControl
```

Remarks

Request handlers can request page-caching support by returning ATLSRV_INIT_USECACHE from IRequestHandler::GetFlags. Request handlers that request page-caching support will use the **IHttpServerContext** implementation provided by this class, which stores the response header and body in order to cache them for future requests to the same URL. This class also provides the implementation of IPageCacheControl that request handlers can use to fine tune how the responses are cached. Request handlers can obtain this interface by calling **QueryInterface** on the **IHttpServerContext** interface already in their possession.

The implementation of IHttpServerContext provided by this class delegates most methods to the **IHttpServerContext** pointer passed to CCacheServerContext::Initialize (this delegation is handled by CWrappedServerContext). Certain **IHttpServerContext** methods are overridden in this class as described below.

The following **IHttpServerContext** methods have been overridden to enable the caching functionality:

- DoneWithSession
- SendResponseHeader
- WriteClient

The following **IHttpServerContext** methods need careful thought when generating cached responses. Calling these methods will generate trace messages in debug builds:

- AppendToLog
- GetImpersonationToken

The following **IHttpServerContext** methods should not be called when generating cached responses. Calling these methods will generate assertion failures in debug builds:

- AsyncReadClient
- AsyncWriteClient
- ReadClient
- RequestIOCompletion
- SendRedirectResponse
- TransmitFile

Requirements

Header: atlisapi.h

Example

See the MantaWeb Sample, the ShowErrors Sample, and the ShowUser Sample.

See Also

Reference

CComObjectRootEx Class
IHttpServerContext Interface
IPageCacheControl Interface

Concepts

Enabling Page Caching

Other Resources

CCacheServerContext Members

CCacheServerContext Members

Methods

AppendToLog	Implementation of IHttpServerContext::AppendToLog.
AsyncReadClient	Non-functional implementation of IHttpServerContext::AsyncReadClient. Always returns FALSE.
AsyncWriteClient	Non-functional implementation of IHttpServerContext::AsyncWriteClient. Always returns FALSE.
Cache	Implementation of IPageCacheControl::Cache.
CCacheServerContext	The constructor.
DoneWithSession	Implementation of IHttpServerContext::DoneWithSession.
GetExpiration	Implementation of IPageCacheControl::GetExpiration.
GetImpersonationToken	Implementation of IHttpServerContext::GetImpersonationToken.
Initialize	Call this method to initialize the object.
IsCached	Implementation of IPageCacheControl::IsCached.
ReadClient	Non-functional implementation of IHttpServerContext::ReadClient. Always returns FALSE.
RequestIOCompletion	Non-functional implementation of IHttpServerContext::RequestIOCompletion. Always returns FALSE.
SendRedirectResponse	Non-functional implementation of IHttpServerContext::SendRedirectResponse. Always returns FALSE.
SendResponseHeader	Implementation of IHttpServerContext::SendResponseHeader.
SetExpiration	Implementation of IPageCacheControl::SetExpiration.
TransmitFile	Non-functional implementation of IHttpServerContext::TransmitFile. Always returns FALSE.
WriteClient	Implementation of IHttpServerContext::WriteClient.

See Also

Reference

CCacheServerContext Class

CCacheServerContext Methods

For information about the methods in **CCacheServerContext**, see CCacheServerContext Members

CCacheServerContext::AppendToLog

Implementation of IHttpServerContext::AppendToLog.

```
BOOL AppendToLog(
   LPCSTR szMessage,
   DWORD* pdwLen
);
```

Remarks

The response generated by this page is going to be cached. Information will be written to the log file only when the response is generated, not when the response is subsequently retrieved from the cache. A trace message will be output in debug builds to remind you of that fact.

See Also

Reference

CCacheServerContext Class

IHttpServerContext::AppendToLog

Other Resources

CCacheServerContext Members

CCacheServerContext::AsyncReadClient

Non-functional implementation of IHttpServerContext::AsyncReadClient. Always returns FALSE.

```
BOOL AsyncReadClient(
   void * /* pvBuffer */,
   DWORD * /* pdwSize */
);
```

Remarks

Calling this method will result in an assertion failure in debug builds.

See Also

Reference

CCacheServerContext Class

IHttpServerContext::AsyncReadClient

Other Resources

CCacheServerContext Members

CCacheServerContext::AsyncWriteClient

Non-functional implementation of IHttpServerContext::AsyncWriteClient. Always returns FALSE.

```
BOOL AsyncWriteClient(
   void * /* pvBuffer */,
   DWORD * /* pdwBytes */
);
```

Remarks

Calling this method will result in an assertion failure in debug builds.

See Also

Reference

CCacheServerContext Class

IHttpServerContext::AsyncWriteClient

Other Resources

CCacheServerContext Members

CCacheServerContext::Cache

Implementation of IPageCacheControl::Cache.

```
BOOL Cache(
   BOOL bCache
) throw( );
```

Remarks

Caching is enabled by default.

See Also

Reference

CCacheServerContext Class

Other Resources

CCacheServerContext Members

CCacheServerContext::CCacheServerContext

The constructor.

```
CCacheServerContext( ) throw( );
```

Remarks

CCacheServerContext::Initialize must be called before any other methods in this class.

See Also

Reference

CCacheServerContext Class

Other Resources

CCacheServerContext Members

CCacheServerContext::DoneWithSession

Implementation of IHttpServerContext::DoneWithSession.

```
BOOL DoneWithSession(
   DWORD dwHttpStatusCode
);
```

Remarks

This implementation makes the response available to the cache.

See Also

Reference

CCacheServerContext Class

IHttpServerContext::DoneWithSession

Other Resources

CCacheServerContext Members

CCacheServerContext::GetExpiration

Implementation of IPageCacheControl::GetExpiration.

```
HRESULT GetExpiration(
   FILETIME * pftExpiration
) throw( );
```

Remarks

The expiration time defaults to 0 meaning that the cached response never expires.

See Also

Reference

CCacheServerContext Class

IPageCacheControl::GetExpiration

Other Resources

CCacheServerContext Members

CCacheServerContext::GetImpersonationToken

Implementation of IHttpServerContext::GetImpersonationToken.

```
BOOL GetImpersonationToken(
   HANDLE * pToken
);
```

Remarks

Typically, you should not call this method unless you are sure that you know what you are doing. The response generated by this page is going to be cached. The cache is not designed for secure information (that is, the cache does not differentiate between client requests), so it is likely that you should not be doing client impersonation to generate a cached response. A trace message will be output in debug builds to remind you of this.

Example

See the ShowUser Sample.

See Also

Reference

CCacheServerContext Class

IHttpServerContext::GetImpersonationToken

Other Resources

CCacheServerContext Members

CCacheServerContext::Initialize

Call this method to initialize the object.

```
BOOL Initialize(
   IHttpServerContext * pParent,
   IFileCache * pCache
) throw( );
```

Parameters

pParent

The parent server context.

pCache

The object used to cache the response.

Remarks

Call this method to initialize the object before calling any of the other methods.

See Also

Reference

CCacheServerContext Class

Other Resources

CCacheServerContext Members

CCacheServerContext::IsCached

Implementation of IPageCacheControl::IsCached.

```
BOOL IsCached( ) throw( );
```

Remarks

Caching is enabled by default.

See Also

Reference

CCacheServerContext Class

IPageCacheControl::IsCached

Other Resources

CCacheServerContext Members

CCacheServerContext::ReadClient

Non-functional implementation of IHttpServerContext::ReadClient. Always returns FALSE.

```
BOOL ReadClient(
   void * /* pvBuffer */,
   DWORD * /* pdwSize */
);
```

Remarks

Calling this method will result in an assertion failure in debug builds.

See Also

Reference

CCacheServerContext Class

IHttpServerContext::ReadClient

Other Resources

CCacheServerContext Members

CCacheServerContext::RequestIOCompletion

Non-functional implementation of IHttpServerContext::RequestIOCompletion. Always returns FALSE.

```
BOOL RequestIOCompletion(
   PFN_HSE_IO_COMPLETION /* pfn */,
   DWORD * /* pdwContext */
);
```

Remarks

Calling this method will result in an assertion failure in debug builds.

See Also

Reference

CCacheServerContext Class

IHttpServerContext::RequestIOCompletion

Other Resources

CCacheServerContext Members

CCacheServerContext::SendRedirectResponse

Non-functional implementation of IHttpServerContext::SendRedirectResponse. Always returns FALSE.

```
BOOL SendRedirectResponse(
   LPCSTR /* pszRedirectUrl */
);
```

Remarks

Calling this method will result in an assertion failure in debug builds.

See Also

Reference

CCacheServerContext Class

IHttpServerContext::SendRedirectResponse

Other Resources

CCacheServerContext Members

CCacheServerContext::SendResponseHeader

Implementation of IHttpServerContext::SendResponseHeader.

```
BOOL SendResponseHeader(
   LPCSTR pszHeader = "Content-Type: text/html\r\n\r\n",
   LPCSTR pszStatusCode = "200 OK",
   BOOL fKeepConn = FALSE
);
```

Remarks

Keeps track of the response header and status code so that they can be cached and returned along with the response body on subsequent requests.

Example

See the ShowErrors Sample.

See Also

Reference

CCacheServerContext Class

IHttpServerContext::SendResponseHeader

Other Resources

CCacheServerContext Members

CCacheServerContext::SetExpiration

Implementation of IPageCacheControl::SetExpiration.

```
HRESULT SetExpiration(
   FILETIME ftExpiration
) throw( );
```

Remarks

The expiration time defaults to 0 meaning that the cached response never expires.

See Also

Reference

CCacheServerContext Class

IPageCacheControl::SetExpiration

Other Resources

CCacheServerContext Members

CCacheServerContext::TransmitFile

Non-functional implementation of IHttpServerContext::TransmitFile. Always returns FALSE.

```
BOOL TransmitFile(
   HANDLE /* hFile */,
   PFN_HSE_IO_COMPLETION /* pfn */,
   void * /* pContext */,
   LPCSTR /* szStatusCode */,
   DWORD /* dwBytesToWrite */,
   DWORD /* dwOffset */,
   void * /* pvHead */,
   DWORD /* dwHeadLen */,
   void * /* pvTail */,
   DWORD /* dwTailLen */,
   DWORD /* dwFlags */
);
```

Remarks

Calling this method will result in an assertion failure in debug builds.

See Also

Reference

CCacheServerContext Class

IHttpServerContext::TransmitFile

Other Resources

CCacheServerContext Members

CCacheServerContext::WriteClient

Implementation of IHttpServerContext::WriteClient.

```
BOOL WriteClient(
   void * pvBuffer,
   DWORD * pdwBytes
);
```

Remarks

This implementation of IHttpServerContext::WriteClient writes the data to a file that will be cached when the response is complete as well as sending the data to the client in the normal way.

Example

See the MantaWeb Sample and the ShowErrors Sample.

See Also

Reference

CCacheServerContext Class
IHttpServerContext::WriteClient

Other Resources

CCacheServerContext Members

CCookie Class

This class represents a cookie to be sent from the server to a client or a cookie that has been returned by a client to the originating server.

```
class CCookie :
   public CValidateObject< CCookie >
```

Remarks

Cookies provide a way for a server to store a small amount of data on a client and have that data returned to it on each request the client makes. Use this class to represent a cookie to be sent from the server to a client or to represent a cookie that has been returned by a client to the originating server.

At the HTTP level, a cookie is an application-defined name-value pair plus some standard attribute-value pairs that describe the way in which the user agent (web browser) should interact with the cookie.

The **CCookie** class provides methods to set and get the application-defined name and value as well as methods for the standard attributes. In addition, **CCookie** provides an abstraction on top of the application-defined value that allows it to be treated as a collection of name-value pairs. Cookies with a single value are known as single-valued cookies. Cookies whose value consists of name-value pairs are known as multivalued cookies or dictionary cookies.

You can set the name of a cookie by calling CCookie::SetName or using the appropriate constructor.

You can set the value of a cookie by calling CCookie::SetValue or using the appropriate constructor. If the cookie has a value set, it is a single-valued cookie and attempts to add a name-value pair will fail. You can remove the value of a cookie by calling **SetValue(NULL)**.

You can add a name-value pair to a cookie by calling CCookie::AddValue. If the cookie has any name-value pairs, it is a multivalued cookie and attempts to set the primary value will fail. You can remove all the name-value pairs of a cookie by calling CCookie::RemoveAllValues.

Cookie attributes can be set using CCookie::AddAttribute or the more specific methods listed below:

- CCookie::SetComment
- CCookie::SetCommentUrl
- CCookie::SetDiscard
- CCookie::SetDomain
- CCookie::SetExpires
- CCookie::SetMaxAge
- CCookie::SetPath
- CCookie::SetPort
- CCookie::SetSecure
- CCookie::SetVersion

The format of a cookie is described in RFC 2109 (http://ietf.org/rfc/rfc2109.txt). **CCookie** complies with this RFC in addition to the previous informal standard implemented by many web browsers. Attributes that are specified by RFC 2109 are termed RFC 2109 attributes in the documentation for this class. Attributes that are part of the commonly implemented informal standard are described as Version 0 attributes.

Note that the values of a cookie can be easily validated using the functionality of the CValidateObject base class. See CCookie::Lookup for information that affects the operation of the base class.

Requirements

Header: atlisapi.h

See Also

Tasks

MantaWeb Sample: Online Personal Information Management

StencilCache Sample: Uses the ATL Server Stencil Cache

Reference

Cookies

ForceLogin

Other Resources

CCookie Members

ATL Server Classes

CCookie Members

Methods

AddAttribute	Call this method to add an attribute-value pair to the collection of attributes for this cookie.
AddValue	Call this method to add a name-value pair to a cookie object.
CCookie	The constructor for a cookie object.
Empty	Call this method to clear the cookie of all content, including name, value, name-value pairs, and attributes.
GetAttributeValueAt	Call this method to get the value of the attribute at the specified position.
GetFirstAttributePos	Call this method to get the position of the first attribute associated with this cookie.
GetFirstValuePos	Call this method to get the position of the first value associated with this cookie.
GetName	Call this method to get the name from a cookie object.
GetNextAttrAssoc	Call this method to get the name and value of an attribute based on its POSITION within the collection of attributes managed by this cookie.
GetNextAttributeName	Call this method to get the name of an attribute based on its POSITION within the collection of attributes managed by this cookie.
GetNextValueAssoc	Call this method to get the name and value of a value based on its POSITION within the collection of values managed by this cookie.
GetNextValueName	Call this method to get the name of a value based on its POSITION within the collection of values managed by this cookie.
GetValue	Call this method to get the value from a cookie object.
GetValueAt	Call this method to get the value at the specified position.
IsEmpty	Call this method to determine whether a cookie object is empty.
Lookup	Call this method to look up the value or a name-value pair applied to this cookie.
ModifyAttribute	Call this method to modify an attribute-value pair associated with the cookie.
ModifyValue	Call this method to modify a name-value pair associated with the cookie.
Parse	Call this method to populate a **CCookie** from a cookie request header.
RemoveAllAttributes	Call this method to remove all the attribute-value pairs from the collection of attributes managed by this cookie.
RemoveAllValues	Call this method to remove all the name-value pairs from the collection managed by this cookie.

RemoveAttribute	Call this method to remove an attribute-value pair from the collection of attributes managed by this cookie.
RemoveValue	Call this method to remove a name-value pair from the collection managed by this cookie.
Render	Call this method to put the correct HTTP representation of this cookie into a buffer.
SetComment	Call this method to set the Comment attribute of the cookie.
SetCommentUrl	Call this method to set the CommentUrl attribute of the cookie.
SetDiscard	Call this method to add or remove the Discard attribute of the cookie.
SetDomain	Call this method to set the Domain attribute of the cookie.
SetExpires	Call this method to set the Expires attribute of the cookie.
SetMaxAge	Call this method to set the Max-Age attribute of the cookie.
SetName	Call this method to set the name of a cookie.
SetPath	Call this method to set the Path attribute of the cookie.
SetPort	Call this method to set the Port attribute of the cookie.
SetSecure	Call this method to add or remove the Secure attribute of the cookie.
SetValue	Call this method to set the value of this cookie.
SetVersion	Call this method to set the Version attribute of the cookie.

Operators

operator =	The copy assignment operator.

See Also

Reference

CCookie Class

CCookie Methods

For information about the methods in **CCookie**, see CCookie Members.

CCookie::AddAttribute

Call this method to add an attribute-value pair to the collection of attributes for this cookie.

```
BOOL AddAttribute(
   LPCSTR szName,
   LPCSTR szValue
) throw( );
```

Parameters

szName

The name.

szValue

The value.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This method is equivalent to CCookie::ModifyAttribute. Both methods will add the attribute if not already present or change its value if it has already been applied to the cookie.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::AddValue

Call this method to add a name-value pair to a cookie object.

```
BOOL AddValue(
   LPCSTR szName,
   LPCSTR szValue
) throw( );
```

Parameters

szName

The name.

szValue

The value.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

If the named value is already present in the cookie, calling this method will modify the current value; otherwise a new name-value pair is added to the cookie.

Call CCookie::RemoveValue or CCookie::RemoveAllValues to remove the name-value pairs added by this method.

This method will fail if the cookie is single-valued. If necessary, call CCookie::SetValue**(NULL)** to remove the single value before calling this method.

Example

See the MantaWeb Sample and the Cookies Sample.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::CCookie

The constructor for a cookie object.

```
CCookie( ) throw( );
CCookie(
   LPCSTR szName
) throw(...);
CCookie(
   LPCSTR szName,
   LPCSTR szValue
) throw(...);
CCookie(
   const CCookie& thatCookie
) throw(...);
```

Parameters

szName

The name of the cookie.

szValue

The value of the cookie.

thatCookie

The cookie to be copied.

Remarks

The constructors allow you to create an empty cookie, a valueless cookie, a single-valued cookie, or a copy of an existing cookie, respectively.

Example

See the Cookies Sample.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::Empty

Call this method to clear the cookie of all content, including name, value, name-value pairs, and attributes.

```
void Empty( ) throw( );
```

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::GetAttributeValueAt

Call this method to get the value of the attribute at the specified position.

```
const CStringA& GetAttributeValueAt(
   POSITION pos
) const throw( );
```

Parameters

pos

The position of an attribute in the collection.

Return Value

Returns the value of the attribute at the specified position.

Remarks

The POSITION passed to this method can be obtained from a previous call to GetFirstAttributePos, GetNextAttrAssoc, or GetNextAttributeName.

See Also

Reference

CCookie Class

CCookie::GetFirstAttributePos

Other Resources

CCookie Members

CCookie::GetFirstAttributePos

Call this method to get the position of the first attribute associated with this cookie.

```
POSITION GetFirstAttributePos( ) const throw( );
```

Return Value

Returns the position of the first attribute associated with this cookie or NULL if no attributes are associated with this cookie.

Remarks

The POSITION returned by this method can be passed to CCookie::GetNextAttrAssoc to get the name and value of the attribute and advance the position to the next attribute.

See Also

Reference

CCookie Class

CCookie::GetNextAttrAssoc

Other Resources

CCookie Members

CCookie::GetFirstValuePos

Call this method to get the position of the first value associated with this cookie.

```
POSITION GetFirstValuePos( ) const throw( );
```

Return Value

Returns the position of the first value associated with this cookie or NULL if no values are associated with this cookie.

Remarks

The POSITION returned by this method can be passed to CCookie::GetNextValueAssoc to get the name and value of the value and advance the position to the next value.

Example

See the Cookies Sample.

See Also

Reference

CCookie Class

CCookie::GetNextValueAssoc

Other Resources

CCookie Members

CCookie::GetName

Call this method to get the name from a cookie object.

```
BOOL GetName(
   CStringA & strName
) const throw( );
BOOL GetName(
   LPSTR szBuff,
   DWORD* pdwSize
) const throw( );
```

Parameters

strName

A reference to a string that will hold the name of the cookie on successful return of this method.

szBuff

A caller-allocated buffer that will hold the name of the cookie on successful return of this method.

pdwSize

A pointer to a DWORD variable. On entry, its value must be the size in bytes of the buffer *szBuff*. On exit, the value will be the length of the string copied to the buffer, not including the terminating null character.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This method returns the name that was passed to the constructor or that was set by calling CCookie::SetName.

Example

See the Cookies Sample.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::GetNextAttrAssoc

Call this method to get the name and value of an attribute based on its POSITION within the collection of attributes managed by this cookie.

```
BOOL GetNextAttrAssoc(
   POSITION& pos,
   CStringA& key,
   CStringA& val
) const throw( );
```

Parameters

pos

On entry, the position of an attribute in the collection. On exit, the position of the next attribute in the collection, or NULL if there are no further attributes.

key

The name of the attribute corresponding to *pos*.

val

The value of the attribute corresponding to *pos*.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The POSITION passed to this method should have been returned by a previous call to CCookie::GetFirstAttributePos or **CCookie::GetNextAttrAssoc** on the same object.

See Also

Reference

CCookie Class

CCookie::GetFirstAttributePos

Other Resources

CCookie Members

CCookie::GetNextAttributeName

Call this method to get the name of an attribute based on its POSITION within the collection of attributes managed by this cookie.

```
const CStringA& GetNextAttributeName(
   POSITION& pos
) const throw( );
```

Parameters

pos

On entry, the position of an attribute in the collection. On exit, the position of the next attribute in the collection, or NULL if there are no further attributes.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The POSITION passed to this method should have been returned by a previous call to CCookie::GetFirstAttributePos, CCookie::GetNextAttrAssoc, or **CCookie::GetNextAttributeName** on the same object.

See Also

Reference

CCookie Class

CCookie::GetFirstAttributePos

Other Resources

CCookie Members

CCookie::GetNextValueAssoc

Call this method to get the name and value of a value based on its POSITION within the collection of values managed by this cookie.

```
BOOL GetNextValueAssoc(
   POSITION& pos,
   CStringA& key,
   CStringA& val
) const throw( );
```

Parameters

pos

On entry, the position of a value in the collection. On exit, the position of the next value in the collection, or NULL if there are no further values.

key

The name of the value corresponding to *pos*.

val

The value of the value corresponding to *pos*.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The POSITION passed to this method should have been returned by a previous call to CCookie::GetFirstValuePos or **CCookie::GetNextValueAssoc** on the same object.

Example

See the Cookies Sample.

See Also

Reference

CCookie Class

CCookie::GetFirstValuePos

Other Resources

CCookie Members

CCookie::GetNextValueName

Call this method to get the name of a value based on its POSITION within the collection of values managed by this cookie.

```
const CStringA& GetNextValueName(
   POSITION& pos
) const throw( );
```

Parameters

pos

On entry, the position of a value in the collection. On exit, the position of the next value in the collection, or NULL if there are no further values.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The POSITION passed to this method should have been returned by a previous call to CCookie::GetFirstValuePos, CCookie::GetNextValueAssoc, or **CCookie::GetNextValueName** on the same object.

See Also

Reference

CCookie Class

CCookie::GetFirstValuePos

Other Resources

CCookie Members

CCookie::GetValue

Call this method to get the value from a cookie object.

```
BOOL GetValue(
   CStringA& strValue
) const throw( );
BOOL GetValue(
   LPSTR szBuff,
   DWORD* pdwSize
) const throw( );
```

Parameters

strValue

A reference to a string that will hold the value of the cookie on successful return of this method.

szBuff

A caller-allocated buffer that will hold the value of the cookie on successful return of this method.

pdwSize

A pointer to a DWORD variable. On entry, its value must be the size in bytes of the buffer *szBuff*. On exit, the value will be the length of the string copied to the buffer, not including the terminating null character.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This method returns the value passed to the constructor or set by calling CCookie::SetValue.

Example

See the Cookies Sample, the ForceLogin Sample, and the StencilCache Sample.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::GetValueAt

Call this method to get the value at the specified position.

```
const CStringA& GetValueAt(
   POSITION pos
) const throw( );
```

Parameters

pos

The position of a value in the collection.

Return Value

Returns the value at the specified position.

Remarks

The POSITION passed to this method can be obtained from a previous call to GetFirstValuePos, GetNextValueAssoc, or GetNextValueName.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::IsEmpty

Call this method to determine whether a cookie object is empty.

```
BOOL IsEmpty( ) const throw( );
```

Return Value

Returns TRUE if the cookie is empty, FALSE otherwise.

Remarks

A cookie is empty if it doesn't have a name.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::Lookup

Call this method to look up the value or a name-value pair applied to this cookie.

```
LPCSTR Lookup(
   LPCSTR szName = NULL
) const throw( );
```

Parameters

szName

The name for which you want the corresponding value.

Return Value

Returns the requested value if present or NULL if the name was not found or the cookie is empty.

Remarks

You can look up the value associated with a name for a multivalued cookie, or you can pass **NULL** to get the value of a single-valued cookie.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::ModifyAttribute

Call this method to modify an attribute-value pair associated with the cookie.

```
BOOL ModifyAttribute(
   LPCSTR szName,
   LPCSTR szValue
) throw( );
```

Parameters

szName

The name.

szValue

The value.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This method is equivalent to calling CCookie::AddAttribute. Both methods will add the attribute if not already present or change its value if it has already been applied to the cookie.

Use this method instead of **AddAttribute** to document the intentions of your call.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::ModifyValue

Call this method to modify a name-value pair associated with the cookie.

```
BOOL ModifyValue(
   LPCSTR szName,
   LPCSTR szValue
) throw( );
```

Parameters

szName

The name.

szValue

The value.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This method is equivalent to calling CCookie::AddValue. Both methods will add the name-value pair if not already present or change its value if it has already been applied to the cookie.

Use this method instead of **AddValue** to document the intentions of your call.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::Parse

Call this method to populate a **CCookie** from a cookie request header.

```
ATL_NOINLINE bool Parse(
   LPCSTR pstart
) throw( );
```

Parameters

pstart

The string containing the contents of the cookie.

Return Value

Returns true on success, false on failure.

Remarks

Call this method to create a **CCookie** from a buffer. The passed-in buffer should contain a cookie header retrieved from the **HTTP_COOKIE** request variable or a compatible string.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::RemoveAllAttributes

Call this method to remove all the attribute-value pairs from the collection of attributes managed by this cookie.

```
void RemoveAllAttributes( ) throw( );
```

Remarks

This method removes the attributes added using CCookie::AddAttribute or one of the more specific attribute methods.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::RemoveAllValues

Call this method to remove all the name-value pairs from the collection managed by this cookie.

```
void RemoveAllValues( ) throw( );
```

Remarks

This method removes the name-value pairs added using CCookie::AddValue.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::RemoveAttribute

Call this method to remove an attribute-value pair from the collection of attributes managed by this cookie.

```
BOOL RemoveAttribute(
   LPCSTR szName
) throw( );
```

Parameters

szName

The name.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This method removes the attributes added using CCookie::AddAttribute or one of the more specific attribute methods.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::RemoveValue

Call this method to remove a name-value pair from the collection managed by this cookie.

```
BOOL RemoveValue(
   LPCSTR szName
) throw( );
```

Parameters

szName

The name.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This method removes the name-value pairs added using CCookie::AddValue.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::Render

Call this method to put the correct HTTP representation of this cookie into a buffer.

```
ATL_NOINLINE BOOL Render(
   LPSTR szCookieBuffer,
   DWORD* pdwLen
) const throw( );
```

Parameters

szCookieBuffer

The cookie buffer or NULL. If this parameter is the cookie buffer, the value contained in *pdwLen* on exit will be the number of bytes transferred into the buffer. If NULL, the value contained in *pdwLen* on exit will be the required length of the buffer (not including the null-terminating byte).

pdwLen

On entry, *pdwLen* should point to a DWORD that indicates the size of the buffer in bytes. On exit, the DWORD contains the number of bytes transferred or available to be transferred into the buffer (not including the null-terminating byte).

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

On success, the buffer will contain the correct HTTP representation of the current cookie suitable for sending to a client as the body of a Set-Cookie header.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::SetComment

Call this method to set the Comment attribute of the cookie.

```
BOOL SetComment(
   LPCSTR szComment
) throw( );
```

Parameters

szComment

The comment.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The Comment attribute allows a web server to document its intended use of a cookie. This information can be displayed by supporting browsers so that the user of the web site can decide whether to initiate or continue a session with this cookie.

This attribute is optional.

RFC 2109 attribute. For more information on attribute version numbers, see the CCookie overview.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::SetCommentUrl

Call this method to set the CommentUrl attribute of the cookie.

```
BOOL SetCommentUrl(
   LPCSTR szUrl
) throw( );
```

Parameters

szUrl

The url.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The CommentUrl attribute allows a web server to document its intended use of a cookie using a URL that the user of the web site can navigate to. The URL specified here should not send further cookies to the client to avoid frustrating the user.

This attribute is optional.

RFC 2109 attribute. For more information on attribute version numbers, see the CCookie overview.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::SetDiscard

Call this method to add or remove the Discard attribute of the cookie.

```
BOOL SetDiscard(
   BOOL bDiscard
) throw( );
```

Parameters

bDiscard

Specifies whether to add or remove the attribute.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The Discard attribute does not have a value. Call **SetDiscard(TRUE)** to add the Discard attribute or **SetDiscard(FALSE)** to remove the Discard attribute.

Setting the Discard attribute tells a web browser that it should discard this cookie when the browser exits regardless of the value of the Max-Age attribute.

This attribute is optional.

When omitted, the default behavior is that the Max-Age attribute controls the lifetime of the cookie.

RFC 2109 attribute. For more information on attribute version numbers, see the CCookie overview.

Example

See the MantaWeb Sample.

See Also

Reference

CCookie Class

CCookie::SetMaxAge

Other Resources

CCookie Members

CCookie::SetDomain

Call this method to set the Domain attribute of the cookie.

```
BOOL SetDomain(
   LPCSTR szDomain
) throw( );
```

Parameters

szDomain

The domain.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The Domain attribute is used to indicate the domain to which the current cookie applies. Browsers should only send cookies back to the relevant domains.

This attribute is optional.

When omitted, the default behavior is for browsers to use the full domain of the server originating the cookie. You can set this attribute value explicitly if you want to share cookies among several servers.

Version 0 and RFC 2109 attribute. For more information on attribute version numbers, see the CCookie overview.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::SetExpires

Call this method to set the Expires attribute of the cookie.

```
BOOL SetExpires(
   const SYSTEMTIME& st
) throw( );
BOOL SetExpires(
   LPCSTR szExpires
) throw( );
```

Parameters

st

The time at which the cookie should expire expressed in Greenwich Mean Time (GMT).

szExpires

The time at which the cookie should expire expressed in Greenwich Mean Time (GMT) as a string in the following format: Wdy, DD-Mon-YY HH:MM:SS GMT.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The Expires attribute specifies an absolute date and time at which this cookie should be discarded by web browsers.

This attribute is optional.

When omitted, the default behavior is for browsers to discard cookies when the user closes the browser.

This attribute has been superceded in RFC 2109 by the Max-Age attribute, but you should continue to use this attribute for clients that only understand Version 0 attributes.

Version 0 attribute. For more information on attribute version numbers, see the CCookie overview.

Example

See the MantaWeb Sample and the Cookies Sample.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::SetMaxAge

Call this method to set the Max-Age attribute of the cookie.

```
BOOL SetMaxAge(
   UINT nMaxAge
) throw( );
```

Parameters

nMaxAge

The max age.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The value of the Max-Age attribute is a lifetime in seconds for the cookie. When the time has expired, compliant browsers will discard this cookie (if they haven't already done so as a result of the Discard attribute). If Max-Age is set to zero, the browser discards the cookie immediately.

This attribute is the RFC 2109 replacement for the Expires attribute.

This attribute is optional.

When omitted, the default behavior is for browsers to discard cookies when the user closes the browser.

RFC 2109 attribute. For more information on attribute version numbers, see the CCookie overview.

See Also

Reference

CCookie Class

CCookie::SetMaxAge

Other Resources

CCookie Members

CCookie::SetName

Call this method to set the name of a cookie.

```
BOOL SetName(
   LPCSTR szName
) throw( );
```

Parameters

szName

The new name of the cookie.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The name of a cookie cannot contain whitespace, semicolons, or commas. The name should not begin with a dollar sign ($) since such names are reserved for future use.

Example

See the MantaWeb Sample.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::SetPath

Call this method to set the Path attribute of the cookie.

```
BOOL SetPath(
   LPCSTR szPath
) throw( );
```

Parameters

szPath

The path.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The Path attribute specifies the subset of URLs to which this cookie applies. Only URLs that contain that path are allowed to read or modify the cookie.

This attribute is optional.

When omitted, the default behavior is for browsers to treat the path of a cookie as the path of the request URL that generated the Set-Cookie response, up to, but not including, the rightmost slash mark (/).

Version 0 and RFC 2109 attribute. For more information on attribute version numbers, see the CCookie overview.

Example

See the MantaWeb Sample and the Cookies Sample.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::SetPort

Call this method to set the Port attribute of the cookie.

```
BOOL SetPort(
   LPCSTR szPort
) throw( );
```

Parameters

szPort

The port.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The Port attribute specifies the port to which this cookie applies. Only URLs accessed through that port are allowed to read or modify the cookie.

This attribute is optional.

When omitted, the default behavior is for browsers to return the cookie through any port.

RFC 2109 attribute. For more information on attribute version numbers, see the CCookie overview.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::SetSecure

Call this method to add or remove the Secure attribute of the cookie.

```
BOOL SetSecure(
   BOOL bSecure
) throw( );
```

Parameters

bSecure

Specifies whether to add or remove the Secure attribute.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The Secure attribute does not have a value. Call **SetSecure(TRUE)** to add the Secure attribute or **SetSecure(FALSE)** to remove the Secure attribute.

Setting the Secure attribute tells a browser that it should transmit the cookie to the web server only through secure means such as HTTPS.

This attribute is optional.

When omitted, the default behavior is that the cookie will be sent through unsecured protocols.

Version 0 and RFC 2109 attribute. For more information on attribute version numbers, see the CCookie overview.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::SetValue

Call this method to set the value of this cookie.

```
BOOL SetValue(
   LPCSTR szValue
) throw( );
```

Parameters

szValue

The value.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Pass NULL to remove the cookie's value.

This method will fail if the cookie is multivalued. If necessary, call CCookie::RemoveAllValues to remove multiple values before calling this method.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie::SetVersion

Call this method to set the Version attribute of the cookie.

```
BOOL SetVersion(
   UINT nVersion
) throw( );
```

Parameters

nVersion

The version.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This attribute is required for cookies by RFC 2109 (http://ietf.org/rfc/rfc2109.txt) and must have a value of 1. However, you do not need to call **SetVersion** explicitly from your own code unless you need to force RFC 2109 compliance. **CCookie** will automatically set this attribute whenever you use an RFC 2109 attribute in your cookie.

RFC 2109 attribute. For more information on attribute version numbers, see the CCookie overview.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCookie Operators

For information about the operators in **CCookie**, see CCookie Members.

CCookie::operator =

The copy assignment operator.

```
CCookie& operator =(
   const CCookie& thatCookie
) throw(...);
```

Parameters

thatCookie

The cookie being assigned.

Return Value

Returns a reference to this cookie.

Remarks

Copies the name and value(s) from *thatCookie* into this cookie.

See Also

Reference

CCookie Class

Other Resources

CCookie Members

CCryptDerivedKey Class

This class represents a cryptographic session key that is derived from a hashed password.

```
class CCryptDerivedKey :
   public CCryptKey
```

Remarks

This class guarantees that two keys derived from the same password are identical.

Requirements

Header: atlcrypt.h

See Also

Reference

CCryptKey Class

Other Resources

CCryptDerivedKey Members

ATL Server Classes

CCryptDerivedKey Members

Methods

Initialize	Call this method to initialize the key.

See Also

Reference

CCryptDerivedKey Class

Other Resources

CCryptKey Members

CCryptDerivedKey Methods

For information about the methods in **CCryptDerivedKey**, see CCryptDerivedKey Members.

CCryptDerivedKey::Initialize

Call this method to initialize a cryptographic key derived from a password.

```
HRESULT Initialize(
   CCryptProv &Prov,
   CCryptHash &Hash,
   ALG_ID algid = CALG_RC4,
   DWORD dwFlags = CRYPT_EXPORTABLE
) throw( );
```

Parameters

Prov

A cryptographic service provider (CSP).

Hash

A hash object that has been fed the exact password data.

algid

The ALG_ID structure identifying the symmetric encryption algorithm for which the key is to be generated. Set to the RC4 stream cipher by default.

dwFlags

Specifies the type of key generated. The CRYPT_EXPORTABLE flag is set by default, meaning the session key can be transferred out of the CSP into a key BLOB. For a table of possible values, see the description of *dwFlags* in CryptDeriveKey.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method guarantees that two keys derived from the same password are identical.

See Also

Reference

CCryptDerivedKey Class
CryptDeriveKey

Other Resources

CCryptDerivedKey Members

CCryptHash Class

This class represents a generic hash.

```
class CCryptHash
```

Remarks

This class contains the base functionality of hashes.

Requirements

Header: atlcrypt.h

See Also

Tasks

MantaWeb Sample: Online Personal Information Management

Reference

ConfirmUser

ForceLogin

Other Resources

CCryptHash Members

ATL Server Classes

CCryptHash Members

Methods

AddData	Call this method to add data to the hash.
AddString	Call this method to add a string to the hash.
Attach	Call this method to attach this **CCryptHash** object to the specified hash.
CCryptHash	The constructor for the hash.
~CCryptHash	The destructor for the hash.
Destroy	Call this method to destroy the hash.
Detach	Call this method to detach a hash handle from the **CCryptHash** object.
Duplicate	Call this method to duplicate the hash.
GetAlgId	Call this method to get an identifier for the hash algorithm.
GetHandle	Call this method to get the hash handle.
GetParam	Call this method to retrieve information about the hash, such as the hash algorithm, the hash value, or the has h value size.
GetSize	Call this method to get the size of the hash.
GetValue	Call this method to get the hash value.
SetParam	Call this method to customize the operations of the hash in some way, such as by setting the hash value or sp ecifying the hash algorithm.
SetValue	Call this method to set the value of the hash.
Sign	Call this method to sign the hash.
Uninitialize	Call this method to destroy the hash.
VerifySignature	Call this method to verify the signature of the hash.

Static Data Members

EmptyHash	An empty hash.

See Also

Reference

CCryptHash Class

CCryptHash Methods

For information about the methods in **CCryptHash**, see CCryptHash Members.

CCryptHash::AddData

Call this method to add data to the hash.

```
HRESULT AddData(
   const BYTE * pbData,
   DWORD dwDataLen,
   DWORD dwFlags = 0
) throw( );
```

Parameters

pbData

Pointer to a buffer containing the data to be added to the hash.

dwDataLen

Number of bytes of data to be added. Must be 0 if the CRYPT_USERDATA flag is set.

dwFlags

Set the CRYPT_USERDATA flag to specify that the cryptographic service provider prompts the user to input data directly. In this case, the data is added to the hash, and the application is not allowed to access the data. This flag can be set to allow the user to enter a PIN into the system.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptHashData in the *Platform SDK*.

Example

See the MantaWeb Sample, the ForceLogin Sample, and the ConfirmUser Sample.

See Also

Reference

CCryptHash Class

Other Resources

CCryptHash Members

CCryptHash::AddString

Call this method to add a string to the hash.

```
HRESULT AddString(
   LPCTSTR szData,
   DWORD dwFlags = 0
) throw( );
```

Parameters

szData

The string to add.

dwFlags

Set the CRYPT_USERDATA flag to specify that the cryptographic service provider prompts the user to input data directly. In this case, the data is added to the hash, and the application is not allowed to access the data. This flag can be set to allow the user to enter a PIN into the system.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CCryptHash Class

CCryptHash::AddData

Other Resources

CCryptHash Members

CCryptHash::Attach

Call this method to attach this **CCryptHash** object to the specified hash.

```
void Attach(
   HCRYPTHASH hHash,
   BOOL bTakeOwnership = FALSE
) throw( );
```

Parameters

hHash

The hash to attach to.

bTakeOwnership

Specifies whether the attaching object takes ownership of the hash. If FALSE, the hash is duplicated.

Example

See the ForceLogin Sample.

See Also

Reference

CCryptHash Class

CCryptHash::Detach

Other Resources

CCryptHash Members

CCryptHash::CCryptHash

The constructor for the hash.

```
CCryptHash( ) throw( );
CCryptHash(
   const CCryptHash& hash
) throw( );
explicit CCryptHash(
   HCRYPTHASH hHash,
   BOOL bTakeOwnership = FALSE
) throw( );
```

Parameters

hash

The hash.

hHash

The handle to the hash.

bTakeOwnership

Specifies whether the new object takes ownership of the hash. If FALSE, the hash is duplicated.

See Also

Reference

CCryptHash Class

Other Resources

CCryptHash Members

CCryptHash::~CCryptHash

The destructor for the hash.

```
~CCryptHash( ) throw( );
```

See Also

Reference

CCryptHash Class

Other Resources

CCryptHash Members

CCryptHash::Destroy

Call this method to destroy the hash.

```
void Destroy( ) throw( );
```

Remarks

For more information, see CryptDestroyHash in the *Platform SDK*.

See Also

Reference

CCryptHash Class

CCryptHash::Uninitialize

Other Resources

CCryptHash Members

CCryptHash::Detach

Call this method to detach a hash handle from the **CCryptHash** object.

```
HCRYPTHASH Detach( ) throw( );
HRESULT Detach(
   HCRYPTHASH * phHash
) throw( );
```

Parameters

phHash

The hash.

Return Value

The first overload returns the hash handle. The second returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CCryptHash Class

CCryptHash::Attach

Other Resources

CCryptHash Members

CCryptHash::Duplicate

Call this method to duplicate the hash.

```
HCRYPTHASH Duplicate() const throw();
```

Return Value

The duplicated hash.

Remarks

For more information, see CryptDuplicateHash in the *Platform SDK*.

See Also

Reference

CCryptHash Class

Other Resources

CCryptHash Members

CCryptHash::GetAlgId

Call this method to get an identifier for the hash algorithm.

```
HRESULT GetAlgId(
   ALG_ID * pAlgId
) throw( );
```

Parameters

pAlgId

An ALG_ID structure specifying algorithm identifiers. For a list of possible values, see ALG_ID in the *Platform SDK*.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CCryptHash Class

CCryptHash::GetParam

Other Resources

CCryptHash Members

CCryptHash::GetHandle

Call this method to get the hash handle.

```
inline HCRYPTHASH GetHandle( );
```

Return Value

Returns the hash handle.

See Also

Reference

CCryptHash Class

Other Resources

CCryptHash Members

CCryptHash::GetParam

Call this method to retrieve information about the hash, such as the hash algorithm, the hash value, or the hash value size.

```
HRESULT GetParam(
   DWORD dwParam,
   BYTE * pbData,
   DWORD * pdwDataLen,
   DWORD dwFlags = 0
) throw( );
```

Parameters

dwParam

The type of information to retrieve. Specify HP_ALGID to retrieve the hash algorithm, HP_HASHSIZE to retrieve the hash value size, or HP_HASHVAL to retrieve the actual hash value. For more information, see CryptGetHashParam in the *Platform SDK*.

pbData

The buffer that receives the specified value data.

pdwDataLen

The size, in bytes, of the *pbData* buffer.

dwFlags

Reserved for future use and must be 0.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CCryptHash Class

CCryptHash::SetParam

Other Resources

CCryptHash Members

CCryptHash::GetSize

Call this method to get the size of the hash.

```
HRESULT GetSize(
   DWORD * pdwSize
) throw( );
```

Parameters

pdwSize

The number of bytes in the hash value. This value will usually be 16 or 20, depending on the hash algorithm.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See the MantaWeb Sample, the ForceLogin Sample, and the ConfirmUser Sample.

See Also

Reference

CCryptHash Class

CCryptHash::GetParam

Other Resources

CCryptHash Members

CCryptHash::GetValue

Call this method to get the hash value.

```
HRESULT GetValue(
   BYTE * pBuf,
   DWORD * pdwSize
) throw( );
```

Parameters

pBuf

The hash value.

pdwSize

The number of bytes in the hash value. This value will usually be 16 or 20, depending on the hash algorithm.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See the MantaWeb Sample, the ForceLogin Sample, and the ConfirmUser Sample.

See Also

Reference

CCryptHash Class
CCryptHash::SetValue
CCryptHash::GetParam

Other Resources

CCryptHash Members

CCryptHash::SetParam

Call this method to customize the operations of the hash in some way, such as by setting the hash value or specifying the hash algorithm.

```
HRESULT SetParam(
   DWORD dwParam,
   BYTE * pbData,
   DWORD dwFlags = 0
) throw( );
```

Parameters

dwParam

The parameter to set. Specify the value HP_HMAC_INFO to set the hash algorithm. Specify the value HP_HASHVAL to set the hash value. For more information, see CryptSetHashParam in the *Platform SDK*.

pbData

Contains information on the parameter to set. If *dwParam* is HP_HMAC_INFO, *pbData* should contain a pointer to an HMAC_INFO structure specifying the cryptographic hash algorithm and the inner and outer strings to be used. If *dwParam* is HP_ HASHVAL, *pbData* should contain a byte array containing a hash value.

dwFlags

Reserved for future use and must be 0.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CCryptHash Class

CCryptHash::GetParam

Other Resources

CCryptHash Members

CCryptHash::SetValue

Call this method to set the value of the hash.

```
HRESULT SetValue(
   BYTE * pBuf
) throw( );
```

Parameters

pBuf

The hash value.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CCryptHash Class
CCryptHash::GetValue
CCryptHash::SetParam

Other Resources

CCryptHash Members

CCryptHash::Sign

Call this method to sign the hash.

```
HRESULT Sign(
   BYTE * pbSignature,
   DWORD * pdwSigLen,
   DWORD dwFlags = 0,
   DWORD dwKeySpec = AT_SIGNATURE
) throw( );
```

Parameters

pbSignature

The signature.

pdwSigLen

The number of bytes in the signature.

dwFlags

See *dwFlags* in CryptSignHash.

dwKeySpec

Identifies the private key to use from the provider's container. It can be AT_KEYEXCHANGE or AT_SIGNATURE. See *dwKeySpec* in CryptSignHash.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptSignHash in the *Platform SDK*.

See Also

Reference

CCryptHash Class

Other Resources

CCryptHash Members

CCryptHash::Uninitialize

Call this method to destroy the hash.

```
HRESULT Uninitialize( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Unlike CCryptHash::Destroy, this method can fail, in which case the hash is still valid.

For more information, see CryptDestroyHash in the *Platform SDK*.

See Also

Reference

CCryptHash Class

Other Resources

CCryptHash Members

CCryptHash::VerifySignature

Call this method to verify the signature of the hash.

```
HRESULT VerifySignature(
   const BYTE * pbSignature,
   DWORD pdwSignLen,
   CCryptKey &PubKey,
   DWORD dwFlags = 0
) throw();
```

Parameters

pbSignature

The signature.

pdwSignLen

The number of bytes in the signature.

PubKey

The public key used to authenticate the signature.

dwFlags

Specify CRYPT_NOHASHOID if you do not expect the hash OID to be present or checked.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptVerifySignature in the *Platform SDK*.

See Also

Reference

CCryptHash Class

Other Resources

CCryptHash Members

CCryptHash Data Members

For information about the static data members in **CCryptHash**, see CCryptHash Members.

CCryptHash::EmptyHash

An empty hash.

```
static CCryptHash EmptyHash;
```

See Also

Reference

CCryptHash Class

Other Resources

CCryptHash Members

CCryptHMACHash Class

This class represents a Hashed Message Authentication Code (HMAC) keyed hash.

```
class CCryptHMACHash :
   public CCryptHash
```

Requirements

Header: atlcrypt.h

See Also

Reference

CCryptHash Class

Other Resources

CCryptHMACHash Members

ATL Server Classes

CCryptHMACHash Members

Methods

Initialize	Call this method to initialize the HMAC hash.

See Also

Reference

CCryptHMACHash Class

Other Resources

CCryptHash Members

CCryptHMACHash Methods

For information about the methods in **CCryptHMACHash**, see CCryptHMACHash Members.

CCryptHMACHash::Initialize

Call this method to initialize the Hashed Message Authentication Code (HMAC) hash.

```
HRESULT Initialize(
   CCryptProv &Prov,
   CCryptKey &Key,
   LPCTSTR szText = NULL
) throw( );
```

Parameters

Prov

The cryptographic service provide (CSP).

Key

The key for the hash.

szText

The string to add to the hash.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CCryptHMACHash Class

CryptCreateHash

Other Resources

CCryptHMACHash Members

CCryptImportKey Class

This class represents a cryptographic key created by importing a key BLOB.

```
class CCryptImportKey :
   public CCryptKey
```

Remarks

This class can be used to import a regular session key, public key, or public/private key pair. For all but the public key, the key or key pair is encrypted.

Requirements

Header: atlcrypt.h

See Also

Reference

CCryptKey Class

Other Resources

CCryptImportKey Members

ATL Server Classes

CCryptImportKey Members

Methods

Initialize	Call this method to initialize a cryptographic key formed from a key BLOB.

See Also

Reference

CCryptImportKey Class

Other Resources

CCryptKey Members

CCryptImportKey Methods

For information about the methods in **CCryptImportKey**, see CCryptImportKey Members.

CCryptImportKey::Initialize

Call this method to initialize a cryptographic key formed from a key BLOB.

```
HRESULT Initialize(
   CCryptProv &Prov,
   BYTE * pbData,
   DWORD dwDataLen,
   CCryptKey &PubKey,
   DWORD dwFlags) throw();
```

Parameters

Prov

A cryptographic service provider (CSP).

pbData

The key BLOB. You can create a key BLOB using CCryptKey::ExportSimpleBlob, CCryptKey::ExportPublicKeyBlob, or CCryptKey::ExportPrivateKeyBlob.

dwDataLen

The number of bytes in the key BLOB.

PubKey

See *hPubKey* in CryptImportKey in the *Platform SDK*.

dwFlags

See *dwFlags* in **CryptImportKey** in the *Platform SDK*.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptImportKey in the *Platform SDK*.

See Also

Reference

CCryptImportKey Class

Other Resources

CCryptImportKey Members

CCryptKey Class

This class represents a generic cryptographic key.

```
class CCryptKey
```

Remarks

This class provides the basic functionality for cryptographic keys such as encrypting and decrypting.

Requirements

Header: atlcrypt.h

See Also

Other Resources

CCryptKey Members

ATL Server Classes

CCryptKey Members

Methods

Attach	Call this method to attach this **CCryptKey** object to the specified cryptographic key.
CCryptKey	The constructor for the cryptographic key.
~CCryptKey	The destructor for the cryptographic key.
Decrypt	Call this method to decrypt data using this cryptographic key.
Destroy	Call this method to destroy the cryptographic key.
Detach	Call this method to detach a cryptographic key from the **CCryptKey** object.
Duplicate	Call this method to create an exact copy of the cryptographic key and the key's state.
Encrypt	Call this method to encrypt data using this cryptographic key.
EncryptString	Call this method to encrypt a string using this cryptographic key.
ExportPrivateKeyBlob	Call this method to export a public/private key pair from a cryptographic service provider (CSP) in a secure manner.
ExportPublicKeyBlob	Call this method to export public keys from a cryptographic service provider (CSP) in a secure manner.
ExportSimpleBlob	Call this method to export session keys from a cryptographic service provider (CSP) in a secure manner.
GetAlgId	Call this method to get the algorithm that was used to generate the cryptographic key.
GetBlockLength	Call this method to get the cryptographic key cipher's block length.
GetEffKeyLen	Call this method to get the effective key length of an RC2 cryptographic key.
GetG	Call this method to get the generator G from the DSS key **BLOB**.
GetHandle	Call this method to get the cryptographic key handle.
GetIV	Call this method to get the initialization vector for the cryptographic key.
GetKeyLength	Call this method to get the cryptographic key length.
GetMode	Call this method to get the cipher mode for the cryptographic key.
GetModeBits	Call this method to get the number of bits that are processed per cycle when the OFB or CFB cipher mode of the cryptographic key is used.
GetP	Call this method to get the prime modulus P of a DSS key **BLOB**.
GetPadding	Call this method to get the padding mode used by the cipher created by the cryptographic key.

GetParam	Call this method to retrieve parameters that govern the operations of the cryptographic key.
GetPermissions	Call this method to get the cryptographic key permissions.
GetQ	Call this method to get the prime Q from the DSS key **BLOB**.
GetSalt	Call this method to get the salt value in the cryptographic key.
SetAlgId	Call this method to enable a cryptographic session key by setting its algorithm type.
SetEffKeyLen	Call this method to set the effective key length of an RC2 cryptographic key.
SetG	Call this method to set the generator G from the DSS key **BLOB**.
SetIV	Call this method to set the initialization vector for the cryptographic key.
SetMode	Call this method to set the cipher mode for the cryptographic key.
SetModeBits	Call this method to set the number of bits that are processed per cycle when the OFB or CFB cipher mod e of the cryptographic key is used.
SetP	Call this method to set the prime modulus P of a DSS key **BLOB**.
SetPadding	Call this method to set the padding mode used by the cipher created by the cryptographic key.
SetParam	Call this method to customize the operations of the cryptographic key.
SetPermissions	Call this method to set the cryptographic key permissions.
SetQ	Call this method to set the prime Q from the DSS key **BLOB**.
SetSalt	Call this method to set the new salt value from a byte array.
SetSaltEx	Call this method to set the new salt value from a BLOB.
SetX	Call this method to generate the X (secret exponent) and Y (public key) values for the cryptographic key after setting the P, Q, and G values (with SetP, SetQ, and SetG).
Uninitialize	Call this method to destroy the cryptographic key.

Data Members

EmptyKey	An empty key.

See Also

Reference

CCryptKey Class

CCryptKey Methods

For information about the methods in **CCryptKey**, see CCryptKey Members.

CCryptKey::Attach

Call this method to attach this **CCryptKey** object to the specified cryptographic key.

```
void Attach(
   HCRYPTKEY hKey,
     BOOL bTakeOwnership = FALSE
) throw();
```

Parameters

hKey

The cryptographic key.

bTakeownership

Specifies whether the attaching object takes ownership of the key. If FALSE, the hash is duplicated.

See Also

Reference

CCryptKey Class

CCryptKey::Detach

Other Resources

CCryptKey Members

CCryptKey::CCryptKey

The constructor for the cryptographic key.

```
CCryptKey( ) throw( );
explicit CCryptKey(
   HCRYPTKEY hKey,
   BOOL bTakeOwnership = FALSE
) throw( );
CCryptKey(
   const CCryptKey& key
) throw( );
```

Parameters

hKey

The cryptographic key.

bTakeOwnership

Specifies whether the new object takes ownership of the key. If FALSE, the hash is duplicated.

key

The cryptographic key.

See Also

Reference

CCryptKey Class

Other Resources

CCryptKey Members

CCryptKey::~CCryptKey

The destructor for the cryptographic key.

```
~CCryptKey( ) throw( );
```

See Also

Reference

CCryptKey Class

Other Resources

CCryptKey Members

CCryptKey::Decrypt

Call this method to decrypt data using this cryptographic key.

```
HRESULT Decrypt(
   const BYTE * pbCipherText,
   DWORD dwCipherTextLen,
   BYTE * pbPlainText,
   DWORD * pdwPlainTextLen,
   CCryptHash & Hash = CCryptHash::EmptyHash
) throw( );
HRESULT Decrypt(
   BOOL final,
   BYTE * pbData,
   DWORD * pdwDataLen,
   CCryptHash & Hash = CCryptHash::EmptyHash
) throw( );
```

Parameters

pbCipherText

The cipher text.

dwCipherTextLen

The number of bytes in the cipher text.

pbPlainText

The plain text.

pdwPlainTextLen

The number of bytes in the plain text.

Hash

The hash. For more information, see the description of *hHash* in CryptDecrypt.

final

Specifies whether this is the last section in a series being decrypted. This value is TRUE if this is the last or only block. If it is not the last block, it is FALSE.

pbData

The buffer holding the data to be decrypted. After the decryption has been performed, the plain text is placed back in this same buffer.

pdwDataLen

The number of bytes in the *pbData* buffer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptDecrypt in the *Platform SDK*.

See Also

Reference

CCryptKey Class

CCryptKey::Encrypt

Other Resources

CCryptKey Members

CCryptKey::Destroy

Call this method to destroy the cryptographic key.

```
void Destroy( ) throw( );
```

Remarks

For more information, see CryptDestroyKey in the *Platform SDK*.

See Also

Reference

CCryptKey Class

CCryptKey::Uninitialize

Other Resources

CCryptKey Members

CCryptKey::Detach

Call this method to detach a cryptographic key from the **CCryptKey** object.

```
HCRYPTKEY Detach( ) throw( );
```

Return Value

Returns the cryptographic key handle.

See Also

Reference

CCryptKey Class

CCryptKey::Attach

Other Resources

CCryptKey Members

CCryptKey::Duplicate

Call this method to create an exact copy of the cryptographic key and the key's state.

```
HCRYPTKEY Duplicate( ) const throw( );
```

Return Value

Returns a handle to the duplicate key, or NULL on failure.

Remarks

For more information, see CryptDuplicateKey.

See Also

Reference

CCryptKey Class

Other Resources

CCryptKey Members

CCryptKey::Encrypt

Call this method to encrypt data using this cryptographic key.

```
HRESULT Encrypt(
   const BYTE * pbPlainText,
   DWORD dwPlainTextLen,
   BYTE * pbCipherText,
   DWORD * pdwCipherTextLen,
   CCryptHash & Hash = CCryptHash::EmptyHash
) throw( );
HRESULT Encrypt(
   BOOL final,
   BYTE * pbData,
   DWORD * pdwDataLen,
   DWORD dwBufLen,
   CCryptHash & Hash = CCryptHash::EmptyHash
) throw( );
```

Parameters

pbPlainText

The plain text.

dwPlainTextLen

The number of bytes in the plain text.

pbCipherText

The cipher text.

pdwCipherTextLen

The number of bytes in the cipher text.

Hash

The hash. For more information, see the description of *hHash* in CryptEncrypt.

final

Specifies whether this is the last section in a series being encrypted. Set to TRUE for the last or only block and FALSE if there are more blocks to be encrypted.

pbData

The buffer holding the data to be encrypted. The encrypted data overwrites the data to be encrypted in this buffer.

pdwDataLen

The number of bytes in the *pbData* buffer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptEncrypt in the *Platform SDK*.

See Also

Reference

CCryptKey Class
CCryptKey::Decrypt
CCryptKey::EncryptString

Other Resources

CCryptKey Members

CCryptKey::EncryptString

Call this method to encrypt a string using this cryptographic key.

```
HRESULT EncryptString(
   LPCTSTR szPlainText,
   BYTE * pbCipherText,
   DWORD * pdwCipherTextLen,
   CCryptHash & Hash = CCryptHash::EmptyHash
) throw( );
```

Parameters

szPlainText

The plain text.

pbCipherText

The cipher text.

pdwCipherTextLen

The number of bytes in the cipher text.

Hash

The hash. For more information, see the description of *hHash* in CryptEncrypt.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptEncrypt in the *Platform SDK*.

See Also

Reference

CCryptKey Class

CCryptKey::Encrypt

Other Resources

CCryptKey Members

CCryptKey::ExportPrivateKeyBlob

Call this method to export a public/private key pair from a cryptographic service provider (CSP) in a secure manner.

```
HRESULT ExportPrivateKeyBlob(
   CCryptKey & ExpKey,
   DWORD dwFlags,
   BYTE * pbData,
   DWORD * pdwDataLen
) throw( );
```

Parameters

ExpKey

The cryptographic key of the destination user. See the *hExpKey* description in CryptExportKey in the *Platform SDK*.

dwFlags

See the *dwFlags* description in **CryptExportKey**.

pbData

The buffer to receive the key **BLOB** data.

pdwDataLen

The number of bytes in the *pbData* buffer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptExportKey in the *Platform SDK*.

See Also

Reference

CCryptKey Class
CCryptKey::ExportPublicKeyBlob
CCryptKey::ExportSimpleBlob

Other Resources

CCryptKey Members

CCryptKey::ExportPublicKeyBlob

Call this method to export public keys from a cryptographic service provider (CSP) in a secure manner.

```
HRESULT ExportPublicKeyBlob(
   CCryptKey &ExpKey,
   DWORD dwFlags,
   BYTE * pbData,
   DWORD * pdwDataLen
) throw( );
```

Parameters

ExpKey

The cryptographic key of the destination user. See the *hExpKey* description in CryptExportKey in the *Platform SDK*.

dwFlags

See the *dwFlags* description in **CryptExportKey**.

pbData

The buffer to receive the key **BLOB** data.

pdwDataLen

The number of bytes in the *pbData* buffer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptExportKey in the *Platform SDK*.

See Also

Reference

CCryptKey Class
CCryptKey::ExportPrivateKeyBlob
CCryptKey::ExportSimpleBlob

Other Resources

CCryptKey Members

CCryptKey::ExportSimpleBlob

Call this method to export session keys from a cryptographic service provider (CSP) in a secure manner.

```
HRESULT ExportSimpleBlob(
   CCryptKey &ExpKey,
   DWORD dwFlags,
   BYTE * pbData,
   DWORD * pdwDataLen
) throw( );
```

Parameters

ExpKey

The cryptographic key of the destination user. See the *hExpKey* description in CryptExportKey in the *Platform SDK*.

dwFlags

See the *dwFlags* description in **CryptExportKey**.

pbData

The buffer to receive the key **BLOB** data.

pdwDataLen

The number of bytes in the *pbData* buffer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptExportKey in the *Platform SDK*.

See Also

Reference

CCryptKey Class
CCryptKey::ExportPrivateKeyBlob
CCryptKey::ExportPublicKeyBlob

Other Resources

CCryptKey Members

CCryptKey::GetAlgId

Call this method to get the algorithm that was used to generate the cryptographic key.

```
HRESULT GetAlgId(
   ALG_ID * pAlgId
) throw( );
```

Parameters

pAlgId

An ALG_ID structure specifying algorithm identifiers. For a list of possible values, see ALG_ID in the *Platform SDK*.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_ALGID value for the *dwParam* parameter in CryptGetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::SetAlgId

Other Resources

CCryptKey Members

CCryptKey::GetBlockLength

Call this method to get the cryptographic key cipher's block length.

```
HRESULT GetBlockLength(
   DWORD * pdwBlockLen
) throw( );
```

Parameters

pdwBlockLen

The key cipher's block length.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_BLOCKLEN value for the *dwParam* parameter in CryptGetKeyParam.

See Also

Reference

CCryptKey Class

Other Resources

CCryptKey Members

CCryptKey::GetEffKeyLen

Call this method to get the effective key length of an RC2 cryptographic key.

```
HRESULT GetEffKeyLen(
   DWORD * pdwEffKeyLen
) throw( );
```

Parameters

pdwEffKeyLen

The effective key length.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_EFFECTIVE_KEYLEN value for the *dwParam* parameter in CryptGetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::SetEffKeyLen

Other Resources

CCryptKey Members

CCryptKey::GetG

Call this method to get the generator G from the DSS key **BLOB**.

```
HRESULT GetG(
   BYTE * pbG,
   DWORD * pdwLength
) throw( );
```

Parameters

pbG

The generator G.

pdwLength

The number of bytes in *pbG*.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_G value for the *dwParam* parameter in CryptGetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::SetG

Other Resources

CCryptKey Members

CCryptKey::GetHandle

Call this method to get the cryptographic key handle.

```
inline HCRYPTKEY GetHandle( ) throw( );
```

Return Value

Returns the handle.

See Also

Reference

CCryptKey Class

Other Resources

CCryptKey Members

CCryptKey::GetIV

Call this method to get the initialization vector for the cryptographic key.

```
HRESULT GetIV(
   BYTE * pbIV,
   DWORD * pdwLength
) throw( );
```

Parameters

pbIV

The initialization vector.

pdwLength

The number of bytes in *pbIV*.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_IV value for the *dwParam* parameter in CryptGetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::SetIV

Other Resources

CCryptKey Members

CCryptKey::GetKeyLength

Call this method to get the cryptographic key length.

```
HRESULT GetKeyLength(
   DWORD * pdwKeyLen
) throw( );
```

Parameters

pdwKeyLen

The length of the key in bits.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_KEYLEN value for the *dwParam* parameter in CryptGetKeyParam.

See Also

Reference

CCryptKey Class

Other Resources

CCryptKey Members

CCryptKey::GetMode

Call this method to get the cipher mode for the cryptographic key.

```
HRESULT GetMode(
   DWORD * pdwMode
) throw( );
```

Parameters

pdwMode

The cipher mode. For a list of valid cipher modes, see CryptGetKeyParam.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_MODE value for the *dwParam* parameter in CryptGetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::SetMode

Other Resources

CCryptKey Members

CCryptKey::GetModeBits

Call this method to get the number of bits that are processed per cycle when the OFB or CFB cipher mode of the cryptographic key is used.

```
HRESULT GetModeBits(
   DWORD * pdwModeBits
) throw( );
```

Parameters

pdwModeBits

The number of bits.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_MODE_BITS value for the *dwParam* parameter in CryptGetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::SetModeBits

Other Resources

CCryptKey Members

CCryptKey::GetP

Call this method to get the prime modulus P of a DSS key **BLOB**.

```
HRESULT GetP(
   BYTE * pbP,
   DWORD * pdwLength
) throw( );
```

Parameters

pbP

The prime modulus P.

pdwLength

The number of bytes in *pbP*.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_P value for the *dwParam* parameter in CryptGetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::SetP

Other Resources

CCryptKey Members

CCryptKey::GetPadding

Call this method to get the padding mode used by the cipher created by the cryptographic key.

```
HRESULT GetPadding(
   DWORD * pdwPadding
) throw( );
```

Parameters

pdwPadding

The padding method used by the cipher.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_PADDING value for the *dwParam* parameter in CryptGetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::SetPadding

Other Resources

CCryptKey Members

CCryptKey::GetParam

Call this method to retrieve parameters that govern the operations of the cryptographic key.

```
HRESULT GetParam(
   DWORD dwParam,
   BYTE * pbData,
   DWORD * pdwDataLen,
   DWORD dwFlags = 0
) throw( );
```

Parameters

dwParam

Specifies the nature of the query. For a table of possible values, see the description of the *dwParam* parameter in CryptGetKeyParam.

pbData

The buffer that will receive the returned data.

pdwDataLen

The length, in bytes, of the *pbData* buffer.

dwFlags

The flags that give more information about the type of information to return. For possible values, see the description of the *dwFlags* parameter fin **CryptGetKeyParam**.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptGetKeyParam in the *Platform SDK*.

See Also

Reference

CCryptKey Class
CCryptKey::SetParam

Other Resources

CCryptKey Members

CCryptKey::GetPermissions

Call this method to get the cryptographic key permissions.

```
HRESULT GetPermissions(
   DWORD * pdwPerms
) throw( );
```

Parameters

pdwPerms

A DWORD with zero or more permission flags set. For a description of these flags, see CryptGetKeyParam.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_PERMISSIONS value for the *dwParam* parameter in CryptGetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::SetPermissions

Other Resources

CCryptKey Members

CCryptKey::GetQ

Call this method to get the prime Q from the DSS key **BLOB**.

```
HRESULT GetQ(
   BYTE * pbQ,
   DWORD * pdwLength
) throw( );
```

Parameters

pbQ

The prime Q.

pdwLength

The number of bytes in *pbQ*.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_Q value for the *dwParam* parameter in CryptGetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::SetQ

Other Resources

CCryptKey Members

CCryptKey::GetSalt

Call this method to get the salt value in the cryptographic key.

```
HRESULT GetSalt(
   BYTE * pbSalt,
   DWORD * pdwLength
) throw( );
```

Parameters

pbSalt

The salt value.

pdwLength

The number of bytes in *pbSalt*.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Salt values make up a portion of many session keys but are not a part of public/private key pairs. For more information, see the description of the KP_SALT value for the *dwParam* parameter in CryptGetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::SetSalt

Other Resources

CCryptKey Members

CCryptKey::SetAlgId

Call this method to enable a cryptographic session key by setting its algorithm type.

```
HRESULT SetAlgId(
   ALG_ID AlgId,
   DWORD dwFlags
) throw( );
```

Parameters

AlgId

An ALG_ID structure specifying algorithm identifiers. For a list of possible values, see ALG_ID in the *Platform SDK*.

dwFlags

The *dwFlags* parameter is used to pass in flag values for the enabled key. The *dwFlags* parameter can hold values such as the key size and the other flag values allowed when generating the same type of key with **CryptGenKey**. For information on allowable flag values, see CryptGenKey.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_ALGID value for the *dwParam* parameter in CryptGetKeyParam.

See Also

Reference

CCryptKey Class
CCryptKey::GetAlgId

Other Resources

CCryptKey Members

CCryptKey::SetEffKeyLen

Call this method to set the effective key length of an RC2 cryptographic key.

```
HRESULT SetEffKeyLen(
   DWORD dwEffKeyLen
) throw( );
```

Parameters

dwEffKeyLen

The effective key length.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_EFFECTIVE_KEYLEN value for the *dwParam* parameter in CryptSetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::GetEffKeyLen

Other Resources

CCryptKey Members

CCryptKey::SetG

Call this method to set the generator G from the DSS key **BLOB**.

```
HRESULT SetG(
   BYTE * pbG,
   DWORD dwLength
) throw( );
HRESULT SetG(
   _CRYPTOAPI_BLOB * pBlobG
) throw( );
```

Parameters

pbG

The generator G.

dwLength

The number of bytes in *pbG*.

pBlobG

The CryptoAPI **BLOB** structure whose **pbData** member contains the generator G value and whose **cbData** member contains the length of the value.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_G value for the *dwParam* parameter in CryptSetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::GetG

Other Resources

CCryptKey Members

CCryptKey::SetIV

Call this method to set the initialization vector for the cryptographic key.

```
HRESULT SetIV(
   BYTE * pbIV
) throw( );
```

Parameters

pbIV

The initialization vector.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_IV value for the *dwParam* parameter in CryptSetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::GetIV

Other Resources

CCryptKey Members

CCryptKey::SetMode

Call this method to set the cipher mode for the cryptographic key.

```
HRESULT SetMode(
   DWORD dwMode
) throw( );
```

Parameters

dwMode

The cipher mode. For a list of valid cipher modes, see CryptGetKeyParam.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_MODE value for the *dwParam* parameter in CryptSetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::GetMode

Other Resources

CCryptKey Members

CCryptKey::SetModeBits

Call this method to set the number of bits that are processed per cycle when the OFB or CFB cipher mode of the cryptographic key is used.

```
HRESULT SetModeBits(
   DWORD dwModeBits
) throw( );
```

Parameters

dwModeBits

The number of bits.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_MODE_BITS value for the *dwParam* parameter in CryptSetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::GetModeBits

Other Resources

CCryptKey Members

CCryptKey::SetP

Call this method to set the prime modulus P of a DSS key **BLOB**.

```
HRESULT SetP(
   BYTE * pbP,
   DWORD dwLength
) throw( );
HRESULT SetP(
   _CRYPTOAPI_BLOB * pBlobP
) throw( );
```

Parameters

pbP

The prime modulus P.

dwLength

The number of bytes in *pbP*.

pBlobP

The CryptoAPI **BLOB** structure whose **pbData** member contains the prime modulus P value and whose **cbData** member contains the length of the value.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_P value for the *dwParam* parameter in CryptSetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::GetP

Other Resources

CCryptKey Members

CCryptKey::SetPadding

Call this method to set the padding mode used by the cipher created by the cryptographic key.

```
HRESULT SetPadding(
   DWORD dwPadding
) throw( );
```

Parameters

dwPadding

The padding method used by the cipher.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_PADDING value for the *dwParam* parameter in CryptSetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::GetPadding

Other Resources

CCryptKey Members

CCryptKey::SetParam

Call this method to customize the operations of the cryptographic key.

```
HRESULT SetParam(
   DWORD dwParam,
   BYTE * pbData,
   DWORD dwFlags = 0
) throw( );
```

Parameters

dwParam

The parameter to set. For a table of possible values, see the description of the *dwParam* parameter in CryptSetKeyParam.

pbData

The buffer containing the value to set.

dwFlags

Used only when *dwParam* is KP_ALGID. The *dwFlags* parameter is used to pass in flag values for the enabled key. The *dwFlags* parameter can hold values such as the key size and the other flag values allowed when generating the same type of key with **CryptGenKey**. For information on allowable flag values, see CryptGenKey.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The values set by this method are not persisted to memory and can only be used within a single session.

See Also

Reference

CCryptKey Class

CCryptKey::GetParam

Other Resources

CCryptKey Members

CCryptKey::SetPermissions

Call this method to set the cryptographic key permissions.

```
HRESULT SetPermissions(
   DWORD dwPerms
) throw( );
```

Parameters

dwPerms

A DWORD with zero or more permission flags set. For a description of these flags, see CryptSetKeyParam.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_PERMISSIONS value for the *dwParam* parameter in CryptSetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::GetPermissions

Other Resources

CCryptKey Members

CCryptKey::SetQ

Call this method to set the prime Q from the DSS key **BLOB**.

```
HRESULT SetQ(
   _CRYPTOAPI_BLOB * pBlobQ
) throw( );
HRESULT SetQ(
   BYTE * pbQ,
   DWORD dwLength
) throw( );
```

Parameters

pBlobQ

The CryptoAPI **BLOB** structure whose **pbData** member contains the prime Q value and whose **cbData** member contains the length of the value.

pbQ

The prime Q.

dwLength

The number of bytes in *pbQ*.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_Q value for the *dwParam* parameter in CryptSetKeyParam.

See Also

Reference

CCryptKey Class

CCryptKey::GetQ

Other Resources

CCryptKey Members

CCryptKey::SetSalt

Call this method to set the new salt value from a byte array.

```
HRESULT SetSalt(
   BYTE * pbSalt
) throw( );
```

Parameters

pbSalt

The salt value.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Salt values make up a portion of many session keys but are not a part of public/private key pairs. For more information, see the description of the KP_SALT value for the *dwParam* parameter in CryptSetKeyParam.

SetSalt is provided for backward compatibility; newer applications should use SetSaltEx.

See Also

Reference

CCryptKey Class
CCryptKey::GetSalt
CCryptKey::SetSaltEx
Specifying a Salt Value

Other Resources

CCryptKey Members

CCryptKey::SetSaltEx

Call this method to set the new salt value from a BLOB.

```
HRESULT SetSaltEx(
   _CRYPTOAPI_BLOB * pBlobSalt
) throw( );
```

Parameters

pBlobSalt

The CryptoAPI **BLOB** structure whose **pbData** member contains the salt value and whose **cbData** member contains the length of the value.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Salt values make up a portion of many session keys but are not a part of public/private key pairs. For more information, see the description of the KP_SALT_EX value for the *dwParam* parameter in CryptSetKeyParam.

SetSalt is provided for backward compatibility; newer applications should use **SetSaltEx**.

See Also

Reference

CCryptKey Class
CCryptKey::SetSalt
Specifying a Salt Value

Other Resources

CCryptKey Members

CCryptKey::SetX

Call this method to generate the X (secret exponent) and Y (public key) values for the cryptographic key after setting the P, Q, and G values (with SetP, SetQ, and SetG).

```
HRESULT SetX( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the KP_X value for the *dwParam* parameter in CryptSetKeyParam.

See Also

Reference

CCryptKey Class

Other Resources

CCryptKey Members

CCryptKey::Uninitialize

Call this method to destroy the cryptographic key.

```
HRESULT Uninitialize( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Unlike CCryptKey::Destroy, this method can fail, in which case the hash is still valid.

For more information, see CryptDestroyKey.

See Also

Reference

CCryptKey Class

Other Resources

CCryptKey Members

CCryptKey Data Members

For information about the static data members in **CCryptKey**, see CCryptKey Members.

CCryptKey::EmptyKey

An empty key.

```
static CCryptKey EmptyKey;
```

See Also

Reference

CCryptKey Class

Other Resources

CCryptKey Members

CCryptKeyedHash Class

This class represents a keyed hash, such as a Hashed Message Authentication Code (HMAC) or a Message Authentication Code (MAC) hash.

```
class CCryptKeyedHash :
   public CCryptHash
```

Requirements

Header: atlcrypt.h

See Also

Reference

CCryptHash Class

Other Resources

CCryptKeyedHash Members

ATL Server Classes

CCryptKeyedHash Members

Methods

Initialize	Call this method to initialize the keyed hash.

See Also

Reference

CCryptKeyedHash Class

Other Resources

CCryptHash Members

CCryptKeyedHash Methods

For information about the methods in **CCryptKeyedHash**, see CCryptKeyedHash Members.

CCryptKeyedHash::Initialize

Call this method to initialize the keyed hash.

```
HRESULT Initialize(
   CCryptProv &Prov,
   ALG_ID Algid,
   CCryptKey &Key,
   DWORD dwFlags
) throw( );
```

Parameters

Prov

The cryptographic service provide (CSP).

Algid

Specify CALG_HMAC for a Hashed Message Authentication Code (HMAC) keyed hash or CALG_MAC for a Message Authentication Code (MAC) keyed hash. For more information, see ALG_ID in the *Platform SDK*.

Key

The key for the hash.

dwFlags

Reserved for future use and must be zero.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CCryptKeyedHash Class

CryptCreateHash

Other Resources

CCryptKeyedHash Members

CCryptMACHash Class

This class represents a Message Authentication Code (MAC) keyed hash.

```
class CCryptMACHash :
   public CCryptHash
```

Remarks

A MAC hash is believed to be less secure than a Hashed Message Authentication Code (HMAC) hash.

Requirements

Header: atlcrypt.h

See Also

Reference

CCryptHash Class

Other Resources

CCryptMACHash Members

ATL Server Classes

CCryptMACHash Members

Methods

Initialize	Call this method to initialize the MAC hash.

See Also

Reference

CCryptMACHash Class

Other Resources

CCryptHash Members

CCryptMACHash Methods

For information about the methods in **CCryptMACHash**, see CCryptMACHash Members.

CCryptMACHash::Initialize

Call this method to initialize the Message Authentication Code (MAC) hash.

```
HRESULT Initialize(
   CCryptProv &Prov,
   CCryptKey &Key,
   LPCTSTR szText = NULL
) throw( );
```

Parameters

Prov

The cryptographic service provide (CSP).

Key

The key for the hash.

szText

The string to add to the hash.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CCryptMACHash Class

CryptCreateHash

Other Resources

CCryptMACHash Members

CCryptMD2Hash Class

This class represents an MD2 hash.

```
class CCryptMD2Hash :
   public CCryptHash
```

Requirements

Header: atlcrypt.h

See Also

Reference

CCryptHash Class

Other Resources

CCryptMD2Hash Members

ATL Server Classes

CCryptMD2Hash Members

Methods

Initialize	Call this method to initialize the MD2 hash.

See Also

Reference

CCryptMD2Hash Class

Other Resources

CCryptHash Members

CCryptMD2Hash Methods

For information about the methods in **CCryptMD2Hash**, see CCryptMD2Hash Members.

CCryptMD2Hash::Initialize

Call this method to initialize the MD2 hash.

```
HRESULT Initialize(
   CCryptProv &Prov,
   LPCTSTR szText = NULL
) throw( );
```

Parameters

Prov

The cryptographic service provide (CSP).

szText

The string to add to the hash.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CCryptMD2Hash Class

CryptCreateHash

Other Resources

CCryptMD2Hash Members

CCryptMD4Hash Class

This class represents an MD4 hash.

```
class CCryptMD4Hash :
   public CCryptHash
```

Requirements

Header: atlcrypt.h

See Also

Reference

CCryptHash Class

Other Resources

CCryptMD4Hash Members

ATL Server Classes

CCryptMD4Hash Members

Methods

Initialize	Call this method to initialize the MD4 hash.

See Also

Reference

CCryptMD4Hash Class

Other Resources

CCryptHash Members

CCryptMD4Hash Methods

For information about the methods in **CCryptMD4Hash**, see CCryptMD4Hash Members.

CCryptMD4Hash::Initialize

Call this method to initialize the MD4 hash.

```
HRESULT Initialize(
   CCryptProv &Prov,
   LPCTSTR szText = NULL
) throw( );
```

Parameters

Prov

The cryptographic service provide (CSP).

szText

The string to add to the hash.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CCryptMD4Hash Class

CryptCreateHash

Other Resources

CCryptMD4Hash Members

CCryptMD5Hash Class

This class represents an MD5 hash.

```
class CCryptMD5Hash :
   public CCryptHash
```

Requirements

Header: atlcrypt.h

See Also

Reference

CCryptHash Class

Other Resources

CCryptMD5Hash Members

ATL Server Classes

CCryptMD5Hash Members

Methods

Initialize	Call this method to initialize the MD5 hash.

See Also

Reference

CCryptMD5Hash Class

Other Resources

CCryptHash Members

CCryptMD5Hash Methods

For information about the methods in **CCryptMD5Hash**, see CCryptMD5Hash Members.

CCryptMD5Hash::Initialize

Call this method to initialize the MD5 hash.

```
HRESULT Initialize(
   CCryptProv &Prov,
   LPCTSTR szText = NULL
) throw( );
```

Parameters

hProv

The cryptographic service provider (CSP).

szText

The string to add to the hash.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CCryptMD5Hash Class

CryptCreateHash

Other Resources

CCryptMD5Hash Members

CCryptProv Class

This class represents a cryptographic service provider (CSP).

```
class CCryptProv
```

Remarks

For more information see HCRYPTPROV in the *Platform SDK*.

Requirements

Header: atlcrypt.h

See Also

Tasks

MantaWeb Sample: Online Personal Information Management

Reference

ConfirmUser

ForceLogin

Other Resources

CCryptProv Members

ATL Server Classes

CCryptProv Members

Methods

AddRef	Call this method to add one to the reference count of the cryptographic service provider (CSP).
Attach	Call this method to attach this **CCryptProv** object to the specified cryptographic service provider (CSP).
CCryptProv	The constructor for the cryptographic service provider (CSP).
~CCryptProv	The destructor for the cryptographic service provider (CSP).
DeleteKeySet	Call this method to delete the specified key container within a cryptographic service provider (CSP).
Detach	Call this method to detach a cryptographic service provider (CSP) handle from the **CCryptProv** object.
GenRandom	Call this method to fill a buffer with cryptographically random bytes.
GetContainer	Call this method to get the name of the current key container from the cryptographic service provider (CSP).
GetHandle	Call this method to get the handle of the cryptographic service provider (CSP).
GetImpType	Call this method to get the implementation type of the cryptographic service provider (CSP).
GetName	Call this method to get the name of the cryptographic service provider (CSP).
GetParam	Call this method to retrieve parameters that govern the operations of the cryptographic service provider (CSP).
GetProvType	Call this method to get the provider type of the cryptographic service provider (CSP).
GetSecurityDesc	Call this method to get the security descriptor for the registry entry where the key set is stored from the cryptographic service provider (CSP).
GetVersion	Call this method to get the version number of the cryptographic service provider (CSP).
InitCreateKeySet	Call this method to create a new key container within the cryptographic service provider (CSP).
Initialize	Call this method to initialize the cryptographic service provider (CSP).
InitVerifyContext	Call this method to acquire a handle to a key container within a particular cryptographic service provider (CSP) in an application that does not use private keys.
Release	Call this method to release the cryptographic service provider (CSP).
SetParam	Call this method to customize the operations of the cryptographic service provider (CSP).
SetSecurityDesc	Call this method to set the security descriptor on the stored key set of the cryptographic service provider (CSP).
Uninitialize	Call this method to destroy the cryptographic service provider (CSP).

Operators

operator =	Assigns a **CCryptProv** object to the current object.

See Also

Reference

CCryptProv Class

CCryptProv Methods

For information about the methods in **CCryptProv**, see CCryptProv Members.

CCryptProv::~CCryptProv

The destructor for the cryptographic service provider (CSP).

```
~CCryptProv( ) throw( );
```

See Also

Reference

CCryptProv Class

Other Resources

CCryptProv Members

CCryptProv::AddRef

Call this method to add one to the reference count of the cryptographic service provider (CSP).

```
HRESULT AddRef( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptContextAddRef.

See Also

Reference

CCryptProv Class

CCryptProv::Release

Other Resources

CCryptProv Members

CCryptProv::Attach

Call this method to attach this **CCryptProv** object to the specified cryptographic service provider (CSP).

```
void Attach(
   HCRYPTPROV hProv,
   BOOL bTakeOwnership = FALSE
) throw( );
```

Parameters

hProv

The cryptographic service provider.

bTakeOwnership

Specifies whether the attaching object takes ownership of the CSP. If FALSE, the CSP's reference count is incremented.

See Also

Reference

CCryptProv Class

CCryptProv::Detach

Other Resources

CCryptProv Members

CCryptProv::CCryptProv

The constructor for the cryptographic service provider (CSP).

```
CCryptProv( ) throw( );
explicit CCryptProv(
   HCRYPTPROV hProv,
   BOOL bTakeOwnership = FALSE
) throw( );
CCryptProv(
   const CCryptProv& prov
) throw( );
```

Parameters

hProv

The handle to the cryptographic service provider.

bTakeOwnership

Specifies whether the new object takes ownership of the CSP. If FALSE, the CSP's reference count is incremented.

prov

The constant reference to the cryptographic service provider to copy.

See Also

Reference

CCryptProv Class

Other Resources

CCryptProv Members

CCryptProv::DeleteKeySet

Call this method to delete the specified key container within a cryptographic service provider (CSP).

```
HRESULT DeleteKeySet(
   DWORD dwProviderType = PROV_RSA_FULL,
   LPCTSTR szContainer = NULL,
   LPCTSTR szProvider = MS_DEF_PROV,
   DWORD dwFlags = 0
) throw( );
```

Parameters

dwProviderType

The type of provider. Defined provider types are discussed in Cryptographic Provider Types.

szContainer

The key container name. If *szContainer* is NULL, a default key container name is used.

szProvider

The cryptographic service provider.

dwFlags

For a table of possible values, see the description of the *dwFlags* parameter in CryptAcquireContext. Note that CRYPT_DELETEKEYSET is set automatically.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information see CryptAcquireContext in the *Platform SDK*.

See Also

Reference

CCryptProv Class

CCryptProv::InitCreateKeySet

Other Resources

CCryptProv Members

CCryptProv::Detach

Call this method to detach a cryptographic service provider (CSP) handle from the **CCryptProv** object.

```
HCRYPTPROV Detach( ) throw( );
```

Return Value

Returns the CSP handle.

See Also

Reference

CCryptProv Class

CCryptProv::Attach

Other Resources

CCryptProv Members

CCryptProv::GenRandom

Call this method to fill a buffer with cryptographically random bytes.

```
HRESULT GenRandom(
   ULONG nLength,
   BYTE* pbBuffer
) throw( );
```

Parameters

nLength

The number of bytes of random data to generate.

pbBuffer

The buffer that will receive the returned data. This buffer must be at least *nLength* bytes in length.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptGenRandom in the *Platform SDK*.

Example

See the MantaWeb Sample, the ForceLogin Sample, and the ConfirmUser Sample.

See Also

Reference

CCryptProv Class

Other Resources

CCryptProv Members

CCryptProv::GetContainer

Call this method to get the name of the current key container from the cryptographic service provider (CSP).

```
HRESULT GetContainer(
   LPSTR szBuf,
   DWORD * pdwLength
) throw( );
```

Parameters

szBuf

The buffer that will receive the returned data.

pdwLength

The length, in bytes, of the *szData* buffer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the **PP_CONTAINER** flag in CryptGetProvParam.

See Also

Reference

CCryptProv Class

Other Resources

CCryptProv Members

CCryptProv::GetHandle

Call this method to get the handle of the cryptographic service provider (CSP).

```
inline HCRYPTPROV GetHandle( ) throw( );
```

Return Value

Returns the handle of the CSP.

Example

See the ForceLogin Sample and the ConfirmUser Sample.

See Also

Reference

CCryptProv Class

Other Resources

CCryptProv Members

CCryptProv::GetImpType

Call this method to get the implementation type of the cryptographic service provider (CSP).

```
HRESULT GetImpType(
   DWORD * pdwImpType
) throw( );
```

Parameters

pdwImpType

The type of CSP implementation. See the Remarks in CryptGetProvParam.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the **PP_IMPTYPE** flag in CryptGetProvParam.

See Also

Reference

CCryptProv Class

Other Resources

CCryptProv Members

CCryptProv::GetName

Call this method to get the name of the cryptographic service provider (CSP).

```
HRESULT GetName(
   LPSTR szBuf,
   DWORD * pdwLength
) throw( );
```

Parameters

szBuf

The buffer that will receive the returned data.

pdwLength

The length, in bytes, of the *szData* buffer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the **PP_NAME** flag in CryptGetProvParam.

See Also

Reference

CCryptProv Class

Other Resources

CCryptProv Members

CCryptProv::GetParam

Call this method to retrieve parameters that govern the operations of the cryptographic service provider (CSP).

```
HRESULT GetParam(
   DWORD dwParam,
   BYTE * pbData,
   DWORD * pdwDataLen,
   DWORD dwFlags = 0
) throw( );
```

Parameters

dwParam

Specifies the nature of the query. For a table of possible values, see the description of the *dwParam* parameter in CryptGetProvParam.

pbData

The buffer that will receive the returned data.

pdwDataLen

The length, in bytes, of the *pbData* buffer.

dwFlags

The flags that give more information about the type of information to return. For possible values, see the description of the *dwFlags* parameter in **CryptGetProvParam**.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptGetProvParam.

See Also

Reference

CCryptProv Class

CCryptProv::SetParam

Other Resources

CCryptProv Members

CCryptProv::GetProvType

Call this method to get the provider type of the cryptographic service provider (CSP).

```
HRESULT GetProvType(
   DWORD * pdwType
) throw( );
```

Parameters

pdwType

The provider type of the CSP.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the **PP_PROVTYPE** flag in CryptGetProvParam.

See Also

Reference

CCryptProv Class

Other Resources

CCryptProv Members

CCryptProv::GetSecurityDesc

Call this method to get the security descriptor for the registry entry where the key set is stored from the cryptographic service provider (CSP).

```
HRESULT GetSecurityDesc(
   SECURITY_INFORMATION * pSecInfo
) throw( );
```

Parameters

pSecInfo

The security descriptor.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the **PP_KEYSET_SEC_DESCR** flag in CryptGetProvParam or see SECURITY_INFORMATION.

See Also

Reference

CCryptProv Class

CCryptProv::SetSecurityDesc

Other Resources

CCryptProv Members

CCryptProv::GetVersion

Call this method to get the version number of the cryptographic service provider (CSP).

```
HRESULT GetVersion(
   DWORD * pdwVersion
) throw( );
```

Parameters

pdwVersion

The version number of the CSP.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the **PP_VERSION** flag in CryptGetProvParam.

See Also

Reference

CCryptProv Class

Other Resources

CCryptProv Members

CCryptProv::InitCreateKeySet

Call this method to create a new key container within the cryptographic service provider (CSP).

```
HRESULT InitCreateKeySet(
   DWORD dwProviderType = PROV_RSA_FULL,
   LPCTSTR szContainer = NULL,
   LPCTSTR szProvider = MS_DEF_PROV,
   DWORD dwFlags = 0
) throw( );
```

Parameters

dwProviderType

The type of provider. Defined provider types are discussed in Cryptographic Provider Types.

szContainer

The key container name. If *szContainer* is NULL, a default key container name is used.

szProvider

The cryptographic service provider.

dwFlags

See the description of *dwFlags* in CryptAcquireContext. The **CRYPT_NEWKEYSET** flag is set automatically by **InitCreateKeySet**.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the **CRYPT_NEWKEYSET** flag in CryptAcquireContext.

See Also

Reference

CCryptProv Class

CCryptProv::DeleteKeySet

Other Resources

CCryptProv Members

CCryptProv::Initialize

Call this method to initialize the cryptographic service provider (CSP).

```
HRESULT Initialize(
   DWORD dwProviderType = PROV_RSA_FULL,
   LPCTSTR szContainer = NULL,
   LPCTSTR szProvider = MS_DEF_PROV,
   DWORD dwFlags = 0
) throw( );
```

Parameters

dwProviderType

The type of provider. Defined provider types are discussed in Cryptographic Provider Types.

szContainer

The key container name. If *szContainer* is NULL, a default key container name is used.

szProvider

The cryptographic service provider.

dwFlags

For a table of possible values, see the description of the *dwFlags* parameter in CryptAcquireContext.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptAcquireContext.

Example

See the MantaWeb Sample, the ForceLogin Sample, and the ConfirmUser Sample.

See Also

Reference

CCryptProv Class
CCryptProv::Uninitialize

Other Resources

CCryptProv Members

CCryptProv::InitVerifyContext

Call this method to acquire a handle to a key container within a particular cryptographic service provider (CSP) in an application that does not use private keys.

```
HRESULT InitVerifyContext(
   DWORD dwProviderType = PROV_RSA_FULL,
   LPCTSTR szProvider = MS_DEF_PROV,
   DWORD dwFlags = 0
) throw( );
```

Parameters

dwProviderType

The type of provider. Defined provider types are discussed in Cryptographic Provider Types.

szProvider

The cryptographic service provider.

dwFlags

See the description of *dwFlags* in CryptAcquireContext. The **CRYPT_VERIFYCONTEXT** flag is set automatically by **InitVerifyContext**.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Use this method if your application has no access to the private keys of the CSP. For more information, see the description of the **CRYPT_VERIFYCONTEXT** flag in CryptAcquireContext.

See Also

Reference

CCryptProv Class

Other Resources

CCryptProv Members

CCryptProv::Release

Call this method to release the cryptographic service provider (CSP).

```
HRESULT Release( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptReleaseContext.

See Also

Reference

CCryptProv Class

CCryptProv::AddRef

Other Resources

CCryptProv Members

CCryptProv::SetParam

Call this method to customize the operations of the cryptographic service provider (CSP).

```
HRESULT SetParam(
   DWORD dwParam,
   BYTE* pbData,
   DWORD dwFlags = 0
) throw( );
```

Parameters

dwParam

The parameter to set. See the description of the *dwParam* parameter in CryptSetProvParam.

pbData

Contains information on the parameter to set. For more information, see **CryptSetProvParam**.

dwFlags

Specifies additional information about the parameters to set. For more information, see **CryptSetProvParam**.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptSetProvParam.

See Also

Reference

CCryptProv Class

CCryptProv::GetParam

Other Resources

CCryptProv Members

CCryptProv::SetSecurityDesc

Call this method to set the security descriptor on the stored key set of the cryptographic service provider (CSP).

```
HRESULT SetSecurityDesc(
   SECURITY_INFORMATION SecInfo
) throw( );
```

Parameters

SecInfo

The security descriptor.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see the description of the **PP_KEYSET_SEC_DESCR** flag in CryptSetProvParam or see SECURITY_INFORMATION.

See Also

Reference

CCryptProv Class

CCryptProv::GetSecurityDesc

Other Resources

CCryptProv Members

CCryptProv::Uninitialize

Call this method to destroy the cryptographic service provider (CSP).

```
HRESULT Uninitialize( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptReleaseContext.

See Also

Reference

CCryptProv Class

CCryptProv::Initialize

Other Resources

CCryptProv Members

CCryptProv Operators

For information about the operators in **CCryptProv**, see CCryptProv Members.

CCryptProv::operator =

Assigns a CCryptProv object to the current object.

```
CCryptProv& operator =(
   const CCryptProv& prov
) throw( );
```

Parameters

prov

The cryptographic service provider (CSP).

Return Value

Returns the constant reference to the cryptographic service provider.

See Also

Reference

CCryptProv Class

Other Resources

CCryptProv Members

CCryptRandomKey Class

This class represents a randomly generated cryptographic session key.

```
class CCryptRandomKey :
   public CCryptKey
```

Remarks

You can export a **CCryptRandomKey** object or use it internally in a program to protect data during execution.

Requirements

Header: atlcrypt.h

See Also

Reference

CCryptKey Class

Other Resources

CCryptRandomKey Members

ATL Server Classes

CCryptRandomKey Members

Methods

Initialize	Call this method to initialize the randomly generated cryptographic session key.

See Also

Reference

CCryptRandomKey Class

Other Resources

CCryptKey Members

CCryptRandomKey Methods

For information about the methods in **CCryptRandomKey**, see CCryptRandomKey Members.

CCryptRandomKey::Initialize

Call this method to initialize the randomly generated cryptographic session key.

```
HRESULT Initialize(
   CCryptProv &Prov,
   ALG_ID algid = CALG_RC4,
   DWORD dwFlags = CRYPT_EXPORTABLE
) throw( );
```

Parameters

Prov

A cryptographic service provider (CSP).

algid

The ALG_ID structure identifying the algorithm for which the key is to be generated. For a table of possible values, see the *Algid* parameter in CryptGenKey.

dwFlags

Specifies the type of key generated. The CRYPT_EXPORTABLE flag is set by default, meaning the key can be transferred out of the CSP into a key BLOB. See the *dwFlags* parameter in CryptGenKey.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptGenKey.

See Also

Reference

CCryptRandomKey Class

Other Resources

CCryptRandomKey Members

CCryptSHAHash Class

This class represents a Secure Hash Algorithm (SHA) hash.

```
class CCryptSHAHash :
   public CCryptHash
```

Remarks

This class represents the Secure Hash Algorithm (technically, SHA-1) hash, from the National Institute of Standards and Technology (NIST) and the National Security Agency (NSA).

Requirements

Header: atlcrypt.h

See Also

Reference

CCryptHash Class

Other Resources

CCryptSHAHash Members

ATL Server Classes

CCryptSHAHash Members

Methods

Initialize	Call this method to initialize the SHA hash.

See Also

Reference

CCryptSHAHash Class

Other Resources

CCryptHash Members

CCryptSHAHash Methods

For information about the methods in **CCryptSHAHash**, see CCryptSHAHash Members.

CCryptSHAHash::Initialize

Call this method to initialize the SHA hash.

```
HRESULT Initialize(
   CCryptProv &Prov,
   LPCTSTR szText = NULL
) throw( );
```

Parameters

Prov

The cryptographic service provide (CSP).

szText

The string to add to the hash.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptCreateHash.

See Also

Reference

CCryptSHAHash Class

Other Resources

CCryptSHAHash Members

CCryptSSL3SHAMD5Hash Class

This class represents an SSL3 client authentication hash.

```
class CCryptSSL3SHAMD5Hash :
   public CCryptHash
```

Remarks

The SSL3 algorithm is used for client authentication in Secure Sockets Layer (SSL) version 3.

Requirements

Header: atlcrypt.h

See Also

Reference

CCryptHash Class

CALG_SSL3_SHAMD5

Other Resources

CCryptSSL3SHAMD5Hash Members

ATL Server Classes

CCryptSSL3SHAMD5Hash Members

Methods

Initialize	Call this method to initialize the SSL3 client authentication hash.

See Also

Reference

CCryptSSL3SHAMD5Hash Class

Other Resources

CCryptHash Members

CCryptSSL3SHAMD5Hash Methods

For information about the methods in **CCryptSSL3SHAMD5Hash**, see CCryptSSL3SHAMD5Hash Members.

CCryptSSL3SHAMD5Hash::Initialize

Call this method to initialize the SSL3 client authentication hash.

```
HRESULT Initialize(
   CCryptProv &Prov,
   CCryptKey &Key,
   LPCTSTR szText = NULL
) throw( );
```

Parameters

Prov

The cryptographic service provide (CSP).

Key

The key for the hash.

szText

The string to add to the hash.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptCreateHash.

See Also

Reference

CCryptSSL3SHAMD5Hash Class

Other Resources

CCryptSSL3SHAMD5Hash Members

CCryptUserExKey Class

This class represents the user's exchange key pair (used to encrypt and decrypt session keys).

```
class CCryptUserExKey :
   public CCryptKey
```

Requirements

Header: atlcrypt.h

See Also

Reference

CCryptKey Class

Other Resources

CCryptUserExKey Members

ATL Server Classes

CCryptUserExKey Members

Methods

Create	Call this method to generate an exchange key pair.
Initialize	Call this method to initialize an exchange key pair.

See Also

Reference

CCryptUserExKey Class

Other Resources

CCryptKey Members

CCryptUserExKey Methods

For information about the methods in **CCryptUserExKey**, see CCryptUserExKey Members.

CCryptUserExKey::Create

Call this method to generate an exchange key pair.

```
HRESULT Create(
   CCryptProv &Prov
) throw( );
```

Parameters

Prov

The cryptographic service provider (CSP).

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptGenKey.

See Also

Reference

CCryptUserExKey Class

Other Resources

CCryptUserExKey Members

CCryptUserExKey::Initialize

Call this method to initialize an exchange key pair.

```
HRESULT Initialize(
   CCryptProv &Prov
) throw( );
```

Parameters

Prov

The cryptographic service provider (CSP).

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptGetUserKey.

See Also

Reference

CCryptUserExKey Class

Other Resources

CCryptUserExKey Members

CCryptUserSigKey Class

This class represents the user's signature key pair (used to create and verify digital signatures).

```
class CCryptUserSigKey :
   public CCryptKey
```

Requirements

Header: atlcrypt.h

See Also

Reference

CCryptKey Class

Other Resources

CCryptUserSigKey Members

ATL Server Classes

CCryptUserSigKey Members

Methods

Create	Call this method to create a signature key.
Initialize	Call this method to initialize a signature key.

See Also

Reference

CCryptUserSigKey Class

Other Resources

CCryptKey Members

CCryptUserSigKey Methods

For information about the methods in **CCryptUserSigKey**, see CCryptUserSigKey Members.

CCryptUserSigKey::Create

Call this method to generate a signature key.

```
HRESULT Create(
   CCryptProv &Prov
) throw( );
```

Parameters

Prov

The cryptographic service provider (CSP).

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptGenKey.

See Also

Reference

CCryptUserSigKey Class

Other Resources

CCryptUserSigKey Members

CCryptUserSigKey::Initialize

Call this method to initialize a signature key.

```
HRESULT Initialize(
   CCryptProv &Prov
) throw( );
```

Parameters

Prov

The cryptographic service provider (CSP).

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see CryptGetUserKey.

See Also

Reference

CCryptUserSigKey Class

Other Resources

CCryptUserSigKey Members

CCullerCacheData Structure

This structure holds information about cache entry that can be used by classes conforming to the cache culler archetype to determine whether, and in what order, items should be removed from the cache.

```
struct CCullerCacheData
```

Remarks

Each entry in a CMemoryCacheBase-derived cache has an associated **CCullerCacheData** object that holds information about the lifespan and expiration time of that item. This information can be set and read by a cache culler to provide a time-based mechanism for removing items from the cache. Each **CCullerCacheData** object also serves as a doubly linked list node so that the cache culler can quickly find the next candidate for removal from the cache.

Requirements

Header: atlcache.h

See Also

Other Resources

CCullerCacheData Members

CCullerCacheData Members

Methods

CCullerCacheData	The constructor.

Data Members

cftExpireTime	The absolute time after which the item should be removed from the cache.
nLifespan	The period after which the item should be removed from the cache.
pNext	The address of the next item in a linked list of **CCullerCacheData** structures.
pPrev	The address of the previous item in a linked list of **CCullerCacheData** structures.

See Also

Reference

Cache Culler Archetype

CCullerCacheData Structure

Other Resources

Caching

Caching Reference

CCullerCacheData Methods

For information about the methods in **CCullerCacheData**, see CCullerCacheData Members

CCullerCacheData::CCullerCacheData

The constructor.

```
CCullerCacheData( );
```

Remarks

Initializes all data members to zero.

See Also

Reference

CCullerCacheData Structure

Other Resources

CCullerCacheData Members

Caching Reference

CCullerCacheData Data Members

For information about the data members in **CCullerCacheData**, see CCullerCacheData Members

CCullerCacheData::cftExpireTime

The absolute time after which the item should be removed from the cache.

```
CFileTime cftExpireTime;
```

Remarks

The exact meaning of this member depends on the cache culler implementation. In general, however, cullers treat this as an absolute time after which the item should be removed from the cache.

See Also

Reference

Cache Culler Archetype

CCullerCacheData Structure

Other Resources

Caching

CCullerCacheData Members

Caching Reference

CCullerCacheData::nLifespan

The period after which the item should be removed from the cache.

```
ULONGLONG nLifespan;
```

Remarks

The exact meaning of this member depends on cache culler implementation. In general, however, cullers should treat this as a time span specifying the period after which the item should be removed from the cache.

CFixedLifetimeCuller interprets this member as being relative to the time at which the item was added to the cache. CLifetimeCuller interprets this member as being relative to the time at which the item was last accessed.

See Also

Reference

Cache Culler Archetype
CCullerCacheData Structure

Other Resources

Caching
CCullerCacheData Members
Caching Reference

CCullerCacheData::pNext

The address of the next item in a linked list of **CCullerCacheData** structures.

```
CCullerCacheData* pNext;
```

Remarks

Will be **NULL** if this node represents the end of the list.

See Also

Reference

Cache Culler Archetype

CCullerCacheData Structure

Other Resources

Caching

CCullerCacheData Members

Caching Reference

CCullerCacheData::pPrev

The address of the previous item in a linked list of **CCullerCacheData** structures.

```
CCullerCacheData* pPrev;
```

Remarks

Will be **NULL** if this node represents the start of the list.

See Also

Reference

Cache Culler Archetype

CCullerCacheData Structure

Other Resources

Caching

CCullerCacheData Members

Caching Reference

CDataSourceCache Class

This class caches open data connections and implements the IDataSourceCache interface.

```
template <
   class TCritSec = CComFakeCriticalSection
>
class CDataSourceCache :
   public IDataSourceCache,
   public CComObjectRootEx< CComGlobalsThreadModel >
```

Parameters

TCritSec

The critical section class used to synchronize access to the data held by this class. The class used as the default, CComFakeCriticalSection, provides no synchronization. If the data source cache is to be accessed from multiple threads, set this parameter to CComCriticalSection.

Remarks

Use **CDataSourceCache** as a template argument to one of ATL's object classes, such as CComObject or CComObjectGlobal, and instantiate that class.

Requirements

Header: atlcache.h

See Also

Reference

IDataSourceCache Interface

CComObjectRootEx Class

Other Resources

CDataSourceCache Members

ATL Server Classes

Caching

Caching Reference

CDataSourceCache Members

Methods

Add	Implementation of IDataSourceCache::Add.
CDataSourceCache	The constructor.
~CDataSourceCache	The destructor.
Lookup	Implementation of IDataSourceCache::Lookup.
Remove	Implementation of IDataSourceCache::Remove.
Uninitialize	Implementation of IDataSourceCache::Uninitialize.

See Also

Reference

CDataSourceCache Class

Other Resources

Caching

IDataSourceCache Members

CComObjectRootEx Members

CDataSourceCache Methods

For information about the methods in **CDataSourceCache**, see CDataSourceCache Members.

CDataSourceCache::Add

Implementation of IDataSourceCache::Add.

```
STDMETHOD(Add)(
   LPCTSTR szID,
   LPCOLESTR szConn,
   CDataConnection * pSession
);
```

See Also

Reference

CDataSourceCache Class
IDataSourceCache::Add
CDataConnection Class

Other Resources

CDataSourceCache Members
Caching Reference

CDataSourceCache::CDataSourceCache

The constructor.

```
CDataSourceCache( );
```

See Also

Reference

CDataSourceCache Class

Other Resources

CDataSourceCache Members

Caching Reference

CDataSourceCache::~CDataSourceCache

The destructor.

```
virtual ~CDataSourceCache( );
```

Remarks

The destructor calls CDataSourceCache::Uninitialize.

See Also

Reference

CDataSourceCache Class

Other Resources

CDataSourceCache Members

Caching Reference

CDataSourceCache::Lookup

Implementation of IDataSourceCache::Lookup.

```
STDMETHOD(Lookup)(
   LPCTSTR szID,
   CDataConnection* pSession
);
```

See Also

Reference

CDataSourceCache Class

IDataSourceCache::Lookup

Other Resources

CDataSourceCache Members

Caching Reference

CDataSourceCache::Remove

Implementation of IDataSourceCache::Remove.

```
STDMETHOD(Remove)(
   LPCTSTR szID
);
```

See Also

Reference

CDataSourceCache Class

IDataSourceCache::Remove

Other Resources

CDataSourceCache Members

Caching Reference

CDataSourceCache::Uninitialize

Implementation of IDataSourceCache::Uninitialize.

```
STDMETHOD(Uninitialize)( );
```

See Also

Reference

CDataSourceCache Class

IDataSourceCache::Uninitialize

Other Resources

CDataSourceCache Members

Caching Reference

CDBSession Class

This class template provides an implementation of ISession that stores the session variables in a database.

```
template <
   class QueryClass = CDefaultQueryClass
>
class CDBSession :
   public ISession,
   public CComObjectRootEx< CComGlobalsThreadModel >
```

Parameters

QueryClass

The class providing the SQL queries used to store and retrieve session variables in the database. See CDefaultQueryClass for more information.

Remarks

This class template is typically used with CDBSessionServiceImplT.

Requirements

Header: atlsession.h

See Also

Tasks

OnlineAddressBook Sample: Stores an Address Book in an XML Web Service
SessionSettings Sample: Session State Demonstration

Reference

ShowSession
ISession Interface
CDefaultQueryClass Class
CSessionStateService Class
CMemSessionServiceImpl Class

Concepts

Session-State Tasks
Session-State Support
Session-State Services in ATL Server

Other Resources

CDBSession Members

CDBSession Members

Methods

Access	Call this method to update the last access time of the session.
BeginVariableEnum	Implementation of ISession::BeginVariableEnum.
CDBSession	The constructor.
~CDBSession	The destructor.
CloseEnum	Implementation of ISession::CloseEnum.
FreeSession	Call this method to free the session if there are no outstanding references on it.
GetCount	Implementation of ISession::GetCount.
GetNextVariable	Implementation of ISession::GetNextVariable.
GetSessionConnection	Call this method to get the connection to the session database.
GetVariable	Implementation of ISession::GetVariable.
Initialize	Call this method to initialize the object before calling any other methods.
IsExpired	Implementation of ISession::IsExpired.
RemoveAllVariables	Implementation of ISession::RemoveAllVariables.
RemoveVariable	Implementation of ISession::RemoveVariable.
SessionLock	Call this method to increment the reference count for this session and prevent it from being deleted.
SessionUnlock	Call this method to decrement the reference count for this session and prevent it from being deleted.
SetTimeout	Implementation of ISession::SetTimeout.
SetVariable	Implementation of ISession::SetVariable.

See Also

Reference

CDBSession Class

Other Resources

ISession Members

CDBSession Methods

For information about the methods in **CDBSession**, see CDBSession Members.

CDBSession::Access

Call this method to update the last access time of the session.

```
HRESULT Access( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CDBSession Class

Other Resources

CDBSession Members

CDBSession::BeginVariableEnum

Implementation of ISession::BeginVariableEnum.

```
STDMETHOD(BeginVariableEnum)(
   POSITION * pPOS,
   HSESSIONENUM * phEnum
) throw( );
```

See Also

Reference

CDBSession Class

HSESSIONENUM

ISession::BeginVariableEnum

Other Resources

CDBSession Members

CDBSession::CDBSession

The constructor.

```
CDBSession( ) throw( );
```

See Also

Reference

CDBSession Class

Other Resources

CDBSession Members

CDBSession::~CDBSession

The destructor.

```
~CDBSession( ) throw( );
```

See Also

Reference

CDBSession Class

Other Resources

CDBSession Members

CDBSession::CloseEnum

Implementation of ISession::CloseEnum.

```
STDMETHOD(CloseEnum)(
   HSESSIONENUM hEnum
) throw( );
```

See Also

Reference

CDBSession Class

HSESSIONENUM

ISession::CloseEnum

Other Resources

CDBSession Members

CDBSession::FreeSession

Call this method to free the session if there are no outstanding references on it.

```
HRESULT FreeSession( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CDBSession Class

Other Resources

CDBSession Members

CDBSession::GetCount

Implementation of ISession::GetCount.

```
STDMETHOD(GetCount)(
   long * pnCount
) throw( );
```

Example

See the ShowSession Sample.

See Also

Reference

CDBSession Class

ISession::GetCount

Other Resources

CDBSession Members

CDBSession::GetNextVariable

Implementation of ISession::GetNextVariable.

```
STDMETHOD(GetNextVariable)(
   POSITION * pPOS,
   VARIANT * pVal,
   HSESSIONENUM hEnum,
   LPSTR szName = NULL,
   DWORD dwLen = 0
) throw( );
```

See Also

Reference

CDBSession Class

HSESSIONENUM

ISession::GetNextVariable

Other Resources

CDBSession Members

CDBSession::GetSessionConnection

Call this method to get the connection to the session database.

```
HRESULT GetSessionConnection(
   CDataConnection * pConn,
   IServiceProvider * pProv
) throw( );
```

Parameters

pConn

[out] Address of the CDataConnection variable that, on success, receives a copy of the data connection.

pProv

The service provider expected to expose **IDataSourceCache**.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CDBSession Class

CDataConnection Class

Other Resources

CDBSession Members

CDBSession::GetVariable

Implementation of ISession::GetVariable.

```
STDMETHOD(GetVariable)(
   LPCSTR szName,
   VARIANT * pVal
) throw( );
```

Example

See the OnlineAddressBook Sample, the SessionSettings Sample, and the ShowSession Sample.

See Also

Reference

CDBSession Class

ISession::GetVariable

Other Resources

CDBSession Members

CDBSession::Initialize

Call this method to initialize the object before calling any other methods.

```
HRESULT Initialize(
   LPCSTR szSessionName,
   IServiceProvider * pServiceProvider,
   DWORD_PTR dwCookie,
   PFN_GETPROVIDERINFO pfnInfo
) throw( );
```

Parameters

szSessionName

The name of the session object.

pServiceProvider

The service provider expected to expose **IDataSourceCache**.

dwCookie

The cookie to pass to *pfnInfo* when called.

pfnInfo

A pointer to a function that returns a connection string of the datasource.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CDBSession Class

PFN_GETPROVIDERINFO

Other Resources

CDBSession Members

CDBSession::IsExpired

Implementation of ISession::IsExpired.

```
STDMETHOD(IsExpired)( ) throw( );
```

See Also

Reference

CDBSession Class

ISession::IsExpired

Other Resources

CDBSession Members

CDBSession::RemoveAllVariables

Implementation of ISession::RemoveAllVariables.

```
STDMETHOD(RemoveAllVariables)( ) throw( );
```

See Also

Reference

CDBSession Class

ISession::RemoveAllVariables

Other Resources

CDBSession Members

CDBSession::RemoveVariable

Implementation of ISession::RemoveVariable.

```
STDMETHOD(RemoveVariable)(
   LPCSTR szName
) throw( );
```

See Also

Reference

CDBSession Class

ISession::RemoveVariable

Other Resources

CDBSession Members

CDBSession::SessionLock

Call this method to increment the reference count for this session and prevent it from being deleted.

```
HRESULT SessionLock( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CDBSession Class

CDBSession::SessionUnlock

Other Resources

CDBSession Members

CDBSession::SessionUnlock

Call this method to decrement the reference count for this session and prevent it from being deleted.

```
HRESULT SessionUnlock( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CDBSession Class

CDBSession::SessionLock

Other Resources

CDBSession Members

CDBSession::SetTimeout

Implementation of ISession::SetTimeout.

```
STDMETHOD(SetTimeout)(
   unsigned __int64 dwNewTimeout
) throw( );
```

See Also

Reference

CDBSession Class

ISession::SetTimeout

Other Resources

CDBSession Members

CDBSession::SetVariable

Implementation of ISession::SetVariable.

```
STDMETHOD(SetVariable)(
   LPCSTR szName,
   VARIANT Val
) throw( );
```

See Also

Reference

CDBSession Class

ISession::SetVariable

Other Resources

CDBSession Members

CDBSessionServiceImplT Class

This class provides a database-backed session-state implementation for use within ATL Server ISAPI extension DLLs.

```
template < class TDBSession = CDBSession >
class CDBSessionServiceImplT
```

Parameters

TDBSession

Name of the class used for database access. If no class is given, a class providing default behavior is used.

Remarks

The **CDBSessionServiceImplT** class provides the implementation for database-backed ATL Server session-state support. Used as a template argument to the **CSessionStateService** class, **CDBSessionServiceImplT** provides the database support necessary to manage session-state data. The **CDBSessionServiceImplT** class uses OLEDB to communicate with the persistence database, and was designed for use with a SQL server based database, but will also work with other types of databases that have OLEDB providers, such as Access. See Session-State Database Schema for a description of how ATL Server stores session-state data.

As an implementation class, **CDBSessionServiceImplT** serves a behind-the-scenes role and should be used in your code only as a template parameter to **CSessionStateService**. See Providing Session-state Services for more information.

Requirements

Header: atlsession.h

See Also

Tasks

OnlineAddressBook Sample: Stores an Address Book in an XML Web Service
SessionSettings Sample: Session State Demonstration

Reference

ShowSession
CSessionStateService Class
CMemSessionServiceImpl Class

Concepts

Session-State Tasks
Session-State Support
Session-State Services in ATL Server

Other Resources

Session-State Services (ATL Server)
ATL Server Classes

CDebugReportHook Class

Use this class to send debug reports to a named pipe.

```
class CDebugReportHook
```

Remarks

Create an instance of this class in debug builds of your services or applications to send debug reports to a named pipe. Debug reports are generated by calling _CrtDbgReport or using a wrapper for this function such as the ATLTRACE and ATLASSERT macros.

Use of this class allows you to interactively debug components running in non-interactive window stations.

Note that debug reports are sent using the underlying security context of the thread. Impersonation is temporarily disabled so that debug reports can be viewed in situations where impersonation of low privilege users is taking place, such as in web applications.

Follow the instructions in Viewing Trace Messages and Handling Asserts to view the trace messages and interact with asserts.

Requirements

Header: atlutil.h

See Also

Tasks

WebDbg Sample: A Client for Debugging Server Applications Using CDebugReportHook

Other Resources

CDebugReportHook Members

ATL Server Classes

CDebugReportHook Members

Methods

CDebugReportHook	Calls SetPipeName, SetTimeout, and SetHook.
~CDebugReportHook	Calls CDebugReportHook::RemoveHook.
RemoveHook	Call this method to stop sending debug reports to the named pipe and restore the previous report hook.
SetHook	Call this method to start sending debug reports to the named pipe.
SetPipeName	Call this method to set the machine and name of the pipe to which the debug reports will be sent.
SetTimeout	Call this method to set the time in milliseconds that this class will wait for the named pipe to become available.

Static Functions

CDebugReportHookProc	The custom reporting function that is hooked into the C run-time debug reporting process.

See Also

Reference

CDebugReportHook Class

CDebugReportHook Methods

For information about the methods in **CDebugReportHook**, see CDebugReportHook Members.

CDebugReportHook::CDebugReportHook

Calls SetPipeName, SetTimeout, and SetHook.

```
CDebugReportHook(
   LPCSTR szMachineName = ".",
   LPCSTR szPipeName = "AtlsDbgPipe",
   DWORD dwTimeout = 20000
) throw();
```

Parameters

szMachineName

The name of the machine to which the debug output should be sent. Defaults to the local machine.

szPipeName

The name of the named pipe to which the debug output should be sent.

dwTimeout

The time in milliseconds that this class will wait for the named pipe to become available.

See Also

Reference

CDebugReportHook Class

Other Resources

CDebugReportHook Members

CDebugReportHook::~CDebugReportHook

Calls CDebugReportHook::RemoveHook.

```
~CDebugReportHook( ) throw( );
```

See Also

Reference

CDebugReportHook Class

Other Resources

CDebugReportHook Members

CDebugReportHook::RemoveHook

Call this method to stop sending debug reports to the named pipe and restore the previous report hook.

```
void RemoveHook( ) throw( );
```

Remarks

Calls _CrtSetReportHook2 to restore the previous report hook.

See Also

Reference

CDebugReportHook Class

CDebugReportHook::SetHook

Other Resources

CDebugReportHook Members

CDebugReportHook::SetHook

Call this method to start sending debug reports to the named pipe.

```
void SetHook( ) throw( );
```

Remarks

Calls _CrtSetReportHook2 to have debug reports routed through CDebugReportHookProc to the named pipe. This class keeps track of the previous report hook so that it can be restored when RemoveHook is called.

See Also

Reference

CDebugReportHook Class

Other Resources

CDebugReportHook Members

CDebugReportHook::SetPipeName

Call this method to set the machine and name of the pipe to which the debug reports will be sent.

```
BOOL SetPipeName(
   LPCSTR szMachineName = ".",
   LPCSTR szPipeName = "AtlsDbgPipe"
) throw( );
```

Parameters

szMachineName

The name of the machine to which the debug output should be sent.

szPipeName

The name of the named pipe to which the debug output should be sent.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CDebugReportHook Class

Other Resources

CDebugReportHook Members

CDebugReportHook::SetTimeout

Call this method to set the time in milliseconds that this class will wait for the named pipe to become available.

```
void SetTimeout(
   DWORD dwTimeout
);
```

Parameters

dwTimeout

The time in milliseconds that this class will wait for the named pipe to become available.

See Also

Reference

CDebugReportHook Class

Other Resources

CDebugReportHook Members

CDebugReportHook Static Functions

For information about the static functions in **CDebugReportHook**, see CDebugReportHook Members.

CDebugReportHook::CDebugReportHookProc

The custom reporting function that is hooked into the C run-time debug reporting process.

```
static int __cdecl CDebugReportHookProc(
   int reportType,
   char* message,
   int* returnValue
) throw( );
```

Parameters

reportType

The type of the report (_CRT_WARN, _CRT_ERROR, or _CRT_ASSERT).

message

The message string.

returnValue

The value that should be returned by _CrtDbgReport.

Return Value

Returns FALSE if the hook handles the message in question completely so that no further reporting is required. Returns TRUE if **_CrtDbgReport** should report the message in the normal way.

Remarks

The reporting function attempts to open the named pipe and communicate with the process at the other end. If the pipe is busy, the reporting function will wait until the pipe is free or the timeout expires. The timeout can be set by the constructor or a call to CDebugReportHook::SetTimeout.

The code in this function is executed in the underlying security context of the calling thread, that is, impersonation is disabled for the duration of this function.

See Also

Reference

CDebugReportHook Class

_CrtDbgReport, _CrtDbgReportW

Other Resources

CDebugReportHook Members

CDefaultAuth Class

This class allows you to check whether the user making an HTTP request is in a particular group.

```
class CDefaultAuth :
   public CVerifyAuth< CDefaultAuth >
```

Remarks

This class is used by ATL Server's extension management services. See ATL_NO_DEFAULT_AUTHORITY for more details.

Requirements

Header: atlisapi.h

See Also

Reference

CVerifyAuth Class

Other Resources

CDefaultAuth Members

ATL Server Classes

CDefaultAuth Members

Methods

CheckAccount	Call this method to check whether the user is a member of the specified group.
GetErrorResponse	Call this method to get a string containing the HTML to send to the client in the event that the user is not au thorized.
HandleError	Call this method to send the message returned by GetErrorResponse to the client.

See Also

Reference

CDefaultAuth Class

Other Resources

CVerifyAuth Members

CDefaultAuth Methods

For information about the methods in **CDefaultAuth**, see CDefaultAuth Members

CDefaultAuth::CheckAccount

Call this method to check whether the user is a member of the specified group.

```
virtual bool CheckAccount(
   IHttpServerContext* pContext,
   const SID* psidAuthGroup
) throw( );
```

Parameters

pContext

The server context that contains information about the user.

psidAuthGroup

The security identifier for the group.

Return Value

Returns **true** if an impersonation token can be obtained for the user and the account being impersonated is a member of the group identified by *psidAuthGroup*. Otherwise, returns **false**.

Remarks

Override of CVerifyAuth::CheckAccount.

See Also

Reference

CDefaultAuth Class

Other Resources

CDefaultAuth Members

CDefaultAuth::GetErrorResponse

Call this method to get a string containing the HTML to send to the client in the event that the user is not authorized.

```
virtual LPCSTR GetErrorResponse( );
```

Return Value

Returns the string containing the HTML-formatted error message.

See Also

Reference

CDefaultAuth Class

Other Resources

CDefaultAuth Members

CDefaultAuth::HandleError

Call this method to send the message returned by GetErrorResponse to the client.

```
HTTP_CODE HandleError(
   AtlServerRequest * pRequestInfo
) throw( );
```

Parameters

pRequestInfo

The information about the current request.

Return Value

Returns HTTP_SUCCESS_NO_PROCESS on success, or HTTP_FAIL on failure.

Remarks

Non-virtual override of CVerifyAuth::HandleError.

See Also

Reference

CDefaultAuth Class
AtlServerRequest Structure
HTTP_CODE

Other Resources

CDefaultAuth Members

CDefaultErrorProvider Structure

This class provides access to error information that can be displayed to the user or used for diagnostic messages.

```
struct CDefaultErrorProvider
```

Remarks

This class is typically used as the *HttpUserErrorTextProvider* of CIsapiExtension, which calls the static GetErrorText function to get the error response corresponding to an HTTP status code and an ATL Server subcode.

User-defined error providers should offer the same public interface as this class.

Requirements

Header: atlisapi.h

Example

See the ShowErrors Sample.

See Also

Reference

CIsapiExtension Class

Other Resources

CDefaultErrorProvider Members

ATL Server Classes

CDefaultErrorProvider Members

Static Functions

GetErrorText	Call this function to get error text corresponding to an HTTP status code and an ATL Server subcode.

See Also

Reference

CDefaultErrorProvider Structure

CDefaultErrorProvider Static Functions

For information about the static functions in **CDefaultErrorProvider**, see CDefaultErrorProvider Members.

CDefaultErrorProvider::GetErrorText

Call this function to get error text corresponding to an HTTP status code and an ATL Server subcode.

```
static BOOL GetErrorText(
   UINT uError,
   UINT uSubErr,
   LPCSTR* ppszHeader,
   UINT* puResId
) throw( );
```

Parameters

uError

The HTTP status code.

uSubErr

The ATL Server subcode.

ppszHeader

[out] Address of the LPCSTR variable that, on success, receives the header string of the response corresponding to the status code and subcode. Can be NULL.

puResId

[out] Address of the UINT variable that, on success, receives the resource ID of the body corresponding to the status code and subcode. Can be NULL.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This function retrieves the HTTP response header string (in *ppszHeader*) and a resource ID of the response body (in *puResId*) for a given HTTP status code (*uError*) and ATL Server subcode (*uSubErr*). If no response body is provided, ***puResId* will be zero after calling the function.

Some of the errors recognized by this implementation of **GetErrorText** have response bodies that are defined in atlsrv.rc. This resource file is automatically included in the resources of projects generated using the ATL Server Project Wizard.

When this, or a compatible, class is used as the *HttpUserErrorTextProvider* of CIsapiExtension, and the resource specified by ***puResId* is not available to the ISAPI extension DLL, no error results and the response simply gets sent with an empty body.

Example

See the ShowErrors Sample.

See Also

Reference

CDefaultErrorProvider Structure

Other Resources

CDefaultErrorProvider Members

CDefaultQueryClass Class

This class provides access to the SQL queries used for database-backed session state.

```
class CDefaultQueryClass
```

Remarks

This class is used as a default template parameter value by CDBSession and provides queries that match the schema described in Session-State Database Schema.

The documentation for the methods of this class provides guidelines for implementing a compatible class that matches your own database schema but can still be used with CDBSession.

Requirements

Header: atlsession.h

See Also

Reference

CDBSession Class

Session-State Database Schema

Other Resources

CDefaultQueryClass Members

CDefaultQueryClass Members

Methods

GetSessionRefAccess	Call this method to get the SQL query that will be executed when a session is accessed.
GetSessionRefAddRef	Call this method to get the SQL query that will be executed when the count of a session referenc e is incremented.
GetSessionRefCreate	Call this method to get the SQL query that will be executed when a session reference is created.
GetSessionRefDelete	Call this method to get the SQL query that will be executed when a session reference is conditio nally deleted.
GetSessionRefDeleteFinal	Call this method to get the SQL query that will be executed when a session reference must be d eleted.
GetSessionReferencesSet	Call this method to get the SQL query that will be executed to set the timeout of all session refer ences.
GetSessionRefGetCount	Call this method to get the SQL query that will be executed when the count of session reference s is requested.
GetSessionRefIsExpired	Call this method to get the SQL query that will be executed to determine whether or not a sessi on reference has expired.
GetSessionRefRemoveRef	Call this method to get the SQL query that will be executed when the count of a session referenc e is decremented.
GetSessionRefSelect	Call this method to get the SQL query that will be executed to get information about all the refer ences for a particular session.
GetSessionRefUpdateTimeout	Call this method to get the SQL query that will be executed to update the timeout of a particular session.
GetSessionVarCount	Call this method to get the SQL query that will be executed to get the count of variables in a par ticular session.
GetSessionVarDeleteAllVars	Call this method to get the SQL query that will be executed to delete all the variables in a particu lar session.
GetSessionVarDeleteVar	Call this method to get the SQL query that will be executed to delete a named variable from a p articular session.
GetSessionVarInsert	Call this method to get the SQL query that will be executed to insert a variable into a particular s ession.
GetSessionVarSelectAllVars	Call this method to get the SQL query that will be executed to get information about all the vari ables in a particular session.
GetSessionVarSelectVar	Call this method to get the SQL query that will be executed to get information about a named v ariable in a particular session.

GetSessionVarUpdate	Call this method to get the SQL query that will be executed to update the value of a named variable in a particular session.

See Also

Reference

CDefaultQueryClass Class

CDefaultQueryClass Methods

For information about the methods in **CDefaultQueryClass**, see CDefaultQueryClass Members

CDefaultQueryClass::GetSessionRefAccess

Call this method to get the SQL query that will be executed when a session is accessed.

```
LPCTSTR GetSessionRefAccess( ) throw( );
```

Return Value

Returns the SQL query that will be executed when a session is accessed.

Remarks

When implementing a compatible method, the query returned should have the following parameters:

Position	Description
1	Session ID

See Also

Reference

CDefaultQueryClass Class

Other Resources

CDefaultQueryClass Members

CDefaultQueryClass::GetSessionRefAddRef

Call this method to get the SQL query that will be executed when the count of a session reference is incremented.

```
LPCTSTR GetSessionRefAddRef( ) throw( );
```

Return Value

Returns the SQL query that will be executed when the count of a session reference is incremented.

Remarks

When implementing a compatible method, the query returned should have the following parameters:

Position	Description
1	Session ID

See Also

Reference

CDefaultQueryClass Class

Other Resources

CDefaultQueryClass Members

CDefaultQueryClass::GetSessionRefCreate

Call this method to get the SQL query that will be executed when a session reference is created.

```
LPCTSTR GetSessionRefCreate( ) throw( );
```

Return Value

Returns the SQL query that will be executed when a session reference is created.

Remarks

When implementing a compatible method, the query returned should have the following parameters:

Position	Description
1	Session ID
2	Timeout in milliseconds

See Also

Reference

CDefaultQueryClass Class

Other Resources

CDefaultQueryClass Members

CDefaultQueryClass::GetSessionRefDelete

Call this method to get the SQL query that will be executed when a session reference is conditionally deleted.

```
LPCTSTR GetSessionRefDelete( ) throw( );
```

Return Value

Returns the SQL query that will be executed when a session reference is conditionally deleted.

Remarks

When implementing a compatible method, the query returned should have the following parameters:

Position	Description
1	Session ID

See Also

Reference

CDefaultQueryClass Class

Other Resources

CDefaultQueryClass Members

CDefaultQueryClass::GetSessionRefDeleteFinal

Call this method to get the SQL query that will be executed when a session reference must be deleted.

```
LPCTSTR GetSessionRefDeleteFinal( ) throw( );
```

Return Value

Returns the SQL query that will be executed when a session reference must be deleted.

Remarks

When implementing a compatible method, the query returned should have the following parameters:

Position	Description
1	Session ID

See Also

Reference

CDefaultQueryClass Class

Other Resources

CDefaultQueryClass Members

CDefaultQueryClass::GetSessionReferencesSet

Call this method to get the SQL query that will be executed to set the timeout of all session references.

```
LPCTSTR GetSessionReferencesSet( ) throw( );
```

Return Value

Returns the SQL query that will be executed to set the timeout of all session references.

Remarks

When implementing a compatible method, the query returned should have the following parameters:

Position	Description
1	Timeout in milliseconds

See Also

Reference

CDefaultQueryClass Class

Other Resources

CDefaultQueryClass Members

CDefaultQueryClass::GetSessionRefGetCount

Call this method to get the SQL query that will be executed when the count of session references is requested.

```
LPCTSTR GetSessionRefGetCount( ) throw( );
```

Return Value

Returns the SQL query that will be executed when the count of session references is requested.

Remarks

When implementing a compatible method, the query returned should have no parameters.

See Also

Reference

CDefaultQueryClass Class

Other Resources

CDefaultQueryClass Members

CDefaultQueryClass::GetSessionRefIsExpired

Call this method to get the SQL query that will be executed to determine whether or not a session reference has expired.

```
LPCTSTR GetSessionRefIsExpired( ) throw( );
```

Return Value

Returns the SQL query that will be executed to determine whether or not a session reference has expired.

Remarks

When implementing a compatible method, the query returned should have the following parameters:

Position	Description
1	Session ID

See Also

Reference

CDefaultQueryClass Class

Other Resources

CDefaultQueryClass Members

CDefaultQueryClass::GetSessionRefRemoveRef

Call this method to get the SQL query that will be executed when the count of a session reference is decremented.

```
LPCTSTR GetSessionRefRemoveRef( ) throw( );
```

Return Value

Returns the SQL query that will be executed when the count of a session reference is decremented.

Remarks

When implementing a compatible method, the query returned should have the following parameters:

Position	Description
1	Session ID

See Also

Reference

CDefaultQueryClass Class

Other Resources

CDefaultQueryClass Members

CDefaultQueryClass::GetSessionRefSelect

Call this method to get the SQL query that will be executed to get information about all the references for a particular session.

```
LPCTSTR GetSessionRefSelect( ) throw( );
```

Return Value

Returns the SQL query that will be executed to get information about all the references for a particular session.

Remarks

When implementing a compatible method, the query returned should have the following parameters:

Position	Description
1	Session ID

See Also

Reference

CDefaultQueryClass Class

Other Resources

CDefaultQueryClass Members

CDefaultQueryClass::GetSessionRefUpdateTimeout

Call this method to get the SQL query that will be executed to update the timeout of a particular session.

```
LPCTSTR GetSessionRefUpdateTimeout( ) throw( );
```

Return Value

Returns the SQL query that will be executed to update the timeout of a particular session.

Remarks

When implementing a compatible method, the query returned should have the following parameters:

Position	Description
1	Timeout in milliseconds
2	Session ID

See Also

Reference

CDefaultQueryClass Class

Other Resources

CDefaultQueryClass Members

CDefaultQueryClass::GetSessionVarCount

Call this method to get the SQL query that will be executed to get the count of variables in a particular session.

```
LPCTSTR GetSessionVarCount( ) throw( );
```

Return Value

Returns the SQL query that will be executed to get the count of variables in a particular session.

Remarks

When implementing a compatible method, the query returned should have the following parameters:

Position	Description
1	Session ID

See Also

Reference

CDefaultQueryClass Class

Other Resources

CDefaultQueryClass Members

CDefaultQueryClass::GetSessionVarDeleteAllVars

Call this method to get the SQL query that will be executed to delete all the variables in a particular session.

```
LPCTSTR GetSessionVarDeleteAllVars( ) throw( );
```

Return Value

Returns the SQL query that will be executed to delete all the variables in a particular session.

Remarks

When implementing a compatible method, the query returned should have the following parameters:

Position	Description
1	Session ID

See Also

Reference

CDefaultQueryClass Class

Other Resources

CDefaultQueryClass Members

CDefaultQueryClass::GetSessionVarDeleteVar

Call this method to get the SQL query that will be executed to delete a named variable from a particular session.

```
LPCTSTR GetSessionVarDeleteVar( ) throw( );
```

Return Value

Returns the SQL query that will be executed to delete a named variable from a particular session.

Remarks

When implementing a compatible method, the query returned should have the following parameters:

Position	Description
1	Session ID
2	Variable name

See Also

Reference

CDefaultQueryClass Class

Other Resources

CDefaultQueryClass Members

CDefaultQueryClass::GetSessionVarInsert

Call this method to get the SQL query that will be executed to insert a variable into a particular session.

```
LPCTSTR GetSessionVarInsert( ) throw( );
```

Return Value

Returns the SQL query that will be executed to insert a variable into a particular session.

Remarks

When implementing a compatible method, the query returned should have the following parameters:

Position	Description
1	Variable value
2	Session ID
3	Variable name

See Also

Reference

CDefaultQueryClass Class

CDefaultQueryClass::GetSessionVarUpdate

Other Resources

CDefaultQueryClass Members

CDefaultQueryClass::GetSessionVarSelectAllVars

Call this method to get the SQL query that will be executed to get information about all the variables in a particular session.

```
LPCTSTR GetSessionVarSelectAllVars( ) throw( );
```

Return Value

Returns the SQL query that will be executed to get information about all the variables in a particular session.

Remarks

When implementing a compatible method, the query returned should have the following parameters:

Position	Description
1	Session ID

See Also

Reference

CDefaultQueryClass Class

Other Resources

CDefaultQueryClass Members

CDefaultQueryClass::GetSessionVarSelectVar

Call this method to get the SQL query that will be executed to get information about a named variable in a particular session.

```
LPCTSTR GetSessionVarSelectVar( )throw( );
```

Return Value

Returns the SQL query that will be executed to get information about a named variable in a particular session.

Remarks

When implementing a compatible method, the query returned should have the following parameters:

Position	Description
1	Session ID
2	Variable name

See Also

Reference

CDefaultQueryClass Class

Other Resources

CDefaultQueryClass Members

CDefaultQueryClass::GetSessionVarUpdate

Call this method to get the SQL query that will be executed to update the value of a named variable in a particular session.

```
LPCTSTR GetSessionVarUpdate( ) throw( );
```

Return Value

Returns the SQL query that will be executed to update the value of a named variable in a particular session.

Remarks

When implementing a compatible method, the query returned should have the following parameters:

Position	Description
1	Variable value
2	Session ID
3	Variable name

See Also

Reference

CDefaultQueryClass Class

CDefaultQueryClass::GetSessionVarInsert

Other Resources

CDefaultQueryClass Members

CDllCache Class

This class implements a DLL cache.

```
template <
   class MonitorClass,
   class Peer = CNoDllCachePeer
>
class CDllCache :
   public IDllCache,
   public IWorkerThreadClient
```

Parameters

MonitorClass

A worker thread class used for performing periodic cache maintenance. Typically, this will be CWorkerThread, but CNoWorkerthread can be used to disable cache maintenance.

Peer

A class conforming to the DLL cache peer archetype that contains data specific to each cached DLL. If the default CNoDllCachePeer is used, the cache will not contain DLL peer data.

Remarks

CDllCache caches DLLs and implements IDllCache. The cache manages the loading and unloading of each DLL and uses reference counting to ensure that DLLs are not unloaded prematurely. With the help of *MonitorClass*, periodic cache maintenance is performed when Flush is called to remove stale entries from the cache, and flush items if the cache has reached the limited set through the IMemoryCacheControl interface. The frequency with which **Flush** is called is determined by the timeout value passed to Initialize.

Requirements

Header: atlcache.h

Example

See the Hotswap Sample.

See Also

Reference

IDllCache Interface
IWorkerThreadClient Interface
CNoDllCachePeer Class

Other Resources

CDllCache Members
Caching Reference
Caching

CDllCache Members

Methods

AddRef	Implementation of IUnknown::AddRef.
AddRefModule	Implementation of IDllCache::AddRefModule.
CDllCache	The constructor.
CloseHandle	Implementation of IWorkerThreadClient::CloseHandle.
Execute	Implementation of IWorkerThreadClient::Execute.
Flush	Implementation of IDllCache::Flush.
Free	Implementation of IDllCache::Free.
GetEntries	Implementation of IDllCache::GetEntries.
Initialize	Call this method to initialize the cache and specify the frequency with which asynchronous maintenance shoul d be performed.
Load	Implementation of IDllCache::Load.
QueryInterface	Implementation of IUnknown::QueryInterface.
Release	Implementation of IUnknown::Release.
ReleaseModule	Implementation of IDllCache::ReleaseModule.
Uninitialize	Call this method to remove all the DLLs from the cache and uninitialize the object.

See Also

Reference

CDllCache Class

Other Resources

Caching Reference

CDllCache Methods

For information about the methods in **CDllCache**, see CDllCache Members

CDllCache::AddRef

Implementation of IUnknown::AddRef.

```
ULONG STDMETHODCALLTYPE AddRef( );
```

Return Value

Always returns 1.

Remarks

This class does not implement reference counting.

See Also

Reference

CDllCache Class

Other Resources

CDllCache Members

Caching Reference

CDllCache::AddRefModule

Implementation of IDllCache::AddRefModule.

```
BOOL AddRefModule(
   HINSTANCE hInstDll
);
```

See Also

Reference

CDllCache Class

Other Resources

CDllCache Members

Caching Reference

CDllCache::CDllCache

The constructor.

```
CDllCache( );
```

Remarks

The **CDllCache** constructor performs no operations. Cache objects must be initialized with the Initialize method.

See Also

Reference

CDllCache Class

Other Resources

CDllCache Members

Caching Reference

CDllCache::CloseHandle

Implementation of IWorkerThreadClient::CloseHandle.

```
HRESULT CloseHandle(
   HANDLE hObject
);
```

Parameters

hObject

The handle of the cache monitoring timer object.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

This method is called to close the timer handle. The cache monitoring class, typically CWorkerThread, calls this method to disable the monitoring timer. It is not necessary to call **CloseHandle** explicitly.

See Also

Reference

CDllCache Class

Other Resources

CDllCache Members

Caching Reference

CDllCache::Execute

Implementation of IWorkerThreadClient::Execute.

```
HRESULT Execute(
   DWORD_PTR /* dwParam */,
   HANDLE /* hObject */
);
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The **Execute** method is typically called asynchronously by the *MonitorClass* used to construct the cache. If, however, the CNoWorkerThread class was used as the monitor class argument during construction, **Execute** will not be called.

It is not normally necessary to call this method explicitly.

See Also

Reference

CDllCache Class
CWorkerThread Class
IWorkerThreadClient Interface

Other Resources

CDllCache Members
Caching Reference

CDllCache::Flush

Implementation of IDllCache::Flush.

```
HRESULT Flush( );
```

Remarks

Flush is called automatically to remove stale entries from the cache with a frequency determined by the timeout value passed to Initialize. A DLL will be unloaded and removed from the cache if its reference count is zero and if it has not been used since the last time **Flush** was called.

Example

See the Hotswap Sample.

See Also

Reference

CDllCache Class

Other Resources

CDllCache Members

Caching Reference

CDllCache::Free

Implementation of IDllCache::Free.

```
BOOL Free(
   HINSTANCE hInstDll
);
```

See Also

Reference

CDllCache Class

IDllCache Interface

Other Resources

CDllCache Members

Caching Reference

CDllCache::GetEntries

Implementation of IDllCache::GetEntries.

```
HRESULT GetEntries(
   DWORD dwCount,
   DLL_CACHE_ENTRY* pEntries,
   DWORD* pdwCopied
);
```

See Also

Reference

CDllCache Class

IDllCache Interface

DLL_CACHE_ENTRY Structure

Other Resources

CDllCache Members

Caching Reference

CDllCache::Initialize

Call this method to initialize the cache and specify the frequency with which asynchronous maintenance should be performed.

```
HRESULT Initialize(
   DWORD dwTimeout = ATL_DLL_CACHE_TIMEOUT
);
template < class ThreadTraits >
HRESULT Initialize(
   CWorkerThread< ThreadTraits > * pWorkerThread,
   DWORD dwTimeout = ATL_DLL_CACHE_TIMEOUT
);
```

Parameters

dwTimeout

The time, in milliseconds, between asynchronous calls to Flush. The default timeout ATL_DLL_CACHE_TIMEOUT.

pWorkerThread

The address of a CWorkerThread object that this object will rely upon to perform periodic cache maintenance. If not provided, the object will create its own thread.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

This class uses CWorkerThread to perform periodic asynchronous maintenance. By default, each instance creates its own thread. One instance of CWorkerThread can, however, be used to service multiple objects by providing a worker thread pointer to the template version of **Initialize.**

See Also

Reference

CDllCache Class

CDllCache::Uninitialize

Other Resources

CDllCache Members

Caching Reference

CDllCache::Load

Implementation of IDllCache::Load.

```
HINSTANCE Load(
   LPCSTR szDllName,
   void * pPeerInfo
) throw(...);
```

See Also

Reference

CDllCache Class

IDllCache Interface

Other Resources

CDllCache Members

Caching Reference

CDllCache::QueryInterface

Implementation of IUnknown::QueryInterface.

```
HRESULT STDMETHODCALLTYPE QueryInterface(
   REFIID riid,
   void ** ppv
);
```

Remarks

Objects of this class can be successfully queried for the IUnknown and IDllCache interfaces.

See Also

Reference

CDllCache Class

Other Resources

CDllCache Members

Caching Reference

CDllCache::Release

Implementation of IUnknown::Release.

```
ULONG STDMETHODCALLTYPE Release( );
```

Return Value

Always returns 1.

Remarks

This class does not implement reference counting.

See Also

Reference

CDllCache Class

IUnknown

Other Resources

CDllCache Members

Caching Reference

CDllCache::ReleaseModule

Implementation of IDllCache::ReleaseModule.

```
BOOL ReleaseModule(
   HINSTANCE hInstDll
);
```

See Also

Reference

CDllCache Class

IDllCache Interface

Other Resources

CDllCache Members

Caching Reference

CDllCache::Uninitialize

Call this method to remove all the DLLs from the cache and uninitialize the object.

```
HRESULT Uninitialize( );
```

See Also

Reference

CDllCache Class

CDllCache::Initialize

Other Resources

CDllCache Members

Caching Reference

CDllCacheManager Class

This class implements the web service that allows you to manage the DLL cache in an ATL Server ISAPI extension.

```
[ soap_handler(
     name = ID_DLLCACHEMGR_WEBSERVICE_NAME,
     namespace = ID_DLLCACHEMGR_WEBSERVICE_URL,
     protocol = "soap"
   ),
   request_handler(
     name = ID_DLLCACHEMGR_WEBSERVICE_NAME,
     sdl = ID_DLLCACHEMGR_WEBSERVICE_WSDL
   )
]
class CDllCacheManager :
   public IDllCacheMgr
```

Remarks

See Extension Management Services for more details.

Requirements

Header: atlextmgmt.h

See Also

Reference

CSoapHandler Class
IDllCacheMgr Interface
soap_handler
request_handler
ID_DLLCACHEMGR_WEBSERVICE_NAME
ID_DLLCACHEMGR_WEBSERVICE_URL
ID_DLLCACHEMGR_WEBSERVICE_WSDL

Other Resources

CDllCacheManager Members

CDllCacheManager Members

Methods

GetEntries	Implementation of IDllCacheMgr::GetEntries.
GetEntryCount	Implementation of IDllCacheMgr::GetEntryCount.
HandleRequest	This method handles requests sent to the DLL cache manager service, including authorizing the client.

See Also

Reference

CDllCacheManager Class

Other Resources

IDllCacheMgr Members

CDllCacheManager Methods

For information about the methods in **CDllCacheManager**, see CDllCacheManager Members

CDllCacheManager::GetEntries

Implementation of IDllCacheMgr::GetEntries.

```
[soap_method]
HRESULT GetEntries(
   DWORD dwCount,
   _DLL_CACHE_ENTRY * pEntries,
   DWORD * pdwCopied
);
```

See Also

Reference

CDllCacheManager Class

Other Resources

CDllCacheManager Members

CDllCacheManager::GetEntryCount

Implementation of IDllCacheMgr::GetEntryCount.

```
[soap_method]
STDMETHOD(GetEntryCount)(
   DWORD * pdwCount
);
```

See Also

Reference

CDllCacheManager Class

Other Resources

CDllCacheManager Members

CDllCacheManager::HandleRequest

This method handles requests sent to the DLL cache manager service, including authorizing the client.

```
HTTP_CODE HandleRequest(
   AtlServerRequest* pRequestInfo,
   IServiceProvider* pProvider
);
```

Parameters

pRequestInfo

The information about the request to be handled.

pProvider

The service provider.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

CDllCacheManager Class

Other Resources

CDllCacheManager Members

CDllCacheMgr Class

This class implements the request handler that allows you to manage the DLL cache in an ATL Server ISAPI extension using an HTML interface.

```
[ request_handler(name=ID_DLLCACHEMGR_SRFHANDLER_NAME) ]
class CDllCacheMgr
```

Remarks

See Extension Management Services for more details.

Requirements

Header: atlextmgmt.h

See Also

Reference

CRequestHandlerT Class
request_handler
ID_DLLCACHEMGR_SRFHANDLER_NAME

Other Resources

CDllCacheMgr Members

CDllCacheMgr Members

Methods

CDllCacheMgr	The constructor.
EnumEntries	This replacement method can be used in a server response file to get information about the next cache entry so that it can be returned by GetDllName and GetDllReferences.
GetBodyColor	This replacement method can be used in a server response file to write a color value for the body of the HTML page.
GetDllName	This replacement method can be used in a server response file to get the name of the current cache entry.
GetDllReferences	This replacement method can be used in a server response file to get the reference count of the current cache entry.
GetNumEntries	This replacement method can be used in a server response file to get the number of entries in the cache.
GetTRColor	This replacement method can be used in a server response file to write a color value for the current row.
ValidateAndExchange	This method validates requests sent to the request handler, including authorizing the client.

See Also

Reference

CDllCacheMgr Class

CDllCacheMgr Methods

For information about the methods in **CDllCacheMgr**, see CDllCacheMgr Members

CDllCacheMgr::CDllCacheMgr

The constructor.

```
CDllCacheMgr( );
```

See Also

Reference

CDllCacheMgr Class

Other Resources

CDllCacheMgr Members

CDllCacheMgr::EnumEntries

This replacement method can be used in a server response file to get information about the next cache entry so that it can be returned by GetDllName and GetDllReferences.

```
[ tag_name( "EnumEntries" ) ]
HTTP_CODE EnumEntries( );
```

Return Value

Returns HTTP_SUCCESS if information about a cache entry can be obtained, HTTP_S_FALSE if there are no more entries, or an ATL Server status code indicating failure.

Remarks

This replacement method is designed to be used in a **while** replacement tag.

See Also

Reference

CDllCacheMgr Class

Other Resources

CDllCacheMgr Members

CDllCacheMgr::GetBodyColor

This replacement method can be used in a server response file to write a color value for the body of the HTML page.

```
[ tag_name("GetBodyColor") ]
HTTP_CODE GetBodyColor( );
```

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

CDllCacheMgr Class

CDllCacheMgr::GetTRColor

Other Resources

CDllCacheMgr Members

CDllCacheMgr::GetDllName

This replacement method can be used in a server response file to get the name of the current cache entry.

```
[ tag_name( "GetDllName" )]
HTTP_CODE GetDllName( );
```

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

The current cache entry is set by using the EnumEntries replacement method.

See Also

Reference

CDllCacheMgr Class

Other Resources

CDllCacheMgr Members

CDllCacheMgr::GetDllReferences

This replacement method can be used in a server response file to get the reference count of the current cache entry.

```
[ tag_name( "GetDllReferences" ) ]
HTTP_CODE GetDllReferences( );
```

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

The current cache entry is set by using the EnumEntries replacement method.

See Also

Reference

CDllCacheMgr Class

Other Resources

CDllCacheMgr Members

CDllCacheMgr::GetNumEntries

This replacement method can be used in a server response file to get the number of entries in the cache.

```
[ tag_name( "GetNumEntries" ) ]
HTTP_CODE GetNumEntries( );
```

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

Corresponds to IDllCacheMgr::GetEntryCount.

See Also

Reference

CDllCacheMgr Class

Other Resources

CDllCacheMgr Members

CDllCacheMgr::GetTRColor

This replacement method can be used in a server response file to write a color value for the current row.

```
[ tag_name( "GetTRColor" ) ]
HTTP_CODE GetTRColor( );
```

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

The color value written by this method switches between one of two colors on each call.

See Also

Reference

CDllCacheMgr Class

CDllCacheMgr::GetBodyColor

Other Resources

CDllCacheMgr Members

CDllCacheMgr::ValidateAndExchange

This method validates requests sent to the request handler, including authorizing the client.

```
HTTP_CODE ValidateAndExchange( );
```

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

CDllCacheMgr Class

Other Resources

CDllCacheMgr Members

CDllCachePeer Class

This class conforms to the DLL cache peer archetype and is used to cache pointers to ATL Server request handler functions.

```
class CDllCachePeer
```

Remarks

This class is passed as a template argument to CDllCache by CIsapiExtension. **CIsapiExtension** uses this class for caching handles, function pointers, and other useful information related to ATL Server request handler DLLs.

Requirements

Header: atlisapi.h

See Also

Other Resources

CDllCachePeer Members

ATL Server Classes

CDllCachePeer Members

Methods

Add	Implementation of DllCachePeerArchetype::Add.
Remove	Implementation of DllCachePeerArchetype::Remove.

Structures

DllInfo	Implementation of DllCachePeerArchetype::DllInfo.

See Also

Reference

CDllCachePeer Class

CDllCachePeer Methods

For information about the methods in **CDllCachePeer**, see CDllCachePeer Members

CDllCachePeer::Add

Implementation of DllCachePeerArchetype::Add.

```
BOOL Add(
   HINSTANCE hInst,
   DllInfo* pInfo
) throw(...);
```

Remarks

This method assumes that the pContext and pExtension members of *pInfo* are correctly initialized to NULL or valid pointer values before this method is called.

Add calls GetProcAddress to get the pointers to the three ATL Server request handler DLL functions that it stores in *pInfo*->pfnInitHandlers, *pInfo*->pfnUninitHandlers, and *pInfo*->pfnGetHandler. **Add** then uses *pInfo*->pfnInitHandlers to initialize the DLL. Finally, it sets *pInfo*->pContext to NULL.

See Also

Reference

CDllCachePeer Class

Other Resources

CDllCachePeer Members

CDllCachePeer::Remove

Implementation of DllCachePeerArchetype::Remove.

```
void Remove(
   HINSTANCE /*hInst*/,
   DllInfo* pInfo
) throw(...);
```

Remarks

Calls *pInfo*->pfnUninitHandlers to uninitialize the DLL.

See Also

Reference

CDllCachePeer Class

Other Resources

CDllCachePeer Members

CDllCachePeer Structures

For information about the structures in **CDllCachePeer**, see CDllCachePeer Members

CDllCachePeer::DllInfo Structure

Implementation of DllCachePeerArchetype::DllInfo.

```
struct DllInfo :
   public ATLServerDllInfo
```

Remarks

This structure adds an assignment operator to the members inherited from ATLServerDllInfo.

See Also

Reference

CDllCachePeer Class

ATLServerDllInfo Structure

Other Resources

CDllCachePeer Members

CDllMgrObject Class

This class provides methods that can be used to implement IDllCacheMgr by wrapping calls to IDllCache.

```
class CDllMgrObject
```

Remarks

This class is used in the implementation of the extension management services for the ISAPI extension's DLL cache.

Requirements

Header: atlextmgmt.h

See Also

Other Resources

CDllMgrObject Members

CDllMgrObject Members

Methods

GetEntries	Delegates to IDllCache::GetEntries on the interface obtained from the service provider passed to Initialize.
GetEntryCount	Call this method to get the number of items in the DLL cache.
Initialize	Call this method to initialize the object.

See Also

Reference

CDllMgrObject Class

CDllMgrObject Methods

For information about the methods in **CDllMgrObject**, see CDllMgrObject Members

CDllMgrObject::GetEntries

Delegates to IDllCache::GetEntries on the interface obtained from the service provider passed to Initialize.

```
HRESULT GetEntries(
   DWORD dwCount,
   _DLL_CACHE_ENTRY* pEntries,
   DWORD* pdwCopied
);
```

Remarks

This method converts between the DLL_CACHE_ENTRY structures used by IDllCache::GetEntries and the _DLL_CACHE_ENTRY structures used by this method.

See Also

Reference

CDllMgrObject Class

_DLL_CACHE_ENTRY Structure

Other Resources

CDllMgrObject Members

CDllMgrObject::GetEntryCount

Call this method to get the number of items in the DLL cache.

```
HRESULT GetEntryCount(
   DWORD* pdwCount
);
```

Parameters

pdwCount

[out] Address of the variable that, on success, receives the number of items in the DLL cache.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls IDllCache::GetEntries on the interface obtained from the service provider passed to Initialize to obtain the value returned by this method.

See Also

Reference

CDllMgrObject Class

Other Resources

CDllMgrObject Members

CDllMgrObject::Initialize

Call this method to initialize the object.

```
HTTP_CODE Initialize(
   IServiceProvider* pProvider
);
```

Parameters

pProvider

The service provider to be queried for the IDllCache service.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

CDllMgrObject Class

Other Resources

CDllMgrObject Members

CExpireCuller Class

This class conforms to the cache culler archetype and allows the automatic removal of cache items that exceed the absolute expiration time set by users of the cache.

```
class CExpireCuller
```

Remarks

CExpireCuller keeps a linked list of CCullerCacheData objects sorted in order of their absolute expiration time and allows items that have exceeded their expiration time to be removed from the cache. This class interprets an expiration time of zero as meaning that the item should never expire.

Requirements

Header: atlcache.h

See Also

Reference

CLifetimeCuller Class
CFixedLifetimeCuller Class
COrCullers Class
CNoExpireCuller Class

Other Resources

CExpireCuller Members
Caching Reference
Caching

CExpireCuller Members

Methods

Access	Implementation of CacheCullerArchetype::Access.
Add	Implementation of CacheCullerArchetype::Add.
CExpireCuller	The constructor.
Commit	Implementation of CacheCullerArchetype::Commit.
GetExpired	Implementation of CacheCullerArchetype::GetExpired.
IsExpired	Implementation of CacheCullerArchetype::IsExpired.
Release	Implementation of CacheCullerArchetype::Release.
Remove	Implementation of CacheCullerArchetype::Remove.
Start	Implementation of CacheCullerArchetype::Start.

Data Members

m_cftCurrent	Time used for comparisons in IsExpired and GetExpired.
pHead	The head of the linked list.
pTail	The tail of the linked list.

See Also

Reference

CExpireCuller Class

Other Resources

Caching

Caching Reference

CExpireCuller Methods

For information about the methods in **CExpireCuller**, see CExpireCuller Members

CExpireCuller::Access

Implementation of CacheCullerArchetype::Access.

```
void Access(
   CCullerCacheData * /* pItem */
);
```

Remarks

This implementation does nothing. Items are removed according to the absolute expiration time set when they are added to the cache.

See Also

Reference

CExpireCuller Class

Cache Culler Archetype

Other Resources

CExpireCuller Members

Caching

Caching Reference

CExpireCuller::Add

Implementation of CacheCullerArchetype::Add.

```
void Add(
   CCullerCacheData* pItem
);
```

Remarks

This implementation does nothing. Code associated with adding an item is provided by Commit.

See Also

Reference

CExpireCuller Class
Cache Culler Archetype

Other Resources

CExpireCuller Members
Caching
Caching Reference

CExpireCuller::CExpireCuller

The constructor.

```
CExpireCuller( );
```

Remarks

Initializes all data members to zero.

See Also

Reference

CExpireCuller Class

Cache Culler Archetype

Other Resources

CExpireCuller Members

Caching

Caching Reference

CExpireCuller::Commit

Implementation of CacheCullerArchetype::Commit.

```
void Commit(
   CCullerCacheData* pItem
);
```

Remarks

Adds the item to the linked list pointed to by pHead and pTail in expiration order. Items that expire first are at the head of the list.

See Also

Reference

CExpireCuller Class
Cache Culler Archetype

Other Resources

CExpireCuller Members
Caching
Caching Reference

CExpireCuller::GetExpired

Implementation of CacheCullerArchetype::GetExpired.

```
CCullerCacheData* GetExpired( );
```

Remarks

Returns pHead if it has expired, otherwise returns **NULL**.

See Also

Reference

CExpireCuller Class

Other Resources

CExpireCuller Members

Caching

Caching Reference

CExpireCuller::IsExpired

Implementation of CacheCullerArchetype::IsExpired.

```
BOOL IsExpired(
   CCullerCacheData* pItem
);
```

Remarks

Returns TRUE if *pItem*->cftExpireTime is greater than the time set when CExpireCuller::Start was called.

See Also

Reference

CExpireCuller Class

Other Resources

CExpireCuller Members

Caching

Caching Reference

CExpireCuller::Release

Implementation of CacheCullerArchetype::Release.

```
void Release(
   CCullerCacheData * /* pItem */
);
```

Remarks

This implementation does nothing. Code associated with adding an item is provided by Commit.

See Also

Reference

CExpireCuller Class

Other Resources

CExpireCuller Members

Caching

Caching Reference

CExpireCuller::Remove

Implementation of CacheCullerArchetype::Remove.

```
void Remove(
   CCullerCacheData* pItem
);
```

Remarks

Removes the item from the linked list.

See Also

Reference

CExpireCuller Class

Other Resources

CExpireCuller Members

Caching

Caching Reference

CExpireCuller::Start

Implementation of CacheCullerArchetype::Start.

```
void Start( );
```

Remarks

Sets the current time to use for comparisons in IsExpired and GetExpired.

See Also

Reference

CExpireCuller Class

Other Resources

CExpireCuller Members

Caching

Caching Reference

CExpireCuller Data Members

For information about the data members in **CExpireCuller**, see CExpireCuller Members

CExpireCuller::m_cftCurrent

Time used for comparisons in IsExpired and GetExpired.

```
CFileTime m_cftCurrent;
```

Remarks

This member is set to the current time when Start is called.

See Also

Reference

CExpireCuller Class

Cache Culler Archetype

Other Resources

CExpireCuller Members

Caching

Caching Reference

CExpireCuller::pHead

The head of the linked list.

```
CCullerCacheData * pHead;
```

See Also

Reference

CExpireCuller Class

Cache Culler Archetype

Other Resources

CExpireCuller Members

Caching

Caching Reference

CExpireCuller::pTail

The tail of the linked list.

```
CCullerCacheData * pTail;
```

See Also

Reference

CExpireCuller Class

Cache Culler Archetype

Other Resources

CExpireCuller Members

Caching

Caching Reference

CFileCache Class

This class implements a file cache.

```
template <
   class MonitorClass,
   class StatClass     = CStdStatClass,
   class FileCachePeer = CNoFileCachePeer,
   class FlushClass    = COldFlusher,
   class SyncClass     = CComCriticalSection,
   class CullClass     = CExpireCuller
>
class CFileCache :
   public CMemoryCacheBase< CFileCache, LPSTR,
               CCacheDataPeer< FileCachePeer >, CFixedStringKey,
               CStringElementTraits< CFixedStringKey >, FlushClass,
               CullClass, SyncClass, StatClass >,
   public IWorkerThreadClient,
   public IFileCache,
   public IMemoryCacheControl,
   public IMemoryCacheStats
```

Parameters

MonitorClass

The class to be used for periodic cache maintenance. Normally this will be CWorkerThread. If no cache maintenance is desired, use CNoWorkerThread.

StatClass

The class used to collect and expose statistics related to the use of the cache including such information as effectiveness of the cache and memory used. Must conform to the cache statistics archetype.

FlushClass

The class used to determine whether, and in what order, items should be removed from the cache using the number and sequence of accesses as the basis for making the decision. Must conform to the cache flusher archetype.

This class is used to remove items from the cache in order to comply with the size and memory constraints specified through the IMemoryCacheControl interface.

SyncClass

The class to be used for thread synchronization. The default value of CComCriticalSection is suitable for most situations, but the CComFakeCriticalSection class can be used to avoid thread synchronization overhead in single-threaded environments.

CullClass

The class used to determine whether, and in what order, items should be removed from the cache using time-based criteria. Must conform to the cache culler archetype.

Remarks

Although **CFileCache** is a memory-based cache, it does not store files in memory. Only the name of the file is stored in the cache. The file itself remains on disk and is managed by the cache only in the sense that it is deleted when its cache entry expires. As a result, it is very important to provide the file cache with the names of expendable files.

Requirements

Header: atlcache.h

See Also

Tasks

BlobCache Sample: Generic Memory Caching

StencilCache Sample: Uses the ATL Server Stencil Cache

Reference

CMemoryCacheBase Class
IWorkerThreadClient Interface
IFileCache Interface
IMemoryCacheControl Interface
IMemoryCacheStats Interface

Other Resources

Caching
CFileCache Members
Caching Reference

CFileCache Members

Methods

AddFile	Implementation of IFileCache::AddFile.
AddRef	Implementation of IUnknown::AddRef.
~CFileCache	The destructor.
ClearStats	Implementation of IMemoryCacheStats::ClearStats.
CloseHandle	Implementation of IWorkerThreadClient::CloseHandle.
Execute	Implementation of IWorkerThreadClient::Execute.
Flush	Implementation of IFileCache::Flush.
GetCurrentAllocSize	Implementation of IMemoryCacheStats::GetCurrentAllocSize.
GetCurrentEntryCount	Implementation of IMemoryCacheStats::GetCurrentEntryCount.
GetFile	Implementation of IFileCache::GetFile.
GetHitCount	Implementation of IMemoryCacheStats::GetHitCount.
GetMaxAllocSize	Implementation of IMemoryCacheStats::GetMaxAllocSize.
GetMaxAllowedEntries	Implementation of IMemoryCacheControl::GetMaxAllowedEntries.
GetMaxAllowedSize	Implementation of IMemoryCacheControl::GetMaxAllowedSize.
GetMaxEntryCount	Implementation of IMemoryCacheStats::GetMaxEntryCount.
GetMissCount	Implementation of IMemoryCacheStats::GetMissCount.
Initialize	Call this method to initialize the file cache.
LookupFile	Implementation of IFileCache::LookupFile.
OnDestroyEntry	This method is called to destroy a file cache entry.
QueryInterface	Implementation of IUnknown::QueryInterface.
Release	Implementation of IUnknown::Release.
ReleaseFile	Implementation of IFileCache::ReleaseFile.
RemoveFile	Implementation of IFileCache::RemoveFile.
RemoveFileByName	Implementation of IFileCache::RemoveFileByName.

ResetCache	Implementation of IMemoryCacheControl::ResetCache.
SetMaxAllowedEntries	Implementation of IMemoryCacheControl::SetMaxAllowedEntries.
SetMaxAllowedSize	Implementation of IMemoryCacheControl::SetMaxAllowedSize.
Uninitialize	Call this method to uninitialize the cache.

See Also

Reference

CFileCache Class

Other Resources

Caching

Caching Reference

CFileCache Methods

For information about the methods in **CFileCache**, see CFileCache Members.

CFileCache::AddFile

Implementation of IFileCache::AddFile.

```
HRESULT STDMETHODCALLTYPE AddFile(
   LPCSTR szFileName,
   LPCSTR szTempFileName,
   FILETIME* pftExpireTime,
   void* pPeerInfo,
   HCACHEITEM* phKey = NULL
);
```

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::AddRef

Implementation of IUnknown::AddRef.

```
ULONG STDMETHODCALLTYPE AddRef( );
```

Return Value

Always returns 1.

Remarks

This class does not implement lifetime control through reference counting.

See Also

Reference

CFileCache Class

CFileCache::Release

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::~CFileCache

The destructor.

```
~CFileCache( );
```

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::ClearStats

Implementation of IMemoryCacheStats::ClearStats.

```
HRESULT STDMETHODCALLTYPE ClearStats( );
```

Example

See the BlobCache Sample.

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::CloseHandle

Implementation of IWorkerThreadClient::CloseHandle.

```
HRESULT CloseHandle(
   HANDLE hObject
);
```

Remarks

The cache monitoring class, typically CWorkerThread, calls this method to disable the monitoring timer. It is not necessary to call this method explicitly.

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::Execute

Implementation of IWorkerThreadClient::Execute.

```
HRESULT Execute(
   DWORD_PTR dwParam,
   HANDLE /*hObject*/
);
```

Remarks

The **Execute** method performs cache maintenance. The effects on the cache vary, depending on the flushing and culling template parameters used to instantiate the CFileCache template, and the settings specified through the IMemoryCacheControl interface.

If CNoWorkerThread was used as the monitor class argument during construction, the **Execute** method will not be called.

It is not necessary to call this method explicitly.

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::Flush

Implementation of IFileCache::Flush.

```
HRESULT STDMETHODCALLTYPE Flush( );
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The **Flush** method is called by the monitor class used to construct the file cache, unless the CNoWorkerThread class was used. **Flush** can also be called explicitly.

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::GetCurrentAllocSize

Implementation of IMemoryCacheStats::GetCurrentAllocSize.

```
HRESULT STDMETHODCALLTYPE GetCurrentAllocSize(
   DWORD* pdwSize
);
```

Example

See the StencilCache Sample.

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::GetCurrentEntryCount

Implementation of IMemoryCacheStats::GetCurrentEntryCount.

```
HRESULT STDMETHODCALLTYPE GetCurrentEntryCount(
   DWORD* pdwSize
);
```

Example

See the BlobCache Sample and the StencilCache Sample.

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::GetFile

Implementation of IFileCache::GetFile.

```
HRESULT STDMETHODCALLTYPE GetFile(
   const HCACHEITEM hKey,
   LPSTR* pszFileName,
   void** ppPeerInfo
);
```

Parameters

hKey

A cache item handle, retrieved previously with either the AddFile or LookupFile methods, indicating the desired file name.

pszFileName

The address of a string that is to receive the cached file name. This retrieved name is the name of the temporary file that will be deleted by **CFileCache** when the cache entry is destroyed.

ppPeerInfo

The address of a pointer that is to receive file peer information. The data type of this information depends on the class used as the *FileCachePeer* template argument to CFileCache. By default, the CNoFileCachePeer class is used, which provides no peer data. Use **NULL** if no peer information is to be retrieved.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::GetHitCount

Implementation of IMemoryCacheStats::GetHitCount.

```
HRESULT STDMETHODCALLTYPE GetHitCount(
   DWORD* pdwSize
);
```

Example

See the BlobCache Sample and the StencilCache Sample.

See Also

Reference

CFileCache Class
CFileCache::GetMissCount

Other Resources

Caching
CFileCache Members
Caching Reference

CFileCache::GetMaxAllocSize

Implementation of IMemoryCacheStats::GetMaxAllocSize.

```
HRESULT STDMETHODCALLTYPE GetMaxAllocSize(
   DWORD* pdwSize
);
```

Example

See the StencilCache Sample.

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::GetMaxAllowedEntries

Implementation of IMemoryCacheControl::GetMaxAllowedEntries

```
HRESULT STDMETHODCALLTYPE GetMaxAllowedEntries(
   DWORD* pdwSize
);
```

Example

See the BlobCache Sample.

See Also

Reference

CFileCache Class

CFileCache::SetMaxAllowedEntries

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::GetMaxAllowedSize

Implementation of IMemoryCacheControl::GetMaxAllowedSize

```
HRESULT STDMETHODCALLTYPE GetMaxAllowedSize(
   DWORD* pdwSize
);
```

See Also

Reference

CFileCache Class

CFileCache::SetMaxAllowedSize

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::GetMaxEntryCount

Implementation of IMemoryCacheStats::GetMaxEntryCount.

```
HRESULT STDMETHODCALLTYPE GetMaxEntryCount(
   DWORD* pdwSize
);
```

Example

See the BlobCache Sample and the StencilCache Sample.

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::GetMissCount

Implementation of IMemoryCacheStats::GetMissCount.

```
HRESULT STDMETHODCALLTYPE GetMissCount(
   DWORD* pdwSize
);
```

Example

See the BlobCache Sample and the StencilCache Sample.

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::Initialize

Call this method to initialize the file cache.

```
HRESULT Initialize( );
template <class ThreadTraits>
HRESULT Initialize(
   CWorkerThread<ThreadTraits> * pWorkerThread
);
```

Parameters

pWorkerThread

The address of an existing worker thread object that this object should use to perform periodic cache maintenance.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

This method initializes data structures, and must be called before any other methods are called. The templatized version allows multiple objects to share the overhead of a single worker thread. Otherwise this object will create its own worker thread.

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::LookupFile

Implementation of IFileCache::LookupFile.

```
HRESULT STDMETHODCALLTYPE LookupFile(
   LPCSTR szFileName,
   HCACHEITEM* phKey
);
```

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::OnDestroyEntry

This method is called to destroy a file cache entry.

```
virtual void OnDestroyEntry(
   const NodeType* pEntry
);
```

Parameters

pEntry

The address of the cache node that is to be destroyed.

Remarks

The **OnDestroyEntry** method is called by the base class, CMemoryCacheBase, to allow derived classes to perform custom node destruction steps. **CFileCache** overrides this method to delete the temporary file to which the cache entry corresponds. This behavior can be overridden by deriving from **CFileCache** and providing a custom version of **OnDestroyEntry**.

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::QueryInterface

Implementation of IUnknown::QueryInterface.

```
HRESULT STDMETHODCALLTYPE QueryInterface(
   REFIID riid,
   void** ppv
);
```

Remarks

Objects of this class can be successfully queried for the **IUnknown**, IFileCache, ImemoryCacheStats, and IMemoryCacheControl interfaces.

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::Release

Implementation of IUnknown::Release.

```
ULONG STDMETHODCALLTYPE Release( );
```

Return Value

Always returns 1.

Remarks

This class does not implement lifetime control through reference counting.

See Also

Reference

CFileCache Class
CFileCache::AddRef

Other Resources

Caching
CFileCache Members
Caching Reference

CFileCache::ReleaseFile

Implementation of IFileCache::ReleaseFile.

```
HRESULT STDMETHODCALLTYPE ReleaseFile(
   const HCACHEITEM hKey
);
```

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::RemoveFile

Implementation of IFileCache::RemoveFile.

```
HRESULT STDMETHODCALLTYPE RemoveFile(
   const HCACHEITEM hKey
);
```

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::RemoveFileByName

Implementation of IFileCache::RemoveFileByName.

```
HRESULT STDMETHODCALLTYPE RemoveFileByName(
   LPCSTR szFileName
);
```

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::ResetCache

Implementation of IMemoryCacheControl::ResetCache.

```
HRESULT STDMETHODCALLTYPE ResetCache( );
```

Example

See the BlobCache Sample.

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::SetMaxAllowedEntries

Implementation of IMemoryCacheControl::SetMaxAllowedEntries.

```
HRESULT STDMETHODCALLTYPE SetMaxAllowedEntries(
   DWORD dwSize
);
```

Example

See the BlobCache Sample.

See Also

Reference

CFileCache Class

CFileCache::GetMaxAllowedEntries

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::SetMaxAllowedSize

Implementation of IMemoryCacheControl::SetMaxAllowedSize.

```
HRESULT STDMETHODCALLTYPE SetMaxAllowedSize(
   DWORD dwSize
);
```

See Also

Reference

CFileCache Class

CFileCache::GetMaxAllowedSize

Other Resources

Caching

CFileCache Members

Caching Reference

CFileCache::Uninitialize

Call this method to uninitialize the cache.

```
HRESULT Uninitialize( );
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Releases all entries in the cache.

See Also

Reference

CFileCache Class

Other Resources

Caching

CFileCache Members

Caching Reference

CFixedLifetimeCuller Class

This class conforms to the cache culler archetype and allows the automatic removal of cache items that haven't been accessed in a certain period. The lifetime of items in the cache cannot be changed at runtime. The lifetime is the same for all items.

```
template < __int64 ftLifespan >
class CFixedLifetimeCuller :
   public CExpireCuller
```

Parameters

ftLifespan

The period of inactivity after which a cache entry expires.

Remarks

This class interprets a life span of zero as meaning that the item should never expire.

Requirements

Header: atlcache.h

See Also

Reference

CExpireCuller Class

Other Resources

CFixedLifetimeCuller Members

Caching

Caching Reference

CFixedLifetimeCuller Members

Methods

Access	Implementation of CacheCullerArchetype::Access.
Commit	Implementation of CacheCullerArchetype::Commit.
GetExpired	Implementation of CacheCullerArchetype::GetExpired.

See Also

Reference

CFixedLifetimeCuller Class

Other Resources

Caching

Caching Reference

CFixedLifetimeCuller Methods

For information about the methods in **CFixedLifetimeCuller**, see CFixedLifetimeCuller Members

CFixedLifetimeCuller::Access

Implementation of CacheCullerArchetype::Access.

```
void Access(
   CCullerCacheData* pItem
);
```

Remarks

Sets the expiration time of the item by adding the fixed life span, *ftLifespan*, to the current time.

See Also

Reference

CFixedLifetimeCuller Class
Cache Culler Archetype

Other Resources

CFixedLifetimeCuller Members
Caching
Caching Reference

CFixedLifetimeCuller::Commit

Implementation of CacheCullerArchetype::Commit.

```
void Commit(
   CCullerCacheData* pItem
);
```

Remarks

Sets the expiration time of the item by adding the fixed life span, *ftLifespan*, to the current time.

See Also

Reference

CFixedLifetimeCuller Class
Cache Culler Archetype

Other Resources

CFixedLifetimeCuller Members
Caching
Caching Reference

CFixedLifetimeCuller::GetExpired

Implementation of CacheCullerArchetype::GetExpired.

```
CCullerCacheData* GetExpired( );
```

Remarks

Delegates to CExpireCuller::GetExpired.

See Also

Reference

CFixedLifetimeCuller Class
Cache Culler Archetype

Other Resources

CFixedLifetimeCuller Members
Caching
Caching Reference

CFlusherCacheData Structure

This structure holds information about an entry in a cache that can be used by classes conforming to the cache flusher archetype to determine whether, and in what order, items should be removed from the cache.

```
struct CFlusherCacheData
```

Remarks

Each entry in a CMemoryCacheBase-derived cache has an associated **CFlusherCacheData** object that holds information about the number of times that item has been accessed. This information can be set and read by a cache flusher to provide an access-based mechanism for removing items from the cache. Each **CFlusherCacheData** object also serves as a doubly-linked list node so that the cache flusher can quickly find the next candidate for removal from the cache.

Requirements

Header: atlcache.h

See Also

Other Resources

CFlusherCacheData Members
Caching
Caching Reference

CFlusherCacheData Members

Methods

CFlusherCacheData	The constructor.

Data Members

dwAccessed	The number of times that the item corresponding to this object has been retrieved from the cache.
pNext	The address of the next item in a linked list of **CFlusherCacheData** structures.
pPrev	The address of the previous item in a linked list of **CFlusherCacheData** structures.

See Also

Reference

CFlusherCacheData Structure

CFlusherCacheData Methods

For information about the methods in **CFlusherCacheData**, see CFlusherCacheData Members

CFlusherCacheData::CFlusherCacheData

The constructor.

```
CFlusherCacheData( );
```

Remarks

Initializes all data members to zero.

See Also

Reference

CFlusherCacheData Structure

Other Resources

CFlusherCacheData Members

CFlusherCacheData Data Members

For information about the data members in **CFlusherCacheData**, see CFlusherCacheData Members

CFlusherCacheData::dwAccessed

The number of times that the item corresponding to this object has been retrieved from the cache.

```
DWORD dwAccessed;
```

See Also

Reference

CFlusherCacheData Structure

Other Resources

CFlusherCacheData Members

Caching

Caching Reference

CFlusherCacheData::pNext

The address of the next item in a linked list of **CFlusherCacheData** structures.

```
CFlusherCacheData* pNext;
```

Remarks

Will be NULL if this node represents the end of the list.

See Also

Reference

CFlusherCacheData Structure

Other Resources

CFlusherCacheData Members

Caching

Caching Reference

CFlusherCacheData::pPrev

The address of the previous item in a linked list of **CFlusherCacheData** structures.

```
CFlusherCacheData* pPrev;
```

Remarks

Will be NULL if this node represents the start of the list.

See Also

Reference

CFlusherCacheData Structure

Other Resources

CFlusherCacheData Members

Caching

Caching Reference

CHtmlGen Class

This class is used to generate HTML elements in an output stream.

```
class CHtmlGen :
   public CHtmlGenBase< CHtmlGen >
```

Remarks

This class is just a specialization of CHtmlGenBase. It doesn't add any members of its own.

Requirements

Header: atlhtml.h

See Also

Reference

CHtmlGenBase Class

Other Resources

ATL Server Classes

CHtmlGenBase Class

This class is used to generate HTML elements in a text stream.

```
template <
   class T
>
class CHtmlGenBase :
   public CStreamFormatter
```

Parameters

T

A class derived from **CHtmlGenBase**. Used to allow overriding of nonvirtual functions.

Remarks

This class is designed as a base class; you should derive a class from **CHtmlGenBase**, passing the new class's name as the template parameter. CHtmlGen is already defined for your convenience.

Use the Initialize method to assign a stream (IStream) to a **CHtmlGenBase**-derived object:

```
CHtmlGen h;
h.Initialize( &aStream );
```

CHtmlGenBase provides convenient methods for injecting HTML into a stream. For example, the a and aEnd methods inject `<a>` and `</a>` tags, respectively.

```
h.a( "www.microsoft.com" );  // injects <a href="www.microsoft.com">
h.WriteRaw( "Microsoft" );      // injects Microsoft
h.aEnd();                    // injects </a>
```

The second argument to **a**, if present, is the anchor content:

```
h.a( "www.microsoft.com", "Microsoft" );
// injects <a href="www.microsoft.com">Microsoft</a>
```

Note that the `</a>` tag is injected automatically in this case. The third argument, if present, specifies any additional attributes:

```
h.a( "www.microsoft.com", "Microsoft", "target=\"_blank\"" );
// injects <a target="_blank" href="www.microsoft.com">Microsoft</a>
```

Additional attributes can be given as a string, as in this example, or can be built up using the AtlHtmlAttrs class.

Most methods in **CHtmlGenBase** correspond to HTML elements and are used like the **a** and **aEnd** methods demonstrated above.

You can use the SetScheme method to set certain default attributes, such as background color. For more information, see HTML_SCHEME.

Example

This program generates the HTML for a web page that includes a table and a form.

```
#include "stdafx.h"
#include "mystream.h"
#include <atlhtml.h>

int main(int argc, char* argv[])
{
   CMyStream ms;
```

```
    HTML_SCHEME scheme;

    scheme.strTableBgColor="#a0a0a0";
    scheme.strBgColor="#fefefe";
    scheme.strLinkColor="#ff0000";
    scheme.strALinkColor="#00ff00";
    scheme.strVLinkColor="#0000ff";

    CHtmlGen html;
    html.Initialize( &ms );
    html.SetScheme(&scheme);

    html.html();
    html.head();
    html.title("Test Html generation classes");
    html.headEnd();

    html.body();

    html.a("http://www.microsoft.com", "Microsoft");
    html.br();

    html.SetSizePercent( 75, 25 );
    html.table(1, AtlHtmlAttrs( "cellpadding=%d", 2 ) );
    for (int i=0; i<4; i++)
    {
       scheme.strTdBgColor="#f0f0f0";
       html.tr();
       html.td("Hello");
       scheme.strTdBgColor.Empty();
       html.td("test");
       html.tdEnd();
       html.trEnd();
    }
    html.tableEnd();

    html.form("http://localhost/vcisapi/test");
    html.WriteRaw("Name: ");
    html.input("text", "name", "someone");
    html.submit();
    html.formEnd();

    html.bodyEnd();
    html.htmlEnd();

    return 0;
}
```

Output (reformatted for clarity)

```
<html>
<head>
<title>Test Html generation classes</title>
</head>
<body bgColor="#fefefe" link="#ff0000" alink="#00ff00" vlink="#0000ff">
<a href="http://www.microsoft.com">Microsoft</a><br>
<table cellpadding=2 border="1" width="75%" height="25%" bgcolor="#a0a0a0">
<tr><td bgColor="#f0f0f0">Hello</td><td>test</td></tr>
<tr><td bgColor="#f0f0f0">Hello</td><td>test</td></tr>
<tr><td bgColor="#f0f0f0">Hello</td><td>test</td></tr>
<tr><td bgColor="#f0f0f0">Hello</td><td>test</td></tr>
</table>
<form action="http://localhost/vcisapi/test" method="get">Name:
<input type="text" name="name" value="someone">
<input type="submit"></form>
</body>
</html>
```

Requirements

Header: atlhtml.h

See Also

Reference

CStreamFormatter Class

Other Resources

CHtmlGenBase Members

ATL Server Classes

CHtmlGenBase Members

Methods

a	Call this method to add an a element to the HTML stream.
abbr	Call this method to add an abbr element to the HTML stream.
abbrEnd	Call this method to close the abbr element previously added to the HTML stream.
acronym	Call this method to add an acronym element to the HTML stream.
acronymEnd	Call this method to close the acronym element previously added to the HTML stream.
address	Call this method to add an address element to the HTML stream.
addressEnd	Call this method to close the address element previously added to the HTML stream.
aEnd	Call this method to close the a element previously added to the HTML stream.
area	Call this method to add an area element to the HTML stream.
b	Call this method to add a b element to the HTML stream.
base	Call this method to add a base element to the HTML stream.
bdo	Call this method to add a bdo element to the HTML stream.
bdoEnd	Call this method to close the bdo element previously added to the HTML stream.
bEnd	Call this method to close the b element previously added to the HTML stream.
big	Call this method to add a big element to the HTML stream.
bigEnd	Call this method to close the big element previously added to the HTML stream.
blockquote	Call this method to add a blockquote element to the HTML stream.
blockquoteEnd	Call this method to close the blockquote element previously added to the HTML stream.
body	Call this method to add a body element to the HTML stream.
bodyEnd	Call this method to close the body element previously added to the HTML stream.
br	Call this method to add a br element to the HTML stream.
button	Call this method to add a button element to the HTML stream.
buttonEnd	Call this method to close the button element previously added to the HTML stream.
CHtmlGenBase	The constructor.

cite	Call this method to add a cite element to the HTML stream.
citeEnd	Call this method to close the cite element previously added to the HTML stream.
code	Call this method to add a code element to the HTML stream.
codeEnd	Call this method to close the code element previously added to the HTML stream.
col	Call this method to add a col element to the HTML stream.
colgroup	Call this method to add a colgroup element to the HTML stream.
colgroupEnd	Call this method to close the colgroup element previously added to the HTML stream.
dd	Call this method to add a dd element to the HTML stream.
ddEnd	Call this method to close the dd element previously added to the HTML stream.
del	Call this method to add a del element to the HTML stream.
delEnd	Call this method to close the del element previously added to the HTML stream.
dfn	Call this method to add a dfn element to the HTML stream.
dfnEnd	Call this method to close the dfn element previously added to the HTML stream.
div	Call this method to add a div element to the HTML stream.
divEnd	Call this method to close the div element previously added to the HTML stream.
dl	Call this method to add a dl element to the HTML stream.
dlEnd	Call this method to close the dl element previously added to the HTML stream.
dt	Call this method to add a dt element to the HTML stream.
dtEnd	Call this method to close the dt element previously added to the HTML stream.
em	Call this method to add an em element to the HTML stream.
emEnd	Call this method to close the em element previously added to the HTML stream.
fieldset	Call this method to add a fieldset element to the HTML stream.
fieldsetEnd	Call this method to close the fieldset element previously added to the HTML stream.
font	Call this method to add a font element to the HTML stream.
fontEnd	Call this method to close the font element previously added to the HTML stream.
form	Call this method to add a form element to the HTML stream.
formEnd	Call this method to close the form element previously added to the HTML stream.

GetOuter	Call this method to get a pointer to the class derived from **CHtmlGenBase**.
h	Call this method to add an h element to the HTML stream.
head	Call this method to add a head element to the HTML stream.
headEnd	Call this method to close the head element previously added to the HTML stream.
hEnd	Call this method to close the h element previously added to the HTML stream.
hr	Call this method to add an hr element to the HTML stream.
html	Call this method to add an html element to the HTML stream.
htmlEnd	Call this method to close the html element previously added to the HTML stream.
i	Call this method to add an i element to the HTML stream.
iEnd	Call this method to close the i element previously added to the HTML stream.
iframe	Call this method to add an iframe element to the HTML stream.
iframeEnd	Call this method to close the iframe element previously added to the HTML stream.
img	Call this method to add an img element to the HTML stream.
input	Call this method to add an input element to the HTML stream.
ins	Call this method to add an ins element to the HTML stream.
insEnd	Call this method to close the ins element previously added to the HTML stream.
kbd	Call this method to add a kbd element to the HTML stream.
kbdEnd	Call this method to close the kbd element previously added to the HTML stream.
label	Call this method to add a label element to the HTML stream.
labelEnd	Call this method to close the label element previously added to the HTML stream.
legend	Call this method to add a legend element to the HTML stream.
legendEnd	Call this method to close the legend element previously added to the HTML stream.
li	Call this method to add an li element to the HTML stream.
liEnd	Call this method to close the li element previously added to the HTML stream.
link	Call this method to add a link element to the HTML stream.
map	Call this method to add a map element to the HTML stream.
mapEnd	Call this method to close the map element previously added to the HTML stream.

meta	Call this method to add a meta element to the HTML stream.
noframes	Call this method to add a noframes element to the HTML stream.
noframesEnd	Call this method to close the noframes element previously added to the HTML stream.
noscript	Call this method to add a noscript element to the HTML stream.
noscriptEnd	Call this method to close the noscript element previously added to the HTML stream.
object	Call this method to add an object element to the HTML stream.
objectEnd	Call this method to close the object element previously added to the HTML stream.
ol	Call this method to add an ol element to the HTML stream.
olEnd	Call this method to close the ol element previously added to the HTML stream.
option	Call this method to add an option element to the HTML stream.
optionEnd	Call this method to close the option element previously added to the HTML stream.
p	Call this method to add a p element to the HTML stream.
param	Call this method to add a param element to the HTML stream.
pEnd	Call this method to close the p element previously added to the HTML stream.
pre	Call this method to add a pre element to the HTML stream.
preEnd	Call this method to close the pre element previously added to the HTML stream.
q	Call this method to add a q element to the HTML stream.
qEnd	Call this method to close the q element previously added to the HTML stream.
samp	Call this method to add a samp element to the HTML stream.
sampEnd	Call this method to close the samp element previously added to the HTML stream.
select	Call this method to add a select element to the HTML stream.
selectEnd	Call this method to close the select element previously added to the HTML stream.
SetScheme	Call this method to set the HTML scheme.
SetSizePercent	Call this method to set the relative size of a table.
_small	Call this method to add a small element to the HTML stream.
_smallEnd	Call this method to close the small element previously added to the HTML stream.
span	Call this method to add a span element to the HTML stream.

spanEnd	Call this method to close the span element previously added to the HTML stream.
strong	Call this method to add a strong element to the HTML stream.
strongEnd	Call this method to close the strong element previously added to the HTML stream.
sub	Call this method to add a sub element to the HTML stream.
subEnd	Call this method to close the sub element previously added to the HTML stream.
submit	Call this method to add a submit element to the HTML stream.
sup	Call this method to add a sup element to the HTML stream.
supEnd	Call this method to close the sup element previously added to the HTML stream.
table	Call this method to add a table element to the HTML stream.
tableEnd	Call this method to close the table element previously added to the HTML stream.
tbody	Call this method to add a tbody element to the HTML stream.
tbodyEnd	Call this method to close the tbody element previously added to the HTML stream.
td	Call this method to add a td element to the HTML stream.
tdEnd	Call this method to close the td element previously added to the HTML stream.
textarea	Call this method to add a textarea element to the HTML stream.
textareaEnd	Call this method to close the textarea element previously added to the HTML stream.
tfoot	Call this method to add a tfoot element to the HTML stream.
tfootEnd	Call this method to close the tfoot element previously added to the HTML stream.
th	Call this method to add a th element to the HTML stream.
thEnd	Call this method to close the th element previously added to the HTML stream.
title	Call this method to add a title element to the HTML stream.
titleEnd	Call this method to close the title element previously added to the HTML stream.
tr	Call this method to add a tr element to the HTML stream.
trEnd	Call this method to close the tr element previously added to the HTML stream.
tt	Call this method to add a tt element to the HTML stream.
ttEnd	Call this method to close the tt element previously added to the HTML stream.
u	Call this method to add a u element to the HTML stream.

uEnd	Call this method to close the u element previously added to the HTML stream.
ul	Call this method to add a ul element to the HTML stream.
ulEnd	Call this method to close the ul element previously added to the HTML stream.
var	Call this method to add a var element to the HTML stream.
varEnd	Call this method to close the var element previously added to the HTML stream.

Enums

ATL_HTML_FORM_METHOD	An enumeration of HTML form methods.

See Also

Reference

CHtmlGenBase Class

CHtmlGenBase Methods

For information about the methods in **CHtmlGenBase**, see CHtmlGenBase Members.

CHtmlGenBase::a

Call this method to add an a element to the HTML stream.

```
HRESULT a(
   LPCTSTR szHref,
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szHref

The href attribute.

szContent

The content of the element.

szAttrs

The string containing other attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</a>` tag will be generated automatically.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::aEnd
a Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::abbr

Call this method to add an abbr element to the HTML stream.

```
HRESULT abbr(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</abbr>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class

CHtmlGenBase::abbrEnd

Other Resources

CHtmlGenBase Members

CHtmlGenBase::abbrEnd

Call this method to close the abbr element previously added to the HTML stream.

```
HRESULT abbrEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class

CHtmlGenBase::abbr

Other Resources

CHtmlGenBase Members

CHtmlGenBase::acronym

Call this method to add an acronym element to the HTML stream.

```
HRESULT acronym(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</acronym>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::acronymEnd
acronym Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::acronymEnd

Call this method to close the acronym element previously added to the HTML stream.

```
HRESULT acronymEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::acronym
acronym Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::address

Call this method to add an address element to the HTML stream.

```
HRESULT address(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</address>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::addressEnd
address Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::addressEnd

Call this method to close the address element previously added to the HTML stream.

```
HRESULT addressEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::address
address Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::aEnd

Call this method to close the a element previously added to the HTML stream.

```
HRESULT aEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::a
a Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::area

Call this method to add an area element to the HTML stream.

```
HRESULT area(
   LPCTSTR szAlt,
   LPCTSTR szHref = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szAlt

The alternative text.

szHref

The destination URL or anchor point.

szAttrs

The string containing other attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class

area Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::b

Call this method to add a b element to the HTML stream.

```
HRESULT b(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</b>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::bEnd
b Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::base

Call this method to add a base element to the HTML stream.

```
HRESULT base(
   LPCTSTR szHref,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szHref

The baseline URL on which relative links will be based.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class

base Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::bdo

Call this method to add a bdo element to the HTML stream.

```
HRESULT bdo(
   LPCTSTR szDir,
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szDir

The direction ("LTR" or "RTL").

szContent

The content of the element.

szAttrs

The string containing other attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</bdo>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::bdoEnd
bdo Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::bdoEnd

Call this method to close the bdo element previously added to the HTML stream.

```
HRESULT bdoEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::bdo
bdo Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::bEnd

Call this method to close the b element previously added to the HTML stream.

```
HRESULT bEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::b
b Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::big

Call this method to add a big element to the HTML stream.

```
HRESULT big(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</big>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::bigEnd
big Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::bigEnd

Call this method to close the big element previously added to the HTML stream.

```
HRESULT bigEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::big
big Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::blockquote

Call this method to add a blockquote element to the HTML stream.

```
HRESULT blockquote(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</blockquote>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::blockquoteEnd
blockquote Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::blockquoteEnd

Call this method to close the blockquote element previously added to the HTML stream.

```
HRESULT blockquoteEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::blockquote
blockquote Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::body

Call this method to add a body element to the HTML stream.

```
HRESULT body(
   LPCTSTR szBgColor = NULL,
   LPCTSTR szBackground = NULL,
   LPCTSTR szTopMargin = NULL,
   LPCTSTR szLeftMargin = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szBgColor

The background color.

szBackground

The background.

szTopMargin

The top margin.

szLeftMargin

The left margin.

szAttrs

The string containing other attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Default formatting attributes will be added if defined in the current scheme.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::bodyEnd
body Element
CHtmlGenBase::SetScheme

Other Resources

CHtmlGenBase Members

CHtmlGenBase::bodyEnd

Call this method to close the body element previously added to the HTML stream.

```
HRESULT bodyEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::body
body Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::br

Call this method to add a br element to the HTML stream.

```
HRESULT br(
   LPCTSTR szAttrs = NULL
);
```

Parameters

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class

br Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::button

Call this method to add a button element to the HTML stream.

```
HRESULT button(
   LPCTSTR szName = NULL,
   LPCTSTR szValue = NULL,
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szName

The name of the button.

szValue

The value of the button.

szContent

The content of the element.

szAttrs

The string containing other attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</button>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::buttonEnd
button Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::buttonEnd

Call this method to close the button element previously added to the HTML stream.

```
HRESULT buttonEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::button
button Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::CHtmlGenBase

The constructor.

```
CHtmlGenBase( );
```

See Also

Reference

CHtmlGenBase Class

Other Resources

CHtmlGenBase Members

CHtmlGenBase::cite

Call this method to add a cite element to the HTML stream.

```
HRESULT cite(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</cite>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::citeEnd
cite Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::citeEnd

Call this method to close the cite element previously added to the HTML stream.

```
HRESULT citeEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::cite
cite Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::code

Call this method to add a code element to the HTML stream.

```
HRESULT code(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</code>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::codeEnd
code Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::codeEnd

Call this method to close the code element previously added to the HTML stream.

```
HRESULT codeEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::code
code Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::col

Call this method to add a col element to the HTML stream.

```
HRESULT col(
   int nSpan = 1,
   LPCTSTR szWidth = NULL,
   LPCTSTR szHeight = NULL,
   LPCTSTR szVAlign = NULL,
   LPCTSTR szHAlign = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

nSpan

The number of columns affected.

szWidth

The width of the column.

szHeight

The height of the column.

szVAlign

The vertical alignment of elements in the column.

szHAlign

The horizontal alignment of elements in the column.

szAttrs

The string containing other attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class

col Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::colgroup

Call this method to add a colgroup element to the HTML stream.

```
HRESULT colgroup(
   int nSpan = 1,
   LPCTSTR szWidth = NULL,
   LPCTSTR szHeight = NULL,
   LPCTSTR szVAlign = NULL,
   LPCTSTR szHAlign = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

nSpan

The number of columns in the group.

szWidth

The width of the column group.

szHeight

The height of the column group.

szVAlign

The vertical alignment of elements in the column group.

szHAlign

The horizontal alignment of elements in the column group.

szAttrs

The string containing other attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::colgroupEnd
colgroup Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::colgroupEnd

Call this method to close the colgroup element previously added to the HTML stream.

```
HRESULT colgroupEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::colgroup
colgroup Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::dd

Call this method to add a dd element to the HTML stream.

```
HRESULT dd(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</dd>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::ddEnd
dd Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::ddEnd

Call this method to close the dd element previously added to the HTML stream.

```
HRESULT ddEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::dd
dd Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::del

Call this method to add a del element to the HTML stream.

```
HRESULT del(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</del>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::delEnd
del Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::delEnd

Call this method to close the del element previously added to the HTML stream.

```
HRESULT delEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::del
del Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::dfn

Call this method to add a dfn element to the HTML stream.

```
HRESULT dfn(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</dfn>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::dfnEnd
dfn Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::dfnEnd

Call this method to close the dfn element previously added to the HTML stream.

```
HRESULT dfnEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::dfn
dfn Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::div

Call this method to add a div element to the HTML stream.

```
HRESULT div(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</div>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::divEnd
div Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::divEnd

Call this method to close the div element previously added to the HTML stream.

```
HRESULT divEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::div
div Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::dl

Call this method to add a dl element to the HTML stream.

```
HRESULT dl(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</dl>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::dlEnd
dl Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::dlEnd

Call this method to close the dl element previously added to the HTML stream.

```
HRESULT dlEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::dl
dl Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::dt

Call this method to add a dt element to the HTML stream.

```
HRESULT dt(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</dt>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::dtEnd
dt Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::dtEnd

Call this method to close the dt element previously added to the HTML stream.

```
HRESULT dtEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::dt
dt Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::em

Call this method to add an em element to the HTML stream.

```
HRESULT em(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</em>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::emEnd
em Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::emEnd

Call this method to close the em element previously added to the HTML stream.

```
HRESULT emEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::em
em Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::fieldset

Call this method to add a fieldset element to the HTML stream.

```
HRESULT fieldset(
   LPCTSTR szAttrs = NULL
);
```

Parameters

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::fieldsetEnd
fieldset Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::fieldsetEnd

Call this method to close the fieldset element previously added to the HTML stream.

```
HRESULT fieldsetEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::fieldset
fieldset Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::font

Call this method to add a font element to the HTML stream.

```
HRESULT font(
   COLORREF clrColor,
   LPCTSTR szAttrs = NULL
);
HRESULT font(
   LPCTSTR szFace,
   LPCTSTR szSize = NULL,
   LPCTSTR szColor = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

clrColor

The text color.

szAttrs

The string containing other attributes and their values to apply to this element.

szFace

The name of the typeface.

szSize

The size of the font.

szColor

The text color.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::fontEnd
font Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::fontEnd

Call this method to close the font element previously added to the HTML stream.

```
HRESULT fontEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::font
font Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::form

Call this method to add a form element to the HTML stream.

```
HRESULT form(
   LPCTSTR szAction,
   LPCTSTR szMethod,
   LPCTSTR szAttrs = NULL
);
HRESULT form(
   LPCTSTR szAction,
   ATL_HTML_FORM_METHOD nMethod = ATL_HTML_FORM_METHOD_GET,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szAction

The URL of the server to which the form content is sent for processing.

szMethod

The method ("get", "put", or "multipart").

szAttrs

The string containing other attributes and their values to apply to this element.

nMethod

The method. See CHtmlGenBase::ATL_HTML_FORM_METHOD.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::formEnd
form Element
CHtmlGenBase::ATL_HTML_FORM_METHOD

Other Resources

CHtmlGenBase Members

CHtmlGenBase::formEnd

Call this method to close the form element previously added to the HTML stream.

```
HRESULT formEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::form
form Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::GetOuter

Call this method to get a pointer to the class derived from **CHtmlGenBase**.

```
T* GetOuter( );
```

Return Value

Returns a pointer to the class derived from **CHtmlGenBase**.

See Also

Reference

CHtmlGenBase Class

Other Resources

CHtmlGenBase Members

CHtmlGenBase::h

Call this method to add an h element to the HTML stream.

```
HRESULT h(
   int nLevel = 1,
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

nLevel

The heading level.

szContent

The content of the element.

szAttrs

The string containing other attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</h>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::hEnd
Hn Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::head

Call this method to add a head element to the HTML stream.

```
HRESULT head(
   LPCTSTR szAttrs = NULL
);
```

Parameters

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::headEnd
head Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::headEnd

Call this method to close the head element previously added to the HTML stream.

```
HRESULT headEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::head
head Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::hEnd

Call this method to close the h element previously added to the HTML stream.

```
HRESULT hEnd(
   int nLevel = 1
);
```

Parameters

nLevel

The heading level.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class

CHtmlGenBase::h

Hn Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::hr

Call this method to add an hr element to the HTML stream.

```
HRESULT hr(
   LPCTSTR szAttrs = NULL
);
```

Parameters

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class

HR Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::html

Call this method to add an html element to the HTML stream.

```
HRESULT html(
   LPCTSTR szAttrs = NULL
);
```

Parameters

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::htmlEnd
html Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::htmlEnd

Call this method to close the html element previously added to the HTML stream.

```
HRESULT htmlEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::html
html Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::i

Call this method to add an i element to the HTML stream.

```
HRESULT i(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</i>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::iEnd
i Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::iEnd

Call this method to close the i element previously added to the HTML stream.

```
HRESULT iEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::i
i Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::iframe

Call this method to add an iframe element to the HTML stream.

```
HRESULT iframe(
   LPCTSTR szSrc = NULL,
   LPCTSTR szWidth = NULL,
   LPCTSTR szHeight = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szSrc

The URL to be loaded by the iframe.

szWidth

The width of the iframe.

szHeight

The height of the iframe.

szAttrs

The string containing other attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::iframeEnd
iframe Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::iframeEnd

Call this method to close the iframe element previously added to the HTML stream.

```
HRESULT iframeEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::iframe
iframe Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::img

Call this method to add an img element to the HTML stream.

```
HRESULT img(
   LPCTSTR szSrc,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szSrc

The URL of the image to be loaded.

szAttrs

The string containing other attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class

img Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::input

Call this method to add an input element to the HTML stream.

```
HRESULT input(
   LPCTSTR szType,
   LPCTSTR szName,
   LPCTSTR szValue,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szType

The type (for example, "button", "checkbox", and so on).

szName

The name of the input element.

szValue

The value of the input element.

szAttrs

The string containing other attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class
input Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::ins

Call this method to add an ins element to the HTML stream.

```
HRESULT ins(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</ins>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::insEnd
ins Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::insEnd

Call this method to close the ins element previously added to the HTML stream.

```
HRESULT insEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::ins
ins Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::kbd

Call this method to add a kbd element to the HTML stream.

```
HRESULT kbd(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</kbd>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::kbdEnd
kbd Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::kbdEnd

Call this method to close the kbd element previously added to the HTML stream.

```
HRESULT kbdEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::kbd
kbd Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::label

Call this method to add a label element to the HTML stream.

```
HRESULT label(
   LPCTSTR szFor = NULL,
   LPCTSTR szAccessKey = NULL,
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szFor

The ID of the control being labeled.

szAccessKey

The access key.

szContent

The content of the element.

szAttrs

The string containing other attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</label>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::labelEnd
label Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::labelEnd

Call this method to close the label element previously added to the HTML stream.

```
HRESULT labelEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::label
label Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::legend

Call this method to add a legend element to the HTML stream.

```
HRESULT legend(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</legend>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::legendEnd
legend Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::legendEnd

Call this method to close the legend element previously added to the HTML stream.

```
HRESULT legendEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::legend
legend Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::li

Call this method to add an li element to the HTML stream.

```
HRESULT li(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</li>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::liEnd
li Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::liEnd

Call this method to close the li element previously added to the HTML stream.

```
HRESULT liEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::li
li Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::link

Call this method to add a link element to the HTML stream.

```
HRESULT link(
   LPCTSTR szRel = NULL,
   LPCTSTR szHref = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szRel

The relationship between the link and its destination.

szHref

The destination URL or anchor point.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class

link Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::map

Call this method to add a map element to the HTML stream.

```
HRESULT map(
   LPCTSTR szName,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szName

The name of the element.

szAttrs

The string containing other attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::mapEnd
map Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::mapEnd

Call this method to close the map element previously added to the HTML stream.

```
HRESULT mapEnd();
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::map
map Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::meta

Call this method to add a meta element to the HTML stream.

```
HRESULT meta(
   LPCTSTR szName = NULL,
   LPCTSTR szContent = NULL,
   LPCTSTR szHttpEquiv = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szName

The name of the meta element.

szContent

The content of the element.

szHttpEquiv

The "http-equivalent" information used to bind the META tag's content to an HTTP response header.

szAttrs

The string containing other attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class

meta Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::noframes

Call this method to add a noframes element to the HTML stream.

```
HRESULT noframes(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</noframes>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::noframesEnd
noframes Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::noframesEnd

Call this method to close the noframes element previously added to the HTML stream.

```
HRESULT noframesEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::noframes
noframes Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::noscript

Call this method to add a noscript element to the HTML stream.

```
HRESULT noscript(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</noscript>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::noscriptEnd
noscript Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::noscriptEnd

Call this method to close the noscript element previously added to the HTML stream.

```
HRESULT noscriptEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::noscript
noscript Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::object

Call this method to add an object element to the HTML stream.

```
HRESULT object(
   LPCTSTR szClassId,
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
HRESULT object(
   REFCLSID rclsid,
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szClassId

The CLSID of the object.

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

rclsid

The CLSID of the object.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</object>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::objectEnd
object Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::objectEnd

Call this method to close the object element previously added to the HTML stream.

```
HRESULT objectEnd();
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::object
object Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::ol

Call this method to add an ol element to the HTML stream.

```
HRESULT ol(
   LPCTSTR szAttrs = NULL
);
```

Parameters

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::olEnd
ol Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::olEnd

Call this method to close the ol element previously added to the HTML stream.

```
HRESULT olEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::ol
ol Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::option

Call this method to add an option element to the HTML stream.

```
HRESULT option(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</option>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::optionEnd
option Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::optionEnd

Call this method to close the option element previously added to the HTML stream.

```
HRESULT optionEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::option
option Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::p

Call this method to add a p element to the HTML stream.

```
HRESULT p(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</p>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::pEnd
p Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::param

Call this method to add a param element to the HTML stream.

```
HRESULT param(
   LPCTSTR szName,
   LPCTSTR szValue,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szName

The parameter name.

szValue

The parameter value.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class

param Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::pEnd

Call this method to close the p element previously added to the HTML stream.

```
HRESULT pEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::p
p Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::pre

Call this method to add a pre element to the HTML stream.

```
HRESULT pre(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</pre>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::preEnd
pre Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::preEnd

Call this method to close the pre element previously added to the HTML stream.

```
HRESULT preEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::pre
pre Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::q

Call this method to add a q element to the HTML stream.

```
HRESULT q(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</q>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::qEnd
q Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::qEnd

Call this method to close the q element previously added to the HTML stream.

```
HRESULT qEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::q
q Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::samp

Call this method to add a samp element to the HTML stream.

```
HRESULT samp(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</samp>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::sampEnd
samp Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::sampEnd

Call this method to close the samp element previously added to the HTML stream.

```
HRESULT sampEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::samp
samp Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::select

Call this method to add a select element to the HTML stream.

```
HRESULT select(
   LPCTSTR szName,
   BOOL bMultiple = FALSE,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szName

The name of the select element.

bMultiple

TRUE if multiple items can be selected, FALSE otherwise.

szAttrs

The string containing other attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::selectEnd
SELECT Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::selectEnd

Call this method to close the select element previously added to the HTML stream.

```
HRESULT selectEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::select
SELECT Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::SetScheme

Call this method to set the scheme.

```
void SetScheme(
   HTML_SCHEME * pScheme
);
```

Parameters

pScheme

The scheme.

Remarks

The scheme contains default HTML formatting information, such as background color, link colors, and margins. For more information, see HTML_SCHEME.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class

Other Resources

CHtmlGenBase Members

CHtmlGenBase::SetSizePercent

Call this method to set the relative size of a table.

```
void SetSizePercent(
   int nWidth,
   int nHeight
);
```

Parameters

nWidth

The width (as a percentage).

nHeight

The height (as a percentage).

Remarks

Call this method before calling table.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class

Other Resources

CHtmlGenBase Members

CHtmlGenBase::_small

Call this method to add a small element to the HTML stream.

```
HRESULT _small(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Unlike the other methods that add elements to the HTML stream, this method is prefixed with an underscore because the identifier **small** is defined as a macro in rpcndr.h.

If *szContent* is non-NULL, a closing `</small>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::_smallEnd
small Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::_smallEnd

Call this method to close the small element previously added to the HTML stream.

```
HRESULT _smallEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Unlike the other methods that add elements to the HTML stream, this method is prefixed with an underscore to match CHtmlGenBase::_small because the identifier **small** is defined as a macro in rpcndr.h.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::_small
small Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::span

Call this method to add a span element to the HTML stream.

```
HRESULT span(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</span>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::spanEnd
span Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::spanEnd

Call this method to close the span element previously added to the HTML stream.

```
HRESULT spanEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::span
span Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::strong

Call this method to add a strong element to the HTML stream.

```
HRESULT strong(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</strong>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::strongEnd
strong Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::strongEnd

Call this method to close the strong element previously added to the HTML stream.

```
HRESULT strongEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::strong
strong Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::sub

Call this method to add a sub element to the HTML stream.

```
HRESULT sub(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</sub>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::subEnd
sub Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::subEnd

Call this method to close the sub element previously added to the HTML stream.

```
HRESULT subEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::sub
sub Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::submit

Call this method to add a submit element to the HTML stream.

```
HRESULT submit(
   LPCTSTR szValue = NULL,
   LPCTSTR szName = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szValue

The value of the element.

szName

The name of the element.

szAttrs

The string containing other attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

A "submit element" is actually an input element whose type is "submit."

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class
input Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::sup

Call this method to add a sup element to the HTML stream.

```
HRESULT sup(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</sup>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::supEnd
sup Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::supEnd

Call this method to close the sup element previously added to the HTML stream.

```
HRESULT supEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::sup
sup Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::table

Call this method to add a table element to the HTML stream.

```
HRESULT table(
   int nBorderWidth = 0,
   LPCTSTR szAttrs = NULL
);
```

Parameters

nBorderWidth

The border width.

szAttrs

The string containing other attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If a table size has been set, width and height attributes will be added automatically. If strTableBgColor has been set in the current scheme, a bgcolor attribute will be added.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::tableEnd
table Element
CHtmlGenBase::SetSizePercent
CHtmlGenBase::SetScheme

Other Resources

CHtmlGenBase Members

CHtmlGenBase::tableEnd

Call this method to close the table element previously added to the HTML stream.

```
HRESULT tableEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::table
table Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::tbody

Call this method to add a tbody element to the HTML stream.

```
HRESULT tbody(
   LPCTSTR szAttrs = NULL
);
```

Parameters

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::tbodyEnd
tbody Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::tbodyEnd

Call this method to close the tbody element previously added to the HTML stream.

```
HRESULT tbodyEnd();
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::tbody
tbody Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::td

Call this method to add a td element to the HTML stream.

```
HRESULT td(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If strTdBgColor has been set in the current scheme, a bgcolor attribute will be added automatically.

If *szContent* is non-NULL, a closing `</td>` tag will be generated automatically.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::tdEnd
td Element
CHtmlGenBase::SetScheme

Other Resources

CHtmlGenBase Members

CHtmlGenBase::tdEnd

Call this method to close the td element previously added to the HTML stream.

```
HRESULT tdEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::td
td Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::textarea

Call this method to add a textarea element to the HTML stream.

```
HRESULT textarea(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</textarea>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::textareaEnd
textarea Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::textareaEnd

Call this method to close the textarea element previously added to the HTML stream.

```
HRESULT textareaEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::textarea
textarea Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::tfoot

Call this method to add a tfoot element to the HTML stream.

```
HRESULT tfoot(
   LPCTSTR szAttrs = NULL
);
```

Parameters

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::tfootEnd
tfoot Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::tfootEnd

Call this method to close the tfoot element previously added to the HTML stream.

```
HRESULT tfootEnd();
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::tfoot
tfoot Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::th

Call this method to add a th element to the HTML stream.

```
HRESULT th(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</th>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::thEnd
th Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::thEnd

Call this method to close the th element previously added to the HTML stream.

```
HRESULT thEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::th
th Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::title

Call this method to add a title element to the HTML stream.

```
HRESULT title(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</title>` tag will be generated automatically.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::titleEnd
title Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::titleEnd

Call this method to close the title element previously added to the HTML stream.

```
HRESULT titleEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::title
title Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::tr

Call this method to add a tr element to the HTML stream.

```
HRESULT tr(
   LPCTSTR szAttrs = NULL
);
```

Parameters

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If a background color has been set with SetScheme, a bgcolor attribute will be added automatically.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::trEnd
tr Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::trEnd

Call this method to close the tr element previously added to the HTML stream.

```
HRESULT trEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::tr
tr Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::tt

Call this method to add a tt element to the HTML stream.

```
HRESULT tt(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</tt>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::ttEnd
tt Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::ttEnd

Call this method to close the tt element previously added to the HTML stream.

```
HRESULT ttEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::tt
tt Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::u

Call this method to add a u element to the HTML stream.

```
HRESULT u(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</u>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::uEnd
u Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::uEnd

Call this method to close the u element previously added to the HTML stream.

```
HRESULT uEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::u
u Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::ul

Call this method to add a ul element to the HTML stream.

```
HRESULT ul(
   LPCTSTR szAttrs = NULL
);
```

Parameters

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::ulEnd
ul Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::ulEnd

Call this method to close the ul element previously added to the HTML stream.

```
HRESULT ulEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::ul
ul Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::var

Call this method to add a var element to the HTML stream.

```
HRESULT var(
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

szContent

The content of the element.

szAttrs

The string containing attributes and their values to apply to this element.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is non-NULL, a closing `</var>` tag will be generated automatically.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::varEnd
var Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase::varEnd

Call this method to close the var element previously added to the HTML stream.

```
HRESULT varEnd( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlGenBase Class
CHtmlGenBase::var
var Element

Other Resources

CHtmlGenBase Members

CHtmlGenBase Enums

For information about the enums in **CHtmlGenBase**, see CHtmlGenBase Members.

CHtmlGenBase::ATL_HTML_FORM_METHOD

An enumeration of HTML form methods.

```
enum ATL_HTML_FORM_METHOD
{
   ATL_HTML_FORM_METHOD_NONE=-1,
   ATL_HTML_FORM_METHOD_GET,
   ATL_HTML_FORM_METHOD_POST,
   ATL_HTML_FORM_METHOD_MULTIPART
};
```

Remarks

This enum is used in CHtmlGenBase::form to specify the form method.

See Also

Reference

CHtmlGenBase Class

Other Resources

CHtmlGenBase Members

CHtmlStencil Class

This class is used to parse a server response file and generate a text stream by replacing elements found within the server response file with output generated by an object implementing the ITagReplacer interface.

```
class CHtmlStencil :
   public CStencil
```

Remarks

CStencil and **CHtmlStencil** are used in ATL Server web applications for handling server response files. **CHtmlStencil** provides the code for parsing and rendering the following tags in a server response file:

- Comment Tag
- Include Tag
- Subhandler Tag

If you are using wizard-generated ATL Server projects and find that the server response file syntax is adequate for your needs, you might not need to use these classes directly.

However, if you wish to load and manipulate server response files directly or extend the SRF syntax for your own needs, you will probably want to use these classes directly.

Requirements

Header: atlstencil.h

See Also
Reference
CStencil Class
Other Resources
CHtmlStencil Members

CHtmlStencil Members

Methods

CHtmlStencil	The constructor.
GetBaseDir	Call this method to get the base directory of the stencil.
GetExtraHandlers	Call this method to get a pointer to the map of subhandlers found in the stencil.
Initialize	Call this method to initialize the stencil before calling CStencil::Parse.
ParseInclude	Call this method to parse an include tag.
ParseSubhandler	Call this method to parse a subhandler tag.
ParseToken	Implementation of CStencil::ParseToken.
RenderToken	Implementation of CStencil::RenderToken.
SetBaseDirFromFile	Call this method to set the base directory of the stencil based on a file name.

Data Members

m_arrExtraHandlers	A collection of subhandler tags.
m_spDllCache	The DLL cache returned by the service provider used to initialize this object.
m_spExtension	The ISAPI extension returned by the service provider used to initialize this object.
m_spServiceProvider	The service provider used to initialize this object.
m_spStencilCache	The stencil cache returned by the service provider used to initialize this object.
m_szBaseDir	The base directory of the stencil.

Typedefs

mapType	A typedef for the map used to hold information about the subhandler tags found in the stencil.
baseType	A typedef for the base class of **CHtmlStencil**.

See Also

Reference

CHtmlStencil Class

Other Resources

CStencil Members

CHtmlStencil Methods

For information about the methods in **CHtmlStencil**, see CHtmlStencil Members

CHtmlStencil::CHtmlStencil

The constructor.

```
CHtmlStencil(
   IAtlMemMgr * pMemMgr = NULL
) throw( );
```

Parameters

pMemMgr

The memory manager used for allocating and releasing memory used by this object.

Remarks

Passes *pMemMgr* to the CStencil constructor.

See Also

Reference

CHtmlStencil Class

CStencil::CStencil

Other Resources

CHtmlStencil Members

CHtmlStencil::GetBaseDir

Call this method to get the base directory of the stencil.

```
LPCSTR GetBaseDir( );
```

Return Value

Returns the base directory of the stencil.

Remarks

The base directory of the stencil is used to locate files included by an include tag when only a relative path is given.

The base directory is set by calling CHtmlStencil::SetBaseDirFromFile.

See Also

Reference

CHtmlStencil Class
CHtmlStencil::SetBaseDirFromFile
CHtmlTagReplacer::LoadStencil
CHtmlTagReplacer::LoadStencilResource

Other Resources

CHtmlStencil Members

CHtmlStencil::GetExtraHandlers

Call this method to get a pointer to the map of subhandlers found in the stencil.

```
mapType* GetExtraHandlers( ) throw( );
```

Return Value

Returns a pointer to the map of subhandlers found in the stencil.

Remarks

Each subhandler tag found in the stencil is represented by an entry in the map returned by this method. The key into the map is the *handler_alias*, and the value is a string pair holding the *dll_path* and *request_handler*.

See Also

Reference

CHtmlStencil Class
CHtmlStencil::mapType
Subhandler Tag

Other Resources

CHtmlStencil Members

CHtmlStencil::Initialize

Call this method to initialize the stencil before calling CStencil::Parse.

```
void Initialize(
   IServiceProvider * pProvider
) throw(...);
```

Parameters

pProvider

The service provider.

Remarks

This method stores the service provider in m_spServiceProvider. It queries the service provider for IDllCache and IIsapiExtension, which it stores in m_spDllCache and CHtmlStencil::m_spExtension respectively.

During rendering of include tags, the service provider will be queried for IStencilCache.

See Also

Reference

CHtmlStencil Class

Other Resources

CHtmlStencil Members

CHtmlStencil::ParseInclude

Call this method to parse an include tag.

```
DWORD ParseInclude(
   LPCSTR szTokenStart,
   LPCSTR szTokenEnd
) throw(...);
```

Parameters

szTokenStart

The start of the token (after the braces and whitespace).

szTokenEnd

The end of the token.

Return Value

Returns the index of the token added to the collection managed by this object, or **STENCIL_INVALIDINDEX** if an error occurs.

See Also

Reference

CHtmlStencil Class

Other Resources

CHtmlStencil Members

CHtmlStencil::ParseSubhandler

Call this method to parse a subhandler tag.

```
PARSE_TOKEN_RESULT ParseSubhandler(
   LPCSTR szTokenStart,
   LPCSTR szTokenEnd
) throw( );
```

Parameters

szTokenStart

The start of the token (after the braces and whitespace).

szTokenEnd

The end of the token.

Return Value

See CStencil::PARSE_TOKEN_RESULT for possible values and their meanings.

Remarks

Adds an entry to the array of handlers available through the map returned from CHtmlStencil::GetExtraHandlers.

See Also

Reference

CHtmlStencil Class

Other Resources

CHtmlStencil Members

CHtmlStencil::ParseToken

Implementation of CStencil::ParseToken.

```
virtual PARSE_TOKEN_RESULT ParseToken(
   LPCSTR szTokenStart,
   LPCSTR szTokenEnd,
   DWORD * pBlockStack,
   DWORD * pdwTop
);
```

Remarks

The implementation in this class handles the following tags directly:

- Comment Tag
- Include Tag
- Subhandler Tag

For tags that it does not recognize, this method delegates to CStencil::ParseToken.

Note that successfully parsed comment tags and subhandler tags do not cause a stencil token to be added to the collection of tokens managed by **CStencil**. Comment tags are discarded. Subhandler tags cause elements to be added to the map available through CHtmlStencil::GetExtraHandlers.

See Also

Reference

CHtmlStencil Class
CStencil::ParseToken
Comment Tag
Include Tag
Subhandler Tag

Other Resources

CHtmlStencil Members

CHtmlStencil::RenderToken

Implementation of CStencil::RenderToken.

```
DWORD RenderToken(
   DWORD dwIndex,
   ITagReplacer* pReplacer,
   IWriteStream * pWriteStream,
   HTTP_CODE * phcErrorCode,
   CStencilState* pState = NULL
) const throw(...);
```

Remarks

If you need to provide custom actions for your own stencil tokens, you can override this method and take the necessary steps to produce output or configure the environment in which stencil rendering takes place.

The **CHtmlStencil** implementation of this method handles the following stencil tokens:

- **STENCIL_STENCILINCLUDE**
- **STENCIL_STATICINCLUDE**

Tokens of any other type are delegated to CStencil::RenderToken.

During rendering of **STENCIL_STENCILINCLUDE** tokens, the service provider passed to CHtmlStencil::Initialize is asked to provide IHttpServerContext and IHttpRequestLookup.

If the included handler returns **HTTP_SUCCESS_NO_CACHE**, the IHttpServerContext interface obtained from the service provider is queried for IPageCacheControl and caching is disabled. This means that included handlers can disable caching for the entire response.

See Also

Reference

CHtmlStencil Class

Other Resources

CHtmlStencil Members

CHtmlStencil::SetBaseDirFromFile

Call this method to set the base directory of the stencil based on a file name.

```
BOOL SetBaseDirFromFile(
   LPCSTR szBaseDir
);
```

Parameters

szBaseDir

The name of a file whose parent directory will be used as the base directory of this stencil.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The base directory of the stencil is used to locate files included by an include tag when only a relative path is given.

See CHtmlTagReplacer::LoadStencil and CHtmlTagReplacer::LoadStencilResource for information on how this value is set in wizard-generated ATL Server projects.

See Also

Reference

CHtmlStencil Class

CHtmlStencil::GetBaseDir

Other Resources

CHtmlStencil Members

CHtmlStencil Data Members

For information about the data members in **CHtmlStencil**, see CHtmlStencil Members

CHtmlStencil::m_arrExtraHandlers

A collection of subhandler tags.

```
CAtlMap<
   CStringA, CStringPair, CStringElementTraits<CStringA>,
   CStringPairElementTraits
> m_arrExtraHandlers;
```

Remarks

Each subhandler tag found in the stencil is represented by an entry in this map. The key into the map is the *handler_alias*, and the value is a string pair holding the *dll_path* and *request_handler*.

See Also

Reference

CHtmlStencil Class

CHtmlStencil::GetExtraHandlers

Other Resources

CHtmlStencil Members

CHtmlStencil::m_spDllCache

The DLL cache returned by the service provider used to initialize this object.

```
CComPtr<IDllCache> m_spDllCache;
```

See Also

Reference

CHtmlStencil Class

CHtmlStencil::Initialize

Other Resources

CHtmlStencil Members

CHtmlStencil::m_spExtension

The ISAPI extension returned by the service provider used to initialize this object.

```
CComPtr<IIsapiExtension> m_spExtension;
```

See Also

Reference

CHtmlStencil Class

CHtmlStencil::Initialize

Other Resources

CHtmlStencil Members

CHtmlStencil::m_spServiceProvider

The service provider used to initialize this object.

```
CComPtr<IServiceProvider> m_spServiceProvider;
```

See Also

Reference

CHtmlStencil Class

CHtmlStencil::Initialize

Other Resources

CHtmlStencil Members

CHtmlStencil::m_spStencilCache

The stencil cache returned by the service provider used to initialize this object.

```
CComPtr<IStencilCache> m_spStencilCache;
```

Remarks

This member is not obtained from the service provider until it is necessary to render an include tag.

See Also

Reference

CHtmlStencil Class

CHtmlStencil::Initialize

Other Resources

CHtmlStencil Members

CHtmlStencil::m_szBaseDir

The base directory of the stencil.

```
CHAR m_szBaseDir[MAX_PATH];
```

Remarks

The base directory of the stencil is used to locate files included by an include tag when only a relative path is given.

See Also

Reference

CHtmlStencil Class
CHtmlStencil::GetBaseDir
CHtmlStencil::SetBaseDirFromFile

Other Resources

CHtmlStencil Members

CHtmlStencil Typedefs

For information about the typedefs in **CHtmlStencil**, see CHtmlStencil Members

CHtmlStencil::mapType

A typedef for the map used to hold information about the subhandler tags found in the stencil.

```
typedef CAtlMap<
   CStringA, CStringPair, CStringElementTraits<CStringA>,
   CStringPairElementTraits
> mapType;
```

Remarks

Each subhandler tag found in the stencil is represented by an entry in the map described by this typedef. The key into the map is the *handler_alias*, and the value is a string pair holding the *dll_path* and *request_handler*.

See Also

Reference

CHtmlStencil Class

CHtmlStencil::GetExtraHandlers

Other Resources

CHtmlStencil Members

CHtmlStencil::baseType

A typedef for the base class of **CHtmlStencil**.

```
typedef CStencil baseType;
```

See Also

Reference

CHtmlStencil Class

Other Resources

CHtmlStencil Members

CHtmlTagReplacer Class

This class acts as the link between CStencil-derived classes and CRequestHandlerT and provides stencil caching.

```
template <
   class THandler,
   class StencilType = CHtmlStencil
>
class CHtmlTagReplacer :
   public ITagReplacerImpl< THandler >
```

Parameters

THandler

A class derived from **CHtmlTagReplacer**.

StencilType

The type of stencil handled by this class. This class is expected to supply the following members:

- mapType typedef
- GetExtraHandlers
- SetBaseDirFromFile
- Initialize
- SetCacheItem
- SetErrorResource
- LoadFromResource
- LoadFromFile
- ParseReplacements
- FinishParseReplacements

It is also expected to be able to be statically cast to IMemoryCacheClient

Remarks

This class manages CStencil-derived objects used for handling HTTP requests. This class will retrieve **CStencil**-derived objects from the stencil cache, store **CStencil**-derived objects in the stencil cache, and allocate and initialize **CStencil**-derived objects on a per request basis. Typically, this classed is passed as a template parameter to CRequestHandlerT where it is used as a base class.

Requirements

Header: atlstencil.h

See Also

Reference

ITagReplacerImpl Class

Other Resources

CHtmlTagReplacer Members

CHtmlTagReplacer Members

Methods

CHtmlTagReplacer	The constructor.
~CHtmlTagReplacer	The destructor.
GetHandlerOffset	Override of ITagReplacerImpl::GetHandlerOffset.
GetReplacementObject	Override of ITagReplacerImpl::GetReplacementObject.
Initialize	Call this method to initialize the object.
LoadStencil	Call this method to load a stencil from a file.
LoadStencilResource	Call this method to load a stencil from a resource.
RenderStencil	Call this method to render the currently loaded stencil.

See Also

Reference

CHtmlTagReplacer Class

Other Resources

ITagReplacerImpl Members

CHtmlTagReplacer Methods

For information about the methods in **CHtmlTagReplacer**, see CHtmlTagReplacer Members

CHtmlTagReplacer::CHtmlTagReplacer

The constructor.

```
CHtmlTagReplacer( ) throw( );
```

Remarks

Initializes internal data members.

See Also

Reference

CHtmlTagReplacer Class

Other Resources

CHtmlTagReplacer Members

CHtmlTagReplacer::~CHtmlTagReplacer

The destructor.

```
~CHtmlTagReplacer( ) throw( );
```

Remarks

An assertion error will occur if resources have not been properly cleaned up before the constructor is executed.

See Also

Reference

CHtmlTagReplacer Class

Other Resources

CHtmlTagReplacer Members

CHtmlTagReplacer::GetHandlerOffset

Override of ITagReplacerImpl::GetHandlerOffset.

```
HTTP_CODE GetHandlerOffset(
   LPCSTR szHandlerName,
   DWORD* pdwOffset
) throw( );
```

Remarks

Implements this method using information parsed from the stencil loaded by LoadStencil or LoadStencilResource.

See Also

Reference

CHtmlTagReplacer Class

Other Resources

CHtmlTagReplacer Members

CHtmlTagReplacer::GetReplacementObject

Override of ITagReplacerImpl::GetReplacementObject.

```
HTTP_CODE GetReplacementObject(
   DWORD dwObjOffset,
   ITagReplacer ** ppReplacer
) throw( );
```

Remarks

Implements this method using information parsed from the stencil loaded by LoadStencil or LoadStencilResource.

See Also

Reference

CHtmlTagReplacer Class

Other Resources

CHtmlTagReplacer Members

CHtmlTagReplacer::Initialize

Call this method to initialize the object.

```
HTTP_CODE Initialize(
   AtlServerRequest * pRequestInfo,
   IHttpServerContext * pSafeSrvCtx = NULL
) throw(...);
```

Parameters

pRequestInfo

The information about the current request.

pSafeSrvCtx

The server context.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

Calls *THandler*::GetContext for IServiceProvider then requests the IStencilCache interface.

See Also

Reference

CHtmlTagReplacer Class

IHttpServerContext Interface

Other Resources

CHtmlTagReplacer Members

AtlServerRequest Members

CHtmlTagReplacer::LoadStencil

Call this method to load a stencil from a file.

```
HTTP_CODE LoadStencil(
   LPCSTR szFileName,
   IHttpRequestLookup * pLookup = NULL
) throw(...);
```

Parameters

szFileName

The filename of the stencil to load.

pLookup

If the stencil being loaded is used from a subhandler tag or include tag, *pLookup* provides access to the file, form variable, and query parameter collections associated with the HTTP request. Otherwise, this parameter is NULL.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

Checks whether the stencil is in the cache. If it is not, this method loads the stencil, ensures that it's parsed successfully, adds it to the cache, and makes its contents available through GetHandlerOffset and GetReplacementObject.

See Also

Reference

CHtmlTagReplacer Class

IHttpRequestLookup Interface

Other Resources

CHtmlTagReplacer Members

CHtmlTagReplacer::LoadStencilResource

Call this method to load a stencil from a resource.

```
HTTP_CODE LoadStencilResource(
   HINSTANCE hInstResource,
   UINT nID,
   LPCSTR szResourceType = NULL
) throw(...);
HTTP_CODE LoadStencilResource(
   HINSTANCE hInstResource,
   LPCSTR szResourceID,
   LPCSTR szResourceType = NULL,
   LPCSTR szStencilName = NULL
) throw(...);
```

Parameters

hInstResource

The handle to the module whose executable file contains the resource.

nID, szResourceID

The resource identifier.

szResourceType

The type of resource. If not specified, the stencil is assumed to be in an RT_HTML resource.

szStencilName

The name used to add or retrieve the stencil from the cache. If no name is given, the resource ID is used to add or retrieve the stencil from the cache.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

Checks whether the stencil is in the cache. If it is not, this method loads the stencil, ensures that it's parsed successfully, adds it to the cache, and makes its contents available through GetHandlerOffset and GetReplacementObject.

See Also

Reference

CHtmlTagReplacer Class

Other Resources

CHtmlTagReplacer Members

CHtmlTagReplacer::RenderStencil

Call this method to render the currently loaded stencil.

```
HTTP_CODE RenderStencil(
   IWriteStream* pStream,
   CStencilState* pState = NULL
) throw(...);
```

Parameters

pStream

The stream to which the response should be written.

pState

A pointer to state information to be used by the stencil.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

CHtmlTagReplacer Class
CStencilState Structure
IWriteStream Interface

Other Resources

CHtmlTagReplacer Members

CHttpMap Class

This class is a simple map class that allows the contents of the map to be shared.

```
template <
   typename K,
   typename V,
   typename KTraits = CElementTraits< K >,
   typename VTraits = CElementTraits< V >
>
class CHttpMap
```

Parameters

K

The key element type.

V

The value element type.

KTraits

The code used to copy or move key elements.

VTraits

The code used to copy or move value elements.

Remarks

This class provides methods for manipulating the map similar to those provided by CAtlMap, CRBMap, and CRBMultiMap. **CHttpMap** also adds a boolean property, represented by the CHttpMap::IsShared and CHttpMap::SetShared methods, that tells whether or not the values in this map are shared.

See ATL_HTTP_PARAM_MULTIMAP for details on how this macro affects the behavior of **CHttpMap**.

Requirements

Header: atlisapi.h

Example

See the following samples:

- Cookies Sample
- Mailer Sample
- RegExp Sample
- ShowFiles Sample
- ShowForm Sample
- ShowImage Sample

See Also

Other Resources

CHttpMap Members

ATL Server Classes

CHttpMap Members

Methods

CHttpMap	The constructor.
~CHttpMap	The destructor.
GetAt	Call this method to get the element at a given position in the map.
GetCount	Call this method to get the number of elements in the map.
GetKeyAt	Call this method to get the key from a given position in the map.
GetNext	Call this method to obtain a pointer to an element stored in the map and advance the position to the next element.
GetNextAssoc	Call this method to get the key and value of an element stored in the map and advance the position to the next element.
GetNextKey	Call this method to get the key of an element stored in the map and advance the position to the next element.
GetNextValue	Call this method to get the value of an element stored in the map and advance the position to the next element.
GetStartPosition	Call this method to get the position of the first element in the map.
GetValueAt	Call this method to get the value from a given position in the map.
IsEmpty	Call this method to determine whether or not the map is empty.
IsShared	Call this method to determine whether or not the map is shared.
Lookup	Call this method to lookup elements in the map with a given key.
RemoveAll	Call this method to remove all the elements from the map.
RemoveKey	Call this method to remove an element from the map.
SetAt	Call this method to add an element to the map.
SetShared	Call this method to specify whether or not the elements in the map are shared.

Typedefs

KINARGTYPE	The type used when a key is passed in as an argument to a method.
KOUTARGTYPE	The type used when a key is passed out as an argument to a method.
VINARGTYPE	The type used when a value is passed in as an argument to a method.
VOUTARGTYPE	The type used when a value is passed out as an argument to a method.
CPair	A class containing the key and value of an element in the map.

See Also

Reference

CHttpMap Class

CHttpMap Methods

For information about the methods in **CHttpMap**, see CHttpMap Members.

CHttpMap::CHttpMap

The constructor.

```
CHttpMap( ) throw( );
```

Remarks

Sets the shared property to false.

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::~CHttpMap

The destructor.

```
virtual ~CHttpMap( );
```

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::GetAt

Call this method to get the element at a given position in the map.

```
CPair* GetAt(
   POSITION pos
) throw( );
const CPair* GetAt(
   POSITION pos
) const throw( );
void GetAt(
   POSITION pos,
   KOUTARGTYPE key,
   VOUTARGTYPE value
) const throw(...);
```

Parameters

pos

The position of the item to retrieve.

key

The variable that receives the key.

value

The variable that receives the value.

Return Value

The first two overloads return a pointer to a CPair describing the requested item.

Remarks

A position can be obtained by a previous call to CHttpMap::GetStartPosition or any of the following methods:

- CHttpMap::GetNext
- CHttpMap::GetNextAssoc
- CHttpMap::GetNextKey
- CHttpMap::GetNextValue
- CHttpMap::SetAt

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::GetCount

Call this method to get the number of elements in the map.

```
inline size_t GetCount( ) const throw( );
```

Return Value

Returns the number of elements in the map.

Example

- Cookies Sample
- RegExp Sample
- ShowForm Sample
- ShowImage Sample

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::GetKeyAt

Call this method to get the key from a given position in the map.

```
const K& GetKeyAt(
   POSITION pos
) const throw( );
```

Parameters

pos

The position of the element whose key is to be returned.

Return Value

Returns the key stored at position *pos* in the tree.

Remarks

A position can be obtained by a previous call to CHttpMap::GetStartPosition or any of the following methods:

- CHttpMap::GetNext
- CHttpMap::GetNextAssoc
- CHttpMap::GetNextKey
- CHttpMap::GetNextValue
- CHttpMap::SetAt

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::GetNext

Call this method to obtain a pointer to an element stored in the map and advance the position to the next element.

```
CPair* GetNext(
   POSITION& pos
) throw( );
const CPair* GetNext(
   POSITION& pos
) const throw( );
```

Parameters

pos

On entry, the position of the element to be returned. On exit, the position of the next element, or NULL if there are no more elements.

Return Value

Returns a pointer to the element whose position is passed into the method.

Remarks

A position can be obtained by a previous call to CHttpMap::GetStartPosition or any of the following methods:

- CHttpMap::GetNext
- CHttpMap::GetNextAssoc
- CHttpMap::GetNextKey
- CHttpMap::GetNextValue
- CHttpMap::SetAt

Example

See the Mailer Sample.

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::GetNextAssoc

Call this method to get the key and value of an element stored in the map and advance the position to the next element.

```
void GetNextAssoc(
   POSITION& pos,
   KOUTARGTYPE key,
   VOUTARGTYPE value
) const throw(...);
```

Parameters

pos

On entry, the position of the element whose key and value are to be returned. On exit, the position of the next element, or NULL if there are no more elements.

key

The variable that receives the key.

value

The variable that receives the value.

Remarks

A position can be obtained by a previous call to CHttpMap::GetStartPosition or any of the following methods:

- CHttpMap::GetNext
- CHttpMap::GetNextAssoc
- CHttpMap::GetNextKey
- CHttpMap::GetNextValue
- CHttpMap::SetAt

Example

See the ShowForm Sample.

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::GetNextKey

Call this method to get the key of an element stored in the map and advance the position to the next element.

```
const K& GetNextKey(
   POSITION& pos
) const throw( );
```

Parameters

pos

On entry, the position of the element whose key is to be returned. On exit, the position of the next element, or NULL if there are no more elements.

Return Value

Returns the key of the element whose position is passed into the method.

Remarks

A position can be obtained by a previous call to CHttpMap::GetStartPosition or any of the following methods:

- CHttpMap::GetNext
- CHttpMap::GetNextAssoc
- CHttpMap::GetNextKey
- CHttpMap::GetNextValue
- CHttpMap::SetAt

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::GetNextValue

Call this method to get the value of an element stored in the map and advance the position to the next element.

```
V& GetNextValue(
   POSITION& pos
) throw( );
const V& GetNextValue(
   POSITION& pos
) const throw( );
```

Parameters

pos

On entry, the position of the element whose value is to be returned. On exit, the position of the next element, or NULL if there are no more elements.

Return Value

Returns the value of the element whose position is passed into the method.

Remarks

A position can be obtained by a previous call to CHttpMap::GetStartPosition or any of the following methods:

- CHttpMap::GetNext
- CHttpMap::GetNextAssoc
- CHttpMap::GetNextKey
- CHttpMap::GetNextValue
- CHttpMap::SetAt

Example

See the following samples:

- Cookies Sample
- RegExp Sample
- ShowFiles Sample
- ShowImage Sample

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::GetStartPosition

Call this method to get the position of the first element in the map.

```
inline POSITION GetStartPosition( ) const throw( );
```

Return Value

Returns the position of the first element in the map, or NULL if there are no elements in the map.

Example

See the following samples:

- Cookies Sample
- Mailer Sample
- RegExp Sample
- ShowForm Sample
- ShowImage Sample

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::GetValueAt

Call this method to get the value from a given position in the map.

```
V& GetValueAt(
   POSITION pos
) throw( );
const V& GetValueAt(
   POSITION pos
) const throw( );
```

Parameters

pos

The position of the element whose value is to be returned.

Return Value

Returns the value stored at position *pos* in the tree.

Remarks

A position can be obtained by a previous call to CHttpMap::GetStartPosition or any of the following methods:

- CHttpMap::GetNext
- CHttpMap::GetNextAssoc
- CHttpMap::GetNextKey
- CHttpMap::GetNextValue
- CHttpMap::SetAt

Example

See the Mailer Sample.

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::IsEmpty

Call this method to determine whether or not the map is empty.

```
inline bool IsEmpty( ) const throw( );
```

Return Value

Returns true if the map is empty, false otherwise.

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::IsShared

Call this method to determine whether or not the map is shared.

```
inline bool IsShared( ) const throw( );
```

Return Value

Returns true if the elements in the map are shared, false otherwise.

Remarks

Call CHttpMap::SetShared to specify whether the contents of the map are shared.

See Also

Reference

CHttpMap Class

CHttpMap::SetShared

Other Resources

CHttpMap Members

CHttpMap::Lookup

Call this method to look up an element in the map with a given key.

```
CPair* Lookup(
   KINARGTYPE key
) throw( );
const CPair* Lookup(
   KINARGTYPE key
) const throw( );
bool Lookup(
   KINARGTYPE key,
   VOUTARGTYPE value
) const throw( );
```

Parameters

key

The key of the element to be retrieved.

value

The variable that receives the value.

Return Value

The first two overloads return a pointer to the requested element, or NULL if the element cannot be found. The final overload returns **true** if the element is present, **false** otherwise.

Remarks

If ATL_HTTP_PARAM_MULTIMAP is defined, **Lookup** returns the first item with a matching key. (Otherwise the map can hold at most one item with a matching key.)

Example

See the ShowImage Sample.

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::RemoveAll

Call this method to remove all the elements from the map.

```
virtual void RemoveAll( );
```

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::RemoveKey

Call this method to remove an element from the map.

```
bool RemoveKey(
   KINARGTYPE key
) throw( );
```

Parameters

key

The key of the element to be removed.

Return Value

Returns **true** if the key is found and removed, **false** on failure.

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::SetAt

Call this method to add an element to the map.

```
POSITION SetAt(
   KINARGTYPE key,
   VINARGTYPE value
) throw(...);
```

Parameters

key

The key of the element to be added.

value

The value of the element to be added.

Return Value

Returns the position of the newly added element or NULL if the element could not be added.

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::SetShared

Call this method to specify whether or not the elements in the map are shared.

```
inline void SetShared(
   bool bShared
) throw( );
```

Parameters

bShared

Pass true if the elements in the map are shared, false otherwise.

Remarks

Call CHttpMap::IsShared to find out whether the contents of the map are shared.

See Also

Reference

CHttpMap Class

CHttpMap::IsShared

Other Resources

CHttpMap Members

CHttpMap Typedefs

For information about the typedefs in **CHttpMap**, see CHttpMap Members

CHttpMap::KINARGTYPE

The type used when a key is passed in as an argument to a method.

```
typedef KTraits::INARGTYPE KINARGTYPE;
```

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::KOUTARGTYPE

The type used when a key is passed out as an argument to a method.

```
typedef KTraits::OUTARGTYPE KOUTARGTYPE;
```

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::VINARGTYPE

The type used when a value is passed in as an argument to a method.

```
typedef VTraits::INARGTYPE VINARGTYPE;
```

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::VOUTARGTYPE

The type used when a value is passed out as an argument to a method.

```
typedef VTraits::OUTARGTYPE VOUTARGTYPE;
```

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpMap::CPair

A class containing the key and value of an element in the map.

```
typedef MAPTYPE::CPair CPair;
```

Remarks

MAPTYPE is an implementation detail, but you can assume that **CPair** has the following members:

- **const** *K* **m_key;**
- *V* **m_value;**

See Also

Reference

CHttpMap Class

Other Resources

CHttpMap Members

CHttpPtrMap Class

This class represents a map of pointers.

```
template <
   typename K,
   typename V,
   typename KTraits = CElementTraits< K >,
   typename VTraits = CElementTraits< V >
>
class CHttpPtrMap :
   public CHttpMap< K, V, KTraits, VTraits >
```

Parameters

K

The key element type.

V

The value element type.

KTraits

The code used to copy or move key elements.

VTraits

The code used to copy or move value elements.

Remarks

This class is a wrapper for CHttpMap that assumes the contained values are pointers to objects that should be deleted when CHttpPtrMap::RemoveAll is called or when the object destructs.

Requirements

Header: atlisapi.h

See Also

Other Resources

CHttpPtrMap Members

ATL Server Classes

CHttpPtrMap Members

Methods

CHttpPtrMap::~CHttpPtrMap	The destructor.
CHttpPtrMap::RemoveAll	Call this method to remove all of the elements from the map.

See Also

Reference

CHttpPtrMap Class

CHttpPtrMap Methods

For information about the methods in **CHttpPtrMap**, see CHttpPtrMap Members.

CHttpPtrMap::RemoveAll

Call this method to remove all of the elements from the map.

```
void RemoveAll( );
```

Remarks

This function assumes the values contained in the map are pointers to objects that should be deleted.

See Also

Reference

CHttpPtrMap Class

Other Resources

CHttpPtrMap Members

CHttpPtrMap::~CHttpPtrMap

The destructor.

```
~CHttpPtrMap( );
```

Remarks

The destructor calls CHttpPtrMap::RemoveAll to delete the contained elements.

See Also

Reference

CHttpPtrMap Class

Other Resources

CHttpPtrMap Members

CHttpRequest Class

This class represents an HTTP request received by the server.

```
class CHttpRequest :
   public IHttpRequestLookup
```

Remarks

This class provides access to the information contained in an HTTP request submitted to a web server. **CHttpRequest** provides access to the query string parameters, form variables, cookies, and files that make up an HTTP request, as well as many other important properties of the request.

Requirements

Header: atlisapi.h

Example

The following samples demonstrate **CHttpRequest**.

- BlobCache Sample
- ConfirmUser Sample
- Cookies Sample
- ForceLogin Sample
- Input Sample
- Mailer Sample
- MantaWeb Sample
- RegExp Sample
- ShowAccept Sample
- ShowBrowser Sample
- ShowErrors Sample
- ShowFiles Sample
- ShowForm Sample
- ShowImage Sample
- ShowLocalized Sample
- ShowRequest Sample
- ShowVariables Sample
- StencilCache Sample

See Also

Reference

IHttpRequestLookup Interface

Other Resources

CHttpRequest Members

ATL Server Classes

CHttpRequest Members

Methods

CHttpRequest	The constructor.
~CHttpRequest	The destructor.
AddRef	Implementation of IUnknown::AddRef.
Cookies	Call this method to retrieve a reference to the cookie with the specified name.
DeleteFiles	Call this method to delete the files managed by this object.
GetAcceptEncodings	Call this method to retrieve the value of the "HTTP_ACCEPT_ENCODING" server variable.
GetAcceptTypes	Call this method to retrieve the value of the "HTTP_ACCEPT" server variable.
GetAuthenticated	Call this method to determine whether the request was authenticated.
GetAuthenticationType	Call this method to retrieve the value of the "AUTH_TYPE" server variable.
GetAuthUserName	Call this method to retrieve the value of the "AUTH_USER" server variable.
GetAuthUserPassword	Call this method to retrieve the value of the "AUTH_PASSWORD" server variable.
GetAvailableBytes	Call this method to get the number of available bytes in the HTTP request.
GetAvailableData	Call this method to get at the available data from the current request.
GetContentType	Call this method to get the content type of the current request.
GetCookies	Call this method to retrieve the value of the "HTTP_COOKIE" server variable.
GetFirstCookie	Call this method to get the first cookie associated with an HTTP request.
GetFirstFile	Call this method to get the first file associated with an HTTP request.
GetFirstFormVar	Call this method to get the first form variable associated with an HTTP request.
GetFirstQueryParam	Call this method to get the first query parameter associated with an HTTP request.
GetFormVars	Call this method to get the form variables from the HTTP request.
GetMethod	Call this method to get the HTTP method used to make the request.
GetMethodString	Call this method to get the HTTP method used to make the request.
GetNextCookie	Call this method to get the next cookie associated with the HTTP request.
GetNextFile	Call this method to get the next file associated with the HTTP request.

GetNextFormVar	Call this method to get the next form variable associated with the HTTP request.
GetNextQueryParam	Call this method to get the next query parameter associated with an HTTP request.
GetPathInfo	Call this method to get the path of the current request from the HTTP request.
GetPathTranslated	Call this method to get the translated path of the requested resource from the HTTP request.
GetPhysicalPath	Call this method to retrieve the value of the "APPL_PHYSICAL_PATH" server variable.
GetQueryParams	Call this method to get a reference to the collection of query parameters obtained from the query string.
GetQueryString	Call this method to get the query string from the HTTP server context object.
GetScriptName	Call this method to retrieve the value of the "SCRIPT_NAME" server variable.
GetScriptPathTranslated	Call this method to get the translated path of the script handling the current request from the HTTP server context object.
GetServerContext	Call this method to get the server context associated with this request.
GetServerVariable	Call this method to get the value of a server variable from the HTTP request.
GetSessionCookie	Call this method to get the session cookie from an HTTP request.
GetTotalBytes	Call this method to get the total number of bytes in the current request.
GetUrl	Call this method to retrieve the value of the "URL" server variable.
GetUrlReferer	Call this method to retrieve the value of the "HTTP_REFERER" server variable.
GetUserAgent	Call this method to retrieve a string containing the value of the "HTTP_USER_AGENT" server variable.
GetUserHostAddress	Call this method to retrieve the value of the "REMOTE_ADDR" server variable.
GetUserHostName	Call this method to retrieve the value of the "REMOTE_HOST" server variable.
GetUserLanguages	Call this method to retrieve the value of the "HTTP_ACCEPT_LANGUAGE" server variable.
GetUserName	Call this method to retrieve the value of the "REMOTE_USER" server variable.
Initialize	Call this method to initialize the object with information about the current request.
QueryInterface	Implementation of IUnknown::QueryInterface.
Release	Implementation of IUnknown::Release.
ReadData	Call this method to read a specified amount of data from the body of the HTTP request.
SetServerContext	Call this method to set the server context of an HTTP request object.

Data Members

m_Files	The collection of files in the request.
m_requestCookies	The collection of cookies in the request.

Properties

FormVars	A property accessor for CHttpRequest::GetFormVars.
QueryParams	A property accessor for CHttpRequest::GetQueryParams.

Typedefs

FileMap	The type of map used to hold the collection of files.
CookieMap	The type of map used to hold the collection of cookies.

Enums

HTTP_METHOD	This enumeration defines the values for the HTTP methods used to make a request.

See Also

Reference

CHttpRequest Class

Other Resources

IHttpRequestLookup Members

CHttpRequest Methods

For information about the methods in **CHttpRequest**, see CHttpRequest Members.

CHttpRequest::CHttpRequest

The constructor.

```
CHttpRequest( ) throw( );
CHttpRequest(
   IHttpRequestLookup* pRequestLookup
) throw(...);
CHttpRequest(
   IHttpServerContext* pServerContext,
   DWORD dwMaxFormSize = DEFAULT_MAX_FORM_SIZE,
   DWORD dwFlags = ATL_FORM_FLAG_NONE
) throw(...);
```

Parameters

pRequestLookup

The request lookup interface used to provide access to the server context and existing collections of files, form variables, and query parameters.

pServerContext

The server context.

dwMaxFormSize

The maximum size of a request that will be parsed. If the request exceeds this size, the body of the request will not be parsed. See DEFAULT_MAX_FORM_SIZE.

dwFlags

The form flags.

Remarks

The default constructor must be followed by a call to CHttpRequest::Initialize before the object can be used.

The other constructors create and initialize the object in one step by calling **CHttpRequest::Initialize** automatically.

See Also

Reference

CHttpRequest Class

CHttpRequest::Initialize

Other Resources

CHttpRequest Members

CHttpRequest::~CHttpRequest

The destructor.

```
~CHttpRequest( ) throw( );
```

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::AddRef

Implementation of IUnknown::AddRef.

```
ULONG STDMETHODCALLTYPE AddRef( );
```

Return Value

Always returns 1.

Remarks

CHttpRequest does not implement lifetime control using reference counting.

See Also

Reference

CHttpRequest Class

CHttpRequest::Release

Other Resources

CHttpRequest Members

CHttpRequest::Cookies

Call this method to retrieve a reference to the cookie with the specified name.

```
ATL_NOINLINE const CCookie& Cookies(
   LPCSTR szName
) throw(...);
```

Parameters

szName

The name of the cookie.

Return Value

Returns a reference to the specified cookie or a reference to an empty cookie if the name cannot be found in the collection of cookies sent with the request.

Example

See the MantaWeb Sample, the ForceLogin Sample, and the StencilCache Sample.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::DeleteFiles

Call this method to delete the files managed by this object.

```
int DeleteFiles( ) throw( );
```

Return Value

Returns the number of files deleted.

Remarks

Information about the files managed by this object can be obtained from CHttpRequest::GetFirstFile and related methods. This method is called from the destructor.

Example

See the following samples:

- Mailer Sample
- RegExp Sample
- ShowFiles Sample
- ShowForm Sample

See Also

Reference

CHttpRequest Class
CHttpRequest::GetFirstFile

Other Resources

CHttpRequest Members

CHttpRequest::GetAcceptEncodings

Call this method to retrieve the value of the "HTTP_ACCEPT_ENCODING" server variable.

```
BOOL GetAcceptEncodings(
   CStringA& str
) throw( );
BOOL GetAcceptEncodings(
   LPSTR szBuff,
   DWORD* pdwSize
) throw(...);
```

Parameters

str

A reference to a **CString** that will receive the string on successful return of this method.

szBuff

A pointer to a caller-allocated buffer that will receive the string on successful return of this method.

pdwSize

[in, out] On entry, *pdwSize* should point to a DWORD that indicates the size of the buffer in bytes. On exit, the DWORD contains the number of bytes transferred or available to be transferred into the buffer (including the null-terminating byte).

Return Value

Returns TRUE on success, FALSE on failure. Call GetLastError to get extended error information.

Remarks

Call this method to retrieve a null-terminated string containing the value of the "HTTP_ACCEPT_ENCODING" server variable.

Example

See the ShowAccept Sample and the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::GetAcceptTypes

Call this method to retrieve the value of the "HTTP_ACCEPT" server variable.

```
BOOL GetAcceptTypes(
   CStringA& str
) throw( );
BOOL GetAcceptTypes(
   LPSTR szBuff,
   DWORD* pdwSize
) throw(...);
```

Parameters

str

A reference to a **CString** that will receive the string on successful return of this method.

szBuff

A pointer to a caller-allocated buffer that will receive the string on successful return of this method.

pdwSize

[in, out] On entry, *pdwSize* should point to a DWORD that indicates the size of the buffer in bytes. On exit, the DWORD contains the number of bytes transferred or available to be transferred into the buffer (including the null-terminating byte).

Return Value

Returns TRUE on success, FALSE on failure. Call GetLastError to get extended error information.

Remarks

Call this method to retrieve a null-terminated string containing the value of the "HTTP_ACCEPT" server variable.

Example

See the ShowAccept Sample and the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::GetAuthenticated

Call this method to determine whether the request was authenticated.

```
BOOL GetAuthenticated( ) throw(...);
```

Return Value

Returns TRUE if the authentication type is not empty. Returns FALSE otherwise.

Example

See the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::GetAuthenticationType

Call this method to retrieve the value of the "AUTH_TYPE" server variable.

```
BOOL GetAuthenticationType(
   CStringA& str
) throw( );
BOOL GetAuthenticationType(
   LPSTR szBuff,
   DWORD* pdwSize
) throw(...);
```

Parameters

str

A reference to a **CString** that will receive the string on successful return of this method.

szBuff

A pointer to a caller-allocated buffer that will receive the string on successful return of this method.

pdwSize

[in, out] On entry, *pdwSize* should point to a DWORD that indicates the size of the buffer in bytes. On exit, the DWORD contains the number of bytes transferred or available to be transferred into the buffer (including the null-terminating byte).

Return Value

Returns TRUE on success, FALSE on failure. Call GetLastError to get extended error information.

Remarks

Call this method to retrieve a null-terminated string containing the value of the "AUTH_TYPE" server variable.

Example

See the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::GetAuthUserName

Call this method to retrieve the value of the "AUTH_USER" server variable.

```
BOOL GetAuthUserName(
   CStringA& str
) throw( );
BOOL GetAuthUserName(
   LPSTR szBuff,
   DWORD* pdwSize
) throw(...);
```

Parameters

str

A reference to a **CString** that will receive the string on successful return of this method.

szBuff

A pointer to a caller-allocated buffer that will receive the string on successful return of this method.

pdwSize

[in, out] On entry, *pdwSize* should point to a DWORD that indicates the size of the buffer in bytes. On exit, the DWORD contains the number of bytes transferred or available to be transferred into the buffer (including the null-terminating byte).

Return Value

Returns TRUE on success, FALSE on failure. Call GetLastError to get extended error information.

Remarks

Call this method to retrieve a null-terminated string containing the value of the "AUTH_USER" server variable.

Example

See the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::GetAuthUserPassword

Call this method to retrieve the value of the "AUTH_PASSWORD" server variable.

```
BOOL GetAuthUserPassword(
   CStringA& str
) throw( );
BOOL GetAuthUserPassword(
   LPSTR szBuff,
   DWORD* pdwSize
) throw(...);
```

Parameters

str

A reference to a **CString** that will receive the string on successful return of this method.

szBuff

A pointer to a caller-allocated buffer that will receive the string on successful return of this method.

pdwSize

[in, out] On entry, *pdwSize* should point to a DWORD that indicates the size of the buffer in bytes. On exit, the DWORD contains the number of bytes transferred or available to be transferred into the buffer (including the null-terminating byte).

Return Value

Returns TRUE on success, FALSE on failure. Call GetLastError to get extended error information.

Remarks

Call this method to retrieve a null-terminated string containing the value of the "AUTH_PASSWORD" server variable.

Example

See the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::GetAvailableBytes

Call this method to get the number of available bytes in the HTTP request.

```
DWORD GetAvailableBytes( ) throw(...);
```

Return Value

Returns the number of bytes available in the request buffer accessible through CHttpRequest::GetAvailableData.

Remarks

This method is equivalent to calling IHttpServerContext::GetAvailableBytes on the server context associated with this request.

Example

See the ShowRequest Sample.

See Also

Reference

CHttpRequest Class
CHttpRequest::GetAvailableData
IHttpServerContext::GetAvailableBytes

Other Resources

CHttpRequest Members

CHttpRequest::GetAvailableData

Call this method to get at the available data from the current request.

```
LPBYTE GetAvailableData( ) throw(...);
```

Return Value

Returns a pointer to the request buffer containing the data sent by the client.

Remarks

This method is equivalent to calling IHttpServerContext::GetAvailableData on the server context associated with this request.

Example

See the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

IHttpServerContext::GetAvailableData

Other Resources

CHttpRequest Members

CHttpRequest::GetContentType

Call this method to get the content type of the current request.

```
LPCSTR GetContentType( ) throw(...);
```

Return Value

Returns a null-terminated string that contains the content type of the data sent by the client.

Remarks

This method is equivalent to calling IHttpServerContext::GetContentType on the server context associated with this request.

Example

See the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

IHttpServerContext::GetContentType

Other Resources

CHttpRequest Members

CHttpRequest::GetCookies

Call this method to retrieve the value of the "HTTP_COOKIE" server variable.

```
BOOL GetCookies(
   CStringA& strBuff
) const throw( );
BOOL GetCookies(
   LPSTR szBuf,
   LPDWORD pdwSize
) const throw(...);
```

Parameters

strBuff

A reference to a **CString** that will receive the string on successful return of this method.

szBuf

A pointer to a caller-allocated buffer that will receive the string on successful return of this method.

pdwSize

[in, out] On entry, *pdwSize* should point to a DWORD that indicates the size of the buffer in bytes. On exit, the DWORD contains the number of bytes transferred or available to be transferred into the buffer (including the null-terminating byte).

Return Value

Returns TRUE on success, FALSE on failure. Call GetLastError to get extended error information.

Remarks

Call this method to retrieve a null-terminated string containing the value of the "HTTP_COOKIE" server variable. Cookies are also available individually by calling CHttpRequest::GetCookies, CHttpRequest::GetFirstCookie, and related methods.

Example

See the ForceLogin Sample.

See Also

Reference

CHttpRequest Class
CHttpRequest::GetCookies
CHttpRequest::GetFirstCookie

Other Resources

CHttpRequest Members

CHttpRequest::GetFirstCookie

Call this method to get the first cookie associated with an HTTP request.

```
POSITION GetFirstCookie(
   LPCSTR* ppszName,
   const CCookie** ppCookie
) throw( );
```

Parameters

ppszName

The address of an LPCSTR variable that receives the name of the cookie.

ppCookie

The address of a **CCookie*** variable that receives the pointer to the cookie.

Return Value

Returns the POSITION of the next cookie in the collection or NULL if there are no cookies associated with this request.

Remarks

The POSITION returned by this method can be passed to CHttpRequest::GetNextCookie to get the next cookie in the collection.

See Also

Reference

CHttpRequest Class

CHttpRequest::GetNextCookie

Other Resources

CHttpRequest Members

CHttpRequest::GetFirstFile

Call this method to get the first file associated with an HTTP request.

```
POSITION GetFirstFile(
   LPCSTR* ppszName,
   IHttpFile** ppFile
);
```

Parameters

ppszName

The address of an LPCSTR variable that receives the local name of the file.

ppFile

The address of an **IHttpFile*** variable that receives the interface pointer to the file.

Return Value

Returns the POSITION of the next file in the collection or NULL if there are no files associated with this request.

Remarks

The local name of the file in *ppszName* is the same as the name returned by IHttpFile::GetTempFileName.

The POSITION returned by this method can be passed to CHttpRequest::GetNextFile to get the next file in the collection.

Example

See the MantaWeb Sample.

See Also

Reference

CHttpRequest Class
IHttpFile::GetTempFileName
CHttpRequest::GetNextFile

Other Resources

CHttpRequest Members

CHttpRequest::GetFirstFormVar

Call this method to get the first form variable associated with an HTTP request.

```
POSITION GetFirstFormVar(
   LPCSTR* ppszName,
   LPCSTR* ppszValue
);
```

Parameters

ppszName

The address of an LPCSTR variable that receives the name of the form variable.

ppszValue

The address of an LPCSTR variable that receives the value of the form variable.

Return Value

Returns the POSITION of the next form variable in the collection or NULL if there are no form variables associated with this request.

Remarks

The POSITION returned by this method can be passed to CHttpRequest::GetNextFormVar to get the next form variable in the collection.

Example

See the MantaWeb Sample.

See Also

Reference

CHttpRequest Class

CHttpRequest::GetNextFormVar

Other Resources

CHttpRequest Members

CHttpRequest::GetFirstQueryParam

Call this method to get the first query parameter associated with an HTTP request.

```
POSITION GetFirstQueryParam(
   LPCSTR* ppszName,
   LPCSTR* ppszValue
);
```

Parameters

ppszName

The address of an LPCSTR variable that receives the name of the query parameter.

ppszValue

The address of an LPCSTR variable that receives the value of the query parameter.

Return Value

Returns the POSITION of the next query parameter in the collection or NULL if there are no query parameters associated with this request.

Remarks

The POSITION returned by this method can be passed to CHttpRequest::GetNextQueryParam to get the next query parameter in the collection.

See Also

Reference

CHttpRequest Class
CHttpRequest::GetNextQueryParam

Other Resources

CHttpRequest Members

CHttpRequest::GetFormVars

Call this method to get the form variables from the HTTP request.

```
const CHttpRequestParams& GetFormVars( ) const throw( );
```

Return Value

Returns a reference to a collection of form variables.

Remarks

Returns a reference to the collection of form variables (name-value pairs) obtained from the query string for a GET request or from the body of the request for a POST request.

Example

See the following samples:

- ConfirmUser Sample
- Input Sample
- MantaWeb Sample
- ShowForm Sample
- ShowImage Sample

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::GetMethod

Call this method to get the HTTP method used to make the request.

```
HTTP_METHOD GetMethod( ) throw(...);
```

Return Value

Returns an HTTP_METHOD value corresponding to the HTTP method of the current request.

Example

See the ShowErrors Sample and the ShowImage Sample.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::GetMethodString

Call this method to get the HTTP method used to make the request.

```
LPCSTR GetMethodString( ) throw(...);
```

Return Value

Returns the HTTP method string.

Remarks

Returns a null-terminated string that contains the HTTP method of the current request. Equivalent to calling IHttpServerContext::GetRequestMethod on the server context associated with this request.

Example

See the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

IHttpServerContext::GetRequestMethod

Other Resources

CHttpRequest Members

CHttpRequest::GetNextCookie

Call this method to get the next cookie associated with the HTTP request.

```
POSITION GetNextCookie(
   POSITION pos,
   LPCSTR* ppszName,
   const CCookie** ppCookie
) throw( );
```

Parameters

pos

The position of the cookie.

ppszName

The address of an LPCSTR variable that receives the name of the cookie.

ppCookie

The address of a **CCookie*** variable that receives the pointer to the cookie.

Return Value

Returns the POSITION of the next cookie in the collection associated with the HTTP request or NULL if there are no further cookies.

Remarks

The POSITION passed to this method should have been returned by a previous call to CHttpRequest::GetFirstCookie or **CHttpRequest::GetNextCookie** on the same object.

The POSITION returned by this method can be passed to **CHttpRequest::GetNextCookie** to get the next cookie in the collection.

See Also

Reference

CHttpRequest Class

CHttpRequest::GetFirstCookie

Other Resources

CHttpRequest Members

CHttpRequest::GetNextFile

Call this method to get the next file associated with the HTTP request.

```
POSITION GetNextFile(
   POSITION pos,
   LPCSTR* ppszName,
   IHttpFile** ppFile
);
```

Parameters

pos

The position of the file.

ppszName

The address of an LPCSTR variable that receives the local name of the file.

ppFile

The address of an **IHttpFile*** variable that receives the interface pointer to the file.

Return Value

Returns the POSITION of the next file in the collection associated with the HTTP request or NULL if there are no further files.

Remarks

The POSITION passed to this method should have been returned by a previous call to CHttpRequest::GetFirstFile or **CHttpRequest::GetNextFile** on the same object.

The local name of the file in *ppszName* is the same as the name returned by IHttpFile::GetTempFileName.

The POSITION returned by this method can be passed to **CHttpRequest::GetNextFile** to get the next file in the collection.

Example

See the MantaWeb Sample.

See Also

Reference

CHttpRequest Class
CHttpRequest::GetFirstFile
IHttpFile::GetTempFileName

Other Resources

CHttpRequest Members

CHttpRequest::GetNextFormVar

Call this method to get the next form variable associated with the HTTP request.

```
POSITION GetNextFormVar(
   POSITION pos,
   LPCSTR* ppszName,
   LPCSTR* ppszValue
);
```

Parameters

pos

The position of the form variable.

ppszName

The address of an LPCSTR variable that receives the name of the form variable.

ppszValue

The address of an LPCSTR variable that receives the value of the form variable.

Return Value

Returns the POSITION of the next form variable in the collection associated with the HTTP request or NULL if there are no further form variables.

Remarks

The POSITION passed to this method should have been returned by a previous call to IHttpRequestLookup::GetFirstFormVar or **IHttpRequestLookup::GetNextFormVar** on the same object.

The POSITION returned by this method can be passed to **IHttpRequestLookup::GetNextFormVar** to get the next form variable in the collection.

Example

See the MantaWeb Sample.

See Also

Reference

CHttpRequest Class
IHttpRequestLookup::GetFirstFormVar
IHttpRequestLookup::GetNextFormVar

Other Resources

CHttpRequest Members

CHttpRequest::GetNextQueryParam

Call this method to get the next query parameter associated with an HTTP request.

```
POSITION GetNextQueryParam(
   POSITION pos,
   LPCSTR* ppszName,
   LPCSTR* ppszValue
);
```

Parameters

pos

The position of the query parameter.

ppszName

The address of an LPCSTR variable that receives the name of the query parameter.

ppszValue

The address of an LPCSTR variable that receives the value of the query parameter.

Return Value

Returns the POSITION of the next query parameter in the collection associated with the HTTP request or NULL if there are no further query parameters.

Remarks

The POSITION passed to this method should have been returned by a previous call to IHttpRequestLookup::GetFirstQueryParam or **IHttpRequestLookup::GetNextQueryParam** on the same object.

The POSITION returned by this method can be passed to **IHttpRequestLookup::GetNextQueryParam** to get the next query parameter in the collection.

See Also

Reference

CHttpRequest Class
IHttpRequestLookup::GetFirstQueryParam
IHttpRequestLookup::GetNextQueryParam

Other Resources

CHttpRequest Members

CHttpRequest::GetPathInfo

Call this method to get the path of the current request from the HTTP request.

```
LPCSTR GetPathInfo( ) throw(...);
```

Return Value

Returns a null-terminated string that contains the path of the current request.

Remarks

This method is equivalent to calling IHttpServerContext::GetPathInfo on the server context associated with this request.

Example

See the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

IHttpServerContext::GetPathInfo

Other Resources

CHttpRequest Members

CHttpRequest::GetPathTranslated

Call this method to get the translated path of the requested resource from the HTTP request.

```
LPCSTR GetPathTranslated( ) throw(...);
```

Return Value

Returns a null-terminated string that contains the translated path of the requested resource.

Remarks

This method is equivalent to calling IHttpServerContext::GetPathTranslated on the server context associated with this request.

Example

See the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::GetPhysicalPath

Call this method to retrieve the value of the "APPL_PHYSICAL_PATH" server variable.

```
BOOL GetPhysicalPath(
   CStringA& str
) throw( );
BOOL GetPhysicalPath(
   LPSTR szBuff,
   DWORD* pdwSize
) throw(...);
```

Parameters

str

A reference to a **CString** that will receive the string on successful return of this method.

szBuff

A pointer to a caller-allocated buffer that will receive the string on successful return of this method.

pdwSize

[in, out] On entry, *pdwSize* should point to a DWORD that indicates the size of the buffer in bytes. On exit, the DWORD contains the number of bytes transferred or available to be transferred into the buffer (including the null-terminating byte).

Return Value

Returns TRUE on success, FALSE on failure. Call GetLastError to get extended error information.

Remarks

Call this method to retrieve a null-terminated string containing the value of the "APPL_PHYSICAL_PATH" server variable.

Example

See the BlobCache Sample.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::GetQueryParams

Call this method to get a reference to the collection of query parameters obtained from the query string.

```
const CHttpRequestParams& GetQueryParams( ) const throw( );
```

Return Value

Returns a reference to the collection of query parameters (name-value pairs) obtained from the query string.

Example

See the MantaWeb Sample, the ConfirmUser Sample, and the ShowLocalized Sample.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::GetQueryString

Call this method to get the query string from the HTTP server context object.

```
LPCSTR GetQueryString( ) throw(...);
```

Return Value

Returns a null-terminated string that contains the query information from the current request.

Remarks

This method is equivalent to calling IHttpServerContext::GetQueryString on the server context associated with this request.

Example

See the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

IHttpServerContext::GetQueryString

Other Resources

CHttpRequest Members

CHttpRequest::GetScriptName

Call this method to retrieve the value of the "SCRIPT_NAME" server variable.

```
BOOL GetScriptName(
   CStringA& str
) throw( );
BOOL GetScriptName(
   LPSTR szBuff,
   DWORD* pdwSize
) throw(...);
```

Parameters

str

A reference to a **CString** that will receive the string on successful return of this method.

szBuff

A pointer to a caller-allocated buffer that will receive the string on successful return of this method.

pdwSize

[in, out] On entry, *pdwSize* should point to a DWORD that indicates the size of the buffer in bytes. On exit, the DWORD contains the number of bytes transferred or available to be transferred into the buffer (including the null-terminating byte).

Return Value

Returns TRUE on success, FALSE on failure. Call GetLastError to get extended error information.

Remarks

Call this method to retrieve a null-terminated string containing the value of the "SCRIPT_NAME" server variable.

Example

See the ConfirmUser Sample and the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::GetScriptPathTranslated

Call this method to get the translated path of the script handling the current request from the HTTP server context object.

```
LPCSTR GetScriptPathTranslated( ) throw(...);
```

Return Value

Returns a null-terminated string containing the physical path of the script.

Remarks

This method is equivalent to calling IHttpServerContext::GetScriptPathTranslated on the server context associated with this request.

Example

See the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

IHttpServerContext::GetScriptPathTranslated

Other Resources

CHttpRequest Members

CHttpRequest::GetServerContext

Call this method to get the server context associated with this request.

```
HRESULT GetServerContext(IHttpServerContext** ppOut);
```

Parameters

ppOut

[out] Address of the pointer variable that, on success, receives the IHttpServerContext interface pointer for the current request.

Return Value

Returns S_OK on success or an error HRESULT on failure.

Example

See the MantaWeb Sample.

See Also

Reference

CHttpRequest Class

IHttpServerContext Interface

Other Resources

CHttpRequest Members

CHttpRequest::GetServerVariable

Call this method to get the value of a server variable from the HTTP request.

```
BOOL GetServerVariable(
   LPCSTR szVariable,
   CStringA& str
) const throw( );
```

Parameters

szVariable

The name of the server variable.

str

A reference to a **CString** that will receive the string on successful return of this method.

Return Value

Returns TRUE on success, FALSE on failure. Call GetLastError to get extended error information.

Remarks

This method is equivalent to calling IHttpServerContext::GetServerVariable on the server context associated with this request.

Example

See the following samples:

- ConfirmUser Sample
- Cookies Sample
- ShowAccept Sample
- ShowBrowser Sample
- ShowVariables Sample

See Also

Reference

CHttpRequest Class

IHttpServerContext::GetServerVariable

Other Resources

CHttpRequest Members

CHttpRequest::GetSessionCookie

Call this method to get the session cookie from an HTTP request.

```
const CCookie& GetSessionCookie( ) throw(...);
```

Return Value

Returns a reference to the session cookie or a reference to an empty cookie if the session cookie isn't found in the request.

Remarks

The session cookie is a cookie with the name SESSION_COOKIE_NAME.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::GetTotalBytes

Call this method to get the total number of bytes in the current request.

```
DWORD GetTotalBytes( ) throw(...);
```

Return Value

Returns the total number of bytes to be received from the client.

Remarks

This method is equivalent to calling IHttpServerContext::GetTotalBytes on the server context associated with this request.

Example

See the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

IHttpServerContext::GetTotalBytes

Other Resources

CHttpRequest Members

CHttpRequest::GetUrl

Call this method to retrieve the value of the "URL" server variable.

```
BOOL GetUrl(
   CStringA& str
) throw( );
BOOL GetUrl(
   LPSTR szBuff,
   DWORD * pdwSize
) throw(...);
```

Parameters

str

A reference to a **CString** that will receive the string on successful return of this method.

szBuff

A pointer to a caller-allocated buffer that will receive the string on successful return of this method.

pdwSize

[in, out] On entry, *pdwSize* should point to a DWORD that indicates the size of the buffer in bytes. On exit, the DWORD contains the number of bytes transferred or available to be transferred into the buffer (including the null-terminating byte).

Return Value

Returns TRUE on success, FALSE on failure. Call GetLastError to get extended error information.

Remarks

Call this method to retrieve a null-terminated string containing the value of the "URL" server variable.

Example

See the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::GetUrlReferer

Call this method to retrieve the value of the "HTTP_REFERER" server variable.

```
BOOL GetUrlReferer(
   CStringA& str
) throw( );
BOOL GetUrlReferer(
   LPSTR szBuff,
   DWORD* pdwSize
) throw(...);
```

Parameters

str

A reference to a **CString** that will receive the string on successful return of this method.

szBuff

A pointer to a caller-allocated buffer that will receive the string on successful return of this method.

pdwSize

[in, out] On entry, *pdwSize* should point to a DWORD that indicates the size of the buffer in bytes. On exit, the DWORD contains the number of bytes transferred or available to be transferred into the buffer (including the null-terminating byte).

Return Value

Returns TRUE on success, FALSE on failure. Call GetLastError to get extended error information.

Remarks

Call this method to retrieve a null-terminated string containing the value of the "HTTP_REFERER" server variable.

Example

See the ShowFiles Sample and the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::GetUserAgent

Call this method to retrieve a string containing the value of the "HTTP_USER_AGENT" server variable.

```
BOOL GetUserAgent(
   CStringA& str
) throw( );
BOOL GetUserAgent(
   LPSTR szBuff,
   DWORD* pdwSize
) throw(...);
```

Parameters

str

A reference to a **CString** that will receive the string on successful return of this method.

szBuff

A pointer to a caller-allocated buffer that will receive the string on successful return of this method.

pdwSize

[in, out] On entry, *pdwSize* should point to a DWORD that indicates the size of the buffer in bytes. On exit, the DWORD contains the number of bytes transferred or available to be transferred into the buffer (including the null-terminating byte).

Return Value

Returns TRUE on success, FALSE on failure. Call GetLastError to get extended error information.

Remarks

Call this method to retrieve a null-terminated string containing the value of the "HTTP_USER_AGENT" server variable.

Example

See the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::GetUserHostAddress

Call this method to retrieve the value of the "REMOTE_ADDR" server variable.

```
BOOL GetUserHostAddress(
   CStringA& str
) throw( );
BOOL GetUserHostAddress(
   LPSTR szBuff,
   DWORD* pdwSize
) throw(...);
```

Parameters

str

A reference to a **CString** that will receive the string on successful return of this method.

szBuff

A pointer to a caller-allocated buffer that will receive the string on successful return of this method.

pdwSize

[in, out] On entry, *pdwSize* should point to a DWORD that indicates the size of the buffer in bytes. On exit, the DWORD contains the number of bytes transferred or available to be transferred into the buffer (including the null-terminating byte).

Return Value

Returns TRUE on success, FALSE on failure. Call GetLastError to get extended error information.

Remarks

Call this method to retrieve a null-terminated string containing the value of the "REMOTE_ADDR" server variable.

Example

See the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::GetUserHostName

Call this method to retrieve the value of the "REMOTE_HOST" server variable.

```
BOOL GetUserHostName(
   CStringA& str
) throw( );
BOOL GetUserHostName(
   LPSTR szBuff,
   DWORD* pdwSize
) throw(...);
```

Parameters

str

A reference to a **CString** that will receive the string on successful return of this method.

szBuff

A pointer to a caller-allocated buffer that will receive the string on successful return of this method.

pdwSize

[in, out] On entry, *pdwSize* should point to a DWORD that indicates the size of the buffer in bytes. On exit, the DWORD contains the number of bytes transferred or available to be transferred into the buffer (including the null-terminating byte).

Return Value

Returns TRUE on success, FALSE on failure. Call GetLastError to get extended error information.

Remarks

Call this method to retrieve a null-terminated string containing the value of the "REMOTE_HOST" server variable.

Example

See the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::GetUserLanguages

Call this method to retrieve the value of the "HTTP_ACCEPT_LANGUAGE" server variable.

```
BOOL GetUserLanguages(
   CStringA& str
) throw( );
BOOL GetUserLanguages(
   LPSTR szBuff,
   DWORD* pdwSize
) throw(...);
```

Parameters

str

A reference to a **CString** that will receive the string on successful return of this method.

szBuff

A pointer to a caller-allocated buffer that will receive the string on successful return of this method.

pdwSize

[in, out] On entry, *pdwSize* should point to a DWORD that indicates the size of the buffer in bytes. On exit, the DWORD contains the number of bytes transferred or available to be transferred into the buffer (including the null-terminating byte).

Return Value

Returns TRUE on success, FALSE on failure. Call GetLastError to get extended error information.

Remarks

Call this method to retrieve a null-terminated string containing the value of the "HTTP_ACCEPT_LANGUAGE" server variable.

Example

See the ShowAccept Sample and the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::GetUserName

Call this method to retrieve the value of the "REMOTE_USER" server variable.

```
BOOL GetUserName(
   CStringA& str
) throw( );
BOOL GetUserName(
   LPSTR szBuff,
   DWORD* pdwSize
) throw(...);
```

Parameters

str

A reference to a **CString** that will receive the string on successful return of this method.

szBuff

A pointer to a caller-allocated buffer that will receive the string on successful return of this method.

pdwSize

[in, out] On entry, *pdwSize* should point to a DWORD that indicates the size of the buffer in bytes. On exit, the DWORD contains the number of bytes transferred or available to be transferred into the buffer (including the null-terminating byte).

Return Value

Returns TRUE on success, FALSE on failure. Call GetLastError to get extended error information.

Remarks

Call this method to retrieve a null-terminated string containing the value of the "REMOTE_USER" server variable.

Example

See the ShowRequest Sample.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::Initialize

Call this method to initialize the object with information about the current request.

```
BOOL Initialize(
   IHttpRequestLookup* pRequestLookup
) throw( );
BOOL Initialize(
   IHttpServerContext* pServerContext,
   DWORD dwMaxFormSize = DEFAULT_MAX_FORM_SIZE,
   DWORD dwFlags = ATL_FORM_FLAG_NONE
) throw( );
```

Parameters

pRequestLookup

The request lookup interface used to provide access to the server context, and existing collections of files, form variables, and query parameters.

pServerContext

The server context.

dwMaxFormSize

The maximum size of a request that will be parsed. If the request exceeds this size, the body of the request will not be parsed. See DEFAULT_MAX_FORM_SIZE.

dwFlags

The form flags.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Call **Initialize** directly or using the appropriate constructor before using the methods and properties of the request object.

The first overload initializes the request object using an IHttpRequestLookup interface to supply the server context and existing collections of files, form variables, and query parameters.

The second overload initializes the request object using a server context and builds up the files, form variables, and query parameters from the raw data exposed by that context. This overload of **Initialize** does the following:

- Parses and decodes the query string into a collection of name-value pairs. This collection is accessible through the CHttpRequest::GetQueryParams method.
- Parses the body of a POST request if the size of the request data is less than or equal to *dwMaxFormSize*. The body of the request will consist of simple form variables and may also contain files if the request is encoded as multipart/form-data. In that case, the *dwFlags* parameter is passed to CMultiPartFormParser::GetMultiPartData to control the creation of the files. The collection of form variables is accessible through the CHttpRequest::GetFormVars method or the **FormVars** property. The collection of files is accessible through the CHttpRequest::m_Files member.

Note that **Initialize** does not parse the cookies associated with a request. Cookies are not processed until an attempt is made to access a cookie in the collection.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::QueryInterface

Implementation of IUnknown::QueryInterface.

```
HRESULT STDMETHODCALLTYPE QueryInterface(
   REFIID riid,
   void ** ppv
);
```

Remarks

Objects of this class can be successfully queried for the IUnknown and IHttpRequestLookup interfaces.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::Release

Implementation of IUnknown::Release.

```
ULONG STDMETHODCALLTYPE Release( );
```

Return Value

Always returns 1.

Remarks

CHttpRequest does not implement lifetime control using reference counting.

See Also

Reference

CHttpRequest Class

CHttpRequest::AddRef

Other Resources

CHttpRequest Members

CHttpRequest::ReadData

Call this method to read a specified amount of data from the body of the HTTP request.

```
BOOL ReadData(
   LPSTR pDest,
   LPDWORD pdwLen
) throw( );
```

Parameters

pDest

A pointer to a caller-allocated buffer that will receive the data on successful return of this method.

pdwLen

[in, out] On entry, *pdwLen* should point to a DWORD that indicates the size of the buffer in bytes. On exit, the DWORD contains the number of bytes transferred or available to be transferred into the buffer.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Typically there is no need to call this method unless the request is larger than the value used for the maximum form size and you need to do custom parsing of the request body.

Note that the data is unlikely to be null-terminated.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest::SetServerContext

Call this method to set the server context of an HTTP request object.

```
void SetServerContext(
   IHttpServerContext* pServerContext
) throw( );
```

Parameters

pServerContext

The server context.

See Also

Reference

CHttpRequest Class

Other Resources

CHttpRequest Members

CHttpRequest Data Members

For information about the data members in **CHttpRequest**, see CHttpRequest Members.

CHttpRequest::m_Files

The collection of files in the request.

```
FileMap m_Files;
```

Example

See the RegExp Sample and the ShowFiles Sample.

See Also

Reference

CHttpRequest Class

CHttpRequest::FileMap

Other Resources

CHttpRequest Members

CHttpRequest::m_requestCookies

The collection of cookies in the request.

```
CookieMap m_requestCookies;
```

Example

See the Cookies Sample.

See Also

Reference

CHttpRequest Class

CHttpRequest::CookieMap

Other Resources

CHttpRequest Members

CHttpRequest Typedefs

For information about the typedefs in **CHttpRequest**, see CHttpRequest Members.

CHttpRequest::FileMap

The type of map used to hold the collection of files.

```
typedef CHttpPtrMap<
   CStringA,
   IHttpFile*,
   CStringElementTraits<CStringA>
> FileMap;
```

Example

See the ShowFiles Sample and the ShowImage Sample.

See Also

Reference

CHttpRequest Class

CHttpRequest::m_Files

Other Resources

CHttpRequest Members

CHttpRequest::CookieMap

The type of map used to hold the collection of cookies.

```
typedef CHttpMap<
   CStringA,
   CCookie,
   CStringElementTraits<CStringA>
> CookieMap;
```

See Also

Reference

CHttpRequest Class

CHttpRequest::m_requestCookies

Other Resources

CHttpRequest Members

CHttpRequest Enums

For information about the enums in **CHttpRequest**, see CHttpRequest Members.

CHttpRequest::HTTP_METHOD

This enumeration defines the values for the HTTP methods used to make a request.

```
enum HTTP_METHOD {
   HTTP_METHOD_UNKNOWN = -1, // Unknown request method
   HTTP_METHOD_GET,          // GET request
   HTTP_METHOD_POST,         // POST request
   HTTP_METHOD_HEAD,         // HEAD request
   HTTP_METHOD_DELETE,       // DELETE request
   HTTP_METHOD_LINK,         // LINK request
   HTTP_METHOD_UNLINK,       // UNLINK request
   HTTP_METHOD_DEBUG         // Debugging support
};
```

Remarks

See the HTTP 1.1 RFC (http://ietf.org/rfc/rfc2616.txt) for a description of the different HTTP request methods identified by these enumeration values.

Example

See the ShowErrors Sample and the ShowImage Sample.

See Also

Reference

CHttpRequest Class
CHttpRequest::GetMethod

Other Resources

CHttpRequest Members

Properties (ATL Server)

For information about the properties in **CHttpRequest**, see CHttpRequest Members.

CHttpRequest::FormVars

A property accessor for CHttpRequest::GetFormVars.

```
__declspec(property( get = GetFormVars ))
const CHttpRequestParams& FormVars;
```

See Also

Reference

CHttpRequest Class

CHttpRequest::GetFormVars

Other Resources

CHttpRequest Members

CHttpRequest::QueryParams

A property accessor for CHttpRequest::GetQueryParams.

```
__declspec(property( get = GetQueryParams ))
const CHttpRequestParams& QueryParams;
```

See Also

Reference

CHttpRequest Class

CHttpRequest::GetQueryParams

Other Resources

CHttpRequest Members

CHttpRequestFile Class

This class represents an uploaded file and implements the IHttpFile interface.

```
class CHttpRequestFile :
   public IHttpFile
```

Remarks

This class trivially implements the IHttpFile interface. Pass the necessary information to the constructor and use the base class methods to access the properties of the file.

Requirements

Header: atlisapi.h

See Also

Tasks

Mailer Sample: Demonstrates SMTP Mail Classes

RegExp Sample: Demonstrates ATL Regular Expression Classes

Reference

ShowFiles

ShowImage

IHttpFile Interface

Other Resources

CHttpRequestFile Members

ATL Server Classes

CHttpRequestFile Members

Methods

CHttpRequestFile	The constructor.
Free	Calls **delete this**.
GetContentType	Implementation of IHttpFile::GetContentType.
GetFileName	Implementation of IHttpFile::GetFileName.
GetFileSize	Implementation of IHttpFile::GetFileSize.
GetFullFileName	Implementation of IHttpFile::GetFullFileName.
GetParamName	Implementation of IHttpFile::GetParamName.
GetTempFileName	Implementation of IHttpFile::GetTempFileName.
Initialize	Call this method to initialize the object.

See Also

Reference

CHttpRequestFile Class

Other Resources

IHttpFile Members

CHttpRequestFile Methods

For information about the methods in **CHttpRequestFile**, see CHttpRequestFile Members and IHttpFile Members.

CHttpRequestFile::CHttpRequestFile

The constructor.

```
CHttpRequestFile( ) throw( );
```

Remarks

Initialization data is passed to CHttpRequestFile::Initialize.

See Also

Reference

CHttpRequestFile Class

Other Resources

CHttpRequestFile Members

CHttpRequestFile::Free

Calls **delete this**.

```
void Free( );
```

See Also

Reference

CHttpRequestFile Class

Other Resources

CHttpRequestFile Members

CHttpRequestFile::GetContentType

Implementation of IHttpFile::GetContentType.

```
LPCSTR GetContentType( );
```

Example

See the ShowFiles Sample and theShowImage Sample.

See Also

Reference

CHttpRequestFile Class

Other Resources

CHttpRequestFile Members

CHttpRequestFile::GetFileName

Implementation of IHttpFile::GetFileName.

```
LPCSTR GetFileName( );
```

Example

See the Mailer Sample, the ShowFiles Sample, and theShowImage Sample.

See Also

Reference

CHttpRequestFile Class

Other Resources

CHttpRequestFile Members

CHttpRequestFile::GetFileSize

Implementation of IHttpFile::GetFileSize.

```
ULONGLONG GetFileSize( );
```

Example

See the RegExp Sample, the ShowFiles Sample, and theShowImage Sample.

See Also

Reference

CHttpRequestFile Class

Other Resources

CHttpRequestFile Members

CHttpRequestFile::GetFullFileName

Implementation of IHttpFile::GetFullFileName.

```
LPCSTR GetFullFileName( );
```

Example

See the ShowFiles Sample and theShowImage Sample.

See Also

Reference

CHttpRequestFile Class

Other Resources

CHttpRequestFile Members

CHttpRequestFile::GetParamName

Implementation of IHttpFile::GetParamName.

```
LPCSTR GetParamName( );
```

Example

See the ShowFiles Sample and theShowImage Sample.

See Also

Reference

CHttpRequestFile Class

Other Resources

CHttpRequestFile Members

CHttpRequestFile::GetTempFileName

Implementation of IHttpFile::GetTempFileName.

```
LPCSTR GetTempFileName( );
```

Example

See the following samples:

- Mailer Sample
- RegExp Sample
- ShowFiles Sample
- ShowImage Sample

See Also

Reference

CHttpRequestFile Class

Other Resources

CHttpRequestFile Members

CHttpRequestFile::Initialize

Call this method to initialize the object.

```
BOOL Initialize(
   LPCSTR pParamName,
   LPCSTR pFileName,
   LPCSTR pTempFileName,
   LPCSTR pContentType,
   const ULONGLONG & nFileSize
) throw( );
```

Parameters

pParamName

The name of the form field used to upload the file. Provides the data for IHttpFile::GetParamName. Only the first MAX_TOKEN_LENGTH - 1 characters are copied.

pFileName

The original path and file name of the uploaded file as set by the client. Provides the data for IHttpFile::GetFullFileName. Only the first MAX_PATH - 1 characters are copied.

pTempFileName

The name of the uploaded file on the server. Provides the data for IHttpFile::GetTempFileName. Only the first MAX_PATH - 1 characters are copied.

pContentType

The MIME type of the uploaded file. Provides the data for IHttpFile::GetContentType. Only the first MAX_TOKEN_LENGTH - 1 characters are copied.

nFileSize

The size of the file in bytes. Provides the data for IHttpFile::GetFileSize.

Remarks

The information necessary for implementing the IHttpFile interface is passed to this method. IHttpFile::GetFileName is calculated from the *pFileName* argument.

See Also

Reference

CHttpRequestFile Class

Other Resources

CHttpRequestFile Members

CHttpRequestParams Class

This class represents a collection of name-value pairs such as query parameters or form variables.

```
class CHttpRequestParams :
#if (defined(ATL_HTTP_PARAM_MAP_CASEINSENSITIVE))
    public CHttpMap<CStringA, CStringA, CStringElementTraitsI<CStringA>,
                    CStringElementTraitsI<CStringA> >,
#else
    public CHttpMap<CStringA, CStringA, CStringElementTraits<CStringA>,
                    CStringElementTraits<CStringA> >,
#endif
    public CValidateObject<CHttpRequestParams>
```

Remarks

This class represents a collection of request parameters — the name-value pairs found, for example, in a query string or in the data provided when a form is submitted to the server. Call CHttpRequestParams::Parse to build the collection from a string of URL-encoded data. Use the standard collection methods of the CHttpMap base class to retrieve the decoded names and values. Use the methods of the CValidateObject base class to validate the parameters.

The behavior of this class is altered by the ATL_HTTP_PARAM_MAP_CASEINSENSITIVE and ATL_HTTP_PARAM_MULTIMAP macros.

When `ATL_HTTP_PARAM_MAP_CASEINSENSITIVE` is defined, request parameters are treated without regard to case, meaning that a parameter named **UserName** is equivalent to one named **username**. When this macro is not defined, these parameters are treated as different.

When `ATL_HTTP_PARAM_MULTIMAP` is defined, multiple request parameters with the same name are available. When this macro is not defined, only the last encountered parameter with a particular name is available.

Requirements

Header: atlisapi.h

Example

See the following samples:

- ConfirmUser Sample
- Input Sample
- MantaWeb Sample
- ShowForm Sample
- ShowImage Sample

See Also

Other Resources

CHttpRequestParams Members
ATL Server Classes
ATL Server Classes

CHttpRequestParams Members

Methods

Lookup	Call this method to look up a value in the collection of request parameters.
Parse	Call this method to build a collection from a string of URL-encoded name-value pairs.
Render	Call this method to render the map of names and values into a buffer as a URL-encoded string.

Typedefs

BaseMap	This typedef allows easy access to the type of map used as the base class.

See Also

Reference

CHttpRequestParams Class

Other Resources

CValidateObject Members

CHttpMap Members

CHttpRequestParams Methods

For information about the methods in **CHttpRequestParams**, see CHttpRequestParams Members.

CHttpRequestParams::Lookup

Call this method to look up a value in the collection of request parameters.

```
LPCSTR Lookup(
   LPCSTR szName
) const throw( );
```

Parameters

szName

The name of a query parameter or form variable.

Return Value

Returns the value corresponding to *szName* or NULL if the name is not found in the collection.

Example

See the ConfirmUser Sample, and the MantaWeb Sample, and the Input Sample.

See Also

Reference

CHttpRequestParams Class

Other Resources

CHttpRequestParams Members

CHttpRequestParams::Parse

Call this method to build a collection from a string of URL-encoded name-value pairs.

```
ATL_NOINLINE BOOL Parse(
   LPSTR szQueryString
) throw( );
```

Parameters

szQueryString

The URL-encoded data containing the collection of name-value pairs.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

On success, the collection represented by this object is populated with the name-value pairs in the string passed to this method.

The URL-encoded data is expected to have the following format:

- Each name-value pair is separated from the next by an ampersand (&).
- Each name is separated from its corresponding value by an equal sign (=).
- The end of the data to be parsed is indicated by a null character (\0) or a pound symbol (#).
- A plus sign (+) in the input will be decoded as a space.
- A percent sign (%) in the input signifies the start of an escaped octet. The next two digits represent the hexadecimal code of the character. For example, %21 is the escaped encoding for the US-ASCII exclamation mark and will be decoded as !.

Common sources of URL-encoded data are query strings and the bodies of POST requests with content type of application/x-www-form-urlencoded.

Parse and Render are complementary operations. **Parse** creates a collection from a string. **Render** creates a string from a collection.

See Also

Reference

CHttpRequestParams Class

CHttpRequestParams::Render

Other Resources

CHttpRequestParams Members

CHttpRequestParams::Render

Call this method to render the map of names and values into a buffer as a URL-encoded string.

```
ATL_NOINLINE BOOL Render(
   LPSTR szParameters,
   LPDWORD pdwLen
) throw( );
```

Parameters

szParameters

The caller-allocated buffer to receive the output of the method. This parameter may not be **NULL**.

pdwLen

On entry, *pdwLen* should point to a **DWORD** that indicates the size of the *szParameters* buffer, in bytes. On exit, the value of this parameter will depend on the return value. See the Remarks section for more information.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

On success, the buffer will contain the correct URL-encoded representation of the current object suitable for sending to a server as a query string or in the body of a form.

This function creates the string representation of the collection's contents in the following way:

- Each name-value pair is separated from the next by an ampersand (&).
- Each name is separated from its corresponding value by an equal sign (=).
- A space is encoded as a plus sign (+).
- Other unsafe characters (as determined by AtlIsUnsafeUrlChar) are encoded as escaped octets. An escaped octet is a percent sign (%) followed by two digits representing the hexadecimal code of the character.

Parse and **Render** are complementary operations. **Parse** creates a collection from a string. **Render** creates a string from a collection.

Note that if this function returns **FALSE**, the **DWORD** value pointed to by *pdwLen* will contain the number of bytes required in order for the operation to succeed, including the null-terminating byte. If this function returns **TRUE**, the **DWORD** value pointed to by *pdwLen* will contain the number of bytes transferred into the buffer, excluding the null-terminating byte.

Example

```
// The following example demonstrates rendering a map of
// name, value pairs into into a buffer as a URL-encoded string

CHttpRequestParams requestParams;

// Add a few key, value pairs to the request map
// The tab characters (\t) will be encoded as escaped octets
requestParams.SetAt("P1", "\t1");
requestParams.SetAt("P2", "\t2");

// byte length of the above string buffer
DWORD dwLen = 0L;
// buffer to hold URL-encoded string
LPSTR strParams = new CHAR[dwLen];

// Attempt to render the map to the provided string buffer
BOOL bSuccess = requestParams.Render(strParams, &dwLen);
if (!bSuccess)
{
```

```
    // dwLen now refers to the number of bytes required in order
    // to succeed
    // Reattempt rendering using the updated string length
    delete[] strParams;
    strParams = new CHAR[dwLen];
    bSuccess = requestParams.Render(strParams, &dwLen);
    if (!bSuccess)
    {
       // Rendering failed
       delete[] strParams;
       return E_FAIL;
    }
 }

 // Clean up
 delete[] strParams;
 return S_OK;
```

See Also

Reference

CHttpRequestParams Class

CHttpRequestParams::Parse

Other Resources

CHttpRequestParams Members

CHttpRequestParams Typedefs

For information about the typedefs in **CHttpRequestParams**, see CHttpRequestParams Members.

CHttpRequestParams::BaseMap

This typedef allows easy access to the type of map used as the base class.

```
#if (defined(ATL_HTTP_PARAM_MAP_CASEINSENSITIVE))
   typedef CHttpMap<CStringA, CStringA, CStringElementTraitsI<CStringA>, CStringElementTrai
tsI<CStringA> > BaseMap;
#else
   typedef CHttpMap<CStringA, CStringA, CStringElementTraits<CStringA>, CStringElementTrait
s<CStringA> > BaseMap;
#endif
```

Example

See the ShowImage Sample.

See Also

Reference

CHttpRequestParams Class

CHttpMap Class

Other Resources

CHttpRequestParams Members

CHttpResponse Class

This class represents an HTTP response that can be sent back to the client.

```
class CHttpResponse :
   public IWriteStream,
   public CWriteStreamHelper
```

Remarks

CHttpResponse provides friendly methods for building up the headers, cookies, and body of an HTTP response. The class derives from IWriteStream and CWriteStreamHelper, allowing you to call those classes' methods to build up the body of the response. By default, the class improves performance by buffering the response until it is complete before sending it back to the client.

To use this class, be sure to provide a pointer to an IHttpServerContext using the constructor or CHttpResponse::Initialize before calling other methods.

Requirements

Header: atlisapi.h

Example

See the following samples:

- ConfirmUser Sample
- Cookies Sample
- ForceLogin Sample
- MantaWeb Sample
- NotModified Sample
- PooledAsync Sample
- SessionSettings Sample
- ShowErrors Sample
- ShowImage Sample
- SRFSyntax Sample
- StencilCache Sample

See Also

Tasks

WebFeatures Sample: Demonstrates Various ATL Server Tasks
StencilCache Sample: Uses the ATL Server Stencil Cache
SRFSyntax Sample: Server Response File Demonstration
MantaWeb Sample: Online Personal Information Management

Reference

IWriteStream Interface
CWriteStreamHelper Class

Other Resources

CHttpResponse Members
ATL Server Classes

CHttpResponse Members

Methods

AppendCookie	Call this method to add a Set-Cookie header to the collection of HTTP headers managed by this object.
AppendHeader	Call this method to append a header to the collection of HTTP headers managed by this object.
AsyncFlush	Call this method to write buffered data to the client asynchronously.
AsyncPrep	Call this method to prepare an HTTP response object for writing asynchronously.
CHttpResponse	The constructor.
~CHttpResponse	The destructor.
ClearContent	Call this method to clear the response buffer without sending the contents to the client.
ClearHeaders	Call this method to clear the collection of HTTP response headers maintained by this object.
ClearResponse	Call this method to clear the response object of any headers and the contents of the buffer.
DeleteCookie	Call this method to add a Set-Cookie header to the collection of HTTP headers managed by this object th at will tell the client to discard a cookie value that it already holds.
Detach	Call this method to release the server context held by this object to ensure that any buffered data will not be sent when the destructor is called.
Flush	Call this method to empty the response buffer and send its current contents to the client.
FlushStream	Call this method to empty the response buffer and send its current contents to the client.
GetBufferLimit	Returns the current size limit of the response buffer in bytes.
GetBufferOutput	Call this method to determine whether data written to the response object is being buffered or not.
GetContentType	Call this method to get the content type from an HTTP response object.
GetServerContext	Call this method to get the server context from an HTTP response object.
GetStatusCode	Call this method to get the status code from an HTTP response object.
GetWriteToClient	Call this method to determine whether or not the response body will be sent to the client.
HaveSentHeaders	Call this method to find out or specify whether response headers have been sent to the client.
Initialize	Call this method to initialize an HTTP response object.
Redirect	Call this method to redirect the client to a different resource.
RenderHeaders	Call this method to get a string containing all of the HTTP headers associated with this object in a format suitable for sending to a client.

SendHeadersInternal	Call this method to send the current headers associated with this object to the client.
SetBufferLimit	Call this method to set a size limit on the amount of data buffered by the response object.
SetBufferOutput	Call this method to set buffering options for the response.
SetCacheControl	Call this method to set the Cache-Control HTTP header of the response.
SetContentType	Call this method to set the content type of the HTTP response.
SetExpires	Call this method to set the Expires header of the HTTP response.
SetExpiresAbsolute	Call this method to set the expiration time of the HTTP response.
SetStatusCode	Call this method to set the HTTP status code of the response.
SetWriteToClient	Call this method to specify whether a response body will be sent to the client.
TransmitFile	Call this method to transmit a file as the HTTP response.
WriteLen	Call this method to write data to the body of the HTTP response.
WriteStream	Call this method to write data to the response object.

Data Members

m_strContent	This data member holds the buffered content.

Enums

HTTP_REDIRECT	This enumeration defines the values for the HTTP status codes used to redirect a client to another resource.

See Also

Reference

CHttpResponse Class

Other Resources

IWriteStream Members

CWriteStreamHelper Members

CHttpResponse Methods

For information about the methods in **CHttpResponse**, see CHttpResponse Members.

CHttpResponse::AppendCookie

Call this method to add a Set-Cookie header to the collection of HTTP headers managed by this object.

```
BOOL AppendCookie(
   LPCSTR szName,
   LPCSTR szValue
) throw( );
BOOL AppendCookie(
   const CCookie& cookie
) throw( );
BOOL AppendCookie(
   const CCookie* pCookie
) throw( );
```

Parameters

szName

A null-terminated string containing the name of the cookie to be sent to the client.

szValue

A null-terminated string containing the value of the cookie to be sent to the client.

cookie

A reference to a CCookie object describing the cookie to be sent to the client.

pCookie

A pointer to a CCookie object describing the cookie to be sent to the client.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Note that cookies are immediately added to the other response headers managed by this object. Call CHttpResponse::ClearHeaders if you need to remove cookies and other headers from the response.

Example

See the following samples:

- Cookies Sample
- ForceLogin Sample
- MantaWeb Sample
- StencilCache Sample

See Also

Reference

CHttpResponse Class
CCookie Class
CHttpResponse::ClearHeaders

Other Resources

CHttpResponse Members

CHttpResponse::AppendHeader

Call this method to append a header to the collection of HTTP headers managed by this object.

```
BOOL AppendHeader(
   LPCSTR szName,
   LPCSTR szValue
) throw();
```

Parameters

szName

A null-terminated string containing the name of the HTTP header.

szValue

A null-terminated string containing the value of the HTTP header.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Call CHttpResponse::ClearHeaders if you need to remove cookies and other headers from the response.

Example

See the following samples:

- ConfirmUser Sample
- MantaWeb Sample
- NotModified Sample
- ShowImage Sample

See Also

Reference

CHttpResponse Class
CHttpResponse::ClearHeaders

Other Resources

CHttpResponse Members

CHttpResponse::AsyncFlush

Call this method to write buffered data to the client asynchronously.

```
BOOL AsyncFlush( ) throw( );
```

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This method writes the buffered response to the client asynchronously. If response headers have not yet been sent, they will be sent as a result of a call to this method.

See Also

Reference

CHttpResponse Class

Concepts

Writing Asynchronous Web Applications

Other Resources

CHttpResponse Members

CHttpResponse::AsyncPrep

Call this method to prepare an HTTP response object for writing asynchronously.

```
BOOL AsyncPrep(
   BOOL fKeepConn = FALSE
) throw( );
```

Parameters

fKeepConn

TRUE if the HTTP connection to the client should be kept, FALSE otherwise.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This method sends the response headers to the client if they have not already been sent.

Example

See the PooledAsync Sample.

See Also

Reference

CHttpResponse Class

Concepts

Writing Asynchronous Web Applications

Other Resources

CHttpResponse Members

CHttpResponse::CHttpResponse

The constructor.

```
CHttpResponse( ) throw( );
CHttpResponse(
   IHttpServerContext* pServerContext
) throw( );
```

Parameters

pServerContext

The server context for the current request.

Remarks

If you use the default constructor, call CHttpResponse::Initialize before calling other methods in this object.

See Also

Reference

CHttpResponse Class

CHttpResponse::Initialize

Other Resources

CHttpResponse Members

CHttpResponse::~CHttpResponse

The destructor.

```
~CHttpResponse( ) throw( );
```

Remarks

The destructor sends any remaining buffered content to the client.

See Also

Reference

CHttpResponse Class

Other Resources

CHttpResponse Members

CHttpResponse::ClearContent

Call this method to clear the response buffer without sending the contents to the client.

```
void ClearContent( ) throw( );
```

Remarks

Note that calling this method does not clear the response headers. To clear the response headers call CHttpResponse::ClearHeaders or CHttpResponse::ClearResponse.

If you need to empty the buffer by sending the current contents to the client, call CHttpResponse::Flush instead.

See Also

Reference

CHttpResponse Class
CHttpResponse::Flush
CHttpResponse::ClearHeaders
CHttpResponse::ClearResponse

Other Resources

CHttpResponse Members

CHttpResponse::ClearHeaders

Call this method to clear the collection of HTTP response headers maintained by this object.

```
void ClearHeaders( ) throw( );
```

Remarks

Note that clearing the headers includes removing all cookies associated with the response object because cookies are sent to the client as Set-Cookie HTTP headers.

See Also

Reference

CHttpResponse Class

Other Resources

CHttpResponse Members

CHttpResponse::ClearResponse

Call this method to clear the response object of any headers and the contents of the buffer.

```
void ClearResponse( ) throw( );
```

Remarks

Equivalent to calling CHttpResponse::ClearHeaders and CHttpResponse::ClearContent together.

Example

See the following samples:

- ConfirmUser Sample
- MantaWeb Sample
- ShowErrors Sample
- ShowImage Sample

See Also

Reference

CHttpResponse Class
CHttpResponse::ClearHeaders
CHttpResponse::ClearContent

Other Resources

CHttpResponse Members

CHttpResponse::DeleteCookie

Call this method to add a Set-Cookie header to the collection of HTTP headers managed by this object that will tell the client to discard a cookie value that it already holds.

```
BOOL DeleteCookie(
   LPCSTR szName
) throw( );
```

Parameters

szName

A null-terminated string containing the name of the cookie to be discarded.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This method adds a Set-Cookie header with a Max-Age attribute of 0. When the client receives the response, it should discard the cookie of the same name that it already holds.

Calling this method will *not* remove a Set-Cookie header from the collection of HTTP headers managed by this object.

See Also

Reference

CHttpResponse Class

Other Resources

CHttpResponse Members

CHttpResponse::Detach

Call this method to release the server context held by this object to ensure that any buffered data will not be sent when the destructor is called.

```
void Detach( );
```

See Also

Reference

CHttpResponse Class

Other Resources

CHttpResponse Members

CHttpResponse::Flush

Call this method to empty the response buffer and send its current contents to the client.

```
BOOL Flush(
   BOOL bFinal = FALSE
) throw( );
```

Parameters

bFinal

TRUE if this is the final time **Flush** will be called.

Return Value

Returns TRUE on success, FALSE on failure. Call GetLastError to get extended error information.

Remarks

Any headers that have been set in the response will be sent just before the data is written to the client if no headers have been sent up to that point.

If *bFinal* is TRUE, if the response is buffered, and if the client is communicating using HTTP 1.1, **Flush** will add a Connection: Keep-Alive header and a Content-Length header to the response.

To empty the response buffer *without* sending the current contents to the client, call CHttpResponse::ClearContent or CHttpResponse::ClearResponse.

Example

See the following samples:

- ConfirmUser Sample
- MantaWeb Sample
- ShowErrors Sample
- ShowImage Sample

See Also

Reference

CHttpResponse Class
CHttpResponse::ClearContent
CHttpResponse::ClearResponse

Other Resources

CHttpResponse Members

CHttpResponse::FlushStream

Call this method to empty the response buffer and send its current contents to the client.

```
HRESULT FlushStream( );
```

Return Value

Returns S_OK on success or an error HRESULT on failure.

Remarks

Calls CHttpResponse::Flush and returns extended error information on failure.

See Also

Reference

CHttpResponse Class

CHttpResponse::Flush

Other Resources

CHttpResponse Members

CHttpResponse::GetBufferLimit

Returns the current size limit of the response buffer in bytes.

```
DWORD GetBufferLimit( ) throw( );
```

Return Value

Returns the current size limit of the response buffer in bytes.

Remarks

See CHttpResponse::SetBufferLimit.

See Also

Reference

CHttpResponse Class

CHttpResponse::SetBufferLimit

Other Resources

CHttpResponse Members

CHttpResponse::GetBufferOutput

Call this method to determine whether data written to the response object is being buffered or not.

```
BOOL GetBufferOutput( ) throw( );
```

Return Value

Returns TRUE if output is being buffered, FALSE otherwise.

See Also

Reference

CHttpResponse Class

CHttpResponse::SetBufferOutput

Other Resources

CHttpResponse Members

CHttpResponse::GetContentType

Call this method to get the content type from an HTTP response object.

```
LPCSTR GetContentType( ) throw( );
```

Return Value

Returns the current value of the Content-Type header if present, otherwise returns NULL.

See Also

Reference

CHttpResponse Class

Other Resources

CHttpResponse Members

CHttpResponse::GetServerContext

Call this method to get the server context from an HTTP response object.

```
HRESULT GetServerContext(
   IHttpServerContext** ppOut
) throw();
```

Parameters

ppOut

[out] Address of the pointer variable that, on success, receives the IHttpServerContext interface for the current request.

Return Value

Returns S_OK on success or an error HRESULT on failure.

Example

See the ShowErrors Sample and the ShowImage Sample.

See Also

Reference

CHttpResponse Class

IHttpServerContext Interface

Other Resources

CHttpResponse Members

CHttpResponse::GetStatusCode

Call this method to get the status code from an HTTP response object.

```
int GetStatusCode( ) throw( );
```

Return Value

Returns the current HTTP status code of the response.

See Also

Reference

CHttpResponse Class

CHttpResponse::SetStatusCode

Other Resources

CHttpResponse Members

CHttpResponse::GetWriteToClient

Call this method to determine whether or not the response body will be sent to the client.

```
BOOL GetWriteToClient( ) throw( );
```

Return Value

Returns TRUE if the response body will be sent to the client, FALSE otherwise.

Remarks

If **GetWriteToClient** returns FALSE, none of the methods that write data to the client will send the response body (headers will still be sent). This method, along with CHttpResponse::SetWriteToClient, is intended primarily to allow for HEAD requests.

See Also

Reference

CHttpResponse Class

CHttpResponse::SetWriteToClient

Other Resources

CHttpResponse Members

CHttpResponse::HaveSentHeaders

Call this method to find out or specify whether response headers have been sent to the client.

```
BOOL HaveSentHeaders( ) throw( );
void HaveSentHeaders(
   BOOL bSent
) throw( );
```

Parameters

bSent

A Boolean value indicating whether or not response headers have been sent.

Return Value

Returns TRUE if headers have been sent to the client, FALSE on failure.

Remarks

The first overload returns the current information about whether headers have been sent. The second overload allows you to set that information. Setting the value simply determines whether or not the object will attempt to send headers next time it sends data to the client.

See Also

Reference

CHttpResponse Class

Other Resources

CHttpResponse Members

CHttpResponse::Initialize

Call this method to initialize an HTTP response object.

```
BOOL Initialize(
   IHttpServerContext* pServerContext
) throw( );
BOOL Initialize(
   IHttpRequestLookup * pLookup
) throw(...);
```

Parameters

pServerContext

The server context.

pLookup

The pointer to the lookup interface used to provide the server context.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

When the constructor overload with the **IHttpRequestLookup*** parameter is used, the response object is initialized in such a way as to prevent HTTP headers from being sent with the response.

See Also

Reference

CHttpResponse Class

Other Resources

CHttpResponse Members

CHttpResponse::Redirect

Call this method to redirect the client to a different resource.

```
BOOL Redirect(
   LPCSTR szUrl,
   LPCSTR szBody,
   HTTP_REDIRECT statusCode = HTTP_REDIRECT_MOVED
) throw( );
BOOL Redirect(
   LPCSTR szUrl,
   HTTP_REDIRECT statusCode = HTTP_REDIRECT_MOVED,
   BOOL bSendBody = TRUE
) throw(...);
```

Parameters

szUrl

A null-terminated string specifying the resource the client should navigate to.

szBody

A null-terminated string containing the body of the response to be sent to the client.

statusCode

An HTTP status code from the CHttpResponse::HTTP_REDIRECT enumeration describing the reason for the redirection.

bSendBody

TRUE if a response body should be automatically created and sent with the response. FALSE if no body should be sent.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Use the first overload of this method if you want to supply your own body with the redirect response. Use the second overload if you want to have a response body automatically generated for you.

The automatically generated response body contains a short HTML note with a hyperlink to the new resource and a META refresh tag to allow some browsers to automatically redirect the user to the resource even if they don't understand the redirect header.

Note that, in compliance with RFC 2616, a response body will only be generated with the following redirect types:

- **HTTP_REDIRECT_MOVED**
- **HTTP_REDIRECT_FOUND**
- **HTTP_REDIRECT_SEE_OTHER**
- **HTTP_REDIRECT_TEMPORARY_REDIRECT**

No body will be generated with other redirect types.

See RFC 2616 section 10.3 (http://ietf.org/rfc/rfc2616.txt) for more information on redirection.

Example

See the following samples:

- MantaWeb Sample
- ForceLogin Sample
- SessionSettings Sample

See Also

Reference

CHttpResponse Class

CHttpResponse::HTTP_REDIRECT

Other Resources

CHttpResponse Members

CHttpResponse::RenderHeaders

Call this method to get a string containing all of the HTTP headers associated with this object in a format suitable for sending to a client.

```
void RenderHeaders(
   CStringA& strHeaders
) throw( );
```

Parameters

strHeaders

A string reference to which will be appended the HTTP headers.

See Also

Reference

CHttpResponse Class

Other Resources

CHttpResponse Members

CHttpResponse::SendHeadersInternal

Call this method to send the current headers associated with this object to the client.

```
BOOL SendHeadersInternal(
   BOOL fKeepConn = FALSE
) throw( );
```

Parameters

fKeepConn

TRUE to keep the connection to the client open, FALSE otherwise.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The response headers are sent to the client using the current status code for the response object. See CHttpResponse::SetStatusCode and CHttpResponse::GetStatusCode.

Example

See the MantaWeb Sample.

See Also

Reference

CHttpResponse Class
CHttpResponse::SetStatusCode
CHttpResponse::GetStatusCode

Other Resources

CHttpResponse Members

CHttpResponse::SetBufferLimit

Call this method to set a size limit on the amount of data buffered by the response object.

```
void SetBufferLimit(
   DWORD dwBufferLimit
) throw( );
```

Parameters

dwBufferLimit

The buffer limit.

Remarks

When the size of the buffer is reduced below the current size of the buffered content, the entire buffer is flushed. See CHttpResponse::GetBufferLimit.

See Also

Reference

CHttpResponse Class

CHttpResponse::GetBufferLimit

Other Resources

CHttpResponse Members

CHttpResponse::SetBufferOutput

Call this method to set buffering options for the response.

```
void SetBufferOutput(
   BOOL bBufferOutput,
   DWORD dwBufferLimit = ATL_ISAPI_BUFFER_SIZE
) throw( );
```

Parameters

bBufferOutput

TRUE if buffering should be enabled, FALSE otherwise.

dwBufferLimit

The maximum size in bytes of the buffer.

Remarks

This method allows you to turn buffering on or off and to set a size limit on the amount of data that will be buffered before being sent to the client. When you turn off buffering, the current contents of the buffer will be sent to the client. When the buffer limit is reduced below the current size of the buffered content, the entire buffer is flushed.

See Also

Reference

CHttpResponse Class
CHttpResponse::GetBufferOutput
ATL_ISAPI_BUFFER_SIZE

Other Resources

CHttpResponse Members

CHttpResponse::SetCacheControl

Call this method to set the Cache-Control HTTP header of the response.

```
BOOL SetCacheControl(
   LPCSTR szCacheControl
) throw( );
```

Parameters

szCacheControl

The value of the Cache-Control header.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Examples of common Cache-Control header values:

- public
- private
- max-age=*delta-seconds*

Example

See the following samples:

- ConfirmUser Sample
- ShowErrors Sample
- ShowImage Sample

See Also

Reference

CHttpResponse Class

Other Resources

CHttpResponse Members

CHttpResponse::SetContentType

Call this method to set the content type of the HTTP response.

```
BOOL SetContentType(
   LPCSTR szContentType
) throw( );
```

Parameters

szContentType

The MIME content type.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Call this method to set the Content-Type header of the HTTP response. Examples of common MIME content types include text/html and text/plain.

Example

See the following samples:

- ConfirmUser Sample
- MantaWeb Sample
- NotModified Sample
- ShowImage Sample
- SRFSyntax Sample
- StencilCache Sample

See Also

Reference

CHttpResponse Class

Other Resources

CHttpResponse Members

CHttpResponse::SetExpires

Call this method to set the Expires header of the HTTP response.

```
BOOL SetExpires(
   long lMinutes
) throw( );
```

Parameters

lMinutes

The number of minutes (from the time at which this method is called) until the response is no longer valid.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Call this method to set the Expires HTTP header to a relative date/time value specified in minutes. To set the expiration time to an absolute GMT time, use CHttpResponse::SetExpiresAbsolute.

Example

See the following samples:

- ConfirmUser Sample
- ShowErrors Sample
- ShowImage Sample

See Also

Reference

CHttpResponse Class

CHttpResponse::SetExpiresAbsolute

Other Resources

CHttpResponse Members

CHttpResponse::SetExpiresAbsolute

Call this method to set the expiration time of the HTTP response.

```
BOOL SetExpiresAbsolute(
   const SYSTEMTIME& stExpires
) throw( );
```

Parameters

stExpires

The GMT date/time at which the HTTP response is no longer valid.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Call this method to set the Expires HTTP header to the absolute date/time specified in the *stExpires* parameter. To set the expiration time as a number of minutes relative to the current time, use CHttpResponse::SetExpires.

See Also

Reference

CHttpResponse Class

CHttpResponse::SetExpires

Other Resources

CHttpResponse Members

CHttpResponse::SetStatusCode

Call this method to set the HTTP status code of the response.

```
void SetStatusCode(
   int nCode
) throw( );
```

Parameters

nCode

The HTTP status code. See RFC 2616 (http://ietf.org/rfc/rfc2616.txt) for definitions of the various status codes.

Remarks

Call this method to set the HTTP status code of the response. If not set explicitly, the default status code is 200 (OK).

Example

See the following samples:

- ConfirmUser Sample
- ForceLogin Sample
- ShowErrors Sample

See Also

Reference

CHttpResponse Class
CHttpResponse::GetStatusCode

Other Resources

CHttpResponse Members

CHttpResponse::SetWriteToClient

Call this method to specify whether a response body will be sent to the client.

```
BOOL SetWriteToClient(
   BOOL bSendOutput
) throw( );
```

Parameters

bSendOutput

TRUE if the response body should be sent, FALSE otherwise.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

See CHttpResponse::GetWriteToClient for more details.

See Also

Reference

CHttpResponse Class

CHttpResponse::GetWriteToClient

Other Resources

CHttpResponse Members

CHttpResponse::TransmitFile

Call this method to transmit a file as the HTTP response.

```
BOOL TransmitFile(
   HANDLE hFile,
   LPCSTR szContentType = "text/plain"
) throw( );
```

Parameters

hFile

The file handle of the file to be transmitted.

szContentType

The MIME content type of the file.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This method transmits the file asynchronously. The file handle should be to a file opened with the FILE_FLAG_SEQUENTIAL_SCAN and FILE_FLAG_OVERLAPPED flags.

See Also

Reference

CHttpResponse Class

Other Resources

CHttpResponse Members

CHttpResponse::WriteLen

Call this method to write data to the body of the HTTP response.

```
BOOL WriteLen(
   LPCSTR szOut,
   DWORD dwLen
) throw( );
```

Parameters

szOut

A pointer to the first byte of the data to be written.

dwLen

The number of bytes to write.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

If buffering is disabled, data is written directly to the client. If buffering is enabled, this method attempts to write to the buffer. If the buffer is too small to contain its existing data and the new data, the current contents of the buffer are flushed. If the buffer is still too small to contain the new data, that data is written directly to the client. Otherwise the new data is written to the buffer.

Any headers that have been set in the response will be sent just before the data is written to the client if no headers have been sent up to that point.

Example

See the ShowImage Sample.

See Also

Reference

CHttpResponse Class

Other Resources

CHttpResponse Members

CHttpResponse::WriteStream

Call this method to write data to the response object.

```
HRESULT WriteStream(
   LPCSTR szOut,
   int nLen,
   DWORD* pdwWritten
);
```

Parameters

szOut

A pointer to the first byte of the data to be written.

nLen

The number of bytes to write. If this parameter is -1, *szOut* is assumed to be a null-terminated string and the whole string will be written.

pdwWritten

A DWORD pointer that can be used to get the number of bytes written. This parameter can be NULL.

Return Value

Returns S_OK on success or an error HRESULT on failure.

Remarks

See CHttpResponse::WriteLen for comments on buffering.

See Also

Reference

CHttpResponse Class

CHttpResponse::WriteLen

Other Resources

CHttpResponse Members

CHttpResponse Data Members

For information about the data members in **CHttpResponse**, see CHttpResponse Members.

CHttpResponse::m_strContent

This data member holds the buffered content.

```
CAtlIsapiBuffer< > m_strContent;
```

See Also

Reference

CHttpResponse Class

CAtlIsapiBuffer Class

Other Resources

CHttpResponse Members

CHttpResponse Enums

For information about the enums in **CHttpResponse**, see CHttpResponse Members.

CHttpResponse::HTTP_REDIRECT

This enumeration defines the values for the HTTP status codes used to redirect a client to another resource.

```
enum HTTP_REDIRECT
{
   HTTP_REDIRECT_MULTIPLE = 300,
   HTTP_REDIRECT_MOVED = 301,
   HTTP_REDIRECT_FOUND = 302,
   HTTP_REDIRECT_SEE_OTHER = 303,
   HTTP_REDIRECT_NOT_MODIFIED = 304,
   HTTP_REDIRECT_USE_PROXY = 305,
   HTTP_REDIRECT_TEMPORARY_REDIRECT = 307
};
```

Remarks

See the HTTP 1.1 RFC (http://ietf.org/rfc/rfc2616.txt) for a description of the different HTTP status codes identified by these enumeration values.

See Also

Reference

CHttpResponse Class

Other Resources

CHttpResponse Members

CIDServerContext Class

This class provides an implementation of IHttpServerContext for use by a subhandler.

```
class CIDServerContext :
   public CComObjectRootEx<CComMultiThreadModel>,
   public CWrappedServerContext
```

Remarks

The implementation of **IHttpServerContext** provided by this class delegates most methods to the request object passed to CIDServerContext::Initialize (or to the **IHttpServerContext** pointer obtained from it).

Requirements

Header: atlstencil.h

See Also

Tasks

MantaWeb Sample: Online Personal Information Management

Reference

ForceLogin

ShowErrors

IHttpServerContext Interface

Other Resources

CIDServerContext Members

CIDServerContext Members

Methods

CIDServerContext	The constructor.
GetAvailableBytes	This implementation of IHttpServerContext::GetAvailableBytes obtains the information from m_pRequest.
GetAvailableData	This implementation of IHttpServerContext::GetAvailableData obtains the information from m_pRequest.
GetContentType	This implementation of IHttpServerContext::GetContentType obtains the information from m_pRequest.
GetPathInfo	This implementation of IHttpServerContext::GetPathInfo obtains the information from m_pRequest.
GetPathTranslated	This implementation of IHttpServerContext::GetPathTranslated obtains the information from m_pRequest.
GetQueryString	This implementation of IHttpServerContext::GetQueryString obtains the information from m_pRequest.
GetRequestMethod	This implementation of IHttpServerContext::GetRequestMethod obtains the information from m_pRequest.
GetScriptPathTranslated	This implementation of IHttpServerContext::GetScriptPathTranslated obtains the information from m_pRequest.
GetTotalBytes	This implementation of IHttpServerContext::GetTotalBytes obtains the information from m_pRequest.
Initialize	Call this method to initialize the object before calling any other methods.
ReadClient	This implementation of IHttpServerContext::ReadClient obtains the data from m_pRequest.
SendRedirectResponse	This implementation of IHttpServerContext::SendRedirectResponse writes to m_pResponse.
TransmitFile	This implementation of IHttpServerContext::TransmitFile writes to m_pResponse.
WriteClient	This implementation of IHttpServerContext::WriteClient writes to m_pResponse.

Data Members

m_pRequest	The object containing information about the current request.
m_pResponse	The object used to return the response to the client.

See Also

Reference

CIDServerContext Class

Other Resources

IHttpServerContext Members

CIDServerContext Methods

For information about the methods in **CIDServerContext**, see CIDServerContext Members.

CIDServerContext::CIDServerContext

The constructor.

```
CIDServerContext( ) throw( );
```

Remarks

Initializes internal data structures. This object must be initialized by a call to Initialize before any other methods are called.

See Also

Reference

CIDServerContext Class

CIDServerContext::Initialize

Other Resources

CIDServerContext Members

CIDServerContext::GetAvailableBytes

This implementation of IHttpServerContext::GetAvailableBytes obtains the information from m_pRequest.

```
DWORD GetAvailableBytes( );
```

See Also

Reference

CIDServerContext Class

Other Resources

CIDServerContext Members

CIDServerContext::GetAvailableData

This implementation of IHttpServerContext::GetAvailableData obtains the information from m_pRequest.

```
BYTE* GetAvailableData( );
```

See Also

Reference

CIDServerContext Class

Other Resources

CIDServerContext Members

CIDServerContext::GetContentType

This implementation of IHttpServerContext::GetContentType obtains the information from m_pRequest.

```
LPCSTR GetContentType( );
```

See Also

Reference

CIDServerContext Class

Other Resources

CIDServerContext Members

CIDServerContext::GetPathInfo

This implementation of IHttpServerContext::GetPathInfo obtains the information from m_pRequest.

```
LPCSTR GetPathInfo( );
```

See Also

Reference

CIDServerContext Class

Other Resources

CIDServerContext Members

CIDServerContext::GetPathTranslated

This implementation of IHttpServerContext::GetPathTranslated obtains the information from m_pRequest.

```
LPCSTR GetPathTranslated( );
```

See Also

Reference

CIDServerContext Class

Other Resources

CIDServerContext Members

CIDServerContext::GetQueryString

This implementation of IHttpServerContext::GetQueryString obtains the information from m_pRequest.

```
LPCSTR GetQueryString( );
```

See Also

Reference

CIDServerContext Class

Other Resources

CIDServerContext Members

CIDServerContext::GetRequestMethod

This implementation of IHttpServerContext::GetRequestMethod obtains the information from m_pRequest.

```
LPCSTR GetRequestMethod( );
```

See Also

Reference

CIDServerContext Class

Other Resources

CIDServerContext Members

CIDServerContext::GetScriptPathTranslated

This implementation of IHttpServerContext::GetScriptPathTranslated obtains the information from m_pRequest.

```
LPCSTR GetScriptPathTranslated( );
```

Example

See the ShowErrors Sample.

See Also

Reference

CIDServerContext Class

Other Resources

CIDServerContext Members

CIDServerContext::GetTotalBytes

This implementation of IHttpServerContext::GetTotalBytes obtains the information from m_pRequest.

```
DWORD GetTotalBytes( );
```

Example

See the ForceLogin Sample.

See Also

Reference

CIDServerContext Class

Other Resources

CIDServerContext Members

CIDServerContext::Initialize

Call this method to initialize the object before calling any other methods.

```
BOOL Initialize(
   CHttpResponse* pResponse,
   CHttpRequest* pRequest
) throw( );
```

Parameters

pResponse

The object used to return the response to the client.

pRequest

The object containing information about the current request.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

Call this method to initialize the object before calling any of the IHttpServerContext methods.

See Also

Reference

CIDServerContext Class

Other Resources

CIDServerContext Members

CIDServerContext::ReadClient

This implementation of IHttpServerContext::ReadClient obtains the data from m_pRequest.

```
BOOL ReadClient(
   void* pvBuffer,
   DWORD* pdwSize
);
```

See Also

Reference

CIDServerContext Class

Other Resources

CIDServerContext Members

CIDServerContext::SendRedirectResponse

This implementation of IHttpServerContext::SendRedirectResponse writes to m_pResponse.

```
BOOL SendRedirectResponse(
   LPCSTR pszRedirectURL
);
```

See Also

Reference

CIDServerContext Class

Other Resources

CIDServerContext Members

CIDServerContext::TransmitFile

This implementation of IHttpServerContext::TransmitFile writes to m_pResponse.

```
BOOL TransmitFile(
   HANDLE hFile,
   PFN_HSE_IO_COMPLETION pfn,
   void* pContext,
   LPCSTR szStatusCode,
   DWORD dwBytesToWrite,
   DWORD dwOffset,
   void* pvHead,
   DWORD dwHeadLen,
   void* pvTail,
   DWORD dwTailLen,
   DWORD dwFlags
);
```

See Also

Reference

CIDServerContext Class

Other Resources

CIDServerContext Members

CIDServerContext::WriteClient

This implementation of IHttpServerContext::WriteClient writes to m_pResponse.

```
BOOL WriteClient(
   void* pvBuffer,
   DWORD* pdwBytes
);
```

Example

See the MantaWeb Sample and the ShowErrors Sample.

See Also

Reference

CIDServerContext Class

Other Resources

CIDServerContext Members

CIDServerContext Data Members

For information about the data members in **CIDServerContext**, see CIDServerContext Members.

CIDServerContext::m_pRequest

The object containing information about the current request.

```
CHttpRequest* m_pRequest;
```

Remarks

This member is set by a call to Initialize.

See Also

Reference

CIDServerContext Class

CIDServerContext::m_pResponse

Other Resources

CIDServerContext Members

CIDServerContext::m_pResponse

The object used to return the response to the client.

```
CHttpResponse* m_pResponse;
```

Remarks

This member is set by a call to Initialize.

See Also

Reference

CIDServerContext Class

CIDServerContext::m_pRequest

Other Resources

CIDServerContext Members

CIncludeServerContext Class

This class provides an implementation of IHttpServerContext for use by included request handlers.

```
class CIncludeServerContext :
   public CComObjectRootEx<CComMultiThreadModel>,
   public CWrappedServerContext
```

Remarks

The implementation of IHttpServerContext provided by this class delegates most methods to the IHttpServerContext pointer passed to CIncludeServerContext::Initialize. However, non-functional implementations of a number of methods are provided to prevent strange interactions between multiple handlers processing the same request.

URL path and query string information is parsed from the content of the include tag and passed to Initialize to be returned by the implementations of GetPathTranslated, GetScriptPathTranslated, and GetQueryString.

The members list of this class provides a summary of the operations that cannot be performed from an included request handler or that return different results when executed from an included request handler.

Requirements

Header: atlstencil.h

See Also

Tasks

MantaWeb Sample: Online Personal Information Management

Reference

ForceLogin
ShowErrors
CComObjectRootEx Class
IHttpServerContext Interface
CWrappedServerContext Class

Other Resources

CIncludeServerContext Members

CIncludeServerContext Members

Methods

AsyncReadClient	Non-functional implementation of IHttpServerContext::AsyncReadClient. Always returns **FALSE**.
CIncludeServerContext	The constructor.
DoneWithSession	Non-functional implementation of IHttpServerContext::DoneWithSession. Always returns **FALSE**.
GetAvailableBytes	Non-functional implementation of IHttpServerContext::GetAvailableBytes. Always returns 0.
GetAvailableData	Non-functional implementation of IHttpServerContext::GetAvailableData. Always returns NULL.
GetContentType	Non-functional implementation of IHttpServerContext::GetContentType. Always returns NULL.
GetPathTranslated	Implementation of IHttpServerContext::GetPathTranslated. Returns the path obtained from the data passed to CIncludeServerContext::Initialize.
GetQueryString	Implementation of IHttpServerContext::GetQueryString. Returns the query string obtained from the data passed to CIncludeServerContext::Initialize.
GetRequestMethod	Implementation of IHttpServerContext::GetRequestMethod. Always returns "GET".
GetScriptPathTranslated	Implementation of IHttpServerContext::GetScriptPathTranslated. Returns the path obtained from the data passed to CIncludeServerContext::Initialize.
GetTotalBytes	Non-functional implementation of IHttpServerContext::GetTotalBytes. Always returns 0.
Initialize	This method is called by the ATL Server framework to initialize the object.
ReadClient	Non-functional implementation of IHttpServerContext::ReadClient. Always returns **FALSE**.
RequestIOCompletion	Non-functional implementation of IHttpServerContext::RequestIOCompletion. Always returns **FALSE**.
SendRedirectResponse	Non-functional implementation of IHttpServerContext::SendRedirectResponse. Always returns **FALSE**.
SendResponseHeader	Non-functional implementation of IHttpServerContext::SendResponseHeader. Always returns **FALSE**.
WriteClient	Implementation of IHttpServerContext::WriteClient. Writes the data to the stream passed to CIncludeServerContext::Initialize.

See Also

Reference

CIncludeServerContext Class

Other Resources

CComObjectRootEx Members

CWrappedServerContext Members

CIncludeServerContext Methods

For information about the methods in **CIncludeServerContext**, see CIncludeServerContext Members.

CIncludeServerContext::AsyncReadClient

Non-functional implementation of IHttpServerContext::AsyncReadClient.

```
BOOL AsyncReadClient(
   void * /* pvBuffer */,
   DWORD * /* pdwSize */
);
```

Return Value

Always returns **FALSE**.

See Also

Reference

CIncludeServerContext Class

Other Resources

CIncludeServerContext Members

CIncludeServerContext::CIncludeServerContext

The constructor.

```
CIncludeServerContext( ) throw( );
```

Remarks

Initializes internal data structures.

See Also

Reference

CIncludeServerContext Class

Other Resources

CIncludeServerContext Members

CIncludeServerContext::DoneWithSession

Non-functional implementation of IHttpServerContext::DoneWithSession.

```
BOOL DoneWithSession(
   DWORD /* dwHttpStatusCode */
);
```

Return Value

Always returns **FALSE**.

See Also

Reference

CIncludeServerContext Class

Other Resources

CIncludeServerContext Members

CIncludeServerContext::GetAvailableBytes

Non-functional implementation of IHttpServerContext::GetAvailableBytes.

```
DWORD GetAvailableBytes( );
```

Return Value

Always returns 0.

See Also

Reference

CIncludeServerContext Class

Other Resources

CIncludeServerContext Members

CIncludeServerContext::GetAvailableData

Non-functional implementation of IHttpServerContext::GetAvailableData.

```
BYTE* GetAvailableData( );
```

Return Value

Always returns NULL.

See Also

Reference

CIncludeServerContext Class

Other Resources

CIncludeServerContext Members

CIncludeServerContext::GetContentType

Non-functional implementation of IHttpServerContext::GetContentType.

```
LPCSTR GetContentType( );
```

Return Value

Always returns NULL.

See Also

Reference

CIncludeServerContext Class

Other Resources

CIncludeServerContext Members

CIncludeServerContext::GetPathTranslated

Implementation of IHttpServerContext::GetPathTranslated. Returns the path obtained from the data passed to CIncludeServerContext::Initialize.

```
LPCSTR GetPathTranslated( );
```

See Also

Reference

CIncludeServerContext Class

Other Resources

CIncludeServerContext Members

CIncludeServerContext::GetQueryString

Implementation of IHttpServerContext::GetQueryString. Returns the query string obtained from the data passed to CIncludeServerContext::Initialize.

```
LPCSTR GetQueryString( );
```

See Also

Reference

CIncludeServerContext Class

Other Resources

CIncludeServerContext Members

CIncludeServerContext::GetRequestMethod

Implementation of IHttpServerContext::GetRequestMethod.

```
LPCSTR GetRequestMethod( );
```

Return Value

Always returns "GET".

See Also

Reference

CIncludeServerContext Class

Other Resources

CIncludeServerContext Members

CIncludeServerContext::GetScriptPathTranslated

Implementation of IHttpServerContext::GetScriptPathTranslated. Returns the path obtained from the data passed to CIncludeServerContext::Initialize.

```
LPCSTR GetScriptPathTranslated( );
```

Example

See the ShowErrors Sample.

See Also

Reference

CIncludeServerContext Class

Other Resources

CIncludeServerContext Members

CIncludeServerContext::GetTotalBytes

Non-functional implementation of IHttpServerContext::GetTotalBytes.

```
DWORD GetTotalBytes( );
```

Return Value

Always returns 0.

Example

See the ForceLogin Sample.

See Also

Reference

CIncludeServerContext Class

Other Resources

CIncludeServerContext Members

CIncludeServerContext::Initialize

This method is called by the ATL Server framework to initialize the object.

```
void Initialize(
   IWriteStream* pStream,
   IHttpServerContext* pServerContext,
   const StencilIncludeInfo* pIncludeInfo
) throw();
void Initialize(
   CIncludeServerContext* pOtherContext
);
```

Parameters

pStream

The stream to which the response should be written.

pServerContext

The server context for the current request.

pIncludeInfo

The query string and path information for the included file.

pOtherContext

The context for the including file.

Remarks

This method must be called to initialize the object before any of the IHttpServerContext methods may be called.

See Also

Reference

CIncludeServerContext Class

Other Resources

CIncludeServerContext Members

CIncludeServerContext::ReadClient

Non-functional implementation of IHttpServerContext::ReadClient.

```
BOOL ReadClient(
   void * /* pvBuffer */,
   DWORD * /* pdwSize */
);
```

Return Value

Always returns **FALSE**.

See Also

Reference

CIncludeServerContext Class

Other Resources

CIncludeServerContext Members

CIncludeServerContext::RequestIOCompletion

Non-functional implementation of IHttpServerContext::RequestIOCompletion.

```
BOOL RequestIOCompletion(
   PFN_HSE_IO_COMPLETION /* pfn */,
   DWORD * /* pdwContext */
);
```

Return Value

Always returns **FALSE**.

See Also

Reference

CIncludeServerContext Class

Other Resources

CIncludeServerContext Members

CIncludeServerContext::SendRedirectResponse

Non-functional implementation of IHttpServerContext::SendRedirectResponse.

```
BOOL SendRedirectResponse(
   LPCSTR /* pszRedirectURL */
);
```

Return Value

Always returns **FALSE**.

See Also

Reference

CIncludeServerContext Class

Other Resources

CIncludeServerContext Members

CIncludeServerContext::SendResponseHeader

Non-functional implementation of IHttpServerContext::SendResponseHeader.

```
BOOL SendResponseHeader(
   LPCSTR /* pszHeader */,
   LPCSTR /* pszStatusCode */,
   BOOL /* fKeepConn */
);
```

Return Value

Always returns **FALSE**.

Example

See the ShowErrors Sample.

See Also

Reference

CIncludeServerContext Class

Other Resources

CIncludeServerContext Members

CIncludeServerContext::WriteClient

Implementation of IHttpServerContext::WriteClient. Writes the data to the stream passed to CIncludeServerContext::Initialize.

```
BOOL WriteClient(
   void* pvBuffer,
   DWORD* pdwBytes
);
```

Example

See the MantaWeb Sample and the ShowErrors Sample.

See Also

Reference

CIncludeServerContext Class

Other Resources

CIncludeServerContext Members

CIsapiExtension Class

This class provides the code necessary for implementing the ISAPI extension DLL that ATL Server uses to pass requests from IIS to your Web application DLLs.

```
template <
   class ThreadPoolClass           = CThreadPool<CIsapiWorker>,
   class CRequestStatClass         = CNoRequestStats,
   class HttpUserErrorTextProvider = CDefaultErrorProvider,
   class WorkerThreadTraits        = DefaultThreadTraits,
   class CPageCacheStats           = CNoStatClass,
   class CStencilCacheStats        = CNoStatClass
>
class CIsapiExtension :
   public IServiceProvider,
   public IIsapiExtension,
   public IRequestStats
```

Parameters

ThreadPoolClass

The class used to provide the thread pool on which the ISAPI extension will queue HTTP requests. See CThreadPool for details of the interface that should be provided by this class.

If necessary, change this parameter to use a different thread pool or, more commonly, change it to use a different worker thread class derived from CIsapiWorker.

Request processing code can access a pointer to the worker thread class, which allows the request handling code to easily access per-thread data. The pointer to the thread worker can be obtained by calling IIsapiExtension::GetThreadWorker and data or services provided by the derived class can be obtained by calling CIsapiWorker::GetWorkerData.

CRequestStatClass

The class used to keep track of request-handling statistics. Must conform to the request statistics archetype.

The statistics managed by this class are available programmatically from the IRequestStats interface which may be obtained from the ISAPI extension by calling QueryInterface.

CIsapiExtension has an object of this class as a member named m_reqStats.

HttpUserErrorTextProvider

The class used to provide error responses in the case of failure. This class provides the response headers as text and, optionally, the response bodies as resource IDs to be loaded by ATL Server code.

If necessary, change this parameter to provide your own response headers or bodies to be used when errors are encountered during request processing.

See CDefaultErrorProvider for information about the interface that must be provided by this class.

CIsapiExtension has an object of this class as a member named m_UserErrorProvider.

WorkerThreadTraits

The class used to create the worker thread used by the page, stencil, and DLL caches. See DefaultThreadTraits and m_WorkerThread.

CPageCacheStats, CStencilCacheStats

The classes used to keep track of cache-hit statistics for the page and stencil caches. Must conform to the cache statistics archetype.

The statistics managed by *CPageCacheStats* are available programmatically from the IMemoryCacheStats interface which may be obtained by casting m_PageCache.

The statistics managed by *CStencilCacheStats* are available programmatically from the IMemoryCacheStats interface which may be obtained from the ISAPI extension by first calling QueryService to obtain the IStencilCache interface and then calling

QueryInterface on that interface.

Remarks

This class provides the code necessary for implementing the ISAPI extension DLL that ATL Server uses to pass requests from IIS to your Web application DLLs. The class queues requests on to the thread pool (which frees IIS threads to receive other requests and improves scalability), provides code for dispatching requests to the application DLLs, offers caching services and the ability to add or remove further services, and provides the ability to gather statistics related to the behavior of the application.

CIsapiExtension exposes the following interfaces through QueryInterface:

- IIsapiExtension
- IRequestStats
- IServiceProvider

CIsapiExtension exposes the following interfaces through QueryService:

- IDllCache
- IStencilCache
- IThreadPoolConfig
- IAtlMemMgr

Requirements

Header: atlisapi.h

Example

See the following samples:

- Hotswap Sample
- MantaWeb Sample
- PooledAsync Sample
- SOAPState Sample

See Also

Reference

IServiceProvider
IIsapiExtension Interface
IRequestStats Interface

Other Resources

CIsapiExtension Members
ATL Server Classes

CIsapiExtension Members

Methods

AddRef	This implementation of **IUnknown::AddRef** always returns 1.
AddService	Implementation of IIsapiExtension::AddService.
CIsapiExtension	The constructor.
CreateRequest	Implementation of IIsapiExtension::CreateRequest.
DispatchStencilCall	Implementation of IIsapiExtension::DispatchStencilCall.
FreeRequest	Implementation of IIsapiExtension::FreeRequest.
GetActiveThreads	Implementation of IRequestStats::GetActiveThreads.
GetAvgResponseTime	Implementation of IRequestStats::GetAvgResponseTime.
GetCacheServerContext	Call this method to obtain a server context object that can be used to cache a response. Override this method to control the way in which cached responses are generated.
GetCriticalIsapiError	Call this method to get the value of a critical ISAPI error set by a previous call to SetCriticalIsapiError.
GetCurrWaiting	Implementation of IRequestStats::GetCurrWaiting.
GetDllCacheTimeout	Override this method to return the timeout value in milliseconds used by the DLL cache.
GetExtensionDesc	Override this method to return the ISAPI extension description used by IIS for logging.
GetExtensionVersion	Call this method from your ISAPI DLL's GetExtensionVersion entry point to initialize this object.
GetFailedRequests	Implementation of IRequestStats::GetFailedRequests.
GetHandlerName	Call this method to get the name of the primary handler from a server response file.
GetIOCompletionHandle	Override this method to return a handle if you want to initialize the thread pool to associate its IO co mpletion port with the object associated with that handle.
GetMaxWaiting	Implementation of IRequestStats::GetMaxWaiting.
GetNumPoolThreads	Override this method to return the number of threads with which to start the thread pool.
GetPoolStackSize	Override this method to return the stack size to be used for each thread in the pool.
GetService	Call this method to get an interface pointer on a dynamic service supported by the ISAPI extension.
GetStencilCacheTimeout	Override this method to return the timeout value in milliseconds used by the stencil cache.
GetStencilLifespan	Override this method to return the default life span in milliseconds of a stencil to be used by the sten cil cache.

GetThreadWorker	Implementation of IIsapiExtension::GetThreadWorker.
GetTotalRequests	Implementation of IRequestStats::GetTotalRequests.
HandleError	Call this method to send an error response to the client.
HttpExtensionProc	Call this method from your ISAPI DLL's HttpExtensionProc entry point to queue HTTP requests on the thread pool.
LoadDispatchFile	Call this method to populate the AtlServerRequest structure with the request handler interface pointe r and DLL instance handle for the handler specified in a server response file.
LoadDllHandler	Call this method to load the request handler for the current request based on the DLL file name and t he server context.
LoadRequestHandler	Call this method to retrieve the request handler interface pointer and DLL instance handle for the spe cified handler.
OnThreadAttach	Override this method to execute code when a worker thread is initialized.
OnThreadTerminate	Override this method to execute code when a worker thread is about to be destroyed.
QueryInterface	Implementation of **IUnknown::QueryInterface**.
QueryService	Call this method to get a service from the ISAPI extension or override this method to support new ser vices.
QueueRequest	Call this method to queue a request on the thread pool.
Release	This implementation of **IUnknown::Release** always returns 1.
RemoveService	Implementation of IIsapiExtension::RemoveService.
RequestComplete	Call this method when the handling of the request is complete.
ReturnCriticalError	Call this method to return a response describing the critical ISAPI extension error.
SetCriticalIsapiError	Call this method to notify the ISAPI extension that a critical error has occurred.
SetThreadWorker	Implementation of IIsapiExtension::SetThreadWorker.
StartAsyncFlush	Call this method to start writing buffered content to the client asynchronously.
TerminateExtension	Call this method from your ISAPI DLL's TerminateExtension entry point to free the resources used by the ISAPI extension.
TransferRequest	Implementation of IIsapiExtension::TransferRequest.
TransmitFromCache	Call this method to transmit an item from the page cache. Override this method to control the way in which cached items are transmitted to the client.

Data Members

m_DllCache	The DLL cache used to speed up access to Web application DLLs.

m_PageCache	The page cache used to cache responses generated by request handlers.
m_reqStats	The object responsible for tracking request-handling statistics.
m_StencilCache	The stencil cache used to hold pre-parsed server response files and speed up rendering of their response s.
m_UserErrorProvider	The error provider.
m_WorkerThread	The worker thread used to maintain the page, stencil, and DLL caches.

Typedefs

extWorkerType	An alias for a worker thread class based on the *WorkerThreadTraits* parameter used to specialize the class tem plate.

See Also

Reference

CIsapiExtension Class

IServiceProvider Members

Other Resources

IIsapiExtension Members

IRequestStats Members

CIsapiExtension Methods

For information about the methods in **CIsapiExtension**, see CIsapiExtension Members.

CIsapiExtension::AddRef

This implementation of **IUnknown::AddRef** always returns 1.

```
ULONG STDMETHODCALLTYPE AddRef( );
```

Return Value

Returns 1.

See Also

Reference

CIsapiExtension Class

CIsapiExtension::Release

Other Resources

CIsapiExtension Members

CIsapiExtension::AddService

Implementation of IIsapiExtension::AddService.

```
virtual HRESULT AddService(
   REFGUID guidService,
   REFIID riid,
   IUnknown * punkService,
   HINSTANCE hInstance
);
```

See Also

Reference

CIsapiExtension Class

IIsapiExtension::AddService

CIsapiExtension::RemoveService

Other Resources

CIsapiExtension Members

CIsapiExtension::CIsapiExtension

The constructor.

```
CIsapiExtension( ) throw( );
```

Remarks

Initializes internal data members.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::CreateRequest

Implementation of IIsapiExtension::CreateRequest.

```
AtlServerRequest* CreateRequest( );
```

See Also

Reference

CIsapiExtension Class

IIsapiExtension::CreateRequest

Other Resources

CIsapiExtension Members

CIsapiExtension::DispatchStencilCall

Implementation of IIsapiExtension::DispatchStencilCall.

```
BOOL DispatchStencilCall(
   AtlServerRequest* pRequestInfo
);
```

Remarks

This method is called by CIsapiWorker::Execute.

Example

See the Hotswap Sample.

See Also

Reference

CIsapiExtension Class

IIsapiExtension::DispatchStencilCall

Other Resources

CIsapiExtension Members

CIsapiExtension::FreeRequest

Implementation of IIsapiExtension::FreeRequest.

```
void FreeRequest(
   AtlServerRequest* pRequest
);
```

See Also

Reference

CIsapiExtension Class

IIsapiExtension::FreeRequest

Other Resources

CIsapiExtension Members

CIsapiExtension::GetActiveThreads

Implementation of IRequestStats::GetActiveThreads.

```
long GetActiveThreads( );
```

Remarks

Delegates to m_reqStats.

See Also

Reference

CIsapiExtension Class

IRequestStats::GetActiveThreads

Other Resources

CIsapiExtension Members

CIsapiExtension::GetAvgResponseTime

Implementation of IRequestStats::GetAvgResponseTime.

```
long GetAvgResponseTime( );
```

Remarks

Delegates to m_reqStats.

See Also

Reference

CIsapiExtension Class

IRequestStats::GetAvgResponseTime

Other Resources

CIsapiExtension Members

CIsapiExtension::GetCacheServerContext

Call this method to obtain a server context object that can be used to cache a response. Override this method to control the way in which cached responses are generated.

```
virtual BOOL GetCacheServerContext(
   AtlServerRequest * pRequestInfo,
   IFileCache * pCache,
   IHttpServerContext ** pCacheCtx
);
```

Parameters

pRequestInfo

Information about the current request.

pCache

The implementation of IFileCache which should be used to cache the response (unless the request handler prevents caching during the course of processing the request).

pCacheCtx

[out] Address of the pointer variable that, on success, receives the IHttpServerContext interface pointer of the object responsible for providing the context for the response that is about to be cached.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

The default implementation of this method returns a newly created CCacheServerContext object. If necessary, you can override this implementation in a derived class to return an object with a custom implementation of IHttpServerContext. You may wish to override this method if you have a special policy for cached responses or if you wish to cache the response using a specific security context. Custom objects can delegate to CCacheServerContext where necessary.

By default **CCacheServerContext** uses the security context of the current client to create the file that holds the cached response. If authenticated access is enabled, a response cached by one client may not be available to another client. (If authenticated access is not enabled, the creation and subsequent attempts to read the cached response will be made from the same security context). **CIsapiExtension** generates a new non-cached response for clients that cannot access the currently cached response, but the benefits of caching may be lost as a result of allowing authenticated access.

By overriding this method to return a custom server context object, you can take control of the way in which the file containing the cached response is created. By overriding CIsapiExtension::TransmitFromCache, you can take control of the way in which the file containing the cached response is read.

See Also

Reference

CIsapiExtension Class

CIsapiExtension::TransmitFromCache

Other Resources

CIsapiExtension Members

CIsapiExtension::GetCriticalIsapiError

Call this method to get the value of a critical ISAPI error set by a previous call to SetCriticalIsapiError.

```
DWORD GetCriticalIsapiError( ) throw( );
```

Return Value

Returns the error code set by a previous call to SetCriticalIsapiError.

Remarks

Always returns 0 if ATL_NO_CRITICAL_ISAPI_ERROR is defined.

See Also

Reference

CIsapiExtension Class
CIsapiExtension::SetCriticalIsapiError
CIsapiExtension::ReturnCriticalError

Other Resources

CIsapiExtension Members

CIsapiExtension::GetCurrWaiting

Implementation of IRequestStats::GetCurrWaiting.

```
long GetCurrWaiting( );
```

Remarks

Delegates to m_reqStats.

See Also

Reference

CIsapiExtension Class

IRequestStats::GetCurrWaiting

Other Resources

CIsapiExtension Members

CIsapiExtension::GetDllCacheTimeout

Override this method to return the timeout value in milliseconds used by the DLL cache.

```
virtual DWORD GetDllCacheTimeout( ) throw( );
```

Return Value

Returns the timeout value to be used by the DLL cache.

Remarks

The default implementation of the method in this class returns ATL_DLL_CACHE_TIMEOUT.

The timeout value is the time between calls to IDllCache::Flush. If a DLL has not been used since the last call to **Flush**, it will be removed from the cache.

See Also

Reference

CIsapiExtension Class

CIsapiExtension::m_DllCache

Other Resources

CIsapiExtension Members

CIsapiExtension::GetExtensionDesc

Override this method to return the ISAPI extension description used by IIS for logging.

```
virtual LPCSTR GetExtensionDesc( ) throw( );
```

Return Value

Returns the ISAPI extension description used by IIS for logging.

Remarks

The default implementation of the method in this class returns the string "ATL Server Classes". You can return another string to identify your ISAPI extension.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::GetExtensionVersion

Call this method from your ISAPI DLL's **GetExtensionVersion** entry point to initialize this object.

```
BOOL GetExtensionVersion(
   HSE_VERSION_INFO* pVer
) throw( );
```

Parameters

pVer

The version information.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This method is used to initialize the heaps, threads, and caches used by this class. It also provides the implementation of the **GetExtensionVersion** entry point expected by IIS.

Example

See the Hotswap Sample and the MantaWeb Sample.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::GetFailedRequests

Implementation of IRequestStats::GetFailedRequests.

```
long GetFailedRequests( );
```

Remarks

Delegates to m_reqStats.

See Also

Reference

CIsapiExtension Class

IRequestStats::GetFailedRequests

Other Resources

CIsapiExtension Members

CIsapiExtension::GetHandlerName

Call this method to get the name of the primary handler from a server response file.

```
HTTP_CODE GetHandlerName(
   LPCSTR szFileName,
   LPSTR szHandlerName
) throw( );
```

Parameters

szFileName

The name of the file from which to extract the handler name.

szHandlerName

A pointer to a caller-allocated buffer of at least MAX_PATH + ATL_MAX_HANDLER_NAME_LEN + 2 characters that will receive the handler name on successful return of this method.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

The file identified by *szFileName* is parsed for the handler tag and the name and path identified by that tag is returned in *szHandlerName*.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::GetIOCompletionHandle

Override this method to return a handle if you want to initialize the thread pool to associate its IO completion port with the object associated with that handle.

```
virtual HANDLE GetIOCompletionHandle( ) throw( );
```

Return Value

Returns the handle of the object to be associated with the thread pool's IO completion port.

Remarks

The default implementation of the method in this class returns INVALID_HANDLE_VALUE. If you override this method to return a file handle, the file must have been opened with the FILE_FLAG_OVERLAPPED flag.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::GetMaxWaiting

Implementation of IRequestStats::GetMaxWaiting.

```
long GetMaxWaiting( );
```

Return Value

Delegates to m_reqStats.

See Also

Reference

CIsapiExtension Class

IRequestStats::GetMaxWaiting

Other Resources

CIsapiExtension Members

CIsapiExtension::GetNumPoolThreads

Override this method to return the number of threads with which to start the thread pool.

```
virtual int GetNumPoolThreads( ) throw( );
```

Return Value

Returns the number of threads with which to start the thread pool.

Remarks

The default implementation of the method in this class returns 0, indicating that the thread pool should start with ATLS_DEFAULT_THREADSPERPROC threads per processor. You can override this method to return a positive integer to set an absolute number of threads, or a negative integer to set the number of threads per processor. You can also adjust the size of the thread pool at runtime using the IThreadPoolConfig interface which you can obtain by calling QueryService.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::GetPoolStackSize

Override this method to return the stack size to be used for each thread in the pool.

```
virtual int GetPoolStackSize( ) throw( );
```

Return Value

Returns the stack size to be used for each thread in the pool.

Remarks

The default implementation of the method in this class returns 0, indicating that the pool threads should use the same stack size as the thread that calls GetExtensionVersion.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::GetService

Call this method to get an interface pointer on a dynamic service supported by the ISAPI extension.

```
HRESULT GetService(
   REFGUID guidService,
   REFIID riid,
   void** ppvObject
) throw( );
```

Parameters

guidService

The GUID identifying the service.

riid

The IID of an interface provided by the service.

ppvObject

[out] Address of the caller-allocated variable to receive the interface pointer of the service on successful return from this method. The caller becomes responsible for calling **IUnknown::Release** on this interface pointer when the service is no longer needed.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method is called by the implementation of **IServiceProvider::QueryService**. It only returns dynamic services served by the ISAPI extension. Call CIsapiExtension::QueryService to get access to all of the services supported by the ISAPI extension.

See Also

Reference

CIsapiExtension Class

CIsapiExtension::QueryService

Other Resources

CIsapiExtension Members

CIsapiExtension::GetStencilCacheTimeout

Override this method to return the timeout value in milliseconds used by the stencil cache.

```
virtual DWORD GetStencilCacheTimeout( ) throw( );
```

Return Value

Returns the timeout value to be used by the stencil cache.

Remarks

The default implementation of the method in this class returns ATL_STENCIL_CACHE_TIMEOUT.

The timeout value is the period with which the cache is checked for stale entries.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::GetStencilLifespan

Override this method to return the default life span in milliseconds of a stencil to be used by the stencil cache.

```
virtual LONGLONG GetStencilLifespan( ) throw( );
```

Return Value

Returns the default life span of a stencil to be used by the stencil cache.

Remarks

The default implementation of the method in this class returns ATL_STENCIL_LIFESPAN.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::GetThreadWorker

Implementation of IIsapiExtension::GetThreadWorker.

```
CIsapiWorker* GetThreadWorker( );
```

See Also

Reference

CIsapiExtension Class

CIsapiExtension::SetThreadWorker

Other Resources

CIsapiExtension Members

CIsapiExtension::GetTotalRequests

Implementation of IRequestStats::GetTotalRequests.

```
long GetTotalRequests( );
```

Remarks

Delegates to m_reqStats.

See Also

Reference

CIsapiExtension Class

IRequestStats::GetTotalRequests

Other Resources

CIsapiExtension Members

CIsapiExtension::HandleError

Call this method to send an error response to the client.

```
void HandleError(
   IHttpServerContext* pServerContext,
   DWORD dwStatus,
   DWORD dwSubStatus
) throw( );
```

Parameters

pServerContext

The server context.

dwStatus

The HTTP status code to return to the client.

dwSubStatus

The ATL Server subcode.

Remarks

This method uses the *HttpUserErrorTextProvider* template parameter to generate the response body.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::HttpExtensionProc

Call this method from your ISAPI DLL's **HttpExtensionProc** entry point to queue HTTP requests on the thread pool.

```
DWORD HttpExtensionProc(
   LPEXTENSION_CONTROL_BLOCK lpECB
) throw( );
```

Parameters

lpECB

The extension control block.

Return Value

Returns HSE_STATUS_PENDING if the request is successfully queued, HSE_STATUS_ERROR otherwise.

Remarks

This method implements the **ISAXXMLReader** entry point expected by IIS by initializing an AtlServerRequest object with information from the extension control block then queuing it on the thread pool.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::LoadDispatchFile

Call this method to populate the AtlServerRequest structure with the request handler interface pointer and DLL instance handle for the handler specified in a server response file.

```
HTTP_CODE LoadDispatchFile(
   LPCSTR szFileName,
   AtlServerRequest* pRequestInfo
);
```

Parameters

szFileName

The name of a server response file.

pRequestInfo

Pointer to the object holding information about the request.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

Call this method to load a server response file, parse it for information about the request handler, load the request handler, and populate the AtlServerRequest structure with the request handler interface pointer and DLL instance handle.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::LoadDllHandler

Call this method to load the request handler for the current request based on the DLL file name and the server context.

```
HTTP_CODE LoadDllHandler(
   LPCSTR szFileName,
   AtlServerRequest* pRequestInfo
);
```

Parameters

szFileName

The file name of the request handler DLL.

pRequestInfo

Pointer to the object holding information about the request with a valid pServerContext member.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

This method loads the handler identified by the **Handler** query string parameter (or the default handler if the query string parameter is not provided) from the DLL identified by *szFileName*. The hInstDll and pHandler members of *pRequestInfo* will hold the instance handle of the request handler DLL and the request handler's **IRequestHandler** interface pointer on successful return of this method.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::LoadRequestHandler

Call this method to retrieve the request handler interface pointer and DLL instance handle for the specified handler.

```
HTTP_CODE LoadRequestHandler(
   LPCSTR szDllPath,
   LPCSTR szHandlerName,
   IHttpServerContext* pServerContext,
   HINSTANCE* phInstance,
   IRequestHandler** ppHandler
);
```

Parameters

szDllPath

The name path of the request handler DLL.

szHandlerName

The name of a request handler.

pServerContext

The server context.

phInstance

[out] Address of the HINSTANCE variable that, on success, receives the instance handle of the request handler DLL.

ppHandler

[out] Address of the pointer variable that, on success, receives the request handler's **IRequestHandler** interface pointer.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::OnThreadAttach

Override this method to execute code when a worker thread is initialized.

```
BOOL OnThreadAttach( );
```

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This method is the complement to CIsapiExtension::OnThreadTerminate.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::OnThreadTerminate

Override this method to execute code when a worker thread is about to be destroyed.

```
void OnThreadTerminate( );
```

Remarks

This method is the complement to IIsapiExtension::OnThreadAttach.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::QueryInterface

Implementation of **IUnknown::QueryInterface**.

```
HRESULT STDMETHODCALLTYPE QueryInterface(
   REFIID riid,
   void** ppv
);
```

Remarks

CIsapiExtension can be successfully queried for the following interfaces:

- IRequestStats
- IServiceProvider
- IIsapiExtension

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::QueryService

Call this method to get a service from the ISAPI extension or override this method to support new services.

```
virtual HRESULT STDMETHODCALLTYPE QueryService(
   REFGUID guidService,
   REFIID riid,
   void** ppvObject
);
```

Parameters

guidService

The identifier of the service.

riid

The ID of the requested interface in the service identified by *guidService*.

ppvObject

[out] Address of the pointer variable that, on success, receives the interface pointer identified by *guidService* and *riid*.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

CIsapiExtension::QueryService supports the following interfaces: IDllCache, IStencilCache, IThreadPoolConfig, and IAtlMemMgr where the service identifier is the same as the interface ID, plus any dynamic services added by a call to CIsapiExtension::AddService.

CIsapiExtension::QueryService can also be successfully queried for an ISAXXMLReader interface when ATL_NO_SOAP is not defined.

Example

See the SOAPState Sample.

See Also

Reference

CIsapiExtension Class
CIsapiExtension::AddService
CIsapiExtension::RemoveService

Other Resources

CIsapiExtension Members

CIsapiExtension::QueueRequest

Call this method to queue a request on the thread pool.

```
BOOL QueueRequest(
   AtlServerRequest* pRequestInfo
);
```

Parameters

pRequestInfo

Pointer to the object holding information about the request.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::Release

This implementation of **IUnknown::Release** always returns 1.

```
ULONG STDMETHODCALLTYPE Release( );
```

Return Value

Returns 1.

See Also

Reference

CIsapiExtension Class

CIsapiExtension::AddRef

Other Resources

CIsapiExtension Members

CIsapiExtension::RemoveService

Implementation of IIsapiExtension::RemoveService.

```
virtual HRESULT RemoveService(
   REFGUID guidService,
   REFIID riid
);
```

See Also

Reference

CIsapiExtension Class

CIsapiExtension::AddService

Other Resources

CIsapiExtension Members

CIsapiExtension::RequestComplete

Call this method when the handling of the request is complete.

```
void RequestComplete(
   AtlServerRequest* pRequestInfo,
   DWORD dwStatus,
   DWORD dwSubStatus
);
```

Parameters

pRequestInfo

Pointer to the object holding information about the request.

dwStatus

The HTTP status code.

dwSubStatus

The ATL Server subcode.

Remarks

This method uninitializes the request handler, calls CIsapiExtension::HandleError if the HTTP status code is greater than or equal to 400, and frees the request object.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::ReturnCriticalError

Call this method to return a response describing the critical ISAPI extension error.

```
DWORD ReturnCriticalError(
   EXTENSION_CONTROL_BLOCK * pECB
) throw( );
```

Parameters

pECB

The extension control block for the current request.

Return Value

Returns HSE_STATUS_SUCCESS on success or HSE_STATUS_ERROR on failure.

Remarks

By default, this method is called by HttpExtensionProc to return an HTTP response to every request to this ISAPI extension when there has been a previous call to SetCriticalIsapiError.

By default, the error response will include a description of the error, but when ATL_CRITICAL_ISAPI_ERROR_LOGONLY is defined, only a generic error message is returned.

When ATL_NO_CRITICAL_ISAPI_ERROR is defined, this method is no longer called or defined.

See Also

Reference

CIsapiExtension Class
CIsapiExtension::GetCriticalIsapiError
CIsapiExtension::SetCriticalIsapiError

Other Resources

CIsapiExtension Members

CIsapiExtension::SetCriticalIsapiError

Call this method to notify the ISAPI extension that a critical error has occurred.

```
BOOL SetCriticalIsapiError(
   DWORD dwErr = 1
) throw( );
```

Parameters

dwErr

The error that occurred. This value can also be the ID of a string resource containing a plain text description of the error that can be used in the Windows Event Log and returned as part of an HTTP response.

Return Value

Returns **FALSE** if ATL_NO_CRITICAL_ISAPI_ERROR is defined. Returns **TRUE** otherwise.

Remarks

This method is called during GetExtensionVersion when an error occurs during initialization.

By default this method stores the error code so that it can be retrieved by calling GetCriticalIsapiError, loads the error message from a resource, writes the message to the Event Log and returns TRUE.

When ATL_NO_CRITICAL_ISAPI_ERROR is defined, this method ignores the error and simply returns FALSE.

See Also

Reference

CIsapiExtension Class
CIsapiExtension::GetCriticalIsapiError
CIsapiExtension::ReturnCriticalError

Other Resources

CIsapiExtension Members

CIsapiExtension::SetThreadWorker

Implementation of IIsapiExtension::SetThreadWorker.

```
BOOL SetThreadWorker(
   CIsapiWorker* pWorker
);
```

See Also

Reference

CIsapiExtension Class

CIsapiExtension::GetThreadWorker

Other Resources

CIsapiExtension Members

CIsapiExtension::StartAsyncFlush

Call this method to start writing buffered content to the client asynchronously.

```
BOOL StartAsyncFlush(
   AtlServerRequest* pRequestInfo
) throw( );
```

Parameters

pRequestInfo

Pointer to the object holding information about the request.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

See Also

Reference

CIsapiExtension Class

Concepts

Writing Asynchronous Web Applications

Other Resources

CIsapiExtension Members

CIsapiExtension::TerminateExtension

Call this method from your ISAPI DLL's **TerminateExtension** entry point to free the resources used by the ISAPI extension.

```
BOOL TerminateExtension(
   DWORD /*dwFlags*/
) throw( );
```

Parameters

dwFlags

The shutdown flags.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This method implements the **TerminateExtension** entry point expected by IIS by cleaning up the caches, performance monitoring objects, dynamic services, and other resources acquired by the extension.

Example

See the Hotswap Sample and the MantaWeb Sample.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::TransferRequest

Implementation of IIsapiExtension::TransferRequest.

```
HTTP_CODE TransferRequest(
   AtlServerRequest * pRequest,
   IServiceProvider * pServiceProvider,
   IWriteStream * pWriteStream,
   IHttpRequestLookup * pLookup,
   LPCSTR szNewUrl,
   WORD nCodePage,
   bool bContinueAfterProcess,
   CStencilState * pState
);
```

See Also

Reference

CIsapiExtension Class

IIsapiExtension::TransferRequest

Other Resources

CIsapiExtension Members

CIsapiExtension::TransmitFromCache

Call this method to transmit an item from the page cache. Override this method to control the way in which cached items are transmitted to the client.

```
virtual BOOL TransmitFromCache(
   AtlServerRequest* pRequestInfo,
   BOOL * pbAllowCaching
) throw( );
```

Parameters

pRequestInfo

Pointer to the object holding information about the request.

pbAllowCaching

Pointer to the variable that, if **TransmitFromCache** returns FALSE, indicates whether the response to the current request can be cached.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

If the item identified by *strCacheUrl* is in the cache and the current user has access, it is transmitted asynchronously.

This method returns FALSE if the HTTP request method is anything other than GET or if the page is not in the cache.

**pbAllowCaching* is set to FALSE if the file is invalid or could not be opened.

Typically this method is executed in the security context of the current client. If necessary, you can override this method in a derived class to change the security context before calling the base class implementation then revert to the original security context before returning. See CIsapiExtension::GetCacheServerContext for more details on why you might want to do this.

See Also

Reference

CIsapiExtension Class

CIsapiExtension::GetCacheServerContext

Other Resources

CIsapiExtension Members

CIsapiExtension Data Members

For information about the data members in **CIsapiExtension**, see CIsapiExtension Members.

CIsapiExtension::m_DllCache

The DLL cache used to speed up access to Web application DLLs.

```
CDllCache<extWorkerType, CDllCachePeer> m_DllCache;
```

Example

See the Hotswap Sample.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::m_PageCache

The page cache used to cache responses generated by request handlers.

```
CFileCache<extWorkerType, CPageCacheStats, CPageCachePeer> m_PageCache;
```

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::m_reqStats

The object responsible for tracking request-handling statistics.

```
CRequestStatClass m_reqStats;
```

Remarks

CRequestStatClass is a parameter used to specialize the class template.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::m_StencilCache

The stencil cache used to hold pre-parsed server response files and speed up rendering of their responses.

```
CComObjectGlobal< CStencilCache<extWorkerType, CStencilCacheStats > >
   m_StencilCache;
```

Example

See the Hotswap Sample.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::m_UserErrorProvider

The error provider.

```
HttpUserErrorTextProvider m_UserErrorProvider;
```

Remarks

HttpUserErrorTextProvider is a parameter used to specialize the class template.

See Also

Reference

CIsapiExtension Class

Other Resources

CIsapiExtension Members

CIsapiExtension::m_WorkerThread

The worker thread used to maintain the page, stencil, and DLL caches.

```
extWorkerType m_WorkerThread;
```

Remarks

This worker thread can also be used to maintain other caches that you add to the ISAPI extension class.

See Also

Reference

CIsapiExtension Class

CIsapiExtension::extWorkerType

Other Resources

CIsapiExtension Members

CIsapiExtension Typedefs

For information about the typedefs in **CIsapiExtension**, see CIsapiExtension Members.

CIsapiExtension::extWorkerType

An alias for a worker thread class based on the *WorkerThreadTraits* parameter used to specialize the class template.

```
typedef CWorkerThread<WorkerThreadTraits> extWorkerType;
```

Remarks

This item is protected.

See Also

Reference

CIsapiExtension Class

CIsapiExtension::m_WorkerThread

Other Resources

CIsapiExtension Members

CIsapiWorker Class

Receives ATL Server requests from the thread pool and passes them on to an object implementing IIsapiExtension.

```
class CIsapiWorker
```

Remarks

This class represents a worker thread as used by CIsapiExtension and conforming to the standards for interoperating with CThreadPool described in Worker Archetype.

CIsapiWorker is a worker class that integrates with **CThreadPool**. It takes requests that have been queued on the thread pool and hands them back to an object implementing **IIsapiExtension** by calling IIsapiExtension::DispatchStencilCall. It also calls IIsapiExtension::OnThreadAttach when the thread is created followed by a call to IIsapiExtension::SetThreadWorker. The class calls IIsapiExtension::OnThreadTerminate when the thread is destroyed.

If you need to make per-thread data or services available to your ISAPI extension or the request handlers that use it, you can provide those facilities in a class derived from **CIsapiWorker**. Just change the *ThreadPoolClass* template parameter used to specialize CIsapiExtension to use your derived class in place of **CIsapiWorker**. Each thread in the thread pool will have a single instance of that class, which can be accessed by calling IIsapiExtension::GetThreadWorker. If you expose the unique features of your class by overriding CIsapiWorker::GetWorkerData, users of your class will not have to cast the pointer that they get back from IIsapiExtension::GetThreadWorker.

Requirements

Header: atlisapi.h

Example

See the MantaWeb Sample.

See Also

Other Resources

CIsapiWorker Members

ATL Server Classes

CIsapiWorker Members

Methods

CIsapiWorker	The constructor.
~CIsapiWorker	The destructor.
Execute	This method is called by CThreadPool when a request needs to be processed.
GetWorkerData	Call this method to get access to services provided by the worker class.
Initialize	This method is called by CThreadPool to initialize the worker object before any requests are passed to Execute.
Terminate	This method is called by CThreadPool to uninitialize the worker object after all requests have been passed to **Execute**.

Data Members

m_hHeap	The handle to the thread heap.
m_spReader	A pointer to a **SAXXMLReader** object.

Typedefs

RequestType	A synonym for an ATLServerRequest pointer.

See Also

Reference

CIsapiWorker Class

CIsapiWorker Methods

For information about the methods in **CIsapiWorker**, see CIsapiWorker Members.

CIsapiWorker::CIsapiWorker

The constructor.

```
CIsapiWorker( ) throw( );
```

Remarks

Initializes CIsapiWorker::m_hHeap to NULL.

See Also

Reference

CIsapiWorker Class

Other Resources

CIsapiWorker Members

CIsapiWorker::~CIsapiWorker

The destructor.

```
~CIsapiWorker( ) throw( );
```

See Also

Reference

CIsapiWorker Class

Other Resources

CIsapiWorker Members

CIsapiWorker::Execute

This method is called by CThreadPool when a request needs to be processed.

```
void Execute(
   AtlServerRequest* pRequestInfo,
   void* pvParam,
   OVERLAPPED* pOverlapped
) throw( );
```

Parameters

pRequestInfo

Pointer to the object holding information about the request.

pvParam

A pointer to the IIsapiExtension interface of the ISAPI extension object handling the request.

pOverlapped

The overlapped structure. Not used.

Remarks

This method just passes the request on to the ISAPI extension by calling IIsapiExtension::DispatchStencilCall through *pvParam*.

Note

The call to **DispatchStencilCall** is wrapped in an exception handler that traces the fact that an exception has been generated and triggers an assertion failure in debug builds. The exception is not rethrown. If you need other exception handling behavior in your ISAPI extension, you should use a class other than **CIsapiWorker** or override CIsapiExtension::DispatchStencilCall to handle exceptions before they reach this method.

See Also

Reference

CIsapiWorker Class

WorkerArchetype::Execute

Other Resources

CIsapiWorker Members

CIsapiWorker::GetWorkerData

Call this method to get access to services provided by the worker class.

```
virtual BOOL GetWorkerData(
   DWORD  /* dwParam */,
   void** /* ppvData */
) throw( );
```

Parameters

dwParam

The identifier for the requested service.

ppvData

[out] Address of the pointer variable that, on success, receives the pointer to the requested service.

Return Value

Returns **TRUE** on success and **FALSE** on failure.

Remarks

The implementation of this method in this class always returns FALSE. This virtual method is present so that it can be overridden in derived classes. See CIsapiWorker Overview for more information.

Example

See the MantaWeb Sample.

See Also

Reference

CIsapiWorker Class

Other Resources

CIsapiWorker Members

CIsapiWorker::Initialize

This method is called by CThreadPool to initialize the worker object before any requests are passed to Execute.

```
virtual BOOL Initialize(
   void* pvParam
) throw( );
```

Parameters

pvParam

A pointer to the IIsapiExtension interface of the ISAPI extension object handling the requests.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This method does the following:

- Calls IIsapiExtension::OnThreadAttach.
- Creates a heap and stores the handle in m_hHeap.
- Creates a **SAXXMLReader** object and stores the interface pointer in m_spReader (unless ATL_NO_SOAP is defined).
- Calls IIsapiExtension::SetThreadWorker passing the "this" pointer.

See Also

Reference

CIsapiWorker Class

WorkerArchetype::Initialize

Other Resources

CIsapiWorker Members

CIsapiWorker::Terminate

This method is called by CThreadPool to uninitialize the worker object after all requests have been passed to Execute.

```
virtual void Terminate(
   void* pvParam
) throw( );
```

Parameters

pvParam

A pointer to the IIsapiExtension interface of the ISAPI extension object handling the requests.

Remarks

This method destroys the thread's heap and releases m_spReader before calling IIsapiExtension::OnThreadTerminate.

See Also

Reference

CIsapiWorker Class

WorkerArchetype::Terminate

Other Resources

CIsapiWorker Members

CIsapiWorker Data Members

For information about the data members in **CIsapiWorker**, see CIsapiWorker Members.

CIsapiWorker::m_hHeap

The handle to the thread heap.

```
HANDLE m_hHeap;
```

Remarks

The heap is created in Initialize. This heap is not synchronized so all allocations and deallocations must be performed from the same thread.

See Also

Reference

CIsapiWorker Class

Other Resources

CIsapiWorker Members

CIsapiWorker::m_spReader

A pointer to a **SAXXMLReader** object.

```
CComPtr<ISAXXMLReader> m_spReader;
```

Remarks

The pointer is obtained in Initialize. This member is not available if ATL_NO_SOAP is defined.

See Also

Reference

CIsapiWorker Class

Other Resources

CIsapiWorker Members

CIsapiWorker Typedefs

For information about the typedefs in **CIsapiWorker**, see CIsapiWorker Members.

CIsapiWorker::RequestType

A synonym for an ATLServerRequest pointer.

```
typedef AtlServerRequest* RequestType;
```

Remarks

This typedef provides the information required by CThreadPool as described in WorkerArchetype::RequestType.

See Also

Reference

CIsapiWorker Class

WorkerArchetype::RequestType

Other Resources

CIsapiWorker Members

CLifetimeCuller Class

This class conforms to the cache culler archetype and allows the automatic removal of cache items that haven't been accessed in a certain period. The lifetime of an item can be changed at runtime and can be different for each of the items in the cache.

```
class CLifetimeCuller :
   public CExpireCuller
```

Remarks

This class interprets a life span of zero as meaning that the item should never expire.

Requirements

Header: atlcache.h

See Also

Reference

CExpireCuller Class

Other Resources

CLifetimeCuller Members

Caching

Caching Reference

CLifetimeCuller Members

Methods

Access	Implementation of CacheCullerArchetype::Access.
Add	Implementation of CacheCullerArchetype::Add.
Commit	Implementation of CacheCullerArchetype::Commit.
GetExpired	Implementation of CacheCullerArchetype::GetExpired.

See Also

Reference

CLifetimeCuller Class

Other Resources

Caching Reference

Caching

CLifetimeCuller Methods

For information about the methods in **CLifetimeCuller**, see CLifetimeCuller Members

CLifetimeCuller::Access

Implementation of CacheCullerArchetype::Access.

```
void Access(
   CCullerCacheData* pItem
);
```

Remarks

Sets the expiration time of the item by adding the item's life span to the current time.

See Also

Reference

CLifetimeCuller Class
Cache Culler Archetype

Other Resources

CLifetimeCuller Members
Caching
Caching Reference

CLifetimeCuller::Add

Implementation of CacheCullerArchetype::Add.

```
void Add(
   CCullerCacheData* pItem
);
```

Remarks

Sets the life span of the item to zero.

See Also

Reference

CLifetimeCuller Class
Cache Culler Archetype

Other Resources

CLifetimeCuller Members
Caching
Caching Reference

CLifetimeCuller::Commit

Implementation of CacheCullerArchetype::Commit.

```
void Commit(
   CCullerCacheData* pItem
);
```

Remarks

Sets the expiration time of the item by adding the item's life span to the current time.

See Also

Reference

CLifetimeCuller Class
Cache Culler Archetype

Other Resources

CLifetimeCuller Members
Caching
Caching Reference

CLifetimeCuller::GetExpired

Implementation of CacheCullerArchetype::GetExpired.

```
CCullerCacheData* GetExpired( );
```

Remarks

Delegates to CExpireCuller::GetExpired.

See Also

Reference

CLifetimeCuller Class
Cache Culler Archetype

Other Resources

CLifetimeCuller Members
Caching
Caching Reference

CLOUFlusher Class

This class conforms to the cache flusher archetype and allows the automatic removal of cache items based on their access pattern. Items that are used infrequently are presented for removal before items that are used more often.

```
class CLOUFlusher :
   public COldFlusher
```

Remarks

This class promotes the removal of items that are used infrequently by ordering items primarily by the number of times that they have been accessed and secondarily by the order in which they were added to the cache. Thus an item that has been accessed a certain number of times over a long period will be removed before an item that has been accessed the same number of times, but over a shorter period of time.

Requirements

Header: atlcache.h

See Also

Reference

COldFlusher Class

Other Resources

CLOUFlusher Members

Caching

Caching Reference

CLOUFlusher Members

Methods

Access	Implementation of CacheFlusherArchetype::Access.
Add	Implementation of CacheFlusherArchetype::Add.
GetNext	Implementation of CacheFlusherArchetype::GetNext.
GetStart	Implementation of CacheFlusherArchetype::GetStart.

Data Members

pHead	The leading node of a linked list.
pTail	The tail node of a linked list.

See Also

Reference

CLOUFlusher Class

Other Resources

Caching

Caching Reference

CLOUFlusher Methods

For information about the methods in **CLOUFlusher**, see CLOUFlusher Members

CLOUFlusher::Access

Implementation of CacheFlusherArchetype::Access.

```
void Access(
   CFlusherCacheData* pItem
);
```

Remarks

If necessary, reorders the linked list using *pItem*->dwAccessed so that items with low numbers of accesses are near the tail of the list.

See Also

Reference

CLOUFlusher Class

Cache Flusher Archetype

Other Resources

CLOUFlusher Members

Caching

Caching Reference

CLOUFlusher::Add

Implementation of CacheFlusherArchetype::Add.

```
void Add(
   CFlusherCacheData* pItem
);
```

Remarks

Sets *pItem*->dwAccessed to 1 and calls COldFlusher::Add to add the item to the linked list.

See Also

Reference

CLOUFlusher Class

Cache Flusher Archetype

Other Resources

CLOUFlusher Members

Caching

Caching Reference

CLOUFlusher::GetNext

Implementation of CacheFlusherArchetype::GetNext.

```
CFlusherCacheData* GetNext(
   CFlusherCacheData* pCur
) const;
```

Remarks

Returns *pCur*->pPrev.

See Also

Reference

CLOUFlusher Class

Cache Flusher Archetype

Other Resources

CLOUFlusher Members

Caching

Caching Reference

CLOUFlusher::GetStart

Implementation of CacheFlusherArchetype::GetStart.

```
CFlusherCacheData* GetStart( ) const;
```

Remarks

Returns pTail.

See Also

Reference

CLOUFlusher Class

Cache Flusher Archetype

Other Resources

CLOUFlusher Members

Caching

Caching Reference

CLOUFlusher Data Members

For information about the data members in **CLOUFlusher**, see CLOUFlusher Members

CLOUFlusher::pHead

The leading node of a linked list.

```
CFlusherCacheData * pHead;
```

Remarks

This node describes the most frequently accessed item and therefore the last to be offered as a candidate for removal.

See Also

Reference

CLOUFlusher Class

Cache Flusher Archetype

Other Resources

CLOUFlusher Members

Caching

Caching Reference

CLOUFlusher::pTail

The tail node of a linked list.

```
CFlusherCacheData * pTail;
```

Remarks

This node describes the least frequently accessed cache entry and therefore the first to be offered as a candidate for removal.

See Also

Reference

CLOUFlusher Class

Cache Flusher Archetype

Other Resources

CLOUFlusher Members

Caching

Caching Reference

CLRUFlusher Class

This class conforms to the cache flusher archetype and allows the automatic removal of cache items based on their access pattern. Items that have not been used for a long time are presented for removal before items that have been used more recently.

```
class CLRUFlusher :
   public COldFlusher
```

Remarks

This class promotes the removal of items that have not been used for a long time by ordering items based on when they were last accessed. Thus an item that has been accessed recently is unlikely to be removed from the cache.

Requirements

Header: atlcache.h

See Also

Reference

COldFlusher Class

Other Resources

CLRUFlusher Members

Caching

Caching Reference

CLRUFlusher Members

Methods

Access	Implementation of CacheFlusherArchetype::Access.

See Also

Reference

CLRUFlusher Class

Other Resources

Caching

Caching Reference

CLRUFlusher Methods

For information about the methods in **CLRUFlusher**, see CLRUFlusher Members

CLRUFlusher::Access

Implementation of CacheFlusherArchetype::Access.

```
void Access(
   CFlusherCacheData* pItem
);
```

Remarks

Moves the item to the tail of the linked list to make it the least likely item to be removed from the cache.

See Also

Reference

CLRUFlusher Class

Cache Flusher Archetype

Other Resources

CLRUFlusher Members

Caching

Caching Reference

CMemoryCache Class

This class represents a generic memory cache.

```
template <
   typename DataType,
   class StatClass  = CStdStatClass,
   class FlushClass = COldFlusher,
   class keyType    = CFixedStringKey,
   class KeyTrait   = CStringElementTraits< CFixedStringKey >,
   class SyncClass  = CComCriticalSection,
   class CullClass  = CExpireCuller
>
class CMemoryCache :
   public CMemoryCacheBase< CMemoryCache, DataType, CCacheDataEx,
                                                    keyType, KeyTrait, FlushClass, Cull
Class,
                                                    SyncClass, StatClass >
```

Parameters

DataType

The data structure that defines the data to be cached. This structure holds the data that users of the cache must provide when adding items to the cache.

StatClass

The class used to collect and expose statistics related to the use of the cache including such information as effectiveness of the cache and memory used. Must conform to the cache statistics archetype.

FlushClass

The class used to determine whether, and in what order, items should be removed from the cache using the number and sequence of accesses as the basis for making the decision. Must conform to the cache flusher archetype.

keyType

The type used to identify elements in the cache.

KeyTrait

The type used to compare *keyType* objects.

SyncClass

A class to be used for thread synchronization. The default value of CComCriticalSection is suitable for most situations, but the CComFakeCriticalSection class can be used to avoid thread synchronization in single-threaded environments.

CullClass

The class used to determine whether, and in what order, items should be removed from the cache using time-based criteria. Must conform to the cache culler archetype.

Remarks

This class implements a cache that stores items in memory. **CMemoryCache** uses CMemoryCacheBase as a base class and adds the ability to associate cached DLLs with cache entries and provide custom destruction for items being removed from the cache.

Requirements

Header: atlcache.h

See Also

Reference

CMemoryCacheBase Class

Other Resources

Caching
CMemoryCache Members
Caching Reference

CMemoryCache Members

Methods

AddEntry	Call this method to add an entry to the cache.
Initialize	Call this method to initialize the memory cache.
OnDestroyEntry	This method is called to destroy each item that is purged from the cache.

See Also

Reference

CMemoryCache Class

Other Resources

Caching

Caching Reference

CMemoryCache Methods

For information about the methods in **CMemoryCache**, see CMemoryCache Members.

CMemoryCache::AddEntry

Call this method to add an entry to the cache.

```
HRESULT AddEntry(
   const keyType& Key,
   const DataType& data,
   DWORD dwSize,
   FILETIME* pftExpireTime = NULL,
   HINSTANCE hInstance = NULL,
   IMemoryCacheClient* pClient = NULL,
   HCACHEITEM* phEntry = NULL
);
```

Parameters

Key

A reference to the key for the item being added to the cache. The data type of the key is dependant on the *keyType* class template argument.

data

A reference to the data being added to the cache. The type of the data is dependent on the *DataType* class template argument.

dwSize

The size of the data to be cached.

pftExpireTime

The expiration time after which the entry is to be removed from the cache. A **FILETIME** of zero indicates that the item will not expire.

hInstance

The instance handle of a DLL upon which the cached item relies. The reference count of this DLL is incremented in the DLL cache obtained from the IServiceProvider interface passed to Initialize. Use **NULL** if there is no dependency.

pClient

The optional address of an implementation of IMemoryCacheClient that will be asked to free the cached item when it is removed from the cache. See CMemoryCache::OnDestroyEntry.

phEntry

[out] The optional address of a cache item handle that can later be used to access the new cache entry. If no handle is retrieved, the item can still be accessed later using the LookupEntry method.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Adds an entry to the cache.

If *phEntry* is not NULL, the initial reference count for the item is set to 1. See CMemoryCacheBase::AddRefEntry and CMemoryCacheBase::ReleaseEntry for more details on reference counting.

See Also

Reference

CMemoryCache Class

Other Resources

Caching

CMemoryCache Members

Caching Reference

CMemoryCache::Initialize

Call this method to initialize the memory cache.

```
HRESULT Initialize(
   IServiceProvider* pProvider
);
```

Parameters

pProvider

The address of an IServiceProvider implementation that supports the IDllCache interface. The DLL cache is used to manage the reference count for DLLs upon which cached items depend.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

See Also

Reference

CMemoryCache Class

Other Resources

Caching

CMemoryCache Members

Caching Reference

CMemoryCache::OnDestroyEntry

This method is called to destroy each item that is purged from the cache.

```
virtual void OnDestroyEntry(
   const NodeType* pEntry
);
```

Parameters

pEntry

The address of a NodeType structure that describes the entry that is to be destroyed.

Remarks

If a memory cache client was associated with the item being removed, this method calls IMemoryCacheClient::Free passing the address of the data to be freed.

If a DLL was associated with the item being removed, this method releases the reference count on the cached DLL.

See CMemoryCache::AddEntry for details on how to associate a memory cache client or a DLL with the corresponding cache item.

See Also

Reference

CMemoryCache Class

Other Resources

Caching

CMemoryCache Members

Caching Reference

CMemoryCacheBase Class

This class provides features for caching generic items in memory.

```
template <
   class T,
   class DataType,
   class NodeInfo  = CCacheDataBase,
   class keyType   = CFixedStringKey,
   class KeyTrait  = CStringElementTraits< CFixedStringKey >,
   class Flusher   = COldFlusher,
   class Culler    = CExpireCuller,
   class SyncClass = CComCriticalSection,
   class StatClass = CStdStatClass
>
class CMemoryCacheBase
```

Parameters

T

A class derived from **CMemoryCacheBase<***T*, ...**>**. This class must implement a method with this signature:

```
void OnDestroyEntry(NodeType* pEntry);
```

The implementation of this method should free any custom data allocated for the cache entry and perform other custom cleanup when an item is removed from the cache. See CMemoryCacheBase::NodeType Structure.

DataType

The data structure that defines the data to be cached. This structure holds the data that users of the cache must provide when adding items to the cache.

NodeInfo

The data structure that provides auxiliary information about each cached item. This structure is used by derived classes to hold data that is used in the implementation of the cache and is not intended for direct manipulation by users of the cache. See CMemoryCacheBase::NodeType Structure.

keyType

The type used to identify elements in the cache.

KeyTrait

The type used to compare *keyType* objects.

Flusher

The class used to determine whether, and in what order, items should be removed from the cache using the number and sequence of accesses as the basis for making the decision. Must conform to the cache flusher archetype.

CMemoryCacheBase asks a data member of this type to provide candidates for removal when FlushEntries is called and the number of entries in the cache or the size of the cache exceeds the limit set by a previous call to SetMaxAllowedEntries or SetMaxAllowedSize.

Culler

The class used to determine to determine whether, and in what order, items should be removed from the cache using time-based criteria. Must conform to the cache culler archetype.

CMemoryCacheBase asks a data member of this type to provide candidates for removal when either CullEntries or FlushEntries is called.

SyncClass

The name of the class to be used for thread synchronization. The default value of CComCriticalSection is suitable for most

situations, but CComFakeCriticalSection can be used to avoid thread synchronization in single-threaded environments.

StatClass

The class used to collect and expose statistics related to the use of the cache including such information as effectiveness of the cache and memory used. Must conform to the cache statistics archetype.

Remarks

This class provides the implementation of a generic cache that stores elements in memory. The ATL Server cache implementations CMemoryCache, CBlobCache, CFileCache, and CStencilCache classes all use **CMemoryCacheBase** as a base class.

The items in the cache are reference counted to ensure that multiple clients can access the same items without ill effects. See AddRefEntry and RemoveEntry for more details.

The *keyType* and *keyTrait* template arguments define the data type that the cache uses to access entries. By default, CFixedStringKey is used, providing cache keys that use ANSI string of a fixed length defined by ATL_CACHE_KEY_LENGTH. Cache entries must have unique key values. If you try to add an entry with a key that is exactly the same as an existing key, the existing entry will be overwritten.

Header: atlcache.h

See Also

Other Resources

CMemoryCacheBase Members
Caching
Caching Reference

CMemoryCacheBase Members

Methods

AddEntry	Call this method to add an entry to the cache.
AddRefEntry	Call this method to increment the reference count of a cached item.
CanAddEntry	This method is called to determine if an entry can be added without violating the item count and size limits set by calls to SetMaxAllowedEntries and SetMaxAllowedSize.
CheckAlloc	This method is called to determine if an entry can be added without violating the size limit set by a call to SetMaxAllowedSize.
CheckEntryCount	This method is called to determine if an entry can be added without violating the item count limit set by a call to SetMaxAllowedEntries.
ClearStats	Call this method to reset all the statistics to zero.
CMemoryCacheBase	The constructor.
CommitEntry	This method is called to commit cache entries to the cache.
CullEntries	This method is called to remove expired entries from the cache.
FlushEntries	This method is called to remove entries from the cache using expiration times or access patterns as the criteria for removal.
GetEntryData	Call this method to retrieve data from the cache.
GetMaxAllowedEntries	Call this method to retrieve the maximum number of entries currently allowed in the cache.
GetMaxAllowedSize	Call this method to retrieve the maximum size of the entries currently allowed in the cache.
Initialize	Call this method to initialize the cache.
LookupEntry	Call this method to get the handle of an item in the cache.
ReleaseEntry	Call this method to decrement the reference count of a cached item.
RemoveAllEntries	Call this method to remove all the items from the cache.
RemoveEntry	Call this method to remove an item from the cache given its handle.
RemoveEntryByKey	Call this method to remove an item from the cache given its key.
ResetCache	Call this method to remove the contents of the cache and reset the statistics to zero.
SetMaxAllowedEntries	Call this method to set the maximum number of entries allowed in the cache.
SetMaxAllowedSize	Call this method to set the maximum size of the entries allowed in the cache.
Uninitialize	Call this method to uninitialize the cache.

Methods

NodeType Structure	This protected structure holds the data for each item in the cache.

See Also

Reference

CMemoryCacheBase Class

Other Resources

Caching

Caching Reference

CMemoryCacheBase Methods

For information about the methods in **CMemoryCacheBase**, see CMemoryCacheBase Members

CMemoryCacheBase::AddEntry

Call this method to add an entry to the cache.

```
HRESULT AddEntry(
   const keyType& Key,
   const DataType& data,
   DWORD dwSize,
   HCACHEITEM* phEntry = NULL
);
```

Parameters

Key

A reference to the key for the item being added to the cache. The data type of the key is dependant on the *keyType* class template argument.

data

A reference to the data being added to the cache. The type of the data is dependent on the *DataType* class template argument.

dwSize

The size of the data to be cached.

phEntry

[out] The optional address of a cache item handle that can later be used to access the new cache entry. If no handle is retrieved, the item can still be accessed later using the LookupEntry method.

If *phEntry* is not NULL, the caller of this method must call CommitEntry. Otherwise CommitEntry is called automatically.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Adds an entry to the cache.

If *phEntry* is not NULL, the initial reference count for the item is set to 1. See AddRefEntry and ReleaseEntry for more details on reference counting.

There are two stages to adding an item to the cache using **CMemoryCacheBase**. **AddEntry** ensures that the item can be successfully stored. Following a call to **AddEntry**, it's possible for derived classes to manipulate the cache item and add extra information with confidence that the item has been successfully stored in the cache. Finally CommitEntry is called to let the cache culler act on the final data.

Note that this method does not enforce the restrictions specified by SetMaxAllowedEntries and SetMaxAllowedSize. Derived classes are expected to use CanAddEntry before adding items or to call CullEntries or FlushEntries periodically after adding items to maintain the cache within the boundaries set by the user.

Users typically do not use **CMemoryCacheBase** directly and do not need to be concerned with these details.

See Also

Reference

CMemoryCacheBase Class
HCACHEITEM

Other Resources

CMemoryCacheBase Members
Caching Reference

CMemoryCacheBase::AddRefEntry

Call this method to increment the reference count of a cached item.

```
DWORD AddRefEntry(
   const HCACHEITEM hEntry
);
```

Parameters

hEntry

The handle of the cache item whose reference count is to be incremented.

Return Value

The new reference count.

Remarks

The reference count on items in the cache lets the cache know when items are still being used and prevents them from being deleted prematurely. Handles returned by a call to LookupEntry have a single reference count associated with them. The reference count can be decremented by calling ReleaseEntry and incremented by calling **AddRefEntry**. When the reference count associated with a handle reaches zero, the handle should be discarded.

Use **AddRefEntry** when you want to copy a handle and pass ownership on to some other piece of code.

See Also

Reference

CMemoryCacheBase Class

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase::CanAddEntry

This method is called to determine if an entry can be added without violating the item count and size limits set by calls to SetMaxAllowedEntries and SetMaxAllowedSize.

```
bool CanAddEntry(
   DWORD dwSizeToAdd
);
```

Parameters

dwSizeToAdd

The size of the data item under consideration for addition to the cache.

Return Value

Returns **true** if the item can be added to the cache, otherwise **false**.

Remarks

This method checks both the item count and size limits currently in effect for the cache, and returns **false** if either limit will be violated by the new entry.

This protected method is available for use by derived classes.

See Also

Reference

CMemoryCacheBase Class

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase::CheckAlloc

This method is called to determine if an entry can be added without violating the size limit set by a call to SetMaxAllowedSize.

```
bool CheckAlloc(
   DWORD dwSizeToAdd
);
```

Parameters

dwSizeToAdd

The size of the data item under consideration for addition to the cache.

Return Value

Returns **true** if the memory is available, otherwise **false**.

Remarks

This method checks the memory use limits currently in effect for the cache. The item count limit is not checked.

This protected method is available for use by derived classes.

See Also

Reference

CMemoryCacheBase Class

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase::CheckEntryCount

This method is called to determine if an entry can be added without violating the item count limit set by a call to SetMaxAllowedEntries.

```
bool CheckEntryCount(
   DWORD dwNumEntriesToAdd
);
```

Parameters

dwNumEntriesToAdd

The number of entries under consideration for addition to the cache.

Return Value

Returns **true** if the items can be added to the cache, otherwise **false**.

Remarks

This method checks the item count use limit currently in effect for the cache. The memory usage limit is not checked.

This protected method is available for use by derived classes.

See Also

Reference

CMemoryCacheBase Class

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase::ClearStats

Call this method to reset all the statistics to zero.

```
HRESULT ClearStats( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

See the description of the *StatClass* template parameter in the CMemoryCacheBase Overview.

See Also

Reference

CMemoryCacheBase Class

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase::CMemoryCacheBase

The constructor.

```
CMemoryCacheBase( );
```

Remarks

The **CMemoryCacheBase** constructor performs no operations. The cache is initialized with the Initialize method.

See Also

Reference

CMemoryCacheBase Class

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase::CommitEntry

This method is called to commit cache entries to the cache.

```
HRESULT CommitEntry(
   const HCACHEITEM hEntry
);
```

Parameters

hEntry

The handle of the cache item to be commited.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

See CMemoryCacheBase::AddEntry for instructions on when this method must be called.

See Also

Reference

CMemoryCacheBase Class

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase::CullEntries

This method is called to remove expired entries from the cache.

```
HRESULT CullEntries( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

CullEntries removes all expired items from the cache. Expired items are determined by the cache culler used as the CMemoryCacheBase *Culler* template parameter.

Removal means that the items are no longer available in the cache, but final cleanup of the items will not occur before their reference counts reach zero. See RemoveEntry for further details.

CullEntries is called by FlushEntries.

Derived classes typically call this method automatically so cache users may not need to call it explicitly.

See Also

Reference

CMemoryCacheBase Class

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase::FlushEntries

This method is called to remove entries from the cache using expiration times or access patterns as the criteria for removal.

```
HRESULT FlushEntries( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

FlushEntries calls CullEntries to remove items that have expired. If the current size or number of entries in the cache exceeds a limit set by a call to SetMaxAllowedEntries or SetMaxAllowedSize, this method will also remove the items suggested by the cache flusher until the cache meets the configuration requirements. See the description of the CMemoryCacheBase *Flusher* template parameter.

Removal means that the items are no longer available in the cache, but final cleanup of the items will not occur before their reference counts reach zero. See RemoveEntry for further details.

Derived classes typically call this method automatically so cache users may not need to call it explicitly.

See Also

Reference

CMemoryCacheBase Class

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase::GetEntryData

Call this method to retrieve data from the cache.

```
HRESULT GetEntryData(
   const HCACHEITEM hEntry,
   DataType* pData,
   DWORD* pdwSize
) const;
```

Parameters

hEntry

The handle of the cache item whose data should be retrieved.

pData

The address of the variable to receive the cached item. The type of this data is determined by the *DataType* class template argument.

pdwSize

The address of a DWORD that is to receive the size of the retrieved item.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The handle to an entry in the cache can be obtained when the item is added to the cache by a call to AddEntry, or it can be retrieved for an item already present in the cache by a call to LookupEntry.

See Also

Reference

CMemoryCacheBase Class

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase::GetMaxAllowedEntries

Call this method to retrieve the maximum number of entries currently allowed in the cache.

```
HRESULT STDMETHODCALLTYPE GetMaxAllowedEntries(
   DWORD* pdwSize
);
```

Parameters

pdwSize

[out] Address of the DWORD variable that, on success, receives the maximum number of entries currently allowed in the cache.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

See AddEntry and FlushEntries for information on how the limit represented by this method is used.

See Also

Reference

CMemoryCacheBase Class

CMemoryCacheBase::SetMaxAllowedEntries

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase::GetMaxAllowedSize

Call this method to retrieve the maximum size of the entries currently allowed in the cache.

```
HRESULT STDMETHODCALLTYPE GetMaxAllowedSize(
   DWORD* pdwSize
);
```

Parameters

pdwSize

[out] Address of the DWORD variable that, on success, receives the maximum size of the entries currently allowed in the cache.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

See AddEntry and FlushEntries for information on how the limit represented by this method is used.

See Also

Reference

CMemoryCacheBase Class

CMemoryCacheBase::SetMaxAllowedSize

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase::Initialize

Call this method to initialize the cache.

```
HRESULT Initialize( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Initializes the cache data structures. Must be called before any other methods.

See Also

Reference

CMemoryCacheBase Class

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase::LookupEntry

Call this method to get the handle of an item in the cache.

```
HRESULT LookupEntry(
   const keyType& Key,
   HCACHEITEM* phEntry
);
```

Parameters

Key

The key of the item whose handle should be returned in *phEntry*.

phEntry

[out] Address of the HCACHEITEM variable that, on success, receives the handle of the requested item.

Return Value

Returns S_OK on success, or an error HRESULT on failure. Returns E_FAIL if the item is not in the cache.

Remarks

This method returns a handle to the requested item and increments the reference count on that item. The handle and reference count should be returned to the cache by a subsequent call to ReleaseEntry when the item is no longer needed.

See AddRefEntry for more details on reference counting.

See Also

Reference

CMemoryCacheBase Class
HCACHEITEM

Other Resources

CMemoryCacheBase Members
Caching Reference

CMemoryCacheBase::ReleaseEntry

Call this method to decrement the reference count of a cached item.

```
DWORD ReleaseEntry(
   const HCACHEITEM hEntry
);
```

Parameters

hEntry

The handle of the cache item whose reference count is to be decremented.

Return Value

The new reference count.

Remarks

Every call to LookupEntry and AddRefEntry should have a matching call to this method when the handle is no longer being used so that the cache has an opportunity to remove the corresponding item. Mismatched calls to these methods will result in leaks or premature removal of cached items.

See AddRefEntry for more details on reference counting.

See Also

Reference

CMemoryCacheBase Class

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase::RemoveAllEntries

Call this method to remove all the items from the cache.

```
HRESULT RemoveAllEntries( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

See RemoveEntry for details.

This method is called internally by the Uninitialize and ResetCache methods.

See Also

Reference

CMemoryCacheBase Class

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase::RemoveEntry

Call this method to remove an item from the cache given its handle.

```
HRESULT RemoveEntry(
   const HCACHEITEM hEntry
);
```

Parameters

hEntry

The handle of the item to remove from the cache.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calling this method will decrement the reference count of the item and remove it from the cache immediately. Subsequent calls to LookupEntry will fail to retrieve an item removed in this way. However, the item itself may not be cleaned up immediately. Cached items are only cleaned up when their reference count reaches zero to ensure that multiple clients can use the same item without conflicts. The reference count may reach zero as a result of a call to this method or ReleaseEntry.

See AddRefEntry for more details on reference counting.

See Also

Reference

CMemoryCacheBase Class

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase::RemoveEntryByKey

Call this method to remove an item from the cache given its key.

```
HRESULT RemoveEntryByKey(
   const keyType& Key
);
```

Parameters

hEntry

The key of the item to remove from the cache.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

See RemoveEntry for details.

See Also

Reference

CMemoryCacheBase Class

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase::ResetCache

Call this method to remove the contents of the cache and reset the statistics to zero.

```
HRESULT ResetCache( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CMemoryCacheBase Class
CMemoryCacheBase::ClearStats
CMemoryCacheBase::RemoveAllEntries

Other Resources

CMemoryCacheBase Members
Caching Reference

CMemoryCacheBase::SetMaxAllowedEntries

Call this method to set the maximum number of entries allowed in the cache.

```
HRESULT STDMETHODCALLTYPE SetMaxAllowedEntries(
   DWORD dwSize
);
```

Parameters

dwSize

The maximum number of entries allowed in the cache.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

See AddEntry and FlushEntries for information on how the limit represented by this method is used.

See Also

Reference

CMemoryCacheBase Class

CMemoryCacheBase::GetMaxAllowedEntries

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase::SetMaxAllowedSize

Call this method to set the maximum size of the entries allowed in the cache.

```
HRESULT STDMETHODCALLTYPE SetMaxAllowedSize(
   DWORD dwSize
);
```

Parameters

dwSize

The maximum size in bytes of the entries allowed in the cache.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

See AddEntry and FlushEntries for information on how the limit represented by this method is used.

See Also

Reference

CMemoryCacheBase Class

CMemoryCacheBase::GetMaxAllowedSize

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase::Uninitialize

Call this method to uninitialize the cache.

```
HRESULT Uninitialize( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Removes all entries. This method should be called before the cache object itself is deleted.

See Also

Reference

CMemoryCacheBase Class

Other Resources

CMemoryCacheBase Members

Caching Reference

CMemoryCacheBase Structures

For information about the structures in **CMemoryCacheBase**, see CMemoryCacheBase Members

CMemoryCacheBase::NodeType Structure

This protected structure holds the data for each item in the cache.

```
struct NodeType :
   public __CACHEITEM,
   public NodeInfo,
   public CFlusherCacheData,
   public CCullerCacheData
```

Remarks

This protected structure stores the user's data in the Data member. Auxiliary data used by classes derived from CMemoryCacheBase is provided by the *NodeInfo* base class. Data used by the cache flusher and cache culler is provided by the CFlusherCacheData and CCullerCacheData base classes. The reference count for the item is held in dwRef.

Requirements

Header: atlcache.h

See Also

Reference

CFlusherCacheData Structure

CCullerCacheData Structure

Other Resources

CMemoryCacheBase::NodeType Members

CMemoryCacheBase::NodeType Members

Methods

NodeType	The constructor.

Data Members

Data	The user's data.
dwRef	The reference count of the item.
dwSize	The size of the cache entry.
pos	The position of the item.

See Also

Reference

CMemoryCacheBase::NodeType Structure

CMemoryCacheBase::NodeType Methods

For information about the methods in **CMemoryCacheBase::NodeType**, see CMemoryCacheBase::NodeType Members

CMemoryCacheBase::NodeType::NodeType

The constructor.

```
NodeType( );
```

Remarks

The constructor zero-initializes dwRef, dwSize, and pos.

See Also

Reference

CMemoryCacheBase::NodeType Structure

Other Resources

CMemoryCacheBase::NodeType Members

CMemoryCacheBase::NodeType Data Members

For information about the data members in **CMemoryCacheBase::NodeType**, see CMemoryCacheBase::NodeType Members

CMemoryCacheBase::NodeType::Data

The user's data.

```
DataType Data;
```

Remarks

Corresponds to the *data* parameter of CMemoryCacheBase::AddEntry.

See the description of the CMemoryCacheBase *DataType* template parameter.

See Also

Reference

CMemoryCacheBase::NodeType Structure

Other Resources

CMemoryCacheBase::NodeType Members

CMemoryCacheBase::NodeType::dwRef

The reference count of the item.

```
DWORD dwRef;
```

Remarks

See CMemoryCacheBase::AddRefEntry and CMemoryCacheBase::RemoveEntry for more details about reference counting.

See Also

Reference

CMemoryCacheBase::NodeType Structure

Other Resources

CMemoryCacheBase::NodeType Members

CMemoryCacheBase::NodeType::dwSize

The size of the cache entry.

```
DWORD dwSize;
```

Remarks

This member corresponds to the *dwSize* parameter of CMemoryCacheBase::AddEntry.

Note that this member is used to indicate the relative cost of keeping an item in the cache. Typically this will be the size in bytes of the Data member, but that is not a requirement. CDllCache, for example, uses this member to store the size of a file on disk. Other derived classes could use this member for another measure of the cost.

CMemoryCacheBase passes the size to *StatClass*::**AddElement** and compares the sum of the sizes of the cache entries to the value passed to CMemoryCacheBase::SetMaxAllowedSize, but it never makes any attempt to interpret this value.

See Also

Reference

CMemoryCacheBase::NodeType Structure

Other Resources

CMemoryCacheBase::NodeType Members

CMemoryCacheBase::NodeType::pos

The position of the item.

```
POSITION pos;
```

Remarks

If **pos** is zero, the item has been removed from the cache and the cache is waiting for the reference count to drop to zero before the item can be deleted.

See Also

Reference

CMemoryCacheBase::NodeType Structure

Other Resources

CMemoryCacheBase::NodeType Members

CMemSession Class

This class provides an implementation of ISession that stores the session variables in memory.

```
class CMemSession :
   public ISession,
   public CComObjectRootEx<CComGlobalsThreadModel>
```

Remarks

This class is typically used with CMemSessionServiceImpl.

Requirements

Header: atlsession.h

See Also

Tasks

OnlineAddressBook Sample: Stores an Address Book in an XML Web Service

SessionSettings Sample: Session State Demonstration

Reference

ShowSession

ISession Interface

CSessionStateService Class

CMemSessionServiceImpl Class

Concepts

Session-State Tasks

Session-State Support

Session-State Services in ATL Server

Other Resources

CMemSession Members

CMemSession Members

Methods

Access	Call this method to update the last access time of the session.
BeginVariableEnum	Implementation of ISession::BeginVariableEnum.
CloseEnum	Implementation of ISession::CloseEnum.
CMemSession	The constructor.
GetCount	Implementation of ISession::GetCount.
GetNextVariable	Implementation of ISession::GetNextVariable.
GetVariable	Implementation of ISession::GetVariable.
IsExpired	Implementation of ISession::IsExpired.
RemoveAllVariables	Implementation of ISession::RemoveAllVariables.
RemoveVariable	Implementation of ISession::RemoveVariable.
SessionLock	Call this method to increment the reference count for this session and prevent it from being deleted.
SessionUnlock	Call this method to decrement the reference count for this session and allow it to be deleted.
SetTimeout	Implementation of ISession::SetTimeout.
SetVariable	Implementation of ISession::SetVariable.

See Also

Reference

CMemSession Class

Other Resources

ISession Members

CMemSession Methods

For information about the methods in **CMemSession**, see CMemSession Members

CMemSession::Access

Call this method to update the last access time of the session.

```
HRESULT Access( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CMemSession Class

Other Resources

CMemSession Members

CMemSession::BeginVariableEnum

Implementation of ISession::BeginVariableEnum.

```
STDMETHOD(BeginVariableEnum)(
   POSITION * pPOS,
   HSESSIONENUM * phEnumHandle = NULL
) throw( );
```

See Also

Reference

CMemSession Class

HSESSIONENUM

ISession::BeginVariableEnum

Other Resources

CMemSession Members

CMemSession::CloseEnum

Implementation of ISession::CloseEnum.

```
STDMETHOD(CloseEnum)(
   HSESSIONENUM /* hEnumHandle */
) throw( );
```

See Also

Reference

CMemSession Class

ISession::CloseEnum

Other Resources

CMemSession Members

CMemSession::CMemSession

The constructor.

```
CMemSession( ) throw(...);
```

See Also

Reference

CMemSession Class

Other Resources

CMemSession Members

CMemSession::GetCount

Implementation of ISession::GetCount.

```
STDMETHOD(GetCount)(
   long* pnCount
) throw( );
```

Example

See the ShowSession Sample.

See Also

Reference

CMemSession Class

ISession::GetCount

Other Resources

CMemSession Members

CMemSession::GetNextVariable

Implementation of ISession::GetNextVariable.

```
STDMETHOD(GetNextVariable)(
   POSITION * pPOS,
   VARIANT * pVal,
   HSESSIONENUM hEnum = NULL,
   LPSTR szName = NULL,
   DWORD dwLen = 0
) throw( );
```

See Also

Reference

CMemSession Class

HSESSIONENUM

ISession::GetNextVariable

Other Resources

CMemSession Members

CMemSession::GetVariable

Implementation of ISession::GetVariable.

```
STDMETHOD(GetVariable)(
   LPCSTR szName,
   VARIANT* pVal
) throw( );
```

Example

See the OnlineAddressBook Sample, the SessionSettings Sample, and the ShowSession Sample.

See Also

Reference

CMemSession Class

ISession::GetVariable

Other Resources

CMemSession Members

CMemSession::IsExpired

Implementation of ISession::IsExpired.

```
STDMETHOD(IsExpired)( ) throw( );
```

See Also

Reference

CMemSession Class

ISession::IsExpired

Other Resources

CMemSession Members

CMemSession::RemoveAllVariables

Implementation of ISession::RemoveAllVariables.

```
STDMETHOD(RemoveAllVariables)( ) throw( );
```

See Also

Reference

CMemSession Class

ISession::RemoveAllVariables

Other Resources

CMemSession Members

CMemSession::RemoveVariable

Implementation of ISession::RemoveVariable.

```
STDMETHOD(RemoveVariable)(
   LPCSTR szName
) throw( );
```

See Also

Reference

CMemSession Class

ISession::RemoveVariable

Other Resources

CMemSession Members

CMemSession::SessionLock

Call this method to increment the reference count for this session and prevent it from being deleted.

```
HRESULT SessionLock( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CMemSession Class

Other Resources

CMemSession Members

CMemSession::SessionUnlock

Call this method to decrement the reference count for this session and allow it to be deleted.

```
HRESULT SessionUnlock( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CMemSession Class

Other Resources

CMemSession Members

CMemSession::SetTimeout

Implementation of ISession::SetTimeout.

```
STDMETHOD(SetTimeout)(
   unsigned __int64 dwNewTimeout
) throw( );
```

See Also

Reference

CMemSession Class

ISession::SetTimeout

Other Resources

CMemSession Members

CMemSession::SetVariable

Implementation of ISession::SetVariable.

```
STDMETHOD(SetVariable)(
   LPCSTR szName,
   VARIANT vNewVal
) throw( );
```

See Also

Reference

CMemSession Class

ISession::SetVariable

Other Resources

CMemSession Members

CMemSessionServiceImpl Class

This class provides an in-memory based session-state implementation for use within ATL Server ISAPI extension DLLs.

```
class CMemSessionServiceImpl
```

Remarks

The **CMemSessionServiceImpl** class provides the implementation for an in-memory-based ATL Server session-state support. Used as a template argument to the CSessionStateService class, **CMemSessionServiceImpl** provides high-performance session-state support, because session-state data is kept in server memory, as opposed to in a central database. The caveat is that memory-backed session-state support works only with single-machine server scenarios. Also, excessive concurrent sessions and large session data sets, depending on available memory, may overload the server.

As an implementation class, **CMemSessionServiceImpl** serves a behind-the-scenes role and should be used in your code only as a template parameter to CSessionStateService. See Providing Session-state Services for more information.

Requirements

Header: atlsession.h

See Also

Tasks

OnlineAddressBook Sample: Stores an Address Book in an XML Web Service

SessionSettings Sample: Session State Demonstration

Reference

ShowSession

CSessionStateService Class

CDBSessionServiceImplT Class

Concepts

Session-State Tasks

Session-State Support

Session-State Services in ATL Server

Other Resources

Session-State Services (ATL Server)

ATL Server Classes

CMimeAttachment Class

This class represents an attachment in a MIME-encoded message.

```
class CMimeAttachment :
   public CMimeBodyPart
```

Remarks

This class serves as a base class for CMimeFileAttachment and CMimeRawAttachment.

Requirements

Header: atlmime.h

See Also

Reference

CMimeBodyPart Class

Other Resources

CMimeAttachment Members

ATL Server Classes

CMimeAttachment Members

Methods

CMimeAttachment	The constructor.
~CMimeAttachment	The destructor.
Copy	Call this method to create a copy of the MIME-encoded message attachment.
GetCharset	Call this method to get the character set for the MIME-encoded message attachment.
GetContentType	Call this method to get the content type from the MIME-encoded message attachment.
SetContentType	Call this method to set the content type of the MIME-encoded message attachment.
SetEncodingScheme	Call this method to set the encoding scheme of the MIME-encoded message attachment.
WriteData	Call this method to write the data for the MIME-encoded message attachment to a file.

Operators

operator =	Assigns the specified message attachment to the current attachment object.

See Also

Reference

CMimeAttachment Class

Other Resources

CMimeBodyPart Members

CMimeAttachment Methods

For information about the methods in **CMimeAttachment**, see CMimeAttachment Members.

CMimeAttachment::CMimeAttachment

The constructor.

```
CMimeAttachment( ) throw( );
```

See Also

Reference

CMimeAttachment Class

CMimeAttachment::~CMimeAttachment

Other Resources

CMimeAttachment Members

CMimeAttachment::~CMimeAttachment

The destructor.

```
virtual ~CMimeAttachment( ) throw( );
```

See Also

Reference

CMimeAttachment Class

CMimeAttachment::CMimeAttachment

Other Resources

CMimeAttachment Members

CMimeAttachment::Copy

This method is called to create a copy of the MIME-encoded message attachment.

```
virtual ATL_NOINLINE CMimeBodyPart * Copy( ) = 0;
```

Return Value

Returns a pointer to the copy of the attachment.

Remarks

Implement this method in your CMimeAttachment-derived class.

See Also

Reference

CMimeAttachment Class

Other Resources

CMimeAttachment Members

CMimeAttachment::GetCharset

Call this method to get the character set for the MIME-encoded message attachment.

```
virtual inline LPCSTR GetCharset( ) throw( );
```

Return Value

Returns the character set.

Remarks

The character set returned corresponds to the "charset" attribute parameter of the Content-Type header field of RFC 2045 (http://ietf.org/rfc/rfc2045.txt).

See Also

Reference

CMimeAttachment Class

Other Resources

CMimeAttachment Members

CMimeAttachment::GetContentType

Call this method to get the content type from the MIME-encoded message attachment.

```
virtual inline LPCSTR GetContentType( ) throw( );
```

Return Value

Returns the content type, formatted "<type>/<subtype>", as in "text/plain".

Remarks

The content type returned corresponds to the type and subtype of the Content-Type header field of RFC 2045 (http://ietf.org/rfc/rfc2045.txt).

See Also

Reference

CMimeAttachment Class

CMimeAttachment::SetContentType

Other Resources

CMimeAttachment Members

CMimeAttachment::SetContentType

Call this method to set the content type of the MIME-encoded message attachment.

```
inline BOOL SetContentType(
   LPCTSTR szContent
) throw( );
```

Parameters

szContent

The string containing the content type information. The format for the contents of this string is "<type>/<subtype>", as in "text/plain".

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The content type corresponds to the type and subtype of the Content-Type header field of RFC 2045 (http://ietf.org/rfc/rfc2045.txt).

See Also

Reference

CMimeAttachment Class
CMimeAttachment::GetContentType

Other Resources

CMimeAttachment Members

CMimeAttachment::SetEncodingScheme

Call this method to set the encoding scheme of the MIME-encoded message attachment.

```
inline BOOL SetEncodingScheme(
   int nScheme
) throw( );
```

Parameters

nScheme

The encoding scheme.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Possible values are ATLSMTP_BASE64_ENCODE (base64 encoding), ATLSMTP_UUENCODE (uuencode), and ATLSMTP_QP_ENCODE (quoted-printable).

See Also

Reference

CMimeAttachment Class

Other Resources

CMimeAttachment Members

CMimeAttachment::WriteData

This method is called to write the data for the attachment to a file.

```
virtual inline BOOL WriteData(
   HANDLE hFile,
   LPOVERLAPPED pOverlapped,
   LPCSTR szBoundary,
   DWORD dwFlags = 0
) = 0;
```

Parameters

hFile

Handle of the file to be written to.

pOverlapped

Pointer to the OVERLAPPED data structure that supplies data to be used during the overlapped (asynchronous) write operation.

szBoundary

The MIME boundary for multipart messages.

dwFlags

Contains formatting flags. Set to ATLSMTP_FORMAT_SMTP to specify SMTP format.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Classes derived from **CMimeAttachment** must implement this function.

See Also

Reference

CMimeFileAttachment Class

Other Resources

CMimeFileAttachment Members

CMimeAttachment Operators

For information about the operators in **CMimeAttachment**, see CMimeAttachment Members.

CMimeAttachment::operator =

Assigns the specified message attachment to the current attachment object.

```
const CMimeAttachment& operator =(
   const CMimeAttachment& that
) throw( ... );
```

Parameters

that

The attachment object to copy into the current object.

Return Value

Returns a reference to the current object.

See Also

Reference

CMimeAttachment Class

Other Resources

CMimeAttachment Members

CMimeBodyPart Class

This class represents a body part of a MIME-encoded message.

```
class CMimeBodyPart
```

Remarks

The **CMimeBodyPart** class is an abstract base class for the classes CMimeAttachment, CMimeText, and CMimeHeader.

Requirements

Header: atlmime.h

See Also

Tasks

Mailer Sample: Demonstrates SMTP Mail Classes

Other Resources

CMimeBodyPart Members

ATL Server Classes

CMimeBodyPart Members

Methods

~CMimeBodyPart	The destructor.
Copy	Call this method to copy a body part of a MIME-encoded message.
GetCharset	Call this method to get the character set used in a body part of a MIME-encoded message.
GetContentType	Call this method to get the content type for a body part of a MIME-encoded message.
WriteData	Call this method to write the data for a MIME-encoded message body part to a file.

See Also

Reference

CMimeBodyPart Class

CMimeBodyPart Methods

For information about the methods in **CMimeBodyPart**, see CMimeBodyPart Members.

CMimeBodyPart::~CMimeBodyPart

The destructor.

```
virtual ~CMimeBodyPart( ) = 0;
```

See Also

Reference

CMimeBodyPart Class

Other Resources

CMimeBodyPart Members

CMimeBodyPart::Copy

This method is called to copy a body part of a MIME-encoded message.

```
virtual CMimeBodyPart * Copy( ) = 0;
```

Return Value

Returns a pointer to the copy of the body part that is created.

See Also

Reference

CMimeBodyPart Class

Other Resources

CMimeBodyPart Members

CMimeBodyPart::GetCharset

Call this method to get the character set used in a body part of a MIME-encoded message.

```
virtual LPCSTR GetCharset( );
```

Return Value

Returns a string containing the character set information.

Remarks

The character set returned corresponds to the "charset" attribute parameter of the Content-Type header field of RFC 2045 (http://ietf.org/rfc/rfc2045.txt).

See Also

Reference

CMimeBodyPart Class

Other Resources

CMimeBodyPart Members

CMimeBodyPart::GetContentType

This method is called to get the content type for a body part of a MIME-encoded message.

```
virtual LPCSTR GetContentType( ) = 0;
```

Return Value

Returns a string containing the content type information. The format for the contents of this string is "<type>/<subtype>", as is "text/plain".

Remarks

Returns TRUE on success, FALSE on failure.

See Also

Reference

CMimeBodyPart Class

Other Resources

CMimeBodyPart Members

CMimeBodyPart::WriteData

This method is called to write the data for the body part to a file.

```
virtual BOOL WriteData(
   HANDLE hFile,
   LPOVERLAPPED pOverlapped,
   LPCSTR szBoundary,
   DWORD dwFlags = 0
) = 0;
```

Parameters

hFile

Handle of the file to be written to.

pOverlapped

Pointer to the OVERLAPPED data structure that supplies data to be used during the overlapped (asynchronous) write operation.

szBoundary

The MIME boundary for multipart messages.

dwFlags

Contains formatting flags. Set to ATLSMTP_FORMAT_SMTP to specify SMTP format.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CMimeFileAttachment Class

Other Resources

CMimeFileAttachment Members

CMimeFileAttachment Class

This class represents a file attachment in a MIME-encoded message.

```
class CMimeFileAttachment :
   public CMimeAttachment
```

Remarks

This class derives from CMimeAttachment.

Requirements

Header: atlmime.h

See Also

Other Resources

CMimeFileAttachment Members

ATL Server Classes

CMimeFileAttachment Members

Methods

CMimeFileAttachment	The constructor.
Copy	Call this method to create a copy of the file attachment.
Initialize	Call this method to initialize the file attachment to a MIME-encoded message.
WriteData	Call this method to write the data for the file attachment to a file.

Operators

operator =	Assigns the specified file attachment to the current attachment object.

See Also

Reference

CMimeFileAttachment Class

Other Resources

CMimeAttachment Members

CMimeFileAttachment Methods

For information about the methods in **CMimeFileAttachment**, see CMimeFileAttachment Members.

CMimeFileAttachment::CMimeFileAttachment

The constructor.

```
CMimeFileAttachment( ) throw( );
```

See Also

Reference

CMimeFileAttachment Class

Other Resources

CMimeFileAttachment Members

CMimeFileAttachment::Copy

This method is called to create a copy of the file attachment.

```
virtual ATL_NOINLINE CMimeBodyPart* Copy( ) throw(...);
```

Return Value

Returns a pointer to the copy of the attachment.

See Also

Reference

CMimeFileAttachment Class

Other Resources

CMimeFileAttachment Members

CMimeFileAttachment::Initialize

Call this method to initialize the file attachment to a MIME-encoded message.

```
inline BOOL Initialize(
   LPCTSTR szFileName,
   LPCTSTR szDisplayName = NULL,
   IMultiLanguage* pMultiLanguage = NULL,
   UINT uiCodePage = 0
) throw( );
```

Parameters

szFileName

The file name for the attachment.

szDisplayName

The display name for the attachment.

pMultiLanguage

Pointer to the IMultiLanguage interface containing methods that perform code page to character set conversion.

uiCodePage

The code page.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CMimeFileAttachment Class

Other Resources

CMimeFileAttachment Members

CMimeFileAttachment::WriteData

Call this method to write the data for the file attachment to a file.

```
virtual inline BOOL WriteData(
   HANDLE hFile,
   LPOVERLAPPED pOverlapped,
   LPCSTR szBoundary,
   DWORD dwFlags = 0
) throw();
```

Parameters

hFile

Handle of the file to be written to.

pOverlapped

Pointer to the OVERLAPPED data structure that supplies data to be used during the overlapped (asynchronous) write operation.

szBoundary

The MIME boundary for multipart messages.

dwFlags

Contains formatting flags. Set to ATLSMTP_FORMAT_SMTP to specify SMTP format.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CMimeFileAttachment Class

Other Resources

CMimeFileAttachment Members

CMimeFileAttachment Operators

For information about the operators in **CMimeFileAttachment**, see CMimeFileAttachment Members.

CMimeFileAttachment::operator =

Call this method to assign the specified file attachment to the current attachment object.

```
const CMimeFileAttachment& operator =(
   const CMimeFileAttachment& that
) throw( ... );
```

Parameters

that

The attachment object to copy into the current object.

Return Value

Returns a reference to the current object.

See Also

Reference

CMimeFileAttachment Class

Other Resources

CMimeFileAttachment Members

CMimeHeader Class

This class represents a MIME-encoded message header.

```
class CMimeHeader :
   public CMimeBodyPart
```

Remarks

This class describes a basic RFC 822 (http://ietf.org/rfc/rfc0822.txt) message header. It also serves as the base class for the CMimeMessage class.

Requirements

Header: atlmime.h

See Also

Tasks

MantaWeb Sample: Online Personal Information Management

Mailer Sample: Demonstrates SMTP Mail Classes

Reference

ConfirmUser

CMimeBodyPart Class

Other Resources

CMimeHeader Members

ATL Server Classes

CMimeHeader Members

Methods

AddBcc	Call this method to add a Bcc recipient to a MIME-encoded message header.
AddCc	Call this method to add a Cc recipient to a MIME-encoded message header.
AddRecipient	Call this method to add a recipient to a MIME-encoded message header.
AppendUserDefinedHeader	Call this method to append a user-defined header.
CMimeHeader	The constructor.
~CMimeHeader	The destructor.
ClearBcc	Call this method to clear the Bcc recipients list of a MIME-encoded message header.
ClearCc	Call this method to clear the Cc recipients list of a MIME-encoded message header.
ClearRecipients	Call this method to clear the recipients list of a MIME-encoded message header.
Copy	Call this method to copy a MIME-encoded message header.
GetBcc	Call this method to get the Bcc recipients list of a MIME-encoded message header.
GetCc	Call this method to get the Cc recipients list of a MIME-encoded message header.
GetCharset	Call this method to get the character set for a MIME-encoded message header.
GetContentType	Call this method to get the content type for a MIME-encoded message header.
GetPriority	Call this method to get the priority from a MIME-encoded message header.
GetRecipients	Call this method to get the list of recipients on the To line of a MIME-encoded message header.
GetRecipientsString	Call this method to get a list of all recipients on the To, Cc, and Bcc lines of a MIME-encoded message header.
GetRequiredRecipientsStringLength	Call this method to get the required recipients' string length.
GetSender	Call this method to get the e-mail address for the sender of a MIME-encoded message header.
GetSenderName	Call this method to get the display name for the sender of a MIME-encoded message header.
GetSubject	Call this method to get the subject of a MIME-encoded message header.
Initialize	Call this method to initialize a MIME-encoded message header.
SetPriority	Call this method to set the priority of a MIME-encoded message header.

SetSender	Call this method to set the e-mail address for the sender of a MIME-encoded message header.
SetSenderName	Call this method to set the display name for the sender of a MIME-encoded message header.
SetSubject	Call this method to set the subject of a MIME-encoded message header.
WriteData	Call this method to write the data for a MIME-encoded message header to a file.

Operators

operator =	Assigns the specified MIME-encoded message header to the current header object.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeBodyPart Members

CMimeHeader Methods

For information about the methods in **CMimeHeader**, see CMimeHeader Members.

CMimeHeader::AddBcc

Call this method to add a Bcc recipient to a MIME-encoded message header.

```
inline BOOL AddBcc(
   LPCTSTR szAddress
) throw();
```

Parameters

szAddress

The address of the recipient.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The contents of the Bcc recipients list are removed when the message is delivered.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::AddCc

Call this method to add a Cc recipient to a MIME-encoded message header.

```
inline BOOL AddCc(
   LPCTSTR szAddress,
   LPCTSTR szName = NULL,
   UINT uiCodePage = 0
) throw();
```

Parameters

szAddress

The address of the recipient.

szName

The display name of the recipient.

uiCodePage

The code page.

Return Value

Returns TRUE on success, FALSE on failure.

Example

See the Mailer Sample.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::AddRecipient

Call this method to add a recipient to a MIME-encoded message header.

```
inline BOOL AddRecipient(
   LPCTSTR szAddress,
   LPCTSTR szName = NULL,
   UINT uiCodePage = 0
) throw();
```

Parameters

szAddress

The address of the recipient.

szName

The display name of the recipient.

uiCodePage

The code page.

Return Value

Returns TRUE on success, FALSE on failure.

Example

See the MantaWeb Sample, the Mailer Sample, and the ConfirmUser Sample.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::AppendUserDefinedHeader

Call this method to append a user-defined header.

```
inline BOOL AppendUserDefinedHeader(
   LPCTSTR szHeaderName,
   LPCTSTR szHeader,
   UINT uiCodePage = 0
) throw( );
```

Parameters

szHeaderName

The header name.

szHeader

The header text.

uiCodePage

The code page.

Remarks

The header must not contain a CR/LF.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::CMimeHeader

The constructor.

```
CMimeHeader( ) throw( );
```

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::~CMimeHeader

The destructor.

```
~CMimeHeader( ) throw( );
```

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::ClearBcc

Call this method to clear the Bcc recipients list of a MIME-encoded message header.

```
inline BOOL ClearBcc( ) throw( );
```

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::ClearCc

Call this method to clear the Cc recipients list of a MIME-encoded message header.

```
inline BOOL ClearCc( ) throw( );
```

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::ClearRecipients

Call this method to clear the recipients list of a MIME-encoded message header.

```
inline BOOL ClearRecipients( ) throw( );
```

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::Copy

Call this method to copy a MIME-encoded message header.

```
virtual ATL_NOINLINE CMimeBodyPart* Copy( ) throw( ... );
```

Return Value

Returns a pointer to the copy of the message header that is created.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::GetBcc

Call this method to get the Bcc recipients list of a MIME-encoded message header.

```
inline LPCSTR GetBcc( ) throw( );
```

Return Value

Returns a string containing a list of comma-separated Bcc recipients.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::GetCc

Call this method to get the Cc recipients list of a MIME-encoded message header.

```
inline LPCSTR GetCc( ) throw( );
```

Return Value

Returns a string containing a list of comma-separated Cc recipients.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::GetCharset

Call this method to get the character set for a MIME-encoded message header.

```
virtual inline LPCSTR GetCharset( ) throw( );
```

Return Value

Returns a string containing character set information.

Remarks

The character set returned corresponds to the "charset" attribute parameter of the Content-Type header field of RFC 2045 (http://ietf.org/rfc/rfc2045.txt).

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::GetContentType

Call this method to get the content type for a MIME-encoded message header.

```
virtual inline LPCSTR GetContentType( ) throw( );
```

Return Value

Returns a string containing the content type information. The format for the contents of this string is "<type>/<subtype>", as in "text/plain".

Remarks

The content type returned corresponds to the type and subtype of the Content-Type header field of RFC 2045 (http://ietf.org/rfc/rfc2045.txt).

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::GetPriority

Call this method to get the priority from a MIME-encoded message header.

```
inline ATL_MIME_PRIORITY GetPriority() throw();
```

Return Value

Returns an ATL_MIME_PRIORITY value (ATL_MIME_HIGH_PRIORITY, ATL_MIME_NORMAL_PRIORITY, ATL_MIME_LOW_PRIORITY, or ATL_MIME_PRIORITY_ERROR) that indicates whether the message header is marked as high, normal, or low priority.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::GetRecipients

Call this method to get the list of recipients on the To line of a MIME-encoded message header.

```
inline LPCSTR GetRecipients( ) throw( );
```

Return Value

Returns a string containing a list of recipients.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::GetRecipientsString

Call this method to get a list of all recipients on the To, Cc, and Bcc lines of a MIME-encoded message header.

```
ATL_NOINLINE BOOL GetRecipientsString(
   LPSTR szRecip,
   LPDWORD pdwLen
) throw();
```

Parameters

szRecip

The string containing a list of recipients.

pdwLen

The length of the string.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::GetRequiredRecipientsStringLength

Call this method to get the required recipients' string length.

```
inline DWORD GetRequiredRecipientsStringLength( ) throw( );
```

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::GetSender

Call this method to get the e-mail address for the sender of a MIME-encoded message header.

```
inline LPCSTR GetSender( ) throw( );
```

Return Value

Returns the e-mail address of the sender.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::GetSenderName

Call this method to get the display name for the sender of a MIME-encoded message header.

```
inline LPCSTR GetSenderName( ) throw( );
```

Return Value

Returns the display name of the sender.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::GetSubject

Call this method to get the subject of a MIME-encoded message header.

```
inline LPCSTR GetSubject( ) throw( );
```

Return Value

Returns the subject of the message.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::Initialize

Call this method to initialize a MIME-encoded message header.

```
inline BOOL Initialize(
   IMultiLanguage* pMultiLanguage = NULL
) throw( );
```

Parameters

pMultiLanguage

Pointer to the IMultiLanguage interface containing methods that perform code page to character set conversion.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::SetPriority

Call this method to set the priority of a MIME-encoded message header.

```
inline BOOL SetPriority(
   ATL_MIME_PRIORITY nPriority
) throw();
```

Parameters

nPriority

The ATL_MIME_PRIORITY value (ATL_MIME_HIGH_PRIORITY, ATL_MIME_NORMAL_PRIORITY, ATL_MIME_LOW_PRIORITY, or ATL_MIME_PRIORITY_ERROR) that indicates whether the message header is marked as high, normal, or low priority.

Return Value

Returns TRUE on success, FALSE on failure.

Example

See the Mailer Sample.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::SetSender

Call this method to set the e-mail address for the sender of a MIME-encoded message header.

```
inline BOOL SetSender(
   LPCTSTR szSender
) throw();
```

Parameters

szSender

The e-mail address of the sender.

Return Value

Returns TRUE on success, FALSE on failure.

Example

See the MantaWeb Sample, the Mailer Sample, and the ConfirmUser Sample.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::SetSenderName

Call this method to set the display name for the sender of a MIME-encoded message header.

```
inline BOOL SetSenderName(
   LPCTSTR szName,
   UINT uiCodePage = 0
) throw();
```

Parameters

szName

The display name of the sender.

uiCodePage

The code page.

Return Value

Returns TRUE on success, FALSE on failure.

Example

See the MantaWeb Sample and the Mailer Sample.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::SetSubject

Call this method to set the subject of a MIME-encoded message header.

```
inline BOOL SetSubject(
   LPCTSTR szSubject,
   UINT uiCodePage = 0
) throw();
```

Parameters

szSubject

The subject of the message.

uiCodePage

The code page.

Return Value

Returns TRUE on success, FALSE on failure.

Example

See the MantaWeb Sample, the Mailer Sample, and the ConfirmUser Sample.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader::WriteData

Call this method to write the data for a MIME-encoded message header to a file.

```
virtual inline BOOL WriteData(
   HANDLE hFile,
   LPOVERLAPPED pOverlapped,
   LPCSTR /*szBoundary*/,
   DWORD dwFlags = 0
) throw();
```

Parameters

hFile

Handle to the file to be written to.

pOverlapped

Pointer to the OVERLAPPED data structure that supplies data to be used during the overlapped (asynchronous) write operation.

szBoundary

Not used.

dwFlags

Contains formatting flags. Set to ATLSMTP_FORMAT_SMTP to specify SMTP format.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeHeader Operators

For information about the operators in **CMimeHeader**, see CMimeHeader Members.

CMimeHeader::operator =

Assigns the specified MIME-encoded message header to the current header object.

```
const CMimeHeader& operator =(
   const CMimeHeader& that
) throw( ... );
```

Parameters

that

The message header to copy into the current object.

Return Value

Returns a reference to the current object.

See Also

Reference

CMimeHeader Class

Other Resources

CMimeHeader Members

CMimeMessage Class

This class represents a MIME-encoded message.

```
class CMimeMessage :
   public CMimeHeader
```

Requirements

Header: atlmime.h

See Also

Tasks

Mailer Sample: Demonstrates SMTP Mail Classes

MantaWeb Sample: Online Personal Information Management

Reference

ConfirmUser

ConfirmUser

CMimeHeader Class

Other Resources

CMimeMessage Members

ATL Server Classes

CMimeMessage Members

Methods

AddText	Call this method to add text to a MIME-encoded message.
AttachFile	Call this method to attach a file to a MIME-encoded message.
AttachMessage	Call this method to attach a MIME-encoded message to the current MIME-encoded message.
AttachRaw	Call this method to attach raw data to a MIME-encoded message.
CMimeMessage	The constructor.
~CMimeMessage	The destructor.
Copy	Call this method to copy a MIME-encoded message.
RemoveParts	Call this method to delete all body parts in a MIME-encoded message.
SetDisplayName	Call this method to set the display name of the MIME-encoded message.
WriteData	Call this method to write the data for a MIME-encoded message to a file.

Operators

operator =	Assigns the specified MIME-encoded message to the current message object.

See Also

Reference

CMimeMessage Class

Other Resources

CMimeHeader Members

CMimeMessage Methods

For information about the methods in **CMimeMessage**, see CMimeMessage Members.

CMimeMessage::AddText

Call this method to add text to a MIME-encoded message.

```
inline BOOL AddText(
   LPCTSTR szText,
   int nTextLen = -1,
   int nPos = 1,
   UINT uiCodePage = 0
) throw();
```

Parameters

szText

The text to add.

nTextLen

The size of the text to add in bytes. If this value is less than 0, the text size will be computed with _tcslen.

nPos

The position in the message at which to insert the text.

uiCodePage

The code page.

Return Value

Returns TRUE on success, FALSE on failure.

Example

See the MantaWeb Sample, the Mailer Sample, and the ConfirmUser Sample.

See Also

Reference

CMimeMessage Class

Other Resources

CMimeMessage Members

CMimeMessage::AttachFile

Call this method to attach a file to a MIME-encoded message.

```
inline BOOL AttachFile(
   LPCTSTR szFileName,
   LPCTSTR szDisplayName = NULL,
   LPCTSTR szContentType = NULL,
   int nEncodingScheme = ATLSMTP_BASE64_ENCODE,
   UINT uiCodepage = 0
);
```

Parameters

szFileName

The name of the file to attach to the message.

szDisplayName

The name to display with the attachment's placeholder in the message.

szContentType

The content type.

nEncodingScheme

The encoding scheme to use for the attachment (ATLSMTP_BASE64_ENCODE, ATLSMTP_UUENCODE_ENCODE, or ATLSMTP_QP_ENCODE).

uiCodepage

The code page.

Return Value

Returns TRUE on success, FALSE on failure.

Example

See the Mailer Sample.

See Also

Reference

CMimeMessage Class

Other Resources

CMimeMessage Members

CMimeMessage::AttachMessage

Call this method to attach a MIME-encoded message to the current MIME-encoded message.

```
inline BOOL AttachMessage(
   CMimeMessage* pMsg
) throw( ... );
```

Parameters

pMsg

The message to attach.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CMimeMessage Class

Other Resources

CMimeMessage Members

CMimeMessage::AttachRaw

Call this method to attach raw data to a MIME-encoded message.

```
inline BOOL AttachRaw(
   void* pRawData,
   DWORD dwDataLength,
   int nEncodingScheme = ATLSMTP_BASE64_ENCODE,
   BOOL bCopyData = TRUE,
   LPCTSTR szDisplayName = NULL,
   LPCTSTR szContentType = _T("application/octet-stream"),
   UINT uiCodepage = 0
);
```

Parameters

pRawData

The data to attach.

dwDataLength

The size of the data in bytes.

nEncodingScheme

The encoding scheme to use for the attachment (ATLSMTP_BASE64_ENCODE, ATLSMTP_UUENCODE_ENCODE, or ATLSMTP_QP_ENCODE).

bCopyData

Set to TRUE to make a copy of the data.

szDisplayName

The name to display with the attachment's placeholder in the message.

szContentType

The content type.

uiCodepage

The code page.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CMimeMessage Class

Other Resources

CMimeMessage Members

CMimeMessage::CMimeMessage

The constructor.

```
CMimeMessage(
   IMultiLanguage *pMultiLanguage = NULL
) throw();
```

Parameters

pMultiLanguage

See IMultiLanguage Interface.

See Also

Reference

CMimeMessage Class

Other Resources

CMimeMessage Members

CMimeMessage::~CMimeMessage

The destructor.

```
virtual ~CMimeMessage( ) throw( );
```

See Also

Reference

CMimeMessage Class

Other Resources

CMimeMessage Members

CMimeMessage::Copy

Call this method to copy a MIME-encoded message.

```
virtual ATL_NOINLINE CMimeBodyPart* Copy() throw( ... );
```

Return Value

Returns a pointer to the copy of the message that is created.

See Also

Reference

CMimeMessage Class

Other Resources

CMimeMessage Members

CMimeMessage::RemoveParts

Call this method to delete all body parts in a MIME-encoded message.

```
void RemoveParts( ) throw( );
```

See Also

Reference

CMimeMessage Class

Other Resources

CMimeMessage Members

CMimeMessage::SetDisplayName

Call this method to set the display name of the MIME-encoded message.

```
inline BOOL SetDisplayName(
   LPCTSTR szDisplayName
) throw();
```

Parameters

szDisplayName

The display name of the message.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CMimeMessage Class

Other Resources

CMimeMessage Members

CMimeMessage::WriteData

Call this method to write the data for a MIME-encoded message to a file.

```
virtual BOOL WriteData(
   HANDLE hFile,
   LPOVERLAPPED pOverlapped,
   LPCSTR szBoundary = NULL,
   DWORD dwFlags = 0
) throw();
```

Parameters

hFile

Handle to the file to be written to.

pOverlapped

Pointer to the OVERLAPPED data structure that supplies data to be used during the overlapped (asynchronous) write operation.

szBoundary

The MIME boundary for multipart messages.

dwFlags

Contains formatting flags. Set to ATLSMTP_FORMAT_SMTP to specify SMTP format.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CMimeMessage Class

Other Resources

CMimeMessage Members

CMimeMessage Operators

For information about the operators in **CMimeMessage**, see CMimeMessage Members.

CMimeMessage::operator =

Assigns the specified MIME-encoded message to the current message object.

```
const CMimeMessage& operator=(
   const CMimeMessage& that
) throw( ... );
```

Parameters

that

The message to copy into the current object.

Return Value

Returns a reference to the current object.

See Also

Reference

CMimeMessage Class

Other Resources

CMimeMessage Members

CMimeRawAttachment Class

This class represents an attachment of raw data in a MIME-encoded message.

```
class CMimeRawAttachment :
   public CMimeAttachment
```

Remarks

This class derives from CMimeAttachment.

Requirements

Header: atlmime.h

See Also

Other Resources

CMimeRawAttachment Members

ATL Server Classes

CMimeRawAttachment Members

Methods

CMimeRawAttachment	The constructor.
~CMimeRawAttachment	The destructor.
Copy	Call this method to create a copy of the raw data attachment.
Initialize	Call this method to initialize the raw data attachment to a MIME-encoded message.
WriteData	Call this method to write the data for the raw data attachment to a file.

Operators

operator =	Assigns the specified raw data attachment to the current attachment object.

See Also

Reference

CMimeRawAttachment Class

Other Resources

CMimeAttachment Members

CMimeRawAttachment Methods

For information about the methods in **CMimeRawAttachment**, see CMimeRawAttachment Members.

CMimeRawAttachment::CMimeRawAttachment

The constructor.

```
CMimeRawAttachment( ) throw( );
```

See Also

Reference

CMimeRawAttachment Class

Other Resources

CMimeRawAttachment Members

CMimeRawAttachment::~CMimeRawAttachment

The destructor.

```
~CMimeRawAttachment( ) throw( );
```

See Also

Reference

CMimeRawAttachment Class

Other Resources

CMimeRawAttachment Members

CMimeRawAttachment::Copy

Call this method to create a copy of the raw data attachment.

```
virtual ATL_NOINLINE CMimeBodyPart* Copy() throw( ... );
```

Return Value

Returns a pointer to the copy of the attachment.

See Also

Reference

CMimeRawAttachment Class

Other Resources

CMimeRawAttachment Members

CMimeRawAttachment::Initialize

Call this method to initialize the raw data attachment to a MIME-encoded message.

```
inline BOOL Initialize(
   void* pData,
   DWORD nDataLength,
   BOOL bCopyData = TRUE,
   LPCTSTR szDisplayName = NULL,
   IMultiLanguage* pMultiLanguage = NULL,
   UINT uiCodePage = 0
) throw( );
```

Parameters

pData

The data.

nDataLength

The number of bytes in the data.

bCopyData

Flag specifying whether to make a copy of the data. If you don't make a copy of the data, the user is responsible for managing the data, and the data must exist as long as this class has a pointer to it.

szDisplayName

The display name for the attachment.

pMultiLanguage

Pointer to the IMultiLanguage interface containing methods that perform code page to character set conversion.

uiCodePage

The code page.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CMimeRawAttachment Class

Other Resources

CMimeRawAttachment Members

CMimeRawAttachment::WriteData

Call this method to dump the data for the raw data attachment to a file.

```
virtual inline BOOL WriteData(
   HANDLE hFile,
   LPOVERLAPPED pOverlapped,
   LPCSTR szBoundary,
   DWORD dwFlags = 0
) throw( );
```

Parameters

hFile

Handle to the file to be written to.

pOverlapped

Pointer to the OVERLAPPED data structure that supplies data to be used during the overlapped (asynchronous) write operation.

szBoundary

The MIME boundary for multipart messages.

dwFlags

Contains formatting flags. Set to ATLSMTP_FORMAT_SMTP to specify SMTP format.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CMimeRawAttachment Class

Other Resources

CMimeRawAttachment Members

CMimeRawAttachment Operators

For information about the operators in **CMimeRawAttachment**, see CMimeRawAttachment Members.

CMimeRawAttachment::operator =

Assigns the specified raw data attachment to the current attachment object.

```
const CMimeRawAttachment& operator =(
   const CMimeRawAttachment& that
) throw( ... );
```

Parameters

that

The attachment object to copy into the current object.

Return Value

Returns a reference to the current object.

See Also

Reference

CMimeRawAttachment Class

Other Resources

CMimeRawAttachment Members

CMimeText Class

This class represents the body text in a MIME-encoded message.

```
class CMimeText :
   public CMimeBodyPart
```

Requirements

Header: atlmime.h

See Also

Reference

CMimeBodyPart Class

Other Resources

CMimeText Members

ATL Server Classes

CMimeText Members

Methods

CMimeText	The constructor.
~CMimeText	The destructor.
Copy	Call this method to create a copy of the body text of a MIME-encoded message.
GetCharset	Call this method to get the character set for the body text of a MIME-encoded message.
GetContentType	Call this method to get the content type from the body text of a MIME-encoded message.
Initialize	Call this method to initialize the body text of a MIME-encoded message.
WriteData	Call this method to write the data for the body text of a MIME-encoded message to a file.

Operators

operator =	Assigns the specified body text to the current text object.

See Also

Reference

CMimeText Class

Other Resources

CMimeBodyPart Members

CMimeText Methods

For information about the methods in **CMimeText**, see CMimeText Members.

CMimeText::CMimeText

The constructor.

```
CMimeText( ) throw( );
```

See Also

Reference

CMimeText Class

Other Resources

CMimeText Members

CMimeText::~CMimeText

The destructor.

```
virtual ~CMimeText( ) throw( );
```

See Also

Reference

CMimeText Class

Other Resources

CMimeText Members

CMimeText::Copy

Call this method to create a copy of the body text of a MIME-encoded message.

```
virtual ATL_NOINLINE CMimeBodyPart* Copy() throw( ... );
```

Return Value

Returns a pointer to the copy of the body text.

See Also

Reference

CMimeText Class

Other Resources

CMimeText Members

CMimeText::GetCharset

Call this method to get the character set for the body text of a MIME-encoded message.

```
virtual inline LPCSTR GetCharset( ) throw( );
```

Return Value

Returns a string containing the character set information.

Remarks

The character set returned corresponds to the "charset" attribute parameter of the Content-Type header field of RFC 2045 (http://ietf.org/rfc/rfc2045.txt).

See Also

Reference

CMimeText Class

Other Resources

CMimeText Members

CMimeText::GetContentType

Call this method to get the content type from the body text of a MIME-encoded message.

```
virtual inline LPCSTR GetContentType( ) throw( );
```

Return Value

Returns a string containing the content type information. The format for the contents of this string is "<type>/<subtype>", as in "text/plain".

Remarks

The content type returned corresponds to the type and subtype of the Content-Type header field of RFC 2045 (http://ietf.org/rfc/rfc2045.txt).

See Also

Reference

CMimeText Class

Other Resources

CMimeText Members

CMimeText::Initialize

Call this method to initialize the body text of a MIME-encoded message.

```
inline BOOL Initialize(
   LPCTSTR szText,
   int nTextLen = -1,
   IMultiLanguage* pMultiLanguage = NULL,
   UINT uiCodePage = 0
) throw( );
```

Parameters

szText

The body text.

nTextLen

The number of bytes in the text. If this value is less than 0, the length will be computed with _tcslen.

pMultiLanguage

Pointer to the IMultiLanguage interface containing methods that perform code page to character set conversion.

uiCodePage

The code page.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CMimeText Class

Other Resources

CMimeText Members

CMimeText::WriteData

Call this method to write the data for the body text of a MIME-encoded message to a file.

```
virtual inline BOOL WriteData(
   HANDLE hFile,
   LPOVERLAPPED pOverlapped,
   LPCSTR szBoundary,
   DWORD dwFlags = 0
) throw( );
```

Parameters

hFile

Handle to the file to be written to.

pOverlapped

Pointer to the OVERLAPPED data structure that supplies data to be used during the overlapped (asynchronous) write operation.

szBoundary

The MIME boundary for multipart messages.

dwFlags

Contains formatting flags. Set to ATLSMTP_FORMAT_SMTP to specify SMTP format.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CMimeText Class

Other Resources

CMimeText Members

CMimeText Operators

For information about the operators in **CMimeText**, see CMimeText Members.

CMimeText::operator =

Call this method to assign the specified body text to the current text object.

```
const CMimeText& operator =(
   const CMimeText& that
) throw( ... );
```

Parameters

that

The body text object to copy into the current object.

Return Value

Returns a reference to the current object.

See Also

Reference

CMimeText Class

Other Resources

CMimeText Members

CMultiPartFormParser Class

Use this class to read multipart/form-data from the associated server context and generate files as necessary from the data in the body of the request.

```
class CMultiPartFormParser
```

Remarks

Pass a server context to the constructor of this class, then call CMultiPartFormParser::GetMultiPartData to parse the body of the request.

Requirements

Header: atlisapi.h

See Also

Other Resources

CMultiPartFormParser Members

ATL Server Classes

CMultiPartFormParser Members

Methods

CMultiPartFormParser	The constructor.
~CMultiPartFormParser	The destructor.
GetMultiPartData	Call this method to read multipart/form-data from the current HTTP request.

Typedefs

FILEMAPTYPE	The type of the collection to which pointers to objects implementing IHttpFile will be added if there are files in the current request.

See Also

Reference

CMultiPartFormParser Class

CMultiPartFormParser Methods

For information about the methods in **CMultiPartFormParser**, see CMultiPartFormParser Members.

CMultiPartFormParser::CMultiPartFormParser

The constructor.

```
CMultiPartFormParser(
   IHttpServerContext* pServerContext
) throw( );
```

Parameters

pServerContext

The server context.

Remarks

The constructor stores the server context to get access to information about the current request.

See Also

Reference

CMultiPartFormParser Class

Other Resources

CMultiPartFormParser Members

CMultiPartFormParser::~CMultiPartFormParser

The destructor.

```
~CMultiPartFormParser( ) throw( );
```

Remarks

The destructor frees resources allocated by this object.

See Also

Reference

CMultiPartFormParser Class

Other Resources

CMultiPartFormParser Members

CMultiPartFormParser::GetMultiPartData

Call this method to read multipart/form-data from the current HTTP request.

```
ATL_NOINLINE BOOL GetMultiPartData(
   FILEMAPTYPE& Files,
   CHttpRequestParams* pQueryParams,
   DWORD dwFlags = ATL_FORM_FLAG_NONE
) throw( );
```

Parameters

Files

The collection to which pointers to objects implementing IHttpFile will be added if there are files in the current request.

pQueryParams

The collection to which form fields will be added if present in the current request.

dwFlags

The form flags describing how form processing should be carried out.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Forms can be sent to a web server using one of two encodings: application/x-www-form-urlencoded or multipart/form-data. In addition to the simple name-value pairs typically associated with application/x-www-form-urlencoded form data, multipart/form-data (as described in RFC 2388 (http://ietf.org/rfc/rfc2388.txt)) can also contain files to be uploaded to the web server.

This method will generate a physical file for each file contained in the multipart/form-data request body. The generated files are stored in the server's temporary directory as returned by GetTempPath and are named using GetTempFileName.

The information about each file can be obtained from the elements of the *Files* collection. You can retrieve the original name of the file on the client, the name of the generated file on the server, the MIME content type of the uploaded file, the name of the form field associated with that file, and the size in bytes of the file. All this information is exposed by the IHttpFile interface provided by each object in the array.

In addition to generating files and populating the *Files* collection with information about them, this method also populates the *pQueryParams* collection with the names and values of the other form fields contained in the current request. The file fields are also added to this array. The value of these fields is the full name of the generated file on the server.

Note that files can be generated even if this method returns FALSE unless you specify either **ATL_FORM_FLAG_IGNORE_FILES** or **ATL_FORM_FLAG_REFUSE_FILES** in *dwFlags*. If you don't specify one of these flags, you should always check the *Files* array for generated files and delete any that are no longer needed to prevent your server from running out of disk space.

See Also

Reference

CMultiPartFormParser Class
CMultiPartFormParser::FILEMAPTYPE
ATL_FORM_FLAGS

Other Resources

CMultiPartFormParser Members

CMultiPartFormParser Typedefs

For information about the typedefs in **CMultiPartFormParser**, see CMultiPartFormParser Members.

CMultiPartFormParser::FILEMAPTYPE

The type of the collection to which pointers to objects implementing IHttpFile will be added if there are files in the current request.

```
typedef CHttpMap<CStringA, IHttpFile*, CStringElementTraits<CStringA> > FILEMAPTYPE;
```

See Also

Reference

CMultiPartFormParser Class

CHttpMap Class

Other Resources

CMultiPartFormParser Members

CNoDllCachePeer Class

This class conforms to the DLL cache peer archetype and can be used when no extra data needs to be stored with the DLLs in the cache.

```
class CNoDllCachePeer
```

Remarks

This class can be used with CDllCache when no extra data needs to be stored with the DLLs in the cache. The Add and Remove methods do nothing and DllInfo is an empty structure.

Requirements

Header: atlcache.h

See Also

Other Resources

CNoDllCachePeer Members

Caching Reference

CNoDllCachePeer Members

Methods

Add	Implementation of DllCachePeerArchetype::Add.
Remove	Implementation of DllCachePeerArchetype::Remove.

Structures

DllInfo	Implementation of DllCachePeerArchetype::DllInfo.

See Also

Reference

CNoDllCachePeer Class

Other Resources

Caching

Caching Reference

CNoDllCachePeer Methods

For information about the methods in **CNoDllCachePeer**, see CNoDllCachePeer Members.

CNoDllCachePeer::Add

Implementation of DllCachePeerArchetype::Add.

```
BOOL Add(
   HINSTANCE /* hInst */,
   DllInfo * /* pInfo */
);
```

Return Value

Always returns **TRUE**.

Remarks

This method does nothing when a DLL is added to the cache.

See Also

Reference

CNoDllCachePeer Class

Other Resources

CNoDllCachePeer Members

Caching Reference

CNoDllCachePeer::Remove

Implementation of DllCachePeerArchetype::Remove.

```
void Remove(
   HINSTANCE /* hInst */,
   DllInfo * /* pInfo */
);
```

Remarks

This method does nothing when a DLL is removed from the cache.

See Also

Reference

CNoDllCachePeer Class

Other Resources

CNoDllCachePeer Members

Caching Reference

CNoDllCachePeer Structures

For information about the structures in **CNoDllCachePeer**, see CNoDllCachePeer Members.

CNoDllCachePeer::DllInfo Structure

Implementation of DllCachePeerArchetype::DllInfo.

```
struct DllInfo
```

Remarks

This structure has no members.

See Also

Reference

CNoDllCachePeer Class

Other Resources

CNoDllCachePeer Members

Caching Reference

CNoExpireCuller Class

This class conforms to the cache culler archetype and can be used to disable automated removal of cache items based on expiration times.

```
class CNoExpireCuller
```

Remarks

This class never suggests candidates for removal from the cache. None of the methods in this class do anything. CNoExpireCuller::IsExpired always returns FALSE and CNoExpireCuller::GetExpired always returns NULL.

Requirements

Header: atlcache.h

See Also

Other Resources

CExpireCuller Members
Caching Reference
Caching

CNoExpireCuller Members

Methods

Access	Implementation of CacheCullerArchetype::Access.
Add	Implementation of CacheCullerArchetype::Add.
Commit	Implementation of CacheCullerArchetype::Commit.
GetExpired	Implementation of CacheCullerArchetype::GetExpired.
IsExpired	Implementation of CacheCullerArchetype::IsExpired.
Release	Implementation of CacheCullerArchetype::Release.
Remove	Implementation of CacheCullerArchetype::Remove.
Start	Implementation of CacheCullerArchetype::Start.

See Also

Reference

CExpireCuller Class

Other Resources

Caching

Caching Reference

CNoExpireCuller Methods

For information about the methods in **CExpireCuller**, see CExpireCuller Members

CNoExpireCuller::Access

Implementation of CacheCullerArchetype::Access.

```
void Access(
   CCullerCacheData * /* pItem */
);
```

Remarks

This method provides no functionality.

See Also

Reference

CNoExpireCuller Class

Other Resources

CNoExpireCuller Members

CNoExpireCuller::Add

Implementation of CacheCullerArchetype::Add.

```
void Add(
   CCullerCacheData * /* pItem */
);
```

Remarks

This method provides no functionality.

See Also

Reference

CNoExpireCuller Class

Other Resources

CNoExpireCuller Members

CNoExpireCuller::Commit

Implementation of CacheCullerArchetype::Commit.

```
void Commit(
   CCullerCacheData * /* pItem */
);
```

Remarks

This method provides no functionality.

See Also

Reference

CNoExpireCuller Class

Other Resources

CNoExpireCuller Members

CNoExpireCuller::GetExpired

Implementation of CacheCullerArchetype::GetExpired.

```
CCullerCacheData* GetExpired( );
```

Remarks

This method provides no functionality.

See Also

Reference

CNoExpireCuller Class

Other Resources

CNoExpireCuller Members

CNoExpireCuller::IsExpired

Implementation of CacheCullerArchetype::IsExpired.

```
BOOL IsExpired(
   CCullerCacheData * /* pItem */
);
```

Remarks

This method provides no functionality.

See Also

Reference

CNoExpireCuller Class

Other Resources

CNoExpireCuller Members

CNoExpireCuller::Release

Implementation of CacheCullerArchetype::Release.

```
void Release(
   CCullerCacheData * /* pItem */
);
```

Remarks

This method provides no functionality.

See Also

Reference

CNoExpireCuller Class

Other Resources

CNoExpireCuller Members

CNoExpireCuller::Remove

Implementation of CacheCullerArchetype::Remove.

```
void Remove(
   CCullerCacheData * /* pItem */
);
```

Remarks

This method provides no functionality.

See Also

Reference

CNoExpireCuller Class

Other Resources

CNoExpireCuller Members

CNoExpireCuller::Start

Implementation of CacheCullerArchetype::Start.

```
void Start( );
```

Remarks

This method provides no functionality.

See Also

Reference

CNoExpireCuller Class

Other Resources

CNoExpireCuller Members

CNoFileCachePeer Class

This class conforms to the file cache peer archetype and can be used with CFileCache when no auxiliary information needs to be stored with the entries.

```
class CNoFileCachePeer
```

Requirements

Header: atlcache.h

See Also

Other Resources

CNoFileCachePeer Members

Caching Reference

CNoFileCachePeer Members

Methods

Add	Implementation of FileCachePeerArchetype::Add.
Remove	Implementation of FileCachePeerArchetype::Remove.

Structures

PeerInfo	Implementation of FileCachePeerArchetype::PeerInfo.

See Also

Reference

CNoFileCachePeer Class

Other Resources

Caching Reference

CNoFileCachePeer Methods

For information about the methods in **CNoFileCachePeer**, see CNoFileCachePeer Members.

CNoFileCachePeer::Add

Implementation of FileCachePeerArchetype::Add.

```
static BOOL Add(
   PeerInfo* /* pDest */,
   PeerInfo * /* pSrc */
);
```

Remarks

Does nothing.

See Also

Reference

CNoFileCachePeer Class

File Cache Peer Archetype

Other Resources

CNoFileCachePeer Members

Caching Reference

CNoFileCachePeer::Remove

Implementation of FileCachePeerArchetype::Remove.

```
static BOOL Remove(
   const PeerInfo* /* pFileInfo */
);
```

Remarks

Does nothing.

See Also

Reference

CNoFileCachePeer Class

File Cache Peer Archetype

Other Resources

CNoFileCachePeer Members

Caching Reference

CNoFileCachePeer Structures

For information about the structures in **CNoFileCachePeer**, see CNoFileCachePeer Members.

CNoFileCachePeer::PeerInfo Structure

Implementation of FileCachePeerArchetype::PeerInfo.

```
struct PeerInfo
```

Remarks

This structure has no members.

See Also

Reference

CNoFileCachePeer Class

File Cache Peer Archetype

Other Resources

CNoFileCachePeer Members

Caching Reference

CNoFlusher Class

This class conforms to the cache flusher archetype and can be used to disable automated removal of cache items based on access patterns.

```
class CNoFlusher
```

Remarks

This class never suggests candidates for removal from the cache. None of the methods in this class do anything. CNoFlusher::GetStart and CNoFlusher::GetNext always return NULL.

Requirements

Header: atlcache.h

See Also

Other Resources

CNoFlusher Members
Caching
ATL Server Classes
Caching Reference

CNoFlusher Members

Methods

Access	Implementation of CacheFlusherArchetype::Access.
Add	Implementation of CacheFlusherArchetype::Add.
GetNext	Implementation of CacheFlusherArchetype::GetNext.
GetStart	Implementation of CacheFlusherArchetype::GetStart.
Remove	Implementation of CacheFlusherArchetype::Remove.
Release	Implementation of CacheFlusherArchetype::Release.

See Also

Reference

CNoFlusher Class

Other Resources

Caching

Caching Reference

CNoFlusher Methods

For information about the methods in **CNoFlusher**, see CNoFlusher Members.

CNoFlusher::Access

Implementation of CacheFlusherArchetype::Access.

```
void Access(
   CFlusherCacheData * /* pItem */
);
```

Remarks

This method provides no functionality.

See Also

Reference

CNoFlusher Class

Cache Flusher Archetype

Other Resources

CNoFlusher Members

Caching

Caching Reference

CNoFlusher::Add

Implementation of CacheFlusherArchetype::Add.

```
void Add(
   CFlusherCacheData * /* pItem */
);
```

Remarks

This method provides no functionality.

See Also

Reference

CNoFlusher Class

Cache Flusher Archetype

Other Resources

CNoFlusher Members

Caching

Caching Reference

CNoFlusher::GetNext

Implementation of CacheFlusherArchetype::GetNext.

```
CFlusherCacheData * GetNext(
   CFlusherCacheData * /* pCur */
) const;
```

Remarks

This method provides no functionality. It always returns NULL.

See Also

Reference

CNoFlusher Class

Cache Flusher Archetype

Other Resources

CNoFlusher Members

Caching

Caching Reference

CNoFlusher::GetStart

Implementation of CacheFlusherArchetype::GetStart.

```
CFlusherCacheData* GetStart( ) const;
```

Remarks

This method provides no functionality. It always returns NULL.

See Also

Reference

CNoFlusher Class

Cache Flusher Archetype

Other Resources

CNoFlusher Members

Caching

Caching Reference

CNoFlusher::Release

Implementation of CacheFlusherArchetype::Release.

```
void Release(
   CFlusherCacheData * /* pItem */
);
```

Remarks

This method provides no functionality.

See Also

Reference

CNoFlusher Class

Cache Flusher Archetype

Other Resources

CNoFlusher Members

Caching

Caching Reference

CNoFlusher::Remove

Implementation of CacheFlusherArchetype::Remove.

```
void Remove(
   CFlusherCacheData * /* pItem */
);
```

Remarks

This method provides no functionality.

See Also

Reference

CNoFlusher Class

Cache Flusher Archetype

Other Resources

CNoFlusher Members

Caching

Caching Reference

CNonStatelessWorker Class

Receives requests from a thread pool and passes them on to a worker object that is created and destroyed on each request.

```
template <
   class Worker
>
class CNonStatelessWorker
```

Parameters

Worker

A worker thread class conforming to the worker archetype suitable for handling requests queued on CThreadPool.

Remarks

This class is a simple worker thread for use with CThreadPool. This class doesn't provide any request-handling capabilities of its own. Instead, it instantiates one instance of *Worker* per request and delegates the implementation of its methods to that instance.

The benefit of this class is that it provides a convenient way to change the state model for existing worker thread classes. **CThreadPool** will create a single worker for the lifetime of the thread, so if the worker class holds state, it will hold it across multiple requests. By simply wrapping that class in the **CNonStatelessWorker** template before using it with **CThreadPool**, the lifetime of the worker and the state it holds is limited to a single request.

Requirements

Header: atlutil.h

See Also

Reference

CThreadPool Class

Worker Archetype

Other Resources

CNonStatelessWorker Members

ATL Server Classes

CNonStatelessWorker Members

Methods

Execute	Implementation of WorkerArchetype::Execute.
Initialize	Implementation of WorkerArchetype::Initialize.
Terminate	Implementation of WorkerArchetype::Terminate.

Typedefs

RequestType	Implementation of WorkerArchetype::RequestType.

See Also

Reference

CNonStatelessWorker Class

CNonStatelessWorker Methods

For information about the methods in **CNonStatelessWorker**, see CNonStatelessWorker Members.

CNonStatelessWorker::Execute

Implementation of WorkerArchetype::Execute.

```
void Execute(
   Worker::RequestType request,
   void* pvWorkerParam,
   OVERLAPPED* pOverlapped
);
```

Remarks

This method creates an instance of the *Worker* class on the stack and calls Initialize on that object. If the initialization is successful, this method also calls Execute and Terminate on the same object.

See Also

Reference

CNonStatelessWorker Class

Other Resources

CNonStatelessWorker Members

CNonStatelessWorker::Initialize

Implementation of WorkerArchetype::Initialize.

```
BOOL Initialize(
   void* /*pvParam*/
) throw( );
```

Return Value

Always returns TRUE.

Remarks

This class does not do any initialization in **Initialize**.

See Also

Reference

CNonStatelessWorker Class

Other Resources

CNonStatelessWorker Members

CNonStatelessWorker::Terminate

Implementation of WorkerArchetype::Terminate.

```
void Terminate(
   void* /*pvParam*/
) throw( );
```

Remarks

This class does not do any cleanup in **Terminate**.

See Also

Reference

CNonStatelessWorker Class

Other Resources

CNonStatelessWorker Members

CNonStatelessWorker Typedefs

For information about the typedefs in **CNonStatelessWorker**, see CNonStatelessWorker Members.

CNonStatelessWorker::RequestType

Implementation of WorkerArchetype::RequestType.

```
typedef Worker::RequestType RequestType;
```

Remarks

This class handles the same type of work item as the class used for the *Worker* template parameter. See CNonStatelessWorker Overview for details.

See Also

Reference

CNonStatelessWorker Class

Other Resources

CNonStatelessWorker Members

CNoRequestStats Class

Use this class as the argument for the *CRequestStatClass* template parameter to CIsapiExtension if you don't want to collect request-handling statistics.

```
class CNoRequestStats
```

Remarks

This class provides the public interface expected to be provided by the *CRequestStatClass* template parameter to CIsapiExtension, but the methods are implemented to do nothing. The methods in this class that return an HRESULT always return S_OK, and the methods that return a statistic always return 0.

This class conforms to the Request Statistics Archetype. See the documentation for that archetype for other conforming classes.

Requirements

Header: atlisapi.h

See Also

Reference

CStdRequestStats Class
CPerfMonRequestStats Class
Request Statistics Archetype

Other Resources

CNoRequestStats Members
ATL Server Classes

CNoRequestStats Members

Methods

GetActiveThreads	Call this method to get the number of threads currently handling a request.
GetAvgResponseTime	Call this method to get the average response time for handling a request.
GetCurrWaiting	Call this method to get the current number of queued requests.
GetFailedRequests	Call this method to get the total number of failed requests.
GetMaxWaiting	Call this method to get the maximum number of requests that have been queued at any one time since the ISAPI extension was loaded.
GetTotalRequests	Call this method to get the total number of handled requests.
Initialize	This method is called by CIsapiExtension to initialize the object.
OnRequestDequeued	This method is called by CIsapiExtension when a request is removed from the queue and passed to a worker thread.
OnRequestReceived	This method is called by CIsapiExtension when a request is received and added to the queue.
RequestHandled	This method is called by CIsapiExtension when a request has been handled.
Uninitialize	This method is called by CIsapiExtension to uninitialize the object.

See Also

Reference

CNoRequestStats Class

CNoRequestStats Methods

For information about the methods in **CNoRequestStats**, see CNoRequestStats Members

CNoRequestStats::GetActiveThreads

Call this method to get the number of threads currently handling a request.

```
long GetActiveThreads( ) throw( );
```

Return Value

Returns 0.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoRequestStats Class

Other Resources

CNoRequestStats Members

CNoRequestStats::GetAvgResponseTime

Call this method to get the average response time for handling a request.

```
long GetAvgResponseTime( ) throw( );
```

Return Value

Returns 0.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoRequestStats Class

Other Resources

CNoRequestStats Members

CNoRequestStats::GetCurrWaiting

Call this method to get the current number of queued requests.

```
long GetCurrWaiting( ) throw( );
```

Return Value

Returns 0.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoRequestStats Class

Other Resources

CNoRequestStats Members

CNoRequestStats::GetFailedRequests

Call this method to get the total number of failed requests.

```
long GetFailedRequests( ) throw( );
```

Return Value

Returns 0.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoRequestStats Class

Other Resources

CNoRequestStats Members

CNoRequestStats::GetMaxWaiting

Call this method to get the maximum number of requests that have been queued at any one time since the ISAPI extension was loaded.

```
long GetMaxWaiting( ) throw( );
```

Return Value

Returns 0.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoRequestStats Class

Other Resources

CNoRequestStats Members

CNoRequestStats::GetTotalRequests

Call this method to get the total number of handled requests.

```
long GetTotalRequests( ) throw( );
```

Return Value

Returns 0.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoRequestStats Class

Other Resources

CNoRequestStats Members

CNoRequestStats::Initialize

This method is called by CIsapiExtension to initialize the object.

```
HRESULT Initialize( ) throw( );
```

Return Value

Returns S_OK.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoRequestStats Class

Other Resources

CNoRequestStats Members

CNoRequestStats::OnRequestDequeued

This method is called by CIsapiExtension when a request is removed from the queue and passed to a worker thread.

```
void OnRequestDequeued( ) throw( );
```

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoRequestStats Class

Other Resources

CNoRequestStats Members

CNoRequestStats::OnRequestReceived

This method is called by CIsapiExtension when a request is received and added to the queue.

```
void OnRequestReceived( ) throw( );
```

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoRequestStats Class

Other Resources

CNoRequestStats Members

CNoRequestStats::RequestHandled

This method is called by CIsapiExtension when a request has been handled.

```
void RequestHandled(
   AtlServerRequest * /* pRequestInfo */,
   BOOL /* bSuccess */
) throw( );
```

Parameters

pRequestInfo

The information about the current request.

bSuccess

TRUE if the request was successfully handled, FALSE otherwise.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoRequestStats Class

Other Resources

CNoRequestStats Members

AtlServerRequest Members

CNoRequestStats::Uninitialize

This method is called by CIsapiExtension to uninitialize the object.

```
void Uninitialize( ) throw( );
```

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoRequestStats Class

Other Resources

CNoRequestStats Members

CNoStatClass Class

Use this class anywhere that a class conforming to the cache statistics archetype is expected if you don't want to collect cache statistics.

```
class CNoStatClass
```

Remarks

This class provides a public interface that conforms to the cache statistics archetype, but the methods are implemented to do nothing. The methods in this class that return an HRESULT always return S_OK, and the methods that return a statistic always return 0.

Classes that make use of the cache statistics archetype typically expose the statistics gathered by this class through the IMemoryCacheStats interface.

Typically, you don't need to call the methods provided by this class directly.

Requirements

Header: atlcache.h

See Also

Reference

Cache Statistics Archetype

Other Resources

Caching

CNoStatClass Members

Caching Reference

CNoStatClass Members

Methods

AddElement	Call this method when an item is added to the cache.
GetCurrentAllocSize	Call this method to get the current number of bytes used by the cache for storing data.
GetCurrentEntryCount	Call this method to get the current number of items in the cache.
GetHitCount	Call this method to get the number of times the cache successfully returned an item.
GetMaxAllocSize	Call this method to get the maximum number of bytes used by the cache at one time since statistics gathering was started.
GetMaxEntryCount	Call this method to get the maximum number of items in the cache at one time since statistics gathering was started.
GetMissCount	Call this method to get the number of times the cache did not contain a requested item.
Hit	Call this method when an item is successfully retrieved from the cache.
Initialize	Call this method to initialize the object.
Miss	Call this method when a requested item cannot be found in the cache.
ReleaseElement	Call this method when an item is removed from the cache.
ResetCounters	Call this method to reset all the statistics to zero.
Uninitialize	Call this method to uninitialize the object.

See Also

Reference

CNoStatClass Class

Other Resources

Caching

Caching Reference

CNoStatClass Methods

For information about the methods in **CNoStatClass**, see CNoStatClass Members

CNoStatClass::AddElement

Call this method when an item is added to the cache.

```
void AddElement(
   DWORD
);
```

Parameters

dwBytes

The number of bytes used by the newly added item.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoStatClass Class

CNoStatClass::ReleaseElement

Other Resources

Caching

CNoStatClass Members

CNoStatClass::GetCurrentAllocSize

Call this method to get the current number of bytes used by the cache for storing data.

```
DWORD GetCurrentAllocSize( );
```

Return Value

Returns 0.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoStatClass Class

CNoStatClass::GetMaxAllocSize

Other Resources

Caching

CNoStatClass Members

Caching Reference

CNoStatClass::GetCurrentEntryCount

Call this method to get the current number of items in the cache.

```
DWORD GetCurrentEntryCount( );
```

Return Value

Returns 0.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoStatClass Class

CNoStatClass::GetMaxEntryCount

Other Resources

Caching

CNoStatClass Members

Caching Reference

CNoStatClass::GetHitCount

Call this method to get the number of times the cache successfully returned an item.

```
DWORD GetHitCount( );
```

Return Value

Returns 0.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoStatClass Class

CNoStatClass::GetMissCount

Other Resources

Caching

CNoStatClass Members

Caching Reference

CNoStatClass::GetMaxAllocSize

Call this method to get the maximum number of bytes used by the cache at one time since statistics gathering was started.

```
DWORD GetMaxAllocSize( );
```

Return Value

Returns 0.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoStatClass Class

CNoStatClass::GetCurrentAllocSize

Other Resources

Caching

CNoStatClass Members

Caching Reference

CNoStatClass::GetMaxEntryCount

Call this method to get the maximum number of items in the cache at one time since statistics gathering was started.

```
DWORD GetMaxEntryCount( );
```

Return Value

Returns 0.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoStatClass Class

CNoStatClass::GetCurrentEntryCount

Other Resources

Caching

CNoStatClass Members

Caching Reference

CNoStatClass::GetMissCount

Call this method to get the number of times the cache did not contain a requested item.

```
DWORD GetMissCount( );
```

Return Value

Returns 0.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoStatClass Class
CNoStatClass::GetHitCount

Other Resources

Caching
CNoStatClass Members
Caching Reference

CNoStatClass::Hit

Call this method when an item is successfully retrieved from the cache.

```
void Hit( );
```

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoStatClass Class

CNoStatClass::Miss

Other Resources

Caching

CNoStatClass Members

Caching Reference

CNoStatClass::Initialize

Call this method to initialize the object.

```
HRESULT Initialize( );
```

Return Value

Returns S_OK.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoStatClass Class

Other Resources

Caching

CNoStatClass Members

Caching Reference

CNoStatClass::Miss

Call this method when a requested item cannot be found in the cache.

```
void Miss( );
```

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoStatClass Class

CNoStatClass::Hit

Other Resources

Caching

CNoStatClass Members

Caching Reference

CNoStatClass::ReleaseElement

Call this method when an item is removed from the cache.

```
void ReleaseElement(
   DWORD
);
```

Parameters

dwBytes

The number of bytes freed by the item just removed.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoStatClass Class

CNoStatClass::AddElement

Other Resources

Caching

CNoStatClass Members

Caching Reference

CNoStatClass::ResetCounters

Call this method to reset all the statistics to zero.

```
void ResetCounters( );
```

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoStatClass Class

Other Resources

Caching

CNoStatClass Members

Caching Reference

CNoStatClass::Uninitialize

Call this method to uninitialize the object.

```
HRESULT Uninitialize( );
```

Return Value

Returns S_OK.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoStatClass Class

Other Resources

Caching

CNoStatClass Members

Caching Reference

CNoWorkerThread Class

Use this class as the argument for the *MonitorClass* template parameter to CBlobCache, CStencilCache, or other cache classes if you want to disable dynamic cache maintenance.

```
class CNoWorkerThread
```

Remarks

This class provides the same public interface as CWorkerThread. This interface is expected to be provided by the *MonitorClass* template parameter to CBlobCache, CStencilCache and other cache classes.

The methods in this class are implemented to do nothing. The methods that return an HRESULT always return S_OK, and the methods that return a HANDLE or thread ID always return 0.

Requirements

Header: atlutil.h

See Also

Other Resources

CNoWorkerThread Members

CNoWorkerThread Members

Methods

AddHandle	Non-functional equivalent of CWorkerThread::AddHandle.
AddTimer	Non-functional equivalent of CWorkerThread::AddTimer.
GetThreadHandle	Non-functional equivalent of CWorkerThread::GetThreadHandle.
GetThreadId	Non-functional equivalent of CWorkerThread::GetThreadId.
Initialize	Non-functional equivalent of CWorkerThread::Initialize.
RemoveHandle	Non-functional equivalent of CWorkerThread::RemoveHandle.
Shutdown	Non-functional equivalent of CWorkerThread::Shutdown.

See Also

Reference

CNoWorkerThread Class

CNoWorkerThread Methods

For information about the methods in **CNoWorkerThread**, see CNoWorkerThread Members

CNoWorkerThread::AddHandle

Non-functional equivalent of CWorkerThread::AddHandle.

```
HRESULT AddHandle(
   HANDLE /* hObject */,
   IWorkerThreadClient * /* pClient */,
   DWORD_PTR /* dwParam */
) throw( );
```

Return Value

Always returns S_OK.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoWorkerThread Class

Other Resources

CNoWorkerThread Members

CNoWorkerThread::AddTimer

Non-functional equivalent of CWorkerThread::AddTimer.

```
HRESULT AddTimer(
   DWORD /* dwInterval */,
   IWorkerThreadClient * /* pClient */,
   DWORD_PTR /* dwParam */,
   HANDLE * /* phTimer */
) throw( );
```

Return Value

Always returns S_OK.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoWorkerThread Class

Other Resources

CNoWorkerThread Members

CNoWorkerThread::GetThreadHandle

Non-functional equivalent of CWorkerThread::GetThreadHandle.

```
HANDLE GetThreadHandle( ) throw( );
```

Return Value

Always returns NULL.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoWorkerThread Class

Other Resources

CNoWorkerThread Members

CNoWorkerThread::GetThreadId

Non-functional equivalent of CWorkerThread::GetThreadId.

```
DWORD GetThreadId( ) throw( );
```

Return Value

Always returns 0.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoWorkerThread Class

Other Resources

CNoWorkerThread Members

CNoWorkerThread::Initialize

Non-functional equivalent of CWorkerThread::Initialize.

```
HRESULT Initialize( ) throw( );
```

Return Value

Always returns S_OK.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoWorkerThread Class

Other Resources

CNoWorkerThread Members

CNoWorkerThread::RemoveHandle

Non-functional equivalent of CWorkerThread::RemoveHandle.

```
HRESULT RemoveHandle(
   HANDLE /* hObject */
) throw( );
```

Return Value

Always returns S_OK.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoWorkerThread Class

Other Resources

CNoWorkerThread Members

CNoWorkerThread::Shutdown

Non-functional equivalent of CWorkerThread::Shutdown.

```
HRESULT Shutdown(
   DWORD dwWait = ATL_WORKER_THREAD_WAIT
) throw( );
```

Return Value

Always returns S_OK.

Remarks

The implementation provided by this class does nothing.

See Also

Reference

CNoWorkerThread Class

Other Resources

CNoWorkerThread Members

CNTLMAuthObject Class

This class implements support for performing HTTP navigation using the NTLM authentication scheme.

```
class CNTLMAuthObject : public CAtlBaseAuthObject
```

Remarks

This class performs NTLM authentication for protected URLs and is provided to the CAtlHttpClientT Class class through AddAuthObj before the request is launched. In addition to an instance of the **CNTLMAuthObject** class itself, an implementation of the IAuthInfo interface can be provided to AddAuthObj so that the CAtlHttpClientT class can retrieve the required credentials when needed.

Note
If no IAuthInfo implementation is provided, the **CNTLMAuthObject** class will attempt to satisfy security requirements using the user name and password of the current user.

ATL Server also support BASIC authentication through the CBasicAuthObject class.

Requirements

Header: atlhttp.h

Example

See the HttpClient Sample.

See Also

Reference

CAtlBaseAuthObject Class

Concepts

HTTP Client Services

Other Resources

CNTLMAuthObject Members

HTTP Client Reference

CNTLMAuthObject Members

Methods

Authenticate	This method is called to perform NTLM authentication. It is not typically necessary to call this method exp licitly.
CNTLMAuthObject	Constructor overloads.
~CNTLMAuthObject	The destructor.
GetCredentialNames	Call this method to get the user name for the credential held by this object.
Init	This method is called to initialize the NTLM authentication object. It is not normally necessary to call this method explicitly.
SetAuthInfo	This method is called to attach authorization information. It is not normally necessary to call this method explicitly.

See Also

Reference

CNTLMAuthObject Class

Concepts

HTTP Client Services

Other Resources

HTTP Client Reference

CNTLMAuthObject Methods

For information about the methods in **CNTLMAuthObject**, see CNTLMAuthObject Members

CNTLMAuthObject::Authenticate

This method is called to perform NTLM authentication. It is not typically necessary to call this method explicitly.

```
virtual bool Authenticate(
   LPCTSTR szAuthTypes,
   bool bProxy
) throw( );
```

Parameters

szAuthTypes

A string containing the authentication scheme name. For the NTLM scheme, this string will be "NTLM", as specified by the ATL_HTTP_AUTHTYPE_NTLM macro.

bProxy

true if a proxy server is being used for HTTP transactions, otherwise **false**.

Return Value

Returns **true** on success, **false** on failure.

Remarks

See the CAtlBaseAuthObject::Authenticate method for more information.

See Also

Reference

CNTLMAuthObject Class

Concepts

HTTP Client Services

Other Resources

CNTLMAuthObject Members

HTTP Client Reference

CNTLMAuthObject::~CNTLMAuthObject

The destructor.

```
~CNTLMAuthObject( ) throw( );
```

Remarks

Frees resources used for NTLM authentication.

See Also

Reference

CNTLMAuthObject Class

Concepts

HTTP Client Services

Other Resources

CNTLMAuthObject Members

HTTP Client Reference

CNTLMAuthObject::CNTLMAuthObject

Constructor overloads.

```
CNTLMAuthObject( ) throw( );
CNTLMAuthObject(
   IAuthInfo* pAuthInfo
) throw( );
```

Parameters

pAuthInfo

The address of an object that implements the IAuthInfo interface for the purposes of providing authorization credentials.

Remarks

The **CNTLMAuthObject** class, if not provided with an IAuthInfo implementation, will attempt to negotiate the NTLM protection scheme using the current user name and password. Alternately, security credentials can be provided by an IAuthInfo implementation, either as provided to the **CNTLMAuthObject** constructor, or to the CAtlHttpClientT::AddAuthObj method.

See Also

Reference

CNTLMAuthObject Class

Concepts

HTTP Client Services

Other Resources

CNTLMAuthObject Members

HTTP Client Reference

CNTLMAuthObject::GetCredentialNames

Call this method to get the user name for the credential held by this object.

```
bool GetCredentialNames(
   CString& theName
);
```

Parameters

theName

A reference to a string that will contain the user name on successful return of this method.

Return Value

Returns true on success, false on failure.

See Also

Reference

CNTLMAuthObject Class

Concepts

HTTP Client Services

Other Resources

CNTLMAuthObject Members

HTTP Client Reference

CNTLMAuthObject::Init

This method is called to initialize the NTLM authentication object. It is not normally necessary to call this method explicitly.

```
virtual void Init(
   CAtlHttpClient* pSocket,
   IAuthInfo* pAuthInfo = NULL
) throw( );
```

Remarks

See the CAtlBaseAuthObject::Init method for more information.

See Also

Reference

CNTLMAuthObject Class

Concepts

HTTP Client Services

Other Resources

CNTLMAuthObject Members

HTTP Client Reference

CNTLMAuthObject::SetAuthInfo

This method is called to attach authorization information. It is not normally necessary to call this method explicitly.

```
void SetAuthInfo(
   IAuthInfo* pAuthInfo
) throw( );
```

Parameters

pAuthInfo

The address on an object implementing the IAuthInfo interface.

Remarks

This method is called internally by the CNTLMAuthObject::Init method and is not normally called by the application code.

See Also

Reference

CNTLMAuthObject Class

Concepts

HTTP Client Services

Other Resources

CNTLMAuthObject Members

HTTP Client Reference

COldFlusher Class

This class conforms to the cache flusher archetype and allows the automatic removal of cache items based on their access pattern. Items are presented for removal in the order in which they were added to the cache.

```
class COldFlusher
```

Remarks

This class promotes the removal of old items by proposing items as candidates for removal in the order in which they were added to the cache.

Requirements

Header: atlcache.h

See Also

Other Resources

COldFlusher Members
Caching
ATL Server Classes
Caching Reference

COldFlusher Members

Methods

Access	Implementation of CacheFlusherArchetype::Access.
Add	Implementation of CacheFlusherArchetype::Add.
COldFlusher	The constructor.
GetNext	Implementation of CacheFlusherArchetype::GetNext.
GetStart	Implementation of CacheFlusherArchetype::GetStart.
Release	Implementation of CacheFlusherArchetype::Release.
Remove	Implementation of CacheFlusherArchetype::Remove.

Data Members

pHead	The leading node of the linked list.
pTail	The tail node of the linked list.

See Also

Reference

COldFlusher Class

Other Resources

Caching

Caching Reference

COldFlusher Methods

For information about the methods in **COldFlusher**, see COldFlusher Members.

COldFlusher::Access

Implementation of CacheFlusherArchetype::Access.

```
void Access(
   CFlusherCacheData * /* pItem */
);
```

Remarks

This implementation does nothing. **COldFlusher** only uses the order in which items are added to the cache to determine the order in which they should be removed.

See Also

Reference

COldFlusher Class

Cache Flusher Archetype

Other Resources

CNoFlusher Members

Caching

Caching Reference

COldFlusher::Add

Implementation of CacheFlusherArchetype::Add.

```
void Add(
   CFlusherCacheData* pItem
);
```

Remarks

Adds the item to the tail of the linked list. Items that expire first are at the head of the list.

See Also

Reference

COldFlusher Class
Cache Flusher Archetype
COldFlusher::pTail

Other Resources

CNoFlusher Members
Caching
Caching Reference

COldFlusher::COldFlusher

The constructor.

```
COldFlusher( );
```

Remarks

Zero-initializes the data members.

See Also

Reference

COldFlusher Class

Cache Flusher Archetype

Other Resources

CNoFlusher Members

Caching

Caching Reference

COldFlusher::GetNext

Implementation of CacheFlusherArchetype::GetNext.

```
CFlusherCacheData * GetNext(
   CFlusherCacheData * pCur
) const;
```

Remarks

Returns *pCur*->pNext.

See Also

Reference

COldFlusher Class

Cache Flusher Archetype

Other Resources

CNoFlusher Members

Caching

Caching Reference

COldFlusher::GetStart

Implementation of CacheFlusherArchetype::GetStart.

```
CFlusherCacheData* GetStart( ) const;
```

Remarks

Returns pHead.

See Also

Reference

COldFlusher Class

Cache Flusher Archetype

Other Resources

CNoFlusher Members

Caching

Caching Reference

COldFlusher::Release

Implementation of CacheFlusherArchetype::Release.

```
void Release(
   CFlusherCacheData * /* pItem */
);
```

Remarks

This implementation does nothing. **COldFlusher** only uses the order in which items are added to the cache to determine the order in which they should be removed.

See Also

Reference

COldFlusher Class

Cache Flusher Archetype

Other Resources

CNoFlusher Members

Caching

Caching Reference

COldFlusher::Remove

Implementation of CacheFlusherArchetype::Remove.

```
void Remove(
   CFlusherCacheData* pItem
);
```

Remarks

Removes the item from the linked list.

See Also

Reference

COldFlusher Class

Cache Flusher Archetype

Other Resources

CNoFlusher Members

Caching

Caching Reference

COldFlusher Data Members

For information about the data members in **COldFlusher**, see COldFlusher Members.

COldFlusher::pHead

The leading node of the linked list.

```
CFlusherCacheData * pHead;
```

Remarks

This node describes the oldest cache entry and therefore the first candidate for removal.

See Also

Reference

COldFlusher Class

Cache Flusher Archetype

Other Resources

CNoFlusher Members

Caching

Caching Reference

COldFlusher::pTail

The tail node of the linked list.

```
CFlusherCacheData * pTail;
```

Remarks

This node describes the newest cache entry and therefore the least likely item to be removed.

See Also

Reference

COldFlusher Class

Cache Flusher Archetype

Other Resources

CNoFlusher Members

Caching

Caching Reference

COrCullers Class

This class combines the functionality of two cache cullers into a single class.

```
template <
   class CFirst,
   class CSecond
>
class COrCullers
```

Parameters

CFirst

A class conforming to the cache culler archetype.

CSecond

A class conforming to the cache culler archetype.

Remarks

If either of the classes, *CFirst* or *CSecond*, considers an item to be a candidate for removal from the cache, this class will consider it to be a candidate for removal.

The methods that notify the cache culler about changes to the cache are implemented to call the corresponding method on both *CFirst* and *CSecond* in order. COrCullers::IsExpired returns TRUE if either *CFirst***::IsExpired** or *CSecond***::IsExpired** return TRUE for the same item. COrCullers::GetExpired returns the pointer returned by *CFirst***::GetExpired** unless it's NULL, in which case it returns the pointer returned by *CSecond***::GetExpired**.

Requirements

Header: atlcache.h

See Also

Other Resources

COrCullers Members
Caching
Caching Reference

COrCullers Members

Methods

Access	Implementation of CacheCullerArchetype::Access.
Add	Implementation of CacheCullerArchetype::Add.
Commit	Implementation of CacheCullerArchetype::Commit.
GetExpired	Implementation of CacheCullerArchetype::GetExpired.
IsExpired	Implementation of CacheCullerArchetype::IsExpired.
Release	Implementation of CacheCullerArchetype::Release.
Remove	Implementation of CacheCullerArchetype::Remove.
Start	Implementation of CacheCullerArchetype::Start.

See Also

Reference

COrCullers Class

Other Resources

Caching

Caching Reference

COrCullers Methods

For information about the methods in **COrCullers**, see COrCullers Members

COrCullers::Access

Implementation of CacheCullerArchetype::Access.

```
void Access(
   CCullerCacheData* pItem
);
```

Remarks

Calls **Access** on *CFirst* and *CSecond*.

See Also

Reference

COrCullers Class

Cache Culler Archetype

Other Resources

COrCullers Members

Caching

Caching Reference

COrCullers::Add

Implementation of CacheCullerArchetype::Add.

```
void Add(
   CCullerCacheData* pItem
);
```

Remarks

Calls **Add** on *CFirst* and *CSecond*.

See Also

Reference

COrCullers Class

Cache Culler Archetype

Other Resources

COrCullers Members

Caching

Caching Reference

COrCullers::Commit

Implementation of CacheCullerArchetype::Commit.

```
void Commit(
   CCullerCacheData* pItem
);
```

Remarks

Calls **Commit** on *CFirst* and *CSecond*.

See Also

Reference

COrCullers Class

Cache Culler Archetype

Other Resources

COrCullers Members

Caching

Caching Reference

COrCullers::GetExpired

Implementation of CacheCullerArchetype::GetExpired.

```
CCullerCacheData* GetExpired( );
```

Remarks

Returns the pointer returned by *CFirst*::**GetExpired** unless it's NULL, in which case it returns the pointer returned by *CSecond*::**GetExpired**.

See Also

Reference

COrCullers Class

Cache Culler Archetype

Other Resources

COrCullers Members

Caching

Caching Reference

COrCullers::IsExpired

Implementation of CacheCullerArchetype::IsExpired.

```
BOOL IsExpired(
   CCullerCacheData* pItem
);
```

Remarks

Returns TRUE if either *CFirst*::**IsExpired** or *CSecond*::**IsExpired** return TRUE for the same item.

See Also

Reference

COrCullers Class

Cache Culler Archetype

Other Resources

COrCullers Members

Caching

Caching Reference

COrCullers::Release

Implementation of CacheCullerArchetype::Release.

```
void Release(
   CCullerCacheData* pItem
);
```

Remarks

Calls **Release** on *CFirst* and *CSecond*.

See Also

Reference

COrCullers Class

Cache Culler Archetype

Other Resources

COrCullers Members

Caching

Caching Reference

COrCullers::Remove

Implementation of CacheCullerArchetype::Remove.

```
void Remove(
   CCullerCacheData* pItem
);
```

Remarks

Calls **Remove** on *CFirst* and *CSecond*.

See Also

Reference

COrCullers Class

Cache Culler Archetype

Other Resources

COrCullers Members

Caching

Caching Reference

COrCullers::Start

Implementation of CacheCullerArchetype::Start.

```
void Start( );
```

Remarks

Calls **Start** on *CFirst* and *CSecond*.

See Also

Reference

COrCullers Class

Cache Culler Archetype

Other Resources

COrCullers Members

Caching

Caching Reference

COrFlushers Class

This class combines the functionality of two cache flushers into a single class.

```
template <
   class CFirst,
   class CSecond
>
class COrFlushers
```

Parameters

CFirst

A class conforming to the cache flusher archetype.

CSecond

A class conforming to the cache flusher archetype.

Remarks

The methods that notify the cache flusher about changes to the cache are implemented to call the corresponding method on both *CFirst* and *CSecond* in order. However, the methods that return candidates for removal, GetNext and GetStart, are implemented to use only one of the classes at a time. The class that will be used defaults to *CFirst* but can be changed using COrFlushers::Switch.

Requirements

Header: atlcache.h

See Also

Other Resources

COrFlushers Members
Caching
Caching Reference

COrFlushers Members

Methods

Access	Implementation of CacheFlusherArchetype::Access.
Add	Implementation of CacheFlusherArchetype::Add.
COrFlushers	The constructor.
GetNext	Implementation of CacheFlusherArchetype::GetNext.
GetStart	Implementation of CacheFlusherArchetype::GetStart.
Release	Implementation of CacheFlusherArchetype::Release.
Remove	Implementation of CacheFlusherArchetype::Remove.
Switch	Call this method to swap the order in which the two flushers, *CFirst* and *CSecond*, are used.

See Also

Reference

COrFlushers Class

COrFlushers Class

Other Resources

Caching

Caching Reference

COrFlushers Methods

For information about the methods in **COrFlushers**, see COrFlushers Members

COrFlushers::Access

Implementation of CacheFlusherArchetype::Access.

```
void Access(
   CFlusherCacheData* pItem
);
```

Remarks

Calls **Access** on *CFirst* and *CSecond*.

See Also

Reference

COrFlushers Class

Other Resources

COrFlushers Members

Caching

Caching Reference

COrFlushers::Add

Implementation of CacheFlusherArchetype::Add.

```
void Add(
   CFlusherCacheData* pItem
);
```

Remarks

Calls **Add** on *CFirst* and *CSecond*.

See Also

Reference

COrFlushers Class

Other Resources

Caching

COrFlushers Members

Caching Reference

COrFlushers::COrFlushers

The constructor.

```
COrFlushers( );
```

Remarks

Sets the current flusher to be *CFirst*.

See Also

Reference

COrFlushers Class

Other Resources

COrFlushers Members

Caching

Caching Reference

COrFlushers::GetNext

Implementation of CacheFlusherArchetype::GetNext.

```
CFlusherCacheData * GetNext(
   CFlusherCacheData * pCur
) const;
```

Remarks

Calls the **GetNext** method of one of the two internal flushing objects. Use the Switch method to alternate between the internal objects. See COrFlushers Overview.

See Also

Reference

COrFlushers Class

Other Resources

COrFlushers Members

Caching

Caching Reference

COrFlushers::GetStart

Implementation of CacheFlusherArchetype::GetStart.

```
CFlusherCacheData* GetStart( ) const;
```

Remarks

Calls the **GetStart** method of one of the two internal flushing objects. Use the Switch method to alternate between the internal objects. See COrFlushers Overview.

See Also

Reference

COrFlushers Class

Other Resources

COrFlushers Members

Caching

Caching Reference

COrFlushers::Release

Implementation of CacheFlusherArchetype::Release.

```
void Release(
   CFlusherCacheData* pItem
);
```

Remarks

Calls **Release** on *CFirst* and *CSecond*.

See Also

Reference

COrFlushers Class

Other Resources

COrFlushers Members

Caching

Caching Reference

COrFlushers::Remove

Implementation of CacheFlusherArchetype::Remove.

```
void Remove(
   CFlusherCacheData* pItem
);
```

Remarks

Calls **Remove** on *CFirst* and *CSecond*.

See Also

Reference

COrFlushers Class

Other Resources

COrFlushers Members

Caching

Caching Reference

COrFlushers::Switch

Call this method to swap the order in which the two flushers, *CFirst* and *CSecond*, are used.

```
BOOL Switch( );
```

Return Value

Returns TRUE if the original order of the two internal flushing objects has been reversed or FALSE if the original order has been restored.

Remarks

The Add, Access, Remove, and Release methods affect both internal flushing objects. The object affected by the GetStart and GetNext methods, however, is determined by the value returned from **Switch**. See the COrFlushers Overview.

See Also

Reference

COrFlushers Class

Other Resources

COrFlushers Members

Caching

Caching Reference

CPageCachePeer Class

This class is used internally by ATL Server to manage auxiliary information for cached responses returned by request handlers.

```
class CPageCachePeer
```

Remarks

CPageCachePeer conforms to the file cache peer archetype. It is not normally necessary to use this class directly.

Requirements

Header: atlisapi.h

See Also

Reference

CNoFileCachePeer Class

Other Resources

ATL Server Classes

Caching Reference

Caching

CPathT Class

This class represents a path.

```
template< typename StringType >
class CPathT
```

Parameters

StringType

The ATL/MFC string class to use for the path (see CStringT).

Remarks

CPath, **CPathA**, and **CPathW** are instantiations of **CPathT** defined as follows:

```
typedef CPathT< CString > CPath;
typedef CPathT< CStringA > CPathA;
typedef CPathT< CStringW > CPathW;
```

Requirements

Header: atlpath.h

Example

See the ISAPIFilter Sample.

See Also

Reference

CStringT Class

Other Resources

CPathT Members

ATL Server Classes

CPathT Members

Methods

AddBackslash	Call this method to add a backslash to the end of a string to create the correct syntax for a path.
AddExtension	Call this method to add a file extension to a path.
Append	Call this method to append a string to the current path.
BuildRoot	Call this method to create a root path from a given drive number.
Canonicalize	Call this method to convert the path to canonical form.
Combine	Call this method to concatenate a string representing a directory name and a string representing a file path name into one path.
CommonPrefix	Call this method to determine whether the specified path shares a common prefix with the current path.
CompactPath	Call this method to truncate a file path to fit within a given pixel width by replacing path components with e llipses.
CompactPathEx	Call this method to truncate a file path to fit within a given number of characters by replacing path compon ents with ellipses.
CPathT	The constructor for the path.
FileExists	Call this method to check whether the file at this path name exists.
FindExtension	Call this method to find the position of the file extension within the path.
FindFileName	Call this method to find the position of the file name within the path.
GetDriveNumber	Call this method to search the path for a drive letter within the range of 'A' to 'Z' and return the correspond ing drive number.
GetExtension	Call this method to get the file extension from the path.
IsDirectory	Call this method to check whether the path is a valid directory.
IsFileSpec	Call this method to search a path for any path-delimiting characters (for example, ':' or '\'). If there are no p ath-delimiting characters present, the path is considered to be a File Spec path.
IsPrefix	Call this method to determine whether a path contains a valid prefix of the type passed by *pszPrefix*.
IsRelative	Call this method to determine if the path is relative.
IsRoot	Call this method to determine if the path is a directory root.
IsSameRoot	Call this method to determine whether another path has a common root component with the current path.
IsUNC	Call this method to determine whether the path is a valid UNC (universal naming convention) path for a ser ver and share.

IsUNCServer	Call this method to determine whether the path is a valid UNC (universal naming convention) path for a server only.
IsUNCServerShare	Call this method to determine whether the path is a valid UNC (universal naming convention) share path, *server**share*.
MakePretty	Call this method to convert a path to all lowercase characters to give the path a consistent appearance.
MatchSpec	Call this method to search the path for a string containing a wildcard match type.
QuoteSpaces	Call this method to enclose the path in quotation marks if it contains any spaces.
RelativePathTo	Call this method to create a relative path from one file or folder to another.
RemoveArgs	Call this method to remove any command-line arguments from the path.
RemoveBackslash	Call this method to remove the trailing backslash from the path.
RemoveBlanks	Call this method to remove all leading and trailing spaces from the path.
RemoveExtension	Call this method to remove the file extension from the path, if there is one.
RemoveFileSpec	Call this method to remove the trailing file name and backslash from the path, if it has them.
RenameExtension	Call this method to replace the file name extension in the path with a new extension. If the file name does not contain an extension, the extension will be attached to the end of the string.
SkipRoot	Call this method to parse a path, ignoring the drive letter or UNC server/share path parts.
StripPath	Call this method to remove the path portion of a fully qualified path and file name.
StripToRoot	Call this method to remove all parts of the path except for the root information.
UnquoteSpaces	Call this method to remove quotation marks from the beginning and end of a path.

Operators

operator +=	This operator appends a string to the path.
operator const StringType &	This operator allows the object to be treated like a string.
operator CPathT::PCXSTR	This operator allows the object to be treated like a string.
operator StringType &	This operator allows the object to be treated like a string.

Data Members

m_strPath	The path.

Typedefs

XCHAR	A character type.
PCXSTR	A constant string type.
PXSTR	A string type.

See Also

Reference

CPathT Class

CPathT Methods

For information about the methods in **CPathT**, see CPathT Members.

CPathT::AddBackslash

Call this method to add a backslash to the end of a string to create the correct syntax for a path. If the path already has a trailing backslash, no backslash will be added.

```
void AddBackslash( );
```

Remarks

For more information, see PathAddBackSlash.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::AddExtension

Call this method to add a file extension to a path.

```
BOOL AddExtension(
   PCXSTR pszExtension
);
```

Parameters

pszExtension

The file extension to add.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

For more information, see PathAddExtension.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::Append

Call this method to append a string to the current path.

```
BOOL Append(
   PCXSTR pszMore
);
```

Parameters

pszMore

The string to append.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

For more information, see PathAppend.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::BuildRoot

Call this method to create a root path from a given drive number.

```
void BuildRoot(
   int iDrive
);
```

Parameters

iDrive

The drive number (0 is A:, 1 is B:, and so on).

Remarks

For more information, see PathBuildRoot.

See Also

Reference

CPathT Class

CPathT::GetDriveNumber

Other Resources

CPathT Members

CPathT::Canonicalize

Call this method to convert the path to canonical form.

```
void Canonicalize( );
```

Remarks

For more information, see PathCanonicalize.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::Combine

Call this method to concatenate a string representing a directory name and a string representing a file path name into one path.

```
void Combine(
   PCXSTR pszDir,
   PCXSTR pszFile
);
```

Parameters

pszDir

The directory path.

pszFile

The file path.

Remarks

For more information, see PathCombine.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::CommonPrefix

Call this method to determine whether the specified path shares a common prefix with the current path.

```
CPathT< StringType > CommonPrefix(
   PCXSTR pszOther
);
```

Parameters

pszOther

The path to compare to the current one.

Return Value

Returns the common prefix.

Remarks

A prefix is one of these types: "C:\\", ".", "..", "..\\". For more information, see PathCommonPrefix.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::CompactPath

Call this method to truncate a file path to fit within a given pixel width by replacing path components with ellipses.

```
BOOL CompactPath(
   HDC hDC,
   UINT nWidth
);
```

Parameters

hDC

The device context used for font metrics.

nWidth

The width, in pixels, that the string will be forced to fit in.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

For more information, see PathCompactPath.

See Also

Reference

CPathT Class

CPathT::CompactPathEx

Other Resources

CPathT Members

CPathT::CompactPathEx

Call this method to truncate a file path to fit within a given number of characters by replacing path components with ellipses.

```
BOOL CompactPathEx(
   UINT nMaxChars,
   DWORD dwFlags = 0
);
```

Parameters

nMaxChars

The maximum number of characters to be contained in the new string, including the terminating NULL character.

dwFlags

Reserved.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

For more information, see PathCompactPathEx.

See Also

Reference

CPathT Class

CPathT::CompactPath

Other Resources

CPathT Members

CPathT::CPathT

The constructor.

```
CPathT(
   PCXSTR pszPath
);
CPathT(
   const CPathT< StringType >& path
);
CPathT( ) throw( );
```

Parameters

pszPath

The pointer to a path string.

path

The path string.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::FileExists

Call this method to check whether the file at this path name exists.

```
BOOL FileExists( ) const;
```

Return Value

Returns TRUE if the file exists, FALSE otherwise.

Remarks

For more information, see PathFileExists.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::FindExtension

Call this method to find the position of the file extension within the path.

```
int FindExtension( ) const;
```

Return Value

Returns the position of the "." preceding the extension. If no extension is found, returns –1.

Remarks

For more information, see PathFindExtension.

See Also

Reference

CPathT Class

CPathT::GetExtension

Other Resources

CPathT Members

CPathT::FindFileName

Call this method to find the position of the file name within the path.

```
int FindFileName( ) const;
```

Return Value

Returns the position of the file name. If no file name is found, returns –1.

Remarks

For more information, see PathFindFileName.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::GetDriveNumber

Call this method to search the path for a drive letter within the range of 'A' to 'Z' and return the corresponding drive number.

```
int GetDriveNumber( ) const;
```

Return Value

Returns the drive number as an integer from 0 through 25 (corresponding to 'A' through 'Z') if the path has a drive letter, or -1 otherwise.

Remarks

For more information, see PathGetDriveNumber.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::GetExtension

Call this method to get the file extension from the path.

```
StringType GetExtension( ) const;
```

Return Value

Returns the file extension.

See Also

Reference

CPathT Class

CPathT::FindExtension

Other Resources

CPathT Members

CPathT::IsDirectory

Call this method to check whether the path is a valid directory.

```
BOOL IsDirectory( ) const;
```

Return Value

Returns TRUE if the path is a directory, FALSE otherwise.

Remarks

For more information, see PathIsDirectory.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::IsFileSpec

Call this method to search a path for any path-delimiting characters (for example, ':' or '\'). If there are no path-delimiting characters present, the path is considered to be a File Spec path.

```
BOOL IsFileSpec( ) const;
```

Return Value

Returns TRUE if there are no path-delimiting characters within the path, or FALSE if there are path-delimiting characters.

Remarks

For more information, see PathIsFileSpec.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::IsPrefix

Call this method to determine whether a path contains a valid prefix of the type passed by *pszPrefix*.

```
BOOL IsPrefix(
   PCXSTR pszPrefix
) const;
```

Parameters

pszPrefix

The prefix for which to search. A prefix is one of these types: "C:\\", ".", "..", "..\\".

Return Value

Returns TRUE if the path contains the prefix, or FALSE otherwise.

Remarks

For more information, see PathIsPrefix.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::IsRelative

Call this method to determine if the path is relative.

```
BOOL IsRelative( ) const;
```

Return Value

Returns TRUE if the path is relative, or FALSE if it is absolute.

Remarks

For more information, see PathIsRelative.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::IsRoot

Call this method to determine if the path is a directory root.

```
BOOL IsRoot( ) const;
```

Return Value

Returns TRUE if the path is a root, or FALSE otherwise.

Remarks

For more information, see PathIsRoot.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::IsSameRoot

Call this method to determine whether another path has a common root component with the current path.

```
BOOL IsSameRoot(
   PCXSTR pszOther
) const;
```

Parameters

pszOther

The other path.

Return Value

Returns TRUE if both strings have the same root component, or FALSE otherwise.

Remarks

For more information, see PathIsSameRoot.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::IsUNC

Call this method to determine whether the path is a valid UNC (universal naming convention) path for a server and share.

```
BOOL IsUNC( ) const;
```

Return Value

Returns TRUE if the path is a valid UNC path, or FALSE otherwise.

Remarks

For more information, see PathIsUNC.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::IsUNCServer

Call this method to determine whether the path is a valid UNC (universal naming convention) path for a server only.

```
BOOL IsUNCServer( ) const;
```

Return Value

Returns TRUE if the string is a valid UNC path for a server only (no share name), or FALSE otherwise.

Remarks

For more information, see PathIsUNCServer.

See Also

Reference

CPathT Class

CPathT::IsUNCServerShare

Other Resources

CPathT Members

CPathT::IsUNCServerShare

Call this method to determine whether the path is a valid UNC (universal naming convention) share path, *server**share*.

```
BOOL IsUNCServerShare( ) const;
```

Return Value

Returns TRUE if the path is in the form *server**share*, or FALSE otherwise.

Remarks

For more information, see PathIsUNCServerShare.

See Also

Reference

CPathT Class

CPathT::IsUNCServer

Other Resources

CPathT Members

CPathT::MakePretty

Call this method to convert a path to all lowercase characters to give the path a consistent appearance.

```
BOOL MakePretty( );
```

Return Value

Returns TRUE if the path has been converted, or FALSE otherwise.

Remarks

For more information, see PathMakePretty.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::MatchSpec

Call this method to search the path for a string containing a wildcard match type.

```
BOOL MatchSpec(
   PCXSTR pszSpec
) const;
```

Parameters

pszSpec

Pointer to a null-terminated string with the file type for which to search. For example, to test whether the file at the current path is a DOC file, *pszSpec* should be set to "*.doc".

Return Value

Returns TRUE if the string matches, or FALSE otherwise.

Remarks

For more information, see PathMatchSpec.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::QuoteSpaces

Call this method to enclose the path in quotation marks if it contains any spaces.

```
void QuoteSpaces( );
```

Remarks

For more information, see PathQuoteSpaces.

See Also

Reference

CPathT Class

CPathT::UnquoteSpaces

Other Resources

CPathT Members

CPathT::RelativePathTo

Call this method to create a relative path from one file or folder to another.

```
BOOL RelativePathTo(
   PCXSTR pszFrom,
   DWORD dwAttrFrom,
   PCXSTR pszTo,
   DWORD dwAttrTo
);
```

Parameters

pszFrom

The start of the relative path.

dwAttrFrom

The File attributes of *pszFrom*. If this value contains FILE_ATTRIBUTE_DIRECTORY, *pszFrom* is assumed to be a directory; otherwise, *pszFrom* is assumed to be a file.

pszTo

The end point of the relative path.

dwAttrTo

The File attributes of *pszTo*. If this value contains FILE_ATTRIBUTE_DIRECTORY, *pszTo* is assumed to be a directory; otherwise, *pszTo* is assumed to be a file.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

For more information, see PathRelativePathTo.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::RemoveArgs

Call this method to remove any command-line arguments from the path.

```
void RemoveArgs( );
```

Remarks

For more information, see PathRemoveArgs.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::RemoveBackslash

Call this method to remove the trailing backslash from the path.

```
void RemoveBackslash( );
```

Remarks

For more information, see PathRemoveBackslash.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::RemoveBlanks

Call this method to remove all leading and trailing spaces from the path.

```
void RemoveBlanks( );
```

Remarks

For more information, see PathRemoveBlanks.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::RemoveExtension

Call this method to remove the file extension from the path, if there is one.

```
void RemoveExtension( );
```

Remarks

For more information, see PathRemoveExtension.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::RemoveFileSpec

Call this method to remove the trailing file name and backslash from the path, if it has them.

```
BOOL RemoveFileSpec( );
```

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

For more information, see PathRemoveFileSpec.

Example

See the ISAPIFilter Sample.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::RenameExtension

Call this method to replace the file name extension in the path with a new extension. If the file name does not contain an extension, the extension will be attached to the end of the path.

```
BOOL RenameExtension(
   PCXSTR pszExtension
);
```

Parameters

pszExtension

The new file name extension, preceded by a "." character.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

For more information, see PathRenameExtension.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::SkipRoot

Call this method to parse a path, ignoring the drive letter or UNC (universal naming convention) server/share path parts.

```
int SkipRoot( ) const;
```

Return Value

Returns the position of the beginning of the subpath that follows the root (drive letter or UNC server/share).

Remarks

For more information, see PathSkipRoot.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::StripPath

Call this method to remove the path portion of a fully qualified path and file name.

```
void StripPath( );
```

Remarks

For more information, see PathStripPath.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::StripToRoot

Call this method to remove all parts of the path except for the root information.

```
BOOL StripToRoot( );
```

Return Value

Returns TRUE if a valid drive letter was found in the path, or FALSE otherwise.

Remarks

For more information, see PathStripToRoot.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::UnquoteSpaces

Call this method to remove quotation marks from the beginning and end of a path.

```
void UnquoteSpaces( );
```

Remarks

For more information, see PathUnquoteSpaces.

See Also

Reference

CPathT Class

CPathT::QuoteSpaces

Other Resources

CPathT Members

CPathT Operators

For information about the operators in **CPathT**, see CPathT Members.

CPathT::operator +=

This operator appends a string to the path.

```
CPathT< StringType >& operator +=(
   PCXSTR pszMore
);
```

Parameters

pszMore

The string to append.

Return Value

Returns the updated path.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::operator const StringType &

This operator allows the object to be treated like a string.

```
operator const StringType&( ) const throw( );
```

Return Value

Returns a string representing the current path managed by this object.

See Also

Reference

CPathT Class
CPathT::operator StringType &
CPathT::operator CPathT::PCXSTR

Other Resources

CPathT Members

CPathT::operator CPathT::PCXSTR

This operator allows the object to be treated like a string.

```
operator PCXSTR( ) const throw( );
```

Return Value

Returns a string representing the current path managed by this object.

See Also

Reference

CPathT Class
CPathT::operator const StringType &
CPathT::operator StringType &

Other Resources

CPathT Members

CPathT::operator StringType &

This operator allows the object to be treated like a string.

```
operator StringType&( ) throw( );
```

Return Value

Returns a string representing the current path managed by this object.

See Also

Reference

CPathT Class
CPathT::operator const StringType &
CPathT::operator CPathT::PCXSTR

Other Resources

CPathT Members

CPathT Data Members

For information about the data members in **CPathT**, see CPathT Members.

CPathT::m_strPath

The path.

```
StringType m_strPath;
```

Remarks

StringType is the template parameter to **CPathT**.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT Typedefs

For information about the typedefs in **CPathT**, see CPathT Members.

CPathT::XCHAR

A character type.

```
typedef StringType::XCHAR XCHAR;
```

Remarks

StringType is the template parameter to **CPathT**.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::PCXSTR

A constant string type.

```
typedef StringType::PCXSTR PCXSTR;
```

Remarks

StringType is the template parameter to **CPathT**.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPathT::PXSTR

A string type.

```
typedef StringType::PXSTR PXSTR;
```

Remarks

StringType is the template parameter to **CPathT**.

See Also

Reference

CPathT Class

Other Resources

CPathT Members

CPerfLock Class

This class manages the lock state of performance monitor objects.

```
class CPerfLock
```

Remarks

The **CPerfLock** class is a helper class intended for use in managing the lock state of **CPerfMon** objects. Because the **CPerfLock** class unlocks locked **CPerfMon** objects in its destructor, using **CPerfLock** to lock and unlock **CPerfMon** objects is safer than using explicit calls to the **CPerfMon** object.

Example

See the PerformanceCounter Sample and the PerformanceScribble Sample.

Requirements

Header: atlperf.h

See Also

Concepts

Performance Monitoring

Other Resources

CPerfLock Members

ATL Server Classes

Performance Monitoring Reference

CPerfLock Members

Methods

CPerfLock	The constructor for a performance monitor object.
~CPerfLock	The destructor for a performance monitor object.
GetStatus	Call this method to get the status of the lock attempt.

See Also

Reference

CPerfLock Class

Concepts

Performance Monitoring

Other Resources

Performance Monitoring Reference

CPerfLock Methods

For information about the methods in **CPerfLock**, see CPerfLock Members.

CPerfLock::CPerfLock

The constructor for a performance monitor object.

```
CPerfLock(
   CPerfMon* pPerfMon,
   DWORD dwTimeout = INFINITE
) throw( );
```

Parameters

pPerfMon

The performance monitor object to be locked.

dwTimeout

The maximum time to wait for a lock before returning. By default, the **CPerfLock** constructor will wait indefinitely for a lock.

Remarks

Because constructors do not have return values, the GetStatus function must be used to determine success or failure of the lock attempt.

See Also

Reference

CPerfLock Class

Concepts

Performance Monitoring

Other Resources

CPerfLock Members

Performance Monitoring Reference

CPerfLock::~CPerfLock

The destructor for a performance monitor object.

```
~CPerfLock( ) throw( );
```

Remarks

The **CPerfLock** destructor releases locks set by the **CPerfLock** constructor.

See Also

Reference

CPerfLock Class

Concepts

Performance Monitoring

Other Resources

CPerfLock Members

Performance Monitoring Reference

CPerfLock::GetStatus

Call this method to get the lock attempt status of a performance monitor object.

```
HRESULT GetStatus( ) const throw( );
```

Return Value

Returns **S_OK** on success, **WAIT_TIMEOUT** if the timeout supplied to the **CPerfLock** constructor elapses before the **CPerfMon** object is locked, or an error **HRESULT** on failure.

Remarks

The **GetStatus** function should be used immediately after the CPerfLock constructor to check for failure.

Example

See the PerformanceCounter Sample and the PerformanceScribble Sample.

See Also

Reference

CPerfLock Class

Concepts

Performance Monitoring

Other Resources

CPerfLock Members

Performance Monitoring Reference

CPerfMon Class

This class represents a performance monitor object manager.

```
class CPerfMon
```

Remarks

The **CPerfMon** class serves as a base class for application-specific performance monitor objects responsible for managing performance objects. Explicit use of the **CPerfMon** class is not necessary for attributed code (see the perfmon attribute).

Requirements

Header: atlperf.h

See Also

Tasks

PerformanceCounter Sample: Performance Monitoring in an ATL Server Application
PerformanceScribble Sample: Performance Monitoring in an MFC Application
PerformancePersist Sample: Provides Custom Performance Monitoring Data

Concepts

Performance Monitoring

Other Resources

CPerfMon Members
ATL Server Classes
Performance Monitoring Reference

CPerfMon Members

Methods

AddCategoryDefinition	Add a new performance monitor category.
AddCounterDefinition	Add a new performance monitor counter.
CreateInstance	Call this method to create an instance of a performance object.
CreateInstanceByName	Call this method to create an instance of a performance object, given a name.
_FindCategoryInfo	Retrieves the category info given a category ID.
Initialize	Call this method to initialize the performance monitor object.
LoadFromXML	Call this method to load previously saved performance data from a stream.
LockPerf	Call this method to lock a performance monitor object for synchronized access.
PersistToXML	Call this method to save a performance monitor object to a stream.
Register	Call this method to register this performance monitor object with the Windows performance subsystem.
RegisterAllStrings	Call this method to register all strings in a performance monitor object.
RegisterStrings	Call this method to register the strings in a performance monitor object.
ReleaseInstance	Call this method to release an instance of a performance monitor object.
UnInitialize	Call this method to uninitialize a performance monitor object.
UnlockPerf	Call this method to unlock a performance monitor object.
Unregister	Call this method to unregister this performance monitor object from the Windows performance subsystem.

See Also

Reference

CPerfMon Class

Concepts

Performance Monitoring

Other Resources

Performance Monitoring Reference

CPerfMon Methods

For information about the methods in **CPerfMon**, see CPerfMon Members.

CPerfMon::_FindCategoryInfo

This function retrieves the category info given a category ID.

```
CategoryInfo* _FindCategoryInfo(
   DWORD dwCategoryId
) throw();
```

Parameters

dwCategoryId

Numeric ID that uniquely represents each category.

Return Value

Returns a pointer to **CategoryInfo** structure on success, or NULL on failure.

See Also

Reference

CPerfMon Class

Concepts

Performance Monitoring

Other Resources

CPerfMon Members

Performance Monitoring Reference

CPerfMon::AddCategoryDefinition

Adds a new performance monitor category.

```
HRESULT AddCategoryDefinition(
   DWORD dwCategoryId,
   LPCTSTR szCategoryName,
   LPCTSTR szHelpString,
   DWORD dwDetailLevel,
   INT nDefaultCounter,
   BOOL bInstanceLess,
   UINT nStructSize,
   UINT nMaxInstanceNameLen
) throw( );
```

Parameters

dwCategoryId ,

A numeric ID that uniquely represents each performance object contained within the **CPerfMon** derived class. This value is used to identify the object type when calling **CPerfMon::CreateInstance** or **CPerfMon::CreateInstanceByName**.

szCategoryName

The name that is displayed for this object within the Performance Monitoring Console. Can be provided as a string or a resource ID.

szHelpString

A help string that is displayed for this object within the Performance Monitoring Console. Can be provided as a string or a string resource ID.

dwDetailLevel

Object detail levelCPerfObject Structure.

nDefaultCounter

A zero-based index of the counter that should be selected by default when viewing objects of this type, Or -1 if there is no default counter.

bInstanceLess

Define if performance object is InstanceLess.

nStructSize

Size of object class.

nMaxInstanceNameLen

The maximum size of instance name string.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

It is not recommended to use this function directly to add a new performance monitor object. Instead, use the macro DECLARE_PERF_OBJECT**.**

See Also

Reference

CPerfMon Class

Concepts

Performance Monitoring

Other Resources

CPerfMon Members

CPerfMon::AddCounterDefinition

Call this method to add a new performance monitor counter.

```
HRESULT AddCounterDefinition(
   DWORD dwCounterId,
   LPCTSTR szCounterName,
   LPCTSTR szHelpString,
   DWORD dwDetailLevel,
   DWORD dwCounterType,
   ULONG nMaxCounterSize,
   UINT nOffset,
   INT nDefaultScale
) throw();
```

Parameters

dwCounterId ,

A numeric ID representing the counter.

szCounterName,

A name for this counter that will be displayed in the Performance Monitoring Console. Can be either a string or a string resource ID.

szHelpString

A help string that is displayed for this object within the Performance Monitoring Console. Can be provided as a string or a string resource ID.

dwDetailLevel

The detail level of this counter. This argument can be one of following values: Detail level Description

PERF_DETAIL_NOVICE Indicates that this counter may be meaningful to most users. This is the most common counter detail level.

PERF_DETAIL_ADVANCED Indicates that this counter is likely to be useful only to advanced users.

PERF_DETAIL_EXPERT Indicates that this counter is likely to be useful only to the most advanced users.

PERF_DETAIL_WIZARD Indicates that this counter is not likely to be useful to any users.

dwCounterType

The counter type. For more information, see Counter Data Types.

nMaxCounterSize

Indicates the maximum amount of space to reserve for the string data of a string type counter. Must be 0 for integral counter types.

nOffset

The data member representing the performance counter.

nDefaultScale

Indicates the power of 10 by which to multiply the counter value before displaying the counter. For example, if you set defscale to 2, the counter value is multiplied by 100. If you set this parameter to –3, the counter value is divided by 1,000.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

It is recommended to use the macro **DEFINE_COUNTER** and not directly use this function to add a new performance monitor counter.

See Also

Reference

CPerfMon Class

Concepts

Performance Monitoring

Other Resources

CPerfMon Members

Performance Monitoring Reference

CPerfMon::CreateInstance

Call this method to create an instance of a performance object.

```
HRESULT CreateInstance(
   DWORD dwObjectId,
   DWORD dwInstance,
   LPCWSTR szInstanceName,
   CPerfObject** ppInstance
) throw( );
template <class T>
HRESULT CreateInstance(
   DWORD dwInstance,
   LPCWSTR szInstanceName,
   T** ppInstance
) throw( );
```

Parameters

dwObjectId

The ID used in defining the desired performance object. If the desired performance object was created using nonattributed code, then this will be the object ID used with the DECLARE_PERF_OBJECT or DECLARE_PERF_OBJECT_EX macros. For performance objects defined with attributed code, use the templatized version of **CreateInstance**.

dwInstance

This parameter can be used to specify a specific instance ID of an object instance if this ID is known ahead of time. If an instance with the given ID already exists, a reference to the existing object will be returned. If this parameter is 0, a new instance will be created with a unique ID and the new object will be returned. This parameter must be 0 for instanceless objects.

szInstanceName

A Unicode string indicating the name of the instance to be created.

ppInstance

The address of a pointer that points to the new object. When **CPerfMon** creates instances of classes derived from CPerfObject, the constructors of the derived classes are not called.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Before performance counter values can be manipulated, an instance of the performance object containing the counters must be retrieved. If successful, the **CreateInstance** function creates an instance of the desired performance object and grants access using the supplied pointer.

Use this function to create instances if the object is instanceless or if objects are to be created with a well-defined scheme that lends itself to generating instances by ID. An example is creating instances to track how many of each window message an application receives; in this case, the window message itself will be the *dwInstance*.

Example

See the PerfPersist Sample.

See Also

Reference

CPerfMon Class

Concepts

Performance Monitoring

Other Resources

CPerfMon Members

Performance Monitoring Reference

CPerfMon::CreateInstanceByName

Call this method to create an instance of a performance object, given a name.

```
HRESULT CreateInstanceByName(
   DWORD dwObjectId,
   LPCWSTR szInstanceName,
   CPerfObject** ppInstance
) throw( );
template <class T>
HRESULT CreateInstanceByName(
   LPCWSTR szInstanceName,
   T** ppInstance
) throw( );
```

Parameters

dwObjectId

The ID used in defining the desired performance object. If the desired performance object was created using nonattributed code, then this will be the object ID used with the DECLARE_PERF_OBJECT or DECLARE_PERF_OBJECT_EX macros. For performance objects defined with attributed code, use the templatized version of **CreateInstanceByName**.

szInstanceName

A Unicode string indicating the name of the instance to be created.

ppInstance

The address of a pointer that is to point to the new object. When **CPerfMon** creates instances of classes derived from CPerfObject, the constructors of the derived classes are not called.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Before performance counter values can be set, an instance of the performance object containing the counters must be retrieved. If successful, this function creates an instance of the desired performance object and grants access using the supplied pointer.

Use **CreateInstanceByName**, rather than CreateInstance, to create instances for which no numeric IDs are known in advance.

Example

See the PerformanceCounter Sample and the PerformanceScribble Sample.

See Also

Reference

CPerfMon Class

Concepts

Performance Monitoring

Other Resources

CPerfMon Members

Performance Monitoring Reference

CPerfMon::Initialize

Call this method to initialize the performance monitor object.

```
HRESULT Initialize( ) throw();
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The **Initialize** function initializes the CPerfMon object and should be called before any other member functions are called.

Example

See the PerformanceCounter Sample and the PerformanceScribble Sample.

See Also

Reference

CPerfMon Class

Concepts

Performance Monitoring

Other Resources

CPerfMon Members

Performance Monitoring Reference

CPerfMon::LoadFromXML

Call this method to load previously saved performance data from a stream.

```
HRESULT LoadFromXML(
   IStream * pStream
) throw(...);
```

Parameters

pStream

A pointer to a stream containing previously saved performance data.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The **LoadFromXML** function, along with the PersistToXML function, can be used persist performance data over multiple executions.

See Also

Reference

CPerfMon Class

Concepts

Performance Monitoring

Other Resources

CPerfMon Members

Performance Monitoring Reference

CPerfMon::LockPerf

Call this method to lock a performance monitor object for synchronized access.

```
HRESULT LockPerf(
   DWORD dwTimeout = INFINITE
) throw();
```

Parameters

dwTimeout

The time, in milliseconds, that is acceptable to wait for access.

Return Value

Returns **S_OK** on success, **WAIT_TIMEOUT** if the lock attempt fails before the *dwTimeout* interval elapses, or an error **HRESULT** on failure.

Remarks

The **LockPerf** function attempts to lock the performance monitor object for exclusive access. If no timeout is specified, **LockPerf** will not return until exclusive access has been granted.

See the CPerfLock class for an alternative to using the **LockPerf** function.

See Also

Reference

CPerfMon Class

CPerfMon::UnlockPerf

Concepts

Performance Monitoring

Other Resources

CPerfMon Members

Performance Monitoring Reference

CPerfMon::PersistToXML

Call this method to save a performance monitor object to a stream.

```
HRESULT PersistToXML(
   IStream* pStream,
   BOOL bFirst = TRUE,
   BOOL bLast = TRUE
) throw(...);
```

Parameters

pStream

A pointer to the stream to which the performance data should be written.

bFirst

A Boolean indicating whether this is the first object to be saved to the stream. If **TRUE**, the **PersistToXML** function writes the necessary version and root element to the stream.

bLast

A Boolean indicating whether this is the last object to be saved to the stream. If **TRUE**, the **PersistToXML** function closes the root element.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The **PersistToXML** function, along with the CPerfMon::LoadFromXML function, can be used to generate performance data that spans over multiple executions.

See Also

Reference

CPerfMon Class

Concepts

Performance Monitoring

Other Resources

CPerfMon Members

Performance Monitoring Reference

CPerfMon::Register

Call this method to register this performance monitor object with the Windows performance subsystem.

```
HRESULT Register(
   LPCTSTR szOpenFunc,
   LPCTSTR szCollectFunc,
   LPCTSTR szCloseFunc,
   HINSTANCE hDllInstance = _AtlBaseModule.GetModuleInstance()
) throw();
```

Parameters

szOpenFunc

A string containing the name of the function responsible for responding to initialize performance monitor requests.

szCollectFunc

A string containing the name of the function responsible for responding to collect requests from performance monitor queries.

szCloseFunc

A string containing the name of the function responsible for shutting down performance monitor transactions.

hDllInstance

The instance handle for the DLL providing the open, collect, and close functions.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The **Register** function adds the given function named and DLL instance handle to the registry so that the Windows performance monitor mechanisms can query your application. ATL Server normally calls this function and the Unregister function implicitly, so these functions do not need to appear in your code.

See Also

Reference

CPerfMon Class
CPerfMon::Unregister
PERFREG_ENTRY

Concepts

Performance Monitoring

Other Resources

CPerfMon Members
Performance Monitoring Reference

CPerfMon::RegisterAllStrings

Call this method to register all strings in a performance monitor object.

```
HRESULT RegisterAllStrings(
   HINSTANCE hResInstance = NULL
) throw();
```

Parameters

hResInstance

A DLL instance handle. By default, the resource instance of the current module is used.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The **RegisterAllStrings** function copies all performance counter strings to the registry, for use in viewing performance data. If the supplied resource instance supports multiple languages, then all strings for each language available are copied.

See Also

Reference

CPerfMon Class

Concepts

Performance Monitoring

Other Resources

CPerfMon Members

Performance Monitoring Reference

CPerfMon::RegisterStrings

Call this method to register the strings in a performance monitor object.

```
HRESULT RegisterStrings(
   LANGID wLanguage = MAKELANGID(LANG_NEUTRAL, SUBLANG_NEUTRAL),
   HINSTANCE hResInstance = _AtlBaseModule.GetResourceInstance()
) throw();
```

Parameters

wLanguage

Indicates the language for which strings should be registered.

hResInstance

A DLL instance handle. By default, the resource instance of the current module is used.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The **RegisterStrings** function copies performance counter strings to the registry for use in viewing performance data.

See Also

Reference

CPerfMon Class

Concepts

Performance Monitoring

Other Resources

CPerfMon Members

Performance Monitoring Reference

CPerfMon::ReleaseInstance

Call this method to release an instance of a performance monitor object.

```
HRESULT ReleaseInstance(
   CPerfObject* pInstance
) throw();
```

Parameters

pInstance

The address of the performance object to be released.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Performance objects created previously with CreateInstance or CreateInstanceByName can be destroyed using the **ReleaseInstance** function, freeing the shared memory used internally to store performance counter values.

See Also

Reference

CPerfMon Class

Concepts

Performance Monitoring

Other Resources

CPerfMon Members

Performance Monitoring Reference

CPerfMon::UnInitialize

Call this method to uninitialize a performance monitor object.

```
void UnInitialize( ) throw();
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The **UnInitialize** function is usually called shortly before application termination.

Example

See the PerformanceCounter Sample and the PerformanceScribble Sample.

See Also

Reference

CPerfMon Class

Concepts

Performance Monitoring

Other Resources

CPerfMon Members

Performance Monitoring Reference

CPerfMon::UnlockPerf

Call this method to unlock a performance monitor object.

```
void UnlockPerf( ) throw();
```

Remarks

Releases previously set locks on the performance monitor object. See the CPerfLock class for an alternative to using this function.

See Also

Reference

CPerfMon Class

CPerfMon::LockPerf

Concepts

Performance Monitoring

Other Resources

CPerfMon Members

Performance Monitoring Reference

CPerfMon::Unregister

Call this method to unregister this performance monitor object from the Windows performance subsystem.

```
HRESULT Unregister( ) throw();
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The **Unregister** function undoes the registration performed by the Register function. Normally, both functions are called implicitly by ATL Server and therefore will not appear in your code.

See Also

Reference

CPerfMon Class

Concepts

Performance Monitoring

Other Resources

CPerfMon Members

Performance Monitoring Reference

CPerfMonRequestStats Class

Use this class as the argument for the *CRequestStatClass* template parameter to CIsapiExtension if you want to expose request-handling statistics as PerfMon counters.

```
class CPerfMonRequestStats
```

Remarks

The request handling statistics captured by this class can be accessed through the IRequestStats interface implemented by CIsapiExtension and through PerfMon performance counters. For more information on the counters exposed by this class, see Predefined ISAPI Performance Counters

For complete details on the actions that you need to take to expose the performance counters, see Enabling ISAPI Performance Monitor Support.

This class conforms to the Request Statistics Archetype. See the documentation for that archetype for other conforming classes.

Requirements

Header: atlisapi.h

See Also

Reference

Request Statistics Archetype

Other Resources

CPerfMonRequestStats Members

ATL Server Classes

CPerfMonRequestStats Members

Methods

CPerfMonRequestStats	The constructor.
GetActiveThreads	Call this method to get the number of threads currently handling a request.
GetAvgResponseTime	Call this method to get the average response time for handling a request.
GetCurrWaiting	Call this method to get the current number of queued requests.
GetFailedRequests	Call this method to get the total number of failed requests.
GetMaxWaiting	Call this method to get the maximum number of requests that have been queued at any one time since the ISAPI extension was loaded.
GetTotalRequests	Call this method to get the total number of handled requests.
Initialize	This method is called by CIsapiExtension to initialize the object.
OnRequestDequeued	This method is called by CIsapiExtension when a request is removed from the queue and passed to a w orker thread.
OnRequestReceived	This method is called by CIsapiExtension when a request is received and added to the queue.
RequestHandled	This method is called by CIsapiExtension when a request has been handled.
Uninitialize	This method is called by CIsapiExtension to uninitialize the object.

See Also

Reference

CPerfMonRequestStats Class

CPerfMonRequestStats Methods

For information about the methods in **CPerfMonRequestStats**, see CPerfMonRequestStats Members.

CPerfMonRequestStats::CPerfMonRequestStats

The constructor.

```
CPerfMonRequestStats( ) throw( );
```

Remarks

Initializes internal data members. No major work is done until CPerfMonRequestStats::Initialize is called.

See Also

Reference

CPerfMonRequestStats Class

CPerfMonRequestStats::Initialize

Other Resources

CPerfMonRequestStats Members

CPerfMonRequestStats::GetActiveThreads

Call this method to get the number of threads currently handling a request.

```
long GetActiveThreads( ) throw( );
```

Return Value

Returns the number of threads currently handling a request.

Remarks

Returns information for the current ISAPI extension.

See Also

Reference

CPerfMonRequestStats Class

Other Resources

CPerfMonRequestStats Members

CPerfMonRequestStats::GetAvgResponseTime

Call this method to get the average response time for handling a request.

```
long GetAvgResponseTime( ) throw( );
```

Return Value

Returns the average response time for handling a request.

Remarks

Returns information for the current ISAPI extension. The average response time is measured as the total response time divided by the total number of requests handled. Information about currently queued requests is not included in this figure.

See Also

Reference

CPerfMonRequestStats Class

Other Resources

CPerfMonRequestStats Members

CPerfMonRequestStats::GetCurrWaiting

Call this method to get the current number of queued requests.

```
long GetCurrWaiting( ) throw( );
```

Return Value

Returns the current number of queued requests.

Remarks

Returns information for the current ISAPI extension.

See Also

Reference

CPerfMonRequestStats Class

Other Resources

CPerfMonRequestStats Members

CPerfMonRequestStats::GetFailedRequests

Call this method to get the total number of failed requests.

```
long GetFailedRequests( ) throw( );
```

Return Value

Returns the total number of failed requests.

Remarks

Returns information for the current ISAPI extension.

See Also

Reference

CPerfMonRequestStats Class

Other Resources

CPerfMonRequestStats Members

CPerfMonRequestStats::GetMaxWaiting

Call this method to get the maximum number of requests that have been queued at any one time since the ISAPI extension was loaded.

```
long GetMaxWaiting( ) throw( );
```

Return Value

Returns the maximum number of queued requests.

Remarks

Returns information for the current ISAPI extension.

See Also

Reference

CPerfMonRequestStats Class

Other Resources

CPerfMonRequestStats Members

CPerfMonRequestStats::GetTotalRequests

Call this method to get the total number of handled requests.

```
long GetTotalRequests( ) throw( );
```

Return Value

Returns the total number of handled requests.

Remarks

Returns information for the current ISAPI extension.

Add the result of **GetTotalRequests** and CPerfMonRequestStats::GetCurrWaiting to get the total number of requests received.

See Also

Reference

CPerfMonRequestStats Class

Other Resources

CPerfMonRequestStats Members

CPerfMonRequestStats::Initialize

This method is called by CIsapiExtension to initialize the object.

```
HRESULT Initialize( ) throw( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Creates and initializes the performance monitor objects used to track the request handling statistics. One object with the name of the ISAPI extension DLL tracks the statistics for that DLL, and another object named **_Total** tracks statistics for all PerfMon-enabled ATL Server ISAPI extension DLLs on the system.

See Also

Reference

CPerfMonRequestStats Class

Other Resources

CPerfMonRequestStats Members

CPerfMonRequestStats::OnRequestDequeued

This method is called by CIsapiExtension when a request is removed from the queue and passed to a worker thread.

```
void OnRequestDequeued( ) throw( );
```

Remarks

Adjusts the statistics reported by the performance counters.

See Also

Reference

CPerfMonRequestStats Class

Other Resources

CPerfMonRequestStats Members

CPerfMonRequestStats::OnRequestReceived

This method is called by CIsapiExtension when a request is received and added to the queue.

```
void OnRequestReceived( ) throw( );
```

Remarks

Adjusts the statistics reported by the performance counters.

See Also

Reference

CPerfMonRequestStats Class

Other Resources

CPerfMonRequestStats Members

CPerfMonRequestStats::RequestHandled

This method is called by CIsapiExtension when a request has been handled.

```
void RequestHandled(
   AtlServerRequest * pRequestInfo,
   BOOL bSuccess
) throw( );
```

Parameters

pRequestInfo

The information about the current request.

bSuccess

TRUE if the request was successfully handled, FALSE otherwise.

Remarks

Adjusts the statistics reported by the performance counters.

See Also

Reference

CPerfMonRequestStats Class

Other Resources

CPerfMonRequestStats Members

AtlServerRequest Members

CPerfMonRequestStats::Uninitialize

This method is called by CIsapiExtension to uninitialize the object.

```
void Uninitialize( ) throw( );
```

Remarks

Frees the performance counters.

See Also

Reference

CPerfMonRequestStats Class

Other Resources

CPerfMonRequestStats Members

CPerfObject Structure

This structure represents a performance monitor object.

```
struct CPerfObject
```

Remarks

The **CPerfObject** structure serves as a base class for performance monitoring objects. Classes derived from **CPerfObject** are expected to contain data members that serve as performance counters. For attributed code, the **CPerfObject** class is used implicitly.

Note
When CPerfMon creates instances of classes derived from **CPerfObject**, the constructors of the derived classes are not called.

Requirements

Header: atlperf.h

Example

See the PerformanceCounter Sample and the PerformanceScribble Sample.

See Also

Concepts

Performance Monitoring

Other Resources

ATL Server Classes

Performance Monitoring Reference

CPerfRequestStatObject Structure

This structure is used in the implementation of CPerfMonRequestStats.

```
struct CPerfRequestStatObject :
   public CPerfObject,
   public CRequestStats
```

Remarks

This structure uses the ATL Server performance monitoring macros to expose the CRequestStats data members as performance counters. There is no need to use this class directly.

Requirements

Header: atlisapi.h

See Also

Reference

CPerfMonRequestStats Class
CPerfObject Structure
CRequestStats Class

Concepts

Performance Monitoring

Other Resources

ATL Server Classes
Performance Monitoring Reference

CPerfStatClass Class

Use this class anywhere that a class conforming to the cache statistics archetype is expected if you want to collect cache statistics in the class's data members and expose the information as PerfMon counters.

```
class CPerfStatClass :
   public CStdStatClass
```

Remarks

This class provides a public interface that conforms to the cache statistics archetype. The methods (most of which are implemented by CStdStatClass) store the statistics in the class's data members using thread-safe interlocked operations.

This class provides implementations of Initialize and Uninitialize that create and register the performance counters necessary to expose this data to PerfMon. See Predefined ISAPI Performance Counters for more details.

Classes that make use of the cache statistics archetype also typically expose the statistics gathered by this class through the IMemoryCacheStats interface.

Typically, you don't need to call the methods provided by this class directly.

Requirements

Header: atlcache.h

See Also

Reference

CStdStatClass Class

Concepts

Performance Monitoring

Other Resources

Caching

CPerfStatClass Members

Caching Reference

Performance Monitoring Reference

CPerfStatClass Members

Methods

Initialize	Call this method to initialize the object.
Uninitialize	Call this method to uninitialize the object.

See Also

Reference

CPerfStatClass Class

CPerfStatClass Methods

For information about the methods in **CPerfStatClass**, see CPerfStatClass Members

CPerfStatClass::Initialize

Call this method to initialize the object.

```
HRESULT Initialize(
   LPWSTR szName = NULL
);
```

Parameters

szName

The name of the performance object used to expose information about the cache. If *szName* is **NULL**, the performance object is named after the module using this class.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Creates and initializes the performance monitor objects used to track the cache statistics.

See Also

Reference

CPerfStatClass Class

CPerfStatClass::Uninitialize

Other Resources

CPerfStatClass Members

CPerfStatClass::Uninitialize

Call this method to uninitialize the object.

```
HRESULT Uninitialize( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Frees the performance counters.

See Also

Reference

CPerfStatClass Class

CPerfStatClass::Initialize

Other Resources

CPerfStatClass Members

CPerfStatObject Structure

This structure is used in the implementation of CStdStatClass and CPerfStatClass.

```
struct CPerfStatObject :
   public CPerfObject
```

Remarks

This structure uses the ATL Server performance monitoring macros to expose its data members as performance counters. There is no need to use this class directly.

Requirements

Header: atlcache.h

See Also

Concepts

Performance Monitoring

Other Resources

CPerfStatObject Members

Performance Monitoring Reference

CPerfStatObject Members

Data Members

m_nCurrentAllocations	The current number of bytes allocated by the cache.
m_nCurrentEntries	The current number of items in the cache.
m_nHitCount	The number of times the cache successfully returned an item.
m_nMaxAllocations	The maximum number of bytes allocated by the cache.
m_nMaxEntries	The maximum number of items in the cache.
m_nMissCount	The number of times the cache did not contain a requested item.

See Also

Reference

CPerfStatObject Structure

CPerfStatObject Data Members

For information about the data members in **CPerfStatObject**, see CPerfStatObject Members

CPerfStatObject::m_nCurrentAllocations

The current number of bytes allocated by the cache.

```
DWORD m_nCurrentAllocations;
```

See Also

Reference

CPerfStatObject Structure

Other Resources

CPerfStatObject Members

CPerfStatObject::m_nCurrentEntries

The current number of items in the cache.

```
DWORD m_nCurrentEntries;
```

See Also

Reference

CPerfStatObject Structure

Other Resources

CPerfStatObject Members

CPerfStatObject::m_nHitCount

The number of times the cache successfully returned an item.

```
DWORD m_nHitCount;
```

See Also

Reference

CPerfStatObject Structure

Other Resources

CPerfStatObject Members

CPerfStatObject::m_nMaxAllocations

The maximum number of bytes allocated by the cache.

```
DWORD m_nMaxAllocations;
```

See Also

Reference

CPerfStatObject Structure

Other Resources

CPerfStatObject Members

CPerfStatObject::m_nMaxEntries

The maximum number of items in the cache.

```
DWORD m_nMaxEntries;
```

See Also

Reference

CPerfStatObject Structure

Other Resources

CPerfStatObject Members

CPerfStatObject::m_nMissCount

The number of times the cache did not contain a requested item.

```
DWORD m_nMissCount;
```

See Also

Reference

CPerfStatObject Structure

Other Resources

CPerfStatObject Members

CReadStreamOnInet Class

This class provides an implementation of IStream that implements Read to return data from a WinInet **HINTERNET** handle returned from a previous call to **InternetOpenUrl**, **FtpOpenFile**, **GopherOpenFile**, or **HttpOpenRequest**.

```
class CReadStreamOnInet :
   public IStreamImpl
```

Remarks

The WinInet **HINTERNET** handle from which data is to be read can be passed to Init.

Requirements

Header: atlsoap.h

See Also

Reference

IStreamImpl Class

Other Resources

CReadStreamOnInet Members

CReadStreamOnInet Members

Methods

AddRef	Implementation of **IUnknown::AddRef**.
CReadStreamOnInet	The constructor.
Init	Call this method to pass the WinInet handle used to provide the data returned by Read.
QueryInterface	Implementation of **IUnknown::QueryInterface**.
Read	Implementation of ISequentialStream::Read.
Release	Implementation of **IUnknown::Release**.

See Also

Reference

CReadStreamOnInet Class
IStreamImpl Class

CReadStreamOnInet Methods

For information about the methods in **CReadStreamOnInet**, see CReadStreamOnInet Members

CReadStreamOnInet::AddRef

Implementation of **IUnknown::AddRef**.

```
ULONG __stdcall AddRef( );
```

Return Value

Always returns 1.

Remarks

This class does not implement lifetime control via reference counting.

See Also

Reference

CReadStreamOnInet Class

CReadStreamOnInet::Release

Other Resources

CReadStreamOnInet Members

CReadStreamOnInet::CReadStreamOnInet

The constructor.

```
CReadStreamOnInet( );
```

Remarks

Initializes internal data members.

See Also

Reference

CReadStreamOnInet Class

Other Resources

CReadStreamOnInet Members

CReadStreamOnInet::Init

Call this method to pass the WinInet handle used to provide the data returned by Read.

```
void Init(
   HINTERNET hFile
);
```

Parameters

hFile

Valid **HINTERNET** handle returned from a previous call to **InternetOpenUrl**, **FtpOpenFile**, **GopherOpenFile**, or **HttpOpenRequest**.

See Also

Reference

CReadStreamOnInet Class

Other Resources

CReadStreamOnInet Members

CReadStreamOnInet::QueryInterface

Implementation of **IUnknown::QueryInterface**.

```
HRESULT __stdcall QueryInterface(
   REFIID riid,
   void** ppv
);
```

Remarks

Objects of this class can be successfully queried for the **IUnknown**, ISequentialStream, and IStream interfaces.

See Also

Reference

CReadStreamOnInet Class

Other Resources

CReadStreamOnInet Members

CReadStreamOnInet::Read

Implementation of ISequentialStream::Read.

```
HRESULT STDMETHODCALLTYPE Read(
   void * pDest,
   ULONG dwMaxLen,
   ULONG * pdwRead
);
```

Remarks

Reads client data from the handle passed to Init using InternetReadFile.

See Also

Reference

CReadStreamOnInet Class

Other Resources

CReadStreamOnInet Members

CReadStreamOnInet::Release

Implementation of **IUnknown::Release**.

```
ULONG __stdcall Release( );
```

Return Value

Always returns 1.

Remarks

This class does not implement lifetime control via reference counting.

See Also

Reference

CReadStreamOnInet Class

CReadStreamOnInet::AddRef

Other Resources

CReadStreamOnInet Members

CReadStreamOnSocket Class

This class provides an implementation of IStream that implements Read to return data from the body of an HTTP response provided by CAtlHttpClientT.

```
template <
   typename TSocketClass
>
class CReadStreamOnSocket :
   public IStreamImpl
```

Parameters

TSocketClass

The class used to provide the underlying network socket support. For the current version of the library this parameter should be set to the internal class, **ZEvtSyncSocket**.

Remarks

The HTTP client object from which data is to be read can be passed to Init.

Requirements

Header: atlsoap.h

See Also

Reference

IStreamImpl Class

Other Resources

CReadStreamOnSocket Members

ATL Server Classes

CReadStreamOnSocket Members

Methods

AddRef	Implementation of **IUnknown::AddRef**.
CReadStreamOnSocket	The constructor for the stream.
Init	Call this method to pass a pointer to the HTTP client object whose body is to be used to provide the da ta returned by Read.
QueryInterface	Implementation of **IUnknown::QueryInterface**.
Read	Implementation of ISequentialStream::Read.
Release	Implementation of **IUnknown::Release**.

See Also

Reference

CReadStreamOnSocket Class
IStreamImpl Class

CReadStreamOnSocket Methods

For information about the methods in **CReadStreamOnSocket**, see CReadStreamOnSocket Members

CReadStreamOnSocket::AddRef

Implementation of **IUnknown::AddRef**.

```
ULONG __stdcall AddRef( );
```

Return Value

Always returns 1.

Remarks

This class does not implement lifetime control using reference counting.

See Also

Reference

CReadStreamOnSocket Class

CReadStreamOnSocket::Release

Other Resources

CReadStreamOnSocket Members

CReadStreamOnSocket::CReadStreamOnSocket

The constructor for the stream.

```
CReadStreamOnSocket( );
```

Remarks

Initializes internal data members.

See Also

Reference

CReadStreamOnSocket Class

Other Resources

CReadStreamOnSocket Members

CReadStreamOnSocket::Init

Call this method to pass a pointer to the HTTP client object whose body is to be used to provide the data returned by Read.

```
BOOL Init(
   CAtlHttpClientT< TSocketClass > * pSocket
);
```

Parameters

pSocket

The HTTP client object.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CReadStreamOnSocket Class

CReadStreamOnSocket::Read

Other Resources

CReadStreamOnSocket Members

CReadStreamOnSocket::QueryInterface

Implementation of **IUnknown::QueryInterface**.

```
HRESULT __stdcall QueryInterface(
   REFIID riid,
   void** ppv
);
```

Remarks

Objects of this class can be successfully queried for the IUnknown, ISequentialStream, and IStream interfaces.

See Also

Reference

CReadStreamOnSocket Class

Other Resources

CReadStreamOnSocket Members

CReadStreamOnSocket::Read

Implementation of ISequentialStream::Read.

```
HRESULT __stdcall Read(
   void* pDest,
   ULONG nMaxLen,
   ULONG* pnRead
);
```

Remarks

Reads data from the body of the HTTP response in the HTTP client object passed to Init.

See Also

Reference

CReadStreamOnSocket Class

Other Resources

CReadStreamOnSocket Members

CReadStreamOnSocket::Release

Implementation of **IUnknown::Release**.

```
ULONG __stdcall Release( );
```

Return Value

Always returns 1.

Remarks

This class does not implement lifetime control using reference counting.

See Also

Reference

CReadStreamOnSocket Class

CReadStreamOnSocket::AddRef

Other Resources

CReadStreamOnSocket Members

CRequestHandlerT Class

This class acts as the base class for all user request handlers.

```
template <
   class THandler,
   class ThreadModel     = CComSingleThreadModel,
   class TagReplacerType = CHtmlTagReplacer< THandler >
>
class CRequestHandlerT :
   public TagReplacerType,
   public CComObjectRootEx< ThreadModel >,
   public IRequestHandlerImpl< THandler >
```

Parameters

THandler

A class derived from **CRequestHandlerT**.

ThreadModel

The class whose methods implement the desired threading model.

TagReplacerType

A class implementing the ITagReplacer interface. Typically CHtmlTagReplacer or a derived class.

Remarks

This is the base class for all user request handlers. This class provides methods that can be overridden to provide opportunities for you to initialize your request handler (CheckValidRequest, ValidateAndExchange) or to pass information back to the base classes (MaxFormSize, FormFlags). It also provides access to the server context (GetContext), and the request and response objects for the current request (m_HttpRequest, m_HttpResponse).

Requirements

Header: atlstencil.h

Example

See the following samples:

- ConfirmUser Sample
- Input Sample
- MantaWeb Sample
- RegExp Sample
- ShowFiles Sample
- ShowForm Sample
- ShowImage Sample
- ShowRequest Sample
- SRFSyntax Sample

See Also

Tasks

RegExp Sample: Demonstrates ATL Regular Expression Classes
SRFSyntax Sample: Server Response File Demonstration

Reference

HelloWorld
ShowRequest
ConfirmUser

IRequestHandlerImpl Class
CHtmlTagReplacer Class
CComSingleThreadModel Class

Other Resources

CRequestHandlerT Members
ATL Server Classes

CRequestHandlerT Members

Methods

CheckValidRequest	Override this method to check if the request is valid using IHttpServerContext.
ClearResponse	Call this method to clear the response object of any headers and buffered data.
CRequestHandlerT	The constructor.
~CRequestHandlerT	The destructor.
FormFlags	Override this nonvirtual method to return the flags describing how form processing should be carried out.
GetContext	Call this method to get access to the context for the current request.
GetResourceInstance	Call this method to get the handle of the module providing resources.
HandleRequest	Override this method to handle the current request.
InitializeChild	This method is called to initialize the request handler when it is used from a subhandler tag or include tag.
InitializeHandler	This method is called to initialize the request handler when it is used from a handler tag or loaded by a ny means except a subhandler tag or include tag.
InitializeInternal	This method initializes the current object.
MaxFormSize	Override this nonvirtual method to return the maximum size of a form that should be processed by thi s request handler.
ServerTransferRequest	Call this method to transfer request processing to another request handler.
Uninitialize	Override this nonvirtual method to uninitialize the request handler.
ValidateAndExchange	Override this nonvirtual method to initialize the request handler.

Data Members

m_dwRequestType	The type of the current request.
m_HttpRequest	Use the request object to get information about the current request.
m_HttpResponse	Use the response object to write the response to the client.
m_pRequestInfo	The information about the request to be handled.

See Also

Reference

CRequestHandlerT Class

CRequestHandlerT Methods

For information about the methods in **CRequestHandlerT**, see CRequestHandlerT Members.

CRequestHandlerT::CheckValidRequest

Override this method to check if the request is valid using IHttpServerContext.

```
HTTP_CODE CheckValidRequest( );
```

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

This method is provided to allow you to do simple checking of the validity of a request using the IHttpServerContext interface returned by GetContext. This method is called before the relatively expensive initialization of the request object has occurred, so do not use m_HttpRequest or any method that uses that member in this method. The response object, m_HttpResponse, has been initialized by the time this method is called, so you can use that.

See Also

Tasks

Initializing Request Handlers

Reference

CRequestHandlerT Class

Other Resources

CRequestHandlerT Members

CRequestHandlerT::ClearResponse

Call this method to clear the response object of any headers and buffered data.

```
void ClearResponse( ) throw( );
```

Remarks

Equivalent to calling CHttpResponse::ClearResponse on m_HttpResponse.

See Also

Reference

CRequestHandlerT Class

Other Resources

CRequestHandlerT Members

CRequestHandlerT::CRequestHandlerT

The constructor.

```
CRequestHandlerT( ) throw( );
```

Remarks

Initializes internal data members.

See Also

Reference

CRequestHandlerT Class

Other Resources

CRequestHandlerT Members

CRequestHandlerT::~CRequestHandlerT

The destructor.

```
~CRequestHandlerT( ) throw( );
```

Remarks

Frees any resources acquired by the object.

See Also

Reference

CRequestHandlerT Class

Other Resources

CRequestHandlerT Members

CRequestHandlerT::FormFlags

Override this nonvirtual method to return the flags describing how form processing should be carried out.

```
inline DWORD FormFlags( );
```

Return Value

Return one or more of the ATL Server form flags.

Remarks

The implementation of this method provided by this class returns ATL_FORM_FLAG_IGNORE_FILES. You only need to override this method if the default implementation does not meet your needs.

Example

See the following samples:

- ConfirmUser Sample
- RegExp Sample
- ShowFiles Sample
- ShowForm Sample
- ShowImage Sample

See Also

Reference

CRequestHandlerT Class

ATL_FORM_FLAGS

Other Resources

CRequestHandlerT Members

CRequestHandlerT::GetContext

Call this method to get access to the context for the current request.

```
template <typename Interface>
HRESULT GetContext(
   Interface** ppInterface
) throw(...);
HRESULT GetContext(
   REFIID riid,
   void** ppv
);
```

Parameters

riid

[in] The interface ID of the requested interface.

ppv, *ppInterface*

[out] Address of the pointer variable that, on success, receives the interface pointer to the requested interface.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

You can successfully query for the following interfaces using **GetContext**:

- IhttpServerContext
- IhttpRequestLookup
- IServiceProvider

Typically this method is an override for ITagReplacerImpl::GetContext.

See Also

Reference

CRequestHandlerT Class

Other Resources

CRequestHandlerT Members

CRequestHandlerT::GetResourceInstance

Call this method to get the handle of the module providing resources.

```
HINSTANCE GetResourceInstance( );
```

Return Value

Returns the handle of the module providing resources, or NULL if no handle is available.

Remarks

If m_pRequestInfo is not NULL, returns m_pRequestInfo->hInstDll. Otherwise, returns NULL.

Typically this method is an override for ITagReplacerImpl::GetResourceInstance.

See Also

Reference

CRequestHandlerT Class

Other Resources

CRequestHandlerT Members

CRequestHandlerT::HandleRequest

Override this method to handle the current request.

```
HTTP_CODE HandleRequest(
   AtlServerRequest * pRequestInfo,
   IServiceProvider* /* pServiceProvider */
);
```

Parameters

pRequestInfo

The information about the current request to be handled.

pServiceProvider

The service provider.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

HandleRequest is called to perform processing of HTTP requests. You can override this method in your derived class if you need to change the way the request is processed, but the implementation provided by this class is probably sufficient for most needs. In essence, it calls LoadStencil, and RenderStencil through a pointer to the most derived class. These methods (and the class's template parameters) provide points at which you can customize the request processing without overriding **HandleRequest**.

See Also

Reference

CRequestHandlerT Class

Other Resources

CRequestHandlerT Members

CRequestHandlerT::InitializeChild

This method is called to initialize the request handler when it is used from a subhandler tag or include tag.

```
HTTP_CODE InitializeChild(
   AtlServerRequest* pRequestInfo,
   IServiceProvider* pProvider,
   IHttpRequestLookup* pRequestLookup
);
```

Parameters

pRequestInfo

The information about the request to be handled.

pProvider

The service provider.

pRequestLookup

The interface providing access to the form variables, query parameters, and cookies of the parent request.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

This implementation of IRequestHandler::InitializeChild initializes the request and response objects, and calls ValidateAndExchange.

See Also

Tasks

Initializing Request Handlers

Reference

CRequestHandlerT Class

IRequestHandler::InitializeChild

CRequestHandlerT::ValidateAndExchange

Other Resources

CRequestHandlerT Members

CRequestHandlerT::InitializeHandler

This method is called to initialize the request handler when it is used from a handler tag or loaded by any means except a subhandler tag or include tag.

```
HTTP_CODE InitializeHandler(
   AtlServerRequest* pRequestInfo,
   IServiceProvider* pProvider
);
```

Parameters

pRequestInfo

The information about the request to be handled.

pProvider

The service provider.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

This implementation of IRequestHandler::InitializeHandler initializes the response object, calls CRequestHandlerT::CheckValidRequest, initializes the request object, and calls ValidateAndExchange.

See Also

Tasks

Initializing Request Handlers

Reference

CRequestHandlerT Class
IRequestHandler::InitializeHandler
CRequestHandlerT::CheckValidRequest
CRequestHandlerT::ValidateAndExchange

Other Resources

CRequestHandlerT Members

CRequestHandlerT::InitializeInternal

This method initializes the current object.

```
HTTP_CODE InitializeInternal(
   AtlServerRequest* pRequestInfo,
   IServiceProvider* pProvider
);
```

Parameters

pRequestInfo

The information about the request to be handled.

pProvider

The service provider.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

CRequestHandlerT Class

Other Resources

CRequestHandlerT Members

CRequestHandlerT::MaxFormSize

Override this nonvirtual method to return the maximum size of a form that should be processed by this request handler.

```
inline DWORD MaxFormSize( );
```

Return Value

Returns the maximum size of a form that should be processed by this request handler.

Remarks

The implementation of this method provided by this class returns DEFAULT_MAX_FORM_SIZE. You only need to override this method if the default implementation does not meet your needs.

Example

See the ShowImage Sample.

See Also

Tasks

Limiting the Size of a Request

Reference

CRequestHandlerT Class

Other Resources

CRequestHandlerT Members

CRequestHandlerT::ServerTransferRequest

Call this method to transfer request processing to another request handler.

```
HTTP_CODE ServerTransferRequest(
   LPCSTR szRequest,
   bool bContinueAfterTransfer = false,
   WORD nCodePage = 0,
   CStencilState * pState = NULL
) throw(...);
```

Parameters

szRequest

The URL of the new request handler.

bContinueAfterTransfer

True if processing should continue in the current handler after the new handler has finished, false otherwise.

nCodePage

The code page.

pState

A pointer to state information.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

This method calls IIsapiExtension::TransferRequest passing information about the current request.

See Also

Reference

CRequestHandlerT Class
IIsapiExtension::TransferRequest
CStencilState Structure

Other Resources

CRequestHandlerT Members

CRequestHandlerT::Uninitialize

Override this nonvirtual method to uninitialize the request handler.

```
HTTP_CODE Uninitialize(
   HTTP_CODE hcError
);
```

Parameters

hcError

The current ATL Server status code for the request.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

Put your cleanup code in the override for this method.

See Also

Reference

CRequestHandlerT Class

Other Resources

CRequestHandlerT Members

CRequestHandlerT::ValidateAndExchange

Override this nonvirtual method to initialize the request handler.

```
HTTP_CODE ValidateAndExchange( );
```

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

Add initialization code to your override of this method. Form processing can be done from this method. Both the request and response object are available for use from this method. The default implementation of the method in this class simply returns HTTP_SUCCESS.

Example

See the MantaWeb Sample, the ConfirmUser Sample, and the Input Sample.

See Also

Reference

CRequestHandlerT Class

CRequestHandlerT::CheckValidRequest

Other Resources

CRequestHandlerT Members

CRequestHandlerT Data Members

For information about the data members in **CRequestHandlerT**, see CRequestHandlerT Members.

CRequestHandlerT::m_dwRequestType

The type of the current request.

```
ATLSRV_REQUESTTYPE m_dwRequestType;
```

Remarks

Tells you whether the request was for a SRF file or a DLL. Can be ATLSRV_REQUEST_STENCIL or ATLSRV_REQUEST_DLL.

See Also

Reference

CRequestHandlerT Class

ATLSRV_REQUESTTYPE

Other Resources

CRequestHandlerT Members

CRequestHandlerT::m_HttpRequest

Use the request object to get information about the current request.

```
CHttpRequest m_HttpRequest;
```

Example

See the MantaWeb Sample and the ShowRequest Sample.

See Also

Reference

CRequestHandlerT Class

CHttpRequest Class

Other Resources

CRequestHandlerT Members

CRequestHandlerT::m_HttpResponse

Use the response object to write the response to the client.

```
CHttpResponse m_HttpResponse;
```

Example

See the MantaWeb Sample, the ConfirmUser Sample, and the SRFSyntax Sample.

See Also

Reference

CRequestHandlerT Class

CHttpResponse Class

Other Resources

CRequestHandlerT Members

CRequestHandlerT::m_pRequestInfo

The information about the request to be handled.

```
AtlServerRequest* m_pRequestInfo;
```

See Also

Reference

CRequestHandlerT Class

AtlServerRequest Structure

Other Resources

CRequestHandlerT Members

CRequestPerfMon Class

This class is used in the implementation of CPerfMonRequestStats.

```
class CRequestPerfMon :
   public CPerfMon
```

Remarks

This class uses the ATL Server performance monitoring macros to manage CPerfRequestStatObject performance objects. There is no need to use this class directly except to wrap it in a PERFREG_ENTRY macro to ensure that the performance counters are properly registered. See Enabling ISAPI Performance Monitor Support for more details.

Requirements

Header: atlisapi.h

See Also

Reference

CPerfMon Class

CPerfMonRequestStats Class

Concepts

Predefined ISAPI Performance Counters

Other Resources

ATL Server Classes

CRequestStats Class

This structure is used to collect request-handling statistics.

```
struct CRequestStats
```

Remarks

CRequestStats provides three methods that can be called to indicate the progress of a request:

- CRequestStats::OnRequestReceived
- CRequestStats::OnRequestDequeued
- CRequestStats::RequestHandled

The remaining methods return statistics that have been built up by analyzing the calls made to these three methods.

This class uses thread-safe interlocked functions to update its data members.

CRequestStats is used as a base class by CStdRequestStats and CPerfRequestStatObject.

Requirements

Header: atlisapi.h

See Also

Other Resources

CRequestStats Members

ATL Server Classes

CRequestStats Members

Methods

CRequestStats	The constructor.
GetActiveThreads	Call this method to get the number of threads currently handling a request.
GetAvgResponseTime	Call this method to get the average response time for handling a request.
GetCurrWaiting	Call this method to get the current number of queued requests.
GetFailedRequests	Call this method to get the total number of failed requests.
GetMaxWaiting	Call this method to get the maximum number of requests that have been queued at any one time since the ISAPI extension was loaded.
GetTotalRequests	Call this method to get the total number of handled requests.
OnRequestDequeued	Call this method when a request is removed from the queue.
OnRequestReceived	Call this method when a request is received and added to the queue.
RequestHandled	Call this method when a request has been handled.

Data Members

m_lActiveThreads	The number of threads currently handling a request.
m_lAvgResponseTime	The average response time for handling a request.
m_lCurrWaiting	The current number of queued requests.
m_lFailedRequests	The total number of failed requests.
m_liTotalResponseTime	The total time to handle all requests.
m_lMaxWaiting	The maximum number of queued requests.
m_lTotalRequests	The total number of handled requests.

See Also

Reference

CRequestStats Class

CRequestStats Methods

For information about the methods in **CRequestStats**, see CRequestStats Members.

CRequestStats::CRequestStats

The constructor.

```
CRequestStats( ) throw( );
```

Remarks

Initializes internal data structures.

See Also

Reference

CRequestStats Class

Other Resources

CRequestStats Members

CRequestStats::GetActiveThreads

Call this method to get the number of threads currently handling a request.

```
long GetActiveThreads( ) throw( );
```

Return Value

Returns the number of threads currently handling a request.

See Also

Reference

CRequestStats Class

Other Resources

CRequestStats Members

CRequestStats::GetAvgResponseTime

Call this method to get the average response time for handling a request.

```
long GetAvgResponseTime( ) throw( );
```

Return Value

Returns the average response time for handling a request.

Remarks

The average response time is measured as the total response time divided by the total number of requests handled. Information about currently queued requests is not included in this figure.

See Also

Reference

CRequestStats Class

Other Resources

CRequestStats Members

CRequestStats::GetCurrWaiting

Call this method to get the current number of queued requests.

```
long GetCurrWaiting( ) throw( );
```

Return Value

Returns the current number of queued requests.

Remarks

Queued requests are those that have been received and placed in the queue but are not yet being handled by one of the active threads.

See Also

Reference

CRequestStats Class

Other Resources

CRequestStats Members

CRequestStats::GetFailedRequests

Call this method to get the total number of failed requests.

```
long GetFailedRequests( ) throw( );
```

Return Value

Returns the total number of failed requests.

Remarks

A failed request is one where the argument used for the *bSuccess* parameter of RequestHandled is FALSE.

See Also

Reference

CRequestStats Class

Other Resources

CRequestStats Members

CRequestStats::GetMaxWaiting

Call this method to get the maximum number of requests that have been queued at any one time since the ISAPI extension was loaded.

```
long GetMaxWaiting( ) throw( );
```

Return Value

Returns the maximum number of queued requests.

See Also

Reference

CRequestStats Class

Other Resources

CRequestStats Members

CRequestStats::GetTotalRequests

Call this method to get the total number of handled requests.

```
long GetTotalRequests( ) throw( );
```

Return Value

Returns the total number of handled requests.

Remarks

Add the result of **GetTotalRequests** and CRequestStats::GetCurrWaiting to get the total number of requests received.

See Also

Reference

CRequestStats Class

Other Resources

CRequestStats Members

CRequestStats::OnRequestDequeued

Call this method when a request is removed from the queue.

```
void OnRequestDequeued( ) throw( );
```

See Also

Reference

CRequestStats Class
CRequestStats::OnRequestReceived
CRequestStats::RequestHandled

Other Resources

CRequestStats Members

CRequestStats::OnRequestReceived

Call this method when a request is received and added to the queue.

```
void OnRequestReceived( ) throw( );
```

See Also

Reference

CRequestStats Class
CRequestStats::OnRequestDequeued
CRequestStats::RequestHandled

Other Resources

CRequestStats Members

CRequestStats::RequestHandled

Call this method when a request has been handled.

```
void RequestHandled(
   AtlServerRequest* pRequestInfo,
   BOOL bSuccess
) throw( );
```

Parameters

pRequestInfo

Information about the request.

bSuccess

TRUE if the request was successfully handled, FALSE if an error occurred.

See Also

Reference

CRequestStats Class
CRequestStats::OnRequestReceived
CRequestStats::OnRequestDequeued

Other Resources

CRequestStats Members

CRequestStats Data Members

For information about the data members in **CRequestStats**, see CRequestStats Members

CRequestStats::m_lActiveThreads

The number of threads currently handling a request.

```
long m_lActiveThreads;
```

Remarks

Incremented in CRequestStats::OnRequestDequeued. Decremented in CRequestStats::RequestHandled.

See Also

Reference

CRequestStats Class

CRequestStats::GetActiveThreads

Other Resources

CRequestStats Members

CRequestStats::m_lAvgResponseTime

The average response time for handling a request.

```
long m_lAvgResponseTime;
```

Remarks

Calculated in CRequestStats::RequestHandled.

See Also

Reference

CRequestStats Class

CRequestStats::GetAvgResponseTime

Other Resources

CRequestStats Members

CRequestStats::m_lCurrWaiting

The current number of queued requests.

```
long m_lCurrWaiting;
```

Remarks

Incremented in CRequestStats::OnRequestReceived. Decremented in CRequestStats::OnRequestDequeued.

See Also

Reference

CRequestStats Class

CRequestStats::GetCurrWaiting

Other Resources

CRequestStats Members

CRequestStats::m_lFailedRequests

The total number of failed requests.

```
long m_lFailedRequests;
```

Remarks

Incremented in CRequestStats::RequestHandled when *bSuccess* is **FALSE**.

See Also

Reference

CRequestStats Class

CRequestStats::GetFailedRequests

Other Resources

CRequestStats Members

CRequestStats::m_liTotalResponseTime

The cumulative time taken by all handled requests.

```
__int64 m_liTotalResponseTime;
```

Remarks

This is the sum of the time taken from receipt of a request to the corresponding call to CRequestStats::RequestHandled. Time spent by requests sitting in the queue is not included in this total and, as a result, not included in CRequestStats::m_lAvgResponseTime.

See Also

Reference

CRequestStats Class

CRequestStats::GetAvgResponseTime

Other Resources

CRequestStats Members

CRequestStats::m_lMaxWaiting

The maximum number of queued requests.

```
long m_lMaxWaiting;
```

Remarks

Calculated in CRequestStats::OnRequestReceived.

See Also

Reference

CRequestStats Class

CRequestStats::GetMaxWaiting

Other Resources

CRequestStats Members

CRequestStats::m_lTotalRequests

The total number of handled requests.

```
long m_lTotalRequests;
```

Remarks

Incremented in CRequestStats::OnRequestReceived.

See Also

Reference

CRequestStats Class

CRequestStats::GetTotalRequests

Other Resources

CRequestStats Members

CSDLGenerator Class

This class provides a request handler responsible for generating a Web Service Description Language (WSDL) description of an XML Web service.

```
template <
   class THandler,
   const char* szHandlerName
>
class CSDLGenerator :
   public _CSDLGenerator,
   public IRequestHandlerImpl<CSDLGenerator>,
   public CComObjectRootEx<CComSingleThreadModel>
```

Parameters

THandler

A SOAP request handler class derived from CSoapRootHandler which implements the XML Web service that is to be described by the WSDL generated by this specialization of CSDLGenerator.

szHandlerName

The name of the XML Web service to be used in the WSDL generated by this specialization of **CSDLGenerator**.

Remarks

Typically, it is unnecessary to use this class directly. This class is used by the soap_handler attribute and the HANDLER_ENTRY_SDL macro.

Requirements

Header: atlsoap.h

See Also

Reference

IRequestHandlerImpl Class
CComObjectRootEx Class
ITagReplacerImpl Class

Other Resources

CSDLGenerator Members

CSDLGenerator Members

Methods

InitializeHandler	This implementation of IRequestHandler::InitializeHandler creates an instance of the class *THandler*, gets information about the SOAP methods and headers provided by that object, and then generates a WSDL response describing the XML Web service.
GetHandlerName	Call this method to obtain the name of the handler described by this specialization of **CSDLGenerator**.

Typedefs

_sdlGenerator	The type of the current specialization of this class template.

See Also

Reference

CSDLGenerator Class

CSDLGenerator Methods

For information about the methods in **CSDLGenerator**, see CSDLGenerator Members

CSDLGenerator::GetHandlerName

Call this method to obtain the name of the handler described by this specialization of **CSDLGenerator**.

```
const char * GetHandlerName( );
```

Return Value

Returns the template argument *szHandlerName* used to specialize this class template.

See Also

Reference

CSDLGenerator Class

Other Resources

CSDLGenerator Members

CSDLGenerator::InitializeHandler

This implementation of IRequestHandler::InitializeHandler creates an instance of the class *THandler*, gets information about the SOAP methods and headers provided by that object, and then generates a WSDL response describing the XML Web service.

```
HTTP_CODE InitializeHandler(
   AtlServerRequest * pRequestInfo,
   IServiceProvider * pServiceProvider
);
```

Parameters

pRequestInfo

Information about the current request.

pServiceProvider

The service provider.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

CSDLGenerator Class

Other Resources

CSDLGenerator Members

CSDLGenerator Typedefs

For information about the typedefs in **CSDLGenerator**, see CSDLGenerator Members

CSDLGenerator::_sdlGenerator

The type of the current specialization of this class template.

```
typedef CSDLGenerator<THandler, szHandlerName> _sdlGenerator;
```

See Also

Reference

CSDLGenerator Class

Other Resources

CSDLGenerator Members

CServerContext Class

This class provides an implementation of IHttpServerContext for use within an ISAPI extension DLL or a primary request handler.

```
class CServerContext :
   public CComObjectRootEx< CComMultiThreadModel >,
   public IHttpServerContext
```

Remarks

See IHttpServerContext for a description of the members in that interface.

Requirements

Header: atlisapi.h

Example

See the following samples:

- ForceLogin Sample
- MantaWeb Sample
- NotModified Sample
- ShowErrors Sample
- ShowFiles Sample
- ShowImage Sample
- ShowUser Sample

See Also

Reference

IHttpServerContext Interface

CComObjectRootEx Class

Other Resources

CServerContext Members

ATL Server Classes

CServerContext Members

Methods

AppendToLog	Implementation of IHttpServerContext::AppendToLog.
AsyncReadClient	Implementation of IHttpServerContext::AsyncReadClient.
AsyncWriteClient	Implementation of IHttpServerContext::AsyncWriteClient.
CServerContext	The constructor.
DoneWithSession	Implementation of IHttpServerContext::DoneWithSession.
GetAvailableBytes	Implementation of IHttpServerContext::GetAvailableBytes.
GetAvailableData	Implementation of IHttpServerContext::GetAvailableData.
GetContentType	Implementation of IHttpServerContext::GetContentType.
GetImpersonationToken	Implementation of IHttpServerContext::GetImpersonationToken.
GetPathInfo	Implementation of IHttpServerContext::GetPathInfo.
GetPathTranslated	Implementation of IHttpServerContext::GetPathTranslated.
GetQueryString	Implementation of IHttpServerContext::GetQueryString.
GetRequestMethod	Implementation of IHttpServerContext::GetRequestMethod.
GetScriptPathTranslated	Implementation of IHttpServerContext::GetScriptPathTranslated.
GetServerVariable	Implementation of IHttpServerContext::GetServerVariable.
GetTotalBytes	Implementation of IHttpServerContext::GetTotalBytes.
Initialize	This method is called by the ATL Server framework to initialize the object.
MapUrlToPathEx	Implementation of IHttpServerContext::MapUrlToPathEx.
ReadClient	Implementation of IHttpServerContext::ReadClient.
RequestIOCompletion	Implementation of IHttpServerContext::RequestIOCompletion.
SendRedirectResponse	Implementation of IHttpServerContext::SendRedirectResponse.
SendResponseHeader	Implementation of IHttpServerContext::SendResponseHeader.
TransmitFile	Implementation of IHttpServerContext::TransmitFile.
WriteClient	Implementation of IHttpServerContext::WriteClient.

See Also

Reference

CServerContext Class

Other Resources

IHttpServerContext Members

CComObjectRootEx Members

CServerContext Methods

For information about the methods in **CServerContext**, see CServerContext Members.

CServerContext::AppendToLog

Implementation of IHttpServerContext::AppendToLog.

```
BOOL AppendToLog(
   LPCSTR szMessage,
   DWORD* pdwLen
);
```

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::AsyncReadClient

Implementation of IHttpServerContext::AsyncReadClient.

```
BOOL AsyncReadClient(
   void* pvBuffer,
   DWORD* pdwSize
);
```

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::AsyncWriteClient

Implementation of IHttpServerContext::AsyncWriteClient.

```
BOOL AsyncWriteClient(
   void* pvBuffer,
   DWORD* pdwBytes
);
```

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::CServerContext

The constructor.

```
CServerContext( ) throw( );
```

Remarks

Initializes internal data structures.

See Also

Reference

CServerContext Class

CServerContext::Initialize

Other Resources

CServerContext Members

CServerContext::DoneWithSession

Implementation of IHttpServerContext::DoneWithSession.

```
BOOL DoneWithSession(
   DWORD dwHttpStatusCode
);
```

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::GetAvailableBytes

Implementation of IHttpServerContext::GetAvailableBytes.

```
DWORD GetAvailableBytes( );
```

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::GetAvailableData

Implementation of IHttpServerContext::GetAvailableData.

```
BYTE* GetAvailableData( );
```

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::GetContentType

Implementation of IHttpServerContext::GetContentType.

```
LPCSTR GetContentType( );
```

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::GetImpersonationToken

Implementation of IHttpServerContext::GetImpersonationToken.

```
BOOL GetImpersonationToken(
   HANDLE* pToken
);
```

Example

See the ShowUser Sample.

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::GetPathInfo

Implementation of IHttpServerContext::GetPathInfo.

```
LPCSTR GetPathInfo( );
```

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::GetPathTranslated

Implementation of IHttpServerContext::GetPathTranslated.

```
LPCSTR GetPathTranslated( );
```

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::GetQueryString

Implementation of IHttpServerContext::GetQueryString.

```
LPCSTR GetQueryString( );
```

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::GetRequestMethod

Implementation of IHttpServerContext::GetRequestMethod.

```
LPCSTR GetRequestMethod( );
```

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::GetScriptPathTranslated

Implementation of IHttpServerContext::GetScriptPathTranslated.

```
LPCSTR GetScriptPathTranslated( );
```

Example

See the ShowErrors Sample.

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::GetServerVariable

Implementation of IHttpServerContext::GetServerVariable.

```
BOOL GetServerVariable(
   LPCSTR pszVariableName,
   LPSTR pvBuffer,
   DWORD* pdwSize
);
```

Example

See the NotModified Sample and the ShowImage Sample.

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::GetTotalBytes

Implementation of IHttpServerContext::GetTotalBytes.

```
DWORD GetTotalBytes( );
```

Example

See the ForceLogin Sample.

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::Initialize

This method is called by the ATL Server framework to initialize the object.

```
void Initialize(
   EXTENSION_CONTROL_BLOCK* pECB
) throw( );
```

Parameters

pECB

The extension control block for the current request.

Remarks

Call this method to initialize the object before calling any of the IHttpServerContext methods.

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::MapUrlToPathEx

Implementation of IHttpServerContext::MapUrlToPathEx.

```
BOOL MapUrlToPathEx(
   LPCSTR szLogicalPath,
   DWORD dwLen,
   HSE_URL_MAPEX_INFO* pumInfo
);
```

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::ReadClient

Implementation of IHttpServerContext::ReadClient.

```
BOOL ReadClient(
   void* pvBuffer,
   DWORD* pdwSize
);
```

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::RequestIOCompletion

Implementation of IHttpServerContext::RequestIOCompletion.

```
BOOL RequestIOCompletion(
   PFN_HSE_IO_COMPLETION pfn,
   DWORD* pdwContext
);
```

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::SendRedirectResponse

Implementation of IHttpServerContext::SendRedirectResponse.

```
BOOL SendRedirectResponse(
   LPCSTR pszRedirectUrl
);
```

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::SendResponseHeader

Implementation of IHttpServerContext::SendResponseHeader.

```
BOOL SendResponseHeader(
   LPCSTR pszHeader = "Content-Type: text/html\r\n\r\n",
   LPCSTR pszStatusCode = "200 OK",
   BOOL fKeepConn = FALSE
);
```

Example

See the ShowErrors Sample.

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::TransmitFile

Implementation of IHttpServerContext::TransmitFile.

```
BOOL TransmitFile(
   HANDLE hFile,
   PFN_HSE_IO_COMPLETION pfn,
   void* pContext,
   LPCSTR szStatusCode,
   DWORD dwBytesToWrite,
   DWORD dwOffset,
   void* pvHead,
   DWORD dwHeadLen,
   void* pvTail,
   DWORD dwTailLen,
   DWORD dwFlags
);
```

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CServerContext::WriteClient

Implementation of IHttpServerContext::WriteClient.

```
BOOL WriteClient(
   void* pvBuffer,
   DWORD* pdwBytes
);
```

Example

See the MantaWeb Sample and the ShowErrors Sample.

See Also

Reference

CServerContext Class

Other Resources

CServerContext Members

CSessionCookie Class

This class represents a cookie that can be used to identify a user's session.

```
class CSessionCookie :
   public CCookie
```

Remarks

The cookie represented by this class has a particular name (SESSION_COOKIE_NAME) and applies to the whole web site (has a path of /). The session ID for the user can be set by calling CSessionCookie::SetSessionID and is stored as the value of the cookie.

Requirements

Header: atlisapi.h

Example

See the Cookies Sample.

See Also

Reference

CCookie Class

Other Resources

CSessionCookie Members

Session-State Services (ATL Server)

CSessionCookie Members

Methods

CSessionCookie	The constructor.
SetSessionID	Call this method to set the session ID of the cookie.

See Also

Reference

CSessionCookie Class

CSessionCookie Methods

For information about the methods in **CSessionCookie**, see CSessionCookie Members.

CSessionCookie::CSessionCookie

The constructor.

```
CSessionCookie( ) throw(...);
CSessionCookie(
   LPCSTR szSessionID
) throw(...);
```

Parameters

szSessionID

The session ID used to identify a particular user of a web site.

Example

See the Cookies Sample.

See Also

Reference

CSessionCookie Class

Other Resources

CSessionCookie Members

CSessionCookie::SetSessionID

Call this method to set the session ID of the cookie.

```
BOOL SetSessionID(
   LPCSTR szSessionID
) throw( );
```

Parameters

szSessionID

The session ID used to identify a particular user of a web site.

See Also

Reference

CSessionCookie Class

Other Resources

CSessionCookie Members

CSessionNameGenerator Class

Use this class to generate random session IDs.

```
class CSessionNameGenerator :
    public CCryptProv
```

Remarks

This class tries to use the Crypto API to generate random bytes for the session key name. If the Crypto API isn't available, rand is used to generate the random bytes. ATL Server's memory and database-backed session-state services both use this class to generate session names.

Call CSessionNameGenerator::GetNewSessionName to generate a random session ID that can be passed to a web client in a cookie or used in a URL.

Requirements

Header: atlsession.h

See Also

Reference

CCryptProv Class

Other Resources

Session-State Services (ATL Server)

CSessionNameGenerator Members

ATL Server Classes

Session-State Reference

CSessionNameGenerator Members

Methods

CalcMaxInputSize	Call this method to get the number of bytes of random data that, when base-64 encoded, will fit in a buffer of the specified size.
CSessionNameGenerator	The constructor for the session name generator.
GenerateRandomName	Call this method to generate a random name.
GetNewSessionName	Call this method to generate a new, random, base-64-encoded session ID.

Data Members

m_bCryptNotAvailable	Indicates whether or not the Crypto API can be used to generate the session IDs.

Enums

MIN_SESSION_KEY_LEN	Defines the minimum allowed size in bytes for a session name generated by CSessionNameGenerator.

See Also

Reference

CSessionNameGenerator Class

Other Resources

Session-State Services (ATL Server)

Session-State Reference

CSessionNameGenerator Methods

For information about the methods in **CSessionNameGenerator**, see CSessionNameGenerator Members.

CSessionNameGenerator::CalcMaxInputSize

Call this method to get the number of bytes of random data that, when base-64 encoded, will fit in a buffer of the specified size.

```
DWORD CalcMaxInputSize(
   DWORD nOutputSize
) throw( );
```

Parameters

nOutputSize

The size of the buffer that will be used to hold the base-64-encoded session ID.

Return Value

The maximum raw session name size that will not exceed *nOutputSize* once base-64 encoded.

Remarks

This function is used internally by the **CSessionNameGenerator** class, and is not typically necessary to use explicitly.

See Also

Reference

CSessionNameGenerator Class

Other Resources

Session-State Services (ATL Server)

CSessionNameGenerator Members

Session-State Reference

CSessionNameGenerator::CSessionNameGenerator

The constructor for the session name generator.

```
CSessionNameGenerator( ) throw( );
```

Remarks

Initializes internal data structures.

See Also

Reference

CSessionNameGenerator Class

Other Resources

Session-State Services (ATL Server)

CSessionNameGenerator Members

Session-State Reference

CSessionNameGenerator::GenerateRandomName

Call this method to generate a random name.

```
HRESULT GenerateRandomName(
   BYTE* pBuff,
   DWORD dwBuffSize
) throw( );
```

Parameters

pBuff

The buffer to hold the raw binary data.

dwBuffSize

The size of the buffer in bytes.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The resulting name is generated using the Crypto API or rand if the Crypto API isn't available. Note that the data returned by this method is neither base-64 encoded nor null-terminated.

See Also

Reference

CSessionNameGenerator Class

CSessionNameGenerator::GetNewSessionName

Other Resources

Session-State Services (ATL Server)

CSessionNameGenerator Members

Session-State Reference

CSessionNameGenerator::GetNewSessionName

Call this method to generate a new, random, base-64-encoded session ID.

```
HRESULT GetNewSessionName(
   LPSTR szNewID,
   DWORD* pdwSize
) throw( );
```

Parameters

szNewID

A caller-allocated buffer that, on success, will receive the newly generated session ID.

pdwSize

On entry, the address of a **DWORD** containing the size of the buffer in bytes. On exit, the number of bytes transferred into the buffer (including the null-terminating byte).

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

This method generates a random, base-64-encoded, session ID. The buffer provided to **GetNewSessionName** must be greater than or equal to CSessionNameGenerator::MIN_SESSION_KEY_LEN bytes (to ensure that the base-64-encoding algorithm works correctly) and less than or equal to MAX_SESSION_KEY_LEN (a limit imposed for database-backed session-state implementations for reasons of efficiency).

See Also

Reference

CSessionNameGenerator Class
CSessionNameGenerator::GenerateRandomName

Other Resources

Session-State Services (ATL Server)
CSessionNameGenerator Members
Session-State Reference

CSessionNameGenerator Data Members

For information about the data members in **CSessionNameGenerator**, see CSessionNameGenerator Members.

CSessionNameGenerator::m_bCryptNotAvailable

Indicates whether or not the Crypto API can be used to generate the session IDs.

```
bool m_bCryptNotAvailable;
```

Remarks

If this member is set to false (after construction), the Crypto API is not available and rand will be used to generate new session IDs.

See Also

Reference

CSessionNameGenerator Class
rand

Other Resources

Session-State Services (ATL Server)
CSessionNameGenerator Members
Session-State Reference

CSessionNameGenerator Enums

For information about the enums in **CSessionNameGenerator**, see CSessionNameGenerator Members.

CSessionNameGenerator::MIN_SESSION_KEY_LEN

Defines the minimum allowed size in bytes for a session name generated by CSessionNameGenerator.

```
enum {MIN_SESSION_KEY_LEN = 5};
```

Remarks

This limit is imposed to ensure that the session name can be correctly base-64-encoded.

See Also

Other Resources

Session-State Services (ATL Server)
ATL Server Concepts
ATL Server Reference
ATL Server Macros
Session-State Reference

CSessionStateService Class

Provides support for the maintenance and retrieval of session-state data.

```
template <
   class MonitorClass,
   class TServiceImplClass
>
class CSessionStateService :
   public ISessionStateService,
   public ISessionStateControl,
   public IWorkerThreadClient,
   public CComObjectRootEx<CComGlobalsThreadModel>
```

Parameters

MonitorClass

The class to be used for monitoring and retiring expired sessions (typically the CWorkerThread class).

TServiceImplClass

The session-state support implementation class, which determines the method used for session-state (typically either CMemSessionServiceImpl or CDBSessionServiceImplT).

Remarks

The **CSessionStateService** class provides session-state support by implementing the ISessionStateService and ISessionStateControl interfaces. The **CSessionStateService** class delegates the work of persisting session data to the class specified in the *TServiceImplClass* template parameter.

Present only in session-state enabled projects, **CSessionStateService** typically resides in the ISAPI extension DLL and is accessible to request handlers objects via the ISessionStateService interface. **CSessionStateService** is usually configured automatically by the ATL Server Project Wizard, and explicit access is not normally required. See Providing Session-State Services and Enabling Session-State Support for more information.

Requirements

Header: atlsession.h

See Also

Tasks

OnlineAddressBook Sample: Stores an Address Book in an XML Web Service
SessionSettings Sample: Session State Demonstration

Reference

ShowSession
ISessionStateService Interface
ISessionStateControl Interface
IWorkerThreadClient Interface
CComObjectRootEx Class

Other Resources

Session-State Services (ATL Server)
CSessionStateService Members
ATL Server Classes
Session-State Reference

CSessionStateService Members

Methods

CloseSession	Call this method to close a session, releasing session variables and storage resources.
CreateNewSession	Call this method to create a new session and corresponding session ID.
CreateNewSessionByName	Call this method to create a new session, given a session name.
CSessionStateService	This method constructs an instance of the session-state services class.
GetSession	Call this method to retrieve an ISession interface pointer that represents an existing session.
GetSessionCount	Call this method to retrieve the number of active sessions.
GetSessionTimeout	Call this method to get the timeout of a session in milliseconds.
Initialize	Call this method to initialize the session-state service.
SetSessionTimeout	Call this method to set the timeout value for all current and future sessions, in milliseconds.
Shutdown	Call this method to release session data and terminate session maintenance.

See Also

Reference

CSessionStateService Class

Other Resources

Session-State Services (ATL Server)
ISessionStateService Members
ISessionStateControl Members
IWorkerThreadClient Members
CComObjectRootEx Members
Session-State Reference

CSessionStateService Methods

For information about the methods in **CSessionStateService**, see CSessionStateService Members.

CSessionStateService::CloseSession

Call this method to close a session, releasing session variables and storage resources.

```
STDMETHOD(CloseSession)(
   LPCSTR szSessionID
) throw( );
```

Remarks

See ISessionStateService::CloseSession.

Example

See the OnlineAddressBook Sample and the SessionSettings Sample.

See Also

Reference

CSessionStateService Class

Other Resources

Session-State Services (ATL Server)

CSessionStateService Members

Session-State Reference

CSessionStateService::CreateNewSession

Call this method to create a new session and corresponding session ID.

```
STDMETHOD(CreateNewSession)(
   LPSTR szNewID,
   DWORD* pdwSize,
   ISession** ppSession
) throw( );
```

Remarks

See ISessionStateService::CreateNewSession.

Example

See the OnlineAddressBook Sample, the SessionSettings Sample, and the ShowSession Sample.

See Also

Reference

CSessionStateService Class

Other Resources

Session-State Services (ATL Server)

CSessionStateService Members

Session-State Reference

CSessionStateService::CreateNewSessionByName

Call this method to create a new session, given a session name.

```
STDMETHOD(CreateNewSessionByName)(
   LPSTR szNewID,
   ISession** ppSession
) throw( );
```

Remarks

See ISessionStateService::CreateNewSessionByName.

See Also

Reference

CSessionStateService Class

Other Resources

Session-State Services (ATL Server)

CSessionStateService Members

Session-State Reference

CSessionStateService::CSessionStateService

This method constructs an instance of the session-state services class.

```
CSessionStateService( ) throw( );
```

Remarks

Initializes internal data structures.

See Also

Reference

CSessionStateService Class

Other Resources

Session-State Services (ATL Server)

CSessionStateService Members

Session-State Reference

CSessionStateService::GetSession

Call this method to retrieve an ISession interface pointer that represents an existing session.

```
STDMETHOD(GetSession)(
   LPCSTR szID,
   ISession** ppSession
) throw( );
```

Remarks

See ISessionStateService::GetSession.

Example

See the OnlineAddressBook Sample, the SessionSettings Sample, and the ShowSession Sample.

See Also

Reference

CSessionStateService Class

Other Resources

Session-State Services (ATL Server)

CSessionStateService Members

Session-State Reference

CSessionStateService::GetSessionCount

Call this method to retrieve the number of active sessions.

```
STDMETHOD(GetSessionCount)(
   DWORD* pnSessionCount
) throw( );
```

Remarks

See ISessionStateControl::GetSessionCount.

Example

See the ShowSession Sample.

See Also

Reference

CSessionStateService Class

Other Resources

Session-State Services (ATL Server)

CSessionStateService Members

Session-State Reference

CSessionStateService::GetSessionTimeout

Call this method to get the timeout of a session in milliseconds.

```
STDMETHOD(GetSessionTimeout)(
   unsigned __int64 * pnTimeout
) throw( );
```

Remarks

See ISessionStateControl::GetSessionTimeout.

See Also

Reference

CSessionStateService Class
CSessionStateService::SetSessionTimeout

Other Resources

Session-State Services (ATL Server)
CSessionStateService Members
Session-State Reference

CSessionStateService::Initialize

Call this method to initialize the session-state service.

```
template < class ThreadTraits >
HRESULT Initialize(
   CWorkerThread< ThreadTraits > * pWorker,
   IServiceProvider * pServiceProvider = NULL,
   TServiceImplClass::SERVICEIMPL_INITPARAM_TYPE pInitData = NULL,
   unsigned __int64 dwTimeout = ATL_SESSION_TIMEOUT
) throw( );
HRESULT Initialize(
   IServiceProvider * pServiceProvider = NULL,
   TServiceImplClass::SERVICEIMPL_INITPARAM_TYPE pInitData = NULL,
   unsigned __int64 dwTimeout = ATL_SESSION_TIMEOUT
) throw( );
```

Parameters

pWorker

The address of the thread to be used for session maintenance, usually **CWorkerThreadWrapper<>**. If zero (0), no session maintenance will be performed. This is not recommended, especially when using the memory-backed session-state implementation.

pServiceProvider

The address of an optional service provider instance. The specific session implementation determines whether a service provider is necessary and the degree to which it is used.

pInitData

The address of implementation-specific data to be passed to the ***TServiceImplClass*::Initialize** function.

dwTimeout

The time interval after which unaccessed sessions expire.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Call this method to initialize session-state support. Although the **CSessionStateService** class is implementation-independent (it doesn't matter whether you're using database or memory-backed session-state support), the **Initialize** method requires some implementation-specific data, particularly when using database-backed session-state support. See Using Session-State Support and Enabling Session-State Support for specifics.

See Also

Reference

CSessionStateService Class

Other Resources

Session-State Services (ATL Server)

CSessionStateService Members

Session-State Reference

CSessionStateService::SetSessionTimeout

Call this method to set the timeout value for all current and future sessions, in milliseconds.

```
STDMETHOD(SetSessionTimeout)(
   unsigned __int64 nTimeout
) throw();
```

Remarks

See ISessionStateControl::SetSessionTimeout.

See Also

Reference

CSessionStateService Class
ISessionStateControl::GetSessionTimeout

Other Resources

Session-State Services (ATL Server)
CSessionStateService Members
Session-State Reference

CSessionStateService::Shutdown

Call this method to release session data and terminate session maintenance.

```
void Shutdown( ) throw( );
```

Remarks

Closes all session-state services. This method should only be called by the ISAPI extension DLL that owns the **CSessionStateService** object immediately before DLL termination.

See Also

Reference

CSessionStateService Class

Other Resources

Session-State Services (ATL Server)

CSessionStateService Members

Session-State Reference

CSharedCache Class

This class implements the ISharedBlobCache interface using CBlobCache.

```
class CSharedCache :
   public CBlobCache< CWorkerThread< >, CStdStatClass >,
   public IMemoryCacheClient,
   public ISharedBlobCache
```

Remarks

This class adds items to the cache so that they expire after the period specified by ATL_SHAREDBLOBCACHE_TIMEOUT.

Requirements

Header: atlsharedsvc.h

Example

See the BlobCache Sample.

See Also

Reference

CBlobCache Class
IMemoryCacheClient Interface
ISharedBlobCache Interface
CWorkerThread Class
CStdStatClass Class

Other Resources

CSharedCache Members

CSharedCache Members

Methods

AddItem	Implementation of ISharedBlobCache::AddItem.
AddRef	Implementation of IUnknown::AddRef.
Free	Implementation of IMemoryCacheClient::Free.
GetItem	Implementation of ISharedBlobCache::GetItem.
QueryInterface	Implementation of IUnknown::QueryInterface.
Release	Implementation of IUnknown::Release.

See Also

Reference

CSharedCache Class

CSharedCache Methods

For information about the methods in **CSharedCache**, see CSharedCache Members

CSharedCache::AddItem

Implementation of ISharedBlobCache::AddItem.

```
STDMETHODIMP AddItem(
   BSTR szItemName,
   BSTR szData
);
```

Remarks

This class adds items to the cache so that they expire after the period specified by ATL_SHAREDBLOBCACHE_TIMEOUT.

See Also

Reference

CSharedCache Class

Other Resources

CSharedCache Members

CSharedCache::AddRef

Implementation of IUnknown::AddRef.

```
ULONG STDMETHODCALLTYPE AddRef( );
```

Return Value

Always returns 1.

Remarks

This class does not implement lifetime control using reference counting.

See Also

Reference

CSharedCache Class

Other Resources

CSharedCache Members

CSharedCache::Free

Implementation of IMemoryCacheClient::Free.

```
STDMETHOD(Free)(
   const void * pvData
);
```

Example

See the BlobCache Sample.

See Also

Reference

CSharedCache Class

Other Resources

CSharedCache Members

CSharedCache::GetItem

Implementation of ISharedBlobCache::GetItem.

```
STDMETHODIMP GetItem(
   BSTR szItemName,
   BSTR * szData
);
```

See Also

Reference

CSharedCache Class

Other Resources

CSharedCache Members

CSharedCache::QueryInterface

Implementation of IUnknown::QueryInterface.

```
STDMETHODIMP QueryInterface(
   REFIID riid,
   void ** ppv
);
```

Remarks

Objects of this class can be successfully queried for the IUnknown, IMemoryCacheClient and ISharedBlobCache interfaces.

See Also

Reference

CSharedCache Class

Other Resources

CSharedCache Members

CSharedCache::Release

Implementation of IUnknown::Release.

```
ULONG STDMETHODCALLTYPE Release( );
```

Return Value

Always returns 1.

Remarks

This class does not implement lifetime control using reference counting.

See Also

Reference

CSharedCache Class

Other Resources

CSharedCache Members

CSharedCacheHandler Class

This class implements a XML Web service that allows access to a cache that can be shared by SOAP clients.

```
[
   coclass,
   soap_handler(
      name="SharedBlobCache",
      namespace =
      "http://www.microsoft.com/vc/atlserver/soap/SharedBlobCache",
   protocol = "soap"
   ),
   uuid("C518C876-7685-4050-9E01-898271C05F88")
]
class CSharedCacheHandler :
   public CSoapHandler<CSharedCacheHandler>,
   public ISharedBlobCache
```

Remarks

The implementation of ISharedBlobCache provided by this class delegates to the **ISharedBlobCache** pointer obtained from the service provider passed to HandleRequest. (Typically, this is the service provider implemented by CIsapiExtension or a derived class.)

To expose this XML Web service from your ISAPI extension, include atlsharedsvc.h in your project, add a CSharedCache member to your ISAPI extension class, and change the implementation of IServiceProvider::QueryService to return the ISharedBlobCache interface of the **CSharedCache** member.

Requirements

Header: atlsharedsvc.h

See Also

Reference

CSoapHandler Class
ISharedBlobCache Interface
Using Intrinsic Services

Other Resources

CSharedCacheHandler Members

CSharedCacheHandler Members

Methods

AddItem	Implementation of ISharedBlobCache::AddItem.
GetItem	Implementation of ISharedBlobCache::GetItem.
HandleRequest	This method calls CSharedCacheHandler::Initialize and handles requests sent to the cache.
Initialize	This method queries the service provider for the ISharedBlobCache service.

See Also

Reference

CSharedCacheHandler Class

CSharedCacheHandler Methods

For information about the methods in **CSharedCacheHandler**, see CSharedCacheHandler Members.

CSharedCacheHandler::AddItem

Implementation of ISharedBlobCache::AddItem.

```
[soap_method]
STDMETHOD(AddItem )(
   BSTR szItemName,
   BSTR szData
);
```

Remarks

Delegates to the ISharedBlobCache obtained from Initialize.

See Also

Reference

CSharedCacheHandler Class

Other Resources

CSharedCacheHandler Members

CSharedCacheHandler::GetItem

Implementation of ISharedBlobCache::GetItem.

```
[soap_method]
STDMETHOD(GetItem)(
   BSTR szItemName,
   BSTR* szData
);
```

Remarks

Delegates to the ISharedBlobCache obtained from Initialize.

See Also

Reference

CSharedCacheHandler Class

Other Resources

CSharedCacheHandler Members

CSharedCacheHandler::HandleRequest

This method calls CSharedCacheHandler::Initialize and handles requests sent to the cache.

```
DWORD HandleRequest(
   AtlServerRequest* pRequestInfo,
   IServiceProvider* pProvider
);
```

Parameters

pRequestInfo

The information about the request to be handled.

pProvider

The service provider.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

CSharedCacheHandler Class

AtlServerRequest Structure

Other Resources

CSharedCacheHandler Members

CSharedCacheHandler::Initialize

This method queries the service provider for the ISharedBlobCache service.

```
DWORD Initialize(
   IServiceProvider * pProvider
);
```

Parameters

pProvider

The service provider.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

The implementation of ISharedBlobCache provided by this XML Web service is delegated to the same interface obtained from the service provider.

See Also

Reference

CSharedCacheHandler Class

Other Resources

CSharedCacheHandler Members

CSimpleStack Class

This class represents a simple stack whose elements are stored in a fixed size array.

```
template <
   typename TData,
   int nMax = 64
>
class CSimpleStack
```

Parameters

TData

The type of data item to be stored in the stack.

nMax

The maximum number of items to store in the stack.

Remarks

This class creates an array of *TData* items with enough space to store *nMax* items and uses simple indexing to track which item is at the top of the stack. The items are copied using the assignment operator when added to or removed from the stack.

Requirements

Header: atlhtml.h

See Also

Other Resources

CSimpleStack Members

CSimpleStack Members

Methods

CSimpleStack	The constructor.
IsEmpty	Call this method to check whether the stack is empty.
Pop	Call this method to remove the object at the top of the stack and retrieve it.
Push	Call this method to copy the specified object onto the top of the stack.

Data Members

m_Data	The array of stack items.
m_nTop	The integer representing the index of the item at the top of the stack.

See Also

Reference

CSimpleStack Class

CSimpleStack Methods

For information about the methods in **CSimpleStack**, see CSimpleStack Members.

CSimpleStack::CSimpleStack

The constructor.

```
CSimpleStack( );
```

Remarks

Initializes m_nTop to -1.

See Also

Reference

CSimpleStack Class

Other Resources

CSimpleStack Members

CSimpleStack::IsEmpty

Call this method to check whether the stack is empty.

```
bool IsEmpty( );
```

Return Value

Returns **true** if the stack is empty, **false** otherwise.

See Also

Reference

CSimpleStack Class

Other Resources

CSimpleStack Members

CSimpleStack::Pop

Call this method to remove the object at the top of the stack and retrieve it.

```
bool Pop(
   TData * pData
);
```

Parameters

pData

A pointer to a variable that will receive a copy of the item removed from the top of the stack.

Return Value

Returns **true** on success, **false** on failure or if there are no items currently on the stack.

See Also

Reference

CSimpleStack Class

Other Resources

CSimpleStack Members

CSimpleStack::Push

Call this method to copy the specified object onto the top of the stack.

```
bool Push(
   const TData * pData
);
```

Parameters

pData

The item to push onto the stack.

Return Value

Returns **true** on success, **false** on failure or if there is an attempt to push more than the maximum number of items on the stack.

See Also

Reference

CSimpleStack Class

Other Resources

CSimpleStack Members

CSimpleStack Data Members

For information about the data members in **CSimpleStack**, see CSimpleStack Members.

CSimpleStack::m_Data

The array of stack items.

```
TData m_Data[nMax];
```

See Also

Reference

CSimpleStack Class

Other Resources

CSimpleStack Members

CSimpleStack::m_nTop

The integer representing the index of the item at the top of the stack.

```
int m_nTop;
```

See Also

Reference

CSimpleStack Class

Other Resources

CSimpleStack Members

CSMTPConnection Class

This class represents a connection to an SMTP service.

```
class CSMTPConnection
```

Requirements

Header: atlsmtpconnection.h

See Also

Tasks

MantaWeb Sample: Online Personal Information Management

Mailer Sample: Demonstrates SMTP Mail Classes

Reference

ConfirmUser

ConfirmUser

Other Resources

CSMTPConnection Members

ATL Server Classes

CSMTPConnection Members

Methods

Connect	Call this method to attempt to connect to the specified SMTP service.
Connected	Call this method to check whether a connection with an SMTP service has been established.
CSMTPConnection	The constructor.
~CSMTPConnection	The destructor.
Disconnect	Call this method to disconnect from an SMTP service.
SendMessage	Call this method to send a message over the SMTP connection.
SendRaw	Call this method to send raw data over the SMTP connection.
SendSimple	Call this method to send a message with the specified recipients, sender, subject line, and body text over a n SMTP connection.
WriteToFile	Call this method to write a MIME-encoded message to a file.

See Also

Reference

CSMTPConnection Class

CSMTPConnection Methods

For information about the methods in **CSMTPConnection**, see CSMTPConnection Members.

CSMTPConnection::Connect

Call this method to attempt to connect to the specified SMTP service.

```
BOOL Connect(
   LPCTSTR lpszHostName,
   DWORD dwTimeout = 10000
) throw();
```

Parameters

lpszHostName

The name of the server to try to connect to.

dwTimeout

The timeout in milliseconds.

Return Value

Returns TRUE on success, FALSE on failure.

Example

See the MantaWeb Sample, the Mailer Sample, and the ConfirmUser Sample.

See Also

Reference

CSMTPConnection Class

Other Resources

CSMTPConnection Members

CSMTPConnection::Connected

Call this method to check whether a connection with an SMTP service has been established.

```
inline BOOL Connected( ) throw( );
```

Return Value

Returns TRUE if connected, FALSE if not.

See Also

Reference

CSMTPConnection Class

Other Resources

CSMTPConnection Members

CSMTPConnection::CSMTPConnection

The constructor.

```
CSMTPConnection( ) throw( );
```

See Also

Reference

CSMTPConnection Class

Other Resources

CSMTPConnection Members

CSMTPConnection::~CSMTPConnection

The destructor.

```
~CSMTPConnection( ) throw( );
```

See Also

Reference

CSMTPConnection Class

Other Resources

CSMTPConnection Members

CSMTPConnection::Disconnect

Call this method to disconnect from an SMTP service.

```
inline BOOL Disconnect( ) throw( );
```

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CSMTPConnection Class

Other Resources

CSMTPConnection Members

CSMTPConnection::SendMessage

Call this method to send a message over the SMTP connection.

```
inline BOOL SendMessage(
   CMimeMessage& msg,
   LPCTSTR lpszRecipients = NULL,
   LPCTSTR lpszSender = NULL
) throw( );
BOOL SendMessage(
   LPCTSTR lpszFileName,
   LPCTSTR lpszRecipients = NULL,
   LPCTSTR lpszSender = NULL
) throw( );
```

Parameters

msg

The message to send.

lpszRecipients

The recipients to send to. If not specified, the recipients specified in the message will be used.

lpszSender

The message sender. If not specified, the sender specified in the message will be used.

lpszFileName

The file containing the message.

Return Value

Returns TRUE on success, FALSE on failure.

Example

See the MantaWeb Sample, the Mailer Sample, and the ConfirmUser Sample.

See Also

Reference

CSMTPConnection Class

Other Resources

CSMTPConnection Members

CSMTPConnection::SendRaw

Call this method to send raw data over the SMTP connection.

```
inline BOOL SendRaw(
   LPCTSTR lpszRawData,
   DWORD dwLen,
   LPCTSTR lpszRecipients,
   LPCTSTR lpszSender
) throw( );
```

Parameters

lpszRawData

The raw data.

dwLen

The number of bytes in the data.

lpszRecipients

The list of recipients.

lpszSender

The sender.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CSMTPConnection Class

Other Resources

CSMTPConnection Members

CSMTPConnection::SendSimple

Call this method to send a message with the specified recipients, sender, subject line, and body text over an SMTP connection.

```
inline BOOL SendSimple(
   LPCTSTR lpszRecipients,
   LPCTSTR lpszSender,
   LPCTSTR lpszSubject,
   LPCTSTR lpszBody,
   int nTextLen = - 1
) throw( );
```

Parameters

lpszRecipients

The message recipients.

lpszSender

The message sender.

lpszSubject

The message subject.

lpszBody

The text of the message body.

nTextLen

The length of the message body text in characters. If this value is less than 0, the method will determine the actual length of the text.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CSMTPConnection Class

Other Resources

CSMTPConnection Members

CSMTPConnection::WriteToFile

Call this method to write a MIME-encoded message to a file.

```
inline BOOL WriteToFile(
   LPCTSTR lpszFileName,
   CMimeMessage& msg,
   LPCTSTR lpszRecipients = NULL,
   LPCTSTR lpszSender = NULL,
   DWORD dwFlags = 0
) throw( );
```

Parameters

lpszFileName

The file to write to.

msg

The MIME-encoded message.

lpszRecipients

The message recipients.

lpszSender

The message sender.

dwFlags

The flags. Include ATLSMTP_DUMP_SENDER to write sender information to the file and ATLSMTP_DUMP_RECIPS to write recipient information to the file.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CSMTPConnection Class

Other Resources

CSMTPConnection Members

CSMTPWSAStartup Class

This class allows you to initialize a WSADATA data structure that contains details of a Windows Sockets implementation.

```
struct CSMTPWSAStartup;
```

Requirements

Header: atlsmtpconnection.h

See Also

Other Resources

CSMTPWSAStartup Members

ATL Server Classes

CSMTPWSAStartup Members

Methods

CSMTPWSAStartup	The constructor for a **CSMTPWSAStartup** object.
~CSMTPWSAStartup	The destructor for a **CSMTPWSAStartup** object.
Init	Call this method to initialize a **CSMTPWSAStartup** object.
Uninit	Call this method to uninitialize a **CSMTPWSAStartup** object.

See Also

Reference

CSMTPWSAStartup Class

CSMTPWSAStartup Methods

For information about the methods in **CSMTPWSAStartup**, see CSMTPWSAStartup Members.

CSMTPWSAStartup::CSMTPWSAStartup

The constructor for a **CSMTPWSAStartup** object.

```
CSMTPWSAStartup( ) throw( );
```

See Also

Reference

CSMTPWSAStartup Class

Other Resources

CSMTPWSAStartup Members

CSMTPWSAStartup::~CSMTPWSAStartup

The destructor for a **CSMTPWSAStartup** object.

```
~CSMTPWSAStartup( ) throw( );
```

See Also

Reference

CSMTPWSAStartup Class

Other Resources

CSMTPWSAStartup Members

CSMTPWSAStartup::Init

Call this method to initialize a **CSMTPWSAStartup** object.

```
bool Init( ) throw( );
```

Return Value

Returns true on success, false on failure.

Remarks

For more information, see WSAStartup in the *Platform SDK*.

See Also

Reference

CSMTPWSAStartup Class

Other Resources

CSMTPWSAStartup Members

CSMTPWSAStartup::Uninit

Call this method to uninitialize a **CSMTPWSAStartup** object.

```
bool Uninit( ) throw( );
```

Return Value

Returns true on success, false on failure.

Remarks

For more information, see WSACleanup in the *Platform SDK*.

See Also

Reference

CSMTPWSAStartup Class

Other Resources

CSMTPWSAStartup Members

CSoapFault Class

This class represents a SOAP Fault element, which is used to carry error and/or status information.

```
class CSoapFault
```

Remarks

This class can be used to parse a string containing a SOAP Fault and allow access to its elements as data members of this class or to generate a string containing a SOAP Fault from the values of the data members.

Note that not all members of this class need to have values when generating a SOAP Fault or after parsing. See the SOAP specification (http://www.w3.org/TR/SOAP) for more details.

Requirements

Header: atlsoap.h

See Also

Other Resources

CSoapFault Members

CSoapFault Members

Methods

Clear	Call this method to clear the data members.
CSoapFault	The constructor.
GenerateFault	Call this method to write a SOAP Fault element to a stream.
ParseFault	Call this method to parse a SOAP Fault element from a stream.

Data Members

m_soapErrCode	A constant identifying the fault.
m_strDetail	Application-specific error information related to the SOAP Body element.
m_strFaultActor	A URI identifying the source of the fault.
m_strFaultCode	A string identifying the fault.
m_strFaultString	A human readable explanation of the fault.

See Also

Reference

CSoapFault Class

CSoapFault Methods

For information about the methods in **CSoapFault**, see CSoapFault Members

CSoapFault::Clear

Call this method to clear the data members.

```
void Clear( );
```

Remarks

Empties the string data members and sets CSoapFault::m_soapErrCode to SOAP_E_UNK.

See Also

Reference

CSoapFault Class

Other Resources

CSoapFault Members

CSoapFault::CSoapFault

The constructor.

```
CSoapFault( );
```

Remarks

Initializes CSoapFault::m_soapErrCode to SOAP_E_UNK.

See Also

Reference

CSoapFault Class

Other Resources

CSoapFault Members

CSoapFault::GenerateFault

Call this method to write a SOAP Fault element to a stream.

```
HRESULT GenerateFault(
   IWriteStream* pWriteStream
);
```

Parameters

pWriteStream

The stream to which the SOAP Fault element should be written.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Uses the values of the data members to generate the XML for the Fault element and write it to the stream.

If either of the m_strFaultString or m_strFaultCode members is empty, standard values are provided for the faultstring and faultcode elements based on the m_soapErrCode member.

See Also

Reference

CSoapFault Class

IWriteStream Interface

Other Resources

CSoapFault Members

CSoapFault::ParseFault

Call this method to parse a SOAP Fault element from a stream.

```
HRESULT ParseFault(
   IStream* pStream,
   ISAXXMLReader* pReader = NULL
);
```

Parameters

pStream

The stream containing the XML for the SOAP Fault element.

pReader

The XML reader. If NULL, a **SAXXMLReader** object is created.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Parses the SOAP Fault and populates the data members with the values of its elements.

See Also

Reference

CSoapFault Class

Other Resources

CSoapFault Members

CSoapFault Data Members

For information about the data members in **CSoapFault**, see CSoapFault Members

CSoapFault::m_soapErrCode

A constant identifying the fault.

```
SOAP_ERROR_CODE m_soapErrCode;
```

Remarks

This member contains a constant corresponding to the value of the SOAP faultcode element.

See Also

Reference

CSoapFault Class
SOAP_ERROR_CODE
CSoapFault::m_strFaultCode

Other Resources

CSoapFault Members

CSoapFault::m_strDetail

Application-specific error information related to the SOAP Body element.

```
CStringW m_strDetail;
```

Remarks

This member contains the value of the SOAP detail element.

See Also

Reference

CSoapFault Class

Other Resources

CSoapFault Members

CSoapFault::m_strFaultActor

A URI identifying the source of the fault.

```
CStringW m_strFaultActor;
```

Remarks

This member contains the value of the SOAP faultactor element.

See Also

Reference

CSoapFault Class

Other Resources

CSoapFault Members

CSoapFault::m_strFaultCode

A string identifying the fault.

```
CStringW m_strFaultCode;
```

Remarks

This member contains the value of the SOAP faultcode element.

See Also

Reference

CSoapFault Class

CSoapFault::m_soapErrCode

Other Resources

CSoapFault Members

CSoapFault::m_strFaultString

A human-readable explanation of the fault.

```
CStringW m_strFaultString;
```

Remarks

This member contains the value of the SOAP faultstring element.

See Also

Reference

CSoapFault Class

Other Resources

CSoapFault Members

CSoapHandler Class

This class provides functionality for handling SOAP requests in an XML Web service request handler. The class is injected as a base class by the soap_handler attribute.

```
template <typename THandler>
class CSoapHandler :
   public CSoapRootHandler,
   public CComObjectRootEx<CComMultiThreadModel>,
   public IRequestHandlerImpl<THandler>
```

Remarks

CSoapHandler is injected as a base class by the soap_handler attribute.

Soap handlers must use the memory manager returned by CSoapRootHandler::GetMemMgr to allocate memory for **[out]** or **[in, out]** parameters. To enable SOAP handlers to be used as ordinary COM objects or for handling SOAP requests without requiring any code changes, **CSoapHandler** uses the COM memory allocator when the object is being used as a COM object and it uses a heap more appropriate for a server environment when the object is being used to handle SOAP requests. Note that BSTRs must always be allocated with SysAllocString, CComBSTR, or a related function or class.

Requirements

Header: atlsoap.h

See Also

Tasks

PooledAsync Sample: Increases Performance with an Additional Thread Pool

SOAPTransport Sample: Communicates SOAP Messages Over Sockets, MSMQ, the File System, and HTTP Listener

Reference

CComObjectRootEx Class

IRequestHandlerImpl Class

CSoapRootHandler Class

Other Resources

CSoapHandler Members

CSoapHandler Members

Methods

CSoapHandler	The constructor.
GenerateAppError	Call this method to write a SOAP Fault to the response stream.
HandleRequest	This implementation of IRequestHandler::HandleRequest calls the appropriate SOAP method in the derived class and generates a SOAP response.
InitializeHandler	This implementation of IRequestHandler::InitializeHandler makes the handler ready for accepting SOAP requests.
SetHttpError	Call this method to set the ATL Server status code indicating failure.
SoapFault	Call this method to clear the currently buffered response, set the HTTP response code to 500 (Server Error) and write the specified SOAP Fault to the response stream.
UninitializeHandler	This implementation of IRequestHandler::UninitializeHandler calls CSoapRootHandler::UninitializeSOAP.

See Also

Reference

CSoapHandler Class

CSoapHandler Methods

For information about the methods in **CSoapHandler**, see CSoapHandler Members

CSoapHandler::CSoapHandler

The constructor.

```
CSoapHandler( );
```

Remarks

Initializes internal members.

See Also

Reference

CSoapHandler Class

Other Resources

CSoapHandler Members

CSoapHandler::GenerateAppError

Call this method to write a SOAP Fault to the response stream.

```
virtual ATL_NOINLINE HRESULT GenerateAppError(
   IWriteStream* pStream,
   HRESULT hr
);
```

Parameters

pStream

The response stream.

hr

The error code.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Calls FormatMessage on the HRESULT and passes the string as the *szDetail* parameter to SoapFault. The *errCode* parameter of that method is set to SOAP_E_SERVER.

Override this method to provide custom conversions of HRESULTs to strings.

See Also

Reference

CSoapHandler Class

Other Resources

CSoapHandler Members

CSoapHandler::HandleRequest

This implementation of IRequestHandler::HandleRequest calls the appropriate SOAP method in the derived class and generates a SOAP response.

```
HTTP_CODE HandleRequest(
   AtlServerRequest * pRequestInfo,
   IServiceProvider * /* pProvider */
);
```

Parameters

pRequestInfo

The information about the current request.

pProvider

The service provider.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

CSoapHandler Class

Other Resources

CSoapHandler Members

CSoapHandler::InitializeHandler

This implementation of IRequestHandler::InitializeHandler makes the handler ready for accepting SOAP requests.

```
HTTP_CODE InitializeHandler(
   AtlServerRequest * pRequestInfo,
   IServiceProvider * pProvider
);
```

Parameters

pRequestInfo

The information about the current request.

pProvider

The service provider.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

The implementation of this method calls IRequestHandlerImpl::InitializeHandler, CSoapRootHandler::InitializeSOAP, and CSoapRootHandler::SetMemMgr.

See Also

Reference

CSoapHandler Class

Other Resources

CSoapHandler Members

CSoapHandler::SetHttpError

Call this method to set the ATL Server status code indicating failure.

```
void SetHttpError(
   HTTP_CODE hcErr
);
```

Parameters

hcErr

The ATL Server status code indicating failure.

Example

See the SOAPTransport Sample.

See Also

Reference

CSoapHandler Class

Other Resources

CSoapHandler Members

CSoapHandler::SoapFault

Call this method to clear the currently buffered response, set the HTTP response code to 500 (Server Error) and write the specified SOAP Fault to the response stream.

```
HRESULT SoapFault(
   SOAP_ERROR_CODE errCode,
   const wchar_t* wszDetail,
   int cchDetail
);
```

Parameters

errCode

The SOAP_ERROR_CODE value describing the error that has occurred.

wszDetail

Application-specific information describing the error that has occurred.

cchDetail

The number of characters in *wszDetail*.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This override of CSoapRootHandler::SoapFault can itself be overridden to provide custom logging or other actions to be taken in response to error conditions.

Example

See the PooledAsync Sample and the SOAPTransport Sample.

See Also

Reference

CSoapHandler Class
CSoapRootHandler::SoapFault
CSoapHandler::GenerateAppError

Other Resources

CSoapHandler Members

CSoapHandler::UninitializeHandler

This implementation of IRequestHandler::UninitializeHandler calls CSoapRootHandler::UninitializeSOAP.

```
void UninitializeHandler( );
```

See Also

Reference

CSoapHandler Class

Other Resources

CSoapHandler Members

CSoapMSXMLInetClient Class

This class provides client access to XML Web services using Microsoft XML's ServerXMLHTTP object.

```
class CSoapMSXMLInetClient
```

Remarks

This class conforms to ATL Server's SOAP client archetype.

Typically this class is used with SPROXY-generated classes which build up the SOAP messages for you, but it is possible to use this class independently in the following way:

1. Create an instance and pass the location of the XML Web service to the constructor.
2. Call SetProxy or SetTimeout if necessary.
3. Call GetWriteStream and write the SOAP message to that stream.
4. Call SendRequest to send the SOAP message to the server.
5. If **SendRequest** succeeded, call GetReadStream to read the SOAP response.
6. If **SendRequest** failed, call GetClientError and GetStatusCode, or look at m_fault for error information.
7. Call CleanupClient before returning to Step 3 to send further messages.

Requirements

Header: atlsoap.h

Example

See the SecureSOAP Sample.

See Also

Reference

SOAP Client Archetype

Other Resources

CSoapMSXMLInetClient Members

CSoapMSXMLInetClient Members

Methods

CleanupClient	Call this method to clean up any resources allocated by the client.
CSoapMSXMLInetClient	The constructor.
~CSoapMSXMLInetClient	The destructor.
GetClientError	Call this method to get the error code following a request.
GetClientReader	This protected method can be called to obtain the pointer to the ISAXXMLReader interface used to read the SOAP response.
GetReadStream	Call this method to get the pointer to the IStream interface of the response stream.
GetStatusCode	Call this method to get the HTTP status code returned by the server following a request.
GetUrl	Call this method to get the URL of the XML Web service.
GetWriteStream	Call this method to get the pointer to the IWriteStream interface used to generate the request.
SendRequest	Call this method to send the SOAP message contained in the stream returned by GetWriteStream to the server.
SetClientError	Call this method to set the client error.
SetProxy	Call this method to set the proxy used to connect to the XML Web service.
SetTimeout	Call this method to set the timeout for connecting, sending, and receiving.
SetUrl	Call this method to set the URL of the XML Web service.

Data Members

m_fault	The SOAP Fault returned by the XML Web service following a request.
m_spHttpRequest	The pointer to the object used to make the HTTP request.

See Also

Reference

CSoapMSXMLInetClient Class

CSoapMSXMLInetClient Methods

For information about the methods in **CSoapMSXMLInetClient**, see CSoapMSXMLInetClient Members

CSoapMSXMLInetClient::CleanupClient

Call this method to clean up any resources allocated by the client.

```
void CleanupClient( );
```

Remarks

Called by the destructor. This method should be called to clean up the current request and response before making another request.

See Also

Reference

CSoapMSXMLInetClient Class

CSoapMSXMLInetClient::~CSoapMSXMLInetClient

Other Resources

CSoapMSXMLInetClient Members

CSoapMSXMLInetClient::CSoapMSXMLInetClient

The constructor.

```
CSoapMSXMLInetClient(
   LPCTSTR szUrl
);
CSoapMSXMLInetClient(
   LPCTSTR szServer,
   LPCTSTR szUri,
   short nPort = 80
);
```

Parameters

szUrl

The URL of the XML Web service.

szServer

The name of the server providing the XML Web service.

szUri

The URI of the XML Web service.

nPort

The port on which the XML Web service is available.

Remarks

The constructor allows you to pass the location of the XML Web service to the object. The location of the XML Web service can be obtained or set after construction by calling GetUrl and SetUrl.

See Also

Reference

CSoapMSXMLInetClient Class
CSoapMSXMLInetClient::GetUrl
CSoapMSXMLInetClient::SetUrl

Other Resources

CSoapMSXMLInetClient Members

CSoapMSXMLInetClient::~CSoapMSXMLInetClient

The destructor.

```
~CSoapMSXMLInetClient( );
```

Remarks

Calls CleanupClient.

See Also

Reference

CSoapMSXMLInetClient Class

Other Resources

CSoapMSXMLInetClient Members

CSoapMSXMLInetClient::GetClientError

Call this method to get the error code following a request.

```
SOAPCLIENT_ERROR GetClientError( );
```

Return Value

Returns the error code following a request.

Remarks

The error code returned by this method is set to SOAPCLIENT_SUCCESS by the constructor and by CleanupClient and can be set to another value by a call to SetClientError.

See Also

Reference

CSoapMSXMLInetClient Class
SOAPCLIENT_ERROR
CSoapMSXMLInetClient::SetClientError

Other Resources

CSoapMSXMLInetClient Members

CSoapMSXMLInetClient::GetClientReader

This protected method can be called to obtain the pointer to the ISAXXMLReader interface used to read the SOAP response.

```
virtual HRESULT GetClientReader(
   ISAXXMLReader ** pReader
);
```

Parameters

pReader

[out] Address of the pointer variable that, on success, receives the pointer to the ISAXXMLReader interface used to read the SOAP response.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method creates a SAXXMLReader30 object and returns the pointer to its **ISAXXMLReader** interface.

See Also

Reference

CSoapMSXMLInetClient Class

Other Resources

CSoapMSXMLInetClient Members

CSoapMSXMLInetClient::GetReadStream

Call this method to get the pointer to the IStream interface of the response stream.

```
HRESULT GetReadStream(
   IStream ** ppStream
);
```

Parameters

ppStream

[out] Address of the pointer variable that, on success, receives the pointer to the **IStream** interface of the XML response stream.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CSoapMSXMLInetClient Class

CSoapMSXMLInetClient::GetWriteStream

Other Resources

CSoapMSXMLInetClient Members

CSoapMSXMLInetClient::GetStatusCode

Call this method to get the HTTP status code returned by the server following a request.

```
int GetStatusCode( );
```

Return Value

Returns the HTTP status code returned by the server following a request.

See Also

Reference

CSoapMSXMLInetClient Class

Other Resources

CSoapMSXMLInetClient Members

CSoapMSXMLInetClient::GetUrl

Call this method to get the URL of the XML Web service.

```
HRESULT GetUrl(
   LPTSTR szUrl,
   LPDWORD pdwLen
);
```

Parameters

szUrl

The URL of the XML Web service.

pdwLen

On entry, *pdwLen* should point to a DWORD that indicates the size of the buffer in characters. On exit, the DWORD contains the number of characters transferred or available to be transferred into the buffer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CSoapMSXMLInetClient Class
CSoapMSXMLInetClient::CSoapMSXMLInetClient
CSoapMSXMLInetClient::SetUrl

Other Resources

CSoapMSXMLInetClient Members

CSoapMSXMLInetClient::GetWriteStream

Call this method to get the pointer to the IWriteStream interface used to generate the request.

```
IWriteStream* GetWriteStream( );
```

Return Value

Returns the pointer to the **IWriteStream** interface used to generate the request.

Remarks

This method returns the stream to which the SOAP message should be written.

See Also

Reference

CSoapMSXMLInetClient Class
CSoapMSXMLInetClient::GetReadStream
IWriteStream Interface

Other Resources

CSoapMSXMLInetClient Members

CSoapMSXMLInetClient::SendRequest

Call this method to send the SOAP message contained in the stream returned by GetWriteStream to the server.

```
HRESULT SendRequest(
   LPCTSTR szAction
);
```

Parameters

szAction

The SOAPAction HTTP header.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If the server sends back the 500 HTTP response code, the response body will be parsed for a SOAP Fault and the information used to set the members of m_fault.

See Also

Reference

CSoapMSXMLInetClient Class

Other Resources

CSoapMSXMLInetClient Members

CSoapMSXMLInetClient::SetClientError

Call this method to set the client error.

```
void SetClientError(
   SOAPCLIENT_ERROR errorState
);
```

Parameters

errorState

The error code.

See Also

Reference

CSoapMSXMLInetClient Class
SOAPCLIENT_ERROR
CSoapMSXMLInetClient::GetClientError

Other Resources

CSoapMSXMLInetClient Members

CSoapMSXMLInetClient::SetProxy

Call this method to set the proxy used to connect to the XML Web service.

```
HRESULT SetProxy(
   LPCTSTR szProxy = NULL,
   short nProxyPort = 80
);
```

Parameters

szProxy

The proxy server.

nProxyPort

The proxy port.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This class provides a non-functional implementation of this method that always returns S_OK and traces a message in debug builds.

The ServerXMLHTTP component used by this class relies on the WinHTTP proxy settings.

See Also

Reference

CSoapMSXMLInetClient Class

Other Resources

CSoapMSXMLInetClient Members

CSoapMSXMLInetClient::SetTimeout

Call this method to set the timeout for connecting, sending, and receiving.

```
void SetTimeout(
   DWORD dwTimeout
);
```

Parameters

dwTimeout

The timeout in milliseconds.

See Also

Reference

CSoapMSXMLInetClient Class

Other Resources

CSoapMSXMLInetClient Members

CSoapMSXMLInetClient::SetUrl

Call this method to set the URL of the XML Web service.

```
HRESULT SetUrl(
   LPCTSTR szUrl
);
```

Parameters

szUrl

The URL of the XML Web service.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CSoapMSXMLInetClient Class
CSoapMSXMLInetClient::CSoapMSXMLInetClient
CSoapMSXMLInetClient::GetUrl

Other Resources

CSoapMSXMLInetClient Members

CSoapMSXMLInetClient Data Members

For information about the data members in **CSoapMSXMLInetClient**, see CSoapMSXMLInetClient Members

CSoapMSXMLInetClient::m_fault

The SOAP Fault returned by the XML Web service following a request.

```
CSoapFault m_fault;
```

See Also

Reference

CSoapMSXMLInetClient Class

CSoapFault Class

Other Resources

CSoapMSXMLInetClient Members

CSoapMSXMLInetClient::m_spHttpRequest

The pointer to the object used to make the HTTP request.

```
CComPtr< IServerXMLHTTPRequest > m_spHttpRequest;
```

See Also

Reference

CSoapMSXMLInetClient Class

Other Resources

CSoapMSXMLInetClient Members

CSoapRootHandler Class

This class provides the code necessary to generate and parse SOAP messages. It is used as a base by SPROXY-generated SOAP client classes and XML Web service classes using the soap_handler attribute.

```
class CSoapRootHandler :
   public ISAXContentHandlerImpl
```

Remarks

Apart from calling CSoapRootHandler::GetMemMgr, it is unlikely that you will need to call or override the methods provided by this class unless you are providing some advanced functionality. Most SOAP handlers and clients will only need GetMemMgr.

Requirements

Header: atlsoap.h

See Also

Tasks

PooledAsync Sample: Increases Performance with an Additional Thread Pool
SOAPTransport Sample: Communicates SOAP Messages Over Sockets, MSMQ, the File System, and HTTP Listener
SecureSOAP Sample: Implements a Secure SOAP Communication (HTTPS)

Reference

ISAXContentHandlerImpl Class

Other Resources

CSoapRootHandler Members

CSoapRootHandler Members

Methods

BeginParse	This method is called to read a SOAP message from a stream and parse it.
CallFunctionInternal	After the SOAP message has been successfully read and parsed, this method is used to call the derived class methods responsible for handling the SOAP method specified in the SOAP message.
Cleanup	After the SOAP request has been handled or an error has occurred, this method is called cleanup any resources allocated as a result of parsing the SOAP message.
CreateReader	Creates a **SAXXMLReader** object used for reading and parsing the SOAP message.
CSoapRootHandler	The constructor.
GenerateResponse	After the SOAP message has been dispatched to the appropriate method is the derived class, this method is called to generate the response.
GetMemMgr	Call this method to get the memory manager used to allocate SOAP method parameters and SOAP headers.
GetReader	This method is called to get the pointer to the **ISAXXMLReader** interface used to read the SOAP message.
InitializeSOAP	This method is called to initialize the object in preparation for handling a SOAP request.
SetMemMgr	This method is called to set the memory manager used to allocate SOAP method parameters and SOAP headers.
SetReader	This method is called to set the **ISAXXMLReader** interface to be used to read the SOAP message.
SoapFault	This method is called when an error occurs.
UninitializeSOAP	This method is called to uninitialize the object when the SOAP message has been processed.

See Also

Reference

CSoapRootHandler Class

CSoapRootHandler Methods

For information about the methods in **CSoapRootHandler**, see CSoapRootHandler Members

CSoapRootHandler::BeginParse

This method is called to read a SOAP message from a stream and parse it.

```
virtual HRESULT BeginParse(
   IStream* pStream
);
```

Parameters

pStream

The stream containing the SOAP message to be parsed.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See the SOAPTransport Sample.

See Also

Reference

CSoapRootHandler Class

IStream

Other Resources

CSoapRootHandler Members

CSoapRootHandler::CallFunctionInternal

After the SOAP message has been successfully read and parsed, this method is used to call the derived class methods responsible for handling the SOAP method specified in the SOAP message.

```
HRESULT CallFunctionInternal( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See the SOAPTransport Sample.

See Also

Reference

CSoapRootHandler Class

Other Resources

CSoapRootHandler Members

CSoapRootHandler::Cleanup

After the SOAP request has been handled or an error has occurred, this method is called cleanup any resources allocated as a result of parsing the SOAP message.

```
virtual void Cleanup( );
```

Example

See the SecureSOAP Sample.

See Also

Reference

CSoapRootHandler Class

Other Resources

CSoapRootHandler Members

CSoapRootHandler::CreateReader

Creates a **SAXXMLReader** object used for reading and parsing the SOAP message.

```
HRESULT CreateReader( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CSoapRootHandler Class

Other Resources

CSoapRootHandler Members

CSoapRootHandler::CSoapRootHandler

The constructor.

```
CSoapRootHandler(
   ISAXXMLReader* pReader = 0
);
```

Parameters

pReader

A pointer to the **ISAXXMLReader** interface of the object to be used to read and parse the SOAP message.

See Also

Reference

CSoapRootHandler Class

Other Resources

CSoapRootHandler Members

CSoapRootHandler::GenerateResponse

After the SOAP message has been dispatched to the appropriate method is the derived class, this method is called to generate the response.

```
virtual HRESULT GenerateResponse(
   IWriteStream* pStream
);
```

Parameters

pStream

The stream to which the response should be written.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See the SOAPTransport Sample.

See Also

Reference

CSoapRootHandler Class

Other Resources

CSoapRootHandler Members

CSoapRootHandler::GetMemMgr

Call this method to get the memory manager used to allocate SOAP method parameters and SOAP headers.

```
IAtlMemMgr* GetMemMgr( );
```

Return Value

Returns the memory manager used to allocate SOAP method parameters and SOAP headers.

Remarks

Soap handlers and clients must use the memory manager returned by this method to allocate memory for **[out]** or **[in, out]** parameters and SOAP headers. However, note that BSTRs must always be allocated with SysAllocString, CComBSTR, or a related function or class.

See Also

Reference

CSoapRootHandler Class
CSoapRootHandler::SetMemMgr
CSoapHandler Class

Other Resources

CSoapRootHandler Members

CSoapRootHandler::GetReader

This method is called to get the pointer to the **ISAXXMLReader** interface used to read the SOAP message.

```
ISAXXMLReader* GetReader( );
```

Return Value

Returns the pointer to the **ISAXXMLReader** interface used to read the SOAP message.

See Also

Reference

CSoapRootHandler Class

CSoapRootHandler::SetReader

Other Resources

CSoapRootHandler Members

CSoapRootHandler::InitializeSOAP

This method is called to initialize the object in preparation for handling a SOAP request.

```
HRESULT InitializeSOAP(
   IServiceProvider* pProvider
);
```

Parameters

pProvider

The service provider.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Obtains the memory manager offered by the service provider if it provides one and calls CSoapRootHandler::CreateReader to get the XML parser object used to read the SOAP message.

Example

See the SOAPTransport Sample.

See Also

Reference

CSoapRootHandler Class

CSoapRootHandler::UninitializeSOAP

Other Resources

CSoapRootHandler Members

CSoapRootHandler::SetMemMgr

This method is called to set the memory manager used to allocate SOAP method parameters and SOAP headers.

```
IAtlMemMgr* SetMemMgr(
   IAtlMemMgr* pMemMgr
);
```

Parameters

pMemMgr

The memory manager to be used to allocate SOAP method parameters and SOAP headers.

Return Value

Returns the previous memory manager used to allocate SOAP method parameters and SOAP headers.

See Also

Reference

CSoapRootHandler Class

CSoapRootHandler::GetMemMgr

Other Resources

CSoapRootHandler Members

CSoapRootHandler::SetReader

This method is called to set the **ISAXXMLReader** interface to be used to read the SOAP message.

```
ISAXXMLReader* SetReader(
   ISAXXMLReader* pReader
);
```

Parameters

pReader

The pointer to the new **ISAXXMLReader** interface to be used to read the SOAP message.

Return Value

Returns the pointer to the previous **ISAXXMLReader** interface used to read the SOAP message.

See Also

Reference

CSoapRootHandler Class

CSoapRootHandler::GetReader

Other Resources

CSoapRootHandler Members

CSoapRootHandler::SoapFault

This method is called when an error occurs.

```
virtual HRESULT SoapFault(
   SOAP_ERROR_CODE /* errCode */,
   const wchar_t * /* wszDetail */,
   int /* cchDetail */
);
```

Parameters

errCode

The SOAP_ERROR_CODE value describing the error that has occurred.

wszDetail

Application-specific information describing the error that has occurred.

cchDetail

The number of characters in *wszDetail*.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method may be overridden in derived classes. See CSoapHandler::SoapFault, for example.

Example

See the PooledAsync Sample and the SOAPTransport Sample.

See Also

Reference

CSoapRootHandler Class

Other Resources

CSoapRootHandler Members

CSoapRootHandler::UninitializeSOAP

This method is called to uninitialize the object when the SOAP message has been processed.

```
void UninitializeSOAP( );
```

Example

See the SOAPTransport Sample.

See Also

Reference

CSoapRootHandler Class

CSoapRootHandler::InitializeSOAP

Other Resources

CSoapRootHandler Members

CSoapSocketClientT Class

This class provides client access to XML Web services using sockets.

```
template <
   typename TSocketClass = ZEvtSyncSocket
>
class CSoapSocketClientT;
```

Parameters

TSocketClass

The class used to provide the underlying network socket support. For the current version of the library this parameter should be set to the internal class, **ZEvtSyncSocket**.

Remarks

This class conforms to ATL Server's SOAP client archetype.

Typically this class is used with SPROXY-generated classes which build up the SOAP messages for you, but it is possible to use this class independently in the following way:

1. Create an instance and pass the location of the XML Web service to the constructor.
2. Call SetProxy or SetTimeout if necessary.
3. Call GetWriteStream and write the SOAP message to that stream.
4. Call SendRequest to send the SOAP message to the server.
5. If **SendRequest** succeeded, call GetReadStream to read the SOAP response.
6. If **SendRequest** failed, call GetClientError and GetStatusCode, or look at m_fault for error information.
7. Call CleanupClient before returning to Step 3 to send further messages.

Requirements

Header: atlsoap.h

Example

See the PooledAsync Sample and the SecureSOAP Sample.

See Also

Other Resources

CSoapSocketClientT Members

CSoapSocketClientT Members

Methods

CleanupClient	Call this method to clean up any resources allocated by the client.
CSoapSocketClientT	The constructor.
~CSoapSocketClientT	The destructor.
GetClientError	Call this method to get the error code following a request.
GetClientReader	This protected method can be called to obtain the pointer to the ISAXXMLReader interface used to read the SOAP response.
GetReadStream	Call this method to get the pointer to the IStream interface of the response stream.
GetStatusCode	Call this method to get the HTTP status code returned by the server following a request.
GetUrl	Call this method to get the URL of the XML Web service.
GetWriteStream	Call this method to get the pointer to the IWriteStream interface used to generate the request.
SendRequest	Call this method to send the SOAP message contained in the stream returned by GetWriteStream to the server.
SetClientError	Call this method to set the client error.
SetProxy	Call this method to set the proxy used to connect to the XML Web service.
SetTimeout	Call this method to set the timeout for connecting, sending, and receiving.
SetUrl	Call this method to set the URL of the XML Web service.

Data Members

m_fault	The SOAP Fault returned by the XML Web service following a request.
m_socket	The socket used to make the SOAP request.

See Also

Reference

CSoapSocketClientT Class

CSoapSocketClientT Methods

For information about the methods in **CSoapSocketClientT**, see CSoapSocketClientT Members

CSoapSocketClientT::CleanupClient

Call this method to clean up any resources allocated by the client.

```
void CleanupClient( );
```

Remarks

Called by the destructor. This method should be called to clean up the current request and response before making another request.

See Also

Reference

CSoapSocketClientT Class

CSoapSocketClientT::~CSoapSocketClientT

Other Resources

CSoapSocketClientT Members

CSoapSocketClientT::CSoapSocketClientT

The constructor.

```
CSoapSocketClientT(
   LPCTSTR szUrl
);
CSoapSocketClientT(
   LPCTSTR szServer,
   LPCTSTR szUri,
   ATL_URL_PORT nPort = 80
);
```

Parameters

szUrl

The URL of the XML Web service.

szServer

The name of the server providing the XML Web service.

szUri

The URI of the XML Web service.

nPort

The port on which the XML Web service is available.

Remarks

The constructor allows you to pass the location of the XML Web service to the object. The location of the XML Web service can be obtained or set after construction by calling GetUrl and SetUrl.

See Also

Reference

CSoapSocketClientT Class
CSoapSocketClientT::GetUrl
CSoapSocketClientT::SetUrl

Other Resources

CSoapSocketClientT Members

CSoapSocketClientT::~CSoapSocketClientT

The destructor.

```
~CSoapSocketClientT( );
```

Remarks

Calls CleanupClient.

See Also

Reference

CSoapSocketClientT Class

Other Resources

CSoapSocketClientT Members

CSoapSocketClientT::GetClientError

Call this method to get the error code following a request.

```
SOAPCLIENT_ERROR GetClientError( );
```

Return Value

Returns the error code following a request.

Remarks

The error code returned by this method is set to SOAPCLIENT_SUCCESS by the constructor and by CleanupClient and can be set to another value by a call to SetClientError.

See Also

Reference

CSoapSocketClientT Class
SOAPCLIENT_ERROR
CSoapSocketClientT::SetClientError

Other Resources

CSoapSocketClientT Members

CSoapSocketClientT::GetClientReader

This protected method can be called to obtain the pointer to the **ISAXXMLReader** interface used to read the SOAP response.

```
virtual HRESULT GetClientReader(
   ISAXXMLReader ** pReader
);
```

Parameters

pReader

[out] Address of the pointer variable that, on success, receives the pointer to the **ISAXXMLReader** interface used to read the SOAP response.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method creates a SAXXMLReader30 object and returns the pointer to its **ISAXXMLReader** interface.

See Also

Reference

CSoapSocketClientT Class

Other Resources

CSoapSocketClientT Members

CSoapSocketClientT::GetReadStream

Call this method to get the pointer to the IStream interface of the response stream.

```
HRESULT GetReadStream(
   IStream ** ppStream
);
```

Parameters

ppStream

[out] Address of the pointer variable that, on success, receives the pointer to the **IStream** interface of the XML response stream.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CSoapSocketClientT Class

CSoapSocketClientT::GetWriteStream

Other Resources

CSoapSocketClientT Members

CSoapSocketClientT::GetStatusCode

Call this method to get the HTTP status code returned by the server following a request.

```
int GetStatusCode( );
```

Return Value

Returns the HTTP status code returned by the server following a request.

See Also

Reference

CSoapSocketClientT Class

Other Resources

CSoapSocketClientT Members

CSoapSocketClientT::GetUrl

Call this method to get the URL of the XML Web service.

```
HRESULT GetUrl(
   LPTSTR szUrl,
   LPDWORD pdwLen
);
```

Parameters

szUrl

The URL of the XML Web service.

pdwLen

On entry, *pdwLen* should point to a DWORD that indicates the size of the buffer in characters. On exit, the DWORD contains the number of characters transferred or available to be transferred into the buffer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CSoapSocketClientT Class
CSoapSocketClientT::CSoapSocketClientT
CSoapSocketClientT::SetUrl

Other Resources

CSoapSocketClientT Members

CSoapSocketClientT::GetWriteStream

Call this method to get the pointer to the IWriteStream interface used to generate the request.

```
IWriteStream* GetWriteStream( );
```

Return Value

Returns the pointer to the **IWriteStream** interface used to generate the request.

Remarks

This method returns the stream to which the SOAP message should be written.

See Also

Reference

CSoapSocketClientT Class
CSoapSocketClientT::GetReadStream
IWriteStream Interface

Other Resources

CSoapSocketClientT Members

CSoapSocketClientT::SendRequest

Call this method to send the SOAP message contained in the stream returned by GetWriteStream to the server.

```
HRESULT SendRequest(
   LPCTSTR szAction
);
```

Parameters

szAction

The SOAPAction HTTP header.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If the server sends back the 500 HTTP response code, the response body will be parsed for a SOAP Fault and the information used to set the members of m_fault.

See Also

Reference

CSoapSocketClientT Class

Other Resources

CSoapSocketClientT Members

CSoapSocketClientT::SetClientError

Call this method to set the client error.

```
void SetClientError(
   SOAPCLIENT_ERROR errorState
);
```

Parameters

errorState

The error code.

See Also

Reference

CSoapSocketClientT Class
SOAPCLIENT_ERROR
CSoapSocketClientT::GetClientError

Other Resources

CSoapSocketClientT Members

CSoapSocketClientT::SetProxy

Call this method to set the proxy used to connect to the XML Web service.

```
HRESULT SetProxy(
   LPCTSTR szProxy = NULL,
   short nProxyPort = 80
);
```

Parameters

szProxy

The proxy server. If NULL, the proxy server used by Windows Internet Settings is retrieved from the registry.

nProxyPort

The proxy port.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CSoapSocketClientT Class

Other Resources

CSoapSocketClientT Members

CSoapSocketClientT::SetTimeout

Call this method to set the timeout for connecting, sending, and receiving.

```
void SetTimeout(
   DWORD dwTimeout
);
```

Parameters

dwTimeout

The timeout in milliseconds.

See Also

Reference

CSoapSocketClientT Class

Other Resources

CSoapSocketClientT Members

CSoapSocketClientT::SetUrl

Call this method to set the URL of the XML Web service.

```
HRESULT SetUrl(
   LPCTSTR szUrl
);
```

Parameters

szUrl

The URL of the XML Web service.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See the SecureSOAP Sample.

See Also

Reference

CSoapSocketClientT Class
CSoapSocketClientT::CSoapSocketClientT
CSoapSocketClientT::GetUrl

Other Resources

CSoapSocketClientT Members

CSoapSocketClientT Data Members

For information about the data members in **CSoapSocketClientT**, see CSoapSocketClientT Members

CSoapSocketClientT::m_fault

The SOAP Fault returned by the XML Web service following a request.

```
CSoapFault m_fault;
```

See Also

Reference

CSoapSocketClientT Class

CSoapFault Class

Other Resources

CSoapSocketClientT Members

CSoapSocketClientT::m_socket

The socket used to make the SOAP request.

```
CAtlHttpClientT< TSocketClass > m_socket;
```

See Also

Reference

CSoapSocketClientT Class

CAtlHttpClientT Class

Other Resources

CSoapSocketClientT Members

CSoapWininetClient Class

This class provides client access to XML Web services using Microsoft Win32 Internet functions(WinInet).

```
class CSoapWininetClient
```

Remarks

This class conforms to ATL Server's SOAP client archetype.

Typically this class is used with SPROXY-generated classes which build up the SOAP messages for you, but it is possible to use this class independently in the following way:

1. Create an instance and pass the location of the XML Web service to the constructor.
2. Call SetProxy or SetTimeout if necessary.
3. Call GetWriteStream and write the SOAP message to that stream.
4. Call SendRequest to send the SOAP message to the server.
5. If **SendRequest** succeeded, call GetReadStream to read the SOAP response.
6. If **SendRequest** failed, call GetClientError and GetStatusCode, or look at m_fault for error information.
7. Call CleanupClient before returning to Step 3 to send further messages.

Requirements

Header: atlsoap.h

Example

See the SecureSOAP Sample.

See Also

Other Resources

CSoapWininetClient Members

CSoapWininetClient Members

Methods

CleanupClient	Call this method to clean up any resources allocated by the client.
CSoapWininetClient	The constructor.
~CSoapWininetClient	The destructor.
GetClientError	Call this method to get the error code following a request.
GetClientReader	This protected method can be called to obtain the pointer to the **ISAXXMLReader** interface used to read the SOAP response.
GetReadStream	Call this method to get the pointer to the IStream interface of the response stream.
GetStatusCode	Call this method to get the HTTP status code returned by the server following a request.
GetUrl	Call this method to get the URL of the XML Web service.
GetWriteStream	Call this method to get the pointer to the IWriteStream interface used to generate the request.
SendRequest	Call this method to send the SOAP message contained in the stream returned by GetWriteStream to the server.
SetClientError	Call this method to set the client error.
SetProxy	Call this method to set the proxy used to connect to the XML Web service.
SetTimeout	Call this method to set the timeout for connecting, sending, and receiving.
SetUrl	Call this method to set the URL of the XML Web service.

Data Members

m_fault	The SOAP Fault returned by the XML Web service following a request.
m_hConnection	The handle to the WinInet HTTP connection.
m_hInternet	The handle to the WinInet session.
m_hRequest	The handle to the WinInet HTTP request.

See Also

Reference

CSoapWininetClient Class

CSoapWininetClient Methods

For information about the methods in **CSoapWininetClient**, see CSoapWininetClient Members

CSoapWininetClient::CleanupClient

Call this method to clean up any resources allocated by the client.

```
void CleanupClient( );
```

Remarks

Called by the destructor. This method should be called to clean up the current request and response before making another request.

See Also

Reference

CSoapWininetClient Class

CSoapWininetClient::~CSoapWininetClient

Other Resources

CSoapWininetClient Members

CSoapWininetClient::CSoapWininetClient

The constructor.

```
CSoapWininetClient(
   LPCTSTR szUrl
);
CSoapWininetClient(
   LPCTSTR szServer,
   LPCTSTR szUri,
   short nPort = 80
);
```

Parameters

szUrl

The URL of the XML Web service.

szServer

The name of the server providing the XML Web service.

szUri

The URI of the XML Web service.

nPort

The port on which the XML Web service is available.

Remarks

The constructor allows you to pass the location of the XML Web service to the object. The location of the XML Web service can be obtained or set after construction by calling GetUrl and SetUrl.

See Also

Reference

CSoapWininetClient Class
CSoapWininetClient::GetUrl
CSoapWininetClient::SetUrl

Other Resources

CSoapWininetClient Members

CSoapWininetClient::~CSoapWininetClient

The destructor.

```
~CSoapWininetClient( );
```

Remarks

Calls CleanupClient.

See Also

Reference

CSoapWininetClient Class

Other Resources

CSoapWininetClient Members

CSoapWininetClient::GetClientError

Call this method to get the error code following a request.

```
SOAPCLIENT_ERROR GetClientError( );
```

Return Value

Returns the error code following a request.

Remarks

The error code returned by this method is set to SOAPCLIENT_SUCCESS by the constructor and by CleanupClient and can be set to another value by a call to SetClientError.

See Also

Reference

CSoapWininetClient Class
SOAPCLIENT_ERROR
CSoapWininetClient::SetClientError

Other Resources

CSoapWininetClient Members

CSoapWininetClient::GetClientReader

This protected method can be called to obtain the pointer to the **ISAXXMLReader** interface used to read the SOAP response.

```
virtual HRESULT GetClientReader(
   ISAXXMLReader ** pReader
);
```

Parameters

pReader

[out] Address of the pointer variable that, on success, receives the pointer to the **ISAXXMLReader** interface used to read the SOAP response.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method creates a SAXXMLReader30 object and returns the pointer to its **ISAXXMLReader** interface.

See Also

Reference

CSoapWininetClient Class

Other Resources

CSoapWininetClient Members

CSoapWininetClient::GetReadStream

Call this method to get the pointer to the IStream interface of the response stream.

```
HRESULT GetReadStream(
   IStream ** ppStream
);
```

Parameters

ppStream

[out] Address of the pointer variable that, on success, receives the pointer to the **IStream** interface of the XML response stream.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CSoapWininetClient Class

CSoapWininetClient::GetWriteStream

Other Resources

CSoapWininetClient Members

CSoapWininetClient::GetStatusCode

Call this method to get the HTTP status code returned by the server following a request.

```
int GetStatusCode( );
```

Return Value

Returns the HTTP status code returned by the server following a request.

See Also

Reference

CSoapWininetClient Class

Other Resources

CSoapWininetClient Members

CSoapWininetClient::GetUrl

Call this method to get the URL of the XML Web service.

```
HRESULT GetUrl(
   LPTSTR szUrl,
   LPDWORD pdwLen
);
```

Parameters

szUrl

The URL of the XML Web service.

pdwLen

On entry, *pdwLen* should point to a DWORD that indicates the size of the buffer in characters. On exit, the DWORD contains the number of characters transferred or available to be transferred into the buffer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CSoapWininetClient Class
CSoapWininetClient::CSoapWininetClient
CSoapWininetClient::SetUrl

Other Resources

CSoapWininetClient Members

CSoapWininetClient::GetWriteStream

Call this method to get the pointer to the IWriteStream interface used to generate the request.

```
IWriteStream* GetWriteStream( );
```

Return Value

Returns the pointer to the **IWriteStream** interface used to generate the request.

Remarks

This method returns the stream to which the SOAP message should be written.

See Also

Reference

CSoapWininetClient Class
CSoapWininetClient::GetReadStream
IWriteStream Interface

Other Resources

CSoapWininetClient Members

CSoapWininetClient::SendRequest

Call this method to send the SOAP message contained in the stream returned by GetWriteStream to the server.

```
HRESULT SendRequest(
   LPCTSTR szAction
);
```

Parameters

szAction

The SOAPAction HTTP header.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If the server sends back the 500 HTTP response code, the response body will be parsed for a SOAP Fault and the information used to set the members of m_fault.

See Also

Reference

CSoapWininetClient Class

Other Resources

CSoapWininetClient Members

CSoapWininetClient::SetClientError

Call this method to set the client error.

```
void SetClientError(
   SOAPCLIENT_ERROR errorState
);
```

Parameters

errorState

The error code.

See Also

Reference

CSoapWininetClient Class
SOAPCLIENT_ERROR
CSoapWininetClient::GetClientError

Other Resources

CSoapWininetClient Members

CSoapWininetClient::SetProxy

Call this method to set the proxy used to connect to the XML Web service.

```
HRESULT SetProxy(
   LPCTSTR szProxy = NULL,
   short nProxyPort = 80
);
```

Parameters

szProxy

The proxy server.

nProxyPort

The proxy port.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CSoapWininetClient Class

Other Resources

CSoapWininetClient Members

CSoapWininetClient::SetTimeout

Call this method to set the timeout for connecting, sending, and receiving.

```
void SetTimeout(
   DWORD dwTimeout
);
```

Parameters

dwTimeout

The timeout in milliseconds.

See Also

Reference

CSoapWininetClient Class

Other Resources

CSoapWininetClient Members

CSoapWininetClient::SetUrl

Call this method to set the URL of the XML Web service.

```
HRESULT SetUrl(
   LPCTSTR szUrl
);
```

Parameters

szUrl

The URL of the XML Web service.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CSoapWininetClient Class
CSoapWininetClient::CSoapWininetClient
CSoapWininetClient::GetUrl

Other Resources

CSoapWininetClient Members

CSoapWininetClient Data Members

For information about the data members in **CSoapWininetClient**, see CSoapWininetClient Members

CSoapWininetClient::m_fault

The SOAP Fault returned by the XML Web service following a request.

```
CSoapFault m_fault;
```

See Also

Reference

CSoapWininetClient Class

CSoapFault Class

Other Resources

CSoapWininetClient Members

CSoapWininetClient::m_hConnection

The handle to the WinInet HTTP connection.

```
HINTERNET m_hConnection;
```

See Also

Reference

CSoapWininetClient Class

InternetConnect

Other Resources

CSoapWininetClient Members

CSoapWininetClient::m_hInternet

The handle to the WinInet session.

```
HINTERNET m_hInternet;
```

See Also

Reference

CSoapWininetClient Class

InternetOpen

Other Resources

CSoapWininetClient Members

CSoapWininetClient::m_hRequest

The handle to the WinInet HTTP request.

```
HINTERNET m_hRequest;
```

See Also

Reference

CSoapWininetClient Class

HttpOpenRequest

Other Resources

CSoapWininetClient Members

CStdRequestStats Class

Use this class as the argument for the *CRequestStatClass* template parameter to CIsapiExtension if you want to capture request-handling statistics.

```
class CStdRequestStats :
   public CRequestStats
```

Remarks

The request handling statistics captured by this class are available via the IRequestStats interface implemented by CIsapiExtension.

This class conforms to the Request Statistics Archetype. See the documentation for that archetype for other conforming classes.

Requirements

Header: atlisapi.h

See Also

Reference

CRequestStats Class

Request Statistics Archetype

Other Resources

CStdRequestStats Members

ATL Server Classes

CStdRequestStats Members

Methods

Initialize	This method is called by CIsapiExtension to initialize the object.
Uninitialize	This method is called by CIsapiExtension to uninitialize the object.

See Also

Reference

CStdRequestStats Class

Other Resources

CRequestStats Members

CStdRequestStats Methods

For information about the methods in **CStdRequestStats**, see CStdRequestStats Members.

CStdRequestStats::Initialize

This method is called by CIsapiExtension to initialize the object.

```
HRESULT Initialize( ) throw( );
```

Return Value

Returns S_OK on success or an error HRESULT on failure.

See Also

Reference

CStdRequestStats Class

Other Resources

CStdRequestStats Members

CStdRequestStats::Uninitialize

This method is called by CIsapiExtension to uninitialize the object.

```
void Uninitialize( ) throw( );
```

See Also

Reference

CStdRequestStats Class

Other Resources

CStdRequestStats Members

CStdStatClass Class

Use this class anywhere that a class conforming to the cache statistics archetype is expected if you want to collect cache statistics in the class's data members.

```
class CStdStatClass
```

Remarks

This class provides a public interface that conforms to the cache statistics archetype. The methods are implemented to store the statistics in the data members provided by CPerfStatObject using thread-safe interlocked operations.

Classes that make use of the cache statistics archetype typically expose the statistics gathered by this class using the IMemoryCacheStats interface.

Typically, you don't need to call the methods provided by this class directly.

Requirements

Header: atlcache.h

See Also

Reference

Cache Statistics Archetype

Other Resources

Caching

CStdStatClass Members

Caching Reference

CStdStatClass Members

Methods

AddElement	Call this method when an item is added to the cache.
CStdStatClass	The constructor.
GetCurrentAllocSize	Call this method to get the current number of bytes used by the cache for storing data.
GetCurrentEntryCount	Call this method to get the current number of items in the cache.
GetHitCount	Call this method to get the number of times the cache successfully returned an item.
GetMaxAllocSize	Call this method to get the maximum number of bytes used by the cache at one time since statistics gathering was started.
GetMaxEntryCount	Call this method to get the maximum number of items in the cache at one time since statistics gathering was started.
GetMissCount	Call this method to get the number of times the cache did not contain a requested item.
Hit	Call this method when an item is successfully retrieved from the cache.
Initialize	Call this method to initialize the object.
Miss	Call this method when a requested item cannot be found in the cache.
ReleaseElement	Call this method when an item is removed from the cache.
ResetCounters	Call this method to reset all the statistics to zero.
Uninitialize	Call this method to uninitialize the object.

Data Members

m_pStats	The pointer to the object used for tracking the statistics.
m_stats	The object used for tracking the statistics unless a pointer to another object is passed to the constructor.

See Also

Reference

CStdStatClass Class

Other Resources

Caching

Caching Reference

CStdStatClass Methods

For information about the methods in **CStdStatClass**, see CStdStatClass Members

CStdStatClass::AddElement

Call this method when an item is added to the cache.

```
void AddElement(
   DWORD dwBytes
);
```

Parameters

dwBytes

The number of bytes used by the newly added item.

Remarks

Updates the statistics that keep track of the number of entries and memory used by the cache.

See Also

Reference

CStdStatClass Class

CStdStatClass::ReleaseElement

Other Resources

Caching

CStdStatClass Members

Caching Reference

CStdStatClass::CStdStatClass

The constructor.

```
CStdStatClass( );
```

Remarks

Initializes CStdStatClass::m_pStats to point to CStdStatClass::m_stats.

See Also

Reference

CStdStatClass Class

Other Resources

Caching

CStdStatClass Members

Caching Reference

CStdStatClass::GetCurrentAllocSize

Call this method to get the current number of bytes used by the cache for storing data.

```
DWORD GetCurrentAllocSize( );
```

Return Value

Returns the current number of bytes used by the cache for storing data.

Remarks

Returns information from the object pointed to by CStdStatClass::m_pStats.

See Also

Reference

CStdStatClass Class
CStdStatClass::GetMaxAllocSize

Other Resources

Caching
CStdStatClass Members
Caching Reference

CStdStatClass::GetCurrentEntryCount

Call this method to get the current number of items in the cache.

```
DWORD GetCurrentEntryCount( );
```

Return Value

Returns the current number of items in the cache.

Remarks

Returns information from the object pointed to by CStdStatClass::m_pStats.

See Also

Reference

CStdStatClass Class

CStdStatClass::GetMaxEntryCount

Other Resources

Caching

CStdStatClass Members

Caching Reference

CStdStatClass::GetHitCount

Call this method to get the number of times the cache successfully returned an item.

```
DWORD GetHitCount( );
```

Return Value

Returns the number of times the cache successfully returned an item.

Remarks

Returns information from the object pointed to by CStdStatClass::m_pStats.

See Also

Reference

CStdStatClass Class
CStdStatClass::GetMissCount

Other Resources

Caching
CStdStatClass Members
Caching Reference

CStdStatClass::GetMaxAllocSize

Call this method to get the maximum number of bytes used by the cache at one time since statistics gathering was started.

```
DWORD GetMaxAllocSize( );
```

Return Value

Returns the maximum number of bytes used by the cache at one time since statistics gathering was started.

Remarks

Returns information from the object pointed to by CStdStatClass::m_pStats.

See Also

Reference

CStdStatClass Class
CStdStatClass::GetCurrentAllocSize

Other Resources

Caching
CStdStatClass Members
Caching Reference

CStdStatClass::GetMaxEntryCount

Call this method to get the maximum number of items in the cache at one time since statistics gathering was started.

```
DWORD GetMaxEntryCount( );
```

Return Value

Returns the maximum number of items in the cache at one time since statistics gathering was started.

Remarks

Returns information from the object pointed to by CStdStatClass::m_pStats.

See Also

Reference

CStdStatClass Class
CStdStatClass::GetCurrentEntryCount

Other Resources

Caching
CStdStatClass Members
Caching Reference

CStdStatClass::GetMissCount

Call this method to get the number of times the cache did not contain a requested item.

```
DWORD GetMissCount( );
```

Return Value

Returns the number of times the cache did not contain a requested item.

Remarks

Returns information from the object pointed to by CStdStatClass::m_pStats.

See Also

Reference

CStdStatClass Class
CStdStatClass::GetHitCount

Other Resources

Caching
CStdStatClass Members
Caching Reference

CStdStatClass::Hit

Call this method when an item is successfully retrieved from the cache.

```
void Hit( );
```

Remarks

Increments the count of cache hits.

See Also

Reference

CStdStatClass Class

CStdStatClass::Miss

Other Resources

Caching

CStdStatClass Members

Caching Reference

CStdStatClass::Initialize

Call this method to initialize the object.

```
HRESULT Initialize(
   CPerfStatObject* pStats = NULL
);
```

Parameters

pStats

The object used for tracking the statistics. If **NULL**, m_stats is used to track statistics.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sets CStdStatClass::m_pStats then calls CStdStatClass::ResetCounters. This function is not thread safe by design.

See Also

Reference

CStdStatClass Class
CStdStatClass::Uninitialize

Other Resources

Caching
CStdStatClass Members
Caching Reference

CStdStatClass::Miss

Call this method when a requested item cannot be found in the cache.

```
void Miss( );
```

Remarks

Increments the count of cache misses.

See Also

Reference

CStdStatClass Class

CStdStatClass::Hit

Other Resources

Caching

CStdStatClass Members

Caching Reference

CStdStatClass::ReleaseElement

Call this method when an item is removed from the cache.

```
void ReleaseElement(
   DWORD dwBytes
);
```

Parameters

dwBytes

The number of bytes freed by the item just removed.

Remarks

Updates the statistics that keep track of the number of entries and memory used by the cache.

See Also

Reference

CStdStatClass Class

CStdStatClass::AddElement

Other Resources

Caching

CStdStatClass Members

Caching Reference

CStdStatClass::ResetCounters

Call this method to reset all the statistics to zero.

```
void ResetCounters( );
```

Remarks

This method is not thread-safe by design.

See Also

Reference

CStdStatClass Class

Other Resources

Caching

CStdStatClass Members

Caching Reference

CStdStatClass::Uninitialize

Call this method to uninitialize the object.

```
HRESULT Uninitialize( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method is not thread-safe by design.

See Also

Reference

CStdStatClass Class
CStdStatClass::Initialize

Other Resources

Caching
CStdStatClass Members
Caching Reference

CStdStatClass Data Members

For information about the data members in **CStdStatClass**, see CStdStatClass Members

CStdStatClass::m_pStats

The pointer to the object used for tracking the statistics.

```
CPerfStatObject* m_pStats;
```

Remarks

This pointer is initialized in the constructor.

See Also

Reference

CStdStatClass Class
CStdStatClass::CStdStatClass
CStdStatClass::m_stats

Other Resources

Caching
CStdStatClass Members
CPerfStatObject Members
Caching Reference

CStdStatClass::m_stats

The object used for tracking the statistics unless a pointer to another object is passed to the constructor.

```
CPerfStatObject m_stats;
```

See Also

Reference

CStdStatClass Class
CStdStatClass::CStdStatClass
CStdStatClass::m_pStats

Other Resources

Caching
CStdStatClass Members
CPerfStatObject Members
Caching Reference

CStencil Class

This class is used to parse a stencil and generate a text stream by replacing elements found within the stencil by output generated by an object implementing the ITagReplacer interface.

```
class CStencil :
   public IMemoryCacheClient
```

Remarks

CStencil and CHtmlStencil are used in ATL Server Web applications for handling server response files. **CStencil** provides the code for parsing and rendering the following tags in a server response file:

- Codepage Tag
- Handler Tag
- Locale Tag
- Replacement Tag

If you are using wizard-generated ATL Server projects and find that the server response file syntax is adequate for your needs, you may not need to use these classes directly.

However, if you wish to load and manipulate server response files directly, or extend the SRF syntax for your own needs, you will probably want to use these classes directly.

To use this class

1. Create an instance of this class.
2. Load the stencil from a resource, file, or string by calling LoadFromResource, LoadFromFile, or LoadFromString.
3. Call CStencil::Parse.
4. Call CStencil::Render.

To customize parsing

1. Derive from this class.
2. Override CStencil::ParseReplacements.
3. Override CStencil::ParseToken.
4. Override CStencil::FinishParseReplacements.

To customize rendering

- Override CStencil::RenderToken

Requirements

Header: atlstencil.h

See Also

Reference

IMemoryCacheClient Interface

CHtmlStencil Class

Other Resources

CStencil Members

CStencil Members

Methods

AddError	Call this method when an error is encountered during parsing of the stencil.
AddRef	Implementation of **IUnknown::AddRef**.
AddToken	Call this method to add a token to the collection managed by this object.
CStencil	The constructor.
~CStencil	The destructor.
FinishParseReplacements	Override this method to enhance or replace the stencil parsing provided by this class.
Free	Implementation of IMemoryCacheClient::Free.
GetBufferEnd	Call this method to get the pointer to the end of the buffer holding the stencil for this object.
GetBufferStart	Call this method to get the pointer to the start of the buffer holding the stencil for this object.
GetCacheItem	Call this method to get the handle identifying this stencil in a cache.
GetCodePage	Call this method to get the code page for the current stencil.
GetHandlerName	Call this method to get the DLL path and handler name of the request handler parsed from the handler tag of the currently loaded stencil.
GetLastChecked	Call this method to get the date-time at which the last modified time of the stencil loaded by CStencil::LoadFromFile was last checked.
GetLastModified	Call this method to get the date-time at which the stencil loaded by CStencil::LoadFromFile was last modified.
GetToken	Call this method to get a pointer to the specified StencilToken.
GetTokenCount	Call this method to get the number of stencil tokens in the collection managed by this object.
IsValidIndex	Call this method to determine whether a stencil token index is valid.
LoadFromFile	Call this method to load a stencil from a file.
LoadFromResource	Call this method to load a stencil from a resource.
LoadFromResourceEx	Call this method to load a stencil from a resource associated with a particular language.
LoadFromString	Call this method to load a stencil from a string.
Parse	Call this method to parse the loaded stencil.
ParseCodepage	Call this method to parse a codepage tag.

ParseElse	Call this method to parse an {{else}} replacement tag.
ParseEndIf	Call this method to parse an {{endif}} replacement tag.
ParseEndWhile	Call this method to parse an {{endwhile}} replacement tag.
ParseHandler	Call this method to parse a handler tag.
ParseIf	Call this method to parse an {{if}} replacement tag.
ParseLocale	Call this method to parse a locale tag.
ParseReplacement	Call this method to parse a replacement tag.
ParseReplacements	Override this method to enhance or replace the stencil parsing provided by this class.
ParseSuccessful	Call this method to determine whether stencil parsing was successful.
ParseToken	Override this method to parse a token delimited by double braces ({{}}).
ParseWhile	Call this method to parse a {{while}} replacement tag.
QueryInterface	Implementation of **IUnknown::QueryInterface**.
Release	Implementation of **IUnknown::Release**.
Render	Call this method to transform the stencil into a response stream using an object implementing ITagReplacer to replace tokens with dynamic content.
RenderErrors	Call this method to write parse errors to a stream.
RenderToken	Override this method to customize the rendering of a stencil token.
SetCacheItem	Call this method to set the handle identifying this stencil in a cache.
SetErrorResource	Call this method to set the handle of the module supplying the resources used for error strings.
SetLastChecked	Call this method to set the date-time at which the last modified time of the stencil loaded by CStencil::LoadFromFile was last checked.
Uninitialize	Call this method if you need to reuse this object with another stencil or before destruction.

Data Members

m_pMemMgr	The memory manager.
m_pReplacer	The tag replacer used to provide the dynamic content for the replacement tags in the current stencil.

Static Data Members

m_crtHeap	The CRT heap is used to allocate memory if no memory manager is passed to the constructor of this class.

Enumerations

PARSE_TOKEN_RESULT	Used to describe the result of stencil processing and indicate whether a line break following a token sh ould be ignored.

See Also

Reference

CStencil Class

CStencil Methods

For information about the methods in **CStencil**, see CStencil Members.

CStencil::AddError

Call this method when an error is encountered during parsing of the stencil.

```
bool AddError(
   UINT uID,
   LPCSTR szPosition
) throw( );
```

Parameters

uID

The ID of the string resource describing the error.

szPosition

The position at which the error occurred in the stencil.

Remarks

If ATL_DEBUG_STENCILS is defined, the error is added to a collection of errors that can be written to a stream at a later time using CStencil::RenderErrors.

If ATL_DEBUG_STENCILS is not defined, the existence of the error is recorded, but the other error information is discarded.

The presence or absence of errors is recorded and available by calling CStencil::ParseSuccessful.

See Also

Reference

CStencil Class
CStencil::ParseSuccessful
CStencil::RenderErrors

Other Resources

CStencil Members

CStencil::AddRef

Implementation of **IUnknown::AddRef**.

```
STDMETHOD_(ULONG, AddRef)( );
```

Return Value

Always returns 1.

Remarks

CStencil does not implement lifetime control using reference counting.

See Also

Reference

CStencil Class

CStencil::Release

Other Resources

CStencil Members

CStencil::AddToken

Call this method to add a token to the collection managed by this object.

```
ATL_NOINLINE DWORD AddToken(
   LPCSTR pStart,
   LPCSTR pEnd,
   DWORD dwType,
   LPCSTR szHandlerName = NULL,
   LPCSTR szMethodName = NULL,
   DWORD dwFnOffset = STENCIL_INVALIDOFFSET,
   DWORD dwObjOffset = STENCIL_INVALIDOFFSET,
   DWORD_PTR dwData = 0,
   DWORD dwMap = 0,
   BOOL bDynamicAlloc = 0
) throw( );
```

Parameters

pStart

See StencilToken::pStart.

pEnd

See StencilToken::pEnd.

dwType

See StencilToken::type.

szHandlerName

See StencilToken::szHandlerName.

szMethodName

See StencilToken::szMethodName.

dwFnOffset

See StencilToken::dwFnOffset.

dwObjOffset

See StencilToken::dwObjOffset.

dwData

See StencilToken::dwData.

dwMap

See StencilToken::dwMap. Data for this member must be allocated using CStencil::m_pMemMgr and will be freed by CStencil::Uninitialize.

bDynamicAlloc

See StencilToken::bDynamicAlloc.

Return Value

Returns the index of the newly added token. This index can be passed to CStencil::GetToken to get a pointer to the newly added token.

Remarks

The information passed to this method is used to initialize a StencilToken structure and add it to the collection managed by this object.

The number of tokens managed by this object is available from CStencil::GetTokenCount and individual tokens are available by

calling CStencil::GetToken.

See Also

Reference

CStencil Class
CStencil::GetToken
CStencil::GetTokenCount

Other Resources

CStencil Members
StencilToken Members

CStencil::CStencil

The constructor.

```
CStencil(
   IAtlMemMgr * pMemMgr = NULL
) throw( );
```

Parameters

pMemMgr

The memory manager used for allocating and releasing memory used by this object.

Remarks

The CRT heap is used to allocate memory if no memory manager is passed to the constructor of this class. The pointer to the memory manager interface is stored in the CStencil::m_pMemMgr data member.

See Also

Reference

CStencil Class
IAtlMemMgr Class
CStencil::m_pMemMgr

Other Resources

CStencil Members

CStencil::~CStencil

The destructor.

```
~CStencil() throw( );
```

Remarks

Calls CStencil::Uninitialize.

See Also

Reference

CStencil Class

CStencil::Uninitialize

Other Resources

CStencil Members

CStencil::FinishParseReplacements

Override this method to enhance or replace the stencil parsing provided by this class.

```
virtual bool FinishParseReplacements( ) throw(...);
```

Remarks

This method is called after CStencil::ParseReplacements has built up an initial collection of stencil tokens.

This method checks that tokens of the following types have valid links to their corresponding tokens by checking StencilToken::dwLoopIndex:

- STENCIL_CONDITIONALSTART
- STENCIL_CONDITIONALELSE
- STENCIL_ITERATORSTART

If the tokens have invalid loop indexes, they are converted to STENCIL_TEXTTAG tokens and errors are logged using CStencil::AddError.

This method also verifies the existence of replacement methods using the tag replacer in CStencil::m_pReplacer. Tokens that do not have verifiable replacement methods are converted to STENCIL_TEXTTAG tokens and errors are logged using CStencil::AddError. Tokens that are linked to such tokens by StencilToken::dwLoopIndex are also converted to STENCIL_TEXTTAG tokens.

See Also

Reference

CStencil Class

Stencil Tokens

Other Resources

CStencil Members

CStencil::Free

Implementation of IMemoryCacheClient::Free.

```
STDMETHOD(Free)(
   const void * pData
);
```

Remarks

Calls CStencil::Uninitialize followed by **delete this**.

See Also

Reference

CStencil Class

IMemoryCacheClient::Free

Other Resources

CStencil Members

CStencil::GetBufferEnd

Call this method to get the pointer to the end of the buffer holding the stencil for this object.

```
LPCSTR GetBufferEnd( ) const throw( );
```

Return Value

Returns the pointer to the end of the buffer holding the stencil for this object or NULL if no stencil has been loaded.

Remarks

This method will return NULL until one of CStencil::LoadFromFile, CStencil::LoadFromResource, and CStencil::LoadFromString has been called.

See Also

Reference

CStencil Class

CStencil::GetBufferStart

Other Resources

CStencil Members

CStencil::GetBufferStart

Call this method to get the pointer to the start of the buffer holding the stencil for this object.

```
LPCSTR GetBufferStart( ) const throw( );
```

Return Value

Returns the pointer to the start of the buffer holding the stencil for this object or NULL if no stencil has been loaded.

Remarks

This method will return NULL until one of CStencil::LoadFromFile, CStencil::LoadFromResource, and CStencil::LoadFromString has been called.

See Also

Reference

CStencil Class

CStencil::GetBufferEnd

Other Resources

CStencil Members

CStencil::GetCacheItem

Call this method to get the handle identifying this stencil in a cache.

```
HCACHEITEM GetCacheItem( );
```

Return Value

The handle identifying this stencil in a cache.

Remarks

The handle can be set by calling CStencil::SetCacheItem.

See Also

Reference

CStencil Class
HCACHEITEM
CStencil::SetCacheItem

Other Resources

CStencil Members

CStencil::GetCodePage

Call this method to get the code page for the current stencil.

```
WORD GetCodePage( ) const throw( );
```

Return Value

Returns the code page for the current stencil.

Remarks

The code page is initialized to CP_ACP (the system default–ANSI code page) in the constructor.

If a codepage tag is encountered, the code page is set to that specified in the tag. This value can only be changed if another codepage tag is encountered.

If a locale tag is found, the code page is set to the default ANSI code page for the locale specified in the tag. This value can be changed by a subsequent locale tag or codepage tag.

If you derive from this class to extend the default stencil parsing, be sure to use the code page returned by this method to parse the stencil correctly.

See Also

Reference

CStencil Class
CStencil::ParseCodepage
CStencil::ParseLocale
Codepage Tag
Locale Tag

Other Resources

CStencil Members

CStencil::GetHandlerName

Call this method to get the DLL path and handler name of the request handler parsed from the handler tag of the currently loaded stencil.

```
void GetHandlerName(
   LPSTR szDllPath,
   LPSTR szHandlerName
) throw( );
```

Parameters

szDllPath

A pointer to a caller-allocated buffer of at least MAX_PATH + 1 characters that will receive the handler name on return of this method.

szHandlerName

A pointer to a caller-allocated buffer of at least ATL_MAX_HANDLER_NAME_LEN + 1 characters that will receive the handler name on return of this method.

Remarks

This method will not return useful information until a stencil has been loaded, CStencil::ParseReplacements has been called, and a handler tag has been encountered.

See Also

Reference

CStencil Class

Handler Tag

Other Resources

CStencil Members

CStencil::GetLastChecked

Call this method to get the date-time at which the last modified time of the stencil loaded by CStencil::LoadFromFile was last checked.

```
void GetLastChecked(
   FILETIME * pftLastChecked
) throw( );
```

Parameters

pftLastChecked

[out] Address of the variable that, on success, receives the date-time at which the last modified time of the stencil file was last checked.

Remarks

This method, along with CStencil::GetLastModified and CStencil::SetLastChecked, can be used to keep cached stencils up to date with changes made to the file on disk. See CStencil::GetLastModified for details.

The date-time returned by this method is set to 0 until CStencil::SetLastChecked is called.

See Also

Reference

CStencil Class
CStencil::GetLastModified
CStencil::SetLastChecked

Other Resources

CStencil Members

CStencil::GetLastModified

Call this method to get the date-time at which the stencil loaded by CStencil::LoadFromFile was last modified.

```
void GetLastModified(
   FILETIME * pftLastModified
) throw( );
```

Parameters

pftLastModified

[out] Address of the variable that, on success, receives the date-time at which the stencil file was last modified.

Remarks

This method, along with CStencil::GetLastChecked and CStencil::SetLastChecked, can be used to keep cached stencils up to date with changes made to the file on disk.

Assuming a **CStencil** object has been cached, the following algorithm can be used to obtain an up to date object that matches the stencil on disk without incurring the overhead of reading from disk on every request:

- When the stencil needs to be used, retrieve it from the cache and call CStencil::GetLastChecked
- If the difference between the current time and the last checked time is greater than some threshold, call CStencil::GetLastModified.
- If the last-modified time of the disk file does not match the value returned by CStencil::GetLastModified, discard the cached stencil and reload from disk.
- If the last-modified time of the disk file matches the value returned by CStencil::GetLastModified, call CStencil::SetLastChecked and use the stencil in the cache.

The date-time returned by this method is set to 0 if no file is loaded.

See Also

Reference

CStencil Class

CStencil::GetLastChecked

Other Resources

CStencil Members

CStencil::GetToken

Call this method to get a pointer to the specified StencilToken.

```
StencilToken* GetToken(
   DWORD dwIndex
) throw( );
const StencilToken* GetToken(
   DWORD dwIndex
) const throw( );
```

Parameters

dwIndex

The index of the stencil token in the collection managed by this object.

Return Value

Returns a pointer to the requested stencil token or NULL if the index is invalid.

See Also

Reference

CStencil Class

CStencil::GetTokenCount

Other Resources

CStencil Members

StencilToken Members

CStencil::GetTokenCount

Call this method to get the number of stencil tokens in the collection managed by this object.

```
DWORD GetTokenCount( ) const throw( );
```

Return Value

Returns the number of stencil tokens in the collection managed by this object.

See Also

Reference

CStencil Class

CStencil::GetToken

Other Resources

CStencil Members

CStencil::IsValidIndex

Call this method to determine whether a stencil token index is valid.

```
inline BOOL IsValidIndex(
   DWORD dwIndex
) const throw( );
```

Parameters

dwIndex

The index of a stencil token.

Return Value

TRUE if *dwIndex* is a valid index that could be passed to CStencil::GetToken; **FALSE** otherwise.

See Also

Reference

CStencil Class

CStencil::GetToken

CStencil::GetTokenCount

Other Resources

CStencil Members

CStencil::LoadFromFile

Call this method to load a stencil from a file.

```
HTTP_CODE LoadFromFile(
   LPCSTR szFileName
) throw( );
```

Parameters

szFileName

The name of the file from which to load the stencil.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

CStencil Class
HTTP_CODE
CStencil::LoadFromFile
CStencil::LoadFromResource
CStencil::LoadFromString

Other Resources

CStencil Members

CStencil::LoadFromResource

Call this method to load a stencil from a resource.

```
HTTP_CODE LoadFromResource(
   HINSTANCE hInstRes,
   UINT nId,
   LPCSTR szType = NULL
) throw( );
HTTP_CODE LoadFromResource(
   HINSTANCE hInstRes,
   LPCSTR szID,
   LPCSTR szType = NULL
) throw( );
```

Parameters

hInstRes

Handle to the module whose executable file contains the resource.

nId

The integer resource identifier.

szID

The string resource identifier.

szType

The type of resource. If not specified, the stencil is assumed to be in an **RT_HTML** resource.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

For more information on the parameters see FindResource.

See Also

Reference

CStencil Class
CStencil::LoadFromResourceEx
HTTP_CODE
CStencil::LoadFromFile
CStencil::LoadFromString

Other Resources

CStencil Members

CStencil::LoadFromResourceEx

Call this method to load a stencil from a resource associated with a particular language.

```
HTTP_CODE LoadFromResourceEx(
   HINSTANCE hInstRes,
   UINT nId,
   WORD wLanguage,
   LPCSTR szType = NULL
) throw( );
HTTP_CODE LoadFromResourceEx(
   HINSTANCE hInstRes,
   LPCSTR szID,
   WORD wLanguage,
   LPCSTR szType = NULL
) throw( );
```

Parameters

hInstRes

Handle to the module whose executable file contains the resource.

nId

The integer resource identifier.

szID

The string resource identifier.

wLanguage

The language of the resource.

szType

The type of resource. If not specified, the stencil is assumed to be in an **RT_HTML** resource.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

For more information on the parameters see FindResourceEx.

See Also

Reference

CStencil Class
CStencil::LoadFromResource
HTTP_CODE
CStencil::LoadFromFile
CStencil::LoadFromString

Other Resources

CStencil Members

CStencil::LoadFromString

Call this method to load a stencil from a string.

```
HTTP_CODE LoadFromString(
   LPCSTR szString,
   DWORD dwSize
) throw( );
```

Parameters

szString

The string containing the stencil data.

dwSize

The length in characters of the stencil data.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

The string passed in *szString* is not copied by this method. It is the caller's responsibility to ensure that the string is valid throughout the lifetime of the **CStencil** object.

See Also

Reference

CStencil Class
HTTP_CODE
CStencil::LoadFromFile
CStencil::LoadFromResource

Other Resources

CStencil Members

CStencil::Parse

Call this method to parse the loaded stencil.

```
virtual bool Parse(
   ITagReplacer * pReplacer
) throw(...);
```

Parameters

pReplacer

A pointer to the ITagReplacer interface of the object responsible for providing the replacement methods specified in the stencil being parsed.

Return Value

Returns **true** on success, **false** on failure.

Remarks

This method calls CStencil::ParseReplacements and, on success, CStencil::FinishParseReplacements. You can override either of these methods to customize the stencil parsing in a derived class.

See Also

Reference

CStencil Class
ITagReplacer Interface
CStencil::ParseReplacements
CStencil::FinishParseReplacements

Other Resources

CStencil Members

CStencil::ParseCodepage

Call this method to parse a codepage tag.

```
DWORD ParseCodepage(
   LPCSTR szTokenStart,
   LPCSTR szTokenEnd
) throw( );
```

Parameters

szTokenStart

The start of the token (after the braces and white space).

szTokenEnd

The end of the token.

Return Value

Returns STENCIL_INVALIDINDEX.

Remarks

On success, sets the code page returned by CStencil::GetCodePage.

See Also

Reference

CStencil Class
Codepage Tag
CStencil::GetCodePage

Other Resources

CStencil Members

CStencil::ParseElse

Call this method to parse an {{else}} replacement tag.

```
DWORD ParseElse(
   LPCSTR szTokenStart,
   LPCSTR szTokenEnd,
   DWORD * pBlockStack,
   DWORD * pdwTop
) throw( );
```

Parameters

szTokenStart

The start of the token (after the braces and white space).

szTokenEnd

The end of the token.

pBlockStack

An array of indices of tokens in the collection of stencil tokens managed by this object.

pdwTop

Pointer to an index into the *pBlockStack* array.

Return Value

Returns the index of the token added to the collection managed by this object, or STENCIL_INVALIDINDEX if an error occurs.

See Also

Reference

CStencil Class

Codepage Tag

Other Resources

CStencil Members

CStencil::ParseEndIf

Call this method to parse an {{endif}} replacement tag.

```
DWORD ParseEndIf(
   LPCSTR szTokenStart,
   LPCSTR szTokenEnd,
   DWORD * pBlockStack,
   DWORD * pdwTop
) throw( );
```

Parameters

szTokenStart

The start of the token (after the braces and white space).

szTokenEnd

The end of the token.

pBlockStack

An array of indices of tokens in the collection of stencil tokens managed by this object.

pdwTop

Pointer to an index into the *pBlockStack* array.

Return Value

Returns the index of the token added to the collection managed by this object, or STENCIL_INVALIDINDEX if an error occurs.

See Also

Reference

CStencil Class

Codepage Tag

Other Resources

CStencil Members

CStencil::ParseEndWhile

Call this method to parse an {{endwhile}} replacement tag.

```
DWORD ParseEndWhile(
   LPCSTR szTokenStart,
   LPCSTR szTokenEnd,
   DWORD * pBlockStack,
   DWORD * pdwTop
) throw( );
```

Parameters

szTokenStart

The start of the token (after the braces and white space).

szTokenEnd

The end of the token.

pBlockStack

An array of indices of tokens in the collection of stencil tokens managed by this object.

pdwTop

Pointer to an index into the *pBlockStack* array.

Return Value

Returns the index of the token added to the collection managed by this object, or STENCIL_INVALIDINDEX if an error occurs.

See Also

Reference

CStencil Class

Codepage Tag

Other Resources

CStencil Members

CStencil::ParseHandler

Call this method to parse a handler tag.

```
PARSE_TOKEN_RESULT ParseHandler(
   LPCSTR szTokenStart,
   LPCSTR szTokenEnd
) throw( );
```

Parameters

szTokenStart

The start of the token (after the braces and white space).

szTokenEnd

The end of the token.

Return Value

See CStencil::PARSE_TOKEN_RESULT for possible values and their meanings.

Remarks

Sets the DLL path and handler name returned by CStencil::GetHandlerName.

See Also

Reference

CStencil Class
CStencil::PARSE_TOKEN_RESULT
Handler Tag
CStencil::GetHandlerName

Other Resources

CStencil Members

CStencil::ParseIf

Call this method to parse an {{if}} replacement tag.

```
DWORD ParseIf(
   LPCSTR szTokenStart,
   LPCSTR szTokenEnd,
   DWORD * pBlockStack,
   DWORD * pdwTop
) throw( );
```

Parameters

szTokenStart

The start of the token (after the braces and white space).

szTokenEnd

The end of the token.

pBlockStack

An array of indices of tokens in the collection of stencil tokens managed by this object.

pdwTop

Pointer to an index into the *pBlockStack* array.

Return Value

Returns the index of the token added to the collection managed by this object, or STENCIL_INVALIDINDEX if an error occurs.

See Also

Reference

CStencil Class

Codepage Tag

Other Resources

CStencil Members

CStencil::ParseLocale

Call this method to parse a locale tag.

```
DWORD ParseLocale(
   LPCSTR szTokenStart,
   LPCSTR szTokenEnd
) throw( );
```

Parameters

szTokenStart

The start of the token (after the braces and white space).

szTokenEnd

The end of the token.

Return Value

Returns the index of the token added to the collection managed by this object, or STENCIL_INVALIDINDEX if an error occurs.

Remarks

See CStencil::GetCodePage for details of the effect of a locale tag on the code page.

See Also

Reference

CStencil Class
Locale Tag
CStencil::GetCodePage

Other Resources

CStencil Members

CStencil::ParseReplacement

Call this method to parse a replacement tag.

```
DWORD ParseReplacement(
   LPCSTR szTokenStart,
   LPCSTR szTokenEnd,
   DWORD dwTokenType = STENCIL_REPLACEMENT,
   DWORD dwKeywordLen = 0
) throw( );
```

Parameters

szTokenStart

The start of the token (after the braces and white space).

szTokenEnd

The end of the token.

dwTokenType

The type of stencil token.

dwKeywordLen

The length of the keyword associated with this token.

Return Value

Returns the index of the token added to the collection managed by this object, or STENCIL_INVALIDINDEX if an error occurs.

See Also

Reference

CStencil Class
Codepage Tag
Stencil Tokens

Other Resources

CStencil Members

CStencil::ParseReplacements

Override this method to enhance or replace the stencil parsing provided by this class.

```
virtual bool ParseReplacements(
   ITagReplacer* pReplacer
) throw(...);
```

Parameters

pReplacer

A pointer to the ITagReplacer interface of the object responsible for providing the replacement methods specified in the stencil being parsed.

Return Value

Returns **true** on success, **false** on failure.

Remarks

This method is responsible for parsing the stencil into tokens.

The start and end of the stencil are obtained from CStencil::GetBufferStart and CStencil::GetBufferEnd.

The **CStencil** implementation of this method parses the stencil looking for blocks enclosed by double braces {{ }}. Areas outside of such blocks are added to the collection of stencil tokens as text. When such blocks are found, the start and end points are passed to CStencil::ParseToken for further parsing.

If ParseToken returns INVALID_TOKEN, ParseReplacements adds the token to the collection of stencil tokens as text. Otherwise ParseToken is assumed to have taken any actions necessary for that token.

If ParseToken returns RESERVED_TOKEN, any line break immediately following the token is ignored.

See Also

Reference

CStencil Class
ITagReplacer Interface
CStencil::Parse
CStencil::FinishParseReplacements
CStencil::GetToken

Other Resources

CStencil Members

CStencil::ParseSuccessful

Call this method to determine whether stencil parsing was successful.

```
bool ParseSuccessful( );
```

Return Value

Returns **true** if stencil parsing was successful, **false** otherwise.

See Also

Reference

CStencil Class

CStencil::AddError

Other Resources

CStencil Members

CStencil::ParseToken

Override this method to parse a token delimited by double braces ({{}}).

```
virtual PARSE_TOKEN_RESULT ParseToken(
   LPCSTR szTokenStart,
   LPCSTR szTokenEnd,
   DWORD * pBlockStack,
   DWORD * pdwTop
);
```

Parameters

szTokenStart

The start of the token.

szTokenEnd

The end of the token.

pBlockStack

An array of indices of tokens in the collection of stencil tokens managed by this object.

pdwTop

Pointer to an index into the *pBlockStack* array.

Return Value

See CStencil::PARSE_TOKEN_RESULT for possible values and their meanings. See also CStencil::ParseReplacements.

Remarks

This method is responsible for parsing an individual token. Overrides should handle their custom tokens before delegating to the base class for tokens not recognized by the derived implementation. On success, overrides of this method will typically add the token to the collection of stencil tokens by calling CStencil::AddToken. On failure, overrides will typically report errors by calling CStencil::AddError.

The implementation in this class handles the following tokens:

- Codepage Tag
- Handler Tag
- Locale Tag
- Replacement Tag

Note

Successfully parsed codepage tags and handler tags do not cause a stencil token to be added to the collection of tokens man aged by **CStencil**. Codepage tags affect the code page returned by CStencil::GetCodePage. Handler tags affect the informatio n provided by CStencil::GetHandlerName.

See Also

Reference

CStencil Class
CStencil::PARSE_TOKEN_RESULT
CStencil::ParseReplacements
CStencil::AddToken
CStencil::AddError
CHtmlStencil::ParseToken

Other Resources

CStencil Members

CStencil::ParseWhile

Call this method to parse a {{while}} replacement tag.

```
DWORD ParseWhile(
   LPCSTR szTokenStart,
   LPCSTR szTokenEnd,
   DWORD * pBlockStack,
   DWORD * pdwTop
) throw( );
```

Parameters

szTokenStart

The start of the token (after the braces and white space).

szTokenEnd

The end of the token.

pBlockStack

An array of indices of tokens in the collection of stencil tokens managed by this object.

pdwTop

Pointer to an index into the *pBlockStack* array.

Return Value

Returns the index of the token added to the collection managed by this object, or STENCIL_INVALIDINDEX if an error occurs.

See Also

Reference

CStencil Class

Codepage Tag

Other Resources

CStencil Members

CStencil::QueryInterface

Implementation of **IUnknown::QueryInterface**.

```
STDMETHOD(QueryInterface )(
   REFIID riid,
   void ** ppv
);
```

Remarks

Objects of this class can be successfully queried for the **IUnknown** and IMemoryCacheClient interfaces.

See Also

Reference

CStencil Class

Other Resources

CStencil Members

CStencil::Release

Implementation of **IUnknown::Release**.

```
STDMETHOD_(ULONG, Release)( );
```

Return Value

Always returns 1.

Remarks

CStencil does not implement lifetime control using reference counting.

See Also

Reference

CStencil Class

CStencil::AddRef

Other Resources

CStencil Members

CStencil::Render

Call this method to transform the stencil into a response stream using an object implementing ITagReplacer to replace tokens with dynamic content.

```
virtual HTTP_CODE Render(
   ITagReplacer* pReplacer,
   IWriteStream* pWriteStream,
   CStencilState* pState = NULL
) const throw(...);
```

Parameters

pReplacer

The object providing access to the replacement methods referred to in the stencil.

pWriteStream

The stream to which the response should be written.

pState

A pointer to state information to be used by the stencil.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

This method sets the locale of the thread using the locale member of *pState* if specified. (The thread locale may also be set if a locale tag is encountered during CStencil::RenderToken). The original thread locale is restored when the method returns.

This method calls **CStencil::RenderToken** in a loop to render each token. The first token rendered is either the token identified by pState->dwIndex or the first token in the collection. Subsequent tokens are identified by the value returned by **CStencil::RenderToken**. The loop terminates when **RenderToken** returns STENCIL_INVALIDINDEX or sets the HTTP_CODE status code to something other than HTTP_SUCCESS.

If the HTTP status code is for asynchronous operation (see IsAsyncStatus), pState->dwIndex is set to the last index returned by **RenderToken**.

CStencil::Render returns the last HTTP_CODE returned by **CStencil::RenderToken**.

See Also

Reference

CStencil Class
ITagReplacer Interface
IWriteStream Interface
CStencil::RenderToken

Other Resources

CStencil Members
CStencilState Members

CStencil::RenderErrors

Call this method to write parse errors to a stream.

```
bool RenderErrors(
   IWriteStream * pStream
) throw(...);
```

Parameters

pStream

The stream to which the response should be written.

Return Value

Returns **true** on success, **false** on failure.

Remarks

This method is only available if ATL_DEBUG_STENCILS is defined. This method writes the error information to the stream using a simple HTML table which displays a description of the error, a line number, and the corresponding line of the stencil.

See Also

Reference

CStencil Class

IWriteStream Interface

Other Resources

CStencil Members

CStencil::RenderToken

Override this method to customize the rendering of a stencil token.

```
virtual DWORD RenderToken(
   DWORD dwIndex,
   ITagReplacer * pReplacer,
   IWriteStream * pWriteStream,
   HTTP_CODE * phcErrorCode,
   CStencilState* pState = NULL
) const throw(...);
```

Parameters

dwIndex

The index of the stencil token to be rendered in the collection managed by this object.

pReplacer

The object providing access to the replacement methods referred to in the stencil.

pWriteStream

The stream to which the response should be written.

phcErrorCode

[out] Address of the pointer variable that, on success, receives the ATL Server status code indicating success or failure.

pState

A pointer to state information to be used by the stencil.

Return Value

Returns the index of the next stencil token to be rendered in the collection managed by this object, or STENCIL_INVALIDINDEX if no more tokens should be rendered (as a result of an error or because the stencil is complete).

Remarks

If you need to provide custom actions for your own stencil tokens, you can override this method and take the necessary steps to produce output or configure the environment in which stencil rendering takes place.

The **CStencil** implementation of this method handles the following stencil tokens:

- STENCIL_CONDITIONALELSE
- STENCIL_CONDITIONALEND
- STENCIL_CONDITIONALSTART
- STENCIL_ITERATOREND
- STENCIL_ITERATORSTART
- STENCIL_LOCALE
- STENCIL_REPLACEMENT
- STENCIL_TEXTTAG

Tokens of any other type are treated as errors.

See Also

Reference

CStencil Class
ITagReplacer Interface
IWriteStream Interface
HTTP_CODE

Other Resources

CStencil Members

CStencil::SetCacheItem

Call this method to set the handle identifying this stencil in a cache.

```
void SetCacheItem(
   HCACHEITEM hCacheItem
);
```

Parameters

hCacheItem

The handle identifying this stencil in a cache.

Remarks

The handle can be retrieved by calling CStencil::GetCacheItem.

See Also

Reference

CStencil Class

HCACHEITEM

CStencil::GetCacheItem

Other Resources

CStencil Members

CStencil::SetErrorResource

Call this method to set the handle of the module supplying the resources used for error strings.

```
void SetErrorResource(
   HINSTANCE hResInst
);
```

Parameters

hResInst

The handle of the module supplying the resources used for error strings.

Remarks

If ATL_DEBUG_STENCILS is defined, the handle is stored and used to retrieve error string resources.

If ATL_DEBUG_STENCILS is not defined, the handle is discarded.

See Also

Reference

CStencil Class
CStencil::AddError
ATL_DEBUG_STENCILS

Other Resources

CStencil Members

CStencil::SetLastChecked

Call this method to set the date-time at which the last modified time of the stencil loaded by CStencil::LoadFromFile was last checked.

```
void SetLastChecked(
   FILETIME * pftLastChecked
) throw( );
```

Parameters

pftLastChecked

The date-time at which the last modified time of the stencil file was last checked.

Remarks

This method, along with CStencil::GetLastModified and CStencil::GetLastChecked, can be used to keep cached stencils up to date with changes made to the file on disk. See CStencil::GetLastModified for details.

See Also

Reference

CStencil Class

CStencil::GetLastChecked

Other Resources

CStencil Members

CStencil::Uninitialize

Call this method if you need to reuse this object with another stencil or before destruction.

```
void Uninitialize( ) throw( );
```

Remarks

It is not necessary to call this method explicitly unless you wish to reuse the object with another stencil. This method is called from the destructor.

See Also

Reference

CStencil Class

Other Resources

CStencil Members

CStencil Data Members

For information about the data members in **CStencil**, see CStencil Members.

CStencil::m_pMemMgr

The memory manager.

```
IAtlMemMgr * m_pMemMgr;
```

Remarks

The memory manager can be passed to the constructor of this class. If no memory manager is passed to the constructor, this pointer is initialized to use the CRT heap provided by CStencil::m_crtHeap.

If you create custom tokens that use the StencilToken::dwData member, you should allocate the memory using this memory manager. It will be freed automatically in CStencil::Uninitialize.

See Also

Reference

CStencil Class
IAtlMemMgr Class
CStencil::CStencil
CStencil::m_crtHeap

Other Resources

CStencil Members

CStencil::m_pReplacer

The tag replacer used to provide the dynamic content for the replacement tags in the current stencil.

```
ITagReplacer * m_pReplacer;
```

Remarks

This member is set by calling CStencil::Parse.

See Also

Reference

CStencil Class
ITagReplacer Interface
CStencil::Parse

Other Resources

CStencil Members

CStencil Static Data Members

For information about the static data members in **CStencil**, see CStencil Members.

CStencil::m_crtHeap

The CRT heap is used to allocate memory if no memory manager is passed to the constructor of this class.

```
static CCRTHeap m_crtHeap;
```

Remarks

CStencil::m_pMemMgr is initialized to point to this object if no memory manager is passed to the constructor of this class.

See Also

Reference

CStencil Class
CCRTHeap Class
CStencil::CStencil

Other Resources

CStencil Members

Enumerations (ATL Server)

For information about the enumerations in **CStencil**, see CStencil Members.

CStencil::PARSE_TOKEN_RESULT

Used to describe the result of stencil processing and indicate whether a line break following a token should be ignored.

```
enum PARSE_TOKEN_RESULT
{
   INVALID_TOKEN,
   NORMAL_TOKEN,
   RESERVED_TOKEN
};
```

Remarks

The table describes each of the values in this enumeration:

Value	Description
INVALID_TOKEN	An error occurred. Treat the token being processed like a text token.
NORMAL_TOKEN	A token was successfully parsed.
RESERVED_TOKEN	A token was successfully parsed. If it is followed immediately by a new line ('\n' or '\r\n'), those characters should be discarded.

See Also

Reference

CStencil Class

Other Resources

CStencil Members

CStencilCache Class

This class implements a stencil cache, and is used by ATL Server to cache pre-processed stencils to speed response processing.

```
template <
   class MonitorClass,
   class StatClass  = CStdStatClass,
   class SyncClass  = CComCriticalSection,
   class FlushClass = COldFlusher,
   class CullClass  = CLifetimeCuller
>
class CStencilCache :
   public CMemoryCacheBase<CStencilCache, void * , CCacheDataEx,
                     CFixedStringKey,
                     CStringElementTraitsI< CFixedStringKey >,
                                    FlushClass, CullClass, SyncClass, StatClass >,
   public IStencilCache,
   public IStencilCacheControl,
   public IWorkerThreadClient,
   public IMemoryCacheStats,
   public CComObjectRootEx< CComGlobalsThreadModel >
```

Parameters

MonitorClass

The class that is to trigger periodic cache maintenance, usually CWorkerThread. The CNoWorkerThread class can be used if no automatic cache maintenance is desired.

StatClass

The class used to collect and expose statistics related to the use of the cache including effectiveness and memory usage information. Must conform to the cache statistics archetype.

SyncClass

The class to be used for thread synchronization. The CComCriticalSection class is used by default, but the CComFakeCriticalSection class can be used to avoid thread safety overhead if the cache resides in a single-threaded environment.

FlushClass

The class that determines which cache items, if any, will be flushed from the cache when the cache item count limit or memory use limit is reached. By default, the COldFlusher class is used, which selects the oldest items for removal.

CullClass

The class that determines if and when cache entries are removed from the cache based on expiration time.

Remarks

Flushing classes, other than the default COldFlusher class, include:

- CLOUFlusher (least often used).
- CLRUFlusher (least recently used).
- CNoFlusher (performs no cache flushing).

The COrFlushers class can be used to combine two flushing mechanisms, or even to build a hierarchy of flusher classes. Alternatively, custom flushing classes can be used, so long as they match the Cache Flusher Archetype.

Regardless of the *FlushClass* used, items will not be flushed until the cache item count or memory use limits have been reached. By default, these settings are both limited only by the size of the **DWORD** type. Use the IMemoryCacheStats interface to adjust these cache limits and activate the flushing mechanisms.

Requirements

Header: atlcache.h

See Also

Tasks

Hotswap Sample: Replacing ATL Server Application DLLs at Run Time
BlobCache Sample: Generic Memory Caching
StencilCache Sample: Uses the ATL Server Stencil Cache

Reference

CMemoryCacheBase Class
IStencilCache Interface
IStencilCacheControl Interface
IWorkerThreadClient Interface
CComObjectRootEx Class

Other Resources

CStencilCache Members
Caching
Caching Reference

CStencilCache Members

Methods

AddRefStencil	Implementation of IStencilCache::AddRefStencil.
CacheStencil	Implementation of IStencilCache::CacheStencil.
ClearStats	Implementation of IMemoryCacheStats::ClearStats.
CloseHandle	Implementation of IWorkerThreadClient::CloseHandle.
CStencilCache	The constructor.
~CStencilCache	The destructor.
Execute	Implementation of IWorkerThreadClient::Execute.
GetCurrentAllocSize	Implementation of IMemoryCacheStats::GetCurrentAllocSize.
GetCurrentEntryCount	Implementation of IMemoryCacheStats::GetCurrentEntryCount.
GetDefaultLifespan	Implementation of IStencilCacheControl::GetDefaultLifespan.
GetHitCount	Implementation of IMemoryCacheStats::GetHitCount.
GetMaxAllocSize	Implementation of IMemoryCacheStats::GetMaxAllocSize.
GetMaxEntryCount	Implementation of IMemoryCacheStats::GetMaxEntryCount.
GetMissCount	Implementation of IMemoryCacheStats::GetMissCount.
GetStencil	Implementation of IStencilCache::GetStencil.
Initialize	Call this method to initialize the stencil cache.
LookupStencil	Implementation of IStencilCache::LookupStencil.
OnDestroyEntry	This method is called when a cached stencil is to be destroyed.
ReleaseStencil	Implementation of IStencilCache::ReleaseStencil.
RemoveAllStencils	Implementation of IStencilCacheControl::RemoveAllStencils.
RemoveStencil	Implementation of IStencilCacheControl::RemoveStencil.
RemoveStencilByName	Implementation of IStencilCacheControl::RemoveStencilByName.
SetDefaultLifespan	Implementation of IStencilCacheControl::SetDefaultLifespan.
Uninitialize	Call this method to uninitialize the stencil cache.

Data Members

m_dwdwStencilLifespan	The default stencil life span.
m_hTimer	The timer handle used to activate the monitor class.
m_Monitor	The class used to perform periodic cache maintenance.
m_spDllCache	A pointer to the IDllCache interface.

Typedefs

cacheBase	A synonym for the CStencilCache base class.

See Also

Reference

CStencilCache Class

Other Resources

Caching

Caching Reference

CStencilCache Methods

For information about the methods in **CStencilCache**, see CStencilCache Members.

CStencilCache::AddRefStencil

Implementation of IStencilCache::AddRefStencil.

```
STDMETHOD(AddRefStencil )(
   const HCACHEITEM hStencil
);
```

See Also

Reference

CStencilCache Class

IStencilCache Interface

Other Resources

CStencilCache Members

Caching Reference

Caching

CStencilCache::CacheStencil

Implementation of IStencilCache::CacheStencil.

```
STDMETHOD(CacheStencil )(
   LPCSTR szName,
   void* pStencil,
   DWORD dwSize,
   HCACHEITEM* phEntry,
   HINSTANCE hInstance,
   IMemoryCacheClient* pClient
);
```

See Also

Reference

CStencilCache Class

IStencilCache Interface

Other Resources

CStencilCache Members

Caching Reference

Caching

CStencilCache::ClearStats

Implementation of IMemoryCacheStats::ClearStats.

```
HRESULT STDMETHODCALLTYPE ClearStats( );
```

See Also

Reference

CStencilCache Class

IMemoryCacheStats Interface

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::CloseHandle

Implementation of IWorkerThreadClient::CloseHandle.

```
HRESULT CloseHandle(
   HANDLE hObject
);
```

See Also

Reference

CStencilCache Class

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::CStencilCache

The constructor.

```
CStencilCache( );
```

Remarks

Does nothing. The cache must be initialized using Initialize before other methods are called.

See Also

Reference

CStencilCache Class

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::~CStencilCache

The destructor.

```
~CStencilCache( );
```

Remarks

The Uninitialize method performs most of the termination tasks and should be called before the cache object is destroyed.

See Also

Reference

CStencilCache Class

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::Execute

Implementation of IWorkerThreadClient::Execute.

```
HRESULT Execute(
   DWORD_PTR dwParam,
   HANDLE /* hObject */
);
```

Remarks

The **Execute** method performs cache maintenance. The effects on the cache vary depending on the flushing and culling template parameters used to construct the **CStencilCache** object, and the settings specified through the IStencilCacheControl interface.

If CNoWorkerThread was used as the *MonitorClass* template argument during construction, the **Execute** method will not be called.

See Also

Reference

CStencilCache Class

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::GetCurrentAllocSize

Implementation of IMemoryCacheStats::GetCurrentAllocSize.

```
HRESULT STDMETHODCALLTYPE GetCurrentAllocSize(
   DWORD* pdwSize
);
```

Example

See the StencilCache Sample.

See Also

Reference

CStencilCache Class

IMemoryCacheStats Interface

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::GetCurrentEntryCount

Implementation of IMemoryCacheStats::GetCurrentEntryCount.

```
HRESULT STDMETHODCALLTYPE GetCurrentEntryCount(
   DWORD* pdwSize
);
```

Example

See the BlobCache Sample and the StencilCache Sample.

See Also

Reference

CStencilCache Class

IMemoryCacheStats Interface

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::GetDefaultLifespan

Implementation of IStencilCacheControl::GetDefaultLifespan.

```
STDMETHOD(GetDefaultLifespan )(
   unsigned __int64* pdwdwLifepsan
);
```

Example

See the StencilCache Sample.

See Also

Reference

CStencilCache Class

IStencilCacheControl Interface

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::GetHitCount

Implementation of IMemoryCacheStats::GetHitCount.

```
HRESULT STDMETHODCALLTYPE GetHitCount(
   DWORD* pdwSize
);
```

Example

See the BlobCache Sample and the StencilCache Sample.

See Also

Reference

CStencilCache Class

IMemoryCacheStats Interface

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::GetMaxAllocSize

Implementation of IMemoryCacheStats::GetMaxAllocSize.

```
HRESULT STDMETHODCALLTYPE GetMaxAllocSize(
   DWORD* pdwSize
);
```

Example

See the StencilCache Sample.

See Also

Reference

CStencilCache Class

IMemoryCacheStats Interface

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::GetMaxEntryCount

Implementation of IMemoryCacheStats::GetMaxEntryCount.

```
HRESULT STDMETHODCALLTYPE GetMaxEntryCount(
   DWORD* pdwSize
);
```

Example

See the BlobCache Sample and the StencilCache Sample.

See Also

Reference

CStencilCache Class

IMemoryCacheStats Interface

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::GetMissCount

Implementation of IMemoryCacheStats::GetMissCount.

```
HRESULT STDMETHODCALLTYPE GetMissCount(
   DWORD* pdwSize
);
```

Example

See the BlobCache Sample and the StencilCache Sample.

See Also

Reference

CStencilCache Class

IMemoryCacheStats Interface

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::GetStencil

Implementation of IStencilCache::GetStencil.

```
STDMETHOD(GetStencil )(
   const HCACHEITEM hStencil,
   void ** pStencil
) const;
```

See Also

Reference

CStencilCache Class

IStencilCache Interface

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::Initialize

Call this method to initialize the stencil cache.

```
HRESULT Initialize(
   IServiceProvider* pProv,
   DWORD dwStencilCacheTimeout = ATL_STENCIL_CACHE_TIMEOUT,
   __int64 dwdwStencilLifespan = ATL_STENCIL_LIFESPAN
);
template < class ThreadTraits >
HRESULT Initialize(
   IServiceProvider * pProv,
   CWorkerThread< ThreadTraits > * pWorkerThread,
   DWORD dwStencilCacheTimeout = ATL_STENCIL_CACHE_TIMEOUT,
   __int64 dwdwStencilLifespan = ATL_STENCIL_LIFESPAN
);
```

Parameters

pProv

The address of an IServiceProvider implementation object. This object is used to query for an IDllCache implementation that is used to ensure that DLLs upon which cached stencils rely are not unloaded prematurely.

dwStencilCacheTimeout

The frequency at which the cache is to be maintained by the *MonitorClass* as specified during construction. This value is in milliseconds, and is defined as ATL_STENCIL_CACHE_TIMEOUT by default.

dwdwStencilLifespan

The time before a stencil expires. The ATL_STENCIL_LIFESPAN macro is used for the default stencil life span. If the CNoFlusher class is used for the *FlushClass* template argument during construction, this value is ignored.

pWorkerThread

The address of a worker thread that is to be used to trigger cache maintenance. If none is provided, a new thread will be created, assuming that the CWorkerThread class was used for the *MonitorClass* template argument during construction.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

See Also

Reference

CStencilCache Class

CStencilCache::Uninitialize

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::LookupStencil

Implementation of IStencilCache::LookupStencil.

```
STDMETHOD(LookupStencil )(
   LPCSTR szName,
   HCACHEITEM* phStencil
);
```

See Also

Reference

CStencilCache Class

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::OnDestroyEntry

This method is called when a cached stencil is to be destroyed.

```
virtual void OnDestroyEntry(
   const NodeType* pEntry
);
```

Parameters

pEntry

The address of the node that stores the stencil to be destroyed.

Remarks

Called by the CMemoryCache base class to destroy cached items. The CStencilCache class implements this method, so it is not necessary to write or call this method from application code.

See Also

Reference

CStencilCache Class

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::ReleaseStencil

Implementation of IStencilCache::ReleaseStencil.

```
STDMETHOD(ReleaseStencil )(
   const HCACHEITEM hStencil
);
```

See Also

Reference

CStencilCache Class

IStencilCache Interface

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::RemoveAllStencils

Implementation of IStencilCacheControl::RemoveAllStencils.

```
STDMETHOD( RemoveAllStencils )( );
```

Example

See the Hotswap Sample.

See Also

Reference

CStencilCache Class

IStencilCacheControl Interface

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::RemoveStencil

Implementation of IStencilCacheControl::RemoveStencil.

```
STDMETHOD(RemoveStencil )(
   const HCACHEITEM hStencil
);
```

See Also

Reference

CStencilCache Class

IStencilCacheControl Interface

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::RemoveStencilByName

Implementation of IStencilCacheControl::RemoveStencilByName.

```
STDMETHOD(RemoveStencilByName )(
   LPCSTR szStencil
);
```

See Also

Reference

CStencilCache Class

IStencilCacheControl Interface

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::SetDefaultLifespan

Implementation of IStencilCacheControl::SetDefaultLifespan.

```
STDMETHOD(SetDefaultLifespan )(
   unsigned __int64 dwdwLifespan
);
```

Example

See the StencilCache Sample.

See Also

Reference

CStencilCache Class

IStencilCacheControl Interface

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::Uninitialize

Call this method to uninitialize the stencil cache.

```
HRESULT Uninitialize( );
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

This method should be called after the stencils within are no longer needed, and before the cache object is deleted.

See Also

Reference

CStencilCache Class

CStencilCache::Initialize

Other Resources

CStencilCache Members

Caching Reference

CStencilCache Data Members

For information about the data members in **CStencilCache**, see CStencilCache Members.

CStencilCache::m_dwdwStencilLifespan

The default stencil life span.

```
unsigned __int64 m_dwdwStencilLifespan;
```

Remarks

This data member, which defaults to ATL_STENCIL_LIFESPAN, is set by the Initialize method and is used to specify the life span for incoming stencils.

See Also

Reference

CStencilCache Class

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::m_hTimer

The timer handle used to activate the monitor class.

```
HANDLE m_hTimer;
```

Remarks

The **m_hTimer** member is used to store a timer handle that is used to trigger asynchronous cache maintenance.

See Also

Reference

CStencilCache Class

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::m_Monitor

The class used to perform periodic cache maintenance.

```
MonitorClass m_Monitor;
```

Remarks

The type of monitor class is dependent on the *MonitorClass* template argument used during construction, but is usually CWorkerThread. This class periodically performs cache maintenance by calling the Execute method.

See Also

Reference

CStencilCache Class

Other Resources

CStencilCache Members

Caching Reference

CStencilCache::m_spDllCache

A pointer to the IDllCache interface.

```
CComPtr<IDllCache> m_spDllCache;
```

Remarks

This data member is used to store a pointer to a DLL cache as retrieved from an IServiceProvider interface (which in turn was passed to the Initialize member). The DLL cache is used to prevent DLLs from being unloaded while dependant stencils are in the cache.

See Also

Reference

CStencilCache Class

Other Resources

CStencilCache Members

Caching Reference

CStencilCache Typedefs

For information about the typedefs in **CStencilCache**, see CStencilCache Members.

CStencilCache::cacheBase

A synonym for the CStencilCache base class.

```
typedef CMemoryCacheBase<CStencilCache,void*,CCacheDataEx,CFixedStringKey,CStringElementTra
itsI<CFixedStringKey>,FlushClass,CullClass,SyncClass,StatClass> cacheBase;
```

Remarks

This typedef can be used to invoke base class members without resorting to providing the complete and lengthy base class type.

See Also

Reference

CStencilCache Class

Other Resources

CStencilCache Members

Caching Reference

CStencilCacheManager Class

This class implements the XML Web service that allows you to manage the stencil cache in an ATL Server ISAPI extension.

```
[
   soap_handler(
      name = ID_STENCILCACHEMGR_WEBSERVICE_NAME,
      namespace = ID_STENCILCACHEMGR_WEBSERVICE_URL,
      protocol = "soap"
   ),
   request_handler(
      name = ID_STENCILCACHEMGR_WEBSERVICE_NAME,
      sdl = ID_STENCILCACHEMGR_WEBSERVICE_WSDL
   )
]
class CStencilCacheManager :
   public IStencilCacheMgr
```

Remarks

See Extension Management Services for more details.

Requirements

Header: atlextmgmt.h

Example

See the Hotswap Sample.

See Also

Reference

CSoapHandler Class
IStencilCacheMgr Interface
soap_handler
request_handler
ID_STENCILCACHEMGR_WEBSERVICE_NAME
ID_STENCILCACHEMGR_WEBSERVICE_URL
ID_STENCILCACHEMGR_WEBSERVICE_WSDL

Other Resources

CStencilCacheManager Members

CStencilCacheManager Members

Methods

ClearStats	Implementation of IStencilCacheMgr::ClearStats.
GetCurrentAllocSize	Implementation of IStencilCacheMgr::GetCurrentAllocSize.
GetCurrentEntryCount	Implementation of IStencilCacheMgr::GetCurrentEntryCount.
GetDefaultLifespan	Implementation of IStencilCacheMgr::GetDefaultLifespan.
GetHitCount	Implementation of IStencilCacheMgr::GetHitCount.
GetMaxAllocSize	Implementation of IStencilCacheMgr::GetMaxAllocSize.
GetMaxEntryCount	Implementation of IStencilCacheMgr::GetMaxEntryCount.
GetMissCount	Implementation of IStencilCacheMgr::GetMissCount.
HandleRequest	This method handles requests sent to the stencil cache manager service, including authorizing the client.
RemoveAllStencils	Implementation of IStencilCacheMgr::RemoveAllStencils.
RemoveStencil	Implementation of IStencilCacheMgr::RemoveStencil.
RemoveStencilByName	Implementation of IStencilCacheMgr::RemoveStencilByName.
SetDefaultLifespan	Implementation of IStencilCacheMgr::SetDefaultLifespan.

See Also

Reference

CStencilCacheManager Class

CStencilCacheManager Methods

For information about the methods in **CStencilCacheManager**, see CStencilCacheManager Members

CStencilCacheManager::ClearStats

Implementation of IStencilCacheMgr::ClearStats.

```
[ soap_method ]
STDMETHOD( ClearStats )( );
```

See Also

Reference

CStencilCacheManager Class

Other Resources

CStencilCacheManager Members

CStencilCacheManager::GetCurrentAllocSize

Implementation of IStencilCacheMgr::GetCurrentAllocSize.

```
[ soap_method ]
STDMETHOD(GetCurrentAllocSize )(
   __int64* pdwSize
);
```

See Also

Reference

CStencilCacheManager Class

Other Resources

CStencilCacheManager Members

CStencilCacheManager::GetCurrentEntryCount

Implementation of IStencilCacheMgr::GetCurrentEntryCount.

```
[ soap_method ]
STDMETHOD(GetCurrentEntryCount )(
   __int64* pdwSize
);
```

See Also

Reference

CStencilCacheManager Class

Other Resources

CStencilCacheManager Members

CStencilCacheManager::GetDefaultLifespan

Implementation of IStencilCacheMgr::GetDefaultLifespan.

```
[ soap_method ]
STDMETHOD(GetDefaultLifespan )(
   unsigned __int64* pdwdwLifespan
);
```

See Also

Reference

CStencilCacheManager Class

Other Resources

CStencilCacheManager Members

CStencilCacheManager::GetHitCount

Implementation of IStencilCacheMgr::GetHitCount.

```
[ soap_method ]
STDMETHOD(GetHitCount )(
   __int64* pdwSize
);
```

See Also

Reference

CStencilCacheManager Class

Other Resources

CStencilCacheManager Members

CStencilCacheManager::GetMaxAllocSize

Implementation of IStencilCacheMgr::GetMaxAllocSize.

```
[ soap_method ]
STDMETHOD(GetMaxAllocSize )(
   __int64* pdwSize
);
```

See Also

Reference

CStencilCacheManager Class

Other Resources

CStencilCacheManager Members

CStencilCacheManager::GetMaxEntryCount

Implementation of IStencilCacheMgr::GetMaxEntryCount.

```
[ soap_method ]
STDMETHOD(GetMaxEntryCount )(
   __int64* pdwSize
);
```

See Also

Reference

CStencilCacheManager Class

Other Resources

CStencilCacheManager Members

CStencilCacheManager::GetMissCount

Implementation of IStencilCacheMgr::GetMissCount.

```
[ soap_method ]
STDMETHOD(GetMissCount )(
   __int64* pdwSize
);
```

See Also

Reference

CStencilCacheManager Class

Other Resources

CStencilCacheManager Members

CStencilCacheManager::HandleRequest

This method handles requests sent to the stencil cache manager service, including authorizing the client.

```
HTTP_CODE HandleRequest(
   AtlServerRequest* pRequestInfo,
   IServiceProvider* pProvider
);
```

Parameters

pRequestInfo

The information about the request to be handled.

pProvider

The service provider.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

CStencilCacheManager Class

Other Resources

CStencilCacheManager Members

CStencilCacheManager::RemoveAllStencils

Implementation of IStencilCacheMgr::RemoveAllStencils.

```
[ soap_method ]
STDMETHOD( RemoveAllStencils )( );
```

Example

See the Hotswap Sample.

See Also

Reference

CStencilCacheManager Class

Other Resources

CStencilCacheManager Members

CStencilCacheManager::RemoveStencil

Implementation of IStencilCacheMgr::RemoveStencil.

```
[ soap_method ]
STDMETHOD(RemoveStencil )(
   __int64 hStencil
);
```

See Also

Reference

CStencilCacheManager Class

Other Resources

CStencilCacheManager Members

CStencilCacheManager::RemoveStencilByName

Implementation of IStencilCacheMgr::RemoveStencilByName.

```
[ soap_method ]
STDMETHOD(RemoveStencilByName )(
   BSTR bstrStencil
);
```

See Also

Reference

CStencilCacheManager Class

Other Resources

CStencilCacheManager Members

CStencilCacheManager::SetDefaultLifespan

Implementation of IStencilCacheMgr::SetDefaultLifespan.

```
[ soap_method ]
STDMETHOD(SetDefaultLifespan )(
   unsigned __int64 dwdwLifespan
);
```

See Also

Reference

CStencilCacheManager Class

Other Resources

CStencilCacheManager Members

CStencilCacheMgrObject Class

This class provides methods that can be used to implement IStencilCacheMgr by forwarding calls to implementations of IStencilCacheControl and IMemoryCacheStats.

```
class CStencilCacheMgrObject
```

Remarks

This class is used in the implementation of the extension management services for the ISAPI extension's stencil cache.

Requirements

Header: atlextmgmt.h

See Also

Tasks

BlobCache Sample: Generic Memory Caching
Hotswap Sample: Replacing ATL Server Application DLLs at Run Time
StencilCache Sample: Uses the ATL Server Stencil Cache

Other Resources

CStencilCacheMgrObject Members

CStencilCacheMgrObject Members

Methods

ClearStats	Delegates to IMemoryCacheStats::ClearStats on the interface obtained from the service provider passed to Initialize.
CStencilCacheMgrObject	The constructor.
GetCurrentAllocSize	Delegates to IMemoryCacheStats::GetCurrentAllocSize on the interface obtained from the service provider passed to Initialize.
GetCurrentEntryCount	Delegates to IMemoryCacheStats::GetCurrentEntryCount on the interface obtained from the service provider passed to Initialize.
GetDefaultLifespan	Delegates to IStencilCacheControl::GetDefaultLifespan on the interface obtained from the service provider passed to Initialize, but converts to milliseconds.
GetHitCount	Delegates to IMemoryCacheStats::GetHitCount on the interface obtained from the service provider passed to Initialize.
GetMaxAllocSize	Delegates to IMemoryCacheStats::GetMaxAllocSize on the interface obtained from the service provider passed to Initialize.
GetMaxEntryCount	Delegates to IMemoryCacheStats::GetMaxEntryCount on the interface obtained from the service provider passed to Initialize.
GetMissCount	Delegates to IMemoryCacheStats::GetMissCount on the interface obtained from the service provider passed to Initialize.
Initialize	Call this method to initialize the object.
RemoveAllStencils	Delegates to IStencilCacheControl::RemoveAllStencils on the interface obtained from the service provider passed to Initialize.
RemoveStencil	Delegates to IStencilCacheControl::RemoveStencil on the interface obtained from the service provider passed to Initialize.
RemoveStencilByName	Delegates to IStencilCacheControl::RemoveStencilByName on the interface obtained from the service provider passed to Initialize.
SetDefaultLifespan	Delegates to IStencilCacheControl::SetDefaultLifespan on the interface obtained from the service provider passed to Initialize, but converts to milliseconds.

See Also

Reference

CStencilCacheMgrObject Class

CStencilCacheMgrObject Methods

For information about the methods in **CStencilCacheMgrObject**, see CStencilCacheMgrObject Members

CStencilCacheMgrObject::ClearStats

Delegates to IMemoryCacheStats::ClearStats on the interface obtained from the service provider passed to Initialize.

```
HRESULT ClearStats( );
```

See Also

Reference

CStencilCacheMgrObject Class

Other Resources

CStencilCacheMgrObject Members

CStencilCacheMgrObject::CStencilCacheMgrObject

The constructor.

```
CStencilCacheMgrObject( );
```

See Also

Reference

CStencilCacheMgrObject Class

Other Resources

CStencilCacheMgrObject Members

CStencilCacheMgrObject::GetCurrentAllocSize

Delegates to IMemoryCacheStats::GetCurrentAllocSize on the interface obtained from the service provider passed to Initialize.

```
HRESULT GetCurrentAllocSize(
   __int64 * pdwSize
);
```

Example

See the StencilCache Sample.

See Also

Reference

CStencilCacheMgrObject Class

CStencilCacheMgrObject::GetMaxAllocSize

Other Resources

CStencilCacheMgrObject Members

CStencilCacheMgrObject::GetCurrentEntryCount

Delegates to IMemoryCacheStats::GetCurrentEntryCount on the interface obtained from the service provider passed to Initialize.

```
HRESULT GetCurrentEntryCount(
   __int64 * pdwSize
);
```

Example

See the BlobCache Sample and the StencilCache Sample.

See Also

Reference

CStencilCacheMgrObject Class

CStencilCacheMgrObject::GetMaxEntryCount

Other Resources

CStencilCacheMgrObject Members

CStencilCacheMgrObject::GetDefaultLifespan

Delegates to IStencilCacheControl::GetDefaultLifespan on the interface obtained from the service provider passed to Initialize, but converts to milliseconds.

```
HRESULT GetDefaultLifespan(
   unsigned __int64* pdwdwLifespan
) throw( );
```

Parameters

pdwdwLifespan

[out] Address of the variable that, on success, receives the default life span of a stencil in the cache in milliseconds.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See the StencilCache Sample.

See Also

Reference

CStencilCacheMgrObject Class

CStencilCacheMgrObject::SetDefaultLifespan

Other Resources

CStencilCacheMgrObject Members

CStencilCacheMgrObject::GetHitCount

Delegates to IMemoryCacheStats::GetHitCount on the interface obtained from the service provider passed to Initialize.

```
HRESULT GetHitCount(
   __int64 * pdwSize
);
```

Example

See the BlobCache Sample and the StencilCache Sample.

See Also

Reference

CStencilCacheMgrObject Class

CStencilCacheMgrObject::GetMissCount

Other Resources

CStencilCacheMgrObject Members

CStencilCacheMgrObject::GetMaxAllocSize

Delegates to IMemoryCacheStats::GetMaxAllocSize on the interface obtained from the service provider passed to Initialize.

```
HRESULT GetMaxAllocSize(
   __int64 * pdwSize
);
```

Example

See the StencilCache Sample.

See Also

Reference

CStencilCacheMgrObject Class

CStencilCacheMgrObject::GetCurrentAllocSize

Other Resources

CStencilCacheMgrObject Members

CStencilCacheMgrObject::GetMaxEntryCount

Delegates to IMemoryCacheStats::GetMaxEntryCount on the interface obtained from the service provider passed to Initialize.

```
HRESULT GetMaxEntryCount(
   __int64 * pdwSize
);
```

Example

See the BlobCache Sample and the StencilCache Sample.

See Also

Reference

CStencilCacheMgrObject Class
CStencilCacheMgrObject::GetCurrentEntryCount

Other Resources

CStencilCacheMgrObject Members

CStencilCacheMgrObject::GetMissCount

Delegates to IMemoryCacheStats::GetMissCount on the interface obtained from the service provider passed to Initialize.

```
HRESULT GetMissCount(
   __int64 * pdwSize
);
```

Example

See the BlobCache Sample and the StencilCache Sample.

See Also

Reference

CStencilCacheMgrObject Class
CStencilCacheMgrObject::GetHitCount

Other Resources

CStencilCacheMgrObject Members

CStencilCacheMgrObject::Initialize

Call this method to initialize the object.

```
HTTP_CODE Initialize(
   IServiceProvider* pProvider
) throw( );
```

Parameters

pProvider

The service provider to be queried for the IStencilCache service.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

The service provider is queried for the IStencilCache service and the IMemoryCacheStats and IStencilCacheControl interfaces are obtained by calling **QueryInterface** on the returned interface.

See Also

Reference

CStencilCacheMgrObject Class

Other Resources

CStencilCacheMgrObject Members

CStencilCacheMgrObject::RemoveAllStencils

Delegates to IStencilCacheControl::RemoveAllStencils on the interface obtained from the service provider passed to Initialize.

```
HRESULT RemoveAllStencils( ) throw( );
```

Example

See the Hotswap Sample.

See Also

Reference

CStencilCacheMgrObject Class
CStencilCacheMgrObject::RemoveStencil
CStencilCacheMgrObject::RemoveStencilByName

Other Resources

CStencilCacheMgrObject Members

CStencilCacheMgrObject::RemoveStencil

Delegates to IStencilCacheControl::RemoveStencil on the interface obtained from the service provider passed to Initialize.

```
HRESULT RemoveStencil(
   __int64 hStencil
);
```

See Also

Reference

CStencilCacheMgrObject Class
CStencilCacheMgrObject::RemoveAllStencils
CStencilCacheMgrObject::RemoveStencilByName

Other Resources

CStencilCacheMgrObject Members

CStencilCacheMgrObject::RemoveStencilByName

Delegates to IStencilCacheControl::RemoveStencilByName on the interface obtained from the service provider passed to Initialize.

```
HRESULT RemoveStencilByName(
   BSTR szStencil
) throw( );
```

See Also

Reference

CStencilCacheMgrObject Class
CStencilCacheMgrObject::RemoveStencil
CStencilCacheMgrObject::RemoveAllStencils

Other Resources

CStencilCacheMgrObject Members

CStencilCacheMgrObject::SetDefaultLifespan

Delegates to IStencilCacheControl::SetDefaultLifespan on the interface obtained from the service provider passed to Initialize, but converts to milliseconds.

```
HRESULT SetDefaultLifespan(
   unsigned __int64 dwdwLifespan
) throw( );
```

Parameters

dwdwLifespan

The default life span of a stencil in the cache in milliseconds.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See the StencilCache Sample.

See Also

Reference

CStencilCacheMgrObject Class

CStencilCacheMgrObject::GetDefaultLifespan

Other Resources

CStencilCacheMgrObject Members

CStencilMgr Class

This class implements the request handler that allows you to manage the stencil cache in an ATL Server ISAPI extension using an HTML interface.

```
[ request_handler( name = ID_STENCILCACHEMGR_SRFHANDLER_NAME ) ]
class CStencilMgr
```

Remarks

See Extension Management Services for more details.

Requirements

Header: atlextmgmt.h

See Also

Reference

CRequestHandlerT Class
request_handler
ID_STENCILCACHEMGR_SRFHANDLER_NAME

Other Resources

CStencilMgr Members

CStencilMgr Members

Methods

CStencilMgr	The constructor.
ExecCommand	This method is called from ValidateAndExchange to execute the command specified in the request's query string.
GetBodyColor	This replacement method can be used in a server response file to write a color value for the body of the HTML page.
GetCacheQuantity	This replacement method can be used in a server response file to get the value of the current cache statistic.
GetCacheValue	This replacement method can be used in a server response file to get the description of the current cache statistic.
GetNextStencilCacheStats	This replacement method can be used in a server response file to get the next statistic about the cache.
GetTRColor	This replacement method can be used in a server response file to write a color value for the current row.
ValidateAndExchange	This method validates requests sent to the request handler, including authorizing the client.

See Also

Reference

CStencilMgr Class

CStencilMgr Methods

For information about the methods in **CStencilMgr**, see CStencilMgr Members

CStencilMgr::CStencilMgr

The constructor.

```
CStencilMgr( );
```

See Also

Reference

CStencilMgr Class

Other Resources

CStencilMgr Members

CStencilMgr::ExecCommand

This method is called from ValidateAndExchange to execute the command specified in the request's query string.

```
HTTP_CODE ExecCommand(
   int nCmdToExec,
   CString& strOptParam
);
```

Parameters

nCmdToExec

The command to execute.

strOptParam

The parameters for the command.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

This method supports the following actions when the **command** and **DynValue** query parameters are set to the values shown:

Action	command	DynValue
IStencilCacheMgr::ClearStats	0	N/A
IStencilCacheMgr::RemoveStencilByName	1	The name of the stencil to be removed.
IStencilCacheMgr::RemoveAllStencils	2	N/A
IStencilCacheMgr::SetDefaultLifespan	3	The default life span of a stencil in the cache in milliseconds.

See Also

Reference

CStencilMgr Class

Other Resources

CStencilMgr Members

CStencilMgr::GetBodyColor

This replacement method can be used in a server response file to write a color value for the body of the HTML page.

```
[ tag_name("GetBodyColor") ]
HTTP_CODE GetBodyColor( );
```

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

CStencilMgr Class

CStencilMgr::GetTRColor

Other Resources

CStencilMgr Members

CStencilMgr::GetCacheQuantity

This replacement method can be used in a server response file to get the value of the current cache statistic.

```
[ tag_name( "GetCacheQuantity" ) ]
HTTP_CODE GetCacheQuantity( );
```

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

A cache statistic is made *current* by using the GetNextStencilCacheStats replacement method.

See Also

Reference

CStencilMgr Class

CStencilMgr::GetCacheValue

Other Resources

CStencilMgr Members

CStencilMgr::GetCacheValue

This replacement method can be used in a server response file to get the description of the current cache statistic.

```
[ tag_name( "GetCacheValue" ) ]
HTTP_CODE GetCacheValue( );
```

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

A cache statistic is made *current* by using the GetNextStencilCacheStats replacement method.

See Also

Reference

CStencilMgr Class

CStencilMgr::GetCacheQuantity

Other Resources

CStencilMgr Members

CStencilMgr::GetNextStencilCacheStats

This replacement method can be used in a server response file to get the next statistic about the cache.

```
[ tag_name( "GetNextStencilCacheStats" ) ]
HTTP_CODE GetNextStencilCacheStats( );
```

Return Value

Returns HTTP_SUCCESS if information about a cache statistic can be obtained, HTTP_S_FALSE if there are no more statistics, or an ATL Server status code indicating failure.

Remarks

This replacement method is designed to be used in a **while** replacement tag. It should be called before CStencilMgr::GetCacheQuantity and CStencilMgr::GetCacheValue.

The possible cache statistics are:

Statistic	Equivalent Method
Current entry count	IStencilCacheMgr::GetCurrentEntryCount
Hit count	IStencilCacheMgr::GetHitCount
Miss count	IStencilCacheMgr::GetMissCount
Current memory used	IStencilCacheMgr::GetCurrentAllocSize
Maximum memory used	IStencilCacheMgr::GetMaxAllocSize
Maximum entries	IStencilCacheMgr::GetMaxEntryCount
Default life span	IStencilCacheMgr::GetDefaultLifespan

See Also

Reference

CStencilMgr Class

Other Resources

CStencilMgr Members

CStencilMgr::GetTRColor

This replacement method can be used in a server response file to write a color value for the current row.

```
[ tag_name( "GetTRColor" ) ]
HTTP_CODE GetTRColor( );
```

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

The color value written by this method switches between one of two colors on each call.

See Also

Reference

CStencilMgr Class

CStencilMgr::GetBodyColor

Other Resources

CStencilMgr Members

CStencilMgr::ValidateAndExchange

This method validates requests sent to the request handler, including authorizing the client.

```
HTTP_CODE ValidateAndExchange( ) throw( );
```

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

CStencilMgr Class

Other Resources

CStencilMgr Members

CStencilState Structure

This class provides state information for use when rendering stencils using CStencil and CHtmlStencil.

```
struct CStencilState
```

Remarks

The state information provided by this class is typically used during asynchronous responses.

Requirements

Header: atlisapi.h

See Also

Reference

CStencil Class

CHtmlStencil Class

Other Resources

CStencilState Members

CStencilState Members

Methods

CStencilState	The constructor.

Data Members

dwIndex	The index of the token to be rendered next.
locale	The current locale to use during stencil rendering.
pIncludeInfo	The request information for an included request handler and stencil.
pParentInfo	The request information for the parent request handler and stencil.

See Also

Reference

CStencilState Structure

CStencilState Methods

For information about the methods in **CStencilState**, see CStencilState Members.

CStencilState::CStencilState

The constructor.

```
CStencilState( ) throw( );
```

Remarks

Initializes all data members to zero.

See Also

Reference

CStencilState Structure

Other Resources

CStencilState Members

CStencilState Data Members

For information about the data members in **CStencilState**, see CStencilState Members.

CStencilState::dwIndex

The index of the token to be rendered next.

```
DWORD dwIndex;
```

See Also

Reference

CStencilState Structure

CStencil::RenderToken

Other Resources

CStencilState Members

CStencilState::locale

The current locale to use during stencil rendering.

```
LCID locale;
```

See Also

Reference

CStencilState Structure

Locale Tag

Other Resources

CStencilState Members

CStencilState::pIncludeInfo

The request information for an included request handler and stencil.

```
AtlServerRequest* pIncludeInfo;
```

See Also

Reference

CStencilState Structure

Include Tag

Other Resources

CStencilState Members

CStencilState::pParentInfo

The request information for the parent request handler and stencil.

```
AtlServerRequest* pParentInfo;
```

See Also

Reference

CStencilState Structure

Other Resources

CStencilState Members

CStreamFormatter Class

This class writes formatted text to a stream.

```
class CStreamFormatter
```

Remarks

This is a utility class designed to be used as a base class for classes that generate HTML (such as CHtmlGenBase) or XML. It provides methods for writing tags, attributes, and formatted text to a stream.

Requirements

Header: atlhtml.h

See Also

Other Resources

CStreamFormatter Members

ATL Server Classes

CStreamFormatter Members

Methods

AddCRLF	Call this method to add CRLF characters around block-oriented tags.
CStreamFormatter	The constructor.
EmitUnicode	Call this method to specify whether strings are emitted as Unicode.
EndTag	Call this method to insert an end tag.
Initialize	Call this method to initialize the stream formatter.
SetConversionCodepage	Call this method to set the codepage used for conversions between ANSI and Unicode.
StartTag	Call this method to insert a start tag.
WriteAttributes	Call this method to write attributes to the stream.
WriteFormatted	Call this method to write a formatted string to the stream.
WriteRaw	Call this method to write a string to the stream.

See Also

Reference

CStreamFormatter Class

CStreamFormatter Methods

For information about the methods in **CStreamFormatter**, see CStreamFormatter Members.

CStreamFormatter::AddCRLF

Call this method to add CRLF characters around block-oriented tags.

```
void AddCRLF(
   bool bNewVal
);
```

Parameters

bNewVal

If **true**, carriage return and line feed characters will be added around block-oriented tags (such as `<body>` and `<html>`). If **false**, carriage return and line feed characters will not be added.

Remarks

The default action is determined by CStreamFormatter::Initialize.

See Also

Reference

CStreamFormatter Class

Other Resources

CStreamFormatter Members

CStreamFormatter::CStreamFormatter

The constructor.

```
CStreamFormatter( );
```

See Also

Reference

CStreamFormatter Class

CStreamFormatter::Initialize

Other Resources

CStreamFormatter Members

CStreamFormatter::EmitUnicode

Call this method to specify whether strings are emitted as Unicode.

```
void EmitUnicode(
   BOOL bEmitUnicode
);
```

Parameters

bEmitUnicode

If **true**, strings are emitted as Unicode; if **false**, strings are emitted as ANSI.

Remarks

This affects the output of CStreamFormatter::WriteRaw. The default is ANSI.

See Also

Reference

CStreamFormatter Class

Other Resources

CStreamFormatter Members

CStreamFormatter::EndTag

Call this method to insert an end tag.

```
HRESULT EndTag(
   LPCTSTR szTag
);
HRESULT EndTag(
   int nTagIndex
);
```

Parameters

szTag

The tag.

nTagIndex

The tag index. For a list of possible values, see ATL_HTML_TAGS.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

```
CStreamFormatter s;
s.EndTag( "test" );             // writes "</test>"
s.EndTag( ATL_HTML_TAG_BODY );  // writes "</body>"
```

See Also

Reference

CStreamFormatter Class

CStreamFormatter::StartTag

Other Resources

CStreamFormatter Members

CStreamFormatter::Initialize

Call this method to initialize the stream formatter.

```
void Initialize(
   IStream * pStream,
   BOOL bAddCRLF = TRUE
);
void Initialize(
   IWriteStream * pWriteStream,
   BOOL bAddCRLF = TRUE
);
```

Parameters

pStream, *pWriteStream*

The stream to which the output will be written.

bAddCRLF

If TRUE, carriage return and line feed characters will be added around block-oriented tags (such as `<body>` and `<html>`).

Remarks

Call this method to associate a stream with a **CStreamFormatter** instance. The stream must implement the interface.

See Also

Reference

CStreamFormatter Class
IStream
IWriteStream Interface

Other Resources

CStreamFormatter Members

CStreamFormatter::SetConversionCodepage

Call this method to set the codepage used for conversions between ANSI and Unicode.

```
void SetConversionCodepage(
   UINT nConversionCodepage
);
```

Parameters

nConversionCodepage

The codepage used for conversions between ANSI and Unicode.

Remarks

This affects the output of CStreamFormatter::WriteRaw. The codepage defaults to the ANSI code page for the current thread.

See Also

Reference

CStreamFormatter Class

Other Resources

CStreamFormatter Members

CStreamFormatter::StartTag

Call this method to insert a start tag.

```
HRESULT StartTag(
   int nTagIndex,
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
HRESULT StartTag(
   LPCTSTR szTag,
   LPCTSTR szContent = NULL,
   LPCTSTR szAttrs = NULL
);
```

Parameters

nTagIndex

The tag index. For a list of possible values, see ATL_HTML_TAGS.

szContent

The content of the tag.

szAttrs

The attributes of the tag.

szTag

The tag.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *szContent* is not NULL, a corresponding end tag will be inserted automatically.

Example

```
s.StartTag( "a" );             // writes "<a>"
s.StartTag( ATL_HTML_TAG_A );  // writes "<a>"
s.StartTag( ATL_HTML_TAG_A, "click here" );  // writes "<a>click here</a>"
// the following line writes "<a href=www.microsoft.com>click here</a>"
s.StartTag( "a", "click here", "href=www.microsoft.com" );
```

See Also

Reference

CStreamFormatter Class
CStreamFormatter::EndTag
AtlHtmlAttrs Class

Other Resources

CStreamFormatter Members

CStreamFormatter::WriteAttributes

Call this method to write attributes to the stream.

```
HRESULT WriteAttributes(
   LPCTSTR szAttrs
);
```

Parameters

szAttrs

The attributes to write to the stream.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method does not check that *szAttrs* contains valid HTML or XML attributes; *szAttrs* can be an arbitrary string. If *szAttrs* doesn't begin with a space, this method writes a space character before writing *szAttrs* (so consecutive calls to **WriteAttributes** result in space-separated attributes).

Example

```
s.WriteAttributes( "bgcolor=#ffff00" ); // writes " bgcolor=#ffff00"
```

See Also

Reference

CStreamFormatter Class

AtlHtmlAttrs Class

Other Resources

CStreamFormatter Members

CStreamFormatter::WriteFormatted

Call this method to write a formatted string to the stream.

```
HRESULT WriteFormatted(
   LPCTSTR szFormat,
   ...
);
```

Parameters

szFormat

The format control string.

...

Optional parameters to be formatted according to *szFormat*.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method is similar to printf. The optional parameters are formatted according to the format specification in *szFormat*, and the resulting string is written to the stream.

Example

```
s.WriteFormatted( "bgcolor=#%02x%02x%02x", 255, 0, 255 ); // writes "bgcolor=#ff00ff"
```

See Also

Reference

CStreamFormatter Class

Other Resources

CStreamFormatter Members

CStreamFormatter::WriteRaw

Call this method to write a string to the stream.

```
HRESULT WriteRaw(
   LPCTSTR szString,
   int nCount = - 1
);
```

Parameters

szString

The string to write.

nCount

The count of characters to write. If this is –1, the string is assumed to be null-terminated, and the entire string is written.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The string is emitted as Unicode or ANSI, depending on CStreamFormatter::EmitUnicode. Conversion (if necessary) is done with the code page specified by CStreamFormatter::SetConversionCodepage.

Example

```
s.Write( _T("howdy") );     // writes "howdy"
s.Write( _T("howdy"), 3 );  // writes "how"
```

See Also

Reference

CStreamFormatter Class

Other Resources

CStreamFormatter Members

CStreamOnServerContext Class

This class provides an implementation of IStream that delegates Read calls to an IHttpServerContext pointer.

```
class CStreamOnServerContext :
   public IStreamImpl
```

Remarks

The server context from which data is to be read when CStreamOnServerContext::Read is called can be passed to the constructor or to SetServerContext.

Requirements

Header: atlsoap.h

Example

See the PooledAsync Sample.

See Also

Reference

IStreamImpl Class
IStream
ISequentialStream::Read

Other Resources

CStreamOnServerContext Members

CStreamOnServerContext Members

Methods

AddRef	Implementation of **IUnknown::AddRef**.
CStreamOnServerContext	The constructor.
QueryInterface	Implementation of **IUnknown::QueryInterface**.
Read	Implementation of ISequentialStream::Read.
Release	Implementation of **IUnknown::Release**.
SetServerContext	Call this method to set the server context from which to read client data.

See Also

Reference

CStreamOnServerContext Class
IStreamImpl Class

CStreamOnServerContext Methods

For information about the methods in **CStreamOnServerContext**, see CStreamOnServerContext Members

CStreamOnServerContext::AddRef

Implementation of **IUnknown::AddRef**.

```
ULONG __stdcall AddRef( );
```

Return Value

Always returns 1.

Remarks

This class does not implement lifetime control using reference counting.

See Also

Reference

CStreamOnServerContext Class

CStreamOnServerContext::Release

Other Resources

CStreamOnServerContext Members

CStreamOnServerContext::CStreamOnServerContext

The constructor.

```
CStreamOnServerContext(
   IHttpServerContext* pServerContext = NULL
);
```

Parameters

pServerContext

The server context from which to read client data.

See Also

Reference

CStreamOnServerContext Class

CStreamOnServerContext::SetServerContext

Other Resources

CStreamOnServerContext Members

CStreamOnServerContext::QueryInterface

Implementation of **IUnknown::QueryInterface**.

```
HRESULT __stdcall QueryInterface(
   REFIID riid,
   void** ppv
);
```

Remarks

Objects of this class can be successfully queried for the **IUnknown**, ISequentialStream, and IStream interfaces.

See Also

Reference

CStreamOnServerContext Class

Other Resources

CStreamOnServerContext Members

CStreamOnServerContext::Read

Implementation of ISequentialStream::Read.

```
HRESULT __stdcall Read(
   void* pDest,
   ULONG nMaxLen,
   ULONG* pnRead
);
```

Remarks

Reads client data from the server context passed to the constructor or SetServerContext.

See Also

Reference

CStreamOnServerContext Class

Other Resources

CStreamOnServerContext Members

CStreamOnServerContext::Release

Implementation of **IUnknown::Release**.

```
ULONG __stdcall Release( );
```

Return Value

Always returns 1.

Remarks

This class does not implement lifetime control using reference counting.

See Also

Reference

CStreamOnServerContext Class

CStreamOnServerContext::AddRef

Other Resources

CStreamOnServerContext Members

CStreamOnServerContext::SetServerContext

Call this method to set the server context from which to read client data.

```
void SetServerContext(
   IHttpServerContext * pServerContext
);
```

Parameters

pServerContext

The server context from which to read client data.

See Also

Reference

CStreamOnServerContext Class

CStreamOnServerContext::CStreamOnServerContext

Other Resources

CStreamOnServerContext Members

CStreamOnWriteStream Class

This class provides an implementation of IStream that delegates calls to an IWriteStream pointer.

```
class CStreamOnWriteStream :
   public IStream
```

Remarks

The object implementing the **IWriteStream** interface can be passed to Init and is available in m_pWriteStream.

Write is the only **IStream** method implemented using the IWriteStream pointer. All other **IStream** methods return E_NOTIMPL.

Requirements

Header: atlhtml.h

See Also

Reference

IStream

Other Resources

CStreamOnWriteStream Members

CStreamOnWriteStream Members

Methods

AddRef	Implementation of IUnknown::AddRef.
Clone	Non-functional implementation of IStream::Clone.
Commit	Non-functional implementation of IStream::Commit.
CopyTo	Non-functional implementation of IStream::CopyTo.
CStreamOnWriteStream	The constructor.
Init	Call this method to initialize this object with the IWriteStream object used to provide the implementation of the IStream methods.
LockRegion	Non-functional implementation of IStream::LockRegion.
QueryInterface	Implementation of IUnknown::QueryInterface.
Read	Non-functional implementation of ISequentialStream::Read.
Release	Implementation of IUnknown::Release.
Revert	Non-functional implementation of IStream::Revert.
Seek	Non-functional implementation of IStream::Seek.
SetSize	Non-functional implementation of IStream::SetSize.
Stat	Non-functional implementation of IStream::Stat.
UnlockRegion	Non-functional implementation of IStream::UnlockRegion.
Write	Implementation of ISequentialStream::Write.

Data Members

m_pWriteStream	The IWriteStream pointer used to implement Write.

See Also

Reference

CStreamOnWriteStream Class

CStreamOnWriteStream Methods

For information about the methods in **CStreamOnWriteStream**, see CStreamOnWriteStream Members.

CStreamOnWriteStream::AddRef

Implementation of IUnknown::AddRef.

```
ULONG __stdcall AddRef( );
```

Return Value

Always returns 1.

Remarks

This class does not implement lifetime control using reference counting.

See Also

Reference

CStreamOnWriteStream Class

Other Resources

CStreamOnWriteStream Members

CStreamOnWriteStream::Clone

Non-functional implementation of IStream::Clone.

```
HRESULT STDMETHODCALLTYPE Clone(
   IStream** /* ppstm */
);
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

CStreamOnWriteStream Class

Other Resources

CStreamOnWriteStream Members

CStreamOnWriteStream::Commit

Non-functional implementation of IStream::Commit.

```
HRESULT STDMETHODCALLTYPE Commit(
   DWORD /* grfCommitFlags */
);
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

CStreamOnWriteStream Class

Other Resources

CStreamOnWriteStream Members

CStreamOnWriteStream::CopyTo

Non-functional implementation of IStream::CopyTo.

```
HRESULT STDMETHODCALLTYPE CopyTo(
   IStream * /* pstm */,
   ULARGE_INTEGER /* cb */,
   ULARGE_INTEGER * /* pcbRead */,
   ULARGE_INTEGER * /* pcbWritten */
);
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

CStreamOnWriteStream Class

Other Resources

CStreamOnWriteStream Members

CStreamOnWriteStream::CStreamOnWriteStream

The constructor.

```
CStreamOnWriteStream( );
```

Remarks

Initializes m_pWriteStream to NULL.

See Also

Reference

CStreamOnWriteStream Class

Other Resources

CStreamOnWriteStream Members

CStreamOnWriteStream::Init

Call this method to initialize this object with the IWriteStream object used to provide the implementation of the IStream methods.

```
void Init(
   IWriteStream * pWriteStream
);
```

Parameters

pWriteStream

The IWriteStream object used to provide the implementation of the IStream methods.

Remarks

Sets CStreamOnWriteStream::m_pWriteStream to *pWriteStream*.

See Also

Reference

CStreamOnWriteStream Class

Other Resources

CStreamOnWriteStream Members

CStreamOnWriteStream::LockRegion

Non-functional implementation of IStream::LockRegion.

```
HRESULT STDMETHODCALLTYPE LockRegion(
   ULARGE_INTEGER /* libOffset */,
   ULARGE_INTEGER /* cb */,
   DWORD /* dwLockType */
);
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

CStreamOnWriteStream Class

Other Resources

CStreamOnWriteStream Members

CStreamOnWriteStream::QueryInterface

Implementation of IUnknown::QueryInterface.

```
STDMETHOD(QueryInterface)(
   REFIID riid,
   void** ppv
);
```

Remarks

Objects of this class can be successfully queried for the IUnknown, IStream, and **ISequentialStream** interfaces.

See Also

Reference

CStreamOnWriteStream Class

Other Resources

CStreamOnWriteStream Members

CStreamOnWriteStream::Read

Non-functional implementation of ISequentialStream::Read.

```
HRESULT STDMETHODCALLTYPE Read(
   void * /* pDest */,
   ULONG /* dwMaxLen */,
   ULONG * /* pdwRead */
);
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

CStreamOnWriteStream Class

Other Resources

CStreamOnWriteStream Members

CStreamOnWriteStream::Release

Implementation of IUnknown::Release.

```
ULONG __stdcall Release( );
```

Return Value

Always returns 1.

Remarks

This class does not implement lifetime control using reference counting.

See Also

Reference

CStreamOnWriteStream Class

Other Resources

CStreamOnWriteStream Members

CStreamOnWriteStream::Revert

Non-functional implementation of IStream::Revert.

```
HRESULT STDMETHODCALLTYPE Revert( void );
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

CStreamOnWriteStream Class

Other Resources

CStreamOnWriteStream Members

CStreamOnWriteStream::Seek

Non-functional implementation of IStream::Seek.

```
HRESULT STDMETHODCALLTYPE Seek(
   LARGE_INTEGER /* dlibMove */,
   DWORD /* dwOrigin */,
   ULARGE_INTEGER * /* plibNewPosition */
);
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

CStreamOnWriteStream Class

Other Resources

CStreamOnWriteStream Members

CStreamOnWriteStream::SetSize

Non-functional implementation of IStream::SetSize.

```
HRESULT STDMETHODCALLTYPE SetSize(
   ULARGE_INTEGER /* libNewSize */
);
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

CStreamOnWriteStream Class

Other Resources

CStreamOnWriteStream Members

CStreamOnWriteStream::Stat

Non-functional implementation of IStream::Stat.

```
HRESULT STDMETHODCALLTYPE Stat(
   STATSTG * /* pstatstg */,
   DWORD /* grfStatFlag */
);
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

CStreamOnWriteStream Class

Other Resources

CStreamOnWriteStream Members

CStreamOnWriteStream::UnlockRegion

Non-functional implementation of IStream::UnlockRegion.

```
HRESULT STDMETHODCALLTYPE UnlockRegion(
   ULARGE_INTEGER /* libOffset */,
   ULARGE_INTEGER /* cb */,
   DWORD /* dwLockType */
);
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

CStreamOnWriteStream Class

Other Resources

CStreamOnWriteStream Members

CStreamOnWriteStream::Write

Implementation of ISequentialStream::Write.

```
HRESULT STDMETHODCALLTYPE Write(
   const void * pv,
   ULONG cb,
   ULONG * pcbWritten
);
```

Remarks

This method delegates to the IWriteStream::WriteStream method provided by m_pWriteStream

See Also

Reference

CStreamOnWriteStream Class

Other Resources

CStreamOnWriteStream Members

CStreamOnWriteStream Data Members

For information about the data members in **CStreamOnWriteStream**, see CStreamOnWriteStream Members.

CStreamOnWriteStream::m_pWriteStream

The IWriteStream pointer used to implement Write.

```
IWriteStream * m_pWriteStream;
```

Remarks

This member is set by a call to Init.

See Also

Reference

CStreamOnWriteStream Class

Other Resources

CStreamOnWriteStream Members

CTagReplacerMethodEntry Structure

This structure holds information about a replacement method including its name and pointers to the member functions that implement it.

```
template < class TBase >
struct CTagReplacerMethodEntry
```

Parameters

TBase

The class providing the methods used to implement the replacement method represented by this object.

Remarks

An array of these structures is used by ITagReplacerImpl in its implementation of the ITagReplacer interface. See ITagReplacerImpl::GetReplacementMethodMap for details.

Requirements

Header: atlstencil.h

See Also

Other Resources

CTagReplacerMethodEntry Members

CTagReplacerMethodEntry Members

Data Members

Methods	The object that holds pointers to the *TBase* methods that implement the replacement method represented by t his object.
nType	The type of the replacement method.
szMethodName	The name of the replacement method.

See Also

Reference

CTagReplacerMethodEntry Structure

CTagReplacerMethodEntry Data Members

For information about the data members in **CTagReplacerMethodEntry**, see CTagReplacerMethodEntry Members.

CTagReplacerMethodEntry::Methods

The object that holds pointers to the *TBase* methods that implement the replacement method represented by this object.

```
CTagReplacerMethods<TBase> Methods;
```

See Also

Reference

CTagReplacerMethodEntry Structure

CTagReplacerMethods Structure

Other Resources

CTagReplacerMethodEntry Members

CTagReplacerMethodEntry::nType

The type of the replacement method.

```
int nType;
```

Remarks

Can be REPLACEMENT_ENTRY_DEFAULT or REPLACEMENT_ENTRY_ARGS. Determines whether to use the pfnMethod or pfnMethodEx member of the Methods object.

See Also

Reference

CTagReplacerMethodEntry Structure
CTagReplacerMethods::pfnMethod
CTagReplacerMethods::pfnMethodEx

Other Resources

CTagReplacerMethodEntry Members

CTagReplacerMethodEntry::szMethodName

The name of the replacement method.

```
LPCSTR szMethodName;
```

See Also

Reference

CTagReplacerMethodEntry Structure

Other Resources

CTagReplacerMethodEntry Members

CTagReplacerMethods Structure

This structure holds pointers to the member functions necessary for implementing a replacement method.

```
template < class TBase >
struct CTagReplacerMethods
```

Parameters

TBase

The class providing the methods used to implement the replacement method represented by this object.

Remarks

This structure is used as a member of CTagReplacerMethodEntry.

Requirements

Header: atlstencil.h

See Also

Other Resources

CTagReplacerMethods Members

CTagReplacerMethodEntry Members

CTagReplacerMethods Members

Data Members

pfnMethod	This member holds a pointer to the member function implementing a replacement method if that member function takes no arguments.
pfnMethodEx	This member holds a pointer to the member function implementing a replacement method if that member function takes arguments.
pfnParse	This member holds a pointer to the member function used for parsing arguments passed as text so that they can be used by CTagReplacerMethods::pfnMethodEx.

See Also

Reference

CTagReplacerMethods Structure

CTagReplacerMethods Data Members

For information about the data members in **CTagReplacerMethods**, see CTagReplacerMethods Members.

CTagReplacerMethods::pfnMethod

This member holds a pointer to the member function implementing a replacement method if that member function takes no arguments.

```
HTTP_CODE (TBase::*pfnMethod)();
```

Remarks

This member is in a union with CTagReplacerMethods::pfnMethodEx. Typically CTagReplacerMethodEntry::nType is used to determine which member is valid.

See Also

Reference

CTagReplacerMethods Structure

Unions

Other Resources

CTagReplacerMethods Members

CTagReplacerMethods::pfnMethodEx

This member holds a pointer to the member function implementing a replacement method if that member function takes arguments.

```
HTTP_CODE (TBase::*pfnMethodEx)(void *);
```

Remarks

This member is in a union with CTagReplacerMethods::pfnMethod. Typically CTagReplacerMethodEntry::nType is used to determine which member is valid.

See Also

Reference

CTagReplacerMethods Structure

Unions

Other Resources

CTagReplacerMethods Members

CTagReplacerMethods::pfnParse

This member holds a pointer to the member function used for parsing arguments passed as text so that they can be used by CTagReplacerMethods::pfnMethodEx.

```
HTTP_CODE (TBase::*pfnParse)(IAtlMemMgr *pMemMgr, LPCSTR, void **);
```

See Also

Reference

CTagReplacerMethods Structure

Other Resources

CTagReplacerMethods Members

CTagReplacerMethodsEx Structure

This structure provides static functions for casting type-specific replacement methods and parse methods to generic versions that can be stored in the CTagReplacerMethodEntry structure.

```
template <
   class TBase,
   typename T
>
struct CTagReplacerMethodsEx
```

Parameters

TBase

The class providing the methods used to implement a replacement method.

T

The type of data returned from the parse method and passed to the replacement method.

Remarks

The static functions and typedefs in this structure have two purposes:

- Ensure that the types used in parse methods and replacement methods match each other and the signature specified for them.
- Cast the function pointers to generic versions that can be stored in a less strongly typed pointer.

Requirements

Header: atlstencil.h

See Also

Reference

CTagReplacerMethodEntry Structure

CTagReplacerMethods Structure

Other Resources

CTagReplacerMethodsEx Members

CTagReplacerMethodsEx Members

Static Functions

CheckParse	Call this function to cast a member function pointer of type PARSE_FUNC to a member function pointer of type PARSE_FUNC_V.
CheckRepl	Call this function to cast a member function pointer of type REPLACE_FUNC to a member function pointer of type REPLACE_FUNC_EX_V.
CheckReplEx	Call this function to cast a member function pointer of type REPLACE_FUNC_EX to a member function pointer of type REPLACE_FUNC_EX_V.

Typedefs

PARSE_FUNC	The type of a pointer to a parse method that returns a pointer to an object of type *T* through an out parameter.
PARSE_FUNC_V	The type of a pointer to a parse method.
REPLACE_FUNC	The type of a pointer to a replacement method that takes no parameters.
REPLACE_FUNC_EX	The type of a pointer to a replacement method that takes a parameter of type *T**.
REPLACE_FUNC_EX_V	The type of a pointer to a replacement method that takes a parameter.

See Also

Reference

CTagReplacerMethodsEx Structure

CTagReplacerMethodsEx Static Functions

For information about the static functions in **CTagReplacerMethodsEx**, see CTagReplacerMethodsEx Members.

CTagReplacerMethodsEx::CheckParse

Call this function to cast a member function pointer of type PARSE_FUNC to a member function pointer of type PARSE_FUNC_V.

```
static PARSE_FUNC_V CheckParse(
   PARSE_FUNC p
) throw();
```

Parameters

p

A pointer to a parse method.

Return Value

Returns *p* cast to PARSE_FUNC_V

See Also

Reference

CTagReplacerMethodsEx Structure
CTagReplacerMethodsEx::PARSE_FUNC_V
CTagReplacerMethodsEx::PARSE_FUNC

Other Resources

CTagReplacerMethodsEx Members

CTagReplacerMethodsEx::CheckRepl

Call this function to cast a member function pointer of type REPLACE_FUNC to a member function pointer of type REPLACE_FUNC_EX_V.

```
static REPLACE_FUNC_EX_V CheckRepl(
   REPLACE_FUNC p
) throw();
```

Parameters

p

A pointer to a replacement method.

Return Value

Returns *p* cast to REPLACE_FUNC_EX_V

See Also

Reference

CTagReplacerMethodsEx Structure
CTagReplacerMethodsEx::REPLACE_FUNC_EX_V
CTagReplacerMethodsEx::REPLACE_FUNC

Other Resources

CTagReplacerMethodsEx Members

CTagReplacerMethodsEx::CheckReplEx

Call this function to cast a member function pointer of type REPLACE_FUNC_EX to a member function pointer of type REPLACE_FUNC_EX_V.

```
static REPLACE_FUNC_EX_V CheckReplEx(
   REPLACE_FUNC_EX p
) throw();
```

Parameters

p

A pointer to a replacement method.

Return Value

Returns *p* cast to REPLACE_FUNC_EX_V

See Also

Reference

CTagReplacerMethodsEx Structure
CTagReplacerMethodsEx::REPLACE_FUNC_EX_V
CTagReplacerMethodsEx::REPLACE_FUNC_EX

Other Resources

CTagReplacerMethodsEx Members

CTagReplacerMethodsEx Typedefs

For information about the typedefs in **CTagReplacerMethodsEx**, see CTagReplacerMethodsEx Members.

CTagReplacerMethodsEx::PARSE_FUNC

The type of a pointer to a parse method that returns a pointer to an object of type *T* through an out parameter.

```
typedef HTTP_CODE (TBase::*PARSE_FUNC)(IAtlMemMgr *, LPCSTR, T**);
```

See Also

Reference

CTagReplacerMethodsEx Structure

Other Resources

CTagReplacerMethodsEx Members

CTagReplacerMethodsEx::PARSE_FUNC_V

The type of a pointer to a parse method.

```
typedef HTTP_CODE (TBase::*PARSE_FUNC_V)(IAtlMemMgr *, LPCSTR, void**);
```

See Also

Reference

CTagReplacerMethodsEx Structure

Other Resources

CTagReplacerMethodsEx Members

CTagReplacerMethodsEx::REPLACE_FUNC

The type of a pointer to a replacement method that takes no parameters.

```
typedef HTTP_CODE (TBase::*REPLACE_FUNC)(void);
```

See Also

Reference

CTagReplacerMethodsEx Structure

CTagReplacerMethodsEx::REPLACE_FUNC_EX_V

Other Resources

CTagReplacerMethodsEx Members

CTagReplacerMethodsEx::REPLACE_FUNC_EX

The type of a pointer to a replacement method that takes a parameter of type *T**.

```
typedef HTTP_CODE (TBase::*REPLACE_FUNC_EX)(T*);
```

See Also

Reference

CTagReplacerMethodsEx Structure

CTagReplacerMethodsEx::REPLACE_FUNC_EX_V

Other Resources

CTagReplacerMethodsEx Members

CTagReplacerMethodsEx::REPLACE_FUNC_EX_V

The type of a pointer to a replacement method that takes a parameter.

```
typedef HTTP_CODE (TBase::*REPLACE_FUNC_EX_V)(void *);
```

See Also

Reference

CTagReplacerMethodsEx Structure

Other Resources

CTagReplacerMethodsEx Members

CThreadMgrStencil Class

This class implements the request handler that allows you to manage the thread pool in an ATL Server ISAPI extension using an HTML interface.

```
[ request_handler(name = ID_THREADMGR_SRFHANDLER_NAME) ]
class CThreadMgrStencil
```

Remarks

See Extension Management Services for more details.

Requirements

Header: atlextmgmt.h

See Also

Reference

CRequestHandlerT Class
request_handler
ID_THREADMGR_SRFHANDLER_NAME

Other Resources

CThreadMgrStencil Members

CThreadMgrStencil Members

Methods

CThreadMgrStencil	The constructor.
ExecCommand	This method is called from ValidateAndExchange to execute the command specified in the request's query string.
GetBodyColor	This replacement method can be used in a server response file to write a color value for the body of the HTML page.
GetSize	This replacement method can be used in a server response file to write the number of threads in the thread pool.
GetTRColor	This replacement method can be used in a server response file to write a color value for the current row.
ValidateAndExchange	This method validates requests sent to the request handler, including authorizing the client.

See Also

Reference

CThreadMgrStencil Class

CThreadMgrStencil Methods

For information about the methods in **CThreadMgrStencil**, see CThreadMgrStencil Members

CThreadMgrStencil::CThreadMgrStencil

The constructor.

```
CThreadMgrStencil( );
```

See Also

Reference

CThreadMgrStencil Class

Other Resources

CThreadMgrStencil Members

CThreadMgrStencil::ExecCommand

This method is called from ValidateAndExchange to execute the command specified in the request's query string.

```
HTTP_CODE ExecCommand(
   int nCmdToExec,
   CString& strOptParam
);
```

Parameters

nCmdToExec

The command to execute.

strOptParam

The parameters for the command.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

This method supports the following actions when the **command** and **DynValue** query parameters are set to the values shown:

Action	command	DynValue
IThreadPoolMgr::SetSize	0	Number of threads

See Also

Reference

CThreadMgrStencil Class

Other Resources

CThreadMgrStencil Members

CThreadMgrStencil::GetBodyColor

This replacement method can be used in a server response file to write a color value for the body of the HTML page.

```
[ tag_name("GetBodyColor") ]
HTTP_CODE GetBodyColor( );
```

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

CThreadMgrStencil Class

CThreadMgrStencil::GetTRColor

Other Resources

CThreadMgrStencil Members

CThreadMgrStencil::GetSize

This replacement method can be used in a server response file to write the number of threads in the thread pool.

```
[ tag_name( "GetSize" ) ]
HTTP_CODE GetSize( );
```

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

Equivalent to IThreadPoolMgr::GetSize.

See Also

Reference

CThreadMgrStencil Class

Other Resources

CThreadMgrStencil Members

CThreadMgrStencil::GetTRColor

This replacement method can be used in a server response file to write a color value for the current row.

```
[ tag_name( "GetTRColor" ) ]
HTTP_CODE GetTRColor( );
```

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

The color value written by this method switches between one of two colors on each call.

See Also

Reference

CThreadMgrStencil Class

CThreadMgrStencil::GetBodyColor

Other Resources

CThreadMgrStencil Members

CThreadMgrStencil::ValidateAndExchange

This method validates requests sent to the request handler, including authorizing the client.

```
HTTP_CODE ValidateAndExchange( ) throw( );
```

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

CThreadMgrStencil Class

Other Resources

CThreadMgrStencil Members

CThreadPool Class

This class provides a pool of worker threads that process a queue of work items.

```
template <
   class Worker,
   class ThreadTraits = DefaultThreadTraits
>
class CThreadPool :
   public IThreadPoolConfig
```

Parameters

Worker

The class conforming to the worker archetype providing the code used to process work items queued on the thread pool.

ThreadTraits

The class providing the function used to create the threads in the pool.

Remarks

Threads in the pool are created and destroyed when the pool is initialized, resized, or shut down. An instance of class *Worker* will be created on the stack of each worker thread in the pool. Each instance will live for the lifetime of the thread.

Immediately after creation of a thread, *Worker*::**Initialize** will be called on the object associated with that thread. Immediately before destruction of a thread, *Worker*::**Terminate** will be called. Both methods must accept a **void*** argument. The value of this argument is passed to the thread pool through the *pvWorkerParam* parameter of CThreadPool::Initialize.

When there are work items in the queue and worker threads available for work, a worker thread will pull an item off the queue and call the **Execute** method of the *Worker* object for that thread. Three items are then passed to the method: the item from the queue, the same *pvWorkerParam* passed to *Worker*::**Initialize** and *Worker*::**Terminate**, and a pointer to the OVERLAPPED structure used for the IO completion port queue.

The *Worker* class declares the type of the items that will be queued on the thread pool by providing a typedef, *Worker*::**RequestType**. This type must be capable of being cast to and from a **ULONG_PTR**.

Examples of *Worker* classes include CNonStatelessWorker and CIsapiWorker.

Requirements

Header: atlutil.h

Example

See the PooledAsync Sample.

See Also

Reference

IThreadPoolConfig Interface
DefaultThreadTraits
IThreadPoolMgr Interface
CThreadPoolManager Class

Other Resources

CThreadPool Members
ATL Server Classes

CThreadPool Members

Methods

AddRef	Implementation of **IUnknown::AddRef**.
CThreadPool	The constructor for the thread pool.
~CThreadPool	The destructor for the thread pool.
GetNumThreads	Call this method to get the number of threads in the pool.
GetQueueHandle	Call this method to get the handle of the IO completion port used to queue work items.
GetSize	Call this method to get the number of threads in the pool.
GetTimeout	Call this method to get the maximum time in milliseconds that the thread pool will wait for a thread to shut down.
Initialize	Call this method to initialize the thread pool.
QueryInterface	Implementation of **IUnknown::QueryInterface**.
QueueRequest	Call this method to queue a work item to be handled by a thread in the pool.
Release	Implementation of **IUnknown::Release**.
SetSize	Call this method to set the number of threads in the pool.
SetTimeout	Call this method to set the maximum time in milliseconds that the thread pool will wait for a thread to shut down.
Shutdown	Call this method to shut down the thread pool.

See Also

Reference

CThreadPool Class

Other Resources

IThreadPoolConfig Members

CThreadPool Methods

For information about the methods in **CThreadPool**, see CThreadPool Members.

CThreadPool::AddRef

Implementation of **IUnknown::AddRef**.

```
ULONG STDMETHODCALLTYPE AddRef( ) throw( );
```

Return Value

Always returns 1.

Remarks

This class does not implement lifetime control using reference counting.

See Also

Reference

CThreadPool Class

CThreadPool::Release

Other Resources

CThreadPool Members

CThreadPool::CThreadPool

The constructor for the thread pool.

```
CThreadPool( ) throw( );
```

Remarks

Initializes the timeout value to ATLS_DEFAULT_THREADPOOLSHUTDOWNTIMEOUT.

See Also

Reference

CThreadPool Class
CThreadPool::GetTimeout
CThreadPool::SetTimeout

Other Resources

CThreadPool Members

CThreadPool::~CThreadPool

The destructor for the thread pool.

```
~CThreadPool( ) throw( );
```

Remarks

Calls CThreadPool::Shutdown.

See Also

Reference

CThreadPool Class

Other Resources

CThreadPool Members

CThreadPool::GetNumThreads

Call this method to get the number of threads in the pool.

```
int GetNumThreads( ) throw( );
```

Return Value

Returns the number of threads in the pool.

See Also

Reference

CThreadPool Class

CThreadPool::GetSize

Other Resources

CThreadPool Members

CThreadPool::GetQueueHandle

Call this method to get the handle of the IO completion port used to queue work items.

```
HANDLE GetQueueHandle( ) throw( );
```

Return Value

Returns the queue handle or NULL if the thread pool has not been initialized.

See Also

Reference

CThreadPool Class

Other Resources

CThreadPool Members

CThreadPool::GetSize

Call this method to get the number of threads in the pool.

```
HRESULT STDMETHODCALLTYPE GetSize(
   int* pnNumThreads
) throw( );
```

Parameters

pnNumThreads

[out] Address of the variable that, on success, receives the number of threads in the pool.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CThreadPool Class
IThreadPoolConfig::GetSize
CThreadPool::SetSize
CThreadPool::GetNumThreads

Other Resources

CThreadPool Members

CThreadPool::GetTimeout

Call this method to get the maximum time in milliseconds that the thread pool will wait for a thread to shut down.

```
HRESULT STDMETHODCALLTYPE GetTimeout(
   DWORD* pdwMaxWait
) throw( );
```

Parameters

pdwMaxWait

[out] Address of the variable that, on success, receives the maximum time in milliseconds that the thread pool will wait for a thread to shut down.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This timeout value is used by CThreadPool::Shutdown if no other value is supplied to that method.

See Also

Reference

CThreadPool Class
IThreadPoolConfig::GetTimeout
CThreadPool::SetTimeout

Other Resources

CThreadPool Members

CThreadPool::Initialize

Call this method to initialize the thread pool.

```
HRESULT Initialize(
   void * pvWorkerParam = NULL,
   int nNumThreads = 0,
   DWORD dwStackSize = 0,
   HANDLE hCompletion = INVALID_HANDLE_VALUE
) throw( );
```

Parameters

pvWorkerParam

The worker parameter to be passed to the worker thread object's **Initialize**, **Execute**, and **Terminate** methods.

nNumThreads

The requested number of threads in the pool.

If *nNumThreads* is negative, its absolute value will be multiplied by the number of processors in the machine to get the total number of threads.

If *nNumThreads* is zero, ATLS_DEFAULT_THREADSPERPROC will be multiplied by the number of processors in the machine to get the total number of threads.

dwStackSize

The stack size for each thread in the pool.

hCompletion

The handle of an object to associate with the completion port.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See the PooledAsync Sample.

See Also

Reference

CThreadPool Class

Other Resources

CThreadPool Members

CThreadPool::QueryInterface

Implementation of **IUnknown::QueryInterface**.

```
HRESULT STDMETHODCALLTYPE QueryInterface(
   REFIID riid,
   void** ppv
) throw( );
```

Remarks

Objects of this class can be successfully queried for the **IUnknown** and IThreadPoolConfig interfaces.

Example

See the PooledAsync Sample.

See Also

Reference

CThreadPool Class

Other Resources

CThreadPool Members

CThreadPool::QueueRequest

Call this method to queue a work item to be handled by a thread in the pool.

```
BOOL QueueRequest(
   Worker::RequestType request
) throw( );
```

Parameters

request

The request to be queued.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This method adds a work item to the queue. The threads in the pool pick items off the queue in the order in which they are received.

Example

See the PooledAsync Sample.

See Also

Reference

CThreadPool Class

Other Resources

CThreadPool Members

CThreadPool::Release

Implementation of **IUnknown::Release**.

```
ULONG STDMETHODCALLTYPE Release( ) throw( );
```

Return Value

Always returns 1.

Remarks

This class does not implement lifetime control using reference counting.

See Also

Reference

CThreadPool Class

CThreadPool::AddRef

Other Resources

CThreadPool Members

CThreadPool::SetSize

Call this method to set the number of threads in the pool.

```
HRESULT STDMETHODCALLTYPE SetSize(
   int nNumThreads
) throw( );
```

Parameters

nNumThreads

The requested number of threads in the pool.

If *nNumThreads* is negative, its absolute value will be multiplied by the number of processors in the machine to get the total number of threads.

If *nNumThreads* is zero, ATLS_DEFAULT_THREADSPERPROC will be multiplied by the number of processors in the machine to get the total number of threads.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If the number of threads specified is less than the number of threads currently in the pool, the object puts a shutdown message on the queue to be picked up by a waiting thread. When a waiting thread pulls the message off the queue, it notifies the thread pool and exits the thread procedure. This process is repeated until the number of threads in the pool reaches the specified number or until no thread has exited within the period specified by GetTimeout/SetTimeout. In this situation the method will return an HRESULT corresponding to **WAIT_TIMEOUT** and the pending shutdown message is canceled.

See Also

Reference

CThreadPool Class
IThreadPoolConfig::SetSize
CThreadPool::GetSize

Other Resources

CThreadPool Members

CThreadPool::SetTimeout

Call this method to set the maximum time in milliseconds that the thread pool will wait for a thread to shut down.

```
HRESULT STDMETHODCALLTYPE SetTimeout(
   DWORD dwMaxWait
) throw( );
```

Parameters

dwMaxWait

The requested maximum time in milliseconds that the thread pool will wait for a thread to shut down.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The timeout is initialized to ATLS_DEFAULT_THREADPOOLSHUTDOWNTIMEOUT in the constructor.

Note that *dwMaxWait* is the time that the pool will wait for a single thread to shut down. The maximum time that could be taken to remove multiple threads from the pool could be slightly less than *dwMaxWait* multiplied by the number of threads.

See Also

Reference

CThreadPool Class
IThreadPoolConfig::SetTimeout
CThreadPool::GetTimeout

Other Resources

CThreadPool Members

CThreadPool::Shutdown

Call this method to shut down the thread pool.

```
void Shutdown(
   DWORD dwMaxWait = 0
) throw( );
```

Parameters

dwMaxWait

The requested maximum time in milliseconds that the thread pool will wait for a thread to shut down. If 0 or no value is supplied, this method will use the timeout set by CThreadPool::SetTimeout.

Remarks

This method posts a shutdown request to all threads in the pool. If the timeout expires, this method will call TerminateThread on any thread that did not exit. This method is called automatically from the destructor of the class.

Example

See the PooledAsync Sample.

See Also

Reference

CThreadPool Class

Other Resources

CThreadPool Members

CThreadPoolManager Class

This class implements the XML Web service that allows you to manage the thread pool in an ATL Server ISAPI extension.

```
[
   soap_handler(
               name=      ID_THREADMGR_WEBSERVICE_NAME,
               namespace= ID_THREADMGR_WEBSERVICE_URL,
               protocol=  "soap"
            ),
   request_handler(
               name= ID_THREADMGR_WEBSERVICE_NAME,
               sdl=  ID_THREADMGR_WEBSERVICE_WSDL
               )
]
class CThreadPoolManager :
   public IThreadPoolMgr
```

Remarks

See Extension Management Services for more details.

Requirements

Header: atlextmgmt.h

See Also

Reference

IThreadPoolMgr Interface
soap_handler
request_handler
ID_THREADMGR_WEBSERVICE_NAME
ID_THREADMGR_WEBSERVICE_URL
ID_THREADMGR_WEBSERVICE_WSDL
CThreadPool Class

Other Resources

CThreadPoolManager Members

CThreadPoolManager Members

Methods

GetSize	Implementation of IThreadPoolMgr::GetSize.
HandleRequest	This method handles requests sent to the thread pool manager service, including authorizing the client.
SetSize	Implementation of IThreadPoolMgr::SetSize.

See Also

Reference

CThreadPoolManager Class

CThreadPoolManager Methods

For information about the methods in **CThreadPoolManager**, see CThreadPoolManager Members.

CThreadPoolManager::GetSize

Implementation of IThreadPoolMgr::GetSize.

```
[soap_method]
STDMETHOD(GetSize)(
   int* pnNumThreads
);
```

See Also

Reference

CThreadPoolManager Class

IThreadPoolMgr::GetSize

Other Resources

CThreadPoolManager Members

CThreadPoolManager::HandleRequest

This method handles requests sent to the thread pool manager service, including authorizing the client.

```
HTTP_CODE HandleRequest(
   AtlServerRequest* pRequestInfo,
   IServiceProvider* pProvider
);
```

Parameters

pRequestInfo

The information about the request to be handled.

pProvider

The service provider.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

CThreadPoolManager Class

Other Resources

CThreadPoolManager Members

CThreadPoolManager::SetSize

Implementation of IThreadPoolMgr::SetSize.

```
[soap_method]
STDMETHOD(SetSize)(
   int nNumThreads
);
```

See Also

Reference

CThreadPoolManager Class

IThreadPoolMgr::SetSize

Other Resources

CThreadPoolManager Members

CThreadPoolMgrObject Class

This class provides methods that can be used to implement IThreadPoolMgr by forwarding calls to an implementation of IThreadPoolConfig.

```
class CThreadPoolMgrObject
```

Remarks

This class is used in the implementation of the extension management services for the ISAPI extension's thread pool.

Requirements

Header: atlextmgmt.h

See Also

Other Resources

CThreadPoolMgrObject Members

CThreadPoolMgrObject Members

Methods

CThreadPoolMgrObject	The constructor.
GetSize	Delegates to IThreadPoolConfig::GetSize on the interface obtained from the service provider passed to Initialize.
Initialize	Call this method to initialize the object.
SetSize	Delegates to IThreadPoolConfig::SetSize on the interface obtained from the service provider passed to Initialize.

See Also

Reference

CThreadPoolMgrObject Class

CThreadPoolMgrObject Methods

For information about the methods in **CThreadPoolMgrObject**, see CThreadPoolMgrObject Members

CThreadPoolMgrObject::CThreadPoolMgrObject

The constructor.

```
CThreadPoolMgrObject( ) throw( );
```

See Also

Reference

CThreadPoolMgrObject Class

Other Resources

CThreadPoolMgrObject Members

CThreadPoolMgrObject::GetSize

Delegates to IThreadPoolConfig::GetSize on the interface obtained from the service provider passed to Initialize.

```
HRESULT GetSize(
   int * pnNumThreads
) throw( );
```

See Also

Reference

CThreadPoolMgrObject Class

Other Resources

CThreadPoolMgrObject Members

CThreadPoolMgrObject::Initialize

Call this method to initialize the object.

```
HTTP_CODE Initialize(
   IServiceProvider * pProvider
) throw( );
```

Parameters

pProvider

The service provider to be queried for the IThreadPoolConfig service.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

CThreadPoolMgrObject Class

Other Resources

CThreadPoolMgrObject Members

CThreadPoolMgrObject::SetSize

Delegates to IThreadPoolConfig::SetSize on the interface obtained from the service provider passed to Initialize.

```
HRESULT SetSize(
   int nNumThreads
) throw( );
```

See Also

Reference

CThreadPoolMgrObject Class

Other Resources

CThreadPoolMgrObject Members

CTransferServerContext Class

This class provides an implementation of IHttpServerContext for use by request handlers that have had control passed to them by a call to IIsapiExtension::TransferRequest.

```
class CTransferServerContext :
   public CComObjectRootEx< CComMultiThreadModel >,
   public CWrappedServerContext
```

Remarks

The implementation of IHttpServerContext provided by this class delegates most methods to the **IHttpServerContext** pointer passed to CTransferServerContext::Initialize. However, nonfunctional implementations of a number of methods are provided to prevent strange interactions between multiple handlers processing the same request.

The members list of this class provides a summary of the operations that cannot be performed or that return different results when executed from a request handler that has been passed a request by a call to **TransferRequest**.

Requirements

Header: atlisapi.h

See Also

Tasks

MantaWeb Sample: Online Personal Information Management

Reference

ForceLogin

ShowErrors

CComObjectRootEx Class

CWrappedServerContext Class

Other Resources

CTransferServerContext Members

CTransferServerContext Members

Methods

AsyncReadClient	Nonfunctional implementation of IHttpServerContext::AsyncReadClient.
AsyncWriteClient	Nonfunctional implementation of IHttpServerContext::AsyncWriteClient.
CTransferServerContext	The constructor.
GetAvailableBytes	Nonfunctional implementation of IHttpServerContext::GetAvailableBytes.
GetAvailableData	Nonfunctional implementation of IHttpServerContext::GetAvailableData.
GetContentType	Nonfunctional implementation of IHttpServerContext::GetContentType.
GetPathTranslated	Implementation of IHttpServerContext::GetPathTranslated. Returns the path obtained from the data passed to CTransferServerContext::Initialize.
GetQueryString	Implementation of IHttpServerContext::GetQueryString. Returns the query string obtained from the data passed to CTransferServerContext::Initialize.
GetRequestMethod	Implementation of IHttpServerContext::GetRequestMethod.
GetScriptPathTranslated	Implementation of IHttpServerContext::GetScriptPathTranslated. Returns the path obtained from the data passed to CTransferServerContext::Initialize.
GetTotalBytes	Nonfunctional implementation of IHttpServerContext::GetTotalBytes.
Initialize	This method is called by the ATL Server framework to initialize the object.
ReadClient	Nonfunctional implementation of IHttpServerContext::ReadClient.
WriteClient	Implementation of IHttpServerContext::WriteClient. Writes the data to the stream passed to CTransferServerContext::Initialize.

See Also

Reference

CTransferServerContext Class

Other Resources

CWrappedServerContext Members

CTransferServerContext Methods

For information about the methods in **CTransferServerContext**, see CTransferServerContext Members

CTransferServerContext::AsyncReadClient

Nonfunctional implementation of IHttpServerContext::AsyncReadClient.

```
BOOL AsyncReadClient(
   void * /* pvBuffer */,
   DWORD * /* pdwSize */
);
```

Return Value

Always returns **FALSE**.

See Also

Reference

CTransferServerContext Class

Other Resources

CTransferServerContext Members

CTransferServerContext::AsyncWriteClient

Nonfunctional implementation of IHttpServerContext::AsyncWriteClient.

```
BOOL AsyncWriteClient(
   void * /* pvBuffer */,
   DWORD * /* pdwBytes */
);
```

Return Value

Always returns **FALSE**.

See Also

Reference

CTransferServerContext Class

Other Resources

CTransferServerContext Members

CTransferServerContext::CTransferServerContext

The constructor.

```
CTransferServerContext( ) throw( );
```

Remarks

Initializes internal data members.

See Also

Reference

CTransferServerContext Class

Other Resources

CTransferServerContext Members

CTransferServerContext::GetAvailableBytes

Nonfunctional implementation of IHttpServerContext::GetAvailableBytes.

```
DWORD GetAvailableBytes( );
```

Return Value

Always returns 0.

See Also

Reference

CTransferServerContext Class

Other Resources

CTransferServerContext Members

CTransferServerContext::GetAvailableData

Nonfunctional implementation of IHttpServerContext::GetAvailableData.

```
BYTE* GetAvailableData( );
```

Return Value

Always returns **NULL**.

See Also

Reference

CTransferServerContext Class

Other Resources

CTransferServerContext Members

CTransferServerContext::GetContentType

Nonfunctional implementation of IHttpServerContext::GetContentType.

```
LPCSTR GetContentType( );
```

Return Value

Always returns **NULL**.

See Also

Reference

CTransferServerContext Class

Other Resources

CTransferServerContext Members

CTransferServerContext::GetPathTranslated

Implementation of IHttpServerContext::GetPathTranslated. Returns the path obtained from the data passed to CTransferServerContext::Initialize.

```
LPCSTR GetPathTranslated( );
```

See Also

Reference

CTransferServerContext Class

Other Resources

CTransferServerContext Members

CTransferServerContext::GetQueryString

Implementation of IHttpServerContext::GetQueryString. Returns the query string obtained from the data passed to CTransferServerContext::Initialize.

```
LPCSTR GetQueryString( );
```

See Also

Reference

CTransferServerContext Class

Other Resources

CTransferServerContext Members

CTransferServerContext::GetRequestMethod

Implementation of IHttpServerContext::GetRequestMethod.

```
LPCSTR GetRequestMethod( );
```

Return Value

Always returns "GET".

See Also

Reference

CTransferServerContext Class

Other Resources

CTransferServerContext Members

CTransferServerContext::GetScriptPathTranslated

Implementation of IHttpServerContext::GetScriptPathTranslated. Returns the path obtained from the data passed to CTransferServerContext::Initialize.

```
LPCSTR GetScriptPathTranslated( );
```

Example

See the ShowErrors Sample and the ForceLogin Sample.

See Also

Reference

CTransferServerContext Class

Other Resources

CTransferServerContext Members

CTransferServerContext::GetTotalBytes

Nonfunctional implementation of IHttpServerContext::GetTotalBytes.

```
DWORD GetTotalBytes( );
```

Return Value

Always returns 0.

See Also

Reference

CTransferServerContext Class

Other Resources

CTransferServerContext Members

CTransferServerContext::Initialize

This method is called by the ATL Server framework to initialize the object.

```
BOOL Initialize(
   LPCSTR szUrl,
   IWriteStream * pStream,
   IHttpServerContext * pParent
) throw( );
BOOL Initialize(
   CTransferServerContext * pOtherContext
);
```

Parameters

szUrl

The URL of the current request.

pStream

The stream to which the response should be written.

pParent, *pOtherContext*

The context of the parent request handler.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This method must be called to initialize the object before any of the IHttpServerContext methods may be called.

See Also

Reference

CTransferServerContext Class

Other Resources

CTransferServerContext Members

CTransferServerContext::ReadClient

Nonfunctional implementation of IHttpServerContext::ReadClient.

```
BOOL ReadClient(
   void * /* pvBuffer */,
   DWORD * /* pdwSize */
);
```

Return Value

Always returns **FALSE**.

See Also

Reference

CTransferServerContext Class

Other Resources

CTransferServerContext Members

CTransferServerContext::WriteClient

Implementation of IHttpServerContext::WriteClient. Writes the data to the stream passed to CTransferServerContext::Initialize.

```
BOOL WriteClient(
   void* pvBuffer,
   DWORD* pdwBytes
);
```

Example

See the MantaWeb Sample and the ShowErrors Sample.

See Also

Reference

CTransferServerContext Class

Other Resources

CTransferServerContext Members

CUrl Class

This class represents a URL. It allows you to manipulate each element of the URL independently of the others whether parsing an existing URL string or building a string from scratch.

```
class CUrl
```

Remarks

CUrl allows you to manipulate the fields of a URL, such as the path or port number. **CUrl** understands URLs of the following form:

<Scheme>://<UserName>:<Password>@<HostName>:<PortNumber>/<UrlPath><ExtraInfo>

(Some fields are optional.) For example, consider this URL:

http://someone:secret@www.microsoft.com:80/visualc/stuff.htm#contents

CUrl::CrackUrl parses it as follows:

- Scheme: "http" or ATL_URL_SCHEME_HTTP
- UserName: "someone"
- Password: "secret"
- HostName: "www.microsoft.com"
- PortNumber: 80
- UrlPath: "visualc/stuff.htm"
- ExtraInfo: "#contents"

To manipulate the UrlPath field (for instance), you would use GetUrlPath, GetUrlPathLength, and SetUrlPath. You would use CreateUrl to create the complete URL string.

Requirements

Header: atlutil.h

See Also

Other Resources

CUrl Members

ATL Server Classes

CUrl Members

Methods

Canonicalize	Call this method to convert the URL string to canonical form.
Clear	Call this method to clear all of the URL fields.
CrackUrl	Call this method to decode and parse the URL.
CreateUrl	Call this method to create the URL.
CUrl	The constructor.
~CUrl	The destructor.
GetExtraInfo	Call this method to get extra information (such as ?*text* or #*text*) from the URL.
GetExtraInfoLength	Call this method to get the length of the extra information (such as ?*text* or #*text*) to retrieve from the URL.
GetHostName	Call this method to get the host name from the URL.
GetHostNameLength	Call this method to get the length of the host name.
GetPassword	Call this method to get the password from the URL.
GetPasswordLength	Call this method to get the length of the password.
GetPortNumber	Call this method to get the port number in terms of ATL_URL_PORT.
GetScheme	Call this method to get the URL scheme.
GetSchemeName	Call this method to get the URL scheme name.
GetSchemeNameLength	Call this method to get the length of the URL scheme name.
GetUrlLength	Call this method to get the URL length.
GetUrlPath	Call this method to get the URL path.
GetUrlPathLength	Call this method to get the URL path length.
GetUserName	Call this method to get the user name from the URL.
GetUserNameLength	Call this method to get the length of the user name.
SetExtraInfo	Call this method to set the extra information (such as ?*text* or #*text*) of the URL.
SetHostName	Call this method to set the host name.
SetPassword	Call this method to set the password.

SetPortNumber	Call this method to set the port number in terms of ATL_URL_PORT.
SetScheme	Call this method to set the URL scheme.
SetSchemeName	Call this method to set the URL scheme name.
SetUrlPath	Call this method to set the URL path.
SetUserName	Call this method to set the user name.

Operators

operator =	Assigns the specified **CUrl** object to the current **CUrl** object.

See Also

Reference

CUrl Class

CUrl Methods

For information about the methods in **CUrl**, see CUrl Members.

CUrl::Canonicalize

Call this method to convert the URL string to canonical form.

```
inline BOOL Canonicalize(
   DWORD dwFlags = 0
) throw( );
```

Parameters

dwFlags

The flags that control canonicalization. If no flags are specified (*dwFlags* = 0), the method converts all unsafe characters and meta sequences (such as \.,\ .., and \...) to escape sequences. *dwFlags* can be one of the following values:

- ATL_URL_BROWSER_MODE: Does not encode or decode characters after "#" or "?" and does not remove trailing white space after "?". If this value is not specified, the entire URL is encoded and trailing white space is removed.
- ATL_URL _DECODE: Converts all %XX sequences to characters, including escape sequences, before the URL is parsed.
- ATL_URL _ENCODE_PERCENT: Encodes any percent signs encountered. By default, percent signs are not encoded.
- ATL_URL _ENCODE_SPACES_ONLY: Encodes spaces only.
- ATL_URL _NO_ENCODE: Does not convert unsafe characters to escape sequences.
- ATL_URL _NO_META: Does not remove meta sequences (such as "." and "..") from the URL.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

Converting to canonical form involves converting unsafe characters and spaces to escape sequences.

See Also

Reference

CUrl Class

Other Resources

CUrl Members

CUrl::Clear

Call this method to clear all of the URL fields.

```
inline void Clear( ) throw( );
```

See Also

Reference

CUrl Class

Other Resources

CUrl Members

CUrl::CrackUrl

Call this method to decode and parse the URL.

```
BOOL CrackUrl(
   LPCTSTR lpszUrl,
   DWORD dwFlags = 0
) throw( );
```

Parameters

lpszUrl

The URL.

dwFlags

Specify ATL_URL_DECODE or ATL_URL_ESCAPE to convert all escape characters in *lpszUrl* to their real values after parsing. (Before Visual C++ 2005, ATL_URL_DECODE converted all escape characters before parsing.)

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CUrl Class

Other Resources

CUrl Members

CUrl::CreateUrl

This method constructs a URL string from a CUrl object's component fields.

```
inline BOOL CreateUrl(
   LPTSTR lpszUrl,
   DWORD* pdwMaxLength,
   DWORD dwFlags = 0
) const throw( );
```

Parameters

lpszUrl

A string buffer to hold the complete URL string.

pdwMaxLength

The maximum length of the *lpszUrl* string buffer.

dwFlags

Specify ATL_URL_ESCAPE to convert all escape characters in *lpszUrl* to their real values.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This method appends its individual fields in order to construct the complete URL string using the following format:

<scheme>://<user>:<pass>@<domain>:<port><path><extra>

When calling this method, the *pdwMaxLength* parameter should initially contain the maximum length of the string buffer referenced by the *lpszUrl* parameter. The value of the *pdwMaxLength* parameter will be updated with the actual length of the URL string.

Example

This sample demonstrates creation of a CUrl object and retrieving its URL string

```
      CUrl url;

      // Set the CUrl contents
      url.CrackUrl( "http://someone:secret@www.microsoft.com:8080/visualc/stuff.htm#contents
" );

      ...

      // Obtain the length of the URL string and allocate a buffer to
      // hold its contents
      DWORD dwUrlLen = url.GetUrlLength() + 1;
      TCHAR* szUrl = new TCHAR[dwUrlLen];

      // Retrieve the contents of the CUrl object
      url.CreateUrl( szUrl, &dwUrlLen, 0L );

      // Cleanup
      delete[] szUrl;
```

See Also

Reference

CUrl Class
CUrl::CrackUrl
CUrl::GetUrlLength

Other Resources

CUrl::CUrl

The constructor.

```
CUrl( ) throw( );
CUrl(
   const CUrl& urlThat
) throw( );
```

Parameters

urlThat

The **CUrl** object to copy to create the URL.

See Also

Reference

CUrl Class

Other Resources

CUrl Members

CUrl::~CUrl

The destructor.

```
~CUrl( ) throw( );
```

See Also

Reference

CUrl Class

Other Resources

CUrl Members

CUrl::GetExtraInfo

Call this method to get extra information (such as ?*text* or #*text*) from the URL.

```
inline LPCTSTR GetExtraInfo( ) const throw( );
```

Return Value

Returns a string containing the extra information.

See Also

Reference

CUrl Class

CUrl::GetExtraInfoLength

CUrl::SetExtraInfo

Other Resources

CUrl Members

CUrl::GetExtraInfoLength

Call this method to get the length of the extra information (such as ?*text* or #*text*) to retrieve from the URL.

```
inline DWORD GetExtraInfoLength( ) const throw( );
```

Return Value

Returns the length of the string containing the extra information.

See Also

Reference

CUrl Class

CUrl::GetExtraInfo

Other Resources

CUrl Members

CUrl::GetHostName

Call this method to get the host name from the URL.

```
inline LPCTSTR GetHostName( ) const throw( );
```

Return Value

Returns the host name.

See Also

Reference

CUrl Class

CUrl::GetHostNameLength

CUrl::SetHostName

Other Resources

CUrl Members

CUrl::GetHostNameLength

Call this method to get the length of the host name.

```
inline DWORD GetHostNameLength( ) const throw( );
```

Return Value

Returns the host name length.

See Also

Reference

CUrl Class
CUrl::GetHostName
CUrl::SetHostName

Other Resources

CUrl Members

CUrl::GetPassword

Call this method to get the password from the URL.

```
inline LPCTSTR GetPassword( ) const throw( );
```

Return Value

Returns the password.

See Also

Reference

CUrl Class

CUrl::GetPasswordLength

CUrl::SetPassword

Other Resources

CUrl Members

CUrl::GetPasswordLength

Call this method to get the length of the password.

```
inline DWORD GetPasswordLength( ) const throw( );
```

Return Value

Returns the password length.

See Also

Reference

CUrl Class

CUrl::GetPassword

Other Resources

CUrl Members

CUrl::GetPortNumber

Call this method to get the port number.

```
inline ATL_URL_PORT GetPortNumber( ) const throw( );
```

Return Value

Returns the port number.

See Also

Reference

CUrl Class

CUrl::SetPortNumber

Other Resources

CUrl Members

CUrl::GetScheme

Call this method to get the URL scheme.

```
inline ATL_URL_SCHEME GetScheme( ) const throw( );
```

Return Value

Returns the ATL_URL_SCHEME value describing the scheme of the URL.

See Also

Reference

CUrl Class
CUrl::GetSchemeName
CUrl::SetScheme
ATL_URL_SCHEME

Other Resources

CUrl Members

CUrl::GetSchemeName

Call this method to get the URL scheme name.

```
inline LPCTSTR GetSchemeName( ) const throw( );
```

Return Value

Returns the URL scheme name (such as "http" or "ftp").

See Also

Reference

CUrl Class
CUrl::GetSchemeNameLength
CUrl::GetScheme
CUrl::SetSchemeName

Other Resources

CUrl Members

CUrl::GetSchemeNameLength

Call this method to get the length of the URL scheme name.

```
inline DWORD GetSchemeNameLength( ) const throw( );
```

Return Value

Returns the URL scheme name length.

See Also

Reference

CUrl Class

CUrl::GetSchemeName

Other Resources

CUrl Members

CUrl::GetUrlLength

Call this method to get the URL length.

```
inline DWORD GetUrlLength( ) const throw( );
```

Return Value

Returns the URL length.

See Also

Reference

CUrl Class

CUrl::CreateUrl

Other Resources

CUrl Members

CUrl::GetUrlPath

Call this method to get the URL path.

```
inline LPCTSTR GetUrlPath( ) const throw( );
```

Return Value

Returns the URL path.

See Also

Reference

CUrl Class
CUrl::GetUrlPathLength
CUrl::SetUrlPath

Other Resources

CUrl Members

CUrl::GetUrlPathLength

Call this method to get the URL path length.

```
inline DWORD GetUrlPathLength( ) const throw( );
```

Return Value

Returns the URL path length.

See Also

Reference

CUrl Class

CUrl::GetUrlPath

Other Resources

CUrl Members

CUrl::GetUserName

Call this method to get the user name from the URL.

```
inline LPCTSTR GetUserName( ) const throw( );
```

Return Value

Returns the user name.

See Also

Reference

CUrl Class

CUrl::GetUserNameLength

CUrl::SetUserName

Other Resources

CUrl Members

CUrl::GetUserNameLength

Call this method to get the length of the user name.

```
inline DWORD GetUserNameLength( ) const throw( );
```

Return Value

Returns the user name length.

See Also

Reference

CUrl Class

CUrl::GetUserName

Other Resources

CUrl Members

CUrl::SetExtraInfo

Call this method to set the extra information (such as ?*text* or #*text*) of the URL.

```
inline BOOL SetExtraInfo(
   LPCTSTR lpszInfo
) throw( );
```

Parameters

lpszInfo

The string containing the extra information to include in the URL.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CUrl Class

CUrl::GetExtraInfo

Other Resources

CUrl Members

CUrl::SetHostName

Call this method to set the host name.

```
inline BOOL SetHostName(
   LPCTSTR lpszHost
) throw( );
```

Parameters

lpszHost

The host name.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CUrl Class

CUrl::GetHostName

Other Resources

CUrl Members

CUrl::SetPassword

Call this method to set the password.

```
inline BOOL SetPassword(
   LPCTSTR lpszPass
) throw( );
```

Parameters

lpszPass

The password.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CUrl Class

CUrl::GetPassword

Other Resources

CUrl Members

CUrl::SetPortNumber

Call this method to set the port number.

```
inline BOOL SetPortNumber(
   ATL_URL_PORT nPrt
) throw( );
```

Parameters

nPrt

The port number.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CUrl Class

CUrl::GetPortNumber

ATL_URL_PORT

Other Resources

CUrl Members

CUrl::SetScheme

Call this method to set the URL scheme.

```
inline BOOL SetScheme(
   ATL_URL_SCHEME nScheme
) throw( );
```

Parameters

nScheme

One of the ATL_URL_SCHEME values for the scheme.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

You can also set the scheme by name (see CUrl::SetSchemeName).

See Also

Reference

CUrl Class

CUrl::GetScheme

Other Resources

CUrl Members

CUrl::SetSchemeName

Call this method to set the URL scheme name.

```
inline BOOL SetSchemeName(
   LPCTSTR lpszSchm
) throw( );
```

Parameters

lpszSchm

The URL scheme name.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

You can also set the scheme by using an ATL_URL_SCHEME constant (see CUrl::SetScheme).

See Also

Reference

CUrl Class

CUrl::GetSchemeName

Other Resources

CUrl Members

CUrl::SetUrlPath

Call this method to set the URL path.

```
inline BOOL SetUrlPath(
   LPCTSTR lpszPath
) throw( );
```

Parameters

lpszPath

The URL path.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CUrl Class

CUrl::GetUrlPath

Other Resources

CUrl Members

CUrl::SetUserName

Call this method to set the user name.

```
inline BOOL SetUserName(
   LPCTSTR lpszUser
) throw( );
```

Parameters

lpszUser

The user name.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CUrl Class

CUrl::GetUserName

Other Resources

CUrl Members

CUrl Operators

For information about the operators in **CUrl**, see CUrl Members.

CUrl::operator =

Assigns the specified **CUrl** object to the current **CUrl** object.

```
CUrl & operator =(
   const CUrl & urlThat
) throw();
```

Parameters

urlThat

The **CUrl** object to copy into the current object.

Return Value

Returns a reference to the current object.

See Also

Reference

CUrl Class

Other Resources

CUrl Members

CValidateContext Class

This class represents a collection of validation status codes. Use this class in combination with CValidateObject to validate forms variables, cookies, or query parameters and to build up a collection of failures (and, optionally, successes).

```
class CValidateContext
```

Remarks

This class is designed, in conjunction with CValidateObject, to allow you to collect validation failures and use that information to return detailed responses to clients to help them correct errors in their input.

Call AddResult to add a validation status to the collection, SetResultAt to change the type of a validation status code, or GetResultAt to retrieve the items from the collection.

Requirements

Header: atlisapi.h

Example

See the Input Sample and the Mailer Sample.

See Also

Reference

CValidateObject Class

Other Resources

CValidateContext Members

ATL Server Classes

CValidateContext Members

Methods

AddResult	Call this method to add a validation result to the collection managed by this object.
CValidateContext	The constructor.
GetResultAt	Call this method to retrieve the name and type of a validation result based on its index in the collection.
GetResultCount	Call this method to get the number of validation results managed by this collection.
ParamsOK	Call this method to determine whether the collection contains any validation failures.
SetResultAt	Call this method to change the type of a validation result based on its name.

See Also

Reference

CValidateContext Class

CValidateContext Methods

For information about the methods in **CValidateContext**, see CValidateContext Members.

CValidateContext::AddResult

Call this method to add a validation result to the collection managed by this object.

```
bool AddResult(
   LPCSTR szName,
   DWORD type,
   bool bOnlyFailures = true
) throw( );
```

Parameters

szName

The name of the result being added to the collection.

type

The validation status code corresponding to *szName*.

bOnlyFailures

Set this parameter to true to prevent success results from being added to the collection. This parameter makes it easy to keep track of failures without worrying about successes.

Return Value

Returns true if the result was added to the collection, false otherwise.

See Also

Reference

CValidateContext Class

Other Resources

CValidateContext Members

CValidateContext::CValidateContext

The constructor.

```
CValidateContext(
   DWORD dwFlags = 0
) throw( );
```

Parameters

dwFlags

The flags controlling the behavior of this object.

Remarks

If *dwFlags* is zero, only error validation status codes will cause CValidateContext::ParamsOK to return **false**.

If dwFlags includes **CValidateContext::ATL_EMPTY_PARAMS_ARE_FAILURES**, validation status codes of type **VALIDATION_S_EMPTY** will also cause CValidateContext::ParamsOK to return **false**.

See Also

Reference

CValidateContext Class

Other Resources

CValidateContext Members

CValidateContext::GetResultAt

Call this method to retrieve the name and type of a validation result based on its index in the collection.

```
bool GetResultAt(
   int i,
   CStringA& strName,
   DWORD& type
) throw( );
```

Parameters

i

The zero-based index of a result managed by this collection.

strName

On success, returns the name of the result with index *i*.

type

On success, returns the validation status code of the result with index *i*.

Return Value

Returns true on success, false on failure.

Remarks

Call CValidateContext::GetResultCount to get the total number of items in the collection.

Example

See the Mailer Sample.

See Also

Reference

CValidateContext Class

CValidateContext::GetResultCount

Other Resources

CValidateContext Members

CValidateContext::GetResultCount

Call this method to get the number of validation results managed by this collection.

```
int GetResultCount( ) throw( );
```

Return Value

Returns the number of validation results managed by this collection.

Example

See the Mailer Sample.

See Also

Reference

CValidateContext Class

CValidateContext::GetResultAt

Other Resources

CValidateContext Members

CValidateContext::ParamsOK

Call this method to determine whether the collection contains any validation failures.

```
bool ParamsOK( ) throw( );
```

Return Value

Returns true if there are no validation failures in the collection, false otherwise.

Example

See the Input Sample and the Mailer Sample.

See Also

Reference

CValidateContext Class

Other Resources

CValidateContext Members

CValidateContext::SetResultAt

Call this method to change the type of a validation result based on its name.

```
bool SetResultAt(
   LPCSTR szName,
   DWORD type
);
```

Parameters

szName

The name of the result to be changed.

type

The new validation status code of the result *szName*.

Return Value

Returns true on success, false on failure.

See Also

Reference

CValidateContext Class

Other Resources

CValidateContext Members

CValidateObject Class

This class provides methods for retrieving and validating named values.

```
template <
   class TLookupClass,
   class TValidator = CAtlValidator
>
class CValidateObject
```

Parameters

TLookupClass

The class providing named values to be validated. *TLookupClass* must derive from **CValidateObject<** *TLookupClass* **>** and provide a method named **Lookup** with the following signature:

```
LPCSTR Lookup(LPCSTR szName);
```

TValidator

The class providing the static function that determines whether a value lies within a specified range. Defaults to CAtlValidator.

Remarks

CValidateObject provides the means of retrieving named values from class *TLookupClass* converted to data types chosen by you. You can validate the values by specifying a range for numeric values or by specifying a minimum and maximum length for string values.

Call one of the CValidateObject::Exchange overloads to retrieve a named value converted to your chosen data type. Call one of the CValidateObject::Validate specializations to retrieve a named value converted to your chosen data type and validated against a minimum and maximum value or length supplied by you.

Requirements

Header: atlisapi.h

See Also

Tasks

MantaWeb Sample: Online Personal Information Management
Input Sample: Demonstrates User Input Validation On Client And Server
RegExp Sample: Demonstrates ATL Regular Expression Classes

Reference

CValidateContext Class

Other Resources

CValidateObject Members
ATL Server Classes

CValidateObject Members

Methods

ConvertNumber	Call this method to convert a string to a numeric type.
Exchange	Call this method to retrieve a named value converted to the data type of your choice.
Validate	Call this method to retrieve a named value converted to the data type of your choice and have it validated aga inst a minimum and maximum value or length supplied by you.

See Also

Reference

CValidateObject Class

CValidateObject Methods

For information about the methods in **CValidateObject**, see CValidateObject Members.

CValidateObject::ConvertNumber

Call this method to convert a string to a numeric type.

```
DWORD ConvertNumber(
   LPCSTR szVal,
   ULONGLONG * pnVal
) const throw( );
DWORD ConvertNumber(
   LPCSTR szVal,
   LONGLONG * pnVal
) const throw( );
DWORD ConvertNumber(
   LPCSTR szVal,
   double * pdblVal
) const throw( );
DWORD ConvertNumber(
   LPCSTR szVal,
   int * pnVal
) const throw( );
DWORD ConvertNumber(
   LPCSTR szVal,
   unsigned int * pnVal
) const throw( );
DWORD ConvertNumber(
   LPCSTR szVal,
   long * pnVal
) const throw( );
DWORD ConvertNumber(
   LPCSTR szVal,
   unsigned long * pnVal
) const throw( );
DWORD ConvertNumber(
   LPCSTR szVal,
   short * pnVal
) const throw( );
DWORD ConvertNumber(
   LPCSTR szVal,
   unsigned short * pnVal
) const throw( );
```

Parameters

szVal

The string containing the value to be converted.

pnVal, *pdblVal*

[out] Address of the variable that, on success, receives the numeric value converted from the string *szVal*.

Return Value

Returns one of the validation status codes describing whether the conversion succeeded or how it failed.

Remarks

This method attempts to convert the string holding a decimal number to a numeric type using strtol and related functions.

See Also

Reference

CValidateObject Class

Other Resources

CValidateObject Members

CValidateObject::Exchange

Call this method to retrieve a named value converted to the data type of your choice.

```
template <class T>
ATL_NOINLINE DWORD Exchange(
   LPCSTR szParam,
   T* pValue,
   CValidateContext* pContext = NULL
) const throw();
template< >
ATL_NOINLINE DWORD Exchange(
   LPCSTR szParam,
   bool* pbValue,
   CValidateContext* pContext
) const throw();
template< >
ATL_NOINLINE DWORD Exchange(
   LPCSTR szParam,
   CString* pstrValue,
   CValidateContext* pContext
) const throw();
template< >
ATL_NOINLINE DWORD Exchange(
   LPCSTR szParam,
   GUID* pValue,
   CValidateContext* pContext
) const throw();
template< >
ATL_NOINLINE DWORD Exchange(
   LPCSTR szParam,
   LPCSTR* ppszValue,
   CValidateContext * pContext
) const throw( );
```

Parameters

szParam

The name of the value to be retrieved.

pValue, *pbValue*, *pstrValue*, *ppszValue*

A pointer to the variable to receive the value on success.

pContext

A pointer to the validation context used to collect failures.

Return Value

Returns one of the validation status codes describing whether the exchange succeeded or how it failed.

Remarks

This method can be used to convert name-value pairs such as those associated with HTTP requests (query parameters, form variables, or cookie values). Pass a pointer to a validation context object if you want to add failures to the collection managed by that object.

Note that the **bool** specialization is designed specifically for use with checkboxes in HTML forms. This overload will set ***pValue** to **true** if a value is present even if a conversion of the contents of the value would not yield a boolean value. This is because of the way in which checkboxes are submitted as part of an HTML form.

Example

See the MantaWeb Sample and the RegExp Sample.

See Also

Reference

CValidateObject Class

CValidateObject::Validate

CValidateContext Class

Other Resources

CValidateObject Members

CValidateObject::Validate

Call this method to retrieve a named value converted to the data type of your choice and have it validated against a minimum and maximum value or length supplied by you.

```
template <class T, class TCompType>
ATL_NOINLINE DWORD Validate(
   LPCSTR Param,
   T *pValue,
   TCompType nMinValue,
   TCompType nMaxValue,
   CValidateContext *pContext = NULL
) const throw();
template< >
ATL_NOINLINE DWORD Validate(
   LPCSTR Param,
   CString* pstrValue,
   int nMinChars,
   int nMaxChars,
   CValidateContext *pContext
) const throw();
template< >
ATL_NOINLINE DWORD Validate(
   LPCSTR Param,
   LPCSTR* ppszValue,
   int nMinChars,
   int nMaxChars,
   CValidateContext * pContext
) const throw( );
template< >
ATL_NOINLINE DWORD Validate(
   LPCSTR Param,
   double* pdblValue,
   double dblMinValue,
   double dblMaxValue,
   CValidateContext *pContext
) const throw();
```

Parameters

T

The destination type of the value being validated.

TCompType

The type of the minimum and maximum values against which the value is to be validated.

Param

The name of the value to be retrieved.

pValue, *pstrValue*, *ppszValue*, *pdblValue*

A pointer to the variable to receive the value on success.

pContext

A pointer to the validation context used to collect failures.

nMinValue

The minimum acceptable value.

nMaxValue

The maximum acceptable value.

nMinChars

The minimum acceptable length.

nMaxChars

The maximum acceptable length.

Return Value

Returns one of the validation status codes describing whether validation succeeded or how it failed.

Remarks

This method can be used to convert and validate name-value pairs such as those associated with HTTP requests (query parameters, form variables, or cookie values). The numeric specializations validate the minimum and maximum value. The string specializations validate the minimum and maximum length.

The **double** specialization uses ATL_EPSILON to perform the comparison. You can redefine this value to get an acceptable tolerance on the validation.

Pass a pointer to a validation context object if you want to add failures to the collection managed by that object.

Note that you can validate the value of a parameter without storing its value by passing NULL for the second parameter. However, if you pass NULL for the second parameter, *pValue*, make sure you cast the NULL to a type so that the compiler will call the correct specialization of **Validate**.

Example

See the MantaWeb Sample and the Input Sample.

See Also

Reference

CValidateObject Class

CValidateObject::Exchange

Other Resources

CValidateObject Members

CVerifyAuth Class

This class allows you to authorize the client making a request to the application.

```
template <
   class T
>
class CVerifyAuth
```

Parameters

T

The class derived from **CVerifyAuth**.

Remarks

Call CheckAccount to determine whether the user making the current request is a member of a particular group, or call IsAuthorized to authorize the user and send back an error response (assuming an appropriate class such as CDefaultAuth is derived from **CVerifyAuth**).

Requirements

Header: atlisapi.h

See Also

Other Resources

CVerifyAuth Members

ATL Server Classes

CVerifyAuth Members

Methods

CheckAccount	Override this virtual method to check whether the user is a member of the specified group.
CheckAuthAccount	Call this method to check whether the user is a member of the specified group.
HandleError	Override this non-virtual method to send an error response to the client in the event of an authorization failure.
IsAuthorized	Call this method to check whether the user making a request is authorized.

See Also

Reference

CVerifyAuth Class

CVerifyAuth Methods

For information about the methods in **CVerifyAuth**, see CVerifyAuth Members.

CVerifyAuth::CheckAccount

Override this virtual method to check whether the user is a member of the specified group.

```
virtual bool CheckAccount(
   IHttpServerContext* pContext,
   const SID* psidAuthGroup
) throw( );
```

Parameters

pContext

The server context that contains the information about the user.

psidAuthGroup

The security identifier for the group.

Return Value

Return **true** if the user is an authorized member of the group specified by *psidAuthGroup*, **false** otherwise. (The implementation in this class always returns **false**.)

Remarks

This method is called from CVerifyAuth::IsAuthorized. The default implementation always returns **false** without any checks, but derived classes can override the implementation to call CVerifyAuth::CheckAuthAccount or provide a custom implementation.

See Also

Reference

CVerifyAuth Class

Other Resources

CVerifyAuth Members

CVerifyAuth::CheckAuthAccount

Call this method to check whether the user is a member of the specified group.

```
bool CheckAuthAccount(
   IHttpServerContext* pContext,
   const SID* psidAuthGroup
) throw( );
```

Parameters

pContext

The server context that contains information about the user.

psidAuthGroup

The security identifier for the group.

Return Value

Returns **true** if an impersonation token can be obtained for the user and the account being impersonated is a member of the group identified by *psidAuthGroup*. Otherwise, returns **false**.

See Also

Reference

CVerifyAuth Class

Other Resources

CVerifyAuth Members

CVerifyAuth::HandleError

Override this non-virtual method to send an error response to the client in the event of an authorization failure.

```
HTTP_CODE HandleError(
   AtlServerRequest * pRequestInfo
) throw( );
```

Parameters

pRequestInfo

The information about the current request.

Return Value

Return HTTP_SUCCESS or another ATL Server status code indicating success or failure.

This implementation always returns HTTP_FAIL.

Remarks

This method is called by IsAuthorized in the event of an authorization failure.

See Also

Reference

CVerifyAuth Class

Other Resources

CVerifyAuth Members

CVerifyAuth::IsAuthorized

Call this method to check whether the user is authorized to make the current request.

```
HTTP_CODE IsAuthorized(
   AtlServerRequest* pInfo,
   const SID* psidAuthGroup
) throw( );
```

Parameters

pInfo

The information about the current request.

psidAuthGroup

The security identifier for the group in which to test the user for membership. Can be NULL, in which case the method only checks that either NTLM or Negotiate authentication were used to identify the client.

Return Value

Returns HTTP_SUCCESS if the client is authorized, HTTP_UNAUTHORIZED or the return value of HandleError if the client is not authorized, or HTTP_FAIL or another ATL Server status code in the event of a failure.

Remarks

This method gets the type of authentication used to make the current request. If neither NTLM nor Negotiate were used, the method returns HTTP_UNAUTHORIZED. If either of those authentication protocols were used, the method calls CheckAccount if *psidAuthGroup* is not NULL, and HandleError if **CheckAccount** returns **false**.

See Also

Reference

CVerifyAuth Class

Other Resources

CVerifyAuth Members

CWorkerThread Class

This class creates a worker thread or uses an existing one, waits on one or more kernel object handles, and executes a specified client function when one of the handles is signaled.

```
template <
   class ThreadTraits = DefaultThreadTraits
>
class CWorkerThread
```

Parameters

ThreadTraits

The class providing the thread creation function, such as CRTThreadTraits or Win32ThreadTraits.

Remarks

To use CWorkerThread

1. Create an instance of this class.
2. Call CWorkerThread::Initialize.
3. Call CWorkerThread::AddHandle with the handle of a kernel object and a pointer to an implementation of IWorkerThreadClient.

   – or –

   Call CWorkerThread::AddTimer with a pointer to an implementation of IWorkerThreadClient.
4. Implement IWorkerThreadClient::Execute to take some action when the handle or timer is signaled.
5. To remove an object from the list of waitable objects, call CWorkerThread::RemoveHandle.
6. To terminate the thread, call CWorkerThread::Shutdown.

Requirements

Header: atlutil.h

See Also

Reference

DefaultThreadTraits

IWorkerThreadClient Interface

Concepts

Multithreading: Creating Worker Threads

Other Resources

CWorkerThread Members

ATL Server Classes

CWorkerThread Members

Methods

AddHandle	Call this method to add a waitable object's handle to the list maintained by the worker thread.
AddTimer	Call this method to add a periodic waitable timer to the list maintained by the worker thread.
CWorkerThread	The constructor for the worker thread.
~CWorkerThread	The destructor for the worker thread.
GetThreadHandle	Call this method to get the thread handle of the worker thread.
GetThreadId	Call this method to get the thread ID of the worker thread.
Initialize	Call this method to initialize the worker thread.
RemoveHandle	Call this method to remove a handle from the list of waitable objects.
Shutdown	Call this method to shut down the worker thread.

See Also

Reference

CWorkerThread Class

CWorkerThread Methods

For information about the methods in **CWorkerThread**, see CWorkerThread Members.

CWorkerThread::AddHandle

Call this method to add a waitable object's handle to the list maintained by the worker thread.

```
HRESULT AddHandle(
   HANDLE hObject,
   IWorkerThreadClient* pClient,
   DWORD_PTR dwParam
) throw( );
```

Parameters

hObject

The handle to a waitable object.

pClient

The pointer to the IWorkerThreadClient interface on the object to be called when the handle is signaled.

dwParam

The parameter to be passed to IWorkerThreadClient::Execute when the handle is signaled.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

IWorkerThreadClient::Execute will be called through *pClient* when the handle, *hObject*, is signaled.

See Also

Reference

CWorkerThread Class

Other Resources

CWorkerThread Members

CWorkerThread::AddTimer

Call this method to add a periodic waitable timer to the list maintained by the worker thread.

```
HRESULT AddTimer(
   DWORD dwInterval,
   IWorkerThreadClient* pClient,
   DWORD_PTR dwParam,
   HANDLE* phTimer
) throw( );
```

Parameters

dwInterval

Specifies the period of the timer in milliseconds.

pClient

The pointer to the IWorkerThreadClient interface on the object to be called when the handle is signaled.

dwParam

The parameter to be passed to IWorkerThreadClient::Execute when the handle is signaled.

phTimer

[out] Address of the HANDLE variable that, on success, receives the handle to the newly created timer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

IWorkerThreadClient::Execute will be called through *pClient* when the timer is signaled.

Pass the timer handle from *phTimer* to CWorkerThread::RemoveHandle to close the timer.

See Also

Reference

CWorkerThread Class

Other Resources

CWorkerThread Members

CWorkerThread::CWorkerThread

The constructor.

```
CWorkerThread( ) throw( );
```

See Also

Reference

CWorkerThread Class

Other Resources

CWorkerThread Members

CWorkerThread::~CWorkerThread

The destructor.

```
~CWorkerThread( ) throw( );
```

Remarks

Calls CWorkerThread::Shutdown.

See Also

Reference

CWorkerThread Class

Other Resources

CWorkerThread Members

CWorkerThread::GetThreadHandle

Call this method to get the thread handle of the worker thread.

```
HANDLE GetThreadHandle( ) throw( );
```

Return Value

Returns the thread handle or NULL if the worker thread has not been initialized.

See Also

Reference

CWorkerThread Class

Other Resources

CWorkerThread Members

CWorkerThread::GetThreadId

Call this method to get the thread ID of the worker thread.

```
DWORD GetThreadId( ) throw( );
```

Return Value

Returns the thread ID or NULL if the worker thread has not been initialized.

See Also

Reference

CWorkerThread Class

Other Resources

CWorkerThread Members

CWorkerThread::Initialize

Call this method to initialize the worker thread.

```
HRESULT Initialize( ) throw( );
HRESULT Initialize(
   CWorkerThread< ThreadTraits > * pThread
) throw( );
```

Parameters

pThread

An existing worker thread.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method should be called to initialize the object after creation or after a call to CWorkerThread::Shutdown.

To have two or more **CWorkerThread** objects use the same worker thread, initialize one of them without passing any arguments then pass a pointer to that object to the **Initialize** methods of the others. The objects initialized using the pointer should be shut down before the object used to initialize them.

See CWorkerThread::Shutdown for information on how that method's behavior changes when initialized using a pointer to an existing object.

See Also

Reference

CWorkerThread Class

Other Resources

CWorkerThread Members

CWorkerThread::RemoveHandle

Call this method to remove a handle from the list of waitable objects.

```
HRESULT RemoveHandle(
   HANDLE hObject
) throw( );
```

Parameters

hObject

The handle to remove.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

When the handle is removed IWorkerThreadClient::CloseHandle will be called on the associated object that was passed to AddHandle. If this call fails, **CWorkerThread** will call the Windows CloseHandle function on the handle.

See Also

Reference

CWorkerThread Class

Other Resources

CWorkerThread Members

CWorkerThread::Shutdown

Call this method to shut down the worker thread.

```
HRESULT Shutdown(
   DWORD dwWait = ATL_WORKER_THREAD_WAIT
) throw( );
```

Parameters

dwWait

The time in milliseconds to wait for the worker thread to shut down.

Return Value

Returns S_OK on success, or an error HRESULT on failure, such as if the timeout value, *dwWait*, is exceeded.

Remarks

To reuse the object, call CWorkerThread::Initialize after calling this method.

Note that calling **Shutdown** on an object initialized with a pointer to another **CWorkerThread** object has no effect and always returns S_OK.

See Also

Reference

CWorkerThread Class

ATL_WORKER_THREAD_WAIT

Other Resources

CWorkerThread Members

CWrappedServerContext Class

This class provides an implementation of IHttpServerContext that delegates all the methods to another object implementing the same interface.

```
class CWrappedServerContext:
    public IHttpServerContext
```

Remarks

The object which provides the underlying implementation of the IHttpServerContext methods can be passed to the **CWrappedServerContext** constructor.

Requirements

Header: atlisapi.h

Example

See the following samples:

- ForceLogin Sample
- MantaWeb Sample
- NotModified Sample
- ShowErrors Sample
- ShowImage Sample
- ShowUser Sample

See Also

Reference

IHttpServerContext Interface

Other Resources

CWrappedServerContext Members

CWrappedServerContext Members

Methods

AppendToLog	This method calls the corresponding IHttpServerContext method on m_spParent.
AsyncReadClient	This method calls the corresponding IHttpServerContext method on m_spParent.
AsyncWriteClient	This method calls the corresponding IHttpServerContext method on m_spParent.
CWrappedServerContext	The constructor.
DoneWithSession	This method calls the corresponding IHttpServerContext method on m_spParent.
GetAvailableBytes	This method calls the corresponding IHttpServerContext method on m_spParent.
GetAvailableData	This method calls the corresponding IHttpServerContext method on m_spParent.
GetContentType	This method calls the corresponding IHttpServerContext method on m_spParent.
GetImpersonationToken	This method calls the corresponding IHttpServerContext method on m_spParent.
GetPathInfo	This method calls the corresponding IHttpServerContext method on m_spParent.
GetPathTranslated	This method calls the corresponding IHttpServerContext method on m_spParent.
GetQueryString	This method calls the corresponding IHttpServerContext method on m_spParent.
GetRequestMethod	This method calls the corresponding IHttpServerContext method on m_spParent.
GetScriptPathTranslated	This method calls the corresponding IHttpServerContext method on m_spParent.
GetServerVariable	This method calls the corresponding IHttpServerContext method on m_spParent.
GetTotalBytes	This method calls the corresponding IHttpServerContext method on m_spParent.
MapUrlToPathEx	This method calls the corresponding IHttpServerContext method on m_spParent.
ReadClient	This method calls the corresponding IHttpServerContext method on m_spParent.
RequestIOCompletion	This method calls the corresponding IHttpServerContext method on m_spParent.
SendRedirectResponse	This method calls the corresponding IHttpServerContext method on m_spParent.
SendResponseHeader	This method calls the corresponding IHttpServerContext method on m_spParent.
TransmitFile	This method calls the corresponding IHttpServerContext method on m_spParent.
WriteClient	This method calls the corresponding IHttpServerContext method on m_spParent.

Data Members

m_spParent	The parent object providing the useful implementation of the IHttpServerContext methods.

See Also

Reference

CWrappedServerContext Class

Other Resources

IHttpServerContext Members

CWrappedServerContext Methods

For information about the methods in **CWrappedServerContext**, see CWrappedServerContext Members.

CWrappedServerContext::AppendToLog

This method calls the corresponding IHttpServerContext method on m_spParent.

```
BOOL AppendToLog(
   LPCSTR szMessage,
   DWORD* pdwLen
);
```

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::AsyncReadClient

This method calls the corresponding IHttpServerContext method on m_spParent.

```
BOOL AsyncReadClient(
   void* pvBuffer,
   DWORD* pdwSize
);
```

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::AsyncWriteClient

This method calls the corresponding IHttpServerContext method on m_spParent.

```
BOOL AsyncWriteClient(
   void* pvBuffer,
   DWORD* pdwBytes
);
```

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::CWrappedServerContext

The constructor.

```
CWrappedServerContext( ) throw( );
CWrappedServerContext(
   IHttpServerContext* pParent
) throw( );
```

Parameters

pParent

The object to which all of the IHttpServerContext methods will be delegated.

Remarks

The **IHttpServerContext** pointer passed to the constructor is stored in m_spParent.

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::DoneWithSession

This method calls the corresponding IHttpServerContext method on m_spParent.

```
BOOL DoneWithSession(
   DWORD dwHttpStatusCode
);
```

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::GetAvailableBytes

This method calls the corresponding IHttpServerContext method on m_spParent.

```
DWORD GetAvailableBytes( );
```

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::GetAvailableData

This method calls the corresponding IHttpServerContext method on m_spParent.

```
BYTE* GetAvailableData( );
```

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::GetContentType

This method calls the corresponding IHttpServerContext method on m_spParent.

```
LPCSTR GetContentType( );
```

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::GetImpersonationToken

This method calls the corresponding IHttpServerContext method on m_spParent.

```
BOOL GetImpersonationToken(
   HANDLE* pToken
);
```

Example

See the ShowUser Sample.

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::GetPathInfo

This method calls the corresponding IHttpServerContext method on m_spParent.

```
LPCSTR GetPathInfo( );
```

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::GetPathTranslated

This method calls the corresponding IHttpServerContext method on m_spParent.

```
LPCSTR GetPathTranslated( );
```

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::GetQueryString

This method calls the corresponding IHttpServerContext method on m_spParent.

```
LPCSTR GetQueryString( );
```

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::GetRequestMethod

This method calls the corresponding IHttpServerContext method on m_spParent.

```
LPCSTR GetRequestMethod( );
```

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::GetScriptPathTranslated

This method calls the corresponding IHttpServerContext method on m_spParent.

```
LPCSTR GetScriptPathTranslated( );
```

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::GetServerVariable

This method calls the corresponding IHttpServerContext method on m_spParent.

```
BOOL GetServerVariable(
   LPCSTR pszVariableName,
   LPSTR pvBuffer,
   DWORD* pdwSize
);
```

Example

See the NotModified Sample and the ShowImage Sample.

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::GetTotalBytes

This method calls the corresponding IHttpServerContext method on m_spParent.

```
DWORD GetTotalBytes( );
```

Example

See the ForceLogin Sample.

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::MapUrlToPathEx

This method calls the corresponding IHttpServerContext method on m_spParent.

```
BOOL MapUrlToPathEx(
   LPCSTR szLogicalPath,
   DWORD dwLen,
   HSE_URL_MAPEX_INFO* pumInfo
);
```

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::ReadClient

This method calls the corresponding IHttpServerContext method on m_spParent.

```
BOOL ReadClient(
   void* pvBuffer,
   DWORD* pdwSize
);
```

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::RequestIOCompletion

This method calls the corresponding IHttpServerContext method on m_spParent.

```
BOOL RequestIOCompletion(
   PFN_HSE_IO_COMPLETION pfn,
   DWORD* pdwContext
);
```

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::SendRedirectResponse

This method calls the corresponding IHttpServerContext method on m_spParent.

```
BOOL SendRedirectResponse(
   LPCSTR pszRedirectUrl
);
```

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::SendResponseHeader

This method calls the corresponding IHttpServerContext method on m_spParent.

```
BOOL SendResponseHeader(
   LPCSTR pszHeader,
   LPCSTR pszStatusCode,
   BOOL fKeepConn
);
```

Example

See the ShowErrors Sample.

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::TransmitFile

This method calls the corresponding IHttpServerContext method on m_spParent.

```
BOOL TransmitFile(
   HANDLE hFile,
   PFN_HSE_IO_COMPLETION pfn,
   void* pContext,
   LPCSTR szStatusCode,
   DWORD dwBytesToWrite,
   DWORD dwOffset,
   void* pvHead,
   DWORD dwHeadLen,
   void* pvTail,
   DWORD dwTailLen,
   DWORD dwFlags
);
```

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext::WriteClient

This method calls the corresponding IHttpServerContext method on m_spParent.

```
BOOL WriteClient(
   void* pvBuffer,
   DWORD* pdwBytes
);
```

Example

See the MantaWeb Sample and the ShowErrors Sample.

See Also

Reference

CWrappedServerContext Class

Other Resources

CWrappedServerContext Members

CWrappedServerContext Data Members

For information about the data members in **CWrappedServerContext**, see CWrappedServerContext Members.

CWrappedServerContext::m_spParent

The parent object providing the useful implementation of the IHttpServerContext methods.

```
CComPtr< IHttpServerContext > m_spParent;
```

Remarks

This data member is initialized in the constructor.

See Also

Reference

CWrappedServerContext Class

CWrappedServerContext::CWrappedServerContext

Other Resources

CWrappedServerContext Members

CWriteStreamHelper Class

This class provides a convenient way of writing to the IWriteStream interface.

```
class CWriteStreamHelper
```

Remarks

This class provides type conversions via the Write method and overloaded left shift operator (operator <<) for writing data to the IWriteStream interface. The **IWriteStream** interface only accepts strings, but this helper class allows you to transparently pass many different types by providing automatic type conversions.

The type conversions performed by this class are described below:

- Numeric types are converted to their decimal representations.
- Floating-point values are output with a user-specified precision that defaults to ATL_DEFAULT_PRECISION decimal places.
- Currency values are converted according to the locale settings of the current thread.
- Wide character strings are converted to the code page specified by CWriteStreamHelper::SetConversionCodePage.

Requirements

Header: atlisapi.h

Example

See the MantaWeb Sample and the Mailer Sample.

See Also

Reference

CWriteStreamHelper::Write
CWriteStreamHelper::operator <<
IWriteStream Interface

Other Resources

CWriteStreamHelper Members
ATL Server Classes

CWriteStreamHelper Members

Methods

Attach	Call this method to attach an IWriteStream interface to the helper object.
CWriteStreamHelper	The constructor.
SetConversionCodePage	Call this method to set the code page to which wide character strings should be converted.
Write	Call this method to write data to the IWriteStream interface managed by this object.

Operators

operator <<	This operator allows you to write to the attached IWriteStream interface using familiar stream syntax.

See Also

Reference

CWriteStreamHelper Class

CWriteStreamHelper Methods

For information about the methods in **CWriteStreamHelper**, see CWriteStreamHelper Members.

CWriteStreamHelper::Attach

Call this method to attach an IWriteStream interface to the helper object.

```
IWriteStream* Attach(
   IWriteStream* pStream
) throw( );
```

Parameters

pStream

The pointer to an object's **IWriteStream** interface to be attached to the helper object.

Return Value

Returns a pointer to the **IWriteStream** interface of the object previously attached to the helper.

Remarks

Call this method (or use the constructor) to attach an **IWriteStream** interface to the helper so that you can conveniently write to the object supporting that interface.

See Also

Reference

CWriteStreamHelper Class

CWriteStreamHelper::CWriteStreamHelper

Other Resources

CWriteStreamHelper Members

CWriteStreamHelper::CWriteStreamHelper

The constructor.

```
CWriteStreamHelper( ) throw( );
CWriteStreamHelper(
   IWriteStream* pStream
) throw( );
```

Parameters

pStream

The pointer to an object's **IWriteStream** interface to be attached to the helper object.

Remarks

Use the **IWriteStream** constructor (or the default constructor followed by a call to CWriteStreamHelper::Attach) to attach an **IWriteStream** interface to the helper so that you can conveniently write to the object supporting that interface.

See Also

Reference

CWriteStreamHelper Class

CWriteStreamHelper::Attach

Other Resources

CWriteStreamHelper Members

CWriteStreamHelper::SetConversionCodePage

Call this method to set the code page to which wide character strings should be converted.

```
UINT SetConversionCodePage(
   UINT nNewCP
);
```

Parameters

nNewCP

The new code page to which wide character strings should be converted.

Return Value

Returns the previous code page to which wide character strings should be converted.

Remarks

The code page for converting wide character strings defaults to the code page of the current thread.

See Also

Reference

CWriteStreamHelper Class

CWriteStreamHelper::Attach

Other Resources

CWriteStreamHelper Members

CWriteStreamHelper::Write

Call this method to write data to the IWriteStream interface managed by this object.

```
BOOL Write(
   LPCWSTR wsz
) throw();
BOOL Write(
   __int64 i
) throw();
BOOL Write(
   unsigned long int dw
) throw();
BOOL Write(
   unsigned short int w
) throw( );
BOOL Write(
   unsigned int u
) throw();
BOOL Write(
   long int dw
) throw();
BOOL Write(
   short int w
) throw();
BOOL Write(
   int n
) throw();
BOOL Write(
   LPCSTR szOut
) throw();
BOOL Write(
   CURRENCY c
) throw();
BOOL Write(
   unsigned __int64 i
) throw();
BOOL Write(
   double d,
   int nDigitCount = ATL_DEFAULT_PRECISION
) throw();
```

Parameters

wsz, i, dw, u, w, n, szOut, c, i, d

The data to be written to the **IWriteStream** interface managed by the helper object.

nDigitCount

The number of digits to use when displaying the value.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The overloads of this method are equivalent to the corresponding overload of CWriteStreamHelper::operator <<.

See CWriteStreamHelper Overview for important information on the type conversions performed by this class.

Example

See the Mailer Sample.

See Also

Reference

CWriteStreamHelper Class
CWriteStreamHelper::operator <<
ATL_DEFAULT_PRECISION

Other Resources

CWriteStreamHelper Members

CWriteStreamHelper Operators

For information about the operators in **CWriteStreamHelper**, see CWriteStreamHelper Members.

CWriteStreamHelper::operator <<

This operator allows you to write to the attached IWriteStream interface using familiar stream syntax.

```
CWriteStreamHelper& operator<<(
   LPCWSTR wszStr
) throw(...);
CWriteStreamHelper& operator<<(
   unsigned __int64 i
) throw(...);
CWriteStreamHelper& operator<<(
   unsigned int u
) throw(...);
CWriteStreamHelper& operator<<(
   LPCSTR szStr
) throw(...);
CWriteStreamHelper& operator<<(
   __int64 i
) throw(...);
CWriteStreamHelper& operator<<(
   CURRENCY c
) throw(...);
CWriteStreamHelper& operator<<(
   double d
) throw(...);
CWriteStreamHelper& operator<<(
   int n
) throw(...);
CWriteStreamHelper& operator<<(
   long int dw
) throw(...);
CWriteStreamHelper& operator<<(
   short int w
) throw(...);
CWriteStreamHelper& operator<<(
   unsigned long int dw
) throw(...);
```

Parameters

i, u, c, d, n, w, dw, szStr, wszStr

The data to be written to the **IWriteStream** interface managed by the helper object.

Return Value

Returns a reference to the helper object.

Remarks

The overloads of this operator are equivalent to the corresponding overload of CWriteStreamHelper::Write.

See CWriteStreamHelper Overview for important information on the type conversions performed by this class.

Example

See the MantaWeb Sample.

See Also

Reference

CWriteStreamHelper Class

CWriteStreamHelper::Write

Other Resources

CWriteStreamHelper Members

CWriteStreamOnCString Class

This class provides an implementation of IWriteStream that writes the data to a CStringA data member.

```
class CWriteStreamOnCString :
   public IWriteStream
```

Requirements

Header: atlsoap.h

See Also

Reference

IWriteStream Interface

Other Resources

CWriteStreamOnCString Members

ATL Server Classes

CWriteStreamOnCString Members

Methods

Cleanup	Empties m_str of data.
FlushStream	Implementation of IWriteStream::FlushStream.
WriteStream	Implementation of IWriteStream::WriteStream.

Data Members

m_str	The member variable to which data is written when WriteStream is called.

See Also

Reference

CWriteStreamOnCString Class

CWriteStreamOnCString Methods

For information about the methods in **CWriteStreamOnCString**, see CWriteStreamOnCString Members.

CWriteStreamOnCString::Cleanup

Empties m_str of data.

```
void Cleanup();
```

See Also

Reference

CWriteStreamOnCString Class

Other Resources

CWriteStreamOnCString Members

CWriteStreamOnCString::FlushStream

Implementation of IWriteStream::FlushStream.

```
HRESULT FlushStream();
```

Return Value

Always returns S_OK.

Remarks

Doesn't change the state of m_str.

See Also

Reference

CWriteStreamOnCString Class

Other Resources

CWriteStreamOnCString Members

CWriteStreamOnCString::WriteStream

Implementation of IWriteStream::WriteStream.

```
HRESULT WriteStream(
   LPCSTR szOut,
   int nLen,
   LPDWORD pdwWritten
);
```

Remarks

Writes the data to m_str.

See Also

Reference

CWriteStreamOnCString Class

Other Resources

CWriteStreamOnCString Members

CWriteStreamOnCString Data Members

For information about the data members in **CWriteStreamOnCString**, see CWriteStreamOnCString Members.

CWriteStreamOnCString::m_str

The member variable to which data is written when WriteStream is called.

```
CStringA m_str;
```

See Also

Reference

CWriteStreamOnCString Class

Other Resources

CWriteStreamOnCString Members

_DLL_CACHE_ENTRY Structure

This structure represents an entry in a DLL cache.

```
struct _DLL_CACHE_ENTRY
```

Remarks

This structure is equivalent to the DLL_CACHE_ENTRY structure, but is compatible with ATL Server's implementation of SOAP, so it can be used in the definition of IDllCacheMgr implemented by the CDllCacheManager web service.

Requirements

Header: atlextmgmt.h

See Also

Other Resources

_DLL_CACHE_ENTRY Members

_DLL_CACHE_ENTRY Members

Data Members

dwRefs	The reference count on the cached DLL.
hInstDll	The instance handle of the cached DLL.
szDllName	The name of the cached DLL.

See Also

Reference

_DLL_CACHE_ENTRY Structure

_DLL_CACHE_ENTRY Data Members

For information about the data members in **_DLL_CACHE_ENTRY**, see _DLL_CACHE_ENTRY Members

_DLL_CACHE_ENTRY::dwRefs

The reference count on the cached DLL.

```
DWORD dwRefs;
```

See Also

Reference

_DLL_CACHE_ENTRY Structure

DLL_CACHE_ENTRY::dwRefs

Other Resources

_DLL_CACHE_ENTRY Members

_DLL_CACHE_ENTRY::hInstDll

The instance handle of the cached DLL.

```
DWORD hInstDll;
```

See Also

Reference

_DLL_CACHE_ENTRY Structure

DLL_CACHE_ENTRY::hInstDll

Other Resources

_DLL_CACHE_ENTRY Members

_DLL_CACHE_ENTRY::szDllName

The name of the cached DLL.

```
BSTR szDllName;
```

See Also

Reference

_DLL_CACHE_ENTRY Structure

DLL_CACHE_ENTRY::szDllName

Other Resources

_DLL_CACHE_ENTRY Members

DLL_CACHE_ENTRY Structure

This structure describes a cached DLL.

```
struct DLL_CACHE_ENTRY
```

Remarks

This structure contains fields that relate the name, instance handle, and reference count of a DLL currently stored in a cache. This structure is used by the CDllCache cache implementation and the IDllCache interface.

Requirements

Header: atlcache.h

See Also

Reference

CDllCache Class

IDllCache::GetEntries

Other Resources

Caching

DLL_CACHE_ENTRY Members

Caching Reference

DLL_CACHE_ENTRY Members

Data Members

bAlive	This data member indicates whether or not the cached DLL has been used since the cache was last flushed.
dwRefs	This data member contains the reference count of the cached DLL.
hInstDll	This data member contains the instance handle of the cached DLL.
szDllName	This data member contains the name of the cached DLL.

See Also

Reference

DLL_CACHE_ENTRY Structure

Other Resources

Caching Reference

DLL_CACHE_ENTRY Data Members

For information about the data members in **DLL_CACHE_ENTRY**, see DLL_CACHE_ENTRY Members.

DLL_CACHE_ENTRY::bAlive

This data member indicates whether or not the cached DLL has been used since the cache was last flushed.

```
BOOL bAlive;
```

Remarks

This member is used along with the reference count in dwRefs to determine whether the DLL can be removed from the cache. If dwRefs is zero and **bAlive** is **FALSE**, the DLL may be removed from the cache when IDllCache::Flush is called. The presence of this member means that a DLL will need to have a reference count of zero on two consecutive calls to Flush for it to be removed from the cache.

See Also

Reference

DLL_CACHE_ENTRY Structure

Other Resources

DLL_CACHE_ENTRY Members

Caching Reference

DLL_CACHE_ENTRY::dwRefs

This data member contains the reference count of the cached DLL.

```
DWORD dwRefs;
```

Remarks

See bAlive for details.

See Also

Reference

DLL_CACHE_ENTRY Structure

Other Resources

DLL_CACHE_ENTRY Members

Caching Reference

DLL_CACHE_ENTRY::hInstDll

This data member contains the instance handle of the cached DLL.

```
HINSTANCE hInstDll;
```

Remarks

Using the instance handle for purposes other than comparison is not recommended, as this will corrupt the state of the DLL cache. The cache implementation is responsible for loading and unloading the DLL.

See Also

Reference

DLL_CACHE_ENTRY Structure

IDllCache Interface

Other Resources

DLL_CACHE_ENTRY Members

Caching Reference

DLL_CACHE_ENTRY::szDllName

This data member contains the name of the cached DLL.

```
CHAR szDllName[MAX_PATH];
```

Remarks

The **szDllName** member contains the full path and name of the cached DLL.

See Also

Reference

DLL_CACHE_ENTRY Structure

Other Resources

DLL_CACHE_ENTRY Members

Caching Reference

HTML_SCHEME Structure

This class provides HTML formatting information for use with CHtmlGenBase.

```
struct HTML_SCHEME
```

Remarks

You can use this class and CHtmlGenBase to define certain formatting attributes (such as background color) that are applied to HTML elements when appropriate. For example, if CHtmlGenBase::body is called without specifying a background color, a BGCOLOR attribute will be added if a background color has been set in the **HTML_SCHEME** passed to the CHtmlGenBase constructor or CHtmlGenBase::SetScheme.

This class provides a convenient way of providing limited style information for situations in which the HTML viewer does not support style sheets.

Requirements

Header: atlhtml.h

See Also

Other Resources

HTML_SCHEME Members

ATL Server Classes

HTML_SCHEME Members

Methods

HTML_SCHEME	The constructor.

Data Members

nLeftMargin	The LEFTMARGIN attribute of the BODY element.
nTopMargin	The TOPMARGIN attribute of the BODY element.
strALinkColor	The ALINK attribute of the BODY element.
strBackground	The BACKGROUND attribute of the BODY element.
strBgColor	The BGCOLOR attribute of the BODY element.
strLinkColor	The LINK attribute of the BODY element.
strTableBgColor	The BGCOLOR attribute of a TABLE element.
strTdBgColor	The BGCOLOR attribute of a TD element.
strTrBgColor	The BGCOLOR attribute of a TR element.
strVLinkColor	The VLINK attribute of the BODY element.

See Also

Reference

HTML_SCHEME Structure

HTML_SCHEME Methods

For information about the methods in **HTML_SCHEME**, see HTML_SCHEME Members.

HTML_SCHEME::HTML_SCHEME

The constructor.

```
HTML_SCHEME( );
```

Remarks

Initializes nTopMargin and nLeftMargin to -1.

See Also

Reference

HTML_SCHEME Structure

Other Resources

HTML_SCHEME Members

HTML_SCHEME Data Members

For information about the data members in **HTML_SCHEME**, see HTML_SCHEME Members.

HTML_SCHEME::nLeftMargin

The LEFTMARGIN attribute of the BODY element.

```
int nLeftMargin;
```

Remarks

The value -1 is used to indicate that the left margin has not been set.

See Also

Reference

HTML_SCHEME Structure

Other Resources

HTML_SCHEME Members

HTML_SCHEME::nTopMargin

The TOPMARGIN attribute of the BODY element.

```
int nTopMargin;
```

Remarks

The value -1 is used to indicate that the top margin has not been set.

See Also

Reference

HTML_SCHEME Structure

Other Resources

HTML_SCHEME Members

HTML_SCHEME::strALinkColor

The ALINK attribute of the BODY element.

```
CString strALinkColor;
```

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

HTML_SCHEME Structure

Other Resources

HTML_SCHEME Members

HTML_SCHEME::strBackground

The BACKGROUND attribute of the BODY element.

```
CString strBackground;
```

Remarks

The URL of the background image.

See Also

Reference

HTML_SCHEME Structure

Other Resources

HTML_SCHEME Members

HTML_SCHEME::strBgColor

The BGCOLOR attribute of the BODY element.

```
CString strBgColor;
```

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

HTML_SCHEME Structure

Other Resources

HTML_SCHEME Members

HTML_SCHEME::strLinkColor

The LINK attribute of the BODY element.

```
CString strLinkColor;
```

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

HTML_SCHEME Structure

Other Resources

HTML_SCHEME Members

HTML_SCHEME::strTableBgColor

The BGCOLOR attribute of a TABLE element.

```
CString strTableBgColor;
```

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

HTML_SCHEME Structure

Other Resources

HTML_SCHEME Members

HTML_SCHEME::strTdBgColor

The BGCOLOR attribute of a TD element.

```
CString strTdBgColor;
```

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

HTML_SCHEME Structure

Other Resources

HTML_SCHEME Members

HTML_SCHEME::strTrBgColor

The BGCOLOR attribute of a TR element.

```
CString strTrBgColor;
```

See Also

Reference

HTML_SCHEME Structure

Other Resources

HTML_SCHEME Members

HTML_SCHEME::strVLinkColor

The VLINK attribute of the BODY element.

```
CString strVLinkColor;
```

Example

For an example, see the CHtmlGenBase Overview.

See Also

Reference

HTML_SCHEME Structure

Other Resources

HTML_SCHEME Members

IAuthInfo Class

This interface defines methods used to retrieve authentication credentials required for accessing protected URLs.

```
__interface IAuthInfo
```

Remarks

The **IAuthInfo** interface is used to provide authentication information when performing HTTP transactions. Typically, an implementation class, derived from **IAuthInfo** interface, is provided using the CAtlHttpClientT::AddAuthObj method along with a security scheme-specific implementation object. If the given authentication scheme is encountered, the CAtlHttpClientT class attempts to satisfy the security requirement using the authorization object to provide the implementation and the **IAuthInfo** implementation object to provide the user name, password, and domain name of the authorized party. The authorization credentials that the **IAuthInfo** interface provides are requested only if needed and are not stored by the CAtlHttpClientT class.

ATL Server provides HTTP authorization scheme implementations for two protocols: BASIC and NTLM. The class that provides that BASIC implementation, CBasicAuthObject, requires that an **IAuthInfo** implementation class be used so that the CBasicAuthObject can retrieve authorization information if needed. The NTLM class, CNTLMAuthObject, does not require an **IAuthInfo** implementation because it uses the current user's credentials to navigate NTLM protected URLs. Additional HTTP authorization implementations can be provided by deriving from the CAtlBaseAuthObject class.

Requirements

Header: atlhttp.h

Example

See the HttpClient Sample.

See Also

Concepts

HTTP Client Services

Other Resources

IAuthInfo Members

HTTP Client Reference

IAuthInfo Members

Methods

GetDomain	This method is called when an HTTP authorization implementation requires a domain to access a protected URL.
GetPassword	This method is called when an HTTP authorization implementation requires a password to access a protected URL.
GetUsername	This method is called when an HTTP authorization implementation requires a username to access a protected URL.

See Also

Reference

IAuthInfo Class

Concepts

HTTP Client Services

Other Resources

HTTP Client Reference

IAuthInfo Methods

For information about the methods in **IAuthInfo**, see IAuthInfo Members

IAuthInfo::GetDomain

This method is called when an HTTP authorization implementation requires a domain to access a protected URL.

```
HRESULT GetDomain(
   LPTSTR szDomain,
   DWORD *pdwBuffSize
);
```

Parameters

szDomain

A pointer to the buffer where the **IAuthInfo** implementation class is to store the domain name.

pdwBuffSize

The address of a **DWORD** that, on entry, contains the size of the buffer provided as *szDomain*. Implementations of **IAuthInfo** should not exceed the buffer size when providing domain information. On exit, *pdwBuffSize* should contain the size of the domain name, including the **NULL** terminator.

Return Value

An HRESULT indicating the success or failure of the call. Returns 0 if the domain name is successfully retrieved, otherwise nonzero.

Remarks

The **GetDomain** method is typically called by a CAtlBaseAuthObject derived class that is attempting to satisfy the security requirements of a requested URL.

A domain name is not required for all authentication schemes. The BASIC scheme, for example, requires a user name and password, but not a domain name. As a result, the CBasicAuthObject class, ATL Server's implementation of the BASIC scheme, does not call the **GetDomain** method.

Example

See the HttpClient Sample.

See Also

Reference

IAuthInfo Class

Concepts

HTTP Client Services

Other Resources

IAuthInfo Members

HTTP Client Reference

IAuthInfo::GetPassword

This method is called when an HTTP authorization implementation requires a password to access a protected URL.

```
HRESULT GetPassword(
   LPTSTR szPwd,
   DWORD *pdwBuffSize
);
```

Parameters

szPwd

A pointer to a buffer where the **IAuthInfo** implementation is to store the required password.

pdwBuffSize

The address of a **DWORD** that, on entry, contains the size of the buffer provided as *szPwd*. Implementations of **IAuthInfo** should not exceed the buffer size when providing domain information. On exit, *pdwBuffSize* should contain the size of the password including the **NULL** terminator.

Return Value

An HRESULT indicating the success or failure of the call. Returns 0 if the password is successfully retrieved, otherwise nonzero.

Remarks

The **GetPassword** method is typically called by a CAtlBaseAuthObject derived class that is attempting to satisfy the security requirements of a requested URL.

Example

See the HttpClient Sample.

See Also

Reference

IAuthInfo Class

Concepts

HTTP Client Services

Other Resources

IAuthInfo Members

HTTP Client Reference

IAuthInfo::GetUsername

This method is called when an HTTP authorization implementation requires a user name to access a protected URL.

```
HRESULT GetUsername(
   LPTSTR szUid,
   DWORD *pdwBuffSize
);
```

Parameters

szUid

A pointer to a buffer where the **IAuthInfo** implementation is to store the required username.

pdwBuffSize

The address of a **DWORD** that, on entry, contains the size of the buffer provided as *szUid*. Implementations of **IAuthInfo** should not exceed the buffer size when providing domain information. On exit, *pdwBuffSize* should contain the size of the username including the **NULL** terminator.

Return Value

An HRESULT indicating the success or failure of the call. Returns 0 if the user name is successfully retrieved, otherwise nonzero.

Remarks

The **GetUsername** method is typically called by a CAtlBaseAuthObject derived class that is attempting to satisfy the security requirements of a requested URL.

Example

See the HttpClient Sample.

See Also

Reference

IAuthInfo Class

Concepts

HTTP Client Services

Other Resources

IAuthInfo Members

HTTP Client Reference

IBrowserCaps Interface

This interface provides methods for determining the capabilities of a browser.

```
__interface __declspec( uuid( "3339FCE2-99BC-4985-A702-4ABC8304A995" ))
IBrowserCaps :
   public IUnknown
```

Remarks

An object implementing this interface can be obtained by calling one of the IBrowserCapsSvc methods.

Requirements

Header: atlsiface.h

Example

```
CComPtr<IBrowserCaps> m_spBrowserCaps;

[ tag_name(name = "LoadBrowserCaps") ]
HTTP_CODE OnLoadBrowserCaps()
{
    // Get the IBrowserCapsSvc service from the ISAPI extension.
    CComPtr<IBrowserCapsSvc> spBrowserCapsSvc;
    if (FAILED(m_spServiceProvider->QueryService(
                    __uuidof(IBrowserCapsSvc), &spBrowserCapsSvc)))
    {
        return HTTP_FAIL;
    }

    // Get user agent string from the query parameters.
    LPCSTR szUserAgent = m_HttpRequest.QueryParams.Lookup("UserAgent");
    if (szUserAgent != NULL)
    {
        // Get the browser capabilities for the specified user agent.
        if (FAILED(spBrowserCapsSvc->GetCapsUserAgent(
                            CComBSTR(szUserAgent), &m_spBrowserCaps)))
        {
            return HTTP_FAIL;
        }
    }
    else
    {
        // Get the browser capabilities for the user's current user agent.
        if (FAILED(spBrowserCapsSvc->GetCaps(m_spServerContext,
                                                &m_spBrowserCaps)))
        {
            return HTTP_FAIL;
        }
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCapsSvc Interface

Other Resources

IBrowserCaps Members

ATL Server Classes

IBrowserCaps Members

Methods

GetBooleanPropertyValue	Call this method to get the value of the specified Boolean property of the browser.
GetBrowserName	Call this method to get the name of the browser.
GetMajorVer	Call this method to get the major version of the browser.
GetMinorVer	Call this method to get the minor version of the browser.
GetPlatform	Call this method to get the platform that the browser runs on.
GetPropertyString	Call this method to get the value of the specified property as a string.
GetVersion	Call this method to get the browser version.
IsAK	Call this method to determine whether the browser is AK-compatible.
IsAOL	Call this method to determine whether the browser is an America Online-branded browser.
IsBeta	Call this method to determine whether the browser is beta software.
IsCrawler	Call this method to determine whether the browser is a crawler.
IsSK	Call this method to determine whether the browser is SK-compatible.
IsUpdate	Call this method to determine whether the browser is an update.
IsWin16	Call this method to determine whether the browser is running on a 16-bit Windows operating s ystem.
SupportsActiveXControls	Call this method to determine whether the browser supports ActiveX controls.
SupportsAuthenticodeUpdate	Call this method to determine whether the browser supports Authenticode.
SupportsBackgroundSounds	Call this method to determine whether the browser supports playing background sounds.
SupportsCDF	Call this method to determine whether the browser supports Channel Definition Format (CDF).
SupportsCookies	Call this method to determine whether the browser supports cookies.
SupportsFrames	Call this method to determine whether the browser supports frames.
SupportsJavaApplets	Call this method to determine whether the browser supports Java applets.
SupportsJavaScript	Call this method to determine whether the browser supports JavaScript.
SupportsTables	Call this method to determine whether the browser supports HTML tables.
SupportsVBScript	Call this method to determine whether the browser supports VBScript.

See Also

Reference

IBrowserCaps Interface

IBrowserCaps Methods

For information about the methods in **IBrowserCaps**, see IBrowserCaps Members

IBrowserCaps::GetBooleanPropertyValue

Call this method to get the value of the specified Boolean property of the browser.

```
HRESULT GetBooleanPropertyValue(
   BSTR bstrProperty,
   BOOL* pbOut
);
```

Parameters

bstrProperty

The name of the property whose value you want to retrieve.

pbOut

[out] Address of the variable that, on success, receives the value of the property.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

See Also

Reference

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::GetBrowserName

Call this method to get the name of the browser.

```
HRESULT GetBrowserName(
   BSTR * pbstrName
);
```

Parameters

pbstrName

[out] Address of the BSTR variable that, on success, receives the browser name.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "GetBrowserName") ]
HTTP_CODE OnGetBrowserName()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    CComBSTR bstr;
    if (FAILED(m_spBrowserCaps->GetBrowserName(&bstr)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(bstr))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::GetMajorVer

Call this method to get the major version of the browser.

```
HRESULT GetMajorVer(
   BSTR * pbstrMajorVer
);
```

Parameters

pbstrMajorVer

[out] Address of the BSTR variable that, on success, receives the major version.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "GetMajorVer") ]
HTTP_CODE OnGetMajorVer()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    CComBSTR bstr;
    if (FAILED(m_spBrowserCaps->GetMajorVer(&bstr)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(bstr))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::GetMinorVer

Call this method to get the minor version of the browser.

```
HRESULT GetMinorVer(
   BSTR * pbstrMinorVer
);
```

Parameters

pbstrMinorVer

[out] Address of the BSTR variable that, on success, receives the minor version.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "GetMinorVer") ]
HTTP_CODE OnGetMinorVer()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    CComBSTR bstr;
    if (FAILED(m_spBrowserCaps->GetMinorVer(&bstr)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(bstr))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::GetPlatform

Call this method to get the platform that the browser runs on.

```
HRESULT GetPlatform(
   BSTR * pbstrPlatform
);
```

Parameters

pbstrPlatform

[out] Address of the BSTR variable that, on success, receives the platform.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "GetPlatform") ]
HTTP_CODE OnGetPlatform()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    CComBSTR bstr;
    if (FAILED(m_spBrowserCaps->GetPlatform(&bstr)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(bstr))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::GetPropertyString

Call this method to get the value of the specified property as a string.

```
HRESULT GetPropertyString(
   BSTR bstrProperty,
   BSTR * pbstrOut
);
```

Parameters

bstrProperty

The string that identifies the property whose value you want to retrieve.

pbstrOut

[out] Address of the BSTR variable that, on success, receives the property value.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

See Also

Reference

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::GetVersion

Call this method to get the browser version.

```
HRESULT GetVersion(
   BSTR * pbstrVersion
);
```

Parameters

pbstrVersion

[out] Address of the BSTR variable that, on success, receives the version number.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "GetVersion") ]
HTTP_CODE OnGetVersion()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    CComBSTR bstr;
    if (FAILED(m_spBrowserCaps->GetVersion(&bstr)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(bstr))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::IsAK

Call this method to determine whether the browser is AK-compatible.

```
HRESULT IsAK(
   BOOL* pbIsAK
);
```

Parameters

pbIsAK

[out] Address of the variable that, on success, receives the value of the property.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "IsAK") ]
HTTP_CODE OnIsAK()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    BOOL b;
    if (FAILED(m_spBrowserCaps->IsAK(&b)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(b ? "Yes" : "No"))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::IsAOL

Call this method to determine whether the browser is an America Online-branded browser.

```
HRESULT IsAOL(
   BOOL* pbIsAOL
);
```

Parameters

pbIsAOL

[out] Address of the variable that, on success, receives the value of the property.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "IsAOL") ]
HTTP_CODE OnIsAOL()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    BOOL b;
    if (FAILED(m_spBrowserCaps->IsAOL(&b)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(b ? "Yes" : "No"))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::IsBeta

Call this method to determine whether the browser is beta software.

```
HRESULT IsBeta(
   BOOL* pbIsBeta
);
```

Parameters

pbIsBeta

[out] Address of the variable that, on success, receives the value of the property.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "IsBeta") ]
HTTP_CODE OnIsBeta()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    BOOL b;
    if (FAILED(m_spBrowserCaps->IsBeta(&b)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(b ? "Yes" : "No"))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::IsCrawler

Call this method to determine whether the browser is a crawler.

```
HRESULT IsCrawler(
   BOOL* pbIsCrawler
);
```

Parameters

pbIsCrawler

[out] Address of the variable that, on success, receives the value of the property.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "IsCrawler") ]
HTTP_CODE OnIsCrawler()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    BOOL b;
    if (FAILED(m_spBrowserCaps->IsCrawler(&b)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(b ? "Yes" : "No"))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::IsSK

Call this method to determine whether the browser is SK-compatible.

```
HRESULT IsSK(
   BOOL* pbIsSK
);
```

Parameters

pbIsSK

[out] Address of the variable that, on success, receives the value of the property.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "IsSK") ]
HTTP_CODE OnIsSK()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    BOOL b;
    if (FAILED(m_spBrowserCaps->IsSK(&b)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(b ? "Yes" : "No"))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::IsUpdate

Call this method to determine whether the browser is an update.

```
HRESULT IsUpdate(
   BOOL* pbIsUpdate
);
```

Parameters

pbIsUpdate

[out] Address of the variable that, on success, receives the value of the property.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "IsUpdate") ]
HTTP_CODE OnIsUpdate()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    BOOL b;
    if (FAILED(m_spBrowserCaps->IsUpdate(&b)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(b ? "Yes" : "No"))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::IsWin16

Call this method to determine whether the browser is running on a 16-bit Windows operating system.

```
HRESULT IsWin16(
   BOOL* pbIsWin16
);
```

Parameters

pbIsWin16

[out] Address of the variable that, on success, receives the value of the property.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "IsWin16") ]
HTTP_CODE OnIsWin16()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    BOOL b;
    if (FAILED(m_spBrowserCaps->IsWin16(&b)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(b ? "Yes" : "No"))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::SupportsActiveXControls

Call this method to determine whether the browser supports ActiveX controls.

```
HRESULT SupportsActiveXControls(
   BOOL* pbActiveXControls
);
```

Parameters

pbActiveXControls

[out] Address of the variable that, on success, receives the value of the property.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "SupportsActiveXControls") ]
HTTP_CODE OnSupportsActiveXControls()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    BOOL b;
    if (FAILED(m_spBrowserCaps->SupportsActiveXControls(&b)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(b ? "Yes" : "No"))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::SupportsAuthenticodeUpdate

Call this method to determine whether the browser supports Authenticode.

```
HRESULT SupportsAuthenticodeUpdate(
   BOOL* pbAuthenticodeUpdate
);
```

Parameters

pbAuthenticodeUpdate

[out] Address of the variable that, on success, receives the value of the property.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "SupportsAuthenticodeUpdate") ]
HTTP_CODE OnSupportsAuthenticodeUpdate()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    BOOL b;
    if (FAILED(m_spBrowserCaps->SupportsAuthenticodeUpdate(&b)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(b ? "Yes" : "No"))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::SupportsBackgroundSounds

Call this method to determine whether the browser supports playing background sounds.

```
HRESULT SupportsBackgroundSounds(
   BOOL* pbBackgroundSounds
);
```

Parameters

pbBackgroundSounds

[out] Address of the variable that, on success, receives the value of the property.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "SupportsBackgroundSounds") ]
HTTP_CODE OnSupportsBackgroundSounds()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    BOOL b;
    if (FAILED(m_spBrowserCaps->SupportsBackgroundSounds(&b)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(b ? "Yes" : "No"))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::SupportsCDF

Call this method to determine whether the browser supports Channel Definition Format (CDF).

```
HRESULT SupportsCDF(
   BOOL* pbCDF
);
```

Parameters

pbCDF

[out] Address of the variable that, on success, receives the value of the property.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "SupportsCDF") ]
HTTP_CODE OnSupportsCDF()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    BOOL b;
    if (FAILED(m_spBrowserCaps->SupportsCDF(&b)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(b ? "Yes" : "No"))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::SupportsCookies

Call this method to determine whether the browser supports cookies.

```
HRESULT SupportsCookies(
   BOOL* pbCookies
);
```

Parameters

pbCookies

[out] Address of the variable that, on success, receives the value of the property.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "SupportsCookies") ]
HTTP_CODE OnSupportsCookies()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    BOOL b;
    if (FAILED(m_spBrowserCaps->SupportsCookies(&b)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(b ? "Yes" : "No"))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::SupportsFrames

Call this method to determine whether the browser supports frames.

```
HRESULT SupportsFrames(
   BOOL* pbFrames
);
```

Parameters

pbFrames

[out] Address of the variable that, on success, receives the value of the property.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "SupportsFrames") ]
HTTP_CODE OnSupportsFrames()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    BOOL b;
    if (FAILED(m_spBrowserCaps->SupportsFrames(&b)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(b ? "Yes" : "No"))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::SupportsJavaApplets

Call this method to determine whether the browser supports Java applets.

```
HRESULT SupportsJavaApplets(
   BOOL* pbJavaApplets
);
```

Parameters

pbJavaApplets

[out] Address of the variable that, on success, receives the value of the property.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "SupportsJavaApplets") ]
HTTP_CODE OnSupportsJavaApplets()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    BOOL b;
    if (FAILED(m_spBrowserCaps->SupportsJavaApplets(&b)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(b ? "Yes" : "No"))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::SupportsJavaScript

Call this method to determine whether the browser supports JavaScript.

```
HRESULT SupportsJavaScript(
   BOOL* pbJavaScript
);
```

Parameters

pbJavaScript

[out] Address of the variable that, on success, receives the value of the property.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "SupportsJavaScript") ]
HTTP_CODE OnSupportsJavaScript()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    BOOL b;
    if (FAILED(m_spBrowserCaps->SupportsJavaScript(&b)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(b ? "Yes" : "No"))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::SupportsTables

Call this method to determine whether the browser supports HTML tables.

```
HRESULT SupportsTables(
   BOOL* pbTables
);
```

Parameters

pbTables

[out] Address of the variable that, on success, receives the value of the property.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "SupportsTables") ]
HTTP_CODE OnSupportsTables()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    BOOL b;
    if (FAILED(m_spBrowserCaps->SupportsTables(&b)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(b ? "Yes" : "No"))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCaps::SupportsVBScript

Call this method to determine whether the browser supports VBScript.

```
HRESULT SupportsVBScript(
   BOOL* pbVBScript
);
```

Parameters

pbVBScript

[out] Address of the variable that, on success, receives the value of the property.

Return Value

Returns S_OK on success, S_FALSE if the property value is not specified, or an error HRESULT on failure.

Example

```
[ tag_name(name = "SupportsVBScript") ]
HTTP_CODE OnSupportsVBScript()
{
    ATLASSERT(m_spBrowserCaps != NULL);

    BOOL b;
    if (FAILED(m_spBrowserCaps->SupportsVBScript(&b)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(b ? "Yes" : "No"))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

IBrowserCaps Interface

Other Resources

IBrowserCaps Members

IBrowserCapsSvc Interface

This interface allows you to get access to objects that provide information about the capabilities of a particular user agent (web browser).

```
__interface
__declspec( uuid( "391E7418-863B-430e-81BB-1312ED2FF3E9" ))
IBrowserCapsSvc :
   public IUnknown
```

Remarks

The information about a particular user agent is exposed by the IBrowserCaps interface. Pointers to objects implementing this interface are returned by the IBrowserCapsSvc::GetCaps and IBrowserCapsSvc::GetCapsUserAgent methods.

This interface is implemented by CBrowserCapsSvc.

Requirements

Header: atlsiface.h

Example

```
CComPtr<IBrowserCaps> m_spBrowserCaps;

[ tag_name(name = "LoadBrowserCaps") ]
HTTP_CODE OnLoadBrowserCaps()
{
    // Get the IBrowserCapsSvc service from the ISAPI extension.
    CComPtr<IBrowserCapsSvc> spBrowserCapsSvc;
    if (FAILED(m_spServiceProvider->QueryService(
                    __uuidof(IBrowserCapsSvc), &spBrowserCapsSvc)))
    {
        return HTTP_FAIL;
    }

    // Get user agent string from the query parameters.
    LPCSTR szUserAgent = m_HttpRequest.QueryParams.Lookup("UserAgent");
    if (szUserAgent != NULL)
    {
        // Get the browser capabilities for the specified user agent.
        if (FAILED(spBrowserCapsSvc->GetCapsUserAgent(
                            CComBSTR(szUserAgent), &m_spBrowserCaps)))
        {
            return HTTP_FAIL;
        }
    }
    else
    {
        // Get the browser capabilities for the user's current user agent.
        if (FAILED(spBrowserCapsSvc->GetCaps(m_spServerContext,
                                                &m_spBrowserCaps)))
        {
            return HTTP_FAIL;
        }
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowBrowser

CBrowserCapsSvc Class

IBrowserCaps Interface

Other Resources

IBrowserCapsSvc Members

ATL Server Classes

IBrowserCapsSvc Members

Methods

GetCaps	Call this method to get the browser capabilities for the browser identified by the HTTP_USER_AGENT server variable in the supplied server context.
GetCapsUserAgent	Call this method to get the browser capabilities for the browser identified by a string.

See Also

Reference

IBrowserCapsSvc Interface

IBrowserCapsSvc Methods

For information about the methods in **IBrowserCapsSvc**, see IBrowserCapsSvc Members

IBrowserCapsSvc::GetCaps

Call this method to get the browser capabilities for the browser identified by the HTTP_USER_AGENT server variable in the supplied server context.

```
HRESULT GetCaps(
   IHttpServerContext * pContext,
   IBrowserCaps ** ppOut
);
```

Parameters

pContext

The server context identifying the browser via the HTTP_USER_AGENT server variable.

ppOut

[out] Address of the pointer variable that, on success, receives the **IBrowserCaps** interface pointer of the object describing the capabilities of the browser.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method retrieves the string identifying the browser from the server context's HTTP_USER_AGENT server variable, then calls IBrowserCapsSvc::GetCapsUserAgent to get the browser capabilities.

Example

See IBrowserCapsSvc Overview.

See Also

Reference

ShowBrowser
IBrowserCapsSvc Interface
IHttpServerContext Interface
IBrowserCaps Interface

Other Resources

IBrowserCapsSvc Members

IBrowserCapsSvc::GetCapsUserAgent

Call this method to get the browser capabilities for the browser identified by a string.

```
HRESULT GetCapsUserAgent(
   BSTR bstrAgent,
   IBrowserCaps ** ppOut
);
```

Parameters

bstrAgent

The string identifying the user agent (web browser).

ppOut

[out] Address of the pointer variable that, on success, receives the **IBrowserCaps** interface pointer of the object describing the capabilities of the browser.

Return Value

Returns S_OK on success, or an error HRESULT on failure or if no information can be found for the specified user agent.

Example

See IBrowserCapsSvc Overview.

See Also

Reference

ShowBrowser
IBrowserCapsSvc Interface
IBrowserCaps Interface

Other Resources

IBrowserCapsSvc Members

IDataSourceCache Interface

This interface provides methods for adding database connections to and retrieving them from a cache of open connections.

```
__interface
__declspec( uuid( "52E7759B-D6CC-4a03-BDF3-80A6BDCA1F94" ))
IDataSourceCache :
   public IUnknown
```

Remarks

The cache holds open CDataConnection objects. This interface is implemented by CDataSourceCache.

Requirements

Header: atlcache.h

Example

See the OnlineAddressBook Sample.

See Also

Reference

CDataConnection Class
CDataSourceCache Class

Other Resources

IDataSourceCache Members
ATL Server Classes
Caching
Caching Reference

IDataSourceCache Members

Methods

Add	Call this method to retrieve a data connection from the cache or add it if not present.
Lookup	Call this method to retrieve an open data connection from the cache.
Remove	Call this method to close the connection and remove it from the data source cache.
Uninitialize	Call this method to close all of the data connections and remove them from the cache.

See Also

Reference

IDataSourceCache Interface

IDataSourceCache Methods

For information about the methods in **IDataSourceCache**, see IDataSourceCache Members.

IDataSourceCache::Add

Call this method to retrieve a data connection from the cache or add it if not present.

```
STDMETHOD(Add)(
   LPCTSTR szID,
   LPCOLESTR szConn,
   CDataConnection* pDS
);
```

Parameters

szID

The ID of the connection in the cache. Can be the same as *szConn*.

szConn

The connection string of the data source to which to connect.

pDS

[out] Address of the CDataConnection variable that, on success, receives a copy of the data connection.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

This method attempts to open a connection to the data source specified by *szConn* using a **CDataConnection** object. Once the connection is open, the **CDataConnection** is cached, and a copy is returned to the caller of the function in *pDS*.

See Also

Reference

IDataSourceCache Interface

IDataSourceCache::Lookup

Other Resources

IDataSourceCache Members

IDataSourceCache::Lookup

Call this method to retrieve an open data connection from the cache.

```
STDMETHOD(Lookup)(
   LPCTSTR szID,
   CDataConnection* pDS
);
```

Parameters

szID

The ID of the connection in the cache.

pDS

[out] Address of the CDataConnection variable that, on success, receives a copy of the data connection.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

See Also

Reference

IDataSourceCache Interface

IDataSourceCache::Add

Other Resources

IDataSourceCache Members

IDataSourceCache::Remove

Call this method to close the connection and remove it from the data source cache.

```
STDMETHOD(Remove)(
   LPCTSTR szID
);
```

Parameters

szID

The ID of the connection in the cache.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Note that closing the connection and removing it from the cache does not affect the connections previously retrieved from the cache since they are copies of the original.

See Also

Reference

IDataSourceCache Interface

Other Resources

IDataSourceCache Members

IDataSourceCache::Uninitialize

Call this method to close all of the data connections and remove them from the cache.

```
STDMETHOD(Uninitialize)();
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Note that closing the connection and removing it from the cache does not affect the connections previously retrieved from the cache since they are copies of the original.

See Also

Reference

IDataSourceCache Interface

Other Resources

IDataSourceCache Members

IDllCache Interface

This interface provides methods for adding DLLs to and retrieving DLLs from a cache.

```
__interface ATL_NO_VTABLE
__declspec(uuid("A12478AB-D261-42f9-B525-7589143C1C97"))
IDllCache :
   public IUnknown
```

Remarks

This interface is implemented by CDllCache, which can be used to cache loaded DLLs to speed access.

Requirements

Header: atlcache.h

See Also

Tasks

Hotswap Sample: Replacing ATL Server Application DLLs at Run Time

Other Resources

IDllCache Members

Caching Reference

Caching

IDllCache Members

Methods

AddRefModule	Call this method to increment the reference count of the DLL represented by the given instance handle.
Flush	Call this method to remove each DLL with a reference count of zero from the cache.
Free	Call this method to decrement the reference count of the DLL in the cache. If the DLL is not in the cache, it is un loaded using FreeLibrary.
GetEntries	Call this method to information about the DLLs in the cache.
Load	Call this method to load a DLL into the cache or get its instance handle from the cache if it's already loaded.
ReleaseModule	Call this method to decrement the reference count of the DLL in the cache.

See Also

Reference

IDllCache Interface

Other Resources

Caching Reference

IDllCache Methods

For information about the methods in **IDllCache**, see IDllCache Members

IDllCache::AddRefModule

Call this method to increment the reference count of the DLL represented by the given instance handle.

```
virtual BOOL AddRefModule(
   HINSTANCE hInstance
);
```

Parameters

hInstance

The instance handle of the DLL.

Return Value

Returns **TRUE** on success, **FALSE** on failure. Returns **FALSE** if the DLL is not in the cache.

Remarks

Increments the reference count for the DLL represented by *hInstance*, ensuring that the DLL is not removed from the cache. DLLs can be removed from the cache only if their reference count is zero.

Example

```
HTTP_CODE AddRefSample()
{
    if (m_hSample == NULL)
    {
        return HTTP_FAIL;
    }

    CComPtr<IDllCache> spDllCache;
    if (FAILED(m_spServiceProvider->QueryService(
                            __uuidof(CMyDllCache), &spDllCache)))
    {
        return HTTP_FAIL;
    }

    if (!spDllCache->AddRefModule(m_hSample))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IDllCache Interface

Other Resources

IDllCache Members

Caching Reference

IDllCache::Flush

Call this method to remove each DLL with a reference count of zero from the cache.

```
virtual HRESULT Flush( );
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The reference count for each DLL is used to determine whether each cached DLL is still being used. Calling **Flush** unloads only those DLLs whose reference count is zero. The reference count of a DLL is controlled by the Load, AddRefModule, Free, and ReleaseModule methods,

Example

```
HTTP_CODE Flush()
{
    CComPtr<IDllCache> spDllCache;
    if (FAILED(m_spServiceProvider->QueryService(
                            __uuidof(CMyDllCache), &spDllCache)))
    {
        return HTTP_FAIL;
    }

    if (FAILED(spDllCache->Flush()))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Tasks

Hotswap Sample: Replacing ATL Server Application DLLs at Run Time

Reference

IDllCache Interface

Other Resources

IDllCache Members

Caching Reference

IDllCache::Free

Call this method to decrement the reference count of the DLL in the cache. If the DLL is not in the cache, it is unloaded using FreeLibrary.

```
virtual BOOL Free(
   HINSTANCE hInstance
);
```

Parameters

hInstance

The instance handle of the DLL.

Return Value

Returns **TRUE** on success, **FALSE** on failure. Returns **FALSE** if the DLL is not in the cache.

Remarks

If the DLL represented by *hInstance* is in the cache, its reference count is decremented. If the reference count reaches zero, the DLL will be unloaded by a future call to Flush.

If the given DLL is not in the cache, **Free** calls FreeLibrary on *hInstance* and returns FALSE. In the same situation, ReleaseModule simply returns FALSE, This is the only respect in which these two methods differ.

Example

```
HTTP_CODE FreeSample()
{
    if (m_hSample == NULL)
    {
        return HTTP_FAIL;
    }

    CComPtr<IDllCache> spDllCache;
    if (FAILED(m_spServiceProvider->QueryService(
                              __uuidof(CMyDllCache), &spDllCache)))
    {
        return HTTP_FAIL;
    }

    if (!spDllCache->Free(m_hSample))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IDllCache Interface

Other Resources

IDllCache Members

Caching Reference

IDllCache::GetEntries

Call this method to information about the DLLs in the cache.

```
virtual HRESULT GetEntries(
   DWORD dwCount,
   DLL_CACHE_ENTRY * pEntries,
   DWORD * pdwCopied
);
```

Parameters

dwCount

The number of entries to retrieve. If zero, the current number of items in the cache is placed in *pdwCopied*.

pEntries

An array of DLL_CACHE_ENTRY structures, each describing an entry in the cache.

pdwCopied

The address of a **DWORD** to receive the number of entries retrieved. On return, this value will be the number of entries in the cache unless *dwCount* is smaller than the entry count.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure. Returns **E_POINTER** if zero is used for *pdwCopied*.

Remarks

Retrieves an array containing a description of each DLL in the cache, including its name, current reference count, and instance handle.

The data retrieved by **GetEntries** reflects the state of the cache at the time the method is called. Subsequent modifications to the cache may make some or all of the information obsolete.

Example

```
CAtlArray<DLL_CACHE_ENTRY> m_entries;
int m_nCurrentEntry;
HINSTANCE m_hSample;

HTTP_CODE LoadEntries()
{
    CComPtr<IDllCache> spDllCache;
    if (FAILED(m_spServiceProvider->QueryService(
                                __uuidof(CMyDllCache), &spDllCache)))
    {
        return HTTP_FAIL;
    }

    DWORD dwCount = 0;
    if (FAILED(spDllCache->GetEntries(0, NULL, &dwCount)))
    {
        return HTTP_FAIL;
    }

    if (dwCount > 0)
    {
        if (!m_entries.SetCount(dwCount))
        {
            return HTTP_FAIL;
        }

        if (FAILED(spDllCache->GetEntries(
                                dwCount, m_entries.GetData(), &dwCount)))
```

```
        {
            return HTTP_FAIL;
        }

        m_entries.SetCount(dwCount);
    }

    m_hSample = NULL;
    for (int i = 0; i < dwCount; i++)
    {
        if (strstr(m_entries[i].szDllName, "SampleDll.dll") != NULL)
        {
            m_hSample = m_entries[i].hInstDll;
            break;
        }
    }

    m_nCurrentEntry = -1;

    return HTTP_SUCCESS;
}
```

See Also

Reference

IDllCache Interface

Other Resources

IDllCache Members

Caching Reference

IDllCache::Load

Call this method to load a DLL into the cache or get its instance handle from the cache if it's already loaded.

```
virtual HINSTANCE Load(
   LPCSTR szFileName,
   void* pPeerInfo
);
```

Parameters

szFileName

A string containing the file name of the DLL to load.

pPeerInfo

The optional address of implementation-specific information about the newly loaded DLL.

Return Value

Returns the instance handle of the newly loaded DLL or NULL if the operation fails. Call GetLastError to get extended error information.

Remarks

Implementations of this method must load the DLL specified by *szFileName* using LoadLibrary and add the name and instance handle to the cache. Implementations can also relate custom information about the DLL back to the caller using *pPeerInfo*.

When the DLL is no longer needed, callers of this method should return the handle to the cache by calling Free or ReleaseModule.

Example

```
HTTP_CODE LoadSample()
{
    _ATLTRY
    {
        CPathA path = m_spServerContext->GetPathTranslated();
        path.RemoveFileSpec();
        path += "\\SampleDll.dll";

        CComPtr<IDllCache> spDllCache;
        if (FAILED(m_spServiceProvider->QueryService(
                          __uuidof(CMyDllCache), &spDllCache)))
        {
            return HTTP_FAIL;
        }

        CMyDllCachePeer::DllInfo info;
        HINSTANCE h = spDllCache->Load(path, &info);
        if (h == NULL)
        {
            return HTTP_FAIL;
        }

        return HTTP_SUCCESS;
    }
    _ATLCATCHALL()
    {
        return HTTP_FAIL;
    }
}
```

See Also

Reference

IDllCache Interface

Other Resources

IDllCache Members

Caching Reference

IDllCache::ReleaseModule

Call this method to decrement the reference count of the DLL in the cache.

```
virtual BOOL ReleaseModule(
   HINSTANCE hInstance
);
```

Parameters

hInstance

The instance handle of the DLL.

Return Value

Returns **TRUE** on success, **FALSE** on failure. Returns **FALSE** if the DLL is not in the cache.

Remarks

If the DLL represented by *hInstance* is in the cache, its reference count is decremented. If the reference count reaches zero, the DLL will be unloaded by a future call to Flush.

If the given DLL is not in the cache, **ReleaseModule** simply returns FALSE. In the same situation, Free calls FreeLibrary on *hInstance* and returns FALSE, This is the only respect in which these two methods differ.

Example

```
HTTP_CODE ReleaseSample()
{
    if (m_hSample == NULL)
    {
        return HTTP_FAIL;
    }

    CComPtr<IDllCache> spDllCache;
    if (FAILED(m_spServiceProvider->QueryService(
                        __uuidof(CMyDllCache), &spDllCache)))
    {
        return HTTP_FAIL;
    }

    if (!spDllCache->ReleaseModule(m_hSample))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IDllCache Interface

Other Resources

IDllCache Members

Caching Reference

IDllCacheMgr Interface

This interface provides methods for managing a DLL cache.

```
[ uuid("A0C00AF8-CEA5-46b9-97ED-FDEE55B583EF"), object ]
__interface IDllCacheMgr
```

Remarks

This interface is implemented by CDllCacheManager.

Requirements

Header: atlextmgmt.h

See Also

Other Resources

IDllCacheMgr Members

IDllCacheMgr Members

Methods

GetEntries	Call this method to retrieve entries from the DLL cache.
GetEntryCount	Call this method to get the number of entries in the cache.

See Also

Reference

IDllCacheMgr Interface

IDllCacheMgr Methods

For information about the methods in **IDllCacheMgr**, see IDllCacheMgr Members

IDllCacheMgr::GetEntries

Call this method to retrieve entries from the DLL cache.

```
[ id( 0 ) ]
STDMETHOD(GetEntries)(
   [in] DWORD dwCount,
   [out] _DLL_CACHE_ENTRY * pEntries,
   [out, retval] DWORD * pdwCopied
);
```

Parameters

dwCount

The number of entries to retrieve. If zero, the current number of items in the cache is placed in *pdwCopied*.

pEntries

An array of _DLL_CACHE_ENTRY structures, each describing an entry in the cache.

pdwCopied

The address of a variable that, on success, receives the number of entries retrieved.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Retrieves an array containing a description of each DLL in the cache, including its name, current reference count, and instance handle.

See Also

Reference

IDllCacheMgr Interface

Other Resources

IDllCacheMgr Members

IDllCacheMgr::GetEntryCount

Call this method to get the number of entries in the cache.

```
[ id( 1 ) ]
STDMETHOD(GetEntryCount)(
   [out, retval] DWORD * pdwCount
);
```

Parameters

pdwCount

[out] Address of the variable that, on success, receives the number of entries in the cache.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

IDllCacheMgr Interface

Other Resources

IDllCacheMgr Members

IFileCache Interface

This interface provides methods for adding and retrieving files from a cache.

```
__interface ATL_NO_VTABLE
__declspec(uuid("105A8866-4059-45fe-86AE-FA0EABBFBBB4"))
IFileCache :
   public IUnknown
```

Remarks

The cache implementing this interface is for temporary files. The files added to this cache will be deleted from disk when the file is removed from the cache.

This interface is implemented by CFileCache.

Requirements

Header: atlcache.h

See Also

Other Resources

IFileCache Members
Caching Reference
Caching

IFileCache Members

Methods

AddFile	Call this method to add a file to the cache.
Flush	Call this method to remove expired entries from the file cache.
GetFile	Call this method to retrieve the cached file name and peer information.
LookupFile	Call this method to get a handle to a cached file given its file name.
ReleaseFile	Call this method to release a file from a file cache.
RemoveFile	Call this method to remove a file from a file cache.
RemoveFileByName	Call this method to remove a file from the file cache, given the file name.

See Also

Reference

IFileCache Interface

Other Resources

Caching Reference

IFileCache Methods

For information about the methods in **IFileCache**, see IFileCache Members.

IFileCache::AddFile

Call this method to add a file to the cache.

```
STDMETHOD(AddFile)(
   LPCSTR szFileName,
   LPCSTR szTempFileName,
   FILETIME* pftExpireTime,
   void* pPeerInfo,
   HCACHEITEM* phKey
);
```

Parameters

szFileName

A string containing the name of the file to be added to the cache. This argument is used as the cache key, and does not necessarily have to contain a valid file name.

szTempFileName

A string containing the name of a temporary file. This must be a valid file name, indicating an existing file. This is the file name that will be retrieved when cache entries are retrieved. Note that the file specified by this argument will be deleted from the disk when its cache entry is removed.

pftExpireTime

The expiration time, after which the file becomes eligible for being flushed from the cache. An expiration time of zero indicates that the file will not expire.

pPeerInfo

An optional address where implementation-specific data can be stored. Use zero if no peer data is desired.

phKey

The address of a cache key that can be used to identify the cached entry in subsequent operations.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The cache item handle retrieved as *phKey* can be used with the GetFile, ReleaseFile, and RemoveFile methods.

Example

```
HTTP_CODE SubmitMessage()
{
    CComPtr<IFileCache> spFileCache;

    if (FAILED(m_spServiceProvider->QueryService(
                                    __uuidof(IFileCache), &spFileCache)))
    {
        return HTTP_FAIL;
    }

    CAtlTemporaryFile file;
    if (FAILED(file.Create()))
    {
        return HTTP_FAIL;
    }

    if (FAILED(file.Write(m_strMessage, m_strMessage.GetLength())))
    {
        return HTTP_FAIL;
    }
```

```
    if (FAILED(file.HandsOff()))
    {
        return HTTP_FAIL;
    }

    HCACHEITEM hItem;
    CFileTime ft = CFileTime::GetCurrentTime() +
                        CFileTimeSpan(m_nKeepSeconds * CFileTime::Second);
    if (FAILED(spFileCache->AddFile(m_strMessageName, file.TempFileName(),
                                                      &ft, NULL, &hItem)))
    {
        file.HandsOn();
        return HTTP_FAIL;
    }

    spFileCache->ReleaseFile(hItem);

    return HTTP_SUCCESS;
}
```

See Also

Reference

IFileCache Interface

Other Resources

IFileCache Members

Caching Reference

Caching

IFileCache::Flush

Call this method to remove expired entries from the file cache.

```
STDMETHOD(Flush)( );
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The **Flush** method iterates through the cache entries, and removes any files whose expiration time has passed.

Example

```
HTTP_CODE FlushCache()
{
    CComPtr<IFileCache> spFileCache;

    if (FAILED(m_spServiceProvider->QueryService(
                                __uuidof(IFileCache), &spFileCache)))
    {
        return HTTP_FAIL;
    }

    if (FAILED(spFileCache->Flush()))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IFileCache Interface

Other Resources

IFileCache Members

Caching Reference

Caching

IFileCache::GetFile

Call this method to retrieve the cached file name and peer information.

```
STDMETHOD(GetFile)(
   const HCACHEITEM hKey,
   LPSTR* pszFileName,
   void** ppPeerInfo
);
```

Parameters

hKey

A cache entry handle, as retrieved previously with AddFile or LookupFile.

pszFileName

The address of a string where the name of the cached file is to be stored. If successful, this will be the name that was previously passed to AddFile as the *szTempFileName* argument.

ppPeerInfo

The address of a buffer where file peer information is to be stored. This argument can be zero if no peer information is to be retrieved.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

See IFileCache::LookupFile.

See Also

Reference

IFileCache Interface

Other Resources

IFileCache Members

Caching Reference

Caching

IFileCache::LookupFile

Call this method to get a handle to a cached file given its file name.

```
STDMETHOD(LookupFile)(
   LPCSTR szFileName,
   HCACHEITEM* phKey
);
```

Parameters

szFileName

A string containing the name of the cached file for which a handle is to be retrieved.

phKey

The address of a cache item handle that, on success, will identify the file specified by *szFileName*.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

```
HTTP_CODE RetrieveMessage()
{
    m_bFound = false;

    CComPtr<IFileCache> spFileCache;

    if (FAILED(m_spServiceProvider->QueryService(
                                __uuidof(IFileCache), &spFileCache)))
    {
        return HTTP_FAIL;
    }

    HCACHEITEM hItem;
    if (FAILED(spFileCache->LookupFile(m_strMessageName, &hItem)))
    {
        return HTTP_SUCCESS;
    }

    LPSTR szFileName;
    HRESULT hr = spFileCache->GetFile(hItem, &szFileName, NULL);
    if (FAILED(hr))
    {
        spFileCache->ReleaseFile(hItem);
        return HTTP_FAIL;
    }

    CAtlFile file;
    if (FAILED(file.Create(szFileName, GENERIC_READ, FILE_SHARE_READ,
                                                     OPEN_EXISTING)))
    {
        spFileCache->ReleaseFile(hItem);
        return HTTP_FAIL;
    }

    ULONGLONG nSize;
    if (FAILED(file.GetSize(nSize)))
    {
        spFileCache->ReleaseFile(hItem);
        return HTTP_FAIL;
    }
```

```
    _ATLTRY
    {
        CStrBufA buf(m_strMessage, (int) nSize, CStrBufA::SET_LENGTH);
        if (FAILED(file.Read(buf, nSize)))
        {
            spFileCache->ReleaseFile(hItem);
            return HTTP_FAIL;
        }
    }
    _ATLCATCHALL()
    {
        spFileCache->ReleaseFile(hItem);
        return HTTP_FAIL;
    }

    spFileCache->ReleaseFile(hItem);
    m_bFound = true;

    return HTTP_SUCCESS;
}
```

See Also

Reference

IFileCache Interface

Other Resources

IFileCache Members

Caching Reference

Caching

IFileCache::ReleaseFile

Call this method to release a file from a file cache.

```
STDMETHOD(ReleaseFile)(
   const HCACHEITEM hKey
);
```

Parameters

hKey

The handle of the file to be released.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The **ReleaseFile** method decrements the reference count for the file indicated by *hKey*. The file is not deleted unless its reference count reaches zero.

Example

See IFileCache::LookupFile.

See Also

Reference

IFileCache Interface

Other Resources

IFileCache Members

Caching Reference

Caching

IFileCache::RemoveFile

Call this method to remove a file from a file cache.

```
STDMETHOD(RemoveFile)(
   const HCACHEITEM hKey
);
```

Parameters

hKey

The handle of the file to remove.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The **RemoveFile** method decrements the reference count for the file indicated by *hKey*, and removes it from the cache, but does not delete the file unless its reference count reaches zero.

See Also

Reference

IFileCache Interface

Other Resources

IFileCache Members

Caching Reference

Caching

IFileCache::RemoveFileByName

Call this method to remove a file from the file cache, given the file name.

```
STDMETHOD(RemoveFileByName)(
   LPCSTR szFileName
);
```

Parameters

szFileName

A string containing the name of the file to be removed.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The **RemoveFileByName** method locates the file in the cache, decrements its reference count, and removes it from the cache, but does not delete the file unless its reference count reaches zero.

Example

```
HTTP_CODE RemoveMessage()
{
    CComPtr<IFileCache> spFileCache;

    if (FAILED(m_spServiceProvider->QueryService(
                            __uuidof(IFileCache), &spFileCache)))
    {
        return HTTP_FAIL;
    }

    if (FAILED(spFileCache->RemoveFileByName(m_strMessageName)))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IFileCache Interface

Other Resources

IFileCache Members

Caching Reference

Caching

IHttpFile Interface

This interface provides read-only access to information for a file that has been uploaded to the server.

```
__interface IHttpFile
```

Remarks

This interface is implemented by CHttpRequestFile.

Requirements

Header: atlsiface.h

Example

```
[ tag_name(name="PrintFileInfo") ]
HTTP_CODE OnPrintFileInfo(void)
{
    POSITION pos = m_HttpRequest.m_Files.GetStartPosition();
    while (pos != NULL)
    {
        IHttpFile* pFile = m_HttpRequest.m_Files.GetNextValue(pos);
        m_HttpResponse
            << "Uploaded file name : "
            << pFile->GetFileName()
            << "<br>Full uploaded file name : "
            << pFile->GetFullFileName()
            << "<br>Temporary file name : "
            << pFile->GetTempFileName()
            << "<br>Content-Type : "
            << pFile->GetContentType()
            << "<br>File Size : "
            << pFile->GetFileSize()
            << "<br>Parameter name : "
            << pFile->GetParamName()
            << "<hr>";
    }

    return HTTP_SUCCESS;
}
```

See Also

Tasks

Mailer Sample: Demonstrates SMTP Mail Classes

RegExp Sample: Demonstrates ATL Regular Expression Classes

Reference

ShowFiles

ShowImage

CHttpRequestFile Class

Other Resources

IHttpFile Members

ATL Server Classes

IHttpFile Members

Methods

Free	Call this method to delete the object exposing this interface.
GetContentType	Call this method to get the MIME type of the uploaded file.
GetFileName	Call this method to get the original file name of the uploaded file as set by the client.
GetFileSize	Call this method to get the size of the file in bytes.
GetFullFileName	Call this method to get the original path and file name of the uploaded file as set by the client.
GetParamName	Call this method to get the name of the INPUT element used to upload the file.
GetTempFileName	Call this method to get the name of the uploaded file on the server.

See Also

Reference

IHttpFile Interface

IHttpFile Methods

For information about the methods in **IHttpFile**, see IHttpFile Members.

IHttpFile::Free

Call this method to delete the object exposing this interface.

```
void Free( );
```

See Also

Reference

IHttpFile Interface

Other Resources

IHttpFile Members

IHttpFile::GetContentType

Call this method to get the MIME type of the uploaded file.

```
LPCSTR GetContentType( );
```

Return Value

Returns the MIME type of the uploaded file.

Example

See IHttpFile Overview.

See Also

Reference

ShowFiles
ShowImage
IHttpFile Interface

Other Resources

IHttpFile Members

IHttpFile::GetFileName

Call this method to get the original file name of the uploaded file as set by the client.

```
LPCSTR GetFileName( );
```

Return Value

Returns the original file name of the uploaded file as set by the client.

Remarks

To get the original path and file name of the uploaded file as set by the client, call IHttpFile::GetFullFileName.

To get the name of the temporary file created on the server, call IHttpFile::GetTempFileName.

Example

See IHttpFile Overview.

See Also

Tasks

Mailer Sample: Demonstrates SMTP Mail Classes

Reference

ShowFiles
ShowImage
IHttpFile Interface
IHttpFile::GetFullFileName
IHttpFile::GetTempFileName

Other Resources

IHttpFile Members

IHttpFile::GetFileSize

Call this method to get the size of the file in bytes.

```
ULONGLONG GetFileSize( );
```

Return Value

Returns the size of the file in bytes.

Example

See IHttpFile Overview.

See Also

Tasks

RegExp Sample: Demonstrates ATL Regular Expression Classes

Reference

ShowFiles

ShowImage

IHttpFile Interface

Other Resources

IHttpFile Members

IHttpFile::GetFullFileName

Call this method to get the original path and file name of the uploaded file as set by the client.

```
LPCSTR GetFullFileName( );
```

Return Value

Returns the full file name.

Remarks

To get the original file name of the uploaded file as set by the client without the path, call IHttpFile::GetFileName.

To get the name of the temporary file created on the server, call IHttpFile::GetTempFileName.

Example

See IHttpFile Overview.

See Also

Reference

ShowFiles
ShowImage
IHttpFile Interface
IHttpFile::GetFileName
IHttpFile::GetTempFileName

Other Resources

IHttpFile Members

IHttpFile::GetParamName

Call this method to get the name of the INPUT element used to upload the file.

```
LPCSTR GetParamName( );
```

Return Value

Returns the name of the INPUT element used to upload the file.

Example

See IHttpFile Overview.

See Also

Reference

ShowFiles
ShowImage
IHttpFile Interface

Other Resources

IHttpFile Members

IHttpFile::GetTempFileName

Call this method to get the name of the uploaded file on the server.

```
LPCSTR GetTempFileName( );
```

Return Value

Returns the name of the uploaded file on the server.

Remarks

This method returns the local file system path of the file.

To get the original path and file name of the uploaded file as set by the client, call IHttpFile::GetFullFileName.

To get the original file name of the uploaded file as set by the client without the path, call IHttpFile::GetFileName.

Example

See IHttpFile Overview.

See Also

Tasks

Mailer Sample: Demonstrates SMTP Mail Classes
RegExp Sample: Demonstrates ATL Regular Expression Classes

Reference

ShowFiles
ShowImage
IHttpFile Interface
IHttpFile::GetFullFileName
IHttpFile::GetFileName

Other Resources

IHttpFile Members

IHttpRequestLookup Interface

This interface provides access to the file, form variable, and query parameter collections associated with an HTTP request.

```
__interface ATL_NO_VTABLE
__declspec( uuid( "A5990B44-FF74-4bfe-B66D-F9E7E9F42D42" ))
IHttpRequestLookup :
   public IUnknown
```

Remarks

This interface is implemented by CHttpRequest.

Requirements

Header: atlsiface.h

Example

```
HTTP_CODE InitializeChild(
    AtlServerRequest* pRequestInfo,
    IServiceProvider* pProvider,
    IHttpRequestLookup* pLookup
    )
{
    _ATLTRY
    {
        // This override of InitializeChild shows how to use
        // the IHttpRequestLookup interface to obtain
        // query parameters, form variabless,
        // and files from a parent handler.

        HTTP_CODE hcErr = InitializeInternal(pRequestInfo, pProvider);
        if (!hcErr)
        {
            CComPtr<IHttpServerContext> spServerContext;
            HRESULT hr = pLookup->GetServerContext(&spServerContext);

            if (pRequestInfo->pServerContext != spServerContext)
            {
                m_HttpResponse.Initialize(pRequestInfo->pServerContext);
            }
            else
            {
                m_HttpResponse.Initialize(spServerContext);
            }

            m_HttpResponse.HaveSentHeaders(TRUE);

            // Query parameters.
            m_HttpResponse << "Query Params:<br>";
            LPCSTR szName = NULL;
            LPCSTR szValue = NULL;

            POSITION pos = pLookup->GetFirstQueryParam(&szName, &szValue);
            while (pos != NULL)
            {
                m_HttpResponse << "name : " << szName
                               << ", value : " << szValue
                               << "<br>";
                pos = pLookup->GetNextQueryParam(pos, &szName, &szValue);
            }

            // Form variables.
            m_HttpResponse << "<hr>Form Vars:<br>";
```

```
            pos = pLookup->GetFirstFormVar(&szName, &szValue);
            while (pos != NULL)
            {
                m_HttpResponse << "name : " << szName
                               << ", value : " << szValue
                               << "<br>";
                pos = pLookup->GetNextFormVar(pos, &szName, &szValue);
            }

            // Files.
            m_HttpResponse << "<hr>Uploaded Files:<br>";
            IHttpFile *pFile = NULL;
            pos = pLookup->GetFirstFile(&szName, &pFile);
            while (pos != NULL)
            {
                m_HttpResponse << "name : " << szName
                               << ", temp file : "
                               << pFile->GetTempFileName()
                               << "<br>";
                pos = pLookup->GetNextFile(pos, &szName, &pFile);
            }
        }
    }
    _ATLCATCHALL()
    {
        return HTTP_ERROR(500, ISE_SUBERR_OUTOFMEM);
    }

    m_HttpResponse.Flush(TRUE);
    return HTTP_SUCCESS_NO_PROCESS;
}
```

See Also

Other Resources

IHttpRequestLookup Members

ATL Server Classes

IHttpRequestLookup Members

Methods

GetFirstFile	Call this method to get the first file associated with an HTTP request.
GetFirstFormVar	Call this method to get the first form variable associated with an HTTP request.
GetFirstQueryParam	Call this method to get the first query parameter associated with an HTTP request.
GetNextFile	Call this method to get the next file associated with an HTTP request.
GetNextFormVar	Call this method to get the next form variable associated with an HTTP request.
GetNextQueryParam	Call this method to get the next query parameter associated with an HTTP request.
GetServerContext	Call this method to get the server context associated with an HTTP request.

See Also

Reference

IHttpRequestLookup Interface

IHttpRequestLookup Methods

For information about the methods in **IHttpRequestLookup**, see IHttpRequestLookup Members.

IHttpRequestLookup::GetFirstFile

Call this method to get the first file associated with an HTTP request.

```
POSITION GetFirstFile(
   LPCSTR * ppszName,
   IHttpFile ** ppFile
);
```

Parameters

ppszName

The address of an LPCSTR variable that receives the local name of the file.

ppFile

The address of an **IHttpFile*** variable that receives the interface pointer to the file.

Return Value

Returns the POSITION of the next file in the collection or NULL if there are no files associated with this request.

Remarks

The local name of the file in *ppszName* is the same as the name returned by IHttpFile::GetTempFileName.

The POSITION returned by this method can be passed to IHttpRequestLookup::GetNextFile to get the next file in the collection.

Example

See IHttpRequestLookup Overview.

See Also

Reference

IHttpRequestLookup Interface
IHttpRequestLookup::GetNextFile
IHttpFile::GetTempFileName

Other Resources

IHttpRequestLookup Members

IHttpRequestLookup::GetFirstFormVar

Call this method to get the first form variable associated with an HTTP request.

```
POSITION GetFirstFormVar(
   LPCSTR * ppszName,
   LPCSTR * ppszValue
);
```

Parameters

ppszName

The address of an LPCSTR variable that receives the name of the form variable.

ppszValue

The address of an LPCSTR variable that receives the value of the form variable.

Return Value

Returns the POSITION of the next form variable in the collection or NULL if there are no form variables associated with this request.

Remarks

The POSITION returned by this method can be passed to IHttpRequestLookup::GetNextFormVar to get the next form variable in the collection.

Example

See IHttpRequestLookup Overview.

See Also

Reference

IHttpRequestLookup Interface
IHttpRequestLookup::GetNextFormVar

Other Resources

IHttpRequestLookup Members

IHttpRequestLookup::GetFirstQueryParam

Call this method to get the first query parameter associated with an HTTP request.

```
POSITION GetFirstQueryParam(
   LPCSTR * ppszName,
   LPCSTR * ppszValue
);
```

Parameters

ppszName

The address of an LPCSTR variable that receives the name of the query parameter.

ppszValue

The address of an LPCSTR variable that receives the value of the query parameter.

Return Value

Returns the POSITION of the next query parameter in the collection or NULL if there are no query parameters associated with this request.

Remarks

The POSITION returned by this method can be passed to IHttpRequestLookup::GetNextQueryParam to get the next query parameter in the collection.

Example

See IHttpRequestLookup Overview.

See Also

Reference

IHttpRequestLookup Interface
IHttpRequestLookup::GetNextQueryParam

Other Resources

IHttpRequestLookup Members

IHttpRequestLookup::GetNextFile

Call this method to get the next file associated with an HTTP request.

```
POSITION GetNextFile(
   POSITION pos,
   LPCSTR * ppszName,
   IHttpFile ** ppFile
);
```

Parameters

pos

The position of the file.

ppszName

The address of an LPCSTR variable that receives the local name of the file.

ppFile

The address of an **IHttpFile*** variable that receives the interface pointer to the file.

Return Value

Returns the POSITION of the next file in the collection associated with the HTTP request or NULL if there are no further files.

Remarks

The POSITION passed to this method should have been returned by a previous call to IHttpRequestLookup::GetFirstFile or **IHttpRequestLookup::GetNextFile** on the same object.

The local name of the file in *ppszName* is the same as the name returned by IHttpFile::GetTempFileName.

The POSITION returned by this method can be passed to **IHttpRequestLookup::GetNextFile** to get the next file in the collection.

Example

See IHttpRequestLookup Overview.

See Also

Reference

IHttpRequestLookup Interface
IHttpRequestLookup::GetFirstFile
IHttpFile::GetTempFileName

Other Resources

IHttpRequestLookup Members

IHttpRequestLookup::GetNextFormVar

Call this method to get the next form variable associated with an HTTP request.

```
POSITION GetNextFormVar(
   POSITION pos,
   LPCSTR * ppszName,
   LPCSTR * ppszValue
);
```

Parameters

pos

The position of the form variable.

ppszName

The address of an LPCSTR variable that receives the name of the form variable.

ppszValue

The address of an LPCSTR variable that receives the value of the form variable.

Return Value

Returns the POSITION of the next form variable in the collection associated with the HTTP request or NULL if there are no further form variables.

Remarks

The POSITION passed to this method should have been returned by a previous call to IHttpRequestLookup::GetFirstFormVar or **IHttpRequestLookup::GetNextFormVar** on the same object.

The POSITION returned by this method can be passed to **IHttpRequestLookup::GetNextFormVar** to get the next form variable in the collection.

Example

See IHttpRequestLookup Overview.

See Also

Reference

IHttpRequestLookup Interface
IHttpRequestLookup::GetFirstFormVar

Other Resources

IHttpRequestLookup Members

IHttpRequestLookup::GetNextQueryParam

Call this method to get the next query parameter associated with an HTTP request.

```
POSITION GetNextQueryParam(
   POSITION pos,
   LPCSTR * ppszName,
   LPCSTR * ppszValue
);
```

Parameters

pos

The position of the query parameter.

ppszName

The address of an LPCSTR variable that receives the name of the query parameter.

ppszValue

The address of an LPCSTR variable that receives the value of the query parameter.

Return Value

Returns the POSITION of the next query parameter in the collection associated with the HTTP request or NULL if there are no further query parameters.

Remarks

The POSITION passed to this method should have been returned by a previous call to IHttpRequestLookup::GetFirstQueryParam or **IHttpRequestLookup::GetNextQueryParam** on the same object.

The POSITION returned by this method can be passed to **IHttpRequestLookup::GetNextQueryParam** to get the next query parameter in the collection.

Example

See IHttpRequestLookup Overview.

See Also

Reference

IHttpRequestLookup Interface
IHttpRequestLookup::GetFirstQueryParam

Other Resources

IHttpRequestLookup Members

IHttpRequestLookup::GetServerContext

Call this method to get the server context associated with an HTTP request.

```
HRESULT GetServerContext(
   IHttpServerContext** ppOut
);
```

Parameters

ppOut

[out] Address of the pointer variable that, on success, receives the IHttpServerContext interface pointer for the current request.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See IHttpRequestLookup Overview.

See Also

Reference

IHttpRequestLookup Interface

IHttpServerContext Interface

Other Resources

IHttpRequestLookup Members

IHttpServerContext Interface

This interface encapsulates the capabilities of the Web server and provides information about the current request being handled.

```
__interface ATL_NO_VTABLE
__declspec(uuid("813F3F00-3881-11d3-977B-00C04F8EE25E"))
IHttpServerContext :
   public IUnknown
```

Remarks

This interface provides methods that are similar to the capabilities of the EXTENSION_CONTROL_BLOCK structure. CServerContext provides the most commonly used implementation of this interface. Other implementations of this interface are provided by CCacheServerContext, CIDServerContext, CIncludeServerContext and CTransferServerContext.

Requirements

Header: atlsiface.h

Example

See the following samples:

- CLStencil Sample
- ForceLogin Sample
- MantaWeb Sample
- NotModified Sample
- ShowErrors Sample
- ShowImage Sample
- ShowUser Sample
- SOAPDebugApp Sample

See Also

Reference

NotModified
ForceLogin
ShowUser
CServerContext Class
CCacheServerContext Class
CIDServerContext Class
CIncludeServerContext Class

Other Resources

IHttpServerContext Members
ATL Server Classes

IHttpServerContext Members

Methods

AppendToLog	Call this method to append a string to the Web server's log for the current request.
AsyncReadClient	Call this method to asynchronously read information from the body of the Web client's HTTP request into the buffer supplied by the caller.
AsyncWriteClient	Call this method to asynchronously send the data present in the supplied buffer to the client that made the request.
DoneWithSession	Call this method to tell the HTTP server context object that you are done with the current session.
GetAvailableBytes	Call this method to get the available bytes from the HTTP server context object.
GetAvailableData	Call this method to get at the available data from the current request.
GetContentType	Call this method to get the content type of the current request.
GetImpersonationToken	Call this method to retrieve a handle to the impersonation token for this request.
GetPathInfo	Call this method to get the path of the current request from the HTTP server context object.
GetPathTranslated	Call this method to get the translated path of the requested resource from the HTTP server context object.
GetQueryString	Call this method to get the query string from the HTTP server context object.
GetRequestMethod	Call this method to get the request method from the HTTP server context object.
GetScriptPathTranslated	Call this method to get the translated path of the script handling the current request from the HTTP server context object.
GetServerVariable	Call this method to get the value of a server variable from the HTTP server context object.
GetTotalBytes	Call this method to get the total number of bytes in the current request.
MapUrlToPathEx	Call this method to get a physical file system path corresponding to a logical URL path.
ReadClient	Call this method to synchronously read information from the body of the Web client's HTTP request into the buffer supplied by the caller.
RequestIOCompletion	Call this method to set a special callback function that will be used for handling the completion of asynchronous I/O operations.
SendRedirectResponse	Call this method to redirect the client to the specified URL.
SendResponseHeader	Call this method to send an HTTP response header to the client.
TransmitFile	Call this method to transmit a file asynchronously to the client.

WriteClient	Call this method to synchronously send the data present in the supplied buffer to the client that made the request.

See Also

Reference

IHttpServerContext Interface

IHttpServerContext Methods

For information about the methods in **IHttpServerContext**, see IHttpServerContext Members.

IHttpServerContext::AppendToLog

Call this method to append a string to the Web server's log for the current request.

```
BOOL AppendToLog(
   LPCSTR szMessage,
   DWORD* pdwLen
);
```

Parameters

szMessage

The message to log.

pdwLen

Points to a DWORD containing the number of bytes in *szMessage*.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This method is equivalent to the **HSE_APPEND_LOG_PARAMETER** server support function.

Example

```
[ tag_name(name="AppendToLog") ]
HTTP_CODE OnAppendToLog(char* szLogEntry)
{
    // This replacement method will append the log entry
    // specified in the SRF file to the server context's
    // log for the current request.

    DWORD dwLen = strlen(szLogEntry);
    HRESULT hr = m_spServerContext->AppendToLog(szLogEntry, &dwLen);
    if (SUCCEEDED(hr))
    {
        m_HttpResponse << "Appended to log: " << szLogEntry;
        return HTTP_SUCCESS;
    }
    return HTTP_FAIL;
}
```

See Also

Reference

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IHttpServerContext::AsyncReadClient

Call this method to asynchronously read information from the body of the Web client's HTTP request into the buffer supplied by the caller.

```
BOOL AsyncReadClient(
   void* pvBuffer,
   DWORD* pdwSize
);
```

Parameters

pvBuffer

Points to the buffer allocated by the caller that will receive the requested information.

pdwSize

Points to the number of bytes that can be held by the buffer pointed to by *pvBuffer*.

Return Value

Returns **TRUE** on success, **FALSE** on failure. Call GetLastError to get extended error information.

Remarks

To call this function successfully, someone must have already called IHttpServerContext::RequestIOCompletion specifying the callback function to be used for I/O completion.

This method is equivalent to the **HSE_REQ_ASYNC_READ_CLIENT** server support function.

See Also

Reference

IHttpServerContext Interface

IHttpServerContext::RequestIOCompletion

Other Resources

IHttpServerContext Members

IHttpServerContext::AsyncWriteClient

Call this method to asynchronously send the data present in the supplied buffer to the client that made the request.

```
BOOL AsyncWriteClient(
   void* pvBuffer,
   DWORD* pdwBytes
);
```

Parameters

pvBuffer

The buffer containing the data to be sent to the client.

pdwBytes

Pointer to a DWORD set to the number of bytes in the buffer.

Return Value

Returns **TRUE** on success, **FALSE** on failure. Call GetLastError to get extended error information.

Remarks

To call this function successfully someone must have already called IHttpServerContext::RequestIOCompletion specifying the callback function to be used for I/O completion.

This method is equivalent to **EXTENSION_CONTROL_BLOCK::WriteClient**(..., HSE_IO_ASYNC).

See Also

Reference

IHttpServerContext Interface

IHttpServerContext::RequestIOCompletion

Other Resources

IHttpServerContext Members

IHttpServerContext::DoneWithSession

Call this method to tell the HTTP server context object that you are done with the current session.

```
BOOL DoneWithSession(
   DWORD dwHttpStatusCode
);
```

Parameters

dwHttpStatusCode

The status code.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This method is equivalent to the **HSE_REQ_DONE_WITH_SESSION** server support function.

See Also

Reference

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IHttpServerContext::GetAvailableBytes

Call this method to get the available bytes from the HTTP server context object.

```
DWORD GetAvailableBytes( );
```

Return Value

Returns the number of bytes available in the request buffer accessible through IHttpServerContext::GetAvailableData.

Remarks

If **GetAvailableBytes** returns the same value as IHttpServerContext::GetTotalBytes, the request buffer contains the whole request. Otherwise, the remaining data should be read from the client using IHttpServerContext::ReadClient or IHttpServerContext::AsyncReadClient.

This method is equivalent to **EXTENSION_CONTROL_BLOCK::cbAvailable**.

Example

```
[ tag_name(name="ReadClientData") ]
HTTP_CODE OnReadClientData()
{
    // This replacement method will read all data from the client.

    // Set the amount to read equal to the total number of bytes.
    DWORD dwToRead = m_spServerContext->GetTotalBytes();

    // Try to allocate space to read all the data.
    CHeapPtr<BYTE> spData;
    if (!spData.AllocateBytes(dwToRead))
    {
        return AtlsHttpError(500, ISE_SUBERR_OUTOFMEM);
    }

    LPBYTE pbDest = spData;

    // Get the number of available bytes.
    DWORD dwAvailableBytes = m_spServerContext->GetAvailableBytes();
    DWORD dwRead(0);

    // Read from available data first.
    if (dwAvailableBytes)
    {
        // Get the available data.
        LPBYTE pbAvailableData = m_spServerContext->GetAvailableData();

        // Copy the available data into the allocated buffer.
        memcpy(pbDest, pbAvailableData, dwAvailableBytes);

        // Update the amount left to read.
        dwToRead -= dwAvailableBytes;

        // Update the buffer pointer.
        pbDest += dwAvailableBytes;

        // Update the bytes read.
        dwRead += dwAvailableBytes;
    }

    // Check if there is still more to read after the available data
    // is exhausted.
    if (dwToRead)
    {
        // Set the number of bytes to read.
```

```
        DWORD dwClientBytesToRead = dwToRead;
        DWORD dwClientBytesRead = 0;

        // Keep on reading until the amount requested has been read.
        do
        {
            // Set the number of bytes to read.
            dwClientBytesRead = dwClientBytesToRead;

            // Read data from the client.
            if (!m_spServerContext->ReadClient(
                                        pbDest, &dwClientBytesRead))
            {
                return HTTP_FAIL;
            }

            // Update the number of bytes left to read.
            dwClientBytesToRead -= dwClientBytesRead;

            // Update the buffer pointer.
            pbDest += dwClientBytesRead;

        } while (dwClientBytesToRead != 0 && dwClientBytesRead != 0);

        // Update the total read with the bytes read in the loop.
        dwRead += dwToRead - dwClientBytesToRead;
    }

    // Print out the number of bytes read from the client.
    m_HttpResponse << "Read " << dwRead << " bytes from client";

    return HTTP_SUCCESS;
}
```

See Also

Reference

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IHttpServerContext::GetAvailableData

Call this method to get at the available data from the current request.

```
BYTE* GetAvailableData( );
```

Return Value

Returns a pointer to the request buffer containing the data sent by the client.

Remarks

The size of the available data can be determined by calling IHttpServerContext::GetAvailableBytes.

This method is equivalent to EXTENSION_CONTROL_BLOCK::lpdData.

Example

See IHttpServerContext::GetAvailableBytes.

See Also

Reference

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IHttpServerContext::GetContentType

Call this method to get the content type of the current request.

```
LPCSTR GetContentType( );
```

Return Value

Returns a null-terminated string that contains the content type of the data sent by the client.

Remarks

This method is equivalent to the **CONTENT_TYPE** server variable.

Example

```
[ tag_name(name="GetContentType") ]
HTTP_CODE OnGetContentType()
{
    // This replacement method will get the
    // content-type of the current request
    // and return it to the client.

    LPCSTR szContentType = m_spServerContext->GetContentType();
    if (szContentType != NULL)
    {
        m_HttpResponse << "Content-type : " << szContentType;
    }
    else
    {
        m_HttpResponse << "No content-type";
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IHttpServerContext::GetImpersonationToken

Call this method to retrieve a handle to the impersonation token for this request.

```
BOOL GetImpersonationToken(
   HANDLE* pToken
);
```

Parameters

pToken

Pointer to a handle to receive the impersonation token.

Return Value

Returns **TRUE** on success, and **FALSE** on failure.

Remarks

An impersonation token represents a user context. You can use the handle in calls to ImpersonateLoggedOnUser or SetThreadToken. Do not call CloseHandle on the handle.

This method is equivalent to the **HSE_REQ_GET-IMPERSONATION** server support function.

Example

```
[ tag_name(name="GetImpersonationToken") ]
HTTP_CODE OnGetImpersonationToken()
{
    // This replacement method will determine if the
    // user is an administrator.

    // Use HTTP_UNAUTHORIZED return value to force authentication.
    HTTP_CODE hcErr = HTTP_UNAUTHORIZED;
    HANDLE hToken = NULL;

    // Get the user token.
    BOOL bRet = m_spServerContext->GetImpersonationToken(&hToken);
    if (bRet)
    {
        CAccessToken tok;
        tok.Attach(hToken);

        // Get the token groups.
        CTokenGroups tokGroups;
        if (tok.GetGroups(&tokGroups))
        {
            // Determine whether the Administrators group
            // is one of the user's groups.
            if (tokGroups.LookupSid(Sids::Admins()))
            {
                // The user is an administrator.
                hcErr = HTTP_SUCCESS;
            }
        }

        // Do not close the handle to the user's impersonation token.
        tok.Detach();
    }

    if (hcErr == HTTP_SUCCESS)
    {
        m_HttpResponse << "The user is an administrator";
    }

    return hcErr;
```

```
}
```

See Also

Reference

ShowUser

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IHttpServerContext::GetPathInfo

Call this method to get the path of the current request from the HTTP server context object.

```
LPCSTR GetPathInfo( );
```

Return Value

Returns a null-terminated string that contains the path of the current request.

Remarks

The path of the request is the part of the URL that appears after the server name, but before the query string. Equivalent to the **PATH_INFO** server variable or **EXTENSION_CONTROL_BLOCK::lpszPathInfo**.

Example

```
[ tag_name(name="GetPathInfo") ]
HTTP_CODE OnGetPathInfo()
{
    LPCSTR szPathInfo = m_spServerContext->GetPathInfo();
    if (szPathInfo)
    {
        m_HttpResponse << "Path info : " << szPathInfo;
    }
    else
    {
        m_HttpResponse << "Failed to get path info";
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IHttpServerContext::GetPathTranslated

Call this method to get the translated path of the requested resource from the HTTP server context object.

```
LPCSTR GetPathTranslated( );
```

Return Value

Returns a null-terminated string that contains the translated path of the requested resource.

Remarks

The translated path is the local file system path of the resource on the server. Equivalent to the **PATH_TRANSLATED** server variable or **EXTENSION_CONTROL_BLOCK::lpszPathTranslated**.

Example

```
[ tag_name(name="GetPathTranslated") ]
HTTP_CODE OnGetPathTranslated()
{
    LPCSTR szPathTranslated = m_spServerContext->GetPathTranslated();
    if (szPathTranslated)
    {
        m_HttpResponse << "Path translated : " << szPathTranslated;
    }
    else
    {
        m_HttpResponse << "Failed to get path translated";
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IHttpServerContext::GetQueryString

Call this method to get the query string from the HTTP server context object.

```
LPCSTR GetQueryString( );
```

Return Value

Returns a null-terminated string that contains the query information from the current request.

Remarks

The query string is the part of the URL that appears after the question mark (?). This method is equivalent to the **QUERY_STRING** server variable or **EXTENSION_CONTROL_BLOCK::lpszQueryString**.

Example

```
[ tag_name(name="GetQueryString") ]
HTTP_CODE OnGetQueryString()
{
    LPCSTR szQueryString = m_spServerContext->GetQueryString();
    if (szQueryString)
    {
        m_HttpResponse << "Query string : " << szQueryString;
    }
    else
    {
        m_HttpResponse << "Failed to get query string";
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IHttpServerContext::GetRequestMethod

Call this method to get the request method from the HTTP server context object.

```
LPCSTR GetRequestMethod( );
```

Return Value

Returns a null-terminated string that contains the HTTP method of the current request.

Remarks

Examples of common HTTP methods include "GET" and "POST". Equivalent to the **REQUEST_METHOD** server variable or **EXTENSION_CONTROL_BLOCK::lpszMethod**.

Example

```
[ tag_name(name="GetRequestMethod") ]
HTTP_CODE OnGetRequestMethod()
{
    LPCSTR szRequestMethod = m_spServerContext->GetRequestMethod();
    if (szRequestMethod)
    {
        m_HttpResponse << "Request method : " << szRequestMethod;
    }
    else
    {
        m_HttpResponse << "Failed to get request method";
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IHttpServerContext::GetScriptPathTranslated

Call this method to get the translated path of the script handling the current request from the HTTP server context object.

```
LPCSTR GetScriptPathTranslated( );
```

Return Value

Returns a null-terminated string containing the physical path of the script.

Example

```
[ tag_name(name="GetScriptPathTranslated") ]
HTTP_CODE OnGetScriptPathTranslated()
{
    // this replacement method will return the path info

    LPCSTR szScriptPathTranslated =
                        m_spServerContext->GetScriptPathTranslated();
    if (szScriptPathTranslated)
    {
        m_HttpResponse << "Script path translated : "
                        << szScriptPathTranslated;
    }
    else
    {
        m_HttpResponse << "Failed to get script path translated";
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowErrors

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IHttpServerContext::GetServerVariable

Call this method to get the value of a server variable from the HTTP server context object.

```
BOOL GetServerVariable(
   LPCSTR pszVariableName,
   LPSTR pvBuffer,
   DWORD* pdwSize
);
```

Parameters

pszVariableName

The name of the server variable.

pvBuffer

A caller-allocated buffer used to hold the value of the server variable on completion of the function.

pdwSize

On entry, the size of the buffer in bytes. On exit, the DWORD contains the number of bytes transferred or available to be transferred into the buffer (including the null-terminating byte).

Return Value

Returns **TRUE** on success, and **FALSE** on failure. Call GetLastError to get extended error information.

Remarks

This method is equivalent to EXTENSION_CONTROL_BLOCK::GetServerVariable.

Example

```
[ tag_name(name="GetServerVariable") ]
HTTP_CODE OnGetServerVariable(char* szServerVariable)
{
    // This replacement method will get the server
    // variable specified in the SRF file.

    BOOL bRet = TRUE;
    CHeapPtr<char> spBuf;

    char szBuf[256];
    char* pszBuf = szBuf;
    DWORD dwLen = sizeof(szBuf) - 1;

    bRet = m_spServerContext->GetServerVariable(
        szServerVariable,
        pszBuf,
        &dwLen);

    if (!bRet && dwLen >= sizeof(szBuf))
    {
        // If the buffer was too small, allocate a buffer of
        // adequate size.

        if (!spBuf.AllocateBytes(dwLen+1))
        {
            pszBuf = spBuf;

            // Try again to get the server variable.
            bRet = m_spServerContext->GetServerVariable(
                                        szServerVariable, pszBuf, &dwLen);
            if (!bRet)
            {
```

```
                // output message
                m_HttpResponse << "Failed to get server variable : "
                               << szServerVariable;
            }
        }
    }

    if (bRet)
    {
        pszBuf[dwLen] = '\0';
        m_HttpResponse << pszBuf;
    }

    if (bRet)
    {
        return HTTP_SUCCESS;
    }

    return AtlsHttpError(500, ISE_SUBERR_OUTOFMEM);
}
```

See Also

Reference

NotModified

ShowImage

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IHttpServerContext::GetTotalBytes

Call this method to get the total number of bytes in the current request.

```
DWORD GetTotalBytes( );
```

Return Value

Returns the total number of bytes to be received from the client.

Remarks

If this value is 0xffffffff, then there are four gigabytes or more of available data. In this case, if all the data is desired, IHttpServerContext::ReadClient or IHttpServerContext::AsyncReadClient should be called until no more data is returned.

This method is equivalent to the **CONTENT_LENGTH** server variable or **EXTENSION_CONTROL_BLOCK::cbTotalBytes**.

Example

See IHttpServerContext::GetAvailableBytes.

See Also

Reference

ForceLogin

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IHttpServerContext::MapUrlToPathEx

Call this method to get a physical file system path corresponding to a logical URL path.

```
BOOL MapUrlToPathEx(
   LPCSTR szLogicalPath,
   DWORD dwLen,
   HSE_URL_MAPEX_INFO* pumInfo
);
```

Parameters

szLogicalPath

The logical path.

dwLen

The number of bytes in *szLogicalPath*. Can be 0 if *szLogicalPath* is null-terminated.

pumInfo

Pointer to a structure that will contain the information about the physical attributes of *szLogicalPath*.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This method is equivalent to the **HSE_REQ_MAP_URL_TO_PATH_EX** server support function.

Example

```
[ tag_name(name="MapUrlToPathEx") ]
HTTP_CODE OnMapUrlToPathEx(char* szPath)
{
    // This replacement method will map a URL path
    // to a physical path.

    HSE_URL_MAPEX_INFO info;
    memset(&info, 0x00, sizeof(info));

    // Copy the path into the HSE_URL_MAPEX_INFO structure.
    if (SafeStringCopy(info.lpszPath, szPath))
    {
        // Set the permissions to read and execute.
        info.dwFlags = HSE_URL_FLAGS_READ | HSE_URL_FLAGS_EXECUTE;
        info.cchMatchingPath = strlen(szPath);
        info.cchMatchingURL = strlen("/mypath");
        if (!m_spServerContext->MapUrlToPathEx(szPath, 0, &info))
        {
            m_HttpResponse << "MapUrlToPathEx failed";
        }
    }
    else
    {
        m_HttpResponse << "path was too long";
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IHttpServerContext::ReadClient

Call this method to synchronously read information from the body of the Web client's HTTP request into the buffer supplied by the caller.

```
BOOL ReadClient(
   void* pvBuffer,
   DWORD* pdwSize
);
```

Parameters

pvBuffer

Points to the buffer allocated by the caller that will receive the requested information.

pdwSize

Points to a DWORD. On entry, the DWORD should be set to the number of bytes available in the buffer specified by *pvBuffer*. On exit, the DWORD will contain the number of bytes actually transferred into the buffer.

Return Value

Returns **TRUE** on success, **FALSE** on failure. Call GetLastError to get extended error information.

Remarks

This method is equivalent to **EXTENSION_CONTROL_BLOCK::ReadClient**.

Example

See IHttpServerContext::GetAvailableBytes.

See Also

Reference

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IHttpServerContext::RequestIOCompletion

Call this method to set a special callback function that will be used for handling the completion of asynchronous I/O operations.

```
BOOL RequestIOCompletion(
   PFN_HSE_IO_COMPLETION pfn,
   DWORD* pdwContext
);
```

Parameters

pfn

The pointer to the callback function.

pdwContext

A user-defined context that will be passed to the callback function on completion of an asynchronous I/O operation.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

PFN_HSE_IO_COMPLETION indicates the signature of the callback function.

This method is equivalent to the **HSE_REQ_IO_COMPLETION** server support function.

See Also

Reference

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IHttpServerContext::SendRedirectResponse

Call this method to redirect the client to the specified URL.

```
BOOL SendRedirectResponse(
   LPCSTR pszRedirectUrl
);
```

Parameters

pszRedirectUrl

The URL to which the client should be redirected.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

The client receives a 302 (Found) HTTP status code.

This method is equivalent to the **HSE_REQ_SEND_URL_REDIRECT_RESP** server support function.

Example

```
[ tag_name(name="SendRedirectResponse") ]
HTTP_CODE OnSendRedirectResponse(char* szUrl)
{
    // This replacement method will redirect to
    // the URL specified in the SRF file.

    if (m_spServerContext->SendRedirectResponse(szUrl))
    {
        // If the redirect succeeded, do no further processing.
        return HTTP_SUCCESS_NO_PROCESS;
    }

    m_HttpResponse << "failed to redirect";
    return HTTP_SUCCESS;
}
```

See Also

Reference

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IHttpServerContext::SendResponseHeader

Call this method to send an HTTP response header to the client.

```
BOOL SendResponseHeader(
   LPCSTR pszHeader,
   LPCSTR pszStatusCode,
   BOOL fKeepConn
);
```

Parameters

pszHeader

The header.

pszStatusCode

The HTTP status code.

fKeepConn

Indicates whether to keep the connection open.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This method is equivalent to the **HSE_REQ_SEND_RESPONSE_HEADER_EX** server support function.

Example

```
[ tag_name(name="SendResponseHeader") ]
HTTP_CODE OnSendResponseHeader(char* szHeader)
{
    // This replacement method will send the
    // response header specified in the SRF file.

    // Make sure we don't send the headers more than once.
    if (!m_HttpResponse.HaveSentHeaders())
    {
        if (m_spServerContext->SendResponseHeader(
                                        szHeader, "200 OK", FALSE))
        {
            // Make sure we don't send the headers more than once.
            m_HttpResponse.HaveSentHeaders(TRUE);
        }
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

ShowErrors

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IHttpServerContext::TransmitFile

Call this method to transmit a file asynchronously to the client.

```
BOOL TransmitFile(
   HANDLE hFile,
   PFN_HSE_IO_COMPLETION pfn,
   void* pContext,
   LPCSTR szStatusCode,
   DWORD dwBytesToWrite,
   DWORD dwOffset,
   void* pvHead,
   DWORD dwHeadLen,
   void* pvTail,
   DWORD dwTailLen,
   DWORD dwFlags
);
```

Parameters

hFile

Handle to the file to be transmitted. The file must be opened using the CreateFile function with the FILE_FLAG_SEQUENTIAL_SCAN and FILE_FLAG_OVERLAPPED flags turned on.

pfn

Points to the callback function that will be called on completion of the asynchronous transmission of the file. If NULL, the callback function must have been previously set by a call to IHttpServerContext::RequestIOCompletion.

pContext

Points to an application-defined context structure that will be passed as a parameter when the asynchronous I/O callback function is called.

szStatusCode

The HTTP status code.

dwBytesToWrite

The number of bytes to write. If this value is set to 0, the entire file will be sent.

dwOffset

The location in the file to start reading.

pvHead

A pointer to the headers which are to be sent before the file. This field is used only if HSE_IO_SEND_HEADERS is present in *dwFlags*. May be NULL.

dwHeadLen

The length in bytes of the headers pointed to by *pvHead*. This field is used only if HSE_IO_SEND_HEADERS is present in *dwFlags*.

pvTail

A pointer to data to be sent to the client after the file has been transmitted. May be NULL.

dwTailLen

The length in bytes of the data pointed to by *pvTail*.

dwFlags

The flags must include HSE_IO_ASYNC in all cases. If headers are to be sent as a result of the call to this function, you may also include HSE_IO_SEND_HEADERS. If the connection with the client is to be terminated once the data has been sent,

include HSE_IO_DISCONNECT_AFTER_SEND.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This method is equivalent to the **HSE_REQ_TRANSMIT_FILE** server support function.

See Also

Reference

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IHttpServerContext::WriteClient

Call this method to synchronously send the data present in the supplied buffer to the client that made the request.

```
BOOL WriteClient(
   void* pvBuffer,
   DWORD* pdwBytes
);
```

Parameters

pvBuffer

The buffer containing the data to be sent to the client.

pdwBytes

Pointer to a DWORD. On entry, the DWORD should be set to the number of bytes in the buffer. On exit, the DWORD will be set to the number of bytes successfully written to the client.

Return Value

Returns **TRUE** on success, **FALSE** on failure. Call GetLastError to get extended error information.

Remarks

This method is equivalent to EXTENSION_CONTROL_BLOCK::WriteClient(..., HSE_IO_SYNC).

Example

See the ShowErrors Sample and the MantaWeb Sample.

See Also

Reference

IHttpServerContext Interface

Other Resources

IHttpServerContext Members

IIsapiExtension Interface

This interface provides methods for communicating with the ISAPI extension DLL.

```
__interface __declspec(uuid("79DD4A27-D820-4fa6-954D-E1DFC2C05978"))
IIsapiExtension :
   public IUnknown
```

Remarks

This interface is implemented by CIsapiExtension.

Requirements

Header: atlsiface.h

See Also

Tasks

CLStencil Sample: Runs Request Handler DLLs or SRF Files from the Command Line
Hotswap Sample: Replacing ATL Server Application DLLs at Run Time
PerformancePersist Sample: Provides Custom Performance Monitoring Data

Reference

CIsapiExtension Class

Other Resources

IIsapiExtension Members
ATL Server Classes

IIsapiExtension Members

Methods

AddService	Call this method to add a dynamic service to the list of services exposed by the ISAPI extension.
CreateRequest	Call this method to create an ATL Server request object.
DispatchStencilCall	Call this method to handle the request and pass it on to any request handlers.
FreeRequest	Call this method to free an ATL Server request object.
GetThreadWorker	Call this method to get a pointer to the worker thread object for the current thread.
LoadDispatchFile	Call this method to populate the **AtlServerRequest** structure with the request handler interface pointer and DLL instance handle for the handler specified in a server response file.
LoadRequestHandler	Call this method to retrieve the request handler interface pointer and DLL instance handle for the specifie d handler.
OnThreadAttach	Override this method to execute code when a worker thread is initialized.
OnThreadTerminate	Override this method to execute code when a worker thread is about to be destroyed.
QueueRequest	Call this method to queue a request on the thread pool.
RemoveService	Call this method to remove a dynamic service from the list of services exposed by the ISAPI extension.
RequestComplete	Call this method when the handling of the request is complete.
SetThreadWorker	Call this method to set the worker thread object for the current thread.
TransferRequest	Call this method to transfer request processing to another request handler.

See Also

Reference

IIsapiExtension Interface

IIsapiExtension Methods

For information about the methods in **IIsapiExtension**, see IIsapiExtension Members.

IIsapiExtension::AddService

Call this method to add a dynamic service to the list of services exposed by the ISAPI extension.

```
HRESULT AddService(
   REFGUID guidService,
   REFIID riid,
   IUnknown* punk,
   HINSTANCE hInstance
);
```

Parameters

guidService

The GUID identifying the service to be added.

riid

The IID of an interface provided by the service.

punk

The interface pointer corresponding to *riid* of an object providing the service.

hInstance

The instance handle of the DLL providing the service. This instance handle will be used to keep the DLL loaded as long as the service is needed.

Return Value

Returns S_OK on success, S_FALSE if the service is already present, or an error HRESULT on failure.

Remarks

The extension object supporting the **IIsapiExtension** interface will usually support the IServiceProvider interface to allow request handlers to access arbitrary functionality. Some of the services implemented by ATL Server include browser capabilities, session state, and caching services.

The services exposed by **IServiceProvider** may be provided by the extension itself or by other DLLs and added to the list of services offered by the extension by means of **IIsapiExtension::AddService**. These dynamic services can be removed by calling IIsapiExtension::RemoveService.

Example

See the PerfPersist Sample.

See Also

Reference

IIsapiExtension Interface

IIsapiExtension::RemoveService

Other Resources

IIsapiExtension Members

IIsapiExtension::CreateRequest

Call this method to create an ATL Server request object.

```
AtlServerRequest* CreateRequest( );
```

Return Value

Returns a pointer to the newly created **AtlServerRequest** object.

Remarks

Call IIsapiExtension::FreeRequest to free the request.

See Also

Reference

IIsapiExtension Interface

IIsapiExtension::FreeRequest

Other Resources

IIsapiExtension Members

IIsapiExtension::DispatchStencilCall

Call this method to handle the request and pass it on to any request handlers.

```
BOOL DispatchStencilCall(
   AtlServerRequest* pRequestInfo
);
```

Parameters

pRequestInfo

Pointer to the object holding information about the request.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This method is called by CIsapiWorker::Execute.

Example

See the Hotswap Sample.

See Also

Reference

IIsapiExtension Interface

Other Resources

IIsapiExtension Members

IIsapiExtension::FreeRequest

Call this method to free an ATL Server request object.

```
void FreeRequest(
   AtlServerRequest* pRequest
);
```

Parameters

pRequest

The request to be freed.

Remarks

pRequest should have previously been created by a call to IIsapiExtension::CreateRequest.

See Also

Reference

IIsapiExtension Interface

IIsapiExtension::CreateRequest

Other Resources

IIsapiExtension Members

IIsapiExtension::GetThreadWorker

Call this method to get a pointer to the worker thread object for the current thread.

```
CIsapiWorker* GetThreadWorker( );
```

Return Value

Returns a pointer to the worker thread object for the current thread.

Remarks

The worker thread class returned by this method was previously set by a call to IIsapiExtension::SetThreadWorker.

Example

```
[ tag_name(name="WriteUserData") ]
HTTP_CODE OnWriteUserData(void)
{
    // This method demonstrates how to allocate and free memory using
    // the worker thread's heap.

    CIsapiWorker* pWorker = m_spExtension->GetThreadWorker();

    DWORD dwLen = m_spServerContext->GetTotalBytes();
    char* szData = reinterpret_cast<char*>(
                                HeapAlloc(pWorker->m_hHeap, 0, dwLen));
    if (!szData)
    {
        return AtlsHttpError(500, ISE_SUBERR_OUTOFMEM);
    }

    if (!ReadClientData(m_spServerContext, szData, &dwLen, 0))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.WriteLen(szData, dwLen))
    {
        return AtlsHttpError(500, ISE_SUBERR_OUTOFMEM);
    }

    // Free the allocated data.
    if (szData)
    {
        BOOL bRet = HeapFree(pWorker->m_hHeap, 0, szData);
        if (!bRet)
        {
            return HTTP_FAIL;
        }
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IIsapiExtension Interface

IIsapiExtension::SetThreadWorker

Other Resources

IIsapiExtension Members

IIsapiExtension::LoadDispatchFile

Call this method to populate the **AtlServerRequest** structure with the request handler interface pointer and DLL instance handle for the handler specified in a server response file.

```
HTTP_CODE LoadDispatchFile(
   LPCSTR szFileName,
   AtlServerRequest* pRequestInfo
);
```

Parameters

szFileName

The name of a server response file.

pRequestInfo

The structure holding information about the request.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

Call this method to load a server response file, parse it for information about the request handler, load the request handler, and populate the **AtlServerRequest** structure with the request handler interface pointer and DLL instance handle.

See Also

Reference

IIsapiExtension Interface

IIsapiExtension::LoadRequestHandler

Other Resources

IIsapiExtension Members

IIsapiExtension::LoadRequestHandler

Call this method to retrieve the request handler interface pointer and DLL instance handle for the specified handler.

```
HTTP_CODE LoadRequestHandler(
   LPCSTR szDllPath,
   LPCSTR szHandlerName,
   IHttpServerContext* pServerContext,
   HINSTANCE* phInstance,
   IRequestHandler** ppHandler
);
```

Parameters

szDllPath

The path of a request handler.

szHandlerName

The name of a request handler.

pServerContext

The server context.

phInstance

Address of an HINSTANCE variable to hold the instance handle of the request handler DLL on successful return of this method.

ppHandler

Address of an **IRequestHandler**** variable to hold the request handler's interface pointer on successful return of this method.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

IIsapiExtension Interface

IIsapiExtension::LoadDispatchFile

Other Resources

IIsapiExtension Members

IIsapiExtension::OnThreadAttach

Override this method to execute code when a worker thread is initialized.

```
BOOL OnThreadAttach( );
```

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This method is the complement to IIsapiExtension::OnThreadTerminate.

See Also

Reference

IIsapiExtension Interface

IIsapiExtension::OnThreadTerminate

Other Resources

IIsapiExtension Members

IIsapiExtension::OnThreadTerminate

Override this method to execute code when a worker thread is about to be destroyed.

```
void OnThreadTerminate( );
```

Remarks

This method is the complement to IIsapiExtension::OnThreadAttach.

See Also

Reference

IIsapiExtension Interface

IIsapiExtension::OnThreadAttach

Other Resources

IIsapiExtension Members

IIsapiExtension::QueueRequest

Call this method to queue a request on the thread pool.

```
BOOL QueueRequest(
   AtlServerRequest* pRequestInfo
);
```

Parameters

pRequestInfo

The structure holding information about the request.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This method returns immediately. The return value indicates only whether the request was successfully queued; it does not indicate whether the request was processed or if the processing was successful.

See Also

Reference

IIsapiExtension Interface

Other Resources

IIsapiExtension Members

IIsapiExtension::RemoveService

Call this method to remove a dynamic service from the list of services exposed by the ISAPI extension.

```
HRESULT RemoveService(
   REFGUID guidService,
   REFIID riid
);
```

Parameters

guidService

The GUID identifying the service to be removed.

riid

The IID of the interface provided by the service to be removed.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The extension object supporting the **IIsapiExtension** interface will usually support the IServiceProvider interface to allow request handlers to access arbitrary functionality. Some of the services implemented by ATL Server include browser capabilities, session state, and caching services.

The services exposed by **IServiceProvider** may be provided by the extension itself or by other DLLs and added to the list of services offered by the extension by means of IIsapiExtension::AddService. These dynamic services can be removed by calling **IIsapiExtension::RemoveService**.

See Also

Reference

IIsapiExtension Interface

IIsapiExtension::AddService

Other Resources

IIsapiExtension Members

IIsapiExtension::RequestComplete

Call this method when the handling of the request is complete.

```
void RequestComplete(
   AtlServerRequest* pRequestInfo,
   DWORD hStatus,
   DWORD dwSubStatus
);
```

Parameters

pRequestInfo

The structure holding information about the current request.

hStatus

The ATL Server status code.

dwSubStatus

The ATL Server subcode.

See Also

Reference

IIsapiExtension Interface

Other Resources

IIsapiExtension Members

IIsapiExtension::SetThreadWorker

Call this method to set the worker thread object for the current thread.

```
BOOL SetThreadWorker(
   CIsapiWorker* pWorker
);
```

Parameters

pWorker

The worker thread object for the current thread.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

The worker thread class set by this method can be subsequently obtained by a call to IIsapiExtension::GetThreadWorker.

See Also

Reference

IIsapiExtension Interface

IIsapiExtension::GetThreadWorker

Other Resources

IIsapiExtension Members

IIsapiExtension::TransferRequest

Call this method to transfer request processing to another request handler.

```
HTTP_CODE TransferRequest(
   AtlServerRequest * pRequest,
   IServiceProvider * pServiceProvider,
   IWriteStream * pWriteStream,
   IHttpRequestLookup * pLookup,
   LPCSTR szNewUrl,
   WORD nCodePage,
   bool bContinueAfterProcess,
   CStencilState * pState
);
```

Parameters

pRequest

The request currently being handled.

pServiceProvider

The service provider.

pWriteStream

The response stream.

pLookup

The object providing access to the parent handler's form variables and query parameters.

szNewUrl

The URL of the new request handler.

nCodePage

The code page.

bContinueAfterProcess

True if processing should continue in the current handler after the new handler has finished, false otherwise.

pState

A pointer to state information.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

This method is implemented in the same way as the include tag. **TransferRequest** transfers the processing of the current request to another request handler. This new request handler is initialized as a child of the current request handler and has access to the parent's request, including form variable, cookies, query string, and other data, just like an included handler.

This method is called from CRequestHandlerT::ServerTransferRequest.

Example

```
[ tag_name(name="TransferRequest") ]
HTTP_CODE OnTransferRequest(char* szTransfer)
{
    HRESULT hr = m_spExtension->TransferRequest(
        m_pRequestInfo,
        m_spServiceProvider,
        &m_HttpResponse,
```

```
        &m_HttpRequest,
        szTransfer,
        m_nCodePage,
        true,
        &m_state
        );

    if (SUCCEEDED(hr))
    {
        return HTTP_SUCCESS;
    }

    return HTTP_FAIL;
}
```

See Also

Reference

IIsapiExtension Interface

Other Resources

IIsapiExtension Members

IMemoryCache Interface

This interface provides methods for adding, retrieving, and removing items cached in memory.

```
__interface ATL_NO_VTABLE
__declspec(uuid("9c6cfb46-fbde-4f8b-b944-2aa05d96eb5c"))
IMemoryCache :
   public IUnknown
```

Remarks

This interface is implemented by CBlobCache.

Requirements

Header: atlcache.h

Example

See the BlobCache Sample.

See Also

Other Resources

Caching
IMemoryCache Members
ATL Server Classes
Caching Reference

IMemoryCache Members

Methods

Add	Call this method to add an entry to the cache.
Flush	Call this method to flush eligible entries from the cache.
GetData	Call this method to retrieve a cached item.
LookupEntry	Call this method to get the handle of an item in the cache.
ReleaseEntry	Call this method to decrement the reference count of a cached item.
RemoveEntry	Call this method to remove an item from the cache given its handle.
RemoveEntryByKey	Call this method to remove an item from the cache given its key.

See Also

Reference

IMemoryCache Interface

Other Resources

Caching

IMemoryCache Methods

For information about the methods in **IMemoryCache**, see IMemoryCache Members.

IMemoryCache::Add

Call this method to add an entry to the cache.

```
STDMETHOD(Add)(
   LPCSTR szKey,
   void * pvData,
   DWORD dwSize,
   FILETIME * pftExpireTime,
   HINSTANCE hInstClient,
   HCACHEITEM * phEntry,
   IMemoryCacheClient * pClient
);
```

Parameters

szKey

A string containing a key that can be used later to retrieve the new item.

pvData

The address of the data to add to the cache.

dwSize

The size of the data to add to the cache.

pftExpireTime

The expiration time after which the entry is to be removed from the cache. A **FILETIME** of zero indicates that the item will not expire.

hInstClient

The instance handle of a DLL upon which the cached item depends. The reference count of this DLL is incremented when the item is added to the cache and decremented when the item is removed. Use **NULL** if there is no dependency.

phEntry

[out] The optional address of a cache item handle that can later be used to access the new cache entry. If no handle is retrieved, the item can still be accessed later using the LookupEntry method.

pClient

The optional address of an implementation of IMemoryCacheClient that will be asked to free the cached item when it is removed from the cache. See IMemoryCacheClient::Free.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Adds an entry to the cache.

If *phEntry* is not NULL, the initial reference count for the item is set to 1. See IMemoryCache::ReleaseEntry and IMemoryCache::RemoveEntry for more details on reference counting.

Example

```
HTTP_CODE SubmitMessage()
{
    CComPtr<IMemoryCache> spMemoryCache;
    CComPtr<IMemoryCacheClient> spMemoryCacheClient;

    if (FAILED(m_spServiceProvider->QueryService(
                                __uuidof(IMemoryCache), &spMemoryCache)))
    {
```

```
        return HTTP_FAIL;
    }

    if (FAILED(m_spServiceProvider->QueryService(
                    __uuidof(IMemoryCacheClient), &spMemoryCacheClient)))
    {
        return HTTP_FAIL;
    }

    CHAR* psz = static_cast<CHAR*>(::HeapAlloc(::GetProcessHeap(),
                                        0, m_strMessage.GetLength()));
    if (psz == NULL)
    {
        return HTTP_FAIL;
    }
    memcpy(psz, m_strMessage.GetString(), m_strMessage.GetLength());

    HCACHEITEM hItem;
    CFileTime ft = CFileTime::GetCurrentTime() +
                   CFileTimeSpan(m_nKeepSeconds * CFileTime::Second);
    if (FAILED(spMemoryCache->Add(m_strMessageName, psz,
                    m_strMessage.GetLength(), &ft, NULL,
                          &hItem, spMemoryCacheClient)))
    {
        ::HeapFree(::GetProcessHeap(), 0, psz);
        return HTTP_FAIL;
    }

    spMemoryCache->ReleaseEntry(hItem);

    return HTTP_SUCCESS;
}
```

See Also

Tasks

BlobCache Sample: Generic Memory Caching

Reference

IMemoryCache Interface

Other Resources

Caching

IMemoryCache Members

IMemoryCache::Flush

Call this method to flush eligible entries from the cache.

```
STDMETHOD(Flush)();
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The definition of an item that is eligible for removal from the cache will depend on the implementation. However, all implementations should use reference counting to ensure that items currently in use are not removed from the cache unless necessary. Items that have expired are removed when **Flush** is called.

Example

```
HTTP_CODE FlushCache()
{
    CComPtr<IMemoryCache> spMemoryCache;

    if (FAILED(m_spServiceProvider->QueryService(
                                __uuidof(IMemoryCache), &spMemoryCache)))
    {
        return HTTP_FAIL;
    }

    if (FAILED(spMemoryCache->Flush()))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Tasks

BlobCache Sample: Generic Memory Caching

Reference

IMemoryCache Interface

Other Resources

Caching

IMemoryCache Members

IMemoryCache::GetData

Call this method to retrieve a cached item.

```
STDMETHOD(GetData)(
   const HCACHEITEM hEntry,
   void ** ppvData,
   DWORD * pdwSize
) const;
```

Parameters

hEntry

The handle of the item to be retrieved as obtained by a previous call to Add or LookupEntry.

ppvData

The address of the variable to receive the cached item.

pdwSize

The address of a DWORD that is to receive the size of the retrieved item.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

```
HTTP_CODE RetrieveMessage()
{
    m_bFound = false;

    CComPtr<IMemoryCache> spMemoryCache;

    if (FAILED(m_spServiceProvider->QueryService(
                            __uuidof(IMemoryCache), &spMemoryCache)))
    {
        return HTTP_FAIL;
    }

    HCACHEITEM hItem;
    if (FAILED(spMemoryCache->LookupEntry(m_strMessageName, &hItem)))
    {
        return HTTP_SUCCESS;
    }

    void* pData = NULL;
    DWORD dwSize;
    HRESULT hr = spMemoryCache->GetData(hItem, &pData, &dwSize);
    if (FAILED(hr) || pData == NULL)
    {
        spMemoryCache->ReleaseEntry(hItem);
        return HTTP_FAIL;
    }

    m_strMessage = CStringA(static_cast<LPCSTR>(pData), dwSize);

    spMemoryCache->ReleaseEntry(hItem);
    m_bFound = true;

    return HTTP_SUCCESS;
}
```

See Also

Tasks

BlobCache Sample: Generic Memory Caching

Reference

IMemoryCache Interface

IMemoryCache::LookupEntry

Other Resources

Caching

IMemoryCache Members

IMemoryCache::LookupEntry

Call this method to get the handle of an item in the cache.

```
STDMETHOD(LookupEntry)(
   LPCSTR szKey,
   HCACHEITEM * phEntry
);
```

Parameters

szKey

The key of the item whose handle should be returned in *phEntry*.

phEntry

[out] Address of the **HCACHEITEM** variable that, on success, receives the handle of the requested item.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure. Returns **E_FAIL** if the item is not in the cache.

Remarks

This method returns a handle to the requested item and increments the reference count on that item. The handle and reference count should be returned to the cache by a subsequent call to ReleaseEntry when the item is no longer needed.

Example

```
HTTP_CODE RetrieveMessage()
{
    m_bFound = false;

    CComPtr<IMemoryCache> spMemoryCache;

    if (FAILED(m_spServiceProvider->QueryService(
                                __uuidof(IMemoryCache), &spMemoryCache)))
    {
        return HTTP_FAIL;
    }

    HCACHEITEM hItem;
    if (FAILED(spMemoryCache->LookupEntry(m_strMessageName, &hItem)))
    {
        return HTTP_SUCCESS;
    }

    void* pData = NULL;
    DWORD dwSize;
    HRESULT hr = spMemoryCache->GetData(hItem, &pData, &dwSize);
    if (FAILED(hr) || pData == NULL)
    {
        spMemoryCache->ReleaseEntry(hItem);
        return HTTP_FAIL;
    }

    m_strMessage = CStringA(static_cast<LPCSTR>(pData), dwSize);

    spMemoryCache->ReleaseEntry(hItem);
    m_bFound = true;

    return HTTP_SUCCESS;
}
```

See Also

Tasks

BlobCache Sample: Generic Memory Caching

Reference

IMemoryCache Interface

IMemoryCache::GetData

Other Resources

Caching

IMemoryCache Members

IMemoryCache::ReleaseEntry

Call this method to decrement the reference count of a cached item.

```
STDMETHOD(ReleaseEntry)(
   const HCACHEITEM hEntry
);
```

Parameters

hEntry

The handle of the cache item whose reference count is to be decremented.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Every successful call to Add or LookupEntry that gets the handle of an item in the cache should have a matching call to this method (or RemoveEntry) when the handle is no longer being used so that the cache has an opportunity to remove the corresponding item. Mismatched calls to these methods will result in leaks or premature removal of cached items.

Example

```
HTTP_CODE RetrieveMessage()
{
    m_bFound = false;

    CComPtr<IMemoryCache> spMemoryCache;

    if (FAILED(m_spServiceProvider->QueryService(
                            __uuidof(IMemoryCache), &spMemoryCache)))
    {
        return HTTP_FAIL;
    }

    HCACHEITEM hItem;
    if (FAILED(spMemoryCache->LookupEntry(m_strMessageName, &hItem)))
    {
        return HTTP_SUCCESS;
    }

    void* pData = NULL;
    DWORD dwSize;
    HRESULT hr = spMemoryCache->GetData(hItem, &pData, &dwSize);
    if (FAILED(hr) || pData == NULL)
    {
        spMemoryCache->ReleaseEntry(hItem);
        return HTTP_FAIL;
    }

    m_strMessage = CStringA(static_cast<LPCSTR>(pData), dwSize);

    spMemoryCache->ReleaseEntry(hItem);
    m_bFound = true;

    return HTTP_SUCCESS;
}
```

See Also

Tasks

BlobCache Sample: Generic Memory Caching

Reference

IMemoryCache Interface

Other Resources

Caching

IMemoryCache Members

IMemoryCache::RemoveEntry

Call this method to remove an item from the cache given its handle.

```
STDMETHOD(RemoveEntry)(
   const HCACHEITEM hEntry
);
```

Parameters

hEntry

The handle of the item to remove from the cache.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Calling this method will decrement the reference count of the item and remove it from the cache immediately. Subsequent calls to LookupEntry will fail to retrieve an item removed in this way. However, the item itself may not be cleaned up immediately. Cached items are only cleaned up when their reference count reaches zero to ensure that multiple clients can use the same item without conflicts. The reference count may reach zero as a result of a call to this method or ReleaseEntry.

See Also

Reference

IMemoryCache Interface

IMemoryCache::RemoveEntryByKey

Other Resources

Caching

IMemoryCache Members

IMemoryCache::RemoveEntryByKey

Call this method to remove an item from the cache given its key.

```
STDMETHOD(RemoveEntryByKey)(
   LPCSTR szKey
);
```

Parameters

szKey

The key of the item to remove from the cache.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

See RemoveEntry for details.

Example

```
HTTP_CODE RemoveMessage()
{
    CComPtr<IMemoryCache> spMemoryCache;

    if (FAILED(m_spServiceProvider->QueryService(
                            __uuidof(IMemoryCache), &spMemoryCache)))
    {
        return HTTP_FAIL;
    }

    if (FAILED(spMemoryCache->RemoveEntryByKey(m_strMessageName)))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IMemoryCache Interface

IMemoryCache::RemoveEntry

Other Resources

Caching

IMemoryCache Members

IMemoryCacheClient Interface

This interface provides a method that can be called to free an item that is being removed from a cache.

```
__interface ATL_NO_VTABLE
__declspec(uuid("b721b49d-bb57-47bc-ac43-a8d4c07d183d"))
IMemoryCacheClient :
   public IUnknown
```

Remarks

This interface is used by CMemoryCache, CStencilCache, and derived classes to allow clients to be notified when an item is finally removed from the cache. These classes store a pointer to an **IMemoryCacheClient** implementation along with the item data so that they can call Free when the cached item is no longer needed. Implementations of **IMemoryCacheClient** will typically free memory and other resources associated with the cached item at that time.

This interface is implemented by CStencil and CSharedCache.

Requirements

Header: atlcache.h

Example

```
class CMemoryCacheClient :
    public IMemoryCacheClient,
    public CComObjectRootEx<CComGlobalsThreadModel>
{
    BEGIN_COM_MAP(CMemoryCacheClient)
        COM_INTERFACE_ENTRY(IMemoryCacheClient)
    END_COM_MAP()

    STDMETHOD(Free)(const void *pvData)
    {
        void** p = (void**) pvData;
        ::HeapFree(::GetProcessHeap(), 0, p);
        return S_OK;
    }
};
```

See Also

Tasks

BlobCache Sample: Generic Memory Caching

Other Resources

IMemoryCacheClient Members

ATL Server Classes

Caching Reference

Caching

IMemoryCacheClient Members

Methods

Free	This method is called when an item is being removed from the cache.

See Also

Reference

IMemoryCacheClient Interface

IMemoryCacheClient Methods

For information about the methods in **IMemoryCacheClient**, see IMemoryCacheClient Members

IMemoryCacheClient::Free

This method is called when an item is being removed from the cache.

```
STDMETHOD(Free)(
   const void * pvData
);
```

Parameters

pvData

The address of the data being removed from the cache.

Return Value

Return S_OK on success, or an error HRESULT on failure.

Remarks

The implementation of this method will depend on the type of data being deleted from the cache.

Example

See IMemoryCacheClient Overview.

See Also

Tasks

BlobCache Sample: Generic Memory Caching

Reference

IMemoryCacheClient Interface

Other Resources

IMemoryCacheClient Members

IMemoryCacheControl Interface

This interface provides methods for controlling the size of a cache.

```
__interface ATL_NO_VTABLE
__declspec(uuid("7634b28b-d819-409d-b96e-fc9f3aba329f"))
IMemoryCacheControl :
   public IUnknown
```

Remarks

IMemoryCacheControl provides methods for setting and retrieving constraints on the size and the number of entries in the cache using SetMaxAllowedSize, SetMaxAllowedEntries, GetMaxAllowedSize, and GetMaxAllowedEntries. It also allows all of the content and statistics associated with the cache to be removed in a single call to ResetCache.

This interface is implemented by CBlobCache and CFileCache.

Requirements

Header: atlcache.h

Example

See the BlobCache Sample.

See Also

Other Resources

IMemoryCacheControl Members
ATL Server Classes
Caching Reference
Caching

IMemoryCacheControl Members

Methods

GetMaxAllowedEntries	Call this method to retrieve the maximum number of entries currently allowed in the cache.
GetMaxAllowedSize	Call this method to retrieve the maximum size of the entries currently allowed in the cache.
ResetCache	Call this method to remove the contents of the cache and reset the statistics to zero.
SetMaxAllowedEntries	Call this method to set the maximum number of entries allowed in the cache.
SetMaxAllowedSize	Call this method to set the maximum size of the entries allowed in the cache.

See Also

Reference

IMemoryCacheControl Interface

IMemoryCacheControl Methods

For information about the methods in **IMemoryCacheControl**, see IMemoryCacheControl Members.

IMemoryCacheControl::GetMaxAllowedEntries

Call this method to retrieve the maximum number of entries currently allowed in the cache.

```
STDMETHOD(GetMaxAllowedEntries)(
   DWORD * pdwSize
);
```

Parameters

pdwSize

[out] Address of the DWORD variable that, on success, receives the maximum number of entries currently allowed in the cache.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

See SetMaxAllowedEntries for details.

Example

```
[ tag_name(name = "GetMaxAllowedEntries") ]
HTTP_CODE OnGetMaxAllowedEntries(void)
{
    DWORD dw;

    if (FAILED(m_spControl->GetMaxAllowedEntries(&dw)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(dw))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Tasks

BlobCache Sample: Generic Memory Caching

Reference

IMemoryCacheControl Interface

IMemoryCacheControl::SetMaxAllowedEntries

Other Resources

IMemoryCacheControl Members

IMemoryCacheControl::GetMaxAllowedSize

Call this method to retrieve the maximum size of the entries currently allowed in the cache.

```
STDMETHOD(GetMaxAllowedSize)(
   DWORD * pdwSize
);
```

Parameters

pdwSize

[out] Address of the DWORD variable that, on success, receives the maximum size of the entries currently allowed in the cache.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

See SetMaxAllowedSize for details.

Example

```
[ tag_name(name = "GetMaxAllowedSize") ]
HTTP_CODE OnGetMaxAllowedSize(void)
{
    DWORD dw;

    if (FAILED(m_spControl->GetMaxAllowedSize(&dw)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(dw))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IMemoryCacheControl Interface

IMemoryCacheControl::SetMaxAllowedSize

Other Resources

IMemoryCacheControl Members

IMemoryCacheControl::ResetCache

Call this method to remove the contents of the cache and reset the statistics to zero.

```
STDMETHOD(ResetCache)();
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

A cache may collect statistics related to its use including such information as effectiveness of the cache and memory used. Calling this method removes all the items from the cache and clears the historical statistics for the cache.

Example

```
HTTP_CODE ResetCache()
{
    if (FAILED(m_spControl->ResetCache()))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Tasks

BlobCache Sample: Generic Memory Caching

Reference

IMemoryCacheControl Interface

Other Resources

IMemoryCacheControl Members

IMemoryCacheControl::SetMaxAllowedEntries

Call this method to set the maximum number of entries allowed in the cache.

```
STDMETHOD(SetMaxAllowedEntries)(
   DWORD dwSize
);
```

Parameters

dwSize

The maximum number of entries allowed in the cache.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

How the cash enforces this limit is implementation specific. The cache may choose to enforce this limit by preventing the addition of items that would cause the limit to be exceeded, by automatically removing existing cache items to allow new items to be added, or by allowing the limit to be exceeded temporarily and periodically cleaning the cache to bring it back within the bounds set by this method.

Example

```
HTTP_CODE SetMaxEntries(DWORD dwMaxEntries)
{
    if (FAILED(m_spControl->SetMaxAllowedEntries(dwMaxEntries)))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Tasks

BlobCache Sample: Generic Memory Caching

Reference

IMemoryCacheControl Interface

IMemoryCacheControl::GetMaxAllowedEntries

Other Resources

IMemoryCacheControl Members

IMemoryCacheControl::SetMaxAllowedSize

Call this method to set the maximum size of the entries allowed in the cache.

```
STDMETHOD(SetMaxAllowedSize)(
   DWORD dwSize
);
```

Parameters

dwSize

The maximum size of the entries allowed in the cache.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

How the cash enforces this limit is implementation specific. The cache may choose to enforce this limit by preventing the addition of items that would cause the limit to be exceeded, by automatically removing existing cache items to allow new items to be added, or by allowing the limit to be exceeded temporarily and periodically cleaning the cache to bring it back within the bounds set by this method.

Example

```
HTTP_CODE SetMaxSize(DWORD dwMaxSize)
{
    if (FAILED(m_spControl->SetMaxAllowedSize(dwMaxSize)))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IMemoryCacheControl Interface

IMemoryCacheControl::GetMaxAllowedSize

Other Resources

IMemoryCacheControl Members

IMemoryCacheStats Interface

This interface provides methods for retrieving current and historical information about the state of a memory-based cache.

```
__interface ATL_NO_VTABLE
__declspec(uuid("d4b6df2d-4bc0-4734-8ace-b73a0b975956"))
IMemoryCacheStats :
   public IUnknown
```

Remarks

This interface provides information about the current and maximum number of items in the cache and the frequency with which items requested from the cache are actually found there. It also allows the historical information to be cleared.

This interface is implemented by CBlobCache, CStencilCache, and CFileCache.

Requirements

Header: atlcache.h

See Also

Tasks

BlobCache Sample: Generic Memory Caching
CLStencil Sample: Runs Request Handler DLLs or SRF Files from the Command Line
StencilCache Sample: Uses the ATL Server Stencil Cache

Other Resources

Caching
IMemoryCacheStats Members
ATL Server Classes
Caching Reference

IMemoryCacheStats Members

Methods

ClearStats	Call this method to reset all the statistics to zero.
GetCurrentAllocSize	Call this method to get the current number of bytes used by the cache for storing data.
GetCurrentEntryCount	Call this method to get the current number of items in the cache.
GetHitCount	Call this method to get the number of times the cache successfully returned an item.
GetMaxAllocSize	Call this method to get the maximum number of bytes used by the cache at one time since statistics gathering was started.
GetMaxEntryCount	Call this method to get the maximum number of items in the cache at one time since statistics gathering was started.
GetMissCount	Call this method to get the number of times the cache did not contain a requested item.

See Also

Reference

IMemoryCacheStats Interface

Other Resources

Caching

IMemoryCacheStats Methods

For information about the methods in **IMemoryCacheStats**, see IMemoryCacheStats Members.

IMemoryCacheStats::ClearStats

Call this method to reset all the statistics to zero.

```
STDMETHOD(ClearStats)();
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The object implementing this interface keeps a running total of the statistics rather than querying the cache for the current information. Calling this method clears *all* the statistics even those with **Current** in their name.

Example

```
HTTP_CODE ClearStats()
{
    if (FAILED(m_spStats->ClearStats()))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Tasks

BlobCache Sample: Generic Memory Caching

Reference

IMemoryCacheStats Interface

Other Resources

Caching

IMemoryCacheStats Members

IMemoryCacheStats::GetCurrentAllocSize

Call this method to get the current number of bytes used by the cache for storing data.

```
STDMETHOD(GetCurrentAllocSize)(
   DWORD * pdwSize
);
```

Parameters

pdwSize

[out] Address of the variable that, on success, receives the current number of bytes used by the cache for storing data.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

```
[ tag_name(name = "GetCurrentAllocSize") ]
HTTP_CODE OnGetCurrentAllocSize(void)
{
    DWORD dw;

    if (FAILED(m_spStats->GetCurrentAllocSize(&dw)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(dw))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Tasks

CLStencil Sample: Runs Request Handler DLLs or SRF Files from the Command Line

Reference

IMemoryCacheStats Interface

IMemoryCacheStats::GetMaxAllocSize

Other Resources

Caching

IMemoryCacheStats Members

IMemoryCacheStats::GetCurrentEntryCount

Call this method to get the current number of items in the cache.

```
STDMETHOD(GetCurrentEntryCount)(
   DWORD * pdwSize
);
```

Parameters

pdwSize

[out] Address of the variable that, on success, receives the current number of items in the cache.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

```
[ tag_name(name = "GetCurrentEntryCount") ]
HTTP_CODE OnGetCurrentEntryCount(void)
{
    DWORD dw;

    if (FAILED(m_spStats->GetCurrentEntryCount(&dw)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(dw))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Tasks

BlobCache Sample: Generic Memory Caching

CLStencil Sample: Runs Request Handler DLLs or SRF Files from the Command Line

Reference

IMemoryCacheStats Interface

IMemoryCacheStats::GetMaxEntryCount

Other Resources

Caching

IMemoryCacheStats Members

IMemoryCacheStats::GetHitCount

Call this method to get the number of times the cache successfully returned an item.

```
STDMETHOD(GetHitCount)(
   DWORD * pdwSize
);
```

Parameters

pdwSize

[out] Address of the variable that, on success, receives the number of times the cache successfully returned an item.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

```
[ tag_name(name = "GetHitCount") ]
HTTP_CODE OnGetHitCount(void)
{
    DWORD dw;

    if (FAILED(m_spStats->GetHitCount(&dw)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(dw))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Tasks

BlobCache Sample: Generic Memory Caching

CLStencil Sample: Runs Request Handler DLLs or SRF Files from the Command Line

Reference

IMemoryCacheStats Interface

IMemoryCacheStats::GetMissCount

Other Resources

Caching

IMemoryCacheStats Members

IMemoryCacheStats::GetMaxAllocSize

Call this method to get the maximum number of bytes used by the cache at one time since statistics gathering was started.

```
STDMETHOD(GetMaxAllocSize)(
   DWORD * pdwSize
);
```

Parameters

pdwSize

[out] Address of the variable that, on success, receives the maximum number of bytes allocated by the cache.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

```
[ tag_name(name = "GetMaxAllocSize") ]
HTTP_CODE OnGetMaxAllocSize(void)
{
    DWORD dw;

    if (FAILED(m_spStats->GetMaxAllocSize(&dw)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(dw))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Tasks

CLStencil Sample: Runs Request Handler DLLs or SRF Files from the Command Line

Reference

IMemoryCacheStats Interface

IMemoryCacheStats::GetCurrentAllocSize

Other Resources

Caching

IMemoryCacheStats Members

IMemoryCacheStats::GetMaxEntryCount

Call this method to get the maximum number of items in the cache at one time since statistics gathering was started.

```
STDMETHOD(GetMaxEntryCount)(
   DWORD * pdwSize
);
```

Parameters

pdwSize

[out] Address of the variable that, on success, receives the maximum number of items in the cache.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

```
[ tag_name(name = "GetMaxEntryCount") ]
HTTP_CODE OnGetMaxEntryCount(void)
{
    DWORD dw;

    if (FAILED(m_spStats->GetMaxEntryCount(&dw)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(dw))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Tasks

BlobCache Sample: Generic Memory Caching

CLStencil Sample: Runs Request Handler DLLs or SRF Files from the Command Line

Reference

IMemoryCacheStats Interface

IMemoryCacheStats::GetCurrentEntryCount

Other Resources

Caching

IMemoryCacheStats Members

IMemoryCacheStats::GetMissCount

Call this method to get the number of times the cache did not contain a requested item.

```
STDMETHOD(GetMissCount)(
   DWORD * pdwSize
);
```

Parameters

pdwSize

[out] Address of the variable that, on success, receives the number of times the cache has been queried for an item it did not contain.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

```
[ tag_name(name = "GetMissCount") ]
HTTP_CODE OnGetMissCount(void)
{
    DWORD dw;

    if (FAILED(m_spStats->GetMissCount(&dw)))
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(dw))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Tasks

BlobCache Sample: Generic Memory Caching

CLStencil Sample: Runs Request Handler DLLs or SRF Files from the Command Line

Reference

IMemoryCacheStats Interface

IMemoryCacheStats::GetHitCount

Other Resources

Caching

IMemoryCacheStats Members

IPageCacheControl Interface

This interface provides methods for controlling the caching of a response.

```
__interface ATL_NO_VTABLE
__declspec( uuid( "9868BFC0-D44D-4154-931C-D186EC0C45D5" ))
IPageCacheControl :
   public IUnknown
```

Remarks

This interface is implemented by CCacheServerContext so it is available to request handlers that include ATLSRV_INIT_USECACHE in the flags returned from IRequestHandler::GetFlags. See Enabling Page Caching for more details.

Requirements

Header: atlsiface.h

See Also

Concepts

Enabling Page Caching

Other Resources

IPageCacheControl Members

Caching Reference

Caching

IPageCacheControl Members

Methods

Cache	Call this method to specify whether or not the current response should be cached.
GetExpiration	Call this method to discover the time at which the current response will be removed from the cache.
IsCached	Call this method to determine whether the current response will be cached.
SetExpiration	Call this method to specify the time at which the current response should be removed from the cache.

See Also

Reference

IPageCacheControl Interface

IPageCacheControl Methods

For information about the methods in **IPageCacheControl**, see IPageCacheControl Members

IPageCacheControl::Cache

Call this method to specify whether or not the current response should be cached.

```
BOOL Cache(
   BOOL bCache
);
```

Parameters

bCache

Specify TRUE to allow the current response to be cached. Specify FALSE to prevent the current response from being cached.

Return Value

Returns the previous value indicating whether or not the current response would have been cached.

See Also

Reference

IPageCacheControl Interface

IPageCacheControl::IsCached

Other Resources

IPageCacheControl Members

IPageCacheControl::GetExpiration

Call this method to discover the time at which the current response will be removed from the cache.

```
HRESULT GetExpiration(
   FILETIME * pftExpiration
);
```

Parameters

pftExpiration

[out] Address of the variable that, on success, receives the time at which the current response will be removed from the cache.

Return Value

Returns S_OK on success or an error HRESULT on failure.

See Also

Reference

IPageCacheControl Interface

IPageCacheControl::SetExpiration

Other Resources

IPageCacheControl Members

IPageCacheControl::IsCached

Call this method to determine whether the current response will be cached.

```
BOOL IsCached( );
```

Return Value

Returns TRUE if the current response will be cached, FALSE otherwise.

See Also

Reference

IPageCacheControl Interface

Other Resources

IPageCacheControl Members

IPageCacheControl::SetExpiration

Call this method to specify the time at which the current response should be removed from the cache.

```
HRESULT SetExpiration(
   FILETIME ftExpiration
);
```

Parameters

ftExpiration

The time at which the current response should be removed from the cache.

Return Value

Returns S_OK on success or an error HRESULT on failure.

See Also

Reference

IPageCacheControl Interface
FILETIME
IPageCacheControl::GetExpiration

Other Resources

IPageCacheControl Members

IRequestHandler Interface

This interface provides the methods necessary for a request handler.

```
__interface ATL_NO_VTABLE
__declspec( uuid( "D57F8D0C-751A-4223-92BC-0B29F65D2453" ))
IRequestHandler :
   public IUnknown
```

Remarks

This interface is implemented by IRequestHandlerImpl and, as a consequence, CRequestHandlerT and CSoapHandler.

Requirements

Header: atlsiface.h

Example

Although it is rare that you'll need to implement this interface yourself, this example shows an **IRequestHandler** implementation that is independent of CRequestHandlerT:

```
class CRequestHandler :
    public IRequestHandlerImpl<CRequestHandler>,
    public CComObjectRootEx<CComSingleThreadModel>
{
private:

    CHttpResponse m_HttpResponse;
    CHttpRequest m_HttpRequest;
    bool m_bChildRequest;

public:

    BEGIN_COM_MAP(CRequestHandler)
        COM_INTERFACE_ENTRY(IRequestHandler)
    END_COM_MAP()

    CRequestHandler() :
        m_bChildRequest(false)
    {
    }

    HTTP_CODE GetFlags(LPDWORD pdwStatus)
    {
        *pdwStatus = ATLS_FLAG_NONE;
        return HTTP_SUCCESS;
    }

    HTTP_CODE InitializeChild(
        AtlServerRequest *pRequestInfo,
        IServiceProvider *pProvider,
        IHttpRequestLookup *pLookup)
    {
        m_bChildRequest = true;

        // Initialize members inherited from IRequestHandlerImpl.
        m_hInstHandler = pRequestInfo->hInstDll;
        m_spServiceProvider = pProvider;
        if (pRequestInfo->pServerContext)
        {
            m_spServerContext = pRequestInfo->pServerContext;
        }
```

```
    HTTP_CODE hcErr = HTTP_SUCCESS;
    if (pLookup != NULL)
    {
        // Initialize the CHttpResponse member.
        BOOL bRet = m_HttpResponse.Initialize(pLookup);
        if (bRet)
        {
            if (m_spServerContext != NULL)
            {
                bRet = m_HttpResponse.Initialize(m_spServerContext);
            }
            if (bRet)
            {
                // Initialize the CHttpRequest member.
                bRet = m_HttpRequest.Initialize(pLookup);
                if (bRet && m_spServerContext != NULL)
                {
                    bRet =
                         m_HttpRequest.Initialize(m_spServerContext);
                }
            }
        }

        if (!bRet)
        {
            hcErr = HTTP_FAIL;
        }
    }

    // Child handlers should not buffer their output.
    m_HttpResponse.SetBufferOutput(FALSE);

    return hcErr;
}

HTTP_CODE InitializeHandler(
    AtlServerRequest *pRequestInfo,
    IServiceProvider *pProvider)
{
    m_bChildRequest = false;

    // Initialize members inherited from IRequestHandlerImpl.
    m_hInstHandler = pRequestInfo->hInstDll;
    m_spServiceProvider = pProvider;
    m_spServerContext = pRequestInfo->pServerContext;

    // Initialize the CHttpResponse member.
    BOOL bRet = m_HttpResponse.Initialize(m_spServerContext);
    if (bRet)
    {
        // Initialize the CHttpRequest member.
        bRet = m_HttpRequest.Initialize(m_spServerContext);
    }

    return bRet ? HTTP_SUCCESS : HTTP_FAIL;
}

HTTP_CODE HandleRequest(
    AtlServerRequest *pRequestInfo,
    IServiceProvider *pProvider)
{
    _ATLTRY
    {
        m_HttpResponse << "<html><body>";
        if (m_bChildRequest)
        {
            m_HttpResponse << "<h1>Child Request</h1>";
        }
```

```
            else
            {
                m_HttpResponse << "<h1>Top-Level Request</h1>";
            }
            m_HttpResponse << "</html></body>";
        }
        _ATLCATCHALL()
        {
            return HTTP_ERROR(500, ISE_SUBERR_OUTOFMEM);
        }

        return HTTP_SUCCESS;
    }

    void UninitializeHandler()
    {
        // Flush the response explicitly.
        m_HttpResponse.Flush(TRUE);
    }

};  // class CRequestHandler
```

See Also

Other Resources

IRequestHandler Members

ATL Server Classes

IRequestHandler Members

Methods

GetFlags	Implement this method to return the flags that indicate the services such as asynchronous processing or page caching that this request handler expects to be available.
HandleRequest	Implement this method to handle an HTTP request.
InitializeChild	Implement this method to initialize the request handler when it is used from a subhandler tag or include tag.
InitializeHandler	Implement this method to initialize the request handler when it is used from a handler tag or loaded by any means except a subhandler tag or include tag.
UninitializeHandler	Implement this method to uninitialize the request handler.

See Also

Reference

IRequestHandler Interface

IRequestHandler Methods

For information about the methods in **IRequestHandler**, see IRequestHandler Members.

IRequestHandler::GetFlags

Implement this method to return the flags that indicate the services such as asynchronous processing or page caching that this request handler expects to be available.

```
HTTP_CODE GetFlags(
   DWORD* pdwStatus
);
```

Parameters

pdwStatus

The address of a DWORD that will hold the request initialization flags on successful return of this method.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

This method is called for primary request handlers and included handlers. It is not called for subhandlers.

Example

See IRequestHandler Overview.

See Also

Reference

IRequestHandler Interface

Request Initialization Flags

Other Resources

IRequestHandler Members

IRequestHandler::HandleRequest

Implement this method to handle an HTTP request.

```
HTTP_CODE HandleRequest(
   AtlServerRequest* pRequestInfo,
   IServiceProvider* pProvider
);
```

Parameters

pRequestInfo

The information about the request to be handled.

pProvider

The service provider.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Example

See IRequestHandler Overview.

See Also

Reference

IRequestHandler Interface
IRequestHandlerImpl::HandleRequest
CRequestHandlerT::HandleRequest

Other Resources

IRequestHandler Members

IRequestHandler::InitializeChild

Implement this method to initialize the request handler when it is used from a subhandler tag or include tag.

```
HTTP_CODE InitializeChild(
   AtlServerRequest* pRequestInfo,
   IServiceProvider* pProvider,
   IHttpRequestLookup* pLookup
);
```

Parameters

pRequestInfo

The information about the request to be handled.

pProvider

The service provider.

pLookup

The request lookup.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

This method is called for subhandlers and included handlers. It is not called for primary handlers.

Example

See IRequestHandler Overview.

See Also

Reference

IRequestHandler Interface

Other Resources

IRequestHandler Members

IRequestHandler::InitializeHandler

Implement this method to initialize the request handler when it is used from a handler tag or loaded by any means except a subhandler tag or include tag.

```
HTTP_CODE InitializeHandler(
   AtlServerRequest* pRequestInfo,
   IServiceProvider* pProvider
);
```

Parameters

pRequestInfo

The information about the request to be handled.

pProvider

The service provider.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

This method is called for primary request handlers. It is not called for subhandlers and included handlers.

Example

See IRequestHandler Overview.

See Also

Reference

IRequestHandler Interface

Other Resources

IRequestHandler Members

IRequestHandler::UninitializeHandler

Implement this method to uninitialize the request handler.

```
void UninitializeHandler( );
```

Remarks

This method is called just before the request handler is destroyed.

Example

See IRequestHandler Overview.

See Also

Reference

IRequestHandler Interface

Other Resources

IRequestHandler Members

IRequestHandlerImpl Class

This class provides a default implementation of the IRequestHandler interface.

```
template <
   typename THandler
>
class IRequestHandlerImpl :
   public IRequestHandler
```

Parameters

THandler

A class derived from **IRequestHandlerImpl**. This class must be usable as a template argument in the CComObjectNoLock class template.

Remarks

This class acts as a base class for CRequestHandlerT, CSDLGenerator, and CSoapHandler.

Requirements

Header: atlisapi.h

See Also

Tasks

MantaWeb Sample: Online Personal Information Management

Reference

ForceLogin
ShowBrowser
IRequestHandler Interface
CRequestHandlerT Class

Other Resources

IRequestHandlerImpl Members
ATL Server Classes

IRequestHandlerImpl Members

Methods

CachePage	Override this non-virtual method to specify whether the pages generated by this request handler can be cached.
GetAsyncFlags	Override this non-virtual method to return flags specifying whether the request handler requires asynchronous processing to handle the current request.
GetFlags	This implementation of IRequestHandler::GetFlags calls GetAsyncFlags and CachePage to get the flags to return.
HandleRequest	This implementation of IRequestHandler::HandleRequest returns HTTP_FAIL.
InitializeChild	This implementation of IRequestHandler::InitializeChild simply delegates to InitializeHandler.
InitializeHandler	This implementation of IRequestHandler::InitializeHandler stores pointers to the service provider and server context, retains the instance handle of the DLL, and queries the service provider for the IDllCache interface.
IRequestHandlerImpl	The constructor for the request handler.
SetAsyncFlags	Call this method to set the asynchronous flags of the request handler.
UninitializeHandler	This implementation of IRequestHandler::UninitializeHandler does nothing.

Static Functions

CreateRequestHandler	This method is called to create the request handler.
GetHandlerFlags	Override this non-virtual function to return the handler flags from the request handler.
InitRequestHandlerClass	Override this non-virtual function to do any initialization required for your request handler class before any instances of the class have been created.
UninitRequestHandlerClass	Override this non-virtual function to do any cleanup required for your request handler class after all instances of the class have been destroyed.

Data Members

m_dwAsyncFlags	Flags specifying whether the request handler requires asynchronous processing.
m_hInstHandler	The instance handle of the DLL.
m_spExtension	The ISAPI extension.
m_spServerContext	The server context.
m_spServiceProvider	The service provider.

See Also

Reference

IRequestHandlerImpl Class

Other Resources

IRequestHandler Members

IRequestHandlerImpl Methods

For information about the methods in **IRequestHandlerImpl**, see IRequestHandlerImpl Members.

IRequestHandlerImpl::CachePage

Override this non-virtual method to specify whether the pages generated by this request handler can be cached.

```
BOOL CachePage( );
```

Return Value

Returns **TRUE** if the page can be cached, **FALSE** otherwise.

Remarks

The default implementation in this class returns **FALSE**. This method is called by IRequestHandlerImpl::GetFlags.

See Also

Reference

IRequestHandlerImpl Class

IRequestHandlerImpl::GetFlags

Other Resources

IRequestHandlerImpl Members

IRequestHandlerImpl::GetAsyncFlags

Override this non-virtual method to return flags specifying whether the request handler requires asynchronous processing to handle the current request.

```
DWORD GetAsyncFlags( );
```

Return Value

Returns the flags specifying whether the request handler requires asynchronous processing (ATLSRV_INIT_USEASYNC or ATLSRV_INIT_USEASYNC_EX).

Remarks

The default implementation in this class retrieves the flags from IRequestHandlerImpl::m_dwAsyncFlags, which is initialized to zero and may be changed by IRequestHandlerImpl::SetAsyncFlags.

Overrides of this method may be applied to derived classes using DECLARE_ASYNC_HANDLER or DECLARE_ASYNC_HANDLER_EX.

See Also

Reference

IRequestHandlerImpl Class
IRequestHandlerImpl::SetAsyncFlags
IRequestHandlerImpl::m_dwAsyncFlags
Request Initialization Flags

Other Resources

IRequestHandlerImpl Members

IRequestHandlerImpl::GetFlags

This implementation of IRequestHandler::GetFlags returns the flags provided by GetAsyncFlags and CachePage.

```
HTTP_CODE GetFlags(
   DWORD * pdwStatus
);
```

Parameters

pdwStatus

[out] Address of the DWORD variable that, on success, receives the request initialization flags.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

This method returns the flags that indicate the services such as asynchronous processing or page caching that this request handler expects to be available.

See Also

Reference

IRequestHandlerImpl Class
IRequestHandler::GetFlags
Request Initialization Flags

Other Resources

IRequestHandlerImpl Members

IRequestHandlerImpl::HandleRequest

This implementation of IRequestHandler::HandleRequest returns HTTP_SUCCESS.

```
HTTP_CODE HandleRequest(
   AtlServerRequest* /* pRequestInfo */,
   IServiceProvider* /* pServiceProvider */
);
```

Parameters

pRequestInfo

The information about the request to be handled.

pServiceProvider

The service provider.

Return Value

Returns HTTP_SUCCESS.

See Also

Reference

IRequestHandlerImpl Class
HTTP_CODE
IRequestHandler::HandleRequest
CRequestHandlerT::HandleRequest

Other Resources

IRequestHandlerImpl Members

IRequestHandlerImpl::InitializeChild

This implementation of IRequestHandler::InitializeChild simply delegates to IRequestHandlerImpl::InitializeHandler.

```
HTTP_CODE InitializeChild(
   AtlServerRequest* pRequestInfo,
   IServiceProvider* pProvider,
   IHttpRequestLookup* /*pLookup*/
);
```

Parameters

pRequestInfo

The information about the request to be handled.

pProvider

The service provider.

pLookup

The request lookup.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

IRequestHandlerImpl Class
IRequestHandler::InitializeChild
IRequestHandlerImpl::InitializeHandler

Other Resources

IRequestHandlerImpl Members

IRequestHandlerImpl::InitializeHandler

This implementation of IRequestHandler::InitializeHandler stores pointers to the service provider and server context and retains the instance handle of the DLL containing the request handler.

```
HTTP_CODE InitializeHandler(
   AtlServerRequest* pRequestInfo,
   IServiceProvider* pProvider
);
```

Parameters

pRequestInfo

The information about the request to be handled.

pProvider

The service provider.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

See Also

Reference

IRequestHandlerImpl Class

IRequestHandler::InitializeHandler

Other Resources

IRequestHandlerImpl Members

IRequestHandlerImpl::IRequestHandlerImpl

The constructor for the request handler.

```
IRequestHandlerImpl( );
```

See Also

Reference

IRequestHandlerImpl Class

Other Resources

IRequestHandlerImpl Members

IRequestHandlerImpl::SetAsyncFlags

Call this method to set the asynchronous flags of the request handler.

```
void SetAsyncFlags(
   DWORD dwAsyncFlags
);
```

Parameters

dwAsyncFlags

The flags specifying whether the handler requires asynchronous processing. Can be either ATLSRV_INIT_USEASYNC or ATLSRV_INIT_USEASYNC_EX.

Remarks

This method stores the flags in IRequestHandlerImpl::m_dwAsyncFlags.

See Also

Reference

IRequestHandlerImpl Class
IRequestHandlerImpl::GetAsyncFlags
Request Initialization Flags

Other Resources

IRequestHandlerImpl Members

IRequestHandlerImpl::UninitializeHandler

This implementation of IRequestHandler::UninitializeHandler does nothing.

```
void UninitializeHandler( );
```

Remarks

The cleanup required by this class is handled by the destructors of its data members.

See Also

Reference

IRequestHandlerImpl Class

Other Resources

IRequestHandlerImpl Members

IRequestHandlerImpl Static Functions

For information about the static functions in **IRequestHandlerImpl**, see IRequestHandlerImpl Members.

IRequestHandlerImpl::CreateRequestHandler

This method is called to create the request handler.

```
static BOOL CreateRequestHandler(
   IIsapiExtension * pExtension,
   IUnknown ** ppOut
);
```

Parameters

pExtension

[in] The ISAPI extension.

ppOut

[out] Address of the pointer variable that, on success, receives the **IUnknown** interface pointer of the newly created request handler.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

The implementation in this class creates the request handler as an ATL COM object when asynchronous processing is required (as indicated by GetHandlerFlags); otherwise, it creates the request handler on the current thread's heap.

See Also

Reference

IRequestHandlerImpl Class

IRequestHandlerImpl::GetHandlerFlags

Other Resources

IRequestHandlerImpl Members

IRequestHandlerImpl::GetHandlerFlags

Override this non-virtual function to return the handler flags from the request handler.

```
static DWORD GetHandlerFlags( );
```

Return Value

Returns ATLS_FLAG_NONE if asynchronous processing is not required. Returns ATLS_FLAG_ASYNC if asynchronous processing may be required (that is, if the handler might return ATLSRV_INIT_USEASYNC or ATLSRV_INIT_USEASYNC_EX from GetAsyncFlags).

Remarks

The default implementation of this function returns ATLS_FLAG_NONE. Alternate versions of this function can be added to your class by adding DECLARE_ASYNC_HANDLER or DECLARE_ASYNC_HANDLER_EX to your class definition.

This function is called by IRequestHandlerImpl::CreateRequestHandler.

See Also

Reference

IRequestHandlerImpl Class

Other Resources

IRequestHandlerImpl Members

IRequestHandlerImpl::InitRequestHandlerClass

Override this non-virtual function to do any initialization required for your request handler class before any instances of the class have been created.

```
static BOOL InitRequestHandlerClass(
   IHttpServerContext * pContext,
   IIsapiExtension * pExt
);
```

Parameters

pContext

The server context.

pExt

The ISAPI extension.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This function is called by the ATL Server infrastructure when the Web application DLL is first loaded. The corresponding function for cleanup is UninitRequestHandlerClass.

The default implementation in this class simply returns **TRUE**.

See Also

Reference

IRequestHandlerImpl Class

IRequestHandlerImpl::UninitRequestHandlerClass

Other Resources

IRequestHandlerImpl Members

IRequestHandlerImpl::UninitRequestHandlerClass

Override this non-virtual function to do any cleanup required for your request handler class after all instances of the class have been destroyed.

```
static void UninitRequestHandlerClass( );
```

Remarks

This function is called by the ATL Server infrastructure just before the request handler DLL is unloaded if the corresponding initialization function, IRequestHandlerImpl::InitRequestHandlerClass, was successful. It will not be called if the initialization failed.

The default implementation in this class does nothing.

See Also

Reference

IRequestHandlerImpl Class

IRequestHandlerImpl::InitRequestHandlerClass

Other Resources

IRequestHandlerImpl Members

IRequestHandlerImpl Data Members

For information about the data members in **IRequestHandlerImpl**, see IRequestHandlerImpl Members.

IRequestHandlerImpl::m_dwAsyncFlags

Flags specifying whether the request handler requires asynchronous processing.

```
DWORD m_dwAsyncFlags;
```

See Also

Reference

IRequestHandlerImpl Class
IRequestHandlerImpl::GetAsyncFlags
IRequestHandlerImpl::SetAsyncFlags

Other Resources

IRequestHandlerImpl Members

IRequestHandlerImpl::m_hInstHandler

The instance handle of the DLL containing the request handler.

```
HINSTANCE m_hInstHandler;
```

See Also

Reference

IRequestHandlerImpl Class

Other Resources

IRequestHandlerImpl Members

IRequestHandlerImpl::m_spExtension

The ISAPI extension.

```
CComPtr<IIsapiExtension> m_spExtension;
```

Remarks

This member is set by IRequestHandlerImpl::CreateRequestHandler when the handler is created if asynchronous processing is requested.

See Also

Reference

IRequestHandlerImpl Class

IIsapiExtension Interface

Other Resources

IRequestHandlerImpl Members

IRequestHandlerImpl::m_spServerContext

The server context.

```
CComPtr<IHttpServerContext> m_spServerContext;
```

Example

See the ForceLogin Sample.

See Also

Reference

IRequestHandlerImpl Class

IHttpServerContext Interface

Other Resources

IRequestHandlerImpl Members

IRequestHandlerImpl::m_spServiceProvider

The service provider.

```
CComPtr<IServiceProvider> m_spServiceProvider;
```

Remarks

This member can be used to obtain access to services exposed by the ISAPI extension.

Example

See the MantaWeb Sample and the ShowBrowser Sample.

See Also

Reference

IRequestHandlerImpl Class

Other Resources

IRequestHandlerImpl Members

IRequestStats Interface

This interface provides methods to get statistics on how requests are being handled.

```
__interface ATL_NO_VTABLE
__declspec(uuid("2B75C68D-0DDF-48d6-B58A-CC7C2387A6F2"))
IRequestStats :
   public IUnknown
```

Remarks

This interface is implemented by CIsapiExtension.

Requirements

Header: atlsiface.h

See Also

Other Resources

IRequestStats Members

ATL Server Classes

IRequestStats Members

Methods

GetActiveThreads	Call this method to get the number of threads currently handling a request.
GetAvgResponseTime	Call this method to get the average response time for handling a request.
GetCurrWaiting	Call this method to get the current number of queued requests.
GetFailedRequests	Call this method to get the total number of failed requests.
GetMaxWaiting	Call this method to get the maximum number of requests that have been queued at any one time since the ISAPI extension was loaded.
GetTotalRequests	Call this method to get the total number of handled requests.

See Also

Reference

IRequestStats Interface

IRequestStats Methods

For information about the methods in **IRequestStats**, see IRequestStats Members.

IRequestStats::GetActiveThreads

Call this method to get the number of threads currently handling a request.

```
long GetActiveThreads( );
```

Return Value

Returns the number of threads currently handling a request.

Example

```
[ tag_name(name = "GetActiveThreads") ]
HTTP_CODE OnGetActiveThreads(void)
{
    CComPtr<IRequestStats> spRequestStats;

    spRequestStats = com_cast<IRequestStats>(m_spExtension);
    if (spRequestStats == NULL)
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(spRequestStats->GetActiveThreads()))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IRequestStats Interface

Other Resources

IRequestStats Members

IRequestStats::GetAvgResponseTime

Call this method to get the average response time for handling a request.

```
long GetAvgResponseTime( );
```

Return Value

Returns the average response time for handling a request.

Remarks

The average response time is measured as the total response time divided by the total number of requests handled.

Example

```
[ tag_name(name = "GetAvgResponseTime") ]
HTTP_CODE OnGetAvgResponseTime(void)
{
    CComPtr<IRequestStats> spRequestStats;

    spRequestStats = com_cast<IRequestStats>(m_spExtension);
    if (spRequestStats == NULL)
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(spRequestStats->GetAvgResponseTime()))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IRequestStats Interface

Other Resources

IRequestStats Members

IRequestStats::GetCurrWaiting

Call this method to get the current number of queued requests.

```
long GetCurrWaiting( );
```

Return Value

Returns the current number of queued requests.

Remarks

Queued requests are those that have been received and placed in the queue but are not yet being handled by one of the active threads.

Example

```
[ tag_name(name = "GetCurrWaiting") ]
HTTP_CODE OnGetCurrWaiting(void)
{
    CComPtr<IRequestStats> spRequestStats;

    spRequestStats = com_cast<IRequestStats>(m_spExtension);
    if (spRequestStats == NULL)
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(spRequestStats->GetCurrWaiting()))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IRequestStats Interface

Other Resources

IRequestStats Members

IRequestStats::GetFailedRequests

Call this method to get the total number of failed requests.

```
long GetFailedRequests( );
```

Return Value

Returns the total number of failed requests.

Example

```
[ tag_name(name = "GetFailedRequests") ]
HTTP_CODE OnGetFailedRequests(void)
{
    CComPtr<IRequestStats> spRequestStats;

    spRequestStats = com_cast<IRequestStats>(m_spExtension);
    if (spRequestStats == NULL)
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(spRequestStats->GetFailedRequests()))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IRequestStats Interface

Other Resources

IRequestStats Members

IRequestStats::GetMaxWaiting

Call this method to get the maximum number of requests that have been queued at any one time since the ISAPI extension was loaded.

```
long GetMaxWaiting( );
```

Return Value

Returns the maximum number of queued requests.

Example

```
[ tag_name(name = "GetMaxWaiting") ]
HTTP_CODE OnGetMaxWaiting(void)
{
    CComPtr<IRequestStats> spRequestStats;

    spRequestStats = com_cast<IRequestStats>(m_spExtension);
    if (spRequestStats == NULL)
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(spRequestStats->GetMaxWaiting()))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IRequestStats Interface

Other Resources

IRequestStats Members

IRequestStats::GetTotalRequests

Call this method to get the total number of handled requests.

```
long GetTotalRequests( );
```

Return Value

Returns the total number of handled requests.

Remarks

Add the result of **GetTotalRequests** and IRequestStats::GetCurrWaiting to get the total number of requests received.

Example

```
[ tag_name(name = "GetTotalRequests") ]
HTTP_CODE OnGetTotalRequests(void)
{
    CComPtr<IRequestStats> spRequestStats;

    spRequestStats = com_cast<IRequestStats>(m_spExtension);
    if (spRequestStats == NULL)
    {
        return HTTP_FAIL;
    }

    if (!m_HttpResponse.Write(spRequestStats->GetTotalRequests()))
    {
        return HTTP_FAIL;
    }

    return HTTP_SUCCESS;
}
```

See Also

Reference

IRequestStats Interface

IRequestStats::GetCurrWaiting

Other Resources

IRequestStats Members

ISAXContentHandlerImpl Class

This class provides a non-functional implementation of the ISAXContentHandler interface that can be used as a base class by classes that need to provide the binary interface of **ISAXContentHandler** but don't need usable implementations for all the methods.

```
class ISAXContentHandlerImpl :
   public ISAXContentHandler
```

Remarks

All the methods implemented by this class return S_OK. Derived classes can override only those methods for which they need to do special processing.

Requirements

Header: atlsoap.h

See Also

Other Resources

ISAXContentHandlerImpl Members

ISAXContentHandlerImpl Members

Methods

characters	Non-functional implementation of **ISAXContentHandler::characters**
endDocument	Non-functional implementation of **ISAXContentHandler::endDocument**
endElement	Non-functional implementation of **ISAXContentHandlerImpl::endElement**
endPrefixMapping	Non-functional implementation of **ISAXContentHandlerImpl::endPrefixMapping**
ignorableWhitespace	Non-functional implementation of **ISAXContentHandlerImpl::ignorableWhitespace**
processingInstruction	Non-functional implementation of **ISAXContentHandlerImpl::processingInstruction**
putDocumentLocator	Non-functional implementation of **ISAXContentHandlerImpl::putDocumentLocator**
skippedEntity	Non-functional implementation of **ISAXContentHandlerImpl::skippedEntity**
startDocument	Non-functional implementation of **ISAXContentHandlerImpl::startDocument**
startElement	Non-functional implementation of **ISAXContentHandlerImpl::startElement**
startPrefixMapping	Non-functional implementation of **ISAXContentHandlerImpl::startPrefixMapping**

See Also

Reference

ISAXContentHandlerImpl Class

ISAXContentHandlerImpl Methods

For information about the methods in **ISAXContentHandlerImpl**, see ISAXContentHandlerImpl Members

ISAXContentHandlerImpl::characters

Non-functional implementation of **ISAXContentHandler::characters**.

```
HRESULT __stdcall characters(
   const wchar_t * /* wszChars */,
   int /* cchChars */
);
```

Remarks

Always returns S_OK.

See Also

Reference

ISAXContentHandlerImpl Class

Other Resources

ISAXContentHandlerImpl Members

ISAXContentHandlerImpl::endDocument

Non-functional implementation of **ISAXContentHandler::endDocument**.

```
HRESULT __stdcall endDocument( );
```

Remarks

Always returns S_OK.

See Also

Reference

ISAXContentHandlerImpl Class

Other Resources

ISAXContentHandlerImpl Members

ISAXContentHandlerImpl::endElement

Non-functional implementation of ISAXContentHandler::endElement.

```
HRESULT __stdcall endElement(
   const wchar_t * /* wszNamespaceUri */,
   int /* cchNamespaceUri */,
   const wchar_t * /* wszLocalName */,
   int /* cchLocalName */,
   const wchar_t * /* wszQName */,
   int /* cchQName */
);
```

Remarks

Always returns S_OK.

See Also

Reference

ISAXContentHandlerImpl Class

Other Resources

ISAXContentHandlerImpl Members

ISAXContentHandlerImpl::endPrefixMapping

Non-functional implementation of **ISAXContentHandler::endPrefixMapping**.

```
HRESULT __stdcall endPrefixMapping(
   const wchar_t * /* wszPrefix */,
   int /* cchPrefix */
);
```

Remarks

Always returns S_OK.

See Also

Reference

ISAXContentHandlerImpl Class

Other Resources

ISAXContentHandlerImpl Members

ISAXContentHandlerImpl::ignorableWhitespace

Non-functional implementation of ISAXContentHandler::ignorableWhitespace.

```
HRESULT __stdcall ignorableWhitespace(
   const wchar_t * /* wszChars */,
   int /* cchChars */
);
```

Remarks

Always returns S_OK.

See Also

Reference

ISAXContentHandlerImpl Class

Other Resources

ISAXContentHandlerImpl Members

ISAXContentHandlerImpl::processingInstruction

Non-functional implementation of **ISAXContentHandler::processingInstruction**.

```
HRESULT __stdcall processingInstruction(
   const wchar_t * /* wszTarget */,
   int /* cchTarget */,
   const wchar_t * /* wszData */,
   int /* cchData */
);
```

Remarks

Always returns S_OK.

See Also

Reference

ISAXContentHandlerImpl Class

Other Resources

ISAXContentHandlerImpl Members

ISAXContentHandlerImpl::putDocumentLocator

Non-functional implementation of ISAXContentHandler::putDocumentLocator.

```
HRESULT __stdcall putDocumentLocator(
   ISAXLocator * /* pLocator */
);
```

Remarks

Always returns S_OK.

See Also

Reference

ISAXContentHandlerImpl Class

Other Resources

ISAXContentHandlerImpl Members

ISAXContentHandlerImpl::skippedEntity

Non-functional implementation of **ISAXContentHandler::skippedEntity**.

```
HRESULT __stdcall skippedEntity(
   const wchar_t * /* wszName */,
   int /* cchName */
);
```

Remarks

Always returns S_OK.

See Also

Reference

ISAXContentHandlerImpl Class

Other Resources

ISAXContentHandlerImpl Members

ISAXContentHandlerImpl::startDocument

Non-functional implementation of **ISAXContentHandler::startDocument**.

```
HRESULT __stdcall startDocument( );
```

Remarks

Always returns S_OK.

See Also

Reference

ISAXContentHandlerImpl Class

Other Resources

ISAXContentHandlerImpl Members

ISAXContentHandlerImpl::startElement

Non-functional implementation of **ISAXContentHandler::startElement**.

```
HRESULT __stdcall startElement(
   const wchar_t * /* wszNamespaceUri */,
   int /* cchNamespaceUri */,
   const wchar_t * /* wszLocalName */,
   int /* cchLocalName */,
   const wchar_t * /* wszQName */,
   int /* cchQName */,
   ISAXAttributes * /* pAttributes */
);
```

Remarks

Always returns S_OK.

See Also

Reference

ISAXContentHandlerImpl Class

Other Resources

ISAXContentHandlerImpl Members

ISAXContentHandlerImpl::startPrefixMapping

Non-functional implementation of ISAXContentHandler::startPrefixMapping.

```
HRESULT __stdcall startPrefixMapping(
   const wchar_t * /* wszPrefix */,
   int /* cchPrefix */,
   const wchar_t * /* wszUri */,
   int /* cchUri */
);
```

Remarks

Always returns S_OK.

See Also

Reference

ISAXContentHandlerImpl Class

Other Resources

ISAXContentHandlerImpl Members

ISession Interface

This interface allows the storage and retrieval of session-specific data.

```
__interface
__declspec( uuid( "DEB69BE3-7AC9-4a13-9519-266C1EA3AB39" ))
ISession :
   public IUnknown
```

Remarks

The **ISession** interface allows session variables to be stored, modified, deleted, and enumerated, and allows session timeout values to be retrieved and assigned.

The actual storage mechanism used to implement the **ISession** functionality varies depending on how the underlying implementation is configured. For more information, see CSessionStateService, Session State Tasks, and Using Session State Support.

Requirements

Header: atlsiface.h

See Also

Tasks

OnlineAddressBook Sample: Stores an Address Book in an XML Web Service

SessionSettings Sample: Session State Demonstration

Reference

ShowSession

Other Resources

Session-State Services (ATL Server)

ISession Members

ATL Server Classes

ISession Members

Methods

BeginVariableEnum	Call this method to start an enumeration of the variables in the current session.
CloseEnum	Call this method to close an enumeration handle and free any associated resources.
GetCount	Call this method to retrieve the number of variables stored by the current session.
GetNextVariable	Call this method to retrieve a variable from the session and increment the enumeration position.
GetVariable	Call this method to retrieve the value of a variable, given a valid variable name.
IsExpired	Call this method to determine session expiration status.
RemoveAllVariables	Call this method to remove all variables from a session.
RemoveVariable	Call this method to remove a variable from the session.
SetTimeout	Call this method to set the session timeout duration.
SetVariable	Call this method to modify or add a session variable.

See Also

Reference

ISession Interface

Other Resources

Session-State Services (ATL Server)

ISession Methods

For information about the methods in **ISession**, see ISession Members.

ISession::BeginVariableEnum

Call this method to start an enumeration of the variables in the current session.

```
STDMETHOD(BeginVariableEnum)(
   POSITION *pPOS,
   HSESSIONENUM *phEnumHandle
);
```

Parameters

pPOS

[out] The address of a **POSITION** variable that, on success, receives the position of the first session variable for later use with ISession::GetNextVariable.

phEnumHandle

[out] The address of a handle that, on success, identifies this enumeration for later calls to ISession::GetNextVariable and ISession::CloseEnum.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

See Also

Reference

ISession Interface
ISession::GetNextVariable
ISession::CloseEnum

Other Resources

Session-State Services (ATL Server)
ISession Members

ISession::CloseEnum

Call this method to close an enumeration handle and free any associated resources.

```
STDMETHOD(CloseEnum)(
   HSESSIONENUM hEnumHandle
);
```

Parameters

hEnumHandle

An enumeration handle previously obtained by a call to ISession::BeginVariableEnum.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

See Also

Reference

ISession Interface
ISession::BeginVariableEnum
ISession::GetNextVariable

Other Resources

Session-State Services (ATL Server)
ISession Members

ISession::GetCount

Call this method to retrieve the number of variables stored by the current session.

```
STDMETHOD(GetCount)(
   long *pnCount
);
```

Parameters

pnCount

[out] The address of a variable that, on success, receives the count of the variables in the current session.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

See the ShowSession Sample.

See Also

Reference

ISession Interface

Other Resources

Session-State Services (ATL Server)

ISession Members

ISession::GetNextVariable

Call this method to retrieve a variable from the session and increment the enumeration position.

```
STDMETHOD(GetNextVariable)(
   POSITION *pPOS,
   VARIANT *pVal,
   HSESSIONENUM hEnum,
   LPSTR szName,
   DWORD dwLen
);
```

Parameters

pPOS

[in, out] The address of a **POSITION** variable that, on entry, contains the position of the session variable to retrieve and, on success, receives the position of the next variable.

pVal

[out] The address of a **VARIANT** variable that, on success, receives the value of the session variable.

hEnum

The enumeration handle previously obtained with a call to BeginVariableEnum.

szName

A buffer for the storage of the retrieved variable's name.

dwLen

The size of the buffer, *szName*, in bytes.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure. Will return **E_OUTOFMEMORY** if the name of the session variable is too large to be copied to the *szName* buffer. Other return values indicate that there are no more variables for enumeration.

See Also

Reference

ISession Interface
ISession::BeginVariableEnum
ISession::CloseEnum

Other Resources

Session-State Services (ATL Server)
ISession Members

ISession::GetVariable

Call this method to retrieve the value of a variable, given a valid variable name.

```
STDMETHOD(GetVariable)(
   LPCSTR szName,
   VARIANT *pVal
);
```

Parameters

szName

A string containing the name of the desired session variable.

pVal

[out] The address of a **VARIANT** variable that, on success, receives the session variable value.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

See the OnlineAddressBook Sample, the SessionSettings Sample, and the ShowSession Sample.

See Also

Reference

ISession Interface

ISession::SetVariable

Other Resources

Session-State Services (ATL Server)

ISession Members

ISession::IsExpired

Call this method to determine session expiration status.

```
STDMETHOD(IsExpired)();
```

Return Value

Returns **S_OK** if the session has expired, **S_FALSE** if the session has not expired, or an error **HRESULT** on failure.

See Also

Reference

ISession Interface

ISession::SetTimeout

Other Resources

ISession Members

ISession::RemoveAllVariables

Call this method to remove all variables from a session.

```
STDMETHOD(RemoveAllVariables)();
```

Return Value

Returns **S_OK** on success or an error **HRESULT** on failure.

See Also

Reference

ISession Interface

Other Resources

Session-State Services (ATL Server)

ISession Members

ISession::RemoveVariable

Call this method to remove a variable from the session.

```
STDMETHOD(RemoveVariable)(
   LPCSTR szName
);
```

Parameters

szName

A string containing the name of the session variable to be removed.

Return Value

Returns **S_OK** on success or an error **HRESULT** on failure.

See Also

Reference

ISession Interface

Other Resources

Session-State Services (ATL Server)

ISession Members

ISession::SetVariable

Call this method to modify or add a session variable.

```
STDMETHOD(SetVariable)(
   LPCSTR szName,
   VARIANT NewVal
);
```

Parameters

szName

A string containing the name of the session variable to be assigned or added.

NewVal

The new value of the session variable. See CComVariant for **VARIANT** data type conversion support.

Return Value

Returns **S_OK** on success or an error **HRESULT** on failure.

Remarks

If the variable indicated by *szName* does not exist, it is added. Otherwise the existing variable value is overridden with the value given in *NewVal*.

Example

See the OnlineAddressBook Sample, the SessionSettings Sample, and the ShowSession Sample.

See Also

Reference

ISession Interface
ISession::GetVariable

Other Resources

Session-State Services (ATL Server)
ISession Members

ISession::SetTimeout

Call this method to set the session timeout duration.

```
STDMETHOD(SetTimeout)(
   unsigned __int64 dwNewTimeout
);
```

Parameters

dwNewTimeout

The new timeout value, in milliseconds.

Return Value

Returns **S_OK** on success or an error **HRESULT** on failure.

Remarks

The **SetTimeout** method sets the timeout value of the session represented by the ISession interface. To set the timeout value of all sessions, use the ISessionStateControl::SetSessionTimeout method.

See Also

Reference

ISession Interface

ISession::IsExpired

Other Resources

Session-State Services (ATL Server)

ISession Members

ISessionStateControl Interface

This interface provides methods for controlling ATL Server session-state settings.

```
__interface
__declspec(uuid("6C7F5F56-6CBD-49ee-9797-4C837D4C527A")) ISessionStateControl :
   public IUnknown
```

Remarks

The **ISessionStateControl** interface provides methods that allow the manipulation of system-wide session-state settings such as current session count and timeout values.

While the **ISessionStateControl** interface is concerned with system-wide settings, the ISessionStateService interface is concerned with the state of specific session.

Both the **ISessionStateService** and ISessionStateControl interfaces are exposed by the CSessionStateService class.

Requirements

Header: atlsiface.h

Example

See the SessionSettings Sample and the ShowSession Sample.

See Also

Other Resources

Session-State Services (ATL Server)
ISessionStateControl Members
ATL Server Classes

ISessionStateControl Members

Methods

GetSessionCount	Call this method to retrieve the current number of sessions.
GetSessionTimeout	Call this method to retrieve the current global timeout value.
SetSessionTimeout	Call this method to assign the global session timeout value.

See Also

Reference

ISessionStateControl Interface

Other Resources

Session-State Services (ATL Server)

ISessionStateControl Methods

For information about the methods in **ISessionStateControl**, see ISessionStateControl Members.

ISessionStateControl::GetSessionCount

Call this method to retrieve the current number of sessions.

```
STDMETHOD(GetSessionCount)(
   DWORD* pnSessionCount
);
```

Parameters

pnSessionCount

[out] The address of a **DWORD** that, on success, receives the number of active sessions.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The session count returned represents the number of sessions that are currently available to the underlying session-state implementation. For performance reasons, **GetSessionCount** does not iterate through each session to check timeout values, so the resulting session count may include sessions that have expired but have not been removed from storage.

Example

See the ShowSession Sample.

See Also

Reference

ISessionStateControl Interface

Other Resources

Session-State Services (ATL Server)

ISessionStateControl Members

ISessionStateControl::GetSessionTimeout

Call this method to retrieve the current global timeout value.

```
STDMETHOD(GetSessionTimeout)(
   unsigned __int64 * pnTimeout
);
```

Parameters

pnTimeout

[out] The address of a variable that, on success, receives the timeout of a session, in milliseconds.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

The timeout value indicates the minimum time that session data will be maintained without reference before being swept from storage. The default ATL Server session timeout value is determined by the ATL_SESSION_TIMEOUT macro but can be overridden with SetSessionTimeout.

See Also

Reference

ISessionStateControl Interface

ISessionStateControl::SetSessionTimeout

Concepts

Session-State Support

Other Resources

Session-State Services (ATL Server)

ISessionStateControl Members

ISessionStateControl::SetSessionTimeout

Call this method to assign the global session timeout value.

```
STDMETHOD(SetSessionTimeout)(
   unsigned __int64 nTimeout
);
```

Parameters

nTimeout

The timeout to be used for all sessions, in milliseconds.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

SetSessionTimeout sets the timeout for all active sessions and determines the timeout for future sessions. The ATL_SESSION_TIMEOUT macro determins the initial timeout for all ATL Server sessions.

See Also

Reference

ISessionStateControl Interface

ISessionStateControl::GetSessionTimeout

Other Resources

Session-State Services (ATL Server)

ISessionStateControl Members

ISessionStateService Interface

This interface provides methods for creating, retrieving, and closing session objects.

```
__interface
__declspec( uuid( "C5740C4F-0C6D-4b43-92C4-2AF778F35DDE" ))
ISessionStateService :
   public IUnknown
```

Remarks

The **ISessionStateService** interface provides session manipulation and access methods. Use this interface to create new sessions, to obtain access to existing sessions, and to close sessions that are no longer needed.

While the **ISessionStateService** interface allows the manipulation of specific sessions, the ISessionStateControl interface allows the manipulation of system-wide session-state options, such as default timeout values, and the currently active session count.

Both the **ISessionStateService** and ISessionStateControl interfaces are exposed by the CSessionStateService class.

Requirements

Header: atlsiface.h

See Also

Tasks

OnlineAddressBook Sample: Stores an Address Book in an XML Web Service

SessionSettings Sample: Session State Demonstration

Reference

ShowSession

Other Resources

Session-State Services (ATL Server)

ISessionStateService Members

ATL Server Classes

ISessionStateService Members

Methods

CloseSession	Call this method to close a session, releasing session variables and related resources.
CreateNewSession	Call this method to create a new session and corresponding session ID.
CreateNewSessionByName	Call this method to create a new session, given a session name.
GetSession	Call this method to retrieve a pointer to an existing session.

See Also

Reference

ISessionStateService Interface

Other Resources

Session-State Services (ATL Server)

ISessionStateService Methods

For information about the methods in **ISessionStateService**, see ISessionStateService Members.

ISessionStateService::CloseSession

Call this method to close a session, releasing session variables and related resources.

```
STDMETHOD(CloseSession)(
   LPCSTR szID
);
```

Parameters

szID

The session ID, as obtained by a previous call to CreateNewSession or used with the CreateNewSessionByName method.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

See the OnlineAddressBook Sample and the SessionSettings Sample.

See Also

Reference

ISessionStateService Interface

ISessionStateService::CreateNewSession

Other Resources

Session-State Services (ATL Server)

ISessionStateService Members

ISessionStateService::CreateNewSession

Call this method to create a new session and corresponding session ID.

```
STDMETHOD(CreateNewSession)(
   LPSTR szNewID,
   DWORD* pdwSize,
   ISession** ppSession
);
```

Parameters

szNewID

[out] A string that, on success, receives the ID of the new session.

pdwSize

[in, out] The address of a **DWORD** that, on entry, is the size of the buffer provided by *szNewID*, in bytes. On exit, this DWORD is assigned the number of bytes actually used by the ID (including the null-terminating byte).

ppSession

[out] Address of a pointer that, on success, receives the **ISession** interface for the new session.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

This method will generate a cryptographically unique session ID along with a corresponding session object. The session name size should not be less than MIN_SESSION_KEY_LEN in length**.** To use a specific name to create a new session, use the CreateNewSessionByName method.

Example

See the OnlineAddressBook Sample, the SessionSettings Sample, and the ShowSession Sample.

See Also

Reference

ISessionStateService Interface
ISessionStateService::CloseSession
CSessionNameGenerator Class

Other Resources

Session-State Services (ATL Server)
ISessionStateService Members

ISessionStateService::CreateNewSessionByName

Call this method to create a new session, given a session name.

```
STDMETHOD(CreateNewSessionByName)(
   LPSTR szNewID,
   ISession** ppSession
);
```

Parameters

szNewID

[in] A string containing the name of the session to be created.

ppSession

[out] The address of a pointer that, on success, receives the **ISession** interface for the new session.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

Unlike the CreateNewSession method, which creates both a new session and a new session name, the **CreateNewSessionByName** method creates a new session based on a provided name. If the **CreateNewSessionByName** method fails, *ppSession* will be set to NULL.

See Also

Reference

ISessionStateService Interface
ISessionStateService::CloseSession
CSessionNameGenerator Class

Other Resources

Session-State Services (ATL Server)
ISessionStateService Members

ISessionStateService::GetSession

Call this method to retrieve a pointer to an existing session.

```
STDMETHOD(GetSession)(
   LPCSTR szID,
   ISession** ppSession
);
```

Parameters

szID

A string containing the ID of the desired session.

ppSession

[out] The address of a pointer that, on success, receives the **ISession** interface pointer of the requested session. If the session doesn't exist, *ppSession* will be set to zero.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Remarks

If the session ID provided to **GetSession** represents a valid session previously created with the CreateNewSession or CreateNewSessionByName method, a pointer to this existing session object will be retrieved.

Example

See the OnlineAddressBook Sample, the SessionSettings Sample, and the ShowSession Sample.

See Also

Reference

ISessionStateService Interface

Other Resources

Session-State Services (ATL Server)

ISessionStateService Members

ISharedBlobCache Interface

This SOAP-compatible interface provides methods for adding items to and retrieving items from a cache.

```
[ uuid("AB4AF9CD-8DB1-4974-A617-CF0449578FB9"), object ]
__interface ISharedBlobCache
```

Remarks

This interface is implemented by CSharedCache and CSharedCacheHandler.

Requirements

Header: atlsharedsvc.h

See Also

Reference

CSharedCache Class

CSharedCacheHandler Class

Other Resources

Caching

ISharedBlobCache Members

Caching Reference

ISharedBlobCache Members

Methods

AddItem	Call this method to add the specified item to the cache.
GetItem	Call this method to get the specified item from the cache.

See Also

Reference

ISharedBlobCache Interface

ISharedBlobCache Methods

For information about the methods in **ISharedBlobCache**, see ISharedBlobCache Members

ISharedBlobCache::AddItem

Call this method to add the specified item to the cache.

```
[id(0)]
STDMETHOD(AddItem)(
   [in] BSTR szItemName,
   [in] BSTR szData
);
```

Parameters

szItemName

The name of the item to add to the cache.

szData

The data of the item to add to the cache.

Return Value

Returns S_OK on success or an error HRESULT on failure.

Remarks

If an item of the specified name is already present in the cache, it is replaced with the new item.

See Also

Reference

ISharedBlobCache Interface
ISharedBlobCache::GetItem

Other Resources

ISharedBlobCache Members
Caching

ISharedBlobCache::GetItem

Call this method to get the specified item from the cache.

```
[id(1)]
STDMETHOD(GetItem )(
   [in] BSTR szItemName,
   [out, retval] BSTR * szData
);
```

Parameters

szItemName

The name of the item to get from the cache.

szData

[out] Address of the **BSTR** variable that, on success, receives the data of the item.

Return Value

Returns S_OK on success or an error HRESULT on failure.

Remarks

If an item of the specified name is not present in the cache, this method returns **E_FAIL**.

See Also

Reference

ISharedBlobCache Interface
ISharedBlobCache::AddItem

Other Resources

ISharedBlobCache Members
Caching

IStencilCache Interface

This interface provides methods for adding stencils to and retrieving stencils from a cache.

```
__interface ATL_NO_VTABLE
__declspec(uuid("8702269B-707D-49cc-AEF8-5FFCB3D6891B"))
IStencilCache :
   public IUnknown
```

Remarks

This interface is implemented by CStencilCache.

Requirements

Header: atlcache.h

See Also

Tasks

StencilCache Sample: Uses the ATL Server Stencil Cache

Other Resources

IStencilCache Members

Caching

ATL Server Classes

Caching Reference

IStencilCache Members

Methods

AddRefStencil	Call this method to increment the reference count of a cached stencil.
CacheStencil	Call this method to add a stencil to the cache.
GetStencil	Call this method to get a pointer to a stencil in the cache.
LookupStencil	Call this method to get the handle of a cached stencil.
ReleaseStencil	Call this method to decrement the reference count of a cached stencil.

See Also

Reference

IStencilCache Interface

IStencilCache Methods

For information about the methods in **IStencilCache**, see IStencilCache Members.

IStencilCache::AddRefStencil

Call this method to increment the reference count of a cached stencil.

```
STDMETHOD(AddRefStencil)(
   const HCACHEITEM hStencil
);
```

Parameters

hStencil

The handle of the item whose reference count should be incremented.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The reference count on items in the cache lets the cache know when items are still being used and prevents them from being deleted prematurely. Handles returned by a call to LookupStencil have a single reference count associated with them. The reference count can be decremented by calling ReleaseStencil and incremented by calling **AddRefStencil**. When the reference count associated with a handle reaches zero, the handle should be discarded.

Use **AddRefStencil** when you want to copy a handle and pass ownership on to some other piece of code.

See Also

Reference

IStencilCache Interface

Other Resources

IStencilCache Members

IStencilCache::CacheStencil

Call this method to add a stencil to the cache.

```
STDMETHOD(CacheStencil)(
   LPCSTR szName,
   void * pStencil,
   DWORD dwSize,
   HCACHEITEM * pHandle,
   HINSTANCE hInst,
   IMemoryCacheClient * pClient
);
```

Parameters

szName

A string containing the key that can be used to retrieve the stencil. Typically, this will be the name of the stencil file.

pStencil

The address of the stencil data to be cached. Typically, this will be a pointer to a CStencil-derived class.

dwSize

The size of the stencil data. Typically this will be the size in bytes of the CStencil-derived class.

pHandle

[out] The optional address of a cache item handle that can later be used to access the new cache entry. If no handle is retrieved, the item can still be accessed by passing the name of the stencil to LookupStencil.

hInst

The handle of a DLL upon which the stencil depends. This value is used to ensure that the DLL is not unloaded which the stencil is cached. Typically, this is the handle to the request handler DLL used by the stencil.

pClient

The address of an IMemoryCacheClient implementation that will free the stencil data when it is removed from the cache. Typically, this will be the same pointer as *pStencil* when a CStencil-derived class is being added.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If *pHandle* is not NULL, the initial reference count for the item is set to 1. See AddRefStencil and ReleaseStencil for more details on reference counting.

See Also

Reference

IStencilCache Interface

Other Resources

IStencilCache Members

IStencilCache::GetStencil

Call this method to get a pointer to a stencil in the cache.

```
STDMETHOD(GetStencil)(
   const HCACHEITEM hStencil,
   void ** ppStencil
) const;
```

Parameters

hStencil

The handle of the desired stencil, as obtained with a previous call to CacheStencil or LookupStencil.

ppStencil

[out] Address of the pointer that, on success, receives the pointer to the stencil object.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

IStencilCache Interface

Other Resources

IStencilCache Members

IStencilCache::LookupStencil

Call this method to get the handle of a cached stencil.

```
STDMETHOD(LookupStencil)(
   LPCSTR szName,
   HCACHEITEM * phStencil
);
```

Parameters

szName

A string containing the name provided to CacheStencil when the stencil was added to the cache.

phStencil

[out] Address of the HCACHEITEM variable that, on success, receives the handle of the requested item.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method returns a handle to the requested item and increments the reference count on that item. The handle and reference count should be returned to the cache by a subsequent call to ReleaseStencil when the item is no longer needed.

See AddRefStencil for more details on reference counting.

See Also

Reference

IStencilCache Interface

Other Resources

IStencilCache Members

IStencilCache::ReleaseStencil

Call this method to decrement the reference count of a cached stencil.

```
STDMETHOD(ReleaseStencil)(
   const HCACHEITEM hStencil
);
```

Parameters

hStencil

The handle of the item whose reference count should be decremented.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

Every call to LookupStencil and AddRefStencil should have a matching call to this method when the handle is no longer being used so that the cache has an opportunity to remove the corresponding item. Mismatched calls to these methods will result in leaks or premature removal of cached items.

See Also

Reference

IStencilCache Interface

Other Resources

IStencilCache Members

IStencilCacheControl Interface

This interface provides methods for controlling a stencil cache.

```
__interface ATL_NO_VTABLE
__declspec(uuid("55DEF119-D7A7-4eb7-A876-33365E1C5E1A"))
IStencilCacheControl :
   public IUnknown
```

Remarks

This interface is implemented by CStencilCache.

Requirements

Header: atlcache.h

See Also

Tasks

StencilCache Sample: Uses the ATL Server Stencil Cache

Hotswap Sample: Replacing ATL Server Application DLLs at Run Time

Other Resources

Caching

IStencilCacheControl Members

Caching Reference

IStencilCacheControl Members

Methods

GetDefaultLifespan	Call this method to get the default life span of a stencil in the cache.
RemoveAllStencils	Call this method to remove all the stencils from the stencil cache.
RemoveStencil	Call this method to remove a stencil from the stencil cache given its handle.
RemoveStencilByName	Call this method to remove a stencil from the stencil cache given its name.
SetDefaultLifespan	Call this method to set the default life span of a stencil in the cache.

See Also

Reference

IStencilCacheControl Interface

Other Resources

Caching

IStencilCacheControl Methods

For information about the methods in **IStencilCacheControl**, see IStencilCacheControl Members.

IStencilCacheControl::GetDefaultLifespan

Call this method to get the default life span of a stencil in the cache in 100-nanosecond intervals.

```
STDMETHOD(GetDefaultLifespan)(
   unsigned __int64 * pdwdwLifespan
);
```

Parameters

pdwdwLifespan

[out] Address of the variable that, on success, receives the default life span of a stencil in the cache in 100-nanosecond intervals.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

See the StencilCache Sample.

See Also

Reference

IStencilCacheControl Interface

IStencilCacheControl::SetDefaultLifespan

Other Resources

Caching

IStencilCacheControl Members

IStencilCacheControl::RemoveAllStencils

Call this method to remove all the stencils from the stencil cache.

```
STDMETHOD(RemoveAllStencils)( );
```

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

See the Hotswap Sample.

See Also

Reference

IStencilCacheControl Interface

Other Resources

Caching

IStencilCacheControl Members

IStencilCacheControl::RemoveStencil

Call this method to remove a stencil from the stencil cache given its handle.

```
STDMETHOD(RemoveStencil)(
   const HCACHEITEM hStencil
);
```

Parameters

hStencil

The handle of the stencil to be removed.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

See Also

Reference

IStencilCacheControl Interface

IStencilCacheControl::RemoveStencilByName

Other Resources

Caching

IStencilCacheControl Members

IStencilCacheControl::RemoveStencilByName

Call this method to remove a stencil from the stencil cache given its name.

```
STDMETHOD(RemoveStencilByName)(
   LPCSTR szStencil
);
```

Parameters

szStencil

The name of the stencil to be removed.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

See Also

Reference

IStencilCacheControl Interface

IStencilCacheControl::RemoveStencil

Other Resources

Caching

IStencilCacheControl Members

IStencilCacheControl::SetDefaultLifespan

Call this method to set the default life span of a stencil in the cache in 100-nanosecond intervals.

```
STDMETHOD(SetDefaultLifespan)(
   unsigned __int64 dwdwLifespan
);
```

Parameters

dwdwLifespan

The default life span of a stencil in the cache in 100-nanosecond intervals.

Return Value

Returns **S_OK** on success, or an error **HRESULT** on failure.

Example

See the StencilCache Sample.

See Also

Reference

IStencilCacheControl Interface

IStencilCacheControl::GetDefaultLifespan

Other Resources

Caching

IStencilCacheControl Members

IStencilCacheMgr Interface

This interface provides methods for managing a stencil cache.

```
[ uuid( "3813895C-4C4C-41df-95F4-12220140B164" ), object ]
__interface IStencilCacheMgr
```

Remarks

This interface is implemented by CStencilCacheManager.

Requirements

Header: atlextmgmt.h

See Also

Other Resources

IStencilCacheMgr Members

IStencilCacheMgr Members

Methods

ClearStats	Call this method to reset all the statistics to zero.
GetCurrentAllocSize	Call this method to get the current number of bytes used by the cache for storing data.
GetCurrentEntryCount	Call this method to get the current number of items in the cache.
GetDefaultLifespan	Call this method to get the default life span of a stencil in the cache.
GetHitCount	Call this method to get the number of times the cache successfully returned an item.
GetMaxAllocSize	Call this method to get the maximum number of bytes used by the cache at one time since statistics gathering was started.
GetMaxEntryCount	Call this method to get the maximum number of items in the cache at one time since statistics gathering was started.
GetMissCount	Call this method to get the number of times the cache did not contain a requested item.
RemoveAllStencils	Call this method to remove all the stencils from the stencil cache.
RemoveStencil	Call this method to remove a stencil from the stencil cache given its handle.
RemoveStencilByName	Call this method to remove a stencil from the stencil cache given its name.
SetDefaultLifespan	Call this method to set the default life span of a stencil in the cache.

See Also

Reference

IStencilCacheMgr Interface

IStencilCacheMgr Methods

For information about the methods in **IStencilCacheMgr**, see IStencilCacheMgr Members

IStencilCacheMgr::ClearStats

Call this method to reset all the statistics to zero.

```
[id( 7 )]
STDMETHOD(ClearStats)( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

The object implementing this interface keeps a running total of the statistics; it doesn't query the cache for the current information. Calling this method clears *all* the statistics even those with **Current** in their name.

See Also

Reference

IStencilCacheMgr Interface

Other Resources

IStencilCacheMgr Members

IStencilCacheMgr::GetCurrentAllocSize

Call this method to get the current number of bytes used by the cache for storing data.

```
[id( 3 )]
STDMETHOD(GetCurrentAllocSize)(
   [out, retval] __int64 * pdwSize
);
```

Parameters

pdwSize

[out] Address of the variable that, on success, receives the current number of bytes used by the cache for storing data.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

IStencilCacheMgr Interface

IStencilCacheMgr::GetMaxAllocSize

Other Resources

IStencilCacheMgr Members

IStencilCacheMgr::GetCurrentEntryCount

Call this method to get the current number of items in the cache.

```
[id( 0 )]
STDMETHOD(GetCurrentEntryCount)(
   [out, retval] __int64 * pdwSize
);
```

Parameters

pdwSize

[out] Address of the variable that, on success, receives the current number of items in the cache.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

IStencilCacheMgr Interface

IStencilCacheMgr::GetMaxEntryCount

Other Resources

IStencilCacheMgr Members

IStencilCacheMgr::GetDefaultLifespan

Call this method to get the default life span of a stencil in the cache in milliseconds.

```
[id( 6 )]
STDMETHOD(GetDefaultLifespan)(
   [out, retval] unsigned __int64 * pdwdwLifespan
);
```

Parameters

pdwdwLifespan

[out] Address of the variable that, on success, receives the default life span of a stencil in the cache in milliseconds.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

IStencilCacheMgr Interface

IStencilCacheMgr::SetDefaultLifespan

Other Resources

IStencilCacheMgr Members

IStencilCacheMgr::GetHitCount

Call this method to get the number of times the cache successfully returned an item.

```
[id( 1 )]
STDMETHOD(GetHitCount)(
   [out, retval] __int64 * pdwSize
);
```

Parameters

pdwSize

[out] Address of the variable that, on success, receives the number of times the cache successfully returned an item.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

IStencilCacheMgr Interface

IStencilCacheMgr::GetMissCount

Other Resources

IStencilCacheMgr Members

IStencilCacheMgr::GetMaxAllocSize

Call this method to get the maximum number of bytes used by the cache at one time since statistics gathering was started.

```
[id( 4 )]
STDMETHOD(GetMaxAllocSize)(
   [out, retval] __int64 * pdwSize
);
```

Parameters

pdwSize

[out] Address of the variable that, on success, receives the maximum number of bytes allocated by the cache.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

IStencilCacheMgr Interface

IStencilCacheMgr::GetCurrentAllocSize

Other Resources

IStencilCacheMgr Members

IStencilCacheMgr::GetMaxEntryCount

Call this method to get the maximum number of items in the cache at one time since statistics gathering was started.

```
[id( 5 )]
STDMETHOD(GetMaxEntryCount)(
   [out, retval] __int64 * pdwSize
);
```

Parameters

pdwSize

[out] Address of the variable that, on success, receives the maximum number of items in the cache.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

IStencilCacheMgr Interface

IStencilCacheMgr::GetCurrentEntryCount

Other Resources

IStencilCacheMgr Members

IStencilCacheMgr::GetMissCount

Call this method to get the number of times the cache did not contain a requested item.

```
[id( 2 )]
STDMETHOD(GetMissCount)(
   [out, retval] __int64 * pdwSize
);
```

Parameters

pdwSize

[out] Address of the variable that, on success, receives the number of times the cache successfully returned an item.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

IStencilCacheMgr Interface

IStencilCacheMgr::GetHitCount

Other Resources

IStencilCacheMgr Members

IStencilCacheMgr::RemoveAllStencils

Call this method to remove all the stencils from the stencil cache.

```
[id( 10 )]
STDMETHOD(RemoveAllStencils)( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

IStencilCacheMgr Interface

Other Resources

IStencilCacheMgr Members

IStencilCacheMgr::RemoveStencil

Call this method to remove a stencil from the stencil cache given its handle.

```
[id( 8 )]
STDMETHOD(RemoveStencil)(
   [in] __int64 hStencil
);
```

Parameters

hStencil

The handle of the stencil to be removed.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

IStencilCacheMgr Interface

IStencilCacheMgr::RemoveStencilByName

Other Resources

IStencilCacheMgr Members

IStencilCacheMgr::RemoveStencilByName

Call this method to remove a stencil from the stencil cache given its name.

```
[id( 9 )]
STDMETHOD(RemoveStencilByName)(
   [in] BSTR szStencil
);
```

Parameters

szStencil

The name of the stencil to be removed.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

IStencilCacheMgr Interface

IStencilCacheMgr::RemoveStencil

Other Resources

IStencilCacheMgr Members

IStencilCacheMgr::SetDefaultLifespan

Call this method to set the default life span of a stencil in the cache in milliseconds.

```
[id( 11 )]
STDMETHOD(SetDefaultLifespan)(
   [in] unsigned __int64 dwdwLifespan
);
```

Parameters

dwdwLifespan

The default life span of a stencil in the cache in milliseconds.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

IStencilCacheMgr Interface

IStencilCacheMgr::GetDefaultLifespan

Other Resources

IStencilCacheMgr Members

IStreamImpl Class

This class provides a non-functional implementation of IStream that can be used as a base class by classes that need to provide the binary interface of **IStream** but don't need usable implementations for all the methods.

```
class IStreamImpl :
   public IStream
```

Remarks

All the methods implemented by this class return E_NOTIMPL. Derived classes can override only those methods that clients are expected to use.

Requirements

Header: atlsoap.h

See Also

Reference

IStream

Other Resources

IStreamImpl Members

IStreamImpl Members

Methods

Clone	Non-functional implementation of IStream::Clone.
Commit	Non-functional implementation of IStream::Commit.
CopyTo	Non-functional implementation of IStream::CopyTo.
LockRegion	Non-functional implementation of IStream::LockRegion.
Read	Non-functional implementation of ISequentialStream::Read.
Revert	Non-functional implementation of IStream::Revert.
Seek	Non-functional implementation of IStream::Seek.
SetSize	Non-functional implementation of IStream::SetSize.
Stat	Non-functional implementation of IStream::Stat.
UnlockRegion	Non-functional implementation of IStream::UnlockRegion.
Write	Non-functional implementation of ISequentialStream::Write.

See Also

Reference

IStreamImpl Class

IStreamImpl Methods

For information about the methods in **IStreamImpl**, see IStreamImpl Members

IStreamImpl::Clone

Non-functional implementation of IStream::Clone.

```
HRESULT __stdcall Clone(
   IStream ** /* ppstm */
);
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

IStreamImpl Class

Other Resources

IStreamImpl Members

IStreamImpl::Commit

Non-functional implementation of IStream::Commit.

```
HRESULT __stdcall Commit(
   DWORD /* grfCommitFlags */
);
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

IStreamImpl Class

Other Resources

IStreamImpl Members

IStreamImpl::CopyTo

Non-functional implementation of IStream::CopyTo.

```
HRESULT __stdcall CopyTo(
   IStream * /* pStream */,
   ULARGE_INTEGER /* cb */,
   ULARGE_INTEGER * /* pcbRead */,
   ULARGE_INTEGER * /* pcbWritten */
);
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

IStreamImpl Class

Other Resources

IStreamImpl Members

IStreamImpl::LockRegion

Non-functional implementation of IStream::LockRegion.

```
HRESULT __stdcall LockRegion(
   ULARGE_INTEGER /* libOffset */,
   ULARGE_INTEGER /* cb */,
   DWORD /* dwLockType */
);
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

IStreamImpl Class

Other Resources

IStreamImpl Members

IStreamImpl::Read

Non-functional implementation of ISequentialStream::Read.

```
HRESULT __stdcall Read(
   void * /* pDest */,
   ULONG /* nMaxLen */,
   ULONG * /* pnRead */
);
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

IStreamImpl Class

Other Resources

IStreamImpl Members

IStreamImpl::Revert

Non-functional implementation of IStream::Revert.

```
HRESULT __stdcall Revert( );
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

IStreamImpl Class

Other Resources

IStreamImpl Members

IStreamImpl::Seek

Non-functional implementation of IStream::Seek.

```
HRESULT __stdcall Seek(
   LARGE_INTEGER /* dlibMove */,
   DWORD /* dwOrigin */,
   ULARGE_INTEGER * /* pLibNewPosition */
);
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

IStreamImpl Class

Other Resources

IStreamImpl Members

IStreamImpl::SetSize

Non-functional implementation of IStream::SetSize.

```
HRESULT __stdcall SetSize(
   ULARGE_INTEGER /* libNewSize */
);
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

IStreamImpl Class

Other Resources

IStreamImpl Members

IStreamImpl::Stat

Non-functional implementation of IStream::Stat.

```
HRESULT __stdcall Stat(
   STATSTG * /* pstatstg */,
   DWORD /* grfStatFlag */
);
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

IStreamImpl Class

Other Resources

IStreamImpl Members

IStreamImpl::UnlockRegion

Non-functional implementation of IStream::UnlockRegion.

```
HRESULT __stdcall UnlockRegion(
   ULARGE_INTEGER /* libOffset */,
   ULARGE_INTEGER /* cb */,
   DWORD /* dwLockType */
);
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

IStreamImpl Class

Other Resources

IStreamImpl Members

IStreamImpl::Write

Non-functional implementation of ISequentialStream::Write.

```
HRESULT __stdcall Write(
   const void * /* pv */,
   ULONG /* cb */,
   ULONG * /* pcbWritten */
);
```

Return Value

Always returns E_NOTIMPL.

See Also

Reference

IStreamImpl Class

Other Resources

IStreamImpl Members

ITagReplacer Interface

This interface provides the methods necessary for server response file processing. It provides the connection between the *object.method* text of a replacement tag and the object that implements that method.

```
__interface ATL_NO_VTABLE
__declspec( uuid( "8FF5E90C-8CE0-43aa-96C4-3BF930837512" ))
ITagReplacer :
   public IUnknown
```

Remarks

ITagReplacer provides a way of calling a method on an object where the object, method, and arguments are represented as text. ITagReplacer::FindReplacementOffset is used to convert the names of the object and method into numeric identifiers. ITagReplacer::RenderReplacement is used to call a method given the numeric identifiers returned by **FindReplacementOffset**.

Since the objects whose methods are exposed by the **ITagReplacer** interface are assumed to be used to generate a response of some kind, the ITagReplacer::SetStream method is provided to allow the response to be written to a stream.

ITagReplacer::GetContext allows implementations of this interface to provide context information to clients of the interface.

This interface is implemented by ITagReplacerImpl and CHtmlTagReplacer.

Requirements

Header: atlsiface.h

See Also

Other Resources

ITagReplacer Members

ITagReplacer Members

Methods

FindReplacementOffset	Implement this method to return numeric identifiers for a given object and method name. The identifiers can be later passed to ITagReplacer::RenderReplacement.
GetContext	Implement this method to return the context associated with a particular implementation of this interface.
RenderReplacement	Implement this method to call the specified method on the specified object.
SetStream	Implement this method to save the stream and make it available to the methods callable through ITagReplacer::RenderReplacement.

See Also

Reference

ITagReplacer Interface

ITagReplacer Methods

For information about the methods in **ITagReplacer**, see ITagReplacer Members

ITagReplacer::FindReplacementOffset

Implement this method to return numeric identifiers for a given object and method name. The identifiers can be later passed to ITagReplacer::RenderReplacement.

```
HTTP_CODE FindReplacementOffset(
   LPCSTR szMethodName,
   DWORD * pdwMethodOffset,
   LPCSTR szObjectName,
   DWORD * pdwObjOffset,
   DWORD * pdwMap,
   void ** ppvParam,
   IAtlMemMgr * pMemMgr
);
```

Parameters

szMethodName

The name of a method (and optional arguments). Should not be NULL.

pdwMethodOffset

[out] Address of the variable that, on success, receives the numeric identifier corresponding to *szMethodName*.

szObjectName

The name of the object providing the method identified by *szMethodName*. May be NULL.

pdwObjOffset

[out] Address of the variable that, on success, receives the numeric identifier corresponding to *szObjectName*.

pdwMap

[out] Address of the variable that, on success, receives the identifier of the map containing the object/method identified by *szObjectName/szMethodName*.

ppvParam

[out] Address of the pointer variable that, on success, receives a pointer to an object representing the arguments in *szMethodName*.

pMemMgr

The memory manager to be used to allocate the *ppvParam* object.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

The values returned from this method's out parameters can be cached and passed to ITagReplacer::RenderReplacement to make a method call at a later time.

See Also

Reference

ITagReplacer Interface

Other Resources

ITagReplacer Members

ITagReplacer::GetContext

Implement this method to return the context associated with a particular implementation of this interface.

```
HRESULT GetContext(
   REFIID riid,
   void** ppv
);
```

Parameters

riid

Identifier of the interface being requested.

ppv

[out] Address of a pointer variable that receives the interface pointer requested in *riid*.

Return Value

Returns S_OK on success or an error HRESULT on failure.

See Also

Reference

ITagReplacer Interface

Other Resources

ITagReplacer Members

ITagReplacer::RenderReplacement

Implement this method to call the specified method on the specified object.

```
HTTP_CODE RenderReplacement(
   DWORD dwFnOffset,
   DWORD dwObjOffset,
   DWORD dwMap,
   void * pvParam
);
```

Parameters

dwFnOffset

The numeric identifier of a method provided by the object represented by *dwObjOffset*.

dwObjOffset

The numeric identifier of an object in the map represented by *dwMap*.

dwMap

The numeric identifier of a map of objects.

pvParam

A pointer to an object passed as the argument to the method represented by *dwObjOffset*.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

The arguments passed to this method must have previously been returned by a call to ITagReplacer::FindReplacementOffset on the same object.

The stream set by calling ITagReplacer::SetStream must be made available to the methods being called.

See Also

Reference

ITagReplacer Interface
ITagReplacer::FindReplacementOffset
ITagReplacer::SetStream

Other Resources

ITagReplacer Members

ITagReplacer::SetStream

Implement this method to save the stream and make it available to the methods callable through ITagReplacer::RenderReplacement.

```
IWriteStream * SetStream(
   IWriteStream * pStream
);
```

Parameters

pStream

The new stream to be used to write the output of method calls.

Return Value

Returns the previously stored stream.

See Also

Reference

ITagReplacer Interface

Other Resources

ITagReplacer Members

ITagReplacerImpl Class

Implementation of the ITagReplacer interface.

```
template < class T >
class ITagReplacerImpl :
   public ITagReplacer
```

Parameters
T

A class derived from **ITagReplacerImpl**. See **Remarks** for details.

Remarks

ITagReplacerImpl provides a default implementation of the ITagReplacer interface, which is used to provide methods necessary for stencil processing and integration with CStencil and derived classes.

In addition to the **ITagReplacer** methods, **ITagReplacerImpl** provides default parse functions that can be used with the REPLACEMENT_METHOD_ENTRY_EX macro or the tag_name attribute.

This table shows the members expected to be provided by class *T* and the code elements that typically provide their implementation in an ATL Server web application:

Member	Implementation Typically Provided By
GetReplacementMethodMap	BEGIN_REPLACEMENT_METHOD_MAP and related macros
GetAttrReplacementMethodMap	tag_name attribute
GetReplacementObject	CHtmlTagReplacer
GetHandlerOffset	CHtmlTagReplacer

Default implementations of all these members are provided by **ITagReplacerImpl**.

Requirements

Header: atlstencil.h

See Also
Reference
ITagReplacer Interface
Other Resources
ITagReplacerImpl Members

ITagReplacerImpl Members

Methods

DefaultParseBool	A parse function for a single **bool** parameter.
DefaultParseDouble	A parse function for a single **double** parameter.
DefaultParseFloat	A parse function for a single **float** parameter.
DefaultParseInt	A parse function for a single **int** parameter.
DefaultParseInt64	A parse function for a single **__int64** parameter.
DefaultParseShort	A parse function for a single **short** parameter.
DefaultParseString	A parse function for a single **LPCSTR** parameter.
DefaultParseUChar	A parse function for a single **unsigned char** parameter.
DefaultParseUInt	A parse function for a single **unsigned int** parameter.
DefaultParseUInt64	A parse function for a single **unsigned __int64** parameter.
DefaultParseUShort	A parse function for a single **unsigned short** parameter.
FindReplacementOffset	Implementation of ITagReplacer::FindReplacementOffset.
FindReplacementOffsetInMap	Call this method to search an array for the item corresponding to a method name.
GetAttrReplacementMethodMap	Override this non-virtual method to return an array of CTagReplacerMethodEntry objects.
GetContext	Implementation of ITagReplacer::GetContext.
GetHandlerOffset	Override this non-virtual method to return a numeric identifier for an object that can later be passed to ITagReplacerImpl::GetReplacementObject to get the ITagReplacer interface corresp onding to that object.
GetReplacementMethodMap	Override this non-virtual method to return an array of CTagReplacerMethodEntry objects.
GetReplacementObject	Override this non-virtual method to return the ITagReplacer interface for an object whose ide ntifier was retrieved by a call to GetHandlerOffset.
GetResourceInstance	Override this method to return the handle of the module providing resources.
ITagReplacerImpl	The constructor.
RenderReplacement	Implementation of ITagReplacer::RenderReplacement.
SetStream	Implementation of ITagReplacer::SetStream.

Data Members

m_pStream	The stream to which responses generated by the methods called by ITagReplacerImpl::RenderReplacement should be written.

See Also

Reference

ITagReplacerImpl Class

ITagReplacerImpl Methods

For information about the methods in **ITagReplacerImpl**, see ITagReplacerImpl Members

ITagReplacerImpl::DefaultParseBool

A parse function for a single **bool** parameter.

```
HTTP_CODE DefaultParseBool(
   IAtlMemMgr * pMemMgr,
   LPCSTR szParams,
   bool ** ppParam
) throw(...);
```

Parameters

pMemMgr

The memory manager used to allocate memory for the parameter object.

szParams

The string to parse.

ppParam

[out] Address of the pointer variable that, on success, receives the pointer to the parameter object.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating failure.

Remarks

Sets the parameter to **true** if the start of *szParams* is **"true"**. The comparison is case-insensitive.

See Also

Reference

ITagReplacerImpl Class

Concepts

Passing an Argument to a Replacement Method

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::DefaultParseDouble

A parse function for a single **double** parameter.

```
HTTP_CODE DefaultParseDouble(
   IAtlMemMgr * pMemMgr,
   LPCSTR szParams,
   double ** ppParam
) throw(...);
```

Parameters

pMemMgr

The memory manager used to allocate memory for the parameter object.

szParams

The string to parse.

ppParam

[out] Address of the pointer variable that, on success, receives the pointer to the parameter object.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating failure.

Remarks

Converts *szParams* using atof.

See Also

Reference

ITagReplacerImpl Class

Concepts

Passing an Argument to a Replacement Method

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::DefaultParseFloat

A parse function for a single **float** parameter.

```
HTTP_CODE DefaultParseFloat(
   IAtlMemMgr * pMemMgr,
   LPCSTR szParams,
   float ** ppParam
) throw(...);
```

Parameters

pMemMgr

The memory manager used to allocate memory for the parameter object.

szParams

The string to parse.

ppParam

[out] Address of the pointer variable that, on success, receives the pointer to the parameter object.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating failure.

Remarks

Converts *szParams* using atof.

See Also

Reference

ITagReplacerImpl Class

Concepts

Passing an Argument to a Replacement Method

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::DefaultParseInt

A parse function for a single **int** parameter.

```
HTTP_CODE DefaultParseInt(
   IAtlMemMgr * pMemMgr,
   LPCSTR szParams,
   int ** ppParam
) throw(...);
```

Parameters

pMemMgr

The memory manager used to allocate memory for the parameter object.

szParams

The string to parse.

ppParam

[out] Address of the pointer variable that, on success, receives the pointer to the parameter object.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating failure.

Remarks

Converts *szParams* using atoi.

See Also

Reference

ITagReplacerImpl Class

Concepts

Passing an Argument to a Replacement Method

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::DefaultParseInt64

A parse function for a single **__int64** parameter.

```
HTTP_CODE DefaultParseInt64(
   IAtlMemMgr * pMemMgr,
   LPCSTR szParams,
   __int64 ** ppParam
) throw(...);
```

Parameters

pMemMgr

The memory manager used to allocate memory for the parameter object.

szParams

The string to parse.

ppParam

[out] Address of the pointer variable that, on success, receives the pointer to the parameter object.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating failure.

Remarks

Converts *szParams* using _atoi64.

See Also

Reference

ITagReplacerImpl Class

Concepts

Passing an Argument to a Replacement Method

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::DefaultParseShort

A parse function for a single **short** parameter.

```
HTTP_CODE DefaultParseShort(
   IAtlMemMgr * pMemMgr,
   LPCSTR szParams,
   short ** ppParam
) throw(...);
```

Parameters

pMemMgr

The memory manager used to allocate memory for the parameter object.

szParams

The string to parse.

ppParam

[out] Address of the pointer variable that, on success, receives the pointer to the parameter object.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating failure.

Remarks

Converts *szParams* using atoi.

See Also

Reference

ITagReplacerImpl Class

Concepts

Passing an Argument to a Replacement Method

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::DefaultParseString

A parse function for a single **LPCSTR** parameter.

```
HTTP_CODE DefaultParseString(
   IAtlMemMgr * pMemMgr,
   LPCSTR szParams,
   char ** ppParam
) throw(...);
```

Parameters

pMemMgr

The memory manager used to allocate memory for the parameter object.

szParams

The string to parse.

ppParam

[out] Address of the pointer variable that, on success, receives the pointer to the parameter object.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating failure.

Remarks

The string in *szParams* is passed unchanged.

See Also

Reference

ITagReplacerImpl Class

Concepts

Passing an Argument to a Replacement Method

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::DefaultParseUChar

A parse function for a single **unsigned char** parameter.

```
HTTP_CODE DefaultParseUChar(
   IAtlMemMgr * pMemMgr,
   LPCSTR szParams,
   unsigned char ** ppParam
) throw(...);
```

Parameters

pMemMgr

The memory manager used to allocate memory for the parameter object.

szParams

The string to parse.

ppParam

[out] Address of the pointer variable that, on success, receives the pointer to the parameter object.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating failure.

Remarks

Converts *szParams* using strtoul.

See Also

Reference

ITagReplacerImpl Class

Concepts

Passing an Argument to a Replacement Method

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::DefaultParseUInt

A parse function for a single **unsigned int** parameter.

```
HTTP_CODE DefaultParseUInt(
   IAtlMemMgr * pMemMgr,
   LPCSTR szParams,
   unsigned int ** ppParam
) throw(...);
```

Parameters

pMemMgr

The memory manager used to allocate memory for the parameter object.

szParams

The string to parse.

ppParam

[out] Address of the pointer variable that, on success, receives the pointer to the parameter object.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating failure.

Remarks

Converts *szParams* using strtoul.

See Also

Reference

ITagReplacerImpl Class

Concepts

Passing an Argument to a Replacement Method

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::DefaultParseUInt64

A parse function for a single **unsigned __int64** parameter.

```
HTTP_CODE DefaultParseUInt64(
   IAtlMemMgr * pMemMgr,
   LPCSTR szParams,
   unsigned __int64 ** ppParam
) throw(...);
```

Parameters

pMemMgr

The memory manager used to allocate memory for the parameter object.

szParams

The string to parse.

ppParam

[out] Address of the pointer variable that, on success, receives the pointer to the parameter object.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating failure.

Remarks

Converts *szParams* using _strtoui64.

See Also

Reference

ITagReplacerImpl Class

Concepts

Passing an Argument to a Replacement Method

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::DefaultParseUShort

A parse function for a single **unsigned short** parameter.

```
HTTP_CODE DefaultParseUShort(
   IAtlMemMgr * pMemMgr,
   LPCSTR szParams,
   unsigned short ** ppParam
) throw(...);
```

Parameters

pMemMgr

The memory manager used to allocate memory for the parameter object.

szParams

The string to parse.

ppParam

[out] Address of the pointer variable that, on success, receives the pointer to the parameter object.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating failure.

Remarks

Converts *szParams* using strtoul.

See Also

Reference

ITagReplacerImpl Class

Concepts

Passing an Argument to a Replacement Method

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::FindReplacementOffset

Implementation of ITagReplacer::FindReplacementOffset.

```
HTTP_CODE FindReplacementOffset(
   LPCSTR szMethodName,
   DWORD * pdwMethodOffset,
   LPCSTR szHandlerName,
   DWORD * pdwHandlerOffset,
   DWORD * pdwMap,
   void ** ppvParam,
   IAtlMemMgr * pMemMgr
);
```

Remarks

If an object name is specified, this implementation delegates to the **FindReplacementOffset** method on the ITagReplacer interface of that object. The interface is retrieved by calling GetHandlerOffset followed by GetReplacementObject. If either method fails, this method returns a failure.

If no object name is provided, this implementation attempts to find the method in one of two maps provided by this object. First, arguments are removed from the method name (by searching for opening and closing parentheses); then, the implementation calls FindReplacementOffsetInMap on the map returned by GetReplacementMethodMap. If that fails, this method calls FindReplacementOffsetInMap on the map returned by GetAttrReplacementMethodMap.

The first map is represented by the STENCIL_BASIC_MAP constant; the second by STENCIL_ATTR_MAP.

If the method is found in one of these maps and arguments were passed as part of *szMethodName*, the associated parse function is called to create the object returned via *ppvParam*.

See Also

Reference

ITagReplacerImpl Class

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::FindReplacementOffsetInMap

Call this method to search an array for the item corresponding to a method name.

```
HTTP_CODE FindReplacementOffsetInMap(
   LPCSTR szMethodName,
   LPDWORD pdwMethodOffset,
   const CTagReplacerMethodEntry< T > * pEntry
) throw( );
```

Parameters

szMethodName

The name of the method to search for.

pdwMethodOffset

[out] Address of the variable that, on success, receives the offset from *pEntry* that corresponds to *szMethodName*.

pEntry

The array of CTagReplacerMethodEntry objects to search.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

The comparison is case-sensitive.

See Also

Reference

ITagReplacerImpl Class

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::GetAttrReplacementMethodMap

Override this non-virtual method to return an array of CTagReplacerMethodEntry objects.

```
void GetAttrReplacementMethodMap(
   const CTagReplacerMethodEntry< T > ** ppOut
) const;
```

Parameters

ppOut

[out] Address of the pointer variable that, on success, receives the address of the first member of the array of CTagReplacerMethodEntry objects.

Remarks

This method is typically overridden to return the array of entries added by the tag_name attribute. The default implementation provided by this class simply returns an array containing a single null entry.

See Also

Reference

ITagReplacerImpl Class

CTagReplacerMethodEntry Structure

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::GetContext

Implementation of ITagReplacer::GetContext.

```
HRESULT GetContext(
   REFIID,
   void**
);
```

Remarks

The default implementation provided by this class always returns E_NOINTERFACE.

See Also

Reference

ITagReplacerImpl Class

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::GetHandlerOffset

Override this non-virtual method to return a numeric identifier for an object that can later be passed to ITagReplacerImpl::GetReplacementObject to get the ITagReplacer interface corresponding to that object.

```
HTTP_CODE GetHandlerOffset(
   LPCSTR /* szHandlerName */,
   DWORD* pdwOffset
);
```

Parameters

szHandlerName

The name of the object whose ID is to be returned in *pdwOffset*.

pdwOffset

[out] Address of the variable that, on success, receives the numeric identifier that corresponds to *szHandlerName*.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

ITagReplacerImpl provides a non-functional implementation that always returns HTTP_FAIL. A useful implementation of this method is typically provided by CHtmlTagReplacer.

See Also

Reference

ITagReplacerImpl Class
ITagReplacerImpl::GetReplacementObject

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::GetReplacementMethodMap

Override this non-virtual method to return an array of CTagReplacerMethodEntry objects.

```
void GetReplacementMethodMap(
   const CTagReplacerMethodEntry< T > ** ppOut
) const;
```

Parameters

ppOut

[out] Address of the pointer variable that, on success, receives the address of the first member of the array of CTagReplacerMethodEntry objects.

Remarks

This method is typically overridden to return the array of entries added by BEGIN_REPLACEMENT_METHOD_MAP and related macros. The default implementation provided by this class simply returns an array containing a single null entry.

See Also

Reference

ITagReplacerImpl Class

CTagReplacerMethodEntry Structure

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::GetReplacementObject

Override this non-virtual method to return the ITagReplacer interface for an object whose identifier was retrieved by a call to GetHandlerOffset.

```
HTTP_CODE GetReplacementObject(
   DWORD /* dwObjOffset */,
   ITagReplacer ** ppReplacer
);
```

Parameters

dwObjOffset

The numeric identifier of an object.

ppReplacer

[out] Address of the pointer variable that, on success, receives the ITagReplacer interface pointer of the object corresponding to *dwObjOffset*.

Return Value

Returns HTTP_SUCCESS or another ATL Server status code indicating success or failure.

Remarks

ITagReplacerImpl provides a non-functional implementation that always returns HTTP_FAIL. A useful implementation of this method is typically provided by CHtmlTagReplacer.

See Also

Reference

ITagReplacerImpl Class

ITagReplacerImpl::GetHandlerOffset

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::GetResourceInstance

Override this method to return the handle of the module providing resources.

```
virtual HINSTANCE GetResourceInstance( );
```

Return Value

Returns the handle of the module providing resources.

Remarks

The default implementation of this method returns the handle to the file used to create the calling process.

See Also

Reference

ITagReplacerImpl Class

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::ITagReplacerImpl

The constructor.

```
ITagReplacerImpl( ) throw( );
```

See Also

Reference

ITagReplacerImpl Class

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::RenderReplacement

Implementation of ITagReplacer::RenderReplacement.

```
HTTP_CODE RenderReplacement(
   DWORD dwFnOffset,
   DWORD dwObjOffset,
   DWORD dwMap,
   void * pvParam
);
```

Remarks

If an object offset is specified, this implementation delegates to the **RenderReplacement** method on the ITagReplacer interface of that object. The interface is retrieved by calling GetReplacementObject; its SetStream method is called to ensure that output from all objects goes to the same destination.

If no object offset is specified, this implementation retrieves the entry from the appropriate map and calls the method, passing *pvParam* as necessary.

See Also

Reference

ITagReplacerImpl Class

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl::SetStream

Implementation of ITagReplacer::SetStream.

```
IWriteStream * SetStream(
   IWriteStream * pStream
);
```

Remarks

The pointer to the stream is stored in ITagReplacerImpl::m_pStream.

See Also

Reference

ITagReplacerImpl Class
ITagReplacer::SetStream
ITagReplacerImpl::m_pStream

Other Resources

ITagReplacerImpl Members

ITagReplacerImpl Data Members

For information about the data members in **ITagReplacerImpl**, see ITagReplacerImpl Members

ITagReplacerImpl::m_pStream

The stream to which responses generated by the methods called by ITagReplacerImpl::RenderReplacement should be written.

```
IWriteStream * m_pStream;
```

Remarks

Set to NULL in the constructor. Changed by calls to ITagReplacerImpl::SetStream.

See Also

Reference

ITagReplacerImpl Class
ITagReplacerImpl::RenderReplacement
ITagReplacerImpl::SetStream

Other Resources

ITagReplacerImpl Members

IThreadPoolConfig Interface

This interface provides methods for configuring a thread pool.

```
__interface
__declspec(uuid("B1F64757-6E88-4fa2-8886-7848B0D7E660"))
IThreadPoolConfig :
   public IUnknown
```

Remarks

This interface is implemented by CThreadPool.

Requirements

Header: atlsiface.h

See Also

Reference

CThreadPool Class

Other Resources

IThreadPoolConfig Members

ATL Server Classes

IThreadPoolConfig Members

Methods

GetSize	Call this method to get the number of threads in the pool.
GetTimeout	Call this method to get the maximum time in milliseconds that the thread pool will wait for a thread to shut down.
SetSize	Call this method to set the number of threads in the pool.
SetTimeout	Call this method to set the maximum time in milliseconds that the thread pool will wait for a thread to shut down.

See Also

Reference

IThreadPoolConfig Interface

Other Resources

CThreadPool Members

IThreadPoolConfig Methods

For information about the methods in **IThreadPoolConfig**, see IThreadPoolConfig Members.

IThreadPoolConfig::GetSize

Call this method to get the number of threads in the pool.

```
STDMETHOD(GetSize)(
   int * pnNumThreads
);
```

Parameters

pnNumThreads

[out] Address of the variable that, on success, receives the number of threads in the pool.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

```
HRESULT DoPoolOperations(IThreadPoolConfig* pPool, int nSize)
{
    int nCurrSize = 0;
    HRESULT hr = pPool->GetSize(&nCurrSize);
    if (SUCCEEDED(hr))
    {
        printf_s("Current pool size: %d\n", nCurrSize);
        hr = pPool->SetSize(nSize);
        if (SUCCEEDED(hr))
        {
            printf_s("New pool size : %d\n", nSize);
            DWORD dwTimeout = 0;
            hr = pPool->GetTimeout(&dwTimeout);
            if (SUCCEEDED(hr))
            {
                printf_s("Current pool timeout: %u\n", dwTimeout);
                // Increase timeout by 10 seconds.
                dwTimeout += 10 * 1000;
                hr = pPool->SetTimeout(dwTimeout);
                if (SUCCEEDED(hr))
                {
                    printf_s("New pool timeout: %u\n", dwTimeout);
                }
                else
                {
                    printf_s("Failed to set pool timeout: 0x%08X\n",
                             hr);
                }
            }
            else
            {
                printf_s("Failed to get pool timeout: 0x%08X\n", hr);
            }
        }
        else
        {
            printf_s("Failed to resize pool: 0x%08X\n", hr);
        }
    }
    else
    {
        printf_s("Failed to get pool size: 0x%08x\n", hr);
    }

    return hr;
}
```

See Also

Reference

IThreadPoolConfig Interface

IThreadPoolConfig::SetSize

Other Resources

IThreadPoolConfig Members

IThreadPoolConfig::GetTimeout

Call this method to get the maximum time in milliseconds that the thread pool will wait for a thread to shut down.

```
STDMETHOD(GetTimeout)(
   DWORD * pdwMaxWait
);
```

Parameters

pdwMaxWait

[out] Address of the variable that, on success, receives the maximum time in milliseconds that the thread pool will wait for a thread to shut down.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See IThreadPoolConfig::GetSize.

See Also

Reference

IThreadPoolConfig Interface

IThreadPoolConfig::SetTimeout

Other Resources

IThreadPoolConfig Members

IThreadPoolConfig::SetSize

Call this method to set the number of threads in the pool.

```
STDMETHOD(SetSize)(
   int nNumThreads
);
```

Parameters

nNumThreads

The requested number of threads in the pool.

If *nNumThreads* is negative, its absolute value will be multiplied by the number of processors in the machine to get the total number of threads.

If *nNumThreads* is zero, ATLS_DEFAULT_THREADSPERPROC will be multiplied by the number of processors in the machine to get the total number of threads.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See IThreadPoolConfig::GetSize.

See Also

Reference

IThreadPoolConfig Interface
IThreadPoolConfig::GetSize

Other Resources

IThreadPoolConfig Members

IThreadPoolConfig::SetTimeout

Call this method to set the maximum time in milliseconds that the thread pool will wait for a thread to shut down.

```
STDMETHOD(SetTimeout)(
   DWORD dwMaxWait
);
```

Parameters

dwMaxWait

The requested maximum time in milliseconds that the thread pool will wait for a thread to shut down.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See IThreadPoolConfig::GetSize.

See Also

Reference

IThreadPoolConfig Interface

IThreadPoolConfig::GetTimeout

Other Resources

IThreadPoolConfig Members

IThreadPoolMgr Interface

This interface provides methods for managing a thread pool.

```
[ uuid("44e9962a-5207-4d2a-a466-5f08a76e0e5d"), object ]
__interface IThreadPoolMgr
```

Remarks

This interface is implemented by CThreadPoolManager.

Requirements

Header: atlextmgmt.h

Example

```
// Include SPROXY-generated header for
// thread pool manager web service.
#include "ThreadPoolManager.h"

int _tmain(int argc, _TCHAR* argv[])
{
    // Initialize COM
    CComInit cominit;

    using namespace ThreadPoolManager;

    HRESULT hr = S_OK;
    CThreadPoolManager svc;
    if (argc > 1)
    {
        // Use user-specified URL if provided.
        hr = svc.SetUrl(argv[1]);
        if (FAILED(hr))
        {
            printf_s("Failed to set url : 0x%08X\n", hr);
            return 1;
        }
    }

    // Set the NTLM authentication object
    // by calling CAtlHttpClientT::AddAuthObj.
    CNTLMAuthObject ntlm;
    if (!svc.m_socket.AddAuthObj("NTLM", &ntlm))
    {
        printf_s("Failed to set the NTLM authentication object\n");
        return 1;
    }

    // Get the number of threads.
    int nCurrThreads = 0;
    hr = svc.GetSize(&nCurrThreads);
    if (FAILED(hr))
    {
        printf_s("Failed to get the thread pool size : 0x%08X\n", hr);
        return 1;
    }

    // Increase the pool size by 10.
    nCurrThreads += 10;
    hr = svc.SetSize(nCurrThreads);
    if (FAILED(hr))
    {
        printf_s("Failed to set the thread pool size : 0x%08X\n", hr);
```

```
        return 1;
    }

    return 0;
}
```

See Also

Reference

CThreadPoolManager Class

CThreadPool Class

Other Resources

IThreadPoolMgr Members

ATL Server Classes

IThreadPoolMgr Members

Methods

GetSize	Call this method to get the number of threads in the pool.
SetSize	Call this method to set the number of threads in the pool.

See Also

Reference

IThreadPoolMgr Interface

Other Resources

CThreadPoolManager Members

IThreadPoolMgr Methods

For information about the methods in **IThreadPoolMgr**, see IThreadPoolMgr Members.

IThreadPoolMgr::GetSize

Call this method to get the number of threads in the pool.

```
[id(1)]
STDMETHOD(GetSize)(
   [out, retval] int* pnNumThreads
);
```

Parameters

pnNumThreads

[out, retval] Address of the variable that, on success, receives the number of threads in the pool.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See IThreadPoolMgr Overview.

See Also

Reference

IThreadPoolMgr Interface

IThreadPoolMgr::SetSize

Other Resources

IThreadPoolMgr Members

IThreadPoolMgr::SetSize

Call this method to set the number of threads in the pool.

```
[id(0)]
STDMETHOD(SetSize)(
   [in] int nNumThreads
);
```

Parameters

nNumThreads

The requested number of threads in the pool.

If *nNumThreads* is negative, its absolute value will be multiplied by the number of processors in the machine to get the total number of threads.

If *nNumThreads* is zero, ATLS_DEFAULT_THREADSPERPROC will be multiplied by the number of processors in the machine to get the total number of threads.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See IThreadPoolMgr Overview.

See Also

Reference

IThreadPoolMgr Interface

IThreadPoolMgr::GetSize

Other Resources

IThreadPoolMgr Members

IWorkerThreadClient Interface

IWorkerThreadClient is the interface implemented by clients of the CWorkerThread class.

```
__interface IWorkerThreadClient
```

Remarks

Implement this interface when you have code that needs to execute on a worker thread in response to a handle becoming signaled.

This interface is implemented by CBlobCache, CStencilCache, CDllCache, and CFileCache. All of these classes execute code as a result of a timer handle becoming signaled.

Requirements

Header: atlsiface.h

See Also

Reference

CWorkerThread Class

Other Resources

IWorkerThreadClient Members

ATL Server Classes

IWorkerThreadClient Members

Methods

CloseHandle	Implement this method to close the handle associated with this object.
Execute	Implement this method to execute code when the handle associated with this object becomes signaled.

See Also

Reference

IWorkerThreadClient Interface

IWorkerThreadClient Methods

For information about the methods in **IWorkerThreadClient**, see IWorkerThreadClient Members.

IWorkerThreadClient::CloseHandle

Implement this method to close the handle associated with this object.

```
HRESULT CloseHandle(
   HANDLE hHandle
);
```

Parameters

hHandle

The handle to be closed.

Return Value

Return S_OK on success, or an error HRESULT on failure.

Remarks

The handle passed to this method was previously associated with this object by a call to CWorkerThread::AddHandle.

Example

The following code shows a simple implementation of **IWorkerThreadClient::CloseHandle**.

```
HRESULT CloseHandle(HANDLE hObject)
{
    // Users should do any shutdown operation required here.
    // Generally, this means just closing the handle.

    if (!::CloseHandle(hObject))
    {
        // Closing the handle failed for some reason.
        return AtlHresultFromLastError();
    }

    return S_OK;
}
```

See Also

Reference

IWorkerThreadClient Interface

CWorkerThread::AddHandle

Other Resources

IWorkerThreadClient Members

IWorkerThreadClient::Execute

Implement this method to execute code when the handle associated with this object becomes signaled.

```
HRESULT Execute(
   DWORD_PTR dwParam,
   HANDLE hObject
);
```

Parameters

dwParam

The user parameter.

hObject

The handle that has become signaled.

Return Value

Return S_OK on success, or an error HRESULT on failure.

Remarks

The handle and DWORD/pointer passed to this method were previously associated with this object by a call to CWorkerThread::AddHandle.

Example

The following code shows a simple implementation of **IWorkerThreadClient::Execute**.

```
HRESULT Execute(DWORD_PTR dwParam, HANDLE hObject)
{
    // Cast the parameter to its known type.
    LONG* pn = reinterpret_cast<LONG*>(dwParam);

    // Increment the LONG.
    LONG n = InterlockedIncrement(pn);

    // Log the results.
    printf("Handle 0x%08X incremented value to : %d\n",
            hObject, n + 1);

    return S_OK;
}
```

See Also

Reference

IWorkerThreadClient Interface

CWorkerThread::AddHandle

Other Resources

IWorkerThreadClient Members

IWriteStream Interface

This interface provides methods for writing to and flushing a stream buffer.

```
__interface IWriteStream
```

Remarks

This interface is implemented by CHttpResponse. The CWriteStreamHelper class provides type conversion methods that make this interface easier to use.

Example

```
// This class implements the IWriteStream
// interface on top of a Windows file.
class CWriteStreamOnFile :
    public IWriteStream
{
public:

    HRESULT Initialize(LPCSTR szFileName)
    {
        return m_file.Create(szFileName, GENERIC_WRITE,
                             FILE_SHARE_READ, CREATE_ALWAYS);
    }

    HRESULT WriteStream(LPCSTR szOut, int nLen, LPDWORD pdwWritten)
    {
        if (nLen < 0)
        {
            nLen = (int) strlen(szOut);
        }

        DWORD dwWritten = 0;
        HRESULT hr = m_file.Write(szOut, nLen, &dwWritten);
        if (pdwWritten)
        {
            *pdwWritten = dwWritten;
        }

        return hr;
    }

    HRESULT FlushStream()
    {
        if (FlushFileBuffers(m_file))
        {
            return S_OK;
        }
        return AtlHresultFromLastError();
    }

private:
    CAtlFile m_file;
};
```

Requirements

Header: atlsiface.h

See Also

Reference

CHttpResponse Class

CWriteStreamHelper Class

Other Resources

IWriteStream Members

ATL Server Classes

IWriteStream Members

Methods

FlushStream	Call this method to flush the stream.
WriteStream	Call this method to write to the stream.

See Also

Reference

IWriteStream Interface

IWriteStream Methods

For information about the methods in **IWriteStream**, see IWriteStream Members.

IWriteStream::FlushStream

Call this method to flush the stream.

```
HRESULT FlushStream( );
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See IWriteStream Overview.

See Also

Reference

IWriteStream Interface

IWriteStream::WriteStream

Other Resources

IWriteStream Members

IWriteStream::WriteStream

Call this method to write to the stream.

```
HRESULT WriteStream(
   LPCSTR szOut,
   int nLen,
   DWORD * pdwWritten
);
```

Parameters

szOut

A pointer to the start of the data to be written.

nLen

The length in bytes of the data to be written or -1. If *nLen* is -1, *szOut* is assumed to be null-terminated and the entire string will be written to the stream.

pdwWritten

After a successful call, holds the number of bytes written to the stream. Can be NULL if the caller of this method is not interested in this value.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Example

See IWriteStream Overview.

See Also

Reference

IWriteStream Interface

IWriteStream::FlushStream

Other Resources

IWriteStream Members

StencilToken Structure

Use this structure to store information about a token in CStencil or derived classes.

```
struct StencilToken
```

Remarks

CStencil and derived classes parse text and build up a collection of **StencilToken**s describing that text.

Requirements

Header: atlstencil.h

See Also

Other Resources

StencilToken Members

StencilToken Members

Data Members

bDynamicAlloc	TRUE if StencilToken::pStart was dynamically allocated using **new []**. FALSE otherwise.
dwData	Custom data field used to hold extra information related to this token.
dwFnOffset	The index of the replacement method for this token in the map specified by StencilToken::dwMap.
dwLoopIndex	The index of the StencilToken corresponding to the previous or next token in the same control structure.
dwMap	The type of map providing the replacement method specified by StencilToken::dwFnOffset.
dwObjOffset	The index of the object providing the replacement method specified by StencilToken::dwFnOffset.
pEnd	Pointer to the last character of the text corresponding to this object.
pStart	Pointer to the first character of the text corresponding to this object.
szHandlerName	The alias of the subhandler providing the replacement method specified by StencilToken::szMethodName.
szMethodName	The name of the replacement method.
type	A numeric identifier for the type of token.

See Also

Reference

StencilToken Structure

StencilToken Data Members

For information about the data members in **StencilToken**, see StencilToken Members

StencilToken::bDynamicAlloc

TRUE if StencilToken::pStart was dynamically allocated using **new []** and the token owns the data. FALSE otherwise.

```
BOOL bDynamicAlloc;
```

See Also

Reference

StencilToken Structure

Other Resources

StencilToken Members

StencilToken::dwData

Custom data field used to hold extra information related to this token.

```
DWORD_PTR dwData;
```

Remarks

This member is used by the **STENCIL_LOCALE** stencil token to store the specified locale found in a locale tag.

It can also be used for custom data associated with other tokens.

Equivalent to the *ppvParam* parameter of ITagReplacer::FindReplacementOffset.

Equivalent to the *pvParam* parameter of ITagReplacer::RenderReplacement.

See Also

Reference

StencilToken Structure

Other Resources

StencilToken Members

StencilToken::dwFnOffset

The index of the replacement method for this token in the map specified by StencilToken::dwMap.

```
DWORD dwFnOffset;
```

Remarks

Can be **STENCIL_INVALIDOFFSET** if this member is not used or in the event of an error in the stencil.

Equivalent to the *pdwFnOffset* parameter of ITagReplacer::FindReplacementOffset.

Equivalent to the parameter of the same name in ITagReplacer::RenderReplacement.

See Also

Reference

StencilToken Structure

Other Resources

StencilToken Members

StencilToken::dwLoopIndex

The index of the **StencilToken** corresponding to the previous or next token in the same control structure.

```
DWORD dwLoopIndex;
```

Remarks

Can be **STENCIL_INVALIDINDEX** if this member is not used or in the event of an error in the stencil.

Used by **STENCIL_ITERATORSTART**, **STENCIL_ITERATOREND**, **STENCIL_CONDITIONALSTART**, **STENCIL_CONDITIONALELSE**, and **STENCIL_CONDITIONALEND** stencil tokens.

See Also

Reference

StencilToken Structure

Other Resources

StencilToken Members

StencilToken::dwMap

The type of map providing the replacement method specified by StencilToken::dwFnOffset.

```
DWORD dwMap;
```

Remarks

Can be **STENCIL_ATTR_MAP** or **STENCIL_BASIC_MAP**.

Equivalent to the *pdwMap* parameter of ITagReplacer::FindReplacementOffset.

Equivalent to the parameter of the same name in ITagReplacer::RenderReplacement.

See Also

Reference

StencilToken Structure

Other Resources

StencilToken Members

StencilToken::dwObjOffset

The index of the object providing the replacement method specified by StencilToken::dwFnOffset.

```
DWORD dwObjOffset;
```

Remarks

Can be **STENCIL_INVALIDOFFSET** if this member is not used or in the event of an error in the stencil.

Equivalent to the *pdwObjOffset* parameter of ITagReplacer::FindReplacementOffset.

Equivalent to the parameter of the same name in ITagReplacer::RenderReplacement.

See Also

Reference

StencilToken Structure

Other Resources

StencilToken Members

StencilToken::pEnd

Pointer to the last character of the text corresponding to this object.

```
LPCSTR pEnd;
```

See Also

Reference

StencilToken Structure

Other Resources

StencilToken Members

StencilToken::pStart

Pointer to the first character of the text corresponding to this object.

```
LPCSTR pStart;
```

See Also

Reference

StencilToken Structure

StencilToken::bDynamicAlloc

Other Resources

StencilToken Members

StencilToken::szHandlerName

The alias of the subhandler providing the replacement method specified by StencilToken::szMethodName.

```
CHAR szHandlerName[ATL_MAX_HANDLER_NAME_LEN + 1];
```

Remarks

Can be an empty string if this member is not used or in the event of an error in the stencil.

Equivalent to the *szObjectName* parameter of ITagReplacer::FindReplacementOffset.

See Also

Reference

StencilToken Structure

ATL_MAX_HANDLER_NAME_LEN

Other Resources

StencilToken Members

StencilToken::szMethodName

The name of the replacement method.

```
CHAR szMethodName[ATL_MAX_METHOD_NAME_LEN + 1];
```

Remarks

Can be an empty string if this member is not used or in the event of an error in the stencil.

Equivalent to the *szMethodName* parameter of ITagReplacer::FindReplacementOffset.

See Also

Reference

StencilToken Structure

ATL_MAX_METHOD_NAME_LEN

Other Resources

StencilToken Members

StencilToken::type

A numeric identifier for the type of token.

```
DWORD type;
```

Remarks

Can be one of the predefined stencil tokens or a user-defined value greater than STENCIL_USER_TOKEN_BASE.

See Also

Reference

StencilToken Structure
Stencil Tokens
STENCIL_USER_TOKEN_BASE

Other Resources

StencilToken Members

ATL Server Enums

The following enums are included in the ATL Server Library.

Name	Description
ATL_FORM_FLAGS	These flags describe how form processing should be carried out.
ATL_HTML_TAGS	An enumeration of HTML tags.
ATL_MIME_PRIORITY	The members of this enumeration provide constants for the priorities of a MIME message.
ATL_URL_SCHEME	The members of this enumeration provide constants for the schemes understood by CUrl.
ATLSRV_STATE	The members of this enumeration describe the state of an ATL Server request.
ATLSRV_REQUESTTYPE	The members of this enumeration describe the type of an ATL Server request.
REParseError	The members of this enumeration provide values corresponding to errors returned by CAtlRegExp::Parse.
SOAPCLIENT_ERROR	The members of this enumeration provide values corresponding to errors experienced by a SOAP client.
SOAP_ERROR_CODE	The members of this enumeration provide values corresponding to SOAP faultcodes.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

ATL_FORM_FLAGS

These flags describe how form processing should be carried out.

```
enum ATL_FORM_FLAGS{
   ATL_FORM_FLAG_NONE = 0,
   ATL_FORM_FLAG_IGNORE_FILES = 1,
   ATL_FORM_FLAG_REFUSE_FILES = 2,
   ATL_FORM_FLAG_IGNORE_EMPTY_FILES = 4,
   ATL_FORM_FLAG_IGNORE_EMPTY_FIELDS = 8
};
```

Remarks

These flags can be combined and returned from a request handler's FormFlags method or passed directly to CMultiPartFormParser::GetMultiPartData.

Flag	Description
ATL_FORM_FLAG_NONE	Default behavior. Form data is parsed and files are created.
ATL_FORM_FLAG_IGNORE_FILES	Any attempt to upload files is ignored.
ATL_FORM_FLAG_REFUSE_FILES	Any attempt to upload files is treated as a failure.
ATL_FORM_FLAG_IGNORE_EMPTY_FILES	Files with a size of zero bytes are ignored.
ATL_FORM_FLAG_IGNORE_EMPTY_FIELDS	Fields with no content are ignored.

Requirements

Header: atlisapi.h

See Also

Reference

CMultiPartFormParser::GetMultiPartData
CRequestHandlerT::FormFlags

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Enums

ATL_HTML_TAGS

An enumeration of HTML tags.

```
enum ATL_HTML_TAGS{
   ATL_HTML_TAG_BODY,
   ATL_HTML_TAG_A,
   ATL_HTML_TAG_B,
   ATL_HTML_TAG_I,
   ATL_HTML_TAG_U,
   ATL_HTML_TAG_FONT,
   ATL_HTML_TAG_IMG,
   ATL_HTML_TAG_HR,
   ATL_HTML_TAG_BR,
   ATL_HTML_TAG_DIV,
   ATL_HTML_TAG_BLOCKQUOTE,
   ATL_HTML_TAG_ADDRESS,
   ATL_HTML_TAG_P,
   ATL_HTML_TAG_H1,
   ATL_HTML_TAG_H2,
   ATL_HTML_TAG_H3,
   ATL_HTML_TAG_H4,
   ATL_HTML_TAG_H5,
   ATL_HTML_TAG_H6,
   ATL_HTML_TAG_PRE,
   ATL_HTML_TAG_Q,
   ATL_HTML_TAG_SUB,
   ATL_HTML_TAG_SUP,
   ATL_HTML_TAG_INS,
   ATL_HTML_TAG_DEL,
   ATL_HTML_TAG_EM,
   ATL_HTML_TAG_STRONG,
   ATL_HTML_TAG_DFN,
   ATL_HTML_TAG_CODE,
   ATL_HTML_TAG_SAMP,
   ATL_HTML_TAG_KBD,
   ATL_HTML_TAG_VAR,
   ATL_HTML_TAG_CITE,
   ATL_HTML_TAG_ABBR,
   ATL_HTML_TAG_ACRONYM,
   ATL_HTML_TAG_OL,
   ATL_HTML_TAG_UL,
   ATL_HTML_TAG_LI,
   ATL_HTML_TAG_DL,
   ATL_HTML_TAG_DT,
   ATL_HTML_TAG_DD,
   ATL_HTML_TAG_TABLE,
   ATL_HTML_TAG_TR,
   ATL_HTML_TAG_TD,
   ATL_HTML_TAG_FORM,
   ATL_HTML_TAG_INPUT,
   ATL_HTML_TAG_SELECT,
   ATL_HTML_TAG_OPTION,
   ATL_HTML_TAG_HEAD,
   ATL_HTML_TAG_HTML,
   ATL_HTML_TAG_MAP,
   ATL_HTML_TAG_AREA,
   ATL_HTML_TAG_BASE,
   ATL_HTML_TAG_BDO,
   ATL_HTML_TAG_BIG,
   ATL_HTML_TAG_BUTTON,
   ATL_HTML_TAG_IFRAME,
   ATL_HTML_TAG_LABEL,
   ATL_HTML_TAG_LINK,
```

```
    ATL_HTML_TAG_META,
    ATL_HTML_TAG_NOFRAMES,
    ATL_HTML_TAG_NOSCRIPT,
    ATL_HTML_TAG_COL,
    ATL_HTML_TAG_COLGROUP,
    ATL_HTML_TAG_FIELDSET,
    ATL_HTML_TAG_LEGEND,
    ATL_HTML_TAG_TBODY,
    ATL_HTML_TAG_TEXTAREA,
    ATL_HTML_TAG_TFOOT,
    ATL_HTML_TAG_TH,
    ATL_HTML_TAG_TITLE,
    ATL_HTML_TAG_TT,
    ATL_HTML_TAG_SMALL,
    ATL_HTML_TAG_SPAN,
    ATL_HTML_TAG_LAST
};
```

Requirements

Header: atlhtml.h

See Also

Reference

CStreamFormatter::StartTag

CStreamFormatter::EndTag

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Enums

ATL_MIME_PRIORITY

The members of this enumeration provide constants for the priorities of a MIME message.

```
enum ATL_MIME_PRIORITY{
   ATL_MIME_HIGH_PRIORITY   = 1,
   ATL_MIME_NORMAL_PRIORITY = 3,
   ATL_MIME_LOW_PRIORITY    = 5,
   ATL_MIME_PRIORITY_ERROR  = 0
};
```

Requirements

Header: atlmime.h

See Also

Reference

CMimeHeader Class

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Enums

ATLSRV_REQUESTTYPE

The members of this enumeration describe the type of an ATL Server request.

```
enum ATLSRV_REQUESTTYPE{
   ATLSRV_REQUEST_UNKNOWN,
   ATLSRV_REQUEST_STENCIL,
   ATLSRV_REQUEST_DLL
};
```

Remarks

These values are described below:

Value	Description
ATLSRV_REQUEST_UNKNOWN	The request type is currently unknown.
ATLSRV_REQUEST_STENCIL	The request is for a server response file.
ATLSRV_REQUEST_DLL	The request is for a request handler DLL.

Requirements

Header: atlisapi.h

See Also

Reference

AtlServerRequest::dwRequestType

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Enums

ATLSRV_STATE

The members of this enumeration describe the state of an ATL Server request.

```
enum ATLSRV_STATE{
   ATLSRV_STATE_BEGIN,
   ATLSRV_STATE_CONTINUE,
   ATLSRV_STATE_DONE,
   ATLSRV_STATE_CACHE_DONE
};
```

Remarks

These values are described below:

Value	Description
ATLSRV_STATE_BEGIN	The request has just arrived and the type has not yet been determined.
ATLSRV_STATE_CONTINUE	The request is a continuation of an asynchronous request.
ATLSRV_STATE_DONE	The request is a continuation of an asynchronous request and the server is done with it.
ATLSRV_STATE_CACHE_DONE	The request is the callback of a cached page.

Requirements

Header: atlisapi.h

See Also

Reference

AtlServerRequest::dwRequestState

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Enums

ATL_URL_SCHEME

The members of this enumeration provide constants for the schemes understood by CUrl.

```
enum ATL_URL_SCHEME{
   ATL_URL_SCHEME_UNKNOWN = -1,
   ATL_URL_SCHEME_FTP     = 0,
   ATL_URL_SCHEME_GOPHER  = 1,
   ATL_URL_SCHEME_HTTP    = 2,
   ATL_URL_SCHEME_HTTPS   = 3,
   ATL_URL_SCHEME_FILE    = 4,
   ATL_URL_SCHEME_NEWS    = 5,
   ATL_URL_SCHEME_MAILTO  = 6,
   ATL_URL_SCHEME_SOCKS   = 7
};
```

Requirements

Header: atlutil.h

See Also

Reference

CUrl::SetScheme

CUrl::GetScheme

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Enums

REParseError

The members of this enumeration provide values corresponding to errors returned by CAtlRegExp::Parse.

```
enum REParseError{
   REPARSE_ERROR_OK = 0,
   REPARSE_ERROR_OUTOFMEMORY,
   REPARSE_ERROR_BRACE_EXPECTED,
   REPARSE_ERROR_PAREN_EXPECTED,
   REPARSE_ERROR_BRACKET_EXPECTED,
   REPARSE_ERROR_UNEXPECTED,
   REPARSE_ERROR_EMPTY_RANGE,
   REPARSE_ERROR_INVALID_GROUP,
   REPARSE_ERROR_INVALID_RANGE,
   REPARSE_ERROR_EMPTY_REPEATOP,
   REPARSE_ERROR_INVALID_INPUT,
};
```

Remarks

These values are described below:

Value	Description
REPARSE_ERROR_OK	No error occurred.
REPARSE_ERROR_OUTOFMEMORY	Out of memory.
REPARSE_ERROR_BRACE_EXPECTED	A closing brace was expected.
REPARSE_ERROR_PAREN_EXPECTED	A closing parenthesis was expected.
REPARSE_ERROR_BRACKET_EXPECTED	A closing bracket was expected.
REPARSE_ERROR_UNEXPECTED	An unspecified fatal error occurred.
REPARSE_ERROR_EMPTY_RANGE	A range expression was empty.
REPARSE_ERROR_INVALID_GROUP	A back reference was made to a group that did not exist.
REPARSE_ERROR_INVALID_RANGE	An invalid range was specified.
REPARSE_ERROR_EMPTY_REPEATOP	A repeat operator (***** or **+**) was applied to an expression that could be empty.
REPARSE_ERROR_INVALID_INPUT	The input string was invalid.

Requirements

Header: atlrx.h

See Also

Reference

CAtlRegExp::Parse

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Enums

SOAP_ERROR_CODE

The members of this enumeration provide values corresponding to SOAP faultcodes.

```
enum SOAP_ERROR_CODE
{
   SOAP_E_UNK,
   SOAP_E_VERSION_MISMATCH,
   SOAP_E_MUST_UNDERSTAND,
   SOAP_E_CLIENT,
   SOAP_E_SERVER
};
```

Remarks

These values are described below:

Value	Description
SOAP_E_UNK	Unknown faultcode.
SOAP_E_VERSION_MISMATCH	The VersionMismatch faultcode.
SOAP_E_MUST_UNDERSTAND	The MustUnderstand faultcode.
SOAP_E_CLIENT	The Client faultcode.
SOAP_E_SERVER	The Server faultcode.

Requirements

Header: atlsoap.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Enums

SOAPCLIENT_ERROR

The members of this enumeration provide values corresponding to errors experienced by a SOAP client.

```
enum SOAPCLIENT_ERROR{
   SOAPCLIENT_SUCCESS=0,
   SOAPCLIENT_INITIALIZE_ERROR,
   SOAPCLIENT_OUTOFMEMORY,
   SOAPCLIENT_GENERATE_ERROR,
   SOAPCLIENT_CONNECT_ERROR,
   SOAPCLIENT_SEND_ERROR,
   SOAPCLIENT_SERVER_ERROR,
   SOAPCLIENT_SOAPFAULT,
   SOAPCLIENT_PARSEFAULT_ERROR,
   SOAPCLIENT_READ_ERROR,
   SOAPCLIENT_PARSE_ERROR
};
```

Remarks

These values are described below:

Value	Description
SOAPCLIENT_SUCCESS	No errors.
SOAPCLIENT_INITIALIZE_ERROR	Initialization failed. Possibly caused by an MSXML installation problem.
SOAPCLIENT_OUTOFMEMORY	Out of memory.
SOAPCLIENT_GENERATE_ERROR	Failed to generate the response.
SOAPCLIENT_CONNECT_ERROR	Failed to connect to the server.
SOAPCLIENT_SEND_ERROR	Failed to send the message.
SOAPCLIENT_SERVER_ERROR	Server error.
SOAPCLIENT_SOAPFAULT	The server returned a SOAP fault.
SOAPCLIENT_PARSEFAULT_ERROR	Failed to parse a SOAP fault.
SOAPCLIENT_READ_ERROR	Failed while reading the response.
SOAPCLIENT_PARSE_ERROR	Failed while parsing the response.

Requirements

Header: atlsoap.h

See Also

Reference

CSoapSocketClientT::GetClientError
CSoapSocketClientT::SetClientError
CSoapMSXMLInetClient::GetClientError
CSoapMSXMLInetClient::SetClientError
CSoapWininetClient::GetClientError
CSoapWininetClient::SetClientError

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Enums

ATL Server Functions

The following functions are included in the ATL Server Library.

Name	Description
AtlCanonicalizeUrl	Call this function to canonicalize a URL, which includes converting unsafe characters and spaces into escape sequences.
AtlCombineUrl	Call this function to combine a base URL and a relative URL into a single, canonical URL.
AtlEscapeUrl	Call this function to convert all unsafe characters to escape sequences.
AtlGetDefaultUrlPort	Call this function to get the default port number associated with a particular Internet protocol or scheme.
AtlGetHexValue	Call this function to get the numeric value of a hexadecimal digit.
AtlHexDecode	Decodes a string of data that has been encoded as hexadecimal text such as by a previous call to AtlHexEncode.
AtlHexDecodeGetRequiredLength	Call this function to get the size in bytes of a buffer that could contain data decoded from a hex-encoded string of the specified length.
AtlHexEncode	Call this function to encode some data as a string of hexadecimal text.
AtlHexEncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.
AtlHexValue	Call this function to get the numeric value of a hexadecimal digit.
AtlImpersonateClient	Call this function to change the current thread's token to impersonate the client making the current request.
AtlIsUnsafeUrlChar	Call this function to find out whether a character is safe for use in a URL.
AtlsHttpError	Call this function to create an HTTP_CODE from an HTTP status code and a subcode.
AtlsSecErrHandlerFunc	This function is used by ATL Server projects as the security error handler that is called in response to a buffer overrun.
AtlUnescapeUrl	Call this function to convert escaped characters back to their original values.
AtlUnicodeToUTF8	Call this function to convert a Unicode string to UTF-8.
Base64Decode	Decodes a string of data that has been encoded in base64 format such as by a previous call to Base64Encode.
Base64DecodeGetRequiredLength	Call this function to get the size in bytes of a buffer that could contain data decoded from a base64-encoded string of the specified length.
Base64Encode	Call this function to base64-encode some data.

Base64EncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.
BEncode	Call this function to convert some data using the "B" encoding.
BEncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.
ClosePerfMon	This function is a Close function that must be provided by performance extension DLLs that expose performance counters.
CollectPerfMon	This function is a Collect function that must be provided by performance extension DLLs th at expose performance counters.
CreateRequestHandlerSync	Call this function to create a request handler to handle a synchronous call.
CreateServerContext	Call this function to create a new CServerContext.
EscapeToCString	Call this function to URL-encode a string and have the result appended to a **CString** passed to the function by reference.
EscapeXML	Call this function to convert characters that are unsafe for use in XML to their safe equivale nts.
GetContentTypeFromFileName	Call this function to get the MIME content type of a file based on its name.
GetDataSource	Call this function to retrieve a cached data source or add it to the cache if not already prese nt.
GetExtendedChars	Call this function to get the number of extended characters in a string.
GetScriptFullFileName	Call this function to retrieve the full canonical physical path of a file relative to the current s cript.
GetStatusHeader	Call this function to obtain a status header (for example, 200 OK) that can be returned in th e header of an HTTP response.
IsAsyncContinueStatus	Call this function to determine whether an ATL Server HTTP_CODE status code indicates tha t asynchronous processing is still in progress.
IsAsyncDoneStatus	Call this function to determine whether an ATL Server status code indicates that asynchron ous processing has completed.
IsAsyncFlushStatus	Call this function to determine whether a status code indicates that asynchronous processin g has occurred and ATL Server code is responsible for flushing.
IsAsyncNoFlushStatus	Call this function to determine whether a status code indicates that asynchronous processin g has occurred and ATL Server code is not responsible for flushing.
IsAsyncStatus	Call this function to determine whether a status code indicates that asynchronous processin g has occurred.
IsExtendedChar	Call this function to find out if a given character is an extended character (less than 32, grea ter than 126, and not a tab, linefeed or carriage return)

IsNullByType	Specialize this function to define what it means for a type to be null when used with CValidateObject.
OpenPerfMon	This function is an Open function that must be provided by performance extension DLLs th at expose performance counters.
QEncode	Call this function to convert some data using the "Q" encoding.
QEncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.
QPDecode	Decodes a string of data that has been encoded in quoted-printable format such as by a pre vious call to QPEncode.
QPDecodeGetRequiredLength	Call this function to get the size in bytes of a buffer that could contain data decoded from q uoted-printable-encoded string of the specified length.
QPEncode	Call this function to encode some data in quoted-printable format.
QPEncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.
ReadClientData	Call this function to read data from the client making the current request.
RegisterPerfMon	Call this function to register performance objects and counters.
RemoveDataSource	Call this function to close a cached connection and remove it from the data source cache.
RenderError	Call this function to write an error response to the client.
RGBToHtml	Converts a COLORREF value to the HTML text corresponding to that color value.
RSkipSpace	Returns the pointer to the previous non-whitespace character.
SafeStringCopy	Call this function to copy a string into a fixed size character buffer.
SetRfc822Time	Call this function to get the current system time as a string in RFC 822 format.
SkipSpace	Returns the pointer to the next non-whitespace character.
SystemTimeToHttpDate	Call this function to convert a system time to a string in a format suitable for using in HTTP headers.
UnregisterPerfMon	Call this function to unregister performance objects and counters.
UUDecode	Decodes a string of data that has been uuencoded such as by a previous call to UUEncode.
UUDecodeGetRequiredLength	Call this function to get the size in bytes of a buffer that could contain data decoded from a uuencoded string of the specified length.
UUEncode	Call this function to uuencode some data.
UUEncodeGetRequiredLength	Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.

ATLPath::AddBackslash	This function is an overloaded wrapper for PathAddBackslash.
ATLPath::AddExtension	This function is an overloaded wrapper for PathAddExtension.
ATLPath::Append	This function is an overloaded wrapper for PathAppend.
ATLPath::BuildRoot	This function is an overloaded wrapper for PathBuildRoot.
ATLPath::Canonicalize	This function is an overloaded wrapper for PathCanonicalize.
ATLPath::Combine	This function is an overloaded wrapper for PathCombine.
ATLPath::CommonPrefix	This function is an overloaded wrapper for PathCommonPrefix.
ATLPath::CompactPath	This function is an overloaded wrapper for PathCompactPath.
ATLPath::CompactPathEx	This function is an overloaded wrapper for PathCompactPathEx.
ATLPath::FileExists	This function is an overloaded wrapper for PathFileExists.
ATLPath::FindExtension	This function is an overloaded wrapper for PathFindExtension.
ATLPath::FindFileName	This function is an overloaded wrapper for PathFindFileName.
ATLPath::GetDriveNumber	This function is an overloaded wrapper for PathGetDriveNumber.
ATLPath::IsDirectory	This function is an overloaded wrapper for PathIsDirectory.
ATLPath::IsFileSpec	This function is an overloaded wrapper for PathIsFileSpec.
ATLPath::IsPrefix	This function is an overloaded wrapper for PathIsPrefix.
ATLPath::IsRelative	This function is an overloaded wrapper for PathIsRelative.
ATLPath::IsRoot	This function is an overloaded wrapper for PathIsRoot.
ATLPath::IsSameRoot	This function is an overloaded wrapper for PathIsSameRoot.
ATLPath::IsUNC	This function is an overloaded wrapper for PathIsUNC.
ATLPath::IsUNCServer	This function is an overloaded wrapper for PathIsUNCServer.
ATLPath::IsUNCServerShare	This function is an overloaded wrapper for PathIsUNCServerShare.
ATLPath::MakePretty	This function is an overloaded wrapper for PathMakePretty.
ATLPath::MatchSpec	This function is an overloaded wrapper for PathMatchSpec.
ATLPath::QuoteSpaces	This function is an overloaded wrapper for PathQuoteSpaces.
ATLPath::RelativePathTo	This function is an overloaded wrapper for PathRelativePathTo.

ATLPath::RemoveArgs	This function is an overloaded wrapper for PathRemoveArgs.
ATLPath::RemoveBackslash	This function is an overloaded wrapper for PathRemoveBackslash.
ATLPath::RemoveBlanks	This function is an overloaded wrapper for PathRemoveBlanks.
ATLPath::RemoveExtension	This function is an overloaded wrapper for PathRemoveExtension.
ATLPath::RemoveFileSpec	This function is an overloaded wrapper for PathRemoveFileSpec.
ATLPath::RenameExtension	This function is an overloaded wrapper for PathRenameExtension.
ATLPath::SkipRoot	This function is an overloaded wrapper for PathSkipRoot.
ATLPath::StripPath	This function is an overloaded wrapper for PathStripPath.
ATLPath::StripToRoot	This function is an overloaded wrapper for PathStripToRoot.
ATLPath::UnquoteSpaces	This function is an overloaded wrapper for PathUnquoteSpaces.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

AtlCanonicalizeUrl

Call this function to canonicalize a URL, which includes converting unsafe characters and spaces into escape sequences.

```
inline BOOL AtlCanonicalizeUrl(
   LPCTSTR szUrl,
   LPTSTR szCanonicalized,
   DWORD* pdwMaxLength,
   DWORD dwFlags = 0
) throw( );
```

Parameters

szUrl

The URL to be canonicalized.

szCanonicalized

Caller-allocated buffer to receive the canonicalized URL.

pdwMaxLength

Pointer to a variable that contains the length in characters of *szCanonicalized*. If the function succeeds, the variable receives the number of characters written to the buffer not including the terminating null character. If the function fails, the variable receives the required length in bytes of the buffer including space for the terminating null character.

dwFlags

Flags controlling the behavior of this function. See ATL_URL Flags.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

Behaves like the current version of InternetCanonicalizeUrl but does not require WinInet or Internet Explorer to be installed.

Requirements

Header: atlutil.h

See Also

Reference

InternetCanonicalizeUrl

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

AtlCombineUrl

Call this function to combine a base URL and a relative URL into a single, canonical URL.

```
inline BOOL AtlCombineUrl(
   LPCTSTR szBaseUrl,
   LPCTSTR szRelativeUrl,
   LPTSTR szBuffer,
   DWORD* pdwMaxLength,
   DWORD dwFlags = 0
) throw( );
```

Parameters

szBaseUrl

The base URL.

szRelativeUrl

The URL relative to the base URL.

szBuffer

Caller-allocated buffer to receive the canonicalized URL.

pdwMaxLength

Pointer to a variable that contains the length in characters of *szBuffer*. If the function succeeds, the variable receives the number of characters written to the buffer not including the terminating null character. If the function fails, the variable receives the required length in bytes of the buffer including space for the terminating null character.

dwFlags

Flags controlling the behavior of this function. See ATL_URL Flags.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

Behaves like the current version of InternetCombineUrl but does not require WinInet or Internet Explorer to be installed.

Requirements

Header: atlutil.h

See Also

Reference

InternetCombineUrl

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

AtlEscapeUrl

Call this function to convert all unsafe characters to escape sequences.

```
inline BOOL AtlEscapeUrl(
   LPCSTR szStringIn,
   LPSTR szStringOut,
   DWORD* pdwStrLen,
   DWORD dwMaxLength,
   DWORD dwFlags = 0
) throw( );
inline BOOL AtlEscapeUrl(
   LPCWSTR szStringIn,
   LPWSTR szStringOut,
   DWORD* pdwStrLen,
   DWORD dwMaxLength,
   DWORD dwFlags = 0
) throw( );
```

Parameters

lpszStringIn

The URL to be converted.

lpszStringOut

Caller-allocated buffer to which the converted URL will be written.

pdwStrLen

Pointer to a DWORD variable. If the function succeeds, *pdwStrLen* receives the number of characters written to the buffer, not including the terminating null character. If the function fails, the variable receives the required length in bytes of the buffer including space for the terminating null character. When using the wide character version of this method, *pdwStrLen* receives the number of characters required, not the number of bytes.

dwMaxLength

The size of the buffer *lpszStringOut*.

dwFlags

Flags controlling the behavior of this function. See ATL_URL Flags.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Requirements

Header: atlutil.h

See Also

Reference

AtlUnescapeUrl

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

AtlGetDefaultUrlPort

Call this function to get the default port number associated with a particular Internet protocol or scheme.

```
inline ATL_URL_PORT AtlGetDefaultUrlPort(
   ATL_URL_SCHEME m_nScheme
) throw( );
```

Parameters

m_nScheme

The ATL_URL_SCHEME value identifying the scheme for which you want to obtain the port number.

Return Value

The ATL_URL_PORT associated with the specified scheme or ATL_URL_INVALID_PORT_NUMBER if the scheme is not recognized.

Requirements

Header: atlutil.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

AtlGetHexValue

Call this function to get the numeric value of a hexadecimal digit.

```
inline char AtlGetHexValue(
   char chIn
) throw( );
```

Parameters

chIn

The hexadecimal character '0'-'9', 'A'-'F', or 'a'-'f'.

Return Value

The numeric value of the input character interpreted as a hexadecimal digit. For example, an input of '0' returns a value of 0 and an input of 'A' returns a value of 10. If the input character is not a hexadecimal digit, this function returns -1.

Requirements

Header: atlenc.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

AtlHexDecode

Decodes a string of data that has been encoded as hexadecimal text such as by a previous call to AtlHexEncode.

```
inline BOOL AtlHexDecode(
   LPCSTR pSrcData,
   int nSrcLen,
   LPBYTE pbDest,
   int* pnDestLen
) throw( );
```

Parameters

pSrcData

The string containing the data to be decoded.

nSrcLen

The length in characters of *pSrcData*.

pbDest

Caller-allocated buffer to receive the decoded data.

pnDestLen

Pointer to a variable that contains the length in bytes of *pbDest*. If the function succeeds, the variable receives the number of bytes written to the buffer. If the function fails, the variable receives the required length in bytes of the buffer.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Requirements

Header: atlenc.h

See Also

Reference

AtlHexDecodeGetRequiredLength
AtlHexEncode
AtlHexEncodeGetRequiredLength

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

AtlHexDecodeGetRequiredLength

Call this function to get the size in bytes of a buffer that could contain data decoded from a hex-encoded string of the specified length.

```
inline int AtlHexDecodeGetRequiredLength(
   int nSrcLen
) throw( );
```

Parameters

nSrcLen

The number of characters in the encoded string.

Return Value

The number of bytes required for a buffer that could hold a decoded string of *nSrcLen* characters.

Requirements

Header: atlenc.h

See Also

Reference

AtlHexDecode
AtlHexEncode
AtlHexEncodeGetRequiredLength

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

AtlHexEncode

Call this function to encode some data as a string of hexadecimal text.

```
inline BOOL AtlHexEncode(
   const BYTE * pbSrcData,
   int nSrcLen,
   LPSTR szDest,
   int * pnDestLen
) throw( );
```

Parameters

pbSrcData

The buffer containing the data to be encoded.

nSrcLen

The length in bytes of the data to be encoded.

szDest

Caller-allocated buffer to receive the encoded data.

pnDestLen

Pointer to a variable that contains the length in characters of *szDest*. If the function succeeds, the variable receives the number of characters written to the buffer. If the function fails, the variable receives the required length in characters of the buffer.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

Each byte of source data is encoded as 2 hexadecimal characters.

Requirements

Header: atlenc.h

Example

See the ForceLogin Sample.

See Also

Reference

AtlHexDecodeGetRequiredLength
AtlHexDecode
AtlHexEncodeGetRequiredLength

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

AtlHexEncodeGetRequiredLength

Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.

```
inline int AtlHexEncodeGetRequiredLength(
   int nSrcLen
) throw( );
```

Parameters

nSrcLen

The number of bytes of data to be encoded.

Return Value

The number of characters required for a buffer that could hold encoded data of *nSrcLen* bytes.

Requirements

Header: atlenc.h

Example

See the ForceLogin Sample.

See Also

Reference

AtlHexDecodeGetRequiredLength
AtlHexEncode
AtlHexDecode

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

AtlHexValue

Call this function to get the numeric value of a hexadecimal digit.

```
inline short AtlHexValue(
   char chIn
) throw( );
```

Parameters

chIn

The hexadecimal character '0'-'9', 'A'-'F', or 'a'-'f'.

Return Value

The numeric value of the input character interpreted as a hexadecimal digit. For example, an input of '0' returns a value of 0 and an input of 'A' returns a value of 10. If the input character is not a hexadecimal digit, this function returns -1.

Requirements

Header: atlutil.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

AtlImpersonateClient

Call this function to change the current thread's token to impersonate the client making the current request.

```
inline BOOL AtlImpersonateClient(
   IHttpServerContext* pServerContext
) throw( );
```

Parameters

pServerContext

The server context for the current request.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

Calls IHttpServerContext::GetImpersonationToken and SetThreadToken.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

AtlIsUnsafeUrlChar

Call this function to find out whether a character is safe for use in a URL.

```
inline BOOL AtlIsUnsafeUrlChar(
   char chIn
) throw( );
```

Parameters

chIn

The character to be tested for safety.

Return Value

Returns **TRUE** if the input character is unsafe, **FALSE** otherwise.

Remarks

Characters that should not be used in URLs can be tested using this function and converted using EscapeToCString or AtlCanonicalizeUrl.

Requirements

Header: atlutil.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

ATLPath::AddBackslash

This function is an overloaded wrapper for PathAddBackslash.

```
inline char* AddBackslash(
   char* pszPath
);
inline wchar_t* AddBackslash(
   wchar_t* pszPath
);
```

Remarks

See PathAddBackslash for details.

See Also

Reference

PathAddBackslash

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::AddExtension

This function is an overloaded wrapper for PathAddExtension.

```
inline BOOL AddExtension(
   char* pszPath,
   const char* pszExtension
);
inline BOOL AddExtension(
   wchar_t* pszPath,
   const wchar_t* pszExtension
);
```

Remarks

See PathAddExtension for details.

See Also

Reference

PathAddExtension

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::Append

This function is an overloaded wrapper for PathAppend.

```
inline BOOL Append(
   char* pszPath,
   const char* pszMore
);
inline BOOL Append(
   wchar_t* pszPath,
   const wchar_t* pszMore
);
```

Remarks

See PathAppend for details.

See Also

Reference

PathAppend

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::BuildRoot

This function is an overloaded wrapper for PathBuildRoot.

```
inline char* BuildRoot(
   char* pszPath,
   int iDrive
);
inline wchar_t* BuildRoot(
   wchar_t* pszPath,
   int iDrive
);
```

Remarks

See PathBuildRoot for details.

See Also

Reference

PathBuildRoot

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::Canonicalize

This function is an overloaded wrapper for PathCanonicalize.

```
inline BOOL Canonicalize(
   char* pszDest,
   const char* pszSrc
);
inline BOOL Canonicalize(
   wchar_t* pszDest,
   const wchar_t* pszSrc
);
```

Remarks

See PathCanonicalize for details.

See Also

Reference

PathCanonicalize

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::Combine

This function is an overloaded wrapper for PathCombine.

```
inline char* Combine(
   char* pszDest,
   const char* pszDir,
   const char* pszFile
);
inline wchar_t* Combine(
   wchar_t* pszDest,
   const wchar_t* pszDir,
   const wchar_t* pszFile
);
```

Remarks

See PathCombine for details.

See Also

Reference

PathCombine

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::CommonPrefix

This function is an overloaded wrapper for PathCommonPrefix.

```
inline int CommonPrefix(
   const char* pszFile1,
   const char* pszFile2,
   char* pszDest
);
inline int CommonPrefix(
   const wchar_t* pszFile1,
   const wchar_t* pszFile2,
   wchar_t* pszDest
);
```

Remarks

See PathCommonPrefix for details.

See Also

Reference

PathCommonPrefix

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::CompactPath

This function is an overloaded wrapper for PathCompactPath.

```
inline BOOL CompactPath(
   HDC hDC,
   char* pszPath,
   UINT dx
);
inline BOOL CompactPath(
   HDC hDC,
   wchar_t* pszPath,
   UINT dx
);
```

Remarks

See PathCompactPath for details.

See Also

Reference

PathCompactPath

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::CompactPathEx

This function is an overloaded wrapper for PathCompactPathEx.

```
inline BOOL CompactPathEx(
   char* pszDest,
   const char* pszSrc,
   UINT nMaxChars,
   DWORD dwFlags
);
inline BOOL CompactPathEx(
   wchar_t* pszDest,
   const wchar_t* pszSrc,
   UINT nMaxChars,
   DWORD dwFlags
);
```

Remarks

See PathCompactPathEx for details.

See Also

Reference

PathCompactPathEx

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::FileExists

This function is an overloaded wrapper for PathFileExists.

```
inline BOOL FileExists(
   const char* pszPath
);
inline BOOL FileExists(
   const wchar_t* pszPath
);
```

Remarks

See PathFileExists for details.

See Also

Reference

PathFileExists

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::FindExtension

This function is an overloaded wrapper for PathFindExtension.

```
inline char* FindExtension(
   const char* pszPath
);
inline wchar_t* FindExtension(
   const wchar_t* pszPath
);
```

Remarks

See PathFindExtension for details.

See Also

Reference

PathFindExtension

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::FindFileName

This function is an overloaded wrapper for PathFindFileName.

```
inline char* FindFileName(
   const char* pszPath
);
inline wchar_t* FindFileName(
   const wchar_t* pszPath
);
```

Remarks

See PathFindFileName for details.

See Also

Reference

PathFindFileName

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::GetDriveNumber

This function is an overloaded wrapper for PathGetDriveNumber.

```
inline int GetDriveNumber(
   const char* pszPath
);
inline int GetDriveNumber(
   const wchar_t* pszPath
);
```

Remarks

See PathGetDriveNumber for details.

See Also

Reference

PathGetDriveNumber

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::IsDirectory

This function is an overloaded wrapper for PathIsDirectory.

```
inline BOOL IsDirectory(
   const char* pszPath
);
inline BOOL IsDirectory(
   const wchar_t* pszPath
);
```

Remarks

See PathIsDirectory for details.

See Also

Reference

PathIsDirectory

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::IsFileSpec

This function is an overloaded wrapper for PathIsFileSpec.

```
inline BOOL IsFileSpec(
   const char* pszPath
);
inline BOOL IsFileSpec(
   const wchar_t* pszPath
);
```

Remarks

See PathIsFileSpec for details.

See Also

Reference

PathIsFileSpec

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::IsPrefix

This function is an overloaded wrapper for PathIsPrefix.

```
inline BOOL IsPrefix(
   const char* pszPrefix,
   const char* pszPath
);
inline BOOL IsPrefix(
   const wchar_t* pszPrefix,
   const wchar_t* pszPath
);
```

Remarks

See PathIsPrefix for details.

See Also

Reference

PathIsPrefix

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::IsRelative

This function is an overloaded wrapper for PathIsRelative.

```
inline BOOL IsRelative(
   const char* pszPath
);
inline BOOL IsRelative(
   const wchar_t* pszPath
);
```

Remarks

See PathIsRelative for details.

See Also

Reference

PathIsRelative

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::IsRoot

This function is an overloaded wrapper for PathIsRoot.

```
inline BOOL IsRoot(
   const char* pszPath
);
inline BOOL IsRoot(
   const wchar_t* pszPath
);
```

Remarks

See PathIsRoot for details.

See Also

Reference

PathIsRoot

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::IsSameRoot

This function is an overloaded wrapper for PathIsSameRoot.

```
inline BOOL IsSameRoot(
   const char* pszPath1,
   const char* pszPath2
);
inline BOOL IsSameRoot(
   const wchar_t* pszPath1,
   const wchar_t* pszPath2
);
```

Remarks

See PathIsSameRoot for details.

See Also

Reference

PathIsSameRoot

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::IsUNC

This function is an overloaded wrapper for PathIsUNC.

```
inline BOOL IsUNC(
   const char* pszPath
);
inline BOOL IsUNC(
   const wchar_t* pszPath
);
```

Remarks

See PathIsUNC for details.

See Also

Reference

PathIsUNC

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::IsUNCServer

This function is an overloaded wrapper for PathIsUNCServer.

```
inline BOOL IsUNCServer(
   const char* pszPath
);
inline BOOL IsUNCServer(
   const wchar_t* pszPath
);
```

Remarks

See PathIsUNCServer for details.

See Also

Reference

PathIsUNCServer

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::IsUNCServerShare

This function is an overloaded wrapper for PathIsUNCServerShare.

```
inline BOOL IsUNCServerShare(
   const char* pszPath
);
inline BOOL IsUNCServerShare(
   const wchar_t* pszPath
);
```

Remarks

See PathIsUNCServerShare for details.

See Also

Reference

PathIsUNCServerShare

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::MakePretty

This function is an overloaded wrapper for PathMakePretty.

```
inline BOOL MakePretty(
   char* pszPath
);
inline BOOL MakePretty(
   wchar_t* pszPath
);
```

Remarks

See PathMakePretty for details.

See Also

Reference

PathMakePretty

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::MatchSpec

This function is an overloaded wrapper for PathMatchSpec.

```
inline BOOL MatchSpec(
   const char* pszPath,
   const char* pszSpec
);
inline BOOL MatchSpec(
   const wchar_t* pszPath,
   const wchar_t* pszSpec
);
```

Remarks

See PathMatchSpec for details.

See Also

Reference

PathMatchSpec

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::QuoteSpaces

This function is an overloaded wrapper for PathQuoteSpaces.

```
inline void QuoteSpaces(
   char* pszPath
);
inline void QuoteSpaces(
   wchar_t* pszPath
);
```

Remarks

See PathQuoteSpaces for details.

See Also

Reference

PathQuoteSpaces

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::RelativePathTo

This function is an overloaded wrapper for PathRelativePathTo.

```
inline BOOL RelativePathTo(
   char* pszPath,
   const char* pszFrom,
   DWORD dwAttrFrom,
   const char* pszTo,
   DWORD dwAttrTo
);
inline BOOL RelativePathTo(
   wchar_t* pszPath,
   const wchar_t* pszFrom,
   DWORD dwAttrFrom,
   const wchar_t* pszTo,
   DWORD dwAttrTo
);
```

Remarks

See PathRelativePathTo for details.

See Also

Reference

PathRelativePathTo

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::RemoveArgs

This function is an overloaded wrapper for PathRemoveArgs.

```
inline void RemoveArgs(
   char* pszPath
);
inline void RemoveArgs(
   wchar_t* pszPath
);
```

Remarks

See PathRemoveArgs for details.

See Also

Reference

PathRemoveArgs

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::RemoveBackslash

This function is an overloaded wrapper for PathRemoveBackslash.

```
inline char* RemoveBackslash(
   char* pszPath
);
inline wchar_t* RemoveBackslash(
   wchar_t* pszPath
);
```

Remarks

See PathRemoveBackslash for details.

See Also

Reference

PathRemoveBackslash

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::RemoveBlanks

This function is an overloaded wrapper for PathRemoveBlanks.

```
inline void RemoveBlanks(
   char* pszPath
);
inline void RemoveBlanks(
   wchar_t* pszPath
);
```

Remarks

See PathRemoveBlanks for details.

See Also

Reference

PathRemoveBlanks

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::RemoveExtension

This function is an overloaded wrapper for PathRemoveExtension.

```
inline void RemoveExtension(
   char* pszPath
);
inline void RemoveExtension(
   wchar_t* pszPath
);
```

Remarks

See PathRemoveExtension for details.

See Also

Reference

PathRemoveExtension

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::RemoveFileSpec

This function is an overloaded wrapper for PathRemoveFileSpec.

```
inline BOOL RemoveFileSpec(
   char* pszPath
);
inline BOOL RemoveFileSpec(
   wchar_t* pszPath
);
```

Remarks

See PathRemoveFileSpec for details.

See Also

Reference

PathRemoveFileSpec

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::RenameExtension

This function is an overloaded wrapper for PathRenameExtension.

```
inline BOOL RenameExtension(
   char* pszPath,
   const char* pszExt
);
inline BOOL RenameExtension(
   wchar_t* pszPath,
   const wchar_t* pszExt
);
```

Remarks

See PathRenameExtension for details.

See Also

Reference

PathRenameExtension

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::SkipRoot

This function is an overloaded wrapper for PathSkipRoot.

```
inline char* SkipRoot(
   const char* pszPath
);
inline wchar_t* SkipRoot(
   const wchar_t* pszPath
);
```

Remarks

See PathSkipRoot for details.

See Also

Reference

PathSkipRoot

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::StripPath

This function is an overloaded wrapper for PathStripPath.

```
inline void StripPath(
   char* pszPath
);
inline void StripPath(
   wchar_t* pszPath
);
```

Remarks

See PathStripPath for details.

See Also

Reference

PathStripPath

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::StripToRoot

This function is an overloaded wrapper for PathStripToRoot.

```
inline BOOL StripToRoot(
   char* pszPath
);
inline BOOL StripToRoot(
   wchar_t* pszPath
);
```

Remarks

See PathStripToRoot for details.

See Also

Reference

PathStripToRoot

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ATLPath::UnquoteSpaces

This function is an overloaded wrapper for PathUnquoteSpaces.

```
inline void UnquoteSpaces(
   char* pszPath
);
inline void UnquoteSpaces(
   wchar_t* pszPath
);
```

Remarks

See PathUnquoteSpaces for details.

See Also

Reference

PathUnquoteSpaces

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

AtlsHttpError

Call this function to create an HTTP_CODE from an HTTP status code and a subcode.

```
ATL_NOINLINE inline HTTP_CODE AtlsHttpError(
   WORD wStatus,
   WORD wSubErr
) throw( );
```

Parameters

wStatus

Zero or an HTTP status code as defined in the HTTP 1.1 RFC (http://www.ietf.org/rfc/rfc2616.txt).

wSubErr

An HTTP_CODE subcode.

Return Value

The HTTP_CODE with the status code in the low WORD and the subcode in the high WORD.

Requirements

Header: atlserr.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

AtlsSecErrHandlerFunc

This function no longer works. It was used by ATL Server projects as the security error handler that is called in response to a buffer overrun.

```
inline void __cdecl AtlsSecErrHandlerFunc(
   int nCode,
   void* /* pv */
);
```

Parameters

nCode

Failure code. Always _SECERR_BUFFER_OVERRUN.

pv

Pointer to extra data. Always NULL.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

AtlUnescapeUrl

Call this function to convert escaped characters back to their original values.

```
inline BOOL AtlUnescapeUrl(
   LPCSTR szStringIn,
   LPSTR szStringOut,
   LPDWORD pdwStrLen,
   DWORD dwMaxLength
) throw( );
inline BOOL AtlUnescapeUrl(
   LPCWSTR szStringIn,
   LPWSTR szStringOut,
   LPDWORD pdwStrLen,
   DWORD dwMaxLength
) throw( );
```

Parameters

lpszStringIn

The URL to be converted.

lpszStringOut

Caller-allocated buffer to which the converted URL will be written.

pdwStrLen

Pointer to a DWORD variable. If the function succeeds, the variable receives the number of characters written to the buffer not including the terminating null character. If the function fails, the variable receives the required length in bytes of the buffer including space for the terminating null character.

dwMaxLength

The size of the buffer *lpszStringOut*.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

Reverses the conversion process applied by AtlEscapeUrl.

Requirements

Header: atlutil.h

See Also

Reference

AtlEscapeUrl

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

AtlUnicodeToUTF8

Call this function to convert a Unicode string to UTF-8.

```
ATL_NOINLINE inline int AtlUnicodeToUTF8(
   LPCWSTR wszSrc,
   int nSrc,
   LPSTR szDest,
   int nDest
) throw( );
```

Parameters

wszSrc

The Unicode string to be converted

nSrc

The length in characters of the Unicode string.

szDest

Caller-allocated buffer to receive the converted string.

nDest

The length in bytes of the buffer.

Return Value

Returns the number of characters for the converted string.

Remarks

To determine the size of the buffer required for the converted string, call this function passing 0 for *szDest* and *nDest*.

Requirements

Header: atlenc.h

Example

See the ShowLocalized Sample.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

Base64Decode

Decodes a string of data that has been encoded in base64 format such as by a previous call to Base64Encode.

```
inline BOOL Base64Decode(
   LPCSTR szSrc,
   int nSrcLen,
   BYTE* pbDest,
   int* pnDestLen
) throw( );
```

Parameters

szSrc

The string containing the data to be decoded.

nSrcLen

The length in characters of *szSrc*.

pbDest

Caller-allocated buffer to receive the decoded data.

pnDestLen

Pointer to a variable that contains the length in bytes of *pbDest*. If the function succeeds, the variable receives the number of bytes written to the buffer. If the function fails, the variable receives the required length in bytes of the buffer.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

The base64 encoding scheme is described in RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt).

Requirements

Header: atlenc.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

Base64DecodeGetRequiredLength

Call this function to get the size in bytes of a buffer that could contain data decoded from a base64-encoded string of the specified length.

```
inline int Base64DecodeGetRequiredLength(
   int nSrcLen
) throw( );
```

Parameters

nSrcLen

The number of characters in the encoded string.

Return Value

The number of bytes required for a buffer that could hold a decoded string of *nSrcLen* characters.

Remarks

The base64 encoding scheme is described in RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt).

Requirements

Header: atlenc.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

Base64Encode

Call this function to base64-encode some data.

```
inline BOOL Base64Encode(
   const BYTE* pbSrcData,
   int nSrcLen,
   LPSTR szDest,
   int* pnDestLen,
   DWORD dwFlags = ATL_BASE64_FLAG_NONE
) throw( );
```

Parameters

pbSrcData

The buffer containing the data to be encoded.

nSrcLen

The length in bytes of the data to be encoded.

szDest

Caller-allocated buffer to receive the encoded data.

pnDestLen

Pointer to a variable that contains the length in characters of *szDest*. If the function succeeds, the variable receives the number of characters written to the buffer. If the function fails, the variable receives the required length in characters of the buffer.

dwFlags

Flags describing how the conversion is to be performed. See ATL_BASE64 Flags.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

The base64 encoding scheme is described in RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt).

Requirements

Header: atlenc.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

Base64EncodeGetRequiredLength

Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.

```
inline int Base64EncodeGetRequiredLength(
   int nSrcLen,
   DWORD dwFlags = ATL_BASE64_FLAG_NONE
) throw( );
```

Parameters

nSrcLen

The number of bytes of data to be encoded.

dwFlags

Flags describing how the conversion is to be performed. See ATL_BASE64 Flags.

Return Value

The number of characters required for a buffer that could hold encoded data of *nSrcLen* bytes.

Remarks

The base64 encoding scheme is described in RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt).

Requirements

Header: atlenc.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

BEncode

Call this function to convert some data using the "B" encoding.

```
inline BOOL BEncode(
   BYTE* pbSrcData,
   int nSrcLen,
   LPSTR szDest,
   int* pnDestLen,
   LPCSTR pszCharSet
) throw( );
```

Parameters

pbSrcData

The buffer containing the data to be encoded.

nSrcLen

The length in bytes of the data to be encoded.

szDest

Caller-allocated buffer to receive the encoded data.

pnDestLen

Pointer to a variable that contains the length in characters of *szDest*. If the function succeeds, the variable receives the number of characters written to the buffer. If the function fails, the variable receives the required length in characters of the buffer.

pszCharSet

The character set to use for the conversion.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

The "B" encoding scheme is described in RFC 2047 (http://www.ietf.org/rfc/rfc2047.txt).

Requirements

Header: atlenc.h

See Also

Reference

BEncodeGetRequiredLength

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

BEncodeGetRequiredLength

Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.

```
inline int BEncodeGetRequiredLength(
   int nSrcLen,
   int nCharsetLen
) throw( );
```

Parameters

nSrcLen

The number of bytes of data to be encoded.

nCharsetLen

The length in characters of the character set to use for the conversion.

Return Value

The number of characters required for a buffer that could hold encoded data of *nSrcLen* bytes.

Remarks

The "B" encoding scheme is described in RFC 2047 (http://www.ietf.org/rfc/rfc2047.txt).

Requirements

Header: atlenc.h

See Also

Reference

BEncode

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

ClosePerfMon

This function is a Close function that must be provided by performance extension DLLs that expose performance counters.

```
extern "C" ATL_NOINLINE
inline DWORD __declspec( dllexport ) WINAPI ClosePerfMon( ) throw( );
```

Remarks

This method provides the expected functionality by using the performance monitoring macros and attributes that you use in your code.

Requirements

Header: atlperf.h

See Also

Reference

OpenPerfMon

CollectPerfMon

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

CollectPerfMon

This function is a Collect function that must be provided by performance extension DLLs that expose performance counters.

```
extern "C" ATL_NOINLINE
inline DWORD __declspec(dllexport) WINAPI CollectPerfMon(
   LPWSTR lpwszValue,
   LPVOID* lppData,
   LPDWORD lpcbBytes,
   LPDWORD lpcObjectTypes
) throw( );
```

Remarks

This method provides the expected functionality by using the performance monitoring macros and attributes that you use in your code.

Requirements

Header: atlperf.h

See Also

Reference

OpenPerfMon

ClosePerfMon

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

CreateRequestHandlerSync

Call this function to create a request handler to handle a synchronous call.

```
template < typename THandler >
inline BOOL CreateRequestHandlerSync(
   IIsapiExtension* pExtension,
   IUnknown** ppOut
) throw( );
```

Parameters

THandler

The class of request handler to be created.

pExtension

The ISAPI extension handling the current request.

ppOut

[out] Address of the pointer variable that, on success, receives the **IUnknown** interface pointer of the new request handler.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This function is used internally by the ATL Server framework.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

CreateServerContext

Call this function to create a new CServerContext.

```
inline CServerContext* CreateServerContext(
   HANDLE hRequestHeap
) throw( );
```

Parameters

hRequestHeap

The heap to be used by the new server context.

Return Value

A pointer to the newly created server context.

Remarks

This function is used internally by the ATL Server framework.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

EscapeToCString

Call this function to URL-encode a string and have the result appended to a **CString** passed to the function by reference.

```
ATL_NOINLINE inline bool EscapeToCString(
   CStringA& string,
   LPCSTR szBuf
) throw( );
inline bool EscapeToCString(
   CStringA& string,
   LPCWSTR wszBuf
) throw( );
```

Parameters

string

The string to which the URL-encoded data should be appended.

szBuf, wszBuf

The buffer containing the data to be URL-encoded.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

A space in the input string is encoded as a plus sign (+). Other unsafe characters (as determined by AtlIsUnsafeUrlChar) are encoded as escaped octets. An escaped octet is a percent sign (%) followed by two digits representing the hexadecimal code of the character.

Requirements

Header: atlisapi.h

Example

See the MantaWeb Sample.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

EscapeXML

Call this function to convert characters that are unsafe for use in XML to their safe equivalents.

```
inline int EscapeXML(
   const wchar_t * szIn,
   int nSrcLen,
   wchar_t * szEsc,
   int nDestLen,
   DWORD dwFlags = ATL_ESC_FLAG_NONE
) throw( );
```

Parameters

szIn

The string to be converted.

nSrclen

The length in characters of the string to be converted.

szEsc

Caller-allocated buffer to receive the converted string.

nDestLen

The length in characters of the caller-allocated buffer.

dwFlags

Flags describing how the conversion is to be performed. See ATL_ESC Flags.

Return Value

The length in characters of the converted string.

Remarks

Possible conversions performed by this function are shown in the table:

Source	Destination
<	<
>	>
&	&
'	'
"	"

Requirements

Header: atlenc.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

GetContentTypeFromFileName

Call this function to get the MIME content type of a file based on its name.

```
inline DWORD GetContentTypeFromFileName(
   LPCTSTR szFileName,
   CSimpleString& strContentType
) throw( );
```

Parameters

szFileName

The name of a file for which the MIME content type is to be obtained.

strContentType

A reference to a string in which the content type will be placed on successful return of this function.

Remarks

This function finds the MIME content type in the registry based on the extension of the file.

Requirements

Header: atlmime.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

GetDataSource

Call this function to retrieve a cached data source or add it to the cache if not already present.

```
static HRESULT ATL_NOINLINE GetDataSource(
   IServiceProvider * pProvider,
   LPCTSTR szID,
   LPCOLESTR szConn,
   CDataConnection * pSession
);
```

Parameters

pProvider

The service provider expected to expose **IDataSourceCache**.

szID

The ID of the connection in the cache. Can be the same as *szConn*.

szConn

The connection string of the data source to which to connect.

pSession

[out] Address of the CDataConnection variable that, on success, receives a copy of the data connection.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This wrapper function queries *pProvider* for the IDataSourceCache service and, if successful, calls IDataSourceCache::Add using the remaining arguments.

Requirements

Header: atlcache.h

Example

See the MantaWeb Sample and the OnlineAddressBook Sample.

See Also

Reference

RemoveDataSource

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

GetExtendedChars

Call this function to get the number of extended characters in a string.

```
inline int GetExtendedChars(
   LPCSTR szSrc,
   int nSrcLen
) throw( );
```

Parameters

szSrc

The string to be analyzed.

nSrcLen

The length of the string in characters.

Return Value

Returns the number of extended characters found within the string as determined by IsExtendedChar.

Requirements

Header: atlenc.h

See Also

Reference

IsExtendedChar

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

GetScriptFullFileName

Call this function to retrieve the full canonical physical path of a file relative to the current script.

```
inline BOOL GetScriptFullFileName(
   LPCSTR szFile,
   LPSTR szFullFileName,
   IHttpServerContext* pServerContext
) throw(...);
```

Parameters

szFile

A file path relative to the current script directory for which you are trying to retrieve the full path.

szFullFileName

A caller-allocated buffer of at least MAX_PATH + 1 characters in length. On success, this buffer contains the full canonical path of *szFile*.

pServerContext

The context for the current request. The context is used to obtain the current script directory.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

GetStatusHeader

Call this function to obtain a status header (for example, 200 OK) that can be returned in the header of an HTTP response.

```
template <class HttpUserErrorTextProvider>
void GetStatusHeader(
   CStringA & strStatus,
   DWORD dwStatus,
   DWORD dwSubStatus,
   HttpUserErrorTextProvider* pErrorProvider,
   UINT * puResId = NULL
) throw(...);
```

Parameters

strStatus

A reference to a string that will retrieve the status header on return of this function.

dwStatus

The HTTP status code as described in RFC 2616 (http://ietf.org/rfc/rfc2616.txt).

dwSubStatus

The ATL Server HTTP_CODE subcode.

pErrorProvider

A pointer to the error provider from which the status header is to be obtained.

puResId

The address of a variable to receive the resource ID of the text of the response body corresponding to the status code and subcode.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

IsAsyncContinueStatus

Call this function to determine whether an ATL Server HTTP_CODE status code indicates that asynchronous processing is still in progress.

```
inline bool IsAsyncContinueStatus(
   HTTP_CODE hcStatus
);
```

Parameters

hcStatus

The status code being tested.

Return Value

Returns **TRUE** if the status code indicates that asynchronous processing is still in progress, **FALSE** otherwise.

Requirements

Header: atlserr.h

See Also

Reference

IsAsyncStatus
IsAsyncFlushStatus
IsAsyncNoFlushStatus
IsAsyncDoneStatus

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

IsAsyncDoneStatus

Call this function to determine whether an ATL Server status code indicates that asynchronous processing has completed.

```
inline bool IsAsyncDoneStatus(
   HTTP_CODE hcStatus
);
```

Parameters

hcStatus

The status code being tested.

Return Value

Returns **TRUE** if the status code indicates that asynchronous processing has completed, **FALSE** otherwise.

Requirements

Header: atlserr.h

See Also

Reference

IsAsyncStatus
IsAsyncFlushStatus
IsAsyncNoFlushStatus
IsAsyncContinueStatus
HTTP_CODE

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

IsAsyncFlushStatus

Call this function to determine whether a status code indicates that asynchronous processing has occurred and ATL Server code is responsible for flushing.

```
inline bool IsAsyncFlushStatus(
   HTTP_CODE hcStatus
);
```

Parameters

hcStatus

The status code being tested.

Return Value

Returns **TRUE** if the status code indicates that asynchronous processing has occurred and ATL Server code is responsible for flushing, **FALSE** otherwise.

Requirements

Header: atlserr.h

See Also

Reference

IsAsyncStatus
IsAsyncNoFlushStatus
IsAsyncDoneStatus
IsAsyncContinueStatus
HTTP_CODE

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

IsAsyncNoFlushStatus

Call this function to determine whether a status code indicates that asynchronous processing has occurred and ATL Server code is not responsible for flushing.

```
inline bool IsAsyncNoFlushStatus(
   HTTP_CODE hcStatus
);
```

Parameters

hcStatus

The status code being tested.

Return Value

Returns **TRUE** if the status code indicates that asynchronous processing has occurred and ATL Server code is not responsible for flushing, **FALSE** otherwise.

Requirements

Header: atlserr.h

See Also

Reference

IsAsyncStatus
IsAsyncFlushStatus
IsAsyncDoneStatus
IsAsyncContinueStatus
HTTP_CODE

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

IsAsyncStatus

Call this function to determine whether a status code indicates that asynchronous processing has occurred.

```
inline bool IsAsyncStatus(
   HTTP_CODE hcStatus
);
```

Parameters

hcStatus

The status code being tested.

Return Value

Returns **TRUE** if the status code indicates that asynchronous processing has occurred, **FALSE** otherwise.

Requirements

Header: atlserr.h

See Also

Reference

IsAsyncFlushStatus
IsAsyncNoFlushStatus
IsAsyncDoneStatus
IsAsyncContinueStatus
HTTP_CODE

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

IsExtendedChar

Call this function to find out if a given character is an extended character (less than 32, greater than 126, and not a tab, linefeed or carriage return)

```
inline int IsExtendedChar(
   char ch
) throw( );
```

Parameters

ch

The character to be tested

Return Value

TRUE if the character is extended, **FALSE** otherwise.

Requirements

Header: atlenc.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

IsNullByType

Specialize this function to define what it means for a type to be null when used with CValidateObject.

```
template <class TComp>
inline bool IsNullByType(
   TComp type
) throw( );
```

Parameters

TComp

The type of object for which 'nullness' is being defined.

type

The instance being tested for 'nullness'.

Return Value

Returns **TRUE** if *type* is to be considered 'null', **FALSE** otherwise.

Remarks

This function is used in CValidateObject to determine if an empty request parameter really should be empty. You can specialize this function in your own code such as the following specialization for type `long`:

```
template <>
inline bool IsNullByType<long>(long type) throw()
{
   return type == 0;
}
```

You should provide your own specialization for this function if the comparison of *type* == 0 is not adequate to discover whether or not an object is 'null'.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

OpenPerfMon

This function is an Open function that must be provided by performance extension DLLs that expose performance counters.

```
extern "C" ATL_NOINLINE
inline DWORD __declspec(dllexport) WINAPI OpenPerfMon(
   LPWSTR lpDeviceNames
) throw( );
```

Remarks

This method provides the expected functionality by using the performance monitoring macros and attributes that you use in your code.

Requirements

Header: atlperf.h

See Also

Reference

ClosePerfMon

CollectPerfMon

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

QEncode

Call this function to convert some data using the "Q" encoding.

```
inline BOOL QEncode(
   BYTE* pbSrcData,
   int nSrcLen,
   LPSTR szDest,
   int* pnDestLen,
   LPCSTR pszCharSet,
   int* pnNumEncoded = NULL
) throw( );
```

Parameters

pbSrcData

The buffer containing the data to be encoded.

nSrcLen

The length in bytes of the data to be encoded.

szDest

Caller-allocated buffer to receive the encoded data.

pnDestLen

Pointer to a variable that contains the length in characters of *szDest*. If the function succeeds, the variable receives the number of characters written to the buffer. If the function fails, the variable receives the required length in characters of the buffer.

pszCharSet

The character set to use for the conversion.

pnNumEncoded

A pointer to a variable that on return contains the number of unsafe characters that had to be converted.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

The "Q" encoding scheme is described in RFC 2047 (http://www.ietf.org/rfc/rfc2047.txt).

Requirements

Header: atlenc.h

See Also

Reference

QEncodeGetRequiredLength

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

QEncodeGetRequiredLength

Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.

```
inline int QEncodeGetRequiredLength(
   int nSrcLen,
   int nCharsetLen
) throw( );
```

Parameters

nSrcLen

The number of bytes of data to be encoded.

nCharsetLen

The length in characters of the character set to use for the conversion.

Return Value

The number of characters required for a buffer that could hold encoded data of *nSrcLen* bytes.

Remarks

The "Q" encoding scheme is described in RFC 2047 (http://www.ietf.org/rfc/rfc2047.txt).

Requirements

Header: atlenc.h

See Also

Reference

QEncode

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

QPDecode

Decodes a string of data that has been encoded in quoted-printable format such as by a previous call to QPEncode.

```
inline BOOL QPDecode(
   BYTE* pbSrcData,
   int nSrcLen,
   LPSTR szDest,
   int* pnDestLen,
   DWORD dwFlags = 0
) throw( );
```

Parameters

pbSrcData

The buffer containing the data to be decoded.

nSrcLen

The length in bytes of *pbSrcData*.

szDest

Caller-allocated buffer to receive the decoded data.

pnDestLen

Pointer to a variable that contains the length in bytes of *pbDest*. If the function succeeds, the variable receives the number of bytes written to the buffer. If the function fails, the variable receives the required length in bytes of the buffer.

dwFlags

Flags describing how the conversion is to be performed. See ATLSMTP_QPENCODE Flags.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

The quoted-printable encoding scheme is described in RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt).

Requirements

Header: atlenc.h

See Also

Reference

QPDecodeGetRequiredLength
QPEncode
QPEncodeGetRequiredLength

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

QPDecodeGetRequiredLength

Call this function to get the size in bytes of a buffer that could contain data decoded from quoted-printable-encoded string of the specified length.

```
inline int QPDecodeGetRequiredLength(
   int nSrcLen
) throw( );
```

Parameters

nSrcLen

The number of characters in the encoded string.

Return Value

The number of bytes required for a buffer that could hold a decoded string of *nSrcLen* characters.

Remarks

The quoted-printable encoding scheme is described in RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt).

Requirements

Header: atlenc.h

See Also

Reference

QPDecode
QPEncode
QPEncodeGetRequiredLength

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

QPEncode

Call this function to encode some data in quoted-printable format.

```
inline BOOL QPEncode(
   BYTE* pbSrcData,
   int nSrcLen,
   LPSTR szDest,
   int* pnDestLen,
   DWORD dwFlags = 0
) throw( );
```

Parameters

pbSrcData

The buffer containing the data to be encoded.

nSrcLen

The length in bytes of the data to be encoded.

szDest

Caller-allocated buffer to receive the encoded data.

pnDestLen

Pointer to a variable that contains the length in characters of *szDest*. If the function succeeds, the variable receives the number of characters written to the buffer. If the function fails, the variable receives the required length in characters of the buffer.

dwFlags

Flags describing how the conversion is to be performed. See ATLSMTP_QPENCODE Flags.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

The quoted-printable encoding scheme is described in RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt).

Requirements

Header: atlenc.h

See Also

Reference

QPDecode
QPDecodeGetRequiredLength
QPEncodeGetRequiredLength

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

QPEncodeGetRequiredLength

Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.

```
inline int QPEncodeGetRequiredLength(
   int nSrcLen
) throw( );
```

Parameters

nSrcLen

The number of bytes of data to be encoded.

Return Value

The number of characters required for a buffer that could hold encoded data of *nSrcLen* bytes.

Remarks

The quoted-printable encoding scheme is described in RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt).

Requirements

Header: atlenc.h

See Also

Reference

QPDecode
QPDecodeGetRequiredLength
QPEncode

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

ReadClientData

Call this function to read data from the client making the current request.

```
ATL_NOINLINE inline BOOL ReadClientData(
   IHttpServerContext * pServerContext,
   LPSTR pbDest,
   LPDWORD pdwLen,
   DWORD dwBytesRead
) throw( );
```

Parameters

pServerContext

The context of the current request.

pbDest

The caller-allocated buffer.

pdwLen

The address of a variable containing the number of bytes to read on entry to the function and the number of bytes read on exit.

dwBytesRead

The number of bytes previously read from the client.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

Typically, you will not need to call this function directly because the ATL Server framework deals with reading data from the client automatically.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

RenderError

Call this function to write an error response to the client.

```
template <class HttpUserErrorTextProvider>
void RenderError(
   IHttpServerContext * pServerContext,
   DWORD dwStatus,
   DWORD dwSubStatus,
   HttpUserErrorTextProvider* pErrorProvider
) throw( );
```

Parameters

HttpUserErrorTextProvider

A class providing the same public interface as CDefaultErrorProvider.

pServerContext

The server context.

dwStatus

The ID of the connection in the cache.

dwSubStatus

The service provider expected to expose **IDataSourceCache**.

pErrorProvider

A pointer to the object providing the text for the errors.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

RegisterPerfMon

Call this function to register performance objects and counters.

```
ATL_NOINLINE inline HRESULT RegisterPerfMon(
   HINSTANCE hDllInstance = _AtlBaseModule.GetModuleInstance()
) throw( );
```

Parameters

hDllInstance

The instance handle of the current DLL.

Return Value

Returns S_OK on success or an error HRESULT on failure.

Remarks

It is not necessary to call this function directly

Requirements

Header: atlperf.h

See Also

Reference

UnregisterPerfMon

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

RemoveDataSource

Call this function to close a cached connection and remove it from the data source cache.

```
static HRESULT ATL_NOINLINE RemoveDataSource(
   IServiceProvider * pProvider,
   LPCTSTR szID
);
```

Parameters

pProvider

The service provider expected to expose **IDataSourceCache**.

szID

The ID of the connection in the cache.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This wrapper function queries *pProvider* for the IDataSourceCache service and, if successful, calls IDataSourceCache::Remove using the remaining argument.

Note that closing the connection in the cache does not affect the connections retrieved from the cache because they are copies of the original.

Requirements

Header: atlcache.h

See Also

Reference

GetDataSource

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

RGBToHtml

Converts a COLORREF value to the HTML text corresponding to that color value.

```
bool inline RGBToHtml(
   COLORREF color,
   LPTSTR pbOut,
   long nBuffer
);
```

Parameters

color

An RGB color value.

pbOut

Caller-allocated buffer to receive the text for the HTML color value. The buffer must have space for at least 8 characters including space for the null terminator).

nBuffer

The size in bytes of the buffer (including space for the null terminator).

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

An HTML color value is a pound sign followed by a 6-digit hexadecimal value using 2 digits for each of the red, green, and blue components of the color (for example, #FFFFFF is white).

Requirements

Header: atlutil.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

RSkipSpace

Returns the pointer to the previous non-whitespace character.

```
inline LPCSTR RSkipSpace(
   LPCSTR pStart,
   LPCSTR sz,
   WORD nCodePage
) throw( );
```

Parameters

pStart

The start of the string.

sz

The current character in the string.

nCodePage

The code page used by the string.

Return Value

The previous non-whitespace character before *sz*.

Requirements

Header: atlstencil.h

See Also

Reference

SkipSpace

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

SafeStringCopy

Call this function to copy a string into a fixed size character buffer.

```
template <class T>
inline bool SafeStringCopy(
      T& Destination,
   const char* Source
) throw( );
```

Parameters

T

A fixed size character buffer of type `char[Size]` (not `char*`).

Destination

The buffer to which the data should be copied.

Source

The null-terminated string containing the data to be copied.

Return Value

Returns **TRUE** if all of *Source* was copied to *Destination*, **FALSE** if the data was truncated.

Remarks

The function is safe because it will not compile if type *T* is not a fixed size character buffer and if type *T* is a fixed size buffer, it guarantees not to copy more characters than the buffer is capable of holding. In addition, the destination buffer is always null-terminated on return from this function, even if the data was truncated.

Requirements

Header: atlstencil.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

SetRfc822Time

Call this function to get the current system time as a string in RFC 822 format.

```
inline size_t SetRfc822Time(
   LPSTR szDate,
      size_t dwLen
) throw( );
```

Parameters

szDate

Caller allocated buffer to which the current system time will be written in the format described in RFC 822 format (http://www.ietf.org/rfc/rfc822.txt).

dwLen

The size of the buffer in characters (including the space required for the terminating NULL).

Return Value

If *szDate* is not equal to NULL and the function succeeds, the return value is the number of characters copied to the buffer including the terminating NULL. If the function fails (this will happen if the buffer is not sufficiently large), the function returns 0.

If *szDate* is equal to NULL, the function returns the size of the buffer required to store the time string. The size includes the space necessary to store the terminating NULL.

Requirements

Header: atlmime.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

SkipSpace

Returns the pointer to the next non-whitespace character.

```
inline LPCSTR SkipSpace(
   LPCSTR sz,
   WORD nCodePage
) throw( );
```

Parameters

sz

The current character in the string.

nCodePage

The code page used by the string.

Return Value

The pointer to the next non-whitespace character.

Requirements

Header: atlstencil.h

See Also

Reference

RSkipSpace

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

SystemTimeToHttpDate

Call this function to convert a system time to a string in a format suitable for using in HTTP headers.

```
inline void SystemTimeToHttpDate(
   const SYSTEMTIME& st,
   CStringA & strTime
);
```

Parameters

st

The system time to be obtained as an HTTP format string.

strTime

A reference to a string variable to receive the HTTP date time as defined in RFC 2616 (http://www.ietf.org/rfc/rfc2616.txt) and RFC 1123 (http://www.ietf.org/rfc/rfc1123.txt).

Requirements

Header: atlutil.h

Example

See the Cookies Sample.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

UnregisterPerfMon

Call this function to unregister performance objects and counters.

```
ATL_NOINLINE inline HRESULT UnregisterPerfMon( ) throw( );
```

Return Value

Returns S_OK on success or an error HRESULT on failure.

Remarks

It is not necessary to call this function directly

Requirements

Header: atlperf.h

See Also

Reference

RegisterPerfMon

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Functions

UUDecode

Decodes a string of data that has been uuencoded such as by a previous call to UUEncode.

```
inline BOOL UUDecode(
   BYTE* pbSrcData,
   int nSrcLen,
   BYTE* pbDest,
   int* pnDestLen
) throw( );
```

Parameters

pbSrcData

The string containing the data to be decoded.

nSrcLen

The length in bytes of *pbSrcData*.

pbDest

Caller-allocated buffer to receive the decoded data.

pnDestLen

Pointer to a variable that contains the length in bytes of *pbDest*. If the function succeeds, the variable receives the number of bytes written to the buffer. If the function fails, the variable receives the required length in bytes of the buffer.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This uuencoding implementation follows the POSIX P1003.2b/D11 specification.

Requirements

Header: atlenc.h

See Also

Reference

UUDecodeGetRequiredLength
UUEncode
UUEncodeGetRequiredLength

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

UUDecodeGetRequiredLength

Call this function to get the size in bytes of a buffer that could contain data decoded from a uuencoded string of the specified length.

```
inline int UUDecodeGetRequiredLength(
   int nSrcLen
) throw( );
```

Parameters

nSrcLen

The number of characters in the encoded string.

Return Value

The number of bytes required for a buffer that could hold a decoded string of *nSrcLen* characters.

Remarks

This uuencoding implementation follows the POSIX P1003.2b/D11 specification.

Requirements

Header: atlenc.h

See Also

Reference

UUDecode
UUEncode
UUEncodeGetRequiredLength

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

UUEncode

Call this function to uuencode some data.

```
inline BOOL UUEncode(
   const BYTE* pbSrcData,
   int nSrcLen,
   LPSTR szDest,
   int* pnDestLen,
   LPCTSTR lpszFile = _T("file"),
   DWORD dwFlags = 0
) throw( );
```

Parameters

pbSrcData

The buffer containing the data to be encoded.

nSrcLen

The length in bytes of the data to be encoded.

szDest

Caller-allocated buffer to receive the encoded data.

pnDestLen

Pointer to a variable that contains the length in characters of *szDest*. If the function succeeds, the variable receives the number of characters written to the buffer. If the function fails, the variable receives the required length in characters of the buffer.

lpszFile

The file to be added to the header when ATLSMTP_UUENCODE_HEADER is specified in *dwFlags*.

dwFlags

Flags controlling the behavior of this function. See ATLSMTP_UUENCODE Flags.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This uuencoding implementation follows the POSIX P1003.2b/D11 specification.

Requirements

Header: atlenc.h

See Also

Reference

UUDecode
UUDecodeGetRequiredLength
UUEncodeGetRequiredLength

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

UUEncodeGetRequiredLength

Call this function to get the size in characters of a buffer that could contain a string encoded from data of the specified size.

```
inline int UUEncodeGetRequiredLength(
   int nSrcLen
) throw( );
```

Parameters

nSrcLen

The number of bytes of data to be encoded.

Return Value

The number of characters required for a buffer that could hold encoded data of *nSrcLen* bytes.

Remarks

This uuencoding implementation follows the POSIX P1003.2b/D11 specification.

Requirements

Header: atlenc.h

See Also

Reference

UUDecode
UUDecodeGetRequiredLength
UUEncode

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Functions

ATL Server Macros

The following macros are included in the ATL Server Library.

Name	Description
_ATL_DLLCACHE_MANAGEMENT	Define this macro to enable management services for the ISAPI extension's DLL cache.
_ATL_DLLCACHE_NOUI	Define this macro to remove the request handler implementing the HTML interface for managing the ISAPI extension's DLL cache.
_ATL_DLLCACHE_NOWEBSERVICE	Define this macro to remove the request handler implementing the SOAP interface for managing the ISAPI extension's DLL cache.
_ATL_PERF_REGISTER	This macro activates the registration of performance monitoring components with the Windows operating system.
_ATL_SOAP_NO_PARAMETER_VALIDATIONS	Define this macro to turn off SOAP parameter validation.
_ATL_STENCILCACHE_MANAGEMENT	Define this macro to enable management services for the ISAPI extension's stencil cache.
_ATL_STENCILCACHE_NOUI	Define this macro to remove the request handler implementing the HTML interface for managing the ISAPI extension's stencil cache.
_ATL_STENCILCACHE_NOWEBSERVICE	Define this macro to remove the request handler implementing the SOAP interface for managing the ISAPI extension's stencil cache.
_ATL_THREADPOOL_MANAGEMENT	Define this macro to enable management services for the ISAPI extension's thread pool.
_ATL_THREADPOOL_NOUI	Define this macro to remove the request handler implementing the HTML interface for managing the ISAPI extension's thread pool.
_ATL_THREADPOOL_NOWEBSERVICE	Define this macro to remove the request handler implementing the SOAP interface for managing the ISAPI extension's thread pool.
ATL_BASE64 Flags	These flags describe how base64-encoding is to be performed by Base64Encode.
ATL_BLOB_CACHE_TIMEOUT	This macro defines the cache maintenance frequency for the CBlobCache implementation.
ATL_CACHE_KEY_LENGTH	This macro defines the maximum length of a string used as a cache key.
ATL_CRITICAL_ISAPI_ERROR_LOGONLY	Define this macro to ensure that critical ISAPI extension errors are only logged in the Windows Event Log and are not returned as HTTP responses.
ATL_DEBUG_STENCILS	Define this macro to allow stencil-parsing errors found by CStencil-derived classes to be output when the stencil is rendered.

ATL_DEFAULT_AUTHGRP	This macro defines the group used for authorizing access to ATL Server extension management services. Only members of this group can use the Web-based management interfaces to configure the behavior of ATL Server ISAPI extension DLLs.
ATL_DEFAULT_DLL_EXTENSION	This macro defines the default file extension used to identify request handler DLLs.
ATL_DEFAULT_PRECISION	This macro defines the default precision used by CWriteStreamHelper for writing floating-point values.
ATL_DEFAULT_STENCIL_EXTENSION	This macro defines the default file extension used to identify server response files.
ATL_DLL_CACHE_TIMEOUT	This macro defines the default value in milliseconds used as the timeout for the DLL cache. The timeout value determines the frequency with which the DLL cache is swept for stale entries.
ATL_DS_CONN_STRING_LEN	This macro defines the maximum length of data source connection names.
ATL_EPSILON	This macro defines a small value used by CAtlValidator for comparing the equality of floating-point numbers.
ATL_ESC Flags	These flags are used to control the behavior of EscapeXML.
ATL_FILE_CACHE_TIMEOUT	This macro defines the cache maintenance frequency for the CFileCache implementation.
ATL_HTTP Flags	These flags provide information that can be used to modify the behavior of CAtlHttpClientT.
ATL_HTTP_AUTHTYPE_BASIC	This macro defines a string that can be used to indicate the BASIC HTTP authorization scheme.
ATL_HTTP_AUTHTYPE_NTLM	This macro defines a string that can be used to indicate the NTLM authorization scheme.
ATL_HTTP_DEFAULT_BLOCK_SIZE	This macro represents the default memory block sizes used for both incoming and out-going HTTP transaction buffers.
ATL_HTTP_FLAG_AUTO_REDIRECT	This macro is a behavior modification flag for HTTP request transactions that specifies whether redirections should be pursued automatically.
ATL_HTTP_FLAG_INVALID_FLAGS	This macro can be used to test for an invalid state of the HTTP behavior-control flags used by the CAtlNavigateData and CAtlHttpClientT classes.
ATL_HTTP_FLAG_PROCESS_RESULT	This macro is a behavior modification flag for HTTP request transactions that specifies whether additional actions, such as redirection or authorization, should be pursued.
ATL_HTTP_FLAG_SEND_BLOCKS	This macro is a behavior modification flag that indicates default HTTP request behavior.

ATL_HTTP_FLAG_SEND_CALLBACK	This macro is a behavior modification flag that specifies whether HTTP requests should be chunk-transfer encoded.
ATL_HTTP_METHOD_GET	This macro defines a string that indicates HTTP requests based on the GET method.
ATL_HTTP_METHOD_POST	This macro defines a string that indicates HTTP requests based on the POST method.
ATL_HTTP_PARAM_MAP_CASEINSENSITIVE	Define this macro to treat request parameters in a case insensitive manner.
ATL_HTTP_PARAM_MULTIMAP	Define this macro to allow multiple request parameters or form fields with the same name to be stored.
ATL_HTTP_USERAGENT	This macro defines the user agent string that is embedded into each request.
ATL_INVALID_STATUS	This macro defines a value that represents an invalid HTTP response status code.
ATL_ISAPI_BUFFER_SIZE	This macro defines the initial size in bytes of the static buffer used by CAtlIsapiBuffer.
ATL_MAX_BLOCK_STACK	This macro defines the maximum number of levels that can be used for nesting replacement tags.
ATL_MAX_HANDLER_NAME_LEN	This macro defines the maximum length that can be used for the alias of a subhandler in a subhandler tag.
ATL_MAX_METHOD_NAME_LEN	This macro defines the maximum length that can be used for the name of a replacement method.
ATL_NO_ACLAPI	Define this macro to prevent inclusion of the Access Control List API headers.
ATL_NO_CRITICAL_ISAPI_ERROR	Define this macro to prevent critical ISAPI extension errors being logged in the Windows Event Log.
ATL_NO_DEFAULT_AUTHORITY	Define this macro to disable or customize authorization of users of ATL Server extension management services.
ATL_NO_MLANG	Define this macro to prevent inclusion of mlang.h and use of the multilanguage object.
ATL_NO_MMSYS	Define this macro to prevent inclusion of the Multimedia API headers.
ATL_NO_SOAP	Define this macro to turn off any SOAP support.
ATL_PERF_CACHE_OBJECT	This macro defines the ID for the cache performance monitoring object definition.
ATL_SESSION_SWEEPER_TIMEOUT	This macro defines the frequency at which session data is maintained, in milliseconds.

ATL_SESSION_TIMEOUT	This macro defines the time, in milliseconds, after which an unaccessed s ession expires.
ATL_SHAREDBLOBCACHE_TIMEOUT	This macro defines the default value in 100-nanosecond intervals used a s a timeout by CSharedCache and CSharedCacheHandler.
ATL_SOCK_TIMEOUT	This macro defines the default value in milliseconds used as a timeout by CAtlHttpClient.
ATL_STENCIL_CACHE_TIMEOUT	This macro defines the default value in milliseconds used as the timeout for the stencil cache. The timeout value determines the frequency with w hich the stencil cache is swept for stale entries.
ATL_STENCIL_CHECK_TIMEOUT	This macro defines the frequency with which ATL Server checks cached s tencils against the raw stencil file to determine if the cached version is ou t of date.
ATL_STENCIL_LIFESPAN	This macro defines a default filetime value used as the life span for stenci ls in the stencil cache. Stencils are removed from the cache when their lif e span has been reached whether or not they have been used recently.
ATL_URL Flags	These flags modify the behavior of AtlEscapeUrl and AtlCanonicalizeUrl .
ATL_URL_DEFAULT_HTTP_PORT	This macro defines the default port used for HTTP transactions.
ATL_WORKER_THREAD_WAIT	This macro defines the default value in milliseconds that CWorkerThread::Shutdown will wait for the worker thread to shut down.
ATLPERF_DEFAULT_MAXINSTNAMELENGTH	Use this macro to define the maximum name length of object instance st rings.
ATLS_ASYNC_MUTEX_TIMEOUT	This macro defines the maximum timeout in milliseconds for the asynchr onous guard mutex.
ATLS_DEFAULT_THREADPOOLSHUTDOWNTIMEOUT	This macro defines the default time in milliseconds that CThreadPool will wait for a thread to shut down.
ATLS_DEFAULT_THREADSPERPROC	This macro defines the default number of threads per processor used by CThreadPool.
ATLS_ENABLE_DEBUGGING	Define this macro before including atlisapi.h to enable the debugger to a utomatically attach to the Web server in release builds.
ATLS_WORKER_HEAP_SIZE	This macro defines the initial size of the heap used by CIsapiWorker.
ATLSMTP_QPENCODE Flags	These flags describe how quoted-printable encoding is to be performed by QPEncode.
ATLSMTP_UUENCODE Flags	These flags describe how uuencoding is to be performed by UUEncode.
ATLSOAP_NOWININET	Define this macro before including atlsoap.h to ensure that including this header does not introduce a dependency on WinInet into your project.
BEGIN_COUNTER_MAP	Use this macro to begin performance counter definition maps within CPerfObject-derived classes.

BEGIN_HANDLER_MAP	Use this macro in a request handler DLL to mark the start of a handler map.
BEGIN_PERF_MAP	Use this macro to preface performance object map definitions.
BEGIN_REPLACEMENT_METHOD_MAP	Use this macro in a request handler class to mark the start of a replacement method map.
CHAIN_PERF_OBJECT	Use this macro to attach a performance object class to a performance object management class (a class derived from CPerfMon).
DECLARE_ASYNC_HANDLER_EX	Add this macro to the class definition of your request handler if you need to write to the client asynchronously.
DECLARE_ASYNC_HANDLER	Add this macro to the class definition of your request handler if you need to write to the client asynchronously.
DECLARE_PERF_OBJECT_EX	Use this macro to declare a new performance object from within a CPerfObject-derived class.
DECLARE_PERF_OBJECT_NO_INSTANCES	Use this macro to declare a new instanceless performance object from within a CPerfObject-derived class.
DECLARE_PERF_OBJECT	Use this macro to declare a new performance object from within a CPerfObject-derived class.
DECLARE_REQUEST_HANDLER	Add this macro to your project to map a request handler name to the class responsible for handling the corresponding requests.
DEFAULT_MAX_FORM_SIZE	This macro defines the default maximum form size in bytes.
DEFINE_COUNTER_EX	Use this macro to define performance monitor counters within CPerfMon-derived classes.
DEFINE_COUNTER	Use this macro to define performance monitor counters within CPerfMon-derived classes.
END_COUNTER_MAP	Use this macro to terminate performance counter map definitions.
END_HANDLER_MAP	Use this macro in a request handler DLL to mark the end of a handler map.
END_PERF_MAP	Use this macro to end a performance map.
END_REPLACEMENT_METHOD_MAP	Use this macro in a request handler class to mark the end of a replacement method map.
Handler Flags	These macros are used as flags to be returned from a request handler's GetHandlerFlags method.
HANDLER_ENTRY_EX	This macro is a synonym for the DECLARE_REQUEST_HANDLER macro.
HANDLER_ENTRY_SDL	Add this macro to your project to map a request handler name to the class responsible for handling the corresponding requests and expose the Web Service Description Language (WSDL) for the handler.

HANDLER_ENTRY	This macro is a synonym for the DECLARE_REQUEST_HANDLER macro.
HTTP_CODE Status Codes	These macros provide symbols for commonly used ATL Server status co des.
HTTP_CODE Subcodes	These macros are used as subcodes to control the actions that the ATL S erver framework takes when returning from a request handler method o r to distinguish one error code from another.
HTTP_ERROR_CODE	Use this macro to get the HTTP status code from an ATL Server status co de.
HTTP_ERROR	Use this macro to create an HTTP_CODE from an HTTP status code and a subcode.
HTTP_SUBERROR_CODE	Use this macro to get the subcode from an ATL Server status code.
ID_DLLCACHEMGR_SRFHANDLER_NAME	This constant defines the name of the request handler that implements t he HTML interface for managing the DLL cache.
ID_DLLCACHEMGR_WEBSERVICE_NAME	This constant defines the name of the XML Web service that exposes the SOAP interface for managing the DLL cache and the request handler that implements it.
ID_DLLCACHEMGR_WEBSERVICE_URL	This constant defines the namespace of the XML Web service that expose s the SOAP interface for managing the DLL cache.
ID_DLLCACHEMGR_WEBSERVICE_WSDL	This constant defines the name of the WSDL handler for the XML Web se rvice that exposes the SOAP interface for managing the DLL cache.
ID_STENCILCACHEMGR_SRFHANDLER_NAME	This constant defines the name of the request handler that implements t he HTML interface for managing the stencil cache.
ID_STENCILCACHEMGR_WEBSERVICE_NAME	This constant defines the name of the XML Web service that exposes the SOAP interface for managing the stencil cache and the request handler t hat implements it.
ID_STENCILCACHEMGR_WEBSERVICE_URL	This constant defines the namespace of the XML Web service that expose s the SOAP interface for managing the stencil cache.
ID_STENCILCACHEMGR_WEBSERVICE_WSDL	This constant defines the name of the WSDL handler for the XML Web se rvice that exposes the SOAP interface for managing the stencil cache.
ID_THREADMGR_SRFHANDLER_NAME	This constant defines the name of the request handler that implements t he HTML interface for managing the thread pool.
ID_THREADMGR_WEBSERVICE_NAME	This constant defines the name of the XML Web service that exposes the SOAP interface for managing the thread pool and the request handler th at implements it.
ID_THREADMGR_WEBSERVICE_URL	This constant defines the namespace of the XML Web service that expose s the SOAP interface for managing the thread pool.
ID_THREADMGR_WEBSERVICE_WSDL	This constant defines the name of the WSDL handler for the XML Web se rvice that exposes the SOAP interface for managing the thread pool.

IDR_DLLMGR_SRF	This constant defines the name of the stencil resource that provides the HTML interface for managing the DLL cache.
IDR_STENCILMGR_SRF	This constant defines the name of the stencil resource that provides the HTML interface for managing the stencil cache.
IDR_THREADMGR_SRF	This constant defines the name of the stencil resource that provides the HTML interface for managing the thread pool.
MAX_CONNECTION_STRING_LEN	This macro defines the maximum length of the data source connection st ring used for session-state data storage.
MAX_SESSION_KEY_LEN	This macro defines the maximum size allowed for a session name, in byt es.
MAX_TOKEN_LENGTH	This macro defines the maximum allowed size in bytes for the name of a n HTML form element that can be handled by ATL Server.
MAX_VARIABLE_NAME_LENGTH	This macro is defines the maximum size of session-state data variable na mes.
MAX_VARIABLE_VALUE_LENGTH	This macro is defines the maximum size of session-state data values.
NO_ATL_MGMT_STENCIL_WARNING	Define this macro before including atlextmgmt.h to prevent a message b eing displayed during compilation to remind you to include stencil resou rces for the HTML interface used by the extension management services.
PERFREG_ENTRY	Use this macro to register CPerfMon-derived classes with the Windows performance monitoring subsystem.
REPLACEMENT_METHOD_ENTRY_EX_STR	Use this macro in a replacement method map to associate a replacement method with a particular tag name. Use when the replacement method n eeds to receive method arguments in the form of a **char*** parameter.
REPLACEMENT_METHOD_ENTRY_EX	Use this macro in a replacement method map to associate a replacement method with a particular tag name. Use when the replacement method n eeds to receive method arguments.
REPLACEMENT_METHOD_ENTRY	Use this macro in a replacement method map to associate a replacement method with a particular tag name.
Request Initialization Flags	These macros are used as flags to be returned from a request handler's GetFlags method.
SESSION_COOKIE_NAME	This macro is used as the name of the cookie used for tracking session-st ate data.
Stencil Tokens	These constants identify different tokens recognized by the CStencil and CHtmlStencil classes.
STENCIL_USER_TOKEN_BASE	Use this value as the starting point for constants used to identify custom tokens in CStencil-derived classes.
Validation Results	These macros are returned as results from the Exchange or Validate met hods of CValidateObject.

VALIDATION_SUCCEEDED	Use this macro to determine whether validation succeeded or failed.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

_ATL_DLLCACHE_MANAGEMENT

Define this macro to enable management services for the ISAPI extension's DLL cache.

```
#ifdef _ATL_DLLCACHE_MANAGEMENT
```

Remarks

By default this symbol is not defined. Defining this symbol:

- Provides the definition for CDllMgrObject.
- Adds a request handler, CDllCacheMgr, to your module that implements the HTML management interface. This can be disabled by defining _ATL_DLLCACHE_NOUI.
- Adds a request handler, CDllCacheManager, to your module that implements the SOAP management interface, IDllCacheMgr. This can be disabled by defining _ATL_DLLCACHE_NOWEBSERVICE.

Note that both request handlers use ATL Server attributes, so **_ATL_ATTRIBUTES** must be defined for your project.

Requirements

Header: atlextmgmt.h

See Also

Reference

_ATL_DLLCACHE_NOWEBSERVICE
_ATL_DLLCACHE_NOUI

Concepts

Extension Management Services

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

_ATL_DLLCACHE_NOUI

Define this macro to remove the request handler implementing the HTML interface for managing the ISAPI extension's DLL cache.

```
#ifndef _ATL_DLLCACHE_NOUI
```

Remarks

Defining this symbol allows you to remove one of the request handlers added to your project by defining _ATL_DLLCACHE_MANAGEMENT.

Requirements

Header: atlextmgmt.h

See Also

Reference

_ATL_DLLCACHE_NOWEBSERVICE
_ATL_DLLCACHE_MANAGEMENT

Concepts

Extension Management Services

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

_ATL_DLLCACHE_NOWEBSERVICE

Define this macro to remove the request handler implementing the SOAP interface for managing the ISAPI extension's DLL cache.

```
#ifndef _ATL_DLLCACHE_NOWEBSERVICE
```

Remarks

Defining this symbol allows you to remove one of the request handlers added to your project by defining _ATL_DLLCACHE_MANAGEMENT.

Requirements

Header: atlextmgmt.h

See Also

Reference

_ATL_DLLCACHE_NOUI

_ATL_DLLCACHE_MANAGEMENT

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

_ATL_PERF_REGISTER

This macro activates the registration of performance monitoring components with the Windows operating system.

```
#ifdef _ATL_PERF_REGISTER
```

Remarks

By default, the _ATL_PERF_REGISTER macro is not defined, but it is added to the StdAfx.h file by the Performance Object Wizard when performance classes are added to a project. Without this macro, any performance objects and counters an application defines will not be exposed to the Windows performance monitoring subsystem.

See Also

Concepts

Performance Monitoring

Other Resources

ATL Server Reference

Performance Monitoring Reference

_ATL_SOAP_NO_PARAMETER_VALIDATIONS

Define this macro to turn off SOAP parameter validation.

```
#define _ATL_SOAP_NO_PARAMATER_VALIDATIONS
```

Remarks

In an XML Web service created with ATL Server, the default action is for SOAP parameters to be validated after they are read. Define this macro to turn off this validation.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

_ATL_STENCILCACHE_MANAGEMENT

Define this macro to enable management services for the ISAPI extension's stencil cache.

```
#ifdef _ATL_STENCILCACHE_MANAGEMENT
```

Remarks

By default this symbol is not defined. Defining this symbol:

- Provides the definition for CStencilCacheMgrObject.
- Adds a request handler, CStencilMgr, to your module that implements the HTML management interface. This can be disabled by defining _ATL_STENCILCACHE_NOUI.
- Adds a request handler, CStencilCacheManager, to your module that implements the SOAP management interface, IStencilCacheMgr. This can be disabled by defining _ATL_STENCILCACHE_NOWEBSERVICE.

Note that both request handlers use ATL Server attributes, so **_ATL_ATTRIBUTES** must be defined for your project.

Requirements

Header: atlextmgmt.h

See Also

Reference

_ATL_STENCILCACHE_NOWEBSERVICE
_ATL_STENCILCACHE_NOUI

Concepts

Extension Management Services

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

_ATL_STENCILCACHE_NOUI

Define this macro to remove the request handler implementing the HTML interface for managing the ISAPI extension's stencil cache.

```
#ifndef _ATL_STENCILCACHE_NOUI
```

Remarks

Defining this symbol allows you to remove one of the request handlers added to your project by defining _ATL_STENCILCACHE_MANAGEMENT.

Requirements

Header: atlextmgmt.h

See Also

Reference

_ATL_STENCILCACHE_NOWEBSERVICE
_ATL_STENCILCACHE_MANAGEMENT

Concepts

Extension Management Services

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

_ATL_STENCILCACHE_NOWEBSERVICE

Define this macro to remove the request handler implementing the SOAP interface for managing the ISAPI extension's stencil cache.

```
#ifndef _ATL_STENCILCACHE_NOWEBSERVICE
```

Remarks

Defining this symbol allows you to remove one of the request handlers added to your project by defining _ATL_STENCILCACHE_MANAGEMENT.

Requirements

Header: atlextmgmt.h

See Also

Reference

_ATL_STENCILCACHE_NOUI
_ATL_STENCILCACHE_MANAGEMENT

Concepts

Extension Management Services

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

_ATL_THREADPOOL_MANAGEMENT

Define this macro to enable management services for the ISAPI extension's thread pool.

```
#ifdef _ATL_THREADPOOL_MANAGEMENT
```

Remarks

By default this symbol is not defined. Defining this symbol:

- Provides the definition for CThreadPoolMgrObject.
- Adds a request handler, CThreadMgrStencil, to your module that implements the HTML management interface. This can be disabled by defining _ATL_THREADPOOL_NOUI.
- Adds a request handler, CThreadPoolManager, to your module that implements the SOAP management interface, IThreadPoolMgr. This can be disabled by defining _ATL_THREADPOOL_NOWEBSERVICE.

Note that both request handlers use ATL Server attributes, so **_ATL_ATTRIBUTES** must be defined for your project.

Requirements

Header: atlextmgmt.h

See Also

Reference

_ATL_THREADPOOL_NOWEBSERVICE
_ATL_THREADPOOL_NOUI

Concepts

Extension Management Services

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

_ATL_THREADPOOL_NOUI

Define this macro to remove the request handler implementing the HTML interface for managing the ISAPI extension's thread pool.

```
#ifndef _ATL_THREADPOOL_NOUI
```

Remarks

Defining this symbol allows you to remove one of the request handlers added to your project by defining _ATL_THREADPOOL_MANAGEMENT.

Requirements

Header: atlextmgmt.h

See Also

Reference

_ATL_THREADPOOL_NOWEBSERVICE
_ATL_THREADPOOL_MANAGEMENT

Concepts

Extension Management Services

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

_ATL_THREADPOOL_NOWEBSERVICE

Define this macro to remove the request handler implementing the SOAP interface for managing the ISAPI extension's thread pool.

```
#ifndef _ATL_THREADPOOL_NOWEBSERVICE
```

Remarks

Defining this symbol allows you to remove one of the request handlers added to your project by defining _ATL_THREADPOOL_MANAGEMENT.

Requirements

Header: atlextmgmt.h

See Also

Reference

_ATL_THREADPOOL_NOUI

_ATL_THREADPOOL_MANAGEMENT

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ATL_BASE64 Flags

These flags describe how base64-encoding is to be performed by Base64Encode.

```
#define ATL_BASE64_FLAG_NONE
#define ATL_BASE64_FLAG_NOPAD
#define ATL_BASE64_FLAG_NOCRLF
```

Remarks

Flag	Description
ATL_BASE64_FLAG_NONE	Default behavior. The encoded data is padded and line broken.
ATL_BASE64_FLAG_NOPAD	Padding is not added to the encoded data.
ATL_BASE64_FLAG_NOCRLF	Lines are not broken in the encoded data.

The base64 encoding scheme is described in RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt).

Requirements

Header: atlisapi.h

See Also

Reference

Base64Encode

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ATL_BLOB_CACHE_TIMEOUT

This macro defines the cache maintenance frequency for the CBlobCache implementation.

```
#define ATL_BLOB_CACHE_TIMEOUT 1000
```

Remarks

This macro determines the rate at which CBlobCache performs automatic maintenance of the cache such as purging expired items. Unless redefined, this macro resolves to 1,000 milliseconds (1 second).

Requirements

Header: atlcache.h

See Also

Other Resources

Caching
ATL Server Reference
Caching Reference

ATL_CACHE_KEY_LENGTH

This macro defines the maximum length of a string used as a cache key.

```
#define ATL_CACHE_KEY_LENGTH 128
```

Remarks

This macro is used to define the CFixedStringKey type, which serves as the default cache key type for ATL Server cache implementations. The **ATL_CACHE_KEY_LENGTH** macro, unless redefined, resolves to 128, limiting cache keys to 128 characters including the null terminator. If necessary, you can define your own integer literal value for this symbol before including atlcache.h

Requirements

Header: atlcache.h

See Also

Other Resources

Caching
ATL Server Reference
Caching Reference

ATL_CRITICAL_ISAPI_ERROR_LOGONLY

Define this macro to ensure that critical ISAPI extension errors are only logged in the Windows Event Log and are not returned as HTTP responses.

```
#ifdef ATL_CRITICAL_ISAPI_ERROR_LOGONLY
```

Remarks

By default, critical ISAPI extension errors encountered during processing of CIsapiExtension::GetExtensionVersion are logged to the Windows Event Log and are returned as the response to any requests made against that ISAPI extension.

Define ATL_CRITICAL_ISAPI_ERROR_LOGONLY to prevent the ISAPI extension returning error information as an HTTP response while still allowing the error to be logged to the Event Log.

Requirements

Header: atlisapi.h

See Also

Reference

ATL_NO_CRITICAL_ISAPI_ERROR

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ATL_DEBUG_STENCILS

Define this macro to allow stencil-parsing errors found by CStencil-derived classes to be output when the stencil is rendered.

```
#ifdef ATL_DEBUG_STENCILS
```

Remarks

To allow stencil-parsing errors found by CStencil-derived classes to be output when the stencil is rendered, you can define this symbol before including atlstencil.h.

Requirements

Header: atlstencil.h

See Also

Reference

CStencil Class

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ATL_DEFAULT_AUTHGRP

This macro defines the group used for authorizing access to ATL Server extension management services. Only members of this group can use the Web-based management interfaces to configure the behavior of ATL Server ISAPI extension DLLs.

```
#define ATL_DEFAULT_AUTHGRP
```

Remarks

This macro is defined to be Sids::Admins(), so, by default, only members of the local Administrators group are allowed to manage ATL Server ISAPI extensions using the extension management services. If necessary, you can define your own value for this symbol before including atlextmgmt.h. Examples of possible definitions are shown:

Definition	Allows Access To
`#define ATL_DEFAULT_AUTHGRP CSid(_T("ATLSAdmins"))`	Members of the **ATLSAdmins** group.
`#define ATL_DEFAULT_AUTHGRP Sids::World()`	Everyone. (Not recommended for production code.)
`#define ATL_DEFAULT_AUTHGRP Sids::Null()`	No one.

Requirements

Header: atlextmgmt.h

See Also

Reference

CSid Class

ATL_NO_DEFAULT_AUTHORITY

Concepts

Extension Management Services

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ATL_DEFAULT_DLL_EXTENSION

This macro defines the default file extension used to identify request handler DLLs.

```
#define ATL_DEFAULT_DLL_EXTENSION
```

Remarks

This macro is defined to be the string literal **".dll"**. If necessary, you can define your own string literal value for this symbol before including atlisapi.h.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATL_DEFAULT_PRECISION

This macro defines the default precision used by CWriteStreamHelper for writing floating-point values.

```
#define ATL_DEFAULT_PRECISION
```

Remarks

This macro is defined to be the integer literal **6**, so six decimal places will be used by default. If necessary, you can define your own integer literal value for this symbol before including atlisapi.h.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATL_DEFAULT_STENCIL_EXTENSION

This macro defines the default file extension used to identify server response files.

```
#define ATL_DEFAULT_STENCIL_EXTENSION
```

Remarks

This macro is defined to be the string literal **".srf"**. If necessary, you can define your own string literal value for this symbol before including atlisapi.h.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATL_DLL_CACHE_TIMEOUT

This macro defines the default value in milliseconds used as the timeout for the DLL cache. The timeout value determines the frequency with which the DLL cache is swept for stale entries.

```
#define ATL_DLL_CACHE_TIMEOUT
```

Remarks

ATL_DLL_CACHE_TIMEOUT defaults to 1 second in debug builds and 10 minutes in release builds. If necessary, you can define your own value for this symbol before including atlcache.h.

Requirements

Header: atlcache.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATL_DS_CONN_STRING_LEN

This macro defines the maximum length of data source connection names.

```
#define ATL_DS_CONN_STRING_LEN 512
```

Remarks

The **ATL_DS_CONN_STRING_LEN** macro, unless redefined, resolves to 512, limiting data source connection strings to 512 characters including the null terminator.

Requirements

Header: atlcache.h

See Also

Other Resources

Caching
ATL Server Reference
Caching Reference

ATL_EPSILON

This macro defines a small value used by CAtlValidator for comparing the equality of floating-point numbers.

```
#define ATL_EPSILON
```

Remarks

If necessary, you can define your own value for this symbol before including atlisapi.h.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATL_ESC Flags

These flags are used to control the behavior of EscapeXML.

```
#define ATL_ESC_FLAG_NONE
#define ATL_ESC_FLAG_ATTR
```

Remarks

Flag	Description
ATL_ESC_FLAG_NONE	Default behavior. Quote marks and apostrophes are not converted.
ATL_ESC_FLAG_ATTR	Quote marks and apostrophes are converted to " and ' respectively.

Requirements

Header: atlisapi.h

See Also

Reference

EscapeXML

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ATL_FILE_CACHE_TIMEOUT

This macro defines the cache maintenance frequency for the CFileCache implementation.

```
#define ATL_BLOB_CACHE_TIMEOUT 1000
```

Remarks

This macro determines the rate at which CFileCache performs automatic maintenance of the cache such as purging expired items. Unless redefined, this macro resolves to 1,000 milliseconds (10 seconds).

Requirements

Header: atlcache.h

See Also

Other Resources

Caching
ATL Server Reference
Caching Reference

ATL_HTTP Flags

These flags provide information that can be used to modify the behavior of CAtlHttpClientT.

The following symbols can be used:

Flags	Description
ATL_HTTP_FLAG_AUTO_REDIRECT	Indicates that redirections should be handled automatically.
ATL_HTTP_FLAG_PROCESS_RESULT	Indicates that responses requiring security credentials or indicating redirections should be pursued. If this flag is not set, the ATL_HTTP_FLAG_AUTO_REDIRECT flag has no effect.
ATL_HTTP_FLAG_SEND_CALLBACK	Request data is to be supplied by a user-defined callback function. If this flag is set, a callback function must be specified within the **ATL_NAVIGATE_DATA** structure.
ATL_HTTP_FLAG_SEND_BLOCKS	Request data will be sent out in blocks, the size of which is specified within the **ATL_NAVIGATE_DATA** structure.

The ATL_HTTP_FLAG_SEND_BLOCKS and ATL_HTTP_FLAG_SEND_CALLBACK flags are mutually exclusive.

Requirements

Header: atlhttp.h

See Also

Reference

CAtlHttpClientT Class

CAtlNavigateData Class

ATL_NAVIGATE_DATA::dwFlags

Concepts

HTTP Client Services

Other Resources

ATL Server Reference

HTTP Client Reference

ATL_HTTP_AUTHTYPE_BASIC

This macro defines a string that can be used to indicate the BASIC HTTP authorization scheme.

```
#define ATL_HTTP_AUTHTYPE_BASIC _T("BASIC")
```

Remarks

The ATL_HTTP_AUTHTYPE_BASIC macro, with the ATL_HTTP_AUTHTYPE_NTLM macro, are provided to represent the two HTTP authorization schemes natively supported by ATL Server. These macros can be used with the CAtlHttpClientT::AddAuthObj method.

Requirements

Header: atlhttp.h

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

ATL Server Reference

HTTP Client Reference

ATL_HTTP_AUTHTYPE_NTLM

This macro defines a string that can be used to indicate the NTLM authorization scheme.

```
#define ATL_HTTP_AUTHTYPE_NTLM _T("NTLM")
```

Remarks

The ATL_HTTP_AUTHTYPE_NTLM macro, with the ATL_HTTP_AUTHTYPE_BASIC macro, are provided to represent the two HTTP authorization schemes natively supported by ATL Server. These macros might typically be used with the CAtlHttpClientT::AddAuthObj method.

Requirements

Header: atlhttp.h

See Also

Reference

CAtlHttpClientT Class

Concepts

HTTP Client Services

Other Resources

ATL Server Reference

HTTP Client Reference

ATL_HTTP_DEFAULT_BLOCK_SIZE

This macro represents the default memory block sizes used for both incoming and out-going HTTP transaction buffers.

```
#define ATL_HTTP_DEFAULT_BLOCK_SIZE 4096
```

Remarks

ATL Server uses the ATL_HTTP_DEFAULT_BLOCK_SIZE macro to determine the size of the memory buffers used for HTTP send and receive operations. The size of these buffers can be modified on a case-by-case basis using the CAtlNavigateData::SetSendBlockSize and CAtlNavigateData::SetReadBlockSize methods, or globally by defining the ATL_HTTP_DEFAULT_BLOCK_SIZE macro before including the Atlhttp.h header file.

Requirements

Header: atlhttp.h

See Also

Reference

CAtlHttpClientT Class

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

ATL Server Reference

HTTP Client Reference

ATL_HTTP_FLAG_AUTO_REDIRECT

This macro is a behavior modification flag for HTTP request transactions that specifies whether redirections should be pursued automatically.

```
#define ATL_HTTP_FLAG_AUTO_REDIRECT
```

Remarks

If this flag is set, the CAtlHttpClientT class will automatically handle any redirections encountered during the processing of an HTTP request. The ATL_HTTP_FLAG_AUTO_REDIRECT flag is set by default.

Use the CAtlNavigateData::AddFlags,RemoveFlags, GetFlags, and SetFlags methods to manipulate the state of this flag.

Requirements

Header: atlhttp.h

See Also

Reference

CAtlHttpClientT Class

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

ATL Server Reference

HTTP Client Reference

ATL_HTTP_FLAG_INVALID_FLAGS

This macro can be used to test for an invalid state of the HTTP behavior-control flags used by the CAtlNavigateData and CAtlHttpClientT classes.

```
#define ATL_HTTP_FLAG_INVALID_FLAGS
```

Remarks

Use the CAtlNavigateData::AddFlags,RemoveFlags, GetFlags, and SetFlags methods to manipulate the state of the behavior-control flags.

Requirements

Header: atlhttp.h

See Also

Concepts

HTTP Client Services

Other Resources

ATL Server Reference

HTTP Client Reference

ATL_HTTP_FLAG_PROCESS_RESULT

This macro is a behavior modification flag for HTTP request transactions that specifies whether additional actions, such as redirection or authorization, should be pursued.

```
#define ATL_HTTP_FLAG_PROCESS_RESULT
```

Remarks

If this flag is set, the CAtlHttpClientT class will attempt to handle cases where redirection or authorization is required. The ATL_HTTP_FLAG_PROCESS_RESULT flag is set by default. If this flag is not set, redirections will not be processed, regardless of the state of the ATL_HTTP_FLAG_AUTO_REDIRECT flag.

Use the CAtlNavigateData::AddFlags,RemoveFlags, GetFlags, and SetFlags methods to manipulate the state of this flag.

Requirements

Header: atlhttp.h

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

ATL Server Reference

HTTP Client Reference

ATL_HTTP_FLAG_SEND_BLOCKS

This macro is a behavior modification flag that indicates default HTTP request behavior.

```
#define ATL_HTTP_FLAG_SEND_BLOCKS
```

Remarks

This flag is set by default and specifies that the CAtlHttpClientT class should perform HTTP requests normally. Alternately, the ATL_HTTP_FLAG_SEND_CALLBACK flags can be used to specify chunk-transfer encoded requests. The ATL_HTTP_FLAG_SEND_CALLBACK and ATL_HTTP_FLAG_SEND_BLOCKS flags are mutually exclusive.

Use the CAtlNavigateData::AddFlags,RemoveFlags, GetFlags, and SetFlags methods to manipulate the state of this flag.

Requirements

Header: atlhttp.h

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

ATL Server Reference

HTTP Client Reference

ATL_HTTP_FLAG_SEND_CALLBACK

This macro is a behavior modification flag that specifies whether HTTP requests should be chunk-transfer encoded.

```
#define ATL_HTTP_FLAG_SEND_CALLBACK
```

Remarks

If this flag is set, the CAtlHttpClientT class will use chunk-transfer encoding for subsequent requests, and a request data callback must be specified using the CAtlNavigateData::SetChunkCallback method. By default the ATL_HTTP_FLAG_SEND_BLOCKS flag is set, which indicates normal request encoding by default.

Use the CAtlNavigateData::AddFlags,RemoveFlags, GetFlags, and SetFlags methods to manipulate the state of this flag.

Requirements

Header: atlhttp.h

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

ATL Server Reference

HTTP Client Reference

ATL_HTTP_METHOD_GET

This macro defines a string that indicates HTTP requests based on the GET method.

```
#define ATL_HTTP_METHOD_GET _T("GET")
```

Remarks

By default the CAtlHttpClientT class uses the GET method to perform HTTP requests. Alternately, the ATL_HTTP_METHOD_POST macro can be used to specify POST-based requests.

Use the CAtlNavigateData::GetMethod and CAtlNavigateData::SetMethod methods to modify the request method.

Requirements

Header: atlhttp.h

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

ATL Server Reference

HTTP Client Reference

ATL_HTTP_METHOD_POST

This macro defines a string that indicates HTTP requests based on the POST method.

```
#define ATL_HTTP_METHOD_POST _T("POST")
```

Remarks

By default the CAtlHttpClientT class uses the ATL_HTTP_METHOD_GET macro, but the ATL_HTTP_METHOD_POST macro can be used to specify POST-based requests.

Use the CAtlNavigateData::GetMethod and CAtlNavigateData::SetMethod methods to modify the request method.

Requirements

Header: atlhttp.h

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

ATL Server Reference

HTTP Client Reference

ATL_HTTP_PARAM_MAP_CASEINSENSITIVE

Define this macro to treat request parameters in a case insensitive manner.

```
#ifdef ATL_HTTP_PARAM_MAP_CASEINSENSITIVE
```

Remarks

See CHttpRequestParams.

Requirements

Header: atlisapi.h

See Also

Reference

CHttpRequestParams Class

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ATL_HTTP_PARAM_MULTIMAP

Define this macro to allow multiple request parameters or form fields with the same name to be stored.

```
#ifdef ATL_HTTP_PARAM_MULTIMAP
```

Remarks

See CHttpRequestParams.

Requirements

Header: atlisapi.h

See Also

Reference

CHttpRequestParams Class

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ATL_HTTP_USERAGENT

This macro defines the user agent string that is embedded into each request.

```
#define ATL_HTTP_USERAGENT
```

Remarks

By default, the ATL_HTTP_USERAGENT macro is defined as:

```
"User-Agent: Microsoft-ATL-Native/8.00\r\n"
```

By defining the ATL_HTTP_USERAGENT symbol before including the atlhttp.h header file, the user agent string can be customized.

Requirements

Header: atlhttp.h

See Also

Concepts

HTTP Client Services

Other Resources

ATL Server Reference

HTTP Client Reference

ATL_INVALID_STATUS

This macro defines a value that represents an invalid HTTP response status code.

```
#define ATL_INVALID_STATUS
```

Remarks

The return value of the CAtlHttpClientT::GetStatus method can be compared against this macro to determine if the HTTP server has returned an invalid response code.

Requirements

Header: atlhttp.h

See Also

Reference

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

ATL Server Reference

HTTP Client Reference

ATL_ISAPI_BUFFER_SIZE

This macro defines the initial size in bytes of the static buffer used by CAtlIsapiBuffer.

```
#define ATL_ISAPI_BUFFER_SIZE
```

Remarks

If necessary, you can define your own value for this symbol before including atlutil.h. See CAtlIsapiBuffer for a discussion on when you might wish to do this.

Requirements

Header: atlutil.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATL_MAX_BLOCK_STACK

This macro defines the maximum number of levels that can be used for nesting replacement tags.

```
#define ATL_MAX_BLOCK_STACK
```

Remarks

If necessary, you can define your own value for this symbol before including atlstencil.h.

Requirements

Header: atlstencil.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros
ATL Server Response File Reference

ATL_MAX_HANDLER_NAME_LEN

This macro defines the maximum length that can be used for the alias of a subhandler in a subhandler tag.

```
#define ATL_MAX_HANDLER_NAME_LEN
```

Remarks

If necessary, you can define your own value for this symbol before including atlisapi.h.

Requirements

Header: atlisapi.h

See Also

Reference

ATL_MAX_METHOD_NAME_LEN

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ATL_MAX_METHOD_NAME_LEN

This macro defines the maximum length that can be used for the name of a replacement method.

```
#define ATL_MAX_METHOD_NAME_LEN
```

Remarks

If necessary, you can define your own value for this symbol before including atlstencil.h.

Requirements

Header: atlstencil.h

See Also

Reference

ATL_MAX_HANDLER_NAME_LEN

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ATL_NO_ACLAPI

Define this macro to prevent inclusion of the Access Control List API headers.

```
#ifndef ATL_NO_ACLAPI
```

Remarks

To prevent inclusion of the Access Control List API headers, you can define this symbol before including atlisapi.h.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATL_NO_CRITICAL_ISAPI_ERROR

Define this macro to prevent critical ISAPI extension errors being logged in the Windows Event Log.

```
#ifndef ATL_NO_CRITICAL_ISAPI_ERROR
```

Remarks

Critical ISAPI extension errors are indicated by calling CIsapiExtension::SetCriticalIsapiError. This method may be called indirectly during processing of CIsapiExtension::GetExtensionVersion or directly by the user.

By default, **SetCriticalIsapiError** logs the error to the Windows Event Log, makes the error code available to subsequent callers of CIsapiExtension::GetCriticalIsapiError, and returns **TRUE**. The result of this is that CIsapiExtension::GetExtensionVersion will succeed if an error occurs, but all responses returned by CIsapiExtension::HttpExtensionProc will consist of an error message describing the failure.

When ATL_NO_CRITICAL_ISAPI_ERROR is defined, **SetCriticalIsapiError** returns **FALSE**. This error code is also returned by CIsapiExtension::GetExtensionVersion so no responses of any kind will be available from the ISAPI extension DLL.

Requirements

Header: atlisapi.h

See Also

Reference

ATL_CRITICAL_ISAPI_ERROR_LOGONLY

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ATL_NO_DEFAULT_AUTHORITY

Define this macro to disable or customize authorization of users of ATL Server extension management services.

```
#ifndef ATL_NO_DEFAULT_AUTHORITY
```

Remarks

By default, this symbol is not defined and users of ATL Server extension management services are authorized using the **_Authority** object (of type CDefaultAuth) and the group defined by ATL_DEFAULT_AUTHGRP.

By defining this symbol without providing a definition for **_Authority**, you can remove the authorization code from the extension management services. This is not recommended for production sites since it allows any user to change important aspects of the functioning of your ISAPI extension DLL.

If this symbol is defined and a definition for **_Authority** is provided, you can customize the authorization process for users of the extension management services.

Requirements

Header: atlextmgmt.h

See Also

Reference

CDefaultAuth Class

ATL_DEFAULT_AUTHGRP

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ATL_NO_MLANG

Define this macro to prevent inclusion of mlang.h and use of the multilanguage object.

```
#ifndef ATL_NO_MLANG
```

Remarks

To prevent inclusion of mlang.h and use of the multilanguage object, you can define this symbol before including atlstencil.h. If this symbol is defined, your code will not have a dependency on the multilanguage object, however, you will not be able to specify the codepage tag or locale tag as strings in your server response files.

Requirements

Header: atlstencil.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATL_NO_MMSYS

Define this macro to prevent inclusion of the Multimedia API headers.

```
#ifndef ATL_NO_MMSYS
```

Remarks

To prevent inclusion of the Multimedia API headers and use of the multimedia system timer, you can define this symbol before including atlisapi.h. If this symbol is defined, time to handle a request will be measured using GetTickCount instead of timeGetTime.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATL_NO_SOAP

Define this macro to turn off any SOAP support.

```
#ifndef ATL_NO_SOAP
```

Remarks

To prevent inclusion of the msxml2.h header and use of **ISAXXMLReader** which is required for SOAP support, you can define this symbol before including atlisapi.h.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATL_PERF_CACHE_OBJECT

This macro defines the ID for the cache performance monitoring object definition.

```
#define ATL_PERF_CACHE_OBJECT 100
```

Remarks

By default, the ATL Server cache classes used the CStdStatClass class for statistics gathering. Alternately, the CPerfStatClass class can be used to generate performance statistics. This causes a performance monitoring object definition to be exposed through the Windows performance monitoring subsystem. This macro defines the object ID of the caching performance object. See Performance Monitoring for more information.

Requirements

Header: atlcache.h

See Also

Other Resources

Caching
ATL Server Reference
Caching Reference

ATL_SESSION_SWEEPER_TIMEOUT

This macro defines the frequency at which session data is maintained, in milliseconds.

```
#define ATL_SESSION_SWEEPER_TIMEOUT
```

Remarks

ATL Server sweeps session data periodically, primarily to check timeout vales and expire session data if necessary. Define the **ATL_SESSION_SWEEPER_TIMEOUT** symbol before including atlsession.h to override the default sweep interval of 1000, or 1 second.

Requirements

Header: atlsession.h

See Also

Other Resources

Session-State Services (ATL Server)
ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATL_SESSION_TIMEOUT

This macro defines the time, in milliseconds, after which an un-accessed session expires.

```
#define ATL_SESSION_TIMEOUT
```

Remarks

ATL Server periodically sweeps session data, checking time elapsed since each session was last accessed. **The ATL_SESSION_TIMEOUT** macro defines the session timeout interval. If necessary, you can define your own value for this symbol before including atlsession.h. Define the **ATL_SESSION_TIMEOUT** symbol before including atlsession.h to override the default session timeout interval of 600,000, or 10 minutes.

Requirements

Header: atlsession.h

See Also

Other Resources

Session-State Services (ATL Server)
ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATL_SHAREDBLOBCACHE_TIMEOUT

This macro defines the default value in 100-nanosecond intervals used as a timeout by CSharedCache and CSharedCacheHandler.

```
#define ATL_SHAREDBLOBCACHE_TIMEOUT
```

Remarks

ATL_SHAREDBLOBCACHE_TIMEOUT defaults to 1 hour. If necessary, you can define your own value for this symbol before including atlsharedsvc.h.

Requirements

Header: atlsharedsvc.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATL_SOCK_TIMEOUT

This macro defines the default value in milliseconds used as a timeout by CAtlHttpClient.

```
#define ATL_SOCK_TIMEOUT
```

Remarks

ATL_SOCK_TIMEOUT defaults to 10 seconds. If necessary, you can define your own value for this symbol before including atlspriv.h. You may want to do this if you are testing HTTP client applications such as XML Web service clients and intend to pause in the debugger for a length of time. To disable the timeout completely, define this macro as shown:

```
#define ATL_SOCK_TIMEOUT INFINITE
```

Requirements

Header: atlspriv.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATL_STENCIL_CACHE_TIMEOUT

This macro defines the default value in milliseconds used as the timeout for the stencil cache. The timeout value determines the frequency with which the stencil cache is swept for stale entries.

```
#define ATL_STENCIL_CACHE_TIMEOUT
```

Remarks

ATL_STENCIL_CACHE_TIMEOUT defaults to 1 second. If necessary, you can define your own value for this symbol before including atlcache.h.

Requirements

Header: atlcache.h

See Also

Reference

ATL_STENCIL_LIFESPAN

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ATL_STENCIL_CHECK_TIMEOUT

This macro defines the frequency with which ATL Server checks cached stencils against the raw stencil file to determine if the cached version is out of date.

```
#define ATL_STENCIL_CHECK_TIMEOUT 1000
```

Remarks

The CIsapiExtension class uses the CStencilCache class to cache preprocessed stencils. This macro determines how often stencils retrieved from the cache are checked for obsolescence. If the raw stencil file from which the cached stencil was derived has changed, or cannot be opened, the **CIsapiExtension** class removes the cached stencil. Unless redefined, this macro resolves to 1,000 milliseconds (10 seconds).

Requirements

Header: atlcache.h

See Also

Other Resources

Caching
ATL Server Reference
Caching Reference

ATL_STENCIL_LIFESPAN

This macro defines a default filetime value used as the life span for stencils in the stencil cache. Stencils are removed from the cache when their life span has been reached whether or not they have been used recently.

```
#define ATL_STENCIL_LIFESPAN
```

Remarks

ATL_STENCIL_LIFESPAN defaults to 1 second in debug builds and 1 hour in release builds. If necessary, you can define your own value for this symbol before including atlcache.h.

Requirements

Header: atlcache.h

See Also

Reference

ATL_STENCIL_CACHE_TIMEOUT

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ATL_URL Flags

These flags modify the behavior of AtlEscapeUrl and AtlCanonicalizeUrl .

```
#define ATL_URL_ESCAPE
#define ATL_URL_NO_ENCODE
#define ATL_URL_DECODE
#define ATL_URL_NO_META
#define ATL_URL_ENCODE_SPACES_ONLY
#define ATL_URL_BROWSER_MODE
#define ATL_URL_ENCODE_PERCENT
```

Remarks

Flag	Description
ATL_URL_BROWSER_MODE	Does not encode or decode characters after "#" or "?", and does not remove trailing white space after "?". If this value is not specified, the entire URL is encoded and trailing white space is removed.
ATL_URL_DECODE	Converts all %XX sequences to characters, including escape sequences, before the URL is parsed.
ATL_URL_ENCODE_PERCENT	Encodes any percent signs encountered. By default, percent signs are not encoded.
ATL_URL_ENCODE_SPACES_ONLY	Encodes spaces only.
ATL_URL_ESCAPE	Converts all escape sequences (%XX) to their corresponding characters.
ATL_URL_NO_ENCODE	Does not convert unsafe characters to escape sequences.
ATL_URL_NO_META	Does not remove meta sequences (such as "." and "..") from the URL.

Requirements

Header: atlutil.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATL_URL_DEFAULT_HTTP_PORT

This macro defines the default port used for HTTP transactions.

```
#define ATL_URL_DEFAULT_HTTP_PORT
```

Remarks

ATL Server uses this macro to define a default HTTP port number of 80. See AtlGetDefaultUrlPort for more information on default port values.

Requirements

Header: atlutil.h

See Also

Reference

CAtlHttpClientT Class

CAtlNavigateData Class

Concepts

HTTP Client Services

Other Resources

ATL Server Reference

HTTP Client Reference

ATL_WORKER_THREAD_WAIT

This macro defines the default value in milliseconds that CWorkerThread::Shutdown will wait for the worker thread to shut down.

```
#define ATL_WORKER_THREAD_WAIT
```

Remarks

ATL_WORKER_THREAD_WAIT defaults to 10 seconds. If necessary, you can define your own value for this symbol before including atlutil.h.

Requirements

Header: atlutil.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATLPERF_DEFAULT_MAXINSTNAMELENGTH

Use this macro to define the maximum name length of object instance strings.

```
#define ATLPERF_DEFAULT_MAXINSTNAMELENGTH 64
```

Remarks

The default maximum instance string length can be overridden by defining the ATLPERF_DEFAULT_MAXINSTANCENAMELENGTH symbol to the desired value, before including the Atlperf.h header file. If this symbol is already defined, ATL Server respects the initial value.

Requirements

Header: atlperf.h

See Also

Concepts

Performance Monitoring

Other Resources

ATL Server Reference

Performance Monitoring Reference

ATLS_ASYNC_MUTEX_TIMEOUT

This macro defines the maximum timeout in milliseconds for the asynchronous guard mutex.

```
#define ATLS_ASYNC_MUTEX_TIMEOUT
```

Remarks

ATLS_ASYNC_MUTEX_TIMEOUT defaults to 10000 milliseconds (10 seconds). If necessary, you can define your own value for this symbol before including atlisapi.h.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATLS_DEFAULT_THREADPOOLSHUTDOWNTIMEOUT

This macro defines the default time in milliseconds that CThreadPool will wait for a thread to shut down.

```
#define ATLS_DEFAULT_THREADPOOLSHUTDOWNTIMEOUT
```

Remarks

The default time is 36 seconds. If necessary, you can define your own positive integer value for this symbol before including atlutil.h.

To set the timeout at run time or for a particular object, call CThreadPool::SetTimeout.

Requirements

Header: atlutil.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATLS_DEFAULT_THREADSPERPROC

This macro defines the default number of threads per processor used by CThreadPool.

```
#define ATLS_DEFAULT_THREADSPERPROC
```

Remarks

The default is 2 threads per processor. If necessary, you can define your own positive integer value for this symbol before including atlutil.h.

Requirements

Header: atlutil.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATLS_ENABLE_DEBUGGING

Define this macro before including atlisapi.h to enable the debugger to automatically attach to the Web server in release builds.

```
#ifdef ATLS_ENABLE_DEBUGGING
```

Remarks

Define this macro only to ease the debugging process of release builds. Do not leave the macro defined in production code.

See Automatically Attaching To The Web Server Process for more information.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATLS_SAXXMLREADER_CLSID

Defines a CLSID for a SAX XML reader.

```
#define ATLS_SAXXMLREADER_CLSID __uuidof(SAXXMLReader)
```

Remarks

ATLS_SAXXMLREADER_CLSID defines a CLSID for a SAX XML reader (SAXXMLReader). Use this define to specify a SAX XML reader CLSID, or override this define if you want to use a different CLSID for the SAX XML reader.

(The SAX XML reader is like the XmlReader class (see "Comparing XmlReader to SAX Reader"), which reads XML data from a stream or file. The difference is that SAX reader uses a push model while the XmlReader uses a pull model.)

Example

The following code creates an instance of a SAX XML reader and generates an error message if it fails:

```
CComPtr<ISAXXMLReader> spReader;
if (pReader != NULL)
{
   spReader = pReader;
}
else
{
   if (FAILED(spReader.CoCreateInstance(ATLS_SAXXMLREADER_CLSID, NULL, CLSCTX_INPROC_SERVER
)))
   {
   ATLTRACE( _T("ATLSOAP: CSoapFault::ParseFault -- CoCreateInstance of SAXXMLReader failed
.\r\n" ) );
   return E_FAIL;
   }
}
```

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATLS_WORKER_HEAP_SIZE

This macro defines the initial size of the heap used by CIsapiWorker.

```
#define ATLS_WORKER_HEAP_SIZE
```

Remarks

By default, the per-thread heap created by CIsapiWorker is 16 KB. If necessary, you can define your own unsigned integer literal value for this symbol before including atlisapi.h to change the initial size of the heap. Note that the heap will grow as necessary without intervention.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

ATLSMTP_QPENCODE Flags

These flags describe how quoted-printable encoding is to be performed by QPEncode.

```
#define ATLSMTP_QPENCODE_DOT
#define ATLSMTP_QPENCODE_TRAILING_SOFT
```

Remarks

Flag	Description
ATLSMTP_QPENCODE_DOT	If a period appears at the start of a line, it is added to the output as well as encoded.
ATLSMTP_QPENCODE_TRAILING_SOFT	Appends =\r\n to the encoded string.

The quoted-printable encoding scheme is described in RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt).

Requirements

Header: atlisapi.h

See Also

Reference

QPEncode

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ATLSMTP_UUENCODE Flags

These flags describe how uuencoding is to be performed by UUEncode.

```
#define ATLSMTP_UUENCODE_HEADER
#define ATLSMTP_UUENCODE_END
#define ATLSMTP_UUENCODE_DOT
```

Remarks

Flag	Description
ATLSMTP_UUENCODE_HEADER	The header will be encoded.
ATLSMTP_UUENCODE_END	The end will be encoded.
ATLSMTP_UUENCODE_DOT	Data stuffing will be performed.

Requirements

Header: atlisapi.h

See Also

Reference

UUEncode

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ATLSOAP_NOWININET

Define this macro before including atlsoap.h to ensure that including this header does not introduce a dependency on WinInet into your project.

```
#ifdef ATLSOAP_NOWININET
```

Remarks

Defining this macro makes the CReadStreamOnInet and CSoapWininetClient classes unavailable.

Requirements

Header: atlsoap.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

BEGIN_COUNTER_MAP

Use this macro to begin performance counter definition maps within CPerfObject-derived classes.

```
#define BEGIN_COUNTER_MAP(
      objectclass
)
```

Parameters

objectclass

The name of the CPerfObject-derived class currently being defined.

Remarks

The BEGIN_COUNTER_MAP macro, together with the DEFINE_COUNTER, and END_COUNTER_MAP macros, defines performance counters within performance object classes. These macros are used implicitly by the performance monitor attributes, so they do not need to appear in attributed code.

Requirements

Header: atlperf.h

Example

See the PerformanceCounter Sample and the PerformanceScribble Sample.

See Also

Concepts

Performance Monitoring

Other Resources

ATL Server Reference

Performance Monitoring Reference

BEGIN_HANDLER_MAP

Use this macro in a request handler DLL to mark the start of a handler map.

```
#define BEGIN_HANDLER_MAP
```

Remarks

This macro has no functional use; it is merely supplied as a notational convenience.

The handler map is used to map the names of request handlers (as specified in server response files) to the C++ classes that implement those handlers.

A handler map is marked by the **BEGIN_HANDLER_MAP** and END_HANDLER_MAP macros.

Entries in the map can be provided by any of the following macros:

- HANDLER_ENTRY
- HANDLER_ENTRY_EX
- HANDLER_ENTRY_SDL

Requirements

Header: atlisapi.h

Example

See the MantaWeb Sample, the ConfirmUser Sample, and the ShowImage Sample.

See Also

Reference

END_HANDLER_MAP
HANDLER_ENTRY
HANDLER_ENTRY_SDL

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

BEGIN_PERF_MAP

Use this macro to preface performance object map definitions.

```
#define BEGIN_PERF_MAP(
      AppName
)
```

Parameters

AppName

A string containing a name for the performance object management class being declared.

Remarks

The BEGIN_PERF_MAP macro is typically followed by one or more performance object entries using the CHAIN_PERF_OBJECT macro and a closing END_PERF_MAP macro. Performance map definitions are only valid from within CPerfMon derived classes. These macros are used implicitly by the performance monitor attributes, so they do not need to appear in attributed code.

Requirements

Header: atlperf.h

Example

See the PerformanceCounter Sample and the PerformanceScribble Sample.

See Also

Concepts

Performance Monitoring

Other Resources

ATL Server Reference

Performance Monitoring Reference

BEGIN_REPLACEMENT_METHOD_MAP

Use this macro in a request handler class to mark the start of a replacement method map.

```
#define BEGIN_REPLACEMENT_METHOD_MAP(
      className
)
```

Parameters

className

The name of the C++ class containing the replacement method map.

Remarks

A replacement method map is used to map the names of replacement methods (as specified in server response files) to the C++ functions that implement those methods.

A replacement method map is marked by the **BEGIN_REPLACEMENT_METHOD_MAP** and END_REPLACEMENT_METHOD_MAP macros.

Entries in the map can be provided by any of the following macros:

- REPLACEMENT_METHOD_ENTRY
- REPLACEMENT_METHOD_ENTRY_EX
- REPLACEMENT_METHOD_ENTRY_EX_STR

Requirements

Header: atlstencil.h

Example

See the MantaWeb Sample, the ConfirmUser Sample, and the ShowImage Sample.

See Also

Reference

END_REPLACEMENT_METHOD_MAP
REPLACEMENT_METHOD_ENTRY
REPLACEMENT_METHOD_ENTRY_EX
REPLACEMENT_METHOD_ENTRY_EX_STR

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

CHAIN_PERF_OBJECT

Use this macro to attach a performance object class to a performance object management class (a class derived from CPerfMon).

```
#define CHAIN_PERF_OBJECT(
      objectclass
)
```

Parameters

objectclass

A string containing the name of the class representing the performance object.

Remarks

The CHAIN_PERF_OBJECT macro, which must by bracketed by the BEGIN_PERF_MAP and END_PERF_MAP macros, is valid only from within the definition of a CPerfMon derived class. The resulting performance object management, or performance monitor class, provides system registration and management for performance objects. These macros are used implicitly by the performance monitor attributes, so they do not need to appear in attributed code.

Requirements

Header: atlperf.h

Example

See the PerformanceCounter Sample and the PerformanceScribble Sample.

See Also

Concepts

Performance Monitoring

Other Resources

ATL Server Reference

Performance Monitoring Reference

DECLARE_ASYNC_HANDLER

Add this macro to the class definition of your request handler if you need to write to the client asynchronously.

```
#define DECLARE_ASYNC_HANDLER( )
```

Remarks

This macro adds to your class an implementation of GetHandlerFlags that returns ATLS_FLAG_ASYNC and an implementation of GetAsyncFlags that returns ATLSRV_INIT_USEASYNC.

Requirements

Header: atlisapi.h

Example

See the PooledAsync Sample.

See Also

Reference

DECLARE_ASYNC_HANDLER_EX

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

DECLARE_ASYNC_HANDLER_EX

Add this macro to the class definition of your request handler if you need to write to the client asynchronously.

```
#define DECLARE_ASYNC_HANDLER_EX( )
```

Remarks

This macro adds to your class an implementation of GetHandlerFlags that returns ATLS_FLAG_ASYNC and an implementation of GetAsyncFlags that returns ATLSRV_INIT_USEASYNC | ATLSRV_INIT_USEASYNC_EX.

Requirements

Header: atlisapi.h

See Also

Reference

DECLARE_ASYNC_HANDLER

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

DECLARE_PERF_OBJECT

Use this macro to declare a new performance object from within a CPerfObject-derived class.

```
#define DECLARE_PERF_OBJECT(
      objectclass,
      dwObjectId,
      namestring,
      helpstring,
   defcounter
)
```

Parameters

objectclass

The name of the CPerfObject-derived class being defined.

dwObjectId

A numeric ID that uniquely represents each performance object contained within the CPerfMon derived class. This value is used to identify the object type when calling CPerfMon::CreateInstance or CPerfMon::CreateInstanceByName.

namestring

The name that is displayed for this object within the Performance Monitoring Console. Can be provided as a string or a resource ID.

helpstring

A help string that is displayed for this object within the Performance Monitoring Console. Can be provided as a string or a string resource ID.

defcounter

A zero-based index of the counter that should be selected by default when viewing objects of this type, or -1 if there is no default counter.

Remarks

This macro, with the BEGIN_COUNTER_MAP, DEFINE_COUNTER, and END_COUNTER_MAP macros, is used to define performance monitoring objects. The DECLARE_PERF_OBJECT macro should appear before the counter map. These macros are used implicitly by the performance monitor attributes, so they do not need to appear in attributed code.

Requirements

Header: atlperf.h

Example

See the PerformanceCounter Sample and the PerformanceScribble Sample.

See Also

Concepts

Performance Monitoring

Other Resources

ATL Server Reference

Performance Monitoring Reference

DECLARE_PERF_OBJECT_EX

Use this macro to declare a new performance object from within a CPerfObject-derived class.

```
#define DECLARE_PERF_OBJECT_EX(
      dwObjectId,
      namestring,
      helpstring,
      detail,
      instanceless,
      structsize,
      maxinstnamelen,
      defcounter
)
```

Parameters

dwObjectId

A numeric ID that uniquely represents each performance object contained within the CPerfMon-derived class. This value is used to identify the object type when calling CPerfMon::CreateInstance or CPerfMon::CreateInstanceByName.

namestring

The name that is displayed for this object within the Performance Monitoring Console. Can be provided as a string or a resource ID.

helpstring

A help string that is displayed for this object within the Performance Monitoring Console. Can be provided as a string or a string resource ID.

detail

The detail level of this counter. This argument can be one of following values:

Detail level	Description
PERF_DETAIL_NOVICE	Indicates that this counter may be meaningful to most users. This is the most common counter de tail level.
PERF_DETAIL_ADVANCED	Indicates that this counter is likely to be useful only to advanced users.
PERF_DETAIL_EXPERT	Indicates that this counter is likely to be useful only to the most advanced users.
PERF_DETAIL_WIZARD	Indicates that this counter is not likely to be useful to any users.

instanceless

Zero if performance objects are to be tracked by instance, otherwise **PERF_NO_INSTANCE**.

structsize

The size of the class being declared, in bytes. Typically, the sizeof operator is used to calculate this value.

maxinstnamelen

The maximum length of the instance name string that can be given to CPerfMon::CreateInstance or CPerfMon::CreateInstanceByName.

defcounter

A zero-based index of the counter that should be selected by default when viewing objects of this type, or -1 if there is no default counter.

Remarks

This macro, with the BEGIN_COUNTER_MAP, DEFINE_COUNTER, and END_COUNTER_MAP macros, is used to define performance monitoring objects. The DECLARE_PERF_OBJECT_EX macro should appear before the counter map. For most applications, the simpler DECLARE_PERF_OBJECT macro will suffice. Attributed code can make use of the perfmon and perf_object attributes instead of these macros.

Requirements

Header: atlperf.h

See Also

Concepts

Performance Monitoring

Other Resources

ATL Server Reference

Performance Monitoring Reference

DECLARE_PERF_OBJECT_NO_INSTANCES

Use this macro to declare a new instanceless performance object from within a CPerfObject-derived class.

```
#define DECLARE_PERF_OBJECT_NO_INSTANCES(
      objectclass,
      dwObjectId,
      namestring,
      helpstring
)
```

Parameters

objectclass

The name of the CPerfObject-derived classes being declared.

dwObjectId

A numeric ID that uniquely represents each performance object contained within the CPerfMon-derived class. This value is used to identify the object type when calling CPerfMon::CreateInstance or CPerfMon::CreateInstanceByName.

namestring

The name that is displayed for this object within the Performance Monitoring Console. Can be provided as a string or a resource ID.

helpstring

A help string that is displayed for this object within the Performance Monitoring Console. Can be provided as a string or a string resource ID.

Remarks

This macro, with the BEGIN_COUNTER_MAP, DEFINE_COUNTER, and END_COUNTER_MAP macros, allows performance objects to be defined. The DECLARE_PERF_OBJECT_NO_INSTANCES macro should appear before the counter map. For performance objects that track performance data per application instance, use the DECLARE_PERF_OBJECT macro. These macros are used implicitly by the performance monitor attributes, so they do not need to appear in attributed code.

Requirements

Header: atlperf.h

See Also

Concepts

Performance Monitoring

Other Resources

ATL Server Reference

Performance Monitoring Reference

DECLARE_REQUEST_HANDLER

Add this macro to your project to map a request handler name to the class responsible for handling the corresponding requests.

```
#define DECLARE_REQUEST_HANDLER(
      handlerName,
      className,
      classQName
)
```

Parameters

handlerName

A string with the name of the request handler. Corresponds to the *request_handler* string that would be used in a handler tag.

className

The simple name of the request handler class that maps to the name given by *handlerName*.

classQName

The fully qualified name of the request handler class that maps to the name given by *handlerName*.

Remarks

You can add this macro to your code at any point after the class specified by *className* has been declared. The class does not need to be fully defined at the point at which this macro appears.

This macro is equivalent in functionality to the request_handler attribute. *className* and *classQName* are equivalent to the class to which the **request_handler** attribute is applied. *handlerName* is equivalent to the *name* parameter used by the attribute.

Requirements

Header: atlisapi.h

See Also

Reference

Handler Tag
request_handler
HANDLER_ENTRY
HANDLER_ENTRY_SDL
HANDLER_ENTRY_EX

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

DEFAULT_MAX_FORM_SIZE

This macro defines the default maximum form size in bytes.

```
#define DEFAULT_MAX_FORM_SIZE
```

Remarks

Defaults to 48 KB. If necessary, you can define your own value for this symbol before including atlisapi.h.

Requirements

Header: atlisapi.h

See Also

Reference

CRequestHandlerT::MaxFormSize

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

DEFINE_COUNTER

Use this macro to define performance monitor counters within CPerfMon-derived classes.

```
#define DEFINE_COUNTER(
      member,
      namestring,
      helpstring,
      countertype,
      defscale
)
```

Parameters

member

The data member representing the performance counter.

namestring

A name for this counter that will be displayed in the Performance Monitoring Console. Can be either a string or a string resource ID.

helpstring

A help string that is displayed for this object within the Performance Monitoring Console. Can be provided as a string or a string resource ID.

countertype

The counter type. For more information, see Counter Data Types.

defscale

Indicates the power of 10 by which to multiply the counter value before displaying the counter. For example, if you set *defscale* to 2, the counter value is multiplied by 100. If you set *defscale* to –3, the counter value is divided by 1,000.

Remarks

The DEFINE_COUNTER macro should appear between the BEGIN_COUNTER_MAP and END_COUNTER_MAP macros. These macros define counter definition maps that must appear within CPerfObject derived classes. These macros are used implicitly by the performance monitor attributes, so they do not need to appear in attributed code.

Requirements

Header: atlperf.h

Example

See the PerformanceCounter Sample and the PerformanceScribble Sample.

See Also

Reference

DEFINE_COUNTER_EX

Concepts

Performance Monitoring

Other Resources

ATL Server Reference

Performance Monitoring Reference

DEFINE_COUNTER_EX

Use this macro to define performance monitor counters within CPerfMon-derived classes.

```
#define DEFINE_COUNTER_EX(
      member,
      dwCounterId,
      namestring,
      helpstring,
      detail,
      countertype,
      maxcountersize,
      defscale
)
```

Parameters

member

The data member representing the performance counter.

dwCounterId

A numeric ID representing the counter, typically zero.

namestring

A name for this counter that will be displayed in the Performance Monitoring Console. Can be either a string or a string resource ID.

helpstring

A help string that is displayed for this object within the Performance Monitoring Console. Can be provided as a string or a string resource ID.

detail

The detail level of this counter. This argument can be one of following values:

Detail level	Description
PERF_DETAIL_NOVICE	Indicates that this counter may be meaningful to most users. This is the most common counter de tail level.
PERF_DETAIL_ADVANCED	Indicates that this counter is likely to be useful only to advanced users.
PERF_DETAIL_EXPERT	Indicates that this counter is likely to be useful only to the most advanced users.
PERF_DETAIL_WIZARD	Indicates that this counter is not likely to be useful to any users.

countertype

The counter type. For more information, see Counter Data Types.

maxcountersize

Indicates the maximum amount of space to reserve for the string data of a string type counter. Must be 0 for integral counter types.

defscale

Indicates the power of 10 by which to multiply the counter value before displaying the counter. For example, if you set *defscale* to 2, the counter value is multiplied by 100. If you set *defscale* to –3, the counter value is divided by 1,000.

Remarks

In most situations, the simpler DEFINE_COUNTER macro can be used instead. In either case, these macros should appear between the BEGIN_COUNTER_MAP and END_COUNTER_MAP macros. Together, this family of macros defines counter definition maps that must appear within CPerfObject derived class definitions. Used implicitly by the performance monitor attributes, these macros do not need to appear in attributed code.

Requirements

Header: atlperf.h

See Also

Reference

DEFINE_COUNTER

Concepts

Performance Monitoring

Performance Monitoring

Other Resources

ATL Server Reference

Performance Monitoring Reference

END_COUNTER_MAP

Use this macro to terminate performance counter map definitions.

```
#define END_COUNTER_MAP
```

Remarks

The END_COUNTER_MAP macro indicates the end of a performance counter map starting with the BEGIN_COUNTER_MAP macro and containing one or more DEFINE_COUNTER macro usages. Performance counter maps defined with these macros must reside within CPerfObject derived classes. This macro is used implicitly by the performance monitor attributes, so it does not need to appear in attributed code.

Requirements

Header: atlperf.h

Example

See the PerformanceCounter Sample and the PerformanceScribble Sample.

See Also

Concepts

Performance Monitoring

Other Resources

ATL Server Reference

Performance Monitoring Reference

END_HANDLER_MAP

Use this macro in a request handler DLL to mark the end of a handler map.

```
#define END_HANDLER_MAP
```

Remarks

See BEGIN_HANDLER_MAP for details.

Requirements

Header: atlisapi.h

Example

See the ConfirmUser Sample and the ShowImage Sample.

See Also

Reference

BEGIN_HANDLER_MAP

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

END_PERF_MAP

Use this macro to end a performance map.

```
#define END_PERF_MAP
```

Remarks

The END_PERF_MAP macro terminates a performance object map started with the BEGIN_PERF_MAP macro. The actual performance map entries are typically defined using the CHAIN_PERF_OBJECT macro. Performance maps are only valid within CPerfMon-derived classes. This macro is used implicitly by the performance monitor attributes, so it does not need to appear in attributed code.

Requirements

Header: atlperf.h

Example

See the PerformanceCounter Sample and the PerformanceScribble Sample.

See Also

Concepts

Performance Monitoring

Other Resources

ATL Server Reference

Performance Monitoring Reference

END_REPLACEMENT_METHOD_MAP

Use this macro in a request handler class to mark the end of a replacement method map.

```
#define END_REPLACEMENT_METHOD_MAP( )
```

Remarks

See BEGIN_REPLACEMENT_METHOD_MAP for details.

Requirements

Header: atlstencil.h

Example

See the ConfirmUser Sample and the ShowImage Sample.

See Also

Reference

BEGIN_REPLACEMENT_METHOD_MAP

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

Handler Flags

These macros are used as flags to be returned from a request handler's GetHandlerFlags method.

```
#define ATLS_FLAG_NONE
#define ATLS_FLAG_ASYNC
```

Remarks

Flag	Description
ATLS_FLAG_NONE	Default behavior
ATLS_FLAG_ASYNC	The handler might write to the client asynchronously.

Requirements

Header: atlisapi.h

See Also

Concepts

Writing Asynchronous Web Applications

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

HANDLER_ENTRY

This macro is a synonym for the DECLARE_REQUEST_HANDLER macro.

```
#define HANDLER_ENTRY(
      handlerName,
      className
)
```

Remarks

The macro assumes that *className* is the fully qualified name of the class.

Requirements

Header: atlisapi.h

Example

See the MantaWeb Sample, the ConfirmUser Sample, and the ShowImage Sample.

See Also

Reference

HANDLER_ENTRY_SDL
HANDLER_ENTRY_EX
DECLARE_REQUEST_HANDLER

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

HANDLER_ENTRY_EX

This macro is a synonym for the DECLARE_REQUEST_HANDLER macro.

```
#define HANDLER_ENTRY_EX(
      handlerName,
      className,
      classQName
)
```

Requirements

Header: atlisapi.h

See Also

Reference

HANDLER_ENTRY
HANDLER_ENTRY_SDL
DECLARE_REQUEST_HANDLER

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

HANDLER_ENTRY_SDL

Add this macro to your project to map a request handler name to the class responsible for handling the corresponding requests and expose the Web Service Description Language (WSDL) for the handler.

```
#define HANDLER_ENTRY_SDL(
      handlerString,
      handlerClass,
      handlerQClass,
      sdlClassName
)
```

Parameters

handlerString

A string with the name of the request handler.

handlerClass

The simple name of an existing request handler class that maps to the name given by *handlerName*.

handlerQClass

The fully qualified name of an existing request handler class that maps to the name given by *handlerName*.

sdlClassName

The name of a request handler class that will be created for you. This class will be used to expose WSDL information for the XML Web service request handler *handlerClass*.

Remarks

You can add this macro to your code at any point after the class specified by *handlerClass* has been declared. The class does not need to be fully defined at the point at which this macro appears.

This macro works like DECLARE_REQUEST_HANDLER but also adds the code necessary to expose the XML Web service through WSDL.

This macro is equivalent in functionality to the request_handler attribute. *handlerClass* and *handlerQClass* are equivalent to the class to which the **request_handler** attribute is applied. *handlerString* is equivalent to the *name* parameter used by the attribute. *sdlClassName* is equivalent to the *sdl* parameter used by the attribute.

Requirements

Header: atlisapi.h

See Also

Reference

HANDLER_ENTRY
HANDLER_ENTRY_EX
DECLARE_REQUEST_HANDLER

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

HTTP_CODE Status Codes

These macros provide symbols for commonly used ATL Server status codes.

```
#define HTTP_SUCCESS                    0
#define HTTP_FAIL                       HTTP_ERROR(500, SUBERR_NONE)
#define HTTP_SUCCESS_NO_PROCESS         HTTP_ERROR(200, SUBERR_NO_PROCESS)
#define HTTP_S_FALSE                    HTTP_ERROR(HTTP_ERROR_CODE(HTTP_SUCCESS), SUBERR_S_F
ALSE)
#define HTTP_SUCCESS_ASYNC              HTTP_ERROR(200, SUBERR_ASYNC)
#define HTTP_SUCCESS_ASYNC_DONE         HTTP_ERROR(200, SUBERR_ASYNC_DONE)
#define HTTP_SUCCESS_ASYNC_NOFLUSH      HTTP_ERROR(200, SUBERR_ASYNC_NOFLUSH)
#define HTTP_SUCCESS_ASYNC_NOFLUSH_DONE HTTP_ERROR(200, SUBERR_ASYNC_NOFLUSH_DONE)
#define HTTP_SUCCESS_NO_CACHE           HTTP_ERROR(200, SUBERR_NO_CACHE)
#define HTTP_OK                         HTTP_ERROR(200, SUBERR_NONE)
#define HTTP_CONTINUE                   HTTP_ERROR(100, SUBERR_NONE)

#define HTTP_CREATED                    HTTP_ERROR(201, SUBERR_NONE)
#define HTTP_ACCEPTED                   HTTP_ERROR(202, SUBERR_NONE)
#define HTTP_NON_AUTHORITATIVE          HTTP_ERROR(203, SUBERR_NONE)
#define HTTP_NO_CONTENT                 HTTP_ERROR(204, SUBERR_NONE)
#define HTTP_RESET_CONTENT              HTTP_ERROR(205, SUBERR_NONE)
#define HTTP_PARTIAL_CONTENT            HTTP_ERROR(206, SUBERR_NONE)

#define HTTP_MULTIPLE_CHOICES           HTTP_ERROR(300, SUBERR_NONE)
#define HTTP_MOVED_PERMANENTLY          HTTP_ERROR(301, SUBERR_NONE)
#define HTTP_FOUND                      HTTP_ERROR(302, SUBERR_NONE)
#define HTTP_SEE_OTHER                  HTTP_ERROR(303, SUBERR_NONE)
#define HTTP_NOT_MODIFIED               HTTP_ERROR(304, SUBERR_NONE)
#define HTTP_USE_PROXY                  HTTP_ERROR(305, SUBERR_NONE)
#define HTTP_TEMPORARY_REDIRECT         HTTP_ERROR(307, SUBERR_NONE)

#define HTTP_BAD_REQUEST                HTTP_ERROR(400, SUBERR_NONE)
#define HTTP_UNAUTHORIZED               HTTP_ERROR(401, SUBERR_NONE)
#define HTTP_PAYMENT_REQUIRED           HTTP_ERROR(402, SUBERR_NONE)
#define HTTP_FORBIDDEN                  HTTP_ERROR(403, SUBERR_NONE)
#define HTTP_NOT_FOUND                  HTTP_ERROR(404, SUBERR_NONE)
#define HTTP_METHOD_NOT_ALLOWED         HTTP_ERROR(405, SUBERR_NONE)
#define HTTP_NOT_ACCEPTABLE             HTTP_ERROR(406, SUBERR_NONE)
#define HTTP_PROXY_AUTHENTICATION_REQUIRED    HTTP_ERROR(407, SUBERR_NONE)
#define HTTP_REQUEST_TIMEOUT            HTTP_ERROR(408, SUBERR_NONE)
#define HTTP_CONFLICT                   HTTP_ERROR(409, SUBERR_NONE)
#define HTTP_GONE                       HTTP_ERROR(410, SUBERR_NONE)
#define HTTP_LENGTH_REQUIRED            HTTP_ERROR(411, SUBERR_NONE)
#define HTTP_PRECONDITION_FAILED        HTTP_ERROR(412, SUBERR_NONE)
#define HTTP_REQUEST_ENTITY_TOO_LONG    HTTP_ERROR(413, SUBERR_NONE)
#define HTTP_REQUEST_URI_TOO_LONG       HTTP_ERROR(414, SUBERR_NONE)
#define HTTP_UNSUPPORTED_MEDIA_TYPE     HTTP_ERROR(415, SUBERR_NONE)
#define HTTP_RANGE_NOT_SATISFIABLE      HTTP_ERROR(416, SUBERR_NONE)
#define HTTP_EXPECTATION_FAILED         HTTP_ERROR(417, SUBERR_NONE)

#define HTTP_INTERNAL_SERVER_ERROR      HTTP_ERROR(500, SUBERR_NONE)
#define HTTP_NOT_IMPLEMENTED            HTTP_ERROR(501, SUBERR_NONE)
#define HTTP_BAD_GATEWAY                HTTP_ERROR(502, SUBERR_NONE)
#define HTTP_SERVICE_UNAVAILABLE        HTTP_ERROR(503, SUBERR_NONE)
#define HTTP_GATEWAY_TIMEOUT            HTTP_ERROR(504, SUBERR_NONE)
#define HTTP_VERSION_NOT_SUPPORTED      HTTP_ERROR(505, SUBERR_NONE)
```

Remarks

An ATL Server status code is an HTTP_CODE with an HTTP status code in the low WORD and a subcode in the high WORD. Use one of these macros or define your own with AtlsHttpError.

Example

See the following samples:

- PooledAsync Sample
- NotModified Sample
- SRFSyntax Sample
- ShowUser Sample

Requirements

Header: atlserr.h

See Also

Reference

HTTP_CODE

AtlsHttpError

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

HTTP_CODE Subcodes

These macros are used as subcodes to control the actions that the ATL Server framework takes when returning from a request handler method or to distinguish one error code from another.

```
#define SUBERR_NONE                                0
#define SUBERR_S_FALSE
#define SUBERR_NO_PROCESS
#define SUBERR_ASYNC
#define SUBERR_ASYNC_DONE
#define SUBERR_ASYNC_NOFLUSH
#define SUBERR_ASYNC_NOFLUSH_DONE
#define SUBERR_NO_CACHE

#define ISE_SUBERR_BADSRF
#define ISE_SUBERR_HNDLFAIL
#define ISE_SUBERR_SYSOBJFAIL
#define ISE_SUBERR_READFILEFAIL
#define ISE_SUBERR_LOADFILEFAIL
#define ISE_SUBERR_LOADLIB
#define ISE_SUBERR_HANDLERIF
#define ISE_SUBERR_OUTOFMEM
#define ISE_SUBERR_UNEXPECTED
#define ISE_SUBERR_STENCIL_INVALIDFUNCOFFSET
#define ISE_SUBERR_STENCIL_MISMATCHWHILE
#define ISE_SUBERR_STENCIL_MISMATCHIF
#define ISE_SUBERR_STENCIL_UNEXPECTEDTYPE
#define ISE_SUBERR_STENCIL_INVALIDINDEX
#define ISE_SUBERR_STENCIL_INDEXOUTOFRANGE
#define ISE_SUBERR_STENCIL_PARSE_FAIL
#define ISE_SUBERR_STENCIL_LOAD_FAIL
#define ISE_SUBERR_HANDLER_NOT_FOUND
#define ISE_SUBERR_BAD_HANDLER_TAG
#define ISE_SUBERR_NO_HANDLER_TAG
#define ISE_SUBERR_LONGMETHODNAME
#define ISE_SUBERR_LONGHANDLERNAME
#define ISE_SUBERR_IMPERSONATIONFAILED
#define ISE_SUBERR_ISAPISTARTUPFAILED

#define DBG_SUBERR_ALREADY_DEBUGGING
#define DBG_SUBERR_NOT_DEBUGGING
#define DBG_SUBERR_INVALID_SESSION
#define DBG_SUBERR_BAD_ID
#define DBG_SUBERR_COCREATE
#define DBG_SUBERR_ATTACH
```

Remarks

Control Code	Description
SUBERR_NONE	All useful information is in the main status code. Perform default processing.
SUBERR_S_FALSE	Prevents or discontinues rendering of the current **while** or **if** block. See Replacement Tag for details.
SUBERR_NO_PROCESS	Prevents the ATL Server code from carrying out further processing. You must send headers and flush any buffered response manually if you use this subcode and want the client to receive a response.
SUBERR_ASYNC	Writes buffered content to the client asynchronously. Calls HandleRequest or a replacement method again when the buffer has been flushed.

SUBERR_ASYNC_DONE	Writes buffered content to the client asynchronously. Closes the connection with the client when the buffer has been flushed.
SUBERR_ASYNC_NOFLUSH	User code has initiated an asynchronous call. Calls HandleRequest or a replacement method again when the call is complete.
SUBERR_ASYNC_NOFLUSH_DONE	User code has initiated an asynchronous call. Closes the connection with the client when the call is complete.
SUBERR_NO_CACHE	Prevents the response from being cached.

Error Code	**Description**
ISE_SUBERR_BADSRF	The server response file could not be loaded.
ISE_SUBERR_HNDLFAIL	The server response file was loaded but could not be successfully processed.
ISE_SUBERR_SYSOBJFAIL	A Windows system object could not be created.
ISE_SUBERR_READFILEFAIL	A file read operation failed.
ISE_SUBERR_LOADLIB	LoadLibrary failed.
ISE_SUBERR_HANDLERIF	Failed to retrieve the request handler interface.
ISE_SUBERR_OUTOFMEM	The server is out of memory.
ISE_SUBERR_UNEXPECTED	The server encountered an unexpected error.
ISE_SUBERR_STENCIL_INVALIDFUNCOFFSET	The replacement method used by this tag was not found.
ISE_SUBERR_STENCIL_MISMATCHWHILE	A {{while}} replacement tag was not followed by a corresponding {{endwhile}} tag.
ISE_SUBERR_STENCIL_MISMATCHIF	An {{if}} replacement tag was not followed by a corresponding {{endif}} tag.
ISE_SUBERR_STENCIL_UNEXPECTEDTYPE	CStencil encountered a stencil token of an unknown type.
ISE_SUBERR_STENCIL_INVALIDINDEX	The last tag in the stencil should be followed by another tag.

ISE_SUBERR_STENCIL_INDEXOUTOFRANGE	Index out of range.
ISE_SUBERR_STENCIL_PARSE_FAIL	The server encountered an unexpected error while trying to parse the requested stencil.
ISE_SUBERR_STENCIL_LOAD_FAIL	The server failed to load the requested stencil. The stencil file may be damaged or missing on this web server.
ISE_SUBERR_HANDLER_NOT_FOUND	One of the handlers named in a handler tag for the requested stencil could not be found in the specified handler .dll.
ISE_SUBERR_BAD_HANDLER_TAG	This stencil contains a handler tag that could not be properly parsed by the stencil processor or does not contain a handler tag at all. Check the requested stencil for proper stencil syntax.
ISE_SUBERR_NO_HANDLER_TAG	The requested stencil does not contain a handler tag.
ISE_SUBERR_LONGMETHODNAME	A replacement tag was encountered in the requested stencil that had a replacement name that was too long. The maximum length of the replacement name must be less than or equal to the ATL_MAX_METHOD_NAME constant defined in atlstencil.h.
ISE_SUBERR_LONGHANDLERNAME	A replacement tag referring to a subhandler was encountered in the requested stencil that had a handler name that was too long. The maximum length of the handler name must be less than or equal to the ATL_MAX_METHOD_NAME constant defined in atlstencil.h
ISE_SUBERR_IMPERSONATIONFAILED	An attempt to impersonate the client making the request failed.
ISE_SUBERR_ISAPISTARTUPFAILED	The ISAPI extension used to service this request failed to load correctly due to an unknown error.

Requirements

Header: atlisapi.h

See Also

Reference

HTTP_CODE

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

HTTP_ERROR

Use this macro to create an HTTP_CODE from an HTTP status code and a subcode.

```
#define HTTP_ERROR(err, sub)
```

Parameters

err

A WORD containing zero or an HTTP status code as described in RFC 2616 (http://ietf.org/rfc/rfc2616.txt).

sub

A WORD containing a predefined or user-defined subcode.

Return Value

Returns an HTTP_CODE with *err* in the low WORD and *sub* in the high WORD.

Remarks

Prefer the use of AtlsHttpError for creating HTTP_CODEs from their constituent parts.

Requirements

Header: atlserr.h

See Also

Reference

AtlsHttpError

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

HTTP_ERROR_CODE

Use this macro to get the HTTP status code from an ATL Server status code.

```
#define HTTP_ERROR_CODE(err)
```

Parameters

err

An HTTP_CODE.

Return Value

Returns the low WORD of *err* as a DWORD.

Example

See the ShowErrors Sample.

Requirements

Header: atlserr.h

See Also

Reference

HTTP_SUBERROR_CODE

AtlsHttpError

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

HTTP_SUBERROR_CODE

Use this macro to get the subcode from an ATL Server status code.

```
#define HTTP_SUBERROR_CODE(err)
```

Parameters

err

An HTTP_CODE.

Return Value

Returns the high WORD of *err* as a DWORD.

Requirements

Header: atlserr.h

Example

See the ShowErrors Sample.

See Also

Reference

HTTP_ERROR_CODE

AtlsHttpError

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ID_DLLCACHEMGR_SRFHANDLER_NAME

This constant defines the name of the request handler that implements the HTML interface for managing the DLL cache.

```
#define ID_DLLCACHEMGR_SRFHANDLER_NAME
```

Remarks

By default, this symbol is defined to **"DllMgrSrf"**. If necessary, you can define your own string literal value for this symbol before including atlextmgmt.h.

Requirements

Header: atlextmgmt.h

See Also

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ID_DLLCACHEMGR_WEBSERVICE_NAME

This constant defines the name of the XML Web service that exposes the SOAP interface for managing the DLL cache and the request handler that implements it.

```
#define ID_DLLCACHEMGR_WEBSERVICE_NAME
```

Remarks

By default, this symbol is defined to **"DllCacheManager"**. If necessary, you can define your own string literal value for this symbol before including atlextmgmt.h.

Requirements

Header: atlextmgmt.h

See Also

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ID_DLLCACHEMGR_WEBSERVICE_URL

This constant defines the namespace of the XML Web service that exposes the SOAP interface for managing the DLL cache.

```
#define ID_DLLCACHEMGR_WEBSERVICE_URL
```

Remarks

By default, this symbol is defined to **"http://www.microsoft.com/vc/atlserver/soap/DllCacheManager"**. If necessary, you can define your own string literal value for this symbol before including atlextmgmt.h.

Requirements

Header: atlextmgmt.h

See Also

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ID_DLLCACHEMGR_WEBSERVICE_WSDL

This constant defines the name of the WSDL handler for the XML Web service that exposes the SOAP interface for managing the DLL cache.

```
#define ID_DLLCACHEMGR_WEBSERVICE_WSDL
```

Remarks

By default, this symbol is defined to **"GenDllCacheManagerWSDL"**. If necessary, you can define your own string literal value for this symbol before including atlextmgmt.h.

Requirements

Header: atlextmgmt.h

See Also

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ID_STENCILCACHEMGR_SRFHANDLER_NAME

This constant defines the name of the request handler that implements the HTML interface for managing the stencil cache.

```
#define ID_STENCILCACHEMGR_SRFHANDLER_NAME
```

Remarks

By default, this symbol is defined to **"StencilMgrSrf"**. If necessary, you can define your own string literal value for this symbol before including atlextmgmt.h.

Requirements

Header: atlextmgmt.h

See Also

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ID_STENCILCACHEMGR_WEBSERVICE_NAME

This constant defines the name of the XML Web service that exposes the SOAP interface for managing the stencil cache and the request handler that implements it.

```
#define ID_STENCILCACHEMGR_WEBSERVICE_NAME
```

Remarks

By default, this symbol is defined to **"StencilCacheManager"**. If necessary, you can define your own string literal value for this symbol before including atlextmgmt.h.

Requirements

Header: atlextmgmt.h

See Also

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ID_STENCILCACHEMGR_WEBSERVICE_URL

This constant defines the namespace of the XML Web service that exposes the SOAP interface for managing the stencil cache.

```
#define ID_STENCILCACHEMGR_WEBSERVICE_URL
```

Remarks

By default, this symbol is defined to **"http://www.microsoft.com/vc/atlserver/soap/StencilCacheManager"**. If necessary, you can define your own string literal value for this symbol before including atlextmgmt.h.

Requirements

Header: atlextmgmt.h

See Also

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ID_STENCILCACHEMGR_WEBSERVICE_WSDL

This constant defines the name of the WSDL handler for the XML Web service that exposes the SOAP interface for managing the stencil cache.

```
#define ID_STENCILCACHEMGR_WEBSERVICE_WSDL
```

Remarks

By default, this symbol is defined to **"GenStencilCacheManagerWSDL"**. If necessary, you can define your own string literal value for this symbol before including atlextmgmt.h.

Requirements

Header: atlextmgmt.h

See Also

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ID_THREADMGR_SRFHANDLER_NAME

This constant defines the name of the request handler that implements the HTML interface for managing the thread pool.

```
#define ID_THREADMGR_SRFHANDLER_NAME
```

Remarks

By default, this symbol is defined to **"ThreadMgrSrf"**. If necessary, you can define your own string literal value for this symbol before including atlextmgmt.h.

Requirements

Header: atlextmgmt.h

See Also

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ID_THREADMGR_WEBSERVICE_NAME

This constant defines the name of the XML Web service that exposes the SOAP interface for managing the thread pool and the request handler that implements it.

```
#define ID_THREADMGR_WEBSERVICE_NAME
```

Remarks

By default, this symbol is defined to **"ThreadPoolManager"**. If necessary, you can define your own string literal value for this symbol before including atlextmgmt.h.

Requirements

Header: atlextmgmt.h

See Also

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ID_THREADMGR_WEBSERVICE_URL

This constant defines the namespace of the XML Web service that exposes the SOAP interface for managing the thread pool.

```
#define ID_THREADMGR_WEBSERVICE_URL
```

Remarks

By default, this symbol is defined to **"http://www.microsoft.com/vc/atlserver/soap/ThreadPoolManager"**. If necessary, you can define your own string literal value for this symbol before including atlextmgmt.h.

Requirements

Header: atlextmgmt.h

See Also

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ID_THREADMGR_WEBSERVICE_WSDL

This constant defines the name of the WSDL handler for the XML Web service that exposes the SOAP interface for managing the thread pool.

```
#define ID_THREADMGR_WEBSERVICE_WSDL
```

Remarks

By default, this symbol is defined to **"GenThreadPoolManagerWSDL"**. If necessary, you can define your own string literal value for this symbol before including atlextmgmt.h.

Requirements

Header: atlextmgmt.h

See Also

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

IDR_DLLMGR_SRF

This constant defines the name of the stencil resource that provides the HTML interface for managing the DLL cache.

```
#define IDR_DLLMGR_SRF
```

Remarks

By default, this symbol is defined to **"DLLMGR.SRF"**, a resource provided by atlsrv.rc. If necessary, you can define your own string literal value for this symbol before including atlextmgmt.h.

Requirements

Header: atlextmgmt.h

See Also

Reference

_ATL_DLLCACHE_MANAGEMENT

_ATL_DLLCACHE_NOUI

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

IDR_STENCILMGR_SRF

This constant defines the name of the stencil resource that provides the HTML interface for managing the stencil cache.

```
#define IDR_STENCILMGR_SRF
```

Remarks

By default, this symbol is defined to **"STENCILMGR.SRF"**, a resource provided by atlsrv.rc. If necessary, you can define your own string literal value for this symbol before including atlextmgmt.h.

Requirements

Header: atlextmgmt.h

See Also

Reference

_ATL_STENCILCACHE_MANAGEMENT

_ATL_STENCILCACHE_NOUI

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

IDR_THREADMGR_SRF

This constant defines the name of the stencil resource that provides the HTML interface for managing the thread pool.

```
#define IDR_THREADMGR_SRF
```

Remarks

By default, this symbol is defined to **"THREADMGR.SRF"**, a resource provided by atlsrv.rc. If necessary, you can define your own string literal value for this symbol before including atlextmgmt.h.

Requirements

Header: atlextmgmt.h

See Also

Reference

_ATL_THREADPOOL_MANAGEMENT

_ATL_THREADPOOL_NOUI

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

MAX_CONNECTION_STRING_LEN

This macro defines the maximum length of the data source connection string used for session-state data storage.

```
#define MAX_CONNECTION_STRING_LEN
```

Remarks

If necessary, you can define your own value for this symbol before including atlsession.h. This value has no effect on memory-based session-state data. By default, this symbol resolves to 2048.

Requirements

Header: atlsession.h

See Also

Other Resources

Session-State Services (ATL Server)
ATL Server Concepts
ATL Server Reference
ATL Server Macros

MAX_SESSION_KEY_LEN

This macro defines the maximum size allowed for a session name, in bytes.

```
#define MAX_SESSION_KEY_LEN
```

Remarks

This limit is imposed to ensure efficiency and agreement between multiple classes. If necessary, you can define your own value for this symbol before including atlsession.h. By default, this symbol resolves to 128.

Requirements

Header: atlsession.h

See Also

Other Resources

Session-State Services (ATL Server)
ATL Server Concepts
ATL Server Reference
ATL Server Macros

MAX_TOKEN_LENGTH

This macro defines the maximum allowed size in bytes for the name of an HTML form element that can be handled by ATL Server.

```
#define MAX_TOKEN_LENGTH
```

Remarks

This limit is imposed to ensure efficiency and agreement between multiple classes. It is defined equal to MAX_PATH.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

MAX_VARIABLE_NAME_LENGTH

This macro is defines the maximum size of session-state data variable names.

```
#define MAX_VARIABLE_NAME_LENGTH
```

Remarks

If necessary, you can define your own value for this symbol before including atlsession.h. This value affects database-backed session-state data only; it has no effect on the memory-based session-state implementation. By default, this symbol resolves to 50.

Requirements

Header: atlsession.h

See Also

Other Resources

Session-State Services (ATL Server)
ATL Server Concepts
ATL Server Reference
ATL Server Macros

MAX_VARIABLE_VALUE_LENGTH

This macro is defines the maximum size of session-state data values.

```
#define MAX_VARIABLE_VALUE_LENGTH
```

Remarks

If necessary, you can define your own value for this symbol before including atlsession.h. This value affects database-backed session-state data only; it has no effect on the memory-based session-state implementation. By default, this symbol resolves to 1024.

Requirements

Header: atlsession.h

See Also

Other Resources

Session-State Services (ATL Server)
ATL Server Concepts
ATL Server Reference
ATL Server Macros

NO_ATL_MGMT_STENCIL_WARNING

Define this macro before including atlextmgmt.h to prevent a message being displayed during compilation to remind you to include stencil resources for the HTML interface used by the extension management services.

```
#ifndef NO_ATL_MGMT_STENCIL_WARNING
```

Requirements

Header: atlextmgmt.h

See Also

Concepts

Extension Management Services

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

PERFREG_ENTRY

Use this macro to register CPerfMon-derived classes with the Windows performance monitoring subsystem.

```
#define PERFREG_ENTRY(
      className
)
```

Parameters

className

The name of the performance monitoring class to be registered. For attributed code, this class is declared using the perfmon attribute; otherwise, it will be explicitly derived from the CPerfMon class.

Remarks

The PERFREG_ENTRY macro is inserted automatically for performance objects added using the Performance Monitor Wizard. This macro is used implicitly by the performance monitor attributes, so it does not need to appear in attributed code.

Requirements

Header: atlperf.h

See Also

Concepts

Performance Monitoring

Other Resources

ATL Server Reference

Performance Monitoring Reference

REPLACEMENT_METHOD_ENTRY

Use this macro in a replacement method map to associate a replacement method with a particular tag name.

```
#define REPLACEMENT_METHOD_ENTRY(
      methodName,
      methodFunc
)
```

Parameters

methodName

A string containing the name of the replacement tag that will map to the method specified by *methodFunc*.

methodFunc

The C++ method that will be called when tags matching *methodName* are encountered.

Remarks

The replacement method specified by *methodFunc* should have the following signature:

```
HTTP_CODE methodFunc( );
```

This function should be a member of the class containing the replacement method map.

This macro provides similar functionality to the tag_name attribute.

Requirements

Header: atlstencil.h

Example

See the MantaWeb Sample and the ConfirmUser Sample.

See Also

Reference

BEGIN_REPLACEMENT_METHOD_MAP
END_REPLACEMENT_METHOD_MAP
REPLACEMENT_METHOD_ENTRY_EX
REPLACEMENT_METHOD_ENTRY_EX_STR

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

REPLACEMENT_METHOD_ENTRY_EX

Use this macro in a replacement method map to associate a replacement method with a particular tag name. Use when the replacement method needs to receive method arguments.

```
#define REPLACEMENT_METHOD_ENTRY_EX(
      methodName,
      methodFunc,
      paramType,
      parseFunc
)
```

Parameters

methodName

A string containing the name of the replacement tag that will map to the method specified by *methodFunc*.

methodFunc

The C++ method that will be called when tags matching *methodName* are encountered.

paramType

The C++ parameter type handled by *methodFunc* and *parseFunc*.

parseFunc

The C++ method that will be called to parse any arguments passed to the replacement method by means of a replacement tag in a server response file.

Remarks

The replacement method specified by *methodFunc* should have the following signature:

```
HTTP_CODE methodFunc(paramType* pArg);
```

The argument parsing method specified by *parseFunc* should have the following signature:

```
HTTP_CODE parseFunc(IAtlMemMgr* pMemoryManager, LPCSTR szArg, paramType** ppArg);
```

Both functions should be members of the class containing the replacement method map.

parseFunc is responsible for:

- Allocating memory for the argument to be passed to the replacement method using pMemoryManager->Allocate.
- Setting the value of the argument to be passed to the replacement method using the data obtained from *szArg*.
- Passing a pointer to the argument to be passed to the replacement method back through *ppArg*.

parseFunc is called only during stencil parsing. The memory allocated using *pMemoryManager* will be automatically freed by the framework when the stencil is unloaded.

This macro provides similar functionality to the tag_name attribute.

Example

See Nonattributed Request Handler Code.

See the MantaWeb Sample and the ShowImage Sample.

Requirements

Header: atlstencil.h

See Also

Reference

BEGIN_REPLACEMENT_METHOD_MAP
END_REPLACEMENT_METHOD_MAP
REPLACEMENT_METHOD_ENTRY
REPLACEMENT_METHOD_ENTRY_EX_STR

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

REPLACEMENT_METHOD_ENTRY_EX_STR

Use this macro in a replacement method map to associate a replacement method with a particular tag name. Use when the replacement method needs to receive method arguments in the form of a **char*** parameter.

```
#define REPLACEMENT_METHOD_ENTRY_EX_STR(
      methodName,
      methodFunc
)
```

Parameters

methodName

A string containing the name of the replacement tag that will map to the method specified by *methodFunc*.

methodFunc

The C++ method that will be called when tags matching *methodName* are encountered.

Remarks

This macro is equivalent to:

```
REPLACEMENT_METHOD_ENTRY_EX(methodName, methodFunc, char, DefaultParseString)
```

Requirements

Header: atlstencil.h

See Also

Reference

BEGIN_REPLACEMENT_METHOD_MAP
END_REPLACEMENT_METHOD_MAP
REPLACEMENT_METHOD_ENTRY
REPLACEMENT_METHOD_ENTRY_EX
ITagReplacerImpl::DefaultParseString

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Macros

Request Initialization Flags

These macros are used as flags to be returned from a request handler's GetFlags method.

```
#define  ATLSRV_INIT_USECACHE       1
#define  ATLSRV_INIT_USEASYNC       2
#define  ATLSRV_INIT_USEASYNC_EX    4
```

Remarks

Flag	Description
ATLSRV_INIT_USECACHE	Specifies that the response generated by the request handler will be cached unless the request handler later decides that it should not be cached by returning HTTP_SUCCESS_NO_CACHE or calling IPageCacheControl::Cache. See CCacheServerContext for more details.
ATLSRV_INIT_USEASYNC	Specifies that the request handler is intending to write to the client asynchronously and all the data is to be written at once. See Writing Asynchronous Web Applications for details.
ATLSRV_INIT_USEASYNC_EX	Specifies that the client is intending to write to the client asynchronously in multiple stages. See Writing Asynchronous Web Applications for details.

Requirements

Header: atlisapi.h

See Also

Reference

DECLARE_ASYNC_HANDLER

DECLARE_ASYNC_HANDLER_EX

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

SESSION_COOKIE_NAME

This macro is used as the name of the cookie used for tracking session-state data.

```
#define SESSION_COOKIE_NAME
```

Remarks

If necessary, you can define your own value for this symbol before including atlisapi.h or atlsession.h.

Requirements

Header: atlisapi.h or atlsession.h

See Also

Other Resources

Session-State Services (ATL Server)
ATL Server Concepts
ATL Server Reference
ATL Server Macros

Stencil Tokens

These constants identify different tokens recognized by the CStencil and CHtmlStencil classes.

```
#define STENCIL_TEXTTAG
#define STENCIL_REPLACEMENT
#define STENCIL_ITERATORSTART
#define STENCIL_ITERATOREND
#define STENCIL_CONDITIONALSTART
#define STENCIL_CONDITIONALELSE
#define STENCIL_CONDITIONALEND
#define STENCIL_STENCILINCLUDE
#define STENCIL_STATICINCLUDE
#define STENCIL_LOCALE
#define STENCIL_CODEPAGE
```

Remarks

Constant	Token
STENCIL_TEXTTAG	Static content.
STENCIL_REPLACEMENT	Replacement tag.
STENCIL_ITERATORSTART	{{while}} replacement tag.
STENCIL_ITERATOREND	{{endwhile}} replacement tag.
STENCIL_CONDITIONALSTART	{{if}} replacement tag.
STENCIL_CONDITIONALELSE	{{else}} replacement tag.
STENCIL_CONDITIONALEND	{{endif}} replacement tag.
STENCIL_STENCILINCLUDE	Include tag.
STENCIL_STATICINCLUDE	Include tag.
STENCIL_LOCALE	Locale tag.
STENCIL_CODEPAGE	Codepage tag.

Requirements

Header: atlstencil.h

See Also

Reference

CStencil Class

CHtmlStencil Class

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ATL Server Response File Reference

STENCIL_USER_TOKEN_BASE

Use this value as the starting point for constants used to identify custom tokens in CStencil-derived classes.

```
const DWORD STENCIL_USER_TOKEN_BASE;
```

Requirements

Header: atlstencil.h

See Also

Reference

CStencil Class

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

Validation Results

These macros are returned as results from the **Exchange** or **Validate** methods of CValidateObject.

```
#define VALIDATION_S_OK
#define VALIDATION_S_EMPTY
#define VALIDATION_E_PARAMNOTFOUND
#define VALIDATION_E_LENGTHMIN
#define VALIDATION_E_LENGTHMAX
#define VALIDATION_E_INVALIDLENGTH
#define VALIDATION_E_INVALIDPARAM
#define VALIDATION_E_FAIL
```

Remarks

Macro	Description
VALIDATION_S_OK	The named value was found and could be converted successfully.
VALIDATION_S_EMPTY	The name was present, but the value was empty.
VALIDATION_E_PARAMNOTFOUND	The named value was not found.
VALIDATION_E_LENGTHMIN	The name was present and could be converted to the requested data type, but the value was too small.
VALIDATION_E_LENGTHMAX	The name was present and could be converted to the requested data type, but the value was too large.
VALIDATION_E_INVALIDLENGTH	(Not used).
VALIDATION_E_INVALIDPARAM	The name was present, but the value could not be converted to the requested data type.
VALIDATION_E_FAIL	An unspecified error occurred.

Requirements

Header: atlisapi.h

See Also

Reference

CValidateObject Class

VALIDATION_SUCCEEDED

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

VALIDATION_SUCCEEDED

Use this macro to determine whether validation succeeded or failed.

```
#define VALIDATION_SUCCEEDED(x)
```

Parameters

x

A DWORD validation result.

Return Value

If *x* represents a successful validation result (**VALIDATION_S_OK** or **VALIDATION_S_EMPTY**), this macro returns **TRUE**; otherwise, it returns **FALSE**.

Requirements

Header: atlserr.h

See Also

Reference

Validation Results

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Macros

ATL Server Typedefs

The following typedefs are included in the ATL Server Library.

Name	Description
ATL_URL_PORT	The type used by CUrl for specifying a port number.
CAtlHttpClient	A synonym for a class that provides HTTP client support based on the CAtlHttpClientT class.
CAtlRECharTraits	The default character traits type to use with CAtlRegExp.
CCryptSHA1Hash	Synonym for CCryptSHAHash.
CDBSessionServiceImpl	Default specialization of CDBSessionServiceImplT.
CFixedStringKey	The default cache key type for ATL Server cache implementations.
CPath	A specialization of CPathT using **CString**.
CPathA	A specialization of CPathT using **CStringA**.
CPathW	A specialization of CPathT using **CStringW**.
DefaultThreadTraits	The default thread traits class.
GETATLHANDLERBYNAME	A pointer to the function exported by ATL Server Web application DLLs for returning named request handlers.
HCACHEITEM	A handle to an item stored in a CMemoryCacheBase-derived class.
HSESSIONENUM	A handle to an item used to iterate through session variables exposed by ISession.
HTTP_CODE	Used throughout ATL Server to describe the success of a method call.
INITIALIZEATLHANDLERS	A pointer to the function exported by ATL Server Web application DLLs for initialization before any request handlers are created.
LPCURL	A pointer to a constant CUrl object.
LPURL	A pointer to a CUrl object.
PFnAsyncComplete	A function pointer suitable for handling callbacks on completion of asynchronous IO.
PFnHandleRequest	A member function pointer compatible with IRequestHandler::HandleRequest.
PFNATLCHUNKEDCB	A pointer to a function suitable for use as a callback used to prepare chunked-encoded HTTP requests.
PFNATLSTATUSCALLBACK	A pointer to a function suitable for use as a callback used to monitor either HTTP request or response transfer status.
PFN_GETPROVIDERINFO	A function pointer suitable for returning a connection string to a database used for storing session data.

UNINITIALIZEATLHANDLERS	A pointer to the function exported by ATL Server web application DLLs for cleaning up when all request handlers have been released.

See Also

Other Resources

ATL Server Concepts
ATL Server Reference
ATL Server Samples

ATL_URL_PORT

The type used by CUrl for specifying a port number.

```
typedef WORD ATL_URL_PORT;
```

Requirements

Header: atlutil.h

See Also

Reference

CUrl::SetPortNumber

CUrl::GetPortNumber

Other Resources

ATL Server Reference

ATL Server Typedefs

CAtlHttpClient

A synonym for a class that provides HTTP client support based on the CAtlHttpClientT class.

```
typedef CAtlHttpClientT<ZEvtSyncSocket> CAtlHttpClient;
```

Remarks

CAtlHttpClientT, which provides HTTP client services, requires as a template argument the name of a class that provides basic network socket functionality. The **CAtlHttpClient** typedef specializes the CAtlHttpClientT class template with a class that provides default network socket functionality.

Requirements

Header: atlhttp.h

See Also

Concepts

HTTP Client Services

Other Resources

ATL Server Reference

ATL Server Typedefs

HTTP Client Reference

CAtlRECharTraits

The default character traits type to use with CAtlRegExp.

```
#ifndef _UNICODE
   typedef CAtlRECharTraitsA CAtlRECharTraits;
#else
   typedef CAtlRECharTraitsW CAtlRECharTraits;
#endif
```

Remarks

This type resolves to the appropriate regular expression character traits class based on the definition of **_UNICODE**.

Requirements

Header: atlrx.h

See Also

Reference

CAtlRegExp Class

Other Resources

ATL Server Reference

ATL Server Typedefs

CDBSessionServiceImpl

Default specialization of CDBSessionServiceImplT.

```
typedef CDBSessionServiceImplT< > CDBSessionServiceImpl;
```

Requirements

Header: atlsession.h

See Also

Reference

CDBSessionServiceImplT Class

Other Resources

ATL Server Reference

ATL Server Typedefs

CFixedStringKey

The default cache key type for ATL Server cache implementations.

```
typedef CFixedStringT<CStringA, ATL_CACHE_KEY_LENGTH> CFixedStringKey;
```

Remarks

CMemoryCacheBase, CMemoryCache, CFileCache, CBlobCache, and CStencilCache all use the **CFixedStringKey** type by default for cache keys.

Requirements

Header: atlcache.h

See Also

Reference

ATL_CACHE_KEY_LENGTH

Other Resources

Caching

Caching Reference

ATL Server Reference

ATL Server Typedefs

CPath

A specialization of CPathT using **CString**.

```
typedef CPathT< CString > CPath;
```

Requirements

Header: atlpath.h

Example

See the ISAPIFilter Sample.

See Also

Reference

CPathT Class

Other Resources

ATL Server Reference

ATL Server Typedefs

CPathA

A specialization of CPathT using **CStringA**.

```
typedef CPathT< CStringA > CPathA;
```

Requirements

Header: atlpath.h

Example

See the ISAPIFilter Sample.

See Also

Reference

CPathT Class

Other Resources

ATL Server Reference

ATL Server Typedefs

CPathW

A specialization of CPathT using **CStringW**.

```
typedef ATL::CPathT< CStringW > CPathW;
```

Requirements

Header: atlpath.h

See Also

Reference

CPathT Class

Other Resources

ATL Server Reference

ATL Server Typedefs

CCryptSHA1Hash

Synonym for CCryptSHAHash.

```
typedef CCryptSHAHash CCryptSHA1Hash;
```

Requirements

Header: atlcrypt.h

See Also

Reference

CCryptSHAHash Class

Other Resources

ATL Server Reference

ATL Server Typedefs

DefaultThreadTraits

The default thread traits class.

```
#if !defined(_ATL_MIN_CRT) && defined(_MT)
   typedef CRTThreadTraits DefaultThreadTraits;
#else
   typedef Win32ThreadTraits DefaultThreadTraits;
#endif
```

Remarks

If the current project uses the multithreaded CRT, **DefaultThreadTraits** is defined as CRTThreadTraits. Otherwise, Win32ThreadTraits is used.

Requirements

Header: atlbase.h

See Also

Other Resources

ATL Server Reference

ATL Server Typedefs

GETATLHANDLERBYNAME

A pointer to the function exported by ATL Server Web application DLLs for returning named request handlers.

```
typedef BOOL (__stdcall *GETATLHANDLERBYNAME)(
   LPCSTR szHandlerName,
   IIsapiExtension* pExtension,
   IUnknown** ppHandler
);
```

Parameters

szHandlerName

The name of the request handler to be returned in *ppHandler*.

pExtension

The IIsapiExtension interface of the object loading the request handler.

ppHandler

[out] Address of the pointer variable that, on success, receives the IUnknown interface pointer of the request handler specified by *szHandlerName*.

Return Value

Returns **TRUE** on success; **FALSE** on failure.

Remarks

Web application DLLs must export a function with this signature using the name specified by **ATLS_FUNCID_GETATLHANDLERBYNAME**, defined in atlisapi.h.

Requirements

Header: atlisapi.h

See Also

Reference

INITIALIZEATLHANDLERS

UNINITIALIZEATLHANDLERS

Concepts

Web Application DLL

Other Resources

ATL Server Reference

ATL Server Typedefs

HCACHEITEM

A handle to an item stored in a CMemoryCacheBase-derived class.

```
typedef pointer_sized_type HCACHEITEM;
```

Remarks

An object of this type is only understood by the cache from which it was obtained.

Requirements

Header: atlcache.h

See Also

Reference

CMemoryCacheBase Class

Other Resources

ATL Server Reference

ATL Server Typedefs

HSESSIONENUM

A handle to an item used to iterate through session variables exposed by ISession.

```
typedef DWORD_PTR HSESSIONENUM;
```

Remarks

An object of this type is only understood by the session object from which it was obtained.

Requirements

Header: atlsiface.h

See Also

Reference

ISession Interface

Other Resources

ATL Server Reference

ATL Server Typedefs

HTTP_CODE

Used throughout ATL Server to describe the success of a method call.

```
typedef DWORD HTTP_CODE;
```

Remarks

An HTTP_CODE has zero or an HTTP status code in the low WORD and a subcode in the high WORD. HTTP status codes are described in Section 10 of RFC 2068 (http://ietf.org/rfc/rfc2068.txt). Subcodes are used to provide extended error information or provide information to control the behavior of the ATL Server classes.

Use one of the predefined HTTP_CODE status codes or define your own using AtlsHttpError.

Use HTTP_ERROR_CODE and HTTP_SUBERROR_CODE to split the HTTP_CODE into its components.

Requirements

Header: atlserr.h

Example

See the MantaWeb Sample.

See Also

Reference

AtlsHttpError
HTTP_CODE Status Codes
HTTP_CODE Subcodes
HTTP_ERROR_CODE
HTTP_SUBERROR_CODE

Other Resources

ATL Server Reference
ATL Server Typedefs

INITIALIZEATLHANDLERS

A pointer to the function exported by ATL Server Web application DLLs for initialization before any request handlers are created.

```
typedef BOOL (__stdcall * INITIALIZEATLHANDLERS)(
   IHttpServerContext* ,
   IIsapiExtension*
);
```

Remarks

Web application DLLs may export a function with this signature using the name specified by **ATLS_FUNCID_INITIALIZEHANDLERS**, defined in atlisapi.h.

Requirements

Header: atlisapi.h

See Also

Reference

UNINITIALIZEATLHANDLERS

GETATLHANDLERBYNAME

Concepts

Web Application DLL

Other Resources

ATL Server Reference

ATL Server Typedefs

LPCURL

A pointer to a constant CUrl object.

```
typedef const CUrl * LPCURL;
```

Requirements

Header: atlutil.h

See Also

Reference

LPURL

Other Resources

ATL Server Reference

ATL Server Typedefs

LPURL

A pointer to a CUrl object.

```
typedef CUrl* LPURL;
```

Requirements

Header: atlutil.h

See Also

Reference

LPCURL

Other Resources

ATL Server Reference

ATL Server Typedefs

PFnAsyncComplete

A function pointer suitable for handling callbacks on completion of asynchronous IO.

```
typedef void (* PFnAsyncComplete)(
   AtlServerRequest * pRequestInfo,
   DWORD cbIO,
   DWORD dwError
);
```

Remarks

The **PFAsyncComplete** typedef is a function pointer suitable for handling callbacks on completion of asynchronous IO.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Reference

ATL Server Typedefs

PFNATLCHUNKEDCB

A pointer to a function suitable for use as a callback used to prepare chunked-encoded HTTP requests.

```
typedef bool (WINAPI *PFNATLCHUNKEDCB)(
   BYTE** ppData,
   DWORD *pdwSize
);
```

Remarks

The CAtlHttpClientT class, when its NavigateChunked function is called, requires a callback function to prepare the chunked-encoded response. This typedef defines the signature of the callback function necessary. Before the CAtlHttpClientT::NavigateChunked function is called, a pointer to a function matching the **PFNATLCHUNKEDCB** typedef must be provided to the CAtlHttpClientT class through the CAtlNavigateData class.

Requirements

Header: atlisapi.h

See Also

Concepts

HTTP Client Services

Other Resources

ATL Server Reference

ATL Server Typedefs

HTTP Client Reference

PFNATLSTATUSCALLBACK

A pointer to a function suitable for use as a callback used to monitor either HTTP request or response transfer status.

```
typedef bool(WINAPI * PFNATLSTATUSCALLBACK)(
   DWORD dwBytesSent,
   DWORD dwCookie
);
```

Remarks

The CAtlNavigateData class, which is used to provide settings to the CAtlHttpClientT class, can be used to install callback functions that are periodically invoked during network send and receive operations. This typedef defines the function prototype expected by the **CAtlNavigateData** class. See CAtlNavigateData::SetSendStatusCallback and CAtlNavigateData::SetReadStatusCallback for more information.

Requirements

Header: atlisapi.h

See Also

Concepts

HTTP Client Services

Other Resources

ATL Server Reference

ATL Server Typedefs

HTTP Client Reference

PFN_GETPROVIDERINFO

A function pointer suitable for returning a connection string to a database used for storing session data.

```
typedef bool(* PFN_GETPROVIDERINFO)(
   DWORD_PTR,
   wchar_t **
);
```

Remarks

The **PFN_GETPROVIDERINFO** typedef is a function pointer compatible with CDBSession::Initialize.

Requirements

Header: atlsession.h

See Also

Other Resources

ATL Server Reference

ATL Server Typedefs

PFnHandleRequest

A member function pointer compatible with IRequestHandler::HandleRequest.

```
typedef HTTP_CODE (IRequestHandler::* PFnHandleRequest)(
   AtlServerRequest * pRequestInfo,
   IServiceProvider * pProvider
);
```

Remarks

The **PFnHandleRequest** typedef is a member function pointer compatible with IRequestHandler::HandleRequest.

Requirements

Header: atlisapi.h

See Also

Other Resources

ATL Server Reference

ATL Server Typedefs

UNINITIALIZEATLHANDLERS

A pointer to the function exported by ATL Server Web application DLLs for cleaning up when all request handlers have been released.

```
typedef void (__stdcall * UNINITIALIZEATLHANDLERS)( );
```

Remarks

Web application DLLs may export a function with this signature using the name specified by **ATLS_FUNCID_UNINITIALIZEHANDLERS**, defined in atlisapi.h.

Requirements

Header: atlisapi.h

See Also

Reference

INITIALIZEATLHANDLERS

GETATLHANDLERBYNAME

Concepts

Web Application DLL

Other Resources

ATL Server Reference

ATL Server Typedefs

ATL Server Archetypes

In this context, an *archetype* is a theoretical class that supplies a collection of methods, data members, static functions, typedefs, or other features. The archetype also includes a description of the semantics necessary to create or use the class to represent a particular concept. Classes that mimic the archetype by providing the same features embody the same concept and can be used wherever the archetype can be used.

Archetypes are useful in C++ for describing the features of valid values for template parameters. The designer of the template has a clear idea of the necessary and sufficient features of the template parameter, and the compiler will enforce the syntactic requirements at build time, but the user of a template needs documentation to describe the semantics and to allow the relationships between archetypes and classes to be clearly spelled out.

Examples of archetypes in the Standard C++ Library are the different types of iterator and container. These archetypes are described in the topics Iterator Conventions and STL Containers.

ATL Server defines the following archetypes:

Name	Description
Cache Culler Archetype	Classes that conform to the *cache culler* archetype can be used to determine whether, and in what order, items should be removed from a CMemoryCacheBase-derived cache using time-based criteria.
Cache Flusher Archetype	Classes that conform to the *cache flusher* archetype can be used to determine whether, and in what order, items should be removed from a CMemoryCacheBase-derived cache using the number and sequence of accesses as the basis for making the decision.
Cache Statistics Archetype	Classes that conform to the *cache statistics* archetype can be used to collect and expose statistics related to the use of a cache including such information as effectiveness of the cache and memory used.
DLL Cache Peer Archetype	Classes that conform to the *DLL cache peer* archetype can be used to describe extra data to be associated with each cached DLL held by CDllCache and will be notified when DLLs are added to or removed from the cache.
File Cache Peer Archetype	Classes that conform to the *file cache peer* archetype can be used to describe extra data to be associated with each cached file held by CFileCache and will be notified when files are added to or removed from the cache.
Request Statistics Archetype	Classes that conform to the *request statistics* archetype can be used to collect and expose statistics related to the use of an ISAPI extension including such information as number of requests handled and average time per request.
SOAP Client Archetype	Classes that conform to the *SOAP client* archetype can be used to send SOAP messages to a server when used with SPROXY-generated proxy classes.
Worker Archetype	Classes that conform to the *worker* archetype provide the code to process work items queued on a thread pool.
XML Web Service Client Archetype	Classes that conform to the *XML Web Service client* archetype conform to the SOAP client archetype and provide some additional Web-related features.

See Also

Tasks

PooledAsync Sample: Increases Performance with an Additional Thread Pool

Other Resources

ATL Server Concepts

ATL Server Reference

Cache Culler Archetype

Classes that conform to the *cache culler* archetype can be used to determine whether, and in what order, items should be removed from a CMemoryCacheBase-derived cache using time-based criteria.

Implementation

To implement a class conforming to this archetype, the class must provide the following methods:

Method	Description
Access	Called by the cache when the reference count of an item is incremented.
Add	Called by the cache when a new item is being added to the cache.
Commit	Called by the cache when a new item has been added to the cache.
GetExpired	Called by the cache when scanning for items that have expired and can be removed from the cache.
IsExpired	Called by the cache to determine if an item has expired and can be removed from the cache.
Release	Called by the cache when the reference count of an item is decremented.
Remove	Called by the cache when an item has been removed from the cache.
Start	Called by the cache when it intends to call GetExpired multiple times to remove items from the cache.

Existing Implementations

These classes conform to this archetype:

Class	Description
CExpireCuller	Allows the automatic removal of cache items that exceed the absolute expiration time set by users of the cache.
CFixedLifetimeCuller	Allows the automatic removal of cache items that have not been accessed in a certain period. The lifetime of items in the cache cannot be changed at run time and is the same for all items.
CLifetimeCuller	Allows the automatic removal of cache items that have not been accessed in a certain period. The lifetime of an item can be changed at run time and can be different for each of the items in the cache.
CNoExpireCuller	Disables the automatic removal of cache items based on expiration times.
COrCullers	This class combines the functionality of two cache cullers into a single class.

Use

These template parameters expect the class to conform to this archetype:

Parameter name	Used by
Culler	CMemoryCacheBase
CullClass	CMemoryCache
CullClass	CBlobCache

CullClass	CStencilCache
CullClass	CFileCache
CFirst	COrCullers
CSecond	COrCullers

See Also

Reference

ATL Server Archetypes

Other Resources

ATL Server Reference

Caching Reference

CacheCullerArchetype::Access

Called by the cache when the reference count of an item is incremented.

```
void Access(
   CCullerCacheData * pItem
);
```

Parameters

pItem

The CCullerCacheData structure corresponding to the item that has been referenced.

Remarks

This method will be called when the item is retrieved from the cache or otherwise has its reference count incremented such as by a call to CMemoryCacheBase::AddRefEntry.

See Also

Reference

Cache Culler Archetype

CacheCullerArchetype::Release

Other Resources

Caching Reference

CCullerCacheData Members

CacheCullerArchetype::Add

Called by the cache when a new item is being added to the cache.

```
void Add(
   CCullerCacheData * pItem
);
```

Parameters

pItem

The CCullerCacheData structure corresponding to the item that is being added.

Remarks

The *pItem* parameter provides the expiration time and lifetime duration settings provided by the cache client.

See Also

Reference

Cache Culler Archetype

Other Resources

Caching Reference

Caching

CCullerCacheData Members

CacheCullerArchetype::Commit

Called by the cache when a new item has been added to the cache.

```
void Commit(
   CCullerCacheData * pItem
);
```

Parameters

pItem

The CCullerCacheData structure corresponding to the item that has been added to the cache.

Remarks

This method is called when a new item has been added to the cache and is called after the Add method. The *pItem* parameter provides the expiration time and lifespan settings provided by the cache client.

See Also

Reference

Cache Culler Archetype

CacheCullerArchetype::Remove

Other Resources

Caching Reference

CCullerCacheData Members

CacheCullerArchetype::GetExpired

Called by the cache when scanning for items that have expired and can be removed from the cache.

```
CCullerCacheData* GetExpired( );
```

Return Value

Returns a pointer to the CCullerCacheData structure for the first candidate for removal.

Remarks

Note that the cache has the last word on whether the item returned by this method will actually be removed from the cache. The culler must not assume that an item returned from this method has been removed. A call to Remove will notify the culler when an item has been removed.

See Also

Reference

Cache Culler Archetype

Other Resources

Caching Reference

CCullerCacheData Members

CacheCullerArchetype::IsExpired

Called by the cache to determine if an item has expired and can be removed from the cache.

```
BOOL IsExpired(
   CCullerCacheData * pItem
);
```

Parameters

pItem

The CCullerCacheData structure corresponding to the item that the cache is enquiring about.

Return Value

Returns **TRUE** if the item has expired, **FALSE** otherwise.

See Also

Reference

Cache Culler Archetype

Other Resources

Caching Reference

CCullerCacheData Members

CacheCullerArchetype::Release

Called by the cache when the reference count of an item is decremented.

```
void Release(
   CCullerCacheData * pItem
);
```

Parameters

pItem

The CCullerCacheData structure corresponding to the item that has been released.

See Also

Reference

Cache Culler Archetype

CacheCullerArchetype::Access

Other Resources

Caching Reference

CCullerCacheData Members

CacheCullerArchetype::Remove

Called by the cache when an item has been removed from the cache.

```
void Remove(
   CCullerCacheData * pItem
);
```

Parameters

pItem

The CCullerCacheData structure corresponding to the item that has been removed.

See Also

Reference

Cache Culler Archetype

CacheCullerArchetype::Commit

Other Resources

Caching Reference

CCullerCacheData Members

CacheCullerArchetype::Start

Called by the cache when it intends to call GetExpired or IsExpired to remove items from the cache.

```
void Start( );
```

Remarks

Provides the culler with a notification that the cache intends to remove a number of items and allows the culler to organize its data structures for that possibility. One possible use is to set a base time for comparisons with the cache items.

See Also

Reference

Cache Culler Archetype

Other Resources

Caching Reference

CCullerCacheData Members

Cache Flusher Archetype

Classes that conform to the *cache flusher* archetype can be used to determine whether, and in what order, items should be removed from a CMemoryCacheBase-derived cache using the number and sequence of accesses as the basis for making the decision.

Implementation

To implement a class conforming to this archetype, the class must provide the following methods:

Method	Description
Access	Called by the cache when the reference count of an item is incremented.
Add	Called by the cache when a new item has been added to the cache.
GetNext	Called by the cache when scanning for items that can be removed from the cache.
GetStart	Called by the cache to get the first item that can be removed from the cache.
Release	Called by the cache when the reference count of an item is decremented.
Remove	Called by the cache when an item has been removed from the cache.

Existing Implementations

These classes conform to this archetype:

Class	Description
CLOUFlusher	Items that are used infrequently are presented for removal before items that are used more often. (Least Often Used)
CLRUFlusher	Items that have not been used for a long time are presented for removal before items that have been used more recently. (Least Recently Used)
CNoFlusher	Used to disable the automatic removal of cache items based on access patterns.
COldFlusher	Items are presented for removal in the order in which they were added to the cache. (First In, First Out)
COrFlushers	Combines the functionality of two cache flushers into a single class.

Use

These template parameters expect the class to conform to this archetype:

Parameter name	Used by
Flusher	CMemoryCacheBase
FlushClass	CMemoryCache
FlushClass	CBlobCache
FlushClass	CStencilCache
FlushClass	CFileCache

CFirst	COrFlushers
CSecond	COrFlushers

See Also

Reference

ATL Server Archetypes

Other Resources

ATL Server Reference

Caching Reference

Caching

CacheFlusherArchetype::Access

Called by the cache when the reference count of an item is incremented.

```
void Access(
   CFlusherCacheData * pItem
);
```

Parameters

pItem

The CFlusherCacheData structure corresponding to the item that has been referenced.

Remarks

This method will be called when the item is retrieved from the cache or otherwise has its reference count incremented such as by a call to CMemoryCacheBase::AddRefEntry.

See Also

Reference

Cache Flusher Archetype

Other Resources

Caching Reference

CacheFlusherArchetype::Add

Called by the cache when a new item has been added to the cache.

```
void Add(
   CFlusherCacheData * pItem
);
```

Parameters

pItem

The CFlusherCacheData structure corresponding to the item that has been added to the cache.

See Also

Reference

Cache Flusher Archetype

Other Resources

Caching Reference

CacheFlusherArchetype::GetNext

Called by the cache when scanning for items that can be removed from the cache.

```
CFlusherCacheData * GetNext(
   CFlusherCacheData * pCur
);
```

Parameters

pCur

The CFlusherCacheData structure returned by a previous call to GetStart or **GetNext**.

Return Value

Returns a pointer to the CFlusherCacheData structure for the next candidate for removal.

Remarks

Note that the cache has the last word on whether the item returned by this method will actually be removed from the cache. The flusher must not assume that an item returned from this method or GetStart has been removed. A call to Remove will notify the flusher when an item has been removed.

See Also

Reference

Cache Flusher Archetype

Other Resources

Caching Reference

CacheFlusherArchetype::GetStart

Called by the cache to get the first item that can be removed from the cache.

```
CFlusherCacheData * GetStart( );
```

Return Value

Returns a pointer to the CFlusherCacheData structure for the first candidate for removal.

Remarks

Note that the cache has the last word on whether the item returned by this method will actually be removed from the cache. The flusher must not assume that an item returned from this method or GetNext has been removed. A call to Remove will notify the flusher when an item has been removed.

See Also

Reference

Cache Flusher Archetype

Other Resources

Caching Reference

CacheFlusherArchetype::Release

Called by the cache when the reference count of an item is decremented.

```
void Release(
   CFlusherCacheData * pItem
);
```

Parameters

pItem

The CFlusherCacheData structure corresponding to the item that has been released.

See Also

Reference

Cache Flusher Archetype

Other Resources

Caching Reference

CacheFlusherArchetype::Remove

Called by the cache when an item has been removed from the cache.

```
void Remove(
   CFlusherCacheData * pItem
);
```

Parameters

pItem

The CFlusherCacheData structure corresponding to the item that has been removed.

See Also

Reference

Cache Flusher Archetype

Other Resources

Caching Reference

Cache Statistics Archetype

Classes that conform to the *cache statistics* archetype can be used to collect and expose statistics related to the use of a cache including such information as effectiveness of the cache and memory used.

Implementation

To implement a class conforming to this archetype, provide the same public interface as CNoStatClass.

Existing Implementations

These classes conform to this archetype:

Class	Description
CNoStatClass	Nonoperative implementation. Cache-hit statistics not available.
CStdStatClass	Cache-hit statistics available through IMemoryCacheStats. Uses interlocked operations.
CPerfStatClass	Cache-hit statistics available through IMemoryCacheStats and performance monitor counters. Uses interlocked operations.

Note that classes that use interlocked operations to keep track of statistics should be used with caution because the locking can affect server performance and scalability.

Use

These template parameters expect the class to conform to this archetype:

Parameter name	Used by
CPageCacheStats	CIsapiExtension
CStencilCacheStats	CIsapiExtension
StatClass	CMemoryCacheBase
StatClass	CMemoryCache
StatClass	CBlobCache
StatClass	CStencilCache
StatClass	CFileCache

See Also

Reference

ATL Server Archetypes

Request Statistics Archetype

Other Resources

ATL Server Concepts

ATL Server Reference

Caching Reference

DLL Cache Peer Archetype

Classes that conform to the *DLL cache peer* archetype can be used to describe extra data to be associated with each cached DLL held by CDllCache and will be notified when DLLs are added to or removed from the cache.

Implementation

To implement a class conforming to this archetype, the class must provide the following features:

Method	Description
Add	This method will be called when a DLL is being added to the cache.
Remove	This method will be called when a DLL is being removed from the cache.

Type	Description
DllInfo	The type used for holding extra data related to each DLL in the cache.

Existing Implementations

These classes conform to this archetype:

Class	Description
CDllCachePeer	This class is used to cache pointers to ATL Server request handler functions.
CNoDllCachePeer	This class can be used when no extra data needs to be stored with the DLLs in the cache.

Use

These template parameters expect the class to conform to this archetype:

Parameter name	Used by
Peer	CDllCache

See Also

Reference

ATL Server Archetypes

File Cache Peer Archetype

Other Resources

ATL Server Reference

Caching

Caching Reference

DLLCachePeerArchetype::Add

This method will be called when a DLL is being added to the cache.

```
BOOL Add(
   HINSTANCE hInst,
   DllInfo* pInfo
) throw(...);
```

Parameters

hInst

The handle of the DLL being added to the cache.

pInfo

A pointer to the extra information associated with this DLL. See DllInfo for details.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

The class can execute any code that it deems necessary in this method. If the method returns **FALSE**, the DLL will not be added to the cache.

See Also

Reference

DLL Cache Peer Archetype

Other Resources

Caching

Caching Reference

DLLCachePeerArchetype::DllInfo

The type used for holding extra data related to each DLL in the cache.

```
typedef UserDefinedType DllInfo;
```

Remarks

DllInfo can be a typedef or nested type. This type will typically provide members for storing extra data related to each DLL in the cache. This is the same **DllInfo** used by the Add and Remove methods.

See Also

Reference

DLL Cache Peer Archetype

Other Resources

Caching

Caching Reference

DLLCachePeerArchetype::Remove

This method will be called when a DLL is being removed from the cache.

```
void Remove(
   HINSTANCE hInst,
   DllInfo* pInfo
) throw(...);
```

Parameters

hInst

The handle of the DLL being removed from the cache.

pInfo

A pointer to the extra information associated with this DLL. For more information, see DllInfo.

Remarks

The class can execute any code that it deems necessary in this method.

See Also

Reference

DLL Cache Peer Archetype

Other Resources

Caching

Caching Reference

File Cache Peer Archetype

Classes that conform to the *file cache peer* archetype can be used to describe extra data to be associated with each cached file held by CFileCache and will be notified when files are added to or removed from the cache.

Implementation

To implement a class conforming to this archetype, the class must provide the following features:

Method	Description
Add	Called when a file is being added to the cache.
Remove	Called when a file is being removed from the cache.

Type	Description
PeerInfo	The type used for holding extra data related to each file in the cache.

Existing Implementations

These classes conform to this archetype:

Class	Description
CNoFileCachePeer	Used as placeholder when no file cache peer is required.
CPageCachePeer	Used internally by ATL Server to manage auxiliary file cache information

Use

These template parameters expect the class to conform to this archetype:

Parameter name	Used by
FileCachePeer	CFileCache
Peer	CCacheDataPeer

See Also

Reference

ATL Server Archetypes

DLL Cache Peer Archetype

Other Resources

ATL Server Reference

Caching

Caching Reference

FileCachePeerArchetype::Add

Called when a file is being added to the cache.

```
static BOOL Add(
   PeerInfo* pDest,
   PeerInfo* pSrc
);
```

Parameters

pDest

The address of the newly allocated memory within the cache where peer data is to be stored. See PeerInfo.

pSrc

The address of the peer data for the file being added to the cache. See PeerInfo.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This method serves as a custom assignment mechanism for file peer data. The data located at *pSrc* should be copied to the location specified by *pDest*.

See Also

Reference

File Cache Peer Archetype

Other Resources

Caching

Caching Reference

FileCachePeerArchetype::PeerInfo

The type used for holding extra data related to each file in the cache.

```
typedef UserDefinedType PeerInfo;
```

Remarks

PeerInfo can be a typedef or nested type. This type will typically provide members for storing extra data related to each file in the cache. This is the same data type used in the Add and Remove methods.

See Also

Reference

File Cache Peer Archetype

Other Resources

Caching

Caching Reference

FileCachePeerArchetype::Remove

Called when a file is being removed from the cache.

```
static BOOL Remove(
   PeerInfo* pPeerInfo
);
```

Parameters

pPeerInfo

The address of the file peer data corresponding to the file being removed from the file cache. See PeerInfo.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

See Also

Reference

File Cache Peer Archetype

Other Resources

Caching

Caching Reference

Request Statistics Archetype

Classes that conform to the *request statistics* archetype can be used to collect and expose statistics related to the use of an ISAPI extension, including such information as number of requests handled and average time per request.

Implementation

To implement a class conforming to this archetype, provide the same public interface as CNoRequestStats.

Existing Implementations

These classes conform to this archetype:

Class	Description
CNoRequestStats	Nonoperative implementation. Request handling statistics not available.
CStdRequestStats	Request handling statistics available through IRequestStats. Uses interlocked operations.
CPerfMonRequestStats	Request handling statistics available through IRequestStats and performance monitor counters. Uses i nterlocked operations.

Note that classes that use interlocked operations to keep track of statistics should be used with caution because the locking can affect server performance and scalability.

Use

These template parameters expect the class to conform to this archetype:

Parameter name	Used by
CRequestStatClass	CIsapiExtension

See Also

Reference

ATL Server Archetypes

Cache Statistics Archetype

Other Resources

ATL Server Concepts

ATL Server Reference

SOAP Client Archetype

Classes that conform to the *SOAP client* archetype can be used to send SOAP messages to a server when used with SPROXY-generated proxy classes.

Implementation

To implement a class conforming to this archetype, provide the following features:

Method	Description
CleanupClient	Clear the data members used to hold state about the current request and response.
Constructor	The constructor should accept a string identifying the location of the SOAP server.
GetClientError	Return the status code for the current request.
GetClientReader	Return a pointer to the **ISAXXMLReader** interface used to read the SOAP response. A typical implementation will just create a SAXXMLReader30 object.
GetReadStream	Return a pointer to the IStream interface of the response stream.
GetWriteStream	Return a pointer to the IWriteStream interface of the request stream.
SendRequest	Send the SOAP message contained in the stream returned by GetWriteStream to the server.
SetClientError	Set the status code for the current request.

Existing Implementations

These classes conform to this archetype:

Class	Description
CSoapMSXMLInetClient	This class provides client access to XML Web services using the Microsoft XML 3.0 ServerXMLHTTP object.
CSoapSocketClientT	This class provides client access to XML Web services using sockets.
CSoapWininetClient	This class provides client access to XML Web services using the WinInet API.

Note that the SOAPTransport Sample provides a number of other classes that conform to the *SOAP client* archetype. All supported ATL Server client classes communicate using HTTP, but the *SOAP client* classes in that sample can communicate using sockets over any port, Microsoft Message Queue, or the file system.

Use

These template parameters expect the class to conform to this archetype:

Parameter name	Used by
TClient	SPROXY-generated proxy class.

See Also

Reference

ATL Server Archetypes

XML Web Service Client Archetype

Other Resources

ATL Server Concepts

ATL Server Reference

Worker Archetype

Classes that conform to the *worker* archetype provide the code to process work items queued on a thread pool.

Implementation

To implement a class conforming to this archetype, the class must provide the following features:

Method	Description
Initialize	Called to initialize the worker object before any requests are passed to Execute.
Execute	Called to process a work item.
Terminate	Called to uninitialize the worker object after all requests have been passed to Execute.

Typedef	Description
RequestType	A typedef for the type of work item that can be processed by the worker class.

A typical *worker* class looks like this:

```
class CMyWorker
{
public:
   typedef MyRequestType RequestType;

   BOOL Initialize(void* pvWorkerParam);

   void Execute(
      MyRequestType request,
      void* pvWorkerParam,
      OVERLAPPED* pOverlapped
   );

   void Terminate(void* pvWorkerParam);
};
```

Existing Implementations

These classes conform to this archetype:

Class	Description
CNonStatelessWorker	Receives requests from the thread pool and passes them on to a worker object that is created and destroyed for each request.
CIsapiWorker	Receives ATL Server requests from the thread pool and passes them on to an object implementing IIsapiExtension.

Use

These template parameters expect the class to conform to this archetype:

Parameter name	Used by
Worker	CThreadPool
Worker	CNonStatelessWorker

Example

See the PooledAsync Sample.

See Also

Reference

ATL Server Archetypes

Other Resources

ATL Server Concepts

ATL Server Reference

WorkerArchetype::Execute

Called to process a work item.

```
void Execute(
   RequestType request,
   void* pvWorkerParam,
   OVERLAPPED* pOverlapped
);
```

Parameters

request

The work item to be processed. The work item is of the same type as RequestType.

pvWorkerParam

A custom parameter understood by the worker class. Also passed to Initialize and Terminate.

pOverlapped

A pointer to the OVERLAPPED structure used to create the queue on which work items were queued.

Example

See the PooledAsync Sample.

See Also

Reference

Worker Archetype

WorkerArchetype::Initialize

Called to initialize the worker object before any requests are passed to Execute.

```
BOOL Initialize(
   void* pvParam
) throw( );
```

Parameters

pvParam

A custom parameter understood by the worker class. Also passed to Terminate and Execute.

Return Value

Return **TRUE** on success, **FALSE** on failure.

Example

See the PooledAsync Sample.

See Also

Reference

Worker Archetype

WorkerArchetype::RequestType

A typedef for the type of work item that can be processed by the worker class.

```
typedef MyRequestType RequestType;
```

Remarks

This type must be used as the first parameter of Execute and must be capable of being cast to and from a ULONG_PTR.

See Also

Reference

Worker Archetype

WorkerArchetype::Terminate

Called to uninitialize the worker object after all requests have been passed to Execute.

```
void Terminate(
   void* pvParam
) throw( );
```

Parameters

pvParam

A custom parameter understood by the worker class. Also passed to Initialize and Execute.

Example

See the PooledAsync Sample.

See Also

Reference

Worker Archetype

XML Web Service Client Archetype

Classes that conform to the *XML Web Service client* archetype conform to the SOAP client archetype and provide some additional Web-related features.

Implementation

To implement a class conforming to this archetype, the class must conform to the SOAP client archetype and provide the following additional features:

Method	Description
GetStatusCode	Call this method to get the HTTP status code returned by the server following a request.
GetUrl	Call this method to get the URL of the XML Web service.
SetProxy	Call this method to set the proxy used to connect to the XML Web service.
SetTimeout	Call this method to set the timeout for connecting, sending, or receiving.
SetUrl	Call this method to set the URL of the XML Web service.

Data Member	Description
m_fault	The SOAP Fault returned by the XML Web service following a request.

This is equivalent to implementing a class with the same public interface as CSoapMSXMLInetClient.

Existing Implementations

These classes conform to this archetype:

Class	Description
CSoapMSXMLInetClient	This class provides client access to XML Web services using the Microsoft XML 3.0 ServerXMLHTTP object
CSoapSocketClientT	This class provides client access to XML Web services using sockets
CSoapWininetClient	This class provides client access to XML Web services using the WinInet API.

See Also

Reference

ATL Server Archetypes

SOAP Client Archetype

Other Resources

ATL Server Concepts

ATL Server Reference

ATL Server Response File Reference

The elements of a server response file described in this section are those understood by the CHtmlStencil class. Further features can be added by deriving from this class and customizing the parsing code.

Server response files consist of the following elements:

Element	Description
Codepage Tag	A codepage tag can be added to a server response file to indicate the character set used in that file. Specifying the codepage allows the server response file to be parsed correctly if it contains characters that are not part of the ANSI character set.
Comment Tag	A comment tag allows you to add comments to the server response file that will be removed during processing.
Handler Tag	A handler tag identifies a request handler that will be used to render some or all of the replacement tags in that file.
Include Tag	An include tag can be added to a server response file to indicate the point at which the contents of another file should be included in the response.
Locale Tag	A locale tag can be added to a server response file to indicate the locale that should be used for any response generated from that point on in the file.
Replacement Tag	A replacement tag can be added to a server response file to indicate the point at which dynamic content should be inserted into a response. Replacement tags consist of two pairs of braces enclosing the name of a replacement method or a rendering instruction.
Subhandler Tag	A subhandler tag identifies a request handler that will be used to render some or all of the replacement tags in that file.
Static Content	Static content can be any text (most likely, but not limited to, HTML and XML) that doesn't clash with one of the parser's tags.

See Also

Tasks

SRFSyntax Sample: Server Response File Demonstration

SRFSyntax Sample: Server Response File Demonstration

Other Resources

ATL Server Reference

Codepage Tag

A codepage tag can be added to a server response file (SRF) to indicate the character set used in that file. Specifying the codepage allows the SRF to be parsed correctly if it contains characters that are not part of the ANSI character set.

```
{{ codepage codepage_id }}
```

Definition

codepage_id

The codepage as a decimal integer or as an ISO- or IANA-defined character set name.

Remarks

There can be zero or one codepage tag in a single SRF. If present, the codepage tag should appear early in the file to ensure that the rest of the SRF is parsed correctly.

If the codepage tag is not present, the SRF is parsed using the ANSI codepage associated with the default locale or the most recently seen locale specified in a locale tag in the SRF.

Note that UCS-4, UCS-2, and UTF-16 encoded SRFs are not supported. Multilingual responses can be generated using the UTF-8 code page.

See ATL_NO_MLANG for information that can help you minimize dependencies and improve the performance of your request handlers at the cost of removing the ability to specify the codepage and locale as strings.

Example

```
{{codepage utf-8}}
```

See Also

Reference

Comment Tag
Handler Tag
Include Tag
Locale Tag
Replacement Tag
Subhandler Tag
Static Content

Other Resources

ATL Server Response File Reference

Comment Tag

A comment tag allows you to add comments to the server response file (SRF) that will be removed during processing.

```
{{ // comment_text }}
{{ !-- comment_text }}
```

Definition

comment_text

Text that you don't want to appear in the output stream.

Remarks

Comment tags can be in one of two forms: the first form uses C++ comment syntax; the second form uses a style reminiscent of HTML comment syntax. Comment tags can appear at any point in the SRF, even before the handler tag.

Example

```
{{ // This text will not appear in the processed output}}
```

For another example, see the SRFSyntax Sample.

See Also

Reference

Codepage Tag
Handler Tag
Include Tag
Locale Tag
Replacement Tag
Subhandler Tag
Static Content

Other Resources

ATL Server Response File Reference

Handler Tag

A handler tag identifies a request handler that will be used to render some or all of the replacement tags in that file.

```
{{ handler dll_path / request_handler }}
```

Definitions

dll_path

The path to the DLL that contains the request handler that will render the replacement tags. Specify an absolute path or a path relative to the physical location of the SRF.

request_handler

The name of the request handler within the DLL specified by *dll_path* that will be used to render the replacement tags.

The *request_handler* string is mapped to a C++ request handler class by a HANDLER_ENTRY macro in the handler map of the DLL or by a request_handler attribute applied directly to the class.

Remarks

There should be exactly one handler tag in a SRF. Any handler tags following the first tag are silently ignored.

Example

```
{{ handler AuthorDetails.dll/Default }}
```

For another example, see the SRFSyntax Sample.

See Also

Tasks

SRFSyntax Sample: Server Response File Demonstration

Reference

Codepage Tag
Comment Tag
Include Tag
Locale Tag
Replacement Tag
Subhandler Tag
Static Content

Other Resources

ATL Server Response File Reference

Include Tag

An include tag can be added to a server response file (SRF) to indicate the point at which the contents of another file should be included in the response.

```
{{ include file_path }}
```

Definition

file_path

The path to the file whose content will be included in the response at the position of the include tag. Specify an absolute path or a path relative to the physical location of the SRF.

Remarks

If the file identified by *file_path* is a server response file, when the tag is reached during rendering the appropriate handlers are loaded based on the handler tags in that file and SRF rendering takes places. The *file_path* can include a query string in which case the query parameters from that string are added to the query parameters from the including SRF and made available to handlers as part of the request.

If the file identified by *file_path* is a DLL, the DLL is assumed to be the name of a web application DLL. The *file_path* can include a query string in which case the query parameters from that string are added to the query parameters from the including SRF for use by the request handler supplied by the DLL. The request handler can be specified by an initial query parameter of the form **Handler=***request_handler* where *request_handler* is the name of the request handler used to satisfy the request. If this parameter is omitted, the default request handler for that DLL is assumed.

Otherwise, the file is assumed to be entirely static, and the contents of the file are injected into the response in place of the include tag.

See the documentation for CIncludeServerContext to understand the limitations on the functionality of an included request handler.

Example

```
{{include MySRFFile.srf}}
```

For another example, see the SRFSyntax Sample.

See Also

Reference

Codepage Tag
Comment Tag
Handler Tag
Locale Tag
Replacement Tag
Subhandler Tag
Static Content

Other Resources

ATL Server Response File Reference

Locale Tag

A locale tag can be added to a server response file (SRF) to indicate the locale that should be used for any response generated from that point on in the file.

```
{{ locale locale_id }}
```

Definition

locale_id

The locale ID as a decimal integer or as an RFC 1766-compliant (http://ietf.org/rfc/rfc1766.txt) string.

Remarks

The thread locale set by a locale tag can be retrieved from within a replacement handler using GetThreadLocale. You may need the locale for loading resources, or formatting times, dates, and currencies. Use of this tag enables a single request handler method to be called to supply content localized for different languages or locations.

There can be as many locale tags in a single SRF as you find necessary.

See Codepage Tag for information about how locale tags affect the codepage used to parse the SRF.

See ATL_NO_MLANG for information that can help you minimize dependencies and improve the performance of your request handlers at the cost of removing the ability to specify the codepage and locale as strings.

Example

```
{{locale zh-cn}}
```

See Also

Reference

Codepage Tag
Comment Tag
Handler Tag
Include Tag
Replacement Tag
Subhandler Tag
Static Content

Other Resources

ATL Server Response File Reference

Replacement Tag

A replacement tag can be added to a server response file (SRF) to indicate the point at which dynamic content should be inserted into a response. Replacement tags consist of two pairs of braces enclosing the name of a replacement method or a rendering instruction.

```
{{ [rendering_instruction] [handler_alias.]replacement_method[(argument_data)] }}
{{ rendering_instruction_end }}
```

Definitions

rendering_instruction(optional)

A control statement (**if**, **else**, or **while**) that can be used to affect the flow of the rendering. See Remarks for more details.

handler_alias(optional)

The alias of the request handler providing the replacement method identified by *replacement_method*. If *handler_alias* is missing, the replacement method must be provided by the handler specified in the handler tag. Otherwise, the request handler is obtained from the corresponding subhandler tag.

replacement_method

The name of the replacement method that will be called to render the tag.

The *replacement_method* string is mapped to a C++ method call by a REPLACEMENT_METHOD_ENTRY macro in the replacement method map of the request handler class or by a tag_name attribute. The name of this method is limited to ATL_MAX_METHOD_NAME_LEN characters.

argument_data(optional)

The data used to initialize the argument passed to the replacement method. See Passing An Argument To A Replacement Method for more details.

rendering_instruction_end

A control statement (**endif** or **endwhile**) used to indicate the end of a processing block. See Remarks for more details.

Remarks

Use a tag containing only a replacement method to indicate the point at which dynamic content should be inserted into a response.

Use the **if**, **else**, and **endif** rendering instructions to mark blocks of the SRF to be conditionally rendered. If the replacement method in an **if** tag returns HTTP_SUCCESS, then the content between the **if** and its matching **else** or **endif** tag will be rendered. If the replacement method returns HTTP_S_FALSE, the content in that block will be discarded.

Use the **while** and **endwhile** rendering instructions to mark a block of the SRF to be rendered in a loop. While the replacement method in a **while** tag returns HTTP_SUCCESS, the content between the **while** and its matching **endwhile** tag will be repeatedly rendered. The replacement method should return HTTP_S_FALSE to end the **while** loop.

Note that the content of **if** or **while** blocks is not limited to static text. Further replacement tags can be nested in these blocks. The maximum nesting is limited to ATL_MAX_BLOCK_STACK levels.

Example

```
{{if RenderThis}}
Here is some text
{{endif}}
```

In this example, the primary request handler's `RenderThis` method is called. If it returns HTTP_SUCCESS, the text `Here is some text` is rendered into the response stream. Otherwise, nothing is rendered into the response stream. The **if** tag and its matching **endif** tag will be removed from the response stream.

```
<table>
{{while GetNextBirthday}}
```

```
<tr><td>{{GetName}}</td><td>{{GetBirthday}}</td></tr>
{{endwhile}}
</table>
```

In this example, the primary replacement handler's `GetNextBirthday` method is called. While that method returns HTTP_SUCCESS, all the HTML in the **while** block (the table row) is repeatedly rendered. Note that tags have been nested here.

For another example, see the SRFSyntax Sample.

See Also

Tasks

SRFSyntax Sample: Server Response File Demonstration

Reference

Codepage Tag
Comment Tag
Handler Tag
Include Tag
Locale Tag
Subhandler Tag
Static Content

Other Resources

ATL Server Response File Reference

Subhandler Tag

A subhandler tag identifies a request handler that will be used to render some or all of the replacement tags in that file.

```
{{ subhandler handler_alias dll_path / request_handler }}
```

Definitions

handler_alias

A name used to identify this specific request handler in the replacement tag. The length of this name is limited to ATL_MAX_HANDLER_NAME_LEN characters.

dll_path

The path to the DLL that contains the request handler that will render the replacement tags. Specify an absolute path or a path relative to the physical location of the server response file (SRF).

request_handler

The name of the request handler within the DLL specified by *dll_path* that will be used to render the replacement tags.

The *request_handler* string is mapped to a C++ request handler class by a HANDLER_ENTRY macro in the handler map of the DLL or by a request_handler attribute applied directly to the class.

Remarks

There can be as many subhandler tags in a single SRF as you find necessary. Each subhandler tag should have a unique value for the *handler_alias*.

Example

```
{{ subhandler Book Book.dll/Default }}
```

For another example, see the SRFSyntax Sample.

See Also

Reference

Codepage Tag
Comment Tag
Handler Tag
Include Tag
Locale Tag
Replacement Tag
Static Content

Other Resources

ATL Server Response File Reference

Static Content

Static content can be any text (most likely, but not limited to, HTML and XML) that doesn't clash with one of the parser's tags.

Server response files (SRF) can be used to generate HTML, XML, or any other text-based content type. However, the response file parser places a few restrictions on the static content of the SRF.

Double braces ("{{" and "}}") are given special consideration by the response file parser. When it encounters opening braces, the parser treats all text up to one character beyond the first closing brace as a single token. If no closing brace is found, an error is returned.

Content enclosed in double braces is interpreted as being one of the tags described in this reference. If, after parsing the contents of a pair of double braces, it turns out that the content does not exactly match one of these tags (for example, if the text looks like a replacement tag, but the request handler does not support the replacement method used), the whole tag is treated as static text and will appear unchanged in the response. This also applies to tags that are linked to these tags.

Note that line breaks ('\n' or '\r\n') immediately following the tags listed below are discarded and not added to the generated response:

- Codepage Tag
- Comment Tag
- Handler Tag
- Include Tag
- Subhandler Tag

It is unlikely that the static content you want to integrate into the rendered output will clash with the requirements imposed by the parser. However, if it does, you can put that content into a file that does not use the SRF extension and use the include tag to integrate the content into the response.

Note that UCS-4, UCS-2, and UTF-16 encoded SRFs are not supported. Multilingual responses can be generated using the UTF-8 encoding.

See Also

Reference

Codepage Tag
Comment Tag
Handler Tag
Include Tag
Locale Tag
Replacement Tag
Subhandler Tag

Other Resources

ATL Server Response File Reference

Session-State Database Schema

For the database implementation of the ATL Server session-state services, the CDBSessionServiceImplT class communicates with a database through CDBSession. This class template in turn defaults to using CDefaultQueryClass to provide the SQL queries used for accessing the database. **CDefaultQueryClass** expects the database to have two tables, one for storing session variables named *SessionVariables*, and one for storing session reference information named *SessionReferences*. The schema of the two tables is defined as follows:

SessionVariables

Column	Name	Type	Description
1	SessionID	char[MAX_SESSION_KEY_LEN]	Session name
2	VariableName	char[MAX_VARIABLE_NAME_LENGTH]	Variable name
3	VariableValue	varbinary[MAX_VARIABLE_VALUE_LENGTH]	Variable value

SessionReferences

Column	Name	Type	Description
1	SessionID	char[MAX_SESSION_KEY_LEN]	Session name
2	LastAccess	Datetime	Date and time of last access
3	RefCount	INT	Count of current references on this session
4	TimeoutMS	Int	Timeout value for the session

If SQL Server is being used to host the session database, the SQL script, **SessionServices.sql**, which can be found in the \VC7\bin directory of your Visual Studio installation, can be used to create the necessary tables.

As an implementation class, CDBSessionServiceImplT serves a behind-the-scenes role and should be used in your code only as a template parameter to CSessionStateService. See Providing Session-State Services for more information.

See Also

Reference

CMemSessionServiceImpl Class

Other Resources

Session-State Services (ATL Server)

Session-State Reference

MFC Reference

The MFC Reference covers the classes, global functions, global variables, and macros that make up the Microsoft Foundation Class Library version 8.0.

The individual hierarchy charts included with each class are useful for locating base classes. The MFC Reference usually does not describe inherited member functions or inherited operators. For information on these functions, refer to the base classes depicted in the hierarchy diagrams.

The documentation for each class includes a class overview, a member summary by category, and topics for the member functions, overloaded operators, and data members.

Public and protected class members are documented only when they are normally used in application programs or derived classes. See the class header files for a complete listing of class members.

In This Section

Hierarchy Chart

Visually details the class relationships in the class library.

Class Library Overview

Lists the classes in the MFC Library according to category.

MFC Technical Notes

Provides links to specialized topics, written by the MFC development team, on the class library.

MFC Classes

Provides links to and header file information for the MFC classes.

MFC Macros and Globals

Provides links to the macros and global functions in the MFC Library.

Structures, Styles, Callbacks, and Message Maps

Provides links to the structures, styles, callbacks, and message maps used by the MFC Library.

Obsolete MFC Functions

Provides links to functions that are obsolete in MFC.

Related Sections

Hierarchy Chart Categories

Describes the MFC hierarchy chart by category.

Shared Classes

Provides links to classes that are shared between MFC and ATL.

MFC Samples

Provides links to samples that demonstrate how to use MFC.

Visual C++ Libraries

Provides links to the various libraries provided with Visual C++, including ATL, MFC, OLE DB Templates, the C run-time library, and the Standard C++ Library.

Debugging

Provides links to using the Visual Studio debugger to correct logic errors in your application or stored procedures.

Breaking Changes (MFC)

Breaking changes in MFC.

Breaking Changes

The *plRequestNumber* parameter to AfxIsMemoryBlock will now be initialized to zero if it does not represent a currently active memory block.

CString operators << and >> that were global in afxstr.h have been moved to be a member operator in CArchive. The new operators are:

```
template< typename BaseType, class StringTraits >
CArchive& operator<<(const ATL::CStringT<BaseType, StringTraits>& str)
template< typename BaseType, class StringTraits >
CArchive& operator>>(ATL::CStringT<BaseType, StringTraits>& str)
```

This change breaks user code in the following scenarios:

- Existing code that already specializes the global operator << , so specialization doesn't have the primary template anymore.
- Existing code that calls ::operator <<(ar,str); (instead of using the ar << str syntax).
- Create and CreateEx methods are no longer virtual on several MFC control classes: CStatusBarCtrl Class, CListCtrl Class, CTreeCtrl Class, CToolBarCtrl Class, CReBarCtrl Class, CRichEditCtrl Class, CEdit Class. These virtual methods were added in the previous version but broke user code, so they have been removed.

See Also

Concepts

What's New in the Visual C++ Libraries

Other Resources

MFC Reference

MFC Concepts

This section provides conceptual and task-based topics to help you program using the Microsoft Foundation Class (MFC) Library.

In This Section

General MFC Topics

Discusses the technical details of the MFC Library.

MFC Overview

Provides links to topics discussing the programming you can do with MFC.

Using CObject

Provides links to using **CObject**, the base class for most classes in MFC.

Collections

Discusses collection classes created from and not created from C++ templates.

Date and Time

Provides links to topics discussing using date and time with MFC.

Files in MFC

Discusses **CFile** and how to handle files in MFC.

Memory Management (MFC)

Describes how to take advantage of the general-purpose services related to memory management.

Message Handling and Mapping

Describes how messages and commands are processed by the MFC framework and how to connect them to their handler functions.

Serialization in MFC

Explains the serialization mechanism provided to allow objects to persist between runs of your program.

Unicode

Describes MFC support for the Unicode standard for encoding wide characters on Windows NT, Windows 2000, and Windows XP platforms.

Related Sections

Exception Handling (MFC)

Explains the exception-handling mechanisms available in MFC.

MFC Internet Programming Basics

Discusses the MFC classes that support Internet programming.

MFC COM

Discusses a subset of MFC, which is designed to support COM, while most of the Active Template Library (ATL) is designed for COM programming.

Multithreading with C++ and MFC

Describes what processes and threads are and discusses the MFC approach to multithreading.

Windows Sockets in MFC

Covers the MFC implementation of Windows Sockets.

Adding Functionality

Provides links to topics describing conceptual information about the Visual C++ libraries and topics discussing various

coding technologies and techniques.

MFC Reference

Provides reference material for the MFC Library, a set of classes that constitute an application framework, which is the framework of an application written for the Windows API.

MFC Samples

Provides links to samples that show how to use MFC in desktop applications, DLLs, database applications, controls, Web applications, and more.

MFC Overview

For an overview of the kinds of programming you can do with MFC and ideas on how to get started learning MFC, see the following topics:

- MFC Fundamentals
- Key MFC Programming Areas
- Prerequisites for Learning MFC

See Also

Concepts

General MFC Topics

MFC Fundamentals

MFC's strong suit is its fundamental support for programming for Microsoft Windows.

In This Section

Frame Windows

Documents

Views of Documents

Multiple Views

Special View Types, such as scroll views and form views

Dialog Boxes and Property Sheets

Windows Common Controls

Mapping Windows Messages to handler functions

Toolbars and Control Bars

Printing and Print Preview

Serialization of data to and from files and other media

Device Contexts and GDI Drawing Objects

Exception Handling in MFC

Collections of data objects

Memory Diagnostics

Strings, Rectangles, and Points

Date and Time

CArchive Tasks

Files in MFC

WinHelp: Context-Sensitive Help for Your Programs

HTML Help: Context-Sensitive Help for Your Programs

See Also

Other Resources

MFC Overview

Key MFC Programming Areas

What MFC Cannot Do for You

As a general programming framework, MFC cannot anticipate every programmer's every need. For example, MFC makes it easy to build the interface for a spreadsheet application, but you must provide all the important display and computation logic.

Note
MFC is not a general function library, like the C run-time library. You cannot call MFC class member functions in an otherwise non-MFC context. From within MFC, you can still call Win32 API functions directly, particularly those that MFC does not choose to encapsulate. But most MFC functions are members of a class, and you must have an object of the class before you can call any of its member functions.

MFC was designed to be a class library for building applications for the Windows operating system. The goal and design of MFC targets the traditional desktop productivity applications used every day. Because people are so productive with MFC, it is tempting to try and use it for application types not intended to be built with MFC. One such application type is a Windows service. Although it is possible to build a Windows service using pieces of MFC, you will need to be very careful about which pieces you use. The Microsoft Knowledge Base has some information on problems you may run into; however, there are many more that are not documented. Microsoft does not support using MFC to build Windows services.

What MFC Can Do for You

Despite its generality, MFC does support you in many specialized ways:

Support For	See
OLE visual editing	OLE
Automation	Automation
ActiveX Controls	MFC ActiveX Controls
Internet programming	Overview: Internet
Windows Common Controls	Controls
ODBC Database Programming	Open Database Connectivity (ODBC)
Multithreaded Programming	Multithreading
Network Programming	Windows Sockets
Portability	Porting Your Program

See Also

Other Resources

MFC Overview

Prerequisites for Learning MFC

Among the important assumptions made by the MFC documentation are that:

- You already know a little about programming for Windows.
- You know the basics of programming in C++.
- You understand the fundamentals of object-oriented programming.

For more information about MFC, see MFC Fundamentals.

See Also

Other Resources

MFC Overview

General MFC Topics

This family of articles includes technical details about the Microsoft Foundation Class (MFC) Library and provides an overview of the MFC framework and its key components and subsystems.

The Microsoft Foundation Class Library is an application framework for programming in Microsoft Windows. Written in C++, MFC provides much of the code necessary for managing windows, menus, and dialog boxes; performing basic input/output; storing collections of data objects; and so on. All you need to do is add your application-specific code into this framework. Given the nature of C++ class programming, it is easy to extend or override the basic functionality that the MFC framework supplies.

The MFC framework is a powerful approach that lets you build upon the work of expert programmers for Windows. MFC shortens development time; makes code more portable; provides tremendous support without reducing programming freedom and flexibility; and gives easy access to "hard to program" user-interface elements and technologies, like Active technology, OLE, and Internet programming. Furthermore, MFC simplifies database programming through Data Access Objects (DAO) and Open Database Connectivity (ODBC), and network programming through Windows Sockets. MFC makes it easy to program features like property sheets ("tab dialogs"), print preview, and floating, customizable toolbars.

In This Section

MFC Samples

MFC Overview

MFC Fundamentals

Key MFC Programming Areas

Prerequisites for Learning MFC

Using the MFC Source Files, which are supplied with Visual C++

Using the Source Code Browser with the MFC Browse Information File (MFC.bsc)

MFC Library Versions

Building on the Framework

Managing the State Data of MFC Modules

Idle Loop Processing in MFC

Using MFC to Write Windows Programs

Building on the MFC Framework

How the Framework Calls Code

CWinApp: The Application Class

Document Templates and the Document/View Creation Process

Creating Windows with Class CWnd

Working with Window Objects

Working with Windows Device Contexts

Working with Windows Graphic Objects

Adding User-Interface Features

Creating a Project

For an overview of the MFC reference documentation, see Microsoft Foundation Class Library.

For information about ATL, see Active Template Library Reference.

See Also

Other Resources

General Concepts

Using the MFC Source Files

The Microsoft Foundation Class (MFC) Library supplies full source code. Header files (.h) are in the \atlmfc\include directory; implementation files (.cpp) are in the \atlmfc\src\mfc directory.

Note

The \atlmfc\src\mfc directory contains a makefile you can use with NMAKE to build MFC library versions, including a browse version. A browse version of MFC is useful for tracing through the calling structure of MFC itself. The file Readme.Txt in that directory explains how to use this makefile.

This family of articles explains the conventions that MFC uses to comment the various parts of each class, what these comments mean, and what you should expect to find in each section. The Visual C++ wizards use similar conventions for the classes that they create for you, and you will probably find these conventions useful for your own code.

You might be familiar with the **public**, **protected**, and **private** C++ keywords. When looking at the MFC header files, you will find that each class may have several of each of these. For example, public member variables and functions might be under more than one **public** keyword. This is because MFC separates member variables and functions based on their use, not by the type of access allowed. MFC uses **private** sparingly; even items considered implementation details are generally protected and many times are public. Although access to the implementation details is discouraged, MFC leaves the decision to you.

In both the MFC source files and the files that the MFC Application Wizard creates, you will find comments like these within class declarations (usually in this order):

```
// Constructors
// Attributes
// Operations
// Overridables
// Implementation
```

Topics covered in this family of articles include:

- An example of the comments
- The // Implementation comment
- The // Constructors comment
- The // Attributes comment
- The // Operations comment
- The // Overridables comment

See Also

Concepts

General MFC Topics

An Example of the Comments

The following partial listing of class **CStdioFile** uses most of the standard comments that MFC employs in its classes to divide class members by the ways they are used:

```
class CStdioFile : public CFile
{
    DECLARE_DYNAMIC(CStdioFile)

public:
// Constructors
    CStdioFile();
...

// Attributes
    FILE* m_pStream;    // stdio FILE
...

// Operations
    virtual void WriteString(LPCTSTR lpsz);
...
    virtual LPTSTR ReadString(LPTSTR lpsz, UINT nMax);
...

// Implementation
public:
...
};
```

These comments consistently mark sections of the class declaration that contain similar kinds of class members. Keep in mind that these are MFC conventions, not set rules.

See Also

Concepts

Using the MFC Source Files
The // Implementation Comment
The // Constructors Comment
The // Attributes Comment
The // Operations Comment
The // Overridables Comment

The // Implementation Comment

The `// Implementation` section is the most important part of any MFC class declaration.

This section houses all implementation details. Both member variables and member functions can appear in this section. Everything below this line could change in a future release of MFC. Unless you cannot avoid it, you should not rely on details below the `// Implementation` line. In addition, members declared below the implementation line are not documented, although some implementation is discussed in technical notes. Overrides of virtual functions in the base class reside in this section, regardless of which section the base class function is defined in, because the fact that a function overrides the base class implementation is considered an implementation detail. Typically, these members are protected, but not always.

Notice from the **CStdioFile** listing under An Example of the Comments that members declared below the `// Implementation` comment may be declared as **public**, **protected**, or **private**. You should only use these members with caution, because they may change in the future. Declaring a group of members as **public** may be necessary for the class library implementation to work correctly. However, this does not mean that you may safely use the members so declared.

Note
You may find comments of the remaining types either above or below the `// Implementation` comment. In either case, they describe the kinds of members declared below them. If they occur below the `// Implementation` comment, you should assume that the members may change in future versions of MFC.

See Also

Concepts

Using the MFC Source Files
An Example of the Comments
The // Constructors Comment
The // Attributes Comment
The // Operations Comment
The // Overridables Comment

The // Constructors Comment

The `// Constructors` section of an MFC class declaration declares constructors (in the C++ sense) as well as any initialization functions required to really use the object. For example, **CWnd::Create** is in the constructors section because before you use the **CWnd** object, it must be "fully constructed" by first calling the C++ constructor and then calling the **Create** function. Typically, these members are public.

For example, class **CStdioFile** has three constructors, one of which is shown in the listing under An Example of the Comments.

See Also

Concepts

Using the MFC Source Files
The // Implementation Comment
The // Attributes Comment
The // Operations Comment
The // Overridables Comment

The // Attributes Comment

The `// Attributes` section of an MFC class declaration contains the public attributes (or properties) of the object. Typically these are member variables, or Get/Set functions. The "Get" and "Set" functions may or may not be virtual. The "Get" functions are usually **const**, because in most cases they do not have side effects. These members are normally public; protected and private attributes are typically found in the implementation section.

In the sample listing from class **CStdioFile**, under An Example of the Comments, the list includes one member variable, **m_pStream**. Class **CDC** lists nearly 20 members under this comment.

Note
Large classes, such as **CDC** and **CWnd**, may have so many members that simply listing all the attributes in one group would not add much to clarity. In such cases, the class library uses other comments as headings to further delineate the members. For example, **CDC** uses `// Device-Context Functions`, `// Drawing Tool Functions`, `// Drawing Attribute Functions`, and more. Groups that represent attributes will follow the usual syntax described above. Many OLE classes have an implementation section called `// Interface Maps`.

See Also

Concepts

Using the MFC Source Files
An Example of the Comments
The // Implementation Comment
The // Constructors Comment
The // Operations Comment
The // Overridables Comment

The // Operations Comment

The `// Operations` section of an MFC class declaration contains member functions that you can call on the object to make it do things or perform actions (perform operations). These functions are typically non-**const** because they usually have side effects. They may be virtual or nonvirtual depending on the needs of the class. Typically, these members are public.

In the sample listing from class **CStdioFile**, in An Example of the Comments, the list includes two member functions under this comment: **ReadString** and **WriteString**.

As with attributes, operations can be further subdivided.

See Also

Concepts

Using the MFC Source Files
An Example of the Comments
The // Implementation Comment
The // Constructors Comment
The // Attributes Comment
The // Overridables Comment

The // Overridables Comment

The `// Overridables` section of an MFC class declaration contains virtual functions that you can override in a derived class when you need to modify the base class behavior. They are usually named starting with "On", although it is not strictly necessary. Functions here are designed to be overridden, and often implement or provide some sort of "callback" or "hook." Typically, these members are protected.

In MFC itself, pure virtual functions are always placed in this section. A pure virtual function in C++ is one of the form:

```
virtual void OnDraw( ) = 0;
```

In the sample listing from class **CStdioFile**, in An Example of the Comments, the list includes no overridables section. Class **CDocument**, on the other hand, lists approximately 10 overridable member functions.

In some classes, you may also see the comment `// Advanced Overridables`. These are functions that only advanced programmers should attempt to override. You will probably never need to override them.

Note
The conventions described in this article also work well, in general, for Automation (formerly known as OLE Automation) methods and properties. Automation methods are similar to MFC operations. Automation properties are similar to MFC attributes. Automation events (supported for ActiveX controls, formerly known as OLE controls) are similar to MFC overridable member functions.

See Also

Concepts

Using the MFC Source Files
An Example of the Comments
The // Implementation Comment
The // Constructors Comment
The // Attributes Comment
The // Operations Comment

Viewing the MFC Browse Information File

MFC.bsc is the browse information file for the Microsoft Foundation Class Library. You can open MFC.bsc in the Object Browser to look through the MFC source code for debugging purposes or to get a better idea of what is going on behind the scenes.

Not all MFC classes and symbols are included in the browse information file. For example, the ISAPI classes are not included because they are in a separate library. If you want to view symbols that are not in the browse file, the **Find In Files** command on the **File** menu is another good way of locating items.

MFC.bsc is not installed by the setup program, but you can copy it manually from the distribution disk. From the root CD directory, the path is \Program Files\Microsoft Visual Studio .NET 2003\Vc7\atlmfc\src\mfc\MFC.bsc. Copy this file to the same directory on your hard disk where Visual C++ is installed. Once installed, the file can be opened and used the same as any other browse information file.

For more information, see Building Browse Information Files.

See Also

Concepts

General MFC Topics

MFC Library Versions

This article provides information on available versions of the Microsoft Foundation Class (MFC) Library. Topics covered include:

- Automatic linking of MFC library version
- Library naming conventions
- AFXDLL versions
- Dynamic-link library (DLL) support

See also the following Knowledge Base article at http://support.microsoft.com:

- Q249851 : HOWTO: Include Win32 Header Files in MFC Applications

See Also

Concepts

General MFC Topics

Automatic Linking of MFC Library Version

In versions of MFC before version 3.0 (before Visual C++ version 2.0), you had to manually specify the correct version of the MFC library in the input list of libraries for the linker. With MFC version 3.0 and later, it is no longer necessary to manually specify the version of the MFC library. Instead, the MFC header files automatically determine the correct version of the MFC library, based on values defined with **#define**, such as **_DEBUG** or **_UNICODE**. The MFC header files add **/defaultlib** directives instructing the linker to link in a specific version of the MFC library.

For example, the following code fragment from the AFX.H header file instructs the linker to link in either the NAFXCWD.LIB or NAFXCW.LIB version of MFC, depending on whether you are using the debug version of MFC:

```
...
#ifndef _UNICODE
    #ifdef _DEBUG
        #pragma comment(lib, "nafxcwd.lib")
    #else
        #pragma comment(lib, "nafxcw.lib")
    #endif
#else
    #ifdef _DEBUG
        #pragma comment(lib, "uafxcwd.lib")
    #else
        #pragma comment(lib, "uafxcw.lib")
    #endif
#endif...
```

MFC header files also link in all required libraries, including MFC libraries, Win32 libraries, OLE libraries, OLE libraries built from samples, ODBC libraries, and so on. The Win32 libraries include Kernel32.Lib, User32.Lib, and GDI32.Lib.

See Also

Concepts

MFC Library Versions

Library Naming Conventions

Object-code libraries for MFC use the following naming conventions. The library names have the form

*u*AFX*c*W*d*.LIB

where the letters shown in italic lowercase are placeholders for specifiers whose meanings are shown in the following table:

Library Naming Conventions

Specifier	Values and meanings
u	ANSI (N) or Unicode (U)
c	Type of program to create: C=all
d	Debug or Release: D=Debug; omit specifier for Release

The default is to build a debug Windows ANSI application for the Intel platform: NAFXCWD.Lib. All libraries listed in the following table are included prebuilt in the \atlmfc\lib directory on the Visual C++ CD-ROM.

Static-Link Library Naming Conventions

Library	Description
NAFXCW.LIB	MFC Static-Link Library, Release version
NAFXCWD.LIB	MFC Static-Link Library, Debug version
UAFXCW.LIB	MFC Static-Link Library with Unicode support, Release version
UAFXCWD.LIB	MFC Static-Link Library with Unicode support, Debug version

Note

If you need to build a library version, see the Readme.Txt file in the \atlmfc\src\mfc directory. This file describes using the supplied makefile with NMAKE.

For more information, see Naming Conventions for MFC DLLs and Unicode Versions of the MFC Libraries.

See Also

Concepts

MFC Library Versions

AFXDLL Versions

Instead of building your application by statically linking to the MFC object-code libraries, you can build your application to use one of the AFXDLL libraries, which contain MFC in a DLL that multiple running applications can share. For a table of AFXDLL names, see DLLs: Naming Conventions.

Note
By default, the MFC Application Wizard creates an AFXDLL project. To use static linking of MFC code instead, set the **Use MFC in a static library** option in the MFC Application Wizard. Static linking is not available in the Standard Edition of Visual C++.

See Also

Concepts

MFC Library Versions

Dynamic-Link Library Support

The NAFXCW.lib and NAFXCWD.lib libraries (listed in the Static-Link Library Naming Conventions table in Library Naming Conventions) create your project as a dynamic-link library, called a "Regular DLL" (formerly a "USRDLL") that can be used with applications not built with the class library. This DLL support is not the same as MFCx0.DLL and MFCx0D.DLL (known as AFXDLL), which contain the entire class library in a DLL. For more information, see DLLs. For a table of DLL names, see Naming Conventions for MFC DLLs.

See Also

Concepts

MFC Library Versions

Using the Classes to Write Applications for Windows

Taken together, the classes in the Microsoft Foundation Class (MFC) Library make up an "application framework," on which you build an application for the Windows operating system. At a very general level, the framework defines the skeleton of an application and supplies standard user-interface implementations that can be placed onto the skeleton. Your job as programmer is to fill in the rest of the skeleton, which are those things that are specific to your application. You can get a head start by using the MFC Application Wizard to create the files for a very thorough starter application. You use the Microsoft Visual C++ resource editors to design your user-interface elements visually, Class View commands to connect those elements to code, and the class library to implement your application-specific logic.

Version 3.0 and later of the MFC framework supports programming for Win32 platforms, including Microsoft Windows 95 and later, and Windows NT versions 3.51 and later. MFC Win32 support includes multithreading. Use version 1.5*x* if you need to do 16-bit programming.

This family of articles presents a broad overview of the application framework. It also explores the major objects that make up your application and how they are created. Among the topics covered in these articles are the following:

- The framework.
- Division of labor between the framework and your code, as described in Building on the Framework.
- The application class, which encapsulates application-level functionality.
- How document templates create and manage documents and their associated views and frame windows.
- Class CWnd, the root base class of all windows.
- Graphic objects, such as pens and brushes.

Other parts of the framework include:

- Window Objects: Overview
- Message handling and mapping
- CObject, The Root Base Class in MFC
- Document/View Architecture
- Dialog Boxes
- Controls
- Control Bars
- OLE
- Memory Management

  Besides giving you an advantage in writing applications for the Windows operating system, MFC also makes it much easier to write applications that specifically use OLE linking and embedding technology. You can make your application an OLE visual editing container, an OLE visual editing server, or both, and you can add Automation so that other applications can use objects from your application or even drive it remotely.

- MFC ActiveX Controls

  The OLE control development kit (CDK) is now fully integrated with the framework. This article family supplies an overview of ActiveX control development with MFC. (ActiveX controls were formerly known as OLE controls.)

- Database Programming

  MFC also supplies two sets of database classes that simplify writing data-access applications. Using the ODBC database classes, you can connect to databases through an Open Database Connectivity (ODBC) driver, select records from tables, and display record information in an on-screen form. Using the Data Access Object (DAO) classes, you can work with databases through the Microsoft Jet database engine or external (non-Jet) data sources, including ODBC data sources.

  In addition, MFC is fully enabled for writing applications that use Unicode and multibyte character sets (MBCS), specifically double-byte character sets (DBCS).

For a general guide to MFC documentation, see General MFC Topics.

See Also

Concepts

General MFC Topics

The Framework

Your work with the Microsoft Foundation Class (MFC) Library framework is based largely on a few major classes and several Visual C++ tools. Some classes encapsulate a large portion of the Win32 application programming interface (API). Other classes encapsulate application concepts such as documents, views, and the application itself. Still others encapsulate OLE features and ODBC and DAO data-access functionality.

For example, Win32's concept of window is encapsulated by MFC class **CWnd**. That is, a C++ class called **CWnd** encapsulates or "wraps" the **HWND** handle that represents a Windows window. Likewise, class **CDialog** encapsulates Win32 dialog boxes.

Encapsulation means that the C++ class **CWnd**, for example, contains a member variable of type **HWND**, and the class's member functions encapsulate calls to Win32 functions that take an **HWND** as a parameter. The class member functions typically have the same name as the Win32 function they encapsulate.

In This Section

SDI and MDI

Documents, Views, and the Framework

Wizards and Resource Editors

In Related Sections

Building on the Framework

How the Framework Calls Your Code

CWinApp: The Application Class

Document Templates and the Document/View Creation Process

Message Handling and Mapping

Window Objects

See Also

Concepts

Using the Classes to Write Applications for Windows

SDI and MDI

MFC makes it easy to work with both single-document interface (SDI) and multiple-document interface (MDI) applications.

SDI applications allow only one open document frame window at a time. MDI applications allow multiple document frame windows to be open in the same instance of an application. An MDI application has a window within which multiple MDI child windows, which are frame windows themselves, can be opened, each containing a separate document. In some applications, the child windows can be of different types, such as chart windows and spreadsheet windows. In that case, the menu bar can change as MDI child windows of different types are activated.

Note
Under Windows 95 and later, applications are commonly SDI because the operating system has adopted a "document-centered" view.

For more information, see Documents, Views, and the Framework.

See Also

Concepts

Using the Classes to Write Applications for Windows

Documents, Views, and the Framework

At the heart of the MFC framework are the concepts of document and view. A document is a data object with which the user interacts in an editing session. It is created by the **New** or **Open** command on the **File** menu and is typically saved in a file. (Standard MFC documents, derived from class **CDocument**, are different from Active documents and OLE compound documents.) A view is a window object through which the user interacts with a document.

The key objects in a running application are:

- The document or documents.

  Your document class (derived from CDocument) specifies your application's data.

  If you want OLE functionality in your application, derive your document class from COleDocument or one of its derived classes, depending on the type of functionality you need.

- The view or views.

  Your view class (derived from CView) is the user's "window on the data." The view class controls how the user sees your document's data and interacts with it. In some cases, you may want a document to have multiple views of the data.

  If you need scrolling, derive from CScrollView. If your view has a user interface that is laid out in a dialog-template resource, derive from CFormView. For simple text data, use or derive from CEditView. For a form-based data-access application, such as a data-entry program, derive from CRecordView (for ODBC). Also available are classes CTreeView, CListView, and CRichEditView.

- The frame windows

  Views are displayed inside "document frame windows." In an SDI application, the document frame window is also the "main frame window" for the application. In an MDI application, document windows are child windows displayed inside a main frame window. Your derived main frame-window class specifies the styles and other characteristics of the frame windows that contain your views. If you need to customize frame windows, derive from CFrameWnd to customize the document frame window for SDI applications. Derive from CMDIFrameWnd to customize the main frame window for MDI applications. Also derive a class from CMDIChildWnd to customize each distinct kind of MDI document frame windows that your application supports.

- The document template or templates

  A document template orchestrates the creation of documents, views, and frame windows. A particular document-template class, derived from class CDocTemplate, creates and manages all open documents of one type. Applications that support more than one type of document have multiple document templates. Use class CSingleDocTemplate for SDI applications, or use class CMultiDocTemplate for MDI applications.

- The application object

  Your application class (derived from CWinApp) controls all of the objects above and specifies application behavior such as initialization and cleanup. The application's one and only application object creates and manages the document templates for any document types the application supports.

- Thread objects

  If your application creates separate threads of execution — for example, to perform calculations in the background — you'll use classes derived from CWinThread. CWinApp itself is derived from **CWinThread** and represents the primary thread of execution (or the main process) in your application. You can also use MFC in secondary threads.

In a running application, these objects cooperatively respond to user actions, bound together by commands and other messages. A single application object manages one or more document templates. Each document template creates and manages one or more documents (depending on whether the application is SDI or MDI). The user views and manipulates a document through a view contained inside a frame window. The following figure shows the relationships among these objects for an SDI application.

Objects in a Running SDI Application

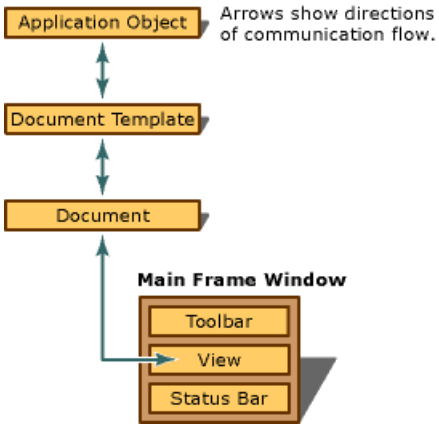

The rest of this family of articles explains how the framework tools, the MFC Application Wizard, and the resource editors, create these objects, how they work together, and how you use them in your programming. Documents, views, and frame windows are discussed in more detail in Window Objects and Document/View Architecture.

See Also

Concepts

Using the Classes to Write Applications for Windows

Wizards and the Resource Editors

Visual C++ includes several wizards for use in MFC programming, along with many integrated resource editors. For ActiveX controls programming, the ActiveX Control Wizard serves a purpose much like that of the MFC Application Wizard. While you can write MFC applications without most of these tools, the tools greatly simplify and speed your work.

Use the MFC Application Wizard to Create an MFC Application

Use the MFC Application Wizard to create an MFC project in Visual C++, which can include OLE and database support. Files in the project contain your application, document, view, and frame-window classes; standard resources, including menus and an optional toolbar; other required Windows files; and optional .rtf files containing standard Windows Help topics that you can revise and augment to create your program's help file.

Use Class View to Manage Classes and Windows Messages

Class View helps you create handler functions for Windows messages and commands, create and manage classes, create class member variables, create Automation methods and properties, create database classes, and more.

Note
Class View also helps you to override virtual functions in the MFC classes. Select the class and the virtual function to override. The rest of the process is similar to message handling, as described in the following paragraphs.

Applications running under Windows are message driven. User actions and other events that occur in the running program cause Windows to send messages to the windows in the program. For example, if the user clicks the mouse in a window, Windows sends a **WM_LBUTTONDOWN** message when the left mouse button is pressed and a **WM_LBUTTONUP** message when the button is released. Windows also sends **WM_COMMAND** messages when the user selects commands from the menu bar.

In the MFC framework, various objects, such as documents, views, frame windows, document templates, and the application object, can "handle" messages. Such an object provides a "handler function" as one of its member functions, and the framework maps the incoming message to its handler.

A large part of your programming task is choosing which messages to map to which objects and then implementing that mapping. To do so, you use Class View and the Properties window.

The Properties window will create empty message-handler member functions, and you use the source code editor to implement the body of the handler. You can also create or edit classes (including classes of your own, not derived from MFC classes) and their members with Class View. For more information on using Class View and about wizards that add code to a project, see Adding Functionality with Code Wizards.

Use the Resource Editors to Create and Edit Resources

Use the Visual C++ resource editors to create and edit menus, dialog boxes, custom controls, accelerator keys, bitmaps, icons, cursors, strings, and version resources. As of Visual C++ version 4.0, a toolbar editor makes creating toolbars much easier.

To help you even more, the Microsoft Foundation Class Library provides a file called COMMON.RES, which contains "clip art" resources that you can copy from COMMON.RES and paste into your own resource file. COMMON.RES includes toolbar buttons, common cursors, icons, and more. You can use, modify, and redistribute these resources in your application. For more information about COMMON.RES, see the Clipart sample.

The MFC Application Wizard, the Visual C++ wizards, resource editors, and the MFC framework do a lot of work for you and make managing your code much easier. The bulk of your application-specific code is in your document and view classes.

See Also

Concepts

Using the Classes to Write Applications for Windows

Building on the Framework

Your role in configuring an application with the MFC framework is to supply the application-specific source code and to connect the components by defining what messages and commands to which they respond. You use the C++ language and standard C++ techniques to derive your own application-specific classes from those supplied by the class library and to override and augment the base class's behavior.

In related topics, the following tables describe the general sequence of operations you will typically follow and your responsibilities versus the framework's responsibilities:

- Sequence for Building an Application with the Framework
- Sequence of Operations for Creating OLE Applications
- Sequence of Operations for Creating ActiveX Controls
- Sequence of Operations for Creating Database Applications

For the most part, you can follow these tables as a sequence of steps for creating an MFC application, although some of the steps are alternative options. For example, most applications use one type of view class from the several types available.

See Also

Concepts

General MFC Topics

Sequence of Operations for Building MFC Applications

The following table explains the general sequence you might typically follow as you develop your MFC application.

Sequence for Building an Application with the Framework

Task	You do	The framework does
Create a skeleton application.	Run the MFC Application Wizard. Specify the options you want in the options pages. Options include making the application a COM component, container, or both; adding Automation; and making the application database-aware.	The MFC Application Wizard creates the files for a skeleton application, including source files for your application, document, view, and frame windows; a resource file; a project file; and others, all tailored to your specifications.
See what the framework and the MFC Application Wizard offer without adding a line of your own code.	Build the skeleton application and run it in Visual C++.	The running skeleton application derives many standard **File**, **Edit**, **View**, and **Help** menu commands from the framework. For MDI applications, you also get a fully functional Windows menu, and the framework manages creation, arrangement, and destruction of MDI child windows.
Construct your application's user interface.	Use the Visual C++ resource editors to visually edit the application's user interface: • Create menus. • Define accelerators. • Create dialog boxes. • Create and edit bitmaps, icons, and cursors. • Edit the toolbar created for you by the MFC Application Wizard. • Create and edit other resources. You can also test the dialog boxes in the dialog editor.	The default resource file created by the MFC Application Wizard supplies many of the resources you need. Visual C++ lets you edit existing resources and add new resources easily and visually.
Map menus to handler functions.	Use the **Events** button in the Properties window to connect menus and accelerators to handler functions in your code.	The Properties window inserts message-map entries and empty function templates into the source files you specify and manages many manual coding tasks.
Write your handler code.	Use Class View to jump directly to the code in the source code editor. Fill in the code for your handler functions. For more information on using Class View and about wizards that add code to a project, see Adding Functionality with Code Wizards.	Class View opens the editor, scrolls to the empty function template and positions the cursor for you.
Map toolbar buttons to commands.	Map each button on your toolbar to a menu or accelerator command by assigning the button the appropriate command ID.	The framework controls the drawing, enabling, disabling, checking, and other visual aspects of the toolbar buttons.
Test your handler functions.	Rebuild the program and use the built-in debugging tools to test that your handlers work correctly.	You can step or trace through the code to see how your handlers are called. If you have filled out the handler code, the handlers carry out commands. The framework will automatically disable menu items and toolbar buttons that are not handled.

Add dialog boxes.	Design dialog-template resources with the dialog editor. Then create a dialog class and the code that handles the dialog box.	The framework manages the dialog box and facilitates retrieving information entered by the user.
Initialize, validate, and retrieve dialog-box data.	You can also define how the dialog box's controls are to be initialized and validated. Use Visual Studio to add member variables to the dialog class and map them to dialog controls. Specify validation rules to be applied to each control as the user enters data. Provide your own custom validations if you wish.	The framework manages dialog-box initialization and validation. If the user enters invalid information, the framework displays a message box and lets the user reenter the data.
Create additional classes.	Use Class View to create additional document, view, and frame-window classes beyond those created automatically by the MFC Application Wizard. You can create additional database recordset classes, dialog classes, and so on. (With Class View, you can create classes not derived from MFC classes.)	Class View adds these classes to your source files and helps you define their connections to any commands they handle.
Add ready-to-use components to your application.	Use the New Item dialog box to add a variety of items.	These items are easy to integrate into your application and save you a great deal of work.
Implement your document class.	Implement your application-specific document class or classes. Add member variables to hold data structures. Add member functions to provide an interface to the data.	The framework already knows how to interact with document data files. It can open and close document files, read and write the document's data, and handle other user interfaces. You can focus on how the document's data is manipulated.
Implement Open, Save, and Save As commands.	Write code for the document's **Serialize** member function.	The framework displays dialog boxes for the **Open**, **Save**, and **Save As** commands on the **File** menu. It writes and reads back a document using the data format specified in your **Serialize** member function.
Implement your view class.	Implement one or more view classes corresponding to your documents. Implement the view's member functions that you mapped to the user interface with Class View. A variety of CView-derived classes are available, including CListView and CTreeView.	The framework manages most of the relationship between a document and its view. The view's member functions access the view's document to render its image on the screen or printed page and to update the document's data structures in response to user editing commands.
Enhance default printing.	If you need to support multipage printing, override view member functions.	The framework supports the **Print**, **Page Setup**, and **Print Preview** commands on the **File** menu. You must tell it how to break your document into multiple pages.
Add scrolling.	If you need to support scrolling, derive your view class or classes from CScrollView.	The view automatically adds scroll bars when the view window becomes too small.
Create form views.	If you want to base your views on dialog-template resources, derive your view class or classes from CFormView.	The view uses the dialog-template resource to display controls. The user can tab from control to control in the view.
Create database forms.	If you want a form-based data-access application, derive your view class from CRecordView (for ODBC programming).	The view works like a form view, but its controls are connected to the fields of a CRecordset object representing a database table. MFC moves data between the controls and the recordset for you.

Create a simple text editor.	If you want your view to be a simple text editor, derive your view class or classes from CEditView or CRichEditView.	The view provides editing functions, Clipboard support, and file input/output. **CRichEditView** provides styled text.
Add splitter windows.	If you want to support window splitting, add a CSplitterWnd object to your SDI frame window or MDI child window and hook it up in the window's OnCreateClient member function.	The framework supplies splitter-box controls next to the scroll bars and manages splitting your view into multiple panes. If the user splits a window, the framework creates and attaches additional view objects to the document.
Build, test, and debug your application.	Use the facilities of Visual C++ to build, test, and debug your application.	Visual C++ lets you adjust compile, link, and other options. It also lets you browse your source code and class structure.

See Also

Concepts

Sequence of Operations for Creating OLE Applications
Sequence of Operations for Creating ActiveX Controls
Sequence of Operations for Creating Database Applications
Building on the Framework

Sequence of Operations for Creating OLE Applications

The following table shows your role and the framework's role in creating OLE linking and embedding applications. These represent options available rather than a sequence of steps to perform.

Creating OLE Applications

Task	You do	The framework does
Create a COM component.	Run the MFC Application Wizard. Choose **Full-server** or **Mini-server** in the **Compound Document Support** tab.	The framework generates a skeleton application with COM component capability enabled. All of the COM capability can be transferred to your existing application with only slight modification.
Create a container application from scratch.	Run the MFC Application Wizard. Choose **Container** in the **Compound Document Support** tab. Using Class View, go to the source code editor. Fill in code for your COM handler functions.	The framework generates a skeleton application that can insert COM objects created by COM component (server) applications.
Create an application that supports Automation from scratch.	Run the MFC Application Wizard. Choose **Automation** from the **Advanced Features** tab. Use Class View to expose methods and properties in your application for automation.	The framework generates a skeleton application that can be activated and automated by other applications.

See Also

Concepts

Building on the Framework
Sequence of Operations for Building MFC Applications
Sequence of Operations for Creating ActiveX Controls
Sequence of Operations for Creating Database Applications

Sequence of Operations for Creating ActiveX Controls

The following table shows your role and the framework's role in creating ActiveX controls (formerly known as OLE controls).

Creating ActiveX Controls

Task	You do	The framework does
Create an ActiveX control framework.	Run the MFC ActiveX Control Wizard to create your control. Specify the options you want in the options pages. Options include the type and name of the control in the project, licensing, subclassing, and an About Box method.	The MFC ActiveX Control Wizard creates the files for an ActiveX control with basic functionality, including source files for your application, control, and property page or pages; a resource file; a project file; and others, all tailored to your specifications.
See what the control and the ActiveX Control Wizard offer without adding a line of your own code.	Build the ActiveX control and test it with Internet Explorer or the TSTCON sample.	The running control has the ability to be resized and moved. It also has an **About Box** method (if chosen) that can be invoked.
Implement the control's methods and properties.	Implement your control-specific methods and properties by adding member functions to provide an exposed interface to the control's data. Add member variables to hold data structures and use event handlers to fire events when you determine.	The framework has already defined a map to support the control's events, properties, and methods, leaving you to focus on how the properties and methods are implemented. The default property page is viewable and a default About Box method is supplied.
Construct the control's property page or pages.	Use the Visual C++ resource editors to visually edit the control's property page interface: • Create additional property pages. • Create and edit bitmaps, icons, and cursors. You can also test the property page(s) in the dialog editor.	The default resource file created by the MFC Application Wizard supplies many of the resources you need. Visual C++ lets you edit existing resources and add new resources easily and visually.
Test the control's events, methods, and properties.	Rebuild the control and use Test Container to test that your handlers work correctly.	You can invoke the control's methods and manipulate its properties through the property page interface or through Test Container. In addition, use Test Container to track events fired from the control and notifications received by the control's container.

See Also

Concepts

Building on the Framework
Sequence of Operations for Building MFC Applications
Sequence of Operations for Creating OLE Applications
Sequence of Operations for Creating Database Applications

Sequence of Operations for Creating Database Applications

The following table shows your role and the framework's role in writing database applications.

Note
As of Visual C++ .NET, the Visual C++ environment and wizards no longer support DAO (although the DAO classes are included and you can still use them). Microsoft recommends that you use ODBC for new MFC projects. You should only use DAO in maintaining existing applications.

Creating Database Applications

Task	You do	The framework does
Decide whether to use the MFC ODBC or DAO classes.	Use ODBC for new MFC projects. Use DAO only to maintain existing applications. See Should I use DAO or ODBC?. For general information, see the article Data Access Programming.	The framework supplies classes that support database access.
Create your skeleton application with database options.	Run the MFC Application Wizard. Select options on the Database Support page. If you choose an option that creates a record view, also specify: • Data source and table name or names • Query name or names.	The MFC Application Wizard creates files and specifies the necessary includes. Depending on the options you specify, the files can include a recordset class.
Design your database form or forms.	Use the Visual C++ dialog editor to place controls on the dialog template resources for your record view classes.	The MFC Application Wizard creates an empty dialog template resource for you to fill in.
Create additional record view and recordset classes as needed.	Use Class View to create the classes and the dialog editor to design the views.	Class View creates additional files for your new classes.
Create recordset objects as needed in your code. Use each recordset to manipulate records...	Your recordsets are based on the classes derived from CRecordset with the wizards.	ODBC uses record field exchange (RFX) to exchange data between the database and your recordset's field data members. If you are using a record view, dialog data exchange (DDX) exchanges data between the recordset and the controls on the record view.
...or create an explicit CDatabase in your code for each database you want to open.	Base your recordset objects on the database objects.	The database object provides an interface to the data source.
Bind data columns to your recordset dynamically.	In ODBC, add code to your derived recordset class to manage the binding. See the article Recordset: Dynamically Binding Data Columns (ODBC).	

See Also

Concepts

Building on the Framework
Sequence of Operations for Building MFC Applications
Sequence of Operations for Creating OLE Applications
Sequence of Operations for Creating ActiveX Controls

How the Framework Calls Your Code

It is crucial to understand the relationship between your source code and the code in the MFC framework. When your application runs, most of the flow of control resides in the framework's code. The framework manages the message loop that gets messages from Windows as the user chooses commands and edits data in a view. Events that the framework can handle by itself do not rely on your code at all. For example, the framework knows how to close windows and how to exit the application in response to user commands. As it handles these tasks, the framework uses message handlers and C++ virtual functions to give you opportunities to respond to these events as well. Your code is not in control, however; the framework is.

The framework calls your code for application-specific events. For example, when the user chooses a menu command, the framework routes the command along a sequence of C++ objects: the current view and frame window, the document associated with the view, the document's document template, and the application object. If one of these objects can handle the command, it does so, calling the appropriate message-handler function. For any given command, the code called may be yours or it may be the framework's.

This arrangement is somewhat familiar to programmers experienced with traditional programming for Windows or event-driven programming.

In related topics, you will read what the framework does as it initializes and runs the application and then cleans up as the application terminates. You will also understand where the code you write fits in.

For more information, see Class CWinApp: The Application Class and Document Templates and the Document/View Creation Process.

See Also

Concepts

Building on the Framework

CWinApp: The Application Class

The main application class in MFC encapsulates the initialization, running, and termination of an application for the Windows operating system. An application built on the framework must have one and only one object of a class derived from CWinApp. This object is constructed before windows are created.

CWinApp is derived from **CWinThread**, which represents the main thread of execution for your application, which might have one or more threads. In recent versions of MFC, the **InitInstance**, **Run**, **ExitInstance**, and **OnIdle** member functions are actually in class **CWinThread**. These functions are discussed here as if they were **CWinApp** members instead, because the discussion concerns the object's role as application object rather than as primary thread.

Note
Your application class constitutes your application's primary thread of execution. Using Win32 API functions, you can also create secondary threads of execution. These threads can use the MFC Library. For more information, see Multithreading.

Like any program for the Windows operating system, your framework application has a **WinMain** function. In a framework application, however, you do not write **WinMain**. It is supplied by the class library and is called when the application starts up. **WinMain** performs standard services such as registering window classes. It then calls member functions of the application object to initialize and run the application. (You can customize **WinMain** by overriding the **CWinApp** member functions that **WinMain** calls.)

To initialize the application, **WinMain** calls your application object's `InitApplication` and `InitInstance` member functions. To run the application's message loop, **WinMain** calls the **Run** member function. On termination, **WinMain** calls the application object's `ExitInstance` member function.

Note
Names shown in **bold** in this documentation indicate elements supplied by the Microsoft Foundation Class Library and Visual C++. Names shown in `monospaced` type indicate elements that you create or override.

See Also

Concepts

General MFC Topics
CWinApp and the MFC Application Wizard
Overridable CWinApp Member Functions
Special CWinApp Services

CWinApp and the MFC Application Wizard

When it creates a skeleton application, the MFC Application Wizard declares an application class derived from CWinApp. The MFC Application Wizard also generates an implementation file that contains the following items:

- A message map for the application class.
- An empty class constructor.
- A variable that declares the one and only object of the class.
- A standard implementation of your `InitInstance` member function.

The application class is placed in the project header and main source files. The names of the class and files created are based on the project name you supply in the MFC Application Wizard. The easiest way to view the code for these classes is through Class View.

The standard implementations and message map supplied are adequate for many purposes, but you can modify them as needed. The most interesting of these implementations is the `InitInstance` member function. Typically, you will add code to the skeletal implementation of `InitInstance`.

See Also

Concepts

CWinApp: The Application Class
Overridable CWinApp Member Functions
Special CWinApp Services

Overridable CWinApp Member Functions

CWinApp provides several key overridable member functions (**CWinApp** overrides these members from class CWinThread, from which **CWinApp** derives):

- InitInstance
- Run
- ExitInstance
- OnIdle

The only **CWinApp** member function that you must override is **InitInstance**.

See Also

Concepts

CWinApp: The Application Class

InitInstance Member Function

The Windows operating system allows you to run more than one copy, or "instance," of the same application. **WinMain** calls InitInstance every time a new instance of the application starts.

The standard `InitInstance` implementation created by the MFC Application Wizard performs the following tasks:

- As its central action, creates the document templates that in turn create documents, views, and frame windows. For a description of this process, see Document Template Creation.
- Loads standard file options from an .ini file or the Windows registry, including the names of the most recently used files.
- Registers one or more document templates.
- For an MDI application, creates a main frame window.
- Processes the command line to open a document specified on the command line or to open a new, empty document.

You can add your own initialization code or modify the code written by the wizard.

Note
MFC applications must be initialized as single threaded apartment (STA). If you call CoInitializeEx in your **InitInstance** override, specify **COINIT_APARTMENTTHREADED** (rather than **COINIT_MULTITHREADED**). For more information, see PRB: MFC Application Stops Responding When You Initialize the Application as a Multithreaded Apartment (828643) at http://support.microsoft.com/default.aspx?scid=kb;en-us;828643.

See Also

Concepts

CWinApp: The Application Class

Run Member Function

A framework application spends most of its time in the Run member function of class CWinApp. After initialization, **WinMain** calls **Run** to process the message loop.

Run cycles through a message loop, checking the message queue for available messages. If a message is available, **Run** dispatches it for action. If no messages are available, which is often true, **Run** calls **OnIdle** to do any idle-time processing that you or the framework may need done. If there are no messages and no idle processing to do, the application waits until something happens. When the application terminates, **Run** calls **ExitInstance**. The figure in OnIdle Member Function shows the sequence of actions in the message loop.

Message dispatching depends on the kind of message. For more information, see Messages and Commands in the Framework.

See Also

Concepts

CWinApp: The Application Class

ExitInstance Member Function

The ExitInstance member function of class CWinApp is called each time a copy of your application terminates, usually as a result of the user quitting the application.

Override **ExitInstance** if you need special cleanup processing, such as freeing graphics device interface (GDI) resources or deallocating memory used during program execution. Cleanup of standard items such as documents and views, however, is provided by the framework, with other overridable functions for doing special cleanup specific to those objects.

See Also

Concepts

CWinApp: The Application Class

OnIdle Member Function

When no Windows messages are being processed, the framework calls the CWinApp member function OnIdle (described in the MFC Library Reference).

Override **OnIdle** to perform background tasks. The default version updates the state of user-interface objects such as toolbar buttons and performs cleanup of temporary objects created by the framework in the course of its operations. The following figure illustrates how the message loop calls **OnIdle** when there are no messages in the queue.

The Message Loop

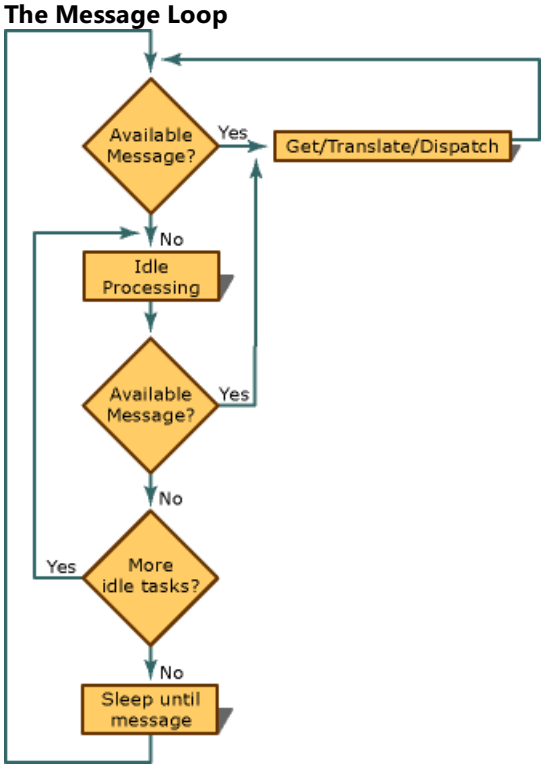

For more information about what you can do in the idle loop, see Idle Loop Processing.

See Also

Concepts

CWinApp: The Application Class

Special CWinApp Services

Besides running the message loop and giving you an opportunity to initialize the application and clean up after it, CWinApp provides several other services.

Shell Registration

By default, the MFC Application Wizard makes it possible for the user to open data files that your application has created by double-clicking them in Windows Explorer or File Manager. If your application is an MDI application and you specify an extension for the files your application creates, the MFC Application Wizard adds calls to the RegisterShellFileTypes and EnableShellOpen member functions of CWinApp to the `InitInstance` override that it writes for you.

RegisterShellFileTypes registers your application's document types with Windows Explorer or File Manager. The function adds entries to the registration database that Windows maintains. The entries register each document type, associate a file extension with the file type, specify a command line to open the application, and specify a dynamic data exchange (DDE) command to open a document of that type.

EnableShellOpen completes the process by allowing your application to receive DDE commands from Windows Explorer or File Manager to open the file chosen by the user.

This automatic registration support in **CWinApp** eliminates the need to ship a .reg file with your application or to do special installation work.

If you want to initialize GDI+ for your application (by calling GdiplusStartup in your InitInstance function), you will need to suppress the GDI+ background thread.

You can do this by setting the **SuppressBackgroundThread** member of the GdiplusStartupInput structure to **TRUE**. When suppressing the GDI+ background thread, the **NotificationHook** and **NotificationUnhook** calls (see GdiplusStartupOutput) should be made just prior to entering and exiting the application's message loop. Therefore, a good place to call **GdiplusStartup** and the hook notification functions would be in an override of the virtual function CWinApp::Run, as shown below:

```
int CTestMDIAppApp::Run()
{
GdiplusStartupInput gdiSI;
GdiplusStartupOutput gdiSO;
ULONG_PTR gdiToken;
ULONG_PTR gdiHookToken;

gdiSI.SuppressBackgroundThread = TRUE;
GdiplusStartup(&gdiToken,&gdiSI,&gdiSO);
gdiSO.NotificationHook(&gdiHookToken);
int nRet = CWinApp::Run();

gdiSO.NotificationUnhook(gdiHookToken);
GdiplusShutdown(gdiToken);

return nRet;
}
```

If you do not suppress the background GDI+ thread, DDE commands can be prematurely issued to the application before its main window has been created. The DDE commands issued by the shell can be prematurely aborted, resulting in error messages.

File Manager Drag and Drop

Windows versions 3.1 and later allow the user to drag file names from the file view window in File Manager (or Windows Explorer in Windows 95 and later and in Windows NT 4.0 and later) and drop them into a window in your application. You might, for example, allow the user to drag one or more filenames into an MDI application's main window, where the application could retrieve the file names and open MDI child windows for those files.

To enable file drag and drop in your application, the MFC Application Wizard writes a call to the CWnd member function DragAcceptFiles for your main frame window in your `InitInstance`. You can remove that call if you do not want to implement the drag-and-drop feature.

Note
You can also implement more general drag-and-drop capabilities — dragging data between or within documents — with OLE. For information, see the article Drag and Drop (OLE).

Keeping Track of the Most Recently Used Documents

As the user opens and closes files, the application object keeps track of the four most recently used files. The names of these files are added to the File menu and updated when they change. The framework stores these file names in either the registry or in the .ini file, with the same name as your project and reads them from the file when your application starts up. The `InitInstance` override that the MFC Application Wizard creates for you includes a call to the CWinApp member function LoadStdProfileSettings, which loads information from the registry or .ini file, including the most recently used file names.

These entries are stored as follows:

- In Windows NT, Windows 2000, and later, the value is stored to a registry key.
- In Windows 3.x, the value is stored in the WIN.INI file.
- In Windows 95 and later, the value is stored in a cached version of WIN.INI.

See Also

Concepts

CWinApp: The Application Class

Document Templates and the Document/View Creation Process

To manage the complex process of creating documents with their associated views and frame windows, the framework uses two document template classes: CSingleDocTemplate for SDI applications and CMultiDocTemplate for MDI applications. A **CSingleDocTemplate** can create and store one document of one type at a time. A **CMultiDocTemplate** keeps a list of many open documents of one type.

Some applications support multiple document types. For example, an application might support text documents and graphics documents. In such an application, when the user chooses the New command on the File menu, a dialog box shows a list of possible new document types to open. For each supported document type, the application uses a distinct document template object. The following figure illustrates the configuration of an MDI application that supports two document types and shows several open documents.

An MDI Application with Two Document Types

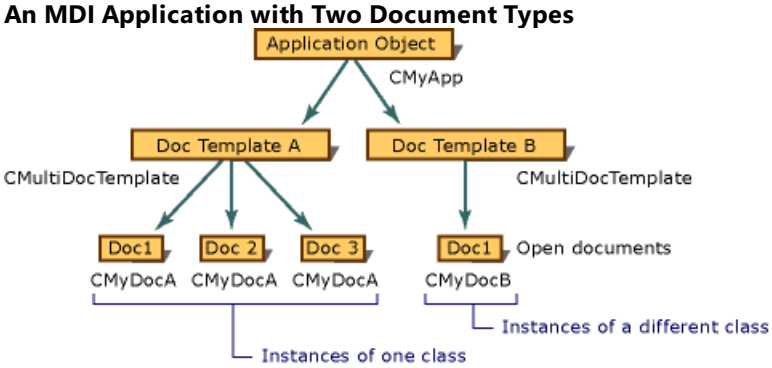

Document templates are created and maintained by the application object. One of the key tasks performed during your application's `InitInstance` function is to construct one or more document templates of the appropriate kind. This feature is described in Document Template Creation. The application object stores a pointer to each document template in its template list and provides an interface for adding document templates.

If you need to support two or more document types, you must add an extra call to AddDocTemplate for each document type.

An icon is registered for each document template based on its position in the application's list of document templates. The order of the document templates is determined by the order they are added with calls to **AddDocTemplate**. MFC assumes that the first Icon resource in the application is the application icon, the next Icon resource is the first document icon, and so on.

For example, a document template is the third of three for the application. If there is an Icon resource in the application at index 3, that icon is used for the document template. If not, the icon at index 0 is used as a default.

See Also

Concepts

General MFC Topics
Document Template Creation
Document/View Creation
Relationships Among MFC Objects
Creating New Documents, Windows, and Views

Document Template Creation

When creating a new document in response to a **New** or **Open** command from the **File** menu, the document template also creates a new frame window through which to view the document.

The document-template constructor specifies what types of documents, windows, and views the template will be able to create. This is determined by the arguments you pass to the document-template constructor. The following code illustrates creation of a CMultiDocTemplate for a sample application:

```
AddDocTemplate( new CMultiDocTemplate( IDR_SCRIBTYPE,
          RUNTIME_CLASS( CScribDoc ),
          RUNTIME_CLASS( CMDIChildWnd ),
          RUNTIME_CLASS( CScribView ) ) );
```

The pointer to a new **CMultiDocTemplate** object is used as an argument to AddDocTemplate. Arguments to the **CMultiDocTemplate** constructor include the resource ID associated with the document type's menus and accelerators, and three uses of the RUNTIME_CLASS macro. **RUNTIME_CLASS** returns the CRuntimeClass object for the C++ class named as its argument. The three **CRuntimeClass** objects passed to the document-template constructor supply the information needed to create new objects of the specified classes during the document creation process. The example shows creation of a document template that creates `CScribDoc` objects with `CScribView` objects attached. The views are framed by standard MDI child frame windows.

See Also

Concepts

Document Templates and the Document/View Creation Process
Document/View Creation
Relationships Among MFC Objects
Creating New Documents, Windows, and Views

Document/View Creation

The framework supplies implementations of the **New** and **Open** commands (among others) on the **File** menu. Creation of a new document and its associated view and frame window is a cooperative effort among the application object, a document template, the newly created document, and the newly created frame window. The following table summarizes which objects create what.

Object Creators

Creator	Creates
Application object	Document template
Document template	Document
Document template	Frame window
Frame window	View

See Also

Concepts

Document Templates and the Document/View Creation Process
Document Template Creation
Relationships Among MFC Objects
Creating New Documents, Windows, and Views

Relationships Among MFC Objects

To help put the document/view creation process in perspective, consider a running program: a document, the frame window used to contain the view, and the view associated with the document.

- A document keeps a list of the views of that document and a pointer to the document template that created the document.
- A view keeps a pointer to its document and is a child of its parent frame window.
- A document frame window keeps a pointer to its current active view.
- A document template keeps a list of its open documents.
- The application keeps a list of its document templates.
- Windows keeps track of all open windows so it can send messages to them.

These relationships are established during document/view creation. The following table shows how objects in a running program can access other objects. Any object can obtain a pointer to the application object by calling the global function AfxGetApp.

Gaining Access to Other Objects in Your Application

From object	How to access other objects
Document	Use GetFirstViewPosition and GetNextView to access the document's view list. Call GetDocTemplate to get the document template.
View	Call GetDocument to get the document. Call GetParentFrame to get the frame window.
Document frame window	Call GetActiveView to get the current view. Call GetActiveDocument to get the document attached to the current view.
MDI frame window	Call MDIGetActive to get the currently active CMDIChildWnd.

Typically, a frame window has one view, but sometimes, as in splitter windows, the same frame window contains multiple views. The frame window keeps a pointer to the currently active view; the pointer is updated any time another view is activated.

Note

A pointer to the main frame window is stored in the m_pMainWnd member variable of the application object. A call to **OnFileNew** in your override of the **InitInstance** member function of **CWinApp** sets **m_pMainWnd** for you. If you do not call **OnFileNew**, you must set the variable's value in **InitInstance** yourself. (SDI COM component (server) applications may not set the variable if /Embedding is on the command line.) Note that **m_pMainWnd** is now a member of class **CWinThread** rather than **CWinApp**.

See Also

Concepts

Document Templates and the Document/View Creation Process
Document Template Creation
Document/View Creation
Creating New Documents, Windows, and Views

Creating New Documents, Windows, and Views

The following figures give an overview of the creation process for documents, views, and frame windows. Other articles that focus on the participating objects provide further details.

Upon completion of this process, the cooperating objects exist and store pointers to each other. The following figures show the sequence in which objects are created. You can follow the sequence from figure to figure.

Sequence in Creating a Document

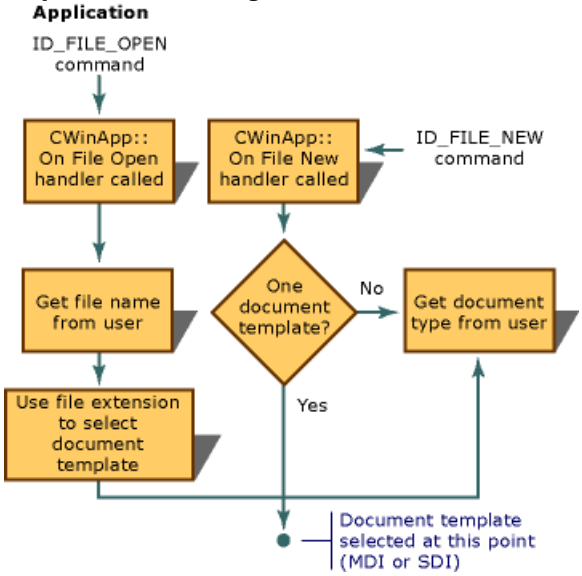

Sequence in Creating a Frame Window

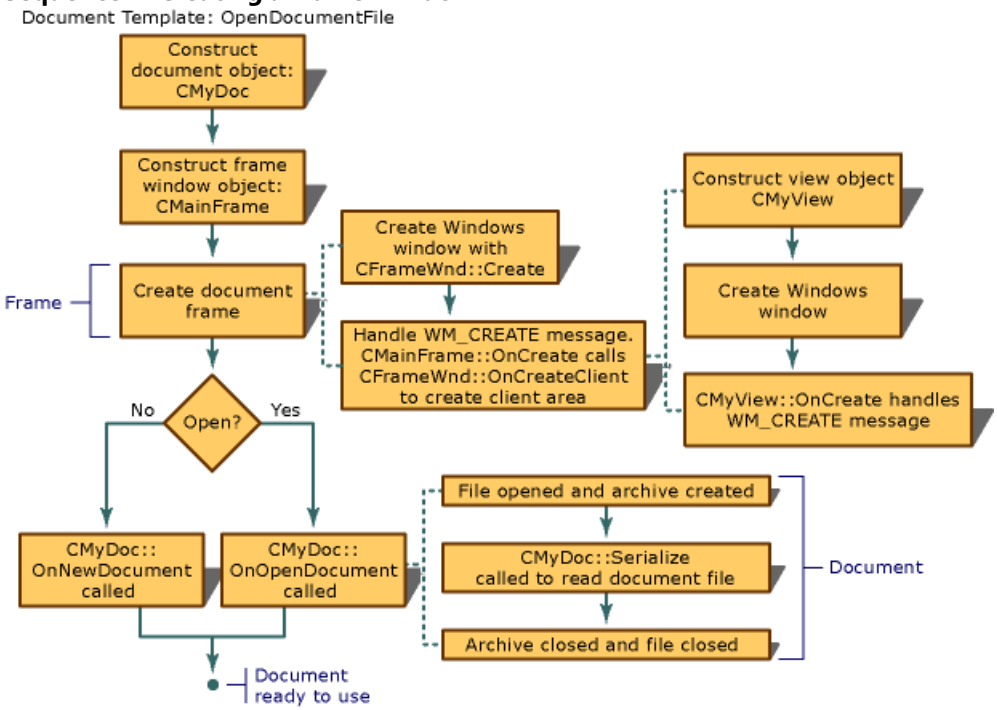

Sequence in Creating a View

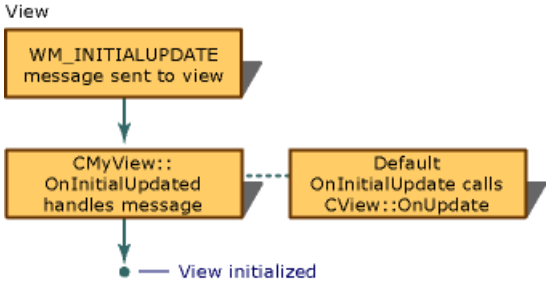

For information about how the framework initializes the new document, view, and frame-window objects, see classes CDocument, CView, CFrameWnd, CMDIFrameWnd, and CMDIChildWnd in the MFC Library Reference. Also see Technical Note 22, which explains the creation and initialization processes further under its discussion of the framework's

standard commands for the **New** and **Open** items on the **File** menu.

Initializing Your Own Additions to These Classes

The preceding figures also suggest the points at which you can override member functions to initialize your application's objects. An override of **OnInitialUpdate** in your view class is the best place to initialize the view. The **OnInitialUpdate** call occurs immediately after the frame window is created and the view within the frame window is attached to its document. For example, if your view is a scroll view (derived from **CScrollView** rather than **CView**), you should set the view size based on the document size in your `OnInitialUpdate` override. (This process is described in the description of class CScrollView.) You can override the **CDocument** member functions **OnNewDocument** and **OnOpenDocument** to provide application-specific initialization of the document. Typically, you must override both since a document can be created in two ways.

In most cases, your override should call the base class version. For more information, see the named member functions of classes CDocument, CView, CFrameWnd, and CWinApp in the MFC Library Reference.

See Also

Concepts

Document Templates and the Document/View Creation Process
Document Template Creation
Document/View Creation
Relationships Among MFC Objects

Managing the State Data of MFC Modules

This article discusses the state data of MFC modules and how this state is updated when the flow of execution (the path code takes through an application when executing) enters and leaves a module. Switching module states with the **AFX_MANAGE_STATE** and **METHOD_PROLOGUE** macros is also discussed.

Note
The term "module" here refers to an executable program, or to a DLL (or set of DLLs) that operate independently of the rest o f the application, but uses a shared copy of the MFC DLL. An ActiveX control is a typical example of a module.

As shown in the following figure, MFC has state data for each module used in an application. Examples of this data include Windows instance handles (used for loading resources), pointers to the current **CWinApp** and **CWinThread** objects of an application, OLE module reference counts, and a variety of maps that maintain the connections between Windows object handles and corresponding instances of MFC objects. However, when an application uses multiple modules, the state data of each module is not application wide. Rather, each module has its own private copy of the MFC's state data.

State Data of a Single Module (Application)

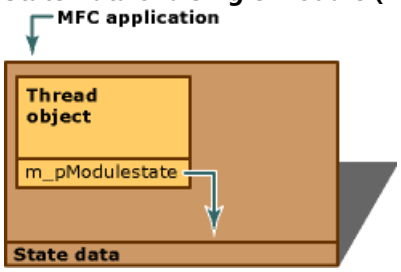

A module's state data is contained in a structure and is always available via a pointer to that structure. When the flow of execution enters a particular module, as shown in the following figure, that module's state must be the "current" or "effective" state. Therefore, each thread object has a pointer to the effective state structure of that application. Keeping this pointer updated at all times is vital to managing the application's global state and maintaining the integrity of each module's state. Incorrect management of the global state can lead to unpredictable application behavior.

State Data of Multiple Modules

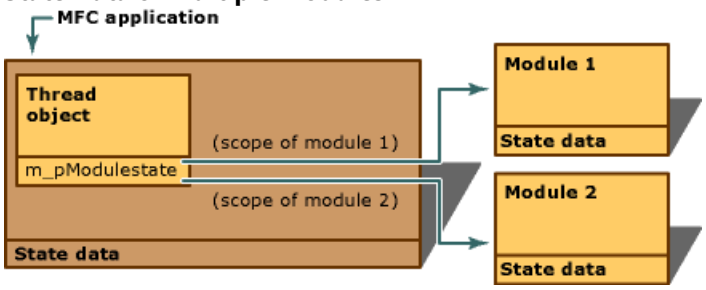

In other words, each module is responsible for correctly switching between module states at all of its entry points. An "entry point" is any place where the flow of execution can enter the module's code. Entry points include:

- Exported functions in a DLL
- Member functions of COM interfaces
- Window procedures

See Also

Concepts

General MFC Topics

Exported DLL Function Entry Points

For exported functions of a DLL, use the AFX_MANAGE_STATE macro to maintain the proper global state when switching from the DLL module to the calling application's DLL.

When called, this macro sets *pModuleState*, a pointer to an **AFX_MODULE_STATE** structure containing global data for the module, as the effective module state for the remainder of the containing scope of the function. Upon leaving the scope containing the macro, the previous effective module state is automatically restored.

This switching is achieved by constructing an instance of an **AFX_MODULE_STATE** class on the stack. In its constructor, this class obtains a pointer to the current module state and stores it in a member variable, and then sets *pModuleState* as the new effective module state. In its destructor, this class restores the pointer stored in its member variable as the effective module state.

If you have an exported function, such as one that launches a dialog box in your DLL, you need to add the following code to the beginning of the function:

```
AFX_MANAGE_STATE(AfxGetStaticModuleState( ))
```

This swaps the current module state with the state returned from AfxGetStaticModuleState until the end of the current scope.

Problems with resources in DLLs will occur if the **AFX_MANAGE_STATE** macro is not used. By default, MFC uses the resource handle of the main application to load the resource template. This template is actually stored in the DLL. The root cause is that MFC's module state information has not been switched by the **AFX_MANAGE_STATE** macro. The resource handle is recovered from MFC's module state. Not switching the module state causes the wrong resource handle to be used.

AFX_MANAGE_STATE does not need to be put into every function in the DLL. For example, `InitInstance` can be called by the MFC code in the application without **AFX_MANAGE_STATE** because MFC automatically shifts the module state before `InitInstance` and then switches it back after `InitInstance` returns. The same is true for all message-map handlers. Regular DLLs actually have a special master window procedure that automatically switches the module state before routing any message.

See Also

Concepts

Managing the State Data of MFC Modules

COM Interface Entry Points

For member functions of a COM interface, use the METHOD_PROLOGUE macro to maintain the proper global state when calling methods of an exported interface.

Typically, member functions of interfaces implemented by **CCmdTarget**-derived objects already use this macro to provide automatic initialization of the *pThis* pointer. For example:

```
// Inner IUnknown implementation (for aggregation)

STDMETHODIMP_(ULONG) CInnerUnknown::AddRef()
{
   METHOD_PROLOGUE_(CCmdTarget, InnerUnknown)
   return pThis->InternalAddRef();
}
```

For additional information, see Technical Note 38 on MFC/OLE **IUnknown** implementation.

The **METHOD_PROLOGUE** macro is defined as:

```
#define METHOD_PROLOGUE(theClass, localClass) \
   theClass* pThis = \
      ((theClass*)((BYTE*)this - offsetof(theClass, m_x##localClass))); \
   AFX_MANAGE_STATE(pThis->m_pModuleState) \
```

The portion of the macro concerned with managing the global state is:

```
AFX_MANAGE_STATE( pThis->m_pModuleState )
```

In this expression, *m_pModuleState* is assumed to be a member variable of the containing object. It is implemented by the **CCmdTarget** base class and is initialized to the appropriate value by **COleObjectFactory**, when the object is instantiated.

See Also

Concepts

Managing the State Data of MFC Modules

Window Procedure Entry Points

To protect MFC window procedures, a module static links with a special window procedure implementation. The linkage occurs automatically when the module is linked with MFC. This window procedure uses the **AFX_MANAGE_STATE** macro to properly set the effective module state, then it calls **AfxWndProc**, which in turn delegates to the **WindowProc** member function of the appropriate **CWnd**-derived object.

See Also

Concepts

Managing the State Data of MFC Modules

Idle Loop Processing

Many applications perform lengthy processing "in the background." Sometimes performance considerations dictate using multithreading for such work. Threads involve extra development overhead, so they are not recommended for simple tasks like the idle-time work that MFC does in the OnIdle function. This article focuses on idle processing. For more information about multithreading, see Multithreading Topics.

Some kinds of background processing are appropriately done during intervals that the user is not otherwise interacting with the application. In an application developed for the Microsoft Windows operating system, an application can perform idle-time processing by splitting a lengthy process into many small fragments. After processing each fragment, the application yields execution control to Windows using a PeekMessage loop.

This article explains two ways to do idle processing in your application:

- Using **PeekMessage** in MFC's main message loop.
- Embedding another **PeekMessage** loop somewhere else in the application.

PeekMessage in the MFC Message Loop

In an application developed with MFC, the main message loop in the **CWinThread** class contains a message loop that calls the PeekMessage Win32 API. This loop also calls the **OnIdle** member function of **CWinThread** between messages. An application can process messages in this idle time by overriding the **OnIdle** function.

Note
Run, **OnIdle**, and certain other member functions are now members of class **CWinThread** rather than of class **CWinApp**. **CWinApp** is derived from **CWinThread**.

For more information about performing idle processing, see OnIdle in the *MFC Reference*.

PeekMessage Elsewhere in Your Application

Another method for performing idle processing in an application involves embedding a message loop in one of your functions. This message loop is very similar to MFC's main message loop, found in CWinThread::Run. That means such a loop in an application developed with MFC must perform many of the same functions as the main message loop. The following code fragment demonstrates writing a message loop that is compatible with MFC:

```
while ( bDoingBackgroundProcessing )
{
    MSG msg;
    while ( ::PeekMessage( &msg, NULL, 0, 0, PM_NOREMOVE ) )
    {
        if ( !PumpMessage( ) )
        {
            bDoingBackgroundProcessing = FALSE;
            ::PostQuitMessage( );
            break;
        }
    }
    // let MFC do its idle processing
    LONG lIdle = 0;
    while ( AfxGetApp()->OnIdle(lIdle++ ) )
        ;
    // Perform some background processing here
    // using another call to OnIdle
}
```

This code, embedded in a function, loops as long as there is idle processing to do. Within that loop, a nested loop repeatedly calls **PeekMessage**. As long as that call returns a nonzero value, the loop calls **CWinThread::PumpMessage** to perform normal message translation and dispatching. Although **PumpMessage** is undocumented, you can examine its source code in the ThrdCore.Cpp file in the \atlmfc\src\mfc directory of your Visual C++ installation.

Once the inner loop ends, the outer loop performs idle processing with one or more calls to **OnIdle**. The first call is for MFC's

purposes. You can make additional calls to **OnIdle** to do your own background work.

For more information about performing idle processing, see OnIdle in the MFC Library Reference.

See Also

Concepts

General MFC Topics

Support for Activation Contexts in the MFC Module State

MFC creates an activation context using a manifest resource provided by the user module. For more information on how activation contexts are created, see the following topics:

- Activation Contexts
- Application Manifests
- Assembly Manifests

Remarks

When reading these Platform SDK topics, note that the MFC activation context mechanism resembles the Platform SDK activation context except that MFC does not use the Platform SDK Activation Context API.

Activation context works in MFC applications, user DLLs, and extension DLLs in the following ways:

- MFC applications use resource ID 1 for their manifest resource. In this case, the MFC does not create its own activation context, but uses the default application context.
- MFC user DLLs use resource ID 2 for their manifest resource. Here, MFC creates an activation context for each User DLL, so different user DLLs can use different versions of the same libraries (for example, the Common Controls library).
- MFC extension DLLs rely on their hosting applications or user DLLs to establish their activation context.

Although the activation context state can be modified using the processes described under Using the Activation Context API, using the MFC activation context mechanism can be useful when developing DLL-based plug-in architectures where it is not easy (or not possible) to manually switch activation state before and after individual calls to external plug-ins.

The activation context is created in AfxWinInit. It is destroyed in the **AFX_MODULE_STATE** destructor. An activation context handle is kept in **AFX_MODULE_STATE**. (**AFX_MODULE_STATE** is described in AfxGetStaticModuleState.)

The AFX_MANAGE_STATE macro activates and deactivates the activation context. As of Visual C++ 2005, the behavior of **AFX_MANAGE_STATE** has changed in that it is enabled for static MFC libraries, as well as MFC DLLs, to allow MFC code to execute in the proper activation context selected by the User DLL.

See Also

Reference

Activation Contexts
Application Manifests
Assembly Manifests
AfxWinInit
AfxGetStaticModuleState
AFX_MANAGE_STATE

Isolation of the MFC Common Controls Library

The Common Controls library is now isolated within MFC, allowing different modules (such as user DLLs) to use different versions of the Common Controls library by specifying the version in their manifests.

An MFC application (or user code called by MFC) makes calls to Common Controls library APIs through wrapper functions named **Afx***FunctionName*, where *FunctionName* is the name of a Common Controls API. Those wrapper functions are defined in afxcomctl32.h and afxcomctl32.inl.

You can use the AFX_COMCTL32_IF_EXISTS and AFX_COMCTL32_IF_EXISTS2 macros (defined in afxcomctl32.h) to determine whether the Common Controls library implements a certain API instead of calling GetProcAddress.

Technically, you make calls to Common Controls Library APIs through a wrapper class, **CComCtlWrapper** (defined in afxcomctl32.h). **CComCtlWrapper** is also responsible for the loading and unloading of comctl32.dll. The MFC Module State contains a pointer to an instance of **CComCtlWrapper**. You can access the wrapper class using the **afxComCtlWrapper** macro.

Note that calling Common Controls API directly (not using the MFC wrapper functions) from an MFC application or user DLL will work in most cases, because the MFC application or user DLL is bound to the Common Controls library it requested in its manifest). However, the MFC code itself has to use the wrappers, because MFC code might be called from user DLLs with different Common Controls library versions.

Using CObject

CObject is the root base class for most of the Microsoft Foundation Class Library (MFC). The **CObject** class contains many useful features that you may want to incorporate into your own program objects, including serialization support, run-time class information, and object diagnostic output. If you derive your class from **CObject**, your class can exploit these **CObject** features.

What do you want to do?

- Derive a class from CObject
- Add support for run-time class information, dynamic creation, and serialization to my derived class
- Access run-time class information
- Create objects dynamically
- Dump the object's data for diagnostic purposes
- Validate the object's internal state (see MFC ASSERT_VALID and CObject::AssertValid)
- Have the class serialize itself to persistent storage
- See a list of CObject Frequently Asked Questions

See Also

Reference

CRuntimeClass Structure

Concepts

General MFC Topics

Files in MFC

Serialization in MFC

Other Resources

MFC Concepts

Deriving a Class from CObject

This article describes the minimum steps necessary to derive a class from CObject. Other **CObject** class articles describe the steps needed to take advantage of specific **CObject** features, such as serialization and diagnostic debugging support.

In the discussions of **CObject**, the terms "interface file" and "implementation file" are used frequently. The interface file (often called the header file, or .H file) contains the class declaration and any other information needed to use the class. The implementation file (or .CPP file) contains the class definition as well as the code that implements the class member functions. For example, for a class named `CPerson`, you would typically create an interface file named PERSON.H and an implementation file named PERSON.CPP. However, for some small classes that will not be shared among applications, it is sometimes easier to combine the interface and implementation into a single .CPP file.

You can choose from four levels of functionality when deriving a class from **CObject**:

- Basic functionality: No support for run-time class information or serialization but includes diagnostic memory management.
- Basic functionality plus support for run-time class information.
- Basic functionality plus support for run-time class information and dynamic creation.
- Basic functionality plus support for run-time class information, dynamic creation, and serialization.

Classes designed for reuse (those that will later serve as base classes) should at least include run-time class support and serialization support, if any future serialization need is anticipated.

You choose the level of functionality by using specific declaration and implementation macros in the declaration and implementation of the classes you derive from **CObject**.

The following table shows the relationship among the macros used to support serialization and run-time information.

Macros Used for Serialization and Run-Time Information

Macro used	CObject::IsKindOf	CRuntimeClass:: CreateObject	CArchive::operator>> CArchive::operator<<
Basic **CObject** functionality	No	No	No
DECLARE_DYNAMIC	Yes	No	No
DECLARE_DYNCREATE	Yes	Yes	No
DECLARE_SERIAL	Yes	Yes	Yes

To use basic CObject functionality

- Use the normal C++ syntax to derive your class from **CObject** (or from a class derived from **CObject**).

  The following example shows the simplest case, the derivation of a class from **CObject**:

```
class CPerson : public CObject
{
    // add CPerson-specific members and functions...
}
```

Normally, however, you may want to override some of **CObject**'s member functions to handle the specifics of your new class. For example, you may usually want to override the **Dump** function of **CObject** to provide debugging output for the contents of your class. For details on how to override **Dump**, see the article Diagnostics: Dumping Object Contents. You may also want to override the **AssertValid** function of **CObject** to provide customized testing to validate the consistency of the data members of class objects. For a description of how to override **AssertValid**, see MFC ASSERT_VALID and CObject::AssertValid.

The article Specifying Levels of Functionality describes how to specify other levels of functionality, including run-time class

information, dynamic object creation, and serialization.

See Also

Concepts

Using CObject

Specifying Levels of Functionality

This article describes how to add the following levels of functionality to your CObject-derived class:

- Run-time class information
- Dynamic creation support
- Serialization support

For a general description of **CObject** functionality, see the article Deriving a Class from CObject.

To add run-time class information

1. Derive your class from **CObject**, as described in the Deriving a Class from CObject article.
2. Use the **DECLARE_DYNAMIC** macro in your class declaration, as shown here:

```
class CPerson : public CObject
{
    DECLARE_DYNAMIC( CPerson )

    // rest of class declaration follows...
};
```

3. Use the **IMPLEMENT_DYNAMIC** macro in the implementation file (.CPP) of your class. This macro takes as arguments the name of the class and its base class, as follows:

```
IMPLEMENT_DYNAMIC( CPerson, CObject )
```

Note

Always put **IMPLEMENT_DYNAMIC** in the implementation file (.CPP) for your class. The **IMPLEMENT_DYNAMIC** macro sh ould be evaluated only once during a compilation and therefore should not be used in an interface file (.H) that could potenti ally be included in more than one file.

To add dynamic creation support

1. Derive your class from **CObject**.
2. Use the **DECLARE_DYNCREATE** macro in the class declaration.
3. Define a constructor with no arguments (a default constructor).
4. Use the **IMPLEMENT_DYNCREATE** macro in the class implementation file.

To add serialization support

1. Derive your class from **CObject**.
2. Override the **Serialize** member function.

   Note

   If you call **Serialize** directly, that is, you do not want to serialize the object through a polymorphic pointer, omit steps 3 through 5.

3. Use the **DECLARE_SERIAL** macro in the class declaration.
4. Define a constructor with no arguments (a default constructor).

5. Use the **IMPLEMENT_SERIAL** macro in the class implementation file.

Note
A "polymorphic pointer" points to an object of a class (call it A) or to an object of any class derived from A (say, B). To serialize through a polymorphic pointer, the framework must determine the run-time class of the object it is serializing (B), since it might be an object of any class derived from some base class (A).

For more details on how to enable serialization when you derive your class from **CObject**, see the articles Files in MFC and Serialization.

See Also

Concepts

Deriving a Class from CObject

Accessing Run-Time Class Information

This article explains how to access information about the class of an object at run time.

Note
MFC does not use the Run-Time Type Information (RTTI) support introduced in Visual C++ 4.0.

If you have derived your class from CObject and used the **DECLARE_DYNAMIC** and **IMPLEMENT_DYNAMIC**, the **DECLARE_DYNCREATE** and **IMPLEMENT_DYNCREATE**, or the **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** macros explained in the article Deriving a Class from CObject, the **CObject** class has the ability to determine the exact class of an object at run time.

This ability is most useful when extra type checking of function arguments is needed and when you must write special-purpose code based on the class of an object. However, this practice is not usually recommended because it duplicates the functionality of virtual functions.

The **CObject** member function **IsKindOf** can be used to determine if a particular object belongs to a specified class or if it is derived from a specific class. The argument to **IsKindOf** is a **CRuntimeClass** object, which you can get using the **RUNTIME_CLASS** macro with the name of the class.

To use the RUNTIME_CLASS macro

- Use **RUNTIME_CLASS** with the name of the class, as shown here for the class **CObject**:

```
CRuntimeClass* pClass = RUNTIME_CLASS( CObject );
```

You will rarely need to access the run-time class object directly. A more common use is to pass the run-time class object to the **IsKindOf** function, as shown in the next procedure. The **IsKindOf** function tests an object to see if it belongs to a particular class.

To use the IsKindOf function

1. Make sure the class has run-time class support. That is, the class must have been derived directly or indirectly from **CObject** and used the **DECLARE_DYNAMIC** and **IMPLEMENT_DYNAMIC**, the **DECLARE_DYNCREATE** and **IMPLEMENT_DYNCREATE**, or the **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** macros explained in the article Deriving a Class from CObject.
2. Call the **IsKindOf** member function for objects of that class, using the **RUNTIME_CLASS** macro to generate the **CRuntimeClass** argument, as shown here:

```
// in .H file
class CPerson : public CObject
{
    DECLARE_DYNAMIC( CPerson )
public:
    CPerson(){};

    // other declaration
};

// in .CPP file
IMPLEMENT_DYNAMIC( CPerson, CObject )

void SomeFunction(void)
{
   CObject* pMyObject = new CPerson;

   if(pMyObject->IsKindOf( RUNTIME_CLASS( CPerson ) ) )
```

```
    {
       //if IsKindOf is true, then cast is all right
       CPerson* pmyPerson = (CPerson*) pMyObject ;
       ...
       delete pmyPerson;
    }
...
    delete [MyObject];
}
```

Note

IsKindOf returns **TRUE** if the object is a member of the specified class or of a class derived from the specified class. **IsKindOf** does not support multiple inheritance or virtual base classes, although you can use multiple inheritance for your derived Microsoft Foundation classes if necessary.

One use of run-time class information is in the dynamic creation of objects. This process is discussed in the article Dynamic Object Creation.

For more detailed information on serialization and run-time class information, see the articles Files in MFC and Serialization.

See Also

Concepts

Using CObject

Dynamic Object Creation

This article explains how to create an object dynamically at run time. The procedure uses run-time class information, as discussed in the article Accessing Run-Time Class Information.

To dynamically create an object given its run-time class

- Use the following code to dynamically create an object by using the **CreateObject** function of the **CRuntimeClass**. Note that on failure, **CreateObject** returns **NULL** instead of raising an exception:

```
CRuntimeClass* pRuntimeClass = RUNTIME_CLASS( CMyClass );
CObject* pObject = pRuntimeClass->CreateObject();
ASSERT( pObject->IsKindOf( RUNTIME_CLASS( CMyClass ) ) );
```

See Also

Concepts

Using CObject

CObject Class: Frequently Asked Questions

This section covers questions on class **CObject**.

What do you want to know more about?

- Do I have to derive new classes from CObject?
- What does it cost me to derive a class from CObject?

See Also

Concepts

Using CObject

Do I Have to Derive New Classes from CObject?

No, you don't.

Derive a class from CObject when you need the facilities it provides, such as serialization or dynamic creatability. Many data classes need to be serialized to files, so it's often a good idea to derive them from **CObject**. For an example of a class derived from **CObject**, see the Scribble sample.

See Also

Concepts

CObject Class: Frequently Asked Questions

What Does it Cost me to Derive a Class from CObject?

The overhead in deriving from class CObject is minimal. Your derived class inherits only four virtual functions and a single CRuntimeClass object.

See Also

Concepts

CObject Class: Frequently Asked Questions

Collections

The Microsoft Foundation Class Library provides collection classes to manage groups of objects. These classes are of two types:

- Collection classes created from C++ templates
- Collection classes not created from templates

Tip
The nontemplate collection classes have been provided by MFC beginning with MFC version 1.0. If your code already uses th ese classes, you can continue to use them. If you write new type-safe collection classes for your own data types, consider usin g the newer template-based classes.

Collection Shapes

A collection class is characterized by its "shape" and by the types of its elements. The shape refers to the way the objects are organized and stored by the collection. MFC provides three basic collection shapes: lists, arrays, and maps (also known as dictionaries). You can pick the collection shape most suited to your particular programming problem.

Each of the three provided collection shapes is described briefly below. The table Collection Shape Features in Recommendations for Choosing a Collection Class compares the features of the shapes to help you decide which is best for your program.

- List

  The list class provides an ordered, nonindexed list of elements, implemented as a doubly linked list. A list has a "head" and a "tail," and adding or removing elements from the head or tail, or inserting or deleting elements in the middle, is very fast.

- Array

  The array class provides a dynamically sized, ordered, and integer-indexed array of objects.

- Map (also known as a dictionary)

  A map is a collection that associates a key object with a value object.

The Template-Based Collection Classes

The easiest way to implement a type-safe collection that contains objects of any type is to use one of the MFC template-based classes. For examples of these classes, see the MFC sample COLLECT.

The following table lists the MFC template-based collection classes.

Collection Template Classes

Collection contents	Arrays	Lists	Maps
Collections of objects of any type	**CArray**	**CList**	**CMap**
Collections of pointers to objects of any type	**CTypedPtrArray**	**CTypedPtrList**	**CTypedPtrMap**

The Collection Classes Not Based on Templates

If your application already uses MFC nontemplate classes, you can continue to use them, although for new collections you should consider using the template-based classes. The following table lists the MFC collection classes not based on templates.

Nontemplate Collection Classes

Arrays	Lists	Maps
CObArray	**CObList**	**CMapPtrToWord**
CByteArray	**CPtrList**	**CMapPtrToPtr**

CDWordArray	**CStringList**	**CMapStringToOb**
CPtrArray		**CMapStringToPtr**
CStringArray		**CMapStringToString**
CWordArray		**CMapWordToOb**
CUIntArray		**CMapWordToPtr**

The table Characteristics of MFC Collection Classes in the article Recommendations for Choosing a Collection Class describes the MFC collection classes in terms of their characteristics (other than shape):

- Whether the class uses C++ templates
- Whether the elements stored in the collection can be serialized
- Whether the elements stored in the collection can be dumped for diagnostics
- Whether the collection is type-safe

What do you want to do?

General Collection-Class Tasks

- Choose a collection class
- Make a type-safe collection class
- Use non-template collection classes
- Create a stack collection
- Create a queue collection
- Add data to a collection

Template-Based Collection-Class Tasks

- Use collection classes based on templates
- Use typed-pointer collection class templates
- Implement helper functions for template-based collection classes

Accessing the Members of a Collection (Template-Based or Not)

- Access all members of a collection
- Iterate a collection
- Delete all objects in a CObject collection
- Delete all elements in an array collection class
- Delete all elements in a map collection class
- Delete all elements in a list collection class

See Also

Concepts

General MFC Topics

Other Resources

MFC Concepts

Recommendations for Choosing a Collection Class

This article contains detailed information designed to help you choose a collection class for your particular application needs.

Your choice of a collection class depends on a number of factors, including:

- The features of the class shape: order, indexing, and performance, as shown in the Collection Shape Features table later in this topic
- Whether the class uses C++ templates
- Whether the elements stored in the collection can be serialized
- Whether the elements stored in the collection can be dumped for diagnostics
- Whether the collection is type-safe

The following table, Collection Shape Features, summarizes the characteristics of the available collection shapes.

- Columns 2 and 3 describe each shape's ordering and access characteristics. In the table, the term "ordered" means that the order in which items are inserted and deleted determines their order in the collection; it does not mean the items are sorted on their contents. The term "indexed" means that the items in the collection can be retrieved by an integer index, much like items in a typical array.
- Columns 4 and 5 describe each shape's performance. In applications that require many insertions into the collection, insertion speed might be especially important; for other applications, lookup speed may be more important.
- Column 6 describes whether each shape allows duplicate elements.

Collection Shape Features

Shape	Ordered?	Indexed?	Insert an element	Search for specified element	Duplicate elements?
List	Yes	No	Fast	Slow	Yes
Array	Yes	By int	Slow	Slow	Yes
Map	No	By key	Fast	Fast	No (keys) Yes (values)

The following table, Characteristics of MFC Collection Classes, summarizes other important characteristics of specific MFC collection classes as a guide to selection. Your choice may depend on whether the class is based on C++ templates, whether its elements can be serialized via MFC's document serialization mechanism, whether its elements can be dumped via MFC's diagnostic dumping mechanism, or whether the class is type-safe — that is, whether you can guarantee the type of elements stored in and retrieved from a collection based on the class.

Characteristics of MFC Collection Classes

Class	Uses C++ templates	Can be serialized	Can be dumped	Is type-safe
CArray	Yes	Yes 1	Yes 1	No
CByteArray	No	Yes	Yes	Yes 3
CDWordArray	No	Yes	Yes	Yes 3
CList	Yes	Yes 1	Yes 1	No
CMap	Yes	Yes 1	Yes 1	No
CMapPtrToPtr	No	No	Yes	No
CMapPtrToWord	No	No	Yes	No

CMapStringToOb	No	Yes	Yes	No
CMapStringToPtr	No	No	Yes	No
CMapStringToString	No	Yes	Yes	Yes 3
CMapWordToOb	No	Yes	Yes	No
CMapWordToPtr	No	No	Yes	No
CObArray	No	Yes	Yes	No
CObList	No	Yes	Yes	No
CPtrArray	No	No	Yes	No
CPtrList	No	No	Yes	No
CStringArray	No	Yes	Yes	Yes 3
CStringList	No	Yes	Yes	Yes 3
CTypedPtrArray	Yes	Depends 2	Yes	Yes
CTypedPtrList	Yes	Depends 2	Yes	Yes
CTypedPtrMap	Yes	Depends 2	Yes	Yes
CUIntArray	No	No	Yes	Yes 3
CWordArray	No	Yes	Yes	Yes 3

1. To serialize, you must explicitly call the collection object's **Serialize** function; to dump, you must explicitly call its **Dump** function. You cannot use the form `ar << collObj` to serialize or the form `dmp << collObj` to dump.

2. Serializability depends on the underlying collection type. For example, if a typed pointer array is based on **CObArray**, it is serializable; if based on **CPtrArray**, it is not serializable. In general, the "Ptr" classes cannot be serialized.

3. If marked Yes in this column, a nontemplate collection class is type-safe provided you use it as intended. For example, if you store bytes in a **CByteArray**, the array is type-safe. But if you use it to store characters, its type safety is less certain.

See Also

Tasks

Accessing All Members of a Collection

Concepts

Collections

Template-Based Classes

How to: Make a Type-Safe Collection

Template-Based Classes

This article explains the type-safe template-based collection classes in MFC version 3.0 and later. Using these templates to create type-safe collections is more convenient and helps provide type safety more effectively than using the collection classes not based on templates.

MFC predefines two categories of template-based collections:

- Simple array, list, and map classes

  CArray, **CList**, **CMap**

- Arrays, lists, and maps of typed pointers

  CTypedPtrArray, **CTypedPtrList**, **CTypedPtrMap**

The simple collection classes are all derived from class **CObject**, so they inherit the serialization, dynamic creation, and other properties of **CObject**. The typed pointer collection classes require you to specify the class you derive from — which must be one of the nontemplate pointer collections predefined by MFC, such as **CPtrList** or **CPtrArray**. Your new collection class inherits from the specified base class, and the new class's member functions use encapsulated calls to the base class members to enforce type safety.

For more information about C++ templates, see Templates in the *C++ Language Reference*.

Using Simple Array, List, and Map Templates

To use the simple collection templates, you need to know what kind of data you can store in these collections and what parameters to use in your collection declarations.

Simple Array and List Usage

The simple array and list classes, CArray and CList, take two parameters: *TYPE* and *ARG_TYPE*. These classes can store any data type, which you specify in the *TYPE* parameter:

- Fundamental C++ data types, such as **int**, **char**, and **float**
- C++ structures and classes
- Other types that you define

For convenience and efficiency, you can use the *ARG_TYPE* parameter to specify the type of function arguments. Typically, you specify *ARG_TYPE* as a reference to the type you named in the *TYPE* parameter. For example:

```
CArray<int, int> myArray;
CList<CPerson, CPerson&> myList;
```

The first example declares an array collection, `myArray`, that contains **int**s. The second example declares a list collection, `myList`, that stores `CPerson` objects. Certain member functions of the collection classes take arguments whose type is specified by the *ARG_TYPE* template parameter. For example, the **Add** member function of class **CArray** takes an *ARG_TYPE* argument:

```
CArray<CPerson, CPerson&> myArray;
CPerson person;
myArray->Add( person );
```

Simple Map Usage

The simple map class, CMap, takes four parameters: *KEY*, *ARG_KEY*, *VALUE*, and *ARG_VALUE*. Like the array and list classes, the map classes can store any data type. Unlike arrays and lists, which index and order the data they store, maps associate keys and values: You access a value stored in a map by specifying the value's associated key. The *KEY* parameter specifies the data type of the keys used to access data stored in the map. If the type of *KEY* is a structure or class, the *ARG_KEY* parameter is typically a reference to the type specified in *KEY*. The *VALUE* parameter specifies the type of the items stored in the map. If the type of *ARG_VALUE* is a structure or class, the *ARG_VALUE* parameter is typically a reference to the type specified in *VALUE*. For example:

```
CMap< int, int, MY_STRUCT, MY_STRUCT& > myMap1;
CMap< CString, LPCSTR, CPerson, CPerson& > myMap2;
```

The first example stores `MY_STRUCT` values, accesses them by **int** keys, and returns accessed `MY_STRUCT` items by reference. The second example stores `CPerson` values, accesses them by **CString** keys, and returns references to accessed items. This example might represent a simple address book, in which you look up persons by last name.

Because the *KEY* parameter is of type **CString** and the *KEY_TYPE* parameter is of type **LPCSTR**, the keys are stored in the map as items of type **CString** but are referenced in functions such as **SetAt** through pointers of type **LPCSTR**. For example:

```
CMap< CString, LPCSTR, CPerson, CPerson& > myMap2;
CPerson person;
LPCSTR lpstrName = "Jones";
myMap2->SetAt( lpstrName, person );
```

Using Typed-Pointer Collection Templates

To use the typed-pointer collection templates, you need to know what kinds of data you can store in these collections and what parameters to use in your collection declarations.

Typed-Pointer Array and List Usage

The typed-pointer array and list classes, CTypedPtrArray and CTypedPtrList, take two parameters: *BASE_CLASS* and *TYPE*. These classes can store any data type, which you specify in the *TYPE* parameter. They are derived from one of the nontemplate collection classes that stores pointers; you specify this base class in *BASE_CLASS*. For arrays, use either **CObArray** or **CPtrArray**. For lists, use either **CObList** or **CPtrList**.

In effect, when you declare a collection based on, say **CObList**, the new class not only inherits the members of its base class, but it also declares a number of additional type-safe member functions and operators that help provide type safety by encapsulating calls to the base class members. These encapsulations manage all necessary type conversion. For example:

```
CTypedPtrArray<CObArray, CPerson*> myArray;
CTypedPtrList<CPtrList, MY_STRUCT*> myList;
```

The first example declares a typed-pointer array, `myArray`, derived from **CObArray**. The array stores and returns pointers to `CPerson` objects (where `CPerson` is a class derived from **CObject**). You can call any **CObArray** member function, or you can call the new type-safe **GetAt** and **ElementAt** functions or use the type-safe **[ ]** operator.

The second example declares a typed-pointer list, `myList`, derived from **CPtrList**. The list stores and returns pointers to `MY_STRUCT` objects. A class based on **CPtrList** is used for storing pointers to objects not derived from **CObject**. **CTypedPtrList** has a number of type-safe member functions: **GetHead**, **GetTail**, **RemoveHead**, **RemoveTail**, **GetNext**, **GetPrev**, and **GetAt**.

Typed-Pointer Map Usage

The typed-pointer map class, CTypedPtrMap, takes three parameters: *BASE_CLASS*, *KEY*, and *VALUE*. The *BASE_CLASS* parameter specifies the class from which to derive the new class: **CMapPtrToWord**, **CMapPtrToPtr**, **CMapStringToPtr**, **CMapWordToPtr**, **CMapStringToOb**, and so on. *KEY* is analogous to *KEY* in **CMap**: It specifies the type of the key used for lookups. *VALUE* is analogous to *VALUE* in **CMap**: It specifies the type of object stored in the map. For example:

```
CTypedPtrMap<CMapPtrToPtr, CString, MY_STRUCT*> myPtrMap;
CTypedPtrMap<CMapStringToOb, CString, CMyObject*> myObjectMap;
```

The first example is a map based on **CMapPtrToPt**r — it uses **CString** keys mapped to pointers to `MY_STRUCT`. You can look up a stored pointer by calling a type-safe **Lookup** member function. You can use the **[ ]** operator to look up a stored pointer and add it if not found. And you can iterate the map using the type-safe **GetNextAssoc** function. You can also call other member functions of class **CMapPtrToPtr**.

The second example is a map based on **CMapStringToO**b — it uses string keys mapped to stored pointers to `CMyObject` objects. You can use the same type-safe members described in the previous paragraph, or you can call members of class **CMapStringToOb**.

Note

If you specify a **class** or **struct** type for the *VALUE* parameter, rather than a pointer or reference to the type, the class or structure must have a copy constructor.

For more information, see How to Make a Type-Safe Collection.

See Also

Concepts

Collections

How to: Make a Type-Safe Collection

This article explains how to make type-safe collections for your own data types. Topics include:

- Using template-based classes for type safety
- Implementing helper functions
- Using nontemplate collection classes

The Microsoft Foundation Class Library provides predefined type-safe collections based on C++ templates. Because they are templates, these classes help provide type safety and ease of use without the type-casting and other extra work involved in using a nontemplate class for this purpose. The MFC sample COLLECT demonstrates the use of template-based collection classes in an MFC application. In general, use these classes any time you write new collections code.

Using Template-Based Classes for Type Safety

To use template-based classes

1. Declare a variable of the collection class type. For example:

```
CList<int, int> m_intList;
```

2. Call the member functions of the collection object. For example:

```
m_intList.AddTail( 100 );
m_intList.RemoveAll( );
```

3. If necessary, implement the helper functions and SerializeElements. For information on implementing these functions, see Implementing Helper Functions.

This example shows the declaration of a list of integers. The first parameter in step 1 is the type of data stored as elements of the list. The second parameter specifies how the data is to be passed to and returned from member functions of the collection class, such as **Add** and **GetAt**.

Implementing Helper Functions

The template-based collection classes **CArray**, **CList**, and **CMap** use five global helper functions that you can customize as needed for your derived collection class. For information on these helper functions, see Collection Class Helpers in the *MFC Reference*. Implementation of the serialization function is necessary for most uses of the template-based collection classes.

Serializing Elements

The **CArray**, **CList**, and **CMap** classes call **SerializeElements** to store collection elements to or read them from an archive.

The default implementation of the **SerializeElements** helper function does a bitwise write from the objects to the archive, or a bitwise read from the archive to the objects, depending on whether the objects are being stored in or retrieved from the archive. Override **SerializeElements** if this action is not appropriate.

If your collection stores objects derived from **CObject** and you use the **IMPLEMENT_SERIAL** macro in the implementation of the collection element class, you can take advantage of the serialization functionality built into **CArchive** and **CObject**:

```
class CPerson : public CObject { . . . };
CArray< CPerson, CPerson& > personArray;

template <> void AFXAPI SerializeElements <CPerson> ( CArchive& ar,
    CPerson* pNewPersons, INT_PTR nCount )
{
    for ( int i = 0; i < nCount; i++, pNewPersons++ )
    {
        // Serialize each CPerson object
        pNewPersons->Serialize( ar );
    }
```

```
}
```

The overloaded insertion operators for **CArchive** call **CObject::Serialize** (or an override of that function) for each **CPerson** object.

Using Nontemplate Collection Classes

MFC also supports the collection classes introduced with MFC version 1.0. These classes are not based on templates. They can be used to contain data of the supported types **CObject***, **UINT**, **DWORD**, and **CString**. You can use these predefined collections (such as **CObList**) to hold collections of any objects derived from **CObject**. MFC also provides other predefined collections to hold primitive types such as **UINT** and void pointers (**void***). In general, however, it is often useful to define your own type-safe collections to hold objects of a more specific class and its derivatives. Note that doing so with the collection classes not based on templates is more work than using the template-based classes.

There are two ways to create type-safe collections with the nontemplate collections:

1. Use the nontemplate collections, with type casting if necessary. This is the easier approach.
2. Derive from and extend a nontemplate type-safe collection.

To use the nontemplate collections with type casting

- Use one of the nontemplate classes, such as **CWordArray**, directly.

  For example, you can create a **CWordArray** and add any 32-bit values to it, then retrieve them. There is nothing more to do. You just use the predefined functionality.

  You can also use a predefined collection, such as **CObList**, to hold any objects derived from **CObject**. A **CObList** collection is defined to hold pointers to **CObject**. When you retrieve an object from the list, you may have to cast the result to the proper type since the **CObList** functions return pointers to **CObject**. For example, if you store `CPerson` objects in a **CObList** collection, you have to cast a retrieved element to be a pointer to a `CPerson` object. The following example uses a **CObList** collection to hold `CPerson` objects:

  ```
  class CPerson : public CObject {...};

  CPerson* p1 = new CPerson(...);
  CObList myList;

  myList.AddHead( p1 );   // No cast needed
  CPerson* p2 = ( CPerson* )myList.GetHead();
  ```

  This technique of using a predefined collection type and casting as necessary may be adequate for many of your collection needs. If you need further functionality or more type safety, use a template-based class, or follow the next procedure.

To derive and extend a nontemplate type-safe collection

- Derive your own collection class from one of the predefined nontemplate classes.

  When you derive your class, you can add type-safe wrapper functions to provide a type-safe interface to existing functions.

  For example, if you derived a list from **CObList** to hold `CPerson` objects, you might add the wrapper functions `AddHeadPerson` and `GetHeadPerson`, as shown below.

  ```
  class CPersonList : public CObList
  {
  public:
      void AddHeadPerson( CPerson* person )
          {AddHead( person );}

      const CPerson* GetHeadPerson()
  ```

```
        {return (CPerson*)GetHead();}
};
```

These wrapper functions provide a type-safe way to add and retrieve `CPerson` objects from the derived list. You can see that for the `GetHeadPerson` function, you are simply encapsulating the type casting.

You can also add new functionality by defining new functions that extend the capabilities of the collection rather than just wrapping existing functionality in type-safe wrappers. For example, the article Deleting All Objects in a CObject Collection describes a function to delete all the objects contained in a list. This function could be added to the derived class as a member function.

See Also

Concepts

Collections

Accessing All Members of a Collection

The MFC array collection classes — both template-based and not — use indexes to access their elements. The MFC list and map collection classes — both template-based and not — use an indicator of type **POSITION** to describe a given position within the collection. To access one or more members of these collections, you first initialize the position indicator and then repeatedly pass that position to the collection and ask it to return the next element. The collection is not responsible for maintaining state information about the progress of the iteration. That information is kept in the position indicator. But, given a particular position, the collection is responsible for returning the next element.

The following procedures show how to iterate over the three main types of collections provided with MFC:

- Iterating an array
- Iterating a list
- Iterating a map

To iterate an array

- Use sequential index numbers with the **GetAt** member function:

```
CTypedPtrArray<CObArray, CPerson*> myArray;

for( int i = 0; i < myArray.GetSize();i++ )
{
    CPerson* thePerson = myArray.GetAt( i );
    ...
}
```

This example uses a typed pointer array that contains pointers to `CPerson` objects. The array is derived from class **CObArray**, one of the nontemplate predefined classes. **GetAt** returns a pointer to a `CPerson` object. For typed pointer collection classes — arrays or lists — the first parameter specifies the base class; the second parameter specifies the type to store.

The **CTypedPtrArray** class also overloads the **[ ]** operator so that you can use the customary array-subscript syntax to access elements of an array. An alternative to the statement in the body of the `for` loop above is

```
CPerson* thePerson = myArray[ i ];
```

This operator exists in both **const** and non-**const** versions. The **const** version, which is invoked for **const** arrays, can appear only on the right side of an assignment statement.

To iterate a list

- Use the member functions **GetHeadPosition** and **GetNext** to work your way through the list:

```
CTypedPtrList<CObList, CPerson*> myList;

POSITION pos = myList.GetHeadPosition();
while( pos != NULL )
{
    CPerson* thePerson = myList.GetNext( pos );
    ...
}
```

This example uses a typed pointer list to contain pointers to `CPerson` objects. The list declaration resembles the one for the array in the procedure To iterate an array but is derived from class **CObList**. **GetNext** returns a pointer to a `CPerson` object.

To iterate a map

- Use **GetStartPosition** to get to the beginning of the map and **GetNextAssoc** to repeatedly get the next key and value from the map, as shown by the following example:

```
CMap<CString, LPCTSTR, CPerson*, CPerson*> myMap;

POSITION pos = myMap.GetStartPosition();
while( pos != NULL )
{
    CPerson* pPerson;
    CString string;
    // Get key ( string ) and value ( pPerson )
    myMap.GetNextAssoc( pos, string, pPerson );
    // Use string and pPerson
}
```

This example uses a simple map template (rather than a typed pointer collection) that uses **CString** keys and stores pointers to `CPerson` objects. When you use access functions such as **GetNextAssoc**, the class provides pointers to `CPerson` objects. If you use one of the nontemplate map collections instead, you must cast the returned **CObject** pointer to a pointer to a `CPerson`.

Note
For nontemplate maps, the compiler requires a reference to a **CObject** pointer in the last parameter to **GetNextAssoc**. On input, you must cast your pointers to that type, as shown in the next example.

The template solution is simpler and helps provide better type safety. The nontemplate code is more complicated, as you can see here:

```
CMapStringToOb myMap;    // A nontemplate collection class

POSITION pos = myMap.GetStartPosition( );
while( pos != NULL )
{
    CPerson* pPerson;
    CString string;
    // Gets key (string) and value (pPerson)
    myMap.GetNextAssoc( pos, string,
                            (CObject*&)pPerson );
    ASSERT( pPerson->IsKindOf(
                 RUNTIME_CLASS( CPerson ) ) );
    // Use string and pPerson
}
```

For more information, see Deleting All Objects in a CObject Collection.

See Also

Concepts

Collections

Deleting All Objects in a CObject Collection

This article explains how to delete all objects in a collection (without deleting the collection object itself).

To delete all the objects in a collection of **CObject**s (or of objects derived from **CObject**), you use one of the iteration techniques described in the article Accessing All Members of a Collection to delete each object in turn.

Caution
Objects in collections can be shared. That is, the collection keeps a pointer to the object, but other parts of the program may also have pointers to the same object. You must be careful not to delete an object that is shared until all the parts have finished using the object.

This article shows you how to delete the objects in:

- A list
- An array
- A map

To delete all objects in a list of pointers to CObject

1. Use **GetHeadPosition** and **GetNext** to iterate through the list.
2. Use the **delete** operator to delete each object as it is encountered in the iteration.
3. Call the **RemoveAll** function to remove all elements from the list after the objects associated with those elements have been deleted.

The following example shows how to delete all objects from a list of `CPerson` objects. Each object in the list is a pointer to a `CPerson` object that was originally allocated on the heap.

```
CTypedPtrList<CObList, CPerson*> myList;
POSITION pos = myList.GetHeadPosition();

while( pos != NULL )
{
    delete myList.GetNext( pos );
}
myList.RemoveAll();
```

The last function call, **RemoveAll**, is a list member function that removes all elements from the list. The member function **RemoveAt** removes a single element.

Notice the difference between deleting an element's object and removing the element itself. Removing an element from the list merely removes the list's reference to the object. The object still exists in memory. When you delete an object, it ceases to exist and its memory is reclaimed. Thus, it is important to remove an element immediately after the element's object has been deleted so that the list won't try to access objects that no longer exist.

To delete all elements in an array

1. Use **GetSize** and integer index values to iterate through the array.
2. Use the **delete** operator to delete each element as it is encountered in the iteration.
3. Call the **RemoveAll** function to remove all elements from the array after they have been deleted.

   The code for deleting all elements of an array is as follows:

   ```
   CArray<CPerson*, CPerson*> myArray;

   int i = 0;
   ```

```
while (i < myArray.GetSize() )
{
    delete myArray.GetAt( i++ );
}

myArray.RemoveAll();
```

As with the list example above, you can call **RemoveAll** to remove all elements in an array or **RemoveAt** to remove an individual element.

To delete all elements in a map

1. Use **GetStartPosition** and **GetNextAssoc** to iterate through the array.
2. Use the **delete** operator to delete the key and/or value for each map element as it is encountered in the iteration.
3. Call the **RemoveAll** function to remove all elements from the map after they have been deleted.

   The code for deleting all elements of a **CMap** collection is as follows. Each element in the map has a string as the key and a `CPerson` object (derived from **CObject**) as the value.

```
CMap<CString, LPCSTR, CPerson*, CPerson*> myMap;
// ... Add some key-value elements ...
// Now delete the elements
POSITION pos = myMap.GetStartPosition();
while( pos != NULL )
{
    CPerson* pPerson;
    CString string;
    // Gets key (string) and value (pPerson)
    myMap.GetNextAssoc( pos, string, pPerson );
    delete pPerson;
}
// RemoveAll deletes the keys
myMap.RemoveAll();
```

You can call **RemoveAll** to remove all elements in a map or **RemoveKey** to remove an individual element with the specified key.

See Also

Tasks

Accessing All Members of a Collection

Creating Stack and Queue Collections

This article explains how to create other data structures, such as stacks and queues, from MFC list classes. The examples use classes derived from **CList**, but you can use **CList** directly unless you need to add functionality.

Stacks

Because the standard list collection has both a head and a tail, it is easy to create a derived list collection that mimics the behavior of a last-in-first-out stack. A stack is like a stack of trays in a cafeteria. As trays are added to the stack, they go on top of the stack. The last tray added is the first to be removed. The list collection member functions **AddHead** and **RemoveHead** can be used to add and remove elements specifically from the head of the list; thus, the most recently added element is the first to be removed.

To create a stack collection

- Derive a new list class from one of the existing MFC list classes and add more member functions to support the functionality of stack operations.

  The following example shows how to add member functions to push elements on to the stack, peek at the top element of the stack, and pop the top element from the stack:

```
class CTray : public CObject { ... };

class CStack : public CTypedPtrList< CObList, CTray* >
{
public:
    // Add element to top of stack
    void Push( CTray* newTray )
        { AddHead( newTray ); }

    // Peek at top element of stack
    CTray* Peek()
        { return IsEmpty() ? NULL : GetHead(); }

    // Pop top element off stack
    CTray* Pop()
        { return RemoveHead(); }
};
```

Note that this approach exposes the underlying **CObList** class. The user can call any **CObList** member function, whether it makes sense for a stack or not.

Queues

Because the standard list collection has both a head and a tail, it is also easy to create a derived list collection that mimics the behavior of a first-in-first-out queue. A queue is like a line of people in a cafeteria. The first person in line is the first to be served. As more people come, they go to the end of the line to wait their turn. The list collection member functions **AddTail** and **RemoveHead** can be used to add and remove elements specifically from the head or tail of the list; thus, the most recently added element is always the last to be removed.

To create a queue collection

- Derive a new list class from one of the predefined list classes provided with the Microsoft Foundation Class Library and add more member functions to support the semantics of queue operations.

  The following example shows how you can append member functions to add an element to the end of the queue and get the element from the front of the queue.

```
class CPerson : public CObject { ... };
```

```
class CQueue : public CTypedPtrList< CObList, CPerson* >
{
public:
    // Go to the end of the line
    void AddToEnd( CPerson* newPerson )
        { AddTail( newPerson ); }        // End of the queue

    // Get first element in line
    CPerson* GetFromFront()
        { return IsEmpty() ? NULL : RemoveHead(); }
};
```

See Also

Concepts

Collections

Exception Handling in MFC

This article explains the exception-handling mechanisms available in MFC. Two mechanisms are available:

- C++ exceptions, available in MFC version 3.0 and later
- The MFC exception macros, available in MFC versions 1.0 and later

If you're writing a new application using MFC, you should use the C++ mechanism. You can use the macro-based mechanism if your existing application already uses that mechanism extensively.

You can readily convert existing code to use C++ exceptions instead of the MFC exception macros. Advantages of converting your code and guidelines for doing so are described in the article Exceptions: Converting from MFC Exception Macros.

If you have already developed an application using the MFC exception macros, you can continue using these macros in your existing code, while using C++ exceptions in your new code. The article Exceptions: Changes to Exception Macros in Version 3.0 gives guidelines for doing so.

Note
To enable C++ exception handling in your code, select Enable C++ Exceptions on the Code Generation page in the C/C++ folder of the project's Property Pages dialog box, or use the /GX compiler option. The default is /GX–, which disables exception handling.

This article covers the following topics:

- When to use exceptions
- MFC exception support
- Further reading about exceptions

When to Use Exceptions

Three categories of outcomes can occur when a function is called during program execution: normal execution, erroneous execution, or abnormal execution. Each category is described below.

- Normal execution

  The function may execute normally and return. Some functions return a result code to the caller, which indicates the outcome of the function. The possible result codes are strictly defined for the function and represent the range of possible outcomes of the function. The result code can indicate success or failure or can even indicate a particular type of failure that is within the normal range of expectations. For example, a file-status function can return a code that indicates that the file does not exist. Note that the term "error code" is not used because a result code represents one of many expected outcomes.

- Erroneous execution

  The caller makes some mistake in passing arguments to the function or calls the function in an inappropriate context. This situation causes an error, and it should be detected by an assertion during program development. (For more information on assertions, see Using Assertions.)

- Abnormal execution

  Abnormal execution includes situations where conditions outside the program's control, such as low memory or I/O errors, are influencing the outcome of the function. Abnormal situations should be handled by catching and throwing exceptions.

Using exceptions is especially appropriate for abnormal execution.

MFC Exception Support

Whether you use the C++ exceptions directly or use the MFC exception macros, you will use CException or **CException**-derived objects that may be thrown by the framework or by your application.

The following table shows the predefined exceptions provided by MFC.

Exception class	Meaning
CMemoryException	Out-of-memory
CFileException	File exception
CArchiveException	Archive/Serialization exception
CNotSupportedException	Response to request for unsupported service
CResourceException	Windows resource allocation exception
CDaoException	Database exceptions (DAO classes)
CDBException	Database exceptions (ODBC classes)
COleException	OLE exceptions
COleDispatchException	Dispatch (automation) exceptions
CUserException	Exception that alerts the user with a message box, then throws a generic CException

Note

MFC supports both C++ exceptions and the MFC exception macros. MFC does not directly support Windows NT structured e xception handlers (SEH), as discussed in Structured Exception Handling.

Further Reading About Exceptions

The following articles explain using the MFC library for exception handing:

- Exceptions: Catching and Deleting Exceptions
- Exceptions: Examining Exception Contents
- Exceptions: Freeing Objects in Exceptions
- Exceptions: Throwing Exceptions from Your Own Functions
- Exceptions: Database Exceptions
- Exceptions: OLE Exceptions

The following articles compare the MFC exception macros with the C++ exception keywords and explain how you can adapt your code:

- Exceptions: Changes to Exception Macros in Version 3.0
- Exceptions: Converting from MFC Exception Macros
- Exceptions: Using MFC Macros and C++ Exceptions

See Also

Reference

C++ Exception Handling

Exceptions: Changes to Exception Macros in Version 3.0

This is an advanced topic.

In MFC version 3.0 and later, the exception-handling macros have been changed to use C++ exceptions. This article tells how those changes can affect the behavior of existing code that uses the macros.

This article covers the following topics:

- Exception types and the CATCH macro
- Re-throwing exceptions

Exception Types and the CATCH Macro

In earlier versions of MFC, the **CATCH** macro used MFC run-time type information to determine an exception's type; the exception's type is determined, in other words, at the catch site. With C++ exceptions, however, the exception's type is always determined at the throw site by the type of the exception object that is thrown. This will cause incompatibilities in the rare case where the type of the pointer to the thrown object differs from the type of the thrown object.

The following example illustrates the consequence of this difference between MFC version 3.0 and earlier versions:

```
TRY
{
    THROW( (CException*) new CCustomException() );
}
CATCH( CCustomException, e )
{
    TRACE( "MFC 2.x will land here\n" );
}
AND_CATCH( CException, e )
{
    TRACE( "MFC 3.0 will land here\n" );
}
END_CATCH
```

This code behaves differently in version 3.0 because control always passes to the first **catch** block with a matching exception-declaration. The result of the throw expression

```
THROW((CException*)new CCustomException());
```

is thrown as a **CException***, even though it is constructed as a **CCustomException**. The **CATCH** macro in MFC versions 2.5 and earlier uses **CObject::IsKindOf** to test the type at run time. Because the expression

```
e->IsKindOf( RUNTIME_CLASS( CException ) )
```

is true, the first catch block catches the exception. In version 3.0, which uses C++ exceptions to implement many of the exception-handling macros, the second catch block matches the thrown **CException**.

Code like this is uncommon. It usually appears when an exception object is passed to another function that accepts a generic **CException***, performs "pre-throw" processing, and finally throws the exception.

To work around this problem, move the throw expression from the function to the calling code and throw an exception of the actual type known to the compiler at the time the exception is generated.

Re-Throwing Exceptions

A catch block cannot throw the same exception pointer that it caught.

For example, this code was valid in previous versions, but will have unexpected results with version 3.0:

```
TRY
{
```

```
    // Do something to throw an exception.
}
CATCH( CSomeException, e )
{
    THROW( e );    // Wrong. Use THROW_LAST() instead
}
END_TRY
```

Using **THROW** in the catch block causes the pointer `e` to be deleted, so that the outer catch site will receive an invalid pointer. Use **THROW_LAST** to re-throw `e`.

For more information, see Exceptions: Catching and Deleting Exceptions.

See Also

Concepts

Exception Handling in MFC

Exceptions: Catching and Deleting Exceptions

The following instructions and examples show you how to catch and delete exceptions. For more information on the **try**, **catch**, and **throw** keywords, see C++ Exception Handling.

Your exception handlers must delete exception objects they handle, because failure to delete the exception causes a memory leak whenever that code catches an exception.

Your **catch** block must delete an exception when:

- The **catch** block throws a new exception.

  Of course, you must not delete the exception if you throw the same exception again:

  ```
  catch(CException* e)
  {
      if (m_bThrowExceptionAgain)
          throw; // Do not delete e
  }
  ```

- Execution returns from within the **catch** block.

Note
When deleting a **CException**, use the **Delete** member function to delete the exception. Do not use the **delete** keyword, beca use it can fail if the exception is not on the heap.

To catch and delete exceptions

- Use the **try** keyword to set up a **try** block. Execute any program statements that might throw an exception within a **try** block.

  Use the **catch** keyword to set up a **catch** block. Place exception-handling code in a **catch** block. The code in the **catch** block is executed only if the code within the **try** block throws an exception of the type specified in the **catch** statement.

  The following skeleton shows how **try** and **catch** blocks are normally arranged:

  ```
  // Normal program statements
  ...

  try
  {
      // Execute some code that might throw an exception.
  }
  catch( CException* e )
  {
      // Handle the exception here.
      // "e" contains information about the exception.
      e->Delete();
  }

  // Other normal program statements
  ...
  ```

  When an exception is thrown, control passes to the first **catch** block whose exception-declaration matches the type of the exception. You can selectively handle different types of exceptions with sequential **catch** blocks as listed below:

  ```
  try
  ```

```
{
    // Execute some code that might throw an exception.
}
catch( CMemoryException* e )
{
    // Handle the out-of-memory exception here.
}
catch( CFileException* e )
{
    // Handle the file exceptions here.
}
catch( CException* e )
{
    // Handle all other types of exceptions here.
}
```

For more information, see Exceptions: Converting from MFC Exception Macros.

See Also

Concepts

Exception Handling in MFC

Exceptions: Converting from MFC Exception Macros

This is an advanced topic.

This article explains how to convert existing code written with Microsoft Foundation Class macros — **TRY**, **CATCH**, **THROW**, and so on — to use the C++ exception-handling keywords **try**, **catch**, and **throw**. Topics include:

- Conversion advantages
- Converting code with exception macros to use C++ exceptions

Advantages of Converting

You probably do not need to convert existing code, although you should be aware of differences between the macro implementations in MFC version 3.0 and the implementations in earlier versions. These differences and subsequent changes in code behavior are discussed in Exceptions: Changes to Exception Macros in Version 3.0.

The principal advantages of converting are:

- Code that uses the C++ exception-handling keywords compiles to a slightly smaller .EXE or .DLL.
- The C++ exception-handling keywords are more versatile: They can handle exceptions of any data type that can be copied (**int**, **float**, **char**, and so on), whereas the macros handle exceptions only of class **CException** and classes derived from it.

The major difference between the macros and the keywords is that code using the macros "automatically" deletes a caught exception when the exception goes out of scope. Code using the keywords does not, so you must explicitly delete a caught exception. For more information, see the article Exceptions: Catching and Deleting Exceptions.

Another difference is syntax. The syntax for macros and keywords differs in three respects:

1. Macro arguments and exception declarations:

   A **CATCH** macro invocation has the following syntax:

   CATCH(*exception_class*, *exception_object_pointer_name* **)**

   Notice the comma between the class name and the object pointer name.

   The exception declaration for the **catch** keyword uses this syntax:

   catch(*exception_type exception_name***)**

   This exception declaration statement indicates the type of exception the catch block handles.

2. Delimitation of catch blocks:

   With the macros, the **CATCH** macro (with its arguments) begins the first catch block; the **AND_CATCH** macro begins subsequent catch blocks, and the **END_CATCH** macro terminates the sequence of catch blocks.

   With the keywords, the **catch** keyword (with its exception declaration) begins each catch block. There is no counterpart to the **END_CATCH** macro; the catch block ends with its closing brace.

3. The throw expression:

   The macros use **THROW_LAST** to re-throw the current exception. The **throw** keyword, with no argument, has the same effect.

Doing the Conversion

To convert code using macros to use the C++ exception-handling keywords

1. Locate all occurrences of the MFC macros **TRY**, **CATCH**, **AND_CATCH**, **END_CATCH**, **THROW**, and **THROW_LAST**.
2. Replace or delete all occurrences of the following macros:

   TRY (Replace it with **try**)

   CATCH (Replace it with **catch**)

AND_CATCH (Replace it with **catch**)

END_CATCH (Delete it)

THROW (Replace it with **throw**)

THROW_LAST (Replace it with **throw**)

3. Modify the macro arguments so that they form valid exception declarations.

   For example, change

   ```
   CATCH( CException, e )
   ```

   to

   ```
   catch( CException* e )
   ```

4. Modify the code in the catch blocks so that it deletes exception objects as necessary. For more information, see the article Exceptions: Catching and Deleting Exceptions.

Here is an example of exception-handling code using MFC exception macros. Note that because the code in the following example uses the macros, the exception `e` is deleted automatically:

```
TRY
{
    // Do something to throw an exception.
}
CATCH(CException, e)
{
    if (m_bPassExceptionsUp)
        THROW_LAST();

    if (m_bReturnFromThisFunction)
        return;

    // Not necessary to delete the exception e.
}
END_CATCH
```

The code in the next example uses the C++ exception keywords, so the exception must be explicitly deleted:

```
try
{
    // Do something to throw an exception.
}
catch(CException* e)
{
    if (m_bPassExceptionsUp)
        throw;

    if (m_bThrowDifferentException)
    {
        e->Delete();
        throw new CMyOtherException;
    }

    if (m_bReturnFromThisFunction)
    {
        e->Delete();
        return;
    }

    e->Delete();
```

```
}
```

For more information, see Exceptions: Using MFC Macros and C++ Exceptions.

See Also

Concepts

Exception Handling in MFC

Exceptions: Using MFC Macros and C++ Exceptions

This article discusses considerations for writing code that uses both the MFC exception-handling macros and the C++ exception-handling keywords.

This article covers the following topics:

- Mixing exception keywords and macros
- Try blocks inside catch blocks

Mixing Exception Keywords and Macros

You can mix MFC exception macros and C++ exception keywords in the same program. But you cannot mix MFC macros with C++ exception keywords in the same block because the macros delete exception objects automatically when they go out of scope, whereas code using the exception-handling keywords does not. For more information, see the article Exceptions: Catching and Deleting Exceptions.

The main difference between the macros and the keywords is that the macros "automatically" delete a caught exception when the exception goes out of scope. Code using the keywords does not; exceptions caught in a catch block must be explicitly deleted. Mixing macros and C++ exception keywords can cause memory leaks when an exception object is not deleted, or heap corruption when an exception is deleted twice.

The following code, for example, invalidates the exception pointer:

```
TRY
{
    TRY
    {
        // Do something to throw an exception.
    }
    CATCH(CException, e)  // The "inner" catch block
    {
        throw;  // Invalid attempt to throw exception
                // to the outer catch block below.
    }
    END_CATCH
}
CATCH(CException, e)  // The "outer" catch block
{
     // Pointer e is invalid because
     // it was deleted in the inner catch block.
}
END_CATCH
```

The problem occurs because `e` is deleted when execution passes out of the "inner" **CATCH** block. Using the **THROW_LAST** macro instead of the **THROW** statement will cause the "outer" **CATCH** block to receive a valid pointer:

```
TRY
{
    TRY
    {
        // Do something to throw an exception.
    }
    CATCH(CException, e)  // The "inner" catch block
    {
        THROW_LAST(); // Throw exception to the outer catch block below.
    }
    END_CATCH
}
CATCH(CException, e)  // The "outer" catch block
{
     // Pointer e is valid because
     // THROW_LAST() was used.
}
```

```
END_CATCH
```

Try Blocks Inside Catch Blocks

You cannot re-throw the current exception from within a **try** block that is inside a **CATCH** block. The following example is invalid:

```
TRY
{
    // Do something to throw an exception.
}
CATCH(CException, e)
{
    try
    {
        throw;  // Wrong.  Causes e (the exception
                // being thrown) to be deleted.
    }
    catch(CException exception)
    {
    }
}
END_CATCH
```

For more information, see Exceptions: Examining Exception Contents.

See Also

Concepts

Exception Handling in MFC

Exceptions: Examining Exception Contents

Although a **catch** block's argument can be of almost any data type, the MFC functions throw exceptions of types derived from the class **CException**. To catch an exception thrown by an MFC function, then, you write a **catch** block whose argument is a pointer to a **CException** object (or an object derived from **CException**, such as **CMemoryException**). Depending on the exact type of the exception, you can examine the data members of the exception object to gather information about the specific cause of the exception.

For example, the **CFileException** type has the **m_cause** data member, which contains an enumerated type that specifies the cause of the file exception. Some examples of the possible return values are **CFileException::fileNotFound** and **CFileException::readOnly**.

The following example shows how to examine the contents of a **CFileException**. Other exception types can be examined similarly.

```
try
{
    // Do something to throw a file exception.
}
catch( CFileException* theException )
{
    if( theException->m_cause == CFileException::fileNotFound )
        TRACE( "File not found\n" );
    theException->Delete();
}
```

For more information, see Exceptions: Freeing Objects in Exceptions and Exceptions: Catching and Deleting Exceptions.

See Also

Concepts

Exception Handling in MFC

Exceptions: Freeing Objects in Exceptions

This article explains the need and the method of freeing objects when an exception occurs. Topics include:

- Handling the exception locally
- Throwing exceptions after destroying objects

Exceptions thrown by the framework or by your application interrupt normal program flow. Thus, it is very important to keep close track of objects so that you can properly dispose of them in case an exception is thrown.

There are two primary methods to do this.

- Handle exceptions locally using the **try** and **catch** keywords, then destroy all objects with one statement.
- Destroy any object in the **catch** block before throwing the exception outside the block for further handling.

These two approaches are illustrated below as solutions to the following problematic example:

```
void SomeFunc()        // Problematic code
{
    CPerson* myPerson = new CPerson;

    // Do something that might throw an exception.
    myPerson->SomeFunc();

    // Now destroy the object before exiting.
    delete myPerson;
}
```

As written above, `myPerson` will not be deleted if an exception is thrown by `SomeFunc`. Execution jumps directly to the next outer exception handler, bypassing the normal function exit and the code that deletes the object. The pointer to the object goes out of scope when the exception leaves the function, and the memory occupied by the object will never be recovered as long as the program is running. This is a memory leak; it would be detected by using the memory diagnostics.

Handling the Exception Locally

The **try/catch** paradigm provides a defensive programming method for avoiding memory leaks and ensuring that your objects are destroyed when exceptions occur. For instance, the example shown earlier in this article could be rewritten as follows:

```
void SomeFunc()
{
    CPerson* myPerson = new CPerson;

    try
    {
        // Do something that might throw an exception.
        myPerson->SomeFunc();
    }
    catch( CException* e )
    {
        // Handle the exception locally
        e->Delete();
    }

    // Now destroy the object before exiting.
    delete myPerson;
}
```

This new example sets up an exception handler to catch the exception and handle it locally. It then exits the function normally and destroys the object. The important aspect of this example is that a context to catch the exception is established with the **try/catch** blocks. Without a local exception frame, the function would never know that an exception had been thrown and would not have the chance to exit normally and destroy the object.

Throwing Exceptions After Destroying Objects

Another way to handle exceptions is to pass them on to the next outer exception-handling context. In your **catch** block, you can do some cleanup of your locally allocated objects and then throw the exception on for further processing.

The throwing function may or may not need to deallocate heap objects. If the function always deallocates the heap object before returning in the normal case, then the function should also deallocate the heap object before throwing the exception. On the other hand, if the function does not normally deallocate the object before returning in the normal case, then you must decide on a case-by-case basis whether the heap object should be deallocated.

The following example shows how locally allocated objects can be cleaned up:

```
void SomeFunc()
{
    CPerson* myPerson = new CPerson;

    try
    {
        // Do something that might throw an exception.
        myPerson->SomeFunc();
    }
    catch( CException e )
    {
         // Destroy the object before passing exception on.
        delete myPerson;
        // Throw the exception to the next handler.
        throw;
    }

    // On normal exits, destroy the object.
    delete myPerson;
}
```

The exception mechanism automatically deallocates frame objects; the destructor of the frame object is also called.

If you call functions that can throw exceptions, you can use **try/catch** blocks to make sure that you catch the exceptions and have a chance to destroy any objects you have created. In particular, be aware that many MFC functions can throw exceptions.

For more information, see Exceptions: Catching and Deleting Exceptions.

See Also

Concepts

Exception Handling in MFC

Exceptions: Throwing Exceptions from Your Own Functions

It is possible to use the MFC exception-handling paradigm solely to catch exceptions thrown by functions in MFC or other libraries. In addition to catching exceptions thrown by library code, you can throw exceptions from your own code if you are writing functions that can encounter exceptional conditions.

When an exception is thrown, execution of the current function is stopped and jumps directly to the **catch** block of the innermost exception frame. The exception mechanism bypasses the normal exit path from a function. Therefore, you must be sure to delete those memory blocks that would be deleted in a normal exit.

To throw an exception

1. Use one of the MFC helper functions, such as **AfxThrowMemoryException**. These functions throw a preallocated exception object of the appropriate type.

   In the following example, a function tries to allocate two memory blocks and throws an exception if either allocation fails:

   ```
   {
       char* p1 = (char*)malloc( SIZE_FIRST );
       if( p1 == NULL )
           AfxThrowMemoryException();
       char* p2 = (char*)malloc( SIZE_SECOND );
       if( p2 == NULL )
       {
           free( p1 );
           AfxThrowMemoryException();
       }

       // ... Do something with allocated blocks ...

       // In normal exit, both blocks are deleted.
       free( p1 );
       free( p2 );
   }
   ```

   If the first allocation fails, you can simply throw the memory exception. If the first allocation is successful but the second one fails, you must free the first allocation block before throwing the exception. If both allocations succeed, you can proceed normally and free the blocks when exiting the function.

   – or –

2. Use a user-defined exception to indicate a problem condition. You can throw an item of any type, even an entire class, as your exception.

   The following example attempts to play a sound through a wave device and throws an exception if there is a failure.

   ```
   #define WAVE_ERROR -5
   {
       // This Win32 API returns 0 if the sound cannot be played.
       // Throw an integer constant if it fails.
       if( !sndPlaySound("SIREN.WAV", SND_ASYNC) )
           throw WAVE_ERROR;
   }
   ```

Note
MFC's default handling of exceptions applies only to pointers to **CException** objects (and objects of **CException**-derived classes). The example above bypasses MFC's exception mechanism.

See Also

Concepts

Exception Handling in MFC

Exceptions: Exceptions in Constructors

When throwing an exception in a constructor, clean up whatever objects and memory allocations you have made prior to throwing the exception, as explained in Exceptions: Throwing Exceptions from Your Own Functions.

When throwing an exception in a constructor, the memory for the object itself has already been allocated by the time the constructor is called. So, the compiler will automatically deallocate the memory occupied by the object after the exception is thrown.

For more information, see Exceptions: Freeing Objects in Exceptions.

See Also

Concepts

Exception Handling in MFC

Exceptions: Database Exceptions

This article explains how to handle database exceptions. Most of the material in this article applies whether you are working with the MFC classes for Open Database Connectivity (ODBC) or the MFC classes for Data Access Objects (DAO). Material specific to one or the other model is explicitly marked. Topics include:

- Approaches to exception handling
- A database exception-handling example

Approaches to Exception Handling

The approach is the same whether you are working with DAO or ODBC.

You should always write exception handlers to handle exceptional conditions.

The most pragmatic approach to catching database exceptions is to test your application with exception scenarios. Determine the likely exceptions that might occur for an operation in your code, and force the exception to occur. Then examine the trace output to see what exception is thrown, or examine the returned error information in the debugger. This lets you know which return codes you'll see for the exception scenarios you are using.

Error Codes Used for ODBC Exceptions

In addition to return codes defined by the framework, which have names of the form **AFX_SQL_ERROR_XXX**, some CDBExceptions are based on ODBC return codes. The return codes for such exceptions have names of the form **SQL_ERROR_XXX**.

The return codes — both framework-defined and ODBC-defined — that the database classes can return are documented under the m_nRetCode data member of class **CDBException**. Additional information about return codes defined by ODBC is available in the ODBC SDK *Programmer's Reference* in the MSDN Library.

Error Codes Used for DAO Exceptions

For DAO exceptions, more information is typically available. You can access error information through three data members of a caught CDaoException object:

- m_pErrorInfo contains a pointer to a CDaoErrorInfo object that encapsulates error information in DAO's collection of error objects associated with the database.
- m_nAfxDaoError contains an extended error code from the MFC DAO classes. These error codes, which have names of the form **AFX_DAO_ERROR_XXX**, are documented under the data member in **CDaoException**.
- m_scode contains an OLE **SCODE** from DAO, if applicable. You'll seldom need to work with this error code, however. Usually more information is available in the other two data members. See the data member for more about **SCODE** values.

Additional information about DAO errors, the DAO Error object type, and the DAO Errors collection is available under class CDaoException.

A Database Exception-Handling Example

The following example attempts to construct a CRecordset-derived object on the heap with the **new** operator, and then open the recordset (for an ODBC data source). For a similar example for the DAO classes, see "DAO Exception Example" below.

ODBC Exception Example

The Open member function could throw an exception (of type CDBException for the ODBC classes), so this code brackets the **Open** call with a **try** block. The subsequent **catch** block will catch a **CDBException**. You could examine the exception object itself, called e, but in this case it is enough to know that the attempt to create a recordset has failed. The **catch** block displays a message box and cleans up by deleting the recordset object.

```
CRecordset* CSectionView::OnGetRecordset()
{
    if ( m_pSet != NULL )
        return m_pSet;        // Recordset already allocated

    m_pSet = new CSectionSet( NULL );
```

```
    try
    {
        m_pSet->Open( );
    }
    catch( CDBException* e )
    {
        AfxMessageBox( e->m_strError,
                       MB_ICONEXCLAMATION );
        // Delete the incomplete recordset object
        delete m_pSet;
        m_pSet = NULL;
        e->Delete();
    }
    return m_pSet;
}
```

DAO Exception Example

The DAO example is similar to the example for ODBC, but you can typically retrieve more kinds of information. The following code also attempts to open a recordset. If that attempt throws an exception, you can examine a data member of the exception object for error information. As with the previous ODBC example, it is probably enough to know that the attempt to create a recordset failed.

```
CDaoRecordset* CSectionView::OnGetRecordset()
{
    if ( m_pSet != NULL )
        return m_pSet;  // Recordset already allocated

    m_pSet = new CSectionSet( NULL );
    try
    {
        m_pSet->Open( );
    }
    catch( CDaoException* e )
    {
         AfxMessageBox(
                e->m_pErrorInfo->m_strDescription,
                MB_ICONEXCLAMATION );
        // Delete the incomplete recordset object
        delete m_pSet;
        m_pSet = NULL;
        e->Delete();
    }
    return m_pSet;
}
```

This code gets an error message string from the m_pErrorInfo member of the exception object. MFC fills this member when it throws the exception.

For a discussion of the error information returned by a **CDaoException** object, see classes CDaoException and CDaoErrorInfo.

When you are working with Microsoft Jet (.mdb) databases, and in most cases when you are working with ODBC, there will be only one error object. In the rare case when you are using an ODBC data source and there are multiple errors, you can loop through DAO's Errors collection based on the number of errors returned by CDaoException::GetErrorCount. Each time through the loop, call CDaoException::GetErrorInfo to refill the **m_pErrorInfo** data member.

See Also

Concepts

Exception Handling in MFC

Exceptions: OLE Exceptions

The techniques and facilities for handling exceptions in OLE are the same as those for handling other exceptions. For further information on exception handling, see the article C++ Exception Handling.

All exception objects are derived from the abstract base class **CException**. MFC provides two classes for handling OLE exceptions:

- COleException For handling general OLE exceptions.
- COleDispatchException For generating and handling OLE dispatch (automation) exceptions.

The difference between these two classes is the amount of information they provide and where they are used. **COleException** has a public data member that contains the OLE status code for the exception. **COleDispatchException** supplies more information, including the following:

- An application-specific error code
- An error description, such as "Disk full"
- A Help context that your application can use to provide additional information for the user
- The name of your application's Help file
- The name of the application that generated the exception

COleDispatchException provides more information so that it can be used with products like Microsoft Visual Basic. The verbal error description can be used in a message box or other notification; the Help information can be used to help the user respond to the conditions that caused the exception.

Two global functions correspond to the two OLE exception classes: AfxThrowOleException and AfxThrowOleDispatchException. Use them to throw general OLE exceptions and OLE dispatch exceptions, respectively.

See Also

Concepts

Exception Handling in MFC

Files in MFC

In the Microsoft Foundation Class Library (MFC), class CFile handles normal file I/O operations. This family of articles explains how to open and close files as well as read and write data to those files. It also discusses file status operations. For a description of how to use the object-based serialization features of MFC as an alternative way of reading and writing data in files, see the article Serialization.

Note
When you use MFC **CDocument** objects, the framework does much of the serialization work for you. In particular, the frame work creates and uses the **CFile** object. You only have to write code in your override of the **Serialize** member function of cla ss **CDocument**.

The **CFile** class provides an interface for general-purpose binary file operations. The **CStdioFile** and **CMemFile** classes derived from **CFile** and the **CSharedFile** class derived from **CMemFile** supply more specialized file services.

For more information about alternatives to MFC file handling, see File Handling in the *Run-Time Library Reference*.

For information about derived **CFile** classes, see the MFC hierarchy chart.

What do you want to do?

Use CFile

- Open a file with CFile
- Read and write a file with CFile
- Close a file with CFile
- Access file status with CFile

Use MFC Serialization (Object Persistence)

- Create a serializable class
- Serialize an object via a CArchive object
- Create a CArchive object
- Use CArchive << and >> operators
- Store and load CObjects and CObject-derived objects via an archive

See Also

Reference

CArchive Class

CObject Class

Concepts

General MFC Topics

Other Resources

MFC Concepts

Opening Files

In MFC, the most common way to open a file is a two-stage process.

To open a file

1. Create the file object without specifying a path or permission flags.

   You usually create a file object by declaring a **CFile** variable on the stack frame.

2. Call the **Open** member function for the file object, supplying a path and permission flags.

   The return value for **Open** will be nonzero if the file was opened successfully or 0 if the specified file could not be opened. The **Open** member function is prototyped as follows:

   ```
   virtual BOOL Open( LPCTSTR lpszFileName, UINT nOpenFlags, CFileException* pError = NUL
   L );
   ```

   The open flags specify which permissions, such as read-only, you want for the file. The possible flag values are defined as enumerated constants within the **CFile** class, so they are qualified with "**CFile::**" as in **CFile::modeRead**. Use the **CFile::modeCreate** flag if you want to create the file.

The following example shows how to create a new file with read/write permission (replacing any previous file with the same path):

```
char* pszFileName = "c:\\test\\myfile.dat";
CFile myFile;
CFileException fileException;

if ( !myFile.Open( pszFileName, CFile::modeCreate |
          CFile::modeReadWrite, &fileException ) )
{
    TRACE( "Can't open file %s, error = %u\n",
       pszFileName, fileException.m_cause );
}
```

Note

This example creates and opens a file. If there are problems, the **Open** call can return a **CFileException** object in its last para meter, as shown here. The **TRACE** macro prints both the file name and a code indicating the reason for failure. You can call th e **AfxThrowFileException** function if you require more detailed error reporting.

See Also

Concepts

Files in MFC

Reading and Writing Files

If you've used the C run-time library file-handling functions, MFC reading and writing operations will appear familiar. This article describes reading directly from and writing directly to a **CFile** object. You can also do buffered file I/O with the CArchive class.

To read from and write to the file

1. Use the **Read** and **Write** member functions to read and write data in the file.

   -or-

2. The **Seek** member function is also available for moving to a specific offset within the file.

Read takes a pointer to a buffer and the number of bytes to read and returns the actual number of bytes that were read. If the required number of bytes could not be read because end-of-file (EOF) is reached, the actual number of bytes read is returned. If any read error occurs, an exception is thrown. **Write** is similar to **Read**, but the number of bytes written is not returned. If a write error occurs, including not writing all the bytes specified, an exception is thrown. If you have a valid **CFile** object, you can read from it or write to it as shown in the following example:

```
char      szBuffer[256];
UINT      nActual = 0;
CFile myFile;

myFile.Write( szBuffer, sizeof( szBuffer ) );
myFile.Seek( 0, CFile::begin );
nActual = myFile.Read( szBuffer, sizeof( szBuffer ) );
```

Note

You should normally carry out input/output operations within a **try**/**catch** exception handling block. For more information, see Exception Handling (MFC).

See Also

Concepts

Files in MFC

Closing Files

As usual in I/O operations, once you finish with a file, you must close it.

To close a file

- Use the **Close** member function. This function closes the file-system file and flushes buffers if necessary.

If you allocated the CFile object on the frame (as in the example shown in Opening Files), the object will automatically be closed and then destroyed when it goes out of scope. Note that deleting the **CFile** object does not delete the physical file in the file system.

See Also

Concepts

Files in MFC

Accessing File Status

CFile also supports getting file status, including whether the file exists, creation and modification dates and times, logical size, and path.

To get file status

- Use the CFile class to get and set information about a file. One useful application is to use the **CFile** static member function **GetStatus** to determine if a file exists. **GetStatus** returns 0 if the specified file does not exist.

Thus, you could use the result of **GetStatus** to determine whether to use the **CFile::modeCreate** flag when opening a file, as shown by the following example:

```
CFile theFile;
char* szFileName = "c:\\test\\myfile.dat";
BOOL bOpenOK;

CFileStatus status;
if( CFile::GetStatus( szFileName, status ) )
{
    // Open the file without the Create flag
    bOpenOK = theFile.Open( szFileName,
                    CFile::modeWrite );
}
else
{
    // Open the file with the Create flag
    bOpenOK = theFile.Open( szFileName,
                    CFile::modeCreate | CFile::modeWrite );
}
```

For related information, see Serialization.

See Also

Concepts

Files in MFC

MAPI

This article describes the Microsoft Messaging Application Programming Interface (MAPI) for client message application developers. MFC supplies support for a subset of MAPI in class **CDocument** but does not encapsulate the entire API. For more information, see MAPI Support in MFC.

MAPI is a set of functions that mail-enabled and mail-aware applications use to create, manipulate, transfer, and store mail messages. It gives application developers the tools to define the purpose and content of mail messages and gives them flexibility in their management of stored mail messages. MAPI also provides a common interface that application developers can use to create mail-enabled and mail-aware applications independent of the underlying messaging system.

Messaging clients provide a human interface for interaction with the Microsoft Windows Messaging System (WMS). This interaction typically includes requesting services from MAPI-compliant providers such as message stores and address books.

For more information about MAPI, see the articles under Guide in Win32 Messaging (MAPI) of the Platform SDK.

In This Section

MAPI Support in MFC

See Also

Reference

CDocument::OnFileSendMail
CDocument::OnUpdateFileSendMail
COleDocument::OnFileSendMail

Other Resources

General Concepts

MAPI Support in MFC

MFC supplies support for a subset of the Microsoft Messaging Application Program Interface (MAPI) in class **CDocument**. Specifically, **CDocument** has member functions that determine whether mail support is present on the end-user's machine and, if so, enable a Send Mail command whose standard command ID is **ID_FILE_SEND_MAIL**. The MFC handler function for this command allows the user to send a document through electronic mail.

Tip
Although MFC does not encapsulate the entire MAPI function set, you can still call MAPI functions directly, just as you can call Win32 API functions directly from MFC programs.

Providing the Send Mail command in your application is very easy. MFC provides the implementation to package a document (that is, a **CDocument**-derived object) as an attachment and send it as mail. This attachment is equivalent to a File Save command that saves (serializes) the document's contents to the mail message. This implementation calls upon the mail client on the user's machine to give the user the opportunity to address the mail and to add subject and message text to the mail message. Users see their familiar mail application's user interface. This functionality is supplied by two **CDocument** member functions: **OnFileSendMail** and **OnUpdateFileSendMail**.

MAPI needs to read the file to send the attachment. If the application keeps its data file open during an **OnFileSendMail** function call, the file needs to be opened with a share mode that allows multiple processes to access the file.

Note
An overriding version of **OnFileSendMail** for class **COleDocument** correctly handles compound documents.

To implement a Send Mail command with MFC

1. Use the Visual C++ menu editor to add a menu item whose command ID is **ID_FILE_SEND_MAIL**.

   This command ID is provided by the framework in AFXRES.H. The command can be added to any menu, but it is usually added to the **File** menu.

2. Manually add the following to your document's message map:

   ```
   ON_COMMAND(ID_FILE_SEND_MAIL, OnFileSendMail)
   ON_UPDATE_COMMAND_UI(ID_FILE_SEND_MAIL, OnUpdateFileSendMail)
   ```

Note
This message map works for a document derived from either **CDocument** or **COleDocument** — it picks up the correc t base class in either case, even though the message map is in your derived document class.

3. Build your application.

If mail support is available, MFC enables your menu item with **OnUpdateFileSendMail** and subsequently processes the command with **OnFileSendMail**. If mail support is not available, MFC automatically removes your menu item so the user will not see it.

Tip
Rather than manually adding message map entries as previously described, you can use the class Properties window to map messages to functions. For more information, see Mapping Messages to Functions.

For related information, see the MAPI overview.

For more information about the **CDocument** member functions that enable MAPI, see:

- CDocument::OnFileSendMail

- CDocument::OnUpdateFileSendMail
- COleDocument::OnFileSendMail

See Also

Concepts

MAPI

MAPI Samples

See the following sample programs that illustrate Microsoft Messaging Application Programming Interface (MAPI) functionality:

- NPP
- DRAWCLI

See Also

Concepts

MAPI

Memory Management

This group of articles describes how to take advantage of the general-purpose services of the Microsoft Foundation Class Library (MFC) related to memory management. Memory allocation can be divided into two main categories: frame allocations and heap allocations.

One main difference between the two allocation techniques is that with frame allocation you typically work with the actual memory block itself, while with heap allocation you are always given a pointer to the memory block. Another major difference between the two schemes is that frame objects are automatically deleted, while heap objects must be explicitly deleted by the programmer.

For non-MFC information about memory management in programs for Windows, see Memory Management in the *Platform SDK*.

What do you want to know more about?

- Frame allocation
- Heap allocation
- Allocating memory for an array
- Deallocating memory for an array from the heap
- Allocating memory for a data structure
- Allocating memory for an object
- Resizable memory blocks

See Also

Concepts

General MFC Topics

Other Resources

MFC Concepts

Memory Management: Frame Allocation

Allocation on the frame takes its name from the "stack frame" that is set up whenever a function is called. The stack frame is an area of memory that temporarily holds the arguments to the function as well as any variables that are defined local to the function. Frame variables are often called "automatic" variables because the compiler automatically allocates the space for them.

There are two key characteristics of frame allocations. First, when you define a local variable, enough space is allocated on the stack frame to hold the entire variable, even if it is a large array or data structure. Second, frame variables are automatically deleted when they go out of scope:

```
void MyFunction( )
{
    // Local object created on the stack
    CString strName;
    ...
    // Object goes out of scope and is deleted as function ends
}
```

For local function variables, this scope transition happens when the function exits, but the scope of a frame variable can be smaller than a function if nested braces are used. This automatic deletion of frame variables is very important. In the case of simple primitive types (such as **int** or **byte**), arrays, or data structures, the automatic deletion simply reclaims the memory used by the variable. Since the variable has gone out of scope, it cannot be accessed anyway. In the case of C++ objects, however, the process of automatic deletion is a bit more complicated.

When an object is defined as a frame variable, its constructor is automatically invoked at the point where the definition is encountered. When the object goes out of scope, its destructor is automatically invoked before the memory for the object is reclaimed. This automatic construction and destruction can be very handy, but you must be aware of the automatic calls, especially to the destructor.

The key advantage of allocating objects on the frame is that they are automatically deleted. When you allocate your objects on the frame, you don't have to worry about forgotten objects causing memory leaks. (For details on memory leaks, see the article Detecting Memory Leaks in MFC.) A disadvantage of frame allocation is that frame variables cannot be used outside their scope. Another factor in choosing frame allocation versus heap allocation is that for large structures and objects, it is often better to use the heap instead of the stack for storage since stack space is often limited.

See Also

Concepts

Memory Management

Memory Management: Heap Allocation

The heap is reserved for the memory allocation needs of the program. It is an area apart from the program code and the stack. Typical C programs use the functions **malloc** and **free** to allocate and deallocate heap memory. The Debug version of MFC provides modified versions of the C++ built-in operators **new** and **delete** to allocate and deallocate objects in heap memory.

When you use **new** and **delete** instead of **malloc** and **free**, you are able to take advantage of the class library's memory-management debugging enhancements, which can be useful in detecting memory leaks. When you build your program with the Release version of MFC, the standard versions of the **new** and **delete** operators provide an efficient way to allocate and deallocate memory (the Release version of MFC does not provide modified versions of these operators).

Note that the total size of objects allocated on the heap is limited only by your system's available virtual memory.

See Also

Concepts

Memory Management

Memory Management: Examples

This article describes how MFC performs frame allocations and heap allocations for each of the three typical kinds of memory allocations:

- An array of bytes
- A data structure
- An object

Allocation of an Array of Bytes

To allocate an array of bytes on the frame

- Define the array as shown by the following code. The array is automatically deleted and its memory reclaimed when the array variable exits its scope.

```
{
    const int BUFF_SIZE = 128;

    // Allocate on the frame
    char myCharArray[BUFF_SIZE];
    int myIntArray[BUFF_SIZE];
    // Reclaimed when exiting scope
}
```

To allocate an array of bytes (or any primitive data type) on the heap

- Use the **new** operator with the array syntax shown in this example:

```
const int BUFF_SIZE = 128;

// Allocate on the heap
char* myCharArray = new char[BUFF_SIZE];
int* myIntArray = new int[BUFF_SIZE];
```

To deallocate the arrays from the heap

- Use the **delete** operator as follows:

```
delete [] myCharArray;
delete [] myIntArray;
```

Allocation of a Data Structure

To allocate a data structure on the frame

- Define the structure variable as follows:

```
struct MyStructType { int topScore;};
void SomeFunc(void)
{
    // Frame allocation
    MyStructType myStruct;

    // Use the struct
```

```
    myStruct.topScore = 297;

    // Reclaimed when exiting scope
}
```

The memory occupied by the structure is reclaimed when it exits its scope.

To allocate data structures on the heap

- Use **new** to allocate data structures on the heap and **delete** to deallocate them, as shown by the following examples:

```
// Heap allocation
MyStructType* myStruct = new MyStructType;

// Use the struct through the pointer ...
myStruct->topScore = 297;

delete myStruct;
```

Allocation of an Object

To allocate an object on the frame

- Declare the object as follows:

```
{
   CPerson myPerson;     // Automatic constructor call here

   myPerson.SomeMemberFunction();     // Use the object

}
```

The destructor for the object is automatically invoked when the object exits its scope.

To allocate an object on the heap

- Use the **new** operator, which returns a pointer to the object, to allocate objects on the heap. Use the **delete** operator to delete them.

  The following heap and frame examples assume that the `CPerson` constructor takes no arguments.

```
// Automatic constructor call here
CPerson* myPerson = new CPerson;

myPerson->SomeMemberFunction();  // Use the object

delete myPerson;  // Destructor invoked during delete
```

If the argument for the `CPerson` constructor is a pointer to **char**, the statement for frame allocation is:

```
CPerson myPerson( "Joe Smith" );
```

The statement for heap allocation is:

```
CPerson* MyPerson = new CPerson( "Joe Smith" );
```

See Also

Concepts

Memory Management: Heap Allocation

Memory Management: Resizable Memory Blocks

The **new** and **delete** operators, described in the article Memory Management: Examples, are good for allocating and deallocating fixed-size memory blocks and objects. Occasionally, your application may need resizable memory blocks. You must use the standard C run-time library functions malloc, realloc, and free to manage resizable memory blocks on the heap.

Important
Mixing the **new** and **delete** operators with the resizable memory-allocation functions on the same memory block will result in corrupted memory in the Debug version of MFC. You should not use **realloc** on a memory block allocated with **new**. Like wise, you should not allocate a memory block with the **new** operator and delete it with **free**, or use the **delete** operator on a block of memory allocated with **malloc**.

See Also

Concepts

Memory Management: Heap Allocation

Message Handling and Mapping

This article family describes how messages and commands are processed by the MFC framework and how you connect them to their handler functions.

In traditional programs for Windows, Windows messages are handled in a large switch statement in a window procedure. MFC instead uses message maps to map direct messages to distinct class member functions. Message maps are more efficient than virtual functions for this purpose, and they allow messages to be handled by the most appropriate C++ object — application, document, view, and so on. You can map a single message or a range of messages, command IDs, or control IDs.

WM_COMMAND messages — usually generated by menus, toolbar buttons, or accelerators — also use the message-map mechanism. MFC defines a standard routing of command messages among the application, frame window, view, and Active documents in your program. You can override this routing if you need to.

Message maps also supply a way to update user-interface objects (such as menus and toolbar buttons), enabling or disabling them to suit the current context.

For general information about messages and message queues in Windows, see Messages and Message Queues in the *Platform SDK*.

What do you want to know more about?

- Messages and Commands in the Framework
- How the framework calls a message handler
- How the Framework Searches Message Maps
- Declaring Message Handler Functions
- Mapping Messages to Functions
- How to Display Command Information in the Status Bar
- Dynamic update of user-interface objects
- How to: Create a Message Map for a Template Class

See Also

Reference

CWnd Class

CCmdTarget Class

Concepts

General MFC Topics

Other Resources

MFC Concepts

Messages and Commands in the Framework

Applications written for Microsoft Windows are "message driven." In response to events such as mouse clicks, keystrokes, window movements, and so on, Windows sends messages to the proper window. Framework applications process Windows messages like any other application for Windows. But the framework also provides some enhancements that make processing messages easier, more maintainable, and better encapsulated.

The following topics introduce the key terms used in the rest of the article family to discuss messages and commands:

- Messages
- Message handlers
- Message categories
- Windows messages and control-notification messages
- Command messages
- Message maps
- User-interface objects and command IDs
- Command targets

See Also

Concepts

Message Handling and Mapping

Messages

The message loop in the **Run** member function of class **CWinApp** retrieves queued messages generated by various events. For example, when the user clicks the mouse, Windows sends several mouse-related messages, such as **WM_LBUTTONDOWN** when the left mouse button is pressed and **WM_LBUTTONUP** when the left mouse button is released. The framework's implementation of the application message loop dispatches the message to the appropriate window.

The important categories of messages are described in Message Categories.

See Also

Concepts

Messages and Commands in the Framework

Message Handlers

In MFC, a dedicated *handler* function processes each separate message. Message-handler functions are member functions of a class. This documentation uses the terms *message-handler member function*, *message-handler function*, *message handler*, and *handler* interchangeably. Some kinds of message handlers are also called "command handlers."

Writing message handlers accounts for a large proportion of your work in writing a framework application. This article family describes how the message-processing mechanism works.

What does the handler for a message do? It does whatever you want done in response to that message. You can create the handlers by using the Properties window of the class, and then fill in the handler's code using the source code editor.

You can use all of the facilities of Microsoft Visual C++ and MFC to write your handlers. For a list of all classes, see Class Library Overview in the *MFC Reference*.

See Also

Concepts

Messages and Commands in the Framework

Message Categories

What kinds of messages do you write handlers for? There are three main categories:

1. Windows messages

   This includes primarily those messages beginning with the **WM_** prefix, except for **WM_COMMAND**. Windows messages are handled by windows and views. These messages often have parameters that are used in determining how to handle the message.

2. Control notifications

   This includes **WM_COMMAND** notification messages from controls and other child windows to their parent windows. For example, an edit control sends its parent a **WM_COMMAND** message containing the **EN_CHANGE** control-notification code when the user has taken an action that may have altered text in the edit control. The window's handler for the message responds to the notification message in some appropriate way, such as retrieving the text in the control.

   The framework routes control-notification messages like other **WM_** messages. One exception, however, is the **BN_CLICKED** control-notification message sent by buttons when the user clicks them. This message is treated specially as a command message and routed like other commands.

3. Command messages

   This includes **WM_COMMAND** notification messages from user-interface objects: menus, toolbar buttons, and accelerator keys. The framework processes commands differently from other messages, and they can be handled by more kinds of objects, as explained in Command Targets.

Windows Messages and Control-Notification Messages

Messages in categories 1 and 2 — Windows messages and control notifications — are handled by windows: objects of classes derived from class **CWnd**. This includes **CFrameWnd**, **CMDIFrameWnd**, **CMDIChildWnd**, **CView**, **CDialog**, and your own classes derived from these base classes. Such objects encapsulate an **HWND**, a handle to a Windows window.

Command Messages

Messages in category 3 — commands — can be handled by a wider variety of objects: documents, document templates, and the application object itself in addition to windows and views. When a command directly affects some particular object, it makes sense to have that object handle the command. For example, the Open command on the File menu is logically associated with the application: the application opens a specified document upon receiving the command. So the handler for the Open command is a member function of the application class. For more about commands and how they are routed to objects, see How the Framework Calls a Handler.

See Also

Concepts

Messages and Commands in the Framework

Mapping Messages

Each framework class that can receive messages or commands has its own "message map." The framework uses message maps to connect messages and commands to their handler functions. Any class derived from class **CCmdTarget** can have a message map. Other articles explain message maps in detail and describe how to use them.

In spite of the name "message map," message maps handle both messages and commands — all three categories of messages listed in Message Categories.

See Also

Concepts

Messages and Commands in the Framework

User-Interface Objects and Command IDs

Menu items, toolbar buttons, and accelerator keys are "user-interface objects" capable of generating commands. Each such user-interface object has an ID. You associate a user-interface object with a command by assigning the same ID to the object and the command. As explained in Messages, commands are implemented as special messages. The figure "Commands in the Framework" below shows how the framework manages commands. When a user-interface object generates a command, such as `ID_EDIT_CLEAR_ALL`, one of the objects in your application handles the command — in the figure below, the document object's `OnEditClearAll` function is called via the document's message map.

Commands in the Framework

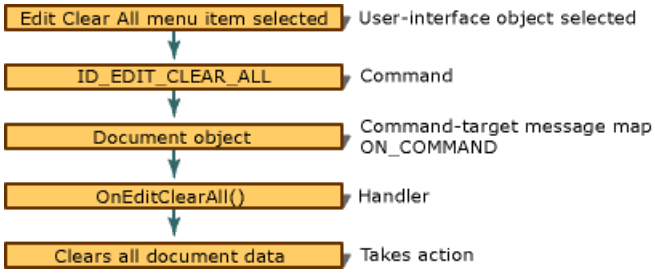

The figure "Command Updating in the Framework" below shows how MFC updates user-interface objects such as menu items and toolbar buttons. Before a menu drops down, or during the idle loop in the case of toolbar buttons, MFC routes an update command. In the figure below, the document object calls its update command handler, `OnUpdateEditClearAll`, to enable or disable the user-interface object.

Command Updating in the Framework

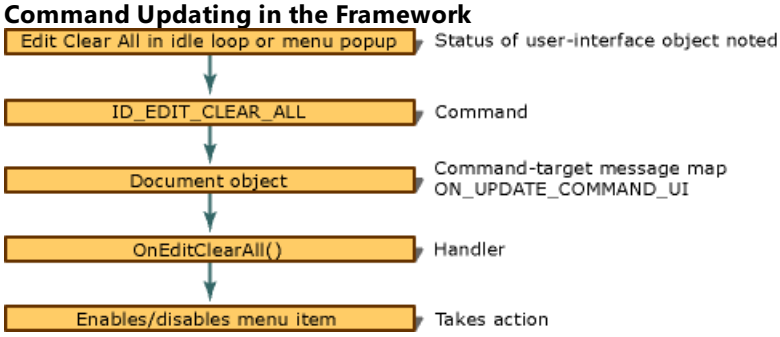

See Also

Concepts

Messages and Commands in the Framework

Command IDs

A command is fully described by its command ID alone (encoded in the **WM_COMMAND** message). This ID is assigned to the user-interface object that generates the command. Typically, IDs are named for the functionality of the user-interface object they are assigned to.

For example, a Clear All item in the Edit menu might be assigned an ID such as **ID_EDIT_CLEAR_ALL**. The class library predefines some IDs, particularly for commands that the framework handles itself, such as **ID_EDIT_CLEAR_ALL** or **ID_FILE_OPEN**. You will create other command IDs yourself.

When you create your own menus in the Visual C++ menu editor, it is a good idea to follow the class library's naming convention as illustrated by **ID_FILE_OPEN**. Standard Commands explains the standard commands defined by the class library.

See Also

Concepts

User-Interface Objects and Command IDs

Standard Commands

The framework defines many standard command messages. The IDs for these commands typically take the form:

ID_ *Source_Item*

where *Source* is usually a menu name and *Item* is a menu item. For example, the command ID for the New command on the File menu is **ID_FILE_NEW**. Standard command IDs are shown in bold type in the documentation. Programmer-defined IDs are shown in a font that is different from the surrounding text.

The following is a list of some of the most important commands supported:

File Menu Commands

New, Open, Close, Save, Save As, Page Setup, Print Setup, Print, Print Preview, Exit, and most-recently-used files.

Edit Menu Commands

Clear, Clear All, Copy, Cut, Find, Paste, Repeat, Replace, Select All, Undo, and Redo.

View Menu Commands

Toolbar and Status Bar.

Window Menu Commands

New, Arrange, Cascade, Tile Horizontal, Tile Vertical, and Split.

Help Menu Commands

Index, Using Help, and About.

OLE Commands (Edit Menu)

Insert New Object, Edit Links, Paste Link, Paste Special, and *typename* Object (verb commands).

The framework provides varying levels of support for these commands. Some commands are supported only as defined command IDs, while others are supported with thorough implementations. For example, the framework implements the Open command on the File menu by creating a new document object, displaying an Open dialog box, and opening and reading the file. In contrast, you must implement commands on the Edit menu yourself, since commands like **ID_EDIT_COPY** depend on the nature of the data you are copying.

For more information about the commands supported and the level of implementation provided, see Technical Note 22. The standard commands are defined in the file AFXRES.H.

See Also

Concepts

User-Interface Objects and Command IDs

Command Targets

The figure Commands in the Framework shows the connection between a user-interface object, such as a menu item, and the handler function that the framework calls to carry out the resulting command when the object is clicked.

Windows sends messages that are not command messages directly to a window whose handler for the message is then called. However, the framework routes commands to a number of candidate objects — called "command targets" — one of which normally invokes a handler for the command. The handler functions work the same way for both commands and standard Windows messages, but the mechanisms by which they are called are different, as explained in How the Framework Calls a Handler.

See Also

Concepts

Messages and Commands in the Framework

How the Framework Calls a Handler

The following topics first examine how the framework routes commands, then examine how other messages and control notifications are sent to windows:

- Message sending and receiving
- How noncommand messages reach their handlers
- Command routing
- Command Routing Example
- The OnCmdMsg Handler
- Overriding the Standard Command Routing

See Also

Concepts

Message Handling and Mapping

Message Sending and Receiving

Consider the sending part of the process and how the framework responds.

Most messages result from user interaction with the program. Commands are generated by mouse clicks in menu items or toolbar buttons or by accelerator keystrokes. The user also generates Windows messages by, for example, moving or resizing a window. Other Windows messages are sent when events such as program startup or termination occur, as windows get or lose the focus, and so on. Control-notification messages are generated by mouse clicks or other user interactions with a control, such as a button or list-box control in a dialog box.

The **Run** member function of class **CWinApp** retrieves messages and dispatches them to the appropriate window. Most command messages are sent to the main frame window of the application. The **WindowProc** predefined by the class library gets the messages and routes them differently, depending on the category of message received.

Now consider the receiving part of the process.

The initial receiver of a message must be a window object. Windows messages are usually handled directly by that window object. Command messages, usually originating in the application's main frame window, get routed to the command-target chain described in Command Routing.

Each object capable of receiving messages or commands has its own message map that pairs a message or command with the name of its handler.

When a command-target object receives a message or command, it searches its message map for a match. If it finds a handler for the message, it calls the handler. For more information about how message maps are searched, see How the Framework Searches Message Maps. Refer again to the figure Commands in the Framework.

See Also

Concepts

How the Framework Calls a Handler

How Noncommand Messages Reach Their Handlers

Unlike commands, standard Windows messages do not get routed through a chain of command targets but are usually handled by the window to which Windows sends the message. The window might be a main frame window, an MDI child window, a standard control, a dialog box, a view, or some other kind of child window.

At run time, each Windows window is attached to a window object (derived directly or indirectly from **CWnd**) that has its own associated message map and handler functions. The framework uses the message map — as for a command — to map incoming messages to handlers.

See Also

Concepts

How the Framework Calls a Handler

Command Routing

Your responsibility in working with commands is limited to making message-map connections between commands and their handler functions, a task for which you use the Properties window. You must also write most command handlers.

Windows messages are usually sent to the main frame window, but command messages are then routed to other objects. The framework routes commands through a standard sequence of command-target objects, one of which is expected to have a handler for the command. Each command-target object checks its message map to see if it can handle the incoming message.

Different command-target classes check their own message maps at different times. Typically, a class routes the command to certain other objects to give them first chance at the command. If none of those objects handles the command, the original class checks its own message map. Then, if it can't supply a handler itself, it may route the command to yet more command targets. The table Standard Command Route below shows how each of the classes structures this sequence. The general order in which a command target routes a command is:

1. To its currently active child command-target object.
2. To itself.
3. To other command targets.

How expensive is this routing mechanism? Compared to what your handler does in response to a command, the cost of the routing is low. Bear in mind that the framework generates commands only when the user interacts with a user-interface object.

Standard Command Route

When an object of this type receives a command . . .	It gives itself and other command-target objects a chance to handle the command in this order:
MDI frame window (**CMDIFrameWnd**)	1. Active **CMDIChildWnd** 2. This frame window 3. Application (**CWinApp** object)
Document frame window (**CFrameWnd**, **CMDIChildWnd**)	1. Active view 2. This frame window 3. Application (**CWinApp** object)
View	1. This view 2. Document attached to the view
Document	1. This document 2. Document template attached to the document
Dialog box	1. This dialog box 2. Window that owns the dialog box 3. Application (**CWinApp** object)

Where numbered entries in the second column of the preceding table mention other objects, such as a document, see the corresponding item in the first column. For instance, when you read in the second column that the view forwards a command to its document, see the "Document" entry in the first column to follow the routing further.

See Also

Concepts

How the Framework Calls a Handler

Command Routing Example

To illustrate, consider a command message from a Clear All menu item in an MDI application's Edit menu. Suppose the handler function for this command happens to be a member function of the application's document class. Here's how that command reaches its handler after the user chooses the menu item:

1. The main frame window receives the command message first.
2. The main MDI frame window gives the currently active MDI child window a chance to handle the command.
3. The standard routing of an MDI child frame window gives its view a chance at the command before checking its own message map.
4. The view checks its own message map first and, finding no handler, next routes the command to its associated document.
5. The document checks its message map and finds a handler. This document member function is called and the routing stops.

If the document did not have a handler, it would next route the command to its document template. Then the command would return to the view and then the frame window. Finally, the frame window would check its message map. If that check failed as well, the command would be routed back to the main MDI frame window and then to the application object — the ultimate destination of unhandled commands.

See Also

Concepts

How the Framework Calls a Handler

The OnCmdMsg Handler

To accomplish the routing of commands, each command target calls the **OnCmdMsg** member function of the next command target in the sequence. Command targets use **OnCmdMsg** to determine whether they can handle a command and to route it to another command target if they cannot handle it.

Each command-target class may override the **OnCmdMsg** member function. The overrides let each class route commands to a particular next target. A frame window, for example, always routes commands to its current child window or view, as shown in the table Standard Command Route.

The default **CCmdTarget** implementation of **OnCmdMsg** uses the message map of the command-target class to search for a handler function for each command message it receives — in the same way that standard messages are searched. If it finds a match, it calls the handler. Message-map searching is explained in How the Framework Searches Message Maps.

See Also

Concepts

How the Framework Calls a Handler

Overriding the Standard Command Routing

In rare cases when you must implement some variation of the standard framework routing, you can override it. The idea is to change the routing in one or more classes by overriding **OnCmdMsg** in those classes. Do so:

- In the class that breaks the order to pass to a nondefault object.
- In the new nondefault object or in command targets it might in turn pass commands to.

If you insert some new object into the routing, its class must be a command-target class. In your overriding versions of **OnCmdMsg**, be sure to call the version that you're overriding. See the OnCmdMsg member function of class **CCmdTarget** in the *MFC Reference* and the versions in such classes as **CView** and **CDocument** in the supplied source code for examples.

See Also

Concepts

How the Framework Calls a Handler

How the Framework Searches Message Maps

The framework searches the message-map table for matches with incoming messages. Once you write a message-map entry for each message you want a class to handle and write the corresponding handlers, the framework calls your handlers automatically. The following topics explain message-map searching:

- Where to find message maps
- Derived message maps
- Mapping ranges of messages, command IDs, or control IDs to one handler

See Also

Concepts

Message Handling and Mapping

Where to Find Message Maps

When you create a new skeleton application with the Application Wizard, the Application Wizard writes a message map for each command-target class it creates for you. This includes your derived application, document, view, and frame-window classes. Some of these message maps already have the entries supplied by the Application Wizard for certain messages and predefined commands, and some are just placeholders for handlers that you will add.

A class's message map is located in the .CPP file for the class. Working with the basic message maps that the Application Wizard creates, you use the Properties window to add entries for the messages and commands that each class will handle. A typical message map might look like the following after you add some entries:

```
BEGIN_MESSAGE_MAP(CMyView, CView)
   ON_WM_MOUSEACTIVATE()
   ON_COMMAND(ID_EDIT_CLEAR_ALL, OnEditClearAll)
   ON_UPDATE_COMMAND_UI(ID_EDIT_CLEAR_ALL, OnUpdateEditClearAll)
   ON_BN_CLICKED(ID_MY_BUTTON, OnMyButton)
END_MESSAGE_MAP()
```

The message map consists of a collection of macros. Two macros, BEGIN_MESSAGE_MAP and END_MESSAGE_MAP, bracket the message map. Other macros, such as **ON_COMMAND**, fill in the message map's contents.

Note
The message-map macros are not followed by semicolons.

When you use the Add Class wizard to create a new class, it provides a message map for the class. Alternatively, you can create a message map manually using the source code editor.

See Also

Concepts

How the Framework Searches Message Maps

Derived Message Maps

During message handling, checking a class's own message map is not the end of the message-map story. What happens if class `CMyView` (derived from **CView**) has no matching entry for a message?

Keep in mind that **CView**, the base class of `CMyView`, is derived in turn from **CWnd**. Thus `CMyView` *is* a **CView** and *is* a **CWnd**. Each of those classes has its own message map. The figure "A View Hierarchy" below shows the hierarchical relationship of the classes, but keep in mind that a `CMyView` object is a single object that has the characteristics of all three classes.

A View Hierarchy

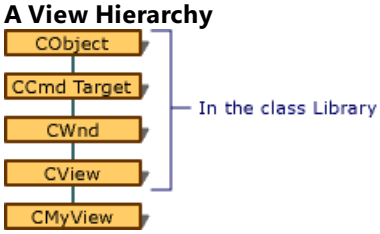

So if a message can't be matched in class `CMyView`'s message map, the framework also searches the message map of its immediate base class. The **BEGIN_MESSAGE_MAP** macro at the start of the message map specifies two class names as its arguments:

```
BEGIN_MESSAGE_MAP(CMyView, CView)
```

The first argument names the class to which the message map belongs. The second argument provides a connection with the immediate base class — **CView** here — so the framework can search its message map, too.

The message handlers provided in a base class are thus inherited by the derived class. This is very similar to normal virtual member functions without needing to make all handler member functions virtual.

If no handler is found in any of the base-class message maps, default processing of the message is performed. If the message is a command, the framework routes it to the next command target. If it is a standard Windows message, the message is passed to the appropriate default window procedure.

To speed message-map matching, the framework caches recent matches on the likelihood that it will receive the same message again. One consequence of this is that the framework processes unhandled messages quite efficiently. Message maps are also more space-efficient than implementations that use virtual functions.

See Also

Concepts

How the Framework Searches Message Maps

Declaring Message Handler Functions

Certain rules and conventions govern the names of your message-handler functions. These depend on the message category, as described in the following topics:

- Handlers for standard Windows messages
- Handlers for commands and control notifications
- Handlers for ranges of messages
- Handling reflected messages

See Also

Concepts

Message Handling and Mapping

Handlers for Standard Windows Messages

Default handlers for standard Windows messages (**WM_**) are predefined in class **CWnd**. The class library bases names for these handlers on the message name. For example, the handler for the **WM_PAINT** message is declared in **CWnd** as:

```
afx_msg void OnPaint();
```

The **afx_msg** keyword suggests the effect of the C++ **virtual** keyword by distinguishing the handlers from other **CWnd** member functions. Note, however, that these functions are not actually virtual; they are instead implemented through message maps. Message maps depend solely on standard preprocessor macros, not on any extensions to the C++ language. The **afx_msg** keyword resolves to white space after preprocessing.

To override a handler defined in a base class, simply define a function with the same prototype in your derived class and to make a message-map entry for the handler. Your handler "overrides" any handler of the same name in any of your class's base classes.

In some cases, your handler should call the overridden handler in the base class so the base class(es) and Windows can operate on the message. Where you call the base-class handler in your override depends on the circumstances. Sometimes you must call the base-class handler first and sometimes last. Sometimes you call the base-class handler conditionally, if you choose not to handle the message yourself. Sometimes you should call the base-class handler, then conditionally execute your own handler code, depending on the value or state returned by the base-class handler.

Caution

It is not safe to modify the arguments passed into a handler if you intend to pass them to a base-class handler. For example, you might be tempted to modify the *nChar* argument of the `OnChar` handler (to convert to uppercase, for example). This beha vior is fairly obscure, but if you need to accomplish this effect, use the **CWnd** member function **SendMessage** instead.

How do you determine the proper way to override a given message? When the Properties window writes the skeleton of the handler function for a given message — an `OnCreate` handler for **WM_CREATE**, for example — it sketches in the form of the recommended overridden member function. The following example recommends that the handler first call the base-class handler and proceed only on condition that it does not return –1.

```
int CMyView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
    // TODO: Add your specialized creation code here
    return 0;
}
```

By convention, the names of these handlers begin with the prefix "On." Some of these handlers take no arguments, while others take several. Some also have a return type other than **void**. The default handlers for all **WM_** messages are documented in the *MFC Reference* as member functions of class **CWnd** whose names begin with "On." The member function declarations in **CWnd** are prefixed with **afx_msg**.

See Also

Concepts

Declaring Message Handler Functions

Handlers for Commands and Control Notifications

There are no default handlers for commands or control-notification messages. Therefore, you are bound only by convention in naming your handlers for these categories of messages. When you map the command or control notification to a handler, the Properties windows proposes a name based on the command ID or control-notification code. You can accept the proposed name, change it, or replace it.

Convention suggests that you name handlers in both categories for the user-interface object they represent. Thus a handler for the Cut command on the Edit menu might be named

```
afx_msg void OnEditCut();
```

Because the Cut command is so commonly implemented in applications, the framework predefines the command ID for the Cut command as **ID_EDIT_CUT**. For a list of all predefined command IDs, see the file AFXRES.H. For more information, see Standard Commands.

In addition, convention suggests a handler for the **BN_CLICKED** notification message from a button labeled "Use As Default" might be named

```
afx_msg void OnClickedUseAsDefault();
```

You might assign this command an ID of `IDC_USE_AS_DEFAULT` because it is equivalent to an application-specific user-interface object.

Both categories of messages take no arguments and return no value.

See Also

Concepts

Declaring Message Handler Functions

Handlers for Message-Map Ranges

This article explains how to map a range of messages to a single message handler function (instead of mapping one message to only one function).

There are times when you need to process more than one message or control notification in exactly the same way. At such times, you might wish to map all of the messages to a single handler function. Message-map ranges allow you to do this for a contiguous range of messages:

- You can map ranges of command IDs to:
  - A command handler function.
  - A command update handler function.
- You can map control-notification messages for a range of control IDs to a message handler function.

Topics covered in this article include:

- Writing the message-map entry
- Declaring the handler function
- Example for a range of command IDs
- Example for a range of control IDs

Writing the Message-Map Entry

In the .CPP file, add your message-map entry, as shown in the following example:

```
...
BEGIN_MESSAGE_MAP(CMyApp, CWinApp)
    ...
ON_COMMAND_RANGE(ID_MYCMD_ONE, ID_MYCMD_TEN, OnDoSomething)
END_MESSAGE_MAP( )
...
```

The message-map entry consists of the following items:

- The message-map range macro:
  - ON_COMMAND_RANGE
  - ON_UPDATE_COMMAND_UI_RANGE
  - ON_CONTROL_RANGE
- Parameters to the macro:

  The first two macros take three parameters:

  - The command ID that starts the range
  - The command ID that ends the range
  - The name of the message handler function

  The range of command IDs must be contiguous.

  The third macro, **ON_CONTROL_RANGE**, takes an additional first parameter: a control-notification message, such as **EN_CHANGE**.

Declaring the Handler Function

Add your handler function declaration in the .H file. The following code shows how this might look, as shown in the next-to-last line below:

```
// Generated message-map functions
protected:
        ...
    afx_msg void OnDoSomething( UINT nID );
    DECLARE_MESSAGE_MAP()
```

Handler functions for single commands normally take no parameters. With the exception of update handler functions, handler functions for message-map ranges require an extra parameter, *nID*, of type **UINT**. This parameter is the first parameter. The extra parameter accommodates the extra command ID needed to specify which command the user actually chose.

For more information about parameter requirements for updating handler functions, see Example for a Range of Command IDs.

Example for a Range of Command IDs

When might you use ranges? One example is in handling commands like the Zoom command in the MFC sample HIERSVR. This command zooms the view, scaling it between 25% and 300% of its normal size. HIERSVR's view class uses a range to handle the Zoom commands with a message-map entry resembling this:

```
ON_COMMAND_RANGE(ID_VIEW_ZOOM25, ID_VIEW_ZOOM300, OnZoom)
```

When you write the message-map entry, you specify:

- Two command IDs, beginning and ending a contiguous range.

  Here they are `ID_VIEW_ZOOM25` and `ID_VIEW_ZOOM300`.

- The name of the handler function for the commands.

  Here it's `OnZoom`.

The function declaration would resemble this:

```
afx_msg void OnZoom(UINT nID);
```

The case of update handler functions is similar, and likely to be more widely useful. It's quite common to write **ON_UPDATE_COMMAND_UI** handlers for a number of commands and find yourself writing, or copying, the same code over and over. The solution is to map a range of command IDs to one update handler function using the **ON_UPDATE_COMMAND_UI_RANGE** macro. The command IDs must form a contiguous range. For an example, see the **OnUpdateZoom** handler and its **ON_UPDATE_COMMAND_UI_RANGE** message-map entry in the HIERSVR sample's view class.

Update handler functions for single commands normally take a single parameter, *pCmdUI*, of type **CCmdUI***. Unlike handler functions, update handler functions for message-map ranges do not require an extra parameter, *nID*, of type **UINT**. The command ID, which is needed to specify which command the user actually chose, is found in the **CCmdUI** object.

Example for a Range of Control IDs

Another interesting case is mapping control-notification messages for a range of control IDs to a single handler. Suppose the user can click any of 10 buttons. To map all 10 buttons to one handler, your message-map entry would look like this:

```
ON_CONTROL_RANGE(BN_CLICKED, IDC_BUTTON1, IDC_BUTTON10, OnButtonClicked)
```

When you write the **ON_CONTROL_RANGE** macro in your message map, you specify:

- A particular control-notification message.

  Here it's **BN_CLICKED**.

- The control ID values associated with the contiguous range of controls.

  Here these are `IDC_BUTTON1` and `IDC_BUTTON10`.

- The name of the message handler function.

Here it's `OnButtonClicked`.

When you write the handler function, specify the extra **UINT** parameter, as shown in the following:

```
...
void CMyDialog::OnButtonClicked( UINT nID )
{
    int nButton = nID - IDC_BUTTON1;
    ASSERT( nButton >= 0 && nButton < 10 );
    // ...
}
```

The `OnButtonClicked` handler for a single **BN_CLICKED** message takes no parameters. The same handler for a range of buttons takes one **UINT**. The extra parameter allows for identifying the particular control responsible for generating the **BN_CLICKED** message.

The code shown in the example is typical: converting the value passed to an **int** within the message range and asserting that this is the case. Then you might take some different action depending on which button was clicked.

See Also

Concepts

Declaring Message Handler Functions

Handling Reflected Messages

Message reflection lets you handle messages for a control, such as **WM_CTLCOLOR**, **WM_COMMAND**, and **WM_NOTIFY**, within the control itself. This makes the control more self-contained and portable. The mechanism works with Windows common controls as well as with ActiveX controls (formerly called OLE controls).

Message reflection lets you reuse your **CWnd**-derived classes more readily. Message reflection works via CWnd::OnChildNotify, using special **ON_XXX_REFLECT** message map entries: for example, **ON_CTLCOLOR_REFLECT** and **ON_CONTROL_REFLECT**. Technical Note 62 explains message reflection in more detail.

What do you want to do?

- Learn more about message reflection
- Implement message reflection for a common control
- Implement message reflection for an ActiveX control

See Also

Concepts

Declaring Message Handler Functions

How to: Display Command Information in the Status Bar

When you run the Application Wizard to create the skeleton of your application, you can easily support a toolbar and a status bar. A single option in the Application Wizard supports both. When a status bar is present, the framework automatically provides helpful feedback as the user of your application moves the mouse through items in the menus. The framework automatically displays a prompt string in the status bar when the menu item is being selected. For example, when the user drags the mouse over the Cut item on the Edit menu, the framework might display "Cut the selection and put it on the Clipboard" in the message area of the status bar. The prompt helps the user understand the menu item's purpose. This also works when the user clicks a toolbar button.

You can easily add to this status-bar help by defining prompt strings for the menu items that you add to the program. To do so, provide the prompt strings when you edit the properties of the menu item in the menu editor. The strings you define this way are stored in your application's resource file; they have the same IDs as the commands they explain.

By default, the Application Wizard adds the ID for a standard prompt, "Ready," which is displayed when the program is waiting for new messages. If you specify the Context-Sensitive Help option in the Application Wizard, the ID for a help prompt, "For Help, press F1," is added to your application. This ID is **AFX_IDS_IDLEMESSAGE**.

See Also

Concepts

Message Handling and Mapping

How to: Create a Message Map for a Template Class

Message mapping in MFC provides an efficient way to direct Windows messages to an appropriate C++ object instance. Examples of MFC message map targets include application classes, document and view classes, control classes, and so on.

Traditional MFC message maps are declared using the BEGIN_MESSAGE_MAP macro to declare the start of the message map, a macro entry for each message-handler class method, and finally the END_MESSAGE_MAP macro to declare the end of the message map.

One limitation with the BEGIN_MESSAGE_MAP macro occurs when it is used in conjunction with a class containing template arguments. When used with a template class, this macro will cause a compile-time error due to the missing template parameters during macro expansion. The BEGIN_TEMPLATE_MESSAGE_MAP macro was designed to allow classes containing a single template argument to declare their own message maps.

Example

Consider an example where the MFC CListBox class is extended to provide synchronization with an external data source. The fictitious **CSyncListBox** class is declared as follows:

```
// Extends the CListBox class to provide synchronization with
// an external data source
template <typename CollectionT>
class CSyncListBox : public CListBox
{
public:
   CSyncListBox();
   virtual ~CSyncListBox();

   afx_msg void OnPaint();
   afx_msg void OnDestroy();
   afx_msg LRESULT OnSynchronize( WPARAM wParam, LPARAM lParam );
   DECLARE_MESSAGE_MAP()

   // ...additional functionality as needed
};
```

The **CSyncListBox** class is templated on a single type that describes the data source it will synchronize with. It also declares three methods that will participate in the message map of the class: **OnPaint**, **OnDestroy**, and **OnSynchronize**. The **OnSynchronize** method is implemented as follows:

```
template <class CollectionT>
LRESULT CSyncListBox<CollectionT>::OnSynchronize( WPARAM, LPARAM lParam )
{
   CollectionT* pCollection = (CollectionT*)( lParam );

   ResetContent();

   if( pCollection != NULL )
   {
      INT nCount = (INT)pCollection->GetCount();
      for( INT n = 0; n < nCount; n++ )
      {
         CString s = StringizeElement( pCollection, n );
         AddString( s );
      }
   }

   return 0L;
}
```

The above implementation allows the **CSyncListBox** class to be specialized on any class type that implements the **GetCount** method, such as **CArray**, **CList**, and **CMap**. The **StringizeElement** function is a template function prototyped by the following:

```
// Template function for converting an element within a collection
// to a CString object
template<typename CollectionT>
CString StringizeElement( CollectionT* pCollection, INT iIndex );
```

Normally, the message map for this class would be defined as:

```
BEGIN_MESSAGE_MAP( CSyncListBox, CListBox )
   ON_WM_PAINT()
   ON_WM_DESTROY()
   ON_MESSAGE( LBN_SYNCHRONIZE, OnSynchronize )
END_MESSAGE_MAP()
```

where **LBN_SYNCHRONIZE** is a custom user message defined by the application, such as:

```
#define LBN_SYNCHRONIZE (WM_USER + 1)
```

The above macro map will not compile, due to the fact that the template specification for the **CSyncListBox** class will be missing during macro expansion. The **BEGIN_TEMPLATE_MESSAGE_MAP** macro solves this by incorporating the specified template parameter into the expanded macro map. The message map for this class becomes:

```
BEGIN_TEMPLATE_MESSAGE_MAP( CSyncListBox, CollectionT, CListBox )
   ON_WM_PAINT()
   ON_WM_DESTROY()
   ON_MESSAGE( LBN_SYNCHRONIZE, OnSynchronize )
END_MESSAGE_MAP()
```

The following demonstrates sample usage of the **CSyncListBox** class using a **CStringList** object:

```
extern CSyncListBox<CStringList>* pctlStringLB;

void CSyncListBox_Test()
{
   // Create a CStringList object and add a few strings
   CStringList stringList;
   stringList.AddTail( "A" );
   stringList.AddTail( "B" );
   stringList.AddTail( "C" );

   // Send a message to the list box control to synchronize its
   // contents with the string list
   pctlStringLB->SendMessage( LBN_SYNCHRONIZE, 0, (LPARAM)&stringList );

   // Verify the contents of the list box by printing out its contents
   INT nCount = pctlStringLB->GetCount();
   for( INT n = 0; n < nCount; n++ )
   {
      CHAR szText[256];
      pctlStringLB->GetText( n, szText );
      TRACE( "%s\n", szText );
   }
}
```

To complete the test, the **StringizeElement** function must be specialized to work with the **CStringList** class:

```
template<>
CString StringizeElement( CStringList* pStringList, INT iIndex )
{
   if( pStringList != NULL && iIndex < pStringList->GetCount() )
   {
      POSITION pos = pStringList->GetHeadPosition();
      for( INT i = 0; i < iIndex; i++ )
```

```
        {
           pStringList->GetNext( pos );
        }
        return pStringList->GetAt( pos );
     }
     return CString(); // or throw, depending on application requirements
  }
```

See Also

Reference

BEGIN_TEMPLATE_MESSAGE_MAP

Concepts

Message Handling and Mapping

MFC COM

A subset of MFC is designed to support COM, while most of the Active Template Library (ATL) is designed for COM programming. This section of topics describes MFC's support for COM.

Active technologies (such as ActiveX controls, Active document containment, OLE, and so on) use the Component Object Model (COM) to enable software components to interact with one another in a networked environment, regardless of the language with which they were created. Active technologies can be used to create applications that run on the desktop or the Internet. For more information see Introduction to COM or The Component Object Model.

Active technologies include both client and server technologies, including the following:

- Active document containment, supported in MFC versions 4.2 and later, allows users to view active documents (such as Microsoft Excel or Word files) and activate the entire interface of the document's native application in the entire client area of an active document container such as the Microsoft Office Binder or Microsoft Internet Explorer. The containers act as clients, while the documents are provided by active document servers. For more information on using active documents in Internet applications, see: Active Documents on the Internet.
- ActiveX controls are interactive objects that can be used in containers such as a Web site. For more information on ActiveX controls, see:
    - MFC ActiveX Controls
    - ActiveX Controls on the Internet
    - Overview: Internet
    - Upgrade an Existing ActiveX Control to be Used on the Internet
    - Debugging an ActiveX Control
- Active scripting controls the integrated behavior of one or more ActiveX controls from a browser or server. For more information on active scripting, see:
    - Script an ActiveX Control on a Web Page
    - Active Technology on the Internet
- Automation (formerly known as OLE Automation) makes it possible for one application to manipulate objects implemented in another application, or to "expose" objects so they can be manipulated.

  The automated object might be local or remote (on another machine accessible across a network). Automation is available for both OLE and COM objects.
- This section also provides information on how to write COM components using MFC, for example, in Connection Points.

For a discussion of what is still called OLE versus what is now called active technology, see the topics on OLE.

Also, see Knowledge Base article Q248019 : HOWTO: Prevent Server Busy Dialog Box From Appearing During a Lengthy COM Operation.

In This Section

Active Document Containment

Automation

Remote Automation

Connection Points

MFC ActiveX Controls

Active Scripting

See Also

Other Resources

General Concepts
MFC Concepts

Active Document Containment

Active document containment is a technology that provides a single frame in which to work with documents, instead of forcing you to create and use multiple application frames for each document type. It differs from basic OLE technology in that OLE works with embedded objects within a compound document in which only a single piece of content can be active. With active document containment, you activate an entire document (that is, an entire application, including associated menus, toolbars, and so on) within the context of a single frame.

The active document containment technology was originally developed for Microsoft Office to implement Office Binder. However, the technology is flexible enough to support active document containers other than Office Binder and can support document servers other than Office and Office-compatible applications.

The application that hosts active documents is called an active document container. Examples of such containers are the Microsoft Office Binder or Microsoft Internet Explorer.

Active document containment is implemented as a set of extensions to OLE documents, the compound document technology of OLE. The extensions are additional interfaces that allow an embeddable, in-place object to represent an entire document instead of a single piece of embedded content. As with OLE documents, active document containment uses a container that provides the display space for active documents, and servers that provide the user interface and manipulation capabilities for the active documents themselves.

An active document server is an application (such as Word, Excel, or PowerPoint) that supports one or more active document classes, where each object itself supports the extension interfaces that allow the object to be activated in a suitable container.

An active document (provided from an active document server such as Word or Excel) is essentially a full-scale, conventional document that is embedded as an object within another active document container. Unlike embedded objects, active documents have complete control over their pages, and the full interface of the application (with all its underlying commands and tools) is available to the user to edit them.

An active document is best understood by distinguishing it from a standard OLE embedded object. Following the OLE convention, an embedded object is one that is displayed within the page of the document that owns it, and the document is managed by an OLE container. The container stores the embedded object's data with the rest of the document. However, embedded objects are limited in that they do not control the page on which they appear.

Users of an active document container application can create active documents (called sections in Office Binder) using their favorite applications (provided these applications are active document enabled), yet the users can manage the resulting project as a single entity, which can be uniquely named, saved, printed, and so on. In the same way, a user of an Internet browser can treat the entire network, as well as local file systems, as a single document storage entity with the ability to browse the documents in that storage from a single location.

Sample Programs

- The MFCBIND sample illustrates the implementation of an active document container application.

See Also

Concepts

MFC COM

Example of Active Document Containment: Office Binder

The Microsoft Office Binder is an example of an active document container. An Office Binder includes two primary panes, as containers typically do. The left pane contains icons that correspond to active documents in the Binder. Each document is called a *section* within the Binder. For example, a Binder can contain Word documents, PowerPoint files, Excel spreadsheets, and so on.

Clicking an icon in the left pane activates the corresponding active document. The right pane of the Binder then displays the contents of the currently selected active document.

If you open and activate a Word document in a Binder, the Word menu bar and toolbars appear at the top of the view frame, and you can edit the document's contents using any Word command or tool. However, the menu bar is a combination of both the Binder's and Word's menu bars. Because both Binder and Word have **Help** menus, the contents of the respective menus are merged. Active document containers such as Office Binder automatically provide **Help** menu merging; for more information, see Help Menu Merging.

When you select an active document of another application type, the Binder's interface changes to accommodate that of the active document's application type. For example, if a Binder contains an Excel spreadsheet, you will observe that the menus in the Binder change when you select the Excel spreadsheet section.

There are, of course, other possible types of containers beside Binders. Windows Explorer uses the typical dual-pane interface in which the left pane uses a tree control to display a hierarchical list of directories in a drive or network, while the right pane displays the files contained in the currently selected directory. An Internet browser–type of container (such as Microsoft Internet Explorer), rather than using a dual-pane interface, usually has a single frame and provides navigation using hyperlinks.

See Also

Concepts

Active Document Containment

Creating an Active Document Container Application

The simplest and most recommended way to create an active document container application is to create an MFC EXE container application using the MFC Application Wizard, then modify the application to support active document containment.

To create an active document container application

1. From the **File** menu, click **Project** from the **New** submenu.
2. From the left pane, click **Visual C++ Projects**.
3. Select **MFC Application** from the right pane.
4. Name the project **MyProj**.
5. Select the **Compound Document Support** page.
6. Select the **Container** or **Container/Full-server** option.
7. Select the **Active document container** check box.
8. Click **Finish**.
9. When the MFC Application Wizard finishes generating the application, open the following files using Solution Explorer:
   - Cntritem.cpp
   - Cntritem.h
   - MyProjview.cpp
   - MyProjview.h
10. In CntrItem.cpp, make the following changes:
    - Replace **COleClientItem** with **COleDocObjectItem**. The latter class augments the OLE code with additional active document containment support.
    - Remove the implementation code from the body of **CMyProjCntrlItem::OnActivate**.

      The **OnActivate** code is not needed because the code in a normal container is used to guarantee that the item being activated is the only active item. Therefore, if there is another active item, it is deactivated first. A basic OLE container lets you have multiple embedded objects in a single document, so it needs this code. The default files generated by the MFC Application Wizard are only set up to allow one active document in the container at a time, so this code is unnecessary. Once you insert an object in your active document container, you cannot insert other objects.
11. In Cntritem.h, make the following changes:
    - Replace **COleClientItem** with **COleDocObjectItem**
12. In MyProjview.cpp, make the following changes:
    - In **CMyProjView::OnPreparePrinting**, add the following code:

      ```
      if (!CView::OnPreparePrinting(pInfo))
         return FALSE;

      if (!COleDocObjectItem::OnPreparePrinting(this, pInfo))
         return FALSE;

      return TRUE;
      ```

    OnPreparePrinting provides printing support. This code replaces **DoPreparePrinting**, which is the default print preparation.

- Add an **OnPrint** override as follows:

```
void CMyProjView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
   // TODO: add code to print the controls
   if(pInfo->m_bDocObject)
      COleDocObjectItem::OnPrint(this, pInfo, TRUE);
}
```

Active document containment provides an improved printing scheme:

- You can first call the active document through its **IPrint** interface and tell it to print itself. This is different from previous OLE containment, in which the container had to render an image of the contained item onto the printer **CDC** object.
- If that fails, tell the contained item to print itself through its **IOleCommandTarget** interface
- If that fails, make your own rendering of the item.

The static member functions **COleDocObjectItem::OnPrint** and **COleDocObjectItem::OnPreparePrinting**, as implemented in the previous code, handle this improved printing scheme.

13. In MyProjview.h, make the following changes:

    - Add the following to Overrides:

```
virtual void OnPrint(CDC* pDC, CPrintInfo* pInfo);
```

Note
In some functions, you do not need to change **COleClientItem** to **COleDocObjectItem**. Because a **COleDocObjectItem** object is a **COleClientItem** object, functions that do not specifically implement active document containment functionality (such as **OnDestroy**, **OnCancelEditCntr**, **OnSetFocus**, and **OnSize**) can implement functionality provided by COM client item objects.

Note
COleDocument maintains a list of **COleClientItem** objects, not **COleDocObjectItem** objects.

14. Add any implementation of your own and build the application.

See Also

Concepts

Active Document Containment

Active Document Containers

An active document container, such as Microsoft Office Binder or Internet Explorer, allows you to work with several documents of different application types within a single frame (instead of forcing you to create and use multiple application frames for each document type).

MFC provides full support for active document containers in the **COleDocObjectItem** class. You can use the MFC Application Wizard to create an active document container by selecting the **Active document container** check box on the **Compound Document Support** page of the MFC Application Wizard. For more information, see Creating an Active Document Container Application.

For more information about active document containers, see:

- Container Requirements
- Document Site Objects
- View Site Objects
- Frame Object
- Help Menu Merging
- Programmatic Printing
- Command Targets

Container Requirements

Active document support in an active document container implies more than just interface implementations: it also requires knowledge of using the interfaces of a contained object. The same applies to active document extensions, where the container must also know how to use those extension interfaces on the active documents themselves.

An active document container that integrates active documents must:

- Be capable of handling object storage through the **IPersistStorage** interface, that is, it must provide an **IStorage** instance to each active document.
- Support the basic embedding features of OLE documents, necessitating "site" objects (one per document or embedding) that implement **IOleClientSite** and **IAdviseSink**.
- Support in-place activation of embedded objects or active documents. The container's site objects must implement **IOleInPlaceSite** and the container's frame object must provide **IOleInPlaceFrame**.
- Support the active documents' extensions by implementing **IOleDocumentSite** to provide the mechanism for the container to talk to the document. Optionally, the container can implement the active document interfaces **IOleCommandTarget** and **IContinueCallback** to pick up simple commands such as printing or saving.

The frame object, the view objects, and the container object can optionally implement **IOleCommandTarget** to support the dispatch of certain commands, as discussed in Command Targets. View and container objects can also optionally implement **IPrint** and **IContinueCallback**, to support programmatic printing, as discussed in Programmatic Printing.

The following figure shows the conceptual relationships between a container and its components (at left), and the active document and its views (at right). The active document manages storage and data, and the view displays or optionally prints that data. Interfaces in bold are those required for active document participation; those bold and italic are optional. All other interfaces are required.

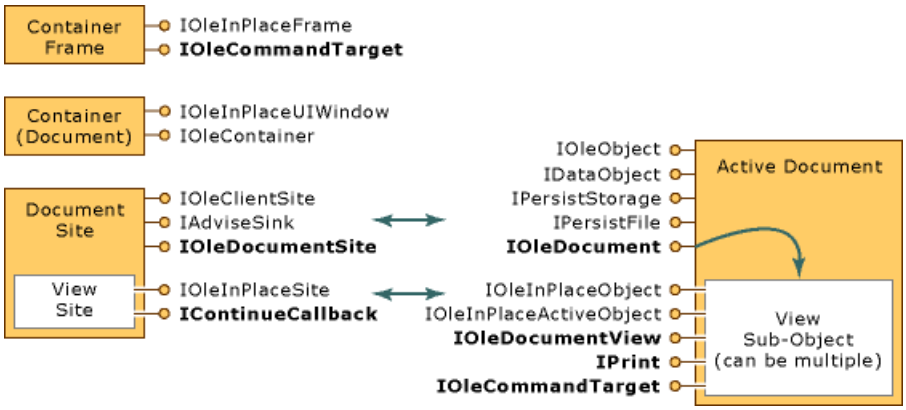

A document that supports only a single view can implement both the view and document components (that is, their corresponding interfaces) on a single concrete class. In addition, a container site that only supports one view at a time can combine the document site and the view site into a single concrete site class. The container's frame object, however, must remain distinct, and the container's document component is merely included here to give a complete picture of the architecture; it is not affected by the active document containment architecture.

Document Site Objects

In the active document containment architecture, a document site is the same as a client site object in OLE Documents with the addition of the **IOleDocument** interface:

```
interface IOleDocumentSite : IUnknown
   {
   HRESULT ActivateMe(IOleDocumentView *pViewToActivate);
   }
```

The document site is conceptually the container for one or more "view site" objects. Each view site object is associated with individual view objects of the document managed by the document site. If the container only supports a single view per document site, then it can implement the document site and the view site with a single concrete class.

View Site Objects

A container's view site object manages the display space for a particular view of a document. In addition to supporting the standard **IOleInPlaceSite** interface, a view site also generally implements **IContinueCallback** for programmatic printing control. (Note that the view object never queries for **IContinueCallback** so it can actually be implemented on any object the container desires.)

A container that supports multiple views must be able to create multiple view site objects within the document site. This provides each view with separate activation and deactivation services as provided through **IOleInPlaceSite**.

Frame Object

The container's frame object is, for the most part, the same frame that is used for in-place activation in OLE Documents, that is, the one that handles menu and toolbar negotiation. A view object has access to this frame object through **IOleInPlaceSite::GetWindowContext**, which also provides access to the container object representing the container document (which can handle pane-level toolbar negotiation and contained object enumeration).

An active document container can augment the frame by adding **IOleCommandTarget**. This allows it to receive commands that originate in the active document's user interface in the same way that this interface can allow a container to send the same commands (such as **File New**, **Open**, **Save As**, **Print**; **Edit Copy**, **Paste**, **Undo**, and others) to an active document. For more information, see Command Targets.

See Also

Concepts

Active Document Containment

Help Menu Merging

When an object is active within a container, the menu merging protocol of OLE Documents gives the object complete control of the **Help** menu. As a result, the container's Help topics are not available unless the user deactivates the object. The active document containment architecture expands on the rules for in-place menu merging to allow both the container and an active document that is active to share the menu. The new rules are simply additional conventions about what component owns what part of the menu and how the shared menu is constructed.

The new convention is simple. In active documents, the **Help** menu has two top-level menu items organized as follows:

```
Help
   Container Help >
   Object Help    >
```

For example, when a Word section is active in the Office Binder, then the **Help** menu would appear as follows:

```
Help
   Binder Help >
   Word Help   >
```

Both menu items are cascading menus under which any additional menu items specific to the container and the object are provided to the user. What items appear here will vary with the container and objects involved.

To construct this merged **Help** menu, the active document containment architecture modifies the normal OLE Documents procedure. According to OLE Documents, the merged menu bar can have six groups of menus, namely **File**, **Edit**, **Container**, **Object**, **Window**, **Help**, in that order. In each group, there can be zero or more menus. The groups **File**, **Container**, and **Window** belong to the container and the groups **Edit**, **Object,** and **Help** belong to the object. When the object wants to do menu merging, it creates a blank menu bar and passes it to the container. The container then inserts its menus, by calling **IOleInPlaceFrame::InsertMenus**. The object also passes a structure that is an array of six LONG values (**OLEMENUGROUPWIDTHS**). After inserting the menus, the container marks how many menus it added in each one of its groups, and then returns. Then the object inserts its menus, paying attention to the count of menus in each container group. Finally, the object passes the merged menu bar and the array (which contains the count of menus in each group) to OLE, which returns an opaque "menu descriptor" handle. Later the object passes that handle and the merged menu bar to the container, via **IOleInPlaceFrame::SetMenu**. At this time, the container displays the merged menu bar and also passes the handle to OLE, so that OLE can do proper dispatching of menu messages.

In the modified active document procedure, the object must first initialize the **OLEMENUGROUPWIDTHS** elements to zero before passing it to the container. Then the container performs a normal menu insertion with one exception: The container inserts a **Help** menu as the last item and stores a value of 1 in the last (sixth) entry of the **OLEMENUGROUPWIDTHS** array (that is, width[5], which belongs to the object's Help group). This **Help** menu will have only one item which is a submenu, the "**Container Help** >" cascade menu as previously described.

The object then executes its normal menu insertion code, except that before inserting its **Help** menu, it checks the sixth entry of the **OLEMENUGROUPWIDTHS** array. If the value is 1 and the name of the last menu is **Help** (or the appropriate localized string), then the object inserts its **Help** menu as submenu of the container's **Help** menu.

The object then sets the sixth element of **OLEMENUGROUPWIDTHS** to zero and increments the fifth element by one. This lets OLE know that the **Help** menu belongs to the container and the menu messages corresponding to that menu (and its submenus) should be routed to the container. It is then the container's responsibility to forward **WM_INITMENUPOPUP**, **WM_SELECT**, **WM_COMMAND**, and other menu-related messages that belong to the object's portion of the **Help** menu. This is accomplished by using **WM_INITMENU** to clear a flag that tells the container whether the user has navigated into the object's **Help** menu. The container then watches **WM_MENUSELECT** for entry into or exit from any item on the **Help** menu that the container did not add itself. On entry, it means the user has navigated into an object menu, so the container sets the "in object Help menu" flag and uses the state of that flag to forward any **WM_MENUSELECT**, **WM_INITMENUPOPUP**, and **WM_COMMAND** messages, as a minimum, to the object window. (On exit, the container clears the flag and then processes these same messages itself.) The container should use the window returned from the object's **IOleInPlaceActiveObejct::GetWindow** function as the destination for these messages.

If the object detects a zero in the sixth element of **OLEMENUGROUPWIDTHS**, it proceeds according to the normal OLE Documents rules. This procedure covers containers that do participate in **Help** menu merging as well as those that do not.

When the object calls **IOleInPlaceFrame::SetMenu**, before displaying the merged menu bar, the container checks whether

the **Help** menu has an additional submenu, in addition to what the container has inserted. If so, the container leaves its **Help** menu in the merged menu bar. If the **Help** menu does not have an additional submenu, the container will remove its **Help** menu from the merged menu bar. This procedure covers objects that participate in **Help** menu merging as well as those that do not.

Finally, when it is time to disassemble the menu, the object removes the inserted **Help** menu in addition to removing the other inserted menus. When the container removes its menus, it will remove its **Help** menu in addition to the other menus that it has inserted.

See Also

Concepts

Active Document Containers

Programmatic Printing

OLE provided the means to uniquely identify persistent documents (**GetClassFile**) and load them into their associated code (**CoCreateInstance**, **QueryInterface(IID_IPersistFile)**, **QueryInterface(IID_IPersistStorage)**, **IPersistFile::Load**, and **IPersistStorage::Load**). To further enable printing documents, active document containment (using an existing OLE design not shipped with OLE 2.0 originally) introduces a base-standard printing interface, **IPrint**, generally available through any object that can load the persistent state of the document type. Each view of an active document can optionally support the **IPrint** interface to provide these capabilities.

The **IPrint** interface is defined as follows:

```
interface IPrint : IUnknown
   {
   HRESULT SetInitialPageNum([in] LONG nFirstPage);
   HRESULT GetPageInfo(
      [out] LONG *nFirstPage,
      [out] LONG *pcPages);
   HRESULT Print(
      [in] DWORD grfFlags,
      [in,out] DVTARGETDEVICE **pptd,
      [in,out] PAGESET ** ppPageSet,
      [in,out] STGMEDIUM **ppstgmOptions,
      [in] IContinueCallback* pCallback,
      [in] LONG nFirstPage,
      [out] LONG *pcPagesPrinted,
      [out] LONG *pnPageLast);
   };
```

Clients and containers simply use **IPrint::Print** to instruct the document to print itself once that document is loaded, specifying printing control flags, the target device, the pages to print, and additional options. The client can also control the continuation of printing through the interface **IContinueCallback** (see below).

In addition, **IPrint::SetInitialPageNum** supports the ability to print a series of documents as one by numbering pages seamlessly, obviously a benefit for active document containers like Office Binder. **IPrint::GetPageInfo** makes displaying pagination information simple by allowing the caller to retrieve the starting page number previously passed to **SetInitialPageNum** (or the document's internal default starting page number) and the number of pages in the document.

Objects that support **IPrint** are marked in the registry with the "Printable" key stored under the object's CLSID:

HKEY_CLASSES_ROOT\CLSID\{...}\Printable

IPrint is usually implemented on the same object that supports either **IPersistFile** or **IPersistStorage**. Callers note the capability to programmatically print the persistent state of some class by looking in the registry for the "Printable" key. Currently, "Printable" indicates support for at least **IPrint**; other interfaces may be defined in the future which would then be available through **QueryInterface** where **IPrint** simply represents the base level of support.

During a print procedure, you may want the client or container that initiated the printing to control whether or not the printing should continue. For example, the container may support a "Stop Print" command that should terminate the print job as soon as possible. To support this capability, the client of a printable object can implement a small notification sink object with the interface **IContinueCallback**:

```
interface IContinueCallback : IUnknown
   {
   HRESULT FContinue(void);
   HRESULT FContinuePrinting(
      [in] LONG cPagesPrinted,
      [in] LONG nCurrentPage,
      [in] LPOLESTR pszPrintStatus);
   };
```

This interface is designed to be useful as a generic continuation callback function that takes the place of the various continuation procedures in the Win32 API (such as the **AbortProc** for printing and the **EnumMetafileProc** for metafile enumeration). Thus this interface design is useful in a wide variety of time-consuming processes.

In the most generic cases, the **IContinueCallback::FContinue** function is called periodically by any lengthy process. The sink object returns **S_OK** to continue the operation, and **S_FALSE** to stop the procedure as soon as possible.

FContinue, however, is not used in the context of **IPrint::Print**; rather, printing uses **IContinueCallback::FContinuePrint**. Any printing object should periodically call **FContinuePrinting** passing the number of pages that have been printing, the number of the page being printed, and an additional string describing the print status that the client may choose to display to the user (such as "Page 5 of 19").

See Also

Concepts

Active Document Containers

Message Handling and Command Targets

The command dispatch interface **IOleCommandTarget** defines a simple and extensible mechanism to query and execute commands. This mechanism is simpler than Automation's **IDispatch** because it relies entirely on a standard set of commands; commands rarely have arguments, and no type information is involved (type safety is diminished for command arguments as well).

In the command dispatch interface design, each command belongs to a "command group" which is itself identified with a **GUID**. Therefore, anyone can define a new group and define all the commands within that group without any need to coordinate with Microsoft or any other vendor. (This is essentially the same means of definition as a **dispinterface** plus **dispIDs** in Automation. There is overlap here, although this command routing mechanism is only for command routing and not for scripting/programmability on a large scale as Automation handles.)

IOleCommandTarget handles the following scenarios:

- When an object is in-place activated, only the object's toolbars are typically displayed and the object's toolbars may have buttons for some of the container commands like **Print**, **Print Preview**, **Save**, **New**, **Zoom**, and others. (In-place activation standards recommend that objects remove such buttons from their toolbars, or at least disable them. This design allows those commands to be enabled and yet routed to the right handler.) Currently, there is no mechanism for the object to dispatch these commands to the container.
- When an active document is embedded in an active document container (such as Office Binder), the container may need to send commands such **Print**, **Page Setup**, **Properties**, and others to the contained active document.

This simple command routing could be handled through existing Automation standards and **IDispatch**. However, the overhead involved with **IDispatch** is more than is necessary here, so **IOleCommandTarget** provides a simpler means to achieve the same ends:

```
interface IOleCommandTarget : IUnknown
   {
   HRESULT QueryStatus(
      [in] GUID *pguidCmdGroup,
      [in] ULONG cCmds,
      [in,out][size_is(cCmds)] OLECMD *prgCmds,
      [in,out] OLECMDTEXT *pCmdText);
   HRESULT Exec(
      [in] GUID *pguidCmdGroup,
      [in] DWORD nCmdID,
      [in] DWORD nCmdExecOpt,
      [in] VARIANTARG *pvaIn,
      [in,out] VARIANTARG *pvaOut);
   }
```

The **QueryStatus** method here tests whether a particular set of commands, the set being identified with a **GUID**, is supported. This call fills an array of **OLECMD** values (structures) with the supported list of commands as well as returning text describing the name of a command and/or status information. When the caller wishes to invoke a command, it can pass the command (and the set **GUID**) to **Exec** along with options and arguments, getting back a return value.

See Also

Concepts

Active Document Containers

Active Document Servers

Active document servers such as Word, Excel, or PowerPoint host documents of other application types called active documents. Unlike OLE embedded objects (which are simply displayed within the page of another document), Active documents provide the full interface and complete native functionality of the server application that creates them. Users can create documents using the full power of their favorite applications (if they are active document enabled), yet can treat the resulting project as a single entity.

Active documents can have more than one page and are always in-place active. Active documents control part of the user interface, merging their menus with the **File** and **Help** menus of the container. They occupy the entire editing area of the container and control the views and the layout of the printer page (margins, footers, and so on).

MFC implements active document servers with document/view interfaces, command dispatch maps, printing, menu management, and registry management. Specific programming requirements are discussed in active documents.

MFC supports active documents with the CDocObjectServer class, derived from CCmdTarget, and CDocObjectServerItem, derived from COleServerItem. MFC supports active document containers with the COleDocObjectItem class, derived from COleClientItem.

CDocObjectServer maps the active document interfaces and initializes and activates an active document. MFC also provides macros to handle command routing in ACTIVE documents. To use active documents in your application, include AfxDocOb.h in your StdAfx.h file.

A regular MFC server hooks up its own **COleServerItem**-derived class. The MFC Application Wizard generates this class for you if you select the **Mini-server** or **Full-server** check box to give your application server compound document support. If you also select the **Active document server** check box, the MFC Application Wizard generates a class derived from **CDocObjectServerItem** instead.

The **COleDocObjectItem** class allows an OLE container to become an active document container. You can use the MFC Application Wizard to create an active document container by selecting the **Active document container** checkbox in the Compound Document Support page of the MFC Application Wizard. For more information, see Creating an Active Document Container Application.

See Also

Concepts

Active Document Containment

Active Documents

Active documents extend the compound document technology of OLE. These extensions are provided in the form of additional interfaces that manage views, so that objects can function within containers and yet retain control over their display and printing functions. This process makes it possible to display documents both in foreign frames (such as the Microsoft Office Binder or Microsoft Internet Explorer) and in native frames (such as the product's own view ports).

This section describes the functional requirements for active documents. The active document owns a set of data and has access to storage where the data can be saved and retrieved. It can create and manage one or more views on its data. In addition to supporting the usual embedding and in-place activation interfaces of OLE documents, the active document communicates its ability to create views through **IOleDocument**. Through this interface, the container can ask to create (and possibly enumerate) the views that the active document can display. Through this interface, the active document can also provide miscellaneous information about itself, such as whether it supports multiple views or complex rectangles.

The following is the **IOleDocument** interface. Note that the **IEnumOleDocumentViews** interface is a standard OLE enumerator for **IOleDocumentView *** types.

```
interface IOleDocument : IUnknown
   {
   HRESULT CreateView(
      [in] IOleInPlaceSite *pIPSite,
      [in] IStream *pstm,
      [in] DWORD dwReserved,
      [out] IOleDocumentView **ppView);
   HRESULT GetDocMiscStatus([out] DWORD *pdwStatus);
   HRESULT EnumViews(
      [out] IEnumOleDocumentViews **ppEnum,
      [out] IOleDocumentView **ppView);
   }
```

Every active document must have a view frame provider with this interface. If the document is not embedded within a container, the active document server itself must provide the view frame. However, when the active document is embedded in an active document container, the container provides the view frame.

An active document can create one or more types of views of its data (for example, normal, outline, page layout, and so on). Views act like filters through which the data can be seen. Even if the document has only one type of view, you may still want to support multiple views as a means of supporting new window functionality (for example, the **New Window** item on the **Window** menu in Office applications).

Requirements for Active Documents

An active document that can be displayed in an active document container must:

- Use OLE's Compound Files as its storage mechanism by implementing **IPersistStorage**.
- Support the basic embedding features of OLE Documents, including **Create From File**. This necessitates the interfaces **IPersistFile**, **IOleObject**, and **IDataObject**.
- Support one or more views, each of which is capable of in-place activation. That is, the views must support the interface **IOleDocumentView** as well as the interfaces **IOleInPlaceObject** and **IOleInPlaceActiveObject** (using the container's **IOleInPlaceSite** and **IOleInPlaceFrame** interfaces).
- Support the standard active document interfaces **IOleDocument**, **IOleCommandTarget**, and **IPrint**.

Knowledge of when and how to use the container-side interfaces is implied in these requirements.

Requirements for View Objects

An active document can create one or more views of its data. Functionally, these views are like ports onto a particular method for displaying the data. If an active document only supports a single view, the active document and that single view can be implemented using a single class. **IOleDocument::CreateView** returns the same object's **IOleDocumentView** interface pointer.

To be represented within an active document container, a view component must support **IOleInPlaceObject** and

IOleInPlaceActiveObject in addition to **IOleDocumentView**:

```
interface IOleDocumentView : IUnknown
   {
   HRESULT SetInPlaceSite([in] IOleInPlaceSite *pIPSite);
   HRESULT GetInPlaceSite([out] IOleInPlaceSite **ppIPSite);
   HRESULT GetDocument([out] IUnknown **ppunk);
   [input_sync] HRESULT SetRect([in] LPRECT prcView);
   HRESULT GetRect([in] LPRECT prcView);
   [input_sync] HRESULT SetRectComplex(
      [in] LPRECT prcView,
      [in] LPRECT prcHScroll,
      [in] LPRECT prcVScroll,
      [in] LPRECT prcSizeBox);
   HRESULT Show([in] BOOL fShow);
   HRESULT UIActivate([in] BOOL fUIActivate);
   HRESULT Open(void);
   HRESULT CloseView([in] DWORD dwReserved);
   HRESULT SaveViewState([in] IStream *pstm);
   HRESULT ApplyViewState([in] IStream *pstm);
   HRESULT Clone(
      [in] IOleInPlaceSite *pIPSiteNew,
      [out] IOleDocumentView **ppViewNew);
   }
```

Every view has an associated view site, which encapsulates the view frame and the view port (HWND and a rectangular area in that window). The site exposes this functionality though the standard **IOleInPlaceSite** interface. Note that it is possible to have more than one view port on a single HWND.

Typically, each type of view has a different printed representation. Hence views and the corresponding view sites should implement the printing interfaces if **IPrint** and **IContinueCallback**, respectively. The view frame must negotiate with the view provider through **IPrint** when printing begins, so that headers, footers, margins, and related elements are printed correctly. The view provider notifies the frame of printing-related events through **IContinueCallback**. For more information on the use of these interfaces, see Programmatic Printing.

Note that if an active document only supports a single view, then the active document and that single view can be implemented using a single concrete class. **IOleDocument::CreateView** simply returns the same object's **IOleDocumentView** interface pointer. In short, it is not necessary that there be two separate object instances when only one view is required.

A view object can also be a command target. By implementing **IOleCommandTarget** a view can receive commands that originate in the container's user interface (such as **New**, **Open**, **Save As**, **Print** on the **File** menu; and **Copy**, **Paste**, **Undo** on the **Edit** menu). For more information, see Command Targets.

See Also

Concepts

Active Document Containment

Automation

Automation (formerly known as OLE Automation) makes it possible for one application to manipulate objects implemented in another application, or to expose objects so they can be manipulated.

An Automation server is an application (a type of COM server) that exposes its functionality through COM interfaces to other applications, called Automation clients. The exposure enables Automation clients to automate certain functions by directly accessing objects and using the services they provide.

Automation servers and clients use COM interfaces that are always derived from **IDispatch** and take and return a specific set of data types called Automation types. You can automate any object that exposes an Automation interface, providing methods and properties that you can access from other applications. Automation is available for both OLE and COM objects. The automated object might be local or remote (on another machine accessible across a network); therefore there are two categories of automation:

- Automation (local).
- Remote Automation (over a network, using Distributed COM, or DCOM).

Exposing objects is beneficial when applications provide functionality useful to other applications. For example, an ActiveX control is a type of Automation server; the application hosting the ActiveX control is the automation client of that control.

As another example, a word processor might expose its spell-checking functionality to other programs. Exposure of objects enables vendors to improve their applications by using the ready-made functionality of other applications. In this way, Automation applies some of the principles of object-oriented programming, such as reusability and encapsulation, at the level of applications themselves.

More important is the support Automation provides to users and solution providers. By exposing application functionality through a common, well-defined interface, Automation makes it possible to build comprehensive solutions in a single general programming language, such as Microsoft Visual Basic, instead of in diverse application-specific macro languages.

Many commercial applications, such as Microsoft Excel and Microsoft Visual C++, allow you to automate much of their functionality. For example, in Visual C++, you can write VBScript macros to automate builds, aspects of code editing, or debugging tasks.

Passing Parameters in Automation

One difficulty in creating Automation methods is helping to provide a uniform "safe" mechanism to pass data between automation servers and clients. Automation uses the **VARIANT** type to pass data. The **VARIANT** type is a tagged union. It has a data member for the value (this is an anonymous C++ union) and a data member indicating the type of information stored in the union. The **VARIANT** type supports a number of standard data types: 2- and 4-byte integers, 4- and 8-byte floating-point numbers, strings, and Boolean values. In addition, it supports the **HRESULT** (OLE error codes), **CURRENCY** (a fixed-point numeric type), and **DATE** (absolute date and time) types, as well as pointers to **IUnknown** and **IDispatch** interfaces.

The **VARIANT** type is encapsulated in the COleVariant class. The supporting **CURRENCY** and **DATE** classes are encapsulated in the COleCurrency and COleDateTime classes.

Automation Samples

- AUTOCLIK Use this sample to learn Automation techniques and as a foundation for learning Remote Automation.
- ACDUAL Adds dual interfaces to an Automation server application.
- CALCDRIV Automation client application driving MFCCALC.
- INPROC Demonstrates an In-Process Automation server application.
- IPDRIVE Automation client application driving INPROC.
- MFCCALC Demonstrates an Automation client application.

What do you want to know more about?

- Automation Clients
- Automation Servers

- Remote Automation
- OLE
- Active Technology

What do you want to do?

- Add an Automation class
- Use type libraries
- Pass parameters in Automation
- Access automation servers
- Write automation clients in C++

See Also

Concepts

MFC COM

Automation Clients

Automation makes it possible for your application to manipulate objects implemented in another application, or to expose objects so they can be manipulated. An Automation client is an application that can manipulate exposed objects belonging to another application. The application that exposes the objects is called the Automation server. The client manipulates the server application's objects by accessing those objects' properties and functions.

Types of Automation Clients

There are two types of Automation clients:

- Clients that dynamically (at run time) acquire information about the properties and operations of the server.
- Clients that possess static information (provided at compile time) that specifies the properties and operations of the server.

Clients of the first kind acquire information about the server's methods and properties by querying the OLE system's **IDispatch** mechanism. Although it is adequate to use for dynamic clients, **IDispatch** is difficult to use for static clients, where the objects being driven must be known at compile time. For static bound clients, the Microsoft Foundation classes provide the COleDispatchDriver class.

Static bound clients use a proxy class that is statically linked with the client application. This class provides a type-safe C++ encapsulation of the server application's properties and operations.

The class **COleDispatchDriver** provides the principal support for the client side of Automation. Using the Add New Item dialog box, you create a class derived from **COleDispatchDriver**.

You then specify the type-library file describing the properties and functions of the server application's object. The Add Item dialog box reads this file and creates the **COleDispatchDriver**-derived class, with member functions that your application can call to access the server application's objects in C++ in a type-safe manner. Additional functionality inherited from **COleDispatchDriver** simplifies the process of calling the proper Automation server.

Handling Events in Automation Clients

If you want to handle events in your automation client, you need to add a sink interface. MFC provides wizard support to add sink interfaces for ActiveX controls, but not support for other COM servers. For information on how to add a sink interface in an MFC client for source interfaces described by COM servers, see HOWTO: Create a Sink Interface in MFC-Based COM Client (KB 181845) at http://support.microsoft.com/default.aspx?scid=kb;en-us;181845.

See Also

Tasks

Automation Clients: Using Type Libraries

Reference

MFC Application Wizard

Concepts

Automation

Automation Clients: Using Type Libraries

Automation clients must have information about server objects' properties and methods if the clients are to manipulate the servers' objects. Properties have data types; methods often return values and accept parameters. The client requires information about the data types of all of these in order to statically bind to the server object type.

This type information can be made known in several ways. The recommended way is to create a type library.

For information on MkTypLib, see the *Platform SDK*.

Visual C++ can read a type-library file and create a dispatch class derived from COleDispatchDriver. An object of that class has properties and operations duplicating those of the server object. Your application calls this object's properties and operations, and functionality inherited from **COleDispatchDriver** routes these calls to the OLE system, which in turn routes them to the server object.

Visual C++ automatically maintains this type-library file for you if you chose to include Automation when the project was created. As part of each build, the .tlb file will be built with MkTypLib.

To create a dispatch class from a type-library (.tlb) file

1. In either Class View or Solution Explorer, right-click the project and click **Add** and then click **Add Class** on the shortcut menu.

2. In the **Add Class** dialog box, select the **Visual C++/MFC** folder in the left pane. Select the **MFC Class From TypeLib** icon from the right pane and click **Open**.

3. In the **Add Class From Typelib Wizard** dialog box, select a type library from the **Available type libraries** drop-down list. The **Interfaces** box displays the interfaces available for the selected type library.

Note
You can select interfaces from more than one type library.

   To select interfaces, double-click them or click the **Add** button. When you do so, names for the dispatch classes will appear in the **Generated classes** box. You can edit the class names in the **Class** box.

   The **File** box displays the file in which the class will be declared. (you can edit this file name as well). You can also use the browse button to select other files, if you prefer to have the header and implementation information written in existing files or in a directory other than the project directory.

Note
All the dispatch classes for the selected interfaces will be put into the file specified here. If you want the interfaces to be declared in separate headers, you must run this wizard for each header file you want to create.

Note
Some type library information may be stored in files with .DLL, .OCX, or .OLB file extensions.

4. Click **Finish**.

   The wizard will then write the code for your dispatch classes using the specified class and file names.

See Also

Concepts

Automation Clients

Automation Servers

Automation makes it possible for your application to manipulate objects implemented in another application, or to expose objects so they can be manipulated. An Automation server is an application that exposes programmable objects (called Automation objects) to other applications (called Automation clients). Automation servers are sometimes called Automation components.

Exposing Automation objects enables clients to automate certain procedures by directly accessing the objects and functionality the server makes available. Exposing objects this way is beneficial when applications provide functionality that is useful for other applications. For example, a word processor might expose its spell-checking functionality so that other programs can use it. Exposure of objects thus enables vendors to improve their applications' functionality by using the ready-made functionality of other applications.

These Automation objects have properties and methods as their external interface. Properties are named attributes of the Automation object. Properties are like the data members of a C++ class. Methods are functions that work on Automation objects. Methods are like the public member functions of a C++ class.

Note
Although properties are like C++ data members, they are not directly accessible. To provide transparent access, set up an internal variable in the Automation object with a pair of get/set member functions to access them.

By exposing application functionality through a common, well-defined interface, Automation makes it possible to build applications in a single general programming language like Microsoft Visual Basic instead of in diverse, application-specific macro languages.

Support for Automation Servers

Visual C++ and the MFC framework provide extensive support for Automation servers. They handle much of the overhead involved in making an Automation server, so you can focus your efforts on the functionality of your application.

The framework's principal mechanism for supporting Automation is the dispatch map, a set of macros that expands into the declarations and calls needed to expose methods and properties for OLE. A typical dispatch map looks like this:

```
BEGIN_DISPATCH_MAP(CMyServerDoc, COleServerDoc)
    DISP_PROPERTY(CMyServerDoc, "Msg", m_strMsg, VT_BSTR)
    DISP_FUNCTION(CMyServerDoc, "SetDirty", SetDirty, VT_EMPTY, VTS_I4)
END_DISPATCH_MAP()
```

The Properties window and Class View assist in maintaining dispatch maps. When you add a new method or property to a class, Visual C++ adds a corresponding `DISP_FUNCTION` or `DISP_PROPERTY` macro with parameters indicating the class name, external and internal names of the method or property, and data types.

The **Add Class** dialog box also simplifies the declaration of Automation classes and the management of their properties and operations. When you use the Add Class dialog box to add a class to your project, you specify its base class. If the base class allows Automation, the Add Class dialog box displays controls you use to specify whether the new class should support Automation, whether it is "OLE creatable" (that is, whether objects of the class can be created on a request from a COM client), and the external name for the COM client to use.

The **Add Class** dialog box then creates a class declaration, including the appropriate macros for the OLE features you have specified. It also adds the skeleton code for implementation of your class's member functions.

The MFC Application Wizard simplifies the steps involved in getting your automation server application off the ground. If you select the **Automation** check box from the **Advanced Features** page, the MFC Application Wizard adds to your application's **InitInstance** function the calls required to register your Automation objects and run your application as an Automation server.

What do you want to do?

- Learn about Automation clients
- Learn more about class CCmdTarget
- Learn more about class COleDispatchDriver

See Also

Reference

MFC Application Wizard

Concepts

Automation

Automation Servers: Object-Lifetime Issues

When an Automation client creates or activates an OLE item, the server passes the client a pointer to that object. The client establishes a reference to the object through a call to the OLE function IUnknown::AddRef. This reference is in effect until the client calls IUnknown::Release. (Client applications written with the Microsoft Foundation Class Library's OLE classes need not make these calls; the framework does so.) The OLE system and the server itself may establish references to the object. A server should not destroy an object as long as external references to the object remain in effect.

The framework maintains an internal count of the number of references to any server object derived from CCmdTarget. This count is updated when an Automation client or other entity adds or releases a reference to the object.

When the reference count becomes 0, the framework calls the virtual function CCmdTarget::OnFinalRelease. The default implementation of this function calls the **delete** operator to delete this object.

The Microsoft Foundation Class Library provides additional facilities for controlling application behavior when external clients have references to the application's objects. Besides maintaining a count of references to each object, servers maintain a global count of active objects. The global functions AfxOleLockApp and AfxOleUnlockApp update the application's count of active objects. If this count is nonzero, the application does not terminate when the user chooses Close from the system menu or Exit from the File menu. Instead, the application's main window is hidden (but not destroyed) until all pending client requests have been completed. Typically, **AfxOleLockApp** and **AfxOleUnlockApp** are called in the constructors and destructors, respectively, of classes that support Automation.

Sometimes circumstances force the server to terminate while a client still has a reference to an object. For example, a resource on which the server depends may become unavailable, causing the server to encounter an error. The user may also close a server document that contains objects to which other applications have references.

In the *Platform SDK*, see **IUnknown::AddRef** and **IUnknown::Release**.

See Also

Reference

AfxOleCanExitApp

Concepts

Automation Servers

Remote Automation

Note
It is recommended that Visual C++ .NET developers use DCOM rather than Remote Automation for new applications. Visual C++ .NET does not support Windows 95. Cases in which you would need to support Remote Automation are described in Where Does Remote Automation Fit In?.

Remote Automation is a type of Automation that allows an interface consumer to execute an interface provider that resides on another machine, for example, on a network.

This article explains how to create Automation objects that can be invoked and executed remotely, and how to create Automation controllers that can use these Remote Automation objects. It also examines configuration options and points out the major differences between Remote Automation and DCOM (the distributed version of COM and OLE that allows interfaces other than those related to automation to be invoked and executed remotely).

In This Section

History of DCOM (Distributed Component Object Model)

Where Does Remote Automation Fit In?

What Does Remote Automation Provide?

Remote Automation Security

Threading Models

Installation

The Automation Manager

- The Remote Automation Connection Manager
- Remote Automation User Components

Creating Programs That Use Remote Automation

Running Remote Automation Using AUTOCLIK and AUTODRIV

See Also

Concepts

MFC COM

Automation

History of DCOM

When Automation was first introduced in early 1993, it was capable of being used only between applications running on the same machine. However, because it shared the same underpinnings as the rest of OLE, that is, COM (or Component Object Model), it was always intended that it would become "remotable" when COM itself was updated to include remoting capabilities. It was also planned that the transition from purely local operation to distributed operation would require little or no change to existing code.

So what does "remoting" mean? Local COM dictated that the consumer of an interface reside and execute on the same machine as the provider of that interface. For example, Microsoft Visual Basic could control a copy of Microsoft Excel on your desktop machine, but it was not capable of directing the execution of Excel on another machine. With the development of distributed COM, the consumer of an interface no longer needs to reside on the same machine as that on which the interface provider executes.

Once COM was adapted to work across a network, then any interface that was not tied to a local execution model (some interfaces have inherent reliance on local machine facilities, such as those drawing interfaces whose methods have handles to device contexts as parameters) would have the capability of being distributed. An interface consumer would make a request for a given interface; that interface may be provided by an instance of an object running (or to be run) on a different machine. The distribution mechanism inside COM would connect the consumer to the provider in such a way that method calls made by the consumer would appear at the provider end, where they would be executed. Any return values would then be sent back to the consumer. To all intents and purposes, the act of distribution is transparent to both the consumer and the provider.

Such a variety of COM does now exist. DCOM (for "distributed COM") has shipped with versions of Windows NT beginning with version 4.0 and including Windows 2000. Since late 1996, it has also been available for Windows 9x. In both cases, DCOM comprises a set of replacement and additional DLLs, with some utilities, which provide both local and remote COM capabilities. It is therefore now an inherent part of Win32-based platforms, and will be made available on other platforms by other organizations over time.

In This Section

Where Does Remote Automation Fit In?

What Does Remote Automation Provide?

See Also

Concepts

Remote Automation

Where Does Remote Automation Fit In?

DCOM was released in 1996 and is available with 32-bit and 64-bit platforms only. The Visual Basic team at Microsoft has always seen Visual Basic as using Automation to allow its components to communicate. The lack of a distributed version severely limited the use of these capabilities in enterprise environments, so the team developing Visual Basic 4.0 Enterprise Edition decided to investigate the creation of its own set of remoting components for the Automation parts of OLE and COM. Clearly, a major goal was to ensure that the result would be compatible with and could be replaced by DCOM when it became available. They then proceeded to implement Remote Automation (RA) for both 16-bit and 32-bit Windows platforms.

Remote Automation is not tied to any specific language, but until the release of Visual C++ 4.2 Enterprise Edition, it was shipped only with Visual Basic 4.0. Be aware that Remote Automation is wholly subsumed by DCOM. If you have the opportunity to use DCOM instead of Remote Automation in your applications, you should do so. Nevertheless, there are scenarios where Remote Automation is more appropriate:

- Wherever you have 16-bit clients.
- If your organization has not rolled out a DCOM-enabled version of Windows NT or Windows 95 yet.
- If you are upgrading an existing application suite that uses Remote Automation to use C++ components in place of one or more Visual Basic components.

There need be no difference between programs created to use Remote Automation and those created to use Automation over DCOM, and the configuration utilities make it very simple to switch operation between Remote Automation and DCOM. Consequently, it is not difficult to upgrade an application from Remote Automation to DCOM once the infrastructure is in place.

See Also

Concepts

What Does Remote Automation Provide?
History of DCOM

What Does Remote Automation Provide?

Remote Automation allows programs to invoke **IDispatch** implementations on one machine from another. It also supports other interfaces required by Automation, specifically **IEnumVARIANT** for collection support. It does not provide the ability to distribute any other COM interface (except **IUnknown**, of course) and, like regular Automation, it contains marshaling support only for those data types supported by Automation.

This set of facilities allows a program to access the methods and properties, including those that return collections or further automation objects, of an object running on an accessible network node. If the client machine is also running the appropriate software, it is possible for the server to call back to the client, again using Automation facilities (this works for 32-bit and 64-bit clients only, and is conceptually similar to events, although it does not use the same mechanism).

For an application to be operable as a Remote Automation server, it must be implemented as an executable (that is, as a "local server" rather than as an "inproc server").

See Also

Concepts

Where Does Remote Automation Fit In?
History of DCOM

Security in Remote Automation

Remote Automation supports a basic level of security to allow a server application writer (or, rather, its administrator) to specify how a specific object may be activated remotely. All automation objects on a given system may be globally set to "disallow remote activation" or to "allow remote activation". Additionally, and more often, individual objects may be given such capabilities. Remote Automation uses a key in each object's registry settings, **AllowRemoteActivation**, to determine whether a given server may be activated remotely. If the systemwide settings use this mode, then each object in the registry may be assigned this key, and the individual status of each one may be set to "yes" or "no" as appropriate.

If the server system is running Windows NT or Windows 2000, then an alternative form of security is allowed. In this case, Remote Automation uses the NT access control list (ACL) to specify which users or group or groups of users may remotely activate a given server.

Note that the security options apply to the whole object; it is not possible to set attributes of a specific interface, or of individual properties or methods on that object.

All security options may be set through the Remote Automation Connection (RAC) Manager.

See Also

Concepts

Remote Automation

Remote Automation Threading Models

Remote Automation supports both the single-threaded and the apartment-threaded models of COM. On 32-bit and 64-bit platforms, which are all that Visual C++ supports, the threading model should be set to "apartment." For more information on how and where to do this, see Creating Programs That Use Remote Automation.

See Also

Concepts

Remote Automation

Remote Automation Installation

Remote Automation has relatively few components:

- The Remote Automation client proxy, AUTPRX32.DLL.
- The Remote Automation server-side component, the Automation Manager, AUTMGR32.EXE.
- The RAC Manager, RACMGR32.EXE, with its matching RACREG32.DLL.

Of these, RAC Manager is written in Visual Basic and therefore needs the Visual Basic run-time support. These and the other Remote Automation files are installed by Setup when you install Visual C++ Enterprise Edition.

If you copy the Remote Automation components to a computer on which Visual C++ version Enterprise Edition is not installed, ensure that REGSRV32.EXE is on the computer's path, and register RACREG32.DLL using the following command line:

REGSRVR32 RACREG32.DLL

Note
Versions of RAC Manager before Visual C++ 5.0 required GUAGE32.OCX and TABCTL32.OCX. Neither of these is required fo r the version of RAC Manager that ships with Visual C++ Enterprise Edition, version 5.0 or later.

In This Section

The Automation Manager

The Remote Automation Connection Manager

Remote Automation User Components

See Also

Concepts

Remote Automation

The Automation Manager

AUTMGR32.EXE should be copied to the Windows system directory of each machine that is intending to provide Remote Automation objects. For Windows 95 and Windows 98, this directory is typically C:\WINDOWS\SYSTEM. For Windows NT and Windows 2000, it is typically C:\WINNT\SYSTEM32.

If you want to enable callbacks from the server to the client, this executable file should also be copied to the system directory of each client machine.

When the Automation Manager is running, it displays a small window on the server machine that contains brief status information. If you want to hide it, refer to article Q138067 in the Microsoft Knowledge Base.

See Also

Concepts

The Remote Automation Connection Manager
Remote Automation User Components
Remote Automation Installation

The Remote Automation Connection Manager

To configure both client and server machines, you need to make registry changes. Rather than doing this by hand, it is far easier to use the Remote Automation Connection (RAC) Manager tool. This tool, RACMGR32.EXE, along with RACREG32.DLL, needs to be copied to any directory you choose. By putting it in the PATH, it can be executed from the taskbar using Run. Alternatively, you can create a shortcut to it or place a reference to it on the Start menu.

RACMGR32 is written in Visual Basic and therefore needs some Visual Basic support DLLs. These files are placed in the same directory as RACMGR32.EXE on the CD-ROM. The versions of these files that are installed by the Setup for Visual C++ Enterprise Edition are equivalent or more recent than those that shipped with Visual Basic Enterprise Edition 5.0. The Visual C++ Setup copies the new versions of the Visual Basic files to your system directory. For Windows 9x, this directory is typically C:\Windows\System. For Windows NT and Windows 2000, it is typically C:\WINNT\system32. Setup also registers these files with the operating system. You may remove them from your Visual Basic installation.

See Also

Concepts

The Automation Manager
Remote Automation User Components
Remote Automation Installation

Remote Automation User Components

You will need to ensure that each client machine contains your client program and any support DLLs it requires. You will also need to ensure that the server application and any support DLLs it requires are present on the server machine. Finally, you will need to ensure that your server program is registered on each client machine before RAC Manager can be run to configure the connection. If the program is self-registering (as most will be), you need only execute the server program on the client machine to register it. Failing that, you may have to execute a registration file that you provide, or manually edit the registry.

See Also

Concepts

The Automation Manager
The Remote Automation Connection Manager
Remote Automation Installation

Creating Programs That Use Remote Automation

Any automation object, and any automation controller, is able to use Remote Automation without any change to the source code, without the need for recompilation, and without the need for relinking. Once you have a setup that works locally (that is, on the same machine), you need go through only a few steps to execute it remotely.

To execute the Remote Automation object

1. Register the application on the client machine or machines.
2. Configure the client access to use remote server.
3. Install and register the application on the server machine or machines.
4. Configure the server to allow remote activation.
5. Install the Automation Manager on the server machine(s).
6. Run Automation Manager on the server(s).
7. Run the application on the client(s).

Step 1 is most easily accomplished by loading and executing the server application on the client machine, as most servers are self registering. If you tested the setup locally beforehand, this stage is already complete. If the server application is not self registering, you may want to make it so. Otherwise, you will need to provide a registration file that the local user can run to perform this step. If you do neither of these things, you or an administrator will need to edit the registry manually, a procedure which is not recommended for all but the most advanced users.

Step 2 involves the use of the Remote Automation Connection (RAC) Manager. Run RAC Manager and ensure that the server connection tab is uppermost. Next, find the entry for the server object in the **OLE Classes** pane and click on it. Then move to the **Network Address** combo box and enter the name of the server machine on which the remote executable file will be run. For example, you may enter \\MyServer here. Then choose the appropriate network protocol from the list provided. You may need to check with your network administrator to determine which protocol is recommended. In many cases, this will be TCP/IP. Finally, you may want to choose an authentication level. In most cases, this will be left as (None) or Default. Now right-click the server in the **OLE Classes** pane. This will produce a shortcut menu from which you can select local or remote operation. Select remote.

Step 3 involves properly installing and registering the server application on the selected server machine or machines. Again, if the application is self registering, executing it once will also register it.

Step 4 involves configuring the server to allow remote execution. Run RAC Manager on the server machine, and ensure that the **Client Access** tab has the focus. Choose the activation model that you want (typically **Allow Remote Creates by Key**. If you do choose this option, you also need to click the **Allow Remote Activation** check box to set the value of the registry entry to 'Y'). If you are running Windows NT or Windows 2000 and you choose the Allow Remote Creates (ACL) option, you also have the option to edit the ACL by pushing the **Edit ACL** button.

To allow Remote Automation to work, you then need to ensure that the Automation Manager is installed and running on the server machine or machines. If it is not installed, copy AUTMGR32.EXE to the Windows system directory. For information on how to do so, see Remote Automation Installation. To start Remote Automation, execute the Automation Manager. It will display a small status window in which a number of messages will be shown. Once it has started, it will minimize itself. If you want to continue to see status information, you can click the **Automation Manager** tab in the task bar to restore the window.

The final step is to execute the client application on the client machine. If you have followed the steps above, it should connect to the server object and execute precisely as it did locally, albeit a little slower.

Notice that the RAC Manager also allows you to switch between Remote Automation and distributed COM (DCOM, on those platforms that support DCOM) with a single click of a radio button. If you choose DCOM, you can set various other configuration options. See the DCOM documentation for further details.

See Also

Concepts

Remote Automation

Running Remote Automation Using AUTOCLIK and AUTODRIV

Running Remote Automation Using AUTOCLIK and AUTODRIV

AUTOCLIK is a simple Automation server sample application that you can use as a base from which to learn more about Remote Automation. AUTODRIV is a simple Automation client application that drives AUTOCLIK. You can use them to demonstrate Remote Automation.

To install AUTOCLIK.EXE on two machines and drive it using Remote Automation

1. Install the AUTOCLIK sample application onto your development machine.
2. Build AUTOCLIK.EXE.
3. Run AUTOCLIK.EXE in standalone fashion and then shut it down. This will register it as an Automation server.
4. Copy AUTOCLIK.EXE to a remote machine, run it there, and then shut it down.
5. Run AUTODRIV.EXE on the local machine and verify that running it starts AUTOCLIK.EXE. To find out more about AUTODRIV.EXE, see AUTOCLIK.
6. On the remote machine, start AUTMGR32.EXE (Automation Manager).
7. On the remote machine, start RACMGR32.EXE (Remote Automation Connection Manager).
8. In the Remote Automation Connection Manager, select AutoClik.Document from the **OLE Classes** list.
9. Choose a system security policy from the **Client Access** tab to grant client access to AutoClik.Document.
10. On the local machine, start RACMGR32.EXE and select AutoClik.Document from the **OLE Classes** list.
11. From the **Server Connection** tab, choose both the network address of the remote machine and the appropriate network protocol.
12. With AutoClik.Document still selected in the **OLE Classes** list box, choose the **Remote** command from the **Register** menu.
13. On the local machine, run AUTODRIV.EXE or the equivalent AUTOCLIK.MAK Visual Basic project if you want to have a Visual Basic, rather than an MFC, client.

On the remote machine, you should now be able to see AUTOCLIK executing commands initiated from the local client.

See Also

Concepts

Remote Automation

Connection Points

This article explains how to implement connection points (formerly known as OLE connection points) using the MFC classes **CCmdTarget** and **CConnectionPoint**.

In the past, the Component Object Model (COM) defined a general mechanism (**IUnknown::QueryInterface**) that allowed objects to implement and expose functionality in interfaces. However, a corresponding mechanism that allowed objects to expose their capability to call specific interfaces was not defined. That is, COM defined how incoming pointers to objects (pointers to that object's interfaces) were handled, but it did not have an explicit model for outgoing interfaces (pointers the object holds to other objects' interfaces). COM now has a model, called connection points, that supports this functionality.

A connection has two parts: the object calling the interface, called the source, and the object implementing the interface, called the sink. A connection point is the interface exposed by the source. By exposing a connection point, a source allows sinks to establish connections to itself (the source). Through the connection point mechanism (the **IConnectionPoint** interface), a pointer to the sink interface is passed to the source object. This pointer provides the source with access to the sink's implementation of a set of member functions. For example, to fire an event implemented by the sink, the source can call the appropriate method of the sink's implementation. The following figure demonstrates the connection point just described.

An Implemented Connection Point

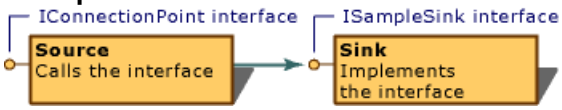

MFC implements this model in the CConnectionPoint and CCmdTarget classes. Classes derived from **CConnectionPoint** implement the **IConnectionPoint** interface, used to expose connection points to other objects. Classes derived from **CCmdTarget** implement the **IConnectionPointContainer** interface, which can enumerate all of an object's available connection points or find a specific connection point.

For each connection point implemented in your class, you must declare a connection part that implements the connection point. If you implement one or more connection points, you must also declare a single connection map in your class. A connection map is a table of connection points supported by the ActiveX control.

The following examples demonstrate a simple connection map and one connection point. The first example declares the connection map and point; the second example implements the map and point. Note that `CMyClass` must be a **CCmdTarget**-derived class. In the first example, code is inserted in the class declaration, under the **protected** section:

```
class CMyClass : public CCmdTarget
{
 ...
 protected:
// Connection point for ISample interface
    BEGIN_CONNECTION_PART(CMyClass, SampleConnPt)
        CONNECTION_IID(IID_ISampleSink)
    END_CONNECTION_PART(SampleConnPt)

    DECLARE_CONNECTION_MAP()

};
```

The **BEGIN_CONNECTION_PART** and **END_CONNECTION_PART** macros declare an embedded class, `XSampleConnPt` (derived from **CConnectionPoint**), that implements this particular connection point. If you want to override any **CConnectionPoint** member functions or add member functions of your own, declare them between these two macros. For example, the **CONNECTION_IID** macro overrides the **CConnectionPoint::GetIID** member function when placed between these two macros.

In the second example, code is inserted in the control's implementation file (.cpp file). This code implements the connection map, which includes the connection point, `SampleConnPt`:

```
BEGIN_CONNECTION_MAP(CMyClass, CMyBaseClass)
    CONNECTION_PART(CMyClass, IID_ISampleSink, SampleConnPt)
END_CONNECTION_MAP()
```

If your class has more than one connection point, insert additional **CONNECTION_PART** macros between the

BEGIN_CONNECTION_MAP and **END_CONNECTION_MAP** macros.

Finally, add a call to **EnableConnections** in the class's constructor. For example:

```
CMyClass::CMyClass()
{
EnableConnections();
...
}
```

Once this code has been inserted, your **CCmdTarget**-derived class exposes a connection point for the **ISampleSink** interface. The following figure illustrates this example.

A Connection Point Implemented with MFC

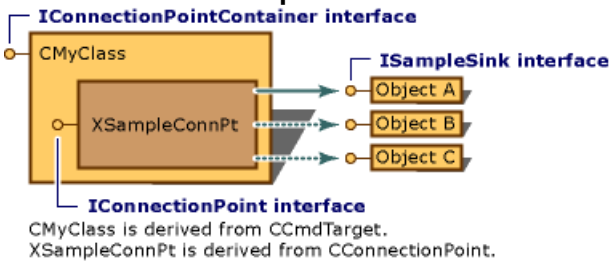

Usually, connection points support "multicasting" — the ability to broadcast to multiple sinks connected to the same interface. The following example fragment demonstrates how to multicast by iterating through each sink on a connection point:

```
void CMyClass::CallSinkFunc()
{
    const CPtrArray* pConnections = m_xSampleConnPt.GetConnections();
    ASSERT(pConnections != NULL);

    int cConnections = pConnections->GetSize();
    ISampleSink* pSampleSink;
    for (int i = 0; i < cConnections; i++)
    {
        pSampleSink = (ISampleSink*)(pConnections->GetAt(i));
        if(pSampleSink != NULL)
            pSampleSink->SinkFunc();
    }
}
```

This example retrieves the current set of connections on the `SampleConnPt` connection point with a call to **CConnectionPoint::GetConnections**. It then iterates through the connections and calls **ISampleSink::SinkFunc** on every active connection.

See Also

Concepts

MFC COM

MFC Internet Programming

For information on Internet programming, see the following sections:

For information on	See
Overview of MFC Internet programming	MFC Internet Programming Basics
Basic tasks for MFC Internet programming	MFC Internet Programming Tasks
Win32 Internet Extensions (WinInet)	Win32 Internet Extensions (WinInet)
Internet Server API (ISAPI) Extensions	Internet Server API (ISAPI) Extensions
ATL Server	ATL Server

See Also

Other Resources

General Concepts

MFC Internet Programming Basics

Microsoft provides many APIs for programming both client and server applications. Many new applications are being written for the Internet, and as technologies, browser capabilities, and security options change, new types of applications will be written. Browsers run on client computers, providing access to the World Wide Web and displaying HTML pages that contain text, graphics, ActiveX controls, and documents. Servers provide FTP, HTTP, and gopher services, and run server extension applications using CGI and ISAPI. Your custom application can retrieve information and provide data on the Internet.

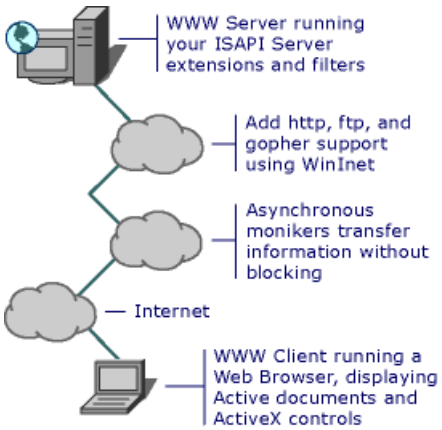

MFC provides classes that support Internet programming. You can use COleControl and CDocObjectServer and related MFC classes to write ActiveX controls and Active documents. You can use MFC classes such as CInternetSession, CFtpConnection, and CAsyncMonikerFile to retrieve files and information using Internet protocols such as FTP, HTTP, and gopher. You can use CHttpServer and CHttpFilter and related MFC ISAPI classes to extend Web servers with custom applications and filters.

In This Section

- Internet-Related MFC Classes
- Internet Information by Topic
- Internet Information by Task
- Active Technology on the Internet
- ISAPI Server Extensions and Filters
- WinInet Basics
- HTML Basics
- HTTP Basics
- Internet Programming Frequently Asked Questions

Related Sections

- ActiveX Controls on the Internet
- Active Documents on the Internet
- Asynchronous Monikers on the Internet
- Win32 Internet Extensions (WinInet)
- Internet Server API (ISAPI) Extensions
- MFC Internet Programming Tasks
- Application Design Choices
- Writing MFC Applications
- Testing Internet Applications
- Internet Security

- ATL Support for DHTML Controls

Web Sites for More Information

For additional information on Microsoft's Internet technology, see the Microsoft Developer Network Online Web site at http://msdn.microsoft.com/ (web links may change without notice).

This Web site for developers contains information on using Microsoft development tools and technologies, and top stories about recent and upcoming conferences. From this page, you can jump to many related developer sites, including the .NET, and XML Developer Centers. You can also download beta SDKs and samples.

The World Wide Web Consortium (W3C) publishes specifications for HTML, HTTP, CGI, and other World Wide Web technologies. You can visit them at http://www.w3.org/.

Note
Web links may change without notice.

More Internet Help

The OLE section of the *Platform SDK* contains additional information about OLE programming. This information provides details about using the Win32 WinInet and ISAPI functions directly, rather than through the MFC classes. It also contains overview information about Internet technologies.

See Also

Other Resources

MFC Internet Programming

Internet-Related MFC Classes

For information about Internet-related classes and functions, see:

Global functions

- AfxParseURL
- AfxGetInternetHandleType

ActiveX control classes

- COleControl

Active document classes

- CDocObjectServer
- CDocObjectServerItem

Asynchronous moniker classes

- CAsyncMonikerFile
- CDataPathProperty

ISAPI classes

- CHttpServer
- CHttpServerContext
- CHttpFilter
- CHttpFilterContext
- CHtmlStream

WinInet classes

- CInternetSession
- CInternetConnection
- CFtpConnection
- CGopherConnection
- CHttpConnection
- CInternetFile
- CGopherFile
- CHttpFile
- CFileFind
- CFtpFileFind
- CGopherFileFind
- CGopherLocator
- CInternetException

See Also

Concepts

MFC Internet Programming Basics

Internet Information by Topic

For information on programming with a specific Internet technology, see:

Internet Server API (ISAPI)

Overviews	ISAPI Server Extensions and Filters ISAPI Server Extensions and Filters Internet Server API (ISAPI) Extensions MFC ISAPI Classes Extending Web Servers with ISAPI
Filters	ISAPI Filters Creating a Typical ISAPI Filter
Servers	ISAPI Server Extensions Parse Maps Creating a Typical ISAPI Extension
Databases	ISAPI and Databases
Debugging	Debugging ISAPI Applications
Designing	ISAPI Programming Tips

WinInet

Overviews	WinInet (HTTP, FTP, Gopher) WinInet Basics Win32 Internet Extensions (WinInet) How WinInet Makes It Easier to Create Internet Client Applications How MFC Makes It Easier to Create Internet Client Applications
Steps in creating WinInet applications	Prerequisites for Internet Client Classes Writing an Internet Client Application Using MFC WinInet Classes Steps in a Typical Internet Client Application Steps in a Typical FTP Client Application Steps in a Typical HTTP Client Application Steps in a Typical Gopher Client Application Steps in a Typical HTTP Client Application

See Also

Concepts

MFC Internet Programming Basics

Internet Information by Task

The tasks listed in this topic cover the use of Internet Server API (ISAPI) extension DLLs, ISAPI filters, and parse maps.

The following categories of tasks are listed in this topic:

- ActiveX Controls, Documents and Asynchronous Moniker Tasks
- Internet Server Application (ISA) Tasks
- ISAPI Filter Tasks
- ISAPI Parse Map Tasks
- Other ISAPI Tasks
- WinInet Tasks

What do you want to do?

ActiveX Controls, Documents and Asynchronous Moniker Tasks

- Learn about Active documents
- Learn about Asynchronous Monikers
- Learn about ActiveX controls in the Internet context
- Learn about Active documents in the Internet context
- Optimize an ActiveX control
- Create Signed CAB Files for MFC and ATL Controls

Internet Server Application (ISA) Tasks

- Write an ISA

ISAPI Filter Tasks

- Filter incoming or outgoing data for an Internet Server
- Register an ISAPI filter application for desired events
- Process ISAPI filter notifications

ISAPI Parse Map Tasks

- Create a parse map for an ISAPI extension
- Use a parse map with ISAPI forms
- Declare a command in a parse map
- Pass a string from the Universal Resource Locator (URL)
- Use named parameters in a parse map
- Manage parse map parameters

Other ISAPI Tasks

- Learn about ISAPI server extensions
- Create an ISAPI server filter
- Create an ISAPI server extension
- Learn about ISAPI and databases
- Learn about ISAPI debugging

- Get some ISAPI programming tips

WinInet Tasks

- Learn about WinInet, the Win32 API functions for Internet access
- Review what's involved with WinInet programming
- Write an Internet client application, using MFC WinInet classes
- Write an FTP client application
- Write an HTTP client application
- Write a Gopher client application

See Also

Concepts

MFC Internet Programming Basics

Active Technology on the Internet

Active technology is an open platform that lets developers create exciting, dynamic content and applications for the global Internet, or for a company's internal network, known as an intranet. The major technologies provided by Microsoft for Internet programming are described below.

ActiveX Controls

ActiveX controls (formerly OLE controls) are objects that can be inserted into Web pages or any other application that is an ActiveX control container. Examples include buttons, stock tickers, and chart controls. For more information, see ActiveX Controls on the Internet.

Active Documents

Active documents can be displayed by Web browsers or document viewers. Traditional embedded objects were limited to one page and were shown embedded in the document. With Active technology, the document can be displayed full frame in the entire client area window. For more information, see Active Documents on the Internet.

Active Server Framework

You can extend a Web server to provide customized Web pages, using content from a database or customized in other ways by an application that runs on the server. The MFC ISAPI classes provide an easy way to write custom server extensions and filters.

Internet Data Download Services

Data can be downloaded over the Internet using common protocols: HTTP, FTP, and gopher. The MFC WinInet classes make it easy to transfer data using HTTP, FTP, and gopher protocols by abstracting the TCP/IP and WinSock protocols. The MFC asynchronous moniker classes provide a way to download files without blocking and to render large objects asynchronously. For more information, see Win32 Internet Extensions (WinInet).

Active Scripts

VBScript and other scripting languages connect controls and add interactive functionality to Web pages. Scripting moves processing from the server to the client. For example, form entries can be validated on the client and then sent to the server. For more information, see How Do I Script an ActiveX Control on a Web Page?

HTML Extensions

HTML extensions, such as the object tag, have been added to support controls and scripting.

See Also

Reference

CHttpServer Class
CHttpFilter Class
CHtmlStream Class

Concepts

MFC Internet Programming Basics
ActiveX Controls on the Internet
Active Documents on the Internet
Win32 Internet Extensions (WinInet)
Internet Server API (ISAPI) Extensions

ISAPI Server Extensions and Filters

Note
ATL Server provides a more efficient, configurable way of writing Web applications and XML Web services using C++ and IS API. For more information see the documentation on ATL Server.

If you already have an application and want to run it on a Web server, there are several tasks to complete:

- Remove any dialog boxes and other user-interface features from the DLL that will run on the server.
- When running on a Web server, you provide content to the client through HTML. Design the HTML pages you will use on the client to collect information that is sent to the server, and to return results to the client.
- Reduce code size and remove unnecessary functions to increase processing efficiency.
- If you are writing a database front-end application using simple queries, you may be able to use DB Connector (which ships as part of Internet Information Services) for your database processing. See the documentation that comes with Internet Information Services.

  If you want to use multiple queries on a page or use advanced features of SQL Server, you will need to write an ISAPI DLL to provide special forms processing and to access the full capabilities of SQL Server.

For more information about ISAPI, see:

- Internet Server API (ISAPI) Extensions
- Extending Web Servers with ISAPI
- Creating a Typical ISAPI Extension
- Creating a Typical ISAPI Filter
- ATL Server

See Also

Concepts

MFC Internet Programming Basics
Internet Server API (ISAPI) Extensions

WinInet Basics

You can use WinInet to add FTP support to download and upload files from within your application. You can override OnStatusCallback and use the *dwContext* parameter to provide progress information to users as you search for and download files.

This article contains the following topics:

- Create a Very Simple Browser
- Download a Web Page
- FTP a File
- Retrieve a Gopher Directory
- Display Progress Information While Transferring Files

The code excerpts below demonstrate how to create a simple browser, download a Web page, FTP a file, and search for a gopher file. They are not meant as complete examples and not all contain exception handling.

For additional information on WinInet, see Win32 Internet Extensions (WinInet).

Create a Very Simple Browser

```
#include <afxinet.h>
//assumes URL names have been initialized
CInternetSession session("My Session");
CStdioFile* pFile = NULL;
//use a URL and display a Web page
while (lpszURL = DisplayPage(...))
{
   pFile = session.OpenURL(lpszURL);
   while (pFile->Read(szBuff,1024) > 0)
   {
      //read file...
   }
   delete pFile;
}
session.Close();
```

Download a Web Page

```
//this code excerpt also demonstrates try/catch exception handling
#include <afxinet.h>
//assumes server, port, and URL names have been initialized
CInternetSession session("My Session");
CHttpConnection* pServer = NULL;
CHttpFile* pFile = NULL;
try
{
   CString strServerName;
   INTERNET_PORT nPort;

   pServer = session.GetHttpConnection(strServerName, nPort);
   pFile = pServer->OpenRequest(CHttpConnection::HTTP_VERB_GET, strObject);
   pFile->AddRequestHeaders(szHeaders);
   pFile->SendRequest();
   pFile->QueryInfoStatusCode(dwRet);

   if (dwRet == HTTP_STATUS_OK)
   {
       UINT nRead = pFile->Read(szBuff, 1023);
       while (nRead > 0)
       {
           //read file...
```

```
        }
    }
    delete pFile;
    delete pServer;
}
catch (CInternetException* pEx)
{
    //catch errors from WinInet
}
session.Close();
```

FTP a File

```
#include <afxinet.h>
//assumes server and file names have been initialized
CInternetSession session("My FTP Session");
CFtpConnection* pConn = NULL;

pConn = session.GetFtpConnection(lpszServerName);
//get the file
if (!pConn->GetFile(pstrRemoteFile, pstrLocalFile))
    //display an error
delete pConn;
session.Close();
```

Retrieve a Gopher Directory

```
#include <afxinet.h>
//assumes file name has been initialized
CInternetSession session("My Gopher Session");
CGopherConnection* pConn = NULL;
CGopherFileFind* pFile;

pConn = session.GetGopherConnection("gopher.yoursite.com");
pFile = new CGopherFileFind(pConn);
BOOL bFound = pFile->FindFile(lpszFileToFind);
while (bFound)
{
    bFound = pFile->FindNextFile();
    //retrieve attributes of found file
}
delete pFile;
delete pConn;
session.Close();
```

Use OnStatusCallback

When using the WinInet classes, you can use the OnStatusCallback member of your application's CInternetSession object to retrieve status information. If you derive your own **CInternetSession** object, override **OnStatusCallback**, and enable status callbacks, MFC will call your **OnStatusCallback** function with progress information about all the activity in that Internet session.

Because a single session might support several connections (which, over their lifetime, might perform many different distinct operations), **OnStatusCallback** needs a mechanism to identify each status change with a particular connection or transaction. That mechanism is provided by the context ID parameter given to many of the member functions in the WinInet support classes. This parameter is always of type **DWORD** and is always named *dwContext*.

The context assigned to a particular Internet object is used only to identify the activity the object causes in the **OnStatusCallback** member of the **CInternetSession** object. The call to **OnStatusCallback** receives several parameters; these parameters work together to tell your application what progress has been made for which transaction and connection.

When you create a **CInternetSession** object, you can specify a *dwContext* parameter to the constructor. **CInternetSession** itself doesn't use the context ID; instead, it passes the context ID on to any **InternetConnection**-derived objects that don't explicitly get a context ID of their own. In turn, those **CInternetConnection** objects will pass the context ID along to **CInternetFile** objects they create if you don't explicitly specify a different context ID. If, on the other hand, you do specify a specific context ID of your own, the object and any work it does will be associated with that context ID. You can use the context

IDs to identify what status information is being given to you in your **OnStatusCallback** function.

Display Progress Information While Transferring Files

For example, if you write an application that creates a connection with an FTP server to read a file and also connects to an HTTP server to get a Web page, you'll have a **CInternetSession** object, two **CInternetConnection** objects (one would be a **CFtpSession** and the other would be a **CHttpSession**), and two **CInternetFile** objects (one for each connection). If you used default values for the *dwContext* parameters, you would not be able to distinguish between the **OnStatusCallback** invocations that indicate progress for the FTP connection and the invocations that indicate progress for the HTTP connection. If you specify a *dwContext* ID, which you can later test for in **OnStatusCallback**, you will know which operation generated the callback.

See Also

Concepts

MFC Internet Programming Basics
Win32 Internet Extensions (WinInet)

HTML Basics

Most browsers have the capability of examining the HTML source of the pages you browse. When you view the source you will see a number of HTML (Hypertext markup language) tags, surrounded by angle brackets(<>), interspersed with text.

The steps below use HTML tags to build a simple Web page. In these steps, you'll type plain text into a file in Notepad, make a few changes, save the file, and reload your page in the browser to see your changes.

To create an HTML file

1. Open Notepad or any plain text editor.
2. From the **File** menu, choose **New**.
3. Type the following lines:

```
<HTML>
<HEAD>
<TITLE>Top HTML Tags</TITLE>
</HEAD>
</HTML>
```

4. From the **File** menu, choose **Save**, and save the file as c:\webpages\First.htm. Leave the file open in the editor.
5. Switch to your browser, and from the **File** menu, choose **Open**, or type `file://C:/webpages/first.htm` in the browser's URL edit box. You should see a blank page with the window caption "Top HTML Tags."

   Notice the tags are paired and are included in angle brackets. Tags are not case-sensitive, but capitalization is often used to make tags stand out.

   The tag <HTML> starts the document, and the tag </HTML> ends it. Ending tags (not always required) are the same as the starting tag, but have a forward slash (/) in front of the tag. There should be no spaces between the angle bracket (<) and the start of your tag.
6. Switch back to Notepad, and after the </HEAD> line, type:

```
<BODY>
HTML is swell.
Life is good.
</BODY>
```

7. From the **File** menu, choose **Save**.
8. Switch back to your browser and refresh the page.

   The words will appear in the client area of your browser's window. Notice that your carriage return is ignored. If you want to have a line break, you must include a `<BR>` tag after the first line.

   For all the steps that follow, insert the text anywhere between <BODY> and </BODY> to add to the body of your document.
9. Add a header:

```
<H3>Here's the big picture</H3>
```

10. Add an image, using a .gif file saved in the same directory as your page:

```
<IMG src="yourfile.gif">
```

11. Add a list:

```
<UL>Make me an unordered list.
<LI>One programmer</LI>
<LI>Ten SDKs</LI>
<LI>Great Internet Apps</LI>
</UL>
```

12. To number the list instead, use paired <OL> and </OL> tags in place of the <UL> and </UL> tags.

That should get you started. If you see a great feature on a Web page, you can find out how it was created by examining the HTML source. HTML editors such as Microsoft Front Page can be used to create both simple and advanced pages.

Here's the entire HTML source for the file you've been building:

```
<HTML>
<HEAD>
<TITLE>Top HTML Tags</TITLE>
</HEAD>
<BODY>
HTML is swell.<BR>
Life is good.
<H3>Here's the big picture</H3>
<IMG src="yourfile.gif">
<UL>Make me an unordered list.
<LI>One programmer</LI>
<LI>Ten SDKs</LI>
<LI>Great Internet Apps</LI>
</UL>
</BODY>
</HTML>
```

For a complete description of tags, attributes, and extensions, see the Hypertext Markup Language (HTML) specification:

http://www.w3.org/pub/WWW/MarkUp/

See Also

Concepts

MFC Internet Programming Basics

HTTP Basics

When writing an ISAPI application, you often examine and add to the information in HTTP header. Return codes indicate the success or failure of the requested event. Several common return codes are listed in the following table.

Return Code	Meaning
200	URL located, transmission follows
400	Unintelligible request
404	Requested URL not found
405	Server does not support requested method
500	Unknown server error
503	Service unavailable

Additional return codes can be found in the AFXISAPI.H file. The HTTP responses are grouped as shown in the following table.

Group	Meaning
200–299	Success
300–399	Information
400–499	Request error
500–599	Server error

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for hypermedia information systems. For more information about HTTP, and how Web browsers and servers communicate, see the Hypertext Transfer Protocol (HTTP) specification:

http://www.w3.org/pub/WWW/Protocols/

See Also

Concepts

MFC Internet Programming Basics

Internet Programming Frequently Asked Questions

Here are some frequently asked questions (FAQs) about Internet programming:

- How Are ISAPI Server Extensions and Filters Called?
- What Is the Difference Between an ISAPI Server Extension and a Filter?
- How Does ISAPI Compare with CGI?
- How Do I Have Multiple Filters and Extensions?
- What Are Some Programming Tips for Using ISAPI on IIS?
- What Are Some Memory and Resource Considerations?
- How Do I Use the MFC ISAPI Classes Without Linking to the MFC DLL?
- How Do I Script an ActiveX Control on a Web Page?

See Also

Concepts

MFC Internet Programming Basics

How Are ISAPI Server Extensions and Filters Called?

An Internet server extension (ISA) is a program you write to extend the capabilities of the server. An ISA is called when explicitly invoked by a client sending a URL that invokes your DLL. For example, `http://yourserver/wwwquote.dll?` asks the server named `yourserver` to run the DLL `wwwquote`. An HTTP request can also be sent when a user clicks a button on a Web page. This method is often used when a Web page is a form that collects information, which is passed as parameters to the DLL. For example, `http://yourserver/wwwquote.dll?Issues?Method=ByCUSIP` invokes the DLL and passes the parameter `Method=ByCUSIP` to the function `Issues`.

In contrast, an ISAPI filter is called each time a specified event occurs, no matter what request is being processed by the server. If you included **SF_NOTIFY_URL_MAP**, the flag for URL mapping, when requesting notifications, your **OnURLMap** function will be called each time a URL is mapped to your server, regardless of the content of the URL command. A filter can process, examine, and change data on each request to the server.

See Also

Concepts

Internet Programming Frequently Asked Questions

What Is the Difference Between an ISAPI Server Extension and a Filter?

A server extension:	**A filter:**
Runs when referenced in a URL.	Is called for every URL the server processes.
Is explicitly invoked, for example by `http://myserver/myprog.dll?`.	Runs automatically for any URL sent to the server if the registered event occurs.
Is loaded on demand, the first time a user calls it.	Is loaded when the service starts because of its registry entry.

Both server extensions and filters:

- Share the process space of the service.
- Must be thread safe.
- Once loaded, remain in memory (until the service is stopped or the memory is needed by another process).

See Also

Concepts

Internet Programming Frequently Asked Questions

How Does ISAPI Compare with CGI?

ISAPI server extensions provide an alternative to the use of Common Gateway Interface (CGI) applications for Internet servers. Unlike CGI applications, ISAs run in the same address space as the HTTP server and have access to all the resources available to the HTTP server. ISAs have lower overhead than CGI applications because they do not require the creation of additional processes and do not perform time-consuming communications across process boundaries. Both extension and filter DLLs may be unloaded if the memory is needed by another process.

An Internet client calls an ISA through the HTTP server the same way it would call a CGI application. For example, a client might call a CGI application as:

```
http://sample/example.exe?Param1&Param2
```

It would call an ISA that performs the same function as:

```
http://sample/example.dll?Param1&Param2
```

ISAPI allows multiple commands in one DLL, implemented as member functions of the **CHttpServer** object in the DLL. CGI requires a separate name and URL mapping to a separate executable file for each task. Every new CGI request launches a new process, and each different request is contained in its own executable file, loaded and unloaded on each request, so overhead is higher than for ISAs.

ISAPI filters have no CGI equivalent. Filters provide the capability of pre-processing and post-processing of all data sent between the client and the server.

See Also

Concepts

Internet Programming Frequently Asked Questions

How Do I Have Multiple ISAPI Filters and Extensions?

You can only have one instance of a class derived from **CHttpServer** in a given DLL. You can have multiple DLLs installed, and each server extension DLL can have more than one command.

You can only have one instance of a class derived from **CHttpFilter** in a given DLL. One filter can register to receive multiple notifications. If you want to have a high-priority filter along with other notifications, use separate filters in separate DLLs, since all filters in the same DLL share the same priority.

Can My Filter and Server Extension Both Be in the Same DLL?

You can have one filter (receiving multiple notifications, if desired) and one server extension in the same DLL. This can be useful if you have a filter and a server extension that are related. For example, you may want to perform post-processing of data before sending it on to the client so that you can customize the look for a particular browser. Another instance would be to communicate between your filter and your extension. You can put the filter and application into the same DLL and add a key value header during the filter **SF_NOTIFY_PREPROC_HEADERS** notification. The application can obtain the key by looking up the corresponding context.

See Also

Concepts

Internet Programming Frequently Asked Questions

What Are Some Programming Tips for Using ISAPI on IIS?

This article discusses programming tips that are specific to an ISAPI extension or filter running on Microsoft Internet Information Services (IIS). If you are running on other Web servers that support ISAPI, they may handle registry keys, memory, and loading of applications differently. Check the documentation for the Web server you are using for more information.

As of IIS 1.0, an ISAPI filter is loaded when the service is started and not unloaded until the service stops. An ISAPI server extension is loaded the first time it is invoked and remains in memory until the service is shut down or the memory is needed for another process. Because your DLL will remain in memory for a long time, consider when to allocate resources, when to deallocate resources, and when to connect or disconnect from resources such as a data source.

How Do I Install a New Version of My Filter or Server Extension?

To install a new version of your filter or ISA, you must shut down the service, copy the DLL to the appropriate directory, update the registry if it's a new filter, and restart the service.

When Is My DLL Loaded or Unloaded?

The registry key HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services/W3SVC/Parameters/FilterDLLs controls which filters are loaded when the service starts. It contains a list of filters, including their full paths, separated by commas. To load additional filters, stop the service, add your filter to the registry, and restart the service.

A server extension is loaded when first called by a client.

After the service is started, a filter remains in memory until the computer or the service is shut down. Once loaded, a server extension may also remain in memory until the computer or the service is shut down. You can force reloading of your extension DLL (for example, for debugging purposes) by specifying the registry setting HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services/W3SVC/Parameters/CacheExtensions=0. This setting doesn't take effect until the service is restarted. Note that this setting should be used only while debugging. It has a negative impact on performance, and should never be used on a production server.

See Also

Concepts

Internet Programming Frequently Asked Questions

What Are Some Memory and Resource Considerations?

Must My Application Be Thread-Safe?

Yes. ISAPI runs as part of the Web server. On a multithreaded server such as Microsoft Internet Information Services, your application runs in the context of a Windows NT service. Your application may be concurrently called multiple times from different threads. Data must be protected by critical sections.

How Can I Plan for Handling Simultaneous Requests?

ISAPI DLLs should support multiple simultaneous requests, and it is the responsibility of the ISAPI DLL developer to ensure that critical data is protected.

Web servers are built to concurrently handle multiple requests. Your application plus many others will be using the server. Client traffic of millions of users is seen daily at busy sites such as `www.microsoft.com` and popular search engines. Users want fast response, and they will cancel the request if it takes too long.

As mentioned earlier, be thread-safe. Do not hold on to resources or connections unless you need to. Don't use global variables.

When you get a filter notification:

1. Process the message quickly.
2. Pass it along to the next filter.
3. Don't request high-priority notification unless you need to: for example, if you have an authentication filter that must run first, or if you are providing or preprocessing data used by other filters.

See Also

Concepts

Internet Programming Frequently Asked Questions

How Do I Use the MFC ISAPI Classes Without Linking to the MFC DLL?

To avoid linking to the MFC DLL

1. Access the project's Property Pages dialog box. See Setting Visual C++ Project Properties for more information.
2. In the **General** property page (**Configuration Properties** folder), set the **Use of MFC** property to **Use Standard Windows Libraries**.

Follow the comments generated by the ISAPI Extension Wizard in your ISAPI PROJECT.CPP file. Generate your own **AfxGetResourceHandle** function and your own special **DllMain** by uncommenting the lines in the file.

Don't include headers for MFC classes other than AFXISAPI.H.

See Also

Concepts

Internet Programming Frequently Asked Questions

How Do I Script an ActiveX Control on a Web Page?

An easy way to get started with scripting is by using an HTML authoring tool like Microsoft ActiveX Control Pad. This topic shows how to use ActiveX Control Pad to script CIRC, an MFC control sample.

The ActiveX Control Pad is an authoring tool that lets you add ActiveX controls and scripting (such as JScript and VB Script) to your HTML pages with point-and-click ease. The ActiveX Control Pad also includes the Microsoft HTML Layout Control, which supports windowless controls and provides new layout capabilities based on the HTML extensions proposed by the World Wide Web Consortium (W3C).

To install and run ActiveX Control Pad

- The ActiveX Control Pad is freely available for downloading from the authoring tools section in the MSDN Online Web Workshop at http://msdn.microsoft.com/workshop/misc/cpad/default.asp. Follow the online instructions for installation.

To insert CIRC

1. Download the CIRC sample files from MSDN and build the DLL (Circ31.dll).
2. Start ActiveX Control Pad. It opens with a default HTML document, including head and body tags.
3. On the **Edit** menu, click **Insert ActiveX Control**.
4. In the **Insert ActiveX Control** dialog box, click the CIRC control to add it to the page.

   The control will be displayed in the **Edit ActiveX Control** window. You should see a rectangular control containing Circ31. Its properties can be set by changing values in the **Properties** window.
5. Close the **Edit ActiveX Control** window.

   The OBJECT tag will be displayed on your HTML page, including the control's class ID and other parameters used to instantiate the control at run time.

Now you are ready to script the control.

To use VBScript to invoke properties and methods on CIRC

1. Start ActiveX Control Pad and open the file created in the procedure To Insert CIRC, if it is not already open.
2. On the **Tools** menu, click **Options**, and then click **Script**.
3. In the **Script Options** dialog box, select Visual Basic Scripting Edition as the Default Script Language and click **OK**.
4. From the **Tools** menu, click **Script Wizard**.

   Steps 5 and 6 are performed in the **Script Wizard** dialog box.
5. In the **Select an Event** list, click **Window**, and then click **Load** to add code to run when the control is loaded.
6. In the **Insert Actions** list, click **AboutBox**, and click **Insert Action**. The following code to invoke the AboutBox method is inserted into the HTML file:

   ```
   <SCRIPT LANGUAGE="VBScript">
   <!--
   Sub window_onLoad()
   call Circ31.AboutBox()
   end sub
   -->
   ```
7. Save the file and open it in a browser. You should see the About box displayed.

To use JScript to invoke properties and methods on CIRC

1. Start ActiveX Control Pad and open the file created in the procedure To Insert CIRC, if it is not already open.

2. On the **Tools** menu, click **Options**, and then click **Script**.
3. In the **Script Options** dialog box, select JScript as the Default Script Language and click **OK**.
4. From the **Tools** menu, click **Script Wizard**.

   Steps 5 and 6 are performed in the **Script Wizard** dialog box.
5. In the **Select an Event** list, click **Window**, and then click **Load** to add code to run when the control is loaded.
6. In the **Insert Actions** list, click **AboutBox**, and click **Insert Action**. The following code to invoke the AboutBox method is inserted into the HTML file:

```
<SCRIPT LANGUAGE="jscript" FOR="window" EVENT="onLoad()">
<!--
Circ31.AboutBox()
-->
```

7. Save the file and open it in a browser. You should see the About box displayed.

Using an authoring tool, it is simple to add a control to a Web page and to use scripting to invoke its properties and methods. You can easily attach JScript or VBScript code to buttons on your HTML pages.

See Also

Concepts

Internet Programming Frequently Asked Questions

MFC Internet Programming Tasks

This section contains detailed steps for adding Internet support to your applications. Topics include how to use the MFC classes to Internet-enable your existing applications, how to create a signed Cabinet (CAB) file, and how to add Active document support to your existing COM component. Do you want to create a document with up-to-the-minute stock quotes, Pittsburgh's football scores, and the latest temperature in Antarctica? Microsoft provides a number of technologies to help you do that over the Internet.

Active technologies include ActiveX controls (formerly OLE controls) and Active documents; ISAPI for creating COM components (Internet server extensions and filters); WinInet for easily retrieving and saving files across the Internet; and asynchronous monikers for efficient data downloading. Visual C++ provides wizards to help you get started quickly with a starter application. For an introduction to these technologies, see MFC Internet Programming Basics and MFC COM.

Have you always wanted to FTP a file but haven't learned WinSock and network programming protocols? WinInet classes encapsulate these protocols, providing you with a simple set of functions you can use to write a client application on the Internet to download files using HTTP, FTP, and gopher. You can use WinInet to search directories on your hard drive or around the world. You can transparently collect data of several different types, and present it to the user in an integrated interface.

Do you have large amounts of data to download? Asynchronous monikers provide a COM (Component Object Model) solution for progressive rendering of large objects. WinInet can also be used asynchronously.

Do you want to run a custom application on your Web server? ISAPI server extension classes provide a way to do that. ISAPI filters let you track and examine selected data sent to your Web server.

The following table describes a few of the things you can do with these technologies.

You have	You want to	You should
A Web server.	Track logons and detailed information about URL requests.	Write an ISAPI filter, request notifications for logon events and URL mapping.
A Web browser.	Provide dynamic content.	Create ActiveX controls and Active documents.
Clients with different browsers connecting to your server.	Change HTML data sent from server to client, so people with newer browsers get the latest features, and people with older browsers can still read the information.	Write an ISAPI filter.
A document-based application.	Add support to FTP a file.	Use WinInet or asynchronous monikers.

See the following topics for details to get you started:

- Application Design Choices
- Writing MFC Applications
- ActiveX Controls on the Internet
- Upgrading an Existing ActiveX Control
- Creating Signed CAB Files for MFC and ATL Controls
- Active Documents on the Internet
- Asynchronous Monikers on the Internet
- Testing Internet Applications
- Internet Security

See Also

Concepts

MFC Internet Programming Basics

Application Design Choices

This article discusses some of the design issues to consider when programming for the Internet.

Topics covered in this article include:

- Intranet Versus Internet
- Client or Server Application
- The Web Page: HTML, Active Documents, ActiveX Controls
- Browser or Stand-Alone Application
- COM on the Internet
- Client Data Download Services
- Server Extensions and Filters

If you are ready to start writing your program now, see Writing MFC Applications.

Intranet Versus Internet

Many applications run on the Internet and are accessible to anyone with a browser and Internet access. Businesses are also implementing intranets, which are company-wide networks using TCP/IP protocols and Web browsers. Intranets offer an easily upgradeable, central source for company-wide information. They can be used for upgrading software, for delivering and tabulating surveys, for customer support, and for information delivery. The following table compares features of the Internet and intranets.

Internet	**Intranet**
Low bandwidth	High bandwidth
Reduced security of data and systems	Controlled access to data and systems
Minimal control of content	High control of content

Client or Server Application

Your application may run on a client computer or on a server computer. Your application may also be stored on a server, and then downloaded across the Internet and run on a client computer. MFC WinInet classes are used for client applications to download files. MFC and asynchronous moniker classes are used to download files and control properties. MFC ISAPI classes are used for server applications. Classes for ActiveX controls and Active documents are used for client applications and for applications that are downloaded from the server to run on a client.

The Web Page: HTML, Active Documents, ActiveX Controls

Microsoft offers several ways of providing content on a Web page. Web pages can use standard HTML or HTML extensions, such as the object tag, to provide dynamic content such as ActiveX controls.

Web browsers typically display HTML pages. Active documents can also display your application's data in the simple point-and-click interface of a COM-enabled browser. Your Active document server can display your document, full frame in the entire client area, with its own menus and toolbars.

ActiveX controls you write can be downloaded asynchronously from the server and displayed on a Web page. You can use a scripting language such as VBScript to perform client-side validation before sending information to the server.

Browser or Stand-Alone Application

You can write ActiveX controls that are embedded in an HTML page and Active document servers that are viewed in a browser. You can write HTML pages that contain a button to submit a request to run your ISAPI application on a Web server. You can write a stand-alone application that uses Internet protocols to download files and display the information to your user, without ever using a browser application.

COM on the Internet

ActiveX controls, Active documents, and asynchronous monikers all use COM (Component Object Model) technologies.

ActiveX controls provide dynamic content to documents and pages on Internet sites. With COM you can build ActiveX controls and full-frame documents using Active documents.

Asynchronous monikers provide features to enable a control to perform well in an Internet environment, including an incremental or progressive means to download data. Controls must also work well with other controls that may also be retrieving their data asynchronously at the same time.

Client Data Download Services

Two sets of APIs that will help transfer data to your client are WinInet and asynchronous monikers. If you have large .gif and .avi files and ActiveX controls on your HTML page, you can increase the responsiveness to the user by downloading asynchronously, either by using asynchronous monikers or using WinInet asynchronously.

A common task on the Internet is transferring data. If you are already using Active technology (for example, if you have an ActiveX control), you can use asynchronous monikers to progressively render data as it downloads. You can use WinInet to transfer data using common Internet protocols like HTTP, FTP, and gopher. Both methods provide protocol independence, and provide an abstract layer to using WinSock and TCP/IP. You can still use WinSock directly.

The following table summarizes several ways of using MFC to transfer data across the Internet.

Use this protocol	Under these conditions	Using these classes
Internet Downloading Using Asynchronous Monikers	For asynchronous transfer using COM, ActiveX controls, and any Internet protocol.	CAsyncMonikerFile, CDataPathProperty
WinInet	For Internet protocols for HTTP, FTP, and gopher. Data can be transferred synchronously or asynchronously and is stored in a system-wide cache.	CInternetSession, CFtpFileFind, CGopherFileFind, and many more.
WinSock	For maximum efficiency and control. Requires understanding of sockets and TCP/IP protocols.	CSocket, CAsyncSocket

Server Extensions and Filters

The Internet Server API (ISAPI) provides a simple and efficient way to extend any ISAPI-compliant Web server. You can write ISAPI server extension DLLs (ISAs) that can be loaded and called by the HTTP server. An ISA is invoked from a browser application and can be used for database applications using forms processing, or to provide other custom functionality.

You can write an ISAPI filter to register for notification of events such as logging on or URL mapping. When the selected events occur, the filter is called, and you can monitor and change the data (on its way from the server to the client or vice versa).

Both server extensions and filters run in the process space of the Web server, providing an efficient way to extend the server's capabilities.

See Also

Concepts

MFC Internet Programming Tasks
MFC Internet Programming Basics
Win32 Internet Extensions (WinInet)
Asynchronous Monikers on the Internet

Writing MFC Applications

This article explains the initial steps you take to develop your application. First, you must decide what kind of application you are writing. Several of the choices were discussed in Application Design Choices. Will your application be:

- Running on the Internet or an intranet?
- Running on a client or on a server?
- Running in a browser or as a stand-alone application?
- Using COM or Active technology?
- Downloading data using WinInet or asynchronous monikers?
- A server extension or a filter?

Your decisions determine which classes are appropriate for your application. Your answers also help determine the selections you make when you run the Application Wizard to begin constructing your application.

After you've made your initial design decisions about your Internet application, you can use the Application Wizard to get started. Use the Application Wizard to create a skeleton application and modify the code as described in the following articles:

- For an ISAPI filter, see Creating a Typical ISAPI Filter.
- For an ISAPI server extension, see Creating a Typical ISAPI Extension.
- For an ActiveX control, see ActiveX Controls on the Internet.
- For an Active document, see Active Documents on the Internet.

The following articles also provide instructions to help you start your programming tasks:

- Application Design Choices
- Asynchronous Monikers on the Internet
- ISAPI Server Extensions and Filters
- WinInet Basics

See Also

Concepts

MFC Internet Programming Tasks
MFC Internet Programming Basics

ActiveX Controls on the Internet

ActiveX controls are the updated version of the OLE control specification. Controls are a primary architecture for developing programmable software components that can be used in a variety of different containers, including COM-aware Web browsers on the Internet. Any ActiveX control can be an Internet control and can add its functionality to an Active document or be part of a Web page. Controls on a Web page can communicate with each other using scripting.

ActiveX controls are not limited to the Internet. An ActiveX control can also be used in any container, as long as the control supports the interfaces required by that container.

ActiveX controls have several advantages, including:

- Fewer required interfaces than previous OLE controls.
- The ability to be windowless and always in-place active.

In order to be an ActiveX control, a control must:

- Support the **IUnknown** interface.
- Be a COM object.
- Export **DLLRegisterServer** and **DLLUnRegisterServer**.
- Support additional interfaces as needed for functionality.

Making Your Existing Controls Internet-Friendly

Designing a control that will work well in an Internet environment requires consideration for the relatively low transmission rates on the Internet. You can use your existing controls; however, there are steps you should take to make your code size smaller and to make your control properties download asynchronously.

To improve performance of your controls, follow these tips on efficiency considerations:

- Implement the techniques described in the article ActiveX Controls: Optimization.
- Consider how a control is instantiated.
- Be asynchronous; don't hold up other programs.
- Download data in small blocks.

  When downloading large streams such as bitmaps or video data, access a control's data asynchronously in cooperation with the container. Retrieve the data in an incremental or progressive fashion, working cooperatively with other controls that may also be retrieving data. Code can also be downloading asynchronously.
- Download code and properties in the background.
- Become user-interface active as quickly as possible.
- Consider how persistent data is stored, both properties and large data BLOBs (such as a bitmap image or video data).

  Controls with significant amounts of persistent data, such as large bitmaps or AVI files, require careful attention to downloading method. A document or page can become visible as soon as possible, and allow the user to interact with the page while controls retrieve data in the background.
- Write efficient routines to keep code size and run time down.

  Small button and label controls, with only a few bytes of persistent data, are suitable for use in the Internet environment and work well inside browsers.
- Consider progress is communicated to the container.

  Notify the container of progress in the asynchronous download, including when the user can start to interact with a page, and when the download is complete. The container can display progress (such as percent complete) to the user.
- Consider how controls are registered on the client computer.

Creating a New ActiveX Control

When creating a new control using the Application Wizard, you can choose to enable support for asynchronous monikers as well as other optimizations. To add support to download control properties asynchronously, follow these steps:

To create your project using the MFC ActiveX Control Wizard

1. Click **New** on the **File** menu.
2. Select **MFC ActiveX Control Wizard** from the Visual C++ projects and name your project.
3. On the **Control Settings** page, select **Loads properties asynchronously**. Selecting this option sets up the ready state property and the ready state changed event for you.

   You can also select other optimizations, such as **Windowless activation**, which is described in ActiveX Controls: Optimization.
4. Choose **Finish** to create the project.

To create a class derived from CDataPathProperty

1. Create a class derived from **CDataPathProperty**.
2. In each of your source files that includes the header file for your control, add the header file for this class before it.
3. In this class, override **OnDataAvailable**. This function is called whenever data is available for display. As data becomes available, you can handle it any way you choose, for example by progressively rendering it.

   The code excerpt below is a simple example of progressively displaying data in an edit control. Note the use of flag **BSCF_FIRSTDATANOTIFICATION** to clear the edit control.

   ```
   void CMyDataPathProperty::OnDataAvailable(DWORD dwSize, DWORD bscfFlag)
   {
       CListCtrl list_ctrl;
       CEdit* edit=list_ctrl.GetEditControl();
       if (bscfFlag & BSCF_FIRSTDATANOTIFICATION && edit->m_hWnd)
       {
           edit->SetSel(0, -1);
           edit->Clear();
       }
       if (!dwSize)
           return;

   CString string;
   LPTSTR str=string.GetBuffer(dwSize);
   UINT nBytesRead=Read(str, dwSize);
   if (!nBytesRead)
           return;
       string.ReleaseBuffer(nBytesRead);
       edit->SetSel(-1, -1);
       edit->ReplaceSel(string);
   }
   ```

   Note that you must include AFXCMN.H to use the **CListCtrl** class.
4. When your control's overall state changes (for example, from loading to initialized or user interactive), call **COleControl::InternalSetReadyState**. If your control has only one data path property, you can add code on **BSCF_LASTDATANOTIFICATION** to notify the container that your download is complete. For example:

   ```
   if (bscfFlag == BSCF_LASTDATANOTIFICATION)
   {
      GetControl()->InternalSetReadyState(READYSTATE_COMPLETE);
   }
   ```

5. Override **OnProgress**. In **OnProgress**, you are passed a number showing the maximum range and a number showing how far along the current download is. You can use these numbers to display status such as percent complete to the user.

The next procedure adds a property to the control to use the class just derived.

To add a property

1. In **Class View**, right-click the interface underneath the library node and select **Add**, then **Add Property**. This will start the **Add Property Wizard**.
2. In the **Add Property Wizard**, select the **Set/Get Methods** radio button, type the **Property Name**, for example, EditControlText, and select BSTR as the **Property type**.
3. Click **Finish**.
4. Declare a member variable of your **CDataPathProperty**-derived class to your ActiveX control class.

```
CMyDataPathProperty EditControlText;
```

5. Implement the **Get/Set** methods. For **Get**, return the string. For **Set**, load the property and call **SetModifiedFlag**.

```
BSTR CDataPathCtrl::GetDataPath()
{
    CString strResult;
    strResult = EditControlText.GetPath();
    return strResult.AllocSysString();
}
void CDataPathCtrl::SetDataPath(LPCTSTR lpszNewValue)
{
    Load(lpszNewValue, EditControlText);
    SetModifiedFlag();
}
```

6. In DoPropExchange, add the following line:

```
PX_DataPath(pPX, _T("DataPath"), EditControlText);
```

7. Override ResetData to notify the property to reset its control by adding this line:

```
EditControlText.ResetData();
```

Deciding Whether to Derive from CDataPathProperty or CCachedDataPathProperty

The previous example describes steps for deriving your control's property from **CDataPathProperty**. This is a good choice if you are downloading real-time data that frequently changes, and for which you do not need to keep all the data, but only the current value. An example is a stock ticker control.

You can also derive from **CCachedDataPathProperty**. In this case, the downloaded data is cached in a memory file. This is a good choice if you need to keep all the downloaded data — for example, a control that progressively renders a bitmap. In this case, the class has a member variable containing your data:

```
    CMemFile m_Cache;
```

In your ActiveX control class, you can use this memory mapped file in **OnDraw** to display the data. In your ActiveX control **CCachedDataPathProperty**-derived class, override the member function **OnDataAvailable** and invalidate the control, after calling the base class implementation.

```
COleControl OLEctrl;
void CMyCachedDataPathProperty::OnDataAvailable(DWORD dwSize, DWORD bscfFlag)
```

```
{
    CCachedDataPathProperty:OnDataAvailable(dwSize, bscfFlag);
    OLEctrl.InvalidateControl(NULL);
}
```

Downloading Data Asynchronously Using ActiveX Controls

Downloading data over a network should be done asynchronously. The advantage of doing so is that if a large amount of data is transferred or if the connection is slow, the download process will not block other processes on the client.

Asynchronous monikers provide a way to download data asynchronously over a network. A Read operation on an Asynchronous moniker returns immediately, even if the operation has not been completed.

For example, if only 10 bytes are available and Read is called asynchronously on a 1K file, Read does not block, but returns with the currently available 10 bytes.

You implement asynchronous monikers using the **CAsyncMonikerFile** class. However, ActiveX controls can use the **CDataPathProperty** class, which is derived from **CAsyncMonikerFile**, to help implement asynchronous control properties.

The ASYNDOWN sample demonstrates how to set up an asynchronous loop using timers to read the data. ASYNDOWN is described in detail in the Knowledge Base article "HOWTO: AsyncDown Demonstrates Asynchronous Data Download" (Q177244) and is available for download from the Microsoft Download Center. (For more information about downloading files from the Microsoft Download Center, see the article "How to Obtain Microsoft Support Files from Online Services" (Q119591) in the Microsoft Knowledge Base.) You can find Knowledge Base articles on the MSDN Library CD-ROM or at http://support.microsoft.com/support.

The basic technique used in ASYNDOWN is to set a timer in **CDataPathProperty::OnDataAvailable** to indicate when data is available. When the timer message is received, the application reads in 128-byte blocks of data and fills an edit control. If data is not available when the timer message is handled, the timer is turned off. **OnDataAvailable** turns on the timer if more data arrives later.

Displaying a Control on a Web Page

Here is an example of an object tag and attributes for inserting a control on a Web page.

```
<OBJECT
CLASSID="clsid:FC25B780-75BE-11CF-8B01-444553540000"
CODEBASE="/ie/download/activex/iechart.ocx"
ID=chart1
WIDTH=400
HEIGHT=200
ALIGN=center
HSPACE=0
VSPACE=0
>
<PARAM NAME="BackColor" value="#ffffff">
<PARAM NAME="ForeColor" value="#0000ff">
<PARAM NAME="url" VALUE="/ie/controls/chart/mychart.txt">
</OBJECT>
```

Updating an Existing OLE Control to Use New ActiveX Control Features

If your OLE control was created with a version of Visual C++ prior to 4.2, there are steps you can take to improve its performance and enhance its functionality. For a detailed discussion of these changes, see ActiveX Controls: Optimization.

If you are adding asynchronous property support to an existing control, you will need to add the ready state property and the `ReadyStateChange` event yourself. In the constructor for your control, add:

```
m_lReadyState = READYSTATE_LOADING;
```

You will update the ready state as your code is downloaded by calling COleControl::InternalSetReadyState. One place you could call **InternalSetReadyState** is from the **OnProgress** override of **CDataPathProperty**-derived class.

Then, follow the steps in Creating a New ActiveX Control.

See Also

Concepts

MFC Internet Programming Tasks

MFC Internet Programming Basics

Upgrading an Existing ActiveX Control

Existing ActiveX controls (formerly OLE controls) can be used on the Internet without modification. However, you may want to modify controls to improve their performance. When using your control on a Web page, there are additional considerations. The .ocx file and all supporting files must be on the target machine or be downloaded across the Internet. This makes code size and download time an important consideration. Downloads can be packaged in a signed .cab file. You can mark your control as safe for scripting, and as safe for initializing.

This article discusses the following topics:

- Packaging Code for Downloading
- Marking a Control Safe for Scripting and Initializing
- Licensing Issues
- Signing Code
- Managing the Palette
- Internet Explorer Browser Safety Levels and Control Behavior

You can also add optimizations, as described in ActiveX Controls: Optimization. Monikers can be used to download properties and large BLOBs asynchronously, as described in ActiveX Controls on the Internet.

Packaging Code for Downloading

For more information on this subject, see the Knowledge Base article "Packaging MFC Controls for Use Over the Internet" (Q167158). You can find Knowledge Base articles on the MSDN Library CD-ROM or at http://support.microsoft.com/support.

The CODEBASE Tag

ActiveX controls are embedded in Web pages using the `<OBJECT>` tag. The `CODEBASE` parameter of the `<OBJECT>` tag specifies the location from which to download the control. `CODEBASE` can point at a number of different file types successfully.

Using the CODEBASE Tag with an OCX File

```
CODEBASE="http://example.microsoft.com/mycontrol.ocx#version=4,70,0,1086"
```

This solution downloads only the control's .ocx file, and requires any supporting DLLs to already be installed on the client machine. This will work for Internet Explorer and MFC ActiveX controls built with Visual C++, because Internet Explorer ships with the supporting DLLs for Visual C++ controls. If another Internet browser that is ActiveX control-capable is used to view this control, this solution will not work.

Using the CODEBASE Tag with an INF File

```
CODEBASE="http://example.microsoft.com/trustme.inf"
```

An .inf file will control the installation of an .ocx and its supporting files. This method is not recommended because it is not possible to sign an .inf file (see Signing Code for pointers on code signing).

Using the CODEBASE Tag with a CAB File

```
CODEBASE="http://example.microsoft.com/acontrol.cab#version=1,2,0,0"
```

Cabinet files are the recommended way to package ActiveX controls that use MFC. Packaging an MFC ActiveX control in a cabinet file allows an .inf file to be included to control installation of the ActiveX control and any dependent DLLs (such as the MFC DLLs). Using a CAB file automatically compresses the code for quicker download. If you are using a .cab file for component download, it is faster to sign the entire .cab file than each individual component.

Creating CAB Files

The CABinet Development Kit can be obtained from the MSDN Web site at http://msdn.microsoft.com/workshop/management/cab/cabdl.asp (Web links may change without notice). In this kit you will find the necessary tools to construct cabinet files.

The cabinet file pointed to by `CODEBASE` should contain the .ocx file for your ActiveX control and an .inf file to control its

installation. You create the cabinet file by specifying the name of your control file and an .inf file. Do not include dependent DLLs that may already exist on the system in this cabinet file. For example, the MFC DLLs are packaged in a separate cabinet file and referred to by the controlling .inf file.

For details on how to create a CAB file, see Creating a CAB File.

The INF File

The following example, spindial.inf, lists the supporting files and the version information needed for the MFC Spindial control. Notice the location for the MFC DLLs is a Microsoft Web site. The mfc42.cab is provided and signed by Microsoft.

```
Contents of spindial.inf:
[mfc42installer]
file-win32-x86=http://activex.microsoft.com/controls/vc/mfc42.cab
[Olepro32.dll] - FileVersion=5,0,4261,0
[Mfc42.dll] - FileVersion=6,0,8168,0
[Msvcrt.dll] - FileVersion=6,0,8168,0
```

The <OBJECT> Tag

The following example illustrates using the `<OBJECT>` tag to package the MFC Spindial sample control.

```
<OBJECT ID="Spindial1" WIDTH=100 HEIGHT=51
  CLASSID="CLSID:06889605-B8D0-101A-91F1-00608CEAD5B3"
  CODEBASE="http://example.microsoft.com/spindial.cab#Version=1,0,0,001">
    <PARAM NAME="_Version" VALUE="65536">
    <PARAM NAME="_ExtentX" VALUE="2646">
    <PARAM NAME="_ExtentY" VALUE="1323">
    <PARAM NAME="_StockProps" VALUE="0">
    <PARAM NAME="NeedlePosition" VALUE="2">
</OBJECT>
```

In this case, spindial.cab will contain two files, spindial.ocx and spindial.inf. The following command will build the cabinet file:

```
C:\CabDevKit\cabarc.exe -s 6144 N spindial.cab spindial.ocx spindial.inf
```

The `-s 6144` parameter reserves space in the cabinet for code signing.

The Version Tag

Note here that the `#Version` information specified with a CAB file applies to the control specified by the `CLASSID` parameter of the `<OBJECT>` tag.

Depending on the version specified, you can force download of your control. For complete specifications of the `OBJECT` tag including the `CODEBASE` parameter, see the W3C reference.

Marking a Control Safe for Scripting and Initializing

ActiveX controls used in Web pages should be marked as safe for scripting and safe for initializing if they are in fact safe. A safe control will not perform disk IO or access the memory or registers of a machine directly.

Controls can be marked as safe for scripting and safe for initializing via the registry. Modify **DllRegisterServer** to add entries similar to the following to mark the control as safe for scripting and persistence in the registry. An alternative method is to implement **IObjectSafety**.

You will define GUIDs (Globally Unique Identifiers) for your control to mark it safe for scripting and for persistence. Controls that can be safely scripted will contain a registry entry similar to the following:

```
HKEY_CLASSES_ROOT\Component Categories\{7DD95801-9882-11CF-9FA9-00AA006C42C4}
```

Controls that can be safely initialized from persistent data are marked safe for persistence with a registry entry similar to:

```
HKEY_CLASSES_ROOT\Component Categories\{7DD95802-9882-11CF-9FA9-00AA006C42C4}
```

Add entries similar to the following (substituting your control's class ID in place of `{06889605-B8D0-101A-91F1-`

00608CEAD5B3}) to associate your keys with the following class ID:

```
HKEY_CLASSES_ROOT\CLSID\{06889605-B8D0-101A-91F1-00608CEAD5B3}\Implemented Categories\{7DD9
5801-9882-11CF-9FA9-00AA006C42C4}
HKEY_CLASSES_ROOT\CLSID\{06889605-B8D0-101A-91F1-00608CEAD5B3}\Implemented Categories\{7DD9
5802-9882-11CF-9FA9-00AA006C42C4}
```

Licensing Issues

If you want to use a licensed control on a Web page, you must verify that the license agreement allows its use on the Internet and create a license package file(LPK) for it.

A licensed ActiveX control will not load properly in an HTML page if the computer running Internet Explorer is not licensed to use the control. For example, if a licensed control was built using Visual C++, the HTML page using the control will load properly on the computer where the control was built, but it will not load on a different computer unless licensing information is included.

To use a licensed ActiveX control in Internet Explorer, you must check the vendor's license agreement to verify that the license for the control permits:

- Redistribution
- Use of the control on the Internet
- Use of the Codebase parameter

To use a licensed control in an HTML page on a nonlicensed machine, you must generate a license package file (LPK). The LPK file contains run-time licenses for licensed controls in the HTML page. This file is generated via LPK_TOOL.EXE which comes with the ActiveX SDK. For more information, see the MSND Web site at http://msdn.microsoft.com.

To create an LPK file

1. Run LPK_TOOL.EXE on a computer that is licensed to use the control.
2. In the **License Package Authoring Tool** dialog box, in the **Available Controls** list box, select each licensed ActiveX control that will be used on the HTML page and click **Add**.
3. Click **Save & Exit** and type a name for the LPK file. This will create the LPK file and close the application.

To embed a licensed control on an HTML page

1. Edit your HTML page. In the HTML page, insert an <OBJECT> tag for the License Manager object before any other <OBJECT> tags. The License Manager is an ActiveX control that is installed with Internet Explorer. Its class ID is shown below. Set the LPKPath property of the License Manager object to the path and name of the LPK file. You can have only one LPK file per HTML page.

   ```
   <OBJECT CLASSID = "clsid:5220cb21-c88d-11cf-b347-00aa00a28331">
       <PARAM NAME="LPKPath" VALUE="relative URL to .LPK file">
   </OBJECT>
   ```

2. Insert the <OBJECT> tag for your licensed control after the License Manager tag.

   For example, an HTML page that displays the Microsoft Masked Edit control is shown below. The first class ID is for the License Manager control, the second class ID is for the Masked Edit control. Change the tags to point to the relative path of the .lpk file you created earlier, and add an object tag including the class ID for your control.

3. Insert the <EMBED> attribute for your LPK file, if using the NCompass ActiveX plug-in.

   If your control may be viewed on other Active enabled browsers — for example, Netscape using the NCompass ActiveX plug-in — you must add the <EMBED> syntax as shown below.

   ```
   <OBJECT CLASSID="clsid:5220cb21-c88d-11cf-b347-00aa00a28331">
       <PARAM NAME="LPKPath" VALUE="maskedit.lpk">
   ```

```
    <EMBED SRC = "maskedit.LPK">

</OBJECT>
<OBJECT CLASSID="clsid:C932BA85-4374-101B-A56C-00AA003668DC" WIDTH=100 HEIGHT=25>
</OBJECT>
```

For more information about control licensing, see ActiveX Controls: Licensing an ActiveX Control.

Signing Code

Code signing is designed to identify the source of code, and to guarantee that the code has not changed since it was signed. Depending on browser safety settings, users may be warned before the code is downloaded. Users may choose to trust certain certificate owners or companies, in which case code signed by those trusted will be downloaded without warning. Code is digitally signed to avoid tampering.

Make sure your final code is signed so that your control can be automatically downloaded without displaying trust warning messages. For details on how to sign code, check the documentation on Authenticode™ in the ActiveX SDK and see Signing a CAB File.

Depending on trust and browser safety level settings, a certificate may be displayed to identify the signing person or company. If the safety level is none, or if the signed control's certificate owner is trusted, a certificate will not be displayed. See Internet Explorer Browser Safety Levels and Control Behavior for details on how the browser safety setting will determine whether your control is downloaded and a certificate displayed.

Digital signing guarantees code has not changed since it's been signed. A hash of the code is taken and embedded in the certificate. This hash is later compared with a hash of the code taken after the code is downloaded but before it runs. Companies such as Verisign can supply private and public keys needed to sign code. The ActiveX SDK ships with MakeCert, a utility for creating test certificates, and two registry files, wvtston.reg and wvtstoff.reg, for specifying whether or not the browser should recognize test certificates as valid.

Managing the Palette

Containers determine the palette and make it available as an ambient property, **DISPID_AMBIENT_PALETTE**. A container (for example, Microsoft Internet Explorer) chooses a palette that is used by all ActiveX controls on a page to determine their own palette. This prevents display flickering and presents a consistent appearance.

A control can override **OnAmbientPropertyChange** to handle notification of changes to the palette.

A control can override **OnGetColorSet** to return a color set to draw the palette. Containers use the return value to determine if a control is palette-aware.

Under OCX 96 guidelines, a control must always realize its palette in the background.

Older containers that do not use the ambient palette property will send **WM_QUERYNEWPALETTE** and **WM_PALETTECHANGED** messages. A control can override **OnQueryNewPalette** and **OnPaletteChanged** to handle these messages.

Internet Explorer Browser Safety Levels and Control Behavior

A browser has options for safety level, configurable by the user. Because Web pages can contain active content that might potentially harm a user's computer, browsers allow the user to select options for safety level. Depending on the way a browser implements safety levels, a control may not be downloaded at all, or will display a certificate or a warning message to allow the user to choose at run time whether or not to download the control. The behavior of ActiveX controls under high, medium, and low safety levels on Internet Explorer is listed below.

High Safety Mode

- Unsigned controls will not be downloaded.
- Signed controls will display a certificate if untrusted (a user can choose an option to always trust code from this certificate owner from now on).
- Only controls marked as safe will have persistent data and/or be scriptable.

Medium Safety Mode

- Unsigned controls will display a warning before downloading.

- Signed controls will display a certificate if untrusted.
- Controls not marked as safe will display a warning.

Low Safety Mode

- Controls are downloaded without warning.
- Scripting and persistence occur without warning.

See Also

Reference

MFC ActiveX Controls: Licensing an ActiveX Control

Concepts

MFC Internet Programming Tasks

MFC Internet Programming Basics

Creating Signed CAB Files for MFC and ATL Controls

If you plan to distribute MFC and ATL controls via the Internet, you should package them as signed Cabinet (CAB) files. Signed files help assure a user downloading your control that the code is safe. A CAB file contains a compressed version of your control plus information about how your control is to be installed, for example, which DLLs need to be installed along with the OCX.

The following is information you will need for creating signed CAB files for ActiveX controls:

- **Installing the Platform SDK.** You can install the tools you need to create and sign CAB files from the Platform SDK. For download information, see Platform Software Development Kit.

  During installation, choose to install "Internet Explorer Tools." Both the code signing files and the CAB utilities can be found in the installed Mssdk\Bin directory. (Note that "Mssdk" might be named "Microsoft Platform SDK" by some installations.)

- **The MSDN Online Web Workshop.** If you have previously used the Internet Client SDK, note that this content has moved to the Web Workshop.

- **Authenticode technology and CryptoAPI Tools.** You can find information on signing code with Microsoft Authenticode technology in CryptoAPI Tools Reference, located in the *Platform SDK* documentation (in MSDN Library/Security/Cryptography).

- **Microsoft Security Advisor site.** See http://www.microsoft.com/security/.

Before you can sign files, you need to get a Software Publisher Certificate. You must apply for your own certificate to a Certification Authority. With the tools in the Mssdk\Bin directory you can create a test certificate for testing purposes, but this certificate cannot be used to sign code for distribution. See step 1 for information about applying for a Software Publisher Certificate.

These are the steps to create a signed CAB file:

1. Get a Software Publisher Certificate (you only need to do this once).
2. Create the CAB file.
3. Sign your files.
4. Embed the signed CAB file in a Web page (optional).

See Also

Concepts

MFC Internet Programming Tasks

Getting a Software Publisher Certificate

Before you can sign files, you need to obtain a Software Publisher Certificate (SPC). To do this, you must make a request to a Certification Authority. During the application process, you must generate a key pair and provide the Certification Authority with identification information, such as your name, address, and public key. You must also make a legally binding pledge that you cannot and will not distribute software you know or should have known contains viruses or will otherwise maliciously harm the user's machine or code.

The Certification Authority generates a Software Publisher Certificate that conforms to the industry standard X.509 certificate format with Version 3 extensions. The certificate identifies you and contains your public key. It is stored by the Certification Authority for reference and a copy is returned to you via electronic mail. After accepting the certificate, you should include a copy in all published software signed with the private key.

See the following resources for information on Authenticode and digitally signed certificates:

- "Certificates and Authenticode" (Microsoft TechNet) at http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/prodtech/certauth/default.asp.
- "Frequently Asked Questions About Authenticode" (MSDN) at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnauth/html/signfaq.asp.
- For information about Authenticode and obtaining a Software Publisher Certificate (SPC), read the white paper "Digital Code Signing Step-by-Step Guide" at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsmarttag/html/odc_dcss.asp.
- For information about creating a test certificate to test sign your files, see Making a Test Software Publisher Certificate.
- For information about Microsoft Internet Explorer Security, see http://www.microsoft.com/Windows/ie/security/default.asp.

Making a Test Software Publisher Certificate

You can use the MAKECERT and CERT2SPC utilities to make a test Software Publisher Certificate. Note that this test SPC is not valid for software publishing, but can be used to test code signing.

To make a private key file called MYKEY.PVK and a company certificate called CERT.CER, run the MAKECERT utility with the following command:

```
C:\MSSDK\BIN\MAKECERT -u:MyKey -n:CN=MySoftwareCompany -k:MYKEY.PVK
CERT.CER
```

MyKey is the name you give to your key and *MySoftwareCompany* is the name of your company. Note that the MAKECERT utility is case sensitive to command-line options, so you must use lowercase **-u**, **-n**, and **-k**. The value for the **-n** option must be an uppercase **CN=**.

To create a test Software Publisher Certificate (SPC) called CERT.SPC, run the CERT2SPC utility with the following command:

```
C:\MSSDK\BIN\CERT2SPC C:\MSSDK\BIN\ROOT.CER CERT.CER CERT.SPC
```

Note that the CERT.SPC file is created from the CERT.CER file you created with MAKECERT and from the ROOT.CER file provided in the same directory.

See Also

Concepts

Creating Signed CAB Files for MFC and ATL Controls

Creating a CAB File

This section describes creating CAB files for distributing ATL and MFC controls over the Internet. If you need more information about CAB files, see the Cabinet File Reference in the *Platform SDK* documentation (in MSDN Library/Setup and System Administration/Setup/Setup API/Overview/Cabinet Files).

To create a CAB file:

1. Create an INF file.
2. Run the CABARC utility.

Creating an INF File

The INF file is a text file that specifies the files (such as DLLs or other OCXs) that need to be present or downloaded for your control to run. An INF file allows you to bundle all the needed files in one compressed CAB file. By default, files with the same version numbers as existing files on the user's hard disk will not be downloaded. For more information about INF files and their options, including how to create platform-independent INF files, see About INF Files and Using INF Files in the *Platform SDK* documentation (in MSDN Library/Setup and System Administration/Setup/Setup API/Overview/Setup Applications).

As an example, the following INF will be used to create a CAB file for the ATL Polygon control. You can build POLYGON.DLL by downloading the ATL POLYGON sample files from the Visual C++ CD and building a MinSize version. If you build a MinSize version of the Polygon control, you need one additional DLL, ATL.DLL. Since ATL.DLL needs to be registered before POLYGON.DLL, put the ATL.DLL first in the INF file:

```
; Sample INF file for POLYGON.DLL
[version]
; version signature (same for both NT and Win95) do not remove
signature="$CHICAGO$"
AdvancedINF=2.0

[Add.Code]
polygon.dll=polygon.dll
atl.dll=atl.dll

; needed DLL
[atl.dll]
file-win32-x86=thiscab
FileVersion=2,00,0,7024
DestDir=11
RegisterServer=yes

[polygon.dll]
file-win32-x86=thiscab
clsid={4CBBC676-507F-11D0-B98B-000000000000}
FileVersion=1,0,0,1
RegisterServer=yes
; end of INF file
```

This INF specifies that ATL.DLL with the given version needs to be installed on the system. If ATL.DLL doesn't exist already on the system, it will be downloaded from the CAB file created with this INF. "thiscab" is a keyword meaning the CAB containing this INF. You can also download a needed DLL from an HTTP location by specifying an absolute or relative path, for example:

```
file-win32-x86=http://example.microsoft.com/mydir/NEEDED.DLL
```

The keyword "file-win32-x86" identifies the platform as x86 specific.

You can get the version number of a file by clicking the right mouse button on the file in Windows Explorer. Select **Properties** from the list that appears, then select the **Version** tab on the dialog box that appears. You will sometimes need to insert an extra 0 in the file version. For example, the version number for the ATL.DLL is shown as 2.00.7024 in the dialog box. This becomes 2, 00, 0, 7024 in the INF file.

The "DestDir" is where the directory where the file will be loaded: 11 specifies the system directory WINDOWS/SYSTEM or

WINNT/SYSTEM32; 10 specifies the windows directory, WINDOWS or WINNT. If no DestDir is specified (typical case), code is installed in the fixed OCCACHE directory.

The "clsid" is the CLSID of the control to be installed.

Once you have created an INF file, run the CABARC utility (available in the Mssdk\Bin directory) to create the CAB file. You should run CABARC in the directory that contains your source files. On the command line, put the source files in the order they appear in the INF and the INF file last. For example, to make a CAB file for the Polygon control from the INF above, use the following command:

```
C:\MSSDK\BIN\CABARC -s 6144 POLYGON.CAB ATL.DLL POLYGON.DLL POLYGON.INF
```

The POLYGON.CAB file contains a compressed version of ATL.DLL and POLYGON.DLL along with the information needed to extract them in the POLYGON.INF file.

For an example of how to parse and extract component files from CAB files, see the CabView sample in the MSDN Online Code Center at http://msdn.microsoft.com/visualc/downloads/samples.asp (select the CabView link).

The DLL files you need to include with an MFC control are MSVCRT.DLL, MFC42.DLL, and OLEPRO32.DLL.

Running the CABARC Utility

The CABARC utility is available in the Mssdk\Bin directory. For example:

```
C:\MSSDK\BIN\CABARC -s 6144 n MYCTL.CAB NEEDED1.DLL NEEDED2.DLL MYCTL.OCX MYCTL.INF
```

CABARC creates a CAB file called MYCTL.CAB.

You should run CABARC in the directory that contains your source files (the INF, OCX, and DLL files). The files to be archived in the CAB file should be listed on the command line in the same order they are listed in the INF file. In the example above, the INF file should list NEEDED1.DLL first, then NEEDED2.DLL, and then MYCTL.OCX.

The **-s** option reserves space in the cabinet for code signing. The **n** command specifies that you want to create a CAB file. For a list of CABARC commands and options, type CABARC alone on the command line:

```
C:\MSSDK\BIN\CABARC
```

See Also

Concepts

Creating Signed CAB Files for MFC and ATL Controls

Signing a CAB File

To sign a CAB file using the Code Signing Wizard, run the SIGNCODE utility, which ships in Visual Studio.

You can sign your DLLs and OCXs directly, without using a CAB file. The advantages of a CAB file are compression and, if used with an INF file, the bundling of all necessary code together.

See Also

Concepts

Creating Signed CAB Files for MFC and ATL Controls

Embedding a Signed CAB File on a Web Page

ATL and MFC controls are embedded in Web pages using the <OBJECT> tag. Within the <OBJECT> tag, you need to specify three attributes for the control:

- ID The name of the control
- CLASSID The CLSID of the control
- CODEBASE The location from which to download the control. Note that CODEBASE can point at a number of different file types successfully.

As an example, the following Web page displays the ATL Polygon control archived in the CAB file created in the section on Creating an INF File. The address `http://example.microsoft.com/mydir/polygon.cab` should be replaced with the actual HTTP address of the CAB file:

```
<HTML>
<HEAD>
<TITLE>ATL 2.0 test page for object PolyCtl</TITLE>
</HEAD>
<BODY>
<OBJECT ID="PolyCtl"
CLASSID="CLSID:4CBBC676-507F-11D0-B98B-000000000000"
CODEBASE="http://example.microsoft.com/mydir/polygon.cab">
</OBJECT>
<SCRIPT LANGUAGE="VBScript">
<!--
Sub PolyCtl_ClickIn(x, y)
PolyCtl.Sides = PolyCtl.Sides + 1
End Sub
Sub PolyCtl_ClickOut(x, y)
PolyCtl.Sides = PolyCtl.Sides - 1
End Sub
-->
</SCRIPT>
</BODY>
</HTML>
```

CODEBASE can point directly to an OCX or DLL file:

```
CODEBASE="http://example.microsoft.com/mydir/polygon.dll#version=1,0,0,1"
```

Since this causes only the DLL or OCX file to be downloaded and installed, any necessary supporting DLLs must already be on the client machine.

If you include the optional version number with a CAB file, the version number should refer to the control being downloaded. For example, since POLYGON.DLL has a version number of 1, 0, 0, 1, the version for the CAB is also 1, 0, 0, 1:

```
CODEBASE="http://example.microsoft.com/mydir/polygon.cab#version=1,0,0,1"
```

If you do not include the version number, older versions of the same control will not be replaced if they are found on the client machine.

See Also

Concepts

Creating Signed CAB Files for MFC and ATL Controls

Active Documents on the Internet

Active documents provide an extension to traditional embedded objects. The Active documents may be multipage and are displayed in the entire client area. They do traditional menu negotiation, and can be edited in-place as well as in an open window in the server application. Instead of displaying as a small rectangle surrounded by a hatched border, Active documents are full frame and always in-place active.

Active documents can be viewed in a container like the Microsoft Office Binder, which provides a way to create a compound document made up of different document types like Excel, Word, and your custom document type, each of which can be edited full frame. Active documents can also be displayed in a browser such as Microsoft Internet Explorer, which is an Active document container.

Active document advantages include:

- Documents can be viewed full frame, in the entire client window.
- Documents can be opened in a separate application window.

  For the document to open, the helper application must exist on the client, or be downloaded separately before the application can run. A viewer may be written to provide limited functionality (Word, PowerPoint, and Excel provide viewers for their documents). The full version of the application can provide full editing support.
- Documents are always in-place active.
- Menu commands invoked from the container can be routed to your document.
- Documents can be viewed in a Web browser. This provides seamless integration between your documents and other Web pages.

  A user can browse an HTML Web page, then an Excel spreadsheet, and then to a document that you have written using MFC support for Active documents. The user can navigate using the familiar Web interface, as the browser switches seamlessly between the menus and views of an HTML page, Excel, and your application's document.
- All applications are displayed in a common frame.

Requirements for Active Documents

The interfaces listed in the table below include interfaces already required for embedded servers and several new interfaces specific to Active documents. MFC provides default implementations for most of these interfaces in the COleServerDoc class.

A document that ...	Implements these interfaces
Uses compound files as its storage mechanism.	**IPersistStorage**.
Supports the basic embedding features of Active documents, including Create From File.	**IPersistFile**, **IOleObject**, and **IDataObject**.
Supports in-place activation.	**IOleInPlaceObject** and **IOleInPlaceActiveObject** (using the container's **IOleInPlaceSite** and **IOleInPlaceFrame** interfaces).
Supports the Active document extensions that involve these new interfaces. Some interfaces are optional.	**IOleDocument**, **IOleDocumentView**, **IOleCommandTarget**, and **IPrint**.

MFC provides support for extending existing embedded server support to Active documents.

Add Active Document Support to a New Application

To create a new application with Active document support: In the MFC Application Wizard, on the **Compound Document Support** page, under "Select compound document support" choose **Full-server** or **Container/Full-server**, and under "Select additional options" select the check box for **Active document server**.

Convert an Existing MFC In-Process Server to an Active Document Server

If your application was created with a version of Visual C++ prior to version 4.2 and is already an in-process server, you can

add Active document support by making changes to the following classes:

Class type	Formerly derived from	Change to derive from
In-Place Frame	**COleIPFrameWnd**	**COleDocIPFrameWnd**
Item	**COleServerItem**	**CDocObjectServerItem**

You will also change how information is entered in the registry, and make several other changes. If your application currently has no COM components support, you can add server support by running the Application Wizard and integrating the COM component-specific code with your existing application.

See Also

Concepts

MFC Internet Programming Tasks
MFC Internet Programming Basics

Asynchronous Monikers on the Internet

The Internet requires new approaches to application design because of its slow network access. Applications should perform network access asynchronously to avoid stalling the user interface. The MFC class CAsyncMonikerFile provides asynchronous support for downloading files.

With asynchronous monikers, you can extend your COM application to download asynchronously across the Internet and to provide progressive rendering of large objects such as bitmaps and VRML objects. Asynchronous monikers enable an ActiveX control property or a file on the Internet to be downloaded without blocking the response of the user interface.

Advantages of Asynchronous Monikers

You can use asynchronous monikers to:

- Download code and files without blocking.
- Download properties in ActiveX controls without blocking.
- Receive notifications of downloading progress.
- Track progress and ready state information.
- Provide status information to the user about progress.
- Allow the user to cancel a download at any time.

MFC Classes for Asynchronous Monikers

CAsyncMonikerFile is derived from CMonikerFile, which in turn is derived from COleStreamFile. A **COleStreamFile** object represents a stream of data; a **CMonikerFile** object uses an **IMoniker** to obtain the data, and a **CAsyncMonikerFile** object does so asynchronously.

Asynchronous monikers are used primarily in Internet-enabled applications and ActiveX controls to provide a responsive user interface during file transfers. A prime example of this is the use of CDataPathProperty to provide asynchronous properties for ActiveX controls.

MFC Classes for Data Paths in ActiveX Controls

The MFC classes **CDataPathProperty** and CCachedDataPathProperty implement ActiveX control properties that can be loaded asynchronously. Asynchronous properties are loaded after synchronous initiation. Asynchronous ActiveX controls repeatedly invoke a callback to indicate availability of new data during a lengthy property exchange process.

CDataPathProperty is derived from **CAsyncMonikerFile**. **CCachedDataPathProperty** is derived from **CDataPathProperty**. To implement asynchronous properties in your ActiveX controls, derive a class from **CDataPathProperty** or **CCachedDataPathProperty**, and override OnDataAvailable and other notifications you wish to receive.

To download a file using asynchronous monikers

1. Declare a class derived from CAsyncMonikerFile.
2. Override OnDataAvailable to display the data.
3. Override other member functions, including OnProgress, OnStartBinding, and OnStopBinding.
4. Declare an instance of this class and use it to open URLs.

For information about downloading asynchronously in an ActiveX control, see ActiveX Controls on the Internet.

See Also

Concepts

MFC Internet Programming Tasks
MFC Internet Programming Basics

Testing Internet Applications

There are some unique testing challenges on the Internet, especially for applications running on a Web server. Your initial testing for your ISAPI extension or filter will probably be done using a single-user client connecting to a test server. This will be useful for debugging your code.

You will also want to test under real conditions: with multiple clients connected over high-speed connections as well as low-speed serial lines, including modem connections. It can be difficult to simulate real conditions, but it is certainly worth spending time designing possible scenarios and executing them. If possible, you will also want to use tools to do capacity and stress testing. Certain classes of bugs, such as timing bugs, are difficult to find and to reproduce.

One of the challenges of Internet programming is its visibility. Many accesses to your site may slow down your server. You want your server to degrade gracefully. You want to prevent anything that could be destructive to a user's computer if your application fails (for example, corruption of data while writing to the registry or while writing cookies on the client).

See Also

Concepts

MFC Internet Programming Tasks
MFC Internet Programming Basics

Internet Security (C++)

Code safety is a major issue for developers and for users of Internet applications. There are risks: malicious code, code that has been tampered with, and code from unknown sites or authors.

There are two basic approaches to security when developing for the Internet. The first is called "sandboxing." In this approach, an application is restricted to a particular set of APIs, and excluded from potentially dangerous ones such as file I/O where a program could destroy data on a user's computer. The second is implemented using digital signatures. This approach is referred to as "shrinkwrap" for the Internet. Code is verified and signed using private key/public key technology. Before the code is run, its digital signature is verified to ensure that the code is from a known authenticated source, and that the code has not been altered since it has been signed.

In the first case, you trust that the application will not do any harm and you trust the origin of the application. In the second, digital signatures are used to verify authenticity. Digital signing is an industry standard used to identify and provide details about the publisher of the code. Its technology is based on standards, including RSA and X.509. Browsers typically allow users to choose if they want to download and run code of unknown origin.

Additional information about code signing and other security measures is available on the Web. Information can be accessed through the MSDN Online Web Workshop site at http://msdn.microsoft.com/, or through the World Wide Web consortium at http://www.w3.org/.

See Also

Concepts

MFC Internet Programming Tasks
MFC Internet Programming Basics

Win32 Internet Extensions (WinInet)

An Internet client application is a program that accesses information from a network data source (server) using Internet protocols such as gopher, FTP, or HTTP. An Internet client application might access a server to retrieve data such as weather maps, stock prices, or newspaper headlines, for example. The Internet client can access the server through an external network (the Internet) or an internal network (sometimes called an intranet).

MFC includes the Win32 Internet Extensions, or WinInet, for creating an Internet client application. MFC encapsulates these Internet extensions in a set of standard, easy-to-use classes. You can write a WinInet client application by calling the Win32 functions directly or by using the MFC WinInet classes.

The Microsoft Win32 Internet functions (WinInet) assist you in making the Internet an integral part of any application. The new functions, contained in WININET.DLL, simplify accessing the Internet using HTTP (Hypertext Transfer Protocol), FTP (File Transfer Protocol), and gopher.

If you want to write an Internet server application, use the MFC Internet Server API (ISAPI) extensions.

The following topics discuss the process of creating an Internet client application:

- How WinInet Makes It Easier to Create Internet Client Applications
- How MFC Makes It Easier to Create Internet Client Applications
- MFC Classes for Creating Internet Client Applications
- Prerequisites for Internet Client Classes
- Writing an Internet Client Application Using MFC WinInet Classes

The following topics provide steps for performing typical WinInet tasks:

- Steps in a Typical Internet Client Application
- Steps in a Typical FTP Client Application
- Steps in a Typical FTP Client Application to Delete a File
- Steps in a Typical Gopher Client Application
- Steps in a Typical HTTP Client Application

See Also

Concepts

MFC Internet Programming Basics

WinInet Basics

Other Resources

MFC Internet Programming

How WinInet Makes It Easier to Create Internet Client Applications

The Win32 Internet Extensions, or WinInet, provide access to common Internet protocols, including gopher, FTP, and HTTP. Using WinInet, you can write Internet client applications at a higher level of programming, without having to deal with WinSock, TCP/IP, or the details of specific Internet protocols. WinInet provides a consistent set of functions for all three protocols, with a familiar Win32 API interface. This consistency minimizes code changes you need to make if the underlying protocol changes (for example, from FTP to HTTP).

Visual C++ provides two ways for you to use WinInet. You can call the Win32 Internet functions directly (see the OLE documentation in the *Platform SDK* for more information) or you can use WinInet through the MFC WinInet classes.

You can use WinInet to:

- Download HTML pages.

  HTTP is a protocol used to transfer HTML pages from a server to a client browser.

- Send FTP requests to upload or download files or get directory listings.

  A typical request is an anonymous logon to download a file.

- Use gopher's menu system for accessing resources on the Internet.

  Menu items can be several types, including other menus, an indexed database you can search, a newsgroup, or a file.

For all three protocols, you establish a connection, make requests to the server, and close the connection.

The MFC WinInet classes make it easy to:

- Read information from HTTP, FTP, and gopher servers as easily as reading files from a hard drive.
- Use HTTP, FTP, and gopher protocols without programming directly to WinSock or TCP/IP.

  Developers who use the Win32 Internet functions do not need to be familiar with TCP/IP or Windows Sockets. You can still program at the socket level, using WinSock and TCP/IP protocols directly, but it's even easier to use the MFC WinInet classes to access HTTP, FTP, and gopher protocols across the Internet. For many common operations, developers do not need to know the details of the particular protocol they are using.

Many operations that can be performed by your computer as a client to other computers on the Internet can take a long time. The speed of these operations is usually limited by the speed of your network connection, but they can also be affected by other network traffic and the complexity of the operation. Connecting to a remote FTP server, for example, requires that your computer first look up the name of that server to find its address. Your application will then attempt to connect to the server at that address. Once the connection is opened, your computer and the remote server will initiate a conversation with the file transfer protocol before you can actually use the connection to retrieve files.

See Also

Concepts

Win32 Internet Extensions (WinInet)
How MFC Makes It Easier to Create Internet Client Applications

How MFC Makes It Easier to Create Internet Client Applications

The Microsoft Foundation Classes encapsulate the Win32 Internet Extension (WinInet) functions in a manner that provides a familiar context for MFC programmers. MFC provides three Internet file classes (CInternetFile, CHttpFile, and CGopherFile) derived from the CStdioFile class. Not only do these classes make retrieving and manipulating Internet data familiar to programmers who have used **CStdioFile** for local files, but with these classes you can handle local files and Internet files in a consistent, transparent manner.

The MFC WinInet classes provide the same functionality as **CStdioFile** for data that is transferred across the Internet. These classes abstract the Internet protocols for HTTP, FTP, and gopher into a high-level application programming interface, providing a fast and straightforward path to making applications Internet-aware. For example, connecting to an FTP server still requires several steps at a low level, but as an MFC developer, you only need to make one call to **CInternetSession::GetFTPConnection** to create that connection.

In addition, the MFC WinInet classes provide the following advantages:

- Buffered I/O
- Type-safe handles for your data
- Default parameters for many functions
- Exception handling for common Internet errors
- Automatic cleanup of open handles and connections

See Also

Concepts

Win32 Internet Extensions (WinInet)
How WinInet Makes It Easier to Create Internet Client Applications

MFC Classes for Creating Internet Client Applications

MFC provides the following classes and global functions for writing Internet client applications. Indentation indicates a class is derived from the unindented class above it. **CGopherFile** and **CHttpFile** derive from **CInternetFile**, for example. These classes and global functions are declared in AFXINET.H, except **CFileFind**, which is declared in AFX.H.

Classes

- CInternetSession
- CInternetConnection
    - CFtpConnection
    - CGopherConnection
    - CHttpConnection
- CInternetFile
    - CGopherFile
    - CHttpFile
- CFileFind
    - CFtpFileFind
    - CGopherFileFind
- CGopherLocator
- CInternetException

Global Functions

- AfxParseURL
- AfxGetInternetHandleType
- AfxThrowInternetException

See Also

Concepts

Win32 Internet Extensions (WinInet)
Prerequisites for Internet Client Classes
Writing an Internet Client Application Using MFC WinInet Classes

Prerequisites for Internet Client Classes

Some actions taken by an Internet client (reading a file, for example) have prerequisite actions (in this case, establishing an Internet connection). The following tables list the prerequisites for some client actions.

General Internet URL (FTP, Gopher, or HTTP)

Action	Prerequisite
Establish a connection.	Create a CInternetSession to establish the basis of an Internet client application.
Open a URL.	Establish a connection. Call CInternetSession::OpenURL. The **OpenURL** function returns a read-only resource object.
Read URL data.	Open the URL. Call CInternetFile::Read.
Query the setting of an Internet option.	Establish a connection. Call CInternetSession::QueryOption.
Set an Internet option.	Establish a connection. Call CInternetSession::SetOption.
Set a function to be called with status information.	Establish a connection. Call CInternetSession::EnableStatusCallback. Override CInternetSession::OnStatusCallback to handle calls.

FTP

Action	Prerequisite
Establish an FTP connection.	Create a CInternetSession as the basis of this Internet client application. Call CInternetSession::GetFtpConnection to create a CFtpConnection object.
Find the first resource.	Establish an FTP connection. Create a CFtpFileFind object. Call CFtpFileFind::FindFile.
Enumerate all available resources.	Find the first file. Call CFtpFileFind::FindNextFile until it returns FALSE.
Open an FTP file.	Establish an FTP connection. Call CFtpConnection::OpenFile to create and open a CInternetFile object.
Read an FTP file.	Open an FTP file with read access. Call CInternetFile::Read.
Write to an FTP file.	Open an FTP file with write access. Call CInternetFile::Write.
Change the client's directory on the server.	Establish an FTP connection. Call CFtpConnection::SetCurrentDirectory.
Retrieve the client's current directory on the server.	Establish an FTP connection. Call CFtpConnection::GetCurrentDirectory.

HTTP

Action	Prerequisite
Establish an HTTP connection.	Create a CInternetSession as the basis of this Internet client application. Call CInternetSession::GetHttpConnection to create a CHttpConnection object.
Open an HTTP file.	Establish an HTTP connection. Call CHttpConnection::OpenRequest to create a CHttpFile object. Call CHttpFile::AddRequestHeaders. Call CHttpFile::SendRequest.
Read an HTTP file.	Open an HTTP file. Call CInternetFile::Read.

Get information about an HTTP request.	Establish an HTTP connection. Call CHttpConnection::OpenRequest to create a CHttpFile object. Call CHttpFile::QueryInfo.

Gopher

Action	Prerequisite
Establish a gopher connection.	Create a CInternetSession as the basis of this Internet client application. Call CInternetSession::GetGopherConnection to create a CGopherConnection.
Find the first file in the current directory.	Establish a gopher connection. Create a CGopherFileFind object. Call CGopherConnection::CreateLocator to create a CGopherLocator object. Pass the locator to CGopherFileFind::FindFile. Call CGopherFileFind::GetLocator to get the locator of a file if you need it later.
Enumerate all available files.	Find the first file. Call CGopherFileFind::FindNextFile until it returns FALSE.
Open a gopher file.	Establish a gopher connection. Create a gopher locator with CGopherConnection::CreateLocator or find a locator with CGopherFileFind::GetLocator. Call CGopherConnection::OpenFile.
Read a gopher file.	Open a gopher file. Use CGopherFile.

See Also

Concepts

Win32 Internet Extensions (WinInet)
MFC Classes for Creating Internet Client Applications
Writing an Internet Client Application Using MFC WinInet Classes

Writing an Internet Client Application Using MFC WinInet Classes

The basis of every Internet client application is the Internet session. MFC implements Internet sessions as objects of class CInternetSession. Using this class, you can create one Internet session or several simultaneous sessions.

To communicate with a server, you need a CInternetConnection object as well as a **CInternetSession**. You can create a **CInternetConnection** by using CInternetSession::GetFtpConnection, CInternetSession::GetHttpConnection, or CInternetSession::GetGopherConnection. Each of these calls is specific to the protocol type. These calls do not open a file on the server for reading or writing. If you intend to read or write data, you must open the file as a separate step.

For most Internet sessions, the **CInternetSession** object works hand-in-hand with a CInternetFile object:

- For an Internet session, you must create an instance of CInternetSession.
- If your Internet session reads or writes data, you must create an instance of **CInternetFile** (or its subclasses, CHttpFile or CGopherFile). The easiest way to read data is to call CInternetSession::OpenURL. This function parses a Universal Resource Locator (URL) supplied by you, opens a connection to the server specified by the URL, and returns a read-only **CInternetFile** object. **CInternetSession::OpenURL** is not specific to one protocol type — the same call works for any FTP, HTTP, or gopher URL. **CInternetSession::OpenURL** even works with local files (returning a **CStdioFile** instead of a **CInternetFile**).
- If your Internet session does not read or write data, but performs other tasks, such as deleting a file in an FTP directory, you may not need to create an instance of **CInternetFile**.

There are two ways to create a **CInternetFile** object:

- If you use **CInternetSession::OpenURL** to establish your server connection, the call to **OpenURL** returns a **CStdioFile**.
- If use **CInternetSession::GetFtpConnection**, **GetGopherConnection**, or **GetHttpConnection** to establish your server connection, you must call **CFtpConnection::OpenFile**, **CGopherConnection::OpenFile**, or **CHttpConnection::OpenRequest,** respectively, to return a **CInternetFile**, **CGopherFile**, or **CHttpFile**, respectively.

The steps in implementing an Internet client application vary depending on whether you create a generic Internet client based on **OpenURL** or a protocol-specific client using one of the **GetConnection** functions.

What do you want to know more about?

- How do I write an Internet client application that works generically with FTP, HTTP, and gopher?
- How do I write an FTP client application that opens a file?
- How do I write an FTP client application that does not open a file but performs a directory operation, such as deleting a file?
- How do I write a gopher client application?
- How do I write an HTTP client application?

See Also

Concepts

Win32 Internet Extensions (WinInet)
MFC Classes for Creating Internet Client Applications
Prerequisites for Internet Client Classes

Steps in a Typical Internet Client Application

The following table shows the steps you might perform in a typical Internet client application.

Your goal	Actions you take	Effects
Begin an Internet session.	Create a CInternetSession object.	Initializes WinInet and connects to server.
Read or set an Internet query option (time-out limit or number of retries, for example).	Use CInternetSession::QueryOption or CInternetSession::SetOption.	Returns FALSE if operation was unsuccessful.
Establish a callback function to monitor the status of the session.	Use CInternetSession::EnableStatusCallback.	Establishes a callback to CInternetSession::OnStatusCallback. Override **OnStatusCallback** to create your own callback routine.
Connect to an Internet server, intranet server, or local file.	Use CInternetSession::OpenURL.	Parses the URL and opens a connection to the specified server. Returns a CStdioFile (if you pass **OpenURL** a local file name). This is the object through which you access data retrieved from the server or file.
Read from the file.	Use CInternetFile::Read.	Reads the specified number of bytes using a buffer you supply.
Handle exceptions.	Use the CInternetException class.	Handles all common Internet exception types.
End the Internet session.	Dispose of the CInternetSession object.	Automatically cleans up open file handles and connections.

See Also

Concepts

Win32 Internet Extensions (WinInet)
Prerequisites for Internet Client Classes
Writing an Internet Client Application Using MFC WinInet Classes

Steps in a Typical FTP Client Application

A typical FTP client application creates a CInternetSession and a CFtpConnection object. Note that these MFC WinInet classes do not actually control the proxy type settings; IIS does.

Also, see these Knowledge Base articles:

- HOWTO: FTP with CERN-Based Proxy Using WinInet API (Article ID: Q166961)
- SAMPLE: FTP with CERN-Based Password Protected Proxy (Article ID: Q216214)
- Internet Services Manager Fails to Show Installed Proxy Services (Article ID: Q216802)

The following table shows the steps you might perform in a typical FTP client application.

Your goal	Actions you take	Effects
Begin an FTP session.	Create a CInternetSession object.	Initializes WinInet and connects to server.
Connect to an FTP server.	Use CInternetSession::GetFtpConnection.	Returns a CFtpConnection object.
Change to a new FTP directory on the server.	Use CFtpConnection::SetCurrentDirectory.	Changes the directory you are currently connected to on the server.
Find the first file in the FTP directory.	Use CFtpFileFind::FindFile.	Finds the first file. Returns FALSE if no files are found.
Find the next file in the FTP directory.	Use CFtpFileFind::FindNextFile.	Finds the next file. Returns FALSE if the file is not found.
Open the file found by **FindFile** or **FindNextFile** for reading or writing.	Use CFtpConnection::OpenFile, using the file name returned by FindFile or FindNextFile.	Opens the file on the server for reading or writing. Returns a CInternetFile object.
Read from or write to the file.	Use CInternetFile::Read or CInternetFile::Write.	Reads or writes the specified number of bytes, using a buffer you supply.
Handle exceptions.	Use the CInternetException class.	Handles all common Internet exception types.
End the FTP session.	Dispose of the CInternetSession object.	Automatically cleans up open file handles and connections.

See Also

Concepts

Win32 Internet Extensions (WinInet)
Prerequisites for Internet Client Classes
Writing an Internet Client Application Using MFC WinInet Classes

Steps in a Typical FTP Client Application to Delete a File

The following table shows the steps you might perform in a typical FTP client application that deletes a file.

Your goal	Actions you take	Effects
Begin an FTP session.	Create a CInternetSession object.	Initializes WinInet and connects to server.
Connect to an FTP server.	Use CInternetSession::GetFtpConnection.	Returns a CFtpConnection object.
Check to make sure you're in the right directory on the FTP server.	Use CFtpConnection::GetCurrentDirectory or CFtpConnection::GetCurrentDirectoryAsURL.	Returns the name or URL of the directory you are currently connected to on the server, depending on the member function selected.
Change to a new FTP directory on the server.	Use CFtpConnection::SetCurrentDirectory.	Changes the directory you are currently connected to on the server.
Find the first file in the FTP directory.	Use CFtpFileFind::FindFile.	Finds the first file. Returns FALSE if no files are found.
Find the next file in the FTP directory.	Use CFtpFileFind::FindNextFile.	Finds the next file. Returns FALSE if the file is not found.
Delete the file found by **FindFile** or **FindNextFile**.	Use CFtpConnection::Remove, using the file name returned by **FindFile** or **FindNextFile**.	Deletes the file on the server for reading or writing.
Handle exceptions.	Use the CInternetException class.	Handles all common Internet exception types.
End the FTP session.	Dispose of the CInternetSession object.	Automatically cleans up open file handles and connections.

See Also

Concepts

Win32 Internet Extensions (WinInet)
Prerequisites for Internet Client Classes
Writing an Internet Client Application Using MFC WinInet Classes

Steps in a Typical Gopher Client Application

The following table shows the steps you might perform in a typical gopher client application.

Your goal	Actions you take	Effects
Begin a gopher session.	Create a CInternetSession object.	Initializes WinInet and connects to server.
Connect to a gopher server.	Use CInternetSession::GetGopherConnection.	Returns a CGopherConnection object.
Find the first resource in the gopher.	Use CGopherFileFind::FindFile.	Finds the first file. Returns FALSE if no files are found.
Find the next resource in the gopher.	Use CGopherFileFind::FindNextFile.	Finds the next file. Returns FALSE if the file is not found.
Open the file found by **FindFile** or **FindNextFile** for reading.	Get a gopher locator using CGopherFileFind::GetLocator. Use CGopherConnection::OpenFile.	Opens the file specified by the locator. **OpenFile** returns a CGopherFile object.
Open a file using a gopher locator you supply.	Create a gopher locator using CGopherConnection::CreateLocator. Use CGopherConnection::OpenFile.	Opens the file specified by the locator. **OpenFile** returns a CGopherFile object.
Read from the file.	Use CGopherFile.	Reads the specified number of bytes, using a buffer you supply.
Handle exceptions.	Use the CInternetException class.	Handles all common Internet exception types.
End the gopher session.	Dispose of the CInternetSession object.	Automatically cleans up open file handles and connections.

See Also

Concepts

Win32 Internet Extensions (WinInet)
Prerequisites for Internet Client Classes
Writing an Internet Client Application Using MFC WinInet Classes

Steps in a Typical HTTP Client Application

The following table shows the steps you might perform in a typical HTTP client application:

Your goal	Actions you take	Effects
Begin an HTTP session.	Create a CInternetSession object.	Initializes WinInet and connects to server.
Connect to an HTTP server.	Use CInternetSession::GetHttpConnection.	Returns a CHttpConnection object.
Open an HTTP request.	Use CHttpConnection::OpenRequest.	Returns a CHttpFile object.
Send an HTTP request.	Use CHttpFile::AddRequestHeaders and CHttpFile::SendRequest.	Finds the file. Returns FALSE if the file is not found.
Read from the file.	Use CHttpFile.	Reads the specified number of bytes using a buffer you supply.
Handle exceptions.	Use the CInternetException class.	Handles all common Internet exception types.
End the HTTP session.	Dispose of the CInternetSession object.	Automatically cleans up open file handles and connections.

See Also

Concepts

Win32 Internet Extensions (WinInet)
Prerequisites for Internet Client Classes
Writing an Internet Client Application Using MFC WinInet Classes

Internet Server API (ISAPI) Extensions

Note
ATL Server provides a more efficient, configurable way of writing Web applications and XML Web services using C++ and IS API. For more information, see the documentation on ATL Server.

This family of articles provides an overview on the Internet server extensions and filters you can write using MFC. ISAPI, the Internet Server Application Programming Interface, is a set of APIs used to write OLE Server extensions and filters for Web servers like Internet Information Services. MFC provides five classes that wrap ISAPI. Topics included in this article are:

- What Is an Internet Server Extension?
- What Is an ISAPI Filter?
- What Kind of HTTP Server Is Needed to Run ISAPI?

What Is an Internet Server Extension?

An ISAPI server extension is a DLL that can be loaded and called by an HTTP server. Internet server extensions, also known as Internet server applications (ISAs), are used to enhance the capabilities of an Internet Server API (ISAPI)-compliant server. An ISA is invoked from a browser application and provides similar functionality to a Common Gateway Interface (CGI) application. For more information, see the articles Extending Web Servers with ISAPI and ISAPI Server Extensions.

Advantages of ISAPI Server Extensions

Users can fill out forms and click a submit button to send data to a Web server and invoke your ISA, which can process the information to provide custom content or store it in a database. Web server extensions can use information in a database to build Web pages dynamically, and then send them to the client computers to be displayed. Your application can add other custom functionality and provide data to the client using HTTP and HTML.

Both server extensions and filters run in the process space of the Web server, providing an efficient way to extend the server's capabilities.

What Is an ISAPI Filter?

An ISAPI filter is a DLL that runs on an ISAPI-enabled HTTP server to filter data traveling to and from the server. The filter registers for notification of events, such as logging on or URL mapping. When the selected events occur, the filter is called, and you can monitor and change the data (on its way from the server to the client or vice versa). ISAPI filters can be used to provide enhanced logging of HTTP requests (for example, to track who is logging on to your server), custom encryption, custom compression, or additional authentication methods. For more information, see ISAPI Filters.

What Kind of HTTP Server Is Needed to Run ISAPI?

To host Web sites, you must have an Internet server that supports the Hypertext Transfer Protocol (HTTP). If you have chosen an ISAPI-compliant Web server (for example, Microsoft Internet Information Services), you can take advantage of server extension DLLs to create small, fast Internet server applications.

Further Reading

For general information about ISAPI programming, see:

- Extending Web Servers with ISAPI
- MFC ISAPI Classes

For more information about ISAPI server extensions, see:

- ISAPI Server Extensions
- Parse Maps
- Creating a Typical ISAPI Extension

For information about ISAPI filters, see:

- ISAPI Filters
- Creating a Typical ISAPI Filter

For tips about ISAPI programming, see:

- ISAPI Programming Tips
- Debugging ISAPI Applications
- Building MFC ISAPI DLLs
- ISAPI and Databases

For information about Internet Information Services, see the following:

- "Internet Information Services Features" on the Windows 2000 web site at http://www.microsoft.com/windows2000/server/evaluation/features/web.asp (web links may change without notice).

See Also

Reference

CHttpServer Class
CHttpServerContext Class
CHttpFilter Class
CHttpFilterContext Class
CHtmlStream Class

Concepts

MFC Internet Programming Basics
How Does ISAPI Compare with CGI?
ISAPI Server Extensions and Filters

Extending Web Servers with ISAPI

This article describes some of the tasks you can perform with ISAPI Web server extensions, and how MFC supports writing them. Topics included in this article are:

- How Does ISAPI Make It Easier to Create Internet Server Applications?
- Advantages of Using MFC to Create Internet Server Applications
- What Can I Do with ISAPI?

How Does ISAPI Make It Easier to Create Internet Server Applications?

You can use ISAPI to write database applications, such as an order entry system or a custom catalog. You can collect information from a user with an HTML form, and send back another HTML page customized for the user. ISAPI server extensions provide similar functionality to CGI applications. For a comparison of the two methods, see How Does ISAPI Compare with CGI?

A convenient feature of the World Wide Web is that you don't have to install your code on the client machine to run your application. This makes upgrading easy. To run an ISAPI server extension DLL, a user types a URL in a browser, for example, `http://myserver/register.dll?`. The DLL runs on the server and sends the HTML data back to the client.

To add an updated DLL with new features, you stop the Web server service, replace the old DLL with your new version in its shared location on the server, and restart the service. The next client request will load the latest version. This process makes it easy to add new functionality, and you upgrade only one computer instead of many.

Advantages of Using MFC to Create Internet Server Applications

Visual C++ includes the ISAPI Extension Wizard to help you create ISAPI filters and servers. This wizard is one of the choices available on the Project tab in the New dialog box, which appears when you create a new project in the development environment. For more information about establishing a new project using wizards, see Creating a Project with a Visual C++ Application Wizard.

The ISAPI Extension Wizard makes it simple to generate a filter or a server extension. You can choose how (or if) to link to MFC; whether to have a filter, a server, or both; what filter notifications to handle at what priority; and whether the filter should get encrypted or unencrypted notifications or both. After the project is generated, you add custom functionality and create parse maps.

Error Handling

MFC adds failure protection beyond any provided by the server. MFC handles the returning of appropriate HTTP errors to the client (for example, bad request). For a list of some common HTTP return codes, see HTTP Basics. To handle error conditions in your program, see CHttpServer::OnParseError.

Forms Processing and Parse Maps

HTML-based forms are used to collect information from the user. This information is sent to your server via a URL that includes the name of your ISAPI DLL and a list of parameters, based on the information the user entered in the form. MFC makes parsing of these parameters easy: You write the function, declare its parameters, and use parse map macros to hook the command line arguments to your function parameters, including optional and default arguments. When a client invokes your server, MFC extracts the arguments and calls your function. MFC also returns results back to the client.

What Can I Do with ISAPI?

The following table lists some tasks you can perform using ISAPI server extensions and ISAPI filters.

ISAPI Server Extensions

I want to:	Use a server extension to:
Count users of my application.	Log information.
Write an order entry system.	Process an HTML form that is filled out at the client and processed on the server. Consider using DB Connector, a part of Internet Information Services, for simple data processing, and ISAPI for full database access.

ISAPI Filters

I want to:	Use a filter to:
Count visitors to my server.	Request **SF_NOTIFY_LOG** notification.
Do custom encryption.	**SF_NOTIFY_READ_RAW_DATA**, **SF_NOTIFY_WRITE_RAW_DATA**
Do custom compression.	**SF_NOTIFY_READ_RAW_DATA**, **SF_NOTIFY_WRITE_RAW_DATA**
Read raw data.	**SF_NOTIFY_READ_RAW_DATA**
Process header information.	Call **GetServerVariable**.
Authenticate a user.	Create a filter with high priority notification request, **SF_NOTIFY_AUTHENTICATION**.
Log requests from a certain user, or requests containing key words.	Request **SF_NOTIFY_URL_MAP** notification.

See Also

Concepts

Internet Server API (ISAPI) Extensions
MFC Internet Programming Basics

MFC ISAPI Classes

This article describes the MFC classes that wrap the Internet Server API (ISAPI). If your Internet server software is ISAPI-compliant, you can create Internet server extensions and filters with MFC. MFC provides five classes that wrap ISAPI to create and handle Internet server extensions and filters:

- CHttpServer An object to create and manage a server extension DLL, also known as an Internet server application (ISA). The ISA communicates with the server by means of the **EXTENSION_CONTROL_BLOCK** structure to handle client commands. Each client command corresponds to a member function of the **CHttpServer** object. The commands are parsed by the server's parse map macros. Only one **CHttpServer** object can exist for each DLL. This object contains the **EXTENSION_CONTROL_BLOCK** provided by the server. A new CHttpServerContext object is created for each separate client request.
- CHttpServerContext An object created by the **CHttpServer** object to handle a single request from a single client to the server object. Because the server may be handling concurrent, multiple requests, more than one active **CHttpServerContext** object can be associated with a particular **CHttpServer** instance.
- CHtmlStream An object created by the **CHttpServer** object to manage a buffer for responding to the client. **CHtmlStream** can contain begin, end, and content HTML tags to format the response.
- CHttpFilter An object you create to manage notification filters for incoming and outgoing client data. Like a **CHttpServer** object, only one **CHttpFilter** object can exist per DLL. When the **CHttpFilter** object is constructed, it creates a CHttpFilterContext object to manage a single notification from a single client. Because the server on which the filter is active may be handling concurrent, multiple requests, more than one active **CHttpFilterContext** object can be associated with a particular **CHttpFilter** instance.
- CHttpFilterContext An object created by the **CHttpFilter** object to handle a single notification for a single client.

CHttpServer and **CHttpFilter** can exist independently of each other; that is, you do not have to use a **CHttpFilter** with your **CHttpServer**, and vice versa. It is also possible for one of each to be in a single DLL.

See Also

Concepts

Internet Server API (ISAPI) Extensions
MFC Internet Programming Basics

ISAPI Filters

This article describes the benefits of ISAPI filters and how to use them with Internet servers. Topics included in this article are:

- How Filters Can Enhance a Server
- Filter Entry-Point Functions
- Types of Filter Notifications

The development environment includes an ISAPI Extension Wizard to help you create an ISAPI filter and set the appropriate notifications. For information on using the wizard, see the article Creating a Typical ISAPI Filter.

How Filters Can Enhance a Server

Use Internet Server API (ISAPI) filters to enhance Internet servers with custom features such as enhanced logging of HTTP requests, custom encryption and compression schemes, or new authentication methods. The filter application sits between the network connection to the clients and the HTTP server. Depending on the options that the filter application requests notification for, it can act on several server actions. Filters can be set to receive notification when selected events occur, including reading raw data from the client, processing the headers, managing communications over an encrypted port using Personal Communications Technology (PCT) or Secure Sockets Layer (SSL), or handling other stages in the processing of the HTTP request.

Filter Entry-Point Functions

Use the MFC class **CHttpFilter** to create a filter to manage the incoming and outgoing data for an ISAPI server. Two entry-point member functions, CHttpFilter::GetFilterVersion and CHttpFilter::HttpFilterProc, establish the filter and configure it for the notifications it will act upon.

GetFilterVersion

Using **CHttpFilter** requires that the filter application's path be inserted into the registry, in HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services/W3SVC/Parameters/FilterDLLs. When the server starts up, it reads this value and loads the DLLs listed. It then calls CHttpFilter::GetFilterVersion to perform three tasks:

- Exchange version information

  The MFC implementation of **GetFilterVersion** compares the version number for which you are creating your filter with the server's version number.

- Determine the requested events

  Register your ISAPI filter applications only for the events that you require. If you register for extraneous events, your choices can have a significant negative impact on the performance and scalability of your filter.

- Specify the priority to deliver the requested events

  The priority of requested events is specified by **GetFilterVersion** in the **dwFlags** member of the HTTP_FILTER_VERSION structure. After the **GetFilterVersion** exchange, every time the server processes one of the events, it will call any filters that have registered for that event. The calling order depends on priority.

HttpFilterProc

The second **CHttpFilter** member function to consider as an entry point is CHttpFilter::HttpFilterProc. As events happen, the server notifies each filter that registered for the event by calling the filter's **HttpFilterProc** entry point. MFC's implementation of **CHttpFilter::HttpFilterProc** determines which event was fired and calls one of the **CHttpFilter** members dedicated to handling the specific notification. You can override **HttpFilterProc** to provide special handling for events. To write code that reacts only to a particular event, see the next section, Types of Filter Notifications.

Types of Filter Notifications

When **CHttpFilter::HttpFilterProc** is called, the notifications received will determine which of the **CHttpFilter** member functions will be called. In your override of an individual **HttpFilterProc** member function, you can set the filter object to change data in a specific manner. For example, you can create an encryption or compression filter by overriding OnSendRawData and providing whatever special functionality your filter needs. Other overridable member functions include:

- OnPreprocHeaders Notifies the filter that the server has preprocessed the client headers.
- OnAuthentication Authenticates the client.
- OnUrlMap Notifies a filter when a server is mapping a logical URL to a physical path.
- OnSendRawData Notifies the filter before raw data is sent from the server to the client.
- OnReadRawData Notifies the filter after raw data is sent from the client to the server, but before the server processes it.
- OnLog Logs information to a server file.
- OnEndOfNetSession Notifies the filter that the session is ending.

See Also

Reference

CHttpFilter Class
CHttpFilterContext Class

Concepts

Internet Server API (ISAPI) Extensions
MFC Internet Programming Basics

Creating a Typical ISAPI Filter

Once you are familiar with the benefits of ISAPI filters and how to use them, you can create a filter project using the ISAPI Extension Wizard. For more information about starting a new project using wizards, see Creating a Project with a Visual C++ Application Wizard. After the project has been created, you add your custom notification handlers. The steps are described below.

To add your filter processing

1. Specify the notifications you want in **GetFilterVersion**.

   dwFlags will already contain the filters for events you specified in the ISAPI Extension Wizard.

2. To add a new notification, add the filter flag to **dwFlags** in CHttpFilter::GetFilterVersion.

   a. Add the filter function declaration to the .h file.

   b. Add the filter definition to the .cpp file.

   You can remove notification flags and functions as well.

3. To implement your filter functions, open the .cpp file and add custom handling for your chosen filter notifications by replacing the `//TODO` comments in the wizard-generated code.

4. To add your filter to the registry, specify the full path and the name of your DLL.

   HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services/W3SVC/Parameters/FilterDLLs is the registry key where filters are specified. Multiple entries are separated by commas and are loaded in the order specified.

To debug your filter, if running Microsoft Internet Information Services

- See Debugging ISAPI Applications for more information.

To load your filter, or to install a new version, if running Microsoft Internet Information Services

- Shut down the WWW service, copy the new version of your DLL over the old one, and restart the service.

See Also

Reference

CHttpFilter Class

CHttpFilterContext Class

Concepts

Internet Server API (ISAPI) Extensions

MFC Internet Programming Basics

ISAPI Server Extensions

This article describes the MFC classes used in writing ISAPI server extensions. Topics include:

- Classes Used in Writing an ISA
- Server Entry-Point Functions
- The EXTENSION_CONTROL_BLOCK Structure

For a detailed list of steps to create your ISAPI server extension, see Creating a Typical ISAPI Extension.

For an introduction to HTTP servers, see the article Internet Server API (ISAPI) Extensions. For a comparison of ISAPI and CGI, see How Does ISAPI Compare with CGI?.

Classes Used in Writing an ISA

To write an ISA in MFC, create a CHttpServer object. Each time the **CHttpServer** receives a client request, it creates a CHttpServerContext object. It passes a pointer to that object to the command handling function. The use of multiple **CHttpServerContext** objects permits the **CHttpServer** object to handle a single command from a single client to the server object. Because the server may be handling concurrent, multiple requests, more than one active **CHttpServerContext** object can be associated with a particular **CHttpServer** instance.

Server Entry-Point Functions

Use the MFC class **CHttpServer** to create and manage a server extension DLL. Your server extension DLL should expose two entry points as exported functions: CHttpServer::GetExtensionVersion and CHttpServer::HttpExtensionProc. An extension DLL generated by the ISAPI Extension Wizard will have a .def file that correctly exports these functions. The functions are implemented by MFC, which simply finds the single **CHttpExtension** instance in the DLL and calls its **CHttpServer::GetExtensionVersion** or **CHttpServer::HttpExtensionProc** implementation as appropriate.

CHttpServer::GetExtensionVersion

When the server application is invoked for the first time, the server calls CHttpServer::GetExtensionVersion to perform two tasks:

- Exchange version information

  The default implementation of **GetExtensionVersion** checks the version number for which you are creating your server extension against the server's version number.

- Provide a short text description of the ISA

  Override **GetExtensionVersion** to set the text string with your own description.

CHttpServer::HttpExtensionProc

The second **CHttpServer** member function to consider as an entry point is CHttpServer::HttpExtensionProc, which is similar to the **main** function of an application. It uses callback functions to read client data and determine the action to be taken. Usually, you won't take direct action with **HttpExtensionProc** because MFC automatically handles incoming client requests by sending them through a parse map. The parse map is used to identify which of your functions to call, and to extract the function parameters. For information on using the parse map macros, see the article Parse Maps.

When **HttpExtensionProc** finishes, it returns a value, which the server then formats and returns to the client. For a list of the values returned, see the description for HttpExtensionProc in the *MFC Reference*.

The default implementation of **HttpExtensionProc** is recommended; however, you can override it to provide custom implementation. For example, you could override this function to allocate per-call data.

The EXTENSION_CONTROL_BLOCK Structure

The HTTP server and the ISA communicate data through the **EXTENSION_CONTROL_BLOCK** structure. The information passed through the structure includes, for example, the raw query string, path information, method name, and translated path. MFC copies the **EXTENSION_CONTROL_BLOCK** pointer it receives from the server and passes it along in the CHttpServerContext object passed to command handling functions.

See Also

Concepts

Internet Server API (ISAPI) Extensions
MFC ISAPI Classes
ISAPI Filters
Creating a Typical ISAPI Filter
MFC Internet Programming Basics

Creating a Typical ISAPI Extension

After you have decided what your ISAPI Extension Server will do, you can get started by using the ISAPI Extension Wizard. For more information about starting a new project using wizards, see Creating a Project with a Visual C++ Application Wizard. After the project has been created, you add your custom functionality. The steps are described below.

To add your functions

1. For each command, write a function.

   MFC passes a pointer to a **CHttpServerContext** object when calling your function. Much of the information you want to work with is in the object's **EXTENSION_CONTROL_BLOCK** structure data member. You can also use member function callbacks to obtain additional header information, such as the user's IP address.

2. For each function, write a parse map entry for the required and optional parameters.

3. Override **CHttpExtensionProc**, if you want to do custom processing.

To install your DLL

- Copy your DLL into a directory for which users have execute privileges.

  For Microsoft Internet Information Services, you set execute permissions in Internet Service Manager.

Clients can now type the URL in the browser to invoke your DLL on the server.

See Also

Concepts

Internet Server API (ISAPI) Extensions
MFC Internet Programming Basics

Parse Maps

A parse map consists of code you write so that a **CHttpServer**-derived object maps client requests to its member functions. MFC provides parse map macros to help you do this. This article demonstrates the use of the macros through a continuing example of a single command: `RegisterUser`. It provides examples of using the parse map, including working with optional and required parameters. Topics included in this article are:

- The Parse Map Macros
- Designing Fill-out Forms for Use with ISAPI Parse Map Macros
- Processing HTML Forms Using Internet Server Extension DLLs and Command Handlers
- Declaring a Command in a Parse Map
- Passing a String from the URL
- Naming Parameters with ON_PARSE_COMMAND_PARAMS
- Managing Required and Optional Parameters

For more information on Internet server extension DLLs, which are also called Internet server applications (ISAs), see the article Internet Server API (ISAPI) Extensions. For more on the Internet Server API (ISAPI) classes wrapped by MFC, including **CHttpServer**, see the article MFC ISAPI Classes.

See Also

Reference

CHttpServer Class
CHttpServerContext Class
CHtmlStream Class

Concepts

Internet Server API (ISAPI) Extensions
MFC Internet Programming Basics

The Parse Map Macros

The parse map process includes five macros:

- BEGIN_PARSE_MAP Starts the definition of a parse map. It takes the member function's class and its base class as parameters.
- ON_PARSE_COMMAND Identifies a command. It takes the command, the member function's class, and a list of parameter types as parameters.
- ON_PARSE_COMMAND_PARAMS Defines the command's parameter list. It must follow immediately after **ON_PARSE_COMMAND**. It takes as its parameters the arguments accepted from the client.
- DEFAULT_PARSE_COMMAND Identifies the command used when no command is explicitly specified.
- END_PARSE_MAP Ends the definition of a parse map.

What do you want to know more about?

- Designing Fill-out Forms for Use with ISAPI Parse Map Macros
- Processing HTML Forms Using Internet Server Extension DLLs and Command Handlers
- Declaring a Command in a Parse Map
- Passing a String from the URL
- Naming Parameters with ON_PARSE_COMMAND_PARAMS
- Managing Required and Optional Parameters

See Also

Concepts

Parse Maps

Designing Fill-out Forms for Use with ISAPI Parse Map Macros

When a **CHttpServer** object receives a client request, it uses a set of macros called a parse map that map the request to the appropriate **CHttpServer** member functions. The client request can either be a server GET method or a server POST method. Other methods, like HEAD, are not supported. You can add your own support by overriding CHttpServer::HttpExtensionProc.

Authors of ISAPI forms are advised to use only the POST method because of browser inconsistencies, and because GET methods are limited to a 1024-byte buffer for their parameters. When writing forms for ISAPI, either use only the POST method, or design the ISA so that only the default function handles the form.

For example, some browsers sending a form via GET with an action of:

```
TestLet.DLL?Command
```

will truncate `Command` and send:

```
TestLet.DLL?name=value
```

instead of the correct command:

```
TestLet.DLL?Command?name=value
```

By truncating `Command`, the browser removes the association to the ISA function needed to map the request. Unless the function `Command` is the default function, the form will not be handled correctly.

What do you want to know more about?

- Processing HTML Forms Using Internet Server Extension DLLs and Command Handlers
- Declaring a Command in a Parse Map
- Passing a String from the URL
- Naming Parameters with ON_PARSE_COMMAND_PARAMS
- Managing Required and Optional Parameters

See Also

Concepts

Parse Maps

Processing HTML Forms Using Internet Server Extension DLLs and Command Handlers

Note

ATL Server provides a more efficient, configurable way of writing Web applications and XML Web services using C++ and IS API. For more information, see the documentation on ATL Server.

This topic describes how to use the **MfcISAPICommand** parameter in HTML forms.

Command Handlers

MFC's ISAPI classes for creating Internet server extension DLLs make use of "command handlers" — function names that are included in Uniform Resource Locators (URLs) and that allow a particular member function of a **CHttpServer** object to handle a request. Command handlers are defined with **PARSE_MAP** macros and are especially useful for processing HTML forms. For more information on defining command handlers, see the reference topic Internet Server API (ISAPI) Parse Maps.

Web Browser Inconsistencies

When creating your server extension DLLs and the Web pages that contain links or forms that call your extensions, you must keep in mind that although you control what happens on the server side, you do not have control over your users' Web browsers. For example, consider the following HTML form:

```
<FORM ACTION="http://moosegrrrl/scripts/MyExtension.dll?MyFunction" METHOD=GET>
Enter your full name: <INPUT NAME="name" VALUE="Anonymous">
<INPUT TYPE=SUBMIT VALUE="Register">
</FORM>
```

This form generates a request whose syntax depends on the individual browser. Microsoft Internet Explorer will send a command line such as:

```
http://moosegrrrl/scripts/MyExtension.dll?MyFunction?name=Anonymous
```

while Netscape Navigator will send:

```
http://moosegrrrl/scripts/MyExtension.dll?name=Anonymous
```

Microsoft Internet Explorer will append a question mark to the form's action if the form is being sent to the server by way of the GET method. It will not append a question mark to the action if the form's method is POST; however, there must be a question mark after the name of the extension DLL to indicate that the DLL is a script to be run, not an item to be downloaded. Netscape Navigator and later will, regardless of the method used to send a form or the syntax of the ACTION, always send a command line that contains one and only one question mark at the end of the action. Other versions of these browsers and browsers written by different companies may behave differently.

Using the MfcISAPICommand Parameter

To provide you with a simple way to get around the issue of browser inconsistencies and still retain the ability to make use of command handlers, MFC implements the **MfcISAPICommand** parameter. If `"MfcISAPICommand"` is the first available parameter, its value will be the command handler used by the **CHttpServer** object to process the request. If the first available parameter is not `"MfcISAPICommand"`, then the **CHttpServer** object will look to the first item on the command line following the name of the extension to determine which function to call, as it does in MFC 4.1 parameter processing.

Note

Unless the `"MfcISAPICommand"` parameter is at the very beginning of the form data, it is processed as an ordinary parameter.

The following HTML form shows the suggested usage of the "`MfcISAPICommand`" parameter, which enables your Internet server extension to correctly handle requests generated by either Microsoft or Netscape browsers:

```
<FORM ACTION=" http://moosegrrrl/scripts/MyExtension.dll" METHOD=GET>
<INPUT TYPE=HIDDEN NAME="MfcISAPICommand" VALUE="MyFunction">
Enter your full name: <INPUT NAME="name" VALUE="Anonymous">
<INPUT TYPE=SUBMIT VALUE="Register">
</FORM>
```

What do you want to know more about?

- Declaring a Command in a Parse Map
- Passing a String from the URL
- Naming Parameters with ON_PARSE_COMMAND_PARAMS
- Managing Required and Optional Parameters
- ATL Server

See Also

Concepts

Parse Maps

Declaring a Command in a Parse Map

The rest of this article is a continuing example of a parse map. The code examples demonstrate the parsing of a URL string, specifying default and required parameters, and using optional parameters. The example uses these elements:

Element	Name
Server	`MOOSE`
Server DLL	`SURVEY.DLL`
CHttpServer object	`CServerDerived`
Command	`RegisterUser`
User name	`Richard M. Jones`

The following parse map declares a single command, `RegisterUser`. This command accepts only one parameter: a pointer to a string:

```
BEGIN_PARSE_MAP( CServerDerived, CHttpServer )
ON_PARSE_COMMAND( RegisterUser, CServerDerived, ITS_PSTR )
END_PARSE_MAP( CServerDerived )
```

The corresponding command handler, `RegisterUser`, is a member of the `CServerDerived` class, and must be declared as follows:

```
CServerDerived::RegisterUser( CHttpServerContext* pCtxt, LPCTSTR pstrParameter )
{
   // do work here!
}
```

Note

The first parameter of the handling member function must be a pointer to a **CHttpServerContext** object.

What do you want to know more about?

- Passing a String from the URL
- Naming Parameters with ON_PARSE_COMMAND_PARAMS
- Managing Required and Optional Parameters

See Also

Concepts

Parse Maps

Passing a String from the URL

The next step in the example demonstrates invoking the command `RegisterUser` on the server `MOOSE`. The command `RegisterUser` is the name of a member function in the server DLL `SURVEY.DLL`. The client invokes `RegisterUser` from the following URL:

```
http://moose/survey/survey.dll?RegisterUser?Richard
```

This URL causes the member function `RegisterUser` to be called with *pstrParameter* pointing at the string "Richard".

If Richard wanted to register his middle initial and last name too, the URL would look like this:

```
http://moose/survey/survey.dll?RegisterUser?Richard+M%2E+Jones
```

The string contains multiple words and punctuation, so the server expands the normal plus sign and percent sign URL character escapes before passing the string to the parse map. The character escapes are automatically filled into the URL by the browser and are not specific to ISAPI. For more information about the URL character escapes, see the following Web site:

http://ietf.org/rfc/rfc1738.txt

What do you want to know more about?

- Naming Parameters with ON_PARSE_COMMAND_PARAMS
- Managing Required and Optional Parameters

See Also

Concepts

Parse Maps

Naming Parameters with ON_PARSE_COMMAND_PARAMS

This part of the example shows how to use the ON_PARSE_COMMAND_PARAMS macro to name parameters. Named parameters can appear in any order, as long as they precede all optional ones. They are useful in writing a function with a variable number of parameters for responding to a submit button on an HTML form. For example, you can create an HTML form to collect name, address, or additional information. When the user clicks the submit button on the form, a URL containing the information the user entered is passed to the server. Your parse maps can specify default values for optional fields. For information about HTML tags that specify how a form's content is sent to the server, see HTML Basics.

The `RegisterUser` function takes three parameters: `First`, `Middle`, and `Last`:

```
BEGIN_PARSE_MAP( CServerDerived, CHttpServer )
ON_PARSE_COMMAND( RegisterUser, CServerDerived, ITS_PSTR ITS_PSTR ITS_PSTR )
ON_PARSE_COMMAND_PARAMS("First Middle Last")
END_PARSE_MAP( CServerDerived )
```

The corresponding command handler is written like this:

```
CServerDerived::RegisterUser( CHttpServerContext* pctxt, LPCTSTR pstrFirst, LPCTSTR pstrMid
dle, LPCTSTR pstrLast )
{
   // do processing here!
}
```

For example, if the client uses a fill-in form with `METHOD=GET, ACTION="http://mooseboy/survey/survey.dll?RegisterUser"` and three text boxes named "First", "Middle", and "Last" to invoke `RegisterUser`, and the user enters the values "Richard", "M.", and "Jones", the URL will be formatted as follows:

```
http://mooseboy/survey/survey.dll?RegisterUser?First=Richard&Middle=M%2E&Last=Jones
```

The parse map looks at each `name=value` pair and assigns the value accordingly — all three parameters to `RegisterUser` will correctly resolve to "Richard", "M.", and "Jones", respectively.

What do you want to know more about?

- Managing Required and Optional Parameters

See Also

Reference

ON_PARSE_COMMAND
ON_PARSE_COMMAND_PARAMS

Concepts

Parse Maps

Managing Required and Optional Parameters

This part of the example examines the technique of setting the parse map's parameters as optional and how to handle data provided to an optional parameter. It also discusses the pitfalls of setting all parameters as required or all parameters as optional. Finally, it describes possible errors from failing to match the parameters with specified defaults.

Managing a Known Number of Parameters

In the example, the server `MOOSE` contains the DLL `SURVEY.DLL`, which contains the **CHttpServer** object `CServerDerived`. The client could use the following URL to invoke the `RegisterUser` command:

```
http://moose/survey/survey.dll?RegisterUser?Richard&M%2E&Jones
```

This command calls the member function `RegisterUser` with the parameters "Richard", "M.", and "Jones" for the *pstrFirst*, *pstrMiddle*, and *pstrLast* parameters, respectively.

Suppose the client used the following URL to call the `RegisterUser` function:

```
http://moose/survey/survey.dll?RegisterUser?Richard&Jones
```

Such a call would get an error from the server, because in this example all three parameters: *pstrFirst*, *pstrMiddle*, and *pstrLast*, are required. You can make parameters optional by specifying defaults for them in the ON_PARSE_COMMAND_PARAMS macro. This map makes all of the parameters optional:

```
BEGIN_PARSE_MAP(CServerDerived, CHttpServer)
ON_PARSE_COMMAND(RegisterUser, CServerDerived, ITS_PSTR ITS_PSTR ITS_PSTR)
ON_PARSE_COMMAND_PARAMS("First=~ Middle=~ Last=~")
END_PARSE_MAP(CServerDerived)
```

The default parameters specified above are tildes (~), an easily identifiable character you can test for in your handling function.

If the client invokes `RegisterUser` by using the following URL:

```
http://moose/survey/survey.dll?RegisterUser?Richard
```

when you have the above map installed, the second and third parameters are resolved as tildes.

If the client invokes `RegisterUser` by using the following URL:

```
http://moose/survey/survey.dll?RegisterUser?Richard&M%2E&Jones
```

when you have the above map installed, all three parameters to the `RegisterUser` function will resolve correctly.

You can make some parameters optional and some parameters required, but the required parameters must always appear first, before any optional parameters. For example, you can modify the map to indicate that only the parameter for the middle name is optional:

```
BEGIN_PARSE_MAP(CServerDerived, CHttpServer)
ON_PARSE_COMMAND(RegisterUser, CServerDerived, ITS_PSTR ITS_PSTR ITS_PSTR)
ON_PARSE_COMMAND_PARAMS("Last First Middle=~")
END_PARSE_MAP(CServerDerived)
```

Take care that the parameter types you specify match the parameter defaults you specify. You can accept both a string and an integer, for example:

```
BEGIN_PARSE_MAP(CServerDerived, CHttpServer)
ON_PARSE_COMMAND(Enlist, CServerDerived, ITS_PSTR ITS_I4)
ON_PARSE_COMMAND_PARAMS("Title=Developer Level=14")
END_PARSE_MAP(CServerDerived)
```

and produce the expected results. But if you were to reverse the two parameters, like this:

```
BEGIN_PARSE_MAP(CServerDerived, CHttpServer)
ON_PARSE_COMMAND(Enlist, CServerDerived, ITS_PSTR ITS_I4)
ON_PARSE_COMMAND_PARAMS("Level=14 Title=Developer")
END_PARSE_MAP(CServerDerived)
```

the default parameter for what you expected to be the level would be 0 and the title would be the string "14".

To add a space to a default string, enclose the string in single quotation marks. You do not need to escape special characters.

```
ON_PARSE_COMMAND_PARAMS("Title='Technical Writer' Level=14")
```

Managing an Unknown Number of Parameters

When passing an unknown number of parameters to the ISAPI extension, use the specification **ITS_RAW** or **ITS_ARGLIST** in the **ON_PARSE_COMMAND** macro.

Use **ITS_RAW** when you have no knowledge of the parameter data being passed.

For example, if your web page has a selection list from which one or more items can be selected, the following **ON_PARSE_COMMAND** specification is not appropriate, because its parameter list specifies that it receive exactly two pointers to strings:

```
ON_PARSE_COMMAND(Enlist, CServerDerived, ITS_PSTR ITS_PSTR)
```

If the ISAPI extension were to receive more or fewer than the specified number of parameters, an error will occur on the server. When the number of parameters to be passed is uncertain, you should call **ON_PARSE_COMMAND** with **ITS_RAW**. Note that you do not call **ON_PARSE_COMMAND_PARAMS** when parsing raw data. However, you must call **DEFAULT_PARSE_COMMAND** to specify the function that will pass the raw data:

```
BEGIN_PARSE_MAP(CDerivedExtension, CHttpServer)
DEFAULT_PARSE_COMMAND(Myfunc, CDerivedExtension)
ON_PARSE_COMMAND(Myfunc, CDerivedExtension, ITS_RAW)
END_PARSE_MAP(CDerivedExtension)
```

Use the following function prototype to send the data:

```
void CDerivedClass::Myfunc(CHttpServerContext* pCtxt, void* pVoid, DWORD dwBytes);
```

The *pVoid* pointer points to the data sent to your extension. The *dwBytes* parameter has a count of bytes at *pVoid*. If *dwBytes* is zero, it might indicate that *pVoid* points to nothing.

If the client invokes `RegisterUser` by using the following URL:

```
http://mooseboy/derived.dll?Myfunc&selection=10&selection=35&confirm=y
```

Myfunc would be called with *pVoid* pointing to the string "Myfunc&selection=10&selection=35&confirm=y" and *dwBytes* containing 42. Note that *dwBytes* does not count the trailing null byte in the data referenced by *pVoid*, because *pVoid* points to exactly what the client sent to the extension. The client might or might not have sent a trailing null byte, or it might have sent more than one null byte in the data.

Use **ITS_ARGLIST** when you know that the parameter data being passed can be parsed into **CHttpArg** objects.

When you use **ITS_ARGLIST**, the CHttpServer server object constructs a single CHttpArgList object containing zero or more CHttpArg objects. The **CHttpArg** objects represent, in order, the individual parameters passed in the URL that activated the **CHttpServer** command.

For example, the following URL:

```
http://mooseboy/yourext.dll&Arg1=hockey&Arg2&Arg3=beer+nuts
```

indicates the user's selections for three arguments: "hockey" for the first argument, nothing for the second argument, and a combination of "beer" and "nuts" for the third argument. The *m_pstrArg*, *m_pstrValue*, and *m_pstrRaw* values in the respective **CHttpArg** objects are set accordingly (note that the value of the second argument is an empty string). To see how the server parses these arguments into **CHttpArgList** and **CHttpArg** objects, see the diagram in CHttpArgList.

The **CHttpArgList** object is then sent to a parse map function. To specify this, the ON_PARSE_COMMAND macro in the parse map should use **ITS_ARGLIST** for the parameter type. Here's an example of a parse map that associates a command function `Enlist` with the **CHttpArgList** object `CHockeyServer`.

```
BEGIN_PARSE_MAP(CHockeyServer, CHttpServer)
ON_PARSE_COMMAND(Enlist, CHockeyServer, ITS_ARGLIST)
END_PARSE_MAP(CHockeyServer)
```

See Also

Reference

CHttpServer Class
CHttpServerContext Class
CHtmlStream Class
ON_PARSE_COMMAND
DEFAULT_PARSE_COMMAND

Concepts

Parse Maps
MFC Internet Programming Basics

ISAPI Programming Tips

It is important to remember that your ISAPI application is running in the context of a service on Windows NT. This means your code must be thread safe, it must be reentrant, and it has no user interface on the server.

- Watch your resource usage. Release acquired resources. For example, you might not be able to keep an ODBC connection for the life of the server.
- Be thread safe. Use synchronization objects to protect data. Don't use global variables.
- Write your filters to use the appropriate priority notification, at the lowest priority possible. A filter to authorize users may need to run at high priority. In this case, if a user fails authorization, the other lower-priority filter notifications will never be called.
- Be efficient in your filter processing. Filter functions can be called for all server requests, and an inefficient filter could slow down processing noticeably. If you are not using all the filter notifications you originally specified, alter **dwFlags** in **GetFilterVersion** to delete unneeded notifications.
- Do not display dialog boxes on the server, which may be running unattended with no one to click OK. Instead, log error information to a log file or a database.

Server Extension (ISA) Memory Management

If you want to allocate data on a per-call basis, you must use the data carefully within a synchronization object. The example below demonstrates overriding the **HttpExtensionProc** member function and using a critical section to associate per-call data with the **ConnID**. The base class is then called to process the request. Any member functions that use the data must protect it with a synchronization object. Upon return from **HttpExtensionProc** processing, the data for that **ConnID** is freed.

```
// in your .H file, declare variables and override HttpExtensionProc
class CISAPIWizExtension : public CHttpServer
{
    // per-call data
    CString* pData;
    CCriticalSection m_critSect;
    CMap < HCONN, HCONN&, CString*, CString* > m_map;

    // allocate per-call data
    DWORD HttpExtensionProc(EXTENSION_CONTROL_BLOCK* pECB);
};

// in your .CPP file, override HttpExtensionProc,
// to set per-call data for this connection
DWORD CISAPIWizExtension::HttpExtensionProc(EXTENSION_CONTROL_BLOCK* pECB)
{
    DWORD dwResult;
    // allocate per-call data
    // using the data type of your choice
    pData = new CString("whatever data you want");
    m_critSect.Lock();
    m_map.SetAt(pECB->ConnID, pData);
    m_critSect.Unlock();

    // call base class to do the work
    // remember, any member functions
    // that use the per-call data
    // must also access it within critical sections
    dwResult = CHttpServer::HttpExtensionProc(pECB);

    // delete per-call data
    m_critSect.Lock();
    m_map.Lookup(pECB->ConnID, pData);
    m_map.RemoveKey(pECB->ConnID);
    delete pData;
    m_critSect.Unlock();
    return dwResult;
```

```
}
```

Filter Memory Management

Most filters will register for the end of net session event. You can use this event to recycle any buffers used by that client request. For performance reasons, most filters will probably keep a pool of filter buffers and only allocate or free memory when the pool becomes empty or so large it no longer saves on the overhead of memory management. To allocate memory that is automatically freed when the communication with the client is terminated, use CHttpFilterContext::AllocMem. Using **AllocMem** presents a tradeoff: it is a valuable memory management tool, but it can affect performance, as noted earlier.

See Also

Concepts

Internet Server API (ISAPI) Extensions
Application Design Choices
Debugging ISAPI Applications
MFC Internet Programming Basics

Building MFC ISAPI DLLs

When you build an MFC ISAPI DLL, use the following guidelines to select which import libraries to build with:

Import library	When to use
eafxis.lib	When you dynamically link to the MFC DLL in a release build.
eafxisd.lib	When you dynamically link to the MFC DLL in a debug build.
nafxis.lib	When you statically link to the MFC DLL in a release build.
nafxisd.lib	When you statically link to the MFC DLL in a debug build.

Be aware that there is no DLL; all the ISAPI import libraries are for static linking.

See Also

Concepts

Internet Server API (ISAPI) Extensions

MFC Internet Programming Basics

Debugging ISAPI Applications

Preparing to debug your program requires careful attention to the setup steps.

Getting Ready to Debug Your Application

The general steps to prepare for debugging include:

- Creating a project file that sets up a workspace for the server application, with options to run the service interactively, and specifying a list of the DLLs you will be debugging.
- Setting up permissions in the Windows 2000 Computer Management tool (select **Administrative Tools** from **Control Panel**) or the Windows NT User Manager. You must log off and log back on for the permissions to take effect.
- Stopping and restarting the service.

If you follow the steps in Debugging an ISAPI Application carefully, you will be able to set breakpoints for any of your filters and extensions from one project file based on the Internet server program you are running. If your debugging session doesn't stop on the breakpoints you've set, go through the steps again. In particular, be sure you have installed the right files on the right computers and have set the appropriate permissions.

After you have successfully completed the setup described in Debugging an ISAPI Application, you can debug locally on your server. It is often useful to be able to debug remotely, from another machine, because you may not always have access to the server, or it may not have a development environment installed. To remotely debug an ISAPI DLL without logging on to the server directly, follow the steps described in Attaching to a Running Program or Multiple Programs.

See Also

Concepts

Internet Server API (ISAPI) Extensions
MFC Internet Programming Basics

ISAPI and Databases

You must verify that the database driver is thread safe: Not all are!

Drivers for Microsoft SQL Server 6.0 and later are thread safe. Check the documentation for other drivers.

Note that you can also use DB Connector, which comes with the Internet Information Services, to create front ends with simple forms and queries for your database applications. See the documentation for Internet Information Services for more information.

Create your database connection in an appropriate way. Remember that your DLL may remain in memory until the server shuts down, and you may need to reestablish your database connection at various times during the lifetime of the service.

When Should I Use ISAPI With a Database?

If you already have client/server solutions using SQL Server, you may be able to use them. If you are using HTML on the client side, you can continue to do so. You can also use DB Connector, which is a part of the Microsoft Internet Information Services that makes it easy for you to build simple front ends to run on your server. ISAPI provides access to the full power of SQL and is needed for more complex database queries.

Browsers handle form processing differently. For more information, see
Processing HTML Forms Using Internet Server Extension DLLs and Command Handlers.

See Also

Reference

TN067: Database Access from an ISAPI Server Extension

Concepts

Internet Server API (ISAPI) Extensions
MFC Internet Programming Basics

OLE in MFC

These articles explain the fundamentals of OLE programming using MFC. MFC provides the easiest way to write programs that use OLE:

- To use OLE visual editing (in-place activation).
- To work as OLE containers or servers.
- To implement drag-and-drop functionality.
- To work with date and time data.
- To manage the state data of MFC modules, including exported DLL function entry points, OLE/COM interface entry points, and window procedure entry points.

You can also use Automation or Remote Automation to operate another program from your program.

Note
The term OLE denotes the technologies associated with linking and embedding, including OLE containers, OLE servers, OLE it ems, in-place activation (or visual editing), trackers, drag and drop, and menu merging. The term Active applies to the Compo nent Object Model (COM) and COM-based objects such as ActiveX controls. OLE Automation is now called Automation.

In This Section

OLE Background

Discusses OLE and provides conceptual information about how it works.

Activation

Describes the role of activation in editing OLE items.

Containers

Provides links to using containers in OLE.

Data Objects and Data Sources

Provides links to topics discussing the use of the **COleDataObject** and **COleDataSource** classes.

Drag and Drop

Discusses using copying and pasting with OLE.

OLE Menus and Resources

Explains the use of menus and resources in MFC OLE document applications.

Registration

Discusses server installation and initialization.

Servers

Describes how to create OLE items (or components) for use by container applications.

Trackers

Provides information about the **CRectTracker** class, which provides a graphical interface to enable users to interact with OLE client items.

Related Sections

Adding Functionality

Provides links to topics describing conceptual information about the Visual C++ libraries and topics discussing various coding technologies and techniques.

Connection Points

Explains how to implement connection points (formerly known as OLE connection points) using the MFC classes **CCmdTarget** and **CConnectionPoint**.

Container/Server COM Components

Describes the steps necessary to incorporate optional advanced features into existing container applications.

The Component Object Model

Describes using OLE without MFC.

See Also

Other Resources

MFC Concepts

OLE Background

OLE is a mechanism that allows users to create and edit documents containing items or "objects" created by multiple applications.

Note
OLE was originally an acronym for Object Linking and Embedding. However, it is now referred to as OLE. Parts of OLE not related to linking and embedding are now part of Active technology.

OLE documents, historically called compound documents, seamlessly integrate various types of data, or components. Sound clips, spreadsheets, and bitmaps are typical examples of components found in OLE documents. Supporting OLE in your application allows your users to use OLE documents without worrying about switching between the different applications; OLE does the switching for you.

You use a container application to create compound documents and a server application or component application to create the items within the container document. Any application you write can be a container, a server, or both.

OLE incorporates many different concepts that all work toward the goal of seamless interaction between applications. These areas include the following:

Linking and Embedding

Linking and embedding are the two methods for storing items created inside an OLE document that were created in another application. For general information on the differences between the two, see the article OLE Background: Linking and Embedding. For more detailed information, see the articles Containers and Servers.

In-Place Activation (Visual Editing)

Activating an embedded item in the context of the container document is called in-place activation or visual editing. The container application's interface changes to incorporate the features of the component application that created the embedded item. Linked items are never activated in place because the actual data for the item is contained in a separate file, out of the context of the application containing the link. For more information on in-place activation, see the article Activation.

Note
Linking and embedding and in-place activation provide the main features of OLE visual editing.

Automation

Automation allows one application to drive another application. The driving application is known as an automation client, and the application being driven is known as an automation server or automation component. For more information on automation, see the articles Automation Clients and Automation Servers.

Note
Automation works in both OLE and Active technology contexts. You can automate any object based on COM.

Compound Files

Compound files provide a standard file format that simplifies structured storing of compound documents for OLE applications. Within a compound file, storages have many features of directories and streams have many features of files. This technology is also called structured storage. For more information on compound files, see the article Containers: Compound Files.

Uniform Data Transfer

Uniform Data Transfer (UDT) is a set of interfaces that allow data to be sent and received in a standard fashion, regardless of the actual method chosen to transfer the data. UDT forms the basis for data transfers by drag and drop. UDT now serves as the basis for existing Windows data transfer, such as the Clipboard and dynamic data exchange (DDE). For more information on UDT, see the article Data Objects and Data Sources (OLE).

Drag and Drop

Drag and drop is an easy-to-use, direct-manipulation technique to transfer data among applications, among windows within an application, or even within a single window in an application. The data to be transferred is selected and dragged to the desired destination. Drag and drop is based on uniform data transfer. For more information on drag and drop, see the article Drag and Drop.

Component Object Model

The Component Object Model (COM) provides the infrastructure used when OLE objects communicate with each other. The MFC OLE classes simplify COM for the programmer. COM is part of Active technology, because COM objects underlie both OLE and Active technology. For more information about COM, see the Active Template Library (ATL) topics.

Some of the more important OLE topics are covered in the following articles:

- OLE Background: Linking and Embedding
- OLE Background: Containers and Servers
- OLE Background: Implementation Strategies
- OLE Background: MFC Implementation

For information about handling context-sensitive Help in OLE applications, see the article Help: OLE Support for Help.

For general OLE information not found in the above articles, search for OLE in MSDN.

See Also

Concepts

OLE in MFC

OLE Background: Linking and Embedding

Using the Paste command in a container application can create an embedded component, or embedded item. The source data for an embedded item is stored as part of the OLE document that contains it. In this way, a document file for a word processor document can contain text and also can contain bitmaps, graphs, formulas, or any other type of data.

OLE provides another way to incorporate data from another application: creating a linked component, or linked item, or a link. The steps for creating a linked item are similar to those for creating an embedded item, except that you use the Paste Link command instead of the Paste command. Unlike an embedded component, a linked component stores a path to the original data, which is often in a separate file.

For example, if you are working in a word processor document and create a linked item to some spreadsheet cells, the data for the linked item is stored in the original spreadsheet document. The word processor document contains only the information specifying where the item can be found, that is, it contains a link to the original spreadsheet document. When you double-click the cells, the spreadsheet application is launched and the original spreadsheet document is loaded from where it was stored.

Every OLE item, whether embedded or linked, has a type associated with it based on the application that created it. For example, a Microsoft Paintbrush item is one type of item, and a Microsoft Excel item is another type. Some applications, however, can create more than one item type. For example, Microsoft Excel can create worksheet items, chart items, and macrosheet items. Each of these items can be uniquely identified by the system using a Class Identifier or **CLSID**.

See Also

Concepts

OLE Background
OLE Background: Containers and Servers
Containers: Client Items
Servers: Server Items

OLE Background: Containers and Servers

A container application is an application that can incorporate embedded or linked items into its own documents. The documents managed by a container application must be able to store and display OLE document components as well as the data created by the application itself. A container application must also allow users to insert new items or edit existing items by activating server applications when necessary. The user-interface requirements of a container application are listed in the article Containers: User-Interface Issues.

A server application or component application is an application that can create OLE document components for use by container applications. Server applications usually support drag and drop or copying their data to the Clipboard so that a container application can insert the data as an embedded or linked item. An application can be both a container and a server.

Most servers are stand-alone applications or full servers; they can either be run as stand-alone applications or can be launched by a container application. A miniserver is a special type of server application that can be launched only by a container. It cannot be run as a stand-alone application. Microsoft Draw and Microsoft Graph servers are examples of miniservers.

Containers and servers do not communicate directly. Instead, they communicate through the OLE system dynamic-link libraries (DLL). These DLLs provide functions that containers and servers call, and the containers and servers provide callback functions that the DLLs call.

Using this means of communication, a container does not need to know the implementation details of the server application. It allows a container to accept items created by any server without having to define the types of servers with which it can work. As a result, the user of a container application can take advantage of future applications and data formats. If these new applications are OLE components, then a compound document will be able to incorporate items created by those applications.

See Also

Concepts

OLE Background
OLE Background: MFC Implementation
Containers
Servers
Containers: Client Items
Servers: Server Items

OLE Background: Implementation Strategies

Depending on your application, there are four possible implementation strategies for adding OLE support:

- You are writing a new application.

  This situation usually requires the least work. You run the MFC Application Wizard and select either Advanced Features or Compound Document Support to create a skeleton application. For information on these options and what they do, see the article Creating an MFC EXE Program.

- You have a program written with the Microsoft Foundation Class Library version 2.0 or higher that does not support OLE.

  Create a new application with the MFC Application Wizard as previously mentioned, and then copy and paste the code from the new application into your existing application. This will work for servers, containers, or automated applications. See the MFC SCRIBBLE sample for an example of this strategy.

- You have a Microsoft Foundation Class Library program that implements OLE version 1.0 support.

  See MFC Technical Note 41 for this conversion strategy.

- You have an application that was not written using the Microsoft Foundation Classes and that may or may not have implemented OLE support.

  This situation requires the most work. One approach is to create a new application, as in the first strategy, and then copy and paste your existing code into it. If your existing code is written in C, then you may need to modify it so it can compile as C++ code. If your C code calls the Windows API, then you do not have to change it to use the Microsoft Foundation classes. This approach likely will require some restructuring of your program to support the document/view architecture used by versions 2.0 and higher of the Microsoft Foundation Classes. For more information on this architecture, see Technical Note 25.

Once you have decided on a strategy, you should either read the Containers or Servers articles (depending on the type of application you are writing) or examine the sample programs, or both. The MFC OLE samples OCLIENT and HIERSVR show how to implement the various aspects of containers and servers, respectively. At various points throughout these articles, you will be referred to certain functions in these samples as examples of the techniques being discussed.

See Also

Reference

MFC Application Wizard

Concepts

OLE Background
Containers: Implementing a Container
Servers: Implementing a Server

OLE Background: MFC Implementation

Because of the size and complexity of the raw OLE API, calling it directly to write OLE applications can be very time consuming. The goal of the Microsoft Foundation Class Library implementation of OLE is to reduce the amount of work you have to do to write full-featured, OLE-capable applications.

This article explains the parts of the OLE API that have not been implemented inside MFC. The discussion also explains how what is implemented maps to the OLE section of the Platform SDK.

Portions of OLE Not Implemented by the Class Library

A few interfaces and features of OLE are not directly provided by MFC. If you want to use these features, you can call the OLE API directly.

IMoniker Interface

The **IMoniker** interface is implemented by the class library (for example, the **COleServerItem** class) but has not previously been exposed to the programmer. For more information about this interface, see OLE Moniker Implementations in the OLE section of the Platform SDK. However, see also class CMonikerFile and CAsyncMonikerFile.

IUnknown and IMarshal Interfaces

The **IUnknown** interface is implemented by the class library but is not exposed to the programmer. The **IMarshal** interface is not implemented by the class library but is used internally. Automation servers built using the class library already have marshaling capabilities built in.

Docfiles (Compound Files)

Compound files are partially supported by the class library. None of the functions that directly manipulate compound files beyond creation are supported. MFC uses class **COleFileStream** to support manipulation of streams with standard file functions. For more information, see the article Containers: Compound Files.

In-Process Servers and Object Handlers

In-process servers and object handlers allow implementation of visual editing data or full Component Object Model (COM) objects in a dynamic-link library (DLL). To do this, you can implement your DLL by calling the OLE API directly. However, if you are writing an Automation server and your server has no user interface, you can use AppWizard to make your server an in-process server and put it completely into a DLL. For more information about these topics, see Automation Servers.

Tip
The easiest way to implement an Automation server is to place it in a DLL. MFC supports this approach.

For more information on how the Microsoft Foundation OLE classes implement OLE interfaces, see MFC Technical Notes 38, 39, and 40.

See Also

Concepts

OLE Background
OLE Background: Implementation Strategies

Activation (C++)

This article explains the role of activation in the visual editing of OLE items. After a user has embedded an OLE item in a container document, it may need to be used. To do this, the user double-clicks the item, which activates that item. The most frequent activity for activation is editing. Many current OLE items, when activated for editing, cause the menus and toolbars in the current frame window to change to reflect those belonging to the server application that created the item. This behavior, known as in-place activation, allows the user to edit any embedded item in a compound document without leaving the container document's window.

It is also possible to edit embedded OLE items in a separate window. This will happen if either the container or server application does not support in-place activation. In this case, when the user double-clicks an embedded item, the server application is launched in a separate window and the embedded item appears as its own document. The user edits the item in this window. When editing is complete, the user closes the server application and returns to the container application.

As an alternative, the user can choose "open editing" with the **<object> Open** command on the **Edit** menu. This opens the object in a separate window.

Note
Editing embedded items in a separate window was standard behavior in version 1 of OLE, and some OLE applications may s upport only this style of editing.

In-place activation promotes a document-centric approach to document creation. The user can treat a compound document as a single entity, working on it without switching between applications. However, in-place activation is used only for embedded items, not for linked items: they must be edited in a separate window. This is because a linked item is actually stored in a different place. The editing of a linked item takes place within the actual context of the data, that is, where the data is stored. Editing a linked item in a separate window reminds the user that the data belongs to another document.

MFC does not support nested in-place activation. If you build a container/server application, and that container/server is embedded in another container and in-place activated, it cannot in-place activate objects embedded inside it.

What happens to an embedded item when the user double-clicks it depends on the verbs defined for the item. For information, see Activation: Verbs.

See Also

Concepts

OLE in MFC
Containers
Servers

Activation: Verbs

This article explains the role primary and secondary verbs play in OLE activation.

Usually, double-clicking an embedded item allows the user to edit it. However, certain items do not behave this way. For example, double-clicking an item created with the Sound Recorder application does not open the server in a separate window; instead, it plays the sound.

The reason for this behavior difference is that Sound Recorder items have a different "primary verb." The primary verb is the action performed when the user double-clicks an OLE item. For most types of OLE items, the primary verb is Edit, which launches the server that created the item. For some types of items, such as Sound Recorder items, the primary verb is Play.

Many types of OLE items support only one verb, and Edit is the most common one. However, some types of items support multiple verbs. For example, Sound Recorder items support Edit as a secondary verb.

Another verb used frequently is Open. The Open verb is identical to Edit, except the server application is launched in a separate window. This verb should be used when either the container application or the server application does not support in-place activation.

Any verbs other than the primary verb must be invoked through a submenu command when the item is selected. This submenu contains all the verbs supported by the item and is usually reached by the *typename* **Object** command on the **Edit** menu. For information on the *typename* **Object** command, see the article Menus and Resources: Container Additions.

The verbs a server application supports are listed in the Windows registration database. If your server application is written with the Microsoft Foundation Class Library, it will automatically register all verbs when the server is started. If not, you should register them during the server application's initialization phase. For more information, see the article Registration.

See Also

Concepts

Activation (C++)
Containers
Servers

Containers

A container application is an application that can incorporate embedded or linked items into its own documents. The documents managed by a container application must be able to store and display OLE compound document components as well as data created by the application itself. A container application must also allow users to insert new items or edit existing items.

In This Section

Implement a Container

Container Client Items

Use Compound Files

Container User-Interface Issues

Advanced Features of Containers

See Also

Concepts

OLE in MFC
Servers
Activation (C++)
Menus and Resources (OLE)

Containers: Implementing a Container

This article summarizes the procedure for implementing a container and points you to other articles that provide more detailed explanations about implementing containers. It also lists some optional OLE features you may want to implement and the articles describing these features.

To prepare your CWinApp-derived class

1. Initialize the OLE libraries by calling **AfxOleInit** in the **InitInstance** member function.
2. Call **CDocTemplate::SetContainerInfo** in **InitInstance** to assign the menu and accelerator resources used when an embedded item is activated in-place. For more information on this topic, see Activation.

These features are provided for you automatically when you use the MFC Application Wizard to create a container application. See Creating an MFC EXE Program.

To prepare your view class

1. Keep track of selected items by maintaining a pointer, or list of pointers if you support multiple selection, to the selected items. Your **OnDraw** function must draw all OLE items.
2. Override **IsSelected** to check whether the item passed to it is currently selected.
3. Implement an **OnInsertObject** message handler to display the **Insert Object** dialog box.
4. Implement an **OnSetFocus** message handler to transfer focus from the view to an in-place active OLE embedded item.
5. Implement an **OnSize** message handler to inform an OLE embedded item that it needs to change its rectangle to reflect the change in size of its containing view.

Because the implementation of these features varies dramatically from one application to the next, the application wizard provides only a basic implementation. You will likely have to customize these functions to get your application to function properly. For an example of this, see the CONTAINER sample.

To handle embedded and linked items

1. Derive a class from COleClientItem. Objects of this class represent items that have been embedded in or linked to your OLE document.
2. Override **OnChange**, **OnChangeItemPosition**, and **OnGetItemPosition**. These functions handle sizing, positioning, and modifying embedded and linked items.

The application wizard will derive the class for you, but you will likely need to override **OnChange** and the other functions listed with it in step 2 in the preceding procedure. The skeleton implementations need to be customized for most applications, because these functions are implemented differently from one application to the next. For examples of this, see the MFC samples DRAWCLI and CONTAINER.

You must add a number of items to the container application's menu structure to support OLE. For more information on these, see Menus and Resources: Container Additions.

You may also want to support some of the following features in your container application:

- In-place activation when editing an embedded item.

  For more information, see Activation.

- Creation of OLE items by dragging and dropping a selection from a server application.

  For more information, see Drag and Drop (OLE).

- Links to embedded objects or combination container/server applications.

  For more information, see Containers: Advanced Features.

See Also

Concepts

Containers
Containers: Client Items

Containers: Client Items

This article explains what client items are and from what classes your application should derive its client items.

Client items are data items belonging to another application that are either contained in or referenced by an OLE container application's document. Client items whose data is contained within the document are embedded; those whose data is stored in another location referenced by the container document are linked.

The document class in an OLE application is derived from the class COleDocument rather than from **CDocument**. The **COleDocument** class inherits from **CDocument** all the functionality necessary for using the document/view architecture on which MFC applications are based. **COleDocument** also defines an interface that treats a document as a collection of **CDocItem** objects. Several **COleDocument** member functions are provided for adding, retrieving, and deleting elements of that collection.

Every container application should derive at least one class from **COleClientItem**. Objects of this class represent items, embedded or linked, in the OLE document. These objects exist for the life of the document containing them, unless they are deleted from the document.

CDocItem is the base class for **COleClientItem** and **COleServerItem**. Objects of classes derived from these two act as intermediaries between the OLE item and the client and server applications, respectively. Each time a new OLE item is added to the document, the MFC framework adds a new object of your client application's **COleClientItem**-derived class to the document's collection of **CDocItem** objects.

See Also

Reference

COleClientItem Class
COleServerItem Class

Concepts

Containers
Containers: Compound Files
Containers: User-Interface Issues
Containers: Advanced Features

Containers: Client-Item Notifications

This article discusses the overridable functions that the MFC framework calls when server applications modify items in your client application's document.

COleClientItem defines several overridable functions that are called in response to requests from the component application, which is also called the server application. These overridables usually act as notifications. They inform the container application of various events, such as scrolling, activation, or a change of position, and of changes that the user makes when editing or otherwise manipulating the item.

The framework notifies your container application of changes through a call to **COleClientItem::OnChange**, an overridable function whose implementation is required. This protected function receives two arguments. The first specifies the reason the server changed the item:

Notification	Meaning
OLE_CHANGED	The OLE item's appearance has changed.
OLE_SAVED	The OLE item has been saved.
OLE_CLOSED	The OLE item has been closed.
OLE_RENAMED	The server document containing the OLE item has been renamed.
OLE_CHANGED_STATE	The OLE item has changed from one state to another.
OLE_CHANGED_ASPECT	The OLE item's draw aspect has been changed by the framework.

These values are from the **OLE_NOTIFICATION** enumeration, which is defined in AFXOLE.H.

The second argument to this function specifies how the item has changed or what state it has entered:

When first argument is	Second argument
OLE_SAVED or **OLE_CLOSED**	Is not used.
OLE_CHANGED	Specifies the aspect of the OLE item that has changed.
OLE_CHANGED_STATE	Describes the state being entered (**emptyState**, **loadedState**, **openState**, **activeState**, or **activeUIState**).

For more information about the states a client item can assume, see Containers: Client-Item States.

The framework calls **COleClientItem::OnGetItemPosition** when an item is being activated for in-place editing. Implementation is required for applications that support in-place editing. The MFC Application Wizard provides a basic implementation, which assigns the item's coordinates to the **CRect** object that is passed as an argument to **OnGetItemPosition**.

If an OLE item's position or size changes during in-place editing, the container's information about the item's position and clipping rectangles must be updated and the server must receive information about the changes. The framework calls **COleClientItem::OnChangeItemPosition** for this purpose. The MFC Application Wizard provides an override that calls the base class's function. You should edit the function that the application wizard writes for your **COleClientItem**-derived class so that the function updates any information retained by your client-item object.

See Also

Reference

COleClientItem::OnChangeItemPosition

Concepts

Containers

Containers: Client-Item States

Containers: Client-Item States

This article explains the different states a client item passes through in its lifetime.

A client item passes through several states as it is created, activated, modified, and saved. Each time the item's state changes, the framework calls COleClientItem::OnChange with the **OLE_CHANGED_STATE** notification. The second parameter is a value from the **COleClientItem::ItemState** enumeration. It can be one of the following:

- **COleClientItem::emptyState**
- **COleClientItem::loadedState**
- **COleClientItem::openState**
- **COleClientItem::activeState**
- **COleClientItem::activeUIState**

In the empty state, a client item is not yet completely an item. Memory has been allocated for it, but it has not yet been initialized with the OLE item's data. This is the state a client item is in when it has been created through a call to **new** but has not yet undergone the second step of the typical two-step creation.

In the second step, performed through a call to **COleClientItem::CreateFromFile** or another **CreateFrom***xxxx* function, the item is completely created. The OLE data (from a file or some other source, such as the Clipboard) has been associated with the **COleClientItem**-derived object. Now the item is in the loaded state.

When an item has been opened in the server's window rather than opened in place in the container's document, it is in the open (or fully open) state. In this state, a cross-hatch usually is drawn over the representation of the item in the container's window to indicate that the item is active elsewhere.

When an item has been activated in place, it passes, usually only briefly, through the active state. It then enters the UI active state, in which the server has merged its menus, toolbars, and other user-interface components with those of the container. The presence of these user-interface components distinguishes the UI active state from the active state. Otherwise, the active state resembles the UI active state. If the server supports Undo, the server is required to retain the OLE item's undo-state information until it reaches the loaded or open state.

See Also

Reference

CRectTracker Class

Concepts

Containers

Activation (C++)

Containers: Client-Item Notifications

Trackers

Containers: Compound Files

This article explains the components and implementation of compound files and the advantages and disadvantages of using compound files in your OLE applications.

Compound files are an integral part of OLE. They are used to facilitate data transfer and OLE document storage. Compound files are an implementation of the Active structured storage model. Consistent interfaces exist that support serialization to a storage, a stream, or a file object. Compound files are supported in the Microsoft Foundation Class Library by the classes **COleStreamFile** and **COleDocument**.

Note
Using a compound file does not imply that the information comes from an OLE document or a compound document. Compound files are just one of the ways to store compound documents, OLE documents, and other data.

Components of a Compound File

The OLE implementation of compound files uses three object types: stream objects, storage objects, and **ILockBytes** objects. These objects are similar to the components of a standard file system in the following ways:

- Stream objects, like files, store data of any type.
- Storage objects, like directories, can contain other storage and stream objects.
- **LockBytes** objects represent the interface between the storage objects and the physical hardware. They determine how the actual bytes are written to whatever storage device the **LockBytes** object is accessing, such as a hard drive or an area of global memory. For more information about **LockBytes** objects and the **ILockBytes** interface, see the *OLE Programmer's Reference*.

Advantages and Disadvantages of Compound Files

Compound files provide benefits not available with earlier methods of file storage. They include:

- Incremental file accessing.
- File access modes.
- Standardization of file structure.

The potential disadvantages of compound files — large size and performance issues relating to storage on floppy discs — should be considered when deciding whether to use them in your application.

Incremental Access to Files

Incremental access to files is an automatic benefit of using compound files. Because a compound file can be viewed as a "file system within a file," individual object types, such as stream or storage, can be accessed without the need to load the entire file. This can dramatically decrease the time an application needs to access new objects for editing by the user. Incremental updating, based on the same concept, offers similar benefits. Instead of saving the entire file just to save the changes made to one object, OLE saves only the stream or storage object edited by the user.

File Access Modes

Being able to determine when changes to objects in a compound file are committed to disk is another benefit of using compound files. The mode in which files are accessed, either transacted or direct, determines when changes are committed.

- Transacted mode uses a two-phase commit operation to make changes to objects in a compound file, thereby keeping both the old and the new copies of the document available until the user chooses to either save or undo the changes.
- Direct mode incorporates changes to the document as they are made, without the ability to later undo them.

For more information about access modes, see the *OLE Programmer's Reference*.

Standardization

The standardized structure of compound files allows different OLE applications to browse through compound files created by your OLE application with no knowledge of the application that actually created the file.

Size and Performance Considerations

Because of the complexity of the compound file storage structure and the ability to save data incrementally, files using this format tend to be larger than other files using unstructured or "flat file" storage. If your application frequently loads and saves files, using compound files can cause the file size to increase much more quickly than noncompound files. Because compound files can get large, the access time for files stored on and loaded from floppy disks can also be affected, resulting in slower access to files.

Another issue that affects performance is compound-file fragmentation. The size of a compound file is determined by the difference between the first and last disk sectors used by the file. A fragmented file can contain many areas of free space that do not contain data, but are counted when calculating the size. During the lifetime of a compound file, these areas are created by the insertion or deletion of storage objects.

Using Compound Files Format for Your Data

After successfully creating an application that has a document class derived from **COleDocument**, ensure that your main document constructor calls **EnableCompoundFile**. When the application wizard creates OLE container applications, this call is inserted for you.

In the *OLE Programmer's Reference*, see IStream, IStorage, and ILockBytes.

See Also

Reference

COleStreamFile Class

COleDocument Class

Concepts

Containers

Containers: User-Interface Issues

Containers: User-Interface Issues

You must add a number of features to a container application's user interface to adequately manage linked and embedded items. These features involve changes to the menu structure and to the events that the application handles. For detailed information about them, see the following articles:

For information on	See
Menu additions for containers	Menus and Resources: Container Additions
Additional resources for containers	Menus and Resources: Container Additions
Painting linked or embedded items	Container sample
New dialog boxes for containers	Dialog Boxes in OLE

See Also

Concepts

Containers
Containers: Advanced Features
Menus and Resources (OLE)

Containers: Advanced Features

This article describes the steps necessary to incorporate optional advanced features into existing container applications. These features are:

- An application that is both a container and a server
- An OLE link to an embedded object

Creating a Container/Server Application

A container/server application is an application that acts as both a container and a server. Microsoft Word for Windows is an example of this. You can embed Word for Windows documents in other applications, and you can also embed items in Word for Windows documents. The process for modifying your container application to be both a container and a full server (you cannot create a combination container/miniserver application) is similar to the process for creating a full server.

The article Servers: Implementing a Server lists a number of tasks required to implement a server application. If you convert a container application to a container/server application, then you need to perform some of those same tasks, adding code to the container. The following lists the important things to consider:

- The container code created by the application wizard already initializes the OLE subsystem. You will not need to change or add anything for that support.
- Wherever the base class of a document class is **COleDocument**, change the base class to **COleServerDoc**.
- Override **COleClientItem::CanActivate** to avoid editing items in place while the server itself is being used to edit in place.

  For example, the MFC OLE sample OCLIENT has embedded an item created by your container/server application. You open the OCLIENT application and in-place edit the item created by your container/server application. While editing your application's item, you decide you want to embed an item created by the MFC OLE sample HIERSVR. To do this, you cannot use in-place activation. You must fully open HIERSVR to activate this item. Because the Microsoft Foundation Class Library does not support this OLE feature, overriding **COleClientItem::CanActivate** allows you to check for this situation and prevent a possible run-time error in your application.

If you are creating a new application and want it to function as a container/server application, choose that option in the OLE Options dialog box in the application wizard and this support will be created automatically. For more information, see the article Overview: Creating an ActiveX Control Container. For information about MFC samples, see MFC Samples.

Note that you cannot insert an MDI application into itself. An application that is a container/server cannot be inserted into itself unless it is an SDI application.

Links to Embedded Objects

The Links to Embedded Objects feature enables a user to create a document with an OLE link to an embedded object inside your container application. For example, create a document in a word processor containing an embedded spreadsheet. If your application supports links to embedded objects, it could paste a link to the spreadsheet contained in the word processor's document. This feature allows your application to use the information contained in the spreadsheet without knowing where the word processor originally got it.

To link to embedded objects in your application

1. Derive your document class from **COleLinkingDoc** instead of **COleDocument**.
2. Create an OLE class ID (**CLSID**) for your application by using the Class ID Generator included with the OLE Development Tools.
3. Register the application with OLE.
4. Create a **COleTemplateServer** object as a member of your application class.
5. In your application class's **InitInstance** member function, do the following:
   - Connect your **COleTemplateServer** object to your document templates by calling the object's **ConnectTemplate** member function.

- Call the **COleTemplateServer::RegisterAll** member function to register all class objects with the OLE system.
- Call **COleTemplateServer::UpdateRegistry**. The only parameter to **UpdateRegistry** should be **OAT_CONTAINER** if the application is not launched with the "/Embedded" switch. This registers the application as a container that can support links to embedded objects.

  If the application is launched with the "/Embedded" switch, it should not show its main window, similar to a server application.

The MFC OLE sample OCLIENT implements this feature. For an example of how this is done, see the **InitInstance** function in the OCLIENT.CPP file of this sample application.

See Also

Concepts

Containers
Servers

Data Objects and Data Sources (OLE)

When you perform a data transfer, either by using the Clipboard or drag and drop, the data has a source and a destination. One application provides the data for copying and another application accepts it for pasting. Each side of the transfer needs to perform different operations on the same data for the transfer to succeed. The Microsoft Foundation Class (MFC) Library provides two classes that represent each side of this transfer:

- Data sources (as implemented by **COleDataSource** objects) represent the source side of the data transfer. They are created by the source application when data is to be copied to the Clipboard, or when data is provided for a drag-and-drop operation.
- Data objects (as implemented by **COleDataObject** objects) represent the destination side of the data transfer. They are created when the destination application has data dropped into it, or when it is asked to perform a paste operation from the Clipboard.

The following articles explain how to use data objects and data sources in your applications. This information applies to both container and server applications, because both may be called upon to copy and paste data.

- Data Objects and Data Sources: Creation and Destruction
- Data Objects and Data Sources: Manipulation

In This Section

Drag and Drop

Clipboard

See Also

Reference

COleDataObject Class

COleDataSource Class

Concepts

OLE in MFC

Data Objects and Data Sources: Creation and Destruction

As explained in the article Data Objects and Data Sources (OLE), data objects and data sources represent both sides of a data transfer. This article explains when to create and destroy these objects and sources to perform your data transfers properly, including:

- Creating data objects
- Destroying data objects
- Creating data sources
- Destroying data sources

Creating Data Objects

Data objects are used by the destination application — either the client or the server. A data object in the destination application is one end of a connection between the source application and the destination application. A data object in the destination application is used to access and interact with the data in the data source.

There are two common situations where a data object is needed. The first situation is when data is dropped in your application using drag and drop. The second situation is when Paste or Paste Special is chosen from the Edit menu.

In a drag-and-drop situation, you do not need to create a data object. A pointer to an existing data object will be passed to your **OnDrop** function. This data object is created by the framework as part of the drag-and-drop operation and will also be destroyed by it. This is not always the case when pasting is done by another method. For more information, see Destroying Data Objects.

If the application is performing a paste or paste special operation, you should create a **COleDataObject** object and call its **AttachClipboard** member function. This associates the data object with the data on the Clipboard. You can then use this data object in your paste function.

For an example of how this is done, see the **DoPasteItem** function in the MAINVIEW.CPP file that is part of the MFC OLE sample OCLIENT. OCLIENT implements a function that performs all paste operations and calls **DoPasteItem** from its **OnDrop**, **OnPaste**, and **OnPasteLink** functions. Because **OnDrop** has a pointer to a data object passed to it, it passes the pointer on to **DoPasteItem**. **OnPaste** and **OnPasteLink** pass **NULL** for this parameter, telling **DoPasteItem** to create a data object and attach it to the Clipboard. This scheme separates your paste code so you only have to debug it in one place, but you can still use it for both kinds of paste operations.

Destroying Data Objects

If you follow the scheme described in Creating Data Objects, destroying data objects is a trivial aspect of data transfers. The data object that was created in your paste function will be destroyed by MFC when your paste function returns.

If you follow another method of handling paste operations, make sure the data object is destroyed after your paste operation is complete. Until the data object is destroyed, it will be impossible for any application to successfully copy data to the Clipboard.

Creating Data Sources

Data sources are used by the source of the data transfer, which can be either the client or the server side of the data transfer. A data source in the source application is one end of a connection between the source application and the destination application. A data object in the destination application is used to interact with the data in the data source.

Data sources are created when an application needs to copy data to the Clipboard. A typical scenario runs like this:

1. The user selects some data.
2. The user chooses **Copy** (or **Cut**) from the **Edit** menu or begins a drag-and-drop operation.
3. Depending on the design of the program, the application creates either a **COleDataSource** object or an object from a class derived from **COleDataSource**.
4. The selected data is inserted into the data source by calling one of the functions in the **COleDataSource::CacheData** or **COleDataSource::DelayRenderData** groups.
5. The application calls the **SetClipboard** member function (or the **DoDragDrop** member function if this is a drag-and-drop operation) belonging to the object created in step 3.

6. If this is a **Cut** operation or **DoDragDrop** returns **DROPEFFECT_MOVE**, the data selected in step 1 is deleted from the document.

This scenario is implemented by the MFC OLE samples OCLIENT and HIERSVR. Look at the source for each application's **CView**-derived class for all but the **GetClipboardData** and **OnGetClipboardData** functions. These two functions are in either the **COleClientItem** or **COleServerItem**-derived class implementations. These sample programs provide a good example of how to implement these concepts.

One other situation in which you might want to create a **COleDataSource** object occurs if you are modifying the default behavior of a drag-and-drop operation. For more information, see the Drag and Drop: Customizing article.

Destroying Data Sources

Data sources must be destroyed by the application currently responsible for them. In situations where you hand the data source to OLE, such as calling **COleDataSource::SetClipboard**, you need to call **pDataSrc->InternalRelease**. For example:

```
void CDragtestView::OnBegindrag(NMHDR* pNMHDR, LRESULT* pResult)
{
   HRESULT hr= S_OK;

   CTestDataSource *pDataSrc= new CTestDataSource();

      pDataSrc->DelayRenderData( RegisterClipboardFormat(_T("TIGroupFiles")) );

   if (GetListCtrl().GetSelectionMark() == 0)
   {
      TRACE(_T("mfc-style drag-drop \n"));
      pDataSrc->DoDragDrop();
      pDataSrc->InternalRelease();

   }
   else
   {
      TRACE(_T("raw-style drag-drop \n"));
      IDataObject *pDataObj;
      CComObject<CDataObj>* pDataObjImpl;
      hr= CreateInstanceQI(&pDataObjImpl, IID_IDataObject, (LPVOID*)&pDataObj);
      if (FAILED(hr)) throw hr;

      IDropSource *pDropSrc;
      CComObject<CDropSource>* pDropSrcImpl;
      hr= CreateInstanceQI(&pDropSrcImpl, IID_IDropSource, (LPVOID*)&pDropSrc);
      if (FAILED(hr)) throw hr;

      DWORD dwEffect= DROPEFFECT_NONE;

      hr= DoDragDrop(pDataObj, pDropSrc, DROPEFFECT_COPY, &dwEffect);

      delete pDataObjImpl;
      delete pDropSrcImpl;
      TRACE(_T("DoDragDrop ret 0x%08x \n"), hr);
   }
   //delete pDataSrc;
}
```

If you have not handed your data source to OLE, then you are responsible for destroying it, as with any typical C++ object.

For more information, see Drag and Drop, Clipboard, and Manipulating Data Objects and Data Sources.

See Also

Reference

COleDataObject Class

COleDataSource Class

Concepts

Data Objects and Data Sources (OLE)

Data Objects and Data Sources: Manipulation

After a data object or data source has been created, you can perform a number of common operations on the data, such as inserting and removing data, enumerating the formats the data is in, and more. This article describes the techniques necessary to complete the most common operations. Topics include:

- Inserting data into a data source
- Determining the formats available in a data object
- Retrieving data from a data object

Inserting Data into a Data Source

How data is inserted into a data source depends on whether the data is supplied immediately or on demand, and in which medium it is supplied. The possibilities are as follows.

Supplying Data Immediately (Immediate Rendering)

- Call **COleDataSource::CacheGlobalData** repeatedly for every Clipboard format in which you are supplying data. Pass the Clipboard format to be used, a handle to the memory containing the data and, optionally, a **FORMATETC** structure describing the data.

  -or-

- If you want to work directly with **STGMEDIUM** structures, you call **COleDataSource::CacheData** instead of **COleDataSource::CacheGlobalData** in the option above.

Supplying Data on Demand (Delayed Rendering)

This is an advanced topic.

- Call **COleDataSource::DelayRenderData** repeatedly for every Clipboard format in which you are supplying data. Pass the Clipboard format to be used and, optionally, a **FORMATETC** structure describing the data. When the data is requested, the framework will call **COleDataSource::OnRenderData**, which you must override.

  -or-

- If you use a **CFile** object to supply the data, call **COleDataSource::DelayRenderFileData** instead of **COleDataSource::DelayRenderData** in the previous option. When the data is requested, the framework will call **COleDataSource::OnRenderFileData**, which you must override.

Determining the Formats Available in a Data Object

Before an application allows the user to paste data into it, it needs to know if there are formats on the Clipboard that it can handle. To do this, your application should do the following:

1. Create a **COleDataObject** object and a **FORMATETC** structure.
2. Call the data object's **AttachClipboard** member function to associate the data object with the data on the Clipboard.
3. Do one of the following:
   - Call the data object's **IsDataAvailable** member function if there are only one or two formats you need. This will save you time in cases where the data on the Clipboard supports significantly more formats than your application.

     -or-

   - Call the data object's **BeginEnumFormats** member function to start enumerating the formats available on the Clipboard. Then call **GetNextFormat** until the Clipboard returns a format your application supports or there are no more formats.

If you are using **ON_UPDATE_COMMAND_UI**, you can now enable the Paste and, possibly, Paste Special items on the Edit menu. To do this, call either **CMenu::EnableMenuItem** or **CCmdUI::Enable**. For more information about what container applications should do with menu items and when, see Menus and Resources: Container Additions.

Retrieving Data from a Data Object

Once you have decided on a data format, all that remains is to retrieve the data from the data object. To do this, the user decides where to put the data, and the application calls the appropriate function. The data will be available in one of the following mediums:

Medium	**Function to call**
Global Memory (**HGLOBAL**)	**COleDataObject::GetGlobalData**
File (**CFile**)	**COleDataObject::GetFileData**
STGMEDIUM structure (**IStorage**)	**COleDataObject::GetData**

Commonly, the medium will be specified along with its Clipboard format. For example, a **CF_EMBEDDEDSTRUCT** object is always in an **IStorage** medium that requires an **STGMEDIUM** structure. Therefore, you would use **GetData** because it is the only one of these functions that can accept an **STGMEDIUM** structure.

For cases where the Clipboard format is in an **IStream** or **HGLOBAL** medium, the framework can provide a **CFile** pointer that references the data. The application can then use file read to get the data in much the same way as it might import data from a file. Essentially, this is the client-side interface to the **OnRenderData** and **OnRenderFileData** routines in the data source.

The user can now insert data into the document just like for any other data in the same format.

What do you want to know more about?

- Drag and drop
- Clipboard

See Also

Reference

COleDataObject Class
COleDataSource Class

Concepts

Data Objects and Data Sources (OLE)

Drag and Drop (OLE)

The drag-and-drop feature of OLE is primarily a shortcut for copying and pasting data. When you use the Clipboard to copy or paste data, a number of steps are required. You select the data, click **Cut** or **Copy** from the **Edit** menu, move to the destination file, window or application, place the cursor in the desired location, and click **Paste** from the **Edit** menu.

OLE drag and drop is different from the File Manager drag-and-drop mechanism, which can only handle filenames and is designed specifically to pass filenames to applications. OLE drag and drop is much more general. It allows you to drag and drop any data that could also be placed on the Clipboard.

When you use OLE drag and drop, you remove two steps from the process. You select the data from the source window (the "drop source"), drag it to the desired destination (the "drop target"), and drop it by releasing the mouse button. The operation eliminates the need for menus and is quicker than the copy/paste sequence. The only requirement is that both the drop source and drop target must be open and at least partially visible on the screen.

Using OLE drag and drop, data can be transferred from one location to another within a document, between different documents, or between applications. It can be implemented in either a container or a server application, and any application can be a drop source, a drop target, or both. If an application has both drop-source and drop-target support implemented, drag and drop is enabled between child windows, or within one window. This feature can make your application much easier to use.

If you only want to use the drag-and-drop capabilities of OLE, see Drag and Drop: Customizing. You can use the techniques explained in that article to make non-OLE applications drop sources. The article Drag and Drop: Implementing a Drop Target describes how to implement drop-target support for both OLE and non-OLE applications. It will also be helpful to examine the MFC OLE samples OCLIENT and HIERSVR.

If you have not read the Data Objects and Data Sources (OLE) family of articles, you may want to do so now. These articles explain the fundamentals of data transfer, and how to implement it in your applications.

For more information about drag and drop, see:

- Drag and Drop: Implementing a Drop Source
- Drag and Drop: Implementing a Drop Target
- Drag and Drop: Customizing

See Also

Concepts

OLE in MFC
Data Objects and Data Sources (OLE)

Drag and Drop: Implementing a Drop Source

This article explains how to get your application to provide data to a drag-and-drop operation.

Basic implementation of a drop source is relatively simple. The first step is to determine what events begin a drag operation. Recommended user interface guidelines define the beginning of a drag operation as the selection of data and a **WM_LBUTTONDOWN** event occurring on a point inside the selected data. The MFC OLE samples OCLIENT and HIERSVR follow these guidelines.

If your application is a container and the selected data is a linked or an embedded object of type **COleClientItem**, call its **DoDragDrop** member function. Otherwise, construct a **COleDataSource** object, initialize it with the selection, and call the data source object's **DoDragDrop** member function. If your application is a server, use **COleServerItem::DoDragDrop**. For information about customizing standard drag-and-drop behavior, see the article Drag and Drop: Customizing.

If **DoDragDrop** returns **DROPEFFECT_MOVE**, delete the source data from the source document immediately. No other return value from **DoDragDrop** has any effect on a drop source.

For more information, see:

- Implementing a Drop Target
- Customizing Drag and Drop
- Creating and Destroying OLE Data Objects and Data Sources
- Manipulating OLE Data Objects and Data Sources

See Also

Reference

COleDataSource::DoDragDrop
COleClientItem::DoDragDrop
CView::OnDragLeave

Concepts

Drag and Drop (OLE)

Drag and Drop: Implementing a Drop Target

This article outlines how to make your application a drop target. Implementing a drop target takes slightly more work than implementing a drop source, but it is still relatively simple. These techniques also apply to non-OLE applications.

To implement a drop target

1. Add a member variable to each view in the application that you want to be a drop target. This member variable must be of type **COleDropTarget** or a class derived from it.
2. From your view class's function that handles the **WM_CREATE** message (typically **OnCreate**), call the new member variable's **Register** member function. **Revoke** will be called automatically for you when your view is destroyed.
3. Override the following functions. If you want the same behavior throughout your application, override these functions in your view class. If you want to modify behavior in isolated cases or want to enable dropping on non-**CView** windows, override these functions in your **COleDropTarget**-derived class.

Override	To allow
OnDragEnter	Drop operations to occur in the window. Called when the cursor first enters the window.
OnDragLeave	Special behavior when the drag operation leaves the specified window.
OnDragOver	Drop operations to occur in the window. Called when the cursor is being dragged across the window.
OnDrop	Handling of data being dropped into the specified window.
OnScrollBy	Special behavior for when scrolling is necessary in the target window.

See the MAINVIEW.CPP file that is part of the MFC OLE sample OCLIENT for an example of how these functions work together.

For more information, see:

- Implementing a Drop Source
- Creating and Destroying OLE Data Objects and Data Sources
- Manipulating OLE Data Objects and Data Sources

See Also

Reference

COleDropTarget Class

Concepts

Drag and Drop (OLE)

Drag and Drop: Customizing

The default implementation of the drag-and-drop feature is sufficient for most applications. However, some applications may require that this standard behavior be changed. This article explains the steps necessary to change these defaults. In addition, you can use this technique to establish applications that do not support compound documents as drop sources.

If you are customizing standard OLE drag-and-drop behavior, or you have a non-OLE application, you must create a **COleDataSource** object to contain the data. When the user starts a drag-and-drop operation, your code should call the **DoDragDrop** function from this object instead of from other classes that support drag-and-drop operations.

Optionally, you can create a **COleDropSource** object to control the drop and override some of its functions depending on the type of behavior you want to change. This drop-source object is then passed to **COleDataSource::DoDragDrop** to change the default behavior of these functions. These different options allow a great deal of flexibility in how you support drag-and-drop operations in your application. For more information about data sources, see the article Data Objects and Data Sources (OLE).

You can override the following functions to customize drag-and-drop operations:

Override	To customize
OnBeginDrag	How dragging is initiated after you call **DoDragDrop**.
GiveFeedback	Visual feedback, such as cursor appearance, for different drop results.
QueryContinueDrag	The termination of a drag-and-drop operation. This function enables you to check modifier key states durin g the drag operation.

See Also

Reference

COleDropSource Class

COleDataSource Class

Concepts

Drag and Drop (OLE)

Menus and Resources (OLE)

This group of articles explains the use of menus and resources in MFC OLE document applications.

OLE visual editing places additional requirements on the menu and other resources provided by OLE document applications because there are a number of modes in which both container and server (component) applications can be started and used. For example, a full-server application can run in any of these three modes:

- Stand alone.
- In place, for editing an item within the context of a container.
- Open, for editing an item outside the context of its container, often in a separate window.

This requires three separate menu layouts, one for each possible mode of the application. Accelerator tables are also necessary for each new mode. A container application may or may not support in-place activation; if it does, it needs a new menu structure and associated accelerator tables.

In-place activation requires that the container and server applications must negotiate for menu, toolbar, and status bar space. All resources must be designed with this in mind. The article Menus and Resources: Menu Merging covers this topic in detail.

Because of these issues, OLE document applications created with the application wizard can have up to four separate menus and accelerator table resources. These are used for the following reasons:

Resource name	Use
IDR_MAINFRAME	Used in an MDI application if no file is open, or in an SDI application regardless of open files. This is the standard menu used in non-OLE applications.
IDR_<project>TYPE	Used in an MDI application if files are open. Used when an application is running stand-alone. This is the standard menu used in non-OLE applications.
IDR_<project>TYPE_SRVR_IP	Used by the server or container when an object is open in place.
IDR_<project>TYPE_SRVR_EMB	Used by a server application if an object is opened without using in-place activation.

Each of these resource names represents a menu and, usually, an accelerator table. A similar scheme should be used in MFC applications that are not created with the application wizard.

The following articles discuss topics related to containers, servers, and the menu merging necessary to implement in-place activation:

- Menus and Resources: Container Additions
- Menus and Resources: Server Additions
- Menus and Resources: Menu Merging

See Also

Concepts

OLE in MFC

Menus and Resources: Container Additions

This article explains the changes that need to be made to the menus and other resources in a visual editing container application.

In container applications, two types of changes need to be made: modifications to existing resources to support OLE visual editing and addition of new resources used for in-place activation. If you use the application wizard to create your container application, these steps will be done for you, but they may require some customization.

If you do not use the application wizard, you may want to look at OCLIENT.RC, the resource script for the OCLIENT sample application, to see how these changes are implemented. See the MFC OLE sample OCLIENT.

Topics covered in this article include:

- Container Menu Additions
- Accelerator Table Additions
- String Table Additions

Container Menu Additions

You must add the following items to the Edit menu:

Item	Purpose
Insert New Object	Opens the OLE Insert Object dialog box to insert a linked or embedded item into the document.
Paste Link	Pastes a link to the item on the Clipboard into the document.
OLE Verb	Calls the selected item's primary verb. The text of this menu item changes to reflect the primary verb of the selected item.
Links	Opens the OLE Edit Links dialog box to change existing linked items.

In addition to the changes listed in this article, your source file must include AFXOLECL.RC, which is required for the Microsoft Foundation Class Library implementation. Insert New Object is the only required menu addition. Other items can be added, but those listed here are the most common.

You must create a new menu for your container application if you want to support in-place activation of contained items. This menu consists of the same File menu and Window pop-up menus used when files are open, but it has two separators placed between them. These separators are used to indicate where the server (component) item (application) should place its menus when activated in place. For more information on this menu-merging technique, see Menus and Resources: Menu Merging.

Container Application Accelerator Table Additions

Small changes to a container application's accelerator table resources are necessary if you are supporting in-place activation. The first change allows the user to press the escape key (ESC) to cancel the in-place editing mode. Add the following entry to the main accelerator table:

ID	Key	Type
ID_CANCEL_EDIT_CNTR	VK_ESCAPE	**VIRTKEY**

The second change is to create a new accelerator table that corresponds to the new menu resource created for in-place activation. This table has entries for the File and Window menus in addition to the **VK_ESCAPE** entry above. The following example is the accelerator table created for in-place activation in the MFC sample CONTAINER:

ID	Key	Type
ID_FILE_NEW	CTRL+N	**VIRTKEY**

ID_FILE_OPEN	CTRL+O	**VIRTKEY**
ID_FILE_SAVE	CTRL+S	**VIRTKEY**
ID_FILE_PRINT	CTRL+P	**VIRTKEY**
ID_NEXT_PANE	VK_F6	**VIRTKEY**
ID_PREV_PANE	SHIFT+VK_F6	**VIRTKEY**
ID_CANCEL_EDIT_CNTR	VK_ESCAPE	**VIRTKEY**

String Table Additions for Container Applications

Most of the changes to string tables for container applications correspond to the additional menu items mentioned in Container Menu Additions. They supply the text displayed in the status bar when each menu item is displayed. As an example, here are the string-table entries the application wizard generates:

ID	String
IDP_OLE_INIT_FAILED	OLE initialization failed. Make sure that the OLE libraries are the correct version.
IDP_FAILED_TO_CREATE	Failed to create object. Make sure that the object is entered in the system registry.

See Also

Concepts

Menus and Resources (OLE)

Menus and Resources: Server Additions

Menus and Resources: Server Additions

This article explains the changes that need to be made to the menus and other resources in a visual editing server (component) application. A server application requires many additions to the menu structure and other resources because it can be started in one of three modes: stand alone, embedded, or in place. As described in the Menus and Resources (OLE) article, there are a maximum of four sets of menus. All four are used for an MDI full-server application, while only three are used for a miniserver. The application wizard will create the menu layout necessary for the type of server you want. Some customization may be necessary.

If you do not use the application wizard, you may want to look at HIERSVR.RC, the resource script for the MFC sample application HIERSVR, to see how these changes are implemented.

Topics covered in this article include:

- Server Menu Additions
- Accelerator Table Additions
- String Table Additions
- Miniserver Additions

Server Menu Additions

Server (component) applications must have menu resources added to support OLE visual editing. The menus used when the application is run in stand-alone mode do not have to be changed, but you must add two new menu resources before building the application: one to support in-place activation and one to support the server being fully open. Both menu resources are used by full- and miniserver applications.

- To support in-place activation, you must create a menu resource that is very similar to the menu resource used when run in stand-alone mode. The difference in this menu is that the File and Window items (and any other menu items that deal with the application, and not the data) are missing. The container application will supply these menu items. For more information on, and an example of, this menu-merging technique, see the article Menus and Resources: Menu Merging.
- To support fully open activation, you must create a menu resource nearly identical to the menu resource used when run in stand-alone mode. The only modification to this menu resource is that some items are reworded to reflect the fact that the server is operating on an item embedded in a compound document.

In addition to the changes listed in this article, your resource file needs to include AFXOLESV.RC, which is required for the Microsoft Foundation Class Library implementation. This file is in the MFC\Include subdirectory.

Server Application Accelerator Table Additions

Two new accelerator table resources must be added to server applications; they correspond directly to the new menu resources previously described. The first accelerator table is used when the server application is activated in place. It consists of all the entries in the view's accelerator table except those tied to the File and Window menus.

The second table is nearly an exact copy of the view's accelerator table. Any differences parallel changes made in the fully open menu mentioned in Server Menu Additions.

For an example of these accelerator table changes, compare the **IDR_HIERSVRTYPE_SRVR_IP** and **IDR_HIERSVRTYPE_SRVR_EMB** accelerator tables with **IDR_MAINFRAME** in the HIERSVR.RC file included in the MFC OLE sample HIERSVR. The File and Window accelerators are missing from the in-place table and exact copies of them are in the embedded table.

String Table Additions for Server Applications

Only one string table addition is necessary in a server application — a string to signify that the OLE initialization failed. As an example, here is the string-table entry that the application wizard generates:

ID	String
IDP_OLE_INIT_FAILED	OLE initialization failed. Make sure that the OLE libraries are the correct version.

Miniserver Additions

The same additions apply for miniservers as those listed above for full-servers. Because a miniserver cannot be run in stand-alone mode, its main menu is much smaller. The main menu created by the application wizard has only a File menu, containing only the items Exit and About. Embedded and in-place menus and accelerators for miniservers are the same as those for full-servers.

See Also

Concepts

Menus and Resources (OLE)
Menus and Resources: Menu Merging

Menus and Resources: Menu Merging

This article details the steps necessary for OLE document applications to handle visual editing and in-place activation properly. In-place activation poses a challenge for both container and server (component) applications. The user remains in the same frame window (within the context of the container document) but is actually running another application (the server). This requires coordination between the resources of the container and server applications.

Topics covered in this article include:

- Menu Layouts
- Toolbars and Status Bars

Menu Layouts

The first step is to coordinate menu layouts. For more information, see the **Menu Creation** section in Menu Programming Considerations in the *Platform SDK*.

Container applications should create a new menu to be used only when embedded items are activated in place. At the minimum, this menu should consist of the following, in the order listed:

1. File menu identical to the one used when files are open. (Usually no other menu items are placed before the next item.)
2. Two consecutive separators.
3. Window menu identical to the one used when files are open (only if the container application in an MDI application). Some applications may have other menus, such as an Options menu, that belong in this group, which remains on the menu when an embedded item is activated in place.

Note
There may be other menus that affect the view of the container document, such as Zoom. These container menus appe ar between the two separators in this menu resource.

Server (component) applications should also create a new menu specifically for in-place activation. It should be like the menu used when files are open, but without menu items, such as File and Window that manipulate the server document instead of the data. Typically, this menu consists of the following:

1. Edit menu identical to the one used when files are open.
2. Separator.
3. Object editing menus, such as the Pen menu in the Scribble sample application.
4. Separator.
5. Help menu.

For an example, look at the layout of some sample in-place menus for a container and a server. The details of each menu item have been removed to make the example clearer. The container's in-place menu has the following entries:

```
IDR_CONTAINERTYPE_CNTR_IP MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File C1"
    MENUITEM SEPARATOR
    POPUP "&Zoom C2"
    MENUITEM SEPARATOR
    POPUP "&Options C3"
    POPUP "&Window C3"
END
```

The consecutive separators indicate where the first part of the server's menu should go. Now look at the server's in-place menu:

```
IDR_SERVERTYPE_SRVR_IP MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&Edit S1"
    MENUITEM SEPARATOR
    POPUP "&Format S2"
    MENUITEM SEPARATOR
    POPUP "&Help S3"
END
```

The separators here indicate where the second group of container menu items should go. The resulting menu structure when an object from this server is activated in place inside this container looks like this:

```
BEGIN
    POPUP "&File C1"
    POPUP "&Edit S1"
    POPUP "&Zoom C2"
    POPUP "&Format S2"
    POPUP "&Options C3
    POPUP "&Window C3"
    POPUP "&Help S3"
END
```

As you can see, the separators have been replaced with the different groups of each application's menu.

Accelerator tables associated with the in-place menu should also be supplied by the server application. The container will incorporate them into its own accelerator tables.

When an embedded item is activated in place, the framework loads the in-place menu. It then asks the server application for its menu for in-place activation and inserts it where the separators are. This is how the menus combine. You get menus from the container for operating on the file and window placement, and you get menus from the server for operating on the item.

Toolbars and Status Bars

Server applications should create a new toolbar and store its bitmap in a separate file. The application wizard–generated applications store this bitmap in a file called ITOOLBAR.BMP. The new toolbar replaces the container application's toolbar when your server's item is activated in place, and should contain the same items as your normal toolbar, but remove icons representing items on the File and Window menus.

This toolbar is loaded in your **COleIPFrameWnd**-derived class, created for you by the application wizard. The status bar is handled by the container application. For more information on the implementation of in-place frame windows, see Servers: Implementing a Server.

See Also

Concepts

Menus and Resources (OLE)
Activation (C++)
Servers
Containers

Registration

When a user wants to insert an OLE item into an application, OLE presents a list of object types to choose from. OLE gets this list from the system registration database, which contains information provided by all server applications. When a server registers itself, the entries it puts into the system registration database (the Registry) describe each type of object it supplies, file extensions, and the path to itself, among other information.

The framework and the OLE system dynamic-link libraries (DLL) use this registry to determine what types of OLE items are available on the system. The OLE system DLLs also use this registry to determine how to launch a server application when a linked or embedded object is activated.

This article describes what each server application needs to do when it is installed and each time it is executed.

For detailed information about the system registration database and the format of the .reg files used to update it, see the *OLE Programmer's Reference*.

Server Installation

When you first install your server application, it should register all the types of OLE items that it supports. You can also have the server update the system registration database every time it executes as a stand-alone application. This keeps the registration database up-to-date if the server's executable file is moved.

Note
MFC applications generated by the application wizard automatically register themselves when they are run as stand-alone applications.

If you want to register your application during installation, use the RegEdit.exe program. (In Windows 95, Windows 98, and Windows ME, RegEdit is in the Windows directory. In Windows NT and Windows 2000, RegEdit is in the Windows System32 directory.) If you include a setup program with your application, have the setup program run "RegEdit /S *appname*.reg". (The /S flag indicates silent operation, that is, it does not display the dialog box reporting successful completion of the command.) Otherwise, instruct the user to run RegEdit manually.

Note
The .reg file created by the application wizard does not include the complete path for the executable. Your installation program must either modify the .reg file to include the complete path to the executable or modify the PATH environment variable to include the installation directory.

RegEdit merges the contents of the .reg text file into the registration database. To verify the database or to repair it, use the registry editor. Take care to avoid deleting essential OLE entries. (In Windows 95, Windows 98, and Windows ME, the registry editor is RegEdit.exe. In Windows NT and Windows 2000, it is RegEdit32.exe.)

Server Initialization

When you create a server application with the application wizard, the wizard completes all initialization tasks for you automatically. This section describes what you must do if you write a server application manually.

When a server application is launched by a container application, the OLE system DLLs add the "/Embedding" option to the server's command line. A server application's behavior differs depending on whether it was launched by a container, so the first thing an application should do when it begins execution is check for the "/Embedding" or "-Embedding" option on the command line. If this switch exists, load a different set of resources that show the server as being either in-place active or fully open. For more information, see Menus and Resources: Server Additions.

Your server application should also call its **CWinApp::RunEmbedded** function to parse the command line. If it returns a nonzero value, the application should not show its window because it has been run from a container application, not as a stand-alone application. This function updates the server's entry in the system registration database and calls the **RegisterAll** member function for you, performing instance registration.

When your server application is starting, you must ensure that it can perform instance registration. Instance registration informs the OLE system DLLs that the server is active and ready to receive requests from containers. It does not add an entry to the registration database. Perform instance registration of the server by calling the **ConnectTemplate** member function defined by **COleTemplateServer**. This connects the **CDocTemplate** object to the **COleTemplateServer** object.

The **ConnectTemplate** function takes three parameters: the server's **CLSID**, a pointer to the **CDocTemplate** object, and a flag indicating whether the server supports multiple instances. A miniserver must be able to support multiple instances, that is, it must be possible for multiple instances of the server to run simultaneously, one for each container. Consequently, pass **TRUE** for this flag when launching a miniserver.

If you are writing a miniserver, by definition it will always be launched by a container. You should still parse the command line to check for the "/Embedding" option. The absence of this option on the command line means that the user has tried to launch the miniserver as a stand-alone application. If this occurs, register the server with the system registration database and then display a message box informing the user to launch the miniserver from a container application.

See Also

Reference

CWinApp::RunAutomated
CWinApp::RunEmbedded
COleTemplateServer Class

Concepts

OLE in MFC
Servers

Servers

A server application (or component application) creates OLE items (or components) for use by container applications. A visual editing server application also supports visual editing or in-place activation. Another form of OLE server is an automation server. Some server applications support only the creation of embedded items; others support the creation of both embedded and linked items. Some support linking only, although this is rare. All server applications must support activation by container applications when the user wants to edit an item. An application can be both a container and a server. In other words, it can both incorporate data into its documents, and create data that can be incorporated as items into other applications' documents.

A miniserver is a special type of server application that can only be launched by a container. Microsoft Draw and Microsoft Graph are examples of miniservers. A miniserver does not store documents as files on disk. Instead, it reads its documents from and writes them to items in documents belonging to containers. As a result, a miniserver supports embedding only, not linking.

A full server can be run either as a stand-alone application or launched by a container application. A full server can store documents as files on disk. It can support embedding only, both embedding and linking, or linking only. The user of a container application can create an embedded item by choosing the Cut or Copy command in the server and the Paste command in the container. A linked item is created by choosing the Copy command in the server and the Paste Link command in the container. Alternatively, the user can create an embedded or linked item using the Insert Object dialog box.

The following table summarizes characteristics of different types of servers:

Server Characteristics

Type of server	Supports multiple instances	Items per document	Documents per instance
Miniserver	Yes	1	1
SDI full server	Yes	1 (if linking is supported, 1 or more)	1
MDI full server	No (not required)	1 (if linking is supported, 1 or more)	0 or more

A server application should support multiple containers simultaneously, in the event that more than one container will be used to edit an embedded or linked item. If the server is an SDI application (or a miniserver with a dialog box interface), multiple instances of the server must be able to run simultaneously. This allows a separate instance of the application to handle each container request.

If the server is an MDI application, it can create a new MDI child window each time a container needs to edit an item. In this way, a single instance of the application can support multiple containers.

Your server application must tell the OLE system DLLs what to do if one instance of the server is already running when another container requests its services: whether it should launch a new instance of the server or direct all containers' requests to one instance of the server.

For more details on servers, see:

- Servers: Implementing a Server
- Servers: Implementing Server Documents
- Servers: Implementing In-Place Frame Windows
- Servers: Server Items
- Servers: User-Interface Issues

See Also

Concepts

OLE in MFC
Containers
Containers: Advanced Features
Menus and Resources (OLE)
Registration
Automation Servers

Servers: Implementing a Server

This article explains the code the MFC Application Wizard creates for a visual editing server application. If you are not using the application wizard, this article lists the areas where you must write code to implement a server application.

If you are using the application wizard to create a new server application, it provides a significant amount of server-specific code for you. If you are adding visual editing server functionality to an existing application, you must duplicate the code that the application wizard would have provided before adding the rest of the necessary server code.

The server code that the application wizard provides falls into several categories:

- Defining server resources:
  - The menu resource used when the server is editing an embedded item in its own window.
  - The menu and toolbar resources used when the server is active in place.

  For more information on these resources, see Menus and Resources: Server Additions.
- Defining an item class derived from **COleServerItem**. For further details on server items, see Servers: Server Items.
- Changing the base class of the document class to **COleServerDoc**. For further details, see Servers: Implementing Server Documents.
- Defining a frame-window class derived from **COleIPFrameWnd**. For further details, see Servers: Implementing In-Place Frame Windows.
- Creating an entry for the server application in the Windows registration database and registering the new instance of the server with the OLE system. For information on this topic, see Registration.
- Initializing and launching the server application. For information on this topic, see Registration.

For more information, see COleServerItem, COleServerDoc, and COleIPFrameWnd in the *Class Library Reference*.

See Also

Concepts

Servers
Containers
Menus and Resources (OLE)
Registration

Servers: Implementing Server Documents

This article explains the steps you must take to successfully implement a server document if you did not specify the OLE Server option in the application wizard.

To define a server document class

1. Derive your document class from **COleServerDoc** instead of **CDocument**.
2. Create a server item class derived from **COleServerItem**.
3. Implement the **OnGetEmbeddedItem** member function of your server document class.

   OnGetEmbeddedItem is called when the user of a container application creates or edits an embedded item. It should return an item representing the entire document. This should be an object of your **COleServerItem**-derived class.
4. Override the **Serialize** member function to serialize the contents of the document. You do not need to serialize the list of server items unless you are using them to represent the native data in your document. For more information, see *Implementing Server Items* in the article Servers: Server Items.

When a server document is created, the framework automatically registers the document with the OLE system DLLs. This allows the DLLs to identify the server documents.

For more information, see COleServerItem and COleServerDoc in the *Class Library Reference*.

See Also

Concepts

Servers
Servers: Server Items
Servers: Implementing a Server
Servers: Implementing In-Place Frame Windows

Servers: Implementing In-Place Frame Windows

This article explains what you must do to implement in-place frame windows in your visual editing server application if you do not use the application wizard to create your server application. In place of following the procedure outlined in this article, you could use an existing in-place frame-window class from either an application wizard–generated application or a sample provided with Visual C++.

To declare an in-place frame-window class

1. Derive an in-place frame-window class from **COleIPFrameWnd**.
   - Use the **DECLARE_DYNCREATE** macro in your class header file.
   - Use the **IMPLEMENT_DYNCREATE** macro in your class implementation (.cpp) file. This allows objects of this class to be created by the framework.
2. Declare a **COleResizeBar** member in the frame-window class. This is needed if you want to support in-place resizing in server applications.

   Declare an **OnCreate** message handler (using the **Properties** window), and call **Create** for your **COleResizeBar** member, if you've defined it.
3. If you have a toolbar, declare a **CToolBar** member in the frame-window class.

   Override the **OnCreateControlBars** member function to create a toolbar when the server is active in place. For example:

   ```
   BOOL CInPlaceFrame::OnCreateControlBars
      (CWnd* pWndFrame, CWnd* pWndDoc)
   {
      // Create toolbar on client's frame window
      if (!m_wndToolBar.Create(pWndFrame) ||
          !m_wndToolBar.LoadToolBar(IDR_PROJ_SRVR_IP))
      {
         TRACE("Failed to create toolbar\n");
         return FALSE;
      }

      // Set this window as owner, so messages are
      // delivered to proper app
      m_wndToolBar.SetOwner(this);

      // Enable docking for the toolbar
      m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
      pWndFrame->EnableDocking(CBRS_ALIGN_ANY);
      pWndFrame->DockControlBar(&m_wndToolBar);

      // Enable tooltips for the toolbar
      m_wndToolBar.SetBarStyle(CBRS_TOOLTIPS |
         CBRS_FLYBY | m_wndToolBar.GetBarStyle());

       return TRUE;
   }
   ```

   See the discussion of this code following step 5.
4. Include the header file for this in-place frame-window class in your main .cpp file.
5. In **InitInstance** for your application class, call the **SetServerInfo** function of the document template object to specify the resources and in-place frame window to be used in open and in-place editing.

The series of function calls in the **if** statement creates the toolbar from the resources the server provided. At this point, the

toolbar is part of the container's window hierarchy. Because this toolbar is derived from **CToolBar**, it will pass its messages to its owner, the container application's frame window, unless you change the owner. That is why the call to **SetOwner** is necessary. This call changes the window where commands are sent to be the server's in-place frame window, causing the messages to be passed to the server. This allows the server to react to operations on the toolbar that it provides.

The ID for the toolbar bitmap should be the same as the other in-place resources defined in your server application. See Menus and Resources: Server Additions for details.

For more information, see COleIPFrameWnd, COleResizeBar, and CDocTemplate::SetServerInfo in the *Class Library Reference*.

See Also

Concepts

Servers
Servers: Implementing a Server
Servers: Implementing Server Documents
Servers: Server Items

Servers: Server Items

When a container launches a server so that a user can edit an embedded or linked OLE item, the server application creates a "server item." The server item, which is an object of a class derived from **COleServerItem**, provides an interface between the server document and the container application.

The **COleServerItem** class defines several overridable member functions that are called by OLE, usually in response to requests from the container. Server items can represent part of the server document or the entire document. When an OLE item is embedded in the container document, the server item represents the entire server document. When the OLE item is linked, the server item can represent a part of the server document or the whole document, depending on whether the link is to a part or to the whole.

In the HIERSVR sample, for example, the server-item class, **CServerItem**, has a member that is a pointer to an object of the class **CServerNode**. The **CServerNode** object is a node in the HIERSVR application's document, which is a tree. When the **CServerNode** object is the root node, the **CServerItem** object represents the whole document. When the **CServerNode** object is a child node, the **CServerItem** object represents a part of the document. See the MFC OLE sample HIERSVR for an example of this interaction.

Implementing Server Items

If you use the application wizard to produce "starter" code for your application, all you have to do to include server items in your starter code is to choose one of the server options from the OLE Options page. If you are adding server items to an existing application, perform the following steps:

To implement a server item

1. Derive a class from **COleServerItem**.
2. In your derived class, override the **OnDraw** member function.

   The framework calls **OnDraw** to render the OLE item into a metafile. The container application uses this metafile to render the item. Your application's view class also has an **OnDraw** member function, which is used to render the item when the server application is active.
3. Implement an override of **OnGetEmbeddedItem** for your server-document class. For further information, see the article Servers: Implementing Server Documents and the MFC OLE sample HIERSVR.
4. Implement your server-item class's **OnGetExtent** member function. The framework calls this function to retrieve the size of the item. The default implementation does nothing.

A Tip for Server-Item Architecture

As noted in Implementing Server Items, server applications must be able to render items both in the server's view and in a metafile used by the container application. In the Microsoft Foundation Class Library's application architecture, the view class's **OnDraw** member function renders the item when it is being edited (see CView::OnDraw in the *Class Library Reference*). The server item's **OnDraw** renders the item into a metafile in all other cases (see COleServerItem::OnDraw).

You can avoid duplication of code by writing helper functions in your server-document class and calling them from the **OnDraw** functions in your view and server-item classes. The MFC OLE sample HIERSVR uses this strategy: the functions **CServerView::OnDraw** and **CServerItem::OnDraw** both call **CServerDoc::DrawTree** to render the item.

The view and the item both have **OnDraw** member functions because they draw under different conditions. The view must take into account such factors as zooming, selection size and extent, clipping, and user-interface elements such as scroll bars. The server item, on the other hand, always draws the entire OLE object.

For more information, see CView::OnDraw, COleServerItem, COleServerItem::OnDraw, and COleServerDoc::OnGetEmbeddedItem in the *Class Library Reference*.

See Also

Concepts

Servers

Servers: User-Interface Issues

A server application has a number of features that must be added to the user interface to supply OLE items to container applications. For further information on the menus and additional resources that need to be added to a server application, see Menus and Resources: Server Additions.

See Also

Concepts

Servers

Menus and Resources (OLE)

Trackers

The CRectTracker class provides a user interface between rectangular items in your application and your user by providing a variety of display styles. These styles include solid, hatched, or dashed borders; a hatched pattern that covers the item; and resize handles that can be located on the outside or inside of a border. Trackers are often used in conjunction with OLE items, that is, objects derived from **COleClientItem**. The tracker rectangles give visual cues on the current status of the item.

The MFC OLE sample OCLIENT demonstrates a common interface using trackers and OLE client items from the viewpoint of a container application. For a demonstration of the different styles and abilities of a tracker object, see the MFC general sample TRACKER.

For more information on implementing trackers in your OLE application, see
Trackers: Implementing Trackers in Your OLE Application

See Also

Reference
COleClientItem Class
Concepts
OLE in MFC

Trackers: Implementing Trackers in Your OLE Application

Trackers provide a graphical interface to enable users to interact with OLE client items. By using different tracker styles, OLE client items can be displayed with hatched borders, resize handles, or a variety of other visual effects. This article describes:

- How to Implement Tracking in Your Code.
- Rubber-Banding and Trackers.

The article also covers the use of styles with trackers. In addition, it makes several references to the MFC OLE sample OCLIENT.

See Also

Concepts

Trackers

How to: Implement Tracking in Your Code

To track an OLE item, you must handle certain events related to the item, such as clicking the item or updating the view of the document. In all cases, it is sufficient to declare a temporary CRectTracker object and manipulate the item by means of this object.

When a user selects an item or inserts an object with a menu command, you must initialize the tracker with the proper styles to represent the state of the OLE item. The following table outlines the conventions used by the OCLIENT sample. For more information on these styles, see **CRectTracker**.

Container Styles and States of the OLE Item

Style displayed	State of OLE item
Dotted border	Item is linked
Solid border	Item is embedded in your document
Resize handles	Item is currently selected
Hatched border	Item is currently in-place active
Hatching pattern overlays item	Item's server is open

You can handle this initialization easily using a procedure that checks the state of the OLE item and sets the appropriate styles. The **SetupTracker** function found in the OCLIENT sample demonstrates tracker initialization. The parameters for this function are the address of the tracker, *pTracker*; a pointer to the client item that is related to the tracker, *pItem*; and a pointer to a rectangle, *pTrueRect*. For a more complete example of this function, see the MFC OLE sample OCLIENT.

The **SetupTracker** code example presents a single function; lines of the function are interspersed with discussion of the function's features:

```
void CMainView::SetupTracker( CRectTracker* pTracker,
   CRectItem* pItem, CRect* pTrueRect )
{
```

The tracker is initialized by setting the minimum size and clearing the style of the tracker.

```
   pTracker->m_sizeMin.cx = 8;
   pTracker->m_sizeMin.cy = 8;

   pTracker->m_nStyle = 0;
```

The following lines check to see whether the item is currently selected and whether the item is linked to the document or embedded in it. Resize handles located on the inside of the border are added to the style, indicating that the item is currently selected. If the item is linked to your document, the dotted border style is used. A solid border is used if the item is embedded.

```
   if ( pItem == m_pSelection )
      pTracker->m_nStyle |= CRectTracker::resizeInside;

   if ( pItem->GetType( ) == OT_LINK )
      pTracker->m_nStyle |= CRectTracker::dottedLine;
   else
      pTracker->m_nStyle |= CRectTracker::solidLine;
```

The following code overlays the item with a hatched pattern if the item is currently open.

```
   if ( pItem->GetItemState( ) ==
           COleClientItem::openState ||
```

```
        pItem->GetItemState( ) ==
           COleClientItem::activeUIState )
      pTracker->m_nStyle |= CRectTracker::hatchInside;
}
```

You can then call this function whenever the tracker has to be displayed. For example, call this function from the **OnDraw** function of your view class. This updates the tracker's appearance whenever the view is repainted. For a complete example, see the **CMainView::OnDraw** function of the MFC OLE sample OCLIENT.

In your application, events that require tracker code, such as resizing, moving, or hit detecting, will occur. These actions usually indicate that an attempt is being made to grab or move the item. In these cases, you will need to decide what was grabbed: a resize handle or a portion of the border between resize handles. The **OnLButtonDown** message handler is a good place to test the position of the mouse in relation to the item. Make a call to **CRectTracker::HitTest**. If the test returns something besides **CRectTracker::hitOutside**, the item is being resized or moved. Therefore, you should make a call to the **Track** member function. See the **CMainView::OnLButtonDown** function located in the MFC OLE sample OCLIENT for a complete example.

The **CRectTracker** class provides several different cursor shapes used to indicate whether a move, resize, or drag operation is taking place. To handle this event, check to see whether the item currently under the mouse is selected. If it is, make a call to **CRectTracker::SetCursor**, or call the default handler. The following example is from the MFC OLE sample OCLIENT:

```
BOOL CMainView::OnSetCursor( CWnd* pWnd, UINT nHitTest,
  UINT message )
{
  if ( pWnd == this && m_pSelection != NULL )
  {
    // give the tracker for the selection a chance
    CRectTracker tracker;
    SetupTracker( &tracker, m_pSelection );
    if ( tracker.SetCursor( this, nHitTest ))
      return TRUE;
  }
  return CScrollView::OnSetCursor( pWnd,
    nHitTest, message );
}
```

See Also

Concepts

Trackers: Implementing Trackers in Your OLE Application

Rubber-Banding and Trackers

Another feature supplied with trackers is the "rubber-band" selection, which allows a user to select multiple OLE items by dragging a sizing rectangle around the items to be selected. When the user releases the left mouse button, items within the region selected by the user are selected and can be manipulated by the user. For instance, the user might drag the selection into another container application.

Implementing this feature requires some additional code in your application's **WM_LBUTTONDOWN** handler function.

The following code sample implements rubber-band selection and additional features and is taken from the **WM_LBUTTONDOWN** handler function of MFC General sample TRACKER.

```
if (pDoc->m_tracker.HitTest(point) < 0)
{
  // just to demonstrate CRectTracker::TrackRubberBand
  CRectTracker tracker;
  if (tracker.TrackRubberBand(this, point,
    pDoc->m_bAllowInvert))
  {
    MessageBeep(0); // beep indicates TRUE

    // See if rubber band intersects
    // with the doc's tracker
    CRect rectT;
    // so intersect rect works
    tracker.m_rect.NormalizeRect();
    if (rectT.IntersectRect(tracker.m_rect,
      pDoc->m_tracker.m_rect))
    {
    // If so, put resize handles on it
    // (i.e. select it)
      if (pDoc->m_tracker.m_nStyle &
        CRectTracker::resizeInside)
      {
      // swap from resize inside to
      // resize outside for effect
        pDoc->m_tracker.m_nStyle &=
          ~CRectTracker::resizeInside;
        pDoc->m_tracker.m_nStyle |=
          CRectTracker::resizeOutside;
      }
      else
      {
      // Just use inside resize handles on first time
        pDoc->m_tracker.m_nStyle &=
          ~CRectTracker::resizeOutside;
        pDoc->m_tracker.m_nStyle |=
          CRectTracker::resizeInside;
      }
      pDoc->SetModifiedFlag();
      pDoc->UpdateAllViews(NULL,
        (LPARAM)(LPCRECT)rectSave);
      pDoc->UpdateAllViews(NULL);
    }
  }
}
```

If you want to allow reversible orientation of the tracker during rubber-banding, you should call CRectTracker::TrackRubberBand with the third parameter set to **TRUE**. Remember that allowing reversible orientation will sometimes cause CRectTracker::m_rect to become inverted. This can be corrected by a call to CRect::NormalizeRect.

For more information, see Container Client Items and Customizing Drag and Drop.

See Also

Reference

CRectTracker Class

Concepts

Trackers: Implementing Trackers in Your OLE Application

Serialization in MFC

This article explains the serialization mechanism provided in the Microsoft Foundation Class Library (MFC) to allow objects to persist between runs of your program.

Serialization is the process of writing or reading an object to or from a persistent storage medium such as a disk file. Serialization is ideal for situations where it is desired to maintain the state of structured data (such as C++ classes or structures) during or after execution of a program. Using the serialization objects provided by MFC allows this to occur in a standard and consistent manner, relieving the user from the need to perform file operations by hand.

MFC supplies built-in support for serialization in the class **CObject**. Thus, all classes derived from **CObject** can take advantage of **CObject**'s serialization protocol.

The basic idea of serialization is that an object should be able to write its current state, usually indicated by the value of its member variables, to persistent storage. Later, the object can be re-created by reading, or deserializing, the object's state from the storage. Serialization handles all the details of object pointers and circular references to objects that are used when you serialize an object. A key point is that the object itself is responsible for reading and writing its own state. Thus, for a class to be serializable, it must implement the basic serialization operations. As shown in the Serialization group of articles, it is easy to add this functionality to a class.

MFC uses an object of the **CArchive** class as an intermediary between the object to be serialized and the storage medium. This object is always associated with a **CFile** object, from which it obtains the necessary information for serialization, including the file name and whether the requested operation is a read or write. The object that performs a serialization operation can use the **CArchive** object without regard to the nature of the storage medium.

A **CArchive** object uses overloaded insertion (**<<**) and extraction (**>>**) operators to perform writing and reading operations. For more information, see Storing and Loading CObjects via an Archive in the article Serialization: Serializing an Object.

Note
Do not confuse the **CArchive** class with general-purpose iostream classes, which are for formatted text only. The **CArchive** c lass is for binary-format serialized objects.

If you want, you can bypass MFC serialization to create your own mechanism for persistent data storage. You will need to override the class member functions that initiate serialization at the user's command. See the discussion in Technical Note 22 of the **ID_FILE_OPEN**, **ID_FILE_SAVE**, and **ID_FILE_SAVE_AS** standard commands.

The following articles cover the two main tasks required for serialization:

- Serialization: Making a Serializable Class
- Serialization: Serializing an Object

The article Serialization: Serialization vs. Database Input/Output describes when serialization is an appropriate input/output technique in database applications.

See Also

Reference

CArchive Class
CObject Class
CDocument Class
CFile Class

Concepts

General MFC Topics

Other Resources

MFC Concepts

Serialization: Making a Serializable Class

Five main steps are required to make a class serializable. They are listed below and explained in the following sections:

1. Deriving your class from CObject (or from some class derived from **CObject**).
2. Overriding the Serialize member function.
3. Using the DECLARE_SERIAL macro in the class declaration.
4. Defining a constructor that takes no arguments.
5. Using the IMPLEMENT_SERIAL macro in the implementation file for your class.

If you call **Serialize** directly rather than through the >> and << operators of CArchive, the last three steps are not required for serialization.

Deriving Your Class from CObject

The basic serialization protocol and functionality are defined in the **CObject** class. By deriving your class from **CObject** (or from a class derived from **CObject**), as shown in the following declaration of class `CPerson`, you gain access to the serialization protocol and functionality of **CObject**.

Overriding the Serialize Member Function

The **Serialize** member function, which is defined in the **CObject** class, is responsible for actually serializing the data necessary to capture an object's current state. The **Serialize** function has a **CArchive** argument that it uses to read and write the object data. The CArchive object has a member function, **IsStoring**, which indicates whether **Serialize** is storing (writing data) or loading (reading data). Using the results of **IsStoring** as a guide, you either insert your object's data in the **CArchive** object with the insertion operator (**<<**) or extract data with the extraction operator (**>>**).

Consider a class that is derived from **CObject** and has two new member variables, of types **CString** and **WORD**. The following class declaration fragment shows the new member variables and the declaration for the overridden **Serialize** member function:

```
class CPerson : public CObject
{
public:
    DECLARE_SERIAL( CPerson )
    // empty constructor is necessary
    CPerson(){};

    CString m_name;
    WORD   m_number;

    void Serialize( CArchive& archive );

    // rest of class declaration
};
```

To override the Serialize member function

1. Call your base class version of **Serialize** to make sure that the inherited portion of the object is serialized.
2. Insert or extract the member variables specific to your class.

   The insertion and extraction operators interact with the archive class to read and write the data. The following example shows how to implement **Serialize** for the `CPerson` class declared above:

   ```
   void CPerson::Serialize( CArchive& archive )
   {
       // call base class function first
       // base class is CObject in this case
       CObject::Serialize( archive );
   ```

```
    // now do the stuff for our specific class
    if( archive.IsStoring() )
        archive << m_name << m_number;
    else
        archive >> m_name >> m_number;
}
```

You can also use the CArchive::Read and CArchive::Write member functions to read and write large amounts of untyped data.

Using the DECLARE_SERIAL Macro

The **DECLARE_SERIAL** macro is required in the declaration of classes that will support serialization, as shown here:

```
class CPerson : public CObject
{
    DECLARE_SERIAL( CPerson )
    // rest of declaration follows...
};
```

Defining a Constructor with No Arguments

MFC requires a default constructor when it re-creates your objects as they are deserialized (loaded from disk). The deserialization process will fill in all member variables with the values required to re-create the object.

This constructor can be declared public, protected, or private. If you make it protected or private, you help make sure that it will only be used by the serialization functions. The constructor must put the object in a state that allows it to be deleted if necessary.

Note
If you forget to define a constructor with no arguments in a class that uses the **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** macros, you will get a "no default constructor available" compiler warning on the line where the **IMPLEMENT_SERIAL** macro is used.

Using the IMPLEMENT_SERIAL Macro in the Implementation File

The **IMPLEMENT_SERIAL** macro is used to define the various functions needed when you derive a serializable class from **CObject**. You use this macro in the implementation file (.CPP) for your class. The first two arguments to the macro are the name of the class and the name of its immediate base class.

The third argument to this macro is a schema number. The schema number is essentially a version number for objects of the class. Use an integer greater than or equal to 0 for the schema number. (Don't confuse this schema number with database terminology.)

The MFC serialization code checks the schema number when reading objects into memory. If the schema number of the object on disk does not match the schema number of the class in memory, the library will throw a **CArchiveException**, preventing your program from reading an incorrect version of the object.

If you want your `Serialize` member function to be able to read multiple versions — that is, files written with different versions of the application — you can use the value **VERSIONABLE_SCHEMA** as an argument to the **IMPLEMENT_SERIAL** macro. For usage information and an example, see the **GetObjectSchema** member function of class **CArchive**.

The following example shows how to use **IMPLEMENT_SERIAL** for a class, `CPerson`, that is derived from **CObject**:

```
IMPLEMENT_SERIAL( CPerson, CObject, 1 )
```

Once you have a serializable class, you can serialize objects of the class, as discussed in the article Serialization: Serializing an Object.

See Also

Concepts

Serialization in MFC

Serialization: Serializing an Object

The article Serialization: Making a Serializable Class shows how to make a class serializable. Once you have a serializable class, you can serialize objects of that class to and from a file via a CArchive object. This article explains:

- What a CArchive object is.
- Two ways to create a CArchive.
- How to use the CArchive << and >> operators.
- Storing and loading CObjects via an archive.

You can let the framework create the archive for your serializable document or explicitly create the **CArchive** object yourself. You can transfer data between a file and your serializable object by using the << and >> operators for **CArchive** or, in some cases, by calling the `Serialize` function of a **CObject**-derived class.

See Also

Concepts

Serialization in MFC

What Is a CArchive Object

A **CArchive** object provides a type-safe buffering mechanism for writing or reading serializable objects to or from a **CFile** object. Usually the **CFile** object represents a disk file; however, it can also be a memory file (**CSharedFile** object), perhaps representing the Clipboard.

A given **CArchive** object either stores (writes, serializes) data or loads (reads, deserializes) data, but never both. The life of a **CArchive** object is limited to one pass through writing objects to a file or reading objects from a file. Thus, two successively created **CArchive** objects are required to serialize data to a file and then deserialize it back from the file.

When an archive stores objects to a file, the archive attaches the **CRuntimeClass** name to the objects. Then, when another archive loads objects from a file to memory, the **CObject**-derived objects are dynamically reconstructed based on the **CRuntimeClass** of the objects. A given object may be referenced more than once as it is written to the file by the storing archive. The loading archive, however, will reconstruct the object only once. The details about how an archive attaches **CRuntimeClass** information to objects and reconstructs objects, taking into account possible multiple references, are described in Technical Note 2.

As data is serialized to an archive, the archive accumulates the data until its buffer is full. Then the archive writes its buffer to the **CFile** object pointed to by the **CArchive** object. Similarly, as you read data from an archive, it reads data from the file to its buffer and then from the buffer to your deserialized object. This buffering reduces the number of times a hard disk is physically read, thus improving your application's performance.

See Also

Concepts

Serialization: Serializing an Object

Two Ways to Create a CArchive Object

There are two ways to create a **CArchive** object:

- Implicit creation of a CArchive object via the framework
- Explicit creation of a CArchive object

Implicit Creation of a CArchive Object via the Framework

The most common, and easiest, way is to let the framework create a **CArchive** object for your document on behalf of the Save, Save As, and Open commands on the File menu.

Here is what the framework does when the user of your application issues the Save As command from the File menu:

1. Presents the **Save As** dialog box and gets the filename from the user.
2. Opens the file named by the user as a **CFile** object.
3. Creates a **CArchive** object that points to this **CFile** object. In creating the **CArchive** object, the framework sets the mode to "store" (write, serialize), as opposed to "load" (read, deserialize).
4. Calls the `Serialize` function defined in your **CDocument**-derived class, passing it a reference to the **CArchive** object.

Your document's `Serialize` function then writes data to the **CArchive** object, as explained shortly. Upon return from your `Serialize` function, the framework destroys the **CArchive** object and then the **CFile** object.

Thus, if you let the framework create the **CArchive** object for your document, all you have to do is implement the document's `Serialize` function that writes and reads to and from the archive. You also have to implement `Serialize` for any **CObject**-derived objects that the document's `Serialize` function in turn serializes directly or indirectly.

Explicit Creation of a CArchive Object

Besides serializing a document via the framework, there are other occasions when you may need a **CArchive** object. For example, you might want to serialize data to and from the Clipboard, represented by a **CSharedFile** object. Or, you may want to use a user interface for saving a file that is different from the one offered by the framework. In this case, you can explicitly create a **CArchive** object. You do this the same way the framework does, using the following procedure.

To explicitly create a CArchive object

1. Construct a **CFile** object or an object derived from **CFile**.
2. Pass the **CFile** object to the constructor for **CArchive**, as shown in the following example:

   ```
   CFile theFile;
   theFile.Open(..., CFile::modeWrite);
   CArchive archive(&theFile, CArchive::store);
   ```

   The second argument to the **CArchive** constructor is an enumerated value that specifies whether the archive will be used for storing or loading data to or from the file. The `Serialize` function of an object checks this state by calling the **IsStoring** function for the archive object.

When you are finished storing or loading data to or from the **CArchive** object, close it. Although the **CArchive** (and **CFile**) objects will automatically close the archive (and file), it is good practice to explicitly do so since it makes recovery from errors easier. For more information about error handling, see the article Exceptions: Catching and Deleting Exceptions.

To close the CArchive object

- The following example illustrates how to close the **CArchive** object:

  ```
  archive.Close();
  theFile.Close();
  ```

See Also

Concepts

Serialization: Serializing an Object

Using the CArchive << and >> Operators

CArchive provides << and >> operators for writing and reading simple data types as well as **CObject**s to and from a file.

To store an object in a file via an archive

- The following example shows how to store an object in a file via an archive:

```
CArchive ar(&theFile, CArchive::store);
WORD wEmployeeID;
...
ar << wEmployeeID;
```

To load an object from a value previously stored in a file

- The following example shows how to load an object from a value previously stored in a file:

```
CArchive ar(&theFile, CArchive::load);
WORD wEmployeeID;
...
ar >> wEmployeeID;
```

Usually, you store and load data to and from a file via an archive in the `Serialize` functions of **CObject**-derived classes, which you must have declared with the **DECLARE_SERIALIZE** macro. A reference to a **CArchive** object is passed to your `Serialize` function. You call the **IsLoading** function of the **CArchive** object to determine whether the `Serialize` function has been called to load data from the file or store data to the file.

The `Serialize` function of a serializable **CObject**-derived class typically has the following form:

```
void CPerson::Serialize(CArchive& ar)
{
    CObject::Serialize(ar);
    if (ar.IsStoring())
    {
        // TODO:  add storing code here
    }
    else
    {
    // TODO:  add loading code here
    }
}
```

The above code template is exactly the same as the one AppWizard creates for the `Serialize` function of the document (a class derived from **CDocument)**. This code template helps you write code that is easier to review, because the storing code and the loading code should always be parallel, as in the following example:

```
void CPerson:Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar << m_strName;
        ar << m_wAge;
    }
    else
    {
        ar >> m_strName;
        ar >> m_wAge;
    }
}
```

The library defines **<<** and **>>** operators for **CArchive** as the first operand and the following data types and class types as the second operand:

CObject*	**SIZE and CSize**	**float**
WORD	**CString**	**POINT** and **CPoint**
DWORD	**BYTE**	**RECT** and **CRect**
double	**LONG**	**CTime** and **CTimeSpan**
int	**COleCurrency**	**COleVariant**
COleDateTime	**COleDateTimeSpan**	

Note

Storing and loading **CObject**s via an archive requires extra consideration. For more information, see Storing and Loading CObjects via an Archive.

The **CArchive** **<<** and **>>** operators always return a reference to the **CArchive** object, which is the first operand. This enables you to chain the operators, as illustrated below:

```
BYTE bSomeByte;
WORD wSomeWord;
DWORD wSomeDoubleWord;
...
ar << bSomeByte << wSomeWord << wSomeDoubleWord;
```

See Also

Concepts

Serialization: Serializing an Object

Storing and Loading CObjects via an Archive

Storing and loading **CObject**s via an archive requires extra consideration. In certain cases, you should call the `Serialize` function of the object, where the **CArchive** object is a parameter of the `Serialize` call, as opposed to using the **<<** or **>>** operator of the **CArchive**. The important fact to keep in mind is that the **CArchive >>** operator constructs the **CObject** in memory based on **CRuntimeClass** information previously written to the file by the storing archive.

Therefore, whether you use the **CArchive <<** and **>>** operators, versus calling `Serialize`, depends on whether you *need* the loading archive to dynamically reconstruct the object based on previously stored **CRuntimeClass** information. Use the `Serialize` function in the following cases:

- When deserializing the object, you know the exact class of the object beforehand.
- When deserializing the object, you already have memory allocated for it.

Caution
If you load the object using the `Serialize` function, you must also store the object using the `Serialize` function. Don't store using the **CArchive <<** operator and then load using the `Serialize` function, or store using the `Serialize` function and then load using **CArchive >>** operator.

The following example illustrates the cases:

```
class CMyObject : public CObject
{
// ...Member functions
   public:
   CMyObject() { }
   virtual void Serialize( CArchive& ar ) { }

// Implementation
   protected:
   DECLARE_SERIAL( CMyObject )
};


class COtherObject : public CObject
{
   // ...Member functions
   public:
   COtherObject() { }
   virtual void Serialize( CArchive& ar ) { }

// Implementation
protected:
   DECLARE_SERIAL( COtherObject )
};


class CCompoundObject : public CObject
{
   // ...Member functions
   public:
   CCompoundObject();
   virtual void Serialize( CArchive& ar );

// Implementation
protected:
   CMyObject m_myob;    // Embedded object
   COtherObject* m_pOther;    // Object allocated in constructor
   CObject* m_pObDyn;    // Dynamically allocated object
   //..Other member data and implementation

   DECLARE_SERIAL( CCompoundObject )
```

```
};

IMPLEMENT_SERIAL(CMyObject,CObject,1)
IMPLEMENT_SERIAL(COtherObject,CObject,1)
IMPLEMENT_SERIAL(CCompoundObject,CObject,1)


CCompoundObject::CCompoundObject()
{
   m_pOther = new COtherObject; // Exact type known and object already
            //allocated.
   m_pObDyn = NULL;    // Will be allocated in another member function
            // if needed, could be a derived class object.
}

void CCompoundObject::Serialize( CArchive& ar )
{
   CObject::Serialize( ar );    // Always call base class Serialize.
   m_myob.Serialize( ar );    // Call Serialize on embedded member.
   m_pOther->Serialize( ar );    // Call Serialize on objects of known exact type.

   // Serialize dynamic members and other raw data
   if ( ar.IsStoring() )
   {
      ar << m_pObDyn;
      // Store other members
   }
   else
   {
      ar >> m_pObDyn; // Polymorphic reconstruction of persistent
            // object
            //load other members
   }
}
```

In summary, if your serializable class defines an embedded **CObjec**t as a member, you should *not* use the **CArchive <<** and **>>** operators for that object, but should call the `Serialize` function instead. Also, if your serializable class defines a pointer to a **CObject** (or an object derived from **CObject**) as a member, but constructs this other object in its own constructor, you should also call `Serialize`.

See Also

Concepts

Serialization: Serializing an Object

Serialization: Serialization vs. Database Input/Output

This article explains when to use document objects and serialization for file-based input/output (I/O) and when other I/O techniques are appropriate — because the application reads and writes data on a per-transaction basis, as in database applications. If you don't use serialization, you also won't need the File Open, Save, and Save As commands. Topics covered include:

- Recommendations for handling input/output
- Handling the File menu in database applications

See Also

Concepts

Serialization in MFC

Recommendations for Handling Input/Output

Whether you use file-based I/O or not depends on how you respond to the questions in the following decision tree:

Does the primary data in your application reside in a disk file?

- Yes, the primary data resides in a disk file:

  Does the application read the whole file into memory on File Open and write the whole file back to disk on File Save?

  - Yes: This is the default MFC document case. Use **CDocument** serialization.
  - No: This is typically the case of transaction-based updating of the file. You update the file on a per-transaction basis and don't need **CDocument** serialization.

- No, the primary data doesn't reside in a disk file:

  Does the data reside in an ODBC data source?

  - Yes, the data resides in an ODBC data source:

    Use MFC's database support. The standard MFC implementation for this case includes a **CDocument** object that stores a **CDatabase** object, as discussed in the article What Is the MFC Database Programming Model?. The application might also read and write an auxiliary file — the purpose of the application wizard "both a database view and file support" option. In this case, you'd use serialization for the auxiliary file.

  - No, the data doesn't reside in an ODBC data source.

    Examples of this case: the data resides in a non-ODBC DBMS; the data is read via some other mechanism, such as OLE or DDE.

    In such cases, you won't use serialization, and your application won't have Open and Save menu items. You might still want to use a **CDocument** as a home base, just as an MFC ODBC application uses the document to store **CRecordset** objects. But you won't use the framework's default File Open/Save document serialization.

To support the Open, Save, and Save As commands on the File menu, the framework provides document serialization. Serialization reads and writes data, including objects derived from class **CObject**, to permanent storage, normally a disk file. Serialization is easy to use and serves many of your needs, but it may be inappropriate in many data-access applications. Data-access applications typically update data on a per-transaction basis. They update the records affected by the transaction rather than reading and writing a whole data file at once.

For information about serialization, see Serialization.

See Also

Concepts

Serialization: Serialization vs. Database Input/Output

The File Menu in an MFC Database Application

If you create an MFC database application and don't use serialization, how should you interpret the Open, Close, Save, and Save As commands on the File menu? While there are no style guidelines for this question, here are a few suggestions:

- Eliminate the File menu's Open command entirely.
- Interpret the Open command as "open database" and show the user a list of data sources your application recognizes.
- Interpret the Open command as, perhaps, "open profile." Retain Open for opening a serialized file, but use the file to store a serialized document containing "user profile" information, such as the user's preferences, including his or her login ID (optionally excluding the password) and the data source he or she most recently worked with.

The MFC Application Wizard supports creating an application with no document-related File menu commands. Select the **Database view without file support** option on the **Database Support** page.

To interpret a File menu command in a special way, you must override one or more command handlers, mostly in your **CWinApp**-derived class. For example, if you completely override **OnFileOpen** (which implements the **ID_FILE_OPEN** command) to mean "open database:"

- Don't call the base class version of **OnFileOpen**, since you're completely replacing the framework's default implementation of the command.
- Use the handler instead to display a dialog box listing data sources. You can display such a dialog by calling **CDatabase::OpenEx** or **CDatabase::Open** with the parameter **NULL**. This opens an ODBC dialog box that displays all available data sources on the user's machine.
- Because database applications typically don't save a whole document, you'll probably want to remove the Save and Save As implementations unless you use a serialized document to store profile information. Otherwise, you might implement the Save command as, for example, "commit transaction." See Technical Note 22 for more information about overriding these commands.

See Also

Concepts

Serialization: Serialization vs. Database Input/Output

Unicode in MFC

MFC supports the Unicode standard for encoding wide characters on Windows NT, Windows 2000, and Windows XP platforms. Unicode applications cannot run on Windows 98 platforms.

Note
The Unicode versions of the MFC libraries are not copied to your hard drive unless you select them during a Custom installation. They are not copied during other types of installation. If you attempt to build or run an MFC Unicode application without the MFC Unicode files, you may get errors.

Note
To copy the files to your hard drive, rerun Setup and click **Add/Remove Features**. Expand the feature tree **Language Tools**, then **Visual C++**, then **Visual C++ Class & Template Libraries**, and select both **ATL MFC Shared Libraries Unicode** and **ATL MFC Static Libraries Unicode**.

The Unicode versions of the MFC libraries are described below:

Static Link Libraries

Release	Debug	Description
UAFXCW.lib, .pdb	UAFXCWD.lib, .pdb	Unicode MFC static link library

Dynamic-Link Libraries

Release	Debug	Description
MFC80U.lib, .dbg, def, .dll, .map, .pdb, .prf	MFC80UD.lib, .def, .dll, .map, .pdb	Unicode MFC import library (see notes below for explanation of file extensions)
	MFCD80UD.lib, .def, .dll, .map, .pdb	Unicode MFC import library for database
	MFCN80UD.lib, .def, .dll, .map, .pdb	Unicode MFC import library for network (sockets)
	MFCO80UD.lib, .def, .dll, .map, .pdb	Unicode MFC DLL for Active technologies
MFCS80U.lib, .pdb	MFCS80UD.lib, .pdb	Unicode MFC import library containing code that must be statically linked in an application or DLL

File Types

- Import library files have the extension (.lib).
- Dynamic-link library files have the extension (.dll).
- Module definition (.def) files are text files that contain statements for defining an .exe or .dll.
- Map (.map) files are text files that contain information that the linker uses when linking a program.
- Library (.lib) files are used in conjunction with the DLL versions of MFC. These files contain code that must be statically linked in the application or DLL.
- Program database (.pdb) files contain debugging and project state information.
- Debug (.dbg) files contain information (COFF FPO, and CodeView) that the Visual C++ Debugger uses.

For detailed information on naming conventions, see Naming Conventions for MFC DLLs and Library Naming Conventions.

For information on using Unicode with MFC, see International Programming and

Strings: Unicode and Multibyte Character Set (MBCS) Support.

See Also

Concepts

General MFC Topics

Other Resources

MFC Concepts

User Interface

For information about creating the user interface for your application with the Microsoft Foundation Class (MFC) Library, see the following topics.

In This Section

ActiveX Controls

Describes using reusable software component based on the Component Object Model (COM) that supports a wide variety of OLE functionality and can be customized to fit many software needs

Clipboard Topics

Describes how to implement support for the Windows Clipboard in MFC applications.

Controls

Describes Windows common controls, including owner-drawn controls, ActiveX controls, and other control classes supplied by the MFC Library.

Control Bars

Describes functionality of toolbars, status bars, and dialog bars.

Dialog Bars

Describes a kind of control bar that can contain any kind of control.

Dialog Boxes

Describes creating dialog boxes with the editors and code wizards.

Document/View Architecture

Describes data management in MFC.

Form Topics

Describes adding forms support to your application.

HTML Help: Context-Sensitive Help for Your Programs

Describes adding context-sensitive help to your applications with HTML Help.

Menus

Describes adding menus to your user interface.

OLE

Provides links to topics discussing object linking and embedding.

Printing and Print Preview

Describes MFC support for printing and print preview from your applications.

Property Sheets

Describes how to use property sheets to manage large numbers of control in a dialog box.

Status Bars

Describes how to use status bars in your applications.

Tool Tips

Describes how to implement tool tips to assist users in using your applications.

Toolbars

Describes the fundamentals of using toolbars in MFC.

Windows

Describes the fundamentals of using windows in MFC.

WinHelp: Context-Sensitive Help for Your Programs

Describes adding context-sensitive help to your applications with WinHelp.

Related Sections

MFC Reference

Provides reference material for the MFC Library, a set of classes in that constitute an application framework, which is the framework of an application written for the Windows API.

General Concepts

Provides links to topics describing conceptual information about the Visual C++ libraries and topics discussing various coding technologies and techniques.

ActiveX Controls

In Visual C++ you can create ActiveX controls using MFC or ATL.

- MFC ActiveX Controls
- ATL

See Also

Other Resources

General Concepts
User Interface

MFC ActiveX Controls

An ActiveX control is a reusable software component based on the Component Object Model (COM) that supports a wide variety of OLE functionality and can be customized to fit many software needs. ActiveX controls are designed for use both in ordinary ActiveX control containers and on the Internet, in World Wide Web pages. You can create ActiveX controls either with MFC, described here, or with the Active Template Library (ATL).

An ActiveX control can draw itself in its own window, respond to events (such as mouse clicks), and be managed through an interface that includes properties and methods similar to those in Automation objects.

These controls can be developed for many uses, such as database access, data monitoring, or graphing. Besides their portability, ActiveX controls support features previously not available to ActiveX controls, such as compatibility with existing OLE containers and the ability to integrate their menus with the OLE container menus. In addition, an ActiveX control fully supports Automation, which allows the control to expose read\write properties and a set of methods that can be called by the control user.

You can create windowless ActiveX controls and controls that only create a window when they become active. Windowless controls speed up the display of your application and make it possible to have transparent and nonrectangular controls. You can also load ActiveX control properties asynchronously.

An ActiveX control is implemented as an in-process server (typically a small object) that can be used in any OLE container. Note that the full functionality of an ActiveX control is available only when used within an OLE container designed to be aware of ActiveX controls. See Port ActiveX Controls to Other Applications for a list of containers that support ActiveX controls. This container type, hereafter called a "control container," can operate an ActiveX control by using the control's properties and methods, and receives notifications from the ActiveX control in the form of events. The following figure demonstrates this interaction.

Interaction Between an ActiveX Control Container and a Windowed ActiveX Control

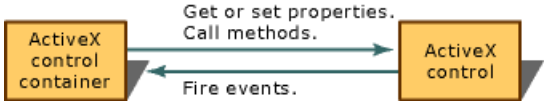

For some recent information on optimizing your ActiveX controls, see MFC ActiveX Controls: Optimization.

To create an MFC ActiveX control, see Create an ActiveX control project.

For more information, see:

- ActiveX Control Containers
- Active Documents
- Using ActiveX Controls
- Understanding ActiveX Controls
- Upgrading an Existing ActiveX Control to be Used on the Internet
- Adding Context-Sensitive Help to an MFC ActiveX Control

Basic Components of an ActiveX Control

An ActiveX control uses several programmatic elements to interact efficiently with a control container and with the user. These are class COleControl, a set of event-firing functions, and a dispatch map.

Every ActiveX control object you develop inherits a powerful set of features from its MFC base class, **COleControl**. These features include in-place activation, and Automation logic. **COleControl** can provide the control object with the same functionality as an MFC window object, plus the ability to fire events. **COleControl** can also provide windowless controls, which rely on their container for help with some of the functionality a window provides (mouse capture, keyboard focus, scrolling), but offer much faster display.

Because the control class derives from **COleControl**, it inherits the capability to send, or "fire," messages, called events, to the control container when certain conditions are met. These events are used to notify the control container when something important happens in the control. You can send additional information about an event to the control container by attaching parameters to the event. For more information about ActiveX control events, see the article MFC ActiveX Controls: Events.

The final element is a dispatch map, which is used to expose a set of functions (called methods) and attributes (called

properties) to the control user. Properties allow the control container or the control user to manipulate the control in various ways. The user can change the appearance of the control, change certain values of the control, or make requests of the control, such as accessing a specific piece of data that the control maintains. This interface is determined by the control developer and is defined using **Class View**. For more information on ActiveX control methods and properties, see the articles MFC ActiveX Controls: Methods and Properties.

Interaction Between Controls with Windows and ActiveX Control Containers

When a control is used within a control container, it uses two mechanisms to communicate: it exposes properties and methods, and it fires events. The following figure demonstrates how these two mechanisms are implemented.

Communication Between an ActiveX Control Container and an ActiveX Control

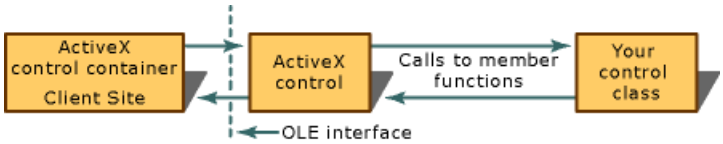

The previous figure also illustrates how other OLE interfaces (besides automation and events) are handled by controls.

All of a control's communication with the container is performed by **COleControl**. To handle some of the container's requests, **COleControl** will call member functions that are implemented in the control class. All methods and some properties are handled in this way. Your control's class can also initiate communication with the container by calling member functions of **COleControl**. Events are fired in this manner.

Active and Inactive States of an ActiveX Control

A control has two basic states: active and inactive. Traditionally, these states were distinguished by whether the control had a window. An active control had a window; an inactive control did not. With the introduction of windowless activation, this distinction is no longer universal, but still applies to many controls.

When a windowless control goes active, it invokes mouse capture, keyboard focus, scrolling, and other window services from its container. You can also provide mouse interaction to inactive controls, as well as create controls that wait until activated to create a window.

When a control with a window becomes active, it is able to interact fully with the control container, the user, and Windows. The figure below demonstrates the paths of communication between the ActiveX control, the control container, and the operating system.

Windows Message Processing in a Windowed ActiveX Control (When Active)

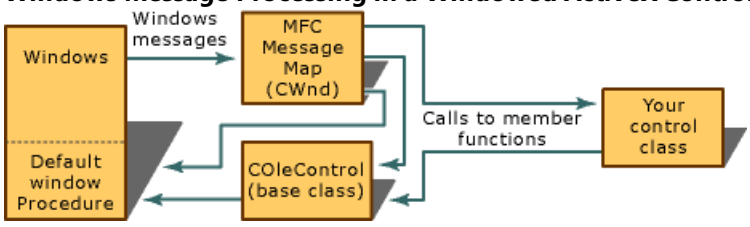

Serialization

The ability to serialize data, sometimes referred to as persistence, allows the control to write the value of its properties to persistent storage. Controls can then be recreated by reading the object's state from the storage.

Note that a control is not responsible for obtaining access to the storage medium. Instead, the control's container is responsible for providing the control with a storage medium to use at the appropriate times. For more information on serialization, see the article MFC ActiveX Controls: Serializing. For information on optimizing serialization, see Optimizing Persistence and Initialization in ActiveX Controls: Optimization.

Installing ActiveX Control Classes and Tools

When you install Visual C++, the MFC ActiveX control classes and retail and debug ActiveX control run-time DLLs are automatically installed if ActiveX controls are selected in Setup (they are selected by default).

By default, the ActiveX control classes and tools are installed in the following subdirectories under \Program Files\Microsoft Visual Studio .NET:

- **\Common7\Tools**

  Contains the Test Container files (TstCon32.exe, as well as its Help files).

- **\Vc7\atlmfc\include**

  Contains the include files needed to develop ActiveX controls with MFC

- **\Vc7\atlmfc\src\mfc**

  Contains the source code for specific ActiveX control classes in MFC

- **\Vc7\atlmfc\lib**

  Contains the libraries required to develop ActiveX controls with MFC

There are also samples for MFC ActiveX controls. For more information about these samples, see Controls Samples: MFC-Based ActiveX Controls

See Also

Other Resources

General Concepts
User Interface

MFC ActiveX Controls: Optimization

This article explains techniques you can use to optimize your ActiveX controls for better performance.

The topics Turning Off the Activate When Visible Option and Providing Mouse Interaction While Inactive discuss controls that don't create a window until activated. The topic Providing Windowless Activation discusses controls that never create a window, even when they are activated.

Windows have two major drawbacks for OLE objects: they prevent objects from being transparent or nonrectangular when active, and they add a large overhead to the instantiation and display of controls. Typically, creating a window takes 60 percent of a control's creation time. With a single shared window (usually the container's) and some dispatching code, a control receives the same window services, generally without a loss of performance. Having a window is mostly unnecessary overhead for the object.

Some optimizations do not necessarily improve performance when your control is used in certain containers. For example, containers released prior to 1996 did not support windowless activation, so implementing this feature will not provide a benefit in older containers. However, nearly every container supports persistence, so optimizing your control's persistence code will likely improve its performance in any container. If your control is specifically intended to be used with one particular type of container, you may want to research which of these optimizations is supported by that container. In general, however, you should try to implement as many of these techniques as are applicable to your particular control to ensure your control performs as well as it possibly can in a wide array of containers.

You can implement many of these optimizations through the MFC ActiveX Control Wizard, on the Control Settings page.

MFC ActiveX Control Wizard OLE Optimization Options

Control setting in the MFC ActiveX Control Wizard	Action	More information
Activate when visible check box	Clear	Turning Off the Activate When Visible Option
Windowless activation check box	Select	Providing Windowless Activation
Unclipped device context check box	Select	Using an Unclipped Device Context
Flicker-free activation check box	Select	Providing Flicker-Free Activation
Mouse pointer notifications when inactive check box	Select	Providing Mouse Interaction While Inactive
Optimized drawing code check box	Select	Optimizing Control Drawing

For detailed information about the member functions that implement these optimizations, see COleControl. The member functions are listed by use, such as Windowless Operations and Inactive Pointer Handling Functions.

For more information, see:

- Optimizing Persistence and Initialization
- Providing Windowless Activation
- Turning Off the Activate When Visible Option
- Providing Mouse Interaction While Inactive
- Providing Flicker-Free Activation
- Using an Unclipped Device Context
- Optimizing Control Drawing

See Also

Concepts

MFC ActiveX Controls

Optimizing Persistence and Initialization

By default, persistence and initialization in a control are handled by the `DoPropExchange` member function. In a typical control, this function contains calls to several **PX_** functions (**PX_Color**, **PX_Font**, and so on), one for each property.

This approach has the advantage that a single `DoPropExchange` implementation can be used for initialization, for persistence in binary format, and for persistence in the so-called "property-bag" format used by some containers. This one function provides all information about the properties and their default values in one convenient place.

However, this generality comes at the expense of efficiency. The **PX_** functions get their flexibility through multilayered implementations that are inherently less efficient than more direct, but less flexible, approaches. Furthermore, if a control passes a default value to a **PX_** function, that default value must be provided every time, even in situations when the default value may not necessarily be used. If generating the default value is a nontrivial task (for example, when the value is obtained from an ambient property), then extra, unnecessary work is done in cases where the default value is not used.

You can improve your control's binary persistence performance by overriding your control's `Serialize` function. The default implementation of this member function makes a call to your `DoPropExchange` function. By overriding it, you can provide a more direct implementation for binary persistence. For example, consider this `DoPropExchange` function:

```
void CMyCtrl::DoPropExchange(CPropExchange* pPX)
{
   ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
   COleControl::DoPropExchange(pPX);

   PX_Bool(pPX, _T("BoolProp"), m_boolProp, TRUE);
   PX_Short(pPX, _T("ShortProp"), m_shortProp, 0);
   PX_Color(pPX, _T("ColorProp"), m_colorProp, RGB(0xFF,0x00,0x00));
   PX_String(pPX, _T("StringProp"), m_stringProp, _T(""));
}
```

To improve the performance of this control's binary persistence, you can override the `Serialize` function as follows:

```
void CMyCtrl::Serialize(CArchive& ar)
{
    DWORD dwVersion =
        SerializeVersion(ar, MAKELONG(_wVerMinor, _wVerMajor));
    SerializeExtent(ar);
    SerializeStockProps(ar);

    if (ar.IsLoading())
    {
        ar >> m_boolProp;
        ar >> m_shortProp;
        ar >> m_colorProp;
        ar >> m_stringProp;
    }
    else
    {
        ar << m_boolProp;
        ar << m_shortProp;
        ar << m_colorProp;
        ar << m_stringProp;
    }
}
```

The `dwVersion` local variable can be used to detect the version of the control's persistent state being loaded or saved. You can use this variable instead of calling CPropExchange::GetVersion.

To save a little space in the persistent format for a **BOOL** property (and to keep it compatible with the format produced by **PX_Bool**), you can store the property as a **BYTE**, as follows:

```
    if (ar.IsLoading())
    {
```

```
        BYTE bTmp;
        ar >> bTmp;
        m_boolProp = (BOOL)bTmp;
        .
        .
        .
    }
    else
    {
        ar << (BYTE)m_boolProp;
        .
        .
        .
    }
```

Note that in the load case, a temporary variable is used and then its value is assigned, rather than casting `m_boolProp` to a **BYTE** reference. The casting technique would result in only one byte of `m_boolProp` being modified, leaving the remaining bytes uninitialized.

For the same control, you can optimize the control's initialization by overriding COleControl::OnResetState as follows:

```
void CMyCtrl::OnResetState()
{
    ResetVersion(MAKELONG(_wVerMinor, _wVerMajor));
    ResetStockProps();

    m_boolProp = TRUE;
    m_shortProp = 0;
    m_colorProp = RGB(0xFF,0x00,0x00);
    m_stringProp.Empty();
}
```

Although **Serialize** and **OnResetState** have been overridden, the `DoPropExchange` function should be kept intact because it is still used for persistence in the property-bag format. It is important to maintain all three of these functions to ensure that the control manages its properties consistently, regardless of which persistence mechanism the container uses.

See Also

Concepts

MFC ActiveX Controls: Optimization

Providing Windowless Activation

Window creation code (that is, everything that happens when you call **CreateWindow**) is costly to execute. A control that maintains an on-screen window has to manage messages for the window. Windowless controls are therefore faster than controls with windows.

A further advantage of windowless controls is that, unlike windowed controls, windowless controls support transparent painting and nonrectangular screen regions. A common example of a transparent control is a text control with a transparent background. The controls paints the text but not the background, so whatever is under the text shows through. Newer forms often make use of nonrectangular controls, such as arrows and round buttons.

Often, a control does not need a window of its own and, instead, can use the window services of its container, provided that the container has been written to support windowless objects. Windowless controls are backward compatible with older containers. In older containers not written to support windowless controls, the windowless controls create a window when active.

Because windowless controls do not have their own windows, the container (which does have a window) is responsible for providing services that would otherwise have been provided by the control's own window. For example, if your control needs to query the keyboard focus, capture the mouse, or obtain a device context, these operations are managed by the container. The container routes user input messages sent to its window to the appropriate windowless control, using the **IOleInPlaceObjectWindowless** interface. (See the *ActiveX SDK* for a description of this interface.) **COleControl** member functions invoke these services from the container.

To make your control use windowless activation, include the **windowlessActivate** flag in the set of flags returned by COleControl::GetControlFlags. For example:

```
DWORD CMyCtrl::GetControlFlags()
{
    return COleControl::GetControlFlags() | windowlessActivate;
}
```

The code to include this flag is automatically generated if you select the **Windowless activation** option on the Control Settings page of the MFC ActiveX Control Wizard.

When windowless activation is enabled, the container will delegate input messages to the control's **IOleInPlaceObjectWindowless** interface. **COleControl**'s implementation of this interface dispatches the messages through your control's message map, after adjusting the mouse coordinates appropriately. You can process the messages like ordinary window messages, by adding the corresponding entries to the message map. In your handlers for these messages, avoid using the `m_hWnd` member variable (or any member function that uses it) without first checking that its value is not **NULL**.

COleControl provides member functions that invoke mouse capture, keyboard focus, scrolling, and other window services from the container as appropriate, including:

- GetFocus, SetFocus
- GetCapture, SetCapture, ReleaseCapture
- GetDC, ReleaseDC
- InvalidateRgn
- ScrollWindow
- GetClientRect

In windowless controls, you should always use the **COleControl** member functions instead of the corresponding **CWnd** member functions or their related Win32 API functions.

You may want a windowless control to be the target of an OLE drag-and-drop operation. Normally, this would require that the control's window be registered as a drop target. Since the control has no window of its own, the container uses its own window as a drop target. The control provides an implementation of the **IDropTarget** interface to which the container can delegate calls at the appropriate time. To expose this interface to the container, override COleControl::GetWindowlessDropTarget. For example:

```
IDropTarget* CMyCtrl::GetWindowlessDropTarget()
```

```
{
    m_xDropTarget.AddRef();
    return &m_xDropTarget;
}
```

See Also

Concepts

MFC ActiveX Controls: Optimization

Turning off the Activate When Visible Option

A control has two basic states: active and inactive. Traditionally, these states were distinguished by whether the control had a window. An active control had a window; an inactive control did not. With the introduction of windowless activation, this distinction is no longer universal, but still applies to many controls.

Compared with the rest of the initialization typically performed by an ActiveX control, the creation of a window is an extremely expensive operation. Ideally, a control would defer creating its window until absolutely necessary.

Many controls do not need to be active the entire time they are visible in a container. Often, a control can remain in the inactive state until the user performs an operation that requires it to become active (for example, clicking with the mouse or pressing the TAB key). To cause a control to remain inactive until the container needs to activate it, remove the **OLEMISC_ACTIVATEWHENVISIBLE** flag from the control's miscellaneous flags:

```
static const DWORD BASED_CODE _dwMyOleMisc =
   OLEMISC_SETCLIENTSITEFIRST |
   OLEMISC_INSIDEOUT |
   OLEMISC_CANTLINKINSIDE |
   OLEMISC_RECOMPOSEONRESIZE;
```

The **OLEMISC_ACTIVATEWHENVISIBLE** flag is automatically omitted if you turn off the **Activate When Visible** option in the Control Settings page of the MFC ActiveX Control Wizard when you create your control.

See Also
Concepts
MFC ActiveX Controls: Optimization

Providing Mouse Interaction While Inactive

If your control is not immediately activated, you may still want it to process **WM_SETCURSOR** and **WM_MOUSEMOVE** messages, even though the control has no window of its own. This can be accomplished by enabling **COleControl**'s implementation of the **IPointerInactive** interface, which is disabled by default. (See the *ActiveX SDK* for a description of this interface.) To enable it, include the `pointerInactive` flag in the set of flags returned by COleControl::GetControlFlags:

```
DWORD CMyCtrl::GetControlFlags()
{
    return COleControl::GetControlFlags() | pointerInactive;
}
```

The code to include this flag is automatically generated if you select the **Mouse Pointer Notifications When Inactive** option on the Control Settings page when creating your control with the **MFC ActiveX Control Wizard**.

When the **IPointerInactive** interface is enabled, the container delegates **WM_SETCURSOR** and **WM_MOUSEMOVE** messages to it. **COleControl**'s implementation of **IPointerInactive** dispatches the messages through your control's message map after adjusting the mouse coordinates appropriately. You can process the messages just like ordinary window messages by adding the corresponding entries to the message map. In your handlers for these messages, avoid using the `m_hWnd` member variable (or any member function that uses it) without first checking that its value is not **NULL**.

You may also want an inactive control to be the target of an OLE drag-and-drop operation. This requires activating the control at the moment the user drags an object over it, so that the control's window can be registered as a drop target. To cause activation to occur during a drag, override COleControl::GetActivationPolicy, and return the **POINTERINACTIVE_ACTIVATEONDRAG** flag:

```
DWORD CMyCtrl::GetActivationPolicy()
{
    return POINTERINACTIVE_ACTIVATEONDRAG;
}
```

Enabling the **IPointerInactive** interface typically means that you want the control to be capable of processing mouse messages at all times. To get this behavior in a container that doesn't support the **IPointerInactive** interface, you need to have your control always activated when visible, which means the control should include the **OLEMISC_ACTIVATEWHENVISIBLE** flag among its miscellaneous flags. However, to prevent this flag from taking effect in a container that does support **IPointerInactive**, you can also specify the **OLEMISC_IGNOREACTIVATEWHENVISIBLE** flag:

```
static const DWORD BASED_CODE _dwMyOleMisc =
    OLEMISC_ACTIVATEWHENVISIBLE |
    OLEMISC_IGNOREACTIVATEWHENVISIBLE |
    OLEMISC_SETCLIENTSITEFIRST |
    OLEMISC_INSIDEOUT |
    OLEMISC_CANTLINKINSIDE |
    OLEMISC_RECOMPOSEONRESIZE;
```

See Also

Concepts

MFC ActiveX Controls: Optimization

Providing Flicker-Free Activation

If your control draws itself identically in the inactive and active states (and does not use windowless activation), you can eliminate the drawing operations and the accompanying visual flicker that normally occur when making the transition between the inactive and active states. To do this, include the **noFlickerActivate** flag in the set of flags returned by COleControl::GetControlFlags. For example:

```
DWORD CMyCtrl::GetControlFlags()
{
    return COleControl::GetControlFlags() | noFlickerActivate;
}
```

The code to include this flag is automatically generated if you select the **Flicker-Free activation** option on the Control Settings page when creating your control with the MFC ActiveX Control Wizard.

If you are using windowless activation, this optimization has no effect.

See Also

Concepts

MFC ActiveX Controls: Optimization

Using an Unclipped Device Context

If you are absolutely certain that your control does not paint outside its client rectangle, you can realize a small but detectable speed gain by disabling the call to **IntersectClipRect** that is made by **COleControl**. To do this, remove the **clipPaintDC** flag from the set of flags returned by COleControl::GetControlFlags. For example:

```
DWORD CMyCtrl::GetControlFlags()
{
    return COleControl::GetControlFlags() & ~clipPaintDC;
}
```

The code to remove this flag is automatically generated if you select the **Unclipped Device Context** option on the Control Settings page, when creating your control with the MFC ActiveX Control Wizard.

If you are using windowless activation, this optimization has no effect.

See Also

Concepts

MFC ActiveX Controls: Optimization

Optimizing Control Drawing

When a control is instructed to draw itself into a container-supplied device context, it typically selects GDI objects (such as pens, brushes, and fonts) into the device context, performs its drawing operations, and restores the previous GDI objects. If the container has multiple controls that are to be drawn into the same device context, and each control selects the GDI objects it requires, time can be saved if the controls do not individually restore previously selected objects. After all the controls have been drawn, the container can automatically restore the original objects.

To detect whether a container supports this technique, a control can call the COleControl::IsOptimizedDraw member function. If this function returns **TRUE**, the control can skip the normal step of restoring the previously selected objects.

Consider a control that has the following (unoptimized) `OnDraw` function:

```
void CMyCtrl::OnDraw(CDC* pdc, CRect& rcBounds, CRect& rcInvalid)
{
    CPen pen(PS_SOLID, 0, TranslateColor(GetForeColor()));
    CBrush brush(TranslateColor(GetBackColor()));
    CPen* pPenSave = pdc->SelectObject(&pen);
    CBrush* pBrushSave = pdc->SelectObject(&brush);
    pdc->Rectangle(rcBounds);
    pdc->SelectObject(pPenSave);
    pdc->SelectObject(pBrushSave);
}
```

The pen and brush in this example are local variables, meaning their destructors will be called when they go out of scope (when the `OnDraw` function ends). The destructors will attempt to delete the corresponding GDI objects. But they should not be deleted if you plan to leave them selected into the device context upon returning from `OnDraw`.

To prevent the CPen and CBrush objects from being destroyed when `OnDraw` finishes, store them in member variables instead of local variables. In the control's class declaration, add declarations for two new member variables:

```
class CMyCtrl : public COleControl
{
    .
    .
    .
    CPen m_pen;
    CBrush m_brush;
}
```

Then, the `OnDraw` function can be rewritten as follows:

```
void CMyCtrl::OnDraw(CDC* pdc, CRect& rcBounds, CRect& rcInvalid)
{
    if (m_pen.m_hObject == NULL)
        m_pen.CreatePen(PS_SOLID, 0, TranslateColor(GetForeColor()));
    if (m_Brush.m_hObject == NULL)
        m.Brush.CreateSolidBrush(TranslateColor(GetBackColor()));
    CPen* pPenSave = pdc->SelectObject(&m_pen);
    CBrush* pBrushSave = pdc->SelectObject(&m_brush);
    pdc->Rectangle(rcBounds);
    pdc->SelectObject(pPenSave);
    pdc->SelectObject(pBrushSave);
}
```

This approach avoids creation of the pen and brush every time `OnDraw` is called. The speed improvement comes at the cost of maintaining additional instance data.

If the ForeColor or BackColor property changes, the pen or brush needs to be created again. To do this, override the OnForeColorChanged and OnBackColorChanged member functions:

```
void CMyCtrl::OnForeColorChanged()
```

```
{
    m_pen.DeleteObject();
}

void CMyCtrl::OnBackColorChanged()
{
    m_brush.DeleteObject();
}
```

Finally, to eliminate unnecessary **SelectObject** calls, modify `OnDraw` as follows:

```
void CMyCtrl::OnDraw(CDC* pdc, CRect& rcBounds, CRect& rcInvalid)
{
    if (m_pen.m_hObject == NULL)
        m_pen.CreatePen(PS_SOLID, 0, TranslateColor(GetForeColor()));
    if (m_Brush.m_hObject == NULL)
        m.Brush.CreateSolidBrush(TranslateColor(GetBackColor()));
    CPen* pPenSave = pdc->SelectObject(&m_pen);
    CBrush* pBrushSave = pdc->SelectObject(&m_brush);
    pdc->Rectangle(rcBounds);
    if (! IsOptimizedDraw())
    {
        pdc->SelectObject(pPenSave);
        pdc->SelectObject(pBrushSave);
    }
}
```

See Also

Reference

COleControl Class

MFC ActiveX Control Wizard

Concepts

MFC ActiveX Controls: Optimization

MFC ActiveX Controls

MFC ActiveX Controls

MFC ActiveX Controls: Painting an ActiveX Control

MFC ActiveX Controls: Painting an ActiveX Control

This article describes the ActiveX control painting process and how you can alter paint code to optimize the process. (See Optimizing Control Drawing for techniques on how to optimize drawing by not having controls individually restore previously selected GDI objects. After all of the controls have been drawn, the container can automatically restore the original objects.)

Examples in this article are from a control created by the MFC ActiveX Control Wizard with default settings. For more information on creating a skeleton control application using the MFC ActiveX Control Wizard, see the article MFC ActiveX Control Wizard.

The following topics are covered:

- The overall process for painting a control and the code created by the ActiveX Control Wizard to support painting
- How to optimize the painting process
- How to paint your control using metafiles

The Painting Process of an ActiveX Control

When ActiveX controls are initially displayed or are redrawn, they follow a painting process similar to other applications developed using MFC, with one important distinction: ActiveX controls can be in an active or an inactive state.

An active control is represented in an ActiveX control container by a child window. Like other windows, it is responsible for painting itself when a **WM_PAINT** message is received. The control's base class, COleControl, handles this message in its **OnPaint** function. This default implementation calls the `OnDraw` function of your control.

An inactive control is painted differently. When the control is inactive, its window is either invisible or nonexistent, so it can not receive a paint message. Instead, the control container directly calls the `OnDraw` function of the control. This differs from an active control's painting process in that the **OnPaint** member function is never called.

As discussed in the preceding paragraphs, how an ActiveX control is updated depends on the state of the control. However, because the framework calls the `OnDraw` member function in both cases, you add the majority of your painting code in this member function.

The `OnDraw` member function handles control painting. When a control is inactive, the control container calls `OnDraw`, passing the device context of the control container and the coordinates of the rectangular area occupied by the control.

The rectangle passed by the framework to the `OnDraw` member function contains the area occupied by the control. If the control is active, the upper-left corner is (0, 0) and the device context passed is for the child window that contains the control. If the control is inactive, the upper-left coordinate is not necessarily (0, 0) and the device context passed is for the control container containing the control.

Note

It is important that your modifications to `OnDraw` do not depend on the rectangle's upper left point being equal to (0, 0) and t hat you draw only inside the rectangle passed to `OnDraw`. Unexpected results can occur if you draw beyond the rectangle's ar ea.

The default implementation provided by the MFC ActiveX Control Wizard in the control implementation file (.CPP), shown below, paints the rectangle with a white brush and fills the ellipse with the current background color.

```
void CSampleCtrl::OnDraw( CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid )
{
    pdc->FillRect( rcBounds,
        CBrush::FromHandle((HBRUSH)GetStockObject(WHITE_BRUSH)));
    pdc->Ellipse( rcBounds );
}
```

Note

When painting a control, you should not make assumptions about the state of the device context that is passed as the *pdc* parameter to the `OnDraw` function. Occasionally the device context is supplied by the container application and will not necessarily be initialized to the default state. In particular, explicitly select the pens, brushes, colors, fonts, and other resources that your drawing code depends upon.

Optimizing Your Paint Code

After the control is successfully painting itself, the next step is to optimize the `OnDraw` function.

The default implementation of ActiveX control painting paints the entire control area. This is sufficient for simple controls, but in many cases repainting the control would be faster if only the portion that needed updating was repainted, instead of the entire control.

The `OnDraw` function provides an easy method of optimization by passing `rcInvalid`, the rectangular area of the control that needs redrawing. Use this area, usually smaller than the entire control area, to speed up the painting process.

Painting Your Control Using Metafiles

In most cases the `pdc` parameter to the `OnDraw` function points to a screen device context (DC). However, when printing images of the control or during a print preview session, the DC received for rendering is a special type called a "metafile DC". Unlike a screen DC, which immediately handles requests sent to it, a metafile DC stores requests to be played back at a later time. Some container applications may also choose to render the control image using a metafile DC when in design mode.

Metafile drawing requests can be made by the container through two interface functions: **IViewObject::Draw** (this function can also be called for non-metafile drawing) and **IDataObject::GetData**. When a metafile DC is passed as one of the parameters, the MFC framework makes a call to COleControl::OnDrawMetafile. Because this is a virtual member function, override this function in the control class to do any special processing. The default behavior calls **COleControl::OnDraw**.

To make sure the control can be drawn in both screen and metafile device contexts, you must use only member functions that are supported in both a screen and a metafile DC. Be aware that the coordinate system may not be measured in pixels.

Because the default implementation of **OnDrawMetafile** calls the control's `OnDraw` function, use only member functions that are suitable for both a metafile and a screen device context, unless you override **OnDrawMetafile**. The following lists the subset of **CDC** member functions that can be used in both a metafile and a screen device context. For more information on these functions, see class CDC in the *MFC Reference*.

Arc	**BibBlt**	**Chord**
Ellipse	**Escape**	**ExcludeClipRect**
ExtTextOut	**FloodFill**	**IntersectClipRect**
LineTo	**MoveTo**	**OffsetClipRgn**
OffsetViewportOrg	**OffsetWindowOrg**	**PatBlt**
Pie	**Polygon**	**Polyline**
PolyPolygon	**RealizePalette**	**RestoreDC**
RoundRect	**SaveDC**	**ScaleViewportExt**
ScaleWindowExt	**SelectClipRgn**	**SelectObject**
SelectPalette	**SetBkColor**	**SetBkMode**
SetMapMode	**SetMapperFlags**	**SetPixel**
SetPolyFillMode	**SetROP2**	**SetStretchBltMode**

SetTextColor	**SetTextJustification**	**SetViewportExt**
SetViewportOrg	**SetWindowExt**	**SetWindowORg**
StretchBlt	**TextOut**	

In addition to **CDC** member functions, there are several other functions that are compatible in a metafile DC. These include CPalette::AnimatePalette, CFont::CreateFontIndirect, and three member functions of **CBrush**: CreateBrushIndirect, CreateDIBPatternBrush, and CreatePatternBrush.

Functions that are not recorded in a metafile are: DrawFocusRect, DrawIcon, DrawText, ExcludeUpdateRgn, FillRect, FrameRect, GrayString, InvertRect, ScrollDC, and TabbedTextOut. Because a metafile DC is not actually associated with a device, you cannot use SetDIBits, GetDIBits, and CreateDIBitmap with a metafile DC. You can use SetDIBitsToDevice and StretchDIBits with a metafile DC as the destination. CreateCompatibleDC, CreateCompatibleBitmap, and CreateDiscardableBitmap are not meaningful with a metafile DC.

Another point to consider when using a metafile DC is that the coordinate system may not be measured in pixels. For this reason, all your drawing code should be adjusted to fit in the rectangle passed to `OnDraw` in the *rcBounds* parameter. This prevents accidental painting outside the control because *rcBounds* represents the size of the control's window.

After you have implemented metafile rendering for the control, use Test Container to test the metafile. See Testing Properties and Events with Test Container for information on how to access the test container.

To test the control's metafile using Test Container

1. On the Test Container's **Edit** menu, click **Insert New Control**.
2. In the **Insert New Control** box, select the control and click **OK**.

   The control will appear in Test container.
3. On the **Control** menu, click **Draw Metafile**.

   A separate window appears in which the metafile is displayed. You can change the size of this window to see how scaling affects the control's metafile. You can close this window at any time.

See Also

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Events

ActiveX controls use events to notify a container that something has happened to the control. Common examples of events include clicks on the control, data entered using the keyboard, and changes in the control's state. When these actions occur, the control fires an event to alert the container.

Events are also called messages.

MFC supports two kinds of events: stock and custom. Stock events are those events that class COleControl handles automatically. For a complete list of stock events, see the article MFC ActiveX Controls: Adding Stock Events. Custom events allow a control the ability to notify the container when an action specific to that control occurs. Some examples would be a change in the internal state of a control or receipt of a certain window message.

For your control to fire events properly, your control class must map each event of the control to a member function that should be called when the related event occurs. This mapping mechanism (called an event map) centralizes information about the event and allows Visual Studio to easily access and manipulate the control's events. This event map is declared by the following macro, located in the header (.H) file of the control class declaration:

```
DECLARE_EVENT_MAP()
```

After the event map has been declared, it must be defined in your control's implementation (.CPP) file. The following lines of code define the event map, allowing your control to fire specific events:

```
BEGIN_EVENT_MAP(CSampleCtrl, COleControl)
//{{AFX_EVENT_MAP(CSampleCtrl)
...
//}}AFX_EVENT_MAP
END_EVENT_MAP()
```

If you use the MFC ActiveX Control Wizard to create the project, it automatically adds these lines. If you do not use the MFC ActiveX Control Wizard, you must add these lines manually.

With Class View, you can add stock events supported by class **COleControl** or custom events that you define. For each new event, Class View automatically adds the proper entry to the control's event map and the control's .IDL file.

Two other articles discuss events in detail:

- MFC ActiveX Controls: Adding Stock Events
- MFC ActiveX Controls: Adding Custom Events

See Also

Reference

COleControl Class

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Methods

MFC ActiveX Controls: Adding Stock Events to an ActiveX Control

Stock events differ from custom events in that they are automatically fired by class COleControl. **COleControl** contains predefined member functions that fire events resulting from common actions. Some common actions implemented by **COleControl** include single- and double-clicks on the control, keyboard events, and changes in the state of the mouse buttons. Event map entries for stock events are always preceded by the **EVENT_STOCK** prefix.

Stock Events Supported by the Add Event Wizard

The **COleControl** class provides ten stock events, listed in the following table. You can specify the events you want in your control using the Add Event Wizard.

Stock Events

Event	Firing function	Comments
Click	**void FireClick( )**	Fired when the control captures the mouse, any **BUTTONUP** (left, middle, or right) message is received, and the button is released over the control. The stock MouseDown and MouseUp events occur before this event. Event map entry: **EVENT_STOCK_CLICK( )**
DblClick	**void FireDblClick( )**	Similar to Click but fired when a **BUTTONDBLCLK** message is received. Event map entry: **EVENT_STOCK_DBLCLICK( )**
Error	**void FireError(SCODE** *scode* **, LPCSTR** *lpszDescription* **, UINT** *nHelpID* **= 0)**	Fired when an error occurs within your ActiveX control outside of the scope of a method call or property access. Event map entry: **EVENT_STOCK_ERROREVENT( )**
KeyDown	**void FireKeyDown(short** *nChar* **, short** *nShiftState* **)**	Fired when a **WM_SYSKEYDOWN** or **WM_KEYDOWN** message is received. Event map entry: **EVENT_STOCK_KEYDOWN( )**
KeyPress	**void FireKeyPress(short*** *pnChar* **)**	Fired when a **WM_CHAR** message is received. Event map entry: **EVENT_STOCK_KEYPRESS( )**
KeyUp	**void FireKeyUp(short** *nChar* **, short** *nShiftState* **)**	Fired when a **WM_SYSKEYUP** or **WM_KEYUP** message is received. Event map entry: **EVENT_STOCK_KEYUP( )**
MouseDown	**void FireMouseDown(short** *nButton* **, short** *nShiftState* **, float** *x* **, float** *y* **)**	Fired if any **BUTTONDOWN** (left, middle, or right) is received. The mouse is captured immediately before this event is fired. Event map entry: **EVENT_STOCK_MOUSEDOWN( )**
MouseMove	**void FireMouseMove(short** *nButton* **, short** *nShiftState* **, float** *x* **, float** *y* **)**	Fired when a **WM_MOUSEMOVE** message is received. Event map entry: **EVENT_STOCK_MOUSEMOVE( )**
MouseUp	**void FireMouseUp(short** *nButton* **, short** *nShiftState* **, float** *x* **, float** *y* **)**	Fired if any **BUTTONUP** (left, middle, or right) is received. The mouse capture is released before this event is fired. Event map entry: **EVENT_STOCK_MOUSEUP( )**
ReadyStateChange	**void FireReadyStateChange( )**	Fired when a control transitions to the next ready state due to the amount of data received. Event map entry: **EVENT_STOCK_READYSTATECHANGE( )**

Adding a Stock Event Using the Add Event Wizard

Adding stock events requires less work than adding custom events because the firing of the actual event is handled automatically by the base class, **COleControl**. The following procedure adds a stock event to a control that was developed using MFC ActiveX Control Wizard. The event, called KeyPress, fires when a key is pressed and the control is active. This procedure can also be used to add other stock events. Substitute the selected stock event name for KeyPress.

To add the KeyPress stock event using the Add Event Wizard

1. Load your control's project.
2. In Class View, right-click your ActiveX control class to open the shortcut menu.
3. From the shortcut menu, click **Add** and then click **Add Event**.

   This opens the Add Event Wizard.
4. In the **Event Name** drop-down list, select **KeyPress**.
5. Click **Finish**.

Add Event Wizard Changes for Stock Events

Because stock events are handled by the control's base class, the Add Event Wizard does not change your class declaration in any way. It adds the event to the control's event map and makes an entry in its .IDL file. The following line is added to the control's event map, located in the control class implementation (.CPP) file:

```
EVENT_STOCK_KEYPRESS()
```

Adding this code fires a KeyPress event when a **WM_CHAR** message is received and the control is active. The KeyPress event can be fired at other times by calling its firing function (for example, `FireKeyPress`) from within the control code.

The Add Event Wizard adds the following line of code to the control's .IDL file:

```
[id(DISPID_KEYPRESS)] void KeyPress(short* KeyAscii);
```

This line associates the KeyPress event with its standard dispatch ID and allows the container to anticipate the KeyPress event.

See Also

Reference

COleControl Class

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Methods

MFC ActiveX Controls: Adding Custom Events

Custom events differ from stock events in that they are not automatically fired by class **COleControl**. A custom event recognizes a certain action, determined by the control developer, as an event. The event map entries for custom events are represented by the **EVENT_CUSTOM** macro. The following section implements a custom event for an ActiveX control project that was created using the ActiveX Control Wizard.

Adding a Custom Event with the Add Event Wizard

The following procedure adds a specific custom event, ClickIn. You can use this procedure to add other custom events. Substitute your custom event name and its parameters for the ClickIn event name and parameters.

To add the ClickIn custom event using the Add Event Wizard

1. Load your control's project.
2. In Class View, right-click your ActiveX control class to open the shortcut menu.
3. From the shortcut menu, click **Add** and then click **Add Event**.

   This opens the Add Event Wizard.
4. In the **Event name** box, type `ClickIn`.
5. In the **Internal name** box, type the name of the event's firing function. For this example, use the default value provided by the Add Event Wizard (`FireClickIn`).
6. Add a parameter, called `xCoord` (type `OLE_XPOS_PIXELS`), using the **Parameter Name** and **Parameter Type** controls.
7. Add a second parameter, called `yCoord` (type `OLE_YPOS_PIXELS`).
8. Click **Finish** to create the event.

Add Event Wizard Changes for Custom Events

When you add a custom event, the Add Event Wizard makes changes to the control class .H, .CPP, and .IDL files. The following code samples are specific to the ClickIn event.

The following lines are added to the header (.H) file of your control class:

```
void FireClickIn(OLE_XPOS_PIXELS xCoord, OLE_YPOS_PIXELS yCoord)
    {FireEvent(eventidClickIn,EVENT_PARAM(VTS_XPOS_PIXELS  VTS_YPOS_PIXELS), xCoord, yCoord
);}
```

This code declares an inline function called `FireClickIn` that calls COleControl::FireEvent with the ClickIn event and parameters you defined using the Add Event Wizard.

In addition, the following line is added to the event map for the control, located in the implementation (.CPP) file of your control class:

```
EVENT_CUSTOM("ClickIn", FireClickIn, VTS_XPOS_PIXELS  VTS_YPOS_PIXELS)
```

This code maps the event ClickIn to the inline function `FireClickIn`, passing the parameters you defined using the Add Event Wizard.

Finally, the following line is added to your control's .IDL file:

```
[id(1)] void ClickIn(OLE_XPOS_PIXELS xCoord, OLE_YPOS_PIXELS yCoord);
```

This line assigns the ClickIn event a specific ID number, taken from the event's position in the Add Event Wizard event list. The entry in the event list allows a container to anticipate the event. For example, it might provide handler code to be executed when the event is fired.

Calling FireClickIn

Now that you have added the ClickIn custom event using the Add Event Wizard, you must decide when this event is to be fired. You do this by calling `FireClickIn` when the appropriate action occurs. For this discussion, the control uses the `InCircle` function inside a **WM_LBUTTONDOWN** message handler to fire the ClickIn event when a user clicks inside a circular or elliptical region. The following procedure adds the **WM_LBUTTONDOWN** handler.

To add a message handler with the Add Event Wizard

1. Load your control's project.
2. In Class View, select your ActiveX control class.
3. In the Properties window, click the **Messages** button.

   The Properties window displays a list of messages that can be handled by the ActiveX control. Any message shown in bold already has a handler function assigned to it.
4. From the Properties window, select the message you want to handle. For this example, select **WM_LBUTTONDOWN**.
5. From the drop-down list box on the right, select **<Add> OnLButtonDown**.
6. Double-click the new handler function in Class View to jump to the message handler code in the implementation (.CPP) file of your ActiveX control.

The following code sample calls the **InCircle** function every time the left mouse button is clicked within the control window. This sample can be found in the **WM_LBUTTONDOWN** handler function, `OnLButtonDown`, in the Circ sample abstract.

```
void CSampleCtrl::OnLButtonDown(UINT nFlags, CPoint point)
{
    if (InCircle(point))
        FireClickIn(point.x, point.y);

    COleControl::OnLButtonDown(nFlags, point);
}
```

Note

When the Add Event Wizard creates message handlers for mouse button actions, a call to the same message handler of the b ase class is automatically added. Do not remove this call. If your control uses any of the stock mouse messages, the message handlers in the base class must be called to ensure that mouse capture is handled properly.

In the following example, the event fires only when the click occurs inside a circular or elliptical region within the control. To achieve this behavior, you can place the `InCircle` function in your control's implementation (.CPP) file:

```
BOOL CSampleCtrl::InCircle(CPoint& point)
{
    CRect rc;
    GetClientRect(rc);
    // Determine radii
    double a = (rc.right - rc.left) / 2;
    double b = (rc.bottom - rc.top) / 2;

    // Determine x, y
    double x = point.x - (rc.left + rc.right) / 2;
    double y = point.y - (rc.top + rc.bottom) / 2;

    // Apply ellipse formula
    return ((x * x) / (a * a) + (y * y) / (b * b) <= 1);
}
```

You will also need to add the following declaration of the `InCircle` function to your control's header (.H) file:

```
BOOL InCircle( CPoint& point );
```

Custom Events with Stock Names

You can create custom events with the same name as stock events, however you can not implement both in the same control. For example, you might want to create a custom event called Click that does not fire when the stock event Click would normally fire. You could then fire the Click event at any time by calling its firing function.

The following procedure adds a custom Click event.

To add a custom event that uses a stock event name

1. Load your control's project.
2. In Class View, right-click your ActiveX control class to open the shortcut menu.
3. From the shortcut menu, click **Add** and then click **Add Event**.

   This opens the Add Event Wizard.
4. In the **Event Name** drop-down list, select a stock event name. For this example, select **Click**.
5. For **Event Type**, select **Custom**.
6. Click **Finish** to create the event.
7. Call `FireClick` at appropriate places in your code.

See Also

Reference

COleControl Class

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Methods

MFC ActiveX Controls: Methods

An ActiveX control fires events to communicate between itself and its control container. A container can also communicate with a control by means of methods and properties. Methods are also called functions.

Methods and properties provide an exported interface for use by other applications, such as Automation clients and ActiveX control containers. For more information on ActiveX control properties, see the article MFC ActiveX Controls: Properties.

Methods are similar in use and purpose to the member functions of a C++ class. There are two types of methods your control can implement: stock and custom. Similar to stock events, stock methods are those methods for which COleControl provides an implementation. For more information on stock methods, see the article MFC ActiveX Controls: Adding Stock Methods. Custom methods, defined by the developer, allow additional customization of the control. For more information, see the article MFC ActiveX Controls: Adding Custom Methods.

The Microsoft Foundation Class Library (MFC) implements a mechanism that allows your control to support stock and custom methods. The first part is class **COleControl**. Derived from **CWnd**, **COleControl** member functions support stock methods that are common to all ActiveX controls. The second part of this mechanism is the dispatch map. A dispatch map is similar to a message map; however, instead of mapping a function to a Windows message ID, a dispatch map maps virtual member functions to IDispatch IDS.

For a control to support various methods properly, its class must declare a dispatch map. This is accomplished by the following line of code located in control class header (.H) file:

```
DECLARE_DISPATCH_MAP()
```

The main purpose of the dispatch map is to establish the relationship between the method names used by an external caller (such as the container) and the member functions of the control's class that implement the methods. After the dispatch map has been declared, it needs to be defined in the control's implementation (.CPP) file. The following lines of code define the dispatch map:

```
BEGIN_DISPATCH_MAP(CSampleCtrl, COleControl)
   ...
END_DISPATCH_MAP()
```

If you used the MFC ActiveX Control Wizard to create the project, these lines were added automatically. If the MFC ActiveX Control Wizard was not used, you must add these lines manually.

The following articles discuss methods in detail:

- MFC ActiveX Controls: Adding Stock Methods
- MFC ActiveX Controls: Adding Custom Methods
- MFC ActiveX Controls: Returning Error Codes From a Method

See Also

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Adding Stock Methods

A stock method differs from a custom method in that it is already implemented by class COleControl. For example, **COleControl** contains a predefined member function that supports the Refresh method for your control. The dispatch map entry for this stock method is **DISP_STOCKFUNC_REFRESH**.

COleControl supports two stock methods: DoClick and Refresh. Refresh is invoked by the control's user to immediately update the control's appearance; DoClick is invoked to fire the control's Click event.

Method	Dispatch map entry	Comment
DoClick	**DISP_STOCKPROP_DOCLICK( )**	Fires a Click event.
Refresh	**DISP_STOCKPROP_REFRESH( )**	Immediately updates the control's appearance.

Adding a Stock Method Using the Add Method Wizard

Adding a stock method is simple using the Add Method Wizard. The following procedure demonstrates adding the Refresh method to a control created using the MFC ActiveX Control Wizard.

To add the stock Refresh method using the Add Method Wizard

1. Load your control's project.
2. In Class View, expand the library node of your control.
3. Right-click the interface node for your control (the second node of the library node) to open the shortcut menu.
4. From the shortcut menu, click **Add** and then click **Add Method**.

   This opens the Add Method Wizard.
5. In the **Method Name** box, click **Refresh**.
6. Click **Finish**.

Add Method Wizard Changes for Stock Methods

Because the stock Refresh method is supported by the control's base class, the **Add Method Wizard** does not change the control's class declaration in any way. It adds an entry for the method to the control's dispatch map and to its .IDL file. The following line is added to the control's dispatch map, located in its implementation (.CPP) file:

```
DISP_STOCKFUNC_REFRESH( )
```

This makes the Refresh method available to the control's users.

The following line is added to the control's .IDL file:

```
[id(DISPID_REFRESH)] void Refresh(void);
```

This line assigns the Refresh method a specific ID number.

See Also

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Adding Custom Methods

Custom methods differ from stock methods in that they are not already implemented by **COleControl**. You must supply the implementation for each custom method you add to your control.

An ActiveX control user can call a custom method at any time to perform control-specific actions. The dispatch map entry for custom methods is of the form **DISP_FUNCTION**.

Adding a Custom Method With the Add Method Wizard

The following procedure demonstrates adding the custom method PtInCircle to an ActiveX control's skeleton code. PtInCircle determines whether the coordinates passed to the control are inside or outside the circle. This same procedure can also be used to add other custom methods. Substitute your custom method name and its parameters for the PtInCircle method name and parameters.

Note

This example uses the `InCircle` function from the article Events. For more information on this function, see the article MFC ActiveX Controls: Adding Custom Events to an ActiveX Control.

To add the PtInCircle custom method using the Add Method Wizard

1. Load the control's project.
2. In Class View, expand the library node of your control.
3. Right-click the interface node for your control (the second node of the library node) to open the shortcut menu.
4. From the shortcut menu, click **Add** and then click **Add Method**.

   This opens the Add Method Wizard.
5. In the **Method Name** box, type `PtInCircle`.
6. In the **Internal Name** box, type the name of the method's internal function or use the default value (in this case, `PtInCircle`).
7. In the **Return Type** box, click **VARIANT_BOOL** for the method's return type.
8. Using the **Parameter Type** and **Parameter Name** controls, add a parameter called `xCoord` (type **OLE_XPOS_PIXELS**).
9. Using the **Parameter Type** and **Parameter Name** controls, add a parameter called `yCoord` (type **OLE_YPOS_PIXELS**).
10. Click **Finish**.

Add Method Wizard Changes for Custom Methods

When you add a custom method, the Add Method Wizard makes some changes to the control class header (.H) and implementation (.CPP) files. The following line is added to the dispatch map declaration in the control class header (.H) file:

```
VARIANT_BOOL PtInCircle(OLE_XPOS_PIXELS xCoord, OLE_YPOS_PIXELS yCoord);
```

This code declares a dispatch method handler called `PtInCircle`. This function can be called by the control user using the external name PtInCircle.

The following line is added to the control's .IDL file:

```
[id(1), helpstring("method PtInCircle")] VARIANT_BOOL
PtInCircle(OLE_XPOS_PIXELS xCoord, OLE_YPOS_PIXELS yCoord);
```

This line assigns the PtInCircle method a specific ID number, the method's position in the Add Method Wizard methods and properties list. Because the Add Method Wizard was used to add the custom method, the entry for it was added automatically to the project's .IDL file.

In addition, the following line, located in the implementation (.CPP) file of the control class, is added to the control's dispatch map:

```
DISP_FUNCTION(CSampleCtrl, "PtInCircle", PtInCircle, VARIANT_BOOL,
VTS_XPOS_PIXELS VTS_YPOS_PIXELS)
```

The **DISP_FUNCTION** macro maps the method PtInCircle to the control's handler function, `PtInCircle`, declares the return type to be **VARIANT_BOOL**, and declares two parameters of type **VTS_XPOS_PIXELS** and **VTS_YPOSPIXELS** to be passed to `PtInCircle`.

Finally, the Add Method Wizard adds the stub function `CSampleCtrl::PtInCircle` to the bottom of the control's implementation (.CPP) file. For `PtInCircle` to function as stated previously, it must be modified as follows:

```
VARIANT_BOOL CSampleCtrl::PtInCircle(OLE_XPOS_PIXELS xCoord, OLE_YPOS_PIXELS yCoord)
{
    return InCircle(CPoint(xCoord, yCoord));
}
```

See Also

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Returning Error Codes From a Method

This article describes how to return error codes from an ActiveX control method.

To indicate that an error has occurred within a method, you should use the COleControl::ThrowError member function, which takes an **SCODE** (status code) as a parameter. You can use a predefined **SCODE** or define one of your own.

Note
ThrowError is meant to be used only as a means of returning an error from within a property's Get or Set function or an automation Method. These are the only times that the appropriate exception handler will be present on the stack.

Helper functions exist for the most common predefined **SCODE**s, such as COleControl::SetNotSupported, COleControl::GetNotSupported, and COleControl::SetNotPermitted.

For a list of predefined **SCODE**s and instructions on defining custom **SCODE**s, see the section Handling Errors in Your ActiveX Control in ActiveX Controls: Advanced Topics.

For more information on reporting exceptions in other areas of your code, see COleControl::FireError and the section Handling Errors in Your ActiveX Control in ActiveX Controls: Advanced Topics.

See Also

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Properties

An ActiveX control fires events to communicate with its control container. The container, in return, uses methods and properties to communicate with the control. Methods and properties are similar in use and purpose, respectively, to member functions and member variables of a C++ class. Properties are data members of the ActiveX control that are exposed to any container. Properties provide an interface for applications that contain ActiveX controls, such as Automation clients and ActiveX control containers.

Properties are also called attributes.

For more information on ActiveX control methods, see the article MFC ActiveX Controls: Methods.

ActiveX controls can implement both stock and custom methods and properties. Class **COleControl** provides an implementation for stock properties. (For a complete list of stock properties, see the article MFC ActiveX Controls: Adding Stock Properties.) Custom properties, defined by the developer, add specialized capabilities to an ActiveX control. For more information, see MFC ActiveX Controls: Adding Custom Properties.

Both custom and stock properties, like methods, are supported by a mechanism that consists of a dispatch map that handles properties and methods and existing member functions of the **COleControl** class. In addition, these properties can have parameters that the developer uses to pass extra information to the control.

The following articles discuss ActiveX control properties in more detail:

- MFC ActiveX Controls: Adding Stock Properties
- MFC ActiveX Controls: Adding Custom Properties
- MFC ActiveX Controls: Advanced Property Implementation
- MFC ActiveX Controls: Accessing Ambient Properties

See Also

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Adding Stock Properties

Stock properties differ from custom properties in that they are already implemented by the class **COleControl**. **COleControl** contains predefined member functions that support common properties for the control. Some common properties include the control's caption and the foreground and background colors. For information on other stock properties, see Stock Properties Supported by the Add Property Wizard later in this article. The dispatch map entries for stock properties are always prefixed by **DISP_STOCKPROP**.

This article describes how to add a stock property (in this case, Caption) to an ActiveX control using the Add Property Wizard and explains the resulting code modifications. Topics include:

- Using the Add Property Wizard to add a stock property
- Add Property Wizard changes for stock properties
- Stock properties supported by the Add Property Wizard
- Stock properties and notification
- Color properties

Note
Visual Basic custom controls typically have properties such as Top, Left, Width, Height, Align, Tag, Name, TabIndex, Tab Stop, and Parent. ActiveX control containers, however, are responsible for implementing these control properties and t herefore ActiveX controls should not support these properties.

Using the Add Property Wizard to Add a Stock Property

Adding stock properties requires less code than adding custom properties because support for the property is handled automatically by **COleControl**. The following procedure demonstrates adding the stock Caption property to an ActiveX control framework and can also be used to add other stock properties. Substitute the selected stock property name for Caption.

To add the stock Caption property using the Add Property Wizard

1. Load your control's project.
2. In Class View, expand the library node of your control.
3. Right-click the interface node for your control (the second node of the library node) to open the shortcut menu.
4. From the shortcut menu, click **Add** and then click **Add Property**.

   This opens the Add Property Wizard.
5. In the **Property Name** box, click **Caption**.
6. Click **Finish**.

Add Property Wizard Changes for Stock Properties

Because **COleControl** supports stock properties, the Add Property Wizard does not change the class declaration in any way; it adds the property to the dispatch map. The Add Property Wizard adds the following line to the dispatch map of the control, which is located in the implementation (.CPP) file:

```
DISP_STOCKPROP_CAPTION( )
```

The following line is added to your control's interface description (.IDL) file:

```
[id(DISPID_CAPTION), bindable, requestedit] BSTR Caption;
```

This line assigns the Caption property a specific ID. Notice that the property is bindable and will request permission from the database before modifying the value.

This makes the Caption property available to users of your control. To use the value of a stock property, access a member variable or member function of the **COleControl** base class. For more information on these member variables and member functions, see the next section, Stock Properties Supported by the Add Property Wizard.

Stock Properties Supported by the Add Property Wizard

The **COleControl** class provides nine stock properties. You can add the properties you want by using the Add Property Wizard.

Property	Dispatch map entry	How to access value
Appearance	**DISP_STOCKPROP_APPEARANCE( )**	Value accessible as **m_sAppearance**.
BackColor	**DISP_STOCKPROP_BACKCOLOR( )**	Value accessible by calling **GetBackColor**.
BorderStyle	**DISP_STOCKPROP_BORDERSTYLE( )**	Value accessible as **m_sBorderStyle**.
Caption	**DISP_STOCKPROP_CAPTION( )**	Value accessible by calling **InternalGetText**.
Enabled	**DISP_STOCKPROP_ENABLED( )**	Value accessible as **m_bEnabled**.
Font	**DISP_STOCKPROP_FONT( )**	See the article MFC ActiveX Controls: Using Fonts for usage.
ForeColor	**DISP_STOCKPROP_FORECOLOR( )**	Value accessible by calling **GetForeColor**.
hWnd	**DISP_STOCKPROP_HWND( )**	Value accessible as **m_hWnd**.
Text	**DISP_STOCKPROP_TEXT( )**	Value accessible by calling **InternalGetText**. This property is the same as **Caption**, except for the property name.
ReadyState	**DISP_STOCKPROP_READYSTATE()**	Value accessible as m_lReadyState or **GetReadyState**

Stock Properties and Notification

Most stock properties have notification functions that can be overridden. For example, whenever the **BackColor** property is changed, the **OnBackColorChanged** function (a member function of the control class) is called. The default implementation (in **COleControl**) calls **InvalidateControl**. Override this function if you want to take additional actions in response to this situation.

Color Properties

You can use the stock **ForeColor** and **BackColor** properties, or your own custom color properties, when painting the control. To use a color property, call the COleControl::TranslateColor member function. The parameters of this function are the value of the color property and an optional palette handle. The return value is a **COLORREF** value that can be passed to GDI functions, such as **SetTextColor** and **CreateSolidBrush**.

The color values for the stock **ForeColor** and **BackColor** properties are accessed by calling either the **GetForeColor** or the **GetBackColor** function, respectively.

The following example demonstrates using these two color properties when painting a control. It initializes a temporary **COLORREF** variable and a **CBrush** object with calls to **TranslateColor**: one using the **ForeColor** property and the other using the **BackColor** property. A temporary **CBrush** object is then used to paint the control's rectangle, and the text color is set using the **ForeColor** property.

```
CBrush bkBrush(TranslateColor(GetBackColor()));
COLORREF clrFore = TranslateColor(GetForeColor());
pdc->FillRect( rcBounds, &bkbrush );
pdc->SetTextColor( clrFore );
pdc->DrawText( InternalGetText(), -1, rcBounds, DT_SINGLELINE | DT_CENTER | DT_VCENTER );
```

See Also

Reference

COleControl Class

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Properties

MFC ActiveX Controls: Methods

MFC ActiveX Controls: Adding Custom Properties

Custom properties differ from stock properties in that custom properties are not already implemented by the **COleControl** class. A custom property is used to expose a certain state or appearance of an ActiveX control to a programmer using the control.

This article describes how to add a custom property to the ActiveX control using the Add Property Wizard and explains the resulting code modifications. Topics include:

- Using the Add Property Wizard to add a custom property
- Add Property Wizard changes for custom properties

Custom properties come in four varieties of implementation: Member Variable, Member Variable with Notification, Get/Set Methods, and Parameterized.

- Member Variable Implementation

  This implementation represents the property's state as a member variable in the control class. Use the Member Variable implementation when it is not important to know when the property value changes. Of the three types, this implementation creates the least amount of support code for the property. The dispatch map entry macro for member variable implementation is DISP_PROPERTY.

- Member Variable with Notification Implementation

  This implementation consists of a member variable and a notification function created by the Add Property Wizard. The notification function is automatically called by the framework after the property value changes. Use the Member Variable with Notification implementation when you need to be notified after a property value has changed. This implementation requires more time because it requires a function call. The dispatch map entry macro for this implementation is DISP_PROPERTY_NOTIFY.

- Get/Set Methods Implementation

  This implementation consists of a pair of member functions in the control class. The Get/Set Methods implementation automatically calls the Get member function when the control's user requests the current value of the property and the Set member function when the control's user requests that the property be changed. Use this implementation when you need to compute the value of a property during run time, validate a value passed by the control's user before changing the actual property, or implement a read- or write-only property type. The dispatch map entry macro for this implementation is DISP_PROPERTY_EX. The following section, Using the Add Property Wizard to Add a Custom Property, uses the CircleOffset custom property to demonstrate this implementation.

- Parameterized Implementation

  Parameterized implementation is supported by the Add Property Wizard. A parameterized property (sometimes called a property array) can be used to access a set of values through a single property of your control. The dispatch map entry macro for this implementation is **DISP_PROPERTY_PARAM**. For more information on implementing this type, see Implementing a Parameterized Property in the article ActiveX Controls: Advanced Topics.

Using the Add Property Wizard to Add a Custom Property

The following procedure demonstrates adding a custom property, CircleOffset, which uses the Get/Set Methods implementation. The CircleOffset custom property allows the control's user to offset the circle from the center of the control's bounding rectangle. The procedure for adding custom properties with an implementation other than Get/Set Methods is very similar.

This same procedure can also be used to add other custom properties you want. Substitute your custom property name for the CircleOffset property name and parameters.

To add the CircleOffset custom property using the Add Property Wizard

1. Load your control's project.
2. In Class View, expand the library node of your control.
3. Right-click the interface node for your control (the second node of the library node) to open the shortcut menu.

4. From the shortcut menu, click **Add** and then click **Add Property**.

   This opens the Add Property Wizard.

5. In the **Property Name** box, type `CircleOffset`.

6. For **Implementation Type**, click **Get/Set Methods**.

7. In the **Property Type** box, select **short**.

8. Type unique names for your Get and Set Functions, or accept the default names.

9. Click **Finish**.

Add Property Wizard Changes for Custom Properties

When you add the CircleOffset custom property, the Add Property Wizard makes changes to the header (.H) and the implementation (.CPP) files of the control class.

The following lines are added to the .H file to declare two functions called `GetCircleOffset` and `SetCircleOffset`:

```
short GetCircleOffset(void);
void SetCircleOffset(short newVal);
```

The following line is added to your control's .IDL file:

```
[id(1)] short CircleOffset;
```

This line assigns the CircleOffset property a specific ID number, taken from the method's position in the methods and properties list of the Add Property Wizard.

In addition, the following line is added to the dispatch map (in the .CPP file of the control class) to map the CircleOffset property to the control's two handler functions:

```
DISP_PROPERTY_EX(CSampleCtrl,"CircleOffset",
   GetCircleOffset, SetCircleOffset, VT_I2)
```

Finally, the implementations of the `GetCircleOffset` and `SetCircleOffset` functions are added to the end of the control's .CPP file. In most cases, you will modify the Get function to return the value of the property. The Set function will usually contain code that should be executed either before or after the property changes.

```
void CMyCtrl::SetCircleOffset(short nNewValue)
{
   // TODO: Add your property handler here
   SetModifiedFlag();
}
```

Note that the Add Property Wizard automatically adds a call, to SetModifiedFlag, to the body of the Set function. Calling this function marks the control as modified. If a control has been modified, its new state will be saved when the container is saved. This function should be called whenever a property, saved as part of the control's persistent state, changes value.

See Also

Reference

COleControl Class

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Properties

MFC ActiveX Controls: Methods

MFC ActiveX Controls: Advanced Property Implementation

This article describes topics related to implementing advanced properties in an ActiveX control:

- Read-only and write-only properties
- Returning error codes from a property

Read-Only and Write-Only Properties

The Add Property Wizard provides a quick and easy method to implement read-only or write-only properties for the control.

To implement a read-only or write-only property

1. Load your control's project.
2. In Class View, expand the library node of your control.
3. Right-click the interface node for your control (the second node of the library node) to open the shortcut menu.
4. From the shortcut menu, click **Add** and then click **Add Property**.

   This opens the Add Property Wizard.
5. In the **Property Name** box, type the name of your property.
6. For **Implementation Type**, click **Get/Set Methods**.
7. In the **Property Type** box, select the proper type for the property.
8. If you want a read-only property, clear the Set function name. If you want a write-only property, clear the Get function name.
9. Click **Finish**.

When you do this, the Add Property Wizard inserts the function **SetNotSupported** or **GetNotSupported** in the dispatch map entry in place of a normal Set or Get function.

If you want to change an existing property to be read-only or write-only, you can edit the dispatch map manually and remove the unnecessary Set or Get function from the control class.

If you want a property to be conditionally read-only or write-only (for example, only when your control is operating in a particular mode), you can provide the Set or Get function, as normal, and call the **SetNotSupported** or **GetNotSupported** function where appropriate. For example:

```
void CSampleCtrl::SetMyProperty( short propVal )
{
    if ( m_bReadOnlyMode )   //  some control-specific state
        SetNotSupported( );
    else
        m_ipropVal = propVal;   //  set property as normal
}
```

This code sample calls **SetNotSupported** if the `m_bReadOnlyMode` data member is **TRUE**. If **FALSE**, then the property is set to the new value.

Returning Error Codes From a Property

To indicate that an error has occurred while attempting to get or set a property, use the **COleControl::ThrowError** function, which takes an **SCODE** (status code) as a parameter. You can use a predefined **SCODE** or define one of your own. For a list of predefined **SCODE**s and instructions for defining custom **SCODE**s, see Handling Errors in Your ActiveX Control in the article ActiveX controls: Advanced Topics.

Helper functions exist for the most common predefined **SCODE**s, such as COleControl::SetNotSupported, COleControl::GetNotSupported, and COleControl::SetNotPermitted.

Note

ThrowError is meant to be used only as a means of returning an error from within a property's Get or Set function or an automation method. These are the only times that the appropriate exception handler will be present on the stack.

For more information on reporting exceptions in other areas of the code, see COleControl::FireError and the section Handling Errors in Your ActiveX Control in the article ActiveX Controls: Advanced Topics.

See Also

Reference

COleControl Class

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Properties

MFC ActiveX Controls: Methods

MFC ActiveX Controls: Accessing Ambient Properties

This article discusses how an ActiveX control can access the ambient properties of its control container.

A control can obtain information about its container by accessing the container's ambient properties. These properties expose visual characteristics, such as the container's background color, the current font used by the container, and operational characteristics, such as whether the container is currently in user mode or designer mode. A control can use ambient properties to tailor its appearance and behavior to the particular container in which it is embedded. However, a control should never assume that its container will support any particular ambient property. In fact, some containers may not support any ambient properties at all. In the absence of an ambient property, a control should assume a reasonable default value.

To access an ambient property, make a call to COleControl::GetAmbientProperty. This function expects the dispatch ID for the ambient property as the first parameter (the file OLECTL.H defines dispatch IDs for the standard set of ambient properties).

The parameters of the **GetAmbientProperty** function are the dispatch ID, a variant tag indicating the expected property type, and a pointer to memory where the value should be returned. The type of data to which this pointer refers will vary depending on the variant tag. The function returns **TRUE** if the container supports the property, otherwise it returns **FALSE**.

The following code example obtains the value of the ambient property called "UserMode." If the property is not supported by the container, a default value of **TRUE** is assumed:

```
BOOL bUserMode;
if( !GetAmbientProperty( DISPID_AMBIENT_USERMODE,
   VT_BOOL, &bUserMode ) )
   bUserMode = TRUE;
```

For your convenience, **COleControl** supplies helper functions that access many of the commonly used ambient properties and return appropriate defaults when the properties are not available. These helper functions are as follows:

- COleControl::AmbientBackColor
- AmbientDisplayName
- AmbientFont

Note
Caller must call **Release( )** on the returned font.

- AmbientForeColor
- AmbientLocaleID
- AmbientScaleUnits
- AmbientTextAlign
- AmbientUserMode
- AmbientUIDead
- AmbientShowHatching
- AmbientShowGrabHandles

If the value of an ambient property changes (through some action of the container), the **OnAmbientPropertyChanged** member function of the control is called. Override this member function to handle such a notification. The parameter for **OnAmbientPropertyChanged** is the dispatch ID of the affected ambient property. The value of this dispatch ID may be **DISPID_UNKNOWN**, which indicates that one or more ambient properties has changed, but information about which properties were affected is unavailable.

See Also
Concepts
MFC ActiveX Controls

MFC ActiveX Controls: Property Pages

Property pages allow an ActiveX control user to view and change ActiveX control properties. These properties are accessed by invoking a control properties dialog box, which contains one or more property pages that provide a customized, graphical interface for viewing and editing the control properties.

ActiveX control property pages are displayed in two ways:

- When the control's Properties verb (**OLEIVERB_PROPERTIES**) is invoked, the control opens a modal property dialog box that contains the control's property pages.
- The container can display its own modeless dialog box that shows the property pages of the selected control.

The properties dialog box (illustrated in the following figure) consists of an area for displaying the current property page, tabs for switching between property pages, and a collection of buttons that perform common tasks such as closing the property page dialog, canceling any changes made, or immediately applying any changes to the ActiveX control.

Properties Dialog Box

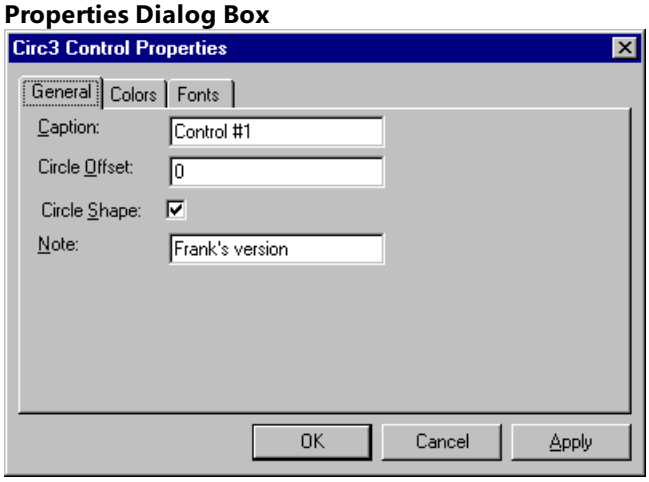

This article covers topics related to using property pages in an ActiveX control. These include:

- Implementing the default property page for an ActiveX control
- Adding controls to a property page
- Customizing the DoDataExchange function

For more information on using property pages in an ActiveX control, see the following articles:

- MFC ActiveX Controls: Adding Another Custom Property Page
- MFC ActiveX Controls: Using Stock Property Pages

For information on using property sheets in an MFC application other than an ActiveX control, see Property Sheets.

Implementing the Default Property Page

If you use the ActiveX Control Wizard to create your control project, the ActiveX Control Wizard provides a default property page class for the control derived from COlePropertyPage. Initially, this property page is blank, but you can add any dialog box control or set of controls to it. Because the ActiveX Control Wizard creates only one property page class by default, additional property page classes (also derived from **COlePropertyPage**) must be created using Class View. For more information on this procedure, see MFC ActiveX Controls: Adding Another Custom Property Page.

Implementing a property page (in this case, the default) is a three-step process:

To implement a property page

1. Add a **COlePropertyPage**-derived class to the control project. If the project was created using the ActiveX Control Wizard (as in this case), the default property page class already exists.
2. Use the dialog editor to add any controls to the property page template.

3. Customize the `DoDataExchange` function of the **COlePropertyPage**-derived class to exchange values between the property page control and the ActiveX control.

For example purposes, the following procedures use a simple control (named "Sample"). Sample was created using the ActiveX Control Wizard and contains only the stock Caption property.

Adding Controls to a Property Page

To add controls to a property page

1. With your control project open, open Resource View.
2. Double-click the **Dialog** directory icon.
3. Open the **IDD_PROPPAGE_SAMPLE** dialog box.

   The ActiveX Control Wizard appends the name of the project to the end of the dialog ID, in this case, Sample.
4. Drag and drop the selected control from the Toolbox onto the dialog box area.
5. For this example, a text label control "Caption :" and an edit box control with an **IDC_CAPTION** identifier are sufficient.
6. Click **Save** on the Toolbar to save your changes.

Now that the user interface has been modified, you need to link the edit box with the Caption property. This is done in the following section by editing the `CSamplePropPage::DoDataExchange` function.

Customizing the DoDataExchange Function

Your property page DoDataExchange function allows you to link property page values with the actual values of properties in the control. To establish links, you must map the appropriate property page fields to their respective control properties.

These mappings are implemented using the property page **DDP_** functions. The **DDP_** functions work like the **DDX_** functions used in standard MFC dialogs, with one exception. In addition to the reference to a member variable, **DDP_** functions take the name of the control property. The following is a typical entry in the `DoDataExchange` function for a property page.

```
DDP_Text(pDX, IDC_CAPTION, m_caption, _T("Caption"));
```

This function associates the property page's `m_caption` member variable with the Caption, using the **DDP_TEXT** function.

After you have the property page control inserted, you need to establish a link between the property page control, `IDC_CAPTION`, and the actual control property, Caption, using the **DDP_Text** function as described above.

DDP functions are available for other dialog control types, such as check boxes, radio buttons, and list boxes. The table below lists the entire set of property page **DDP_** functions and their purposes:

Property Page Functions

Function name	Use this function to link
DDP_CBIndex	The selected string's index in a combo box with a control property.
DDP_CBString	The selected string in a combo box with a control property. The selected string can begin with the same letters as the property's value but need not match it fully.
DDP_CBStringExact	The selected string in a combo box with a control property. The selected string and the property's string value must match exactly.
DDP_Check	A check box with a control property.
DDP_LBIndex	The selected string's index in a list box with a control property.

DDP_LBString	The selected string in a list box with a control property. The selected string can begin with the same letters as the property's value but need not match it fully.
DDP_LBStringExact	The selected string in a list box with a control property. The selected string and the property's string value must match exactly.
DDP_Radio	A radio button with a control property.
DDP_Text	Text with a control property.

See Also

Reference

COlePropertyPage Class

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Adding Another Custom Property Page

Occasionally, an ActiveX control will have more properties than can reasonably fit on one property page. In this case, you can add property pages to the ActiveX control to display these properties.

This article discusses adding new property pages to an ActiveX control that already has at least one property page. For more information on adding stock property pages (font, picture, or color), see the article MFC ActiveX Controls: Using Stock Property Pages.

The following procedures use a sample ActiveX control framework created by the ActiveX Control Wizard. Therefore, the class names and identifiers are unique to this example.

For more information on using property pages in an ActiveX control, see the following articles:

- MFC ActiveX Controls: Property Pages
- MFC ActiveX Controls: Using Stock Property Pages

Note
It is strongly recommended that new property pages adhere to the size standard for ActiveX control property pages. The stock picture and color property pages measure 250x62 dialog units (DLU). The standard font property page is 250x110 DLUs. The default property page created by the ActiveX Control Wizard uses the 250x62 DLU standard.

To insert a new property page template into your project

1. With your control project open, open Resource View in the project workspace.
2. Right-click in Resource View to open the shortcut menu and click **Add Resource**.
3. Expand the **Dialog** node, and select **IDD_OLE_PROPPAGE_SMALL**.
4. Click **New** to add the resource to your project.
5. Select the new property page template to refresh the Properties window.
6. Enter a new value for the **ID** property. This example uses **IDD_PROPPAGE_NEWPAGE**.
7. Click **Save** on the toolbar.

To associate the new template with a class

1. Open Class View.
2. Right-click in Class View to open the shortcut menu.
3. From the shortcut menu, click **Add** and then click **Add Class**.

   This opens the Add Class dialog box.
4. Double-click the **MFC Class** template.
5. In the **Class Name** box in the MFC Class Wizard, type a name for the new dialog class. (In this example, `CAddtlPropPage`.)
6. If you want to change file names, click **Change**. Type in the names for your implementation and header files, or accept the default names.
7. In the **Base Class** box, select **COlePropertyPage**.
8. In the **Dialog ID** box, select **IDD_PROPPAGE_NEWPAGE**.
9. Click **Finish** to create the class.

To allow the control's users access to this new property page, make the following changes to the control's property page IDs macro section (located in the control implementation file):

```
BEGIN_PROPPAGEIDS(CSampleCtrl, 2)
```

```
    PROPPAGEID(CMyPropPage::guid)
   PROPPAGEID(CAddtlPropPage::guid)
...
END_PROPPAGEIDS(CSampleCtrl)
```

Note that you must increase the second parameter of the **BEGIN_PROPPAGEIDS** macro (the property page count) from 1 to 2.

You must also modify the control implementation file (.CPP) file to include the header (.H) file of the new property page class.

The next step involves creating two new string resources that will provide a type name and a caption for the new property page.

To add new string resources to a property page

1. With your control project open, open Resource View.
2. Double-click the **String Table** folder and then double-click the existing string table resource to which you want to add a string.

   This opens the string table in a window.
3. Select the blank line at the end of the string table and type the text, or caption, of the string: for example, "Additional Property Page."

   This opens a **String Properties** page showing **Caption** and **ID** boxes. The **Caption** box contains the string you typed.
4. In the **ID** box, select or type an ID for the string. Press Enter when you finish.

   This example uses **IDS_SAMPLE_ADDPAGE** for the type name of the new property page.
5. Repeat steps 3 and 4 using **IDS_SAMPLE_ADDPPG_CAPTION** for the ID and "Additional Property Page" for the caption.
6. In the .CPP file of your new property page class (in this example, `CAddtlPropPage`) modify the `CAddtlPropPage::CAddtlPropPageFactory::UpdateRegistry` so that IDS_SAMPLE_ADDPAGE is passed by AfxOleRegisterPropertyPageClass, as in the following example:

   ```
   BOOL CAddtlPropPage::CAddtlPropPageFactory::UpdateRegistry(BOOL bRegister)
   {
       if (bRegister)
           return AfxOleRegisterPropertyPageClass(AfxGetInstanceHandle(),
                        m_clsid, IDS_SAMPLE_ADDPAGE);
       else
           return AfxOleUnregisterClass(m_clsid, NULL);
   }
   ```
7. Modify the constructor of `CAddtlPropPage` so that **IDS_SAMPLE_ADDPPG_CAPTION** is passed to the **COlePropertyPage** constructor, as follows:

   ```
   CAddtlPropPage::CAddtlPropPage() :
   // ****** Add your code below this line ********** //
        COlePropertyPage(IDD,  IDS_SAMPLE_ADDPPG_CAPTION)
   // ****** Add your code above this line ********** //
   {
       //{{AFX_DATA_INIT(CAddtlPropPage)
       // NOTE: Class View will add member initialization here
       //    DO NOT EDIT what you see in these blocks of generated code !
       //}}AFX_DATA_INIT
   }
   ```

After you have made the necessary modifications rebuild your project and use Test Container to test the new property page. See Testing Properties and Events with Test Container for information on how to access the test container.

See Also

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Using Stock Property Pages

This article discusses the stock property pages available for ActiveX controls and how to use them.

For more information on using property pages in an ActiveX control, see the following articles:

- MFC ActiveX Controls: Property Pages
- MFC ActiveX Controls: Adding Another Custom Property Page

MFC provides three stock property pages for use with ActiveX controls: **CLSID_CColorPropPage**, **CLSID_CFontPropPage**, and **CLSID_CPicturePropPage**. These pages display a user interface for stock color, font, and picture properties, respectively.

To incorporate these property pages into a control, add their IDs to the code that initializes the control's array of property page IDs. In the following example, this code, located in the control implementation file (.CPP), initializes the array to contain all three stock property pages and the default property page (named `CMyPropPage` in this example):

```
BEGIN_PROPPAGEIDS( CSampleCtrl, 4 )
    PROPPAGEID( CMyPropPage::guid )
    PROPPAGEID( CLSID_CFontPropPage )
    PROPPAGEID( CLSID_CColorPropPage )
    PROPPAGEID( CLSID_CPicturePropPage )
END_PROPPAGEIDS(CSampleCtrl)
```

Note that the count of property pages, in the **BEGIN_PROPPAGEIDS** macro, is 4. This represents the number of property pages supported by the ActiveX control.

After these modifications have been made, rebuild your project. Your control now has property pages for the font, picture, and color properties.

Note

If the control stock property pages cannot be accessed, it may be because the MFC DLL (MFCxx.DLL) has not been properly registered with the current operating system. This usually results from installing Visual C++ under an operating system different from the one currently running.

Tip

If your stock property pages are not visible (see previous Note), register the DLL by running RegSvr32.exe from the command line with the full path name to the DLL.

See Also

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Adding Stock Properties

MFC ActiveX Controls: Creating an Automation Server

You can develop an MFC ActiveX control as an Automation server for the purpose of programmatically embedding that control in another application and calling methods in the control from the application. Such a control would still be available to be hosted in an ActiveX control container.

To create a control as an Automation server

1. Create the control.
2. Add methods.
3. Override IsInvokeAllowed. For more information, see Knowledge Base article Q146120.
4. Build the control.

To programmatically access the methods in an Automation server

1. Create an application, for example, an MFC exe.
2. At the beginning of the **InitInstance** function, add the following line:

   ```
   AfxOleInit();
   ```

3. In Class View, right-click the project node and select **Add class from typelib** to import the type library.

   This will add files with the file name extensions .h and .cpp to the project.

4. In the header file of the class where you will call one or more methods in the ActiveX control, add the following line: `#include filename.h`, where file name is the name of the header file that was created when you imported the type library.
5. In the function where a call will be made to a method in the ActiveX control, add code that creates an object of the control's wrapper class and create the ActiveX object. For example, the following MFC code instantiates a `CtlTest` control, calls the Multiply method, and displays the result when the OK button is clicked in a dialog box:

   ```
   void CMultiply::OnOK() {
   UpdateData();   // Get the current data from the dialog box.
      _DCtlTest test;   // Create a wrapper class for the ActiveX object.
      COleException e;   // In case of errors

   // Create the ActiveX object.
   // The name is the control's progid; look it up using OleView
   if (test.CreateDispatch( "CTLTEST.CtlTestCtrl.1", &e )) {
   // call the Multiply method of your ActiveX object
   // get the result into m_product
      m_product = test.Multiply( m_a, m_b );
         UpdateData( FALSE );   // Display the string in the dialog box.
      }
      else   { // An error
         char   buf[255];
         e.GetErrorMessage( buf, sizeof( buf ) );
         AfxMessageBox( buf );   // Display the error message.
      }
   }
   ```

If you add methods to the ActiveX control after you use it in an application, you can begin using the latest version of the control in the application by deleting the files that were created when you imported the type library. Then import the type library again.

See Also

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Using Fonts

If your ActiveX control displays text, you can allow the control user to change the text appearance by changing a font property. Font properties are implemented as font objects and can be one of two types: stock or custom. Stock Font properties are preimplemented font properties that you can add using the Add Property Wizard. Custom Font properties are not preimplemented and the control developer determines the property's behavior and usage.

This article covers the following topics:

- Using the Stock Font property
- Using Custom Font Properties in Your Control

Using the Stock Font Property

Stock Font properties are preimplemented by the class COleControl. In addition, a standard Font property page is also available, allowing the user to change various attributes of the font object, such as its name, size, and style.

Access the font object through the GetFont, SetFont, and InternalGetFont functions of **COleControl**. The control user will access the font object via the `GetFont` and `SetFont` functions in the same manner as any other Get/Set property. When access to the font object is required from within a control, use the **InternalGetFont** function.

As discussed in MFC ActiveX Controls: Properties, adding stock properties is easy with the Add Property Wizard. You choose the Font property, and the Add Property Wizard automatically inserts the stock Font entry into the control's dispatch map.

To add the stock Font property using the Add Property Wizard

1. Load your control's project.
2. In Class View, expand the library node of your control.
3. Right-click the interface node for your control (the second node of the library node) to open the shortcut menu.
4. From the shortcut menu, click **Add** and then ckucj **Add Property**.

   This opens the Add Property Wizard.
5. In the **Property Name** box, click **Font**.
6. Click **Finish**.

The Add Property Wizard adds the following line to the control's dispatch map, located in the control class implementation file:

```
DISP_STOCKPROP_FONT()
```

In addition, the Add Property Wizard adds the following line to the control .IDL file:

```
[id(DISPID_FONT), bindable, requestedit] IFontDisp* Font;
```

The stock Caption property is an example of a text property that can be drawn using the stock Font property information. Adding the stock Caption property to the control uses steps similar to those used for the stock Font property.

To add the stock Caption property using the Add Property Wizard

1. Load your control's project.
2. In Class View, expand the library node of your control.
3. Right-click the interface node for your control (the second node of the library node) to open the shortcut menu.
4. From the shortcut menu, click **Add** and then click **Add Property**.

   This opens the Add Property Wizard.
5. In the **Property Name** box, click **Caption**.

6. Click **Finish**.

The Add Property Wizard adds the following line to the control's dispatch map, located in the control class implementation file:

```
DISP_STOCKPROP_CAPTION()
```

Modifying the OnDraw Function

The default implementation of `OnDraw` uses the Windows system font for all text displayed in the control. This means that you must modify the `OnDraw` code by selecting the font object into the device context. To do this, call COleControl::SelectStockFont and pass the control's device context, as shown in the following example:

```
void CSampleCtrl::OnDraw( CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
   CFont* pOldFont;
   TEXTMETRIC tm;
    const CString& strCaption = InternalGetText();

   pOldFont = SelectStockFont( pdc );
   pdc->FillRect(rcBounds, CBrush::FromHandle(
   (HBRUSH )GetStockObject(WHITE_BRUSH)));
   pdc->Ellipse(rcBounds);
   pdc->GetTextMetrics(&tm);
   pdc->SetTextAlign(TA_CENTER | TA_TOP);
   pdc->ExtTextOut((rcBounds.left + rcBounds.right) / 2,
   (rcBounds.top + rcBounds.bottom - tm.tmHeight) / 2,
   ETO_CLIPPED, rcBounds, strCaption, strCaption.GetLength(),
   NULL);

   pdc->SelectObject(pOldFont);
}
```

After the `OnDraw` function has been modified to use the font object, any text within the control is displayed with characteristics from the control's stock Font property.

Using Custom Font Properties in Your Control

In addition to the stock Font property, the ActiveX control can have custom Font properties. To add a custom font property you must:

- Use the Add Property Wizard to implement the custom Font property.
- Processing font notifications.
- Implementing a new font notification interface.

Implementing a Custom Font Property

To implement a custom Font property, you use the Add Property Wizard to add the property and then make some modifications to the code. The following sections describe how to add the custom `HeadingFont` property to the Sample control.

To add the custom Font property using the Add Property Wizard

1. Load your control's project.
2. In Class View, expand the library node of your control.
3. Right-click the interface node for your control (the second node of the library node) to open the shortcut menu.
4. From the shortcut menu, click **Add** and then click **Add Property**.

   This opens the Add Property Wizard.
5. In the **Property Name** box, type a name for the property. For this example, use **HeadingFont**.
6. For **Implementation Type**, click **Get/Set Methods**.
7. In the **Property Type** box, select **IDispatch*** for the property's type.

8. Click **Finish**.

The Add Property Wizard creates the code to add the `HeadingFont` custom property to the `CSampleCtrl` class and the SAMPLE.IDL file. Because `HeadingFont` is a Get/Set property type, the Add Property Wizard modifies the `CSampleCtrl` class's dispatch map to include a DISP_PROPERTY_EX macro entry:

```
BEGIN_DISPATCH_MAP(CSampleCtrl, COleControl)
DISP_PROPERTY_EX(CSampleCtrl, "HeadingFont", GetHeadingFont,
   SetHeadingFont, VT_DISPATCH)
END_DISPATCH_MAP()
```

The **DISP_PROPERTY_EX** macro associates the `HeadingFont` property name with its corresponding `CSampleCtrl` class Get and Set methods, `GetHeadingFont` and `SetHeadingFont`. The type of the property value is also specified; in this case, **VT_FONT**.

The Add Property Wizard also adds a declaration in the control header file (.H) for the `GetHeadingFont` and `SetHeadingFont` functions and adds their function templates in the control implementation file (.CPP):

```
LPDISPATCH CSampleCtrl::GetHeadingFont(void)
{
   // TODO: Add your property handler here
   return NULL;
}

void CSampleCtrl::SetHeadingFont(LPDISPATCH newValue)
{
   // TODO: Add your property handler here
   SetModifiedFlag();
}
```

Finally, the Add Property Wizard modifies the control .IDL file by adding an entry for the `HeadingFont` property:

```
[id(1)] IDispatch* HeadingFont;
```

Modifications to the Control Code

Now that you have added the `HeadingFont` property to the control, you must make some changes to the control header and implementation files to fully support the new property.

In the control header file (.H), add the following declaration of a protected member variable:

```
protected:

CFontHolder m_fontHeading;
```

In the control implementation file (.CPP), do the following:

- Initialize `m_fontHeading` in the control constructor.

  ```
  CSampleCtrl::CSampleCtrl( ) : m_fontHeading( &m_xFontNotification )
  {
      // [...body of constructor...]
  }
  ```

- Declare a static **FONTDESC** structure containing default attributes of the font.

  ```
  static const FONTDESC _fontdescHeading =
    { sizeof(FONTDESC), OLESTR("MS Sans Serif"), FONTSIZE( 12 ), FW_BOLD,
       ANSI_CHARSET, FALSE, FALSE, FALSE };
  ```

- In the control `DoPropExchange` member function, add a call to the **PX_Font** function. This provides initialization and persistence for your custom Font property.

```
void CSampleCtrl::DoPropExchange(CPropExchange* pPX)
{
    COleControl::DoPropExchange(pPX);

    // [...other PX_ function calls...]
    PX_Font(pPX, _T("HeadingFont"), m_fontHeading, &_fontdescHeading);
}
```

- Finish implementing the control `GetHeadingFont` member function.

```
LPFONTDISP CSampleCtrl::GetHeadingFont( )
{
    return m_fontHeading.GetFontDispatch( );
}
```

- Finish implementing the control `SetHeadingFont` member function.

```
void CSampleControl::SetHeadingFont( LPFONTDISP newValue )
{
    m_fontHeading.InitializeFont( &_fontdescHeading, newValue);
    OnFontChanged();    //notify any changes
    SetModifiedFlag( );
}
```

- Modify the control `OnDraw` member function to define a variable to hold the previously selected font.

```
CFont* pOldHeadingFont;
```

- Modify the control `OnDraw` member function to select the custom font into the device context by adding the following line wherever the font is to be used.

```
pOldHeadingFont = SelectFontObject(pdc, m_fontHeading);
```

- Modify the control `OnDraw` member function to select the previous font back into the device context by adding the following line after the font has been used.

```
pdc->SelectObject(pOldHeadingFont);
```

After the custom Font property has been implemented, the standard Font property page should be implemented, allowing control users to change the control's current font. To add the property page ID for the standard Font property page, insert the following line after the **BEGIN_PROPPAGEIDS** macro:

```
PROPPAGEID(CLSID_CFontPropPage)
```

You must also increment the count parameter of your **BEGIN_PROPPAGEIDS** macro by one. The following line illustrates this:

```
BEGIN_PROPPAGEIDS(CSampleCtrl, 2)
```

After these changes have been made, rebuild the entire project to incorporate the additional functionality.

Processing Font Notifications

In most cases the control needs to know when the characteristics of the font object have been modified. Each font object is capable of providing notifications when it changes by calling a member function of the **IFontNotification** interface, implemented by **COleControl**.

If the control uses the stock Font property, its notifications are handled by the **OnFontChanged** member function of **COleControl**. When you add custom font properties, you can have them use the same implementation. In the example in the previous section, this was accomplished by passing &**m_xFontNotification** when initializing the **m_fontHeading** member variable.

Implementing Multiple Font Object Interfaces

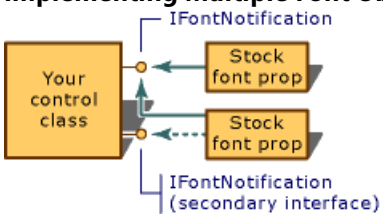

The solid lines in the figure above show that both font objects are using the same implementation of **IFontNotification**. This could cause problems if you wanted to distinguish which font changed.

One way to distinguish between the control's font object notifications is to create a separate implementation of the **IFontNotification** interface for each font object in the control. This technique allows you to optimize your drawing code by updating only the string, or strings, that use the recently modified font. The following sections demonstrate the steps necessary to implement separate notification interfaces for a second Font property. The second font property is assumed to be the `HeadingFont` property that was added in the previous section.

Implementing a New Font Notification Interface

To distinguish between the notifications of two or more fonts, a new notification interface must be implemented for each font used in the control. The following sections describe how to implement a new font notification interface by modifying the control header and implementation files.

Additions to the Header File

In the control header file (.H), add the following lines to the class declaration:

```
protected:
BEGIN_INTERFACE_PART(HeadingFontNotify, IPropertyNotifySink)
INIT_INTERFACE_PART(CSampleCtrl, HeadingFontNotify)
        STDMETHOD(OnRequestEdit)(DISPID);
        STDMETHOD(OnChanged)(DISPID);
    END_INTERFACE_PART(HeadingFontNotify)
```

This creates an implementation of the **IPropertyNotifySink** interface called `HeadingFontNotify`. This new interface contains a method called `OnChanged`.

Additions to the Implementation File

In the code that initializes the heading font (in the control constructor), change `&m_xFontNotification` to `&m_xHeadingFontNotify`. Then add the following code:

```
STDMETHODIMP_(ULONG) CSampleCtrl::XHeadingFontNotify::AddRef( )
{
    METHOD_MANAGE_STATE(CSampleCtrl, HeadingFontNotify)
    return 1;
}
STDMETHODIMP_(ULONG) CSampleCtrl::XHeadingFontNotify::Release( )
{
    METHOD_MANAGE_STATE(CSampleCtrl, HeadingFontNotify)
    return 0;
}

STDMETHODIMP CSampleCtrl::XHeadingFontNotify::QueryInterface( REFIID iid, LPVOID FAR* ppvOb
j )
{
    METHOD_MANAGE_STATE( CSampleCtrl, HeadingFontNotify )
    if( IsEqualIID( iid, IID_IUnknown ) ||
        IsEqualIID( iid, IID_IPropertyNotifySink))
    {
     *ppvObj= this;
     AddRef( );
```

```
        return NOERROR;
    }
    return ResultFromScode(E_NOINTERFACE);
}

STDMETHODIMP CSampleCtrl::XHeadingFontNotify::OnChanged(DISPID)
{
    METHOD_MANAGE_STATE( CSampleCtrl, HeadingFontNotify )
    pThis->InvalidateControl( );
    return NOERROR;
}

STDMETHODIMP CSampleCtrl::XHeadingFontNotify::OnRequestEdit(DISPID)
{
    return NOERROR;
}
```

The `AddRef` and `Release` methods in the **IPropertyNotifySink** interface keep track of the reference count for the ActiveX control object. When the control obtains access to interface pointer, the control calls `AddRef` to increment the reference count. When the control is finished with the pointer, it calls `Release`, in much the same way that **GlobalFree** might be called to free a global memory block. When the reference count for this interface goes to zero, the interface implementation can be freed. In this example, the `QueryInterface` function returns a pointer to a **IPropertyNotifySink** interface on a particular object. This function allows an ActiveX control to query an object to determine what interfaces it supports.

After these changes have been made to your project, rebuild the project and use Test Container to test the interface. See Testing Properties and Events with Test Container for information on how to access the test container.

See Also

Reference

MFC ActiveX Controls: Using Pictures in an ActiveX Control

MFC ActiveX Controls: Using Stock Property Pages

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Using Pictures in an ActiveX Control

This article describes the common Picture type and how to implement it in your ActiveX control. Topics include:

- Overview of Custom Picture Properties
- Implementing a Custom Picture Property in Your ActiveX Control
- Additions to Your Control Project

Overview of Custom Picture Properties

A Picture type is one of a group of types common to some ActiveX controls. The Picture type handles metafiles, bitmaps, or icons and allows the user to specify a picture to be displayed in an ActiveX control. Custom Picture properties are implemented using a picture object and Get/Set functions that allow the control user access to the Picture property. Control users access the custom Picture property using the stock Picture property page.

In addition to the standard Picture type, Font and Color types are also available. For more information on using the standard Font type in your ActiveX control, see the article MFC ActiveX Controls: Using Fonts.

The ActiveX control classes provide several components you can use to implement the Picture property within the control. These components include:

- The CPictureHolder class.

  This class provides easy access to the picture object and functionality for the item displayed by the custom Picture property.

- Support for properties of type **LPPICTUREDISP**, implemented with Get/Set functions.

  Using Class View you can quickly add a custom property, or properties, that supports the Picture type. For more information on adding ActiveX control properties with Class View, see the article MFC ActiveX Controls: Properties.

- A property page that manipulates a control's Picture property or properties.

  This property page is part of a group of stock property pages available to ActiveX controls. For more information on ActiveX control property pages, see the article MFC ActiveX Controls: Using Stock Property Pages

Implementing a Custom Picture Property in Your ActiveX Control

When you have completed the steps outlined in this section, the control can display pictures chosen by its user. The user can change the displayed picture using a property page that shows the current picture and has a Browse button that allows the user to the select different pictures.

A custom Picture property is implemented using a process similar to that used for implementing other properties, the main difference being that the custom property must support a Picture type. Because the item of the Picture property must be drawn by the ActiveX control, a number of additions and modifications must be made to the property before it can be fully implemented.

To implement a custom Picture property, you must do the following:

- Add code to your control project.

  A standard Picture property page ID, a data member of type **CPictureHolder**, and a custom property of type **LPPICTUREDISP** with a Get/Set implementation must be added.

- Modify several functions in your control class.

  These modifications will be made to several functions that are responsible for the drawing of your ActiveX control.

Additions to Your Control Project

To add the property page ID for the standard Picture property page, insert the following line after the **BEGIN_PROPPAGEIDS** macro in the control implementation file (.CPP):

```
PROPPAGEID(CLSID_CPicturePropPage)
```

You must also increment the count parameter of your **BEGIN_PROPPAGEIDS** macro by one. The following line illustrates this:

```
BEGIN_PROPPAGEIDS(CSampleCtrl, 2)
```

To add the **CPictureHolder** data member to the control class, insert the following line under the protected section of the control class declaration in the control header file (.H):

```
CPictureHolder    m_pic;
```

It is not necessary to name your data member `m_pic`; any name will suffice.

Next, add a custom property that supports a Picture type:

To add a custom picture property using the Add Property Wizard

1. Load your control's project.
2. In Class View, expand the library node of your control.
3. Right-click the interface node for your control (the second node of the library node) to open the shortcut menu.
4. From the shortcut menu, choose **Add** and then **Add Property**.
5. In the **Property Name** box, type the property name. For example purposes, `ControlPicture` is used in this procedure.
6. In the **Property Type** box, select **LPPICTUREDISP** for the property type.
7. For **Implementation Type**, click **Get/Set Methods**.
8. Type unique names for your Get and Set Functions or accept the default names. (In this example, the default names `GetControlPicture` and `SetControlPicture` are used.)
9. Click **Finish**.

The Add Property Wizard adds the following code between the dispatch map comments in the control header (.H) file:

```
LPPICTUREDISP GetControlPicture(void);
void SetControlPicture(LPPICTUREDISP newVal);
```

In addition, the following code was inserted in the dispatch map of the control implementation (.CPP) file:

```
DISP_PROPERTY_EX(CSampleCtrl, "ControlPicture", GetControlPicture, SetControlPicture, VT_PI
CTURE)
```

The Add Property Wizard also adds the following two stub functions in the control implementation file:

```
LPPICTUREDISP CSampleCtrl::GetControlPicture(void)
{
    // TODO: Add your property handler here

    return NULL;
}

void CSampleCtrl::SetControlPicture(LPPICTUREDISP newVal)
{
    // TODO: Add your property handler here

    SetModifiedFlag();
}
```

Note

Your control class and function names might differ from the example above.

Modifications to Your Control Project

After you have made the necessary additions to your control project, you need to modify several functions that affect the rendering of your ActiveX control. These functions, `OnResetState`, `OnDraw`, and the Get/Set functions of a custom Picture property, are located in the control implementation file. (Note that in this example the control class is called `CSampleCtrl`, the **CPictureHolder** data member is called `m_pic`, and the custom picture property name is `ControlPicture`.)

In the control `OnResetState` function, add the following optional line after the call to **COleControl::OnResetState**:

```
m_pic.CreateEmpty();
```

This sets the control's picture to a blank picture.

To draw the picture properly, make a call to CPictureHolder::Render in the control `OnDraw` function. Modify your function to resemble the following example:

```
void CSampleCtrl::OnDraw(
   CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
// ****** Add your code below this line ********** //
   m_pic.Render(pdc, rcBounds, rcBounds);
}
```

In the Get function of the control's custom picture property, add the following line:

```
return m_pic.GetPictureDispatch();
```

In the Set function of the control's custom Picture property, add the following lines:

```
m_pic.SetPictureDispatch(newValue);
InvalidateControl();
```

The picture property must be made persistent so that information added at design time will show up at run time. Add the following line to the **COleControl**-derived class's **DoPropExchange** function:

```
PX_Picture(pPX, "ControlPicture",m_pic);
```

Note
Your class and function names might differ from the example above.

After you complete the modifications, rebuild your project to incorporate the new functionality of the custom Picture property and use Test Container to test the new property. See Testing Properties and Events with Test Container for information on how to access the test container.

See Also

Concepts

MFC ActiveX Controls
MFC ActiveX Controls: Using Fonts
MFC ActiveX Controls: Property Pages

MFC ActiveX Controls: Advanced Topics

This article covers advanced topics related to developing ActiveX controls. These include:

- Using Database Classes in ActiveX Controls
- Implementing a Parameterized Property
- Handling Errors in Your ActiveX Control
- Handling Special Keys in the Control
- Accessing Dialog Controls That Are Invisible at Run Time

Using Database Classes in ActiveX Controls

Because the ActiveX control classes are part of the class library, you can apply the same procedures and rules for using database classes in a standard MFC application to developing ActiveX controls that use the MFC database classes.

For a general overview of the MFC database classes, see MFC Database Classes (DAO and ODBC). The article introduces both the MFC ODBC classes and the MFC DAO classes and directs you to more details on either.

Note
As of Visual C++ .NET, the Visual C++ environment and wizards no longer support DAO (although the DAO classes are included and you can still use them). Microsoft recommends that you use OLE DB Templates or ODBC and MFC for new projects. You should only use DAO in maintaining existing applications.

Implementing a Parameterized Property

A parameterized property (sometimes called a property array) is a method for exposing a homogeneous collection of values as a single property of the control. For example, you can use a parameterized property to expose an array or a dictionary as a property. In Visual Basic, such a property is accessed using array notation:

```
x = obj.ArrayProp(2, 3)     ' gets element of 2D array
obj.ArrayProp(2, 3) = 7     ' sets element of 2D array
```

Use the Add Property Wizard to implement a parameterized property. The Add Property Wizard implements the property by adding a pair of Get/Set functions that allow the control user to access the property using the above notation or in the standard fashion.

Similar to methods and properties, parameterized properties also have a limit to the number of parameters allowed. In the case of parameterized properties, the limit is 15 parameters (with one parameter reserved for storing the property value).

The following procedure adds a parameterized property, called Array, which can be accessed as a two-dimensional array of integers.

To add a parameterized property using the Add Property Wizard

1. Load your control's project.
2. In Class View, expand the library node of your control.
3. Right-click the interface node for your control (the second node of the library node) to open the shortcut menu.
4. From the shortcut menu, click **Add** and then click **Add Property**.
5. In the **Property Name** box, type `Array`.
6. In the **Property Type** box, select **short**.
7. For **Implementation** Type, click **Get/Set Methods**.
8. In the **Get Function** and **Set Function** boxes, type unique names for your Get and Set Functions or accept the default names.

9. Add a parameter, called `row` (type `short`), using the **Parameter Name** and **Parameter Type** controls.
10. Add a second parameter called `column` (type `short`).
11. Click **Finish**.

Changes Made by the Add Property Wizard

When you add a custom property, the Add Property Wizard makes changes to the control class header (.H) and the implementation (.CPP) files.

The following lines are added to the control class .H file:

```
short GetArray(short row, short column);
void SetArray(short row, short column, short newVal);
```

This code declares two functions called `GetArray` and `SetArray` that allow the user to request a specific row and column when accessing the property.

In addition, the Add Property Wizard adds the following lines to the control dispatch map, located in the control class implementation (.CPP) file:

```
DISP_PROPERTY_PARAM(CSampleCtrl, "Array", GetArray, SetArray, VT_I2,
       VTS_I2 VTS_I2)
```

Finally, the implementations of the `GetArray` and `SetArray` functions are added to the end of the .CPP file. In most cases, you will modify the Get function to return the value of the property. The Set function will usually contain code that should execute, either before or after the property changes.

For this property to be useful, you could declare a two-dimensional array member variable in the control class, of type **short**, to store values for the parameterized property. You could then modify the Get function to return the value stored at the proper row and column, as indicated by the parameters, and modify the Set function to update the value referenced by the row and column parameters.

Handling Errors in Your ActiveX Control

If error conditions occur in the control, you may need to report the error to the control container. There are two methods for reporting errors, depending on the situation in which the error occurs. If the error occurs within a property's Get or Set function, or within the implementation of an OLE Automation method, the control should call COleControl::ThrowError, which signals to the control user that an error has occurred. If the error occurs at any other time, the control should call COleControl::FireError, which fires a stock Error event.

To indicate the kind of error that has occurred, the control must pass an error code to **ThrowError** or **FireError**. An error code is an OLE status code, which has a 32-bit value. When possible, choose an error code from the standard set of codes defined in the OLECTL.H header file. The following table summarizes these codes.

ActiveX Control Error Codes

Error	Description
CTL_E_ILLEGALFUNCTIONCALL	Illegal function call
CTL_E_OVERFLOW	Overflow
CTL_E_OUTOFMEMORY	Out of memory
CTL_E_DIVISIONBYZERO	Division by zero
CTL_E_OUTOFSTRINGSPACE	Out of string space
CTL_E_OUTOFSTACKSPACE	Out of stack space
CTL_E_BADFILENAMEORNUMBER	Bad file name or number

CTL_E_FILENOTFOUND	File not found
CTL_E_BADFILEMODE	Bad file mode
CTL_E_FILEALREADYOPEN	File already open
CTL_E_DEVICEIOERROR	Device I/O error
CTL_E_FILEALREADYEXISTS	File already exists
CTL_E_BADRECORDLENGTH	Bad record length
CTL_E_DISKFULL	Disk full
CTL_E_BADRECORDNUMBER	Bad record number
CTL_E_BADFILENAME	Bad file name
CTL_E_TOOMANYFILES	Too many files
CTL_E_DEVICEUNAVAILABLE	Device unavailable
CTL_E_PERMISSIONDENIED	Permission denied
CTL_E_DISKNOTREADY	Disk not ready
CTL_E_PATHFILEACCESSERROR	Path/file access error
CTL_E_PATHNOTFOUND	Path not found
CTL_E_INVALIDPATTERNSTRING	Invalid pattern string
CTL_E_INVALIDUSEOFNULL	Invalid use of NULL
CTL_E_INVALIDFILEFORMAT	Invalid file format
CTL_E_INVALIDPROPERTYVALUE	Invalid property value
CTL_E_INVALIDPROPERTYARRAYINDEX	Invalid property array index
CTL_E_SETNOTSUPPORTEDATRUNTIME	Set not supported at run time
CTL_E_SETNOTSUPPORTED	Set not supported (read-only property)
CTL_E_NEEDPROPERTYARRAYINDEX	Need property array index
CTL_E_SETNOTPERMITTED	Set not permitted
CTL_E_GETNOTSUPPORTEDATRUNTIME	Get not supported at run time
CTL_E_GETNOTSUPPORTED	Get not supported (write-only property)
CTL_E_PROPERTYNOTFOUND	Property not found

CTL_E_INVALIDCLIPBOARDFORMAT	Invalid clipboard format
CTL_E_INVALIDPICTURE	Invalid picture
CTL_E_PRINTERERROR	Printer error
CTL_E_CANTSAVEFILETOTEMP	Can't save file to TEMP
CTL_E_SEARCHTEXTNOTFOUND	Search text not found
CTL_E_REPLACEMENTSTOOLONG	Replacements too long

If necessary, use the **CUSTOM_CTL_SCODE** macro to define a custom error code for a condition that is not covered by one of the standard codes. The parameter for this macro should be an integer between 1000 and 32767, inclusive. For example:

```
#define MYCTL_E_SPECIALERROR CUSTOM_CTL_SCODE(1000)
```

If you are creating an ActiveX control to replace an existing VBX control, define your ActiveX control error codes with the same numeric values the VBX control uses to ensure that the error codes are compatible.

Handling Special Keys in the Control

In some cases you may want to handle certain keystroke combinations in a special way; for example, insert a new line when the ENTER key is pressed in a multiline text box control or move between a group of edit controls when a directional key ID pressed.

If the base class of your ActiveX control is **COleControl**, you can override CWnd::PreTranslateMessage to handle messages before the container processes them. When using this technique, always return **TRUE** if you handle the message in your override of **PreTranslateMessage**.

The following code example demonstrates a possible way of handling any messages related to the directional keys.

```
BOOL CSampleControl::PreTranslateMessage(LPMSG lpmsg)
{
    BOOL bHandleNow = FALSE;

    switch (lpmsg->message)
    {
    case WM_KEYDOWN:
        switch (lpmsg->wParam)
        {
        case VK_UP:
        case VK_DOWN:
        case VK_LEFT:
        case VK_RIGHT:
            bHandleNow = TRUE;
            break;
        }
        if (bHandleNow)
            OnKeyDown(lpmsg->wParam, LOWORD(lpmsg
                ->lParam), HIWORD(lpmsg->lParam));
        break;
    }
    return bHandleNow;
}
```

For more information on handling keyboard interfaces for an ActiveX control, see the ActiveX SDK documentation.

Accessing Dialog Controls that Are Invisible at Run Time

You can create dialog controls that have no user interface and are invisible at run time. If you add an invisible at run time ActiveX control to a dialog box and use CWnd::GetDlgItem to access the control, the control will not work correctly. Instead, you

should use one of the following techniques to obtain an object that represents the control:

- Using the Add Member Variable Wizard, select **Control Variable** and then select the control's ID. Enter a member variable name and select the control's wrapper class as the **Control Type**.

  -or-

- Declare a local variable and subclass as the dialog item. Insert code that resembles the following (`CMyCtrl` is the wrapper class, `IDC_MYCTRL1` is the control's ID):

```
CMyCtrl myCtrl;
myCtrl.SubclassDlgItem(IDC_MYCTRL1, this);
// ... use myCtrl ...
myCtrl.UnsubclassWindow();
```

See Also

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Distributing ActiveX Controls

This article discusses several issues related to redistributing ActiveX controls:

- ANSI or Unicode Control Versions
- Installing ActiveX Controls and Redistributable DLLs
- Registering Controls

Note
For additional information on redistributing ActiveX controls, see Redistributing Controls.

ANSI or Unicode Control Versions

You must decide whether to ship an ANSI or Unicode version of the control, or both. This decision is based on portability factors inherent in ANSI and Unicode character sets.

ANSI controls, which work on all Win32 operating systems, allow for maximum portability between the various Win32 operating systems. Unicode controls work on only Windows NT (version 3.51 or later), but not on Windows 95 or Windows 98. If portability is your primary concern, ship ANSI controls. If your controls will run only on Windows NT, you can ship Unicode controls. You could also choose to ship both and have your application install the version most appropriate for the user's operating system.

Installing ActiveX Controls and Redistributable DLLs

The setup program you provide with your ActiveX controls should create a special subdirectory of the Windows directory and install the controls' .OCX files in it.

Note
Use the Windows **GetWindowsDirectory** API in your setup program to obtain the name of the Windows directory. You may want to derive the subdirectory name from the name of your company or product.

The setup program must install the necessary redistributable DLL files in the Windows system directory. If any of the DLLs are already present on the user's machine, the setup program should compare their versions with the versions you are installing. Reinstall a file only if its version number is higher than the file already installed.

Because ActiveX controls can be used only in OLE container applications, there is no need to distribute the full set of OLE DLLs with your controls. You can assume that the containing application (or the operating system itself) has the standard OLE DLLs installed.

Registering Controls

Before a control can be used, appropriate entries must be created for it in the Windows registration database. Some ActiveX control containers provide a menu item for users to register new controls, but this feature may not be available in all containers. Therefore, you may want your setup program to register the controls when they are installed.

If you prefer, you can write your setup program to register the control directly instead.

Use the **LoadLibrary** Windows API to load the control DLL. Next, use **GetProcAddress** to obtain the address of the "DllRegisterServer" function. Finally, call the **DllRegisterServer** function. The following code sample demonstrates one possible method, where `hLib` stores the handle of the control library, and `lpDllEntryPoint` stores the address of the "DllRegisterServer" function.

```
HINSTANCE hLib = LoadLibrary(pszDllName);

if (hLib < (HINSTANCE)HINSTANCE_ERROR)
{
   DisplayMessage(IDS_LOADLIBFAILED, pszDllName); //unable to load DLL
   iReturn = FAIL_LOAD;                           //unable to load DLL
}
```

```
// Find the entry point.
(FARPROC&)lpDllEntryPoint = GetProcAddress(hLib,
   _T("DllRegisterServer"));
if (lpDllEntryPoint != NULL)
   (*lpDllEntryPoint)();
else
   // Unable to locate entry point
```

The advantage of registering the control directly is that you do not need to invoke and load a separate process (namely, REGSVR32), reducing installation time. In addition, because registration is an internal process, the setup program can handle errors and unforeseen situations better than an external process can.

Note
Before your setup program installs an ActiveX control, it should call **OleInitialize**. When your setup program is finished, call **OleUnitialize**. This ensures that the OLE system DLLs are in the proper state for registering an ActiveX control.

You should register MFCx0.DLL.

See Also
Concepts
MFC ActiveX Controls

MFC ActiveX Controls: Licensing an ActiveX Control

Licensing support, an optional feature of ActiveX controls, allows you to control who is able to use or distribute the control. (For additional discussion of licensing issues, see Licensing Issues in Upgrading an Existing ActiveX Control.)

This article discusses the following topics:

- Overview of ActiveX Control Licensing
- Creating a Licensed Control
- Licensing Support
- Customizing the Licensing of an ActiveX Control

ActiveX controls that implement licensing allow you, as the control developer, to determine how other people will use the ActiveX control. You provide the control purchaser with the control and .LIC file, with the agreement that the purchaser may distribute the control, but not the .LIC file, with an application that uses the control. This prevents users of that application from writing new applications that use the control, without first licensing the control from you.

Overview of ActiveX Control Licensing

To provide licensing support for ActiveX controls, the COleObjectFactory class provides an implementation for several functions in the **IClassFactory2** interface: **IClassFactory2::RequestLicKey**, **IClassFactory2::GetLicInfo**, and **IClassFactory2::CreateInstanceLic**. When the container application developer makes a request to create an instance of the control, a call to **GetLicInfo** is made to verify that the control .LIC file is present. If the control is licensed, an instance of the control can be created and placed in the container. After the developer has finished constructing the container application, another function call, this time to **RequestLicKey**, is made. This function returns a license key (a simple character string) to the container application. The returned key is then embedded in the application.

The figure below demonstrates the license verification of an ActiveX control that will be used during the development of a container application. As mentioned previously, the container application developer must have the proper .LIC file installed on the development machine to create an instance of the control.

Verification of a Licensed ActiveX Control During Development

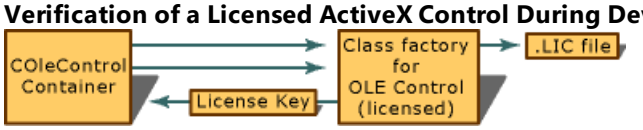

The next process, shown in the following figure, occurs when the end user runs the container application.

When the application is started, an instance of the control usually needs to be created. The container accomplishes this by making a call to **CreateInstanceLic**, passing the embedded license key as a parameter. A string comparison is then made between the embedded license key and the control's own copy of the license key. If the match is successful, an instance of the control is created and the application continues to execute normally. Note that the .LIC file need not be present on the control user's machine.

Verification of a Licensed ActiveX Control During Execution

Control licensing consists of two basic components: specific code in the control implementation DLL and the license file. The code is composed of two (or possibly three) function calls and a character string, hereafter referred to as a "license string", containing a copyright notice. These calls and the license string are found in the control implementation (.CPP) file. The license file, generated by the ActiveX Control Wizard, is a text file with a copyright statement. It is named using the project name with an .LIC extension, for example SAMPLE.LIC. A licensed control must be accompanied by the license file if design-time use is needed.

Creating a Licensed Control

When you use the ActiveX Control Wizard to create the control framework, it is easy to include licensing support. When you specify that the control should have a run-time license, the ActiveX Control Wizard adds code to the control class to support licensing. The code consists of functions that use a key and license file for license verification. These functions also can be modified to customize the control licensing. For more information on license customization, see Customizing the Licensing of an ActiveX Control later in this article.

To add support for licensing with the ActiveX Control Wizard when you create your control project

- Use the instructions in Creating an MFC ActiveX Control. The **Application Settings** page of the ActiveX Control Wizard contains the option to create the control with the run-time license.

The ActiveX Control Wizard now generates an ActiveX control framework that includes basic licensing support. For a detailed explanation of the licensing code, see the next topic.

Licensing Support

When you use the ActiveX Control Wizard to add licensing support to an ActiveX control, the ActiveX Control Wizard adds code that declares and implements the licensing capability is added to the control header and implementation files. This code is composed of a **VerifyUserLicense** member function and a **GetLicenseKey** member function, which override the default implementations found in COleObjectFactory . These functions retrieve and verify the control license.

Note
A third member function, **VerifyLicenseKey** is not generated by the ActiveX Control Wizard, but can be overridden to custo mize the license key verification behavior.

These member functions are:

- VerifyUserLicense

  Verifies that the control allows design-time usage by checking the system for the presence of the control license file. This function is called by the framework as part of processing **IClassFactory2::GetLicInfo** and **IClassFactory::CreateInstanceLic**.

- GetLicenseKey

  Requests a unique key from the control DLL. This key is embedded in the container application and used later, in conjunction with **VerifyLicenseKey**, to create an instance of the control. This function is called by the framework as part of processing **IClassFactory2::RequestLicKey**.

- VerifyLicenseKey

  Verifies that the embedded key and the control's unique key are the same. This allows the container to create an instance of the control for its use. This function is called by the framework as part of processing **IClassFactory2::CreateInstanceLic** and can be overridden to provide customized verification of the license key. The default implementation performs a string comparison. For more information, see Customizing the Licensing of an ActiveX Control, later in this article.

Header File Modifications

The ActiveX Control Wizard places the following code in the control header file. In this example, two member functions of `CSampleCtrl`'s object `factory` are declared, one that verifies the presence of the control .LIC file and another that retrieves the license key to be used in the application containing the control:

```
BEGIN_OLEFACTORY(CSampleCtrl)        // Class factory and guid
    virtual BOOL VerifyUserLicense();
    virtual BOOL GetLicenseKey(DWORD, BSTR FAR*);
END_OLEFACTORY(CSampleCtrl)
```

Implementation File Modifications

The ActiveX Control Wizard places the following two statements in the control implementation file to declare the license filename and license string:

```
static const TCHAR BASED_CODE _szLicFileName[] =
   _T("License.lic");

static const WCHAR BASED_CODE _szLicString[] =
   L"Copyright (c) 2000 ";
```

Note

If you modify **szLicString** in any way, you must also modify the first line in the control .LIC file or licensing will not function properly.

The ActiveX Control Wizard places the following code in the control implementation file to define the control class' `VerifyUserLicense` and `GetLicenseKey` functions:

```
/////////////////////////////////////////////////////////////////////////////
// CLicenseCtrl::CLicenseCtrlFactory::VerifyUserLicense
// Checks for existence of a user license

BOOL CLicenseCtrl::CLicenseCtrlFactory::VerifyUserLicense()
{
   return AfxVerifyLicFile(AfxGetInstanceHandle(),
_szLicFileName, _szLicString);
}

/////////////////////////////////////////////////////////////////////////////
// CLicenseCtrl::CLicenseCtrlFactory::GetLicenseKey -
// Returns a runtime licensing key

BOOL CLicenseCtrl::CLicenseCtrlFactory::GetLicenseKey(DWORD dwReserved,
   BSTR FAR* pbstrKey)
{
   if (pbstrKey == NULL)
      return FALSE;

   *pbstrKey = SysAllocString(_szLicString);
   return (*pbstrKey != NULL);
}
```

Finally, the **ActiveX Control Wizard** modifies the control project .IDL file. The **licensed** keyword is added to the coclass declaration of the control, as in the following example:

```
[ uuid(EF365BF1-5D4C-11D2-875A-00600893AFE8), licensed,
  helpstring("Sample Control"), control ]
coclass Sample2
```

Customizing the Licensing of an ActiveX Control

Because **VerifyUserLicense**, **GetLicenseKey**, and **VerifyLicenseKey** are declared as virtual member functions of the control factory class, you can customize the control's licensing behavior.

For example, you can provide several levels of licensing for the control by overriding the **VerifyUserLicense** or **VerifyLicenseKey** member functions. Inside this function you could adjust which properties or methods are exposed to the user according to the license level you detected.

You can also add code to the **VerifyLicenseKey** function that provides a customized method for informing the user that control creation has failed. For instance, in your **VerifyLicenseKey** member function you could display a message box stating that the control failed to initialize and why.

Note

Another way to customize ActiveX control license verification is to check the registration database for a specific registry key, instead of calling **AfxVerifyLicFile**. For an example of the default implementation, see the Implementation File Modifications section of this article.

For additional discussion of licensing issues, see Licensing Issues in Upgrading an Existing ActiveX Control.

See Also

Reference

MFC ActiveX Control Wizard

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Localizing an ActiveX Control

This article discusses procedures for localizing ActiveX control interfaces.

If you want to adapt an ActiveX control to an international market, you may want to localize the control. Windows supports many languages in addition to the default English, including German, French, and Swedish. This can present problems for the control if its interface is in English only.

In general, ActiveX controls should always base their locale on the ambient LocaleID property. There are three ways to do this:

- Load resources, always on demand, based on the current value of the ambient LocaleID property. The MFC ActiveX controls sample LOCALIZE uses this strategy.
- Load resources when the first control is instanced, based on the ambient LocaleID property, and use these resources for all other instances. This article demonstrates this strategy.

Note
This will not work correctly in some cases, if future instances have different locales.

- Use the **OnAmbientChanged** notification function to dynamically load the proper resources for the container's locale.

Note
This will work for the control, but the run-time DLL will not dynamically update its own resources when the ambient LocaleID property changes. In addition, run-time DLLs for ActiveX controls use the thread locale to determine the locale for its resources.

The rest of this article describes two localizing strategies. The first strategy localizes the control's programmability interface (names of properties, methods, and events). The second strategy localizes the control's user interface, using the container's ambient LocaleID property. For a demonstration of control localization, see the MFC ActiveX controls sample LOCALIZE.

Localizing the Control's Programmability Interface

When localizing the control's programmability interface (the interface used by programmers writing applications that use your control), you must create a modified version of the control .IDL file (a script for building the control type library) for each language you intend to support. This is the only place you need to localize the control property names.

When you develop a localized control, include the locale ID as an attribute at the type library level. For example, if you want to provide a type library with French localized property names, make a copy of your SAMPLE.IDL file, and call it SAMPLEFR.IDL. Add a locale ID attribute to the file (the locale ID for French is 0x040c), similar to the following:

```
[ uuid(xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx), version(1.0), lcid(0x040c) ]
library Sample
{
```

Change the property names in SAMPLEFR.IDL to their French equivalents, and then use MKTYPLIB.EXE to produce the French type library, SAMPLEFR.TLB.

To create multiple localized type libraries you can add any localized .IDL files to the project and they will be built automatically.

To add an .IDL file to your ActiveX control project

1. With your control project open, on the **Project** menu, click **Add Existing Item**.

   The **Add Existing Item** dialog box appears.

2. If necessary, select the drive and directory to view.
3. In the **Files of Type** box, select **All Files (*.*)**.
4. In the file list box, double-click the .IDL file you want to insert into the project.

5. Click **Open** when you have added all necessary .IDL files.

Because the files have been added to the project, they will be built when the rest of the project is built. The localized type libraries are located in the current ActiveX control project directory.

Within your code, the internal property names (usually in English) are always used and are never localized. This includes the control dispatch map, the property exchange functions, and your property page data exchange code.

Only one type library (.TLB) file may be bound into the resources of the control implementation (.OCX) file. This is usually the version with the standardized (typically, English) names. To ship a localized version of your control you need to ship the .OCX (which has already been bound to the default .TLB version) and the .TLB for the appropriate locale. This means that only the .OCX is needed for English versions, since the correct .TLB has already been bound to it. For other locales, the localized type library also must be shipped with the .OCX.

To ensure that clients of your control can find the localized type library, register your locale-specific .TLB file(s) under the TypeLib section of the Windows system registry. The third parameter (normally optional) of the AfxOleRegisterTypeLib function is provided for this purpose. The following example registers a French type library for an ActiveX control:

```
STDAPI DllRegisterServer(void)
{
    AFX_MANAGE_STATE(_afxModuleAddrThis);

    if (!AfxOleRegisterTypeLib(AfxGetInstanceHandle(), _tlid))
        return ResultFromScode(SELFREG_E_TYPELIB);
    AfxOleRegisterTypeLib(AfxGetInstanceHandle(), _tlid,
        _T("samplefr.tlb"))
    if (!COleObjectFactoryEx::UpdateRegistryAll(TRUE))
        return ResultFromScode(SELFREG_E_CLASS);

    return NOERROR;
}
```

When your control is registered, the **AfxOleRegisterTypeLib** function automatically looks for the specified .TLB file in the same directory as the control and registers it in the Windows registration database. If the .TLB file is not found, the function has no effect.

Localizing the Control's User Interface

To localize a control's user interface, place all of the control's user-visible resources (such as property pages and error messages) into language-specific resource DLLs. You then can use the container's ambient LocaleID property to select the appropriate DLL for the user's locale.

The following code example demonstrates one approach to locate and load the resource DLL for a specific locale. This member function, called `GetLocalizedResourceHandle` for this example, can be a member function of your ActiveX control class:

```
HINSTANCE CSampleCtrl::GetLocalizedResourceHandle(LCID lcid)
{
    LPCTSTR lpszResDll;
    HINSTANCE hResHandle = NULL;
    switch (PRIMARYLANGID(lang))
    {
    case LANG_ENGLISH:
        lpszResDll = "myctlen.dll";
        break;

    case LANG_FRENCH:
        lpszResDll = "myctlfr.dll";
        break;

    case LANG_GERMAN:
        lpszResDll = "myctlde.dll";
        break;

    case 0:
    default:
        lpszResDll = NULL;
    }
```

```
    if (lpszResDll != NULL)
        hResHandle = LoadLibrary(lpszResDll);
    #ifndef _WIN32
        if(hResHandle <= HINSTANCE_ERROR)
           hResHandle = NULL;
    #endif

    return hResHandle;
}
```

Note that the sublanguage ID could be checked in each case of the switch statement, to provide more specialized localization (for example, local dialects of German). For a demonstration of this function, see the `GetResourceHandle` function in the MFC ActiveX controls sample LOCALIZE.

When the control first loads itself into a container, it can call COleControl::AmbientLocaleID to retrieve the locale ID. The control can then pass the returned locale ID value to the `GetLocalizedResourceHandle` function, which loads the proper resource library. The control should pass the resulting handle, if any, to AfxSetResourceHandle:

```
m_hResDll = GetLocalizedResourceHandle( AmbientLocaleID() );
if (m_hResDll != NULL)
    AfxSetResourceHandle(m_hResDll);
```

Place the code sample above into a member function of the control, such as an override of COleControl::OnSetClientSite. In addition, `m_hResDLL` should be a member variable of the control class.

You can use similar logic for localizing a control's property page. To localize the property page, add code similar to the following sample to your property page's implementation file (in an override of COlePropertyPage::OnSetPageSite):

```
LPPROPERTYPAGESITE pSite;
LCID lcid = 0;
if((pSite = GetPageSite()) != NULL)
    pSite->GetLocaleID(&lcid);
HINSTANCE hResource = GetLocalizedResourceHandle(lcid);
HINSTANCE hResourceSave = NULL;

if (hResource != NULL)
{
    hResourceSave = AfxGetResourceHandle();
    AfxSetResourceHandle(hResource);
}

//  Load dialog template and caption string.
COlePropertyPage::OnSetPageSite( );

if (hResource != NULL)
    AfxSetResourceHandle(hResourceSave);
```

See Also

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Serializing

This article discusses how to serialize an ActiveX control. Serialization is the process of reading from or writing to a persistent storage medium, such as a disk file. The Microsoft Foundation Class (MFC) Library provides built-in support for serialization in class **CObject**. **COleControl** extends this support to ActiveX controls through the use of a property exchange mechanism.

Serialization for ActiveX controls is implemented by overriding COleControl::DoPropExchange. This function, called during the loading and saving of the control object, stores all properties implemented with a member variable or a member variable with change notification.

The following topics cover the main issues related to serializing an ActiveX control:

- Implementing `DoPropExchange` function to serialize your control object
- Customizing the Serialization Process
- Implementing Version Support

Implementing the DoPropExchange Function

When you use the ActiveX Control Wizard to generate the control project, several default handler functions are automatically added to the control class, including the default implementation of COleControl::DoPropExchange. The following example shows the code added to classes created with the ActiveX Control Wizard:

```
void CSampleCtrl::DoPropExchange( CPropExchange* pPX)
{
   ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
   COleControl::DoPropExchange(pPX);

   // TODO: Call PX_ functions for each persistent custom property.
}
```

If you want to make a property persistent, modify `DoPropExchange` by adding a call to the property exchange function. The following example demonstrates the serialization of a custom Boolean CircleShape property, where the CircleShape property has a default value of **TRUE**:

```
void CSampleCtrl::DoPropExchange(CPropExchange* pPX)
{
   ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
   COleControl::DoPropExchange(pPX);

   PX_Bool(pPX, "CircleShape", m_bCircleShape, TRUE);
}
```

The following table lists the possible property exchange functions you can use to serialize the control's properties:

Property exchange functions	Purpose
PX_Blob( )	Serializes a type Binary Large Object (BLOB) data property.
PX_Bool( )	Serializes a type Boolean property.
PX_Color( )	Serializes a type color property.
PX_Currency( )	Serializes a type **CY** (currency) property.
PX_Double( )	Serializes a type **double** property.
PX_Font( )	Serializes a Font type property.

PX_Float( )	Serializes a type **float** property.
PX_IUnknown( )	Serializes a property of type **LPUNKNOWN**.
PX_Long( )	Serializes a type **long** property.
PX_Picture( )	Serializes a type Picture property.
PX_Short( )	Serializes a type **short** property.
PX_String( )	Serializes a type **CString** property.
PX_ULong( )	Serializes a type **ULONG** property.
PX_UShort( )	Serializes a type **USHORT** property.

For more information on these property exchange functions, see Persistence of OLE Controls in the *MFC Reference*.

Customizing the Default Behavior of DoPropExchange

The default implementation of **DoPropertyExchange** (as shown in the previous topic) makes a call to base class **COleControl**. This serializes the set of properties automatically supported by **COleControl**, which uses more storage space than serializing only the custom properties of the control. Removing this call allows your object to serialize only those properties you consider important. Any stock property states the control has implemented will not be serialized when saving or loading the control object unless you explicitly add **PX_** calls for them.

Implementing Version Support

Version support enables a revised ActiveX control to add new persistent properties, and still be able to detect and load the persistent state created by an earlier version of the control. To make a control's version available as part of its persistent data, call COleControl::ExchangeVersion in the control's `DoPropExchange` function. This call is automatically inserted if the ActiveX control was created using the ActiveX Control Wizard. It can be removed if version support is not needed. However, the cost in control size is very small (4 bytes) for the added flexibility that version support provides.

If the control was not created with the ActiveX Control Wizard, add a call to **COleControl::ExchangeVersion** by inserting the following line at the beginning of your `DoPropExchange` function (before the call to **COleControl::DoPropExchange**):

```
void CSampleCtrl::DoPropExchange(CPropExchange* pPX)
{
   ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
   COleControl::DoPropExchange(pPX);
...
}
```

You can use any **DWORD** as the version number. Projects generated by the ActiveX Control Wizard use **_wVerMinor** and **_wVerMajor** as the default. These are global constants defined in the implementation file of the project's ActiveX control class. Within the remainder of your `DoPropExchange` function, you can call CPropExchange::GetVersion at any time to retrieve the version you are saving or retrieving.

In the following example, version 1 of this sample control has only a "ReleaseDate" property. Version 2 adds an "OriginalDate" property. If the control is instructed to load the persistent state from the old version, it initializes the member variable for the new property to a default value.

```
void CSampleCtrl::DoPropExchange(CPropExchange* pPX)
{
    ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
    COleControl::DoPropExchange(pPX);
    PX_Long(pPX, "ReleaseDate", m_releaseDate);
    if (pPX->GetVersion() >= MAKELONG(0, 2))
    {
        PX_Long(pPX, "OriginalDate", m_originalDate);
```

```
    }
    else
    {
        if (pPX->IsLoading())
            m_originalDate = 0;
    }
}
```

By default, a control "converts" old data to the latest format. For example, if version 2 of a control loads data that was saved by version 1, it will write the version 2 format when it is saved again. If you want the control to save data in the format last read, pass **FALSE** as a third parameter when calling **ExchangeVersion**. This third parameter is optional and is **TRUE** by default.

See Also

Concepts

MFC ActiveX Controls

MFC ActiveX Controls: Subclassing a Windows Control

This article describes the process for subclassing a common Windows control to create an ActiveX control. Subclassing an existing Windows control is a quick way to develop an ActiveX control. The new control will have the abilities of the subclassed Windows control, such as painting and responding to mouse clicks. The MFC ActiveX controls sample BUTTON is an example of subclassing a Windows control.

To subclass a Windows control, complete the following tasks:

- Override the IsSubclassedControl and PreCreateWindow member functions of COleControl
- Modify the OnDraw member function
- Handle any ActiveX control messages (OCM) reflected to the control

> **Note**
>
> Much of this work is done for you by the ActiveX Control Wizard if you select control to be subclassed using the **Select Parent Window Class** drop-down list on the **Control Settings** page.

See Knowledge Base article Q243454 for more information on subclassing a control.

Overriding IsSubclassedControl and PreCreateWindow

To override **PreCreateWindow** and **IsSubclassedControl**, add the following lines of code to the **protected** section of the control class declaration:

```
BOOL PreCreateWindow( CREATESTRUCT& cs );
BOOL IsSubclassedControl( );
```

In the control implementation file (.CPP), add the following lines of code to implement the two overridden functions:

```
BOOL CSampleCtrl::PreCreateWindow( CREATESTRUCT& cs )
{
    cs.lpszClass = _T("BUTTON");
    return COleControl::PreCreateWindow(cs);
}
BOOL CSampleCtrl::IsSubclassedControl( )
{
    return TRUE;
}
```

Notice that, in this example, the Windows button control is specified in **PreCreateWindow**. However, any standard Windows controls can be subclassed. For more information on standard Windows controls, see Controls.

When subclassing a Windows control, you may want to specify particular window style (**WS_**) or extended window style (**WS_EX_**) flags to be used in creating the control's window. You can set values for these parameters in the **PreCreateWindow** member function by modifying the **cs.style** and the **cs.dwExStyle** structure fields. Modifications to these fields should be made using an **OR** operation, to preserve the default flags that are set by class **COleControl**. For example, if the control is subclassing the BUTTON control and you want the control to appear as a check box, insert the following line of code into the implementation of `CSampleCtrl::PreCreateWindow`, before the return statement:

```
cs.style |= BS_CHECKBOX;
```

This operation adds the **BS_CHECKBOX** style flag, while leaving the default style flag (**WS_CHILD**) of class **COleControl** intact.

Modifying the OnDraw Member Function

If you want your subclassed control to keep the same appearance as the corresponding Windows control, the `OnDraw` member function for the control should contain only a call to the **DoSuperclassPaint** member function, as in the following example:

```
void CSampleCtrl::OnDraw( CDC* pdc, const CRect& rcBounds,
    const CRect& rcInvalid )
{
    DoSuperclassPaint( pdc, rcBounds );
}
```

The **DoSuperclassPaint** member function, implemented by **COleControl**, uses the window procedure of the Windows control to draw the control in the specified device context, within the bounding rectangle. This makes the control visible even when it is not active.

Note

The **DoSuperclassPaint** member function will work only with those control types that allow a device context to be passed as the **wParam** of a **WM_PAINT** message. This includes some of the standard Windows controls, such as **SCROLLBAR** and **BUTTON**, and all the common controls. For controls that do not support this behavior, you will have to provide your own code to properly display an inactive control.

Handling Reflected Window Messages

Windows controls typically send certain window messages to their parent window. Some of these messages, such as **WM_COMMAND**, provide notification of an action by the user. Others, such as **WM_CTLCOLOR**, are used to obtain information from the parent window. An ActiveX control usually communicates with the parent window by other means. Notifications are communicated by firing events (sending event notifications), and information about the control container is obtained by accessing the container's ambient properties. Because these communication techniques exist, ActiveX control containers are not expected to process any window messages sent by the control.

To prevent the container from receiving the window messages sent by a subclassed Windows control, **COleControl** creates an extra window to serve as the control's parent. This extra window, called a "reflector," is created only for an ActiveX control that subclasses a Windows control and has the same size and position as the control window. The reflector window intercepts certain window messages and sends them back to the control. The control, in its window procedure, can then process these reflected messages by taking actions appropriate for an ActiveX control (for example, firing an event). See Reflected Window Message IDs for a list of intercepted windows messages and their corresponding reflected messages.

An ActiveX control container may be designed to perform message reflection itself, eliminating the need for **COleControl** to create the reflector window and reducing the run-time overhead for a subclassed Windows control. **COleControl** detects whether the container supports this capability by checking for a MessageReflect ambient property with a value of **TRUE**.

To handle a reflected window message, add an entry to the control message map and implement a handler function. Because reflected messages are not part of the standard set of messages defined by Windows, Class View does not support adding such message handlers. However, it is not difficult to add a handler manually.

To add a message handler for a reflected window message manually do the following:

- In the control class .H file, declare a handler function. The function should have a return type of **LRESULT** and two parameters, with types **WPARAM** and **LPARAM**, respectively. For example:

  ```
  class CSampleCtrl : public COleControl
  {
  protected:
     LRESULT OnOcmCommand( WPARAM wParam, LPARAM lParam );
  ...
  }
  ```

- In the control class .CPP file, add an **ON_MESSAGE** entry to the message map. The parameters of this entry should be the message identifier and the name of the handler function. For example:

  ```
  BEGIN_MESSAGE_MAP(CSampleCtrl, COleControl)
     ...
     ON_MESSAGE(OCM_COMMAND, OnOcmCommand)
     ...
  END_MESSAGE_MAP()
  ```

- Also in the .CPP file, implement the **OnOcmCommand** member function to process the reflected message. The **wParam** and **lParam** parameters are the same as those of the original window message.

For an example of how reflected messages are processed, refer to the MFC ActiveX controls sample BUTTON. It demonstrates an **OnOcmCommand** handler that detects the **BN_CLICKED** notification code and responds by firing (sending) a Click event.

See Also

Concepts

MFC ActiveX Controls

Reflected Window Message IDs

A quick way to create an ActiveX control, or other specialized control, is to subclass a window. For more information, see MFC ActiveX Controls: Subclassing a Windows Control.

To prevent the control's container from receiving the window messages sent by a subclassed Windows control, COleControl creates a "reflector" window to intercept certain window messages and send them back to the control. The control, in its window procedure, can then process these reflected messages by taking actions appropriate for an ActiveX control.

The following table shows the messages that are intercepted and the corresponding messages that the reflector window sends.

Message sent by the control	Message reflected to the control
WM_COMMAND	**OCM_COMMAND**
WM_CTLCOLORBTN	**OCM_CTLCOLORBTN**
WM_CTLCOLOREDIT	**OCM_CTLCOLOREDIT**
WM_CTLCOLORDLG	**OCM_CTLCOLORDLG**
WM_CTLCOLORLISTBOX	**OCM_CTLCOLORLISTBOX**
WM_CTLCOLORSCROLLBAR	**OCM_CTLCOLORSCROLLBAR**
WM_CTLCOLORSTATIC	**OCM_CTLCOLORSTATIC**
WM_DRAWITEM	**OCM_DRAWITEM**
WM_MEASUREITEM	**OCM_MEASUREITEM**
WM_DELETEITEM	**OCM_DELETEITEM**
WM_VKEYTOITEM	**OCM_VKEYTOITEM**
WM_CHARTOITEM	**OCM_CHARTOITEM**
WM_COMPAREITEM	**OCM_COMPAREITEM**
WM_HSCROLL	**OCM_HSCROLL**
WM_VSCROLL	**OCM_VSCROLL**
WM_PARENTNOTIFY	**OCM_PARENTNOTIFY**
WM_NOTIFY	**OCM_NOTIFY**

Note

If the control runs on a Win32 system, there are several types of **WM_CTLCOLOR*** messages it may receive. For more information, see **WM_CTLCOLORBTN**, **WM_CTLCOLORDLG**, **WM_CTLCOLOREDIT**, **WM_CTLCOLORLISTBOX**, **WM_CTLCOLORMSGBOX**, **WM_CTLCOLORSCROLLBAR**, **WM_CTLCOLORSTATIC**.

See Also

Reference

TN062: Message Reflection for Windows Controls

Concepts

MFC ActiveX Controls: Subclassing a Windows Control

MFC ActiveX Controls: Using Data Binding in an ActiveX Control

One of the more powerful uses of ActiveX controls is data binding, which allows a property of the control to bind with a specific field in a database. When a user modifies data in this bound property, the control notifies the database and requests that the record field be updated. The database then notifies the control of the success or failure of the request.

This article covers the control side of your task. Implementing the data binding interactions with the database is the responsibility of the control container. How you manage the database interactions in your container is beyond the scope of this documentation. How you prepare the control for data binding is explained in the rest of this article.

Conceptual Diagram of a Data-Bound Control

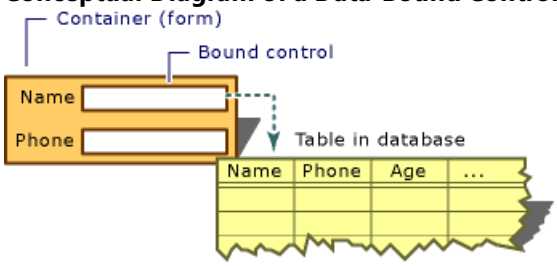

The **COleControl** class provides two member functions that make data binding an easy process to implement. The first function, BoundPropertyRequestEdit, is used to request permission to change the property value. BoundPropertyChanged, the second function, is called after the property value has been successfully changed.

This article covers the following topics:

- Creating a Bindable Stock Property
- Creating a Bindable Get/Set Method

Creating a Bindable Stock Property

It is possible to create a data-bound stock property, although it is more likely that you will want a bindable get/set method.

Note
Stock properties have the **bindable** and **requestedit** attributes by default.

To add a bindable stock property using the Add Property Wizard

1. Begin a project using the MFC ActiveX Control Wizard.
2. Right-click the interface node for your control.

   This opens the shortcut menu.
3. From the shortcut menu, click **Add** and then click **Add Property**.
4. Select one of the entries from the **Property Name** drop-down list. For example, you can select **Text**.

   Because **Text** is a stock property, the **bindable** and **requestedit** attributes are already checked.
5. Select the following check boxes from the **IDL Attributes** tab: **displaybind** and **defaultbind** to add the attributes to the property definition in the project's .IDL file. These attributes make the control visible to the user and make the stock property the default bindable property.

At this point, your control can display data from a data source, but the user will not be able to update data fields. If you want your control to also be able to update data, change the `OnOcmCommand` OnOcmCommand function to look as follows:

```
#ifdef _WIN32
   WORD wNotifyCode = HIWORD(wParam);
#else
   WORD wNotifyCode = HIWORD(lParam);
```

```
#endif

   if(wNotifyCode==EN_CHANGE)
   {
      if(!BoundPropertyRequestEdit(DISPID_TEXT))
         SetNotSupported();
      else
      {
         GetText();
         // Notify container of change
         BoundPropertyChanged(DISPID_TEXT);
      }

   }

   return 0;
```

You can now build the project, which will register the control. When you insert the control in a dialog box, the **Data Field** and **Data Source** properties will have been added and you can now select a data source and field to display in the control.

Creating a Bindable Get/Set Method

In addition to a data-bound get/set method, you can also create a bindable stock property.

Note
This procedure assumes you have an ActiveX control project that subclasses a Windows control.

To add a bindable get/set method using the Add Property Wizard

1. Load your control's project.
2. On the **Control Settings** page, select a window class for the control to subclass. For example, you may want to subclass an EDIT control.
3. Load your control's project.
4. Right-click the interface node for your control.

   This opens the shortcut menu.
5. From the shortcut menu, click **Add** and then click **Add Property**.
6. Type the property name in the **Property Name** box. Use `MyProp` for this example.
7. Select a data type from the **Property Type** drop-down list box. Use **short** for this example.
8. For **Implementation Type**, click **Get/Set Methods**.
9. Click **Finish**.
10. Select the following check boxes from the IDL Attributes tab: **bindable**, **requestedit**, **displaybind**, and **defaultbind** to add the attributes to the property definition in the project's .IDL file. These attributes make the control visible to the user and make the stock property the default bindable property.
11. Modify the body of the `SetMyProp` function so that it contains the following code:

```
if(!BoundPropertyRequestEdit(1))
   {
   SetNotSupported();
   return;
   }
else
   {
   if(AmbientUserMode()) // SendMessage only at run-time
      {
```

```
      sprintf(m_strText.GetBuffer(10), "%d", nNewValue);
      SetWindowText(m_strText);
      m_strText.ReleaseBuffer();
      }
   else
      InvalidateControl();
   // Signal a property change
   // This is the MFC equivalent of OnChanged()
   BoundPropertyChanged(1);
   SetModifiedFlag();
   }
```

12. The parameter passed to the `BoundPropertyChanged` and `BoundPropertyRequestEdit` functions is the dispid of the property, which is the parameter passed to the id() attribute for the property in the .IDL file.

13. Modify the OnOcmCommand function so it contains the following code:

```
#ifdef _WIN32
   WORD wNotifyCode = HIWORD(wParam);
#else
   WORD wNotifyCode = HIWORD(lParam);
#endif
   if(wNotifyCode==EN_CHANGE)
   {
      if(!BoundPropertyRequestEdit(1))
         SetNotSupported();
      else
      {
         GetMyProp();
         // Notify container of change
         BoundPropertyChanged(1);
      }
   }
   return 0;
```

14. Modify the `OnDraw` function so that it contains the following code:

```
if(!AmbientUserMode())
   {
      // Draw the Text property at design-time
      pdc->FillRect(rcBounds,
         CBrush::FromHandle((HBRUSH)GetStockObject(WHITE_BRUSH)));
      pdc->DrawText(m_strText, -1,&(RECT) rcBounds,
         DT_LEFT | DT_TOP | DT_SINGLELINE);
   }
   else
   DoSuperclassPaint(pdc, rcBounds);
```

15. To the public section of the header file the header file for your control class, add the following definitions (constructors) for member variables:

```
CString m_strText;
short m_nMyNum;
```

16. Make the following line the last line in the `DoPropExchange` function:

```
PX_Short(pPX, _T("MyProp"), m_strText);
```

17. Modify the `OnResetState` function so that it contains the following code:

```
COleControl::OnResetState(); // Resets defaults found in DoPropExchange
m_strText=AmbientDisplayName();
```

18. Modify the `GetMyProp` function so that it contains the following code:

```
if(AmbientUserMode())
   {
   GetWindowText(m_strText);
   m_nMyNum = atoi(m_strText);
   }
return m_nMyNum;
```

You can now build the project, which will register the control. When you insert the control in a dialog box, the **Data Field** and **Data Source** properties will have been added and you can now select a data source and field to display in the control.

See Also

Reference

Data-Bound Controls (ADO and RDO)

Concepts

MFC ActiveX Controls

ActiveX Control Containers

An ActiveX control container is a container that fully supports ActiveX controls and can incorporate them into its own windows or dialogs. An ActiveX control is a reusable software element that you can use in many development projects. Controls allow your application's user to access databases, monitor data, and make various selections within your applications. For more information on ActiveX controls, see the article MFC ActiveX Controls.

Control containers typically take two forms in a project:

- Dialogs and dialog-like windows such as form views, where an ActiveX control is used somewhere in the dialog box.
- Windows in an application, where an ActiveX control is used in a toolbar, or other location in the user window.

The ActiveX control container interacts with the control via exposed methods and properties. These methods and properties, which can be accessed and modified by the control container, are accessed through a wrapper class in the ActiveX control container project. The embedded ActiveX control can also interact with the container by firing (sending) events to notify the container that an action has occurred. The control container can choose to act upon these notifications or not.

Additional articles discuss several topics, from creating an ActiveX control container project to basic implementation issues related to ActiveX control containers built with Visual C++:

- Creating an MFC ActiveX Control Container
- Containers for ActiveX Controls
- ActiveX Control Containers: Manually Enabling ActiveX Control Containment
- ActiveX Control Containers: Inserting a Control into a Control Container Application
- ActiveX Control Containers: Connecting an ActiveX Control to a Member Variable
- ActiveX Control Containers: Handling Events from an ActiveX control
- ActiveX Control Containers: Viewing and Modifying Control Properties
- ActiveX Control Containers: Programming ActiveX Controls in an ActiveX Control Container
- ActiveX Control Containers: Using Controls in a Non-Dialog Container

For more information about using ActiveX controls in a dialog box, see the Dialog Editor topics.

For a list of articles that explain the details of developing ActiveX controls using Visual C++ and the MFC ActiveX control classes, see MFC ActiveX controls. The articles are grouped by functional categories.

See Also

Concepts

MFC ActiveX Controls

Containers for ActiveX Controls

You can use ActiveX controls developed in Visual C++ in other applications, as long as they support ActiveX control containment. A number of Microsoft applications, beginning with the versions listed, support ActiveX control containment.

Note
The following list is not a complete list of applications that support ActiveX controls but represents the set used most often in testing:

Applications that support ActiveX control containment include:

- Internet Explorer 3.x and greater
- Visual Basic 4.x and greater
- Visual C++ 4.x and greater
- Access 95 and greater
- Excel 97 and greater
- Word 97 and greater
- Access 97 and greater
- FrontPage 97 and greater
- PowerPoint 97 and greater
- Visual InterDev 97 and greater

The following are non-Microsoft applications that support ActiveX control containment:

- PowerBuilder
- Delphi
- C++ Builder
- NCompass Plug-in for Netscape Navigator

See Also

Concepts

ActiveX Control Containers

ActiveX Control Containers: Manually Enabling ActiveX Control Containment

If you did not enable ActiveX control support when you used the MFC Application Wizard to generate your application, you will have to add this support manually. This article describes the process for manually adding ActiveX control containment to an existing OLE container application. If you know in advance that you want ActiveX control support in your OLE container, see the article Creating an MFC ActiveX Control Container.

Note
This article uses a dialog-based ActiveX control container project named Container and an embedded control named Circ as examples in the procedures and code.

To support ActiveX controls, you must add one line of code to two of your project's files.

- Modify your main dialog's `InitInstance` function (found in CONTAINER.CPP) by the MFC Application Wizard making a call to AfxEnableControlContainer, as in the following example:

```
// CContainerApp initialization

BOOL CContainerApp::InitInstance()
{
   AfxEnableControlContainer();
...
}
```

- Add the following to your project's STDAFX.H header file:

```
#include <Afxdisp.h>
```

After you have completed these steps, rebuild your project by clicking **Build** on the **Build** menu.

See Also
Concepts
ActiveX Control Containers

ActiveX Control Containers: Inserting a Control into a Control Container Application

Before you can access an ActiveX control from an ActiveX control container application, you must add the ActiveX control to the container application using the Insert ActiveX Control dialog box.

To add an ActiveX control to the ActiveX control container project, see Viewing and Adding ActiveX Controls to a Dialog Box.

Once you add the control, you need to add a member variable (of the ActiveX control type) to the dialog box class. For more information on this procedure, see Adding a Member Variable.

Once you have added the member variable a class, referred to as a wrapper class, is automatically generated and added to your project. This class is used as an interface between the control container and the embedded control.

See Also

Concepts

ActiveX Control Containers

ActiveX Control Containers: Connecting an ActiveX Control to a Member Variable

The easiest way to access an ActiveX control from within its control container application is to associate the ActiveX control with a member variable of the dialog class that will contain the control.

Note
This is not the only way to access an embedded control from within a container class, but for the purposes of this article it is sufficient.

Adding a member variable to the dialog class

1. From Class View, right-click the main dialog class to open the shortcut menu. For example, `CContainerDlg`.
2. From the shortcut menu, click **Add** and then **Add Variable**.
3. In the Add Member Variable Wizard, click **Control variable**.
4. In the **Control ID** list box, select the control ID of the embedded ActiveX control. For example, `IDC_CIRCCTRL1`.
5. In the **Variable Name** box, enter a name.

   For example, `m_circctl`.
6. Click **Finish** to accept your choices and exit the Add Member Variable Wizard.

See Also

Concepts

ActiveX Control Containers

ActiveX Control Containers: Handling Events from an ActiveX Control

This article discusses using the Properties window to install event handlers for ActiveX controls in an ActiveX control container. The event handlers are used to receive notifications (from the control) of certain events and perform some action in response. This notification is called "firing" the event.

Note
This article uses a dialog-based ActiveX control container project named Container and an embedded control named Circ as examples in the procedures and code.

Using the Events button in the Properties window, you can create a map of events that can occur in your ActiveX control container application. This map, called an "event sink map,'' is created and maintained by Visual C++ when you add event handlers to the control container class. Each event handler, implemented with an event map entry, maps a specific event to a container event handler member function. This event handler function is called when the specified event is fired by the ActiveX control object.

For more information on event sink maps, see Event Sink Maps in the *Class Library Reference*.

Event Handler Modifications to the Project

When you use the Properties window to add event handlers, an event sink map is declared and defined in your project. The following statements are added to the control .CPP file the first time an event handler is added. This code declares an event sink map for the dialog box class (in this case, `CContainerDlg`):

```
BEGIN_EVENTSINK_MAP(CContainerDlg, CDialog)
    //{{AFX_EVENTSINK_MAP(CContainerDlg)
    //}}AFX_EVENTSINK_MAP
END_EVENTSINK_MAP()
```

As you use the Properties window to add events, an event map entry (**ON_EVENT**) is added to the event sink map and an event handler function is added to the container's implementation (.CPP) file.

The following example declares an event handler, called `OnClickInCircCtrl`, for the Circ control's **ClickIn** event:

```
BEGIN_EVENTSINK_MAP(CContainerDlg, CDialog)
    //{{AFX_EVENTSINK_MAP(CContainerDlg)
   ON_EVENT(CContainerDlg, IDC_CIRCCTRL1, 1 /*
ClickIn */, OnClickInCircctrl, VTS_I4 VTS_I4)
   //}}AFX_EVENTSINK_MAP
END_EVENTSINK_MAP()
```

In addition, the following template is added to the `CContainerDlg` class implementation (.CPP) file for the event handler member function:

```
BOOL CContainerDlg::OnClickInCircctrl1(OLE_XPOS_PIXELS nX, OLE_YPOS_PIXELS nY)
{
   // use nX and nY here
   return TRUE;
}
```

For more information on event sink macros, see Event Sink Maps in the *Class Library Reference*.

To create an event handler function

1. From Class View, select the dialog class that contains the ActiveX control. For this example, use `CContainerDlg`.
2. In the Properties window, click the **Events** button.

3. In the Properties window, select the control ID of the embedded ActiveX control. For this example, use `IDC_CIRCCTRL1`.

   The Properties window displays a list of events that can be fired by the embedded ActiveX control. Any member function shown in bold already has handler functions assigned to it.

4. Select the event you want the dialog class to handle. For this example, select **Click**.

5. From the drop-down list box on the right, select **<Add> ClickCircctrl1**.

6. Double-click the new handler function from Class View to jump to the event handler code in the implementation (.CPP) file of `CContainerDlg`.

See Also

Concepts

ActiveX Control Containers

ActiveX Control Containers: Viewing and Modifying Control Properties

When you insert an ActiveX control into a project, it is useful to view and change the properties supported by the ActiveX control. This article discusses how to use the Visual C++ resource editor to do this.

If your ActiveX control container application uses embedded controls, you can view and modify the control's properties while in the resource editor. You can also use the resource editor to set property values during design time. The resource editor then automatically saves these values in the project's resource file. Any instance of the control will then have its properties initialized to these values.

This procedure assumes that you have inserted a control into your project. For information, see ActiveX Control Containers: Inserting a Control Into a Control Container Application.

The first step in viewing the control's properties is to add an instance of the control to the project's dialog template.

To view the properties of a control

1. In Resource View, open the **Dialog** folder.
2. Open your main dialog box template.
3. Insert an ActiveX control using the **Insert ActiveX Control** dialog box. For more information, see Viewing and Adding ActiveX Controls to a Dialog Box.
4. Select the ActiveX control in the dialog box.
5. From the Properties window, click the **Properties** button.

Use the **Properties** dialog box to modify and test new properties immediately.

See Also

Concepts

ActiveX Control Containers

ActiveX Control Containers: Programming ActiveX Controls in an ActiveX Control Container

This article describes the process for accessing the exposed methods and properties of embedded ActiveX controls. Basically, you will follow these steps:

1. Insert an ActiveX control into the ActiveX container project using Gallery.
2. Define a member variable (or other form of access) of the same type as the ActiveX control wrapper class.
3. Program the ActiveX control using predefined member functions of the wrapper class.

For this discussion, assume you've created a dialog-based project (named Container) with ActiveX control support. The Circ sample control, Circ, will be added to the resulting project.

Once the Circ control is inserted into the project (step 1), insert an instance of the Circ control into the application's main dialog box.

To add the Circ control to the dialog template

1. Load the ActiveX control container project. For this example, use the `Container` project.
2. Click the Resource View tab.
3. Open the **Dialog** folder.
4. Double-click the main dialog box template. For this example, use **IDD_CONTAINER_DIALOG**.
5. Click the Circ control icon on the Toolbox.
6. Click a spot within the dialog box to insert the Circ control.
7. From the **File** menu, choose **Save All** to save all modifications to the dialog box template.

Modifications to the Project

To enable the Container application to access the Circ control, Visual C++ automatically adds the wrapper class (`CCirc`) implementation file (.CPP) to the Container project and the wrapper class header (.H) file to the dialog box implementation file:

```
//{{AFX_INCLUDES(CContainerDlg)
#include "circ.h"
//}}AFX_INCLUDES
// ContainerDlg.cpp : implementation file
//
```

The Wrapper Class Header (.H) File

To get and set properties (and invoke methods) for the Circ control, the `CCirc` wrapper class provides a declaration of all exposed methods and properties. In the example, these declarations are found in CIRC.H. The following sample is the portion of class `CCirc` that defines the exposed interfaces of the ActiveX control:

```
class CCirc : public CWnd
{
...
// Attributes
public:
   OLE_COLOR GetBackColor();
   void SetBackColor(OLE_COLOR);
   BOOL GetCircleShape();
   void SetCircleShape(BOOL);
   short GetCircleOffset();
   void SetCircleOffset(short);
   unsigned long GetFlashColor();
   void SetFlashColor(unsigned long);
```

```
    BSTR GetCaption();
    void SetCaption(LPCTSTR);
    LPFONTDISP GetFont();
    void SetFont(LPFONTDISP);
    OLE_COLOR GetForeColor();
    void SetForeColor(OLE_COLOR);
    CString GetNote();
    void SetNote(LPCTSTR);

// Operations
public:
    void AboutBox();
};
```

These functions can then be called from other of the application's procedures using normal C++ syntax. For more information on using this member function set to access the control's methods and properties, see the section Programming the ActiveX control.

Member Variable Modifications to the Project

Once the ActiveX control has been added to the project and embedded in a dialog box container, it can be accessed by other parts of the project. The easiest way to access the control is to create a member variable of the dialog class, `CContainerDlg` (step 2), that is of the same type as the wrapper class added to the project by Visual C++. You can then use the member variable to access the embedded control at any time.

When the **Add Member Variable** dialog box adds the `m_circctl` member variable to the project, it also adds the following lines to the header file (.H) of the `CContainerDlg` class:

```
class CContainerDlg : public CDialog
{
// Construction
public:
    CContainerDlg(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
    //{{AFX_DATA(CContainerDlg)
    enum { IDD = IDD_CONTAINER_DIALOG };
    CCirc   m_circctl;
    //}}AFX_DATA
};
```

In addition, a call to **DDX_Control** is automatically added to the `CContainerDlg`'s implementation of `DoDataExchange`:

```
DDX_Control(pDX, IDC_CIRCCTRL1, m_circctl);
```

Programming the ActiveX Control

At this point, you have inserted the ActiveX control into your dialog template and created a member variable for it. You can now use common C++ syntax to access the properties and methods of the embedded control.

As noted (in The Wrapper Class Header (.H) File), the header file (.H) for the `CCirc` wrapper class, in this case CIRC.H, contains a list of member functions that you can use to get and set any exposed property value. Member functions for exposed methods are also available.

A common place to modify the control's properties is in the `OnInitDialog` member function of the main dialog class. This function is called just before the dialog box appears and is used to initialize its contents, including any of its controls.

The following code example uses the `m_circctl` member variable to modify the Caption and CircleShape properties of the embedded Circ control:

```
BOOL CContainerDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Add "About..." menu item to system menu.
```

```
    // IDM_ABOUTBOX must be in the system command range.
    ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
    ASSERT(IDM_ABOUTBOX < 0xF000);

    CMenu* pSysMenu = GetSystemMenu(FALSE);
    CString strAboutMenu;
    strAboutMenu.LoadString(IDS_ABOUTBOX);
    if (!strAboutMenu.IsEmpty())
    {
       pSysMenu->AppendMenu(MF_SEPARATOR);
       pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
    }

    m_circctl.SetCaption(_T("Circ 2 Control"));
    if( !m_circctl.GetCircleShape())
       m_circctl.SetCircleShape(TRUE);

    return TRUE;  // return TRUE  unless you set the focus to a control
 }
```

See Also

Concepts

ActiveX Control Containers

ActiveX Control Containers: Using Controls in a Non-Dialog Container

In some applications, such as an SDI or MDI application, you will want to embed a control in a window of the application. The **Create** member function of the wrapper class, inserted by Visual C++, can create an instance of the control dynamically, without the need for a dialog box.

The **Create** member function has the following parameters:

lpszWindowName

A pointer to the text to be displayed in the control's Text or Caption property (if any).

dwStyle

Windows styles. For a complete list, see CWnd::CreateControl.

rect

Specifies the control's size and position.

pParentWnd

Specifies the control's parent window, usually a **CDialog**. It must not be **NULL**.

nID

Specifies the control ID and can be used by the container to refer to the control.

One example of using this function to dynamically create an ActiveX control would be in a form view of an SDI application. You could then create an instance of the control in the **WM_CREATE** handler of the application.

For this example, `CMyView` is the main view class, `CCirc` is the wrapper class, and CIRC.H is the header (.H) file of the wrapper class.

Implementing this feature is a four-step process.

To dynamically create an ActiveX control in a non-dialog window

1. Insert CIRC.H in CMYVIEW.H, just before the `CMyView` class definition:

```
#include "circ.h"
```

2. Add a member variable (of type `CCirc`) to the protected section of the `CMyView` class definition located in CMYVIEW.H:

```
class CMyView : public CView
{
...
protected:

CCirc m_myCtl;
...
};
```

3. Add a **WM_CREATE** message handler to class `CMyView`.

4. In the handler function, `CMyView::OnCreate`, make a call to the control's `Create` function using the **this** pointer as the parent window:

```
int CMyView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (MyView::OnCreate(lpCreateStruct) == -1)
      return -1;
```

```
// ****** Add your code below this line ********** //

   m_myCtl.Create(NULL, WS_VISIBLE,
      CRect(50,50,100,100), this, 0);
   m_myCtl.SetCaption(_T("Control created"));

// ****** Add your code above this line ********** //

   return 0;
}
```

5. Rebuild the project. A Circ control will be created dynamically whenever the application's view is created.

See Also

Concepts

ActiveX Control Containers

Testing Properties and Events with Test Container

The Test Container application, shipped in Visual C++, is an ActiveX control container for testing and debugging ActiveX controls. Test Container allows the control developer to test the control's functionality by changing its properties, invoking its methods, and firing its events. Test Container can display logs of data-binding notifications and also provides facilities for testing an ActiveX control's persistence functionality: you can save properties to a stream or to substorage, reload them, and examine the stored stream data. This section describes how to use the basic features of Test Container. For additional information, select the **Help** menu while running Test Container.

To access the ActiveX Control Test Container

1. Type tstcon32.exe at the command line.

   -or-

2. Select **ActiveX Control Test Container** from the **Tools** menu.

To test your ActiveX control

1. On the **Edit** menu of Test Container, click **Insert New Control**.
2. In the **Insert Control** box, select the desired control and click **OK**. The control will appear in the control container.

Note
If your control is not listed in the **Insert Control** dialog box, make sure you have registered it with the **Register Controls** command from the **File** menu of Test Container.

At this point you can test your control's properties or events.

To test properties

1. On the **Control** menu, click **Invoke Methods**.
2. In the **Method Name** drop-down list, select the PropPut method for the property you want to test.
3. Modify the **Parameter Value** or **Parameter Type** and click on the **Set Value** button.
4. Click **Invoke** to apply the new value to the object.

   The property now contains the new value.

To test events and specify the destination of event information.

1. On the **Options** menu, click **Logging**.
2. Specify the destination of event information.

See Also

Tasks

How to: Debug an ActiveX Control

Concepts

MFC ActiveX Controls

Clipboard

This family of articles explains how to implement support for the Windows Clipboard in MFC applications. The Windows Clipboard is used in two ways:

- Implementing standard Edit menu commands, such as Cut, Copy, and Paste.
- Implementing uniform data transfer with drag and drop (OLE).

The Clipboard is the standard Windows method of transferring data between a source and a destination. It can also be very useful in OLE operations. With the advent of OLE, there are two Clipboard mechanisms in Windows. The standard Windows Clipboard API is still available, but it has been supplemented with the OLE data transfer mechanism. OLE uniform data transfer (UDT) supports Cut, Copy, and Paste with the Clipboard and drag and drop.

The Clipboard is a system service shared by the entire Windows session, so it does not have a handle or class of its own. You manage the Clipboard through member functions of class CWnd.

What do you want to know more about?

- When to use each Clipboard mechanism
- Using the traditional Windows Clipboard API
- Using the OLE Clipboard mechanism
- Copying and pasting data
- Adding other formats
- The Clipboard in the Platform SDK
- Implementing drag and drop (OLE)

See Also

Other Resources

General Concepts
User Interface

Clipboard: When to Use Each Clipboard Mechanism

Follow these guidelines in using the Clipboard:

- Use the OLE Clipboard mechanism now to enable new capabilities in the future. While the standard Clipboard API will be maintained, the OLE mechanism is the future of data transfer.
- Use the OLE Clipboard mechanism if you are writing an OLE application or you want any of the OLE features, such as drag and drop.
- Use the OLE Clipboard mechanism if you are providing OLE formats.

What do you want to do?

- Use the OLE Clipboard mechanism
- Use the Windows Clipboard mechanism

See Also

Concepts

Clipboard

Clipboard: Using the Windows Clipboard

This topic describes how to use the standard Windows Clipboard API within your MFC application.

Most applications for Windows support cutting or copying data to the Windows Clipboard and pasting data from the Clipboard. The Clipboard data formats vary among applications. The framework supports only a limited number of Clipboard formats for a limited number of classes. You will normally implement the Clipboard-related commands — Cut, Copy, and Paste — on the Edit menu for your view. The class library defines the command IDs for these commands: **ID_EDIT_CUT**, **ID_EDIT_COPY**, and **ID_EDIT_PASTE**. Their message-line prompts are also defined.

Messages and Commands in the Framework explains how to handle menu commands in your application by mapping the menu command to a handler function. As long as your application does not define handler functions for the Clipboard commands on the Edit menu, they remain disabled. To write handler functions for the Cut and Copy commands, implement selection in your application. To write a handler function for the Paste command, query the Clipboard to see whether it contains data in a format your application can accept. For example, to enable the Copy command, you might write a handler something like the following:

```
void CMyView::OnEditCopy()
{
  if ( !OpenClipboard() )
  {
    AfxMessageBox( "Cannot open the Clipboard" );
    return;
  }
  // Remove the current Clipboard contents
  if( !EmptyClipboard() )
  {
    AfxMessageBox( "Cannot empty the Clipboard" );
    return;
  }
  // ...
  // Get the currently selected data
  // ...
  // For the appropriate data formats...
  if ( ::SetClipboardData( CF_??, hData ) == NULL )
  {
    AfxMessageBox( "Unable to set Clipboard data" );
    CloseClipboard();
    return;
  }
  // ...
  CloseClipboard();
}
```

The Cut, Copy, and Paste commands are only meaningful in certain contexts. The Cut and Copy commands should be enabled only when something is selected, and the Paste command only when something is in the Clipboard. You can provide this behavior by defining update handler functions that enable or disable these commands depending on the context. For more information, see How to Update User-Interface Objects.

The Microsoft Foundation Class Library does provide Clipboard support for text editing with the **CEdit** and **CEditView** classes. The OLE classes also simplify implementing Clipboard operations that involve OLE items. For more information on the OLE classes, see Clipboard: Using the OLE Clipboard Mechanism.

Implementing other Edit menu commands, such as Undo (**ID_EDIT_UNDO**) and Redo (**ID_EDIT_REDO**), is also left to you. If your application does not support these commands, you can easily delete them from your resource file using the Visual C++ resource editors.

What do you want to know more about?

- Copying and pasting data
- Using the OLE Clipboard mechanism

See Also
Concepts

Clipboard

Clipboard: Using the OLE Clipboard Mechanism

OLE uses standard formats and some OLE-specific formats for transferring data through the Clipboard.

When you cut or copy data from an application, the data is stored on the Clipboard to be used later in paste operations. This data is in a variety of formats. When a user chooses to paste data from the Clipboard, the application can choose which of these formats to use. The application should be written to choose the format that provides the most information, unless the user specifically asks for a certain format, using Paste Special. Before continuing, you may want to read the Data Objects and Data Sources (OLE) topics. They describe the fundamentals of how data transfers work, and how to implement them in your applications.

Windows defines a number of standard formats that can be used for transferring data through the Clipboard. These include metafiles, text, bitmaps, and others. OLE defines a number of OLE-specific formats, as well. For applications that need more detail than given by these standard formats, it is a good idea to register their own custom Clipboard formats. Use the Win32 API function RegisterClipboardFormat to do this.

For example, Microsoft Excel registers a custom format for spreadsheets. This format carries much more information than, for example, a bitmap does. When this data is pasted into an application that supports the spreadsheet format, all the formulas and values from the spreadsheet are retained and can be updated if necessary. Microsoft Excel also puts data on the Clipboard in formats so that it can be pasted as an OLE item. Any OLE document container can paste this information as an embedded item. This embedded item can be changed using Microsoft Excel. The Clipboard also contains a simple bitmap of the image of the selected range on the spreadsheet. This can also be pasted into OLE document containers or into bitmap editors, like Paint. In the case of a bitmap, however, there is no way to manipulate the data as a spreadsheet.

To retrieve the maximum amount of information from the Clipboard, applications should check for these custom formats before pasting data from the Clipboard.

For example, to enable the Copy command, you might write a handler something like the following:

```
void CMyView::OnEditCopy()
{
  // Create an OLE data source on the heap
  COleDataSource* pData = new COleDataSource;
  // ...
  // Get the currently selected data
  // ...
  // For the appropriate data formats...
  pData->CacheData( CF_??, hData );
  // ...
  // The Clipboard now owns the allocated memory
  // and will delete this data object
  // when new data is put on the Clipboard
  pData->SetClipboard();
}
```

What do you want to know more about?

- Copying and pasting data
- Adding other formats
- Using the Windows Clipboard
- OLE
- OLE data objects and data sources and uniform data transfer

See Also

Concepts

Clipboard

Clipboard: Copying and Pasting Data

This topic describes the minimum work necessary to implement copying to and pasting from the Clipboard in your OLE application. It is recommended that you read the Data Objects and Data Sources (OLE) topics before proceeding.

Before you can implement either copying or pasting, you must first provide functions to handle the Copy, Cut, and Paste options on the Edit menu.

Copying or Cutting Data

To copy data to the Clipboard

1. Determine whether the data to be copied is native data or is an embedded or linked item.
   - If the data is embedded or linked, obtain a pointer to the **COleClientItem** object that has been selected.
   - If the data is native and the application is a server, create a new object derived from **COleServerItem** containing the selected data. Otherwise, create a **COleDataSource** object for the data.
2. Call the selected item's **CopyToClipboard** member function.
3. If the user chose a Cut operation instead of a Copy operation, delete the selected data from your application.

To see an example of this sequence, see the **OnEditCut** and **OnEditCopy** functions in the MFC OLE sample programs OCLIENT and HIERSVR. Note that these samples maintain a pointer to the currently selected data, so step 1 is already complete.

Pasting Data

Pasting data is more complicated than copying it because you need to choose the format to use in pasting the data into your application.

To paste data from the Clipboard

1. In your view class, implement **OnEditPaste** to handle users choosing the Paste option from the Edit menu.
2. In the **OnEditPaste** function, create a **COleDataObject** object and call its **AttachClipboard** member function to link this object to the data on the Clipboard.
3. Call **COleDataObject::IsDataAvailable** to check whether a particular format is available.

   Alternately, you can use **COleDataObject::BeginEnumFormats** to look for other formats until you find one most suited to your application.
4. Perform the paste of the format.

For an example of how this works, see the implementation of the **OnEditPaste** member functions in the view classes defined in the MFC OLE sample programs OCLIENT and HIERSVR.

Tip
The main benefit of separating the paste operation into its own function is that the same paste code can be used when data i s dropped in your application during a drag-and-drop operation. As in OCLIENT and HIERSVR, your **OnDrop** function can als o call **DoPasteItem**, reusing the code written to implement Paste operations.

To handle the Paste Special option on the Edit menu, see the topic Dialog Boxes in OLE.

What do you want to know more about?

- Adding other formats
- OLE data objects and data sources and uniform data transfer
- OLE drag and drop
- OLE

See Also

Reference

Clipboard: Using the OLE Clipboard Mechanism

Clipboard: Adding Other Formats

This topic explains how to expand the list of supported formats, particularly for OLE support. The topic Clipboard: Copying and Pasting Data describes the minimum implementation necessary to support copying and pasting from the Clipboard. If this is all you implement, the only formats placed on the Clipboard are **CF_METAFILEPICT**, **CF_EMBEDSOURCE**, **CF_OBJECTDESCRIPTOR**, and possibly **CF_LINKSOURCE**. Most applications will need more formats on the Clipboard than these three.

Registering Custom Formats

To create your own custom formats, follow the same procedure you would use when registering any custom Clipboard format: pass the name of the format to the **RegisterClipboardFormat** function and use its return value as the format ID.

Placing Formats on the Clipboard

To add more formats to those placed on the Clipboard, you must override the **OnGetClipboardData** function in the class you derived from either **COleClientItem** or **COleServerItem** (depending on whether the data to be copied is native). In this function, you should use the following procedure.

To place formats on the Clipboard

1. Create a **COleDataSource** object.
2. Pass this data source to a function that adds your native data formats to the list of supported formats by calling **COleDataSource::CacheGlobalData**.
3. Add standard formats by calling **COleDataSource::CacheGlobalData** for each standard format you want to support.

This technique is used in the MFC OLE sample program HIERSVR (examine the **OnGetClipboardData** member function of the **CServerItem** class). The only difference in this sample is that step three is not implemented because HIERSVR supports no other standard formats.

What do you want to know more about?

- OLE data objects and data sources and uniform data transfer
- OLE drag and drop
- OLE

See Also

Reference

Clipboard: Using the OLE Clipboard Mechanism

Controls (MFC)

Controls are objects that users can interact with to enter or manipulate data. They commonly appear in dialog boxes or on toolbars. This topic family covers three main kinds of controls:

- Windows common controls, including owner-drawn controls
- ActiveX Controls
- Other control classes supplied by the Microsoft Foundation Class Library (MFC)

Windows Common Controls

The Windows operating system has always provided a number of Windows common controls. These control objects are programmable, and the Visual C++ dialog editor supports adding them to your dialog boxes. The Microsoft Foundation Class Library (MFC) supplies classes that encapsulate each of these controls, as shown in the table Windows Common Controls and MFC Classes. (Some items in the table have related topics that describe them further. For controls that lack topics, see the documentation for the MFC class.)

Class CWnd is the base class of all window classes, including all of the control classes. The Windows common controls are supported in the following environments:

- Windows 95, Windows 98, and Windows 2000
- Windows NT, version 3.51 and later
- Win32s, version 1.3 (Visual C++ versions 4.2 and later do not support Win32s)

The older common controls — check boxes, combo boxes, edit boxes, list boxes, option buttons, pushbuttons, scroll bar controls, and static controls — were available in earlier versions of Windows as well.

ActiveX Controls

ActiveX controls, formerly known as OLE controls, can be used in dialog boxes in your applications for Windows, or in HTML pages on the World Wide Web. For more information, see MFC ActiveX Controls.

Other MFC Control Classes

In addition to classes that encapsulate all of the Windows common controls and that support programming your own ActiveX controls (or using ActiveX controls supplied by others), MFC supplies the following control classes of its own:

- CBitmapButton
- CCheckListBox
- CDragListBox

Finding Information About Windows Common Controls

The table below briefly describes each of the Windows common controls, including the control's MFC wrapper class.

Windows Common Controls and MFC Classes

Control	MFC class	Description	New in Windows 95?
animation	CAnimateCtrl	Displays successive frames of an AVI video clip	Yes
button	CButton	Pushbuttons that cause an action; also used for check boxes, radio buttons, and group boxes	No
combo box	CComboBox	Combination of an edit box and a list box	No
date and time picker	CDateTimeCtrl	Allows the user to choose a specific date or time value	Yes

edit box	CEdit	Boxes for entering text	No
extended combo box	CComboBoxEx	A combo box control with the ability to display images	Yes
header	CHeaderCtrl	Button that appears above a column of text; controls width of text displayed	Yes
hotkey	CHotKeyCtrl	Window that enables user to create a "hot key" to perform an action quickl y	Yes
image list	CImageList	Collection of images used to manage large sets of icons or bitmaps (image list isn't really a control; it supports lists used by other controls)	Yes
list	CListCtrl	Window that displays a list of text with icons	Yes
list box	CListBox	Box that contains a list of strings	No
month calendar	CMonthCalCtrl	Control that displays date information	Yes
progress	CProgressCtrl	Window that indicates progress of a long operation	Yes
rebar	CRebarCtrl	Tool bar that can contain additional child windows in the form of controls	Yes
rich edit	CRichEditCtrl	Window in which user can edit with character and paragraph formatting (s ee Classes Related to Rich Edit Controls)	Yes
scroll bar	CScrollBar	Scroll bar used as a control inside a dialog box (not on a window)	No
slider	CSliderCtrl	Window containing a slider control with optional tick marks	Yes
spin button	CSpinButtonCtrl	Pair of arrow buttons user can click to increment or decrement a value	Yes
static-text	CStatic	Text for labeling other controls	No
status bar	CStatusBarCtrl	Window for displaying status information, similar to MFC class **CStatusBar**	Yes
tab	CTabCtrl	Analogous to the dividers in a notebook; used in "tab dialog boxes" or prop erty sheets	Yes
toolbar	CToolBarCtrl	Window with command-generating buttons, similar to MFC class **CToolBa r**	Yes
tool tip	CToolTipCtrl	Small pop-up window that describes purpose of a toolbar button or other t ool	Yes
tree	CTreeCtrl	Window that displays a hierarchical list of items	Yes

What do you want to know more about?

- An individual control: see the table Windows Common Controls and MFC Classes in this topic for links to all controls
- Making and using controls
- Using the dialog editor to add controls
- Adding controls to a dialog box by hand
- Deriving control classes from the MFC control classes

- Using common controls as child windows
- Notifications from common controls
- Add common controls to a dialog box.
- Derive a control from a standard Windows control
- Access dialog-box controls with type safety
- Receive notification messages from common controls
- Samples

For information about Windows common controls in the Platform SDK, see Common Controls.

See Also

Reference

Dialog Editor

Other Resources

General Concepts

User Interface

Common Control Sample List

See the following sample programs that illustrate common controls:

- CMNCTRL1
- CMNCTRL2
- CTRLTEST

See Also

Concepts

Controls (MFC)

Making and Using Controls

You make most controls for dialog boxes in the Visual C++ dialog editor. But you can also create controls in any dialog box or window.

What do you want to know more about?

- Using common controls in a dialog box
- Using the dialog editor to add controls
- Adding controls by hand (without the dialog editor)
- Deriving controls from a standard control
- Using a common control as a child window
- Receiving notification from common controls
- Dialog Data Exchange and Validation
- Type-Safe Access to Controls With Code Wizards
- Individual controls: see Controls (MFC) for links to all common controls
- Dialog boxes
- Dialog bars

See Also
Concepts
Controls (MFC)

Using Common Controls in a Dialog Box

The Windows common controls can be used in dialog boxes, form views, record views, and any other window based on a dialog template. The following procedure, with minor changes, will work for forms as well.

Procedures

To use a common control in a dialog box

1. Place the control on the dialog template using the dialog editor.
2. Add to the dialog class a member variable that represents the control. In the **Add Member Variable** dialog box, check **Control variable** and ensure that **Control** is selected for the **Category**.
3. If this common control is providing input to the program, declare additional member variable(s) in the dialog class to handle those input values.

Note
You can add these member variables using the context menu in Class View (see Adding a Member Variable).

4. In OnInitDialog for your dialog class, set the initial conditions for the common control. Using the member variable created in the previous step, use the member functions to set initial value and other settings. See the following descriptions of the controls for details on settings.

   You can also use dialog data exchange (DDX) to initialize controls in a dialog box.

5. In handlers for controls on the dialog box, use the member variable to manipulate the control. See the following descriptions of the controls for details on methods.

Note
The member variable will exist only as long as the dialog box itself exists. You will not be able to query the control for i nput values after the dialog box has been closed. To work with input values from a common control, override **OnOK** in your dialog class. In your override, query the control for input values and store those values in member variables of the dialog class.

Note
You can also use dialog data exchange to set or retrieve values from the controls in a dialog box.

Remarks

The addition of some common controls to a dialog box will cause the dialog box to no longer function. Refer to Adding Controls to a Dialog Causes the Dialog to No Longer Function for more information on handling this situation.

What do you want to do?

- Add controls to a dialog box by hand instead of with the dialog editor
- Derive my control from one of the standard Windows common controls
- Use a common control as a child window
- Receive notification messages from a control
- Use dialog data exchange (DDX)

See Also

Reference

Making and Using Controls

Concepts

Controls (MFC)

Using the Dialog Editor to Add Controls

When you create your dialog-template resource with the dialog editor, you drag controls from a controls palette and drop them into the dialog box. This adds the specifications for that control type to the dialog-template resource. When you construct a dialog object and call its **Create** or **DoModal** member function, the framework creates a Windows control and places it in the dialog window on screen.

You can instead create controls by hand if you want. This is more work.

See Also

Reference

Making and Using Controls

Concepts

Controls (MFC)

Adding Controls By Hand

You can either add controls to a dialog box with the dialog editor or add them yourself, with code.

To create a control object yourself, you will usually embed the C++ control object in a C++ dialog or frame-window object. Like many other objects in the framework, controls require two-stage construction. You should call the control's **Create** member function as part of creating the parent dialog box or frame window. For dialog boxes, this is usually done in OnInitDialog, and for frame windows, in OnCreate.

The following example shows how you might declare a **CEdit** object in the class declaration of a derived dialog class and then call the **Create** member function in **OnInitDialog**. Because the **CEdit** object is declared as an embedded object, it is automatically constructed when the dialog object is constructed, but it must still be initialized with its own **Create** member function.

```
class CMyDialog : public CDialog
{
protected:
    CEdit m_edit;   // Embedded edit object
public:
    virtual BOOL OnInitDialog();
};
```

The following `OnInitDialog` function sets up a rectangle, then calls **Create** to create the Windows edit control and attach it to the uninitialized **CEdit** object.

```
BOOL CMyDialog::OnInitDialog()
{
    CDialog::OnInitDialog();
    CRect rect(85, 110, 180, 210);

    m_edit.Create(WS_CHILD | WS_VISIBLE | WS_TABSTOP |
             ES_AUTOHSCROLL | WS_BORDER, rect, this, ID_EXTRA_EDIT);
    m_edit.SetFocus();
    return FALSE;
}
```

After creating the edit object, you can also set the input focus to the control by calling the **SetFocus** member function. Finally, you return 0 from **OnInitDialog** to show that you set the focus. If you return a nonzero value, the dialog manager sets the focus to the first control item in the dialog item list. In most cases, you'll want to add controls to your dialog boxes with the dialog editor.

See Also

Reference

Making and Using Controls

CDialog::OnInitDialog

Concepts

Controls (MFC)

Deriving Controls from a Standard Control

As with any CWnd-derived class, you can modify a control's behavior by deriving a new class from an existing control class.

To create a derived control class

1. Derive your class from an existing control class and optionally override the **Create** member function so that it provides the necessary arguments to the base-class **Create** function.
2. Provide message-handler member functions and message-map entries to modify the control's behavior in response to specific Windows messages. See Mapping Messages to Functions.
3. Provide new member functions to extend the functionality of the control (optional).

Using a derived control in a dialog box requires extra work. The types and positions of controls in a dialog box are normally specified in a dialog-template resource. If you create a derived control class, you cannot specify it in a dialog template since the resource compiler knows nothing about your derived class.

To place your derived control in a dialog box

1. Embed an object of the derived control class in the declaration of your derived dialog class.
2. Override the **OnInitDialog** member function in your dialog class to call the **SubclassDlgItem** member function for the derived control.

SubclassDlgItem "dynamically subclasses" a control created from a dialog template. When a control is dynamically subclassed, you hook into Windows, process some messages within your own application, then pass the remaining messages on to Windows. For more information, see the SubclassDlgItem member function of class **CWnd** in the *MFC Reference*. The following example shows how you might write an override of **OnInitDialog** to call **SubclassDlgItem**:

```
BOOL CMyDialog::OnInitDialog()
{
    CDialog::OnInitDialog();
    m_wndMyBtn.SubclassDlgItem(IDC_MYBTN, this);
    return TRUE;
}
```

Because the derived control is embedded in the dialog class, it will be constructed when the dialog box is constructed, and it will be destroyed when the dialog box is destroyed. Compare this code to the example in Adding Controls By Hand.

See Also

Reference

Making and Using Controls

Concepts

Controls (MFC)

Using a Common Control as a Child Window

Any of the Windows common controls can be used as a child window of any other window. The following procedure describes how to create a common control dynamically and then work with it.

To use a common control as a child window

1. Define the control in the related class or handler.
2. Use the control's override of the CWnd::Create method to create the Windows control.
3. After the control has been created (as early as the **OnCreate** handler if you subclass the control), you can manipulate the control using its member functions. See the descriptions of individual controls at Controls (MFC) for details on methods.
4. When you are finished with the control, use CWnd::DestroyWindow to destroy the control.

See Also

Reference

Making and Using Controls

Concepts

Controls (MFC)

Receiving Notification from Common Controls

Common controls are child windows that send notification messages to the parent window when events, such as input from the user, occur in the control.

The application relies on these notification messages to determine what action the user wants it to take. Most common controls send notification messages as **WM_NOTIFY** messages. Windows controls send most notification messages as **WM_COMMAND** messages. CWnd::OnNotify is the handler for the **WM_NOTIFY** message. As with **CWnd::OnCommand**, the implementation of **OnNotify** dispatches the notification message to **OnCmdMsg** for handling in message maps. The message-map entry for handling notifications is **ON_NOTIFY**. For more information, see Technical Note 61: ON_NOTIFY and WM_NOTIFY Messages.

Alternately, a derived class can handle its own notification messages using "message reflection." For more information, see Technical Note 62: Message Reflection for Windows Controls.

Retrieving the Cursor Position in a Notification Message

On occasion, it is useful to determine the current position of the cursor when certain notification messages are received by a common control. For example, it would be helpful to determine the current cursor location when a common control receives a **NM_RCLICK** notification message.

There is a simple way to accomplish this by calling **CWnd::GetCurrentMessage**. However, this method only retrieves the cursor position at the time the message was sent. Because the cursor may have been moved since the message was sent you must call **CWnd::GetCursorPos** to get the current cursor position.

Note
CWnd::GetCurrentMessage should only be called within a message handler.

Add the following code to the body of the notification message handler (in this example, **NM_RCLICK**):

```
CPoint cursorPos;
cursorPos.x= GetCurrentMessage()->pt.x;
cursorPos.y= GetCurrentMessage()->pt.y;
```

At this point, the mouse cursor location is stored in the `cursorPos` object.

See Also

Reference

Making and Using Controls

Concepts

Controls (MFC)

Using CAnimateCtrl

An animation control, represented by the class CAnimateCtrl, is a window that displays a clip in Audio Video Interleaved (AVI) format — the standard Windows video/audio format. An AVI clip is a series of bitmap frames, like a movie.

Since your thread continues executing while the AVI clip is displayed, one common use for an animation control is to indicate system activity during a lengthy operation. For example, the Windows Find dialog box displays a moving magnifying glass as the system searches for a file.

Animation controls can only play simple AVI clips, and they do not support sound. (For a complete list of limitations, see CAnimateCtrl.) Since the capabilities of an animation control are severely limited and subject to change, you should use an alternative such as the MCIWnd control if you need a control to provide multimedia playback and/or recording capabilities. For more information about the MCIWnd control, see the multimedia documentation.

What do you want to know more about?

- Using an Animation Control
- Notifications Sent by Animation Controls

See Also
Concepts
Controls (MFC)

Using an Animation Control

Typical usage of an animation control follows the pattern below:

- The control is created. If the control is specified in a dialog box template, creation is automatic when the dialog box is created. (You should have a CAnimateCtrl member in your dialog class that corresponds to the animation control.) Alternatively, you can use the Create member function to create the control as a child window of any window.
- Load an AVI clip into the animation control by calling the Open member function. If your animation control is in a dialog box, a good place to do this is in the dialog class's OnInitDialog function.
- Play the clip by calling the Play member function. If your animation control is in a dialog box, a good place to do this is in the dialog class's **OnInitDialog** function. Calling **Play** is not necessary if the animation control has the **ACS_AUTOPLAY** style set.
- If you want to display portions of the clip or play it frame by frame, use the **Seek** member function. To stop a clip that is playing, use the **Stop** member function.
- If you are not going to destroy the control right away, remove the clip from memory by calling the **Close** member function.
- If the animation control is in a dialog box, it and the **CAnimateCtrl** object will be destroyed automatically. If not, you need to ensure that both the control and the **CAnimateCtrl** object are properly destroyed. Destroying the control automatically closes the AVI clip.

See Also

Reference

Using CAnimateCtrl

Concepts

Controls (MFC)

Notifications Sent by Animation Controls

An animation control (CAnimateCtrl) sends two different types of notification messages. The notifications are sent in the form of WM_COMMAND messages.

The ACN_START message is sent when the animation control has started playing a clip. The ACN_STOP message is sent when the animation control has finished or stopped playing a clip.

See Also

Reference

Using CAnimateCtrl

Concepts

Controls (MFC)

Using CDateTimeCtrl

The date and time picker control (CDateTimeCtrl) implements an intuitive and recognizable method of entering or selecting a specific date. The main interface of the control is similar in functionality to a combo box. However, if the user expands the control, a month calendar control appears (by default), allowing the user to specify a particular date. When a date is chosen, the month calendar control automatically disappears.

Note
To use both the **CDateTimePicker** and **CMonthCalCtrl** classes in your project, you must include AFXDTCTL.H, usually in yo ur project's STDAFX.H file.

What do you want to know more about?

- Creating the Date and Time Picker Control
- Date and Time Picker Control Examples
- Accessing the Embedded Month Calendar Control
- Using Custom Format Strings in a Date and Time Picker Control
- Using Callback Fields in a Date and Time Picker Control
- Processing Notification Messages in Date and Time Picker Controls

See Also
Concepts
Controls (MFC)

Creating the Date and Time Picker Control

How the date and time picker control is created depends on whether you are using the control in a dialog box or creating it in a nondialog window.

To use CDateTimeCtrl directly in a dialog box

1. In the dialog editor, add a Date and Time Picker Control to your dialog template resource. Specify its control ID.
2. Specify any styles required, using the Properties dialog box of the date and time picker control.
3. Use the Add Member Variable Wizard to add a member variable of type CDateTimeCtrl with the Control property. You can use this member to call **CDateTimeCtrl** member functions.
4. Use the Properties window to map handler functions in the dialog class for any date time picker control notification messages you need to handle (see Mapping Messages to Functions).
5. In OnInitDialog, set any additional styles for the **CDateTimeCtrl** object.

To use CDateTimeCtrl in a nondialog window

1. Declare the control in the view or window class.
2. Call the control's Create member function, possibly in OnInitialUpdate, possibly as early as the parent window's OnCreate handler function (if you're subclassing the control). Set the styles for the control.

See Also

Reference

Using CDateTimeCtrl

Concepts

Controls (MFC)

Date and Time Picker Control Examples

The CMNCTRL1 sample demonstrates the various attributes of the **CDateTimeCtrl** class. A separate page contains a date and time picker control that the user can manipulate by changing various attributes and testing the basic functionality of the control.

See Also

Reference

Using CDateTimeCtrl

Concepts

Controls (MFC)

Accessing the Embedded Month Calendar Control

The embedded month calendar control object can be accessed from the **CDateTimeCtrl** object with a call to the GetMonthCalCtrl member function.

Note
The embedded month calendar control is used only when the date and time picker control does not have the **DTS_UPDOWN** style set.

This is useful if you want to modify certain attributes before the embedded control is displayed. To accomplish this, handle the **DTN_DROPDOWN** notification, retrieve the month calendar control (using CDateTimeCtrl::GetMonthCalCtrl), and make your modifications. Unfortunately, the month calendar control is not persistent.

In other words, when the user requests the display of the month calendar control, a new month calendar control is created (before the **DTN_DROPDOWN** notification). The control is destroyed (after the **DTN_CLOSEUP** notification) when dismissed by the user. This means that any attributes you modify, before the embedded control is displayed, are lost when the embedded control is dismissed.

The following example demonstrates this procedure, using a handler for the **DTN_DROPDOWN** notification. The code changes the background color of the month calendar control, with a call to SetMonthCalColor, to gray. The code is as follows:

```
void CMyDlg::OnDropdown(NMHDR* pNMHDR, LRESULT* pResult)
{

     //set the background color of the month to gray
     COLORREF clr= RGB(100, 100, 100);
     SetMonthCalColor(MCSC_MONTHBK, clr);

     *pResult = 0;
}
```

As stated previously, all modifications to properties of the month calendar control are lost, with two exceptions, when the embedded control is dismissed. The first exception, the colors of the month calendar control, has already been discussed. The second exception is the font used by the month calendar control. You can modify the default font by making a call to CDateTimeCtrl::SetMonthCalFont, passing the handle of an existing font. The following example (where `m_dtPicker` is the date and time control object) demonstrates one possible method:

```
CMonthCalCtrl* pMCCtrl= NULL;

pMCCtrl= m_dtPicker.GetMonthCalCtrl();

//create and initialize the font to be used
LOGFONT logFont;

logFont.lfHeight = -12;
logFont.lfWidth = 0;
logFont.lfWeight = FW_NORMAL;
logFont.lfItalic = FALSE;
logFont.lfUnderline = FALSE;
logFont.lfStrikeOut = FALSE;
logFont.lfEscapement = 0;
logFont.lfOrientation = 0;
lstrcpy(logFont.lfFaceName, _T("Verdana"));

m_MCFont.CreateFontIndirect(&logFont);
m_dtPicker.SetMonthCalFont((HFONT)m_MCFont);
```

Once the font has been changed, with a call to **CDateTimeCtrl::SetMonthCalFont**, the new font is stored and used the next time a month calendar is to be displayed.

See Also

Reference

Using CDateTimeCtrl

Concepts

Controls (MFC)

Using Custom Format Strings in a Date and Time Picker Control

By default, date and time picker controls provide three format types (each format corresponding to a unique style) for displaying the current date or time:

- **DTS_LONGDATEFORMAT** Displays the date in long format, producing output like "Wednesday, January 3, 2000".
- **DTS_SHORTDATEFORMAT** Displays the date in short format, producing output like "1/3/00".
- **DTS_TIMEFORMAT** Displays the time in long format, producing output like "5:31:42 PM".

However, you can customize the appearance of the date or time by using a custom format string. This custom string is made up of either existing format characters, nonformat characters, or a combination of both. Once the custom string is built, make a call to CDateTimeCtrl::SetFormat passing in your custom string. The date and time picker control will then display the current value using your custom format string.

The following example code (where `m_dtPicker` is the **CDateTimeCtrl** object) demonstrates one possible solution:

```
CString formatStr= _T("'Today is: 'yy'/'MM'/'dd");
m_dtPicker.SetFormat(formatStr);
```

In addition to custom format strings, date and time picker controls also support callback fields.

See Also

Reference

Using CDateTimeCtrl

Concepts

Controls (MFC)

Using Callback Fields in a Date and Time Picker Control

In addition to the standard format characters that define date and time picker fields, you can customize your output by specifying certain parts of a custom format string as callback fields. To declare a callback field, include one or more "X" characters (ASCII Code 88) anywhere in the body of the format string. For example, the following string "'Today is: 'yy'/'MM'/'dd' (Day 'X')'"causes the date and time picker control to display the current value as the year followed by the month, date, and finally the day of the year.

Note
The number of X's in a callback field does not correspond to the number of characters that will be displayed.

You can distinguish between multiple callback fields in a custom string by repeating the "X" character. Thus, the format string "XXddddMMMdd', 'yyyXXX" contains two unique callback fields, "XX" and "XXX".

Note
Callback fields are treated as valid fields, so your application must be prepared to handle **DTN_WMKEYDOWN** notification messages.

Implementing callback fields in your date and time picker control consists of three parts:

- Initializing the custom format string
- Handling the **DTN_FORMATQUERY** notification
- Handling the **DTN_FORMAT** notification

Initializing the Custom Format String

Initialize the custom string with a call to **CDateTimeCtrl::SetFormat**. For more information, see Using Custom Format Strings in a Date and Time Picker Control. A common place to set the custom format string is in the **OnInitDialog** function of your containing dialog class or **OnInitialUpdate** function of your containing view class.

Handling the DTN_FORMATQUERY Notification

When the control parses the format string and encounters a callback field, the application sends **DTN_FORMAT** and **DTN_FORMATQUERY** notification messages. The callback field string is included with the notifications so you can determine which callback field is being queried.

The **DTN_FORMATQUERY** notification is sent to retrieve the maximum allowable size in pixels of the string that will be displayed in the current callback field.

To properly calculate this value, you must calculate the height and width of the string, to be substituted for the field, using the control's display font. The actual calculation of the string is easily achieved with a call to the GetTextExtentPoint32 Win32 function. Once the size is determined, pass the value back to the application and exit the handler function.

The following example is one method of supplying the size of the callback string:

```
NMDATETIMEFORMATQUERY* pDTFormatQuery= (NMDATETIMEFORMATQUERY *)pNMHDR;
CDC* pDC;
CFont* pFont;
CFont* pOrigFont;

//  Prepare the device context for the GetTextExtentPoint32 call.
pDC= GetDC();

pFont= GetFont();
if(!pFont)
     pFont->CreateStockObject(DEFAULT_GUI_FONT);


pOrigFont = pDC->SelectObject(pFont);
```

```
// Check to see if this is the callback segment desired. If so,
// use the longest text segment to determine the maximum
// width of the callback field, and then place the information into
// the NMDATETIMEFORMATQUERY structure.
if(!lstrcmp("X",pDTFormatQuery->pszFormat))
     ::GetTextExtentPoint32 (pDC->m_hDC, "366",
          3, &pDTFormatQuery->szMax);

// Reset the font in the device context then release the context.
pDC->SelectObject(pOrigFont);
ReleaseDC(pDC);

*pResult = 0;
```

Once the size of the current callback field has been calculated, you must supply a value for the field. This is done in the handler for the **DTN_FORMAT** notification.

Handling the DTN_FORMAT Notification

The **DTN_FORMAT** notification is used by the application to request the character string that will be substituted. The following example demonstrates one possible method:

```
NMDATETIMEFORMAT* pDTFormat= (NMDATETIMEFORMAT *)pNMHDR;

COleDateTime oCurTime;
m_dtPicker.GetTime(oCurTime);

wsprintf(pDTFormat->pszDisplay,
    "%d",oCurTime.GetDayOfYear());

*pResult = 0;
```

Note

The pointer to the **NMDATETIMEFORMAT** structure is found by casting the first parameter of the notification handler to the proper type.

See Also

Reference

Using CDateTimeCtrl

Concepts

Controls (MFC)

Processing Notification Messages in Date and Time Picker Controls

As users interact with the date and time picker control, the control (**CDateTimeCtrl**) sends notification messages to its parent window, usually a view or dialog object. Handle these messages if you want to do something in response. For example, when the user opens the date and time picker to display the embedded month calendar control, the **DTN_DROPDOWN** notification is sent.

Use the Properties window to add notification handlers to the parent class for those messages you want to implement.

The following list describes the various notifications sent by the date and time picker control.

- **DTN_DROPDOWN** Notifies the parent that the embedded month calendar control is about to be displayed. This notification is only sent when the **DTS_UPDOWN** style has not been set. For more information on this notification, see Accessing the Embedded Month Calendar Control.
- **DTN_CLOSEUP** Notifies the parent that the embedded month calendar control is about to be closed. This notification is only sent when the **DTS_UPDOWN** style has not been set.
- **DTN_DATETIMECHANGE** Notifies the parent that a change has occurred in the control.
- **DTN_FORMAT** Notifies the parent that text is needed to be displayed in a callback field. For more information on this notification and callback fields, see Using Callback Fields in a Date and Time Picker Control.
- **DTN_FORMATQUERY** Requests the parent to supply the maximum allowable size of the string that will be displayed in a callback field. Handling this notification allows the control to properly display output at all times, reducing flicker within the control's display. For more information on this notification, see Using Callback Fields in a Date and Time Picker Control.
- **DTN_USERSTRING** Notifies the parent that the user has finished editing the contents of the date and time picker control. This notification is only sent when the **DTS_APPCANPARSE** style has been set.
- **DTN_WMKEYDOWN** Notifies the parent when the user types in a callback field. Handle this notification to emulate the same keyboard response supported for non-callback fields in a date and time picker control. For more information on this notification, see Supporting Callback Fields in a DTP Control in the *Platform SDK*.

See Also

Reference

Using CDateTimeCtrl

Concepts

Controls (MFC)

Using CComboBoxEx

The extended combo box control is an extension of the standard combo box control that provides native support for item images. These images can be used to indicate the status of individual items in the combo box, such as the currently selected and unselected items. To make item images easily accessible, the control provides image list support.

Use this control to provide the functionality of a combo box without having to manually draw item graphics.

What do you want to know more about?

- Creating an Extended Combo Box Control
- Using Image Lists in an Extended Combo Box Control
- Setting the Images for an Individual Item
- Processing Notification Messages in Extended Combo Box Controls

See Also
Concepts
Controls (MFC)

Creating an Extended Combo Box Control

How the extended combo box control is created depends on whether you are using the control in a dialog box or creating it in a nondialog window.

To use CComboBoxEx directly in a dialog box

1. In the dialog editor, add an Extended Combo Box control to your dialog template resource. Specify its control ID.
2. Specify any styles required, using the Properties dialog box of the extended combo box control.
3. Use the Add Member Variable Wizard to add a member variable of type CComboBoxEx with the Control property. You can use this member to call **CComboBoxEx** member functions.
4. Use the Properties window to map handler functions in the dialog class for any extended combo box control notification messages you need to handle (see Mapping Messages to Functions).
5. In OnInitDialog, set any additional styles for the **CComboBoxEx** object.

To use CComboBoxEx in a nondialog window

1. Define the control in the view or window class.
2. Call the control's Create member function, possibly in OnInitialUpdate, possibly as early as the parent window's OnCreate handler function. Set the styles for the control.

See Also

Reference

Using CComboBoxEx

Concepts

Controls (MFC)

Using Image Lists in an Extended Combo Box Control

The main feature of extended combo box controls is the ability to associate images from an image list with individual items in a combo box control. Each item is able to display three different images: one for its selected state, one for its nonselected state, and a third for an overlay image.

The following procedure associates an image list with an extended combo box control:

To associate an image list with an extended combo box control

1. Construct a new image list (or use an existing image list object), using the CImageList constructor and storing the resultant pointer.
2. Initialize the new image list object by calling CImageList::Create. The following code is one example of this call.

   ```
   m_pImageList->Create(16, 16, ILC_COLOR, 2, 2);
   ```

3. Add optional images for each possible state: selected or nonselected, and an overlay. The following code adds three predefined images.

   ```
   m_pImageList->Add(pApp->m_hIconSelected);
   m_pImageList->Add(pApp->m_hIconNotSelected);
   m_pImageList->Add(pApp->m_hIconOverlay);
   ```

4. Associate the image list with the control with a call to CComboBoxEx::SetImageList.

Once the image list has been associated with the control, you can individually specify the images each item will use for the three possible states. For more information, see Setting the Images for an Individual Item.

See Also

Reference

Using CComboBoxEx

Concepts

Controls (MFC)

Setting the Images for an Individual Item

The different types of images used by the extended combo box item are determined by the values in the **iImage**, **iSelectedImage**, and **iOverlay** members of the COMBOBOXEXITEM structure. Each value is the index of an image in the associated image list of the control. By default, these members are set to 0, causing the control to display no image for the item. If you want to use images for a specific item, you can modify the structure accordingly, either when inserting the combo box item or by modifying an existing combo box item.

Setting the Image for a New Item

If you are inserting a new item, initialize the **iImage**, **iSelectedImage**, and **iOverlay** structure members with the proper values and then insert the item with a call to CComboBoxEx::InsertItem.

The following example inserts a new extended combo box item (`cbi`) into the extended combo box control (`m_comboEx`), supplying indices for all three image states:

```
COMBOBOXEXITEM     cbi;
CString            str;
int                nItem;

cbi.mask = CBEIF_IMAGE | CBEIF_INDENT | CBEIF_OVERLAY |
     CBEIF_SELECTEDIMAGE | CBEIF_TEXT;

cbi.iItem = i;
str.Format(_T("Item %02d"), i);
cbi.pszText = (LPTSTR)(LPCTSTR)str;
cbi.cchTextMax = str.GetLength();
cbi.iImage = 0;
cbi.iSelectedImage = 1;
cbi.iOverlay = 2;
cbi.iIndent = (i & 0x03);   //Set indentation according
                            //to item position

nItem = m_comboEx.InsertItem(&cbi);
ASSERT(nItem == i);
```

Setting the Image for an Existing Item

If you are modifying an existing item, you need to work with the **mask** member of a **COMBOBOXEXITEM** structure.

To modify an existing item to use images

1. Declare a **COMBOBOXEXITEM** structure and set the **mask** data member to the values you are interested in modifying.
2. Using this structure, make a call to CComboBoxEx::GetItem.
3. Modify the **mask**, **iImage**, and **iSelectedImage** members of the newly returned structure, using the appropriate values.
4. Make a call to CComboBoxEx::SetItem, passing in the modified structure.

The following example demonstrates this procedure by swapping the selected and unselected images of the third extended combo box item:

```
COMBOBOXEXITEM     cbi;
CString            str;
int                nItem;

cbi.mask = CBEIF_IMAGE | CBEIF_SELECTEDIMAGE;
cbi.iItem = 2;
m_comboEx.GetItem(&cbi);

cbi.iImage = 1;
cbi.iSelectedImage = 0;
nItem = m_comboEx.SetItem(&cbi);
```

```
ASSERT(nItem != 0);
```

See Also

Reference

Using CComboBoxEx

Concepts

Controls (MFC)

Processing Notification Messages in Extended Combo Box Controls

As users interact with the extended combo box, the control (**CComboBoxEx**) sends notification messages to its parent window, usually a view or dialog object. Handle these messages if you want to do something in response. For example, when the user activates the drop-down list or clicks in the control's edit box, the **CBEN_BEGINEDIT** notification is sent.

Use the Properties window to add notification handlers to the parent class for those messages you want to implement.

The following list describes the various notifications sent by the extended combo box control.

- **CBEN_BEGINEDIT** Sent when the user activates the drop-down list or clicks in the control's edit box.
- **CBEN_DELETEITEM** Sent when an item has been deleted.
- **CBEN_DRAGBEGIN** Sent when the user begins dragging the image of the item displayed in the edit portion of the control.
- **CBEN_ENDEDIT** Sent when the user has concluded an operation within the edit box or has selected an item from the control's drop-down list.
- **CBEN_GETDISPINFO** Sent to retrieve display information about a callback item.
- **CBEN_INSERTITEM** Sent when a new item has been inserted in the control.

See Also
Reference
Using CComboBoxEx
Concepts
Controls (MFC)

Using CHeaderCtrl

Use a header control, represented by class CHeaderCtrl, to display column headers for a columnar list. For example, a header control would be useful for implementing column controls in a spreadsheet.

The header control is usually divided into parts, called "header items," each bearing a title for the associated column of text or numbers. Depending on the styles you set, you can provide a number of direct ways for users to manipulate the header items.

Note
CListCtrl provides an embedded header control, and CListView encapsulates **CListCtrl** in an MFC class. In general, think of us ing **CHeaderCtrl** to label lists that you intend to draw yourself.

What do you want to know more about?

- Header Control and List Control
- Header Control Examples
- Header Items in a Header Control
- Customizing the Header Item's Appearance
- Providing Drag-and-Drop Support for Header Items
- Using Image Lists with Header Controls
- Making Owner-Drawn Header Controls
- Working with a Header Control
- Creating the Header Control
- Adding Items to the Header Control
- Ordering Items in the Header Control
- Processing Header-Control Notifications

See Also
Concepts
Controls (MFC)

Header Control and List Control

In most cases, you will use the header control that is embedded in a CListCtrl or CListView object. However, there are cases where a separate header control object is desirable, such as manipulating data, arranged in columns or rows, in a CView-derived object. In these cases, you need greater control over the appearance and default behavior of an embedded header control.

In the common case that you want a header control to provide standard, default behavior, you may want to use CListCtrl or CListView instead. Use **CListCtrl** when you want the functionality of a default header control, embedded in a list view common control. Use CListView when you want the functionality of a default header control, embedded in a view object.

Note
These controls only include a built-in header control if the list view control is created using the **LVS_REPORT** style.

In most cases, the appearance of the embedded header control can be modified by changing the styles of the containing list view control. In addition, information about the header control can be obtained through member functions of the parent list view control. However, for complete control, and access, to the attributes and styles of the embedded header control, it is recommended that a pointer to the header control object be obtained.

The embedded header control object can be accessed from either **CListCtrl** or **CListView** with a call to the respective class's **GetHeaderCtrl** member function. The following code demonstrates this:

```
CListCtrl& refList = GetListCtrl();
CHeaderCtrl* pCtrl = refList.GetHeaderCtrl();
ASSERT(pCtrl != NULL);
//perform any needed operations on the header
//using pHeader
```

What do you want to know more about?

- Using image lists with header controls

See Also

Reference

Using CHeaderCtrl

Concepts

Controls (MFC)

Header Control Examples

For examples of header controls, see the Header Controls in the *Platform SDK*.

See Also

Reference

Using CHeaderCtrl

Concepts

Controls (MFC)

Header Items in a Header Control

You have considerable control over the appearance and behavior of the header items that make up a header control (CHeaderCtrl). Each header item can have a string, a bitmapped image, an image from an associated image list, or an application-defined 32-bit value associated with it. The string, bitmap, or image is displayed in the header item.

You can customize the appearance and contents of new items when they are created by making a call CHeaderCtrl::InsertItem or by modifying an existing item, with a call to CHeaderCtrl::GetItem and CHeaderCtrl::SetItem.

What do you want to know more about?

- Customizing the header item's appearance
- Ordering items in the header control
- Providing drag-and-drop support for the header items
- Using image lists with header controls

See Also

Reference

Using CHeaderCtrl

Customizing the Header Item's Appearance

By setting the *dwStyle* parameter when you first create a header control (CHeaderCtrl::Create), you can define the appearance and behavior of header items or of the header control itself.

Here is a sampling of the styles you can set, and their purpose:

- To make a header item look like a pushbutton, use the **HDS_BUTTONS** style.

  Use this style if you want to carry out actions in response to mouse clicks on a header item, such as sorting data by a particular column, as is done in Microsoft Outlook.

- To give the header items a "hot tracking" appearance when the mouse cursor passes over them, use the **HDS_HOTTRACK** style.

  Hot tracking displays a 3D outline as the pointer passes over an item in an otherwise flat bar.

- To indicate that the header control should be hidden, use the **HDS_HIDDEN** style.

  The **HDS_HIDDEN** style indicates that the header control is intended to be used as a data container and not a visual control. This style does not automatically hide the control but, instead, affects the behavior of **CHeaderCtrl::Layout**. The value returned in the **cy** member of the **WINDOWPOS** structure will be zero indicating that the control should not be visible to the user.

For more information about these properties, see Items in the *Platform SDK*. For information about adding items to a header control, see Adding Items to the Header Control.

See Also
Reference
Using CHeaderCtrl
Concepts
Controls (MFC)

Providing Drag-and-Drop Support for Header Items

To provide drag-and-drop support for header items, specify the **HDS_DRAGDROP** style. Drag-and-drop support for header items gives the user the ability to reorder the header items of a header control. The default behavior provides a semitransparent drag image of the header item being dragged and a visual indicator of the new position, if the header item is dropped.

As with common drag-and-drop functionality, you can extend the default drag-and-drop behavior by handling the **HDN_BEGINDRAG** and **HDN_ENDDRAG** notifications. You can also customize the appearance of the drag image by overriding the CHeaderCtrl::CreateDragImage member function.

Note
If you are providing drag-and-drop support for an embedded header control in a list control, see the Extended Style section in the Changing List Control Styles topic.

See Also

Reference

Using CHeaderCtrl

Using Image Lists with Header Controls

Header items have the ability to display an image within a header item. This image, stored in an associated image list, is 16 x 16 pixels and has the same characteristics as the icon images used in a list view control. In order to implement this behavior successfully, you must first create and initialize the image list, associate the list with the header control, and then modify the attributes of the header item that will display the image.

The following procedure illustrates the details, using a pointer to a header control (`m_pHdrCtrl`) and a pointer to an image list (`m_pHdrImages`).

To display an image in a header item

1. Construct a new image list (or use an existing image list object) using the CImageList constructor, storing the resultant pointer.
2. Initialize the new image list object by calling CImageList::Create. The following code is one example of this call.

```
m_pHdrImages->Create(16, 16, ILC_MASK, 2, 2);
```

3. Add the images for each header item. The following code adds two predefined images.

```
m_pHdrImages->Add(pApp->LoadIcon(IDI_ICONHDR1));
m_pHdrImages->Add(pApp->LoadIcon(IDI_ICONHDR2));
```

4. Associate the image list with the header control with a call to CHeaderCtrl::SetImageList.
5. Modify the header item to display an image from the associated image list. The following example assigns the first image, from `m_phdrImages`, to the first header item, `m_pHdrCtrl`.

```
HDITEM    curItem;

m_pHdrCtrl->SetImageList(m_pHdrImages);
m_pHdrCtrl->GetItem(0, &curItem);
curItem.mask= HDI_IMAGE | HDI_FORMAT;
curItem.iImage= 0;
curItem.fmt= HDF_LEFT | HDF_IMAGE | HDF_STRING;
m_pHdrCtrl->SetItem(0, &curItem);
```

For detailed information on the parameter values used, consult the pertinent CHeaderCtrl.

Note

It is possible to have multiple controls using the same image list. For instance, in a standard list view control, there could be a n image list (of 16 x 16 pixel images) used by both the small icon view of a list view control and the header items of the list vi ew control.

See Also

Reference

Using CHeaderCtrl

Making Owner-Drawn Header Controls

You can define individual items of a header control (CHeaderCtrl) to be owner-drawn items. For more information, see Owner-Drawn Header Controls in the *Platform SDK*.

See Also

Reference

Using CHeaderCtrl

Concepts

Controls (MFC)

Working with a Header Control

The easy way to use a header control (CHeaderCtrl) is in conjunction with a list control; see Using CListCtrl later in this topic family. You can also use a header control by itself. MFC calls **InitCommonControls** for you. The key tasks are as follows:

- Creating the header control
- Adding items to the header control
- Ordering items in the header control
- Processing header-control notifications

If the header control object is embedded in a parent view or dialog class, the control is destroyed when the parent is destroyed.

See Also
Reference
Using CHeaderCtrl
Concepts
Controls (MFC)

Creating the Header Control

The header control is not directly available in the dialog editor (although you can add a list control, which includes a header control).

To put a header control in a dialog box

1. Manually embed a member variable of type CHeaderCtrl in your dialog class.
2. In OnInitDialog, create and set the styles for the **CHeaderCtrl**, position it, and display it.
3. Add items to the header control.
4. Use the Properties window to map handler functions in the dialog class for any header-control notification messages you need to handle (see Mapping Messages to Functions).

To put a header control in a view (not a CListView)

1. Embed a CHeaderCtrl object in your view class.
2. Style, position, and display the header control window in the view's OnInitialUpdate member function.
3. Add items to the header control.
4. Use the Properties window to map handler functions in the view class for any header-control notification messages you need to handle (see Mapping Messages to Functions).

In either case, the embedded control object is created when the view or dialog object is created. Then you must call CHeaderCtrl::Create to create the control window. To position the control, call CHeaderCtrl::Layout to determine the control's initial size and position and SetWindowPos to set the position you want. Then add items as described in Adding Items to the Header Control.

For more information, see Creating a Header Control in the *Platform SDK*.

See Also

Reference

Using CHeaderCtrl

Concepts

Controls (MFC)

Adding Items to the Header Control

After creating your header control (CHeaderCtrl) in its parent window, add as many "header items" as you need: usually one per column.

To add a header item

1. Prepare an HD_ITEM structure.
2. Call CHeaderCtrl::InsertItem, passing the structure.
3. Repeat steps 1 and 2 for additional items.

For more information, see Adding an Item to a Header Control in the *Platform SDK*.

See Also

Reference

Using CHeaderCtrl

Concepts

Controls (MFC)

Ordering Items in the Header Control

Once you've added items to a header control, you can manipulate or get information about their order with the following functions:

- CHeaderCtrl::GetOrderArray and CHeaderCtrl::SetOrderArray

  Retrieves and sets the left-to-right order of header items.

- CHeaderCtrl::OrderToIndex.

  Retrieves the index value for a specific header item.

In addition to the previous member functions, the **HDS_DRAGDROP** style allows the user to drag and drop header items within the header control. For more information, see Providing Drag-and-Drop Support for Header Items.

See Also
Reference
Using CHeaderCtrl

Processing Header-Control Notifications

In your view or dialog class, use the Properties window to create an OnChildNotify handler function with a switch statement for any header-control (CHeaderCtrl) notification messages you want to handle (see Mapping Messages to Functions). Notifications are sent to the parent window when the user clicks or double-clicks a header item, drags a divider between items, and so on.

The notification messages associated with a header control are listed in Header Control Reference in the *Platform SDK*.

See Also

Reference

Using CHeaderCtrl

Concepts

Controls (MFC)

Using CHotKeyCtrl

A hot key control, represented by class CHotKeyCtrl, is a window that displays a text representation of the key combination the user types into it, such as CTRL+SHIFT+Q. It also maintains an internal representation of this key in the form of a virtual key code and a set of flags that represent the shift state. The hot key control does not actually set the hot key — doing that is up to your program. (For a list of standard virtual key codes, see Winuser.h.)

Use a hot key control to get a user's input for which hot key to associate with a window or thread. Hot key controls are often used in dialog boxes, such as you might display when asking the user to assign a hot key. It is your program's responsibility to retrieve the values describing the hot key from the hot key control and to call the appropriate functions to associate the hot key with a window or thread.

What do you want to know more about?

- Using a Hot Key Control
- Setting a Hot Key
- Global Hot Keys
- Thread-Specific Hot Keys

See Also
Concepts
Controls (MFC)

Using a Hot Key Control

Typical usage of a hot key control follows the pattern below:

- The control is created. If the control is specified in a dialog box template, creation is automatic when the dialog box is created. (You should have a CHotKeyCtrl member in your dialog class that corresponds to the hot key control.) Alternatively, you can use the Create member function to create the control as a child window of any window.
- If you want to set a default value for the control, call the SetHotKey member function. If you want to prohibit certain shift states, call SetRules. For controls in a dialog box, a good time to do this is in the dialog box's OnInitDialog function.
- The user interacts with the control by pressing a hot key combination when the hot key control has focus. The user then somehow indicates that this task is complete, perhaps by clicking a button in the dialog box.
- When your program is notified that the user has selected a hot key, it should use the member function GetHotKey to retrieve the virtual key and shift state values from the hot key control.
- Once you know what key the user selected, you can set the hot key using one of the methods described in Setting a Hot Key.
- If the hot key control is in a dialog box, it and the **CHotKeyCtrl** object will be destroyed automatically. If not, you need to ensure that both the control and the **CHotKeyCtrl** object are properly destroyed.

See Also
Reference
Using CHotKeyCtrl
Concepts
Controls (MFC)

Setting a Hot Key

Your application can use the information provided by a hot key (CHotKeyCtrl) control in one of two ways:

- Set up a global hot key for activating a nonchild window by sending a WM_SETHOTKEY message to the window to be activated.
- Set up a thread-specific hot key by calling the Windows function RegisterHotKey.

See Also

Reference

Using CHotKeyCtrl

Concepts

Controls (MFC)

Global Hot Keys

A global hot key is associated with a particular nonchild window. It allows the user to activate the window from any part of the system. An application sets a global hot key for a particular window by sending the WM_SETHOTKEY message to that window. For instance, if `m_HotKeyCtrl` is the CHotKeyCtrl object and `pMainWnd` is a pointer to the window to be activated when the hot key is pressed, you could use the following code to associate the hot key specified in the control with the window pointed to by `pMainWnd`.

```
WORD wKeyAndShift = m_HotKeyCtrl.GetHotKey( );
pMainWnd->SendMessage( WM_SETHOTKEY, wKeyAndShift );
```

Whenever the user presses a global hot key, the window specified receives a WM_SYSCOMMAND message that specifies **SC_HOTKEY** as the type of the command. This message also activates the window that receives it. Because this message does not include any information on the exact key that was pressed, using this method does not allow distinguishing between different hot keys that may be attached to the same window. The hot key remains valid until the application that sent **WM_SETHOTKEY** exits.

See Also

Reference

Using CHotKeyCtrl

Concepts

Controls (MFC)

Thread-Specific Hot Keys

An application sets a thread-specific hot key (CHotKeyCtrl) by using the Windows **RegisterHotKey** function. When the user presses a thread-specific hot key, Windows posts a WM_HOTKEY message to the beginning of a particular thread's message queue. The **WM_HOTKEY** message contains the virtual key code, shift state, and user-defined ID of the specific hot key that was pressed. For a list of standard virtual key codes, see Winuser.h. For more information on this method, see RegisterHotKey.

Note that the shift state flags used in the call to **RegisterHotKey** are not the same as those returned by the GetHotKey member function; you'll have to translate these flags before calling **RegisterHotKey**.

See Also

Reference

Using CHotKeyCtrl

Concepts

Controls (MFC)

Using CImageList

An image list, represented by class CImageList, is a collection of same-sized images, each of which can be referred to by its index. Image lists are used to efficiently manage large sets of icons or bitmaps. Image lists are not themselves controls since they are not windows; however, they are used with several different types of controls, including list controls (CListCtrl), tree controls (CTreeCtrl), and tab controls (CTabCtrl).

All images in an image list are contained in a single, wide bitmap in screen-device format. An image list may also include a monochrome bitmap that contains masks used to draw images transparently (icon style). **CImageList** provides member functions that enable you to draw images, create and destroy image lists, add and remove images, replace images, merge images, and drag images.

What do you want to know more about?

- Types of Image Lists
- Using an Image List
- Manipulating Image Lists
- Drawing Images from an Image List
- Image Overlays in Image Lists
- Dragging Images from an Image List
- Image Information in Image Lists

See Also
Concepts
Controls (MFC)

Types of Image Lists

There are two types of image lists (CImageList): nonmasked and masked. A "nonmasked image list" consists of a color bitmap that contains one or more images. A "masked image list" consists of two bitmaps of equal size. The first is a color bitmap that contains the images, and the second is a monochrome bitmap that contains a series of masks — one for each image in the first bitmap.

One of the overloads of the **Create** member function takes a flag to indicate whether or not the image list is masked. (The other overloads create masked image lists.)

When a nonmasked image is drawn, it is simply copied into the target device context; that is, it is drawn over the existing background color of the device context. When a masked image is drawn, the bits of the image are combined with the bits of the mask, typically producing transparent areas in the bitmap where the background color of the target device context shows through. You can specify several drawing styles when drawing a masked image. For example, you can specify that the image be dithered to indicate a selected object.

See Also

Reference

Using CImageList

Concepts

Controls (MFC)

Using an Image List

Typical usage of an image list follows the pattern below:

- Construct a CImageList object and call one of the overloads of its Create function to create an image list and attach it to the **CImageList** object.
- If you didn't add images when you created the image list, add images to the image list by calling the Add or Read member function.
- Associate the image list with a control by calling the appropriate member function of that control, or draw images from the image list yourself using the image list's Draw member function.
- Perhaps allow the user to drag an image, using the image list's built-in support for dragging.

Note
If the image list was created with the **new** operator, you must destroy the **CImageList** object when you are done with it.

See Also

Reference

Using CImageList

Concepts

Controls (MFC)

Manipulating Image Lists

The Replace member function replaces an image in an image list (CImageList) with a new image. This function is also useful if you need to dynamically increase the number of images in an image list object. The SetImageCount function dynamically changes the number of images stored in the image list. If you increase the size of the image list, call **Replace** to add images to the new image slots. If you decrease the size of the image list, the images beyond the new size are freed.

The Remove member function removes an image from an image list. The Copy member function can copy or swap images within an image list. This function allows you to indicate whether the source image should be copied to the destination index or the source and destination images should be swapped.

To create a new image list by merging two image lists, use the appropriate overload of the Create member function. This overload of **Create** merges the first image of the existing image lists, storing the resultant image in a new image list object. The new image is created by drawing the second image transparently over the first. The mask for the new image is the result of performing a logical-OR operation on the bits of the masks for the two existing images.

This is repeated until all images are merged and added to the new image list object.

You can write the image information to an archive by calling the Write member function, and read it back by calling the Read member function.

The GetSafeHandle, Attach, and Detach member functions allow you to manipulate the handle of the image list attached to the **CImageList** object, while the DeleteImageList member function deletes the image list without destroying the **CImageList** object.

See Also

Reference

Using CImageList

Concepts

Controls (MFC)

Drawing Images from an Image List

To draw an image, use the CImageList::Draw member function. You'll specify a pointer to a device context object, the index of the image to draw, the location in the device context at which to draw the image, and a set of flags to indicate the drawing style.

When you specify the **ILD_TRANSPARENT** style, **Draw** uses a two-step process to draw a masked image. First, it performs a logical-AND operation on the bits of the image and the bits of the mask. Then it performs a logical-XOR operation on the results of the first operation and the background bits of the destination device context. This process creates transparent areas in the resulting image; that is, each white bit in the mask causes the corresponding bit in the resulting image to be transparent.

Before drawing a masked image on a solid color background, you should use the SetBkColor member function to set the background color of the image list to the same color as the destination. Setting the color eliminates the need to create transparent areas in the image and enables **Draw** to simply copy the image to the destination device context, resulting in a significant increase in performance. To draw the image, specify the **ILD_NORMAL** style when you call **Draw**.

You can set the background color for a masked image list (CImageList) at any time so that it draws correctly on any solid background. Setting the background color to **CLR_NONE** causes images to be drawn transparently by default. To retrieve the background color of an image list, use the GetBkColor member function.

The **ILD_BLEND25** and **ILD_BLEND50** styles dither the image with the system highlight color. These styles are useful if you use a masked image to represent an object that the user can select. For example, you can use the **ILD_BLEND50** style to draw the image when the user selects it.

A nonmasked image is copied to the destination device context using the **SRCCOPY** raster operation. The colors in the image appear the same regardless of the background color of the device context. The drawing styles specified in **Draw** also have no effect on the appearance of a nonmasked image.

In addition to the Draw member function, another function, DrawIndirect, extends the ability to render an image. **DrawIndirect** takes, as a parameter, an IMAGELISTDRAWPARAMS structure. This structure can be used to customize the rendering of the current image, including the use of raster operation (ROP) codes. For more information on ROP codes, see Raster Operation Codes and Bitmaps as Brushes in the *Platform SDK*.

See Also

Reference

Using CImageList

Concepts

Controls (MFC)

Image Overlays in Image Lists

Every image list (CImageList) includes a list of images to use as overlay masks. An "overlay mask" is an image drawn transparently over another image. Any image can be used as an overlay mask. You can specify up to four overlay masks per image list.

You add the index of an image to the list of overlay masks by using the SetOverlayImage member function, the index of an image, and the index of an overlay mask. Note that the indices for the overlay masks are one-based rather than zero-based.

You draw an overlay mask over an image using a single call to **Draw**. The parameters include the index of the image to draw and the index of an overlay mask. You must use the INDEXTOOVERLAYMASK macro to specify the index of the overlay mask. You can also specify an overlay image when calling the DrawIndirect member function.

See Also

Reference

Using CImageList

Concepts

Controls (MFC)

Dragging Images from an Image List

CImageList includes functions for dragging an image on the screen. The dragging functions move an image smoothly, in color, and without any flashing of the cursor. Both masked and unmasked images can be dragged.

The BeginDrag member function begins a drag operation. The parameters include the index of the image to drag and the location of the hot spot within the image. The hot spot is a single pixel that the dragging functions recognize as the exact screen location of the image. Typically, an application sets the hot spot so that it coincides with the hot spot of the mouse cursor. The DragMove member function moves the image to a new location.

The DragEnter member function sets the initial position of the drag image within a window and draws the image at the position. The parameters include a pointer to the window in which to draw the image and a point that specifies the coordinates of the initial position within the window. The coordinates are relative to the window's upper-left corner, not the client area. The same is true for all of the image-dragging functions that take coordinates as parameters. This means you must compensate for the widths of window elements, such as the border, title bar, and menu bar, when specifying the coordinates. If you specify a **NULL** window handle when calling **DragEnter**, the dragging functions draw the image in the device context associated with the desktop window, and the coordinates are relative to the upper-left corner of the screen.

DragEnter locks all other updates to the given window during the drag operation. If you need to do any drawing during a drag operation, such as highlighting the target of a drag-and-drop operation, you can temporarily hide the dragged image by using the DragLeave member function. You can also use the DragShowNoLock member function.

Call EndDrag when you're done dragging the image.

The SetDragCursorImage member function creates a new drag image by combining the given image (typically a mouse cursor image) with the current drag image. Because the dragging functions use the new image during a drag operation, you should use the Windows ShowCursor function to hide the actual mouse cursor after calling **SetDragCursorImage**. Otherwise, the system may appear to have two mouse cursors for the duration of the drag operation.

When an application calls **BeginDrag**, the system creates a temporary, internal image list and copies the specified drag image to the internal list. You can retrieve a pointer to the temporary drag image list by using the GetDragImage member function. The function also retrieves the current drag position and the offset of the drag image relative to the drag position.

See Also

Reference

Using CImageList

Concepts

Controls (MFC)

Image Information in Image Lists

CImageList includes a number of functions that retrieve information from an image list. The GetImageInfo member function fills an **IMAGEINFO** structure with information about a single image, including the handles of the image and mask bitmaps, the number of color planes and bits per pixel, and the bounding rectangle of the image within the image bitmap. You can use this information to directly manipulate the bitmaps for the image.

The GetImageCount member function retrieves the number of images in an image list.

You can create an icon based on an image and mask in an image list by using the ExtractIcon member function. The function returns the handle of the new icon.

See Also

Reference

Using CImageList

Concepts

Controls (MFC)

Using CListCtrl

Use a list control to display any arrangement of icons with labels, as in the Windows Explorer, or columnar lists of text, with or without icons. For a description of the four possible "views" (not to be confused with MFC views) you can have in a list control — icon view, small icon view, list view, and report view — see Views in the CListCtrl class overview.

In some views, users can drag icons to different positions or edit icon labels. For example, see the right-hand pane in Windows Explorer, which uses a list control in a nondialog window. You can experiment with the available views in Explorer's View menu.

For related information, see List View Controls in the *Platform SDK*.

Note
The *Platform SDK* refers to list controls as "list view controls." This usage of "view" does not refer to MFC view classes, particularly **CListView**. For more information, see List Control and List View.

What do you want to know more about?

- List Control and List View
- List Items and Image Lists
- Callback Items and the Callback Mask
- Creating the List Control
- Creating the Image Lists
- Adding Columns to the Control (Report View)
- Adding Items to the Control
- Scrolling, Arranging, Sorting, and Finding in List Controls
- Implementing Working Areas in List Controls
- Processing Notification Messages in List Controls
- Changing List Control Styles
- Virtual List Controls
- Destroying the List Control

See Also
Concepts
Controls (MFC)

List Control and List View

For convenience, MFC encapsulates the list control in two ways. You can use list controls:

- Directly, by embedding a CListCtrl object in a dialog class.
- Indirectly, by using class CListView.

CListView makes it easy to integrate a list control with the MFC document/view architecture, encapsulating the control much as CEditView encapsulates an edit control: the control fills the entire surface area of an MFC view. (The view *is* the control, cast to **CListView**.)

A **CListView** object inherits from CCtrlView and its base classes and adds a member function to retrieve the underlying list control. Use view members to work with the view as a view. Use the GetListCtrl member function to gain access to the list control's member functions. Use these members to:

- Add, delete, or manipulate "items" in the list.
- Set or get list control attributes.

To obtain a reference to the **CListCtrl** underlying a **CListView**, call **GetListCtrl** from your list view class:

```
CListCtrl& ctlList = GetListCtrl();
```

This topic describes both ways to use the list control.

See Also

Reference

Using CListCtrl

Concepts

Controls (MFC)

List Items and Image Lists

An "item" in a list control (CListCtrl) consists of an icon, a label, and possibly other information (in "subitems").

The icons for list control items are contained in image lists. One image list contains full-sized icons used in icon view. A second, optional, image list contains smaller versions of the same icons for use in other views of the control. A third optional list contains "state" images, such as check boxes, for display in front of the small icons in certain views. A fourth optional list contains images that are displayed in individual header items of the list control.

Note

If a list view control is created with the **LVS_SHAREIMAGELISTS** style, you are responsible for destroying the image lists when they are no longer in use. Specify this style if you assign the same image lists to multiple list view controls; otherwise, more than one control might try to destroy the same image list.

For more information about list items, see List View Image Lists and Items and Subitems in the *Platform SDK*. Also see class CImageList in the *MFC Reference* and Using CImageList in this family of articles.

To create a list control, you need to supply image lists to be used when you insert new items into the list. The following example demonstrates this procedure, where `m_pImagelist` is a pointer of type **CImageList** and `m_listctrl` is a **CListCtrl** data member.

```
// create, initialize, and hook up image list
m_pImageList = new CImageList();
ASSERT(m_pImageList != NULL);    // serious allocation failure checking
m_pImageList->Create(32, 32, TRUE,   4, 4);
m_pImageList->Add(pApp->LoadIcon(IDI_ICONLIST1));
m_pImageList->Add(pApp->LoadIcon(IDI_ICONLIST2));
m_listctrl.SetImageList(m_pImageList, LVSIL_NORMAL);
```

However, if you don't plan to display icons in your list view or list control, you don't need image lists.

See Also

Reference

Using CListCtrl

Concepts

Controls (MFC)

Callback Items and the Callback Mask

For each of its items, a list view control typically stores the label text, the image list index of the item's icons, and a set of bit flags for the item's state. You can define individual items as callback items, which are useful if your application already stores some of the information for an item.

You define an item as a callback item by specifying appropriate values for the *pszText* and *iImage* members of the **LV_ITEM** structure (see CListCtrl::GetItem). If the application maintains the item's or subitem's text, specify the **LPSTR_TEXTCALLBACK** value for the *pszText* member. If the application keeps track of the icon for the item, specify the **I_IMAGECALLBACK** value for the *iImage* member.

In addition to defining callback items, you can also modify the control's callback mask. This mask is a set of bit flags that specify the item states for which the application, rather than the control, stores the current data. The callback mask applies to all of the control's items, unlike the callback item designation, which applies to a specific item. The callback mask is zero by default, meaning that the control tracks all item states. To change this default behavior, initialize the mask to any combination of the following values:

- **LVIS_CUT** The item is marked for a cut-and-paste operation.
- **LVIS_DROPHILITED** The item is highlighted as a drag-and-drop target.
- **LVIS_FOCUSED** The item has the focus.
- **LVIS_SELECTED** The item is selected.
- **LVIS_OVERLAYMASK** The application stores the image list index of the current overlay image for each item.
- **LVIS_STATEIMAGEMASK** The application stores the image list index of the current state image for each item.

For further information on retrieving and setting this mask, see CListCtrl::GetCallbackMask and CListCtrl::SetCallbackMask.

See Also

Reference

Using CListCtrl

Concepts

Controls (MFC)

Creating the List Control

How the list control (CListCtrl) is created depends on whether you're using the control directly or using class CListView instead. If you use **CListView**, the framework constructs the view as part of its document/view creation sequence. Creating the list view creates the list control as well (the two are the same thing). The control is created in the view's OnCreate handler function. In this case, the control is ready for you to add items, via a call to GetListCtrl.

To use CListCtrl directly in a dialog box

1. In the dialog editor, add a List Control to your dialog template resource. Specify its control ID.
2. Use the Add Member Variable Wizard to add a member variable of type **CListCtrl** with the Control property. You can use this member to call **CListCtrl** member functions.
3. Use the Properties window to map handler functions in the dialog class for any list control notification messages you need to handle (see Mapping Messages to Functions).
4. In OnInitDialog, set the styles for the **CListCtrl**. See Changing List Control Styles. This determines the kind of "view" you get in the control, although you can change the view later.

To use CListCtrl in a nondialog window

1. Define the control in the view or window class.
2. Call the control's Create member function, possibly in OnInitialUpdate, possibly as early as the parent window's OnCreate handler function (if you're subclassing the control). Set the styles for the control.

See Also

Reference

Using CListCtrl

Concepts

Controls (MFC)

Creating the Image Lists

Creating image lists is the same whether you use CListView or CListCtrl.

Note
You only need image lists if your list control includes the **LVS_ICON** style.

Use class **CImageList** to create one or more image lists (for full-size icons, small icons, and states). See CImageList, and see List View Image Lists in the *Platform SDK*.

Call CListCtrl::SetImageList for each image list; pass a pointer to the appropriate **CImageList** object.

See Also
Reference
Using CListCtrl
Concepts
Controls (MFC)

Adding Columns to the Control (Report View)

Note
The following procedure applies to either a CListView or CListCtrl object.

When a list control is in report view, columns are displayed, providing a method of organizing the various subitems of each list control item. This organization is implemented with a one-to-one correspondence between a column in the list control and the associated subitem of the list control item. For more information on subitems, see Adding Items to the Control. An example of a list control in report view is provided by the Details view in Windows 95 and Windows 98 Explorer. The first column lists folder, file icons, and labels. Other columns list file size, file type, date last modified, and so on.

Even though columns can be added to a list control at any time, the columns are visible only when the control has the **LVS_REPORT** style bit turned on.

Each column has an associated header item (see CHeaderCtrl) object that labels the column and allows users to resize the column.

If your list control supports a report view, you need to add a column for each possible subitem in a list control item. Add a column by preparing an LV_COLUMN structure and then making a call to InsertColumn. After adding the necessary columns (sometimes referred to as header items), you can reorder them using member functions and styles belonging to the embedded header control. For more information, see Ordering Items in the Header Control.

Note
If the list control is created with the **LVS_NOCOLUMNHEADER** style, any attempt to insert columns will be ignored.

See Also

Reference

Using CListCtrl

Concepts

Controls (MFC)

Adding Items to the Control

To add items to the list control (CListCtrl), call one of several versions of the InsertItem member function, depending on what information you have. One version takes a LV_ITEM structure that you prepare. Because the **LV_ITEM** structure contains numerous members, you have greater control over the attributes of the list control item.

Two important members (in regard to the report view) of the **LV_ITEM** structure are the **iItem** and **iSubItem** members. The **iItem** member is the zero-based index of the item the structure is referencing and the **iSubItem** member is the one-based index of a subitem, or zero if the structure contains information about an item. With these two members you determine, per item, the type and value of subitem information that is displayed when the list control is in report view. For more information, see CListCtrl::SetItem.

Additional members specify the item's text, icon, state, and item data. "Item data" is an application-defined value associated with a list view item. For more information about the **LV_ITEM** structure, see CListCtrl::GetItem.

Other versions of **InsertItem** take one or more separate values, corresponding to members in the **LV_ITEM** structure, allowing you to initialize only those members you want to support. Generally, the list control manages storage for list items, but you can store some of the information in your application instead, using "callback items." For more information, see Callback Items and the Callback Mask in this topic and Callback Items and the Callback Mask in the *Platform SDK*.

For more information, see Items and Subitems

See Also
Reference
Using CListCtrl
Concepts
Controls (MFC)

Scrolling, Arranging, Sorting, and Finding in List Controls

List controls (CListCtrl) are scrollable by default. For more information, see Scroll Position in the *Platform SDK* and the Scroll member function.

You can call **CListCtrl** member functions to arrange list items in the control, sort items, and find particular items. For more information, see Arranging, Sorting, and Finding in the *Platform SDK* and the CListCtrl members Arrange, SortItems, and FindItem.

See Also

Reference

Using CListCtrl

Concepts

Controls (MFC)

Implementing Working Areas in List Controls

By default, a list control arranges all items in a standard grid fashion. However, another method is supported, working areas, that arranges the list items into rectangular groups. For an image of a list control that implements working areas, see Using List-View Controls in the *Platform SDK*.

Note
Working areas are visible only when the list control is in icon or small icon mode. However, any current working areas are m aintained if the view is switched to the report or list mode.

Working areas can be used to display an empty border (on the left, top and/or right of the items), or cause a horizontal scroll bar to be displayed when there normally wouldn't be one. Another common usage is to create multiple working areas to which items can be moved or dropped. With this method, you could create areas in a single view that have different meanings. The user could then categorize the items by placing them in a different area. An example of this would be a view of a file system that has an area for read/write files and another area for read-only files. If a file item were moved into the read-only area, it would automatically become read-only. Moving a file from the read-only area into the read/write area would make the file read/write.

CListCtrl provides several member functions for creating and managing working areas in your list control. GetWorkAreas and SetWorkAreas retrieve and set an array of **CRect** objects (or **RECT** structures), which store the currently implemented working areas for your list control. In addition, GetNumberOfWorkAreas retrieves the current number of working areas for your list control (by default, zero).

Items and Working Areas

When a working area is created, items that lie within the working area become members of it. Similarly, if an item is moved into a working area, it becomes a member of the working area to which it was moved. If an item does not lie within any working area, it automatically becomes a member of the first (index 0) working area. If you want to create an item and have it placed within a specific working area, you will need to create the item and then move it into the desired working area with a call to SetItemPosition. The second example below demonstrates this technique.

The following example implements four working areas (`rcWorkAreas`), of equal size with a 10-pixel-wide border around each working area, in a list control (`m_listctrl`).

```
CRect curRect;
CSize size;

size= m_listctrl.ApproximateViewRect();
size.cx+= 100;
size.cy+= 100;

CRect rcWorkAreas[4];

rcWorkAreas[0].SetRect(0, 0, (size.cx / 2) - 5, (size.cy / 2) - 5);
rcWorkAreas[1].SetRect((size.cx / 2) + 5, 0, size.cx, (size.cy / 2) - 5);
rcWorkAreas[2].SetRect(0, (size.cy / 2) + 5, (size.cx / 2) - 5, size.cy);
rcWorkAreas[3].SetRect((size.cx / 2) + 5, (size.cy / 2) + 5, size.cx, size.cy);

//set work areas
m_listctrl.SetWorkAreas(4, rcWorkAreas);
```

The call to ApproximateViewRect was made to get an estimate of the total area required to display all items in one region. This estimate is then divided into four regions and padded with a 5-pixel-wide border.

The next example assigns the existing list items to each group (`rcWorkAreas`) and refreshes the control view (`m_listctrl`) to complete the effect.

```
// set insertion points for each work area
CPoint     rgptWork[4];
for (int i = 0; i < 4; i++)
{
     rgptWork[i].x = rcWorkAreas[i].left + 10;
```

```
        rgptWork[i].y = rcWorkAreas[i].top + 10;
    }
    // now move all the items to the different quadrants
    for (i = 0; i < 20; i++)
        m_listctrl.SetItemPosition(i, rgptWork[i % 4]);

    // force the control to rearrange the shuffled items
    m_listctrl.Arrange(LVA_DEFAULT);
```

See Also

Reference

Using CListCtrl

Concepts

Controls (MFC)

Processing Notification Messages in List Controls

As users click column headers, drag icons, edit labels, and so on, the list control (CListCtrl) sends notification messages to its parent window. Handle these messages if you want to do something in response. For example, when the user clicks a column header, you might want to sort the items based on the contents of the clicked column, as in Microsoft Outlook.

Process **WM_NOTIFY** messages from the list control in your view or dialog class. Use the Properties window to create an OnChildNotify handler function with a switch statement based on which notification message is being handled.

For a list of the notifications a list control can send to its parent window, see List View Control Reference in the *Platform SDK*.

See Also

Reference

Using CListCtrl

Concepts

Controls (MFC)

Changing List Control Styles

You can change the window style of a list control (CListCtrl) at any time after you create it. By changing the window style, you change the kind of view the control uses. For example, to emulate the Explorer, you might supply menu items or toolbar buttons for switching the control between different views: icon view, list view, and so on.

For example, when the user selects your menu item, you could make a call to GetWindowLong to retrieve the current style of the control and then call SetWindowLong to reset the style. For more information, see Using List View Controls in the *Platform SDK*.

Available styles are listed in Create. The styles **LVS_ICON**, **LVS_SMALLICON**, **LVS_LIST**, and **LVS_REPORT** designate the four list control views.

Extended Styles

In addition to the standard styles for a list control, there is another set, referred to as extended styles. These styles, discussed in Extended List View Styles in the *Platform SDK*, provide a variety of useful features that customize the behavior of your list control. To implement the behavior of a certain style (such as hover selection), make a call to CListCtrl::SetExtendedStyle, passing the needed style. The following example demonstrates the function call:

```
m_myListCtrl.SetExtendedStyle(LVS_EX_TRACKSELECT | LVS_EX_ONECLICKACTIVATE);
```

Note

For hover selection to work, you must also have either **LVS_EX_ONECLICKACTIVATE** or **LVS_EX_TWOCLICKACTIVATE** turned on.

See Also

Reference

Using CListCtrl

Concepts

Controls (MFC)

Virtual List Controls

A virtual list control is a list view control that has the **LVS_OWNERDATA** style. This style enables the control to support an item count up to a **DWORD** (the default item count only extends to an **int**). However, the biggest advantage provided by this style is the ability to only have a subset of data items in memory at any one time. This allows the virtual list view control to lend itself for use with large databases of information, where specific methods of accessing data are already in place.

Note
In addition to providing virtual list functionality in **CListCtrl**, MFC also provides the same functionality in the CListView class.

There are some compatibility issues you should be aware of when developing virtual list controls. For more information, see the Compatibility Issues section of the List-View Controls topic in the *Platform SDK*.

Handling the LVN_GETDISPINFO Notification

Virtual list controls maintain very little item information. Except for the item selection and focus information, all item information is managed by the owner of the control. Information is requested by the framework via a **LVN_GETDISPINFO** notification message. To provide the requested information, the owner of the virtual list control (or the control itself) must handle this notification. This can easily be done using the Properties window (see Mapping Messages to Functions). The resultant code should look something like the following example (where `CMyListCtrl` is the virtual list control object and the control is handling the notification):

```
BEGIN_MESSAGE_MAP(CMyListCtrl, CListCtrl)
   ON_NOTIFY_REFLECT(LVN_GETDISPINFO, OnGetdispinfo)
END_MESSAGE_MAP()
```

In the handler for the **LVN_GETDISPINFO** notification message, you must check to see what type of information is being requested. The possible values are:

- **LVIF_TEXT** The *pszText* member must be filled in.
- **LVIF_IMAGE** The *iImage* member must be filled in.
- **LVIF_INDENT** The *iIndent* member must be filled in.
- **LVIF_PARAM** The *lParam* member must be filled in.
- **LVIF_STATE** The *state* member must be filled in.

You should then supply whatever information is requested back to the framework.

The following example (taken from the body of the notification handler for the list control object) demonstrates one possible method by supplying information for the text buffers and image of an item:

```
LV_DISPINFO* pDispInfo = (LV_DISPINFO*)pNMHDR;
LV_ITEM* pItem= &(pDispInfo)->item;

int iItemIndx= pItem->iItem;

if (pItem->mask & LVIF_TEXT) //valid text buffer?
{
    switch(pItem->iSubItem){
        case 0: //fill in main text
            lstrcpy(pItem->pszText,
                m_Items[iItemIndx].m_strItemText);
            break;
        case 1: //fill in sub item 1 text
            lstrcpy(pItem->pszText,
                m_Items[iItemIndx].m_strSubItem1Text);
            break;
        case 2: //fill in sub item 2 text
            lstrcpy(pItem->pszText,
                m_Items[iItemIndx].m_strSubItem2Text);
```

```
            break;
    }
}

if pItem->mask & LVIF_IMAGE) //valid image?
        pItem->iImage=
            m_Items[iItemIndx].m_iImageIndex;
```

Caching and Virtual List Controls

Because this type of list control is intended for large data sets, it is recommended that you cache requested item data to improve retrieval performance. The framework provides a cache-hinting mechanism to assist in optimizing the cache by sending an **LVN_ODCACHEHINT** notification message. However, you must use a slightly different method to handle this notification. Using the Properties window, override the **OnChildNotify** function of your list control object. In the case of this example, `CMyListCtrl`.

Inside the body of the handler, check for the **LVN_ODCACHEHINT** message and, if found, prepare your cache.

The following example (taken from the body of the `OnChildNotify` function) performs this check and calls the `PrepCache` member function of the `CMyListCtrl` class.

```
NMLVCACHEHINT* pcachehint=NULL;

if (message == WM_NOTIFY)
    {
        NMHDR* phdr = (NMHDR*)lParam;

        switch(phdr->code)
        {
        case LVN_ODCACHEHINT:
            pcachehint= (NMLVCACHEHINT*) phdr;
// Load the cache with the recommended range.
            PrepCache(pcachehint->iFrom, pcachehint->iTo);
            break;
        default:
            return CListCtrl::OnChildNotify(message, wParam, lParam, pLResult);
        }
        return FALSE;
    }
    else
        return CListCtrl::OnChildNotify(message, wParam, lParam, pLResult);
```

Notice that the notification is passed on to the base class (**CListCtrl**) if the message type is not **LVN_ODCACHEHINT**. For more information on preparing and maintaining a cache, see the Cache Management section of the List-View Controls topic in the *Platform SDK*.

Finding Specific Items

The **LVN_ODFINDITEM** notification message is sent by the virtual list control when a particular list control item needs to be found. The notification message is sent when the list view control receives quick key access or when it receives an **LVM_FINDITEM** message. Search information is sent in the form of an **LVFINDINFO** structure, which is a member of the **NMLVFINDITEM** structure. Handle this message by overriding the **OnChildNotify** function of your list control object and inside the body of the handler, check for the **LVN_ODFINDITEM** message. If found, perform the appropriate action.

You should be prepared to search for an item that matches the information given by the list view control. You should return the index of the item if successful, or -1 if no matching item is found.

See Also

Reference

Using CListCtrl

Concepts

Controls (MFC)

Destroying the List Control

If you embed your CListCtrl object as a data member of a view or dialog class, it is destroyed when its owner is destroyed. If you use a CListView, the framework destroys the control when it destroys the view.

If you arrange for some of your list data to be stored in the application rather than the list control, you will need to arrange for its deallocation. For more information, see Callback Items and the Callback Mask in the *Platform SDK*.

In addition, you are responsible for deallocating any image lists you created and associated with the list control object.

See Also
Reference
Using CListCtrl
Concepts
Controls (MFC)

Using CMonthCalCtrl

The month calendar control (CMonthCalCtrl) implements a calendar-like user interface. This provides the user with a very intuitive and recognizable method of entering or selecting a date. The control also provides the application with the means to obtain and set the date information in the control using existing data types. By default, the month calendar control displays the current day and month. However, the user is able to scroll to the previous and next months and select a specific month and/or year.

Note
To use the **CMonthCalCtrl** class in your project, you must include AFXDTCTL.H, usually in STDAFX.H.

What do you want to know more about?

- Creating the Month Calendar Control
- Month Calendar Control Examples
- Processing Notification Messages in Month Calendar Controls
- Setting the Day State of a Month Calendar Control

See Also

Concepts

Controls (MFC)

Creating the Month Calendar Control

How the month calendar control is created depends on whether you are using the control in a dialog box or creating it in a nondialog window.

To use CMonthCalCtrl directly in a dialog box

1. In the dialog editor, add a Month Calendar Control to your dialog template resource. Specify its control ID.
2. Specify any styles required, using the Properties dialog box of the month calendar control.
3. Use the Add Member Variable Wizard to add a member variable of type CMonthCalCtrl with the Control property. You can use this member to call **CMonthCalCtrl** member functions.
4. Use the Properties window to map handler functions in the dialog class for any month calendar control notification messages you need to handle (see Mapping Messages to Functions).
5. In OnInitDialog, set any additional styles for the **CMonthCalCtrl** object.

To use CMonthCalCtrl in a nondialog window

1. Define the control in the view or window class.
2. Call the control's Create member function, possibly in OnInitialUpdate, possibly as early as the parent window's OnCreate handler function (if you're subclassing the control). Set the styles for the control.

See Also

Reference

Using CMonthCalCtrl

Concepts

Controls (MFC)

Month Calendar Control Examples

The CMNCTRL1 sample application demonstrates the various attributes of the **CMonthCalCtrl** class. The control, found on a separate tab in the sample, demonstrates basic functionality and allows the user to dynamically modify certain attributes.

See Also

Reference

Using CMonthCalCtrl

Concepts

Controls (MFC)

Processing Notification Messages in Month Calendar Controls

As users interact with the month calendar control (selecting dates and/or viewing a different month), the control (**CMonthCalCtrl**) sends notification messages to its parent window, usually a view or dialog object. Handle these messages if you want to do something in response. For example, when the user selects a new month to view, you could provide a set of dates that should be emphasized.

Use the Properties window to add notification handlers to the parent class for those messages you want to implement.

The following list describes the various notifications sent by the month calendar control.

- **MCN_GETDAYSTATE** Requests information about which days should be displayed in bold. For information on handling this notification, see Setting the Day State of a Month Calendar Control.
- **MCN_SELCHANGE** Notifies the parent that the selected date or range of the date has changed.
- **MCN_SELECT** Notifies the parent that an explicit date selection has been made.

See Also
Reference
Using CMonthCalCtrl
Concepts
Controls (MFC)

Setting the Day State of a Month Calendar Control

One of the attributes of a month calendar control is the ability to store information, referred to as the day state of the control, for each day of the month. This information is used to emphasize certain dates for the month currently displayed.

Note
The **CMonthCalCtrl** object must have the **MCS_DAYSTATE** style to display day state information.

Day state information is expressed as a 32-bit data type, **MONTHDAYSTATE**. Each bit in a **MONTHDAYSTATE** bit field (1 through 31) represents the state of a day in a month. If a bit is on, the corresponding day will be displayed in bold; otherwise it will be displayed with no emphasis.

There are two methods for setting the day state of the month calendar control: explicitly with a call to CMonthCalCtrl::SetDayState or by handling the **MCN_GETDAYSTATE** notification message.

Handling the MCN_GETDAYSTATE Notification Message

The **MCN_GETDAYSTATE** message is sent by the control to determine how the days within the visible months should be displayed.

Note
Because the control caches the previous and following months, in respect to the visible month, you will receive this notificati on every time a new month is chosen.

To properly handle this message, you must determine how many months day state information is being requested for, initialize an array of **MONTHDAYSTATE** structures with the proper values, and initialize the related structure member with the new information. The following procedure, detailing the necessary steps, assumes that you have a **CMonthCalCtrl** object called `m_monthcal` and an array of **MONTHDAYSTATE** objects, `mdState`.

To handle the MCN_GETDAYSTATE notification message

1. Using the Properties window, add a notification handler for the **MCN_GETDAYSTATE** message to the `m_monthcal` object (see Mapping Messages to Functions).

2. In the body of the handler, add the following code:

```
NMDAYSTATE* pDayState= (NMDAYSTATE *)pNMHDR;

#define BOLDDAY(ds,iDay) if(iDay>0 && iDay<32)\
                            (ds)|=(0x00000001<<(iDay-1))
MONTHDAYSTATE mdState[3];
INT i, iMax;
```

   The example converts the **pNMHDR** pointer to the proper type and defines a helpful macro that sets specific bits in the **MONTHDAYSTATE** bitfield.

3. After adding the previous code example, add the following code:

```
iMax=pDayState->cDayState;

for(i=0;i<iMax;i++)
{
    mdState[i] = (MONTHDAYSTATE)0;
    BOLDDAY(mdState[i],15);
 }
pDayState->prgDayState = mdState;
```

The example first determines how many months of information are being requested (`pDayState->cDayState`). For each month, the current bitfield (`mdState`) is initialized to zero and then the needed dates are set (in this case, the 15th of each month). Finally, the `pDayState->prgDayState` member is set, completing the **MCN_DAYSTATE** handler.

See Also

Reference

Using CMonthCalCtrl

Concepts

Controls (MFC)

Using CProgressCtrl

You can use the progress control to indicate the progress of a lengthy operation. It is a rectangle that is gradually filled with the system highlight color as the operation progresses.

The progress control is represented in MFC by class CProgressCtrl.

When you initially create the progress control, you specify its size and position, parent window (usually a dialog box), and ID. By using the *dwStyle* parameter, you can also specify various window styles for the control and styles for how it fills.

What do you want to know more about?

- Styles for the Progress Control
- Settings for the Progress Control
- Manipulating the Progress Control

See Also
Concepts
Controls (MFC)

Styles for the Progress Control

When you initially create the progress control (CProgressCtrl::Create), use the *dwStyle* parameter to specify the desired window styles for your progress control. The following list details the applicable window styles. The control ignores any window style other than the ones listed here. You should always create the control as a child window, usually of a dialog box parent.

Window style	Effect
WS_BORDER	Creates a border around the window.
WS_CHILD	Creates a child window (should always be used for **CProgressCtrl**).
WS_CLIPCHILDREN	Excludes the area occupied by child windows when you draw within the parent window. Used when you create the parent window.
WS_CLIPSIBLINGS	Clips child windows relative to each other.
WS_DISABLED	Creates a window that is initially disabled.
WS_VISIBLE	Creates a window that is initially visible.
WS_TABSTOP	Specifies that the control can receive focus when the user presses the TAB key to move to it.

In addition, you can specify two styles that apply only to the progress control, **PBS_VERTICAL** and **PBS_SMOOTH**.

Use **PBS_VERTICAL** to orient the control vertically, rather than horizontally. Use **PBS_SMOOTH** to fill the control completely, rather than displaying small delineated squares that fill the control incrementally.

Without **PBS_SMOOTH** style:

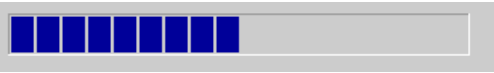

With **PBS_SMOOTH** and **PBS_VERTICAL** styles:

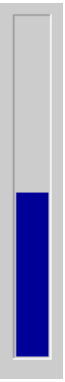

For more information, see Window Styles in the *MFC Reference*.

See Also

Reference

Using CProgressCtrl

Settings for the Progress Control

The basic settings for the progress control (CProgressCtrl) are the range and current position. The range represents the entire duration of the operation. The current position represents the progress that your application has made toward completing the operation. Any changes to the range or position cause the progress control to redraw itself.

By default, the range is set to 0 – 100, and the initial position is set to 0. To retrieve the current range settings for the progress control, use the GetRange member function. To change the range, use the SetRange member function.

To set the position, use SetPos. To retrieve the current position without specifying a new value, use GetPos. For example, you might want to simply query on the status of the current operation.

To step the current position of the progress control, use StepIt. To set the amount of each step, use SetStep

See Also

Reference

Using CProgressCtrl

Concepts

Controls (MFC)

Manipulating the Progress Control

There are three ways to change the current position of a progress control (CProgressCtrl).

- The position can be changed by a preset increment amount.
- The position can be changed by an arbitrary amount.
- The position can be changed to a specific value.

To change the position by a preset amount

1. Use the SetStep member function to set the increment amount. By default, this value is 10. This value is typically set as one of the initial settings for the control. The step value can be negative.
2. Use the StepIt member function to increment the position. This causes the control to redraw itself.

Note
StepIt will cause the position to wrap. For example, given a range of 1 –100, a step of 20, and a position of 90, **StepIt** will set the position to 10.

To change the position by an arbitrary amount

- Use the OffsetPos member function to change the position. **OffsetPos** will accept negative values.

Note
OffsetPos, unlike **StepIt**, will not wrap the position. The new position is adjusted to remain within the range.

To change the position to a specific value

- Use the SetPos member function to set the position to a specific value. If necessary, the new position is adjusted to be within the range.

Typically, the progress control is used solely for output. To get the current position without specifying a new value, use GetPos.

See Also

Reference

Using CProgressCtrl

Concepts

Controls (MFC)

Using CReBarCtrl

A rebar control acts as a container for child windows. These child windows, often other controls, are assigned to a rebar control band. A rebar control can contain one or more bands, with each band having any combination of a gripper bar, a bitmap, a text label, and a child window. However, bands cannot contain more than one child window.

The following illustration shows a rebar control that has two bands. One contains a gripper bar, a text label ("Address"), and a combo box child window. The other band contains a gripper bar, a text label, and a flat toolbar (implemented with a child window).

What do you want to know more about?

- CReBar vs. CReBarCtrl
- Creating a Rebar Control
- Rebar Controls and Bands
- Using an Image List with a Rebar Control
- Using a Dialog Bar with a Rebar Control
- Processing Notification Messages in a Rebar Control

See Also

Concepts

Controls (MFC)

CReBar vs. CReBarCtrl

MFC provides two classes to create rebars: CReBar and CReBarCtrl (which wraps the Windows common control API). **CReBar** provides all of the functionality of the rebar common control, and it handles many of the required common control settings and structures for you.

CReBarCtrl is a wrapper class for the Win32 rebar control, and therefore may be easier to implement if you do not intend to integrate the rebar into the MFC architecture. If you plan to use **CReBarCtrl** and integrate the rebar into the MFC architecture, you must take additional care to communicate rebar control manipulations to MFC. This communication is not difficult; however, it is additional work that is unneeded when you use **CReBar**.

Visual C++ provides two ways to take advantage of the rebar common control.

- Create the rebar using **CReBar**, and then call CReBar::GetReBarCtrl to get access to the **CReBarCtrl** member functions.

Note
CReBar::GetReBarCtrl is an inline member function that casts the **this** pointer of the rebar object. This means that, at run time, the function call has no overhead.

- Create the rebar using CReBarCtrl's constructor.

Either method will give you access to the member functions of the rebar control. When you call **CReBar::GetReBarCtrl**, it returns a reference to a **CReBarCtrl** object so you can use either set of member functions. See CReBar for information on constructing and creating a rebar using **CReBar**.

See Also

Reference

Using CReBarCtrl

Concepts

Controls (MFC)

Creating a Rebar Control

CReBarCtrl objects should be created before the parent object is visible. This minimizes the possibilities of painting problems.

For instance, rebar controls (used in frame window objects) are commonly used as parent windows for toolbar controls. Therefore, the parent of the rebar control is the frame window object. Because the frame window object is the parent, the **OnCreate** member function (of the parent) is an excellent place to create the rebar control.

To use a **CReBarCtrl** object, you will typically follow these steps:

To use a CReBarCtrl object

1. Construct the CReBarCtrl object.
2. Call Create to create the Windows rebar common control and attach it to the **CReBarCtrl** object, specifying any desired styles.
3. Load a bitmap, with a call to CBitmap::LoadBitmap, to be used as the background of the rebar control object.
4. Create and initialize any child window objects (toolbars, dialog controls, and so on) that will be contained by the rebar control object.
5. Initialize a **REBARBANDINFO** structure with the necessary information for the band about to be inserted.
6. Call InsertBand to insert existing child windows (such as `m_wndReToolBar`) into the new rebar control. For more information on inserting bands into an existing rebar control, see Rebar Controls and Bands.

See Also

Reference

Using CReBarCtrl

Concepts

Controls (MFC)

Rebar Controls and Bands

The main purpose of a rebar control is to act as a container for child windows, common dialog controls, menus, toolbars, and so on. This containment is supported by the concept of a "band." Each rebar band can contain any combination of a gripper bar, a bitmap, a text label, and a child window.

Class **CReBarCtrl** has many member functions that you can use to retrieve, and manipulate, information for a specific rebar band:

- GetBandCount Retrieves the number of current bands in the rebar control.
- GetBandInfo Initializes a **REBARBANDINFO** structure with information from the specified band. There is a corresponding SetBandInfo member function.
- GetRect Retrieves the bounding rectangle of a specified band.
- GetRowCount Retrieves the number of band rows in a rebar control.
- IDToIndex Retrieves the index of a specified band.
- GetBandBorders Retrieves the borders of a band.

In addition to manipulation, several member functions are provided that allow you to operate on specific rebar bands.

InsertBand and DeleteBand add and remove rebar bands. MinimizeBand and MaximizeBand affect the current size of a specific rebar band. MoveBand changes the index of a specific rebar band. ShowBand shows or hides a rebar band from the user.

The following example demonstrates adding a toolbar band (`m_wndToolBar`) to an existing rebar control (`m_wndReBar`). The band is described by initializing the `rbi` structure and then calling the **InsertBand** member function:

```
//load bitmap for toolbar background
m_bmap.LoadBitmap(IDB_BITMAP);

//create a toolbar band
m_wndToolBar.Create(this, TBSTYLE_TRANSPARENT | TBSTYLE_FLAT);
m_wndToolBar.LoadToolBar(IDR_MAINFRAME);

REBARBANDINFO rbi;
rbi.cbSize= sizeof(REBARBANDINFO);
rbi.fMask= RBBIM_BACKGROUND | RBBIM_CHILD | RBBIM_CHILDSIZE | RBBIM_STYLE | RBBIM_TEXT;
rbi.fStyle= RBBS_GRIPPERALWAYS;
rbi.cxMinChild= 300;
rbi.cyMinChild= 30;
rbi.lpText= "Band #1";
rbi.cch= 7;
rbi.cx= 300;
rbi.hbmBack= (HBITMAP)m_bmap;
rbi.hwndChild= (HWND)m_wndToolBar;
m_wndReBar.InsertBand(-1, &rbi);
```

See Also

Reference

Using CReBarCtrl

Concepts

Controls (MFC)

Using an Image List with a Rebar Control

Each rebar band can contain, among other things, an image from an associated image list. The following procedure details the necessary steps for displaying an image in a rebar band.

To display images in a rebar band

1. Attach an image list to your rebar control object by making a call to SetImageList, passing a pointer to an existing image list.
2. Modify the **REBARBANDINFO** structure to assign an image to a rebar band:
   - Set the **fMask** member to **RBBIM_IMAGE**, using the bitwise OR operator to include additional flags as necessary.
   - Set the **iImage** member to the image list index of the image to be displayed.
3. Initialize any remaining data members, such as the size, text, and handle of the contained child window, with the necessary information.
4. Insert the new band (with the image) with a call to CReBarCtrl::InsertBand, passing the **REBARBANDINFO** structure.

The following example assumes that an existing image list object with two images was attached to the rebar control object (`m_wndReBar`). A new rebar band (defined by `rbi`), containing the first image, is added with a call to **InsertBand**:

```
REBARBANDINFO rbi;
rbi.cbSize= sizeof(REBARBANDINFO);
rbi.fMask= RBBIM_BACKGROUND | RBBIM_CHILD | RBBIM_IMAGE | RBBIM_CHILDSIZE | RBBIM_STYLE | R
BBIM_TEXT;
rbi.fStyle= RBBS_GRIPPERALWAYS;
rbi.cxMinChild= 200;
rbi.cyMinChild= 50;
rbi.lpText= "Band #1";
rbi.cch= 7;
rbi.cx= 300;
rbi.hbmBack= (HBITMAP)m_bmap3;
rbi.iImage= 0;
rbi.hwndChild= (HWND)m_wndToolBar;
BOOL bRes= m_wndReBar.InsertBand(-1, &rbi);
```

See Also

Reference

Using CReBarCtrl

Concepts

Controls (MFC)

Using a Dialog Bar with a Rebar Control

As mentioned in Rebar Controls and Bands, each band can contain only one child window (or control). This might be a limitation if you want to have more than one child window per band. A convenient workaround is to create a dialog bar resource with multiple controls and then add a rebar band (containing the dialog bar) to the rebar control.

Normally, if you wanted the dialog bar band to appear transparent, you would set the **WS_EX_TRANSPARENT** extended style for the dialog bar object. However, because **WS_EX_TRANSPARENT** has some issues with properly painting the background of a dialog bar, you will need to do a little extra work to achieve the desired effect.

The following procedure details the steps necessary to achieve transparency without using the **WS_EX_TRANSPARENT** extended style.

To implement a transparent dialog bar in a rebar band

1. Using the Add Class dialog box, add a new class (for example, `CMyDlgBar`) that implements your dialog bar object.
2. Add a handler for the **WM_ERASEBKGND** message.
3. In the new handler, modify the existing code to match the following example:

```
BOOL CMyDlgBar::OnEraseBkgnd( CDC* pDC )
{
    CWnd* pParent = GetParent();
    ASSERT_VALID(pParent);
    CPoint pt(0, 0);
    MapWindowPoints(pParent, &pt, 1);
    pt = pDC->OffsetWindowOrg(pt.x, pt.y);
    LRESULT lResult = pParent->SendMessage(WM_ERASEBKGND,
        (WPARAM)pDC->m_hDC, 0L);
    pDC->SetWindowOrg(pt.x, pt.y);
    return lResult;
}
```

4. Add a handler for the **WM_MOVE** message.
5. In the new handler, modify the existing code to match the following example:

```
BOOL CMyDlgBar::OnMove( int cx, int cy )
{
    Invalidate();
}
```

The new handlers simulate the transparency of the dialog bar by forwarding the **WM_ERASEBKGND** message to the parent window and forcing a repaint every time the dialog bar object is moved.

See Also

Reference

Using CReBarCtrl

Concepts

Controls (MFC)

Processing Notification Messages in a Rebar Control

In the parent class of the rebar control, create an **OnChildNotify** handler function with a switch statement for any rebar-control (**CReBarCtrl**) notification messages you want to handle. Notifications are sent to the parent window when the user drags objects over the rebar control, changes the layout of the rebar bands, deletes bands from the rebar control, and so on.

The following notification messages can be sent by the rebar control object:

- **RBN_AUTOSIZE** Sent by a rebar control (created with the **RBS_AUTOSIZE** style) when the rebar automatically resizes itself.
- **RBN_BEGINDRAG** Sent by a rebar control when the user begins dragging a band.
- **RBN_CHILDSIZE** Sent by a rebar control when a band's child window is resized.
- **RBN_DELETEDBAND** Sent by a rebar control after a band has been deleted.
- **RBN_DELETINGBAND** Sent by a rebar control when a band is about to be deleted.
- **RBN_ENDDRAG** Sent by a rebar control when the user stops dragging a band.
- **RBN_GETOBJECT** Sent by a rebar control (created with the **RBS_REGISTERDROP** style) when an object is dragged over a band in the control.
- **RBN_HEIGHTCHANGE** Sent by a rebar control when its height has changed.
- **RBN_LAYOUTCHANGED** Sent by a rebar control when the user changes the layout of the control's bands.

For more information on these notifications, see Rebar Control Reference in the *Platform SDK*.

See Also
Reference
Using CReBarCtrl
Concepts
Controls (MFC)

Using CRichEditCtrl

A rich edit control is a window in which the user can enter and edit text. The text can be assigned character and paragraph formatting, and can include embedded OLE objects. The rich edit control is represented in MFC by the CRichEditCtrl class.

What do you want to know more about?

- Overview of the Rich Edit Control
- Classes Related to Rich Edit Controls
- Rich Edit Control Examples
- Character Formatting in Rich Edit Controls
- Paragraph Formatting in Rich Edit Controls
- Current Selection in a Rich Edit Control
- Word Breaks in a Rich Edit Control
- Clipboard Operations in Rich Edit Controls
- Stream Operations in Rich Edit Controls
- Printing in Rich Edit Controls
- Bottomless Rich Edit Controls
- Notifications from a Rich Edit Control

See Also
Concepts
Controls (MFC)

Overview of the Rich Edit Control

Important
If you are using a rich edit control in a dialog box (regardless of whether your application is SDI, MDI, or dialog-based), you must call AfxInitRichEdit once before the dialog box is displayed. A typical place to call this function is in your program's **InitInstance** member function. You do not need to call it for each time you display the dialog box, only the first time. You do not have to call **AfxInitRichEdit** if you are working with **CRichEditView**.

Rich edit controls (CRichEditCtrl) provide a programming interface for formatting text. However, an application must implement any user interface components necessary to make formatting operations available to the user. That is, the rich edit control supports changing the character or paragraph attributes of the selected text. Some examples of character attributes are bold, italics, font family, and point size. Examples of paragraph attributes include alignment, margins, and tab stops. However, it is up to you to provide the user interface, whether that is toolbar buttons, menu items, or a format character dialog box. There are also functions to query the rich edit control for the attributes of the current selection. Use these functions to display the current settings for the attributes, for example, setting a check mark on the command UI if the selection has the bold character formatting attribute.

For more information on character and paragraph formatting, see Character Formatting and Paragraph Formatting later in this topic.

Rich edit controls support almost all of the operations and notification messages used with multiline edit controls. Thus, applications that already use edit controls can be easily changed to use rich edit controls. Additional messages and notifications enable applications to access the functionality unique to rich edit controls. For information about edit controls, see CEdit.

For more information on notifications, see Notifications from a Rich Edit Control later in this topic.

See Also

Reference

Using CRichEditCtrl

Concepts

Controls (MFC)

Classes Related to Rich Edit Controls

The CRichEditView, CRichEditDoc, and CRichEditCntrItem classes provide the functionality of the rich edit control (CRichEditCtrl) within the context of MFC's document/view architecture. **CRichEditView** maintains the text and formatting characteristic of text. **CRichEditDoc** maintains the list of OLE client items that are in the view. **CRichEditCntrItem** provides container-side access to the OLE client item. To modify the contents of a **CRichEditView**, use CRichEditView::GetRichEditCtrl to access the underlying rich edit control.

See Also

Reference

Using CRichEditCtrl

Concepts

Controls (MFC)

Rich Edit Control Examples

The MFC OLE sample WORDPAD uses the **CRichEditView**, **CRichEditDoc**, and **CRichEditCntrItem** classes. By extension, it uses the CRichEditCtrl. For a quick description of these three classes, see Classes Related to Rich Edit Controls.

See Also

Reference

Using CRichEditCtrl

Concepts

Controls (MFC)

Character Formatting in Rich Edit Controls

You can use member functions of the rich edit control (CRichEditCtrl) to format characters and to retrieve formatting information. For characters, you can specify typeface, size, color, and effects such as bold, italic, and protected.

You can apply character formatting by using the SetSelectionCharFormat and SetWordCharFormat member functions. To determine the current character formatting for the selected text, use the GetSelectionCharFormat member function. The CHARFORMAT structure is used with these member functions to specify character attributes. One of the important members of **CHARFORMAT** is **dwMask**. In **SetSelectionCharFormat** and **SetWordCharFormat**, **dwMask** specifies which character attributes will be set by this function call. **GetSelectionCharFormat** reports the attributes of the first character in the selection; **dwMask** specifies the attributes that are consistent throughout the selection.

You can also get and set the "default character formatting," which is the formatting applied to any subsequently inserted characters. For example, if an application sets the default character formatting to bold and the user then types a character, that character is bold. To get and set default character formatting, use the GetDefaultCharFormat and SetDefaultCharFormat member functions.

The "protected" character attribute does not change the appearance of text. If the user attempts to modify protected text, a rich edit control sends its parent window an **EN_PROTECTED** notification message, allowing the parent window to allow or prevent the change. To receive this notification message, you must enable it by using the SetEventMask member function. For more information about the event mask, see Notifications from a Rich Edit Control, later in this topic.

Foreground color is a character attribute, but background color is a property of the rich edit control. To set the background color, use the SetBackgroundColor member function.

See Also

Reference

Using CRichEditCtrl

Concepts

Controls (MFC)

Paragraph Formatting in Rich Edit Controls

You can use member functions of the rich edit control (CRichEditCtrl) to format paragraphs and to retrieve formatting information. Paragraph formatting attributes include alignment, tabs, indents, and numbering.

You can apply paragraph formatting by using the SetParaFormat member function. To determine the current paragraph formatting for the selected text, use the GetParaFormat member function. The PARAFORMAT structure is used with these member functions to specify paragraph attributes. One of the important members of **PARAFORMAT** is **dwMask**. In **SetParaFormat**, **dwMask** specifies which paragraph attributes will be set by this function call. **GetParaFormat** reports the attributes of the first paragraph in the selection; **dwMask** specifies the attributes that are consistent throughout the selection.

See Also

Reference

Using CRichEditCtrl

Concepts

Controls (MFC)

Current Selection in a Rich Edit Control

The user can select text in a rich edit control (CRichEditCtrl) by using the mouse or the keyboard. The current selection is the range of selected characters, or the position of the insertion point if no characters are selected. An application can get information about the current selection, set the current selection, determine when the current selection changes, and show or hide the selection highlight.

To determine the current selection in a rich edit control, use the GetSel member function. To set the current selection, use the SetSel member function. The CHARRANGE structure is used with these functions to specify a range of characters. To retrieve information about the contents of the current selection, you can use the GetSelectionType member function.

By default, a rich edit control shows and hides the selection highlight when it gains and loses the focus. You can show or hide the selection highlight at any time by using the HideSelection member function. For example, an application might provide a Search dialog box to find text in a rich edit control. The application might select matching text without closing the dialog box, in which case it must use **HideSelection** to highlight the selection.

To get the selected text in a rich edit control, use the GetSelText member function. The text is copied to the specified character array. You must ensure that the array is large enough to hold the selected text plus a terminating null character.

You can search for a string in a rich edit control by using the FindText member function The FINDTEXTEX structure used with this function specifies the text range to search and the string to search for. You can also specify such options as whether the search is case-sensitive.

See Also

Reference

Using CRichEditCtrl

Concepts

Controls (MFC)

Word Breaks in Rich Edit Controls

A rich edit control (CRichEditCtrl) calls a function called a "word break procedure" to find breaks between words and to determine where it can break lines. The control uses this information when performing word-wrap operations and when processing the CTRL+LEFT and CTRL+RIGHT key combinations. An application can send messages to a rich edit control to replace the default word-break procedure, to retrieve word-break information, and to determine what line a given character falls on.

See Also

Reference

Using CRichEditCtrl

Concepts

Controls (MFC)

Clipboard Operations in Rich Edit Controls

Your application can paste the contents of the Clipboard into a rich edit control (CRichEditCtrl) using either the best available Clipboard format or a specific Clipboard format. You can also determine whether a rich edit control is capable of pasting a Clipboard format.

You can copy or cut the contents of the current selection by using the Copy or Cut member function. Similarly, you can paste the contents of the Clipboard into a rich edit control by using the Paste member function. The control pastes the first available format that it recognizes, which presumably is the most descriptive format.

To paste a specific Clipboard format, you can use the PasteSpecial member function. This function is useful for applications with a Paste Special command that enables the user to select the Clipboard format. You can use the CanPaste member function to determine whether a given format is recognized by the control.

You can also use **CanPaste** to determine whether any available Clipboard format is recognized by a rich edit control. This function is useful in the **OnInitMenuPopup** handler. An application might enable or gray its Paste command depending on whether the control can paste any available format.

Rich edit controls register two Clipboard formats: rich-text format and a format called RichEdit Text and Objects. An application can register these formats by using the RegisterClipboardFormat function, specifying the **CF_RTF** and **CF_RETEXTOBJ** values.

See Also

Reference

Using CRichEditCtrl

Concepts

Controls (MFC)

Stream Operations in Rich Edit Controls

You can use streams to transfer data into or out of a rich edit control (CRichEditCtrl). A stream is defined by an EDITSTREAM structure, which specifies a buffer and an application-defined callback function.

To read data into a rich edit control (that is, stream the data in), use the StreamIn member function. The control repeatedly calls the application-defined callback function, which transfers a portion of the data into the buffer each time.

To save the contents of a rich edit control (that is, stream the data out), you can use the StreamOut member function. The control repeatedly writes to the buffer and then calls the application-defined callback function. For each call, the callback function saves the contents of the buffer.

See Also

Reference

Using CRichEditCtrl

Concepts

Controls (MFC)

Printing in Rich Edit Controls

You can tell a rich edit control (CRichEditCtrl) to render its output for a specified device, such as a printer. You can also specify the output device for which a rich edit control formats its text.

To format part of the contents of a rich edit control for a specific device, you can use the FormatRange member function. The FORMATRANGE structure used with this function specifies the range of text to format as well as the device context (DC) for the target device.

After formatting text for an output device, you can send the output to the device by using the DisplayBand member function. By repeatedly using **FormatRange** and **DisplayBand**, an application that prints the contents of a rich edit control can implement banding. (Banding is division of output into smaller parts for printing purposes.)

You can use the SetTargetDevice member function to specify the target device for which a rich edit control formats its text. This function is useful for WYSIWYG (what you see is what you get) formatting, in which an application positions text using the default printer's font metrics instead of the screen's.

See Also

Reference

Using CRichEditCtrl

Concepts

Controls (MFC)

Bottomless Rich Edit Controls

Your application can resize a rich edit control (CRichEditCtrl) as needed so that it is always the same size as its contents. A rich edit control supports this so-called "bottomless" functionality by sending its parent window an EN_REQUESTRESIZE notification message whenever the size of its contents changes.

When processing the **EN_REQUESTRESIZE** notification message, an application should resize the control to the dimensions in the specified REQRESIZE structure. An application might also move any information near the control to accommodate the control's change in height. To resize the control, you can use the **CWnd** function SetWindowPos.

You can force a bottomless rich edit control to send an **EN_REQUESTRESIZE** notification message by using the RequestResize member function. This message can be useful in the OnSize handler.

To receive **EN_REQUESTRESIZE** notification messages, you must enable the notification by using the **SetEventMask** member function.

See Also

Reference

Using CRichEditCtrl

Concepts

Controls (MFC)

Notifications from a Rich Edit Control

Notification messages report events affecting a rich edit control (CRichEditCtrl). They can be processed by the parent window or, using message reflection, by the rich edit control itself. Rich edit controls support all of the notification messages used with edit controls as well as several additional ones. You can determine which notification messages a rich edit control sends its parent window by setting its "event mask."

To set the event mask for a rich edit control, use the SetEventMask member function. You can retrieve the current event mask for a rich edit control by using the GetEventMask member function.

The following paragraphs list several specific notifications and their uses:

- **EN_MSGFILTER** Handling the **EN_MSGFILTER** notification lets a class, either the rich edit control or its parent window, filter all keyboard and mouse input to the control. The handler can prevent the keyboard or mouse message from being processed or can change the message by modifying the specified MSGFILTER structure.
- **EN_PROTECTED** Handle the **EN_PROTECTED** notification message to detect when the user attempts to modify protected text. To mark a range of text as protected, you can set the protected character effect. For more information, see Character Formatting in Rich Edit Controls.
- **EN_DROPFILES** You can enable the user to drop files in a rich edit control by processing the **EN_DROPFILES** notification message. The specified ENDROPFILES structure contains information about the files being dropped.
- **EN_SELCHANGE** An application can detect when the current selection changes by processing the **EN_SELCHANGE** notification message. The notification message specifies a SELCHANGE structure containing information about the new selection.

See Also

Reference

Using CRichEditCtrl

Concepts

Controls (MFC)

Using CSliderCtrl

The CSliderCtrl class represents a slider control, which is also called a trackbar. A "slider control" is a window that contains a slider and optional tick marks. When the user moves the slider, using either the mouse or the arrow keys, the slider control sends notification messages to indicate the change.

Slider controls are useful when you want the user to select a discrete value or a set of consecutive values in a range. For example, you might use a slider control to allow the user to set the repeat rate of the keyboard by moving the slider to a given tick mark.

The slider in a slider control moves in increments that you specify when you create it. For example, if you specify that the slider control should have a range of five, the slider can only occupy six positions: a position at the left side of the slider control and one position for each increment in the range. Typically, each of these positions is identified by a tick mark.

What do you want to know more about?

- Using Slider Controls
- Slider Control Styles
- Slider Control Member Functions
- Slider Notification Messages

See Also
Concepts
Controls (MFC)

Using Slider Controls

Typical usage of an slider control follows the pattern below:

- The control is created. If the control is specified in a dialog box template, creation is automatic when the dialog box is created. (You should have a CSliderCtrl member in your dialog class that corresponds to the slider control.) Alternatively, you can use the Create member function to create the control as a child window of any window.
- Call the various Set member functions to set values for the control. Changes that you can make include setting the minimum and maximum positions for the slider, drawing tick marks, setting a selection range, and repositioning the slider. For controls in a dialog box, a good time to do this is in the dialog's OnInitDialog function.
- As the user interacts with the control, it will send various notification messages. You can extract the slider value from the control by calling the GetPos member function.
- When you're done with the control, you need to make sure it's properly destroyed. If the slider control is in a dialog box, it and the **CSliderCtrl** object will be destroyed automatically. If not, you need to ensure that both the control and the **CSliderCtrl** object are properly destroyed.

See Also

Reference

Using CSliderCtrl

Concepts

Controls (MFC)

Slider Control Styles

Slider controls (CSliderCtrl) can have either a vertical or horizontal orientation. They can have tick marks on either side, both sides, or neither. They can also be used to specify a range of consecutive values. These properties are controlled by using slider control styles, which you specify when you create the slider control.

The **TBS_HORZ** and **TBS_VERT** styles determine the orientation of the slider control. If you do not specify an orientation, the slider control is oriented horizontally.

The **TBS_AUTOTICKS** style creates a slider control that has a tick mark for each increment in its range of values. These tick marks are added automatically when you call the SetRange member function. If you do not specify **TBS_AUTOTICKS**, you can use member functions, such as SetTic and SetTicFreq, to specify the positions of the tick marks. To create a slider control that does not display tick marks, you can use the **TBS_NOTICKS** style.

You can display tick marks on either or both sides of the slider control. For horizontal slider controls, you can specify the **TBS_BOTTOM** or **TBS_TOP** style. For vertical slider controls, you can specify the **TBS_RIGHT** or **TBS_LEFT** style. (**TBS_BOTTOM** and **TBS_RIGHT** are the default settings.) For tick marks on both sides of the slider control in any orientation, specify the **TBS_BOTH** style.

A slider control can display a selection range only if you specify the **TBS_ENABLESELRANGE** style when you create it. When a slider control has this style, the tick marks at the starting and ending positions of a selection range are displayed as triangles (instead of vertical dashes) and the selection range is highlighted. For example, selection ranges might be useful in a simple scheduling application. The user could select a range of tick marks corresponding to hours in a day to identify a scheduled meeting time.

By default, the length of a slider control's slider varies as the selection range changes. If the slider control has the **TBS_FIXEDLENGTH** style, the length of the slider remains the same even if the selection range changes. A slider control that has the **TBS_NOTHUMB** style does not include a slider.

See Also

Reference

Using CSliderCtrl

Concepts

Controls (MFC)

Slider Control Member Functions

An application can call the slider control's member functions to retrieve information about the slider control (CSliderCtrl) and to change its characteristics.

To retrieve the position of the slider (that is, the value the user has chosen), use the GetPos member function. To set the position of the slider, use the SetPos member function. At any time you can use the VerifyPos member function to make sure that the slider is between the minimum and maximum values.

The range of a slider control is the set of contiguous values that the slider control can represent. Most applications use the SetRange member function to set the range of a slider control when it is first created. Applications can dynamically alter the range after the slider control has been created by using the SetRangeMax and SetRangeMin member functions. An application that allows the range to be changed dynamically typically retrieves the final range settings when the user has finished working with the slider control. To retrieve these settings, use the GetRange, GetRangeMax, and GetRangeMin member functions.

An application can use the **TBS_AUTOTICKS** style to have a slider control's tick marks displayed automatically. If an application needs to control the position or frequency of the tick marks, however, a number of member functions can be used.

To set the position of a tick mark, an application can use the SetTic member function. The SetTicFreq member function allows an application to set tick marks that appear at regular intervals in the slider control's range. For example, the application can use this member function to display only 10 tick marks in a range of 1 through 100.

To retrieve the index in the range corresponding to a tick mark, use the GetTic member function. The GetTicArray member function retrieves an array of these indices. To retrieve the position of a tick mark, in client coordinates, use the GetTicPos member function. An application can retrieve the number of tick marks by using the GetNumTics member function.

The ClearTics member function removes all of a slider control's tick marks.

A slider control's line size determines how far the slider moves when an application receives a **TB_LINEDOWN** or **TB_LINEUP** notification message. Similarly, the page size determines the response to the **TB_PAGEDOWN** and **TB_PAGEUP** notification messages. Applications can retrieve and set the line and page size values by using the GetLineSize, SetLineSize, GetPageSize, and SetPageSize member functions.

An application can use member functions to retrieve the dimensions of a slider control. The GetThumbRect member function retrieves the bounding rectangle for the slider. The GetChannelRect member function retrieves the bounding rectangle for the slider control's channel. (The channel is the area over which the slider moves and which contains the highlight when a range is selected.)

If a slider control has the **TBS_ENABLESELRANGE** style, the user can select a range of contiguous values from it. A number of member functions allow the selection range to be adjusted dynamically. The SetSelection member function sets the starting and ending positions of a selection. When the user has finished setting a selection range, an application can retrieve the settings by using the GetSelection member function. To clear a user's selection, use the ClearSel member function.

See Also

Reference

Using CSliderCtrl

Concepts

Controls (MFC)

Slider Notification Messages

A slider control notifies its parent window of user actions by sending the parent **WM_HSCROLL** or **WM_VSCROLL** messages, depending on the orientation of the slider control. To handle these messages, add handlers for the **WM_HSCROLL** and **WM_VSCROLL** messages to the parent window. The OnHScroll and OnVScroll member functions will be passed a notification code, the position of the slider, and a pointer to the CSliderCtrl object. Note that the pointer is of type **CScrollBar *** even though it points to a **CSliderCtrl** object. You may need to typecast this pointer if you need to manipulate the slider control.

Rather than using the scroll bar notification codes, slider controls send a different set of notification codes. A slider control sends the **TB_BOTTOM**, **TB_LINEDOWN**, **TB_LINEUP**, and **TB_TOP** notification codes only when the user interacts with a slider control by using the keyboard. The **TB_THUMBPOSITION** and **TB_THUMBTRACK** notification messages are only sent when the user is using the mouse. The **TB_ENDTRACK**, **TB_PAGEDOWN**, and **TB_PAGEUP** notification codes are sent in both cases.

The following table lists the slider control notification messages and the events (virtual key codes or mouse events) that cause the notifications to be sent. (For a list of standard virtual key codes, see Winuser.h.)

Notification message	Event causing notification to be sent
TB_BOTTOM	**VK_END**
TB_ENDTRACK	**WM_KEYUP** (the user released a key that sent a relevant virtual key code)
TB_LINEDOWN	**VK_RIGHT** or **VK_DOWN**
TB_LINEUP	**VK_LEFT** or **VK_UP**
TB_PAGEDOWN	**VK_NEXT** (the user clicked the channel below or to the right of the slider)
TB_PAGEUP	**VK_PRIOR** (the user clicked the channel above or to the left of the slider)
TB_THUMBPOSITION	**WM_LBUTTONUP** following a **TB_THUMBTRACK** notification message
TB_THUMBTRACK	Slider movement (the user dragged the slider)
TB_TOP	**VK_HOME**

See Also

Reference

Using CSliderCtrl

Concepts

Controls (MFC)

Using CSpinButtonCtrl

The spin button control (also called an up-down control) provides a pair of arrows that the user can click to adjust a value. This value is called the current position. The position stays within the range of the spin button. When the user clicks the up arrow, the position moves toward the maximum; and when the user clicks the down arrow, the position moves toward the minimum.

The spin button control is represented in MFC by class CSpinButtonCtrl.

Note
The default range for the spin button has the maximum set to zero (0) and the minimum set to 100. Since the maximum value is less than the minimum value, clicking the up arrow will decrease the position and clicking the down arrow will increase it. Use CSpinButtonCtrl::SetRange to adjust these values.

Typically, the current position is displayed in a companion control. The companion control is called the "buddy window." For an illustration of a spin button control, see About Up-Down Controls in the *Platform SDK*.

What do you want to know more about?

- Spin Button Styles
- Spin Button Member Functions

See Also
Concepts
Controls (MFC)

Spin Button Styles

Many of the settings for a spin button (CSpinButtonCtrl) are controlled by styles. You can set the following styles using the **Properties** window in the dialog editor.

- **Orientation** Either Vertical or Horizontal. Controls the orientation of the arrow buttons. Associated with the **UDS_HORZ** style.
- **Alignment** One of Unattached, Left, or Right. Controls the location of the spin button. Left and Right position the spin button next to the buddy window. The width of the buddy window is decreased to accommodate the spin button. Associated with the **UDS_ALIGNLEFT** and **UDS_ALIGNRIGHT** styles.
- **Auto Buddy** Automatically selects the previous window in Z-order as buddy window to the spin button. In a dialog template, this is the control which precedes the spin button in the tab order. Associated with the **UDS_AUTOBUDDY** style.
- **Set Buddy Integer** Causes the spin control to increment and decrement the caption of the buddy window as the current position changes. Associated with the **UDS_SETBUDDYINT** style.
- **No Thousands** Does not insert the thousands separator in the value in the caption of the buddy window. Associated with the **UDS_NOTHOUSANDS** style.

Note
Set this style if you want to use dialog data exchange (DDX) to get the integer value from the buddy control. **DDX_Text** does not accept embedded thousand separators.

- **Wrap** Causes the position to "wrap" as the value is incremented or decremented beyond the range of the control. Associated with the **UDS_WRAP** style.
- **Arrow Keys** Causes the spin button to increment or decrement the position when the UP ARROW and DOWN ARROW keys are pressed. Associated with the **UDS_ARROWKEYS** style.

See Also

Reference

Using CSpinButtonCtrl

Concepts

Controls (MFC)

Spin Button Member Functions

There are several member functions available for the spin control (CSpinButtonCtrl). Use these functions to change the following attributes of the spin button.

- **Acceleration** You can adjust the rate at which the position changes when the user holds down the arrow button. To work with acceleration, use the SetAccel and GetAccel member functions.
- **Base** You can change the base (either 10 or 16) used to display the position in the caption of the buddy window. To work with the base, use the GetBase and SetBase member functions.
- **Buddy Window** You can dynamically set the buddy window. To query or change which control is the buddy window, use the GetBuddy and SetBuddy member functions.
- **Position** You can query and change the position. To work directly with position, use the GetPos and SetPos member functions. Since the caption of the buddy control may have changed (for example, in the case that the buddy is an edit control), **GetPos** retrieves the current caption and adjusts the position accordingly.
- **Range** You can change the maximum and minimum positions for the spin button. By default, the maximum is set to 0, and the minimum is set to 100. Since the default maximum is less than the default minimum, the actions of the arrow buttons is counter-intuitive. Typically, you will set the range using the SetRange member function. To query the range use GetRange.

See Also

Reference

Using CSpinButtonCtrl

Concepts

Controls (MFC)

Using CStatusBarCtrl

You can use the status bar control (CStatusBarCtrl) to create a control window that reflects various kinds of status information about the application. The status window can be divided into parts that display more than one type of information.

What do you want to know more about?

- Methods of Creating a Status Bar
- Settings for the CStatusBarCtrl
- Using CStatusBarCtrl to Create a CStatusBarCtrl Object
- Setting the Mode of a CStatusBarCtrl Object
- Initializing the Parts of a CStatusBarCtrl Object
- Using Tooltips in a CStatusBarCtrl Object

See Also
Concepts
Controls (MFC)

Methods of Creating a Status Bar

MFC provides two classes to create status bars: CStatusBar and CStatusBarCtrl (which wraps the Windows common control API). **CStatusBar** provides all of the functionality of the status bar common control, it automatically interacts with menus and toolbars, and it handles many of the required common control settings and structures for you; however, your resulting executable usually will be larger than that created by using **CStatusBarCtrl**.

CStatusBarCtrl usually results in a smaller executable, and you may prefer to use **CStatusBarCtrl** if you do not intend to integrate the status bar into the MFC architecture. If you plan to use **CStatusBarCtrl** and integrate the status bar into the MFC architecture, you must take additional care to communicate status bar control manipulations to MFC. This communication is not difficult; however, it is additional work that is unneeded when you use **CStatusBar**.

Visual C++ provides two ways to take advantage of the status bar common control.

- Create the status bar using **CStatusBar**, and then call CStatusBar::GetStatusBarCtrl to get access to the **CStatusBarCtrl** member functions.
- Create the status bar using CStatusBarCtrl's constructor.

Either method will give you access to the member functions of the status bar control. When you call **CStatusBar::GetStatusBarCtrl**, it returns a reference to a **CStatusBarCtrl** object so you can use either set of member functions. See CStatusBar for information on constructing and creating a status bar using **CStatusBar**.

See Also
Reference
Using CStatusBarCtrl
Concepts
Controls (MFC)

Settings for the CStatusBarCtrl

The default position of a CStatusBarCtrl status window is along the bottom of the parent window, but you can specify the **CCS_TOP** style to have it appear at the top of the parent window's client area.

You can specify the **SBARS_SIZEGRIP** style to include a sizing grip at the right end of the **CStatusBarCtrl** status window. A sizing grip is similar to a sizing border; it is a rectangular area that the user can click and drag to resize the parent window.

Note
If you combine the **CCS_TOP** and **SBARS_SIZEGRIP** styles, the resulting sizing grip is not functional even though the system draws it in the status window.

The window procedure for the status window automatically sets the initial size and position of the control window. The width is the same as that of the parent window's client area. The height is based on the metrics of the font that is currently selected into the status window's device context and on the width of the window's borders.

The window procedure automatically adjusts the size of the status window whenever it receives a **WM_SIZE** message. Typically, when the size of the parent window changes, the parent sends a **WM_SIZE** message to the status window.

You can set the minimum height of a status window's drawing area by calling SetMinHeight, specifying the minimum height in pixels. The drawing area does not include the window's borders.

You retrieve the widths of the borders of a status window by calling GetBorders. This member function includes the pointer to a three-element array that receives the width of the horizontal border, the vertical border, and the border between rectangles.

See Also

Reference

Using CStatusBarCtrl

Concepts

Controls (MFC)

Using CStatusBarCtrl to Create a CStatusBarCtrl Object

Here is an example of a typical use of CStatusBarCtrl:

To use a status bar control with parts

1. Construct the **CStatusBarCtrl** object.
2. Call SetMinHeight if you want to set the minimum height of the status bar control's drawing area.
3. Call SetBkColor to set the background color of the status bar control.
4. Call SetParts to set the number of parts in a status bar control and the coordinate of the right edge of each part.
5. Call SetText to set the text in a given part of the status bar control. The message invalidates the portion of the control that has changed, causing it to display the new text when the control next receives the **WM_PAINT** message.

In some cases, the status bar only needs to display a line of text. In this case, make a call to SetSimple. This puts the status bar control into "simple" mode, which displays a single line of text.

See Also

Reference

Using CStatusBarCtrl

Concepts

Controls (MFC)

Setting the Mode of a CStatusBarCtrl Object

There are two modes for a **CStatusBarCtrl** object: simple and nonsimple. In the majority of cases, your status bar control will have one or more parts, along with text and perhaps an icon or icons. This is called the nonsimple mode. For more information on this mode, see Initializing the Parts of a CStatusBarCtrl Object.

However, there are cases where you only need to display a single line of text. In this case, the simple mode is sufficient for your needs. To change the mode of the **CStatusBarCtrl** object to simple, make a call to the SetSimple member function. Once the status bar control is in simple mode, set the text by calling the **SetText** member function, passing 255 as the value for the **nPane** parameter.

You can use the IsSimple function to determine what mode the **CStatusBarCtrl** object is in.

Note
If the status bar object is being changed from nonsimple to simple, or vice versa, the window is immediately redrawn and, if applicable, any defined parts are automatically restored.

See Also

Reference

Using CStatusBarCtrl

Concepts

Controls (MFC)

Initializing the Parts of a CStatusBarCtrl Object

By default, a status bar displays status information using separate panes. These panes (also referred to as parts) can contain either a text string, an icon, or both.

Use SetParts to define how many parts, and the length, the status bar will have. After you have created the parts of the status bar, make calls to SetText and SetIcon to set the text or icon for a specific part of the status bar. Once the part has been successfully set, the control is automatically redrawn.

The following example initializes an existing **CStatusBarCtrl** object (`m_pStatBarCtrl`) with four panes and then sets an icon (IDR_LIGHT1) and some text (IDC_PART1_STR) in the second part.

```
int strPartDim[4]= {80, 160, 240, -1};
CMyApp* pApp= (CMyApp*) AfxGetApp();

m_icon1= pApp->LoadIcon(IDR_LIGHT1);

m_pStatBarCtrl->SetParts(4, strPartDim);
m_pStatBarCtrl->SetIcon(1, m_icon1);
m_pStatBarCtrl->SetText(IDC_PART1_STR, 1, 0);
```

For more information on setting the mode of a **CStatusBarCtrl** object to simple, see Setting the Mode of a CStatusBarCtrl Object.

See Also

Reference

Using CStatusBarCtrl

Concepts

Controls (MFC)

Using Tooltips in a CStatusBarCtrl Object

To enable tooltips for a status bar control, create the **CStatusBarCtrl** object with the **SBT_TOOLTIPS** style.

Note
If you are using a **CStatusBar** object to implement your status bar, use the **CStatusBar::CreateEx** function. It allows you to s pecify additional styles for the embedded **CStatusBarCtrl** object.

Once the **CStatusBarCtrl** object has been successfully created, use CStatusBarCtrl::SetTipText and CStatusBarCtrl::GetTipText to set and retrieve the tip text for a specific pane.

Once the tool tip has been set, it is displayed only if the part has an icon and no text, or if all of the text cannot be displayed inside the part. Tool tips are not supported in simple mode.

See Also

Reference

Using CStatusBarCtrl

Concepts

Controls (MFC)

Using CTabCtrl

A "tab control" is analogous to the dividers in a notebook or the labeled folders in a file cabinet. Use the tab control, represented by class CTabCtrl, to show multiple pages of information or controls to a user, one at a time, in a format that suggests a peer or logical relationship between each page.

For more information on tab controls, see Tab Controls in the *Platform SDK*.

What do you want to know more about?

- Tab Controls and Property Sheets
- Tab Control Examples
- Tabs and Tab Control Attributes
- Making Owner-Drawn Tabs
- Working with a Tab Control
- Creating the Tab Control
- Adding Tabs to a Tab Control
- Processing Tab Control Notification Messages

See Also
Concepts
Controls (MFC)

Tab Controls and Property Sheets

Property sheets are multiple-page dialogs or "tab dialogs" that can display up to 24 dialog template resources to the user. For examples of property sheets, see the Windows Display Properties dialog box or the following MFC sample application:

CMNCTRL1: Demonstrates Common Control Classes, Part 1

CMNCTRL2: Demonstrates Common Control Classes, Part 2

Property sheets can be easily implemented using the MFC class CPropertySheet.

See Also

Reference

Using CTabCtrl

Concepts

Controls (MFC)

Tab Control Examples

Use of CTabCtrl is illustrated in the following MFC sample application:

See Also

Reference

Using CTabCtrl

Concepts

Controls (MFC)

Tabs and Tab Control Attributes

You have considerable control over the appearance and behavior of tabs that make up a tab control (CTabCtrl). Each tab can have a label, an icon, an item state, and an application-defined 32-bit value associated with it. For each tab, you can display the icon, the label, or both.

In addition, each tab item can have three possible states: pressed, unpressed, or highlighted. This state can only be set by modifying an existing tab item. To modify an existing tab item, retrieve it with a call to GetItem, modify the **TCITEM** structure (specifically the **dwState** and **dwStateMask** data members), and then return the modified **TCITEM** structure with a call to SetItem. If you need to clear the item states of all the tab items in a **CTabCtrl** object, make a call to DeselectAll. This function resets the state of all tab items or all items except the one currently selected.

The following code clears the state of all tab items and then modifies the state of the third item:

```
//modify the third item to be highlighted
TCITEM curItem;

m_tabCtrl.DeselectAll(FALSE); //reset all tab items
curItem.mask= TCIF_STATE;
m_tabCtrl.GetItem(2, &curItem);
curItem.mask= TCIF_STATE;
curItem.dwState= TCIS_HIGHLIGHTED;
curItem.dwStateMask= TCIS_HIGHLIGHTED;
m_tabCtrl.SetItem(2, &curItem);
```

For more information about tab attributes, see Tabs and Tab Attributes in the *Platform SDK*. For more information about adding tabs to a tab control, see Adding Tabs to a Tab Control later in this topic.

See Also

Reference

Using CTabCtrl

Concepts

Controls (MFC)

Making Owner-Drawn Tabs

You can define individual items of a tab control (CTabCtrl) to be owner-drawn items. For more information, see Owner-Drawn Tabs in the *Platform SDK*.

See Also

Reference

Using CTabCtrl

Concepts

Controls (MFC)

Working with a Tab Control

The easiest way to use a tab control (CTabCtrl) is by adding it to a dialog template resource with the dialog editor. You can also use a tab control by itself. MFC calls **InitCommonControls** for you. The key tasks are as follows:

- Creating the tab control
- Adding tabs to a tab control
- Processing tab control notification messages

If the tab control object is embedded in a parent view or dialog class, the control is destroyed when the parent is destroyed.

See Also
Reference
Using CTabCtrl
Concepts
Controls (MFC)

Creating the Tab Control

How the tab control is created depends on whether you are using the control in a dialog box or creating it in a nondialog window.

To use CTabCtrl directly in a dialog box

1. In the dialog editor, add a Tab Control to your dialog template resource. Specify its control ID.
2. Use the Add Member Variable Wizard to add a member variable of type CTabCtrl with the Control property. You can use this member to call **CTabCtrl** member functions.
3. Map handler functions in the dialog class for any tab control notification messages you need to handle. For more information, see Mapping Messages to Functions.
4. In OnInitDialog, set the styles for the **CTabCtrl**.

To use CTabCtrl in a nondialog window

1. Define the control in the view or window class.
2. Call the control's Create member function, possibly in OnInitialUpdate, possibly as early as the parent window's OnCreate handler function (if you're subclassing the control). Set the styles for the control.

After the **CTabCtrl** object has been created, you can set or clear the following extended styles:

- **TCS_EX_FLATSEPARATORS** The tab control will draw separators between the tab items. This extended style only affects tab controls that have the **TCS_BUTTONS** and **TCS_FLATBUTTONS** styles. By default, creating the tab control with the **TCS_FLATBUTTONS** style sets this extended style.
- **TCS_EX_REGISTERDROP** The tab control generates **TCN_GETOBJECT** notification messages to request a drop target object when an object is dragged over the tab items in the control.

Note
To receive the **TCN_GETOBJECT** notification, you must initialize the OLE libraries with a call to AfxOleInit.

These styles can be retrieved and set, after the control has been created, with respective calls to the GetExtendedStyle and SetExtendedStyle member functions.

For instance, set the **TCS_EX_FLATSEPARATORS** style with the following lines of code:

```
DWORD dwExStyle= m_tabCtrl.GetExtendedStyle();
m_tabCtrl.SetExtendedStyle(dwExStyle | TCS_EX_FLATSEPARATORS);
```

Clear the **TCS_EX_FLATSEPARATORS** style from a **CTabCtrl** object with the following lines of code:

```
DWORD dwExStyle= m_tabCtrl.GetExtendedStyle();
m_tabCtrl.SetExtendedStyle(dwExStyle & ~TCS_EX_FLATSEPARATORS);
```

This will remove the separators that appear between the buttons of your **CTabCtrl** object.

See Also

Reference

Using CTabCtrl

Concepts

Controls (MFC)

Adding Tabs to a Tab Control

After creating the tab control (CTabCtrl), add as many tabs as you need.

To add a tab item

1. Prepare a TCITEM structure.
2. Call CTabCtrl::InsertItem, passing the structure.
3. Repeat steps 1 and 2 for additional tab items.

For more information, see Creating a Tab Control in the *Platform SDK*.

See Also

Reference

Using CTabCtrl

Concepts

Controls (MFC)

Processing Tab Control Notification Messages

As users click tabs or buttons, the tab control (CTabCtrl) sends notification messages to its parent window. Handle these messages if you want to do something in response. For example, when the user clicks a tab, you may want to preset control data on the page prior to displaying it.

Process **WM_NOTIFY** messages from the tab control in your view or dialog class. Use the Properties window to create an OnChildNotify handler function with a switch statement based on which notification message is being handled. For a list of the notifications a tab control can send to its parent window, see the **Notifications** section of Tab Control Reference in the *Platform SDK*.

See Also

Reference

Using CTabCtrl

Concepts

Controls (MFC)

Using CToolBarCtrl

You can use the toolbar control (CToolBarCtrl) to create a control window containing buttons and optional spaces. Each button in the toolbar control window sends a command message to the parent window as the user chooses it. Typically, the buttons in a toolbar correspond to items in the application's menu, providing an additional and more direct way for the user to access an application's commands.

What do you want to know more about?

- Methods of Creating a Toolbar
- Settings for the Toolbar Control
- Creating a CToolBarCtrl Object
- Using Image Lists in a Toolbar Control
- Using Drop-Down Buttons in a Toolbar Control
- Customizing the Appearance of a Toolbar Control

See Also
Concepts
Controls (MFC)

Methods of Creating a Toolbar

MFC provides two classes to create toolbars: CToolBar and CToolBarCtrl (which wraps the Windows common control API). **CToolBar** provides all of the functionality of the toolbar common control, and it handles many of the required common control settings and structures for you; however, your resulting executable usually will be larger than that created by using **CToolBarCtrl**.

CToolBarCtrl usually results in a smaller executable, and you may prefer to use **CToolBarCtrl** if you do not intend to integrate the toolbar into the MFC architecture. If you plan to use **CToolBarCtrl** and integrate the toolbar into the MFC architecture, you must take additional care to communicate toolbar control manipulations to MFC. This communication is not difficult; however, it is additional work that is unneeded when you use **CToolBar**.

Visual C++ provides two ways to take advantage of the toolbar common control.

- Create the toolbar using **CToolBar**, and then call CToolBar::GetToolBarCtrl to get access to the **CToolBarCtrl** member functions.
- Create the toolbar using CToolBarCtrl's constructor.

Either method will give you access to the member functions of the toolbar control. When you call **CToolBar::GetToolBarCtrl**, it returns a reference to a **CToolBarCtrl** object so you can use either set of member functions. See CToolBar for information on constructing and creating a toolbar using **CToolBar**.

See Also

Reference

Using CToolBarCtrl

Concepts

Controls (MFC)

Settings for the Toolbar Control

The buttons on a toolbar can display a bitmap, a string, or both. By default, the image size is set to the dimensions of 16 by 15 pixels. All buttons are the same width, by default 24 by 22 pixels. A toolbar's height is determined by the height of the buttons, and a toolbar's width is the same as the width of the parent window's client area, also by default.

A toolbar can have built-in customization features, including a system-defined customization dialog box, that allow the user to insert, delete, or rearrange toolbar buttons. An application determines whether the customization features are available to the user and controls the extent to which the user can customize the toolbar. For more information about customizing the toolbar, see class CToolBarCtrl in the *MFC Reference*.

See Also

Reference

Using CToolBarCtrl

Concepts

Controls (MFC)

Creating a CToolBarCtrl Object

CToolBarCtrl objects contain several internal data structures — a list of button image bitmaps, a list of button label strings, and a list of **TBBUTTON** structures — that associate an image and/or string with the position, style, state, and command ID of the button. Each of the elements of these data structures is referred to by a zero-based index. Before you can use a **CToolBarCtrl** object, you must set up these data structures. For a list of the data structures, see Toolbar Controls in the *Platform SDK*. The list of strings can only be used for button labels; you cannot retrieve strings from the toolbar.

To use a **CToolBarCtrl** object, you will typically follow these steps:

To use a CToolBarCtrl object

1. Construct the CToolBarCtrl object.
2. Call Create to create the Windows toolbar common control and attach it to the **CToolBarCtrl** object. If you want bitmap images for buttons, add the button bitmaps to the toolbar by calling AddBitmap. If you want string labels for buttons, add the strings to the toolbar by calling AddString and/or AddStrings. After calling **AddString** and/or **AddStrings**, you should call AutoSize in order to get the string or strings to appear.
3. Add button structures to the toolbar by calling AddButtons.
4. If you want tool tips, handle **TTN_NEEDTEXT** messages in the toolbar's owner window as described in Handling Tool Tip Notifications.
5. If you want your user to be able to customize the toolbar, handle customization notification messages in the owner window as described in Handling Customization Notifications.

See Also

Reference

Using CToolBarCtrl

Concepts

Controls (MFC)

Using Image Lists in a Toolbar Control

By default, the images used by the buttons in a toolbar control are stored as a single bitmap. However, you can also store button images in a set of image lists. The toolbar control object can use up to three separate image lists:

- Enabled image list Contains images for toolbar buttons that are currently enabled.
- Disabled image list Contains images for toolbar buttons that are currently disabled.
- Highlighted image list Contains images for toolbar buttons that are currently highlighted. This image list is used only when the toolbar uses the **TBSTYLE_FLAT** style.

These image lists are used by the toolbar control when you associate them with the **CToolBarCtrl** object. This association is accomplished by making calls to CToolBarCtrl::SetImageList, SetDisabledImageList, and SetHotImageList.

By default, MFC uses the **CToolBar** class to implement MFC application toolbars. However, the **GetToolBarCtrl** member function can be used to retrieve the embedded **CToolBarCtrl** object. You can then make calls to **CToolBarCtrl** member functions using the returned object.

The following example demonstrates this technique by assigning an enabled (`m_TBarImages`) and disabled (`m_TBarDis`) image list to a **CToolBarCtrl** object (`m_wndToolBar`).

```
CWinApp* pApp= AfxGetApp();

m_TBarImages.Create(16, 16, ILC_COLOR, 4, 4);
m_TBarImages.Add(pApp->LoadIcon(IDI_BLK));
m_TBarImages.Add(pApp->LoadIcon(IDI_RED));
m_TBarImages.Add(pApp->LoadIcon(IDI_YELL));
m_TBarImages.Add(pApp->LoadIcon(IDI_WHI));

m_TBarDis.Create(16, 16, ILC_COLOR, 4, 4);
m_TBarDis.Add(pApp->LoadIcon(IDI_DIS_BLK));
m_TBarDis.Add(pApp->LoadIcon(IDI_DIS_RED));
m_TBarDis.Add(pApp->LoadIcon(IDI_DIS_YELL));
m_TBarDis.Add(pApp->LoadIcon(IDI_DIS_WHI));

m_wndToolBar.SetImageList(&m_TBarImages);
m_wndToolBar.SetDisabledImageList(&m_TBarDis);
```

Note

The image lists used by the toolbar object must be permanent objects. For this reason, they are commonly data members of an MFC class; in this example, the main frame window class.

Once the image lists are associated with the **CToolBarCtrl** object, the framework automatically displays the proper button image.

See Also

Reference

Using CToolBarCtrl

Concepts

Controls (MFC)

Using Drop-Down Buttons in a Toolbar Control

In addition to standard push buttons, a toolbar can also have drop-down buttons. A drop-down button is usually indicated by the presence of an attached down arrow.

Note
The attached down arrow will appear only if the **TBSTYLE_EX_DRAWDARROWS** extended style has been set.

When the user clicks on this arrow (or the button itself, if no arrow is present), a **TBN_DROPDOWN** notification message is sent to the parent of the toolbar control. You can then handle this notification and display a pop-up menu; similar to the behavior of Internet Explorer.

The following procedure illustrates how to implement a drop-down toolbar button with a pop-up menu:

To implement a drop-down button

1. Once your **CToolBarCtrl** object has been created, set the **TBSTYLE_EX_DRAWDDARROWS** style, using the following code:

```
m_wndToolBar.GetToolBarCtrl().SetExtendedStyle(TBSTYLE_EX_DRAWDDARROWS);
```

2. Set the **TBSTYLE_DROPDOWN** style for any new (InsertButton or AddButtons) or existing (SetButtonInfo) buttons that will be drop-down buttons. The following example demonstrates modifying an existing button in a **CToolBarCtrl** object:

```
TBBUTTONINFO tbi;

tbi.dwMask= TBIF_STYLE;
tbi.cbSize= sizeof(TBBUTTONINFO);
m_wndToolBar.GetToolBarCtrl().GetButtonInfo(ID_EDIT_CUT, &tbi);
tbi.fsStyle |= TBSTYLE_DROPDOWN;
m_wndToolBar.GetToolBarCtrl().SetButtonInfo(ID_EDIT_CUT, &tbi);
```

3. Add a **WM_NOTIFY** handler to the parent class of the toolbar object.

4. In the new handler, check for the **TBN_DROPDOWN** notification, using the following code:

```
#define lpnm   ((LPNMHDR)lParam)
#define lpnmTB ((LPNMTOOLBAR)lParam)

   switch(lpnm->code)
   {
      case TBN_DROPDOWN:
         //drop down button was hit
         //handle appropriately
         . . .
         return FALSE; //indicates the TBN_DROPDOWN
                       //notification was handled.
   }
```

5. If the **TBN_DROPDOWN** notification has been sent, display the appropriate popup menu. The following code demonstrates one method:

```
CMenu menu;
VERIFY(menu.LoadMenu(IDR_MENU1));
CMenu* pPopup = menu.GetSubMenu(0);
ASSERT(pPopup != NULL);
```

```
pPopup->TrackPopupMenu(TPM_RIGHTALIGN |
   TPM_RIGHTBUTTON, x, y, this);
```

See Also

Reference

Using CToolBarCtrl

Concepts

Controls (MFC)

Customizing the Appearance of a Toolbar Control

Class **CToolBarCtrl** provides many styles that affect the appearance (and, occasionally, the behavior) of the toolbar object. Modify the toolbar object by setting the *dwCtrlStyle* parameter of the **CToolBarCtrl::Create** (or **CToolBar::CreateEx**) member function, when you first create the toolbar control.

The following styles affect the "3D" aspect of the toolbar buttons and the placement of the button text:

- **TBSTYLE_FLAT** Creates a flat toolbar where both the toolbar and the buttons are transparent. Button text appears under button bitmaps. When this style is used, the button underneath the cursor is automatically highlighted.
- **TBSTYLE_TRANSPARENT** Creates a transparent toolbar. In a transparent toolbar, the toolbar is transparent but the buttons are not. Button text appears under button bitmaps.
- **TBSTYLE_LIST** Places button text to the right of button bitmaps.

Note
To prevent repaint problems, the **TBSTYLE_FLAT** and **TBSTYLE_TRANSPARENT** styles should be set before the toolbar object is visible.

The following styles determine if the toolbar allows a user to reposition individual buttons within a toolbar object using drag and drop:

- **TBSTYLE_ALTDRAG** Allows users to change a toolbar button's position by dragging it while holding down ALT. If this style is not specified, the user must hold down SHIFT while dragging a button.

Note
The **CCS_ADJUSTABLE** style must be specified to enable toolbar buttons to be dragged.

- **TBSTYLE_REGISTERDROP** Generates **TBN_GETOBJECT** notification messages to request drop target objects when the mouse pointer passes over toolbar buttons.

The remaining styles affect visual and nonvisual aspects of the toolbar object:

- **TBSTYLE_WRAPABLE** Creates a toolbar that can have multiple lines of buttons. Toolbar buttons can "wrap" to the next line when the toolbar becomes too narrow to include all buttons on the same line. Wrapping occurs on separation and nongroup boundaries.
- **TBSTYLE_CUSTOMERASE** Generates **NM_CUSTOMDRAW** notification messages when it processes **WM_ERASEBKGND** messages.
- **TBSTYLE_TOOLTIPS** Creates a tool tip control that an application can use to display descriptive text for the buttons in the toolbar.

For a complete listing of toolbar styles and extended styles, see Toolbar Control and Button Styles and Toolbar Extended Styles in the *Platform SDK*.

See Also

Reference

Using CToolBarCtrl

Concepts

Controls (MFC)

Handling Tool Tip Notifications

When you specify the **TBSTYLE_TOOLTIPS** style, the toolbar creates and manages a tool tip control. A tool tip is a small pop-up window that contains a line of text describing a toolbar button. The tool tip is hidden, appearing only when the user puts the cursor on a toolbar button and leaves it there for approximately one-half second. The tool tip is displayed near the cursor.

Before the tool tip is displayed, the **TTN_NEEDTEXT** notification message is sent to the toolbar's owner window to retrieve the descriptive text for the button. If the toolbar's owner window is a **CFrameWnd** window, tool tips are displayed without any extra effort, because **CFrameWnd** has a default handler for the **TTN_NEEDTEXT** notification. If the toolbar's owner window is not derived from **CFrameWnd**, such as a dialog box or form view, you must add an entry to your owner window's message map and provide a notification handler in the message map. The entry to your owner window's message map is as follows:

```
ON_NOTIFY_EX( TTN_NEEDTEXT, 0, memberFxn )
```

Remarks

memberFxn

The member function to be called when text is needed for this button.

Note that the id of a tool tip is always 0.

In addition to the **TTN_NEEDTEXT** notification, a tool tip control can send the following notifications to a toolbar control:

Notification	Meaning
TTN_NEEDTEXTA	Tool tip control requires ASCII text (Win95 only)
TTN_NEEDTEXTW	Tool tip control requires UNICODE text (Windows NT only)
TBN_HOTITEMCHANGE	Indicates that the hot (highlighted) item has changed.
NM_RCLICK	Indicates the user has right-clicked a button.
TBN_DRAGOUT	Indicates the user has clicked the button and dragged the pointer off the button. It allows an application to implement drag and drop from a toolbar button. When receiving this notification, the application will begin the drag and drop operation.
TBN_DROPDOWN	Indicates the user has clicked a button that uses the **TBSTYLE_DROPDOWN** style.
TBN_GETOBJECT	Indicates the user moved the pointer over a button that uses the **TBSTYLE_DROPPABLE** style.

For an example handler function and more information about enabling tool tips, see Tool Tips.

See Also

Reference

Using CToolBarCtrl

Concepts

Controls (MFC)

Handling Customization Notifications

A Windows toolbar common control has built-in customization features, including a system-defined customization dialog box, which allow the user to insert, delete, or rearrange toolbar buttons. The application determines whether the customization features are available and controls the extent to which the user can customize the toolbar.

You can make these customization features available to the user by giving the toolbar the **CCS_ADJUSTABLE** style. The customization features allow the user to drag a button to a new position or to remove a button by dragging it off the toolbar. In addition, the user can double-click the toolbar to display the **Customize Toolbar** dialog box, which allows the user to add, delete, and rearrange toolbar buttons. The application can display the dialog box by using the Customize member function.

The toolbar control sends notification messages to the parent window at each step in the customization process. If the user holds the SHIFT key down and begins dragging a button, the toolbar automatically handles the drag operation. The toolbar sends the **TBN_QUERYDELETE** notification message to the parent window to determine whether the button may be deleted. The drag operation ends if the parent window returns **FALSE**. Otherwise, the toolbar captures mouse input and waits for the user to release the mouse button.

When the user releases the mouse button, the toolbar control determines the location of the mouse cursor. If the cursor is outside the toolbar, the button is deleted. If the cursor is on another toolbar button, the toolbar sends the **TBN_QUERYINSERT** notification message to the parent window to determine if a button may be inserted to the left of the given button. The button is inserted if the parent window returns **TRUE**; otherwise, it is not. The toolbar sends the **TBN_TOOLBARCHANGE** notification message to signal the end of the drag operation.

If the user begins a drag operation without holding down the SHIFT key, the toolbar control sends the **TBN_BEGINDRAG** notification message to the owner window. An application that implements its own button-dragging code can use this message as a signal to begin a drag operation. The toolbar sends the **TBN_ENDDRAG** notification message to signal the end of the drag operation.

A toolbar control sends notification messages when the user customizes a toolbar by using the **Customize Toolbar** dialog box. The toolbar sends the **TBN_BEGINADJUST** notification message after the user double-clicks the toolbar, but before the dialog box is created. Next, the toolbar begins sending a series of **TBN_QUERYINSERT** notification messages to determine whether the toolbar allows buttons to be inserted. When the parent window returns **TRUE**, the toolbar stops sending **TBN_QUERYINSERT** notification messages. If the parent window does not return **TRUE** for any button, the toolbar destroys the dialog box.

Next, the toolbar control determines if any buttons may be deleted from the toolbar by sending one **TBN_QUERYDELETE** notification message for each button in the toolbar. The parent window returns **TRUE** to indicate that a button may be deleted; otherwise, it returns **FALSE**. The toolbar adds all toolbar buttons to the dialog box, but grays those that may not be deleted.

Whenever the toolbar control needs information about a button in the Customize Toolbar dialog box, it sends the **TBN_GETBUTTONINFO** notification message, specifying the index of the button for which it needs information and the address of a **TBNOTIFY** structure. The parent window must fill the structure with the relevant information.

The **Customize Toolbar** dialog box includes a Help button and a Reset button. When the user chooses the Help button, the toolbar control sends the **TBN_CUSTHELP** notification message. The parent window should respond by displaying help information. The dialog box sends the **TBN_RESET** notification message when the user selects the Reset button. This message signals that the toolbar is about to reinitialize the dialog box.

These messages are all **WM_NOTIFY** messages, and they can be handled in your owner window by adding message-map entries of the following form to your owner window's message map:

```
ON_NOTIFY( wNotifyCode, idControl, memberFxn )
```

wNotifyCode

Notification message identifier code, such as **TBN_BEGINADJUST**.

idControl

The identifier of the control sending the notification.

memberFxn

The member function to be called when this notification is received.

Your member function would be declared with the following prototype:

```
afx_msg void memberFxn( NMHDR * pNotifyStruct, LRESULT * result );
```

If the notification message handler returns a value, it should put it in the **LRESULT** pointed to by *result*.

For each message, *pNotifyStruct* points to either an **NMHDR** structure or a **TBNOTIFY** structure. These structures are described below:

The **NMHDR** structure contains the following members:

```
typedef struct tagNMHDR {
HWND hwndFrom;  // handle of control sending message
UINT idFrom;// identifier of control sending message
UINT code;  // notification code; see below
} NMHDR;
```

hwndFrom

Window handle of the control that is sending the notification. To convert this handle to a **CWnd** pointer, use CWnd::FromHandle.

idFrom

Identifier of the control sending the notification.

code

Notification code. This member can be a value specific to a control type, such as **TBN_BEGINADJUST** or **TTN_NEEDTEXT**, or it can be one of the common notification values listed below:

- **NM_CLICK** The user has clicked the left mouse button within the control.
- **NM_DBLCLK** The user has double-clicked the left mouse button within the control.
- **NM_KILLFOCUS** The control has lost the input focus.
- **NM_OUTOFMEMORY** The control could not complete an operation because there is not enough memory available.
- **NM_RCLICK** The user has clicked the right mouse button within the control.
- **NM_RDBLCLK** The user has double-clicked the right mouse button within the control.
- **NM_RETURN** The control has the input focus, and the user has pressed the ENTER key.
- **NM_SETFOCUS** The control has received the input focus.

The **TBNOTIFY** structure contains the following members:

```
typedef struct {
NMHDR hdr; // information common to all WM_NOTIFY messages
int iItem; // index of button associated with notification
TBBUTTON tbButton; // info about button associated with notification
int cchText;   // count of characters in button text
LPSTR lpszText;// address of button text
} TBNOTIFY, FAR* LPTBNOTIFY;
```

Remarks

hdr

Information common to all **WM_NOTIFY** messages.

iItem

Index of button associated with notification.

tbButton

TBBUTTON structure that contains information about the toolbar button associated with the notification.

cchText

Count of characters in button text.

lpszText

Pointer to button text.

The notifications the toolbar sends are as follows:

- **TBN_BEGINADJUST** Sent when the user begins customizing a toolbar control. The pointer points to an **NMHDR** structure that contains information about the notification. The handler doesn't need to return any specific value.
- **TBN_BEGINDRAG** Sent when the user begins dragging a button in a toolbar control. The pointer points to a **TBNOTIFY** structure. The **iItem** member contains the zero-based index of the button being dragged. The handler doesn't need to return any specific value.
- **TBN_CUSTHELP** Sent when the user chooses the Help button in the Customize Toolbar dialog box. No return value. The pointer points to an **NMHDR** structure that contains information about the notification message. The handler doesn't need to return any specific value.
- **TBN_ENDADJUST** Sent when the user stops customizing a toolbar control. The pointer points to an **NMHDR** structure that contains information about the notification message. The handler doesn't need to return any specific value.
- **TBN_ENDDRAG** Sent when the user stops dragging a button in a toolbar control. The pointer points to a **TBNOTIFY** structure. The **iItem** member contains the zero-based index of the button being dragged. The handler doesn't need to return any specific value.
- **TBN_GETBUTTONINFO** Sent when the user is customizing a toolbar control. The toolbar uses this notification message to retrieve information needed by the Customize Toolbar dialog box. The pointer points to a **TBNOTIFY** structure. The **iItem** member specifies the zero-based index of a button. The **pszText** and **cchText** members specify the address and length, in characters, of the current button text. An application should fill the structure with information about the button. Return **TRUE** if button information was copied to the structure, or **FALSE** otherwise.
- **TBN_QUERYDELETE** Sent while the user is customizing a toolbar to determine whether a button may be deleted from a toolbar control. The pointer points to a **TBNOTIFY** structure. The **iItem** member contains the zero-based index of the button to be deleted. Return **TRUE** to allow the button to be deleted or **FALSE** to prevent the button from being deleted.
- **TBN_QUERYINSERT** Sent while the user is customizing a toolbar control to determine whether a button may be inserted to the left of the given button. The pointer points to a **TBNOTIFY** structure. The **iItem** member contains the zero-based index of the button to be inserted. Return **TRUE** to allow a button to be inserted in front of the given button or **FALSE** to prevent the button from being inserted.
- **TBN_RESET** Sent when the user resets the content of the Customize Toolbar dialog box. The pointer points to an **NMHDR** structure that contains information about the notification message. The handler doesn't need to return any specific value.
- **TBN_TOOLBARCHANGE** Sent after the user has customized a toolbar control. The pointer points to an **NMHDR** structure that contains information about the notification message. The handler doesn't need to return any specific value.

See Also

Reference

Using CToolBarCtrl

Concepts

Controls (MFC)

Using CToolTipCtrl

The CToolTipCtrl class encapsulates the functionality of a tool tip control, a small pop-up window that displays a single line of text describing the purpose of a tool in an application. A tool tip is hidden most of the time, appearing only when the user puts the cursor on a tool and leaves it there for approximately one-half second. The tool tip appears near the cursor and disappears when the user clicks a mouse button or moves the cursor off of the tool.

What do you want to know more about?

- Methods of Creating Tool Tips
- Settings for the Tool Tip Control
- Using CToolTipCtrl to Create and Manipulate a CToolTipCtrl Object
- Manipulating the Tool Tip Control

See Also

Reference

Using CToolBarCtrl

Concepts

Controls (MFC)

Methods of Creating Tool Tips

MFC provides three classes to create and manage the tool tip control: CWnd, CToolBarCtrl, and CToolTipCtrl. The tool tip member functions in these classes wrap the Windows common control API. Class **CToolBarCtrl** and class **CToolTipCtrl** are derived from class **CWnd**.

CWnd provides four member functions to create and manage tool tips: EnableToolTips, CancelToolTips, FilterToolTipMessage, and OnToolHitTest. See these individual member functions for more information about how they implement tool tips.

If you create a toolbar using **CToolBarCtrl**, you can implement tool tips for that toolbar directly using the following member functions: GetToolTips and SetToolTips. See these individual member functions and Handling Tool Tip Notifications for more information about how they implement tool tips.

The **CToolTipCtrl** class provides the functionality of the Windows common tool tip control. A single tool tip control can provide information for more than one tool. A tool is either a window, such as a child window or control, or an application-defined rectangular area within a window's client area.

See Also

Reference

Using CToolTipCtrl

Concepts

Controls (MFC)

Settings for the Tool Tip Control

You can set the tool tip control (CToolTipCtrl) to be either active or inactive. When you set it to be active, the tool tip control appears when the cursor is on a tool. When you set it to be inactive, the tool tip control does not appear, even if the cursor is on a tool. Call Activate to activate or deactivate a tool tip control.

You can set an active tool tip to display the tool tip when the cursor is on a tool, whether or not the tool tip control's owner window is active or inactive, by using the **TTS_ALWAYSTIP** style. If you do not use this style, the tool tip control appears when the tool's owner window is active, but not when it is inactive.

Most applications contain toolbars with tools that correspond to menu commands. For such tools, it is convenient for the tool tip control to display the same text as the corresponding menu item. The system automatically strips the ampersand (&) accelerator characters from all strings passed to a tool tip control, unless the control has the **TTS_NOPREFIX** style.

See Also

Reference

Using CToolTipCtrl

Concepts

Controls (MFC)

Using CToolTipCtrl to Create and Manipulate a CToolTipCtrl Object

Here is an example of CToolTipCtrl usage:

To create and manipulate a CToolTipCtrl

1. Construct the **CToolTipCtrl** object.
2. Call Create to create the Windows tool tip common control and attach it to the **CToolTipCtrl** object.
3. Call AddTool to register a tool with the tool tip control, so that the information stored in the tool tip is displayed when the cursor is on the tool.
4. Call SetToolInfo to set the information that a tool tip maintains for a tool.
5. Call SetToolRect to set a new bounding rectangle for a tool.
6. Call HitTest to test a point to determine whether it is within the bounding rectangle of the given tool and, if so, retrieve information about the tool.
7. Call GetToolCount to retrieve a count of the tools registered with the tool tip control.

See Also

Reference

Using CToolTipCtrl

Concepts

Controls (MFC)

Manipulating the Tool Tip Control

Class **CToolTipCtrl** provides a group of member functions that control the various attributes of the **CToolTipCtrl** object and the tool tip window.

The initial, pop-up, and reshow durations for the tool tip windows can be set and retrieved with calls to GetDelayTime and SetDelayTime.

Change the appearance of the tool tip windows with the following functions:

- GetMargin and SetMargin Retrieves and sets the width between the tool tip border and the tool tip text.
- GetMaxTipWidth and SetMaxTipWidth Retrieves and sets the maximum width of the tool tip window.
- GetTipBkColor and SetTipBkColor Retrieves and sets the background color of the tool tip window.
- GetTipTextColor and SetTipTextColor Retrieves and sets the text color of the tool tip window.

In order for the tool tip control to be notified of important messages, such as **WM_LBUTTONXXX** messages, you must relay the messages to your tool tip control. The best method for this relay is to make a call to CToolTipCtrl::RelayEvent, in the **PreTranslateMessage** function of the owner window. The following example illustrates one possible method (assuming the tool tip control is called `m_ToolTip`):

```
if(pMsg->message== WM_LBUTTONDOWN ||
        pMsg->message== WM_LBUTTONUP ||
        pMsg->message== WM_MOUSEMOVE)
m_ToolTip.RelayEvent(pMsg);

return CMyView::PreTranslateMessage(pMsg);
```

To immediately remove a tool tip window, call the Pop member function.

See Also
Reference
Using CToolTipCtrl
Concepts
Controls (MFC)

Using CTreeCtrl

A tree control, represented by the class CTreeCtrl, is a window that displays a hierarchical list of items, such as the headings in a document, the entries in an index, or the files and directories on a disk. Each item consists of a label and an optional bitmapped image, and each item can have a list of subitems associated with it. By clicking an item, the user can expand and collapse the associated list of subitems. The directory tree in the left-hand pane of the Windows Explorer is an example of a tree control.

What do you want to know more about?

- CTreeCtrl vs. CTreeView
- Using Tree Controls
- Communicating with a Tree Control
- Tree Control Styles
- Tree Control Parent and Child Items
- Tree Control Item Position
- Tree Control Item Labels
- Tree Control Label Editing
- Tree Control Item States Overview
- Tree Control Image Lists
- Tree Control Item Selection
- Tree Control Drag-and-Drop Operations
- Tree Control Item Information
- Tree Control Notification Messages

See Also
Concepts
Controls (MFC)

CTreeCtrl vs. CTreeView

MFC provides two classes that encapsulate tree controls: CTreeCtrl and CTreeView. Each class is useful in different situations.

Use **CTreeCtrl** when you need a plain child window control; for instance, in a dialog box. You'd especially want to use **CTreeCtrl** if there will be other child controls in the window, as in a typical dialog box.

Use **CTreeView** when you want the tree control to act like a view window in document/view architecture as well as a tree control. A **CTreeView** will occupy the entire client area of a frame window or splitter window. It will be automatically resized when its parent window is resized, and it can process command messages from menus, accelerator keys, and toolbars. Since a tree control contains the data necessary to display the tree, the corresponding document object does not have to be complicated — you could even use CDocument as the document type in your document template.

See Also

Reference

Using CTreeCtrl

Concepts

Controls (MFC)

Using Tree Controls

Typical usage of a tree control (CTreeCtrl) follows the pattern below:

- The control is created. If the control is specified in a dialog box template or if you're using **CTreeView**, creation is automatic when the dialog box or view is created. If you want to create the tree control as a child window of some other window, use the Create member function.
- If you want your tree control to use images, set an image list by calling SetImageList. You can also change the indentation by calling SetIndent. A good time to do this is in OnInitDialog (for controls in dialog boxes) or OnInitialUpdate (for views).
- Put data into the control by calling the **CTreeCtrl**'s InsertItem function once for each data item. **InsertItem** returns a handle to the item you can use to refer to it later, such as when adding child items. A good time to initialize the data is in **OnInitDialog** (for controls in dialog boxes) or **OnInitialUpdate** (for views).
- As the user interacts with the control, it will send various notification messages. You can specify a function to handle each of the messages you want to handle by adding an **ON_NOTIFY_REFLECT** macro in your control window's message map or by adding an **ON_NOTIFY** macro to your parent window's message map. See Tree Control Notification Messages later in this topic for a list of possible notifications.
- Call the various Set member functions to set values for the control. Changes that you can make include setting the indentation and changing the text, image, or data associated with an item.
- Use the various Get functions to examine the contents of the control. You can also traverse the contents of the tree control with functions that allow you to retrieve handles to parents, children, and siblings of a specified item. You can even sort the children of a particular node.
- When you're done with the control, make sure it's properly destroyed. If the tree control is in a dialog box or if it's a view, it and the **CTreeCtrl** object will be destroyed automatically. If not, you need to ensure that both the control and the **CTreeCtrl** object are properly destroyed.

See Also
Reference
Using CTreeCtrl
Concepts
Controls (MFC)

Communicating with a Tree Control

You use different methods for calling member functions in a CTreeCtrl object depending on how the object was created:

- If the tree control is in a dialog box, use a member variable of type **CTreeCtrl** that you create in the dialog box class.
- If the tree control is a child window, use the **CTreeCtrl** object (or pointer) you used to construct the object.
- If you're using a **CTreeView** object, use the function CTreeView::GetTreeCtrl to get a reference to the tree control. You can initialize another reference with this value or assign the address of the reference to a **CTreeCtrl** pointer.

See Also

Reference

Using CTreeCtrl

Concepts

Controls (MFC)

Tree Control Styles

Tree control (CTreeCtrl) styles govern aspects of a tree control's appearance. You set the initial styles when you create the tree control. You can retrieve and change the styles after creating the tree control by using the GetWindowLong and SetWindowLong Windows functions, specifying **GWL_STYLE** for the *nIndex* parameter. For a complete list of styles, see Tree View Control Window Styles in the *Platform SDK*.

The **TVS_HASLINES** style enhances the graphic representation of a tree control's hierarchy by drawing lines that link child items to their corresponding parent item. This style does not link items at the root of the hierarchy. To do so, you need to combine the **TVS_HASLINES** and **TVS_LINESATROOT** styles.

The user can expand or collapse a parent item's list of child items by double-clicking the parent item. A tree control that has the **TVS_SINGLEEXPAND** style causes the item being selected to expand and the item being unselected to collapse. If the mouse is used to single-click the selected item and that item is closed, it will be expanded. If the selected item is single-clicked when it is open, it will be collapsed.

A tree control that has the **TVS_HASBUTTONS** style adds a button to the left side of each parent item. The user can click the button to expand or collapse the child items as an alternative to double-clicking the parent item. **TVS_HASBUTTONS** does not add buttons to items at the root of the hierarchy. To do so, you must combine **TVS_HASLINES**, **TVS_LINESATROOT**, and **TVS_HASBUTTONS**.

The **TVS_EDITLABELS** style makes it possible for the user to edit the labels of tree control items. For more information about editing labels, see Tree Control Label Editing later in this topic.

The **TVS_NOTOOLTIPS** style disables the automatic tool tip feature of tree view controls. This feature automatically displays a tool tip, containing the title of the item under the mouse cursor, if the entire title is not currently visible.

See Also

Reference

Using CTreeCtrl

Concepts

Controls (MFC)

Tree Control Parent and Child Items

Any item in a tree control (CTreeCtrl) can have a list of subitems, which are called child items, associated with it. An item that has one or more child items is called a parent item. A child item is displayed below its parent item and is indented to indicate it is subordinate to the parent. An item that has no parent is at the top of the hierarchy and is called a root item.

At any given time, the state of a parent item's list of child items can be either expanded or collapsed. When the state is expanded, the child items are displayed below the parent item. When it is collapsed, the child items are not displayed. The list automatically toggles between the expanded and collapsed states when the user double-clicks the parent item or, if the parent has the **TVS_HASBUTTONS** style, when the user clicks the button associated with the parent item. An application can expand or collapse the child items by using the Expand member function.

You add an item to a tree control by calling the InsertItem member function. This function returns a handle of the **HTREEITEM** type, which uniquely identifies the item. When adding an item, you must specify the handle of the new item's parent item. If you specify **NULL** or the **TVI_ROOT** value instead of a parent item handle in the TVINSERTSTRUCT structure or *hParent* parameter, the item is added as a root item.

A tree control sends a TVN_ITEMEXPANDING notification message when a parent item's list of child items is about to be expanded or collapsed. The notification gives you the opportunity to prevent the change or to set any attributes of the parent item that depend on the state of the list of child items. After changing the state of the list, the tree control sends a TVN_ITEMEXPANDED notification message.

When a list of child items is expanded, it is indented relative to the parent item. You can set the amount of indentation by using the SetIndent member function or retrieve the current amount by using the GetIndent member function.

See Also

Reference

Using CTreeCtrl

Concepts

Controls (MFC)

Tree Control Item Position

An item's initial position is set when the item is added to the tree control (CTreeCtrl) by using the **InsertItem** member function. The member function call specifies the handle of the parent item and the handle of the item after which the new item is to be inserted. The second handle must identify either a child item of the given parent or one of these values: **TVI_FIRST**, **TVI_LAST**, or **TVI_SORT**.

When **TVI_FIRST** or **TVI_LAST** is specified, the tree control places the new item at the beginning or end of the given parent item's list of child items. When **TVI_SORT** is specified, the tree control inserts the new item into the list of child items in alphabetical order based on the text of the item labels.

You can put a parent item's list of child items into alphabetical order by calling the SortChildren member function. This function includes a parameter that specifies whether all levels of child items descending from the given parent item are also sorted in alphabetical order.

The SortChildrenCB member function allows you to sort child items based on criteria that you define. When you call this function, you specify an application-defined callback function that the tree control can call whenever the relative order of two child items needs to be decided. The callback function receives two 32-bit application-defined values for the items being compared and a third 32-bit value that you specify when calling **SortChildrenCB**.

See Also

Reference

Using CTreeCtrl

Concepts

Controls (MFC)

Tree Control Item Labels

You typically specify the text of an item's label when adding the item to the tree control (CTreeCtrl). The **InsertItem** member function can pass a TVITEM structure that defines the item's properties, including a string containing the text of the label. **InsertItem** has several overloads that can be called with various combinations of parameters.

A tree control allocates memory for storing each item; the text of the item labels takes up a significant portion of this memory. If your application maintains a copy of the strings in the tree control, you can decrease the memory requirements of the control by specifying the **LPSTR_TEXTCALLBACK** value in the **pszText** member of **TV_ITEM** or the *lpszItem* parameter instead of passing actual strings to the tree control. Using **LPSTR_TEXTCALLBACK** causes the tree control to retrieve the text of an item's label from the application whenever the item needs to be redrawn. To retrieve the text, the tree control sends a TVN_GETDISPINFO notification message, which includes the address of a NMTVDISPINFO structure. You must respond by setting the appropriate members of the included structure.

A tree control uses memory allocated from the heap of the process that creates the tree control. The maximum number of items in a tree control is based on the amount of memory available in the heap. Each item takes 64 bytes.

See Also

Reference

Using CTreeCtrl

Concepts

Controls (MFC)

Tree Control Label Editing

The user can directly edit the labels of items in a tree control (CTreeCtrl) that has the **TVS_EDITLABELS** style. The user begins editing by clicking the label of the item that has the focus. An application begins editing by using the EditLabel member function. The tree control sends the notification when editing begins and when it is canceled or completed. When editing is completed, you are responsible for updating the item's label, if appropriate.

When label editing begins, a tree control sends a TVN_BEGINLABELEDIT notification message. By processing this notification, you can allow editing of some labels and prevent editing of others. Returning 0 allows editing, and returning nonzero prevents it.

When label editing is canceled or completed, a tree control sends a TVN_ENDLABELEDIT notification message. The *lParam* parameter is the address of a NMTVDISPINFO structure. The **item** member is a TVITEM structure that identifies the item and includes the edited text. You are responsible for updating the item's label, if appropriate, perhaps after validating the edited string. The **pszText** member of **TV_ITEM** is 0 if editing is canceled.

During label editing, typically in response to the TVN_BEGINLABELEDIT notification message, you can get a pointer to the edit control used for label editing by using the GetEditControl member function. You can call the edit control's SetLimitText member function to limit the amount of text a user can enter or subclass the edit control to intercept and discard invalid characters. Note, however, that the edit control is displayed only *after* **TVN_BEGINLABELEDIT** is sent.

See Also

Reference

Using CTreeCtrl

Concepts

Controls (MFC)

Tree Control Item States Overview

Each item in a tree control (CTreeCtrl) has a current state. For example, an item can be selected, disabled, expanded, and so on. For the most part, the tree control automatically sets an item's state to reflect user actions, such as selection of an item. However, you can also set an item's state by using the SetItemState member function and retrieve the current state of an item by using the GetItemState member function. For a complete list of item states, see Tree-View Control Constants in the *Platform SDK*.

An item's current state is specified by the *nState* parameter. A tree control might change an item's state to reflect a user action, such as selecting the item or setting the focus to the item. In addition, an application might change an item's state to disable or hide the item or to specify an overlay image or state image.

When you specify or change an item's state, the *nStateMask* parameter specifies which state bits to set, and the *nState* parameter contains the new values for those bits. For example, the following example changes the current state of a parent item (specified by `hParentItem`) in a **CTreeCtrl** object (`m_treeCtrl`) to **TVIS_EXPANDPARTIAL**:

```
TVITEM curItem;
HTREEITEM hParentItem;

hParentItem= m_treeCtrl.GetSelectedItem();

//modify the parent item to keep the '+' sign
curItem.mask= TVIF_HANDLE;
curItem.hItem= hParentItem;
m_treeCtrl.GetItem(&curItem);

curItem.mask= TVIF_STATE;
curItem.state= TVIS_EXPANDPARTIAL;
curItem.stateMask= TVIS_EXPANDPARTIAL;
m_treeCtrl.SetItem(&curItem);
```

Another example of changing the state would be to set an item's overlay image. To accomplish this, *nStateMask* must include the **TVIS_OVERLAYMASK** value, and *nState* must include the one-based index of the overlay image shifted left eight bits by using the INDEXTOOVERLAYMASK macro. The index can be 0 to specify no overlay image. The overlay image must have been added to the tree control's list of overlay images by a previous call to the CImageList::SetOverlayImage function. The function specifies the one-based index of the image to add; this is the index used with the **INDEXTOOVERLAYMASK** macro. A tree control can have up to four overlay images.

To set an item's state image, *nStateMask* must include the **TVIS_STATEIMAGEMASK** value, and *nState* must include the one-based index of the state image shifted left 12 bits by using the INDEXTOSTATEIMAGEMASK macro. The index can be 0 to specify no state image. For more information about overlay and state images, see Tree Control Image Lists.

See Also

Reference

Using CTreeCtrl

Concepts

Controls (MFC)

Tree Control Image Lists

Each item in a tree control (CTreeCtrl) can have a pair of bitmapped images associated with it. The images appear on the left side of an item's label. One image is displayed when the item is selected, and the other is displayed when the item is not selected. For example, an item might display an open folder when it is selected and a closed folder when it is not selected.

To use item images, you must create an image list by constructing a CImageList object and using the CImageList::Create function to create the associated image list. Then add the desired bitmaps to the list, and associate the list with the tree control by using the SetImageList member function. By default, all items display the first image in the image list for both the selected and nonselected states. You can change the default behavior for a particular item by specifying the indexes of the selected and nonselected images when adding the item to the tree control using the InsertItem member function. You can change the indexes after adding an item by using the SetItemImage member function.

A tree control's image lists can also contain overlay images, which are designed to be superimposed on item images. A nonzero value in bits 8 through 11 of a tree control item's state specifies the one-based index of an overlay image (0 indicates no overlay image). Because a 4-bit, one-based index is used, overlay images must be among the first 15 images in the image lists. For more information about tree control item states, see Tree Control Item States Overview earlier in this topic.

If a state image list is specified, a tree control reserves space to the left of each item's icon for a state image. An application can use state images, such as checked and cleared check boxes, to indicate application-defined item states. A nonzero value in bits 12 through 15 specifies the one-based index of a state image (0 indicates no state image).

By specifying the **I_IMAGECALLBACK** value instead of the index of an image, you can delay specifying the selected or nonselected image until the item is about to be redrawn. **I_IMAGECALLBACK** directs the tree control to query the application for the index by sending the TVN_GETDISPINFO notification message.

The GetImageList member function retrieves the handle of a tree control's image list. This function is useful if you need to add more images to the list. For more information about image lists, see Using CImageList, CImageList in the *MFC Reference*, and Image Lists in the *Platform SDK*.

See Also

Reference

Using CTreeCtrl

Concepts

Controls (MFC)

Tree Control Item Selection

When the selection changes from one item to another, a tree control (CTreeCtrl) sends TVN_SELCHANGING and TVN_SELCHANGED notification messages. Both notifications include a value that specifies whether the change is the result of a mouse click or a keystroke. The notifications also include information about the item that is gaining the selection and the item that is losing the selection. You can use this information to set item attributes that depend on the selection state of the item. Returning **TRUE** in response to **TVN_SELCHANGING** prevents the selection from changing; returning **FALSE** allows the change.

An application can change the selection by calling the SelectItem member function.

See Also

Reference

Using CTreeCtrl

Concepts

Controls (MFC)

Tree Control Drag-and-Drop Operations

A tree control (CTreeCtrl) sends a notification when the user starts to drag an item. The control sends a TVN_BEGINDRAG notification message when the user begins dragging an item with the left mouse button and a TVN_BEGINRDRAG notification message when the user begins dragging with the right button. You can prevent a tree control from sending these notifications by giving the tree control the **TVS_DISABLEDRAGDROP** style.

You obtain an image to display during a drag operation by calling the CreateDragImage member function. The tree control creates a dragging bitmap based on the label of the item being dragged. Then the tree control creates an image list, adds the bitmap to it, and returns a pointer to the CImageList object.

You must provide the code that actually drags the item. This typically involves using the dragging capabilities of the image list functions and processing the WM_MOUSEMOVE and WM_LBUTTONUP (or WM_RBUTTONUP) messages sent after the drag operation has begun. For more information about the image list functions, see CImageList in the *MFC Reference* and Image Lists in the *Platform SDK*. For more information about dragging a tree control item, see Dragging the Tree View Item, also in the *Platform SDK*.

If items in a tree control are to be the targets of a drag-and-drop operation, you need to know when the mouse cursor is on a target item. You can find out by calling the HitTest member function. You specify either a point and integer, or the address of a TVHITTESTINFO structure that contains the current coordinates of the mouse cursor. When the function returns, the integer or structure contains a flag indicating the location of the mouse cursor relative to the tree control. If the cursor is over an item in the tree control, the structure contains the handle of the item as well.

You can indicate that an item is the target of a drag-and-drop operation by calling the SetItem member function to set the state to the **TVIS_DROPHILITED** value. An item that has this state is drawn in the style used to indicate a drag-and-drop target.

See Also

Reference

Using CTreeCtrl

Concepts

Controls (MFC)

Tree Control Item Information

Tree controls (CTreeCtrl) have a number of member functions that retrieve information about items in the control. The GetItem member function retrieves some or all of the data associated with an item. This data could include the item's text, state, images, count of child items, and an application-defined 32-bit data value. There is also a SetItem function that can set some or all of the data associated with an item.

The GetItemState, GetItemText, GetItemData, and GetItemImage member functions retrieve individual attributes of an item. Each of these functions has a corresponding Set function for setting the attributes of an item.

The GetNextItem member function retrieves the tree control item that bears the specified relationship to the current item. This function can retrieve an item's parent, the next or previous visible item, the first child item, and so on. There are also member functions to traverse the tree: GetRootItem, GetFirstVisibleItem, GetNextVisibleItem, GetPrevVisibleItem, GetChildItem, GetNextSiblingItem, GetPrevSiblingItem, GetParentItem, GetSelectedItem, and GetDropHilightItem.

The GetItemRect member function retrieves the bounding rectangle for a tree control item. The GetCount and GetVisibleCount member functions retrieve a count of the items in a tree control and a count of the items that are currently visible in the tree control's window, respectively. You can ensure that a particular item is visible by calling the EnsureVisible member function.

See Also

Reference

Using CTreeCtrl

Concepts

Controls (MFC)

Tree Control Notification Messages

A tree control (CTreeCtrl) sends the following notification messages as **WM_NOTIFY** messages:

Notification message	Description
TVN_BEGINDRAG	Signals the start of a drag-and-drop operation
TVN_BEGINLABELEDIT	Signals the start of in-place label editing
TVN_BEGINRDRAG	Signals the start of a drag-and-drop operation, using the right mouse button
TVN_DELETEITEM	Signals the deletion of a specific item
TVN_ENDLABELEDIT	Signals the end of label editing
TVN_GETDISPINFO	Requests information that the tree control requires to display an item
TVN_ITEMEXPANDED	Signals that a parent item's list of child items was expanded or collapsed
TVN_ITEMEXPANDING	Signals that a parent item's list of child items is about to be expanded or collapsed
TVN_KEYDOWN	Signals a keyboard event
TVN_SELCHANGED	Signals that the selection has changed from one item to another
TVN_SELCHANGING	Signals that the selection is about to be changed from one item to another
TVN_SETDISPINFO	Notification to update the information maintained for an item

See Also

Reference

Using CTreeCtrl

Concepts

Controls (MFC)

Control Bars

"Control bar" is the general name for toolbars, status bars, and dialog bars. MFC classes **CToolBar**, **CStatusBar**, **CDialogBar**, **COleResizeBar**, and **CReBar** derive from class CControlBar, which implements their common functionality.

Control bars are windows that display rows of controls with which users can select options, execute commands, or obtain program information. Types of control bars include toolbars, dialog bars, and status bars.

- Toolbars, in class CToolBar
- Status bars, in class CStatusBar
- Dialog bars, in class CDialogBar
- Rebars, in class CReBar

Important
As of MFC version 4.0, toolbars, status bars, and tool tips are implemented using system functionality implemented in the comctl32.dll instead of the previous implementation specific to MFC. In MFC version 6.0, **CReBar**, which also wraps comctl32.dll functionality, was added.

Brief introductions to the control-bar types follow. For further information, see the links below.

Control Bars

Control bars greatly enhance a program's usability by providing quick, one-step command actions. Class **CControlBar** provides the common functionality of all toolbars, status bars, and dialog bars. **CControlBar** provides the functionality for positioning the control bar in its parent frame window. Because a control bar is usually a child window of a parent frame window, it is a "sibling" to the client view or MDI client of the frame window. A control-bar object uses information about its parent window's client rectangle to position itself. Then it alters the parent's remaining client-window rectangle so that the client view or MDI client window fills the rest of the client window.

Note
If a button on the control bar doesn't have a **COMMAND** or **UPDATE_COMMAND_UI** handler, the framework automatically disables the button.

Toolbars

A toolbar is a control bar that displays a row of bitmapped buttons that carry out commands. Pressing a toolbar button is equivalent to choosing a menu item; it calls the same handler mapped to a menu item if that menu item has the same ID as the toolbar button. The buttons can be configured to appear and behave as pushbuttons, radio buttons, or check boxes. A toolbar is usually aligned to the top of a frame window, but an MFC toolbar can "dock" to any side of its parent window or float in its own mini-frame window. A toolbar can also "float" and you can change its size and drag it with a mouse. A toolbar can also display tool tips as the user moves the mouse over the toolbar's buttons. A tool tip is a tiny popup window that briefly describes the button's purpose.

Note
As of MFC version 4.0, class CToolBar uses the Windows toolbar common control. A **CToolBar** contains a CToolBarCtrl. Older toolbars are still supported, however. See the article ToolBars.

Status Bars

A status bar is a control bar that contains text-output panes, or "indicators." The output panes are commonly used as message lines and as status indicators. Message line examples include the command help-message lines that briefly explain the selected menu or toolbar command in the leftmost pane of the default status bar created by the MFC Application Wizard. Status indicator examples include the SCROLL LOCK, NUM LOCK, and other keys. Status bars are usually aligned to the bottom of a frame window. See class CStatusBar and class CStatusBarCtrl.

Dialog Bars

A dialog bar is a control bar, based on a dialog-template resource, with the functionality of a modeless dialog box. Dialog bars can contain Windows, custom, or ActiveX controls. As in a dialog box, the user can tab among the controls. Dialog bars can be aligned to the top, bottom, left, or right side of a frame window and they can also be floated in their own frame window. See class CDialogBar.

Rebars

A rebar is a control bar that provides docking, layout, state, and persistence information for rebar controls. A rebar object can contain a variety of child windows, usually other controls, including edit boxes, toolbars, and list boxes. A rebar object can display its child windows over a specified bitmap. It can be automatically or manually resized by clicking or dragging its gripper bar. See class CReBar.

See Also

Other Resources

General Concepts
User Interface

Dialog Bars

A dialog bar is a toolbar, a kind of control bar that can contain any kind of control. Because it has the characteristics of a modeless dialog box, a CDialogBar object provides a more powerful toolbar.

There are several key differences between a toolbar and a **CDialogBar** object. A **CDialogBar** object is created from a dialog-template resource, which you can create with the Visual C++ dialog editor and which can contain any kind of Windows control. The user can tab from control to control. And you can specify an alignment style to align the dialog bar with any part of the parent frame window or even to leave it in place if the parent is resized. The following figure shows a dialog bar with a variety of controls.

A Dialog Bar

In other respects, working with a **CDialogBar** object is like working with a modeless dialog box. Use the dialog editor to design and create the dialog resource.

One of the virtues of dialog bars is that they can include controls other than buttons.

While it is normal to derive your own dialog classes from **CDialog**, you do not typically derive your own class for a dialog bar. Dialog bars are extensions to a main window and any dialog-bar control-notification messages, such as **BN_CLICKED** or **EN_CHANGE**, will be sent to the parent of the dialog bar, the main window.

See Also

Tasks

CTRLBARS Sample: Illustrates Custom Control Bars

Other Resources

General Concepts

User Interface

Dialog Boxes

Applications for Windows frequently communicate with the user through dialog boxes. Class CDialog provides an interface for managing dialog boxes, the Visual C++ dialog editor makes it easy to design dialog boxes and create their dialog-template resources, and Code wizards simplify the process of initializing and validating the controls in a dialog box and of gathering the values entered by the user.

Dialog boxes contain controls, including:

- Windows common controls such as edit boxes, pushbuttons, list boxes, combo boxes, tree controls, list controls, and progress indicators.
- ActiveX controls.
- Owner-drawn controls: controls that you are responsible for drawing in the dialog box.

Most dialog boxes are modal, which require the user to close the dialog box before using any other part of the program. But it is possible to create modeless dialog boxes, which let users work with other windows while the dialog box is open. MFC supports both kinds of dialog box with class **CDialog**. The controls are arranged and managed using a dialog-template resource, created with the dialog editor.

Property sheets, also known as tab dialog boxes, are dialog boxes that contain "pages" of distinct dialog-box controls. Each page has a file folder "tab" at the top. Clicking a tab brings that page to the front of the dialog box.

What do you want to know more about?

- Example: Displaying a Dialog Box via a Menu Command
- Dialog-box components in the framework
- Modal and modeless dialog boxes
- Property sheets and property pages in a dialog box
- Creating the dialog resource
- Creating a dialog class with Code Wizards
- Life cycle of a dialog box
- Dialog data exchange (DDX) and validation (DDV)
- Type-safe access to controls in a dialog box
- Mapping Windows messages to your class
- Commonly Overridden Member Functions
- Commonly Added Member Functions
- Common dialog classes
- Dialog boxes in OLE
- Create an application whose user interface is a dialog box: see the CMNCTRL1 or CMNCTRL2 sample programs. The Application Wizard provides this option as well.
- Samples

Smart Device Developer Notes

For information on Device Resolution Awareness in CDialog-derived classes, see AfxEnableDRA.

See Also

Other Resources

General Concepts
User Interface

Example: Displaying a Dialog Box via a Menu Command

This topic contains procedures to:

- Display a modal dialog box through a menu command.
- Display a modeless dialog box through a menu command.

Both sample procedures are for MFC applications and will work in an application you create with the MFC Application Wizard.

The procedures use the following names and values:

Item	Name or value
Application	DisplayDialog
Menu command	Test command on View menu; Command ID = ID_VIEW_TEST
Dialog box	Test dialog box; Class = CTestDialog; Header file = TestDialog.h; Variable = testdlg, ptestdlg
Command handler	OnViewTest

To display a modal dialog box

1. Create the menu command; see Creating Menus or Menu Items.
2. Create the dialog box; see Starting the Dialog Editor.
3. Add a class for your dialog box. See Adding a Class for more information.
4. In **Class View**, select the document class (CDisplayDialogDoc). In the **Properties** window, click the **Events** button. Double-click the ID of the menu command (ID_VIEW_TEST) in the left pane of the **Properties** window and select **Command**. In the right pane, click the down arrow and select **<Add> OnViewTest**.

   If you added the menu command to the mainframe of an MDI application, select the application class (CDisplayDialogApp) instead.
5. Add the following include statement to CDisplayDialogDoc.cpp (or CDisplayDialogApp.cpp) after the existing include statements:

   ```
   #include "TestDialog.h"
   ```
6. Add the following code to `OnViewTest` to implement the function:

   ```
   CTestDialog testdlg;
   testdlg.DoModal();
   ```

To display a modeless dialog box

1. Do the first four steps to display a modal dialog box, except select the view class (CDisplayDialogView) in step 4.
2. Edit DisplayDialogView.h:
   - Declare the dialog box class preceding the first class declaration:

     ```
     class CTestDialog;
     ```
   - Declare a pointer to the dialog box after the Attributes public section:

     ```
     CTestDialog* ptestdlg;
     ```

3. Edit DisplayDialogView.cpp:

   - Add the following include statement after the existing include statements:

     ```
     #include "TestDialog.h"
     ```

   - Add the following code to the constructor:

     ```
     ptestdlg = NULL;
     ```

   - Add the following code to the destructor:

     ```
     delete ptestdlg;
     ```

   - Add the following code to `OnViewTest` to implement the function:

     ```
     ptestdlg = new CTestDialog(this);
     ptestdlg ->Create(CTestDialog::IDD,this);
     ptestdlg ->ShowWindow(SW_SHOW);
     ```

Also, see the following Knowledge Base article:

- Q251059 : HOWTO: Provide Your Own Window Class Name for an MFC Dialog Box

See Also

Concepts

Dialog Boxes
Modal and Modeless Dialog Boxes

Dialog Sample List

See the following sample programs that illustrate dialog boxes and property sheets:

MDI Sample Application with Dialog Boxes

- SCRIBBLE

Modeless Dialog Box

- MODELESS

Property Sheet Dialog Box (Tab Dialog Box)

- PROPDLG
- CMNCTRL1
- CMNCTRL2

Application Based on a Dialog Box

- CMNCTRL1
- CMNCTRL2

Dialog-Box Controls

- CMNCTRL1
- CMNCTRL2
- CTRLTEST

Dialog-Like Form Views

- VIEWEX

In-Memory Dialog Template

- DLGTEMPL

See Also

Concepts

Dialog Boxes

Dialog-Box Components in the Framework

In the MFC framework, a dialog box has two components:

- A dialog-template resource that specifies the dialog box's controls and their placement.

  The dialog resource stores a dialog template from which Windows creates the dialog window and displays it. The template specifies the dialog box's characteristics, including its size, location, style, and the types and positions of the dialog box's controls. You will usually use a dialog template stored as a resource, but you can also create your own template in memory.

- A dialog class, derived from CDialog, to provide a programmatic interface for managing the dialog box.

  A dialog box is a window and will be attached to a Windows window when visible. When the dialog window is created, the dialog-template resource is used as a template for creating child window controls for the dialog box.

See Also

Concepts

Dialog Boxes
Life Cycle of a Dialog Box

Modal and Modeless Dialog Boxes

You can use class CDialog to manage two kinds of dialog boxes:

- *Modal dialog boxes*, which require the user to respond before continuing the program
- *Modeless dialog boxes*, which stay on the screen and are available for use at any time but permit other user activities

The resource editing and procedures for creating a dialog template are the same for modal and modeless dialog boxes.

Creating a dialog box for your program requires the following steps:

1. Use the dialog editor to design the dialog box and create its dialog-template resource.
2. Create a dialog class.
3. Connect the dialog resource's controls to message handlers in the dialog class.
4. Add data members associated with the dialog box's controls and to specify dialog data exchange and dialog data validations for the controls.

See Also

Concepts

Dialog Boxes
Life Cycle of a Dialog Box

Property Sheets and Property Pages (MFC)

An MFC dialog box can take on a "tab dialog" look by incorporating property sheets and property pages. Called a "property sheet" in MFC, this kind of dialog box, similar to many dialog boxes in Microsoft Word, Excel, and Visual C++, appears to contain a stack of tabbed sheets, much like a stack of file folders seen from front to back, or a group of cascaded windows. Controls on the front tab are visible; only the labeled tab is visible on the rear tabs. Property sheets are particularly useful for managing large numbers of properties or settings that fall fairly neatly into several groups. Typically, one property sheet can simplify a user interface by replacing several separate dialog boxes.

As of MFC version 4.0, property sheets and property pages are implemented using the common controls that come with Windows 95 and Windows NT version 3.51 and later.

Property sheets are implemented with classes CPropertySheet and CPropertyPage (described in the *MFC Reference*). **CPropertySheet** defines the overall dialog box, which can contain multiple "pages" based on **CPropertyPage**.

For information on creating and working with property sheets, see the topic Property Sheets.

See Also

Reference

Exchanging Data
Creating a Modeless Property Sheet
Handling the Apply Button

Concepts

Dialog Boxes
Life Cycle of a Dialog Box
Property Sheets and Property Pages in MFC

Creating the Dialog Resource

To design the dialog box and create the dialog resource, you use the dialog editor. In the dialog editor, you can:

- Adjust the size and location your dialog box will have when it appears.
- Drag various kinds of controls from a controls palette and drop them where you want them in the dialog box.
- Position the controls with alignment buttons on the toolbar.
- Test your dialog box by simulating the appearance and behavior it will have in your program. In Test mode, you can manipulate the dialog box's controls by typing text in text boxes, clicking pushbuttons, and so on.

When you finish, your dialog-template resource is stored in your application's resource script file. You can edit it later if needed. For a full description of how to create and edit dialog resources, see the dialog editor topics. This technique is also used to create the dialog-template resources for CFormView and CRecordView classes.

When the dialog box's appearance suits you, create a dialog class and map its messages, as discussed in Creating a Dialog Class with Code Wizards.

See Also

Concepts

Dialog Boxes

Life Cycle of a Dialog Box

Creating a Dialog Class with Code Wizards

The following table lists dialog-related tasks that Code Wizards help you manage.

Dialog-Related Tasks

Task	Apply to . . .
Create a new CDialog-derived class to manage your dialog box.	Each dialog box.
Map Windows messages to your dialog class.	Each message you want handled.
Declare class member variables to represent the controls in the dialog box.	Each control that yields a text or numeric value you want to access from your program.
Specify how data is to be exchanged between the controls and the member variables.	Each control you want to access from your program.
Specify validation rules for the member variables.	Each control that yields a text or numeric value, if desired.

What do you want to know more about?

- Creating your dialog class
- Handling Windows messages in your dialog box
- Dialog data exchange and validation

See Also

Concepts

Dialog Boxes

Life Cycle of a Dialog Box

Creating Your Dialog Class

For each dialog box in your program, create a new dialog class to work with the dialog resource.

Adding a Class explains how to create a new dialog class. When you create a dialog class with the Add Class Wizard, it writes the following items in the .H and .CPP files you specify:

In the .H file:

- A class declaration for the dialog class. The class is derived from CDialog.

In the .CPP file:

- A message map for the class.
- A standard constructor for the dialog box.
- An override of the DoDataExchange member function. Edit this function. It is used for dialog data exchange and validation capabilities as described later in Dialog data exchange and validation.

See Also

Reference

Creating a Dialog Class with Code Wizards

Concepts

Life Cycle of a Dialog Box

Life Cycle of a Dialog Box

During the life cycle of a dialog box, the user invokes the dialog box, typically inside a command handler that creates and initializes the dialog object, the user interacts with the dialog box, and the dialog box closes.

For modal dialog boxes, your handler gathers any data the user entered once the dialog box closes. Since the dialog object exists after its dialog window has closed, you can simply use the member variables of your dialog class to extract the data.

For modeless dialog boxes, you may often extract data from the dialog object while the dialog box is still visible. At some point, the dialog object is destroyed; when this happens depends on your code.

What do you want to know more about?

- Creating and displaying dialog boxes
- Creating modal dialog boxes
- Creating modeless dialog boxes
- Using a dialog template in memory
- Setting the dialog box's background color
- Initializing the dialog box
- Handling Windows messages in your dialog box
- Retrieving data from the dialog object
- Closing the dialog box
- Destroying the dialog box
- Dialog data exchange (DDX) and validation (DDV)
- Property sheet dialog boxes

See Also

Concepts

Dialog Boxes

Creating and Displaying Dialog Boxes

Creating a dialog object is a two-phase operation. First, construct the dialog object, then create the dialog window. Modal and modeless dialog boxes differ somewhat in the process used to create and display them. The following table lists how modal and modeless dialog boxes are normally constructed and displayed.

Dialog Creation

Dialog type	How to create it
Modeless	Construct **CDialog**, then call **Create** member function.
Modal	Construct **CDialog**, then call **DoModal** member function.

You can, if you want, create your dialog box from an in-memory dialog template that you have constructed rather than from a dialog template resource. This is an advanced topic, however.

See Also

Concepts

Life Cycle of a Dialog Box

Creating Modal Dialog Boxes

To create a modal dialog box, call either of the two public constructors declared in CDialog. Next, call the dialog object's DoModal member function to display the dialog box and manage interaction with it until the user chooses OK or Cancel. This management by **DoModal** is what makes the dialog box modal. For modal dialog boxes, **DoModal** loads the dialog resource.

See Also

Concepts

Life Cycle of a Dialog Box

Creating Modeless Dialog Boxes

For a modeless dialog box, you must provide your own public constructor in your dialog class. To create a modeless dialog box, call your public constructor and then call the dialog object's Create member function to load the dialog resource. You can call **Create** either during or after the constructor call. If the dialog resource has the property **WS_VISIBLE**, the dialog box appears immediately. If not, you must call its ShowWindow member function.

See Also

Concepts

Life Cycle of a Dialog Box

Using a Dialog Template in Memory

Instead of using the methods given in the Dialog Creation table, you can create either kind of dialog box indirectly from a dialog template in memory. For more information, see class CDialog in the *MFC Reference*.

See Also

Concepts

Life Cycle of a Dialog Box

Setting the Dialog Box's Background Color

You can set the background color of your dialog boxes by handling **WM_CTLCOLOR** messages for the dialog box window. The color you set is used for only the specified dialog box.

See Also

Reference

Handling Windows Messages in Your Dialog Box

Concepts

Life Cycle of a Dialog Box

Initializing the Dialog Box

After the dialog box and all of its controls are created but just before the dialog box (of either type) appears on the screen, the dialog object's OnInitDialog member function is called. For a modal dialog box, this occurs during the **DoModal** call. For a modeless dialog box, **OnInitDialog** is called when **Create** is called. You typically override **OnInitDialog** to initialize the dialog box's controls, such as setting the initial text of an edit box. You must call the **OnInitDialog** member function of the base class, **CDialog**, from your `OnInitDialog` override.

If you want to set your dialog box's background color (and that of all other dialog boxes in your application), see Setting the Dialog Box's Background Color.

See Also
Concepts
Life Cycle of a Dialog Box

Handling Windows Messages in Your Dialog Box

Dialog boxes are windows, so they can handle Windows messages if you supply the appropriate handler functions. When you create your dialog class with the Add Class Wizard, the wizard adds an empty message map to the class. Use the Properties window to map any Windows messages or commands you want your class to handle.

See Mapping Windows Messages to Your Dialog Class for more information.

See Also

Concepts

Life Cycle of a Dialog Box

Retrieving Data from the Dialog Object

The framework provides an easy way to initialize the values of controls in a dialog box and to retrieve values from the controls. The more laborious manual approach is to call functions such as the **SetDlgItemText** and **GetDlgItemText** member functions of class **CWnd**, which apply to control windows. With these functions, you access each control individually to set or get its value, calling functions such as **SetWindowText** and **GetWindowText**. The framework's approach automates both initialization and retrieval.

Dialog data exchange (DDX) lets you exchange data between the controls in the dialog box and member variables in the dialog object more easily. This exchange works both ways. To initialize the controls in the dialog box, you can set the values of data members in the dialog object, and the framework will transfer the values to the controls before the dialog box is displayed. Then you can at any time update the dialog data members with data entered by the user. At that point, you can use the data by referring to the data member variables.

You can also arrange for the values of dialog controls to be validated automatically with dialog data validation (DDV).

DDX and DDV are explained in more detail in Dialog Data Exchange and Validation.

For a modal dialog box, you can retrieve any data the user entered when **DoModal** returns **IDOK** but before the dialog object is destroyed. For a modeless dialog box, you can retrieve data from the dialog object at any time by calling **UpdateData** with the argument **TRUE** and then accessing dialog class member variables. This subject is discussed in more detail in Dialog Data Exchange and Validation.

See Also

Concepts

Life Cycle of a Dialog Box

Closing the Dialog Box

A modal dialog box closes when the user chooses one of its buttons, typically the OK button or the Cancel button. Choosing the OK or Cancel button causes Windows to send the dialog object a **BN_CLICKED** control-notification message with the button's ID, either **IDOK** or **IDCANCEL**. **CDialog** provides default handler functions for these messages: **OnOK** and **OnCancel**. The default handlers call the **EndDialog** member function to close the dialog window. You can also call **EndDialog** from your own code. For more information, see the EndDialog member function of class **CDialog** in the *MFC Reference*.

To arrange for closing and deleting a modeless dialog box, override **PostNcDestroy** and invoke the **delete** operator on the **this** pointer. Destroying the Dialog Box explains what happens next.

See Also
Concepts
Life Cycle of a Dialog Box

Destroying the Dialog Box

Modal dialog boxes are normally created on the stack frame and destroyed when the function that created them ends. The dialog object's destructor is called when the object goes out of scope.

Modeless dialog boxes are normally created and owned by a parent view or frame window — the application's main frame window or a document frame window. The default OnClose handler calls DestroyWindow, which destroys the dialog-box window. If the dialog box stands alone, with no pointers to it or other special ownership semantics, you should override PostNcDestroy to destroy the C++ dialog object. You should also override OnCancel and call **DestroyWindow** from within it. If not, the owner of the dialog box should destroy the C++ object when it is no longer necessary.

See Also
Concepts
Life Cycle of a Dialog Box

Dialog Data Exchange and Validation

Dialog data exchange (DDX) is an easy way to initialize the controls in your dialog box and to gather data input by the user. Dialog data validation (DDV) is an easy way to validate data entry in a dialog box. To take advantage of DDX and DDV in your dialog boxes, use the Add Member Variable Wizard to create the data members and set their data types and specify validation rules.

What do you want to know more about?

- Dialog data exchange
- Dialog data validation

See Also

Concepts

Dialog Boxes
Life Cycle of a Dialog Box

Dialog Data Exchange

If you use the DDX mechanism, you set the initial values of the dialog object's member variables, typically in your **OnInitDialog** handler or the dialog constructor. Immediately before the dialog is displayed, the framework's DDX mechanism transfers the values of the member variables to the controls in the dialog box, where they appear when the dialog box itself appears in response to **DoModal** or **Create**. The default implementation of **OnInitDialog** in **CDialog** calls the **UpdateData** member function of class **CWnd** to initialize the controls in the dialog box.

The same mechanism transfers values from the controls to the member variables when the user clicks the OK button (or whenever you call the **UpdateData** member function with the argument **TRUE**). The dialog data validation mechanism validates any data items for which you specified validation rules.

The following figure illustrates dialog data exchange.

Dialog Data Exchange

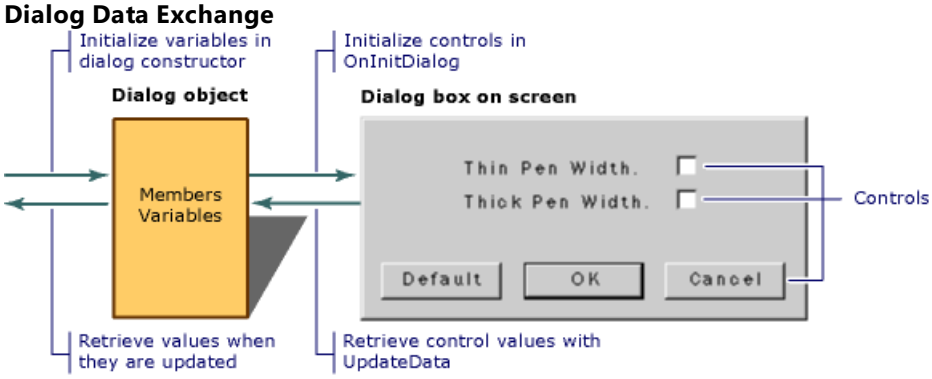

UpdateData works in both directions, as specified by the **BOOL** parameter passed to it. To carry out the exchange, **UpdateData** sets up a **CDataExchange** object and calls your dialog class's override of **CDialog**'s **DoDataExchange** member function. **DoDataExchange** takes an argument of type **CDataExchange**. The **CDataExchange** object passed to **UpdateData** represents the context of the exchange, defining such information as the direction of the exchange.

When you (or a Code wizard) override **DoDataExchange**, you specify a call to one DDX function per data member (control). Each DDX function knows how to exchange data in both directions based on the context supplied by the **CDataExchange** argument passed to your `DoDataExchange` by **UpdateData**.

MFC provides many DDX functions for different kinds of exchange. The following example shows a `DoDataExchange` override in which two DDX functions and one DDV function are called:

```
void CMyDialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);    // Call base class version
    DDX_Check(pDX, IDC_MY_CHECKBOX, m_bVar);
    DDX_Text(pDX, IDC_MY_TEXTBOX, m_strName);
    DDV_MaxChars(pDX, m_strName, 20);
}
```

The `DDX_` and `DDV_` lines are a data map. The sample DDX and DDV functions shown are for a check-box control and an edit-box control, respectively.

If the user cancels a modal dialog box, the **OnCancel** member function terminates the dialog box and **DoModal** returns the value **IDCANCEL**. In that case, no data is exchanged between the dialog box and the dialog object.

See Also

Concepts

Dialog Data Exchange and Validation
Life Cycle of a Dialog Box
Dialog Data Validation

Dialog Data Validation

You can specify validation in addition to data exchange by calling DDV functions, as shown in the example in Dialog Data Exchange. The **DDV_MaxChars** call in the example validates that the string entered in the text-box control is not longer than 20 characters. The DDV function typically alerts the user with a message box if the validation fails and puts the focus on the offending control so the user can reenter the data. A DDV function for a given control must be called immediately after the DDX function for the same control.

You can also define your own custom DDX and DDV routines. For details on this and other aspects of DDX and DDV, see MFC Technical Note 26.

The Add Member Variable Wizard will write all of the DDX and DDV calls in the data map for you.

See Also

Concepts

Dialog Data Exchange and Validation
Life Cycle of a Dialog Box
Dialog Data Exchange

Type-Safe Access to Controls in a Dialog Box

The controls in a dialog box can use the interfaces of MFC control classes such as **CListBox** and **CEdit**. You can create a control object and attach it to a dialog control. Then you can access the control through its class interface, calling member functions to operate on the control. The methods described here are designed to give you type-safe access to a control. This is especially useful for controls such as edit boxes and list boxes.

There are two approaches to making a connection between a control in a dialog box and a C++ control member variable in a **CDialog**-derived class:

- Without Code Wizards
- With Code Wizards

See Also

Concepts

Dialog Boxes

Type-Safe Access to Controls Without Code Wizards

The first approach to creating type-safe access to controls uses an inline member function to cast the return type of class **CWnd**'s **GetDlgItem** member function to the appropriate C++ control type, as in this example:

```
// Declared inline in class CMyDialog
CButton* GetMyCheckbox()
{
    return (CButton*)GetDlgItem(ID_MYCHECKBOX);
}
```

You can then use this member function to access the control in a type-safe manner with code similar to the following:

```
GetMyCheckbox()->SetState(TRUE);
```

See Also

Concepts

Type-Safe Access to Controls in a Dialog Box
Type-Safe Access to Controls With Code Wizards

Type-Safe Access to Controls With Code Wizards

If you are familiar with DDX features, you can use the Control property in the Add Member Variable Wizard to create type-safe access. This approach is easier than creating controls without code wizards.

If you simply want access to a control's value, DDX provides it. If you want to do more than access a control's value, use the Add Member Variable Wizard to add a member variable of the appropriate class to your dialog class. Attach this member variable to the Control property.

Member variables can have a Control property instead of a Value property. The Value property refers to the type of data returned from the control, such as **CString** or **int**. The Control property enables direct access to the control through a data member whose type is one of the control classes in MFC, such as **CButton** or **CEdit**.

Note
For a given control, you can, if you wish, have multiple member variables with the Value property and at most one member variable with the Control property. You can have only one MFC object mapped to a control because multiple objects attached to a control, or any other window, would lead to an ambiguity in the message map.

You can use this object to call any member functions for the control object. Such calls affect the control in the dialog box. For example, for a check-box control represented by a variable `m_checkboxDefault`, of type **CButton**, you could call:

```
m_checkboxDefault.SetState(TRUE);
```

Here the member variable `m_checkboxDefault` serves the same purpose as the member function `GetMyCheckbox` shown in Type-Safe Access to Controls Without Code Wizards. If the check box is not an auto check box, you would still need a handler in your dialog class for the **BN_CLICKED** control-notification message when the button is clicked.

For more information about controls, see Controls.

See Also

Concepts

Type-Safe Access to Controls in a Dialog Box
Life Cycle of a Dialog Box
Type-Safe Access to Controls Without Code Wizards

Mapping Windows Messages to Your Class

If you need your dialog box to handle Windows messages, override the appropriate handler functions. To do so, use the Properties window to map the messages to the dialog class. This writes a message-map entry for each message and adds the message-handler member functions to the class. Use the Visual C++ source code editor to write code in the message handlers.

You can also override member functions of CDialog and its base classes, especially CWnd.

What do you want to know more about?

- Message handling and mapping
- Commonly overridden member functions
- Commonly added member functions

See Also

Concepts

Dialog Boxes

Life Cycle of a Dialog Box

Commonly Overridden Member Functions

The following table lists the most likely member functions to override in your **CDialog**-derived class.

Commonly Overridden Member Functions of Class CDialog

Member function	Message it responds to	Purpose of the override
OnInitDialog	**WM_INITDIALOG**	Initialize the dialog box's controls.
OnOK	**BN_CLICKED** for button **IDOK**	Respond when the user clicks the OK button.
OnCancel	**BN_CLICKED** for button **IDCANCEL**	Respond when the user clicks the Cancel button.

OnInitDialog, **OnOK**, and **OnCancel** are virtual functions. To override them, you declare an overriding function in your derived dialog class using the Properties window.

OnInitDialog is called just before the dialog box is displayed. You must call the default **OnInitDialog** handler from your override — usually as the first action in the handler. By default, **OnInitDialog** returns **TRUE** to indicate that the focus should be set to the first control in the dialog box.

OnOK is typically overridden for modeless but not modal dialog boxes. If you override this handler for a modal dialog box, call the base class version from your override — to ensure that **EndDialog** is called — or call **EndDialog** yourself.

OnCancel is usually overridden for modeless dialog boxes.

For more information about these member functions, see class CDialog in the *MFC Reference* and the discussion on Life Cycle of a Dialog Box.

See Also

Concepts

Dialog Boxes
Commonly Added Member Functions

Commonly Added Member Functions

If your dialog box contains pushbuttons other than OK or Cancel, you need to write message-handler member functions in your dialog class to respond to the control-notification messages they generate. For an example, see the Scribble sample program. You can also handle control-notification messages from other controls in your dialog box.

See Also

Concepts

Dialog Boxes
Life Cycle of a Dialog Box
Commonly Overridden Member Functions

Common Dialog Classes

In addition to class CDialog, MFC supplies several classes derived from **CDialog** that encapsulate commonly used dialog boxes, as shown in the following table. The dialog boxes encapsulated are called the "common dialog boxes" and are part of the Windows common dialog library (COMMDLG.DLL). The dialog-template resources and code for these classes are provided in the Windows common dialog boxes that are part of Windows versions 3.1 and later.

Common Dialog Classes

Derived dialog class	Purpose
CColorDialog	Lets user select colors.
CFileDialog	Lets user select a filename to open or to save.
CFindReplaceDialog	Lets user initiate a find or replace operation in a text file.
CFontDialog	Lets user specify a font.
CPrintDialog	Lets user specify information for a print job.
CPrintDialogEx	Windows 2000 print property sheet.

For more information about the common dialog classes, see the individual class names in the *MFC Reference*. MFC also supplies a number of standard dialog classes used for OLE. For information about these classes, see the base class, COleDialog, in the *MFC Reference*.

Three other classes in MFC have dialog-like characteristics. For information about classes CFormView, CRecordView, and CDaoRecordView, see the classes in the *MFC Reference*. For information about class CDialogBar, see Dialog Bars.

See Also

Concepts

Dialog Boxes
Life Cycle of a Dialog Box
Dialog Boxes in OLE

Dialog Boxes in OLE

While a user runs an OLE-enabled application, there are times when the application needs information from the user in order to carry out the operation. The MFC OLE classes provide a number of dialog boxes to gather the required information. This topic lists the tasks handled by the OLE dialog boxes and the classes needed to display those dialog boxes. For details on OLE dialog boxes and the structures used to customize their behavior, see MFC Reference.

Insert Object

This dialog box allows the user to insert newly created or existing objects into the compound document. It also allows the user to choose to display the item as an icon and enables the Change Icon command button. Display this dialog box when the user chooses Insert Object from the Edit menu. Use the COleInsertDialog class to display this dialog box. Note that you cannot insert an MDI application into itself. An application that is a container/server cannot be inserted into itself unless it is an SDI application.

Paste Special

This dialog box allows the user to control the format used when pasting data into a compound document. The user can choose the format of the data, whether to embed or link the data, and whether to display it as an icon. Display this dialog box when the user chooses Paste Special from the Edit menu. Use the COlePasteSpecialDialog class to display this dialog box.

Change Icon

This dialog box allows the user to select which icon is displayed to represent the linked or embedded item. Display this dialog box when the user chooses Change Icon from the Edit menu or chooses the Change Icon button in either the Paste Special or Convert dialog boxes. Also display it when the user opens the Insert Object dialog box and chooses Display as Icon. Use the COleChangeIconDialog class to display this dialog box.

Convert

This dialog box allows the user to change the type of an embedded or linked item. For example, if you have embedded a metafile in a compound document and later want to use another application to modify the embedded metafile, you can use the Convert dialog box. This dialog box is usually displayed by clicking *item type* Object on the Edit menu and then, on the cascading menu, clicking Convert. Use the COleConvertDialog class to display this dialog box. For an example, run the MFC OLE sample OCLIENT.

Edit Links or Update Links

The Edit Links dialog box allows the user to change information about the source of a linked object. The Update Links dialog box verifies the sources of all the linked items in the current dialog box and displays the Edit Links dialog box if necessary. Display the Edit Links dialog box when the user chooses Links from the Edit menu. The Update Links dialog box is usually displayed when a compound document is first opened. Use either the COleLinksDialog or the COleUpdateDialog class, depending on which dialog box you want to display.

Server Busy or Server Not Responding

The Server Busy dialog box is displayed when the user attempts to activate an item and the server is currently unable to handle the request, usually because the server is in use by another user or task. The Server Not Responding dialog box is displayed if the server does not respond to the activation request at all. These dialog boxes are displayed via **COleMessageFilter**, based on an implementation of the OLE interface **IMessageFilter**, and the user can decide whether to attempt the activation request again. Use the COleBusyDialog class to display this dialog box.

See Also

Concepts

Dialog Boxes
Life Cycle of a Dialog Box
OLE in MFC

Document/View Architecture

By default, the MFC Application Wizard creates an application skeleton with a document class and a view class. MFC separates data management into these two classes. The document stores the data and manages printing the data and coordinates updating multiple views of the data. The view displays the data and manages user interaction with it, including selection and editing.

In this model, an MFC document object reads and writes data to persistent storage. The document may also provide an interface to the data wherever it resides (such as in a database). A separate view object manages data display, from rendering the data in a window to user selection and editing of data. The view obtains display data from the document and communicates back to the document any data changes.

While you can easily override or ignore the document/view separation, there are compelling reasons to follow this model in most cases. One of the best is when you need multiple views of the same document, such as both a spreadsheet and a chart view. The document/view model lets a separate view object represent each view of the data, while code common to all views (such as a calculation engine) can reside in the document. The document also takes on the task of updating all views whenever the data changes.

The MFC document/view architecture makes it easy to support multiple views, multiple document types, splitter windows, and other valuable user-interface features.

The parts of the MFC framework most visible both to the user and to you, the programmer, are the document and view. Most of your work in developing an application with the framework goes into writing your document and view classes. This article family describes:

- The purposes of documents and views and how they interact in the framework.
- What you must do to implement them.

At the heart of document/view are four key classes:

The CDocument (or COleDocument) class supports objects used to store or control your program's data and provides the basic functionality for programmer-defined document classes. A document represents the unit of data that the user typically opens with the Open command on the File menu and saves with the Save command on the File menu.

The CView (or one of its many derived classes) provides the basic functionality for programmer-defined view classes. A view is attached to a document and acts as an intermediary between the document and the user: the view renders an image of the document on the screen and interprets user input as operations upon the document. The view also renders the image for both printing and print preview.

CFrameWnd (or one of its variations) supports objects that provides the frame around one or more views of a document.

CDocTemplate (or CSingleDocTemplate or CMultiDocTemplate) supports an object that coordinates one or more existing documents of a given type and manages creating the correct document, view, and frame window objects for that type.

The following figure shows the relationship between a document and its view.

Document and View

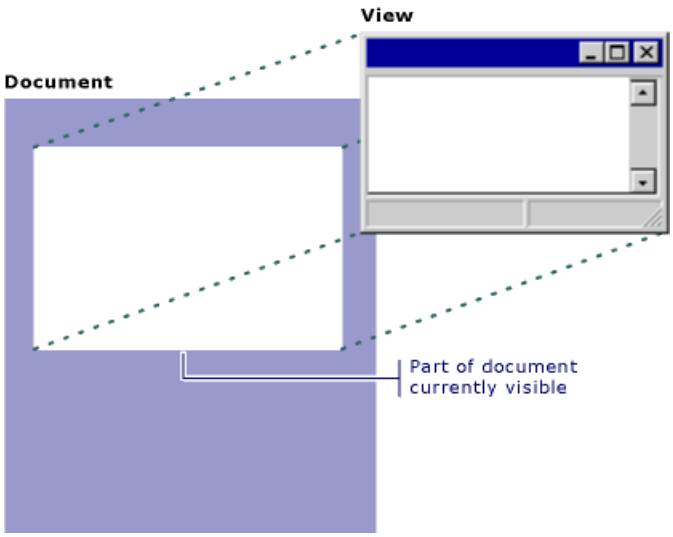

The document/view implementation in the class library separates the data itself from its display and from user operations on

the data. All changes to the data are managed through the document class. The view calls this interface to access and update the data.

Documents, their associated views, and the frame windows that frame the views are created by a document template. The document template is responsible for creating and managing all documents of one document type.

What do you want to know more about?

- A portrait of the document/view architecture
- Advantages of the document/view architecture
- Document and view classes created by the Application Wizard
- Alternatives to the document/view architecture
- Adding Multiple Views to a Single Document
- Using Documents
- Using Views
- Multiple Document Types, Views, and Frame Windows
- Initializing and cleaning up documents and views
- Initialize your own additions to document & view classes
- Using database classes with documents and views
- Using database classes without documents and views
- Samples

See Also

Concepts

Windows
Frame Windows
Document Templates and the Document/View Creation Process
Document/View Creation
Creating New Documents, Windows, and Views

Other Resources

General Concepts
User Interface

Document/View Sample List

See the following sample programs that illustrate using MFC's document/view architecture in interesting ways:

Document/View Variations

- MDI
- SCRIBBLE
- VIEWEX

Dialog-Box Interface Replaces Document/View

- CMNCTRL1
- CMNCTRL2

Using COleDocument and Its Derived Classes

- CONTAINER
- HIERSVR
- OCLIENT

See Also

Concepts

Document/View Architecture

A Portrait of the Document/View Architecture

Documents and views are paired in a typical MFC application. Data is stored in the document, but the view has privileged access to the data. The separation of document from view separates the storage and maintenance of data from its display.

Gaining Access to Document Data from the View

The view accesses its document's data either with the GetDocument function, which returns a pointer to the document, or by making the view class a C++ **friend** of the document class. The view then uses its access to the data to obtain the data when it is ready to draw or otherwise manipulate it.

For example, from the view's OnDraw member function, the view uses **GetDocument** to obtain a document pointer. Then it uses that pointer to access a **CString** data member in the document. The view passes the string to the **TextOut** function. To see the code for this example, see Drawing in a View.

User Input to the View

The view might also interpret a mouse click within itself as either selection or editing of data. Similarly it might interpret keystrokes as data entry or editing. Suppose the user types a string in a view that manages text. The view obtains a pointer to the document and uses the pointer to pass the new data to the document, which stores it in some data structure.

Updating Multiple Views of the Same Document

In an application with multiple views of the same document — such as a splitter window in a text editor — the view first passes the new data to the document. Then it calls the document's UpdateAllViews member function, which tells all views of the document to update themselves, reflecting the new data. This synchronizes the views.

What do you want to know more about?

- Advantages of the document/view architecture
- Alternatives to the document/view architecture

See Also

Concepts

Document/View Architecture

Advantages of the Document/View Architecture

The key advantage to using the MFC document/view architecture is that the architecture supports multiple views of the same document particularly well. (If you don't need multiple views and the small overhead of document/view is excessive in your application, you can avoid the architecture. Alternatives to the Document/View Architecture.)

Suppose your application lets users view numerical data either in spreadsheet form or in chart form. A user might want to see simultaneously both the raw data, in spreadsheet form, and a chart that results from the data. You display these separate views in separate frame windows or in splitter panes within a single window. Now suppose the user can edit the data in the spreadsheet and see the changes instantly reflected in the chart.

In MFC, the spreadsheet view and the chart view would be based on different classes derived from CView. Both views would be associated with a single document object. The document stores the data (or perhaps obtains it from a database). Both views access the document and display the data they retrieve from it.

When a user updates one of the views, that view object calls **CDocument::UpdateAllViews**. That function notifies all of the document's views, and each view updates itself using the latest data from the document. The single call to **UpdateAllViews** synchronizes the different views.

This scenario would be difficult to code without the separation of data from view, particularly if the views stored the data themselves. With document/view, it's easy. The framework does most of the coordination work for you.

What do you want to know more about?

- Alternatives to document/view
- CDocument
- CView
- CDocument::UpdateAllViews
- CView::GetDocument

See Also

Concepts

Document/View Architecture

Document and View Classes Created by the MFC Application Wizard

The MFC Application Wizard gives you a head start on your program development by creating skeletal document and view classes for you. You can then map commands and messages to these classes and use the Visual C++ source code editor to write their member functions.

The document class created by the MFC Application Wizard is derived from class CDocument. The view class is derived from CView. The names that the Application Wizard gives these classes and the files that contain them are based on the project name you supply in the Application Wizard dialog box. In the Application Wizard, you can use the Generated Classes page to alter the default names.

Some applications might need more than one document class, view class, or frame-window class. For more information, see Multiple Document Types, Views, and Frame Windows.

See Also

Concepts

Document/View Architecture

Alternatives to the Document/View Architecture

MFC applications normally use the document/view architecture to manage information, file formats, and the visual representation of data to users. For the majority of desktop applications, the document/view architecture is an appropriate and efficient application architecture. This architecture separates data from viewing and, in most cases, simplifies your application and reduces redundant code.

However, the document/view architecture is not appropriate for some situations. Consider these examples:

- If you are porting an application written in C for Windows, you might want to complete your port before adding document/view support to your application.
- If you are writing a lightweight utility, you might find that you can do without the document/view architecture.
- If your original code already mixes data management with data viewing, moving the code to the document/view model is not worth the effort because you must separate the two. You might prefer to leave the code as is.

To create an application that does not use the document/view architecture, clear the **Document/View architecture support** check box in step 1 of the MFC Application Wizard. See MFC Application Wizard for details.

Note
Dialog-based applications produced by the MFC Application Wizard do not use the document/view architecture, so the **Document/View architecture support** check box is disabled if you select the dialog application type.

The Visual C++ wizards, as well as the source and dialog editors, work with the generated application just as they would with any other Wizard-generated application. The application can support toolbars, scrollbars, and a status bar, and has an **About** box. Your application will not register any document templates, and it will not contain a document class.

Note that your generated application has a view class, **CChildView**, derived from **CWnd**. MFC creates and positions one instance of the view class within the frame windows created by your application. MFC still enforces using a view window, because it simplifies positioning and managing the application's content. You can add painting code to the **OnPaint** member of this class. Your code should add scrollbars to the view rather than to the frame.

Because the document/view architecture provided by MFC is responsible for implementing many of an application's basic features, its absence in your project means that you are responsible for implementing many important features of your application:

- As provided by the MFC Application Wizard, the menu for your application contains only **New** and **Exit** commands on the **File** menu. (The **New** command is supported only for MDI applications, not SDI applications without Document/View support.) The generated menu resource will not support an MRU (most recently used) list.
- You must add handler functions and implementations for any commands that your application will support, including **Open** and **Save** on the **File** menu. MFC normally provides code to support these features, but that support is tightly bound to the document/view architecture.
- The toolbar for your application, if you requested one, will be minimal.

It is strongly recommended that you use the MFC Application Wizard to create applications without the document/view architecture, because the wizard guarantees a correct MFC architecture. However, if you must avoid using the wizard, here are several approaches for bypassing the document/view architecture in your code:

- Treat the document as an unused appendage and implement your data management code in the view class, as suggested above. Overhead for the document is relatively low. A single CDocument object incurs a small amount of overhead by itself, plus the small overhead of **CDocument**'s base classes, CCmdTarget and CObject. Both of the latter classes are small.

  Declared in **CDocument**:

  - Two **CString** objects.
  - Three **BOOL**s.
  - One **CDocTemplate** pointer.

- One **CPtrList** object, which contains a list of the document's views.

Additionally, the document requires the amount of time to create the document object, its view objects, a frame window, and a document template object.

- Treat both the document and view as unused appendages. Put your data management and drawing code in the frame window rather than the view. This approach is closer to the C-language programming model.
- Override the parts of the MFC framework that create the document and view to eliminate creating them at all. The document creation process begins with a call to **CWinApp::AddDocTemplate**. Eliminate that call from your application class's **InitInstance** member function and, instead, create a frame window in **InitInstance** yourself. Put your data management code in your frame window class. The document/view creation process is illustrated in Document/View Creation. This is more work and requires a deeper understanding of the framework, but it frees you entirely of the document/view overhead.

The article MFC: Using Database Classes Without Documents and Views gives more concrete examples of document/view alternatives in the context of database applications.

See Also

Concepts

Document/View Architecture

Using Documents

Working together, documents and views:

- Contain, manage, and display your application-specific data.
- Provide an interface consisting of document data variables for manipulating the data.
- Participate in writing and reading files.
- Participate in printing.
- Handle most of your application's commands and messages.

The document is particularly involved in managing data. Store your data, normally, in document class member variables. The view uses these variables to access the data for display and update. The document's default serialization mechanism manages reading and writing the data to and from files. Documents can also handle commands (but not Windows messages other than **WM_COMMAND**).

What do you want to know more about?

- Deriving a document class from CDocument
- Managing data with document data variables
- Serializing data to and from files
- Bypassing the serialization mechanism
- Handling commands in the document
- The OnNewDocument member function
- The DeleteContents member function

See Also
Concepts
Document/View Architecture

Deriving a Document Class from CDocument

Documents contain and manage your application's data. To use the MFC Application Wizard-supplied document class, you must do the following:

- Derive a class from **CDocument** for each type of document.
- Add member variables to store each document's data.
- Override **CDocument**'s **Serialize** member function in your document class. **Serialize** writes and reads the document's data to and from disk.

Other Document Functions Often Overridden

You may also want to override other **CDocument** member functions. In particular, you will often need to override OnNewDocument and OnOpenDocument to initialize the document's data members and DeleteContents to destroy dynamically allocated data. For information about overridable members, see class CDocument in the *MFC Reference*.

See Also
Reference
Using Documents

Managing Data with Document Data Variables

Implement your document's data as member variables of your document class. For example, the Scribble program declares a data member of type **CObList** — a linked list that stores pointers to **CObject** objects. This list is used to store arrays of points that make up a freehand line drawing.

How you implement your document's member data depends on the nature of your application. To help you out, MFC supplies a group of "collection classes" — arrays, lists, and maps (dictionaries), including collections based on C++ templates — along with classes that encapsulate a variety of common data types such as **CString**, **CRect**, **CPoint**, **CSize**, and **CTime**. For more information about these classes, see the Class Library Overview in the *MFC Reference*.

When you define your document's member data, you will usually add member functions to the document class to set and get data items and perform other useful operations on them.

Your views access the document object by using the view's pointer to the document, installed in the view at creation time. You can retrieve this pointer in a view's member functions by calling the **CView** member function **GetDocument**. Be sure to cast this pointer to your own document type. Then you can access public document members through the pointer.

If frequent data transfer requires direct access, or you wish to use the nonpublic members of the document class, you may want to make your view class a friend (in C++ terms) of the document class.

See Also
Reference
Using Documents

Serializing Data to and from Files

The basic idea of persistence is that an object should be able to write its current state, indicated by the values of its member variables, to persistent storage. Later, the object can be re-created by reading, or "deserializing," the object's state from persistent storage. A key point here is that the object itself is responsible for reading and writing its own state. Thus, for a class to be persistent, it must implement the basic serialization operations.

The framework provides a default implementation for saving documents to disk files in response to the Save and Save As commands on the File menu and for loading documents from disk files in response to the Open command. With very little work, you can implement a document's ability to write and read its data to and from a file. The main thing you must do is override the Serialize member function in your document class.

The MFC Application Wizard places a skeletal override of the **CDocument** member function **Serialize** in the document class it creates for you. After you have implemented your application's member variables, you can fill in your `Serialize` override with code that sends the data to an "archive object" connected to a file. A CArchive object is similar to the **cin** and **cout** input/output objects from the C++ iostream library. However, **CArchive** writes and reads binary format, not formatted text.

What do you want to know more about?

- Serialization
- The document's role in serialization
- The data's role in serialization
- Bypassing the serialization mechanism

The Document's Role in Serialization

The framework responds automatically to the File menu's Open, Save, and Save As commands by calling the document's `Serialize` member function if it is implemented. An **ID_FILE_OPEN** command, for example, calls a handler function in the application object. During this process, the user sees and responds to the File Open dialog box and the framework obtains the filename the user chooses. The framework creates a **CArchive** object set up for loading data into the document and passes the archive to `Serialize`. The framework has already opened the file. The code in your document's `Serialize` member function reads the data in through the archive, reconstructing data objects as needed. For more information about serialization, see the article Serialization.

The Data's Role in Serialization

In general, class-type data should be able to serialize itself. That is, when you pass an object to an archive, the object should know how to write itself to the archive and how to read itself from the archive. MFC provides support for making classes serializable in this way. If you design a class to define a data type and you intend to serialize data of that type, design for serialization.

See Also
Reference
Using Documents

Bypassing the Serialization Mechanism

As you have seen, the framework provides a default way to read and write data to and from files. Serializing through an archive object suits the needs of a great many applications. Such an application reads a file entirely into memory, lets the user update the file, and then writes the updated version to disk again.

However, some applications operate on data very differently, and for these applications serialization through an archive is not suitable. Examples include database programs, programs that edit only parts of large files, programs that write text-only files, and programs that share data files.

In these cases, you can override the Serialize function in a different way to mediate file actions through a CFile object rather than a CArchive object.

You can use the **Open**, **Read**, **Write**, **Close**, and **Seek** member functions of class **CFile** to open a file, move the file pointer (seek) to a specific point in the file, read a record (a specified number of bytes) at that point, let the user update the record, then seek to the same point again and write the record back to the file. The framework will open the file for you, and you can use the **GetFile** member function of class **CArchive** to obtain a pointer to the **CFile** object. For even more sophisticated and flexible use, you can override the OnOpenDocument and OnSaveDocument member functions of class **CWinApp**. For more information, see class CFile in the *MFC Reference*.

In this scenario, your `Serialize` override does nothing, unless, for example, you want to have it read and write a file header to keep it up to date when the document closes.

See Also
Reference
Using Documents

Handling Commands in the Document

Your document class may also handle certain commands generated by menu items, toolbar buttons, or accelerator keys. By default, **CDocument** handles the Save and Save As commands on the File menu, using serialization. Other commands that affect the data may also be handled by member functions of your document. For example, in the Scribble program, class `CScribDoc` provides a handler for the Edit Clear All command, which deletes all of the data currently stored in the document. Documents can have message maps, but unlike views, documents cannot handle standard Windows messages — only **WM_COMMAND** messages, or "commands."

See Also
Reference
Using Documents

Using Views

The view's responsibilities are to display the document's data graphically to the user and to accept and interpret user input as operations on the document. Your tasks in writing your view class are to:

- Write your view class's OnDraw member function, which renders the document's data.
- Connect appropriate Windows messages and user-interface objects such as menu items to message-handler member functions in the view class.
- Implement those handlers to interpret user input.

In addition, you may need to override other **CView** member functions in your derived view class. In particular, you may want to override OnInitialUpdate to perform special initialization for the view and OnUpdate to do any special processing needed just before the view redraws itself. For multipage documents, you also must override OnPreparePrinting to initialize the Print dialog box with the number of pages to print and other information. For more information on overriding **CView** member functions, see class CView in the *MFC Reference*.

What do you want to know more about?

- Derived view classes available in MFC
- Drawing in a view
- Interpreting user input through a view
- The role of the view in printing
- Scrolling and scaling views
- Initializing and cleaning up documents and views

See Also
Reference
CFormView Class
Bypassing the Serialization Mechanism
Concepts
Document/View Architecture
Record Views

Derived View Classes Available in MFC

The following table shows MFC's view classes and their relationships to one another. The capabilities of your view class depend on the MFC view class from which it derives.

View Classes

Class	Description
CView	Base class of all views.
CCtrlView	Base class of **CTreeView**, **CListView**, **CEditView**, and **CRichEditView**. These classes let you use document/view architecture with the indicated Windows common controls.
CEditView	A simple view based on the Windows edit box control. Allows entering and editing text and can be used as the basis for a simple text editor application. See also **CRichEditView**.
CRichEditView	A view containing a CRichEditCtrl object. This class is analogous to **CEditView**, but unlike **CEditView**, **CRichEditView** handles formatted text.
CListView	A view containing a CListCtrl object.
CTreeView	A view containing a CTreeCtrl object, for views that resemble the Solution Explorer window in Visual C++.
CScrollView	Base class of **CFormView**, **CRecordView**, and **CDaoRecordView**. Implements scrolling the view's contents.
CFormView	A form view, a view that contains controls. A forms-based application provides one or more such form interfaces.
CHtmlView	A Web browser view with which the application's user can browse sites on the World Wide Web, as well as folders in the local file system and on a network. The Web browser view can also work as an Active document container.
CRecordView	A form view that displays ODBC database records in controls. If you select ODBC support in your project, the view's base class is **CRecordView**. The view is connected to a **CRowset** object.
CDaoRecordView	A form view that displays DAO database records in controls. If you select DAO support in your project, the view's base class is **CDaoRecordView**. The view is connected to a **CDaoRecordset** object.
COleDBRecordView	A form view that displays OLE DB records in controls. If you select OLE DB support in your project, the view's base class is **COleDBRecordView**. The view is connected to a **CRowset** object.

Note
As of MFC version 4.0, **CEditView** is derived from **CCtrlView**.

To use these classes in your application, derive the application's view classes from them. For related information, see Scrolling and Scaling Views. For more information on the database classes, see Overview: Database Programming.

See Also

Reference

Using Views

Drawing in a View

Nearly all drawing in your application occurs in the view's `OnDraw` member function, which you must override in your view class. (The exception is mouse drawing, discussed in Interpreting User Input Through a View.) Your `OnDraw` override:

1. Gets data by calling the document member functions you provide.
2. Displays the data by calling member functions of a device-context object that the framework passes to `OnDraw`.

When a document's data changes in some way, the view must be redrawn to reflect the changes. Typically, this happens when the user makes a change through a view on the document. In this case, the view calls the document's UpdateAllViews member function to notify all views on the same document to update themselves. **UpdateAllViews** calls each view's OnUpdate member function. The default implementation of **OnUpdate** invalidates the view's entire client area. You can override it to invalidate only those regions of the client area that map to the modified portions of the document.

The **UpdateAllViews** member function of class **CDocument** and the **OnUpdate** member function of class **CView** let you pass information describing what parts of the document were modified. This "hint" mechanism lets you limit the area that the view must redraw. **OnUpdate** takes two "hint" arguments. The first, *lHint*, of type **LPARAM**, lets you pass any data you like, while the second, *pHint*, of type **CObject***, lets you pass a pointer to any object derived from **CObject**.

When a view becomes invalid, Windows sends it a **WM_PAINT** message. The view's OnPaint handler function responds to the message by creating a device-context object of class CPaintDC and calls your view's `OnDraw` member function. You do not normally have to write an overriding `OnPaint` handler function.

A device context is a Windows data structure that contains information about the drawing attributes of a device such as a display or a printer. All drawing calls are made through a device-context object. For drawing on the screen, `OnDraw` is passed a **CPaintDC** object. For drawing on a printer, it is passed a CDC object set up for the current printer.

Your code for drawing in the view first retrieves a pointer to the document, then makes drawing calls through the device context. The following simple `OnDraw` example illustrates the process:

```
void CMyView::OnDraw( CDC* pDC )
{
    CMyDoc* pDoc = GetDocument();
    CString s = pDoc->GetData();   // Returns a CString
    CRect rect;

    GetClientRect( &rect );
    pDC->SetTextAlign( TA_BASELINE | TA_CENTER );
    pDC->TextOut( rect.right / 2, rect.bottom / 2,
                  s, s.GetLength() );
}
```

In this example, you would define the `GetData` function as a member of your derived document class.

The example prints whatever string it gets from the document, centered in the view. If the `OnDraw` call is for screen drawing, the **CDC** object passed in *pDC* is a **CPaintDC** whose constructor has already called **BeginPaint**. Calls to drawing functions are made through the device-context pointer. For information about device contexts and drawing calls, see class CDC in the *MFC Reference* and Working with Window Objects.

For more examples of how to write `OnDraw`, see the MFC Samples.

See Also
Reference
Using Views

Interpreting User Input Through a View

Other member functions of the view handle and interpret all user input. You will usually define message-handler member functions in your view class to process:

- Windows messages generated by mouse and keyboard actions.
- Commands from menus, toolbar buttons, and accelerator keys.

These message-handler member functions interpret the following actions as data input, selection, or editing, including moving data to and from the Clipboard:

- Mouse movements and clicks, drags, and double-clicks
- Keystrokes
- Menu commands

Which Windows messages your view handles depends on your application's needs.

Message Handling and Mapping Topics explains how to assign menu items and other user-interface objects to commands and how to bind the commands to handler functions. Message Handling and Mapping Topics also explains how MFC routes commands and sends standard Windows messages to the objects that contain handlers for them.

For example, your application might need to implement direct mouse drawing in the view. The Scribble sample shows how to handle the **WM_LBUTTONDOWN**, **WM_MOUSEMOVE**, and **WM_LBUTTONUP** messages respectively to begin, continue, and end the drawing of a line segment. On the other hand, you might sometimes need to interpret a mouse click in your view as a selection. Your view's `OnLButtonDown` handler function would determine whether the user was drawing or selecting. If selecting, the handler would determine whether the click was within the bounds of some object in the view and, if so, alter the display to show the object as selected.

Your view might also handle certain menu commands, such as those from the Edit menu to cut, copy, paste, or delete selected data using the Clipboard. Such a handler would call some of the Clipboard-related member functions of class **CWnd** to transfer a selected data item to or from the Clipboard.

See Also
Reference
Using Views

The Role of the View in Printing

Your view also plays two important roles in printing its associated document.

The view:

- Uses the same OnDraw code to draw on the printer as to draw on the screen.
- Manages dividing the document into pages for printing.

For more information about printing and about the view's role in printing, see Printing and Print Preview.

See Also
Reference
Using Views

Scrolling and Scaling Views

MFC supports views that scroll and views that are automatically scaled to the size of the frame window that displays them. Class **CScrollView** supports both kinds of views.

For more information about scrolling and scaling, see class CScrollView in the *MFC Reference*. For a scrolling example, see the Scribble sample.

What do you want to know more about?

- Scrolling a view
- Scaling a view
- View coordinates

Scrolling a View

Frequently the size of a document is greater than the size its view can display. This may occur because the document's data increases or the user shrinks the window that frames the view. In such cases, the view must support scrolling.

Any view can handle scroll-bar messages in its **OnHScroll** and **OnVScroll** member functions. You can either implement scroll-bar message handling in these functions, doing all the work yourself, or you can use the **CScrollView** class to handle scrolling for you.

CScrollView does the following:

- Manages window and viewport sizes and mapping modes
- Scrolls automatically in response to scroll-bar messages

You can specify how much to scroll for a "page" (when the user clicks in a scroll-bar shaft) and a "line" (when the user clicks in a scroll arrow). Plan these values to suit the nature of your view. For example, you might want to scroll in 1-pixel increments for a graphics view but in increments based on the line height in text documents.

Scaling a View

When you want the view to automatically fit the size of its frame window, you can use **CScrollView** for scaling instead of scrolling. The logical view is stretched or shrunk to fit the window's client area exactly. A scaled view has no scroll bars.

See Also
Reference
Using Views

Multiple Document Types, Views, and Frame Windows

The standard relationship among a document, its view, and its frame window is described in Document/View Creation. Many applications support a single document type (but possibly multiple open documents of that type) with a single view on the document and only one frame window per document. But some applications may need to alter one or more of those defaults.

What do you want to know more about?

- Multiple document types
- Multiple views
- Multiple frame windows
- Splitter windows

Multiple Document Types

The MFC Application Wizard creates a single document class for you. In some cases, though, you may need to support more than one document type. For example, your application may need worksheet and chart documents. Each document type is represented by its own document class and probably by its own view class as well. When the user chooses the File New command, the framework displays a dialog box that lists the supported document types. Then it creates a document of the type that the user chooses. Each document type is managed by its own document-template object.

To create extra document classes, see Adding a Class. Choose CDocument as the Class Type to derive from and supply the requested document information. Then implement the new class's data.

To let the framework know about your extra document class, you must add a second call to AddDocTemplate in your application class's InitInstance override. For more information, see Document Templates.

Multiple Views

Many documents require only a single view, but it is possible to support more than one view of a single document. To help you implement multiple views, a document object keeps a list of its views, provides member functions for adding and removing views, and supplies the UpdateAllViews member function for letting multiple views know when the document's data has changed.

MFC supports three common user interfaces requiring multiple views on the same document. These models are:

- View objects of the same class, each in a separate MDI document frame window.

  You might want to support creating a second frame window on a document. The user could choose a New Window command to open a second frame with a view of the same document and then use the two frames to view different portions of the document simultaneously. The framework supports the New Window command on the Window menu for MDI applications by duplicating the initial frame window and view attached to the document.

- View objects of the same class in the same document frame window.

  Splitter windows split the view space of a single document window into multiple separate views of the document. The framework creates multiple view objects from the same view class. For more information, see Splitter Windows.

- View objects of different classes in a single frame window.

  In this model, a variation of the splitter window, multiple views share a single frame window. The views are constructed from different classes, each view providing a different way to view the same document. For example, one view might show a word-processing document in normal mode while the other view shows it in outline mode. A splitter control allows the user to adjust the relative sizes of the views.

The following figure, divided into parts a, b, and c, shows the three user-interface models in the order presented above.

Multiple-View User Interfaces

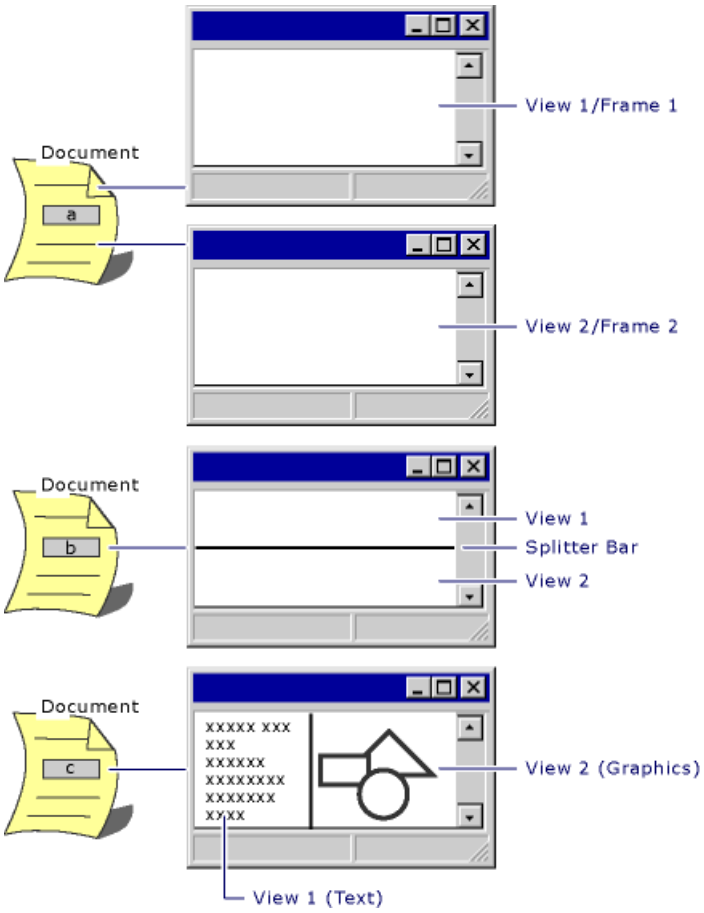

The framework provides these models by implementing the New Window command and by providing class CSplitterWnd, as discussed in Splitter Windows. You can implement other models using these as your starting point. For sample programs that illustrate different configurations of views, frame windows, and splitters, see MFC Samples.

For more information about **UpdateAllViews**, see class CView in the *MFC Reference* and the Scribble sample.

Multiple Frame Windows

You can use the New Window command on the Window menu for MDI applications to create a second frame window on the same document. For more information, see the first model in the figure Multiple-View User Interfaces.

Splitter Windows

In a splitter window, the window is, or can be, split into two or more scrollable panes. A splitter control (or "split box") in the window frame next to the scroll bars allows the user to adjust the relative sizes of the panes. Each pane is a view on the same document. In "dynamic" splitters, the views are of the same class, as shown in part b of the figure Multiple-View User Interfaces. In "static" splitters, the views can be of different classes. Splitter windows of both kinds are supported by class CSplitterWnd.

Dynamic splitter windows, with views of the same class, allow the user to split a window into multiple panes at will and then scroll different panes to see different parts of the document. The user can also unsplit the window to remove the additional views. The splitter windows added to the Scribble sample are an example. That topic describes the technique for creating dynamic splitter windows. A dynamic splitter window is shown in part b of the figure Multiple-View User Interfaces.

Static splitter windows, with views of different classes, start with the window split into multiple panes, each with a different purpose. For example, in the Visual C++ bitmap editor, the image window shows two panes side by side. The left-hand pane displays a life-sized image of the bitmap. The right-hand pane displays a zoomed or magnified image of the same bitmap. The panes are separated by a "splitter bar" that the user can drag to change the relative sizes of the panes. A static splitter window is shown in part c of the figure Multiple-View User Interfaces.

For more information, see class CSplitterWnd in the *MFC Reference* and MFC Samples.

See Also

Concepts

Document/View Architecture

Initializing and Cleaning Up Documents and Views

Use the following guidelines for initializing and cleaning up after your documents and views:

- The MFC framework initializes documents and views; you initialize any data you add to them.
- The framework cleans up as documents and views close; you must deallocate any memory that you allocated on the heap from within the member functions of those documents and views.

Note
Recall that initialization for the whole application is best done in your override of the InitInstance member function of class **CWinApp**, and cleanup for the whole application is best done in your override of the **CWinApp** member function ExitInstance.

The life cycle of a document (and its frame window and view or views) in an MDI application is as follows:

1. During dynamic creation, the document constructor is called.
2. For each new document, the document's OnNewDocument or OnOpenDocument is called.
3. The user interacts with the document throughout its lifetime. Typically this happens as the user works on document data through the view, selecting and editing the data. The view passes changes on to the document for storage and updating other views. During this time both the document and the view might handle commands.
4. The framework calls DeleteContents to delete data specific to a document.
5. The document's destructor is called.

In an SDI application, step 1 is performed once, when the document is first created. Then steps 2 through 4 are performed repeatedly each time a new document is opened. The new document reuses the existing document object. Finally, step 5 is performed when the application ends.

What do you want to know more about?

- Initializing Documents and Views
- Cleaning Up Documents and Views

See Also
Concepts
Document/View Architecture

Initializing Documents and Views

Documents are created in two different ways, so your document class must support both ways. First, the user can create a new, empty document with the File New command. In that case, initialize the document in your override of the OnNewDocument member function of class CDocument. Second, the user can use the Open command on the File menu to create a new document whose contents are read from a file. In that case, initialize the document in your override of the OnOpenDocument member function of class **CDocument**. If both initializations are the same, you can call a common member function from both overrides, or **OnOpenDocument** can call **OnNewDocument** to initialize a clean document and then finish the open operation.

Views are created after their documents are created. The best time to initialize a view is after the framework has finished creating the document, frame window, and view. You can initialize your view by overriding the OnInitialUpdate member function of CView. If you need to reinitialize or adjust anything each time the document changes, you can override OnUpdate.

See Also
Reference
Initializing and Cleaning Up Documents and Views

Cleaning Up Documents and Views

When a document is closing, the framework first calls its DeleteContents member function. If you allocated any memory on the heap during the course of the document's operation, **DeleteContents** is the best place to deallocate it.

Note
You should not deallocate document data in the document's destructor. In the case of an SDI application, the document objec t might be reused.

You can override a view's destructor to deallocate any memory you allocated on the heap.

See Also

Reference

Initializing and Cleaning Up Documents and Views

Adding Multiple Views to a Single Document

In a single-document interface (SDI) application created with the Microsoft Foundation Class (MFC) Library, each document type is associated with a single view type. In some cases, it is desirable to have the ability to switch the current view of a document with a new view.

Tip

For additional procedures on implementing multiple views for a single document, see CDocument::AddView and the COLLECT MFC sample.

You can implement this functionality by adding a new **CView**-derived class and additional code for switching the views dynamically to an existing MFC application.

The steps are as follows:

- Modify the Existing Application Class
- Create and Modify the New View Class
- Create and Attach the New View
- Implement the Switching Function
- Add Support for Switching the View

The remainder of this topic assumes the following:

- The name of the **CWinApp**-derived object is `CMyWinApp`, and `CMyWinApp` is declared and defined in MYWINAPP.H and MYWINAPP.CPP.
- `CNewView` is the name of the new **CView**-derived object, and `CNewView` is declared and defined in NEWVIEW.H and NEWVIEW.CPP.

Modify the Existing Application Class

For the application to switch between views, you need to modify the application class by adding member variables to store the views and a method to switch them.

Add the following code to the declaration of `CMyWinApp` in MYWINAPP.H:

```
CView* m_pOldView;
CView* m_pNewView;
CView* SwitchView( );
```

The new member variables, `m_pOldView` and `m_pNewView`, point to the current view and the newly created one. The new method (`SwitchView`) switches the views when requested by the user. The body of the method is discussed later in this topic in Implement the Switching Function.

The last modification to the application class requires including a new header file that defines a Windows message (**WM_INITIALUPDATE**) that is used in the switching function.

Insert the following line in the include section of MYWINAPP.CPP:

```
#include <AFXPRIV.H>
```

Save your changes and continue to the next step.

Create and Modify the New View Class

Creating the new view class is made easy by using the **New Class** command available from Class View. The only requirement for this class is that it derives from **CView**. Add this new class to the application. For specific information on adding a new class to the project, see Adding a Class.

Once you have added the class to the project, you need to change the accessibility of some view class members.

Modify NEWVIEW.H by changing the access specifier from **protected** to **public** for the constructor and destructor. This allows the class to be created and destroyed dynamically and to modify the view appearance before it is visible.

Save your changes and continue to the next step.

Create and Attach the New View

To create and attach the new view, you need to modify the **InitInstance** function of your application class. The modification adds new code that creates a new view object and then initializes both `m_pOldView` and `m_pNewView` with the two existing view objects.

Because the new view is created within the **InitInstance** function, both the new and existing views persist for the lifetime of the application. However, the application could just as easily create the new view dynamically.

Insert this code after the call to `ProcessShellCommand`:

```
...
CView* pActiveView = ((CFrameWnd*) m_pMainWnd)->GetActiveView();
m_pOldView = pActiveView;
m_pNewView = (CView*) new CNewView;

CDocument* pCurrentDoc = ((CFrameWnd*)m_pMainWnd)->GetActiveDocument();

// Initialize a CCreateContext to point to the active document.
// With this context, the new view is added to the document
// when the view is created in CView::OnCreate().
CCreateContext newContext;
newContext.m_pNewViewClass = NULL;
newContext.m_pNewDocTemplate = NULL;
newContext.m_pLastView = NULL;
newContext.m_pCurrentFrame = NULL;
newContext.m_pCurrentDoc = pCurrentDoc;

// The ID of the initial active view is AFX_IDW_PANE_FIRST.
// Incrementing this value by one for additional views works
// in the standard document/view case but the technique cannot
// be extended for the CSplitterWnd case.
UINT viewID = AFX_IDW_PANE_FIRST + 1;
CRect rect(0, 0, 0, 0); // Gets resized later.

// Create the new view. In this example, the view persists for
// the life of the application. The application automatically
// deletes the view when the application is closed.
m_pNewView->Create(NULL, "AnyWindowName", WS_CHILD, rect, m_pMainWnd, viewID, &newContext);

// When a document template creates a view, the WM_INITIALUPDATE
// message is sent automatically. However, this code must
// explicitly send the message, as follows.
m_pNewView->SendMessage(WM_INITIALUPDATE, 0, 0);
...
```

Save your changes and continue to the next step.

Implement the Switching Function

In the previous step, you added code that created and initialized a new view object. The last major piece is to implement the switching method, `SwitchView`.

At the end of the implementation file for your application class (MYWINAPP.CPP), add the following method definition:

```
CView* CMyWinApp::SwitchView( )
{
   CView* pActiveView =
      ((CFrameWnd*) m_pMainWnd)->GetActiveView();

   CView* pNewView= NULL;
   if(pActiveView == m_pOldView)
      pNewView= m_pNewView;
```

```
    else
       pNewView= m_pOldView;

    // Exchange view window IDs so RecalcLayout() works.
    #ifndef _WIN32
    UINT temp = ::GetWindowWord(pActiveView->m_hWnd, GWW_ID);
    ::SetWindowWord(pActiveView->m_hWnd, GWW_ID, ::GetWindowWord(pNewView->m_hWnd, GWW_ID));
    ::SetWindowWord(pNewView->m_hWnd, GWW_ID, temp);
    #else
    UINT temp = ::GetWindowLong(pActiveView->m_hWnd, GWL_ID);
    ::SetWindowLong(pActiveView->m_hWnd, GWL_ID, ::GetWindowLong(pNewView->m_hWnd, GWL_ID));
    ::SetWindowLong(pNewView->m_hWnd, GWL_ID, temp);
    #endif

    pActiveView->ShowWindow(SW_HIDE);
    pNewView->ShowWindow(SW_SHOW);
    ((CFrameWnd*) m_pMainWnd)->SetActiveView(pNewView);
    ((CFrameWnd*) m_pMainWnd)->RecalcLayout();
    pNewView->Invalidate();
    return pActiveView;
 }
```

Save your changes and continue to the next step.

Add Support for Switching the View

The final step involves adding code that calls the `SwitchView` method when the application needs to switch between views. This can be done in several ways: by either adding a new menu item for the user to choose or switching the views internally when certain conditions are met.

For more information on adding new menu items and command handler functions, see Handlers for Commands and Control Notifications.

See Also

Concepts

Document/View Architecture

Form Views (MFC)

You can add forms to any Visual C++ application that supports the MFC libraries, including a forms-based application (one whose view class is derived from **CFormView**). If you did not initially create your application to support forms, Visual C++ will add this support for you when you insert a new form. In an SDI or MDI application, which implements the default document/view architecture, when the user chooses the **New** command (by default, on the **File** menu), Visual C++ prompts the user to choose from the available forms.

With an SDI application, when the user chooses the **New** command, the current instance of the form continues to run but a new instance of the application with the selected form is created if one is not found. In an MDI application, the current instance of the form continues to run when the user chooses the **New** command.

Note
You can insert a form into a dialog-based application (one whose dialog class is based on **CDialog** and one in which no view class is implemented). However, without the document/view architecture, Visual C++ does not automatically implement the **File**|**New** functionality. You must create a way for the user to view additional forms, such as by implementing a tabbed dialog box with various property pages.

When you insert a new form into your application, Visual C++ does the following:

- Creates a class based on one of the form-style classes that you choose (**CFormView**, **CRecordView**, **CDaoRecordView**, or **CDialog**).
- Creates a dialog resource with appropriate styles (or you can use an existing dialog resource that has not yet been associated with a class).

  If you choose an existing dialog resource, you may need to set these styles by using the Properties page for the dialog box. Styles for a dialog box must include:

  WS_CHILD=On

  WS_BORDER=Off

  WS_VISIBLE=Off

  WS_CAPTION=Off

For applications based on the document/view architecture, the **New Form** command (right-click in Class View) also:

- Creates a **CDocument**-based class

  Instead of having a new class created, you can use any existing **CDocument**-based class in your project.
- Generates a document template (derived from **CDocument**) with string, menu, and icon resources.

  You can also create a new class on which to base the template.
- Adds a call to **AddDocumentTemplate** in your application's **InitInstance** code.

  Visual C++ adds this code for each new form you create, which adds the form to the list of available forms when the user chooses the **New** command. This code includes the form's associated resource ID and the names of the associated document, view, and frame classes that together make up the new form object.

  Document templates serve as the connection between documents, frame windows, and views. For a single document, you can create many templates.

For more information, see:

- Create a Forms-Based Application
- Inserting a Form into a Project

See Also

Other Resources

General Concepts

Inserting a Form into a Project

Forms provide a convenient container for controls. You can easily insert an MFC-based form into your application as long as the application supports the MFC libraries.

To insert a form into your project

1. From Class View, select the project to which you want to add the form, and click the right mouse button.
2. From the shortcut menu, click **Add** and then click **Add Class**.

   If the **New Form** command is not available, your project may be based on the Active Template Library (ATL). To add a form to an ATL project, you must specify certain settings when first creating the project.
3. From the **MFC** folder, click **MFC Class**.
4. Using the MFC Class Wizard, make the new class derive from CFormView.

Visual C++ adds the form to your application, opening it inside the Dialog editor so that you can begin adding controls and working on its overall design.

You may want to perform the following additional steps (not applicable for dialog-based applications):

1. Override the **OnUpdate** member function.
2. Implement a member function to move data from your view to your document.
3. Create an **OnPrint** member function.

See Also

Concepts

Form Views (MFC)

HTML Help: Context-Sensitive Help for Your Programs

Visual C++ ships with an online help system called HTML Help. HTML Help uses the underlying components of Microsoft Internet Explorer to display help content.

To create online help for your application, you use the authoring tool in HTML Help called HTML Help Workshop.

The topics in this section explain how to create context-sensitive help for an application by using HTML Help Workshop. Applications that use context-sensitive help typically have a user interface. Commonly, an MFC (.exe), MFC ActiveX control, or an ATL ActiveX control are types of applications that use context-sensitive help. The examples used in the topics in this section use handlers available in an MFC (.exe) application to implement context-sensitive help.

You can select HTMLHelp for your application's context-sensitive help on the Advanced Features, MFC Application Wizard page of the MFC Application Wizard.

What do you want to know more about?

Displaying F1 Help for a Dialog Box or Menu Option

Displaying Context-Sensitive Help

Adding HTMLHelp Context-Sensitive Help to an Existing MFC Application

Receiving HTML Help Notification Messages in an MFC Application

See Also

Concepts

WinHelp: Context-Sensitive Help for Your Programs

Other Resources

General Concepts

User Interface

Displaying F1 Help for a Dialog Box or Menu Option

When an MFC application is enabled for context sensitive help in HTMLHelp format, you can provide for the display of a topic when a user presses F1 on a menu option or on a dialog box.

To manually make the HTML Help viewer display

1. Assuming a dialog box with an ID of IDD_MYDLG, add the following alias to the project's .hhp file: **HIDD_MYDLG = HID_APP_MYDLG.htm**.
2. Add the following to hlp\afxcore.htm:

```
<OBJECT type="application/x-oleobject" classid="clsid:1e2a7bd0-dab9-11d0-b93a-00c04fc9
9f9e" VIEWASTEXT>
<param name="New HTML file" value="hid_app_mydlg.htm">
<param name="New HTML title" value="MyDlg command (Help menu)">
</OBJECT>
<P>Help text goes here.</P>
```

The same procedure will work if the ID is that of a menu option.

See Also

Tasks

Displaying Context-Sensitive Help

Reference

Adding Context-Sensitive Help to an Existing MFC Application
CWinApp::EnableHtmlHelp
CWnd::HtmlHelp
CWinApp::HtmlHelp

Concepts

HTML Help: Context-Sensitive Help for Your Programs

Displaying Context-Sensitive Help

Context-sensitive help, for the purpose of this discussion, refers to help support for the controls in a dialog box that users access:

- By pressing the F1 key.
- By right-clicking a control (What's This? Help).
- By using the question-mark pointer (the What's This? Help pointer).

For each control that you want to support context-sensitive help, set the **HelpID** property to **True**.

If necessary, add hhctrl.lib to the **Additional Dependencies** property, which is in the **Input** property page of the **Linker** folder in your project's **Property Pages** dialog box.

The source information for context-sensitive help is stored in a .txt file that you include in your HTML Help project.

To create the context-sensitive help text file

1. Use a text editor to create a .txt file.
2. Format the topics as follows:

```
.topic 1
help text for control 1
.topic 2
help text for control 2
```

Note

For more information, see "Designing context-sensitive help" in HTML Help's online help. From the **Help** menu (in HTML Hel p Workshop), choose **Help Topics**.

After you create the .txt file, add it to the [Files] section in your .hhp file.

To support help for resources in a dialog box, you must create a two-dimensional array that maps control IDs to help IDs (topic numbers).

To create the two-dimensional array

- In the .cpp file, for every class that represents a dialog box, add a two-dimensional array to the end of the class. For example:

```
static DWORD myarray[] = {
   IDC_CHECK1, 1,
   IDC_CHECK2, 2,
   IDC_CHECK3, -1,
   0,0
};
```

Each entry in the two-dimensional array pairs a resource ID for a dialog-box control with a topic number from the context-sensitive help text file. If you do not want a specific resource to have What's This? Help, use –1. The last pair in this array should be 0,0.

F1 access to context-sensitive help means that users will be able to press F1 when a control has focus to access help.

To enable F1 access to context-sensitive help

- Implement a handler for the **WM_HELPINFO** message (in each dialog-box class where you want F1 access to context-sensitive help) and implement the following code for the handler:

```
BOOL CMyDialog::OnHelpInfo(HELPINFO* pHelpInfo)
{
   if (pHelpInfo->iContextType == HELPINFO_WINDOW)
   {
      return ::HtmlHelp(
         (HWND)pHelpInfo->hItemHandle,
         "\\my_chm.chm::/ctrlhlp.txt",
         HH_TP_HELP_WM_HELP,
         (DWORD)(LPVOID)myarray)
         != NULL;
}
return TRUE;
}
```

What's This? Help displays the control's help when a user right-clicks the control.

To implement right-click What's This? Help

- Implement a handler (see Mapping Messages to Functions) for the **WM_CONTEXTMENU** message in each dialog-box class where you want What's This? Help (select the ID for the dialog box from the list of Object IDs). Implement the following code for the handler:

```
void CMyDialog::OnContextMenu(CWnd* pWnd, CPoint point)
{
   ::HtmlHelp(
      pWnd->GetSafeHwnd(),
      "hlp\\my_chm.chm::/ctrlhlp.txt",
      HH_TP_HELP_CONTEXTMENU,
      (DWORD)(LPVOID)myarray);
}
```

When you specify the .chm file, the expected location is the project's working directory. See the **Debug** tab of the **Property Pages** dialog box for the location of the working directory (by default, the project directory). When you specify the text file in the .chm that contains the context-sensitive help, you must specify the same location information as is specified for the .txt file in the .chm's .hhp file.

If you already implement F1 access to context-sensitive help, you can easily enable the What's This? pointer, which causes a question mark to appear on the title bar, in the upper right-hand corner of the dialog box.

To enable the What's This? Help question-mark pointer

- Select the **Context help** check box in the **Extended Styles** tab of the dialog box properties.

See Also

Concepts

HTML Help: Context-Sensitive Help for Your Programs

Adding HTML Help Context-Sensitive Help to an Existing MFC Application

If an MFC application was created without the context-sensitive help wizard option, you can subsequently add context-sensitive help.

To add context-sensitive help in HTMLHelp format to an MFC application

1. Follow the steps in Copying Help-Specific Resources to Your Project.
2. Follow the instructions in Copying the Help Message Map Commands.
3. Copy the files from the hlp directory for the project that was created to use context-sensitive help in HTMLHelp format.
4. Rename the *.hh* files so the file names match the name of the application you are developing.
5. Open the .hhp file in HTMLHelp Workshop and change the file names in the .hhp file to match the *.hh* files in your project.
6. Place the following line in the constructor for the *AppName* module (where *AppName* is the name of the application; this example uses an application called Test):

```
CTestApp::CTestApp()
{
   // TODO: add construction code here,
   // Place all significant initialization in InitInstance
   EnableHtmlHelp();
}
```

7. Examine the .vcproj file (use Notepad) of the project that was created to use context-sensitive help in HTMLHelp format and notice the custom build steps used to create the help file (.chm file) when the project is built. Do a search for VCCustomBuildTool in the .vcproj file to locate these custom build steps. Copy those same custom build steps into your project.

See Also

Concepts

HTML Help: Context-Sensitive Help for Your Programs

Receiving HTML Help Notification Messages in an MFC Application

To receive notification messages from HTML Help within an MFC program, you must:

1. Define a symbol in your Visual C++ project. This example uses a symbol called ID_HHNOTIFICATION.
   - To define a symbol, right-click the high-level folder in **ResourceView** and select **Resource Symbols**.
   - In the **Resource Symbols** dialog box, click **New** and define the new symbol.
2. In your Visual C++ project, initialize the **HH_WINTYPE** structure and call the **HTMLHelp** function to set this structure using the **HH_SET_WIN_TYPE** command. Use **ID_HHNOTIFICATION** for the *idNotify* field in the structure.
3. Override the **OnNotify** function in the derivative of the **CWnd** class that you want to receive the message (the **CWnd** class associated with **HWND** specified in the **hwndCaller** field of the **WW_WINTYPE** structure). The following example shows how an **OnNotify** function is used to call an **OnNavComplete(HHN_NOTIFY*, LRESULT)** handler whenever HTML Help navigates to a topic:

```
BOOL CMyView::OnNotify(WPARAM wParam, LPARAM lParam, LRESULT* pResult)
{
    NMHDR* pNMHDR = (NMHDR*)lParam;
   switch (pNMHDR->idFrom) {
   case ID_HHNOTIFICATION:  // whatever id you placed in idNotify of HH_WINTYPE
   if (pNMHDR->code == HHN_NAVCOMPLETE) {
      OnNavComplete((HHN_NOTIFY*) lParam, pResult);
      return TRUE;
      }
      break;
   }
return CView::OnNotify(wParam, lParam, pResult);
}
```

See Also

Concepts

HTML Help: Context-Sensitive Help for Your Programs

Menus (MFC)

MFC supplies two elements to help you work with menus:

- Class CMenu for manipulating your program's menus at run time. Use the documentation for **CMenu** and the sample to learn how to use **CMenu** effectively.
- A mechanism for updating menus and toolbar buttons: enabling or disabling them on the fly to suit current program conditions.

Visual C++ also provides a menu editor for creating and editing your program's menu resources.

What do you want to know more about?

- Manipulating menu objects during program execution
- How to Update User-Interface Objects
- Sample

See Also

Other Resources

General Concepts
User Interface

Menu Sample List

See the following sample programs that illustrate creating, editing, and updating menus:

MDI Sample: Enable and Disable Menu Items

- SCRIBBLE

Dynamically Change Menus

- DYNAMENU

See Also

Concepts

Menus (MFC)

Manipulating Menus During Program Execution

Use class **CMenu** to manipulate menus and menu items on the fly. **CMenu** encapsulates a Windows **HMENU** handle and supplies member functions for working with menus.

See the overview for class CMenu for details.

See Also
Concepts
Menus (MFC)

How to: Update User-Interface Objects

Typically, menu items and toolbar buttons have more than one state. For example, a menu item is grayed (dimmed) if it is unavailable in the present context. Menu items can also be checked or unchecked. A toolbar button can also be disabled if unavailable, or it can be checked.

Who updates the state of these items as program conditions change? Logically, if a menu item generates a command that is handled by, say, a document, it makes sense to have the document update the menu item. The document probably contains the information on which the update is based.

If a command has multiple user-interface objects (perhaps a menu item and a toolbar button), both are routed to the same handler function. This encapsulates your user-interface update code for all of the equivalent user-interface objects in a single place.

The framework provides a convenient interface for automatically updating user-interface objects. You can choose to do the updating in some other way, but the interface provided is efficient and easy to use.

The following topics explain the use of update handlers:

- When update handlers are called
- The ON_UPDATE_COMMAND_UI macro
- The CCmdUI class

See Also

Concepts

Menus (MFC)

When Update Handlers Are Called

Suppose the user clicks the mouse in the File menu, which generates a **WM_INITMENUPOPUP** message. The framework's update mechanism collectively updates all items on the File menu before the menu drops down so the user can see it.

To do this, the framework routes update commands for all menu items in the pop-up menu along the standard command routing. Command targets on the routing have an opportunity to update any menu items by matching the update command with an appropriate message-map entry (of the form **ON_UPDATE_COMMAND_UI**) and calling an "update handler" function. Thus, for a menu with six menu items, six update commands are sent out. If an update handler exists for the command ID of the menu item, it is called to do the updating. If not, the framework checks for the existence of a handler for that command ID and enables or disables the menu item as appropriate.

If the framework does not find an **ON_UPDATE_COMMAND_UI** entry during command routing, it automatically enables the user-interface object if there is an **ON_COMMAND** entry somewhere with the same command ID. Otherwise, it disables the user-interface object. Therefore, to ensure that a user-interface object is enabled, supply a handler for the command the object generates or supply an update handler for it. See the figure in the topic User-Interface Objects and Command IDs.

It is possible to disable the default disabling of user-interface objects. For more information, see the m_bAutoMenuEnable member of class **CFrameWnd** in the *MFC Reference*.

Menu initialization is automatic in the framework, occurring when the application receives a **WM_INITMENUPOPUP** message. During the idle loop, the framework searches the command routing for button update handlers in much the same way as it does for menus.

See Also

Concepts

How to: Update User-Interface Objects

The ON_UPDATE_COMMAND_UI Macro

Use the **Properties** window to connect a user-interface object to a command-update handler in a command-target object. It will automatically connect the user-interface object's ID to the **ON_UPDATE_COMMAND_UI** macro and create a handler in the object that will handle the update. See Mapping Messages to Functions for more information.

For example, to update a Clear All command in your program's Edit menu, use the **Properties** window to add a message-map entry in the chosen class, a function declaration for a command-update handler called `OnUpdateEditClearAll` in the class declaration, and an empty function template in the class's implementation file. The function prototype looks like this:

```
afx_msg void OnUpdateEditClearAll( CCmdUI* pCmdUI );
```

Like all handlers, the function shows the **afx_msg** keyword. Like all update handlers, it takes one argument, a pointer to a **CCmdUI** object.

See Also

Concepts

How to: Update User-Interface Objects

The CCmdUI Class

When it routes an update command to its handler, the framework passes the handler a pointer to a **CCmdUI** object (or to an object of a **CCmdUI**-derived class). This object represents the menu item or toolbar button or other user-interface object that generated the command. The update handler calls member functions of the **CCmdUI** structure through the pointer to update the user-interface object. For example, here is an update handler for the Clear All menu item:

```
void CMyClass::OnUpdateToolsMyTool( CCmdUI* pCmdUI )
{
    if( ToolAvailable() )
        pCmdUI->Enable( TRUE );
}
```

This handler calls the **Enable** member function of an object with access to the menu item. **Enable** makes the item available for use.

See Also

Concepts

How to: Update User-Interface Objects

OLE (MFC)

Implementing OLE functionality in your program affects your user interface in several ways:

- Visual editing (in-place activation) displays the user interface of another program in your program's windows and modifies your program's menus with items from the other program.
- Drag and drop allows users to drag objects within and between windows and even between programs.
- Trackers provide visual cues to the state of objects during visual editing and drag and drop.

For more information, see:

- OLE and MFC
- Visual Editing (Activation)
- Drag and Drop
- Trackers

See Also
Other Resources
General Concepts
User Interface

Printing and Print Preview

MFC supports printing and print preview for your program's documents via class CView. For basic printing and print preview, simply override your view class's OnDraw member function, which you must do anyway. That function can draw to the view on the screen, to a printer device context for an actual printer, or to a device context that simulates your printer on the screen.

You can also add code to manage multipage document printing and preview, to paginate your printed documents, and to add headers and footers to them.

This family of articles explains how printing is implemented in the Microsoft Foundation Class Library (MFC) and how to take advantage of the printing architecture already built into the framework. The articles also explain how MFC supports easy implementation of print preview functionality and how you can use and modify that functionality.

What do you want to know more about?

- Printing
- Print preview architecture
- Sample

See Also

Other Resources

General Concepts

User Interface

Printing

Microsoft Windows implements device-independent display. In MFC, this means that the same drawing calls, in the `OnDraw` member function of your view class, are responsible for drawing on the display and on other devices, such as printers. For print preview, the target device is a simulated printer output to the display.

Your Role in Printing vs. the Framework's Role

Your view class has the following responsibilities:

- Inform the framework how many pages are in the document.
- When asked to print a specified page, draw that portion of the document.
- Allocate and deallocate any fonts or other graphics device interface (GDI) resources needed for printing.
- If necessary, send any escape codes needed to change the printer mode before printing a given page, for example, to change the printing orientation on a per-page basis.

The framework's responsibilities are as follows:

- Display the **Print** dialog box.
- Create a CDC object for the printer.
- Call the StartDoc and EndDoc member functions of the **CDC** object.
- Repeatedly call the StartPage member function of the **CDC** object, inform the view class which page should be printed, and call the EndPage member function of the **CDC** object.
- Call overridable functions in the view at the appropriate times.

The following articles discuss how the framework supports printing and print preview:

What do you want to know more about?

- How default printing is done
- Multipage documents
- Headers and footers
- Allocating GDI resources for printing
- Print preview

See Also
Reference
Printing and Print Preview

How Default Printing Is Done

This article explains the default printing process in Windows in terms of the MFC framework.

In MFC applications, the view class has a member function named `OnDraw` that contains all the drawing code. `OnDraw` takes a pointer to a CDC object as a parameter. That **CDC** object represents the device context to receive the image produced by `OnDraw`. When the window displaying the document receives a WM_PAINT message, the framework calls `OnDraw` and passes it a device context for the screen (a CPaintDC object, to be specific). Accordingly, `OnDraw`'s output goes to the screen.

In programming for Windows, sending output to the printer is very similar to sending output to the screen. This is because the Windows graphics device interface (GDI) is hardware-independent. You can use the same GDI functions for screen display or for printing simply by using the appropriate device context. If the **CDC** object that `OnDraw` receives represents the printer, `OnDraw`'s output goes to the printer.

This explains how MFC applications can perform simple printing without requiring extra effort on your part. The framework takes care of displaying the Print dialog box and creating a device context for the printer. When the user selects the Print command from the File menu, the view passes this device context to `OnDraw`, which draws the document on the printer.

However, there are some significant differences between printing and screen display. When you print, you have to divide the document into distinct pages and display them one at a time, rather than display whatever portion is visible in a window. As a corollary, you have to be aware of the size of the paper (whether it's letter size, legal size, or an envelope). You may want to print in different orientations, such as landscape or portrait mode. The Microsoft Foundation Class Library can't predict how your application will handle these issues, so it provides a protocol for you to add these capabilities.

That protocol is described in the article Multipage Documents.

See Also
Reference
Printing

Multipage Documents

This article describes the Windows printing protocol and explains how to print documents that contain more than one page. The article covers the following topics:

- Printing protocol
- Overriding view class functions
- Pagination
- Printer pages vs. document pages
- Print-time pagination

The Printing Protocol

To print a multipage document, the framework and view interact in the following manner. First the framework displays the **Print** dialog box, creates a device context for the printer, and calls the StartDoc member function of the CDC object. Then, for each page of the document, the framework calls the StartPage member function of the **CDC** object, instructs the view object to print the page, and calls the EndPage member function. If the printer mode must be changed before starting a particular page, the view calls ResetDC, which updates the DEVMODE structure containing the new printer mode information. When the entire document has been printed, the framework calls the EndDoc member function.

Overriding View Class Functions

The CView class defines several member functions that are called by the framework during printing. By overriding these functions in your view class, you provide the connections between the framework's printing logic and your view class's printing logic. The following table lists these member functions.

CView's Overridable Functions for Printing

Name	Reason for overriding
OnPreparePrinting	To insert values in the Print dialog box, especially the length of the document
OnBeginPrinting	To allocate fonts or other GDI resources
OnPrepareDC	To adjust attributes of the device context for a given page, or to do print-time pagination
OnPrint	To print a given page
OnEndPrinting	To deallocate GDI resources

You can do printing-related processing in other functions as well, but these functions are the ones that drive the printing process.

The following figure illustrates the steps involved in the printing process and shows where each of **CView**'s printing member functions are called. The rest of this article explains most of these steps in more detail. Additional parts of the printing process are described in the article Allocating GDI Resources.

The Printing Loop

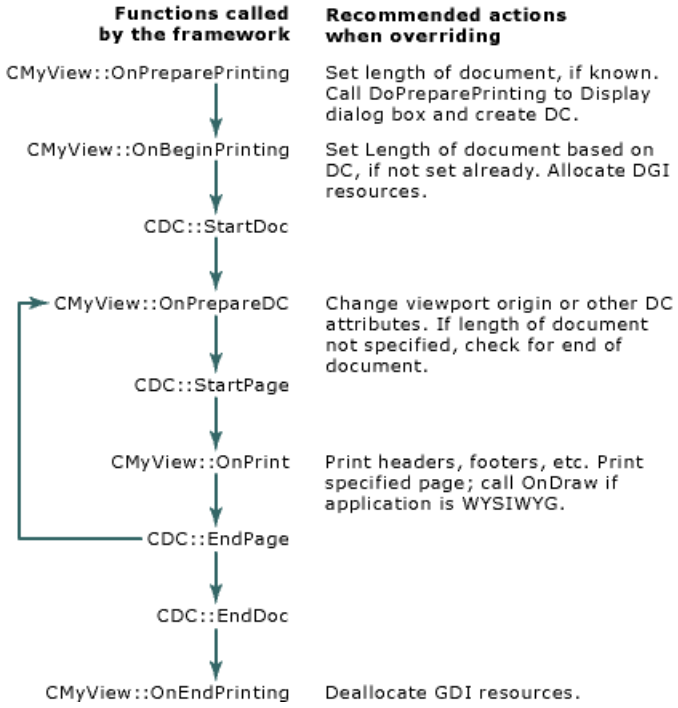

Pagination

The framework stores much of the information about a print job in a CPrintInfo structure. Several of the values in **CPrintInfo** pertain to pagination; these values are accessible as shown in the following table.

Page Number Information Stored in CPrintInfo

Member variable or function name(s)	Page number referenced
GetMinPage/SetMinPage	First page of document
GetMaxPage/SetMaxPage	Last page of document
GetFromPage	First page to be printed
GetToPage	Last page to be printed
m_nCurPage	Page currently being printed

Page numbers start at 1, that is, the first page is numbered 1, not 0. For more information about these and other members of CPrintInfo, see the *MFC Reference*.

At the beginning of the printing process, the framework calls the view's OnPreparePrinting member function, passing a pointer to a **CPrintInfo** structure. The Application Wizard provides an implementation of **OnPreparePrinting** that calls DoPreparePrinting, another member function of **CView**. **DoPreparePrinting** is the function that displays the Print dialog box and creates a printer device context.

At this point the application doesn't know how many pages are in the document. It uses the default values 1 and 0xFFFF for the numbers of the first and last page of the document. If you know how many pages your document has, override **OnPreparePrinting** and call SetMaxPage for the **CPrintInfo** structure before you send it to **DoPreparePrinting**. This lets you specify the length of your document.

DoPreparePrinting then displays the Print dialog box. When it returns, the **CPrintInfo** structure contains the values specified by the user. If the user wishes to print only a selected range of pages, he or she can specify the starting and ending page numbers in the Print dialog box. The framework retrieves these values using the **GetFromPage** and **GetToPage** functions of CPrintInfo. If the user doesn't specify a page range, the framework calls **GetMinPage** and **GetMaxPage** and uses the values returned to print the entire document.

For each page of a document to be printed, the framework calls two member functions in your view class, OnPrepareDC and OnPrint, and passes each function two parameters: a pointer to a CDC object and a pointer to a **CPrintInfo** structure. Each time the framework calls **OnPrepareDC** and **OnPrint**, it passes a different value in the **m_nCurPage** member of the **CPrintInfo** structure. In this way the framework tells the view which page should be printed.

The OnPrepareDC member function is also used for screen display. It makes adjustments to the device context before drawing takes place. **OnPrepareDC** serves a similar role in printing, but there are a couple of differences: first, the **CDC** object represents a printer device context instead of a screen device context, and second, a **CPrintInfo** object is passed as a second parameter. (This parameter is **NULL** when **OnPrepareDC** is called for screen display.) Override **OnPrepareDC** to make adjustments to the device context based on which page is being printed. For example, you can move the viewport origin and the clipping region to ensure that the appropriate portion of the document gets printed.

The OnPrint member function performs the actual printing of the page. The article How Default Printing Is Done shows how the framework calls OnDraw with a printer device context to perform printing. More precisely, the framework calls **OnPrint** with a **CPrintInfo** structure and a device context, and **OnPrint** passes the device context to **OnDraw**. Override **OnPrint** to perform any rendering that should be done only during printing and not for screen display. For example, to print headers or footers (see the article Headers and Footers for more information). Then call **OnDraw** from the override of **OnPrint** to do the rendering common to both screen display and printing.

The fact that **OnDraw** does the rendering for both screen display and printing means that your application is WYSIWYG: "What you see is what you get." However, suppose you aren't writing a WYSIWYG application. For example, consider a text editor that uses a bold font for printing but displays control codes to indicate bold text on the screen. In such a situation, you use **OnDraw** strictly for screen display. When you override **OnPrint**, substitute the call to **OnDraw** with a call to a separate drawing function. That function draws the document the way it appears on paper, using the attributes that you don't display on the screen.

Printer Pages vs. Document Pages

When you refer to page numbers, it's sometimes necessary to distinguish between the printer's concept of a page and a document's concept of a page. From the point of view of the printer, a page is one sheet of paper. However, one sheet of paper doesn't necessarily equal one page of the document. For example, if you're printing a newsletter, where the sheets are to be folded, one sheet of paper might contain both the first and last pages of the document, side by side. Similarly, if you're printing a spreadsheet, the document doesn't consist of pages at all. Instead, one sheet of paper might contain rows 1 through 20, columns 6 through 10.

All the page numbers in the CPrintInfo structure refer to printer pages. The framework calls **OnPrepareDC** and **OnPrint** once for each sheet of paper that will pass through the printer. When you override the OnPreparePrinting function to specify the length of the document, you must use printer pages. If there is a one-to-one correspondence (that is, one printer page equals one document page), then this is easy. If, on the other hand, document pages and printer pages do not directly correspond, you must translate between them. For example, consider printing a spreadsheet. When overriding **OnPreparePrinting**, you must calculate how many sheets of paper will be required to print the entire spreadsheet and then use that value when calling the **SetMaxPage** member function of **CPrintInfo**. Similarly, when overriding **OnPrepareDC**, you must translate **m_nCurPage** into the range of rows and columns that will appear on that particular sheet and then adjust the viewport origin accordingly.

Print-Time Pagination

In some situations, your view class may not know in advance how long the document is until it has actually been printed. For example, suppose your application isn't WYSIWYG, so a document's length on the screen doesn't correspond to its length when printed.

This causes a problem when you override OnPreparePrinting for your view class: you can't pass a value to the **SetMaxPage** function of the CPrintInfo structure, because you don't know the length of a document. If the user doesn't specify a page number to stop at using the Print dialog box, the framework doesn't know when to stop the print loop. The only way to determine when to stop the print loop is to print out the document and see when it ends. Your view class must check for the end of the document while it is being printed, and then inform the framework when the end is reached.

The framework relies on your view class's OnPrepareDC function to tell it when to stop. After each call to **OnPrepareDC**, the framework checks a member of the **CPrintInfo** structure called **m_bContinuePrinting**. Its default value is **TRUE.** As long as it remains so, the framework continues the print loop. If it is set to **FALSE**, the framework stops. To perform print-time pagination, override **OnPrepareDC** to check whether the end of the document has been reached, and set **m_bContinuePrinting** to **FALSE** when it has.

The default implementation of **OnPrepareDC** sets **m_bContinuePrinting** to **FALSE** if the current page is greater than 1. This means that if the length of the document wasn't specified, the framework assumes the document is one page long. One consequence of this is that you must be careful if you call the base class version of **OnPrepareDC**. Do not assume that **m_bContinuePrinting** will be **TRUE** after calling the base class version.

What do you want to know more about?

- Headers and footers
- Allocating GDI resources

See Also

Reference

Printing
CView Class
CDC Class

Headers and Footers

This article explains how to add headers and footers to a printed document.

When you look at a document on the screen, the name of the document and your current location in the document are commonly displayed in a title bar and a status bar. When looking at a printed copy of a document, it's useful to have the name and page number shown in a header or footer. This is a common way in which even WYSIWYG programs differ in how they perform printing and screen display.

The OnPrint member function is the appropriate place to print headers or footers because it is called for each page, and because it is called only for printing, not for screen display. You can define a separate function to print a header or footer, and pass it the printer device context from **OnPrint**. You might need to adjust the window origin or extent before calling OnDraw to avoid having the body of the page overlap the header or footer. You might also have to modify **OnDraw** because the amount of the document that fits on the page could be reduced.

One way to compensate for the area taken by the header or footer is to use the **m_rectDraw** member of CPrintInfo. Each time a page is printed, this member is initialized with the usable area of the page. If you print a header or footer before printing the body of the page, you can reduce the size of the rectangle stored in **m_rectDraw** to account for the area taken by the header or footer. Then **OnPrint** can refer to **m_rectDraw** to find out how much area remains for printing the body of the page.

You cannot print a header, or anything else, from OnPrepareDC, because it is called before the **StartPage** member function of CDC has been called. At that point, the printer device context is considered to be at a page boundary. You can perform printing only from the **OnPrint** member function.

What do you want to know more about?

- Printing multipage documents
- Allocating GDI resources for printing

See Also

Reference

Printing

Allocating GDI Resources

This article explains how to allocate and deallocate the Windows graphics device interface (GDI) objects needed for printing.

Note
GDI+ is included with Windows XP and is available as a redistributable for Windows NT 4.0 SP6, Windows 2000, Windows 98, and Windows Me. To download the latest redistributable, see http://www.microsoft.com/msdownload/platformsdk/sdkupdate/psdkredist.htm. For more information, see the GDI+ SDK documentation in MSDN: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/gdicpp/GDIPlus/GDIPlus.asp.

Suppose you need to use certain fonts, pens, or other GDI objects for printing, but not for screen display. Because of the memory they require, it's inefficient to allocate these objects when the application starts up. When the application isn't printing a document, that memory might be needed for other purposes. It's better to allocate them when printing begins, and then delete them when printing ends.

To allocate these GDI objects, override the OnBeginPrinting member function. This function is well suited to this purpose for two reasons: the framework calls this function once at the beginning of each print job and, unlike OnPreparePrinting, this function has access to the CDC object representing the printer device driver. You can store these objects for use during the print job by defining member variables in your view class that point to GDI objects (for example, **CFont *** members, and so on).

To use the GDI objects you've created, select them into the printer device context in the OnPrint member function. If you need different GDI objects for different pages of the document, you can examine the **m_nCurPage** member of the CPrintInfo structure and select the GDI object accordingly. If you need a GDI object for several consecutive pages, Windows requires that you select it into the device context each time **OnPrint** is called.

To deallocate these GDI objects, override the OnEndPrinting member function. The framework calls this function at the end of each print job, giving you the opportunity to deallocate printing-specific GDI objects before the application returns to other tasks.

See Also
Reference
Printing
How Default Printing Is Done

Print Preview Architecture

This article explains how the MFC framework implements print preview functionality. Topics covered include:

- Print preview process
- Modifying print preview

Print preview is somewhat different from screen display and printing because, instead of directly drawing an image on a device, the application must simulate the printer using the screen. To accommodate this, the Microsoft Foundation Class Library defines a special (undocumented) class derived from CDC, called **CPreviewDC**. All **CDC** objects contain two device contexts, but usually they are identical. In a **CPreviewDC** object, they are different: the first represents the printer being simulated, and the second represents the screen on which output is actually displayed.

The Print Preview Process

When the user selects the Print Preview command from the **File** menu, the framework creates a **CPreviewDC** object. Whenever your application performs an operation that sets a characteristic of the printer device context, the framework also performs a similar operation on the screen device context. For example, if your application selects a font for printing, the framework selects a font for screen display that simulates the printer font. Whenever your application would send output to the printer, the framework instead sends the output to the screen.

Print preview also differs from printing in the order that each draws the pages of a document. During printing, the framework continues a print loop until a certain range of pages has been rendered. During print preview, one or two pages are displayed at any time, and then the application waits; no further pages are displayed until the user responds. During print preview, the application must also respond to **WM_PAINT** messages, just as it does during ordinary screen display.

The OnPreparePrinting function is called when preview mode is invoked, just as it is at the beginning of a print job. The CPrintInfo structure passed to the function contains several members whose values you can set to adjust certain characteristics of the print preview operation. For example, you can set the **m_nNumPreviewPages** member to specify whether you want to preview the document in one-page or two-page mode.

Modifying Print Preview

You can modify the behavior and appearance of print preview in a number of ways rather easily. For example, you can, among other things:

- Cause the print preview window to display a scroll bar for easy access to any page of the document.
- Cause print preview to maintain the user's position in the document by beginning its display at the current page.
- Cause different initialization to be performed for print preview and printing.
- Cause print preview to display page numbers in your own formats.

If you know how long the document is and call **SetMaxPage** with the appropriate value, the framework can use this information in preview mode as well as during printing. Once the framework knows the length of the document, it can provide the preview window with a scroll bar, allowing the user to page back and forth through the document in preview mode. If you haven't set the length of the document, the framework cannot position the scroll box to indicate the current position, so the framework doesn't add a scroll bar. In this case, the user must use the Next Page and Previous Page buttons on the preview window's control bar to page through the document.

For print preview, you may find it useful to assign a value to the **m_nCurPage** member of **CPrintInfo**, even though you would never do so for ordinary printing. During ordinary printing, this member carries information from the framework to your view class. This is how the framework tells the view which page should be printed.

By contrast, when print preview mode is started, the **m_nCurPage** member carries information in the opposite direction: from the view to the framework. The framework uses the value of this member to determine which page should be previewed first. The default value of this member is 1, so the first page of the document is displayed initially. You can override **OnPreparePrinting** to set this member to the number of the page being viewed at the time the Print Preview command was invoked. This way, the application maintains the user's current position when moving from normal display mode to print preview mode.

Sometimes you may want **OnPreparePrinting** to perform different initialization depending on whether it is called for a print job or for print preview. You can determine this by examining the **m_bPreview** member variable in the **CPrintInfo** structure.

This member is set to **TRUE** when print preview is invoked.

The **CPrintInfo** structure also contains a member named **m_strPageDesc**, which is used to format the strings displayed at the bottom of the screen in single-page and multiple-page modes. By default these strings are of the form "Page *n*" and "Pages *n* - *m*," but you can modify **m_strPageDesc** from within **OnPreparePrinting** and set the strings to something more elaborate. See CPrintInfo in the *MFC Reference* for more information.

See Also

Reference

Printing and Print Preview
Printing
CView Class
CDC Class

Property Sheets (MFC)

This family of articles explains how to implement support for property sheets in MFC applications. A property sheet, also known as a tab dialog box, provides a way to manage large numbers of controls in a dialog box. The property sheet contains property pages, each based on a separate dialog template resource. You can divide your dialog box's controls into logical groups and put each group on its own property page.

What do you want to know more about?

- Property sheets and property pages
- Using Property Sheets in Your Application
- Adding Controls to a Property Sheet (as opposed to a property page)
- Exchanging data between a property sheet and your program
- Creating a modeless property sheet
- Handling the Apply button
- Property Sheets as Wizards
- Class CPropertySheet
- Class CPropertyPage
- Sample

See Also

Other Resources

General Concepts
User Interface

Property Sheets and Property Pages in MFC

A property sheet, also known as a tab dialog box, is a dialog box that contains property pages. Each page contains controls, is based on a dialog template resource, and appears on a tab, an enclosure that looks like a file folder with a tab at the top. The tab names the page and indicates its purpose. Users click a tab in the property sheet to select a set of controls.

Use pages to group the controls in the property sheet into meaningful sets. The containing property sheet normally has several controls of its own. These apply to all pages.

For example, several dialog boxes in the development environment are property sheets, including the Customize dialog box.

Property sheets are based on class CPropertySheet. Property pages are based on class CPropertyPage.

A property sheet is a special kind of dialog box that is generally used to modify the attributes of some external object, such as the current selection in a view. The property sheet has three main parts: the containing dialog box, one or more property pages shown one at a time, and a tab at the top of each page that the user clicks to select that page. Property sheets are useful for situations where you have a number of similar groups of settings or options to change. A property sheet allows a large amount of information to be grouped in an easily understood fashion.

See Also

Concepts

Property Sheets (MFC)

Using Property Sheets in Your Application

To use a property sheet in your application, complete the following steps:

1. Create a dialog template resource for each property page. Keep in mind that the user may be switching from one page to another, so lay out each page as consistently as possible.

   The dialog templates for all pages do not have to be the same size. The framework uses the size of the largest page to determine how much space to allocate in the property sheet for the property pages.

   When you create the dialog template resource for a property page, you must specify the following styles in the Dialog Properties property sheet:

   - Set the **Caption** edit box on the **General** page to the text you wish to appear in the tab for this page.
   - Set the **Style** list box on the **Styles** page to **Child**.
   - Set the **Border** list box on the **Styles** page to **Thin**.
   - Ensure that the **Titlebar** check box on the **Styles** page is selected.
   - Ensure that the **Disabled** check box on the **More Styles** page is selected.

2. Create a CPropertyPage-derived class corresponding to each property page dialog template. See Adding a Class. Choose **CPropertyPage** as the base class.

3. Create member variables to hold the values for this property page. The process for adding member variables to a property page is exactly the same as adding member variables to a dialog box, because a property page is a specialized dialog box. For more information, see Defining Member Variables for Dialog Controls.

4. Construct a CPropertySheet object in your source code. Usually, you construct the **CPropertySheet** object in the handler for the command that displays the property sheet. This object represents the entire property sheet. If you create a modal property sheet with the DoModal function, the framework supplies three command buttons by default: OK, Cancel, and Apply. The framework creates no command buttons for modeless property sheets created with the Create function. You do not need to derive a class from **CPropertySheet** unless you want to either add other controls (such as a preview window) or display a modeless property sheet. This step is necessary for modeless property sheets because they do not contain any default controls that could be used to close the property sheet.

5. For each page to be added to the property sheet, do the following:

   - Construct one object for each **CPropertyPage**-derived class that you created earlier in this process.
   - Call CPropertySheet::AddPage for each page.

   Typically, the object that creates the **CPropertySheet** also creates the **CPropertyPage** objects in this step. However, if you implement a **CPropertySheet**-derived class, you can embed the **CPropertyPage** objects in the **CPropertySheet** object and call **AddPage** for each page from the **CPropertySheet**-derived class constructor. **AddPage** adds the **CPropertyPage** object to the property sheet's list of pages but does not actually create the window for that page. Therefore, it is not necessary to wait until creation of the property sheet window to call **AddPage**; you can call **AddPage** from the property sheet's constructor.

   By default, if a property sheet has more tabs than will fit in a single row of the property sheet, the tabs will stack in multiple rows. To disable stacking, call CPropertySheet::EnableStackedTabs with the parameter set to **FALSE**. You must call **EnableStackedTabs** when you create the property sheet.

6. Call CPropertySheet::DoModal or Create to display the property sheet. Call **DoModal** to create a property sheet as a modal dialog box. Call **Create** to create the property sheet as a modeless dialog box.

7. Exchange data between property pages and the owner of the property sheet. This is explained in the article Exchanging Data.

For an example of how to use property sheets, see the MFC General sample PROPDLG.

See Also
Concepts
Property Sheets (MFC)

Adding Controls to a Property Sheet

By default, a property sheet allocates window area for the property pages, the tab index, and the OK, Cancel, and Apply buttons. (A modeless property sheet does not have the OK, Cancel, and Apply buttons.) You can add other controls to the property sheet. For example, you can add a preview window to the right of the property page area to show the user what the current settings would look like if applied to an external object.

You can add controls to the property sheet dialog in the **OnCreate** handler. Accommodating additional controls usually requires expanding the size of the property sheet dialog. After calling the base class **CPropertySheet::OnCreate**, call GetWindowRect to get the width and height of the currently allocated property sheet window, expand the rectangle's dimensions, and call MoveWindow to change the size of the property sheet window.

See Also

Reference

CPropertyPage Class

CPropertySheet Class

Concepts

Property Sheets (MFC)

Exchanging Data

As with most dialog boxes, the exchange of data between the property sheet and the application is one of the most important functions of the property sheet. This article describes how to accomplish this task.

Exchanging data with a property sheet is actually a matter of exchanging data with the individual property pages of the property sheet. The procedure for exchanging data with a property page is the same as for exchanging data with a dialog box, since a CPropertyPage object is just a specialized CDialog object. The procedure takes advantage of the framework's dialog data exchange (DDX) facility, which exchanges data between controls in a dialog box and member variables of the dialog box object.

The important difference between exchanging data with a property sheet and with a normal dialog box is that the property sheet has multiple pages, so you must exchange data with all the pages in the property sheet. For more information on DDX, see Dialog Data Exchange and Validation.

The following example illustrates exchanging data between a view and two pages of a property sheet:

```
void CMyView::DoModalPropertySheet()
{
   CPropertySheet propsheet;
   CMyFirstPage pageFirst; // derived from CPropertyPage
   CMySecondPage pageSecond; // derived from CPropertyPage

   // Move member data from the view (or from the currently
   // selected object in the view, for example).
   pageFirst.m_nMember1 = m_nMember1;
   pageFirst.m_nMember2 = m_nMember2;

   pageSecond.m_strMember3 = m_strMember3;
   pageSecond.m_strMember4 = m_strMember4;

   propsheet.AddPage(&pageFirst);
   propsheet.AddPage(&pageSecond);

   if (propsheeet.DoModal() == IDOK)
   {
      m_nMember1 = pageFirst.m_nMember1;
      m_nMember2 = pageFirst.m_nMember2;
      m_strMember3 = pageSecond.m_strMember3;
      m_strMember4 = pageSecond.m_strMember4;
      GetDocument()->SetModifiedFlag();
      GetDocument()->UpdateAllViews(NULL);
   }
}
```

See Also

Concepts

Property Sheets (MFC)

Creating a Modeless Property Sheet

Normally, the property sheets you create will be modal. When using a modal property sheet, the user must close the property sheet before using any other part of the application. This article describes methods you can use to create a modeless property sheet that allows the user to keep the property sheet open while using other parts of the application.

To display a property sheet as a modeless dialog box instead of as a modal dialog box, call CPropertySheet::Create instead of DoModal. You must also implement some extra tasks to support a modeless property sheet.

One of the additional tasks is exchanging data between the property sheet and the external object it is modifying when the property sheet is open. This is generally the same task as for standard modeless dialog boxes. Part of this task is implementing a channel of communication between the modeless property sheet and the external object to which the property settings apply. This implementation is far easier if you derive a class from CPropertySheet for your modeless property sheet. This article assumes you have done so.

One method for communicating between the modeless property sheet and the external object (the current selection in a view, for example) is to define a pointer from the property sheet to the external object. Define a function (called something like `SetMyExternalObject`) in the **CPropertySheet**-derived class to change the pointer whenever the focus changes from one external object to another. The `SetMyExternalObject` function needs to reset the settings for each property page to reflect the newly selected external object. To accomplish this, the `SetMyExternalObject` function must be able to access the CPropertyPage objects belonging to the **CPropertySheet** class.

The most convenient way to provide access to property pages within a property sheet is to embed the **CPropertyPage** objects in the **CPropertySheet**-derived object. Embedding **CPropertyPage** objects in the **CPropertySheet**-derived object differs from the typical design for modal dialog boxes, where the owner of the property sheet creates the **CPropertyPage** objects and passes them to the property sheet via CPropertySheet::AddPage.

There are many user-interface alternatives for determining when the settings of the modeless property sheet should be applied to an external object. One alternative is to apply the settings of the current property page whenever the user changes any value. Another alternative is to provide an Apply button, which allows the user to accumulate changes in the property pages before committing them to the external object. For information on ways to handle the Apply button, see the article Handling the Apply Button.

See Also

Reference

Exchanging Data

Concepts

Property Sheets (MFC)

Life Cycle of a Dialog Box

Handling the Apply Button

Property sheets have a capability that standard dialog boxes do not: They allow the user to apply changes they have made before closing the property sheet. This is done using the Apply button. This article discusses methods you can use to implement this feature properly.

Modal dialog boxes usually apply the settings to an external object when the user clicks OK to close the dialog box. The same is true for a property sheet: When the user clicks OK, the new settings in the property sheet take effect.

However, you may want to allow the user to save settings without having to close the property sheet dialog box. This is the function of the Apply button. The Apply button applies the current settings in all of the property pages to the external object, as opposed to applying only the current settings of the currently active page.

By default, the Apply button is always disabled. You must write code to enable the Apply button at the appropriate times, and you must write code to implement the effect of Apply, as explained below.

If you do not wish to offer the Apply functionality to the user, it is not necessary to remove the Apply button. You can leave it disabled, as will be common among applications that use standard property sheet support available in future versions of Windows.

To report a page as being modified and enable the Apply button, call **CPropertyPage::SetModified(TRUE)**. If any of the pages report being modified, the Apply button will remain enabled, regardless of whether the currently active page has been modified.

You should call CPropertyPage::SetModified whenever the user changes any settings in the page. One way to detect when a user changes a setting in the page is to implement change notification handlers for each of the controls in the property page, such as **EN_CHANGE** or **BN_CLICKED**.

To implement the effect of the Apply button, the property sheet must tell its owner, or some other external object in the application, to apply the current settings in the property pages. At the same time, the property sheet should disable the Apply button by calling **CPropertyPage::SetModified(FALSE)** for all pages that applied their modifications to the external object.

For an example of this process, see the MFC General sample PROPDLG.

See Also
Concepts
Property Sheets (MFC)

Property Sheets as Wizards

A key characteristic of a wizard property sheet is that navigation is provided with Next or Finish, Back, and Cancel buttons instead of tabs. You need to call CPropertySheet::SetWizardMode before calling CPropertySheet::DoModal on the property sheet object to take advantage of this feature.

The user receives the same CPropertyPage::OnSetActive and CPropertyPage::OnKillActive notifications while moving from one page to another page. Next and Finish buttons are mutually exclusive controls; that is, only one of them will be shown at a time. On the first page, the Next button should be enabled. If the user is on the last page, the Finish button should be enabled. This is not done automatically by the framework. You have to call CPropertySheet::SetWizardButton on the last page to achieve this.

To display all of the default buttons, you mush show the Finish button and move the Next button. Then move the Back button so that its relative position to the Next button is maintained. For more explanation, search for KB article Q143210. Knowledge Base articles are available in the MSDN Library.

Example

```
CPropertySheet sheet;
// CMyPage1 and CMyPage2 are derived from CPropertyPage
CMyPage1 page1;
CMyPage2 page2;

sheet.AddPage(&page1);
sheet.AddPage(&page2);
sheet.SetWizardMode();
sheet.DoModal();
```

See Also

Concepts

Property Sheets (MFC)

Status Bars

Status bars give your application a place to display messages and useful information to the user without interrupting the user's work. Typically displayed at the bottom of a window, status bars have "panes," which include "indicators" and a "message line." The indicators give the status of such things as SCROLL LOCK, whether macro recording is turned on, and so on. The message line on the status bar can display information about program status or about a toolbar button or menu item that the user is pointing to with the mouse.

Create a status bar in your program by selecting the **Initial Status Bar** option in the MFC Application Wizard.

What do you want to know more about?

- Status bar implementation in MFC
- Updating the text of a status bar pane
- Creating a new status-bar pane (Updating the text of a status bar pane)
- Making a status-bar pane display text (Updating the text of a status bar pane)
- Display command information in the status bar

See Also

Other Resources

General Concepts
User Interface

Status Bar Implementation in MFC

A CStatusBar object is a control bar with a row of text output panes. The output panes are commonly used as message lines and as status indicators. Examples include the menu help-message lines that briefly explain the selected menu command and the indicators that show the status of the SCROLL LOCK, NUM LOCK, and other keys.

As of MFC version 4.0, status bars are implemented using class CStatusBarCtrl, which encapsulates a status bar common control. For backward compatibility, MFC retains the older status bar implementation in class **COldStatusBar**. The documentation for earlier versions of MFC describes **COldStatusBar** under **CStatusBar**.

CStatusBar::GetStatusBarCtrl, a member function new to MFC 4.0, allows you to take advantage of the Windows common control's support for status bar customization and additional functionality. **CStatusBar** member functions give you most of the functionality of the Windows common controls; however, when you call **GetStatusBarCtrl**, you can give your status bars even more of the characteristics of a status bar. When you call **GetStatusBarCtrl**, it will return a reference to a **CStatusBarCtrl** object. You can use that reference to manipulate the status bar control.

The following figure shows a status bar that displays several indicators.

A Status Bar

Like the toolbar, the status-bar object is embedded in its parent frame window and is constructed automatically when the frame window is constructed. The status bar, like all control bars, is destroyed automatically as well when the parent frame is destroyed.

What do you want to know more about?

- Updating the text of a status bar pane
- MFC classes CStatusBar and CStatusBarCtrl
- Control bars
- Dialog bars
- Toolbars (MFC Toolbar Implementation)

See Also

Concepts

Status Bars

Updating the Text of a Status-Bar Pane

This article explains how to change the text that appears in an MFC status bar pane. A status bar — a window object of class CStatusBar — contains several "panes." Each pane is a rectangular area of the status bar that you can use to display information. For example, many applications display the status of the CAPS LOCK, NUM LOCK, and other keys in the rightmost panes. Applications also often display informative text in the leftmost pane (pane 0), sometimes called the "message pane." For example, the default MFC status bar uses the message pane to display a string explaining the currently selected menu item or toolbar button. The figure in Status Bars shows a status bar from an Application Wizard-created MFC application.

By default, MFC does not enable a **CStatusBar** pane when it creates the pane. To activate a pane, you must use the **ON_UPDATE_COMMAND_UI** macro for each pane on the status bar and update the panes. Because panes do not send **WM_COMMAND** messages (they aren't like toolbar buttons), you must type the code manually.

For example, suppose one pane has `ID_INDICATOR_PAGE` as its command identifier and that it contains the current page number in a document. The following procedure describes how to create a new pane in the status bar.

To make a new pane

1. Define the pane's command ID.

   On the **View** menu, click **Resource View**. Right-click the project resource and click **Resource Symbols**. In the Resource Symbols dialog box, click **New**. Type a command ID name: for example, `ID_INDICATOR_PAGE`. Specify a value for the ID, or accept the value suggested by the Resource Symbols dialog box. For example, for `ID_INDICATOR_PAGE`, accept the default value. Close the Resource Symbols dialog box.

2. Define a default string to display in the pane.

   With Resource View open, double-click **String Table** in the window that lists resource types for your application. With the **String Table** editor open, choose **New String** from the **Insert** menu. In the String Properties window, select your pane's command ID (for example, `ID_INDICATOR_PAGE`) and type a default string value, such as "Page ". Close the string editor. (You need a default string to avoid a compiler error.)

3. Add the pane to the **indicators** array.

   In file MAINFRM.CPP, locate the **indicators** array. This array lists command IDs for all of the status bar's indicators, in order from left to right. At the appropriate point in the array, enter your pane's command ID, as shown here for `ID_INDICATOR_PAGE`:

   ```
   static UINT BASED_CODE indicators[] =
   {
       ID_SEPARATOR,           // status line indicator
       ID_INDICATOR_CAPS,
       ID_INDICATOR_NUM,
       ID_INDICATOR_SCRL,
       ID_INDICATOR_PAGE,
   };
   ```

The recommended way to display text in a pane is to call the **SetText** member function of class **CCmdUI** in an update handler function for the pane. For example, you might want to set up an integer variable `m_nPage` that contains the current page number and use **SetText** to set the pane's text to a string version of that number.

Note
The **SetText** approach is recommended. It is possible to perform this task at a slightly lower level by calling the **CStatusBar** member function **SetPaneText**. Even so, you still need an update handler. Without such a handler for the pane, MFC automa tically disables the pane, erasing its content.

The following procedure shows how to use an update handler function to display text in a pane.

To make a pane display text

1. Add a command update handler for the command.

   Manually add a prototype for the handler, as shown here for `ID_INDICATOR_PAGE` (in MAINFRM.H):

   ```
   afx_msg void OnUpdatePage(CCmdUI *pCmdUI);
   ```

   In the appropriate .CPP file, add the handler's definition, as shown here for `ID_INDICATOR_PAGE` (in MAINFRM.CPP):

   ```
   void CMainFrame::OnUpdatePage(CCmdUI *pCmdUI)
   {
       pCmdUI->Enable();
   }
   ```

   In the appropriate message map, add the **ON_UPDATE_COMMAND_UI** macro, as shown here for `ID_INDICATOR_PAGE` (in MAINFRM.CPP):

   ```
   ON_UPDATE_COMMAND_UI(ID_INDICATOR_PAGE, OnUpdatePage)
   ```

2. Add code to the handler to display your text.

   For `ID_INDICATOR_PAGE`, expand the `OnUpdatePage` handler from step 1 above, adding the last three lines:

   ```
   void CMainFrame::OnUpdatePage(CCmdUI *pCmdUI)
   {
       pCmdUI->Enable();
       CString strPage;
       strPage.Format( "Page %d", m_nPage );
       pCmdUI->SetText( strPage );
   }
   ```

Once you define the value of the `m_nPage` member variable (of class `CMainFrame`), this technique causes the page number to appear in the pane during idle processing in the same manner that the application updates other indicators. If `m_nPage` changes, the display changes during the next idle loop.

What do you want to know more about?

- Updating user-interface objects (how to update toolbar buttons and menu items as program conditions change)

See Also

Reference

CStatusBar Class

Concepts

Status Bar Implementation in MFC

Tool Tips

The procedures are distinct for adding tool tips to controls contained in windows derived from MFC class CFrameWnd and windows not derived from **CFrameWnd**.

What do you want to know more about?

Tool tips for controls in a window that is:

- Toolbar Tooltips (derived from CFrameWnd)
- Tooltips in Windows not derived from CFrameWnd

See Also

Other Resources

General Concepts
User Interface

Tool Tips in Windows Not Derived from CFrameWnd

This article family covers enabling tool tips for controls contained in a window that is not derived from CFrameWnd. The article Toolbars Tool Tips provides information about tool tips for controls in a **CFrameWnd**.

Topics covered in this article family include:

- Enabling Tool Tips
- Handling TTN_NEEDTEXT Notification for Tool Tips
- The TOOLTIPTEXT Structure

Tool tips are automatically displayed for buttons and other controls contained in a parent window derived from **CFrameWnd**. This is because **CFrameWnd** has a default handler for the TTN_GETDISPINFO notification, which handles **TTN_NEEDTEXT** notifications from tool tip controls associated with controls.

However, this default handler is not called when the **TTN_NEEDTEXT** notification is sent from a tool tip control associated with a control in a window that is not a **CFrameWnd**, such as a control on a dialog box or a form view. Therefore, it is necessary for you to provide a handler function for the **TTN_NEEDTEXT** notification message in order to display tool tips for child controls.

The default tool tips provided for your windows by CWnd::EnableToolTips do not have text associated with them. To retrieve text for the tool tip to display, the **TTN_NEEDTEXT** notification is sent to the tool tip control's parent window just before the tool tip window is displayed. If there is no handler for this message to assign some value to the **pszText** member of the **TOOLTIPTEXT** structure, there will be no text displayed for the tool tip.

See Also

Concepts

Tool Tips

Enabling Tool Tips

You can enable tool tip support for the child controls of a window (such as the controls on a form view or dialog box).

To enable tool tips for the child controls of a window

1. Call **EnableToolTips** for the window for which you want to provide tool tips.
2. Provide a string for each control in your TTN_NEEDTEXT notification handler. The handler is in the message map of the window that contains the child controls (for example, your form view class). This handler should call a function that identifies the control and sets **pszText** to specify the text used by the tool tip control.

See Also

Concepts

Tool Tips in Windows Not Derived from CFrameWnd

Handling TTN_NEEDTEXT Notification for Tool Tips

As part of enabling tool tips, you handle the **TTN_NEEDTEXT** message by adding the following entry to your owner window's message map:

```
ON_NOTIFY_EX( TTN_NEEDTEXT, 0, memberFxn )
```

memberFxn

The member function to be called when text is needed for this button.

Note that the ID of a tool tip is always 0.

Declare your handler function in the class definition as follows:

```
BOOL CMyClass::memberFxn( UINT id, NMHDR * pTTTStruct, LRESULT * pResult );
```

where the italicized parameters are:

id

Identifier of the control that sent the notification. Not used. The control id is taken from the **NMHDR** structure.

pTTTStruct

A pointer to the NMTTDISPINFO structure. This structure is also discussed further in The TOOLTIPTEXT Structure.

pResult

A pointer to result code you can set before you return. **TTN_NEEDTEXT** handlers can ignore the *pResult* parameter.

As an example of a form-view notification handler:

```
BOOL CMyFormView::OnToolTipNotify( UINT id, NMHDR * pNMHDR, LRESULT * pResult )
{
    TOOLTIPTEXT *pTTT = (TOOLTIPTEXT *)pNMHDR;
    UINT nID =pNMHDR->idFrom;
    if (pTTT->uFlags & TTF_IDISHWND)
    {
        // idFrom is actually the HWND of the tool
        nID = ::GetDlgCtrlID((HWND)nID);
        if(nID)
        {
            pTTT->lpszText = MAKEINTRESOURCE(nID);
            pTTT->hinst = AfxGetResourceHandle();
            return(TRUE);
        }
    }
    return(FALSE);
}
void CTestView::OnInitialUpdate()
{
    CMyFormView::OnInitialUpdate();
    EnableToolTips(TRUE);
}
```

See Also

Concepts

Tool Tips in Windows Not Derived from CFrameWnd

The TOOLTIPTEXT Structure

In writing your tool tip notification handler, you need to use the **TOOLTIPTEXT** structure. The members of the **TOOLTIPTEXT** structure are:

```
typedef struct {
    NMHDR     hdr;        // required for all WM_NOTIFY messages
    LPTSTR    lpszText;   // see below
    WCHAR     szText[80]; // buffer for tool tip text
    HINSTANCE hinst;      // see below
    UINT      uflags;     // flag indicating how to interpret the
                          // idFrom member of the NMHDR structure
                          // that is included in the structure
} TOOLTIPTEXT, FAR *LPTOOLTIPTEXT;
```

hdr

Identifies the tool that needs text. The only member of this structure you might need is the control's command ID. The control's command ID will be in the **idFrom** member of the **NMHDR** structure, accessed with the syntax **hdr.idFrom**. See NMHDR for a discussion of members of the **NMHDR** structure.

lpszText

Address of a string to receive the text for a tool.

szText

Buffer that receives the tool tip text. An application can copy the text to this buffer as an alternative to specifying a string address.

hinst

Handle of the instance that contains a string resource to be used as the tool tip text. If **lpszText** is the address of the tool tip text, this member is **NULL**.

When you handle the **TTN_NEEDTEXT** notification message, specify the string to be displayed in one of the following ways:

- Copy the text to the buffer specified by the **szText** member.
- Copy the address of the buffer that contains the text to the **lpszText** member.
- Copy the identifier of a string resource to the **lpszText** member, and copy the handle of the instance that contains the resource to the **hinst** member.

See Also

Concepts

Tool Tips in Windows Not Derived from CFrameWnd

Toolbars

The toolbar family of articles describes MFC toolbars and how to create and use them.

What do you want to know more about?

- MFC toolbar implementation
- Toolbar fundamentals
- How to Update User-Interface Objects (enable/disable toolbar buttons)
- The CToolBar and CToolBarCtrl classes
- Sample
- Knowledge Base article Q232017, HOWTO: Add Text to Toolbar Buttons

See Also

Reference

Toolbar Editor

Other Resources

General Concepts

User Interface

Toolbar Sample List

See the following sample programs that illustrate using MFC's toolbars:

- SCRIBBLE
- CTRLBARS
- DOCKTOOL

See Also

Concepts

Toolbars

MFC Toolbar Implementation

A toolbar is a control bar that contains the bitmap images of controls. These images can behave like pushbuttons, check boxes, or radio buttons. MFC supplies class CToolbar to manage toolbars.

If you enable it, users of MFC toolbars can dock them to the edge of a window or "float" them anywhere within the application window. MFC doesn't support customizable toolbars like those in the development environment.

MFC also supports tool tips: small pop-up windows that describe a toolbar button's purpose when you position the mouse over the button. By default, when the user presses a toolbar button, a status string appears in the status bar (if there is one). You can activate "fly by" status bar updating to display the status string when the mouse is positioned over the button without pressing it.

Note
As of MFC version 4.0, toolbars and tool tips are implemented using Windows 95 and later functionality instead of the previous implementation specific to MFC.

For backward compatibility, MFC retains the older toolbar implementation in class **COldToolBar**. The documentation for earlier versions of MFC describe **COldToolBar** under **CToolBar**.

Create the first toolbar in your program by selecting the Toolbar option in the Application Wizard. You can also create additional toolbars.

The following are introduced in this article:

- Toolbar buttons
- Docking and floating toolbars
- Toolbars and tool tips
- The CToolBar and CToolBarCtrl classes
- The Toolbar bitmap

Toolbar Buttons

The buttons in a toolbar are analogous to the items in a menu. Both kinds of user-interface objects generate commands, which your program handles by providing handler functions. Often toolbar buttons duplicate the functionality of menu commands, providing an alternative user interface to the same functionality. Such duplication is arranged simply by giving the button and the menu item the same ID.

You can make the buttons in a toolbar appear and behave as pushbuttons, check boxes, or radio buttons. For more information, see class CToolBar.

Docking and Floating Toolbars

An MFC toolbar can:

- Remain stationary along one side of its parent window.
- Be dragged and "docked," or attached, by the user to any side or sides of the parent window you specify.
- Be "floated," or detached from the frame window, in its own mini-frame window so the user can move it around to any convenient position.
- Be resized while floating.

For more information, see the article Docking and Floating Toolbars.

Toolbars and Tool Tips

MFC toolbars can also be made to display "tool tips" — tiny popup windows containing a short text description of a toolbar button's purpose. As the user moves the mouse over a toolbar button, the tool tip window pops up to offer a hint. For more information, see the article Toolbar Tool Tips.

The CToolBar and CToolBarCtrl Classes

You manage your application's toolbars via class CToolBar. As of MFC version 4.0, **CToolBar** has been reimplemented to use the toolbar common control available under Windows 95 or later and Windows NT version 3.51 or later.

This reimplementation results in less MFC code for toolbars, because MFC makes use of the operating system support. The reimplementation also improves capability. You can use **CToolBar** member functions to manipulate toolbars, or you can obtain a reference to the underlying CToolBarCtrl object and call its member functions for toolbar customization and additional functionality.

Tip
If you have invested heavily in the older MFC implementation of **CToolBar**, that support is still available. See the article Using Your Old Toolbars.

Also see the MFC General sample DOCKTOOL.

The Toolbar Bitmap

Once constructed, a **CToolBar** object creates the toolbar image by loading a single bitmap that contains one image for each button. The Application Wizard creates a standard toolbar bitmap that you can customize with the Visual C++ toolbar editor.

What do you want to know more about?

- Toolbar fundamentals
- Docking and floating toolbars
- Toolbar tool tips
- Working with the Toolbar Control
- Using Your Old Toolbars
- The CToolBar and CToolBarCtrl classes

See Also

Reference

Toolbar Editor

Concepts

Toolbars

Toolbar Fundamentals

This article describes the fundamental MFC implementation that lets you add a default toolbar to your application by selecting an option in the Application Wizard. Topics covered include:

- The Application Wizard toolbar option
- The toolbar in code
- Editing the toolbar resource
- Multiple toolbars

The Application Wizard Toolbar Option

To get a single toolbar with default buttons, select the Standard Docking toolbar option on the page labeled User Interface Features. This adds code to your application that:

- Creates the toolbar object.
- Manages the toolbar, including its ability to dock or to float.

The Toolbar in Code

The toolbar is a CToolBar object declared as a data member of your application's **CMainFrame** class. In other words, the toolbar object is embedded in the main frame window object. This means that MFC creates the toolbar when it creates the frame window and destroys the toolbar when it destroys the frame window. The following partial class declaration, for a multiple document interface (MDI) application, shows data members for an embedded toolbar and an embedded status bar. It also shows the override of the **OnCreate** member function.

```
class CMainFrame : public CMDIFrameWnd
{
   // ...

// Implementation
   // ...

protected:  // control bar embedded members
   CStatusBar  m_wndStatusBar;
   CToolBar    m_wndToolBar;

// Generated message map functions
protected:
   afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
   DECLARE_MESSAGE_MAP()
};
```

Toolbar creation occurs in **CMainFrame::OnCreate**. MFC calls OnCreate after creating the window for the frame but before it becomes visible. The default **OnCreate** that the Application Wizard generates does the following toolbar tasks:

1. Calls the **CToolBar** object's Create member function to create the underlying CToolBarCtrl object.
2. Calls LoadToolBar to load the toolbar resource information.
3. Calls functions to enable docking, floating, and tool tips. For details about these calls, see the article Docking and Floating Toolbars.

Note

The MFC General sample DOCKTOOL includes illustrations of both old and new MFC toolbars. The toolbars that use **COldToolbar** require calls in step 2 to **LoadBitmap** (rather than **LoadToolBar**) and to **SetButtons**. The new toolbars require calls to **LoadToolBar**.

The docking, floating, and tool tips calls are optional. You can remove those lines from **OnCreate** if you prefer. The result is a

toolbar that remains fixed, unable to float or redock and unable to display tool tips.

Editing the Toolbar Resource

The default toolbar you get with the Application Wizard is based on an **RT_TOOLBAR** custom resource, introduced in MFC version 4.0. You can edit this resource with the toolbar editor. The editor lets you easily add, delete, and rearrange buttons. It contains a graphical editor for the buttons that is very similar to the general graphics editor in Visual C++. If you edited toolbars in previous versions of Visual C++, you'll find the task much easier now.

To connect a toolbar button to a command, you give the button a command ID, such as `ID_MYCOMMAND`. Specify the command ID in the button's property page in the toolbar editor. Then create a handler function for the command (see Mapping Messages to Functions for more information).

New CToolBar member functions work with the **RT_TOOLBAR** resource. LoadToolBar now takes the place of LoadBitmap to load the bitmap of the toolbar button images, and SetButtons to set the button styles and connect buttons with bitmap images.

For details about using the toolbar editor, see Toolbar Editor.

Multiple Toolbars

The Application Wizard provides you with one default toolbar. If you need more than one toolbar in your application, you can model your code for additional toolbars based on the wizard-generated code for the default toolbar.

If you want to display a toolbar as the result of a command, you'll need to:

- Create a new toolbar resource with the toolbar editor and load it in **OnCreate** with the LoadToolbar member function.
- Embed a new CToolBar object in your main frame window class.
- Make the appropriate function calls in **OnCreate** to dock or float the toolbar, set its styles, and so on.

What do you want to know more about?

- MFC Toolbar Implementation (overview information on toolbars)
- Docking and floating toolbars
- Toolbar tool tips
- The CToolBar and CToolBarCtrl classes
- Working with the toolbar control
- Using your old toolbars

See Also

Concepts

MFC Toolbar Implementation

Docking and Floating Toolbars

The Microsoft Foundation Class Library supports dockable toolbars. A dockable toolbar can be attached, or docked, to any side of its parent window, or it can be detached, or floated, in its own mini-frame window. This article explains how to use dockable toolbars in your applications.

If you use the Application Wizard to generate the skeleton of your application, you are asked to choose whether you want dockable toolbars. By default, the Application Wizard generates the code that performs the three actions necessary to place a dockable toolbar in your application:

- Enable docking in a frame window.
- Enable docking for a toolbar.
- Dock the toolbar (to the frame window).

If any of these steps are missing, your application will display a standard toolbar. The last two steps must be performed for each dockable toolbar in your application.

Other topics covered in this article include:

- Floating the toolbar
- Dynamically resizing the toolbar
- Setting wrap positions for a fixed-style toolbar

See the MFC General sample DOCKTOOL for examples.

Enabling Docking in a Frame Window

To dock toolbars to a frame window, the frame window (or destination) must be enabled to allow docking. This is done using the CFrameWnd::EnableDocking function, which takes one **DWORD** parameter that is a set of style bits indicating which side of the frame window accepts docking. If a toolbar is about to be docked and there are multiple sides that it could be docked to, the sides indicated in the parameter passed to **EnableDocking** are used in the following order: top, bottom, left, right. If you want to be able to dock control bars anywhere, pass **CBRS_ALIGN_ANY** to **EnableDocking**.

Enabling Docking for a Toolbar

After you have prepared the destination for docking, you must prepare the toolbar (or source) in a similar fashion. Call CControlBar::EnableDocking for each toolbar you want to dock, specifying the destination sides to which the toolbar should dock. If none of the sides specified in the call to **CControlBar::EnableDocking** match the sides enabled for docking in the frame window, the toolbar cannot dock — it will float. Once it has been floated, it remains a floating toolbar, unable to dock to the frame window.

If the effect you want is a permanently floating toolbar, call **EnableDocking** with a parameter of 0. Then call CFrameWnd::FloatControlBar. The toolbar remains floating, permanently unable to dock anywhere.

Docking the Toolbar

The framework calls CFrameWnd::DockControlBar when the user attempts to drop the toolbar on a side of the frame window that allows docking.

In addition, you can call this function at any time to dock control bars to the frame window. This is normally done during initialization. More than one toolbar can be docked to a particular side of the frame window.

Floating the Toolbar

Detaching a dockable toolbar from the frame window is called floating the toolbar. Call CFrameWnd::FloatControlBar to do this. Specify the toolbar to be floated, the point where it should be placed, and an alignment style that determines whether the floating toolbar is horizontal or vertical.

The framework calls this function when a user drags a toolbar off its docked location and drops it in a location where docking is not enabled. This can be anywhere inside or outside the frame window. As with **DockControlBar**, you can also call this function during initialization.

The MFC implementation of dockable toolbars does not provide some of the extended features found in some applications that support dockable toolbars. Features such as customizable toolbars are not provided.

Dynamically Resizing the Toolbar

As of Visual C++ version 4.0, you can make it possible for users of your application to resize floating toolbars dynamically. Typically, a toolbar has a long, linear shape, displayed horizontally. But you can change the toolbar's orientation and its shape. For example, when the user docks a toolbar against one of the vertical sides of the frame window, the shape changes to a vertical layout. It's also possible to reshape the toolbar into a rectangle with multiple rows of buttons.

You can:

- Specify dynamic sizing as a toolbar characteristic.
- Specify fixed sizing as a toolbar characteristic.

To provide this support, there are two new toolbar styles for use in your calls to the CToolBar::Create member function. They are:

- **CBRS_SIZE_DYNAMIC** Control bar is dynamic.
- **CBRS_SIZE_FIXED** Control bar is fixed.

The size dynamic style lets your user resize the toolbar while it is floating, but not while it is docked. The toolbar "wraps" where needed to change shape as the user drags its edges.

The size fixed style preserves the wrap states of a toolbar, fixing the position of the buttons in each column. Your application's user can't change the shape of the toolbar. The toolbar wraps at designated places, such as the locations of separators between the buttons. It maintains this shape whether the toolbar is docked or floating. The effect is a fixed palette with multiple columns of buttons.

You can also use CToolBar::GetButtonStyle to return a state and style for buttons on your toolbars. A button's style determines how the button appears and how it responds to user input; the state tells whether the button is in a wrapped state.

Setting Wrap Positions for a Fixed-Style Toolbar

For a toolbar with the size fixed style, designate toolbar button indexes at which the toolbar will wrap. The following code shows how to do this in your main frame window's `OnCreate` override:

```
// Get the style of the first button separator
UINT nStyle = m_wndToolBar->GetButtonStyle( 3 );
// Augment the state for wrapping
nStyle |= TBBS_WRAPPED;
m_wndToolBar->SetButtonStyle( 3, nStyle );

// Do the same for other wrap locations ...

// Set the bar style to size fixed
m_wndToolBar->SetBarStyle(m_wndToolBar->GetBarStyle() |
       CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_FIXED);

// Call docking/floating functions as needed ...
```

The MFC General sample DOCKTOOL shows how to use member functions of classes CControlBar and CToolBar to manage dynamic layout of a toolbar. See the file EDITBAR.CPP in DOCKTOOL.

What do you want to know more about?

- Toolbar fundamentals
- Toolbar tool tips
- Using your old toolbars

See Also
Concepts
MFC Toolbar Implementation

Toolbar Tool Tips

Tool tips are the tiny popup windows that present short descriptions of a toolbar button's purpose when you position the mouse over a button for a period of time. When you create an application with the Application Wizard that has a toolbar, tool tip support is provided for you. This article explains both the tool tip support created by the Application Wizard and how to add tool tip support to your application.

This article covers:

- Activating tool tips
- Flyby status bar updates

Activating Tool Tips

To activate tool tips in your application, you must do two things:

- Add the **CBRS_TOOLTIPS** style to the other styles (such as **WS_CHILD**, **WS_VISIBLE**, and other **CBRS_** styles) passed as the `dwStyle` parameter to the CToolBar::Create function or in SetBarStyle.
- As described in the procedure below, append the toolbar tip text, separated by a newline character ('\n'), to the string resource containing the command-line prompt for the toolbar command. The string resource shares the ID of the toolbar button.

To add the tool tip text

1. While you are editing the toolbar in the toolbar editor, open the **Toolbar Button Properties** window for a given button.
2. In the **Prompt** box, specify the text you want to appear in the tool tip for that button.

Note
Setting the text as a button property in the toolbar editor replaces the former procedure, in which you had to open and edit t he string resource.

If a control bar with tool tips enabled has child controls placed on it, the control bar will display a tool tip for every child control on the control bar as long as it meets the following criteria:

- The ID of the control is not – 1.
- The string-table entry with the same ID as the child control in the resource file has a tool tip string.

Flyby Status Bar Updates

A feature related to tool tips is "flyby" status bar updating. By default, the message on the status bar describes only a particular toolbar button when the button is activated. By including **CBRS_FLYBY** in your list of styles passed to **CToolBar::Create**, you can have these messages updated when the mouse cursor passes over the toolbar without actually activating the button.

What do you want to know more about?

- MFC Toolbar Implementation (overview information on toolbars)
- Docking and floating toolbars
- The CToolBar and CToolBarCtrl classes
- Working with the toolbar control
- Using your old toolbars

See Also

Concepts

MFC Toolbar Implementation

Working with the Toolbar Control

This article explains how you can access the CToolBarCtrl object underlying a CToolBar for greater control over your toolbars. This is an advanced topic.

Procedures

To access the toolbar common control underlying your CToolBar object

- Call CToolBar::GetToolBarCtrl.

GetToolBarCtrl returns a reference to a CToolBarCtrl object. You can use the reference to call member functions of the toolbar control class.

Caution
While calling **CToolBarCtrl Get** functions is safe, use caution if you call the **Set** functions. This is an advanced topic. Normally you shouldn't need to access the underlying toolbar control.

What do you want to know more about?

- Controls (Windows common controls)
- Toolbar fundamentals
- Docking and floating toolbars
- Dynamically resizing the toolbar
- Toolbar tool tips
- Flyby status bar updates
- Handling tool tip notifications
- The CToolBar and CToolBarCtrl classes
- Handling customization notifications
- Multiple toolbars
- Using your old toolbars
- Control bars

For general information about using Windows common controls, see Common Controls.

See Also

Concepts

MFC Toolbar Implementation

Using Your Old Toolbars

If you have used previous versions of Visual C++ to create customized toolbars, the new implementation of class CToolBar could cause you problems. So that you don't have to give up your old toolbars to use the new functionality, the old implementation is still supported.

The DOCKTOOL sample does not use the old-style toolbars, only the new-style toolbars.

You can't edit old-style toolbars with the toolbar resource editor.

What do you want to know more about?

- Toolbar fundamentals
- Docking and floating toolbars
- Toolbar tool tips
- Working with the toolbar control

See Also
Concepts
MFC Toolbar Implementation

Windows

This family of articles covers window objects in the MFC framework. All MFC windows derive from class CWnd, including frame windows, views, dialog boxes, and controls.

The first group of articles describes window objects in general. Refer to this group for general information about C++ window objects, how they encapsulate an HWND, and how you use them when creating your own windows, such as child windows.

The second group of articles describes frame windows—windows that put a frame around content — in particular. Refer to this group for information about how the MFC framework manages frame windows and the contents that they frame, including control bars and views.

What do you want to know more about?

Topics on Window Objects in General

- Window objects
- Relationship between a C++ window objects and HWND handles
- Derived Window classes
- Creating window objects
- Destroying Window Objects
- Registering window "classes"
- Working with window objects
- Device contexts: objects that make Windows drawing device-independent
- Graphic objects: pens, brushes, fonts, bitmaps, palettes, regions

Frame Window Topics

- Frame windows: window objects that provide frames
- Frame windows and views
- Frame-window classes
- Frame-window styles
- Changing the styles of a window created by MFC
- What frame windows do
- Using frame windows
- Managing MD/Child windows (the MDICLIENT window)
- Managing menus, control bars, and accelerators
- CFrameWnd
- CMDIFrameWnd
- CMDIChildWnd
- Using Views
- Multiple Document Types, Views, and Frame Windows (Splitter windows)
- Messages (maps and handler functions)

Create and Destroy Windows

- General Window Creation Sequence
- Destroy window objects

- Create document frame windows
- Destroy frame windows

Create Splitter Windows

- Create splitter windows

Manage Child Windows and the Current View

- Manage MDI child windows
- Manage the current view
- Manage menus, control bars, and accelerators

Work with Device Contexts and Window Styles

- Use pens and other graphic objects in a device context
- Change the styles of a window created by MFC

See Also

Concepts

Dialog Boxes
Toolbars
Status Bars
Dialog Bars

Other Resources

General Concepts
User Interface

Window Objects

MFC supplies class CWnd to encapsulate the **HWND** handle of a window. The **CWnd** object is a C++ window object, distinct from the **HWND** that represents a Windows window but containing it. Use **CWnd** to derive your own child window classes, or use one of the many MFC classes derived from **CWnd**. Class **CWnd** is the base class for all windows, including frame windows, dialog boxes, child windows, controls, and control bars such as toolbars. A good understanding of the relationship between a C++ window object and an HWND is crucial for effective programming with MFC.

MFC provides some default functionality and management of windows, but you can derive your own class from **CWnd** and use its member functions to customize the provided functionality. You can create child windows by constructing a **CWnd** object and calling its Create member function, then customize the child windows using **CWnd** member functions. You can embed objects derived from CView, such as form views or tree views, in a frame window. And you can support multiple views of your documents via splitter panes, supplied by class CSplitterWnd.

Each object derived from class **CWnd** contains a message map, through which you can map Windows messages or command IDs to your own handlers.

The general literature on programming for Windows is a good resource for learning how to use the **CWnd** member functions, which encapsulate the **HWND** APIs.

Functions for Operating On a CWnd

CWnd and its derived window classes provide constructors, destructors, and member functions to initialize the object, create the underlying Windows structures, and access the encapsulated **HWND**. **CWnd** also provides member functions that encapsulate Windows APIs for sending messages, accessing the window's state, converting coordinates, updating, scrolling, accessing the Clipboard, and many other tasks. Most Windows window-management APIs that take an **HWND** argument are encapsulated as member functions of **CWnd**. The names of the functions and their parameters are preserved in the **CWnd** member function. For details about the Windows APIs encapsulated by **CWnd**, see class CWnd.

CWnd and Windows Messages

One of the primary purposes of **CWnd** is to provide an interface for handling Windows messages, such as **WM_PAINT** or **WM_MOUSEMOVE**. Many of the member functions of **CWnd** are handlers for standard messages — those beginning with the identifier **afx_msg** and the prefix "On," such as **OnPaint** and **OnMouseMove**. Message Handling and Mapping covers messages and message handling in detail. The information there applies equally to the framework's windows and those that you create yourself for special purposes.

What do you want to know more about?

- The relationship between a C++ window object and an HWND
- Derived window classes
- Creating windows
- Destroying window objects
- Detaching a CWnd from Its HWND
- Working with window objects
- Device contexts: objects that make Windows drawing device independent
- Graphic objects: pens, brushes, fonts, bitmaps, palettes, regions

See Also

Concepts

Windows

Relationship Between a C++ Window Object and an HWND

The window *object* is an object of the C++ **CWnd** class (or a derived class) that your program creates directly. It comes and goes in response to your program's constructor and destructor calls. The Windows *window*, on the other hand, is an opaque handle to an internal Windows data structure that corresponds to a window and consumes system resources when present. A Windows window is identified by a "window handle" (**HWND**) and is created after the **CWnd** object is created by a call to the **Create** member function of class **CWnd**. The window may be destroyed either by a program call or by a user's action. The window handle is stored in the window object's **m_hWnd** member variable. The following figure shows the relationship between the C++ window object and the Windows window. Creating windows is discussed in Creating Windows. Destroying windows is discussed in Destroying Window Objects.

Window Object and Windows Window

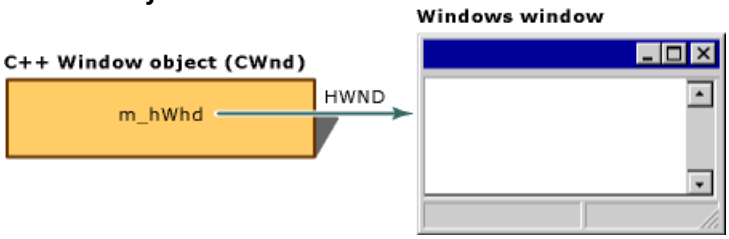

See Also

Concepts

Window Objects

Derived Window Classes

You can create windows directly from CWnd, or derive new window classes from **CWnd**. This is how you typically create your own custom windows. However, most windows used in a framework program are instead created from one of the **CWnd**-derived frame-window classes supplied by MFC.

Frame Window Classes

CFrameWnd

Used for SDI frame windows that frame a single document and its view. The frame window is both the main frame window for the application and the frame window for the current document.

CMDIFrameWnd

Used as the main frame window for MDI applications. The main frame window is a container for all MDI document windows and shares its menu bar with them. An MDI frame window is a top-level window that appears on the desktop.

CMDIChildWnd

Used for individual documents opened in an MDI main frame window. Each document and its view are framed by an MDI child frame window contained by the MDI main frame window. An MDI child window looks much like a typical frame window but is contained inside an MDI frame window instead of sitting on the desktop. However, the MDI child window lacks a menu bar of its own and must share the menu bar of the MDI frame window that contains it.

For more information, see Frame Windows.

Other Window Classes Derived from CWnd

In addition to frame windows, several other major categories of windows are derived from **CWnd**:

Views

Views are created using the **CWnd**-derived class CView (or one of its derived classes). A view is attached to a document and acts as an intermediary between the document and the user. A view is a child window (not an MDI child) that typically fills the client area of an SDI frame window or an MDI child frame window (or that portion of the client area not covered by a toolbar and/or a status bar).

Dialog Boxes

Dialog boxes are created using the **CWnd**-derived class CDialog.

Forms

Form views based on dialog-template resources, such as dialog boxes, are created using classes CFormView, CRecordView, or CDaoRecordView.

Controls

Controls such as buttons, list boxes, and combo boxes are created using other classes derived from **CWnd**. See Control Topics.

Control Bars

Child windows that contain controls. Examples include toolbars and status bars. See Control Bars.

Window Class Hierarchy

Refer to the MFC hierarchy chart in the *MFC Reference*. Views are explained in Document/View Architecture. Dialog boxes are explained in Dialog Boxes.

Creating Your Own Special-Purpose Window Classes

In addition to the window classes provided by the class library, you may need special-purpose child windows. To create such a window, create your own CWnd-derived class and make it a child window of a frame or view. Bear in mind that the framework manages the extent of the client area of a document frame window. Most of the client area is managed by a view, but other windows, such as control bars or your own custom windows, may share the space with the view. You may need to interact with the mechanisms in classes CView and CControlBar for positioning child windows in a frame window's client area.

Creating Windows discusses creation of window objects and the Windows windows they manage.

See Also

Concepts

Window Objects

Creating Windows

The framework automatically creates most of the windows you need in a framework program. Document/View Creation shows how the framework creates the frame windows associated with documents and views. But for special purposes you can create your own windows — including your own child windows of frame windows or views — in addition to the windows supplied by the framework.

What do you want to know more about?

- Registering window "classes" (as opposed to C++ window objects)
- General window creation sequence
- Destroying window objects
- Working with window objects

See Also
Concepts
Window Objects

Registering Window Classes

Window "classes" in traditional programming for Windows define the characteristics of a "class" (not a C++ class) from which any number of windows can be created. This kind of class is a template or model for creating windows.

Window Class Registration in Traditional Programs for Windows

In a traditional program for Windows, without MFC, you process all messages to a window in its "window procedure" or "**WndProc**." A **WndProc** is associated with a window by means of a "window class registration" process. The main window is registered in the **WinMain** function, but other classes of windows can be registered anywhere in the application. Registration depends on a structure that contains a pointer to the **WndProc** function together with specifications for the cursor, background brush, and so forth. The structure is passed as a parameter, along with the string name of the class, in a prior call to the **RegisterClass** function. Thus, a registration class can be shared by multiple windows.

Window Class Registration in MFC Programs

In contrast, most window class registration activity is done automatically in an MFC framework program. If you are using MFC, you typically derive a C++ window class from an existing library class using the normal C++ syntax for class inheritance. The framework still uses traditional "registration classes," and it provides several standard ones, registered for you when needed. You can register additional registration classes by calling the AfxRegisterWndClass global function and then passing the registered class to the **Create** member function of **CWnd**. As described here, the traditional "registration class" in Windows is not to be confused with a C++ class.

For more information, see Technical Note 1.

See Also
Reference
Creating Windows

General Window Creation Sequence

When you create a window of your own, such as a child window, the framework uses much the same process as that described in Document/View Creation.

All the window classes provided by MFC employ two-stage construction. That is, during an invocation of the C++ **new** operator, the constructor allocates and initializes a C++ object but does not create a corresponding Windows window. That is done afterward by calling the Create member function of the window object.

The **Create** member function makes the Windows window and stores its **HWND** in the C++ object's public data member m_hWnd. **Create** gives complete flexibility over the creation parameters. Before calling **Create**, you may want to register a window class with the global function AfxRegisterWndClass in order to set the icon and class styles for the frame.

For frame windows, you can use the LoadFrame member function instead of **Create**. **LoadFrame** makes the Windows window using fewer parameters. It gets many default values from resources, including the frame's caption, icon, accelerator table, and menu.

Note
Your icon, accelerator table, and menu resources must have a common resource ID, such as **IDR_MAINFRAME**, for them to be loaded by LoadFrame.

What do you want to know more about?

- Window objects
- Registering window "classes"
- Destroying window objects
- Creating document frame windows

See Also

Reference

Creating Windows

Destroying Window Objects

Care must be taken with your own child windows to destroy the C++ window object when the user is finished with the window. If these objects are not destroyed, your application will not recover their memory. Fortunately, the framework manages window destruction as well as creation for frame windows, views, and dialog boxes. If you create additional windows, you are responsible for destroying them.

What do you want to know more about?

- Window destruction sequence
- Allocating and deallocating window memory
- Detaching a CWnd from its HWND
- General Window Creation Sequence
- Destroying frame windows

See Also
Concepts
Window Objects

Window Destruction Sequence

In the MFC framework, when the user closes the frame window, the window's default OnClose handler calls DestroyWindow. The last member function called when the Windows window is destroyed is OnNcDestroy, which does some cleanup, calls the Default member function to perform Windows cleanup, and lastly calls the virtual member function PostNcDestroy. The CFrameWnd implementation of **PostNcDestroy** deletes the C++ window object.

What do you want to know more about?

- Allocating and deallocating window memory
- Detaching a CWnd from its HWND

See Also

Reference

Destroying Window Objects

Allocating and Deallocating Window Memory

Do not use the C++ **delete** operator to destroy a frame window or view. Instead, call the **CWnd** member function **DestroyWindow**. Frame windows, therefore, should be allocated on the heap with operator **new**. Be careful when allocating frame windows on the stack frame or globally. Other windows should be allocated on the stack frame whenever possible.

What do you want to know more about?

- Creating windows
- Window destruction sequence
- Detaching a CWnd from its HWND

See Also
Reference
Destroying Window Objects

Detaching a CWnd from Its HWND

If you need to circumvent the object-**HWND** relationship, MFC provides another **CWnd** member function, Detach, which disconnects the C++ window object from the Windows window. This prevents the destructor from destroying the Windows window when the object is destroyed.

What do you want to know more about?

- Creating windows
- Window destruction sequence
- Allocating and deallocating window memory

See Also
Concepts
Window Objects

Working with Window Objects

Working with windows calls for two kinds of activity:

- Handling Windows messages
- Drawing in the window

To handle Windows messages in any window, including your own child windows, see Mapping Messages to Functions to map the messages to your C++ window class. Then write message-handler member functions in your class.

Most drawing in a framework application occurs in the view, whose OnDraw member function is called whenever the window's contents must be drawn. If your window is a child of the view, you might delegate some of the view's drawing to your child window by having `OnDraw` call one of your window's member functions.

In any case, you will need a device context for drawing. You can use the stock pen, brush, and other graphic objects contained in the device context associated with your window. Or you can modify these objects to get the drawing effects you need. With your device context set up as you like, call member functions of class CDC (device-context class) to draw lines, shapes, and text; to use colors; and to work with a coordinate system.

What do you want to know more about?

- Message handling and mapping
- Drawing in a view
- Device contexts
- Graphic objects

See Also
Concepts
Window Objects

Device Contexts

A device context is a Windows data structure containing information about the drawing attributes of a device such as a display or a printer. All drawing calls are made through a device-context object, which encapsulates the Windows APIs for drawing lines, shapes, and text. Device contexts allow device-independent drawing in Windows. Device contexts can be used to draw to the screen, to the printer, or to a metafile.

CPaintDC objects encapsulate the common idiom of Windows, calling the **BeginPaint** function, then drawing in the device context, then calling the **EndPaint** function. The **CPaintDC** constructor calls **BeginPaint** for you, and the destructor calls **EndPaint**. The simplified process is to create the **CDC** object, draw, and destroy the **CDC** object. In the framework, much of even this process is automated. In particular, your `OnDraw` function is passed a **CPaintDC** already prepared (via **OnPrepareDC**), and you simply draw into it. It is destroyed by the framework and the underlying device context is released to Windows upon return from the call to your `OnDraw` function.

CClientDC objects encapsulate working with a device context that represents only the client area of a window. The **CClientDC** constructor calls the **GetDC** function, and the destructor calls the **ReleaseDC** function. CWindowDC objects encapsulate a device context that represents the whole window, including its frame.

CMetaFileDC objects encapsulate drawing into a Windows metafile. In contrast to the **CPaintDC** passed to `OnDraw`, you must in this case call OnPrepareDC yourself.

Mouse Drawing

Most drawing in a framework program — and thus most device-context work — is done in the view's `OnDraw` member function. However, you can still use device-context objects for other purposes. For example, to provide tracking feedback for mouse movement in a view, you need to draw directly into the view without waiting for `OnDraw` to be called.

In such a case, you can use a CClientDC device-context object to draw directly into the view.

What do you want to know more about?

- Device contexts (definition)
- Drawing in a View
- Interpreting User Input Through a View
- Lines and curves
- Filled shapes
- Fonts and text
- Colors
- Coordinate spaces and transformations

See Also

Concepts

Window Objects

Graphic Objects

Windows provides a variety of drawing tools to use in device contexts. It provides pens to draw lines, brushes to fill interiors, and fonts to draw text. MFC provides graphic-object classes equivalent to the drawing tools in Windows. The table below shows the available classes and the equivalent Windows graphics device interface (GDI) handle types.

Note

GDI+ is included with Windows XP and is available as a redistributable for Windows NT 4.0 SP6, Windows 2000, Windows 98, and Windows Me. To download the latest redistributable, see http://www.microsoft.com/msdownload/platformsdk/sdkupdate/psdkredist.htm. For more information, see the GDI+ SDK documentation in MSDN: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/gdicpp/GDIPlus/GDIPlus.asp.

This article explains the use of these graphic-object classes:

Classes for Windows GDI Objects

Class	Windows handle type
CPen	**HPEN**
CBrush	**HBRUSH**
CFont	**HFONT**
CBitmap	**HBITMAP**
CPalette	**HPALETTE**
CRgn	**HRGN**

Note

The class CImage provides enhanced bitmap support.

Each graphic-object class in the class library has a constructor that allows you to create graphic objects of that class, which you must then initialize with the appropriate create function, such as **CreatePen**.

Each graphic-object class in the class library has a cast operator that will cast an MFC object to the associated Windows handle. The resulting handle is valid until the associated object detaches it. Use the object's **Detach** member function to detach the handle.

The following code casts a **CPen** object to a Windows handle:

```
CPen myPen;
myPen.CreateSolidPen( PS_COSMETIC, 1, RGB(255,255,0) );
HPEN hMyPen = (HPEN) myPen;
```

To create a graphic object in a device context

1. Define a graphic object on the stack frame. Initialize the object with the type-specific create function, such as **CreatePen**. Alternatively, initialize the object in the constructor. See the discussion of one-stage and two-stage creation, which provides example code.
2. Select the object into the current device context, saving the old graphic object that was selected before.
3. When done with the current graphic object, select the old graphic object back into the device context to restore its state.
4. Allow the frame-allocated graphic object to be deleted automatically when the scope is exited.

Note

If you will be using a graphic object repeatedly, you can allocate it once and select it into a device context each time it is needed. Be sure to delete such an object when you no longer need it.

What do you want to know more about?

- One-stage and two-stage construction of graphic objects
- Example of constructing a pen in one and two stages
- Selecting a Graphic Object into a Device Context
- Device contexts
- CImage limitations with earlier operating systems

See Also

Concepts

Window Objects

One-Stage and Two-Stage Construction of Objects

You have a choice between two techniques for creating graphic objects, such as pens and brushes:

- *One-stage construction*: Construct and initialize the object in one stage, all with the constructor.
- *Two-stage construction*: Construct and initialize the object in two separate stages. The constructor creates the object and an initialization function initializes it.

Two-stage construction is always safer. In one-stage construction, the constructor could throw an exception if you provide incorrect arguments or memory allocation fails. That problem is avoided by two-stage construction, although you do have to check for failure. In either case, destroying the object is the same process.

Note
These techniques apply to creating any objects, not just graphic objects.

Example of Both Construction Techniques

The following brief example shows both methods of constructing a pen object:

```
void CMyView::OnDraw( CDC* pDC )
{
// One-stage
    CPen myPen1( PS_DOT, 5, RGB(0,0,0) );

// Two-stage: first construct the pen
    CPen myPen2;
    // Then initialize it
    if( myPen2.CreatePen( PS_DOT, 5, RGB(0,0,0) ) )
        // Use the pen
}
```

What do you want to know more about?

- Graphic objects
- Selecting a graphic object into a device context
- Device contexts
- Drawing in a View

See Also

Concepts

Graphic Objects

Selecting a Graphic Object into a Device Context

This topic applies to using graphic objects in a window's device context. After you create a drawing object, you must select it into the device context in place of the default object stored there:

```
void CMyView::OnDraw( CDC* pDC )
{
    CPen penBlack;  // Construct it, then initialize
    if( newPen.CreatePen( PS_SOLID, 2, RGB(0,0,0) ) )
    {
        // Select it into the device context
        // Save the old pen at the same time
        CPen* pOldPen = pDC->SelectObject( &penBlack );

        // Draw with the pen
        pDC->MoveTo(...);
        pDC->LineTo(...);

        // Restore the old pen to the device context
        pDC->SelectObject( pOldPen );
    }
    else
    {
        // Alert the user that resources are low
    }
}
```

Lifetime of Graphic Objects

The graphic object returned by SelectObject is "temporary." That is, it will be deleted by the OnIdle member function of class **CWinApp** the next time the program gets idle time. As long as you use the object returned by **SelectObject** in a single function without returning control to the main message loop, you will have no problem.

What do you want to know more about?

- Graphic objects
- One-stage and two-stage construction of graphic objects
- Device contexts
- Drawing in a View

See Also
Concepts
Graphic Objects

CImage Limitations with Earlier Operating Systems

Many **CImage** functions work only with newer versions of Windows: Windows 95/98 or Windows NT 4.0, or Windows 2000. This article describes the version limitations of certain methods.

CImage::PlgBlt and CImage::MaskBlt work with only Windows NT 4.0 or later. They will not work on applications running on Windows 95/98 or later.

CImage::AlphaBlend and CImage::TransparentBlt work with only Windows 2000 or later and Windows 98 or later because you must link with msimg32.lib to use these methods. (This library is available only to applications running Windows 2000 or later and Windows 98 or later.)

You can include **AlphaBlend** and **TransparentBlt** in an application that runs on Windows 95 or Windows NT 4.0 only if these methods never get called. If your application includes these methods, and it must run on earlier operating systems, you must use the linker's /delayload to delay the loading of msimg32.lib. As long as your application does not call one of these methods while running under Windows NT 4.0 or Windows 95, it will not attempt to load msimg32.lib. You can check the whether transparency support is available using the **CImage::IsTransparencySupported** method.

Example

```
if (CImage::IsTransparencySupported())
   // Safe to call CImage::AlphaBlend and CImage::TransparentBlt
else
   // Transparency not supported.  Fall back to something else.
```

To compile an application that calls these methods, insert a #define _WIN32_WINNT statement before #including any system headers, indicating that the version of Windows is equal to or greater than 5.0:

```
#define _WIN32_WINNT 0x0500
```

If your application does not need to run on an operating system older than Windows 2000 or Windows 98, you can link directly to msimg32.lib without using **/delayload**.

CImage::Draw behaves differently when used with Windows 2000 and Windows 98 than it does with Windows NT 4.0 or Windows 95.

If you compile your application with _WIN32_WINNT set to a value less than 0x0500, **Draw** will work, but it will not handle transparency automatically on systems running Windows 2000 and Windows 98 and later.

If you compile your application with _WIN32_WINNT set to 0x0500 or greater, **Draw** will handle transparency automatically on systems running Windows 2000 or Windows 98 and later. It will also work, but without transparency support, with Windows NT 4.0 and Windows 95; however, you must use **/delayload** to delay the loading of msimg32.LIB, as described above for **AlphaBlend** and **TransparentBlt**.

See Also

Reference

CImage Class

Frame Windows

When an application runs under Microsoft Windows, the user interacts with documents displayed in frame windows. A document frame window has two major components: the frame and the contents that it frames. A document frame window can be a single document interface (SDI) frame window or a multiple document interface (MDI) child window. Windows manages most of the user's interaction with the frame window: moving and resizing the window, closing it, and minimizing and maximizing it. You manage the contents inside the frame.

Frame Windows and Views

The MFC framework uses frame windows to contain views. The two components — frame and contents — are represented and managed by two different classes in MFC. A frame-window class manages the frame, and a view class manages the contents. The view window is a child of the frame window. Drawing and other user interaction with the document take place in the view's client area, not the frame window's client area. The frame window provides a visible frame around a view, complete with a caption bar and standard window controls such as a control menu, buttons to minimize and maximize the window, and controls for resizing the window. The "contents" consist of the window's client area, which is fully occupied by a child window — the view. The following figure shows the relationship between a frame window and a view.

Frame Window and View

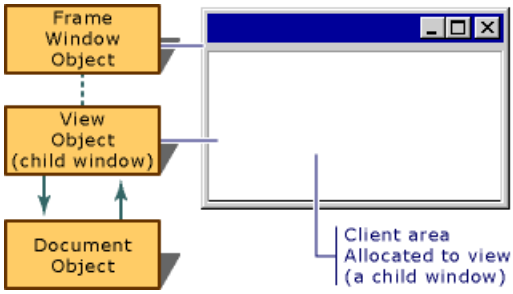

Frame Windows and Splitter Windows

Another common arrangement is for the frame window to frame multiple views, usually using a splitter window. In a splitter window, the frame window's client area is occupied by a splitter window, which in turn has multiple child windows, called panes, which are views.

What do you want to know more about?

General Frame Window Topics

- Window objects
- Frame window classes
- The Frame-Window classes created by the Application Wizard
- Frame window styles
- What frame windows do

Topics on Using Frame Windows

- Using frame windows
- Creating document frame windows
- Destroying frame windows
- Managing MDI child windows
- Managing the current view in a frame window that contains more than one view
- Managing menus, control bars, and accelerators (other objects that share the frame window's space)

Topics on Special Frame Window Capabilities

- Dragging and dropping files from Windows Explorer or File Manager into a frame window
- Responding to dynamic data exchange (DDE)

- Semimodal states: Context-sensitive Windows Help (Orchestrating other window actions)
- Semimodal states: printing and print preview (Orchestrating other window actions)

Topics on Other Kinds of Windows

- Using Views
- Dialog boxes
- Controls

See Also

Concepts

Windows

Frame-Window Classes

Each application has one "main frame window," a desktop window that usually has the application name in its caption. Each document usually has one "document frame window." A document frame window contains at least one view, which presents the document's data.

Frame Windows in SDI and MDI Applications

For an SDI application, there is one frame window derived from class CFrameWnd. This window is both the main frame window and the document frame window. For an MDI application, the main frame window is derived from class CMDIFrameWnd, and the document frame windows, which are MDI child windows, are derived from class CMDIChildWnd.

Use the Frame-Window Class, or Derive from It?

These classes provide most of the frame-window functionality you need for your applications. Under normal circumstances, the default behavior and appearance they provide will suit your needs. If you need additional functionality, derive from these classes.

What do you want to know more about?

- Frame-window classes created by the Application Wizard
- Frame-window styles
- Changing the styles of a window created by MFC

See Also

Concepts

Frame Windows

The Frame-Window Classes Created by the Application Wizard

When you use the Application Wizard to create a skeleton application, in addition to application, document, and view classes, the Application Wizard creates a derived frame-window class for your application's main frame window. The class is called `CMainFrame` by default, and the files that contain it are named MAINFRM.H and MAINFRM.CPP.

If your application is SDI, your `CMainFrame` class is derived from class CFrameWnd.

If your application is MDI, `CMainFrame` is derived from class CMDIFrameWnd. In this case `CMainFrame` implements the main frame, which holds the menu, toolbar, and status bars. The Application Wizard does not derive a new document frame-window class for you. Instead, it uses the default implementation in CMDIChildWnd Class. The MFC framework creates a child window to contain each view (which can be of type **CScrollView**, **CEditView**, **CTreeView**, **CListView**, and so on) that the application requires. If you need to customize your document frame window, you can create a new document frame-window class (see Adding a Class).

If you choose to support a toolbar, the class also has member variables of type CToolBar and CStatusBar and an `OnCreate` message-handler function to initialize the two control bars.

These frame-window classes work as created, but to enhance their functionality, you must add member variables and member functions. You may also want to have your window classes handle other Windows messages. For more information, see Changing the Styles of a Window Created by MFC.

See Also

Reference

MFC Program or Control Source and Header Files

Concepts

Frame-Window Classes

Frame-Window Styles (C++)

The frame windows you get with the framework are suitable for most programs, but you can gain additional flexibility by using the advanced functions PreCreateWindow and the MFC global function AfxRegisterWndClass. **PreCreateWindow** is a member function of **CWnd**.

If you apply the **WS_HSCROLL** and **WS_VSCROLL** styles to the main frame window, they are instead applied to the **MDICLIENT** window so users can scroll the **MDICLIENT** area.

If the window's **FWS_ADDTOTITLE** style bit is set (which it is by default), the view tells the frame window what title to display in the window's title bar based on the view's document name.

What do you want to know more about?

- Managing MDI child windows (MDICLIENT), the window within an MDI frame that contains the MDI child windows
- Changing the styles of a window created by MFC
- Window styles

See Also

Concepts

Frame Windows

Changing the Styles of a Window Created by MFC

In its version of the **WinMain** function, MFC registers several standard window classes for you. Because you don't normally edit MFC's **WinMain**, that function gives you no opportunity to change the MFC default window styles. This article explains how you can change the styles of such a preregistered window class in an existing application.

Changing Styles in a New MFC Application

If you're using Visual C++ 2.0 or later, you can change the default window styles in the Application Wizard when you create your application. In the Application Wizard's User Interface Features page, you can change styles for your main frame window and MDI child windows. For either window type, you can specify its frame thickness (thick or thin) and any of the following:

- Whether the window has Minimize or Maximize controls.
- Whether the window appears initially minimized, maximized, or neither.

For main frame windows, you can also specify whether the window has a System Menu. For MDI child windows, you can specify whether the window supports splitter panes.

Changing Styles in an Existing Application

If you're changing window attributes in an existing application, follow the instructions in the rest of this article instead.

To change the default window attributes used by a framework application created with the Application Wizard, override the window's PreCreateWindow virtual member function. **PreCreateWindow** allows an application to access the creation process normally managed internally by the CDocTemplate class. The framework calls **PreCreateWindow** just prior to creating the window. By modifying the CREATESTRUCT structure passed to **PreCreateWindow**, your application can change the attributes used to create the window. For example, to ensure that a window does not use a caption, use the following bitwise operation:

```
// cs has been declared as CREATESTRUCT& cs;
cs.style &= ~WS_CAPTION;
```

The CTRLBARS sample application demonstrates this technique for changing window attributes. Depending on what your application changes in **PreCreateWindow**, it may be necessary to call the base class implementation of the function.

The following discussion covers the SDI case and the MDI case.

The SDI Case

In a single document interface (SDI) application, the default window style in the framework is a combination of the **WS_OVERLAPPEDWINDOW** and **FWS_ADDTOTITLE** styles. **FWS_ADDTOTITLE** is an MFC-specific style that instructs the framework to add the document title to the window's caption. To change the window attributes in an SDI application, override the **PreCreateWindow** function in your class derived from **CFrameWnd** (which the Application Wizard names `CMainFrame`). For example:

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    // Create a window without min/max buttons or sizable border
    cs.style = WS_OVERLAPPED | WS_SYSMENU | WS_BORDER;

    // Size the window to 1/3 screen size and center it
    cs.cy = ::GetSystemMetrics(SM_CYSCREEN) / 3;
    cs.cx = ::GetSystemMetrics(SM_CXSCREEN) / 3;
    cs.y = ((cs.cy * 3) - cs.cy) / 2;
    cs.x = ((cs.cx * 3) - cs.cx) / 2;

    // Call the base-class version
    return CFrameWnd::PreCreateWindow(cs);
}
```

This code creates a main frame window without Minimize and Maximize buttons and without a sizable border. The window is initially centered on the screen.

The MDI Case

A little more work is required to change the window style of a child window in a multiple document interface (MDI) application. By default, an MDI application created with the Application Wizard uses the default CMDIChildWnd class defined in MFC. To change the window style of an MDI child window, you must derive a new class from **CMDIChildWnd** and replace all references to **CMDIChildWnd** in your project with references to the new class. Most likely, the only reference to **CMDIChildWnd** in the application is located in your application's `InitInstance` member function.

The default window style used in an MDI application is a combination of the **WS_CHILD**, **WS_OVERLAPPEDWINDOW**, and **FWS_ADDTOTITLE** styles. To change the window attributes of an MDI application's child windows, override the PreCreateWindow function in your class derived from **CMDIChildWnd**. For example:

```
BOOL CMyChildWnd::PreCreateWindow(CREATESTRUCT& cs)
{
    // Create a child window without the maximize button
    cs.style &= ~WS_MAXIMIZEBOX;

    // Call the base-class version
    return CMDIChildWnd::PreCreateWindow(cs);
}
```

This code creates MDI child windows without a Maximize button.

What do you want to know more about?

- Windows styles
- Frame-window styles
- Window styles

See Also
Concepts
Frame-Window Styles (C++)

What Frame Windows Do

Besides simply framing a view, frame windows are responsible for numerous tasks involved in coordinating the frame with its view and with the application. CMDIFrameWnd and CMDIChildWnd inherit from CFrameWnd, so they have **CFrameWnd** capabilities as well as new capabilities that they add. Examples of child windows include views, controls such as buttons and list boxes, and control bars, including toolbars, status bars, and dialog bars.

The frame window is responsible for managing the layout of its child windows. In the MFC framework, a frame window positions any control bars, views, and other child windows inside its client area.

The frame window also forwards commands to its views and can respond to notification messages from control windows.

What do you want to know more about?

- Control bars (how they fit into the frame window)
- Managing menus, control bars, and accelerators (how they fit into the frame window)
- Command Routing (from the frame window to its view and other command targets)
- Document /View Architecture
- Control bars
- Controls

See Also

Concepts

Frame Windows

Using Frame Windows

The MFC framework creates document frame windows — and their views and documents — as part of its implementation of the New and Open commands on the File menu. Because the framework does most of the frame-window work for you, you play only a small role in creating, using, and destroying those windows. You can, however, explicitly create your own frame windows and child windows for special purposes.

What do you want to know more about?

- Creating document frame windows
- When to Initialize CWnd Objects
- Destroying frame windows
- Managing MDI child windows
- Managing the current view
- Managing menus, control bars, and accelerators
- Dragging and dropping files in a frame window
- Responding to dynamic data exchange (DDE)
- Orchestrating other window actions
- Managing context-sensitive help
- The frame window's role in printing and print preview

See Also
Concepts
Frame Windows

Creating Document Frame Windows

Document/View Creation shows how the CDocTemplate object orchestrates creating the frame window, document, and view and connecting them all together. Three CRuntimeClass arguments to the **CDocTemplate** constructor specify the frame window, document, and view classes that the document template creates dynamically in response to user commands such as the New command on the File menu or the New Window command on an MDI Window menu. The document template stores this information for later use when it creates a frame window for a view and document.

For the RUNTIME_CLASS mechanism to work correctly, your derived frame-window classes must be declared with the DECLARE_DYNCREATE macro. This is because the framework needs to create document frame windows using the dynamic construction mechanism of class **CObject**.

When the user chooses a command that creates a document, the framework calls upon the document template to create the document object, its view, and the frame window that will display the view. When it creates the document frame window, the document template creates an object of the appropriate class — a class derived from CFrameWnd for an SDI application or from CMDIChildWnd for an MDI application. The framework then calls the frame-window object's LoadFrame member function to get creation information from resources and to create the Windows window. The framework attaches the window handle to the frame-window object. Then it creates the view as a child window of the document frame window.

Use caution in deciding when to initialize your **CWnd**-derived object.

What do you want to know more about?

- Deriving a Class from CObject (its dynamic creation mechanism)
- Document/View Creation (templates and frame window creation)
- Destroying frame windows

See Also
Reference
Using Frame Windows

When to Initialize CWnd Objects

You cannot create your own child windows or call any Windows API functions in the constructor of a **CWnd**-derived object. This is because the **HWND** for the **CWnd** object has not been created yet. Most Windows-specific initialization, such as adding child windows, must be done in an OnCreate message handler.

What do you want to know more about?

- Creating document frame windows
- Document/view creation

See Also

Reference

Using Frame Windows

Destroying Frame Windows

The MFC framework manages window destruction as well as creation for those windows associated with framework documents and views. If you create additional windows, you are responsible for destroying them.

In the framework, when the user closes the frame window, the window's default OnClose handler calls DestroyWindow. The last member function called when the Windows window is destroyed is OnNcDestroy, which does some cleanup, calls the Default member function to perform Windows cleanup, and lastly calls the virtual member function PostNcDestroy. The CFrameWnd implementation of **PostNcDestroy** deletes the C++ window object. You should never use the C++ **delete** operator on a frame window. Use **DestroyWindow** instead.

When the main window closes, the application closes. If there are modified unsaved documents, the framework displays a message box to ask if the documents should be saved and ensures that the appropriate documents are saved if necessary.

What do you want to know more about?

- Creating document frame windows

See Also
Reference
Using Frame Windows

Managing MDI Child Windows

MDI main frame windows (one per application) contain a special child window called the **MDICLIENT** window. The **MDICLIENT** window manages the client area of the main frame window, and itself has child windows: the document windows, derived from **CMDIChildWnd**. Because the document windows are frame windows themselves (MDI child windows), they can also have their own children. In all of these cases, the parent window manages its child windows and forwards some commands to them.

In an MDI frame window, the frame window manages the **MDICLIENT** window, repositioning it in conjunction with control bars. The **MDICLIENT** window, in turn, manages all MDI child frame windows. The following figure shows the relationship between an MDI frame window, its **MDICLIENT** window, and its child document frame windows.

MDI Frame Windows and Children

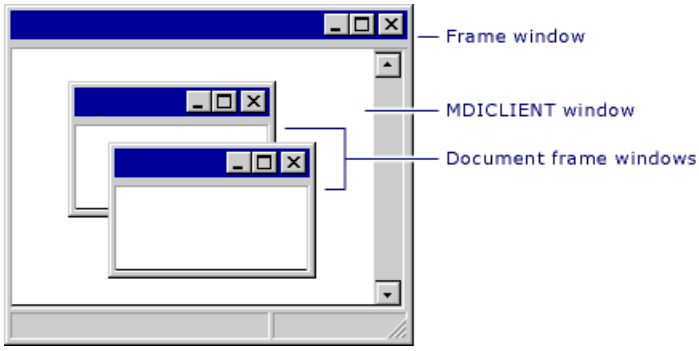

An MDI frame window also works in conjunction with the current MDI child window, if there is one. The MDI frame window delegates command messages to the MDI child before it tries to handle them itself.

What do you want to know more about?

- Creating document frame windows
- Frame-window styles

See Also

Reference

Using Frame Windows

Managing the Current View

As part of the default implementation of frame windows, a frame window keeps track of a currently active view. If the frame window contains more than one view, as for example in a splitter window, the current view is the most recent view in use. The active view is independent of the active window in Windows or the current input focus.

When the active view changes, the framework notifies the current view by calling its OnActivateView member function. You can tell whether the view is being activated or deactivated by examining **OnActivateView**'s *bActivate* parameter. By default, **OnActivateView** sets the focus to the current view on activation. You can override **OnActivateView** to perform any special processing when the view is deactivated or reactivated. For example, you might want to provide special visual cues to distinguish the active view from other, inactive views.

A frame window forwards commands to its current (active) view, as described in Command Routing, as part of the standard command routing.

See Also
Reference
Using Frame Windows

Managing Menus, Control Bars, and Accelerators

The frame window manages updating user-interface objects, including menus, toolbar buttons, the status bar, and accelerators. It also manages sharing the menu bar in MDI applications.

Managing Menus

The frame window participates in updating user-interface items using the **ON_UPDATE_COMMAND_UI** mechanism described in How to Update User-Interface Objects. Buttons on toolbars and other control bars are updated during the idle loop. Menu items in drop-down menus on the menu bar are updated just before the menu drops down.

For MDI applications, the MDI frame window manages the menu bar and caption. An MDI frame window owns one default menu that is used as the menu bar when there are no active MDI child windows. When there are active children, the MDI frame window's menu bar is taken over by the menu for the active MDI child window. If an MDI application supports multiple document types, such as chart and worksheet documents, each type puts its own menus into the menu bar and changes the main frame window's caption.

CMDIFrameWnd provides default implementations for the standard commands on the Window menu that appears for MDI applications. In particular, the New Window command (**ID_WINDOW_NEW**) is implemented to create a new frame window and view on the current document. You need to override these implementations only if you need advanced customization.

Multiple MDI child windows of the same document type share menu resources. If several MDI child windows are created by the same document template, they can all use the same menu resource, saving on system resources in Windows.

Managing the Status Bar

The frame window also positions the status bar within its client area and manages the status bar's indicators. The frame window clears and updates the message area in the status bar as needed and displays prompt strings as the user selects menu items or toolbar buttons, as described in How to Display Command Information in the Status Bar.

Managing Accelerators

Each frame window maintains an optional accelerator table that does keyboard accelerator translation for you automatically. This mechanism makes it easy to define accelerator keys (also called shortcut keys) that invoke menu commands.

See Also

Reference

Using Frame Windows

Dragging and Dropping Files in a Frame Window

The frame window manages a relationship with the Windows Explorer or File Manager.

By adding a few initializing calls in your override of the **CWinApp** member function **InitInstance**, as described in CWinApp: The Application Class, you can have your frame window indirectly open files dragged from the Windows Explorer or File Manager and dropped in the frame window. See File Manager Drag and Drop.

See Also
Reference
Using Frame Windows

Responding to Dynamic Data Exchange (DDE)

The frame window can respond to dynamic data exchange (DDE) requests to open files from the File Manager (if the file extension is registered or associated with the application). See Shell Registration.

See Also

Reference

Using Frame Windows

Orchestrating Other Window Actions

The frame window orchestrates semimodal states such as context-sensitive help and print preview. The framework's role in managing context-sensitive help is described in the article WinHelp: Context-Sensitive Help for Your Programs. For a description of the frame window's role in print preview, see Printing and Print Preview.

See Also
Reference
Using Frame Windows

WinHelp: Context-Sensitive Help for Your Programs

An application can offer user help at just the point in your interface when users will need it.

- Creating an Application with Context-Sensitive Help
- Compiling Your Help File
- Adding Menus and Menu Options
- Adding Help for Dialog Boxes
- Adding Control Help in a Dialog Box
- Adding Context-Sensitive Help to an Existing MFC Application
- Context-Sensitive Help in an MFC Application
- Adding Context-Sensitive Help to an MFC ActiveX Control
- Adding Context-Sensitive Help to an ATL ActiveX Control
- ToolTips
- Status Bar Prompts

You can also view the .hlp file created for your project with the WINHLP32 utility (issue **winhlp32** *file***.hlp** from the command line).

For more information on context-sensitive help, see the following Knowledge Base articles (search for the article number online, in the MSDN Library Visual Studio documentation, with **Search titles only** cleared.):

- Q244232 : HOWTO: Add Context Help Button (? Button) to Title Bar of CPropertySheet
- Q141724 : Context-Sensitive Help in a CDialog Object
- Q167698 : Help95.exe Implements Windows 95 Style Help
- Q149343 : Implement Context-Sensitive Help for Dialog Controls

See Also

Concepts

HTML Help: Context-Sensitive Help for Your Programs

Other Resources

General Concepts

User Interface

Creating an Application with Context-Sensitive Help

Using the MFC Application Wizard, see Creating an MFC Application for more information, create an MFC executable and on the **Advanced Features** tab, enable **Context-sensitive Help**.

When you create an MFC application with context-sensitive Help, you can compile the program and see the kinds of help you get without having to write additional code.

F1 Help

This level of help support enables the user to press the F1 key from an active window, dialog box, or message box, or with a menu item or toolbar button selected, to invoke a Help topic relevant to the selected item.

For menu items, the user can use the arrow keys to highlight a particular menu item, and then press F1. For toolbar buttons, the user can use the mouse to hold down the button and press F1 before letting the button up.

If no user interface object is selected, or if no specific Help topic exists, pressing F1 invokes the main Contents topic for the application Help file.

You can define a key other than F1 for help, but it is common among applications for Windows to use F1.

Context-Sensitive Help Pointer

By either pressing SHIFT+F1 or by clicking the **Help** button on the toolbar, you can activate the context-sensitive Help pointer.

The following figure shows the **Help Mode** button.

While the application is in this help mode, the cursor changes to an arrow beside a question mark. So long as help mode is active, the user can click any window, dialog box, message box, menu item, or toolbar button to summon help specific to the item. This invokes the application's Help file, and ends help mode. Pressing ESC or switching away from the application and back also ends help mode.

Help Topics Dialog Box

The **Help Topics** option of the **Help** menu gives you access to Search, Index and Table of Contents functionality for your application. As you add menus, menu options, dialog boxes and other resources to your application, you will want to update your application's .rtf file and .cnt file (for menus and menu options).

See Also

Concepts

WinHelp: Context-Sensitive Help for Your Programs
HTML Help: Context-Sensitive Help for Your Programs

Compiling Your Help File

Compiling your help file creates a new .hlp file. When you build your project, Visual C++ will also compile the .hlp file if the .rtf file changed. However, you may want to compile your .hlp file outside the development environment.

- Compiling a Help File from Within the Development Environment
- Compiling a Help File from Within Help Compiler Workshop

See Also

Concepts

WinHelp: Context-Sensitive Help for Your Programs

Compiling a Help File from Within the Development Environment

To compile from the development environment, the executable files called by the custom build steps on the .hpj, .cnt and Resource.h files must be in a directory listed in the **Directories** tab of the **Options** dialog box. (Click **Options** on the **Tools** menu.) By default, these files are installed to the \BIN directory of your Visual C++ installation and included in the **Options** dialog box; so, unless you modify either of the default settings, your Help compilation will be a completely transparent part of building your project files.

To compile your Help files from within the development environment

1. Open **Solution Explorer** and expand the **Source Files** folder.
2. Select your help project (.hpj) file.
3. Right click and select **Compile**. This invokes the custom build steps on the .hpj file, which in turn calls the following two programs:
   - MAKEHM.exe, a program that incorporates your context-sensitive topic IDs.
   - The Windows Help Compiler, which builds your .hlp file.

See Also
Reference
Compiling Your Help File

Compiling a Help File from Within Help Compiler Workshop

If your Help file does not behave as expected, you should invoke the HCW.exe and open your project's .hpj file. When you select the **Save and Compile** option in the Help Compiler Workshop, you can see notes and warnings as the help compiler creates your .hlp file. You should fix the warnings; notes are less critical to the success of your .hlp file.

Keeping your free-text search terms current also requires using the Help Compiler Workshop. Invoke the HCW.exe program and search its online help for the string FTS (for full-text search) and read how you can cause your free-text word list to be updated when you compile help.

You may want to become familiar with the online help in the Help Compiler Workshop.

See Also

Reference

Compiling Your Help File

Adding Menus and Menu Options

When you add a menu, add help text to the .rtf file and add an entry to the .cnt file. The .cnt file determines what appears in the **Contents** tab of the **Help Topics** dialog box.

When you add a menu option, add help text and update the list of menu options in the .rtf file.

To add a menu entry to the .cnt file

- The .cnt file controls the appearance of the **Contents** tab in the **Help Topics** dialog box.

  Use a text editor to open the .cnt file in the \Hlp directory of your project. Follow the design of the entries for the **File**, **Edit**, **View**, **Window**, and **Help** options. The value assigned to the menu name is the ID of the help text for this menu option in the .rtf file.

To update the menu list in the .rtf file

1. Use a rich text format editor (or just double-click on the file name in **Solution Explorer**) and open your .rtf file.
2. Modify the list of commands at the top of the file to include the new menu and its ID from the .cnt file and footnote in the .rtf file.

To add help text for the menu

1. Use a rich text format editor and open the .rtf file in the \Hlp directory of your project (you can use new/different .rtf files, just add them to the [FILES] section of your .hpj file).
2. Follow the style of the AxfCore.rtf file and place a help section for the menu towards the top of the file. For your convenience, you can copy the help section of another menu (File Menu Commands, for example), including the page break and table of links, and use that as the documentation template for your new menu.
3. Modify the title of the new section, change the description of the menu, and modify the links to menu options. You can also add K footnotes, which are index entries available from the **Index** tab of the **Help Topics** dialog box. If you are using Microsoft Word, use the **Footnotes** option in the **Insert** menu.
4. Change the # footnote, which is the topic's context string. The context string for a menu will be the value passed to the menu parameter in the .cnt file for the menu.

   In Microsoft Word, you can access the footnote by positioning the pointer to the left of the # symbol in the topic's title and double-clicking. This will open the footnotes window and position the pointer on the footnotes for the selected topic. Then edit the context string in the # footnote.

The text you define for a menu can be accessed as follows.

- From the menu's link in the **Contents** tab of the **Help Topics** dialog box.
- From the menu's link in the topic at the top of the .rtf file.

This text will not be available by means of F1 help on the menu name itself.

To add help text for a menu option

1. Follow the style of the AxfCore.rtf file and place a help section for the menu option in the file. For your convenience, you can copy the help section of another menu option, including the page break, and use that as the documentation template for your new menu option.
2. You can now follow the guidelines in the procedure "To add help text for the menu" (appearing earlier in this topic) to modify text and footnotes for the new menu option. Notice how the context string (# footnote) for a menu option is the resource ID prepended with the letter H.
3. In the .rtf file, update the link to the menu option from the topic for the menu under which the option is found.

See Also

Concepts

Adding Help for Dialog Boxes

An MFC application created with context-sensitive Help allows you to create help text for a dialog box and make it available to users by means of F1. If you wish, you can also provide a **Help** button on the dialog box.

To implement F1 help on a dialog box

1. Create the dialog box and note the ID. For information on how to add a dialog box to your application, see Example: Displaying a Dialog Box via a Menu Command.
2. Add a section to the .rtf. Follow the guidelines in "To add help text for the menu" (see Adding Menus and Menu Options) to modify text and footnotes for the new dialog box. Notice that the context string (# footnote) for a dialog box is the resource ID prepended with the letter H.

When you compile and run the program next, you will have F1 support to help on the dialog box.

To add a Help button to a dialog box

1. Follow the instructions in the section "To implement F1 help on a dialog box" appearing earlier in this topic.
2. Place a command button on the dialog box.
3. Add a function handler for the BN_CLICKED message for the command button's ID.
4. Edit the code for the new function and add the following two lines:

```
HELPINFO lhelpInfo;
CDialog::OnHelpInfo(&lhelpInfo);
```

If you want to use the same help for a dialog box that you use for a menu option, you can use HCW.exe to open your project's .hpj file and create an alias. An alias lets you share one resource's help with other resources.

See Also

Concepts

WinHelp: Context-Sensitive Help for Your Programs

Adding Control Help in a Dialog Box

Within a dialog box, you can make help available to users in several different ways.

- F1 help for a control
- Context-sensitive Help pointer
- Control help for static text strings
- What's This (right-click) help

To add F1 help for a control

- Edit your project's .hpj file and add the following line to the [MAP] section:

```
#include ..\resource.hm
```

Note

Your project's resource.hm file will only be created by the development environment when one or more dialog boxes have controls and when those controls have **HelpID** properties set to **True** in the **Properties Window**.

1. Add the WM_HELPINFO message for the dialog box object ID (selecting the dialog box's ID from the Object IDs list).
2. Add the following code to the OnHelpInfo function:

```
return  ::WinHelp(
        (HWND)pHelpInfo->hItemHandle,
        AfxGetApp()->m_pszHelpFilePath,
        HELP_WM_HELP,
        (DWORD)(LPVOID)rgmapCHID
        );
```

   Delete any comments or code that you may find in OnHelpInfo.

3. Before the OnHelpInfo function, place the declaration for the following array:

```
static DWORD rgmapCHID[] = {
       IDC_CTRL1, HIDC_CTRL1,
       IDC_IGNORE, -1,
       0,0
       };
```

   Each control ID must be mapped to the context ID of the control's help in the .rtf file. Recall that context IDs are the resource ID with an H in front. So in this example, IDC_CTRL1 is the control ID.

   If you create a control for which you do not want control help, specify the control ID and assign it -1. In this example, IDC_IGNORE will not get control help.

   End the array pairs with 0,0.

4. In the same source file that contains OnHelpInfo, put the following line:

```
#include "resource.hm"
```

5. For each control listed in your array, set the **HelpID** box in the **Properties** window to **True**.

To add context-sensitive Help pointer to a dialog box

1. For the dialog box, follow the instructions in the procedure, "To add F1 help for a control" appearing earlier in this topic.
2. In the dialog box's **Properties** window, go to the **Misc** section and set **Context Help** to **True**.

To provide control help for static text strings

1. Do not check the Help ID property for IDC_STATIC static strings.
2. Use the **Tab Order** option in the **Format** menu to make sure the tab order of the static text is just before the tab order of the list box or combo box.

To add What's This help to your dialog box

1. For the dialog box, follow the instructions in the procedure appearing earlier in this topic titled, "To add F1 help for a control"; if you do not want F1 help or a context-sensitive pointer with What's This help, do not add the WM_HELPINFO message.
2. For the dialog box that will support What's This help, add the WM_CONTEXTMENU message for the dialog box object ID.
3. Add the following code to the **OnContextMenu** function:

```
::WinHelp((HWND)*pWnd,
   AfxGetApp()->m_pszHelpFilePath,
   HELP_CONTEXTMENU,
   (DWORD)(LPVOID)rgmapCHID
   );
```

   Delete any comments or code that you may find in **OnContextMenu**.

See Also

Concepts

WinHelp: Context-Sensitive Help for Your Programs

Adding Context-Sensitive Help to an Existing MFC Application

This topic explains how to add context-sensitive Help to an application when you did not originally choose it as an MFC Application Wizard option.

For background information on the resources editors you will need, see the following topics:

- Menu Editor
- Accelerator Editor
- String Editor
- Toolbar Editor

Enabling context-sensitive Help support at a later stage requires several general steps. Each step is explained in more detail in the following topics. The overall steps are:

1. Run the MFC Application Wizard and choose the context-sensitive Help option to create a new application that has the help-related files and code. We will call this project HasHelp.
2. Copy resources from this application to your project.
3. Copy help-related files and code from the HasHelp project into your project.
4. Customize the contents of the files you copied from the HasHelp project, see Adding Menus and Menu Options.
5. Build the new version of your project and compile its Help file.

See Also

Concepts

WinHelp: Context-Sensitive Help for Your Programs

HTML Help: Context-Sensitive Help for Your Programs

Copying Help-Specific Resources to Your Project

In the following procedures, you will copy menu items, accelerator keys, and status-bar prompt strings from HasHelp to your project. As you do this, you'll not only learn about adding help to an application after the fact, you will also learn how easy it is to copy resources from one resource file to another.

In the following four procedures, you'll use the Menu Editor, Accelerator Editor, String Editor, and Toolbar Editor.

- Copying the Help Menu Resources
- Copying the Help Accelerator Resources
- Copying the Help-Related String Resources
- Copying the Help Mode Toolbar Button

See Also
Reference
Adding Context-Sensitive Help to an Existing MFC Application

Copying the Help Menu Resources

First, copy Help-related menu resources from the HasHelp project to your project.

To copy Help menu resources to your project

1. Open your project's .rc file and the HasHelp.rc, which are located in their respective project directories. These files display the contents of the **Resource View** panes for both your project and HasHelp.
2. Select **IDR_MAINFRAME** from the **Menu** folder from both resource files.

   Right-click at the top of the first **IDR_MAINFRAME - Menu** window and select **New Horizontal Tab Group**. This will allow you to view both menu windows.
3. Click the **Help Topics** menu item in the HasHelp **Help** menu. Then hold down the SHIFT key while you click the separator below the **Help Topics** item. Release the SHIFT key.

   This selects the menu item and separator.
4. Hold down the CTRL key and use the mouse to drag the highlighted menu items to the **Help** menu in your project's .rc file, above the **About** menu item. Release the mouse button and the CTRL key.

   This copies the menu item and the separator to your project.
5. Close the two **IDR_MAINFRAME** menu-editing windows.
6. Repeat steps 3 through 6, for the **Help** menu using the application-specific menu resources: `IDR_`*your_proj*`TYPE` and `IDR_HASHELTYPE`, respectively.

See Also

Reference

Copying Help-Specific Resources to Your Project

Copying the Help Accelerator Resources

Next, copy the accelerator resources for the menu items.

To copy the accelerator keys for Help resources

1. Open the Accelerator table resource, **IDR_MAINFRAME**, from both your project and HasHelp, resizing the accelerator editor windows, as described in Copying the Help Menu Resources, so they do not overlap.
2. Select the **ID_HELP** command from the HasHelp Accelerator table.
3. Hold down the CTRL key while you drag this entry to your project's accelerator editor window.

   It doesn't matter where in the window you drop the entry. This copies the accelerator keys (F1 and SHIFT+F1) for the Help menu item to your project.
4. Repeat this procedure for the **ID_CONTEXT_HELP** command.

   This maps to the help mode toolbar button, described later in this lesson, in Copying the Help Mode Toolbar Button.
5. Close both accelerator editor windows.

See Also

Reference

Copying Help-Specific Resources to Your Project

Copying the Help-Related String Resources

In this procedure you copy the command IDs for Help-related menu items.

To edit Help-related string resources

1. Open the String Table resource for both your application and HasHelp resizing both editor windows, as described in Copying the Help Menu Resources, so they do not overlap.
2. Locate the **AFX_IDS_IDLEMESSAGE** string in both tables, noting the difference in the captions.

   For an application without help, the MFC Application Wizard defines the default status-bar prompt to be `Ready`. This is the string displayed in the status bar when no other command prompt is being displayed.
3. Select the **Caption** column for **AFX_IDS_IDLEMESSAGE** in your project's string table, and edit the caption so that it matches the HasHelp version of this string resource: `For Help, press F1`.
4. Next, copy the following strings from the HasHelp String Table to your project's String Table:

   - `AFX_IDS_HELPMODEMESSAGE` (Displays in the status bar while the application is in help mode.)
   - `ID_CONTEXT_HELP` (Displays when the mouse pauses over the help mode toolbar button.)
   - `ID_HELP` (Called when the user presses F1.)
   - `ID_HELP_FINDER` (Displays for the **Help Topics** command on the **Help** menu.)

   The copying procedure is similar to the procedures for copying menus and accelerators. To copy several contiguous strings, hold the SHIFT key down while selecting the strings. Then hold the CTRL key down while dragging the selected strings to the new window.
5. Close the String Table resources.

See Also

Reference

Copying Help-Specific Resources to Your Project

Copying the Help Mode Toolbar Button

When you generate context-sensitive Help from the MFC Application Wizard, a button is created on the toolbar which, when selected, places the application in Help mode, just as pressing SHIFT+F1 does.

In this procedure you'll copy the Help mode button to your project's toolbar resource from the HasHelp project toolbar resource, simply by dragging it.

To copy the Help mode toolbar button

1. Open your project's toolbar resource and HasHelp's toolbar resource, resizing both editor windows, as described in Copying the Help Menu Resources, so they do not overlap and so you can see both subject toolbars.
2. Press CTRL and drag the Help mode button from HasHelp's toolbar onto your project's toolbar.
3. Save the resource file for your project (.rc file), and close the HasHelp resource file.

See Also

Reference

Copying Help-Specific Resources to Your Project

Copying the Help-Related Code and Files

The MFC Application Wizard generates not only help-specific code, but files that are used to build the application's Help system. In the following topics, you'll copy those files and the Help-related code from the HasHelp project (an MFC Application Wizard project created with context-sensitive Help support in Adding Context-Sensitive Help to an Existing MFC Application) into your project.

- Copying the Help Message Map Commands
- Copying the Help-Related Files
- Updating Project Settings for Context-Sensitive Help

See Also

Reference

Adding Context-Sensitive Help to an Existing MFC Application

Copying the Help Message Map Commands

The MFC Application Wizard places help-specific code into the message map in the frame window class implementation file, MainFrm.cpp.

To copy help-related code to your project

1. Open the HasHelp project MainFrm.cpp file and your project's version of MainFrm.cpp.
2. Scroll to the Help-related code, delineated by the `//global help commands` comment, in the message map section of the HasHelp MainFrm.cpp file.
3. Copy this code, shown below, from the HasHelp MainFrm.cpp file, and paste into the same position in the message map in your MainFrm.cpp.

   ```
   // Global help commands
   ON_COMMAND(ID_HELP_FINDER, CMDIFrameWnd::OnHelpFinder)
   ON_COMMAND(ID_HELP, CMDIFrameWnd::OnHelp)
   ON_COMMAND(ID_CONTEXT_HELP, CMDIFrameWnd::OnContextHelp)
   ON_COMMAND(ID_DEFAULT_HELP, CMDIFrameWnd::OnHelpFinder)
   ```

   You have already seen how **ID_HELP_FINDER** and **ID_CONTEXT_HELP** are called. When the user presses F1, the framework calls **ID_HELP** and directly handles this command so long as the application contains help support.
4. Close the HasHelp project's MainFrm.cpp file, and save the changes to your project's MainFrm.cpp.

See Also

Reference

Copying the Help-Related Code and Files

Copying the Help-Related Files

To perform the following procedure, use Windows Explorer or another file management utility.

You should only copy the help-related files of a project that was built at least once.

To copy help-related files to your project

1. Copy the entire HASHELP\HLP directory to your project's directory.

   The files contained in the \HLP directory include, among others:

   - The Help Project (.hpj) file, which tells the Help compiler how to build the Help file.
   - A file with the extension .cnt, needed to display the Help Contents screen.
   - A file with the extension .hm, created during the process of compiling the Help file, needed to provide context-sensitive Help for user interface objects. (If you did not build the HasHelp project, this file won't have been created yet.)

2. In your project's \HLP directory, rename:

   - HelpApp.hpj to *your_proj*.hpj
   - HelpApp.hm to *your_proj*.hm
   - HelpApp.cnt to *your_proj*.cnt
   - HelpApp.hlp to *your_proj*.hlp

   The bitmap and .rtf files do not need to be renamed.

3. In the .hpj file, rename the .hm file that the project expects to find from HelpApp.HM to *your_proj*.hm.

You will also want to edit the .hpj file and change names for the application in several places.

See Also

Reference

Copying the Help-Related Code and Files

Updating Project Settings for Context-Sensitive Help

After you've copied files and resources from the help-enabled project to the project that you are updating for help support, you then have to modify your project's solution and add custom build steps to several files.

Add a Help Files folder to the project's solution and add files

1. Add a new folder to your project by right-clicking the project name and selecting **Add** and then **New Folder**.
2. Name the folder Help Files.
3. Add all files in the hlp subdirectory except the .hpj file to the folder by right-clicking the **Help Files** folder, selecting **Add** and then selecting **Add Existing Item**. Select all the files in the hlp subdirectory and click **OK**.
4. Add the .hpj file from the hlp subdirectory to the Source Files folder.

After you have added files to your project, you must add custom build steps to the .hpj, .cnt, and resource.h files.

Adding custom build steps to the .hpj, .cnt, and resource.h files

1. Highlight the .hpj file in Solution Explorer and select **Property Pages** from the **View** menu.
2. Select the **General** property page in the **Custom Build Step** folder.
3. Copy the following into the **Command Line** box:

```
start /wait hcw /C /E /M "hlp\$(InputName).hpj"
if errorlevel 1 goto :Error
if not exist "hlp\$(InputName).hlp" goto :Error
copy "hlp\$(InputName).hlp" $(OutDir)
goto :done
:Error
echo hlp\$(InputName).hpj(1) : error:
type "hlp\$(InputName).log"
:done
```

4. Click **Outputs** and copy the following into the **Outputs** box:

```
$(OutDir)\$(InputName).hlp
```

5. Click **Dependencies** and copy the following into the **Dependencies** box:

```
hlp\AfxCore.rtf
hlp\AfxPrint.rtf
hlp\$(TargetName).hm
```

6. Select the .cnt file from the **Help Files** folder and copy the following text into the **Command Line** box:

```
copy "hlp\$(InputName).cnt" $(OutDir)
```

7. Click **Outputs** and copy the following into the **Outputs** box:

```
$(OutDir)\$(InputName).cnt
```

8. Select the resource.h file from the Header Files folder and copy the following text into the **Command Line** box:

```
echo. >>"hlp\$(TargetName).hm"
echo // Commands (ID_* and IDM_*) >>"hlp\$(TargetName).hm"
```

```
makehm ID_,HID_,0x10000 IDM_,HIDM_,0x10000 resource.h >>"hlp\$(TargetName).hm"
echo. >>"hlp\$(TargetName).hm"
echo // Prompts (IDP_*) >>"hlp\$(TargetName).hm"
makehm IDP_,HIDP_,0x30000 resource.h >>"hlp\$(TargetName).hm"
echo. >>"hlp\$(TargetName).hm"
echo // Resources (IDR_*) >>"hlp\$(TargetName).hm"
makehm IDR_,HIDR_,0x20000 resource.h >>"hlp\$(TargetName).hm"
echo. >>"hlp\$(TargetName).hm"
echo // Dialogs (IDD_*) >>"hlp\$(TargetName).hm"
makehm IDD_,HIDD_,0x20000 resource.h >>"hlp\$(TargetName).hm"
echo. >>"hlp\$(TargetName).hm"
echo // Frame Controls (IDW_*) >>"hlp\$(TargetName).hm"
makehm IDW_,HIDW_,0x50000 resource.h >>"hlp\$(TargetName).hm"
```

9. Click **Outputs** and copy the following into the **Outputs** box:

```
hlp\$(TargetName).hm
```

You are now ready to build your project with context-sensitive help.

See Also

Reference

Copying the Help-Related Code and Files

Context-Sensitive Help in an MFC Application

See Technical Note 28 for more information on details of MFC context-sensitive Help.

- OLE Support for Help
- Message-Map Support
- Using Help in Property Sheets: CPropertySheet and CPropertyPage

See Also

Concepts

WinHelp: Context-Sensitive Help for Your Programs

HTML Help: Context-Sensitive Help for Your Programs

OLE Support for Help

This article describes the requirements for Help support in OLE visual editing applications.

For SHIFT+F1 support, some special coordination is required between OLE containers and in-place active servers. When the user chooses to activate SHIFT+F1 Help mode, both applications need to enter the mode. Whichever application gets the message to enter the mode first notifies the other application. Then either application is prepared to respond when the user clicks the mouse on an item for which he or she wants help. Likewise, when the user terminates Help mode, the application that gets the message first notifies the other application. Note that MFC automatically handles this notification process between the container and the in-place server.

Caution
Neither the container nor the server can enter SHIFT+F1 help mode unless the other application also supports the mode.

For F1 help support, by contrast, whichever application gets the message when F1 is pressed handles the help request.

See Also

Reference

Context-Sensitive Help in an MFC Application

Concepts

Message-Map Support

Message-Map Support

The MFC Application Wizard adds a new menu command titled **Help Topics** to the two Help menus. Support for this command and for F1 and SHIFT+F1 help is provided through the message map. Topics covered in this article include:

- Help Commands in the Message Map
- About the Help Commands

Help Commands in the Message Map

To support the **Help Topics** menu item, F1 help, and SHIFT+F1 help, the MFC Application Wizard adds four entries to the message map for your **CFrameWnd**-derived or **CMDIFrameWnd**-derived class. This message map is in the MainFrm.cpp file, and the relevant message-map entries are located below the `// Global help commands` comment.

After you run the MFC Application Wizard, click **Context-Sensitive Help**. The message map for class `CMainFrame` will look like the following for an MDI application:

```
// CMainFrame

BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
   //{{AFX_MSG_MAP(CMainFrame)

   //   DO NOT EDIT what you see in these blocks of generated code !
   ON_WM_CREATE()
   //}}AFX_MSG_MAP
   // Global help commands
   ON_COMMAND(ID_HELP_FINDER, CMDIFrameWnd::OnHelpFinder)
   ON_COMMAND(ID_HELP, CMDIFrameWnd::OnHelp)
   ON_COMMAND(ID_CONTEXT_HELP, CMDIFrameWnd::OnContextHelp)
   ON_COMMAND(ID_DEFAULT_HELP, CMDIFrameWnd::OnHelpFinder)
END_MESSAGE_MAP()
```

For an SDI application, references to **CMDIFrameWnd** in this code are replaced by references to **CFrameWnd**.

About the Help Commands

The four help-related message-map entries follow the `// Global help commands` comment. The following table explains the purpose of each command ID used in these entries.

Help-Related Command IDs

Command ID	Purpose
ID_HELP_FINDER	Responds to the **Help Topics** menu item on the **Help** menu by displaying the Windows Contents screen.
ID_HELP	Responds to F1 by displaying a specific topic in Windows Help.
ID_CONTEXT_HELP	Responds to SHIFT+F1 by putting the application into Help mode.
ID_DEFAULT_HELP	Used when a specific help context cannot be found.

Notice that all of these commands are mapped to member functions of class **CMDIFrameWnd** (in the case of an MDI application) or **CFrameWnd** (in the case of an SDI application). Unlike most of the other commands you place into the message map, these have handler functions that are predefined by the class library. Making the message-map entry enables the command.

The application's accelerator table defines F1 for **ID_HELP** and SHIFT+F1 for **ID_CONTEXT_HELP**. You can change the keys used for these help functions by using Visual C++ to change the key values in the accelerator table. On Win32 systems, the operating system will generate the WM_HELP message when F1 is pressed.

When the user chooses a **Help** menu command, the framework calls the **CWinApp::WinHelp** member function. This action, in turn, starts the Windows Help program and passes context information to it.

See Also

Reference

Context-Sensitive Help in an MFC Application

Using Help in Property Sheets: CPropertySheet and CPropertyPage

Objects of class CPropertySheet represent property sheets, also called tab dialog boxes. A property sheet consists of a **CPropertySheet** object and one or more **CPropertyPage** objects. A property sheet is displayed by the framework as a window with a set of tab indices, with which the user selects the current page and an area for the currently selected page.

Help in **CPropertySheet** is supported by the F1 key and the **Help** button only. The **Help** button appears in the application framework by default. No intervention by the user is necessary. When the user adds the help information for each of the pages inside the property sheet, the help mechanism automatically displays the help for that page when the **Help** button is clicked.

To remove the **Help** button from a property sheet, modify the sheet and all its pages as follows:

```
mySheet.m_psh.dwFlags &= ~PSH_HASHELP;
   page1.m_psp.dwFlags &= ~PSP_HASHELP;
   page2.m_psp.dwFlags &= ~PSP_HASHELP;
   mySheet.AddPage( &page1 );
   mySheet.AddPage( &page2 );
   mySheet.DoModal();
```

The **m_psh** variable is of type **PROPSHEETHEADER**. The **m_psp** variables are of type **PROPSHEETPAGE**. If all the **HASHELP** flags are clear (**PSH_HASHELP** for the property sheet object, **PSP_HASHELP** for the property pages), the property sheet will be created without a Help button.

See Also

Reference

Context-Sensitive Help in an MFC Application
CPropertyPage Class
CPropertySheet Class

Adding Context-Sensitive Help to an MFC ActiveX Control

As the developer of an ActiveX control authored using MFC, you may want to provide online help to developers who use your control as they are developing their applications, as well as to those who use the control in a running program.

The following topics discuss several aspects of adding context-sensitive Help to an MFC ActiveX control.

- Creating an ActiveX Control for Help Access
- Adding Help to an Existing ActiveX Control
- Modifying the .rtf File
- Compiling the MFC Control's .hlp File
- Help at Design Time for an MFC Property Page
- Help at Design Time for All Properties
- Help at Run Time for an MFC ActiveX Control

See Also

Reference

Adding Context-Sensitive Help to an ATL ActiveX Control

Concepts

MFC ActiveX Controls

WinHelp: Context-Sensitive Help for Your Programs

Creating an ActiveX Control for Help Access

To create an MFC ActiveX application

1. From the **File** menu, select **Project** from the **New** submenu.
2. In the **New Project** dialog box, select **Visual C++ Projects** from the left pane.
3. Select **MFC ActiveX Control** from the right pane.
4. Select a name for your project and press OK.
5. From the **Application Settings** page, check **Generate help files**.

See Also

Reference

Adding Context-Sensitive Help to an MFC ActiveX Control

Adding Help to an Existing ActiveX Control

If you have an ActiveX control that was not created to support help, you can upgrade the control to support help.

1. Create a \Hlp subdirectory in the directory of your existing control. If your control name is MyControl, create MyControl\Hlp.

   Using the instructions in Creating an ActiveX Control for Help Access, create an ActiveX control with the same name as the control you want to upgrade. You can use any path for the new control, but do not overwrite the existing control. For example, you can use \Test\MyControl.

2. Copy all files in \Test\MyControl\Hlp to MyControl\Hlp (the existing control).

3. Copy \Test\MyControl\MyControl.HPJ to \MyControl.

4. Copy \Test\MyControl\MyControl.VCPROJ to \MyControl.

See Also

Reference

Adding Context-Sensitive Help to an MFC ActiveX Control

Modifying the .rtf File

The procedure for modifying the .rtf file for an MFC ActiveX control is identical to the procedure for modifying the .rtf file for an ATL ActiveX control; see Modifying Your .rtf File for more information.

See Also

Reference

Adding Context-Sensitive Help to an MFC ActiveX Control

Compiling the MFC Control's .hlp File

See Compiling Your Help File for more information on compiling a .hlp file from either the command line or the Help Compiler Workshop.

See Also

Reference

Adding Context-Sensitive Help to an MFC ActiveX Control

Help at Design Time for an MFC Property Page

The default property page has .h and .cpp files with filenames of *project_name*_ppg. You can create additional custom property pages by using the instructions in ActiveX Controls: Adding Another Custom Property Page.

The procedure is the same for enabling a property page for design-time help.

1. Open the property page's .h file and add the following line as a public member function:

```
BOOL OnHelp(LPCTSTR helpdir);
```

2. In the property page's .cpp file, add the following function:

```
BOOL C<project_name>PropPage::OnHelp(LPCTSTR)
   {
   // OnHelp is called by IPropertyPage::Help.  When
   // someone requests it, we supply help by calling
   // CWinApp::WinHelp
   AfxGetApp()->WinHelp(0,HELP_CONTENTS);

   return TRUE;
   }
```

   Be sure to replace <project_name> with your project's name.

See Also

Reference

Adding Context-Sensitive Help to an MFC ActiveX Control

Help at Design Time for All Properties

The Properties Window displays all properties for the currently selected ActiveX control. A user can highlight a property and press F1 to get help.

See the ATL section Help at Design Time for an ATL Property Page for the procedure.

See Also

Reference

Adding Context-Sensitive Help to an MFC ActiveX Control

Help at Run Time for an MFC ActiveX Control

When your MFC ActiveX control is embedded in a dialog box of some program and is displayed during the program's execution, a user can click the control, press F1, and get help.

To enable F1 help for a control at run time

1. In the *project_name*Ctl.h file, add the following line to the public constructor area,

```
LONG OnHelpCommand(UINT wParam, LONG lParam);
```

2. In the <project_name>Ctl.cpp file, add the following line to the end of the Message Map,

```
ON_MESSAGE(WM_HELP, OnHelpCommand)
```

3. In the <project_name>Ctl.cpp file, add the following function to the end of the file,

```
LONG C<project_name>Ctl::OnHelpCommand(UINT wParam, LONG lParam)
     {
     LPHELPINFO hi = (LPHELPINFO)lParam;
     if (AmbientUserMode())
        {
        AfxGetApp()->WinHelp(0x10065,HELP_CONTEXT);
        return TRUE;
        }
     return Default();

     }
```

Be sure to replace <project_name> with your project's name. In this example, the WinHelp function call takes the HELP_CONTEXT parameter and a numeric value that can be found in your project's .hm file. You could also specify

```
AfxGetApp()->WinHelp(0,HELP_CONTENTS);
```

to display the topic that is associated with the CONTENTS entry in the .hpj file.

See Also

Reference

Adding Context-Sensitive Help to an MFC ActiveX Control

Adding Context-Sensitive Help to an ATL ActiveX Control

As the developer of an ActiveX control, you may want to provide online help to developers who use your control as they are developing their applications, as well as to those who use the control when it is in a running program.

The following topics discuss several aspects of adding context-sensitive Help to an ATL ActiveX control using the Windows Help Compiler (HCW.exe).

- Adding Starter Help Files to Your ATL Project
- Modifying Your .rtf File
- Compiling Your .hlp File
- Help at Design Time for an ATL Property Page
- Help at Design Time for All Your Control's Properties
- Help at Run Time for an ATL ActiveX Control

See Knowledge Base article Q201540 for information on adding tooltips to an ATL ActiveX control.

See Also

Reference

Adding Context-Sensitive Help to an MFC ActiveX Control

Concepts

WinHelp: Context-Sensitive Help for Your Programs

Adding Starter Help Files to Your ATL Project

Because the ATL Project Application Wizard does not currently contain an option for .hlp file support, you need to use the files (such as .hpj and .rtf files) from an existing project that does support context-sensitive Help. After you copy the necessary files, you also must implement custom build steps.

To help files to your ATL project

1. Copy the \hlp directory from an MFC ActiveX control or MFC .exe project that was created with support for context-sensitive Help into a hlp directory in your ATL control's project directory. The project you copy files from should have been built at least once.
2. Rename the .hpj file so that it is called *your_project*.hpj. Then edit the file and change all occurrences of the old project name to your ATL project name. Also make sure that the relative paths used in the .hpj file refer correctly to the .rtf, .hlp, .bmp, and .hm files.
3. Make a folder called **Help Files** in **Solution Explorer**. This folder will be on the same level as the **Source Files**, **Header Files** and **Resource Files** folders.
4. Select the **Help Files** folder and select **Add Existing Item** from the **Project** menu. Add all files from the hlp directory, except the .hpj file, to the project.
5. Select the **Source Files** folder and select **Add Existing Item** from the **Project** menu. Add the .hpj file to the project.

Create custom build rules on the help files

1. Highlight the .hpj file in **Solution Explorer** and select **Property Pages** from the **View** menu.
2. Select the **General** property page in the **Custom Build Step** folder.
3. Copy the following into the **Command Line** box:

```
start /wait hcw /C /E /M "hlp\$(InputName).hpj"
if errorlevel 1 goto :Error
if not exist "hlp\$(InputName).hlp" goto :Error
copy "hlp\$(InputName).hlp" $(OutDir)
goto :done
:Error
echo hlp\$(InputName).hpj(1) : error:
type "hlp\$(InputName).log"
:done
```

4. Click **Outputs** and copy the following into the **Outputs** box:

```
$(OutDir)\$(InputName).hlp
```

5. Click **Dependencies** and copy the following into the **Dependencies** box:

```
hlp\AfxCore.rtf
hlp\AfxPrint.rtf
hlp\$(TargetName).hm
```

6. Select the .cnt file from the **Help Files** folder and copy the following text into the **Command Line** box:

```
copy "hlp\$(InputName).cnt" $(OutDir)
```

7. Click **Outputs** and copy the following into the **Outputs** box:

```
$(OutDir)\$(InputName).cnt
```

8. Select the resource.h file from the Header Files folder and copy the following text into the **Command Line** box:

```
echo. >>"hlp\$(TargetName).hm"
echo // Commands (ID_* and IDM_*) >>"hlp\$(TargetName).hm"
makehm ID_,HID_,0x10000 IDM_,HIDM_,0x10000 resource.h >>"hlp\$(TargetName).hm"
echo. >>"hlp\$(TargetName).hm"
echo // Prompts (IDP_*) >>"hlp\$(TargetName).hm"
makehm IDP_,HIDP_,0x30000 resource.h >>"hlp\$(TargetName).hm"
echo. >>"hlp\$(TargetName).hm"
echo // Resources (IDR_*) >>"hlp\$(TargetName).hm"
makehm IDR_,HIDR_,0x20000 resource.h >>"hlp\$(TargetName).hm"
echo. >>"hlp\$(TargetName).hm"
echo // Dialogs (IDD_*) >>"hlp\$(TargetName).hm"
makehm IDD_,HIDD_,0x20000 resource.h >>"hlp\$(TargetName).hm"
echo. >>"hlp\$(TargetName).hm"
echo // Frame Controls (IDW_*) >>"hlp\$(TargetName).hm"
makehm IDW_,HIDW_,0x50000 resource.h >>"hlp\$(TargetName).hm"
```

9. Click **Outputs** and copy the following into the **Outputs** box:

```
hlp\$(TargetName).hm
```

10. Add the `helpfile` line to the **module** attribute block appearing in your *projectname*.cpp file, as shown:

```
[ module(dll, uuid = "{2C929B27-4F22-421B-86D7-EA961BB71713}",
helpfile = "addhlp.hlp",
name = "addhlp",
helpstring = "addhlp 1.0 Type Library",
resource_name = "IDR_ADDHLP")
];
```

Or, if you have a nonattributed ATL project, add the `helpfile` line to your .idl file, between the `version` and `helpstring` lines in the library attributes statement, as shown:

```
uuid(2C929B27-4F22-421B-86D7-EA961BB71713),
version(1.0),
helpfile("addhlp.hlp"),
// added previous line; use the name of your .hlp file
helpstring("addhlp 1.0 Type Library ")
```

See Also

Reference

Adding Context-Sensitive Help to an ATL ActiveX Control

Modifying Your .rtf File

You will probably want to develop a help file that has an access point for each property, property page, and possibly a section for run-time help.

To add help text

1. In **Solution Explorer**, double-click to open the .rtf file. Any .rtf file you use should be listed in the [FILES] section of your .hpj file.
2. Follow the style of the .rtf file and create a help section. For your convenience, you should copy an existing help section, including the page break, and use that as the documentation template for your new section.
3. Modify the title of the new section.
4. Change the # footnote, which is the topic's context string. The context string for a section in the help file is the text's associated resource ID prepended with the letter H. For example, if you have a property page whose resource ID is **IDD_PP1**, the # footnote would be **HIDD_PP1**.

   You can access the footnote by positioning the pointer to the left of the # symbol in the topic's title and double-clicking. You can insert a footnote from the **Insert** menu.

See Also

Reference

Adding Context-Sensitive Help to an ATL ActiveX Control

Compiling Your .hlp File

Because of the custom build steps on the .hpj and .rtf files (see Adding Starter Help Files to Your ATL Project), anytime you build your project, the .hlp file will be updated (if you changed your .hpj or .rtf files).

However, if your help file does not behave as expected, you should invoke the Microsoft Help Workshop (HCW.exe) and open your project's .hpj file. When you select the **Save and Compile** option in Microsoft Help Workshop, you can see notes and warnings as the help compiler creates your .hlp file. You should resolve the warnings; notes are less critical to the success of your .hlp file.

Keeping your free-text search terms current also requires using Microsoft Help Workshop. Invoke the HCW.exe program and search its online help for the string FTS (for full-text search) and read how you can set a switch to cause your free-text word list to be updated when you compile help.

The online help for Microsoft Help Workshop is a good source of information on the process of creating an .hlp file.

See Also

Reference

Adding Context-Sensitive Help to an ATL ActiveX Control

Help at Design Time for an ATL Property Page

To successfully perform the procedures in this section, you must first follow the instructions in Adding Starter Help Files to Your ATL Project.

You can create custom property pages by using the instructions in Step 6: Adding a Property Page of the ATL Tutorial.

To enable a property page for design-time help

1. Edit the .h file for the full control you created for your ATL object and add the following entry to the Property Map:

   ```
   PROP_ENTRY("pp", 1, CLSID_pp)         // if property page class is not attributed
   PROP_ENTRY("pp", 1, __uuidof(Cpp))   // if property page class is attributed
   ```

   If property page class is attributed, #include the property page's .h file in control's header file.

   In this example, `pp` is the title you gave the property sheet in the **Strings** tab when you created the page (**IDS_DOCSTRING** in String Table). The number `1` is the sequential number of the property page; if this were the second page, you would use 2. `CLSID_pp` is the class ID of the property page.

2. Add the following definition to the .h file of the property page class after the **MSG_MAP**.

   ```
   // IPropertyPage::Help() implementation
   STDMETHOD(Help)(LPCOLESTR szPath)
   {
      USES_CONVERSION;
      TCHAR buff[100];
      TCHAR szFileName[256];

   // IDS_HELPFILEpp is the help filename stored as a string resource
      ::LoadString(_AtlBaseModule.GetModuleInstance(),IDS_HELPFILEpp, buff, 100);
   // assume helpfile string resource is in same module
   // alternatively use ATLLoadString
      lstrcpy(szFileName, OLE2T(szPath));
      lstrcat(szFileName, _T("\\"));
      lstrcat(szFileName, buff);
      ::WinHelp(0, szFileName, HELP_CONTENTS, 0);
      return S_OK;
   }
   ```

   This sample code uses `HELP_CONTENTS` as the parameter to the WinHelp function. The `HELP_CONTENTS` parameter accesses the text whose context string is equal to the **CONTENTS** entry in the .hpj file. You could also use the `HELP_CONTEXT` parameter and pass in an ID from the *projname*.hm file. For example,

   ```
   ::WinHelp(0, szFileName, HELP_CONTEXT, 0x10065);
   ```

   So if a property page resource has ID of **IDD_PP1**, you could look in *projname*.hm to see the numeric value of **HIDD_PP1**. The # footnote in the .rtf file for this section would then be **HIDD_PP1**. You may want to add the *projname*.hm file to the project and place it in the Help Files folder.

See Also

Reference

Adding Context-Sensitive Help to an ATL ActiveX Control

Help at Design Time for All Your Control's Properties

To successfully perform the procedures in this section, you must first follow the instructions in Adding Starter Help Files to Your ATL Project.

The **Properties** window (docking window) displays all properties for the ActiveX control. At the bottom of the **Properties** window, a short description of the highlighted property appears. This text is taken from a property's helpstring keyword.

Some containers may support the display of properties in a grid that is sensitive to F1 on individual properties.

To provide F1 help for individual properties

1. For each property, create a resource ID through the **Resource Symbols** option, which is available if you right-click the .rc name in **Resource View**. Furthermore, the resource has to begin either with **ID_**, **IDM_**, **IDP_**, **IDR_**, **IDP**, or **IDW_** because, as per the custom build step on the Resource.h file, these IDs will be processed and placed into the *projname*.hm file.

2. After you compile the project, look in the *projname*.hm file and notice the numeric value that is now assigned to your ID. You will also notice that the ID now begins with an H. This is the context string that you will use in the # footnote for the property's topic in the .rtf file.

3. In your project's .idl file, for each property, you can specify the helpcontext(*value*) parameter. This parameter must go within the square brackets in the comma-separated list.

   So, if you see the following line in the .hm file,

   ```
   HIDD_P1    0x2006C
   ```

   You would specify the following in the .idl file,

   ```
   [propget, id(1), helpstring("property Sides"), helpcontext(0x2006C)] HRESULT Sides([ou
   t, retval] short *pVal);
   [propput, id(1), helpstring("property Sides")] HRESULT Sides([in] short newVal);
   ```

See Also

Reference

Adding Context-Sensitive Help to an ATL ActiveX Control

Help at Run Time for an ATL ActiveX Control

To successfully perform the procedures in this section, you must first follow the instructions in Adding Starter Help Files to Your ATL Project.

When your ATL ActiveX control is embedded in a dialog box of a program and is displayed during the program's execution, a user can click the control and get F1 help on it.

To enable F1 help for a control at run time

1. Create a resource ID so that you will have a place in the .rtf file for the run-time help. See Help at Design Time for All Your Control's Properties for information.

2. Edit the .h file for the full control you created for your ATL object and add the following handler in the MSG_MAP:

```
MESSAGE_HANDLER(WM_HELP, OnHelpCommand)
```

3. Add the following definition to the .h file. You can put this code just after the **MSG_MAP**:

```
STDMETHOD(OnHelpCommand)(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
{
   TCHAR buff[100];

   ::LoadString(_AtlBaseModule.GetResourceInstance(),IDS_HELPFILEpp2, buff, 100);
   WinHelp(buff, HELP_CONTEXT, 0x200C4);
   return 0;
}
```

   This sample code uses **HELP_CONTEXT** as the parameter to the WinHelp function.

   In this example, **IDS_HELPFILEpp2** is the String Table ID for the help file name. If your control is only supplying run-time help, you need to create an entry in the **String Table** to make the association to a help file.

If your ATL control is not in the first tab position, you may need to handle the **WM_ONLBUTTONDOWN** message as follows:

```
STDMETHOD(OnLButtonDown)(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
{
   SetFocus();
   return 0;
}
```

See Also

Reference

Adding Context-Sensitive Help to an ATL ActiveX Control

Windows Sockets

This family of articles covers the MFC implementation of Windows Sockets. MFC supplies two classes to support programming network applications with the Windows Sockets API. Class CAsyncSocket encapsulates the Windows Sockets API one for one, giving advanced network programmers the most power and flexibility. Class CSocket provides a simplified interface for serializing data to and from a CArchive object.

In This Section

Windows Sockets in MFC

See Also

Other Resources

General Concepts

Windows Sockets in MFC

Note
MFC supports Windows Sockets 1 but does not support Windows Sockets 2. Windows Sockets 2 first shipped with Windows 98 and is the version included with Windows 2000.

MFC supplies two models for writing network communications programs with Windows Sockets, embodied in two MFC classes. This article describes these models and further details MFC sockets support. A "socket" is an endpoint of communication: an object through which your application communicates with other Windows Sockets applications across a network.

For information on Windows Sockets, including an explanation of the socket concept, see Windows Sockets: Background.

Sockets Programming Models

The two MFC Windows Sockets programming models are supported by the following classes:

- **CAsyncSocket**

  This class encapsulates the Windows Sockets API. CAsyncSocket is for programmers who know network programming and want the flexibility of programming directly to the sockets API but also want the convenience of callback functions for notification of network events. Other than packaging sockets in object-oriented form for use in C++, the only additional abstraction this class supplies is converting certain socket-related Windows messages into callbacks. For more information, see Windows Sockets: Socket Notifications.

- **CSocket**

  This class, derived from **CAsyncSocket**, supplies a higher level abstraction for working with sockets through an MFC CArchive object. Using a socket with an archive greatly resembles using MFC's file serialization protocol. This makes it easier to use than the **CAsyncSocket** model. CSocket inherits many member functions from **CAsyncSocket** that encapsulate Windows Sockets APIs; you will have to use some of these functions and understand sockets programming generally. But **CSocket** manages many aspects of the communication that you would have to do yourself using either the raw API or class **CAsyncSocket**. Most importantly, **CSocket** provides blocking (with background processing of Windows messages), which is essential to the synchronous operation of **CArchive**.

Creating and using **CSocket** and **CAsyncSocket** objects is described in Windows Sockets: Using Sockets with Archives and Windows Sockets: Using Class CAsyncSocket.

Windows Sockets DLLs

The Microsoft Windows operating systems supply the Windows Sockets dynamic-link libraries (DLL). Visual C++ supplies the appropriate header files and libraries and the Windows Sockets specification.

Note
Under Windows NT and Windows 2000, Windows Sockets support for 16-bit applications is based on WINSOCK.DLL. For 32-bit applications, the support is in WSOCK32.DLL. The APIs provided are identical except that the 32-bit versions have parameters widened to 32 bits. Under Win32, thread safety is supplied.

For more information about Windows Sockets, see:

- Windows Sockets: Stream Sockets
- Windows Sockets: Datagram Sockets
- Windows Sockets: Using Sockets with Archives
- Windows Sockets: Sequence of Operations
- Windows Sockets: Example of Sockets Using Archives

- Windows Sockets: How Sockets with Archives Work
- Windows Sockets: Using Class CAsyncSocket
- Windows Sockets: Deriving from Socket Classes
- Windows Sockets: Socket Notifications
- Windows Sockets: Blocking
- Windows Sockets: Byte Ordering
- Windows Sockets: Converting Strings
- Windows Sockets: Ports and Socket Addresses
- Windows Sockets: Sample

See Also

Concepts

Windows Sockets

Windows Sockets: Background

This article explains the nature and purpose of Windows Sockets. The article also:

- Defines the term "socket".
- Describes the SOCKET handle data type.
- Describes uses for sockets.

The Windows Sockets specification defines a binary-compatible network programming interface for Microsoft Windows. Windows Sockets are based on the UNIX sockets implementation in the Berkeley Software Distribution (BSD, release 4.3) from the University of California at Berkeley. The specification includes both BSD-style socket routines and extensions specific to Windows. Using Windows Sockets permits your application to communicate across any network that conforms to the Windows Sockets API. On Win32, Windows Sockets provide for thread safety.

Many network software vendors support Windows Sockets under network protocols including Transmission Control Protocol/Internet Protocol (TCP/IP), Xerox Network System (XNS), Digital Equipment Corporation's DECNet protocol, Novell Corporation's Internet Packet Exchange/Sequenced Packed Exchange (IPX/SPX), and others. Although the present Windows Sockets specification defines the sockets abstraction for TCP/IP, any network protocol can comply with Windows Sockets by supplying its own version of the dynamic-link library (DLL) that implements Windows Sockets. Examples of commercial applications written with Windows Sockets include X Windows servers, terminal emulators, and electronic mail systems.

Note
The purpose of Windows Sockets is to abstract away the underlying network so that you do not have to be knowledgeable a bout that network and so your application can run on any network that supports sockets. Consequently, this documentation does not discuss the details of network protocols.

The Microsoft Foundation Class Library (MFC) supports programming with the Windows Sockets API by supplying two classes. One of these classes, **CSocket**, provides a high level of abstraction to simplify your network communications programming.

The Windows Sockets specification, Windows Sockets: An Open Interface for Network Computing Under Microsoft Windows, now at version 1.1, was developed as an open networking standard by a large group of individuals and corporations in the TCP/IP community and is freely available for use. The sockets programming model supports one "communication domain" currently, using the Internet Protocol Suite. The specification is available in the Platform SDK.

Tip
Because sockets use the Internet Protocol Suite, they are the preferred route for applications that support Internet communic ations on the "information highway."

Definition of a Socket

A socket is a communication endpoint — an object through which a Windows Sockets application sends or receives packets of data across a network. A socket has a type and is associated with a running process, and it may have a name. Currently, sockets generally exchange data only with other sockets in the same "communication domain," which uses the Internet Protocol Suite.

Both kinds of sockets are bidirectional; they are data flows that can be communicated in both directions simultaneously (full-duplex).

Two socket types are available:

- Stream sockets

  Stream sockets provide for a data flow without record boundaries: a stream of bytes. Streams are guaranteed to be delivered and to be correctly sequenced and unduplicated.

- Datagram sockets

  Datagram sockets support a record-oriented data flow that is not guaranteed to be delivered and may not be sequenced as sent or unduplicated.

"Sequenced" means that packets are delivered in the order sent. "Unduplicated" means that you get a particular packet only

once.

Note
Under some network protocols, such as XNS, streams can be record oriented, as streams of records rather than streams of b ytes. Under the more common TCP/IP protocol, however, streams are byte streams. Windows Sockets provides a level of abs traction independent of the underlying protocol.

For information about these types and which kind of socket to use in which situations, see Windows Sockets: Stream Sockets and Windows Sockets: Datagram Sockets.

The SOCKET Data Type

Each MFC socket object encapsulates a handle to a Windows Sockets object. The data type of this handle is **SOCKET**. A **SOCKET** handle is analogous to the **HWND** for a window. MFC socket classes provide operations on the encapsulated handle.

The **SOCKET** data type is described in detail in the Platform SDK. See "Socket Data Type and Error Values" under Windows Sockets.

Uses for Sockets

Sockets are highly useful in at least three communications contexts:

- Client/server models.
- Peer-to-peer scenarios, such as chat applications.
- Making remote procedure calls (RPC) by having the receiving application interpret a message as a function call.

Tip
The ideal case for using MFC sockets is when you are writing both ends of the communication: using MFC at both ends. For more information on this topic, including how to manage the case when you're communicating with non-MFC applications, s ee Windows Sockets: Byte Ordering.

For more information, see Windows Sockets Specification: **ntohs**, **ntohl**, **htons**, **htonl**. Also, see the following topics:

- Windows Sockets: Using Sockets with Archives
- Windows Sockets: Example of Sockets Using Archives
- Windows Sockets: Using Class CAsyncSocket

See Also

Concepts

Windows Sockets in MFC

Windows Sockets: Stream Sockets

This article describes stream sockets, one of the two Windows Socket types available. (The other type is the datagram socket.)

Stream sockets provide for a data flow without record boundaries: a stream of bytes that can be bidirectional (the application is full duplex: it can both transmit and receive through the socket). Streams can be relied upon to deliver sequenced, unduplicated data. ("Sequenced" means that packets are delivered in the order sent. "Unduplicated" means that you get a particular packet only once.) Receipt of stream messages is guaranteed, and streams are well suited to handling large amounts of data.

The network transport layer may break up or group data into packets of reasonable size. The **CSocket** class will handle the packing and unpacking for you.

Streams are based on explicit connections: socket A requests a connection to socket B; socket B accepts or rejects the connection request.

A telephone call provides a good analogy for a stream. Under normal circumstances, the receiving party hears what you say in the order that you say it, without duplication or loss. Stream sockets are appropriate, for example, for implementations such as the File Transfer Protocol (FTP), which facilitates transferring ASCII or binary files of arbitrary size.

Stream sockets are preferable to datagram sockets when the data must be guaranteed to arrive and when data size is large. For more information about stream sockets, see the Windows Sockets specification. The specification is available in the Platform SDK.

Using stream sockets can be superior to applications designed to use a datagram socket for broadcasting to all receiving sockets on the network because

- The broadcast model is subject to network flood (or "storm") problems.
- The client-server model adopted subsequently is more efficient.
- The stream model supplies reliable data transfer, where the datagram model does not.
- The final model takes advantage of the ability to communicate between Unicode and ANSI socket applications that class CArchive lends to class CSocket.

Note
If you use class **CSocket**, you must use a stream. An MFC assertion fails if you specify the socket type as **SOCK_DGRAM**.

See Also

Concepts

Windows Sockets in MFC
Windows Sockets: Background

Windows Sockets: Datagram Sockets

This article describes datagram sockets, one of the two Windows Socket types available. (The other type is the stream socket.)

Datagram sockets support a bidirectional data flow that is not guaranteed to be sequenced or unduplicated. Datagrams also are not guaranteed to be reliable; they can fail to arrive. Datagram data may arrive out of order and possibly duplicated, but record boundaries in the data are preserved, as long as the records are smaller than the receiver's internal size limit. You are responsible for managing sequencing and reliability. (Reliability tends to be good on local-area networks [LAN] but less so on wide-area networks [WAN], such as the Internet.)

Datagrams are "connectionless", that is, no explicit connection is established; you send a datagram message to a specified socket and you can receive messages from a specified socket.

An example of a datagram socket is an application that keeps system clocks on the network synchronized. This illustrates an additional capability of datagram sockets in at least some settings: broadcasting messages to a large number of network addresses.

Datagram sockets are better than stream sockets for record-oriented data. For more information about datagram sockets, see the Windows Sockets specification, available in the Platform SDK.

See Also

Concepts

Windows Sockets in MFC
Windows Sockets: Background

Windows Sockets: Using Sockets with Archives

This article describes the CSocket programming model. Class CSocket supplies socket support at a higher level of abstraction than does class CAsyncSocket. **CSocket** uses a version of the MFC serialization protocol to pass data to and from a socket object through an MFC CArchive object. **CSocket** provides blocking (while managing background processing of Windows messages) and gives you access to **CArchive**, which manages many aspects of the communication that you would have to do yourself using either the raw API or class **CAsyncSocket**.

Tip
You can use class **CSocket** by itself, as a more convenient version of **CAsyncSocket**, but the simplest programming model is to use **CSocket** with a **CArchive** object.

For more information about how the implementation of sockets with archives works, see Windows Sockets: How Sockets with Archives Work. For example code, see Windows Sockets: Sequence of Operations and Windows Sockets: Example of Sockets Using Archives. For information about some of the functionality you can gain by deriving your own classes from the sockets classes, see Windows Sockets: Deriving from Socket Classes.

Note
If you are writing an MFC client program to communicate with established (non-MFC) servers, do not send C++ objects through the archive. Unless the server is an MFC application that understands the kinds of objects you want to send, it will not be able to receive and deserialize your objects. For related material on the subject of communicating with non-MFC applications, also see the article Windows Sockets: Byte Ordering.

The CSocket Programming Model

Using a **CSocket** object involves creating and associating together several MFC class objects. In the general procedure below, each step is taken by both the server socket and the client socket, except for step 3, in which each socket type requires a different action.

Tip
At run time, the server application usually starts first to be ready and "listening" when the client application seeks a connection. If the server is not ready when the client tries to connect, you typically require the user application to try connecting again later.

To set up communication between a server socket and a client socket

1. Construct a CSocket object.
2. Use the object to create the underlying **SOCKET** handle.

   For a **CSocket** client object, you should normally use the default parameters to Create, unless you need a datagram socket. For a **CSocket** server object, you must specify a port in the **Create** call.

Note
CArchive does not work with datagram sockets. If you want to use **CSocket** for a datagram socket, you must use the class as you would use **CAsyncSocket**, that is, without an archive. Because datagrams are unreliable (not guaranteed to arrive and may be repeated or out of sequence), they are not compatible with serialization through an archive. You expect a serialization operation to complete reliably and in sequence. If you try to use **CSocket** with a **CArchive** object for a datagram, an MFC assertion fails.

3. If the socket is a client, call CAsyncSocket::Connect to connect the socket object to a server socket.

   -or-

   If the socket is a server, call CAsyncSocket::Listen to begin listening for connect attempts from a client. Upon receiving a connection request, accept it by calling CAsyncSocket::Accept.

> **Note**
>
> The **Accept** member function takes a reference to a new, empty **CSocket** object as its parameter. You must construct this object before you call **Accept**. If this socket object goes out of scope, the connection closes. Do not call **Create** for this new socket object.

4. Create a CSocketFile object, associating the **CSocket** object with it.
5. Create a CArchive object for either loading (receiving) or storing (sending) data. The archive is associated with the **CSocketFile** object.

   Keep in mind that **CArchive** does not work with datagram sockets.
6. Use the **CArchive** object to pass data between the client and server sockets.

   Keep in mind that a given **CArchive** object moves data in one direction only: either for loading (receiving) or storing (sending). In some cases, you will use two **CArchive** objects: one for sending data, the other for receiving acknowledgments.

   After accepting a connection and setting up the archive, you can perform such tasks as validating passwords.
7. Destroy the archive, socket file, and socket objects.

> **Note**
>
> Class **CArchive** supplies the **IsBufferEmpty** member function specifically for use with class **CSocket**. If the buffer contains multiple data messages, for example, you need to loop until all of them are read and the buffer is cleared. Otherwise, your next notification that there is data to be received may be indefinitely delayed. Use **IsBufferEmpty** to assure that you retrieve all data.

The article Windows Sockets: Sequence of Operations illustrates both sides of this process with example code.

For more information, see:

- Windows Sockets: Stream Sockets
- Windows Sockets: Datagram Sockets

See Also

Reference

CSocket::Create

Concepts

Windows Sockets in MFC

Windows Sockets: Sequence of Operations

This article illustrates, side by side, the sequence of operations for a server socket and a client socket. Because the sockets use **CArchive** objects, they are necessarily stream sockets.

Sequence of Operations for a Stream Socket Communication

Up to the point of constructing a **CSocketFile** object, the following sequence is accurate (with a few parameter differences) for both **CAsyncSocket** and **CSocket**. From that point on, the sequence is strictly for **CSocket**. The following table illustrates the sequence of operations for setting up communication between a client and a server.

Setting Up Communication Between a Server and a Client

Server	Client
`// construct a socket` `CSocket sockSrvr;`	`// construct a socket` `CSocket sockClient;`
`// create the SOCKET` `sockSrvr.Create(nPort);`1,2	`// create the SOCKET` `sockClient.Create( );`2
`// start listening` `sockSrvr.Listen( );`	
	`// seek a connection` `sockClient.Connect(strAddr, nPort);`3,4

`// construct a new, empty socket`
`CSocket sockRecv;`
`// accept connection`
`sockSrvr.Accept( sockRecv );` 5	

`// construct file object` `CSocketFile file(&sockRecv);`	`// construct file object` `CSocketFile file(&sockClient);`

`// construct an archive`
`CArchive arIn(&file, CArchive::load);`
-or-
`CArchive arOut(&file, CArchive::store);`
– or Both –
	`// construct an archive`
	`CArchive arIn(&file, CArchive::load);`
	-or-
	`CArchive arOut(&file, CArchive::store);`
	– or Both –

`// use the archive to pass data:`
`arIn >> dwValue;`
-or-
`arOut << dwValue;`6
	`// use the archive to pass data:`
	`arIn >> dwValue;`
	-or-
	`arOut << dwValue;`6

1. Where *nPort* is a port number. See Windows Sockets: Ports and Socket Addresses for details about ports.

2. The server must always specify a port so clients can connect. The **Create** call sometimes also specifies an address. On the client side, use the default parameters, which ask MFC to use any available port.

3. Where *nPort* is a port number and *strAddr* is a machine address or an Internet Protocol (IP) address.

4. Machine addresses can take several forms: "ftp.microsoft.com", "microsoft.com". IP addresses use the "dotted number" form "127.54.67.32". The **Connect** function checks to see if the address is a dotted number (although it does not check to ensure the number is a valid machine on the network). If not, **Connect** assumes a machine name of one of the other forms.

5. When you call **Accept** on the server side, you pass a reference to a new socket object. You must construct this object first, but do not call **Create** for it. Keep in mind that if this socket object goes out of scope, the connection closes. MFC connects the new object to a **SOCKET** handle. You can construct the socket on the stack, as shown, or on the heap.

6. The archive and the socket file are closed when they go out of scope. The socket object's destructor also calls the Close member function for the socket object when the object goes out of scope or is deleted.

Additional Notes About the Sequence

The sequence of calls shown in the preceding table is for a stream socket. Datagram sockets, which are connectionless, do not require the CAsyncSocket::Connect, Listen, and Accept calls (although you can optionally use **Connect**). Instead, if you are using class **CAsyncSocket**, datagram sockets use the **CAsyncSocket::SendTo** and **ReceiveFrom** member functions. (If you use **Connect** with a datagram socket, you use **Send** and **Receive**.) Because **CArchive** does not work with datagrams, do not use **CSocket** with an archive if the socket is a datagram.

CSocketFile does not support all of **CFile**'s functionality; **CFile** members such as **Seek**, which make no sense for a socket communication, are unavailable. Because of this, some default MFC **Serialize** functions are not compatible with **CSocketFile**. This is particularly true of the **CEditView** class. You should not try to serialize **CEditView** data through a **CArchive** object attached to a **CSocketFile** object using **CEditView::SerializeRaw**; use **CEditView::Serialize** instead (not documented). The SerializeRaw function expects the file object to have functions, such as **Seek**, that **CSocketFile** does not support.

For more information, see:

- Windows Sockets: Using Sockets with Archives
- Windows Sockets: Using Class CAsyncSocket
- Windows Sockets: Ports and Socket Addresses
- Windows Sockets: Stream Sockets
- Windows Sockets: Datagram Sockets

See Also

Reference

CSocket Class
CAsyncSocket::Create
CAsyncSocket::Close

Concepts

Windows Sockets in MFC

Windows Sockets: Example of Sockets Using Archives

This article presents an example of using class CSocket. The example employs **CArchive** objects to serialize data through a socket. Note that this is not document serialization to or from a file.

The following example illustrates how you use the archive to send and receive data through **CSocket** objects. The example is designed so that two instances of the application (on the same machine or on different machines on the network) exchange data. One instance sends data, which the other instance receives and acknowledges. Either application can initiate an exchange, and either can act as server or as client to the other application. The following function is defined in the application's view class:

```
void CBlabberView::PacketSerialize(long nPackets, CArchive& arData,
                                   CArchive& arAck)
{
    if (arData.IsStoring())
    {
        CString strText;

        for(int p = 0; p < nPackets; p++)
        {
            BYTE bValue = (BYTE)(rand()%256);
            WORD nCopies = (WORD)(rand()%32000);

            // Send header information
            arData << bValue << nCopies;
            for(int c = 0; c < nCopies; c++)
            {
                // Send data
                arData << bValue;
            }

            Text.Format("Received Packet %d of %d
                (Value=%d,Copies=%d)",p,nPackets,(int)bValue,nCopies);

            // Send receipt string
            arData << strText;
            arData.Flush();

            // Receive acknowledgment
            arAck >> strText;
            // display it
            DisplayMessage(strText);
        }
    }
    else
    {
        CString strText;
        BYTE bCheck;
        WORD nCopies;

        for(int p = 0; p < nPackets; p++)
        {
            // Receive header information
            arData >> bCheck >> nCopies;
            for(int c = 0; c < nCopies; c++)
            {
                // Receive data
                arData >> bValue;
                if (nCheck != bValue)
                    AfxMessageBox("Packet Failure");
            }
        }

        // Receive receipt string and display it
        arData >> strText;
        DisplayMessage(strText);
```

```
        Text.Format("Sent Packet %d of %d
            (Value=%d,Copies=%d)",p,nPackets,(int)bValue,nCopies);

        // Send acknowledgment
        arAck << strText;
        arAck.Flush();
    }
}
```

The most important thing about this example is that its structure parallels that of an MFC **Serialize** function. The **PacketSerialize** member function consists of an **if** statement with an **else** clause. The function receives two CArchive references as parameters: *arData* and *arAck*. If the *arData* archive object is set for storing (sending), the **if** branch executes; otherwise, if *arData* is set for loading (receiving) the function takes the **else** branch. For more information about serialization in MFC, see Serialization.

Note

The *arAck* archive object is assumed to be the opposite of *arData*. If *arData* is for sending, *arAck* receives, and the converse is true.

For sending, the example function loops for a specified number of times, each time generating some random data for demonstration purposes. Your application would obtain real data from some source, such as a file. The *arData* archive's insertion operator (**<<**) is used to send a stream of three consecutive chunks of data:

- A "header" that specifies the nature of the data (in this case, the value of the `bValue` variable and how many copies will be sent).

  Both items are generated randomly for this example.

- The specified number of copies of the data.

  The inner **for** loop sends `bValue` the specified number of times.

- A string called `strText` that the receiver displays to its user.

For receiving, the function operates similarly, except that it uses the archive's extraction operator (**>>**) to get data from the archive. The receiving application verifies the data it receives, displays the final "Received" message, and then sends back a message that says "Sent" for the sending application to display.

In this communications model, the word "Received", the message sent in the `strText` variable, is for display at the other end of the communication, so it specifies to the receiving user that a certain number of packets of data have been received. The receiver replies with a similar string that says "Sent", for display on the original sender's screen. Receipt of both strings indicates that successful communication has occurred.

Caution

If you are writing an MFC client program to communicate with established (non-MFC) servers, do not send C++ objects through the archive. Unless the server is an MFC application that understands the kinds of objects you want to send, it won't be able to receive and deserialize your objects. An example in the article Windows Sockets: Byte Ordering shows a communication of this type.

For more information, see Windows Sockets Specification: **htonl**, **htons**, **ntohl**, **ntohs**. Also, for more information, see:

- Windows Sockets: Deriving from Socket Classes
- Windows Sockets: How Sockets with Archives Work
- Windows Sockets: Background

See Also

Reference

CArchive::IsStoring
CArchive::operator <<
CArchive::operator >>

CArchive::Flush
CObject::Serialize

Concepts

Windows Sockets in MFC

Windows Sockets: How Sockets with Archives Work

This article explains how a CSocket object, a CSocketFile object, and a CArchive object are combined to simplify sending and receiving data through a Windows Socket.

The article Windows Sockets: Example of Sockets Using Archives presents the **PacketSerialize** function. The archive object in the **PacketSerialize** example works much like an archive object passed to an MFC Serialize function. The essential difference is that for sockets, the archive is attached not to a standard CFile object (typically associated with a disk file) but to a **CSocketFile** object. Rather than connecting to a disk file, the **CSocketFile** object connects to a **CSocket** object.

A **CArchive** object manages a buffer. When the buffer of a storing (sending) archive is full, an associated **CFile** object writes out the buffer's contents. Flushing the buffer of an archive attached to a socket is equivalent to sending a message. When the buffer of a loading (receiving) archive is full, the **CFile** object stops reading until the buffer is available again.

Class **CSocketFile** derives from **CFile**, but it does not support CFile member functions such as the positioning functions (**Seek**, **GetLength**, **SetLength**, and so on), the locking functions (**LockRange**, **UnlockRange**), or the **GetPosition** function. All the CSocketFile object must do is write or read sequences of bytes to or from the associated **CSocket** object. Because a file is not involved, operations such as **Seek** and **GetPosition** make no sense. **CSocketFile** is derived from **CFile**, so it would normally inherit all of these member functions. To prevent this, the unsupported **CFile** member functions are overridden in **CSocketFile** to throw a CNotSupportedException.

The **CSocketFile** object calls member functions of its **CSocket** object to send or receive data.

The following figure shows the relationships among these objects on both sides of the communication.

CArchive, CSocketFile, and CSocket

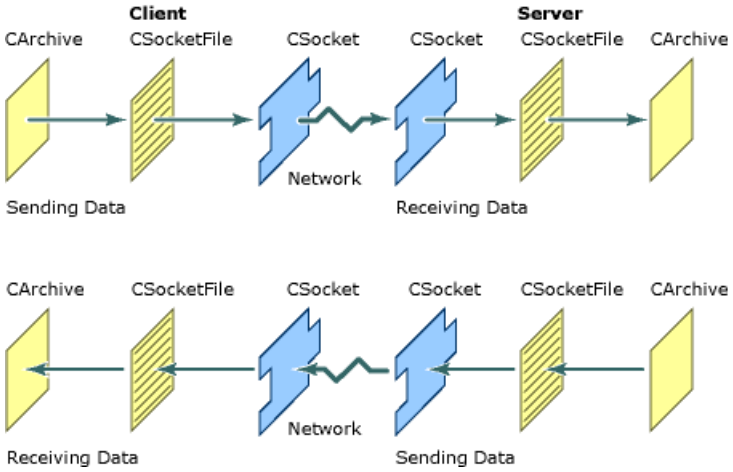

The purpose of this apparent complexity is to shield you from the necessity of managing the details of the socket yourself. You create the socket, the file, and the archive, and then begin sending or receiving data by inserting it to the archive or extracting it from the archive. CArchive, CSocketFile, and CSocket manage the details behind the scenes.

A **CSocket** object is actually a two-state object: sometimes asynchronous (the usual state) and sometimes synchronous. In its asynchronous state, a socket can receive asynchronous notifications from the framework. However, during an operation such as receiving or sending data the socket becomes synchronous. This means the socket will receive no further asynchronous notifications until the synchronous operation has completed. Because it switches modes, you can, for example, do something like the following:

```
CMySocket::OnReceive( )
{
    // ...
    ar >> str;
    // ...
}
```

If **CSocket** were not implemented as a two-state object, it might be possible to receive additional notifications for the same kind of event while you were processing a previous notification. For example, you might get an `OnReceive` notification while processing an `OnReceive`. In the code fragment above, extracting `str` from the archive might lead to recursion. By switching states, **CSocket** prevents recursion by preventing additional notifications. The general rule is no notifications within notifications.

Note

A **CSocketFile** can also be used as a (limited) file without a **CArchive** object. By default, the **CSocketFile** constructor's *bArchiveCompatible* parameter is **TRUE**. This specifies that the file object is for use with an archive. To use the file object without an archive, pass **FALSE** in the *bArchiveCompatible* parameter.

In its "archive compatible" mode, a **CSocketFile** object provides better performance and reduces the danger of a "deadlock." A deadlock occurs when both the sending and receiving sockets are waiting on each other, or waiting for a common resource. This situation might occur if the **CArchive** object worked with the **CSocketFile** the way it does with a **CFile** object. With **CFile**, the archive can assume that if it receives fewer bytes than it requested, the end of file has been reached. With **CSocketFile**, however, data is message based; the buffer can contain multiple messages, so receiving fewer than the number of bytes requested does not imply end of file. The application does not block in this case as it might with **CFile**, and it can continue reading messages from the buffer until the buffer is empty. The IsBufferEmpty function in **CArchive** is useful for monitoring the state of the archive's buffer in such a case.

For more information, see Windows Sockets: Using Sockets with Archives

See Also

Reference

CObject::Serialize

Concepts

Windows Sockets in MFC

Windows Sockets: Using Class CAsyncSocket

This article explains how to use class CAsyncSocket. Be aware that this class encapsulates the Windows Sockets API at a very low level. **CAsyncSocket** is for use by programmers who know network communications in detail but want the convenience of callbacks for notification of network events. Based on this assumption, this article provides only basic instruction. You should probably consider using **CAsyncSocket** if you want Windows Sockets' ease of dealing with multiple network protocols in an MFC application but do not want to sacrifice flexibility. You might also feel that you can get better efficiency by programming the communications more directly yourself than you could using the more general alternative model of class **CSocket**.

CAsyncSocket is documented in the *MFC Reference*. Visual C++ also supplies the Windows Sockets specification, located in the Platform SDK. The details are left to you. Visual C++ does not supply a sample application for **CAsyncSocket**.

If you are not highly knowledgeable about network communications and want a simple solution, use class CSocket with a **CArchive** object. See Windows Sockets: Using Sockets with Archives for more information.

This article covers:

- Creating and using a **CAsyncSocket** object.
- Your responsibilities with CAsyncSocket.

Creating and Using a CAsyncSocket Object

To use CAsyncSocket

1. Construct a CAsyncSocket object and use the object to create the underlying **SOCKET** handle.

   Creation of a socket follows the MFC pattern of two-stage construction.

   For example:

   ```
   CAsyncSocket sock;
   sock.Create( );    // Use the default parameters
   ```

   -or-

   ```
   CAsyncSocket* pSocket = new CAsyncSocket;
   int nPort = 27;
   pSocket->Create( nPort, SOCK_DGRAM );
   ```

   The first constructor above creates a **CAsyncSocket** object on the stack. The second constructor creates a **CAsyncSocket** on the heap. The first Create call above uses the default parameters to create a stream socket. The second **Create** call creates a datagram socket with a specified port and address. (You can use either **Create** version with either construction method.)

   The parameters to **Create** are:

   - A "port": a short integer.

     For a server socket, you must specify a port. For a client socket, you typically accept the default value for this parameter, which lets Windows Sockets select a port.

   - A socket type: **SOCK_STREAM** (the default) or **SOCK_DGRAM**.

   - A socket "address," such as "ftp.microsoft.com" or "128.56.22.8".

     This is your Internet Protocol (IP) address on the network. You will probably always rely on the default value for this parameter.

   The terms "port" and "socket address" are explained in Windows Sockets: Ports and Socket Addresses.

2. If the socket is a client, connect the socket object to a server socket, using CAsyncSocket::Connect.

   -or-

If the socket is a server, set the socket to begin listening (with CAsyncSocket::Listen) for connect attempts from a client. Upon receiving a connection request, accept it with CAsyncSocket::Accept.

After accepting a connection, you can perform such tasks as validating passwords.

Note
The **Accept** member function takes a reference to a new, empty **CSocket** object as its parameter. You must construct this object before you call **Accept**. If this socket object goes out of scope, the connection closes. Do not call **Create** for this new socket object. For an example, see the article Windows Sockets: Sequence of Operations.

3. Carry out communications with other sockets by calling the **CAsyncSocket** object's member functions that encapsulate the Windows Sockets API functions.

   See the Windows Sockets specification and class CAsyncSocket in the *MFC Reference*.

4. Destroy the **CAsyncSocket** object.

   If you created the socket object on the stack, its destructor is called when the containing function goes out of scope. If you created the socket object on the heap, using the **new** operator, you are responsible for using the **delete** operator to destroy the object.

   The destructor calls the object's Close member function before destroying the object.

For an example of this sequence in code (actually for a **CSocket** object), see Windows Sockets: Sequence of Operations.

Your Responsibilities with CAsyncSocket

When you create an object of class CAsyncSocket, the object encapsulates a Windows **SOCKET** handle and supplies operations on that handle. When you use **CAsyncSocket**, you must deal with all the issues you might face if using the API directly. For example:

- "Blocking" scenarios.
- Byte order differences between the sending and receiving machines.
- Converting between Unicode and multibyte character set (MBCS) strings.

For definitions of these terms and additional information, see Windows Sockets: Blocking, Windows Sockets: Byte Ordering, Windows Sockets: Converting Strings.

Despite these issues, class **CAsycnSocket** may be the right choice for you if your application requires all the flexibility and control you can get. If not, you should consider using class **CSocket** instead. **CSocket** hides a lot of detail from you: it pumps Windows messages during blocking calls and gives you access to **CArchive**, which manages byte order differences and string conversion for you.

For more information, see:

- Windows Sockets: Background
- Windows Sockets: Stream Sockets
- Windows Sockets: Datagram Sockets

See Also

Concepts

Windows Sockets in MFC

Windows Sockets: Deriving from Socket Classes

This article describes some of the functionality you can gain by deriving your own class from one of the socket classes.

You can derive your own socket classes from either CAsyncSocket or CSocket to add your own functionality. In particular, these classes supply a number of virtual member functions that you can override. These functions include OnReceive, OnSend, OnAccept, OnConnect, and OnClose. You can override the functions in your derived socket class to take advantage of the notifications they provide when network events occur. The framework calls these notification callback functions to notify you of important socket events, such as the receipt of data that you can begin reading. For more information about notification functions, see Windows Sockets: Socket Notifications.

Additionally, class **CSocket** supplies the OnMessagePending member function (an advanced overridable). MFC calls this function while the socket is pumping Windows-based messages. You can override **OnMessagePending** to look for particular messages from Windows and respond to them.

The default version of **OnMessagePending** supplied in class **CSocket** examines the message queue for **WM_PAINT** messages while waiting for a blocking call to complete. It dispatches paint messages to improve display quality. Aside from doing something useful, this illustrates one way you might override the function yourself. As another example, consider using **OnMessagePending** for the following task. Suppose you display a modeless dialog box while waiting for a network transaction to complete. The dialog box contains a Cancel button that the user can use to cancel blocking transactions that take too long. Your `OnMessagePending` override might pump messages related to this modeless dialog box.

In your `OnMessagePending` override, return either **TRUE** or the return from a call to the base-class version of **OnMessagePending**. Call the base-class version if it performs work that you still want done.

For more information, see:

- Windows Sockets: Using Sockets with Archives
- Windows Sockets: Using Class CAsyncSocket
- Windows Sockets: Blocking
- Windows Sockets: Byte Ordering
- Windows Sockets: Converting Strings

See Also

Concepts

Windows Sockets in MFC

Windows Sockets: Socket Notifications

This article describes the notification functions in the socket classes. These member functions are callback functions that the framework calls to notify your socket object of important events. The notification functions are:

- OnReceive: Notifies this socket that there is data in the buffer for it to retrieve by calling Receive.
- OnSend: Notifies this socket that it can now send data by calling Send.
- OnAccept: Notifies this listening socket that it can accept pending connection requests by calling Accept.
- OnConnect: Notifies this connecting socket that its connection attempt completed: perhaps successfully or perhaps in error.
- OnClose: Notifies this socket that the socket it is connected to has closed.

Note
An additional notification function is OnOutOfBandData. This notification tells the receiving socket that the sending socket has "out-of-band" data to send. Out-of-band data is a logically independent channel associated with each pair of connected stream sockets. The out-of-band channel is typically used to send "urgent" data. MFC supports out-of-band data. Advanced users working with class CAsyncSocket might need to use the out-of-band channel, but users of class CSocket are discouraged from using it. The easier way is to create a second socket for passing such data. For more information about out-of-band data, see the Windows Sockets specification, available in the Platform SDK.

If you derive from class **CAsyncSocket**, you must override the notification functions for those network events of interest to your application. If you derive a class from class **CSocket**, it is your choice whether to override the notification functions of interest. You can also use **CSocket** itself, in which case the notification functions default to doing nothing.

These functions are overridable callback functions. **CAsyncSocket** and **CSocket** convert messages to notifications, but you must implement how the notification functions respond if you wish to use them. The notification functions are called at the time your socket is notified of an event of interest, such as the presence of data to be read.

MFC calls the notification functions to let you customize your socket's behavior at the time it is notified. For example, you might call **Receive** from your **OnReceive** notification function, that is, on being notified that there is data to read, you call **Receive** to read it. This approach is not necessary, but it is a valid scenario. As an alternative, you might use your notification function to track progress, print **TRACE** messages, and so on.

You can take advantage of these notifications by overriding the notification functions in a derived socket class and providing an implementation.

During an operation such as receiving or sending data, a **CSocket** object becomes synchronous. During the synchronous state, any notifications meant for other sockets are queued while the current socket waits for the notification it wants. (For example, during a **Receive** call, the socket wants a notification to read.) Once the socket completes its synchronous operation and becomes asynchronous again, other sockets can begin receiving the queued notifications.

Note
In **CSocket**, the **OnConnect** notification function is never called. For connections, you call **Connect**, which will return when the connection is completed (either successfully or in error). How connection notifications are handled is an MFC implementation detail.

For details about each notification function, see the function under class **CAsyncSocket** in the *MFC Reference*. For source code and information about MFC samples, see MFC Samples.

For more information, see:

- Windows Sockets: Using Class CAsyncSocket
- Windows Sockets: Deriving from Socket Classes
- Windows Sockets: How Sockets with Archives Work
- Windows Sockets: Blocking

- Windows Sockets: Byte Ordering
- Windows Sockets: Converting Strings

See Also

Concepts

Windows Sockets in MFC

Windows Sockets: Blocking

This article and two companion articles explain several issues in Windows Sockets programming. This article covers blocking. The other issues are covered in the articles: Windows Sockets: Byte Ordering and Windows Sockets: Converting Strings.

If you use or derive from class CAsyncSocket, you will need to manage these issues yourself. If you use or derive from class CSocket, MFC manages them for you.

Blocking

A socket can be in "blocking mode" or "nonblocking mode." The functions of sockets in blocking (or synchronous) mode do not return until they can complete their action. This is called blocking because the socket whose function was called cannot do anything — is blocked — until the call returns. A call to the **Receive** member function, for example, might take an arbitrarily long time to complete as it waits for the sending application to send (this is if you are using **CSocket**, or using **CAsyncSocket** with blocking). If a **CAsyncSocket** object is in nonblocking mode (operating asynchronously), the call returns immediately and the current error code, retrievable with the GetLastError member function, is **WSAEWOULDBLOCK**, indicating that the call would have blocked had it not returned immediately because of the mode. (**CSocket** never returns **WSAEWOULDBLOCK**. The class manages blocking for you.)

The behavior of sockets is different under 32-bit and 64-bit operating systems (such as Windows 95 or Windows 98) than under 16-bit operating systems (such as Windows 3.1). Unlike 16-bit operating systems, the 32-bit and 64-bit operating systems use preemptive multitasking and provide multithreading. Under the 32-bit and 64-bit operating systems, you can put your sockets in separate worker threads. A socket in a thread can block without interfering with other activities in your application and without spending compute time on the blocking. For information on multithreaded programming, see the article Multithreading.

Note
In multithreaded applications, you can use the blocking nature of **CSocket** to simplify your program's design without affecting the responsiveness of the user interface. By handling user interactions in the main thread and **CSocket** processing in alternate threads, you can separate these logical operations. In an application that is not multithreaded, these two activities must be combined and handled as a single thread, which usually means using **CAsyncSocket** so you can handle communications requests on demand, or overriding **CSocket::OnMessagePending** to handle user actions during lengthy synchronous activity.

The rest of this discussion is for programmers targeting 16-bit operating systems:

Normally, if you are using **CAsyncSocket**, you should avoid using blocking operations and operate asynchronously instead. In asynchronous operations, from the point at which you receive a **WSAEWOULDBLOCK** error code after calling **Receive**, for example, you wait until your **OnReceive** member function is called to notify you that you can read again. Asynchronous calls are made by calling back your socket's appropriate callback notification function, such as OnReceive.

Under Windows, blocking calls are considered bad practice. By default, CAsyncSocket supports asynchronous calls, and you must manage the blocking yourself using callback notifications. Class CSocket, on the other hand, is synchronous. It pumps Windows messages and manages blocking for you.

For more information about blocking, see the Windows Sockets specification. For more information about "On" functions, see Windows Sockets: Socket Notifications and Windows Sockets: Deriving from Socket Classes.

For more information, see:

- Windows Sockets: Using Class CAsyncSocket
- Windows Sockets: Using Sockets with Archives
- Windows Sockets: Background
- Windows Sockets: Stream Sockets
- Windows Sockets: Datagram Sockets

See Also

Reference

CAsyncSocket::OnSend

Concepts

Windows Sockets: Byte Ordering

This article and two companion articles explain several issues in Windows Sockets programming. This article covers byte ordering. The other issues are covered in the articles: Windows Sockets: Blocking and Windows Sockets: Converting Strings.

If you use or derive from class CAsyncSocket, you will need to manage these issues yourself. If you use or derive from class CSocket, MFC manages them for you.

Byte Ordering

Different machine architectures sometimes store data using different byte orders. For example, Intel-based machines store data in the reverse order of Macintosh (Motorola) machines. The Intel byte order, called "little-Endian," is also the reverse of the network standard "big-Endian" order. The following table explains these terms.

Big- and Little-Endian Byte Ordering

Byte ordering	Meaning
Big-Endian	The most significant byte is on the left end of a word.
Little-Endian	The most significant byte is on the right end of a word.

Typically, you do not have to worry about byte-order conversion for data that you send and receive over the network, but there are situations in which you must convert byte orders.

When You Must Convert Byte Orders

You need to convert byte orders in the following situations:

- You are passing information that needs to be interpreted by the network, as opposed to the data you are sending to another machine. For example, you might pass ports and addresses, which the network must understand.
- The server application with which you are communicating is not an MFC application (and you do not have source code for it). This calls for byte order conversions if the two machines do not share the same byte ordering.

When You Do Not Have to Convert Byte Orders

You can avoid the work of converting byte orders in the following situations:

- The machines on both ends can agree not to swap bytes, and both machines use the same byte order.
- The server you are communicating with is an MFC application.
- You have source code for the server you're communicating with, so you can tell explicitly whether you must convert byte orders or not.
- You can port the server to MFC. This is fairly easy to do, and the result is usually smaller, faster code.

Working with CAsyncSocket, you must manage any necessary byte-order conversions yourself. Windows Sockets standardizes the "big-Endian" byte-order model and provides functions to convert between this order and others. CArchive, however, which you use with CSocket, uses the opposite ("little-Endian") order, but **CArchive** takes care of the details of byte-order conversions for you. By using this standard ordering in your applications, or using Windows Sockets byte-order conversion functions, you can make your code more portable.

The ideal case for using MFC sockets is when you are writing both ends of the communication: using MFC at both ends. If you are writing an application that will communicate with non-MFC applications, such as an FTP server, you will probably need to manage byte-swapping yourself before you pass data to the archive object, using the Windows Sockets conversion routines **ntohs**, **ntohl**, **htons**, and **htonl**. An example of these functions used in communicating with a non-MFC application appears later in this article.

Note
When the other end of the communication is not an MFC application, you also must avoid streaming C++ objects derived from **CObject** into your archive because the receiver will not be able to handle them. See the note in Windows Sockets: Using Sockets with Archives.

For more information about byte orders, see the Windows Sockets specification, available in the Platform SDK.

A Byte-Order Conversion Example

The following example shows a serialization function for a **CSocket** object that uses an archive. It also illustrates using the byte-order conversion functions in the Windows Sockets API.

This example presents a scenario in which you are writing a client that communicates with a non-MFC server application for which you have no access to the source code. In this scenario, you must assume that the non-MFC server uses standard network byte order. In contrast, your MFC client application uses a **CArchive** object with a **CSocket** object, and **CArchive** uses "little-Endian" byte order, the opposite of the network standard.

Suppose the non-MFC server with which you plan to communicate has an established protocol for a message packet like the following:

```
struct Message
{
    long MagicNumber;
    unsigned short Command;
    short Param1;
    long Param2;
};
```

In MFC terms, this would be expressed as follows:

```
struct Message
{
    long m_lMagicNumber;
    short m_nCommand;
    short m_nParam1;
    long m_lParam2;

    void Serialize( CArchive& ar );
};
```

In C++, a **struct** is essentially the same thing as a class. The `Message` structure can have member functions, such as the `Serialize` member function declared above. The `Serialize` member function might look like this:

```
void Message::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar << (DWORD)htonl(m_lMagicNumber);
        ar << (WORD)htons(m_nCommand);
        ar << (WORD)htons(m_nParam1);
        ar << (DWORD)htonl(m_lParam2);
    }
    else
    {
        WORD w;
        DWORD dw;
        ar >> dw;
        m_lMagicNumber = ntohl((long)dw);
        ar >> w ;
        m_nCommand = ntohs((short)w);
        ar >> w;
        m_nParam1 = ntohs((short)w);
        ar >> dw;
        m_lParam2 = ntohl((long)dw);
    }
}
```

This example calls for byte-order conversions of data because there is a clear mismatch between the byte ordering of the non-MFC server application on one end and the **CArchive** used in your MFC client application on the other end. The example illustrates several of the byte-order conversion functions that Windows Sockets supplies. The following table describes these

functions.

Windows Sockets Byte-Order Conversion Functions

Function	Purpose
ntohs	Convert a 16-bit quantity from network byte order to host byte order (big-Endian to little-Endian).
ntohl	Convert a 32-bit quantity from network byte order to host byte order (big-Endian to little-Endian).
htons	Convert a 16-bit quantity from host byte order to network byte order (little-Endian to big-Endian).
htonl	Convert a 32-bit quantity from host byte order to network byte order (little-Endian to big-Endian).

Another point of this example is that when the socket application on the other end of the communication is a non-MFC application, you must avoid doing something like the following:

```
ar << pMsg;
```

where `pMsg` is a pointer to a C++ object derived from class **CObject**. This will send extra MFC information associated with objects and the server will not understand it, as it would if it were an MFC application.

For more information, see:

- Windows Sockets: Using Class CAsyncSocket
- Windows Sockets: Background
- Windows Sockets: Stream Sockets
- Windows Sockets: Datagram Sockets

See Also

Concepts

Windows Sockets in MFC

Windows Sockets: Converting Strings

This article and two companion articles explain several issues in Windows Sockets programming. This article covers converting strings. The other issues are covered in Windows Sockets: Blocking and Windows Sockets: Byte Ordering.

If you use or derive from class CAsyncSocket, you will need to manage these issues yourself. If you use or derive from class CSocket, MFC manages them for you.

Converting Strings

If you communicate between applications that use strings stored in different wide-character formats, such as Unicode or multibyte character sets (MBCS), or between one of these and an application using ANSI character strings, you must manage the conversions yourself under **CAsyncSocket**. The **CArchive** object used with a **CSocket** object manages this conversion for you through the capabilities of class CString. For more information, see the Windows Sockets specification, located in the Platform SDK.

For more information, see:

- Windows Sockets: Using Class CAsyncSocket
- Windows Sockets: Using Sockets with Archives
- Windows Sockets: Background
- Windows Sockets: Stream Sockets
- Windows Sockets: Datagram Sockets

See Also

Concepts

Windows Sockets in MFC

Windows Sockets: Ports and Socket Addresses

This article explains the terms "port" and "address" as used with Windows Sockets.

Port

A port identifies a unique process for which a service can be provided. In the present context, a port is associated with an application that supports Windows Sockets. The idea is to identify each Windows Sockets application uniquely so you can have more than one Windows Sockets application running on a machine at the same time.

Certain ports are reserved for common services, such as FTP. You should avoid using those ports unless you are providing that kind of service. The Windows Sockets specification details these reserved ports. The file WINSOCK.H also lists them.

To let the Windows Sockets DLL select a usable port for you, pass 0 as the port value. MFC selects a port value greater than 1,024 decimal. You can retrieve the port value that MFC selected by calling the CAsyncSocket::GetSockName member function.

Socket Address

Each socket object is associated with an Internet Protocol (IP) address on the network. Typically, the address is a machine name, such as "ftp.microsoft.com", or a dotted number, such as "128.56.22.8".

When you seek to create a socket, you typically do not need to specify your own address.

Note
It is possible that your machine has multiple network cards (or your application might someday run on such a machine), eac h representing a different network. If so, you might need to give an address to specify which network card the socket will use. This is certain to be an advanced usage and a possible portability issue.

For more information, see:

- Windows Sockets: Using Class CAsyncSocket
- Windows Sockets: Using Sockets with Archives
- Windows Sockets: How Sockets with Archives Work
- Windows Sockets: Stream Sockets
- Windows Sockets: Datagram Sockets

See Also

Concepts

Windows Sockets in MFC

Windows Sockets Sample List

The following topic describes Windows Sockets functionality:

- Windows Sockets

Hierarchy Chart

The following hierarchy chart represents the MFC library.

Microsoft Foundation Class Library Version 7.0

- **CObject**
  - **Application Architecture**
    - **CCmdTarget**
      - CWinThread
        - CWinApp
          - COleControlModule
            - user application
      - CDocTemplate
        - CSingleDocTemplate
        - CMultiDocTemplate
      - COleObjectFactory
        - COleTemplateServer
      - COleDataSource
      - COleDropSource
      - COleDropTarget
      - COleMessageFilter
      - CConnectionPoint
      - CDocument
        - CHtmlEditDoc
        - COleDocument
          - COleLinkingDoc
            - COleServerDoc
              - CRichEditDoc
        - user documents
      - CDocItem
        - COleClientItem
          - COleDocObjectItem
          - CRichEditCntrItem
          - user client items
        - COleServerItem
          - CDocObjectServerItem
          - user server items
      - CDocObjectServer
      - COleControlContainer
      - COleControlSite
      - user objects
  - **Exceptions**
    - CException
      - CArchiveException
      - CDaoException
      - CDBException
      - CFileException
      - CInternetException
      - COleException
      - COleDispatchException
      - CSimpleException
        - CMemoryException
        - CNotSupportedException
        - CResourceException
        - CUserException
  - **File Services**
    - CFile
      - CMemFile
        - CSharedFile
      - COleStreamFile
        - CMonikerFile
          - CAsyncMonikerFile
            - CDataPathProperty
              - CCachedDataPathProperty
      - CSocketFile
      - CStdioFile
        - CInternetFile
          - CGopherFile
          - CHttpFile
  - **Window Support**
    - **CWnd**
      - **Frame Windows**
        - CFrameWnd
          - CMDIChildWnd
            - user MDI windows
          - CMDIFrameWnd
            - user MDI workspaces
          - CMiniFrameWnd
            - user SDI windows
          - COleIPFrameWnd
        - CSplitterWnd
      - **Control Bars**
        - CControlBar
          - CDialogBar
          - COleResizeBar
          - CReBar
          - CStatusBar
          - CToolBar
      - **Property Sheets**
        - CPropertySheet
      - **Dialog Boxes**
        - CDialog
          - CCommonDialog
            - CColorDialog
            - CFileDialog
            - CFindReplaceDialog
            - CFontDialog
            - COleDialog
              - COleBusyDialog
              - COleChangeIconDialog
              - COleChangeSourceDialog
              - COleConvertDialog
              - COleInsertDialog
              - COleLinksDialog
                - COleUpdateDialog
              - COlePasteSpecialDialog
              - COlePropertiesDialog
            - CPageSetupDialog
            - CPrintDialog
            - CPrintDialogEx
          - COlePropertyPage
          - CPropertyPage
          - user dialog boxes
          - CDHtmlDialog
            - CMultiPageDHtmlDialog
      - **Views**
        - CView
          - CCtrlView
            - CEditView
            - CListView
            - CRichEditView
            - CTreeView
          - CScrollView
            - user scroll views
            - CFormView
              - user form views
              - CDaoRecordView
              - CHtmlEditView
              - CHtmlView
              - COleDBRecordView
              - CRecordView
                - user record views
      - **Controls**
        - CAnimateCtrl
        - CButton
          - CBitmapButton
        - CComboBox
          - CComboBoxEx
        - CDateTimeCtrl
        - CEdit
        - CHeaderCtrl
        - CHtmlEditCtrlBase
        - CHtmlEditCtrl
        - CHotKeyCtrl
        - CIPAddressCtrl
        - CLinkCtrl
        - CListBox
          - CCheckListBox
          - CDragListBox
        - CListCtrl
        - CMonthCalCtrl
        - COleControl
        - CProgressCtrl
        - CReBarCtrl
        - CRichEditCtrl
        - CScrollBar
        - CSliderCtrl
        - CSpinButtonCtrl
        - CStatic
        - CStatusBarCtrl
        - CTabCtrl
        - CToolBarCtrl
        - CToolTipCtrl
        - CTreeCtrl
  - **Graphical Drawing**
    - CDC
      - CClientDC
      - CMetaFileDC
      - CPaintDC
      - CWindowDC
  - **Control Support**
    - CDockState
    - CImageList
  - **Graphical Drawing Objects**
    - CGdiObject
      - CBitmap
      - CBrush
      - CFont
      - CPalette
      - CPen
      - CRgn
  - **Menus**
    - CMenu
  - **Command Line**
    - CCommandLineInfo
  - **ODBC Database Support**
    - CDatabase
    - CRecordset
      - user recordsets
    - CLongBinary
  - **DAO Database Support**
    - CDaoDatabase
    - CDaoQueryDef
    - CDaoRecordset
    - CDaoTableDef
    - CDaoWorkspace
  - **Synchronization**
    - CSyncObject
      - CCriticalSection
      - CEvent
      - CMutex
      - CSemaphore
  - **Windows Sockets**
    - CAsyncSocket
      - CSocket
  - **Arrays**
    - CArray (template)
    - CByteArray
    - CDWordArray
    - CObArray
    - CPtrArray
    - CStringArray
    - CUIntArray
    - CWordArray
    - arrays of user types
  - **Lists**
    - CList (template)
    - CPtrList
    - CObList
    - CStringList
    - lists of user types
  - **Maps**
    - CMap (template)
    - CMapWordToPtr
    - CMapPtrToWord
    - CMapPtrToPtr
    - CMapWordToOb
    - CMapStringToPtr
    - CMapStringToOb
    - CMapStringToString
    - maps of user types
  - **Internet Services**
    - CInternetSession
    - CInternetConnection
      - CFtpConnection
      - CGopherConnection
      - CHttpConnection
    - CFileFind
      - CFtpFileFind
      - CGopherFileFind
    - CGopherLocator
- **Classes Not Derived from CObject**
  - **Internet Server API**
    - CHttpArgList
    - CHtmlStream
    - CHttpFilter
    - CHttpFilterContext
    - CHttpServer
    - CHttpServerContext
  - **Run-Time Object Model Support**
    - CArchive
    - CDumpContext
    - CRuntimeClass
  - **Structures**
    - CCreateContext
    - CHttpArg
    - CMemoryState
    - COleSafeArray
    - CPrintInfo
  - **Simple Value Types**
    - CFileTime
    - CFileTimeSpan
    - CPoint
    - CRect
    - CSize
    - CSimpleStringT
      - CStringT
        - CFixedStringT
    - CTime
    - CTimeSpan
  - **Support Classes**
    - CCmdUI
      - COleCmdUI
    - CDaoFieldExchange
    - CDataExchange
    - CDBVariant
    - CFieldExchange
    - CImage
    - COccManager
    - COleDataObject
    - COleDispatchDriver
    - CPropExchange
    - CRecentFileList
    - CRectTracker
    - CWaitCursor
  - **Typed Template Collections**
    - CTypedPtrArray
    - CTypedPtrList
    - CTypedPtrMap
  - **OLE Type Wrappers**
    - CFontHolder
    - CPictureHolder
  - **OLE Automation Types**
    - COleCurrency
    - COleDateTime
    - COleDateTimeSpan
    - COleVariant
  - **Synchronization**
    - CMultiLock
    - CSingleLock

See Also

Concepts

Class Library Overview

Other Resources

Hierarchy Chart Categories

Hierarchy Chart Categories

Microsoft Foundation Class Library Version 7.0

- **CObject**
  - Application Architecture
  - Exceptions
  - File Services
  - **CCmdTarget**
    - Window Support
    - **CWnd**
      - Frame Windows
      - Dialog Boxes
      - Views
      - Controls
      - Control Bars
      - Property Sheets
  - Graphical Drawing
  - Control Support
  - Graphical Drawing Objects
  - Menus
  - Command Line
  - ODBC Database Support
  - DAO Database Support
  - Synchronization
  - Windows Sockets
  - Arrays
  - Lists
  - Maps
  - Internet Services

Classes Not Derived from CObject

Internet Server API

Run-Time Object Model Support

Structures

Simple Value Types

Support Classes

Typed Template Collections

OLE Type Wrappers

OLE Automation Types

Synchronization

See Also

Reference

Hierarchy Chart

Other Resources

MFC Reference

Class Library Overview

This overview categorizes and describes the classes in the Microsoft Foundation Class Library (MFC) version 8.0. The classes in MFC, taken together, constitute an application framework — the framework of an application written for the Windows API. Your programming task is to fill in the code that is specific to your application.

The library's classes are presented here in the following categories:

- Root Class: CObject
- MFC Application Architecture Classes
  - Application and Thread Support Classes
  - Command Routing Classes
  - Document Classes
  - View Classes (Architecture)
  - Frame Window Classes (Architecture)
  - Document-Template Classes
- Window, Dialog, and Control Classes
  - Frame Window Classes (Windows)
  - View Classes (Windows)
  - Dialog Box Classes
  - Control Classes
  - Control Bar Classes
- Drawing and Printing Classes
  - Output (Device Context) Classes
  - Drawing Tool Classes
- Simple Data Type Classes
- Array, List, and Map Classes
  - Template Classes for Arrays, Lists, and Maps
  - Ready-to-Use Array Classes
  - Ready-to-Use List Classes
  - Ready-to-Use Map Classes
- File and Database Classes
  - File I/O Classes
  - DAO Classes
  - ODBC Classes
  - OLE DB Classes
- Internet and Networking Classes
  - ISAPI Classes
  - Windows Sockets Classes
  - Win32 Internet Classes
- OLE Classes

    - OLE Container Classes
    - OLE Server Classes
    - OLE Drag-and-Drop and Data Transfer Classes
    - OLE Common Dialog Classes
    - OLE Automation Classes
    - OLE Control Classes
    - Active Document Classes
    - OLE-Related Classes
- Debugging and Exception Classes
    - Debugging Support Classes
    - Exception Classes

The section General Class Design Philosophy explains how the MFC Library was designed.

For an overview of the framework, see Using the Classes to Write Applications for Windows. Some of the classes listed above are general-purpose classes that can be used outside of the framework and provide useful abstractions such as collections, exceptions, files, and strings.

To see the inheritance of a class, use the Class Hierarchy Chart.

In addition to the classes listed in this overview, the MFC Library contains a number of global functions, global variables, and macros. There is an overview and detailed listing of these in the topic MFC Macros and Globals, which follows the alphabetical reference to the MFC classes.

See Also

Other Resources

MFC Reference

General Class Design Philosophy

Microsoft Windows was designed long before the C++ language became popular. Because thousands of applications use the C-language Windows application programming interface (API), that interface will be maintained for the foreseeable future. Any C++ Windows interface must therefore be built on top of the procedural C-language API. This guarantees that C++ applications will be able to coexist with C applications.

The Microsoft Foundation Class Library is an object-oriented interface to Windows that meets the following design goals:

- Significant reduction in the effort to write an application for Windows.
- Execution speed comparable to that of the C-language API.
- Minimum code size overhead.
- Ability to call any Windows C function directly.
- Easier conversion of existing C applications to C++.
- Ability to leverage from the existing base of C-language Windows programming experience.
- Easier use of the Windows API with C++ than with C.
- Easier to use yet powerful abstractions of complicated features such as ActiveX controls, database support, printing, toolbars, and status bars.
- True Windows API for C++ that effectively uses C++ language features.

For more on the design of the MFC Library, see:

- The Application Framework
- Relationship to the C-Language API

See Also

Concepts

Class Library Overview

The Application Framework

The core of the Microsoft Foundation Class (MFC) Library is an encapsulation of a large portion of the Windows API in C++ form. Library classes represent windows, dialog boxes, device contexts, common GDI objects such as brushes and pens, controls, and other standard Windows items. These classes provide a convenient C++ member function interface to the structures in Windows that they encapsulate. For more about using these classes, see Window Object Topics.

But the MFC Library also supplies a layer of additional application functionality built on the C++ encapsulation of the Windows API. This layer is a working application framework for Windows that provides most of the common user interface expected of programs for Windows, including toolbars, status bars, printing, print preview, database support, and ActiveX support. Using the Classes to Write Applications for Windows explains the framework in detail.

See Also

Concepts

General Class Design Philosophy

Relationship to the C-Language API

The single characteristic that sets the Microsoft Foundation Class (MFC) Library apart from other class libraries for Windows is the very close mapping to the Windows API written in the C language. Further, you can generally mix calls to the class library freely with direct calls to the Windows API. This direct access does not, however, imply that the classes are a complete replacement for that API. Developers must still occasionally make direct calls to some Windows functions, such as SetCursor and GetSystemMetrics, for example. A Windows function is wrapped by a class member function only when there is a clear advantage to doing so.

Because you sometimes need to make native Windows function calls, you should have access to the C-language Windows API documentation. This documentation is included with Microsoft Visual C++.

Note
For an overview of how the MFC Library framework operates, see Using the Classes to Write Applications for Windows.

See Also

Concepts

General Class Design Philosophy

Root Class: CObject

Most of the classes in the Microsoft Foundation Class (MFC) Library are derived from a single base class at the root of the class hierarchy. **CObject** provides a number of useful capabilities to all classes derived from it, with very low overhead. For more information about **CObject** and its capabilities, see Using CObject.

CObject

The ultimate base class of most MFC classes. Supports serializing data and obtaining run-time information about a class.

CRuntimeClass

Structure used to determine the exact class of an object at run time.

See Also

Concepts

Class Library Overview

MFC Application Architecture Classes

Classes in this category contribute to the architecture of a framework application. They supply functionality common to most applications. You fill in the framework to add application-specific functionality. Typically, you do so by deriving new classes from the architecture classes, and then adding new members or overriding existing member functions.

Application wizards generate several types of applications, all of which use the application framework in differing ways. SDI (single document interface) and MDI (multiple document interface) applications make full use of a part of the framework called document/view architecture. Other types of applications, such as dialog-based applications, form-based applications, and DLLs, use only some of document/view architecture features.

Document/view applications contain one or more sets of documents, views, and frame windows. A document-template object associates the classes for each document/view/frame set.

Although you do not have to use document/view architecture in your MFC application, there are a number of advantages to doing so. The MFC OLE container and server support is based on document/view architecture, as is support for printing and print preview.

All MFC applications have at least two objects: an application object derived from CWinApp, and some sort of main window object, derived (often indirectly) from CWnd. (Most often, the main window is derived from CFrameWnd, CMDIFrameWnd, or CDialog, all of which are derived from **CWnd**.)

Applications that use document/view architecture contain additional objects. The principal objects are:

- An application object derived from class CWinApp, as mentioned before.
- One or more document class objects derived from class CDocument. Document class objects are responsible for the internal representation of the data manipulated in the view. They may be associated with a data file.
- One or more view objects derived from class CView. Each view is a window that is attached to a document and associated with a frame window. Views display and manipulate the data contained in a document class object.

Document/view applications also contain frame windows (derived from CFrameWnd) and document templates (derived from CDocTemplate).

See Also

Concepts

Class Library Overview

Application and Thread Support Classes

Each application has one and only one application object; this object coordinates other objects in the running program and is derived from **CWinApp**.

The Microsoft Foundation Class (MFC) Library supports multiple threads of execution within an application. All applications must have at least one thread; the thread used by your **CWinApp** object is this primary thread.

CWinThread encapsulates a portion of the operating system's threading capabilities. To make using multiple threads easier, MFC also provides synchronization object classes to provide a C++ interface to Win32 synchronization objects.

Application and Thread Classes

CWinApp

Encapsulates the code to initialize, run, and terminate the application. You will derive your application object from this class.

CWinThread

The base class for all threads. Use directly, or derive a class from **CWinThread** if your thread performs user-interface functions. **CWinApp** is derived from **CWinThread**.

ISAPI Application Classes

CHttpFilter

Filters selected HTTP requests sent to an ISAPI server.

CHttpServer

Extends the functionality of an ISAPI server by processing client requests.

Synchronization Object Classes

CSyncObject

Base class of the synchronization object classes.

CCriticalSection

A synchronization class that allows only one thread within a single process to access an object.

CSemaphore

A synchronization class that allows between one and a specified maximum number of simultaneous accesses to an object.

CMutex

A synchronization class that allows only one thread within any number of processes to access an object.

CEvent

A synchronization class that notifies an application when an event has occurred.

CSingleLock

Used in member functions of thread-safe classes to lock on one synchronization object.

CMultiLock

Used in member functions of thread-safe classes to lock on one or more synchronization objects from an array of synchronization objects.

Related Classes

CCommandLineInfo

Parses the command line with which your program was started.

CWaitCursor

Puts a wait cursor on the screen. Used during lengthy operations.

CDockState

Handles persistent storage of docking state data for control bars.

CRecentFileList

Maintains the most recently used (MRU) file list.

See Also

Concepts

Class Library Overview

Command Routing Classes

As the user interacts with the application by choosing menus or control-bar buttons with the mouse, the application sends messages from the affected user-interface object to an appropriate command-target object. Command-target classes derived from **CCmdTarget** include CWinApp, CWnd, CDocTemplate, CDocument, CView, and the classes derived from them. The framework supports automatic command routing so that commands can be handled by the most appropriate object currently active in the application.

An object of class **CCmdUI** is passed to your command targets' update command UI (ON_UPDATE_COMMAND_UI) handlers to allow you to update the state of the user interface for a particular command (for instance, to check or remove the check from menu items). You call member functions of the **CCmdUI** object to update the state of the UI object. This process is the same whether the UI object associated with a particular command is a menu item or a button or both.

CCmdTarget

Serves as the base class for all classes of objects that can receive and respond to messages.

CCmdUI

Provides a programmatic interface for updating user-interface objects such as menu items or control-bar buttons. The command target object enables, disables, checks, and/or clears the user-interface object with this object.

See Also

Concepts

Class Library Overview

Document Classes

Document class objects, created by document-template objects, manage the application's data. You will derive a class for your documents from one of these classes.

Document class objects interact with view objects. View objects represent the client area of a window, display a document's data, and allow users to interact with it. Documents and views are created by a document-template object.

CDocument

The base class for application-specific documents. Derive your document class or classes from **CDocument**.

COleDocument

Used for compound document implementation, as well as basic container support. Serves as a container for classes derived from CDocItem. This class can be used as the base class for container documents and is the base class for **COleServerDoc**.

COleLinkingDoc

A class derived from **COleDocument** that provides the infrastructure for linking. You should derive the document classes for your container applications from this class instead of from **COleDocument** if you want them to support links to embedded objects.

CRichEditDoc

Maintains the list of OLE client items that are in the rich edit control. Used with CRichEditView and CRichEditCntrItem.

COleServerDoc

Used as the base class for server-application document classes. **COleServerDoc** objects provide the bulk of server support through interactions with COleServerItem objects. Visual editing capability is provided using the class library's document/view architecture.

CHtmlEditDoc

Provides, with CHtmlEditView, the functionality of the WebBrowser HTML editing platform within the context of the MFC document-view architecture.

Related Classes

Document class objects can be persistent — in other words, they can write their state to a storage medium and read it back. MFC provides the **CArchive** class to facilitate transferring the document's data to a storage medium.

CArchive

Cooperates with a CFile object to implement persistent storage for objects through serialization (see CObject::Serialize).

Documents can also contain OLE objects. **CDocItem** is the base class of the server and client items.

CDocItem

Abstract base class of COleClientItem and COleServerItem. Objects of classes derived from **CDocItem** represent parts of documents.

See Also

Concepts

Class Library Overview

View Classes (Architecture)

CView and its derived classes are child windows that represent the client area of a frame window. Views show data and accept input for a document.

A view class is associated with a document class and a frame window class using a document-template object.

CView

The base class for application-specific views of a document's data. Views display data and accept user input to edit or select the data. Derive your view class(es) from **CView**.

CScrollView

The base class for views with scrolling capabilities. Derive your view class from **CScrollView** for automatic scrolling.

Form and Record Views

Form views are also scrolling views. They are based on a dialog box template.

Record views are derived from form views. In addition to the dialog box template, they also have a connection to a database.

CFormView

A scroll view whose layout is defined in a dialog box template. Derive a class from **CFormView** to implement a user interface based on a dialog box template.

CDaoRecordView

Provides a form view directly connected to a Data Access Object (DAO) recordset object. Like all form views, a **CDaoRecordView** is based on a dialog box template.

CHtmlView

Supports a control for Web browsing within an application. The control supports dynamic HTML in MFC.

COLEDBRecordView

Provides MFC OLE DB support for form views.

CRecordView

Provides a form view directly connected to an Open Database Connectivity (ODBC) recordset object. Like all form views, a **CRecordView** is based on a dialog box template.

Control Views

Control views display a control as their view.

CCtrlView

The base class for all views associated with Windows controls. The views based on controls are described below.

CEditView

A view that contains a Windows standard edit control (see CEdit). Edit controls support text editing, searching, replacing, and scrolling capabilities.

CRichEditView

A view that contains a Windows rich edit control (see CRichEditCtrl). In addition to the capabilities of an edit control, rich edit controls support fonts, colors, paragraph formatting, and embedded OLE objects.

CListView

A view that contains a Windows list control (see CListCtrl). A list control displays icons and strings in a manner similar to the right pane of Windows Explorer.

CTreeView

A view that contains a Windows tree control (see CTreeCtrl). A tree control displays icons and strings arranged in a hierarchy in a manner similar to the left pane of Windows Explorer.

See Also

Concepts

Class Library Overview

Frame Window Classes (Architecture)

In document/view architecture, frame windows are windows that contain a view window. They also support having control bars attached to them.

In multiple document interface (MDI) applications, the main window is derived from **CMDIFrameWnd**. It indirectly contains the documents' frames, which are **CMDIChildWnd** objects. The **CMDIChildWnd** objects, in turn, contain the documents' views.

In single document interface (SDI) applications, the main window, derived from **CFrameWnd**, contains the view of the current document.

CFrameWnd

The base class for an SDI application's main frame window. Also the base class for all other frame window classes.

CMDIFrameWnd

The base class for an MDI application's main frame window.

CMDIChildWnd

The base class for an MDI application's document frame windows.

COleIPFrameWnd

Provides the frame window for a view when a server document is being edited in place.

See Also

Concepts

Class Library Overview

Document-Template Classes

Document-template objects coordinate the creation of document, view, and frame window objects when a new document or view is created.

CDocTemplate

The base class for document templates. You will never use this class directly; instead, you use one of the other document-template classes derived from this class.

CMultiDocTemplate

A template for documents in the multiple document interface (MDI). MDI applications can have multiple documents open at a time.

CSingleDocTemplate

A template for documents in the single document interface (SDI). SDI applications have only one document open at a time.

Related Class

CCreateContext

A structure passed by a document template to window-creation functions to coordinate the creation of document, view, and frame-window objects.

See Also

Concepts

Class Library Overview

Window, Dialog, and Control Classes

Class **CWnd** and its derived classes encapsulate an **HWND**, a handle to a Windows window. **CWnd** can be used by itself or as a base for deriving new classes. The derived classes supplied by the class library represent various kinds of windows.

CWnd

The base class for all windows. You can use one of the classes derived from **CWnd** or derive your own classes directly from it.

See Also

Concepts

Class Library Overview

Frame Window Classes (Windows)

Frame windows are windows that frame an application or a part of an application. Frame windows usually contain other windows, such as views, tool bars, and status bars. In the case of **CMDIFrameWnd**, they may contain **CMDIChildWnd** objects indirectly.

CFrameWnd

The base class for an SDI application's main frame window. Also the base class for all other frame window classes.

CMDIFrameWnd

The base class for an MDI application's main frame window.

CMDIChildWnd

The base class for an MDI application's document frame windows.

CMiniFrameWnd

A half-height frame window typically seen around floating toolbars.

COleIPFrameWnd

Provides the frame window for a view when a server document is being edited in place.

Related Class

Class **CMenu** provides an interface through which to access your application's menus. It is useful for manipulating menus dynamically at run time; for example, when adding or deleting menu items according to context. Although menus are most often used with frame windows, they can also be used with dialog boxes and other nonchild windows.

CMenu

Encapsulates an **HMENU** handle to the application's menu bar and pop-up menus.

See Also

Concepts

Class Library Overview

View Classes (Windows)

CView and its derived classes are child windows that represent the client area of a frame window. Views show data and accept input for a document.

A view class is associated with a document class and a frame window class using a document-template object.

CView

The base class for application-specific views of a document's data. Views display data and accept user input to edit or select the data. Derive your view class or classes from **CView**.

CScrollView

The base class for views with scrolling capabilities. Derive your view class from **CScrollView** for automatic scrolling.

Form and Record Views

Form views are also scrolling views. They are based on a dialog box template.

Record views are derived from form views. In addition to the dialog box template, they also have a connection to a database.

CFormView

A scroll view whose layout is defined in a dialog box template. Derive a class from **CFormView** to implement a user interface based on a dialog box template.

CDaoRecordView

Provides a form view directly connected to a Data Access Object (DAO) recordset object. Like all form views, a **CDaoRecordView** is based on a dialog box template.

CRecordView

Provides a form view directly connected to an Open Database Connectivity (ODBC) recordset object. Like all form views, a **CRecordView** is based on a dialog box template.

CHtmlEditView

A form view that provides the functionality of the WebBrowser HTML editing platform.

Control Views

Control views display a control as their view.

CCtrlView

The base class for all views associated with Windows controls. The views based on controls are described below.

CEditView

A view that contains a Windows standard edit control (see CEdit). Edit controls support text editing, searching, replacing, and scrolling capabilities.

CRichEditView

A view that contains a Windows rich edit control (see CRichEditCtrl). In addition to the capabilities of an edit control, rich edit controls support fonts, colors, paragraph formatting, and embedded OLE objects.

CListView

A view that contains a Windows list control (see CListCtrl). A list control displays a collection of items, each consisting of an icon and a label, in a manner similar to the right pane of Windows Explorer.

CTreeView

A view that contains a Windows tree control (see CTreeCtrl). A tree control displays a hierarchical list of icons and labels arranged in a manner similar to the left pane of Windows Explorer.

Related Classes

CSplitterWnd allows you to have multiple views within a single frame window. **CPrintDialog** and **CPrintInfo** support the

print and print preview ability of views. **CRichEditDoc** and **CRichEditCntrItem** are used with **CRichEditView** to implement OLE container capabilities.

CSplitterWnd

A window that the user can split into multiple panes. These panes can be resizable by the user or fixed size.

CPrintDialog

Provides a standard dialog box for printing a file.

CPrintInfo

A structure containing information about a print or print preview job. Used by **CView**'s printing architecture.

CRichEditDoc

Maintains the list of OLE client items that are in a **CRichEditView**.

CRichEditCntrItem

Provides client-side access to an OLE item stored in a **CRichEditView**.

See Also

Concepts

Class Library Overview

Dialog Box Classes

Class **CDialog** and its derived classes encapsulate dialog-box functionality. Since a dialog box is a special kind of window, **CDialog** is derived from **CWnd**. Derive your dialog classes from **CDialog** or use one of the common dialog classes for standard dialog boxes, such as opening or saving a file, printing, selecting a font or color, initiating a search-and-replace operation, or performing various OLE-related operations.

CDialog

The base class for all dialog boxes, both modal and modeless.

CDataExchange

Supplies data exchange and validation information for dialog boxes.

Common Dialogs

These dialog box classes encapsulate the Windows common dialog boxes. They provide easy-to-use implementations of complicated dialog boxes.

CCommonDialog

Base class for all common dialog boxes.

CFileDialog

Provides a standard dialog box for opening or saving a file.

CColorDialog

Provides a standard dialog box for selecting a color.

CFontDialog

Provides a standard dialog box for selecting a font.

CFindReplaceDialog

Provides a standard dialog box for a search-and-replace operation.

CPrintDialog

Provides a standard dialog box for printing a file.

CPrintDialogEx

Provides a Windows 2000 Print property sheet.

CPageSetupDialog

Encapsulates the services provided by the Windows common Page Setup dialog box with additional support for setting and modifying print margins.

OLE Common Dialogs

OLE adds several common dialog boxes to Windows. These classes encapsulate the OLE common dialog boxes.

COleDialog

Used by the framework to contain common implementations for all OLE dialog boxes. All dialog box classes in the user-interface category are derived from this base class. **COleDialog** cannot be used directly.

COleInsertDialog

Displays the Insert Object dialog box, the standard user interface for inserting new OLE linked or embedded items.

COlePasteSpecialDialog

Displays the Paste Special dialog box, the standard user interface for implementing the Edit Paste Special command.

COleLinksDialog

Displays the Edit Links dialog box, the standard user interface for modifying information about linked items.

COleChangeIconDialog

Displays the Change Icon dialog box, the standard user interface for changing the icon associated with an OLE embedded or linked item.

COleConvertDialog

Displays the Convert dialog box, the standard user interface for converting OLE items from one type to another.

COlePropertiesDialog

Encapsulates the Windows common OLE Properties dialog box. Common OLE Properties dialog boxes provide an easy way to display and modify the properties of an OLE document item in a manner consistent with Windows standards.

COleUpdateDialog

Displays the Update dialog box, the standard user interface for updating all links in a document. The dialog box contains a progress indicator to indicate how close the update procedure is to completion.

COleChangeSourceDialog

Displays the Change Source dialog box, the standard user interface for changing the destination or source of a link.

COleBusyDialog

Displays the Server Busy and Server Not Responding dialog boxes, the standard user interface for handling calls to busy applications. Usually displayed automatically by the COleMessageFilter implementation.

Property Sheet Classes

The property sheet classes allow your applications to use property sheets, also known as tabbed dialogs. Property sheets are an efficient way to organize a large number of controls in a single dialog box.

CPropertyPage

Provides the individual pages within a property sheet. Derive a class from **CPropertyPage** for each page to be added to your property sheet.

CPropertySheet

Provides the frame for multiple property pages. Derive your property sheet class from **CPropertySheet** to implement your property sheets quickly.

COlePropertyPage

Displays the properties of an OLE control in a graphical interface, similar to a dialog box.

HTML-based Dialog Classes

CDHtmlDialog

Used to create dialog boxes that implement their user interface with HTML rather than dialog resources.

CMultiPageDHtmlDialog

Displays multiple HTML pages sequentially and handles the events from each page.

Related Classes

These classes are not dialog boxes per se, but they use dialog box templates and have much of the behavior of dialog boxes.

CDialogBar

A control bar that is based on a dialog box template.

CFormView

A scroll view whose layout is defined in a dialog box template. Derive a class from **CFormView** to implement a user interface based on a dialog box template.

CDaoRecordView

Provides a form view directly connected to a Data Access Object (DAO) recordset object. Like all form views, a **CDaoRecordView** is based on a dialog box template.

CRecordView

Provides a form view directly connected to an Open Database Connectivity (ODBC) recordset object. Like all form views, a **CRecordView** is based on a dialog box template.

CPrintInfo

A structure containing information about a print or print preview job. Used by the printing architecture of CView.

See Also

Concepts

Class Library Overview

Control Classes

Control classes encapsulate a wide variety of standard Windows controls ranging from static text controls to tree controls. In addition, MFC provides some new controls, including buttons with bitmaps and control bars.

The controls whose class names end in "**Ctrl**" were new in Windows 95 and Windows NT version 3.51.

Static Display Controls

CStatic

A static-display window. Static controls are used to label, box, or separate other controls in a dialog box or window. They may also display graphical images rather than text or a box.

Text Controls

CEdit

An editable-text control window. Edit controls are used to accept textual input from the user.

CIPAddressCtrl

Supports an edit box for manipulating an Internet Protocol (IP) address.

CRichEditCtrl

A control in which the user can enter and edit text. Unlike the control encapsulated in **CEdit**, a rich edit control supports character and paragraph formatting and OLE objects.

Controls That Represent Numbers

CSliderCtrl

A control containing a slider, which the user moves to select a value or set of values.

CSpinButtonCtrl

A pair of arrow buttons the user can click to increment or decrement a value.

CProgressCtrl

Displays a rectangle that is gradually filled from left to right to indicate the progress of an operation.

CScrollBar

A scroll-bar control window. The class provides the functionality of a scroll bar, for use as a control in a dialog box or window, through which the user can specify a position within a range.

Buttons

CButton

A button control window. The class provides a programmatic interface for a push button, check box, or radio button in a dialog box or window.

CBitmapButton

A button with a bitmap rather than a text caption.

Lists

CListBox

A list-box control window. A list box displays a list of items that the user can view and select.

CDragListBox

Provides the functionality of a Windows list box; allows the user to move list box items, such as filenames and string literals, within the list box. List boxes with this capability are useful for an item list in an order other than alphabetical, such as include pathnames or files in a project.

CComboBox

A combo-box control window. A combo box consists of an edit control plus a list box.

CComboBoxEx

Extends the combo box control by providing support for image lists.

CCheckListBox

Displays a list of items with check boxes, which the user can check or clear, next to each item.

CListCtrl

Displays a collection of items, each consisting of an icon and a label, in a manner similar to the right pane of Windows Explorer.

CTreeCtrl

Displays a hierarchical list of icons and labels arranged in a manner similar to the left pane of Windows Explorer.

Toolbars and Status Bars

CToolBarCtrl

Provides the functionality of the Windows toolbar common control. Most MFC programs use CToolBar instead of this class.

CStatusBarCtrl

A horizontal window, usually divided into panes, in which an application can display status information. Most MFC programs use CStatusBar instead of this class.

Miscellaneous Controls

CAnimateCtrl

Displays a simple video clip.

CToolTipCtrl

A small pop-up window that displays a single line of text describing the purpose of a tool in an application.

CDateTimeCtrl

Supports either an extended edit control, or a simple calendar interface control, that allows a user to choose a specific date or time value.

CHeaderCtrl

Displays titles or labels for columns.

CMonthCalCtrl

Supports a simple calendar interface control that allows a user to select a date.

CTabCtrl

A control with tabs on which the user can click, analogous to the dividers in a notebook.

CHotKeyCtrl

Enables the user to create a hot key combination, which the user can press to perform an action quickly.

CLinkCtrl

Renders marked-up text and launches appropriate applications when the user clicks the embedded link.

CHtmlEditCtrl

Provides the functionality of the WebBrowser ActiveX control in an MFC window.

Related Classes

CImageList

Provides the functionality of the Windows image list. Image lists are used with list controls and tree controls. They can also be used to store and archive a set of same-sized bitmaps.

CCtrlView

The base class for all views associated with Windows controls. The views based on controls are described below.

CEditView

A view that contains a Windows standard edit control.

CRichEditView

A view that contains a Windows rich edit control.

CListView

A view that contains a Windows list control.

CTreeView

A view that contains a Windows tree control.

See Also

Concepts

Class Library Overview

Control Bar Classes

Control bars are attached to a frame window. They contain buttons, status panes, or a dialog template. Free-floating control bars, also called tool palettes, are implemented by attaching them to a CMiniFrameWnd object.

Framework Control Bars

These control bars are an integral part of the MFC framework. They are easier to use and more powerful than the Windows control bars because they are integrated with the framework. Most MFC applications use these control bars rather than the Windows control bars.

CControlBar

The base class for MFC control bars listed in this section. A control bar is a window aligned to the edge of a frame window. The control bar contains either **HWND**-based child controls or controls not based on an **HWND**, such as toolbar buttons.

CDialogBar

A control bar that is based on a dialog box template.

CReBar

Supports a toolbar that can contain additional child windows in the form of controls.

CToolBar

Toolbar control windows that contain bitmap command buttons not based on an **HWND**. Most MFC applications use this class rather than **CToolBarCtrl**.

CStatusBar

The base class for status-bar control windows. Most MFC applications use this class rather than **CStatusBarCtrl**.

Windows Control Bars

These control bars are thin wrappers for the corresponding Windows controls. Because they are not integrated with the framework, they are harder to use than the control bars previously listed. Most MFC applications use the control bars previously listed.

CRebarCtrl

Implements the internal control of the **CRebar** object.

CStatusBarCtrl

A horizontal window, usually divided into panes, in which an application can display status information.

CToolBarCtrl

Provides the functionality of the Windows toolbar common control.

Related Classes

CToolTipCtrl

A small pop-up window that displays a single line of text describing the purpose of a tool in an application.

CDockState

Handles persistent storage of docking state data for control bars.

See Also

Concepts

Class Library Overview

Drawing and Printing Classes

In Windows, all graphical output is drawn on a virtual drawing area called a device context (DC). MFC provides classes to encapsulate the various types of DCs, as well as encapsulations for Windows drawing tools such as bitmaps, brushes, palettes, and pens.

See Also

Concepts

Class Library Overview

Output (Device Context) Classes

These classes encapsulate the different types of device contexts available in Windows.

Most of the following classes encapsulate a handle to a Windows device context. A device context is a Windows object that contains information about the drawing attributes of a device such as a display or a printer. All drawing calls are made through a device-context object. Additional classes derived from **CDC** encapsulate specialized device-context functionality, including support for Windows metafiles.

CDC

The base class for device contexts. Used directly for accessing the whole display and for accessing nondisplay contexts such as printers.

CPaintDC

A display context used in **OnPaint** member functions of windows. Automatically calls **BeginPaint** on construction and **EndPaint** on destruction.

CClientDC

A display context for client areas of windows. Used, for example, to draw in an immediate response to mouse events.

CWindowDC

A display context for entire windows, including both the client and nonclient areas.

CMetaFileDC

A device context for Windows metafiles. A Windows metafile contains a sequence of graphics device interface (GDI) commands that can be replayed to create an image. Calls made to the member functions of a **CMetaFileDC** are recorded in a metafile.

Related Classes

CPoint

Holds coordinate (x, y) pairs.

CSize

Holds distance, relative positions, or paired values.

CRect

Holds coordinates of rectangular areas.

CRgn

Encapsulates a GDI region for manipulating an elliptical, polygonal, or irregular area within a window. Used in conjunction with the clipping member functions in class **CDC**.

CRectTracker

Displays and handles the user interface for resizing and moving rectangular objects.

CColorDialog

Provides a standard dialog box for selecting a color.

CFontDialog

Provides a standard dialog box for selecting a font.

CPrintDialog

Provides a standard dialog box for printing a file.

See Also

Concepts

Class Library Overview

Drawing Tool Classes

These classes encapsulate drawing tools that are used to draw on a device context.

CGdiObject

The base class for GDI drawing tools.

CBrush

Encapsulates a GDI brush that can be selected as the current brush in a device context. Brushes are used for filling interiors of objects being drawn.

CPen

Encapsulates a GDI pen that can be selected as the current pen in a device context. Pens are used for drawing the border lines of objects.

CFont

Encapsulates a GDI font that can be selected as the current font in a device context.

CBitmap

Encapsulates a GDI bitmap, providing an interface for manipulating bitmaps.

CPalette

Encapsulates a GDI color palette for use as an interface between the application and a color output device such as a display.

CRectTracker

Displays and handles the user interface for resizing and moving rectangular objects.

See Also

Concepts

Class Library Overview

Simple Data Type Classes

The following classes encapsulate drawing coordinates, character strings, and time and date information, allowing convenient use of C++ syntax. These objects are used widely as parameters to the member functions of Windows classes in the class library. Because **CPoint**, **CSize**, and **CRect** correspond to the **POINT**, **SIZE**, and **RECT** structures, respectively, in the Platform SDK, you can use objects of these C++ classes wherever you can use these C-language structures. The classes provide useful interfaces through their member functions. **CStringT** provides very flexible dynamic character strings. **CTime**, **COleDateTime**, **CTimeSpan**, and **COleTimeSpan** represent time and date values. For more information about these classes, see the article Date and Time.

The classes that begin with "**COle**" are encapsulations of data types provided by OLE. These data types can be used in Windows programs regardless of whether other OLE features are used.

CStringT Class

Holds character strings.

CTime

Holds absolute time and date values.

COleDateTime

Wrapper for the OLE automation type **DATE**. Represents date and time values.

CTimeSpan

Holds relative time and date values.

COleDateTimeSpan

Holds relative **COleDateTime** values, such as the difference between two **COleDateTime** values.

CPoint

Holds coordinate (x, y) pairs.

CSize

Holds distance, relative positions, or paired values.

CRect

Holds coordinates of rectangular areas.

CImageList

Provides the functionality of the Windows image list. Image lists are used with list controls and tree controls. They can also be used to store and archive a set of same-sized bitmaps.

COleVariant

Wrapper for the OLE automation type **VARIANT**. Data in **VARIANT**s can be stored in many formats.

COleCurrency

Wrapper for the OLE automation type **CURRENCY**, a fixed-point arithmetic type, with 15 digits before the decimal point and 4 digits after.

Note
Beginning with Visual C++ .NET, **CRect**, **CSize**, and **CPoint** have been modified to be usable in either ATL or MFC applications. In addition, **CStringT** has been added to provide an MFC-independent **CString**-like class. For more information on shared utility classes, see Shared Classes.

See Also

Concepts

Class Library Overview

Array, List, and Map Classes

For handling aggregates of data, the class library provides a group of collection classes — arrays, lists, and maps — that can hold a variety of object and predefined types. The collections are dynamically sized. These classes can be used in any program, whether written for Windows or not. However, they are most useful for implementing the data structures that define your document classes in the application framework. You can readily derive specialized collection classes from these, or you can create them based on the template classes. For more information about these approaches, see the article Collections. For a list of the template collection classes, see the article Template Classes for Arrays, Lists, and Maps.

Arrays are one-dimensional data structures that are stored contiguously in memory. They support very fast random access since the memory address of any given element can be calculated by multiplying the index of the element by the size of an element and adding the result to the base address of the array. But arrays are very expensive if you have to insert elements into the array, since the entire array past the element inserted has to be moved to make room for the element to be inserted. Arrays can grow and shrink as necessary.

Lists are similar to arrays but are stored very differently. Each element in a list also includes a pointer to the previous and next elements, making it a doubly linked list. It is very fast to add or delete items because doing so only involves changing a few pointers. However, searching a list can be expensive since all searches need to start at one of the list's ends.

Maps relate a key value to a data value. For instance, the key of a map could be a string and the data a pointer into a list. You would ask the map to give you the pointer associated with a particular string. Map lookups are fast because maps use hash tables for key lookups. Adding and deleting items is also fast. Maps are often used with other data structures as auxiliary indices. MFC uses a special kind of map called a message map to map Windows messages to a pointer to the handler function for that message.

See Also

Concepts

Class Library Overview

Template Classes for Arrays, Lists, and Maps

These collection classes are templates whose parameters determine the types of the objects stored in the aggregates. The **CArray**, **CMap**, and **CList** classes use global helper functions that must usually be customized. For more information about these helper functions, see Collection Class Helpers. The typed pointer classes are wrappers for other classes in the class library. By using these wrappers, you enlist the compiler's type-checking to help you avoid errors. For more information on using these classes, see Collections.

These classes provide templates you can use to create arrays, lists, and maps using any type you like.

CArray

Template class for making arrays of arbitrary types.

CList

Template class for making lists of arbitrary types.

CMap

Template class for making maps with arbitrary key and value types.

CTypedPtrArray

Template class for type-safe arrays of pointers.

CTypedPtrList

Template class for type-safe lists of pointers.

CTypedPtrMap

Template class for type-safe maps with pointers.

See Also

Concepts

Class Library Overview

Ready-to-Use Array Classes

The following classes are ready-to-use array classes.

CByteArray

Stores elements of type **BYTE** in an array.

CDWordArray

Stores elements of type **DWORD** in an array.

CObArray

Stores pointers to objects of class **CObject** or to objects of classes derived from **CObject** in an array.

CPtrArray

Stores pointers to **void** (generic pointers) in an array.

CUIntArray

Stores elements of type **UINT** in an array.

CWordArray

Stores elements of type **WORD** in an array.

CStringArray

Stores **CString** objects in an array.

See Also

Concepts

Class Library Overview

Ready-to-Use List Classes

The following classes are ready-to-use list classes.

CObList

Stores pointers to objects of class **CObject** or to objects of classes derived from **CObject** in a linked list.

CPtrList

Stores pointers to **void** (generic pointers) in a linked list.

CStringList

Stores **CString** objects in a linked list.

See Also

Concepts

Class Library Overview

Ready-to-Use Map Classes

The following classes are ready-to-use map classes.

CMapPtrToPtr

Uses **void** pointers as keys for finding other **void** pointers.

CMapPtrToWord

Uses **void** pointers as keys for finding data of type **WORD**.

CMapStringToOb

Uses **CString** objects as keys for finding **CObject** pointers.

CMapStringToPtr

Uses **CString** objects as keys for finding **void** pointers.

CMapStringToString

Uses **CString** objects as keys for finding other **CString** objects.

CMapWordToOb

Uses data of type **WORD** to find **CObject** pointers.

CMapWordToPtr

Uses data of type **WORD** to find **void** pointers.

See Also

Concepts

Class Library Overview

File and Database Classes

These classes allow you to store information to a database or a disk file. There are three sets of database classes — OLE DB, ODBC, and DAO — that provide similar functionality. The OLE DB group is implemented using OLE DB and works with the OLE DB consumer templates, the DAO group is implemented using the Data Access Object, and the ODBC group is implemented using Open Database Connectivity. There are also a set of classes for manipulating standard files, Active streams, and HTML streams.

The following categories of classes support data persistence.

- File I/O Classes
- OLE DB Classes
- DAO Classes
- ODBC Classes

See Also

Concepts

Class Library Overview

File I/O Classes

These classes provide an interface to traditional disk files, in-memory files, Active streams, and Windows sockets. All of the classes derived from **CFile** can be used with a **CArchive** object to perform serialization.

Use the following classes, particularly **CArchive** and **CFile**, if you write your own input/output processing. Normally you do not need to derive from these classes. If you use the application framework, the default implementations of the **Open** and **Save** commands on the **File** menu will handle file I/O (using class **CArchive**), as long as you override your document's **Serialize** function to supply details about how a document serializes its contents. For more information about the file classes and serialization, see the article Files in MFC and the article Serialization.

CFile

Provides a file interface to binary disk files.

CStdioFile

Provides a **CFile** interface to buffered stream disk files, usually in text mode.

CMemFile

Provides a **CFile** interface to in-memory files.

CSharedFile

Provides a **CFile** interface to shared in-memory files.

COleStreamFile

Uses the COM **IStream** interface to provide **CFile** access to compound files.

CSocketFile

Provides a **CFile** interface to a Windows Socket.

Related Classes

CArchive

Cooperates with a **CFile** object to implement persistent storage for objects through serialization (see CObject::Serialize).

CArchiveException

An archive exception.

CFileException

A file-oriented exception.

CFileDialog

Provides a standard dialog box for opening or saving a file.

CHtmlStream

Handles caching HTML output. Functionally similar to **CMemFile**.

CRecentFileList

Maintains the most recently used (MRU) file list.

See Also

Concepts

Class Library Overview

OLE DB Classes

The OLE DB support in MFC currently consists of the class COLEDBRecordView. **COleDBRecordView** displays database records in controls, through a form view directly connected to a CRowset object. For more information about the OLE DB consumer templates, see List of OLE DB Consumer Templates.

See Also
Concepts
Class Library Overview

DAO Classes

These classes work with the other application framework classes to give easy access to Data Access Object (DAO) databases, which use the same database engine as Microsoft Visual Basic and Microsoft Access. The DAO classes can also access a wide variety of databases for which Open Database Connectivity (ODBC) drivers are available.

Programs that use DAO databases will have at least a **CDaoDatabase** object and a **CDaoRecordset** object.

Note
As of Visual C++ .NET, the Visual C++ environment and wizards no longer support DAO (although the DAO classes are included and you can still use them). Microsoft recommends that you use ODBC for new MFC projects. You should only use DAO in maintaining existing applications.

CDaoWorkspace

Manages a named, password-protected database session from login to logoff. Most programs use the default workspace.

CDaoDatabase

A connection to a database through which you can operate on the data.

CDaoRecordset

Represents a set of records selected from a data source.

CDaoRecordView

A view that displays database records in controls.

CDaoQueryDef

Represents a query definition, usually one saved in a database.

CDaoTableDef

Represents the stored definition of a base table or an attached table.

CDaoException

Represents an exception condition arising from the DAO classes.

CDaoFieldExchange

Supports the DAO record field exchange (DFX) routines used by the DAO database classes. You will normally not directly use this class.

Related Classes

CLongBinary

Encapsulates a handle to storage for a binary large object (BLOB), such as a bitmap. **CLongBinary** objects are used to manage large data objects stored in database tables.

COleCurrency

Wrapper for the OLE automation type **CURRENCY**, a fixed-point arithmetic type, with 15 digits before the decimal point and 4 digits after.

COleDateTime

Wrapper for the OLE automation type **DATE**. Represents date and time values.

COleVariant

Wrapper for the OLE automation type **VARIANT**. Data in **VARIANT**s can be stored in many formats.

See Also

Concepts

Class Library Overview

ODBC Classes

These classes work with the other application framework classes to give easy access to a wide variety of databases for which Open Database Connectivity (ODBC) drivers are available.

Programs that use ODBC databases will have at least a **CDatabase** object and a **CRecordset** object.

CDatabase

Encapsulates a connection to a data source, through which you can operate on the data source.

CRecordset

Encapsulates a set of records selected from a data source. Recordsets enable scrolling from record to record, updating records (adding, editing, and deleting records), qualifying the selection with a filter, sorting the selection, and parameterizing the selection with information obtained or calculated at run time.

CRecordView

Provides a form view directly connected to a recordset object. The dialog data exchange (DDX) mechanism exchanges data between the recordset and the controls of the record view. Like all form views, a record view is based on a dialog template resource. Record views also support moving from record to record in the recordset, updating records, and closing the associated recordset when the record view closes.

CDBException

An exception resulting from failures in data access processing. This class serves the same purpose as other exception classes in the exception-handling mechanism of the class library.

CFieldExchange

Supplies context information to support record field exchange (RFX), which exchanges data between the field data members and parameter data members of a recordset object and the corresponding table columns on the data source. Analogous to class CDataExchange, which is used similarly for dialog data exchange (DDX).

Related Classes

CLongBinary

Encapsulates a handle to storage for a binary large object (BLOB), such as a bitmap. **CLongBinary** objects are used to manage large data objects stored in database tables.

CDBVariant

Allows you to store a value without worrying about the value's data type. **CDBVariant** tracks the data type of the current value, which is stored in a union.

See Also

Concepts

Class Library Overview

Internet and Networking Classes

These classes allow you to exchange information with another computer using ISAPI or a Windows Socket. There are also a set of classes for creating ISAPI extension DLLs and a set of classes for manipulating Windows Sockets.

The following categories of classes support connectivity.

- ISAPI Classes
- Windows Sockets Classes
- Win32 Internet Classes

See Also

Concepts

Class Library Overview

ISAPI Classes

ISAPI describes an interface for Internet servers. An example of an ISAPI server is Windows 2000 Server running Microsoft Internet Information Services.

HTTP filters handle server requests. They can be used to handle the following types of applications:

- Custom authentication schemes
- Data compression
- Encryption
- Logging

Filter Classes

CHttpFilter

Filters selected HTTP requests sent to an ISAPI server.

CHttpFilterContext

Manages the context for an HTTP filter. This is a helper class to handle multiple, concurrent requests of a **CHttpFilter** object.

Server Classes

ISAPI server extensions process server requests. MFC ISAPI classes will not process requests from a Common Gateway Interface (CGI).

CHttpServer

Extends the functionality of an ISAPI server by processing client requests.

CHttpServerContext

Manages the context for an ISAPI server extension. This is a helper class to handle multiple, concurrent requests of a **CHttpServer** object.

Related Classes

CHtmlStream

Handles caching HTML output. Functionally similar to **CMemFile**.

See Also

Concepts

Class Library Overview

Windows Sockets Classes

Windows Sockets provide a network protocol-independent way to communicate between two computers. These sockets can be synchronous (your program waits until the communication is done) or asynchronous (your program continues running while the communication is going on).

CAsyncSocket

Encapsulates the Windows Sockets API in a thin wrapper.

CSocket

Higher level abstraction derived from **CAsyncSocket**. It operates synchronously.

CSocketFile

Provides a **CFile** interface to a Windows Socket.

See Also

Concepts

Class Library Overview

Win32 Internet Classes

MFC wraps the Win32 Internet (WinInet) and ActiveX technology to make Internet programming easier.

CInternetSession

Creates and initializes one Internet session or several simultaneous Internet sessions and, if necessary, describes the connection to a proxy server.

CInternetConnection

Manages your connection to an Internet server.

CInternetFile

This class and its derived classes allow access to files on remote systems that use Internet protocols.

CHttpConnection

Manages your connection to an HTTP server.

CHttpFile

Provides the functionality to find and read files on an HTTP server.

CGopherFile

Provides the functionality to find and read files on a gopher server.

CFtpConnection

Manages your connection to an FTP server.

CGopherConnection

Manages your connection to a gopher server.

CFileFind

Performs local and Internet file searches.

CFtpFileFind

Aids in Internet file searches of FTP servers.

CGopherFileFind

Aids in Internet file searches of gopher servers.

CGopherLocator

Gets a gopher "locator" from a gopher server, determines the locator's type, and makes the locator available to **CGopherFileFind**.

CInternetException

Represents an exception condition related to an Internet operation.

See Also

Concepts

Class Library Overview

OLE Classes

The OLE classes work with the other application framework classes to provide easy access to the ActiveX API, giving your programs an easy way to provide the power of ActiveX to your users. Using ActiveX, you can:

- Create compound documents, which allow users to create and edit documents containing data created by multiple applications, including text, graphics, spreadsheets, sound, or other types of data.
- Create OLE objects that can be embedded in compound documents.
- Use OLE drag and drop to copy data between applications.
- Use Automation to control one program with another.
- Create ActiveX controls and ActiveX control containers (formerly called OLE controls and OLE control containers, respectively).

The following categories of classes support ActiveX:

- OLE Container Classes
- OLE Server Classes
- OLE Drag-and-Drop and Data Transfer Classes
- OLE Common Dialog Classes
- OLE Automation Classes
- OLE Control Classes
- Active Document Classes
- OLE-Related Classes

To see the inheritance of a class, use the Class Hierarchy Chart.

See Also

Concepts

Class Library Overview

OLE Container Classes

These classes are used by container applications. Both **COleLinkingDoc** and **COleDocument** manage collections of **COleClientItem** objects. Rather than deriving your document class from **CDocument**, you'll derive it from **COleLinkingDoc** or **COleDocument**, depending on whether you want support for links to objects embedded in your document.

Use a **COleClientItem** object to represent each OLE item in your document that is embedded from another document or is a link to another document.

COleDocObjectItem

Supports active document containment.

COleDocument

Used for compound document implementation, as well as basic container support. Serves as a container for classes derived from **CDocItem**. This class can be used as the base class for container documents and is the base class for **COleServerDoc**.

COleLinkingDoc

A class derived from **COleDocument** that provides the infrastructure for linking. You should derive the document classes for your container applications from this class instead of from **COleDocument** if you want them to support links to embedded objects.

CRichEditDoc

Maintains the list of OLE client items that are in the rich edit control. Used with CRichEditView and CRichEditCntrItem.

CDocItem

Abstract base class of **COleClientItem** and **COleServerItem**. Objects of classes derived from **CDocItem** represent parts of documents.

COleClientItem

A client item class that represents the client's side of the connection to an embedded or linked OLE item. Derive your client items from this class.

CRichEditCntrItem

Provides client-side access to an OLE item stored in a rich edit control when used with **CRichEditView** and **CRichEditDoc**.

COleException

An exception resulting from a failure in OLE processing. This class is used by both containers and servers.

See Also

Concepts

Class Library Overview

OLE Server Classes

These classes are used by server applications. Server documents are derived from **COleServerDoc** rather than from **CDocument**. Note that because **COleServerDoc** is derived from **COleLinkingDoc**, server documents can also be containers that support linking.

The **COleServerItem** class represents a document or portion of a document that can be embedded in another document or linked to.

COleIPFrameWnd and **COleResizeBar** support in-place editing while the object is in a container, and **COleTemplateServer** supports creation of document/view pairs so OLE objects from other applications can be edited.

COleServerDoc

Used as the base class for server-application document classes. **COleServerDoc** objects provide the bulk of server support through interactions with **COleServerItem** objects. Visual editing capability is provided using the class library's document/view architecture.

CDocItem

Abstract base class of **COleClientItem** and **COleServerItem**. Objects of classes derived from **CDocItem** represent parts of documents.

COleServerItem

Used to represent the OLE interface to **COleServerDoc** items. There is usually one **COleServerDoc** object, which represents the embedded part of a document. In servers that support links to parts of documents, there can be many **COleServerItem** objects, each of which represents a link to a portion of the document.

COleIPFrameWnd

Provides the frame window for a view when a server document is being edited in place.

COleResizeBar

Provides the standard user interface for in-place resizing. Objects of this class are always used in conjunction with **COleIPFrameWnd** objects.

COleTemplateServer

Used to create documents using the framework's document/view architecture. A **COleTemplateServer** object delegates most of its work to an associated **CDocTemplate** object.

COleException

An exception resulting from a failure in OLE processing. This class is used by both containers and servers.

See Also

Concepts

Class Library Overview

OLE Drag-and-Drop and Data Transfer Classes

These classes are used in OLE data transfers. They allow data to be transferred between applications by using the Clipboard or through drag and drop.

COleDropSource

Controls the drag-and-drop operation from start to finish. This class determines when the drag operation starts and when it ends. It also displays cursor feedback during the drag-and-drop operation.

COleDataSource

Used when an application provides data for a data transfer. **COleDataSource** could be viewed as an object-oriented Clipboard object.

COleDropTarget

Represents the target of a drag-and-drop operation. A **COleDropTarget** object corresponds to a window on screen. It determines whether to accept any data dropped onto it and implements the actual drop operation.

COleDataObject

Used as the receiver side to **COleDataSource**. **COleDataObject** objects provide access to the data stored by a **COleDataSource** object.

See Also

Concepts

Class Library Overview

OLE Common Dialog Classes

These classes handle common OLE tasks by implementing a number of standard OLE dialog boxes. They also provide a consistent user interface for OLE functionality.

COleDialog

Used by the framework to contain common implementations for all OLE dialog boxes. All dialog box classes in the user-interface category are derived from this base class. **COleDialog** cannot be used directly.

COleInsertDialog

Displays the Insert Object dialog box, the standard user interface for inserting new OLE linked or embedded items.

COlePasteSpecialDialog

Displays the Paste Special dialog box, the standard user interface for implementing the Edit Paste Special command.

COleLinksDialog

Displays the Edit Links dialog box, the standard user interface for modifying information about linked items.

COleChangeIconDialog

Displays the Change Icon dialog box, the standard user interface for changing the icon associated with an OLE embedded or linked item.

COleConvertDialog

Displays the Convert dialog box, the standard user interface for converting OLE items from one type to another.

COlePropertiesDialog

Encapsulates the Windows common OLE Properties dialog box. Common OLE Properties dialog boxes provide an easy way to display and modify the properties of an OLE document item in a manner consistent with Windows standards.

COleUpdateDialog

Displays the Update dialog box, the standard user interface for updating all links in a document. The dialog box contains a progress indicator to indicate how close the update procedure is to completion.

COleChangeSourceDialog

Displays the Change Source dialog box, the standard user interface for changing the destination or source of a link.

COleBusyDialog

Displays the Server Busy and Server Not Responding dialog boxes, the standard user interface for handling calls to busy applications. Usually displayed automatically by the **COleMessageFilter** implementation.

See Also

Concepts

Class Library Overview

OLE Automation Classes

These classes support automation clients (applications that control other applications). Automation servers (applications that can be controlled by other applications) are supported through dispatch maps.

COleDispatchDriver

Used to call automation servers from your automation client. When adding a class, this class is used to create type-safe classes for automation servers that provide a type library.

COleDispatchException

An exception resulting from an error during OLE automation. Automation exceptions are thrown by automation servers and caught by automation clients.

See Also

Concepts

Class Library Overview

OLE Control Classes

These are the primary classes you use when writing OLE controls. The **COleControlModule** class in an OLE control module is like the CWinApp class in an application. Each module implements one or more OLE controls; these controls are represented by **COleControl** objects. These controls communicate with their containers using **CConnectionPoint** objects.

The **CPictureHolder** and **CFontHolder** classes encapsulate COM interfaces for pictures and fonts, while the **COlePropertyPage** and **CPropExchange** classes help you implement property pages and property persistence for your control.

COleControlModule

Replaces the **CWinApp** class for your OLE control module. Derive from the **COleControlModule** class to develop an OLE control module object. It provides member functions for initializing your OLE control's module.

COleControl

Derive from the **COleControl** class to develop an OLE control. Derived from **CWnd**, this class inherits all the functionality of a Windows window object plus additional OLE-specific functionality, such as event firing and the ability to support methods and properties.

CConnectionPoint

The **CConnectionPoint** class defines a special type of interface used to communicate with other OLE objects, called a connection point. A connection point implements an outgoing interface that is able to initiate actions on other objects, such as firing events and change notifications.

CPictureHolder

Encapsulates the functionality of a Windows picture object and the **IPicture** COM interface; used to implement the custom Picture property of an OLE control.

CFontHolder

Encapsulates the functionality of a Windows font object and the **IFont** COM interface; used to implement the stock Font property of an OLE control.

COlePropertyPage

Displays the properties of an OLE control in a graphical interface, similar to a dialog box.

CPropExchange

Supports the implementation of property persistence for your OLE controls. Analogous to CDataExchange for dialog boxes.

CMonikerFile

Takes a moniker, or a string representation that it can make into a moniker, and binds it synchronously to the stream for which the moniker is a name.

CAsyncMonikerFile

Works similarly to **CMonikerFile**; however, it binds the moniker asynchronously to the stream for which the moniker is a name.

CDataPathProperty

Implements an OLE control property that can be loaded asynchronously.

CCachedDataPathProperty

Implements an OLE control property transferred asynchronously and cached in a memory file.

COleCmdUI

Allows an Active document to receive commands that originate in its container's user interface (such as FileNew, Open, Print, and so on), and allows a container to receive commands that originate in the Active document's user interface.

COleSafeArray

Works with arrays of arbitrary type and dimension.

See Also

Concepts

Class Library Overview

Active Document Classes

Active documents can be displayed either in the entire client window of a Web browser, such as Internet Explorer 5.5, or in an Active container, such as the Microsoft Office Binder, that supports Active documents.

CDocObjectServer

Maps the Active document interfaces, and initializes and activates an Active document object.

CDocObjectServerItem

Implements OLE server verbs specifically for Active document servers.

COleDocObjectItem

Implements Active document containment.

See Also

Concepts

Class Library Overview

OLE-Related Classes

These classes provide a number of different services, ranging from exceptions to file input and output.

COleObjectFactory

Used to create items when requested from other containers. This class serves as the base class for more specific types of factories, including **COleTemplateServer**.

COleMessageFilter

Used to manage concurrency with OLE Lightweight Remote Procedure Calls (LRPC).

COleStreamFile

Uses the COM **IStream** interface to provide **CFile** access to compound files. This class (derived from **CFile**) enables MFC serialization to use OLE structured storage.

CRectTracker

Used to allow moving, resizing, and reorientation of in-place items.

See Also

Concepts

Class Library Overview

Debugging and Exception Classes

These classes provide support for debugging dynamic memory allocation and for passing exception information from the function where the exception is thrown to the function where it is caught.

Use classes CDumpContext and CMemoryState during development to assist with debugging, as described in Debugging MFC Applications. Use CRuntimeClass to determine the class of any object at run time, as described in the article Accessing Run-Time Class Information. The framework uses **CRuntimeClass** to create objects of a particular class dynamically.

See Also

Concepts

Class Library Overview

Debugging Support Classes

MFC provides the following classes to help you debug dynamic memory allocation problems.

CDumpContext

Provides a destination for diagnostic dumps.

CMemoryState

Structure that provides snapshots of memory use. Also used to compare earlier and later memory snapshots.

See Also

Concepts

Class Library Overview

Exception Classes

The class library provides an exception-handling mechanism based on class **CException**. The application framework uses exceptions in its code; you can also use them in yours. For more information, see the article Exceptions. You can derive your own exception types from **CException**.

MFC provides an exception class from which you can derive your own exception as well as exception classes for all of the exceptions it supports.

CException

The base class for exceptions.

CArchiveException

An archive exception.

CDaoException

An exception resulting from a failure in a DAO database operation.

CDBException

An exception resulting from a failure in ODBC database processing.

CFileException

A file-oriented exception.

CMemoryException

An out-of-memory exception.

CNotSupportedException

An exception resulting from using an unsupported feature.

COleException

An exception resulting from a failure in OLE processing. This class is used by both containers and servers.

COleDispatchException

An exception resulting from an error during automation. Automation exceptions are thrown by automation servers and caught by automation clients.

CResourceException

An exception resulting from a failure to load a Windows resource.

CUserException

An exception used to stop a user-initiated operation. Typically, the user has been notified of the problem before this exception is thrown.

See Also

Concepts

Class Library Overview

MFC Technical Notes

A technical note is a document written for programmers by programmers.

Each technical note describes a problem or feature that is beyond the scope of the rest of the MFC documentation. The technical notes supplied reflect requests for information from users, as well as specialized information that the MFC developers anticipate advanced users will want.

There are two ways to browse through the technical notes:

- Technical Notes By Number
- Technical Notes By Category

See Also

Other Resources

Visual C++ Libraries Reference

MFC Reference

Technical Notes by Category

Technical notes are divided into the following categories. For a numerical listing of the technical notes, see Technical Notes by Number.

MFC and Windows

TN001: Window Class Registration
TN003: Mapping of Windows Handles to Objects
TN012: Using MFC with Windows 3.1 Robustness Features
TN015: Windows for Pen
TN017: Destroying Window Objects
TN034: Writing a Windows 3.0 Compatible MFC Application
TN051: Using CTL3D Now and in the Future
TN052: Writing Windows 95 Applications with MFC3.1

MFC Architecture

TN002: Persistent Object Data Format
TN006: Message Maps
TN016: Using C++ Multiple Inheritance with MFC
TN019: Updating Existing MFC Applications to MFC 3.0
TN021: Command and Message Routing
TN022: Standard Commands Implementation
TN025: Document, View, and Frame Creation
TN026: DDX and DDV Routines
TN029: Splitter Windows
TN030: Customizing Printing and Print Preview
TN031: Control Bars
TN032: MFC Exception Mechanism
TN037: Multithreaded MFC 2.1 Applications
TN044: MFC Support for DBCS
TN046: Commenting Conventions for the MFC Classes

TN058: MFC Module State Implementation
TN059: Using MFC MBCS/Unicode Conversion Macros
TN066: Common MFC 3.x to 4.0 Porting Issues

MFC Controls

TN014: Custom Controls
TN027: Emulation Support for Visual Basic Custom Controls
TN060: Windows Common Controls
TN061: ON_NOTIFY and WM_NOTIFY Messages
TN062: Message Reflection for Windows Controls

MFC Database

TN042: ODBC Driver Developer Recommendations
TN043: RFX Routines
TN045: MFC/Database Support for Long Varchar/Varbinary
TN047: Relaxing Database Transaction Requirements
TN048: Writing ODBC Setup and Administration Programs for MFC Database Applications
TN053: Custom DFX Routines for MFC DAO Classes
TN054: Calling DAO Directly While Using MFC DAO Classes
TN055: Migrating MFC ODBC Database Class Applications to MFC DAO Classes
TN068: Performing Transactions with the Microsoft Access 7 ODBC Driver

MFC Debugging

TN007: Debugging Trace Options

MFC DLLs

TN011: Using MFC as Part of a DLL
TN033: DLL Version of MFC
TN056: Installation of Localized MFC Components
TN057: Localization of MFC Components

MFC OLE

TN008: MFC OLE Support
TN018: Migrating OLE Applications from MFC 1.0 to MFC 2.0

TN038: MFC/OLE IUnknown Implementation
TN039: MFC/OLE Automation Implementation
TN040: MFC/OLE In-Place Resizing and Zooming
TN041: MFC/OLE1 Migration to MFC/OLE2
TN049: MFC/OLE MBCS to Unicode Translation Layer (MFCANS32)
TN050: MFC/OLE Common Dialogs (MFCUIx32)
TN064: Apartment-Model Threading in OLE Controls
TN065: Dual-Interface Support for OLE Automation Servers
TN071: MFC IOleCommandTarget Implementation

MFC Resources

TN020: ID Naming and Numbering Conventions
TN023: Standard MFC Resources
TN024: MFC-Defined Messages and Resources
TN028: Context-Sensitive Help Support
TN035: Using Multiple Resource Files and Header Files with Visual C++
TN036: Using CFormView with AppWizard and ClassWizard
TN070: MFC Window Class Names

MFC Internet

TN063: Debugging Internet Extension DLLs
TN067: Database Access from an ISAPI Server Extension
TN069: Processing HTML Forms Using Internet Server Extension DLLs and Command Handlers

Technical Notes by Number

The technical notes below are listed numerically, with the most recently written technical note first. For a listing by category, see Technical Notes by Category.

Number	Title
71	MFC IOleCommandTarget Implementation
70	MFC Window Class Names
69	Processing HTML Forms Using Internet Server Extension DLLs and Command Handlers
68	Performing Transactions with the Microsoft Access 7 ODBC Driver
67	Database Access from an ISAPI Server Extension
66	Common MFC 3.x to 4.0 Porting Issues
65	Dual-Interface Support for OLE Automation Servers
64	Apartment-Model Threading in OLE Controls
63	Debugging Internet Extension DLLs
62	Message Reflection for Windows Controls
61	ON_NOTIFY and WM_NOTIFY Messages
60	Windows Common Controls
59	Using MFC MBCS/Unicode Conversion Macros
58	MFC Module State Implementation
57	Localization of MFC Components
56	Installation of Localized MFC Components
55	Migrating MFC ODBC Database Class Applications to MFC DAO Classes
54	Calling DAO Directly While Using MFC DAO Classes
53	Writing Custom DFX Routines for DAO Database Classes
52	Writing Windows 95 Applications with MFC 3.1
51	Using CTL3D Now and in the Future
50	MFC/OLE Common Dialogs (MFCUIx32)
49	MFC/OLE MBCS to Unicode Translation Layer (MFCANS32)

48	Writing ODBC Setup and Administration Programs for MFC Database Applications
47	Relaxing Database Transaction Requirements
46	Commenting Conventions for the MFC Classes
45	MFC/Database Support for Long Varchar/Varbinary
44	MFC Support for DBCS
43	RFX Routines
42	ODBC Driver Developer Recommendations
41	MFC/OLE1 Migration to MFC/OLE2
40	MFC/OLE In-Place Resizing and Zooming
39	MFC/OLE Automation Implementation
38	MFC/OLE IUnknown Implementation
37	Multithreaded MFC 2.1 Applications
36	Using CFormView with AppWizard and ClassWizard
35	Using Multiple Resource Files and Header Files with Visual C++
34	Writing a Windows 3.0 Compatible MFC Application
33	DLL Version of MFC
32	MFC Exception Mechanism
31	Control Bars
30	Customizing Printing and Print Preview
29	Splitter Windows
28	Context-Sensitive Help Support
27	Emulation Support for Visual Basic Custom Controls
26	DDX and DDV Routines
25	Document, View, and Frame Creation
24	MFC-Defined Messages and Resources
23	Standard MFC Resources
22	Standard Commands Implementation

21	Command and Message Routing
20	ID Naming and Numbering Conventions
19	Updating existing MFC Applications to MFC 3.0
18	Migrating OLE Applications From MFC 1.0 to MFC 2.0
17	Destroying Window Objects
16	Using C++ Multiple Inheritance with MFC
15	Windows for Pen
14	Custom Controls
12	Using Windows 3.1 Robustness Features
11	Using MFC as Part of a DLL
8	MFC OLE Support
7	Debugging Trace Options
6	Message Maps
3	Mapping of Windows Handles to Objects
2	Persistent Object Data Format
1	Window Class Registration

Technical notes 1 to 17 are general topics that apply to MFC 1.0 and 2.0.

Technical notes 18 and 19 focus on migration of applications from MFC 1.0 to MFC 2.0.

Technical notes 20 to 36 are topics specific to MFC 2.0 (and higher)

Technical note 37 is specific to 32-bit and 64-bit versions of MFC.

Technical notes 38 to 48 are topics specific to MFC 2.5 (and higher)

Technical notes 49 to 52 are topics specific to MFC 3.0 (and higher)

Technical notes 53 to 71 are topics specific to MFC 4.0 (and higher)

Gaps in the Technical Note numbering are due to MFC 1.0 technical notes that are now obsolete.

TN001: Window Class Registration

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes the MFC routines that register the special **WNDCLASS**es needed by Microsoft Windows. Specific **WNDCLASS** attributes used by MFC and Windows are discussed.

The Problem

The attributes of a **CWnd** object, like an **HWND** in Windows, are stored in two places: the window object and the **WNDCLASS**. A **WNDCLASS** is different than a C++ class. The name of the **WNDCLASS** is passed to general window creation functions such as **CWnd::Create** and **CFrameWnd::Create** in the *lpszClassName* parameter.

This **WNDCLASS** must be registered through one of four means:

- Implicitly by MFC provided **WNDCLASS**es.
- Implicitly by subclassing a Windows control (or some other control).
- Explicitly by calling the MFC **AfxRegisterWndClass** or **AfxRegisterClass**.
- Explicitly by calling the Windows routine **RegisterClass**.

WNDCLASSes and MFC

The **WNDCLASS** structure consists of various fields that describe a window class. Following are the fields and how they are used in an MFC application.

Style	Style of window
LpfnWndProc	window proc, must be **AfxWndProc**
CbClsExtra	not used (should be zero)
CbWndExtra	not used (should be zero)
HInstance	automatically filled with **AfxGetInstanceHandle**
HIcon	icon for frame windows, see below
HCursor	cursor for when mouse is over window, see below
HbrBackground	background color, see below
LpszMenuName	not used (should be NULL)
LpszClassName	class name, see below

Provided WNDCLASSes

In previous versions of MFC (before MFC 4.0), there were a number of predefined Window classes provided. These Window classes are no longer provided by default because of technical problems related to versioning (multiple versions of MFC loaded in one address space) as well as concerns relating to the fact that both MFC applications and OLE Controls may use the MFC DLLs.

The following reference is provided to help migrate code that uses these previously provided **WNDCLASS**es. Applications

should use **AfxRegisterWndClass** (with the appropriate parameters) in place of these classes.

The following shows the classes and their attributes:

- "AfxWnd" is used for all child windows created with **CWnd::Create**.
    - class style : CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW
    - no icon
    - arrow cursor
    - no background color
- "AfxFrameOrView" is used for frame windows and views (including stand-alone **CFrameWnd**s and **CMDIChildWnd**s).
    - class style : CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
    - icon AFX_IDI_STD_FRAME
    - arrow cursor
    - COLOR_WINDOW background color
- "AfxMDIFrame" is used for the MDI frame window (that is, the parent) created with **CMDIFrameWnd::Create**.
    - class style : CS_DBLCLKS [reduces flash during resizing]
    - icon AFX_IDI_STD_MDIFRAME
    - arrow cursor
    - no background color
- "AfxControlBar" is used for the standard control bar implementation.
    - class style : 0 [reduces flash during resizing, no double clicks]
    - no icon
    - arrow cursor
    - gray background color (**COLOR_BTNFACE**)

If the application provides a resource with the specified resource ID (for example, AFX_IDI_STD_FRAME) ID, MFC will use that resource. Otherwise the default resource is used. For the icon, the standard application icon is used, and for the cursor, the standard arrow cursor is used.

There are two icons that support MDI applications with single document types (one icon for the main application, the other icon for iconic document/MDIChild windows). For multiple document types with different icons, you must register additional **WNDCLASS**es or use the **CFrameWnd::LoadFrame** function.

CFrameWnd::LoadFrame will automatically register a **WNDCLASS** using the standard "AfxFrameOrView" attributes but using the icon ID you specify as the first parameter to LoadFrame.

The values for background color and cursor for the **MDIFrameWnd** are not used since the client area of the **MDIFrameWnd** is completely covered by the "MDICLIENT" window. Microsoft does not encourage subclassing the "MDICLIENT" window so use the standard colors and cursor types when possible.

Subclassing Controls

If you subclass or superclass a Windows control (for example, **CButton**) then your class automatically gets the **WNDCLASS** attributes provided in the Windows implementation of that control.

The AfxRegisterWndClass Function

MFC provides a helper routine for registering a window class. Given a set of attributes (window class style, cursor, background brush, and icon), a synthetic name is generated, and the resulting window class is registered. For example,

```
const char* AfxRegisterWndClass(UINT nClassStyle, HCURSOR hCursor,    HBRUSH hbrBackground,
HICON hIcon);
```

This function returns a temporary string of the generated registered window class name. See the *Class Library Reference* for more details.

The string returned is a temporary pointer to a static string buffer which is valid until the next call to **AfxRegisterWndClass**. If you want to keep this string around, store it in a **CString** variable. For example,

```
CString strWndClass = AfxRegisterWndClass(CS_DBLCLK, ...);
...
CWnd* pWnd = new CWnd;
pWnd->Create(strWndClass, ...);
...
```

AfxRegisterWndClass will throw a **CResourceException** if the window class failed to register (either because of bad parameters, or out of Windows memory).

The RegisterClass and AfxRegisterClass Functions

If you want to do anything more sophisticated than what **AfxRegisterWndClass** provides, you may call the Windows API **RegisterClass** or the MFC function **AfxRegisterClass**. The **CWnd**, **CFrameWnd** and **CMDIChildWnd Create** functions take a *lpszClassName* string name for the window class as the first parameter. Any window class name can be used, regardless of how it was registered.

It is important to use **AfxRegisterClass** (or **AfxRegisterWndClass**) in a DLL on Win32. Win32 does not automatically unregister classes registered by a DLL, so this must be done explicitly when the DLL is terminated. By using **AfxRegisterClass** instead of **RegisterClass** this is done automatically for you. **AfxRegisterClass** maintains a list of unique classes registered by your DLL and will automatically unregister then when the DLL terminates. When using **RegisterClass** in a DLL, you must insure that all classes are unregistered when the DLL is terminated (in your **DllMain** function). Failure to do so may cause **RegisterClass** to fail unexpectedly when your DLL is used by another client application.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN002: Persistent Object Data Format

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes the MFC routines that support persistent C++ objects and the format of the object data when it is stored in a file. This only applies to classes with the **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** macros.

The Problem

The MFC implementation for persistent data relies on a compact binary format for saving the data for many objects in a single contiguous part of a file. This binary format provides the structure for how the data is stored, but it is the object's **Serialize** member function that provides the actual data saved by the object.

The MFC solves the structuring problem by using the class **CArchive**. A **CArchive** object provides a context for persistence that lasts from the time the archive is created until the **CArchive::Close** member function is called, either explicitly by the programmer or implicitly by the destructor when the scope containing the **CArchive** is exited.

This note describes the implementation of the **CArchive** members **ReadObject** and **WriteObject**. **ReadObject** and **WriteObject** are not called directly by instead are used by class-specific type-safe insertion and extraction operators generated automatically by the **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** macros.

```
class CMyObject : public CObject
{
    DECLARE_SERIAL(CMyObject)
};

IMPLEMENT_SERIAL(CMyObj, CObject, 1)

// example usage (ar is a CArchive&)
CMyObject* pObj;
CArchive& ar;
ar << pObj;        // calls ar.WriteObject(pObj)
ar >> pObj;        // calls ar.ReadObject(RUNTIME_CLASS(CObj))
```

This note describes code located in the MFC source file ARCOBJ.CPP. The main **CArchive** implementation can be found in ARCCORE.CPP.

Saving Objects to the Store (CArchive::WriteObject)

The member function **CArchive::WriteObject** writes header data used to reconstruct the object. This data consists of two parts: the type of the object and the state of the object. This member function is also responsible for maintaining the identity of the object being written out, so that only a single copy is saved, regardless of the number of pointers to that object (including circular pointers).

Saving (inserting) and restoring (extracting) objects relies on several "manifest constants." These are values that are stored in binary and provide important information to the archive (note the "w" prefix indicates 16-bit quantities):

Tag	Description
wNullTag	Used for NULL object pointers (0).
wNewClassTag	Indicates class description that follows is new to this archive context — (-1).
wOldClassTag	Indicates class of the object being read has been seen in this context (0x8000).

When storing objects, the archive maintains a **CMapPtrToPtr** (the **m_pStoreMap**) which is a mapping from a stored object to a 32-bit persistent identifier (PID). A PID is assigned to every unique object and every unique class name that is saved in the

context of the archive. These PIDs are handed out sequentially starting at 1. It is important to note that these PIDs have no significance outside the scope of the archive and, in particular, are not to be confused with record number or other identity items.

Starting with MFC version 4.0 the **CArchive** class has been extended to support very large archives. In previous versions, a PID was a 16-bit quantity, limiting the archive to 0x7FFE (32766) objects. PIDs are now 32-bit, but they are written out as 16-bit unless they are larger than 0x7FFE. Large PIDs are written as 0x7FFF followed by the 32-bit PID. This technique maintains file backward compatibility.

When a request is made to save an object to an archive (usually through the global insertion operator), a check is made for a NULL **CObject** pointer; if the pointer is NULL, the wNullTag is inserted into the archive stream.

If we have a real object pointer that is capable of being serialized (the class is a **DECLARE_SERIAL** class), we then check the **m_pStoreMap** to see if the object has been saved already. If it has, we insert the 32-bit PID associated with that object.

If the object has not been saved before, there are two possibilities we must take into account: either both the object and the exact type (that is, class) of the object are new to this archive context, or the object is of an exact type already seen. To determine if the type has been seen we query the **m_pStoreMap** for a **CRuntimeClass** object that matches the **CRuntimeClass** object associated with the object we are saving. If we have seen this class before, **WriteObject** inserts a tag that is the bit-wise OR'ing of wOldClassTag and this index. If the **CRuntimeClass** is new to this archive context, then **WriteObject** assigns a new PID to that class and insert it into the archive, preceded by the **wNewClassTag** value.

The descriptor for this class is then inserted into the archive using the **CRuntimeClass** member function **Store**. **CRuntimeClass::Store** inserts the schema number of the class (see below) and the ASCII text name of the class. Note that the use of the ASCII text name does not guarantee uniqueness of the archive across applications, thus it is advisable to tag your data files to prevent corruption. Following the insertion of the class information, the archive places the object into the **m_pStoreMap** and then calls the **Serialize** member function to insert class-specific data into the archive. Placing the object into the **m_pStoreMap** before calling **Serialize** prevents multiple copies of the object from being saved to the store.

When returning to the initial caller (usually the root of the network of objects), it is important to **Close** the archive. If other **CFile** operations are going to be done, the **CArchive** member function **Flush** MUST be called. Failure to do so will result in a corrupt archive.

Note
This implementation imposes a hard limit of 0x3FFFFFFE indices per archive context. This number represents the maximum n umber of unique objects and classes that can be saved in a single archive, but note that a single disk file can have an unlimite d number of archive contexts.

Loading Objects from the Store (CArchive::ReadObject)

Loading (extracting) objects uses the **CArchive::ReadObject** member function and is the converse of **WriteObject**. As with **WriteObject**, **ReadObject** is not called directly by user code; user code should call the type-safe extraction operator that calls **ReadObject** with the expected **CRuntimeClass**. This insures the type integrity of the extract operation.

Since the **WriteObject** implementation assigned increasing PIDs, starting with 1 (0 is predefined as the NULL object), the **ReadObject** implementation can use an array to maintain the state of the archive context. When a PID is read from the store, if the PID is greater than the current upper bound of the **m_pLoadArray**, then **ReadObject** knows that a new object (or class description) follows.

Schema Numbers

The schema number, which is assigned to the class when the class' **IMPLEMENT_SERIAL** is encountered, is the "version" of the class implementation. The schema refers to the implementation of the class, not to the number of times a given object has been made persistent (usually referred to as the object version).

If you intend to maintain several different implementations of the same class over time, incrementing the schema as you revise your object's **Serialize** member function implementation will enable you to write code that can load objects stored using older versions of the implementation.

The **CArchive::ReadObject** member function will throw a **CArchiveException** when it encounters a schema number in the persistent store that differs from the schema number of the class description in memory. It is not easy to recover from this exception.

You can use **VERSIONABLE_SCHEMA** OR'd with your schema version to keep this exception from being thrown. By using **VERSIONABLE_SCHEMA**, your code can take the appropriate action in its **Serialize** function by checking the return value from **CArchive::GetObjectSchema**.

Calling Serialize Directly

There are many cases where the overhead of the general object archive scheme of **WriteObject** and **ReadObject** is not necessary or desired. This is the common case of serializing the data into a **CDocument**. In this case the **Serialize** member function of the **CDocument** is called directly, not with the extract or insert operators. The contents of the document may in turn use the more general object archive scheme.

Calling **Serialize** directly has the following advantages and disadvantages:

- No extra bytes are added to the archive before or after the object is serialized. This not only makes the saved data smaller, but allows you to implement **Serialize** routines that can handle any file formats.
- The MFC is tuned so the **WriteObject** and **ReadObject** implementations and related collections will not be linked into your application unless you need the more general object archive scheme for some other purpose.
- Your code does not need to recover from old schema numbers. This makes your document serialization code responsible for encoding schema numbers, file format version numbers or whatever magic numbers desired at the start of your data files.
- Any object that is serialized with a direct call to **Serialize** must not use **CArchive::GetObjectSchema** or must handle a return value of **(UINT)-1** indicating that the version was unknown.

Because **Serialize** is called directly on your document, it is not usually possible for the sub-objects of the document to archive references to their parent document. These objects must be given a pointer to their container document explicitly or you must use **CArchive::MapObject** function to map the **CDocument** pointer to a PID before these back pointers are archived.

As noted above, you should encode the version and class information yourself when calling Serialize directly, allowing you to change the format later while still maintaining backward compatibility with older files. The **CArchive::SerializeClass** function can be called explicitly before directly serializing an object or before calling a base class.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN003: Mapping of Windows Handles to Objects

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes the MFC routines that support mapping Windows object handles to C++ objects.

The Problem

Windows objects are normally represented by **HANDLE**s. The MFC classes wrap Windows object handles with C++ objects. The handle wrapping functions of the MFC class library provide a way to find the C++ object that is wrapping the Windows object with a particular handle. There are times when a Windows object does not have a C++ wrapper object, however, and at these times a temporary object is created to act as the C++ wrapper.

The Windows objects that use handle maps are:

- **HWND** (**CWnd** and **CWnd**-derived classes)
- **HDC** (**CDC** and **CDC**-derived classes)
- **HMENU** (**CMenu**)
- **HPEN** (**CGdiObject**)
- **HBRUSH** (**CGdiObject**)
- **HFONT** (**CGdiObject**)
- **HBITMAP** (**CGdiObject**)
- **HPALETTE** (**CGdiObject**)
- **HRGN** (**CGdiObject**)
- **HIMAGELIST** (**CImageList**)
- **SOCKET** (**CSocket**)

Given a handle to any of these objects, you can find the MFC object that wraps the handle by calling the static member function **FromHandle**. For example, given an **HWND** called *hWnd*:

```
CWnd::FromHandle(hWnd)
```

will return a pointer to the **CWnd** that wraps the *hWnd*. If that *hWnd* does not have a specific wrapper object, then a temporary **CWnd** is created to wrap the **hWnd**. This makes it possible to get a valid C++ object from any handle.

Once you have a wrapper object, you can get to its handle through a public member variable. In the case of a **CWnd**, *m_hWnd* contains the **HWND** for that object.

Attaching Handles to MFC Objects

Given a newly created handle-wrapper object and a handle to a Windows object, you can associate the two by calling **Attach**. For example:

```
CWnd myWnd;
myWnd.Attach(hWnd);
```

This makes an entry in the permanent map associating *myWnd* and *hWnd*. Calling **CWnd::FromHandle**(*hWnd*) will now return a pointer to *myWnd*. When *myWnd* is deleted, the destructor will automatically destroy the hWnd by calling the Windows **DestroyWindow** function. If this is not desired, the *hWnd* must be detached from *myWnd* before the *myWnd* object

is destroyed (normally when leaving the scope at which *myWnd* was defined). The **Detach** member function does this.

```
myWnd.Detach();
```

More About Temporary Objects

Temporary objects are created whenever **FromHandle** is given a handle that does not already have a wrapper object. These temporary objects are detached from their handle and deleted by the **DeleteTempMap** functions. The default **OnIdle** processing in **CWinThread** automatically calls **DeleteTempMap** for each class that supports temporary handle maps. This means that you cannot assume a pointer to a temporary object will be valid past the point of exit from the function where the pointer was obtained, as the temporary object will be deleted during the Windows message-loop idle time.

Wrapper Objects and Multiple Threads

Both temporary and permanent objects are maintained on a per-thread basis. That is, one thread cannot access another threads C++ wrapper objects, regardless of whether it is temporary or permanent. As stated above, temporary objects are deleted when the thread which that temporary object belongs enters **OnIdle**.

To pass these objects from one thread to another, always send them as their native **HANDLE** type. Passing a C++ wrapper object from one thread to another will often result in unexpected results.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN006: Message Maps

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes the MFC message map facility.

The Problem

Microsoft Windows implements what are essentially virtual functions in window classes using its messaging facility. Due to the large number of messages involved, providing a separate virtual function for each Windows message results in a prohibitively large vtable.

Also, since the number of system-defined Windows messages changes over time, and since a specific application may want to define some Windows messages of its own, the message-map mechanism provides a level of indirection that prevents interface changes from breaking existing code.

Overview

MFC provides an alternative to the switch statement used in traditional Windows programs to handle messages sent to a window. A mapping from messages to member functions may be defined so that when a message is to be handled by a window, the appropriate member function is called automatically. This message-map facility is designed to be similar to virtual functions but has additional benefits not possible with C++ virtual functions.

Defining a Message Map

The **DECLARE_MESSAGE_MAP** macro declares three members for a class.

- A private array of **AFX_MSGMAP_ENTRY** entries called *_messageEntries*.
- A protected **AFX_MSGMAP** structure called **messageMap** that points to the *_messageEntries* array.
- A protected virtual function called **GetMessageMap** that returns the address of **messageMap**.

This macro should be placed in the declaration of any class using message maps. By convention, it is at the end of the class declaration. For example:

```
class CMyWnd : public CMyParentWndClass
{
    // my stuff...

protected:
    //{{AFX_MSG(CMyWnd)
    afx_msg void OnPaint();
    //}}AFX_MSG

    DECLARE_MESSAGE_MAP()
};
```

This is the format generated by AppWizard and ClassWizard when they create new classes. The //{{ and //}} brackets are needed for ClassWizard.

The message map's table is defined with a set of macros that expand to message map entries. A table begins with a **BEGIN_MESSAGE_MAP** macro call, which defines the class that is handled by this message map and the parent class to which unhandled messages are passed. The table ends with the **END_MESSAGE_MAP** macro call.

Between these two macro calls is an entry for each message to be handled by this message map. Every standard Windows message has a macro of the form ON_WM_xxx (where xxx is the name of the message) that generates an entry for that message.

A standard function signature has been defined for unpacking the parameters of each Windows message and providing type safety. These signatures may be found in the file AFXWIN.H in the declaration of CWnd. Each one is marked with the word **afx_msg** for easy identification.

Note
ClassWizard requires that you use the **afx_msg** keyword in your message map handler declarations.

These function signatures were derived using a simple convention. The name of the function always starts with "On". This is followed by the name of the Windows message with the WM_ removed and only the first letter of each word capitalized. The ordering of the parameters is wParam followed by **LOWORD(***lParam***)** then **HIWORD**(*lParam*). Unused parameters are not passed. Any handles that are wrapped by MFC classes are converted to pointers to the appropriate MFC objects. The following example shows how to handle the **WM_PAINT** message and cause the CMyWnd**::OnPaint** function to be called:

```
BEGIN_MESSAGE_MAP(CMyWnd, CMyParentWndClass)
    //{{AFX_MSG_MAP(CMyWnd)
    ON_WM_PAINT()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

The message map table must be defined outside the scope of any function or class definition. It should not be placed within an extern "C" block.

Note
ClassWizard will edit the message map entries that are found between the //{{ and //}} comment bracket.

User Defined Windows Messages

User-defined messages may be included in a message map by using the **ON_MESSAGE** macro. This macro accepts a message number and a member function of the form:

```
    // inside the class declaration
    afx_msg LRESULT OnMyMessage(WPARAM wParam, LPARAM lParam);

For example:
    #define WM_MYMESSAGE (WM_USER + 100)

BEGIN_MESSAGE_MAP(CMyWnd, CMyParentWndClass)
    ON_MESSAGE(WM_MYMESSAGE, OnMyMessage)
END_MESSAGE_MAP()
```

In this example, we establish a handler for a custom message with a Windows message ID derived from the standard **WM_USER** base for user-defined messages. You might invoke this handler with code such as:

```
CWnd* pWnd = ...;
pWnd->SendMessage(WM_MYMESSAGE);
```

The range of user-defined messages using this approach must be in the range WM_USER to 0x7fff.

Note
ClassWizard does not support entering **ON_MESSAGE** handler routines from the ClassWizard user interface: you must man ually enter them from the Visual C++ editor. Once entered, ClassWizard will parse these entries and allow you to browse the m just like any other message-map entries.

Registered Windows Messages

The **::RegisterWindowMessage** function is used to define a new window message that is guaranteed to be unique throughout the system. The macro **ON_REGISTERED_MESSAGE** is used to handle these messages. This macro accepts a the name of a near UINT variable that contains the registered windows message ID. For example

```
class CMyWnd : public CMyParentWndClass
{
public:
    CMyWnd();

    //{{AFX_MSG(CMyWnd)
    afx_msg LRESULT OnFind(WPARAM wParam, LPARAM lParam);
    //}}AFX_MSG

    DECLARE_MESSAGE_MAP()
};

static UINT NEAR WM_FIND = RegisterWindowMessage("COMMDLG_FIND");

BEGIN_MESSAGE_MAP(CMyWnd, CMyParentWndClass)
    //{{AFX_MSG_MAP(CMyWnd)
    ON_REGISTERED_MESSAGE(WM_FIND, OnFind)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

The registered Windows message ID variable (WM_FIND in the example above) must be a NEAR variable because of the way **ON_REGISTERED_MESSAGE** is implemented.

The range of user-defined messages using this approach will be in the range 0xC000 to 0xFFFF.

Note

ClassWizard does not support entering **ON_REGISTERED_MESSAGE** handler routines from the ClassWizard user interface, you must manually enter them from the text editor. Once entered, ClassWizard will parse these entries and allow you to browse them just like any other message-map entries.

Command Messages

Command messages from menus and accelerators are handled in message maps with the **ON_COMMAND** macro. This macro accepts a command ID as well as a member function. Only the specific **WM_COMMAND** message with a *wParam* equal to the specified command ID is handled by the member function specified in the message-map entry. Command handler member functions take no parameters and return void. The macro has the form:

```
ON_COMMAND(id, memberFxn)
```

Command update messages are routed through the same mechanism as **ON_COMMAND** handlers. The **ON_UPDATE_COMMAND_UI** macro is used instead. Command update handler member functions take a single parameter, a pointer to a **CCmdUI** object, and return void. The macro has the form

```
ON_UPDATE_COMMAND_UI(id, memberFxn)
```

An extended form of command message handlers is available for advanced uses. The **ON_COMMAND_EX** macro is used instead and provides a superset of the **ON_COMMAND** functionality. Extended command-handler member functions take a single parameter, a UINT containing the command ID, and return a BOOL. The BOOL return should be TRUE to indicate that the command has been handled, otherwise routing will continue to other command target objects.

Examples of the above forms:

- Inside RESOURCE.H (usually generated by Visual C++)

  ```
  #define    ID_MYCMD      100
  #define    ID_COMPLEX    101
  ```

- Inside the class declaration

  ```
  afx_msg void OnMyCommand();
  afx_msg void OnUpdateMyCommand(CCmdUI* pCmdUI);
```

```
afx_msg BOOL OnComplexCommand(UINT nID);
```

- Inside the message map definition

```
ON_COMMAND(ID_MYCMD, OnMyCommand)
ON_UPDATE_COMMAND_UI(ID_MYCMD, OnUpdateMyCommand)
ON_COMMAND_EX(ID_MYCMD, OnComplexCommand)
```

- In the implementation file

```
void CMyClass::OnMyCommand()
{
    // handle the command
}

void CMyClass::OnUpdateMyCommand(CCmdUI* pCmdUI)
{
    // set the UI state with pCmdUI
}

BOOL CMyClass::OnComplexCommand(UINT nID)
{
    // handle the command
    return TRUE;
}
```

Also available for advanced use is **ON_COMMAND_RANGE** and **ON_COMMAND_RANGE_EX** which allow you to handle a range of commands with a single command handler. See the product documentation for more information on these macros.

Note

ClassWizard supports creating **ON_COMMAND** and **ON_UPDATE_COMMAND_UI** handlers, but it does not support creatin g **ON_COMMAND_EX** or **ON_COMMAND_RANGE** handlers. However, Class Wizard will parse and allow you to browse all three command handler variants.

Control Notification Messages

Messages that are sent from child controls to a window have an extra bit of information in their message map entry: the control's ID. The message handler specified in a message map entry is called only if (1) the control notification code (high word of lParam), such as BN_CLICKED, matches the notification code specified in the message-map entry and (2) control ID (wParam) matches the control ID specified in the message-map entry.

Custom control notification messages may use the **ON_CONTROL** macro to define a message map entry with a custom notification code. This macro has the form

```
ON_CONTROL(wNotificationCode, id, memberFxn)
```

For advanced usage **ON_CONTROL_RANGE** can be used to handle a specific control notification from a range of controls with the same handler.

ClassWizard does not support creating an **ON_CONTROL** or **ON_CONTROL_RANGE** handler in the user interface; you must manually enter them with the text editor. Once entered, ClassWizard will parse these entries and allow you to browse them just like any other message map entries.

The Windows Common Controls make use of the more powerful **WM_NOTIFY** for complex control notifications. This version of MFC has direct support for this new message with the **ON_NOTIFY** and **ON_NOTIFY_RANGE** macros. See the product documentation for more information on these macros.

See Also

Other Resources

Technical Notes by Number

TN007: Debugging Trace Options

This technical note, which previously discussed the **TRACE** output mechanism and MFC, is now obsolete. For current information on this feature, see Using the TRACE Macro.

For information on displaying trace information in a running program, see ATLTraceTool.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN008: MFC OLE Support

This technical note, which previously discussed the MFC/OLE1 implementation and design, is now obsolete. MFC 1.0 and MFC 2.0 supported the original OLE 1.0 API. OLE 1.0 has been replaced by OLE 2.0 which is supported by MFC 2.5 (and higher). The following technical notes describe the MFC/OLE2 support included in this version of MFC:

- Technical Note 38 MFC/OLE IUnknown Implementation
- Technical Note 39 MFC/OLE Automation Implementation
- Technical Note 40 MFC/OLE In-Place Resizing and Zooming
- Technical Note 41 MFC/OLE1 Migration to MFC/OLE2

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN011: Using MFC as Part of a DLL

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes regular DLLs, which allow you to use the MFC library as part of a Windows dynamic-link library (DLL). It assumes you are familiar with Windows DLLs and how to build them. For information about MFC extension DLLs, which allow you to create extensions to the MFC library, see DLL Version of MFC.

DLL Interfaces

Regular DLLs assume interfaces between the application and the DLL are specified in normal C-like functions or explicitly exported classes. MFC class interfaces cannot be exported.

If both a DLL and an application want to use MFC, then both have a choice to either use the shared version of the MFC libraries or have a copy of the MFC library statically linked into them. In versions before Visual C++ 4.0, the MFC static link libraries were different for applications and DLLs. With the current version of MFC, the application and DLL may both use one of the standard versions of the MFC library. There is not a separate library for DLLs in this version (MFC makes the choice at runtime).

Regular DLLs have several advantages:

- The application using the DLL does not have to use MFC or, for that matter, it does not have to be a Visual C++ application.
- With regular DLLs that statically link to MFC, the size of the DLL depends only on those MFC and C runtime routines that are used and linked by the linker.
- The file size of regular DLLs that dynamically link to MFC may be much smaller than regular DLLs that statically link to MFC, and the savings in memory from using the shared version of MFC can be significant. However, you must distribute the shared DLLs MFCx0.DLL and MSVCRTx0.DLL (or similar files) with your DLL.
- There are no problems with classes changing underneath you. Your DLL design exports to only those APIs you want.
- With regular DLLs that statically link to MFC, if both DLL and application use MFC, there are no problems with the application wanting a different version of MFC than the DLL (or vice versa). Because the MFC library is statically linked into each DLL or EXE, there is no question about which version you have.

API Limitations

Some MFC capabilities are not applicable to the DLL version, either because of technical limitations or because those services are usually provided by the application. These limitations are listed below:

- **CWinApp::SetDialogBkColor** (color is ignored for message boxes)

Building Your DLL

When compiling regular DLLs that statically link to MFC, the symbols "**_USRDLL**" and "**_WINDLL**" must be defined. Your DLL code must also be compiled with the following compiler switches:

- /D_WINDLL signifies the compilation is for a DLL
- /D_USRDLL specifies you are building a regular DLL

When compiling regular DLLs that dynamically link to MFC, you must define the above symbols and use the above compiler switches. Additionally, the symbol "**_AFXDLL**" must be defined and your DLL code must be compiled with:

- /D_AFXDLL specifies that you are building a regular DLL that dynamically links to MFC

The interfaces (APIs) between the application and the DLL must be explicitly exported. It is recommended that you define your interfaces to be low bandwidth, sticking to C interfaces where possible. More direct C interfaces are easier to maintain than more complex C++ classes.

Place your APIs in a separate header that can be included by both C and C++ files (that way you won't limit your DLL customers to C++ programmers). See the header ScreenCap.h in the MFC Advanced Concepts sample DLLScreenCap for an example. To export your functions, enter them in the **EXPORTS** section of your module definition file (.DEF) or include **__declspec(dllexport)** on your function definitions. Use **__declspec(dllimport)** to import these functions into the client executable.

You must add the **AFX_MANAGE_STATE** macro at the beginning of all the exported functions in regular DLLs that dynamically link to MFC to set the current module state to the one for the DLL. This is done by adding the following line of code to the beginning of functions exported from the DLL:

```
AFX_MANAGE_STATE(AfxGetStaticModuleState( ))
```

WinMain -> DllMain

The MFC library defines the standard Win32 **DllMain** entry point that initializes your **CWinApp** derived object as in a normal MFC application. Place all DLL-specific initialization in the **InitInstance** member function as in a normal MFC application.

Note that the **CWinApp::Run** mechanism doesn't apply to a DLL, since the application owns the main message pump. If your DLL brings up modeless dialogs or has a main frame window of its own, your application's main message pump must call a DLL-exported routine that calls **CWinApp::PreTranslateMessage**.

See the DLLScreenCap sample for use of this function.

The **ExitInstance** member function of your CWinApp derived class will be called from the MFC provided **DllMain** function before the DLL is unloaded.

What to Do to Link It All Together

With regular DLLs that statically link to MFC, you must link your DLL with this library (NAFXCWD.LIB or NAFXCW.LIB) along with the version of the C runtimes called 'LIBCMT.LIB'. These libraries are pre-built and may be installed by specifying them when you run Visual C++ setup.

Sample Code

Please see the MFC Advanced Concepts sample program DLLScreenCap for a complete sample. Several interesting thing to note:

- The compiler flags of the DLL and the application are very different.
- The link lines and .DEF files for the DLL and the application are also very different.
- The application using the DLL doesn't even have to be in C++.
- The interface between the application and the DLL is a "C"-like API and are exported with DLLScreenCap.DEF.

The following illustrates what is needed for one API that is defined in a regular DLL that statically links to MFC:

```
#ifdef __cplusplus
extern "C" {
#endif  /* __cplusplus */

struct TracerData
{
    BOOL    bEnabled;
    UINT    flags;
};

BOOL FAR PASCAL EXPORT PromptTraceFlags(TracerData FAR* lpData);

#ifdef __cplusplus
}
#endif
```

In this example, the declaration is enclosed in an 'extern "C" { }' block for C++ users. This has several advantages. First, it makes your DLL APIs usable by non-C++ client applications. Second, it reduces DLL overhead since C++ name mangling will not be applied to the exported name. Lastly, it makes it easier to explicitly add to a .DEF file (for exporting by ordinal) without having to worry about name mangling.

All API functions are "**FAR PASCAL EXPORT**". Although not strictly necessary for Win32 DLLs, these definitions have been kept for easy back-porting to 16-bit Windows. The **FAR**, **PASCAL**, and **EXPORT** macros all expand to nothing under Win32.

The structures used by the API are not derived from MFC classes and are defined completely in the API header. This reduces the complexity of the interface between the DLL and the application and, once again, makes the DLL usable by C programs as well.

Any data pointers used in the API are explicit **FAR** pointers. Again, **FAR**, is not really necessary for Win32, but is useful if you plan to compile the code for 16-bit Windows sometime in the future.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN012: Using MFC with Windows 3.1 Robustness Features

This technical note, which previously discussed programming with various features of MFC for the Windows 3.1 platform, is now obsolete.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN014: Custom Controls

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes the MFC Support for custom and self-drawing controls. Dynamic subclassing is also described. General advice on ownership of **CWnd** objects vs. **HWND**s is presented.

The MFC sample application CTRLTEST illustrates many of these features. Please refer to the source code for the MFC General sample CTRLTEST and online help.

Owner-Draw Controls/Menus

Windows provides support for "owner draw" controls and menus. These are Windows messages sent to a parent window of a control or menu that allow you to customize the visual appearance and behavior of the control or menu.

MFC directly supports owner draw with the message map entries:

- **CWnd::OnDrawItem**
- **CWnd::OnMeasureItem**
- **CWnd::OnCompareItem**
- **CWnd::OnDeleteItem**

You can override these in your **CWnd**-derived class (usually a dialog or main frame window) to implement the owner-draw behavior.

This approach does not lead to reusable code. If you have two similar controls in two different dialogs, you must implement the custom control behavior in two places. The MFC-supported self-drawing control architecture solves this problem.

Self-Drawing Controls and Menus

MFC provides a default implementation (in **CWnd** and **CMenu**) for the standard owner-draw messages. This default implementation will decode the owner-draw parameters and delegate the owner-draw messages to the controls or menu. This is called "self-draw" since the drawing (/measuring/comparing) code is in the class of the control or menu, not in the owner window.

This allows you to build reusable control classes that display the control using "owner draw" semantics. The code for drawing the control, not the owner of the control, is in the control class. This is an object-oriented approach to custom control programming.

- For self-draw buttons:

```
CButton:DrawItem(LPDRAWITEMSTRUCT);
        // draw this button
```

- For self-draw menus:

```
CMenu:MeasureItem(LPMEASUREITEMSTRUCT);
        // measure the size of an item in this menu
CMenu:DrawItem(LPDRAWITEMSTRUCT);
        // draw an item in this menu
```

- For self-draw list boxes:

```
CListBox:MeasureItem(LPMEASUREITEMSTRUCT);
```

```
        // measure the size of an item in this list box
CListBox:DrawItem(LPDRAWITEMSTRUCT);
        // draw an item in this list box

CListBox:CompareItem(LPCOMPAREITEMSTRUCT);
        // compare two items in this list box if LBS_SORT
CListBox:DeleteItem(LPDELETEITEMSTRUCT);
        // delete an item from this list box
```

- For self-draw combo boxes:

```
CComboBox:MeasureItem(LPMEASUREITEMSTRUCT);
        // measure the size of an item in this combo box
CComboBox:DrawItem(LPDRAWITEMSTRUCT);
        // draw an item in this combo box

CComboBox:CompareItem(LPCOMPAREITEMSTRUCT);
        // compare two items in this combo box if CBS_SORT
CComboBox:DeleteItem(LPDELETEITEMSTRUCT);
        // delete an item from this combo box
```

For details on the owner-draw structures (**DRAWITEMSTRUCT**, **MEASUREITEMSTRUCT**, **COMPAREITEMSTRUCT**, and **DELETEITEMSTRUCT**) refer to the MFC documentation for **CWnd::OnDrawItem**, **CWnd::OnMeasureItem**, **CWnd::OnCompareItem**, and **CWnd::OnDeleteItem** respectively.

Using self-drawing controls and menus

For self-drawing menus, you must override both **MeasureItem** and **DrawItem** member functions.

For self-drawing list boxes and combo boxes, you must override **MeasureItem** and **DrawItem**. You must specify the OWNERDRAWVARIABLE style in the dialog template (**LBS_OWNERDRAWVARIABLE** and **CBS_OWNERDRAWVARIABLE** respectively). The **OWNERDRAWFIXED** style will not work with self-drawing items since the fixed item height is determined before self-drawing controls are attached to the list box. (The Win 3.1 member functions **CListBox::SetItemHeight** and **CComboBox::SetItemHeight** can be used to get around this limitation.)

In addition, note that switching to an **OWNERDRAWVARIABLE** style will affect the **NOINTEGRALHEIGHT** style. Because the control can not calculate an integral height with variable sized items, the default style of **INTEGRALHEIGHT** is ignored and the control is always **NOINTEGRALHEIGHT**. If your items are fixed height, you can prevent partial items from being drawn by specifying the control size to be an integral multiplier of the item size.

For self-drawing list boxes and combo boxes with the SORT style (**LBS_SORT** and **CBS_SORT** respectively), you must override the **CompareItem** member function.

For self-drawing list boxes and combo boxes, **DeleteItem** is not normally overridden. **DeleteItem** can be overridden if additional memory or other resources are stored with each list box or combo box item.

Examples of Self-Drawing Controls/Menus

The MFC General sample CTRLTEST provides samples of a self-draw menu (showing colors) and a self-draw list box (also showing colors).

The most typical example of a self-drawing button is a bitmap button (a button that shows one, two, or three bitmap images for the different states). This is provided in the MFC class **CBitmapButton**.

Dynamic Subclassing

Subclassing is the Windows term for replacing the **WndProc** of a window with a different **WndProc** and calling the old **WndProc** for default (superclass) functionality.

This should not be confused with C++ class derivation (C++ terminology uses the words "base" and "derived" while the Windows object model uses "super" and "sub"). C++ derivation with MFC and Windows subclassing are functionally very similar, except C++ does not support a feature similar to dynamic subclassing.

The **CWnd** class provides the connection between a C++ object (derived from **CWnd**) and a Windows window object (also known as an **HWND**).

There are three common ways these are related:

- **CWnd** creates the **HWND**. The behavior can be modified in a derived class. "Class derivation" is done by creating a class derived from **CWnd** and created with calls to **Create**.
- **CWnd** gets attached to an existing **HWND**. The behavior of the existing window is not modified. This is a case of "delegation" and is made possible by calling **Attach** to alias an existing **HWND** to a **CWnd** C++ object.
- **CWnd** gets attached to an existing **HWND** and you can modify the behavior in a derived class. This is called "dynamic subclassing," since we are changing the behavior (and hence the class) of a Windows object at run time.

This last case is done with the member functions:

- **CWnd::SubclassWindow**
- **CWnd::SubclassDlgItem**.

Both routines attach a **CWnd** object to an existing Windows **HWND**. **SubclassWindow** takes the **HWND** directly, and **SubclassDlgItem** is a helper that takes a control ID and the parent window (usually a dialog). **SubclassDlgItem** is designed for attaching C++ objects to dialog controls created from a dialog template.

Please refer to the CTRLTEST example for several examples of when to use **SubclassWindow** and **SubclassDlgItem**.

See Also
Other Resources
Technical Notes by Number
Technical Notes by Category

TN015: Windows for Pen Computing

The MFC 2.0 product for Windows 3.1 supports the Windows for Pen Computing environment that is available as an add-on to Windows 3.1. MFC 4.0 and later versions do not support the Pen Computing add-on to Windows 3.1, but do support Pen Computing on Windows 95/98. Windows NT does not yet provide the Windows for Pen Computing environment.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN016: Using C++ Multiple Inheritance with MFC

This note describes how to use Multiple Inheritance (MI) with the Microsoft Foundation Classes.

Why Multiple Inheritance?

There is an ongoing debate in the C++ and object-oriented communities over the value of multiple inheritance (MI). The Visual C++ compiler and development environment fully support MI.

The MFC class library has been designed so that you do not need to understand MI to use MFC. MI is not used in any of the MFC classes. We have found that MI is not required to write a class library, nor is it required for writing serious applications. To use MI or not can be a personal decision, so we leave that decision to you.

So You Want to Use MI?

If you already understand how to use MI, understand the performance trade-offs, and want to use MFC, this tech note will tell you what you must do. Some of the restrictions are general C++ restrictions, others are imposed by the MFC architecture.

The following describes some of the technical issues of how MI affects the use of common MFC idioms. At the end of this technical note a complete MFC application using MI is included which you can extract and compile.

CRuntimeClass

The persistence and dynamic object creation mechanisms of MFC use the **CRuntimeClass** data structure to uniquely identify classes. MFC associates one structure of this type with each dynamic and/or serializable class in the application. These structures are initialized at application startup time using a special static object of type **AFX_CLASSINIT**. You need not concern yourself with the implementation of this information, as it is likely to change between revisions of MFC.

The current implementation of **CRuntimeClass** does not support Multiple Inheritance runtime type information. This does not mean you cannot use MI in your MFC application, but if you do, you will have certain responsibilities when working with objects that have more than one base class.

The **CObject::IsKindOf** member function will not correctly determine the type of an object if it has multiple base classes. Therefore, you cannot use **CObject** as a virtual base class, and all calls to **CObject** member functions such as **Serialize** and **operator new** will need to have scope qualifiers so that C++ can disambiguate the appropriate function call. If you do find the need to use MI within MFC, then you should be sure to make the class containing the **CObject** base class the left-most class in the list of base classes.

For advice on the uses and abuses of MI, see *Advanced C++ Programming Styles and Idioms* by James O. Coplien (Addison Wesley, 1992).

CObject - The Root of all Classes

As you know, all significant classes derive directly or indirectly from class **CObject**. **CObject** does not have any member data, but does have some default functionality. When using MI, it will be common to inherit from two or more **CObject**-derived classes, for example, a **CFrameWnd** and a **CObList**:

```
class CListWnd : public CFrameWnd, public CObList
{
 ...
};
CListWnd myListWnd;
```

In this case **CObject** is included twice, which leads to two problems:

- Any reference to **CObject** member functions must be disambiguated.

  ```
  myListWnd.Dump(afxDump);
      // compile time error, CFrameWnd::Dump or CObList::Dump ?
  ```

- Static member functions, including '**operator new**' and '**operator delete**' must also be disambiguated.

Recommended Steps

When creating a new class with two or more **CObject** derived base classes, reimplement those **CObject** members that you

expect people to use. Operators **new** and **delete** are mandatory, and **Dump** is recommended. For example:

```
class CListWnd : public CFrameWnd, public CObList
{
public:
    void* operator new(size_t nSize)
        { return CFrameWnd::operator new(nSize); }
    void operator delete(void* p)
        { CFrameWnd::operator delete(p); }

    void Dump(CDumpContent& dc)
        { CFrameWnd::Dump(dc);
          CObList::Dump(dc); }
     ...
};
```

Virtual Inheritance of CObject ?

You may ask, "If you inherit CObject virtually, won't all of the ambiguity problems go away?"

Even in the efficient Microsoft Object Model, virtual inheritance is not as efficient as nonvirtual inheritance (just as Multiple Inheritance is not as efficient as single inheritance in certain cases). Since there is no member data in **CObject**, virtual inheritance is not needed to prevent multiple copies of a base class's member data.

The real answer is no, virtual inheritance will not solve the ambiguity problems illustrated above. For example, the **Dump** virtual member function is still ambiguous (since **CFrameWnd** and **CObList** implement it differently).

Therefore we recommend following the steps above to provide disambiguation:

CObject::IsKindOf and Run-Time Typing

The run-time typing mechanism supported by MFC in **CObject** uses the macros **DECLARE_DYNAMIC**, **IMPLEMENT_DYNAMIC, DECLARE_DYNCREATE**, **IMPLEMENT_DYNCREATE**, **DECLARE_SERIAL** and **IMPLEMENT_SERIAL**. These give the ability to do a run-time type check to allow for safe cast-downs.

These macros only support a single base class and will work in a limited way for multiply inherited classes. The base class you specify in **IMPLEMENT_DYNAMIC** or **IMPLEMENT_SERIAL** should be the first (or left-most) base class. For example,

```
class CListWnd : public CFrameWnd, public CObList
{
    DECLARE_DYNAMIC(CListWnd)
    ...
};
IMPLEMENT_DYNAMIC(CListWnd, CFrameWnd)
```

This will allow you to do type checking for the left-most base class only. The run-time type system will know nothing about additional bases (**CObList** in this case).

CWnd and Message Maps

For the MFC message map system to work correctly, there are two additional requirements:

- There must be only one **CWnd**-derived base class.
- The **CWnd**-derived base class must be the first (or left-most) base class.

In the example above, **CFrameWnd** is the first base class.

Some examples that will not work:

```
class CTwoWindows : public CFrameWnd, public CEdit
    { ... };
        // error : two copies of CWnd

class CListEdit : public CObList, public CEdit
    { ... };
        // error : CEdit (derived from CWnd) must be first
```

A Sample Program using MI

The following sample is a stand-alone application that consists of one class derived from **CFrameWnd** and **CWinApp**. This way of structuring an application is not a recommended, but this is an example of the smallest MFC application with one class.

```
#include <afxwin.h>

class CHelloAppAndFrame : public CFrameWnd, public CWinApp
{
public:
    CHelloAppAndFrame()
        { }

    // Necessary evil for MI disambiguity
    void* operator new(size_t nSize)
        { return CFrameWnd::operator new(nSize); }
    void operator delete(void* p)
        { CFrameWnd::operator delete(p); }

// Implementation
    // CWinApp overrides
    virtual BOOL InitInstance();
    // CFrameWnd overrides
    virtual void PostNcDestroy();
    afx_msg void OnPaint();

    DECLARE_MESSAGE_MAP()

};

BEGIN_MESSAGE_MAP(CHelloAppAndFrame, CFrameWnd)
    ON_WM_PAINT()
END_MESSAGE_MAP()

// since the frame window is not allocated on the heap, we must
// override PostNCDestroy not to delete the frame object
void CHelloAppAndFrame::PostNcDestroy()
{
    // do nothing (do not call base class)
}

void CHelloAppAndFrame::OnPaint()
{
    CPaintDC dc(this);
    CRect rect;
    GetClientRect(rect);

    CString s = "Hello, Windows!";
    dc.SetTextAlign(TA_BASELINE | TA_CENTER);
    dc.SetTextColor(::GetSysColor(COLOR_WINDOWTEXT));
    dc.SetBkMode(TRANSPARENT);
    dc.TextOut(rect.right / 2, rect.bottom / 2, s);
}

// Application initialization
BOOL CHelloAppAndFrame::InitInstance()
{
    // first create the main frame
    if (!CFrameWnd::Create(NULL, "Multiple Inheritance Sample",
        WS_OVERLAPPEDWINDOW, rectDefault))
        return FALSE;

    // the application object is also a frame window
    m_pMainWnd = this;
    ShowWindow(m_nCmdShow);
    return TRUE;
}
```

```
CHelloAppAndFrame theHelloAppAndFrame;
```

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN017: Destroying Window Objects

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes the use of the **CWnd::PostNcDestroy** member function. Use this function if you wish to do customized allocation of **CWnd**-derived objects.

This note also explains some of the reasons for the cardinal rule:

To destroy a C++ Windows object, use **DestroyWindow**, not "delete".

This is important. If you follow the guidelines below, you will have few cleanup problems (such as forgetting to delete/free C++ memory, forgetting to free system resources like **HWND**s, or freeing objects too many times).

The Problem

Windows objects (objects of classes derived from **CWnd**) represent both a C++ object (allocated in the application's heap) and an HWND (allocated in system resources by the window manager). Since there are several ways to destroy a window object, we must provide a set of rules that prevent system resource or application memory leaks and that prevent objects and Windows handles from being destroyed more than once.

This is more than a memory management problem. The presence of a Windows window has user-interface impact: a window drawn on the screen; once it is destroyed, there is also an impact on system resources. Leaking C++ memory in your application address space is not as bad as leaking system resources.

Destroying Windows

The two permitted ways to destroy a Windows object are:

- Calling **CWnd::DestroyWindow** or the Windows API **::DestroyWindow**.
- Explicitly deleting with the **delete** operator.

The first case is by far the most common. This case applies even if **DestroyWindow** is not called directly by your code. This is the case when the user directly closes a frame window (the default **WM_CLOSE** behavior is to call **DestroyWindow**), and when a parent window is destroyed, Windows calls **DestroyWindow** for all the children.

The second case, the use of the **delete** operator on Windows objects, should be very rare and only in the cases outlined below.

Auto Cleanup with CWnd::PostNcDestroy

When destroying a Windows window, the last Windows message sent to the window is **WM_NCDESTROY**. The default **CWnd** handler for that message (**CWnd::OnNcDestroy**) will detach the **HWND** from the C++ object and call the virtual function **PostNcDestroy**. Some classes override this function to delete the C++ object.

The default implementation of **CWnd::PostNcDestroy** does nothing which is appropriate for window objects allocated on the stack frame or embedded in other objects. This is not appropriate for window objects that are designed to be allocated by themselves on the heap (not embedded in other C++ object).

Those classes that are designed to be allocated by themselves on the heap override the **PostNcDestroy** member function to perform a "**delete this**". This statement will free any C++ memory associated with the C++ object. Even though the default **CWnd** destructor calls **DestroyWindow** if **m_hWnd** is non-NULL, this does not lead to infinite recursion since the handle will be detached and NULL during the cleanup phase.

Note
CWnd::PostNcDestroy is normally called after the Windows **WM_NCDESTROY** message is processed, as part of window destruction, and the **HWND** and the C++ window object are no longer attached. **CWnd::PostNcDestroy** will also be called in the implementation of most **Create** calls if failure occurs (see below for auto cleanup rules).

Auto Cleanup Classes

The following classes are not designed for auto-cleanup. They are normally embedded in other C++ object or on the stack:

- All standard Windows controls (**CStatic**, **CEdit**, **CListBox**, and so on).
- Any child windows derived directly from **CWnd** (for example, custom controls).
- Splitter windows (**CSplitterWnd**).
- Default control bars (classes derived from **CControlBar**, see Technical Note 31 for enabling auto-delete for control bar objects).
- Dialogs (**CDialog**) designed for modal dialogs on the stack frame.
- All the standard dialogs except **CFindReplaceDialog**.
- The default dialogs created by ClassWizard.

The following classes are designed for auto-cleanup. They are normally allocated by themselves on the heap:

- Main frame windows (derived directly or indirectly from **CFrameWnd**).
- View windows (derived directly or indirectly from **CView**).

If you wish to break any of these rules, you must override the **PostNcDestroy** member function in your derived class. To add auto-cleanup to your class, call your base class and then do a **delete this**. To remove auto-cleanup from your class, call **CWnd::PostNcDestroy** directly instead of the **PostNcDestroy** member in your direct base class.

The most common use of the above is to create a modeless dialog that can be allocated on the heap.

When to Call 'delete'

The recommended way to destroy a Windows object is to call **DestroyWindow**, either the C++ member function or the global **::DestroyWindow** API.

Do not call the global **::DestroyWindow** API to destroy an MDI Child window; use the virtual member function **CWnd::DestroyWindow** instead.

For C++ Window objects that do not perform auto-cleanup, using **DestroyWindow** instead of **delete** avoids problems of having to call **DestroyWindow** in the **CWnd::~CWnd** destructor where the VTBL is not pointing to the correctly derived class. This can lead to subtle bugs so the diagnostic (debug) version of MFC will warn you with

```
Warning: calling DestroyWindow in CWnd::~CWnd
   OnDestroy or PostNcDestroy in derived class will not be called
```

In the case of C++ Windows objects that do perform auto-cleanup, you must call **DestroyWindow**. If you use operator **delete** directly, the MFC diagnostic memory allocator will alert you that you are freeing memory twice (the first call to **delete** as well as the indirect call to "**delete this**" in the auto-cleanup implementation of **PostNcDestroy**).

After calling **DestroyWindow** on a non-auto-cleanup object, the C++ object will still be around, but **m_hWnd** will be NULL. After calling **DestroyWindow** on an auto-cleanup object, the C++ object will be gone, freed by the C++ delete operator in the auto-cleanup implementation of **PostNcDestroy**.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN018: Migrating OLE Applications from MFC 1.0 to MFC 2.0

This technical note provided guidelines for migrating MFC version 1 applications that use MFC/OLE to MFC 2.0. Because this version of MFC does not support OLE 1.0, this technical note is obsolete.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN019: Updating existing MFC Applications to MFC 3.0

This technical note, which previously discussed upgrading existing MFC applications to MFC 3.0, is now obsolete.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN020: ID Naming and Numbering Conventions

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes the ID naming and numbering conventions used by MFC 2.0 for resources, commands, strings, controls, and child windows.

The Problem

The MFC ID naming and numbering conventions are intended to meet the following requirements:

- Provide a consistent ID-naming standard used across the MFC library and MFC applications that are supported by the Visual C++ resource editor. This allows the programmer to more readily interpret IDs as to their type and origin.
- Emphasize the strong 1-to-1 relationship between certain types of IDs. This helps to clarify the MFC application framework architecture.
- Conform to already widely used standards for naming IDs in Windows.
- Partition the ID-numbering space, so as to avoid unintentional duplication of ID numbers among those assigned by the programmer, by MFC and Windows, and by Visual C++-edited resources.

Overview of ID Prefix Naming Convention

There are several categories or types of IDs in an application. The MFC ID-naming convention defines different prefixes for different resource types.

MFC uses the prefix "**IDR_**" to refer to a resource ID that applies to multiple resource types. For example, for a given frame window, the same "**IDR_**" value is used to refer to a menu, accelerator, string and icon resource all at once.

IDR_	Multiple resource types (Used for Menus, Accelerators primarily).
IDD_	For dialog template resources (for example, **IDD_DIALOG1**).
IDC_	For Cursor resources.
IDI_	For Icon resources.
IDB_	For Bitmap resources.
IDS_	For String resources.

Note that the IDS_ value for a string resource is the ID passed to **LoadString**. The actual implementation of string table resources groups together 16 strings into one segment.

Within a DIALOG resource, we follow the convention of:

IDOK,**IDCANCEL**	For standard push button IDs.
IDC_	For other dialog controls.

The "**IDC_**" prefix is also used for cursors. This naming conflict is not usually a problem since a typical application will have few cursors and a large number of dialog controls.

Within a Menu resource, we follow the convention of:

IDM_	For menu items not using the MFC command architecture.
ID_	For menu item commands using the MFC command architecture.

Commands that follow the MFC command architecture must have an **ON_COMMAND** command handler and may have an **ON_UPDATE_COMMAND_UI** handler. If these command handlers follow the MFC command architecture, they will function correctly whether they are bound to a menu item, a toolbar button or a dialog bar button. The same ID_ is also used for a menu prompt string displayed on the program's message bar. Most of the menu items in your application should follow the MFC command convention. All of the standard command IDs (for example, **ID_FILE_NEW**) follow this convention.

MFC also uses "**IDP**_" as a specialized form of strings (that is, instead of "**IDS**_"). Strings with the "**IDP**_" prefix are "prompts," that is, strings used in message boxes. "**IDP**_" strings may contain "%1" and "%2" as place holders of strings determined by the program. "**IDP**_" strings usually have help topics, while "**IDS**_" strings do not. "**IDP**_" strings are always localized, while "**IDS**_" strings may or may not be localized.

The MFC library also uses the "**IDW**_" prefix as a specialized form of control IDs (that is, instead of "**IDC**_"). These IDs are assigned to child windows such as views and splitters by the framework classes. MFC implementation IDs are prefixed with "**AFX**_".

Overview of ID-Numbering Convention

The following lists the valid ranges for the IDs of the specific types. Some of the limits are technical implementation limits while others are just conventions to prevent your IDs from colliding with Windows predefined IDs or MFC default implementations.

We strongly recommend you do not defined IDs outside the recommended ranges. Even though the lower limit of many of these ranges is 1 (0 is not used), common convention starts practical use of IDs at 100 or 101.

Prefix	**Resource type**	**Valid range**
IDR_	multiple	1 -> 0x6FFF
IDD_	dialog templates	1 -> 0x6FFF
IDC_,IDI_,IDB_	cursors, icons, bitmaps	1 -> 0x6FFF
IDS_, IDP_	general strings	1 -> 0x7FFF
ID_	commands	0x8000 -> 0xDFFF
IDC_	controls	8 -> 0xDFFF

Reasons for these range limits:

- By convention, the ID value of 0 is not used.
- Windows implementation limitations restrict true resource IDs to be less than or equal to 0x7FFF.
- MFC's internal framework implementations reserve several ranges: 0xE000->0xEFFF and 0x7000->0x7FFF.
- Several Windows system commands use the range of 0xF000 -> 0xFFFF.
- Control IDs of 1->7 are reserved by **IDOK**, **IDCANCEL**, and so on.
- The range of 0x8000->0xFFFF for strings is reserved for menu prompts for commands.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN021: Command and Message Routing

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes the command routing and dispatch architecture as well as advanced topics in general window message routing.

Please refer to Visual C++ for general details on the architectures described here, especially the distinction between Windows messages, control notifications, and commands. This note assumes you are very familiar with the issues described in the printed documentation and only addresses very advanced topics.

Command Routing and Dispatch MFC 1.0 Functionality Evolves to MFC 2.0 Architecture

Windows has the **WM_COMMAND** message that is overloaded to provide notifications of menu commands, accelerator keys and dialog-control notifications.

MFC 1.0 built on that a little by allowing a command handler (for example, "OnFileNew") in a **CWnd** derived class to get called in response to a specific **WM_COMMAND**. This is glued together with a data structure called the message map, and results in a very space-efficient command mechanism.

MFC 1.0 also provided additional functionality for separating control notifications from command messages. Commands are represented by a 16-bit ID, sometimes known as a Command ID. Commands normally start from a **CFrameWnd** (that is, a menu select or a translated accelerator) and get routed to a variety of other windows.

MFC 1.0 used command routing in a limited sense for the implementation of Multiple Document Interface (MDI). (An MDI frame window delegate commands to its active MDI Child window.)

This functionality has been generalized and extended in MFC 2.0 to allow commands to be handled by a wider range of objects (not just window objects). It provides a more formal and extensible architecture for routing messages and reuses the command target routing for not only handling of commands, but also for updating UI objects (like menu items and toolbar buttons) to reflect the current availability of a command.

Command IDs

See Visual C++ for an explanation of the command routing and binding process. Technical Note 20 contains information on ID naming.

We use the generic prefix "ID_" for command IDs. Command IDs are >= 0x8000. The message line or status bar will show the command description string if there is a STRINGTABLE resource with the same IDs as the command ID.

In the resources of your application, a command ID can appears in several places:

- In one STRINGTABLE resource that has the same ID as the message-line prompt.
- In possibly many MENU resources that are attached to menu items that invoke the same command.
- (ADVANCED) in a dialog button for a GOSUB command.

In the source code of your application, a command ID can appears in several places:

- In your RESOURCE.H (or other main symbol header file) to define application-specific command IDs.
- PERHAPS In an ID array used to create a toolbar.
- In an **ON_COMMAND** macro.
- PERHAPS In an **ON_UPDATE_COMMAND_UI** macro.

Currently, the only implementation in MFC that requires command IDs be >= 0x8000 is the implementation of GOSUB dialogs/commands.

GOSUB Commands, Using Command Architecture in Dialogs

The command architecture of routing and enabling commands works well with frame windows, menu items, toolbar buttons, dialog bar buttons, other control bars and other user-interface elements designed to update on demand and route commands or control IDs to a main command target (usually the main frame window). That main command target may route the command or control notifications to other command target objects as appropriate.

A dialog (modal or modeless) can benefit from some of the features of the command architecture if you assign the control ID of the dialog control to the appropriate command ID. Support for dialogs is not automatic, so you may have to write some additional code.

Note that for all these features to work properly, your command IDs should be >= 0x8000. Since many dialogs could get routed to the same frame, shared commands should be >= 0x8000, while the nonshared IDCs in a specific dialog should be <= 0x7FFF.

You can place a normal button in a normal modal dialog with the IDC of the button set to the appropriate command ID. When the user selects the button, the owner of the dialog (usually the main frame window) gets the command just like any other command. This is called a GOSUB command since it usually is used to bring up another dialog (a GOSUB of the first dialog).

You can also call the function **CWnd::UpdateDialogControls** on your dialog and pass it the address of your main frame window. This function will enable or disable your dialog controls based on whether they have command handlers in the frame. This function is called automatically for you for control bars in your application's idle loop, but you must call it directly for normal dialogs that you wish to have this feature.

When ON_UPDATE_COMMAND_UI is Called

Maintaining the enabled/checked state of all a program's menu items all the time can be a computationally expensive problem. A common technique is to enable/check menu items only when the user selects the POPUP. The MFC 2.0 implementation of **CFrameWnd** handles the **WM_INITMENUPOPUP** message and uses the command routing architecture to determine the states of menus through **ON_UPDATE_COMMAND_UI** handlers.

CFrameWnd also handles the **WM_ENTERIDLE** message to describe the current menu item selected on the status bar (also known as the message line).

An application's menu structure, edited by Visual C++, is used to represent the potential commands available at **WM_INITMENUPOPUP** time. **ON_UPDATE_COMMAND_UI** handlers can modify the state or text of a menu, or for advanced uses (like the File MRU list or the OLE Verbs pop-up menu), actually modify the menu structure before the menu is drawn.

The same sort of **ON_UPDATE_COMMAND_UI** processing is done for toolbars (and other control bars) when the application enters its idle loop. See the *Class Library Reference* and Technical Note 31 for more information on control bars.

Nested Pop-up Menus

If you are using a nested menu structure, you will notice that the **ON_UPDATE_COMMAND_UI** handler for the first menu item in the pop-up menu is called in two different cases.

First, it is called for the pop-up menu itself. This is necessary because pop-up menus do not have IDs and we use the ID of the first menu item of the pop-up menu to refer to the entire pop-up menu. In this case, the **m_pSubMenu** member variable of the **CCmdUI** object will be non-NULL and will point to the pop-up menu.

Second, it is called just before the menu items in the pop-up menu are to be drawn. In this case, the ID refers just to the first menu item and the **m_pSubMenu** member variable of the **CCmdUI** object will be NULL.

This allows you to enable the pop-up menu distinct from its menu items, but requires that you write some menu aware code. For example, in a nested menu with the following structure:

```
File>
    New>
        Sheet (ID_NEW_SHEET)
        Chart (ID_NEW_CHART)
```

The ID_NEW_SHEET and ID_NEW_CHART commands can be independently enabled or disabled. The **New** pop-up menu should be enabled if either of the two is enabled.

The command handler for ID_NEW_SHEET (the first command in the pop-up) would look something like:

```
void CMyApp::OnUpdateNewSheet(CCmdUI* pCmdUI)
{
```

```
    if (pCmdUI->m_pSubMenu != NULL)
    {
        // enable entire pop-up for "New" sheet and chart
        BOOL bEnable = m_bCanCreateSheet || m_bCanCreateChart;

        // CCmdUI::Enable is a no-op for this case, so we
        //   must do what it would have done.
        pCmdUI->m_pMenu->EnableMenuItem(pCmdUI->m_nIndex,
            MF_BYPOSITION |
                (bEnable ? MF_ENABLED : (MF_DISABLED | MF_GRAYED)));
        return;
    }
    // otherwise just the New Sheet command
    pCmdUI->Enable(m_bCanCreateSheet);
}
```

The command handler for ID_NEW_CHART would be a normal update command handler and look something like:

```
void CMyApp::OnUpdateNewChart(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(m_bCanCreateChart);
}
```

ON_COMMAND and ON_BN_CLICKED

The message map macros for **ON_COMMAND** and **ON_BN_CLICKED** are the same. The MFC command and control notification routing mechanism only uses the command ID to decide where to route to. Control notifications with control notification code of zero (**BN_CLICKED**) are interpreted as commands.

Note In fact, all control notification messages go through the command handler chain. For example, it is technically possible for you to write a control notification handler for **EN_CHANGE** in your document class. This is not generally advisable because the practical applications of this feature are few, the feature is not supported by ClassWizard, and use of the feature can result in fragile code.

Disabling the Automatic Disabling of Button Controls

If you place a button control on a dialog bar, or in a dialog using where you are calling **CWnd::UpdateDialogControls** on your own, you will notice that buttons which do not have **ON_COMMAND** or **ON_UPDATE_COMMAND_UI** handlers will be automatically disabled for you by the framework. In some cases, you will not need to have a handler, but you will want the button to remain enabled. The easiest way to achieve this is to add a dummy command handler (easy to do with ClassWizard) and do nothing in it.

Window Message Routing

The following describes some more advanced topics on the MFC classes and how Windows message routing and other topics impact them. The information here is only described briefly. Refer to the *Class Library Reference* for details about public APIs. Please refer to the MFC library source code for more information on implementation details.

Please refer to Technical Note 17 for details on Window cleanup, a very important topic for all **CWnd**-derived classes.

CWnd Issues

The implementation member function **CWnd::OnChildNotify** provides a powerful and extensible architecture for child windows (also known as controls) to hook or otherwise be informed of messages, commands, and control notifications that go to their parent (or "owner"). If the child window (/control) is a C++ **CWnd** object itself, the virtual function **OnChildNotify** is called first with the parameters from the original message (that is, a **MSG** structure). The child window can leave the message alone, eat it, or modify the message for the parent (rare).

The default **CWnd** implementation handles the following messages and uses the **OnChildNotify** hook to allow child windows (controls) to first access at the message:

- **WM_MEASUREITEM** and **WM_DRAWITEM** (for self-draw)
- **WM_COMPAREITEM** and **WM_DELETEITEM** (for self-draw)
- **WM_HSCROLL** and **WM_VSCROLL**

- **WM_CTLCOLOR**
- **WM_PARENTNOTIFY**

You will notice the **OnChildNotify** hook is used for changing owner-draw messages into self-draw messages.

In addition to the **OnChildNotify** hook, scroll messages have further routing behavior. Please see below for more details on scroll bars and sources of **WM_HSCROLL** and **WM_VSCROLL** messages.

CFrameWnd Issues

The **CFrameWnd** class provides most of the command routing and user-interface updating implementation. This is primarily used for the main frame window of the application (**CWinApp::m_pMainWnd**) but applies to all frame windows.

The main frame window is the window with the menu bar and is the parent of the status bar or message line. Please refer to the above discussion on command routing and **WM_INITMENUPOPUP.**

The **CFrameWnd** class provides management of the active view. The following messages are routed through the active view:

- All command messages (the active view gets first access to them).
- **WM_HSCROLL** and **WM_VSCROLL** messages from sibling scroll bars (see below).
- **WM_ACTIVATE** (and **WM_MDIACTIVATE** for MDI) get turned into calls to the virtual function **CView::OnActivateView**.

CMDIFrameWnd/CMDIChildWnd Issues

Both MDI frame window classes derive from **CFrameWnd** and therefore are both enabled for the same sort of command routing and user-interface updating provided in **CFrameWnd**. In a typical MDI application, only the main frame window (that is, the **CMDIFrameWnd** object) holds the menu bar and the status bar and therefore is the main source of the command routing implementation.

The general routing scheme is that the active MDI child window gets first access to commands. The default **PreTranslateMessage** functions handle accelerator tables for both MDI child windows (first) and the MDI frame (second) as well as the standard MDI system-command accelerators normally handled by **TranslateMDISysAccel** (last).

Scroll Bar Issues

When handling scroll-message (**WM_HSCROLL**/**OnHScroll** and/or **WM_VSCROLL**/**OnVScroll**), you should try to write the handler code so it does not rely on where the scroll bar message came from. This is not only a general Windows issue, since scroll messages can come from true scroll bar controls or from **WS_HSCROLL**/**WS_VSCROLL** scroll bars which are not scroll bar controls.

MFC extends that to allow for scroll bar controls to be either child or siblings of the window being scrolled (in fact, the parent/child relationship between the scroll bar and window being scrolled can be anything). This is especially important for shared scroll bars with splitter windows. Please refer to Technical Note 29 for details on the implementation of **CSplitterWnd** including more information on shared scroll bar issues.

On a side note, there are two **CWnd** derived classes where the scroll bar styles specified at create time are trapped and not passed to Windows. When passed to a creation routine, **WS_HSCROLL** and **WS_VSCROLL** can be independently set, but after creation cannot be changed. Of course, you should not directly test or set the WS_?SCROLL style bits of the window that they created.

For **CMDIFrameWnd** the scroll bar styles you pass in to **Create** or **LoadFrame** are used to create the MDICLIENT. If you wish to have a scrollable MDICLIENT area (like the Windows Program Manager) be sure to set both scroll bar styles (**WS_HSCROLL** | **WS_VSCROLL**) for the style used to create the **CMDIFrameWnd**.

For **CSplitterWnd** the scroll bar styles apply to the special shared scroll bars for the splitter regions. For static splitter windows, you will normally not set either scroll bar style. For dynamic splitter windows, you will usually have the scroll bar style set for the direction you will split, That is, **WS_HSCROLL** if you can split rows, **WS_VSCROLL** if you can split columns.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN022: Standard Commands Implementation

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes the standard command implementations provided by MFC 2.0. Read Technical Note 21 first because it describes the mechanisms used to implement many of the standard commands.

This description assumes knowledge of the MFC architectures, APIs, and common programming practice. Documented as well as undocumented "implementation only" APIs are described. This is not a place to start learning about the features of or how to program in MFC. Refer to Visual C++ for more general information and for details of documented APIs.

The Problem

MFC defines many standard command IDs in the header file AFXRES.H. Framework support for these commands varies. Understanding where and how the framework classes handle these commands will not only show you how the framework works internally but will provide useful information on how to customize the standard implementations and teach you a few techniques for implementing your own command handlers.

Contents of This Technical Note

Each command ID is described in two sections:

- The title: the symbolic name of the command ID (for example, **ID_FILE_SAVE**) followed by the purpose of the command (for example, "saves the current document") separated by a colon.
- One or more paragraphs describing which classes implement the command, and what the default implementation does

Most default command implementations are prewired in the framework's base class message map. There are some command implementations that require explicit wiring in your derived class. These are described under "Note". If you chose the right options in AppWizard, these default handlers will be connected for you in the generated skeleton application.

Naming Convention

Standard commands follow a simple naming convention that we recommend you use if possible. Most standard commands are located in standard places in an application's menu bar. The symbolic name of the command starts with "ID_" followed by the standard pop-up menu name, followed by the menu item name. The symbolic name is in upper case with underscore word-breaks. For commands that do not have standard menu item names, a logical command name is defined starting with "ID_" (for example, **ID_NEXT_PANE**).

We use the prefix "ID_" to indicate commands that are designed to be bound to menu items, toolbar buttons, or other command user-interface objects. Command handlers handling "ID_" commands should use the **ON_COMMAND** and **ON_UPDATE_COMMAND_UI** mechanisms of the MFC command architecture.

We recommend you use the standard "IDM_" prefix for menu items which do not follow the command architecture and need menu-specific code to enable and disable them. Of course the number of menu specific commands should be small since following the MFC command architecture not only makes command handlers more powerful (since they will work with toolbars) but makes the command handler code reusable.

ID Ranges

Please refer to Technical Note 20 for more details on the use of ID ranges in MFC.

MFC standard commands fall in the range 0xE000 to 0xEFFF. Please do not rely on the specific values of these IDs since they are subject to change in future versions of the library.

Your application should define its commands in the range 0x8000 to 0xDFFF.

Standard Command IDs

For each command ID, there is a standard message line prompt string that can be found in the file PROMPTS.RC. The string ID for that menu prompt must be the same as for the command ID.

- ID_FILE_NEW Creates a new/empty document.

Note
You must connect this to your **CWinApp**-derived class's message map to enable this functionality.

CWinApp::OnFileNew implements this command differently depending on the number of document templates in the application. If there is only one **CDocTemplate**, **CWinApp::OnFileNew** will create a new document of that type, as well as the proper frame and view class.

If there is more than one **CDocTemplate**, **CWinApp::OnFileNew** will prompt the user with a dialog (**AFX_IDD_NEWTYPEDLG**) letting them select which document type to use. The selected **CDocTemplate** is used to create the document.

One common customization of **ID_FILE_NEW** is to provide a different and more graphical choice of document types. In this case you can implement your own **CMyApp::OnFileNew** and place it in your message map instead of **CWinApp::OnFileNew**. There is no need to call the base class implementation.

Another common customization of **ID_FILE_NEW** is to provide a separate command for creating a document of each type. In this case you should define new command IDs, for example ID_FILE_NEW_CHART and ID_FILE_NEW_SHEET.

- ID_FILE_OPEN Opens an existing document.

Note
You must connect this to your **CWinApp**-derived class's message map to enable this functionality.

CWinApp::OnFileOpen has a very simple implementation of calling **CWinApp::DoPromptFileName** followed by **CWinApp::OpenDocumentFile** with the file or path name of the file to open. The **CWinApp** implementation routine **DoPromptFileName** brings up the standard FileOpen dialog and fills it with the file extensions obtained from the current document templates.

One common customization of **ID_FILE_OPEN** is to customize the FileOpen dialog or add additional file filters. The recommended way to customize this is to replace the default implementation with your own FileOpen dialog, and call **CWinApp::OpenDocumentFile** with the document's file or path name. There is no need to call the base class.

- ID_FILE_CLOSE Closes the currently open document.

CDocument::OnFileClose calls **CDocument::SaveModified** to prompt the user to save the document if it has been modified and then calls **OnCloseDocument**. All the closing logic, including destroying the document, is done in the **OnCloseDocument** routine.

Note
ID_FILE_CLOSE acts differently from a **WM_CLOSE** message or an **SC_CLOSE** system command sent to the document s frame window. Closing a window will close the document only if that is the last frame window showing the document . Closing the document with **ID_FILE_CLOSE** will not only close the document but will close down all frame windows sh owing the document.

- ID_FILE_SAVE Saves the current document.

The implementation uses a helper routine **CDocument::DoSave** which is used for both **OnFileSave** and **OnFileSaveAs**. If you save a document that has not been saved before (that is, it does not have a path name, as in the case of FileNew) or that was read from a read-only document, the **OnFileSave** logic will act like the **ID_FILE_SAVE_AS** command and ask the user to provide a new file name. The actual process of opening the file and doing the saving is done through the virtual function **OnSaveDocument**.

There are two common reasons to customize **ID_FILE_SAVE**. For documents that do not save, simply remove the **ID_FILE_SAVE** menu items and toolbar buttons from your user interface. Also make sure that you never dirty your document (that is, never call **CDocument::SetModifiedFlag**) and the framework will never cause the document to be saved. For documents that save to someplace other than a disk file, define a new command for that operation.

In the case of a **COleServerDoc**, **ID_FILE_SAVE** is used both for file save (for normal documents) and file update (for embedded documents).

If your document data is stored in individual disk files, but you don't want to use the default **CDocument** serialize implementation, you should override **CDocument::OnSaveDocument** instead of **OnFileSave**.

- ID_FILE_SAVE_AS Saves the current document under a different file name.

  The **CDocument::OnFileSaveAs** implementation uses the same **CDocument::DoSave** helper routine as **OnFileSave**. The **OnFileSaveAs** command is handled just as **ID_FILE_SAVE** if the documents had no file name before the save. **COleServerDoc::OnFileSaveAs** implements the logic to save a normal document data file or to save a server document representing an OLE object embedded in some other application as a separate file.

  If you customize the logic of **ID_FILE_SAVE**, you will probably want to customize **ID_FILE_SAVE_AS** in a similar fashion or the operation of "Save As" may not apply to your document. You can remove the menu item from your menu bar if it is not needed.

- ID_FILE_SAVE_COPY_AS Saves a copy current document under a new name.

  The **COleServerDoc::OnFileSaveCopyAs** implementation is very similar to **CDocument::OnFileSaveAs**, except that the document object is not "attached" to the underlying file after the save. That is, if the in-memory document was "modified" before the save, it is still "modified". In addition, this command has no effect on the path name or title stored in the document.

- ID_FILE_UPDATE Notifies the container to save an embedded document.

  The **COleServerDoc::OnUpdateDocument** implementation simply notifiies the container that the embedding should be saved. The container then calls the appropriate OLE APIs in order to save the embedded object.

- ID_FILE_PAGE_SETUP Invokes an application-specific page setup/layout dialog.

  Currently there is no standard for this dialog, and the framework has no default implementation of this command.

  If you choose to implement this command, we recommend you use this command ID.

- ID_FILE_PRINT_SETUP Invoke the standard Print Setup dialog.

Note
You must connect this to your **CWinApp**-derived class's message map to enable this functionality.

  This command invokes the standard print setup dialog that allows the user to customize the printer and print settings for at least this document or at most all the documents in this application. You must use the Control Panel to change the default printer settings for the entire system.

  CWinApp::OnFilePrintSetup has a very simple implementation creating a **CPrintDialog** object and calling the **CWinApp::DoPrintDialog** implementation function. This sets the application default printer setup.

  The common need for customizing this command is to allow for per-document printer settings, which should be stored with the document when saved. To do this you should add a message-map handler in your **CDocument** class that creates a **CPrintDialog** object, initializes it with the appropriate printer attributes (usually **hDevMode** and **hDevNames**), call the **CPrintDialog::DoModal,** and save the changed printer settings. For a robust implementation, you should look at the implementation of **CWinApp::DoPrintDialog** for detecting errors and **CWinApp::UpdatePrinterSelection** for dealing with sensible defaults and tracking system-wide printer changes.

- ID_FILE_PRINT Standard printing of the current document

Note
You must connect this to your **CView**-derived class's message map to enable this functionality.

  This command prints the current document, or more correctly, starts the printing process, which involves invoking the standard print dialog and running the print engine.

  CView::OnFilePrint implements this command and the main print loop. It calls the virtual **CView::OnPreparePrinting** to prompt of the user with the print dialog. It then prepares the output DC to go to the printer, brings up the printing progress dialog (**AFX_IDD_PRINTDLG**), and sends the **StartDoc** escape to the printer. **CView::OnFilePrint** also contains the main page-oriented print loop. For each page, it calls the virtual **CView::OnPrepareDC** followed by a **StartPage** escape and calling the virtual **CView::OnPrint** for that page. When complete, the virtual **CView::OnEndPrinting** is

called, and the printing progress dialog is closed.

The MFC printing architecture is designed to hook in many different ways for printing and print preview. You will normally find the various **CView** overridable functions adequate for any page-oriented printing tasks. Only in the case of an application that uses the printer for non-page oriented output, should you find the need to replace the **ID_FILE_PRINT** implementation.

- ID_FILE_PRINT_PREVIEW Enter print-preview mode for the current document.

Note
You must connect this to your **CView**-derived class's message map to enable this functionality.

CView::OnFilePrintPreview starts the print preview mode by calling the documented helper function **CView::DoPrintPreview**. **CView::DoPrintPreview** is the main engine for the print preview loop, just as **OnFilePrint** is the main engine for the printing loop.

The print preview operation can be customized in a variety of ways by passing different parameters to **DoPrintPreview**. Please refer to Technical Note 30, which discusses some of the details of print preview and how to customize it.

- **ID_FILE_MRU_FILE1**...**FILE16** A range of command IDs for the File MRU *list*.

  CWinApp::OnUpdateRecentFileMenu is a update command UI handler that is one of the more advanced uses of the **ON_UPDATE_COMMAND_UI** mechanism. In your menu resource, you need only define a single menu item with ID **ID_FILE_MRU_FILE1**. That menu item remains initially disabled.

  As the MRU list grows, more menu items are added to the list. The standard **CWinApp** implementation defaults to the standard limit of the four most recently used files. You can change the default by calling **CWinApp::LoadStdProfileSettings** with a larger or smaller value. The MRU list is stored in the application's .INI file. The list is loaded in your application's **InitInstance** function if you call **LoadStdProfileSettings**, and is saved when your application exits. The MRU update command UI handler also will convert absolute paths to relative paths for display on the file menu.

  CWinApp::OnOpenRecentFile is the **ON_COMMAND** handler that performs the actual command. It simply gets the file name from the MRU list and calls **CWinApp::OpenDocumentFile**, which does all the work of opening the file and updating the MRU list.

  Customization of this command handler is not recommended.

- ID_EDIT_CLEAR Clears the current selection

  Currently there is no standard implementation for this command. You must implement this for each **CView**-derived class.

  CEditView provides an implementation of this command using **CEdit::Clear**. The command is disabled if there is no current selection.

  If you choose to implement this command, we recommend you use this command ID.

- ID_EDIT_CLEAR_ALL Clears the entire document.

  Currently there is no standard implementation for this command. You must implement this for each **CView**-derived class.

  If you choose to implement this command, we recommend you use this command ID. See the MFC Tutorial sample SCRIBBLE for an example implementation.

- ID_EDIT_COPY Copies the current selection to the Clipboard.

  Currently there is no standard implementation for this command. You must implement this for each **CView**-derived class.

  CEditView provides an implementation of this command, which copies the currently selected text to the Clipboard as CF_TEXT using **CEdit::Copy**. The command is disabled if there is no current selection.

  If you choose to implement this command, we recommend you use this command ID.

- ID_EDIT_CUT Cuts the current selection to the Clipboard.

Currently there is no standard implementation for this command. You must implement this for each **CView**-derived class.

CEditView provides an implementation of this command, which cuts the currently selected text to the Clipboard as CF_TEXT using **CEdit::Cut**. The command is disabled if there is no current selection.

If you choose to implement this command, we recommend you use this command ID.

- ID_EDIT_FIND Begins the find operation, brings up the modeless find dialog.

  Currently there is no standard implementation for this command. You must implement this for each **CView**-derived class.

  CEditView provides an implementation of this command, which calls the implementation helper function **OnEditFindReplace** to use and store the previous find/replace settings in private implementation variables. The **CFindReplaceDialog** class is used to manage the modeless dialog for prompting the user.

  If you choose to implement this command, we recommend you use this command ID.

- ID_EDIT_PASTE Inserts the current Clipboard contents.

  Currently there is no standard implementation for this command. You must implement this for each **CView**-derived class.

  CEditView provides an implementation of this command, which copies the current Clipboard data replacing the selected text using **CEdit::Paste**. The command is disabled if there is no **CF_TEXT** in the Clipboard.

  COleClientDoc just provides a update command UI handler for this command. If the Clipboard does not contain an embeddable OLE item/object, the command will be disabled. You are responsible for writing the handler for the actual command to do the actual pasting. If your OLE application can also paste other formats, you should provide your own update command UI handler in your view or document (that is, somewhere before **COleClientDoc** in the command target routing).

  If you choose to implement this command, we recommend you use this command ID.

  For replacing the standard OLE implementation, use **COleClientItem::CanPaste**.

- ID_EDIT_PASTE_LINK Inserts a link from the current Clipboard contents.

  Currently there is no standard implementation for this command. You must implement this for each **CView**-derived class.

  COleDocument just provides a update command UI handler for this command. If the Clipboard does not contain linkable OLE item/object, the command will be disabled. You are responsible for writing the handler for the actual command to do the actual pasting. If your OLE application can also paste other formats, you should provide your own update command UI handler in your view or document (that is, somewhere before **COleDocument** in the command target routing).

  If you choose to implement this command, we recommend you use this command ID.

  For replacing the standard OLE implementation, use **COleClientItem::CanPasteLink**.

- ID_EDIT_PASTE_SPECIAL Inserts the current Clipboard contents with options.

  Currently there is no standard implementation for this command. You must implement this for each **CView**-derived class. MFC does not provide this dialog.

  If you choose to implement this command, we recommend you use this command ID.

- ID_EDIT_REPEAT Repeats the last operation.

  Currently there is no standard implementation for this command. You must implement this for each **CView**-derived class.

  CEditView provides an implementation of this command to repeat the last find operation. The private implementation variables for the last find are used. The command is disabled if a find cannot be attempted.

  If you choose to implement this command, we recommend you use this command ID.

- ID_EDIT_REPLACE Begins the replace operation, brings up the modeless replace dialog.

Currently there is no standard implementation for this command. You must implement this for each **CView**-derived class.

CEditView provides an implementation of this command, which calls the implementation helper function **OnEditFindReplace** to use and store the previous find/replace settings in private implementation variables. The **CFindReplaceDialog** class is used to manage the modeless dialog that prompts the user.

If you choose to implement this command, we recommend you use this command ID.

- ID_EDIT_SELECT_ALL Selects the entire document.

  Currently there is no standard implementation for this command. You must implement this for each **CView**-derived class.

  CEditView provides an implementation of this command, which selects all the text in the document. The command is disabled if there is no text to select.

  If you choose to implement this command, we recommend you use this command ID.

- ID_EDIT_UNDO Undoes the last operation.

  Currently there is no standard implementation for this command. You must implement this for each **CView**-derived class.

  CEditView provides an implementation of this command, using **CEdit::Undo**. The command is disabled if **CEdit::CanUndo** returns FALSE.

  If you choose to implement this command, we recommend you use this command ID.

- ID_EDIT_REDO Redoes the last operation.

  Currently there is no standard implementation for this command. You must implement this for each **CView**-derived class.

  If you choose to implement this command, we recommend you use this command ID.

- ID_WINDOW_NEW Opens another window on the active document.

  CMDIFrameWnd::OnWindowNew implements this powerful feature by using the document template of the current document to create another frame containing another view of the current document.

  Like most multiple document interface (MDI) Window menu commands, the command is disabled if there is no active MDI child window.

  Customization of this command handler is not recommended. If you wish to provide a command that creates additional views or frame windows, you will probably be better off inventing your own command. You can clone the code from **CMDIFrameWnd::OnWindowNew** and modify it to the specific frame and view classes of your liking.

- ID_WINDOW_ARRANGE Arranges icons at the bottom of an MDI window.

  CMDIFrameWnd implements this standard MDI command in an implementation helper function **OnMDIWindowCmd**. This helper maps command IDs to MDI Windows messages and can therefore share a lot of code.

  Like most MDI Window menu commands, the command is disabled if there is no active MDI child window.

  Customization of this command handler is not recommended.

- ID_WINDOW_CASCADE Cascades windows so they overlap.

  CMDIFrameWnd implements this standard MDI command in an implementation helper function **OnMDIWindowCmd**. This helper maps command IDs to MDI Windows messages and can therefore share a lot of code.

  Like most MDI Window menu commands, the command is disabled if there is no active MDI child window.

  Customization of this command handler is not recommended.

- ID_WINDOW_TILE_HORZ Tiles windows horizontally.

  This command is implemented in **CMDIFrameWnd** just like **ID_WINDOW_CASCADE**, except a different MDI Windows message is used for the operation.

  You should pick the default tile orientation for your application. You can do this by changing the ID for the Window "Tile"

menu item to either **ID_WINDOW_TILE_HORZ** or **ID_WINDOW_TILE_VERT**.

- ID_WINDOW_TILE_VERT Tiles windows vertically.

  This command is implemented in **CMDIFrameWnd** just like **ID_WINDOW_CASCADE**, except a different MDI Windows message is used for the operation.

  You should pick the default tile orientation for your application. You can do this by changing the ID for the Window "Tile" menu item to either **ID_WINDOW_TILE_HORZ** or **ID_WINDOW_TILE_VERT**.

- ID_WINDOW_SPLIT Keyboard interface to splitter.

  CView handles this command for the **CSplitterWnd** implementation. If the view is part of a splitter window, this command will delegate to the implementation function **CSplitterWnd::DoKeyboardSplit**. This will place the splitter in a mode that will allow keyboard users to split or unsplit a splitter window.

  This command is disabled if the view is not in a splitter.

  Customization of this command handler is not recommended.

- ID_APP_ABOUT Invokes the About dialog box.

  There is no standard implementation for an application's About box. The default AppWizard-created application will create a custom dialog class for your application and use it as your About box. AppWizard will also write the trivial command handler which handles this command and invokes the dialog.

  You will almost always implement this command.

- ID_APP_EXIT Exit the application.

  CWinApp::OnAppExit handles this command by sending a **WM_CLOSE** message to the application's main window. The standard shutting down of the application (prompting for dirty files and so on) is handled by the **CFrameWnd** implementation.

  Customization of this command handler is not recommended. Overriding **CWinApp::SaveAllModified** or the **CFrameWnd** closing logic is recommended.

  If you choose to implement this command, we recommend you use this command ID.

- ID_HELP_INDEX Lists Help topics from .HLP file.

Note
You must connect this to your **CWinApp**-derived class's message map to enable this functionality.

  CWinApp::OnHelpIndex handles this command by trivially calling **CWinApp::WinHelp**.

  Customization of this command handler is not recommended.

- ID_HELP_USING Displays help on how to use Help.

Note
You must connect this to your **CWinApp**-derived class's message map to enable this functionality.

  CWinApp::OnHelpUsing handles this command by trivially calling **CWinApp::WinHelp**.

  Customization of this command handler is not recommended.

- ID_CONTEXT_HELP Enters SHIFT-F1 help mode.

Note
You must connect this to your **CWinApp**-derived class's message map to enable this functionality.

  CWinApp::OnContextHelp handles this command by setting the help mode cursor, entering a modal loop and waiting for the user to select a window to get help on. Please refer to Technical Note 28 for more details on the MFC Help implementation.

Customization of this command handler is not recommended.

- ID_HELP Gives help on the current context

Note
You must connect this to your **CWinApp**-derived class's message map to enable this functionality.

CWinApp::OnHelp handles this command by getting the right help context for the current application context. This handles simple F1 help, help on message boxes and so on. Please refer to Technical Note 28 for more details on the MFC help implementation.

Customization of this command handler is not recommended.

- ID_DEFAULT_HELP Displays default help for context

Note
You must connect this to your **CWinApp**-derived class's message map to enable this functionality.

This command is usually mapped to **CWinApp::OnHelpIndex**.

A different command handler can be provided if a distinction between default Help and the Help index is desired.

- ID_NEXT_PANE Goes to next pane

CView handles this command for the **CSplitterWnd** implementation. If the view is part of a splitter window, this command will delegate to the implementation function **CSplitterWnd::OnNextPaneCmd**. This will move the active view to the next pane in the splitter.

This command is disabled if the view is not in a splitter or there is no next pane to go to.

Customization of this command handler is not recommended.

- ID_PREV_PANE Goes to previous pane

CView handles this command for the **CSplitterWnd** implementation. If the view is part of a splitter window, this command will delegate to the implementation function **CSplitterWnd::OnNextPaneCmd**. This will move the active view to the previous pane in the splitter.

This command is disabled if the view is not in a splitter or there is no previous pane to go to.

Customization of this command handler is not recommended.

- ID_OLE_INSERT_NEW Inserts a new OLE object

Currently there is no standard implementation for this command. You must implement this for your **CView**-derived class to insert a new OLE item/object at the current selection.

All OLE client applications should implement this command. AppWizard, with the OLE option, will create a skeleton implementation of **OnInsertObject** in your view class that you will have to complete.

See the MFC OLE sample OCLIENT example for a complete implementation of this command.

- ID_OLE_EDIT_LINKS Edits OLE links

COleDocument handles this command by using the MFC-provided implementation of the standard OLE links dialog. The implementation of this dialog is accessed through the **COleLinksDialog** class. If the current document does not contain any links, the command is disabled.

Customization of this command handler is not recommended.

- ID_OLE_VERB_FIRST...LAST An ID range for OLE verbs

COleDocument uses this command ID range for the verbs supported by the currently selected OLE item/object. This must be a range since a given OLE item/object type can support zero or more custom verbs. In your application's menu, you should have one menu item with the ID of **ID_OLE_VERB_FIRST**. When the program is run, the menu will be updated with the appropriate menu verb description (or pop-up menu with many verbs). The management of the OLE menu is handled by **AfxOleSetEditMenu**, done in the update command UI handler for this command.

There are no explicit command handlers for handling each of the command ID in this range. **COleDocument::OnCmdMsg** is overridden to trap all command IDs in this range, turn them into zero-based verb numbers, and launch the server for that verb (using **COleClientItem::DoVerb**).

Customization or other use of this command ID range is not recommended.

- ID_VIEW_TOOLBAR Toggles the toolbar on and off

  CFrameWnd handles this command and the update-command UI handler to toggle the visible state of the toolbar. The toolbar must be a child window of the frame with child window ID of **AFX_IDW_TOOLBAR**. The command handler actually toggles the visibility of the toolbar window. **CFrameWnd::RecalcLayout** is used to redraw the frame window with the toolbar in its new state. The update-command UI handler checks the menu item when the toolbar is visible.

  Customization of this command handler is not recommended. If you wish to add additional toolbars, you will want to clone and modify the command handler and the update-command UI handler for this command.

- ID_VIEW_STATUS_BAR Toggles the status bar on and off

  This command is implemented in **CFrameWnd** just like **ID_VIEW_TOOLBAR**, except a different child window ID (**AFX_IDW_STATUS_BAR**) is used.

Update-Only Command Handlers

Several standard command IDs are used as indicators in status bars. These use the same update-command UI handling mechanism to display their current visual state during application idle time. Since they can't be selected by the user (that is, you cannot push a status bar pane), then it makes no sense to have an **ON_COMMAND** handler for these command IDs.

- **ID_INDICATOR_CAPS** : CAP lock indicator.
- **ID_INDICATOR_NUM** : NUM lock indicator.
- **ID_INDICATOR_SCRL** : SCRL lock indicator.
- **ID_INDICATOR_KANA** : KANA lock indicator (applicable only to Japanese systems).

All three of these are implemented in **CFrameWnd::OnUpdateKeyIndicator**, an implementation helper that uses the command ID to map to the appropriate Virtual Key. A common implementation enables or disables (for status panes disabled = no text) the **CCmdUI** object depending on whether the appropriate Virtual Key is currently locked.

Customization of this command handler is not recommended.

- **ID_INDICATOR_EXT : EXT**ended select indicator.
- **ID_INDICATOR_OVR : OV**e**R**strike indicator.
- **ID_INDICATOR_REC : REC**ording indicator.

Currently there is no standard implementation for these indicators.

If you choose to implement these indicators, we recommend you use these indicator IDs and maintaining the ordering of the indicators in your status bar (that is, in this order: EXT, CAP, NUM, SCRL, OVR, REC).

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN023: Standard MFC Resources

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes the standard resources provided with and needed by the MFC library.

Standard Resources

MFC offers two categories of predefined resources that you can use in your application: clip-art resources and standard framework resources.

Clip-art resources are additional resources that the framework does not depend on, but which you might want to add to your application's user interface. The following clip-art resources are contained in the MFC General sample CLIPART:

- **COMMON.RC**: A single file of resources containing:
  - A large collection of icons that represent a variety of business and data-processing tasks.
  - Several common cursors (see also AFXRES.RC).
  - A toolbar bitmap containing several toolbar buttons.
  - The bitmap and icon resources used by COMMDLG.DLL.
- **INDICATE.RC**: Contains string resources for the status-bar key-state indicators, such as "CAP" for Caps Lock.
- **PROMPTS.RC:** Contains menu-prompt string resources for each of the predefined commands, such as "Create a new document" for **ID_FILE_NEW**.
- **COMMDLG.RC**: An Visual C++ compatible .RC file containing the standard COMMDLG dialog templates.

Standard framework resources are resources with AFX-defined IDs that the framework depends on for internal implementations. You will rarely need to change these AFX-defined resources. If you do, you should follow the procedure outlined below.

The following framework resources are contained in the MFC\INCLUDE directory:

- **AFXRES.RC**: Common resources used by the framework.
- **AFXPRINT.RC**: Resources specific to printing.
- **AFXOLECL.RC**: Resources specific to OLE client applications.
- **AFXOLESV.RC**: Resources specific to full OLE server applications.

Using Clip-Art Resources

To use a clip-art binary resource

1. Open your application's resource file in Visual C++.
2. Open COMMON.RC, which contains all the binary clip-art resources. This may take some time because the COMMON.RC file is compiled.
3. Press CTRL and drag to copy the resource or resources you want from COMMON.RC to your application's resource file.
4. If you want, rename the resource from a string name to a symbol.

To use other clip-art resources, the steps are as above except you open the appropriate .RC file instead of COMMON.RC. Renaming the resource ID is not needed since the clip-art resources will already be assigned the appropriate symbols for you.

Note

Be careful not to unintentionally move resources out of COMMON.RC permanently. You can avoid this by CTRL-dragging (copy) rather than dragging (move the resources). You can also avoid this by always responding with "No" when you are asked whether you want to save the changes to COMMON.RC.

Note

The .RC resource files have a special **TEXTINCLUDE** resource in them that will prevent you from accidentally saving on top of the standard .RC files.

Customizing Standard Framework Resources

Standard framework resources are usually #include'd in your application's .RC file. AppWizard will generate a .RC file that includes the appropriate standard framework resources depending on which AppWizard options you choose. You can review, add, or remove which resources are #include'd by using Visual C++'s **Set Includes** command in the **Resource** menu of Visual C++ and looking at the "Compile-Time Directives" edit item. For example:

```
#include "afxres.rc"
#include "afxprint.rc"
```

The most common case of customizing standard framework resources is adding or removing additional includes for printing, OLE Client, and OLE Server support.

In some rare case you may wish to customize the contents of the standard framework resources for your particular application, not just add and remove the entire file.

To customize the contents of a standard resource file

1. Open your application's resource file in Visual C++.
2. Using the Resource Set Includes command, remove the #include for the standard .rc file you want to customize (for example, to customize the print-preview toolbar, remove the #include "afxprint.rc" line).
3. Open the appropriate standard resources files in MFC\INCLUDE (for example, MFC\INCLUDE\AFXPRINT.RC)
4. Copy all of the resources from the standard .RC file to your application resource file.
5. This is an all or none proposition. Either you #include all of the resources from the .RC file in MFC\INCLUDE or you have a (customized) copy of these resources in your application resource file.
6. Modify the copy of the standard resources in your application resource file.

Note

Be especially careful not to modify the resources directly in the standard .RC files.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN024: MFC-Defined Messages and Resources

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes the internal Windows messages and resource formats used by MFC. This information explains the implementation of the framework, and will assist you in debugging your application. For the adventurous, even though all this information is officially unsupported, you may use some of this information for advanced implementations.

This note contains MFC private implementation details; all the contents are subject to change in the future. MFC private Windows messages have meaning in the scope of one application only but will change in the future to contain system-wide messages.

The range of MFC private Windows messages and resource types are in the reserved "system" range set aside by Microsoft Windows. Currently not all numbers in the ranges are used and, in the future, new numbers in the range may be used. The currently used numbers may be changed.

MFC private Windows messages are in the range 0x360->0x37F.

MFC private resource types are in the range 0xF0->0xFF.

MFC Private Windows Messages

These Windows messages are used in place of C++ virtual functions where relatively loose coupling is required between window objects and where a C++ virtual function would not be appropriate.

These private Windows messages and associated parameter structures are declared in the MFC private header 'AFXPRIV.H'. Be warned that any of your code that includes this header may be relying on undocumented behavior and will likely break in future versions of MFC.

In the rare case of needing to handle one of these messages, you should use the **ON_MESSAGE** message map macro and handle the message in the generic LRESULT/WPARAM/LPARAM format.

WM_QUERYAFXWNDPROC

This message is sent to a window that is being created. This is sent very early in the creation process as a method of determining if the WndProc is **AfxWndProc. AfxWndProc** returns 1.

wParam	Not used
lParam	Not used
returns	1 if processed by **AfxWndProc**

WM_SIZEPARENT

This message is sent by a frame window to its immediate children during resizing (**CFrameWnd::OnSize** calls **CFrameWnd::RecalcLayout** which calls **CWnd::RepositionBars**) to reposition the control bars around the side of the frame. The **AFX_SIZEPARENTPARAMS** structure contains the current available client rectangle of the parent and a HDWP (which may be NULL) with which to call **DeferWindowPos** to minimize repainting.

wParam	Not used
lParam	Address of an **AFX_SIZEPARENTPARAMS** structure
returns	Not used (0)

Ignoring the message indicates that the window doesn't take part in the layout.

WM_SETMESSAGESTRING

This message is sent to a frame window to ask it to update the message line in the status bar. Either a string ID or a LPCSTR can be specified (but not both).

wParam	String ID (or zero)
lParam	LPCSTR for the string (or NULL)
returns	Not used (0)

WM_IDLEUPDATECMDUI

This message is sent in idle time to implement the idle-time update of update-command UI handlers. If the window (usually a control bar) handles the message, it creates a **CCmdUI** object (or an object of a derived class) and call **CCmdUI::DoUpdate** for each of the "items" in the window. This will in turn check for an **ON_UPDATE_COMMAND_UI** handler for the objects in the command-handler chain.

wParam	BOOL bDisableIfNoHandler
lParam	Not used (0)
returns	Not used (0)

bDisableIfNoHandler is nonzero to disable the UI object if there is neither an **ON_UPDATE_COMMAND_UI** nor an **ON_COMMAND** handler.

WM_EXITHELPMODE

This message is posted to a **CFrameWnd** that to exit context sensitive help mode. The receipt of this message terminates the modal loop started by **CFrameWnd::OnContextHelp.**

wParam	Not used (0)
lParam	Not used (0)
returns	Not used

WM_INITIALUPDATE

This message is sent by the document template to all descendants of a frame window when it is safe for them to do their initial update. It maps to a call to **CView::OnInitialUpdate** but can be used in other **CWnd**-derived classes for other one-shot updating.

wParam	Not used (0)
lParam	Not used (0)
returns	Not used (0)

WM_RECALCPARENT

This message is sent by a view to its parent window (obtained via **GetParent**) to force a layout recalculation (usually, the parent will call **RecalcLayout**). This is used in OLE server applications where it is necessary for the frame to grow in size as the view's total size grows.

If the parent window processes this message it should return TRUE and fill the RECT passed in lParam with the new size of the client area. This is used in **CScrollView** to properly handle scrollbars (place then on the outside of the window when they are added) when a server object is in-place activated.

wParam	Not used (0)
lParam	LPRECT rectClient, may be NULL
returns	TRUE if new client rectangle returned, FALSE otherwise

WM_SIZECHILD

This message is sent by **COleResizeBar** to its owner window (via **GetOwner**) when the user resizes the resize bar with the resize handles. **COleIPFrameWnd** responds to this message by attempting to reposition the frame window as the user has requested.

The new rectangle, given in client coordinates relative to the frame window which contains the resize bar, is pointed at by lParam.

wParam	Not used (0)
lParam	LPRECT rectNew
returns	Not used (0)

WM_DISABLEMODAL

This message is sent to all pop-up windows owned by a frame window that is being deactivated. The frame window uses the result to determine whether or not to disable the pop-up window.

You can use this to perform special processing in your pop-up window when the frame enters a modal state or to keep certain pop-up windows from getting disabled. Tooltips use this message to destroy themselves when the frame window goes into a modal state, for example.

wParam	Not used (0)
lParam	Not used (0)
returns	Non-zero to **NOT** disable the window, 0 indicates the window will be disabled

WM_FLOATSTATUS

This message is sent to all pop-up windows owned by a frame window when the frame is either activated or deactivated by another top-level frame window. This is used by the implementation of **MFS_SYNCACTIVE** in **CMiniFrameWnd**, to keep the activation of these pop-up windows in sync with the activation of the top level frame window.

wParam	Is one of the following values: **FS_SHOW** **FS_HIDE** **FS_ACTIVATE** **FS_DEACTIVATE** **FS_ENABLEFS_DISABLE** **FS_SYNCACTIVE**
lParam	Not used (0)

The return value should be non-zero if **FS_SYNCACTIVE** is set and the window syncronizes its activation with the parent frame. **CMiniFrameWnd** returns non-zero when the style is set to **MFS_SYNCACTIVE.**

For more information, see the implementation of **CMiniFrameWnd**.

WM_ACTIVATETOPLEVEL

This message is sent to a top-level window when a window in its "top-level group" is either activated or deactivated. A window is part of a top-level group if it is a top-level window (no parent or owner), or it is owned by such a window. This message is similar in use to **WM_ACTIVATEAPP,** but works in situations where windows belonging to different processes are mixed in a single window hierarchy (common in OLE applications).

WM_COMMANDHELP, WM_HELPHITTEST, WM_EXITHELPMODE

These messages are used in the implementation of context-sensitive Help. Please refer to Technical Note 28 for more information.

MFC Private Resource Formats

Currently, MFC defines two private resource formats: **RT_TOOLBAR** and **RT_DLGINIT**.

RT_TOOLBAR Resource Format

The default toolbar supplied by AppWizard is based on an **RT_TOOLBAR** custom resource, which was introduced in MFC 4.0. You can edit this resource using the Toolbar editor.

RT_DLGINIT Resource Format

One MFC private resource format is used to store extra dialog initialization information. This includes the initial strings stored in a combo box. The format of this resource is not designed to be manually edited, but is handled by Visual C++.

Visual C++ and this **RT_DLGINIT** resource are not required to use the related features of MFC since there are API alternative to using the information in the resource. Using Visual C++ makes it much easier to write, maintain, and translate your application in the long run.

The basic structure of a **RT_DLGINIT** resource is as follows:

```
+---------------+                   \
| Control ID    |   UINT            |
+---------------+                   |
| Message #     |   UINT            |
+---------------+                   |
|length of data |   DWORD           |
+---------------+                   |   Repeated
|   Data        |   Variable Length |   for each control
|   ...         |   and Format      |   and message
+---------------+                   /
|     0         |   BYTE
+---------------+
```

A repeated section contains the control ID to send the message to, the Message # to send (a normal Windows message) and a variable length of data. The Windows message is sent in a form:

```
SendDlgItemMessage(<Control ID>, <Message #>, 0, &<Data>);
```

This is a very general format, allowing any Windows messages and data content. The Visual C++ resource editor and MFC only support a limited subset of Windows messages: CB_ADDSTRING for the initial list-choices for combo boxes (the data is a text string).

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN025: Document, View, and Frame Creation

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes the creation and ownership issues for WinApps, DocTemplates, Documents, Frames and Views.

WinApp

There is one **CWinApp** object in the system.

It is statically constructed and initialized by the framework's internal implementation of **WinMain**. You must derive from **CWinApp** to do anything useful (exception: extension DLLs should not have a **CWinApp** instance — initialization is done in **DllMain** instead).

The one **CWinApp** object owns a list of document templates (a **CPtrList**). There is one or more document template per application. DocTemplates are usually loaded from the resource file (that is, a string array) in **CWinApp::InitInstance**.

```
pTemplate = new CDocTemplate(IDR_MYDOCUMENT, ...);
AddDocTemplate(pTemplate);
```

The one **CWinApp** object owns all frame windows in the application. The main frame window for the application should be stored in **CWinApp::m_pMainWnd**; usually you set **m_pMainWnd** in the **InitInstance** implementation if you have not let AppWizard do it for you. For single document interface (SDI) this is one **CFrameWnd** that serves as the main application frame window as well as the only document frame window. For multiple document interface (MDI) this is an MDI-Frame (class **CMDIFrameWnd**) that serves as the main application frame window that contains all the child **CFrameWnd**s. Each child window is of class **CMDIChildWnd** (derived from **CFrameWnd**) and serves as one of potentially many document frame windows.

DocTemplates

The **CDocTemplate** is the creator and manager of documents. It owns the documents that it creates. If your application uses the resource-based approach described below, it will not need to derive from **CDocTemplate**.

For an SDI application, the class **CSingleDocTemplate** keeps track of one open document. For an MDI application, the class **CMultiDocTemplate** keeps a list (a **CPtrList**) of all the currently open documents created from that template. **CDocTemplate::AddDocument** and **CDocTemplate::RemoveDocument** provide the virtual member functions for adding or removing a document from the template. **CDocTemplate** is a friend of **CDocument** so we can set the protected **CDocument::m_pDocTemplate** back pointer to point back to the doc template that created the document.

CWinApp handles the default **OnFileOpen** implementation, which will in turn query all the doc templates. The implementation includes looking for already open documents and deciding what format to open new documents in.

CDocTemplate manages the UI binding for documents and frames.

CDocTemplate keeps a count of the number of unnamed documents.

CDocument

A **CDocument** is owned by a **CDocTemplate**.

Documents have a list of currently open views (derived from **CView**) that are viewing the document (a **CPtrList**).

Documents do not create/destroy the views, but they are attached to each other after they are created. When a document is closed (that is, through File/Close), all attached views will be closed. When the last view on a document is closed (that is, Window/Close) the document will be closed.

The **CDocument::AddView**, **RemoveView** interface is used to maintain the view list. **CDocument** is a friend of **CView** so we can set the **CView::m_pDocument** back pointer.

CFrameWnd

A **CFrameWnd** (also known as a frame) plays the same role as in MFC 1.0, but now the **CFrameWnd** class is designed to be used in many cases without deriving a new class. The derived classes **CMDIFrameWnd** and **CMDIChildWnd** are also enhanced so many standard commands are already implemented.

The **CFrameWnd** is responsible for creating windows in the client area of the frame. Normally there is one main window filling the client area of the frame.

For an MDI-Frame window, the client area is filled with the MDICLIENT control which is in turn the parent of all the MDI-Child frame windows. For an SDI-Frame window or an MDI-Child frame window, the client area is usually filled with a **CView**-derived window object. In the case of **CSplitterWnd**, the client area of the view is filled with the **CSplitterWnd** window object, and the **CView**-derived window objects (one per split pane) are created as child windows of the **CSplitterWnd**.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN026: DDX and DDV Routines

Note

The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes the dialog data exchange (DDX) and dialog data validation (DDV) architecture. It also describes how you write a DDX_ or DDV_ procedure and how you can extend ClassWizard to use your routines.

Overview of Dialog Data Exchange

All dialog data functions are done with C++ code. There are no special resources or magic macros. The heart of the mechanism is a virtual function that is overridden in every dialog class that does dialog data exchange and validation. It is always found in this form:

```
void CMyDialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);    // call base class

    //{{AFX_DATA_MAP(CMyDialog)
        <data_exchange_function_call>
        <data_validation_function_call>
    //}}AFX_DATA_MAP
}
```

The special format AFX comments allow ClassWizard to locate and edit the code within this function. Code that is not compatible with ClassWizard should be placed outside of the special format comments.

In the above example, <data_exchange_function_call> is in the form:

```
    DDX_Custom(pDX, nIDC, field);
```

and <data_validation_function_call> is optional and is in the form:

```
    DDV_Custom(pDX, field, ...);
```

More than one DDX_/DDV_ pair may be included in each **DoDataExchange** function.

See 'afxdd_.h' for a list of all the dialog data exchange routines and dialog data validation routines provided with MFC.

Dialog data is just that: member data in the **CMyDialog** class. It is not stored in a struct or anything similar.

Notes

Although we call this "dialog data," all features are available in any class derived from **CWnd** and are not limited to just dialogs.

Initial values of data are set in the standard C++ constructor, usually in a block with `//{{AFX_DATA_INIT` and `//}}AFX_DATA_INIT` comments.

CWnd::UpdateData is the operation that does the initialization and error handling around the call to **DoDataExchange**.

You can call **CWnd::UpdateData** at any time to perform data exchange and validation. By default **UpdateData**(TRUE) is called in the default **CDialog::OnOK** handler and **UpdateData**(FALSE) is called in the default **CDialog::OnInitDialog**.

The DDV_ routine should immediately follow the DDX_ routine for that *field*.

How Does It Work?

You do not need to understand the following in order to use dialog data. However, understanding how this works behind the

scenes will help you write your own exchange or validation procedure.

The **DoDataExchange** member function is much like the **Serialize** member function - it is responsible for getting or setting data to/from an external form (in this case controls in a dialog) from/to member data in the class. The *pDX* parameter is the context for doing data exchange and is similar to the **CArchive** parameter to **CObject::Serialize**. The *pDX* (a **CDataExchange** object) has a direction flag much like **CArchive** has a direction flag:

- If **!m_bSaveAndValidate**, then load the data state into the controls.
- If **m_bSaveAndValidate**, then set the data state from the controls.

Validation only occurs when **m_bSaveAndValidate** is set. The value of **m_bSaveAndValidate** is determined by the BOOL parameter to **CWnd::UpdateData**.

There are three other interesting **CDataExchange** members:

- **m_pDlgWnd**: The window (usually a dialog) that contains the controls. This is to prevent callers of the DDX_ and DDV_ global functions from having to pass 'this' to every DDX/DDV routine.
- **PrepareCtrl**, and **PrepareEditCtrl**: Prepares a dialog control for data exchange. Stores that control's handle for setting the focus if a validation fails. **PrepareCtrl** is used for nonedit controls and **PrepareEditCtrl** is used for edit controls.
- **Fail**: Called after bringing up a message box alerting the user to the input error. This routine will restore the focus to the last control (the last call to **PrepareCtrl**/**PrepareEditCtrl**) and throw an exception. This member function may be called from both DDX_ and DDV_ routines.

User Extensions

There are several ways to extend the default DDX/DDV mechanism. You can:

- Add new data types.

```
CTime
```

- Add new exchange procedures (DDX_???).

```
void PASCAL DDX_Time(CDataExchange* pDX, int nIDC, CTime& tm);
```

- Add new validation procedures (DDV_???).

```
void PASCAL DDV_TimeFuture(CDataExchange* pDX, CTime tm, BOOL bFuture);
// make sure time is in the future or past
```

- Pass arbitrary expressions to the validation procedures.

```
DDV_MinMax(pDX, age, 0, m_maxAge);
```

Note

Such arbitrary expressions cannot be edited by ClassWizard and therefore should be moved outside of the special form at comments (//{{AFX_DATA_MAP(CMyClass)).

Have the **DoDialogExchange** member function include conditionals or any other valid C++ statements with intermixed exchange and validation function calls.

```
//{{AFX_DATA_MAP(CMyClass)
DDX_Check(pDX, IDC_SEX, m_bFemale);
DDX_Text(pDX, IDC_EDIT1, m_age);
//}}AFX_DATA_MAP
if (m_bFemale)
    DDV_MinMax(pDX, age, 0, m_maxFemaleAge);
else
```

```
    DDV_MinMax(pDX, age, 0, m_maxMaleAge);
```

Note

As shown above, such code cannot be edited by ClassWizard and should be used only outside of the special format comments.

ClassWizard Support

ClassWizard supports a subset of DDX/DDV customizations by allowing you to integrate your own DDX_ and DDV_ routines into the ClassWizard user interface. Doing this is only cost beneficial if you plan to reuse particular DDX and DDV routines in a project or in many projects.

To do this, special entries are made in DDX.CLW (previous versions of Visual C++ stored this information in APSTUDIO.INI) or in your project's .CLW file. The special entries can be entered either in the [General Info] section of your project's .CLW file or in the [ExtraDDX] section of the DDX.CLW file in the \Program Files\Microsoft Visual Studio\Visual C++\bin directory. You may need to create the DDX.CLW file if it doesn't already exist. If you plan to use the custom DDX_/DDV_ routines only in a certain project, add the entries to the [General Info] section of your project .CLW file instead. If you plan to use the routines on many projects, add the entries to the [ExtraDDX] section of DDX.CLW.

The general format of these special entries is:

```
ExtraDDXCount=n
```

where n is the number of ExtraDDX? lines to follow

```
ExtraDDX?=<keys>;<vb-keys>; <prompt>; <type>; <initValue>; <DDX_Proc>
[;<DDV_Proc>; <prompt1>; <arg1>; [<prompt2>; <fmt2>]]
```

where ? is a number 1 – n indicating which DDX type in the list that is being defined.

Each field is delimited by a ';' character. The fields and their purpose are described below.

<keys>

= list of single characters indicating for which dialog controls this variable type is allowed.

E = edit

C = two-state check box

c = tri-state check box

R = first radio button in a group

L = nonsorted list box

l = sorted list box

M = combo box (with edit item)

N = nonsorted drop list

n = sorted drop list

1 = if the DDX insert should be added to head of list (default is add to tail) This is generally used for DDX routines that transfer the 'Control' property.

<vb-keys>

This field is used only in the 16-bit product for VBX controls (VBX controls are not supported in the 32-bit product)

<prompt>

String to place in the Property combo box (no quotes)

<type>

Single identifier for type to emit in the header file. In our example above with DDX_Time, this would be set to CTime.

<vb-keys>

Not used in this version and should always be empty

<initValue>

Initial value — 0 or blank. If it is blank, then no initialization line will be written in the //{{AFX_DATA_INIT section of the implementation file. A blank entry should be used for C++ objects (such as **CString**, **CTime**, and so on) that have constructors that guarantee correct initialization.

<DDX_Proc>

Single identifier for the DDX_ procedure. The C++ function name must start with "DDX_," but don't include "DDX_" in the <DDX_Proc> identifier. In the example above, the <DDX_Proc> identifier would be Time. When ClassWizard writes the function call to the implementation file in the {{AFX_DATA_MAP section, it appends this name to DDX_, thus arriving at DDX_Time.

<comment>

Comment to show in dialog for variable with this DDX. Place any text you would like here, and usually provide something that describes the operation performed by the DDX/DDV pair.

<DDV_Proc>

The DDV portion of the entry is optional. Not all DDX routines have corresponding DDV routines. Often, it is more convenient to include the validation phase as an integral part of the transfer. This is often the case when your DDV routine doesn't require any parameters, because ClassWizard does not support DDV routines without any parameters.

<arg>

Single identifier for the DDV_ procedure. The C++ function name must start with "DDV_", but do not include "DDX_" in the <DDX_Proc> identifier.

followed by 1 or 2 DDV args:

<promptX>

string to place above the edit item (with & for accelerator)

<fmtX>

format character for the arg type, one of

d = int

u = unsigned

D = long int (that is, long)

U = long unsigned (that is, DWORD)

f = float

F = double

s = string

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN027: Emulation Support for Visual Basic Custom Controls

This technical note, which previously discussed VBX custom controls, is now obsolete.

ActiveX Controls (OCX) is a new custom control architecture for 32-bit and 64-bit systems. Building and using OLE Controls is fully supported in this version of Visual C++. For more information, see MFC ActiveX Controls.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN028: Context-Sensitive Help Support

This note describes the rules for assigning Help contexts IDs (that is, topic numbers) and other help issues in MFC. Context sensitive help support requires the help compiler that is available in Visual C++.

Note

In addition to implementing context-sensitive help using WinHelp, MFC also supports using HTML Help. For more informatio n on this support and programming with HTML Help, see HTML Help: Context-Sensitive Help for Your Programs.

Types of Help Supported

There are two types of context-sensitive help implemented in Windows applications. The first, referred to as "F1 Help" involves launching WinHelp with the appropriate context based on the currently active object. The second is SHIFT+F1 mode. In this mode, the mouse cursor changes to the help cursor (a combination arrow + question mark), and the user proceeds to click the object on which they want help. At that point, WinHelp is launched to give help for the object on which the user clicked.

The Microsoft Foundation Classes implement both of these forms of help. In addition, the framework supports two simple help commands, Help Index and Using Help.

Help Files

The Microsoft Foundation classes assume a single Help file. That Help file must have the same name and path as the application (.EXE -> .HLP).

This is a public **CWinApp** member variable named **m_pszHelpFilePath** that the user can change if desired.

Help Context Ranges

```
0x00000000 - 0x0000FFFF : user defined
0x00010000 - 0x0001FFFF : commands (menus/command buttons)
   0x00010000 + ID_
   (note: 0x18000-> 0x1FFFF is practical range since command IDs are >=0x8000)
0x00020000 - 0x0002FFFF : windows and dialogs
   0x00020000 + IDR_
   (note: 0x20000-> 0x27FFF is practical range since IDRs are <= 0x7FFF)
0x00030000 - 0x0003FFFF : error messages (based on error string ID)
   0x00030000 + IDP_
0x00040000 - 0x0004FFFF : special purpose (non-client areas)
   0x00040000 + HitTest area
0x00050000 - 0x0005FFFF : controls (those that are not commands)
   0x00040000 + IDW_
```

These rules are hard-coded into the default implementation of the Microsoft Foundation classes. They can be overridden by providing different implementations of the various Help-related member functions.

Simple "Help" Commands

There are two simple Help commands that are implemented by the Microsoft Foundation Classes:

- ID_HELP_INDEX which is implemented by **CWinApp::OnHelpIndex**
- ID_HELP_USING which is implemented by **CWinApp::OnHelpUsing**

These two commands simply show the Help index for the application and show the user help on using the WinHelp program, respectively.

Context-Sensitive Help (F1 Help)

This is the first form of context-sensitive Help (usually referred to as F1 Help). The user presses F1 to get help on the task (the active window or menu item). No special "help mode" is involved.

The F1 key is usually translated to a command with an ID of **ID_HELP** by an accelerator placed into the main window's accelerator table. The **ID_HELP** command may also be generated by a button with an ID of **ID_HELP** on the main window or dialog box. Also, when a menu or a dialog box is active and the user presses F1, the keystroke is hard-coded to translate into an **ID_HELP** command.

However the **ID_HELP** command is generated, it is routed as a normal command until it reaches a command handler (for more information on the Microsoft Foundation classes command-routing architecture, refer to Technical Note 21.) If the application is Help enabled, the **ID_HELP** command will be handled by the **CWinApp::OnHelp** function. Since the default command routing is not adequate for determining the most specific context the command is instead always routed to the application object and then undergoes custom routing for Help.

CWinApp::OnHelp attempts to launch WinHelp in the following order:

1. Checks for an active **AfxMessageBox** call with a Help ID. If a message box is currently active, WinHelp is launched with the context appropriate to that message box.
2. If no message box is active, **CWinApp::OnHelp** sends a **WM_COMMANDHELP** (a message private to the Microsoft Foundation classes) to the active window. If that window does not respond by launching WinHelp, the same message is then sent to the parent of that window, until the message is processed or the current window is a top-level window (and therefore does not have a parent window).
3. If the message remains unprocessed, then the default Help is invoked. This is done by sending a **ID_DEFAULT_HELP** command to the main window. This command is generally mapped to **CWinApp::OnHelpIndex**.

To globally override the ID base values (0x10000 for commands, 0x20000 for resources such as dialogs, and so on), the application should override **CWinApp::WinHelp**.

To override this functionality and the way that a Help context is determined, an application should handle the **WM_COMMANDHELP** message (see below). You may wish to provide more specific Help routing than the framework provides, as it only goes as deep as the current MDI child window. You may also want to provide more specific help for a particular window or dialog, perhaps based on the current internal state of that object or the active control within the dialog.

WM_COMMANDHELP

```
afx_msg LRESULT CWnd::OnCommandHelp(WPARAM, LPARAM lParam)
```

WM_COMMANDHELP is an MFC private Windows message that is received by the active window when Help is requested. When the window receives this message, it may call **CWinApp::WinHelp** with context that matches the window's internal state.

lParam

contains the currently available Help context. lParam is zero if no Help context has been determined yet. An implementation of **OnCommandHelp** can use the context ID in lParam to determine a "better" context or can just pass it to **CWinApp::WinHelp**.

wParam

is not used and will be zero.

If the **OnCommandHelp** function calls **CWinApp::WinHelp**, it should return TRUE. Returning TRUE stops the routing of this command to other classes (base classes) and to other windows.

Help Mode (Shift+F1 Help)

This is the second form of context-sensitive Help. Generally, this mode is entered by pressing SHIFT+F1 or via the menu/toolbar. It is implemented as a command (**ID_CONTEXT_HELP**). The message filter hook is not used to translate this command while a modal dialog box or menu is active, therefore this command is only available to the user when the application is executing the main message pump (**CWinApp::Run**).

After entering this mode, the Help mouse cursor is displayed over all areas of the application, even if the application would normally display its own cursor for that area (such as the sizing border around the window). The user is able to use the mouse or keyboard to select a command. Instead of executing the command, Help on that command is displayed. Also, the user can click a visible object on the screen, such as a button on the toolbar, and Help will be displayed for that object. This mode of Help is provided by **CWinApp::OnContextHelp**.

During the execution of this loop, all keyboard input is inactive, except for keys that access the menu. Also, command translation is still performed via **PreTranslateMessage** to allow the user to press an accelerator key and receive help on that command.

If there are particular translations or actions taking place in the **PreTranslateMessage** function that shouldn't take place during SHIFT+F1 Help mode, you should check the **m_bHelpMode** member of **CWinApp** before performing those

operations. The **CDialog** implementation of **PreTranslateMessage** checks this before calling **IsDialogMessage**, for example. This disables "dialog navigation" keys on modeless dialogs during SHIFT+F1 mode. In addition, **CWinApp::OnIdle** is still called during this loop.

If the user chooses a command from the menu, it is handled as help on that command (through **WM_COMMANDHELP**, see below). If the user clicks a visible area of the applications window, a determination is made as to whether it is a nonclient click or a client click. **OnContextHelp** handles mapping of nonclient clicks to client clicks automatically. If it is a client click, it then sends a **WM_HELPHITTEST** to the window that was clicked. If that window returns a nonzero value, that value is used as the context for help. If it returns zero, **OnContextHelp** tries the parent window (and failing that, its parent, and so on). If a Help context cannot be determined, the default is to send a **ID_DEFAULT_HELP** command to the main window, which is then (usually) mapped to **CWinApp::OnHelpIndex**.

WM_HELPHITTEST

```
afx_msg LRESULT CWnd::OnHelpHitTest(
WPARAM, LPARAM lParam)
```

WM_HELPHITTEST is an MFC private windows message that is received by the active window clicked during SHIFT+F1 Help mode. When the Window receives this message, it returns a DWORD Help ID for use by WinHelp.

LOWORD(lParam)

contains the X-axis device coordinate where the mouse was clicked relative to the client area of the window.

HIWORD(lParam)

contains the Y-axis coordinate.

wParam

is not used and will be zero. If the return value is nonzero, WinHelp is called with that context. If the return value is zero, the parent window is queried for help.

In many cases, you can leverage hit-testing code you may already have. See the implementation of **CToolBar::OnHelpHitTest** for an example of handling the **WM_HELPHITTEST** message (the code leverages the hit-test code used on buttons and tooltips in **CControlBar**).

MFC Application Wizard Support and MAKEHM

The MFC Application Wizard creates the necessary files to build a Help file (.cnt and .hpj files). It also includes a number of prebuilt .rtf files that are accepted by the Microsoft Help Compiler. Many of the topics are complete, but some may need to be modified for your specific application.

Automatic creation of a "help mapping" file is supported by a utility called MAKEHM. The MAKEHM utility can translate an application's RESOURCE.H file to a Help mapping file. For example:

```
#define IDD_MY_DIALOG   2000
#define ID_MY_COMMAND   150
```

will be translated into:

```
HIDD_MY_DIALOG    0x207d0
HID_MY_COMMAND    0x10096
```

This format is compatible with the Help compiler's facility, which maps context IDs (the numbers on the right side) with topic names (the symbols on the left side).

The source code for MAKEHM is available in the MFC Programming Utilities sample MAKEHM.

Adding Help Support After Running the MFC Application Wizard

The best way to add Help to your application is to check the "Context-sensitive Help" option on the Advanced Features page of the MFC Application Wizard before creating your application. That way the MFC Application Wizard automatically adds the necessary message map entries to your **CWinApp**-derived class to support Help.

Help on Message Boxes

Help on Message Boxes (sometimes called alerts) is supported through the **AfxMessageBox** function, a wrapper for the **MessageBox** Windows API.

There are two versions of **AfxMessageBox**, one for use with a string ID and another for use with a pointer to string (**LPCSTR**):

```
int AFXAPI AfxMessageBox(LPCSTR lpszText, UINT nType, UINT nIDHelp);
int AFXAPI AfxMessageBox(UINT nIDPrompt, UINT nType, UINT nIDHelp);
```

In both cases, there is an optional Help ID.

In the first case, the default for nIDHelp is 0, which indicates no Help for this message box. If the user presses F1 while such as message box is active, the user will not receive Help (even if the application supports Help). If this is not desirable, a Help ID should be provided for nIDHelp.

In the second case, the default value for nIDHelp is -1, which indicates the Help ID is the same as nIDPrompt. Help will work only if the application is Help-enabled, of course). You should provide 0 for nIDHelp if you wish that the message box have no help support. Should you want the message to be Help enabled, but desire a different help ID than nIDPrompt, simply provide a positive value for nIDHelp different from that of nIDPrompt.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN029: Splitter Windows

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes the MFC **CSplitterWnd** class, which is used to provide window splits and to manage the resizing of other Pane windows.

Note
Be sure to read Technical Note 20 regarding use of IDs and valid ID ranges.

Splitter Styles

A **CSplitterWnd** supports two different styles of splitting windows.

In "static splitters," the panes are created when the splitter window is created, and the order and number of panes never change. Splitter bars are used to resize the different panes, and the different panes are usually of different view classes. The Visual C++ graphics editor and the Windows File Manager are examples of programs that use this splitter style. Splitter boxes are not used by this style of splitter.

In "Dynamic splitters," additional panes are created and destroyed as the user splits and un-splits new views. This splitter starts out with a single view, and splitter boxes are provided to initiate splitting. If the view is split in one direction, an additional view object is dynamically created to represent the new pane. If the view is split in two directions (possible with the keyboard interface), then three new views are created to represent the three new panes. When the split is active, the splitter box is drawn as a splitter bar between the panes. Additional view objects are destroyed when the user removes a split, but the original view (row 0, column 0) remains until the splitter window itself is destroyed. Microsoft Excel and Microsoft Word are examples of the dynamic splitter style.

When creating either kind of splitter window, you must specify the maximum number of rows and columns that the splitter will manage. For a static splitter, panes must be created to fill all the rows and columns. For a dynamic splitter, the first pane is automatically created when the **CSplitterWnd** is created.

The maximum number of panes you can specify for static splitters is 16 rows by 16 columns. The recommended configurations are:

- 1 row x 2 columns : usually with dissimilar panes
- 2 rows x 1 column : usually with dissimilar panes
- 2 rows x 2 columns : usually with similar panes

The maximum number of panes you can specify for dynamic splitters is 2 rows by 2 columns. The recommended configurations are:

- 1 row x 2 columns : for columnar data
- 2 rows x 1 column : for textual or other data
- 2 rows x 2 columns : for grid or table oriented data

Splitter Examples

Many of the MFC sample programs use splitter windows directly or indirectly. The MFC General sample VIEWEX illustrates several uses of static splitters, including how to place a splitter in a splitter.

ClassWizard will also create a new multiple document interface (MDI) Child frame window class which contains a splitter window. For more information on splitter windows, see Multiple Document Types, Views, and Frame Windows.

Terminology Used by Implementation

Terminology of the parts of a **CSplitterWnd** and related objects.

CSplitterWnd:

This is a window that provides pane-splitting controls and scroll bars that are shared between all panes on a row or column. Rows and columns are specified with zero-based numbers [that is, the first pane is row = 0 and column = 0]

Pane:

An application-specific window that is managed by a **CSplitterWnd**. A pane is usually a **CView**-derived object, but in fact can be any **CWnd** object that has the appropriate child window ID.

To do so, simply pass the **RUNTIME_CLASS** of your **CWnd** derived class as you would if you were using a **CView** derived class. Your class must use **DECLARE_DYNCREATE** and **IMPLEMENT_DYNCREATE** — the framework uses dynamic creation at runtime. Although there is a lot of code that is **CView** specific in **CSplitterWnd**, **CObject::IsKindOf** is always used before those actions are performed. Certainly, it is much easier to use **CSplitterWnd** with **CView** derived classes than **CWnd** derived classes.

Splitter Bar:

A control that is placed between rows and columns of panes. It may be used to adjust the sizes of rows or columns of Panes.

Splitter Box:

A small control at the top of the vertical scroll bars or to the left of the horizontal scroll bars in a dynamic **CSplitterWnd**. Used to create new rows or columns of panes.

Splitter Intersection:

The intersection of a vertical splitter bar and a horizontal splitter bar. May be dragged to adjust the size of a row and column of panes simultaneously.

Shared Scroll Bars

The **CSplitterWnd** class also supports shared scroll bars. These scroll bar controls are children of the **CSplitterWnd** and are shared with the different panes in the splitter.

For example, in a 1 row x 2 column window, you can specify **WS_VSCROLL** when creating the **CSplitterWnd**. A special scroll bar control will be created that is shared between the two panes.

```
[      ][      ][^]
[pane00][pane01][|]
[      ][      ][v]
```

When the user moves the scroll bar, **WM_VSCROLL** messages will be sent to both views. When the views set the scroll bar position, the shared scroll bar will be set.

Note that shared scroll bars are most useful with dynamic or static splits, splitting similar view objects. If you mix views of different types in a splitter, then you may have to write special code to coordinate their scroll positions. Any **CView**-derived class that uses the **CWnd** scroll bar APIs will delegate to the shared scroll bar if it exists. The **CScrollView** implementation is one such example of a **CView** class that supports shared scroll bars. Non-**CView** derived classes, classes that rely on noncontrol scroll bars, or classes that use standard Windows implementations (for example, **CEditView**) will not work with the shared scroll bar feature of **CSplitterWnd**.

Minimum Sizes

For each row there is a minimum row height, and similarly for each column there is a minimum column width. This minimum is used to decide if the pane is to small to be shown in complete detail.

For a static splitter window, the initial minimum row height and column width is 0. For a dynamic splitter window, the initial minimum row height and column width are set by the *sizeMin* parameter to the **CSplitterWnd::Create** function.

These minimum sizes can be changed with the **SetRowInfo** and **SetColumnInfo** APIs.

Actual vs. Ideal Sizes

The layout of the panes in the splitter window depends on the size of the containing frame (which in turn resizes the **CSplitterWnd**. **CSplitterWnd** repositions and resizes the panes so that they fit as ideally as possible).

The row height and column width sizes set by the user, or through the **CSplitterWnd** API, represent the ideal size. The actual size can be smaller than that ideal size (if there is not enough room to make that pane the ideal size) or larger than the ideal

size (if that pane must be made larger to fill the left-over space on the right or bottom of the splitter window).

Protected Interface

The following describes some of the splitter window implementation overridables that can be used by advanced users of **CSplitterWnd** to customize the features and user interface of this class. These APIs are officially undocumented and are subject to change in future versions of MFC. Refer to the implementation source code for more details on these implementation APIs.

Drawing the splitter bars, boxes and trackers:

```
enum ESplitType
    { splitBox, splitBar, splitIntersection, splitBorder };
virtual void OnDrawSplitter(CDC* pDC, ESplitType nType, const CRect& rect);
virtual void OnInvertTracker(const CRect& rect);
```

These virtual function can be overridden to provide alternate imagery for the various graphical components of a splitter window. The default imagery is similar to the splitter in Microsoft Works for Windows: only intersections of splitter bars are blended together. The imagery is also quite different when the framework detects Windows 4.0 — in order to match the visuals in the shell on that (future) operating system.

Creating controls and views:

```
virtual BOOL CreateScrollBarCtrl(DWORD dwStyle, UINT nID);
```

This is called to create a shared scroll bar control. It can be overridden to include extra controls next to a scroll bar. The default behavior is to just create normal Windows scroll bar controls.

```
virtual void DeleteView(int row, int col);
virtual BOOL SplitRow(int cyBefore);
virtual BOOL SplitColumn(int cxBefore);
virtual void DeleteRow(int row);
virtual void DeleteColumn(int row);
```

These functions are called to implement the logic of the dynamic splitter window (that is, if the splitter window has the **SPLS_DYNAMIC_SPLIT** style). They can be customized, along with the virtual function **CreateView**, to implement more advanced dynamic splitters.

The following are high level commands that are used by the **CView** class to delegate to the **CSplitterWnd** implementation. They are virtual so that the standard **CView** implementation will not require the entire **CSplitterWnd** implementation to be linked in. For applications that use **CView** but not **CSplitterWnd**, the **CSplitterWnd** implementation will not be linked with the application.

virtual BOOL CanActivateNext(BOOL bPrev = FALSE);

Checks to see if the "Next Pane" or "Previous Pane" command is currently possible.

virtual void ActivateNext(BOOL bPrev = FALSE);

Performs the "Next Pane" or "Previous Pane" command.

virtual BOOL DoKeyboardSplit();

Performs the keyboard split command, usually "Window Split".

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN030: Customizing Printing and Print Preview

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes the process of customizing printing and print preview and describes the purposes of the callback routines used in **CView** and the callback routines and member functions of **CPreviewView**.

The Problem

MFC provides a complete solution for most printing and print preview needs. In most cases, little additional code is required to have a view able to print and preview. However, there are ways to optimize printing that require significant effort on the part of the developer, and some applications need to add specific user interface elements to the print preview mode.

Efficient Printing

When an MFC application prints using the standard methods, Windows directs all Graphical Device Interface (GDI) output calls to an in-memory metafile. When **EndPage** is called, Windows plays the metafile once for each physical band that the printer requires to print one page. During this rendering, GDI frequently queries the Abort Procedure to determine if it should continue. Typically the abort procedure allows messages to be processed so that the user may abort the print job using a printing dialog.

Unfortunately, this can slow the printing process. If the printing in your application must be faster than can be achieved using the standard technique, you must implement manual banding.

Print Banding

In order to manually band, you must re implement the print loop such that **OnPrint** is called multiple times per page (once per band). The print loop is implemented in the **OnFilePrint** function in viewprnt.cpp. In your **CView**-derived class, you overload this function so that the message map entry for handling the print command calls your print function. Copy the **OnFilePrint** routine and change the print loop to implement banding. You will probably also want to pass the banding rectangle to your printing functions so that you can optimize drawing based on the section of the page being printed.

Second, you must frequently call **QueryAbort** while drawing the band. Otherwise, the Abort Procedure will not get called and the user will be unable to cancel the print job.

Print Preview: Electronic Paper with User Interface

Print Preview, in essence, tries to turn the display into an emulation of a printer. By default, the client area of the main window is used to display one or two pages fully within the window. The user is able to zoom in on an area of the page to see it in more detail. With additional support, the user may even be allowed to edit the document in preview mode.

Customizing Print Preview

This note only deals with one aspect of modifying print preview: Adding UI to preview mode. Other modifications are possible, but such changes are out of the scope of this discussion.

To add UI to the preview mode

1. Derive a view class from **CPreviewView**.
2. Add command handlers for the UI aspects you desire.
3. If you are adding visual aspects to the display, override **OnDraw** and perform your drawing after calling **CPreviewView::OnDraw.**

OnFilePrintPreview

This is the command handler for print preview. Its default implementation is:

```
void CView::OnFilePrintPreview()
{
```

```
    // In derived classes, implement special window handling here
    // Be sure to Unhook Frame Window close if hooked.

    // must not create this on the frame. Must outlive this function
    CPrintPreviewState* pState = new CPrintPreviewState;

    if (!DoPrintPreview(AFX_IDD_PREVIEW_TOOLBAR, this,
                RUNTIME_CLASS(CPreviewView), pState))
    {
        // In derived classes, reverse special window handling
        // here for Preview failure case

        TRACE0("Error: DoPrintPreview failed");
        AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
        delete pState;      // preview failed to initialize,
                    // delete State now
    }
}
```

DoPrintPreview will hide the main pane of the application. Control Bars, such as the status bar, can be retained by specifying them in the pState->**dwStates** member (This is a bit mask and the bits for individual control bars are defined by **AFX_CONTROLBAR_MASK**(AFX_IDW_MYBAR)). The window pState->**nIDMainPane** is the window that will be automatically hidden and reshown. **DoPrintPreview** will then create a button bar for the standard Preview UI. If special window handling is needed, such as to hide or show other windows, that should be done before **DoPrintPreview** is called.

By default, when print preview finishes, it returns the control bars to their original states and the main pane to visible. If special handling is needed, it should be done in an override of **EndPrintPreview.** If **DoPrintPreview** fails, also provide special handling.

DoPrintPreview is called with:

- The Resource ID of the dialog template for the preview toolbar.
- A pointer to the view to perform the printing for the print preview.
- The run-time class of the Preview View class. This will be dynamically created in DoPrintPreview.
- The CPrintPreviewState pointer. Note that the CPrintPreviewState structure (or the derived structure if the application needs more state preserved) must *not* be created on the frame. DoPrintPreview is modeless and this structure must survive until EndPrintPreview is called.

Note
If a separate view or view class is needed for printing support, a pointer to that object should be passed as the second parameter.

EndPrintPreview

This is called to terminate the print preview mode. It is often desirable to move to the page in the document that was last displayed in print preview. **EndPrintPreview** is the application's chance to do that. The pInfo->**m_nCurPage** member is the page that was last displayed (leftmost if two pages were displayed), and the pointer is a hint as to where on the page the user was interested. Since the structure of the application's view is unknown to the framework, then you must provide the code to move to the chosen point.

You should perform most actions before calling **CView::EndPrintPreview**. This call reverses the effects of **DoPrintPreview** and deletes pView, pDC, and pInfo.

```
// Any further cleanup should be done here.
CView::EndPrintPreview(pDC, pInfo, point, pView);
```

CWinApp::OnFilePrintSetup

This must be mapped for the Print Setup menu item. In most cases, it is not necessary to override the implementation.

Page Nomenclature

Another issue is that of page numbering and order. For simple word processor type applications, this is a straightforward issue.

Most print preview systems assume that each printed page corresponds to one page in the document.

In trying to provide a generalized solution, there are several things to consider. Imagine a CAD system. The user has a drawing that covers several E-size sheets. On an E-size (or a smaller, scaled) plotter, page numbering would be as in the simple case. But on a laser printer, printing 16 A-size pages per sheet, what does print preview consider a "page"?

As the introductory paragraph states, Print Preview is acting like a printer. Therefore, the user will see what would come out of the particular printer that is selected. It is up to the view to determine what image is printed on each page.

The page description string in the **CPrintInfo** structure provides a means of displaying the page number to the user if it can be represented as one number per page (as in "Page 1" or "Pages 1-2"). This string is used by the default implementation of **CPreviewView::OnDisplayPageNumber**. If a different display is needed, one may override this virtual function to provide, for example, "Sheet1, Sections A, B".

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN031: Control Bars

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes the control bar classes in MFC: the general CControlBar, CStatusBar, CToolBar, CDialogBar, and **CDockBar**.

CControlBar

A **ControlBar** is a **CWnd**-derived class that:

- Is aligned to the top or bottom of a frame window.
- May contain child items that are either HWND-based controls (for example, **CDialogBar**) or non-**HWND** based items (for example, **CToolBar**, **CStatusBar**).

Control bars support the additional styles:

- **CBRS_TOP** (The default) pin the control bar to the top.
- **CBRS_BOTTOM** Pin the control bar to the bottom.
- **CBRS_NOALIGN** Do not reposition the control bar when the parent resizes.

Classes derived from **CControlBar** provide more interesting implementations:

- **CStatusBar** A status bar, items are status bar panes containing text.
- **CToolBar** A toolbar, items are bitmap buttons aligned in a row.
- **CDialogBar** A toolbar-like frame containing standard windows controls (created from a dialog template resource).
- **CDockBar** A generalized docking area for other **CControlBar** derived objects. The specific member functions and variables available in this class are likely to change in future releases.

All control bar objects/windows will be child windows of some parent frame window. They are usually added as a sibling to the client area of the frame (for example, an MDI Client or view). The child window ID of a control bar is important. The default layout of control bar only works for control bars with IDs in the range of **AFX_IDW_CONTROLBAR_FIRST** to **AFX_IDW_CONTROLBAR_LAST**. Note that even though there is a range of 256 control bar IDs, the first 32 of these control bar IDs are special since they are directly supported by the print preview architecture.

The **CControlBar** class gives standard implementation for:

- Aligning the control bar to the top, bottom, or either side of the frame.
- Allocating control item arrays.
- Supporting the implementation of derived classes.

C++ control bar objects will usually be embedded as members of a **CFrameWnd** derived class, and will be cleaned up when the parent **HWND** and object are destroyed. If you need to allocate a control bar object on the heap, you can simply set the **m_bAutoDestruct** member to **TRUE** to make the control bar "**delete this**" when the **HWND** is destroyed.

Note
If you create your own **CControlBar**-derived class, rather than using one of MFC's derived classes, such as **CStatusBar**, **CToolBar**, or **CDialogBar**, you will need to set the **m_dwStyle** data member. This can be done in the override of **Create**:

```
// CMyControlBar is derived from CControlBar
BOOL CMyControlBar::Create( CWnd* pParentWnd, DWORD dwStyle, UINT nID )
{
```

```
    m_dwStyle = dwStyle;

    .
    .
    .
}
```

Control Bar Layout Algorithm

The control bar layout algorithm is very simple. The frame window sends a message **WM_SIZEPARENT** to all children in the control bar range. Along with this message, a pointer to the parent's client rectangle is passed. This message is sent to children in Z-order. The control-bar children use this information to position themselves and to decrease the size of the parent's client area. The final rectangle that is left for the normal client area (less control bars) is used to position the main client window (usually an MDI client, view or splitter window).

See **CWnd::RepositionBars** and **CFrameWnd::RecalcLayout** for more details.

MFC private Windows messages, including **WM_SIZEPARENT**, are documented in Technical Note 24.

CStatusBar

A status bar is a control bar that has a row of text output panes. There are two common ways to use text output panes:

- As a message line

  (for example, the standard menu help message line). These are usually accessed by a 0-based indexed

- As status indicators

  (for example, the CAP, NUM and SCRL indicators). These are usually accessed by string/command ID.

The font for the status bar is 10-point MS Sans Serif (dictated by the Windows Interface Application Design Guide or the font mappers best match of a 10-point Swiss proportional font). On certain versions of Windows, such as the Japanese edition and Windows NT 4.0, the fonts selected are different.

The colors used in the status bar are also consistent with the recommendation of the Windows Interface Application Design Guide. These colors are not hard coded and are changed dynamically in response to user customization in Control Panel.

Item	Windows COLOR value	Default RGB
Status bar background	**COLOR_BTNFACE**	RGB(192, 192, 192)
Status bar text	**COLOR_BTNTEXT**	RGB(000, 000, 000)
Status bar top/left edges	**COLOR_BTNHIGHLIGHT**	RGB(255, 255, 255)
Status bar bot/right edges	**COLOR_BTNSHADOW**	RGB(128, 128, 128)

CCmdUI Support for CStatusBar

The way indicators are usually updated is through the **ON_UPDATE_COMMAND_UI** mechanism. On idle time, the status bar will call the **ON_UPDATE_COMMAND_UI** handler with the string ID of the indicator pane.

The **ON_UPDATE_COMMAND_UI** handler can call:

- **Enable**: To enable or disable the pane. A disabled pane looks exactly like an enabled pane but the text is invisible (that is, turns off the text indicator).
- **SetText**: To change the text. Be careful if you use this because the pane will not automatically resize.

Refer to class CStatusBar in the *Class Library Reference* for details about **CStatusBar** creation and customization APIs. Most customization of status bars should be done before the status bar is initially made visible.

The status bar supports only one stretchy pane, usually the first pane. The size of that pane is really a minimum size. If the status bar is bigger than the minimum size of all the panes, any extra width will be given to the stretchy pane. The default application with a status bar has right-aligned indicators for CAP, NUM and SCRL since the first pane is stretchy.

CToolBar

A toolbar is a control bar with a row of bitmap buttons that may include separators. Two styles of buttons are supported: pushbuttons and check box buttons. Radio group functionality can be built with check box buttons and **ON_UPDATE_COMMAND_UI**.

All the bitmap buttons in the toolbar are taken from one bitmap. This bitmap must contain one image or glyph for each button. Typically the order of the images/glyphs in the bitmap is the same order they will be drawn on the screen. (This can be changed using the customization APIs.)

Each button must be the same size. The default is the standard 24x22 pixels. Each image/glyph must be the same size and must be side-by-side in the bitmap. The default image/glyph size is 16x15 pixels. Therefore, for a toolbar with 10 buttons (using standard sizes), you need a bitmap that is 160 pixels wide and 15 pixels high.

Each button has one and only one image/glyph. The different button states and styles (for example, pressed, up, down, disabled, disabled down, indeterminate) are algorithmically generated from that one image/glyph. Any color bitmap or DIB can be used in theory. The algorithm for generating the different button states works best if the original image is shades of gray. Look at the standard toolbar buttons and the toolbar button clipart provided in MFC General sample CLIPART for examples.

The colors used in the toolbar are also consistent with the recommendation of the Windows Interface Application Design Guide. These colors are not hard coded and are changed dynamically in response to user customization in Control Panel.

Item	**Windows COLOR value**	**Default RGB**
ToolBar background	**COLOR_BTNFACE**	RGB(192,192,192)
ToolBar buttons top/left edges	**COLOR_BTNHIGHLIGHT**	RGB(255,255,255)
ToolBar buttons bot/right edges	**COLOR_BTNSHADOW**	RGB(128,128,128)

In addition, the toolbar bitmap buttons are recolored as though they were standard Windows button controls. This recoloring occurs when the bitmap is loaded from the resource and in response to a change in system colors in response to user customization in Control Panel. The following colors in a toolbar bitmap will be recolored automatically so they should be used with caution. If you do not wish to have a portion of your bitmap recolored, then use a color that closely approximates one of the mapped RGB values. The mapping is done based on exact RGB values.

RGB value	**Dynamically mapped COLOR value**
RGB(000, 000, 000)	COLOR_BTNTEXT
RGB(128, 128, 128)	COLOR_BTNSHADOW
RGB(192, 192, 192)	COLOR_BTNFACE
RGB(255, 255, 255)	COLOR_BTNHIGHLIGHT

Refer to class CToolBar the *Class Library Reference* for details about the **CToolBar** creation and customization APIs. Most customization of toolbars should be done before the toolbar is initially made visible.

The customization APIs can be used to adjust the button IDs, styles, spacer width and which image/glyph is used for what button. By default you do not need to use these APIs.

CCmdUI Support for CToolBar

The way toolbar buttons are always updated is through the **ON_UPDATE_COMMAND_UI** mechanism. On idle time, the toolbar will call the **ON_UPDATE_COMMAND_UI** handler with the command ID of that button. **ON_UPDATE_COMMAND_UI** is not called for separators, but it is called for pushbuttons and check box buttons.

The **ON_UPDATE_COMMAND_UI** handler can call:

- **Enable**: To enable or disable the button. This works equally for pushbuttons and check box buttons.
- **SetCheck**: To set the check state of a button. Calling this for a toolbar button will turn it into a check box button. **SetCheck** takes a parameter which can be 0 (not checked), 1 (checked) or 2 (indeterminate)
- **SetRadio**: Shorthand for **SetCheck**.

Check box buttons are "AUTO" check box buttons; that is, when the user presses them they will immediately change state.

Checked is the down or depressed state. There is no built-in user interface way to change a button into the "indeterminate" state; that must be done through code.

The customization APIs will permit you to change the state of a given toolbar button, preferably you should change these states in the **ON_UPDATE_COMMAND_UI** handler for the command the toolbar button represents. Remember, the idle processing will change the state of toolbar buttons with the **ON_UPDATE_COMMAND_UI** handler, so any changes to these states made through SetButtonStyle may get lost after the next idle.

Toolbar buttons will send **WM_COMMAND** messages like normal buttons or menu items and are normally handled by an **ON_COMMAND** handler in the same class that provides the **ON_UPDATE_COMMAND_UI** handler.

There are four Toolbar button styles (TBBS_ values) used for display states:

- TBBS_CHECKED: Check box is currently checked (down).
- TBBS_INDETERMINATE: Check box is currently indeterminate.
- TBBS_DISABLED: Button is currently disabled.
- TBBS_PRESSED: Button is currently pressed.

The six official Windows Interface Application Design Guide button styles are represented by the following TBBS values:

- Up = 0
- Mouse Down = TBBS_PRESSED (| any other style)
- Disabled = TBBS_DISABLED
- Down = TBBS_CHECKED
- Down Disabled = TBBS_CHECKED | TBBS_DISABLED
- Indeterminate = TBBS_INDETERMINATE

CDialogBar

A dialog bar is a control bar that contains standard Windows controls. It acts like a dialog in that it contains the controls and supports tabbing between them. It also acts like a dialog in that it uses a dialog template to represent the bar.

A **CDialogBar** is used for the print-preview toolbar, which contains standard pushbutton controls.

Using a **CDialogBar** is like using a **CFormView**. You must define a dialog template for the dialog bar and remove all the styles except **WS_CHILD**. Note that the dialog must not be visible.

The control notifications for a **CDialogBar** will be sent to the parent of the control bar (just like toolbar buttons).

CCmdUI Support for CDialogBar

Dialog bar buttons should be updated through the **ON_UPDATE_COMMAND_UI** handler mechanism. At idle time, the dialog bar will call the **ON_UPDATE_COMMAND_UI** handler with the command ID of all the buttons that have a ID >= 0x8000 (that is, in the range of command IDs).

The **ON_UPDATE_COMMAND_UI** handler can call:

- Enable: to enable or disable the button.
- SetText: to change the text of the button.

Customization can be done through standard window manager APIs.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN032: MFC Exception Mechanism

Previous versions of Visual C++ did not support the standard C++ exception mechanism, and MFC provided macros **TRY/CATCH/THROW** that were used instead. This version of Visual C++ fully supports C++ exceptions. This note covered some of the advanced implementation details of the previous macros including how to automatically cleanup stack based objects. Because C++ exceptions support stack unwinding by default, this technical note is no longer necessary.

Refer to Exceptions: Using MFC Macros and C++ Exceptions for more information on the differences between the MFC macros and the new C++ keywords.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN033: DLL Version of MFC

This note describes how you can use the MFCxx.DLL and MFCxxD.DLL (where x is the MFC version number) shared dynamic link libraries with MFC applications and extension DLLs. For more information about regular DLLs, see Using MFC as Part of a DLL.

This technical note covers three aspects of DLLs. The last two are for the more advanced users:

- How you build an MFC Extension DLL
- How you build an MFC application that uses the DLL version of MFC
- How the MFC shared dynamic-link libraries are implemented

If you are interested in building a DLL using MFC that can be used with non-MFC applications (this is called a regular DLL), refer to Technical Note 11.

Overview of MFCxx.DLL Support: Terminology and Files

Regular DLL: You use a regular DLL to build a stand-alone DLL using some of the MFC classes. Interfaces across the App/DLL boundary are "C" interfaces, and the client application does not have to be an MFC application.

This is the version of DLL support supported in MFC 1.0. It is described in Technical Note 11 and the MFC Advanced Concepts sample DLLScreenCap.

Note
As of Visual C++ version 4.0, the term **USRDLL** is obsolete and has been replaced by a regular DLL that statically links to MFC. You may also build a regular DLL that dynamically links to MFC.

MFC 3.0 (and above) supports regular DLLs with all the new functionality including the OLE and Database classes.

AFXDLL: This is also referred to as the shared version of the MFC libraries. This is the new DLL support added in MFC 2.0. The MFC library itself is in a number of DLLs (described below) and a client application or DLL dynamically links the DLLs that it requires. Interfaces across the application/DLL boundary are C++/MFC class interfaces. The client application MUST be an MFC application. This supports all MFC 3.0 functionality (exception: UNICODE is not supported for the database classes).

Note
As of Visual C++ version 4.0, this type of DLL is referred to as an "Extension DLL."

This note will use MFCxx.DLL to refer to the entire MFC DLL set, which includes:

- Debug: MFCxxD.DLL (combined) and MFCSxxD.LIB (static).
- Release: MFCxx.DLL (combined) and MFCSxx.LIB (static).
- Unicode Debug: MFCxxUD.DLL (combined) and MFCSxxD.LIB (static).
- Unicode Release: MFCxxU.DLL (combined) and MFCSxxU.LIB (static).

Note
The MFCSxx[U][D].LIB libraries are used in conjunction with the MFC shared DLLs. These libraries contain code that must be statically linked to the application or DLL.

An application links to the corresponding import libraries:

- Debug: MFCxxD.LIB
- Release: MFCxx.LIB
- Unicode Debug: MFCxxUD.LIB
- Unicode Release: MFCxxU.LIB

An "MFC Extension DLL" is a DLL built upon MFCxx.DLL (and/or the other MFC shared DLLs). Here the MFC component architecture kicks in. If you derive a useful class from an MFC class, or build another MFC-like toolkit, you can place it in a DLL. That DLL uses MFCxx.DLL, as does the ultimate client application. This permits reusable leaf classes, reusable base classes, and reusable view/document classes.

Pros and Cons

Why should you use the shared version of MFC?

- Using the shared library can result in smaller applications (a minimal application that uses most of the MFC library is less than 10K).
- The shared version of MFC supports MFC Extension DLLs and regular DLLs.
- Building an application that uses the shared MFC libraries is faster than building a statically linked MFC application because it is not necessary to link MFC itself. This is especially true in **DEBUG** builds where the linker must compact the debug information — by linking with a DLL that already contains the debug information, there is less debug information to compact within your application.

Why should you not use the shared version of MFC:

- Shipping an application that uses the shared library requires that you ship the MFCxx.DLL (and others) library with your program. MFCxx.DLL is freely redistributable like many DLLs, but you still must install the DLL in your SETUP program. In addition, you must ship the MSVCRTxx.DLL, which contains the C-runtime library which is used both by your program and the MFC DLLs themselves.

How to Write an MFC Extension DLL

An MFC Extension DLL is a DLL containing classes and functions written to embellish the functionality of the MFC classes. An MFC Extension DLL uses the shared MFC DLLs in the same way an application uses it, with a few additional considerations:

- The build process is similar to building an application that uses the shared MFC libraries with a few additional compiler and linker options.
- An MFC Extension DLL does not have a **CWinApp**-derived class.
- An MFC Extension DLL must provide a special **DllMain**. AppWizard supplies a **DllMain** function that you can modify.
- An MFC Extension DLL will usually provide an initialization routine to create a **CDynLinkLibrary** if the extension DLL wishes to export **CRuntimeClass**es or resources to the application. A derived class of **CDynLinkLibrary** may be used if per-application data must be maintained by the extension DLL.

These considerations are described in more detail below. You should also refer to the MFC Advanced Concepts sample DLLHUSK since it illustrates:

- Building an application using the shared libraries. (DLLHUSK.EXE is an MFC application that dynamically links to the MFC libraries as well as other DLLs.)
- Building an MFC Extension DLL. (Note the special flags such as **_AFXEXT** that are used in building an extension DLL)
- Two examples of MFC Extension DLLs. One shows the basic structure of an MFC Extension DLL with limited exports (TESTDLL1) and the other shows exporting an entire class interface (TESTDLL2).

Both the client application and any extension DLLs must use the same version of MFCxx.DLL. You should follow the convention of MFC DLL and provide both a debug and retail (/release) version of your extension DLL. This permits client programs to build both debug and retail versions of their applications and link them with the appropriate debug or retail version of all DLLs.

Note
Because C++ name mangling and export issues, the export list from an extension DLL may be different between the debug and retail versions of the same DLL and DLLs for different platforms. The retail MFCxx.DLL has about 2000 exported entry points; the debug MFCxxD.DLL has about 3000 exported entry points.

Quick Note on Memory Management

The section titled "Memory Management," near the end of this technical note, describes the implementation of the MFCxx.DLL with the shared version of MFC. The information you need to know to implement just an extension DLL is described here.

MFCxx.DLL and all extension DLLs loaded into a client application's address space will use the same memory allocator, resource loading and other MFC "global" states as if they were in the same application. This is significant because the non-MFC DLL libraries and regular DLLs that statically link to MFC do the exact opposite and have each DLL allocating out of its own memory pool.

If an extension DLL allocates memory, then that memory can freely intermix with any other application-allocated object. Also, if an application that uses the shared MFC libraries crashes, the protection of the operating system will maintain the integrity of any other MFC application sharing the DLL.

Similarly other "global" MFC states, like the current executable file to load resources from, are also shared between the client application and all MFC extension DLLs as well as MFCxx.DLL itself.

Building an Extension DLL

You can use AppWizard to create an MFC extension DLL project, and it will automatically generate the appropriate compiler and linker settings. It was also generate a **DllMain** function that you can modify.

If you are converting an existing project to an MFC extension DLL, start with the standard rules for building an application using the shared version of MFC, then do the following:

- Add **/D_AFXEXT** to the compiler flags. On the Project Properties dialog, select the C/C++ node. Then select the Preprocessor category. Add **_AFXEXT** to the Define Macros field, separating each of the items with semicolons.
- Remove the **/Gy** compiler switch. On the Project Properties dialog, select the C/C++ node. Then select the Code Generation category. Ensure that the "Enable Function-Level Linking" option is not enabled. This will make it easier to export classes because the linker will not remove unreferenced functions. If the original project is used to build a Regular DLL statically linked to MFC, change the **/MT[d]** compiler option to **/MD[d]**.
- Build an export library with the **/DLL** option to LINK. This will be set when you create a new target, specifying Win32 Dynamic-Link Library as the target type.

Changing your Header Files

The goal of an extension DLL is usually to export some common functionality to one or more applications that can use that functionality. This boils down to exporting classes and global functions that are available for your client applications.

In order to do this you must insure that each of the member functions is marked as import or export as appropriate. This requires special declarations: **__declspec(dllexport)** and **__declspec(dllimport)**. When your classes are used by the client applications, you want them to be declared as **__declspec(dllimport)**. When the extension DLL itself is being built, they should be declared as **__declspec(dllexport)**. In addition, the functions must be actually exported, so that the client programs bind to them at load time.

To export your entire class, use **AFX_EXT_CLASS** in the class definition. This macro is defined by the framework as **__declspec(dllexport)** when **_AFXDLL** and **_AFXEXT** is defined, but defined as **__declspec(dllimport)** when **_AFXEXT** is not defined. **_AFXEXT** as described above, is only defined when building your extension DLL. For example:

```
class AFX_EXT_CLASS CExampleExport : public CObject
{ ... class definition ... };
```

Not Exporting the Entire Class

Sometimes you may want to export just the individual necessary members of your class. For example, if you are exporting a **CDialog**-derived class, you might only need to export the constructor and the **DoModal** call. You can export these members using the DLL's .DEF file, but you can also use **AFX_EXT_CLASS** in much the same way on the individual members you need to export.

For example:

```
class CExampleDialog : public CDialog
{
public:
   AFX_EXT_CLASS CExampleDialog();
   AFX_EXT_CLASS int DoModal();
   // rest of class definition
   .
   .
   .
};
```

When you do this, you may run into an additional problem because you are no longer exporting all members of the class. The problem is in the way that MFC macros work. Several of MFC's helper macros actually declare or define data members. Therefore, these data members will also need to be exported from your DLL.

For example, the **DECLARE_DYNAMIC** macro is defined as follows when building an extension DLL:

```
#define DECLARE_DYNAMIC(class_name) \
protected: \
   static CRuntimeClass* PASCAL _GetBaseClass(); \
   public: \
   static AFX_DATA CRuntimeClass class##class_name; \
   virtual CRuntimeClass* GetRuntimeClass() const; \
```

The line that begins "static **AFX_DATA**" is declaring a static object inside of your class. To export this class correctly and access the runtime information from a client .EXE, you need to export this static object. Because the static object is declared with the modifier **AFX_DATA**, you only need to define **AFX_DATA** to be **__declspec(dllexport)** when building your DLL and define it as **__declspec(dllimport)** when building your client executable.

As discussed above, **AFX_EXT_CLASS** is already defined in this way. You just need to re-define **AFX_DATA** to be the same as **AFX_EXT_CLASS** around your class definition.

For example:

```
   #undef  AFX_DATA
   #define AFX_DATA AFX_EXT_CLASS
   class CExampleView : public CView
   {
     DECLARE_DYNAMIC()
     // ... class definition ...
   };
   #undef  AFX_DATA
   #define AFX_DATA
```

MFC always uses the **AFX_DATA** symbol on data items it defines within its macros, so this technique will work for all such scenarios. For example, it will work for **DECLARE_MESSAGE_MAP**.

Note
If you are exporting the entire class rather than selected members of the class, static data members are automatically exported.

You can use the same technique to automatically export the **CArchive** extraction operator for classes that use the **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** macros. Export the archive operator by bracketing the class declarations (located in the .H file) with the following code:

```
#undef AFX_API
#define AFX_API AFX_EXT_CLASS

<your class declarations here>

#undef AFX_API
#define AFX_API
```

Limitations of _AFXEXT

You can use the **_AFXEXT** pre-processor symbol for your extension DLLs as long as you do not have multiple layers of extension DLLs. If you have extension DLLs that call or derive from classes in your own extension DLLs, which then derive from the MFC classes, you must use your own preprocessor symbol to avoid ambiguity.

The problem is that in Win32, you must explicitly declare any data as **__declspec(dllexport)** if it is to be exported from a DLL, and **__declspec(dllimport)** if it is to be imported from a DLL. When you define **_AFXEXT**, the MFC headers make sure that **AFX_EXT_CLASS** is defined correctly.

When you have multiple layers, one symbol such as **AFX_EXT_CLASS** is not sufficient, since an extension DLL may be exporting new classes as well as importing other classes from another extension DLL. In order to deal with this problem, use a special preprocessor symbol that indicates that you are building the DLL itself versus using the DLL. For example, imagine two extension DLLs, A.DLL, and B.DLL. They each export some classes in A.H and B.H, respectively. B.DLL uses the classes from A.DLL. The header files would look something like this:

```
/* A.H */
#ifdef A_IMPL
   #define CLASS_DECL_A   __declspec(dllexport)
#else
   #define CLASS_DECL_A   __declspec(dllimport)
#endif

class CLASS_DECL_A CExampleA : public CObject
{ ... class definition ... };

/* B.H */
#ifdef B_IMPL
   #define CLASS_DECL_B   __declspec(dllexport)
#else
   #define CLASS_DECL_B   __declspec(dllimport)
#endif

class CLASS_DECL_B CExampleB : public CExampleA
{ ... class definition .. };
```

When A.DLL is built, it is built with **/D A_IMPL** and when B.DLL is built, it is built with **/D B_IMPL**. By using separate symbols for each DLL, CExampleB is exported and CExampleA is imported when building B.DLL. CExampleA is exported when building A.DLL and imported when used by B.DLL (or some other client).

This type of layering cannot be done when using the built-in **AFX_EXT_CLASS** and **_AFXEXT** preprocessor symbols. The technique described above solves this problem in a manner not unlike the mechanism MFC itself uses when building its OLE, Database, and Network extension DLLs.

Not Exporting the Entire Class

Again, you will have to take special care when you are not exporting an entire class. You have to ensure that the necessary data items created by the MFC macros are exported correctly. This can be done by re-defining **AFX_DATA** to your specific class' macro. This should be done any time you are not exporting the entire class.

For example:

```
// A.H
#ifdef A_IMPL
   #define CLASS_DECL_A  _declspec(dllexport)
#else
   #define CLASS_DECL_A  _declspec(dllimport)
   #endif

#undef  AFX_DATA
#define AFX_DATA CLASS_DECL_A

class CExampleA : public CObject
{
   DECLARE_DYNAMIC()
   CLASS_DECL_A int SomeFunction();
   //class definition
   .
   .
   .
};

#undef AFX_DATA
#define AFX_DATA
```

DllMain

The following is the exact code you should place in your main source file for your extension DLL. It should come after the standard includes. Note that when you use AppWizard to create starter files for an extension DLL, it supplies a **DllMain** for you.

```
#include "afxdllx.h"

static AFX_EXTENSION_MODULE extensionDLL;

extern "C" int APIENTRY
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID)
{
   if (dwReason == DLL_PROCESS_ATTACH)
   {
      // Extension DLL one-time initialization
      if (!AfxInitExtensionModule(
             extensionDLL, hInstance))
         return 0;

      // TODO: perform other initialization tasks here
   }
   else if (dwReason == DLL_PROCESS_DETACH)
   {
      // Extension DLL per-process termination
      AfxTermExtensionModule(extensionDLL);

          // TODO: perform other cleanup tasks here
   }
   return 1;   // ok
}
```

The call to **AfxInitExtensionModule** captures the modules runtime-classes (**CRuntimeClass** structures) as well as its object factories (**COleObjectFactory** objects) for use later when the **CDynLinkLibrary** object is created. The (optional) call to **AfxTermExtensionModule** allows MFC to cleanup the extension DLL when each process detaches (which happens when the process exits, or when the DLL is unloaded as a result of a **FreeLibrary** call) from the extension DLL. Since most extension DLLs are not dynamically loaded (usually, they are linked via their import libraries), the call to **AfxTermExtensionModule** is usually not necessary.

If your application loads and frees extension DLLs dynamically, be sure to call **AfxTermExtensionModule** as shown above. Also be sure to use **AfxLoadLibrary** and **AfxFreeLibrary** (instead of Win32 functions **LoadLibrary** and **FreeLibrary**) if your application uses multiple threads or if it dynamically loads an extension DLL. Using **AfxLoadLibrary** and **AfxFreeLibrary** insures that the startup and shutdown code that executes when the extension DLL is loaded and unloaded does not corrupt the global MFC state.

The header file AFXDLLX.H contains special definitions for structures used in extension DLLs, such as the definition for **AFX_EXTENSION_MODULE** and **CDynLinkLibrary**.

The global *extensionDLL* must be declared as shown. Unlike the 16-bit version of MFC, you can allocate memory and call MFC functions during this time, since the MFCxx.DLL is fully initialized by the time your **DllMain** is called.

Sharing Resources and Classes

Simple MFC extension DLLs need only export a few low-bandwidth functions to the client application and nothing more. More user-interface intensive DLLs may want to export resources and C++ classes to the client application.

Exporting resources is done through a resource list. In each application is a singly linked list of **CDynLinkLibrary** objects. When looking for a resource, most of the standard MFC implementations that load resources look first at the current resource module (**AfxGetResourceHandle**) and if not found walk the list of **CDynLinkLibrary** objects attempting to load the requested resource.

Dynamic creation of C++ objects given a C++ class name is similar. The MFC object deserialization mechanism needs to have all of the **CRuntimeClass** objects registered so that it can reconstruct by dynamically creating C++ object of the required type based on what was stored earlier.

If you want the client application to use classes in your extension DLL that are **DECLARE_SERIAL**, then you will need to export your classes to be visible to the client application. This is also done by walking the **CDynLinkLibrary** list.

In the case of the MFC Advanced Concepts sample DLLHUSK, the list looks something like:

```
head ->   DLLHUSK.EXE   - or -   DLLHUSK.EXE
               |                     |
          TESTDLL2.DLL           TESTDLL2.DLL
               |                     |
          TESTDLL1.DLL           TESTDLL1.DLL
               |                     |
               |                     |
            MFC80D.DLL            MFC80.DLL
```

The MFCxx.DLL is usually last on the resource and class list. MFCxx.DLL includes all of the standard MFC resources, including prompt strings for all the standard command IDs. Placing it at the tail of the list allows DLLs and the client application itself to not have a their own copy of the standard MFC resources, but to rely on the shared resources in the MFCxx.DLL instead.

Merging the resources and class names of all DLLs into the client application's name space has the disadvantage that you have to be careful what IDs or names you pick. You can of course disable this feature by not exporting either your resources or a **CDynLinkLibrary** object to the client application. The DLLHUSK sample manages the shared resource name space by using multiple header files. See Technical Note 35 for more tips on using shared resource files.

Initializing the DLL

As mentioned above, you will usually want to create a **CDynLinkLibrary** object in order to export your resources and classes to the client application. You will need to provide an exported entry point to initialize the DLL. Minimally, this is a void routine that takes no arguments and returns nothing, but it can be anything you like.

Each client application that wants to use your DLL must call this initialization routine, if you use this approach. You may also allocate this **CDynLinkLibrary** object in your **DllMain** just after calling **AfxInitExtensionModule**.

The initialization routine must create a **CDynLinkLibrary** object in the current application's heap, wired up to your extension DLL information. This can be done with the following:

```
extern "C" extern void WINAPI InitXxxDLL()
{
   new CDynLinkLibrary(extensionDLL);
}
```

The routine name, *InitXxxDLL* in this example, can be anything you want. It does not need to be `extern "C"`, but doing so makes the export list easier to maintain.

Note
If you use your extension DLL from a regular DLL, you must export this initialization function. This function must be called fro m the regular DLL before using any extension DLL classes or resources.

Exporting Entries

The simple way to export your classes is to use **__declspec(dllimport)** and **__declspec(dllexport)** on each class and global function you wish to export. This makes it a lot easier, but is less efficient than naming each entry point (described below) since you have less control over what functions are exported and you cannot export the functions by ordinal. TESTDLL1 and TESTDLL2 use this method to export their entries.

A more efficient method (and the method used by MFCxx.DLL) is to export each entry by hand by naming each entry in the .DEF file. Since we are exporting selective exports from our DLL (that is, not everything), we must decide which particular interfaces we wish to export. This is difficult since you must specify the mangled names to the linker in the form of entries in the .DEF file. Don't export any C++ classes unless you really need to have a symbolic link for it.

If you have tried exporting C++ classes with a .DEF file before, you may want to develop a tool to generate this list automatically. This can be done using a two-stage link process. Link your DLL once with no exports, and allow the linker to generate a .MAP file. The .MAP file can be used to generate a list of functions that should be exported, so with some rearranging, it can be used to generate your EXPORT entries for your .DEF file. The export list for MFCxx.DLL and the OLE and Database extension DLLs, several thousand in number, was generated with such a process (although it is not completely automatic and requires some hand tuning every once in a while).

CWinApp vs. CDynLinkLibrary

An MFC Extension DLL does not have a **CWinApp**-derived object of its own; instead it must work with the **CWinApp**-derived object of the client application. This means that the client application owns the main message pump, the idle loop and so on.

If your MFC Extension DLL needs to maintain extra data for each application, you can derive a new class from **CDynLinkLibrary** and create it in the InitXxxDLL routine describe above. When running, the DLL can check the current application's list of **CDynLinkLibrary** objects to find the one for that particular extension DLL.

Using Resources in Your DLL Implementation

As mentioned above, the default resource load will walk the list of **CDynLinkLibrary** objects looking for the first EXE or DLL that has the requested resource. All MFC APIs as well as all the internal code uses **AfxFindResourceHandle** to walk the resource list to find any resource, no matter where it may reside.

If you wish to only load resources from a specific place, use the APIs **AfxGetResourceHandle** and **AfxSetResourceHandle** to save the old handle and set the new handle. Be sure to restore the old resource handle before you return to the client application. The sample TESTDLL2 uses this approach for explicitly loading a menu.

Walking the list has the disadvantages that it is slightly slower and requires managing resource ID ranges. It has the advantage that a client application that links to several extension DLLs can use any DLL-provided resource without having to specify the DLL instance handle. **AfxFindResourceHandle** is an API used for walking the resource list to look for a given match. It takes the name and type of a resource and returns the resource handle where it was first found (or NULL).

Writing an Application That Uses the DLL Version

Application Requirements

An application that uses the shared version of MFC must follow a few simple rules:

- It must have a **CWinApp** object and follow the standard rules for a message pump.
- It must be compiled with a set of required compiler flags (see below).
- It must link with the MFCxx import libraries. By setting the required compiler flags, the MFC headers determine at link time which library the application should link with.
- To run the executable, MFCxx.DLL must be on the path or in the Windows system directory.

Building with the Development Environment

If you are using the internal makefile with most of the standard defaults, you can easily change the project to build the DLL version.

The following step assumes you have a correctly functioning MFC application linked with NAFXCWD.LIB (for debug) and NAFXCW.LIB (for retail) and you want to convert it to use the shared version of the MFC library. You are running the Visual C++ environment and have an internal project file.

1. On the **Projects** menu, click **Properties**. In the **General** page under **Project Defaults**, set Microsoft Foundation Classes to **Use MFC in a Shared DLL** (MFCxx(d).dll).

Building with NMAKE

If you are using the external makefile feature of the Visual C++, or are using NMAKE directly, you will have to edit your makefile to support compiler and linker options

Required compiler flags:

/D_AFXDLL /MD

/D_AFXDLL

The standard MFC headers need this symbol to be defined:

/MD

The application must use the DLL version of the C run-time library

All other compiler flags follow the MFC defaults (for example, _DEBUG for debug).

Edit the linker list of libraries. Change NAFXCWD.LIB to MFCxxD.LIB and change NAFXCW.LIB to MFCxx.LIB. Replace LIBC.LIB with MSVCRT.LIB. As with any other MFC library it is important that MFCxxD.LIB is placed **before** any C-runtime libraries.

Optionally add **/D_AFXDLL** to both your retail and debug resource compiler options (the one that actually compiles the resources with **/R**). This makes your final executable smaller by sharing the resources that are present in the MFC DLLs.

A full rebuild is required after these changes are made.

Building the Samples

Most of the MFC sample programs can be built from Visual C++ or from a shared NMAKE-compatible MAKEFILE from the command line.

To convert any of these samples to use MFCxx.DLL, you can load the .MAK file into the Visual C++ and set the Project options as described above. If you are using the NMAKE build, you can specify "AFXDLL=1" on the NMAKE command line and that will build the sample using the shared MFC libraries.

The MFC Advanced Concepts sample DLLHUSK is built with the DLL version of MFC. This sample not only illustrates how to build an application linked with MFCxx.DLL, but it also illustrates other features of the MFC DLL packaging option such as MFC Extension DLLs described later in this technical note.

Packaging Notes

The retail version of the DLLs (MFCxx[U].DLL) are freely redistributable. The debug version of the DLLs are not freely redistributable and should be used only during the development of your application.

The debug DLLs are provided with debugging information. By using the Visual C++ debugger, you can trace execution of your application as well as the DLL. The Release DLLs (MFCxx[U].DLL) do not contain debugging information.

If you customize or rebuild the DLLs, then you should call them something other than "MFCxx" The MFC SRC file MFCDLL.MAK describes build options and contains the logic for renaming the DLL. Renaming the files is necessary, since these DLLs are potentially shared by many MFC applications. Having your custom version of the MFC DLLs replace those installed on the system may break another MFC application using the shared MFC DLLs.

Rebuilding the MFC DLLs is not recommended.

How the MFCxx.DLL Is Implemented

The following section describes how the MFC DLL (MFCxx.DLL and MFCxxD.DLL) is implemented. Understanding the details here are also not important if all you want to do is use the MFC DLL with your application. The details here are not essential for understanding how to write an MFC extension DLL, but understanding this implementation may help you write your own DLL.

Implementation Overview

The MFC DLL is really a special case of an MFC Extension DLL as described above. It has a very large number of exports for a large number of classes. There are a few additional things we do in the MFC DLL that make it even more special than a regular Extension DLL.

Win32 Does Most of the Work

The 16-bit version of MFC needed a number of special techniques including per-app data on the stack segment, special segments created by some 80x86 assembly code, per-process exception contexts, and other techniques. Win32 directly supports per-process data in a DLL, which is what you want most of the time. For the most part MFCxx.DLL is just NAFXCW.LIB packaged in a DLL. If you look at the MFC source code, you'll find very few #ifdef _AFXDLL, since there are very few special cases that need to be made. The special cases that are there are specifically to deal with Win32 on Windows 3.1 (otherwise known as Win32s). Win32s does not support per-process DLL data directly so the MFC DLL must use the thread-local storage (TLS) Win32 APIs to obtain process local data.

Impact on Library Sources, Additional Files

The impact of the **_AFXDLL** version on the normal MFC class library sources and headers is relatively minor. There is a special version file (AFXV_DLL.H) as well as an additional header file (AFXDLL_.H) included by the main AFXWIN.H header. The AFXDLL_.H header includes the **CDynLinkLibrary** class and other implementation details of both **_AFXDLL** applications and MFC Extension DLLs. The AFXDLLX.H header is provided for building MFC Extension DLLs (see above for details).

The regular sources to the MFC library in MFC SRC have some additional conditional code under the **_AFXDLL** #ifdef. An additional source file (DLLINIT.CPP) contains the extra DLL initialization code and other glue for the shared version of MFC.

In order to build the shared version of MFC, additional files are provided. (See below for details on how to build the DLL.)

- A special makefile (MFCDLL.MAK) is used for building the DLLs. This makefile includes the standard MAKEFILE to get all the normal MFC library building rules.
- Two .DEF files are used for exporting the MFC DLL entry points for debug (MFCxxD.DEF) and release (MFCxx.DEF) versions of the DLL.
- An .RC file (MFCDLL.RC) contains all the standard MFC resources and a VERSIONINFO resource for the DLL.
- A .CLW file (MFCDLL.CLW) is provided to allow browsing the MFC classes using ClassWizard. Note: this feature is not particular to the DLL version of MFC.

Building the MFC DLL

Rebuilding the MFC DLL is intentionally difficult, so you think twice (or three times) before doing it. If you understand the potential packaging problems and redistribution restrictions described below, and you still *really* need to rebuild the MFC DLL, you can.

The MFCDLL.MAK file will build the Debug DLL with CodeView info:

```
NMAKE /f mfcdll.mak DEBUG=1 LIBNAME=MYMFC
```

The MFCDLL.MAK file will build the Release DLL without CodeView info:

```
NMAKE /f mfcdll.mak DEBUG=0 LIBNAME=MYMFC
```

This will build a private version of the MFC DLL in your MFC SRC directory with the standard MFCxx.DLL and MFCxxD.DLL names. You will need to copy them to an appropriate place on your path to use the new DLLs. The MFCDLL.MAK makefile will also rebuild the import libraries (MFCxx.LIB and MFCxxD.LIB) and place them in the standard MFC LIB directory. This will replace the prebuilt MFCxx.LIB and MFCxxD.LIB libraries, so be careful.

If you wish to redistribute a modified version of the MFC DLL library, be sure to change the name of the DLL in the MFCDLL.MAK makefile and the two .DEF files. See the makefile MFCDLL.MAK for more info.

You may modify the library and redistribute a retail (/release) of your modified library only if you rename it to something other than MFCxx.DLL. You may not redistribute the debug version of either the pre-built or custom built debug DLL.

These redistribution restrictions are primarily to avoid a proliferation of nonstandard and potentially virus containing DLLs. Ideally you should not need to rebuild the DLLs and if you redistribute your application with the prebuilt MFCxx.DLL provided with the Visual C++ product, you will avoid a lot of problems for yourself and your users.

Memory Management

An application using MFCxx.DLL uses a common memory allocator provided by MSVCRTxx.DLL, the shared C-runtime DLL. The application, any extension DLLs, and well as the MFC DLLs themselves use this shared memory allocator. By using a shared DLL for memory allocation, the MFC DLLs can allocate memory that is later freed by the application or vice versa. Because both the application and the DLL must use the same allocator, you should not override the C++ global **operator new** or **operator delete**. The same rules apply to the rest of the C run-time memory allocation routines (such as **malloc**, **realloc**, **free**, and others).

Ordinals and class __declspec(dllexport) and DLL naming

We do not use the `class` **__declspec(dllexport)** functionality of the C++ compiler. Instead, a list of exports is included with the class library sources (MFCxx.DEF and MFCxxD.DEF). Only these select set of entry points (functions and data) are exported. Other symbols, such as MFC private implementation functions or classes, are not exported All exports are done by ordinal without a string name in the resident or non-resident name table.

Using `class` **__declspec(dllexport)** may be a viable alternative for building smaller DLLs, but in the case of a large DLL like MFC, the default exporting mechanism has efficiency and capacity limits.

What this all means is that we can package a large amount of functionality in the release MFCxx.DLL that is only around 800 KB without compromising much execution or loading speed. MFCxx.DLL would have been 100K larger had this technique not been used. This also makes it possible to add additional entry points at the end of the .DEF file to allow simple versioning without compromising the speed and size efficiency of exporting by ordinal. Major version revisions in the MFC class library will change the library name. That is, MFC30.DLL is the redistributable DLL containing version 3.0 of the MFC class library. An upgrade of this DLL, say, in a hypothetical MFC 3.1, the DLL would be named MFC31.DLL instead. Again, if you modify the MFC source code to produce a custom version of the MFC DLL, use a different name (and preferably one without "MFC" in the name).

See Also

Other Resources

Technical Notes by Number
Technical Notes by Category

TN034: Writing a Windows 3.0 Compatible MFC Application

This technical note, which previously discussed compatability for Windows 3.0, is now obsolete.

The initial release of Windows NT contains the Windows 3.1 features. Therefore, the support for the older Windows 3.0 programming has been removed from the MFC provided with this version of Visual C++.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN035: Using Multiple Resource Files and Header Files with Visual C++

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes how the Visual C++ resource editor supports multiple resource files and header files shared in a single project or shared across multiple projects and how you can take advantage of that support. This note answers these questions:

- When might you want to split a project into multiple resource files and/or header files, and how you do it?
- How do you share a common header .H file between two .RC files?
- How do you divide project resources into multiple .RC files?
- How do you (and the tools) manage build dependencies between .RC, .CPP, and .H files?

You should be aware that if you add an additional resource file to your project, ClassWizard will not recognize the resources in the added file.

This note is structured to answer the above questions as follows:

- Overview of How Visual C++ Manages Resource Files and Header Files provides an overview of how the Resource Set Includes command in Visual C++ lets you use multiple resource files and header files in the same project.
- Analysis of AppWizard-created .RC and .H Files looks at the multiple resource and header files that are used by an AppWizard-created application. These files serve as a good model for additional resource files and header files you might want to add to your project.
- Including Additional Header Files describes where you might want to include multiple header files, and provides details how to do so.
- Sharing a Header File Between Two .RC Files shows how you can share one header file between multiple .RC files in different projects, or perhaps in the same project.
- Using Multiple Resource Files in the Same Project describes where you might want to break up your project into multiple .RC files, and provides details how to do so.
- Enforcement of Non-Editable Visual C++ Files describes how you can make sure Visual C++ does not edit and unintentionally reformat a custom resource.
- Managing Symbols Shared by Multiple Visual C++-Edited .RC Files describes how to share the same symbols across multiple .RC files and how to avoid assigning duplicate ID numeric values.
- Managing Dependencies Between .RC, .CPP, and .H Files describes how Visual C++ avoids unnecessary recompiling .CPP files that are dependent on resource symbol files.
- How Visual C++ Manages Set Includes Information provides technical details about how Visual C++ keeps track of multiple (nested) .RC files and multiple header files that are #include'd by an .RC file.

Overview of How Visual C++ Manages Resource Files and Header Files

Visual C++ manages a single .RC resource file and a corresponding .H header file as a tightly coupled pair of files. When you edit and save resources in an .RC file, you indirectly edit and save symbols in the corresponding .H file. Although you can open and edit multiple .RC files at a time (using Visual C++'s MDI user interface) for any given .RC file you indirectly edit exactly one corresponding header file.

Symbol Header File

By default, Visual C++ always names the corresponding header file RESOURCE.H, regardless of the name of the resource file

(e.g., MYAPP.RC). Using the **Resource Includes** command from the **View** menu in Visual C++, you can change the name of this header file by updating the Symbol Header File file in the **Set Includes** dialog box.

Read-Only Symbol Directives

Although Visual C++ only edits one header file for any given .RC file, Visual C++ supports references to symbols defined in additional read-only header files. Using the **Resource Includes** command from the **View** menu in Visual C++, you can specify any number of additional read-only header files as Read-Only Symbol Directives. The "read-only" restriction means that when you add a new resource in the .RC file, you can use a symbol defined in the read-only header file; but if you delete the resource, the symbol still remains defined in the read-only header file. You cannot change the numeric value assigned to a read-only symbol.

Compile-Time Directives

Visual C++ also supports nesting of resource files, where one .RC file is #include'd within another. When you edit a given .RC file using Visual C++, any resources in the #include'd files are not visible. But when you compile the .RC file, the #include'd files are also compiled. Using the **Resource Includes** command from the **View** menu in Visual C++, you can specify any number of #include'd .RC files as Compile-Time Directives.

Note what happens if you read into Visual C++ an .RC file that #include's another .RC file that is *not* specified as a Compile-Time Directive. This situation might arise when you bring to Visual C++ an .RC file that you had been previously maintaining manually with a text editor. When Visual C++ reads the #include'd .RC file, it merges the #include'd resources into the parent .RC file. When you save the parent .RC file, the #include statement, in effect, will be replaced by the #include'd resources. If you do not want this merge to happen, you should remove the #include statement from the parent .RC file *prior* to reading it into Visual C++; then using Visual C++, add back the same #include statement as a Compile-Time Directive.

Visual C++ saves in an .RC file the three kinds of above Set Includes information (Symbol Header File, Read-Only Symbol Directives, and Compile-Time Directives) in #include directives *and* in TEXTINCLUDE resources. The TEXTINCLUDE resources, an implementation detail that you do not normally need to deal with, are explained in
How Visual C++ Manages Set Includes Information.

Analysis of AppWizard-Created .RC and .H Files

Examining the application code produced by AppWizard provides insight into how Visual C++ manages multiple resource files and header files. The code excerpts examined below are from a MYAPP application produced by AppWizard using the default options.

An AppWizard-created application uses multiple resource files and multiple header files, as summarized in the diagram below:

```
    RESOURCE.H     AFXRES.H
           \       /
            \     /
           MYAPP.RC
               |
               |
         RES\MYAPP.RC2
         AFXRES.RC
         AFXPRINT.RC
```

You can view these multiple file relationships using the Visual C++ File/Set Includes command.

MYAPP.RC

The application resource file that you edit using Visual C++.

RESOURCE.H is the application-specific header file. It is always named RESOURCE.H by AppWizard, consistent with Visual C++'s default naming of the header file. The #include for this header file is the first statement in the resource file (MYAPP.RC):

```
//Microsoft Visual C++ generated resource script
//
#include "resource.h"
```

RES\MYAPP.RC2

Contains resources that will not be edited by Visual C++ but will be included in the final compiled .EXE file. AppWizard creates no such resources by default, since Visual C++ can edit all of the standard resources, including the version resource (a new feature in this release). An empty file is generated by AppWizard in case you wish to add your own custom formatted

resources to this file.

If you use custom formatted resources, you can add them to RES\MYAPP.RC2 and edit them using the Visual C++ text editor.

AFXRES.RC and AFXPRINT.RC contain standard resources required by certain features of the framework. Like RES\MYAPP.RC2, these two framework-provided resource files are #include'd at the end of MYAPP.RC, and they are specified in the Compile-Time Directives of the Set Includes dialog box. Thus, you do not directly view or edit these framework resources while you edit MYAPP.RC in Visual C++, but they are compiled into the application's binary .RES file and final .EXE file. For more information on the standard framework resources, including procedures for modifying them, see Technical Note 23.

AFXRES.H defines standard symbols, such as **ID_FILE_NEW**, used by the framework and specifically used in AFXRES.RC. AFXRES.H also #include's WINRES.H, which contains a subset of WINDOWS.H that are needed by Visual C++ generated .RC files as well as AFXRES.RC. The symbols defined in AFXRES.H are available as you edit the application resource file (MYAPP.RC). For example, **ID_FILE_NEW** is used for the File New menu item in MYAPP.RC's menu resource. You cannot change or delete these framework-defined symbols.

Including Additional Header Files

The AppWizard-created application includes only two header files: RESOURCE.H and AFXRES.H. Only RESOURCE.H is application-specific. You may need to include additional read-only header files in the following cases:

The header file is provided by an external source, or you want to share the header file among multiple projects or multiple parts of the same project.

The header file has formatting and comments that you do not want Visual C++ to change or filter out when it saves the file. For example, maybe you want to preserve #define's that use symbolic arithmetic such as:

```
#define RED 0
#define BLUE 1
#define GREEN 2
#define ID_COLOR_BUTTON 1001
#define ID_RED_BUTTON (ID_COLOR_BUTTON + RED)
#define ID_BLUE_BUTTON (ID_COLOR_BUTTON + BLUE)
#define ID_GREEN_BUTTON (ID_COLOR_BUTTON + GREEN)
```

You can include additional read-only header files by using the **Resource Includes** command to specify the #include statement as a second Read-Only Symbol Directive, as in:

```
#include "afxres.h"
#include "second.h"
```

The new file relationship diagram now looks like this:

```
                  AFXRES.H
    RESOURCE.H    SECOND.H
          \       /
           \     /
          MYAPP.RC
              |
              |
        RES\MYAPP.RC2
        AFXRES.RC
        AFXPRINT.RC
```

Sharing a Header File Between Two .RC Files

You may want to share a header file between two .RC files that are in different projects, or possibly the same project. To do so, simply apply the Read-Only Directives technique described above to both .RC files. In the case where the two .RC files are for different applications (different projects), the result is illustrated in the following diagram:

```
     RESOURCE.H   AFXRES.H   RESOURCE.H
    (for MYAPP1)  SECOND.H   (for MYAPP2)
          \       /     \       /
           \     /       \     /
          MYAPP1.RC      MYAPP2.RC
```

```
            /    \        /     \
           /      \      /       \
RES\MYAPP1.RC2  AFXRES.RC     RES\MYAPP2.RC2
                AFXPRINT.RC
```

The case where the second header file is shared by two .RC files in the same application (project) is discussed below.

Using Multiple Resource Files in the Same Project

Visual C++ and the Resource Compiler support multiple .RC files in the same project through #include's of one .RC file within another. Multiple nesting is allowed. There are various reasons to split your project's resources into multiple .RC files:

- It is easier to manage a large number of resources among multiple project team members if you split the resources into multiple .RC files. If you use a source control management package for checking out files and checking in changes, splitting the resources into multiple .RC files will give you finer control over managing changes to resources.
- If you want to use preprocessor directives, such as #ifdef, #endif, and #define, for portions of your resources, you must isolate them in read-only resources that will be compiled by the Resource Compiler.
- Component .RC files will load and save faster in Visual C++ than one composite .RC file.
- If you want to maintain a resource with a text editor in a human-readable form, you should keep it in a .RC file separate from the one Visual C++ edits.
- If you need to keep a user-defined resource in a binary or text form that is interpretable by another specialized data editor, then you should keep it in a separate .RC file so Visual C++ does not change the format to hexadecimal data. The .WAV (sound) file resources in the MFC Advanced Concepts sample SPEAKN are a good example.

You can #include a SECOND.RC in the Compile-Time Directives in the Set Includes dialog box:

```
#include "res\myapp.rc2"  // non-Visual C++ edited resources
#include "second.rc"  // THE SECOND .RC FILE

#include "afxres.rc"  // Standard components
#include "afxprint.rc"  // printing/print preview resources
```

The result is illustrated in the following diagram:

```
   RESOURCE.H     AFXRES.H
          \       /
           \     /
          MYAPP.RC
              |
              |
        RES\MYAPP.RC2
        SECOND.RC
        AFXRES.RC
        AFXPRINT.RC
```

Using Compile-Time Directives, you can organize your Visual C++-editable and non-editable resources into multiple .RC files, where the "master" MYAPP.RC does nothing but #include the other .RC files. If you are using a Visual C++ project .MAK file, then you should include the "master" .RC file in the project so that all the #include'd resources are compiled with your application.

Enforcement of Noneditable Visual C++ Files

The AppWizard-created RES\MYAPP.RC2 file is an example of a file that contains resources that you do *not* want to accidentally read into Visual C++ and then write it back out with loss of formatting information. To protect against this, place the following lines in the beginning of the RES\MYAPP.RC2 file:

```
#ifdef APSTUDIO_INVOKED
    #error this file is not editable by Visual C++
#endif //APSTUDIO_INVOKED
```

When Visual C++ compiles the .RC file, it defines **APSTUDIO_INVOKED** as well as **RC_INVOKED**. If the AppWizard-created

file structure is corrupted and Visual C++ reads the #error line above, it reports a fatal error and abort the reading of the .RC file.

Managing Symbols Shared by Multiple Visual C++-Edited .RC Files

Two issues arise when you split up your resources into multiple .RC files that you want to edit separately in Visual C++:

- You might want to share the same symbols across multiple .RC files.
- You need to help Visual C++ avoid assigning the same ID numeric values to distinct resources (symbols).

The following diagram illustrates an organization of .RC and .H files that deals with the first issue:

```
                MYAPP.RC
               /         \
              /           \
 MYSTRS.H    / MYSHARED.H  \  MYMENUS.H
      \    /    /      \   \    \
       \  /    /        \   \    \
    MYSTRS.RC            MYMENUS.RC
```

In this example, string resources are kept in one resource file, MYSTRS.RC, and menus are kept in another, MYMENUS.RC. Some symbols, such as for commands, may need to be shared between the two files. For example, a ID_TOOLS_SPELL may be the menu command ID for the Spell item in a Tools menu; and it may also be the string ID of the command prompt displayed by the framework in the application's main window status bar.

The ID_TOOLS_SPELL symbol is kept in the shared header file, MYSHARED.H. You maintain this shared header file manually with a text editor; Visual C++ does not directly edit it. In the two resource files MYSTRS.RC and MYMENUS.RC, you specify #include MYSHARED.H in the Read-Only Directives for MYAPP.RC, using the **Resource Includes** command, as described earlier.

It is most convenient to anticipate a symbol you will share before you attempt use it to identify any resource. Add the symbol to the shared header file and, if you have not already #include'd the shared header file in the Read-Only Directives for the .RC file, do so before using the symbol. If you did not anticipate sharing the symbol in this way, then you will have to manually (using a text editor) move the #define statement for the symbol from, say, MYMENUS.H to MYSHARED.H before using it in MYSTRS.RC.

When you manage symbols in multiple .RC files, you also must help Visual C++ avoid assigning the same ID numeric values to distinct resources (symbols). For any given .RC file, Visual C++ incrementally assigns IDs in each of four ID domains. Between editing sessions, Visual C++ keeps track of the last ID it assigned in each of the domains in the symbol header file for the .RC file. Here is what the APS_NEXT values are for an empty (new) .RC file:

```
#define _APS_NEXT_RESOURCE_VALUE  101
#define _APS_NEXT_COMMAND_VALUE   40001
#define _APS_NEXT_CONTROL_VALUE   1000
#define _APS_NEXT_SYMED_VALUE     101
```

_APS_NEXT_RESOURCE_VALUE is the next symbol value that will be used for a dialog resource, menu resource, and so on. The valid range for resource symbol values is 1 to 0x6FFF.

_APS_NEXT_COMMAND_VALUE is the next symbol value that will be used for a command identification. The valid range for command symbol values is 0x8000 to 0xDFFF.

_APS_NEXT_CONTROL_VALUE is the next symbol value that will be used for a dialog control. The valid range for dialog control symbol values is 8 to 0xDFFF.

_APS_NEXT_SYMED_VALUE is the next symbol value that will be issued when you manually assign a symbol value using the New command in the Symbol Browser.

Visual C++ starts with slightly higher values that the lowest legal value when creating a new .RC file. AppWizard will also initialize these values to something more appropriate for MFC applications. For more information about ID value ranges, see Technical Note 20.

Now every time you create a new resource file, even in the same project, Visual C++ defines the same **_APS_NEXT_** values. This means that if you add, say, multiple dialogs in two different .RC files, it is highly likely that the same #define value will be assigned to different dialogs. For example, IDD_MY_DLG1 in the first .RC file might be assigned the same number, 101, as IDD_MY_DLG2 in a second .RC file.

To avoid this, you should reserve a separate numeric range for each of the four domains of IDs in the respective .RC files. Do this by manually updating the **_APS_NEXT** values in each of the .RC files *before* you start adding resources. For example, if the first .RC file uses the default **_APS_NEXT** values, then you might want to assign the following **_APS_NEXT** values to the second .RC file:

```
#define _APS_NEXT_RESOURCE_VALUE  2000
#define _APS_NEXT_COMMAND_VALUE   42000
#define _APS_NEXT_CONTROL_VALUE   2000
#define _APS_NEXT_SYMED_VALUE     2000
```

Of course, it is still possible that Visual C++ will assign so many IDs in the first .RC file that the numeric values start to overlap those reserved for the second .RC file. You should reserve sufficiently large ranges so that this does not happen.

Managing Dependencies Between .RC, .CPP, and .H Files

When Visual C++ saves an .RC file, it also saves symbol changes to the corresponding RESOURCE.H file. Any of your .CPP files that refer to resources in the .RC file must #include the RESOURCE.H file, usually from within your project's master header file. This leads to an undesirable side-effect because of the development environment's internal project management which scans source files for header dependencies. Every time you add a new symbol in Visual C++, all the .CPP file that #include RESOURCE.H would need to be recompiled.

Visual C++, circumvents the dependency on RESOURCE.H by including the following comment as the first line of the RESOURCE.H file:

```
//{{NO_DEPENDENCIES}}
```

The development environment interprets this comment by ignoring the changes to RESOURCE.H so that dependent .CPP files will not need to be recompiled.

Visual C++ always adds the //{{NO_DEPENDENCIES}} comment line to a .RC file when it saves the file. In some cases, circumventing of the build dependency on RESOURCE.H may lead to run-time errors undetected at link time. For example, if you use the Symbol Browser to change the numeric value assigned to a symbol for a resource, the resource will not be correctly found and loaded at application run-time if the .CPP file referring to the resource is not recompiled. In such cases, you should explicitly recompile any .CPP files that you know are affected by the symbol changes in RESOURCE.H or select **Rebuild All**. If you have the need to frequently change symbol values for a certain group of resources, you will probably find it more convenient and safer to break out these symbols into a separate read-only header file, as described in the above section Including Additional Header Files.

How Visual C++ Manages Set Includes Information

As discussed above, the File menu Set Includes command lets you specify three types of information:

- Symbol Header File
- Read-Only Symbol Directives
- Compile-Time Directives

The following describes how Visual C++ maintains this information in a .RC file. You do not need this information to use Visual C++, but it may enhance your understanding so that you can more confidently use the Set Includes feature.

Each of the above three types of Set Includes information is stored in the .RC file in two forms: (1) as #include or other directives interpretable by the Resource Compiler, and (2) as special TEXTINCLUDE resources interpretable only by Visual C++.

The purpose of the TEXTINCLUDE resource is to safely store Set Include information in a form that is readily presentable in Visual C++'s **Set Includes** dialog box. TEXTINCLUDE is a *resource type* defined by Visual C++. Visual C++ recognizes three specific TEXTINCLUDE resources that have the resource identification numbers 1, 2 and 3:

TEXTINCLUDE resource ID	Type of Set Includes information
1	Symbol Header File
2	Read-Only Symbol Directives
3	Compile-Time Directives

Each of the three types of Set Includes information is illustrated by the default MYAPP.RC and RESOURCE.H files created by AppWizard, as described below. The extra \0 and "" tokens between BEGIN and END blocks are required by the RC syntax to specify zero terminated strings and the double quote character respectively.

Symbol Header File

The form of the Symbol Header File information interpreted by the Resource Compiler is simply a #include statement:

```
#include "resource.h"
```

The corresponding TEXTINCLUDE resource is:

```
1 TEXTINCLUDE DISCARDABLE
BEGIN
   #resource.h\0"
END
```

Read-Only Symbol Directives

Read-Only Symbol Directives are included at the top of MYAPP.RC in the following form interpretable by the Resource Compiler:

```
#include "afxres.h"
```

The corresponding TEXTINCLUDE resource is:

```
2 TEXTINCLUDE DISCARDABLE
BEGIN
   "#include ""afxres.h""\r\n"
   "\0"
END
```

Compile-Time Directives

Compile-Time Directives are included at the end of MYAPP.RC in the following form interpretable by the Resource Compiler:

```
#ifndef APSTUDIO_INVOKED
///////////////////////
//
// From TEXTINCLUDE 3
//
#include "res\myapp.rc2"  // non-Visual C++ edited resources

#include "afxres.rc"  // Standard components
#include "afxprint.rc"  // printing/print preview resources
#endif  // not APSTUDIO_INVOKED
```

The #ifndef APSTUDIO_INVOKED directive instructs Visual C++ to skip over Compile-Time Directives.

The corresponding TEXTINCLUDE resource is:

```
3 TEXTINCLUDE DISCARDABLE
BEGIN
"#include ""res\myapp.rc2""  // non-Visual C++ edited resources\r\n"
"\r\n"
"#include ""afxres.rc""  // Standard components\r\n"
"#include ""afxprint.rc""  // printing/print preview resources\r\n"
"\0"
END
```

See Also

Other Resources

Technical Notes by Number

TN036: Using CFormView with AppWizard and ClassWizard

This technical note described how to modify an AppWizard generated application so that it used a **CFormView** instead of the default **CView** as its main view class. This is supported directly with this version of Visual C++.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN037: Multithreaded MFC 2.1 Applications

This technical note originally described the limitations of multithreaded programs with MFC 2.1, originally provided with Visual C++ 1.0 for Windows NT. MFC 3.0 supports multithreading directly and is documented. See that reference for more information.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN038: MFC/OLE IUnknown Implementation

Note

The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

At the heart of OLE 2 is the "OLE Component Object Model", or COM. COM defines a standard for how cooperating objects communicate to one another. This includes the details of what an "object" looks like, including how methods are dispatched on an object. COM also defines a base class, from which all COM compatible classes are derived. This base class is **IUnknown**. Although the **IUnknown** interface is referred to as a C++ class, COM is not specific to any one language — it can be implemented in C, PASCAL, or any other language that can support the binary layout of a COM object.

OLE refers to all classes derived from **IUnknown** as "interfaces." This is an important distinction, since an "interface" such as **IUnknown** carries with it no implementation. It simply defines the protocol by which objects communicate, not the specifics of what those implementations do. This is reasonable for a system that allows for maximum flexibility. It is MFC's job to implement a default behavior for MFC/C++ programs.

To understand MFC's implementation of **IUnknown** you must first understand what this interface is. A simplified version of **IUnknown** is defined below:

```
class IUnknown
{
public:
    virtual HRESULT QueryInterface(REFIID iid, void** ppvObj) = 0;
    virtual ULONG AddRef() = 0;
    virtual ULONG Release() = 0;
};
```

Note

Certain necessary calling convention details, such as **__stdcall** are left out for this illustration.

The **AddRef** and **Release** member functions control memory management of the object. COM uses a reference counting scheme to keep track of objects. An object is never referenced directly as you would in C++. Instead, COM objects are always referenced through a pointer. To release the object when the owner is done using it, the object's **Release** member is called (as opposed to using operator delete, as would be done for a traditional C++ object). The reference counting mechanism allows for multiple references to a single object to be managed. An implementation of **AddRef** and **Release** maintains a reference count on the object — the object is not deleted until its reference count reaches zero.

AddRef and **Release** are fairly straightforward from an implementation standpoint. Here is a trivial implementation:

```
ULONG CMyObj::AddRef()
{
    return ++m_dwRef;
}

ULONG CMyObj::Release()
{
    if (--m_dwRef == 0)
    {
        delete this;
        return 0;
    }
    return m_dwRef;
}
```

The **QueryInterface** member function is a little more interesting. It is not very interesting to have an object whose only

member functions are **AddRef** and **Release** — it would be nice to tell the object to do more things than **IUnknown** provides. This is where **QueryInterface** is useful. It allows you to obtain a different "interface" on the same object. These interfaces are usually derived from **IUnknown** and add additional functionality by adding new member functions. COM interfaces never have member variables declared in the interface, and all member functions are declared as pure-virtual. For example,

```
class IPrintInterface : public IUnknown
{
public:
    virtual void PrintObject() = 0;
};
```

To get an **IPrintInterface** if you only have an **IUnknown**, call **IUnknown::QueryInterface** using the **IID** of the **IPrintInterface**. An **IID** is a 128-bit number that uniquely identifies the interface. There is an **IID** for each interface that either you or OLE define. If *pUnk* is a pointer to an **IUnknown** object, the code to retrieve an **IPrintInterface** from it might be:

```
IPrintInterface* pPrint = NULL;
if (pUnk->QueryInterface(IID_IPrintInterface,
    (void**)&pPrint) == NOERROR)
{
    pPrint->PrintObject();
    pPrint->Release();
        // release pointer obtained via QueryInterface
}
```

That seems fairly easy, but how would you implement an object supporting both the IPrintInterface and **IUnknown** interface? In this case it is simple since the IPrintInterface is derived directly from **IUnknown** — by implementing IPrintInterface, **IUnknown** is automatically supported. For example:

```
class CPrintObj : public CPrintInterface
{
    virtual HRESULT QueryInterface(REFIID iid, void** ppvObj);
    virtual ULONG AddRef();
    virtual ULONG Release();
    virtual void PrintObject();
};
```

The implementations of **AddRef** and **Release** would be exactly the same as those implemented above. **CPrintObj::QueryInterface** would look something like this:

```
HRESULT CPrintObj::QueryInterface(REFIID iid, void FAR* FAR* ppvObj)
{
    if (iid == IID_IUnknown || iid == IID_IPrintInterface)
    {
        *ppvObj = this;
        AddRef();
        return NOERROR;
    }
    return E_NOINTERFACE;
}
```

As you can see, if the interface identifier (**IID**) is recognized, a pointer is returned to your object; otherwise an error occurs. Also note that a successful **QueryInterface** results in an implied **AddRef**. Of course, you'd also have to implement **CEditObj::Print**. That is simple because the **IPrintInterface** was directly derived from the **IUnknown** interface. However if you wanted to support two different interfaces, both derived from **IUnknown**, consider the following:

```
class IEditInterface : public IUnkown
{
public:
    virtual void EditObject() = 0;
};
```

Although there are a number of different ways to implement a class supporting both **IEditInterface** and **IPrintInterface**,

including using C++ multiple inheritance, this note will concentrate on the use of nested classes to implement this functionality.

```
class CEditPrintObj
{
public:
    CEditPrintObj();

    HRESULT QueryInterface(REFIID iid, void**);
    ULONG AddRef();
    ULONG Release();
    DWORD m_dwRef;

    class CPrintObj : public IPrintInterface
    {
    public:
        CEditPrintObj* m_pParent;
        virtual HRESULT QueryInterface(REFIID iid, void** ppvObj);
        virtual ULONG AddRef();
        virtual ULONG Release();
    } m_printObj;

    class CEditObj : public IEditInterface
    {
    public:
        CEditPrintObj* m_pParent;
        virtual ULONG QueryInterface(REFIID iid, void** ppvObj);
        virtual ULONG AddRef();
        virtual ULONG Release();
    } m_editObj;
};
```

The entire implementation is included below:

```
CEditPrintObj::CEditPrintObj()
{
    m_editObj.m_pParent = this;
    m_printObj.m_pParent = this;
}

ULONG CEditPrintObj::AddRef()
{
    return ++m_dwRef;
}

CEditPrintObj::Release()
{
    if (--m_dwRef == 0)
    {
        delete this;
        return 0;
    }
    return m_dwRef;
}

HRESULT CEditPrintObj::QueryInterface(REFIID iid, void** ppvObj)
{
    if (iid == IID_IUnknown || iid == IID_IPrintInterface)
    {
        *ppvObj = &m_printObj;
        AddRef();
        return NOERROR;
    }
    else if (iid == IID_IEditInterface)
    {
        *ppvObj = &m_editObj;
        AddRef();
```

```
        return NOERROR;
    }
    return E_NOINTERFACE;
}

ULONG CEditPrintObj::CEditObj::AddRef()
{
    return m_pParent->AddRef();
}

ULONG CEditPrintObj::CEditObj::Release()
{
    return m_pParent->Release();
}

HRESULT CEditPrintObj::CEditObj::QueryInterface(
    REFIID iid, void** ppvObj)
{
    return m_pParent->QueryInterface(iid, ppvObj);
}

ULONG CEditPrintObj::CPrintObj::AddRef()
{
    return m_pParent->AddRef();
}

ULONG CEditPrintObj::CPrintObj::Release()
{
    return m_pParent->Release();
}

HRESULT CEditPrintObj::CPrintObj::QueryInterface(
    REFIID iid, void** ppvObj)
{
    return m_pParent->QueryInterface(iid, ppvObj);
}
```

Notice that most of the **IUnknown** implementation is placed into the **CEditPrintObj** class rather than duplicating the code in CEditPrintObj::CEditObj and CEditPrintObj::CPrintObj. This reduces the amount of code and avoids bugs. The key point here is that from the IUnknown interface it is possible to call **QueryInterface** to retrieve any interface the object might support, and from each of those interfaces it is possible to do the same. This means that all **QueryInterface** functions available from each interface must behave exactly the same way. In order for these embedded objects to call the implementation in the "outer object", a back-pointer is used (m_pParent). The m_pParent pointer is initialized during the CEditPrintObj constructor. Then you would implement CEditPrintObj::CPrintObj::PrintObject and CEditPrintObj::CEditObj::EditObject as well. Quite a bit of code was added to add one feature — the ability to edit the object. Fortunately, it is quite uncommon for interfaces to have only a single member function (although it does happen) and in this case, EditObject and PrintObject would usually be combined into a single interface.

That's a lot of explanation and a lot of code for such a simple scenario. The MFC/OLE classes provide a simpler alternative. The MFC implementation uses a technique similar to the way Windows messages are wrapped with Message Maps. This facility is called *Interface Maps* and is discussed in the next section.

MFC Interface Maps

MFC/OLE includes an implementation of "Interface Maps" similar to MFC's "Message Maps" and "Dispatch Maps" in concept and execution. The core features of MFC's Interface Maps are as follows:

- A standard implementation of **IUnknown**, built into the **CCmdTarget** class.
- Maintenance of the reference count, modified by **AddRef** and **Release**
- Data driven implementation of **QueryInterface**

In addition, interface maps support the following advanced features:

- Support for creating aggregatable COM objects
- Support for using aggregate objects in the implementation of a COM object

- The implementation is hookable and extensible

For more information on aggregation, see the *OLE Programmer's Reference*.

MFC's interface map support is rooted in the **CCmdTarget** class. **CCmdTarget** "*has-a*" reference count as well as all the member functions associated with the **IUnknown** implementation (the reference count for example is in **CCmdTarget**). To create a class that supports OLE COM, you derive a class from **CCmdTarget** and use various macros as well as member functions of **CCmdTarget** to implement the desired interfaces. MFC's implementation uses nested classes to define each interface implementation much like the example above. This is made easier with a standard implementation of IUnknown as well as a number of macros that eliminate some of the repetitive code.

Interface Map Basics

To implement a class using MFC's interface maps

1. Derive a class either directly or indirectly from **CCmdTarget.**
2. Use the **DECLARE_INTERFACE_MAP** function in the derived class definition.
3. For each interface you wish to support, use the **BEGIN_INTERFACE_PART** and **END_INTERFACE_PART** macros in the class definition.
4. In the implementation file, use the **BEGIN_INTERFACE_MAP** and **END_INTERFACE_MAP** macros to define the class's interface map.
5. For each IID supported, use the **INTERFACE_PART** macro between the **BEGIN_INTERFACE_MAP** and **END_INTERFACE_MAP** macros to map that IID to a specific "part" of your class.
6. Implement each of the nested classes that represent the interfaces you support.
7. Use the **METHOD_PROLOGUE** macro to access the parent, **CCmdTarget**-derived object.
8. **AddRef**, **Release**, and **QueryInterface** can delegate to the **CCmdTarget** implementation of these functions (**ExternalAddRef**, **ExternalRelease**, and **ExternalQueryInterface**).

The CPrintEditObj example above could be implemented as follows:

```
class CPrintEditObj : public CCmdTarget
{
public:
    // member data and member functions for CPrintEditObj go here

// Interface Maps
protected:
    DECLARE_INTERFACE_MAP()

    BEGIN_INTERFACE_PART(EditObj, IEditInterface)
        STDMETHOD_(void, EditObject)();
    END_INTERFACE_PART(EditObj)

    BEGIN_INTERFACE_PART(PrintObj, IPrintInterface)
        STDMETHOD_(void, PrintObject)();
    END_INTERFACE_PART(PrintObj)
};
```

The above declaration creates a class derived from **CCmdTarget**. The **DECLARE_INTERFACE_MAP** macro tells the framework that this class will have a custom interface map. In addition, the **BEGIN_INTERFACE_PART** and **END_INTERFACE_PART** macros define nested classes, in this case with names CEditObj and CPrintObj (the X is used only to differentiate the nested classes from global classes which start with "C" and interface classes which start with "I"). Two nested members of these classes are created: m_CEditObj, and m_CPrintObj, respectively. The macros automatically declare the **AddRef**, **Release**, and **QueryInterface** functions; therefore you only declare the functions specific to this interface: EditObject and PrintObject (the OLE macro **STDMETHOD** is used such so that **_stdcall** and virtual keywords are provided as appropriate for the target platform).

To implement the interface map for this class:

```
BEGIN_INTERFACE_MAP(CPrintEditObj, CCmdTarget)
```

```
    INTERFACE_PART(CPrintEditObj, IID_IPrintInterface, PrintObj)
    INTERFACE_PART(CPrintEditObj, IID_IEditInterface, EditObj)
END_INTERFACE_MAP()
```

This connects the IID_IPrintInterface IID with m_CPrintObj and IID_IEditInterface with m_CEditObj respectively. The **CCmdTarget** implementation of **QueryInterface** (**CCmdTarget::ExternalQueryInterface**) uses this map to return pointers to m_CPrintObj and m_CEditObj when requested. It is not necessary to include an entry for **IID_IUnknown**; the framework will use the first interface in the map (in this case, m_CPrintObj) when **IID_IUnknown** is requested.

Even though the **BEGIN_INTERFACE_PART** macro automatically declared the **AddRef**, **Release** and **QueryInterface** functions for you, you still need to implement them:

```
ULONG FAR EXPORT CEditPrintObj::XEditObj::AddRef()
{
    METHOD_PROLOGUE(CEditPrintObj, EditObj)
    return pThis->ExternalAddRef();
}

ULONG FAR EXPORT CEditPrintObj::XEditObj::Release()
{
    METHOD_PROLOGUE(CEditPrintObj, EditObj)
    return pThis->ExternalRelease();
}

HRESULT FAR EXPORT CEditPrintObj::XEditObj::QueryInterface(
    REFIID iid, void FAR* FAR* ppvObj)
{
    METHOD_PROLOGUE(CEditPrintObj, EditObj)
    return (HRESULT)pThis->ExternalQueryInterface(&iid, ppvObj);
}

void FAR EXPORT CEditPrintObj::XEditObj::EditObject()
{
    METHOD_PROLOGUE(CEditPrintObj, EditObj)
    // code to "Edit" the object, whatever that means...
}
```

The implementation for CEditPrintObj::CPrintObj, would be similar to the above definitions for CEditPrintObj::CEditObj. Although it would be possible to create a macro that could be used to automatically generate these functions (but earlier in MFC/OLE development this was the case), it becomes difficult to set break points when a macro generates more than one line of code. For this reason, this code is expanded manually.

By using the framework implementation of message maps, there are a number of things that were not necessary to do:

- Implement QueryInterface
- Implement AddRef and Release
- Declare either of these built-in methods on both of your interfaces

In addition, the framework uses message maps internally. This allows you to derive from a framework class, say **COleServerDoc**, that already supports certain interfaces and provides either replacements or additions to the interfaces provided by the framework. This is enabled by the fact that the framework fully supports *inheriting* an interface map from a base class — that is the reason why **BEGIN_INTERFACE_MAP** takes as its second parameter the name of the base class.

Note

It is generally not possible to reuse the implementation of MFC's built-in implementations of the OLE interfaces by *inheriting* the embedded specialization of that interface from the MFC version. This is not possible because the use of the **METHOD_PROLOGUE** macro to get access to the containing **CCmdTarget**-derived object implies a **fixed offset** of the embedded object from the **CCmdTarget**-derived object. This means, for example, you cannot derive an embedded XMyAdviseSink from MFC's implementation in **COleClientItem::XAdviseSink**, because XAdviseSink relies on being at a specific offset from the top of the **COleClientItem** object.

Note

You can, however, delegate to the MFC implementation for all of the functions that you want MFC's default behavior. This is done in the MFC implementation of **IOleInPlaceFrame** (XOleInPlaceFrame) in the **COleFrameHook** class (it delegates to m_xOleInPlaceUIWindow for many functions). This design was chosen to reduce the runtime size of objects which implement many interfaces; it eliminates the need for a back-pointer (such as the way m_pParent was used in the previous section).

Aggregation and Interface Maps

In addition to supporting stand-alone COM objects, MFC also supports aggregation. Aggregation itself is too complex a topic to discuss here; refer to the *OLE Programmer's Reference* for more information on aggregation. This note will simply describe the support for aggregation built into the framework and interface maps.

There are two ways to use aggregation: (1) using a COM object that supports aggregation, and (2) implementing an object that can be aggregated by another. These capabilities can be referred to as "using an aggregate object" and "making an object aggregatable". MFC supports both.

Using an Aggregate Object

To use an aggregate object, there needs to be some way to tie the aggregate into the QueryInterface mechanism. In other words, the aggregate object must behave as though it is a native part of your object. So how does this tie into MFC's interface map mechanism? In addition to the **INTERFACE_PART** macro, where a nested object is mapped to an IID, you can also declare an aggregate object as part of your **CCmdTarget** derived class. To do so, the **INTERFACE_AGGREGATE** macro is used. This allows you to specify a member variable (which must be a pointer to an **IUnknown** or derived class), which is to be integrated into the interface map mechanism. If the pointer is not NULL when **CCmdTarget::ExternalQueryInterface** is called, the framework will automatically call the aggregate object's **QueryInterface** member function, if the **IID** requested is not one of the native **IID**s supported by the **CCmdTarget** object itself.

To use the INTERFACE_AGGREGATE macro

1. Declare a member variable (an **IUnknown***) which will contain a pointer to the aggregate object.
2. Include an **INTERFACE_AGGREGATE** macro in your interface map, which refers to the member variable by name.
3. At some point (usually during **CCmdTarget::OnCreateAggregates**), initialize the member variable to something other than NULL.

For example:

```
class CAggrExample : public CCmdTarget
{
public:
    CAggrExample();

protected:
    LPUNKNOWN m_lpAggrInner;
    virtual BOOL OnCreateAggregates();

    DECLARE_INTERFACE_MAP()
    // "native" interface part macros may be used here
};

CAggrExample::CAggrExample()
{
    m_lpAggrInner = NULL;
}

BOOL CAggrExample::OnCreateAggregates()
{
    // wire up aggregate with correct controlling unknown
    m_lpAggrInner = CoCreateInstance(CLSID_Example,
        GetControllingUnknown(), CLSCTX_INPROC_SERVER,
        IID_IUnknown, (LPVOID*)&m_lpAggrInner);
    if (m_lpAggrInner == NULL)
        return FALSE;
    // optionally, create other aggregate objects here
    return TRUE;
}
```

```
BEGIN_INTERFACE_MAP(CAggrExample, CCmdTarget)
    // native "INTERFACE_PART" entries go here
    INTERFACE_AGGREGATE(CAggrExample, m_lpAggrInner)
END_INTERFACE_MAP()
```

The `m_lpAggrInner` is initialized in the constructor to NULL. The framework will ignore a NULL member variable in the default implementation of **QueryInterface**. **OnCreateAggregates** is a good place to actually create your aggregate objects. You'll have to call it explicitly if you are creating the object outside of the MFC implementation of **COleObjectFactory**. The reason for creating aggregates in **CCmdTarget::OnCreateAggregates** as well as the usage of **CCmdTarget::GetControllingUnknown** will become apparent when creating aggregatable objects is discussed.

This technique will give your object all of the interfaces that the aggregate object supports plus its native interfaces. If you only want a subset of the interfaces that the aggregate supports, you can override **CCmdTarget::GetInterfaceHook**. This allows you very low-level hookability, similar to **QueryInterface**. Usually, you want all the interfaces that the aggregate supports.

Making an Object Implementation Aggregatable

For an object to be aggregatable, the implementation of **AddRef**, **Release**, and **QueryInterface** must delegate to a "controlling unknown." In other words, for it to be part of the object, it must delegate **AddRef**, **Release**, and **QueryInterface** to a different object, also derived from **IUnknown**. This "controlling unknown" is provided to the object when it is created, that is, it is provided to the implementation of **COleObjectFactory**. Implementing this carries a small amount of overhead, and in some cases is not desirable, so MFC makes this optional. To enable an object to be aggregatable, you call **CCmdTarget::EnableAggregation** from the object's constructor.

If the object also uses aggregates, you must also be sure to pass the correct "controlling unknown" to the aggregate objects. Usually this **IUnknown** pointer is passed to the object when the aggregate is created. For example, the pUnkOuter parameter is the "controlling unknown" for objects created with **CoCreateInstance**. The correct "controlling unknown" pointer can be retrieved by calling **CCmdTarget::GetControllingUnknown**. The value returned from that function, however, is not valid during the constructor. For this reason, it is suggested that you create your aggregates only in an override of **CCmdTarget::OnCreateAggregates**, where the return value from **GetControllingUnknown** is reliable, even if created from the **COleObjectFactory** implementation.

It is also important that the object manipulate the correct reference count when adding or releasing artificial reference counts. To ensure this is the case, always call **ExternalAddRef** and **ExternalRelease** instead of **InternalRelease** and **InternalAddRef**. It is rare to call **InternalRelease** or **InternalAddRef** on a class that supports aggregation.

Reference Material

Advanced usage of OLE, such as defining your own interfaces or overriding the framework's implementation of the OLE interfaces requires the use of the underlying interface map mechanism.

This section discusses each macro and the APIs which is used to implement these advanced features.

CCmdTarget::EnableAggregation — Function Description

```
void EnableAggregation();
```

Remarks

Call this function in the constructor of the derived class if you wish to support OLE aggregation for objects of this type. This prepares a special IUnknown implementation that is required for aggregatable objects.

CCmdTarget::ExternalQueryInterface — Function Description

```
DWORD ExternalQueryInterface(
   const void FAR* lpIID,
   LPVOID FAR* ppvObj
);
```

Remarks

Parameters

lpIID

A far pointer to an IID (the first argument to QueryInterface)

ppvObj

A pointer to an IUnknown* (second argument to QueryInterface)

Remarks

Call this function in your implementation of IUnknown for each interface your class implements. This function provides the standard data-driven implementation of QueryInterface based on your object's interface map. It is necessary to cast the return value to an HRESULT. If the object is aggregated, this function will call the "controlling IUnknown" instead of using the local interface map.

CCmdTarget::ExternalAddRef — Function Description

```
DWORD ExternalAddRef();
```

Remarks

Call this function in your implementation of IUnknown::AddRef for each interface your class implements. The return value is the new reference count on the CCmdTarget object. If the object is aggregated, this function will call the "controlling IUnknown" instead of manipulating the local reference count.

CCmdTarget::ExternalRelease — Function Description

```
DWORD ExternalRelease();
```

Remarks

Call this function in your implementation of IUnknown::Release for each interface your class implements. The return value indicates the new reference count on the object. If the object is aggregated, this function will call the "controlling IUnknown" instead of manipulating the local reference count.

DECLARE_INTERFACE_MAP — Macro Description

```
DECLARE_INTERFACE_MAP
```

Remarks

Use this macro in any class derived from **CCmdTarget** that will have an interface map. Used in much the same way as **DECLARE_MESSAGE_MAP**. This macro invocation should be placed in the class definition, usually in a header (.H) file. A class with **DECLARE_INTERFACE_MAP** must define the interface map in the implementation file (.CPP) with the **BEGIN_INTERFACE_MAP** and **END_INTERFACE_MAP** macros.

BEGIN_INTERFACE_PART and END_INTERFACE_PART — Macro Descriptions

```
BEGIN_INTERFACE_PART(
   localClass,
   iface
);
END_INTERFACE_PART(
   localClass
)
```

Remarks

Parameters

localClass

The name of the class that implements the interface

iface

The name of the interface that this class implements

Remarks

For each interface that your class will implement, you need to have a **BEGIN_INTERFACE_PART** and **END_INTERFACE_PART** pair. These macros define a local class derived from the OLE interface that you define as well as an embedded member variable of that class. The **AddRef**, **Release**, and **QueryInterface** members are declared automatically. You must include the declarations for the other member functions that are part of the interface being implemented (those declarations are placed between the **BEGIN_INTERFACE_PART** and **END_INTERFACE_PART** macros).

The *iface* argument is the OLE interface that you wish to implement, such as **IAdviseSink**, or **IPersistStorage** (or your own custom interface).

The *localClass* argument is the name of the local class that will be defined. An 'X' will automatically be prepended to the name. This naming convention is used to avoid collisions with global classes of the same name. In addition, the name of the embedded member, the same as the *localClass* name except it is prefixed by 'm_x'.

For example:

```
BEGIN_INTERFACE_PART(MyAdviseSink, IAdviseSink)
   STDMETHOD_(void,OnDataChange)(LPFORMATETC, LPSTGMEDIUM);
   STDMETHOD_(void,OnViewChange)(DWORD, LONG);
   STDMETHOD_(void,OnRename)(LPMONIKER);
   STDMETHOD_(void,OnSave)();
   STDMETHOD_(void,OnClose)();
END_INTERFACE_PART(MyAdviseSink)
```

would define a local class called XMyAdviseSink derived from IAdviseSink, and a member of the class in which it is declared called m_xMyAdviseSink.Note:

Note

The lines beginning with **STDMETHOD_** are essentially copied from OLE2.H and modified slightly. Copying them from OLE2.H can reduce errors that are hard to resolve.

BEGIN_INTERFACE_MAP and END_INTERFACE_MAP — Macro Descriptions

```
            BEGIN_INTERFACE_MAP(
   theClass,
   baseClass
)
END_INTERFACE_MAP
```

Remarks

Parameters

theClass

The class in which the interface map is to be defined

baseClass

The class from which *theClass* derives from.

Remarks

The **BEGIN_INTERFACE_MAP** and **END_INTERFACE_MAP** macros are used in the implementation file to actually define the interface map. For each interface that is implemented there is one or more **INTERFACE_PART** macro invocations. For each aggregate that the class uses, there is one **INTERFACE_AGGREGATE** macro invocation.

INTERFACE_PART — Macro Description

```
            INTERFACE_PART(
   theClass,
   iid,
   localClass
)
```

Remarks

Parameters

theClass

The name of the class that contains the interface map.

iid

The **IID** that is to be mapped to the embedded class.

localClass

The name of the local class (less the 'X').

Remarks

This macro is used between the **BEGIN_INTERFACE_MAP** macro and the **END_INTERFACE_MAP** macro for each interface your object will support. It allows you to map an IID to a member of the class indicated by *theClass* and *localClass*. The 'm_x' will be added to the *localClass* automatically. Note that more than one **IID** may be associated with a single member. This is very useful when you are implementing only a "most derived" interface and wish to provide all intermediate interfaces as well. A good example of this is the **IOleInPlaceFrameWindow** interface. Its hierarchy looks like this:

```
IUnknown
    IOleWindow
        IOleUIWindow
            IOleInPlaceFrameWindow
```

If an object implements **IOleInPlaceFrameWindow**, a client may **QueryInterface** on any of these interfaces: **IOleUIWindow**, **IOleWindow**, or **IUnknown**, besides the "most derived" interface **IOleInPlaceFrameWindow** (the one you are actually implementing). To handle this you can use more than one **INTERFACE_PART** macro to map each and every base interface to the **IOleInPlaceFrameWindow** interface:

in the class definition file:

```
BEGIN_INTERFACE_PART(CMyFrameWindow, IOleInPlaceFrameWindow)
```

in the class implementation file:

```
BEGIN_INTERFACE_MAP(CMyWnd, CFrameWnd)
    INTERFACE_PART(CMyWnd, IID_IOleWindow, MyFrameWindow)
    INTERFACE_PART(CMyWnd, IID_IOleUIWindow, MyFrameWindow)
    INTERFACE_PART(CMyWnd, IID_IOleInPlaceFrameWindow, MyFrameWindow)
END_INTERFACE_MAP
```

The framework takes care of IUnknown because it is always required.

INTERFACE_PART — Macro Description

```
            INTERFACE_AGGREGATE(
   theClass,
   theAggr
)
```

Remarks

Parameters

theClass

The name of the class that contains the interface map,

theAggr

The name of the member variable that is to be aggregated.

Remarks

This macro is used to tell the framework that the class is using an aggregate object. It must appear between the

BEGIN_INTERFACE_PART and **END_INTERFACE_PART** macros. An aggregate object is a separate object, derived from **IUnknown**. By using an aggregate and the **INTERFACE_AGGREGATE** macro, you can make all the interfaces that the aggregate supports appear to be directly supported by the object. The *theAggr* argument is simply the name of a member variable of your class which is derived from **IUnknown** (either directly or indirectly). All **INTERFACE_AGGREGATE** macros must follow the **INTERFACE_PART** macros when placed in an interface map.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN039: MFC/OLE Automation Implementation

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

Overview of OLE IDispatch Interface

The **IDispatch** interface is the means by which applications expose methods and properties such that other applications such as Visual BASIC, or other languages, can make use of the application's features. The most important part of this interface is the **IDispatch::Invoke** function. MFC uses "dispatch maps" to implement **IDispatch::Invoke**. The dispatch map provides the MFC implementation information on the layout or "shape" of your **CCmdTarget**-derived classes, such that it can directly manipulate the properties of the object, or call member functions within your object to satisfy **IDispatch::Invoke** requests.

For the most part, ClassWizard and MFC cooperate to hide most of the details of OLE automation from the application programmer. The programmer concentrates on the actual functionality to expose in the application and doesn't have to worry about the underlying plumbing.

There are cases, however, where it is necessary to understand what MFC is doing behind the scenes. This note will address how the framework assigns **DISPID**s to member functions and properties. Knowledge of the algorithm MFC uses for assigning **DISPID**s is only necessary when you need to know the IDs, such as when you create a "type library" for your application's objects.

MFC DISPID assignment

Although the end-user of automation (a Visual Basic user, for example), sees the actual names of the automation enabled properties and methods in their code (such as obj.ShowWindow), the implementation of **IDispatch::Invoke** does not receive the actual names. For optimization reasons, it receives a **DISPID**, which is a 32-bit "magic cookie" that describes the method or property that is to be accessed. These **DISPID** values are returned from the **IDispatch** implementation through another method, called **IDispatch::GetIDsOfNames**. An automation client application will call **GetIDsOfNames** once for each member or property it intends to access, and cache them for later calls to **IDispatch::Invoke**. This way, the expensive string lookup is only done once per object use, instead of once per **IDispatch::Invoke** call.

MFC determines the **DISPID**s for each method and property based on two things:

- The distance from the top of the dispatch map (1 relative)
- The distance of the dispatch map from the most derived class (0 relative)

The **DISPID** is divided into two parts. The **LOWORD** of the **DISPID** contains the first component, the distance from the top of the dispatch map. The **HIWORD** contains the distance from the most derived class. For example:

```
class CDispPoint : public CCmdTarget
{
public:
    short m_x, m_y;
    ...
    DECLARE_DISPATCH_MAP()
    ...
};

class CDisp3DPoint : public CDispPoint
{
public:
    short m_z;
    ...
    DECLARE_DISPATCH_MAP()
    ...
};
```

```
BEGIN_DISPATCH_MAP(CDispPoint, CCmdTarget)
    DISP_PROPERTY(CDispPoint, "x", m_x, VT_I2)
    DISP_PROPERTY(CDispPoint, "y", m_y, VT_I2)
END_DISPATCH_MAP()

BEGIN_DISPATCH_MAP(CDisp3DPoint, CDispPoint)
    DISP_PROPERTY(CDisp3DPoint, "z", m_z, VT_I2)
END_DISPATCH_MAP()
```

As you can see, there are two classes, both of which expose OLE automation interfaces. One of these classes is derived from the other and thus leverages the base class's functionality, including the OLE automation part ("x" and "y" properties in this case).

MFC will generate **DISPID**s for class CDispPoint as follows:

```
property X    (DISPID)0x00000001
property Y    (DISPID)0x00000002
```

Since the properties are not in a base class, the **HIWORD** of the **DISPID** is always zero (the distance from the most derived class for CDispPoint is zero).

MFC will generate **DISPID**s for class CDisp3DPoint as follows:

```
property Z    (DISPID)0x00000001
property X    (DISPID)0x00010001
property Y    (DISPID)0x00010002
```

The Z property is given a **DISPID** with a zero **HIWORD** since it is defined in the class that is exposing the properties, CDisp3DPoint. Since the X and Y properties are defined in a base class, the **HIWORD** of the **DISPID** is 1, since the class in which these properties are defined is at a distance of one derivation from the most derived class.

Note

The **LOWORD** is always determined by the position in the map, even if there exist entries in the map with explicit **DISPID** (se e next section for information on the **_ID** versions of the **DISP_PROPERTY** and **DISP_FUNCTION** macros).

Advanced MFC Dispatch Map Features

There are a number of additional features that ClassWizard does not support with this release of Visual C++. ClassWizard supports **DISP_FUNCTION**, **DISP_PROPERTY**, and **DISP_PROPERTY_EX** which define a method, member variable property, and get/set member function property, respectively. These capabilities are usually all that is needed to create most automation servers.

The following additional macros can be used when the ClassWizard supported macros are not adequate: **DISP_PROPERTY_NOTIFY**, and **DISP_PROPERTY_PARAM**.

DISP_PROPERTY_NOTIFY — Macro Description

```
        DISP_PROPERTY_NOTIFY(
    theClass,
    pszName,
    memberName,
    pfnAfterSet,
    vtPropType
)
```

Remarks

Parameters

theClass

Name of the class.

pszName

External name of the property.

External name of the property.

memberName

Name of the member variable in which the property is stored.

pfnAfterSet

Name of member function to call when property is changed.

vtPropType

A value specifying the property's type.

Remarks

This macro is much like **DISP_PROPERTY**, except that it accepts an additional argument. The additional argument, *pfnAfterSet,* should be a member function that returns nothing and takes no parameters, 'void OnPropertyNotify()'. It will be called after the member variable has been modified.

DISP_PROPERTY_PARAM — Macro Description

```
        DISP_PROPERTY_PARAM(
    theClass,
    pszName,
    pfnGet,
    pfnSet,
    vtPropType,
    vtsParams
  )
```

Remarks

Parameters

theClass

Name of the class.

pszName

External name of the property.

memberGet

Name of the member function used to get the property.

memberSet

Name of the member function used to set the property.

vtPropType

A value specifying the property's type.

vtsParams

A string of space separated VTS_ for each parameter.

Remarks

Much like the **DISP_PROPERTY_EX** macro, this macro defines a property accessed with separate Get and Set member functions. This macro, however, allows you to specify a parameter list for the property. This is useful for implementing properties that are indexed or parameterized in some other way. The parameters will always be placed first, followed by the new value for the property. For example:

```
DISP_PROPERTY_PARAM(CMyObject, "item", GetItem, SetItem, VT_DISPATCH,    VTS_I2 VTS_I2)
```

would correspond to get and set member functions:

```
LPDISPATCH CMyObject::GetItem(short row, short col)
void CMyObject::SetItem(short row, short col, LPDISPATCH newValue)
```

DISP_XXXX_ID — Macro Descriptions

```
        DISP_FUNCTION_ID(
    theClass,
    pszName,
    dispid,
    pfnMember,
    vtRetVal,
    vtsParams
  )
  DISP_PROPERTY_ID(
    theClass,
    pszName,
    dispid,
    memberName,
    vtPropType
  )
  DISP_PROPERTY_NOTIFY_ID(
    theClass,
    pszName,
    dispid,
    memberName,
    pfnAfterSet,
    vtPropType
  )
  DISP_PROPERTY_EX_ID(
    theClass,
    pszName,
    dispid,
    pfnGet,
    pfnSet,
    vtPropType
  )
  DISP_PROPERTY_PARAM_ID(
    theClass,
    pszName,
    dispid,
    pfnGet,
    pfnSet,
    vtPropType,
    vtsParams
  )
```

Remarks

Parameters

theClass

Name of the class.

pszName

External name of the property.

dispid

The fixed DISPID for the property or method.

pfnGet

Name of the member function used to get the property.

pfnSet

Name of the member function used to set the property.

memberName

The name of the member variable to map to the property

vtPropType

A value specifying the property's type.

vtsParams

A string of space separated VTS_ for each parameter.

Remarks

These macros allow you to specify a **DISPID** instead of letting MFC automatically assign one. These advanced macros have the same names except that ID is appended to the macro name (e.g. **DISP_PROPERTY_ID**) and the ID is determined by the parameter specified just after the *pszName* parameter. See AFXDISP.H for more information on these macros. The **_ID** entries must be placed at the end of the dispatch map. They will affect the automatic **DISPID** generation in the same way as a non-**_ID** version of the macro would (the **DISPID**s are determined by position). For example:

```
BEGIN_DISPATCH_MAP(CDisp3DPoint, CCmdTarget)
    DISP_PROPERTY(CDisp3DPoint, "y", m_y, VT_I2)
    DISP_PROPERTY(CDisp3DPoint, "z", m_z, VT_I2)
    DISP_PROPERTY_ID(CDisp3DPoint, "x", 0x00020003, m_x, VT_I2)
END_DISPATCH_MAP()
```

MFC will generate DISPIDs for class CDisp3DPoint as follows:

```
property X    (DISPID)0x00020003
property Y    (DISPID)0x00000002
property Z     (DISPID)0x00000001
```

Specifying a fixed **DISPID** is useful to maintain backward compatibility to a previously existing dispatch interface, or to implement certain system defined methods or properties (usually indicated by a negative **DISPID**, such as the **DISPID_NEWENUM** collection).

Retrieving the IDispatch Interface for a COleClientItem

Many servers will support automation within their document objects, along with the OLE server functionality. In order to gain access to this automation interface, it is necessary to directly access the **COleClientItem::m_lpObject** member variable. The code below will retrieve the **IDispatch** interface for an object derived from **COleClientItem**. You can include the code below in your application if you find this functionality necessary:

```
LPDISPATCH CMyClientItem::GetIDispatch()
{
    ASSERT_VALID(this);
    ASSERT(m_lpObject != NULL);

    LPUNKNOWN lpUnk = m_lpObject;

    Run();    // must be running

    LPOLELINK lpOleLink = NULL;
    if (m_lpObject->QueryInterface(IID_IOleLink,
        (LPVOID FAR*)&lpOleLink) == NOERROR)
    {
        ASSERT(lpOleLink != NULL);
        lpUnk = NULL;
        if (lpOleLink->GetBoundSource(&lpUnk) != NOERROR)
        {
            TRACE0("Warning: Link is not connected!\n");
            lpOleLink->Release();
            return NULL;
        }
        ASSERT(lpUnk != NULL);
    }

    LPDISPATCH lpDispatch = NULL;
    if (lpUnk->QueryInterface(IID_IDispatch, &lpDispatch)
        != NOERROR)
```

```
    {
        TRACE0("Warning: does not support IDispatch!\n");
        return NULL;
    }

    ASSERT(lpDispatch != NULL);
    return lpDispatch;
}
```

The dispatch interface returned from this function could then be used directly or attached to a **COleDispatchDriver** for type-safe access. If you use it directly, make sure that you call its **Release** member when through with the pointer (the **COleDispatchDriver** destructor does this by default).

See Also

Other Resources

Technical Notes by Number
Technical Notes by Category

TN040: MFC/OLE In-Place Resizing and Zooming

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note will discuss the issues relating to in-place editing and how a server should accomplish correct zooming and in-place resizing. With in-place activation, the WYSIWYG concept is taken one step further in that containers and servers cooperate with each other, and in particular interpret the OLE specification in much the same way.

Because of the close interaction between a container and server supporting in-place activation there are a number of expectations from the end-user that should be maintained:

- The presentation display (the metafile drawn in the **COleServerItem::OnDraw** override) should look exactly the same as when it is drawn for editing (except that editing tools are not visible).
- When the container zooms, the server window should too!
- Both the container and server should display objects for editing using the same metrics. This means using a mapping mode based on the number of *logical* pixels per inch — not physical pixels per inch, when rendering on the display device.

Note
Because in-place activation only applies to items that are embedded (not linked), zooming only applies to embedded objects. You will see APIs in both **COleServerDoc** and **COleServerItem** that are used for zooming. The reason for this dichotomy is that only functions that are valid for both linked and embedded items are in **COleServerItem** (this allows you to have a common implementation) and functions that are valid only for embedded objects are located in the **COleServerDoc** class (from the server's perspective, it is the *document* which is embedded).

Most of the burden is placed on the server implementer, in that the server must be aware of the container's zoom factor and modify its editing interface as appropriate. But how does the server determine the zoom factor that the container is using?

MFC Support for Zooming

The current zoom factor can be determined by calling **COleServerDoc::GetZoomFactor**. Calling this when the document is not in-place active will always result in a 100% zoom factor (or 1:1 ratio). Calling it while in-place active may return something other than 100%.

For an example of zooming correctly see the MFC OLE sample HIERSVR. Zooming in HIERSVR is complicated by the fact that it displays text, and text, in general, does not scale in a linear fashion (hints, typographic conventions, design widths, and heights all complicate the matter). Still, HIERSVR is a reasonable reference for implementing zooming correctly, and so is the MFC Tutorial SCRIBBLE (step 7).

COleServerDoc::GetZoomFactor determines the zoom factor based on a number of different metrics available either from the container or from the implementation of your **COleServerItem** and **COleServerDoc** classes. In short, the current zoom factor is determined by the following formula:

```
Position Rectangle (PR) / Container Extent (CE)
```

The POSITION RECTANGLE is determined by the container. It is returned to the server during in-place activation when **COleClientItem::OnGetItemPosition** is called and is updated when the container calls the server's **COleServerDoc::OnSetItemRects** (with a call to **COleClientItem::SetItemRects**).

The CONTAINER EXTENT is slightly more complex to calculate. If the container has called **COleServerItem::OnSetExtent** (with a call to **COleClientItem::SetExtent**), then the CONTAINER EXTENT is this value converted to pixels based on the number of pixels per logical inch. If the container has not called SetExtent (which is usually the case), then the CONTAINER EXTENT is the size returned from **COleServerItem::OnGetExtent**. So, if the container has not called SetExtent, the framework assumes that if

it did the container would have called it with 100% of the natural extent (the value returned from **COleServerItem::GetExtent**). Stated another way, the framework assumes that the container is displaying 100% (no more, no less) of the item.

It is important to note that although **COleServerItem::OnSetExtent** and **COleServerItem::OnGetExtent** have similar names, they do not manipulate the same attribute of the item. **OnSetExtent** is called to let the server know how much of the object is visible in the container (regardless of the zoom factor) and **OnGetExtent** is called by the container to determine ideal size of the object.

By looking at each of the APIs involved, you can get a clearer picture:

COleServerItem::OnGetExtent

This function should return the "natural size" in HIMETRIC units of the item. The best way to think of the "natural size" is to define it as the size it might appear when printed. The size returned here is constant for a particular item contents (much like the metafile, which is constant for a particular item). This size does not change when zooming is applied to the item. It usually does not change when the container gives the item more or less space by calling **OnSetExtent**. An example of a change might be that of a simple text editor with no "margin" capability that wrapped text based on the last extent sent by the container. If a server does change, the server should probably set the OLEMISC_RECOMPOSEONRESIZE bit in the system registry (see the OLE SDK documentation for more information on this option).

COleServerItem::OnSetExtent

This function is called when the container shows "more or less" of the object. Most containers will not call this at all. The default implementation stores the last value received from the container in 'm_sizeExtent', which is used in **COleServerDoc::GetZoomFactor** when computing the CONTAINER EXTENT value described above.

COleServerDoc::OnSetItemRects

This function is called only when the document is in-place active. It is called when the container updates either the item's position or the clipping applied to the item. The POSITION RECTANGLE, as discussed above, provides the numerator for the zoom factor calculation. A server can request that the item position be changed by calling **COleServerDoc::RequestPositionChange**. The container may or may not respond to this request by calling **OnSetItemRects** (with a call to **COleServerItem::SetItemRects**).

COleServerDoc::OnDraw

It is important to realize that the metafile created by overriding of **COleServerItem::OnDraw** produces exactly the same metafile, regardless of the current zoom factor. The container will scale the metafile as appropriate. This is an important distinction between the view's **OnDraw** and the server item's **OnDraw**. The view handles zooming, the item just creates a *zoomable* metafile and leaves it up to the container to do the appropriate zooming.

The best way to insure that your server behaves correctly is to use the implementation of **COleServerDoc::GetZoomFactor** if your document is in-place active.

MFC Support for In-Place Resizing

MFC fully implements the in-place resizing interface as described in the OLE 2 specification. The user-interface is supported by the **COleResizeBar** class, a custom message **WM_SIZECHILD**, and special handling of this message in **COleIPFrameWnd**.

You may want to implement different handling of this message than what is provided by the framework. As described above, the framework leaves the results of in-place resizing up to the container — the server responds to the change in the zoom factor. If the container reacts by setting the both CONTAINER EXTENT and POSITION RECTANGLE during the processing of its **COleClientItem::OnChangeItemPosition** (called as a result of a call to **COleServerDoc::RequestPositionChange**) then the in-place resize will result in showing "more or less" of the item in the editing window. If the container reacts by just setting the POSITION RECTANGLE during the processing of **COleClientItem::OnChangeItemPosition**, the zoom factor will change and the item will be shown "zoomed in or out."

A server can control (to some degree) what happens during this negotiation. A spreadsheet, for example might elect to show more or fewer cells when the user resizes the window while editing the item in-place. A word-processor might elect to change the "page margins" so they are the same as the window and rewrap the text to the new margin. Servers implement this by changing the natural extent (the size returned from **COleServerItem::OnGetExtent**) when the resizing is done. This will cause both the POSITION RECTANGLE and the CONTAINER EXTENT to change by the same amount, resulting in the same zoom factor, but a bigger or smaller viewing area. In addition, more or less of the document will be visible in the metafile generated by **OnDraw**. In this case, the document itself is changing when the user resizes the item, instead of just the viewing area.

You can implement custom resizing and still leverage the user interface provided by **COleResizeBar** by overriding the **WM_SIZECHILD** message in your **COleIPFrameWnd** class. For more information on the specifics of **WM_SIZECHILD**, see Technical Note 24.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN041: MFC/OLE1 Migration to MFC/OLE 2

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

General Issues Relating to Migration

One of the design goals for the OLE 2 classes in MFC 2.5 (and higher) was to retain much of the same architecture put in place in MFC 2.0 for OLE 1.0 support. As a result, many of the same OLE classes in MFC 2.0 still exist in this version of MFC (**COleDocument**, **COleServerDoc**, **COleClientItem**, **COleServerItem**). In addition, many of the APIs in these classes are exactly the same. However, OLE 2 is drastically different from OLE 1.0 so you can expect that some of the details have changed. If you are familiar with MFC 2.0's OLE1 support, you'll feel at home with MFC's 2.0 support.

If you are taking an existing MFC/OLE1 application and adding OLE 2 functionality to it, you should read this note first. This note covers some general issues you may encounter while porting your OLE1 functionality to MFC/OLE 2 and then discusses the problems uncovered while porting two applications included in MFC 2.0: the MFC OLE samples OCLIENT and HIERSVR.

MFC Document/View Architecture Is Important

If your application does not use MFC's Document/View architecture and you want to add OLE 2 support to your application, now is the time to move to Document/View. Many of the benefits of MFC's OLE 2 classes are only realized once your application is using the built-in architecture and components of MFC.

Implementing a server or container without using the MFC architecture is possible, but not recommended.

Use MFC Implementation Instead of Your Own

MFC "canned implementation" classes such as **CToolBar**, **CStatusBar**, and **CScrollView** have built-in special case code for OLE 2 support. So, if you can use these classes in your application you'll benefit from the effort put into them to make them OLE aware. Again, it is possible to "roll-your-own" classes here for these purposes, but it is not suggested. If you need to implement similar functionality, the MFC source code is an excellent reference for dealing with some of the finer points of OLE (especially when it comes to in-place activation).

Examine the MFC Sample Code

There are a number of MFC samples that include OLE functionality. Each of these applications implements OLE from a different angle:

- HIERSVR Meant mostly for use as a server application. It was included in MFC 2.0 as an MFC/OLE1 application and has been ported to MFC/OLE 2 and then extended such that it implements many OLE features available in OLE 2.
- OCLIENT This is a stand-alone container application, meant to demonstrate many of the OLE features from a container standpoint. It too was ported from MFC 2.0, and then extended to support many of the more advanced OLE features, such as custom clipboard formats and links to embedded items.
- DRAWCLI This application implements OLE container support much like OCLIENT does, except that it does so within the framework of an existing object-oriented drawing program. It shows you how you might implement OLE container support and integrate it into your existing application.
- SUPERPAD This application, as well as being a fine stand-alone application, is also an OLE server. The server support it implements is quite minimalist. Of particular interest is how it uses OLE clipboard services to copy data to the clipboard, but uses the functionality built into the Windows "edit" control to implement clipboard paste functionality. This shows an interesting mix of traditional Windows API usage as well as integration with the new OLE APIs.

For more information on the sample applications, see the "MFC Sample Help".

Case Study: OCLIENT from MFC 2.0

As discussed above, OCLIENT was included in MFC 2.0 and implemented OLE with MFC/OLE1. The steps by which this application was initially converted to use the MFC/OLE 2 classes are described below. A number of features were added after the initial port was completed to better illustrate the MFC/OLE classes. These features will not be covered here; refer to the

sample itself for more information on those advanced features.

Note
The compiler errors and step-by-step process was created with Visual C++ 2.0. Specific error messages and locations may have changed with Visual C++ 4.0, but the conceptual information remains valid.

Getting It Up and Running

The approach taken to port the OCLIENT sample to MFC/OLE is to start by building it and fixing the obvious compiler errors that will result. If you take the OCLIENT sample from MFC 2.0 and compile it under this version of MFC, you'll find that there are not that many errors to resolve. The errors in the order in which they occurred are described below.

Compile and Fix Errors

```
\oclient\mainview.cpp(104) : error C2660: 'Draw' : function does not take 4 parameters
```

The first error concerns **COleClientItem::Draw**. In MFC/OLE1 it took more parameters than the MFC/OLE version takes. The extra parameters were often not necessary and usually NULL (as in this example). This version of MFC can automatically determine the values for the lpWBounds when the CDC that is being drawn to is a metafile DC. In addition, the pFormatDC parameter is no longer necessary since the framework will build one from the "attribute DC" of the pDC passed in. So to fix this problem, you simply remove the two extra NULL parameters to the Draw call.

```
\oclient\mainview.cpp(273) : error C2065: 'OLE_MAXNAMESIZE' : undeclared identifier
\oclient\mainview.cpp(273) : error C2057: expected constant expression
\oclient\mainview.cpp(280) : error C2664: 'CreateLinkFromClipboard' : cannot convert parame
ter 1 from 'char [1]' to 'enum ::tagOLERENDER '
\oclient\mainview.cpp(286) : error C2664: 'CreateFromClipboard' : cannot convert parameter
1 from 'char [1]' to 'enum ::tagOLERENDER '
\oclient\mainview.cpp(288) : error C2664: 'CreateStaticFromClipboard' : cannot convert para
meter 1 from 'char [1]' to 'enum ::tagOLERENDER '
```

The errors above result from the fact that all of the **COleClientItem::CreateXXXX** functions in MFC/OLE1 required that a unique name be passed to represent the item. This was a requirement of the underlying OLE API. This is not necessary in MFC/OLE 2 since OLE 2 does not use DDE as the underlying communications mechanism (the name was used in DDE conversations). To fix this problem, you can remove the **CreateNewName** function as well as all references to it. It is easy to find out what each MFC/OLE function is expecting in this version simply by placing your cursor on the call and pressing F1.

Another area that is significantly different is OLE 2 clipboard handling. With OLE1, you used the Windows clipboard APIs interact with the clipboard. With OLE 2 this is done with a different mechanism. The MFC/OLE1 APIs assumed that the clipboard was open before copying a **COleClientItem** object to the clipboard. This is no longer necessary and will cause all MFC/OLE clipboard operations to fail. While you edit the code to remove dependencies on **CreateNewName**, you should also remove the code that opens and closes the Windows clipboard.

```
\oclient\mainview.cpp(332) : error C2065: 'AfxOleInsertDialog' : undeclared identifier
\oclient\mainview.cpp(332) : error C2064: term does not evaluate to a function
\oclient\mainview.cpp(344) : error C2057: expected constant expression
\oclient\mainview.cpp(347) : error C2039: 'CreateNewObject' : is not a member of 'CRectItem
'
```

These errors result from the **CMainView::OnInsertObject** handler. Handling the "Insert New Object" command is another area where things have changed quite a bit. In this case, it is easiest to simply merge the original implementation with that provided by AppWizard for a new OLE Container application. In fact, this is a technique that you can apply to porting other applications. In MFC/OLE1, you displayed the "Insert Object" dialog by calling **AfxOleInsertDialog** function. In this version you construct a **COleInsertObject** dialog object and call **DoModal**. In addition, new OLE items are created with a **CLSID** instead of a classname string. The end result should look something like this

```
COleInsertDialog dlg;
if (dlg.DoModal() != IDOK)
    return;

BeginWaitCursor();

CRectItem* pItem = NULL;
```

```
TRY
{
    // First create the C++ object
    pItem = GetDocument()->CreateItem();
    ASSERT_VALID(pItem);

    // Initialize the item from the dialog data.
    if (!dlg.CreateItem(pItem))
        AfxThrowMemoryException();
           // any exception will do
    ASSERT_VALID(pItem);

    // run the object if appropriate
    if (dlg.GetSelectionType() ==
            COleInsertDialog::createNewItem)
        pItem->DoVerb(OLEIVERB_SHOW, this);

    // update right away
    pItem->UpdateLink();
    pItem->UpdateItemRectFromServer();

    // set selection to newly inserted item
    SetSelection(pItem);
    pItem->Invalidate();
}
CATCH (CException, e)
{
    // clean up item
    if (pItem != NULL)
        GetDocument()->DeleteItem(pItem);

    AfxMessageBox(IDP_FAILED_TO_CREATE);
}
END_CATCH

EndWaitCursor();
```

Note

Insert New Object may be different for your application):

It is also necessary to include <afxodlgs.h>, which contains the declaration for the **COleInsertObject** dialog class as well as the other standard dialogs provided by MFC.

```
\oclient\mainview.cpp(367) : error C2065: 'OLEVERB_PRIMARY' : undeclared identifier
\oclient\mainview.cpp(367) : error C2660: 'DoVerb' : function does not take 1 parameters
```

These errors are caused by the fact that some OLE1 constants have changed in OLE 2, even though in concept they are the same. In this case **OLEVERB_PRIMARY** has changed to **OLEIVERB_PRIMARY**. In both OLE1 and OLE 2, the primary verb is usually executed by a container when the user double-clicks on an item.

In addition, **DoVerb** now takes an extra parameter — a pointer to a view (**CView***). This parameter is only used to implement "Visual Editing" (or in-place activation). For now you set that parameter to NULL, because you are not implementing this feature at this time.

To make sure that the framework never attempts to in-place activate, you should override **COleClientItem::CanActivate** as follows:

```
BOOL CRectItem::CanActivate()
{
    return FALSE;
}

\oclient\rectitem.cpp(53) : error C2065: 'GetBounds' : undeclared identifier
\oclient\rectitem.cpp(53) : error C2064: term does not evaluate to a function
```

```
\oclient\rectitem.cpp(84) : error C2065: 'SetBounds' : undeclared identifier
\oclient\rectitem.cpp(84) : error C2064: term does not evaluate to a function
```

In MFC/OLE1, **COleClientItem::GetBounds** and **SetBounds** were used to query and manipulate the extent of an item (the **left** and **top** members were always zero). In MFC/OLE 2 this is more directly supported by **COleClientItem::GetExtent** and **SetExtent**, which deal with a **SIZE** or **CSize** instead.

The code for your new SetItemRectToServer, and UpdateItemRectFromServer calls look like this:

```
BOOL CRectItem::UpdateItemRectFromServer()
{
   ASSERT(m_bTrackServerSize);
   CSize size;
   if (!GetExtent(&size))
      return FALSE;    // blank

   // map from HIMETRIC to screen coordinates
   {
      CClientDC screenDC(NULL);
      screenDC.SetMapMode(MM_HIMETRIC);
      screenDC.LPtoDP(&size);
   }
   // just set the item size
   if (m_rect.Size() != size)
   {
      // invalidate the old size/position
      Invalidate();
      m_rect.right = m_rect.left + size.cx;
      m_rect.bottom = m_rect.top + size.cy;
      // as well as the new size/position
      Invalidate();
   }
   return TRUE;
}

BOOL CRectItem::SetItemRectToServer()
{
   // set the official bounds for the embedded item
   CSize size = m_rect.Size();
   {
      CClientDC screenDC(NULL);
      screenDC.SetMapMode(MM_HIMETRIC);
      screenDC.DPtoLP(&size);
   }
   TRY
   {
      SetExtent(size);  // may do a wait
   }
   CATCH(CException, e)
   {
      return FALSE;  // links will not allow SetBounds
   }
   END_CATCH
   return TRUE;
}

\oclient\frame.cpp(50) : error C2039: 'InWaitForRelease' : is not a member of 'COleClientIt
em'
\oclient\frame.cpp(50) : error C2065: 'InWaitForRelease' : undeclared identifier
\oclient\frame.cpp(50) : error C2064: term does not evaluate to a function
```

In MFC/OLE1 synchronous API calls from a container to a server were *simulated*, because OLE1 was inherently asynchronous in many cases. It was necessary to check for an outstanding asynchronous call in progress before processing commands from the user. MFC/OLE1 provided the **COleClientItem::InWaitForRelease** function for doing so. In MFC/OLE 2 this is not necessary, so you can to remove the override of OnCommand in CMainFrame all together.

At this point OCLIENT will compile and link.

Other Necessary Changes

There are few things that are not done that will keep OCLIENT from running, however. It is better to fix these problems now instead of later.

First, it is necessary to initialize the OLE libraries. This is done by calling **AfxOleInit** from **InitInstance**:

```
if (!AfxOleInit())
{
  AfxMessageBox("Failed to initialize OLE libraries");
  return FALSE;
}
```

It is also a good idea to check for virtual functions for parameter list changes. One such function is **COleClientItem::OnChange**, overridden in every MFC/OLE container application. By looking at online help, you'll see that an extra 'DWORD dwParam' was added. The new CRectItem::OnChange looks as follows:

```
void
CRectItem::OnChange(OLE_NOTIFICATION wNotification, DWORD dwParam)
{
  if (m_bTrackServerSize &&
        !UpdateItemRectFromServer())
  {
    // Blank object
    if (wNotification == OLE_CLOSED)
    {
      // no data received for the object - destroy it
      ASSERT(!IsVisible());
      GetDocument()->DeleteItem(this);
      return;   // no update (item is gone now)
    }
  }
  if (wNotification != OLE_CLOSED)
      Dirty();
  Invalidate();  // any change will cause a redraw
}
```

In MFC/OLE1, container applications derived the document class from **COleClientDoc**. In MFC/OLE 2 this class has been removed and replaced by **COleDocument** (this new organization makes it easier to build container/server applications). There is a **#define** that maps **COleClientDoc** to **COleDocument** to simplify porting of MFC/OLE1 applications to MFC/OLE 2, such as OCLIENT. One of the features not supplied by **COleDocument** that was provided by **COleClientDoc** is the standard command message map entries. This is done so that server applications, which also use **COleDocument** (indirectly), do not carry with them the overhead of these command handlers unless they are a container/server application. You need to add the following entries to the CMainDoc message map:

```
ON_UPDATE_COMMAND_UI(ID_EDIT_PASTE, OnUpdatePasteMenu)
ON_UPDATE_COMMAND_UI(ID_EDIT_PASTE_LINK, OnUpdatePasteLinkMenu)
ON_UPDATE_COMMAND_UI(ID_OLE_EDIT_LINKS, OnUpdateEditLinksMenu)
ON_COMMAND(ID_OLE_EDIT_LINKS, COleDocument::OnEditLinks)
ON_UPDATE_COMMAND_UI(ID_OLE_VERB_FIRST, OnUpdateObjectVerbMenu)
ON_UPDATE_COMMAND_UI(ID_OLE_EDIT_CONVERT, OnUpdateObjectVerbMenu)
ON_COMMAND(ID_OLE_EDIT_CONVERT, OnEditConvert)
```

The implementation of all of these commands is in **COleDocument**, which is the base class for your document.

At this point, OCLIENT is a functional OLE container application. It is possible to insert items of any type (OLE1 or OLE 2). Since the necessary code to enable in-place activation is not implemented, items are edited in a separate window much like with OLE1. The next section discusses the necessary changes to enable in-place editing (sometimes called "Visual Editing").

Adding "Visual Editing"

One of the most interesting features of OLE is in-place activation (or "Visual Editing"). This feature allows the server application to take over portions of the container's user interface to provided a more seamless editing interface for the user. To implement in-place activation to OCLIENT, some special resources need to be added, as well as some additional code. These resources and the code are normally provided by AppWizard — in fact, much of the code here was borrowed directly from a fresh AppWizard

application with "Container" support.

First of all, it is necessary to add a menu resource to be used when there is an item which is in-place active. You can create this extra menu resource in Visual C++ by copying the IDR_OCLITYPE resource and removing all but the File and Window pop-ups. Two separator bars are inserted between the File and Window pop-ups to indicate the separation of groups (it should look like: File | | Window). For more information on what these separators mean and how the server and container menus are merged see "Menus and Resources: Menu Merging" in *OLE 2 Classes*.

Once you have these menus created, you need to let the framework know about them. This is done by calling **CDocTemplate::SetContainerInfo** for the document template before you add it to the document template list in your InitInstance. The new code to register the document template looks like this:

```
CDocTemplate* pTemplate = new CMultiDocTemplate(
    IDR_OLECLITYPE,
    RUNTIME_CLASS(CMainDoc),
    RUNTIME_CLASS(CMDIChildWnd),    // standard MDI child frame
    RUNTIME_CLASS(CMainView));
pTemplate->SetContainerInfo(IDR_OLECLITYPE_INPLACE);
AddDocTemplate(pTemplate);
```

The IDR_OLECLITYPE_INPLACE resource is the special in-place resource created in Visual C++.

To enable in-place activation, there are some things that need to change in both the **CView** (CMainView) derived class as well as the **COleClientItem** derived class (CRectItem). All of these overrides are provided by AppWizard and most of the implementation will come directly from a default AppWizard application.

In the first step of this port, in-place activation was disabled entirely by overriding **COleClientItem::CanActivate**. This override should be removed to allow in-place activation. In addition, NULL was passed to all calls to **DoVerb** (there are two of them) because providing the view was only necessary for in-place activation. To fully implement in-place activation, it is necessary to pass the correct view in the **DoVerb** call. One of these calls is in **CMainView::OnInsertObject**:

```
pItem->DoVerb(OLEIVERB_SHOW, this);
```

Another is in **CMainView::OnLButtonDblClk**:

```
m_pSelection->DoVerb(OLEIVERB_PRIMARY, this);
```

It is necessary to override **COleClientItem::OnGetItemPosition**. This tells the server where to put its window relative to the container's window when the item is in-place activated. For OCLIENT, the implementation is trivial:

```
void CRectItem::OnGetItemPosition(CRect& rPosition)
{
    rPosition = m_rect;
}
```

Most servers also implement what is called "in-place resizing." This allows the server window to be sized and moved while the user is editing the item. The container must participate in this action, since moving or resizing the window usually affects the position and size within the container document itself. The implementation for OCLIENT synchronizes the internal rectangle maintained by m_rect with the new position and size.

```
BOOL CRectItem::OnChangeItemPosition(const CRect& rectPos)
{
    ASSERT_VALID(this);

    if (!COleClientItem::OnChangeItemPosition(rectPos))
        return FALSE;

    Invalidate();
    m_rect = rectPos;
    Invalidate();
    GetDocument()->SetModifiedFlag();

    return TRUE;
```

```
}
```

At this point, there is enough code to allow an item to be in-place activated and to deal with sizing and moving the item when it is active, but no code will allow the user to exit the editing session. Although some servers will provide this functionality themselves by handling the escape key, it is suggested that containers provide two ways to deactivate an item: (1) by clicking outside the item, and (2) by pressing the ESCAPE key.

For the ESCAPE key, add an accelerator with Visual C++ that maps the VK_ESCAPE key to a command, ID_CANCEL_EDIT is added to the resources. The handler for this command follows:

```
// The following command handler provides the standard
// keyboard user interface to cancel an in-place
// editing session.void CMainView::OnCancelEdit()
{
    // Close any in-place active item on this view.
    COleClientItem* pActiveItem =
        GetDocument()->GetInPlaceActiveItem(this);
    if (pActiveItem != NULL)
        pActiveItem->Close();
    ASSERT(GetDocument()->GetInPlaceActiveItem(this) == NULL);
}
```

To handle the case where the user clicks outside the item, you add the following code to the start of **CMainView::SetSelection**:

```
if (pNewSel != m_pSelection || pNewSel == NULL)
{
    COleClientItem* pActiveItem =
        GetDocument()->GetInPlaceActiveItem(this);
    if (pActiveItem != NULL && pActiveItem != pNewSel)
        pActiveItem->Close();
}
```

When an item is in-place active, it should have the focus. To make sure this is the case you handle OnSetFocus so that focus is always transferred to the active item when your view receives the focus:

```
// Special handling of OnSetFocus and OnSize are required
// when an object is being edited in-place.
void CMainView::OnSetFocus(CWnd* pOldWnd)
{
    COleClientItem* pActiveItem =
        GetDocument()->GetInPlaceActiveItem(this);
    if (pActiveItem != NULL &&
    pActiveItem->GetItemState() == COleClientItem::activeUIState)
    {
        // need to set focus to this item if it is same view
        CWnd* pWnd = pActiveItem->GetInPlaceWindow();
        if (pWnd != NULL)
        {
            pWnd->SetFocus();  // don't call the base class
            return;
        }
    }

    CView::OnSetFocus(pOldWnd);
}
```

When the view is resized, you need to notify the active item that the clipping rectangle has changed. To do this you provide a handler for **OnSize**:

```
void CMainView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
```

```
    COleClientItem* pActiveItem =
        GetDocument()->GetInPlaceActiveItem(this);
    if (pActiveItem != NULL)
        pActiveItem->SetItemRects();
}
```

Case Study: HIERSVR from MFC 2.0

HIERSVR was also included in MFC 2.0 and implemented OLE with MFC/OLE1. This note briefly describes the steps by which this application was initially converted to use the MFC/OLE 2 classes. A number of features were added after the initial port was completed to better illustrate the MFC/OLE 2 classes. These features will not be covered here; refer to the sample itself for more information on those advanced features.

Note
The compiler errors and step-by-step process was created with Visual C++ 2.0. Specific error messages and locations may have changed with Visual C++ 4.0, but the conceptual information remains valid.

Getting It Up and Running

The approach taken to port the HIERSVR sample to MFC/OLE is to start by building it and fixing the obvious compiler errors that will result. If you take the HIERSVR sample from MFC 2.0 and compile it under this version of MFC, you'll find that there are not many errors to resolve (although there are more than with the OCLIENT sample). The errors in the order in which they usually occur are described below.

Compile and Fix Errors

```
\hiersvr\hiersvr.cpp(83) : error C2039: 'RunEmbedded' : is not a member of 'COleTemplateSer
ver'
```

This first error points out a much larger problem with the **InitInstance** function for servers. The initialization required for an OLE server is probably one of the biggest changes you'll have to make to your MFC/OLE1 application to get it running. The best thing to do is look at what AppWizard creates for an OLE server and modify your code as appropriate. Here are some points to keep in mind:

It is necessary to initialize the OLE libraries by calling **AfxOleInit**

Call SetServerInfo on the document template object to set server resource handles and runtime class information that you can't set with the **CDocTemplate** constructor.

Don't show the main window of your application if /Embedding is present on the command line.

You'll need a **GUID** for your document. This is a unique identifier for your document's type (128 bits). AppWizard will create one for you — so if you use the technique described here of copying the new code from a new AppWizard generated server application, you can simply "steal" the GUID from that application. If not, you can use the GUIDGEN.EXE utility in the BIN directory.

It is necessary to "connect" your **COleTemplateServer** object to the document template by calling **COleTemplateServer::ConnectTemplate**.

Update the system registry when your application is run stand-alone. This way, if the user moves the .EXE for your application, running it from its new location will update the Windows system registration database to point to the new location.

After applying all of these changes based on what AppWizard creates for **InitInstance**, the **InitInstance** (and related GUID) for HIERSVR should read as follows:

```
// this is the GUID for HIERSVR documents
static const GUID BASED_CODE clsid =
    { 0xA0A16360L, 0xC19B, 0x101A, { 0x8C, 0xE5, 0x00, 0xDD, 0x01, 0x11, 0x3F, 0x12 } };

/////////////////////////////////////////////////////////////////////////////
// COLEServerApp initialization

BOOL COLEServerApp::InitInstance()
{
    // OLE 2 initialization
    if (!AfxOleInit())
```

```
    {
        AfxMessageBox("Initialization of the OLE failed!");
        return FALSE;
    }

    // Standard initialization
    LoadStdProfileSettings(); // Load standard INI file options

    // Register document templates
    CDocTemplate* pDocTemplate;
    pDocTemplate = new CMultiDocTemplate(IDR_HIERSVRTYPE,
        RUNTIME_CLASS(CServerDoc),
        RUNTIME_CLASS(CMDIChildWnd),
        RUNTIME_CLASS(CServerView));
    pDocTemplate->SetServerInfo(IDR_HIERSVRTYPE_SRVR_EMB);
    AddDocTemplate(pDocTemplate);

    // create main MDI Frame window
    CMainFrame* pMainFrame = new CMainFrame;
    if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
        return FALSE;
    m_pMainWnd = pMainFrame;

    SetDialogBkColor();   // gray look

    // enable file manager drag/drop and DDE Execute open
    m_pMainWnd->DragAcceptFiles();
    EnableShellOpen();

    m_server.ConnectTemplate(clsid, pDocTemplate, FALSE);
    COleTemplateServer::RegisterAll();

    // try to launch as an OLE server
    if (RunEmbedded())
    {
        // "short-circuit" initialization -- run as server!
        return TRUE;
    }
    m_server.UpdateRegistry();
    RegisterShellFileTypes();

    // not run as OLE server, so show the main window
    if (m_lpCmdLine[0] == '\0')
    {
        // create a new (empty) document
        OnFileNew();
    }
    else
    {
        // open an existing document
        OpenDocumentFile(m_lpCmdLine);
    }

    pMainFrame->ShowWindow(m_nCmdShow);
    pMainFrame->UpdateWindow();

    return TRUE;
}
```

You will notice that the code above refers to a new resource ID, IDR_HIERSVRTYPE_SRVR_EMB. This is the menu resource to be used when a document that is embedded in another container is edited. In MFC/OLE1 the menu items specific to editing an embedded item were modified on the fly. Using an entirely different menu structure when editing an embedded item instead of editing a file-based document makes it much easier to provide different user interfaces for these two separate modes. As you'll see later, an entirely separate menu resource is used when editing an embedded object in-place.

To create this resource, load the resource script into Visual C++ and copy the existing IDR_HIERSVRTYPE menu resource. Rename the new resource to IDR_HIERSVRTYPE_SRVR_EMB (this is the same naming convention that AppWizard uses). Next

change "File Save" to "File Update"; give it command ID **ID_FILE_UPDATE**. Also change "File Save As" to "File Save Copy As"; give it command ID **ID_FILE_SAVE_COPY_AS**. The framework provides the implementation of both of these commands.

```
\hiersvr\svritem.h(60) : error C2433: 'OLESTATUS' : 'virtual' not permitted on data declara
tions
\hiersvr\svritem.h(60) : error C2501: 'OLESTATUS' : missing decl-specifiers
\hiersvr\svritem.h(60) : error C2146: syntax error : missing ';' before identifier 'OnSetDa
ta'
\hiersvr\svritem.h(60) : error C2061: syntax error : identifier 'OLECLIPFORMAT'
\hiersvr\svritem.h(60) : error C2501: 'OnSetData' : missing decl-specifiers
```

There are a number of errors resulting from the override of **OnSetData**, since it is referring to the **OLESTATUS** type. **OLESTATUS** was the way OLE1 returned errors. This has changed to **HRESULT** in OLE 2, although MFC usually converts an **HRESULT** into a **COleException** containing the error. In this particular case, the override of **OnSetData** is no longer necessary, so the easiest thing to do is to remove it.

```
\hiersvr\svritem.cpp(30) : error C2660: 'COleServerItem::COleServerItem' : function does no
t take 1 parameters
```

The **COleServerItem** constructor takes an extra 'BOOL' parameter. This flag determines how memory management is done on the **COleServerItem** objects. By setting it to TRUE, the framework handles the memory management of these objects — deleting them when they are no longer necessary. HIERSVR uses **CServerItem** (derived from **COleServerItem**) objects as part of its native data, so you'll set this flag to FALSE. This lets HIERSVR determine when each server item is deleted.

```
\hiersvr\svritem.cpp(44) : error C2259: 'CServerItem' : illegal attempt to instantiate abst
ract class
\hiersvr\svritem.cpp(44) : error C2259: 'CServerItem' : illegal attempt to instantiate abst
ract class
```

As these errors imply, there are some 'pure-virtual' functions that have not been overridden in CServerItem. Most likely this is caused by the fact that OnDraw's parameter list has changed. To fix this error, change **CServerItem::OnDraw** as follows (as well as the declaration in svritem.h):

```
BOOL CServerItem::OnDraw(CDC* pDC, CSize& rSize)
{
    // request from OLE to draw node
    pDC->SetMapMode(MM_TEXT); // always in pixels
    return DoDraw(pDC, CPoint(0,0), FALSE);
}
```

The new parameter is 'rSize'. This allows you to fill in the size of the drawing, if convenient. This size must be in **HIMETRIC**. In this case, it is not convenient to fill this value in, so the framework calls **OnGetExtent** to retrieve the extent. For that to work, you'll have to implement **OnGetExtent**:

```
BOOL CServerItem::OnGetExtent(DVASPECT dwDrawAspect, CSize& rSize)
{
    if (dwDrawAspect != DVASPECT_CONTENT)
        return COleServerItem::OnGetExtent(dwDrawAspect, rSize);

    rSize = CalcNodeSize();
    return TRUE;
}

\hiersvr\svritem.cpp(104) : error C2065: 'm_rectBounds' : undeclared identifier
\hiersvr\svritem.cpp(104) : error C2228: left of '.SetRect' must have class/struct/union ty
pe
\hiersvr\svritem.cpp(106) : error C2664: 'void __pascal __far DPtoLP(struct ::tagPOINT __fa
r *,int )__far const ' : cannot convert parameter 1 from 'int __far *' to 'struct ::tagPOIN
T __far *'
```

In the CServerItem::CalcNodeSize function the item size is converted to **HIMETRIC** and stored in **m_rectBounds**. The undocumented '**m_rectBounds**' member of **COleServerItem** does not exist (it has been partially replaced by **m_sizeExtent**,

but in OLE 2 this member has a slightly different usage than **m_rectBounds** did in OLE1). Instead of setting the **HIMETRIC** size into this member variable, you'll return it. This return value is used in **OnGetExtent**, implemented previously.

```
CSize CServerItem::CalcNodeSize()
{
    CClientDC dcScreen(NULL);

    m_sizeNode = dcScreen.GetTextExtent(m_strDescription,
      m_strDescription.GetLength());
    m_sizeNode += CSize(CX_INSET * 2, CY_INSET * 2);

    // set suggested HIMETRIC size
    CSize size(m_sizeNode.cx, m_sizeNode.cy);
    dcScreen.SetMapMode(MM_HIMETRIC);
    dcScreen.DPtoLP(&size);
    return size;
}
```

CServerItem also overrides **COleServerItem::OnGetTextData**. This function is obsolete in MFC/OLE and is replaced by a different mechanism. The MFC 3.0 version of the MFC OLE sample HIERSVR implements this functionality by overriding **COleServerItem::OnRenderFileData**. This functionality is not important for this basic port, so you can remove the OnGetTextData override.

There are many more errors in svritem.cpp that have not been addressed. They are not "real" errors — just errors caused by previous errors.

```
\hiersvr\svrview.cpp(325) : error C2660: 'CopyToClipboard' : function does not take 2 param
eters
```

COleServerItem::CopyToClipboard no longer supports the 'bIncludeNative' flag. The native data (the data written out by the server item's Serialize function) is always copied, so you remove the first parameter. In addition, **CopyToClipboard** will throw an exception when an error happens instead of returning FALSE. Change the code for CServerView::OnEditCopy as follows:

```
void CServerView::OnEditCopy()
{
    if (m_pSelectedNode == NULL)
        AfxThrowNotSupportedException();

    TRY
    {
        m_pSelectedNode->CopyToClipboard(TRUE);
    }
    CATCH_ALL(e)
    {
        AfxMessageBox("Copy to clipboard failed");
    }
    END_CATCH_ALL
}
```

Although there were more errors resulting from the compilation of the MFC 2.0 version of HIERSVR than there were for the same version of OCLIENT, there were actually fewer changes.

At this point HIERSVR will compile and link and function as an OLE server, but without the in-place editing feature, which will be implemented next.

Adding "Visual Editing"

To add "Visual Editing" (or in-place activation) to this server application, there are only a few things you must take care of:

- You need a special menu resource to be used when the item is in-place active.
- This application has a toolbar, so you'll need a toolbar with only a subset of the normal toolbar to match the menu commands available from the server (matches the menu resource mentioned above).
- You need a new class derived from **COleIPFrameWnd** that provides the in-place user interface (much like CMainFrame,

derived from **CMDIFrameWnd**, provides the MDI user interface).

- You need to tell the framework about these special resources and classes.

The menu resource is easy to create. Run Visual C++, copy the menu resource IDR_HIERSVRTYPE to a menu resource called IDR_HIERSVRTYPE_SRVR_IP. Modify the menu so that only the Edit and Help menu popups are left. Add two separators to the menu in between the Edit and Help menus (it should look like: Edit | | Help). For more information on what these separators mean and how the server and container menus are merged, see "Menus and Resources: Menu Merging" in *OLE 2 Classes*.

The bitmap for the subset toolbar can be easily created by copying the one from a fresh AppWizard generated application with a "Server" option checked. This bitmap can then be imported into Visual C++. Be sure to give the bitmap an ID of IDR_HIERSVRTYPE_SRVR_IP.

The class derived from **COleIPFrameWnd** can be copied from an AppWizard generated application with server support as well. Copy both files, IPFRAME.CPP and IPFRAME.H and add them to the project. Make sure that the **LoadBitmap** call refers to IDR_HIERSVRTYPE_SRVR_IP, the bitmap created in the previous step.

Now that all the new resources and classes are created, add the necessary code so that the framework knows about these (and knows that this application now supports in-place editing). This is done by adding some more parameters to the **SetServerInfo** call in the **InitInstance** function:

```
pDocTemplate->SetServerInfo(IDR_HIERSVRTYPE_SRVR_EMB,
    IDR_HIERSVRTYPE_SRVR_IP, RUNTIME_CLASS(CInPlaceFrame));
```

It is now ready to run in-place in any container that also supports in-place activation. But, there is one minor bug still lurking in the code. HIERSVR supports a context menu, displayed when the user presses the right mouse button. This menu works when HIERSVR is fully open, but does not work when editing an embedding in-place. The reason can be pinned down to this single line of code in CServerView::OnRButtonDown:

```
pMenu->TrackPopupMenu(TPM_CENTERALIGN | TPM_RIGHTBUTTON,
    point.x, point.y, AfxGetApp()->m_pMainWnd);
```

Notice the reference to **AfxGetApp()->m_pMainWnd**. When the server is in-place activated, it has a main window and m_pMainWnd is set, but it is usually invisible. Furthermore, this window refers to the *main* window of the application, the MDI frame window that appears when the server is fully open or run stand-alone. It does not refer to the active frame window — which when in-place activated is a frame window derived from **COleIPFrameWnd**. To get the correct active window even when in-place editing, this version of MFC adds a new function, **AfxGetMainWnd**. Generally, you should use this function instead of **AfxGetApp()->m_pMainWnd**. This code needs to change as follows:

```
pMenu->TrackPopupMenu(TPM_CENTERALIGN | TPM_RIGHTBUTTON,
    point.x, point.y, AfxGetMainWnd());
```

Now you have an OLE server minimally enabled for functional in-place activation. But there are still many features available with MFC/OLE 2 that were not available in MFC/OLE1. See the HIERSVR sample for more ideas on features you might want to implement. Some of the features that HIERSVR implements are listed below:

- Zooming, for true WYSISYG behavior with respect to the container.
- Drag / drop and a custom clipboard format.
- Scrolling the container window as the selection is changed.

The HIERSVR sample in MFC 3.0 also uses a slightly different design for its server items. This helps conserve memory and makes your links more flexible. With the 2.0 version of HIERSVR each node in the tree *is-a* **COleServerItem**. **COleServerItem** carries a bit more overhead than is strictly necessary for each of these nodes, but a **COleServerItem** is required for each active link. But for the most part, there are very few active links at any given time. To make this more efficient, the HIERSVR in this version of MFC separates the node from the **COleServerItem**. It has both a CServerNode and a **CServerItem** class. The **CServerItem** (derived from **COleServerItem**) is only created as necessary. Once the container (or containers) stop using that particular link to that particular node, the CServerItem object associated with the CServerNode is deleted. This design is more efficient and more flexible. Its flexibility comes in when dealing with multiple selection links. Neither of these two versions of HIERSVR support multiple selection, but it would be much easier to add (and to support links to such selections) with the MFC 3.0 version of HIERSVR, since the **COleServerItem** is separated from the native data.

See Also

Other Resources

Technical Notes by Number
Technical Notes by Category

TN042: ODBC Driver Developer Recommendations

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes guidelines for ODBC driver writers. It outlines general requirements and assumptions of ODBC functionality that the MFC Database classes make, and various expected semantic details. Required driver functionality to support the three **CRecordset** Open modes (**forwardOnly**, **snapshot** and **dynaset**) are described.

ODBC's Cursor Library

The MFC Database classes present functionality to the user that in many cases surpasses the functionality provided by most level 1 ODBC drivers. Fortunately, ODBC's Cursor Library will layer itself between the database classes and the driver, and will automatically provide much of this additional functionality.

For instance, most 1.0 drivers do not support backward scrolling. The Cursor Library can detect this, and will cache rows from the driver and present them as requested on FETCH_PREV calls in **SQLExtendedFetch**.

Another important example of cursor library dependence is positioned updates. Most 1.0 drivers also do not have positioned updates, but the cursor library will generate update statements which identify a target row on the data source based upon its current cached data values, or a cached timestamp value.

The class library never makes use of multiple rowsets. Therefore, the few **SQLSetPos** statements are always applied to row 1 of the rowset.

CDatabases

Each **CDatabase** allocates a single **HDBC**. (If **CDatabase**'s **ExecuteSQL** function is used, an **HSTMT** is temporarily allocated.) So if multiple **CDatabase**'s are required, multiple **HDBC**s per **HENV** must be supported.

The database classes require the cursor library. This is reflected in a **SQLSetConnections** call **SQL_ODBC_CURSORS**, **SQL_CUR_USE_ODBC**.

SQLDriverConnect, **SQL_DRIVER_COMPLETE** is used by **CDatabase::Open** to establish the connection to the data source.

The driver must support **SQLGetInfo SQL_ODBC_API_CONFORMANCE** >= **SQL_OAC_LEVEL1**, **SQLGetInfo SQL_ODBC_SQL_CONFORMANCE** >= **SQL_OSC_MINIMUM**.

In order for transactions to be supported for the **CDatabase** and its dependent recordsets, **SQLGetInfo SQL_CURSOR_COMMIT_BEHAVIOR** and **SQL_CURSOR_ROLLBACK_BEHAVIOR** must have **SQL_CR_PRESERVE**. Otherwise, attempts to perform transaction control will be ignored.

SQLGetInfo SQL_DATA_SOURCE_READ_ONLY must be supported. If it returns "Y", no update operations will be performed on the data source.

If the **CDatabase** is opened ReadOnly, an attempt to set the data source read only will be made with **SQLSetConnectOption SQL_ACCESS_MODE**, **SQL_MODE_READ_ONLY**.

If identifiers require quoting, this information should be returned from the driver with an **SQLGetInfo SQL_IDENTIFIER_QUOTE_CHAR** call.

For debugging purposes, **SQLGetInfo SQL_DBMS_VER** and **SQL_DBMS_NAME** are retrieved from the driver.

SQLSetStmtOption SQL_QUERY_TIMEOUT and **SQL_ASYNC_ENABLE** may be called on a **CDatabase**'s **HDBC**.

SQLError may be called with any or all arguments NULL.

Of course, **SQLAllocEnv**, **SQLAllocConnect**, **SQLDisconnect** and **SQLFreeConnect** must be supported.

ExecuteSQL

In addition to allocating and freeing a temporary **HSTMT**, **ExecuteSQL** calls **SQLExecDirect**, **SQLFetch**, **SQLNumResultCol** and **SQLMoreResults**. **SQLCancel** may be called on the **HSTMT**.

GetDatabaseName

SQLGetInfo SQL_DATABASE_NAME will be called.

BeginTrans, CommitTrans, Rollback

SQLSetConnectOption SQL_AUTOCOMMIT and **SQLTransact SQL_COMMIT**, **SQL_ROLLBACK** and **SQL_AUTOCOMMIT** will be called if transaction requests are made.

CRecordsets

SQLAllocStmt, **SQLPrepare**, **SQLExecute** (For **Open** and **Requery**), **SQLExecDirect** (for update operations), **SQLFreeStmt** must be supported. **SQLNumResultCols** and **SQLDescribeCol** will be called on the results set at various times.

SQLSetParam is used extensively for binding parameter data and **DATA_AT_EXEC** functionality.

SQLBindCol is used extensively to register output Column data storage locations with ODBC.

Two **SQLGetData** calls are used to retrieve **SQL_LONG_VARCHAR** and **SQL_LONG_VARBINARY** data. The first call attempts to find the total length of the column value by calling **SQLGetData** with cbMaxValue of 0, but with a valid pcbValue. If pcbValue holds **SQL_NO_TOTAL**, an exception is thrown. Otherwise, an **HGLOBAL** is allocated, and another **SQLGetData** call made to retrieve the entire result.

Updating

If pessimistic locking is requested, **SQLGetInfo SQL_LOCK_TYPES** will be queried. If **SQL_LCK_EXCLUSIVE** is not supported, an exception will be thrown.

Attempts to update a **CRecordset** (**snapshot** or **dynaset**) will cause a second **HSTMT** to be allocated. For drivers that do not support second **HSTMT**, the cursor library will simulate this functionality. Unfortunately, this may sometimes mean forcing the current query on the first **HSTMT** to completion before processing the second **HSTMT**'s request.

SQLFreeStmt SQL_CLOSE and **SQL_RESET_PARAMS** and **SQLGetCursorName** will be called during update operations.

If there are **CLongBinarys** in the **outputColumns**, ODBC's **DATA_AT_EXEC** functionality must be supported. This includes returning **SQL_NEED_DATA** from **SQLExecDirect**, **SQLParamData** and **SQLPutData**.

SQLRowCount is called after executing to verify that only 1 record was updated by the **SQLExecDirect**.

ForwardOnly Cursors

Only **SQLFetch** is required for the **Move** operations. Note that **forwardOnly** cursors do not support updates.

Snapshot Cursors

Snapshot functionality requires **SQLExtendedFetch** support. As noted above, the ODBC cursor library will detect when a driver does not support **SQLExtendedFetch**, and provide the necessary support itself.

SQLGetInfo, **SQL_SCROLL_OPTIONS** must support **SQL_SO_STATIC**.

Dynaset Cursors

Below is the minimum support required to open a dynaset:

SQLGetInfo, **SQL_ODBC_VER** must return > "01".

SQLGetInfo, **SQL_SCROLL_OPTIONS** must support **SQL_SO_KEYSET_DRIVEN**.

SQLGetInfo, **SQL_ROW_UPDATES** must return "Y".

SQLGetInfo, **SQL_POSITIONED_UPDATES** must support **SQL_PS_POSITIONED_DELETE** and **SQL_PS_POSITIONED_UPDATE**.

In addition, if pessimistic locking is requested, a call to **SQLSetPos** with irow 1, fRefresh FALSE and fLock **SQL_LCK_EXCLUSIVE** will be made.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN043: RFX Routines

Note

The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes the record field exchange (RFX) architecture. It also describes how you write an **RFX_** procedure.

Overview of Record Field Exchange

All recordset field functions are done with C++ code. There are no special resources or magic macros. The heart of the mechanism is a virtual function that must be overridden in every derived recordset class. It is always found in this form:

```
void CMySet::DoFieldExchange(CFieldExchange* pFX)
{
  //{{AFX_FIELD_MAP(CMySet)
  <recordset exchange field type call>
  <recordset exchange function call>
  //}}AFX_FIELD_MAP
}
```

The special format AFX comments allow ClassWizard to locate and edit the code within this function. Code that is not compatible with ClassWizard should be placed outside of the special format comments.

In the above example, <recordset_exchange_field_type_call> is in the form:

```
pFX->SetFieldType(CFieldExchange::outputColumn);
```

and <recordset_exchange_function_call> is in the form:

```
RFX_Custom(pFX, "Col2", m_Col2);
```

Most **RFX_** functions have three arguments as shown above, but some (e.g. **RFX_Text** and **RFX_Binary**) have additional optional arguments.

More than one **RFX_** may be included in each **DoDataExchange** function.

See 'afxdb.h' for a list of all the recordset field exchange routines provided with MFC.

Recordset field calls are a way of registering memory locations (usually data members) to store field data for a **CMySet** class.

Notes

Recordset field functions are designed to work only with the **CRecordset** classes. They are not generally usable by any other MFC classes.

Initial values of data are set in the standard C++ constructor, usually in a block with `//{{AFX_FIELD_INIT(CMylSet)` and `//}}AFX_FIELD_INIT` comments.

Each **RFX_** function must support various operations, ranging from returning the dirty status of the field to archiving the field in preparation for editing the field.

Each function that calls **DoFieldExchange** (for instance **SetFieldNull**, **IsFieldDirty**), does its own initialization around the call to **DoFieldExchange**.

How Does It Work?

You do not need to understand the following in order to use record field exchange. However, understanding how this works behind the scenes will help you write your own exchange procedure.

The **DoFieldExchange** member function is much like the **Serialize** member function — it is responsible for getting or setting data to/from an external form (in this case columns from the result of an ODBC query) from/to member data in the class. The *pFX* parameter is the context for doing data exchange and is similar to the **CArchive** parameter to **CObject::Serialize**. The *pFX* (a **CFieldExchange** object) has an operation indicator, which is similar to, but a generalization of the **CArchive** direction flag. An RFX function may have to support the following operations:

- **BindParam** — Indicate where ODBC should retrieve parameter data
- **BindFieldToColumn** — Indicate where ODBC must retrieve/deposit outputColumn data
- **Fixup** — Set **CString/CByteArray** lengths, set NULL status bit
- **MarkForAddNew** — Mark dirty if value has changed since AddNew call
- **MarkForUpdate** — Mark dirty if value has changed since Edit call
- **Name** — Append field names for fields marked dirty
- **NameValue** — Append "<column name>=?" for fields marked dirty
- **Value** — Append "?" followed by separator, like ',' or ' '
- **SetFieldDirty** — Set status bit dirty (i.e. changed) field
- **SetFieldNull** — Set status bit indicating null value for field
- **IsFieldDirty** — Return value of dirty status bit
- **IsFieldNull** — Return value of null status bit
- **IsFieldNullable** — Return TRUE if field can hold NULL values
- **StoreField** — Archive field value
- **LoadField** — Reload archived field value
- **GetFieldInfoValue** — Return general information on a field
- **GetFieldInfoOrdinal** — Return general information on a field

User Extensions

There are several ways to extend the default RFX mechanism. You can

- Add new data types. For example:

```
CBookmark
```

- Add new exchange procedures (RFX_???).

```
void AFXAPI RFX_Bigint(CFieldExchange* pFX, const char *szName,
    BIGINT& value);
```

- Have the **DoFieldExchange** member function conditionally include additional RFX calls or any other valid C++ statements.

```
while (posExtraFields != NULL)
{
    RFX_Text(pFX, m_listName.GetNext(posExtraFields),
        m_listValue.GetNext(posExtraValues));
}
```

Note
Such code cannot be edited by ClassWizard and should be used only outside of the special format comments.

Writing a Custom RFX

To write your own Custom RFX function, it is suggested that you copy an existing RFX function and modify it to your own purposes. Selecting the right RFX to copy can make your job much easier. Some RFX functions have some unique properties that you should take into account when deciding which to copy.

RFX_Long and RFX_Int:

These are the simplest RFX functions. The data value does not need any special interpretation, and the data size is fixed.

RFX_Single and RFX_Double:

Like RFX_Long and RFX_Int above, these functions are simple and can make use of the default implementation extensively. They are stored in dbflt.cpp instead of dbrfx.cpp, however, to enable loading the runtime floating point library only when they are explicitly reference.

RFX_Text and RFX_Binary:

These two functions preallocate a static buffer to hold string/binary information, and must register these buffers with ODBC SQLBindCol instead of registering &value. Because of this, these two functions have lots of special-case code.

RFX_Date:

ODBC returns date and time information in their own TIMESTAMP_STRUCT data structure. This function dynamically allocates a TIMESTAMP_STRUCT as a "proxy" for sending and receiving date time data. Various operations must transfer the date and time information between the C++ **CTime** object and the TIMESTAMP_STRUCT proxy. This complicates this function considerably, but it is a good example of how to use a proxy for data transfer.

RFX_LongBinary:

This is the only class library RFX function that does not use column binding to receive and send data. This function ignores the BindFieldToColumn operation and instead, during the Fixup operation, allocates storage to hold the incoming SQL_LONGVARCHAR or SQL_LONGVARBINARY data, then performs an SQLGetData call to retrieve the value into the allocated storage. When preparing to send data values back to the data source (such as NameValue and Value operations), this function uses ODBC's DATA_AT_EXEC functionality. See Technical Note 45 for more information on working with SQL_LONGVARBINARY and SQL_LONGVARCHARs.

When writing your own **RFX_** function, you will often be able to use **CFieldExchange::Default** to implement a given operation. Look at the implementation of Default for the operation in question. If it performs the operation you would be writing in your **RFX_** function you can delegate to the **CFieldExchange::Default.** You can see examples of calling **CFieldExchange::Default** in dbrfx.cpp

It is important to call **IsFieldType** at the start of your RFX function, and return immediately if it returns FALSE. This mechanism keeps parameter operations from being performed on **outputColumns**, and vice versa (like calling **BindParam** on an **outputColumn**). In addition, **IsFieldType** automatically keeps track of the count of **outputColumns** (**m_nFields**) and params (**m_nParams**).

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN044: MFC Support for DBCS

This technical note described the support in MFC for "double-byte character sets" or DBCS. This information as well as information on MFC's support for UNICODE is now available in the *Class Library Reference*.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN045: MFC/Database Support for Long Varchar/Varbinary

Note

The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes how to retrieve and send the ODBC **SQL_LONGVARCHAR** and **SQL_LONGVARBINARY** data types using the MFC database classes.

Overview of Long Varchar/Varbinary Support

The ODBC **SQL_LONG_VARCHAR** and **SQL_LONGBINARY** data types (referred to here as long data columns) can hold huge amounts of data. There are 3 ways you can handle this data:

- Bind it to a **CString**/**CByteArray**.
- Bind it to a **CLongBinary**.
- Do not bind it at all and retrieve and send the long data value manually, independent of the database classes.

Each of the three methods has advantages and disadvantages.

Long data columns are not supported for parameters to a query. They are only supported for outputColumns.

Binding a Long Data Column to a CString/CByteArray

Advantages:

This approach is simple to understand, and you work with familiar classes. The framework provides **CFormView** support for **CString** with **DDX_Text**. You have lots of general string or collection functionality with the **CString** and **CByteArray** classes, and you can control the amount of memory allocated locally to hold the data value. The framework maintains an old copy of field data during **Edit** or **AddNew** function calls, and the framework can automatically detect changes to the data for you.

Note

Since **CString** is designed for working on character data, and **CByteArray** for working on binary data, it is recommended that you put the character data (**SQL_LONGVARCHAR**) into **CString**, and the binary data (**SQL_LONGVARBINARY**) into **CByteArray**.

The RFX functions for **CString** and **CByteArray** have an additional argument which lets you override the default size of allocated memory to hold the retrieved value for the data column. Note the nMaxLength argument in the following function declarations:

```
void AFXAPI RFX_Text(CFieldExchange* pFX, const char *szName,
    CString& value, int nMaxLength = 255, int nColumnType =
    SQL_VARCHAR);

void AFXAPI RFX_Binary(CFieldExchange* pFX, const char *szName,
    CByteArray& value,int nMaxLength = 255);
```

If you retrieve a long data column into a **CString** or **CByteArray**, the maximum returned amount of data is, by default, 255 bytes. Anything beyond this is ignored. In this case, the framework will throw the exception **AFX_SQL_ERROR_DATA_TRUNCATED**. Fortunately, you can explicitly increase nMaxLength to greater values, up to **MAXINT**.

Note

The value of nMaxLength is used by MFC to set the local buffer of the **SQLBindColumn** function. This is the local buffer for s torage of the data and does not actually affect the amount of data returned by the ODBC driver. **RFX_Text** and **RFX_Binary** o nly make one call using **SQLFetch** to retrieve the data from the back-end database. Each ODBC driver has a different limitati on on the amount of data they can return in a single fetch. This limit may be much smaller than the value set in nMaxLength, in which case the exception **AFX_SQL_ERROR_DATA_TRUNCATED** will be thrown. Under these circumstances, switch to usi ng **RFX_LongBinary** instead of **RFX_Text** or **RFX_Binary** so that all the data can be retrieved.

ClassWizard will bind a **SQL_LONGVARCHAR** to a **CString**, or a **SQL_LONGVARBINARY** to a **CByteArray** for you. If you want to allocate more than 255 bytes into which you retrieve your long data column, you can then supply an explicit value for nMaxLength.

When a long data column is bound to a **CString** or **CByteArray**, updating the field works just the same as when it is bound to a SQL_**VARCHAR** or SQL_**VARBINARY**. During **Edit**, the data value is cached away and later compared when **Update** is called to detect changes to the data value and set the Dirty and Null values for the column appropriately.

Binding a Long Data Column to a CLongBinary

If your long data column may contain more **MAXINT** bytes of data, you should probably consider retrieving it into a **CLongBinary**.

Advantages:

This retrieves an entire long data column, up to available memory.

Disadvantages:

The data is held in memory. This approach is also prohibitively expensive for very large amounts of data. You must call **SetFieldDirty** for the bound data member to ensure the field is included in an **Update** operation.

If you retrieve long data columns into a **CLongBinary**, the database classes will check the total size of the long data column, then allocate an **HGLOBAL** memory segment large enough to hold it the entire data value. The database classes then retrieve the entire data value into the allocated **HGLOBAL**.

If the data source cannot return the expected size of the long data column, the framework will throw the exception **AFX_SQL_ERROR_SQL_NO_TOTAL**. If the attempt to allocate the **HGLOBAL** fails, a standard memory exception is thrown.

ClassWizard will bind an **SQL_LONGVARCHAR** or **SQL_LONGVARBINARY** to a **CLongBinary** for you. Select **CLongBinary** as the Variable Type in the Add Member Variable dialog. ClassWizard will then add an **RFX_LongBinary** call to your **DoFieldExchange** call and increment the total number of bound fields.

To update long data column values, first make sure the allocated **HGLOBAL** is large enough to hold your new data by calling **::GlobalSize** on the **m_hData** member of the **CLongBinary**. If it is too small, release the **HGLOBAL** and allocate one the appropriate size. Then set **m_dwDataLength** to reflect the new size.

Otherwise, if **m_dwDataLength** is larger than the size of the data you're replacing, you can either free and reallocate the **HGLOBAL**, or leave it allocated. Make sure to indicate the number of bytes actually used in **m_dwDataLength**.

How Updating a CLongBinary Works

It is not necessary to understand how updating a **CLongBinary** works, but it may be useful as an example on how to send long data values to a data source, if you choose this third method, described below.

Note

In order for a **CLongBinary** field to be included in an update, you must explicitly call **SetFieldDirty** for the field. If you make any change to a field, including setting it Null, you must call **SetFieldDirty**. You must also call **SetFieldNull**, with the second parameter being **FALSE**, to mark the field as having a value.

When updating a **CLongBinary** field, the database classes use ODBC's **DATA_AT_EXEC** mechanism (see ODBC documentation on **SQLSetPos**'s rgbValue argument). When the framework prepares the insert or update statement, instead of pointing to the **HGLOBAL** containing the data, the *address* of the **CLongBinary** is set as the *value* of the column instead, and the length indicator set to **SQL_DATA_AT_EXEC**. Later, when the update statement is sent to the data source, **SQLExecDirect** will return **SQL_NEED_DATA**. This alerts the framework that the value of the param for this column is actually the address of a **CLongBinary**. The framework calls **SQLGetData** once with a small buffer, expecting the driver to return the actual length of the data. If the driver returns the actual length of the binary large object (the BLOB), MFC reallocates as much space as

necessary to fetch the BLOB. If the data source returns **SQL_NO_TOTAL**, indicating that it can't determine the size of the BLOB, MFC will create smaller blocks. The default initial size is 64K, and subsequent blocks will be double the size; for example, the second will be 128K, the third is 256K, and so on. The initial size is configurable.

Not Binding: Retrieving/Sending Data Directly from ODBC with SQLGetData

With this method you completely bypass the database classes, and deal with the long data column yourself.

Advantages:

You can cache data to disk if necessary, or decide dynamically how much data to retrieve.

Disadvantages:

You don't get the framework's **Edit** or **AddNew** support, and you must write code yourself to perform basic functionality (**Delete** does work though, since it is not a column level operation).

In this case, the long data column must be in the select list of the recordset, but should not be bound to by the framework. One way to do this is to supply your own SQL statement via **GetDefaultSQL** or as the lpszSQL argument to **CRecordset**'s **Open** function, and not bind the extra column with an RFX_ function call. ODBC requires that unbound fields appear to the right of bound fields, so add your unbound column or columns to the end of the select list.

Note
Because your long data column is not bound by the framework, changes to it will not be handled with **CRecordset::Update** calls. You must create and send the required SQL **INSERT** and **UPDATE** statements yourself.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN046: Commenting Conventions for the MFC Classes

This technical note originally described the conventions used to comment the MFC classes. This information is now covered in MFC: Using the MFC Source Files.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN047: Relaxing Database Transaction Requirements

This tech note, which discussed the transaction requirements of the MFC ODBC database classes, is now obsolete. Before MFC 4.2, the database classes required that cursors be preserved on recordsets after a **CommitTrans** or **Rollback** operation. If the ODBC driver and DBMS did not support this level of cursor preservation, then the database classes did not enable transactions.

Beginning with MFC 4.2, the database classes have relaxed the restriction of requiring cursor preservation. Transactions will be enabled if the driver supports them. However, you must now check the effect of a **CommitTrans** or **Rollback** operation on open recordsets. See the member functions CDatabase::GetCursorCommitBehavior and CDatabase::GetCursorRollbackBehavior for more information.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN048: Writing ODBC Setup and Administration Programs for MFC Database Applications

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some p rocedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the t opic of interest in the online documentation index.

Applications using MFC database classes will need a setup program that installs ODBC components. They may also need an ODBC Administration program that will retrieve information about the available drivers, to specify default drivers and to configure data sources. This note describes the use of the ODBC Installer API to write these programs.

Writing an ODBC Setup Program

An MFC database application requires the ODBC Driver Manager (ODBC.DLL) and ODBC drivers to be able to get to data sources. Many ODBC drivers also require additional network and communication DLLs. Most ODBC drivers ship with a setup program that will install the required ODBC components. Application developers using MFC database classes can:

- Rely on the driver-specific setup programs for installing ODBC components. This will require no further work on the developer's part — you can just redistribute the driver's setup program.
- Alternatively, you can write your own setup program, which will install the driver manager and the driver.

The ODBC installer API can be used to write application-specific setup programs. The functions in the installer API are implemented by the ODBC installer DLL — ODBCINST.DLL on 16-bit Windows and ODBCCP32.DLL on Win32. An application can call **SQLInstallODBC** in the installer DLL, which will install the ODBC driver manager, ODBC drivers, and any required translators. It then records the installed drivers and translators in the ODBCINST.INI file (or the registry, on NT). **SQLInstallODBC** requires the full path to the ODBC.INF file, which contains the list of drivers to be installed and describes the files that comprise each driver. It also contains similar information about the driver manager and translators. ODBC.INF files are typically supplied by driver developers.

A program can also install individual ODBC components. To install the Driver Manager, a program first calls **SQLInstallDriverManager** in the installer DLL to get the target directory for the Driver Manager. This is usually the directory in which Windows DLLs reside. The program then uses the information in the [ODBC Driver Manager] section of the ODBC.INF file to copy the Driver Manager and related files from the installation disk to this directory. To install an individual driver, a program first calls **SQLInstallDriver** in the installer DLL to add the driver specification to the ODBCINST.INI file (or the registry, on NT). **SQLInstallDriver** returns the driver's target directory — usually the directory in which Windows DLLs reside. The program then uses the information in the driver's section of the ODBC.INF file to copy the driver DLL and related files from the installation disk to this directory.

For more information on ODBC.INF, ODBCINST.INI and using the installer API, see ODBC SDK *Programmer's Reference,* Chapter 19, Installing ODBC Software.

Writing an ODBC Administrator

An MFC database application can set up and configure ODBC data sources in one of two ways, as follows:

- Use the ODBC Administrator (available as a program or as a Control Panel item).
- Create your own program to configure data sources.

A program that configures data sources makes function calls to the installer DLL. The installer DLL calls a setup DLL to configure a data source. There is one setup DLL for each driver; it may be the driver DLL itself, or a separate DLL. The setup DLL prompts the user for information that the driver needs to connect to the data source and the default translator, if supported. It then calls the installer DLL and Windows APIs to record this information in the ODBC.INI file (or registry).

To display a dialog box with which a user can add, modify, and delete data sources, a program calls **SQLManageDataSources** in the installer DLL. This function is invoked when the installer DLL is called from the Control Panel. To add, modify, or delete a data source, **SQLManageDataSources** calls **ConfigDSN** in the setup DLL for the driver associated with that data source. To directly add, modify, or delete data sources, a program calls **SQLConfigDataSource** in the installer DLL. The program passes

the name of the data source and an option that specifies the action to take. **SQLConfigDataSource** calls **ConfigDSN** in the setup DLL and passes it the arguments from **SQLConfigDataSource**.

For more information, see ODBC SDK *Programmer's Reference,* Chapter 23, Setup DLL Function Reference, and Chapter 24, Installer DLL Function Reference.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN049: MFC/OLE MBCS to Unicode Translation Layer (MFCANS32)

This note originally described how MFCANS32.DLL provides ANSI interfaces in the primarily Unicode world of 32-bit OLE. This DLL is no longer used by MFC.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN050: MFC/OLE Common Dialogs (MFCUIx32)

This note originally covered some issues and the future of the OLE common dialogs provided and used by MFC. The OLE common dialogs are now provided as a component built-in to the system (OLEDLG.DLL) and are fully documented in the product documentation.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN051: Using CTL3D Now and in the Future

This technical note, which previously discussed CTL3D and MFC, is now obsolete. The 3D effect for controls is automatically implemented by MFC.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN052: Writing Windows 95 Applications with MFC 3.1

This note originally covered issues specific to writing applications for pre-release versions of Windows 95. MFC fully supports writing great Windows 95 applications. The issues and APIs specific to Windows 95 are fully covered in the product documentation.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN053: Custom DFX Routines for DAO Database Classes

Note

As of Visual C++ .NET, the Visual C++ environment and wizards no longer support DAO (although the DAO classes are included and you can still use them). Microsoft recommends that you use OLE DB Templates or ODBC and MFC for new projects. You should only use DAO in maintaining existing applications.

This technical note describes the DAO record field exchange (DFX) mechanism. To help understand what is happening in the DFX routines, the **DFX_Text** function will be explained in detail as an example. As an additional source of information to this technical note, you can examine the code for the other the individual DFX functions. You probably will not need a custom DFX routine as often as you might need a custom RFX routine (used with ODBC database classes).

This technical note contains:

- DFX Overview
- Examples using DAO Record Field Exchange and Dynamic Binding
- How DFX Works
- What Your Custom DFX Routine Does
- Details of DFX_Text

DFX Overview

The DAO record field exchange mechanism (DFX) is used to simplify the procedure of retrieving and updating data when using the **CDaoRecordset** class. The process is simplified using data members of the **CDaoRecordset** class. By deriving from **CDaoRecordset**, you can add data members to the derived class representing each field in a table or query. This "static binding" mechanism is simple, but it may not be the data fetch/update method of choice for all applications. DFX retrieves every bound field each time the current record is changed. If you are developing a performance-sensitive application that does not require fetching every field when currency is changed, "dynamic binding" via **CDaoRecordset::GetFieldValue** and **CDaoRecordset::SetFieldValue** may be the data access method of choice.

Note

DFX and dynamic binding are not mutually exclusive, so a hybrid use of static and dynamic binding can be used.

Example 1 — Use of DAO Record Field Exchange only

(assumes **CDaoRecordset** — derived class `CMySet` already open)

```
// Add a new record to the customers table
myset.AddNew();
myset.m_strCustID = _T("MSFT");
myset.m_strCustName = _T("Microsoft");
myset.Update();
```

Example 2 — Use of dynamic binding only

(assumes using **CDaoRecordset** class, `rs`, and it is already open)

```
// Add a new record to the customers table
COleVariant  varFieldValue1 ( _T("MSFT"), VT_BSTRT );
//Note: VT_BSTRT flags string type as ANSI, instead of UNICODE default
COleVariant  varFieldValue2  (_T("Microsoft"), VT_BSTRT );
rs.AddNew();
rs.SetFieldValue(_T("Customer_ID"), varFieldValue1);
rs.SetFieldValue(_T("Customer_Name"), varFieldValue2);
rs.Update();
```

Example 3 — Use of DAO Record Field Exchange and dynamic binding

(assumes browsing employee data with **CDaoRecordset**-derived class `emp`)

```
// Get the employee's data so that it can be displayed
emp.MoveNext();

// If user wants to see employee's photograph,
// fetch it
COleVariant varPhoto;
if (bSeePicture)
   emp.GetFieldValue(_T("photo"), varPhoto);

// Display the data
PopUpEmployeeData(emp.m_strFirstName,
    emp.m_strLastName, varPhoto);
```

How DFX Works

The DFX mechanism works in a similar fashion to the record field exchange (RFX) mechanism used by the MFC ODBC classes. The principles of DFX and RFX are the same but there are numerous internal differences. The design of the DFX functions was such that virtually all the code is shared by the individual DFX routines. At the highest level DFX only does a few things.

- DFX constructs the SQL **SELECT** clause and SQL **PARAMETERS** clause if necessary.
- DFX constructs the binding structure used by DAO's **GetRows** function (more on this later).
- DFX manages the data buffer used to detect dirty fields (if double-buffering is being used)
- DFX manages the **NULL** and **DIRTY** status arrays and sets values if necessary on updates.

At the heart of the DFX mechanism is the **CDaoRecordset** derived class's **DoFieldExchange** function. This function dispatches calls to the individual DFX functions of an appropriate operation type. Before calling **DoFieldExchange** the internal MFC functions set the operation type. The following list shows the various operation types and a brief description.

Operation	Description
AddToParameterList	Builds PARAMETERS clause
AddToSelectList	Builds SELECT clause
BindField	Sets up binding structure
BindParam	Sets parameter values
Fixup	Sets NULL status
AllocCache	Allocates cache for dirty check
StoreField	Saves current record to cache
LoadField	Restores cache to member values
FreeCache	Frees cache
SetFieldNull	Sets field status & value to NULL
MarkForAddNew	Marks fields dirty if not PSEUDO NULL
MarkForEdit	Marks fields dirty if don't match cache

SetDirtyField	Sets field values marked as dirty

In the next section, each operation will be explained in more detail for **DFX_Text**.

The most important feature to understand about the DAO record field exchange process is that it uses the **GetRows** function of the **CDaoRecordset** object. The DAO **GetRows** function can work in several ways. This technical note will only briefly describe **GetRows** as it is outside of the scope of this technical note.

DAO **GetRows** can work in several ways.

- It can fetch multiple records and multiple fields of data at one time. This allows for faster data access with the complication of dealing with a large data structure and the appropriate offsets to each field and for each record of data in the structure. MFC does not take advantage of this multiple record fetching mechanism.
- Another way **GetRows** can work is to allow programmers to specify binding addresses for the retrieved data of each field for one record of data.
- DAO will also "call back" into the caller for variable length columns in order to allow the caller to allocate memory. This second feature has the benefit of minimizing the number of copies of data as well as allowing direct storage of data into members of a class (the **CDaoRecordset** derived class). This second mechanism is the method MFC uses to bind to data members in **CDaoRecordset** derived classes.

What Your Custom DFX Routine Does

It is apparent from this discussion that the most important operation implemented in any DFX function must be the ability to set up the required data structures to successfully call **GetRows**. There are a number of other operations that a DFX function must support as well, but none as important or complex as correctly preparing for the **GetRows** call.

The use of DFX is described in the online documentation. Essentially, there are two requirements. First, members must be added to the **CDaoRecordset** derived class for each bound field and parameter. Following this **CDaoRecordset::DoFieldExchange** should be overridden. Note that the data type of the member is important. It should match the data from the field in the database or at least be convertible to that type. For example a numeric field in database, such as a long integer, can always be converted to text and bound to a **CString** member, but a text field in a database may not necessarily be converted to a numeric representation, such as long integer and bound to a long integer member. DAO and the Microsoft Jet database engine are responsible for the conversion (rather than MFC).

Details of DFX_Text

As mentioned previously, the best way to explain how DFX works is to work through an example. For this purpose going through the internals of **DFX_Text** should work quite well to help provide at least a basic understanding of DFX.

AddToParameterList

This operation builds the SQL **PARAMETERS** clause ("`Parameters <param name>, <param type> ... ;`") required by Jet. Each parameter is named and typed (as specified in the RFX call). See the function **CDaoFieldExchange::AppendParamType** function to see the names of the individual types. In the case of **DFX_Text**, the type used is `text`.

AddToSelectList

Builds the SQL **SELECT** clause. This is pretty straight forward as the column name specified by the DFX call is simply appended ("`SELECT <column name>, ...`").

BindField

The most complex of the operations. As mentioned previously this is where the DAO binding structure used by **GetRows** is set up. As you can see from the code in **DFX_Text** the types of information in the structure include the DAO type used (**DAO_CHAR** or **DAO_WCHAR** in the case of **DFX_Text**). Additionally, the type of binding used is also set up. In an earlier section **GetRows** was described only briefly, but it was sufficient to explain that the type of binding used by MFC is always direct address binding (**DAOBINDING_DIRECT**). In addition for variable-length column binding (like **DFX_Text**) callback binding is used so that MFC can control the memory allocation and specify an address of the correct length. What this means is that MFC can always tell DAO "where" to put the data, thus allowing binding directly to member variables. The rest of the binding structure is filled in with things like the address of the memory allocation callback function and the type of column binding (binding by column name).

BindParam

This is a simple operation that calls **SetParamValue** with the parameter value specified in your parameter member.

Fixup

Fills in the **NULL** status for each field.

SetFieldNull

This operation only marks each field status as **NULL** and sets the member variable's value to **PSEUDO_NULL**.

SetDirtyField

Calls **SetFieldValue** for each field marked dirty.

All the remaining operations only deal with using the data cache. The data cache is an extra buffer of the data in the current record that is used to make certain things simpler. For instance, "dirty" fields can be automatically detected. As described in the online documentation it can be turned off completely or at the field level. The implementation of the buffer utilizes a map. This map is used to match up dynamically allocated copies of the data with the address of the "bound" field (or **CDaoRecordset** derived data member).

AllocCache

Dynamically allocates the cached field value and adds it to the map.

FreeCache

Deletes the cached field value and removes it from the map.

StoreField

Copies the current field value into the data cache.

LoadField

Copies the cached value into the field member.

MarkForAddNew

Checks if current field value is non-**NULL** and marks it dirty if necessary.

MarkForEdit

Compares current field value with data cache and marks dirty if necessary.

Tip
Model your custom DFX routines on the existing DFX routines for standard data types.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN054: Calling DAO Directly While Using MFC DAO Classes

Note
As of Visual C++ .NET, the Visual C++ environment and wizards no longer support DAO (although the DAO classes are included and you can still use them). Microsoft recommends that you use OLE DB Templates or ODBC and MFC for new projects. You should only use DAO in maintaining existing applications.

When using the MFC DAO database classes, there may be situations where it is necessary to use DAO directly. Usually, this will not be the case, but MFC has provided some helper mechanisms to facilitate making direct DAO calls simple when combining the use of the MFC classes with direct DAO calls. Making direct DAO calls to the methods of an MFC-managed DAO object should require only a few lines of code. If you need to create and use DAO objects that are *not* managed by MFC, you will have to do a little more work by actually calling **Release** on the object. This technical note explains when you might want to call DAO directly, what the MFC helpers can do to help you, and how to use the DAO OLE interfaces. Finally, this note provides some sample functions showing how to call DAO directly for DAO security features.

When to Make Direct DAO Calls

The most common situations for making direct DAO calls occur when collections need to be refreshed or when you are implementing features not wrapped by MFC. The most significant feature not exposed by MFC is security. If you want to implement security features, you will need to use the DAO User(s) and Group(s) objects directly. Besides security, there are only a few other DAO features not supported by MFC. These include recordset cloning and database replication features as well as a few late additions to DAO.

A Brief Overview of DAO and MFC's Implementation

MFC's wrapping of DAO makes using DAO easier by handling many of the details so you do not have to worry about the little things. This includes the initialization of OLE, the creation and management of the DAO objects (especially the collection objects), error checking, and providing a strongly typed, simpler interface (no **VARIANT** or **BSTR** arguments). You can make direct DAO calls and still take advantage of these features. All your code must do is call **Release** for any objects created by direct DAO calls and *not* modify any of the interface pointers that MFC may rely on internally. For example, do not modify the **m_pDAORecordset** member of an open **CDaoRecordset** object unless you understand *all* the internal ramifications. You could, however, use the **m_pDAORecordset** interface to call DAO directly to get the Fields collection. In this case the **m_pDAORecordset** member would not be modified. You simply have to call **Release** on the Fields collection object when you are finished with the object.

Description of Helpers to Make DAO Calls Easier

The helpers provided to make calling DAO easier are the same helpers that are used internally in the MFC DAO Database classes. These helpers are used to check the return codes when making a direct DAO call, logging debug output, checking for expected errors, and throwing appropriate exceptions if necessary. There are two underlying helper functions and four macros that map to one of these two helpers. The best explanation would be to simply read the code. See **DAO_CHECK**, **DAO_CHECK_ERROR**, **DAO_CHECK_MEM**, and **DAO_TRACE** in AFXDAO.H to see the macros, and see **AfxDaoCheck** and **AfxDaoTrace** in DAOCORE.CPP.

Using the DAO OLE Interfaces

The OLE interfaces for each object in the DAO object hierarchy are defined in the header file DBDAOINT.H, which is found in the \Program Files\Microsoft Visual Studio .NET 2003\VC7\include directory. These interfaces provide methods that allow you to manipulate the entire DAO hierarchy.

For many of the methods in the DAO interfaces, you will need to manipulate a **BSTR** object (a length-prefixed string used in OLE automation). The **BSTR** object typically is encapsulated within the **VARIANT** data type. The MFC class **COleVariant** itself inherits from the **VARIANT** data type. Depending on whether you build your project for ANSI or Unicode, the DAO interfaces will return ANSI or Unicode **BSTR**s. Two macros, **V_BSTR** and **V_BSTRT**, are useful for assuring that the DAO interface gets the **BSTR** of the expected type.

V_BSTR will extract the **bstrVal** member of a **COleVariant**. This macro is typically used when you need to pass the contents of a **COleVariant** to a method of a DAO interface. The following code fragment shows both the declarations and actual use for two methods of the DAO DAOUser interface that take advantage of the **V_BSTR** macro:

```
COleVariant varOldName;
```

```
COleVariant varNewName( _T("NewUser"), VT_BSTRT );

// Code to assign pUser to a valid value omitted
DAOUser *pUser = NULL;

// These method declarations were taken from DBDAOINT.H
// STDMETHOD(get_Name) (THIS_ BSTR FAR* pbstr) PURE;
// STDMETHOD(put_Name) (THIS_ BSTR bstr) PURE;

DAO_CHECK( pUser->get_Name( &V_BSTR ( &varOldName ) ));
DAO_CHECK( pUser->put_Name( V_BSTR ( &varNewName ) ));
```

Note that the **VT_BSTRT** argument specified in the **COleVariant** constructor above ensures that there will be an ANSI **BSTR** in the **COleVariant** if you build an ANSI version of your application and a Unicode **BSTR** for a Unicode version of your application. This is what DAO expects.

The other macro, **V_BSTRT**, will extract either an ANSI or Unicode **bstrVal** member of **COleVariant** depending on the type of build (ANSI or Unicode). The following code demonstrates how to extract the **BSTR** value from a **COleVariant** into a **CString**:

```
COleVariant varName( _T( "MyName" ), VT_BSTRT );
CString str = V_BSTRT( &varName );
```

The **V_BSTRT** macro, along with other tricks to crack open other types stored within **COleVariant**, is demonstrated in the DAOVIEW sample included on the Visual C++ CD. Specifically, this translation is performed in the **CCrack::strVARIANT** method. This method, where possible, translates the value of a **COleVariant** into an instance of **CString**.

Simple Example of a Direct Call to DAO

Situations may arise when it is necessary to refresh the underlying DAO collection objects. Normally, this should not be necessary, but it is a simple procedure if it is necessary. An example of when a collection might need to be refreshed is when operating in a multiuser environment with multiple users creating new tabledefs. In this case your tabledefs collection might become stale. To refresh the collection, you simply need to call the **Refresh** method of the particular collection object and check for errors:

```
DAO_CHECK( pMyDaoDatabase->
    m_pDAOTableDefs->Refresh( ) );
```

Note that currently all DAO collection object interfaces are undocumented implementation details of the MFC DAO database classes.

Using DAO Directly for DAO Security Features

The MFC DAO database classes do not wrap DAO security features. You must call methods of DAO interfaces to use some DAO security features. The following function sets the system database and then changes the user's password. This function calls three other functions, which are subsequently defined.

```
void ChangeUserPassword( )
{
   // Specify path to the Microsoft Access
   // system database
   CString strSystemDB =
     _T( "c:\\Program Files\\MSOffice\\access\\System.mdw" );

   // Set system database before MFC initilizes DAO
   // NOTE: An MFC module uses only one instance
   // of a DAO database engine object. If you have
   // called a DAO object in your application prior
   // to calling the function below, you must call
   // AfxDaoTerm to destroy the existing database
   // engine object. Otherwise, the database engine
   // object already in use will be reused, and setting
   // a system datbase will have no effect.
   //
   // If you have used a DAO object prior to calling
   // this function it is important that DAO be
```

```
    // terminated with AfxDaoTerm since an MFC
    // module only gets one copy of the database engine
    // and that engine will be reused if it hasn't been
    // terminated. In other words, if you do not call
    // AfxDaoTerm and there is currently a database
    // initialized, setting the system database will
    // have no affect.

    SetSystemDB( strSystemDB );

    // User name and password manually added
    // by using Microsoft Access
    CString strUserName = _T( "NewUser" );
    CString strOldPassword = _T( "Password" );
    CString strNewPassword = _T( "NewPassword" );

    // Set default user so that MFC will be able
    // to log in by default using the user name and
    // password from the system database
    SetDefaultUser( strUserName, strOldPassword );

    // Change the password. You should be able to
    // call this function from anywhere in your
    // MFC application
    ChangePassword( strUserName, strOldPassword,
                    strNewPassword );

    .
    .
    .

  }
```

The next four examples demonstrate how to:

- Set the system DAO database (.MDW file).
- Set the default user and password.
- Change the password of a user.
- Change the password of an .MDB file.

Setting the System Database

Below is a sample function to set the system database that will be used by an application. This function must be called before any other DAO calls are made.

```
  // Set the system database that the
  // DAO database engine will use

  void SetSystemDB( CString & strSystemMDB )
  {
     COleVariant varSystemDB( strSystemMDB, VT_BSTRT );

     // Initialize DAO for MFC
     AfxDaoInit( );
     DAODBEngine* pDBEngine = AfxDaoGetEngine( );

     ASSERT( pDBEngine != NULL );

     // Call put_SystemDB method to set the
     // system database for DAO engine
     DAO_CHECK( pDBEngine->put_SystemDB( varSystemDB.bstrVal ) );
  }
```

Setting the Default User and Password

To set the default user and password for a system database, use the following function:

To set the default user and password for a system database, use the following function:

```
void SetDefaultUser(CString & strUserName, CString & strPassword)
{
  COleVariant varUserName( strUserName, VT_BSTRT );
  COleVariant varPassword( strPassword, VT_BSTRT );

  DAODBEngine* pDBEngine = AfxDaoGetEngine( );
  ASSERT( pDBEngine != NULL );

  // Set default user:
  DAO_CHECK( pDBEngine->put_DefaultUser( varUserName.bstrVal ) );

  // Set default password:
  DAO_CHECK( pDBEngine->put_DefaultPassword( varPassword.bstrVal ) );
}
```

Changing a User's Password

To change a user's password, use the following function:

```
void ChangePassword( CString &strUserName,
                     CString &strOldPassword,
                     CString &strNewPassword )
{
   // Create (open) a workspace
   CDaoWorkspace wsp;
   CString strWspName = _T( "Temp Workspace" );

   wsp.Create( strWspName, strUserName,
               strOldPassword );
   wsp.Append( );

   // Determine how many objects there are
   // in the Users collection
   short nUserCount;
   short nCurrentUser;
   DAOUser *pUser  = NULL;
   DAOUsers *pUsers = NULL;

   // Side-effect is implicit OLE AddRef( )
   // on DAOUser object:
   DAO_CHECK( wsp.m_pDAOWorkspace->get_Users( &pUsers ) );

   // Side-effect is implicit OLE AddRef( )
   // on DAOUsers object
    DAO_CHECK( pUsers->get_Count( &nUserCount ) );

   // Traverse through the list of users
   // and change password for the userid
   // used to create/open the workspace
   for( nCurrentUser = 0; nCurrentUser < nUserCount;
        nCurrentUser++ )
   {
       COleVariant varIndex( nCurrentUser, VT_I2 );
       COleVariant varName;

       // Retrieve information for user nCurrentUser
       DAO_CHECK( pUsers->get_Item( varIndex, &pUser ) );

       // Retrieve name for user nCurrentUser
       DAO_CHECK( pUser->get_Name( &V_BSTR( &varName ) ) );

       CString strTemp = V_BSTRT( &varName );

       // If there is a match, change the password
       if( strTemp == strUserName )
       {
```

```
            COleVariant varOldPwd( strOldPassword,
                                   VT_BSTRT );
            COleVariant varNewPwd( strNewPassword,
                                   VT_BSTRT );

            DAO_CHECK( pUser->NewPassword( V_BSTR( &varOldPwd ),
                       V_BSTR( &varNewPwd ) ) );

            TRACE( "\t Password is changed\n" );
        }
    }

    // Clean up: decrement the usage count
    // on the OLE objects
    pUser->Release( );
    pUsers->Release( );

    wsp.Close( );
}
```

Changing the Password of an .MDB File

To change the password of an .MDB file, use the following function:

```
void SetDBPassword( LPCTSTR pDB, LPCTSTR pszOldPassword, LPCTSTR pszNewPassword )
{
   CDaoDatabase db;
   CString strConnect( _T( ";pwd=" ) );

   // the database must be opened as exclusive
   // to set a password
   db.Open( pDB, TRUE, FALSE,
            strConnect + pszOldPassword );

   COleVariant NewPassword( pszNewPassword, VT_BSTRT ),
               OldPassword( pszOldPassword, VT_BSTRT );

   DAO_CHECK( db.m_pDAODatabase->NewPassword( V_BSTR( &OldPassword ),
              V_BSTR( &NewPassword ) ) );

   db.Close();
}
```

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN055: Migrating MFC ODBC Database Class Applications to MFC DAO Classes

Note
As of Visual C++ .NET, the Visual C++ environment and wizards no longer support DAO (although the DAO classes are included and you can still use them). Microsoft recommends that you use OLE DB Templates or ODBC and MFC for new projects. You should only use DAO in maintaining existing applications.

Overview

In many situations it may be desirable to migrate applications that use MFC's ODBC database classes to MFC's DAO database classes. This technical note will detail most of the differences between the MFC ODBC and DAO classes. With the differences in mind, it should not be overly difficult to migrate applications from the ODBC classes to the MFC classes if desired.

Why Migrate from ODBC to DAO?

There are a number of reasons why you might want to migrate applications from the ODBC Database Classes to the DAO Database Classes, but the decision is not necessarily simple or obvious. One thing to keep in mind is that the Microsoft Jet database engine that is used by DAO can read any ODBC data source for which you have an ODBC driver. It may be more efficient to use the ODBC Database Classes or call ODBC directly yourself, but the Microsoft Jet database engine can read ODBC data.

Some simple cases that make the ODBC/DAO decision easy. For instance, when you only need access to data in a format that the Microsoft Jet engine can read directly (Access format, Excel format, and so on) the obvious choice is to use the DAO Database Classes.

More complex cases arise when your data exists on a server or on a variety of different servers. In this case, the decision to use the ODBC Database classes or the DAO Database classes is a difficult one. If you want to do things like heterogeneous joins (join data from servers in multiple formats like SQL Server and Oracle), then the Microsoft Jet database engine will perform the join for you rather than forcing you to do the work necessary if you used the ODBC Database Classes or called ODBC directly. If you are using an ODBC driver that supports driver cursors, your best choice might be the ODBC Database classes.

The choice can be complicated, so you might want to write some sample code to test the performance of various methods given your special needs. This technical note assumes that you have made the decision to migrate from the ODBC Database Classes to the DAO Database classes.

Similarities Between ODBC Database Classes and MFC DAO Database Classes

The original design of the MFC ODBC classes was based on the DAO object model that has been in use in Microsoft Access and Microsoft Visual Basic. This means that there are many common features of the ODBC and DAO MFC classes, which will not all be listed in this section. In general, the programming models are the same.

To highlight a few similarities:

- Both the ODBC and DAO classes have database objects that manage using the underlying database management system (DBMS).
- Both have recordset objects representing a set of results returned from that DBMS.
- The DAO database and recordset objects have members nearly identical to the ODBC classes.
- With both sets of classes, the code to retrieve data is identical except for some object and member name changes. Changes will be required, but usually the process is a straightforward name change when switching from the ODBC classes to DAO classes.

For example, in both models the procedure to retrieve data is to create and open a database object, create and open a recordset object, and navigate (move) though the data performing some operation.

Differences Between ODBC and DAO MFC Classes

The DAO classes include more objects and a richer set of methods, but this section will only detail the differences in similar classes and functionality.

Probably the most obvious differences between the classes are the name changes for similar classes and global functions. The following list shows the name changes of the objects, methods and global functions associated with the database classes:

Class or function	Equivalent in MFC DAO classes
CDatabase	**CDaoDatabase**
CDatabase::ExecuteSQL	**CDaoDatabase::Execute**
CRecordset	**CDaoRecordset**
CRecordset::GetDefaultConnect	**CDaoRecordset::GetDefaultDBName**
CFieldExchange	**CDaoFieldExchange**
RFX_Bool	**DFX_Bool**
RFX_Byte	**DFX_Byte**
RFX_Int	**DFX_Short**
RFX_Long	**DFX_Long**
	DFX_Currency
RFX_Single	**DFX_Single**
RFX_Double	**DFX_Double**
RFX_Date *	**DFX_Date** (**COleDateTime**-based)
RFX_Text	**DFX_Text**
RFX_Binary	**DFX_Binary**
RFX_LongBinary	**DFX_LongBinary**

* The **RFX_Date** function is based on **CTime** and **TIMESTAMP_STRUCT**.

The major changes to functionality which may affect your application and require more than simple name changes are listed below.

- The constants and macros used to specify things like recordset open type and recordset open options have been changed.

  With the ODBC classes MFC needed to define these options via macros or enumerated types.

  With the DAO classes, DAO provides the definition of these options in a header file (DBDAOINT.H). Thus the recordset type is an enumerated member of **CRecordset**, but with DAO it is a constant instead. For example you would use **snapshot** when specifying the type of **CRecordset** in ODBC but **DB_OPEN_SNAPSHOT** when specifying the type of **CDaoRecordset**.

- The default recordset type for **CRecordset** is **snapshot** while the default recordset type for **CDaoRecordset** is **dynaset** (see the Note below for an additional issue about ODBC class snapshots).

- The ODBC **CRecordset** class has an option to create a forward-only recordset type. In the **CDaoRecordset** class, forward-only is not a recordset type, but rather a property (or option) of certain types of recordsets.

- An append-only recordset when opening a **CRecordset** object meant that the recordset's data could be read and appended. With **CDaoRecordset** object, the append-only option means literally that the recordset's data can only be

appended (and not read).

- The ODBC classes' transaction member functions are members of **CDatabase** and act at the database level. In the DAO classes, the transaction member functions are members of a higher level class (**CDaoWorkspace**) and thus may impact multiple **CDaoDatabase** objects sharing the same workspace (transaction space).
- The exception class has been changed. **CDBExceptions** are thrown in the ODBC classes and **CDaoExceptions** in the DAO classes.
- **RFX_Date** uses **CTime** and **TIMESTAMP_STRUCT** objects while **DFX_Date** uses **COleDateTime**. The **COleDateTime** is nearly identical to **CTime**, but is based on a 8-byte OLE **DATE** rather than a 4-byte **time_t** so it can hold a much bigger range of data.

Note
DAO (**CDaoRecordset**) snapshots are read-only while ODBC (**CRecordset**) snapshots may be updateable depending on the driver and use of the ODBC cursor library. If you are using the cursor library, **CRecordset** snapshots are updateable. If you are using any of the Microsoft drivers from Desktop Driver Pack 3.0 without the ODBC cursor library, the **CRecordset** snapshots are read-only. If you are using another driver, check the driver's documentation to see if snapshots (**STATIC_CURSORS**) are read-only.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN056: Installation of Localized MFC Components

This tech note, which discussed the installation of localized MFC components, is now obsolete.

See Deploying Applications for additional **information on redistributing Visual C++ applications. Also see** TechNote 57 for more information on localizing MFC applications.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN057: Localization of MFC Components

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes some of the designs and procedures you can use to localize your component, if it an application or an OLE control or a DLL that uses MFC.

Overview

There are really two issues to resolve when localizing a component that uses MFC. First, you must localize your own resources — strings, dialogs, and other resources that are specific to your component. Most components built using MFC also include and use a number of resources that are defined by MFC. You must provide localized MFC resources as well. Fortunately, several languages are already provided by MFC itself.

In addition, your component should be prepared to run in its target environment (European or DBCS-enabled environment). For the most part, this depends on your application treating characters with the high bit set correctly and handling strings with double byte characters. MFC is enabled, by default, for both of these environments, such that it is possible to have a single worldwide binary that is used on all platforms with just different resources plugged in at setup time.

Localizing your Component's Resources

Localizing your application or DLL should involve simply replacing the resources with resources that match the target language. For your own resources, this is relatively simple: edit the resources in the resource editor and build your application. If your code is written properly there will be no strings or text that you wish to localize hard-coded into your C++ source code – all localization can be done by simply modifying resources. In fact, you can implement your component such that all providing a localized version does not even involve a build of the original code. This is more complex, but is well worth it and is the mechanism chosen for MFC itself. It is also possible to localize an application by loading the EXE or DLL file into the resource editor and editing the resources directly. While possible, it is requires reapplication of those changes each time you build a new version of your application.

One way to avoid that is to locate all resources in a separate DLL, sometimes called a satellite DLL. This DLL is then loaded dynamically at runtime and the resources are loaded from that DLL instead of from the main module with all your code. MFC directly supports this approach. Consider an application called MYAPP.EXE; it could have all of its resources located in a DLL called MYRES.DLL. In the application's **InitInstance** it would perform the following to load that DLL and cause MFC to load resources from that location:

```
CMyApp::InitInstance()
{
   // one of the first things in the init code
   HINSTANCE hInst = LoadLibrary("myres.dll");
   if (hInst != NULL)
      AfxSetResourceHandle(hInst);

   // other initialization code would follow
   .
   .
   .
}
```

From then on, MFC will load resources from that DLL instead of from myapp.exe. All resources, however, must be present in that DLL; MFC will not search the application's instance in search of a given resource. This technique applies equally well to regular DLLs as well as OLE Controls. Your setup program would copy the appropriate version of MYRES.DLL depending upon which resource locale the user would like.

It is relatively easy to create a resource only DLL. You create a DLL project, add your .RC file to it, and add the necessary resources. If you have an existing project that does not use this technique, you can copy the resources from that project. After adding the resource file to the project, you are almost ready to build the project. The only thing you must do is set the linker

options to include **/NOENTRY**. This tells the linker that the DLL has no entry point – since it has no code, it has no entry point.

Note
The resource editor in Visual C++ 4.0 and later supports multiple languages per .RC file. This can make it very easy to manage your localization in a single project. The resources for each language are controlled by preprocessor directives generated by the resource editor.

Using the Provided MFC Localized Resources

Any MFC application that you build reuses two things from MFC: code and resources. That is, MFC has various error messages, built-in dialogs, and other resources that are used by the MFC classes. In order to completely localize your application, you need to localize not only your application's resources, but also the resources that come directly from MFC. MFC provides a number of different language resource files automatically, so that if the language you are targeting is one of the languages MFC already supports, you just need to make sure you use those localized resources.

As of this writing, MFC supports Chinese, German, Spanish, French, Italian, Japanese, and Korean. The files which contain these localized versions are in the MFC\INCLUDE\L.* (the 'L' stands for localized) directories. The German files are in MFC\INCLUDE\L.DEU, for example. To cause your application to use these RC files instead of the files located in MFC\INCLUDE, add a `/IC:\PROGRAM FILES\MICROSOFT VISUAL STUDIO .NET 2003\VC7\MFC\INCLUDE\L.DEU` to your RC command line (this is just an example; you would need to substitute your locale of choice as well as the directory into which you installed Visual C++).

The above instructions will work if your application links statically with MFC. Most applications link dynamically (because that is the AppWizard default). In this scenario, not only the code is dynamically linked – so are the resources. As a result, you can localize your resources in your application, but the MFC implementation resources will still be loaded from the MFC7x.DLL (or a later version) or from MFC7xLOC.DLL if it exists. You can approach this from two different angles.

The more complex approach is to ship one of the localized MFC7xLOC.DLLs (such as MFC7xDEU, for German, MFC7xESP.DLL for Spanish, etc.), or a later version, and install the appropriate MFC7xLOC.DLL into the system directory when the user installs your application. This can be very complex for both the developer and the end user and as such is not recommended. See Technical Note 56 for more information on this technique and its caveats.

The simplest and safest approach is to include the localized MFC resources in your application or DLL itself (or its satellite DLL if you are using one). This avoids the problems of installing MFC7xLOC.DLL properly. To do so, you follow the same instructions for the static case given above (setting the RC command line properly to point to the localized resources), except that you must also remove the `/D_AFXDLL` define that was added by AppWizard. When `/D_AFXDLL` is defined, AFXRES.H (and the other MFC RC files) do not actually define any resources (because they will be pulled from the MFC DLLs instead).

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN058: MFC Module State Implementation

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This technical note describes the implementation of MFC "module state" constructs. An understanding of the module state implementation is critical for using the MFC shared DLLs from a DLL (or OLE in-process server).

Before reading this note, refer to "Managing the State Data of MFC Modules" in Creating New Documents, Windows, and Views. This article contains important usage information and overview information on this subject.

Overview

There are three kinds of MFC state information: Module State, Process State, and Thread State. Sometimes these state types can be combined. For example, MFC's handle maps are both module local and thread local. This allows two different modules to have different maps in each of their threads.

Process State and Thread State are similar. These data items are things that have traditionally been global variables, but have need to be specific to a given process or thread for proper Win32s support or for proper multithreading support. Which category a given data item fits in depends on that item and its desired semantics with regard to process and thread boundaries.

Module State is unique in that it can contain either truly global state or state that is process local or thread local. In addition, it can be switched quickly.

Module State Switching

Each thread contains a pointer to the "current" or "active" module state (not surprisingly, the pointer is part of MFC's thread local state). This pointer is changed when the thread of execution passes a module boundary, such as an application calling into an OLE Control or DLL, or an OLE Control calling back into an application.

The current module state is switched by calling **AfxSetModuleState**. For the most part, you will never deal directly with the API. MFC, in many cases, will call it for you (at WinMain, OLE entry-points, **AfxWndProc**, etc.). This is done in any component you write by statically linking in a special **WndProc**, and a special **WinMain** (or **DllMain**) that knows which module state should be current. You can see this code by looking at DLLMODUL.CPP or APPMODUL.CPP in the MFC\SRC directory.

It is rare that you want to set the module state and then not set it back. Most of the time you want to "push" your own module state as the current one and then, after you are done, "pop" the original context back. This is done by the macro AFX_MANAGE_STATE and the special class **AFX_MAINTAIN_STATE**.

CCmdTarget has special features for supporting module state switching. In particular, a **CCmdTarget** is the root class used for OLE automation and OLE COM entry points. Like any other entry point exposed to the system, these entry points must set the correct module state. How does a given **CCmdTarget** know what the "correct" module state should be? The answer is that it "remembers" what the "current" module state is when it is constructed, such that it can set the current module state to that "remembered" value when it is later called. As a result, the module state that a given **CCmdTarget** object is associated with is the module state that was current when the object was constructed. Take a simple example of loading an INPROC server, creating an object, and calling its methods.

1. The DLL is loaded by OLE using **LoadLibrary**.
2. **RawDllMain** is called first. It sets the module state to the known static module state for the DLL. For this reason **RawDllMain** is statically linked to the DLL.
3. The constructor for the class factory associated with our object is called. **COleObjectFactory** is derived from **CCmdTarget** and as a result, it remembers in which module state it was instantiated. This is important — when the class factory is asked to create objects, it knows now what module state to make current.
4. **DllGetClassObject** is called to obtain the class factory. MFC searches the class factory list associated with this module and returns it.

5. **COleObjectFactory::XClassFactory2::CreateInstance** is called. Before creating the object and returning it, this function sets the module state to the module state that was current in step 3 (the one that was current when the **COleObjectFactory** was instantiated). This is done inside of METHOD_PROLOGUE.
6. When the object is created, it too is a **CCmdTarget** derivative and in the same way **COleObjectFactory** remembered which module state was active, so does this new object. Now the object knows which module state to switch to whenever it is called.
7. The client calls a function on the OLE COM object it received from its **CoCreateInstance** call. When the object is called it uses **METHOD_PROLOGUE** to switch the module state just like **COleObjectFactory** does.

As you can see, the module state is propagated from object to object as they are created. It is important to have the module state set appropriately. If it is not set, your DLL or COM object may interact poorly with an MFC application that is calling it, or may be unable to find its own resources, or may fail in other miserable ways.

Note that certain kinds of DLLs, specifically "MFC Extension" DLLs do not switch the module state in their **RawDllMain** (actually, they usually don't even have a **RawDllMain**). This is because they are intended to behave "as if" they were actually present in the application that uses them. They are very much a part of the application that is running and it is their intention to modify that application's global state.

OLE Controls and other DLLs are very different. They do not want to modify the calling application's state; the application that is calling them may not even be an MFC application and so there may be no state to modify. This is the reason that module state switching was invented.

For exported functions from a DLL, such as one that launches a dialog box in your DLL, you need to add the following code to the beginning of the function:

```
AFX_MANAGE_STATE(AfxGetStaticModuleState( ))
```

This swaps the current module state with the state returned from AfxGetStaticModuleState until the end of the current scope.

Problems with resources in DLLs will occur if the **AFX_MODULE_STATE** macro is not used. By default, MFC uses the resource handle of the main application to load the resource template. This template is actually stored in the DLL. The root cause is that MFC's module state information has not been switched by the **AFX_MODULE_STATE** macro. The resource handle is recovered from MFC's module state. Not switching the module state causes the wrong resource handle to be used.

AFX_MODULE_STATE does not need to be put in every function in the DLL. For example, `InitInstance` can be called by the MFC code in the application without **AFX_MODULE_STATE** because MFC automatically shifts the module state before `InitInstance` and then switches it back after `InitInstance` returns. The same is true for all message map handlers. Regular DLLs actually have a special master window procedure that automatically switches the module state before routing any message.

Process Local Data

Process local data would not be of such great concern had it not been for the difficulty of the Win32s DLL model. In Win32s all DLLs share their global data, even when loaded by multiple applications. This is very different from the "real" Win32 DLL data model, where each DLL gets a separate copy of its data space in each process that attaches to the DLL. To add to the complexity, data allocated on the heap in a Win32s DLL is in fact process specific (at least as far as ownership goes). Consider the following data and code:

```
static CString strGlobal; // at file scope

__declspec(dllexport)
void SetGlobalString(LPCTSTR lpsz)
{
   strGlobal = lpsz;
}

__declspec(dllexport)
void GetGlobalString(LPCTSTR lpsz, int cb)
{
   lstrcpyn(lpsz, strGlobal, cb);
}
```

Consider what happens if the above code is in located in a DLL and that DLL is loaded by two processes A and B (it could, in fact, be two instances of the same application). A calls `SetGlobalString("Hello from A")`. As a result, memory is allocated for

the **CString** data in the context of process A. Keep in mind that the **CString** itself is global and is visible to both A and B. Now B calls `GetGlobalString(sz, sizeof(sz))`. B will be able to see the data that A set. This is because Win32s offers no protection between processes like Win32 does. That is the first problem; in many cases it is not desirable to have one application affect global data that is considered to be owned by a different application.

There are additional problems as well. Let's say that A now exits. When A exits, the memory used by the '`strGlobal`' string is made available for the system — that is, all memory allocated by process A is freed automatically by the operating system. It is not freed because the **CString** destructor is being called; it hasn't been called yet. It is freed simply because the application which allocated it has left the scene. Now if B called `GetGlobalString(sz, sizeof(sz))`, it may not get valid data. Some other application may have used that memory for something else.

Clearly a problem exists. MFC 3.x used a technique called thread-local storage (TLS). MFC 3.x would allocate a TLS index that under Win32s really acts as a process-local storage index, even though it is not called that and then would reference all data based on that TLS index. This is similar to the TLS index that was used to store thread-local data on Win32 (see below for more information on that subject). This caused every MFC DLL to utilize at least two TLS indices per process. When you account for loading many OLE Control DLLs (OCXs), you quickly run out of TLS indices (there are only 64 available). In addition, MFC had to place all this data in one place, in a single structure. It was not very extensible and was not ideal with regard to its use of TLS indices.

MFC 4.x addresses this with a set of class templates you can "wrap" around the data that should be process local. For example, the problem mentioned above could be fixed by writing:

```
struct CMyGlobalData : public CNoTrackObject
{
   CString strGlobal;
};
CProcessLocal<CMyGlobalData> globalData;

__declspec(dllexport)
void SetGlobalString(LPCTSTR lpsz)
{
   globalData->strGlobal = lpsz;
}

__declspec(dllexport)
void GetGlobalString(LPCTSTR lpsz, int cb)
{
   lstrcpyn(lpsz, globalData->strGlobal, cb);
}
```

MFC implements this in two steps. First, there is a layer on top of the Win32 **Tls*** APIs (**TlsAlloc**, **TlsSetValue**, **TlsGetValue**, etc.) which use only two TLS indexes per process, no matter how many DLLs you have. Second, the `CProcessLocal` template is provided to access this data. It overrides operator-> which is what allows the intuitive syntax you see above. All objects that are wrapped by `CProcessLocal` must be derived from `CNoTrackObject`. `CNoTrackObject` provides a lower-level allocator (**LocalAlloc**/**LocalFree**) and a virtual destructor such that MFC can automatically destroy the process local objects when the process is terminated. Such objects can have a custom destructor if additional cleanup is required. The above example doesn't require one, since the compiler will generate a default destructor to destroy the embedded **CString** object.

There are other interesting advantages to this approach. Not only are all `CProcessLocal` objects destroyed automatically, they are not constructed until they are needed. `CProcessLocal::operator->` will instantiate the associated object the first time it is called, and no sooner. In the example above, that means that the '`strGlobal`' string will not be constructed until the first time **SetGlobalString** or **GetGlobalString** is called. In some instances, this can help decrease DLL startup time.

Thread Local Data

Similar to process local data, thread local data is used when the data must be local to a given thread. That is, you need a separate instance of the data for each thread that accesses that data. This can many times be used in lieu of extensive synchronization mechanisms. If the data does not need to be shared by multiple threads, such mechanisms can be expensive and unnecessary. Suppose we had a **CString** object (much like the sample above). We can make it thread local by wrapping it with a `CThreadLocal` template:

```
struct CMyThreadData : public CNoTrackObject
{
   CString strThread;
};
CThreadLocal<CMyThreadData> threadData;
```

```
void MakeRandomString()
{
   // a kind of card shuffle (not a great one)
   CString& str = threadData->strThread;
   str.Empty();
   while (str.GetLength() != 52)
   {
      TCHAR ch = rand() % 52 + 1;
      if (str.Find(ch) < 0)
         str += ch; // not found, add it
   }
}
```

If `MakeRandomString` was called from two different threads, each would "shuffle" the string in different ways without interfering with the other. This is because there is actually a `strThread` instance per thread instead of just one global instance.

Note how a reference is used to capture the **CString** address once instead of once per loop iteration. The loop code could have been written with `threadData->strThread` everywhere '`str`' is used, but the code would be much slower in execution. It is best to cache a reference to the data when such references occur in loops.

The `CThreadLocal` class template uses the same mechanisms that `CProcessLocal` does and the same implementation techniques.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN059: Using MFC MBCS/Unicode Conversion Macros

Note

The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes how to use the macros for MBCS/Unicode conversion, which are defined in AFXPRIV.H. These macros are most useful if your application deals directly with the OLE API or for some reason, often needs to convert between Unicode and MBCS.

Overview

In MFC 3.x, a special DLL was used (MFCANS32.DLL) to automatically convert between Unicode and MBCS when OLE interfaces were called. This DLL was an almost transparent layer that allowed OLE applications to be written as if the OLE APIs and interfaces were MBCS, even though they are always Unicode (except on the Macintosh). While this layer was convenient and allowed applications to be quickly ported from Win16 to Win32 (MFC, Microsoft Word, Microsoft Excel, and VBA, are just some of the Microsoft applications that used this technology), it had a sometimes significant performance hit. For this reason, MFC 4.x does not use this DLL and instead talks directly to the Unicode OLE interfaces. To do this, MFC needs to convert to Unicode to MBCS when making a call to an OLE interface, and often needs to convert to MBCS from Unicode when implementing an OLE interface. In order to handle this efficiently and easily, a number of macros were created to make this conversion easier.

One of the biggest hurdles of creating such a set of macros is memory allocation. Because the strings cannot be converted in place, new memory to hold the converted results must be allocated. This could have been done with code similar to the following:

```
// we want to convert an MBCS string in lpszA
int nLen = MultiByteToWideChar(CP_ACP, 0,lpszA, -1, NULL, NULL);
LPWSTR lpszW = new WCHAR[nLen];
MultiByteToWideChar(CP_ACP, 0,
   lpszA, -1, lpszW, nLen);
// use it to call OLE here
pI->SomeFunctionThatNeedsUnicode(lpszW);
// free the string
delete[] lpszW;
```

This approach as a number of problems. The main problem is that it is a lot of code to write, test, and debug. Something that was a simple function call, is now much more complex. In addition, there is a significant runtime overhead in doing so. Memory has to be allocated on the heap and freed each time a conversion is done. Finally, the code above would need to have appropriate `#ifdefs` added for Unicode and Macintosh builds (which don't require this conversion to take place).

The solution we came up with is to create some macros which 1) mask the difference between the various platforms, and 2) use an efficient memory allocation scheme, and 3) are easy to insert into the existing source code. Here is an example of one of the definitions:

```
#define A2W(lpa) (\
    ((LPCSTR)lpa == NULL) ? NULL : (\
          _convert = (strlen(lpa)+1),\
        AfxA2WHelper((LPWSTR) alloca(_convert*2),
      lpa, _convert)\
    )\
)
```

Using this macro instead of the code above and things are much simpler:

```
// use it to call OLE here
USES_CONVERSION;
```

```
pI->SomeFunctionThatNeedsUnicode(T2OLE(lpszA));
```

There are extra calls where conversion is necessary, but using the macros is simple and effective.

The implementation of each macro uses the _alloca() function to allocate memory from the stack instead of the heap. Allocating memory from the stack is much faster than allocating memory on the heap, and the memory is automatically freed when the function is exited. In addition, the macros avoid calling **MultiByteToWideChar** (or **WideCharToMultiByte**) more than one time. This is done by allocating a little bit more memory than is necessary. We know that an MBC will convert into at most one **WCHAR** and that for each **WCHAR** we will have a maximum of two MBC bytes. By allocating a little more than necessary, but always enough to handle the conversion the second call second call to the conversion function is avoided. The call to the helper function **AfxA2Whelper** reduces the number of argument pushes that must be done in order to perform the conversion (this results in smaller code, than if it called **MultiByteToWideChar** directly).

In order to for the macros to have space to store the a temporary length, it is necessary to declare a local variable called _convert that does this in each function that uses the conversion macros. This is done by invoking the **USES_CONVERSION** macro as seen above in the example.

There are both generic conversion macros and OLE specific macros. These two different macro sets are discussed below. All of the macros reside in AFXPRIV.H.

Generic Conversion Macros

The generic conversion macros form the underlying mechanism. The macro example and implementation shown in the previous section, A2W, is one such "generic" macro. It is not related to OLE specifically. The set of generic macros is listed below:

```
A2CW      (LPCSTR) -> (LPCWSTR)
A2W      (LPCSTR) -> (LPWSTR)
W2CA      (LPCWSTR) -> (LPCSTR)
W2A      (LPCWSTR) -> (LPSTR)
```

Besides doing text conversions, there are also macros and helper functions for converting **TEXTMETRIC**, **DEVMODE**, **BSTR**, and OLE allocated strings. These macros are beyond the scope of this discussion – refer to AFXPRIV.H for more information on those macros.

OLE Conversion Macros

The OLE conversion macros are designed specifically for handling functions that expect **OLESTR** characters. If you examine the OLE headers, you will see many references to **LPCOLESTR** and **OLECHAR**. These types are used to refer to the type of characters used in OLE interfaces in a way that is not specific to the platform. **OLECHAR** maps to **char** in Win16 and Macintosh platforms and **WCHAR** in Win32.

In order to keep the number of **#ifdef** directives in the MFC code to a minimum we have a similar macro for each conversion that where OLE strings are involved. The following macros are the most commonly used:

```
T2COLE   (LPCTSTR) -> (LPCOLESTR)
T2OLE   (LPCTSTR) -> (LPOLESTR)
OLE2CT   (LPCOLESTR) -> (LPCTSTR)
OLE2T   (LPCOLESTR) -> (LPCSTR)
```

Again, there are similar macros for doing **TEXTMETRIC**, **DEVMODE**, **BSTR**, and OLE allocated strings. Refer to AFXPRIV.H for more information.

Other Considerations

Do not use the macros in a tight loop. For example, you do not want to write the following kind of code:

```
void BadIterateCode(LPCTSTR lpsz)
{
   USES_CONVERSION;
   for (int ii = 0; ii < 10000; ii++)
      pI->SomeMethod(ii, T2COLE(lpsz));
}
```

The code above could result in allocating megabytes of memory on the stack depending on what the contents of the string

`lpsz` is! It also takes time to convert the string for each iteration of the loop. Instead, move such constant conversions out of the loop:

```
void MuchBetterIterateCode(LPCTSTR lpsz)
{
   USES_CONVERSION;
   LPCOLESTR lpszT = T2COLE(lpsz);
   for (int ii = 0; ii < 10000; ii++)
      pI->SomeMethod(ii, lpszT);
}
```

If the string is not constant, then encapsulate the method call into a function. This will allow the conversion buffer to be freed each time. For example:

```
void CallSomeMethod(int ii, LPCTSTR lpsz)
{
   USES_CONVERSION;
   pI->SomeMethod(ii, T2COLE(lpsz));
}

void MuchBetterIterateCode2(LPCTSTR* lpszArray)
{
   for (int ii = 0; ii < 10000; ii++)
      CallSomeMethod(ii, lpszArray[ii]);
}
```

Never return the result of one of the macros, unless the return value implies making a copy of the data before the return. For example, this code is bad:

```
LPTSTR BadConvert(ISomeInterface* pI)
{
   USES_CONVERSION;
   LPOLESTR lpsz = NULL;
   pI->GetFileName(&lpsz);
   LPTSTR lpszT = OLE2T(lpsz);
   CoMemFree(lpsz);
   return lpszT; // bad! returning alloca memory
}
```

The code above could be fixed by changing the return value to something that copies the value:

```
CString BetterConvert(ISomeInterface* pI)
{
   USES_CONVERSION;
   LPOLESTR lpsz = NULL;
   pI->GetFileName(&lpsz);
   LPTSTR lpszT = OLE2T(lpsz);
   CoMemFree(lpsz);
   return lpszT; // CString makes copy
}
```

The macros are easy to use and easy to insert into your code, but as you can tell from the caveats above, you need to be careful when using them.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN060: The New Windows Common Controls

Technical Note 60, describing the new Windows common controls and how to use them, has been incorporated into Controls.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN061: ON_NOTIFY and WM_NOTIFY Messages

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This technical note provides background information on the new **WM_NOTIFY** message and describes the recommended (and most common) way of handling **WM_NOTIFY** messages in your MFC application.

Notification Messages in Windows 3.x

In Windows 3.x, controls notify their parents of events such as mouse clicks, changes in content and selection, and control background painting by sending a message to the parent. Simple notifications are sent as special **WM_COMMAND** messages, with the notification code (such as **BN_CLICKED**) and control ID packed into *wParam* and the control's handle in *lParam*. Note that since *wParam* and *lParam* are full, there is no way to pass any additional data — these messages can be only simple notification. For instance, in the **BN_CLICKED** notification, there's no way to send information about the location of the mouse cursor when the button was clicked.

When controls in Windows 3.x need to send a notification message that includes additional data, they use a variety of special-purpose messages, including **WM_CTLCOLOR**, **WM_VSCROLL**, **WM_HSCROLL**, **WM_DRAWITEM**, **WM_MEASUREITEM**, **WM_COMPAREITEM**, **WM_DELETEITEM**, **WM_CHARTOITEM**, **WM_VKEYTOITEM**, and so on. These messages can be reflected back to the control that sent them. For more information, see TN062: Message Reflection for Windows Controls.

Notification Messages in Win32

For controls that existed in Windows 3.1, the Win32 API uses most of the notification messages that were used in Windows 3.x. However, Win32 also adds a number of sophisticated, complex controls to those supported in Windows 3.x. Frequently, these controls need to send additional data with their notification messages. Rather than adding a new **WM_*** message for each new notification that needs additional data, the designers of the Win32 API chose to add just one message, **WM_NOTIFY**, which can pass any amount of additional data in a standardized fashion.

WM_NOTIFY messages contain the ID of the control sending the message in *wParam* and a pointer to a structure in *lParam*. This structure is either an **NMHDR** structure or some larger structure that has an **NMHDR** structure as its first member. Note that since the **NMHDR** member is first, a pointer to this structure can be used as either a pointer to an **NMHDR** or as a pointer to the larger structure depending on how you cast it.

In most cases, the pointer will point to a larger structure and you'll need to cast it when you use it. In only a few notifications, such as the common notifications (whose names start with **NM_**) and the tool tip control's **TTN_SHOW** and **TTN_POP** notifications, is an **NMHDR** structure actually used.

The **NMHDR** structure or initial member contains the handle and ID of the control sending the message and the notification code (such as **TTN_SHOW**). The format of the **NMHDR** structure is shown below:

```
typedef struct tagNMHDR {
    HWND hwndFrom;
    UINT idFrom;
    UINT code;
} NMHDR;
```

For a **TTN_SHOW** message, the **code** member would be set to **TTN_SHOW**.

Most notifications pass a pointer to a larger structure that contains an **NMHDR** structure as its first member. For instance, consider the structure used by the list view control's **LVN_KEYDOWN** notification message, which is sent when a key is pressed in a list view control. The pointer points to an **LV_KEYDOWN** structure, which is defined as shown below:

```
typedef struct tagLV_KEYDOWN {
    NMHDR hdr;
    WORD wVKey;
    UINT flags;
```

```
} LV_KEYDOWN;
```

Note that since the **NMHDR** member is first in this structure, the pointer you're passed in the notification message can be cast to either a pointer to an **NMHDR** or a pointer to an **LV_KEYDOWN**.

Notifications Common to All New Windows Controls

Some notifications are common to all of the new Windows controls. These notifications pass a pointer to an **NMHDR** structure.

Notification code	Sent because
NM_CLICK	User clicked left mouse button in the control
NM_DBLCLK	User double-clicked left mouse button in the control
NM_RCLICK	User clicked right mouse button in the control
NM_RDBLCLK	User double-clicked right mouse button in the control
NM_RETURN	User pressed the ENTER key while control has input focus
NM_SETFOCUS	Control has been given input focus
NM_KILLFOCUS	Control has lost input focus
NM_OUTOFMEMORY	Control could not complete an operation because there was not enough memory available

ON_NOTIFY: Handling WM_NOTIFY Messages in MFC Applications

The function **CWnd::OnNotify** handles notification messages. Its default implementation checks the message map for notification handlers to call. In general, you do not override **OnNotify**. Instead, you provide a handler function and add a message-map entry for that handler to the message map of your owner window's class.

ClassWizard, via the ClassWizard property sheet, can create the **ON_NOTIFY** message-map entry and provide you with a skeleton handler function. For more information on using ClassWizard to make this easier, see Mapping Messages to Functions.

The **ON_NOTIFY** message-map macro has the following syntax:

```
ON_NOTIFY( wNotifyCode, id, memberFxn )
```

where the italicized parameters are replaced with:

wNotifyCode

The code for the notification message to be handled, such as **LVN_KEYDOWN**.

id

The child identifier of the control for which the notification is sent.

memberFxn

The member function to be called when this notification is sent.

Your member function must be declared with the following prototype:

```
afx_msg void memberFxn( NMHDR * pNotifyStruct, LRESULT * result );
```

Remarks

where the italicized parameters are:

pNotifyStruct

A pointer to the notification structure, as described in the section above.

result

A pointer to the result code you'll set before you return.

Example

To specify that you want the member function `OnKeydownList1` to handle **LVN_KEYDOWN** messages from the **CListCtrl** whose ID is `IDC_LIST1`, you would use ClassWizard to add the following to your message map:

```
ON_NOTIFY( LVN_KEYDOWN, IDC_LIST1, OnKeydownList1 )
```

In the example above, the function provided by ClassWizard is:

```
void CMessageReflectionDlg::OnKeydownList1(NMHDR* pNMHDR, LRESULT* pResult)
{
   LV_KEYDOWN* pLVKeyDow = (LV_KEYDOWN*)pNMHDR;
   // TODO: Add your control notification handler
   //       code here

   *pResult = 0;
}
```

Note that ClassWizard provides a pointer of the proper type automatically. You can access the notification structure through either `pNMHDR` or `pLVKeyDow`.

ON_NOTIFY_RANGE

If you need to process the same **WM_NOTIFY** message for a set of controls, you can use **ON_NOTIFY_RANGE** rather than **ON_NOTIFY**. For instance, you may have a set of buttons for which you want to perform the same action for a certain notification message.

When you use **ON_NOTIFY_RANGE**, you specify a contiguous range of child identifiers for which to handle the notification message by specifying the beginning and ending child identifiers of the range.

ClassWizard does not handle **ON_NOTIFY_RANGE**; to use it, you need to edit your message map yourself.

The message-map entry and function prototype for **ON_NOTIFY_RANGE** are as follows:

```
ON_NOTIFY_RANGE(
wNotifyCode
,
id
,
idLast
,
memberFxn
 )
```

where the italicized parameters are replaced with:

wNotifyCode

The code for the notification message to be handled, such as **LVN_KEYDOWN**.

id

The first identifier in the contiguous range of identifiers.

idLast

The last identifier in the contiguous range of identifiers.

memberFxn

The member function to be called when this notification is sent.

Your member function must be declared with the following prototype:

```
afx_msg void memberFxn( UINT id, NMHDR * pNotifyStruct, LRESULT * result );
```

Remarks

where the italicized parameters are:

id

The child identifier of the control that sent the notification.

pNotifyStruct

A pointer to the notification structure, as described above.

result

A pointer to the result code you'll set before you return.

ON_NOTIFY_EX, ON_NOTIFY_EX_RANGE

If you want more than one object in the notification routing to handle a message, you can use **ON_NOTIFY_EX** (or **ON_NOTIFY_EX_RANGE**) rather than **ON_NOTIFY** (or **ON_NOTIFY_RANGE**). The only difference between the **EX** version and the regular version is that the member function called for the **EX** version returns a **BOOL** that indicates whether or not message processing should continue. Returning **FALSE** from this function allows you to process the same message in more than one object.

ClassWizard does not handle **ON_NOTIFY_EX** or **ON_NOTIFY_EX_RANGE**; if you want to use either of them, you need to edit your message map yourself.

The message-map entry and function prototype for **ON_NOTIFY_EX** and **ON_NOTIFY_EX_RANGE** are as follows. The meanings of the parameters are the same as for the non-**EX** versions.

```
        ON_NOTIFY_EX( nCode, id, memberFxn )
ON_NOTIFY_EX_RANGE( wNotifyCode, id, idLast, memberFxn )
```

The prototype for both of the above is the same:

```
afx_msg BOOL memberFxn( UINT id, NMHDR * pNotifyStruct, LRESULT * result );
```

Remarks

In both cases, *id* holds the child identifier of the control that sent the notification.

Your function must return **TRUE** if the notification message has been completely handled or **FALSE** if other objects in the command routing should have a chance to handle the message.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN062: Message Reflection for Windows Controls

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This technical note describes message reflection, a new feature in MFC 4.0. It also contains directions for creating a simple reusable control that uses message reflection.

This technical note does not discuss message reflection as it applies to ActiveX controls (formerly called OLE controls). Please see the article ActiveX Controls: Subclassing a Windows Control.

What Is Message Reflection?

Windows controls frequently send notification messages to their parent windows. For instance, many controls send a control color notification message (**WM_CTLCOLOR** or one of its variants) to their parent to allow the parent to supply a brush for painting the background of the control.

In Windows and in MFC before version 4.0, the parent window, often a dialog box, is responsible for handling these messages. This means that the code for handling the message needs to be in the parent window's class and that it has to be duplicated in every class that needs to handle that message. In the case above, every dialog box that wanted controls with custom backgrounds would have to handle the control color notification message. It would be much easier to reuse code if a control class could be written that would handle its own background color.

In MFC 4.0, the old mechanism still works — parent windows can handle notification messages. In addition, however, MFC 4.0 facilitates reuse by providing a feature called "message reflection" that allows these notification messages to be handled in either the child control window or the parent window, or in both. In the control background color example, you can now write a control class that sets its own background color by handling the reflected **WM_CTLCOLOR** message — all without relying on the parent. (Note that since message reflection is implemented by MFC, not by Windows, the parent window class must be derived from **CWnd** for message reflection to work.)

Older versions of MFC did something similar to message reflection by providing virtual functions for a few messages, such as messages for owner-drawn list boxes (**WM_DRAWITEM**, and so on). The new message reflection mechanism is generalized and consistent.

Message reflection is backward compatible with code written for versions of MFC before 4.0.

If you have supplied a handler for a specific message, or for a range of messages, in your parent window's class, it will override reflected message handlers for the same message provided you don't call the base class handler function in your own handler. For example, if you handle **WM_CTLCOLOR** in your dialog box class, your handling will override any reflected message handlers.

If, in your parent window class, you supply a handler for a specific **WM_NOTIFY** message or a range of **WM_NOTIFY** messages, your handler will be called only if the child control sending those messages does not have a reflected message handler through **ON_NOTIFY_REFLECT()**. If you use **ON_NOTIFY_REFLECT_EX()** in your message map, your message handler may or may not allow the parent window to handle the message. If the handler returns **FALSE**, the message will be handled by the parent as well, while a call that returns **TRUE** does not allow the parent to handle it. Note that the reflected message is handled before the notification message.

When a **WM_NOTIFY** message is sent, the control is offered the first chance to handle it. If any other reflected message is sent, the parent window has the first chance to handle it and the control will receive the reflected message. To do so, it will need a handler function and an appropriate entry in the control's class message map.

The message-map macro for reflected messages is slightly different than for regular notifications: it has **_REFLECT** appended to its usual name. For instance, to handle a **WM_NOTIFY** message in the parent, you use the macro **ON_NOTIFY** in the parent's message map. To handle the reflected message in the child control, use the **ON_NOTIFY_REFLECT** macro in the child control's message map. In some cases, the parameters are different, as well. Note that ClassWizard can usually add the message-map entries for you and provide skeleton function implementations with correct parameters.

See TN061: ON_NOTIFY and WM_NOTIFY Messages for information on the new **WM_NOTIFY** message.

Message-Map Entries and Handler Function Prototypes for Reflected Messages

To handle a reflected control notification message, use the message-map macros and function prototypes listed in the table below.

ClassWizard can usually add these message-map entries for you and provide skeleton function implementations. See Defining a Message Handler for a Reflected Message for information about how to define handlers for reflected messages.

To convert from the message name to the reflected macro name, prepend **ON_** and append **_REFLECT**. For example, **WM_CTLCOLOR** becomes **ON_WM_CTLCOLOR_REFLECT**. (To see which messages can be reflected, do the opposite conversion on the macro entries in the table below.)

The three exceptions to the rule above are as follows:

- The macro for **WM_COMMAND** notifications is **ON_CONTROL_REFLECT**.
- The macro for **WM_NOTIFY** reflections is **ON_NOTIFY_REFLECT**.
- The macro for **ON_UPDATE_COMMAND_UI** reflections is **ON_UPDATE_COMMAND_UI_REFLECT**.

In each of the above special cases, you must specify the name of the handler member function. In the other cases, you must use the standard name for your handler function.

The meanings of the parameters and return values of the functions are documented under either the function name or the function name with **On** prepended. For instance, **CtlColor** is documented in **OnCtlColor**. Several reflected message handlers need fewer parameters than the similar handlers in a parent window. Just match the names in the table below with the names of the formal parameters in the documentation.

Map entry	Function prototype
ON_CONTROL_REFLECT(*wNotifyCode* **,** *memberFxn* **)**	**afx_msg void** *memberFxn* **( );**
ON_NOTIFY_REFLECT(*wNotifyCode* **,** *memberFxn* **)**	**afx_msg void** *memberFxn* **(NMHDR *** *pNotifyStruct* **, LRESULT*** *result* **);**
ON_UPDATE_COMMAND_UI_REFLECT(*memberFxn* **)**	**afx_msg void** *memberFxn* **(CCmdUI*** *pCmdUI* **);**
ON_WM_CTLCOLOR_REFLECT( )	**afx_msg HBRUSH CtlColor (CDC*** *pDC* **, UINT** *nCtlColor* **);**
ON_WM_DRAWITEM_REFLECT( )	**afx_msg void DrawItem (LPDRAWITEMSTRUCT** *lpDrawItemStruct* **);**
ON_WM_MEASUREITEM_REFLECT( )	**afx_msg void MeasureItem (LPMEASUREITEMSTRUCT** *lpMeasureItemStruct* **);**
ON_WM_DELETEITEM_REFLECT( )	**afx_msg void DeleteItem (LPDELETEITEMSTRUCT** *lpDeleteItemStruct* **);**
ON_WM_COMPAREITEM_REFLECT( )	**afx_msg int CompareItem (LPCOMPAREITEMSTRUCT** *lpCompareItemStruct* **);**
ON_WM_CHARTOITEM_REFLECT( )	**afx_msg int CharToItem (UINT** *nKey* **, UINT** *nIndex* **);**
ON_WM_VKEYTOITEM_REFLECT( )	**afx_msg int VKeyToItem (UINT** *nKey* **, UINT** *nIndex* **);**
ON_WM_HSCROLL_REFLECT( )	**afx_msg void HScroll (UINT** *nSBCode* **, UINT** *nPos* **);**
ON_WM_VSCROLL_REFLECT( )	**afx_msg void VScroll (UINT** *nSBCode* **, UINT** *nPos* **);**
ON_WM_PARENTNOTIFY_REFLECT( )	**afx_msg void ParentNotify (UINT** *message* **, LPARAM** *lParam* **);**

The **ON_NOTIFY_REFLECT** and **ON_CONTROL_REFLECT** macros have variations that allow more than one object (such as the control and its parent) to handle a given message.

Map entry	Function prototype
ON_NOTIFY_REFLECT_EX(*wNotifyCode* **,** *memberFxn* **)**	**afx_msg BOOL** *memberFxn* **(NMHDR *** *pNotifyStruct* **, LRESULT*** *result* **);**
ON_CONTROL_REFLECT_EX(*wNotifyCode* **,** *memberFxn* **)**	**afx_msg BOOL** *memberFxn* **( );**

Handling Reflected Messages: An Example of a Reusable control

This simple example creates a reusable control called `CYellowEdit`. The control works the same as a regular edit control except that it displays black text on a yellow background. It would be easy to add member functions that would allow the `CYellowEdit` control to display different colors.

To try the example that creates a reusable control

1. Create a new dialog box in an existing application. For more information, see the dialog editor topic.

   You must have an application in which to develop the reusable control. If you don't have an existing application to use, create a dialog-based application using AppWizard.

2. With your project loaded into Visual C++, use ClassWizard to create a new class called `CYellowEdit` based on **CEdit**.

3. Add three member variables to your `CYellowEdit` class. The first two will be **COLORREF** variables to hold the text color and the background color. The third will be a **CBrush** object that will hold the brush for painting the background. The **CBrush** object allows you to create the brush once, merely referencing it after that, and to destroy the brush automatically when the `CYellowEdit` control is destroyed.

4. Initialize the member variables by writing the constructor as follows:

```
CYellowEdit::CYellowEdit()
{
   m_clrText = RGB( 0, 0, 0 );
   m_clrBkgnd = RGB( 255, 255, 0 );
   m_brBkgnd.CreateSolidBrush( m_clrBkgnd );
}
```

5. Using ClassWizard, add a handler for the reflected **WM_CTLCOLOR** message to your `CYellowEdit` class. Note that the equal sign in front of the message name in the list of messages you can handle indicates that the message is reflected. This is described in Defining a Message Handler for a Reflected Message.

   ClassWizard adds the following message-map macro and skeleton function for you:

```
ON_WM_CTLCOLOR_REFLECT()

// Note: other code will be in between....

HBRUSH CYellowEdit::CtlColor(CDC* pDC, UINT nCtlColor)
{
   // TODO: Change any attributes of the DC here

   // TODO: Return a non-NULL brush if the
   //   parent's handler should not be called
   return NULL;
}
```

6. Replace the body of the function with the following code. The code specifies the text color, the text background color, and the background color for rest of the control.

```
    pDC->SetTextColor( m_clrText );   // text
    pDC->SetBkColor( m_clrBkgnd );   // text bkgnd
    return m_brBkgnd;            // ctl bkgnd
```

7. Create an edit control in your dialog box, then attach it to a member variable by double-clicking the edit control while holding a control key down. In the Add Member Variable dialog box, finish the variable name and choose "Control" for the category, then "CYellowEdit" for the variable type. Don't forget to set the tab order in the dialog box. Also, be sure to include the header file for the `CYellowEdit` control in your dialog box's header file.

8. Build and run your application. The edit control will have a yellow background.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN063: Debugging Internet Extension DLLs

This information is obsolete and has been removed.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN064: Apartment-Model Threading in ActiveX Controls

Note

The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This technical note explains how to enable apartment-model threading in an ActiveX control. Note that apartment-model threading is only supported in Visual C++ versions 4.2 or later.

What Is Apartment-Model Threading?

The apartment model is an approach to supporting embedded objects, such as ActiveX controls, within a multithreaded container application. Although the application may have multiple threads, each instance of an embedded object will be assigned to one "apartment," which will execute on only one thread. In other words, all calls into an instance of a control will happen on the same thread.

However, different instances of the same type of control may be assigned to different apartments. So, if multiple instances of a control share any data in common (for example, static or global data), then access to this shared data will need to be protected by a synchronization object, such as a critical section.

For complete details on the apartment threading model, please see Processes and Threads in the *OLE Programmer's Reference*.

Why Support Apartment-Model Threading?

Controls that support apartment-model threading can be used in multithreaded container applications that also support the apartment model. If you do not enable apartment-model threading, you will limit the potential set of containers in which your control could be used.

Enabling apartment-model threading is easy for most controls, particularly if they have little or no shared data.

Protecting Shared Data

If your control uses shared data, such as a static member variable, access to that data should be protected with a critical section to prevent more than one thread from modifying the data at the same time. To set up a critical section for this purpose, declare a static member variable of class **CCriticalSection** in your control's class. Use the **Lock** and **Unlock** member functions of this critical section object wherever your code accesses the shared data.

Consider, for example, a control class that needs to maintain a string that is shared by all instances. This string can be maintained in a static member variable and protected by a critical section. The control's class declaration would contain the following:

```
class CSampleCtrl : public COleControl
{
    ...
    static CString _strShared;
    static CCriticalSection _critSect;
};
```

The implementation for the class would include definitions for these variables:

```
int CString CSampleCtrl::_strShared;
CCriticalSection CSampleCtrl::_critSect;
```

Access to the `_strShared` static member can then be protected by the critical section:

```
void CSampleCtrl::SomeMethod()
{
    _critSect.Lock();
    if (_strShared.Empty())
```

```
        _strShared = "<text>";
    _critSect.Unlock();
    ...
}
```

Registering an Apartment-Model-Aware Control

Controls that support apartment-model threading should indicate this capability in the registry, by adding the named value "ThreadingModel" with a value of "Apartment" in their class ID registry entry under the *class id***InprocServer32** key. To cause this key to be automatically registered for your control, pass the `afxRegApartmentThreading` flag in the sixth parameter to **AfxOleRegisterControlClass**:

```
BOOL CSampleCtrl::CSampleCtrlFactory::UpdateRegistry(BOOL bRegister)
{
    if (bRegister)
        return AfxOleRegisterControlClass(
            AfxGetInstanceHandle(),
            m_clsid,
            m_lpszProgID,
            IDS_SAMPLE,
            IDB_SAMPLE,
            afxRegApartmentThreading,
            _dwSampleOleMisc,
            _tlid,
            _wVerMajor,
            _wVerMinor);
    else
        return AfxOleUnregisterClass(m_clsid, m_lpszProgID);
}
```

If your control project was generated by ControlWizard in Visual C++ version 4.1 or later, this flag will already be present in your code. No changes are necessary to register the threading model.

If your project was generated by an earlier version of ControlWizard, your existing code will have a Boolean value as the sixth parameter. If the existing parameter is TRUE, change it to `afxRegInsertable | afxRegApartmentThreading`. If the existing parameter is FALSE, change it to `afxRegApartmentThreading`.

If your control does not follow the rules for apartment-model threading, you must not pass `afxRegApartmentThreading` in this parameter.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN065: Dual-Interface Support for OLE Automation Servers

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note discusses how to add dual-interface support to an MFC-based OLE Automation server application. The ACDUAL sample illustrates dual-interface support, and the example code in this note is taken from ACDUAL. The macros described in this note, such as `DECLARE_DUAL_ERRORINFO`, `DUAL_ERRORINFO_PART`, and `IMPLEMENT_DUAL_ERRORINFO`, are part of the ACDUAL sample and can be found in MFCDUAL.H.

Dual Interfaces

Although OLE Automation allows you to implement an **IDispatch** interface, a VTBL interface, or a dual interface (which encompasses both), Microsoft strongly recommends that you implement dual interfaces for all exposed OLE Automation objects. Dual interfaces have significant advantages over **IDispatch**-only or VTBL-only interfaces:

- Binding can take place at compile time through the VTBL interface, or at run time through **IDispatch**.
- OLE Automation controllers that can use the VTBL interface may benefit from improved performance.
- Existing OLE Automation controllers that use the **IDispatch** interface will continue to work.
- The VTBL interface is easier to call from C++.
- Dual interfaces are required for compatibility with Visual Basic object support features.

Adding Dual-Interface Support to a CCmdTarget-Based Class

A dual interface is really just a custom interface derived from **IDispatch**. The most straightforward way to implement dual-interface support in a **CCmdTarget**-based class is to first implement the normal dispatch interface on your class using MFC and ClassWizard, then add the custom interface later. For the most part, your custom interface implementation will simply delegate back to the MFC **IDispatch** implementation.

First, modify the ODL file for your server to define dual interfaces for your objects. To define a dual interface, you must use an interface statement, instead of the `DISPINTERFACE` statement that the Visual C++ wizards generate. Rather than removing the existing `DISPINTERFACE` statement, add a new interface statement. By retaining the `DISPINTERFACE` form, you can continue to use ClassWizard to add properties and methods to your object, but you must add the equivalent properties and methods to your interface statement.

An interface statement for a dual interface must have the **OLEAUTOMATION** and **DUAL** attributes, and the interface must be derived from **IDispatch**. You can use the GUIDGEN sample to create a **IID** for the dual interface:

```
[ uuid(0BDD0E81-0DD7-11cf-BBA8-444553540000), // IID_IDualAClick
   oleautomation,
   dual
]
interface IDualAClick : IDispatch
  {
  };
```

Once you have the interface statement in place, start adding entries for the methods and properties. For dual interfaces, you need to rearrange the parameter lists so that your methods and property accessor functions in the dual interface return an **HRESULT** and pass their return values as parameters with the attributes `[retval,out]`. Remember that for properties, you will need to add both a read (`propget`) and write (`propput`) access function with the same id. For example:

```
[propput, id(1)] HRESULT text([in] BSTR newText);
[propget, id(1)] HRESULT text([out, retval] BSTR* retval);
```

After your methods and properties are defined, you need to add a reference to the interface statement in your coclass statement. For example:

```
[ uuid(4B115281-32F0-11cf-AC85-444553540000) ]
coclass Document
{
   dispinterface IAClick;
   [default] interface IDualAClick;
};
```

Once your ODL file has been updated, use MFC's interface map mechanism to define an implementation class for the dual interface in your object class and make the corresponding entries in MFC's **QueryInterface** mechanism. You need one entry in the `INTERFACE_PART` block for each entry in the interface statement of the ODL, plus the entries for a dispatch interface. Each ODL entry with the **propput** attribute needs a function named `put_propertyname`. Each entry with the **propget** attribute needs a function named `get_propertyname`.

To define an implementation class for your dual interface, add a `DUAL_INTERFACE_PART` block to your object class definition. For example:

```
BEGIN_DUAL_INTERFACE_PART(DualAClick, IDualAClick)
  STDMETHOD(put_text)(THIS_ BSTR newText);
  STDMETHOD(get_text)(THIS_ BSTR FAR* retval);
  STDMETHOD(put_x)(THIS_ short newX);
  STDMETHOD(get_x)(THIS_ short FAR* retval);
  STDMETHOD(put_y)(THIS_ short newY);
  STDMETHOD(get_y)(THIS_ short FAR* retval);
  STDMETHOD(put_Position)(THIS_ IDualAutoClickPoint FAR* newPosition);
  STDMETHOD(get_Position)(THIS_ IDualAutoClickPoint FAR* FAR* retval);
  STDMETHOD(RefreshWindow)(THIS);
  STDMETHOD(SetAllProps)(THIS_ short x, short y, BSTR text);
  STDMETHOD(ShowWindow)(THIS);
END_DUAL_INTERFACE_PART(DualAClick)
```

To connect the dual interface to MFC's QueryInterface mechanism, add an `INTERFACE_PART` entry to the interface map:

```
BEGIN_INTERFACE_MAP(CAutoClickDoc, CDocument)
  INTERFACE_PART(CAutoClickDoc, DIID_IAClick, Dispatch)
  INTERFACE_PART(CAutoClickDoc, IID_IDualAClick, DualAClick)
END_INTERFACE_MAP()
```

Next, you need to fill in the implementation of the interface. For the most part, you will be able to delegate to the existing MFC **IDispatch** implementation. For example:

```
STDMETHODIMP_(ULONG) CAutoClickDoc::XDualAClick::AddRef()
{
   METHOD_PROLOGUE(CAutoClickDoc, DualAClick)
   return pThis->ExternalAddRef();
}
STDMETHODIMP_(ULONG) CAutoClickDoc::XDualAClick::Release()
{
   METHOD_PROLOGUE(CAutoClickDoc, DualAClick)
   return pThis->ExternalRelease();
}
STDMETHODIMP CAutoClickDoc::XDualAClick::QueryInterface(
             REFIID iid, LPVOID* ppvObj)
{
   METHOD_PROLOGUE(CAutoClickDoc, DualAClick)
   return pThis->ExternalQueryInterface(&iid, ppvObj);
}
STDMETHODIMP CAutoClickDoc::XDualAClick::GetTypeInfoCount(
            UINT FAR* pctinfo)
{
   METHOD_PROLOGUE(CAutoClickDoc, DualAClick)
   LPDISPATCH lpDispatch = pThis->GetIDispatch(FALSE);
```

```
   ASSERT(lpDispatch != NULL);
   return lpDispatch->GetTypeInfoCount(pctinfo);
}
STDMETHODIMP CAutoClickDoc::XDualAClick::GetTypeInfo(
          UINT itinfo, LCID lcid, ITypeInfo FAR* FAR* pptinfo)
{
   METHOD_PROLOGUE(CAutoClickDoc, DualAClick)
   LPDISPATCH lpDispatch = pThis->GetIDispatch(FALSE);
   ASSERT(lpDispatch != NULL);
   return lpDispatch->GetTypeInfo(itinfo, lcid, pptinfo);
}
STDMETHODIMP CAutoClickDoc::XDualAClick::GetIDsOfNames(
       REFIID riid, OLECHAR FAR* FAR* rgszNames, UINT cNames,
       LCID lcid, DISPID FAR* rgdispid)
{
   METHOD_PROLOGUE(CAutoClickDoc, DualAClick)
   LPDISPATCH lpDispatch = pThis->GetIDispatch(FALSE);
   ASSERT(lpDispatch != NULL);
   return lpDispatch->GetIDsOfNames(riid, rgszNames, cNames,
                                    lcid, rgdispid);
}
STDMETHODIMP CAutoClickDoc::XDualAClick::Invoke(
    DISPID dispidMember, REFIID riid, LCID lcid, WORD wFlags,
    DISPPARAMS FAR* pdispparams, VARIANT FAR* pvarResult,
    EXCEPINFO FAR* pexcepinfo, UINT FAR* puArgErr)
{
   METHOD_PROLOGUE(CAutoClickDoc, DualAClick)
   LPDISPATCH lpDispatch = pThis->GetIDispatch(FALSE);
   ASSERT(lpDispatch != NULL);
   return lpDispatch->Invoke(dispidMember, riid, lcid,
                             wFlags, pdispparams, pvarResult,
                             pexcepinfo, puArgErr);
}
```

For your object's methods and property accessor functions, you need to fill in the implementation. Your method and property functions can generally delegate back to the methods generated using ClassWizard. However, if you set up properties to access variables directly, you need to write the code to get/put the value into the variable. For example:

```
STDMETHODIMP CAutoClickDoc::XDualAClick::put_text(BSTR newText)
{
   METHOD_PROLOGUE(CAutoClickDoc, DualAClick)
   // MFC automatically converts from Unicode BSTR to
   // Ansi CString, if necessary...
   pThis->m_str = newText;
   return NOERROR;
}
STDMETHODIMP CAutoClickDoc::XDualAClick::get_text(BSTR* retval)
{
   METHOD_PROLOGUE(CAutoClickDoc, DualAClick)
   // MFC automatically converts from Ansi CString to
   // Unicode BSTR, if necessary...
   pThis->m_str.SetSysString(retval);
   return NOERROR;
}
```

Passing Dual-Interface Pointers

Passing your dual-interface pointer isn't straightforward, especially if you need to call **CCmdTarget::FromIDispatch**. **FromIDispatch** only works on MFC's **IDispatch** pointers. One way to work around this is to query for the original **IDispatch** pointer set up by MFC and pass that pointer to functions that need it. For example:

```
STDMETHODIMP CAutoClickDoc::XDualAClick::put_Position(
      IDualAutoClickPoint FAR* newPosition)
{
   METHOD_PROLOGUE(CAutoClickDoc, DualAClick)
   LPDISPATCH lpDisp = NULL;
```

```
    newPosition->QueryInterface(IID_IDispatch, (LPVOID*)&lpDisp);
    pThis->SetPosition(lpDisp);
    lpDisp->Release();
    return NOERROR;
}
```

Before passing a pointer back through the dual-interface method, you might need to convert it from the MFC **IDispatch** pointer to your dual-interface pointer. For example:

```
STDMETHODIMP CAutoClickDoc::XDualAClick::get_Position(
      IDualAutoClickPoint FAR* FAR* retval)
{
   METHOD_PROLOGUE(CAutoClickDoc, DualAClick)
   LPDISPATCH lpDisp;
   lpDisp = pThis->GetPosition();
   lpDisp->QueryInterface(IID_IDualAutoClickPoint, (LPVOID*)retval);
   return NOERROR;
}
```

Registering the Application's Type Library

AppWizard does not generate code to register an OLE Automation server application's type library with the system. While there are other ways to register the type library, it is convenient to have the application register the type library when it is updating its OLE type information, that is, whenever the application is run stand-alone.

To register the application's type library whenever the application is run stand alone:

- Include AFXCTL.H in your standard includes header file, STDAFX.H, to access the definition of the **AfxOleRegisterTypeLib** function.
- In your application's `InitInstance` function, locate the call to **COleObjectFactory::UpdateRegistryAll**. Following this call, add a call to **AfxOleRegisterTypeLib**, specifying the **LIBID** corresponding to your type library, along with the name of your type library:

  ```
  // When a server application is launched stand-alone, it is a good idea
  // to update the system registry in case it has been damaged.
  m_server.UpdateRegistry(OAT_DISPATCH_OBJECT);
  COleObjectFactory::UpdateRegistryAll();
  // DUAL_SUPPORT_START
  // Make sure the type library is registered or dual interface won't work.
  AfxOleRegisterTypeLib(AfxGetInstanceHandle(), LIBID_ACDual, _T("AutoClik.TLB"));
  // DUAL_SUPPORT_END
  ```

Modifying Project Build Settings to Accommodate Type Library Changes

To modify a project's build settings so that a header file containing **UUID** definitions is generated by MkTypLib whenever the type library is rebuilt:

1. On the **Build** menu, click **Settings**, and then select the ODL file from the file list for each configuration.
2. Click the **OLE Types** tab and specify a filename in the **Output header** filename field. Use a filename that is not already used by your project, because MkTypLib will overwrite any existing file. Click **OK** to close the **Build Settings** dialog box.

To add the **UUID** definitions from the MkTypLib-generated header file to your project:

1. Include the MkTypLib-generated header file in your standard includes header file, STDAFX.H.
2. Create a new file, INITIIDS.CPP, and add it to your project. In this file, include your MkTypLib-generated header file after including OLE2.H and INITGUID.H:

   ```
   // initIIDs.c: defines IIDs for dual interfaces
   // This must not be built with precompiled header.
     #include <ole2.h>
   ```

```
#include <initguid.h>
#include "acdual.h"
```

3. On the **Build** menu, click **Settings**, and then select INITIIDS.CPP from the file list for each configuration.
4. Click the **C++** tab, click category **Precompiled Headers**, and select the **Not using precompiled headers** radio button. Click OK to close the **Build Settings** dialog box.

Specifying the Correct Object Class Name in the Type Library

The wizards shipped with Visual C++ incorrectly use the implementation class name to specify the coclass in the server's ODL file for OLE-creatable classes. While this will work, the implementation class name is probably not the class name you want users of your object to use. To specify the correct name, open the ODL file, locate each coclass statement, and replace the implementation class name with the correct external name.

Note that when the coclass statement is changed, the variable names of **CLSID**s in the MkTypLib-generated header file will change accordingly. You will need to update your code to use the new variable names.

Handling Exceptions and the Automation Error Interfaces

Your automation object's methods and property accessor functions may throw exceptions. If so, you should handle them in your dual-interface implementation and pass information about the exception back to the controller through the OLE Automation error-handling interface, **IErrorInfo**. This interface provides for detailed, contextual error information through both **IDispatch** and VTBL interfaces. To indicate that an error handler is available, you should implement the **ISupportErrorInfo** interface.

To illustrate the error-handling mechanism, assume that the ClassWizard-generated functions used to implement the standard dispatch support throw exceptions. MFC's implementation of **IDispatch::Invoke** typically catches these exceptions and converts them into an EXCEPTINFO structure that is returned through the **Invoke** call. However, when VTBL interface is used, you are responsible for catching the exceptions yourself. As an example of protecting your dual-interface methods:

```
STDMETHODIMP CAutoClickDoc::XDualAClick::put_text(BSTR newText)
{
   METHOD_PROLOGUE(CAutoClickDoc, DualAClick)
   TRY_DUAL(IID_IDualAClick)
   {
      // MFC automatically converts from Unicode BSTR to
      // Ansi CString, if necessary...
      pThis->m_str = newText;
      return NOERROR;
   }
   CATCH_ALL_DUAL
}
```

`CATCH_ALL_DUAL` takes care of returning the correct error code when an exception occurs. `CATCH_ALL_DUAL` converts an MFC exception into OLE Automation error-handling information using the **ICreateErrorInfo** interface. (An example `CATCH_ALL_DUAL` macro is in the file MFCDUAL.H in the ACDUAL sample. The function it calls to handle exceptions, `DualHandleException`, is in the file MFCDUAL.CPP.) `CATCH_ALL_DUAL` determines the error code to return based on the type of exception that occurred:

- COleDispatchException – In this case, **HRESULT** is constructed using the following code:

  ```
  hr = MAKE_HRESULT(SEVERITY_ERROR, FACILITY_ITF,
                              (e->m_wCode + 0x200));
  ```

  This creates an **HRESULT** specific to the interface that caused the exception. The error code is offset by 0x200 to avoid any conflicts with system-defined **HRESULT**s for standard OLE interfaces.

- CMemoryException – In this case, `E_OUTOFMEMORY` is returned.
- Any other exception – In this case, `E_UNEXPECTED` is returned.

To indicate that the OLE Automation error handler is used, you should also implement the **ISupportErrorInfo** interface.

First, add code to your automation class definition to show it supports **ISupportErrorInfo**.

Second, add code to your automation class's interface map to associate the **ISupportErrorInfo** implementation class with

MFC's **QueryInterface** mechanism. The `INTERFACE_PART` statement matches the class defined for **ISupportErrorInfo**.

Finally, implement the class defined to support **ISupportErrorInfo**.

(The ACDUAL sample contains three macros to help do these three steps, `DECLARE_DUAL_ERRORINFO`, `DUAL_ERRORINFO_PART`, and `IMPLEMENT_DUAL_ERRORINFO`, all contained in MFCDUAL.H.)

The following example implements a class defined to support **ISupportErrorInfo**. `CAutoClickDoc` is the name of your automation class and `IID_IDualAClick` is the **IID** for the interface that is the source of errors reported through the OLE Automation error object:

```
STDMETHODIMP_(ULONG) CAutoClickDoc::XSupportErrorInfo::AddRef()
{
   METHOD_PROLOGUE(CAutoClickDoc, SupportErrorInfo)
   return pThis->ExternalAddRef();
}
STDMETHODIMP_(ULONG) CAutoClickDoc::XSupportErrorInfo::Release()
{
   METHOD_PROLOGUE(CAutoClickDoc, SupportErrorInfo)
   return pThis->ExternalRelease();
}
STDMETHODIMP CAutoClickDoc::XSupportErrorInfo::QueryInterface(
   REFIID iid, LPVOID* ppvObj)
{
   METHOD_PROLOGUE(CAutoClickDoc, SupportErrorInfo)
   return pThis->ExternalQueryInterface(&iid, ppvObj);
}
STDMETHODIMP CAutoClickDoc::XSupportErrorInfo::InterfaceSupportsErrorInfo(
   REFIID iid)
{
   METHOD_PROLOGUE(CAutoClickDoc, SupportErrorInfo)
   return (iid == IID_IDualAClick) ? S_OK : S_FALSE;
}
```

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN066: Common MFC 3.x to 4.0 Porting Issues

This technical note described the most common problems that can occur when attempting to port an application written with MFC 3.x (the MFC included with Visual C++ 2.x) to MFC 4.0.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN067: Database Access from an ISAPI Server Extension

Note

As of Visual C++ .NET, the Visual C++ environment and wizards no longer support DAO (although the DAO classes are included and you can still use them). Microsoft recommends that you use OLE DB Templates or ODBC and MFC for new projects. You should only use DAO in maintaining existing applications.

ISAPI server extensions are DLLs. They work in the process space of the server. Concurrent requests made to an ISAPI extension are handled by separate threads. Therefore, the code in an ISAPI extension must be thread-safe.

Situation

Database access methods that use COM to handle initialization are not thread-safe. The specific problem lies in initializing the COM libraries in one thread and then using another thread to access the data. The current implementation of the Microsoft ODBC Desktop Drivers falls into this category. These drivers (dBase, Microsoft Access, Microsoft Excel, Paradox, and Text) that ship with Visual C++ use the Microsoft Jet 3.0 database engine, which is not thread-safe. Also, Data Access Objects (DAO) is a COM-based API, so it does not run reliably in the multithreaded environment of an ISAPI extension.

There may be other database access methods that are not thread-safe.

Resolution

To access a database from an ISAPI server extension, use an ODBC driver that has been designed and tested for multithreaded use. Microsoft's ODBC drivers for Microsoft SQL Server 6.x are an example.

The MFC ODBC database classes are thread-safe as of MFC 4.2. Be sure that any ODBC database driver you use with these classes is also thread-safe.

Applications that use the MFC ODBC database classes before version MFC 4.2 need to have the database code placed inside a critical section to ensure that only one thread accesses the MFC database code at a time. For more information, see class CCriticalSection.

Currently, the MFC DAO database classes are not thread-safe because they use the DAO COM-based objects.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN068: Performing Transactions with the Microsoft Access 7 ODBC Driver

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

This note describes how to perform transactions when using the MFC ODBC database classes and the Microsoft Access 7.0 ODBC driver included in the Microsoft ODBC Desktop Driver Pack version 3.0.

Overview

If your database application performs transactions, you must be careful to call **CDatabase::BeginTrans** and **CRecordset::Open** in the correct sequence in your application. The Microsoft Access 7.0 driver uses the Microsoft Jet database engine, and Jet requires that your application not begin a transaction on any database that has an open cursor. For the MFC ODBC database classes, an open cursor equates to an open **CRecordset** object.

If you open a recordset before calling **BeginTrans**, you may not see any error messages. However, any recordset updates your application makes become permanent after calling **CRecordset::Update**, and the updates will not be rolled back by calling **Rollback**. To avoid this problem, you must call **BeginTrans** first and then open the recordset.

MFC checks the driver functionality for cursor commit and rollback behavior. Class **CDatabase** provides two member functions, **GetCursorCommitBehavior** and **GetCursorRollbackBehavior**, to determine the effect of any transaction on your open **CRecordset** object. For the Microsoft Access 7.0 ODBC driver, these member functions return **SQL_CB_CLOSE** because the Access driver does not support cursor preservation. Therefore, you must call **CRecordset::Requery** following a **CommitTrans** or **Rollback** operation.

When you need to perform multiple transactions one after another, you cannot call **Requery** after the first transaction and then start the next one. You must close the recordset before the next call to **BeginTrans** in order to satisfy Jet's requirement. This technical note describes two methods of handling this situation:

- Closing the recordset after each **CommitTrans** or **Rollback** operation.
- Using the ODBC API function **SQLFreeStmt**.

Closing the Recordset after each CommitTrans or Rollback Operation

Before starting a transaction, make sure the recordset object is closed. After calling **BeginTrans**, call the recordset's **Open** member function. Close the recordset immediately after calling **CommitTrans** or **Rollback**. Note that repeatedly opening and closing the recordset can slow an application's performance.

Using SQLFreeStmt

You can also use the ODBC API function **SQLFreeStmt** to explicitly close the cursor after ending a transaction. To start another transaction, call **BeginTrans** followed by **CRecordset::Requery**. When calling **SQLFreeStmt**, you must specify the recordset's HSTMT as the first parameter and **SQL_CLOSE** as the second parameter. This method is faster than closing and opening the recordset at the start of every transaction. The following code demonstrates how to implement this technique:

```
CMyDatabase db;
db.Open( "MYDATASOURCE" );
CMyRecordset rs( &db );

// start transaction 1 and
// open the recordset
db.BeginTrans( );
rs.Open( );

// manipulate data

// end transaction 1
```

```
db.CommitTrans( );  // or Rollback( )

// close the cursor
::SQLFreeStmt( rs.m_hstmt, SQL_CLOSE );

// start transaction 2
db.BeginTrans( );

// now get the result set
rs.Requery( );

// manipulate data

// end transaction 2
db.CommitTrans( );

rs.Close( );
db.Close( );
```

Another way to implement this technique is to write a new function, **RequeryWithBeginTrans**, which you can call to start the next transaction after you commit or rollback the first one. To write such a function, do the following steps:

1. Copy the code for **CRecordset::Requery( )** to the new function.
2. Add the following line immediately after the call to **SQLFreeStmt**:

   ```
   m_pDatabase->BeginTrans( );
   ```

Now you can call this function between each pair of transactions:

```
// start transaction 1 and
// open the recordset
db.BeginTrans( );
rs.Open( );

// manipulate data

// end transaction 1
db.CommitTrans( );  // or Rollback( )

// close the cursor, start new transaction,
// and get the result set
rs.RequeryWithBeginTrans( );

// manipulate data

// end transaction 2
db.CommitTrans( );  // or Rollback( )
```

Note

Do not use this technique if you need to change the recordset member variables **m_strFilter** or **m_strSort** between transactions. In that case, you should close the recordset after each **CommitTrans** or **Rollback** operation.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN069: Processing HTML Forms Using Internet Server Extension DLLs and Command Handlers

Technical Note 69, describing how to use the new **MfcISAPICommand** parameter in HTML forms, can now be found in Processing HTML Forms Using Internet Server Extension DLLs and Command Handlers.

Note
In addition to MFC, ATL Server provides a different mechanism for writing web applications and web services, using C++ and ISAPI. For more information, see ATL Server.

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

TN070: MFC Window Class Names

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

MFC windows use a dynamically created class name that reflects the features of the window. MFC generates class names dynamically for frame windows, views, and popup windows produced by the application. Dialog boxes and controls produced by an MFC application have the Windows-supplied name for the class of window in question.

You can replace the dynamically provided class name by registering your own window class and using it in an override of PreCreateWindow. Their MFC-supplied class names fit one of the two following forms:

```
Afx:%x:%x
Afx:%x:%x:%x:%x:%x
```

The hex digits that replace the `%x` characters are filled in from data from the WNDCLASS structure. MFC uses this technique so that multiple C++ classes requiring identical **WNDCLASS** structures can share the same registered window class. Unlike most simple Win32 applications, MFC applications have only one **WNDPROC**, so you can easily share **WNDCLASS** structures to save time and memory. The replaceable values for the `%x` characters shown above are as follows:

- **WNDCLASS.hInstance**
- **WNDCLASS.style**
- **WNDCLASS.hCursor**
- **WNDCLASS.hbrBackground**
- **WNDCLASS.hIcon**

The first form (`Afx:%x:%x`) is used when **hCursor**, **hbrBackground**, and **hIcon** are all **NULL**.

See Also

Reference

TN020: ID Naming and Numbering Conventions

Other Resources

Technical Notes by Number

Technical Notes by Category

TN071: MFC IOleCommandTarget Implementation

Note
The following technical note has not been updated since it was first included in the online documentation. As a result, some procedures and topics might be out of date or incorrect. For the latest information, it is recommended that you search for the topic of interest in the online documentation index.

The **IOleCommandTarget** interface enables objects and their containers to dispatch commands to each other. For example, an object's toolbars may contain buttons for commands such as **Print**, **Print Preview**, **Save**, **New**, and **Zoom**. If such an object were embedded in a container that supports **IOleCommandTarget**, the object could enable its buttons and forward the commands to the container for processing when the user clicked them. If a container wanted the embedded object to print itself, it could make this request by sending a command through the **IOleCommandTarget** interface of the embedded object.

IOleCommandTarget is an Automation-like interface in that it is used by a client to invoke methods on a server. However, using **IOleCommandTarget** saves the overhead of making calls via Automation interfaces because programmers don't have to use the typically expensive **Invoke** method of **IDispatch**.

In MFC, the **IOleCommandTarget** interface is used by Active document servers to allow Active document containers to dispatch commands to the server. The Active document server class, **CDocObjectServerItem**, uses MFC interface maps (see TN038: MFC/OLE IUnknown Implementation) to implement the **IOleCommandTarget** interface.

IOleCommandTarget is also implemented in the **COleFrameHook** class. **COleFrameHook** is an undocumented MFC class that implements the frame window functionality of in-place editing containers. **COleFrameHook** also uses MFC interface maps to implement the **IOleCommandTarget** interface. **COleFrameHook**'s implementation of **IOleCommandTarget** forwards OLE commands to **COleDocObjectItem**-derived Active document containers. This allows any MFC Active document container to receive messages from contained Active document servers.

MFC OLE Command Maps

MFC developers can take advantage of **IOleCommandTarget** by using MFC OLE command maps. OLE command maps are like message maps because they can be used to map OLE commands to member functions of the class that contains the command map. To make this work, place macros in the command map to specify the OLE command group of the command you want to handle, the OLE command, and the command ID of the WM_COMMAND message that will be sent when the OLE command is received. MFC also provides a number of predefined macros for standard OLE commands. For a list of the standard OLE commands that were originally designed for use with Microsoft Office applications, see the OLECMDID enumeration, which is defined in docobj.h.

When an OLE command is received by an MFC application that contains an OLE command map, MFC tries to find the command ID and command group for the requested command in the OLE command map of the application. If a match is found, a **WM_COMMAND** message is dispatched to the application containing the command map with the ID of the requested command. (See the description of **ON_OLECMD** below.) In this way, OLE commands dispatched to an application are turned into **WM_COMMAND** messages by MFC. The **WM_COMMAND** messages are then routed through the application's message maps using the MFC standard command routing architecture.

Unlike message maps, MFC OLE command maps are not supported by ClassWizard. MFC developers must add OLE command map support and OLE command map entries by hand. OLE command maps can be added to MFC Active document servers in any class that is in the **WM_COMMAND** message-routing chain at the time the Active document is in-place active in a container. These classes include the application's classes derived from CWinApp, CView, CDocument, and COleIPFrameWnd. In Active document containers, OLE command maps can only be added to the COleDocObjectItem-derived class. Also, in Active document containers, the **WM_COMMAND** messages will only be dispatched to the message map in the **COleDocObjectItem**-derived class.

OLE Command Map Macros

Use the following macros to add command map functionality to your class:

```
DECLARE_OLECMD_MAP ()
```

This macro goes in the class declaration (typically in the header file) of the class that contains the command map.

```
BEGIN_OLECMD_MAP(
theClass
,
baseClass
)
```

theClass

Name of the class that contains the command map.

baseClass

Name of the base class of the class that contains the command map.

This macro marks the beginning of the command map. Use this macro in the implementation file for the class that contains the command map.

```
END_OLECMD_MAP()
```

This macro marks the end of the command map. Use this macro in the implementation file for the class that contains the command map. This macro must always follow the **BEGIN_OLECMD_MAP** macro.

```
ON_OLECMD(
pguid
,
olecmdid
,
id
)
```

pguid

Pointer to the GUID of the OLE command's command group. This parameter is **NULL** for the standard OLE command group.

olecmdid

OLE command ID of the command to be invoked.

id

ID of the **WM_COMMAND** message to be sent to the application containing the command map when this OLE command is invoked.

Use the **ON_OLECMD** macro in the command map to add entries for the OLE commands you want to handle. When the OLE commands are received, they will be converted to the specified **WM_COMMAND** message and routed through the application's message map using the standard MFC command-routing architecture.

Example

The following example shows how to add OLE command-handling capability to an MFC Active document server to handle the OLECMDID_PRINT OLE command. This example assumes that you used AppWizard to generate an MFC application that is an Active document server.

1. In your **CView**-derived class's header file, add the `DECLARE_OLECMD_MAP` macro to the class declaration.

Note
Use the **CView**-derived class because it is one of the classes in the Active document server that is in the **WM_COMMAND** message-routing chain.

```
class CMyServerView : public CView
{
protected: // create from serialization only
   CMyServerView();
   DECLARE_DYNCREATE(CMyServerView)
   DECLARE_OLECMD_MAP()
. . .
};
```

2. In the implementation file for the **CView**-derived class, add the `BEGIN_OLECMD_MAP` and `END_OLECMD_MAP` macros:

```
BEGIN_OLECMD_MAP(CMyServerView, CView)

END_OLECMD_MAP()
```

3. To handle the standard OLE print command, add an ON_OLECMD macro to the command map specifying the OLE command ID for the standard print command and **ID_FILE_PRINT** for the **WM_COMMAND** ID. **ID_FILE_PRINT** is the standard print command ID used by AppWizard-generated MFC applications:

```
BEGIN_OLECMD_MAP(CMyServerView, CView)
   ON_OLECMD(NULL,OLECMDID_PRINT,ID_FILE_PRINT)
END_OLECMD_MAP()
```

Note that one of the standard OLE command macros, defined in afxdocob.h, could be used in place of the **ON_OLECMD** macro because **OLECMDID_PRINT** is a standard OLE command ID. The **ON_OLECMD_PRINT** macro will accomplish the same task as the **ON_OLECMD** macro shown above.

When a container application sends this server an **OLECMDID_PRINT** command through the server's **IOleCommandTarget** interface, the MFC printing command handler will be invoked in the server, causing the server to print the application. The Active document container's code to invoke the print command added in the steps above would look something like this:

```
void CContainerCntrItem::DoOleCmd()
{
   IOleCommandTarget *pCmd = NULL;
   HRESULT hr = E_FAIL;
   OLECMD ocm={OLECMDID_PRINT, 0};

   hr = m_lpObject->QueryInterface(IID_IOleCommandTarget,reinterpret_cast<void**>(&pCmd));
   if(FAILED(hr))
      return;

   hr = pCmd->QueryStatus(NULL, 1, &ocm, NULL);
   if(SUCCEEDED(hr) && (ocm.cmdf & OLECMDF_ENABLED))
   {
      //Command is available and enabled so call it
      COleVariant vIn;
      COleVariant vOut;
      hr = pCmd->Exec(NULL, OLECMDID_PRINT,
 OLECMDEXECOPT_DODEFAULT, &vIn, &vOut);
      ASSERT(SUCCEEDED(hr));
   }
   pCmd->Release();
}
```

See Also

Other Resources

Technical Notes by Number

Technical Notes by Category

MFC Classes

The following classes are included in the MFC Library.

Class	Header file
CAnimateCtrl	afxcmn.h
CArchive	afx.h
CArchiveException	afx.h
CArray	afxtempl.h
CAsyncMonikerFile	afxole.h
CAsyncSocket	afxsock.h
CBitmap	afxwin.h
CBitmapButton	afxext.h
CBrush	afxwin.h
CButton	afxwin.h
CByteArray	afxcoll.h
CCachedDataPathProperty	afxctl.h
CCheckListBox	afxwin.h
CClientDC	afxwin.h
CCmdTarget	afxwin.h
CCmdUI	afxwin.h
CColorDialog	afxdlgs.h
CComboBox	afxwin.h
CComboBoxEx	afxcmn.h
CCommandLineInfo	afxwin.h
CCommonDialog	afxdlgs.h
CConnectionPoint	afxdisp.h
CControlBar	afxext.h
CCriticalSection	afxmt.h

CCtrlView	afxwin.h
CDaoDatabase	afxdao.h
CDaoException	afxdao.h
CDaoFieldExchange	afxdao.h
CDaoQueryDef	afxdao.h
CDaoRecordset	afxdao.h
CDaoRecordView	afxdao.h
CDaoTableDef	afxdao.h
CDaoWorkspace	afxdao.h
CDatabase	afxdb.h
CDataExchange	afxwin.h
CDataPathProperty	afxctl.h
CDateTimeCtrl	afxdtctl.h
CDBException	afxdb.h
CDBVariant	afxdb.h
CDC	afxwin.h
CDHtmlDialog	afxdhtml.h
CDialog	afxwin.h
CDialogBar	afxext.h
CDocItem	afxole.h
CDockState	afxadv.h
CDocObjectServer	afxdocob.h
CDocObjectServerItem	afxdocob.h
CDocTemplate	afxwin.h
CDocument	afxwin.h
CDragListBox	afxcmn.h
CDumpContext	afx.h

CDWordArray	afxcoll.h
CEdit	afxwin.h
CEditView	afxext.h
CEvent	afxmt.h
CException	afx.h
CFieldExchange	afxdb.h
CFile	afx.h
CFileDialog	afxdlgs.h
CFileException	afx.h
CFileFind	afx.h
CFindReplaceDialog	afxdlgs.h
CFont	afxwin.h
CFontDialog	afxdlgs.h
CFontHolder	afxctl.h
CFormView	afxext.h
CFrameWnd	afxwin.h
CFtpConnection	afxinet.h
CFtpFileFind	afxinet.h
CGdiObject	afxwin.h
CGopherConnection	afxinet.h
CGopherFile	afxinet.h
CGopherFileFind	afxinet.h
CGopherLocator	afxinet.h
CHeaderCtrl	afxcmn.h
CHotKeyCtrl	afxcmn.h
CHtmlEditCtrl	afxhtml.h
CHtmlEditCtrlBase	afxhtml.h

CHtmlEditDoc	afxhtml.h
CHtmlEditView	afxhtml.h
CHtmlStream	afxisapi.h
CHtmlView	afxhtml.h
CHttpArgList	afxisapi.h
CHttpConnection	afxinet.h
CHttpFile	afxinet.h
CHttpFilter	afxisapi.h
CHttpFilterContext	afxisapi.h
CHttpServer	afxisapi.h
CHttpServerContext	afxisapi.h
CImageList	afxcmn.h
CInternetConnection	afxinet.h
CInternetException	afxinet.h
CInternetFile	afxinet.h
CInternetSession	afxinet.h
CIPAddressCtrl	afxcmn.h
CLinkCtrl	afxcmn.h
CList	afxtempl.h
CListBox	afxwin.h
CListCtrl	afxcmn.h
CListView	afxcview.h
CLongBinary	afxdb_.h
CMap	afxtempl.h
CMapPtrToPtr	afxcoll.h
CMapPtrToWord	afxcoll.h
CMapStringToOb	afxcoll.h

CMapStringToPtr	afxcoll.h
CMapStringToString	afxcoll.h
CMapWordToOb	afxcoll.h
CMapWordToPtr	afxcoll.h
CMDIChildWnd	afxwin.h
CMDIFrameWnd	afxwin.h
CMemFile	afx.h
CMemoryException	afx.h
CMenu	afxwin.h
CMetaFileDC	afxext.h
CMiniFrameWnd	afxwin.h
CMonikerFile	afxole.h
CMonthCalCtrl	afxdtctl.h
CMultiDocTemplate	afxwin.h
CMultiLock	afxmt.h
CMultiPageDHtmlDialog	afxdhtml.h
CMutex	afxmt.h
CNotSupportedException	afx.h
CObArray	afxcoll.h
CObject	afx.h
CObList	afxcoll.h
COccManager	afxocc.h
COleBusyDialog	afxodlgs.h
COleChangeIconDialog	afxodlgs.h
COleChangeSourceDialog	afxodlgs.h
COleClientItem	afxole.h
COleCmdUI	afxdocob.h

COleControl	afxctl.h
COleControlContainer	afxocc.h
COleControlModule	afxctl.h
COleControlSite	afxocc.h
COleConvertDialog	afxodlgs.h
COleCurrency	afxdisp.h
COleDataObject	afxole.h
COleDataSource	afxole.h
COleDBRecordView	afxoledb.h
COleDialog	afxodlgs.h
COleDispatchDriver	afxdisp.h
COleDispatchException	afxdisp.h
COleDocObjectItem	afxole.h
COleDocument	afxole.h
COleDropSource	afxole.h
COleDropTarget	afxole.h
COleException	afxdisp.h
COleInsertDialog	afxodlgs.h
COleIPFrameWnd	afxole.h
COleLinkingDoc	afxole.h
COleLinksDialog	afxodlgs.h
COleMessageFilter	afxole.h
COleObjectFactory	afxdisp.h
COlePasteSpecialDialog	afxodlgs.h
COlePropertiesDialog	afxodlgs.h
COlePropertyPage	afxctl.h
COleResizeBar	afxole.h

COleSafeArray	afxdisp.h
COleServerDoc	afxole.h
COleServerItem	afxole.h
COleStreamFile	afxole.h
COleTemplateServer	afxdisp.h
COleUpdateDialog	afxodlgs.h
COleVariant	afxdisp.h
CPageSetupDialog	afxdlgs.h
CPaintDC	afxwin.h
CPalette	afxwin.h
CPen	afxwin.h
CPictureHolder	afxctl.h
CPoint	atltypes.h
CPrintDialog	afxdlgs.h
CPrintDialogEx	afxdlgs.h
CProgressCtrl	afxcmn.h
CPropertyPage	afxdlgs.h
CPropertySheet	afxdlgs.h
CPropExchange	afxctl.h
CPtrArray	afxcoll.h
CPtrList	afxcoll.h
CReBar	afxext.h
CReBarCtrl	afxcmn.h
CRecentFileList	afxadv.h
CRecordset	afxdb.h
CRecordView	afxdb.h
CRect	atltypes.h

CRectTracker	afxext.h
CResourceException	afxwin.h
CRgn	afxwin.h
CRichEditCntrItem	afxrich.h
CRichEditCtrl	afxcmn.h
CRichEditDoc	afxrich.h
CRichEditView	afxrich.h
CScrollBar	afxwin.h
CScrollView	afxwin.h
CSemaphore	afxmt.h
CSharedFile	afxadv.h
CSingleDocTemplate	afxwin.h
CSingleLock	afxmt.h
CSize	atltypes.h
CSliderCtrl	afxcmn.h
CSocket	afxsock.h
CSocketFile	afxsock.h
CSpinButtonCtrl	afxcmn.h
CSplitterWnd	afxext.h
CStatic	afxwin.h
CStatusBar	afxext.h
CStatusBarCtrl	afxcmn.h
CStdioFile	afx.h
CStringArray	afxcoll.h
CStringList	afxcoll.h
CSyncObject	afxmt.h
CTabCtrl	afxcmn.h

CToolBar	afxext.h
CToolBarCtrl	afxcmn.h
CToolTipCtrl	afxcmn.h
CTreeCtrl	afxcmn.h
CTreeView	afxcview.h
CTypedPtrArray	afxtempl.h
CTypedPtrList	afxtempl.h
CTypedPtrMap	afxtempl.h
CUIntArray	afxcoll.h
CUserException	afxwin.h
CView	afxwin.h
CWaitCursor	afxwin.h
CWinApp	afxwin.h
CWindowDC	afxwin.h
CWinThread	afxwin.h
CWnd	afxwin.h
CWordArray	afxcoll.h

See Also

Other Resources

MFC Reference

CAnimateCtrl Class

Provides the functionality of the Windows common animation control.

```
class CAnimateCtrl : public CWnd
```

Remarks

This control (and therefore the **CAnimateCtrl** class) is available only to programs running under Windows 95, Windows 98, and Windows NT version 3.51 and later.

An animation control is a rectangular window that displays a clip in AVI (Audio Video Interleaved) format— the standard Windows video/audio format. An AVI clip is a series of bitmap frames, like a movie.

Animation controls can play only simple AVI clips. Specifically, the clips to be played by an animation control must meet the following requirements:

- There must be exactly one video stream and it must have at least one frame.
- There can be at most two streams in the file (typically the other stream, if present, is an audio stream, although the animation control ignores audio information).
- The clip must either be uncompressed or compressed with RLE8 compression.
- No palette changes are allowed in the video stream.

You can add the AVI clip to your application as an AVI resource, or it can accompany your application as a separate AVI file.

Because your thread continues executing while the AVI clip is displayed, one common use for an animation control is to indicate system activity during a lengthy operation. For example, the Find dialog box of Windows Explorer displays a moving magnifying glass as the system searches for a file.

If you create a **CAnimateCtrl** object within a dialog box or from a dialog resource using the dialog editor, it will be automatically destroyed when the user closes the dialog box.

If you create a **CAnimateCtrl** object within a window, you may need to destroy it. If you create the **CAnimateCtrl** object on the stack, it is destroyed automatically. If you create the **CAnimateCtrl** object on the heap by using the **new** function, you must call **delete** on the object to destroy it. If you derive a new class from **CAnimateCtrl** and allocate any memory in that class, override the **CAnimateCtrl** destructor to dispose of the allocations.

For more information on using **CAnimateCtrl**, see Controls and Using CAnimateCtrl.

Requirements

Header: afxcmn.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

CMNCTRL1 Sample: Demonstrates Common Control MFC Classes, Part 1

MFCIE Sample: Demonstrates the MFC CHtmlView and CReBar Classes

Reference

CWnd Class

Hierarchy Chart

CAnimateCtrl::Create

ON_CONTROL

Other Resources

CAnimateCtrl Members

CAnimateCtrl Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Construction

CAnimateCtrl	Constructs a **CAnimateCtrl** object.

Initialization

Create	Creates an animation control and attaches it to a **CAnimateCtrl** object.
CreateEx	Creates an animation control with the specified Windows extended styles and attaches it to a **CAnimateCtrl** object.

Operations

Close	Closes the AVI clip that was previously opened.
Open	Opens an AVI clip from a file or resource and displays the first frame.
Play	Plays the AVI clip without sound.
Seek	Displays a selected single frame of the AVI clip.
Stop	Stops playing the AVI clip.

See Also

Reference

CAnimateCtrl Class

Hierarchy Chart

CAnimateCtrl Member Functions

For information about the member functions in **CAnimateCtrl**, see CAnimateCtrl Members.

CAnimateCtrl::CAnimateCtrl

Constructs a **CAnimateCtrl** object.

```
CAnimateCtrl( );
```

Remarks

You must call the Create member function before you can perform any other operations on the object you create.

Example

```
// This example creates a secondary thread that implements
// the methods of CAnimateCtrl. The procedure of the thread
// is MyClipThreadProc and the thread was created with the
// code AfxBeginThread( MyClipThreadProc, (LPVOID) pParentWnd).
// The example code creates and initializes an animation control,
// then proceeds to pump messages from the queue until one the
// private messages WM_STOPCLIP, WM_PLAYCLIP, WM_SHOWFIRSTFRAME or
// WM_SHOWLASTFRAME is received. The appropriate action is done for
// these messages. The thread ends when the WM_STOPCLIP is received.
// NOTE: the thread parameter, pParam, is a pointer to a CWnd object
// that will be the parent of the animation control.

#define WM_STOPCLIP               WM_USER+1
#define WM_PLAYCLIP               WM_USER+2
#define WM_SHOWFIRSTFRAME         WM_USER+3
#define WM_SHOWLASTFRAME         WM_USER+4

UINT MyClipThreadProc( LPVOID pParam )
{
   // NOTE: pParentWnd is the parent window of the animation control.
   CWnd* pParentWnd = (CWnd*) pParam;
   CAnimateCtrl cAnimCtrl;

   // Create the animation control.
   if (!cAnimCtrl.Create(WS_CHILD|WS_VISIBLE|ACS_CENTER,
      CRect(10,10,100,100), pParentWnd, 1))
         return false;

   // Open the AVI file.
   if (!cAnimCtrl.Open(_T("myavi.avi")))
      return false;

   // Pump message from the queue until the stop play message is received.
   MSG msg;
   while (GetMessage(&msg, NULL, 0, 0) && (msg.message != WM_STOPCLIP))
   {
      switch (msg.message)
      {
         // Start playing from the first frame to the last,
         // continuously repeating.
         case WM_PLAYCLIP:
            if (!cAnimCtrl.Play(0, -1, -1))
               return false;
            break;

         // Show the first frame.
         case WM_SHOWFIRSTFRAME:
            if (!cAnimCtrl.Seek(0))
               return false;
            cAnimCtrl.RedrawWindow();
            break;
```

```
            // Show the last frame.
            case WM_SHOWLASTFRAME:
               if (!cAnimCtrl.Seek(-1))
                  return false;
               cAnimCtrl.RedrawWindow();
               break;
         }

         TranslateMessage(&msg);
         DispatchMessage(&msg);
      }

      cAnimCtrl.Stop();
      cAnimCtrl.Close();

      return true;
   }
```

See Also

Reference

CAnimateCtrl Class

Hierarchy Chart

CAnimateCtrl::Create

CAnimateCtrl::CreateEx

Other Resources

CAnimateCtrl Members

CAnimateCtrl::Close

Closes the AVI clip that was previously opened in the animation control (if any) and removes it from memory.

```
BOOL Close( );
```

Return Value

Nonzero if successful; otherwise zero.

Example

See the example for CAnimateCtrl::CAnimateCtrl.

See Also

Reference

CAnimateCtrl Class
Hierarchy Chart
CAnimateCtrl::Open

Other Resources

CAnimateCtrl Members

CAnimateCtrl::Create

Creates an animation control and attaches it to a **CAnimateCtrl** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the animation control's style. Apply any combination of the windows styles described in the Remarks section below and the animation control styles described in Animation Control Styles in the *Platform SDK*.

rect

Specifies the animation control's position and size. It can be either a CRect object or a RECT structure.

pParentWnd

Specifies the animation control's parent window, usually a **CDialog**. It must not be **NULL.**

nID

Specifies the animation control's ID.

Return Value

Nonzero if successful; otherwise zero.

Remarks

You construct a **CAnimateCtrl** in two steps. First, call the constructor, and then call **Create**, which creates the animation control and attaches it to the **CAnimateCtrl** object.

Apply the following window styles to an animation control.

- **WS_CHILD** Always
- **WS_VISIBLE** Usually
- **WS_DISABLED** Rarely

If you want to use extended windows styles with your animation control, call CreateEx instead of **Create**.

In addition to the window styles listed above, you may want to apply one or more of the animation control styles to an animation control. See the *Platform SDK* for more information on animation control styles.

Example

See the example for CAnimateCtrl::CAnimateCtrl.

See Also

Reference

CAnimateCtrl Class
Hierarchy Chart
CAnimateCtrl::CAnimateCtrl
CAnimateCtrl::Open
CAnimateCtrl::Play
CAnimateCtrl::Seek

Other Resources

CAnimateCtrl Members

CAnimateCtrl::CreateEx

Creates a control (a child window) and associates it with the **CAnimateCtrl** object.

```
virtual BOOL CreateEx(
   DWORD dwExStyle,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwExStyle

Specifies the extended style of the control being created. For a list of extended Windows styles, see the *dwExStyle* parameter for CreateWindowEx in the *Platform SDK*.

dwStyle

Specifies the animation control's style. Apply any combination of the window and animation control styles described in Animation Control Styles in the *Platform SDK*.

rect

A reference to a RECT structure describing the size and position of the window to be created, in client coordinates of *pParentWnd*.

pParentWnd

A pointer to the window that is the control's parent.

nID

The control's child-window ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Use **CreateEx** instead of Create to apply extended Windows styles, specified by the Windows extended style preface **WS_EX_**.

See Also

Reference

CAnimateCtrl Class
Hierarchy Chart
CAnimateCtrl::CAnimateCtrl
CAnimateCtrl::Open
CAnimateCtrl::Play
CAnimateCtrl::Seek

Other Resources

CAnimateCtrl Members

CAnimateCtrl::Open

Call this function to open an AVI clip and display its first frame.

```
BOOL Open(
   LPCTSTR lpszFileName
);
BOOL Open(
   UINT nID
);
```

Parameters

lpszFileName

A **CString** object or a pointer to a null-terminated string that contains either the name of the AVI file or the name of an AVI resource. If this parameter is **NULL**, the system closes the AVI clip that was previously opened for the animation control, if any.

nID

The AVI resource identifier. If this parameter is **NULL**, the system closes the AVI clip that was previously opened for the animation control, if any.

Return Value

Nonzero if successful; otherwise zero.

Remarks

The AVI resource is loaded from the module that created the animation control.

Open does not support sound in an AVI clip; you can open only silent AVI clips.

If the animation control has the **ACS_AUTOPLAY** style, the animation control will automatically start playing the clip immediately after it opens it. It will continue to play the clip in the background while your thread continues executing. When the clip is done playing, it will automatically be repeated.

If the animation control has the **ACS_CENTER** style, the AVI clip will be centered in the control and the size of the control will not change. If the animation control does not have the **ACS_CENTER** style, the control will be resized when the AVI clip is opened to the size of the images in the AVI clip. The position of the top left corner of the control will not change, only the size of the control.

If the animation control has the **ACS_TRANSPARENT** style, the first frame will be drawn using a transparent background rather than the background color specified in the animation clip.

Example

See the example for CAnimateCtrl::CAnimateCtrl.

See Also

Reference

CAnimateCtrl Class
Hierarchy Chart
CAnimateCtrl::Close
CAnimateCtrl::Create

Other Resources

CAnimateCtrl Members

CAnimateCtrl::Play

Call this function to play an AVI clip in an animation control.

```
BOOL Play(
   UINT nFrom,
   UINT nTo,
   UINT nRep
);
```

Parameters

nFrom

Zero-based index of the frame where playing begins. Value must be less than 65,536. A value of 0 means begin with the first frame in the AVI clip.

nTo

Zero-based index of the frame where playing ends. Value must be less than 65,536. A value of – 1 means end with the last frame in the AVI clip.

nRep

Number of times to replay the AVI clip. A value of – 1 means replay the file indefinitely.

Return Value

Nonzero if successful; otherwise zero.

Remarks

The animation control will play the clip in the background while your thread continues executing. If the animation control has **ACS_TRANSPARENT** style, the AVI clip will be played using a transparent background rather than the background color specified in the animation clip.

Example

See the example for CAnimateCtrl::CAnimateCtrl.

See Also

Reference

CAnimateCtrl Class
Hierarchy Chart
CAnimateCtrl::Open
CAnimateCtrl::Stop
CAnimateCtrl::Seek
CAnimateCtrl::Create

Other Resources

CAnimateCtrl Members

CAnimateCtrl::Seek

Call this function to statically display a single frame of your AVI clip.

```
BOOL Seek(
   UINT nTo
);
```

Parameters

nTo

Zero-based index of the frame to display. Value must be less than 65,536. A value of 0 means display the first frame in the AVI clip. A value of –1 means display the last frame in the AVI clip.

Return Value

Nonzero if successful; otherwise zero.

Remarks

If the animation control has **ACS_TRANSPARENT** style, the AVI clip will be drawn using a transparent background rather than the background color specified in the animation clip.

Example

See the example for CAnimateCtrl::CAnimateCtrl.

See Also

Reference

CAnimateCtrl Class
Hierarchy Chart
CAnimateCtrl::Open
CAnimateCtrl::Play
CAnimateCtrl::Create

Other Resources

CAnimateCtrl Members

CAnimateCtrl::Stop

Call this function to stop playing an AVI clip in an animation control.

```
BOOL Stop( );
```

Return Value

Nonzero if successful; otherwise zero.

Example

See the example for CAnimateCtrl::CAnimateCtrl.

See Also

Reference

CAnimateCtrl Class
Hierarchy Chart
CAnimateCtrl::Play

Other Resources

CAnimateCtrl Members

CArchive Class

Allows you to save a complex network of objects in a permanent binary form (usually disk storage) that persists after those objects are deleted.

```
class CArchive
```

Remarks

CArchive does not have a base class.

Later you can load the objects from persistent storage, reconstituting them in memory. This process of making data persistent is called "serialization."

You can think of an archive object as a kind of binary stream. Like an input/output stream, an archive is associated with a file and permits the buffered writing and reading of data to and from storage. An input/output stream processes sequences of ASCII characters, but an archive processes binary object data in an efficient, nonredundant format.

You must create a CFile object before you can create a **CArchive** object. In addition, you must ensure that the archive's load/store status is compatible with the file's open mode. You are limited to one active archive per file.

When you construct a **CArchive** object, you attach it to an object of class **CFile** (or a derived class) that represents an open file. You also specify whether the archive will be used for loading or storing. A **CArchive** object can process not only primitive types but also objects of CObject-derived classes designed for serialization. A serializable class usually has a **Serialize** member function, and it usually uses the DECLARE_SERIAL and IMPLEMENT_SERIAL macros, as described under class **CObject**.

The overloaded extraction (**>>**) and insertion (**<<**) operators are convenient archive programming interfaces that support both primitive types and **CObject**-derived classes.

CArchive also supports programming with the MFC Windows Sockets classes CSocket and CSocketFile. The IsBufferEmpty member function supports that usage.

For more information on **CArchive**, see the articles Serialization and Windows Sockets: Using Sockets with Archives.

Requirements

Header: afx.h

See Also

Reference

Hierarchy Chart
CFile Class
CObject Class
CSocket Class
CSocketFile Class

Other Resources

CArchive Members

CArchive Members

Data Members

m_pDocument	Points to the **CDocument** object being serialized.

Construction

Abort	Closes an archive without throwing an exception.
CArchive	Creates a **CArchive** object.
Close	Flushes unwritten data and disconnects from the **CFile**.

Basic Input/Output

Flush	Flushes unwritten data from the archive buffer.
operator <<	Stores objects and primitive types to the archive.
operator >>	Loads objects and primitive types from the archive.
Read	Reads raw bytes.
ReadString	Reads a single line of text.
Write	Writes raw bytes.
WriteString	Writes a single line of text.

Status

GetFile	Gets the **CFile** object pointer for this archive.
GetObjectSchema	Called from the **Serialize** function to determine the version of the object that is being deserialized.
IsBufferEmpty	Determines whether the buffer has been emptied during a Windows Sockets receive process.
IsByteSwapping	MFC no longer supports this method.
IsLoading	Determines whether the archive is loading.
IsStoring	Determines whether the archive is storing.
SetObjectSchema	Sets the object schema stored in the archive object.

Object Input/Output

MapObject	Places objects in the map that are not serialized to the file, but that are available for subobjects to reference.
ReadClass	Reads a class reference previously stored with **WriteClass**.
ReadObject	Calls an object's **Serialize** function for loading.
SerializeClass	Reads or writes the class reference to the **CArchive** object depending on the direction of the **CArchive**.

SetLoadParams	Sets the size to which the load array grows. Must be called before any object is loaded or before **MapObject** or **ReadObject** is called.
SetStoreParams	Sets the hash table size and the block size of the map used to identify unique objects during the serialization process.
WriteClass	Writes a reference to the **CRuntimeClass** to the **CArchive**.
WriteObject	Calls an object's **Serialize** function for storing.

See Also

Reference

CArchive Class

Hierarchy Chart

CArchive Member Functions

For information about the member functions in **CArchive**, see CArchive Class Members.

CArchive::Abort

Call this function to close the archive without throwing an exception.

```
void Abort ( );
```

Remarks

The **CArchive** destructor will normally call **Close**, which will flush any data that has not been saved to the associated **CFile** object. This can cause exceptions.

When catching these exceptions, it is a good idea to use **Abort**, so that destructing the **CArchive** object doesn't cause further exceptions. When handling exceptions, **CArchive::Abort** will not throw an exception on failures because, unlike CArchive::Close, **Abort** ignores failures.

If you used **new** to allocate the **CArchive** object on the heap, then you must delete it after closing the file.

Example

See the example for CArchive::WriteClass.

See Also

Reference

CArchive Class
Hierarchy Chart
CArchive::Close
CFile::Close

Other Resources

CArchive Members

CArchive::CArchive

Constructs a **CArchive** object and specifies whether it will be used for loading or storing objects.

```
CArchive(
   CFile* pFile,
   UINT nMode,
   int nBufSize = 4096,
   void* lpBuf = NULL
);
```

Parameters

pFile

A pointer to the **CFile** object that is the ultimate source or destination of the persistent data.

nMode

A flag that specifies whether objects will be loaded from or stored to the archive. The *nMode* parameter must have one of the following values:

- **CArchive::load** Loads data from the archive. Requires only **CFile** read permission.
- **CArchive::store** Saves data to the archive. Requires **CFile** write permission.
- **CArchive::bNoFlushOnDelete** Prevents the archive from automatically calling **Flush** when the archive destructor is called. If you set this flag, you are responsible for explicitly calling **Close** before the destructor is called. If you do not, your data will be corrupted.

nBufSize

An integer that specifies the size of the internal file buffer, in bytes. Note that the default buffer size is 4,096 bytes. If you routinely archive large objects, you will improve performance if you use a larger buffer size that is a multiple of the file buffer size.

lpBuf

An optional pointer to a user-supplied buffer of size *nBufSize*. If you do not specify this parameter, the archive allocates a buffer from the local heap and frees it when the object is destroyed. The archive does not free a user-supplied buffer.

Remarks

You cannot change this specification after you have created the archive.

You may not use **CFile** operations to alter the state of the file until you have closed the archive. Any such operation will damage the integrity of the archive. You may access the position of the file pointer at any time during serialization by obtaining the archive's file object from the GetFile member function and then using the CFile::GetPosition function. You should call CArchive::Flush before obtaining the position of the file pointer.

Example

```
extern char* pFileName;
CFile f;
char buf[512];
if( !f.Open( pFileName, CFile::modeCreate | CFile::modeWrite ) ) {
   #ifdef _DEBUG
      afxDump << "Unable to open file" << "\n";
      exit( 1 );
   #endif
}
CArchive ar( &f, CArchive::store, 512, buf );
```

See Also

Reference

CArchive Class

Hierarchy Chart
CArchive::Close
CArchive::Flush
CFile::Close

Other Resources

CArchive Members

CArchive::Close

Flushes any data remaining in the buffer, closes the archive, and disconnects the archive from the file.

```
void Close( );
```

Remarks

No further operations on the archive are permitted. After you close an archive, you can create another archive for the same file or you can close the file.

The member function **Close** ensures that all data is transferred from the archive to the file, and it makes the archive unavailable. To complete the transfer from the file to the storage medium, you must first use CFile::Close and then destroy the **CFile** object.

Example

See the example for CArchive::WriteString.

See Also

Reference

CArchive Class
Hierarchy Chart
CArchive::Flush
CArchive::Abort

Other Resources

CArchive Members

CArchive::Flush

Forces any data remaining in the archive buffer to be written to the file.

```
void Flush( );
```

Remarks

The member function **Flush** ensures that all data is transferred from the archive to the file. You must call CFile::Close to complete the transfer from the file to the storage medium.

Example

```
CFile myFile("myfile.dat", CFile::modeCreate | CFile::modeWrite);
CArchive ar(&myFile, CArchive::store);

// Write a string to the archive.
ar.WriteString( "My string." );

// Flush all of the data to the file.
ar.Flush();
```

See Also

Reference

CArchive Class
Hierarchy Chart
CArchive::Close
CFile::Flush
CFile::Close

Other Resources

CArchive Members

CArchive::GetFile

Gets the **CFile** object pointer for this archive.

```
CFile* GetFile( ) const;
```

Return Value

A constant pointer to the **CFile** object in use.

Remarks

You must flush the archive before using **GetFile**.

Example

```
extern CArchive ar;
const CFile* fp = ar.GetFile();
```

See Also

Reference

CArchive Class

Hierarchy Chart

CArchive::Flush

Other Resources

CArchive Members

CArchive::GetObjectSchema

Call this function from the **Serialize** function to determine the version of the object that is currently being deserialized.

```
UINT GetObjectSchema( );
```

Return Value

During deserialization, the version of the object being read.

Remarks

Calling this function is only valid when the **CArchive** object is being loaded (CArchive::IsLoading returns nonzero). It should be the first call in the **Serialize** function and called only once. A return value of (**UINT**)–1 indicates that the version number is unknown.

A **CObject**-derived class may use **VERSIONABLE_SCHEMA** combined (using bitwise **OR**) with the schema version itself (in the **IMPLEMENT_SERIAL** macro) to create a "versionable object," that is, an object whose **Serialize** member function can read multiple versions. The default framework functionality (without **VERSIONABLE_SCHEMA**) is to throw an exception when the version is mismatched.

Example

```
IMPLEMENT_SERIAL(CMyObject, CObject, VERSIONABLE_SCHEMA|1)

void CMyObject::Serialize(CArchive& ar)
{
   if (ar.IsLoading())
   {
      int nVersion = ar.GetObjectSchema();

      switch(nVersion)
      {
      case 0:
         // read in previous version of
         // this object
         break;
      case 1:
         // read in current version of
         // this object
         break;
      default:
         // report unknown version of
         // this object
         break;
      }
   }
   else
   {
      // Normal storing code goes here
   }
}
```

See Also

Reference

CArchive Class
Hierarchy Chart
CObject::Serialize
CObject::IsSerializable
IMPLEMENT_SERIAL
DECLARE_SERIAL
CArchive::IsLoading

Other Resources

CArchive::IsBufferEmpty

Call this member function to determine whether the archive object's internal buffer is empty.

```
BOOL IsBufferEmpty( ) const;
```

Return Value

Nonzero if the archive's buffer is empty; otherwise 0.

Remarks

This function is supplied to support programming with the MFC Windows Sockets class **CSocketFile**. You do not need to use it for an archive associated with a **CFile** object.

The reason for using **IsBufferEmpty** with an archive associated with a **CSocketFile** object is that the archive's buffer might contain more than one message or record. After receiving one message, you should use **IsBufferEmpty** to control a loop that continues receiving data until the buffer is empty. For more information, see the Receive member function of class **CAsyncSocket**, which shows how to use **IsBufferEmpty**.

For more information, see Windows Sockets: Using Sockets with Archives.

See Also

Reference

CArchive Class
Hierarchy Chart
CSocketFile Class
CAsyncSocket::Receive

Other Resources

CArchive Members

CArchive::IsLoading

Determines whether the archive is loading data.

```
BOOL IsLoading( ) const;
```

Return Value

Nonzero if the archive is currently being used for loading; otherwise 0.

Remarks

This member function is called by the **Serialize** functions of the archived classes.

Example

```
int i;
extern CArchive ar;
if( ar.IsLoading() )
  ar >> i;
else
  ar << i;
```

See Also

Reference

CArchive Class
Hierarchy Chart
CArchive::IsStoring

Other Resources

CArchive Members

CArchive::IsStoring

Determines whether the archive is storing data.

```
BOOL IsStoring( ) const;
```

Return Value

Nonzero if the archive is currently being used for storing; otherwise 0.

Remarks

This member function is called by the **Serialize** functions of the archived classes.

If the **IsStoring** status of an archive is nonzero, then its **IsLoading** status is 0, and vice versa.

Example

```
int i;
extern CArchive ar;
if( ar.IsStoring() )
  ar << i;
else
  ar >> i;
```

See Also

Reference

CArchive Class

Hierarchy Chart

CArchive::IsLoading

Other Resources

CArchive Members

CArchive::MapObject

Call this member function to place objects in the map that are not really serialized to the file, but that are available for subobjects to reference.

```
void MapObject(
   const CObject* pOb
);
```

Parameters

pOb

A constant pointer to the object being stored.

Remarks

For example, you might not serialize a document, but you would serialize the items that are part of the document. By calling **MapObject**, you allow those items, or subobjects, to reference the document. Also, serialized subitems can serialize their **m_pDocument** back pointer.

You can call **MapObject** when you store to and load from the **CArchive** object. **MapObject** adds the specified object to the internal data structures maintained by the **CArchive** object during serialization and deserialization, but unlike ReadObject and WriteObject**,** it does not call serialize on the object.

Example

```
// MyDoc.h
// Document should have DECLARE_SERIAL and IMPLEMENT_SERIAL

class CMyDocument : public CDocument
{
   CObList m_listOfSubItems;
    ...
   DECLARE_SERIAL(CMyDocument)
};

// MyDoc.cpp
...
IMPLEMENT_SERIAL(CMyDocument, CObject, 1)
...
void CMyDocument::Serialize(CArchive& ar)
{
   if (ar.IsStoring())
   {
      // TODO: add storing code here
   }
   else
   {
      // TODO: add loading code here
   }

   ar.MapObject(this);
   //serialize the subitems in the document;
   //they will be able to serialize their m_pDoc
   //back pointer
   m_listOfSubItems.Serialize(ar);

}


//SubItem.h
class CSubItem : public CObject
{
public:
   CSubItem(CMyDocument * pDoc)
```

```
        { m_pDoc = pDoc; }

    // back pointer to owning document
    CMyDocument* m_pDoc;
    WORD m_i; // other item data

    virtual void Serialize(CArchive& ar);
};

//SubItem.cpp
void CSubItem::Serialize(CArchive& ar)

{
    if (ar.IsStoring( ) )
    {
        // will serialize a reference
        // to the "mapped" document pointer
        ar << (CObject *)m_pDoc;
        ar << m_l;
    }
    else
    {
        // Will load a reference to
        // the "mapped" document pointer
        ar >> (CObject *&) m_pDoc;
        ar >> m_l;
    }
}
```

See Also

Reference

CArchive Class

Hierarchy Chart

CArchive::ReadObject

CArchive::WriteObject

Other Resources

CArchive Members

CArchive::Read

Reads a specified number of bytes from the archive.

```
UINT Read(
   void* lpBuf,
   UINT nMax
);
```

Parameters

lpBuf

A pointer to a user-supplied buffer that is to receive the data read from the archive.

nMax

An unsigned integer specifying the number of bytes to be read from the archive.

Return Value

An unsigned integer containing the number of bytes actually read. If the return value is less than the number requested, the end of file has been reached. No exception is thrown on the end-of-file condition.

Remarks

The archive does not interpret the bytes.

You can use the **Read** member function within your **Serialize** function for reading ordinary structures that are contained in your objects.

Example

```
extern CArchive ar;
char pb[100];
UINT nr = ar.Read( pb, 100 );
```

See Also

Reference

CArchive Class

Hierarchy Chart

Other Resources

CArchive Members

CArchive::ReadClass

Call this member function to read a reference to a class previously stored with WriteClass.

```
CRuntimeClass* ReadClass(
   const CRuntimeClass* pClassRefRequested = NULL,
   UINT* pSchema = NULL,
   DWORD* pObTag = NULL
);
```

Parameters

pClassRefRequested

A pointer to the CRuntimeClass structure that corresponds to the class reference requested. Can be **NULL**.

pSchema

A pointer to a schema of the run-time class previously stored.

pObTag

A number that refers to an object's unique tag. Used internally by the implementation of ReadObject. Exposed for advanced programming only; *pObTag* normally should be **NULL**.

Return Value

A pointer to the CRuntimeClass structure.

Remarks

If *pClassRefRequested* is not **NULL**, **ReadClass** verifies that the archived class information is compatible with your runtime class. If it is not compatible, **ReadClass** will throw a CArchiveException.

Your runtime class must use DECLARE_SERIAL and IMPLEMENT_SERIAL; otherwise, **ReadClass** will throw a CNotSupportedException.

If *pSchema* is **NULL**, the schema of the stored class can be retrieved by calling CArchive::GetObjectSchema; otherwise, ***pSchema* will contain the schema of the run-time class that was previously stored.

You can use SerializeClass instead of **ReadClass**, which handles both reading and writing of the class reference.

Example

See the example for CArchive::WriteClass.

See Also

Reference

CArchive Class
Hierarchy Chart
CArchive::WriteClass
CArchive::GetObjectSchema
CArchive::SetObjectSchema
CArchiveException Class
CNotSupportedException Class
CArchive::SerializeClass

Other Resources

CArchive Members

CArchive::ReadObject

Reads object data from the archive and constructs an object of the appropriate type.

```
CObject* ReadObject(
   const CRuntimeClass* pClass
);
```

Parameters

pClass

A constant pointer to the CRuntimeClass structure that corresponds to the object you expect to read.

Return Value

A CObject pointer that must be safely cast to the correct derived class by using CObject::IsKindOf.

Remarks

This function is normally called by the **CArchive** extraction (**>>**) operator overloaded for a CObject pointer. **ReadObject**, in turn, calls the **Serialize** function of the archived class.

If you supply a nonzero *pClass* parameter, which is obtained by the RUNTIME_CLASS macro, then the function verifies the run-time class of the archived object. This assumes you have used the **IMPLEMENT_SERIAL** macro in the implementation of the class.

Example

See the example for CArchive::WriteObject.

See Also

Reference

CArchive Class
Hierarchy Chart
CArchive::WriteObject
CObject::IsKindOf

Other Resources

CArchive Members

CArchive::ReadString

Call this member function to read text data into a buffer from the file associated with the **CArchive** object.

```
BOOL ReadString(
   CString& rString
);
LPTSTR ReadString(
   LPTSTR lpsz,
   UINT nMax
);
```

Parameters

rString

A reference to a CString that will contain the resultant string after it is read from the file associated with the CArchive object.

lpsz

Specifies a pointer to a user-supplied buffer that will receive a null-terminated text string.

nMax

Specifies the maximum number of characters to read. Should be one less than the size of the *lpsz* buffer.

Return Value

In the version that returns **Bool**, **TRUE** if successful; **FALSE** otherwise.

In the version that returns an **LPTSTR**, a pointer to the buffer containing the text data; **NULL** if end-of-file was reached.

Remarks

In the version of the member function with the *nMax* parameter, the buffer will hold up to a limit of *nMax* - 1 characters. Reading is stopped by a carriage return-linefeed pair. Trailing newline characters are always removed. A null character ('\0') is appended in either case.

CArchive::Read is also available for text-mode input, but it does not terminate on a carriage return-linefeed pair.

Example

See the example for CArchive::WriteString.

See Also

Reference

CArchive Class
Hierarchy Chart
CArchive::Read
CArchive::Write
CArchive::WriteString
CArchiveException Class

Other Resources

CArchive Members

CArchive::SerializeClass

Call this member function when you want to store and load the version information of a base class.

```
void SerializeClass(
   const CRuntimeClass* pClassRef
);
```

Parameters

pClassRef

A pointer to a run-time class object for the base class.

Remarks

SerializeClass reads or writes the reference to a class to the **CArchive** object, depending on the direction of the **CArchive**. Use **SerializeClass** in place of ReadClass and WriteClass as a convenient way to serialize base-class objects; **SerializeClass** requires less code and fewer parameters.

Like **ReadClass**, **SerializeClass** verifies that the archived class information is compatible with your runtime class. If it is not compatible, **SerializeClass** will throw a CArchiveException.

Your runtime class must use DECLARE_SERIAL and IMPLEMENT_SERIAL; otherwise, **SerializeClass** will throw a CNotSupportedException.

Use the RUNTIME_CLASS macro to retrieve the value for the *pRuntimeClass* parameter. The base class must have used the IMPLEMENT_SERIAL macro.

Example

```
class CBaseClass : public CObject { ... };
class CDerivedClass : public CBaseClass { ... };
void CDerivedClass::Serialize(CArchive& ar)
{
   if (ar.IsStoring())
   {
      //normal code for storing contents
      //of this object
   }
   else
   {
      //normal code for reading contents
      //of this object
   }

   //allow the base class to serialize along
   //with its version information
   ar.SerializeClass(RUNTIME_CLASS(CBaseClass));
   CBaseClass::Serialize(ar);
}
```

See Also

Reference

CArchive Class
Hierarchy Chart
CArchive::ReadClass
CArchive::WriteClass
CArchive::GetObjectSchema
CArchive::SetObjectSchema
CArchiveException Class
CNotSupportedException Class

Other Resources

CArchive Members

CArchive::SetLoadParams

Call **SetLoadParams** when you are going to read a large number of **CObject**-derived objects from an archive.

```
void SetLoadParams(
   UINT nGrowBy = 1024
);
```

Parameters

nGrowBy

The minimum number of element slots to allocate if a size increase is necessary.

Remarks

CArchive uses a load array to resolve references to objects stored in the archive. **SetLoadParams** allows you to set the size to which the load array grows.

You must not call **SetLoadParams** after any object is loaded, or after MapObject or ReadObject is called.

Example

```
class CMyLargeDocument : public CDocument { ... };
void CMyLargeDocument::Serialize(CArchive& ar)
{
   if (ar.IsStoring())
      ar.SetStoreParams();  // use large defaults
   else
      ar.SetLoadParams();

   if (ar.IsStoring())
   {
      // code for storing CMyLargeDocument
   }
   else
   {
      // code for loading CMyLargeDocument
   }
}
```

See Also

Reference

CArchive Class

Hierarchy Chart

CArchive::SetStoreParams

Other Resources

CArchive Members

CArchive::SetObjectSchema

Call this member function to set the object schema stored in the archive object to *nSchema*.

```
void SetObjectSchema(
   UINT nSchema
);
```

Parameters

nSchema

Specifies the object's schema.

Remarks

The next call to GetObjectSchema will return the value stored in *nSchema*.

Use **SetObjectSchema** for advanced versioning; for example, when you want to force a particular version to be read in a **Serialize** function of a derived class.

Example

```
CFile myFile("myfile.dat", CFile::modeCreate | CFile::modeRead);
CArchive ar(&myFile, CArchive::load);

// Set schema to 2 and verify.
ar.SetObjectSchema(2);
ASSERT(ar.GetObjectSchema() == 2);
```

See Also

Reference

CArchive Class
Hierarchy Chart
CArchive::GetObjectSchema

Other Resources

CArchive Members

CArchive::SetStoreParams

Use **SetStoreParams** when storing a large number of **CObject**-derived objects in an archive.

```
void SetStoreParams(
   UINT nHashSize = 2053,
   UINT nBlockSize = 128
);
```

Parameters

nHashSize

The size of the hash table for interface pointer maps. Should be a prime number.

nBlockSize

Specifies the memory-allocation granularity for extending the parameters. Should be a power of 2 for the best performance.

Remarks

SetStoreParams allows you to set the hash table size and the block size of the map used to identify unique objects during the serialization process.

You must not call **SetStoreParams** after any objects are stored, or after MapObject or WriteObject is called.

Example

```
class CMyLargeDocument : public CDocument { ... };
void CMyLargeDocument::Serialize(CArchive& ar)
{
   if (ar.IsStoring())
      ar.SetStoreParams();  // use large defaults
   else
      ar.SetLoadParams();

   if (ar.IsStoring())
   {
      // code for storing CMyLargeDocument
   }
   else
   {
      // code for loading CMyLargeDocument
   }
}
```

See Also

Reference

CArchive Class

Hierarchy Chart

CArchive::SetLoadParams

Other Resources

CArchive Members

CArchive::Write

Writes a specified number of bytes to the archive.

```
void Write(
   const void* lpBuf,
   UINT nMax
);
```

Parameters

lpBuf

A pointer to a user-supplied buffer that contains the data to be written to the archive.

nMax

An integer that specifies the number of bytes to be written to the archive.

Remarks

The archive does not format the bytes.

You can use the **Write** member function within your **Serialize** function to write ordinary structures that are contained in your objects.

Example

```
extern CArchive ar;
char pb[100];
ar.Write( pb, 100 );
```

See Also

Reference

CArchive Class

Hierarchy Chart

CArchive::Read

Other Resources

CArchive Members

CArchive::WriteClass

Use **WriteClass** to store the version and class information of a base class during serialization of the derived class.

```
void WriteClass(
   const CRuntimeClass* pClassRef
);
```

Parameters

pClassRef

A pointer to the CRuntimeClass structure that corresponds to the class reference requested.

Remarks

WriteClass writes a reference to the CRuntimeClass for the base class to the **CArchive**. Use CArchive::ReadClass to retrieve the reference.

WriteClass verifies that the archived class information is compatible with your runtime class. If it is not compatible, **WriteClass** will throw a CArchiveException.

Your runtime class must use DECLARE_SERIAL and IMPLEMENT_SERIAL; otherwise, **WriteClass** will throw a CNotSupportedException.

You can use SerializeClass instead of **WriteClass**, which handles both reading and writing of the class reference.

Example

```
CFile myFile("myfile.dat", CFile::modeCreate | CFile::modeReadWrite);

// Create a storing archive.
CArchive arStore(&myFile, CArchive::store);

// Store the class CAge in the archive.
arStore.WriteClass( RUNTIME_CLASS(CAge) );

// Close the storing archive.
arStore.Close();

// Create a loading archive.
myFile.SeekToBegin();
CArchive arLoad(&myFile, CArchive::load);

// Load a class from the archive.
CRuntimeClass* pClass = arLoad.ReadClass( );
if (!pClass->IsDerivedFrom(RUNTIME_CLASS(CAge)))
      arLoad.Abort();
```

See Also

Reference

CArchive Class
Hierarchy Chart
CArchive::ReadClass
CArchive::GetObjectSchema
CArchive::SetObjectSchema
CArchive::SerializeClass
CArchiveException Class
CNotSupportedException Class
CObList::CObList

Other Resources

CArchive Members

CArchive::WriteObject

Stores the specified **CObject** to the archive.

```
void WriteObject(
   const CObject* pOb
);
```

Parameters

pOb

A constant pointer to the object being stored.

Remarks

This function is normally called by the **CArchive** insertion (**<<**) operator overloaded for **CObject**. **WriteObject**, in turn, calls the **Serialize** function of the archived class.

You must use the **IMPLEMENT_SERIAL** macro to enable archiving. **WriteObject** writes the ASCII class name to the archive. This class name is validated later during the load process. A special encoding scheme prevents unnecessary duplication of the class name for multiple objects of the class. This scheme also prevents redundant storage of objects that are targets of more than one pointer.

The exact object encoding method (including the presence of the ASCII class name) is an implementation detail and could change in future versions of the library.

Note

Finish creating, deleting, and updating all your objects before you begin to archive them. Your archive will be corrupted if you mix archiving with object modification.

Example

For a definition of the class `CAge`, see the example for CObList::CObList.

```
CFile myFile("myfile.dat", CFile::modeCreate | CFile::modeReadWrite);
CAge  age(21), *pAge;

// Create a storing archive.
CArchive arStore(&myFile, CArchive::store);

// Write the object to the archive
arStore.WriteObject( &age );

// Close the storing archive
arStore.Close();

// Create a loading archive.
myFile.SeekToBegin();
CArchive arLoad(&myFile, CArchive::load);

// Verify the object is in the archive.
pAge = (CAge*) arLoad.ReadObject( RUNTIME_CLASS(CAge) );
ASSERT( age == *pAge );
```

See Also

Reference

CArchive Class

Hierarchy Chart

CArchive::ReadObject

Other Resources

CArchive Members

CArchive::WriteString

Use this member function to write data from a buffer to the file associated with the **CArchive** object.

```
void WriteString(
   LPCTSTR lpsz
);
```

Parameters

lpsz

Specifies a pointer to a buffer containing a null-terminated text string.

Remarks

The terminating null character ('\0') is not written to the file; nor is a newline automatically written.

WriteString throws an exception in response to several conditions, including the disk-full condition.

Write is also available, but rather than terminating on a null character, it writes the requested number of bytes to the file.

Example

```
CFile myFile("myfile", CFile::modeCreate | CFile::modeReadWrite);
CString str1="String1", str2="String2", str;

// Create a storing archive.
CArchive arStore(&myFile, CArchive::store);

// Write str1 and str2 to the archive
arStore.WriteString( str1 );
arStore.WriteString( "\n" );
arStore.WriteString( str2 );
arStore.WriteString( "\n" );

// Close the storing archive
arStore.Close();

// Create a loading archive.
myFile.SeekToBegin();
CArchive arLoad(&myFile, CArchive::load);

// Verify the two strings are in the archive.
arLoad.ReadString( str );
ASSERT( str == str1 );
arLoad.ReadString( str );
ASSERT( str == str2 );
```

See Also

Reference

CArchive Class

Hierarchy Chart

Other Resources

CArchive Members

CArchive Operators

For information about the operators in **CArchive**, see CArchive Class Members.

CArchive::operator <<

Stores the indicated object or primitive type to the archive.

```
friend CArchive& operator <<(
   CArchive& ar,
   const CObject* pOb
);
throw(
   CArchiveException*,
   CFileException*
);
CArchive& AFXAPI operator <<(
   CArchive& ar,
   const RECT& rect
);
CArchive& AFXAPI operator <<(
   CArchive& ar,
   POINT point
);
CArchive& AFXAPI operator <<(
  CArchive& ar,
   SIZE size
);
template<
   typename BaseType,
   class StringTraits
>
CArchive& operator<<(
   const ATL::CStringT<
      BaseType,
      StringTraits
   >& str
);
CArchive& operator <<(
   BYTE by
);
CArchive& operator <<(
   WORD w
);
CArchive& operator <<(
   LONG l
);
CArchive& operator <<(
   DWORD dw
);
CArchive& operator <<(
   float f
);
CArchive& operator <<(
   double d
);
CArchive& operator <<(
   int i
);
CArchive& operator <<(
   short w
);
CArchive& operator <<(
   char ch
);
CArchive& operator<<(
   wchar_t ch
);
```

```
CArchive& operator <<(
   unsigned u
);
CArchive& operator <<(
   bool b
);
CArchive& operator<<(
   ULONGLONG dwdw
);
CArchive& operator<<(
   LONGLONG dwdw
);
```

Return Value

A **CArchive** reference that enables multiple extraction operators on a single line.

Remarks

The last two versions above are specifically for storing 64-bit integers.

If you used the **IMPLEMENT_SERIAL** macro in your class implementation, then the insertion operator overloaded for **CObject** calls the protected **WriteObject**. This function, in turn, calls the **Serialize** function of the class.

The CStringT insertion operator (<<) supports diagnostic dumping and storing to an archive.

Example

This example demonstrates the use of the **CArchive** insertion operator << with the **int** and **long** types.

```
long l;
int i;
extern CArchive ar;
if( ar.IsStoring() )
  ar << l << i;
```

This example 2 demonstrates the use of the **CArchive** insertion operator << with the **CStringT** type.

```
   extern CArchive ar;
   CStringT s( "abc" );
#ifdef _DEBUG
   afxDump << s;  // Prints the value (abc)
   afxDump << &s;  // Prints the address
#endif
```

See Also

Reference

CArchive Class
Hierarchy Chart
CArchive::WriteObject
CObject::Serialize
CStringT Class
CDumpContext Class

Other Resources

CArchive Members

CArchive::operator >>

Loads the indicated object or primitive type from the archive.

```
friend CArchive& operator >>(
   CArchive& ar,
   CObject *& pOb
);
throw(
   CArchiveException*,
   CFileException*,
   CMemoryException*
);
friend CArchive& operator >>(
   CArchive& ar,
   const CObject *& pOb
);
throw(
   CArchiveException*,
   CFileException*,
   CMemoryException*
);
CArchive& AFXAPI operator >>(
   CArchive& ar,
   const RECT& rect
);
CArchive& AFXAPI operator >>(
   CArchive& ar,
   POINT point
);
CArchive& AFXAPI operator >>(
  CArchive& ar,
   SIZE size
);
template<
   typename BaseType,
   class StringTraits
>
CArchive& operator>>(
   ATL::CStringT<
      BaseType,
      StringTraits
   >& str
);
CArchive& operator >>(
   BYTE& by
);
CArchive& operator >>(
   WORD& w
);
CArchive& operator >>(
   int& i
);
CArchive& operator >>(
   LONG& l
);
CArchive& operator >>(
   DWORD& dw
);
CArchive& operator >>(
   float& f
);
CArchive& operator >>(
   double& d
```

```
);
CArchive& operator >>(
   short& w
);
CArchive& operator >>(
   char& ch
);
CArchive& operator>>(
   wchar_t& ch);
CArchive& operator >>(
   unsigned& u
);
CArchive& operator >>(
   bool& b
);
CArchive& operator >>(
   ULONGLONG& dwdw
);
CArchive& operator >>(
   LONGLONG& dwdw
);
```

Return Value

A **CArchive** reference that enables multiple insertion operators on a single line.

Remarks

The last two versions above are specifically for loading 64-bit integers.

If you used the **IMPLEMENT_SERIAL** macro in your class implementation, then the extraction operators overloaded for **CObject** call the protected **ReadObject** function (with a nonzero run-time class pointer). This function, in turn, calls the **Serialize** function of the class.

The CStringT extraction operator (>>) supports loading from an archive.

Example

This example demonstrates the use of the **CArchive** extraction operator >> with the **int** type.

```
int i;
extern CArchive ar;
if( ar.IsLoading() )
  ar >> i;
```

This example demonstrates the use of the **CArchive** insertion and extraction operators << and >> with the **CStringT** type.

```
   extern CArchive ar;
   CStringT s( "abc" );

   if( ar.IsLoading() )
      ar >> s;
   else
      ar << s;
```

See Also

Reference

CArchive Class
Hierarchy Chart
CArchive::ReadObject
CObject::Serialize

Other Resources

CArchive Members

CArchive Data Members

For information about the data members in **CArchive**, see CArchive Class Members.

CArchive::m_pDocument

Set to **NULL** by default, this pointer to a **CDocument** can be set to anything the user of the **CArchive** instance wants.

```
CDocument* m_pDocument;
```

Remarks

A common usage of this pointer is to convey additional information about the serialization process to all objects being serialized. This is achieved by initializing the pointer with the document (a **CDocument**-derived class) that is being serialized, in such a way that objects within the document can access the document if necessary. This pointer is also used by **COleClientItem** objects during serialization.

The framework sets **m_pDocument** to the document being serialized when a user issues a File Open or Save command. If you serialize an Object Linking and Embedding (OLE) container document for reasons other than File Open or Save, you must explicitly set **m_pDocument**. For example, you would do this when serializing a container document to the Clipboard.

Example

```
CFile myFile("myfile.dat", CFile::modeCreate | CFile::modeWrite);
CArchive ar(&myFile, CArchive::store);

// Serialize the document to the archive.
if (ar.m_pDocument != NULL)
   ar.m_pDocument->Serialize(ar);
```

See Also

Reference

CArchive Class
Hierarchy Chart
CDocument Class
COleClientItem Class

Other Resources

CArchive Members

CArchiveException Class

Represents a serialization exception condition

```
class CArchiveException : public CException
```

Remarks

The **CArchiveException** class includes a public data member that indicates the cause of the exception.

CArchiveException objects are constructed and thrown inside CArchive member functions. You can access these objects within the scope of a **CATCH** expression. The cause code is independent of the operating system. For more information about exception processing, see Exception Handling (MFC).

Requirements

Header: afx.h

See Also

Reference

CException Class
Hierarchy Chart
CArchive Class
AfxThrowArchiveException

Other Resources

CArchiveException Members
Exception Processing

CArchiveException Members

Base Class Members

CObject Members

CException Members

Data Members

m_cause	Indicates the exception cause.
m_strFileName	Specifies the name of the file for this exception condition.

Construction

CArchiveException	Constructs a **CArchiveException** object.

See Also

Reference

CArchiveException Class

Hierarchy Chart

CArchiveException Member Functions

For information about the member functions in **CArchiveException**, see CArchiveException Class Members.

CArchiveException::CArchiveException

Constructs a **CArchiveException** object, storing the value of *cause* in the object.

```
CArchiveException(
   int cause = CArchiveException::none,
   LPCTSTR lpszArchiveName = NULL
);
```

Parameters

cause

An enumerated type variable that indicates the reason for the exception. For a list of the enumerators, see the m_cause data member.

lpszArchiveName

Points to a string containing the name of the **CArchive** object causing the exception.

Remarks

You can create a **CArchiveException** object on the heap and throw it yourself or let the global function AfxThrowArchiveException handle it for you.

Do not use this constructor directly; instead, call the global function **AfxThrowArchiveException**.

See Also

Reference

CArchiveException Class

Hierarchy Chart

Other Resources

CArchiveException Members

CArchiveException Data Members

For information about the data members in **CArchiveException**, see CArchiveException Class Members.

CArchiveException::m_cause

Specifies the cause of the exception.

```
int m_cause;
```

Remarks

This data member is a public variable of type **int**. Its values are defined by a **CArchiveException** enumerated type. The enumerators and their meanings are as follows:

- **CArchiveException::none** No error occurred.
- **CArchiveException::genericException** Unspecified error.
- **CArchiveException::readOnly** Tried to write into an archive opened for loading.
- **CArchiveException::endOfFile** Reached end of file while reading an object.
- **CArchiveException::writeOnly** Tried to read from an archive opened for storing.
- **CArchiveException::badIndex** Invalid file format.
- **CArchiveException::badClass** Tried to read an object into an object of the wrong type.
- **CArchiveException::badSchema** Tried to read an object with a different version of the class.

Note
These **CArchiveException** cause enumerators are distinct from the **CFileException** cause enumerators.

Note
CArchiveException::generic is deprecated. Use **genericException** instead. If **generic** is used in an application and b uilt with /clr, there will be syntax errors that are not easy to decipher.

See Also

Reference

CArchiveException Class

Hierarchy Chart

Other Resources

CArchiveException Members

CArchiveException::m_strFileName

Specifies the name of the file for this exception condition.

```
CString m_strFileName;
```

See Also

Reference

CArchiveException Class

Hierarchy Chart

Other Resources

CArchiveException Members

CArray Class

Supports arrays that are similar to C arrays, but can dynamically shrink and grow as necessary.

```
template < class TYPE, class ARG_TYPE = const TYPE& >
class CArray :
   public CObject
```

Parameters

TYPE

Template parameter specifying the type of objects stored in the array. TYPE is a parameter that is returned by **CArray**.

ARG _ TYPE

Template parameter specifying the argument type used to access objects stored in the array. Often a reference to *TYPE*. ARG_TYPE is a parameter that is passed to **CArray**.

Remarks

Array indexes always start at position 0. You can decide whether to fix the upper bound or allow the array to expand when you add elements past the current bound. Memory is allocated contiguously to the upper bound, even if some elements are null.

As with a C array, the access time for a **CArray** indexed element is constant and is independent of the array size.

Tip
Before using an array, use SetSize to establish its size and allocate memory for it. If you do not use **SetSize**, adding elements to your array causes it to be frequently reallocated and copied. Frequent reallocation and copying are inefficient and can frag ment memory.

If you need a dump of individual elements in an array, you must set the depth of the CDumpContext object to 1 or greater.

Certain member functions of this class call global helper functions that must be customized for most uses of the **CArray** class. See the topic Collection Class Helpers in the MFC Macros and Globals section.

Array class derivation is similar to list derivation.

For more information on using **CArray**, see the article Collections.

Requirements

Header: afxtempl.h

See Also

Tasks

COLLECT Sample: Illustrates MFC Collection Classes

Reference

CObject Class

Hierarchy Chart

CObArray Class

Other Resources

CArray Members

Collection Class Helpers

CArray Members

Base Class Members

CObject Members

Construction

CArray	Constructs an empty array.

Attributes

GetCount	Gets the number of elements in this array.
GetSize	Gets the number of elements in this array.
GetUpperBound	Returns the largest valid index.
IsEmpty	Determines whether the array is empty.
SetSize	Sets the number of elements to be contained in this array.

Operations

FreeExtra	Frees all unused memory above the current upper bound.
RemoveAll	Removes all the elements from this array.
RelocateElements	Relocates data to a new buffer when the array should grow or shrink.

Element Access

ElementAt	Returns a temporary reference to the element pointer within the array.
GetAt	Returns the value at a given index.
GetData	Allows access to elements in the array. Can be **NULL**.
SetAt	Sets the value for a given index; array not allowed to grow.

Growing the Array

Add	Adds an element to the end of the array; grows the array if necessary.
Append	Appends another array to the array; grows the array if necessary
Copy	Copies another array to the array; grows the array if necessary.
SetAtGrow	Sets the value for a given index; grows the array if necessary.

Insertion/Removal

InsertAt	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt	Removes an element at a specific index.

Operators

operator []	Sets or gets the element at the specified index.

See Also

Reference

CArray Class

Hierarchy Chart

CArray Member Functions

For information about the member functions in **CArray**, see CArray Class Members.

CArray::Add

Adds a new element to the end of an array, growing the array by 1.

```
INT_PTR Add(
      ARG_TYPE newElement
);
```

Parameters

ARG_TYPE

Template parameter specifying the type of arguments referencing elements in this array.

newElement

The element to be added to this array.

Return Value

The index of the added element.

Remarks

If SetSize has been used with an *nGrowBy* value greater than 1, then extra memory may be allocated. However, the upper bound will increase by only 1.

Example

```
// example for CArray::Add
CArray<CPoint,CPoint> ptArray;

CPoint pt(10,20);
ptArray.Add(pt);             // Element 0
ptArray.Add(CPoint(30,40));  // Element 1
```

See Also

Reference

CArray Class
Hierarchy Chart
CArray::SetAt
CArray::SetAtGrow
CArray::InsertAt
CArray::operator []

Other Resources

CArray Members

CArray::Append

Call this member function to add the contents of one array to the end of another.

```
INT_PTR Append(
   const CArray& src
);
```

Parameters

src

Source of the elements to be appended to an array.

Return Value

The index of the first appended element.

Remarks

The arrays must be of the same type.

If necessary, **Append** may allocate extra memory to accommodate the elements appended to the array.

Example

```
CArray<CPoint,CPoint> myArray1, myArray2;

// Add elements to the second array.
myArray2.Add( CPoint(11, 22) );
myArray2.Add( CPoint(12, 42) );

// Add elements to the first array and also append the second array.
myArray1.Add( CPoint(1, 2) );
myArray1.Append( myArray2 );
```

See Also

Reference

CArray Class

Hierarchy Chart

CArray::Copy

Other Resources

CArray Members

CArray::CArray

Constructs an empty array.

```
CArray( );
```

Remarks

The array grows one element at a time.

Example

```
// declares an array of points
CArray<CPoint,CPoint> ptArray;
```

See Also

Reference

CArray Class

Hierarchy Chart

CObArray::CObArray

Other Resources

CArray Members

CArray::Copy

Use this member function to copy the elements of one array to another.

```
void Copy(
   const CArray& src
);
```

Parameters

src

Source of the elements to be copied to an array.

Remarks

Call this member function to overwrite the elements of one array with the elements of another array.

Copy does not free memory; however, if necessary, **Copy** may allocate extra memory to accommodate the elements copied to the array.

Example

```
CArray<CPoint,CPoint> myArray1, myArray2;

// Add elements to the second array.
myArray2.Add( CPoint(11, 22) );
myArray2.Add( CPoint(12, 42) );

// Copy the elements from the second array to the first.
myArray1.Copy( myArray2 );
```

See Also

Reference

CArray Class

Hierarchy Chart

CArray::Append

Other Resources

CArray Members

CArray::ElementAt

Returns a temporary reference to the specified element within the array.

```
TYPE& ElementAt(
   INT_PTR nIndex
);
const TYPE& ElementAt(
   INT_PTR nIndex
) const;
```

Parameters

nIndex

An integer index that is greater than or equal to 0 and less than or equal to the value returned by GetUpperBound.

Return Value

A reference to an array element.

Remarks

It is used to implement the left-side assignment operator for arrays.

Example

See the example for GetSize.

See Also

Reference

CArray Class
Hierarchy Chart
CArray::operator []

Other Resources

CArray Members

CArray::FreeExtra

Frees any extra memory that was allocated while the array was grown.

```
void FreeExtra( );
```

Remarks

This function has no effect on the size or upper bound of the array.

Example

See the example for GetData.

See Also

Reference

CArray Class

Hierarchy Chart

Other Resources

CArray Members

CArray::GetAt

Returns the array element at the specified index.

```
TYPE& GetAt(
   INT_PTR nIndex
);
const TYPE& GetAt(
   INT_PTR nIndex
) const;
```

Parameters

TYPE

Template parameter specifying the type of the array elements.

nIndex

An integer index that is greater than or equal to 0 and less than or equal to the value returned by GetUpperBound.

Return Value

The array element currently at this index.

Remarks

Passing a negative value or a value greater than the value returned by **GetUpperBound** will result in a failed assertion.

Example

```
CArray<CPoint,CPoint> myArray;
CPoint pt;

// Add elements to the array.
for (int i=0;i < 10;i++)
   myArray.Add( CPoint(i, 2*i) );

// Modify all the points in the array.
for (i=0;i <= myArray.GetUpperBound();i++)
{
   pt = myArray.GetAt(i);
   pt.x = 0;
   myArray.SetAt(i, pt);
}
```

See Also

Reference

CArray Class
Hierarchy Chart
CArray::SetAt
CArray::operator []

Other Resources

CArray Members

CArray::GetCount

Returns the number of array elements.

```
INT_PTR GetCount( ) const;
```

Return Value

The number of items in the array.

Remarks

Call this method to retrieve the number of elements in the array. Because indexes are zero-based, the size is 1 greater than the largest index. Calling this method will generate the same result as the CArray::GetSize method.

Example

```
CArray<CPoint,CPoint> myArray;

// Modify all the points in the array.
for (i=0;i < myArray.GetCount();i++)
{
   CPoint& pt = myArray.ElementAt(i);
   pt.x = 0;
}
```

See Also

Reference

CArray Class
Hierarchy Chart
CArray::GetUpperBound
CArray::SetSize
CArray::GetSize

Other Resources

CArray Members

CArray::GetData

Use this member function to gain direct access to the elements in an array.

```
const TYPE* GetData( ) const;
TYPE* GetData( );
```

Parameters

TYPE

Template parameter specifying the type of the array elements.

Return Value

A pointer to an array element.

Remarks

If no elements are available, **GetData** returns a null value.

While direct access to the elements of an array can help you work more quickly, use caution when calling **GetData**; any errors you make directly affect the elements of your array.

Example

```
CArray<CPoint,CPoint> myArray;
int i;

// Allocate memory for at least 32 elements.
myArray.SetSize(32, 128);

// Add elements to the array.
CPoint* pPt = (CPoint*) myArray.GetData();
for (i=0;i < 32;i++,pPt++)
   *pPt = CPoint(i, 2*i);

// Only keep first 5 elements and free extra (unused) bytes.
myArray.SetSize(5, 128);
myArray.FreeExtra();

#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "myArray: " << &myArray << "\n";
#endif
```

See Also

Reference

CArray Class
Hierarchy Chart
CArray::GetAt
CArray::SetAt
CArray::ElementAt

Other Resources

CArray Members

CArray::GetSize

Returns the size of the array.

```
INT_PTR GetSize( ) const;
```

Remarks

Because indexes are zero-based, the size is 1 greater than the largest index. Calling this method will generate the same result as the CArray::GetCount method.

Example

```
CArray<CPoint,CPoint> myArray;

// Add elements to the array.
for (int i=0;i < 10;i++)
      myArray.Add( CPoint(i, 2*i) );

// Modify all the points in the array.
for (i=0;i < myArray.GetSize();i++)
{
   CPoint& pt = myArray.ElementAt(i);
   pt.x = 0;
}
```

See Also

Reference

CArray Class
Hierarchy Chart
CArray::GetUpperBound
CArray::GetCount

Other Resources

CArray Members

CArray::GetUpperBound

Returns the current upper bound of this array.

```
INT_PTR GetUpperBound( ) const;
```

Remarks

Because array indexes are zero-based, this function returns a value 1 less than **GetSize**.

The condition **GetUpperBound( )** = –1 indicates that the array contains no elements.

Example

See the example for CArray::GetAt.

See Also

Reference

CArray Class
Hierarchy Chart
CArray::GetSize
CArray::SetSize

Other Resources

CArray Members

CArray::InsertAt

The first version of **InsertAt** inserts one element (or multiple copies of an element) at a specified index in an array.

```
void InsertAt(
   INT_PTR nIndex,
      ARG_TYPE newElement,
   INT_PTR nCount = 1
);
void InsertAt(
   INT_PTR nStartIndex,
   CArray* pNewArray
);
```

Parameters

nIndex

An integer index that may be greater than the value returned by **GetUpperBound**.

ARG_TYPE

Template parameter specifying the type of elements in this array.

newElement

The element to be placed in this array.

nCount

The number of times this element should be inserted (defaults to 1).

nStartIndex

An integer index that may be greater than the value returned by GetUpperBound.

pNewArray

Another array that contains elements to be added to this array.

Remarks

In the process, it shifts up (by incrementing the index) the existing element at this index, and it shifts up all the elements above it.

The second version inserts all the elements from another **CArray** collection, starting at the *nStartIndex* position.

The **SetAt** function, in contrast, replaces one specified array element and does not shift any elements.

Example

```
// example for CArray::InsertAt

CArray<CPoint,CPoint> ptArray;

ptArray.Add(CPoint(10,20));   // Element 0
ptArray.Add(CPoint(30,40));   // Element 1 (will become element 2)
ptArray.InsertAt(1, CPoint(50,60));   // New element 1
```

See Also

Reference

CArray Class
Hierarchy Chart
CArray::GetUpperBound
CArray::SetAt
CArray::RemoveAt

Other Resources

CArray Members

CArray::IsEmpty

Determines whether the array is empty.

```
BOOL IsEmpty( ) const;
```

Return Value

Nonzero if the array contains no elements; otherwise 0.

See Also

Reference

CArray Class

Hierarchy Chart

Other Resources

CArray Members

CArray::RelocateElements

Relocates data to a new buffer when the array should grow or shrink.

```
template<class TYPE, class ARG_TYPE>
AFX_INLINE void CArray<TYPE, ARG_TYPE>::RelocateElements(
   TYPE* pNewData,
   const TYPE* pData,
   INT_PTR nCount
);
```

Parameters

pNewData

A new buffer for the array of elements.

pData

The old array of elements.

nCount

Number of elements in the old array.

Remarks

pNewData is always large enough to hold all the *pData* elements.

The CArray implementation uses this method to copy the old data to a new buffer when the array should grow or shrink (when SetSize or FreeExtra are called). The default implementation just copies the data.

For arrays in which an element contains a pointer to one of its own members, or another structure contains a pointer to one of the array elements, the pointers are not updated in plain copy. In this case, you can correct pointers by implementing a specialization of **RelocateElements** with the relevant types. You are also responsible for data copying.

See Also

Reference

CArray::SetSize
CArray::FreeExtra

CArray::RemoveAll

Removes all the elements from this array.

```
void RemoveAll( );
```

Remarks

If the array is already empty, the function still works.

Example

```
CArray<CPoint,CPoint> myArray;

// Add elements to the array.
for (int i=0;i < 10;i++)
   myArray.Add( CPoint(i, 2*i) );

myArray.RemoveAll();

#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "myArray: " << &myArray << "\n";
#endif
```

See Also

Reference

CArray Class

Hierarchy Chart

Other Resources

CArray Members

CArray::RemoveAt

Removes one or more elements starting at a specified index in an array.

```
void RemoveAt(
   INT_PTR nIndex,
   INT_PTR nCount = 1
);
```

Parameters

nIndex

An integer index that is greater than or equal to 0 and less than or equal to the value returned by GetUpperBound.

nCount

The number of elements to remove.

Remarks

In the process, it shifts down all the elements above the removed element(s). It decrements the upper bound of the array but does not free memory.

If you try to remove more elements than are contained in the array above the removal point, then the Debug version of the library asserts.

Example

```
CArray<CPoint,CPoint> myArray;

// Add elements to the array.
for (int i=0;i < 10;i++)
   myArray.Add( CPoint(i, 2*i) );

myArray.RemoveAt(5);

#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "myArray: " << &myArray << "\n";
#endif
```

See Also

Reference

CArray Class
Hierarchy Chart
CArray::SetAt
CArray::SetAtGrow
CArray::InsertAt

Other Resources

CArray Members

CArray::SetAt

Sets the array element at the specified index.

```
void SetAt(
   INT_PTR nIndex,
      ARG_TYPE newElement
);
```

Parameters

nIndex

An integer index that is greater than or equal to 0 and less than or equal to the value returned by GetUpperBound.

ARG_TYPE

Template parameter specifying the type of arguments used for referencing array elements.

newElement

The new element value to be stored at the specified position.

Remarks

SetAt will not cause the array to grow. Use SetAtGrow if you want the array to grow automatically.

You must ensure that your index value represents a valid position in the array. If it is out of bounds, then the Debug version of the library asserts.

Example

See the example for GetAt.

See Also

Reference

CArray Class
Hierarchy Chart
CArray::GetAt
CArray::SetAtGrow
CArray::ElementAt
CArray::operator []

Other Resources

CArray Members

CArray::SetAtGrow

Sets the array element at the specified index.

```
void SetAtGrow(
   INT_PTR nIndex,
      ARG_TYPE newElement
);
```

Parameters

nIndex

An integer index that is greater than or equal to 0.

ARG_TYPE

Template parameter specifying the type of elements in the array.

newElement

The element to be added to this array. A **NULL** value is allowed.

Remarks

The array grows automatically if necessary (that is, the upper bound is adjusted to accommodate the new element).

Example

```
// example for CArray::SetAtGrow
CArray<CPoint,CPoint> ptArray;

ptArray.Add(CPoint(10,20));   // Element 0
ptArray.Add(CPoint(30,40));   // Element 1
                     // Element 2 deliberately skipped
ptArray.SetAtGrow(3, CPoint(50,60));   // Element 3
```

See Also

Reference

CArray Class
Hierarchy Chart
CArray::GetAt
CArray::SetAt
CArray::ElementAt
CArray::operator []

Other Resources

CArray Members

CArray::SetSize

Establishes the size of an empty or existing array; allocates memory if necessary.

```
void SetSize(
   INT_PTR nNewSize,
   INT_PTR nGrowBy = -1
);
```

Parameters

nNewSize

The new array size (number of elements). Must be greater than or equal to 0.

nGrowBy

The minimum number of element slots to allocate if a size increase is necessary.

Remarks

If the new size is smaller than the old size, then the array is truncated and all unused memory is released.

Use this function to set the size of your array before you begin using the array. If you do not use **SetSize**, adding elements to your array causes it to be frequently reallocated and copied. Frequent reallocation and copying are inefficient and can fragment memory.

The *nGrowBy* parameter affects internal memory allocation while the array is growing. Its use never affects the array size as reported by GetSize and GetUpperBound. If the default value is used, MFC allocates memory in a way calculated to avoid memory fragmentation and optimize efficiency for most cases.

Example

See the example for GetData.

See Also

Reference

CArray Class
Hierarchy Chart
CArray::GetUpperBound
CArray::GetSize

Other Resources

CArray Members

CArray Operators

For information about the operators in **CArray**, see CArray Class Members.

CArray::operator []

These subscript operators are a convenient substitute for the SetAt and GetAt functions.

```
TYPE& operator[](
   INT_PTR nIndex
);
const TYPE& operator[](
   INT_PTR nIndex
) const;
```

Parameters

TYPE

Template parameter specifying the type of elements in this array.

nIndex

Index of the element to be accessed.

Remarks

The first operator, called for arrays that are not **const**, may be used on either the right (r-value) or the left (l-value) of an assignment statement. The second, called for **const** arrays, may be used only on the right.

The Debug version of the library asserts if the subscript (either on the left or right side of an assignment statement) is out of bounds.

Example

```
CArray<CPoint,CPoint> myArray;

// Add elements to the array.
for (int i=0;i < 10;i++)
      myArray.Add( CPoint(i, 2*i) );

// Modify all the points in the array.
for (i=0;i <= myArray.GetUpperBound();i++)
{
   myArray[i].x = 0;
}
```

See Also

Reference

CArray Class
Hierarchy Chart
CArray::GetAt
CArray::SetAt
CArray::ElementAt

Other Resources

CArray Members

CAsyncMonikerFile Class

Provides functionality for the use of asynchronous monikers in ActiveX controls (formerly OLE controls).

```
class CAsyncMonikerFile : public CMonikerFile
```

Remarks

Derived from CMonikerFile, which in turn is derived from COleStreamFile, **CAsyncMonikerFile** uses the IMoniker interface to access any data stream asynchronously, including loading files asynchronously from a URL. The files can be datapath properties of ActiveX controls.

Asynchronous monikers are used primarily in Internet-enabled applications and ActiveX controls to provide a responsive user-interface during file transfers. A prime example of this is the use of CDataPathProperty to provide asynchronous properties for ActiveX controls. The **CDataPathProperty** object will repeatedly get a callback to indicate availability of new data during a lengthy property exchange process.

For more information about how to use asynchronous monikers and ActiveX controls in Internet applications, see the following articles:

- Internet First Steps: Asynchronous Monikers
- Internet First Steps: ActiveX Controls

Requirements

Header: afxole.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CMonikerFile Class
Hierarchy Chart
CMonikerFile Class
CDataPathProperty Class
Asynchronous Versus Synchronous Monikers

Other Resources

CAsyncMonikerFile Members

CAsyncMonikerFile Members

Base Class Members

CObject Members

CFile Members

COleStreamFile Members

CMonikerFile Members

Construction

CAsyncMonikerFile	Constructs a **CAsyncMonikerFile** object.

Operations

GetBinding	Retrieves a pointer to the asynchronous transfer binding.
GetFormatEtc	Retrieves the format of the data in the stream.

Overridables

Close	Closes and releases all resources.
CreateBindStatusCallback	Creates a COM object that implements **IBindStatusCallback**.
GetBindInfo	Called by the OLE system library to request information on the type of bind to be created.
GetPriority	Called by the OLE system library to get the priority of the binding.
OnDataAvailable	Called to provide data as it becomes available to the client during asynchronous bind operations.
OnLowResource	Called when resources are low.
OnProgress	Called to indicate progress on the data downloading process.
OnStartBinding	Called when binding is starting up.
OnStopBinding	Called when asynchronous transfer is stopped.
Open	Opens a file asynchronously.

See Also

Reference

CAsyncMonikerFile Class

Hierarchy Chart

CAsyncMonikerFile Member Functions

For information about the member functions in **CAsyncMonikerFile**, see CAsyncMonikerFile Class Members.

CAsyncMonikerFile::CAsyncMonikerFile

Constructs a **CAsyncMonikerFile** object.

```
CAsyncMonikerFile( );
```

Remarks

It does not create the **IBindHost** interface. **IBindHost** is used only if you provide it in the **Open** member function.

For a description of the **IBindHost** interface, see the *Platform SDK*.

See Also

Reference

CAsyncMonikerFile Class
Hierarchy Chart
CDataPathProperty Class
CAsyncMonikerFile::Open

Other Resources

CAsyncMonikerFile Members

CAsyncMonikerFile::Close

Call this function to close and release all resources.

```
virtual void Close( );
```

Remarks

Can be called on unopened or already closed files.

See Also

Reference

CAsyncMonikerFile Class
Hierarchy Chart
CAsyncMonikerFile::Open

Other Resources

CAsyncMonikerFile Members

CAsyncMonikerFile::CreateBindStatusCallback

Creates a COM object that implements **IBindStatusCallback**.

```
virtual IUnknown* CreateBindStatusCallback(
   IUnknown* pUnkControlling
);
```

Parameters

pUnkControlling

A pointer to the controlling unknown (the outer **IUnknown**) or **NULL** if aggregation is not being used.

Return Value

If *pUnkControlling* is not **NULL**, the function returns a pointer to the inner **IUnknown** on a new COM object supporting **IBindStatusCallback**. If *pUnkControlling* is **NULL**, the function returns a pointer to an **IUnknown** on a new COM object supporting **IBindStatusCallback**.

Remarks

CAsyncMonikerFile requires a COM object that implements **IBindStatusCallback**. MFC implements such an object, and it is aggregatable. You can override **CreateBindStatusCallback** to return your own COM object. Your COM object can aggregate MFC's implementation by calling **CreateBindStatusCallback** with the controlling unknown of your COM object. COM objects implemented using the **CCmdTarget** COM support can retrieve the controlling unknown using **CCmdTarget::GetControllingUnknown**.

Alternately, your COM object can delegate to MFC's implementation by calling **CreateBindStatusCallback(NULL)**.

CAsyncMonikerFile::Open calls **CreateBindStatusCallback**.

For more information about asynchronous monikers and asynchronous binding, see the IBindStatusCallback interface and How Asynchronous Binding and Storage Work. For a discussion of aggregation, see Aggregation. All three topics are in the *Platform SDK*.

See Also

Reference

CAsyncMonikerFile Class

Hierarchy Chart

Other Resources

CAsyncMonikerFile Members

CAsyncMonikerFile::GetBindInfo

Called from the client of an asynchronous moniker to tell the asynchronous moniker how it wants to bind.

```
virtual DWORD GetBindInfo( ) const;
```

Return Value

Retrieves the settings for **IBindStatusCallBack**. For a description of the **IBindStatusCallback** interface, see the *Platform SDK*.

Remarks

The default implementation sets the binding to be asynchronous, to use a storage medium (a stream), and to use the data-push model. Override this function if you want to change the behavior of the binding.

One reason for doing this would be to bind using the data-pull model instead of the data-push model. In a data-pull model, the client drives the bind operation, and the moniker only provides data to the client when it is read. In a data-push model, the moniker drives the asynchronous bind operation and continuously notifies the client whenever new data is available.

See Also

Reference

CAsyncMonikerFile Class

Hierarchy Chart

Other Resources

CAsyncMonikerFile Members

CAsyncMonikerFile::GetBinding

Call this function to retrieve a pointer to the asynchronous transfer binding.

```
IBinding* GetBinding( ) const;
```

Return Value

A pointer to the **IBinding** interface provided when asynchronous transfer begins. Returns **NULL** if for any reason the transfer cannot be made asynchronously.

Remarks

This allows you to control the data transfer process through the **IBinding** interface, for example, with **IBinding::Abort**, **IBinding::Pause**, and **IBinding::Resume**.

For a description of the **IBinding** interface, see the *Platform SDK*.

See Also

Reference

CAsyncMonikerFile Class

Hierarchy Chart

Other Resources

CAsyncMonikerFile Members

CAsyncMonikerFile::GetFormatEtc

Call this function to retrieve the format of the data in the stream.

```
FORMATETC* GetFormatEtc( ) const;
```

Return Value

A pointer to the Windows structure FORMATETC for the currently opened stream. Returns **NULL** if the moniker has not been bound, if it is not asynchronous, or if the asynchronous operation has not begun.

See Also

Reference

CAsyncMonikerFile Class

Hierarchy Chart

Other Resources

CAsyncMonikerFile Members

CAsyncMonikerFile::GetPriority

Called from the client of an asynchronous moniker as the binding process starts to receive the priority given to the thread for the binding operation.

```
virtual LONG GetPriority( ) const;
```

Return Value

The priority at which the asynchronous transfer will take place. One of the standard thread priority flags: **THREAD_PRIORITY_ABOVE_NORMAL**, **THREAD_PRIORITY_BELOW_NORMAL**, **THREAD_PRIORITY_HIGHEST**, **THREAD_PRIORITY_IDLE**, **THREAD_PRIORITY_LOWEST**, **THREAD_PRIORITY_NORMAL**, and **THREAD_PRIORITY_TIME_CRITICAL**. See the Windows function SetThreadPriority for a description of these values.

Remarks

GetPriority should not be called directly. **THREAD_PRIORITY_NORMAL** is returned by the default implementation.

See Also

Reference

CAsyncMonikerFile Class

Hierarchy Chart

Other Resources

CAsyncMonikerFile Members

CAsyncMonikerFile::OnDataAvailable

An asynchronous moniker calls **OnDataAvailable** to provide data to the client as it becomes available, during asynchronous bind operations.

```
virtual void OnDataAvailable(
   DWORD dwSize,
   DWORD bscfFlag
);
```

Parameters

dwSize

The cumulative amount (in bytes) of data available since the beginning of the binding. Can be zero, indicating that the amount of data is not relevant to the operation, or that no specific amount became available.

bscfFlag

A **BSCF** enumeration value. Can be one or more of the following values:

- **BSCF_FIRSTDATANOTIFICATION** Identifies the first call to **OnDataAvailable** for a given bind operation.
- **BSCF_INTERMEDIATEDATANOTIFICATION** Identifies an intermediary call to **OnDataAvailable** for a bind operation.
- **BSCF_LASTDATANOTIFICATION** Identifies the last call to **OnDataAvailable** for a bind operation.

Remarks

The default implementation of this function does nothing. See the following example for a sample implementation.

Example

```
// refer to CDataPathProperty.
void CAsyncMyTextProperty::OnDataAvailable(DWORD dwSize, DWORD bscfFlag)
{
   if ((bscfFlag & BSCF_FIRSTDATANOTIFICATION) != 0)
   {
      m_dwReadBefore = 0;
      m_strText.Empty();
   }

   DWORD dwArriving = dwSize - m_dwReadBefore;

   if (dwArriving > 0)
   {
      int nLen = m_strText.GetLength();
      ASSERT(nLen == m_dwReadBefore);
      LPTSTR psz = m_strText.GetBuffer(nLen + dwArriving);
      Read(psz + nLen, dwArriving);
      m_strText.ReleaseBuffer(nLen + dwArriving);
      m_dwReadBefore = dwSize;
      GetControl()->Invalidate();
   }
}
```

See Also

Reference

CAsyncMonikerFile Class
Hierarchy Chart
CDataPathProperty Class

Other Resources

CAsyncMonikerFile Members

CAsyncMonikerFile::OnLowResource

Called by the moniker when resources are low.

```
virtual void OnLowResource( );
```

Remarks

The default implementation calls `GetBinding( )-> Abort( )`.

See Also

Reference

CAsyncMonikerFile Class

Hierarchy Chart

Other Resources

CAsyncMonikerFile Members

CAsyncMonikerFile::OnProgress

Called by the moniker repeatedly to indicate the current progress of this bind operation, typically at reasonable intervals during a lengthy operation.

```
virtual void OnProgress(
   ULONG ulProgress,
   ULONG ulProgressMax,
   ULONG ulStatusCode,
   LPCTSTR szStatusText
);
```

Parameters

ulProgress

Indicates the current progress of the bind operation relative to the expected maximum indicated in *ulProgressMax*.

ulProgressMax

Indicates the expected maximum value of *ulProgress* for the duration of calls to **OnProgress** for this operation.

ulStatusCode

Provides additional information regarding the progress of the bind operation. Valid values are taken from the **BINDSTATUS** enumeration. See Remarks for possible values.

szStatusText

Information about the current progress, depending on the value of *ulStatusCode*. See Remarks for possible values.

Remarks

Possible values for *ulStatusCode* (and the *szStatusText* for each value) are:

BINDSTATUS_FINDINGRESOURCE

The bind operation is finding the resource that holds the object or storage being bound to. The *szStatusText* provides the display name of the resource being searched for (for example, "www.microsoft.com").

BINDSTATUS_CONNECTING

The bind operation is connecting to the resource that holds the object or storage being bound to. The *szStatusText* provides the display name of the resource being connected to (for example, an IP address).

BINDSTATUS_SENDINGREQUEST

The bind operation is requesting the object or storage being bound to. The *szStatusText* provides the display name of the object (for example, a file name).

BINDSTATUS_REDIRECTING

The bind operation has been redirected to a different data location. The *szStatusText* provides the display name of the new data location.

BINDSTATUS_USINGCACHEDCOPY

The bind operation is retrieving the requested object or storage from a cached copy. The *szStatusText* is **NULL**.

BINDSTATUS_BEGINDOWNLOADDATA

The bind operation has begun receiving the object or storage being bound to. The *szStatusText* provides the display name of the data location.

BINDSTATUS_DOWNLOADINGDATA

The bind operation continues to receive the object or storage being bound to. The *szStatusText* provides the display name of the data location.

BINDSTATUS_ENDDOWNLOADDATA

The bind operation has finished receiving the object or storage being bound to. The *szStatusText* provides the display name of the data location.

BINDSTATUS_CLASSIDAVAILABLE

An instance of the object being bound to is just about to be created. The *szStatusText* provides the CLSID of the new object in string format, allowing the client an opportunity to cancel the bind operation, if desired.

See Also

Reference

CAsyncMonikerFile Class

Hierarchy Chart

Other Resources

CAsyncMonikerFile Members

CAsyncMonikerFile::OnStartBinding

Override this function in your derived classes to perform actions when binding is starting up.

```
virtual void OnStartBinding( );
```

Remarks

This function is called back by the moniker. The default implementation does nothing.

See Also

Reference

CAsyncMonikerFile Class
Hierarchy Chart
CAsyncMonikerFile::OnStopBinding

Other Resources

CAsyncMonikerFile Members

CAsyncMonikerFile::OnStopBinding

Called by the moniker at the end of the bind operation.

```
virtual void OnStopBinding(
   HRESULT hresult,
      LPCTSTR szError
);
```

Parameters

hresult

An **HRESULT** that is the error or warning value.

szErrort

A character string describing the error.

Remarks

Override this function to perform actions when the transfer is stopped. By default, the function releases **IBinding**.

For a description of the **IBinding** interface, see the *Platform SDK*.

See Also

Reference

CAsyncMonikerFile Class
Hierarchy Chart
CAsyncMonikerFile::OnStartBinding

Other Resources

CAsyncMonikerFile Members

CAsyncMonikerFile::Open

Call this member function to open a file asynchronously.

```
virtual BOOL Open(
   LPCTSTR lpszURL,
      CFileException* pError = NULL
);
virtual BOOL Open(
   IMoniker* pMoniker,
      CFileException* pError = NULL
);
virtual BOOL Open(
   LPCTSTR lpszURL,
      IBindHost* pBindHost,
      CFileException* pError = NULL
);
virtual BOOL Open(
   IMoniker* pMoniker,
      IBindHost* pBindHost,
      CFileException* pError = NULL
);
virtual BOOL Open(
   LPCTSTR lpszURL,
      IServiceProvider* pServiceProvider,
      CFileException* pError = NULL
);
virtual BOOL Open(
   IMoniker* pMoniker,
      IServiceProvider* pServiceProvider,
      CFileException* pError = NULL
);
virtual BOOL Open(
   LPCTSTR lpszURL,
      IUnknown* pUnknown,
      CFileException* pError = NULL
);
virtual BOOL Open(
   IMoniker* pMoniker,
      IUnknown* pUnknown,
      CFileException* pError = NULL
);
```

Parameters

lpszURL

A pointer to file to be opened asynchronously. The file can be any valid URL or filename.

pError

A pointer to the file exceptions. In the event of an error, it will be set to the cause.

pMoniker

A pointer to the asynchronous moniker interface **IMoniker**, a precise moniker that is the combination of the document's own moniker, which you can retrieve with **IOleClientSite::GetMoniker(** *OLEWHICHMK_CONTAINER* **)**, and a moniker created from the path name. The control can use this moniker to bind, but this is not the moniker the control should save.

pBindHost

A pointer to the **IBindHost** interface that will be used to create the moniker from a potentially relative pathname. If the bind host is invalid or does not provide a moniker, the call defaults to **Open(** *lpszFileName*,*pError* **)**. For a description of the **IBindHost** interface, see the *Platform SDK*.

pServiceProvider

A pointer to the **IServiceProvider** interface. If the service provider is invalid or fails to provide the service for **IBindHost**, the call defaults to **Open(** *lpszFileName*,*pError* **)**.

pUnknown

A pointer to the **IUnknown** interface. If **IServiceProvider** is found, the function queries for **IBindHost**. If the service provider is invalid or fails to provide the service for **IBindHost**, the call defaults to **Open(** *lpszFileName*,*pError* **)**.

Return Value

Nonzero if the file is opened successfully; otherwise 0.

Remarks

This call initiates the binding process.

You can use a URL or a filename for the *lpszURL* parameter. For example:

```
CMyAsyncMonFile  mamf;
mamf.Open(_T("http://www.microsoft.com"));
```

– or –

```
CMyAsyncMonFile  mamf;
mamf.Open(_T("file:c:\mydata.dat"));
```

See Also

Reference

CAsyncMonikerFile Class
Hierarchy Chart
CAsyncMonikerFile::CAsyncMonikerFile

Other Resources

CAsyncMonikerFile Members

CAsyncSocket Class

Represents a Windows Socket — an endpoint of network communication.

```
class CAsyncSocket : public CObject
```

Remarks

Class **CAsyncSocket** encapsulates the Windows Socket Functions API, providing an object-oriented abstraction for programmers who want to use Windows Sockets in conjunction with MFC.

This class is based on the assumption that you understand network communications. You are responsible for handling blocking, byte-order differences, and conversions between Unicode and multibyte character set (MBCS) strings. If you want a more convenient interface that manages these issues for you, see class CSocket.

To use a **CAsyncSocket** object, call its constructor, then call the Create function to create the underlying socket handle (type **SOCKET**), except on accepted sockets. For a server socket call the Listen member function, and for a client socket call the Connect member function. The server socket should call the Accept function upon receiving a connection request. Use the remaining **CAsyncSocket** functions to carry out communications between sockets. Upon completion, destroy the **CAsyncSocket** object if it was created on the heap; the destructor automatically calls the Close function. The **SOCKET** data type is described in the article Windows Sockets: Background.

For more information, see Windows Sockets: Using Class CAsyncSocket and related articles., as well as Windows Sockets 2 API.

Requirements

Header: afxsock.h

See Also

Reference

COBJECT Class
Hierarchy Chart
CSocket Class
CSocketFile Class

Other Resources

CAsyncSocket Members

CAsyncSocket Members

Base Class Members

CObject Members

Construction

CAsyncSocket	Constructs a **CAsyncSocket** object.
Create	Creates a socket.

Attributes

Attach	Attaches a socket handle to a **CAsyncSocket** object.
Detach	Detaches a socket handle from a **CAsyncSocket** object.
FromHandle	Returns a pointer to a **CAsyncSocket** object, given a socket handle.
GetLastError	Gets the error status for the last operation that failed.
GetPeerName	Gets the address of the peer socket to which the socket is connected.
GetPeerNameEx	Gets the address of the peer socket to which the socket is connected (handles IPv6 addresses).
GetSockName	Gets the local name for a socket.
GetSockNameEx	Gets the local name for a socket (handles IPv6 addresses).
GetSockOpt	Retrieves a socket option.
SetSockOpt	Sets a socket option.

Operations

Accept	Accepts a connection on the socket.
AsyncSelect	Requests event notification for the socket.
Bind	Associates a local address with the socket.
Close	Closes the socket.
Connect	Establishes a connection to a peer socket.
IOCtl	Controls the mode of the socket.
Listen	Establishes a socket to listen for incoming connection requests.
Receive	Receives data from the socket.
ReceiveFrom	Receives a datagram and stores the source address.
ReceiveFromEx	Receives a datagram and stores the source address (handles IPv6 addresses).

Send	Sends data to a connected socket.
SendTo	Sends data to a specific destination.
SendToEx	Sends data to a specific destination (handles IPv6 addresses).
ShutDown	Disables **Send** and/or **Receive** calls on the socket.
Socket	Allocates a socket handle.

Overridable Notification Functions

OnAccept	Notifies a listening socket that it can accept pending connection requests by calling **Accept**.
OnClose	Notifies a socket that the socket connected to it has closed.
OnConnect	Notifies a connecting socket that the connection attempt is complete, whether successfully or in error.
OnOutOfBandData	Notifies a receiving socket that there is out-of-band data to be read on the socket, usually an urgent message.
OnReceive	Notifies a listening socket that there is data to be retrieved by calling **Receive**.
OnSend	Notifies a socket that it can send data by calling **Send**.

Operators

operator =	Assigns a new value to a **CAsyncSocket** object.
operator SOCKET	Use this operator to retrieve the **SOCKET** handle of the **CAsyncSocket** object.

Data Members

m_hSocket	Indicates the **SOCKET** handle attached to this **CAsyncSocket** object.

See Also

Reference

CAsyncSocket Class

Hierarchy Chart

CAsyncSocket Member Functions

For information about the member functions in **CAsyncSocket**, see CAsyncSocket Class Members.

CAsyncSocket::Accept

Call this member function to accept a connection on a socket.

```
virtual BOOL Accept(
   CAsyncSocket& rConnectedSocket,
   SOCKADDR* lpSockAddr = NULL,
   int* lpSockAddrLen = NULL
);
```

Parameters

rConnectedSocket

A reference identifying a new socket that is available for connection.

lpSockAddr

A pointer to a SOCKADDR structure that receives the address of the connecting socket, as known on the network. The exact format of the *lpSockAddr* argument is determined by the address family established when the socket was created. If *lpSockAddr* and/or *lpSockAddrLen* are equal to **NULL**, then no information about the remote address of the accepted socket is returned.

lpSockAddrLen

A pointer to the length of the address in *lpSockAddr* in bytes. The *lpSockAddrLen* is a value-result parameter: it should initially contain the amount of space pointed to by *lpSockAddr*; on return it will contain the actual length (in bytes) of the address returned.

Return Value

Nonzero if the function is successful; otherwise 0, and a specific error code can be retrieved by calling GetLastError. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAEFAULT** The *lpSockAddrLen* argument is too small (less than the size of a SOCKADDR structure).
- **WSAEINPROGRESS** A blocking Windows Sockets call is in progress.
- **WSAEINVAL** **Listen** was not invoked prior to accept.
- **WSAEMFILE** The queue is empty upon entry to accept and there are no descriptors available.
- **WSAENOBUFS** No buffer space is available.
- **WSAENOTSOCK** The descriptor is not a socket.
- **WSAEOPNOTSUPP** The referenced socket is not a type that supports connection-oriented service.
- **WSAEWOULDBLOCK** The socket is marked as nonblocking and no connections are present to be accepted.

Remarks

This routine extracts the first connection in the queue of pending connections, creates a new socket with the same properties as this socket, and attaches it to *rConnectedSocket*. If no pending connections are present on the queue, **Accept** returns zero and **GetLastError** returns an error. The accepted socket (*rConnectedSocket)* cannot be used to accept more connections. The original socket remains open and listening.

The argument *lpSockAddr* is a result parameter that is filled in with the address of the connecting socket, as known to the communications layer. **Accept** is used with connection-based socket types such as **SOCK_STREAM**.

See Also

Reference

CAsyncSocket Class

Hierarchy Chart

CAsyncSocket::Bind
CAsyncSocket::Connect
CAsyncSocket::Listen
CAsyncSocket::Create
WSAAsyncSelect

Other Resources

CAsyncSocket Members

CAsyncSocket::AsyncSelect

Call this member function to request event notification for a socket.

```
BOOL AsyncSelect(
   long lEvent = FD_READ | FD_WRITE | FD_OOB | FD_ACCEPT | FD_CONNECT | FD_CLOSE
);
```

Parameters

lEvent

A bitmask which specifies a combination of network events in which the application is interested.

- **FD_READ** Want to receive notification of readiness for reading.
- **FD_WRITE** Want to receive notification when data is available to be read.
- **FD_OOB** Want to receive notification of the arrival of out-of-band data.
- **FD_ACCEPT** Want to receive notification of incoming connections.
- **FD_CONNECT** Want to receive notification of connection results.
- **FD_CLOSE** Want to receive notification when a socket has been closed by a peer.

Return Value

Nonzero if the function is successful; otherwise 0, and a specific error code can be retrieved by calling GetLastError. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAEINVAL** Indicates that one of the specified parameters was invalid.
- **WSAEINPROGRESS** A blocking Windows Sockets operation is in progress.

Remarks

This function is used to specify which MFC callback notification functions will be called for the socket. **AsyncSelect** automatically sets this socket to nonblocking mode. For more information, see the article Windows Sockets: Socket Notifications.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::GetLastError
WSAAsyncSelect

Other Resources

CAsyncSocket Members

CAsyncSocket::Attach

Call this member function to attach the *hSocket* handle to an **CAsyncSocket** object.

```
BOOL Attach(
   SOCKET hSocket,
      long lEvent = FD_READ | FD_WRITE | FD_OOB | FD_ACCEPT | FD_CONNECT | FD_CLOSE
);
```

Parameters

hSocket

Contains a handle to a socket.

lEvent

A bitmask which specifies a combination of network events in which the application is interested.

- **FD_READ** Want to receive notification of readiness for reading.
- **FD_WRITE** Want to receive notification when data is available to be read.
- **FD_OOB** Want to receive notification of the arrival of out-of-band data.
- **FD_ACCEPT** Want to receive notification of incoming connections.
- **FD_CONNECT** Want to receive notification of connection results.
- **FD_CLOSE** Want to receive notification when a socket has been closed by a peer.

Return Value

Nonzero if the function is successful.

Remarks

The **SOCKET** handle is stored in the object's m_hSocket data member.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::Detach

Other Resources

CAsyncSocket Members

CAsyncSocket::Bind

Call this member function to associate a local address with the socket.

```
BOOL Bind(
   UINT nSocketPort,
   LPCTSTR lpszSocketAddress = NULL
);
BOOL Bind (
   const SOCKADDR* lpSockAddr,
   int nSockAddrLen
);
```

Parameters

nSocketPort

The port identifying the socket application.

lpszSocketAddress

The network address, a dotted number such as "128.56.22.8". Passing the **NULL** string for this parameter indicates the **CAsyncSocket** instance should listen for client activity on all network interfaces.

lpSockAddr

A pointer to a SOCKADDR structure that contains the address to assign to this socket.

nSockAddrLen

The length of the address in *lpSockAddr* in bytes.

Return Value

Nonzero if the function is successful; otherwise 0, and a specific error code can be retrieved by calling GetLastError. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAEADDRINUSE** The specified address is already in use. (See the **SO_REUSEADDR** socket option under SetSockOpt.)
- **WSAEFAULT** The *nSockAddrLen* argument is too small (less than the size of a SOCKADDR structure).
- **WSAEINPROGRESS** A blocking Windows Sockets call is in progress.
- **WSAEAFNOSUPPORT** The specified address family is not supported by this port.
- **WSAEINVAL** The socket is already bound to an address.
- **WSAENOBUFS** Not enough buffers available, too many connections.
- **WSAENOTSOCK** The descriptor is not a socket.

Remarks

This routine is used on an unconnected datagram or stream socket, before subsequent **Connect** or **Listen** calls. Before it can accept connection requests, a listening server socket must select a port number and make it known to Windows Sockets by calling **Bind**. **Bind** establishes the local association (host address/port number) of the socket by assigning a local name to an unnamed socket.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::Connect

CAsyncSocket::Listen
CAsyncSocket::GetSockName
CAsyncSocket::SetSockOpt
CAsyncSocket::Create

Other Resources

CAsyncSocket Members

CAsyncSocket::CAsyncSocket

Constructs a blank socket object.

```
CAsyncSocket( );
```

Remarks

After constructing the object, you must call its **Create** member function to create the **SOCKET** data structure and bind its address. (On the server side of a Windows Sockets communication, when the listening socket creates a socket to use in the **Accept** call, you do not call **Create** for that socket.)

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::Create

Other Resources

CAsyncSocket Members

CAsyncSocket::Close

Closes the socket.

```
virtual void Close( );
```

Remarks

This function releases the socket descriptor so that further references to it will fail with the error **WSAENOTSOCK**. If this is the last reference to the underlying socket, the associated naming information and queued data are discarded. The socket object's destructor calls **Close** for you.

For **CAsyncSocket**, but not for **CSocket**, the semantics of **Close** are affected by the socket options **SO_LINGER** and **SO_DONTLINGER**. For further information, see member function **GetSockOpt**.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::Accept
CAsyncSocket::CAsyncSocket
CAsyncSocket::IOCtl
CAsyncSocket::GetSockOpt
CAsyncSocket::SetSockOpt
CAsyncSocket::AsyncSelect

Other Resources

CAsyncSocket Members

CAsyncSocket::Connect

Call this member function to establish a connection to an unconnected stream or datagram socket.

```
BOOL Connect(
   LPCTSTR lpszHostAddress,
   UINT nHostPort
);
BOOL Connect(
   const SOCKADDR* lpSockAddr,
   int nSockAddrLen
);
```

Parameters

lpszHostAddress

The network address of the socket to which this object is connected: a machine name such as "ftp.microsoft.com", or a dotted number such as "128.56.22.8".

nHostPort

The port identifying the socket application.

lpSockAddr

A pointer to a SOCKADDR structure that contains the address of the connected socket.

nSockAddrLen

The length of the address in *lpSockAddr* in bytes.

Return Value

Nonzero if the function is successful; otherwise 0, and a specific error code can be retrieved by calling GetLastError. If this indicates an error code of **WSAEWOULDBLOCK**, and your application is using the overridable callbacks, your application will receive an **OnConnect** message when the connect operation is complete. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAEADDRINUSE** The specified address is already in use.
- **WSAEINPROGRESS** A blocking Windows Sockets call is in progress.
- **WSAEADDRNOTAVAIL** The specified address is not available from the local machine.
- **WSAEAFNOSUPPORT** Addresses in the specified family cannot be used with this socket.
- **WSAECONNREFUSED** The attempt to connect was rejected.
- **WSAEDESTADDRREQ** A destination address is required.
- **WSAEFAULT** The *nSockAddrLen* argument is incorrect.
- **WSAEINVAL** Invalid host address.
- **WSAEISCONN** The socket is already connected.
- **WSAEMFILE** No more file descriptors are available.
- **WSAENETUNREACH** The network cannot be reached from this host at this time.
- **WSAENOBUFS** No buffer space is available. The socket cannot be connected.
- **WSAENOTSOCK** The descriptor is not a socket.
- **WSAETIMEDOUT** Attempt to connect timed out without establishing a connection.

- **WSAEWOULDBLOCK** The socket is marked as nonblocking and the connection cannot be completed immediately.

Remarks

If the socket is unbound, unique values are assigned to the local association by the system, and the socket is marked as bound. Note that if the address field of the name structure is all zeroes, **Connect** will return zero. To get extended error information, call the **GetLastError** member function.

For stream sockets (type **SOCK_STREAM**), an active connection is initiated to the foreign host. When the socket call completes successfully, the socket is ready to send/receive data.

For a datagram socket (type **SOCK_DGRAM**), a default destination is set, which will be used on subsequent **Send** and **Receive** calls.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::Accept
CAsyncSocket::Bind
CAsyncSocket::GetSockName
CAsyncSocket::Create
CAsyncSocket::AsyncSelect

Other Resources

CAsyncSocket Members

CAsyncSocket::Create

Call the **Create** member function after constructing a socket object to create the Windows socket and attach it.

```
BOOL Create(
   UINT nSocketPort = 0,
   int nSocketType = SOCK_STREAM,
   long lEvent = FD_READ | FD_WRITE | FD_OOB | FD_ACCEPT | FD_CONNECT | FD_CLOSE,
   LPCTSTR lpszSocketAddress = NULL
);
```

Parameters

nSocketPort

A well-known port to be used with the socket, or 0 if you want Windows Sockets to select a port.

nSocketType

SOCK_STREAM or **SOCK_DGRAM**.

lEvent

A bitmask which specifies a combination of network events in which the application is interested.

- **FD_READ** Want to receive notification of readiness for reading.
- **FD_WRITE** Want to receive notification of readiness for writing.
- **FD_OOB** Want to receive notification of the arrival of out-of-band data.
- **FD_ACCEPT** Want to receive notification of incoming connections.
- **FD_CONNECT** Want to receive notification of completed connection.
- **FD_CLOSE** Want to receive notification of socket closure.

lpszSockAddress

A pointer to a string containing the network address of the connected socket, a dotted number such as "128.56.22.8".Passing the **NULL** string for this parameter indicates the **CAsyncSocket** instance should listen for client activity on all network interfaces.

Return Value

Nonzero if the function is successful; otherwise 0, and a specific error code can be retrieved by calling GetLastError. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAEAFNOSUPPORT** The specified address family is not supported.
- **WSAEINPROGRESS** A blocking Windows Sockets operation is in progress.
- **WSAEMFILE** No more file descriptors are available.
- **WSAENOBUFS** No buffer space is available. The socket cannot be created.
- **WSAEPROTONOSUPPORT** The specified port is not supported.
- **WSAEPROTOTYPE** The specified port is the wrong type for this socket.
- **WSAESOCKTNOSUPPORT** The specified socket type is not supported in this address family.

Remarks

Create calls Socket and if successful, it calls Bind to bind the socket to the specified address. The following socket types are supported:

- **SOCK_STREAM** Provides sequenced, reliable, full-duplex, connection-based byte streams. Uses the Transmission Control Protocol (TCP) for the Internet address family.
- **SOCK_DGRAM** Supports datagrams, which are connectionless, unreliable packets of a fixed (typically small) maximum length. Uses the User Datagram Protocol (UDP) for the Internet address family.

Note
The **Accept** member function takes a reference to a new, empty **CSocket** object as its parameter. You must construct this object before you call **Accept**. Keep in mind that if this socket object goes out of scope, the connection closes. Do not call **Create** for this new socket object.

For more information about stream and datagram sockets, see the articles Windows Sockets: Background and Windows Sockets: Ports and Socket Addresses and Windows Sockets 2 API.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::Accept
CAsyncSocket::Bind
CAsyncSocket::Connect
CAsyncSocket::GetSockName
CAsyncSocket::IOCtl
CAsyncSocket::Listen
CAsyncSocket::Receive
CAsyncSocket::Send
CAsyncSocket::ShutDown

Other Resources

CAsyncSocket Members

CAsyncSocket::Detach

Call this member function to detach the **SOCKET** handle in the **m_hSocket** data member from the **CAsyncSocket** object and set **m_hSocket** to **NULL**.

```
SOCKET Detach( );
```

See Also

Reference

CAsyncSocket Class

Hierarchy Chart

CAsyncSocket::Attach

Other Resources

CAsyncSocket Members

CAsyncSocket::FromHandle

Returns a pointer to a **CAsyncSocket** object.

```
static CAsyncSocket* PASCAL FromHandle(
   SOCKET hSocket
);
```

Parameters

hSocket

Contains a handle to a socket.

Return Value

A pointer to an **CAsyncSocket** object, or **NULL** if there is no **CAsyncSocket** object attached to *hSocket*.

Remarks

When given a **SOCKET** handle, if a **CAsyncSocket** object is not attached to the handle, the member function returns **NULL**.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CSocket::FromHandle
CAsyncSocket::Attach
CAsyncSocket::Detach

Other Resources

CAsyncSocket Members

CAsyncSocket::GetLastError

Call this member function to get the error status for the last operation that failed.

```
static int PASCAL GetLastError( );
```

Return Value

The return value indicates the error code for the last Windows Sockets API routine performed by this thread.

Remarks

When a particular member function indicates that an error has occurred, **GetLastError** should be called to retrieve the appropriate error code. See the individual member function descriptions for a list of applicable error codes.

For more information about the error codes, see Windows Sockets 2 API.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
WSASetLastError

Other Resources

CAsyncSocket Members

CAsyncSocket::GetPeerName

Call this member function to get the address of the peer socket to which this socket is connected.

```
BOOL GetPeerName(
   CString& rPeerAddress,
   UINT& rPeerPort
);
BOOL GetPeerName(
   SOCKADDR* lpSockAddr,
   int* lpSockAddrLen
);
```

Parameters

rPeerAddress

Reference to a **CString** object that receives a dotted number IP address.

rPeerPort

Reference to a **UINT** that stores a port.

lpSockAddr

A pointer to the SOCKADDR structure that receives the name of the peer socket.

lpSockAddrLen

A pointer to the length of the address in *lpSockAddr* in bytes. On return, the *lpSockAddrLen* argument contains the actual size of *lpSockAddr* returned in bytes.

Return Value

Nonzero if the function is successful; otherwise 0, and a specific error code can be retrieved by calling GetLastError. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAEFAULT** The *lpSockAddrLen* argument is not large enough.
- **WSAEINPROGRESS** A blocking Windows Sockets call is in progress.
- **WSAENOTCONN** The socket is not connected.
- **WSAENOTSOCK** The descriptor is not a socket.

Remarks

To handle IPv6 addresses, use CAsyncSocket::GetPeerNameEx.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::Bind
CAsyncSocket::Connect
CAsyncSocket::Create
CAsyncSocket::GetSockName

Other Resources

CAsyncSocket Members

CAsyncSocket::GetPeerNameEx

Call this member function to get the address of the peer socket to which this socket is connected (handles IPv6 addresses).

```
BOOL GetPeerNameEx(
   CString& rPeerAddress,
   UINT& rPeerPort
);
```

Parameters

rPeerAddress

Reference to a **CString** object that receives a dotted number IP address.

rPeerPort

Reference to a **UINT** that stores a port.

Return Value

Nonzero if the function is successful; otherwise 0, and a specific error code can be retrieved by calling GetLastError. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAEFAULT** The *lpSockAddrLen* argument is not large enough.
- **WSAEINPROGRESS** A blocking Windows Sockets call is in progress.
- **WSAENOTCONN** The socket is not connected.
- **WSAENOTSOCK** The descriptor is not a socket.

Remarks

This function is the same as CAsyncSocket::GetPeerName except that it handles IPv6 addresses as well as older protocols.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::Bind
CAsyncSocket::Connect
CAsyncSocket::Create
CAsyncSocket::GetSockName

Other Resources

CAsyncSocket Members

CAsyncSocket::GetSockName

Call this member function to get the local name for a socket.

```
BOOL GetSockName(
   CString& rSocketAddress,
   UINT& rSocketPort
);
BOOL GetSockName(
   SOCKADDR* lpSockAddr,
   int* lpSockAddrLen
);
```

Parameters

rSocketAddress

Reference to a **CString** object that receives a dotted number IP address.

rSocketPort

Reference to a **UINT** that stores a port.

lpSockAddr

A pointer to a SOCKADDR structure that receives the address of the socket.

lpSockAddrLen

A pointer to the length of the address in *lpSockAddr* in bytes.

Return Value

Nonzero if the function is successful; otherwise 0, and a specific error code can be retrieved by calling GetLastError. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAEFAULT** The *lpSockAddrLen* argument is not large enough.
- **WSAEINPROGRESS** A blocking Windows Sockets operation is in progress.
- **WSAENOTSOCK** The descriptor is not a socket.
- **WSAEINVAL** The socket has not been bound to an address with **Bind**.

Remarks

This call is especially useful when a **Connect** call has been made without doing a **Bind** first; this call provides the only means by which you can determine the local association which has been set by the system.

To handle IPv6 addresses, use CAsyncSocket::GetSockNameEx

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::Bind
CAsyncSocket::Create
CAsyncSocket::GetPeerName

Other Resources

CAsyncSocket Members

CAsyncSocket::GetSockNameEx

Call this member function to get the local name for a socket (handles IPv6 addresses).

```
BOOL GetSockNameEx(
   CString& rSocketAddress,
   UINT& rSocketPort
);
```

Parameters

rSocketAddress

Reference to a **CString** object that receives a dotted number IP address.

rSocketPort

Reference to a **UINT** that stores a port.

Return Value

Nonzero if the function is successful; otherwise 0, and a specific error code can be retrieved by calling GetLastError. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAEFAULT** The *lpSockAddrLen* argument is not large enough.
- **WSAEINPROGRESS** A blocking Windows Sockets operation is in progress.
- **WSAENOTSOCK** The descriptor is not a socket.
- **WSAEINVAL** The socket has not been bound to an address with **Bind**.

Remarks

This call is the same as CAsyncSocket::GetSockName except that it handles IPv6 addresses as well as older protocols.

This call is especially useful when a **Connect** call has been made without doing a **Bind** first; this call provides the only means by which you can determine the local association which has been set by the system.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::Bind
CAsyncSocket::Create
CAsyncSocket::GetPeerName

Other Resources

CAsyncSocket Members

CAsyncSocket::GetSockOpt

Call this member function to retrieve a socket option.

```
BOOL GetSockOpt(
   int nOptionName,
   void* lpOptionValue,
   int* lpOptionLen,
   int nLevel = SOL_SOCKET
);
```

Parameters

nOptionName

The socket option for which the value is to be retrieved.

lpOptionValue

A pointer to the buffer in which the value for the requested option is to be returned. The value associated with the selected option is returned in the buffer *lpOptionValue*. The integer pointed to by *lpOptionLen* should originally contain the size of this buffer in bytes; and on return, it will be set to the size of the value returned. For **SO_LINGER**, this will be the size of a **LINGER** structure; for all other options it will be the size of a **BOOL** or an **int**, depending on the option. See the list of options and their sizes in the Remarks section.

lpOptionLen

A pointer to the size of the *lpOptionValue* buffer in bytes.

nLevel

The level at which the option is defined; the only supported levels are **SOL_SOCKET** and **IPPROTO_TCP**.

Return Value

Nonzero if the function is successful; otherwise 0, and a specific error code can be retrieved by calling GetLastError. If an option was never set with **SetSockOpt**, then **GetSockOpt** returns the default value for the option. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAEFAULT** The *lpOptionLen* argument was invalid.
- **WSAEINPROGRESS** A blocking Windows Sockets operation is in progress.
- **WSAENOPROTOOPT** The option is unknown or unsupported. In particular, **SO_BROADCAST** is not supported on sockets of type **SOCK_STREAM**, while **SO_ACCEPTCONN**, **SO_DONTLINGER**, **SO_KEEPALIVE**, **SO_LINGER**, and **SO_OOBINLINE** are not supported on sockets of type **SOCK_DGRAM**.
- **WSAENOTSOCK** The descriptor is not a socket.

Remarks

GetSockOpt retrieves the current value for a socket option associated with a socket of any type, in any state, and stores the result in *lpOptionValue*. Options affect socket operations, such as the routing of packets, out-of-band data transfer, and so on.

The following options are supported for **GetSockOpt**. The Type identifies the type of data addressed by *lpOptionValue*. The **TCP_NODELAY** option uses level **IPPROTO_TCP**; all other options use level **SOL_SOCKET**.

Value	Type	Meaning
SO_ACCEPTCONN	**BOOL**	Socket is listening.
SO_BROADCAST	**BOOL**	Socket is configured for the transmission of broadcast messages.

SO_DEBUG	**BOOL**	Debugging is enabled.
SO_DONTLINGER	**BOOL**	If true, the **SO_LINGER** option is disabled.
SO_DONTROUTE	**BOOL**	Routing is disabled.
SO_ERROR	**int**	Retrieve error status and clear.
SO_KEEPALIVE	**BOOL**	Keep-alives are being sent.
SO_LINGER	**struct LINGER**	Returns the current linger options.
SO_OOBINLINE	**BOOL**	Out-of-band data is being received in the normal data stream.
SO_RCVBUF	**int**	Buffer size for receives.
SO_REUSEADDR	**BOOL**	The socket can be bound to an address which is already in use.
SO_SNDBUF	**int**	Buffer size for sends.
SO_TYPE	**int**	The type of the socket (for example, **SOCK_STREAM**).
TCP_NODELAY	**BOOL**	Disables the Nagle algorithm for send coalescing.

Berkeley Software Distribution (BSD) options not supported for **GetSockOpt** are:

Value	Type	Meaning
SO_RCVLOWAT	**int**	Receive low water mark.
SO_RCVTIMEO	**int**	Receive timeout.
SO_SNDLOWAT	**int**	Send low water mark.
SO_SNDTIMEO	**int**	Send timeout.
IP_OPTIONS		Get options in IP header.
TCP_MAXSEG	**int**	Get TCP maximum segment size.

Calling **GetSockOpt** with an unsupported option will result in an error code of **WSAENOPROTOOPT** being returned from **GetLastError**.

See Also

Reference

CAsyncSocket Class

Hierarchy Chart

CAsyncSocket::SetSockOpt

Other Resources

CAsyncSocket Members

CAsyncSocket::IOCtl

Call this member function to control the mode of a socket.

```
BOOL IOCtl(
   long lCommand,
   DWORD* lpArgument
);
```

Parameters

lCommand

The command to perform on the socket.

lpArgument

A pointer to a parameter for *lCommand*.

Return Value

Nonzero if the function is successful; otherwise 0, and a specific error code can be retrieved by calling GetLastError. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAEINVAL** *lCommand* is not a valid command, or *lpArgument* is not an acceptable parameter for *lCommand*, or the command is not applicable to the type of socket supplied.
- **WSAEINPROGRESS** A blocking Windows Sockets operation is in progress.
- **WSAENOTSOCK** The descriptor is not a socket.

Remarks

This routine can be used on any socket in any state. It is used to get or retrieve operating parameters associated with the socket, independent of the protocol and communications subsystem. The following commands are supported:

- **FIONBIO** Enable or disable nonblocking mode on the socket. The *lpArgument* parameter points at a **DWORD**, which is nonzero if nonblocking mode is to be enabled and zero if it is to be disabled. If **AsyncSelect** has been issued on a socket, then any attempt to use **IOCtl** to set the socket back to blocking mode will fail with **WSAEINVAL**. To set the socket back to blocking mode and prevent the **WSAEINVAL** error, an application must first disable **AsyncSelect** by calling **AsyncSelect** with the *lEvent* parameter equal to 0, then call **IOCtl**.
- **FIONREAD** Determine the maximum number of bytes that can be read with one **Receive** call from this socket. The *lpArgument* parameter points at a **DWORD** in which **IOCtl** stores the result. If this socket is of type **SOCK_STREAM**, **FIONREAD** returns the total amount of data which can be read in a single **Receive**; this is normally the same as the total amount of data queued on the socket. If this socket is of type **SOCK_DGRAM**, **FIONREAD** returns the size of the first datagram queued on the socket.
- **SIOCATMARK** Determine whether all out-of-band data has been read. This applies only to a socket of type **SOCK_STREAM** which has been configured for in-line reception of any out-of-band data (**SO_OOBINLINE**). If no out-of-band data is waiting to be read, the operation returns nonzero. Otherwise it returns 0, and the next **Receive** or **ReceiveFrom** performed on the socket will retrieve some or all of the data preceding the "mark"; the application should use the **SIOCATMARK** operation to determine whether any data remains. If there is any normal data preceding the "urgent" (out-of-band) data, it will be received in order. (Note that a **Receive** or **ReceiveFrom** will never mix out-of-band and normal data in the same call.) The *lpArgument* parameter points at a **DWORD** in which **IOCtl** stores the result.

This function is a subset of **ioctl()** as used in Berkeley sockets. In particular, there is no command which is equivalent to **FIOASYNC**, while **SIOCATMARK** is the only socket-level command which is supported.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::AsyncSelect
CAsyncSocket::Create
CAsyncSocket::GetSockOpt
CAsyncSocket::SetSockOpt

Other Resources

CAsyncSocket Members

CAsyncSocket::Listen

Call this member function to listen for incoming connection requests.

```
BOOL Listen(
   int nConnectionBacklog = 5
);
```

Parameters

nConnectionBacklog

The maximum length to which the queue of pending connections can grow. Valid range is from 1 to 5.

Return Value

Nonzero if the function is successful; otherwise 0, and a specific error code can be retrieved by calling GetLastError. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAEADDRINUSE** An attempt has been made to listen on an address in use.
- **WSAEINPROGRESS** A blocking Windows Sockets operation is in progress.
- **WSAEINVAL** The socket has not been bound with **Bind** or is already connected.
- **WSAEISCONN** The socket is already connected.
- **WSAEMFILE** No more file descriptors are available.
- **WSAENOBUFS** No buffer space is available.
- **WSAENOTSOCK** The descriptor is not a socket.
- **WSAEOPNOTSUPP** The referenced socket is not of a type that supports the **Listen** operation.

Remarks

To accept connections, the socket is first created with **Create**, a backlog for incoming connections is specified with **Listen**, and then the connections are accepted with **Accept**. **Listen** applies only to sockets that support connections, that is, those of type **SOCK_STREAM**. This socket is put into "passive" mode where incoming connections are acknowledged and queued pending acceptance by the process.

This function is typically used by servers (or any application that wants to accept connections) that could have more than one connection request at a time: if a connection request arrives with the queue full, the client will receive an error with an indication of **WSAECONNREFUSED**.

Listen attempts to continue to function rationally when there are no available ports (descriptors). It will accept connections until the queue is emptied. If ports become available, a later call to **Listen** or **Accept** will refill the queue to the current or most recent "backlog," if possible, and resume listening for incoming connections.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::Accept
CAsyncSocket::Connect
CAsyncSocket::Create

Other Resources

CAsyncSocket Members

CAsyncSocket::OnAccept

Called by the framework to notify a listening socket that it can accept pending connection requests by calling the Accept member function.

```
virtual void OnAccept(
   int nErrorCode
);
```

Parameters

nErrorCode

The most recent error on a socket. The following error codes applies to the **OnAccept** member function:

- **0** The function executed successfully.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.

Remarks

For more information, see Windows Sockets: Socket Notifications.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::Accept
CAsyncSocket::GetLastError
CAsyncSocket::OnClose
CAsyncSocket::OnConnect
CAsyncSocket::OnOutOfBandData
CAsyncSocket::OnReceive
CAsyncSocket::OnSend

Other Resources

CAsyncSocket Members

CAsyncSocket::OnClose

Called by the framework to notify this socket that the connected socket is closed by its process.

```
virtual void OnClose(
   int nErrorCode
);
```

Parameters

nErrorCode

The most recent error on a socket. The following error codes apply to the **OnClose** member function:

- **0** The function executed successfully.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAECONNRESET** The connection was reset by the remote side.
- **WSAECONNABORTED** The connection was aborted due to timeout or other failure.

Remarks

For more information, see Windows Sockets: Socket Notifications.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::Close
CAsyncSocket::GetLastError
CAsyncSocket::OnAccept
CAsyncSocket::OnConnect
CAsyncSocket::OnOutOfBandData
CAsyncSocket::OnReceive
CAsyncSocket::OnSend

Other Resources

CAsyncSocket Members

CAsyncSocket::OnConnect

Called by the framework to notify this connecting socket that its connection attempt is completed, whether successfully or in error.

```
virtual void OnConnect(
   int nErrorCode
);
```

Parameters

nErrorCode

The most recent error on a socket. The following error codes apply to the **OnConnect** member function:

- **0** The function executed successfully.
- **WSAEADDRINUSE** The specified address is already in use.
- **WSAEADDRNOTAVAIL** The specified address is not available from the local machine.
- **WSAEAFNOSUPPORT** Addresses in the specified family cannot be used with this socket.
- **WSAECONNREFUSED** The attempt to connect was forcefully rejected.
- **WSAEDESTADDRREQ** A destination address is required.
- **WSAEFAULT** The *lpSockAddrLen* argument is incorrect.
- **WSAEINVAL** The socket is already bound to an address.
- **WSAEISCONN** The socket is already connected.
- **WSAEMFILE** No more file descriptors are available.
- **WSAENETUNREACH** The network cannot be reached from this host at this time.
- **WSAENOBUFS** No buffer space is available. The socket cannot be connected.
- **WSAENOTCONN** The socket is not connected.
- **WSAENOTSOCK** The descriptor is a file, not a socket.
- **WSAETIMEDOUT** The attempt to connect timed out without establishing a connection.

Remarks

Note

In CSocket, the **OnConnect** notification function is never called. For connections, you simply call **Connect**, which will return when the connection is completed (either successfully or in error). How connection notifications are handled is an MFC implementation detail.

For more information, see Windows Sockets: Socket Notifications.

Example

```
void CMyAsyncSocket::OnConnect(int nErrorCode)   // CMyAsyncSocket is
                                                 // derived from CAsyncSocket
{
   if (0 != nErrorCode)
   {
      switch( nErrorCode )
      {
         case WSAEADDRINUSE:
            AfxMessageBox("The specified address is already in use.\n");
            break;
         case WSAEADDRNOTAVAIL:
```

```
            AfxMessageBox("The specified address is not available from
               the local machine.\n");
            break;
         case WSAEAFNOSUPPORT:
            AfxMessageBox("Addresses in the specified family cannot be
               used with this socket.\n");
            break;
         case WSAECONNREFUSED:
            AfxMessageBox("The attempt to connect was forcefully
               rejected.\n");
            break;
         case WSAEDESTADDRREQ:
            AfxMessageBox("A destination address is required.\n");
            break;
         case WSAEFAULT:
            AfxMessageBox("The lpSockAddrLen argument is incorrect.\n");
            break;
         case WSAEINVAL:
            AfxMessageBox("The socket is already bound to an
               address.\n");
            break;
         case WSAEISCONN:
            AfxMessageBox("The socket is already connected.\n");
            break;
         case WSAEMFILE:
            AfxMessageBox("No more file descriptors are available.\n");
            break;
         case WSAENETUNREACH:
            AfxMessageBox("The network cannot be reached from this host
               at this time.\n");
            break;
         case WSAENOBUFS:
            AfxMessageBox("No buffer space is available. The socket
               cannot be connected.\n");
            break;
         case WSAENOTCONN:
            AfxMessageBox("The socket is not connected.\n");
            break;
         case WSAENOTSOCK:
            AfxMessageBox("The descriptor is a file, not a socket.\n");
            break;
         case WSAETIMEDOUT:
            AfxMessageBox("The attempt to connect timed out without
               establishing a connection. \n");
            break;
         default:
            TCHAR szError[256];
            wsprintf(szError, "OnConnect error: %d", nErrorCode);
            AfxMessageBox(szError);
            break;
      }
      AfxMessageBox("Please close the application");
   }
   CAsyncSocket::OnConnect(nErrorCode);
}
```

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::Connect
CAsyncSocket::GetLastError
CAsyncSocket::OnAccept
CAsyncSocket::OnClose
CAsyncSocket::OnOutOfBandData
CAsyncSocket::OnReceive
CAsyncSocket::OnSend

Other Resources

CAsyncSocket Members

CAsyncSocket::OnOutOfBandData

Called by the framework to notify the receiving socket that the sending socket has out-of-band data to send.

```
virtual void OnOutOfBandData(
   int nErrorCode
);
```

Parameters

nErrorCode

The most recent error on a socket. The following error codes apply to the **OnOutOfBandData** member function:

- **0** The function executed successfully.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.

Remarks

Out-of-band data is a logically independent channel that is associated with each pair of connected sockets of type **SOCK_STREAM**. The channel is generally used to send urgent data.

MFC supports out-of-band data, but users of class **CAsyncSocket** are discouraged from using it. The easier way is to create a second socket for passing such data. For more information about out-of-band data, see Windows Sockets: Socket Notifications.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::GetLastError
CAsyncSocket::OnAccept
CAsyncSocket::OnClose
CAsyncSocket::OnConnect
CAsyncSocket::OnReceive
CAsyncSocket::OnSend

Other Resources

CAsyncSocket Members

CAsyncSocket::OnReceive

Called by the framework to notify this socket that there is data in the buffer that can be retrieved by calling the **Receive** member function.

```
virtual void OnReceive(
   int nErrorCode
);
```

Parameters

nErrorCode

The most recent error on a socket. The following error codes apply to the **OnReceive** member function:

- **0** The function executed successfully.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.

Remarks

For more information, see Windows Sockets: Socket Notifications.

Example

```
void CMyAsyncSocket::OnReceive(int nErrorCode)   // CMyAsyncSocket is
                                                 // derived from CAsyncSocket
{
   static int i=0;

   i++;

   TCHAR buff[4096];
   int nRead;
   nRead = Receive(buff, 4096);

   switch (nRead)
   {
   case 0:
      Close();
      break;
   case SOCKET_ERROR:
      if (GetLastError() != WSAEWOULDBLOCK)
      {
         AfxMessageBox ("Error occurred");
         Close();
      }
      break;
   default:
      buff[nRead] = 0; //terminate the string
      CString szTemp(buff);
      m_strRecv += szTemp;   // m_strRecv is a CString declared
                             // in CMyAsyncSocket
      if (szTemp.CompareNoCase("bye") == 0 )    ShutDown();
   }
   CAsyncSocket::OnReceive(nErrorCode);
}
```

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::GetLastError
CAsyncSocket::OnAccept
CAsyncSocket::OnClose

CAsyncSocket::OnConnect
CAsyncSocket::OnOutOfBandData
CAsyncSocket::OnSend
CAsyncSocket::Receive

Other Resources

CAsyncSocket Members

CAsyncSocket::OnSend

Called by the framework to notify the socket that it can now send data by calling the **Send** member function.

```
virtual void OnSend(
   int nErrorCode
);
```

Parameters

nErrorCode

The most recent error on a socket. The following error codes apply to the **OnSend** member function:

- **0** The function executed successfully.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.

Remarks

For more information, see Windows Sockets: Socket Notifications.

Example

```
// CMyAsyncSocket is derived from CAsyncSocket and defines the
// following variables:
//    CString      m_sendBuffer;   //for async send
//    int      m_nBytesSent;
//    int      m_nBytesBufferSize;

void CMyAsyncSocket ::OnSend(int nErrorCode)
{
   while (m_nBytesSent < m_nBytesBufferSize)
   {
      int dwBytes;

      if ((dwBytes = Send((LPCTSTR)m_sendBuffer + m_nBytesSent,
         m_nBytesBufferSize - m_nBytesSent)) == SOCKET_ERROR)
      {
         if (GetLastError() == WSAEWOULDBLOCK) break;
         else
         {
            TCHAR szError[256];
            wsprintf(szError, "Server Socket failed to send: %d",
               GetLastError());
            Close();
            AfxMessageBox (szError);
         }
      }
      else
      {
         m_nBytesSent += dwBytes;
      }
   }
   if (m_nBytesSent == m_nBytesBufferSize)
      {
         m_nBytesSent = m_nBytesBufferSize = 0;
         m_sendBuffer = "";
      }
   CAsyncSocket::OnSend(nErrorCode);
}
```

See Also

Reference

CAsyncSocket Class

Hierarchy Chart
CAsyncSocket::GetLastError
CAsyncSocket::OnAccept
CAsyncSocket::OnClose
CAsyncSocket::OnConnect
CAsyncSocket::OnOutOfBandData
CAsyncSocket::OnReceive
CAsyncSocket::Send

Other Resources

CAsyncSocket Members

CAsyncSocket::Receive

Call this member function to receive data from a socket.

```
virtual int Receive(
   void* lpBuf,
   int nBufLen,
   int nFlags = 0
);
```

Parameters

lpBuf

A buffer for the incoming data.

nBufLen

The length of *lpBuf* in bytes.

nFlags

Specifies the way in which the call is made. The semantics of this function are determined by the socket options and the *nFlags* parameter. The latter is constructed by combining any of the following values with the C++ **OR** operator:

- **MSG_PEEK** Peek at the incoming data. The data is copied into the buffer but is not removed from the input queue.
- **MSG_OOB** Process out-of-band data.

Return Value

If no error occurs, **Receive** returns the number of bytes received. If the connection has been closed, it returns 0. Otherwise, a value of **SOCKET_ERROR** is returned, and a specific error code can be retrieved by calling GetLastError. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAENOTCONN** The socket is not connected.
- **WSAEINPROGRESS** A blocking Windows Sockets operation is in progress.
- **WSAENOTSOCK** The descriptor is not a socket.
- **WSAEOPNOTSUPP** **MSG_OOB** was specified, but the socket is not of type **SOCK_STREAM**.
- **WSAESHUTDOWN** The socket has been shut down; it is not possible to call **Receive** on a socket after **ShutDown** has been invoked with *nHow* set to 0 or 2.
- **WSAEWOULDBLOCK** The socket is marked as nonblocking and the **Receive** operation would block.
- **WSAEMSGSIZE** The datagram was too large to fit into the specified buffer and was truncated.
- **WSAEINVAL** The socket has not been bound with **Bind**.
- **WSAECONNABORTED** The virtual circuit was aborted due to timeout or other failure.
- **WSAECONNRESET** The virtual circuit was reset by the remote side.

Remarks

This function is used for connected stream or datagram sockets and is used to read incoming data.

For sockets of type **SOCK_STREAM**, as much information as is currently available up to the size of the buffer supplied is returned. If the socket has been configured for in-line reception of out-of-band data (socket option **SO_OOBINLINE**) and out-of-band data is unread, only out-of-band data will be returned. The application can use the **IOCtl SIOCATMARK** option or OnOutOfBandData to determine whether any more out-of-band data remains to be read.

For datagram sockets, data is extracted from the first enqueued datagram, up to the size of the buffer supplied. If the datagram is larger than the buffer supplied, the buffer is filled with the first part of the datagram, the excess data is lost, and **Receive** returns a value of **SOCKET_ERROR** with the error code set to **WSAEMSGSIZE**. If no incoming data is available at the socket, a value of **SOCKET_ERROR** is returned with the error code set to **WSAEWOULDBLOCK**. The OnReceive callback function can be used to determine when more data arrives.

If the socket is of type **SOCK_STREAM** and the remote side has shut down the connection gracefully, a **Receive** will complete immediately with 0 bytes received. If the connection has been reset, a **Receive** will fail with the error **WSAECONNRESET**.

Receive should be called only once for each time CAsyncSocket::OnReceive is called.

Example

See the example for CAsyncSocket::OnReceive.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::AsyncSelect
CAsyncSocket::Create
CAsyncSocket::ReceiveFrom
CAsyncSocket::Send

Other Resources

CAsyncSocket Members

CAsyncSocket::ReceiveFrom

Call this member function to receive a datagram and store the source address in the SOCKADDR structure or in *rSocketAddress*.

```
int ReceiveFrom(
   void* lpBuf,
   int nBufLen,
   CString& rSocketAddress,
   UINT& rSocketPort,
   int nFlags = 0
);
int ReceiveFrom(
   void* lpBuf,
   int nBufLen,
   SOCKADDR* lpSockAddr,
   int* lpSockAddrLen,
   int nFlags = 0
);
```

Parameters

lpBuf

A buffer for the incoming data.

nBufLen

The length of *lpBuf* in bytes.

rSocketAddress

Reference to a **CString** object that receives a dotted number IP address.

rSocketPort

Reference to a **UINT** that stores a port.

lpSockAddr

A pointer to a SOCKADDR structure that holds the source address upon return.

lpSockAddrLen

A pointer to the length of the source address in *lpSockAddr* in bytes.

nFlags

Specifies the way in which the call is made. The semantics of this function are determined by the socket options and the *nFlags* parameter. The latter is constructed by combining any of the following values with the C++ **OR** operator:

- **MSG_PEEK** Peek at the incoming data. The data is copied into the buffer but is not removed from the input queue.
- **MSG_OOB** Process out-of-band data.

Return Value

If no error occurs, **ReceiveFrom** returns the number of bytes received. If the connection has been closed, it returns 0. Otherwise, a value of **SOCKET_ERROR** is returned, and a specific error code can be retrieved by calling **GetLastError**. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAEFAULT** The *lpSockAddrLen* argument was invalid: the *lpSockAddr* buffer was too small to accommodate the peer address.
- **WSAEINPROGRESS** A blocking Windows Sockets operation is in progress.

- **WSAEINVAL** The socket has not been bound with **Bind**.
- **WSAENOTCONN** The socket is not connected (**SOCK_STREAM** only).
- **WSAENOTSOCK** The descriptor is not a socket.
- **WSAEOPNOTSUPP** **MSG_OOB** was specified, but the socket is not of type **SOCK_STREAM**.
- **WSAESHUTDOWN** The socket has been shut down; it is not possible to call **ReceiveFrom** on a socket after **ShutDown** has been invoked with *nHow* set to 0 or 2.
- **WSAEWOULDBLOCK** The socket is marked as nonblocking and the **ReceiveFrom** operation would block.
- **WSAEMSGSIZE** The datagram was too large to fit into the specified buffer and was truncated.
- **WSAECONNABORTED** The virtual circuit was aborted due to timeout or other failure.
- **WSAECONNRESET** The virtual circuit was reset by the remote side.

Remarks

This function is used to read incoming data on a (possibly connected) socket and capture the address from which the data was sent.

To handle IPv6 addresses, use CAsyncSocket::ReceiveFromEx.

For sockets of type **SOCK_STREAM**, as much information as is currently available up to the size of the buffer supplied is returned. If the socket has been configured for in-line reception of out-of-band data (socket option **SO_OOBINLINE**) and out-of-band data is unread, only out-of-band data will be returned. The application can use the **IOCtl SIOCATMARK** option or **OnOutOfBandData** to determine whether any more out-of-band data remains to be read. The *lpSockAddr* and *lpSockAddrLen* parameters are ignored for **SOCK_STREAM** sockets.

For datagram sockets, data is extracted from the first enqueued datagram, up to the size of the buffer supplied. If the datagram is larger than the buffer supplied, the buffer is filled with the first part of the message, the excess data is lost, and **ReceiveFrom** returns a value of **SOCKET_ERROR** with the error code set to **WSAEMSGSIZE**.

If *lpSockAddr* is nonzero, and the socket is of type **SOCK_DGRAM**, the network address of the socket which sent the data is copied to the corresponding SOCKADDR structure. The value pointed to by *lpSockAddrLen* is initialized to the size of this structure, and is modified on return to indicate the actual size of the address stored there. If no incoming data is available at the socket, the **ReceiveFrom** call waits for data to arrive unless the socket is nonblocking. In this case, a value of **SOCKET_ERROR** is returned with the error code set to **WSAEWOULDBLOCK**. The **OnReceive** callback can be used to determine when more data arrives.

If the socket is of type **SOCK_STREAM** and the remote side has shut down the connection gracefully, a **ReceiveFrom** will complete immediately with 0 bytes received.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::AsyncSelect
CAsyncSocket::Create
CAsyncSocket::Receive
CAsyncSocket::Send

Other Resources

CAsyncSocket Members

CAsyncSocket::ReceiveFromEx

Call this member function to receive a datagram and store the source address in the SOCKADDR structure or in *rSocketAddress* (handles IPv6 addresses).

```
int ReceiveFromEx(
   void* lpBuf,
   int nBufLen,
   CString& rSocketAddress,
   UINT& rSocketPort,
   int nFlags = 0
);
```

Parameters

lpBuf

A buffer for the incoming data.

nBufLen

The length of *lpBuf* in bytes.

rSocketAddress

Reference to a **CString** object that receives a dotted number IP address.

rSocketPort

Reference to a **UINT** that stores a port.

nFlags

Specifies the way in which the call is made. The semantics of this function are determined by the socket options and the *nFlags* parameter. The latter is constructed by combining any of the following values with the C++ **OR** operator:

- **MSG_PEEK** Peek at the incoming data. The data is copied into the buffer but is not removed from the input queue.
- **MSG_OOB** Process out-of-band data.

Return Value

If no error occurs, **ReceiveFromEx** returns the number of bytes received. If the connection has been closed, it returns 0. Otherwise, a value of **SOCKET_ERROR** is returned, and a specific error code can be retrieved by calling **GetLastError**. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAEFAULT** The *lpSockAddrLen* argument was invalid: the *lpSockAddr* buffer was too small to accommodate the peer address.
- **WSAEINPROGRESS** A blocking Windows Sockets operation is in progress.
- **WSAEINVAL** The socket has not been bound with **Bind**.
- **WSAENOTCONN** The socket is not connected (**SOCK_STREAM** only).
- **WSAENOTSOCK** The descriptor is not a socket.
- **WSAEOPNOTSUPP** **MSG_OOB** was specified, but the socket is not of type **SOCK_STREAM**.
- **WSAESHUTDOWN** The socket has been shut down; it is not possible to call **ReceiveFromEx** on a socket after **ShutDown** has been invoked with *nHow* set to 0 or 2.
- **WSAEWOULDBLOCK** The socket is marked as nonblocking and the **ReceiveFromEx** operation would block.
- **WSAEMSGSIZE** The datagram was too large to fit into the specified buffer and was truncated.

- **WSAECONNABORTED** The virtual circuit was aborted due to timeout or other failure.
- **WSAECONNRESET** The virtual circuit was reset by the remote side.

Remarks

This function is used to read incoming data on a (possibly connected) socket and capture the address from which the data was sent.

This function is the same as CAsyncSocket::ReceiveFrom except that it handles IPv6 addresses as well as older protocols.

For sockets of type **SOCK_STREAM**, as much information as is currently available up to the size of the buffer supplied is returned. If the socket has been configured for in-line reception of out-of-band data (socket option **SO_OOBINLINE**) and out-of-band data is unread, only out-of-band data will be returned. The application can use the **IOCtl SIOCATMARK** option or **OnOutOfBandData** to determine whether any more out-of-band data remains to be read. The *lpSockAddr* and *lpSockAddrLen* parameters are ignored for **SOCK_STREAM** sockets.

For datagram sockets, data is extracted from the first enqueued datagram, up to the size of the buffer supplied. If the datagram is larger than the buffer supplied, the buffer is filled with the first part of the message, the excess data is lost, and **ReceiveFromEx** returns a value of **SOCKET_ERROR** with the error code set to **WSAEMSGSIZE**.

If *lpSockAddr* is nonzero, and the socket is of type **SOCK_DGRAM**, the network address of the socket which sent the data is copied to the corresponding SOCKADDR structure. The value pointed to by *lpSockAddrLen* is initialized to the size of this structure, and is modified on return to indicate the actual size of the address stored there. If no incoming data is available at the socket, the **ReceiveFromEx** call waits for data to arrive unless the socket is nonblocking. In this case, a value of **SOCKET_ERROR** is returned with the error code set to **WSAEWOULDBLOCK**. The **OnReceive** callback can be used to determine when more data arrives.

If the socket is of type **SOCK_STREAM** and the remote side has shut down the connection gracefully, a **ReceiveFromEx** will complete immediately with 0 bytes received.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::AsyncSelect
CAsyncSocket::Create
CAsyncSocket::Receive
CAsyncSocket::Send

Other Resources

CAsyncSocket Members

CAsyncSocket::Send

Call this member function to send data on a connected socket.

```
virtual int Send(
   const void* lpBuf,
   int nBufLen,
   int nFlags = 0
);
```

Parameters

lpBuf

A buffer containing the data to be transmitted.

nBufLen

The length of the data in *lpBuf* in bytes.

nFlags

Specifies the way in which the call is made. The semantics of this function are determined by the socket options and the *nFlags* parameter. The latter is constructed by combining any of the following values with the C++ **OR** operator:

- **MSG_DONTROUTE** Specifies that the data should not be subject to routing. A Windows Sockets supplier can choose to ignore this flag.
- **MSG_OOB** Send out-of-band data (**SOCK_STREAM** only).

Return Value

If no error occurs, **Send** returns the total number of characters sent. (Note that this can be less than the number indicated by *nBufLen*.) Otherwise, a value of **SOCKET_ERROR** is returned, and a specific error code can be retrieved by calling GetLastError. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAEACCES** The requested address is a broadcast address, but the appropriate flag was not set.
- **WSAEINPROGRESS** A blocking Windows Sockets operation is in progress.
- **WSAEFAULT** The *lpBuf* argument is not in a valid part of the user address space.
- **WSAENETRESET** The connection must be reset because the Windows Sockets implementation dropped it.
- **WSAENOBUFS** The Windows Sockets implementation reports a buffer deadlock.
- **WSAENOTCONN** The socket is not connected.
- **WSAENOTSOCK** The descriptor is not a socket.
- **WSAEOPNOTSUPP** **MSG_OOB** was specified, but the socket is not of type **SOCK_STREAM**.
- **WSAESHUTDOWN** The socket has been shut down; it is not possible to call **Send** on a socket after **ShutDown** has been invoked with *nHow* set to 1 or 2.
- **WSAEWOULDBLOCK** The socket is marked as nonblocking and the requested operation would block.
- **WSAEMSGSIZE** The socket is of type **SOCK_DGRAM**, and the datagram is larger than the maximum supported by the Windows Sockets implementation.
- **WSAEINVAL** The socket has not been bound with **Bind**.
- **WSAECONNABORTED** The virtual circuit was aborted due to timeout or other failure.
- **WSAECONNRESET** The virtual circuit was reset by the remote side.

Remarks

Send is used to write outgoing data on connected stream or datagram sockets. For datagram sockets, care must be taken not to exceed the maximum IP packet size of the underlying subnets, which is given by the **iMaxUdpDg** element in the WSADATA structure returned by **AfxSocketInit**. If the data is too long to pass atomically through the underlying protocol, the error **WSAEMSGSIZE** is returned via **GetLastError**, and no data is transmitted.

Note that for a datagram socket the successful completion of a **Send** does not indicate that the data was successfully delivered.

On **CAsyncSocket** objects of type **SOCK_STREAM**, the number of bytes written can be between 1 and the requested length, depending on buffer availability on both the local and foreign hosts.

Example

See the example for CAsyncSocket::OnSend.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::Create
CAsyncSocket::Receive
CAsyncSocket::ReceiveFrom
CAsyncSocket::SendTo

Other Resources

CAsyncSocket Members

CAsyncSocket::SendTo

Call this member function to send data to a specific destination.

```
int SendTo(
   const void* lpBuf,
   int nBufLen,
   UINT nHostPort,
   LPCTSTR lpszHostAddress = NULL,
   int nFlags = 0
);
int SendTo(
   const void* lpBuf,
   int nBufLen,
   const SOCKADDR* lpSockAddr,
   int nSockAddrLen,
   int nFlags = 0
);
```

Parameters

lpBuf

A buffer containing the data to be transmitted.

nBufLen

The length of the data in *lpBuf* in bytes.

nHostPort

The port identifying the socket application.

lpszHostAddress

The network address of the socket to which this object is connected: a machine name such as "ftp.microsoft.com," or a dotted number such as "128.56.22.8".

nFlags

Specifies the way in which the call is made. The semantics of this function are determined by the socket options and the *nFlags* parameter. The latter is constructed by combining any of the following values with the C++ **OR** operator:

- **MSG_DONTROUTE** Specifies that the data should not be subject to routing. A Windows Sockets supplier can choose to ignore this flag.
- **MSG_OOB** Send out-of-band data (**SOCK_STREAM** only).

lpSockAddr

A pointer to a SOCKADDR structure that contains the address of the target socket.

nSockAddrLen

The length of the address in *lpSockAddr* in bytes.

Return Value

If no error occurs, **SendTo** returns the total number of characters sent. (Note that this can be less than the number indicated by *nBufLen*.) Otherwise, a value of **SOCKET_ERROR** is returned, and a specific error code can be retrieved by calling GetLastError. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAEACCES** The requested address is a broadcast address, but the appropriate flag was not set.
- **WSAEINPROGRESS** A blocking Windows Sockets operation is in progress.

- **WSAEFAULT** The *lpBuf* or *lpSockAddr* parameters are not part of the user address space, or the *lpSockAddr* argument is too small (less than the size of a SOCKADDR structure).
- **WSAEINVAL** The host name is invalid.
- **WSAENETRESET** The connection must be reset because the Windows Sockets implementation dropped it.
- **WSAENOBUFS** The Windows Sockets implementation reports a buffer deadlock.
- **WSAENOTCONN** The socket is not connected (**SOCK_STREAM** only).
- **WSAENOTSOCK** The descriptor is not a socket.
- **WSAEOPNOTSUPP** **MSG_OOB** was specified, but the socket is not of type **SOCK_STREAM**.
- **WSAESHUTDOWN** The socket has been shut down; it is not possible to call **SendTo** on a socket after **ShutDown** has been invoked with *nHow* set to 1 or 2.
- **WSAEWOULDBLOCK** The socket is marked as nonblocking and the requested operation would block.
- **WSAEMSGSIZE** The socket is of type **SOCK_DGRAM**, and the datagram is larger than the maximum supported by the Windows Sockets implementation.
- **WSAECONNABORTED** The virtual circuit was aborted due to timeout or other failure.
- **WSAECONNRESET** The virtual circuit was reset by the remote side.
- **WSAEADDRNOTAVAIL** The specified address is not available from the local machine.
- **WSAEAFNOSUPPORT** Addresses in the specified family cannot be used with this socket.
- **WSAEDESTADDRREQ** A destination address is required.
- **WSAENETUNREACH** The network cannot be reached from this host at this time.

Remarks

SendTo is used on datagram or stream sockets and is used to write outgoing data on a socket. For datagram sockets, care must be taken not to exceed the maximum IP packet size of the underlying subnets, which is given by the **iMaxUdpDg** element in the WSADATA structure filled out by AfxSocketInit. If the data is too long to pass atomically through the underlying protocol, the error **WSAEMSGSIZE** is returned, and no data is transmitted.

Note that the successful completion of a **SendTo** does not indicate that the data was successfully delivered.

SendTo is only used on a **SOCK_DGRAM** socket to send a datagram to a specific socket identified by the *lpSockAddr* parameter.

To send a broadcast (on a **SOCK_DGRAM** only), the address in the *lpSockAddr* parameter should be constructed using the special IP address **INADDR_BROADCAST** (defined in the Windows Sockets header file WINSOCK.H) together with the intended port number. Or, if the *lpszHostAddress* parameter is **NULL**, the socket is configured for broadcast. It is generally inadvisable for a broadcast datagram to exceed the size at which fragmentation can occur, which implies that the data portion of the datagram (excluding headers) should not exceed 512 bytes.

To handle IPv6 addresses, use CAsyncSocket::SendToEx.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::Create
CAsyncSocket::Receive
CAsyncSocket::ReceiveFrom
CAsyncSocket::Send

Other Resources

CAsyncSocket Members

CAsyncSocket::SendToEx

Call this member function to send data to a specific destination (handles IPv6 addresses).

```
int SendToEx(
   const void* lpBuf,
   int nBufLen,
   UINT nHostPort,
   LPCTSTR lpszHostAddress = NULL,
   int nFlags = 0
);
```

Parameters

lpBuf

A buffer containing the data to be transmitted.

nBufLen

The length of the data in *lpBuf* in bytes.

nHostPort

The port identifying the socket application.

lpszHostAddress

The network address of the socket to which this object is connected: a machine name such as "ftp.microsoft.com," or a dotted number such as "128.56.22.8".

nFlags

Specifies the way in which the call is made. The semantics of this function are determined by the socket options and the *nFlags* parameter. The latter is constructed by combining any of the following values with the C++ **OR** operator:

- **MSG_DONTROUTE** Specifies that the data should not be subject to routing. A Windows Sockets supplier can choose to ignore this flag.
- **MSG_OOB** Send out-of-band data (**SOCK_STREAM** only).

Return Value

If no error occurs, **SendToEx** returns the total number of characters sent. (Note that this can be less than the number indicated by *nBufLen*.) Otherwise, a value of **SOCKET_ERROR** is returned, and a specific error code can be retrieved by calling GetLastError. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAEACCES** The requested address is a broadcast address, but the appropriate flag was not set.
- **WSAEINPROGRESS** A blocking Windows Sockets operation is in progress.
- **WSAEFAULT** The *lpBuf* or *lpSockAddr* parameters are not part of the user address space, or the *lpSockAddr* argument is too small (less than the size of a SOCKADDR structure).
- **WSAEINVAL** The host name is invalid.
- **WSAENETRESET** The connection must be reset because the Windows Sockets implementation dropped it.
- **WSAENOBUFS** The Windows Sockets implementation reports a buffer deadlock.
- **WSAENOTCONN** The socket is not connected (**SOCK_STREAM** only).
- **WSAENOTSOCK** The descriptor is not a socket.
- **WSAEOPNOTSUPP** **MSG_OOB** was specified, but the socket is not of type **SOCK_STREAM**.

- **WSAESHUTDOWN** The socket has been shut down; it is not possible to call **SendToEx** on a socket after **ShutDown** has been invoked with *nHow* set to 1 or 2.
- **WSAEWOULDBLOCK** The socket is marked as nonblocking and the requested operation would block.
- **WSAEMSGSIZE** The socket is of type **SOCK_DGRAM**, and the datagram is larger than the maximum supported by the Windows Sockets implementation.
- **WSAECONNABORTED** The virtual circuit was aborted due to timeout or other failure.
- **WSAECONNRESET** The virtual circuit was reset by the remote side.
- **WSAEADDRNOTAVAIL** The specified address is not available from the local machine.
- **WSAEAFNOSUPPORT** Addresses in the specified family cannot be used with this socket.
- **WSAEDESTADDRREQ** A destination address is required.
- **WSAENETUNREACH** The network cannot be reached from this host at this time.

Remarks

This method is the same as CAsyncSocket::SendTo except that it handles IPv6 addresses as well as older protocols.

SendToEx is used on datagram or stream sockets and is used to write outgoing data on a socket. For datagram sockets, care must be taken not to exceed the maximum IP packet size of the underlying subnets, which is given by the **iMaxUdpDg** element in the WSADATA structure filled out by AfxSocketInit. If the data is too long to pass atomically through the underlying protocol, the error **WSAEMSGSIZE** is returned, and no data is transmitted.

Note that the successful completion of a **SendToEx** does not indicate that the data was successfully delivered.

SendToEx is only used on a **SOCK_DGRAM** socket to send a datagram to a specific socket identified by the *lpSockAddr* parameter.

To send a broadcast (on a **SOCK_DGRAM** only), the address in the *lpSockAddr* parameter should be constructed using the special IP address **INADDR_BROADCAST** (defined in the Windows Sockets header file WINSOCK.H) together with the intended port number. Or, if the *lpszHostAddress* parameter is **NULL**, the socket is configured for broadcast. It is generally inadvisable for a broadcast datagram to exceed the size at which fragmentation can occur, which implies that the data portion of the datagram (excluding headers) should not exceed 512 bytes.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::Create
CAsyncSocket::Receive
CAsyncSocket::ReceiveFrom
CAsyncSocket::Send

Other Resources

CAsyncSocket Members

CAsyncSocket::SetSockOpt

Call this member function to set a socket option.

```
BOOL SetSockOpt(
   int nOptionName,
   const void* lpOptionValue,
   int nOptionLen,
   int nLevel = SOL_SOCKET
);
```

Parameters

nOptionName

The socket option for which the value is to be set.

lpOptionValue

A pointer to the buffer in which the value for the requested option is supplied.

nOptionLen

The size of the *lpOptionValue* buffer in bytes.

nLevel

The level at which the option is defined; the only supported levels are **SOL_SOCKET** and **IPPROTO_TCP**.

Return Value

Nonzero if the function is successful; otherwise 0, and a specific error code can be retrieved by calling GetLastError. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAEFAULT** *lpOptionValue* is not in a valid part of the process address space.
- **WSAEINPROGRESS** A blocking Windows Sockets operation is in progress.
- **WSAEINVAL** *nLevel* is not valid, or the information in *lpOptionValue* is not valid.
- **WSAENETRESET** Connection has timed out when **SO_KEEPALIVE** is set.
- **WSAENOPROTOOPT** The option is unknown or unsupported. In particular, **SO_BROADCAST** is not supported on sockets of type **SOCK_STREAM**, while **SO_DONTLINGER**, **SO_KEEPALIVE**, **SO_LINGER**, and **SO_OOBINLINE** are not supported on sockets of type **SOCK_DGRAM**.
- **WSAENOTCONN** Connection has been reset when **SO_KEEPALIVE** is set.
- **WSAENOTSOCK** The descriptor is not a socket.

Remarks

SetSockOpt sets the current value for a socket option associated with a socket of any type, in any state. Although options can exist at multiple protocol levels, this specification only defines options that exist at the uppermost "socket" level. Options affect socket operations, such as whether expedited data is received in the normal data stream, whether broadcast messages can be sent on the socket, and so on.

There are two types of socket options: Boolean options that enable or disable a feature or behavior, and options which require an integer value or structure. To enable a Boolean option, *lpOptionValue* points to a nonzero integer. To disable the option *lpOptionValue* points to an integer equal to zero. *nOptionLen* should be equal to **sizeof(BOOL)** for Boolean options. For other options, *lpOptionValue* points to the integer or structure that contains the desired value for the option, and *nOptionLen* is the length of the integer or structure.

SO_LINGER controls the action taken when unsent data is queued on a socket and the **Close** function is called to close the

socket.

By default, a socket cannot be bound (see Bind) to a local address which is already in use. On occasion, however, it may be desirable to "reuse" an address in this way. Since every connection is uniquely identified by the combination of local and remote addresses, there is no problem with having two sockets bound to the same local address as long as the remote addresses are different.

To inform the Windows Sockets implementation that a **Bind** call on a socket should not be disallowed because the desired address is already in use by another socket, the application should set the **SO_REUSEADDR** socket option for the socket before issuing the **Bind** call. Note that the option is interpreted only at the time of the **Bind** call: it is therefore unnecessary (but harmless) to set the option on a socket which is not to be bound to an existing address, and setting or resetting the option after the **Bind** call has no effect on this or any other socket.

An application can request that the Windows Sockets implementation enable the use of "keep-alive" packets on Transmission Control Protocol (TCP) connections by turning on the **SO_KEEPALIVE** socket option. A Windows Sockets implementation need not support the use of keep-alives: if it does, the precise semantics are implementation-specific but should conform to section 4.2.3.6 of RFC 1122: "Requirements for Internet Hosts — Communication Layers." If a connection is dropped as the result of "keep-alives" the error code **WSAENETRESET** is returned to any calls in progress on the socket, and any subsequent calls will fail with **WSAENOTCONN**.

The **TCP_NODELAY** option disables the Nagle algorithm. The Nagle algorithm is used to reduce the number of small packets sent by a host by buffering unacknowledged send data until a full-size packet can be sent. However, for some applications this algorithm can impede performance, and **TCP_NODELAY** can be used to turn it off. Application writers should not set **TCP_NODELAY** unless the impact of doing so is well-understood and desired, since setting **TCP_NODELAY** can have a significant negative impact on network performance. **TCP_NODELAY** is the only supported socket option which uses level **IPPROTO_TCP**; all other options use level **SOL_SOCKET**.

Some implementations of Windows Sockets supply output debug information if the **SO_DEBUG** option is set by an application.

The following options are supported for **SetSockOpt**. The Type identifies the type of data addressed by *lpOptionValue*.

Value	Type	Meaning
SO_BROADCAST	**BOOL**	Allow transmission of broadcast messages on the socket.
SO_DEBUG	**BOOL**	Record debugging information.
SO_DONTLINGER	**BOOL**	Don't block **Close** waiting for unsent data to be sent. Setting this option is equivalent to setting **SO_LINGER** with **l_onoff** set to zero.
SO_DONTROUTE	**BOOL**	Don't route: send directly to interface.
SO_KEEPALIVE	**BOOL**	Send keep-alives.
SO_LINGER	**struct LINGER**	Linger on **Close** if unsent data is present.
SO_OOBINLINE	**BOOL**	Receive out-of-band data in the normal data stream.
SO_RCVBUF	**int**	Specify buffer size for receives.
SO_REUSEADDR	**BOOL**	Allow the socket to be bound to an address which is already in use. (See Bind.)
SO_SNDBUF	**int**	Specify buffer size for sends.

TCP_NODELAY	**BOOL**	Disables the Nagle algorithm for send coalescing.

Berkeley Software Distribution (BSD) options not supported for **SetSockOpt** are:

Value	Type	Meaning
SO_ACCEPTCONN	**BOOL**	Socket is listening
SO_ERROR	**int**	Get error status and clear.
SO_RCVLOWAT	**int**	Receive low water mark.
SO_RCVTIMEO	**int**	Receive timeout
SO_SNDLOWAT	**int**	Send low water mark.
SO_SNDTIMEO	**int**	Send timeout.
SO_TYPE	**int**	Type of the socket.
IP_OPTIONS		Set options field in IP header.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::AsyncSelect
CAsyncSocket::Bind
CAsyncSocket::Create
CAsyncSocket::GetSockOpt
CAsyncSocket::IOCtl

Other Resources

CAsyncSocket Members

CAsyncSocket::ShutDown

Call this member function to disable sends, receives, or both on the socket.

```
BOOL ShutDown(
   int nHow = sends
);
```

Parameters

nHow

A flag that describes what types of operation will no longer be allowed, using the following enumerated values:

- **receives = 0**
- **sends = 1**
- **both = 2**

Return Value

Nonzero if the function is successful; otherwise 0, and a specific error code can be retrieved by calling GetLastError. The following errors apply to this member function:

- **WSANOTINITIALISED** A successful AfxSocketInit must occur before using this API.
- **WSAENETDOWN** The Windows Sockets implementation detected that the network subsystem failed.
- **WSAEINVAL** *nHow* is not valid.
- **WSAEINPROGRESS** A blocking Windows Sockets operation is in progress.
- **WSAENOTCONN** The socket is not connected (**SOCK_STREAM** only).
- **WSAENOTSOCK** The descriptor is not a socket.

Remarks

ShutDown is used on all types of sockets to disable reception, transmission, or both. If *nHow* is 0, subsequent receives on the socket will be disallowed. This has no effect on the lower protocol layers.

For Transmission Control Protocol (TCP), the TCP window is not changed and incoming data will be accepted (but not acknowledged) until the window is exhausted. For User Datagram Protocol (UDP), incoming datagrams are accepted and queued. In no case will an ICMP error packet be generated. If *nHow* is 1, subsequent sends are disallowed. For TCP sockets, a FIN will be sent. Setting *nHow* to 2 disables both sends and receives as described above.

Note that **ShutDown** does not close the socket, and resources attached to the socket will not be freed until **Close** is called. An application should not rely on being able to reuse a socket after it has been shut down. In particular, a Windows Sockets implementation is not required to support the use of **Connect** on such a socket.

Example

See the example for CAsyncSocket::OnReceive.

See Also

Reference

CAsyncSocket Class
Hierarchy Chart
CAsyncSocket::Connect
CAsyncSocket::Create

Other Resources

CAsyncSocket Members

CAsyncSocket::Socket

Allocates a socket handle.

```
BOOL Socket(
   int nSocketType = SOCK_STREAM,
   long lEvent = FD_READ | FD_WRITE | FD_OOB | FD_ACCEPT | FD_CONNECT | FD_CLOSE,
   int nProtocolType = 0,
   int nAddressFormat = PF_INET
);
```

Parameters

nSocketType

Specifies **SOCK_STREAM** or **SOCK_DGRAM**.

lEvent

A bitmask that specifies a combination of network events in which the application is interested.

- **FD_READ**: Want to receive notification of readiness for reading.
- **FD_WRITE**: Want to receive notification of readiness for writing.
- **FD_OOB**: Want to receive notification of the arrival of out-of-band data.
- **FD_ACCEPT**: Want to receive notification of incoming connections.
- **FD_CONNECT**: Want to receive notification of completed connection.
- **FD_CLOSE**: Want to receive notification of socket closure.

nProtocolType

Protocol to be used with the socket that is specific to the indicated address family.

nAddressFormat

Address family specification.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This method allocates a socket handle. It does not call CAsyncSocket::Bind to bind the socket to a specified address, so you need to call **Bind** later to bind the socket to a specified address. You can use CAsyncSocket::SetSockOpt to set the socket option before it is bound.

See Also

Reference

CAsyncSocket Class

CAsyncSocket Data Members

For information about the data members in **CAsyncSocket**, see CAsyncSocket Class Members.

CAsyncSocket::m_hSocket

Contains the **SOCKET** handle for the socket encapsulated by this **CAsyncSocket** object.

```
SOCKET m_hSocket;
```

See Also

Reference

CAsyncSocket Class

Hierarchy Chart

Other Resources

CAsyncSocket Members

CAsyncSocket Operators

For information about the operators in **CAsyncSocket**, see CAsyncSocket Class Members.

CAsyncSocket::operator =

Assigns a new value to a **CAsyncSocket** object.

```
void operator=(
   const CAsyncSocket& rSrc
);
```

Parameters

rSrc

A reference to an existing **CAsyncSocket** object.

Remarks

Call this function to copy an existing **CAsyncSocket** object to another **CAsyncSocket** object.

See Also

Reference

CAsyncSocket Class

Hierarchy Chart

Other Resources

CAsyncSocket Members

CAsyncSocket::operator SOCKET

Use this operator to retrieve the **SOCKET** handle of the **CAsyncSocket** object.

```
operator SOCKET( ) const;
```

Return Value

If successful, the handle of the **SOCKET** object; otherwise, **NULL**.

Remarks

You can use the handle to call Windows APIs directly.

See Also

Reference

CAsyncSocket Class

Hierarchy Chart

Other Resources

CAsyncSocket Members

CBitmap Class

Encapsulates a Windows graphics device interface (GDI) bitmap and provides member functions to manipulate the bitmap.

```
class CBitmap : public CGdiObject
```

Remarks

To use a **CBitmap** object, construct the object, attach a bitmap handle to it with one of the initialization member functions, and then call the object's member functions.

For more information on using graphic objects like **CBitmap**, see Graphic Objects.

Requirements

Header: afxwin.h

See Also

Tasks

MDI Sample: Demonstrates MDI Without Using Doc/View Architecture

Reference

CGdiObject Class

Hierarchy Chart

Other Resources

CBitmap Members

CBitmap Members

Base Class Members

CObject Members

CGdiObject Members

Construction

CBitmap	Constructs a **CBitmap** object.

Initialization

CreateBitmap	Initializes the object with a device-dependent memory bitmap that has a specified width, height, and bit pattern.
CreateBitmapIndirect	Initializes the object with a bitmap with the width, height, and bit pattern (if one is specified) given in a **BITMAP** structure.
CreateCompatibleBitmap	Initializes the object with a bitmap so that it is compatible with a specified device.
CreateDiscardableBitmap	Initializes the object with a discardable bitmap that is compatible with a specified device.
LoadBitmap	Initializes the object by loading a named bitmap resource from the application's executable file and a ttaching the bitmap to the object.
LoadMappedBitmap	Loads a bitmap and maps colors to current system colors.
LoadOEMBitmap	Initializes the object by loading a predefined Windows bitmap and attaching the bitmap to the object .

Attributes

GetBitmap	Fills a **BITMAP** structure with information about the bitmap.
operator HBITMAP	Returns the Windows handle attached to the **CBitmap** object.

Operations

FromHandle	Returns a pointer to a **CBitmap** object when given a handle to a Windows **HBITMAP** bitmap.
GetBitmapBits	Copies the bits of the specified bitmap into the specified buffer.
GetBitmapDimension	Returns the width and height of the bitmap. The height and width are assumed to have been set previou sly by the SetBitmapDimension member function.
SetBitmapBits	Sets the bits of a bitmap to the specified bit values.
SetBitmapDimension	Assigns a width and height to a bitmap in 0.1-millimeter units.

See Also

Reference

CBitmap Class

Hierarchy Chart

CBitmap Member Functions

For information about the member functions in **CBitmap**, see CBitmap Members.

CBitmap::CBitmap

Constructs a **CBitmap** object.

```
CBitmap( );
```

Remarks

The resulting object must be initialized with one of the initialization member functions.

See Also

Reference

CBitmap Class
Hierarchy Chart
CBitmap::LoadBitmap
CBitmap::LoadOEMBitmap
CBitmap::CreateBitmap
CBitmap::CreateBitmapIndirect
CBitmap::CreateCompatibleBitmap
CBitmap::CreateDiscardableBitmap

Other Resources

CBitmap Members

CBitmap::CreateBitmap

Initializes a device-dependent memory bitmap that has the specified width, height, and bit pattern.

```
BOOL CreateBitmap(
   int nWidth,
   int nHeight,
   UINT nPlanes,
   UINT nBitcount,
   const void* lpBits
);
```

Parameters

nWidth

Specifies the width (in pixels) of the bitmap.

nHeight

Specifies the height (in pixels) of the bitmap.

nPlanes

Specifies the number of color planes in the bitmap.

nBitcount

Specifies the number of color bits per display pixel.

lpBits

Points to a short-integer array that contains the initial bitmap bit values. If it is **NULL**, the new bitmap is left uninitialized.

Return Value

Nonzero if successful; otherwise 0.

Remarks

For a color bitmap, either the *nPlanes* or *nBitcount* parameter should be set to 1. If both of these parameters are set to 1, **CreateBitmap** creates a monochrome bitmap.

Although a bitmap cannot be directly selected for a display device, it can be selected as the current bitmap for a "memory device context" by using CDC::SelectObject and copied to any compatible device context by using the CDC::BitBlt function.

When you finish with the **CBitmap** object created by the **CreateBitmap** function, first select the bitmap out of the device context, then delete the **CBitmap** object.

For more information, see the description of the **bmBits** field in the **BITMAP** structure. The BITMAP structure is described under the CBitmap::CreateBitmapIndirect member function.

See Also

Reference

CBitmap Class
Hierarchy Chart
CDC::SelectObject
CGdiObject::DeleteObject
CDC::BitBlt
CreateBitmap

Other Resources

CBitmap Members

CBitmap::CreateBitmapIndirect

Initializes a bitmap that has the width, height, and bit pattern (if one is specified) given in the structure pointed to by *lpBitmap*.

```
BOOL CreateBitmapIndirect(
   LPBITMAP lpBitmap
);
```

Parameters

lpBitmap

Points to a BITMAP structure that contains information about the bitmap.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Although a bitmap cannot be directly selected for a display device, it can be selected as the current bitmap for a memory device context by using CDC::SelectObject and copied to any compatible device context by using the CDC::BitBlt or CDC::StretchBlt function. (The CDC::PatBlt function can copy the bitmap for the current brush directly to the display device context.)

If the **BITMAP** structure pointed to by the *lpBitmap* parameter has been filled in by using the **GetObject** function, the bits of the bitmap are not specified and the bitmap is uninitialized. To initialize the bitmap, an application can use a function such as CDC::BitBlt or SetDIBits to copy the bits from the bitmap identified by the first parameter of **CGdiObject::GetObject** to the bitmap created by **CreateBitmapIndirect**.

When you finish with the **CBitmap** object created with **CreateBitmapIndirect** function, first select the bitmap out of the device context, then delete the **CBitmap** object.

See Also

Reference

CBitmap Class
Hierarchy Chart
CDC::SelectObject
CDC::BitBlt
CGdiObject::DeleteObject
CGdiObject::GetObject
CreateBitmapIndirect

Other Resources

CBitmap Members

CBitmap::CreateCompatibleBitmap

Initializes a bitmap that is compatible with the device specified by *pDC*.

```
BOOL CreateCompatibleBitmap(
   CDC* pDC,
   int nWidth,
   int nHeight
);
```

Parameters

pDC

Specifies the device context.

nWidth

Specifies the width (in pixels) of the bitmap.

nHeight

Specifies the height (in pixels) of the bitmap.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The bitmap has the same number of color planes or the same bits-per-pixel format as the specified device context. It can be selected as the current bitmap for any memory device that is compatible with the one specified by *pDC*.

If *pDC* is a memory device context, the bitmap returned has the same format as the currently selected bitmap in that device context. A "memory device context" is a block of memory that represents a display surface. It can be used to prepare images in memory before copying them to the actual display surface of the compatible device.

When a memory device context is created, GDI automatically selects a monochrome stock bitmap for it.

Since a color memory device context can have either color or monochrome bitmaps selected, the format of the bitmap returned by the **CreateCompatibleBitmap** function is not always the same; however, the format of a compatible bitmap for a nonmemory device context is always in the format of the device.

When you finish with the **CBitmap** object created with the **CreateCompatibleBitmap** function, first select the bitmap out of the device context, then delete the **CBitmap** object.

See Also

Reference

CBitmap Class
Hierarchy Chart
CreateCompatibleBitmap
CGdiObject::DeleteObject

Other Resources

CBitmap Members

CBitmap::CreateDiscardableBitmap

Initializes a discardable bitmap that is compatible with the device context identified by *pDC*.

```
BOOL CreateDiscardableBitmap(
   CDC* pDC,
   int nWidth,
   int nHeight
);
```

Parameters

pDC

Specifies a device context.

nWidth

Specifies the width (in bits) of the bitmap.

nHeight

Specifies the height (in bits) of the bitmap.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The bitmap has the same number of color planes or the same bits-per-pixel format as the specified device context. An application can select this bitmap as the current bitmap for a memory device that is compatible with the one specified by *pDC*.

Windows can discard a bitmap created by this function only if an application has not selected it into a display context. If Windows discards the bitmap when it is not selected and the application later attempts to select it, the CDC::SelectObject function will return **NULL**.

When you finish with the **CBitmap** object created with the **CreateDiscardableBitmap** function, first select the bitmap out of the device context, then delete the **CBitmap** object.

See Also

Reference

CBitmap Class
Hierarchy Chart
CreateDiscardableBitmap
CGdiObject::DeleteObject

Other Resources

CBitmap Members

CBitmap::FromHandle

Returns a pointer to a **CBitmap** object when given a handle to a Windows GDI bitmap.

```
static CBitmap* PASCAL FromHandle(
   HBITMAP hBitmap
);
```

Parameters

hBitmap

Specifies a Windows GDI bitmap.

Return Value

A pointer to a **CBitmap** object if successful; otherwise **NULL**.

Remarks

If a **CBitmap** object is not already attached to the handle, a temporary **CBitmap** object is created and attached. This temporary **CBitmap** object is valid only until the next time the application has idle time in its event loop, at which time all temporary graphic objects are deleted. Another way of saying this is that the temporary object is only valid during the processing of one window message.

See Also

Reference

CBitmap Class

Hierarchy Chart

Other Resources

CBitmap Members

CBitmap::GetBitmap

Call this member function to retrieve information about a **CBitmap** object.

```
int GetBitmap(
   BITMAP* pBitMap
);
```

Parameters

pBitMap

Pointer to a BITMAP structure. Must not be **NULL**.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This information is returned in the **BITMAP** structure referred to by *pBitmap*.

See Also

Reference

CBitmap Class

Hierarchy Chart

BITMAP Structure

Other Resources

CBitmap Members

CBitmap::GetBitmapBits

Copies the bit pattern of the **CBitmap** object into the buffer that is pointed to by *lpBits*.

```
DWORD GetBitmapBits(
   DWORD dwCount,
   LPVOID lpBits
) const;
```

Parameters

dwCount

Specifies the number of bytes to be copied.

lpBits

Points to the buffer that is to receive the bitmap. The bitmap is an array of bytes. The bitmap byte array conforms to a structure where horizontal scan lines are multiples of 16 bits.

Return Value

The actual number of bytes in the bitmap, or 0 if there is an error.

Remarks

The *dwCount* parameter specifies the number of bytes to be copied to the buffer. Use CGdiObject::GetObject to determine the correct *dwCount* value for the given bitmap.

See Also

Reference

CBitmap Class
Hierarchy Chart
CGdiObject::GetObject
GetBitmapBits

Other Resources

CBitmap Members

CBitmap::GetBitmapDimension

Returns the width and height of the bitmap.

```
CSize GetBitmapDimension( ) const;
```

Return Value

The width and height of the bitmap, measured in 0.1-millimeter units. The height is in the **cy** member of the **CSize** object, and the width is in the **cx** member. If the bitmap width and height have not been set by using **SetBitmapDimension**, the return value is 0.

Remarks

The height and width are assumed to have been set previously by using the SetBitmapDimension member function.

See Also

Reference

CBitmap Class
Hierarchy Chart
CBitmap::SetBitmapDimension

Other Resources

CBitmap Members

CBitmap::LoadBitmap

Loads the bitmap resource named by *lpszResourceName* or identified by the ID number in *nIDResource* from the application's executable file.

```
BOOL LoadBitmap(
   LPCTSTR lpszResourceName
);
BOOL LoadBitmap(
   UINT nIDResource
);
```

Parameters

lpszResourceName

Points to a null-terminated string that contains the name of the bitmap resource.

nIDResource

Specifies the resource ID number of the bitmap resource.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The loaded bitmap is attached to the **CBitmap** object.

If the bitmap identified by *lpszResourceName* does not exist or if there is insufficient memory to load the bitmap, the function returns 0.

You can use the CGdiObject::DeleteObject function to delete bitmap loaded by the **LoadBitmap** function, or the **CBitmap** destructor will delete the object for you.

Caution
Before you delete the object, make sure it is not selected into a device context.

The following bitmaps were added to Windows versions 3.1 and later:

OBM_UPARRROWIOBM_DNARROWIOBM_RGARROWIOBM_LFARROWI

These bitmaps are not found in device drivers for Windows versions 3.0 and earlier. For a complete list of bitmaps and a display of their appearance, see the *Platform SDK*.

See Also

Reference

CBitmap Class
Hierarchy Chart
CBitmap::LoadOEMBitmap
LoadBitmap
CGdiObject::DeleteObject

Other Resources

CBitmap Members

CBitmap::LoadMappedBitmap

Call this member function to load a bitmap and map the colors to the current system colors.

```
BOOL LoadMappedBitmap(
   UINT nIDBitmap,
   UINT nFlags = 0,
   LPCOLORMAP lpColorMap = NULL,
   int nMapSize = 0
);
```

Parameters

nIDBitmap

The ID of the bitmap resource.

nFlags

A flag for a bitmap. Can be zero or **CMB_MASKED**.

lpColorMap

A pointer to a **COLORMAP** structure that contains the color information needed to map the bitmaps. If this parameter is **NULL**, the function uses the default color map.

nMapSize

The number of color maps pointed to by *lpColorMap*.

Return Value

Nonzero if successful; otherwise 0.

Remarks

By default, **LoadMappedBitmap** will map colors commonly used in button glyphs.

For information about creating a mapped bitmap, see the Windows function CreateMappedBitmap and the COLORMAP structure in the *Platform SDK*.

See Also

Reference

CBitmap Class
Hierarchy Chart
LoadBitmap
CreateMappedBitmap

Other Resources

CBitmap Members

CBitmap::LoadOEMBitmap

Loads a predefined bitmap used by Windows.

```
BOOL LoadOEMBitmap(
   UINT nIDBitmap
);
```

Parameters

nIDBitmap

ID number of the predefined Windows bitmap. The possible values are listed below from WINDOWS.H:

OBM_BTNCORNERS	**OBM_OLD_RESTORE**
OBM_BTSIZE	**OBM_OLD_RGARROW**
OBM_CHECK	**OBM_OLD_UPARROW**
OBM_CHECKBOXES	**OBM_OLD_ZOOM**
OBM_CLOSE	**OBM_REDUCE**
OBM_COMBO	**OBM_REDUCED**
OBM_DNARROW	**OBM_RESTORE**
OBM_DNARROWD	**OBM_RESTORED**
OBM_DNARROWI	**OBM_RGARROW**
OBM_LFARROW	**OBM_RGARROWD**
OBM_LFARROWD	**OBM_RGARROWI**
OBM_LFARROWI	**OBM_SIZE**
OBM_MNARROW	**OBM_UPARROW**
OBM_OLD_CLOSE	**OBM_UPARROWD**
OBM_OLD_DNARROW	**OBM_UPARROW**
OBM_OLD_LFARROW	**OBM_ZOOM**
OBM_OLD_REDUCE	**OBM_ZOOMD**

Return Value

Nonzero if successful; otherwise 0.

Remarks

Bitmap names that begin with **OBM_OLD** represent bitmaps used by Windows versions prior to 3.0.

Note that the constant **OEMRESOURCE** must be defined before including WINDOWS.H in order to use any of the **OBM_**

constants.

See Also

Reference

CBitmap Class
Hierarchy Chart
CBitmap::LoadBitmap
LoadBitmap

Other Resources

CBitmap Members

CBitmap::operator HBITMAP

Use this operator to get the attached Windows GDI handle of the **CBitmap** object.

```
operator HBITMAP( ) const;
```

Return Value

If successful, a handle to the Windows GDI object represented by the **CBitmap** object; otherwise **NULL**.

Remarks

This operator is a casting operator, which supports direct use of an **HBITMAP** object.

For more information about using graphic objects, see Graphic Objects in the *Platform SDK*.

See Also

Reference

CBitmap Class

Hierarchy Chart

Other Resources

CBitmap Members

CBitmap::SetBitmapBits

Sets the bits of a bitmap to the bit values given by *lpBits*.

```
DWORD SetBitmapBits(
   DWORD dwCount,
   const void* lpBits
);
```

Parameters

dwCount

Specifies the number of bytes pointed to by *lpBits*.

lpBits

Points to the **BYTE** array that contains the bit values to be copied to the **CBitmap** object.

Return Value

The number of bytes used in setting the bitmap bits; 0 if the function fails.

See Also

Reference

CBitmap Class
Hierarchy Chart
SetBitmapBits

Other Resources

CBitmap Members

CBitmap::SetBitmapDimension

Assigns a width and height to a bitmap in 0.1-millimeter units.

```
CSize SetBitmapDimension(
   int nWidth,
   int nHeight
);
```

Parameters

nWidth

Specifies the width of the bitmap (in 0.1-millimeter units).

nHeight

Specifies the height of the bitmap (in 0.1-millimeter units).

Return Value

The previous bitmap dimensions. Height is in the **cy** member variable of the **CSize** object, and width is in the **cx** member variable.

Remarks

The GDI does not use these values except to return them when an application calls the GetBitmapDimension member function.

See Also

Reference

CBitmap Class
Hierarchy Chart
CBitmap::GetBitmapDimension

Other Resources

CBitmap Members

CBitmapButton Class

This topic has been updated for Visual Studio 2005 SP1.

Creates pushbutton controls labeled with bitmapped images instead of text.

```
class CBitmapButton : public CButton
```

Remarks

CBitmapButton objects contain up to four bitmaps, which contain images for the different states a button can assume: up (or normal), down (or selected), focused, and disabled. Only the first bitmap is required; the others are optional.

Bitmap-button images include the border around the image as well as the image itself. The border typically plays a part in showing the state of the button. For example, the bitmap for the focused state usually is like the one for the up state but with a dashed rectangle inset from the border or a thick solid line at the border. The bitmap for the disabled state usually resembles the one for the up state but has lower contrast (like a dimmed or grayed menu selection).

These bitmaps can be of any size, but all are treated as if they were the same size as the bitmap for the up state.

Various applications demand different combinations of bitmap images:

Up	Down	Focused	Disabled	Application
×				Bitmap
×	×			Button without **WS_TABSTOP** style
×	×	×	×	Dialog button with all states
×	×	×		Dialog button with **WS_TABSTOP** style

When creating a bitmap-button control, set the **BS_OWNERDRAW** style to specify that the button is owner-drawn. This causes Windows to send the **WM_MEASUREITEM** and **WM_DRAWITEM** messages for the button; the framework handles these messages and manages the appearance of the button for you.

To create a bitmap-button control in a window's client area

1. Create one to four bitmap images for the button.
2. Construct the CBitmapButton object.
3. Call the Create function to create the Windows button control and attach it to the **CBitmapButton** object.
4. Call the LoadBitmaps member function to load the bitmap resources after the bitmap button is constructed.

To include a bitmap-button control in a dialog box

1. Create one to four bitmap images for the button.
2. Create a dialog template with an owner-draw button positioned where you want the bitmap button. The size of the button in the template does not matter.
3. Set the button's caption to a value such as "**MYIMAGE**" and define a symbol for the button such as **IDC_MYIMAGE**.
4. In your application's resource script, give each of the images created for the button an ID constructed by appending one of the letters "U," "D," "F," or "X" (for up, down, focused, and disabled) to the string used for the button caption in step 3. For the button caption "**MYIMAGE**," for example, the IDs would be "**MYIMAGEU**," "**MYIMAGED**," "**MYIMAGEF**," and "**MYIMAGEX**." You **must** specify the ID of your bitmaps within double quotes. Otherwise the resource editor will assign an integer to the resource and MFC will fail when loading the image.
5. In your application's dialog class (derived from **CDialog**), add a **CBitmapButton** member object.

6. In the **CDialog** object's OnInitDialog routine, call the **CBitmapButton** object's AutoLoad function, using as parameters the button's control ID and the **CDialog** object's **this** pointer.

If you want to handle Windows notification messages, such as **BN_CLICKED**, sent by a bitmap-button control to its parent (usually a class derived from **CDialog)**, add to the **CDialog**-derived object a message-map entry and message-handler member function for each message. The notifications sent by a **CBitmapButton** object are the same as those sent by a CButton object.

The class CToolBar takes a different approach to bitmap buttons.

For more information on **CBitmapButton**, seeControls.

Requirements

Header: afxext.h

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

See Also

Tasks

CTRLTEST Sample: Implements Custom Controls

Reference

CButton Class

Hierarchy Chart

Other Resources

CBitmapButton Members

CBitmapButton Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CButton Members

Construction

AutoLoad	Associates a button in a dialog box with an object of the **CBitmapButton** class, loads the bitmap(s) by name, and sizes the button to fit the bitmap.
CBitmapButton	Constructs a **CBitmapButton** object.
LoadBitmaps	Initializes the object by loading one or more named bitmap resources from the application's resource file and attaching the bitmaps to the object.

Operations

SizeToContent	Sizes the button to accommodate the bitmap.

See Also

Reference

CBitmapButton Class

Hierarchy Chart

CBitmapButton Member Functions

For information about the member functions in **CBitmapButton**, see CBitmapButton Members.

CBitmapButton::AutoLoad

Associates a button in a dialog box with an object of the **CBitmapButton** class, loads the bitmap(s) by name, and sizes the button to fit the bitmap.

```
BOOL AutoLoad(
   UINT nID,
   CWnd* pParent
);
```

Parameters

nID

The button's control ID.

pParent

Pointer to the object that owns the button.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Use the **AutoLoad** function to initialize an owner-draw button in a dialog box as a bitmap button. Instructions for using this function are in the remarks for the CBitmapButton class.

Example

```
CBitmapButton myButton;

// Initialize the owner-drawn button with the id IDC_MYBUTTON as a bitmap
// button. This code is used in the OnInitDialog handler of my dialog.
myButton.AutoLoad(IDC_MYBUTTON, this);
```

See Also

Reference

CBitmapButton Class
Hierarchy Chart
CBitmapButton::LoadBitmaps
CBitmapButton::SizeToContent

Other Resources

CBitmapButton Members

CBitmapButton::CBitmapButton

Creates a **CBitmapButton** object.

```
CBitmapButton( );
```

Remarks

After creating the C++ **CBitmapButton** object, call CButton::Create to create the Windows button control and attach it to the **CBitmapButton** object.

Example

```
// Declare a bitmap button object on the stack.
CBitmapButton myButton;

// Declare a bitmap button object on the heap.
CBitmapButton* pmyButton = new CBitmapButton;
```

See Also

Reference

CBitmapButton Class
Hierarchy Chart
CBitmapButton::LoadBitmaps
CBitmapButton::AutoLoad
CBitmapButton::SizeToContent
CButton::Create

Other Resources

CBitmapButton Members

CBitmapButton::LoadBitmaps

Use this function when you want to load bitmap images identified by their resource names or ID numbers, or when you cannot use the **AutoLoad** function because, for example, you are creating a bitmap button that is not part of a dialog box.

```
BOOL LoadBitmaps(
   LPCTSTR lpszBitmapResource,
   LPCTSTR lpszBitmapResourceSel = NULL,
   LPCTSTR lpszBitmapResourceFocus = NULL,
   LPCTSTR lpszBitmapResourceDisabled = NULL
);
BOOL LoadBitmaps(
   UINT nIDBitmapResource,
   UINT nIDBitmapResourceSel = 0,
   UINT nIDBitmapResourceFocus = 0,
   UINT nIDBitmapResourceDisabled = 0
);
```

Parameters

lpszBitmapResource

Points to the null-terminated string that contains the name of the bitmap for a bitmap button's normal or "up" state. Required.

lpszBitmapResourceSel

Points to the null-terminated string that contains the name of the bitmap for a bitmap button's selected or "down" state. May be **NULL**.

lpszBitmapResourceFocus

Points to the null-terminated string that contains the name of the bitmap for a bitmap button's focused state. May be **NULL**.

lpszBitmapResourceDisabled

Points to the null-terminated string that contains the name of the bitmap for a bitmap button's disabled state. May be **NULL**.

nIDBitmapResource

Specifies the resource ID number of the bitmap resource for a bitmap button's normal or "up" state. Required.

nIDBitmapResourceSel

Specifies the resource ID number of the bitmap resource for a bitmap button's selected or "down" state. May be 0.

nIDBitmapResourceFocus

Specifies the resource ID number of the bitmap resource for a bitmap button's focused state. May be 0.

nIDBitmapResourceDisabled

Specifies the resource ID number of the bitmap resource for a bitmap button's disabled state. May be 0.

Return Value

Nonzero if successful; otherwise 0.

Example

```
CBitmapButton myButton;

// Create the bitmap button (must include the BS_OWNERDRAW style).
myButton.Create(NULL, WS_CHILD|WS_VISIBLE|BS_OWNERDRAW,
   CRect(10,10,100,100), pParentWnd, 1);

// Load the bitmaps for this button.
myButton.LoadBitmaps(IDB_UP, IDB_DOWN, IDB_FOCUS, IDB_DISABLE);
```

See Also

Reference

CBitmapButton Class
Hierarchy Chart
CBitmapButton::AutoLoad
CBitmapButton::SizeToContent
CButton::Create
CBitmap::LoadBitmap

Other Resources

CBitmapButton Members

CBitmapButton::SizeToContent

Call this function to resize a bitmap button to the size of the bitmap.

```
void SizeToContent( );
```

Example

```
CBitmapButton myButton;

// Create the bitmap button (must include the BS_OWNERDRAW style).
myButton.Create(NULL, WS_CHILD|WS_VISIBLE|BS_OWNERDRAW,
   CRect(10,10,100,100), pParentWnd, 1);

// Load the bitmaps for this button.
myButton.LoadBitmaps(IDB_UP, IDB_DOWN, IDB_FOCUS, IDB_DISABLE);

// Resize the button to be the size of the bitmaps.
myButton.SizeToContent();
```

See Also

Reference

CBitmapButton Class
Hierarchy Chart
CBitmapButton::LoadBitmaps
CBitmapButton::AutoLoad

Other Resources

CBitmapButton Members

CBrush Class

Encapsulates a Windows graphics device interface (GDI) brush.

```
class CBrush : public CGdiObject
```

Remarks

To use a **CBrush** object, construct a **CBrush** object and pass it to any **CDC** member function that requires a brush.

Brushes can be solid, hatched, or patterned.

For more information on **CBrush**, see Graphic Objects.

Requirements

Header: afxwin.h

See Also

Tasks

PROPDLG Sample: Demonstrates Property Sheet Support

Reference

CGdiObject Class

Hierarchy Chart

CBitmap Class

CDC Class

Other Resources

CBrush Members

CBrush Members

Base Class Members

CObject Members

CGdiObject Members

Construction

CBrush	Constructs a **CBrush** object.

Initialization

CreateBrushIndirect	Initializes a brush with the style, color, and pattern specified in a LOGBRUSH structure.
CreateDIBPatternBrush	Initializes a brush with a pattern specified by a device-independent bitmap (DIB).
CreateHatchBrush	Initializes a brush with the specified hatched pattern and color.
CreatePatternBrush	Initializes a brush with a pattern specified by a bitmap.
CreateSolidBrush	Initializes a brush with the specified solid color.
CreateSysColorBrush	Creates a brush that is the default system color.

Operations

FromHandle	Returns a pointer to a **CBrush** object when given a handle to a Windows **HBRUSH** object.

Attributes

GetLogBrush	Gets a LOGBRUSH structure.
operator HBRUSH	Returns the Windows handle attached to the **CBrush** object.

See Also

Reference

CBrush Class

Hierarchy Chart

CBrush Member Functions

For information about the member functions in **CBrush**, see CBrush Members.

CBrush::CBrush

Constructs a **CBrush** object.

```
CBrush( );
CBrush(
   COLORREF crColor
);
CBrush(
   int nIndex,
   COLORREF crColor
);
explicit CBrush(
   CBitmap* pBitmap
);
```

Parameters

crColor

Specifies the foreground color of the brush as an RGB color. If the brush is hatched, this parameter specifies the color of the hatching.

nIndex

Specifies the hatch style of the brush. It can be any one of the following values:

- **HS_BDIAGONAL** Downward hatch (left to right) at 45 degrees
- **HS_CROSS** Horizontal and vertical crosshatch
- **HS_DIAGCROSS** Crosshatch at 45 degrees
- **HS_FDIAGONAL** Upward hatch (left to right) at 45 degrees
- **HS_HORIZONTAL** Horizontal hatch
- **HS_VERTICAL** Vertical hatch

pBitmap

Points to a **CBitmap** object that specifies a bitmap with which the brush paints.

Remarks

CBrush has four overloaded constructors.The constructor with no arguments constructs an uninitialized **CBrush** object that must be initialized before it can be used.

If you use the constructor with no arguments, you must initialize the resulting **CBrush** object with CreateSolidBrush, CreateHatchBrush, CreateBrushIndirect, CreatePatternBrush, or CreateDIBPatternBrush. If you use one of the constructors that takes arguments, then no further initialization is necessary. The constructors with arguments can throw an exception if errors are encountered, while the constructor with no arguments will always succeed.

The constructor with a single COLORREF parameter constructs a solid brush with the specified color. The color specifies an RGB value and can be constructed with the **RGB** macro in WINDOWS.H.

The constructor with two parameters constructs a hatch brush. The *nIndex* parameter specifies the index of a hatched pattern. The *crColor* parameter specifies the color.

The constructor with a **CBitmap** parameter constructs a patterned brush. The parameter identifies a bitmap. The bitmap is assumed to have been created by using CBitmap::CreateBitmap, CBitmap::CreateBitmapIndirect, CBitmap::LoadBitmap, or CBitmap::CreateCompatibleBitmap. The minimum size for a bitmap to be used in a fill pattern is 8 pixels by 8 pixels.

Example

```
void CBrushView::OnDraw(CDC* pDC)
{
   CBrushDoc* pDoc = GetDocument();
```

```
    ASSERT_VALID(pDoc);

    // CBrush::CBrush.
    CBrush brush1;   // Must initialize!
    brush1.CreateSolidBrush(RGB(0,0,255));   // Blue brush.

    CBrush* pTempBrush = NULL;
    CBrush OrigBrush;

    CRect rc;
    GetClientRect(&rc);
    ScreenToClient(&rc);

    pTempBrush = (CBrush*)pDC->SelectObject(&brush1);
    // Save original brush.
    OrigBrush.FromHandle((HBRUSH)pTempBrush);

    // Paint upper left corner with blue brush.
    pDC->Rectangle(0, 0, rc.Width() / 2, rc.Height() / 2);

    // These constructors throw resource exceptions.
    try
    {
       // CBrush::CBrush(COLORREF crColor)
       CBrush brush2(RGB(255,0,0));   // Solid red brush.

       // CBrush::CBrush(int nIndex, COLORREF crColor)
       // Hatched green brush.
       CBrush brush3(HS_DIAGCROSS, RGB(0,255,0));

       // CBrush::CBrush(CBitmap* pBitmap)
       CBitmap bmp;
       // Load a resource bitmap.
       bmp.LoadBitmap(IDB_BRUSH);
       CBrush brush4(&bmp);

       pTempBrush = (CBrush*)pDC->SelectObject(&brush2);

       // Paint upper right corner with red brush.
       pDC->Rectangle(rc.Width() / 2, 0, rc.Width(),
           rc.Height() / 2);

       pTempBrush = (CBrush*)pDC->SelectObject(&brush3);

       // Paint lower left corner with green hatched brush.
       pDC->Rectangle(0, rc.Height() / 2, rc.Width() / 2,
           rc.Height());

       pTempBrush = (CBrush*)pDC->SelectObject(&brush4);

       // Paint lower right corner with resource brush.
       pDC->Rectangle(rc.Width() / 2, rc.Height() / 2,
           rc.Width(), rc.Height());
    }
    catch(CResourceException* e)
    {
       e->ReportError();
       e->Delete();
    }

    // Reselect original brush into device context.
    pDC->SelectObject(&OrigBrush);
 }
```

See Also

Reference

CBrush Class

Hierarchy Chart

CBrush::CreateSolidBrush
CBrush::CreateHatchBrush
CBrush::CreateBrushIndirect
CBrush::CreatePatternBrush
CBrush::CreateDIBPatternBrush
CGdiObject::CreateStockObject

Other Resources

CBrush Members

CBrush::CreateBrushIndirect

Initializes a brush with a style, color, and pattern specified in a LOGBRUSH structure.

```
BOOL CreateBrushIndirect(
   const LOGBRUSH* lpLogBrush
);
```

Parameters

lpLogBrush

Points to a LOGBRUSH structure that contains information about the brush.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The brush can subsequently be selected as the current brush for any device context.

A brush created using a monochrome (1 plane, 1 bit per pixel) bitmap is drawn using the current text and background colors. Pixels represented by a bit set to 0 will be drawn with the current text color. Pixels represented by a bit set to 1 will be drawn with the current background color.

Example

```
// Initialize a LOGBRUSH structure.
LOGBRUSH logBrush;
logBrush.lbStyle = BS_HATCHED;
logBrush.lbColor = RGB(0, 192, 192);
logBrush.lbHatch = HS_CROSS;

// Declare an uninitialized CBrush ...
CBrush brush;
// ... and initialize it with the LOGBRUSH.
brush.CreateBrushIndirect(&logBrush);

// Select the brush (and perhaps a pen) into
// the device context.
CBrush* pOldBrush = (CBrush*)pDC->SelectObject(&brush);
CPen* pOldPen = (CPen*)pDC->SelectStockObject(BLACK_PEN);

// Have fun!
pDC->Pie(CRect(100, 100, 300, 300), CPoint(0, 0), CPoint(50, 200));

// Restore the original device context objects.
pDC->SelectObject(pOldBrush);
pDC->SelectObject(pOldPen);
```

See Also

Reference

CBrush Class
Hierarchy Chart
CBrush::CreateDIBPatternBrush
CBrush::CreatePatternBrush
CBrush::CreateSolidBrush
CBrush::CreateHatchBrush
CGdiObject::CreateStockObject
CGdiObject::DeleteObject
CreateBrushIndirect

Other Resources

CBrush Members

CBrush::CreateDIBPatternBrush

Initializes a brush with the pattern specified by a device-independent bitmap (DIB).

```
BOOL CreateDIBPatternBrush(
      HGLOBAL hPackedDIB,
   UINT nUsage
);
BOOL CreateDIBPatternBrush(
   const void* lpPackedDIB,
   UINT nUsage
);
```

Parameters

hPackedDIB

Identifies a global-memory object containing a packed device-independent bitmap (DIB).

nUsage

Specifies whether the **bmiColors[]** fields of the BITMAPINFO data structure (a part of the "packed DIB") contain explicit RGB values or indices into the currently realized logical palette. The parameter must be one of the following values:

- **DIB_PAL_COLORS** The color table consists of an array of 16-bit indexes.
- **DIB_RGB_COLORS** The color table contains literal RGB values.

lpPackedDIB

Points to a packed DIB consisting of a **BITMAPINFO** structure immediately followed by an array of bytes defining the pixels of the bitmap.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The brush can subsequently be selected for any device context that supports raster operations.

The two versions differ in the way you handle the DIB:

- In the first version, to obtain a handle to the DIB you call the Windows **GlobalAlloc** function to allocate a block of global memory and then fill the memory with the packed DIB.
- In the second version, it is not necessary to call **GlobalAlloc** to allocate memory for the packed DIB.

A packed DIB consists of a **BITMAPINFO** data structure immediately followed by the array of bytes that defines the pixels of the bitmap. Bitmaps used as fill patterns should be 8 pixels by 8 pixels. If the bitmap is larger, Windows creates a fill pattern using only the bits corresponding to the first 8 rows and 8 columns of pixels in the upper-left corner of the bitmap.

When an application selects a two-color DIB pattern brush into a monochrome device context, Windows ignores the colors specified in the DIB and instead displays the pattern brush using the current text and background colors of the device context. Pixels mapped to the first color (at offset 0 in the DIB color table) of the DIB are displayed using the text color. Pixels mapped to the second color (at offset 1 in the color table) are displayed using the background color.

For information about using the following Windows functions, see the *Platform SDK*:

- CreateDIBPatternBrush (This function is provided only for compatibility with applications written for versions of Windows earlier than 3.0; use the **CreateDIBPatternBrushPt** function.)
- CreateDIBPatternBrushPt (This function should be used for Win32-based applications.)
- GlobalAlloc

Example

```
// Resource handle to bitmap.
HRSRC hRes;
// Global handles to bitmap resource.
HGLOBAL hData;
void* hLockedData;
CBrush brush;

// Find the resource handle.
hRes = ::FindResource(AfxGetResourceHandle(),
    MAKEINTRESOURCE(IDB_BRUSH), RT_BITMAP);
if (hRes != NULL)
{
   // Lock and Load (or Load and Lock).
   if (((hData = ::LoadResource(AfxGetResourceHandle(),
       hRes)) != NULL) &&
      ((hLockedData = ::LockResource(hData)) != NULL))
   {
      // Initialize the brush.
      brush.CreateDIBPatternBrush((const void*)hLockedData,
          DIB_RGB_COLORS);

      // Select the brush into the device context.
      CBrush* pOldBrush = pDC->SelectObject(&brush);

      // Draw.
      pDC->Rectangle(50, 50, 200, 200);

      // Restore the original device context.
      pDC->SelectObject(pOldBrush);

      // Free the resource.
      ::FreeResource(hLockedData);
   }
}
```

See Also

Reference

CBrush Class
Hierarchy Chart
CBrush::CreatePatternBrush
CBrush::CreateBrushIndirect
CBrush::CreateSolidBrush
CBrush::CreateHatchBrush
CGdiObject::CreateStockObject
CDC::SelectObject
CGdiObject::DeleteObject
CDC::GetBrushOrg
CDC::SetBrushOrg

Other Resources

CBrush Members

CBrush::CreateHatchBrush

Initializes a brush with the specified hatched pattern and color.

```
BOOL CreateHatchBrush(
   int nIndex,
   COLORREF crColor
);
```

Parameters

nIndex

Specifies the hatch style of the brush. It can be any one of the following values:

- **HS_BDIAGONAL** Downward hatch (left to right) at 45 degrees
- **HS_CROSS** Horizontal and vertical crosshatch
- **HS_DIAGCROSS** Crosshatch at 45 degrees
- **HS_FDIAGONAL** Upward hatch (left to right) at 45 degrees
- **HS_HORIZONTAL** Horizontal hatch
- **HS_VERTICAL** Vertical hatch

crColor

Specifies the foreground color of the brush as an RGB color (the color of the hatches). See COLORREF in the *Platform SDK* for more information.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The brush can subsequently be selected as the current brush for any device context.

Example

```
CBrush brush;
brush.CreateHatchBrush(HS_BDIAGONAL, RGB(255, 0, 0));

CBrush* pOldBrush;
CPen* pOldPen;

pOldBrush = (CBrush*)pDC->SelectObject(&brush);
pOldPen = (CPen*)pDC->SelectStockObject(NULL_PEN);
pDC->Ellipse(CRect(50, 50, 250, 250));

pDC->SelectObject(pOldBrush);
pDC->SelectObject(pOldPen);
```

See Also

Reference

CBrush Class
Hierarchy Chart
CBrush::CreateBrushIndirect
CBrush::CreateDIBPatternBrush
CBrush::CreatePatternBrush
CBrush::CreateSolidBrush
CGdiObject::CreateStockObject
CreateHatchBrush

Other Resources

CBrush::CreatePatternBrush

Initializes a brush with a pattern specified by a bitmap.

```
BOOL CreatePatternBrush(
   CBitmap* pBitmap
);
```

Parameters

pBitmap

Identifies a bitmap.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The brush can subsequently be selected for any device context that supports raster operations. The bitmap identified by *pBitmap* is typically initialized by using the CBitmap::CreateBitmap, CBitmap::CreateBitmapIndirect, CBitmap::LoadBitmap, or CBitmap::CreateCompatibleBitmap function.

Bitmaps used as fill patterns should be 8 pixels by 8 pixels. If the bitmap is larger, Windows will only use the bits corresponding to the first 8 rows and columns of pixels in the upper-left corner of the bitmap.

A pattern brush can be deleted without affecting the associated bitmap. This means the bitmap can be used to create any number of pattern brushes.

A brush created using a monochrome bitmap (1 color plane, 1 bit per pixel) is drawn using the current text and background colors. Pixels represented by a bit set to 0 are drawn with the current text color. Pixels represented by a bit set to 1 are drawn with the current background color.

For information about using CreatePatternBrush, a Windows function, see the *Platform SDK*.

Example

```
// Create a hatched bit pattern.
WORD HatchBits[8] = { 0x11, 0x22, 0x44, 0x88, 0x11,
    0x22, 0x44, 0x88 };

// Use the bit pattern to create a bitmap.
CBitmap bm;
bm.CreateBitmap(8,8,1,1, HatchBits);

// Create a pattern brush from the bitmap.
CBrush brush;
brush.CreatePatternBrush(&bm);

// Select the brush into a device context, and draw.
CBrush* pOldBrush = (CBrush*)pDC->SelectObject(&brush);
pDC->RoundRect(CRect(50, 50, 200, 200), CPoint(10,10));

// Restore the original brush.
pDC->SelectObject(pOldBrush);
```

See Also

Reference

CBrush Class
Hierarchy Chart
CBitmap Class
CBrush::CreateBrushIndirect
CBrush::CreateDIBPatternBrush
CBrush::CreateHatchBrush
CBrush::CreateSolidBrush

CGdiObject::CreateStockObject

Other Resources

CBrush Members

CBrush::CreateSolidBrush

Initializes a brush with a specified solid color.

```
BOOL CreateSolidBrush(
   COLORREF crColor
);
```

Parameters

crColor

A COLORREF structure that specifies the color of the brush. The color specifies an RGB value and can be constructed with the **RGB** macro in WINDOWS.H.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The brush can subsequently be selected as the current brush for any device context.

When an application has finished using the brush created by **CreateSolidBrush**, it should select the brush out of the device context.

Example

See the example for CBrush::CBrush.

See Also

Reference

CBrush Class
Hierarchy Chart
CBrush::CreateBrushIndirect
CBrush::CreateDIBPatternBrush
CBrush::CreateHatchBrush
CBrush::CreatePatternBrush
CreateSolidBrush
CGdiObject::DeleteObject

Other Resources

CBrush Members

CBrush::CreateSysColorBrush

Initializes a brush color.

```
BOOL CreateSysColorBrush(
   int nIndex
);
```

Parameters

nIndex

Specifies a color index. This value corresponds to the color used to paint one of the 21 window elements. See GetSysColor in the *Platform SDK* for a list of values.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The brush can subsequently be selected as the current brush for any device context.

When an application has finished using the brush created by **CreateSysColorBrush**, it should select the brush out of the device context.

Example

```
// Declare a CBrush and initialize to a system color.
CBrush brush;
brush.CreateSysColorBrush(COLOR_BTNFACE);

// Select the brush into the device context.
CBrush* pOldBrush = (CBrush*)pDC->SelectObject(&brush);

// Draw.
CRect rect(50, 50, 150, 150);
pDC->Rectangle(rect);

// Reselect the original brush.
pDC->SelectObject(pOldBrush);
```

See Also

Reference

CBrush Class
Hierarchy Chart
CBrush::CreateBrushIndirect
CBrush::CreateDIBPatternBrush
CBrush::CreateHatchBrush
CBrush::CreatePatternBrush
CreateSolidBrush
CBrush::CreateSolidBrush
GetSysColorBrush
CGdiObject::DeleteObject

Other Resources

CBrush Members

CBrush::FromHandle

Returns a pointer to a **CBrush** object when given a handle to a Windows HBRUSH object.

```
static CBrush* PASCAL FromHandle(
   HBRUSH hBrush
);
```

Parameters

hBrush

HANDLE to a Windows GDI brush.

Return Value

A pointer to a **CBrush** object if successful; otherwise **NULL**.

Remarks

If a **CBrush** object is not already attached to the handle, a temporary **CBrush** object is created and attached. This temporary **CBrush** object is valid only until the next time the application has idle time in its event loop. At this time, all temporary graphic objects are deleted. In other words, the temporary object is valid only during the processing of one window message.

For more information about using graphic objects, see Graphic Objects in the *Platform SDK*.

Example

See the example for CBrush::CBrush.

See Also

Reference

CBrush Class

Hierarchy Chart

Other Resources

CBrush Members

CBrush::GetLogBrush

Call this member function to retrieve the **LOGBRUSH** structure.

```
int GetLogBrush(
   LOGBRUSH* pLogBrush
);
```

Parameters

pLogBrush

Points to a LOGBRUSH structure that contains information about the brush.

Return Value

If the function succeeds, and *pLogBrush* is a valid pointer, the return value is the number of bytes stored into the buffer.

If the function succeeds, and *pLogBrush* is **NULL**, the return value is the number of bytes required to hold the information the function would store into the buffer.

If the function fails, the return value is 0.

Remarks

The **LOGBRUSH** structure defines the style, color, and pattern of a brush.

For example, call **GetLogBrush** to match the particular color or pattern of a bitmap.

Example

```
// Example for CBrush::GetLogBrush
LOGBRUSH logbrush;
brushExisting.GetLogBrush( &logbrush );
CBrush brushOther( logbrush.lbColor);

// Another example
// Declare a LOGBRUSH
LOGBRUSH logBrush;

// Using a bitmap for this example.
// The bitmap should be a project resource.
CBitmap bm;
bm.LoadBitmap(IDB_BRUSH);

try
{
   // Create a brush
   CBrush brush1(&bm);

   // Use GetLogBrush to fill the LOGBRUSH structure
   brush1.GetLogBrush(&logBrush);

   // Create a second brush using the LOGBRUSH data
   CBrush brush2;
   brush2.CreateBrushIndirect(&logBrush);

   // Use the first brush
   CBrush* pOldBrush = (CBrush*)pDC->SelectObject(&brush1);
   pDC->Rectangle(CRect(50,50,150,150));

   // The second brush has the specified characteristics
   // of the first brush
   CBrush* pTempBrush = (CBrush*)pDC->SelectObject(&brush2);
   pDC->Ellipse(200,50,300,150);

   // Reselect the original brush
```

```
    pDC->SelectObject(pOldBrush);
}
catch(CResourceException* e)
{
    e->ReportError();
    e->Delete();
}
```

See Also

Reference

CBrush Class

Hierarchy Chart

LOGBRUSH

GetObject

Other Resources

CBrush Members

CBrush::operator HBRUSH

Use this operator to get the attached Windows GDI handle of the **CBrush** object.

```
operator HBRUSH( ) const;
```

Return Value

If successful, a handle to the Windows GDI object represented by the **CBrush** object; otherwise **NULL**.

Remarks

This operator is a casting operator, which supports direct use of an **HBRUSH** object.

For more information about using graphic objects, see Graphic Objects in the *Platform SDK*.

Example

```
RECT rc = { 50, 50, 200, 200 };

Rectangle(hDC, rc.left, rc.top, rc.right, rc.bottom);

// The Win32 call to FillRect requires an HBRUSH.
// The HBRUSH operator used here makes a temporary
// CBrush object which is cast to the required type.
FillRect(hDC, &rc, (HBRUSH)(COLOR_BTNFACE+1));
```

See Also

Reference

CBrush Class

Hierarchy Chart

Other Resources

CBrush Members

CButton Class

Provides the functionality of Windows button controls.

```
class CButton : public CWnd
```

Remarks

A button control is a small, rectangular child window that can be clicked on and off. Buttons can be used alone or in groups and can either be labeled or appear without text. A button typically changes appearance when the user clicks it.

Typical buttons are the check box, radio button, and pushbutton. A **CButton** object can become any of these, according to the button style specified at its initialization by the Create member function.

In addition, the CBitmapButton class derived from **CButton** supports creation of button controls labeled with bitmap images instead of text. A **CBitmapButton** can have separate bitmaps for a button's up, down, focused, and disabled states.

You can create a button control either from a dialog template or directly in your code. In both cases, first call the constructor **CButton** to construct the **CButton** object; then call the **Create** member function to create the Windows button control and attach it to the **CButton** object.

Construction can be a one-step process in a class derived from **CButton**. Write a constructor for the derived class and call **Create** from within the constructor.

If you want to handle Windows notification messages sent by a button control to its parent (usually a class derived from CDialog), add a message-map entry and message-handler member function to the parent class for each message.

Each message-map entry takes the following form:

ON_Notification**(** *id*, *memberFxn* **)**

where *id* specifies the child window ID of the control sending the notification and *memberFxn* is the name of the parent member function you have written to handle the notification.

The parent's function prototype is as follows:

afx_msg void *memberFxn***( );**

Potential message-map entries are as follows:

Map entry	Sent to parent when...
ON_BN_CLICKED	The user clicks a button.
ON_BN_DOUBLECLICKED	The user double-clicks a button.

If you create a **CButton** object from a dialog resource, the **CButton** object is automatically destroyed when the user closes the dialog box.

If you create a **CButton** object within a window, you may need to destroy it. If you create the **CButton** object on the heap by using the **new** function, you must call **delete** on the object to destroy it when the user closes the Windows button control. If you create the **CButton** object on the stack, or it is embedded in the parent dialog object, it is destroyed automatically.

Requirements

Header: afxwin.h

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd Class
CComboBox Class
CEdit Class
CListBox Class

CScrollBar Class
CStatic Class
CBitmapButton Class
CDialog Class
Other Resources
CButton Members

CButton Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Construction

CButton	Constructs a **CButton** object.

Initialization

Create	Creates the Windows button control and attaches it to the **CButton** object.

Operations

GetBitmap	Retrieves the handle of the bitmap previously set with SetBitmap.
GetButtonStyle	Retrieves information about the button control style.
GetCheck	Retrieves the check state of a button control.
GetCursor	Retrieves the handle of the cursor image previously set with SetCursor.
GetIdealSize	Retrieves the ideal size of the button control.
GetImageList	Retrieves the image list of the button control.
GetTextMargin	Retrieves the text margin of the button control.
GetIcon	Retrieves the handle of the icon previously set with SetIcon.
GetState	Retrieves the check state, highlight state, and focus state of a button control.
SetBitmap	Specifies a bitmap to be displayed on the button.
SetButtonStyle	Changes the style of a button.
SetCheck	Sets the check state of a button control.
SetCursor	Specifies a cursor image to be displayed on the button.
SetImageList	Retrieves the image list of the button control.
SetTextMargin	Retrieves the text margin of the button control.
SetIcon	Specifies an icon to be displayed on the button.
SetState	Sets the highlighting state of a button control.

Overridables

DrawItem	Override to draw an owner-drawn **CButton** object.

See Also

Reference

CButton Class

Hierarchy Chart

CButton Member Functions

For information about the member functions in **CButton**, see CButton Members.

CButton::CButton

Constructs a **CButton** object.

```
CButton( );
```

Example

```
// Declare a button object.
CButton myButton;
```

See Also

Reference

CButton Class

Hierarchy Chart

CButton::Create

Other Resources

CButton Members

CButton::Create

Creates the Windows button control and attaches it to the **CButton** object.

```
virtual BOOL Create(
   LPCTSTR lpszCaption,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

lpszCaption

Specifies the button control's text.

dwStyle

Specifies the button control's style. Apply any combination of button styles to the button.

rect

Specifies the button control's size and position. It can be either a **CRect** object or a **RECT** structure.

pParentWnd

Specifies the button control's parent window, usually a **CDialog**. It must not be **NULL**.

nID

Specifies the button control's ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

You construct a **CButton** object in two steps. First, call the constructor and then call **Create**, which creates the Windows button control and attaches it to the **CButton** object.

If the **WS_VISIBLE** style is given, Windows sends the button control all the messages required to activate and show the button.

Apply the following window styles to a button control:

- **WS_CHILD** Always
- **WS_VISIBLE** Usually
- **WS_DISABLED** Rarely
- **WS_GROUP** To group controls
- **WS_TABSTOP** To include the button in the tabbing order

Example

```
CButton myButton1, myButton2, myButton3, myButton4;

// Create a push button.
myButton1.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
   CRect(10,10,100,30), pParentWnd, 1);

// Create a radio button.
myButton2.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_RADIOBUTTON,
   CRect(10,40,100,70), pParentWnd, 2);

// Create an auto 3-state button.
```

```
myButton3.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_AUTO3STATE,
   CRect(10,70,100,100), pParentWnd, 3);

// Create an auto check box.
myButton4.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_AUTOCHECKBOX,
   CRect(10,100,100,130), pParentWnd, 4);
```

See Also

Reference

CButton Class

Hierarchy Chart

CButton::CButton

Other Resources

CButton Members

CButton::DrawItem

Called by the framework when a visual aspect of an owner-drawn button has changed.

```
virtual void DrawItem(
   LPDRAWITEMSTRUCT lpDrawItemStruct
);
```

Parameters

lpDrawItemStruct

A long pointer to a DRAWITEMSTRUCT structure. The structure contains information about the item to be drawn and the type of drawing required.

Remarks

An owner-drawn button has the **BS_OWNERDRAW** style set. Override this member function to implement drawing for an owner-drawn **CButton** object. The application should restore all graphics device interface (GDI) objects selected for the display context supplied in *lpDrawItemStruct* before the member function terminates.

Also see the BS_ style values.

Example

```
// NOTE: CMyButton is a class derived from CButton. The CMyButton
// object was created as follows:
//
// CMyButton myButton;
// myButton.Create(_T("My button"),
//      WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON|BS_OWNERDRAW,
//      CRect(10,10,100,30), pParentWnd, 1);
//

// This example implements the DrawItem method for a CButton-derived
// class that draws the button's text using the color red.
void CMyButton::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
   UINT uStyle = DFCS_BUTTONPUSH;

   // This code only works with buttons.
   ASSERT(lpDrawItemStruct->CtlType == ODT_BUTTON);

   // If drawing selected, add the pushed style to DrawFrameControl.
   if (lpDrawItemStruct->itemState & ODS_SELECTED)
      uStyle |= DFCS_PUSHED;

   // Draw the button frame.
   ::DrawFrameControl(lpDrawItemStruct->hDC, &lpDrawItemStruct->rcItem,
      DFC_BUTTON, uStyle);

   // Get the button's text.
   CString strText;
   GetWindowText(strText);

   // Draw the button text using the text color red.
   COLORREF crOldColor = ::SetTextColor(lpDrawItemStruct->hDC, RGB(255,0,0));
   ::DrawText(lpDrawItemStruct->hDC, strText, strText.GetLength(),
      &lpDrawItemStruct->rcItem, DT_SINGLELINE|DT_VCENTER|DT_CENTER);
   ::SetTextColor(lpDrawItemStruct->hDC, crOldColor);
}
```

See Also

Reference

CButton Class

Hierarchy Chart
CButton::SetButtonStyle
WM_DRAWITEM
Other Resources
CButton Members

CButton::GetBitmap

Call this member function to get the handle of a bitmap, previously set with SetBitmap, that is associated with a button.

```
HBITMAP GetBitmap( ) const;
```

Return Value

A handle to a bitmap. **NULL** if no bitmap is previously specified.

Example

```
CButton myButton;

// Create a bitmap button.
myButton.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_BITMAP,
   CRect(10,10,60,50), pParentWnd, 1);

// If no bitmap is defined for the button, define the bitmap to the
// system close bitmap.
if (myButton.GetBitmap() == NULL)
   myButton.SetBitmap( ::LoadBitmap(NULL, MAKEINTRESOURCE(OBM_CLOSE)) );
```

See Also

Reference

CButton Class
Hierarchy Chart
CButton::SetBitmap
CBitmapButton::LoadBitmaps
Bitmaps

Other Resources

CButton Members

CButton::GetButtonStyle

Retrieves information about the button control style.

```
UINT GetButtonStyle( ) const;
```

Return Value

Returns the button styles for this **CButton** object. This function returns only the BS_ style values, not any of the other window styles.

Example

```
CButton myButton;

// Create a radio button.
myButton.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_RADIOBUTTON,
   CRect(10,10,100,30), pParentWnd, 1);

// Change the button style to use one of the "auto" styles; for
// push button, change to def push button.
UINT uStyle = myButton.GetButtonStyle();
if (uStyle == BS_PUSHBUTTON)
   uStyle = BS_DEFPUSHBUTTON;
else if (uStyle == BS_RADIOBUTTON)
   uStyle = BS_AUTORADIOBUTTON;
else if (uStyle == BS_CHECKBOX)
   uStyle = BS_AUTOCHECKBOX;
else if (uStyle == BS_3STATE)
   uStyle = BS_AUTO3STATE;

// Change the button style to the one wanted.
myButton.SetButtonStyle( uStyle );
```

See Also

Reference

CButton Class
Hierarchy Chart
CButton::SetButtonStyle
GetWindowLong

Other Resources

CButton Members

CButton::GetCheck

Retrieves the check state of a radio button or check box.

```
int GetCheck( ) const;
```

Return Value

The return value from a button control created with the **BS_AUTOCHECKBOX**, **BS_AUTORADIOBUTTON**, **BS_AUTO3STATE**, **BS_CHECKBOX**, **BS_RADIOBUTTON**, or **BS_3STATE** style is one of the following values:

Value	Meaning
BST_UNCHECKED	Button state is unchecked.
BST_CHECKED	Button state is checked.
BST_INDETERMINATE	Button state is indeterminate (applies only if the button has the **BS_3STATE** or **BS_AUTO3STATE** style).

If the button has any other style, the return value is **BST_UNCHECKED**.

Example

```
CButton myButton;

// Create an auto 3-state button.
myButton.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_AUTO3STATE,
   CRect(10,10,100,30), pParentWnd, 1);

// Set the check state to the next state
// (i.e. BST_UNCHECKED changes to BST_CHECKED
// BST_CHECKED changes to BST_INDETERMINATE
// BST_INDETERMINATE changes to BST_UNCHECKED).
myButton.SetCheck( ((myButton.GetCheck()+ 1 ) % 3) );
```

See Also

Reference

CButton Class
Hierarchy Chart
CButton::GetState
CButton::SetState
CButton::SetCheck
BM_GETCHECK

Other Resources

CButton Members

CButton::GetCursor

Call this member function to get the handle of a cursor, previously set with SetCursor, that is associated with a button.

```
HCURSOR GetCursor( );
```

Return Value

A handle to a cursor image. **NULL** if no cursor is previously specified.

Example

```
CButton myButton;

// Create an icon button.
myButton.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_ICON,
   CRect(10,10,60,50), pParentWnd, 1);

// If no image is defined for the button, define the image to the
// system arrow and question mark cursor.
if (myButton.GetCursor() == NULL)
   myButton.SetCursor( ::LoadCursor(NULL, IDC_HELP) );
```

See Also

Reference

CButton Class
Hierarchy Chart
CButton::SetCursor
CBitmapButton::LoadBitmaps
Bitmaps

Other Resources

CButton Members

CButton::GetIcon

Call this member function to get the handle of an icon, previously set with SetIcon, that is associated with a button.

```
HICON GetIcon( ) const;
```

Return Value

A handle to an icon. **NULL** if no icon is previously specified.

Example

```
CButton myButton;

// Create an icon button.
myButton.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_ICON,
   CRect(10,10,60,50), pParentWnd, 1);

// If no icon is defined for the button, define the icon to the
// system error icon.
if (myButton.GetIcon() == NULL)
   myButton.SetIcon( ::LoadIcon(NULL, IDI_ERROR) );
```

See Also

Reference

CButton Class
Hierarchy Chart
CButton::SetIcon
CBitmapButton::LoadBitmaps
Bitmaps

Other Resources

CButton Members

CButton::GetIdealSize

Retrieves the ideal size for the button control.

```
BOOL GetIdealSize(
   SIZE* psize
);
```

Parameters

psize

A pointer to the current size of the button.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function emulates the functionality of the **BCM_GETIDEALSIZE** message, as described in the Buttons section of the *Platform SDK*.

See Also

Reference

CButton Class

Other Resources

CButton Members

CButton::GetImageList

Call this method to get the image list from the button control.

```
BOOL GetImageList(
   PBUTTON_IMAGELIST pbuttonImagelist
);
```

Parameters

pbuttonImagelist

A pointer to the image list of the **CButton** object.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function emulates the functionality of the **BCM_GETIMAGELIST** message, as described in the Buttons section of the *Platform SDK*.

See Also

Reference

CButton Class

Other Resources

CButton Members

CButton::GetState

Retrieves the state of a radio button or check box.

```
UINT GetState( ) const;
```

Return Value

Specifies the current state of the button control. You can use the following masks against the return value to extract information about the state:

Mask	Meaning
0x0003	Specifies the check state (radio buttons and check boxes only). A 0 indicates the button is unchecked. A 1 indicates the button is checked. A radio button is checked when it contains a bullet (•). A check box is checked when it contains an **X**. A 2 indicates the check state is indeterminate (three-state check boxes only). The state of a three-state check box is indeterminate when it contains a halftone pattern.
0x0004	Specifies the highlight state. A nonzero value indicates that the button is highlighted. A button is highlighted when the user clicks and holds the left mouse button. The highlighting is removed when the user releases the mouse button.
0x0008	Specifies the focus state. A nonzero value indicates that the button has the focus.

Example

```
CButton myButton;

// Create a push button.
myButton.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
   CRect(10,10,100,30), pParentWnd, 1);

// Invert the highlight state of the button.
myButton.SetState( !(myButton.GetState() & 0x0004) );
```

See Also

Reference

CButton Class
Hierarchy Chart
CButton::GetCheck
CButton::SetCheck
CButton::SetState
BM_GETSTATE

Other Resources

CButton Members

CButton::GetTextMargin

Call this method to get the text margin of the **CButton** object.

```
BOOL GetTextMargin(
   RECT* pmargin
);
```

Parameters

pmargin

A pointer to the text margin of the **CButton** object.

Return Value

Returns the text margin.

Remarks

Nonzero if successful; otherwise 0.

Remarks

This member function emulates the functionality of the **BCM_GETTEXTMARGIN** message, as described in the Buttons section of the *Platform SDK*.

See Also

Reference

CButton Class

CButton::SetTextMargin

Other Resources

CButton Members

CButton::SetBitmap

Call this member function to associate a new bitmap with the button.

```
HBITMAP SetBitmap(
   HBITMAP hBitmap
);
```

Parameters

hBitmap

The handle of a bitmap.

Return Value

The handle of a bitmap previously associated with the button.

Remarks

The bitmap will be automatically placed on the face of the button, centered by default. If the bitmap is too large for the button, it will be clipped on either side. You can choose other alignment options, including the following:

- **BS_TOP**
- **BS_LEFT**
- **BS_RIGHT**
- **BS_CENTER**
- **BS_BOTTOM**
- **BS_VCENTER**

Unlike CBitmapButton, which uses four bitmaps per button, **SetBitmap** uses only one bitmap per the button. When the button is pressed, the bitmap appears to shift down and to the right.

Example

```
CButton myButton;

// Create a bitmap button.
myButton.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_BITMAP,
   CRect(10,10,60,50), pParentWnd, 1);

// Set the bitmap of the button to be the system check mark bitmap.
myButton.SetBitmap( ::LoadBitmap(NULL, MAKEINTRESOURCE(OBM_CHECK)) );
```

See Also

Reference

CButton Class
Hierarchy Chart
CButton::GetBitmap
CBitmapButton Class
CBitmapButton::LoadBitmaps
Bitmaps

Other Resources

CButton Members

CButton::SetButtonStyle

Changes the style of a button.

```
void SetButtonStyle(
   UINT nStyle,
   BOOL bRedraw = TRUE
);
```

Parameters

nStyle

Specifies the button style.

bRedraw

Specifies whether the button is to be redrawn. A nonzero value redraws the button. A 0 value does not redraw the button. The button is redrawn by default.

Remarks

Use the **GetButtonStyle** member function to retrieve the button style. The low-order word of the complete button style is the button-specific style.

Example

```
CButton myButton;

// Create an auto 3-state button.
myButton.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
   CRect(10,10,100,30), pParentWnd, 1);

// Actually, want an auto check box, change the button style to
// auto check box.
myButton.SetButtonStyle( BS_AUTOCHECKBOX );
```

See Also

Reference

CButton Class
Hierarchy Chart
CButton::GetButtonStyle
BM_SETSTYLE

Other Resources

CButton Members

CButton::SetCheck

Sets or resets the check state of a radio button or check box.

```
void SetCheck(
   int nCheck
);
```

Parameters

nCheck

Specifies the check state. This parameter can be one of the following:

Value	Meaning
BST_UNCHECKED	Set the button state to unchecked.
BST_CHECKED	Set the button state to checked.
BST_INDETERMINATE	Set the button state to indeterminate. This value can be used only if the button has the **BS_3STATE** or **BS_AUTO3STATE** style.

Remarks

This member function has no effect on a pushbutton.

Example

```
CButton myButton;

// Create an auto 3-state button.
myButton.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_AUTO3STATE,
   CRect(10,10,100,30), pParentWnd, 1);

// Set the check state to indeterminate.
myButton.SetCheck( BST_INDETERMINATE );
```

See Also

Reference

CButton Class
Hierarchy Chart
CButton::GetCheck
CButton::GetState
CButton::SetState
BM_SETCHECK

Other Resources

CButton Members

CButton::SetCursor

Call this member function to associate a new cursor with the button.

```
HCURSOR SetCursor(
   HCURSOR hCursor
);
```

Parameters

hCursor

The handle of a cursor.

Return Value

The handle of a cursor previously associated with the button.

Remarks

The cursor will be automatically placed on the face of the button, centered by default. If the cursor is too large for the button, it will be clipped on either side. You can choose other alignment options, including the following:

- **BS_TOP**
- **BS_LEFT**
- **BS_RIGHT**
- **BS_CENTER**
- **BS_BOTTOM**
- **BS_VCENTER**

Unlike CBitmapButton, which uses four bitmaps per button, **SetCursor** uses only one cursor per the button. When the button is pressed, the cursor appears to shift down and to the right.

Example

```
CButton myButton;

// Create an icon button.
myButton.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_ICON,
   CRect(10,10,60,50), pParentWnd, 1);

// Set the image of the button to be the system arrow and
// small hourglass cursor.
myButton.SetCursor( ::LoadCursor(NULL, IDC_APPSTARTING) );
```

See Also

Reference

CButton Class
Hierarchy Chart
CButton::GetCursor
CBitmapButton::LoadBitmaps
Bitmaps

Other Resources

CButton Members

CButton::SetIcon

Call this member function to associate a new icon with the button.

```
HICON SetIcon(
   HICON hIcon
);
```

Parameters

hIcon

The handle of an icon.

Return Value

The handle of an icon previously associated with the button.

Remarks

The icon will be automatically placed on the face of the button, centered by default. If the icon is too large for the button, it will be clipped on either side. You can choose other alignment options, including the following:

- **BS_TOP**
- **BS_LEFT**
- **BS_RIGHT**
- **BS_CENTER**
- **BS_BOTTOM**
- **BS_VCENTER**

Unlike CBitmapButton, which uses four bitmaps per button, **SetIcon** uses only one icon per the button. When the button is pressed, the icon appears to shift down and to the right.

Example

```
CButton myButton;

// Create an icon button.
myButton.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_ICON,
   CRect(10,10,60,50), pParentWnd, 1);

// Set the icon of the button to be the system question mark icon.
myButton.SetIcon( ::LoadIcon(NULL, IDI_QUESTION) );
```

See Also

Reference

CButton Class
Hierarchy Chart
CButton::GetIcon
CBitmapButton::LoadBitmaps
Bitmaps

Other Resources

CButton Members

CButton::SetImageList

Call this method to set the image list of the **CButton** object.

```
BOOL SetImageList(
   PBUTTON_IMAGELIST pbuttonImagelist
);
```

Parameters

pbuttonImagelist

A pointer to the new image list.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This member function emulates the functionality of the **BCM_SETIMAGELIST** message, as described in the Buttons section of the *Platform SDK*.

See Also

Reference

CButton Class

CButton::GetImageList

Other Resources

CButton Members

CButton::SetState

Sets the highlighting state of a button control.

```
void SetState(
   BOOL bHighlight
);
```

Parameters

bHighlight

Specifies whether the button is to be highlighted. A nonzero value highlights the button; a 0 value removes any highlighting.

Remarks

Highlighting affects the exterior of a button control. It has no effect on the check state of a radio button or check box.

A button control is automatically highlighted when the user clicks and holds the left mouse button. The highlighting is removed when the user releases the mouse button.

Example

```
CButton myButton;

// Create a push button.
myButton.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
   CRect(10,10,100,30), pParentWnd, 1);

// Show it as pushed.
myButton.SetState( TRUE );
```

See Also

Reference

CButton Class
Hierarchy Chart
CButton::GetState
CButton::SetCheck
CButton::GetCheck
BM_SETSTATE

Other Resources

CButton Members

CButton::SetTextMargin

Call this method to set the text margin of the **CButton** object.

```
BOOL SetTextMargin(
   RECT* pmargin
);
```

Parameters

pmargin

A pointer to the new text margin.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

This member function emulates the functionality of the **BCM_SETTEXTMARGIN** message, as described in the Buttons section of the *Platform SDK*.

See Also

Reference

CButton Class

CButton::GetTextMargin

Other Resources

CButton Members

CByteArray Class

Supports dynamic arrays of bytes.

```
class CByteArray : public CObject
```

Remarks

The member functions of **CByteArray** are similar to the member functions of class CObArray. Because of this similarity, you can use the **CObArray** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a function parameter or return value, substitute a **BYTE**.

```
CObject* CObArray::GetAt( int <nIndex> ) const;
```

for example, translates to

```
BYTE CByteArray::GetAt( int <nIndex> ) const;
```

CByteArray incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. If an array of bytes is stored to an archive, either with the overloaded insertion (**<<**) operator or with the **Serialize** member function, each element is, in turn, serialized.

Note
Before using an array, use **SetSize** to establish its size and allocate memory for it. If you do not use **SetSize**, adding elements to your array causes it to be frequently reallocated and copied. Frequent reallocation and copying are inefficient and can fragment memory.

If you need debug output from individual elements in the array, you must set the depth of the **CDumpContext** object to 1 or greater.

For more information on using **CByteArray**, see the article Collections.

Requirements

Header: afxcoll.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class
Hierarchy Chart
CObArray Class

Other Resources

CByteArray Members

CByteArray Members

Base Class Members

COBJECT Members

Construction

CByteArray	Constructs an empty array for bytes.

Bounds

GetCount	Gets the number of elements in this array.
GetSize	Gets the number of elements in this array.
GetUpperBound	Returns the largest valid index.
SetSize	Sets the number of elements to be contained in this array.

Operations

FreeExtra	Frees all unused memory above the current upper bound.
IsEmpty	Determines if the array is empty.
RemoveAll	Removes all the elements from this array.

Element Access

ElementAt	Returns a temporary reference to the byte within the array.
GetAt	Returns the value at a given index.
GetData	Allows access to elements in the array. Can be **NULL**.
SetAt	Sets the value for a given index; array not allowed to grow.

Growing the Array

Add	Adds an element to the end of the array; grows the array if necessary.
Append	Appends another array to the array; grows the array if necessary.
Copy	Copies another array to the array; grows the array if necessary.
SetAtGrow	Sets the value for a given index; grows the array if necessary.

Insertion/Removal

InsertAt	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt	Removes an element at a specific index.

Operators

operator []	Sets or gets the element at the specified index.

See Also

Reference

CByteArray Class

Hierarchy Chart

CCachedDataPathProperty Class

Implements an OLE control property transferred asynchronously and cached in a memory file.

```
class CCachedDataPathProperty : public CDataPathProperty
```

Remarks

A memory file is stored in RAM rather than on disk and is useful for fast temporary transfers.

Along with **CAysncMonikerFile** and **CDataPathProperty**, **CCachedDataPathProperty** provides functionality for the use of asynchronous monikers in OLE controls. With **CCachedDataPathProperty** objects, you are able to transfer data asynchronously from a URL or file source and store it in a memory file via the **m_Cache** public variable. All the data is stored in the memory file, and there is no need to override OnDataAvailable unless you want to watch for notifications and respond. For example, if you are transferring a large .GIF file and want to notify your control that more data has arrived and it should redraw itself, override **OnDataAvailable** to make the notification.

The class **CCachedDataPathProperty** is derived from **CDataPathProperty**.

For more information about how to use asynchronous monikers and ActiveX controls in Internet applications, see the following topics:

- Internet First Steps: ActiveX Controls
- Internet First Steps: Asynchronous Monikers

Requirements

Header: afxctl.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CDataPathProperty Class

Hierarchy Chart

CDataPathProperty Class

Other Resources

CCachedDataPathProperty Members

CCachedDataPathProperty Members

Base Class Members

CObject Members

CFile Members

COleStreamFile Members

CMonikerFile Members

CAsyncMonikerFile Members

CDataPathProperty Members

Constructors

CCachedDataPathProperty	Constructs a **CCachedDataPathProperty** object.

Data Members

m_Cache	**CMemFile** object in which to cache data.

See Also

Reference

CCachedDataPathProperty Class

Hierarchy Chart

CCachedDataPathProperty Member Functions

For information about the member functions in **CCachedDataPathProperty**, see CCachedDataPathProperty Members.

CCachedDataPathProperty::CCachedDataPathProperty

Constructs a **CCachedDataPathProperty** object.

```
CCachedDataPathProperty(
   COleControl* pControl = NULL
);
CCachedDataPathProperty(
   LPCTSTR lpszPath,
   COleControl* pControl = NULL
);
```

Parameters

pControl

A pointer to the ActiveX control object to be associated with this **CCachedDataPathProperty** object.

lpszPath

The path, which may be absolute or relative, used to create an asynchronous moniker that references the actual absolute location of the property. **CCachedDataPathProperty** uses URLs, not filenames. If you want a **CCachedDataPathProperty** object for a file, prepend file:// to the path.

Remarks

The **COleControl** object pointed to by *pControl* is used by Open and retrieved by derived classes. If *pControl* is **NULL**, the control used with **Open** should be set with SetControl. If *lpszPath* is **NULL**, you can pass in the path through **Open** or set it with SetPath.

See Also

Reference

CCachedDataPathProperty Class
Hierarchy Chart
CDataPathProperty Class

Other Resources

CCachedDataPathProperty Members

CCachedDataPathProperty Data Members

For information about the data members in **CCachedDataPathProperty**, see CCachedDataPathProperty Members.

CCachedDataPathProperty::m_Cache

Contains the class name of the memory file into which data is cached.

```
CMemFile m_Cache;
```

Remarks

A memory file is stored in RAM rather than on disk.

See Also

Reference

CCachedDataPathProperty Class

Hierarchy Chart

CDataPathProperty Class

Other Resources

CCachedDataPathProperty Members

CCheckListBox Class

Provides the functionality of a Windows checklist box.

```
class CCheckListBox : public CListBox
```

Remarks

A "checklist box" displays a list of items, such as filenames. Each item in the list has a check box next to it that the user can check or clear.

CCheckListBox is only for owner-drawn controls because the list contains more than text strings. At its simplest, a checklist box contains text strings and check boxes, but you do not need to have text at all. For example, you could have a list of small bitmaps with a check box next to each item.

To create your own checklist box, you must derive your own class from **CCheckListBox**. To derive your own class, write a constructor for the derived class, then call **Create**.

If you want to handle Windows notification messages sent by a list box to its parent (usually a class derived from CDialog), add a message-map entry and message-handler member function to the parent class for each message.

Each message-map entry takes the following form:

ON_Notification**(** *id*, *memberFxn* **)**

where *id* specifies the child window ID of the control sending the notification and *memberFxn* is the name of the parent member function you have written to handle the notification.

The parent's function prototype is as follows:

afx_msg void *memberFxn***( );**

There is only one message-map entry that pertains specifically to **CCheckListBox** (but see also the message-map entries for CListBox):

- **ON_CLBN_CHKCHANGE** The user has changed the state of an item's checkbox.

If your checklist box is a default checklist box (a list of strings with the default-sized checkboxes to the left of each), you can use the default CCheckListBox::DrawItem to draw the checklist box. Otherwise, you must override the CListBox::CompareItem function and the CCheckListBox::DrawItem and CCheckListBox::MeasureItem functions.

You can create a checklist box either from a dialog template or directly in your code.

Requirements

Header: afxwin.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

TSTCON Sample: ActiveX Control Test Container

Reference

CListBox Class

Hierarchy Chart

CListBox Class

Other Resources

CCheckListBox Members

CCheckListBox Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CListBox Members

Construction

CCheckListBox	Constructs a **CCheckListBox** object.
Create	Creates the Windows checklist box and attaches it to the **CCheckListBox** object.

Attributes

Enable	Enables or disables a checklist box item.
GetCheck	Gets the state of an item's check box.
GetCheckStyle	Gets the style of the control's check boxes.
IsEnabled	Determines whether an item is enabled.
OnGetCheckPosition	Called by the framework to get the position of an item's check box.
SetCheck	Sets the state of an item's check box.
SetCheckStyle	Sets the style of the control's check boxes.

Overridables

DrawItem	Called by the framework when a visual aspect of an owner-draw list box changes.
MeasureItem	Called by the framework when a list box with an owner-draw style is created.

See Also

Reference

CCheckListBox Class

Hierarchy Chart

CCheckListBox Member Functions

For information about the member functions in **CCheckListBox**, see CCheckListBox Members.

CCheckListBox::CCheckListBox

Constructs a **CCheckListBox** object.

```
CCheckListBox( );
```

Remarks

You construct a **CCheckListBox** object in two steps. First define a class derived from **CCheckListBox**, then call **Create**, which initializes the Windows checklist box and attaches it to the **CCheckListBox** object.

Example

```
class CMyCheckListBox : public CCheckListBox
{
   DECLARE_DYNAMIC(CMyCheckListBox)

// Constructors
public:
   CMyCheckListBox();
   BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);
...
```

See Also

Reference

CCheckListBox Class

Hierarchy Chart

CCheckListBox::Create

Other Resources

CCheckListBox Members

CCheckListBox::Create

Creates the Windows checklist box and attaches it to the **CCheckListBox** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the style of the checklist box. The style must be **LBS_HASSTRINGS** and either **LBS_OWNERDRAWFIXED** (all items in the list are the same height) or **LBS_OWNERDRAWVARIABLE** (items in the list are of varying heights). This style can be combined with other list-box styles except **LBS_USETABSTOPS**.

rect

Specifies the checklist-box size and position. Can be either a CRect object or a RECT structure.

pParentWnd

Specifies the checklist box's parent window (usually a **CDialog** object). It must not be **NULL**.

nID

Specifies the checklist box's control ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

You construct a **CCheckListBox** object in two steps. First, define a class derived from **CcheckListBox** and then call **Create**, which initializes the Windows checklist box and attaches it to the **CCheckListBox**. See CCheckListBox::CCheckListBox for a sample.

When **Create** executes, Windows sends the WM_NCCREATE, WM_CREATE, WM_NCCALCSIZE, and WM_GETMINMAXINFO messages to the checklist-box control.

These messages are handled by default by the OnNcCreate, OnCreate, OnNcCalcSize, and OnGetMinMaxInfo member functions in the **CWnd** base class. To extend the default message handling, add a message map to the your derived class and override the preceding message-handler member functions. Override **OnCreate**, for example, to perform needed initialization for a new class.

Apply the following window styles to a checklist-box control:

- **WS_CHILD** Always
- **WS_VISIBLE** Usually
- **WS_DISABLED** Rarely
- **WS_VSCROLL** To add a vertical scroll bar
- **WS_HSCROLL** To add a horizontal scroll bar
- **WS_GROUP** To group controls
- **WS_TABSTOP** To allow tabbing to this control

See Also

Reference

CCheckListBox Class

Hierarchy Chart

CCheckListBox::CCheckListBox

Other Resources

CCheckListBox Members

CCheckListBox::DrawItem

Called by the framework when a visual aspect of an owner-drawn checklist box changes.

```
virtual void DrawItem(
   LPDRAWITEMSTRUCT lpDrawItemStruct
);
```

Parameters

lpDrawItemStruct

A long pointer to a DRAWITEMSTRUCT structure that contains information about the type of drawing required.

Remarks

The **itemAction** and **itemState** members of the **DRAWITEMSTRUCT** structure define the drawing action that is to be performed.

By default, this function draws a default checkbox list, consisting of a list of strings each with a default-sized checkbox to the left. The checkbox list size is the one specified in Create.

Override this member function to implement drawing of owner-draw checklist boxes that are not the default, such as checklist boxes with lists that aren't strings, with variable-height items, or with checkboxes that aren't on the left. The application should restore all graphics device interface (GDI) objects selected for the display context supplied in *lpDrawItemStruct* before the termination of this member function.

If checklist box items are not all the same height, the checklist box style (specified in **Create**) must be **LBS_OWNERVARIABLE**, and you must override the MeasureItem function.

See Also

Reference

CCheckListBox Class
Hierarchy Chart
CCheckListBox::Create
CCheckListBox::MeasureItem

Other Resources

CCheckListBox Members

CCheckListBox::Enable

Call this function to enable or disable a checklist box item.

```
void Enable(
   int nIndex,
   BOOL bEnabled = TRUE
);
```

Parameters

nIndex

Index of the checklist box item to be enabled.

bEnabled

Specifies whether the item is enabled or disabled.

See Also

Reference

CCheckListBox Class
Hierarchy Chart
CCheckListBox::IsEnabled

Other Resources

CCheckListBox Members

CCheckListBox::GetCheck

Call this function to determine the check state of an item.

```
int GetCheck(
   int nIndex
);
```

Parameters

nIndex

Index of the item whose check status is to be retrieved.

Return Value

Zero if the item is not checked, 1 if it is checked, and 2 if it is indeterminate.

See Also

Reference

CCheckListBox Class
Hierarchy Chart
CCheckListBox::OnGetCheckPosition
CCheckListBox::SetCheck
CCheckListBox::SetCheckStyle
CCheckListBox::GetCheckStyle

Other Resources

CCheckListBox Members

CCheckListBox::GetCheckStyle

Call this function to get the checklist box's style.

```
UINT GetCheckStyle( );
```

Return Value

The style of the control's check boxes.

Remarks

For information on possible styles, see SetCheckStyle.

See Also

Reference

CCheckListBox Class
Hierarchy Chart
CCheckListBox::OnGetCheckPosition
CCheckListBox::SetCheck
CCheckListBox::SetCheckStyle
CCheckListBox::GetCheck

Other Resources

CCheckListBox Members

CCheckListBox::IsEnabled

Call this function to determine whether an item is enabled.

```
BOOL IsEnabled(
   int nIndex
);
```

Parameters

nIndex

Index of the item.

Return Value

Nonzero if the item is enabled; otherwise 0.

See Also

Reference

CCheckListBox Class

Hierarchy Chart

CCheckListBox::Enable

Other Resources

CCheckListBox Members

CCheckListBox::MeasureItem

Called by the framework when a checklist box with a nondefault style is created.

```
virtual void MeasureItem(
   LPMEASUREITEMSTRUCT lpMeasureItemStruct
);
```

Parameters

lpMeasureItemStruct

A long pointer to a MEASUREITEMSTRUCT structure.

Remarks

By default, this member function does nothing. Override this member function and fill in the **MEASUREITEMSTRUCT** structure to inform Windows of the dimensions of checklist-box items. If the checklist box is created with the LBS_OWNERDRAWVARIABLE style, the framework calls this member function for each item in the list box. Otherwise, this member is called only once.

See Also

Reference

CCheckListBox Class
Hierarchy Chart
CCheckListBox::Create
CCheckListBox::DrawItem

Other Resources

CCheckListBox Members

CCheckListBox::OnGetCheckPosition

The framework calls this function to get the position and size of the check box in an item.

```
virtual CRect OnGetCheckPosition(
   CRect rectItem,
   CRect rectCheckBox
);
```

Parameters

rectItem

The position and size of the list item.

rectCheckBox

The default position and size of an item's check box.

Return Value

The position and size of an item's check box.

Remarks

The default implementation only returns the default position and size of the check box (*rectCheckBox*). By default, a check box is aligned in the upper-left corner of an item and is the standard check box size. There may be cases where you want the check boxes on the right, or want a larger or smaller check box. In these cases, override **OnGetCheckPosition** to change the check box position and size within the item.

See Also

Reference

CCheckListBox Class
Hierarchy Chart
CCheckListBox::SetCheck
CCheckListBox::SetCheckStyle
CCheckListBox::GetCheck
CCheckListBox::GetCheckStyle

Other Resources

CCheckListBox Members

CCheckListBox::SetCheck

Call this function to set the check box of the item specified by *nIndex*.

```
void SetCheck(
   int nIndex,
   int nCheck
);
```

Parameters

nIndex

Index of the item whose check box is to be set.

nCheck

State of the check box: 0 for clear, 1 for checked, and 2 for indeterminate.

See Also

Reference

CCheckListBox Class
Hierarchy Chart
CCheckListBox::SetCheckStyle
CCheckListBox::GetCheck
CCheckListBox::GetCheckStyle

Other Resources

CCheckListBox Members

CCheckListBox::SetCheckStyle

Call this function to set the style of check boxes in the checklist box.

```
void SetCheckStyle(
   UINT nStyle
);
```

Parameters

nStyle

Determines the style of check boxes in the checklist box.

Remarks

Valid styles are:

- **BS_CHECKBOX**
- **BS_AUTOCHECKBOX**
- **BS_AUTO3STATE**
- **BS_3STATE**

For information on these styles, see Button Styles.

See Also

Reference

CCheckListBox Class
Hierarchy Chart
CCheckListBox::SetCheck
CCheckListBox::GetCheck
CCheckListBox::GetCheckStyle

Other Resources

CCheckListBox Members

CClientDC Class

Takes care of calling the Windows functions GetDC at construction time and ReleaseDC at destruction time.

```
class CClientDC : public CDC
```

Remarks

This means that the device context associated with a **CClientDC** object is the client area of a window.

For more information on **CClientDC**, see Device Contexts.

Requirements

Header: afxwin.h

See Also

Tasks

MDI Sample: Demonstrates MDI Without Using Doc/View Architecture

Reference

CDC Class

Hierarchy Chart

CDC Class

Other Resources

CClientDC Members

CClientDC Members

Base Class Members

CObject Members

CDC Members

Construction

CClientDC	Constructs a **CClientDC** object connected to the **CWnd**.

Data Members

m_hWnd	The **HWND** of the window for which this **CClientDC** is valid.

See Also

Reference

CClientDC Class

Hierarchy Chart

CClientDC Member Functions

For information about the member functions in **CClientDC**, see CClientDC Members.

CClientDC::CClientDC

Constructs a **CClientDC** object that accesses the client area of the CWnd pointed to by *pWnd*.

```
explicit CClientDC(
   CWnd* pWnd
);
```

Parameters

pWnd

The window whose client area the device context object will access.

Remarks

The constructor calls the Windows function GetDC.

An exception (of type **CResourceException**) is thrown if the Windows **GetDC** call fails. A device context may not be available if Windows has already allocated all of its available device contexts. Your application competes for the five common display contexts available at any given time under Windows.

Example

```
// Get a dc for a CWnd object pointer.
CClientDC dc(pWnd);

// Send my private message.
::SendMessage(pWnd->m_hWnd, WM_MYMESSAGE, 0, 0);
```

See Also

Reference

CClientDC Class

Hierarchy Chart

Other Resources

CClientDC Members

CClientDC Data Members

For information about the data members in **CClientDC**, see CClientDC Members.

CClientDC::m_hWnd

The **HWND** of the **CWnd** pointer used to construct the **CClientDC** object.

```
HWND m_hWnd;
```

Remarks

m_hWnd is a protected variable.

Example

See the example for CClientDC::CClientDC.

See Also

Reference

CClientDC Class

Hierarchy Chart

Other Resources

CClientDC Members

CCmdTarget Class

The base class for the Microsoft Foundation Class Library message-map architecture.

```
class CCmdTarget : public CObject
```

Remarks

A message map routes commands or messages to the member functions you write to handle them. (A command is a message from a menu item, command button, or accelerator key.)

Key framework classes derived from **CCmdTarget** include CView, CWinApp, CDocument, CWnd, and CFrameWnd. If you intend for a new class to handle messages, derive the class from one of these **CCmdTarget**-derived classes. You will rarely derive a class from **CCmdTarget** directly.

For an overview of command targets and **OnCmdMsg** routing, see Command Targets, Command Routing, and Mapping Messages.

CCmdTarget includes member functions that handle the display of an hourglass cursor. Display the hourglass cursor when you expect a command to take a noticeable time interval to execute.

Dispatch maps, similar to message maps, are used to expose OLE automation **IDispatch** functionality. By exposing this interface, other applications (such as Visual Basic) can call into your application.

Requirements

Header: afxwin.h

See Also

Tasks

ACDUAL Sample: Adds Dual Interfaces to an Automation Application

Reference

CObject Class
Hierarchy Chart
CCmdUI Class
CDocument Class
CDocTemplate Class
CWinApp Class
CWnd Class
CView Class
CFrameWnd Class
COleDispatchDriver Class

Other Resources

CCmdTarget Members

CCmdTarget Members

Base Class Members

CObject Members

Attributes

FromIDispatch	Returns a pointer to the **CCmdTarget** object associated with the **IDispatch** pointer.
GetIDispatch	Returns a pointer to the **IDispatch** object associated with the **CCmdTarget** object.
IsResultExpected	Returns nonzero if an automation function should return a value.

Constructors

CCmdTarget	Constructs a **CCmdTarget** object.

Operations

BeginWaitCursor	Displays the cursor as an hourglass cursor.
DoOleVerb	Causes an action specified by an OLE verb to be performed.
EnableAutomation	Allows OLE automation for the **CCmdTarget** object.
EnableConnections	Enables event firing over connection points.
EndWaitCursor	Returns to the previous cursor.
EnumOleVerbs	Enumerates an object's OLE verbs.
RestoreWaitCursor	Restores the hourglass cursor.

Overridables

EnableTypeLib	Enables an object's type library.
GetDispatchIID	Gets the primary dispatch interface ID.
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides.
GetTypeInfoOfGuid	Retrieves the type description that corresponds to the specified GUID.
GetTypeLib	Gets a pointer to a type library.
GetTypeLibCache	Gets the type library cache.
IsInvokeAllowed	Enables automation method invocation.
OnCmdMsg	Routes and dispatches command messages.
OnFinalRelease	Cleans up after the last OLE reference is released.

See Also

Reference

CCmdTarget Class

Hierarchy Chart

CCmdTarget Member Functions

For information about the member functions in **CCmdTarget**, see CCmdTarget Members.

CCmdTarget::BeginWaitCursor

Call this function to display the cursor as an hourglass when you expect a command to take a noticeable time interval to execute.

```
void BeginWaitCursor( );
```

Remarks

The framework calls this function to show the user that it is busy, such as when a **CDocument** object loads or saves itself to a file.

The actions of **BeginWaitCursor** are not always effective outside of a single message handler as other actions, such as **OnSetCursor** handling, could change the cursor.

Call **EndWaitCursor** to restore the previous cursor.

Example

```
// The following example illustrates the most common case
// of displaying the hourglass cursor during some lengthy
// processing of a command handler implemented in some
// CCmdTarget-derived class, such as a document or view.
void CMyView::OnSomeCommand()
{
BeginWaitCursor(); // display the hourglass cursor
// do some lengthy processing
EndWaitCursor(); // remove the hourglass cursor
}

// The next example illustrates RestoreWaitCursor.
void CMyView::OnSomeCommand()
{
BeginWaitCursor(); // display the hourglass cursor
// do some lengthy processing
// The dialog box will normally change the cursor to
// the standard arrow cursor, and leave the cursor in
// as the standard arrow cursor when the dialog box is
// closed.
CMyDialog dlg;
dlg.DoModal();

// It is necessary to call RestoreWaitCursor here in order
// to change the cursor back to the hourglass cursor.
RestoreWaitCursor();
// do some more lengthy processing
EndWaitCursor(); // remove the hourglass cursor
}

// In the above example, the dialog was clearly invoked between
// the pair of calls to BeginWaitCursor and EndWaitCursor.
// Sometimes it may not be clear whether the dialog is invoked
// in between a pair of calls to BeginWaitCursor and EndWaitCursor.
// It is permissable to call RestoreWaitCursor, even if
// BeginWaitCursor was not previously called.  This case is
// illustrated below, where CMyView::AnotherFunction does not
// need to know whether it was called in the context of an
// hourglass cursor.
void CMyView::AnotherFunction()
{
// some processing ...
CMyDialog dlg;
dlg.DoModal();
RestoreWaitCursor();
```

```
// some more processing ...
}

// If the dialog is invoked from a member function of
// some non-CCmdTarget, then you can call CWinApp::DoWaitCursor
// with a 0 parameter value to restore the hourglass cursor.
void CMyObject::AnotherFunction()
{
CMyDialog dlg;
dlg.DoModal();
AfxGetApp()->DoWaitCursor(0); // same as CCmdTarget::RestoreWaitCursor
}
```

See Also

Reference

CCmdTarget Class
Hierarchy Chart
CWaitCursor Class
CCmdTarget::EndWaitCursor
CCmdTarget::RestoreWaitCursor
CWinApp::DoWaitCursor

Other Resources

CCmdTarget Members

CCmdTarget::CCmdTarget

Constructs a **CCmdTarget** object.

```
CCmdTarget( );
```

See Also

Reference

CCmdTarget Class

Hierarchy Chart

Other Resources

CCmdTarget Members

CCmdTarget::DoOleVerb

Causes an action specified by an OLE verb to be performed.

```
BOOL DoOleVerb(
   LONG iVerb,
   LPMSG lpMsg,
   HWND hWndParent,
   LPCRECT lpRect
);
```

Parameters

iVerb

Numerical identifier of the verb.

lpMsg

Pointer to the MSG structure describing the event (such as a double-click) that invoked the verb.

hWndParent

Handle of the document window containing the object.

lpRect

Pointer to the RECT structure containing the coordinates, in pixels, that define an object's bounding rectangle in *hwndParent*.

Return Value

TRUE if successful, otherwise FALSE.

Remarks

This member function is basically an implementation of IOleObject::DoVerb. The possible actions are enumerated by CCmdTarget::EnumOleVerbs.

See Also

Reference

CCmdTarget Class

Hierarchy Chart

Other Resources

CCmdTarget Members

CCmdTarget::EnableAutomation

Call this function to enable OLE automation for an object.

```
void EnableAutomation( );
```

Remarks

This function is typically called from the constructor of your object and should only be called if a dispatch map has been declared for the class. For more information on automation see the articles Automation Clients and Automation Servers.

See Also

Reference

CCmdTarget Class
Hierarchy Chart
DECLARE_DISPATCH_MAP
DECLARE_OLECREATE

Other Resources

CCmdTarget Members

CCmdTarget::EnableConnections

Enables event firing over connection points.

```
void EnableConnections( );
```

Remarks

To enable connection points, call this member function in the constructor of your derived class.

See Also

Reference

CCmdTarget Class

Hierarchy Chart

Other Resources

CCmdTarget Members

CCmdTarget::EnableTypeLib

Enables an object's type library.

```
void EnableTypeLib( );
```

Remarks

Call this member function in the constructor of your **CCmdTarget**-derived object if it provides type information. For more information, see Knowledge Base article Q185720, "HOWTO: Provide Type Information From an MFC Automation Server." Knowledge Base articles are available in the MSDN Library Visual Studio documentation or at http://support.microsoft.com.

See Also

Reference

CCmdTarget Class

Hierarchy Chart

Other Resources

CCmdTarget Members

CCmdTarget::EndWaitCursor

Call this function after you have called the **BeginWaitCursor** member function to return from the hourglass cursor to the previous cursor.

```
void EndWaitCursor( );
```

Remarks

The framework also calls this member function after it has called the hourglass cursor.

Example

```
// The following example illustrates the most common case
// of displaying the hourglass cursor during some lengthy
// processing of a command handler implemented in some
// CCmdTarget-derived class, such as a document or view.
void CMyView::OnSomeCommand()
{
BeginWaitCursor(); // display the hourglass cursor
// do some lengthy processing
EndWaitCursor(); // remove the hourglass cursor
}

// The next example illustrates RestoreWaitCursor.
void CMyView::OnSomeCommand()
{
BeginWaitCursor(); // display the hourglass cursor
// do some lengthy processing
// The dialog box will normally change the cursor to
// the standard arrow cursor, and leave the cursor in
// as the standard arrow cursor when the dialog box is
// closed.
CMyDialog dlg;
dlg.DoModal();

// It is necessary to call RestoreWaitCursor here in order
// to change the cursor back to the hourglass cursor.
RestoreWaitCursor();
// do some more lengthy processing
EndWaitCursor(); // remove the hourglass cursor
}

// In the above example, the dialog was clearly invoked between
// the pair of calls to BeginWaitCursor and EndWaitCursor.
// Sometimes it may not be clear whether the dialog is invoked
// in between a pair of calls to BeginWaitCursor and EndWaitCursor.
// It is permissable to call RestoreWaitCursor, even if
// BeginWaitCursor was not previously called.  This case is
// illustrated below, where CMyView::AnotherFunction does not
// need to know whether it was called in the context of an
// hourglass cursor.
void CMyView::AnotherFunction()
{
// some processing ...
CMyDialog dlg;
dlg.DoModal();
RestoreWaitCursor();

// some more processing ...
}

// If the dialog is invoked from a member function of
// some non-CCmdTarget, then you can call CWinApp::DoWaitCursor
```

```
// with a 0 parameter value to restore the hourglass cursor.
void CMyObject::AnotherFunction()
{
CMyDialog dlg;
dlg.DoModal();
AfxGetApp()->DoWaitCursor(0); // same as CCmdTarget::RestoreWaitCursor
}
```

See Also

Reference

CCmdTarget Class
Hierarchy Chart
CWaitCursor Class
CCmdTarget::BeginWaitCursor
CCmdTarget::RestoreWaitCursor
CWinApp::DoWaitCursor

Other Resources

CCmdTarget Members

CCmdTarget::EnumOleVerbs

Enumerates an object's OLE verbs.

```
BOOL EnumOleVerbs(
   LPENUMOLEVERB* ppenumOleVerb
);
```

Parameters

ppenumOleVerb

A pointer to a pointer to an IEnumOLEVERB interface.

Return Value

TRUE if the object supports at least one OLE verb (in which case **ppenumOleVerb* points to an **IEnumOLEVERB** enumerator interface), otherwise FALSE.

Remarks

This member function is basically an implementation of IOleObject::EnumVerbs.

See Also

Reference

CCmdTarget Class
Hierarchy Chart
CCmdTarget::DoOleVerb

Other Resources

CCmdTarget Members

CCmdTarget::FromIDispatch

Call this function to map an **IDispatch** pointer, received from automation member functions of a class, into the **CCmdTarget** object that implements the interfaces of the **IDispatch** object.

```
static CCmdTarget* PASCAL FromIDispatch(
   LPDISPATCH lpDispatch
);
```

Parameters

lpDispatch

A pointer to an **IDispatch** object.

Return Value

A pointer to the **CCmdTarget** object associated with *lpDispatch*. This function returns **NULL** if the **IDispatch** object is not recognized as a Microsoft Foundation Class **IDispatch** object.

Remarks

The result of this function is the inverse of a call to the member function **GetIDispatch**.

See Also

Reference

CCmdTarget Class
Hierarchy Chart
CCmdTarget::GetIDispatch
COleDispatchDriver Class

Other Resources

CCmdTarget Members

CCmdTarget::GetDispatchIID

Gets the primary dispatch interface ID.

```
virtual BOOL GetDispatchIID(
   IID* pIID
);
```

Parameters

pIID

A pointer to an interface ID (a GUID).

Return Value

TRUE if successful, otherwise FALSE. If successful, **pIID* is set to the primary dispatch interface ID.

Remarks

Derived classes should override this member function (if not overridden, **GetDispatchIID** returns FALSE). See COleControl.

For more information, see Knowledge Base article Q185720, "HOWTO: Provide Type Information From an MFC Automation Server." Knowledge Base articles are available in the MSDN Library Visual Studio documentation or at http://support.microsoft.com.

See Also

Reference

CCmdTarget Class

Hierarchy Chart

Other Resources

CCmdTarget Members

CCmdTarget::GetIDispatch

Call this member function to retrieve the **IDispatch** pointer from an automation method that either returns an **IDispatch** pointer or takes an **IDispatch** pointer by reference.

```
LPDISPATCH GetIDispatch(
   BOOL bAddRef
);
```

Parameters

bAddRef

Specifies whether to increment the reference count for the object.

Return Value

The **IDispatch** pointer associated with the object.

Remarks

For objects that call **EnableAutomation** in their constructors, making them automation enabled, this function returns a pointer to the Foundation Class implementation of **IDispatch** that is used by clients who communicate via the **IDispatch** interface. Calling this function automatically adds a reference to the pointer, so it is not necessary to make a call to IUnknown::AddRef.

See Also

Reference

CCmdTarget Class
Hierarchy Chart
CCmdTarget::EnableAutomation
COleDispatchDriver Class
IUnknown::Release

Other Resources

CCmdTarget Members

CCmdTarget::GetTypeInfoCount

Retrieves the number of type information interfaces that an object provides.

```
virtual UINT GetTypeInfoCount( );
```

Return Value

The number of type information interfaces.

Remarks

This member function basically implements IDispatch::GetTypeInfoCount.

Derived classes should override this function to return the number of type information interfaces provided (either 0 or 1). If not overridden, **GetTypeInfoCount** returns 0. To override, use the IMPLEMENT_OLETYPELIB macro, which also implements **GetTypeLib** and **GetTypeLibCache**.

See Also

Reference

CCmdTarget Class
Hierarchy Chart
CCmdTarget::GetTypeLib
CCmdTarget::GetTypeLibCache

Other Resources

CCmdTarget Members

CCmdTarget::GetTypeInfoOfGuid

Retrieves the type description that corresponds to the specified GUID.

```
HRESULT GetTypeInfoOfGuid(
   LCID lcid,
   const GUID& guid,
   LPTYPEINFO* ppTypeInfo
);
```

Parameters

lcid

A locale identifier (LCID).

guid

The GUID of the type description.

ppTypeInfo

Pointer to a pointer to the ITypeInfo interface.

Return Value

An HRESULT indicating the success or failure of the call. If successful, **ppTypeInfo* points to the type information interface.

See Also

Reference

CCmdTarget Class

Hierarchy Chart

Other Resources

CCmdTarget Members

CCmdTarget::GetTypeLib

Gets a pointer to a type library.

```
virtual HRESULT GetTypeLib(
   LCID lcid,
   LPTYPELIB* ppTypeLib
);
```

Parameters

lcid

A locale identifier (LCID).

ppTypeLib

A pointer to a pointer to the ITypeLib interface.

Return Value

An HRESULT indicating the success or failure of the call. If successful, **ppTypeLib* points to the type library interface.

Remarks

Derived classes should override this member function (if not overridden, **GetTypeLib** returns TYPE_E_CANTLOADLIBRARY). Use the IMPLEMENT_OLETYPELIB macro, which also implements **GetTypeInfoCount** and **GetTypeLibCache**.

See Also

Reference

CCmdTarget Class
Hierarchy Chart
CCmdTarget::GetTypeInfoCount
CCmdTarget::GetTypeLibCache

Other Resources

CCmdTarget Members

CCmdTarget::GetTypeLibCache

Gets the type library cache.

```
virtual CTypeLibCache* GetTypeLibCache( );
```

Return Value

A pointer to a **CTypeLibCache** object.

Remarks

Derived classes should override this member function (if not overridden, **GetTypeLibCache** returns NULL). Use the IMPLEMENT_OLETYPELIB macro, which also implements **GetTypeInfoCount** and **GetTypeLib**.

See Also

Reference

CCmdTarget Class
Hierarchy Chart
CCmdTarget::GetTypeInfoCount
CCmdTarget::GetTypeLib

Other Resources

CCmdTarget Members

CCmdTarget::IsInvokeAllowed

This function is called by MFC's implementation of IDispatch::Invoke to determine if a given automation method (identified by *dispid*) can be invoked.

```
virtual BOOL IsInvokeAllowed(
   DISPID dispid
);
```

Parameters

dispid

A dispatch ID.

Return Value

TRUE if the method can be invoked, otherwise FALSE.

Remarks

If **IsInvokeAllowed** returns TRUE, **Invoke** proceeds to call the method; otherwise, **Invoke** will fail, returning E_UNEXPECTED.

Derived classes can override this function to return appropriate values (if not overridden, **IsInvokeAllowed** returns TRUE). See in particular COleControl::IsInvokeAllowed.

See Also

Reference

CCmdTarget Class
Hierarchy Chart
COleControl::IsInvokeAllowed

Other Resources

CCmdTarget Members

CCmdTarget::IsResultExpected

Use **IsResultExpected** to ascertain whether a client expects a return value from its call to an automation function.

```
BOOL IsResultExpected( );
```

Return Value

Nonzero if an automation function should return a value; otherwise 0.

Remarks

The OLE interface supplies information to MFC about whether the client is using or ignoring the result of a function call, and MFC in turn uses this information to determine the result of a call to **IsResultExpected**. If production of a return value is time- or resource-intensive, you can increase efficiency by calling this function before computing the return value.

This function returns 0 only once so that you will get valid return values from other automation functions if you call them from the automation function that the client has called.

IsResultExpected returns a nonzero value if called when an automation function call is not in progress.

See Also

Reference

CCmdTarget Class
Hierarchy Chart
CCmdTarget::GetIDispatch
CCmdTarget::EnableAutomation

Other Resources

CCmdTarget Members

CCmdTarget::OnCmdMsg

Called by the framework to route and dispatch command messages and to handle the update of command user-interface objects.

```
virtual BOOL OnCmdMsg(
   UINT nID,
   int nCode,
   void* pExtra,
   AFX_CMDHANDLERINFO* pHandlerInfo
);
```

Parameters

nID

Contains the command ID.

nCode

Identifies the command notification code. See **Remarks** for more information about values for *nCode*.

pExtra

Used according to the value of *nCode*. See **Remarks** for more information about *pExtra*.

pHandlerInfo

If not **NULL**, **OnCmdMsg** fills in the **pTarget** and **pmf** members of the *pHandlerInfo* structure instead of dispatching the command. Typically, this parameter should be **NULL**.

Return Value

Nonzero if the message is handled; otherwise 0.

Remarks

This is the main implementation routine of the framework command architecture.

At run time, **OnCmdMsg** dispatches a command to other objects or handles the command itself by calling the root class **CCmdTarget::OnCmdMsg**, which does the actual message-map lookup. For a complete description of the default command routing, see Message Handling and Mapping Topics.

On rare occasions, you may want to override this member function to extend the framework's standard command routing. Refer to Technical Note 21 for advanced details of the command-routing architecture.

If you override **OnCmdMsg**, you must supply the appropriate value for *nCode*, the command notification code, and *pExtra*, which depends on the value of *nCode*. The following table lists their corresponding values:

***nCode* value**	***pEvent* value**
CN_COMMAND	CCmdUI*
CN_EVENT	AFX_EVENT*
CN_UPDATE_COMMAND_UI	CCmdUI*
CN_OLECOMMAND	COleCmdUI*
CN_OLE_UNREGISTER	NULL

Example

```
// This example illustrates extending the framework's standard command
// route from the view to objects managed by the view.  This example
// is from an object-oriented drawing application, similar to the
```

```
// DRAWCLI sample application, which draws and edits "shapes".
BOOL CMyView::OnCmdMsg(UINT nID, int nCode, void* pExtra,
AFX_CMDHANDLERINFO* pHandlerInfo)
{
// Extend the framework's command route from the view to
// the application-specific CMyShape that is currently selected
// in the view. m_pActiveShape is NULL if no shape object
// is currently selected in the view.
if ((m_pActiveShape != NULL)
&& m_pActiveShape->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
return TRUE;
// If the object(s) in the extended command route don't handle
// the command, then let the base class OnCmdMsg handle it.
return CView::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo);
}

// The command handler for ID_SHAPE_COLOR (menu command to change
// the color of the currently selected shape) was added to the message
// map of CMyShape (note, not CMyView) using the Properties window.
// The menu item will be automatically enabled or disabled, depending
// on whether a CMyShape is currently selected in the view, that is,
// depending on whether CMyView::m_pActiveView is NULL.  It is not
// necessary to implement an ON_UPDATE_COMMAND_UI handler to enable
// or disable the menu item.
BEGIN_MESSAGE_MAP(CMyShape, CCmdTarget)
ON_COMMAND(ID_SHAPE_COLOR, OnShapeColor)
END_MESSAGE_MAP()
```

See Also

Reference

CCmdTarget Class
Hierarchy Chart
CCmdUI Class
COleCmdUI Class

Other Resources

CCmdTarget Members

CCmdTarget::OnFinalRelease

Called by the framework when the last OLE reference to or from the object is released.

```
virtual void OnFinalRelease( );
```

Remarks

Override this function to provide special handling for this situation. The default implementation deletes the object.

See Also

Reference

CCmdTarget Class
Hierarchy Chart
COleServerItem Class

Other Resources

CCmdTarget Members

CCmdTarget::RestoreWaitCursor

Call this function to restore the appropriate hourglass cursor after the system cursor has changed (for example, after a message box has opened and then closed while in the middle of a lengthy operation).

```
void RestoreWaitCursor( );
```

Example

```
// The following example illustrates the most common case
// of displaying the hourglass cursor during some lengthy
// processing of a command handler implemented in some
// CCmdTarget-derived class, such as a document or view.
void CMyView::OnSomeCommand()
{
BeginWaitCursor(); // display the hourglass cursor
// do some lengthy processing
EndWaitCursor(); // remove the hourglass cursor
}

// The next example illustrates RestoreWaitCursor.
void CMyView::OnSomeCommand()
{
BeginWaitCursor(); // display the hourglass cursor
// do some lengthy processing
// The dialog box will normally change the cursor to
// the standard arrow cursor, and leave the cursor in
// as the standard arrow cursor when the dialog box is
// closed.
CMyDialog dlg;
dlg.DoModal();

// It is necessary to call RestoreWaitCursor here in order
// to change the cursor back to the hourglass cursor.
RestoreWaitCursor();
// do some more lengthy processing
EndWaitCursor(); // remove the hourglass cursor
}

// In the above example, the dialog was clearly invoked between
// the pair of calls to BeginWaitCursor and EndWaitCursor.
// Sometimes it may not be clear whether the dialog is invoked
// in between a pair of calls to BeginWaitCursor and EndWaitCursor.
// It is permissable to call RestoreWaitCursor, even if
// BeginWaitCursor was not previously called.  This case is
// illustrated below, where CMyView::AnotherFunction does not
// need to know whether it was called in the context of an
// hourglass cursor.
void CMyView::AnotherFunction()
{
// some processing ...
CMyDialog dlg;
dlg.DoModal();
RestoreWaitCursor();

// some more processing ...
}

// If the dialog is invoked from a member function of
// some non-CCmdTarget, then you can call CWinApp::DoWaitCursor
// with a 0 parameter value to restore the hourglass cursor.
void CMyObject::AnotherFunction()
{
CMyDialog dlg;
```

```
dlg.DoModal();
AfxGetApp()->DoWaitCursor(0); // same as CCmdTarget::RestoreWaitCursor
}
```

See Also

Reference

CCmdTarget Class
Hierarchy Chart
CWaitCursor Class
CCmdTarget::EndWaitCursor
CCmdTarget::BeginWaitCursor
CWinApp::DoWaitCursor

Other Resources

CCmdTarget Members

CCmdUI Class

Is used only within an **ON_UPDATE_COMMAND_UI** handler in a **CCmdTarget**-derived class.

```
class CCmdUI
```

Remarks

CCmdUI does not have a base class.

When a user of your application pulls down a menu, each menu item needs to know whether it should be displayed as enabled or disabled. The target of a menu command provides this information by implementing an **ON_UPDATE_COMMAND_UI** handler. For each of the command user-interface objects in your application, use the Properties window to create a message-map entry and function prototype for each handler.

When the menu is pulled down, the framework searches for and calls each **ON_UPDATE_COMMAND_UI** handler, each handler calls **CCmdUI** member functions such as **Enable** and **Check**, and the framework then appropriately displays each menu item.

A menu item can be replaced with a control-bar button or other command user-interface object without changing the code within the **ON_UPDATE_COMMAND_UI** handler.

The following table summarizes the effect **CCmdUI**'s member functions have on various command user-interface items.

User-Interface Item	Enable	SetCheck	SetRadio	SetText
Menu item	Enables or disables	Checks (×) or unchecks	Checks using dot (•)	Sets item text
Toolbar button	Enables or disables	Selects, unselects, or indeterminate	Same as **SetCheck**	(Not applicable)
Status-bar pane	Makes text visible or invisible	Sets pop-out or normal border	Same as **SetCheck**	Sets pane text
Normal button in **CDialogBar**	Enables or disables	Checks or unchecks check box	Same as **SetCheck**	Sets button text
Normal control in **CDialogBar**	Enables or disables	(Not applicable)	(Not applicable)	Sets window text

For more on the use of this class, see How to Update User-Interface Objects.

Requirements

Header: afxwin.h

See Also

Tasks

MDI Sample: Demonstrates MDI Without Using Doc/View Architecture

Reference

Hierarchy Chart

CCmdTarget Class

Other Resources

CCmdUI Members

CCmdUI Members

Data Members

m_nID	The ID of the user-interface object.
m_nIndex	The index of the user-interface object.
m_pMenu	Points to the menu represented by the **CCmdUI** object.
m_pOther	Points to the window object that sent the notification.
m_pSubMenu	Points to the contained sub-menu represented by the **CCmdUI** object.

Operations

ContinueRouting	Tells the command-routing mechanism to continue routing the current message down the chain of handlers.
Enable	Enables or disables the user-interface item for this command.
SetCheck	Sets the check state of the user-interface item for this command.
SetRadio	Like the **SetCheck** member function, but operates on radio groups.
SetText	Sets the text for the user-interface item for this command.

See Also

Reference

CCmdUI Class

Hierarchy Chart

CCmdUI Member Functions

For information about the member functions in **CCmdUI**, see CCmdUI Members.

CCmdUI::ContinueRouting

Call this member function to tell the command-routing mechanism to continue routing the current message down the chain of handlers.

```
void ContinueRouting( );
```

Remarks

This is an advanced member function that should be used in conjunction with an **ON_COMMAND_EX** handler that returns **FALSE**. For more information, see Technical Note 6.

See Also

Reference

CCmdUI Class

Hierarchy Chart

Other Resources

CCmdUI Members

CCmdUI::Enable

Call this member function to enable or disable the user-interface item for this command.

```
virtual void Enable(
   BOOL bOn = TRUE
);
```

Parameters

bOn

TRUE to enable the item, **FALSE** to disable it.

Example

```
BEGIN_MESSAGE_MAP(CFileDoc, CDocument)
    // Many ON_* macros removed...
    ON_UPDATE_COMMAND_UI(ID_FILE_SAVE, OnUpdateFileSave)
END_MESSAGE_MAP()
.
.
.
void CFileDoc::OnUpdateFileSave(CCmdUI* pCmdUI)
{
    // Enable the menu item if the file has been modified.
    pCmdUI->Enable(m_fModified);
}
```

See Also

Reference

CCmdUI Class

Hierarchy Chart

CCmdUI::SetCheck

Other Resources

CCmdUI Members

CCmdUI::SetCheck

Call this member function to set the user-interface item for this command to the appropriate check state.

```
virtual void SetCheck(
   int nCheck = 1
);
```

Parameters

nCheck

Specifies the check state to set. If 0, unchecks; if 1, checks; and if 2, sets indeterminate.

Remarks

This member function works for menu items and toolbar buttons. The indeterminate state applies only to toolbar buttons.

See Also

Reference

CCmdUI Class

Hierarchy Chart

CCmdUI::SetRadio

Other Resources

CCmdUI Members

CCmdUI::SetRadio

Call this member function to set the user-interface item for this command to the appropriate check state.

```
virtual void SetRadio(
   BOOL bOn = TRUE
);
```

Parameters

bOn

TRUE to enable the item; otherwise **FALSE**.

Remarks

This member function operates like **SetCheck**, except that it operates on user-interface items acting as part of a radio group. Unchecking the other items in the group is not automatic unless the items themselves maintain the radio-group behavior.

See Also

Reference

CCmdUI Class

Hierarchy Chart

CCmdUI::SetCheck

Other Resources

CCmdUI Members

CCmdUI::SetText

Call this member function to set the text of the user-interface item for this command.

```
virtual void SetText(
   LPCTSTR lpszText
);
```

Parameters

lpszText

A pointer to a text string.

Example

```
void CMyRichEditView::OnUpdateLineNumber (CCmdUI* pCmdUI)
{
   int nLine = GetRichEditCtrl ().LineFromChar (-1) + 1;

   CString string;
   string.Format ("Line %d", nLine);
   pCmdUI->Enable (TRUE);
   pCmdUI->SetText (string);
}
```

See Also

Reference

CCmdUI Class

Hierarchy Chart

CCmdUI::Enable

Other Resources

CCmdUI Members

CCmdUI Data Members

For information about the data members in **CCmdUI**, see CCmdUI Members.

CCmdUI::m_nID

The ID of the menu item, toolbar button, or other user-interface object represented by the **CCmdUI** object.

```
UINT m_nID;
```

See Also

Reference

CCmdUI Class

Hierarchy Chart

Other Resources

CCmdUI Members

CCmdUI::m_nIndex

The index of the menu item, toolbar button, or other user-interface object represented by the **CCmdUI** object.

```
UINT m_nIndex;
```

See Also

Reference

CCmdUI Class

Hierarchy Chart

Other Resources

CCmdUI Members

CCmdUI::m_pMenu

Pointer (of **CMenu** type) to the menu represented by the **CCmdUI** object.

```
CMenu* m_pMenu;
```

Remarks

NULL if the item is not a menu.

See Also

Reference

CCmdUI Class

Hierarchy Chart

CMenu Class

Other Resources

CCmdUI Members

CCmdUI::m_pSubMenu

Pointer (of **CMenu** type) to the contained sub-menu represented by the **CCmdUI** object.

```
CMenu* m_pSubMenu;
```

Remarks

NULL if the item is not a menu. If the sub menu is a pop-up, **m_nID** contains the ID of the first item in the pop-up menu. For more information, see Technical Note 21.

See Also

Reference

CCmdUI Class

Hierarchy Chart

CMenu Class

Other Resources

CCmdUI Members

CCmdUI::m_pOther

Pointer (of type **CWnd**) to the window object, such as a tool or status bar, that sent the notification.

```
CWnd* m_pOther;
```

Remarks

NULL if the item is a menu or a non-**CWnd** object.

See Also

Reference

CCmdUI Class

Hierarchy Chart

CWnd Class

Other Resources

CCmdUI Members

CColorDialog Class

Allows you to incorporate a color-selection dialog box into your application.

```
class CColorDialog : public CCommonDialog
```

Remarks

A **CColorDialog** object is a dialog box with a list of colors that are defined for the display system. The user can select or create a particular color from the list, which is then reported back to the application when the dialog box exits.

To construct a **CColorDialog** object, use the provided constructor or derive a new class and use your own custom constructor.

Once the dialog box has been constructed, you can set or modify any values in them_cc structure to initialize the values of the dialog box's controls. The **m_cc** structure is of type CHOOSECOLOR.

After initializing the dialog box's controls, call the **DoModal** member function to display the dialog box and allow the user to select a color. **DoModal** returns the user's selection of either the dialog box's OK (**IDOK**) or Cancel (**IDCANCEL**) button.

If **DoModal** returns **IDOK**, you can use one of **CColorDialog**'s member functions to retrieve the information input by the user.

You can use the Windows CommDlgExtendedError function to determine whether an error occurred during initialization of the dialog box and to learn more about the error.

CColorDialog relies on the COMMDLG.DLL file that ships with Windows versions 3.1 and later.

To customize the dialog box, derive a class from **CColorDialog**, provide a custom dialog template, and add a message map to process the notification messages from the extended controls. Any unprocessed messages should be passed to the base class.

Customizing the hook function is not required.

Note
On some installations the **CColorDialog** object will not display with a gray background if you have used the framework to make other **CDialog** objects gray.

For more information on using **CColorDialog**, see Common Dialog Classes

Requirements

Header: afxdlgs.h

Smart Device Developer Notes

This class is only supported in Windows CE Standard SDK. It is not support in Windows Mobile Pocket PC or Smartphone platforms.

The Smart Device version of CColorDialog Class does not support the following methods:

- CColorDialog::SetCurrentColor

See Also

Tasks

MDI Sample: Demonstrates MDI Without Using Doc/View Architecture

DRAWCLI Sample: Illustrates Integrating Active Container Support with Application-Specific Features

Reference

CCommonDialog Class

Hierarchy Chart

Concepts

CColorDialog Members

CColorDialog Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CDialog Members

CCommonDialog Members

Data Members

m_cc	A structure used to customize the settings of the dialog box.

Construction

CColorDialog	Constructs a **CColorDialog** object.

Operations

DoModal	Displays a color dialog box and allows the user to make a selection.
GetColor	Returns a **COLORREF** structure containing the values of the selected color.
GetSavedCustomColors	Retrieves custom colors created by the user.
SetCurrentColor	Forces the current color selection to the specified color.

Overridables

OnColorOK	Override to validate the color entered into the dialog box.

Smart Device Developer Notes

This class is only supported in Windows CE Standard SDK. It is not support in Windows Mobile Pocket PC or Smartphone platforms.

The Smart Device version of CColorDialog Class does not support the following methods:

- CColorDialog::SetCurrentColor

See Also

Reference

CColorDialog Class

Hierarchy Chart

CColorDialog Member Functions

For information about the member functions in **CColorDialog**, see CColorDialog Members.

CColorDialog::CColorDialog

Constructs a **CColorDialog** object.

```
CColorDialog(
   COLORREF clrInit = 0,
   DWORD dwFlags = 0,
   CWnd* pParentWnd = NULL
);
```

Parameters

clrInit

The default color selection. If no value is specified, the default is RGB(0,0,0) (black).

dwFlags

A set of flags that customize the function and appearance of the dialog box. For more information, see the CHOOSECOLOR structure in the *Platform SDK*.

pParentWnd

A pointer to the dialog box's parent or owner window.

Example

```
// Show the Color dialog with all the default settings.
CColorDialog dlg;
dlg.DoModal();

// Show the fully opened Color dialog with red as the selected color.
CColorDialog dlg(RGB(255, 0, 0), CC_FULLOPEN);
dlg.DoModal();
```

See Also

Reference

CColorDialog Class

Hierarchy Chart

CDialog::DoModal

Concepts

CColorDialog Members

CColorDialog::DoModal

Call this function to display the Windows common color dialog box and allow the user to select a color.

```
virtual INT_PTR DoModal( );
```

Return Value

IDOK or **IDCANCEL**. If **IDCANCEL** is returned, call the Windows CommDlgExtendedError function to determine whether an error occurred.

IDOK and **IDCANCEL** are constants that indicate whether the user selected the OK or Cancel button.

Remarks

If you want to initialize the various color dialog-box options by setting members of the m_cc structure, you should do this before calling **DoModal** but after the dialog-box object is constructed.

After calling **DoModal**, you can call other member functions to retrieve the settings or information input by the user into the dialog box.

Example

See the example for CColorDialog::CColorDialog.

See Also

Reference

CColorDialog Class
Hierarchy Chart
CDialog::DoModal
CColorDialog::CColorDialog

Concepts

CColorDialog Members

CColorDialog::GetColor

Call this function after calling **DoModal** to retrieve the information about the color the user selected.

```
COLORREF GetColor( ) const;
```

Return Value

A COLORREF value that contains the RGB information for the color selected in the color dialog box.

Example

```
// Get the selected color from the CColorDialog.
CColorDialog dlg;
if (dlg.DoModal() == IDOK)
{
   COLORREF color = dlg.GetColor();
   TRACE("RGB value of the selected color - red = %u,
      green = %u, blue = %u\n",
      GetRValue(color), GetGValue(color), GetBValue(color));
}
```

See Also

Reference

CColorDialog Class

Hierarchy Chart

CColorDialog::SetCurrentColor

Concepts

CColorDialog Members

CColorDialog::GetSavedCustomColors

CColorDialog objects permit the user, in addition to choosing colors, to define up to 16 custom colors.

```
static COLORREF * PASCAL GetSavedCustomColors( );
```

Return Value

A pointer to an array of 16 RGB color values that stores custom colors created by the user.

Remarks

The **GetSavedCustomColors** member function provides access to these colors. These colors can be retrieved after DoModal returns **IDOK**.

Each of the 16 RGB values in the returned array is initialized to RGB(255,255,255) (white). The custom colors chosen by the user are saved only between dialog box invocations within the application. If you wish to save these colors between invocations of the application, you must save them in some other manner, such as in an initialization (.INI) file.

Example

```
// Get a pointer to an array of 16 RGB color values that stores
// custom colors created by the user from CColorDialog.
CColorDialog dlg;
if (dlg.DoModal() == IDOK)
{
   COLORREF* ccolor = dlg.GetSavedCustomColors();
   for (int i=0; i < 16; i++)
   {
      TRACE("RGB value of the selected color - red = %u,
         green = %u, blue = %u\n",
         GetRValue(ccolor[i]),
         GetGValue(ccolor[i]),
         GetBValue(ccolor[i]));
   }
}
```

See Also

Reference

CColorDialog Class
Hierarchy Chart
CColorDialog::GetColor

Concepts

CColorDialog Members

CColorDialog::OnColorOK

Override to validate the color entered into the dialog box.

```
virtual BOOL OnColorOK( );
```

Return Value

Nonzero if the dialog box should not be dismissed; otherwise 0 to accept the color that was entered.

Remarks

Override this function only if you want to provide custom validation of the color the user selects in the color dialog box.

The user can select a color by one of the following two methods:

- Clicking a color on the color palette. The selected color's RGB values are then reflected in the appropriate RGB edit boxes.
- Entering values in the RGB edit boxes

Overriding **OnColorOK** allows you to reject a color the user enters into a common color dialog box for any application-specific reason.

Normally, you do not need to use this function because the framework provides default validation of colors and displays a message box if an invalid color is entered.

You can call SetCurrentColor from within **OnColorOK** to force a color selection. Once **OnColorOK** has been fired (that is, the user clicks **OK** to accept the color change), you can call GetColor to get the RGB value of the new color.

Example

```
// Override OnColorOK to validate the color entered to the
// Red, Green, and Blue edit controls. If the color
// is BLACK (i.e. RGB(0, 0,0)), then force the current color
// selection to be the color initially selected when the
// dialog box is created. The color dialog won't close so
// user can enter a new color.

BOOL CMyColorDialog::OnColorOK( )
{
   // Value in Red edit control.
   if (GetDlgItemInt(COLOR_RED) == 0 &&
      // Value in Green edit control.
       GetDlgItemInt(COLOR_GREEN) == 0 &&
      // Value in Blue edit control.
       GetDlgItemInt(COLOR_BLUE) == 0)

   {
      AfxMessageBox("BLACK is not an acceptable color.
          Please enter a color again");

      // GetColor() returns initially selected color.
      SetCurrentColor(GetColor());

      // Won't dismiss color dialog.
      return TRUE;
   }

   // OK to dismiss color dialog.
   return FALSE;
}
```

See Also

Reference

CColorDialog Class

Hierarchy Chart

Concepts

CColorDialog Members

CColorDialog::SetCurrentColor

Call this function after calling **DoModal** to force the current color selection to the color value specified in *clr*.

```
void SetCurrentColor(
   COLORREF clr
);
```

Parameters

clr

An RGB color value.

Remarks

This function is called from within a message handler or **OnColorOK**. The dialog box will automatically update the user's selection based on the value of the *clr* parameter.

Example

See the example for CColorDialog::OnColorOK.

Smart Device Developer Notes

This method is not supported in smart device projects.

See Also

Reference

CColorDialog Class
Hierarchy Chart
CColorDialog::GetColor
CColorDialog::OnColorOK

Concepts

CColorDialog Members

CColorDialog Data Members

For information about the data members in **CColorDialog**, see CColorDialog Members.

CColorDialog::m_cc

A structure of type CHOOSECOLOR, whose members store the characteristics and values of the dialog box.

```
CHOOSECOLOR m_cc;
```

Remarks

After constructing a **CColorDialog** object, you can use **m_cc** to set various aspects of the dialog box before calling the DoModal member function.

Example

```
// The code below uses CColorDialog::m_cc data member to
// customize the settings of CColorDialog. The CColorDialog will
// be shown as full open and with red as the selected color.
CColorDialog dlg;
dlg.m_cc.Flags |= CC_FULLOPEN | CC_RGBINIT;
dlg.m_cc.rgbResult = RGB(255, 0, 0);
dlg.DoModal();
```

See Also

Reference

CColorDialog Class

Hierarchy Chart

Concepts

CColorDialog Members

CComboBox Class

Provides the functionality of a Windows combo box.

```
class CComboBox : public CWnd
```

Remarks

A combo box consists of a list box combined with either a static control or edit control. The list-box portion of the control may be displayed at all times or may only drop down when the user selects the drop-down arrow next to the control.

The currently selected item (if any) in the list box is displayed in the static or edit control. In addition, if the combo box has the drop-down list style, the user can type the initial character of one of the items in the list, and the list box, if visible, will highlight the next item with that initial character.

The following table compares the three combo-box styles.

Style	When is list box visible?	Static or edit control?
Simple	Always	Edit
Drop-down	When dropped down	Edit
Drop-down list	When dropped down	Static

You can create a **CComboBox** object from either a dialog template or directly in your code. In both cases, first call the constructor **CComboBox** to construct the **CComboBox** object; then call the Create member function to create the control and attach it to the **CComboBox** object.

If you want to handle Windows notification messages sent by a combo box to its parent (usually a class derived from **CDialog**), add a message-map entry and message-handler member function to the parent class for each message.

Each message-map entry takes the following form:

ON_Notification**(** *id*, *memberFxn* **)**

where *id* specifies the child-window ID of the combo-box control sending the notification and *memberFxn* is the name of the parent member function you have written to handle the notification.

The parent's function prototype is as follows:

afx_msg void *memberFxn***( );**

The order in which certain notifications will be sent cannot be predicted. In particular, a **CBN_SELCHANGE** notification may occur either before or after a **CBN_CLOSEUP** notification.

Potential message-map entries are the following:

- **ON_CBN_CLOSEUP** (Windows 3.1 and later.) The list box of a combo box has closed. This notification message is not sent for a combo box that has theCBS_SIMPLE style.
- **ON_CBN_DBLCLK** The user double-clicks a string in the list box of a combo box. This notification message is only sent for a combo box with the **CBS_SIMPLE** style. For a combo box with the CBS_DROPDOWN or CBS_DROPDOWNLIST style, a double-click cannot occur because a single click hides the list box.
- **ON_CBN_DROPDOWN** The list box of a combo box is about to drop down (be made visible). This notification message can occur only for a combo box with the **CBS_DROPDOWN** or **CBS_DROPDOWNLIST** style.
- **ON_CBN_EDITCHANGE** The user has taken an action that may have altered the text in the edit-control portion of a combo box. Unlike the **CBN_EDITUPDATE** message, this message is sent after Windows updates the screen. It is not sent if the combo box has the **CBS_DROPDOWNLIST** style.
- **ON_CBN_EDITUPDATE** The edit-control portion of a combo box is about to display altered text. This notification message is sent after the control has formatted the text but before it displays the text. It is not sent if the combo box has

the **CBS_DROPDOWNLIST** style.

- **ON_CBN_ERRSPACE** The combo box cannot allocate enough memory to meet a specific request.
- **ON_CBN_SELENDCANCEL** (Windows 3.1 and later.) Indicates the user's selection should be canceled. The user clicks an item and then clicks another window or control to hide the list box of a combo box. This notification message is sent before the **CBN_CLOSEUP** notification message to indicate that the user's selection should be ignored. The **CBN_SELENDCANCEL** or **CBN_SELENDOK** notification message is sent even if the **CBN_CLOSEUP** notification message is not sent (as in the case of a combo box with the **CBS_SIMPLE** style).
- **ON_CBN_SELENDOK** The user selects an item and then either presses the ENTER key or clicks the DOWN ARROW key to hide the list box of a combo box. This notification message is sent before the **CBN_CLOSEUP** message to indicate that the user's selection should be considered valid. The **CBN_SELENDCANCEL** or **CBN_SELENDOK** notification message is sent even if the **CBN_CLOSEUP** notification message is not sent (as in the case of a combo box with the **CBS_SIMPLE** style).
- **ON_CBN_KILLFOCUS** The combo box is losing the input focus.
- **ON_CBN_SELCHANGE** The selection in the list box of a combo box is about to be changed as a result of the user either clicking in the list box or changing the selection by using the arrow keys. When processing this message, the text in the edit control of the combo box can only be retrieved via **GetLBText** or another similar function. **GetWindowText** cannot be used.
- **ON_CBN_SETFOCUS** The combo box receives the input focus.

If you create a **CComboBox** object within a dialog box (through a dialog resource), the **CComboBox** object is automatically destroyed when the user closes the dialog box.

If you embed a **CComboBox** object within another window object, you do not need to destroy it. If you create the **CComboBox** object on the stack, it is destroyed automatically. If you create the **CComboBox** object on the heap by using the **new** function, you must call **delete** on the object to destroy it when the Windows combo box is destroyed.

Note If you want to handle **WM_KEYDOWN** and **WM_CHAR** messages, you have to subclass the combo box's edit and list box controls, derive classes from **CEdit** and **CListBox**, and add handlers for those messages to the derived classes. For more information, see http://support.microsoft.com/default.aspx?scid=kb;en-us;Q174667 and CWnd::SubclassWindow.

Requirements

Header: afxwin.h

See Also

Tasks

CTRLBARS Sample: Illustrates Custom Control Bars

Reference

CWnd Class
Hierarchy Chart
CWnd Class
CButton Class
CEdit Class
CListBox Class
CScrollBar Class
CStatic Class
CDialog Class

Other Resources

CComboBox Members

CComboBox Members

Base Class Members

Construction/Destruction

Initialization

General Operations

String Operations

Overridables

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Construction

CComboBox	Constructs a **CComboBox** object.

Initialization

Create	Creates the combo box and attaches it to the **CComboBox** object.
InitStorage	Preallocates blocks of memory for items and strings in the list-box portion of the combo box.

General Operations

Clear	Deletes (clears) the current selection (if any) in the edit control.
Copy	Copies the current selection (if any) onto the Clipboard in **CF_TEXT** format.
Cut	Deletes (cuts) the current selection, if any, in the edit control and copies the deleted text onto the Clipboard in **CF_TEXT** format.
GetComboBoxInfo	Retrieves information on the **CComboBox** object.
GetCount	Retrieves the number of items in the list box of a combo box.
GetCurSel	Retrieves the index of the currently selected item, if any, in the list box of a combo box.
GetDroppedControlRect	Retrieves the screen coordinates of the visible (dropped-down) list box of a drop-down combo box.
GetDroppedState	Determines whether the list box of a drop-down combo box is visible (dropped down).
GetDroppedWidth	Retrieves the minimum allowable width for the drop-down list-box portion of a combo box.
GetEditSel	Gets the starting and ending character positions of the current selection in the edit control of a combo box.
GetExtendedUI	Determines whether a combo box has the default user interface or the extended user interface.
GetHorizontalExtent	Returns the width in pixels that the list-box portion of the combo box can be scrolled horizontally.
GetItemData	Retrieves the application-supplied 32-bit value associated with the specified combo-box item.

GetItemDataPtr	Retrieves the application-supplied 32-bit value associated with the specified combo-box item as a pointer (**void***).
GetItemHeight	Retrieves the height of list items in a combo box.
GetLBText	Gets a string from the list box of a combo box.
GetLBTextLen	Gets the length of a string in the list box of a combo box.
GetLocale	Retrieves the locale identifier for a combo box.
GetTopIndex	Returns the index of the first visible item in the list-box portion of the combo box.
LimitText	Limits the length of the text that the user can enter into the edit control of a combo box.
Paste	Inserts the data from the Clipboard into the edit control at the current cursor position. Data is inserted only if the Clipboard contains data in **CF_TEXT** format.
SetCurSel	Selects a string in the list box of a combo box.
SetDroppedWidth	Sets the minimum allowable width for the drop-down list-box portion of a combo box.
SetEditSel	Selects characters in the edit control of a combo box.
SetExtendedUI	Selects either the default user interface or the extended user interface for a combo box that has the **CBS_DROPDOWN** or **CBS_DROPDOWNLIST** style.
SetHorizontalExtent	Sets the width in pixels that the list-box portion of the combo box can be scrolled horizontally.
SetItemData	Sets the 32-bit value associated with the specified item in a combo box.
SetItemDataPtr	Sets the 32-bit value associated with the specified item in a combo box to the specified pointer (**void***).
SetItemHeight	Sets the height of list items in a combo box or the height of the edit-control (or static-text) portion of a combo box.
SetLocale	Sets the locale identifier for a combo box.
SetTopIndex	Tells the list-box portion of the combo box to display the item with the specified index at the top.
ShowDropDown	Shows or hides the list box of a combo box that has the **CBS_DROPDOWN** or **CBS_DROPDOWNLIST** style.

String Operations

AddString	Adds a string to the end of the list in the list box of a combo box or at the sorted position for list boxes with the **CBS_SORT** style.
DeleteString	Deletes a string from the list box of a combo box.
Dir	Adds a list of filenames to the list box of a combo box.
FindString	Finds the first string that contains the specified prefix in the list box of a combo box.

FindStringExact	Finds the first list-box string (in a combo box) that matches the specified string.
InsertString	Inserts a string into the list box of a combo box.
ResetContent	Removes all items from the list box and edit control of a combo box.
SelectString	Searches for a string in the list box of a combo box and, if the string is found, selects the string in the list box and copies the string to the edit control.

Overridables

CompareItem	Called by the framework to determine the relative position of a new list item in a sorted owner-draw combo box.
DeleteItem	Called by the framework when a list item is deleted from an owner-draw combo box.
DrawItem	Called by the framework when a visual aspect of an owner-draw combo box changes.
MeasureItem	Called by the framework to determine combo box dimensions when an owner-draw combo box is created.

See Also

Reference

CComboBox Class

Hierarchy Chart

CComboBox Member Functions

For information about the member functions in **CComboBox**, see CComboBox Members.

CComboBox::AddString

Adds a string to the list box of a combo box.

```
int AddString(
   LPCTSTR lpszString
);
```

Parameters

lpszString

Points to the null-terminated string that is to be added.

Return Value

If the return value is greater than or equal to 0, it is the zero-based index to the string in the list box. The return value is **CB_ERR** if an error occurs; the return value is **CB_ERRSPACE** if insufficient space is available to store the new string.

Remarks

If the list box was not created with the CBS_SORT style, the string is added to the end of the list. Otherwise, the string is inserted into the list, and the list is sorted.

Note

This function is not supported by the Windows **ComboBoxEx** control. For more information on this control, see ComboBoxEx Controls in the *Platform SDK*.

To insert a string into a specific location within the list, use the InsertString member function.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Add 20 items to the combo box.
CString str;
for (int i=0;i < 20;i++)
{
   str.Format(_T("item string %d"), i);
   pmyComboBox->AddString( str );
}
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::InsertString
CComboBox::DeleteString
CB_ADDSTRING

Other Resources

CComboBox Members

CComboBox::CComboBox

Constructs a **CComboBox** object.

```
CComboBox( );
```

Example

```
// Declare a local CComboBox object.
CComboBox myComboBox;

// Declare a dynamic CComboBox object.
CComboBox* pmyComboBox = new CComboBox;
```

See Also

Reference

CComboBox Class

Hierarchy Chart

CComboBox::Create

Other Resources

CComboBox Members

CComboBox::Clear

Deletes (clears) the current selection, if any, in the edit control of the combo box.

```
void Clear( );
```

Remarks

To delete the current selection and place the deleted contents onto the Clipboard, use the Cut member function.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Delete all of the text from the combo box's edit control.
pmyComboBox->SetEditSel(0, -1);
pmyComboBox->Clear();
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::Copy
CComboBox::Cut
CComboBox::Paste
WM_CLEAR

Other Resources

CComboBox Members

CComboBox::CompareItem

Called by the framework to determine the relative position of a new item in the list-box portion of a sorted owner-draw combo box.

```
virtual int CompareItem(
   LPCOMPAREITEMSTRUCT lpCompareItemStruct
);
```

Parameters

lpCompareItemStruct

A long pointer to a COMPAREITEMSTRUCT structure.

Return Value

Indicates the relative position of the two items described in the **COMPAREITEMSTRUCT** structure. It can be any of the following values:

Value	Meaning
– 1	Item 1 sorts before item 2.
0	Item 1 and item 2 sort the same.
1	Item 1 sorts after item 2.

See CWnd::OnCompareItem for a description of **COMPAREITEMSTRUCT**.

Remarks

By default, this member function does nothing. If you create an owner-draw combo box with the **LBS_SORT** style, you must override this member function to assist the framework in sorting new items added to the list box.

Example

```
// CMyComboBox is my owner-drawn combo box derived from CComboBox. This
// example compares two items using strcmp to sort items in reverse
// alphabetical order. The combo box control was created with the
// following code:
//   pmyListBox->Create(
//      WS_CHILD|WS_VISIBLE|WS_BORDER|WS_HSCROLL|WS_VSCROLL|
//      LBS_SORT|LBS_MULTIPLESEL|LBS_OWNERDRAWVARIABLE,
//      myRect, pParentWnd, 1);
//
int CMyComboBox::CompareItem(LPCOMPAREITEMSTRUCT lpCompareItemStruct)
{
   ASSERT(lpCompareItemStruct->CtlType == ODT_COMBOBOX);
   LPCTSTR lpszText1 = (LPCTSTR) lpCompareItemStruct->itemData1;
   ASSERT(lpszText1 != NULL);
   LPCTSTR lpszText2 = (LPCTSTR) lpCompareItemStruct->itemData2;
   ASSERT(lpszText2 != NULL);

   return strcmp(lpszText2, lpszText1);
}
```

See Also

Reference

CComboBox Class
Hierarchy Chart
WM_COMPAREITEM
CComboBox::DrawItem
CComboBox::MeasureItem

CComboBox::DeleteItem

Other Resources

CComboBox Members

CComboBox::Copy

Copies the current selection, if any, in the edit control of the combo box onto the Clipboard in **CF_TEXT** format.

```
void Copy( );
```

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Copy all of the text from the combo box's edit control
// to the clipboard.
pmyComboBox->SetEditSel(0, -1);
pmyComboBox->Copy();
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::Clear
CComboBox::Cut
CComboBox::Paste
WM_COPY

Other Resources

CComboBox Members

CComboBox::Create

Creates the combo box and attaches it to the **CComboBox** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the style of the combo box. Apply any combination of combo-box styles to the box.

rect

Points to the position and size of the combo box. Can be a RECT structure or a **CRect** object.

pParentWnd

Specifies the combo box's parent window (usually a **CDialog**). It must not be **NULL**.

nID

Specifies the combo box's control ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

You construct a **CComboBox** object in two steps. First, call the constructor and then call **Create**, which creates the Windows combo box and attaches it to the **CComboBox** object.

When **Create** executes, Windows sends the WM_NCCREATE, WM_CREATE, WM_NCCALCSIZE, and WM_GETMINMAXINFO messages to the combo box.

These messages are handled by default by the OnNcCreate, OnCreate, OnNcCalcSize, and OnGetMinMaxInfo member functions in the **CWnd** base class. To extend the default message handling, derive a class from **CComboBox**, add a message map to the new class, and override the preceding message-handler member functions. Override **OnCreate**, for example, to perform needed initialization for a new class.

Apply the following window styles to a combo-box control. :

- **WS_CHILD** Always
- **WS_VISIBLE** Usually
- **WS_DISABLED** Rarely
- **WS_VSCROLL** To add vertical scrolling for the list box in the combo box
- **WS_HSCROLL** To add horizontal scrolling for the list box in the combo box
- **WS_GROUP** To group controls
- **WS_TABSTOP** To include the combo box in the tabbing order

Example

```
// pParentWnd is an external pointer to the parent window.
extern CWnd* pParentWnd;
// The pointer to my combo box.
extern CComboBox* pmyComboBox;
```

```
pmyComboBox->Create(
      WS_CHILD|WS_VISIBLE|WS_VSCROLL|CBS_DROPDOWNLIST,
      CRect(10,10,200,100), pParentWnd, 1);
```

See Also

Reference

CComboBox Class

Hierarchy Chart

CComboBox::CComboBox

Combo-Box Styles

Other Resources

CComboBox Members

CComboBox::Cut

Deletes (cuts) the current selection, if any, in the combo-box edit control and copies the deleted text onto the Clipboard in **CF_TEXT** format.

```
void Cut( );
```

Remarks

To delete the current selection without placing the deleted text onto the Clipboard, call the Clear member function.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Delete all of the text from the combo box's edit control and copy it
// to the clipboard.
pmyComboBox->SetEditSel(0, -1);
pmyComboBox->Cut();
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::Clear
CComboBox::Copy
CComboBox::Paste
WM_CUT

Other Resources

CComboBox Members

CComboBox::DeleteItem

Called by the framework when the user deletes an item from an owner-draw **CComboBox** object or destroys the combo box.

```
virtual void DeleteItem(
   LPDELETEITEMSTRUCT lpDeleteItemStruct
);
```

Parameters

lpDeleteItemStruct

A long pointer to a Windows DELETEITEMSTRUCT structure that contains information about the deleted item. See CWnd::OnDeleteItem for a description of this structure.

Remarks

The default implementation of this function does nothing. Override this function to redraw the combo box as needed.

Example

```
// CMyComboBox is my owner-drawn combo box derived from CComboBox. This
// example simply frees the item's text. The combo box control was
// created with the following code:
//   pmyListBox->Create(
//      WS_CHILD|WS_VISIBLE|WS_BORDER|WS_HSCROLL|WS_VSCROLL|
//      LBS_SORT|LBS_MULTIPLESEL|LBS_OWNERDRAWVARIABLE,
//      myRect, pParentWnd, 1);
//
void CMyComboBox::DeleteItem(LPDELETEITEMSTRUCT lpDeleteItemStruct)
{
   ASSERT(lpDeleteItemStruct->CtlType == ODT_COMBOBOX);
   LPVOID lpszText = (LPVOID) lpDeleteItemStruct->itemData;
   ASSERT(lpszText != NULL);

   free(lpszText);
}
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::CompareItem
CComboBox::DrawItem
CComboBox::MeasureItem
WM_DELETEITEM

Other Resources

CComboBox Members

CComboBox::DeleteString

Deletes the item in position *nIndex* from the combo box.

```
int DeleteString(
   UINT nIndex
);
```

Parameters

nIndex

Specifies the index to the string that is to be deleted.

Return Value

If the return value is greater than or equal to 0, then it is a count of the strings remaining in the list. The return value is **CB_ERR** if *nIndex* specifies an index greater than the number of items in the list.

Remarks

All items following *nIndex* now move down one position. For example, if a combo box contains two items, deleting the first item will cause the remaining item to now be in the first position. *nIndex*=0 for the item in the first position.

Example

```
// Pointer to my combo box
extern CComboBox* pmyComboBox;

// Delete every item from the combo box.
for (int i = pmyComboBox->GetCount()-1; i >= 0; i--)
{
   pmyComboBox->DeleteString( i );
}
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::InsertString
CComboBox::AddString
CB_DELETESTRING

Other Resources

CComboBox Members

CComboBox::Dir

Adds a list of filenames or drives to the list box of a combo box.

```
int Dir(
   UINT attr,
   LPCTSTR lpszWildCard
);
```

Parameters

attr

Can be any combination of the **enum** values described in CFile::GetStatus or any combination of the following values:

- **DDL_READWRITE** File can be read from or written to.
- **DDL_READONLY** File can be read from but not written to.
- **DDL_HIDDEN** File is hidden and does not appear in a directory listing.
- **DDL_SYSTEM** File is a system file.
- **DDL_DIRECTORY** The name specified by *lpszWildCard* specifies a directory.
- **DDL_ARCHIVE** File has been archived.
- **DDL_DRIVES** Include all drives that match the name specified by *lpszWildCard*.
- **DDL_EXCLUSIVE** Exclusive flag. If the exclusive flag is set, only files of the specified type are listed. Otherwise, files of the specified type are listed in addition to "normal" files.

lpszWildCard

Points to a file-specification string. The string can contain wildcards (for example, *.*).

Return Value

If the return value is greater than or equal to 0, it is the zero-based index of the last filename added to the list. The return value is **CB_ERR** if an error occurs; the return value is **CB_ERRSPACE** if insufficient space is available to store the new strings.

Remarks

This function is not supported by the Windows **ComboBoxEx** control. For more information on this control, see ComboBoxEx Controls in the *Platform SDK*.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Add all the files and directories in the windows directory.
TCHAR lpszWinPath[MAX_PATH], lpszOldPath[MAX_PATH];
::GetWindowsDirectory(lpszWinPath, MAX_PATH);

// Make the windows directory the current directory.
::GetCurrentDirectory(MAX_PATH, lpszOldPath);
::SetCurrentDirectory(lpszWinPath);

pmyComboBox->ResetContent();
pmyComboBox->Dir(DDL_READWRITE|DDL_DIRECTORY, _T("*.*"));

// Reset the current directory to its previous path.
::SetCurrentDirectory(lpszOldPath);
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CWnd::DlgDirList
CB_DIR
CFile::GetStatus
Other Resources
CComboBox Members

CComboBox::DrawItem

Called by the framework when a visual aspect of an owner-draw combo box changes.

```
virtual void DrawItem(
   LPDRAWITEMSTRUCT lpDrawItemStruct
);
```

Parameters

lpDrawItemStruct

A pointer to a DRAWITEMSTRUCT structure that contains information about the type of drawing required.

Remarks

The **itemAction** member of the **DRAWITEMSTRUCT** structure defines the drawing action that is to be performed. See CWnd::OnDrawItem for a description of this structure.

By default, this member function does nothing. Override this member function to implement drawing for an owner-draw **CComboBox** object. Before this member function terminates, the application should restore all graphics device interface (GDI) objects selected for the display context supplied in *lpDrawItemStruct*.

Example

```
// CMyComboBox is my owner-drawn combo box derived from CComboBox. This
// example draws an item's text centered vertically and horizontally. The
// combo box control was created with the following code:
//   pmyComboBox->Create(
//      WS_CHILD|WS_VISIBLE|WS_BORDER|WS_HSCROLL|WS_VSCROLL|
//      CBS_SORT|CBS_OWNERDRAWVARIABLE,
//      CRect(10,10,200,200), pParentWnd, 1);
//
void CMyComboBox::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
   ASSERT(lpDrawItemStruct->CtlType == ODT_COMBOBOX);
   LPCTSTR lpszText = (LPCTSTR) lpDrawItemStruct->itemData;
   ASSERT(lpszText != NULL);
   CDC dc;

   dc.Attach(lpDrawItemStruct->hDC);

   // Save these value to restore them when done drawing.
   COLORREF crOldTextColor = dc.GetTextColor();
   COLORREF crOldBkColor = dc.GetBkColor();

   // If this item is selected, set the background color
   // and the text color to appropriate values. Erase
   // the rect by filling it with the background color.
   if ((lpDrawItemStruct->itemAction | ODA_SELECT) &&
      (lpDrawItemStruct->itemState  & ODS_SELECTED))
   {
      dc.SetTextColor(::GetSysColor(COLOR_HIGHLIGHTTEXT));
      dc.SetBkColor(::GetSysColor(COLOR_HIGHLIGHT));
      dc.FillSolidRect(&lpDrawItemStruct->rcItem, ::GetSysColor(COLOR_HIGHLIGHT));
   }
   else
      dc.FillSolidRect(&lpDrawItemStruct->rcItem, crOldBkColor);

   // Draw the text.
   dc.DrawText(
      lpszText,
      strlen(lpszText),
      &lpDrawItemStruct->rcItem,
      DT_CENTER|DT_SINGLELINE|DT_VCENTER);
```

```
    // Reset the background color and the text color back to their
    // original values.
    dc.SetTextColor(crOldTextColor);
    dc.SetBkColor(crOldBkColor);

    dc.Detach();
}
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::CompareItem
WM_DRAWITEM
CComboBox::MeasureItem
CComboBox::DeleteItem

Other Resources

CComboBox Members

CComboBox::FindString

Finds, but doesn't select, the first string that contains the specified prefix in the list box of a combo box.

```
int FindString(
   int nStartAfter,
   LPCTSTR lpszString
) const;
```

Parameters

nStartAfter

Contains the zero-based index of the item before the first item to be searched. When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by *nStartAfter*. If –1, the entire list box is searched from the beginning.

lpszString

Points to the null-terminated string that contains the prefix to search for. The search is case independent, so this string can contain any combination of uppercase and lowercase letters.

Return Value

If the return value is greater than or equal to 0, it is the zero-based index of the matching item. It is **CB_ERR** if the search was unsuccessful.

Remarks

This function is not supported by the Windows **ComboBoxEx** control. For more information on this control, see ComboBoxEx Controls in the *Platform SDK*.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;
// The string to match.
extern LPCTSTR lpszmyString;

// Delete all items that begin with the specified string.
int nIndex = 0;
while ((nIndex=pmyComboBox->FindString(nIndex, lpszmyString)) != CB_ERR)
{
   pmyComboBox->DeleteString( nIndex );
}
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::SelectString
CComboBox::SetCurSel
CB_FINDSTRING

Other Resources

CComboBox Members

CComboBox::FindStringExact

Call the **FindStringExact** member function to find the first list-box string (in a combo box) that matches the string specified in *lpszFind*.

```
int FindStringExact(
   int nIndexStart,
   LPCTSTR lpszFind
) const;
```

Parameters

nIndexStart

Specifies the zero-based index of the item before the first item to be searched. When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by *nIndexStart*. If *nIndexStart* is –1, the entire list box is searched from the beginning.

lpszFind

Points to the null-terminated string to search for. This string can contain a complete filename, including the extension. The search is not case sensitive, so this string can contain any combination of uppercase and lowercase letters.

Return Value

The zero-based index of the matching item, or **CB_ERR** if the search was unsuccessful.

Remarks

If the combo box was created with an owner-draw style but without the CBS_HASSTRINGS style, **FindStringExact** attempts to match the doubleword value against the value of *lpszFind*.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;
// The string to match.
extern LPCTSTR lpszmyString;

// Delete all items that exactly match the specified string.
int nIndex = 0;
while ((nIndex=pmyComboBox->FindStringExact(nIndex, lpszmyString)) != CB_ERR)
{
   pmyComboBox->DeleteString( nIndex );
}
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::FindString
CB_FINDSTRINGEXACT

Other Resources

CComboBox Members

CComboBox::GetComboBoxInfo

Retrieves information for the **CComboBox** object.

```
BOOL GetComboBoxInfo(
   PCOMBOBOXINFO pcbi
) const;
```

Parameters

pcbi

A pointer to the COMBOBOXINFO structure.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This member function emulates the functionality of the CB_GETCOMBOBOXINFO message, as described in the *Platform SDK*.

See Also

Reference

CComboBox Class

Hierarchy Chart

Other Resources

CComboBox Members

CComboBox::GetCount

Call this member function to retrieve the number of items in the list-box portion of a combo box.

```
int GetCount( ) const;
```

Return Value

The number of items. The returned count is one greater than the index value of the last item (the index is zero-based). It is **CB_ERR** if an error occurs.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Add 10 items to the combo box.
CString str;
for (int i=0;i < 10;i++)
{
   str.Format(_T("item %d"), i);
   pmyComboBox->AddString( str );
}

// Verify the 10 items were added to the combo box.
ASSERT(pmyComboBox->GetCount() == 10);
```

See Also

Reference

CComboBox Class

Hierarchy Chart

CB_GETCOUNT

Other Resources

CComboBox Members

CComboBox::GetCurSel

Call this member function to determine which item in the combo box is selected.

```
int GetCurSel( ) const;
```

Return Value

The zero-based index of the currently selected item in the list box of a combo box, or **CB_ERR** if no item is selected.

Remarks

GetCurSel returns an index into the list.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Select the next item of the currently selected item
// in the combo box.
int nIndex = pmyComboBox->GetCurSel();
int nCount = pmyComboBox->GetCount();
if ((nIndex != CB_ERR) && (nCount > 1))
{
   if (++nIndex < nCount)
      pmyComboBox->SetCurSel(nIndex);
   else
      pmyComboBox->SetCurSel(0);
}
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::SetCurSel
CB_GETCURSEL

Other Resources

CComboBox Members

CComboBox::GetDroppedControlRect

Call the **GetDroppedControlRect** member function to retrieve the screen coordinates of the visible (dropped-down) list box of a drop-down combo box.

```
void GetDroppedControlRect(
   LPRECT lprect
) const;
```

Parameters

lprect

Points to the RECT structure that is to receive the coordinates.

Example

```
// This example move a combo box so that the upper left
// corner of the combo box is at a specific point.

// The pointer to my combo box.
extern CComboBox* pmyComboBox;
// The point to move the combo box to.
extern CPoint myPoint;

CRect r;

pmyComboBox->GetDroppedControlRect(&r);

pmyComboBox->GetParent()->ScreenToClient(&r);
r.OffsetRect(myPoint - r.TopLeft());
pmyComboBox->MoveWindow(&r);
```

See Also

Reference

CComboBox Class

Hierarchy Chart

CB_GETDROPPEDCONTROLRECT

Other Resources

CComboBox Members

CComboBox::GetDroppedState

Call the **GetDroppedState** member function to determine whether the list box of a drop-down combo box is visible (dropped down).

```
BOOL GetDroppedState( ) const;
```

Return Value

Nonzero if the list box is visible; otherwise 0.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Show the dropdown list box if it is not already dropped.
if (!pmyComboBox->GetDroppedState())
   pmyComboBox->ShowDropDown(TRUE);
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CB_SHOWDROPDOWN
CB_GETDROPPEDSTATE

Other Resources

CComboBox Members

CComboBox::GetDroppedWidth

Call this function to retrieve the minimum allowable width, in pixels, of the list box of a combo box.

```
int GetDroppedWidth( ) const;
```

Return Value

If successful, the minimum allowable width, in pixels; otherwise, **CB_ERR**.

Remarks

This function only applies to combo boxes with the CBS_DROPDOWN or CBS_DROPDOWNLIST style.

By default, the minimum allowable width of the drop-down list box is 0. The minimum allowable width can be set by calling SetDroppedWidth. When the list-box portion of the combo box is displayed, its width is the larger of the minimum allowable width or the combo box width.

Example

See the example for SetDroppedWidth.

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::SetDroppedWidth
CB_GETDROPPEDWIDTH

Other Resources

CComboBox Members

CComboBox::GetEditSel

Gets the starting and ending character positions of the current selection in the edit control of a combo box.

```
DWORD GetEditSel( ) const;
```

Return Value

A 32-bit value that contains the starting position in the low-order word and the position of the first nonselected character after the end of the selection in the high-order word. If this function is used on a combo box without an edit control, **CB_ERR** is returned.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;
DWORD dwSel;

// Set the selection to be all characters after the current selection.
if ((dwSel=pmyComboBox->GetEditSel()) != CB_ERR)
{
   pmyComboBox->SetEditSel(HIWORD(dwSel), -1);
}
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::SetEditSel
CB_GETEDITSEL

Other Resources

CComboBox Members

CComboBox::GetExtendedUI

Call the **GetExtendedUI** member function to determine whether a combo box has the default user interface or the extended user interface.

```
BOOL GetExtendedUI( ) const;
```

Return Value

Nonzero if the combo box has the extended user interface; otherwise 0.

Remarks

The extended user interface can be identified in the following ways:

- Clicking the static control displays the list box only for combo boxes with the CBS_DROPDOWNLIST style.
- Pressing the DOWN ARROW key displays the list box (F4 is disabled).

Scrolling in the static control is disabled when the item list is not visible (arrow keys are disabled).

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Use the extended UI if it is not already set.
if (!pmyComboBox->GetExtendedUI())
   pmyComboBox->SetExtendedUI(TRUE);
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::SetExtendedUI
CB_GETEXTENDEDUI

Other Resources

CComboBox Members

CComboBox::GetHorizontalExtent

Retrieves from the combo box the width in pixels by which the list-box portion of the combo box can be scrolled horizontally.

```
UINT GetHorizontalExtent( ) const;
```

Return Value

The scrollable width of the list-box portion of the combo box, in pixels.

Remarks

This is applicable only if the list-box portion of the combo box has a horizontal scroll bar.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Find the longest string in the combo box.
CString str;
CSize   sz;
int     dx=0;
CDC*    pDC = pmyComboBox->GetDC();
for (int i=0;i < pmyComboBox->GetCount();i++)
{
   pmyComboBox->GetLBText( i, str );
   sz = pDC->GetTextExtent(str);

   if (sz.cx > dx)
      dx = sz.cx;
}
pmyComboBox->ReleaseDC(pDC);

// Set the horizontal extent only if the current extent is not large enough.
if (pmyComboBox->GetHorizontalExtent() < dx)
{
   pmyComboBox->SetHorizontalExtent(dx);
   ASSERT(pmyComboBox->GetHorizontalExtent() == dx);
}
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CListBox::SetHorizontalExtent
CB_GETHORIZONTALEXTENT

Other Resources

CComboBox Members

CComboBox::GetItemData

Retrieves the application-supplied 32-bit value associated with the specified combo-box item.

```
DWORD_PTR GetItemData(
   int nIndex
) const;
```

Parameters

nIndex

Contains the zero-based index of an item in the combo box's list box.

Return Value

The 32-bit value associated with the item, or **CB_ERR** if an error occurs.

Remarks

The 32-bit value can be set with the *dwItemData* parameter of a SetItemData member function call. Use the **GetItemDataPtr** member function if the 32-bit value to be retrieved is a pointer (**void***).

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// If any item's data is equal to zero then reset it to -1.
for (int i=0;i < pmyComboBox->GetCount();i++)
{
   if (pmyComboBox->GetItemData(i) == 0)
   {
      pmyComboBox->SetItemData(i, (DWORD) -1);
   }
}
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::SetItemData
CComboBox::GetItemDataPtr
CComboBox::SetItemDataPtr
CB_GETITEMDATA

Other Resources

CComboBox Members

CComboBox::GetItemDataPtr

Retrieves the application-supplied 32-bit value associated with the specified combo-box item as a pointer (**void***).

```
void* GetItemDataPtr(
   int nIndex
) const;
```

Parameters

nIndex

Contains the zero-based index of an item in the combo box's list box.

Return Value

Retrieves a pointer, or –1 if an error occurs.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;
extern LPVOID    lpmyPtr;

// Check all the items in the combo box; if an item's
// data pointer is equal to my pointer then reset it to NULL.
for (int i=0;i < pmyComboBox->GetCount();i++)
{
   if (pmyComboBox->GetItemDataPtr(i) == lpmyPtr)
   {
      pmyComboBox->SetItemDataPtr(i, NULL);
   }
}
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::SetItemDataPtr
CComboBox::GetItemData
CComboBox::SetItemData
CB_GETITEMDATA

Other Resources

CComboBox Members

CComboBox::GetItemHeight

Call the **GetItemHeight** member function to retrieve the height of list items in a combo box.

```
int GetItemHeight(
   int nIndex
) const;
```

Parameters

nIndex

Specifies the component of the combo box whose height is to be retrieved. If the *nIndex* parameter is –1, the height of the edit-control (or static-text) portion of the combo box is retrieved. If the combo box has the CBS_OWNERDRAWVARIABLE style, *nIndex* specifies the zero-based index of the list item whose height is to be retrieved. Otherwise, *nIndex* should be set to 0.

Return Value

The height, in pixels, of the specified item in a combo box. The return value is **CB_ERR** if an error occurs.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Set the height of every item so the item
// is completely visible.
CString str;
CSize   sz;
int     dx=0;
CDC*    pDC = pmyComboBox->GetDC();
for (int i=0;i < pmyComboBox->GetCount();i++)
{
   pmyComboBox->GetLBText( i, str );
   sz = pDC->GetTextExtent(str);

   // Only want to set the item height if the current height
   // is not big enough.
   if (pmyComboBox->GetItemHeight(i) < sz.cy)
      pmyComboBox->SetItemHeight( i, sz.cy );
}
pmyComboBox->ReleaseDC(pDC);
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::SetItemHeight
WM_MEASUREITEM
CB_GETITEMHEIGHT

Other Resources

CComboBox Members

CComboBox::GetLBText

Gets a string from the list box of a combo box.

```
int GetLBText(
   int nIndex,
   LPTSTR lpszText
) const;
void GetLBText(
   int nIndex,
   CString& rString
) const;
```

Parameters

nIndex

Contains the zero-based index of the list-box string to be copied.

lpszText

Points to a buffer that is to receive the string. The buffer must have sufficient space for the string and a terminating null character.

rString

A reference to a **CString**.

Return Value

The length (in bytes) of the string, excluding the terminating null character. If *nIndex* does not specify a valid index, the return value is **CB_ERR**.

Remarks

The second form of this member function fills a **CString** object with the item's text.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Dump all of the items in the combo box.
#ifdef _DEBUG
   CString str, str2;
   int n;
   for (int i=0;i < pmyComboBox->GetCount();i++)
   {
      n = pmyComboBox->GetLBTextLen( i );
      pmyComboBox->GetLBText( i, str.GetBuffer(n) );
      str.ReleaseBuffer();

      str2.Format(_T("item %d: %s\r\n"), i, str.GetBuffer(0));
      afxDump << str2;
   }
#endif
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::GetLBTextLen
CB_GETLBTEXT

Other Resources

CComboBox Members

CComboBox::GetLBTextLen

Gets the length of a string in the list box of a combo box.

```
int GetLBTextLen(
   int nIndex
) const;
```

Parameters

nIndex

Contains the zero-based index of the list-box string.

Return Value

The length of the string in bytes, excluding the terminating null character. If *nIndex* does not specify a valid index, the return value is **CB_ERR**.

Example

See the example for CComboBox::GetLBText.

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::GetLBText
CB_GETLBTEXTLEN

Other Resources

CComboBox Members

CComboBox::GetLocale

Retrieves the locale used by the combo box.

```
LCID GetLocale( ) const;
```

Return Value

The locale identifier (LCID) value for the strings in the combo box.

Remarks

The locale is used, for example, to determine the sort order of the strings in a sorted combo box.

Example

See the example for CComboBox::SetLocale.

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::SetLocale
GetStringTypeW
GetSystemDefaultLCID
GetUserDefaultLCID

Other Resources

CComboBox Members

CComboBox::GetTopIndex

Retrieves the zero-based index of the first visible item in the list-box portion of the combo box.

```
int GetTopIndex( ) const;
```

Return Value

The zero-based index of the first visible item in the list-box portion of the combo box if successful, **CB_ERR** otherwise.

Remarks

Initially, item 0 is at the top of the list box, but if the list box is scrolled, another item may be at the top.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Want an item in the bottom half to be the first visible item.
int n = pmyComboBox->GetCount()/2;
if (pmyComboBox->GetTopIndex() < n)
{
   pmyComboBox->SetTopIndex(n);
   ASSERT(pmyComboBox->GetTopIndex() == n);
}
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::SetTopIndex
CB_GETTOPINDEX

Other Resources

CComboBox Members

CComboBox::InitStorage

Allocates memory for storing list box items in the list-box portion of the combo box.

```
int InitStorage(
   int nItems,
   UINT nBytes
);
```

Parameters

nItems

Specifies the number of items to add.

nBytes

Specifies the amount of memory, in bytes, to allocate for item strings.

Return Value

If successful, the maximum number of items that the list-box portion of the combo box can store before a memory reallocation is needed, otherwise **CB_ERRSPACE**, meaning not enough memory is available.

Remarks

Call this function before adding a large number of items to the list-box portion of the **CComboBox**.

Windows 95/98 only: The *wParam* parameter is limited to 16-bit values. This means list boxes cannot contain more than 32,767 items. Although the number of items is restricted, the total size of the items in a list box is limited only by available memory.

This function helps speed up the initialization of list boxes that have a large number of items (more than 100). It preallocates the specified amount of memory so that subsequent AddString, InsertString, and Dir functions take the shortest possible time. You can use estimates for the parameters. If you overestimate, some extra memory is allocated; if you underestimate, the normal allocation is used for items that exceed the preallocated amount.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Initialize the storage of the combo box to be 256 strings with
// about 10 characters per string, performance improvement.
int n = pmyComboBox->InitStorage(256, 10);
ASSERT(n != CB_ERRSPACE);

// Add 256 items to the combo box.
CString str;
for (int i=0;i < 256;i++)
{
   str.Format(_T("item string %d"), i);
   pmyComboBox->AddString( str );
}
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::CComboBox
CComboBox::Create
CComboBox::ResetContent
CB_INITSTORAGE

Other Resources

CComboBox Members

CComboBox::InsertString

Inserts a string into the list box of a combo box.

```
int InsertString(
   int nIndex,
   LPCTSTR lpszString
);
```

Parameters

nIndex

Contains the zero-based index to the position in the list box that will receive the string. If this parameter is –1, the string is added to the end of the list.

lpszString

Points to the null-terminated string that is to be inserted.

Return Value

The zero-based index of the position at which the string was inserted. The return value is **CB_ERR** if an error occurs. The return value is **CB_ERRSPACE** if insufficient space is available to store the new string.

Remarks

Unlike the AddString member function, the **InsertString** member function does not cause a list with the CBS_SORT style to be sorted.

Note

This function is not supported by the Windows **ComboBoxEx** control. For more information on this control, see ComboBoxEx Controls in the *Platform SDK*.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Insert items in between existing items.
CString str;
int n = pmyComboBox->GetCount();
for (int i=0;i < n;i++)
{
   str.Format(_T("item string %c"), (char)('A'+i));
   pmyComboBox->InsertString( 2*i, str );
}
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::AddString
CComboBox::DeleteString
CComboBox::ResetContent
CB_INSERTSTRING

Other Resources

CComboBox Members

CComboBox::LimitText

Limits the length in bytes of the text that the user can enter into the edit control of a combo box.

```
BOOL LimitText(
   int nMaxChars
);
```

Parameters

nMaxChars

Specifies the length (in bytes) of the text that the user can enter. If this parameter is 0, the text length is set to 65,535 bytes.

Return Value

Nonzero if successful. If called for a combo box with the style CBS_DROPDOWNLIST or for a combo box without an edit control, the return value is **CB_ERR**.

Remarks

If the combo box does not have the style CBS_AUTOHSCROLL, setting the text limit to be larger than the size of the edit control will have no effect.

LimitText only limits the text the user can enter. It has no effect on any text already in the edit control when the message is sent, nor does it affect the length of the text copied to the edit control when a string in the list box is selected.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Limit the number of characters in the combo box's edit control to
// be the maximum number visible.

// Get the text metrics for the combo box; needed for the
// average character width.
TEXTMETRIC tm;
CDC* pDC = pmyComboBox->GetDC();
pDC->GetTextMetrics(&tm);
pmyComboBox->ReleaseDC(pDC);

CRect r;
pmyComboBox->GetClientRect(&r);

pmyComboBox->LimitText(r.Width()/tm.tmAveCharWidth);
```

See Also

Reference

CComboBox Class

Hierarchy Chart

CB_LIMITTEXT

Other Resources

CComboBox Members

CComboBox::MeasureItem

Called by the framework when a combo box with an owner-draw style is created.

```
virtual void MeasureItem(
   LPMEASUREITEMSTRUCT lpMeasureItemStruct
);
```

Parameters

lpMeasureItemStruct

A long pointer to a MEASUREITEMSTRUCT structure.

Remarks

By default, this member function does nothing. Override this member function and fill in the **MEASUREITEMSTRUCT** structure to inform Windows of the dimensions of the list box in the combo box. If the combo box is created with the CBS_OWNERDRAWVARIABLE style, the framework calls this member function for each item in the list box. Otherwise, this member is called only once.

Using the **CBS_OWNERDRAWFIXED** style in an owner-draw combo box created with the SubclassDlgItem member function of **CWnd** involves further programming considerations. See the discussion in Technical Note 14.

See CWnd::OnMeasureItem for a description of the **MEASUREITEMSTRUCT** structure.

Example

```
// CMyComboBox is my owner-drawn combo box derived from CComboBox. This
// example measures an item and sets the height of the item to twice the
// vertical extent of its text. The combo box control was created with
// the following code:
//   pmyComboBox->Create(
//      WS_CHILD|WS_VISIBLE|WS_BORDER|WS_HSCROLL|WS_VSCROLL|
//      CBS_SORT|CBS_OWNERDRAWVARIABLE,
//      CRect(10,10,200,200), pParentWnd, 1);
//
void CMyComboBox::MeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct)
{
   ASSERT(lpMeasureItemStruct->CtlType == ODT_COMBOBOX);

   if (lpMeasureItemStruct->itemID != (UINT) -1)
   {
      LPCTSTR lpszText = (LPCTSTR) lpMeasureItemStruct->itemData;
      ASSERT(lpszText != NULL);
      CSize   sz;
      CDC*    pDC = GetDC();

      sz = pDC->GetTextExtent(lpszText);

      ReleaseDC(pDC);

      lpMeasureItemStruct->itemHeight = 2*sz.cy;
   }
}
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::CompareItem
CComboBox::DrawItem
WM_MEASUREITEM
CComboBox::DeleteItem

Other Resources

CComboBox::Paste

Inserts the data from the Clipboard into the edit control of the combo box at the current cursor position.

```
void Paste( );
```

Remarks

Data is inserted only if the Clipboard contains data in **CF_TEXT** format.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Replace all of the text in the combo box's edit control with the text
// in the clipboard.
pmyComboBox->SetEditSel(0, -1);
pmyComboBox->Paste();
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::Clear
CComboBox::Copy
CComboBox::Cut
WM_PASTE

Other Resources

CComboBox Members

CComboBox::ResetContent

Removes all items from the list box and edit control of a combo box.

```
void ResetContent( );
```

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Delete all the items from the combo box.
pmyComboBox->ResetContent();
ASSERT(pmyComboBox->GetCount() == 0);
```

See Also

Reference

CComboBox Class

Hierarchy Chart

CB_RESETCONTENT

Other Resources

CComboBox Members

CComboBox::SelectString

Searches for a string in the list box of a combo box, and if the string is found, selects the string in the list box and copies it to the edit control.

```
int SelectString(
   int nStartAfter,
   LPCTSTR lpszString
);
```

Parameters

nStartAfter

Contains the zero-based index of the item before the first item to be searched. When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by *nStartAfter*. If –1, the entire list box is searched from the beginning.

lpszString

Points to the null-terminated string that contains the prefix to search for. The search is case independent, so this string can contain any combination of uppercase and lowercase letters.

Return Value

The zero-based index of the selected item if the string was found. If the search was unsuccessful, the return value is **CB_ERR** and the current selection is not changed.

Remarks

A string is selected only if its initial characters (from the starting point) match the characters in the prefix string.

Note that the **SelectString** and **FindString** member functions both find a string, but the **SelectString** member function also selects the string.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;
// The string to match.
extern LPCTSTR lpszmyString;

// Select the item that begins with the specified string.
int nIndex = pmyComboBox->SelectString(0, lpszmyString);
ASSERT(nIndex != CB_ERR);
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::FindString
CB_SELECTSTRING

Other Resources

CComboBox Members

CComboBox::SetCurSel

Selects a string in the list box of a combo box.

```
int SetCurSel(
   int nSelect
);
```

Parameters

nSelect

Specifies the zero-based index of the string to select. If –1, any current selection in the list box is removed and the edit control is cleared.

Return Value

The zero-based index of the item selected if the message is successful. The return value is **CB_ERR** if *nSelect* is greater than the number of items in the list or if *nSelect* is set to –1, which clears the selection.

Remarks

If necessary, the list box scrolls the string into view (if the list box is visible). The text in the edit control of the combo box is changed to reflect the new selection. Any previous selection in the list box is removed.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Select the last item in the combo box.
int nCount = pmyComboBox->GetCount();
if (nCount > 0)
   pmyComboBox->SetCurSel(nCount-1);
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::GetCurSel
CB_SETCURSEL

Other Resources

CComboBox Members

CComboBox::SetDroppedWidth

Call this function to set the minimum allowable width, in pixels, of the list box of a combo box.

```
int SetDroppedWidth(
   UINT nWidth
);
```

Parameters

nWidth

The minimum allowable width of the list-box portion of the combo box, in pixels.

Return Value

If successful, the new width of the list box, otherwise **CB_ERR**.

Remarks

This function only applies to combo boxes with the CBS_DROPDOWN or CBS_DROPDOWNLIST style.

By default, the minimum allowable width of the drop-down list box is 0. When the list-box portion of the combo box is displayed, its width is the larger of the minimum allowable width or the combo box width.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Find the longest string in the combo box.
CString      str;
CSize      sz;
int      dx = 0;
TEXTMETRIC   tm;
CDC*      pDC = pmyComboBox->GetDC();
CFont*      pFont = pmyComboBox->GetFont();

// Select the listbox font, save the old font
CFont* pOldFont = pDC->SelectObject(pFont);
// Get the text metrics for avg char width
pDC->GetTextMetrics(&tm);

for (int i = 0; i < pmyComboBox->GetCount(); i++)
{
   pmyComboBox->GetLBText(i, str);
   sz = pDC->GetTextExtent(str);

   // Add the avg width to prevent clipping
   sz.cx += tm.tmAveCharWidth;

   if (sz.cx > dx)
      dx = sz.cx;
}
// Select the old font back into the DC
pDC->SelectObject(pOldFont);
pmyComboBox->ReleaseDC(pDC);

// Adjust the width for the vertical scroll bar and the left and right border.
dx += ::GetSystemMetrics(SM_CXVSCROLL) + 2*::GetSystemMetrics(SM_CXEDGE);

// If the width of the list box is too small, adjust it so that every
// item is completely visible.
if (pmyComboBox->GetDroppedWidth() < dx)
{
   pmyComboBox->SetDroppedWidth(dx);
   ASSERT(pmyComboBox->GetDroppedWidth() == dx);
```

```
}
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::GetDroppedWidth
CB_SETDROPPEDWIDTH

Other Resources

CComboBox Members

CComboBox::SetEditSel

Selects characters in the edit control of a combo box.

```
BOOL SetEditSel(
   int nStartChar,
   int nEndChar
);
```

Parameters

nStartChar

Specifies the starting position. If the starting position is set to –1, then any existing selection is removed.

nEndChar

Specifies the ending position. If the ending position is set to –1, then all text from the starting position to the last character in the edit control is selected.

Return Value

Nonzero if the member function is successful; otherwise 0. It is **CB_ERR** if **CComboBox** has the CBS_DROPDOWNLIST style or does not have a list box.

Remarks

The positions are zero-based. To select the first character of the edit control, you specify a starting position of 0. The ending position is for the character just after the last character to select. For example, to select the first four characters of the edit control, you would use a starting position of 0 and an ending position of 4.

Note
This function is not supported by the Windows **ComboBoxEx** control. For more information on this control, see ComboBoxEx Controls in the *Platform SDK*.

Example

See the example for CComboBox::GetEditSel.

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::GetEditSel
CB_SETEDITSEL

Other Resources

CComboBox Members

CComboBox::SetExtendedUI

Call the **SetExtendedUI** member function to select either the default user interface or the extended user interface for a combo box that has the CBS_DROPDOWN or CBS_DROPDOWNLIST style.

```
int SetExtendedUI(
   BOOL bExtended = TRUE
);
```

Parameters

bExtended

Specifies whether the combo box should use the extended user interface or the default user interface. A value of **TRUE** selects the extended user interface; a value of **FALSE** selects the standard user interface.

Return Value

CB_OKAY if the operation is successful, or **CB_ERR** if an error occurs.

Remarks

The extended user interface can be identified in the following ways:

- Clicking the static control displays the list box only for combo boxes with the **CBS_DROPDOWNLIST** style.
- Pressing the DOWN ARROW key displays the list box (F4 is disabled).

Scrolling in the static control is disabled when the item list is not visible (the arrow keys are disabled).

Example

See the example for CComboBox::GetExtendedUI.

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::GetExtendedUI
CB_SETEXTENDEDUI

Other Resources

CComboBox Members

CComboBox::SetHorizontalExtent

Sets the width, in pixels, by which the list-box portion of the combo box can be scrolled horizontally.

```
void SetHorizontalExtent(
   UINT nExtent
);
```

Parameters

nExtent

Specifies the number of pixels by which the list-box portion of the combo box can be scrolled horizontally.

Remarks

If the width of the list box is smaller than this value, the horizontal scroll bar will horizontally scroll items in the list box. If the width of the list box is equal to or greater than this value, the horizontal scroll bar is hidden or, if the combo box has the CBS_DISABLENOSCROLL style, disabled.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Find the longest string in the combo box.
CString      str;
CSize      sz;
int      dx = 0;
TEXTMETRIC   tm;
CDC*      pDC = pmyComboBox->GetDC();
CFont*      pFont = pmyComboBox->GetFont();

// Select the listbox font, save the old font
CFont* pOldFont = pDC->SelectObject(pFont);
// Get the text metrics for avg char width
pDC->GetTextMetrics(&tm);

for (int i = 0; i < pmyComboBox->GetCount(); i++)
{
   pmyComboBox->GetLBText(i, str);
   sz = pDC->GetTextExtent(str);

   // Add the avg width to prevent clipping
   sz.cx += tm.tmAveCharWidth;

   if (sz.cx > dx)
      dx = sz.cx;
}
// Select the old font back into the DC
pDC->SelectObject(pOldFont);
pmyComboBox->ReleaseDC(pDC);

// Set the horizontal extent so every character of all strings can
// be scrolled to.
pmyComboBox->SetHorizontalExtent(dx);
```

See Also

Reference

CComboBox Class

Hierarchy Chart

CComboBox::GetHorizontalExtent

CB_SETHORIZONTALEXTENT

Other Resources

CComboBox Members

CComboBox::SetItemData

Sets the 32-bit value associated with the specified item in a combo box.

```
int SetItemData(
   int nIndex,
   DWORD_PTR dwItemData
);
```

Parameters

nIndex

Contains a zero-based index to the item to set.

dwItemData

Contains the new value to associate with the item.

Return Value

CB_ERR if an error occurs.

Remarks

Use the **SetItemDataPtr** member function if the 32-bit item is to be a pointer.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Set the data of each item to be equal to its index.
for (int i=0;i < pmyComboBox->GetCount();i++)
{
   pmyComboBox->SetItemData(i, i);
}
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::GetItemData
CComboBox::GetItemDataPtr
CComboBox::SetItemDataPtr
CB_SETITEMDATA
CComboBox::AddString
CComboBox::InsertString

Other Resources

CComboBox Members

CComboBox::SetItemDataPtr

Sets the 32-bit value associated with the specified item in a combo box to be the specified pointer (**void***).

```
int SetItemDataPtr(
   int nIndex,
   void* pData
);
```

Parameters

nIndex

Contains a zero-based index to the item.

pData

Contains the pointer to associate with the item.

Return Value

CB_ERR if an error occurs.

Remarks

This pointer remains valid for the life of the combo box, even though the item's relative position within the combo box might change as items are added or removed. Hence, the item's index within the box can change, but the pointer remains reliable.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Set the data pointer of each item to be NULL.
for (int i=0;i < pmyComboBox->GetCount();i++)
{
   pmyComboBox->SetItemDataPtr(i, NULL);
}
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::GetItemData
CComboBox::GetItemDataPtr
CComboBox::SetItemData
CB_SETITEMDATA
CComboBox::AddString
CComboBox::InsertString

Other Resources

CComboBox Members

CComboBox::SetItemHeight

Call the **SetItemHeight** member function to set the height of list items in a combo box or the height of the edit-control (or static-text) portion of a combo box.

```
int SetItemHeight(
   int nIndex,
   UINT cyItemHeight
);
```

Parameters

nIndex

Specifies whether the height of list items or the height of the edit-control (or static-text) portion of the combo box is set.

If the combo box has the CBS_OWNERDRAWVARIABLE style, *nIndex* specifies the zero-based index of the list item whose height is to be set; otherwise, *nIndex* must be 0 and the height of all list items will be set.

If *nIndex* is –1, the height of the edit-control or static-text portion of the combo box is to be set.

cyItemHeight

Specifies the height, in pixels, of the combo-box component identified by *nIndex*.

Return Value

CB_ERR if the index or height is invalid; otherwise 0.

Remarks

The height of the edit-control (or static-text) portion of the combo box is set independently of the height of the list items. An application must ensure that the height of the edit-control (or static-text) portion is not smaller than the height of a particular list-box item.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Set the height of every item to be the
// vertical size of the item's text extent.
CString str;
CSize   sz;
int     dx=0;
CDC*    pDC = pmyComboBox->GetDC();
for (int i=0;i < pmyComboBox->GetCount();i++)
{
   pmyComboBox->GetLBText( i, str );
   sz = pDC->GetTextExtent(str);

   pmyComboBox->SetItemHeight( i, sz.cy );
}
pmyComboBox->ReleaseDC(pDC);
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::GetItemHeight
WM_MEASUREITEM
CB_SETITEMHEIGHT

Other Resources

CComboBox Members

CComboBox::SetLocale

Sets the locale identifier for this combo box.

```
LCID SetLocale(
   LCID nNewLocale
);
```

Parameters

nNewLocale

The new locale identifier (LCID) value to set for the combo box.

Return Value

The previous locale identifier (LCID) value for this combo box.

Remarks

If **SetLocale** is not called, the default locale is obtained from the system. This system default locale can be modified by using Control Panel's Regional (or International) application.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;
// My LCID to use.
LCID mylcid = MAKELCID(MAKELANGID(LANG_SPANISH, SUBLANG_SPANISH_MEXICAN),
   SORT_DEFAULT);

// Force the list box to use my locale.
pmyComboBox->SetLocale(mylcid);
ASSERT(pmyComboBox->GetLocale() == mylcid);
```

See Also

Reference

CComboBox Class

Hierarchy Chart

CComboBox::GetLocale

Other Resources

CComboBox Members

CComboBox::SetTopIndex

Ensures that a particular item is visible in the list-box portion of the combo box.

```
int SetTopIndex(
   int nIndex
);
```

Parameters

nIndex

Specifies the zero-based index of the list-box item.

Return Value

Zero if successful, or **CB_ERR** if an error occurs.

Remarks

The system scrolls the list box until either the item specified by *nIndex* appears at the top of the list box or the maximum scroll range has been reached.

Example

```
// The pointer to my combo box.
extern CComboBox* pmyComboBox;

// Set the first visible item in the combo box to be the middle item
pmyComboBox->SetTopIndex(pmyComboBox->GetCount()/2);
```

See Also

Reference

CComboBox Class
Hierarchy Chart
CComboBox::GetTopIndex
CB_SETTOPINDEX

Other Resources

CComboBox Members

CComboBox::ShowDropDown

Shows or hides the list box of a combo box that has the CBS_DROPDOWN or CBS_DROPDOWNLIST style.

```
void ShowDropDown(
   BOOL bShowIt = TRUE
);
```

Parameters

bShowIt

Specifies whether the drop-down list box is to be shown or hidden. A value of **TRUE** shows the list box. A value of **FALSE** hides the list box.

Remarks

By default, a combo box of this style will show the list box.

This member function has no effect on a combo box created with the CBS_SIMPLE style.

Example

See the example for CComboBox::GetDroppedState.

See Also

Reference

CComboBox Class
Hierarchy Chart
CB_SHOWDROPDOWN

Other Resources

CComboBox Members

CComboBoxEx Class

Extends the combo box control by providing support for image lists.

```
class CComboBoxEx : public CComboBox
```

Remarks

By using **CComboBoxEx** to create combo box controls, you no longer need to implement your own image drawing code. Instead, use **CComboBoxEx** to access images from an image list.

Image List Support

In a standard combo box, the owner of the combo box is responsible for drawing an image by creating the combo box as an owner-draw control. When you use **CComboBoxEx**, you do not need to set the drawing styles **CBS_OWNERDRAWFIXED** and **CBS_HASSTRINGS** because they are implied. Otherwise, you must write code to perform drawing operations. A **CComboBoxEx** control supports up to three images per item: one for a selected state, one for an unselected state, and one for an overlay image.

Styles

CComboBoxEx supports the styles **CBS_SIMPLE**, **CBS_DROPDOWN**, **CBS_DROPDOWNLIST**, and **WS_CHILD**. All other styles passed when you create the window are ignored by the control. After the window is created, you can provide other combo box styles by calling the **CComboBoxEx** member function SetExtendedStyle. With these styles, you can:

- Set string searches in the list to be case-sensitive.
- Create a combo box control that uses the slash ('/'), backslash ('\'), and period ('.') characters as word delimiters. This allow users to jump from word to word, using the keyboard shortcut CTRL+ ARROW.
- Set the combo box control to either display or not display an image. If no image is displayed, the combo box can remove the text indent that accommodates an image.
- Create a narrow combo box control, including sizing it so it clips the wider combo box it contains.

These style flags are described further in Using CComboBoxEx.

Item Retention and Callback Item Attributes

Item information, such as indexes for items and images, indentation values, and text strings, is stored in the Win32 structure COMBOBOXEXITEM, as described in the *Platform SDK*. The structure also contains members that correspond to callback flags.

For a detailed, conceptual discussion, see Using CComboBoxEx.

Requirements

Header: afxcmn.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

MFCIE Sample: Demonstrates the MFC CHtmlView and CReBar Classes

Reference

CComboBox Class

Hierarchy Chart

CComboBox Class

Other Resources

CComboBoxEx Members

CComboBoxEx Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CComboBox Members

Constructors

CComboBoxEx	Constructs a **CComboBoxEx** object.

Initialization

Create	Creates the combo box and attaches it to the **CComboBoxEx** object.
CreateEx	Creates a combo box with the specified Windows extended styles and attaches it to a **ComboBoxEx** object.

Operators

DeleteItem	Removes an item from a **ComboBoxEx** control.
GetItem	Retrieves item information for a given **ComboBoxEx** item.
InsertItem	Inserts a new item in a **ComboBoxEx** control.
SetItem	Sets the attributes for an item in a **ComboBoxEx** control.

Attributes

GetComboBoxCtrl	Retrieves a pointer to the child combo box control.
GetEditCtrl	Retrieves the handle to the edit control portion of a **ComboBoxEx** control.
GetExtendedStyle	Retrieves the extended styles that are in use for a **ComboBoxEx** control.
GetImageList	Retrieves a pointer to the image list assigned to a **ComboBoxEx** control.
HasEditChanged	Determines if the user has changed the contents of the **ComboBoxEx** edit control by typing.
SetExtendedStyle	Sets extended styles within a **ComboBoxEx** control.
SetImageList	Sets an image list for a **ComboBoxEx** control.
SetWindowTheme	Sets the visual style of the extended combo box control.

See Also

Reference

CComboBoxEx Class

Hierarchy Chart

CComboBoxEx Member Functions

For information about the member functions in **CComboBoxEx**, see CComboBoxEx Members.

CComboBoxEx::CComboBoxEx

Call this member function to create a **CComboBoxEx** object.

```
CComboBoxEx( );
```

See Also

Reference

CComboBoxEx Class
Hierarchy Chart
CComboBoxEx::Create
CComboBoxEx::CreateEx
CComboBox Class

Other Resources

CComboBoxEx Members

CComboBoxEx::Create

Creates the combo box and attaches it to the **CComboBoxEx** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the combination of combo box styles applied to the combo box. See **Remarks** below for more information about styles.

rect

A reference to a CRect object or RECT structure, which is the position and size of the combo box.

pParentWnd

A pointer to a CWnd object that is the parent window of the combo box (usually a **CDialog**). It must not be **NULL**.

nID

Specifies the combo box's control ID.

Return Value

Nonzero if the object was created successfully; otherwise 0.

Remarks

Create a **CComboBoxEx** object in two steps:

1. Call CComboBoxEx to construct a **CComboBoxEx** object.
2. Call this member function, which creates the extended Windows combo box and attaches it to the **CComboBoxEx** object.

When you call **Create**, MFC initializes the common controls.

When you create the combo box, you can specify any or all of the following combo-box styles:

- **CBS_SIMPLE**
- **CBS_DROPDOWN**
- **CBS_DROPDOWNLIST**
- **CBS_AUTOHSCROLL**
- **WS_CHILD**

All other styles passed when you create the window are ignored. The **ComboBoxEx** control also supports extended styles that provide additional features. These styles are described in ComboBoxEx control extended styles, in the *Platform SDK*. Set these styles by calling SetExtendedStyle.

If you want to use extended windows styles with your control, call CreateEx instead of **Create**.

See Also

Reference

CComboBoxEx Class
Hierarchy Chart
CComboBoxEx::CComboBoxEx

Other Resources

CComboBoxEx Members

CComboBoxEx::CreateEx

Call this function to create an extended combo box control (a child window) and associate it with the **CComboBoxEx** object.

```
virtual BOOL CreateEx(
   DWORD dwExStyle,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwExStyle

Specifies the extended style of the control being created. For a list of extended Windows styles, see the *dwExStyle* parameter for CreateWindowEx in the *Platform SDK*.

dwStyle

The combo box control's style. See Create for a list of styles.

rect

A reference to a RECT structure describing the size and position of the window to be created, in client coordinates of *pParentWnd*.

pParentWnd

A pointer to the window that is the control's parent.

nID

The control's child-window ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Use **CreateEx** instead of **Create** to apply extended Windows styles, specified by the Windows extended style preface **WS_EX_**.

CreateEx creates the control with the extended Windows styles specified by *dwExStyle*. You must set extended styles specific to an extended combo box control using SetExtendedStyle. For example, use **CreateEx** to set such styles as **WS_EX_CONTEXTHELP**, but use **SetExtendedStyle** to set such styles as **CBES_EX_CASESENSITIVE**. For more information, see the styles described in the topic ComboBoxEx Control Extended Styles in the *Platform SDK*.

See Also

Reference

CComboBoxEx Class
Hierarchy Chart
CComboBoxEx::CComboBoxEx
CComboBox Class

Other Resources

CComboBoxEx Members

CComboBoxEx::DeleteItem

Removes an item from a **ComboBoxEx** control.

```
int DeleteItem(
   int iIndex
);
```

Parameters

iIndex

Zero-based index of the item to be removed.

Return Value

The number of items remaining in the control. If *iIndex* is invalid, the function returns **CB_ERR**.

Remarks

This member function implements the functionality of the message CBEM_DELETEITEM, as described in the *Platform SDK*. When you call DeleteItem, a WM_NOTIFY message with **CBEN_DELETEITEM** notification will be sent to the parent window.

See Also

Reference

CComboBoxEx Class

Hierarchy Chart

Other Resources

CComboBoxEx Members

CComboBoxEx::GetComboBoxCtrl

Call this member function to get a pointer to a combo box control within a **CComboBoxEx** object.

```
CComboBox* GetComboBoxCtrl( );
```

Return Value

A pointer to a **CComboBox** object.

Remarks

The **CComboBoxEx** control consists of a parent window, which encapsulates a **CComboBox**.

The **CComboBox** object pointed to by the return value is a temporary object and is destroyed during the next idle processing time.

See Also

Reference

CComboBoxEx Class
Hierarchy Chart
CComboBoxEx::GetEditCtrl

Other Resources

CComboBoxEx Members

CComboBoxEx::GetEditCtrl

Call this member function to get a pointer to the edit control for a combo box.

```
CEdit* GetEditCtrl( );
```

Return Value

A pointer to a CEdit object.

Remarks

A **CComboBoxEx** control uses an edit box when it is created with the **CBS_DROPDOWN** style.

The **CEdit** object pointed to by the return value is a temporary object and is destroyed during the next idle processing time.

See Also

Reference

CComboBoxEx Class
Hierarchy Chart
CComboBoxEx::Create
CComboBoxEx::CreateEx

Other Resources

CComboBoxEx Members

CComboBoxEx::GetExtendedStyle

Call this member function to get the extended styles used for a **CComboBoxEx** control.

```
DWORD GetExtendedStyle( ) const;
```

Return Value

The **DWORD** value that contains the extended styles that are used for the combo box control.

Remarks

See ComboBoxEx Control Extended Styles in the *Platform SDK* for more information about these styles.

See Also

Reference

CComboBoxEx Class
Hierarchy Chart
CComboBoxEx::SetExtendedStyle

Other Resources

CComboBoxEx Members

CComboBoxEx::GetImageList

Call this member function to get a pointer to the image list used by a **CComboBoxEx** control.

```
CImageList* GetImageList( ) const;
```

Return Value

A pointer to a CImageList object. If it fails, this member function returns **NULL**.

Remarks

The **CImageList** object pointed to by the return value is a temporary object and is destroyed during the next idle processing time.

See Also

Reference

CComboBoxEx Class
Hierarchy Chart
CComboBoxEx::SetImageList
COMBOBOXEXITEM

Other Resources

CComboBoxEx Members

CComboBoxEx::GetItem

Retrieves item information for a given **ComboBoxEx** item.

```
BOOL GetItem(
   COMBOBOXEXITEM* pCBItem
);
```

Parameters

pCBItem

A pointer to a COMBOBOXEXITEM structure that will receive the item information.

Return Value

Nonzero if the operation was successful; otherwise 0.

Remarks

This member function implements the functionality of the message CBEM_GETITEM, as described in the *Platform SDK*.

See Also

Reference

CComboBoxEx Class
Hierarchy Chart
CComboBoxEx::SetItem
CComboBoxEx::InsertItem

Other Resources

CComboBoxEx Members

CComboBoxEx::HasEditChanged

Determines if the user has changed the contents of the **ComboBoxEx** edit control by typing.

```
BOOL HasEditChanged( );
```

Return Value

Nonzero if the user has typed in the control's edit box; otherwise 0.

Remarks

This member function implements the functionality of the message CBEM_HASEDITCHANGED, as described in the *Platform SDK*.

See Also

Reference

CComboBoxEx Class

Hierarchy Chart

Other Resources

CComboBoxEx Members

CComboBoxEx::InsertItem

Inserts a new item in a **ComboBoxEx** control.

```
int InsertItem(
   const COMBOBOXEXITEM* pCBItem
);
```

Parameters

pCBItem

A pointer to a COMBOBOXEXITEM structure that will receive the item information. This structure contains callback flag values for the item.

Return Value

The index at which the new item was inserted if successful; otherwise -1.

Remarks

When you call **InsertItem**, a WM_NOTIFY message with CBEN_INSERTITEM notification will be sent to the parent window.

See Also

Reference

CComboBoxEx Class
Hierarchy Chart
CComboBoxEx::GetItem
CComboBoxEx::SetItem

Other Resources

CComboBoxEx Members

CComboBoxEx::SetExtendedStyle

Call this member function to set the extended styles used for a combo box extended control.

```
DWORD SetExtendedStyle(
   DWORD dwExMask,
   DWORD dwExStyles
);
```

Parameters

dwExMask

A **DWORD** value that indicates which styles in *dwExStyles* are to be affected. Only the extended styles in *dwExMask* will be changed. All other styles will be maintained as is. If this parameter is zero, then all of the styles in *dwExStyles* will be affected.

dwExStyles

A **DWORD** value that contains the combo box control extended styles to set for the control.

Return Value

A **DWORD** value that contains the extended styles previously used for the control.

Remarks

See ComboBoxEx Control Extended Styles in the *Platform SDK* for more information about these styles.

To create a combo box extended control with extended windows styles, use CreateEx.

See Also

Reference

CComboBoxEx Class
Hierarchy Chart
CComboBoxEx::GetExtendedStyle

Other Resources

CComboBoxEx Members

CComboBoxEx::SetImageList

Sets an image list for a **ComboBoxEx** control.

```
CImageList* SetImageList(
   CImageList* pImageList
);
```

Parameters

pImageList

A pointer to a **CImageList** object containing the images to use with the **CComboBoxEx** control.

Return Value

A pointer to a CImageList object containing the images previously used by the **CComboBoxEx** control. **NULL** if no image list was previously set.

Remarks

This member function implements the functionality of the message CBEM_SETIMAGELIST, as described in the *Platform SDK*. If you change the height of the default edit control, call the Win32 function SetWindowPos to resize your control after you call **SetImageList**, or it will not display properly.

The **CImageList** object pointed to by the return value is a temporary object and is destroyed during the next idle processing time.

See Also

Reference

CComboBoxEx Class
Hierarchy Chart
CComboBoxEx::GetImageList

Other Resources

CComboBoxEx Members

CComboBoxEx::SetItem

Sets the attributes for an item in a **ComboBoxEx** control.

```
BOOL SetItem(
   const COMBOBOXEXITEM* pCBItem
);
```

Parameters

pCBItem

A pointer to a COMBOBOXEXITEM structure that will receive the item information.

Return Value

Nonzero if the operation was successful; otherwise 0.

Remarks

This member function implements the functionality of the message CBEM_SETITEM, as described in the *Platform SDK*.

See Also

Reference

CComboBoxEx Class
Hierarchy Chart
CComboBoxEx::GetItem
CComboBoxEx::InsertItem

Other Resources

CComboBoxEx Members

CComboBoxEx::SetWindowTheme

Sets the visual style of the extended combo box control.

```
HRESULT SetWindowTheme(
   LPCWSTR pszSubAppName
);
```

Parameters

pszSubAppName

A pointer to a Unicode string that contains the extended combo box visual style to set.

Return Value

The return value is not used.

Remarks

This member function emulates the functionality of the CBEM_SETWINDOWTHEME message, as described in the *Platform SDK*.

See Also

Reference

CComboBoxEx Class

Hierarchy Chart

Other Resources

CComboBoxEx Members

CCommandLineInfo Class

Aids in parsing the command line at application startup.

```
class CCommandLineInfo : public CObject
```

Remarks

An MFC application will typically create a local instance of this class in the InitInstance function of its application object. This object is then passed to CWinApp::ParseCommandLine, which repeatedly calls ParseParam to fill the **CCommandLineInfo** object. The **CCommandLineInfo** object is then passed to CWinApp::ProcessShellCommand to handle the command-line arguments and flags.

You can use this object to encapsulate the following command-line options and parameters:

Command-line argument	Command executed
app	New file.
app filename	Open file.
app **/p** filename	Print file to default printer.
app **/pt** filename printer driver port	Print file to the specified printer.
app **/dde**	Start up and await DDE command.
app **/Automation**	Start up as an OLE automation server.
app **/Embedding**	Start up to edit an embedded OLE item.
app **/Register** *app* **/Regserver**	Informs the application to perform any registration tasks.
app **/Unregister** *app* **/Unregserver**	Informs the application to perform any un-registration tasks.

Derive a new class from **CCommandLineInfo** to handle other flags and parameter values. Override ParseParam to handle the new flags.

Requirements

Header: afxwin.h

See Also

Reference

CObject Class
Hierarchy Chart
CWinApp::ParseCommandLine
CWinApp::ProcessShellCommand

Other Resources

CCommandLineInfo Members

CCommandLineInfo Members

Base Class Members

CObject Members

CCommandLineInfo Members

Construction

CommandLineInfo	Constructs a default **CCommandLineInfo** object.

Operations

ParseParam	Override this callback to parse individual parameters.

Data Members

m_bRunAutomated	Indicates the command-line **/Automation** option was found.
m_bRunEmbedded	Indicates the command-line **/Embedding** option was found.
m_bShowSplash	Indicates if a splash screen should be shown.
m_nShellCommand	Indicates the shell command to be processed.
m_strDriverName	Indicates the driver name if the shell command is Print To; otherwise empty.
m_strFileName	Indicates the filename to be opened or printed; empty if the shell command is New or DDE.
m_strPortName	Indicates the port name if the shell command is Print To; otherwise empty.
m_strPrinterName	Indicates the printer name if the shell command is Print To; otherwise empty.

See Also

Reference

CCommandLineInfo Class

Hierarchy Chart

CCommandLineInfo Member Functions

For information about the member functions in **CCommandLineInfo**, see CCommandLineInfo Members.

CCommandLineInfo::CCommandLineInfo

This constructor creates a **CCommandLineInfo** object with default values.

```
CCommandLineInfo( );
```

Remarks

The default is to show the splash screen (**m_bShowSplash = TRUE**) and to execute the New command on the File menu (**m_nShellCommand = NewFile**).

The application framework calls ParseParam to fill data members of this object.

Example

```
CCommandLineInfo cmdinfo;
ProcessCommandLine(cmdinfo);
```

See Also

Reference

CCommandLineInfo Class
Hierarchy Chart
CCommandLineInfo::ParseParam

Other Resources

CCommandLineInfo Members

CCommandLineInfo::ParseParam

The framework calls this function to parse/interpret individual parameters from the command line. The second version is only available in Unicode projects.

```
virtual void ParseParam(
   const char* pszParam,
   BOOL bFlag,
   BOOL bLast
);
virtual void ParseParam(
   const TCHAR* pszParam,
   BOOL bFlag,
   BOOL bLast
);
```

Parameters

pszParam

The parameter or flag.

bFlag

Indicates whether *pszParam* is a parameter or a flag.

bLast

Indicates if this is the last parameter or flag on the command line.

Remarks

CWinApp::ParseCommandLine calls **ParseParam** once for each parameter or flag on the command line, passing the argument to *pszParam*. If the first character of the parameter is a '**-**' or a '**/**', then it is removed and *bFlag* is set to **TRUE**. When parsing the final parameter, *bLast* is set to **TRUE**.

The default implementation of this function recognizes the following flags: **/p**, **/pt**, **/dde**, **/Automation**, and **/Embedding**, as shown in the following table:

Command-line argument	**Command executed**
app	New file.
app filename	Open file.
app **/p** filename	Print file to default printer.
app **/pt** filename printer driver port	Print file to the specified printer.
app **/dde**	Start up and await DDE command.
app **/Automation**	Start up as an OLE automation server.
app **/Embedding**	Start up to edit an embedded OLE item.
app **/Register** *app* **/Regserver**	Informs the application to perform any registration tasks.
app **/Unregister** *app* **/Unregserver**	Informs the application to perform any un-registration tasks.

This information is stored in m_bRunAutomated, m_bRunEmbedded, and m_nShellCommand. Flags are marked by either a forward-slash '**/**' or hyphen '**-**'.

The default implementation puts the first non-flag parameter into m_strFileName. In the case of the **/pt** flag, the default implementation puts the second, third, and fourth non-flag parameters into m_strPrinterName, m_strDriverName, and m_strPortName, respectively.

The default implementation also sets m_bShowSplash to **TRUE** only in the case of a new file. In the case of a new file, the user has taken action involving the application itself. In any other case, including opening existing files using the shell, the user action involves the file directly. In a document-centric standpoint, the splash screen does not need to announce the application starting up.

Override this function in your derived class to handle other flag and parameter values.

See Also

Reference

CCommandLineInfo Class

Hierarchy Chart

CWinApp::ParseCommandLine

Other Resources

CCommandLineInfo Members

CCommandLineInfo Data Members

For information about the data members in **CCommandLineInfo**, see CCommandLineInfo Members.

CCommandLineInfo::m_bRunAutomated

Indicates that the **/Automation** flag was found on the command line.

```
BOOL m_bRunAutomated;
```

Remarks

If **TRUE**, this means start up as an OLE automation server.

See Also

Reference

CCommandLineInfo Class
Hierarchy Chart
CCommandLineInfo::ParseParam
CWinApp::ProcessShellCommand

Other Resources

CCommandLineInfo Members

CCommandLineInfo::m_bRunEmbedded

Indicates that the **/Embedding** flag was found on the command line.

```
BOOL m_bRunEmbedded;
```

Remarks

If **TRUE**, this means start up for editing an embedded OLE item.

See Also

Reference

CCommandLineInfo Class
Hierarchy Chart
CCommandLineInfo::m_bShowSplash
CWinApp::ProcessShellCommand

Other Resources

CCommandLineInfo Members

CCommandLineInfo::m_bShowSplash

Indicates that the splash screen should be displayed.

```
BOOL m_bShowSplash;
```

Remarks

If **TRUE**, this means the splash screen for this application should be displayed during startup. The default implementation of ParseParam sets this data member to **TRUE** if m_nShellCommand is equal to **CCommandLineInfo::FileNew**.

See Also

Reference

CCommandLineInfo Class
Hierarchy Chart
CCommandLineInfo::m_bRunAutomated
CCommandLineInfo::m_bRunEmbedded
CCommandLineInfo::m_nShellCommand
CCommandLineInfo::ParseParam
CWinApp::ProcessShellCommand

Other Resources

CCommandLineInfo Members

CCommandLineInfo::m_nShellCommand

Indicates the shell command for this instance of the application.

```
m_nShellCommand;
```

Remarks

The type for this data member is the following enumerated type, which is defined within the **CCommandLineInfo** class.

```
enum{
   FileNew,
   FileOpen,
   FilePrint,
   FilePrintTo,
   FileDDE,
   FileNothing = -1
};
```

For a brief description of these values, see the following list.

- **CCommandLineInfo::FileNew** Indicates that no filename was found on the command line.
- **CCommandLineInfo::FileOpen** Indicates that a filename was found on the command line and that none of the following flags were found on the command line: **/p**, **/pt**, **/dde**.
- **CCommandLineInfo::FilePrint** Indicates that the **/p** flag was found on the command line.
- **CCommandLineInfo::FilePrintTo** Indicates that the **/pt** flag was found on the command line.
- **CCommandLineInfo::FileDDE** Indicates that the **/dde** flag was found on the command line.
- **CCommandLineInfo::FileNothing** Turns off the display of a new MDI child window on startup. By design, Application Wizard-generated MDI applications display a new child window on startup. To turn off this feature, an application can use **CCommandLineInfo::FileNothing** as the shell command when calling ProcessShellCommand. **ProcessShellCommand** is called by the **InitInstance( )** of all **CWinApp** derived classes.

Example

```
BOOL CMyWinApp::InitInstance()
{
...
// Parse command line for standard shell commands, DDE, file open
   CCommandLineInfo cmdInfo;
   ParseCommandLine(cmdInfo);
// DON'T display a new MDI child window during startup!!!
   cmdInfo.m_nShellCommand = CCommandLineInfo::FileNothing;
// Dispatch commands specified on the command line
   if (!ProcessShellCommand(cmdInfo))
      return FALSE;
...
};
```

See Also

Reference

CCommandLineInfo Class
Hierarchy Chart
CCommandLineInfo::m_strFileName
CCommandLineInfo::m_strPrinterName
CCommandLineInfo::m_strDriverName
CCommandLineInfo::m_strPortName
CWinApp::ProcessShellCommand

Other Resources

CCommandLineInfo Members

CCommandLineInfo::m_strDriverName

Stores the value of the third non-flag parameter on the command line.

```
CString m_strDriverName;
```

Remarks

This parameter is typically the name of the printer driver for a Print To shell command. The default implementation of ParseParam sets this data member only if the **/pt** flag was found on the command line.

See Also

Reference

CCommandLineInfo Class
Hierarchy Chart
CCommandLineInfo::m_strFileName
CCommandLineInfo::m_strPrinterName
CCommandLineInfo::m_strPortName
CWinApp::ProcessShellCommand

Other Resources

CCommandLineInfo Members

CCommandLineInfo::m_strFileName

Stores the value of the first non-flag parameter on the command line.

```
CString m_strFileName;
```

Remarks

This parameter is typically the name of the file to open.

See Also

Reference

CCommandLineInfo Class
Hierarchy Chart
CCommandLineInfo::m_strPrinterName
CCommandLineInfo::m_strDriverName
CCommandLineInfo::m_strPortName
CWinApp::ProcessShellCommand

Other Resources

CCommandLineInfo Members

CCommandLineInfo::m_strPortName

Stores the value of the fourth non-flag parameter on the command line.

```
CString m_strPortName;
```

Remarks

This parameter is typically the name of the printer port for a Print To shell command. The default implementation of ParseParam sets this data member only if the **/pt** flag was found on the command line.

See Also

Reference

CCommandLineInfo Class
Hierarchy Chart
CCommandLineInfo::m_strFileName
CCommandLineInfo::m_strPrinterName
CCommandLineInfo::m_strDriverName
CWinApp::ProcessShellCommand

Other Resources

CCommandLineInfo Members

CCommandLineInfo::m_strPrinterName

Stores the value of the second non-flag parameter on the command line.

```
CString m_strPrinterName;
```

Remarks

This parameter is typically the name of the printer for a Print To shell command. The default implementation of ParseParam sets this data member only if the **/pt** flag was found on the command line.

See Also

Reference

CCommandLineInfo Class
Hierarchy Chart
CCommandLineInfo::m_strFileName
CCommandLineInfo::m_strDriverName
CCommandLineInfo::m_strPortName
CWinApp::ProcessShellCommand

Other Resources

CCommandLineInfo Members

CCommonDialog Class

The base class for classes that encapsulate functionality of the Windows common dialogs.

```
class CCommonDialog : public CDialog
```

Remarks

The following classes encapsulate the functionality of the Windows common dialogs:

- CFileDialog
- CFontDialog
- CColorDialog
- CPageSetupDialog
- CPrintDialog
- CPrintDialogEx
- CFindReplaceDialog
- COleDialog

Requirements

Header: afxdlgs.h

See Also

Reference

CDialog Class
Hierarchy Chart
CFileDialog Class
CFontDialog Class
CColorDialog Class
CPageSetupDialog Class
CPrintDialog Class
CFindReplaceDialog Class
COleDialog Class

Other Resources

CCommonDialog Members

CCommonDialog Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CDialog Members

Construction

CCommonDialog	Constructs a **CCommonDialog** object.

See Also

Reference

CCommonDialog Class

Hierarchy Chart

CCommonDialog Member Functions

For information about the member functions in **CCommonDialog**, see CCommonDialog Members.

CCommonDialog::CCommonDialog

Constructs a **CCommonDialog** object.

```
explicit CCommonDialog(
   CWnd* pParentWnd
);
```

Parameters

pParentWnd

Points to the parent or owner window object (of type CWnd) to which the dialog object belongs. If it is **NULL**, the dialog object's parent window is set to the main application window.

Remarks

See CDialog::CDialog for complete information.

See Also

Reference

CCommonDialog Class
Hierarchy Chart
CDialog::CDialog

Other Resources

CCommonDialog Members

CConnectionPoint Class

Defines a special type of interface used to communicate with other OLE objects, called a "connection point."

```
class CConnectionPoint : public CCmdTarget
```

Remarks

Unlike normal OLE interfaces, which are used to implement and expose the functionality of an OLE control, a connection point implements an outgoing interface that is able to initiate actions on other objects, such as firing events and change notifications.

A connection consists of two parts: the object calling the interface, called the "source," and the object implementing the interface, called the "sink." By exposing a connection point, a source allows sinks to establish connections to itself. Through the connection point mechanism, a source object obtains a pointer to the sink's implementation of a set of member functions. For example, to fire an event implemented by the sink, the source can call the appropriate method of the sink's implementation.

By default, a **COleControl**-derived class implements two connection points: one for events and one for property change notifications. These connections are used, respectively, for event firing and for notifying a sink (for example, the control's container) when a property value has changed. Support is also provided for OLE controls to implement additional connection points. For each additional connection point implemented in your control class, you must declare a "connection part" that implements the connection point. If you implement one or more connection points, you also need to declare a single "connection map" in your control class.

The following example demonstrates a simple connection map and one connection point for the `Sample` OLE control, consisting of two fragments of code: the first portion declares the connection map and point; the second implements this map and point. The first fragment is inserted into the declaration of the control class, under the **protected** section:

```
    // Connection point for ISample interface
    BEGIN_CONNECTION_PART(CSampleCtrl, SampleConnPt)
        CONNECTION_IID(IID_ISampleSink)
    END_CONNECTION_PART(SampleConnPt)

    DECLARE_CONNECTION_MAP()
```

The **BEGIN_CONNECTION_PART** and **END_CONNECTION_PART** macros declare an embedded class, `XSampleConnPt` (derived from **CConnectionPoint**) that implements this particular connection point. If you want to override any **CConnectionPoint** member functions, or add member functions of your own, declare them between these two macros. For example, the **CONNECTION_IID** macro overrides the **CConnectionPoint::GetIID** member function when placed between these two macros.

The second code fragment is inserted into the implementation file (.CPP) of your control class. This code implements the connection map, which includes the additional connection point, `SampleConnPt`:

```
BEGIN_CONNECTION_MAP(CSampleCtrl, COleControl)
    CONNECTION_PART(CSampleCtrl, IID_ISampleSink, SampleConnPt)
END_CONNECTION_MAP()
```

Once these code fragments have been inserted, the Sample OLE control exposes a connection point for the **ISampleSink** interface.

Typically, connection points support "multicasting", which is the ability to broadcast to multiple sinks connected to the same interface. The following code fragment demonstrates how to accomplish multicasting by iterating through each sink on a connection point:

```
void CSampleCtrl::CallSinkFunc()
{
    const CPtrArray* pConnections = m_xSampleConnPt.GetConnections();
    ASSERT(pConnections != NULL);

    int cConnections = pConnections->GetSize();
    ISampleSink* pSampleSink;
```

```
    for (int i = 0; i < cConnections; i++)
    {
        pSampleSink = (ISampleSink*)(pConnections->GetAt(i));
        if(pSampleSink != NULL)
            pSampleSink->SinkFunc();
    }
}
```

This example retrieves the current set of connections on the `SampleConnPt` connection point with a call to `CConnectionPoint::GetConnections`. It then iterates through the connections and calls `ISampleSink::SinkFunc` on every active connection.

For more information on using **CConnectionPoint**, see the article Connection Points.

Requirements

Header: afxdisp.h

See Also

Reference

CCmdTarget Class

Hierarchy Chart

Other Resources

CConnectionPoint Members

CConnectionPoint Members

Base Class Members

CObject Members

CCmdTarget Members

Constructors

CConnectionPoint	Constructs a **CConnectionPoint** object.

Operations

GetConnections	Retrieves all connection points in a connection map.
GetNextConnection	Retrieves a pointer to the connection element at *pos*.
GetStartPosition	Starts a map iteration by returning a **POSITION** value that can be passed to a **GetNextConnection** call.
QuerySinkInterface	Retrieves a pointer to the requested sink interface.

Overridables

GetContainer	Retrieves the container of the control that owns the connection map.
GetIID	Retrieves the interface ID of a connection point.
GetMaxConnections	Retrieves the maximum number of connection points supported by a control.
OnAdvise	Called by the framework when establishing or breaking connections.

See Also

Reference

CConnectionPoint Class

Hierarchy Chart

CConnectionPoint Member Functions

For information about the member functions in **CConnectionPoint**, see CConnectionPoint Members.

CConnectionPoint::CConnectionPoint

Constructs a **CConnectionPoint** object.

```
CConnectionPoint( );
```

See Also

Reference

CConnectionPoint Class

Hierarchy Chart

Other Resources

CConnectionPoint Members

CConnectionPoint::GetConnections

Call this function to retrieve all active connections for a connection point.

```
const CPtrArray* GetConnections( );
```

Return Value

A pointer to an array of active connections (sinks). Some of the pointers in the array may be NULL. Each non-NULL pointer in this array can be safely converted to a pointer to the sink interface using a cast operator.

See Also

Reference

CConnectionPoint Class
Hierarchy Chart
CConnectionPoint::GetMaxConnections

Other Resources

CConnectionPoint Members

CConnectionPoint::GetContainer

Called by the framework to retrieve the **IConnectionPointContainer** for the connection point.

```
virtual LPCONNECTIONPOINTCONTAINER GetContainer();
```

Return Value

If successful, a pointer to the container; otherwise **NULL**.

Remarks

This function is typically implemented by the **BEGIN_CONNECTION_PART** macro.

See Also

Reference

CConnectionPoint Class
Hierarchy Chart
BEGIN_CONNECTION_PART

Other Resources

CConnectionPoint Members

CConnectionPoint::GetIID

Called by the framework to retrieve the interface ID of a connection point.

```
virtual REFIID GetIID( ) = 0;
```

Return Value

A reference to the connection point's interface ID.

Remarks

Override this function to return the interface ID for this connection point.

See Also

Reference

CConnectionPoint Class
Hierarchy Chart
CONNECTION_IID

Other Resources

CConnectionPoint Members

CConnectionPoint::GetMaxConnections

Called by the framework to retrieve the maximum number of connections supported by the connection point.

```
virtual int GetMaxConnections( );
```

Return Value

The maximum number of connections supported by the control, or -1 if no limit.

Remarks

The default implementation returns -1, indicating no limit.

Override this function if you want to limit the number of sinks that can connect to your control.

See Also

Reference

CConnectionPoint Class
Hierarchy Chart
CConnectionPoint::GetConnections

Other Resources

CConnectionPoint Members

CConnectionPoint::GetNextConnection

Retrieves a pointer to the connection element at *pos*.

```
LPUNKNOWN GetNextConnection(
   POSITION& pos
) const;
```

Parameters

pos

Specifies a reference to a **POSITION** value returned by a previous **GetNextConnection** or GetStartPosition call.

Return Value

A pointer to the connection element specified by *pos*, or NULL.

Remarks

This function is most useful for iterating through all the elements in the connection map. When iterating, skip any NULLs returned from this function.

Example

```
void CSampleCtrl::CallSinkFunc()
{
    POSITION pos = m_xSampleConnPt.GetStartPosition();
    ISampleSink* pSampleSink;
    while( pos != NULL )
    {
        pSampleSink = (ISampleSink*)(m_xSampleConnPt.GetNextConnection(pos));
        if(pSampleSink != NULL)
            pSampleSink->SinkFunc();
    }
}
```

See Also

Reference

CConnectionPoint Class

Hierarchy Chart

Other Resources

CConnectionPoint Members

CConnectionPoint::GetStartPosition

Starts a map iteration by returning a **POSITION** value that can be passed to a GetNextConnection call.

```
POSITION GetStartPosition( ) const;
```

Return Value

A **POSITION** value that indicates a starting position for iterating the map; or **NULL** if the map is empty.

Remarks

The iteration sequence is not predictable; therefore, the "first element in the map" has no special significance.

Example

See the example for CConnectionPoint::GetNextConnection.

See Also

Reference

CConnectionPoint Class

Hierarchy Chart

Other Resources

CConnectionPoint Members

CConnectionPoint::OnAdvise

Called by the framework when a connection is being established or broken.

```
virtual void OnAdvise(
   BOOL bAdvise
);
```

Parameters

bAdvise

TRUE, if a connection is being established; otherwise **FALSE**.

Remarks

The default implementation does nothing.

Override this function if you want notification when sinks connect to or disconnect from your connection point.

See Also

Reference

CConnectionPoint Class

Hierarchy Chart

Other Resources

CConnectionPoint Members

CConnectionPoint::QuerySinkInterface

Retrieves a pointer to the requested sink interface.

```
virtual HRESULT QuerySinkInterface(
   LPUNKNOWN pUnkSink,
   void** ppInterface
);
```

Parameters

pUnkSink

The identifier of the sink interface being requested.

ppInterface

A pointer to the interface pointer identified by *pUnkSink*. If the object does not support this interface, **ppInterface* is set to **NULL**.

Return Value

A standard **HRESULT** value.

See Also

Reference

CConnectionPoint Class

Hierarchy Chart

Other Resources

CConnectionPoint Members

CControlBar Class

The base class for the control-bar classes CStatusBar, CToolBar, CDialogBar, CReBar, and COleResizeBar.

```
class CControlBar : public CWnd
```

Remarks

A control bar is a window that is usually aligned to the left or right of a frame window. It may contain child items that are either **HWND**-based controls, which are Windows windows that generate and respond to Windows messages, or non-**HWND**-based items, which are not windows and are managed by application code or framework code. List boxes and edit controls are examples of **HWND**-based controls; status-bar panes and bitmap buttons are examples of non-**HWND**-based controls.

Control-bar windows are usually child windows of a parent frame window and are usually siblings to the client view or MDI client of the frame window. A **CControlBar** object uses information about the parent window's client rectangle to position itself. It then informs the parent window as to how much space remains unallocated in the parent window's client area.

For more information on **CControlBar**, see:

- Control Bars
- Technical Note 31: Control Bars.
- Knowledge Base article Q242577 : PRB: Update Command UI Handlers Do Not Work for Menu Attached to a Dialog Box

Requirements

Header: afxext.h

See Also

Tasks

CTRLBARS Sample: Illustrates Custom Control Bars

Reference

CWnd Class
Hierarchy Chart
CToolBar Class
CDialogBar Class
CStatusBar Class
CReBar Class

Other Resources

CControlBar Members

CControlBar Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Data Members

m_bAutoDelete	If nonzero, the **CControlBar** object is deleted when the Windows control bar is destroyed.
m_pInPlaceOwner	The in-place owner of the control bar.

Attributes

CalcDynamicLayout	Returns the size of a dynamic control bar as a CSize object.
CalcFixedLayout	Returns the size of the control bar as a CSize object.
CControlBar	Constructs a **CControlBar** object.
GetBarStyle	Retrieves the control bar style settings.
GetBorders	Retrieves the border values of the control bar.
GetCount	Returns the number of non-**HWND** elements in the control bar.
GetDockingFrame	Returns a pointer to the frame to which a control bar is docked.
IsFloating	Returns a nonzero value if the control bar in question is a floating control bar.
SetBarStyle	Modifies the control bar style settings.
SetBorders	Sets the border values of the control bar.
SetInPlaceOwner	Changes the in-place owner of a control bar.

Overridables

OnUpdateCmdUI	Calls the Command UI handlers.

Operations

EnableDocking	Allows a control bar to be docked or floating.

Gripper Bar Painting

CalcInsideRect	Returns the current dimensions of the control bar area; including the borders.
DoPaint	Renders the borders and gripper of the control bar.
DrawBorders	Renders the borders of the control bar.
DrawGripper	Renders the gripper of the control bar.

See Also

Reference

CControlBar Class
Hierarchy Chart

CControlBar Member Functions

For information about the member functions in **CControlBar**, see CControlBar Members.

CControlBar::CalcDynamicLayout

The framework calls this member function to calculate the dimensions of a dynamic toolbar.

```
virtual CSize CalcDynamicLayout(
   int nLength,
   DWORD nMode
);
```

Parameters

nLength

The requested dimension of the control bar, either horizontal or vertical, depending on *dwMode*.

nMode

The following predefined flags are used to determine the height and width of the dynamic control bar. Use the bitwise-OR (|) operator to combine the flags.

Layout mode flags	What it means
LM_STRETCH	Indicates whether the control bar should be stretched to the size of the frame. Set if the bar is not a docking bar (not available for docking). Not set when the bar is docked or floating (available for docking). If set, **LM_STRETCH** ignores *nLength* and returns dimensions based on the **LM_HORZ** state. **LM_STRETCH** works similarly to the *bStretch* parameter used in CalcFixedLayout; see that member function for more information about the relationship between stretching and orientation.
LM_HORZ	Indicates that the bar is horizontally or vertically oriented. Set if the bar is horizontally oriented, and if it is vertically oriented, it is not set. **LM_HORZ** works similarly to the *bHorz* parameter used in CalcFixedLayout; see that member function for more information about the relationship between stretching and orientation.
LM_MRUWIDTH	Most Recently Used Dynamic Width. Ignores *nLength* parameter and uses the remembered most recently used width.
LM_HORZDOCK	Horizontal Docked Dimensions. Ignores *nLength* parameter and returns the dynamic size with the largest width.
LM_VERTDOCK	Vertical Docked Dimensions. Ignores *nLength* parameter and returns the dynamic size with the largest height.
LM_LENGTHY	Set if *nLength* indicates height (Y-direction) instead of width.

LM_COMMIT	Resets **LM_MRUWIDTH** to current width of floating control bar.

Return Value

The control bar size, in pixels, of a CSize object.

Remarks

Override this member function to provide your own dynamic layout in classes you derive from **CControlBar**. MFC classes derived from **CControlBar**, such as CToolbar, override this member function and provide their own implementation.

See Also

Reference

CControlBar Class
Hierarchy Chart
CControlBar::CalcFixedLayout
CToolBar Class

Other Resources

CControlBar Members

CControlBar::CalcFixedLayout

Call this member function to calculate the horizontal size of a control bar.

```
virtual CSize CalcFixedLayout(
   BOOL bStretch,
   BOOL bHorz
);
```

Parameters

bStretch

Indicates whether the bar should be stretched to the size of the frame. The *bStretch* parameter is nonzero when the bar is not a docking bar (not available for docking) and is 0 when it is docked or floating (available for docking).

bHorz

Indicates that the bar is horizontally or vertically oriented. The *bHorz* parameter is nonzero if the bar is horizontally oriented and is 0 if it is vertically oriented.

Return Value

The control bar size, in pixels, of a **CSize** object.

Remarks

Control bars such as toolbars can stretch horizontally or vertically to accommodate the buttons contained in the control bar.

If *bStretch* is **TRUE**, stretch the dimension along the orientation provided by *bHorz*. In other words, if *bHorz* is **FALSE**, the control bar is stretched vertically. If *bStretch* is **FALSE**, no stretch occurs. The following table shows the possible permutations, and resulting control-bar styles, of *bStretch* and *bHorz*.

bStretch	bHorz	Stretching	Orientation	Docking/Not docking
TRUE	**TRUE**	Horizontal stretching	Horizontally oriented	Not docking
TRUE	**FALSE**	Vertical stretching	Vertically oriented	Not docking
FALSE	**TRUE**	No stretching available	Horizontally oriented	Docking
FALSE	**FALSE**	No stretching available	Vertically oriented	Docking

See Also

Reference

CControlBar Class

Hierarchy Chart

CControlBar::CalcDynamicLayout

Other Resources

CControlBar Members

CControlBar::CalcInsideRect

The framework calls this function to calculate the client area of the control bar.

```
virtual void CalcInsideRect(
   CRect& rect,
   BOOL bHorz
) const;
```

Parameters

rect

Contains the current dimensions of the control bar; including the borders.

bHorz

Indicates that the bar is horizontally or vertically oriented. The *bHorz* parameter is nonzero if the bar is horizontally oriented and is 0 if it is vertically oriented.

Remarks

This function is called before the control bar is painted.

Override this function to customize the rendering of the borders and gripper bar of the control bar.

See Also

Reference

CControlBar Class
Hierarchy Chart
CControlBar::DrawBorders
CControlBar::DrawGripper

Other Resources

CControlBar Members

CControlBar::CControlBar

Constructs a **CControlBar** object.

```
CControlBar( );
```

See Also

Reference

CControlBar Class

Hierarchy Chart

Other Resources

CControlBar Members

CControlBar::DoPaint

Called by the framework to render the borders and gripper bar of the control bar.

```
virtual void DoPaint(
   CDC* pDC
);
```

Parameters

pDC

Points to the device context to be used for rendering the borders and gripper of the control bar.

Remarks

Override this function to customize the drawing behavior of the control bar.

Another customization method is to override the **DrawBorders** and **DrawGripper** functions and add custom drawing code for the borders and gripper. Because these methods are called by the default **DoPaint** method, an override of **DoPaint** is not needed.

See Also

Reference

CControlBar Class
Hierarchy Chart
CControlBar::DrawBorders
CControlBar::DrawGripper

Other Resources

CControlBar Members

CControlBar::DrawBorders

Called by the framework to render the borders of the control bar.

```
virtual void DrawBorders(
   CDC* pDC,
   CRect& rect
);
```

Parameters

pDC

Points to the device context to be used for rendering the borders of the control bar.

rect

A **CRect** object containing the dimensions of the control bar.

Remarks

Override this function to customize the appearance of the control bar borders.

See Also

Reference

CControlBar Class
Hierarchy Chart
CControlBar::DoPaint
CControlBar::DrawGripper

Other Resources

CControlBar Members

CControlBar::DrawGripper

Called by the framework to render the gripper of the control bar.

```
virtual void DrawGripper(
   CDC* pDC,
   const CRect& rect
);
```

Parameters

pDC

Points to the device context to be used for rendering the control bar gripper.

rect

A **CRect** object containing the dimensions of the control bar gripper.

Remarks

Override this function to customize the appearance of the control bar gripper.

See Also

Reference

CControlBar Class
Hierarchy Chart
CControlBar::DoPaint
CControlBar::DrawBorders

Other Resources

CControlBar Members

CControlBar::EnableDocking

Call this function to enable a control bar to be docked.

```
void EnableDocking(
   DWORD dwDockStyle
);
```

Parameters

dwDockStyle

Specifies whether the control bar supports docking and the sides of its parent window to which the control bar can be docked, if supported. Can be one or more of the following:

- **CBRS_ALIGN_TOP** Allows docking at the top of the client area.
- **CBRS_ALIGN_BOTTOM** Allows docking at the bottom of the client area.
- **CBRS_ALIGN_LEFT** Allows docking on the left side of the client area.
- **CBRS_ALIGN_RIGHT** Allows docking on the right side of the client area.
- **CBRS_ALIGN_ANY** Allows docking on any side of the client area.
- **CBRS_FLOAT_MULTI** Allows multiple control bars to be floated in a single mini-frame window.

If 0 (that is, indicating no flags), the control bar will not dock.

Remarks

The sides specified must match one of the sides enabled for docking in the destination frame window, or the control bar cannot be docked to that frame window.

See Also

Reference

CControlBar Class
Hierarchy Chart
CFrameWnd::EnableDocking
CFrameWnd::DockControlBar
CFrameWnd::FloatControlBar
CControlBar::SetBarStyle

Other Resources

CControlBar Members

CControlBar::GetBarStyle

Call this function to determine which **CBRS_** (control bar styles) settings are currently set for the control bar.

```
DWORD GetBarStyle( );
```

Return Value

The current **CBRS_** (control bar styles) settings for the control bar. See CControlBar::SetBarStyle for the complete list of available styles.

Remarks

Does not handle **WS_** (window style) styles.

See Also

Reference

CControlBar Class
Hierarchy Chart
CControlBar::SetBarStyle

Other Resources

CControlBar Members

CControlBar::GetBorders

Returns the current border values for the control bar.

```
CRect GetBorders( ) const;
```

Return Value

A **CRect** object that contains the current width (in pixels) of each side of the control bar object. For example, the value of the *left* member, of CRect object, is the width of the left hand border.

See Also

Reference

CControlBar Class

Hierarchy Chart

CControlBar::SetBorders

Other Resources

CControlBar Members

CControlBar::GetCount

Returns the number of non-**HWND** items on the **CControlBar** object.

```
int GetCount( ) const;
```

Return Value

The number of non-**HWND** items on the **CControlBar** object. This function returns 0 for a CDialogBar object.

Remarks

The type of the item depends on the derived object: panes for CStatusBar objects, and buttons and separators for CToolBar objects.

See Also

Reference

CControlBar Class
Hierarchy Chart
CToolBar::SetButtons
CStatusBar::SetIndicators
CStatusBar Class
CToolBar Class
CDialogBar Class

Other Resources

CControlBar Members

CControlBar::GetDockingFrame

Call this member function to obtain a pointer to the current frame window to which your control bar is docked.

```
CFrameWnd* GetDockingFrame( ) const;
```

Return Value

A pointer to a frame window if successful; otherwise **NULL**.

If the control bar is not docked to a frame window (that is, if the control bar is floating), this function will return a pointer to its parent CMiniFrameWnd.

Remarks

For more information about dockable control bars, see CControlBar::EnableDocking and CFrameWnd::DockControlBar.

See Also

Reference

CControlBar Class
Hierarchy Chart
CControlBar::EnableDocking
CFrameWnd::DockControlBar

Other Resources

CControlBar Members

CControlBar::IsFloating

Call this member function to determine whether the control bar is floating or docked.

```
BOOL IsFloating( ) const;
```

Return Value

Nonzero if the control bar is floating; otherwise 0.

Remarks

To change the state of a control bar from docked to floating, call CFrameWnd::FloatControlBar.

See Also

Reference

CControlBar Class
Hierarchy Chart
CFrameWnd::FloatControlBar

Other Resources

CControlBar Members

CControlBar::OnUpdateCmdUI

This member function is called by the framework to update the status of the toolbar or status bar.

```
virtual void OnUpdateCmdUI(
   CFrameWnd* pTarget,
   BOOL bDisableIfNoHndler
) = 0;
```

Parameters

pTarget

Points to the main frame window of the application. This pointer is used for routing update messages.

bDisableIfNoHndler

Flag that indicates whether a control that has no update handler should be automatically displayed as disabled.

Remarks

To update an individual button or pane, use the **ON_UPDATE_COMMAND_UI** macro in your message map to set an update handler appropriately. See ON_UPDATE_COMMAND_UI for more information about using this macro.

OnUpdateCmdUI is called by the framework when the application is idle. The frame window to be updated must be a child window, at least indirectly, of a visible frame window. **OnUpdateCmdUI** is an advanced overridable.

See Also

Reference

CControlBar Class
Hierarchy Chart
ON_UPDATE_COMMAND_UI
TN031: Control Bars

Other Resources

CControlBar Members

CControlBar::SetBarStyle

Call this function to set the desired **CBRS_** styles for the control bar.

```
void SetBarStyle(
   DWORD dwStyle
);
```

Parameters

dwStyle

The desired styles for the control bar. Can be one or more of the following:

- **CBRS_ALIGN_TOP** Allows the control bar to be docked to the top of the client area of a frame window.
- **CBRS_ALIGN_BOTTOM** Allows the control bar to be docked to the bottom of the client area of a frame window.
- **CBRS_ALIGN_LEFT** Allows the control bar to be docked to the left side of the client area of a frame window.
- **CBRS_ALIGN_RIGHT** Allows the control bar to be docked to the right side of the client area of a frame window.
- **CBRS_ALIGN_ANY** Allows the control bar to be docked to any side of the client area of a frame window.
- **CBRS_BORDER_TOP** Causes a border to be drawn on the top edge of the control bar when it would be visible.
- **CBRS_BORDER_BOTTOM** Causes a border to be drawn on the bottom edge of the control bar when it would be visible.
- **CBRS_BORDER_LEFT** Causes a border to be drawn on the left edge of the control bar when it would be visible.
- **CBRS_BORDER_RIGHT** Causes a border to be drawn on the right edge of the control bar when it would be visible.
- **CBRS_FLOAT_MULTI** Allows multiple control bars to be floated in a single mini-frame window.
- **CBRS_TOOLTIPS** Causes tool tips to be displayed for the control bar.
- **CBRS_FLYBY** Causes message text to be updated at the same time as tool tips.
- **CBRS_GRIPPER** Causes a gripper, similar to that used on bands in a **CReBar** object, to be drawn for any **CControlBar**-derived class.

Remarks

Does not affect the **WS_** (window style) settings.

See Also

Reference

CControlBar Class
Hierarchy Chart
CControlBar::GetBarStyle

Other Resources

CControlBar Members

CControlBar::SetBorders

Call this function to set the size of the control bar's borders.

```
void SetBorders(
   int cxLeft = 0,
   int cyTop = 0,
   int cxRight = 0,
   int cyBottom = 0
);
void SetBorders(
   LPCRECT lpRect
);
```

Parameters

cxLeft

The width (in pixels) of the control bar's left border.

cyTop

The height (in pixels) of the control bar's top border.

cxRight

The width (in pixels) of the control bar's right border.

cyBottom

The height (in pixels) of the control bar's bottom border.

lpRect

A pointer to a CRect object that contains the current width (in pixels)of each border of the control bar object.

Example

The following code example sets the top and bottom borders of the control bar to 5 pixels, and the left and right borders to 2 pixels:

```
m_myControlBar.SetBorders(2, 5, 2, 5);
```

See Also

Reference

CControlBar Class
Hierarchy Chart
CControlBar::GetBorders

Other Resources

CControlBar Members

CControlBar::SetInPlaceOwner

Changes the in-place owner of a control bar.

```
void SetInPlaceOwner(
   CWnd *pWnd
);
```

Parameters

pWnd

A pointer to a **CWnd** object.

Remarks

See Also

Reference

CControlBar Class

Hierarchy Chart

CControlBar::m_pInPlaceOwner

Other Resources

CControlBar Members

CControlBar Data Members

For information about the data members in **CControlBar**, see CControlBar Members.

CControlBar::m_bAutoDelete

If nonzero, the **CControlBar** object is deleted when the Windows control bar is destroyed.

```
BOOL m_bAutoDelete;
```

Remarks

m_bAutoDelete is a public variable of type **BOOL**.

A control-bar object is usually embedded in a frame-window object. In this case, **m_bAutoDelete** is 0 because the embedded control-bar object is destroyed when the frame window is destroyed.

Set this variable to a nonzero value if you allocate a **CControlBar** object on the heap and you do not plan to call **delete**.

See Also

Reference

CControlBar Class
Hierarchy Chart
CWnd::DestroyWindow

Other Resources

CControlBar Members

CControlBar::m_pInPlaceOwner

The in-place owner of the control bar.

```
CWnd* m_pInPlaceOwner;
```

See Also

Reference

CControlBar Class

Hierarchy Chart

CControlBar::SetInPlaceOwner

Other Resources

CControlBar Members

CCreateContext Structure

The framework uses the **CCreateContext** structure when it creates the frame windows and views associated with a document.

```
struct CCreateContext
```

Remarks

CCreateContext is a structure and does not have a base class.

When creating a window, the values in this structure provide information used to connect the components that make up a document and the view of its data. You will only need to use **CCreateContext** if you are overriding parts of the creation process.

A **CCreateContext** structure contains pointers to the document, the frame window, the view, and the document template. It also contains a pointer to a **CRuntimeClass** that identifies the type of view to create. The run-time class information and the current document pointer are used to create a new view dynamically. The following table suggests how and when each **CCreateContext** member might be used:

Member	What it is for
m_pNewViewClass	**CRuntimeClass** of the new view to create.
m_pCurrentDoc	The existing document to be associated with the new view.
m_pNewDocTemplate	The document template associated with the creation of a new MDI frame window.
m_pLastView	The original view upon which additional views are modeled, as in the creation of a splitter window's views or the creation of a second view on a document.
m_pCurrentFrame	The frame window upon which additional frame windows are modeled, as in the creation of a second frame window on a document.

When a document template creates a document and its associated components, it validates the information stored in the **CCreateContext** structure. For example, a view should not be created for a nonexistent document.

Note
All of the pointers in **CCreateContext** are optional and can be **NULL** if unspecified or unknown.

CCreateContext is used by the member functions listed under "See Also." Consult the descriptions of these functions for specific information if you plan to override them.

Here are a few general guidelines:

- When passed as an argument for window creation, as in **CWnd::Create**, **CFrameWnd::Create**, and **CFrameWnd::LoadFrame**, the create context specifies what the new window should be connected to. For most windows, the entire structure is optional and a **NULL** pointer can be passed.
- For overridable member functions, such as **CFrameWnd::OnCreateClient**, the **CCreateContext** argument is optional.
- For member functions involved in view creation, you must provide enough information to create the view. For example, for the first view in a splitter window, you must supply the view class information and the current document.

In general, if you use the framework defaults, you can ignore **CCreateContext**. If you attempt more advanced modifications, the Microsoft Foundation Class Library source code or the sample programs, such as VIEWEX, will guide you. If you do forget a required parameter, a framework assertion will tell you what you forgot.

For more information on **CCreateContext**, see the MFC sample VIEWEX.

Requirements

Header: afxext.h

See Also

Reference

Hierarchy Chart
CFrameWnd::Create
CFrameWnd::LoadFrame
CFrameWnd::OnCreateClient
CSplitterWnd::Create
CSplitterWnd::CreateView
CWnd::Create

CCriticalSection Class

Represents a "critical section" — a synchronization object that allows one thread at a time to access a resource or section of code.

```
class CCriticalSection : public CSyncObject
```

Remarks

Critical sections are useful when only one thread at a time can be allowed to modify data or some other controlled resource. For example, adding nodes to a linked list is a process that should only be allowed by one thread at a time. By using a **CCriticalSection** object to control the linked list, only one thread at a time can gain access to the list.

Note
The functionality of the **CCriticalSection** class is provided by an actual Win32 **CRITICAL_SECTION** object.

Critical sections are used instead of mutexes (see CMutex) when speed is critical and the resource will not be used across process boundaries.

There are two methods for using a **CCriticalSection** object: stand-alone and embedded in a class.

- Stand-alone method To use a stand-alone **CCriticalSection** object, construct the **CCriticalSection** object when it is needed. After a successful return from the constructor, explicitly lock the object with a call to Lock. Call Unlock when you are done accessing the critical section. This method, while clearer to someone reading your source code, is more prone to error as you must remember to lock and unlock the critical section before and after access.

  A more preferable method is to use the CSingleLock class. It also has a **Lock** and **Unlock** method, but you don't have to worry about unlocking the resource if an exception occurs.

- Embedded method You can also share a class with multiple threads by adding a **CCriticalSection**-type data member to the class and locking the data member when needed.

For more information on using **CCriticalSection** objects, see the article Multithreading: How to Use the Synchronization Classes.

Requirements

Header: afxmt.h

See Also

Reference

CSyncObject Class
Hierarchy Chart
CMutex Class

Other Resources

CCriticalSection Members

CCriticalSection Members

Base Class Members

CObject Members

CSyncObject Members

Construction

CCriticalSection	Constructs a **CCriticalSection** object.

Methods

Lock	Use to gain access to the **CCriticalSection** object.
Unlock	Releases the **CCriticalSection** object.

Operators

operator CRITICAL_SECTION *	Retrieves a pointer to the internal **CRITICAL_SECTION** object.

Data Members

m_sect	A **CRITICAL_SECTION** object.

See Also

Reference

CCriticalSection Class

Hierarchy Chart

CCriticalSection Member Functions

For information about the member functions in **CCriticalSection**, see CCriticalSection Members.

CCriticalSection::CCriticalSection

Constructs a **CCriticalSection** object.

```
CCriticalSection( );
```

Remarks

To access or release a **CCriticalSection** object, create a CSingleLock object and call its Lock and Unlock member functions. If the **CCriticalSection** object is being used stand-alone, call its Unlock member function to release it.

If the constructor fails to allocate the required system memory, a memory exception (of type CMemoryException) is automatically thrown.

Example

See the example for CCriticalSection::Lock.

See Also

Reference

CCriticalSection Class

Hierarchy Chart

Other Resources

CCriticalSection Members

CCriticalSection::Lock

Call this member function to gain access to the critical section object.

```
BOOL Lock( );
BOOL Lock(
   DWORD dwTimeout
);
```

Parameters

dwTimeout

Lock ignores this parameter value.

Return Value

Nonzero if the function was successful; otherwise 0.

Remarks

Lock is a blocking call that will not return until the critical section object is signaled (becomes available).

If timed waits are necessary, you can use a CMutex object instead of a **CCriticalSection** object.

If **Lock** fails to allocate the necessary system memory, a memory exception (of type CMemoryException) is automatically thrown.

Example

This example demonstrates the nested critical section approach by controlling access to a shared resource (the static `_strShared` object) using a shared **CCriticalSection** object. The `SomeMethod` function demonstrates updating a shared resource in a safe manner.

```
//Definition of critical section class
class CMyClass
{
   static CString _strShared; //shared resource
   static CCriticalSection _critSect;
public:
   CMyClass(void);
   ~CMyClass(void);
   void MyMethod(void); //locks, modifies, and unlocks shared resource
};

//Declaration of static members and SomeMethod
CString CMyClass::_strShared;
CCriticalSection CMyClass::_critSect;

void CMyClass::SomeMethod()
{
   _critSect.Lock();
   if (_strShared == "")
      _strShared = "<text>";
   _critSect.Unlock();
}
```

See Also

Reference

CCriticalSection Class

Hierarchy Chart

CSingleLock::Lock

Other Resources

CCriticalSection Members

CCriticalSection::Unlock

Releases the **CCriticalSection** object for use by another thread.

```
BOOL Unlock( );
```

Return Value

Nonzero if the **CCriticalSection** object was owned by the thread and the release was successful; otherwise 0.

Remarks

If the **CCriticalSection** is being used stand-alone, **Unlock** must be called immediately after completing use of the resource controlled by the critical section. If a CSingleLock object is being used, **CCriticalSection::Unlock** will be called by the lock object's **Unlock** member function.

Example

See the example for CCriticalSection::Lock.

See Also

Reference

CCriticalSection Class

Hierarchy Chart

Other Resources

CCriticalSection Members

CCriticalSection Operators

For information about the operators in **CCriticalSection**, see CCriticalSection Members.

CCriticalSection::operator CRITICAL_SECTION*

Retrieves a **CRITICAL_SECTION** object.

```
operator CRITICAL_SECTION*( );
```

Remarks

Call this function to retrieve a pointer to the internal **CRITICAL_SECTION** object.

See Also

Reference

CCriticalSection Class
Hierarchy Chart
CCriticalSection::m_sect

Other Resources

CCriticalSection Members

CCriticalSection Data Members

For information about the data members in **CCriticalSection**, see CCriticalSection Members.

CCriticalSection::m_sect

Contains a critical section object that is used by all **CCriticalSection** methods.

```
CRITICAL_SECTION m_sect;
```

See Also

Reference

CCriticalSection Class

Hierarchy Chart

CCriticalSection::operator CRITICAL_SECTION*

Other Resources

CCriticalSection Members

CCtrlView Class

Adapts the document-view architecture to the common controls supported by Windows 98 and Windows NT versions 3.51 and later.

```
class CCtrlView : public CView
```

Remarks

The class **CCtrlView** and its derivatives, CEditView, CListView, CTreeView, and CRichEditView, adapt the document-view architecture to the new common controls supported by Windows 95/98 and Windows NT versions 3.51 and later. For more information on the document-view architecture, see Document/View Architecture.

Requirements

Header: afxwin.h

See Also

Reference

CView Class
Hierarchy Chart
CTreeView Class
CListView Class
CRichEditView Class

Other Resources

CCtrlView Members

CCtrlView Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CView Members

Construction

CCtrlView	Constructs a **CCtrlView** object.

Overridables

OnDraw	Called by the framework to draw using the specified device context.
PreCreateWindow	Called before the creation of the Windows window attached to this **CCtrlView** object.

Operations

PreCreateWindow	Called before the creation of the Windows window attached to this **CCtrlView** object.

Data Members

m_dwDefaultStyle	Contains the default style for the view class.
m_strClass	Contains the Windows class name for the view class.

See Also

Reference

CCtrlView Class

Hierarchy Chart

CCtrlView Member Functions

For information about the member functions in **CCtrlView**, see CCtrlView Members.

CCtrlView::CCtrlView

Constructs a **CCtrlView** object.

```
CCtrlView(
   LPCTSTR lpszClass,
   DWORD dwStyle
);
```

Parameters

lpszClass

Windows class name of the view class.

dwStyle

Style of the view class.

Remarks

The framework calls the constructor when a new frame window is created or a window is split. Override CView::OnInitialUpdate to initialize the view after the document is attached. Call CWnd::Create or CWnd::CreateEx to create the Windows object.

See Also

Reference

CCtrlView Class
Hierarchy Chart
CWnd::PreCreateWindow

Other Resources

CCtrlView Members

CCtrlView::OnDraw

Called by the framework to draw the contents of the **CCtrlView** object using the specified device context.

```
virtual void OnDraw(
CDC* pDC
);
```

Parameters

pDC

A pointer to the device context in which the drawing occurs.

Remarks

OnDraw is typically called for screen display, passing a screen device context specified by *pDC*.

See Also

Reference

CCtrlView Class

Hierarchy Chart

Other Resources

CCtrlView Members

CCtrlView::PreCreateWindow

Called before the creation of the Windows window attached to this **CWnd** object.

```
virtual BOOL PreCreateWindow(
   CREATESTRUCT& cs
);
```

Parameters

cs

A CREATESTRUCT structure.

Return Value

Nonzero if the window creation should continue; 0 to indicate creation failure.

Remarks

Never call this function directly.

The default implementation of this function checks for a **NULL** window class name and substitutes an appropriate default. Override this member function to modify the **CREATESTRUCT** structure before the window is created.

Each class derived from **CCtrlView** adds its own functionality to its override of **PreCreateWindow**. By design, these derivations of **PreCreateWindow** are not documented. To determine the styles appropriate to each class and the interdependencies between the styles, you can examine the MFC source code for your application's base class. If you choose to override **PreCreateWindow**, you can determine whether the styles used in your application's base class provide the functionality you need by using information gathered from the MFC source code.

For more information on changing window styles, see the Changing the Styles of a Window Created by MFC.

See Also

Reference

CCtrlView Class

Hierarchy Chart

Other Resources

CCtrlView Members

CCtrlView Data Members

For information about the data members in **CCtrlView**, see CCtrlView Members.

CCtrlView::m_dwDefaultStyle

Contains the default style for the view class.

```
DWORD m_dwDefaultStyle;
```

Remarks

This style is applied when a window is created.

See Also

Reference

CCtrlView Class

Hierarchy Chart

CCtrlView::m_strClass

Other Resources

CCtrlView Members

CCtrlView::m_strClass

Contains the Windows class name for the view class.

```
CString m_strClass;
```

See Also

Reference

CCtrlView Class

Hierarchy Chart

CCtrlView::m_dwDefaultStyle

Other Resources

CCtrlView Members

CDaoDatabase Class

Represents a connection to a database through which you can operate on the data.

```
class CDaoDatabase : public CObject
```

Remarks

For information about the database formats supported, see the GetName member function. You can have one or more **CDaoDatabase** objects active at a time in a given "workspace," represented by a CDaoWorkspace object. The workspace maintains a collection of open database objects, called the Databases collection.

Note

The MFC DAO database classes are distinct from the MFC database classes based on ODBC. All DAO database class names have the "CDao" prefix. Class **CDaoDatabase** supplies an interface similar to that of the ODBC class CDatabase. The main difference is that **CDatabase** accesses the DBMS through Open Database Connectivity (ODBC) and an ODBC driver for that DBMS. **CDaoDatabase** accesses data through a Data Access Object (DAO) based on the Microsoft Jet database engine. In general, the MFC classes based on DAO are more capable than the MFC classes based on ODBC; the DAO-based classes can access data, including through ODBC drivers, via their own database engine. The DAO-based classes also support Data Definition Language (DDL) operations, such as adding tables via the classes, without having to call DAO directly.

Usage

You can create database objects implicitly, when you create recordset objects. But you can also create database objects explicitly. To use an existing database explicitly with **CDaoDatabase**, do either of the following:

- Construct a **CDaoDatabase** object, passing a pointer to an open CDaoWorkspace object.
- Or construct a **CDaoDatabase** object without specifying the workspace (MFC creates a temporary workspace object).

To create a new Microsoft Jet (.MDB) database, construct a **CDaoDatabase** object and call its Create member function. Do *not* call **Open** after **Create**.

To open an existing database, construct a **CDaoDatabase** object and call its Open member function.

Any of these techniques appends the DAO database object to the workspace's Databases collection and opens a connection to the data. When you then construct CDaoRecordset, CDaoTableDef, or CDaoQueryDef objects for operating on the connected database, pass the constructors for these objects a pointer to your **CDaoDatabase** object. When you finish using the connection, call the Close member function and destroy the **CDaoDatabase** object. **Close** closes any recordsets you have not closed previously.

Transactions

Database transaction processing is supplied at the workspace level — see the BeginTrans, CommitTrans, and Rollback member functions of class **CDaoWorkspace**.

ODBC Connections

The recommended way to work with ODBC data sources is to attach external tables to a Microsoft Jet (.MDB) database.

Collections

Each database maintains its own collections of tabledef, querydef, recordset, and relation objects. Class **CDaoDatabase** supplies member functions for manipulating these objects.

Note

The objects are stored in DAO, not in the MFC database object. MFC supplies classes for tabledef, querydef, and recordset objects but not for relation objects.

Requirements

Header: afxdao.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class
Hierarchy Chart
CDaoWorkspace Class
CDaoRecordset Class
CDaoTableDef Class
CDaoQueryDef Class
CDatabase Class
CDaoException Class

Other Resources

CDaoDatabase Members

CDaoDatabase Members

Base Class Members

CObject Members

CDaoDatabase Members

Data Members

m_pDAODatabase	A pointer to the underlying DAO database object.
m_pWorkspace	A pointer to the CDaoWorkspace object that contains the database and defines its transaction space.

Construction

CDaoDatabase	Constructs a **CDaoDatabase** object. Call **Open** to connect the object to a database.

Attributes

CanTransact	Returns nonzero if the database supports transactions.
CanUpdate	Returns nonzero if the **CDaoDatabase** object is updatable (not read-only).
GetConnect	Returns the connection string used to connect the **CDaoDatabase** object to a database. Used for ODBC.
GetName	Returns the name of the database currently in use.
GetQueryTimeout	Returns the number of seconds after which database query operations will time out. Affects all subsequent open, add new, update, and edit operations and other operations on ODBC data sources (only) such as **Execute** calls.
GetRecordsAffected	Returns the number of records affected by the last update, edit, or add operation or by a call to **Execute**.
GetVersion	Returns the version of the database engine associated with the database.
IsOpen	Returns nonzero if the **CDaoDatabase** object is currently connected to a database.
SetQueryTimeout	Sets the number of seconds after which database query operations (on ODBC data sources only) will time out. Affects all subsequent open, add new, update, and delete operations.

Operations

Close	Closes the database connection.
Create	Creates the underlying DAO database object and initializes the **CDaoDatabase** object.
CreateRelation	Defines a new relation among the tables in the database.
DeleteQueryDef	Deletes a querydef object saved in the database's QueryDefs collection.
DeleteRelation	Deletes an existing relation between tables in the database.
DeleteTableDef	Deletes the definition of a table in the database. This deletes the actual table and all of its data.
Execute	Executes an action query. Calling **Execute** for a query that returns results throws an exception.
GetQueryDefCount	Returns the number of queries defined for the database.

GetQueryDefInfo	Returns information about a specified query defined in the database.
GetRelationCount	Returns the number of relations defined between tables in the database.
GetRelationInfo	Returns information about a specified relation defined between tables in the database.
GetTableDefCount	Returns the number of tables defined in the database.
GetTableDefInfo	Returns information about a specified table in the database.
Open	Establishes a connection to a database.

See Also

Reference

CDaoDatabase Class

Hierarchy Chart

CDaoDatabase Member Functions

For information about the member functions in **CDaoDatabase**, see CDaoDatabase Members.

CDaoDatabase::CanTransact

Call this member function to determine whether the database allows transactions.

```
BOOL CanTransact( );
```

Return Value

Nonzero if the database supports transactions; otherwise 0.

Remarks

Transactions are managed in the database's workspace.

See Also

Reference

CDaoDatabase Class
Hierarchy Chart
CDaoWorkspace::BeginTrans
CDaoWorkspace::CommitTrans
CDaoWorkspace::Rollback

Other Resources

CDaoDatabase Members

CDaoDatabase::CanUpdate

Call this member function to determine whether the **CDaoDatabase** object allows updates.

```
BOOL CanUpdate( );
```

Return Value

Nonzero if the **CDaoDatabase** object allows updates; otherwise 0, indicating either that you passed **TRUE** in *bReadOnly* when you opened the **CDaoDatabase** object or that the database itself is read-only. See the Open member function.

Remarks

For information about database updatability, see the topic "Updatable Property" in DAO Help.

See Also

Reference

CDaoDatabase Class

Hierarchy Chart

Other Resources

CDaoDatabase Members

CDaoDatabase::CDaoDatabase

Constructs a **CDaoDatabase** object.

```
CDaoDatabase(
  CDaoWorkspace* pWorkspace = NULL
);
```

Parameters

pWorkspace

A pointer to the **CDaoWorkspace** object that will contain the new database object. If you accept the default value of **NULL**, the constructor creates a temporary **CDaoWorkspace** object that uses the default DAO workspace. You can get a pointer to the workspace object via the m_pWorkspace data member.

Remarks

After constructing the object, if you are creating a new Microsoft Jet (.MDB) database, call the object's Create member function. If you are, instead, opening an existing database, call the object's Open member function.

When you finish with the object, you should call its Close member function and then destroy the **CDaoDatabase** object.

You might find it convenient to embed the **CDaoDatabase** object in your document class.

Note
A **CDaoDatabase** object is also created implicitly if you open a CDaoRecordset object without passing a pointer to an existing **CDaoDatabase** object. This database object is closed when you close the recordset object.

See Also

Reference

CDaoDatabase Class

Hierarchy Chart

Other Resources

CDaoDatabase Members

CDaoDatabase::Close

Call this member function to disconnect from a database and close any open recordsets, tabledefs, and querydefs associated with the database.

```
virtual void Close( );
```

Remarks

It is good practice to close these objects yourself before you call this member function. Closing a **CDaoDatabase** object removes it from the Databases collection in the associated workspace. Because **Close** does not destroy the **CDaoDatabase** object, you can reuse the object by opening the same database or a different database.

Caution

Call the Update member function (if there are pending edits) and the **Close** member function on all open recordset objects b efore you close a database. If you exit a function that declares CDaoRecordset or **CDaoDatabase** objects on the stack, the da tabase is closed, any unsaved changes are lost, all pending transactions are rolled back, and any pending edits to your data ar e lost.

Caution

If you try to close a database object while any recordset objects are open, or if you try to close a workspace object while any database objects belonging to that specific workspace are open, those recordset objects will be closed and any pending upda tes or edits will be rolled back. If you try to close a workspace object while any database objects belonging to it are open, the operation closes all database objects belonging to that specific workspace object, which may result in unclosed recordset obj ects being closed. If you do not close your database object, MFC reports an assertion failure in debug builds.

If the database object is defined outside the scope of a function, and you exit the function without closing it, the database object will remain open until explicitly closed or the module in which it is defined is out of scope.

See Also

Reference

CDaoDatabase Class
Hierarchy Chart
CDaoDatabase::Open
CDaoRecordset::Close
CDaoWorkspace::Close
CDaoQueryDef::Close
CDaoTableDef::Close

Other Resources

CDaoDatabase Members

CDaoDatabase::Create

To create a new Microsoft Jet (.MDB) database, call this member function after you construct a **CDaoDatabase** object.

```
virtual void Create(
   LPCTSTR lpszName,
   LPCTSTR lpszLocale = dbLangGeneral,
   int dwOptions = 0
);
```

Parameters

lpszName

A string expression that is the name of the database file that you are creating. It can be the full path and filename, such as "C:\\MYDB.MDB". You must supply a name. If you do not supply a filename extension, .MDB is appended. If your network supports the uniform naming convention (UNC), you can also specify a network path, such as "\\\\MYSERVER\\MYSHARE\\MYDIR\\MYDB". Only Microsoft Jet (.MDB) database files can be created using this member function. (Double backslashes are required in string literals because "\" is the C++ escape character.)

lpszLocale

A string expression used to specify collating order for creating the database. The default value is **dbLangGeneral**. Possible values are:

- **dbLangGeneral** English, German, French, Portuguese, Italian, and Modern Spanish
- **dbLangArabic** Arabic
- **dbLangCyrillic** Russian
- **dbLangCzech** Czech
- **dbLangDutch** Dutch
- **dbLangGreek** Greek
- **dbLangHebrew** Hebrew
- **dbLangHungarian** Hungarian
- **dbLangIcelandic** Icelandic
- **dbLangNordic** Nordic languages (Microsoft Jet database engine version 1.0 only)
- **dbLangNorwdan** Norwegian and Danish
- **dbLangPolish** Polish
- **dbLangSpanish** Traditional Spanish
- **dbLangSwedfin** Swedish and Finnish
- **dbLangTurkish** Turkish

dwOptions

An integer that indicates one or more options. Possible values are:

- **dbEncrypt** Create an encrypted database.
- **dbVersion10** Create a database with Microsoft Jet database version 1.0.
- **dbVersion11** Create a database with Microsoft Jet database version 1.1.
- **dbVersion20** Create a database with Microsoft Jet database version 2.0.
- **dbVersion30** Create a database with Microsoft Jet database version 3.0.

If you omit the encryption constant, an unencrypted database is created. You can specify only one version constant. If you omit a version constant, a database that uses the Microsoft Jet database version 3.0 is created.

Caution
If a database is not encrypted, it is possible, even if you implement user/password security, to directly read the binary disk file that constitutes the database.

Remarks

Create creates the database file and the underlying DAO database object and initializes the C++ object. The object is appended to the associated workspace's Databases collection. The database object is in an open state; do not call **Open** after **Create**.

Note
With **Create**, you can create only Microsoft Jet (.MDB) databases. You cannot create ISAM databases or ODBC databases.

See Also

Reference

CDaoDatabase Class

Hierarchy Chart

CDaoDatabase::CDaoDatabase

Other Resources

CDaoDatabase Members

CDaoDatabase::CreateRelation

Call this member function to establish a relation between one or more fields in a primary table in the database and one or more fields in a foreign table (another table in the database).

```
void CreateRelation(
   LPCTSTR lpszName,
   LPCTSTR lpszTable,
   LPCTSTR lpszForeignTable,
   long lAttributes,
   LPCTSTR lpszField,
   LPCTSTR lpszForeignField
);
void CreateRelation(
   CDaoRelationInfo& relinfo
);
```

Parameters

lpszName

The unique name of the relation object. The name must start with a letter and can contain a maximum of 40 characters. It can include numbers and underscore characters but cannot include punctuation or spaces.

lpszTable

The name of the primary table in the relation. If the table does not exist, MFC throws an exception of type CDaoException.

lpszForeignTable

The name of the foreign table in the relation. If the table does not exist, MFC throws an exception of type **CDaoException**.

lAttributes

A long value that contains information about the relationship type. You can use this value to enforce referential integrity, among other things. You can use the bitwise-OR operator (**|**) to combine any of the following values (as long as the combination makes sense):

- **dbRelationUnique** Relationship is one-to-one.
- **dbRelationDontEnforce** Relationship is not enforced (no referential integrity).
- **dbRelationInherited** Relationship exists in a noncurrent database that contains the two attached tables.
- **dbRelationUpdateCascade** Updates will cascade (for more on cascades, see Remarks).
- **dbRelationDeleteCascade** Deletions will cascade.

lpszField

A pointer to a null-terminated string containing the name of a field in the primary table (named by *lpszTable*).

lpszForeignField

A pointer to a null-terminated string containing the name of a field in the foreign table (named by *lpszForeignTable*).

relinfo

A reference to a CDaoRelationInfo object that contains information about the relation you want to create.

Remarks

The relationship cannot involve a query or an attached table from an external database.

Use the first version of the function when the relation involves one field in each of the two tables. Use the second version when the relation involves multiple fields. The maximum number of fields in a relation is 14.

This action creates an underlying DAO relation object, but this is an MFC implementation detail since MFC's encapsulation of relation objects is contained within class **CDaoDatabase**. MFC does not supply a class for relations.

If you set the relation object's attributes to activate cascade operations, the database engine automatically updates or deletes records in one or more other tables when changes are made to related primary key tables.

For example, suppose you establish a cascade delete relationship between a Customers table and an Orders table. When you delete records from the Customers table, records in the Orders table related to that customer are also deleted. In addition, if you establish cascade delete relationships between the Orders table and other tables, records from those tables are automatically deleted when you delete records from the Customers table.

For related information, see the topic "CreateRelation Method" in DAO Help.

See Also

Reference

CDaoDatabase Class
Hierarchy Chart
CDaoDatabase::DeleteRelation

Other Resources

CDaoDatabase Members

CDaoDatabase::DeleteQueryDef

Call this member function to delete the specified querydef — saved query — from the **CDaoDatabase** object's QueryDefs collection.

```
void DeleteQueryDef(
   LPCTSTR lpszName
);
```

Parameters

lpszName

The name of the saved query to delete.

Remarks

Afterwards, that query is no longer defined in the database.

For information about creating querydef objects, see class CDaoQueryDef. A querydef object becomes associated with a particular **CDaoDatabase** object when you construct the **CDaoQueryDef** object, passing it a pointer to the database object.

See Also

Reference

CDaoDatabase Class
Hierarchy Chart
CDaoQueryDef::Create
CDaoDatabase::CreateRelation
CDaoTableDef::Create

Other Resources

CDaoDatabase Members

CDaoDatabase::DeleteRelation

Call this member function to delete an existing relation from the database object's Relations collection.

```
void DeleteRelation(
   LPCTSTR lpszName
);
```

Parameters

lpszName

The name of the relation to delete.

Remarks

Afterwards, the relation no longer exists.

For related information, see the topic "Delete Method" in DAO Help.

See Also

Reference

CDaoDatabase Class
Hierarchy Chart
CDaoDatabase::CreateRelation
CDaoTableDef::Create
CDaoQueryDef::Create

Other Resources

CDaoDatabase Members

CDaoDatabase::DeleteTableDef

Call this member function to delete the specified table and all of its data from the **CDaoDatabase** object's TableDefs collection.

```
void DeleteTableDef(
   LPCTSTR lpszName
);
```

Parameters

lpszName

The name of the tabledef to delete.

Remarks

Afterwards, that table is no longer defined in the database.

Note
Be very careful not to delete system tables.

For information about creating tabledef objects, see class CDaoTableDef. A tabledef object becomes associated with a particular **CDaoDatabase** object when you construct the **CDaoTableDef** object, passing it a pointer to the database object.

For related information, see the topic "Delete Method" in DAO Help.

See Also

Reference

CDaoDatabase Class
Hierarchy Chart
CDaoTableDef::Create
CDaoQueryDef::Create
CDaoDatabase::CreateRelation

Other Resources

CDaoDatabase Members

CDaoDatabase::Execute

Call this member function to run an action query or execute a SQL statement on the database.

```
void Execute(
   LPCTSTR lpszSQL,
   int nOptions = dbFailOnError
);
```

Parameters

lpszSQL

Pointer to a null-terminated string containing a valid SQL command to execute.

nOptions

An integer that specifies options relating to the integrity of the query. You can use the bitwise-OR operator (**|**) to combine any of the following constants (provided the combination makes sense — for example, you would not combine **dbInconsistent** with **dbConsistent**):

- **dbDenyWrite** Deny write permission to other users.
- **dbInconsistent** (Default) Inconsistent updates.
- **dbConsistent** Consistent updates.
- **dbSQLPassThrough** SQL pass-through. Causes the SQL statement to be passed to an ODBC data source for processing.
- **dbFailOnError** Roll back updates if an error occurs.
- **dbSeeChanges** Generate a run-time error if another user is changing data you are editing.

Note

If both **dbInconsistent** and **dbConsistent** are included or if neither is included, the result is the default. For an explanation of these constants, see the topic "Execute Method" in DAO Help.

Remarks

Execute works only for action queries or SQL pass-through queries that do not return results. It does not work for select queries, which return records.

For a definition and information about action queries, see the topics "Action Query" and "Execute Method" in DAO Help.

Tip

Given a syntactically correct SQL statement and proper permissions, the **Execute** member function will not fail even if not a single row can be modified or deleted. Therefore, always use the **dbFailOnError** option when using the **Execute** member function to run an update or delete query. This option causes MFC to throw an exception of type CDaoException and rolls back all successful changes if any of the records affected are locked and cannot be updated or deleted. Note that you can always call **GetRecordsAffected** to see how many records were affected.

Call the GetRecordsAffected member function of the database object to determine the number of records affected by the most recent **Execute** call. For example, **GetRecordsAffected** returns information about the number of records deleted, updated, or inserted when executing an action query. The count returned will not reflect changes in related tables when cascade updates or deletes are in effect.

Execute does not return a recordset. Using **Execute** on a query that selects records causes MFC to throw an exception of type **CDaoException**. (There is no **ExecuteSQL** member function analogous to **CDatabase::ExecuteSQL**.)

See Also

Reference

CDaoDatabase Class

Hierarchy Chart

Other Resources

CDaoDatabase Members

CDaoDatabase::GetConnect

Call this member function to retrieve the connection string used to connect the **CDaoDatabase** object to an ODBC or ISAM database.

```
CString GetConnect( );
```

Return Value

The connection string if Open has been called successfully on an ODBC data source; otherwise, an empty string. For a Microsoft Jet (.MDB) database, the string is always empty unless you set it for use with the **dbSQLPassThrough** option used with the Execute member function or used in opening a recordset.

Remarks

The string provides information about the source of an open database or a database used in a pass-through query. The connection string is composed of a database type specifier and zero or more parameters separated by semicolons.

Note
Using the MFC DAO classes to connect to a data source via ODBC is less efficient than connecting via an attached table.

Note
The connection string is used to pass additional information to ODBC and certain ISAM drivers as needed. It is not used for .MDB databases. For Microsoft Jet database base tables, the connection string is an empty string ("") except when you use it for a SQL pass-through query as described under Return Value above.

See the Open member function for a description of how the connection string is created. Once the connection string has been set in the **Open** call, you can later use it to check the setting to determine the type, path, user ID, Password, or ODBC data source of the database.

See Also

Reference

CDaoDatabase Class

Hierarchy Chart

Other Resources

CDaoDatabase Members

CDaoDatabase::GetName

Call this member function to retrieve the name of the currently open database, which is the name of an existing database file or registered ODBC data source name.

```
CString GetName( );
```

Return Value

The full path and filename for the database if successful; otherwise, an empty CString.

Remarks

If your network supports the uniform naming convention (UNC), you can also specify a network path, such as "\\\\MYSERVER\\MYSHARE\\MYDIR\\MYDB.MDB". (Double backslashes are required in string literals because "\" is the C++ escape character.)

You might, for example, want to display this name in a heading. If an error occurs while retrieving the name, MFC throws an exception of type CDaoException.

Note
For better performance when accessing external databases, it is recommended that you attach external database tables to a Microsoft Jet engine database (.MDB) rather than connecting directly to the data source.

The database type is indicated by the file or directory that the path points to, as follows:

Pathname points to..	**Database type**
.MDB file	Microsoft Jet database (Microsoft Access)
Directory containing .DBF file(s)	dBASE® database
Directory containing .XLS file	Microsoft Excel database
Directory containing .PDX file(s)	Paradox® database
Directory containing appropriately formatted text database files	Text format database

For ODBC databases, such as Microsoft SQL Server and Oracle®, the database's connection string identifies a data source name (DSN) registered by ODBC.

See Also

Reference

CDaoDatabase Class
Hierarchy Chart
CDatabase::Open
CDatabase::GetConnect

Other Resources

CDaoDatabase Members

CDaoDatabase::GetQueryDefCount

Call this member function to retrieve the number of queries defined in the database's QueryDefs collection.

```
short GetQueryDefCount( );
```

Return Value

The number of queries defined in the database.

Remarks

GetQueryDefCount is useful if you need to loop through all querydefs in the QueryDefs collection. To obtain information about a given query in the collection, see GetQueryDefInfo.

See Also

Reference

CDaoDatabase Class

Hierarchy Chart

Other Resources

CDaoDatabase Members

CDaoDatabase::GetQueryDefInfo

Call this member function to obtain various kinds of information about a query defined in the database.

```
void GetQueryDefInfo(
   int nIndex,
   CDaoQueryDefInfo& querydefinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
void GetQueryDefInfo(
   LPCTSTR lpszName,
   CDaoQueryDefInfo& querydefinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
```

Parameters

nIndex

The index of the predefined query in the database's QueryDefs collection, for lookup by index.

querydefinfo

A reference to a CDaoQueryDefInfo object that returns the information requested.

dwInfoOptions

Options that specify which information about the recordset to retrieve. The available options are listed here along with what they cause the function to return about the recordset:

- **AFX_DAO_PRIMARY_INFO** (Default) Name, Type
- **AFX_DAO_SECONDARY_INFO** Primary information plus: Date Created, Date of Last Update, Returns Records, Updatable
- **AFX_DAO_ALL_INFO** Primary and secondary information plus: SQL, Connect, ODBCTimeout

lpszName

A string containing the name of a query defined in the database, for lookup by name.

Remarks

Two versions of the function are supplied so you can select a query either by index in the database's QueryDefs collection or by the name of the query.

For a description of the information returned in *querydefinfo*, see the CDaoQueryDefInfo structure. This structure has members that correspond to the items of information listed above in the description of *dwInfoOptions*. If you request one level of information, you get any prior levels of information as well.

See Also

Reference

CDaoDatabase Class
Hierarchy Chart
CDaoDatabase::GetQueryDefCount

Other Resources

CDaoDatabase Members

CDaoDatabase::GetQueryTimeout

Call this member function to retrieve the current number of seconds to allow before subsequent operations on the connected database are timed out.

```
short GetQueryTimeout( );
```

Return Value

A short integer containing the timeout value in seconds.

Remarks

An operation might time out due to network access problems, excessive query processing time, and so on. While the setting is in effect, it affects all open, add new, update, and delete operations on any recordsets associated with this **CDaoDatabase** object. You can change the current timeout setting by calling SetQueryTimeout. Changing the query timeout value for a recordset after opening does not change the value for the recordset. For example, subsequent Move operations do not use the new value. The default value is initially set when the database engine is initialized.

The default value for query timeouts is taken from the Windows registry. If there is no registry setting, the default is 60 seconds. Not all databases support the ability to set a query timeout value. If you set a query timeout value of 0, no timeout occurs; and communication with the database may hang. This behavior may be useful during development. If the call fails, MFC throws an exception of type CDaoException.

For related information, see the topic "QueryTimeout Property" in DAO Help.

See Also

Reference

CDaoDatabase Class
Hierarchy Chart
CDaoWorkspace::SetLoginTimeout

Other Resources

CDaoDatabase Members

CDaoDatabase::GetRecordsAffected

Call this member function to determine the number of records affected by the most recent call of the Execute member function.

```
long GetRecordsAffected( );
```

Return Value

A long integer containing the number of records affected.

Remarks

The value returned includes the number of records deleted, updated, or inserted by an action query run with **Execute**. The count returned will not reflect changes in related tables when cascade updates or deletes are in effect.

For related information, see the topic "RecordsAffected Property" in DAO Help.

See Also

Reference

CDaoDatabase Class

Hierarchy Chart

Other Resources

CDaoDatabase Members

CDaoDatabase::GetRelationCount

Call this member function to obtain the number of relations defined between tables in the database.

```
short GetRelationCount( );
```

Return Value

The number of relations defined between tables in the database.

Remarks

GetRelationCount is useful if you need to loop through all defined relations in the database's Relations collection. To obtain information about a given relation in the collection, see GetRelationInfo.

To illustrate the concept of a relation, consider a Suppliers table and a Products table, which might have a one-to-many relationship. In this relationship, one supplier can supply more than one product. Other relations are one-to-one and many-to-many.

See Also

Reference

CDaoDatabase Class

Hierarchy Chart

Other Resources

CDaoDatabase Members

CDaoDatabase::GetRelationInfo

Call this member function to obtain information about a specified relation in the database's Relations collection.

```
void GetRelationInfo(
   int nIndex,
   CDaoRelationInfo& relinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
void GetRelationInfo(
   LPCTSTR lpszName,
   CDaoRelationInfo& relinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
```

Parameters

nIndex

The index of the relation object in the database's Relations collection, for lookup by index.

relinfo

A reference to a CDaoRelationInfo object that returns the information requested.

dwInfoOptions

Options that specify which information about the relation to retrieve. The available options are listed here along with what they cause the function to return about the relation:

- **AFX_DAO_PRIMARY_INFO** (Default) Name, Table, Foreign Table
- **AFX_DAO_SECONDARY_INFO** Attributes, Field Information

The Field Information is a CDaoRelationFieldInfo object containing the fields from the primary table involved in the relation.

lpszName

A string containing the name of the relation object, for lookup by name.

Remarks

Two versions of this function provide access either by index or by name. For a description of the information returned in *relinfo*, see the CDaoRelationInfo structure. This structure has members that correspond to the items of information listed above in the description of *dwInfoOptions*. If you request information at one level, you also get information at any prior levels as well.

Note

If you set the relation object's attributes to activate cascade operations (**dbRelationUpdateCascades** or **dbRelationDeleteCascades**), the Microsoft Jet database engine automatically updates or deletes records in one or more other tables when changes are made to related primary key tables. For example, suppose you establish a cascade delete relationship between a Customers table and an Orders table. When you delete records from the Customers table, records in the Orders table related to that customer are also deleted. In addition, if you establish cascade delete relationships between the Orders table and other tables, records from those tables are automatically deleted when you delete records from the Customers table.

See Also

Reference

CDaoDatabase Class
Hierarchy Chart
CDaoDatabase::GetRelationCount

Other Resources

CDaoDatabase Members

CDaoDatabase::GetTableDefCount

Call this member function to retrieve the number of tables defined in the database.

```
short GetTableDefCount( );
```

Return Value

The number of tabledefs defined in the database.

Remarks

GetTableDefCount is useful if you need to loop through all tabledefs in the database's TableDefs collection. To obtain information about a given table in the collection, see GetTableDefInfo.

See Also

Reference

CDaoDatabase Class

Hierarchy Chart

Other Resources

CDaoDatabase Members

CDaoDatabase::GetTableDefInfo

Call this member function to obtain various kinds of information about a table defined in the database.

```
void GetTableDefInfo(
   int nIndex,
   CDaoTableDefInfo& tabledefinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
void GetTableDefInfo(
   LPCTSTR lpszName,
   CDaoTableDefInfo& tabledefinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
```

Parameters

nIndex

The index of the tabledef object in the database's TableDefs collection, for lookup by index.

tabledefinfo

A reference to a CDaoTableDefInfo object that returns the information requested.

dwInfoOptions

Options that specify which information about the table to retrieve. The available options are listed here along with what they cause the function to return about the relation:

- **AFX_DAO_PRIMARY_INFO** (Default) Name, Updatable, Attributes
- **AFX_DAO_SECONDARY_INFO** Primary information plus: Date Created, Date Last Updated, Source Table Name, Connect
- **AFX_DAO_ALL_INFO** Primary and secondary information plus: Validation Rule, Validation Text, Record Count

lpszName

The name of the tabledef object, for lookup by name.

Remarks

Two versions of the function are supplied so you can select a table either by index in the database's TableDefs collection or by the name of the table.

For a description of the information returned in *tabledefinfo*, see the CDaoTableDefInfo structure. This structure has members that correspond to the items of information listed above in the description of *dwInfoOptions*. If you request information at one level, you get information for any prior levels as well.

Note
The **AFX_DAO_ALL_INFO** option provides information that can be slow to obtain. In this case, counting the records in the table could be very time consuming if there are many records.

See Also

Reference

CDaoDatabase Class
Hierarchy Chart
CDaoDatabase::GetTableDefCount

Other Resources

CDaoDatabase Members

CDaoDatabase::GetVersion

Call this member function to determine the version of the Microsoft Jet database file.

```
CString GetVersion( );
```

Return Value

A CString that indicates the version of the database file associated with the object.

Remarks

The value returned represents the version number in the form "major.minor"; for example, "3.0". The product version number (for example, 3.0) consists of the version number (3), a period, and the release number (0). The versions to date are 1.0, 1.1, 2.0, and 3.0.

For related information, see the topic "Version Property" in DAO Help.

See Also

Reference

CDaoDatabase Class

Hierarchy Chart

Other Resources

CDaoDatabase Members

CDaoDatabase::IsOpen

Call this member function to determine whether the **CDaoDatabase** object is currently open on a database.

```
BOOL IsOpen( ) const;
```

Return Value

Nonzero if the **CDaoDatabase** object is currently open; otherwise 0.

Remarks

See Also

Reference

CDaoDatabase Class

Hierarchy Chart

CDatabase::Open

Other Resources

CDaoDatabase Members

CDaoDatabase::Open

You must call this member function to initialize a newly constructed **CDaoDatabase** object that represents an existing database.

```
virtual void Open(
   LPCTSTR lpszName,
   BOOL bExclusive = FALSE,
   BOOL bReadOnly = FALSE,
   LPCTSTR lpszConnect = _T(
   "" )
);
```

Parameters

lpszName

A string expression that is the name of an existing Microsoft Jet (.MDB) database file. If the filename has an extension, it is required. If your network supports the uniform naming convention (UNC), you can also specify a network path, such as "\\\\MYSERVER\\MYSHARE\\MYDIR\\MYDB.MDB". (Double backslashes are required in string literals because "\" is the C++ escape character.)

Some considerations apply when using *lpszName*. If it:

- Refers to a database that is already open for exclusive access by another user, MFC throws an exception of type CDaoException. Trap that exception to let your user know that the database is unavailable.
- Is an empty string ("") and *lpszConnect* is "ODBC;", a dialog box listing all registered ODBC data source names is displayed so the user can select a database. You should avoid direct connections to ODBC data sources; use an attached table instead.
- Otherwise does not refer to an existing database or valid ODBC data source name, MFC throws an exception of type **CDaoException**.

Note
For details about DAO error codes, see the DAOERR.H file. For related information, see the topic "Trappable Data Access Errors" in DAO Help.

bExclusive

A Boolean value that is **TRUE** if the database is to be opened for exclusive (nonshared) access and **FALSE** if the database is to be opened for shared access. If you omit this argument, the database is opened for shared access.

bReadOnly

A Boolean value that is **TRUE** if the database is to be opened for read-only access and **FALSE** if the database is to be opened for read/write access. If you omit this argument, the database is opened for read/write access. All dependent recordsets inherit this attribute.

lpszConnect

A string expression used for opening the database. This string constitutes the ODBC connect arguments. You must supply the exclusive and read-only arguments to supply a source string. If the database is a Microsoft Jet database (.MDB), this string is empty ("). The syntax for the default value — **_T**("") — provides portability for Unicode as well as ANSI builds of your application.

Remarks

Open associates the database with the underlying DAO object. You cannot use the database object to construct recordset, tabledef, or querydef objects until it is initialized. **Open** appends the database object to the associated workspace's Databases collection.

Use the parameters as follows:

- If you are opening a Microsoft Jet (.MDB) database, use the *lpszName* parameter and pass an empty string for the *lpszConnect* parameter or pass a password string of the form ";PWD=password" if the database is password-protected (.MDB databases only).
- If you are opening an ODBC data source, pass a valid ODBC connection string in *lpszConnect* and an empty string in *lpszName*.

For related information, see the topic "OpenDatabase Method" in DAO Help.

Note
For better performance when accessing external databases, including ISAM databases and ODBC data sources, it is recommended that you attach external database tables to a Microsoft Jet engine database (.MDB) rather than connecting directly to the data source.

It is possible for a connection attempt to time out if, for example, the DBMS host is unavailable. If the connection attempt fails, **Open** throws an exception of type CDaoException.

The remaining remarks apply only to ODBC databases:

If the database is an ODBC database and the parameters in your **Open** call do not contain enough information to make the connection, the ODBC driver opens a dialog box to obtain the necessary information from the user. When you call **Open**, your connection string, *lpszConnect*, is stored privately and is available by calling the GetConnect member function.

If you wish, you can open your own dialog box before you call **Open** to get information from the user, such as a password, then add that information to the connection string you pass to **Open**. Or you might want to save the connection string you pass (perhaps in the Windows registry) so you can reuse it the next time your application calls **Open** on a **CDaoDatabase** object.

You can also use the connection string for multiple levels of login authorization (each for a different **CDaoDatabase** object) or to convey other database-specific information.

See Also

Reference

CDaoDatabase Class
Hierarchy Chart
CDatabase::CDatabase
CDatabase::Close

Other Resources

CDaoDatabase Members

CDaoDatabase::SetQueryTimeout

Call this member function to override the default number of seconds to allow before subsequent operations on the connected database time out.

```
void SetQueryTimeout(
   short nSeconds
);
```

Parameters

nSeconds

The number of seconds to allow before a query attempt times out.

Remarks

An operation might time out because of network access problems, excessive query processing time, and so on. Call **SetQueryTimeout** before opening your recordset or before calling the recordset's AddNew, Update, or Delete member functions if you want to change the query timeout value. The setting affects all subsequent Open, **AddNew**, **Update**, and **Delete** calls to any recordsets associated with this **CDaoDatabase** object. Changing the query timeout value for a recordset after opening does not change the value for the recordset. For example, subsequent Move operations do not use the new value.

The default value for query timeouts is 60 seconds. Not all databases support the ability to set a query timeout value. If you set a query timeout value of 0, no timeout occurs; the communication with the database may stop responding. This behavior may be useful during development.

For related information, see the topic "QueryTimeout Property" in DAO Help.

See Also

Reference

CDaoDatabase Class
Hierarchy Chart
CDaoWorkspace::SetLoginTimeout

Other Resources

CDaoDatabase Members

CDaoDatabase Data Members

For information about the data members in **CDaoDatabase**, see CDaoDatabase Members.

CDaoDatabase::m_pDAODatabase

Contains a pointer to the OLE interface for the DAO database object underlying the **CDaoDatabase** object.

Remarks

Use this pointer if you need to access the DAO interface directly.

For information about calling DAO directly, see Technical Note 54.

See Also

Reference

CDaoDatabase Class

Hierarchy Chart

Other Resources

CDaoDatabase Members

CDaoDatabase::m_pWorkspace

Contains a pointer to the CDaoWorkspace object that contains the database object.

Remarks

Use this pointer if you need to access the workspace directly — for example, to obtain pointers to other database objects in the workspace's Databases collection.

See Also

Reference

CDaoDatabase Class

Hierarchy Chart

Other Resources

CDaoDatabase Members

CDaoException Class

Represents an exception condition arising from the MFC database classes based on data access objects (DAO).

```
class CDaoException : public CException
```

Remarks

The class includes public data members you can use to determine the cause of the exception. **CDaoException** objects are constructed and thrown by member functions of the DAO database classes.

Note

The DAO database classes are distinct from the MFC database classes based on Open Database Connectivity (ODBC). All DAO database class names have the "CDao" prefix. You can still access ODBC data sources with the DAO classes. In general, the MFC classes based on DAO are more capable than the MFC classes based on ODBC; the DAO-based classes can access data, including through ODBC drivers, via their own database engine. The DAO-based classes also support Data Definition Language (DDL) operations, such as adding tables via the classes, without having to call DAO directly. For information on exceptions thrown by the ODBC classes, see CDBException.

You can access exception objects within the scope of a CATCH expression. You can also throw **CDaoException** objects from your own code with the AfxThrowDaoException global function.

In MFC, all DAO errors are expressed as exceptions, of type **CDaoException**. When you catch an exception of this type, you can use **CDaoException** member functions to retrieve information from any DAO error objects stored in the database engine's Errors collection. As each error occurs, one or more error objects are placed in the Errors collection. (Normally the collection contains only one error object; if you are using an ODBC data source, you are more likely to get multiple error objects.) When another DAO operation generates an error, the Errors collection is cleared, and the new error object is placed in the Errors collection. DAO operations that do not generate an error have no effect on the Errors collection.

For DAO error codes, see the file DAOERR.H. For related information, see the topic "Trappable Data Access Errors" in DAO Help.

For more information about exception handling in general, or about **CDaoException** objects, see the articles Exception Handling (MFC) and Exceptions: Database Exceptions. The second article contains example code that illustrates exception handling in DAO.

Requirements

Header: afxdao.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CException Class

Hierarchy Chart

CException Class

Other Resources

CDaoException Members

CDaoException Members

Base Class Members

CObject Members

CException Members

Data Members

m_nAfxDaoError	Contains an extended error code for any error in the MFC DAO classes.
m_pErrorInfo	A pointer to a CDaoErrorInfo object that contains information about one DAO error object.
m_scode	The SCODE value associated with the error.

Construction

CDaoException	Constructs a **CDaoException** object.

Operations

GetErrorCount	Returns the number of errors in the database engine's Errors collection.
GetErrorInfo	Returns error information about a particular error object in the Errors collection.

See Also

Reference

CDaoException Class
Hierarchy Chart

CDaoException Member Functions

For information about the member functions in **CDaoException**, see CDaoException Members.

CDaoException::CDaoException

Constructs a **CDaoException** object.

```
CDaoException( );
```

Remarks

Ordinarily, the framework creates exception objects when its code throws an exception. You seldom need to construct an exception object explicitly. If you want to throw a **CDaoException** from your own code, call the global function AfxThrowDaoException.

However, you might want to explicitly create an exception object if you are making direct calls to DAO via the DAO interface pointers that MFC classes encapsulate. In that case, you might need to retrieve error information from DAO. Suppose an error occurs in DAO when you call a DAO method via the DAODatabases interface to a workspace's Databases collection.

To retrieve the DAO error information

1. Construct a **CDaoException** object.
2. Call the exception object's GetErrorCount member function to determine how many error objects are in the database engine's Errors collection. (Normally only one, unless you are using an ODBC data source.)
3. Call the exception object's GetErrorInfo member function to retrieve one specific error object at a time, by index in the collection, via the exception object. Think of the exception object as a proxy for one DAO error object.
4. Examine the current CDaoErrorInfo structure that **GetErrorInfo** returns in the m_pErrorInfo data member. Its members provide information on the DAO error.
5. In the case of an ODBC data source, repeat steps 3 and 4 as needed, for more error objects.
6. If you constructed the exception object on the heap, delete it with the **delete** operator when you finish.

For more information about handling errors in the MFC DAO classes, see the article Exceptions: Database Exceptions.

See Also

Reference

CDaoException Class

Hierarchy Chart

Other Resources

CDaoException Members

CDaoException::GetErrorCount

Call this member function to retrieve the number of DAO error objects in the database engine's Errors collection.

```
short GetErrorCount( );
```

Return Value

The number of DAO error objects in the database engine's Errors collection.

Remarks

This information is useful for looping through the Errors collection to retrieve each of the one or more DAO error objects in the collection. To retrieve an error object by index or by DAO error number, call the GetErrorInfo member function.

Note
Normally there is only one error object in the Errors collection. If you are working with an ODBC data source, however, there could be more than one.

See Also

Reference

CDaoException Class

Hierarchy Chart

Other Resources

CDaoException Members

CDaoException::GetErrorInfo

Returns error information about a particular error object in the Errors collection.

```
void GetErrorInfo(
   int nIndex
);
```

Parameters

nIndex

The index of the error information in the database engine's Errors collection, for lookup by index.

Remarks

Call this member function to obtain the following kinds of information about the exception:

- Error code
- Source
- Description
- Help file
- Help context

GetErrorInfo stores the information in the exception object's **m_pErrorInfo** data member. For a brief description of the information returned, see m_pErrorInfo. If you catch an exception of type **CDaoException** thrown by MFC, the **m_pErrorInfo** member will already be filled in. If you choose to call DAO directly, you must call the exception object's **GetErrorInfo** member function yourself to fill **m_pErrorInfo**. For a more detailed description, see the CDaoErrorInfo structure.

For information about DAO exceptions, and example code, see the article Exceptions: Database Exceptions.

See Also

Reference

CDaoException Class
Hierarchy Chart
CDaoException::GetErrorCount

Other Resources

CDaoException Members

CDaoException Data Members

For information about the data members in **CDaoException**, see CDaoException Members.

CDaoException::m_nAfxDaoError

Contains an MFC extended error code.

Remarks

This code is supplied in cases where a specific component of the MFC DAO classes has erred.

Possible values are:

- **NO_AFX_DAO_ERROR** The most recent operation did not result in an MFC extended error. However, the operation could have produced other errors from DAO or OLE, so you should check m_pErrorInfo and possibly m_scode.
- **AFX_DAO_ERROR_ENGINE_INITIALIZATION** MFC could not initialize the Microsoft Jet database engine. OLE might have failed to initialize, or it might have been impossible to create an instance of the DAO database engine object. These problems usually suggest a bad installation of either DAO or OLE.
- **AFX_DAO_ERROR_DFX_BIND** An address used in a DAO record field exchange (DFX) function call does not exist or is invalid (the address was not used to bind data). You might have passed a bad address in a DFX call, or the address might have become invalid between DFX operations.
- **AFX_DAO_ERROR_OBJECT_NOT_OPEN** You attempted to open a recordset based on a querydef or a tabledef object that was not in an open state.

See Also

Reference

CDaoException Class
Hierarchy Chart
CDaoException::GetErrorCount
CDaoException::GetErrorInfo

Other Resources

CDaoException Members

CDaoException::m_pErrorInfo

Contains a pointer to a **CDaoErrorInfo** structure that provides information on the DAO error object that you last retrieved by calling GetErrorInfo.

Remarks

This object contains the following information:

CDaoErrorInfo member	Information	Meaning
m_lErrorCode	Error Code	The DAO error code
m_strSource	Source	The name of the object or application that originally generated the error
m_strDescription	Description	A descriptive string associated with the error
m_strHelpFile	Help File	A path to a Windows Help file in which the user can get information about the problem
m_lHelpContext	Help Context	The context ID for a topic in the DAO Help file

For full details about the information contained in the **CDaoErrorInfo** object, see the CDaoErrorInfo structure.

See Also

Reference

CDaoException Class
Hierarchy Chart
CDaoException::m_scode
CDaoException::m_nAfxDaoError

Other Resources

CDaoException Members

CDaoException::m_scode

Contains a value of type **SCODE** that describes the error.

Remarks

This is an OLE code. You will seldom need to use this value because, in almost all cases, more specific MFC or DAO error information is available in the other **CDaoException** data members.

For information about **SCODE**, see the topic Structure of OLE Error Codes in the *Platform SDK*. The **SCODE** data type maps to the **HRESULT** data type.

See Also

Reference

CDaoException Class
Hierarchy Chart
CDaoException::m_pErrorInfo
CDaoException::m_nAfxDaoError

Other Resources

CDaoException Members

CDaoFieldExchange Class

Supports the DAO record field exchange (DFX) routines used by the DAO database classes.

```
class CDaoFieldExchange
```

Remarks

CDaoFieldExchange does not have a base class.

Use this class if you are writing data exchange routines for custom data types; otherwise, you will not directly use this class. DFX exchanges data between the field data members of your CDaoRecordset object and the corresponding fields of the current record on the data source. DFX manages the exchange in both directions, from the data source and to the data source. See Technical Note 53 for information about writing custom DFX routines.

Note

The DAO database classes are distinct from the MFC database classes based on Open Database Connectivity (ODBC). All DAO database class names have the "CDao" prefix. You can still access ODBC data sources with the DAO classes. In general, the MFC classes based on DAO are more capable than the MFC classes based on ODBC. The DAO-based classes can access data, including through ODBC drivers, via their own database engine. They also support Data Definition Language (DDL) operations, such as adding tables via the classes instead of having to call DAO yourself.

Note

DAO record field exchange (DFX) is very similar to record field exchange (RFX) in the ODBC-based MFC database classes (**CDatabase**, **CRecordset**). If you understand RFX, you will find it easy to use DFX.

A **CDaoFieldExchange** object provides the context information needed for DAO record field exchange to take place. **CDaoFieldExchange** objects support a number of operations, including binding parameters and field data members and setting various flags on the fields of the current record. DFX operations are performed on recordset-class data members of types defined by the **enum FieldType** in **CDaoFieldExchange**. Possible **FieldType** values are:

- **CDaoFieldExchange::outputColumn** for field data members.
- **CDaoFieldExchange::param** for parameter data members.

The IsValidOperation member function is provided for writing your own custom DFX routines. You will use SetFieldType frequently in your CDaoRecordset::DoFieldExchange functions. For details about the DFX global functions, see Record Field Exchange Functions. For information about writing custom DFX routines for your own data types, see Technical Note 53.

Requirements

Header: afxdao.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart

CDaoRecordset Class

Other Resources

CDaoFieldExchange Members

CDaoFieldExchange Members

Data Members

m_nOperation	The DFX operation being performed by the current call to the recordset's **DoFieldExchange** member function.
m_prs	A pointer to the recordset on which DFX operations are being performed.

Member Functions

IsValidOperation	Returns nonzero if the current operation is appropriate for the type of field being updated.
SetFieldType	Specifies the type of recordset data member — column or parameter — represented by all subsequent calls to DFX functions until the next call to **SetFieldType**.

See Also

Reference

CDaoFieldExchange Class

Hierarchy Chart

CDaoFieldExchange Member Functions

For information about the member functions in **CDaoFieldExchange**, see CDaoFieldExchange Members.

CDaoFieldExchange::IsValidOperation

If you write your own DFX function, call **IsValidOperation** at the beginning of your function to determine whether the current operation can be performed on a particular field data member type (a **CDaoFieldExchange::outputColumn** or a **CDaoFieldExchange::param**).

```
BOOL IsValidOperation( );
```

Return Value

Nonzero if the current operation is appropriate for the type of field being updated.

Remarks

Some of the operations performed by the DFX mechanism apply only to one of the possible field types. Follow the model of the existing DFX functions.

For additional information on writing custom DFX routines, see Technical Note 53.

See Also

Reference

CDaoFieldExchange Class
Hierarchy Chart
CDaoFieldExchange::SetFieldType

Other Resources

CDaoFieldExchange Members

CDaoFieldExchange::SetFieldType

Call **SetFieldType** in your **CDaoRecordset** class's **DoFieldExchange** override.

```
void SetFieldType(
   UINT nFieldType
);
```

Parameters

nFieldType

A value of the **enum FieldType**, declared in **CDaoFieldExchange**, which can be either of the following:

- **CDaoFieldExchange::outputColumn**
- **CDaoFieldExchange::param**

Remarks

Normally, ClassWizard writes this call for you. If you write your own function and are using the wizard to write your **DoFieldExchange** function, add calls to your own function outside the field map. If you do not use the wizard, there will not be a field map. The call precedes calls to DFX functions, one for each field data member of your class, and identifies the field type as **CDaoFieldExchange::outputColumn**.

If you parameterize your recordset class, you should add DFX calls for all parameter data members (outside the field map) and precede these calls with a call to **SetFieldType**. Pass the value **CDaoFieldExchange::param**. (You can, instead, use a CDaoQueryDef and set its parameter values.)

In general, each group of DFX function calls associated with field data members or parameter data members must be preceded by a call to **SetFieldType**. The *nFieldType* parameter of each **SetFieldType** call identifies the type of the data members represented by the DFX function calls that follow the **SetFieldType** call.

See Also

Reference

CDaoFieldExchange Class
Hierarchy Chart
CDaoFieldExchange::IsValidOperation
CDaoRecordset::DoFieldExchange

Other Resources

CDaoFieldExchange Members

CDaoFieldExchange Data Members

For information about the data members in **CDaoFieldExchange**, see CDaoFieldExchange Members.

CDaoFieldExchange::m_nOperation

Identifies the operation to be performed on the CDaoRecordset object associated with the field exchange object.

Remarks

The **CDaoFieldExchange** object supplies the context for a number of different DFX operations on the recordset.

Note
The **PSEUDO NULL** value described under the MarkForAddNew and SetFieldNull operations below is a value used to mark fields Null. The DAO record field exchange mechanism (DFX) uses this value to determine which fields have been explicitly marked Null. **PSEUDO NULL** is not required for COleDateTime and COleCurrency fields.

Possible values of **m_nOperation** are:

Operation	Description
AddToParameterList	Builds the **PARAMETERS** clause of the SQL statement.
AddToSelectList	Builds the **SELECT** clause of the SQL statement.
BindField	Binds a field in the database to a memory location in your application.
BindParam	Sets parameter values for the recordset's query.
Fixup	Sets the Null status for a field.
AllocCache	Allocates the cache used to check for "dirty" fields in the recordset.
StoreField	Saves the current record to the cache.
LoadField	Restores the cached data member variables in the recordset.
FreeCache	Frees the cache used to check for "dirty" fields in the recordset.
SetFieldNull	Sets a field's status to Null and value to **PSEUDO NULL**.
MarkForAddNew	Marks fields "dirty" if not **PSEUDO NULL**.
MarkForEdit	Marks fields "dirty" if they do not match the cache.
SetDirtyField	Sets field values marked as "dirty."
DumpField	Dumps a field's contents (debug only).
MaxDFXOperation	Used for input checking.

See Also

Reference

CDaoFieldExchange Class
Hierarchy Chart
CDaoFieldExchange::IsValidOperation
CDaoFieldExchange::m_prs
CDaoRecordset::DoFieldExchange

Other Resources

CDaoFieldExchange Members

CDaoFieldExchange::m_prs

Contains a pointer to the CDaoRecordset object associated with the **CDaoFieldExchange** object.

Remarks

See Also

Reference

CDaoFieldExchange Class

Hierarchy Chart

CDaoFieldExchange::m_nOperation

CDaoRecordset Class

Other Resources

CDaoFieldExchange Members

CDaoQueryDef Class

Represents a query definition, or "querydef," usually one saved in a database.

```
class CDaoQueryDef : public CObject
```

Remarks

A querydef is a data access object that contains the SQL statement that describes a query, and its properties, such as "Date Created" and "ODBC Timeout." You can also create temporary querydef objects without saving them, but it is convenient — and much more efficient — to save commonly reused queries in a database. A CDaoDatabase object maintains a collection, called the QueryDefs collection, that contains its saved querydefs.

Note

The DAO database classes are distinct from the MFC database classes based on Open Database Connectivity (ODBC). All DAO database class names have the "CDao" prefix. You can still access ODBC data sources with the DAO classes. In general, the MFC classes based on DAO are more capable than the MFC classes based on ODBC; the DAO-based classes can access data, including through ODBC drivers, via their own database engine. The DAO-based classes also support Data Definition Language (DDL) operations, such as adding tables via the classes, without having to call DAO directly.

Usage

Use querydef objects either to work with an existing saved query or to create a new saved query or temporary query:

1. In all cases, first construct a **CDaoQueryDef** object, supplying a pointer to the CDaoDatabase object to which the query belongs.
2. Then do the following, depending on what you want:
   - To use an existing saved query, call the querydef object's Open member function, supplying the name of the saved query.
   - To create a new saved query, call the querydef object's Create member function, supplying the name of the query. Then call Append to save the query by appending it to the database's QueryDefs collection. **Create** puts the querydef into an open state, so after calling **Create** you do not call **Open**.
   - To create a temporary querydef, call **Create**. Pass an empty string for the query name. Do not call **Append**.

When you finish using a querydef object, call its Close member function; then destroy the querydef object.

Tip

The easiest way to create saved queries is to create them and store them in your database using Microsoft Access. Then you can open and use them in your MFC code.

Purposes

You can use a querydef object for any of the following purposes:

- To create a **CDaoRecordset** object
- To call the object's **Execute** member function to directly execute an action query or a SQL pass-through query

You can use a querydef object for any type of query, including select, action, crosstab, delete, update, append, make-table, data definition, SQL pass-through, union, and bulk queries. The query's type is determined by the content of the SQL statement that you supply. For information about query types, see the **Execute** and GetType member functions. Recordsets are commonly used for row-returning queries, usually those using the **SELECT ... FROM** keywords. **Execute** is most commonly used for bulk operations. For more information, see Execute and CDaoRecordset.

Querydefs and Recordsets

To use a querydef object to create a **CDaoRecordset** object, you typically create or open a querydef as described above. Then construct a recordset object, passing a pointer to your querydef object when you call CDaoRecordset::Open. The querydef you

pass must be in an open state. For more information, see class CDaoRecordset.

You cannot use a querydef to create a recordset (the most common use for a querydef) unless it is in an open state. Put the querydef into an open state by calling either **Open** or **Create**.

External Databases

Querydef objects are the preferred way to use the native SQL dialect of an external database engine. For example, you can create a Transact SQL query (as used on Microsoft SQL Server) and store it in a querydef object. When you need to use a SQL query not based on the Microsoft Jet database engine, you must provide a connection string that points to the external data source. Queries with valid connection strings bypass the database engine and pass the query directly to the external database server for processing.

Tip
The preferred way to work with ODBC tables is to attach them to a Microsoft Jet (.MDB) database.

For related information, see the topics "QueryDef Object", "QueryDefs Collection", and "CdbDatabase Object" in the DAO SDK.

Requirements

Header: afxdao.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class
Hierarchy Chart
CDaoRecordset Class
CDaoDatabase Class
CDaoTableDef Class
CDaoException Class

Other Resources

CDaoQueryDef Members

CDaoQueryDef Members

Base Class Members

CObject Members

CDaoQueryDef Members

Data Members

m_pDAOQueryDef	A pointer to the OLE interface for the underlying DAO querydef object.
m_pDatabase	A pointer to the **CDaoDatabase** object with which the querydef is associated. The querydef might be saved in the database or not.

Construction

Append	Appends the querydef to the database's QueryDefs collection as a saved query.
CDaoQueryDef	Constructs a **CDaoQueryDef** object. Next call **Open** or **Create**, depending on your needs.
Close	Closes the querydef object. Destroy the C++ object when you finish with it.
Create	Creates the underlying DAO querydef object. Use the querydef as a temporary query, or call **Append** to save it in the database.
Open	Opens an existing querydef stored in the database's QueryDefs collection.

Attributes

CanUpdate	Returns nonzero if the query can update the database.
GetConnect	Returns the connection string associated with the querydef. The connection string identifies the data source. (For SQL pass-through queries only; otherwise an empty string.)
GetDateCreated	Returns the date the saved query was created.
GetDateLastUpdated	Returns the date the saved query was last updated.
GetName	Returns the name of the querydef.
GetODBCTimeout	Returns the timeout value used by ODBC (for an ODBC query) when the querydef is executed. This determines how long to allow for the query's action to complete.
GetRecordsAffected	Returns the number of records affected by an action query.
GetReturnsRecords	Returns nonzero if the query defined by the querydef returns records.
GetSQL	Returns the SQL string that specifies the query defined by the querydef.
GetType	Returns the query type: delete, update, append, make-table, and so on.
IsOpen	Returns nonzero if the querydef is open and can be executed.
SetConnect	Sets the connection string for a SQL pass-through query on an ODBC data source.
SetName	Sets the name of the saved query, replacing the name in use when the querydef was created.

SetODBCTimeout	Sets the timeout value used by ODBC (for an ODBC query) when the querydef is executed.
SetReturnsRecords	Specifies whether the querydef returns records. Setting this attribute to **TRUE** is only valid for SQL pass-through queries.
SetSQL	Sets the SQL string that specifies the query defined by the querydef.

Operations

Execute	Executes the query defined by the querydef object.
GetFieldCount	Returns the number of fields defined by the querydef.
GetFieldInfo	Returns information about a specified field defined in the query.
GetParameterCount	Returns the number of parameters defined for the query.
GetParameterInfo	Returns information about a specified parameter to the query.
GetParamValue	Returns the value of a specified parameter to the query.
SetParamValue	Sets the value of a specified parameter to the query.

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart

CDaoQueryDef Member Functions

For information about the member functions in **CDaoQueryDef**, see CDaoQueryDef Members.

CDaoQueryDef::Append

Call this member function after you call Create to create a new querydef object.

```
virtual void Append( );
```

Remarks

Append saves the querydef in the database by appending the object to the database's QueryDefs collection. You can use the querydef as a temporary object without appending it, but if you want it to persist, you must call **Append**.

If you attempt to append a temporary querydef object, MFC throws an exception of type CDaoException.

See Also

Reference

CDaoQueryDef Class

Hierarchy Chart

Other Resources

CDaoQueryDef Members

CDaoQueryDef::CanUpdate

Call this member function to determine whether you can modify the querydef — such as changing its name or SQL string.

```
BOOL CanUpdate( );
```

Return Value

Nonzero if you are permitted to modify the querydef; otherwise 0.

Remarks

You can modify the querydef if:

- It is not based on a database that is open read-only.
- You have update permissions for the database.

  This depends on whether you have implemented security features. MFC does not provide support for security; you must implement it yourself by calling DAO directly or by using Microsoft Access. See the topic "Permissions Property" in DAO Help.

See Also

Reference

CDaoQueryDef Class

Hierarchy Chart

Other Resources

CDaoQueryDef Members

CDaoQueryDef::CDaoQueryDef

Constructs a **CDaoQueryDef** object.

```
CDaoQueryDef(
   CDaoDatabase* pDatabase
);
```

Parameters

pDatabase

A pointer to an open CDaoDatabase object.

Remarks

The object can represent an existing querydef stored in the database's QueryDefs collection, a new query to be stored in the collection, or a temporary query, not to be stored. Your next step depends on the type of querydef:

- If the object represents an existing querydef, call the object's Open member function to initialize it.
- If the object represents a new querydef to be saved, call the object's Create member function. This adds the object to the database's QueryDefs collection. Then call **CDaoQueryDef** member functions to set the object's attributes. Finally, call Append.
- If the object represents a temporary querydef (not to be saved in the database), call **Create**, passing an empty string for the query's name. After calling **Create**, initialize the querydef by directly setting its attributes. Do not call **Append**.

To set the attributes of the querydef, you can use the SetName, SetSQL, SetConnect, SetODBCTimeout, and SetReturnsRecords member functions.

When you finish with the querydef object, call its Close member function. If you have a pointer to the querydef, use the **delete** operator to destroy the C++ object.

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart
CDaoQueryDef::GetConnect
CDaoQueryDef::GetDateCreated
CDaoQueryDef::GetDateLastUpdated
CDaoQueryDef::GetName
CDaoQueryDef::GetODBCTimeout
CDaoQueryDef::GetReturnsRecords
CDaoQueryDef::GetSQL

Other Resources

CDaoQueryDef Members

CDaoQueryDef::Close

Call this member function when you finish using the querydef object.

```
virtual void Close( );
```

Remarks

Closing the querydef releases the underlying DAO object but does not destroy the saved DAO querydef object or the C++ **CDaoQueryDef** object. This is not the same as CDaoDatabase::DeleteQueryDef, which deletes the querydef from the database's QueryDefs collection in DAO (if not a temporary querydef).

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart
CDaoQueryDef::Open
CDaoQueryDef::Create
CDaoQueryDef::CDaoQueryDef

Other Resources

CDaoQueryDef Members

CDaoQueryDef::Create

Call this member function to create a new saved query or a new temporary query.

```
virtual void Create(
   LPCTSTR lpszName = NULL,
   LPCTSTR lpszSQL = NULL
);
```

Parameters

lpszName

The unique name of the query saved in the database. For details about the string, see the topic "CreateQueryDef Method" in DAO Help. If you accept the default value, an empty string, a temporary querydef is created. Such a query is not saved in the QueryDefs collection.

lpszSQL

The SQL string that defines the query. If you accept the default value of **NULL**, you must later call SetSQL to set the string. Until then, the query is undefined. You can, however, use the undefined query to open a recordset; see Remarks for details. The SQL statement must be defined before you can append the querydef to the QueryDefs collection.

Remarks

If you pass a name in *lpszName*, you can then call Append to save the querydef in the database's QueryDefs collection. Otherwise, the object is a temporary querydef and is not saved. In either case, the querydef is in an open state, and you can either use it to create a CDaoRecordset object or call the querydef's Execute member function.

If you do not supply a SQL statement in *lpszSQL*, you cannot run the query with **Execute** but you can use it to create a recordset. In that case, MFC uses the recordset's default SQL statement.

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart
CDaoQueryDef::Open
CDaoQueryDef::CDaoQueryDef
CDaoRecordset::GetSQL

Other Resources

CDaoQueryDef Members

CDaoQueryDef::Execute

Call this member function to run the query defined by the querydef object.

```
virtual void Execute(
   int nOptions = dbFailOnError
);
```

Parameters

nOptions

An integer that determines the characteristics of the query. For related information, see the topic "Execute Method" in DAO Help. You can use the bitwise-OR operator (**|**) to combine the following constants for this argument:

- **dbDenyWrite** Deny write permission to other users.
- **dbInconsistent** Inconsistent updates.
- **dbConsistent** Consistent updates.
- **dbSQLPassThrough** SQL pass-through. Causes the SQL statement to be passed to an ODBC database for processing.
- **dbFailOnError** Default value. Roll back updates if an error occurs and report the error to the user.
- **dbSeeChanges** Generate a run-time error if another user is changing data you are editing.

Note
For an explanation of the terms "inconsistent" and "consistent," see the topic "Execute Method" in DAO Help.

Remarks

Querydef objects used for execution in this manner can only represent one of the following query types:

- Action queries
- SQL pass-through queries

Execute does not work for queries that return records, such as select queries. **Execute** is commonly used for bulk operation queries, such as **UPDATE**, **INSERT**, or **SELECT INTO**, or for data definition language (DDL) operations.

Tip
The preferred way to work with ODBC data sources is to attach tables to a Microsoft Jet (.MDB) database. For more informati on, see the topic "Accessing External Databases with DAO" in DAO Help.

Call the GetRecordsAffected member function of the querydef object to determine the number of records affected by the most recent **Execute** call. For example, **GetRecordsAffected** returns information about the number of records deleted, updated, or inserted when executing an action query. The count returned will not reflect changes in related tables when cascade updates or deletes are in effect.

If you include both **dbInconsistent** and **dbConsistent** or if you include neither, the result is the default, **dbInconsistent**.

Execute does not return a recordset. Using **Execute** on a query that selects records causes MFC to throw an exception of type CDaoException.

See Also

Reference

CDaoQueryDef Class

Hierarchy Chart

Other Resources

CDaoQueryDef Members

CDaoQueryDef::GetConnect

Call this member function to get the connection string associated with the querydef's data source.

```
CString GetConnect( );
```

Return Value

A CString containing the connection string for the querydef.

Remarks

This function is used only with ODBC data sources and certain ISAM drivers. It is not used with Microsoft Jet (.MDB) databases; in this case, **GetConnect** returns an empty string. For more information, see SetConnect.

Tip
The preferred way to work with ODBC tables is to attach them to an .MDB database. For more information, see the topic "Accessing External Databases with DAO" in DAO Help.

For information about connection strings, see the topic "Connect Property" in DAO Help.

See Also

Reference

CDaoQueryDef Class

Hierarchy Chart

Other Resources

CDaoQueryDef Members

CDaoQueryDef::GetDateCreated

Call this member function to get the date the querydef object was created.

```
COleDateTime GetDateCreated( );
```

Return Value

A COleDateTime object containing the date and time the querydef was created.

Remarks

For related information, see the topic "DateCreated, LastUpdated Properties" in DAO Help.

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart
CDaoQueryDef::GetDateLastUpdated

Other Resources

CDaoQueryDef Members

CDaoQueryDef::GetDateLastUpdated

Call this member function to get the date the querydef object was last updated — when any of its properties were changed, such as its name, its SQL string, or its connection string.

```
COleDateTime GetDateLastUpdated( );
```

Return Value

A COleDateTime object containing the date and time the querydef was last updated.

Remarks

For related information, see the topic "DateCreated, LastUpdated Properties" in DAO Help.

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart
CDaoQueryDef::GetDateCreated

Other Resources

CDaoQueryDef Members

CDaoQueryDef::GetFieldCount

Call this member function to retrieve the number of fields in the query.

```
short GetFieldCount( );
```

Return Value

The number of fields defined in the query.

Remarks

GetFieldCount is useful for looping through all fields in the querydef. For that purpose, use **GetFieldCount** in conjunction with GetFieldInfo.

See Also

Reference

CDaoQueryDef Class

Hierarchy Chart

Other Resources

CDaoQueryDef Members

CDaoQueryDef::GetFieldInfo

Call this member function to obtain various kinds of information about a field defined in the querydef.

```
void GetFieldInfo(
   int nIndex,
   CDaoFieldInfo& fieldinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
void GetFieldInfo(
   LPCTSTR lpszName,
   CDaoFieldInfo& fieldinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
```

Parameters

nIndex

The zero-based index of the desired field in the querydef's Fields collection, for lookup by index.

fieldinfo

A reference to a **CDaoFieldInfo** object that returns the information requested.

dwInfoOptions

Options that specify which information about the field to retrieve. The available options are listed here along with what they cause the function to return:

- **AFX_DAO_PRIMARY_INFO** (Default) Name, Type, Size, Attributes
- **AFX_DAO_SECONDARY_INFO** Primary information plus: Ordinal Position, Required, Allow Zero Length, Source Field, Foreign Name, Source Table, Collating Order
- **AFX_DAO_ALL_INFO** Primary and secondary information plus: Default Value, Validation Text, Validation Rule

lpszName

A string containing the name of the desired field, for lookup by name. You can use a CString.

Remarks

For a description of the information returned in *fieldinfo*, see the CDaoFieldInfo structure. This structure has members that correspond to the descriptive information under *dwInfoOptions* above. If you request one level of information, you get any prior levels of information as well.

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart
CDaoQueryDef::GetFieldCount

Other Resources

CDaoQueryDef Members

CDaoQueryDef::GetName

Call this member function to retrieve the name of the query represented by the querydef.

```
CString GetName( );
```

Return Value

The name of the query.

Remarks

Querydef names are unique user-defined names. For more information about querydef names, see the topic "Name Property" in DAO Help.

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart
CDaoQueryDef::SetName
CDaoQueryDef::GetSQL
CDaoQueryDef::GetReturnsRecords
CDaoQueryDef::GetODBCTimeout

Other Resources

CDaoQueryDef Members

CDaoQueryDef::GetODBCTimeout

Call this member function to retrieve the current time limit before a query to an ODBC data source times out.

```
short GetODBCTimeout( );
```

Return Value

The number of seconds before a query times out.

Remarks

For information about this time limit, see the topic "ODBCTimeout Property" in DAO Help.

Tip
The preferred way to work with ODBC tables is to attach them to a Microsoft Jet (.MDB) database. For more information, see the topic "Accessing External Databases with DAO" in DAO Help.

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart
CDaoQueryDef::SetODBCTimeout
CDaoQueryDef::GetName
CDaoQueryDef::GetSQL
CDaoQueryDef::GetReturnsRecords

Other Resources

CDaoQueryDef Members

CDaoQueryDef::GetParameterCount

Call this member function to retrieve the number of parameters in the saved query.

```
short GetParameterCount( );
```

Return Value

The number of parameters defined in the query.

Remarks

GetParameterCount is useful for looping through all parameters in the querydef. For that purpose, use **GetParameterCount** in conjunction with GetParameterInfo.

For related information, see the topics "Parameter Object", "Parameters Collection", and "PARAMETERS Declaration (SQL)" in DAO Help.

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart
CDaoQueryDef::GetParamValue
CDaoQueryDef::SetParamValue

Other Resources

CDaoQueryDef Members

CDaoQueryDef::GetParameterInfo

Call this member function to obtain information about a parameter defined in the querydef.

```
void GetParameterInfo(
   int nIndex,
   CDaoParameterInfo& paraminfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
void GetParameterInfo(
   LPCTSTR lpszName,
   CDaoParameterInfo& paraminfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
```

Parameters

nIndex

The zero-based index of the desired parameter in the querydef's Parameters collection, for lookup by index.

paraminfo

A reference to a CDaoParameterInfo object that returns the information requested.

dwInfoOptions

Options that specify which information about the parameter to retrieve. The available option is listed here along with what it causes the function to return:

- **AFX_DAO_PRIMARY_INFO** (Default) Name, Type

lpszName

A string containing the name of the desired parameter, for lookup by name. You can use a CString.

Remarks

For a description of the information returned in *paraminfo*, see the CDaoParameterInfo structure. This structure has members that correspond to the descriptive information under *dwInfoOptions* above.

For related information, see the topic "PARAMETERS Declaration (SQL)" in DAO Help.

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart
CDaoQueryDef::GetParameterCount

Other Resources

CDaoQueryDef Members

CDaoQueryDef::GetParamValue

Call this member function to retrieve the current value of the specified parameter stored in the querydef's Parameters collection.

```
virtual COleVariant GetParamValue(
   LPCTSTR lpszName
);
virtual COleVariant GetParamValue(
   int nIndex
);
```

Parameters

lpszName

The name of the parameter whose value you want, for lookup by name.

nIndex

The zero-based index of the parameter in the querydef's Parameters collection, for lookup by index. You can obtain this value with calls to GetParameterCount and GetParameterInfo.

Return Value

An object of class COleVariant that contains the parameter's value.

Remarks

You can access the parameter either by name or by its ordinal position in the collection.

For related information, see the topic "PARAMETERS Declaration (SQL)" in DAO Help.

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart
CDaoQueryDef::SetParamValue

Other Resources

CDaoQueryDef Members

CDaoQueryDef::GetRecordsAffected

Call this member function to determine how many records were affected by the last call of Execute.

```
long GetRecordsAffected( );
```

Return Value

The number of records affected.

Remarks

The count returned will not reflect changes in related tables when cascade updates or deletes are in effect.

For related information see the topic "RecordsAffected Property" in DAO Help.

See Also

Reference

CDaoQueryDef Class

Hierarchy Chart

Other Resources

CDaoQueryDef Members

CDaoQueryDef::GetReturnsRecords

Call this member function to determine whether the querydef is based on a query that returns records.

```
BOOL GetReturnsRecords( );
```

Return Value

Nonzero if the querydef is based on a query that returns records; otherwise 0.

Remarks

This member function is only used for SQL pass-through queries. For more information about SQL queries, see the Execute member function. For more information about working with SQL pass-through queries, see the SetReturnsRecords member function.

For related information, see the topic "ReturnsRecords Property" in DAO Help.

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart
CDaoQueryDef::GetName
CDaoQueryDef::GetSQL
CDaoQueryDef::GetODBCTimeout

Other Resources

CDaoQueryDef Members

CDaoQueryDef::GetSQL

Call this member function to retrieve the SQL statement that defines the query on which the querydef is based.

```
CString GetSQL( );
```

Return Value

The SQL statement that defines the query on which the querydef is based.

Remarks

You will then probably parse the string for keywords, table names, and so on.

For related information, see the topics "SQL Property", "Comparison of Microsoft Jet Database Engine SQL and ANSI SQL", and "Querying a Database with SQL in Code" in DAO Help.

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart
CDaoQueryDef::SetSQL
CDaoQueryDef::GetName
CDaoQueryDef::GetReturnsRecords
CDaoQueryDef::GetODBCTimeout

Other Resources

CDaoQueryDef Members

CDaoQueryDef::GetType

Call this member function to determine the query type of the querydef.

```
short GetType( );
```

Return Value

The type of the query defined by the querydef. For values, see Remarks.

Remarks

The query type is set by what you specify in the querydef's SQL string when you create the querydef or call an existing querydef's SetSQL member function. The query type returned by this function can be one of the following values:

- **dbQSelect** Select
- **dbQAction** Action
- **dbQCrosstab** Crosstab
- **dbQDelete** Delete
- **dbQUpdate** Update
- **dbQAppend** Append
- **dbQMakeTable** Make-table
- **dbQDDL** Data-definition
- **dbQSQLPassThrough** Pass-through
- **dbQSetOperation** Union
- **dbQSPTBulk** Used with **dbQSQLPassThrough** to specify a query that does not return records.

Note
To create a SQL pass-through query, don't set the **dbSQLPassThrough** constant. This is set automatically by the Microsoft Jet database engine when you create a querydef object and set the connection string.

For information about SQL strings, see GetSQL. For information about query types, see Execute.

See Also

Reference

CDaoQueryDef Class

Hierarchy Chart

Other Resources

CDaoQueryDef Members

CDaoQueryDef::IsOpen

Call this member function to determine whether the **CDaoQueryDef** object is currently open.

```
BOOL IsOpen( ) const;
```

Return Value

Nonzero if the **CDaoQueryDef** object is currently open; otherwise 0.

Remarks

A querydef must be in an open state before you use it to call Execute or to create a CDaoRecordset object. To put a querydef into an open state call either Create (for a new querydef) or Open (for an existing querydef).

See Also

Reference

CDaoQueryDef Class

Hierarchy Chart

Other Resources

CDaoQueryDef Members

CDaoQueryDef::Open

Call this member function to open a querydef previously saved in the database's QueryDefs collection.

```
virtual void Open(
   LPCTSTR lpszName = NULL
);
```

Parameters

lpszName

A string that contains the name of the saved querydef to open. You can use a CString.

Remarks

Once the querydef is open, you can call its Execute member function or use the querydef to create a CDaoRecordset object.

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart
CDaoQueryDef::IsOpen
CDaoQueryDef::Close
CDaoQueryDef::SetName
CDaoQueryDef::Create

Other Resources

CDaoQueryDef Members

CDaoQueryDef::SetConnect

Call this member function to set the querydef object's connection string.

```
void SetConnect(
   LPCTSTR lpszConnect
);
```

Parameters

lpszConnect

A string that contains a connection string for the associated CDaoDatabase object.

Remarks

The connection string is used to pass additional information to ODBC and certain ISAM drivers as needed. It is not used for Microsoft Jet (.MDB) databases.

Tip
The preferred way to work with ODBC tables is to attach them to an .MDB database.

Before executing a querydef that represents a SQL pass-through query to an ODBC data source, set the connection string with **SetConnect** and call SetReturnsRecords to specify whether the query returns records.

For more information about the connection string's structure and examples of connection string components, see the topic "Connect Property" in DAO Help.

See Also

Reference

CDaoQueryDef Class

Hierarchy Chart

Other Resources

CDaoQueryDef Members

CDaoQueryDef::SetName

Call this member function if you want to change the name of a querydef that is not temporary.

```
void SetName(
   LPCTSTR lpszName
);
```

Parameters

lpszName

A string that contains the new name for a nontemporary query in the associated CDaoDatabase object.

Remarks

Querydef names are unique, user-defined names. You can call **SetName** before the querydef object is appended to the QueryDefs collection.

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart
CDaoQueryDef::GetName
CDaoQueryDef::SetSQL
CDaoQueryDef::SetConnect
CDaoQueryDef::SetODBCTimeout
CDaoQueryDef::SetReturnsRecords

Other Resources

CDaoQueryDef Members

CDaoQueryDef::SetODBCTimeout

Call this member function to set the time limit before a query to an ODBC data source times out.

```
void SetODBCTimeout(
   short nODBCTimeout
);
```

Parameters

nODBCTimeout

The number of seconds before a query times out.

Remarks

This member function lets you override the default number of seconds before subsequent operations on the connected data source "time out." An operation might time out due to network access problems, excessive query processing time, and so on. Call **SetODBCTimeout** prior to executing a query with this querydef if you want to change the query timeout value. (As ODBC reuses connections, the timeout value is the same for all clients on the same connection.)

The default value for query timeouts is 60 seconds.

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart
CDaoQueryDef::GetODBCTimeout
CDaoQueryDef::SetName
CDaoQueryDef::SetSQL
CDaoQueryDef::SetConnect
CDaoQueryDef::SetReturnsRecords

Other Resources

CDaoQueryDef Members

CDaoQueryDef::SetParamValue

Call this member function to set the value of a parameter in the querydef at run time.

```
virtual void SetParamValue(
   LPCTSTR lpszName,
   const COleVariant& varValue
);
virtual void SetParamValue(
   int nIndex,
   const COleVariant& varValue
);
```

Parameters

lpszName

The name of the parameter whose value you want to set.

varValue

The value to set; see Remarks.

nIndex

The ordinal position of the parameter in the querydef's Parameters collection. You can obtain this value with calls to GetParameterCount and GetParameterInfo.

Remarks

The parameter must already have been established as part of the querydef's SQL string. You can access the parameter either by name or by its ordinal position in the collection.

Specify the value to set as a **COleVariant** object. For information about setting the desired value and type in your **COleVariant** object, see class COleVariant.

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart
CDaoQueryDef::GetParamValue

Other Resources

CDaoQueryDef Members

CDaoQueryDef::SetReturnsRecords

Call this member function as part of the process of setting up a SQL pass-through query to an external database.

```
void SetReturnsRecords(
   BOOL bReturnsRecords
);
```

Parameters

bReturnsRecords

Pass **TRUE** if the query on an external database returns records; otherwise, **FALSE**.

Remarks

In such a case, you must create the querydef and set its properties using other **CDaoQueryDef** member functions. For a description of external databases, see SetConnect.

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart
CDaoQueryDef::GetReturnsRecords
CDaoQueryDef::SetName
CDaoQueryDef::SetSQL
CDaoQueryDef::SetConnect
CDaoQueryDef::SetODBCTimeout

Other Resources

CDaoQueryDef Members

CDaoQueryDef::SetSQL

Call this member function to set the SQL statement that the querydef executes.

```
void SetSQL(
   LPCTSTR lpszSQL
);
```

Parameters

lpszSQL

A string containing a complete SQL statement, suitable for execution. The syntax of this string depends on the DBMS that your query targets. For a discussion of syntax used in the Microsoft Jet database engine, see the topic "Building SQL Statements in Code" in DAO Help.

Remarks

A typical use of **SetSQL** is setting up a querydef object for use in a SQL pass-through query. (For the syntax of SQL pass-through queries on your target DBMS, see the documentation for your DBMS.)

See Also

Reference

CDaoQueryDef Class
Hierarchy Chart
CDaoQueryDef::GetSQL
CDaoQueryDef::SetName
CDaoQueryDef::SetConnect
CDaoQueryDef::SetODBCTimeout
CDaoQueryDef::SetReturnsRecords

Other Resources

CDaoQueryDef Members

CDaoQueryDef Data Members

For information about the data members in **CDaoQueryDef**, see CDaoQueryDef Members.

CDaoQueryDef::m_pDatabase

Contains a pointer to the CDaoDatabase object associated with the querydef object.

Remarks

Use this pointer if you need to access the database directly — for example, to obtain pointers to other querydef or recordset objects in the database's collections.

See Also

Reference

CDaoQueryDef Class

Hierarchy Chart

Other Resources

CDaoQueryDef Members

CDaoQueryDef::m_pDAOQueryDef

Contains a pointer to the OLE interface for the underlying DAO querydef object.

Remarks

This pointer is provided for completeness and consistency with the other classes. However, because MFC rather fully encapsulates DAO querydefs, you are unlikely to need it. If you do use it, do so cautiously — in particular, do not change the value of the pointer unless you know what you are doing.

See Also

Reference

CDaoQueryDef Class

Hierarchy Chart

Other Resources

CDaoQueryDef Members

CDaoRecordset Class

Represents a set of records selected from a data source.

```
class CDaoRecordset : public CObject
```

Remarks

Known as "recordsets," **CDaoRecordset** objects are available in the following three forms:

- Table-type recordsets represent a base table that you can use to examine, add, change, or delete records from a single database table.
- Dynaset-type recordsets are the result of a query that can have updateable records. These recordsets are a set of records that you can use to examine, add, change, or delete records from an underlying database table or tables. Dynaset-type recordsets can contain fields from one or more tables in a database.
- Snapshot-type recordsets are a static copy of a set of records that you can use to find data or generate reports. These recordsets can contain fields from one or more tables in a database but cannot be updated.

Each form of recordset represents a set of records fixed at the time the recordset is opened. When you scroll to a record in a table-type recordset or a dynaset-type recordset, it reflects changes made to the record after the recordset is opened, either by other users or by other recordsets in your application. (A snapshot-type recordset cannot be updated.) You can use **CDaoRecordset** directly or derive an application-specific recordset class from **CDaoRecordset**. You can then:

- Scroll through the records.
- Set an index and quickly look for records using Seek (table-type recordsets only).
- Find records based on a string comparison: "<", "<=", "=", ">=", or ">" (dynaset-type and snapshot-type recordsets).
- Update the records and specify a locking mode (except snapshot-type recordsets).
- Filter the recordset to constrain which records it selects from those available on the data source.
- Sort the recordset.
- Parameterize the recordset to customize its selection with information not known until run time.

Class **CDaoRecordset** supplies an interface similar to that of class **CRecordset**. The main difference is that class **CDaoRecordset** accesses data through a Data Access Object (DAO) based on OLE. Class **CRecordset** accesses the DBMS through Open Database Connectivity (ODBC) and an ODBC driver for that DBMS.

Note
The DAO database classes are distinct from the MFC database classes based on Open Database Connectivity (ODBC). All DAO database class names have the "CDao" prefix. You can still access ODBC data sources with the DAO classes; the DAO classes generally offer superior capabilities because they are specific to the Microsoft Jet database engine.

You can either use **CDaoRecordset** directly or derive a class from **CDaoRecordset**. To use a recordset class in either case, open a database and construct a recordset object, passing the constructor a pointer to your **CDaoDatabase** object. You can also construct a **CDaoRecordset** object and let MFC create a temporary **CDaoDatabase** object for you. Then call the recordset's Open member function, specifying whether the object is a table-type recordset, a dynaset-type recordset, or a snapshot-type recordset. Calling **Open** selects data from the database and retrieves the first record.

Use the object's member functions and data members to scroll through the records and operate on them. The operations available depend on whether the object is a table-type recordset, a dynaset-type recordset, or a snapshot-type recordset, and whether it is updateable or read-only — this depends on the capability of the database or Open Database Connectivity (ODBC) data source. To refresh records that may have been changed or added since the **Open** call, call the object's Requery member function. Call the object's **Close** member function and destroy the object when you finish with it.

CDaoRecordset uses DAO record field exchange (DFX) to support reading and updating of record fields through type-safe C++ members of your **CDaoRecordset** or **CDaoRecordset**-derived class. You can also implement dynamic binding of

columns in a database without using the DFX mechanism using GetFieldValue and SetFieldValue.

For related information, see the topic "Recordset Object" in DAO Help.

Requirements

Header: afxdao.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class
Hierarchy Chart
CDaoTableDef Class
CDaoWorkspace Class
CDaoDatabase Class
CDaoQueryDef Class

Other Resources

CDaoRecordset Members

CDaoRecordset Members

Base Class Members

Data Members

Construction

Attributes

Recordset Update Operations

Recordset Navigation Operations

Other Recordset Operations

Overridables

Base Class Members

CObject Members

Data Members

m_bCheckCacheForDirtyFields	Contains a flag indicating whether fields are automatically marked as changed.
m_nFields	Contains the number of field data members in the recordset class and the number of columns s elected by the recordset from the data source.
m_nParams	Contains the number of parameter data members in the recordset class — the number of para meters passed with the recordset's query
m_pDAORecordset	A pointer to the DAO interface underlying the recordset object.
m_pDatabase	Source database for this result set. Contains a pointer to a CDaoDatabase object.
m_strFilter	Contains a string used to construct a SQL **WHERE** statement.
m_strSort	Contains a string used to construct a SQL **ORDER BY** statement.

Construction

CDaoRecordset	Constructs a **CDaoRecordset** object.
Close	Closes the recordset.
Open	Creates a new recordset from a table, dynaset, or snapshot.

Attributes

CanAppend	Returns nonzero if new records can be added to the recordset via the AddNew member function.
CanBookmark	Returns nonzero if the recordset supports bookmarks.
CanRestart	Returns nonzero if Requery can be called to run the recordset's query again.
CanScroll	Returns nonzero if you can scroll through the records.
CanTransact	Returns nonzero if the data source supports transactions.
CanUpdate	Returns nonzero if the recordset can be updated (you can add, update, or delete records).

GetCurrentIndex	Returns a **CString** containing the name of the index most recently used on an indexed, table-type **CDaoRecordset**.
GetDateCreated	Returns the date and time the base table underlying a **CDaoRecordset** object was created
GetDateLastUpdated	Returns the date and time of the most recent change made to the design of a base table underlying a **CDaoRecordset** object.
GetEditMode	Returns a value that indicates the state of editing for the current record.
GetLastModifiedBookmark	Used to determine the most recently added or updated record.
GetName	Returns a **CString** containing the name of the recordset.
GetParamValue	Retrieves the current value of the specified parameter stored in the underlying DAOParameter object.
GetRecordCount	Returns the number of records accessed in a recordset object.
GetSQL	Gets the SQL string used to select records for the recordset.
GetType	Called to determine the type of a recordset: table-type, dynaset-type, or snapshot-type.
GetValidationRule	Returns a **CString** containing the value that validates data as it is entered into a field.
GetValidationText	Retrieves the text that is displayed when a validation rule is not satisfied.
IsBOF	Returns nonzero if the recordset has been positioned before the first record. There is no current record.
IsDeleted	Returns nonzero if the recordset is positioned on a deleted record.
IsEOF	Returns nonzero if the recordset has been positioned after the last record. There is no current record.
IsFieldDirty	Returns nonzero if the specified field in the current record has been changed.
IsFieldNull	Returns nonzero if the specified field in the current record is Null (having no value).
IsFieldNullable	Returns nonzero if the specified field in the current record can be set to Null (having no value).
IsOpen	Returns nonzero if Open has been called previously.
SetCurrentIndex	Called to set an index on a table-type recordset.
SetParamValue	Sets the current value of the specified parameter stored in the underlying DAOParameter object
SetParamValueNull	Sets the current value of the specified parameter to Null (having no value).

Recordset Update Operations

AddNew	Prepares for adding a new record. Call Update to complete the addition.
CancelUpdate	Cancels any pending updates due to an Edit or AddNew operation.

Delete	Deletes the current record from the recordset. You must explicitly scroll to another record after the deletion.
Edit	Prepares for changes to the current record. Call **Update** to complete the edit.
Update	Completes an **AddNew** or **Edit** operation by saving the new or edited data on the data source.

Recordset Navigation Operations

Find	Locates the first, next, previous, or last location of a particular string in a dynaset-type recordset that satisfies the specified criteria and makes that record the current record.
FindFirst	Locates the first record in a dynaset-type or snapshot-type recordset that satisfies the specified criteria and makes that record the current record.
FindLast	Locates the last record in a dynaset-type or snapshot-type recordset that satisfies the specified criteria and makes that record the current record.
FindNext	Locates the next record in a dynaset-type or snapshot-type recordset that satisfies the specified criteria and makes that record the current record.
FindPrev	Locates the previous record in a dynaset-type or snapshot-type recordset that satisfies the specified criteria and makes that record the current record.
GetAbsolutePosition	Returns the record number of a recordset object's current record.
GetBookmark	Returns a value that represents the bookmark on a record.
GetPercentPosition	Returns the position of the current record as a percentage of the total number of records.
Move	Positions the recordset to a specified number of records from the current record in either direction.
MoveFirst	Positions the current record on the first record in the recordset.
MoveLast	Positions the current record on the last record in the recordset.
MoveNext	Positions the current record on the next record in the recordset .
MovePrev	Positions the current record on the previous record in the recordset.
Seek	Locates the record in an indexed table-type recordset object that satisfies the specified criteria for the current index and makes that record the current record.
SetAbsolutePosition	Sets the record number of a recordset object's current record.
SetBookmark	Positions the recordset on a record containing the specified bookmark.
SetPercentPosition	Sets the position of the current record to a location corresponding to a percentage of the total number of records in a recordset.

Other Recordset Operations

FillCache	Fills all or a part of a local cache for a recordset object that contains data from an ODBC data source.
GetCacheSize	Returns a value that specifies the number of records in a dynaset-type recordset containing data to be locally cached from an ODBC data source.

GetCacheStart	Returns a value that specifies the bookmark of the first record in the recordset to be cached.
GetFieldCount	Returns a value that represents the number of fields in a recordset.
GetFieldInfo	Returns specific kinds of information about the fields in the recordset.
GetFieldValue	Returns the value of a field in a recordset.
GetIndexCount	Retrieves the number of indexes in a table underlying a recordset.
GetIndexInfo	Returns various kinds of information about an index.
GetLockingMode	Returns a value that indicates the type of locking that is in effect during editing.
Requery	Runs the recordset's query again to refresh the selected records.
SetCacheSize	Sets a value that specifies the number of records in a dynaset-type recordset containing data to be locally cached from an ODBC data source.
SetCacheStart	Sets a value that specifies the bookmark of the first record in the recordset to be cached.
SetFieldDirty	Marks the specified field in the current record as changed.
SetFieldNull	Sets the value of the specified field in the current record to Null (having no value).
SetFieldValue	Sets the value of a field in a recordset.
SetFieldValueNull	Sets the value of a field in a recordset to Null. (having no value).
SetLockingMode	Sets a value that indicates the type of locking to put into effect during editing.

Overridables

DoFieldExchange	Called to exchange data (in both directions) between the field data members of the recordset and the corresponding record on the data source. Implements DAO record field exchange (DFX).
GetDefaultDBName	Returns the name of the default data source.
GetDefaultSQL	Called to get the default SQL string to execute.

See Also

Reference

CDaoRecordset Class

Hierarchy Chart

CDaoRecordset Member Functions

For information about the member functions in **CDaoRecordset**, see CDaoRecordset Members.

CDaoRecordset::AddNew

Call this member function to add a new record to a table-type or dynaset-type recordset.

```
virtual void AddNew( );
```

Remarks

The record's fields are initially Null. (In database terminology, Null means "having no value" and is not the same as **NULL** in C++.) To complete the operation, you must call the Update member function. **Update** saves your changes to the data source.

Caution
If you edit a record and then scroll to another record without calling **Update**, your changes are lost without warning.

If you add a record to a dynaset-type recordset by calling AddNew, the record is visible in the recordset and included in the underlying table where it becomes visible to any new **CDaoRecordset** objects.

The position of the new record depends on the type of recordset:

- In a dynaset-type recordset, where the new record is inserted is not guaranteed. This behavior changed with Microsoft Jet 3.0 for reasons of performance and concurrency. If your goal is to make the newly added record the current record, get the bookmark of the last modified record and move to that bookmark:

  ```
  rs.SetBookmark( rs.GetLastModifiedBookmark( ) );
  ```

- In a table-type recordset for which an index has been specified, records are returned in their proper place in the sort order. If no index has been specified, new records are returned at the end of the recordset.

The record that was current before you used **AddNew** remains current. If you want to make the new record current and the recordset supports bookmarks, call SetBookmark to the bookmark identified by the LastModified property setting of the underlying DAO recordset object. Doing so is useful for determining the value for counter (auto-increment) fields in an added record. For more information, see GetLastModifiedBookmark.

If the database supports transactions, you can make your **AddNew** call part of a transaction. For more information about transactions, see class CDaoWorkspace. Note that you should call CDaoWorkspace::BeginTrans before calling **AddNew**.

It is illegal to call **AddNew** for a recordset whose Open member function has not been called. A **CDaoException** is thrown if you call **AddNew** for a recordset that cannot be appended. You can determine whether the recordset is updateable by calling CanAppend.

The framework marks changed field data members to ensure they will be written to the record on the data source by the DAO record field exchange (DFX) mechanism. Changing the value of a field generally sets the field dirty automatically, so you will seldom need to call SetFieldDirty yourself, but you might sometimes want to ensure that columns will be explicitly updated or inserted regardless of what value is in the field data member. The DFX mechanism also employs the use of **PSEUDO NULL**. For more information, see CDaoFieldExchange::m_nOperation.

If the double-buffering mechanism is not being used, then changing the value of the field does not automatically set the field as dirty. In this case, it will be necessary to explicitly set the field dirty. The flag contained in m_bCheckCacheForDirtyFields controls this automatic field checking.

Note
If records are double-buffered (that is, automatic field checking is enabled), calling **CancelUpdate** will restore the member v ariables to the values they had before **AddNew** or **Edit** was called.

For related information, see the topics "AddNew Method", "CancelUpdate Method", "LastModified Property", and "EditMode Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::CanUpdate
CDaoRecordset::CancelUpdate
CDaoRecordset::Delete
CDaoRecordset::Edit
CDaoRecordset::Update
CDaoRecordset::CanTransact

Other Resources

CDaoRecordset Members

CDaoRecordset::CanAppend

Call this member function to determine whether the previously opened recordset allows you to add new records by calling the AddNew member function.

```
BOOL CanAppend( ) const;
```

Return Value

Nonzero if the recordset allows adding new records; otherwise 0. **CanAppend** will return 0 if you opened the recordset as read-only.

Remarks

For related information, see the topic "Append Method" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::CanBookmark
CDaoRecordset::CanRestart
CDaoRecordset::CanScroll
CDaoRecordset::CanTransact
CDaoRecordset::CanUpdate

Other Resources

CDaoRecordset Members

CDaoRecordset::CanBookmark

Call this member function to determine whether the previously opened recordset allows you to individually mark records using bookmarks.

```
BOOL CanBookmark( );
```

Return Value

Nonzero if the recordset supports bookmarks, otherwise 0.

Remarks

If you are using recordsets based entirely on Microsoft Jet database engine tables, bookmarks can be used except on snapshot-type recordsets flagged as forward-only scrolling recordsets. Other database products (external ODBC data sources) may not support bookmarks.

For related information, see the topic "Bookmarkable Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::CanAppend
CDaoRecordset::CanRestart
CDaoRecordset::CanScroll
CDaoRecordset::CanTransact
CDaoRecordset::CanUpdate

Other Resources

CDaoRecordset Members

CDaoRecordset::CancelUpdate

The **CancelUpdate** member function cancels any pending updates due to an Edit or AddNew operation.

```
virtual void CancelUpdate( );
```

Remarks

For example, if an application calls the **Edit** or **AddNew** member function and has not called Update, **CancelUpdate** cancels any changes made after **Edit** or **AddNew** was called.

Note
If records are double-buffered (that is, automatic field checking is enabled), calling **CancelUpdate** will restore the member variables to the values they had before **AddNew** or **Edit** was called.

If there is no **Edit** or **AddNew** operation pending, **CancelUpdate** causes MFC to throw an exception. Call the GetEditMode member function to determine if there is a pending operation that can be canceled.

For related information, see the topic "CancelUpdate Method" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::AddNew
CDaoRecordset::Delete
CDaoRecordset::Edit
CDaoRecordset::Update
CDaoRecordset::CanTransact

Other Resources

CDaoRecordset Members

CDaoRecordset::CanRestart

Call this member function to determine whether the recordset allows restarting its query (to refresh its records) by calling the **Requery** member function.

```
BOOL CanRestart( );
```

Return Value

Nonzero if **Requery** can be called to run the recordset's query again, otherwise 0.

Remarks

Table-type recordsets do not support **Requery**.

If **Requery** is not supported, call Close then Open to refresh the data. You can call **Requery** to update a recordset object's underlying parameter query after the parameter values have been changed.

For related information, see the topic "Restartable Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::CanAppend
CDaoRecordset::CanBookmark
CDaoRecordset::CanScroll
CDaoRecordset::CanTransact
CDaoRecordset::CanUpdate

Other Resources

CDaoRecordset Members

CDaoRecordset::CanScroll

Call this member function to determine whether the recordset allows scrolling.

```
BOOL CanScroll( ) const;
```

Return Value

Nonzero if you can scroll through the records, otherwise 0.

Remarks

If you call Open with **dbForwardOnly**, the recordset can only scroll forward.

For related information, see the topic "Positioning the Current Record Pointer with DAO" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::CanAppend
CDaoRecordset::CanBookmark
CDaoRecordset::CanRestart
CDaoRecordset::CanTransact
CDaoRecordset::CanUpdate
CDaoRecordset::Open

Other Resources

CDaoRecordset Members

CDaoRecordset::CanTransact

Call this member function to determine whether the recordset allows transactions.

```
BOOL CanTransact( );
```

Return Value

Nonzero if the underlying data source supports transactions, otherwise 0.

Remarks

For related information, see the topic "Transactions Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::AddNew
CDaoRecordset::CanAppend
CDaoRecordset::CancelUpdate
CDaoRecordset::CanScroll
CDaoRecordset::CanRestart
CDaoRecordset::CanUpdate
CDaoRecordset::Delete
CDaoRecordset::Edit
CDaoRecordset::Update

Other Resources

CDaoRecordset Members

CDaoRecordset::CanUpdate

Call this member function to determine whether the recordset can be updated.

```
BOOL CanUpdate( ) const;
```

Return Value

Nonzero if the recordset can be updated (add, update, and delete records), otherwise 0.

Remarks

A recordset might be read-only if the underlying data source is read-only or if you specified **dbReadOnly** for *nOptions* when you called Open for the recordset.

For related information, see the topics "AddNew Method", "Edit Method", "Delete Method", "Update Method", and "Updatable Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::CanAppend
CDaoRecordset::CanBookmark
CDaoRecordset::CanScroll
CDaoRecordset::CanRestart
CDaoRecordset::CanTransact

Other Resources

CDaoRecordset Members

CDaoRecordset::CDaoRecordset

Constructs a **CDaoRecordset** object.

```
CDaoRecordset(
   CDaoDatabase* pDatabase = NULL
);
```

Parameters

pDatabase

Contains a pointer to a CDaoDatabase object or the value **NULL**. If not **NULL** and the **CDaoDatabase** object's **Open** member function has not been called to connect it to the data source, the recordset attempts to open it for you during its own Open call. If you pass **NULL**, a **CDaoDatabase** object is constructed and connected for you using the data source information you specified if you derived your recordset class from **CDaoRecordset**.

Remarks

You can either use **CDaoRecordset** directly or derive an application-specific class from **CDaoRecordset**. You can use ClassWizard to derive your recordset classes.

Note
If you derive a **CDaoRecordset** class, your derived class must supply its own constructor. In the constructor of your derived class, call the constructor **CDaoRecordset::CDaoRecordset**, passing the appropriate parameters along to it.

Pass **NULL** to your recordset constructor to have a **CDaoDatabase** object constructed and connected for you automatically. This is a useful shortcut that does not require you to construct and connect a **CDaoDatabase** object prior to constructing your recordset. If the **CDaoDatabase** object is not open, a CDaoWorkspace object will also be created for you that uses the default workspace. For more information, see CDaoDatabase::CDaoDatabase.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetDefaultDBName
CDaoRecordset::GetDefaultSQL
CDaoRecordset::GetDateCreated
CDaoRecordset::GetDateLastUpdated

Other Resources

CDaoRecordset Members

CDaoRecordset::Close

Closing a **CDaoRecordset** object removes it from the collection of open recordsets in the associated database.

```
virtual void Close( );
```

Remarks

Because **Close** does not destroy the **CDaoRecordset** object, you can reuse the object by calling **Open** on the same data source or a different data source.

All pending AddNew or Edit statements are canceled, and all pending transactions are rolled back. If you want to preserve pending additions or edits, call Update before you call **Close** for each recordset.

You can call **Open** again after calling **Close**. This lets you reuse the recordset object. A better alternative is to call Requery, if possible.

For related information, see the topic "Close Method" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::Open
CDaoRecordset::CDaoRecordset

Other Resources

CDaoRecordset Members

CDaoRecordset::Delete

Call this member function to delete the current record in an open dynaset-type or table-type recordset object.

```
virtual void Delete( );
```

Remarks

After a successful deletion, the recordset's field data members are set to a Null value, and you must explicitly call one of the recordset navigation member functions (Move, Seek, SetBookmark, and so on) in order to move off the deleted record. When you delete records from a recordset, there must be a current record in the recordset before you call **Delete**; otherwise, MFC throws an exception.

Delete removes the current record and makes it inaccessible. Although you cannot edit or use the deleted record, it remains current. Once you move to another record, however, you cannot make the deleted record current again.

Caution
The recordset must be updatable and there must be a valid record current in the recordset when you call **Delete**. For exampl e, if you delete a record but do not scroll to a new record before you call **Delete** again, **Delete** throws a CDaoException.

You can undelete a record if you use transactions and you call the CDaoWorkspace::Rollback member function. If the base table is the primary table in a cascade delete relationship, deleting the current record may also delete one or more records in a foreign table. For more information, see the definition "cascade delete" in DAO Help.

Unlike **AddNew** and **Edit**, a call to **Delete** is not followed by a call to **Update**.

For related information, see the topics "AddNew Method", "Edit Method", "Delete Method", "Update Method", and "Updatable Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::AddNew
CDaoRecordset::CancelUpdate
CDaoRecordset::Edit
CDaoRecordset::Update
CDaoRecordset::CanTransact

Other Resources

CDaoRecordset Members

CDaoRecordset::DoFieldExchange

The framework calls this member function to automatically exchange data between the field data members of your recordset object and the corresponding columns of the current record on the data source.

```
virtual void DoFieldExchange(
   CDaoFieldExchange* pFX
);
```

Parameters

pFX

Contains a pointer to a **CDaoFieldExchange** object. The framework will already have set up this object to specify a context for the field exchange operation.

Remarks

It also binds your parameter data members, if any, to parameter placeholders in the SQL statement string for the recordset's selection. The exchange of field data, called DAO record field exchange (DFX), works in both directions: from the recordset object's field data members to the fields of the record on the data source, and from the record on the data source to the recordset object. If you are binding columns dynamically, you are not required to implement **DoFieldExchange**.

The only action you must normally take to implement **DoFieldExchange** for your derived recordset class is to create the class with ClassWizard and specify the names and data types of the field data members. You might also add code to what ClassWizard writes to specify parameter data members. If all fields are to be bound dynamically, this function will be inactive unless you specify parameter data members.

When you declare your derived recordset class with ClassWizard, the wizard writes an override of **DoFieldExchange** for you, which resembles the following example:

```
void CCustSet::DoFieldExchange(CDaoFieldExchange* pFX)
{
   //{{AFX_FIELD_MAP(CCustSet)
   pFX->SetFieldType(CDaoFieldExchange::outputColumn);
   DFX_Text(pFX, "Name", m_strName);
   DFX_Short(pFX, "Age", m_wAge);
   //}}AFX_FIELD_MAP
}
```

See Also

Reference

CDaoRecordset Class

Hierarchy Chart

CDaoException Class

Other Resources

CDaoRecordset Members

CDaoRecordset::Edit

Call this member function to allow changes to the current record.

```
virtual void Edit( );
```

Remarks

Once you call the **Edit** member function, changes made to the current record's fields are copied to the copy buffer. After you make the desired changes to the record, call **Update** to save your changes. **Edit** saves the values of the recordset's data members. If you call **Edit**, make changes, then call **Edit** again, the record's values are restored to what they were before the first **Edit** call.

Caution
If you edit a record and then perform any operation that moves to another record without first calling **Update**, your changes are lost without warning. In addition, if you close the recordset or the parent database, your edited record is discarded witho ut warning.

In some cases, you may want to update a column by making it Null (containing no data). To do so, call **SetFieldNull** with a parameter of **TRUE** to mark the field Null; this also causes the column to be updated. If you want a field to be written to the data source even though its value has not changed, call **SetFieldDirty** with a parameter of **TRUE**. This works even if the field had the value Null.

The framework marks changed field data members to ensure they will be written to the record on the data source by the DAO record field exchange (DFX) mechanism. Changing the value of a field generally sets the field dirty automatically, so you will seldom need to call SetFieldDirty yourself, but you might sometimes want to ensure that columns will be explicitly updated or inserted regardless of what value is in the field data member. The DFX mechanism also employs the use of **PSEUDO NULL**. For more information, see CDaoFieldExchange::m_nOperation.

If the double-buffering mechanism is not being used, then changing the value of the field does not automatically set the field as dirty. In this case, it will be necessary to explicitly set the field dirty. The flag contained in m_bCheckCacheForDirtyFields controls this automatic field checking.

When the recordset object is pessimistically locked in a multiuser environment, the record remains locked from the time **Edit** is used until the updating is complete. If the recordset is optimistically locked, the record is locked and compared with the pre-edited record just before it is updated in the database. If the record has changed since you called **Edit**, the **Update** operation fails and MFC throws an exception. You can change the locking mode with **SetLockingMode**.

Note
Optimistic locking is always used on external database formats, such as ODBC and installable ISAM.

The current record remains current after you call **Edit**. To call **Edit**, there must be a current record. If there is no current record or if the recordset does not refer to an open table-type or dynaset-type recordset object, an exception occurs. Calling **Edit** causes a **CDaoException** to be thrown under the following conditions:

- There is no current record.
- The database or recordset is read-only.
- No fields in the record are updatable.
- The database or recordset was opened for exclusive use by another user.
- Another user has locked the page containing your record.

If the data source supports transactions, you can make the **Edit** call part of a transaction. Note that you should call **CDaoWorkspace::BeginTrans** before calling **Edit** and after the recordset has been opened. Also note that calling **CDaoWorkspace::CommitTrans** is not a substitute for calling **Update** to complete the **Edit** operation. For more information about transactions, see class **CDaoWorkspace**.

For related information, see the topics "AddNew Method", "Edit Method", "Delete Method", "Update Method", and "Updatable Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::AddNew
CDaoRecordset::CancelUpdate
CDaoRecordset::CanTransact
CDaoRecordset::Delete
CDaoRecordset::Update

Other Resources

CDaoRecordset Members

CDaoRecordset::FillCache

Call this member function to cache a specified number of records from the recordset.

```
void FillCache(
   long* pSize = NULL,
   COleVariant* pBookmark = NULL
);
```

Parameters

pSize

Specifies the number of rows to fill in the cache. If you omit this parameter, the value is determined by the CacheSize property setting of the underlying DAO object.

pBookmark

A COleVariant specifying a bookmark. The cache is filled starting from the record indicated by this bookmark. If you omit this parameter, the cache is filled starting from the record indicated by the CacheStart property of the underlying DAO object.

Remarks

Caching improves the performance of an application that retrieves, or fetches, data from a remote server. A cache is space in local memory that holds the data most recently fetched from the server on the assumption that the data will probably be requested again while the application is running. When data is requested, the Microsoft Jet database engine checks the cache for the data first rather than fetching it from the server, which takes more time. Using data caching on non-ODBC data sources has no effect as the data is not saved in the cache.

Rather than waiting for the cache to be filled with records as they are fetched, you can explicitly fill the cache at any time by calling the **FillCache** member function. This is a faster way to fill the cache because **FillCache** fetches several records at once instead of one at a time. For example, while each screenful of records is being displayed, you can have your application call **FillCache** to fetch the next screenful of records.

Any ODBC database accessed with recordset objects can have a local cache. To create the cache, open a recordset object from the remote data source, and then call the **SetCacheSize** and **SetCacheStart** member functions of the recordset. If *lSize* and *lBookmark* create a range that is partly or wholly outside the range specified by **SetCacheSize** and **SetCacheStart**, the portion of the recordset outside this range is ignored and is not loaded into the cache. If **FillCache** requests more records than remain in the remote data source, only the remaining records are fetched, and no exception is thrown.

Records fetched from the cache do not reflect changes made concurrently to the source data by other users.

FillCache fetches only records not already cached. To force an update of all the cached data, call the **SetCacheSize** member function with an *lSize* parameter equal to 0, call **SetCacheSize** again with the *lSize* parameter equal to the size of the cache you originally requested, and then call **FillCache**.

For related information, see the topic "FillCache Method" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetCacheSize
CDaoRecordset::GetCacheStart
CDaoRecordset::SetCacheSize
CDaoRecordset::SetCacheStart

Other Resources

CDaoRecordset Members

CDaoRecordset::Find

Call this member function to locate a particular string in a dynaset- or snapshot-type recordset using a comparison operator.

```
virtual BOOL Find(
   long lFindType,
   LPCTSTR lpszFilter
);
```

Parameters

lFindType

A value indicating the type of Find operation desired. The possible values are:

- **AFX_DAO_NEXT** Find the next location of a matching string.
- **AFX_DAO_PREV** Find the previous location of a matching string.
- **AFX_DAO_FIRST** Find the first location of a matching string.
- **AFX_DAO_LAST** Find the last location of a matching string.

lpszFilter

A string expression (like the **WHERE** clause in a SQL statement without the word **WHERE**) used to locate the record. For example:

```
Find(AFX_DAO_FIRST, "colRecID = 7")
Find(AFX_DAO_NEXT, "customerName = 'Jones'")
```

Return Value

Nonzero if matching records are found, otherwise 0.

Remarks

You can find the first, next, previous, or last instance of the string. **Find** is a virtual function, so you can override it and add your own implementation. The **FindFirst**, **FindLast**, **FindNext**, and **FindPrev** member functions call the **Find** member function, so you can use **Find** to control the behavior of all Find operations.

To locate a record in a table-type recordset, call the Seek member function.

Tip

The smaller the set of records you have, the more effective **Find** will be. In general, and especially with ODBC data, it is better to create a new query that retrieves just the records you want.

For related information, see the topic "FindFirst, FindLast, FindNext, FindPrevious Methods" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::FindFirst
CDaoRecordset::FindLast
CDaoRecordset::FindNext
CDaoRecordset::FindPrev

Other Resources

CDaoRecordset Members

CDaoRecordset::FindFirst

Call this member function to find the first record that matches a specified condition.

```
BOOL FindFirst(
   LPCTSTR lpszFilter
);
```

Parameters

lpszFilter

A string expression (like the **WHERE** clause in a SQL statement without the word **WHERE**) used to locate the record.

Return Value

Nonzero if matching records are found, otherwise 0.

Remarks

The **FindFirst** member function begins its search from the beginning of the recordset and searches to the end of the recordset.

If you want to include all the records in your search (not just those that meet a specific condition) use one of the Move operations to move from record to record. To locate a record in a table-type recordset, call the **Seek** member function.

If a record matching the criteria is not located, the current record pointer is undetermined, and **FindFirst** returns zero. If the recordset contains more than one record that satisfies the criteria, **FindFirst** locates the first occurrence, **FindNext** locates the next occurrence, and so on.

Caution
If you edit the current record, be sure to save the changes by calling the **Update** member function before you move to another record. If you move to another record without updating, your changes are lost without warning.

The **Find** member functions search from the location and in the direction specified in the following table:

Find operations	Begin	Search direction
FindFirst	Beginning of recordset	End of recordset
FindLast	End of recordset	Beginning of recordset
FindNext	Current record	End of recordset
FindPrevious	Current record	Beginning of recordset

Note
When you call **FindLast**, the Microsoft Jet database engine fully populates your recordset before beginning the search, if this has not already been done. The first search may take longer than subsequent searches.

Using one of the Find operations is not the same as calling **MoveFirst** or **MoveNext**, however, which simply makes the first or next record current without specifying a condition. You can follow a Find operation with a Move operation.

Keep the following in mind when using the Find operations:

- If **Find** returns nonzero, the current record is not defined. In this case, you must position the current record pointer back to a valid record.
- You cannot use a Find operation with a forward-only scrolling snapshot-type recordset.
- You should use the U.S. date format (month-day-year) when you search for fields containing dates, even if you are not using the U.S. version of the Microsoft Jet database engine; otherwise, matching records may not be found.

- When working with ODBC databases and large dynasets, you may discover that using the Find operations is slow, especially when working with large recordsets. You can improve performance by using SQL queries with customized **ORDER BY** or **WHERE** clauses, parameter queries, or **CDaoQuerydef** objects that retrieve specific indexed records.

For related information, see the topic "FindFirst, FindLast, FindNext, FindPrevious Methods" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::Find
CDaoRecordset::FindLast
CDaoRecordset::FindNext
CDaoRecordset::FindPrev

Other Resources

CDaoRecordset Members

CDaoRecordset::FindLast

Call this member function to find the last record that matches a specified condition.

```
BOOL FindLast(
   LPCTSTR lpszFilter
);
```

Parameters

lpszFilter

A string expression (like the **WHERE** clause in a SQL statement without the word **WHERE**) used to locate the record.

Return Value

Nonzero if matching records are found, otherwise 0.

Remarks

The **FindLast** member function begins its search at the end of the recordset and searches backward towards the beginning of the recordset.

If you want to include all the records in your search (not just those that meet a specific condition) use one of the Move operations to move from record to record. To locate a record in a table-type recordset, call the **Seek** member function.

If a record matching the criteria is not located, the current record pointer is undetermined, and **FindLast** returns zero. If the recordset contains more than one record that satisfies the criteria, **FindFirst** locates the first occurrence, **FindNext** locates the next occurrence after the first occurrence, and so on.

Caution
If you edit the current record, be sure you save the changes by calling the **Update** member function before you move to another record. If you move to another record without updating, your changes are lost without warning.

Using one of the Find operations is not the same as calling **MoveFirst** or **MoveNext**, however, which simply makes the first or next record current without specifying a condition. You can follow a Find operation with a Move operation.

Keep the following in mind when using the Find operations:

- If **Find** returns nonzero, the current record is not defined. In this case, you must position the current record pointer back to a valid record.
- You cannot use a Find operation with a forward-only scrolling snapshot-type recordset.
- You should use the U.S. date format (month-day-year) when you search for fields containing dates, even if you are not using the U.S. version of the Microsoft Jet database engine; otherwise, matching records may not be found.
- When working with ODBC databases and large dynasets, you may discover that using the Find operations is slow, especially when working with large recordsets. You can improve performance by using SQL queries with customized **ORDER BY** or **WHERE** clauses, parameter queries, or **CDaoQuerydef** objects that retrieve specific indexed records.

For related information, see the topic "FindFirst, FindLast, FindNext, FindPrevious Methods" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::Find
CDaoRecordset::FindFirst
CDaoRecordset::FindNext
CDaoRecordset::FindPrev

Other Resources

CDaoRecordset Members

CDaoRecordset::FindNext

Call this member function to find the next record that matches a specified condition.

```
BOOL FindNext(
   LPCTSTR lpszFilter
);
```

Parameters

lpszFilter

A string expression (like the **WHERE** clause in a SQL statement without the word **WHERE**) used to locate the record.

Return Value

Nonzero if matching records are found, otherwise 0.

Remarks

The **FindNext** member function begins its search at the current record and searches to the end of the recordset.

If you want to include all the records in your search (not just those that meet a specific condition) use one of the Move operations to move from record to record. To locate a record in a table-type recordset, call the **Seek** member function.

If a record matching the criteria is not located, the current record pointer is undetermined, and **FindNext** returns zero. If the recordset contains more than one record that satisfies the criteria, **FindFirst** locates the first occurrence, **FindNext** locates the next occurrence, and so on.

Caution
If you edit the current record, be sure you save the changes by calling the **Update** member function before you move to another record. If you move to another record without updating, your changes are lost without warning.

Using one of the Find operations is not the same as calling **MoveFirst** or **MoveNext**, however, which simply makes the first or next record current without specifying a condition. You can follow a Find operation with a Move operation.

Keep the following in mind when using the Find operations:

- If **Find** returns nonzero, the current record is not defined. In this case, you must position the current record pointer back to a valid record.
- You cannot use a Find operation with a forward-only scrolling snapshot-type recordset.
- You should use the U.S. date format (month-day-year) when you search for fields containing dates, even if you are not using the U.S. version of the Microsoft Jet database engine; otherwise, matching records may not be found.
- When working with ODBC databases and large dynasets, you may discover that using the Find operations is slow, especially when working with large recordsets. You can improve performance by using SQL queries with customized **ORDER BY** or **WHERE** clauses, parameter queries, or **CDaoQuerydef** objects that retrieve specific indexed records.

For related information, see the topic "FindFirst, FindLast, FindNext, FindPrevious Methods" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::Find
CDaoRecordset::FindFirst
CDaoRecordset::FindLast
CDaoRecordset::FindPrev

Other Resources

CDaoRecordset Members

CDaoRecordset::FindPrev

Call this member function to find the previous record that matches a specified condition.

```
BOOL FindPrev(
   LPCTSTR lpszFilter
);
```

Parameters

lpszFilter

A string expression (like the **WHERE** clause in a SQL statement without the word **WHERE**) used to locate the record.

Return Value

Nonzero if matching records are found, otherwise 0.

Remarks

The **FindPrev** member function begins its search at the current record and searches backward towards the beginning of the recordset.

If you want to include all the records in your search (not just those that meet a specific condition) use one of the Move operations to move from record to record. To locate a record in a table-type recordset, call the **Seek** member function.

If a record matching the criteria is not located, the current record pointer is undetermined, and **FindPrev** returns zero. If the recordset contains more than one record that satisfies the criteria, **FindFirst** locates the first occurrence, **FindNext** locates the next occurrence, and so on.

Caution
If you edit the current record, be sure you save the changes by calling the **Update** member function before you move to another record. If you move to another record without updating, your changes are lost without warning.

Using one of the Find operations is not the same as calling **MoveFirst** or **MoveNext**, however, which simply makes the first or next record current without specifying a condition. You can follow a Find operation with a Move operation.

Keep the following in mind when using the Find operations:

- If **Find** returns nonzero, the current record is not defined. In this case, you must position the current record pointer back to a valid record.
- You cannot use a Find operation with a forward-only scrolling snapshot-type recordset.
- You should use the U.S. date format (month-day-year) when you search for fields containing dates, even if you are not using the U.S. version of the Microsoft Jet database engine; otherwise, matching records may not be found.
- When working with ODBC databases and large dynasets, you may discover that using the Find operations is slow, especially when working with large recordsets. You can improve performance by using SQL queries with customized **ORDER BY** or **WHERE** clauses, parameter queries, or **CDaoQuerydef** objects that retrieve specific indexed records.

For related information, see the topic "FindFirst, FindLast, FindNext, FindPrevious Methods" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::Find
CDaoRecordset::FindFirst
CDaoRecordset::FindLast
CDaoRecordset::FindNext

Other Resources

CDaoRecordset Members

CDaoRecordset::GetAbsolutePosition

Returns the record number of a recordset object's current record.

```
long GetAbsolutePosition( );
```

Return Value

An integer from 0 to the number of records in the recordset. Corresponds to the ordinal position of the current record in the recordset.

Remarks

The AbsolutePosition property value of the underlying DAO object is zero-based; a setting of 0 refers to the first record in the recordset. You can determine the number of populated records in the recordset by calling GetRecordCount. Calling **GetRecordCount** may take some time because it must access all records to determine the count.

If there is no current record, as when there are no records in the recordset, – 1 is returned. If the current record is deleted, the AbsolutePosition property value is not defined, and MFC throws an exception if it is referenced. For dynaset-type recordsets, new records are added to the end of the sequence.

Note
This property is not intended to be used as a surrogate record number. Bookmarks are still the recommended way of retaining and returning to a given position and are the only way to position the current record across all types of recordset objects. In particular, the position of a given record changes when record(s) preceding it are deleted. There is also no assurance that a given record will have the same absolute position if the recordset is re-created again because the order of individual records within a recordset is not guaranteed unless it is created with a SQL statement using an **ORDER BY** clause.

Note
This member function is valid only for dynaset-type and snapshot-type recordsets.

For related information, see the topic "AbsolutePosition Property" in DAO Help.

See Also

Reference

CDaoRecordset Class

Hierarchy Chart

CDaoRecordset::SetAbsolutePosition

Other Resources

CDaoRecordset Members

CDaoRecordset::GetBookmark

Call this member function to obtain the bookmark value in a particular record.

```
COleVariant GetBookmark( );
```

Return Value

Returns a value representing the bookmark on the current record.

Remarks

When a recordset object is created or opened, each of its records already has a unique bookmark if it supports them. Call **CanBookmark** to determine whether a recordset supports bookmarks.

You can save the bookmark for the current record by assigning the value of the bookmark to a **COleVariant** object. To quickly return to that record at any time after moving to a different record, call **SetBookmark** with a parameter corresponding to the value of that **COleVariant** object.

Note
Calling Requery changes DAO bookmarks.

For related information, see the topic "Bookmark Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::SetBookmark
CDaoRecordset::CanBookmark

Other Resources

CDaoRecordset Members

CDaoRecordset::GetCacheSize

Call this member function to obtain the number of records cached.

```
long GetCacheSize( );
```

Return Value

A value that specifies the number of records in a dynaset-type recordset containing data to be locally cached from an ODBC data source.

Remarks

Data caching improves the performance of an application that retrieves data from a remote server through dynaset-type recordset objects. A cache is a space in local memory that holds the data most recently retrieved from the server in the event that the data will be requested again while the application is running. When data is requested, the Microsoft Jet database engine checks the cache for the requested data first rather than retrieving it from the server, which takes more time. Data that does not come from an ODBC data source is not saved in the cache.

Any ODBC data source, such as an attached table, can have a local cache.

For related information, see the topic "CacheSize, CacheStart Properties" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::FillCache
CDaoRecordset::GetCacheStart
CDaoRecordset::SetCacheSize
CDaoRecordset::SetCacheStart

Other Resources

CDaoRecordset Members

CDaoRecordset::GetCacheStart

Call this member function to obtain the bookmark value of the first record in the recordset to be cached.

```
COleVariant GetCacheStart( );
```

Return Value

A **COleVariant** that specifies the bookmark of the first record in the recordset to be cached.

Remarks

The Microsoft Jet database engine requests records within the cache range from the cache, and it requests records outside the cache range from the server.

Note
Records retrieved from the cache do not reflect changes made concurrently to the source data by other users.

For related information, see the topic "CacheSize, CacheStart Properties" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::FillCache
CDaoRecordset::GetCacheSize
CDaoRecordset::SetCacheSize
CDaoRecordset::SetCacheStart

Other Resources

CDaoRecordset Members

CDaoRecordset::GetCurrentIndex

Call this member function to determine the index currently in use in an indexed table-type **CDaoRecordset** object.

```
CString GetCurrentIndex( );
```

Return Value

A **CString** containing the name of the index currently in use with a table-type recordset. Returns an empty string if no index has been set.

Remarks

This index is the basis for ordering records in a table-type recordset, and is used by the Seek member function to locate records.

A **CDaoRecordset** object can have more than one index but can use only one index at a time (although a CDaoTableDef object may have several indexes defined on it).

For related information, see the topic "Index Object" and the definition "current index" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::SetCurrentIndex

Other Resources

CDaoRecordset Members

CDaoRecordset::GetDateCreated

Call this member function to retrieve the date and time a base table was created.

```
COleDateTime GetDateCreated( );
```

Return Value

A COleDateTime object containing the date and time the base table was created.

Remarks

Date and time settings are derived from the computer on which the base table was created.

For related information, see the topic "DateCreated, LastUpdated Properties" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetDateLastUpdated

Other Resources

CDaoRecordset Members

CDaoRecordset::GetDateLastUpdated

Call this member function to retrieve the date and time the schema was last updated.

```
COleDateTime GetDateLastUpdated( );
```

Return Value

A COleDateTime object containing the date and time the base table structure (schema) was last updated.

Remarks

Date and time settings are derived from the computer on which the base table structure (schema) was last updated.

For related information, see the topic "DateCreated, LastUpdated Properties" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetDateCreated

Other Resources

CDaoRecordset Members

CDaoRecordset::GetDefaultDBName

Call this member function to determine the name of the database for this recordset.

```
virtual CString GetDefaultDBName( );
```

Return Value

A **CString** that contains the path and name of the database from which this recordset is derived.

Remarks

If a recordset is created without a pointer to a CDaoDatabase, then this path is used by the recordset to open the default database. By default, this function returns an empty string. When ClassWizard derives a new recordset from **CDaoRecordset**, it will create this function for you.

The following example illustrates the use of the double backslash (\\) in the string, as is required for the string to be interpreted correctly.

```
CString CMyRecordset::GetDefaultDBName(void)
{
   return _T("c:\\mydir\\datasrc.mdb");
}
```

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetDefaultSQL
CDaoRecordset::GetName
CDaoRecordset::GetSQL
CDaoRecordset::GetType

Other Resources

CDaoRecordset Members

CDaoRecordset::GetDefaultSQL

The framework calls this member function to get the default SQL statement on which the recordset is based.

```
virtual CString GetDefaultSQL( );
```

Return Value

A **CString** that contains the default SQL statement.

Remarks

This might be a table name or a SQL **SELECT** statement.

You indirectly define the default SQL statement by declaring your recordset class with ClassWizard, and ClassWizard performs this task for you.

If you pass a null SQL string to Open, then this function is called to determine the table name or SQL for your recordset.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetDefaultDBName
CDaoRecordset::GetName
CDaoRecordset::GetSQL
CDaoRecordset::GetType

Other Resources

CDaoRecordset Members

CDaoRecordset::GetEditMode

Call this member function to determine the state of editing, which is one of the following values:

```
short GetEditMode( );
```

Return Value

Returns a value that indicates the state of editing for the current record.

Remarks

Value	Description
dbEditNone	No editing operation is in progress.
dbEditInProgress	**Edit** has been called.
dbEditAdd	**AddNew** has been called.

For related information, see the topic "EditMode Property" in DAO Help.

See Also

Reference

CDaoRecordset Class

Hierarchy Chart

Other Resources

CDaoRecordset Members

CDaoRecordset::GetFieldCount

Call this member function to retrieve the number of fields (columns) defined in the recordset.

```
short GetFieldCount( );
```

Return Value

The number of fields in the recordset.

Remarks

For related information, see the topic "Count Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetFieldInfo
CDaoRecordset::GetFieldValue
CDaoRecordset::GetIndexCount
CDaoRecordset::GetIndexInfo

Other Resources

CDaoRecordset Members

CDaoRecordset::GetFieldInfo

Call this member function to obtain information about the fields in a recordset.

```
void GetFieldInfo(
   int nIndex,
   CDaoFieldInfo& fieldinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
void GetFieldInfo(
   LPCTSTR lpszName,
   CDaoFieldInfo& fieldinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
```

Parameters

nIndex

The zero-based index of the predefined field in the recordset's Fields collection, for lookup by index.

fieldinfo

A reference to a CDaoFieldInfo structure.

dwInfoOptions

Options that specify which information about the recordset to retrieve. The available options are listed here along with what they cause the function to return. For best performance, retrieve only the level of information you need:

- **AFX_DAO_PRIMARY_INFO** (Default) Name, Type, Size, Attributes
- **AFX_DAO_SECONDARY_INFO** Primary information, plus: Ordinal Position, Required, Allow Zero Length, Collating Order, Foreign Name, Source Field, Source Table
- **AFX_DAO_ALL_INFO** Primary and secondary information, plus: Default Value, Validation Rule, Validation Text

lpszName

The name of the field.

Remarks

One version of the function lets you look up a field by index. The other version lets you look up a field by name.

For a description of the information returned, see the CDaoFieldInfo structure. This structure has members that correspond to the items of information listed above in the description of *dwInfoOptions*. When you request information at one level, you get information for any prior levels as well.

For related information, see the topic "Attributes Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetFieldCount
CDaoRecordset::GetFieldValue
CDaoRecordset::GetIndexCount
CDaoRecordset::GetIndexInfo

Other Resources

CDaoRecordset Members

CDaoRecordset::GetFieldValue

Call this member function to retrieve data in a recordset.

```
virtual void GetFieldValue(
   LPCTSTR lpszName,
   COleVariant& varValue
);
virtual void GetFieldValue(
   int nIndex,
   COleVariant& varValue
);
virtual COleVariant GetFieldValue(
   LPCTSTR lpszName
);
virtual COleVariant GetFieldValue(
   int nIndex
);
```

Parameters

lpszName

A pointer to a string that contains the name of a field.

varValue

A reference to a **COleVariant** object that will store the value of a field.

nIndex

A zero-based index of the field in the recordset's Fields collection, for lookup by index.

Return Value

The two versions of **GetFieldValue** that return a value return a COleVariant object that contains the value of a field.

Remarks

You can look up a field by name or by ordinal position.

Note
It is more efficient to call one of the versions of this member function that takes a **COleVariant** object reference as a parameter, rather than calling a version that returns a **COleVariant** object. The latter versions of this function are kept for backward compatibility.

Use **GetFieldValue** and SetFieldValue to dynamically bind fields at run time rather than statically binding columns using the DoFieldExchange mechanism.

GetFieldValue and the **DoFieldExchange** mechanism can be combined to improve performance. For example, use **GetFieldValue** to retrieve a value that you need only on demand, and assign that call to a "More Information" button in the interface.

For related information, see the topics "Field Object" and "Value Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::SetFieldValue

Other Resources

CDaoRecordset Members

CDaoRecordset::GetIndexCount

Call this member function to determine the number of indexes available on the table-type recordset.

```
short GetIndexCount( );
```

Return Value

The number of indexes in the table-type recordset.

Remarks

GetIndexCount is useful for looping through all indexes in the recordset. For that purpose, use **GetIndexCount** in conjunction with GetIndexInfo. If you call this member function on dynaset-type or snapshot-type recordsets, MFC throws an exception.

For related information, see the topic "Attributes Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetFieldCount
CDaoRecordset::GetFieldInfo
CDaoRecordset::GetIndexInfo

Other Resources

CDaoRecordset Members

CDaoRecordset::GetIndexInfo

Call this member function to obtain various kinds of information about an index defined in the base table underlying a recordset.

```
void GetIndexInfo(
   int nIndex,
   CDaoIndexInfo& indexinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
void GetIndexInfo(
   LPCTSTR lpszName,
   CDaoIndexInfo& indexinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
```

Parameters

nIndex

The zero-based index in the table's Indexes collection, for lookup by numerical position.

indexinfo

A reference to a CDaoIndexInfo structure.

dwInfoOptions

Options that specify which information about the index to retrieve. The available options are listed here along with what they cause the function to return. For best performance, retrieve only the level of information you need:

- **AFX_DAO_PRIMARY_INFO** (Default) Name, Field Info, Fields
- **AFX_DAO_SECONDARY_INFO** Primary information, plus: Primary, Unique, Clustered, IgnoreNulls, Required, Foreign
- **AFX_DAO_ALL_INFO** Primary and secondary information, plus: Distinct Count

lpszName

A pointer to the name of the index object, for lookup by name.

Remarks

One version of the function lets you look up a index by its position in the collection. The other version lets you look up an index by name.

For a description of the information returned, see the CDaoIndexInfo structure. This structure has members that correspond to the items of information listed above in the description of *dwInfoOptions*. When you request information at one level, you get information for any prior levels as well.

For related information, see the topic "Attributes Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetFieldCount
CDaoRecordset::GetFieldInfo
CDaoRecordset::GetIndexCount
CDaoRecordset::GetLastModifiedBookmark

Other Resources

CDaoRecordset Members

CDaoRecordset::GetLastModifiedBookmark

Call this member function to retrieve the bookmark of the most recently added or updated record.

```
COleVariant GetLastModifiedBookmark( );
```

Return Value

A **COleVariant** containing a bookmark that indicates the most recently added or changed record.

Remarks

When a recordset object is created or opened, each of its records already has a unique bookmark if it supports them. Call GetBookmark to determine if the recordset supports bookmarks. If the recordset does not support bookmarks, a **CDaoException** is thrown.

When you add a record, it appears at the end of the recordset, and is not the current record. To make the new record current, call **GetLastModifiedBookmark** and then call **SetBookmark** to return to the newly added record.

For related information, see the topic "LastModified Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetBookmark
CDaoRecordset::SetBookmark

Other Resources

CDaoRecordset Members

CDaoRecordset::GetLockingMode

Call this member function to determine the type of locking in effect for the recordset.

```
BOOL GetLockingMode( );
```

Return Value

Nonzero if the type of locking is pessimistic, otherwise 0 for optimistic record locking.

Remarks

When pessimistic locking is in effect, the data page containing the record you are editing is locked as soon as you call the Edit member function. The page is unlocked when you call the Update or Close member function or any of the Move or Find operations.

When optimistic locking is in effect, the data page containing the record is locked only while the record is being updated with the **Update** member function.

When working with ODBC data sources, the locking mode is always optimistic.

For related information, see the topics "LockEdits Property" and "Locking Behavior in Multiuser Applications" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::SetLockingMode

Other Resources

CDaoRecordset Members

CDaoRecordset::GetName

Call this member function to retrieve the name of the recordset.

```
CString GetName( );
```

Return Value

A **CString** containing the name of the recordset.

Remarks

The name of the recordset must start with a letter and can contain a maximum of 40 characters. It can include numbers and underscore characters but can't include punctuation or spaces.

For related information, see the topic "Name Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetDefaultDBName
CDaoRecordset::GetDefaultSQL
CDaoRecordset::GetSQL
CDaoRecordset::GetType

Other Resources

CDaoRecordset Members

CDaoRecordset::GetParamValue

Call this member function to retrieve the current value of the specified parameter stored in the underlying DAOParameter object.

```
virtual COleVariant GetParamValue(
   int nIndex
);
virtual COleVariant GetParamValue(
   LPCTSTR lpszName
);
```

Parameters

nIndex

The numerical position of the parameter in the underlying DAOParameter object.

lpszName

The name of the parameter whose value you want.

Return Value

An object of class COleVariant that contains the parameter's value.

Remarks

You can access the parameter either by name or by its numerical position in the collection.

For related information, see the topic "Parameter Object" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::SetParamValue
CDaoRecordset::m_nParams

Other Resources

CDaoRecordset Members

CDaoRecordset::GetPercentPosition

When working with a dynaset-type or snapshot-type recordset, if you call **GetPercentPosition** before fully populating the recordset, the amount of movement is relative to the number of records accessed as indicated by calling GetRecordCount.

```
float GetPercentPosition( );
```

Return Value

A number between 0 and 100 that indicates the approximate location of the current record in the recordset object based on a percentage of the records in the recordset.

Remarks

You can move to the last record by calling MoveLast to complete the population of all recordsets, but this may take a significant amount of time.

You can call **GetPercentPosition** on all three types of recordset objects, including tables without indexes. However, you cannot call **GetPercentPosition** on forward-only scrolling snapshots, or on a recordset opened from a pass-through query against an external database. If there is no current record, or he current record has been deleted, a **CDaoException** is thrown.

For related information, see the topic "PercentPosition Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::SetPercentPosition

Other Resources

CDaoRecordset Members

CDaoRecordset::GetRecordCount

Call this member function to find out how many records in a recordset have been accessed.

```
long GetRecordCount( );
```

Return Value

Returns the number of records accessed in a recordset object.

Remarks

GetRecordCount does not indicate how many records are contained in a dynaset-type or snapshot-type recordset until all records have been accessed. This member function call may take a significant amount of time to complete.

Once the last record has been accessed, the return value indicates the total number of undeleted records in the recordset. To force the last record to be accessed, call the **MoveLast** or **FindLast** member function for the recordset. You can also use a SQL Count to determine the approximate number of records your query will return.

As your application deletes records in a dynaset-type recordset, the return value of **GetRecordCount** decreases. However, records deleted by other users are not reflected by **GetRecordCount** until the current record is positioned to a deleted record. If you execute a transaction that affects the record count and subsequently roll back the transaction, **GetRecordCount** will not reflect the actual number of remaining records.

The value of **GetRecordCount** from a snapshot-type recordset is not affected by changes in the underlying tables.

The value of **GetRecordCount** from a table-type recordset reflects the approximate number of records in the table and is affected immediately as table records are added and deleted.

A recordset with no records returns a value of 0. When working with attached tables or ODBC databases, **GetRecordCount** always returns – 1. Calling the **Requery** member function on a recordset resets the value of **GetRecordCount** just as if the query were re-executed.

For related information, see the topic "RecordCount Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetFieldCount
CDaoRecordset::GetFieldInfo
CDaoRecordset::GetIndexCount
CDaoRecordset::GetIndexInfo

Other Resources

CDaoRecordset Members

CDaoRecordset::GetSQL

Call this member function to get the SQL statement that was used to select the recordset's records when it was opened.

```
CString GetSQL( ) const;
```

Return Value

A **CString** that contains the SQL statement.

Remarks

This will generally be a SQL **SELECT** statement.

The string returned by **GetSQL** is typically different from any string you may have passed to the recordset in the *lpszSQL* parameter to the Open member function. This is because the recordset constructs a full SQL statement based on what you passed to **Open**, what you specified with ClassWizard, and what you may have specified in the m_strFilter and m_strSort data members.

Note
Call this member function only after calling **Open**.

For related information, see the topic "SQL Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetDefaultSQL
CDaoRecordset::GetDefaultDBName
CDaoRecordset::GetName
CDaoRecordset::GetType

Other Resources

CDaoRecordset Members

CDaoRecordset::GetType

Call this member function after opening the recordset to determine the type of the recordset object.

```
short GetType( );
```

Return Value

One of the following values that indicates the type of a recordset:

- **dbOpenTable** Table-type recordset
- **dbOpenDynaset** Dynaset-type recordset
- **dbOpenSnapshot** Snapshot-type recordset

Remarks

For related information, see the topic "Type Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetDefaultDBName
CDaoRecordset::GetDefaultSQL
CDaoRecordset::GetName
CDaoRecordset::GetSQL

Other Resources

CDaoRecordset Members

CDaoRecordset::GetValidationRule

Call this member function to determine the rule used to validate data.

```
CString GetValidationRule( );
```

Return Value

A **CString** object containing a value that validates the data in a record as it is changed or added to a table.

Remarks

This rule is text-based, and is applied each time the underlying table is changed. If the data is not legal, MFC throws an exception. The returned error message is the text of the ValidationText property of the underlying field object, if specified, or the text of the expression specified by the ValidationRule property of the underlying field object. You can call GetValidationText to obtain the text of the error message.

For example, a field in a record that requires the day of the month might have a validation rule such as "DAY BETWEEN 1 AND 31."

For related information, see the topic "ValidationRule Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetValidationText

Other Resources

CDaoRecordset Members

CDaoRecordset::GetValidationText

Call this member function to retrieve the text of the ValidationText property of the underlying field object.

```
CString GetValidationText( );
```

Return Value

A **CString** object containing the text of the message that is displayed if the value of a field does not satisfy the validation rule of the underlying field object.

Remarks

For related information, see the topic "ValidationText Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetValidationRule

Other Resources

CDaoRecordset Members

CDaoRecordset::IsBOF

Call this member function before you scroll from record to record to learn whether you have gone before the first record of the recordset.

```
BOOL IsBOF( ) const;
```

Return Value

Nonzero if the recordset contains no records or if you have scrolled backward before the first record; otherwise 0.

Remarks

You can also call **IsBOF** along with **IsEOF** to determine whether the recordset contains any records or is empty. Immediately after you call **Open**, if the recordset contains no records, **IsBOF** returns nonzero. When you open a recordset that has at least one record, the first record is the current record and **IsBOF** returns 0.

If the first record is the current record and you call **MovePrev**, **IsBOF** will subsequently return nonzero. If **IsBOF** returns nonzero and you call **MovePrev**, an exception is thrown. If **IsBOF** returns nonzero, the current record is undefined, and any action that requires a current record will result in an exception.

Effect of specific methods on **IsBOF** and **IsEOF** settings:

- Calling **Open** internally makes the first record in the recordset the current record by calling **MoveFirst**. Therefore, calling **Open** on an empty set of records causes **IsBOF** and **IsEOF** to return nonzero. (See the following table for the behavior of a failed **MoveFirst** or **MoveLast** call.)
- All Move operations that successfully locate a record cause both **IsBOF** and **IsEOF** to return 0.
- An **AddNew** call followed by an **Update** call that successfully inserts a new record will cause **IsBOF** to return 0, but only if **IsEOF** is already nonzero. The state of **IsEOF** will always remain unchanged. As defined by the Microsoft Jet database engine, the current record pointer of an empty recordset is at the end of a file, so any new record is inserted after the current record.
- Any **Delete** call, even if it removes the only remaining record from a recordset, will not change the value of **IsBOF** or **IsEOF**.

This table shows which Move operations are allowed with different combinations of **IsBOF**/**IsEOF**.

	MoveFirst, MoveLast	**MovePrev, Move < 0**	**Move 0**	**MoveNext, Move > 0**
IsBOF=nonzero, **IsEOF**=0	Allowed	Exception	Exception	Allowed
IsBOF=0, **IsEOF**=nonzero	Allowed	Allowed	Exception	Exception
Both nonzero	Exception	Exception	Exception	Exception
Both 0	Allowed	Allowed	Allowed	Allowed

Allowing a Move operation does not mean that the operation will successfully locate a record. It merely indicates that an attempt to perform the specified Move operation is allowed and will not generate an exception. The value of the **IsBOF** and **IsEOF** member functions may change as a result of the attempted move.

The effect of Move operations that do not locate a record on the value of **IsBOF** and **IsEOF** settings is shown in the following table.

	IsBOF	**IsEOF**
MoveFirst, **MoveLast**	Nonzero	Nonzero

Move 0	No change	No change
MovePrev, **Move** < 0	Nonzero	No change
MoveNext, **Move** > 0	No change	Nonzero

For related information, see the topic "BOF, EOF Properties" in DAO Help.

See Also

Reference

CDaoRecordset Class

Hierarchy Chart

CDaoRecordset::IsEOF

Other Resources

CDaoRecordset Members

CDaoRecordset::IsDeleted

Call this member function to determine whether the current record has been deleted.

```
BOOL IsDeleted( ) const;
```

Return Value

Nonzero if the recordset is positioned on a deleted record; otherwise 0.

Remarks

If you scroll to a record and **IsDeleted** returns **TRUE** (nonzero), then you must scroll to another record before you can perform any other recordset operations.

Note
You don't need to check the deleted status for records in a snapshot or table-type recordset. Because records cannot be deleted from a snapshot, there is no need to call **IsDeleted**. For table-type recordsets, deleted records are actually removed from the recordset. Once a record has been deleted, either by you, another user, or in another recordset, you cannot scroll back to that record. Therefore, there is no need to call **IsDeleted**.

When you delete a record from a dynaset, it is removed from the recordset and you cannot scroll back to that record. However, if a record in a dynaset is deleted either by another user or in another recordset based on the same table, **IsDeleted** will return **TRUE** when you later scroll to that record.

For related information, see the topics "Delete Method", "LastModified Property", and "EditMode Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::Delete
CDaoRecordset::IsBOF
CDaoRecordset::IsEOF

Other Resources

CDaoRecordset Members

CDaoRecordset::IsEOF

Call this member function as you scroll from record to record to learn whether you have gone beyond the last record of the recordset.

```
BOOL IsEOF( ) const;
```

Return Value

Nonzero if the recordset contains no records or if you have scrolled beyond the last record; otherwise 0.

Remarks

You can also call **IsEOF** to determine whether the recordset contains any records or is empty. Immediately after you call **Open**, if the recordset contains no records, **IsEOF** returns nonzero. When you open a recordset that has at least one record, the first record is the current record and **IsEOF** returns 0.

If the last record is the current record when you call **MoveNext**, **IsEOF** will subsequently return nonzero. If **IsEOF** returns nonzero and you call **MoveNext**, an exception is thrown. If **IsEOF** returns nonzero, the current record is undefined, and any action that requires a current record will result in an exception.

Effect of specific methods on **IsBOF** and **IsEOF** settings:

- Calling **Open** internally makes the first record in the recordset the current record by calling **MoveFirst**. Therefore, calling **Open** on an empty set of records causes **IsBOF** and **IsEOF** to return nonzero. (See the following table for the behavior of a failed **MoveFirst** call.)
- All Move operations that successfully locate a record cause both **IsBOF** and **IsEOF** to return 0.
- An **AddNew** call followed by an **Update** call that successfully inserts a new record will cause **IsBOF** to return 0, but only if **IsEOF** is already nonzero. The state of **IsEOF** will always remain unchanged. As defined by the Microsoft Jet database engine, the current record pointer of an empty recordset is at the end of a file, so any new record is inserted after the current record.
- Any **Delete** call, even if it removes the only remaining record from a recordset, will not change the value of **IsBOF** or **IsEOF**.

This table shows which Move operations are allowed with different combinations of **IsBOF**/**IsEOF**.

	MoveFirst, MoveLast	**MovePrev, Move < 0**	**Move 0**	**MoveNext, Move > 0**
IsBOF=nonzero, **IsEOF**=0	Allowed	Exception	Exception	Allowed
IsBOF=0, **IsEOF**=nonzero	Allowed	Allowed	Exception	Exception
Both nonzero	Exception	Exception	Exception	Exception
Both 0	Allowed	Allowed	Allowed	Allowed

Allowing a Move operation does not mean that the operation will successfully locate a record. It merely indicates that an attempt to perform the specified Move operation is allowed and will not generate an exception. The value of the **IsBOF** and **IsEOF** member functions may change as a result of the attempted Move.

The effect of Move operations that do not locate a record on the value of **IsBOF** and **IsEOF** settings is shown in the following table.

	IsBOF	**IsEOF**
MoveFirst, **MoveLast**	Nonzero	Nonzero

Move 0	No change	No change
MovePrev, **Move** < 0	Nonzero	No change
MoveNext, **Move** > 0	No change	Nonzero

For related information, see the topic "BOF, EOF Properties" in DAO Help.

See Also

Reference

CDaoRecordset Class

Hierarchy Chart

CDaoRecordset::IsBOF

Other Resources

CDaoRecordset Members

CDaoRecordset::IsFieldDirty

Call this member function to determine whether the specified field data member of a dynaset has been flagged as "dirty" (changed).

```
BOOL IsFieldDirty(
   void* pv
);
```

Parameters

pv

A pointer to the field data member whose status you want to check, or **NULL** to determine if any of the fields are dirty.

Return Value

Nonzero if the specified field data member is flagged as dirty; otherwise 0.

Remarks

The data in all dirty field data members will be transferred to the record on the data source when the current record is updated by a call to the **Update** member function of **CDaoRecordset** (following a call to **Edit** or **AddNew**). With this knowledge, you can take further steps, such as unflagging the field data member to mark the column so it will not be written to the data source.

IsFieldDirty is implemented through **DoFieldExchange**.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::IsFieldNull
CDaoRecordset::IsFieldNullable

Other Resources

CDaoRecordset Members

CDaoRecordset::IsFieldNull

Call this member function to determine whether the specified field data member of a recordset has been flagged as Null.

```
BOOL IsFieldNull(
   void* pv
);
```

Parameters

pv

A pointer to the field data member whose status you want to check, or **NULL** to determine if any of the fields are Null.

Return Value

Nonzero if the specified field data member is flagged as Null; otherwise 0.

Remarks

(In database terminology, Null means "having no value" and is not the same as **NULL** in C++.) If a field data member is flagged as Null, it is interpreted as a column of the current record for which there is no value.

Note

In certain situations, using **IsFieldNull** can be inefficient, as the following code example illustrates:

```
COleVariant varValue;
int nField;

// this code is inefficient because data
// must be retrieved for both IsFieldNull
// and GetFieldValue
if ( !rs.IsFieldNull( pField ) )
    rs.GetFieldValue( nField, varValue );

// this code is more efficient
rs.GetFieldValue( nField, varValue );
if ( varValue.vt == VT_NULL )
    // do something
```

Note

If you are using dynamic record binding, without deriving from **CDaoRecordset**, be sure to use **VT_NULL** as shown in the example.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::IsFieldDirty
CDaoRecordset::IsFieldNullable

Other Resources

CDaoRecordset Members

CDaoRecordset::IsFieldNullable

Call this member function to determine whether the specified field data member is "nullable" (can be set to a Null value; C++ **NULL** is not the same as Null, which, in database terminology, means "having no value").

```
BOOL IsFieldNullable(
   void* pv
);
```

Parameters

pv

A pointer to the field data member whose status you want to check, or **NULL** to determine if any of the fields are Null.

Return Value

Nonzero if the specified field data member can be made Null; otherwise 0.

Remarks

A field that cannot be Null must have a value. If you attempt to set such a field to Null when adding or updating a record, the data source rejects the addition or update, and **Update** will throw an exception. The exception occurs when you call **Update**, not when you call **SetFieldNull**.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::IsFieldDirty
CDaoRecordset::IsFieldNull

Other Resources

CDaoRecordset Members

CDaoRecordset::IsOpen

Call this member function to determine if the recordset is open.

```
BOOL IsOpen( ) const;
```

Return Value

Nonzero if the recordset object's **Open** or **Requery** member function has previously been called and the recordset has not been closed; otherwise 0.

Remarks

See Also

Reference

CDaoRecordset Class

Hierarchy Chart

CDaoRecordset::Open

CDaoRecordset::Close

Other Resources

CDaoRecordset Members

CDaoRecordset::Move

Call this member function to position the recordset *lRows* records from the current record.

```
virtual void Move(
   long lRows
);
```

Parameters

lRows

The number of records to move forward or backward. Positive values move forward, toward the end of the recordset. Negative values move backward, toward the beginning.

Remarks

You can move forward or backward. `Move( 1 )` is equivalent to **MoveNext**, and `Move( -1 )` is equivalent to **MovePrev**.

Caution

Calling any of the **Move** functions throws an exception if the recordset has no records. In general, call both **IsBOF** and **IsEOF** before a Move operation to determine whether the recordset has any records. After you call **Open** or **Requery**, call either **IsBOF** or **IsEOF**.

Note

If you have scrolled past the beginning or end of the recordset (**IsBOF** or **IsEOF** returns nonzero), a call to **Move** throws a **CDaoException**.

Note

If you call any of the **Move** functions while the current record is being updated or added, the updates are lost without warning.

When you call **Move** on a forward-only scrolling snapshot, the *lRows* parameter must be a positive integer and bookmarks are not allowed, so you can move forward only.

To make the first, last, next, or previous record in a recordset the current record, call the **MoveFirst**, **MoveLast**, **MoveNext**, or **MovePrev** member function.

For related information, see the topics "Move Method" and "MoveFirst, MoveLast, MoveNext, MovePrevious Methods" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::MoveFirst
CDaoRecordset::MoveLast
CDaoRecordset::MoveNext
CDaoRecordset::MovePrev

Other Resources

CDaoRecordset Members

CDaoRecordset::MoveFirst

Call this member function to make the first record in the recordset (if any) the current record.

```
void MoveFirst( );
```

Remarks

You do not have to call **MoveFirst** immediately after you open the recordset. At that time, the first record (if any) is automatically the current record.

Caution

Calling any of the **Move** functions throws an exception if the recordset has no records. In general, call both **IsBOF** and **IsEOF** before a Move operation to determine whether the recordset has any records. After you call **Open** or **Requery**, call either **IsBOF** or **IsEOF**.

Note

If you call any of the **Move** functions while the current record is being updated or added, the updates are lost without warning.

Use the **Move** functions to move from record to record without applying a condition. Use the Find operations to locate records in a dynaset-type or snapshot-type recordset object that satisfy a certain condition. To locate a record in a table-type recordset object, call **Seek**.

If the recordset refers to a table-type recordset, movement follows the table's current index. You can set the current index by using the Index property of the underlying DAO object. If you do not set the current index, the order of returned records is undefined.

If you call **MoveLast** on a recordset object based on a SQL query or querydef, the query is forced to completion and the recordset object is fully populated.

You cannot call the **MoveFirst** or **MovePrev** member function with a forward-only scrolling snapshot.

To move the position of the current record in a recordset object a specific number of records forward or backward, call **Move**.

For related information, see the topics "Move Method" and "MoveFirst, MoveLast, MoveNext, MovePrevious Methods" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::Move
CDaoRecordset::MoveLast
CDaoRecordset::MoveNext
CDaoRecordset::MovePrev

Other Resources

CDaoRecordset Members

CDaoRecordset::MoveLast

Call this member function to make the last record (if any) in the recordset the current record.

```
void MoveLast( );
```

Remarks

Caution

Calling any of the **Move** functions throws an exception if the recordset has no records. In general, call both **IsBOF** and **IsEOF** before a Move operation to determine whether the recordset has any records. After you call **Open** or **Requery**, call either **IsBOF** or **IsEOF**.

Note

If you call any of the **Move** functions while the current record is being updated or added, the updates are lost without warning.

Use the **Move** functions to move from record to record without applying a condition. Use the Find operations to locate records in a dynaset-type or snapshot-type recordset object that satisfy a certain condition. To locate a record in a table-type recordset object, call **Seek**.

If the recordset refers to a table-type recordset, movement follows the table's current index. You can set the current index by using the Index property of the underlying DAO object. If you do not set the current index, the order of returned records is undefined.

If you call **MoveLast** on a recordset object based on a SQL query or querydef, the query is forced to completion and the recordset object is fully populated.

To move the position of the current record in a recordset object a specific number of records forward or backward, call **Move**.

For related information, see the topics "Move Method" and "MoveFirst, MoveLast, MoveNext, MovePrevious Methods" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::Move
CDaoRecordset::MoveFirst
CDaoRecordset::MoveNext
CDaoRecordset::MovePrev

Other Resources

CDaoRecordset Members

CDaoRecordset::MoveNext

Call this member function to make the next record in the recordset the current record.

```
void MoveNext( );
```

Remarks

It is recommended that you call **IsBOF** before you attempt to move to the previous record. A call to **MovePrev** will throw a **CDaoException** if **IsBOF** returns nonzero, indicating either that you have already scrolled before the first record or that no records were selected by the recordset.

Caution

Calling any of the **Move** functions throws an exception if the recordset has no records. In general, call both **IsBOF** and **IsEOF** before a Move operation to determine whether the recordset has any records. After you call **Open** or **Requery**, call either **IsBOF** or **IsEOF**.

Note

If you call any of the **Move** functions while the current record is being updated or added, the updates are lost without warning.

Use the **Move** functions to move from record to record without applying a condition. Use the Find operations to locate records in a dynaset-type or snapshot-type recordset object that satisfy a certain condition. To locate a record in a table-type recordset object, call **Seek**.

If the recordset refers to a table-type recordset, movement follows the table's current index. You can set the current index by using the Index property of the underlying DAO object. If you do not set the current index, the order of returned records is undefined.

To move the position of the current record in a recordset object a specific number of records forward or backward, call **Move**.

For related information, see the topics "Move Method" and "MoveFirst, MoveLast, MoveNext, MovePrevious Methods" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::Move
CDaoRecordset::MoveFirst
CDaoRecordset::MoveLast
CDaoRecordset::MovePrev

Other Resources

CDaoRecordset Members

CDaoRecordset::MovePrev

Call this member function to make the previous record in the recordset the current record.

```
void MovePrev( );
```

Remarks

It is recommended that you call **IsBOF** before you attempt to move to the previous record. A call to **MovePrev** will throw a **CDaoException** if **IsBOF** returns nonzero, indicating either that you have already scrolled before the first record or that no records were selected by the recordset.

Caution

Calling any of the **Move** functions throws an exception if the recordset has no records. In general, call both **IsBOF** and **IsEOF** before a Move operation to determine whether the recordset has any records. After you call **Open** or **Requery**, call either **IsBOF** or **IsEOF**.

Note

If you call any of the **Move** functions while the current record is being updated or added, the updates are lost without warning.

Use the **Move** functions to move from record to record without applying a condition. Use the Find operations to locate records in a dynaset-type or snapshot-type recordset object that satisfy a certain condition. To locate a record in a table-type recordset object, call **Seek**.

If the recordset refers to a table-type recordset, movement follows the table's current index. You can set the current index by using the Index property of the underlying DAO object. If you do not set the current index, the order of returned records is undefined.

You cannot call the **MoveFirst** or **MovePrev** member function with a forward-only scrolling snapshot.

To move the position of the current record in a recordset object a specific number of records forward or backward, call **Move**.

For related information, see the topics "Move Method" and "MoveFirst, MoveLast, MoveNext, MovePrevious Methods" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::Move
CDaoRecordset::MoveFirst
CDaoRecordset::MoveLast
CDaoRecordset::MoveNext

Other Resources

CDaoRecordset Members

CDaoRecordset::Open

You must call this member function to retrieve the records for the recordset.

```
virtual void Open(
   int nOpenType = AFX_DAO_USE_DEFAULT_TYPE,
   LPCTSTR lpszSQL = NULL,
   int nOptions = 0
);
virtual void Open(
   CDaoTableDef* pTableDef,
   int nOpenType = dbOpenTable,
   int nOptions = 0
);
virtual void Open(
   CDaoQueryDef* pQueryDef,
   int nOpenType = dbOpenDynaset,
   int nOptions = 0
);
```

Parameters

nOpenType

One of the following values:

- **dbOpenDynaset** A dynaset-type recordset with bidirectional scrolling. This is the default.
- **dbOpenTable** A table-type recordset with bidirectional scrolling.
- **dbOpenSnapshot** A snapshot-type recordset with bidirectional scrolling.

lpszSQL

A string pointer containing one of the following:

- A **NULL** pointer.
- The name of one or more tabledefs and/or querydefs (comma-separated).
- A SQL **SELECT** statement (optionally with a SQL **WHERE** or **ORDER BY** clause).
- A pass-through query.

nOptions

One or more of the options listed below. The default value is 0. Possible values are as follows:

- **dbAppendOnly** You can only append new records (dynaset-type recordset only). This option means literally that records may only be appended. The MFC ODBC database classes have an append-only option that allows records to be retrieved and appended.
- **dbForwardOnly** The recordset is a forward-only scrolling snapshot.
- **dbSeeChanges** Generate an exception if another user is changing data you are editing.
- **dbDenyWrite** Other users cannot modify or add records.
- **dbDenyRead** Other users cannot view records (table-type recordset only).
- **dbReadOnly** You can only view records; other users can modify them.
- **dbInconsistent** Inconsistent updates are allowed (dynaset-type recordset only).
- **dbConsistent** Only consistent updates are allowed (dynaset-type recordset only).

Note

The constants **dbConsistent** and **dbInconsistent** are mutually exclusive. You can use one or the other, but not both in a given instance of **Open**.

pTableDef

A pointer to a CDaoTableDef object. This version is valid only for table-type recordsets. When using this option, the **CDaoDatabase** pointer used to construct the **CDaoRecordset** is not used; rather, the database in which the tabledef resides is used.

pQueryDef

A pointer to a CDaoQueryDef object. This version is valid only for dynaset-type and snapshot-type recordsets. When using this option, the **CDaoDatabase** pointer used to construct the **CDaoRecordset** is not used; rather, the database in which the querydef resides is used.

Remarks

Before calling **Open**, you must construct the recordset object. There are several ways to do this:

- When you construct the recordset object, pass a pointer to a **CDaoDatabase** object that is already open.
- When you construct the recordset object, pass a pointer to a **CDaoDatabase** object that is not open. The recordset opens a **CDaoDatabase** object, but will not close it when the recordset object closes.
- When you construct the recordset object, pass a **NULL** pointer. The recordset object calls **GetDefaultDBName** to get the name of the Microsoft Access .MDB file to open. The recordset then opens a **CDaoDatabase** object and keeps it open as long as the recordset is open. When you call **Close** on the recordset, the **CDaoDatabase** object is also closed.

Note
When the recordset opens the **CDaoDatabase** object, it opens the data source with nonexclusive access.

For the version of **Open** that uses the *lpszSQL* parameter, once the recordset is open you can retrieve records in one of several ways. The first option is to have DFX functions in your **DoFieldExchange**. The second option is to use dynamic binding by calling the **GetFieldValue** member function. These options can be implemented separately or in combination. If they are combined, you will have to pass in the SQL statement yourself on the call to **Open**.

When you use the second version of **Open** where you pass in a **CDaoTableDef** object, the resulting columns will be available for you to bind via **DoFieldExchange** and the DFX mechanism, and/or bind dynamically via **GetFieldValue**.

Note
You can only call **Open** using a **CDaoTableDef** object for table-type recordsets.

When you use the third version of **Open** where you pass in a **CDaoQueryDef** object, that query will be executed, and the resulting columns will be available for you to bind via **DoFieldExchange** and the DFX mechanism, and/or bind dynamically via **GetFieldValue**.

Note
You can only call **Open** using a **CDaoQueryDef** object for dynaset-type and snapshot-type recordsets.

For the first version of **Open** that uses the *lpszSQL* parameter, records are selected based on criteria shown in the following table.

Value of the *lpszSQL* parameter	**Records selected are determined by**	**Example**
NULL	The string returned by **GetDefaultSQL**.	
A comma-separated list of one or more tabledefs and/or querydef names.	All columns represented in the **DoFieldExchange**.	`"Customer"`

SELECT column-list **FROM** table-list	The specified columns from the specified tabledef(s) and/or querydef(s).	`"SELECT CustId , CustName FROM Customer"`

The usual procedure is to pass **NULL** to **Open**; in that case, **Open** calls **GetDefaultSQL**, an overridable member function that ClassWizard generates when creating a **CDaoRecordset**-derived class. This value gives the tabledef(s) and/or querydef name(s) you specified in ClassWizard. You can instead specify other information in the *lpszSQL* parameter.

Whatever you pass, **Open** constructs a final SQL string for the query (the string may have SQL **WHERE** and **ORDER BY** clauses appended to the *lpszSQL* string you passed) and then executes the query. You can examine the constructed string by calling **GetSQL** after calling **Open**.

The field data members of your recordset class are bound to the columns of the data selected. If any records are returned, the first record becomes the current record.

If you want to set options for the recordset, such as a filter or sort, set **m_strSort** or **m_strFilter** after you construct the recordset object but before you call **Open**. If you want to refresh the records in the recordset after the recordset is already open, call **Requery**.

If you call **Open** on a dynaset-type or snapshot-type recordset, or if the data source refers to a SQL statement or a tabledef that represents an attached table, you cannot use **dbOpenTable** for the type argument; if you do, MFC throws an exception. To determine whether a tabledef object represents an attached table, create a CDaoTableDef object and call its GetConnect member function.

Use the **dbSeeChanges** flag if you wish to trap changes made by another user or another program on your machine when you are editing or deleting the same record. For example, if two users start editing the same record, the first user to call the **Update** member function succeeds. When **Update** is called by the second user, a **CDaoException** is thrown. Similarly, if the second user tries to call **Delete** to delete the record, and it has already been changed by the first user, a **CDaoException** occurs.

Typically, if the user gets this **CDaoException** while updating, your code should refresh the contents of the fields and retrieve the newly modified values. If the exception occurs in the process of deleting, your code could display the new record data to the user and a message indicating that the data has recently changed. At this point, your code can request a confirmation that the user still wants to delete the record.

Tip
Use the forward-only scrolling option (**dbForwardOnly**) to improve performance when your application makes a single pass through a recordset opened from an ODBC data source.

For related information, see the topic "OpenRecordset Method" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::Close
CDaoRecordset::CDaoRecordset

Other Resources

CDaoRecordset Members

CDaoRecordset::Requery

Call this member function to rebuild (refresh) a recordset.

```
virtual void Requery( );
```

Remarks

If any records are returned, the first record becomes the current record.

In order for the recordset to reflect the additions and deletions that you or other users are making to the data source, you must rebuild the recordset by calling **Requery**. If the recordset is a dynaset, it automatically reflects updates that you or other users make to its existing records (but not additions). If the recordset is a snapshot, you must call **Requery** to reflect edits by other users as well as additions and deletions.

For either a dynaset or a snapshot, call **Requery** any time you want to rebuild the recordset using parameter values. Set the new filter or sort by setting m_strFilter and m_strSort before calling **Requery**. Set new parameters by assigning new values to parameter data members before calling **Requery**.

If the attempt to rebuild the recordset fails, the recordset is closed. Before you call **Requery**, you can determine whether the recordset can be requeried by calling the CanRestart member function. **CanRestart** does not guarantee that **Requery** will succeed.

Caution
Call **Requery** only after you have called **Open**.

Note
Calling Requery changes DAO bookmarks.

You can't call **Requery** on a dynaset-type or snapshot-type recordset if calling **CanRestart** returns 0, nor can you use it on a table-type recordset.

If both **IsBOF** and **IsEOF** return nonzero after you call **Requery**, the query didn't return any records and the recordset will contain no data.

For related information, see the topic "Requery Method" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::CanRestart

Other Resources

CDaoRecordset Members

CDaoRecordset::Seek

Call this member function to locate the record in an indexed table-type recordset object that satisfies the specified criteria for the current index and make that record the current record.

```
BOOL Seek(
   LPCTSTR lpszComparison,
   COleVariant* pKey1,
   COleVariant* pKey2 = NULL,
   COleVariant* pKey3 = NULL
);
BOOL Seek(
   LPCTSTR lpszComparison,
   COleVariant* pKeyArray,
   WORD nKeys
);
```

Parameters

lpszComparison

One of the following string expressions: "<", "<=", "=", ">=", or ">".

pKey1

A pointer to a COleVariant whose value corresponds to the first field in the index. Required.

pKey2

A pointer to a **COleVariant** whose value corresponds to the second field in the index, if any. Defaults to **NULL**.

pKey3

A pointer to a **COleVariant** whose value corresponds to the third field in the index, if any. Defaults to **NULL**.

pKeyArray

A pointer to an array of variants. The array size corresponds to the number of fields in the index.

nKeys

An integer corresponding to the size of the array, which is the number of fields in the index.

Note
Do not specify wildcards in the keys. Wildcards will cause **Seek** to return no matching records.

Return Value

Nonzero if matching records are found, otherwise 0.

Remarks

Use the second (array) version of **Seek** to handle indexes of four fields or more.

Seek enables high-performance index searching on table-type recordsets. You must set the current index by calling **SetCurrentIndex** before calling **Seek**. If the index identifies a nonunique key field or fields, **Seek** locates the first record that satisfies the criteria. If you do not set an index, an exception is thrown.

Note that if you are not creating a UNICODE recordset, the **COleVariant** objects must be explicitly declared ANSI. This can be done by using the COleVariant::COleVariant**(** *lpszSrc***,** *vtSrc* **)** form of constructor with *vtSrc* set to **VT_BSTRT** (ANSI) or by using the **COleVariant** function SetString**(** *lpszSrc***,** *vtSrc* **)** with *vtSrc* set to **VT_BSTRT**.

When you call **Seek**, you pass one or more key values and a comparison operator ("<", "<=", "=", ">=", or ">"). **Seek** searches through the specified key fields and locates the first record that satisfies the criteria specified by *lpszComparison* and *pKey1*. Once found, **Seek** returns nonzero, and makes that record current. If **Seek** fails to locate a match, **Seek** returns zero, and the current record is undefined. When using DAO directly, you must explicitly check the NoMatch property.

If *lpszComparison* is "=", ">=", or ">", **Seek** starts at the beginning of the index. If *lpszComparison* is "<" or "<=", **Seek** starts at the end of the index and searches backward unless there are duplicate index entries at the end. In this case, **Seek** starts at an arbitrary entry among the duplicate index entries at the end of the index.

There does not have to be a current record when you use **Seek**.

To locate a record in a dynaset-type or snapshot-type recordset that satisfies a specific condition, use the Find operations. To include all records, not just those that satisfy a specific condition, use the Move operations to move from record to record.

You cannot call **Seek** on an attached table of any type because attached tables must be opened as dynaset-type or snapshot-type recordsets. However, if you call **CDaoDatabase::Open** to directly open an installable ISAM database, you can call **Seek** on tables in that database, although the performance may be slow.

For related information, see the topic "Seek Method" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::FindFirst
CDaoRecordset::FindLast
CDaoRecordset::FindNext
CDaoRecordset::FindPrev
CDaoRecordset::Move
CDaoRecordset::MoveFirst
CDaoRecordset::MoveLast
CDaoRecordset::MoveNext
CDaoRecordset::MovePrev
COleVariant::COleVariant
COleVariant::SetString

Other Resources

CDaoRecordset Members

CDaoRecordset::SetAbsolutePosition

Sets the relative record number of a recordset object's current record.

```
void SetAbsolutePosition(
   long lPosition
);
```

Parameters

lPosition

Corresponds to the ordinal position of the current record in the recordset.

Remarks

Calling **SetAbsolutePosition** enables you to position the current record pointer to a specific record based on its ordinal position in a dynaset-type or snapshot-type recordset. You can also determine the current record number by calling GetAbsolutePosition.

Note
This member function is valid only for dynaset-type and snapshot-type recordsets.

The AbsolutePosition property value of the underlying DAO object is zero-based; a setting of 0 refers to the first record in the recordset. Setting a value greater than the number of populated records causes MFC to throw an exception. You can determine the number of populated records in the recordset by calling the **GetRecordCount** member function.

If the current record is deleted, the AbsolutePosition property value is not defined, and MFC throws an exception if it is referenced. New records are added to the end of the sequence.

Note
This property is not intended to be used as a surrogate record number. Bookmarks are still the recommended way of retaining and returning to a given position and are the only way to position the current record across all types of recordset objects that support bookmarks. In particular, the position of a given record changes when record(s) preceding it are deleted. There is also no assurance that a given record will have the same absolute position if the recordset is re-created again because the order of individual records within a recordset is not guaranteed unless it is created with a SQL statement using an **ORDER BY** clause.

For related information, see the topic "AbsolutePosition Property" in DAO Help.

See Also

Reference

CDaoRecordset Class

Hierarchy Chart

CDaoRecordset::GetAbsolutePosition

Other Resources

CDaoRecordset Members

CDaoRecordset::SetBookmark

Call this member function to position the recordset on the record containing the specified bookmark.

```
void SetBookmark(
   COleVariant varBookmark
);
```

Parameters

varBookmark

A COleVariant object containing the bookmark value for a specific record.

Remarks

When a recordset object is created or opened, each of its records already has a unique bookmark. You can retrieve the bookmark for the current record by calling **GetBookmark** and saving the value to a **COleVariant** object. You can later return to that record by calling **SetBookmark** using the saved bookmark value.

Note
Calling Requery changes DAO bookmarks.

Note that if you are not creating a UNICODE recordset, the **COleVariant** object must be explicitly declared ANSI. This can be done by using the COleVariant::COleVariant**(** *lpszSrc***,** *vtSrc* **)** form of constructor with *vtSrc* set to **VT_BSTRT** (ANSI) or by using the **COleVariant** function SetString**(** *lpszSrc***,** *vtSrc* **)** with *vtSrc* set to **VT_BSTRT**.

For related information, see the topics "Bookmark Property" and Bookmarkable Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetBookmark

Other Resources

CDaoRecordset Members

CDaoRecordset::SetCacheSize

Call this member function to set the number of records to be cached.

```
void SetCacheSize(
   long lSize
);
```

Parameters

lSize

Specifies the number of records. A typical value is 100. A setting of 0 turns off caching. The setting must be between 5 and 1200 records. The cache may use a considerable amount of memory.

Remarks

A cache is a space in local memory that holds the data most recently retrieved from the server in the event that the data will be requested again while the application is running. Data caching improves the performance of an application that retrieves data from a remote server through dynaset-type recordset objects. When data is requested, the Microsoft Jet database engine checks the cache for the requested data first rather than retrieving it from the server, which takes more time. Data that does not come from an ODBC data source is not saved in the cache.

Any ODBC data source, such as an attached table, can have a local cache. To create the cache, open a recordset object from the remote data source, call the **SetCacheSize** and **SetCacheStart** member functions, and then call the **FillCache** member function or step through the records by using one of the Move operations. The *lSize* parameter of the **SetCacheSize** member function can be based on the number of records your application can work with at one time. For example, if you are using a recordset as the source of the data to be displayed on screen, you could pass the **SetCacheSize** *lSize* parameter as 20 to display 20 records at one time.

For related information, see the topic "CacheSize, CacheStart Properties" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::FillCache
CDaoRecordset::GetCacheSize
CDaoRecordset::GetCacheStart
CDaoRecordset::SetCacheStart

Other Resources

CDaoRecordset Members

CDaoRecordset::SetCacheStart

Call this member function to specify the bookmark of the first record in the recordset to be cached.

```
void SetCacheStart(
   COleVariant varBookmark
);
```

Parameters

varBookmark

A COleVariant that specifies the bookmark of the first record in the recordset to be cached.

Remarks

You can use the bookmark value of any record for the *varBookmark* parameter of the **SetCacheStart** member function. Make the record you want to start the cache with the current record, establish a bookmark for that record using SetBookmark, and pass the bookmark value as the parameter for the **SetCacheStart** member function.

The Microsoft Jet database engine requests records within the cache range from the cache, and it requests records outside the cache range from the server.

Records retrieved from the cache do not reflect changes made concurrently to the source data by other users.

To force an update of all the cached data, pass the *lSize* parameter of **SetCacheSize** as 0, call **SetCacheSize** again with the size of the cache you originally requested, and then call the **FillCache** member function.

Note that if you are not creating a UNICODE recordset, the **COleVariant** object must be explicitly declared ANSI. This can be done by using the COleVariant::COleVariant**(** *lpszSrc***,** *vtSrc* **)** form of constructor with *vtSrc* set to **VT_BSTRT** (ANSI) or by using the **COleVariant** function SetString**(** *lpszSrc***,** *vtSrc* **)** with *vtSrc* set to **VT_BSTRT**.

For related information, see the topic CacheSize, CacheStart Properties" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::FillCache
CDaoRecordset::GetCacheSize
CDaoRecordset::GetCacheStart
CDaoRecordset::SetCacheSize

Other Resources

CDaoRecordset Members

CDaoRecordset::SetCurrentIndex

Call this member function to set an index on a table-type recordset.

```
void SetCurrentIndex(
   LPCTSTR lpszIndex
);
```

Parameters

lpszIndex

A pointer containing the name of the index to be set.

Remarks

Records in base tables are not stored in any particular order. Setting an index changes the order of records returned from the database, but it does not affect the order in which the records are stored. The specified index must already be defined. If you try to use an index object that does not exist, or if the index is not set when you call Seek, MFC throws an exception.

You can create a new index for the table by calling CDaoTableDef::CreateIndex and appending the new index to the Indexes collection of the underlying tabledef by calling CDaoTableDef::Append, and then reopening the recordset.

Records returned from a table-type recordset can be ordered only by the indexes defined for the underlying tabledef. To sort records in some other order, you can open a dynaset-type or snapshot-type recordset using a SQL **ORDER BY** clause stored in CDaoRecordset::m_strSort.

For related information, see the topic "Index Object" and the definition "current index" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetCurrentIndex

Other Resources

CDaoRecordset Members

CDaoRecordset::SetFieldDirty

Call this member function to flag a field data member of the recordset as changed or as unchanged.

```
void SetFieldDirty(
   void* pv,
   BOOL bDirty = TRUE
);
```

Parameters

pv

Contains the address of a field data member in the recordset or **NULL**. If **NULL**, all field data members in the recordset are flagged. (C++ **NULL** is not the same as Null in database terminology, which means "having no value.")

bDirty

TRUE if the field data member is to be flagged as "dirty" (changed). Otherwise **FALSE** if the field data member is to be flagged as "clean" (unchanged).

Remarks

Marking fields as unchanged ensures the field is not updated.

The framework marks changed field data members to ensure they will be written to the record on the data source by the DAO record field exchange (DFX) mechanism. Changing the value of a field generally sets the field dirty automatically, so you will seldom need to call **SetFieldDirty** yourself, but you might sometimes want to ensure that columns will be explicitly updated or inserted regardless of what value is in the field data member. The DFX mechanism also employs the use of **PSEUDO NULL**. For more information, see CDaoFieldExchange::m_nOperation.

If the double-buffering mechanism is not being used, then changing the value of the field does not automatically set the field as dirty. In this case, it will be necessary to explicitly set the field as dirty. The flag contained in m_bCheckCacheForDirtyFields controls this automatic field checking.

Note
Call this member function only after you have called Edit or AddNew.

Using **NULL** for the first argument of the function will apply the function to all **outputColumn** fields, not **param** fields in **CDaoFieldExchange**. For instance, the call

```
SetFieldDirty( NULL );
```

will set only **outputColumn** fields to **NULL**; **param** fields will be unaffected.

To work on a **param**, you must supply the actual address of the individual **param** you want to work on, such as:

```
SetFieldDirty( &m_strParam );
```

This means you cannot set all **param** fields to **NULL**, as you can with **outputColumn** fields.

SetFieldDirty is implemented through **DoFieldExchange**.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::SetFieldNull
CDaoRecordset::SetFieldValue

Other Resources

CDaoRecordset Members

CDaoRecordset::SetFieldNull

Call this member function to flag a field data member of the recordset as Null (specifically having no value) or as non-Null.

```
void SetFieldNull(
   void* pv,
   BOOL bNull = TRUE
);
```

Parameters

pv

Contains the address of a field data member in the recordset or **NULL**. If **NULL**, all field data members in the recordset are flagged. (C++ **NULL** is not the same as Null in database terminology, which means "having no value.")

bNull

Nonzero if the field data member is to be flagged as having no value (Null). Otherwise 0 if the field data member is to be flagged as non-Null.

Remarks

SetFieldNull is used for fields bound in the **DoFieldExchange** mechanism.

When you add a new record to a recordset, all field data members are initially set to a Null value and flagged as "dirty" (changed). When you retrieve a record from a data source, its columns either already have values or are Null. If it is not appropriate to make a field Null, a CDaoException is thrown.

If you are using the double-buffering mechanism, for example, if you specifically wish to designate a field of the current record as not having a value, call **SetFieldNull** with *bNull* set to **TRUE** to flag it as Null. If a field was previously marked Null and you now want to give it a value, simply set its new value. You do not have to remove the Null flag with **SetFieldNull**. To determine whether the field is allowed to be Null, call IsFieldNullable.

If you are not using the double-buffering mechanism, then changing the value of the field does not automatically set the field as dirty and non-Null. You must specifically set the fields dirty and non-Null. The flag contained in m_bCheckCacheForDirtyFields controls this automatic field checking.

The DFX mechanism employs the use of **PSEUDO NULL**. For more information, see CDaoFieldExchange::m_nOperation.

Note
Call this member function only after you have called Edit or AddNew.

Using **NULL** for the first argument of the function will apply the function only to **outputColumn** fields, not **param** fields in **CDaoFieldExchange**. For instance, the call

```
SetFieldNull( NULL );
```

will set only **outputColumn** fields to **NULL**; **param** fields will be unaffected.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::SetParamValue

Other Resources

CDaoRecordset Members

CDaoRecordset::SetFieldValue

Call this member function to set the value of a field, either by ordinal position or by changing the value of the string.

```
virtual void SetFieldValue(
   LPCTSTR lpszName,
   const COleVariant& varValue
);
virtual void SetFieldValue(
   int nIndex,
   const COleVariant& varValue
);
void SetFieldValue(
   LPCTSTR lpszName,
   LPCTSTR lpszValue
);
void SetFieldValue(
   int nIndex,
   LPCTSTR lpszValue
);
```

Parameters

lpszName

A pointer to a string containing the name of a field.

varValue

A reference to a COleVariant object containing the value of the field's contents.

nIndex

An integer that represents the ordinal position of the field in the recordset's Fields collection (zero-based).

lpszValue

A pointer to a string containing the value of the field's contents.

Remarks

Use **SetFieldValue** and GetFieldValue to dynamically bind fields at run time rather than statically binding columns using the DoFieldExchange mechanism.

Note that if you are not creating a UNICODE recordset, you must either use a form of **SetFieldValue** that does not contain a **COleVariant** parameter, or the **COleVariant** object must be explicitly declared ANSI. This can be done by using the COleVariant::COleVariant**(** *lpszSrc*, *vtSrc* **)** form of constructor with *vtSrc* set to **VT_BSTRT** (ANSI) or by using the **COleVariant** function SetString**(** *lpszSrc*, *vtSrc* **)** with *vtSrc* set to **VT_BSTRT**.

For related information, see the topics "Field Object" and "Value Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetFieldValue
CDaoRecordset::m_nParams
CDaoRecordset::SetFieldValueNull
COleVariant::COleVariant
COleVariant::SetString

Other Resources

CDaoRecordset Members

CDaoRecordset::SetFieldValueNull

Call this member function to set the field to a Null value.

```
void SetFieldValueNull(
   int nIndex
);
void SetFieldValueNull(
   LPCTSTR lpszName
);
```

Parameters

nIndex

The index of the field in the recordset, for lookup by zero-based index.

lpszName

The name of the field in the recordset, for lookup by name.

Remarks

C++ **NULL** is not the same as Null, which, in database terminology, means "having no value."

For related information, see the topics "Field Object" and "Value Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::SetFieldValue

Other Resources

CDaoRecordset Members

CDaoRecordset::SetLockingMode

Call this member function to set the type of locking for the recordset.

```
void SetLockingMode(
   BOOL bPessimistic
);
```

Parameters

bPessimistic

A flag that indicates the type of locking.

Remarks

When pessimistic locking is in effect, the 2K page containing the record you are editing is locked as soon as you call the **Edit** member function. The page is unlocked when you call the **Update** or **Close** member function or any of the Move or Find operations.

When optimistic locking is in effect, the 2K page containing the record is locked only while the record is being updated with the **Update** member function.

If a page is locked, no other user can edit records on the same page. If you call **SetLockingMode** and pass a nonzero value and another user already has the page locked, an exception is thrown when you call **Edit**. Other users can read data from locked pages.

If you call **SetLockingMode** with a zero value and later call **Update** while the page is locked by another user, an exception occurs. To see the changes made to your record by another user (and lose your changes), call the **SetBookmark** member function with the bookmark value of the current record.

When working with ODBC data sources, the locking mode is always optimistic.

See Also

Reference

CDaoRecordset Class

Hierarchy Chart

CDaoRecordset::GetLockingMode

Other Resources

CDaoRecordset Members

CDaoRecordset::SetParamValue

Call this member function to set the value of a parameter in the recordset at run time.

```
virtual void SetParamValue(
   int nIndex,
   const COleVariant& varValue
);
virtual void SetParamValue(
   LPCTSTR lpszName,
   const COleVariant& varValue
);
```

Parameters

nIndex

The numerical position of the parameter in the querydef's Parameters collection.

var

The value to set; see Remarks.

lpszName

The name of the parameter whose value you want to set.

Remarks

The parameter must already have been established as part of the recordset's SQL string. You can access the parameter either by name or by its index position in the collection.

Specify the value to set as a **COleVariant** object. For information about setting the desired value and type in your **COleVariant** object, see class COleVariant. Note that if you are not creating a UNICODE recordset, the **COleVariant** object must be explicitly declared ANSI. This can be done by using the COleVariant::COleVariant**(** *lpszSrc***,** *vtSrc* **)** form of constructor with *vtSrc* set to **VT_BSTRT** (ANSI) or by using the **COleVariant** function SetString**(** *lpszSrc***,** *vtSrc* **)** with *vtSrc* set to **VT_BSTRT**.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::GetParamValue
CDaoRecordset::m_nParams
CDaoRecordset::SetParamValueNull

Other Resources

CDaoRecordset Members

CDaoRecordset::SetParamValueNull

Call this member function to set the parameter to a Null value.

```
void SetParamValueNull(
   int nIndex
);
void SetParamValueNull(
   LPCTSTR lpszName
);
```

Parameters

nIndex

The index of the field in the recordset, for lookup by zero-based index.

lpszName

The name of the field in the recordset, for lookup by name.

Remarks

C++ **NULL** is not the same as Null, which, in database terminology, means "having no value."

See Also

Reference

CDaoRecordset Class

Hierarchy Chart

Other Resources

CDaoRecordset Members

CDaoRecordset::SetPercentPosition

Call this member function to set a value that changes the approximate location of the current record in the recordset object based on a percentage of the records in the recordset.

```
void SetPercentPosition(
   float fPosition
);
```

Parameters

fPosition

A number between 0 and 100.

Remarks

When working with a dynaset-type or snapshot-type recordset, first populate the recordset by moving to the last record before you call **SetPercentPosition**. If you call **SetPercentPosition** before fully populating the recordset, the amount of movement is relative to the number of records accessed as indicated by the value of GetRecordCount. You can move to the last record by calling **MoveLast**.

Once you call **SetPercentPosition**, the record at the approximate position corresponding to that value becomes current.

Note
Calling **SetPercentPosition** to move the current record to a specific record in a recordset is not recommended. Call the SetBookmark member function instead.

For related information, see the topic "PercentPosition Property" in DAO Help.

See Also

Reference

CDaoRecordset Class

Hierarchy Chart

CDaoRecordset::GetPercentPosition

Other Resources

CDaoRecordset Members

CDaoRecordset::Update

Call this member function after a call to the **AddNew** or **Edit** member function.

```
virtual void Update( );
```

Remarks

This call is required to complete the **AddNew** or **Edit** operation.

Both **AddNew** and **Edit** prepare an edit buffer in which the added or edited data is placed for saving to the data source. **Update** saves the data. Only those fields marked or detected as changed are updated.

If the data source supports transactions, you can make the **Update** call (and its corresponding **AddNew** or **Edit** call) part of a transaction.

Caution

If you call **Update** without first calling either **AddNew** or **Edit**, **Update** throws a **CDaoException**. If you call **AddNew** or **Edit**, you must call **Update** before you call MoveNext or close either the recordset or the data source connection. Otherwise, your changes are lost without notification.

When the recordset object is pessimistically locked in a multiuser environment, the record remains locked from the time **Edit** is used until the updating is complete. If the recordset is optimistically locked, the record is locked and compared with the pre-edited record just before it is updated in the database. If the record has changed since you called **Edit**, the **Update** operation fails and MFC throws an exception. You can change the locking mode with **SetLockingMode**.

Note

Optimistic locking is always used on external database formats, such as ODBC and installable ISAM.

For related information, see the topics "AddNew Method", "CancelUpdate Method", "Delete Method", "LastModified Property", "Update Method", and "EditMode Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::AddNew
CDaoRecordset::CancelUpdate
CDaoRecordset::Delete
CDaoRecordset::Edit
CDaoRecordset::CanTransact

Other Resources

CDaoRecordset Members

CDaoRecordset Data Members

For information about the data members in **CDaoRecordset**, see CDaoRecordset Members.

CDaoRecordset::m_bCheckCacheForDirtyFields

Contains a flag indicating whether cached fields are automatically marked as dirty (changed) and Null.

Remarks

The flag defaults to **TRUE**. The setting in this data member controls the entire double-buffering mechanism. If you set the flag to **TRUE**, you can turn off the caching on a field-by-field basis using the DFX mechanism. If you set the flag to **FALSE**, you must call **SetFieldDirty** and **SetFieldNull** yourself.

Set this data member before calling **Open**. This mechanism is primarily for ease-of-use. Performance may be slower because of the double-buffering of fields as changes are made.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::SetFieldNull
CDaoRecordset::IsFieldNull
CDaoRecordset::IsFieldDirty
CDaoRecordset::SetFieldDirty

Other Resources

CDaoRecordset Members

CDaoRecordset::m_nFields

Contains the number of field data members in the recordset class and the number of columns selected by the recordset from the data source.

Remarks

The constructor for the recordset class must initialize **m_nFields** with the correct number of statically bound fields. ClassWizard writes this initialization for you when you use it to declare your recordset class. You can also write it manually.

The framework uses this number to manage interaction between the field data members and the corresponding columns of the current record on the data source.

Note
This number must correspond to the number of output columns registered in **DoFieldExchange** after a call to **SetFieldType** with the parameter **CDaoFieldExchange::outputColumn**.

You can bind columns dynamically by way of **CDaoRecordset::GetFieldValue** and **CDaoRecordset::SetFieldValue**. If you do so, you do not need to increment the count in **m_nFields** to reflect the number of DFX function calls in your **DoFieldExchange** member function.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::SetFieldValue
CDaoRecordset::GetFieldValue

Other Resources

CDaoRecordset Members

CDaoRecordset::m_nParams

Contains the number of parameter data members in the recordset class — the number of parameters passed with the recordset's query.

Remarks

If your recordset class has any parameter data members, the constructor for the class must initialize **m_nParams** with the correct number. The value of **m_nParams** defaults to 0. If you add parameter data members — which you must do manually — you must also manually add an initialization in the class constructor to reflect the number of parameters (which must be at least as large as the number of '?' placeholders in your **m_strFilter** or **m_strSort** string).

The framework uses this number when it parameterizes the recordset's query.

Note
This number must correspond to the number of "params" registered in **DoFieldExchange** after a call to **SetFieldType** with the parameter **CFieldExchange::param**.

For related information, see the topic "Parameter Object" in DAO Help.

See Also

Reference

CDaoRecordset Class

Hierarchy Chart

Other Resources

CDaoRecordset Members

CDaoRecordset::m_pDAORecordset

Contains a pointer to the OLE interface for the DAO recordset object underlying the **CDaoRecordset** object.

Remarks

Use this pointer if you need to access the DAO interface directly.

For related information, see the topic "Recordset Object" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::m_pDatabase

Other Resources

CDaoRecordset Members

CDaoRecordset::m_pDatabase

Contains a pointer to the **CDaoDatabase** object through which the recordset is connected to a data source.

Remarks

This variable is set in two ways. Typically, you pass a pointer to an already open **CDaoDatabase** object when you construct the recordset object. If you pass **NULL** instead, **CDaoRecordset** creates a **CDaoDatabase** object for you and opens it. In either case, **CDaoRecordset** stores the pointer in this variable.

Normally you will not directly need to use the pointer stored in **m_pDatabase**. If you write your own extensions to **CDaoRecordset**, however, you might need to use the pointer. For example, you might need the pointer if you throw your own **CDaoException**(s).

For related information, see the topic "Database Object" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::m_pDAORecordset

Other Resources

CDaoRecordset Members

CDaoRecordset::m_strFilter

Contains a string that is used to construct the **WHERE** clause of a SQL statement.

Remarks

It does not include the reserved word **WHERE** to filter the recordset. The use of this data member is not applicable to table-type recordsets. The use of **m_strFilter** has no effect when opening a recordset using a **CDaoQueryDef** pointer.

Use the U.S. date format (month-day-year) when you filter fields containing dates, even if you are not using the U.S. version of the Microsoft Jet database engine; otherwise, the data may not be filtered as you expect.

For related information, see the topic "Filter Property" in DAO Help.

See Also

Reference

CDaoRecordset Class

Hierarchy Chart

CDaoRecordset::m_strSort

Other Resources

CDaoRecordset Members

CDaoRecordset::m_strSort

Contains a string containing the **ORDER BY** clause of a SQL statement without the reserved words **ORDER BY**.

Remarks

You can sort on dynaset- and snapshot-type recordset objects.

You cannot sort table-type recordset objects. To determine the sort order of a table-type recordset, call SetCurrentIndex.

The use of **m_strSort** has no effect when opening a recordset using a **CDaoQueryDef** pointer.

For related information, see the topic "Sort Property" in DAO Help.

See Also

Reference

CDaoRecordset Class
Hierarchy Chart
CDaoRecordset::m_strFilter

Other Resources

CDaoRecordset Members

CDaoRecordView Class

A view that displays database records in controls.

```
class AFX_NOVTABLE CDaoRecordView : public CFormView
```

Remarks

The view is a form view directly connected to a **CDaoRecordset** object. The view is created from a dialog template resource and displays the fields of the **CDaoRecordset** object in the dialog template's controls. The **CDaoRecordView** object uses dialog data exchange (DDX) and DAO record field exchange (DFX) to automate the movement of data between the controls on the form and the fields of the recordset. **CDaoRecordView** also supplies a default implementation for moving to the first, next, previous, or last record and an interface for updating the record currently in view.

Note
The DAO database classes are distinct from the MFC database classes based on Open Database Connectivity (ODBC). All DAO database class names have the "CDao" prefix. You can still access ODBC data sources with the DAO classes; the DAO classes generally offer superior capabilities because they use the Microsoft Jet database engine.

The most common way to create your record view is with the Application Wizard. The Application Wizard creates both the record view class and its associated recordset class as part of your skeleton starter application.

If you simply need a single form, the Application Wizard approach is easier. ClassWizard lets you decide to use a record view later in the development process. If you don't create the record view class with the Application Wizard, you can create it later with ClassWizard. Using ClassWizard to create a record view and a recordset separately and then connect them is the most flexible approach because it gives you more control in naming the recordset class and its .H/.CPP files. This approach also lets you have multiple record views on the same recordset class.

To make it easy for end-users to move from record to record in the record view, the Application Wizard creates menu (and optionally toolbar) resources for moving to the first, next, previous, or last record. If you create a record view class with ClassWizard, you need to create these resources yourself with the menu and bitmap editors.

For information about the default implementation for moving from record to record, see **IsOnFirstRecord** and **IsOnLastRecord** and the article Using a Record View, which applies to both **CRecordView** and **CDaoRecordView**.

CDaoRecordView keeps track of the user's position in the recordset so that the record view can update the user interface. When the user moves to either end of the recordset, the record view disables user interface objects — such as menu items or toolbar buttons — for moving further in the same direction.

For more information about declaring and using your record view and recordset classes, see "Designing and Creating a Record View" in the article Record Views. For more information about how record views work and how to use them, see the article Using a Record View. All the articles mentioned above apply to both **CRecordView** and **CDaoRecordView**.

Requirements

Header: afxdao.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CFormView Class
Hierarchy Chart
CDaoRecordset Class
CDaoTableDef Class
CDaoQueryDef Class
CDaoDatabase Class
CDaoWorkspace Class
CFormView Class

Other Resources

CDaoRecordView Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CView Members

CScrollView Members

CFormView Members

Construction

CDaoRecordView	Constructs a **CDaoRecordView** object.

Attributes

IsOnFirstRecord	Returns nonzero if the current record is the first record in the associated recordset.
IsOnLastRecord	Returns nonzero if the current record is the last record in the associated recordset.
OnGetRecordset	Returns a pointer to an object of a class derived from **CDaoRecordset**. ClassWizard overrides this function for you and creates the recordset if necessary.

Operations

OnMove	If the current record has changed, updates it on the data source, then moves to the specified record (next, previous, first, or last).

See Also

Reference

CDaoRecordView Class

Hierarchy Chart

CDaoRecordView Member Functions

For information about the member functions in **CDaoRecordView**, see CDaoRecordView Members.

CDaoRecordView::CDaoRecordView

When you create an object of a type derived from **CDaoRecordView**, call either form of the constructor to initialize the view object and identify the dialog resource on which the view is based.

```
explicit CDaoRecordView(
   LPCTSTR lpszTemplateName
);
explicit CDaoRecordView(
   UINT nIDTemplate
);
```

Parameters

lpszTemplateName

Contains a null-terminated string that is the name of a dialog template resource.

nIDTemplate

Contains the ID number of a dialog template resource.

Remarks

You can either identify the resource by name (pass a string as the argument to the constructor) or by its ID (pass an unsigned integer as the argument). Using a resource ID is recommended.

Note

Your derived class must supply its own constructor. In the constructor of your derived class, call the constructor **CDaoRecordView::CDaoRecordView** with the resource name or ID as an argument.

CDaoRecordView::OnInitialUpdate calls **CWnd::UpdateData**, which calls **CWnd::DoDataExchange**. This initial call to **DoDataExchange** connects **CDaoRecordView** controls (indirectly) to **CDaoRecordset** field data members created by ClassWizard. These data members cannot be used until after you call the base class **CFormView::OnInitialUpdate** member function.

Note

If you use ClassWizard, the wizard defines an **enum** value `CDaoRecordView::IDD` and specifies it in the member initialization list for the constructor where you see `IDD_MYFORM`.

```
CMyRecordView::CMyRecordView()

   : CDaoRecordView( IDD_MYFORM )
{
   //{{AFX_DATA_INIT( CMyRecordView )
      // NOTE: the ClassWizard will add member initialization here
   //}}AFX_DATA_INIT
   // Other construction code, such as data initialization
}
```

See Also

Reference

CDaoRecordView Class
Hierarchy Chart
CWnd::UpdateData
CWnd::DoDataExchange

Other Resources

CDaoRecordView Members

CDaoRecordView::IsOnFirstRecord

Call this member function to determine whether the current record is the first record in the recordset object associated with this record view.

```
BOOL IsOnFirstRecord( );
```

Return Value

Nonzero if the current record is the first record in the recordset; otherwise 0.

Remarks

This function is useful for writing your own implementations of the default command update handlers written by ClassWizard.

If the user moves to the first record, the framework disables any user interface objects (for example, menu items or toolbar buttons) you have for moving to the first or the previous record.

See Also

Reference

CDaoRecordView Class
Hierarchy Chart
CDaoRecordView::IsOnLastRecord

Other Resources

CDaoRecordView Members

CDaoRecordView::IsOnLastRecord

Call this member function to determine whether the current record is the last record in the recordset object associated with this record view.

```
BOOL IsOnLastRecord( );
```

Return Value

Nonzero if the current record is the last record in the recordset; otherwise 0.

Remarks

This function is useful for writing your own implementations of the default command update handlers that ClassWizard writes to support a user interface for moving from record to record.

Caution
The result of this function is reliable except that the view may not be able to detect the end of the recordset until the user has moved past it. The user might have to move beyond the last record before the record view can tell that it must disable any user interface objects for moving to the next or last record. If the user moves past the last record and then moves back to the last record (or before it), the record view can track the user's position in the recordset and disable user interface objects correctly.

See Also

Reference

CDaoRecordView Class

Hierarchy Chart

CDaoRecordView::IsOnFirstRecord

Other Resources

CDaoRecordView Members

CDaoRecordView::OnGetRecordset

Returns a pointer to the **CDaoRecordset**-derived object associated with the record view.

```
virtual CDaoRecordset* OnGetRecordset( ) = 0;
```

Return Value

A pointer to a **CDaoRecordset**-derived object if the object was successfully created; otherwise a **NULL** pointer.

Remarks

You must override this member function to construct or obtain a recordset object and return a pointer to it. If you declare your record view class with ClassWizard, the wizard writes a default override for you. ClassWizard's default implementation returns the recordset pointer stored in the record view if one exists. If not, it constructs a recordset object of the type you specified with ClassWizard and calls its **Open** member function to open the table or run the query, and then returns a pointer to the object.

For more information and examples, see the article Record Views: Using a Record View.

See Also

Reference

CDaoRecordView Class
Hierarchy Chart
CDaoRecordset Class
CDaoRecordset::Open

Other Resources

CDaoRecordView Members

CDaoRecordView::OnMove

Call this member function to move to a different record in the recordset and display its fields in the controls of the record view.

```
virtual BOOL OnMove(
   UINT nIDMoveCommand
);
```

Parameters

nIDMoveCommand

One of the following standard command ID values:

- **ID_RECORD_FIRST** Move to the first record in the recordset.
- **ID_RECORD_LAST** Move to the last record in the recordset.
- **ID_RECORD_NEXT** Move to the next record in the recordset.
- **ID_RECORD_PREV** Move to the previous record in the recordset.

Return Value

Nonzero if the move was successful; otherwise 0 if the move request was denied.

Remarks

The default implementation calls the appropriate Move member function of the **CDaoRecordset** object associated with the record view.

By default, **OnMove** updates the current record on the data source if the user has changed it in the record view.

The Application Wizard creates a menu resource with First Record, Last Record, Next Record, and Previous Record menu items. If you select the Initial Toolbar option, the Application Wizard also creates a toolbar with buttons corresponding to these commands.

If you move past the last record in the recordset, the record view continues to display the last record. If you move backward past the first record, the record view continues to display the first record.

Caution
Calling **OnMove** throws an exception if the recordset has no records. Call the appropriate user interface update handler function — **OnUpdateRecordFirst**, **OnUpdateRecordLast**, **OnUpdateRecordNext**, or **OnUpdateRecordPrev** — before the corresponding move operation to determine whether the recordset has any records.

See Also

Reference

CDaoRecordView Class
Hierarchy Chart
CDaoRecordset::Move

Other Resources

CDaoRecordView Members

CDaoTableDef Class

Represents the stored definition of a base table or an attached table.

```
class CDaoTableDef : public CObject
```

Remarks

Each DAO database object maintains a collection, called TableDefs, that contains all saved DAO tabledef objects.

You manipulate a table definition using a **CDaoTableDef** object. For example, you can:

- Examine the field and index structure of any local, attached, or external table in a database.
- Call the **SetConnect** and **SetSourceTableName** member functions for attached tables, and use the **RefreshLink** member function to update connections to attached tables.
- Call the **CanUpdate** member function to determine if you can edit field definitions in the table.
- Get or set validation conditions using the **GetValidationRule** and **SetValidationRule**, and the **GetValidationText** and **SetValidationText** member functions.
- Use the **Open** member function to create a table-, dynaset-, or snapshot-type **CDaoRecordset** object.

Note
The DAO database classes are distinct from the MFC database classes based on Open Database Connectivity (ODBC). All DAO database class names have the "CDao" prefix. You can still access ODBC data sources with the DAO classes; the DAO classes generally offer superior capabilities because they are specific to the Microsoft Jet database engine.

To use tabledef objects either to work with an existing table or to create a new table

1. In all cases, first construct a **CDaoTableDef** object, supplying a pointer to a CDaoDatabase object to which the table belongs.
2. Then do the following, depending on what you want:
   - To use an existing saved table, call the tabledef object's Open member function, supplying the name of the saved table.
   - To create a new table, call the tabledef object's Create member function, supplying the name of the table. Call CreateField and CreateIndex to add fields and indexes to the table.
   - Call Append to save the table by appending it to the database's TableDefs collection. **Create** puts the tabledef into an open state, so after calling **Create** you do not call **Open**.

Tip
The easiest way to create saved tables is to create them and store them in your database using Microsoft Access. Then you can open and use them in your MFC code.

To use the tabledef object you have opened or created, create and open a **CDaoRecordset** object, specifying the name of the tabledef with a **dbOpenTable** value in the *nOpenType* parameter.

To use a tabledef object to create a **CDaoRecordset** object, you typically create or open a tabledef as described above, then construct a recordset object, passing a pointer to your tabledef object when you call CDaoRecordset::Open. The tabledef you pass must be in an open state. For more information, see class CDaoRecordset.

When you finish using a tabledef object, call its Close member function; then destroy the tabledef object.

Requirements

Header: afxdao.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class
Hierarchy Chart
CDaoDatabase Class
CDaoRecordset Class

Other Resources

CDaoTableDef Members

CDaoTableDef Members

Base Class Members

CObject Members

Data Members

m_pDAOTableDef	A pointer to the DAO interface underlying the tabledef object.
m_pDatabase	Source database for this table.

Construction

Append	Adds a new table to the database.
CDaoTableDef	Constructs a **CDaoTableDef** object.
Close	Closes an open tabledef.
Create	Creates a table which can be added to the database using Append.
Open	Opens an existing tabledef stored in the database's TableDef's collection.

Attributes

CanUpdate	Returns nonzero if the table can be updated (you can modify the definition of fields or the table properti es).
GetAttributes	Returns a value that indicates one or more characteristics of a **CDaoTableDef** object.
GetConnect	Returns a value that provides information about the source of a table.
GetDateCreated	Returns the date and time the base table underlying a **CDaoTableDef** object was created.
GetDateLastUpdated	Returns the date and time of the most recent change made to the design of the base table.
GetFieldCount	Returns a value that represents the number of fields in the table.
GetFieldInfo	Returns specific kinds of information about the fields in the table.
GetIndexCount	Returns the number of indexes for the table.
GetIndexInfo	Returns specific kinds of information about the indexes for the table.
GetName	Returns the user-defined name of the table.
GetRecordCount	Returns the number of records in the table.
GetSourceTableName	Returns a value that specifies the name of the attached table in the source database.
GetValidationRule	Returns a value that validates the data in a field as it is changed or added to a table.
GetValidationText	Returns a value that specifies the text of the message that your application displays if the value of a Fiel d object does not satisfy the specified validation rule.

IsOpen	Returns nonzero if the table is open.
SetAttributes	Sets a value that indicates one or more characteristics of a **CDaoTableDef** object.
SetConnect	Sets a value that provides information about the source of a table.
SetName	Sets the name of the table.
SetSourceTableName	Sets a value that specifies the name of an attached table in the source database.
SetValidationRule	Sets a value that validates the data in a field as it is changed or added to a table.
SetValidationText	Sets a value that specifies the text of the message that your application displays if the value of a Field object does not satisfy the specified validation rule.

Operations

CreateField	Called to create a field for a table.
CreateIndex	Called to create an index for a table.
DeleteField	Called to delete a field from a table.
DeleteIndex	Called to delete an index from a table.
RefreshLink	Updates the connection information for an attached table.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart

CDaoTableDef Member Functions

For information about the member functions in **CDaoTableDef**, see CDaoTableDef Members.

CDaoTableDef::Append

Call this member function after you call Create to create a new tabledef object to save the tabledef in the database.

```
virtual void Append( );
```

Remarks

The function appends the object to the database's TableDefs collection. You can use the tabledef as a temporary object while defining it by not appending it, but if you want to save and use it, you must call **Append**.

Note
If you attempt to append an unnamed tabledef (containing a null or empty string), MFC throws an exception.

For related information, see the topic "Append Method" in DAO Help.

See Also

Reference

CDaoTableDef Class

Hierarchy Chart

CDaoTableDef::Create

Other Resources

CDaoTableDef Members

CDaoTableDef::CanUpdate

Call this member function to determine whether the definition of the table underlying a **CDaoTableDef** object can be changed.

```
BOOL CanUpdate( );
```

Return Value

Nonzero if the table structure (schema) can be modified (add or delete fields and indexes), otherwise 0.

Remarks

By default, a newly created table underlying a **CDaoTableDef** object can be updated, and an attached table underlying a **CDaoTableDef** object cannot be updated. A **CDaoTableDef** object may be updatable, even if the resulting recordset is not updatable.

For related information, see the topic "Updatable Property" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::GetDateLastUpdated

Other Resources

CDaoTableDef Members

CDaoTableDef::CDaoTableDef

Constructs a **CDaoTableDef** object.

```
CDaoTableDef(
   CDaoDatabase* pDatabase
);
```

Parameters

pDatabase

A pointer to a CDaoDatabase object.

Remarks

After constructing the object, you must call the Create or Open member function. When you finish with the object, you must call its Close member function and destroy the **CDaoTableDef** object.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::Open
CDaoTableDef::Close
CDaoTableDef::Create
CDaoDatabase Class

Other Resources

CDaoTableDef Members

CDaoTableDef::Close

Call this member function to close and release the tabledef object.

```
virtual void Close( );
```

Remarks

Usually after calling **Close**, you delete the tabledef object if it was allocated with **new**.

You can call Open again after calling **Close**. This lets you reuse the tabledef object.

For related information, see the topic "Close Method" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::Open
CDaoTableDef::Create

Other Resources

CDaoTableDef Members

CDaoTableDef::Create

Call this member function to create a new saved table.

```
virtual void Create(
   LPCTSTR lpszName,
   long lAttributes = 0,
   LPCTSTR lpszSrcTable = NULL,
   LPCTSTR lpszConnect = NULL
);
```

Parameters

lpszName

A pointer to a string containing the name of the table.

lAttributes

A value corresponding to characteristics of the table represented by the tabledef object. You can use the bitwise-OR to combine any of the following constants:

Constant	Description
dbAttachExclusive	For databases that use the Microsoft Jet database engine, indicates the table is an attached table opened for exclusive use.
dbAttachSavePWD	For databases that use the Microsoft Jet database engine, indicates that the user ID and password for the attached table are saved with the connection information.
dbSystemObject	Indicates the table is a system table provided by the Microsoft Jet database engine.
dbHiddenObject	Indicates the table is a hidden table provided by the Microsoft Jet database engine.

lpszSrcTable

A pointer to a string containing the source table name. By default this value is initialized as **NULL**.

lpszConnect

A pointer to a string containing the default connection string. By default this value is initialized as **NULL**.

Remarks

Once you have named the tabledef, you can then call Append to save the tabledef in the database's TableDefs collection. After calling **Append**, the tabledef is in an open state, and you can use it to create a CDaoRecordset object.

For related information, see the topic "CreateTableDef Method" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::Open
CDaoTableDef::Close
CDaoRecordset Class

Other Resources

CDaoTableDef Members

CDaoTableDef::CreateField

Call this member function to add a field to the table.

```
void CreateField(
   LPCTSTR lpszName,
   short nType,
   long lSize,
   long lAttributes = 0
);
void CreateField(
   CDaoFieldInfo& fieldinfo
);
```

Parameters

lpszName

A pointer to a string expression specifying the name of this field.

nType

A value indicating the data type of the field. The setting can be one of these values:

Type	Size (bytes)	Description
dbBoolean	1 byte	BOOL
dbByte	1	BYTE
dbInteger	2	int
dbLong	4	long
dbCurrency	8	Currency (COleCurrency)
dbSingle	4	float
dbDouble	8	double
dbDate	8	Date/Time (COleDateTime)
dbText	1 – 255	Text (CString)
dbLongBinary	0	Long Binary (OLE Object), CLongBinary or CByteArray
dbMemo	0	Memo (CString)

lSize

A value that indicates the maximum size, in bytes, of a field that contains text, or the fixed size of a field that contains text or numeric values. The *lSize* parameter is ignored for all but text fields.

lAttributes

A value corresponding to characteristics of the field and that can be combined using a bitwise-OR.

Constant	Description

dbFixedField	The field size is fixed (default for Numeric fields).
dbVariableField	The field size is variable (Text fields only).
dbAutoIncrField	The field value for new records is automatically incremented to a unique long integer that cannot be changed. Only supported for Microsoft Jet database tables.
dbUpdatableField	The field value can be changed.
dbDescending	The field is sorted in descending (Z – A or 100 – 0) order (applies only to a Field object in a Fields collection of an Index object). If you omit this constant, the field is sorted in ascending (A – Z or 0 – 100) order (default).

fieldinfo

A reference to a CDaoFieldInfo structure.

Remarks

A **DAOField** (OLE) object is created and appended to the Fields collection of the **DAOTableDef** (OLE) object. Besides its use for examining object properties, you can also use **CDaoFieldInfo** to construct an input parameter for creating new fields in a tabledef. The first version of **CreateField** is simpler to use, but if you want finer control, you can use the second version of **CreateField**, which takes a **CDaoFieldInfo** parameter.

If you use the version of **CreateField** that takes a **CDaoFieldInfo** parameter, you must carefully set each of the following members of the **CDaoFieldInfo** structure:

- **m_strName**
- **m_nType**
- **m_lSize**
- **m_lAttributes**
- **m_bAllowZeroLength**

The remaining members of **CDaoFieldInfo** should be set to **0**, **FALSE**, or an empty string, as appropriate for the member, or a **CDaoException** may occur.

For related information, see the topic "CreateField Method" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::DeleteField
CDaoTableDef::CreateIndex
CDaoTableDef::DeleteIndex

Other Resources

CDaoTableDef Members

CDaoTableDef::CreateIndex

Call this function to add an index to a table.

```
void CreateIndex(
   CDaoIndexInfo& indexinfo
);
```

Parameters

indexinfo

A reference to a CDaoIndexInfo structure.

Remarks

Indexes specify the order of records accessed from database tables and whether or not duplicate records are accepted. Indexes also provide efficient access to data.

You do not have to create indexes for tables, but in large, unindexed tables, accessing a specific record or creating a recordset can take a long time. On the other hand, creating too many indexes slows down update, append, and delete operations as all indexes are automatically updated. Consider these factors as you decide which indexes to create.

The following members of the **CDaoIndexInfo** structure must be set:

- **m_strName** A name must be supplied.
- **m_pFieldInfos** Must point to an array of **CDaoIndexFieldInfo** structures.
- **m_nFields** Must specify the number of fields in the array of **CDaoFieldInfo** structures.

The remaining members will be ignored if set to **FALSE**. In addition, the **m_lDistinctCount** member is ignored during creation of the index.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::DeleteIndex
CDaoTableDef::CreateField
CDaoTableDef::DeleteField
CDaoIndexInfo Structure

Other Resources

CDaoTableDef Members

CDaoTableDef::DeleteField

Call this member function to remove a field and make it inaccessible.

```
void DeleteField(
   LPCTSTR lpszName
);
void DeleteField(
   int nIndex
);
```

Parameters

lpszName

A pointer to a string expression that is the name of an existing field.

nIndex

The index of the field in the table's zero-based Fields collection, for lookup by index.

Remarks

You can use this member function on a new object that has not been appended to the database or when CanUpdate returns nonzero.

For related information, see the topic "Delete Method" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::CreateField
CDaoTableDef::CreateIndex
CDaoTableDef::DeleteIndex

Other Resources

CDaoTableDef Members

CDaoTableDef::DeleteIndex

Call this member function to delete an index in an underlying table.

```
void DeleteIndex(
   LPCTSTR lpszName
);
void DeleteIndex(
   int nIndex
);
```

Parameters

lpszName

A pointer to a string expression that is the name of an existing index.

nIndex

The array index of the index object in the database's zero-based TableDefs collection, for lookup by index.

Remarks

You can use this member function on a new object that hasn't been appended to the database or when CanUpdate returns nonzero.

For related information, see the topic "Delete Method" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::CreateIndex
CDaoTableDef::CreateField
CDaoTableDef::DeleteField

Other Resources

CDaoTableDef Members

CDaoTableDef::GetAttributes

For a **CDaoTableDef** object, the return value specifies characteristics of the table represented by the **CDaoTableDef** object and can be a sum of these constants:

```
long GetAttributes( );
```

Return Value

Returns a value that indicates one or more characteristics of a **CDaoTableDef** object.

Remarks

Constant	Description
dbAttachExclusive	For databases that use the Microsoft Jet database engine, indicates the table is an attached table opened for exclusive use.
dbAttachSavePWD	For databases that use the Microsoft Jet database engine, indicates that the user ID and password for the attached table are saved with the connection information.
dbSystemObject	Indicates the table is a system table provided by the Microsoft Jet database engine.
dbHiddenObject	Indicates the table is a hidden table provided by the Microsoft Jet database engine.
dbAttachedTable	Indicates the table is an attached table from a non-ODBC database, such as a Paradox database.
dbAttachedODBC	Indicates the table is an attached table from an ODBC database, such as Microsoft SQL Server.

A system table is a table created by the Microsoft Jet database engine to contain various internal information.

A hidden table is a table created for temporary use by the Microsoft Jet database engine.

For related information, see the topic "Attributes Property" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::SetAttributes

Other Resources

CDaoTableDef Members

CDaoTableDef::GetConnect

Call this member function to obtain the connection string for a data source.

```
CString GetConnect( );
```

Return Value

A **CString** object containing the path and database type for the table.

Remarks

For a **CDaoTableDef** object that represents an attached table, the **CString** object consists of one or two parts (a database type specifier and a path to the database).

The path as shown in the table below is the full path for the directory containing the database files and must be preceded by the identifier "DATABASE=". In some cases (as with Microsoft Jet and Microsoft Excel databases), a specific filename is included in the database path argument.

The table in CDaoTableDef::SetConnect shows possible database types and their corresponding database specifiers and paths:

For Microsoft Jet database base tables, the specifier is a empty string ("").

If a password is required but not provided, the ODBC driver displays a login dialog box the first time a table is accessed and again if the connection is closed and reopened. If an attached table has the **dbAttachSavePWD** attribute, the login prompt will not appear when the table is reopened.

For related information, see the topic "Connect Property" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::SetConnect

Other Resources

CDaoTableDef Members

CDaoTableDef::GetDateCreated

Call this function to determine the date and time the table underlying the **CDaoTableDef** object was created.

```
COleDateTime GetDateCreated( );
```

Return Value

A value containing the date and time of the creation of the table underlying the **CDaoTableDef** object.

Remarks

The date and time settings are derived from the computer on which the base table was created or last updated. In a multiuser environment, users should get these settings directly from the file server to avoid discrepancies; that is, all clients should use a "standard" time source — perhaps from one server.

For related information, see the topic "DateCreated, LastUpdated Properties" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::GetDateLastUpdated

Other Resources

CDaoTableDef Members

CDaoTableDef::GetDateLastUpdated

Call this function to determine the date and time the table underlying the **CDaoTableDef** object was last updated.

```
COleDateTime GetDateLastUpdated( );
```

Return Value

A value that contains the date and time the table underlying the **CDaoTableDef** object was last updated.

Remarks

The date and time settings are derived from the computer on which the base table was created or last updated. In a multiuser environment, users should get these settings directly from the file server to avoid discrepancies; that is, all clients should use a "standard" time source — perhaps from one server.

For related information, see the topic "DateCreated, LastUpdated Properties" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::GetDateCreated

Other Resources

CDaoTableDef Members

CDaoTableDef::GetFieldCount

Call this member function to retrieve the number of fields defined in the table.

```
short GetFieldCount( );
```

Return Value

The number of fields in the table.

Remarks

If its value is 0, there are no objects in the collection.

For related information, see the topic "Count Property" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::GetFieldInfo
CDaoTableDef::GetIndexInfo
CDaoTableDef::GetIndexCount

Other Resources

CDaoTableDef Members

CDaoTableDef::GetFieldInfo

Call this member function to obtain various kinds of information about a field defined in the tabledef.

```
void GetFieldInfo(
   int nIndex,
   CDaoFieldInfo& fieldinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
void GetFieldInfo(
   LPCTSTR lpszName,
   CDaoFieldInfo& fieldinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
```

Parameters

nIndex

The index of the field object in the table's zero-based Fields collection, for lookup by index.

fieldinfo

A reference to a CDaoFieldInfo structure.

dwInfoOptions

Options that specify which information about the field to retrieve. The available options are listed here along with what they cause the function to return:

- **AFX_DAO_PRIMARY_INFO** (Default) Name, Type, Size, Attributes. Use this option for fastest performance.
- **AFX_DAO_SECONDARY_INFO** Primary information, plus: Ordinal Position, Required, Allow Zero Length, Collating Order, Foreign Name, Source Field, Source Table
- **AFX_DAO_ALL_INFO** Primary and secondary information, plus: Validation Rule, Validation Text, Default Value

lpszName

A pointer to the name of the field object, for lookup by name. The name is a string with up to 64 characters that uniquely names the field.

Remarks

One version of the function lets you look up a field by index. The other version lets you look up a field by name.

For a description of the information returned, see the CDaoFieldInfo structure. This structure has members that correspond to the items of information listed above in the description of *dwInfoOptions*. When you request information at one level, you get information for any prior levels as well.

For related information, see the topic "Attributes Property" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::GetIndexInfo
CDaoTableDef::GetIndexCount
CDaoTableDef::GetFieldCount

Other Resources

CDaoTableDef Members

CDaoTableDef::GetIndexCount

Call this member function to obtain the number of indexes for a table.

```
short GetIndexCount( );
```

Return Value

The number of indexes for the table.

Remarks

If its value is 0, there are no indexes in the collection.

For related information, see the topic "Count Property" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::GetIndexInfo
CDaoTableDef::GetFieldInfo
CDaoTableDef::GetFieldCount

Other Resources

CDaoTableDef Members

CDaoTableDef::GetIndexInfo

Call this member function to obtain various kinds of information about an index defined in the tabledef.

```
void GetIndexInfo(
   int nIndex,
   CDaoIndexInfo& indexinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
void GetIndexInfo(
   LPCTSTR lpszName,
   CDaoIndexInfo& indexinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
```

Parameters

nIndex

The numeric index of the Index object in the table's zero-based Indexes collection, for lookup by its position in the collection.

indexinfo

A reference to a CDaoIndexInfo structure.

dwInfoOptions

Options that specify which information about the index to retrieve. The available options are listed here along with what they cause the function to return:

- **AFX_DAO_PRIMARY_INFO** Name, Field Info, Fields. Use this option for fastest performance.
- **AFX_DAO_SECONDARY_INFO** Primary information, plus: Primary, Unique, Clustered, Ignore Nulls, Required, Foreign
- **AFX_DAO_ALL_INFO** Primary and secondary information, plus: Distinct Count

lpszName

A pointer to the name of the index object, for lookup by name.

Remarks

One version of the function lets you look up an index by its position in the collection. The other version lets you look up an index by name.

For a description of the information returned, see the CDaoIndexInfo structure. This structure has members that correspond to the items of information listed above in the description of *dwInfoOptions*. When you request information at one level, you get information for any prior levels as well.

For related information, see the topic "Attributes Property" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::GetFieldInfo
CDaoTableDef::GetIndexCount
CDaoTableDef::GetFieldCount

Other Resources

CDaoTableDef Members

CDaoTableDef::GetName

Call this member function to obtain the user-defined name of the underlying table.

```
CString GetName( );
```

Return Value

A user-defined name for a table.

Remarks

This name starts with a letter and can contain a maximum of 64 characters. It can include numbers and underscore characters but cannot include punctuation or spaces.

For related information, see the topic "Name Property" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::SetName
CDaoTableDef::GetConnect
CDaoTableDef::SetConnect

Other Resources

CDaoTableDef Members

CDaoTableDef::GetRecordCount

Call this member function to find out how many records are in a **CDaoTableDef** object.

```
long GetRecordCount( );
```

Return Value

The number of records accessed in a tabledef object.

Remarks

Calling **GetRecordCount** for a table-type **CDaoTableDef** object reflects the approximate number of records in the table and is affected immediately as table records are added and deleted. Rolled back transactions will appear as part of the record count until you call CDaoWorkSpace::CompactDatabase. A **CDaoTableDef** object with no records has a record count property setting of 0. When working with attached tables or ODBC databases, **GetRecordCount** always returns –1.

For related information, see the topic "RecordCount Property" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::GetSourceTableName
CDaoTableDef::SetSourceTableName

Other Resources

CDaoTableDef Members

CDaoTableDef::GetSourceTableName

Call this member function to retrieve the name of an attached table in a source database.

```
CString GetSourceTableName( );
```

Return Value

A **CString** object that specifies the source name of an attached table, or an empty string if a native data table.

Remarks

An attached table is a table in another database linked to a Microsoft Jet database. Data for attached tables remains in the external database, where it can be manipulated by other applications.

For related information, see the topic "SourceTableName Property" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::GetRecordCount
CDaoTableDef::SetSourceTableName

Other Resources

CDaoTableDef Members

CDaoTableDef::GetValidationRule

Call this member function to retrieve the validation rule for a tabledef.

```
CString GetValidationRule( );
```

Return Value

A **CString** object that validates the data in a field as it is changed or added to a table.

Remarks

Validation rules are used in connection with update operations. If a tabledef contains a validation rule, updates to that tabledef must match predetermined criteria before the data is changed. If the change does not match the criteria, an exception containing the value of GetValidationText is thrown. For a **CDaoTableDef** object, this **CString** is read-only for an attached table and read/write for a base table.

For related information, see the topic "ValidationRule Property" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::SetValidationRule
CDaoTableDef::GetValidationText
CDaoTableDef::SetValidationText

Other Resources

CDaoTableDef Members

CDaoTableDef::GetValidationText

Call this function to retrieve the string to display when a user enters data that does not match the validation rule.

```
CString GetValidationText( );
```

Return Value

A **CString** object that specifies the text displayed if the user enters data that does not match the validation rule.

Remarks

For a **CDaoTableDef** object, this **CString** is read-only for an attached table and read/write for a base table.

For related information, see the topic "ValidationText Property" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::SetValidationRule
CDaoTableDef::SetValidationText
CDaoTableDef::GetValidationRule

Other Resources

CDaoTableDef Members

CDaoTableDef::IsOpen

Call this member function to determine whether the **CDaoTableDef** object is currently open.

```
BOOL IsOpen( ) const;
```

Return Value

Nonzero if the **CDaoTableDef** object is open; otherwise 0.

Remarks

See Also

Reference

CDaoTableDef Class

Hierarchy Chart

CDaoTableDef::Open

Other Resources

CDaoTableDef Members

CDaoTableDef::Open

Call this member function to open a tabledef previously saved in the database's TableDef's collection.

```
virtual void Open(
   LPCTSTR lpszName
);
```

Parameters

lpszName

A pointer to a string that specifies a table name.

Remarks

See Also

Reference

CDaoTableDef Class

Hierarchy Chart

CDaoTableDef::IsOpen

CDaoTableDef::Create

CDaoTableDef::Close

Other Resources

CDaoTableDef Members

CDaoTableDef::RefreshLink

Call this member function to update the connection information for an attached table.

```
void RefreshLink( );
```

Remarks

You change the connection information for an attached table by calling SetConnect on the corresponding **CDaoTableDef** object and then using the **RefreshLink** member function to update the information. When you call **RefreshLink**, the attached table's properties are not changed.

To force the modified connect information to take effect, all open CDaoRecordset objects based on this tabledef must be closed.

For related information, see the topic "RefreshLink Method" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::SetConnect

Other Resources

CDaoTableDef Members

CDaoTableDef::SetAttributes

Sets a value that indicates one or more characteristics of a **CDaoTableDef** object.

```
void SetAttributes(
   long lAttributes
);
```

Parameters

lAttributes

Characteristics of the table represented by the **CDaoTableDef** object and can be a sum of these constants:

Constant	Description
dbAttachExclusive	For databases that use the Microsoft Jet database engine, indicates the table is an attached table opened for exclusive use.
dbAttachSavePWD	For databases that use the Microsoft Jet database engine, indicates that the user ID and password for the attached table are saved with the connection information.
dbSystemObject	Indicates the table is a system table provided by the Microsoft Jet database engine.
dbHiddenObject	Indicates the table is a hidden table provided by the Microsoft Jet database engine.

Remarks

When setting multiple attributes, you can combine them by summing the appropriate constants using the bitwise-OR operator. Setting **dbAttachExclusive** on a nonattached table produces an exception. Combining the following values also produce an exception:

- **dbAttachExclusive | dbAttachedODBC**
- **dbAttachSavePWD | dbAttachedTable**

For related information, see the topic "Attributes Property" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::SetConnect

Other Resources

CDaoTableDef Members

CDaoTableDef::SetConnect

For a **CDaoTableDef** object that represents an attached table, the string object consists of one or two parts (a database type specifier and a path to the database).

```
void SetConnect(
   LPCTSTR lpszConnect
);
```

Parameters

lpszConnect

A pointer to a string expression that specifies additional parameters to pass to ODBC or installable ISAM drivers.

Remarks

The path as shown in the table below is the full path for the directory containing the database files and must be preceded by the identifier "DATABASE=". In some cases (as with Microsoft Jet and Microsoft Excel databases), a specific filename is included in the database path argument.

Note
Do not include whitespace around the equal sign in path statements of the form "DATABASE=drive:\\path". This will result in an exception being thrown and the connection failing.

The following table shows possible database types and their corresponding database specifiers and paths:

Database type	**Specifier**	**Path**
Database using the Jet database engine	"[*database*];"	"*drive*:*path**filename*.MDB"
dBASE III	"dBASE III;"	"*drive*:*path*"
dBASE IV	"dBASE IV;"	"*drive*:*path*"
dBASE 5	"dBASE 5.0;"	"*drive*:*path*"
Paradox 3.x	"Paradox 3.x;"	"*drive*:*path*"
Paradox 4.x	"Paradox 4.x;"	"*drive*:*path*"
Paradox 5.x	"Paradox 5.x;"	"*drive*:*path*"
Excel 3.0	"Excel 3.0;"	"*drive*:*path**filename*.XLS"

Excel 4.0	"Excel 4.0;"	"*drive*:*path**filename*.XLS"
Excel 5.0 or Excel 95	"Excel 5.0;"	"*drive*:*path**filename*.XLS"
Excel 97	"Excel 8.0;"	"*drive*:*path**filename*.XLS"
HTML Import	"HTML Import;"	"*drive*:*path**filename*"
HTML Export	"HTML Export;"	"*drive*:*path*"
Text	"Text;"	"drive:\\path"
ODBC	"ODBC; DATABASE=*database*; UID=*user*;PWD=*password*; DSN=*datasourcename;* LOGINTIMEOUT=*seconds;*" (This may not be a complete connection string for all servers; it is just an example. It is very important not to have spaces between the parameters.)	None
Exchange	"Exchange; MAPILEVEL=*folderpath*; [TABLETYPE={ 0 | 1 };] [PROFILE=*profile*;] [PWD=*password*;] [DATABASE=*database*;]"	"*drive*:*path**filename*.MDB"

Note

Btrieve is no longer supported as of DAO 3.5.

You must use a double backslash (\\) in the connection strings. If you have modified the properties of an existing connection using **SetConnect**, you must subsequently call RefreshLink. If you are initializing the connection properties using **SetConnect**, you need not call **RefreshLink**, but should you choose to do so, first append the tabledef.

If a password is required but not provided, the ODBC driver displays a login dialog box the first time a table is accessed and again if the connection is closed and reopened.

You can set the connection string for a **CDaoTableDef** object by providing a source argument to the **Create** member function. You can check the setting to determine the type, path, user ID, password, or ODBC data source of the database. For more information, see the documentation for the specific driver.

For related information, see the topic "Connect Property" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::RefreshLink
CDaoTableDef::SetAttributes

Other Resources

CDaoTableDef::SetName

Call this member function to set a user-defined name for a table.

```
void SetName(
   LPCTSTR lpszName
);
```

Parameters

lpszName

A pointer to a string expression that specifies a name for a table.

Remarks

The name must start with a letter and can contain a maximum of 64 characters. It can include numbers and underscore characters but cannot include punctuation or spaces.

For related information, see the topic "Name Property" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::RefreshLink
CDaoTableDef::SetConnect

Other Resources

CDaoTableDef Members

CDaoTableDef::SetSourceTableName

Call this member function to specify the name of an attached table or the name of the base table on which the **CDaoTableDef** object is based, as it exists in the original source of the data.

```
void SetSourceTableName(
   LPCTSTR lpszSrcTableName
);
```

Parameters

lpszSrcTableName

A pointer to a string expression that specifies a table name in the external database. For a base table, the setting is an empty string ("").

Remarks

You must then call RefreshLink. This property setting is empty for a base table and read/write for an attached table or an object not appended to a collection.

For related information, see the topic "SourceTableName Property" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::RefreshLink
CDaoTableDef::GetSourceTableName

Other Resources

CDaoTableDef Members

CDaoTableDef::SetValidationRule

Call this member function to set a validation rule for a tabledef.

```
void SetValidationRule(
   LPCTSTR lpszValidationRule
);
```

Parameters

lpszValidationRule

A pointer to a string expression that validates an operation.

Remarks

Validation rules are used in connection with update operations. If a tabledef contains a validation rule, updates to that tabledef must match predetermined criteria before the data is changed. If the change does not match the criteria, an exception containing the text of GetValidationText is displayed.

Validation is supported only for databases that use the Microsoft Jet database engine. The expression cannot refer to user-defined functions, domain aggregate functions, SQL aggregate functions, or queries. A validation rule for a **CDaoTableDef** object can refer to multiple fields in that object.

For example, for fields named `hire_date` and `termination_date`, a validation rule might be:

```
CString strRule = _T("termination_date>hire_date");
MyRs.SetValidationRule(strRule);
```

For related information, see the topic "ValidationRule Property" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::GetValidationText
CDaoTableDef::SetValidationText
CDaoTableDef::GetValidationRule

Other Resources

CDaoTableDef Members

CDaoTableDef::SetValidationText

Call this member function to set the exception text of a validation rule for a **CDaoTableDef** object with an underlying base table supported by the Microsoft Jet database engine.

```
void SetValidationText(
   LPCTSTR lpszValidationText
);
```

Parameters

lpszValidationText

A pointer to a string expression that specifies the text displayed if entered data is invalid.

Remarks

You cannot set the validation text of an attached table.

For related information, see the topic "ValidationText Property" in DAO Help.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::SetValidationRule
CDaoTableDef::GetValidationText
CDaoTableDef::GetValidationRule

Other Resources

CDaoTableDef Members

CDaoTableDef Data Members

For information about the data members in **CDaoTableDef**, see CDaoTableDef Members.

CDaoTableDef::m_pDatabase

Contains a pointer to the CDaoDatabase object for this table.

Remarks

See Also

Reference

CDaoTableDef Class

Hierarchy Chart

CDaoTableDef::m_pDAOTableDef

Other Resources

CDaoTableDef Members

CDaoTableDef::m_pDAOTableDef

Contains a pointer to the OLE interface for the DAO tabledef object underlying the **CDaoTableDef** object.

Remarks

Use this pointer if you need to access the DAO interface directly.

See Also

Reference

CDaoTableDef Class
Hierarchy Chart
CDaoTableDef::m_pDatabase

Other Resources

CDaoTableDef Members

CDaoWorkspace Class

Manages a named, password-protected database session from login to logoff, by a single user.

```
class CDaoWorkspace : public CObject
```

Remarks

In most cases, you will not need multiple workspaces, and you will not need to create explicit workspace objects; when you open database and recordset objects, they use DAO's default workspace. However, if needed, you can run multiple sessions at a time by creating additional workspace objects. Each workspace object can contain multiple open database objects in its own Databases collection. In MFC, a workspace is primarily a transaction manager, specifying a set of open databases all in the same "transaction space."

Note

The DAO database classes are distinct from the MFC database classes based on Open Database Connectivity (ODBC). All DAO database class names have a "CDao" prefix. In general, the MFC classes based on DAO are more capable than the MFC classes based on ODBC. The DAO-based classes access data through the Microsoft Jet database engine, including ODBC drivers. They also support Data Definition Language (DDL) operations, such as creating databases and adding tables and fields via the classes, without having to call DAO directly.

Capabilities

Class **CDaoWorkspace** provides the following:

- Explicit access, if needed, to a default workspace, created by initializing the database engine. Usually you use DAO's default workspace implicitly by creating database and recordset objects.
- A transaction space in which transactions apply to all databases open in the workspace. You can create additional workspaces to manage separate transaction spaces.
- An interface to many properties of the underlying Microsoft Jet database engine (see the static member functions). Opening or creating a workspace, or calling a static member function before open or create, initializes the database engine.
- Access to the database engine's Workspaces collection, which stores all active workspaces that have been appended to it. You can also create and work with workspaces without appending them to the collection.

Security

MFC does not implement the Users and Groups collections in DAO, which are used for security control. If you need those aspects of DAO, you must program them yourself via direct calls to DAO interfaces. For information, see Technical Note 54.

Usage

You can use class **CDaoWorkspace** to:

- Explicitly open the default workspace.

  Usually your use of the default workspace is implicit — when you open new CDaoDatabase or CDaoRecordset objects. But you might need to access it explicitly — for example, to access database engine properties or the Workspaces collection. See "Implicit Use of the Default Workspace" below.
- Create new workspaces. Call Append if you want to add them to the Workspaces collection.
- Open an existing workspace in the Workspaces collection.

Creating a new workspace that does not already exist in the Workspaces collection is described under the Create member function. Workspace objects do not persist in any way between datababase engine sessions. If your application links MFC statically, ending the application uninitializes the database engine. If your application links with MFC dynamically, the database engine is uninitialized when the MFC DLL is unloaded.

Explicitly opening the default workspace, or opening an existing workspace in the Workspaces collection, is described under

the Open member function.

End a workspace session by closing the workspace with the Close member function. **Close** closes any databases you have not closed previously, rolling back any uncommitted transactions.

Transactions

DAO manages transactions at the workspace level; hence, transactions on a workspace with multiple open databases apply to all of the databases. For example, if two databases have uncommitted updates and you call CommitTrans, all of the updates are committed. If you want to limit transactions to a single database, you need a separate workspace object for it.

Implicit Use of the Default Workspace

MFC uses DAO's default workspace implicitly under the following circumstances:

- If you create a new **CDaoDatabase** object but do not do so through an existing **CDaoWorkspace** object, MFC creates a temporary workspace object for you, which corresponds to DAO's default workspace. If you do so for multiple databases, all of the database objects are associated with the default workspace. You can access a database's workspace through a **CDaoDatabase** data member.
- Similarly, if you create a **CDaoRecordset** object without supplying a pointer to a **CDaoDatabase** object, MFC creates a temporary database object and, by extension, a temporary workspace object. You can access a recordset's database, and indirectly its workspace, through a **CDaoRecordset** data member.

Other Operations

Other database operations are also provided, such as repairing a corrupted database or compacting a database.

For information about calling DAO directly and about DAO security, see Technical Note 54.

Requirements

Header: afxdao.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class
Hierarchy Chart
CDaoDatabase Class
CDaoRecordset Class
CDaoTableDef Class
CDaoQueryDef Class
CDaoException Class

Other Resources

CDaoWorkspace Members

CDaoWorkspace Members

Base Class Members

CObject Members

Data Members

m_pDAOWorkspace	Points to the underlying DAO workspace object.

Construction

CDaoWorkspace	Constructs a workspace object. Afterwards, call **Create** or **Open**.

Attributes

GetIsolateODBCTrans	Returns a value that indicates whether multiple transactions that involve the same ODBC data source are isolated via forced multiple connections to the data source.
GetName	Returns the user-defined name for the workspace object.
GetUserName	Returns the user name specified when the workspace was created. This is the name of the workspace owner.
IsOpen	Returns nonzero if the workspace is open.
SetIsolateODBCTrans	Specifies whether multiple transactions that involve the same ODBC data source are isolated by forcing multiple connections to the data source.

Operations

Append	Appends a newly created workspace to the database engine's Workspaces collection.
BeginTrans	Begins a new transaction, which applies to all databases open in the workspace.
Close	Closes the workspace and all of the objects it contains. Pending transactions are rolled back.
CommitTrans	Completes the current transaction and saves the changes.
CompactDatabase	Compacts (or duplicates) a database.
Create	Creates a new DAO workspace object.
GetDatabaseCount	Returns the number of DAO database objects in the workspace's Databases collection.
GetDatabaseInfo	Returns information about a specified DAO database defined in the workspace's Databases collection.
GetWorkspaceCount	Returns the number of DAO workspace objects in the database engine's Workspaces collection.
GetWorkspaceInfo	Returns information about a specified DAO workspace defined in the database engine's Workspaces collection.
Idle	Allows the database engine to perform background tasks.
Open	Explicitly opens a workspace object associated with DAO's default workspace.
RepairDatabase	Attempts to repair a damaged database.

Rollback	Ends the current transaction and does not save the changes.

Database Engine Properties

GetIniPath	Returns the location of the Microsoft Jet database engine's initialization settings in the Windows registry.
GetLoginTimeout	Returns the number of seconds before an error occurs when the user attempts to log in to an ODBC data base.
GetVersion	Returns a string that contains the version of the database engine associated with the workspace.
SetDefaultPassword	Sets the password that the database engine uses when a workspace object is created without a specific pa ssword.
SetDefaultUser	Sets the user name that the database engine uses when a workspace object is created without a specific u ser name.
SetIniPath	Sets the location of the Microsoft Jet database engine's initialization settings in the Windows registry.
SetLoginTimeout	Sets the number of seconds before an error occurs when the user attempts to log in to an ODBC data sou rce.

See Also

Reference

CDaoWorkspace Class
Hierarchy Chart

CDaoWorkspace Member Functions

For information about the member functions in **CDaoWorkspace**, see CDaoWorkspace Members.

CDaoWorkspace::Append

Call this member function after you call Create.

```
virtual void Append( );
```

Remarks

Append appends a newly created workspace object to the database engine's Workspaces collection. Workspaces do not persist between database engine sessions; they are stored only in memory, not on disk. You do not have to append a workspace; if you do not, you can still use it.

An appended workspace remains in the Workspaces collection, in an active, open state, until you call its Close member function.

For related information, see the topic "Append Method" in DAO Help.

See Also

Reference

CDaoWorkspace Class

Hierarchy Chart

Other Resources

CDaoWorkspace Members

CDaoWorkspace::BeginTrans

Call this member function to initiate a transaction.

```
void BeginTrans( );
```

Remarks

After you call **BeginTrans**, updates you make to your data or database structure take effect when you commit the transaction. Because the workspace defines a single transaction space, the transaction applies to all open databases in the workspace. There are two ways to complete the transaction:

- Call the CommitTrans member function to commit the transaction and save changes to the data source.
- Or call the Rollback member function to cancel the transaction.

Closing the workspace object or a database object while a transaction is pending rolls back all pending transactions.

If you need to isolate transactions on one ODBC data source from those on another ODBC data source, see the SetIsolateODBCTrans member function.

See Also

Reference

CDaoWorkspace Class
Hierarchy Chart
CDaoWorkspace::GetIsolateODBCTrans
CDaoWorkspace::CommitTrans
CDaoWorkspace::Rollback

Other Resources

CDaoWorkspace Members

CDaoWorkspace::CDaoWorkspace

Constructs a **CDaoWorkspace** object.

```
CDaoWorkspace( );
```

Remarks

After constructing the C++ object, you have two options:

- Call the object's Open member function to open the default workspace or to open an existing object in the Workspaces collection.
- Or call the object's Create member function to create a new DAO workspace object. This explicitly starts a new workspace session, which you can refer to via the **CDaoWorkspace** object. After calling **Create**, you can call Append if you want to add the workspace to the database engine's Workspaces collection.

See the class overview for CDaoWorkspace for information about when you need to explicitly create a **CDaoWorkspace** object. Usually, you use workspaces created implicitly when you open a CDaoDatabase object without specifying a workspace or when you open a CDaoRecordset object without specifying a database object. MFC DAO objects created in this way use DAO's default workspace, which is created once and reused.

To release a workspace and its contained objects, call the workspace object's Close member function.

See Also

Reference

CDaoWorkspace Class

Hierarchy Chart

Other Resources

CDaoWorkspace Members

CDaoWorkspace::Close

Call this member function to close the workspace object.

```
virtual void Close( );
```

Remarks

Closing an open workspace object releases the underlying DAO object and, if the workspace is a member of the Workspaces collection, removes it from the collection. Calling **Close** is good programming practice.

Caution
Closing a workspace object closes any open databases in the workspace. This results in any recordsets open in the databases being closed as well, and any pending edits or updates are rolled back. For related information, see the CDaoDatabase::Close, CDaoRecordset::Close, CDaoTableDef::Close, and CDaoQueryDef::Close member functions.

Workspace objects are not permanent; they only exist while references to them exist. This means that when the database engine session ends, the workspace and its Databases collection do not persist. You must re-create them for the next session by opening your workspace and database(s) again.

For related information, see the topic "Close Method" in DAO Help.

See Also

Reference

CDaoWorkspace Class

Hierarchy Chart

CDaoWorkspace::Open

Other Resources

CDaoWorkspace Members

CDaoWorkspace::CommitTrans

Call this member function to commit a transaction — save a group of edits and updates to one or more databases in the workspace.

```
void CommitTrans( );
```

Remarks

A transaction consists of a series of changes to the database's data or its structure, beginning with a call to BeginTrans. When you complete the transaction, either commit it or roll it back (cancel the changes) with Rollback. By default, without transactions, updates to records are committed immediately. Calling **BeginTrans** causes commitment of updates to be delayed until you call **CommitTrans**.

Caution
Within one workspace, transactions are always global to the workspace and are not limited to only one database or recordset. If you perform operations on more than one database or recordset within a workspace transaction, **CommitTrans** commits all pending updates, and **Rollback** restores all operations on those databases and recordsets.

When you close a database or workspace with pending transactions, the transactions are all rolled back.

Note
This is not a two-phase commit mechanism. If one update fails to commit, others still will commit.

See Also

Reference

CDaoWorkspace Class

Hierarchy Chart

Other Resources

CDaoWorkspace Members

CDaoWorkspace::CompactDatabase

Call this member function to compact a specified Microsoft Jet (.MDB) database.

```
static void PASCAL CompactDatabase(
   LPCTSTR lpszSrcName,
   LPCTSTR lpszDestName,
   LPCTSTR lpszLocale = dbLangGeneral,
   int nOptions = 0
);
static void PASCAL CompactDatabase(
   LPCTSTR lpszSrcName,
   LPCTSTR lpszDestName,
   LPCTSTR lpszLocale,
   int nOptions,
   LPCTSTR lpszPassword
);
```

Parameters

lpszSrcName

The name of an existing, closed database. It can be a full path and filename, such as "C:\\MYDB.MDB". If the filename has an extension, you must specify it. If your network supports the uniform naming convention (UNC), you can also specify a network path, such as "\\\\MYSERVER\\MYSHARE\\MYDIR\\MYDB.MDB". (Double backslashes are required in the path strings because "\" is the C++ escape character.)

lpszDestName

The full path of the compacted database that you are creating. You can also specify a network path as with *lpszSrcName*. You cannot use the *lpszDestName* argument to specify the same database file as *lpszSrcName*.

lpszPassword

A password, used when you want to compact a password-protected database. Note that if you use the version of **CompactDatabase** that takes a password, you must supply all parameters. Also, because this is a connect parameter, it requires special formatting, as follows: ;PWD=*lpszPassword*. For example: ;PWD="Happy". (The leading semicolon is required.)

lpszLocale

A string expression used to specify collating order for creating *lpszDestName*. If you omit this argument by accepting the default value of **dbLangGeneral** (see below), the locale of the new database is the same as that of the old database. Possible values are:

- **dbLangGeneral** English, German, French, Portuguese, Italian, and Modern Spanish
- **dbLangArabic** Arabic
- **dbLangCyrillic** Russian
- **dbLangCzech** Czech
- **dbLangDutch** Dutch
- **dbLangGreek** Greek
- **dbLangHebrew** Hebrew
- **dbLangHungarian** Hungarian
- **dbLangIcelandic** Icelandic
- **dbLangNordic** Nordic languages (Microsoft Jet database engine version 1.0 only)
- **dbLangNorwdan** Norwegian and Danish
- **dbLangPolish** Polish

- **dbLangSpanish** Traditional Spanish
- **dbLangSwedfin** Swedish and Finnish
- **dbLangTurkish** Turkish

nOptions

Indicates one or more options for the target database, *lpszDestName*. If you omit this argument by accepting the default value, the *lpszDestName* will have the same encryption and the same version as *lpszSrcName*. You can combine the **dbEncrypt** or **dbDecrypt** option with one of the version options using the bitwise-OR operator. Possible values, which specify a database format, not a database engine version, are:

- **dbEncrypt** Encrypt the database while compacting.
- **dbDecrypt** Decrypt the database while compacting.
- **dbVersion10** Create a database that uses the Microsoft Jet database engine version 1.0 while compacting.
- **dbVersion11** Create a database that uses the Microsoft Jet database engine version 1.1 while compacting.
- **dbVersion20** Create a database that uses the Microsoft Jet database engine version 2.0 while compacting.
- **dbVersion30** Create a database that uses the Microsoft Jet database engine version 3.0 while compacting.

You can use **dbEncrypt** or **dbDecrypt** in the options argument to specify whether to encrypt or to decrypt the database as it is compacted. If you omit an encryption constant or if you include both **dbDecrypt** and **dbEncrypt**, *lpszDestName* will have the same encryption as *lpszSrcName*. You can use one of the version constants in the options argument to specify the version of the data format for the compacted database. This constant affects only the version of the data format of *lpszDestName*. You can specify only one version constant. If you omit a version constant, *lpszDestName* will have the same version as *lpszSrcName*. You can compact *lpszDestName* only to a version that is the same or later than that of *lpszSrcName*.

Caution
If a database is not encrypted, it is possible, even if you implement user/password security, to directly read the binary disk file that constitutes the database.

Remarks

As you change data in a database, the database file can become fragmented and use more disk space than necessary. Periodically, you should compact your database to defragment the database file. The compacted database is usually smaller. You can also choose to change the collating order, the encryption, or the version of the data format while you copy and compact the database.

Caution
The **CompactDatabase** member function will not correctly convert a complete Microsoft Access database from one version to another. Only the data format is converted. Microsoft Access-defined objects, such as forms and reports, are not converted. However, the data is correctly converted.

Tip
You can also use **CompactDatabase** to copy a database file.

For more information about compacting databases, see the topic "CompactDatabase Method" in DAO Help.

See Also

Reference

CDaoWorkspace Class
Hierarchy Chart
CDaoWorkspace::RepairDatabase

Other Resources

CDaoWorkspace Members

CDaoWorkspace::Create

Call this member function to create a new DAO workspace object and associate it with the MFC **CDaoWorkspace** object.

```
virtual void Create(
   LPCTSTR lpszName,
   LPCTSTR lpszUserName,
   LPCTSTR lpszPassword
);
```

Parameters

lpszName

A string with up to 14 characters that uniquely names the new workspace object. You must supply a name. For related information, see the topic "Name Property" in DAO Help.

lpszUserName

The user name of the workspace's owner. For requirements, see the *lpszDefaultUser* parameter to the SetDefaultUser member function. For related information, see the topic "UserName Property" in DAO Help.

lpszPassword

The password for the new workspace object. A password can be up to 14 characters long and can contain any character except ASCII 0 (null). Passwords are case-sensitive. For related information, see the topic "Password Property" in DAO Help.

Remarks

The overall creation process is:

1. Construct a CDaoWorkspace object.
2. Call the object's **Create** member function to create the underlying DAO workspace. You must specify a workspace name.
3. Optionally call Append if you want to add the workspace to the database engine's Workspaces collection. You can work with the workspace without appending it.

After the **Create** call, the workspace object is in an open state, ready for use. You do not call **Open** after **Create**. You do not call **Create** if the workspace already exists in the Workspaces collection. **Create** initializes the database engine if it has not already been initialized for your application.

See Also

Reference

CDaoWorkspace Class
Hierarchy Chart
CDaoWorkspace::CDaoWorkspace
CDaoWorkspace::Close
CDaoWorkspace::Open

Other Resources

CDaoWorkspace Members

CDaoWorkspace::GetDatabaseCount

Call this member function to retrieve the number of DAO database objects in the workspace's Databases collection — the number of open databases in the workspace.

```
short GetDatabaseCount( );
```

Return Value

The number of open databases in the workspace.

Remarks

GetDatabaseCount is useful if you need to loop through all defined databases in the workspace's Databases collection. To obtain information about a given database in the collection, see GetDatabaseInfo. Typical usage is to call **GetDatabaseCount** for the number of open databases, then use that number as a loop index for repeated calls to **GetDatabaseInfo**.

See Also

Reference

CDaoWorkspace Class

Hierarchy Chart

Other Resources

CDaoWorkspace Members

CDaoWorkspace::GetDatabaseInfo

Call this member function to obtain various kinds of information about a database open in the workspace.

```
void GetDatabaseInfo(
   int nIndex,
   CDaoDatabaseInfo& dbinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
void GetDatabaseInfo(
   LPCTSTR lpszName,
   CDaoDatabaseInfo& dbinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
```

Parameters

nIndex

The zero-based index of the database object in the workspace's Databases collection, for lookup by index.

dbinfo

A reference to a CDaoDatabaseInfo object that returns the information requested.

dwInfoOptions

Options that specify which information about the database to retrieve. The available options are listed here along with what they cause the function to return:

- **AFX_DAO_PRIMARY_INFO** (Default) Name, Updatable, Transactions
- **AFX_DAO_SECONDARY_INFO** Primary information plus: Version, Collating Order, Query Timeout
- **AFX_DAO_ALL_INFO** Primary and secondary information plus: Connect

lpszName

The name of the database object, for lookup by name. The name is a string with up to 14 characters that uniquely names the new workspace object.

Remarks

One version of the function lets you look up a database by index. The other version lets you look up a database by name.

For a description of the information returned in *dbinfo*, see the CDaoDatabaseInfo structure. This structure has members that correspond to the items of information listed above in the description of *dwInfoOptions*. When you request information at one level, you get information for any prior levels as well.

See Also

Reference

CDaoWorkspace Class
Hierarchy Chart
CDaoWorkspace::GetDatabaseCount

Other Resources

CDaoWorkspace Members

CDaoWorkspace::GetIniPath

Call this member function to obtain the location of the Microsoft Jet database engine's initialization settings in the Windows registry.

```
static CString PASCAL GetIniPath( );
```

Return Value

A CString containing the registry location.

Remarks

You can use the location to obtain information about settings for the database engine. The information returned is actually the name of a registry subkey.

For related information, see the topics "IniPath Property" and "Customizing Windows Registry Settings for Data Access" in DAO Help.

See Also

Reference

CDaoWorkspace Class
Hierarchy Chart
CDaoWorkspace::SetIniPath
CDaoWorkspace::GetVersion

Other Resources

CDaoWorkspace Members

CDaoWorkspace::GetIsolateODBCTrans

Call this member function to get the current value of the DAO IsolateODBCTrans property for the workspace.

```
BOOL GetIsolateODBCTrans( );
```

Return Value

Nonzero if ODBC transactions are isolated; otherwise 0.

Remarks

In some situations, you might need to have multiple simultaneous transactions pending on the same ODBC database. To do this, you need to open a separate workspace for each transaction. Keep in mind that although each workspace can have its own ODBC connection to the database, this slows system performance. Because transaction isolation is not normally required, ODBC connections from multiple workspace objects opened by the same user are shared by default.

Some ODBC servers, such as Microsoft SQL Server, do not allow simultaneous transactions on a single connection. If you need to have more than one transaction at a time pending against such a database, set the IsolateODBCTrans property to **TRUE** on each workspace as soon as you open it. This forces a separate ODBC connection for each workspace.

For related information, see the topic "IsolateODBCTrans Property" in DAO Help.

See Also

Reference

CDaoWorkspace Class
Hierarchy Chart
CDaoWorkspace::SetIsolateODBCTrans

Other Resources

CDaoWorkspace Members

CDaoWorkspace::GetLoginTimeout

Call this member function to get the current value of the DAO LoginTimeout property for the workspace.

```
static short PASCAL GetLoginTimeout( );
```

Return Value

The number of seconds before an error occurs when you attempt to log in to an ODBC database.

Remarks

This value represents the number of seconds before an error occurs when you attempt to log in to an ODBC database. The default LoginTimeout setting is 20 seconds. When LoginTimeout is set to 0, no timeout occurs and the communication with the data source might hang.

When you are attempting to log in to an ODBC database, such as Microsoft SQL Server, the connection may fail as a result of network errors or because the server is not running. Rather than waiting for the default 20 seconds to connect, you can specify how long the database engine waits before it produces an error. Logging in to the server happens implicitly as part of a number of different events, such as running a query on an external server database.

For related information, see the topic "LoginTimeout Property" in DAO Help.

See Also

Reference

CDaoWorkspace Class
Hierarchy Chart
CDaoWorkspace::SetLoginTimeout

Other Resources

CDaoWorkspace Members

CDaoWorkspace::GetName

Call this member function to get the user-defined name of the DAO workspace object underlying the **CDaoWorkspace** object.

```
CString GetName( );
```

Return Value

A CString containing the user-defined name of the DAO workspace object.

Remarks

The name is useful for accessing the DAO workspace object in the database engine's Workspaces collection by name.

For related information, see the topic "Name Property" in DAO Help.

See Also

Reference

CDaoWorkspace Class

Hierarchy Chart

Other Resources

CDaoWorkspace Members

CDaoWorkspace::GetUserName

Call this member function to obtain the name of the owner of the workspace.

```
CString GetUserName( );
```

Return Value

A CString that represents the owner of the workspace object.

Remarks

To get or set the permissions for the workspace owner, call DAO directly to check the Permissions property setting; this determines what permissions that user has. To work with permissions, you need a SYSTEM.MDA file.

For information about calling DAO directly, see Technical Note 54. For related information, see the topic "UserName Property" in DAO Help.

See Also

Reference

CDaoWorkspace Class
Hierarchy Chart
CDaoWorkspace::SetDefaultUser

Other Resources

CDaoWorkspace Members

CDaoWorkspace::GetVersion

Call this member function to determine the version of the Microsoft Jet database engine in use.

```
static CString PASCAL GetVersion( );
```

Return Value

A CString that indicates the version of the database engine associated with the object.

Remarks

The value returned represents the version number in the form "major.minor"; for example, "3.0". The product version number (for example, 3.0) consists of the version number (3), a period, and the release number (0).

For related information, see the topic "Version Property" in DAO Help.

See Also

Reference

CDaoWorkspace Class
Hierarchy Chart
CDaoDatabase::GetVersion

Other Resources

CDaoWorkspace Members

CDaoWorkspace::GetWorkspaceCount

Call this member function to retrieve the number of DAO workspace objects in the database engine's Workspaces collection.

```
short GetWorkspaceCount( );
```

Return Value

The number of open workspaces in the Workspaces collection.

Remarks

This count does not include any open workspaces not appended to the collection. **GetWorkspaceCount** is useful if you need to loop through all defined workspaces in the Workspaces collection. To obtain information about a given workspace in the collection, see GetWorkspaceInfo. Typical usage is to call **GetWorkspaceCount** for the number of open workspaces, then use that number as a loop index for repeated calls to **GetWorkspaceInfo**.

See Also

Reference

CDaoWorkspace Class

Hierarchy Chart

Other Resources

CDaoWorkspace Members

CDaoWorkspace::GetWorkspaceInfo

Call this member function to obtain various kinds of information about a workspace open in the session.

```
void GetWorkspaceInfo(
   int nIndex,
   CDaoWorkspaceInfo& wkspcinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
void GetWorkspaceInfo(
   LPCTSTR lpszName,
   CDaoWorkspaceInfo& wkspcinfo,
   DWORD dwInfoOptions = AFX_DAO_PRIMARY_INFO
);
```

Parameters

nIndex

The zero-based index of the database object in the Workspaces collection, for lookup by index.

wkspcinfo

A reference to a CDaoWorkspaceInfo object that returns the information requested.

dwInfoOptions

Options that specify which information about the workspace to retrieve. The available options are listed here along with what they cause the function to return:

- **AFX_DAO_PRIMARY_INFO** (Default) Name
- **AFX_DAO_SECONDARY_INFO** Primary information plus: User Name
- **AFX_DAO_ALL_INFO** Primary and secondary information plus: Isolate ODBCTrans

lpszName

The name of the workspace object, for lookup by name. The name is a string with up to 14 characters that uniquely names the new workspace object.

Remarks

For a description of the information returned in *wkspcinfo*, see the CDaoWorkspaceInfo structure. This structure has members that correspond to the items of information listed above in the description of *dwInfoOptions*. When you request information at one level, you get information for prior levels as well.

See Also

Reference

CDaoWorkspace Class
Hierarchy Chart
CDaoWorkspace::GetWorkspaceCount

Other Resources

CDaoWorkspace Members

CDaoWorkspace::Idle

Call **Idle** to provide the database engine with the opportunity to perform background tasks that may not be up-to-date because of intense data processing.

```
static void PASCAL Idle(
   int nAction = dbFreeLocks
);
```

Parameters

nAction

An action to take during the idle processing. Currently the only valid action is **dbFreeLocks**.

Remarks

This is often true in multiuser, multitasking environments in which there is not enough background processing time to keep all records in a recordset current.

Note
Calling **Idle** is not necessary with databases created with version 3.0 of the Microsoft Jet database engine. Use **Idle** only for d atabases created with earlier versions.

Usually, read locks are removed and data in local dynaset-type recordset objects is updated only when no other actions (including mouse movements) are occurring. If you periodically call **Idle**, you provide the database engine with time to catch up on background processing tasks by releasing unneeded read locks. Specifying the **dbFreeLocks** constant as an argument delays processing until all read locks are released.

This member function is not needed in single-user environments unless multiple instances of an application are running. The **Idle** member function may increase performance in a multiuser environment because it forces the database engine to flush data to disk, releasing locks on memory. You can also release read locks by making operations part of a transaction.

For related information, see the topic "Idle Method" in DAO Help.

See Also

Reference

CDaoWorkspace Class

Hierarchy Chart

Other Resources

CDaoWorkspace Members

CDaoWorkspace::IsOpen

Call this member function to determine whether the **CDaoWorkspace** object is open — that is, whether the MFC object has been initialized by a call to Open or a call to Create.

```
BOOL IsOpen( ) const;
```

Return Value

Nonzero if the workspace object is open; otherwise 0.

Remarks

You can call any of the member functions of a workspace that is in an open state.

See Also

Reference

CDaoWorkspace Class

Hierarchy Chart

Other Resources

CDaoWorkspace Members

CDaoWorkspace::Open

Explicitly opens a workspace object associated with DAO's default workspace.

```
virtual void Open(
   LPCTSTR lpszName = NULL
);
```

Parameters

lpszName

The name of the DAO workspace object to open — a string with up to 14 characters that uniquely names the workspace. Accept the default value **NULL** to explicitly open the default workspace. For naming requirements, see the *lpszName* parameter for Create. For related information, see the topic "Name Property" in DAO Help.

Remarks

After constructing a **CDaoWorkspace** object, call this member function to do one of the following:

- Explicitly open the default workspace. Pass **NULL** for *lpszName*.
- Open an existing **CDaoWorkspace** object, a member of the Workspaces collection, by name. Pass a valid name for an existing workspace object.

Open puts the workspace object into an open state and also initializes the database engine if it has not already been initialized for your application.

Although many **CDaoWorkspace** member functions can only be called after the workspace has been opened, the following member functions, which operate on the database engine, are available after construction of the C++ object but before a call to **Open**:

Create	GetVersion	SetDefaultUser
GetIniPath	Idle	SetIniPath
GetLoginTimeout	SetDefaultPassword	SetLoginTimeout

See Also

Reference

CDaoWorkspace Class
Hierarchy Chart
CDaoWorkspace::IsOpen
CDaoWorkspace::CDaoWorkspace
CDaoWorkspace::Create
CDaoWorkspace::Close

Other Resources

CDaoWorkspace Members

CDaoWorkspace::RepairDatabase

Call this member function if you need to attempt to repair a corrupted database that accesses the Microsoft Jet database engine.

```
static void PASCAL RepairDatabase(
   LPCTSTR lpszName
);
```

Parameters

lpszName

The path and filename for an existing Microsoft Jet engine database file. If you omit the path, only the current directory is searched. If your system supports the uniform naming convention (UNC), you can also specify a network path, such as: "\\\\MYSERVER\\MYSHARE\\MYDIR\\MYDB.MDB". (Double backslashes are required in the path string because "\" is the C++ escape character.)

Remarks

You must close the database specified by *lpszName* before you repair it. In a multiuser environment, other users cannot have *lpszName* open while you are repairing it. If *lpszName* is not closed or is not available for exclusive use, an error occurs.

This member function attempts to repair a database that was marked as possibly corrupt by an incomplete write operation. This can occur if an application using the Microsoft Jet database engine is closed unexpectedly because of a power outage or computer hardware problem. If you complete the operation and call the Close member function or you quit the application in a usual way, the database will not be marked as possibly corrupt.

Note
After repairing a database, it is also a good idea to compact it using the CompactDatabase member function to defragment t he file and to recover disk space.

For more information about repairing databases, see the topic "RepairDatabase Method" in DAO Help.

See Also

Reference

CDaoWorkspace Class

Hierarchy Chart

Other Resources

CDaoWorkspace Members

CDaoWorkspace::Rollback

Call this member function to end the current transaction and restore all databases in the workspace to their condition before the transaction was begun.

```
void Rollback( );
```

Remarks

Caution

Within one workspace object, transactions are always global to the workspace and are not limited to only one database or recordset. If you perform operations on more than one database or recordset within a workspace transaction, **Rollback** restores all operations on all of those databases and recordsets.

If you close a workspace object without saving or rolling back any pending transactions, the transactions are automatically rolled back. If you call CommitTrans or **Rollback** without first calling BeginTrans, an error occurs.

Note

When you begin a transaction, the database engine records its operations in a file kept in the directory specified by the TEMP environment variable on the workstation. If the transaction log file exhausts the available storage on your TEMP drive, the database engine will cause MFC to throw a **CDaoException** (DAO error 2004). At this point, if you call **CommitTrans**, an indeterminate number of operations are committed but the remaining uncompleted operations are lost, and the operation has to be restarted. Calling **Rollback** releases the transaction log and rolls back all operations in the transaction.

See Also

Reference

CDaoWorkspace Class
Hierarchy Chart
CDaoRecordset Class

Other Resources

CDaoWorkspace Members

CDaoWorkspace::SetDefaultPassword

Call this member function to set the default password that the database engine uses when a workspace object is created without a specific password.

```
static void PASCAL SetDefaultPassword(
   LPCTSTR lpszPassword
);
```

Parameters

lpszPassword

The default password. A password can be up to 14 characters long and can contain any character except ASCII 0 (null). Passwords are case-sensitive.

Remarks

The default password that you set applies to new workspaces you create after the call. When you create subsequent workspaces, you do not need to specify a password in the Create call.

To use this member function:

1. Construct a **CDaoWorkspace** object but do not call **Create**.
2. Call **SetDefaultPassword** and, if you like, SetDefaultUser.
3. Call **Create** for this workspace object or subsequent ones, without specifying a password.

By default, the DefaultUser property is set to "admin" and the DefaultPassword property is set to an empty string ("").

For more about security, see the topic "Permissions Property" in DAO Help. For related information, see the topics "DefaultPassword Property" and "DefaultUser Property" in DAO Help.

See Also

Reference

CDaoWorkspace Class

Hierarchy Chart

Other Resources

CDaoWorkspace Members

CDaoWorkspace::SetDefaultUser

Call this member function to set the default user name that the database engine uses when a workspace object is created without a specific user name.

```
static void PASCAL SetDefaultUser(
   LPCTSTR lpszDefaultUser
);
```

Parameters

lpszDefaultUser

The default user name. A user name can be 1 – 20 characters long and include alphabetic characters, accented characters, numbers, spaces, and symbols except for: " (quotation marks), / (forward slash), \ (backslash), [ ] (brackets), : (colon), | (pipe), < (less-than sign), > (greater-than sign), + (plus sign), = (equal sign), ; (semicolon), , (comma), ? (question mark), * (asterisk), leading spaces, and control characters (ASCII 00 to ASCII 31). For related information, see the topic "UserName Property" in DAO Help.

Remarks

The default user name that you set applies to new workspaces you create after the call. When you create subsequent workspaces, you do not need to specify a user name in the Create call.

To use this member function:

1. Construct a **CDaoWorkspace** object but do not call **Create**.
2. Call **SetDefaultUser** and, if you like, SetDefaultPassword.
3. Call **Create** for this workspace object or subsequent ones, without specifying a user name.

By default, the DefaultUser property is set to "admin" and the DefaultPassword property is set to an empty string ("").

For related information, see the topics "DefaultUser Property" and "DefaultPassword Property" in DAO Help.

See Also

Reference

CDaoWorkspace Class

Hierarchy Chart

Other Resources

CDaoWorkspace Members

CDaoWorkspace::SetIniPath

Call this member function to specify the location of Windows registry settings for the Microsoft Jet database engine.

```
static void PASCAL SetIniPath(
   LPCTSTR lpszRegistrySubKey
);
```

Parameters

lpszRegistrySubkey

A string containing the name of a Windows registry subkey for the location of Microsoft Jet database engine settings or parameters needed for installable ISAM databases.

Remarks

Call **SetIniPath** only if you need to specify special settings. For more information, see the topic "IniPath Property" in DAO Help.

Note

Call **SetIniPath** during application installation, not when the application runs. **SetIniPath** must be called before you open any workspaces, databases, or recordsets; otherwise, MFC throws an exception.

You can use this mechanism to configure the database engine with user-provided registry settings. The scope of this attribute is limited to your application and cannot be changed without restarting your application.

See Also

Reference

CDaoWorkspace Class

Hierarchy Chart

Other Resources

CDaoWorkspace Members

CDaoWorkspace::SetIsolateODBCTrans

Call this member function to set the value of the DAO IsolateODBCTrans property for the workspace.

```
void SetIsolateODBCTrans(
   BOOL bIsolateODBCTrans
);
```

Parameters

bIsolateODBCTrans

Pass **TRUE** if you want to begin isolating ODBC transactions. Pass **FALSE** if you want to stop isolating ODBC transactions.

Remarks

In some situations, you might need to have multiple simultaneous transactions pending on the same ODBC database. To do this, you need to open a separate workspace for each transaction. Although each workspace can have its own ODBC connection to the database, this slows system performance. Because transaction isolation is not normally required, ODBC connections from multiple workspace objects opened by the same user are shared by default.

Some ODBC servers, such as Microsoft SQL Server, do not allow simultaneous transactions on a single connection. If you need to have more than one transaction at a time pending against such a database, set the IsolateODBCTrans property to **TRUE** on each workspace as soon as you open it. This forces a separate ODBC connection for each workspace.

See Also

Reference

CDaoWorkspace Class
Hierarchy Chart
CDaoWorkspace::GetIsolateODBCTrans

Other Resources

CDaoWorkspace Members

CDaoWorkspace::SetLoginTimeout

Call this member function to set the value of the DAO LoginTimeout property for the workspace.

```
static void PASCAL SetLoginTimeout(
   short nSeconds
);
```

Parameters

nSeconds

The number of seconds before an error occurs when you attempt to log in to an ODBC database.

Remarks

This value represents the number of seconds before an error occurs when you attempt to log in to an ODBC database. The default LoginTimeout setting is 20 seconds. When LoginTimeout is set to 0, no timeout occurs and the communication with the data source might hang.

When you are attempting to log in to an ODBC database, such as Microsoft SQL Server, the connection may fail as a result of network errors or because the server is not running. Rather than waiting for the default 20 seconds to connect, you can specify how long the database engine waits before it produces an error. Logging on to the server happens implicitly as part of a number of different events, such as running a query on an external server database. The timeout value is determined by the current setting of the LoginTimeout property.

For related information, see the topic "LoginTimeout Property" in DAO Help.

See Also

Reference

CDaoWorkspace Class
Hierarchy Chart
CDaoWorkspace::GetLoginTimeout

Other Resources

CDaoWorkspace Members

CDaoWorkspace Data Members

For information about the data members in **CDaoWorkspace**, see CDaoWorkspace Members.

CDaoWorkspace::m_pDAOWorkspace

A pointer to the underlying DAO workspace object.

Remarks

Use this data member if you need direct access to the underlying DAO object. You can call the DAO object's interfaces through this pointer.

For information about accessing DAO objects directly, see Technical Note 54.

See Also

Reference

CDaoWorkspace Class

Hierarchy Chart

Other Resources

CDaoWorkspace Members

CDatabase Class

Represents a connection to a data source, through which you can operate on the data source.

```
class CDatabase : public CObject
```

Remarks

A data source is a specific instance of data hosted by some database management system (DBMS). Examples include Microsoft SQL Server, Microsoft Access, Borland dBASE, and xBASE. You can have one or more **CDatabase** objects active at a time in your application.

Note
If you are working with the Data Access Objects (DAO) classes rather than the Open Database Connectivity (ODBC) classes, use class CDaoDatabase instead. For more information, see the article Overview: Database Programming.

To use **CDatabase**, construct a **CDatabase** object and call its **OpenEx** member function. This opens a connection. When you then construct **CRecordset** objects for operating on the connected data source, pass the recordset constructor a pointer to your **CDatabase** object. When you finish using the connection, call the **Close** member function and destroy the **CDatabase** object. **Close** closes any recordsets you have not closed previously.

For more information about **CDatabase**, see the articles Data Source (ODBC) and Overview: Database Programming.

Requirements

Header: afxdb.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class
Hierarchy Chart
CRecordset Class

Other Resources

CDatabase Members

CDatabase Members

Base Class Members

CObject Members

Data Members

m_hdbc	Open Database Connectivity (ODBC) connection handle to a data source. Type **HDBC**.

Construction

CDatabase	Constructs a **CDatabase** object. You must initialize the object by calling **OpenEx** or **Open**.
Close	Closes the data source connection.
Open	Establishes a connection to a data source (through an ODBC driver).
OpenEx	Establishes a connection to a data source (through an ODBC driver).

Database Attributes

CanTransact	Returns nonzero if the data source supports transactions.
CanUpdate	Returns nonzero if the **CDatabase** object is updatable (not read-only).
GetBookmarkPersistence	Identifies the operations through which bookmarks persist on recordset objects.
GetConnect	Returns the ODBC connection string used to connect the **CDatabase** object to a data source.
GetCursorCommitBehavior	Identifies the effect of committing a transaction on an open recordset object.
GetCursorRollbackBehavior	Identifies the effect of rolling back a transaction on an open recordset object.
GetDatabaseName	Returns the name of the database currently in use.
IsOpen	Returns nonzero if the **CDatabase** object is currently connected to a data source.
SetLoginTimeout	Sets the number of seconds after which a data source connection attempt will time out.
SetQueryTimeout	Sets the number of seconds after which database query operations will time out. Affects all subsequent recordset **Open**, **AddNew**, **Edit**, and **Delete** calls.

Database Operations

BeginTrans	Starts a "transaction" — a series of reversible calls to the **AddNew**, **Edit**, **Delete**, and **Update** member functions of class **CRecordset** — on the connected data source. The data source must support transactions for **BeginTrans** to have any effect.
BindParameters	Allows you to bind parameters before calling **CDatabase::ExecuteSQL**.
Cancel	Cancels an asynchronous operation or a process from a second thread.
CommitTrans	Completes a transaction begun by **BeginTrans**. Commands in the transaction that alter the data source are carried out.
ExecuteSQL	Executes a SQL statement. No data records are returned.

Rollback	Reverses changes made during the current transaction. The data source returns to its previous state, as defined at the **BeginTrans** call, unaltered.

Database Overridables

OnSetOptions	Called by the framework to set standard connection options. The default implementation sets the query timeout value. You can establish these options ahead of time by calling **SetQueryTimeout**.

See Also

Reference

CDatabase Class

Hierarchy Chart

CDatabase Member Functions

For information about the member functions in **CDatabase**, see CDatabase Members.

CDatabase::BeginTrans

Call this member function to begin a transaction with the connected data source.

```
BOOL BeginTrans( );
```

Return Value

Nonzero if the call was successful and changes are committed only manually; otherwise 0.

Remarks

A transaction consists of one or more calls to the **AddNew**, **Edit**, **Delete**, and **Update** member functions of a **CRecordset** object. Before beginning a transaction, the **CDatabase** object must already have been connected to the data source by calling its **OpenEx** or **Open** member function. To end the transaction, call CommitTrans to accept all changes to the data source (and carry them out) or call Rollback to abort the entire transaction. Call **BeginTrans** after you open any recordsets involved in the transaction and as close to the actual update operations as possible.

Caution
Depending on your ODBC driver, opening a recordset before calling **BeginTrans** may cause problems when calling **Rollback**. You should check the specific driver you are using. For example, when using the Microsoft Access driver included in the Microsoft ODBC Desktop Driver Pack 3.0, you must account for the Jet database engine's requirement that you should not begin a transaction on any database that has an open cursor. In the MFC database classes, an open cursor means an open **CRecordset** object. For more information, see Technical Note 68.

BeginTrans may also lock data records on the server, depending on the requested concurrency and the capabilities of the data source. For information about locking data, see the article Recordset: Locking Records (ODBC).

User-defined transactions are explained in the article Transaction (ODBC).

BeginTrans establishes the state to which the sequence of transactions can be rolled back (reversed). To establish a new state for rollbacks, commit any current transaction, then call **BeginTrans** again.

Caution
Calling **BeginTrans** again without calling **CommitTrans** or **Rollback** is an error.

Call the CanTransact member function to determine whether your driver supports transactions for a given database. You should also call GetCursorCommitBehavior and GetCursorRollbackBehavior to determine the support for cursor preservation.

For more information about transactions, see the article Transaction (ODBC).

Example

See the article Transaction: Performing a Transaction in a Recordset (ODBC).

See Also

Reference

CDatabase Class
Hierarchy Chart
CDatabase::CommitTrans
CDatabase::Rollback
CRecordset::CanTransact

Other Resources

CDatabase Members

CDatabase::BindParameters

Override **BindParameters** when you need to bind parameters before calling CDatabase::ExecuteSQL.

```
virtual void BindParameters(
   HSTMT hstmt
);
```

Parameters

hstmt

The ODBC statement handle for which you want to bind parameters.

Remarks

This approach is useful when you do not need the result set from a stored procedure.

In your override, call **SQLBindParameters** and related ODBC functions to bind the parameters. MFC calls your override before your call to **ExecuteSQL**. You do not need to call **SQLPrepare**; **ExecuteSQL** calls **SQLExecDirect** and destroys the **hstmt**, which is used only once.

See Also

Reference

CDatabase Class

Hierarchy Chart

Other Resources

CDatabase Members

CDatabase::Cancel

Call this member function to request that the data source cancel either an asynchronous operation in progress or a process from a second thread.

```
void Cancel( );
```

Remarks

Note that the MFC ODBC classes no longer use asynchronous processing; to perform an asychronous operation, you must directly call the ODBC API function SQLSetConnectOption. For more information, see Asynchronous Execution in the *Platform SDK*.

See Also

Reference

CDatabase Class

Hierarchy Chart

Other Resources

CDatabase Members

CDatabase::CanTransact

Call this member function to determine whether the database allows transactions.

```
BOOL CanTransact( ) const;
```

Return Value

Nonzero if recordsets using this **CDatabase** object allow transactions; otherwise 0.

Remarks

For information about transactions, see the article Transaction (ODBC).

See Also

Reference

CDatabase Class
Hierarchy Chart
CDatabase::BeginTrans
CDatabase::CommitTrans
CDatabase::Rollback

Other Resources

CDatabase Members

CDatabase::CanUpdate

Call this member function to determine whether the **CDatabase** object allows updates.

```
BOOL CanUpdate( ) const;
```

Return Value

Nonzero if the **CDatabase** object allows updates; otherwise 0, indicating either that you passed **TRUE** in *bReadOnly* when you opened the **CDatabase** object or that the data source itself is read-only. The data source is read-only if a call to the ODBC API function **SQLGetInfo** for **SQL_DATASOURCE_READ_ONLY** returns "y".

Remarks

Not all drivers support updates.

See Also

Reference

CDatabase Class

Hierarchy Chart

Other Resources

CDatabase Members

CDatabase::CDatabase

Constructs a **CDatabase** object.

```
CDatabase( );
```

Remarks

After constructing the object, you must call its **OpenEx** or **Open** member function to establish a connection to a specified data source.

You may find it convenient to embed the **CDatabase** object in your document class.

Example

```
// This example illustrates using CDatabase
// in a CDocument-derived class.

class CMyDocument : public CDocument
{
public:
   // Declare a CDatabase embedded in the document
   CDatabase m_dbCust;
   // ...
};

// ...

// Initialize when needed
CDatabase* CMyDocument::GetDatabase( )
{
   // Connect the object to a data source
   if( !m_dbCust.IsOpen( ) &&
       !m_dbCust.OpenEx( NULL ) )
      return NULL;

   return &m_dbCust;
}
```

See Also

Reference

CDatabase Class
Hierarchy Chart
CDatabase::OpenEx
CDatabase::Open

Other Resources

CDatabase Members

CDatabase::Close

Call this member function if you want to disconnect from a data source.

```
virtual void Close( );
```

Remarks

You must close any recordsets associated with the **CDatabase** object before you call this member function. Because **Close** does not destroy the **CDatabase** object, you can reuse the object by opening a new connection to the same data source or a different data source.

All pending **AddNew** or **Edit** statements of recordsets using the database are canceled, and all pending transactions are rolled back. Any recordsets dependent on the **CDatabase** object are left in an undefined state.

Example

```
// Close the current connection
m_dbCust.Close( );

// Perhaps connect the object to a
// different data source
m_dbCust.OpenEx("DSN=MYDATASOURCE;UID=JOES");
```

See Also

Reference

CDatabase Class
Hierarchy Chart
CDatabase::OpenEx
CDatabase::Open

Other Resources

CDatabase Members

CDatabase::CommitTrans

Call this member function upon completing transactions.

```
BOOL CommitTrans( );
```

Return Value

Nonzero if the updates were successfully committed; otherwise 0. If **CommitTrans** fails, the state of the data source is undefined. You must check the data to determine its state.

Remarks

A transaction consists of a series of calls to the **AddNew**, **Edit**, **Delete**, and **Update** member functions of a **CRecordset** object that began with a call to the BeginTrans member function. **CommitTrans** commits the transaction. By default, updates are committed immediately; calling **BeginTrans** causes commitment of updates to be delayed until **CommitTrans** is called.

Until you call **CommitTrans** to end a transaction, you can call the Rollback member function to abort the transaction and leave the data source in its original state. To begin a new transaction, call **BeginTrans** again.

For more information about transactions, see the article Transaction (ODBC).

Example

See the article Transaction: Performing a Transaction in a Recordset (ODBC).

See Also

Reference

CDatabase Class
Hierarchy Chart
CDatabase::BeginTrans
CDatabase::Rollback

Other Resources

CDatabase Members

CDatabase::ExecuteSQL

Call this member function when you need to execute a SQL command directly.

```
void ExecuteSQL(
   LPCTSTR lpszSQL
);
```

Parameters

lpszSQL

Pointer to a null-terminated string containing a valid SQL command to execute. You can pass a CString.

Remarks

Create the command as a null-terminated string. **ExecuteSQL** does not return data records. If you want to operate on records, use a recordset object instead.

Most of your commands for a data source are issued through recordset objects, which support commands for selecting data, inserting new records, deleting records, and editing records. However, not all ODBC functionality is directly supported by the database classes, so you may at times need to make a direct SQL call with **ExecuteSQL**.

Example

```
CString strCmd = "UPDATE Taxes SET Federal = 36%";

TRY
{
   m_dbCust.ExecuteSQL( strCmd );
}

CATCH(CDBException, e)
{
   // The error code is in e->m_nRetCode
}

END_CATCH
```

See Also

Reference

CDatabase Class
Hierarchy Chart
CDatabase::SetLoginTimeout
CRecordset Class

Other Resources

CDatabase Members

CDatabase::GetBookmarkPersistence

Call this member function to determine the persistence of bookmarks on a recordset object after certain operations.

```
DWORD GetBookmarkPersistence( ) const;
```

Return Value

A bitmask that identifies the operations through which bookmarks persist on a recordset object. For details, see Remarks.

Remarks

For example, if you call **CRecordset::GetBookmark** and then call **CRecordset::Requery**, the bookmark obtained from **GetBookmark** may no longer be valid. You should call **GetBookmarkPersistence** before calling **CRecordset::SetBookmark**.

The following table lists the bitmask values that can be combined for the return value of **GetBookmarkPersistence**.

Bitmask value	Bookmark persistence
SQL_BP_CLOSE	Bookmarks are valid after a **Requery** operation.
SQL_BP_DELETE	The bookmark for a row is valid after a **Delete** operation on that row.
SQL_BP_DROP	Bookmarks are valid after a **Close** operation.
SQL_BP_SCROLL	Bookmarks are valid after any **Move** operation. This simply identifies if bookmarks are supported on the recordset, as returned by **CRecordset::CanBookmark**.
SQL_BP_TRANSACTION	Bookmarks are valid after a transaction is committed or rolled back.
SQL_BP_UPDATE	The bookmark for a row is valid after an **Update** operation on that row.
SQL_BP_OTHER_HSTMT	Bookmarks associated with one recordset object are valid on a second recordset.

For more information about this return value, see the ODBC API function **SQLGetInfo** in the *Platform SDK*. For more information about bookmarks, see the article Recordset: Bookmarks and Absolute Positions (ODBC).

See Also

Reference

CDatabase Class
Hierarchy Chart
CRecordset Class
CRecordset::CanBookmark
CRecordset::GetBookmark
CRecordset::SetBookmark

Other Resources

CDatabase Members

CDatabase::GetConnect

Call this member function to retrieve the connection string used during the call to **OpenEx** or **Open** that connected the **CDatabase** object to a data source.

```
const CString& GetConnect( ) const;
```

Return Value

A **const** reference to a CString containing the connection string if **OpenEx** or **Open** has been called; otherwise, an empty string.

Remarks

See CDatabase::Open for a description of how the connection string is created.

See Also

Reference

CDatabase Class
Hierarchy Chart
CDatabase::OpenEx
CDatabase::Open

Other Resources

CDatabase Members

CDatabase::GetCursorCommitBehavior

Call this member function to determine how a CommitTrans operation affects cursors on open recordset objects.

```
int GetCursorCommitBehavior( ) const;
```

Return Value

A value indicating the effect of transactions on open recordset objects. For details, see Remarks.

Remarks

The following table lists the possible return values for **GetCursorCommitBehavior** and the corresponding effect on the open recordset.

Return value	Effect on CRecordset objects
SQL_CB_CLOSE	Call **CRecordset::Requery** immediately following the transaction commit.
SQL_CB_DELETE	Call **CRecordset::Close** immediately following the transaction commit.
SQL_CB_PRESERVE	Proceed normally with **CRecordset** operations.

For more information about this return value, see the ODBC API function **SQLGetInfo** in the *Platform SDK*. For more information about transactions, see the article Transaction (ODBC).

See Also

Reference

CDatabase Class
Hierarchy Chart
CDatabase::GetCursorRollbackBehavior
CDatabase::CanTransact
CDatabase::BeginTrans
CDatabase::CommitTrans
CDatabase::Rollback
CRecordset Class

Other Resources

CDatabase Members

CDatabase::GetCursorRollbackBehavior

Call this member function to determine how a Rollback operation affects cursors on open recordset objects.

```
int GetCursorRollbackBehavior( ) const;
```

Return Value

A value indicating the effect of transactions on open recordset objects. For details, see Remarks.

Remarks

The following table lists the possible return values for **GetCursorRollbackBehavior** and the corresponding effect on the open recordset.

Return value	Effect on CRecordset objects
SQL_CB_CLOSE	Call **CRecordset::Requery** immediately following the transaction rollback.
SQL_CB_DELETE	Call **CRecordset::Close** immediately following the transaction rollback.
SQL_CB_PRESERVE	Proceed normally with **CRecordset** operations.

For more information about this return value, see the ODBC API function **SQLGetInfo** in the *Platform SDK*. For more information about transactions, see the article Transaction (ODBC).

See Also

Reference

CDatabase Class
Hierarchy Chart
CDatabase::GetCursorCommitBehavior
CDatabase::CanTransact
CDatabase::BeginTrans
CDatabase::CommitTrans
CDatabase::Rollback
CRecordset Class

Other Resources

CDatabase Members

CDatabase::GetDatabaseName

Call this member function to retrieve the name of the currently connected database (provided that the data source defines a named object called "database").

```
CString GetDatabaseName( ) const;
```

Return Value

A CString containing the database name if successful; otherwise, an empty **CString**.

Remarks

This is not the same as the data source name (DSN) specified in the **OpenEx** or **Open** call. What **GetDatabaseName** returns depends on ODBC. In general, a database is a collection of tables. If this entity has a name, **GetDatabaseName** returns it.

You might, for example, want to display this name in a heading. If an error occurs while retrieving the name from ODBC, **GetDatabaseName** returns an empty **Cstring**.

See Also

Reference

CDatabase Class
Hierarchy Chart
CDatabase::OpenEx
CDatabase::Open
CDatabase::GetConnect

Other Resources

CDatabase Members

CDatabase::IsOpen

Call this member function to determine whether the **CDatabase** object is currently connected to a data source.

```
BOOL IsOpen( ) const;
```

Return Value

Nonzero if the **CDatabase** object is currently connected; otherwise 0.

See Also

Reference

CDatabase Class
Hierarchy Chart
CDatabase::OpenEx
CDatabase::Open

Other Resources

CDatabase Members

CDatabase::OnSetOptions

The framework calls this member function when directly executing a SQL statement with the **ExecuteSQL** member function.

```
virtual void OnSetOptions(
   HSTMT hstmt
);
```

Parameters

hstmt

The ODBC statement handle for which options are being set.

Remarks

CRecordset::OnSetOptions also calls this member function.

OnSetOptions sets the login timeout value. If there have been previous calls to the **SetQueryTimeout** and member function, **OnSetOptions** reflects the current values; otherwise, it sets default values.

Note
Prior to MFC 4.2, **OnSetOptions** also set the processing mode to either snychronous or asynchronous. Beginning with MFC 4.2, all operations are synchronous. To perform an asynchronous operation, you must make a direct call to the ODBC API function **SQLSetPos**.

You do not need to override **OnSetOptions** to change the timeout value. Instead, to customize the query timeout value, call **SetQueryTimeout** before creating a recordset; **OnSetOptions** will use the new value. The values set apply to subsequent operations on all recordsets or direct SQL calls.

Override **OnSetOptions** if you want to set additional options. Your override should call the base class **OnSetOptions** either before or after you call the ODBC API function **SQLSetStmtOption**. Follow the method illustrated in the framework's default implementation of **OnSetOptions**.

See Also

Reference

CDatabase Class
Hierarchy Chart
CDatabase::ExecuteSQL
CDatabase::SetQueryTimeout
CRecordset::OnSetOptions

Other Resources

CDatabase Members

CDatabase::Open

Call this member function to initialize a newly constructed **CDatabase** object.

```
virtual BOOL Open(
   LPCTSTR lpszDSN,
   BOOL bExclusive = FALSE,
   BOOL bReadOnly = FALSE,
   LPCTSTR lpszConnect = _T("ODBC;"),
   BOOL bUseCursorLib = TRUE
);
```

Parameters

lpszDSN

Specifies a data source name — a name registered with ODBC through the ODBC Administrator program. If a DSN value is specified in *lpszConnect* (in the form "DSN=<data-source>"), it must not be specified again in *lpszDSN*. In this case, *lpszDSN* should be **NULL**. Otherwise, you can pass **NULL** if you want to present the user with a Data Source dialog box in which the user can select a data source. For further information, see Remarks.

bExclusive

Not supported in this version of the class library. Currently, an assertion fails if this parameter is **TRUE**. The data source is always opened as shared (not exclusive).

bReadOnly

TRUE if you intend the connection to be read-only and to prohibit updates to the data source. All dependent recordsets inherit this attribute. The default value is **FALSE**.

lpszConnect

Specifies a connection string. The connection string concatenates information, possibly including a data source name, a user ID valid on the data source, a user authentication string (password, if the data source requires one), and other information. The whole connection string must be prefixed by the string "ODBC;" (uppercase or lowercase). The "ODBC;" string is used to indicate that the connection is to an ODBC data source; this is for upward compatibility when future versions of the class library might support non-ODBC data sources.

bUseCursorLib

TRUE if you want the ODBC Cursor Library DLL to be loaded. The cursor library masks some functionality of the underlying ODBC driver, effectively preventing the use of dynasets (if the driver supports them). The only cursors supported if the cursor library is loaded are static snapshots and forward-only cursors. The default value is **TRUE**. If you plan to create a recordset object directly from **CRecordset** without deriving from it, you should not load the cursor library.

Return Value

Nonzero if the connection is successfully made; otherwise 0 if the user chooses Cancel when presented a dialog box asking for more connection information. In all other cases, the framework throws an exception.

Remarks

Your database object must be initialized before you can use it to construct a recordset object.

Note
Calling the OpenEx member function is the preferred way to connect to a data source and initialize your database object.

If the parameters in your **Open** call do not contain enough information to make the connection, the ODBC driver opens a dialog box to obtain the necessary information from the user. When you call **Open**, your connection string, *lpszConnect*, is stored privately in the **CDatabase** object and is available by calling the GetConnect member function.

If you wish, you can open your own dialog box before you call **Open** to get information from the user, such as a password, then add that information to the connection string you pass to **Open**. Or you might want to save the connection string you pass so you can reuse it the next time your application calls **Open** on a **CDatabase** object.

You can also use the connection string for multiple levels of login authorization (each for a different **CDatabase** object) or to convey other data source-specific information. For more information about connection strings, see Chapter 5 in the *Platform SDK*.

It is possible for a connection attempt to time out if, for example, the DBMS host is unavailable. If the connection attempt fails, **Open** throws a **CDBException**.

Example

```
// Embed a CDatabase object
// in your document class
CDatabase m_dbCust;

// Connect the object to a
// data source (no password)
// the ODBC connection dialog box
// will always remain hidden
m_dbCust.Open( _T( "MYDATASOURCE" ), FALSE,
               FALSE, _T( "ODBC;UID=JOES" ),

// ...Or, query the user for all
// connection information
m_dbCust.Open( NULL );
```

See Also

Reference

CDatabase Class
Hierarchy Chart
CDatabase::OpenEx
CDatabase::CDatabase
CDatabase::Close
CDBException Class
CRecordset::Open

Other Resources

CDatabase Members

CDatabase::OpenEx

Call this member function to initialize a newly constructed **CDatabase** object.

```
virtual BOOL OpenEx(
   LPCTSTR lpszConnectString,
   DWORD dwOptions = 0
);
```

Parameters

lpszConnectString

Specifies an ODBC connection string. This includes the data source name as well as other optional information, such as a user ID and password. For example, "DSN=SQLServer_Source;UID=SA;PWD=abc123" is a possible connection string. Note that if you pass **NULL** for *lpszConnectString*, a Data Source dialog box will prompt the user to select a data source.

dwOptions

A bitmask which specifies a combination of the following values. The default value is 0, meaning that the database will be opened as shared with write access, the ODBC Cursor Library DLL will not be loaded, and the ODBC connection dialog box will display only if there is not enough information to make the connection.

- **CDatabase::openExclusive** Not supported in this version of the class library. A data source is always opened as shared (not exclusive). Currently, an assertion fails if you specify this option.
- **CDatabase::openReadOnly** Open the data source as read-only.
- **CDatabase::useCursorLib** Load the ODBC Cursor Library DLL. The cursor library masks some functionality of the underlying ODBC driver, effectively preventing the use of dynasets (if the driver supports them). The only cursors supported if the cursor library is loaded are static snapshots and forward-only cursors. If you plan to create a recordset object directly from **CRecordset** without deriving from it, you should not load the cursor library.
- **CDatabase::noOdbcDialog** Do not display the ODBC connection dialog box, regardless of whether enough connection information is supplied.
- **CDatabase::forceOdbcDialog** Always display the ODBC connection dialog box.

Return Value

Nonzero if the connection is successfully made; otherwise 0 if the user chooses Cancel when presented a dialog box asking for more connection information. In all other cases, the framework throws an exception.

Remarks

Your database object must be initialized before you can use it to construct a recordset object.

If the *lpszConnectString* parameter in your **OpenEx** call does not contain enough information to make the connection, the ODBC driver opens a dialog box to obtain the necessary information from the user, provided you have not set **CDatabase::noOdbcDialog** or **CDatabase::forceOdbcDialog** in the *dwOptions* parameter. When you call **OpenEx**, your connection string, *lpszConnectString*, is stored privately in the **CDatabase** object and is available by calling the GetConnect member function.

If you wish, you can open your own dialog box before you call **OpenEx** to get information from the user, such as a password, and then add that information to the connection string you pass to **OpenEx**. Or you might want to save the connection string you pass so you can reuse it the next time your application calls **OpenEx** on a **CDatabase** object.

You can also use the connection string for multiple levels of login authorization (each for a different **CDatabase** object) or to convey other data source-specific information. For more information about connection strings, see Chapter 6 in the *ODBC Programmer's Reference*.

It is possible for a connection attempt to time out if, for example, the DBMS host is unavailable. If the connection attempt fails, **OpenEx** throws a **CDBException**.

Example

```
// Embed a CDatabase object
// in your document class
CDatabase m_dbCust;

// Connect the object to a
// read-only data source where
// the ODBC connection dialog box
// will always remain hidden
m_dbCust.OpenEx( _T( "DSN=MYDATASOURCE;UID=JOES" ),
                 CDatabase::openReadOnly |
                 CDatabase::noOdbcDialog );
```

See Also

Reference

CDatabase Class
Hierarchy Chart
CDatabase::Open
CDatabase::CDatabase
CDatabase::Close
CDBException Class
CRecordset::Open

Other Resources

CDatabase Members

CDatabase::Rollback

Call this member function to reverse the changes made during a transaction.

```
BOOL Rollback( );
```

Return Value

Nonzero if the transaction was successfully reversed; otherwise 0. If a **Rollback** call fails, the data source and transaction states are undefined. If **Rollback** returns 0, you must check the data source to determine its state.

Remarks

All **CRecordset AddNew**, **Edit**, **Delete**, and **Update** calls executed since the last BeginTrans are rolled back to the state that existed at the time of that call.

After a call to **Rollback**, the transaction is over, and you must call **BeginTrans** again for another transaction. The record that was current before you called **BeginTrans** becomes the current record again after **Rollback**.

After a rollback, the record that was current before the rollback remains current. For details about the state of the recordset and the data source after a rollback, see the article Transaction (ODBC).

Example

See the article Transaction: Performing a Transaction in a Recordset (ODBC).

See Also

Reference

CDatabase Class
Hierarchy Chart
CDatabase::BeginTrans
CDatabase::CommitTrans

Other Resources

CDatabase Members

CDatabase::SetLoginTimeout

Call this member function — before you call **OpenEx** or **Open** — to override the default number of seconds allowed before an attempted data source connection times out.

```
void SetLoginTimeout(
   DWORD dwSeconds
);
```

Parameters

dwSeconds

The number of seconds to allow before a connection attempt times out.

Remarks

A connection attempt might time out if, for example, the DBMS is not available. Call **SetLoginTimeout** after you construct the uninitialized **CDatabase** object but before you call **OpenEx** or **Open**.

The default value for login timeouts is 15 seconds. Not all data sources support the ability to specify a login timeout value. If the data source does not support timeout, you get trace output but not an exception. A value of 0 means "infinite."

See Also

Reference

CDatabase Class
Hierarchy Chart
CDatabase::OnSetOptions
CDatabase::SetQueryTimeout

Other Resources

CDatabase Members

CDatabase::SetQueryTimeout

Call this member function to override the default number of seconds to allow before subsequent operations on the connected data source time out.

```
void SetQueryTimeout(
   DWORD dwSeconds
);
```

Parameters

dwSeconds

The number of seconds to allow before a query attempt times out.

Remarks

An operation might time out due to network access problems, excessive query processing time, and so on. Call **SetQueryTimeout** prior to opening your recordset or prior to calling the recordset's **AddNew**, **Update** or **Delete** member functions if you want to change the query timeout value. The setting affects all subsequent **Open**, **AddNew**, **Update**, and **Delete** calls to any recordsets associated with this **CDatabase** object. Changing the query timeout value for a recordset after opening does not change the value for the recordset. For example, subsequent **Move** operations do not use the new value.

The default value for query timeouts is 15 seconds. Not all data sources support the ability to set a query timeout value. If you set a query timeout value of 0, no timeout occurs; the communication with the data source may hang. This behavior may be useful during development. If the data source does not support timeout, you get trace output but not an exception.

See Also

Reference

CDatabase Class
Hierarchy Chart
CDatabase::SetLoginTimeout

Other Resources

CDatabase Members

CDatabase Data Members

For information about the data members in **CDatabase**, see CDatabase Members.

CDatabase::m_hdbc

Contains a public handle to an ODBC data source connection — a "connection handle."

Remarks

Normally, you will have no need to access this member variable directly. Instead, the framework allocates the handle when you call **OpenEx** or **Open**. The framework deallocates the handle when you call the **delete** operator on the **CDatabase** object. Note that the **Close** member function does not deallocate the handle.

Under some circumstances, however, you may need to use the handle directly. For example, if you need to call ODBC API functions directly rather than through class **CDatabase**, you may need a connection handle to pass as a parameter. See the code example below.

Example

```
// Using m_hdbc for a direct ODBC API call.
// m_db is the CDatabase object; m_hdbc is
// its HDBC member variable
nRetcode = ::SQLGetInfo( m_db.m_hdbc,
                         SQL_ODBC_SQL_CONFORMANCE,
                         &nValue,
                         sizeof( nValue ),
                         &cbValue );
```

See Also

Reference

CDatabase Class
Hierarchy Chart
CDatabase::OpenEx
CDatabase::Open
CDatabase::Close

Other Resources

CDatabase Members

CDataExchange Class

Supports the dialog data exchange (DDX) and dialog data validation (DDV) routines used by the Microsoft Foundation classes.

```
class CDataExchange
```

Remarks

CDataExchange does not have a base class.

Use this class if you are writing data exchange routines for custom data types or controls, or if you are writing your own data validation routines. For more information on writing your own DDX and DDV routines, see Technical Note 26. For an overview of DDX and DDV, see Dialog Data Exchange and Validation and Dialog Boxes.

A **CDataExchange** object provides the context information needed for DDX and DDV to take place. The flag **m_bSaveAndValidate** is **FALSE** when DDX is used to fill the initial values of dialog controls from data members. The flag **m_bSaveAndValidate** is **TRUE** when DDX is used to set the current values of dialog controls into data members and when DDV is used to validate the data values. If the DDV validation fails, the DDV procedure will display a message box explaining the input error. The DDV procedure will then call **Fail** to reset the focus to the offending control and throw an exception to stop the validation process.

Requirements

Header: afxwin.h

See Also

Tasks

VIEWEX Sample: Demonstrates Multiple Views, Scroll Views, and Splitter Windows

Reference

Hierarchy Chart

CWnd::DoDataExchange

CWnd::UpdateData

Other Resources

CDataExchange Members

CDataExchange Members

Data Members

m_bSaveAndValidate	Flag for the direction of DDX and DDV.
m_pDlgWnd	The dialog box or window where the data exchange takes place.

Construction

CDataExchange	Constructs a **CDataExchange** object.

Operations

Fail	Called when validation fails. Resets focus to the previous control and throws an exception.
PrepareCtrl	Prepares the specified control for data exchange or validation. Use for nonedit controls.
PrepareEditCtrl	Prepares the specified edit control for data exchange or validation.
PrepareOleCtrl	Prepares the specified OLE control for data exchange or validation. Use for nonedit controls.

See Also

Reference

CDataExchange Class

Hierarchy Chart

CDataExchange Member Functions

For information about the member functions in **CDataExchange**, see CDataExchange Members.

CDataExchange::CDataExchange

Call this member function to construct a **CDataExchange** object.

```
CDataExchange(
   CWnd* pDlgWnd,
   BOOL bSaveAndValidate
);
```

Parameters

pDlgWnd

A pointer to the parent window that contains the control. Usually this is a CDialog-derived object.

bSaveAndValidate

If **TRUE**, this object validates data, then writes data from the controls to the members. If **FALSE**, this object will move data from members to controls.

Remarks

Construct a **CDataExchange** object yourself to store extra information in the data exchange object to pass to your window's CWnd::DoDataExchange member function.

Example

```
CYourDataExchange dx(this, FALSE);

TRY

{

DoDataExchange(&dx);

}

CATCH(CUserException, e)

{

// some part of the exchange went wrong

// but the user has already been notified

}
```

See Also

Reference

CDataExchange Class

Hierarchy Chart

Other Resources

CDataExchange Members

CDataExchange::Fail

The framework calls this member function when a dialog data validation (DDV) operation fails.

```
void Fail( );
```

Remarks

Fail restores the focus and selection to the control whose validation failed (if there is a control to restore). **Fail** then throws an exception of type CUserException to stop the validation process. The exception causes a message box explaining the error to be displayed. After DDV validation fails, the user can reenter data in the offending control.

Implementors of custom DDV routines can call **Fail** from their routines when a validation fails.

For more information on writing your own DDX and DDV routines, see Technical Note 26. For an overview of DDX and DDV, see Dialog Data Exchange and Validation and Dialog Box Topics.

See Also

Reference

CDataExchange Class
Hierarchy Chart
CDataExchange::PrepareCtrl
CDataExchange::PrepareEditCtrl

Other Resources

CDataExchange Members

CDataExchange::PrepareCtrl

The framework calls this member function to prepare the specified control for dialog data exchange (DDX) and validation (DDV).

```
HWND PrepareCtrl(
   int nIDC
);
```

Parameters

nIDC

The ID of the control to be prepared for DDX or DDV.

Return Value

The **HWND** of the control being prepared for DDX or DDV.

Remarks

Use PrepareEditCtrl instead for edit controls; use this member function for all other controls.

Preparation consists of storing the control's **HWND** in the **CDataExchange** class. The framework uses this handle to restore the focus to the previously focused control in the event of a DDX or DDV failure.

Implementors of custom DDX or DDV routines should call **PrepareCtrl** for all non-edit controls for which they are exchanging data via DDX or validating data via DDV.

For more information on writing your own DDX and DDV routines, see Technical Note 26. For an overview of DDX and DDV, see Dialog Data Exchange and Validation and Dialog Box Topics.

See Also

Reference

CDataExchange Class
Hierarchy Chart
CDataExchange::Fail

Other Resources

CDataExchange Members

CDataExchange::PrepareEditCtrl

The framework calls this member function to prepare the specified edit control for dialog data exchange (DDX) and validation (DDV).

```
HWND PrepareEditCtrl(
   int nIDC
);
```

Parameters

nIDC

The ID of the edit control to be prepared for DDX or DDV.

Return Value

The **HWND** of the edit control being prepared for DDX or DDV.

Remarks

Use PrepareCtrl instead for all non-edit controls.

Preparation consists of two things. First, **PrepareEditCtrl** stores the control's **HWND** in the **CDataExchange** class. The framework uses this handle to restore the focus to the previously focused control in the event of a DDX or DDV failure. Second, **PrepareEditCtrl** sets a flag in the **CDataExchange** class to indicate that the control whose data is being exchanged or validated is an edit control.

Implementors of custom DDX or DDV routines should call **PrepareEditCtrl** for all edit controls for which they are exchanging data via DDX or validating data via DDV.

For more information on writing your own DDX and DDV routines, see Technical Note 26. For an overview of DDX and DDV, see Dialog Data Exchange and Validation and Dialog Box Topics.

See Also

Reference

CDataExchange Class
Hierarchy Chart
CDataExchange::Fail

Other Resources

CDataExchange Members

CDataExchange::PrepareOleCtrl

The framework calls this member function to prepare the specified OLE control for dialog data exchange (DDX) and validation (DDV).

```
COleControlSite* PrepareOleCtrl(
   int nIDC
);
```

Parameters

nIDC

The ID of the OLE control to be prepared for DDX or DDV.

Return Value

A pointer to the OLE control site.

Remarks

Use PrepareEditCtrl instead for edit controls or PrepareCtrl for all other non-OLE controls.

Implementors of custom DDX or DDV routines should call **PrepareOleCtrl** for all OLE controls for which they are exchanging data via DDX or validating data via DDV.

For more information on writing your own DDX and DDV routines, see Technical Note 26. For an overview of DDX and DDV, see Dialog Data Exchange and Validation and Dialog Box Topics.

See Also

Reference

CDataExchange Class

Hierarchy Chart

Other Resources

CDataExchange Members

CDataExchange Data Members

For information about the data members in **CDataExchange**, see CDataExchange Members.

CDataExchange::m_bSaveAndValidate

This flag indicates the direction of a dialog data exchange (DDX) operation.

```
BOOL m_bSaveAndValidate;
```

Remarks

The flag is nonzero if the **CDataExchange** object is being used to move data from the dialog controls to dialog-class data members after the user edits the controls. The flag is zero if the object is being used to initialize dialog controls from dialog-class data members.

The flag is also nonzero during dialog data validation (DDV).

For more information on writing your own DDX and DDV routines, see Technical Note 26. For an overview of DDX and DDV, see Dialog Data Exchange and Validation and Dialog Box Topics.

See Also

Reference

CDataExchange Class

Hierarchy Chart

Other Resources

CDataExchange Members

CDataExchange::m_pDlgWnd

Contains a pointer to the CWnd object for which dialog data exchange (DDX) or validation (DDV) is taking place.

```
CWnd* m_pDlgWnd;
```

Remarks

This object is usually a CDialog object. Implementors of custom DDX or DDV routines can use this pointer to obtain access to the dialog window that contains the controls they are operating on.

For more information on writing your own DDX and DDV routines, see Technical Note 26. For an overview of DDX and DDV, see Dialog Data Exchange and Validation and Dialog Box Topics.

See Also

Reference

CDataExchange Class

Hierarchy Chart

Other Resources

CDataExchange Members

CDataPathProperty Class

Implements an OLE control property that can be loaded asynchronously.

```
class CDataPathProperty : public CAsyncMonikerFile
```

Remarks

Asynchronous properties are loaded after synchronous initiation.

The class **CDataPathProperty** is derived from **CAysncMonikerFile**. To implement asynchronous properties in your OLE controls, derive a class from **CDataPathProperty**, and override OnDataAvailable.

For more information about how to use asynchronous monikers and ActiveX controls in Internet applications, see the following articles:

- Internet First Steps: ActiveX Controls
- Internet First Steps: Asynchronous Monikers

Requirements

Header: afxctl.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

IMAGE Sample: Creates an ActiveX Control That Downloads Asynchronously

Reference

CAsyncMonikerFile Class

Hierarchy Chart

CAsyncMonikerFile Class

Other Resources

CDataPathProperty Members

CDataPathProperty Members

Base Class Members

CObject Members

CFile Members

COleStreamFile Members

CMonikerFile Members

CAsyncMonikerFile Members

Construction

CDataPathProperty	Constructs a **CDataPathProperty** object.

Operations

GetControl	Retrieves the asynchronous OLE control associated with the **CDataPathProperty** object.
GetPath	Retrieves the pathname of the property.
Open	Initiates loading of the asynchronous property for the associated ActiveX (OLE) control.
ResetData	Calls **CAsyncMonikerFile::OnDataAvailable** to notify the container that the control properties have changed.
SetControl	Sets the asynchronous ActiveX (OLE) control associated with the property.
SetPath	Sets the pathname of the property.

See Also

Reference

CDataPathProperty Class

Hierarchy Chart

CDataPathProperty Member Functions

For information about the member functions in **CDataPathProperty**, see CDataPathProperty Members.

CDataPathProperty::CDataPathProperty

Constructs a **CDataPathProperty** object.

```
CDataPathProperty(
   COleControl* pControl = NULL
);
CDataPathProperty(
   LPCTSTR lpszPath,
      COleControl* pControl = NULL
);
```

Parameters

pControl

A pointer to the OLE control object to be associated with this **CDataPathProperty** object.

lpszPath

The path, which may be absolute or relative, used to create an asynchronous moniker that references the actual absolute location of the property. **CDataPathProperty** uses URLs, not filenames. If you want a **CDataPathProperty** object for a file, prepend `file://` to the path.

Remarks

The **COleControl** object pointed to by *pControl* is used by **Open** and retrieved by derived classes. If *pControl* is **NULL**, the control used with **Open** should be set with **SetControl**. If *lpszPath* is **NULL**, you can pass in the path through **Open** or set it with **SetPath**.

See Also

Reference

CDataPathProperty Class
Hierarchy Chart
CDataPathProperty::Open
CDataPathProperty::SetControl

Other Resources

CDataPathProperty Members

CDataPathProperty::GetControl

Call this member function to retrieve the **COleControl** object associated with the **CDataPathProperty** object.

```
COleControl* GetControl( );
```

Return Value

Returns a pointer to the OLE control associated with the **CDataPathProperty** object. **NULL** if not control is associated.

See Also

Reference

CDataPathProperty Class
Hierarchy Chart
CDataPathProperty::SetControl

Other Resources

CDataPathProperty Members

CDataPathProperty::GetPath

Call this member function to retrieve the path, set when the **CDataPathProperty** object was constructed, or specified in **Open**, or specified in a previous call to the **SetPath** member function.

```
CString GetPath( ) const;
```

Return Value

Returns the pathname to the property itself. Can be empty if no path has been specified.

See Also

Reference

CDataPathProperty Class
Hierarchy Chart
CDataPathProperty::SetPath
CDataPathProperty::Open
CDataPathProperty::CDataPathProperty

Other Resources

CDataPathProperty Members

CDataPathProperty::Open

Call this member function to initiate loading of the asynchronous property for the associated control.

```
virtual BOOL Open(
   COleControl* pControl,
   CFileException* pError = NULL
);
virtual BOOL Open(
   LPCTSTR lpszPath,
   COleControl* pControl,
      CFileException* pError = NULL
);
virtual BOOL Open(
   LPCTSTR lpszPath,
   CFileException* pError = NULL
);
virtual BOOL Open(
   CFileException* pError = NULL
);
```

Parameters

pControl

A pointer to the OLE control object to be associated with this **CDataPathProperty** object.

pError

A pointer to a file exception. In the event of an error, will be set to the cause.

lpszPath

The path, which may be absolute or relative, used to create an asynchronous moniker that references the actual absolute location of the property. **CDataPathProperty** uses URLs, not filenames. If you want a **CDataPathProperty** object for a file, prepend `file://` to the path.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The function attempts to obtain the **IBindHost** interface from the control.

Before calling **Open** without a path, the value for the property's path must be set. This can be done when the object is constructed, or by calling the **SetPath** member function.

Before calling **Open** without a control, an ActiveX control (formerly known as an OLE control) can be associated with the object. This can be done when the object is constructed, or by calling **SetControl**.

All overloads of CAsyncMonikerFile::Open are also available from **CDataPathProperty**.

See Also

Reference

CDataPathProperty Class
Hierarchy Chart
CDataPathProperty::SetControl
CDataPathProperty::CDataPathProperty
CAsyncMonikerFile::Open

Other Resources

CDataPathProperty Members

CDataPathProperty::ResetData

Call this function to get **CAsyncMonikerFile::OnDataAvailable** to notify the container that the control properties have changed, and all the information loaded asynchronously is no longer useful.

```
virtual void ResetData( );
```

Remarks

Opening should be restarted. Derived classes can override this function for different defaults.

See Also

Reference

CDataPathProperty Class
Hierarchy Chart
CAsyncMonikerFile::OnDataAvailable
CDataPathProperty::Open

Other Resources

CDataPathProperty Members

CDataPathProperty::SetControl

Call this member function to associate an asynchronous OLE control with the **CDataPathProperty** object.

```
void SetControl(
   COleControl* pControl
);
```

Parameters

pControl

A pointer to the asynchronous OLE control to be associated with the property.

See Also

Reference

CDataPathProperty Class
Hierarchy Chart
CDataPathProperty::GetControl
CDataPathProperty::SetPath
CDataPathProperty::CDataPathProperty

Other Resources

CDataPathProperty Members

CDataPathProperty::SetPath

Call this member function to set the pathname of the property.

```
void SetPath(
   LPCTSTR lpszPath
);
```

Parameters

lpszPath

A path, which may be absolute or relative, to the property being loaded asynchronously. **CDataPathProperty** uses URLs, not filenames. If you want a **CDataPathProperty** object for a file, prepend `file://` to the path.

See Also

Reference

CDataPathProperty Class
Hierarchy Chart
CDataPathProperty::GetPath
CDataPathProperty::SetControl
CDataPathProperty::CDataPathProperty

Other Resources

CDataPathProperty Members

CDateTimeCtrl Class

Encapsulates the functionality of a date and time picker control.

```
class CDateTimeCtrl : public CWnd
```

Remarks

The date and time picker control (DTP control) provides a simple interface to exchange date and time information with a user. This interface contains fields, each of which displays a part of the date and time information stored in the control. The user can change the information stored in the control by changing the content of the string in a given field. The user can move from field to field using the mouse or the keyboard.

You can customize the date and time picker control by applying a variety of styles to the object when you create it. See Date and Time Picker Control Styles in the *Platform SDK* for more information about styles specific to the date and time picker control. You can set the display format of the DTP control using format styles. These format styles are described under "Format Styles" in the *Platform SDK* topic Date and Time Picker Control Styles.

The date and time picker control also uses notifications and callbacks, which are described in Using CDateTimeCtrl.

Requirements

Header: afxdtctl.h

See Also

Tasks

CMNCTRL1 Sample: Demonstrates Common Control MFC Classes, Part 1

Reference

CWnd Class

Hierarchy Chart

CMonthCalCtrl Class

Other Resources

CDateTimeCtrl Members

CDateTimeCtrl Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Construction

CDateTimeCtrl	Constructs a **CDateTimeCtrl** object.
Create	Creates the date and time picker control and attaches it to the **CDateTimeCtrl** object.

Attributes

GetMonthCalColor	Retrieves the color for a given portion of the month calendar within the date and time picker control.
GetMonthCalCtrl	Retrieves the date and time picker's child month calendar control.
GetMonthCalFont	Retrieves the font currently used by the date and time picker control's child month calendar control.
GetRange	Retrieves the current minimum and maximum allowable system times for a date and time picker control.
SetFormat	Sets the display of a date and time picker control based on a given format string.
SetMonthCalColor	Sets the color for a given portion of the month calendar within a date and time picker control.
SetMonthCalFont	Sets the font to be used by the date and time picker control's child month calendar control.
SetRange	Sets the minimum and maximum allowable system times for a date and time picker control.

Operations

GetTime	Retrieves the currently selected time from a date and time picker control and places it in a specified **SYSTEMTIME** structure.
SetTime	Sets the time in a date and time picker control.

See Also

Reference

CDateTimeCtrl Class

Hierarchy Chart

CDateTimeCtrl Member Functions

For information about the member functions in **CDateTimeCtrl**, see CDateTimeCtrl Members.

CDateTimeCtrl::CDateTimeCtrl

Constructs a **CDateTimeCtrl** object.

```
CDateTimeCtrl( );
```

See Also

Reference

CDateTimeCtrl Class

Hierarchy Chart

CDateTimeCtrl::Create

Other Resources

CDateTimeCtrl Members

CDateTimeCtrl::Create

Creates the date and time picker control and attaches it to the **CDateTimeCtrl** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the combination of date time control styles. See Date and Time Picker Control Styles in the *Platform SDK* for more information about date and time picker styles.

rect

A reference to a RECT structure, which is the position and size of the date and time picker control.

pParentWnd

A pointer to a CWnd object that is the parent window of the date and time picker control. It must not be **NULL**.

nID

Specifies the date and time picker control's control ID.

Return Value

Nonzero if creation was successful; otherwise 0.

Remarks

To create a date and time picker control

1. Call CDateTimeCtrl to construct a **CDateTimeCtrl** object.
2. Call this member function, which creates the Windows date and time picker control and attaches it to the **CDateTimeCtrl** object.

When you call **Create**, the common controls are initialized.

Example

```
void CDatesDlg::OnButton2()
{
   // assuming CDatesDlg has a CDateTimeCtrl* member named m_pCtrl...
   m_pCtrl = new CDateTimeCtrl();

   // choose an arbitrary rectangle for creation
   CRect rect(20, 20, 120, 45);
   pCtrl->Create(WS_VISIBLE | WS_CHILD | WS_TABSTOP | DTS_SHOWNONE | DTS_SHORTDATEFORMAT,
      rect, this, 1006);
}
```

See Also

Reference

CDateTimeCtrl Class

Hierarchy Chart

Other Resources

CDateTimeCtrl Members

CDateTimeCtrl::GetMonthCalColor

Retrieves the color for a given portion of the month calendar within the date and time picker control.

```
COLORREF GetMonthCalColor(
   int iColor
) const;
```

Parameters

iColor

An **int** value specifying which color area of the month calendar to retrieve. For a list of values, see the *iColor* parameter for SetMonthCalColor.

Return Value

A **COLORREF** value that represents the color setting for the specified portion of the month calendar control if successful. The function returns -1 if unsuccessful.

Remarks

This member function implements the behavior of the Win32 message DTM_GETMCCOLOR, as described in the *Platform SDK*.

Example

```
void CDatesDlg::OnButton2()
{
   // Gain a pointer to the control
   CDateTimeCtrl* pCtrl = (CDateTimeCtrl*) GetDlgItem(IDC_DATETIMEPICKER1);
   ASSERT(pCtrl != NULL);

   // Set the color for the text in the control and
   // assure it was set properly. Unlike the GetMonthCalCtrl() member,
   // GetMonthCalColor() and SetMonthCalColor() can be used at any time.
   pCtrl->SetMonthCalColor(MCSC_TEXT, RGB(255, 0, 0));
   VERIFY(pCtrl->GetMonthCalColor(MCSC_TEXT) == RGB(255, 0, 0));
}
```

See Also

Reference

CDateTimeCtrl Class
Hierarchy Chart
CDateTimeCtrl::SetMonthCalColor

Other Resources

CDateTimeCtrl Members

CDateTimeCtrl::GetMonthCalCtrl

Retrieves the **CMonthCalCtrl** object associated with the date and time picker control.

```
CMonthCalCtrl* GetMonthCalCtrl( ) const;
```

Return Value

A pointer to a CMonthCalCtrl object, or **NULL** if unsuccessful or if the window is not visible.

Remarks

Date and time picker controls create a child month calendar control when the user clicks the drop-down arrow. When the **CMonthCalCtrl** object is no longer needed, it is destroyed, so your application must not rely on storing the object representing the date time picker control's child month calendar.

Example

```
void CDatesDlg::OnDropDownDateTimePicker1(NMHDR* pNMHDR, LRESULT* pResult)
{
   // note that GetMonthCalCtrl() will only return a pointer to the
   // month calendar control while the control actually exists--that is,
   // while it has been dropped-down by the user. Otherwise, the function
   // returns NULL. One appropriate time to get the control is while
   // handling the DTN_DROPDOWN notification for the date time picker
   // control.

   CDateTimeCtrl* pCtrl = (CDateTimeCtrl*) GetDlgItem(IDC_DATETIMEPICKER1);
   ASSERT(pCtrl != NULL);

   // get the control
   CMonthCalCtrl* pMoCalCtrl = pCtrl->GetMonthCalCtrl();
   ASSERT(pMoCalCtrl != NULL);

   // now, pMoCalCtrl is useful...

   *pResult = 0;
}
```

See Also

Reference

CDateTimeCtrl Class

Hierarchy Chart

Other Resources

CDateTimeCtrl Members

CDateTimeCtrl::GetMonthCalFont

Gets the font currently used by the date and time picker control's month calendar control.

```
CFont* GetMonthCalFont( ) const;
```

Return Value

A pointer to a CFont object, or **NULL** if unsuccessful.

Remarks

The **CFont** object pointed to by the return value is a temporary object and is destroyed during the next idle processing time.

See Also

Reference

CDateTimeCtrl Class
Hierarchy Chart
CDateTimeCtrl::SetMonthCalFont

Other Resources

CDateTimeCtrl Members

CDateTimeCtrl::GetRange

Retrieves the current minimum and maximum allowable system times for a date and time picker control.

```
DWORD GetRange(
   COleDateTime* pMinRange,
   COleDateTime* pMaxRange
) const;
DWORD GetRange(
   CTime* pMinRange,
   CTime* pMaxRange
) const;
```

Parameters

pMinRange

A pointer to a COleDateTime object or a CTime object containing the earliest time allowed in the **CDateTimeCtrl** object.

pMaxRange

A pointer to a **COleDateTime** object or a **CTime** object containing the latest time allowed in the **CDateTimeCtrl** object.

Return Value

A **DWORD** value containing flags that indicate which ranges are set. If

`return value & GDTR_MAX` == 0

then the second parameter is valid. Similarly, if

`return value & GDTR_MIN` == 0

then the first parameter is valid.

Remarks

This member function implements the behavior of the Win32 message DTM_GETRANGE, as described in the *Platform SDK*. In MFC's implementation, you can specify either **COleDateTime** or **CTime** usages.

Example

```
// This function will set several ranges in the control, then
// call the ShowRange() function to show the set ranges to the
// user.

void CDatesDlg::OnButton2()
{
   // Gain a pointer to the control
   // CMonthCalCtrl* pCtrl = (CMonthCalCtrl*)
   // GetDlgItem(IDC_MONTHCALENDAR1);
   CDateTimeCtrl* pCtrl = (CDateTimeCtrl*) GetDlgItem(IDC_DATETIMEPICKER1);
   ASSERT(pCtrl != NULL);

   // Set minimum of January 1st, 1995 with no maximum.
   COleDateTime dtMin;
   COleDateTime dtMax;

   dtMin = COleDateTime(1995, 1, 1, 0, 0, 0);
   dtMax.SetStatus(COleDateTime::null);
   pCtrl->SetRange(&dtMin, &dtMax);
   ShowRange(pCtrl);

   // Set no minimum and maximum of September 30th, 1997.
   dtMin.SetStatus(COleDateTime::null);
   dtMax = COleDateTime(1997, 9, 30, 0, 0, 0);
   pCtrl->SetRange(&dtMin, &dtMax);
   ShowRange(pCtrl);
```

```
    // Set minimum of April 15, 1992 and maximum of June 5, 2002.
    dtMin = COleDateTime(1992, 4, 15, 0, 0, 0);
    dtMax = COleDateTime(2002, 6, 5, 0, 0, 0);
    pCtrl->SetRange(&dtMin, &dtMax);
    ShowRange(pCtrl);
}

void CDatesDlg::ShowRange(CDateTimeCtrl* pCtrl)
{
    ASSERT(pCtrl != NULL);
    CString strMessage;
    COleDateTime dtMinimum;
    COleDateTime dtMaximum;

    // Get the range.
    DWORD dwResult = pCtrl->GetRange(&dtMinimum, &dtMaximum);

    // If a minimum was specified, format it.
    // Otherwise, indicate that there is no lower bound.
    if (dwResult & GDTR_MIN)
        strMessage += dtMinimum.Format(_T("Minimum range is %x %X.\r\n"));
    else
        strMessage += _T("No minimum range.\r\n");

    // Treat maximum similarly.
    if (dwResult & GDTR_MAX)
        strMessage += dtMaximum.Format(_T("Maximum range is %x %X.\r\n"));
    else
        strMessage += _T("No maximum range.\r\n");

    // Show the user.
    AfxMessageBox(strMessage);
}
```

See Also

Reference

CDateTimeCtrl Class

Hierarchy Chart

CDateTimeCtrl::SetRange

Other Resources

CDateTimeCtrl Members

CDateTimeCtrl::GetTime

Retrieves the currently selected time from a date and time picker control and places it in a specified **SYSTEMTIME** structure.

```
BOOL GetTime(
   COleDateTime& timeDest
) const;
DWORD GetTime(
   CTime& timeDest
) const;
DWORD GetTime(
   LPSYSTEMTIME pTimeDest
) const;
```

Parameters

timeDest

In the first version, a reference to a COleDateTime object that will receive the system time information. In the second version, a reference to a CTime object that will receive the system time information.

pTimeDest

A pointer to the SYSTEMTIME structure to receive the system time information. Must not be **NULL**.

Return Value

In the first version, nonzero if the time is successfully written to the **COleDateTime** object; otherwise 0. In the second and third versions, a **DWORD** value equal to the **dwFlag** member set in the NMDATETIMECHANGE structure. See the **Remarks** section below for more information.

Remarks

This member function implements the behavior of the Win32 message DTM_GETSYSTEMTIME, as described in the *Platform SDK*. In the MFC implementation of **GetTime**, you can use **COleDateTime** or **CTime** classes, or you can use a **SYSTEMTIME** structure, to store the time information.

The return value **DWORD** in the second and third versions, above, indicates whether or not the date and time picker control is set to the "no date" status, as indicated in the NMDATETIMECHANGE structure member **dwFlags**. If the value returned equals **GDT_NONE**, the control is set to "no date" status, and uses the **DTS_SHOWNONE** style. If the value returned equals **GDT_VALID**, the system time is successfully stored in the destination location.

Example

```
void CDatesDlg::OnButton2()
{
   // Gain a pointer to the control
   CDateTimeCtrl* pCtrl = (CDateTimeCtrl*) GetDlgItem(IDC_DATETIMEPICKER1);
   ASSERT(pCtrl != NULL);

   // get as a CTime
   CTime timeTime;
   DWORD dwResult = pCtrl->GetTime(timeTime);
   if (dwResult == GDT_VALID)
   {
      // the user checked the box and specified data
      CString str;

      // is it a time-only control, or a date-only control?
      if ((pCtrl->GetStyle() & DTS_TIMEFORMAT) == DTS_TIMEFORMAT)
         str = timeTime.Format(_T("%X"));
      else
         str = timeTime.Format(_T("%x"));
      AfxMessageBox(str);
   }
   else
```

```
    {
       // the user unmarked the "none" box
       AfxMessageBox(_T("Time not set!"));
    }

    // Calling as SYSTIME is much the same, but calling for a COleDateTime
    // has us test the state of the COleDateTime object for validity to
    // see if the user did or didn't check the "none" box.
 }
```

See Also

Reference

CDateTimeCtrl Class

Hierarchy Chart

CDateTimeCtrl::SetTime

Other Resources

CDateTimeCtrl Members

CDateTimeCtrl::SetFormat

Sets the display of a date and time picker control based on a given format string.

```
BOOL SetFormat(
   LPCTSTR pstrFormat
);
```

Parameters

pstrFormat

A pointer to a zero-terminated format string that defines the desired display. Setting this parameter to **NULL** will reset the control to the default format string for the current style.

Return Value

Nonzero if successful; otherwise 0.

Note
User input does not determine success or failure for this call.

Remarks

This member function implements the behavior of the Win32 message DTM_SETFORMAT, as described in the *Platform SDK*.

Example

```
void CDatesDlg::OnButton2()
{
   // Gain a pointer to the control
   CDateTimeCtrl* pCtrl = (CDateTimeCtrl*) GetDlgItem(IDC_DATETIMEPICKER1);
   ASSERT(pCtrl != NULL);

   // The control will create itself with a format that matches the
   // locale setting in Control Panel. But we can force a particular
   // format with a call to SetFormat(). This call forces the format
   // dd-MMM-yy, which would show 03-APR-98 for April 3rd, 1998.

   pCtrl->SetFormat(_T("dd-MMM-yy"));
}
```

See Also

Reference

CDateTimeCtrl Class

Hierarchy Chart

Other Resources

CDateTimeCtrl Members

CDateTimeCtrl::SetMonthCalColor

Sets the color for a given portion of the month calendar within a date and time picker control.

```
COLORREF SetMonthCalColor(
   int iColor,
   COLORREF ref
);
```

Parameters

iColor

int value specifying which area of the month calendar control to set. This value can be one of the following.

Value	Meaning
MCSC_BACKGROUND	Set the background color displayed between months.
MCSC_MONTHBK	Set the background color displayed within a month.
MCSC_TEXT	Set the color used to display text within a month.
MCSC_TITLEBK	Set the background color displayed in the calendar's title.
MCSC_TITLETEXT	Set the color used to display text within the calendar's title.
MCSC_TRAILINGTEXT	Set the color used to display header and trailing-day text. Header and trailing days are the days from the previous and following months that appear on the current calendar.

ref

A **COLORREF** value representing the color that will be set for the specified area of the month calendar.

Return Value

A **COLORREF** value that represents the previous color setting for the specified portion of the month calendar control if successful. Otherwise, the message returns -1.

Remarks

This member function implements the behavior of the Win32 message DTM_SETMCCOLOR, as described in the *Platform SDK*.

Example

See the example for CDateTimeCtrl::GetMonthCalColor.

See Also

Reference

CDateTimeCtrl Class
Hierarchy Chart
CDateTimeCtrl::GetMonthCalColor

Other Resources

CDateTimeCtrl Members

CDateTimeCtrl::SetMonthCalFont

Sets the font to be used by the date and time picker control's child month calendar control.

```
void SetMonthCalFont(
   HFONT hFont,
   BOOL bRedraw = TRUE
);
```

Parameters

hFont

Handle to the font that will be set.

bRedraw

Specifies whether the control should be redrawn immediately upon setting the font. Setting this parameter to **TRUE** causes the control to redraw itself.

Remarks

This member function implements the behavior of the Win32 message DTM_SETMCFONT, as described in the *Platform SDK*.

Example

```
// The following code example creates a font
// (Arial, 10 pixels high) and if successful,
// stores the result in m_pMonthFont. SetMonthCalFont
// is then called passing in the new font, causing
// the month calendar control to display all
// text and dates with an Arial font.

BOOL CYourDialog::OnInitDialog()
{
   CDialog::OnInitDialog();

   // ... other code here ...

//initializing the necessary members of the LOGFONT
// structure

   m_pMonthFont = new CFont;
   LOGFONT lf;   memset(&lf, 0, sizeof(lf));
   lf.lfHeight = 10;
   strcpy(lf.lfFaceName, "Arial");

   if (m_pMonthFont->CreateFontIndirect(&lf))
   {
// if successful, grab the month calendar control from
// our dialog and set the font

      CMonthCalCtrl* pCtrl = (CMonthCalCtrl*) GetDlgItem(IDC_DATETIME1);
      ASSERT(pCtrl != NULL);
      pCtrl->SetMonthCalFont(m_pMonthFont);
   }
   else
   {

// if not successful, clean up the font pointer and
// set equal to NULL

      delete m_pMonthFont;
      m_pMonthFont = NULL;
   }
```

```
    // ... other code here ...
}

CYourDialog::CYourDialog(CWnd* pParent /*=NULL*/)
    : CDialog(CYourDialog::IDD, pParent)
{
    // ... other code here ...
    m_pMonthFont = NULL;
    // ... other code here ...
}

CYourDialog::~CYourDialog()
{
    // ... other code here ...
    delete m_pMonthFont;
    // ... other code here ...
}
```

Note

If you use this code, you'll want to make a member of your **CDialog**-derived class called `m_pMonthFont` of type **CFont***.

See Also

Reference

CDateTimeCtrl Class

Hierarchy Chart

CDateTimeCtrl::GetMonthCalFont

Other Resources

CDateTimeCtrl Members

CDateTimeCtrl::SetRange

Sets the minimum and maximum allowable system times for a date and time picker control.

```
BOOL SetRange(
   const COleDateTime* pMinRange,
   const COleDateTime* pMaxRange
);
BOOL SetRange(
   const CTime* pMinRange,
   const CTime* pMaxRange
);
```

Parameters

pMinRange

A pointer to a COleDateTime object or a CTime object containing the earliest time allowed in the **CDateTimeCtrl** object.

pMaxRange

A pointer to a **COleDateTime** object or a **CTime** object containing the latest time allowed in the **CDateTimeCtrl** object.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function implements the behavior of the Win32 message DTM_SETRANGE, as described in the *Platform SDK*. In MFC's implementation, you can specify either **COleDateTime** or **CTime** usages. If the **COleDateTime** object has a **NULL** status, the range will be removed. If the **CTime** pointer or the **COleDateTime** pointer is **NULL**, the range will be removed.

Example

See the example for CDateTimeCtrl::GetRange.

See Also

Reference

CDateTimeCtrl Class
Hierarchy Chart
CDateTimeCtrl::GetRange

Other Resources

CDateTimeCtrl Members

CDateTimeCtrl::SetTime

Sets the time in a date and time picker control.

```
BOOL SetTime(
   const COleDateTime& timeNew
);
BOOL SetTime(
   const CTime* pTimeNew
);
BOOL SetTime(
   LPSYSTEMTIME pTimeNew = NULL
);
```

Parameters

timeNew

A reference to a COleDateTime object containing the to which the control will be set.

pTimeNew

In the second version above, a pointer to a CTime object containing the time to which the control will be set. In the third version above, a pointer to a SYSTEMTIME structure containing the time to which the control will be set.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function implements the behavior of the Win32 message DTM_SETSYSTEMTIME, as described in the *Platform SDK*. In the MFC implementation of **SetTime**, you can use the **COleDateTime** or **CTime** classes, or you can use a **SYSTEMTIME** structure, to set the time information.

Example

```
void CDatesDlg::OnButton2()
{
   // Gain a pointer to the control
   // CMonthCalCtrl* pCtrl = (CMonthCalCtrl*) GetDlgItem(IDC_MONTHCALENDAR1);
   CDateTimeCtrl* pCtrl = (CDateTimeCtrl*) GetDlgItem(IDC_DATETIMEPICKER1);
   ASSERT(pCtrl != NULL);

   // set with a CTime
   CTime timeTime(1998, 4, 3, 0, 0, 0);
   VERIFY(pCtrl->SetTime(&timeTime));

   // set with a COleDateTime object
   COleDateTime oletimeTime(1998, 4, 3, 0, 0, 0);
   VERIFY(pCtrl->SetTime(oletimeTime));

   // set using the SYSTEMTIME
   SYSTEMTIME sysTime;
   memset(&sysTime, 0, sizeof(sysTime));
   sysTime.wYear = 1998;
   sysTime.wMonth = 4;
   sysTime.wDay = 3;
   VERIFY(pCtrl->SetTime(&sysTime));
}
```

See Also

Reference

CDateTimeCtrl Class

Hierarchy Chart

CDateTimeCtrl::GetTime

Other Resources

CDateTimeCtrl Members

CDBException Class

Represents an exception condition arising from the database classes.

```
class CDBException : public CException
```

Remarks

The class includes two public data members you can use to determine the cause of the exception or to display a text message describing the exception. **CDBException** objects are constructed and thrown by member functions of the database classes.

Note

This class is one of MFC's Open Database Connectivity (ODBC) classes. If you are instead using the newer Data Access Objects (DAO) classes, use CDaoException instead. All DAO class names have "CDao" as a prefix. For more information, see the article Overview: Database Programming.

Exceptions are cases of abnormal execution involving conditions outside the program's control, such as data source or network I/O errors. Errors that you might expect to see in the normal course of executing your program are usually not considered exceptions.

You can access these objects within the scope of a **CATCH** expression. You can also throw **CDBException** objects from your own code with the **AfxThrowDBException** global function.

For more information about exception handling in general, or about **CDBException** objects, see the articles Exception Handling (MFC) and Exceptions: Database Exceptions.

Requirements

Header: afxdb.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CException Class
Hierarchy Chart
CDatabase Class
CRecordset Class
CFieldExchange Class
AfxThrowDBException
CRecordset::Update
CRecordset::Delete
CException Class

Other Resources

CDBException Members

CDBException Members

Base Class Members

CObject Members

CException Members

Data Members

m_nRetCode	Contains an Open Database Connectivity (ODBC) return code, of type **RETCODE**.
m_strError	Contains a string that describes the error in alphanumeric terms.
m_strStateNativeOrigin	Contains a string describing the error in terms of the error codes returned by ODBC.

See Also

Reference

CDBException Class

Hierarchy Chart

CDBException Data Members

For information about the data members in **CDBException**, see CDBException Members.

CDBException::m_nRetCode

Contains an ODBC error code of type **RETCODE** returned by an ODBC application programming interface (API) function.

Remarks

This type includes SQL-prefixed codes defined by ODBC and AFX_SQL-prefixed codes defined by the database classes. For a **CDBException**, this member will contain one of the following values:

- **AFX_SQL_ERROR_API_CONFORMANCE** The driver for a **CDatabase::OpenEx** or **CDatabase::Open** call does not conform to required ODBC API Conformance level 1 (**SQL_OAC_LEVEL1**).
- **AFX_SQL_ERROR_CONNECT_FAIL** Connection to the data source failed. You passed a **NULL CDatabase** pointer to your recordset constructor and the subsequent attempt to create a connection based on **GetDefaultConnect** failed.
- **AFX_SQL_ERROR_DATA_TRUNCATED** You requested more data than you have provided storage for. For information on increasing the provided data storage for **CString** or **CByteArray** data types, see the *nMaxLength* argument for RFX_Text and RFX_Binary under "Macros and Globals."
- **AFX_SQL_ERROR_DYNASET_NOT_SUPPORTED** A call to **CRecordset::Open** requesting a dynaset failed. Dynasets are not supported by the driver.
- **AFX_SQL_ERROR_EMPTY_COLUMN_LIST** You attempted to open a table (or what you gave could not be identified as a procedure call or **SELECT** statement) but there are no columns identified in record field exchange (RFX) function calls in your **DoFieldExchange** override.
- **AFX_SQL_ERROR_FIELD_SCHEMA_MISMATCH** The type of an RFX function in your **DoFieldExchange** override is not compatible with the column data type in the recordset.
- **AFX_SQL_ERROR_ILLEGAL_MODE** You called **CRecordset::Update** without previously calling **CRecordset::AddNew** or **CRecordset::Edit**.
- **AFX_SQL_ERROR_LOCK_MODE_NOT_SUPPORTED** Your request to lock records for update could not be fulfilled because your ODBC driver does not support locking.
- **AFX_SQL_ERROR_MULTIPLE_ROWS_AFFECTED** You called **CRecordset::Update** or **Delete** for a table with no unique key and changed multiple records.
- **AFX_SQL_ERROR_NO_CURRENT_RECORD** You attempted to edit or delete a previously deleted record. You must scroll to a new current record after a deletion.
- **AFX_SQL_ERROR_NO_POSITIONED_UPDATES** Your request for a dynaset could not be fulfilled because your ODBC driver does not support positioned updates.
- **AFX_SQL_ERROR_NO_ROWS_AFFECTED** You called **CRecordset::Update** or **Delete**, but when the operation began the record could no longer be found.
- **AFX_SQL_ERROR_ODBC_LOAD_FAILED** An attempt to load the ODBC.DLL failed; Windows could not find or could not load this DLL. This error is fatal.
- **AFX_SQL_ERROR_ODBC_V2_REQUIRED** Your request for a dynaset could not be fulfilled because a Level 2-compliant ODBC driver is required.
- **AFX_SQL_ERROR_RECORDSET_FORWARD_ONLY** An attempt to scroll did not succeed because the data source does not support backward scrolling.
- **AFX_SQL_ERROR_SNAPSHOT_NOT_SUPPORTED** A call to **CRecordset::Open** requesting a snapshot failed. Snapshots are not supported by the driver. (This should only occur when the ODBC cursor library — ODBCCURS.DLL — is not present.)
- **AFX_SQL_ERROR_SQL_CONFORMANCE** The driver for a **CDatabase::OpenEx** or **CDatabase::Open** call does not conform to the required ODBC SQL Conformance level of "Minimum" (**SQL_OSC_MINIMUM**).
- **AFX_SQL_ERROR_SQL_NO_TOTAL** The ODBC driver was unable to specify the total size of a **CLongBinary** data value. The operation probably failed because a global memory block could not be preallocated.
- **AFX_SQL_ERROR_RECORDSET_READONLY** You attempted to update a read-only recordset, or the data source is read-

only. No update operations can be performed with the recordset or the **CDatabase** object it is associated with.

- **SQL_ERROR** Function failed. The error message returned by the ODBC function **SQLError** is stored in the **m_strError** data member.
- **SQL_INVALID_HANDLE** Function failed due to an invalid environment handle, connection handle, or statement handle. This indicates a programming error. No additional information is available from the ODBC function **SQLError**.

The SQL-prefixed codes are defined by ODBC. The AFX-prefixed codes are defined in AFXDB.H, found in MFC\INCLUDE.

See Also

Reference

CDBException Class
Hierarchy Chart
CDatabase Class
CLongBinary Class
CRecordset Class

Other Resources

CDBException Members

CDBException::m_strError

Contains a string describing the error that caused the exception.

Remarks

The string describes the error in alphanumeric terms. For more detailed information and an example, see **m_strStateNativeOrigin**.

See Also

Reference

CDBException Class

Hierarchy Chart

CDBException::m_strStateNativeOrigin

Other Resources

CDBException Members

CDBException::m_strStateNativeOrigin

Contains a string describing the error that caused the exception.

Remarks

The string is of the form "State:%s,Native:%ld,Origin:%s", where the format codes, in order, are replaced by values that describe:

- The **SQLSTATE**, a null-terminated string containing a five-character error code returned in the *szSqlState* parameter of the ODBC function **SQLError**. **SQLSTATE** values are listed in Appendix A, ODBC Error Codes, in the *ODBC Programmer's Reference*. Example: "S0022".
- The native error code, specific to the data source, returned in the *pfNativeError* parameter of the **SQLError** function. Example: 207.
- The error message text returned in the *szErrorMsg* parameter of the **SQLError** function. This message consists of several bracketed names. As an error is passed from its source to the user, each ODBC component (data source, driver, Driver Manager) appends its own name. This information helps to pinpoint the origin of the error. Example: [Microsoft][ODBC SQL Server Driver][SQL Server]

The framework interprets the error string and puts its components into **m_strStateNativeOrigin**; if **m_strStateNativeOrigin** contains information for more than one error, the errors are separated by newlines. The framework puts the alphanumeric error text into **m_strError**.

For additional information about the codes used to make up this string, see the SQLError function in the *ODBC Programmer's Reference*.

Example

From ODBC: "State:S0022,Native:207,Origin:[Microsoft][ODBC SQL Server Driver][SQL Server] Invalid column name 'ColName'"

In **m_strStateNativeOrigin**: "State:S0022,Native:207,Origin:[Microsoft][ODBC SQL Server Driver][SQL Server]"

In **m_strError**: "Invalid column name 'ColName'"

See Also

Reference

CDBException Class
Hierarchy Chart
CDBException::m_strError

Other Resources

CDBException Members

CDBVariant Class

Represents a variant data type for the MFC ODBC classes.

```
class CDBVariant
```

Remarks

CDBVariant does not have a base class.

CDBVariant is similar to COleVariant; however, **CDBVariant** does not use OLE. **CDBVariant** allows you to store a value without worrying about the value's data type. **CDBVariant** tracks the data type of the current value, which is stored in a union.

Class CRecordset utilizes **CDBVariant** objects in three member functions: **GetFieldValue**, **GetBookmark**, and **SetBookmark**. For example, **GetFieldValue** allows you to dynamically fetch data in a column. Because the data type of the column may not be known at run time, **GetFieldValue** uses a **CDBVariant** object to store the column's data.

Requirements

Header: afxdb.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart
CRecordset Class
CRecordset::GetFieldValue
CRecordset::GetBookmark
CRecordset::SetBookmark

Other Resources

CDBVariant Members

CDBVariant Members

Data Members

m_boolVal	Contains a value of type **BOOL**.
m_chVal	Contains a value of type **unsigned char**.
m_dblVal	Contains a value of type **double**.
m_dwType	Contains the data type of the currently stored value. Type **DWORD**.
m_fltVal	Contains a value of type **float**.
m_iVal	Contains a value of type **short**.
m_lVal	Contains a value of type **long**.
m_pbinary	Contains a pointer to an object of type **CLongBinary**.
m_pdate	Contains a pointer to an object of type **TIMESTAMP_STRUCT**.
m_pstring	Contains a pointer to an object of type **CString**.
m_pstringA	Stores a pointer to an ASCII CString object.
m_pstringW	Stores a pointer to a wide CString object.

Construction

CDBVariant	Constructs a **CDBVariant** object.

Operations

Clear	Clears the **CDBVariant** object.

See Also

Reference

CDBVariant Class

Hierarchy Chart

CDBVariant Member Functions

For information about the member functions in **CDBVariant**, see CDBVariant Members.

CDBVariant::CDBVariant

Creates a NULL **CDBVariant** object.

```
CDBVariant( );
```

Remarks

Sets the m_dwType data member to **DBVT_NULL**.

See Also

Reference

CDBVariant Class
Hierarchy Chart
CDBVariant::m_dwType

Other Resources

CDBVariant Members

CDBVariant::Clear

Call this member function to clear the **CDBVariant** object.

```
void Clear( );
```

Remarks

If the value of the m_dwType data member is **DBVT_DATE**, **DBVT_STRING**, or **DBVT_BINARY**, **Clear** frees the memory associated with the union pointer member. **Clear** sets **m_dwType** to **DBVT_NULL**.

The **CDBVariant** destructor calls **Clear**.

See Also

Reference

CDBVariant Class

Hierarchy Chart

CDBVariant::m_dwType

Other Resources

CDBVariant Members

CDBVariant Data Members

For information about the data members in **CDBVariant**, see CDBVariant Members.

CDBVariant::m_boolVal

Stores a value of type **BOOL**.

Remarks

The **m_boolVal** data member belongs to a union. Before accessing **m_boolVal**, first check the value of CDBVariant::m_dwType. If **m_dwType** is set to **DBVT_BOOL**, then **m_boolVal** will contain a valid value; otherwise, accessing **m_boolVal** will produce unreliable results.

See Also

Reference

CDBVariant Class

Hierarchy Chart

CDBVariant::m_dwType

Other Resources

CDBVariant Members

CDBVariant::m_chVal

Stores a value of type **unsigned char**.

Remarks

The **m_chVal** data member belongs to a union. Before accessing **m_chVal**, first check the value of CDBVariant::m_dwType. If **m_dwType** is set to **DBVT_UCHAR**, then **m_chVal** contains a valid value; otherwise, accessing **m_chVal** will produce unreliable results.

See Also

Reference

CDBVariant Class
Hierarchy Chart
CDBVariant::m_dwType

Other Resources

CDBVariant Members

CDBVariant::m_dblVal

Stores a value of type **double**.

Remarks

The **m_dblVal** data member belongs to a union. Before accessing **m_dblVal**, first check the value of CDBVariant::m_dwType. If **m_dwType** is set to **DBVT_DOUBLE**, then **m_dblVal** contains a valid value; otherwise, accessing **m_dblVal** will produce unreliable results.

See Also

Reference

CDBVariant Class

Hierarchy Chart

CDBVariant::m_dwType

Other Resources

CDBVariant Members

CDBVariant::m_dwType

This data member contains the data type for the value that is currently stored in the **CDBVariant** object's union data member.

Remarks

Before accessing this union, you must check the value of **m_dwType** in order to determine which union data member to access. The following table lists the possible values for **m_dwType** and the corresponding union data member.

m_dwType	Union data member
DBVT_NULL	No union member is valid for access.
DBVT_BOOL	m_boolVal
DBVT_UCHAR	m_chVal
DBVT_SHORT	m_iVal
DBVT_LONG	m_lVal
DBVT_SINGLE	m_fltVal
DBVT_DOUBLE	m_dblVal
DBVT_DATE	m_pdate
DBVT_STRING	m_pstring
DBVT_BINARY	m_pbinary
DBVT_ASTRING	m_pstringA
DBVT_WSTRING	m_pstringW

See Also

Reference

CDBVariant Class

Hierarchy Chart

Other Resources

CDBVariant Members

CDBVariant::m_fltVal

Stores a value of type **float**.

Remarks

The **m_fltVal** data member belongs to a union. Before accessing **m_fltVal**, first check the value of CDBVariant::m_dwType. If **m_dwType** is set to **DBVT_SINGLE**, then **m_fltVal** contains a valid value; otherwise, accessing **m_fltVal** will produce unreliable results.

See Also

Reference

CDBVariant Class
Hierarchy Chart
CDBVariant::m_dwType

Other Resources

CDBVariant Members

CDBVariant::m_iVal

Stores a value of type **short**.

Remarks

The **m_iVal** data member belongs to a union. Before accessing **m_iVal**, first check the value of CDBVariant::m_dwType. If **m_dwType** is set to **DBVT_SHORT**, then **m_iVal** contains a valid value; otherwise, accessing **m_iVal** will produce unreliable results.

See Also

Reference

CDBVariant Class

Hierarchy Chart

CDBVariant::m_dwType

Other Resources

CDBVariant Members

CDBVariant::m_lVal

Stores a value of type **long**.

Remarks

The **m_lVal** data member belongs to a union. Before accessing **m_lVal**, first check the value of CDBVariant::m_dwType. If **m_dwType** is set to **DBVT_LONG**, then **m_lVal** contains a valid value; otherwise, accessing **m_lVal** will produce unreliable results.

See Also

Reference

CDBVariant Class

Hierarchy Chart

CDBVariant::m_dwType

Other Resources

CDBVariant Members

CDBVariant::m_pbinary

Stores a pointer to an object of type CLongBinary.

Remarks

The **m_pbinary** data member belongs to a union. Before accessing **m_pbinary**, first check the value of CDBVariant::m_dwType. If **m_dwType** is set to **DBVT_BINARY**, then **m_pbinary** contains a valid pointer; otherwise, accessing **m_pbinary** will produce unreliable results.

See Also

Reference

CDBVariant Class
Hierarchy Chart
CDBVariant::m_dwType

Other Resources

CDBVariant Members

CDBVariant::m_pdate

Stores a pointer to an object of type **TIMESTAMP_STRUCT**.

Remarks

The **m_pdate** data member belongs to a union. Before accessing **m_pdate**, first check the value of CDBVariant::m_dwType. If **m_dwType** is set to **DBVT_DATE**, then **m_pdate** contains a valid pointer; otherwise, accessing **m_pdate** will produce unreliable results.

For more information about the **TIMESTAMP_STRUCT** data type, see the topic C Data Types in Appendix D of the *ODBC Programmer's Reference* in the *Platform SDK*.

See Also

Reference

CDBVariant Class
Hierarchy Chart
CDBVariant::m_dwType

Other Resources

CDBVariant Members

CDBVariant::m_pstring

Stores a pointer to an object of type CString.

Remarks

The **m_pstring** data member belongs to a union. Before accessing **m_pstring**, first check the value of CDBVariant::m_dwType. If **m_dwType** is set to **DBVT_STRING**, then **m_pstring** contains a valid pointer; otherwise, accessing **m_pstring** will produce unreliable results.

See Also

Reference

CDBVariant Class
Hierarchy Chart
CDBVariant::m_dwType

Other Resources

CDBVariant Members

CDBVariant::m_pstringA

Stores a pointer to an ASCII CString object.

Remarks

The **m_pstringA** data member belongs to a union. Before accessing **m_pstringA**, first check the value of CDBVariant::m_dwType. If **m_dwType** is set to **DBVT_ASTRING**, then **m_pstringA** contains a valid pointer; otherwise, accessing **m_pstringA** will produce unreliable results.

See Also

Reference

CDBVariant Class

Hierarchy Chart

CDBVariant::m_dwType

Other Resources

CDBVariant Members

CDBVariant::m_pstringW

Stores a pointer to a wide CString object.

Remarks

The **m_pstringW** data member belongs to a union. Before accessing **m_pstringW**, first check the value of CDBVariant::m_dwType. If **m_dwType** is set to **DBVT_WSTRING**, then **m_pstringW** contains a valid pointer; otherwise, accessing **m_pstringW** will produce unreliable results.

See Also

Reference

CDBVariant Class

Hierarchy Chart

CDBVariant::m_dwType

Other Resources

CDBVariant Members

CDC Class

Defines a class of device-context objects.

```
class CDC : public CObject
```

Remarks

The **CDC** object provides member functions for working with a device context, such as a display or printer, as well as members for working with a display context associated with the client area of a window.

Do all drawing through the member functions of a **CDC** object. The class provides member functions for device-context operations, working with drawing tools, type-safe graphics device interface (GDI) object selection, and working with colors and palettes. It also provides member functions for getting and setting drawing attributes, mapping, working with the viewport, working with the window extent, converting coordinates, working with regions, clipping, drawing lines, and drawing simple shapes, ellipses, and polygons. Member functions are also provided for drawing text, working with fonts, using printer escapes, scrolling, and playing metafiles.

To use a **CDC** object, construct it, and then call its member functions that parallel Windows functions that use device contexts.

Note
Under Windows 95/98, all screen coordinates are limited to 16 bits. Therefore, an **int** passed to a **CDC** member function mus t lie in the range –32768 to 32767.

For specific uses, the Microsoft Foundation Class Library provides several classes derived from **CDC** . **CPaintDC** encapsulates calls to **BeginPaint** and **EndPaint**. **CClientDC** manages a display context associated with a window's client area. **CWindowDC** manages a display context associated with an entire window, including its frame and controls. **CMetaFileDC** associates a device context with a metafile.

CDC provides two member functions, GetLayout and SetLayout, for reversing the layout of a device context, which does not inherit its layout from a window. Such right-to-left orientation is necessary for applications written for cultures, such as Arabic or Hebrew, where the character layout is not the European standard.

CDC contains two device contexts, m_hDC and m_hAttribDC, which, on creation of a **CDC** object, refer to the same device. **CDC** directs all output GDI calls to **m_hDC** and most attribute GDI calls to **m_hAttribDC**. (An example of an attribute call is **GetTextColor**, while **SetTextColor** is an output call.)

For example, the framework uses these two device contexts to implement a **CMetaFileDC** object that will send output to a metafile while reading attributes from a physical device. Print preview is implemented in the framework in a similar fashion. You can also use the two device contexts in a similar way in your application-specific code.

There are times when you may need text-metric information from both the **m_hDC** and **m_hAttribDC** device contexts. The following pairs of functions provide this capability:

Uses m_hAttribDC	**Uses m_hDC**
GetTextExtent	GetOutputTextExtent
GetTabbedTextExtent	GetOutputTabbedTextExtent
GetTextMetrics	GetOutputTextMetrics
GetCharWidth	GetOutputCharWidth

For more information on **CDC**, see Device Contexts.

Requirements

Header: afxwin.h

See Also

Reference

CObject Class
Hierarchy Chart
CPaintDC Class
CWindowDC Class
CClientDC Class
CMetaFileDC Class

Other Resources

CDC Members

CDC Members

Base Class Members

CObject Members

Color and Color Palette Functions	Font Functions
Construction/Destruction	Line-Output Functions
Coordinate Functions	Path Functions
Data Members	Clipping Functions
Device-Context Functions	Ellipse and Polygon Functions
Drawing-Attribute Functions	Printer Escape Functions
Drawing-Tool Functions	Bitmap Functions
Initialization	Simple Drawing Functions
Layout Functions	Metafile Functions
Mapping Functions	Scrolling Functions
Region Functions	Text Functions
Type-Safe Selection. Helpers	

Data Members

m_hAttribDC	The attribute-device context used by this **CDC** object.
m_hDC	The output-device context used by this **CDC** object.

Construction

CDC	Constructs a **CDC** object.

Initialization

Attach	Attaches a Windows device context to this **CDC** object.
CreateCompatibleDC	Creates a memory-device context that is compatible with another device context. You can use it to prepare images in memory.
CreateDC	Creates a device context for a specific device.
CreateIC	Creates an information context for a specific device. This provides a fast way to get information about the device without creating a device context.
DeleteDC	Deletes the Windows device context associated with this **CDC** object.
DeleteTempMap	Called by the **CWinApp** idle-time handler to delete any temporary **CDC** object created by **FromHandle**. Also detaches the device context.

Detach	Detaches the Windows device context from this **CDC** object.
FromHandle	Returns a pointer to a **CDC** object when given a handle to a device context. If a **CDC** object is not attached to the handle, a temporary **CDC** object is created and attached.
GetCurrentBitmap	Returns a pointer to the currently selected **CBitmap** object.
GetCurrentBrush	Returns a pointer to the currently selected **CBrush** object.
GetCurrentFont	Returns a pointer to the currently selected **CFont** object.
GetCurrentPalette	Returns a pointer to the currently selected **CPalette** object.
GetCurrentPen	Returns a pointer to the currently selected **CPen** object.
GetGraphicsMode	Retrieves the current graphics mode for the specified device context.
GetWindow	Returns the window associated with the display device context.
ReleaseAttribDC	Releases **m_hAttribDC**, the attribute device context.
ReleaseOutputDC	Releases **m_hDC**, the output device context.
SetAttribDC	Sets **m_hAttribDC**, the attribute device context.
SetOutputDC	Sets **m_hDC**, the output device context.
SetGraphicsMode	Sets the current graphics mode for the specified device context.

Device-Context Functions

GetDeviceCaps	Retrieves a specified kind of device-specific information about a given display device's capabilities.
GetSafeHdc	Returns m_hDC, the output device context.
IsPrinting	Determines whether the device context is being used for printing.
ResetDC	Updates the **m_hAttribDC** device context.
RestoreDC	Restores the device context to a previous state saved with **SaveDC**.
SaveDC	Saves the current state of the device context.

Drawing-Tool Functions

EnumObjects	Enumerates the pens and brushes available in a device context.
GetBrushOrg	Retrieves the origin of the current brush.
SetBrushOrg	Specifies the origin for the next brush selected into a device context.

Type-Safe Selection Helpers

SelectObject	Selects a GDI drawing object such as a pen.
SelectStockObject	Selects one of the predefined stock pens, brushes, or fonts provided by Windows.

Color and Color Palette Functions

GetHalftoneBrush	Retrieves a halftone brush.
GetNearestColor	Retrieves the closest logical color to a specified logical color that the given device can represent.
RealizePalette	Maps palette entries in the current logical palette to the system palette.
SelectPalette	Selects the logical palette.
UpdateColors	Updates the client area of the device context by matching the current colors in the client area to the system palette on a pixel-by-pixel basis.

Drawing-Attribute Functions

GetBkColor	Retrieves the current background color.
GetBkMode	Retrieves the background mode.
GetColorAdjustment	Retrieves the color adjustment values for the device context.
GetDCBrushColor	Retrieves the current brush color.
GetDCPenColor	Retrieves the current pen color.
GetPolyFillMode	Retrieves the current polygon-filling mode.
GetROP2	Retrieves the current drawing mode.
GetStretchBltMode	Retrieves the current bitmap-stretching mode.
GetTextColor	Retrieves the current text color.
SetBkColor	Sets the current background color.
SetBkMode	Sets the background mode.
SetColorAdjustment	Sets the color adjustment values for the device context using the specified values.
SetDCBrushColor	Sets the current brush color.
SetDCPenColor	Sets the current pen color.
SetPolyFillMode	Sets the polygon-filling mode.
SetROP2	Sets the current drawing mode.
SetStretchBltMode	Sets the bitmap-stretching mode.
SetTextColor	Sets the text color.

Mapping Functions

GetMapMode	Retrieves the current mapping mode.
GetViewportExt	Retrieves the x- and y-extents of the viewport.

GetViewportOrg	Retrieves the x- and y-coordinates of the viewport origin.
GetWindowExt	Retrieves the x- and y-extents of the associated window.
GetWindowOrg	Retrieves the x- and y-coordinates of the origin of the associated window.
GetWorldTransform	Retrieves the current world-space to page-space transformation.
ModifyWorldTransform	Changes the world transformation for a device context using the specified mode.
OffsetViewportOrg	Modifies the viewport origin relative to the coordinates of the current viewport origin.
OffsetWindowOrg	Modifies the window origin relative to the coordinates of the current window origin.
ScaleViewportExt	Modifies the viewport extent relative to the current values.
ScaleWindowExt	Modifies the window extents relative to the current values.
SetMapMode	Sets the current mapping mode.
SetViewportExt	Sets the x- and y-extents of the viewport.
SetViewportOrg	Sets the viewport origin.
SetWindowExt	Sets the x- and y-extents of the associated window.
SetWindowOrg	Sets the window origin of the device context.
GetWorldTransform	Sets the current world-space to page-space transformation.

Layout Functions

GetLayout	Retrieves the layout of a device context (DC). The layout can be either left to right (default) or right to left (mirrored).
SetLayout	Changes the layout of a device context (DC).

Coordinate Functions

DPtoHIMETRIC	Converts device units into **HIMETRIC** units.
DPtoLP	Converts device units into logical units.
HIMETRICtoDP	Converts **HIMETRIC** units into device units.
HIMETRICtoLP	Converts **HIMETRIC** units into logical units.
LPtoDP	Converts logical units into device units.
LPtoHIMETRIC	Converts logical units into **HIMETRIC** units.

Region Functions

FillRgn	Fills a specific region with the specified brush.
FrameRgn	Draws a border around a specific region using a brush.

InvertRgn	Inverts the colors in a region.
PaintRgn	Fills a region with the selected brush.

Clipping Functions

ExcludeClipRect	Creates a new clipping region that consists of the existing clipping region minus the specified rectangle.
ExcludeUpdateRgn	Prevents drawing within invalid areas of a window by excluding an updated region in the window from a clipping region.
GetBoundsRect	Returns the current accumulated bounding rectangle for the specified device context.
GetClipBox	Retrieves the dimensions of the tightest bounding rectangle around the current clipping boundary.
IntersectClipRect	Creates a new clipping region by forming the intersection of the current region and a rectangle.
OffsetClipRgn	Moves the clipping region of the given device.
PtVisible	Specifies whether the given point is within the clipping region.
RectVisible	Determines whether any part of the given rectangle lies within the clipping region.
SelectClipRgn	Combines the given region with the current clipping region by using the specified mode.
SetBoundsRect	Controls the accumulation of bounding-rectangle information for the specified device context.

Line-Output Functions

AngleArc	Draws a line segment and an arc, and moves the current position to the ending point of the arc.
Arc	Draws an elliptical arc.
ArcTo	Draws an elliptical arc. This function is similar to **Arc**, except that the current position is updated.
GetArcDirection	Returns the current arc direction for the device context.
GetCurrentPosition	Retrieves the current position of the pen (in logical coordinates).
LineTo	Draws a line from the current position up to, but not including, a point.
MoveTo	Moves the current position.
PolyBezier	Draws one or more Bzier splines. The current position is neither used nor updated.
PolyBezierTo	Draws one or more Bzier splines, and moves the current position to the ending point of the last Bzier spline.
PolyDraw	Draws a set of line segments and Bzier splines. This function updates the current position.
Polyline	Draws a set of line segments connecting the specified points.
PolylineTo	Draws one or more straight lines and moves the current position to the ending point of the last line.

PolyPolyline	Draws multiple series of connected line segments. The current position is neither used nor updated by this function.
SetArcDirection	Sets the drawing direction to be used for arc and rectangle functions.

Simple Drawing Functions

Draw3dRect	Draws a three-dimensional rectangle.
DrawDragRect	Erases and redraws a rectangle as it is dragged.
DrawEdge	Draws the edges of a rectangle.
DrawFrameControl	Draw a frame control.
DrawIcon	Draws an icon.
DrawState	Displays an image and applies a visual effect to indicate a state.
FillRect	Fills a given rectangle by using a specific brush.
FillSolidRect	Fills a rectangle with a solid color.
FrameRect	Draws a border around a rectangle.
InvertRect	Inverts the contents of a rectangle.

Ellipse and Polygon Functions

Chord	Draws a chord (a closed figure bounded by the intersection of an ellipse and a line segment).
DrawFocusRect	Draws a rectangle in the style used to indicate focus.
Ellipse	Draws an ellipse.
Pie	Draws a pie-shaped wedge.
Polygon	Draws a polygon consisting of two or more points (vertices) connected by lines.
Polyline	Draws a polygon consisting of a set of line segments connecting specified points.
PolyPolygon	Creates two or more polygons that are filled using the current polygon-filling mode. The polygons may be disj oint or they may overlap.
Rectangle	Draws a rectangle using the current pen and fills it using the current brush.
RoundRect	Draws a rectangle with rounded corners using the current pen and filled using the current brush.

Bitmap Functions

AlphaBlend	Displays bitmaps that have transparent or semitransparent pixels.
BitBlt	Copies a bitmap from a specified device context.
ExtFloodFill	Fills an area with the current brush. Provides more flexibility than the FloodFill member function.

FloodFill	Fills an area with the current brush.
GetPixel	Retrieves the RGB color value of the pixel at the specified point.
GradientFill	Fills rectangle and triangle structures with a gradating color.
MaskBlt	Combines the color data for the source and destination bitmaps using the given mask and raster operation.
PatBlt	Creates a bit pattern.
PlgBlt	Performs a bit-block transfer of the bits of color data from the specified rectangle in the source device context to the specified parallelogram in the given device context.
SetPixel	Sets the pixel at the specified point to the closest approximation of the specified color.
SetPixelV	Sets the pixel at the specified coordinates to the closest approximation of the specified color. **SetPixelV** is faster than **SetPixel** because it does not need to return the color value of the point actually painted.
StretchBlt	Moves a bitmap from a source rectangle and device into a destination rectangle, stretching or compressing the bitmap if necessary to fit the dimensions of the destination rectangle.
TransparentBlt	Transfers a bit-block of color data from the specified source device context into a destination device context, rendering a specified color transparent in the transfer.

Text Functions

DrawText	Draws formatted text in the specified rectangle.
DrawTextEx	Draws formatted text in the specified rectangle using additional formats.
ExtTextOut	Writes a character string within a rectangular region using the currently selected font.
GetCharABCWidthsI	Retrieves the widths, in logical units, of consecutive glyph indices in a specified range from the current TrueType font.
GetCharacterPlacement	Retrieves various types of information on a character string.
GetCharWidthI	Retrieves the widths, in logical coordinates, of consecutive glyph indices in a specified range from the current font.
GetOutputTabbedTextExtent	Computes the width and height of a character string on the output device context.
GetOutputTextExtent	Computes the width and height of a line of text on the output device context using the current font to determine the dimensions.
GetOutputTextMetrics	Retrieves the metrics for the current font from the output device context.
GetTabbedTextExtent	Computes the width and height of a character string on the attribute device context.
GetTextAlign	Retrieves the text-alignment flags.
GetTextCharacterExtra	Retrieves the current setting for the amount of intercharacter spacing.

GetTextExtent	Computes the width and height of a line of text on the attribute device context using the current f ont to determine the dimensions.
GetTextExtentExPointI	Retrieves the number of characters in a specified string that will fit within a specified space and fill s an array with the text extent for each of those characters.
GetTextExtentPointI	Retrieves the width and height of the specified array of glyph indices.
GetTextFace	Copies the typeface name of the current font into a buffer as a null-terminated string.
GetTextMetrics	Retrieves the metrics for the current font from the attribute device context.
GrayString	Draws dimmed (grayed) text at the given location.
SetTextAlign	Sets the text-alignment flags.
SetTextCharacterExtra	Sets the amount of intercharacter spacing.
SetTextJustification	Adds space to the break characters in a string.
TabbedTextOut	Writes a character string at a specified location, expanding tabs to the values specified in an array of tab-stop positions.
TextOut	Writes a character string at a specified location using the currently selected font.

Font Functions

GetAspectRatioFilter	Retrieves the setting for the current aspect-ratio filter.
GetCharABCWidths	Retrieves the widths, in logical units, of consecutive characters in a given range from the current font.
GetCharWidth	Retrieves the fractional widths of consecutive characters in a given range from the current font.
GetFontData	Retrieves font metric information from a scalable font file. The information to retrieve is identified by sp ecifying an offset into the font file and the length of the information to return.
GetFontLanguageInfo	Returns information about the currently selected font for the specified display context.
GetGlyphOutline	Retrieves the outline curve or bitmap for an outline character in the current font.
GetKerningPairs	Retrieves the character kerning pairs for the font that is currently selected in the specified device contex t.
GetOutlineTextMetrics	Retrieves font metric information for TrueType fonts.
GetOutputCharWidth	Retrieves the widths of individual characters in a consecutive group of characters from the current font using the output device context.
SetMapperFlags	Alters the algorithm that the font mapper uses when it maps logical fonts to physical fonts.

Printer Escape Functions

AbortDoc	Terminates the current print job, erasing everything the application has written to the device since the last call of the **StartDoc** member function.

DrawEscape	Accesses drawing capabilities of a video display that are not directly available through the graphics device interface (GDI).
EndDoc	Ends a print job started by the **StartDoc** member function.
EndPage	Informs the device driver that a page is ending.
Escape	Allows applications to access facilities that are not directly available from a particular device through GDI. Also allows access to Windows escape functions. Escape calls made by an application are translated and sent to the device driver.
QueryAbort	Calls the AbortProc callback function for a printing application and queries whether the printing should be terminated.
SetAbortProc	Sets a programmer-supplied callback function that Windows calls if a print job must be aborted.
StartDoc	Informs the device driver that a new print job is starting.
StartPage	Informs the device driver that a new page is starting.

Scrolling Functions

ScrollDC	Scrolls a rectangle of bits horizontally and vertically.

Metafile Functions

AddMetaFileComment	Copies the comment from a buffer into a specified enhanced-format metafile.
PlayMetaFile	Plays the contents of the specified metafile on the given device. The enhanced version of **PlayMetaFile** displays the picture stored in the given enhanced-format metafile. The metafile can be played any number of times.

Path Functions

AbortPath	Closes and discards any paths in the device context.
BeginPath	Opens a path bracket in the device context.
CloseFigure	Closes an open figure in a path.
EndPath	Closes a path bracket and selects the path defined by the bracket into the device context.
FillPath	Closes any open figures in the current path and fills the path's interior by using the current brush and polygon-filling mode.
FlattenPath	Transforms any curves in the path selected into the current device context, and turns each curve into a sequence of lines.
GetMiterLimit	Returns the miter limit for the device context.
GetPath	Retrieves the coordinates defining the endpoints of lines and the control points of curves found in the path that is selected into the device context.
SelectClipPath	Selects the current path as a clipping region for the device context, combining the new region with any existing clipping region by using the specified mode.
SetMiterLimit	Sets the limit for the length of miter joins for the device context.

StrokeAndFillPath	Closes any open figures in a path, strikes the outline of the path by using the current pen, and fills its interior by using the current brush.
StrokePath	Renders the specified path by using the current pen.
WidenPath	Redefines the current path as the area that would be painted if the path were stroked using the pen currently selected into the device context.

Operators

operator HDC	Retrieves the handle of the device context.

See Also

Reference

CDC Class

Hierarchy Chart

CDC Member Functions

For information about the member functions in **CDC**, see CDC Members.

CDC::AbortDoc

Terminates the current print job and erases everything the application has written to the device since the last call to the StartDoc member function.

```
int AbortDoc( );
```

Return Value

A value greater than or equal to 0 if successful, or a negative value if an error has occurred. The following list shows common error values and their meanings:

- **SP_ERROR** General error.
- **SP_OUTOFDISK** Not enough disk space is currently available for spooling, and no more space will become available.
- **SP_OUTOFMEMORY** Not enough memory is available for spooling.
- **SP_USERABORT** User terminated the job through the Print Manager.

Remarks

This member function replaces the **ABORTDOC** printer escape.

AbortDoc should be used to terminate the following:

- Printing operations that do not specify an abort function using SetAbortProc.
- Printing operations that have not yet reached their first **NEWFRAME** or **NEXTBAND** escape call.

If an application encounters a printing error or a canceled print operation, it must not attempt to terminate the operation by using either the EndDoc or **AbortDoc** member functions of class **CDC**. GDI automatically terminates the operation before returning the error value.

If the application displays a dialog box to allow the user to cancel the print operation, it must call **AbortDoc** before destroying the dialog box.

If Print Manager was used to start the print job, calling **AbortDoc** erases the entire spool job — the printer receives nothing. If Print Manager was not used to start the print job, the data may have been sent to the printer before **AbortDoc** was called. In this case, the printer driver would have reset the printer (when possible) and closed the print job.

Example

See the example for CDC::StartDoc.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::StartDoc
CDC::EndDoc
CDC::SetAbortProc

Other Resources

CDC Members

CDC::AbortPath

Closes and discards any paths in the device context.

```
BOOL AbortPath( );
```

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

If there is an open path bracket in the device context, the path bracket is closed and the path is discarded. If there is a closed path in the device context, the path is discarded.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::BeginPath
CDC::EndPath

Other Resources

CDC Members

CDC::AddMetaFileComment

Copies the comment from a buffer into a specified enhanced-format metafile.

```
BOOL AddMetaFileComment(
   UINT nDataSize,
   const BYTE* pCommentData
);
```

Parameters

nDataSize

Specifies the length of the comment buffer, in bytes.

pCommentData

Points to the buffer that contains the comment.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

A comment may include any private information — for example, the source of the picture and the date it was created. A comment should begin with an application signature, followed by the data. Comments should not contain position-specific data. Position-specific data specifies the location of a record, and it should not be included because one metafile may be embedded within another metafile. This function can only be used with enhanced metafiles.

See Also

Reference

CDC Class
Hierarchy Chart
CMetaFileDC::CreateEnhanced
GdiComment

Other Resources

CDC Members

CDC::AlphaBlend

Call this member function to display bitmaps that have transparent or semitransparent pixels.

```
BOOL AlphaBlend(
   int xDest,
   int yDest,
   int nDestWidth,
   int nDestHeight,
   CDC* pSrcDC,
   int xSrc,
   int ySrc,
   int nSrcWidth,
   int nSrcHeight,
   BLENDFUNCTION blend
);
```

Parameters

xDest

Specifies the x-coordinate, in logical units, of the upper-left corner of the destination rectangle.

yDest

Specifies the y-coordinate, in logical units, of the upper-left corner of the destination rectangle.

nDestWidth

Specifies the width, in logical units, of the destination rectangle.

nDestHeight

Specifies the height, in logical units, of the destination rectangle.

pSrcDC

A pointer to the source device context.

xSrc

Specifies the x-coordinate, in logical units, of the upper-left corner of the source rectangle.

ySrc

Specifies the y-coordinate, in logical units, of the upper-left corner of the source rectangle.

nSrcWidth

Specifies the width, in logical units, of the source rectangle.

nSrcHeight

Specifies the height, in logical units, of the source rectangle.

blend

Specifies a BLENDFUNCTION structure.

Return Value

TRUE if successful; otherwise **FALSE**.

Remarks

See AlphaBlend in the *Platform SDK* for more information.

See Also

Reference

CDC Class

Hierarchy Chart

EMRALPHABLEND

Other Resources

CDC Members

CDC::AngleArc

Draws a line segment and an arc.

```
BOOL AngleArc(
   int x,
   int y,
   int nRadius,
   float fStartAngle,
   float fSweepAngle
);
```

Parameters

x

Specifies the logical x-coordinate of the center of the circle.

y

Specifies the logical y-coordinate of the center of the circle.

nRadius

Specifies the radius of the circle in logical units. This value must be positive.

fStartAngle

Specifies the starting angle in degrees relative to the x-axis.

fSweepAngle

Specifies the sweep angle in degrees relative to the starting angle.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The line segment is drawn from the current position to the beginning of the arc. The arc is drawn along the perimeter of a circle with the given radius and center. The length of the arc is defined by the given start and sweep angles.

AngleArc moves the current position to the ending point of the arc. The arc drawn by this function may appear to be elliptical, depending on the current transformation and mapping mode. Before drawing the arc, this function draws the line segment from the current position to the beginning of the arc. The arc is drawn by constructing an imaginary circle with the specified radius around the specified center point. The starting point of the arc is determined by measuring counterclockwise from the x-axis of the circle by the number of degrees in the start angle. The ending point is similarly located by measuring counterclockwise from the starting point by the number of degrees in the sweep angle.

If the sweep angle is greater than 360 degrees the arc is swept multiple times. This function draws lines by using the current pen. The figure is not filled.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::Arc
CDC::ArcTo
CDC::MoveTo
AngleArc

Other Resources

CDC Members

CDC::Arc

Draws an elliptical arc.

```
BOOL Arc(
   int x1,
   int y1,
   int x2,
   int y2,
   int x3,
   int y3,
   int x4,
   int y4
);
BOOL Arc(
   LPCRECT lpRect,
   POINT ptStart,
   POINT ptEnd
);
```

Parameters

x1

Specifies the x-coordinate of the upper-left corner of the bounding rectangle (in logical units).

y1

Specifies the y-coordinate of the upper-left corner of the bounding rectangle (in logical units).

x2

Specifies the x-coordinate of the lower-right corner of the bounding rectangle (in logical units).

y2

Specifies the y-coordinate of the lower-right corner of the bounding rectangle (in logical units).

x3

Specifies the x-coordinate of the point that defines the arc's starting point (in logical units). This point does not have to lie exactly on the arc.

y3

Specifies the y-coordinate of the point that defines the arc's starting point (in logical units). This point does not have to lie exactly on the arc.

x4

Specifies the x-coordinate of the point that defines the arc's endpoint (in logical units). This point does not have to lie exactly on the arc.

y4

Specifies the y-coordinate of the point that defines the arc's endpoint (in logical units). This point does not have to lie exactly on the arc.

lpRect

Specifies the bounding rectangle (in logical units). You can pass either an **LPRECT** or a CRect object for this parameter.

ptStart

Specifies the x- and y-coordinates of the point that defines the arc's starting point (in logical units). This point does not have to lie exactly on the arc. You can pass either a POINT structure or a CPoint object for this parameter.

ptEnd

Specifies the x- and y-coordinates of the point that defines the arc's ending point (in logical units). This point does not have

to lie exactly on the arc. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The arc drawn by using the function is a segment of the ellipse defined by the specified bounding rectangle.

The actual starting point of the arc is the point at which a ray drawn from the center of the bounding rectangle through the specified starting point intersects the ellipse. The actual ending point of the arc is the point at which a ray drawn from the center of the bounding rectangle through the specified ending point intersects the ellipse. The arc is drawn in a counterclockwise direction. Since an arc is not a closed figure, it is not filled. Both the width and height of the rectangle must be greater than 2 units and less than 32,767 units.

Example

```
void CCurvesView::OnDraw(CDC* pDC)
{
   // Fill the client area with a thin circle. The circle's
   // interior is not filled. The circle's perimeter is
   // blue from 6 o'clock to 3 o'clock and red from 3
   // o'clock to 6 o'clock.

   // Get the client area.
   CRect rectClient;
   GetClientRect(rectClient);

   // Make a couple of pens.
   CPen penBlue;
   CPen penRed;
   CPen* pOldPen;

   penBlue.CreatePen(PS_SOLID | PS_COSMETIC, 1, RGB(0, 0, 255));
   penRed.CreatePen(PS_SOLID | PS_COSMETIC, 1, RGB(255, 0, 0));

   // Draw from 3 o'clock to 6 o'clock, counterclockwise,
   // in a blue pen.

   pOldPen = pDC->SelectObject(&penBlue);

   pDC->Arc(rectClient,
      CPoint(rectClient.right, rectClient.CenterPoint().y),
      CPoint(rectClient.CenterPoint().x, rectClient.right));

   // Draw from 6 o'clock to 3 o'clock, counterclockwise,
   // in a red pen.
   pDC->SelectObject(&penRed);

   // Keep the same parameters, but reverse start
   // and end points.
   pDC->Arc(rectClient,
      CPoint(rectClient.CenterPoint().x, rectClient.right),
      CPoint(rectClient.right, rectClient.CenterPoint().y));

   // Restore the previous pen.
   pDC->SelectObject(pOldPen);
}
```

See Also

Reference

CDC Class
Hierarchy Chart
CDC::Chord
Arc
POINT Structure
RECT Structure

Other Resources

CDC Members

CDC::ArcTo

Draws an elliptical arc.

```
BOOL ArcTo(
   int x1,
   int y1,
   int x2,
   int y2,
   int x3,
   int y3,
   int x4,
   int y4
);
BOOL ArcTo(
   LPCRECT lpRect,
   POINT ptStart,
   POINT ptEnd
);
```

Parameters

x1

Specifies the x-coordinate of the upper-left corner of the bounding rectangle (in logical units).

y1

Specifies the y-coordinate of the upper-left corner of the bounding rectangle (in logical units).

x2

Specifies the x-coordinate of the lower-right corner of the bounding rectangle (in logical units).

y2

Specifies the y-coordinate of the lower-right corner of the bounding rectangle (in logical units).

x3

Specifies the x-coordinate of the point that defines the arc's starting point (in logical units). This point does not have to lie exactly on the arc.

y3

Specifies the y-coordinate of the point that defines the arc's starting point (in logical units). This point does not have to lie exactly on the arc.

x4

Specifies the x-coordinate of the point that defines the arc's endpoint (in logical units). This point does not have to lie exactly on the arc.

y4

Specifies the y-coordinate of the point that defines the arc's endpoint (in logical units). This point does not have to lie exactly on the arc.

lpRect

Specifies the bounding rectangle (in logical units). You can pass either a pointer to a RECT data structure or a CRect object for this parameter.

ptStart

Specifies the x- and y-coordinates of the point that defines the arc's starting point (in logical units). This point does not have to lie exactly on the arc. You can pass either a POINT data structure or a CPoint object for this parameter.

ptEnd

Specifies the x- and y-coordinates of the point that defines the arc's ending point (in logical units). This point does not have to lie exactly on the arc. You can pass either a **POINT** data structure or a **CPoint** object for this parameter.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

This function is similar to **CDC::Arc**, except that the current position is updated. The points (*x1*,*y1*) and (*x2*,*y2*) specify the bounding rectangle. An ellipse formed by the given bounding rectangle defines the curve of the arc. The arc extends counterclockwise (the default arc direction) from the point where it intersects the radial line from the center of the bounding rectangle to (*x3*,*y3*). The arc ends where it intersects the radial line from the center of the bounding rectangle to (*x4*,*y4*). If the starting point and ending point are the same, a complete ellipse is drawn.

A line is drawn from the current position to the starting point of the arc. If no error occurs, the current position is set to the ending point of the arc. The arc is drawn using the current pen; it is not filled.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::AngleArc
CDC::Arc
CDC::SetArcDirection
ArcTo

Other Resources

CDC Members

CDC::Attach

Use this member function to attach an *hDC* to the **CDC** object.

```
BOOL Attach(
   HDC hDC
);
```

Parameters

hDC

A Windows device context.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The *hDC* is stored in both **m_hDC**, the output device context, and in **m_hAttribDC**, the attribute device context.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::Detach
CDC::m_hDC
CDC::m_hAttribDC

Other Resources

CDC Members

CDC::BeginPath

Opens a path bracket in the device context.

```
BOOL BeginPath( );
```

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

After a path bracket is open, an application can begin calling GDI drawing functions to define the points that lie in the path. An application can close an open path bracket by calling the **EndPath** member function. When an application calls **BeginPath**, any previous paths are discarded.

See BeginPath in the *Platform SDK* for a list of the drawing functions that define points in a path for Windows NT and Windows 95/98.

Example

```
// This OnDraw() implementation uses GDI paths to draw the outline of
// some text in a TrueType font. The path is used to record the way
// the TrueType font would be drawn. Then, the function uses the data
// returned from CDC::GetPath() to draw the font--without filling it.

void COutlineView::OnDraw(CDC* pDC)
{
   // Describe a 24-point truetype font of normal weight
   LOGFONT lf;
   memset(&lf, 0, sizeof(lf));
   lf.lfHeight = -MulDiv(24, pDC->GetDeviceCaps(LOGPIXELSY), 72);
   lf.lfWeight = FW_NORMAL;
   lf.lfOutPrecision = OUT_TT_ONLY_PRECIS;

   // create and select it
   CFont newFont;
   if (!newFont.CreateFontIndirect(&lf))
      return;
   CFont* pOldFont = pDC->SelectObject(&newFont);

   // use a path to record how the text was drawn
   pDC->BeginPath();
   pDC->TextOut(10, 10, _T("Hockey is Best!"));
   pDC->EndPath();

   // Find out how many points are in the path. Note that
   // for long strings or complex fonts, this number might be
   // gigantic!
   int nNumPts = pDC->GetPath(NULL, NULL, 0);
   if (nNumPts == 0)
      return;

   // Allocate memory to hold points and stroke types from
   // the path.
   LPPOINT lpPoints = new POINT[nNumPts];
   if (lpPoints == NULL)
      return;
   LPBYTE lpTypes = new BYTE[nNumPts];
   if (lpTypes == NULL)
   {
      delete [] lpPoints;
      return;
   }
```

```
   // Now that we have the memory, really get the path data.
   nNumPts = pDC->GetPath(lpPoints, lpTypes, nNumPts);

   // If it worked, draw the lines. Win95 and Win98 don't support
   // the PolyDraw API, so we use our own member function to do
   // similar work. If you're targeting only Windows NT, you can
   // use the PolyDraw() API and avoid the COutlineView::PolyDraw()
   // member function.

   if (nNumPts != -1)
      PolyDraw(pDC, lpPoints, lpTypes, nNumPts);

   // Release the memory we used
   delete [] lpPoints;
   delete [] lpTypes;

   // Put back the old font
   pDC->SelectObject(pOldFont);

   return;
}

void COutlineView::PolyDraw(CDC* pDC, CONST LPPOINT lppt, CONST LPBYTE lpbTypes,
   int cCount)
{
   int nIndex;
   LPPOINT pptLastMoveTo = NULL;

   // for each of the points we have...
   for (nIndex = 0; nIndex < cCount; nIndex++)
   {
      switch(lpbTypes[nIndex])
      {
      // React to information from the path by drawing the data
      // we received. For each of the points, record our own
      // "last active point" so we can close figures, lines, and
      // Beziers.

      case PT_MOVETO:
         if (pptLastMoveTo != NULL && nIndex > 0)
            pDC->LineTo(pptLastMoveTo->x, pptLastMoveTo->y);
         pDC->MoveTo(lppt[nIndex].x, lppt[nIndex].y);
         pptLastMoveTo = &lppt[nIndex];
         break;

      case PT_LINETO | PT_CLOSEFIGURE:
         pDC->LineTo(lppt[nIndex].x, lppt[nIndex].y);
         if (pptLastMoveTo != NULL)
            pDC->LineTo(pptLastMoveTo->x, pptLastMoveTo->y);
         pptLastMoveTo = NULL;
         break;

      case PT_LINETO:
         pDC->LineTo(lppt[nIndex].x, lppt[nIndex].y);
         break;

      case PT_BEZIERTO | PT_CLOSEFIGURE:
         ASSERT(nIndex + 2 <= cCount);
         pDC->PolyBezierTo(&lppt[nIndex], 3);
         nIndex += 2;
         if (pptLastMoveTo != NULL)
            pDC->LineTo(pptLastMoveTo->x, pptLastMoveTo->y);
         pptLastMoveTo = NULL;
         break;

      case PT_BEZIERTO:
         ASSERT(nIndex + 2 <= cCount);
         pDC->PolyBezierTo(&lppt[nIndex], 3);
```

```
            nIndex += 2;
            break;
         }
      }

      // If the figure was never closed and should be,
      // close it now.
      if (pptLastMoveTo != NULL && nIndex > 1)
         pDC->LineTo(pptLastMoveTo->x, pptLastMoveTo->y);
   }
```

See Also

Reference

CDC Class
Hierarchy Chart
CDC::EndPath
CDC::FillPath
CRgn::CreateFromPath
CDC::SelectClipPath
CDC::StrokeAndFillPath
CDC::StrokePath
CDC::WidenPath

Other Resources

CDC Members

CDC::BitBlt

Copies a bitmap from the source device context to this current device context.

```
BOOL BitBlt(
   int x,
   int y,
   int nWidth,
   int nHeight,
   CDC* pSrcDC,
   int xSrc,
   int ySrc,
   DWORD dwRop
);
```

Parameters

x

Specifies the logical x-coordinate of the upper-left corner of the destination rectangle.

y

Specifies the logical y-coordinate of the upper-left corner of the destination rectangle.

nWidth

Specifies the width (in logical units) of the destination rectangle and source bitmap.

nHeight

Specifies the height (in logical units) of the destination rectangle and source bitmap.

pSrcDC

Pointer to a **CDC** object that identifies the device context from which the bitmap will be copied. It must be **NULL** if *dwRop* specifies a raster operation that does not include a source.

xSrc

Specifies the logical x-coordinate of the upper-left corner of the source bitmap.

ySrc

Specifies the logical y-coordinate of the upper-left corner of the source bitmap.

dwRop

Specifies the raster operation to be performed. Raster-operation codes define how the GDI combines colors in output operations that involve a current brush, a possible source bitmap, and a destination bitmap. See BitBlt in the *Platform SDK* for a list of the raster-operation codes for *dwRop* and their descriptions

For a complete list of raster-operation codes, see About Raster Operation Codes in the *Platform SDK*.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The application can align the windows or client areas on byte boundaries to ensure that the **BitBlt** operations occur on byte-aligned rectangles. (Set the **CS_BYTEALIGNWINDOW** or **CS_BYTEALIGNCLIENT** flags when you register the window classes.)

BitBlt operations on byte-aligned rectangles are considerably faster than **BitBlt** operations on rectangles that are not byte aligned. If you want to specify class styles such as byte-alignment for your own device context, you will have to register a window class rather than relying on the Microsoft Foundation classes to do it for you. Use the global function AfxRegisterWndClass.

GDI transforms *nWidth* and *nHeight*, once by using the destination device context, and once by using the source device context. If the resulting extents do not match, GDI uses the Windows **StretchBlt** function to compress or stretch the source bitmap as

necessary.

If destination, source, and pattern bitmaps do not have the same color format, the **BitBlt** function converts the source and pattern bitmaps to match the destination. The foreground and background colors of the destination bitmap are used in the conversion.

When the **BitBlt** function converts a monochrome bitmap to color, it sets white bits (1) to the background color and black bits (0) to the foreground color. The foreground and background colors of the destination device context are used. To convert color to monochrome, **BitBlt** sets pixels that match the background color to white and sets all other pixels to black. **BitBlt** uses the foreground and background colors of the color device context to convert from color to monochrome.

Note that not all device contexts support **BitBlt**. To check whether a given device context does support **BitBlt**, use the **GetDeviceCaps** member function and specify the **RASTERCAPS** index.

Example

See the example for CDC::CreateCompatibleDC.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetDeviceCaps
CDC::PatBlt
CDC::SetTextColor
CDC::StretchBlt
StretchDIBits
BitBlt

Other Resources

CDC Members

CDC::CDC

Constructs a **CDC** object.

```
CDC( );
```

See Also

Reference

CDC Class
Hierarchy Chart
CDC::CreateDC
CDC::CreateIC
CDC::CreateCompatibleDC

Other Resources

CDC Members

CDC::Chord

Draws a chord (a closed figure bounded by the intersection of an ellipse and a line segment).

```
BOOL Chord(
   int x1,
   int y1,
   int x2,
   int y2,
   int x3,
   int y3,
   int x4,
   int y4
);
BOOL Chord(
   LPCRECT lpRect,
   POINT ptStart,
   POINT ptEnd
);
```

Parameters

x1

Specifies the x-coordinate of the upper-left corner of the chord's bounding rectangle (in logical units).

y1

Specifies the y-coordinate of the upper-left corner of the chord's bounding rectangle (in logical units).

x2

Specifies the x-coordinate of the lower-right corner of the chord's bounding rectangle (in logical units).

y2

Specifies the y-coordinate of the lower-right corner of the chord's bounding rectangle (in logical units).

x3

Specifies the x-coordinate of the point that defines the chord's starting point (in logical units).

y3

Specifies the y-coordinate of the point that defines the chord's starting point (in logical units).

x4

Specifies the x-coordinate of the point that defines the chord's endpoint (in logical units).

y4

Specifies the y-coordinate of the point that defines the chord's endpoint (in logical units).

lpRect

Specifies the bounding rectangle (in logical units). You can pass either a **LPRECT** or a CRect object for this parameter.

ptStart

Specifies the x- and y-coordinates of the point that defines the chord's starting point (in logical units). This point does not have to lie exactly on the chord. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

ptEnd

Specifies the x- and y-coordinates of the point that defines the chord's ending point (in logical units). This point does not have to lie exactly on the chord. You can pass either a POINT structure or a CPoint object for this parameter.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The (*x1*, *y1*) and (*x2*, *y2*) parameters specify the upper-left and lower-right corners, respectively, of a rectangle bounding the ellipse that is part of the chord. The (*x3*, *y3*) and (*x4*, *y4*) parameters specify the endpoints of a line that intersects the ellipse. The chord is drawn by using the selected pen and filled by using the selected brush.

The figure drawn by the **Chord** function extends up to, but does not include the right and bottom coordinates. This means that the height of the figure is *y2* – *y1* and the width of the figure is *x2* – *x1*.

Example

```
void CCurvesView::OnDraw(CDC* pDC)
{
   // Fill the client area with a circle. The circle is
   // blue and filled with blue, but has a chord cut out
   // of it from 3 o'clock to 6 o'clock. That chord is
   // red and filled with a red diagonal hatch.

   // Get the client area.
   CRect rectClient;
   GetClientRect(rectClient);

   // Make a couple of pens and similar brushes.
   CPen penBlue, penRed;
   CBrush brushBlue, brushRed;
   CBrush* pOldBrush;
   CPen* pOldPen;

   brushBlue.CreateSolidBrush(RGB(0, 0, 255));
   brushRed.CreateHatchBrush(HS_FDIAGONAL, RGB(255, 0, 0));
   penBlue.CreatePen(PS_SOLID | PS_COSMETIC, 1, RGB(0, 0, 255));
   penRed.CreatePen(PS_SOLID | PS_COSMETIC, 1, RGB(255, 0, 0));

   // Draw from 3 o'clock to 6 o'clock, counterclockwise,
   // in a blue pen with a solid blue fill.
   pOldPen = pDC->SelectObject(&penBlue);
   pOldBrush = pDC->SelectObject(&brushBlue);

   pDC->Chord(rectClient,
      CPoint(rectClient.right, rectClient.CenterPoint().y),
      CPoint(rectClient.CenterPoint().x, rectClient.right));

   // Draw the remaining quarter chord from 6 o'clock
   // to 3 o'clock, counterclockwise, in a red pen
   // with the hatched brush.
   pDC->SelectObject(&penRed);
   pDC->SelectObject(&brushRed);

   // Keep the same parameters, but reverse start and
   // end points.
   pDC->Chord(rectClient,
      CPoint(rectClient.CenterPoint().x, rectClient.right),
      CPoint(rectClient.right, rectClient.CenterPoint().y));

   // Restore the previous pen.
   pDC->SelectObject(pOldPen);
}
```

See Also

Reference

CDC Class
Hierarchy Chart
CDC::Arc
Chord
POINT Structure

Other Resources

CDC Members

CDC::CloseFigure

Closes an open figure in a path.

```
BOOL CloseFigure( );
```

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The function closes the figure by drawing a line from the current position to the first point of the figure (usually, the point specified by the most recent call to the **MoveTo** member function) and connects the lines by using the line join style. If a figure is closed by using the **LineTo** member function instead of **CloseFigure**, end caps are used to create the corner instead of a join. **CloseFigure** should only be called if there is an open path bracket in the device context.

A figure in a path is open unless it is explicitly closed by using this function. (A figure can be open even if the current point and the starting point of the figure are the same.) Any line or curve added to the path after **CloseFigure** starts a new figure.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::BeginPath
CDC::EndPath
CDC::MoveTo
CloseFigure

Other Resources

CDC Members

CDC::CreateCompatibleDC

Creates a memory device context that is compatible with the device specified by *pDC*.

```
BOOL CreateCompatibleDC(
   CDC* pDC
);
```

Parameters

pDC

A pointer to a device context. If *pDC* is **NULL**, the function creates a memory device context that is compatible with the system display.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

A memory device context is a block of memory that represents a display surface. It can be used to prepare images in memory before copying them to the actual device surface of the compatible device.

When a memory device context is created, GDI automatically selects a 1-by-1 monochrome stock bitmap for it. GDI output functions can be used with a memory device context only if a bitmap has been created and selected into that context.

This function can only be used to create compatible device contexts for devices that support raster operations. See the CDC::BitBlt member function for information regarding bit-block transfers between device contexts. To determine whether a device context supports raster operations, see the **RC_BITBLT** raster capability in the member function **CDC::GetDeviceCaps**.

Example

```
// This OnDraw() handler loads a bitmap from system resources,
// centers it in the view, and uses BitBlt() to paint the bitmap
// bits.

void CBlat2View::OnDraw(CDC* pDC)
{
   CBlat2Doc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   // load IDB_BITMAP1 from our resources
   CBitmap bmp;
   if (bmp.LoadBitmap(IDB_BITMAP1))
   {
      // Get the size of the bitmap
      BITMAP bmpInfo;
      bmp.GetBitmap(&bmpInfo);

      // Create an in-memory DC compatible with the
      // display DC we're using to paint
      CDC dcMemory;
      dcMemory.CreateCompatibleDC(pDC);

      // Select the bitmap into the in-memory DC
      CBitmap* pOldBitmap = dcMemory.SelectObject(&bmp);

      // Find a centerpoint for the bitmap in the client area
      CRect rect;
      GetClientRect(&rect);
      int nX = rect.left + (rect.Width() - bmpInfo.bmWidth) / 2;
      int nY = rect.top + (rect.Height() - bmpInfo.bmHeight) / 2;

      // Copy the bits from the in-memory DC into the on-
      // screen DC to actually do the painting. Use the centerpoint
```

```
        // we computed for the target offset.
        pDC->BitBlt(nX, nY, bmpInfo.bmWidth, bmpInfo.bmHeight, &dcMemory,
           0, 0, SRCCOPY);

        dcMemory.SelectObject(pOldBitmap);
     }
     else
        TRACE0("ERROR: Where's IDB_BITMAP1?\n");
  }
```

See Also

Reference

CDC Class
Hierarchy Chart
CDC::CDC
CDC::GetDeviceCaps
CreateCompatibleDC
CDC::BitBlt
CDC::CreateDC
CDC::CreateIC
CDC::DeleteDC

Other Resources

CDC Members

CDC::CreateDC

Creates a device context for the specified device.

```
BOOL CreateDC(
   LPCTSTR lpszDriverName,
   LPCTSTR lpszDeviceName,
   LPCTSTR lpszOutput,
   const void* lpInitData
);
```

Parameters

lpszDriverName

Points to a null-terminated string that specifies the filename (without extension) of the device driver (for example, "EPSON"). You can also pass a **CString** object for this parameter.

lpszDeviceName

Points to a null-terminated string that specifies the name of the specific device to be supported (for example, "EPSON FX-80"). The *lpszDeviceName* parameter is used if the module supports more than one device. You can also pass a **CString** object for this parameter.

lpszOutput

Points to a null-terminated string that specifies the file or device name for the physical output medium (file or output port). You can also pass a **CString** object for this parameter.

lpInitData

Points to a **DEVMODE** structure containing device-specific initialization data for the device driver. The Windows **DocumentProperties** function retrieves this structure filled in for a given device. The *lpInitData* parameter must be **NULL** if the device driver is to use the default initialization (if any) specified by the user through the Control Panel.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The PRINT.H header file is required if the DEVMODE structure is used.

Device names follow these conventions: an ending colon (:) is recommended, but optional. Windows strips the terminating colon so that a device name ending with a colon is mapped to the same port as the same name without a colon. The driver and port names must not contain leading or trailing spaces. GDI output functions cannot be used with information contexts.

See Also

Reference

CDC Class
Hierarchy Chart
DocumentProperties
CreateDC
CDC::DeleteDC
CDC::CreateIC

Other Resources

CDC Members

CDC::CreateIC

Creates an information context for the specified device.

```
BOOL CreateIC(
   LPCTSTR lpszDriverName,
   LPCTSTR lpszDeviceName,
   LPCTSTR lpszOutput,
   const void* lpInitData
);
```

Parameters

lpszDriverName

Points to a null-terminated string that specifies the filename (without extension) of the device driver (for example, "EPSON"). You can pass a **CString** object for this parameter.

lpszDeviceName

Points to a null-terminated string that specifies the name of the specific device to be supported (for example, "EPSON FX-80"). The *lpszDeviceName* parameter is used if the module supports more than one device. You can pass a **CString** object for this parameter.

lpszOutput

Points to a null-terminated string that specifies the file or device name for the physical output medium (file or port). You can pass a **CString** object for this parameter.

lpInitData

Points to device-specific initialization data for the device driver. The *lpInitData* parameter must be **NULL** if the device driver is to use the default initialization (if any) specified by the user through the Control Panel. See **CreateDC** for the data format for device-specific initialization.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The information context provides a fast way to get information about the device without creating a device context.

Device names follow these conventions: an ending colon (:) is recommended, but optional. Windows strips the terminating colon so that a device name ending with a colon is mapped to the same port as the same name without a colon. The driver and port names must not contain leading or trailing spaces. GDI output functions cannot be used with information contexts.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::CreateDC
CreateIC
CDC::DeleteDC

Other Resources

CDC Members

CDC::DeleteDC

In general, do not call this function; the destructor will do it for you.

```
BOOL DeleteDC( );
```

Return Value

Nonzero if the function completed successfully; otherwise 0.

Remarks

The **DeleteDC** member function deletes the Windows device contexts that are associated with **m_hDC** in the current **CDC** object. If this **CDC** object is the last active device context for a given device, the device is notified and all storage and system resources used by the device are released.

An application should not call **DeleteDC** if objects have been selected into the device context. Objects must first be selected out of the device context before it is deleted.

An application must not delete a device context whose handle was obtained by calling CWnd::GetDC. Instead, it must call CWnd::ReleaseDC to free the device context. The CClientDC and CWindowDC classes are provided to wrap this functionality.

The **DeleteDC** function is generally used to delete device contexts created with CreateDC, CreateIC, or CreateCompatibleDC.

Example

See the example for CPrintDialog::GetPrinterDC.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::CDC
DeleteDC
CDC::CreateDC
CDC::CreateIC
CDC::CreateCompatibleDC
CWnd::GetDC
CWnd::ReleaseDC

Other Resources

CDC Members

CDC::DeleteTempMap

Called automatically by the **CWinApp** idle-time handler, **DeleteTempMap** deletes any temporary **CDC** objects created by **FromHandle**, but does not destroy the device context handles (**hDC**s) temporarily associated with the **CDC** objects.

```
static void PASCAL DeleteTempMap( );
```

See Also

Reference

CDC Class
Hierarchy Chart
CDC::Detach
CDC::FromHandle
CWinApp::OnIdle

Other Resources

CDC Members

CDC::Detach

Call this function to detach **m_hDC** (the output device context) from the **CDC** object and set both **m_hDC** and **m_hAttribDC** to **NULL**.

```
HDC Detach( );
```

Return Value

A Windows device context.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::Attach
CDC::m_hDC
CDC::m_hAttribDC

Other Resources

CDC Members

CDC::DPtoHIMETRIC

Use this function when you give **HIMETRIC** sizes to OLE, converting pixels to **HIMETRIC**.

```
void DPtoHIMETRIC(
   LPSIZE lpSize
) const;
```

Parameters

lpSize

Points to a SIZE structure or CSize object.

Remarks

If the mapping mode of the device context object is **MM_LOENGLISH**, **MM_HIENGLISH**, **MM_LOMETRIC**, or **MM_HIMETRIC**, then the conversion is based on the number of pixels in the physical inch. If the mapping mode is one of the other non-constrained modes (e.g., **MM_TEXT**), then the conversion is based on the number of pixels in the logical inch.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::DPtoLP
CDC::LPtoDP
CDC::HIMETRICtoLP
CDC::HIMETRICtoDP
CDC::LPtoHIMETRIC

Other Resources

CDC Members

CDC::DPtoLP

Converts device units into logical units.

```
void DPtoLP(
   LPPOINT lpPoints,
   int nCount = 1
) const;
void DPtoLP(
   LPRECT lpRect
) const;
void DPtoLP(
   LPSIZE lpSize
) const;
```

Parameters

lpPoints

Points to an array of POINT structures or CPoint objects.

nCount

The number of points in the array.

lpRect

Points to a RECT structure or CRect object. This parameter is used for the simple case of converting one rectangle from device points to logical points.

lpSize

Points to a SIZE structure or CSize object.

Remarks

The function maps the coordinates of each point, or dimension of a size, from the device coordinate system into GDI's logical coordinate system. The conversion depends on the current mapping mode and the settings of the origins and extents for the device's window and viewport.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::LPtoDP
CDC::HIMETRICtoDP
DPtoLP
POINT Structure
RECT Structure
CDC::GetWindowExt
CDC::GetWindowOrg

Other Resources

CDC Members

CDC::Draw3dRect

Call this member function to draw a three-dimensional rectangle.

```
void Draw3dRect(
   LPCRECT lpRect,
   COLORREF clrTopLeft,
   COLORREF clrBottomRight
);
void Draw3dRect(
   int x,
   int y,
   int cx,
   int cy,
   COLORREF clrTopLeft,
   COLORREF clrBottomRight
);
```

Parameters

lpRect

Specifies the bounding rectangle (in logical units). You can pass either a pointer to a RECT structure or a CRect object for this parameter.

clrTopLeft

Specifies the color of the top and left sides of the three-dimensional rectangle.

clrBottomRight

Specifies the color of the bottom and right sides of the three-dimensional rectangle.

x

Specifies the logical x-coordinate of the upper-left corner of the three-dimensional rectangle.

y

Specifies the logical y-coordinate of the upper-left corner of the three-dimensional rectangle.

cx

Specifies the width of the three-dimensional rectangle.

cy

Specifies the height of the three-dimensional rectangle.

Remarks

The rectangle will be drawn with the top and left sides in the color specified by *clrTopLeft* and the bottom and right sides in the color specified by *clrBottomRight*.

Example

```
void CMyView::OnDraw(CDC* pDC)
{
   // get the client area
   CRect rect;
   GetClientRect(rect);

   // shrink our rect 20 pixels on all sides
   rect.DeflateRect(20, 20);

   // draw a rectangle with red top and left sides, and
   // green right and bottom sides.
   pDC->Draw3dRect(rect, RGB(255, 0, 0), RGB(0, 255, 0));
```

```
    // This call to the four-integer override would draw
    // the same rectangle with a little less convenience:

    // pDC->Draw3dRect(rect.left, rect.top, rect.Width(), rect.Height(),
    //    RGB(255, 0, 0), RGB(0, 255, 0));
}
```

See Also

Reference

CDC Class

Hierarchy Chart

RECT Structure

CRect Class

Other Resources

CDC Members

CDC::DrawDragRect

Call this member function repeatedly to redraw a drag rectangle.

```
void DrawDragRect(
   LPCRECT lpRect,
   SIZE size,
   LPCRECT lpRectLast,
   SIZE sizeLast,
   CBrush* pBrush = NULL,
   CBrush* pBrushLast = NULL
);
```

Parameters

lpRect

Points to a RECT structure or a CRect object that specifies the logical coordinates of a rectangle — in this case, the end position of the rectangle being redrawn.

size

Specifies the displacement from the top-left corner of the outer border to the top-left corner of the inner border (that is, the thickness of the border) of a rectangle.

lpRectLast

Points to a RECT structure or a CRect object that specifies the logical coordinates of the position of a rectangle — in this case, the original position of the rectangle being redrawn.

sizeLast

Specifies the displacement from the top-left corner of the outer border to the top-left corner of the inner border (that is, the thickness of the border) of the original rectangle being redrawn.

pBrush

Pointer to a brush object. Set to **NULL** to use the default halftone brush.

pBrushLast

Pointer to the last brush object used. Set to **NULL** to use the default halftone brush.

Remarks

Call it in a loop as you sample mouse position, in order to give visual feedback. When you call **DrawDragRect**, the previous rectangle is erased and a new one is drawn. For example, as the user drags a rectangle across the screen, **DrawDragRect** will erase the original rectangle and redraw a new one in its new position. By default, **DrawDragRect** draws the rectangle by using a halftone brush to eliminate flicker and to create the appearance of a smoothly moving rectangle.

The first time you call **DrawDragRect**, the *lpRectLast* parameter should be **NULL**.

See Also

Reference

CDC Class
Hierarchy Chart
RECT Structure
CRect Class
CDC::GetHalftoneBrush

Other Resources

CDC Members

CDC::DrawEdge

Call this member function to draw the edges of a rectangle of the specified type and style.

```
BOOL DrawEdge(
   LPRECT lpRect,
   UINT nEdge,
   UINT nFlags
);
```

Parameters

lpRect

A pointer to a **RECT** structure that contains the logical coordinates of the rectangle.

nEdge

Specifies the type of inner and outer edge to draw. This parameter must be a combination of one inner-border flag and one outer-border flag. See DrawEdge in the *Platform SDK* for a table of the parameter's types.

nFlags

The flags that specify the type of border to be drawn. See **DrawEdge** in the *Platform SDK* for a table of the parameter's values. For diagonal lines, the **BF_RECT** flags specify the end point of the vector bounded by the rectangle parameter.

Return Value

Nonzero if successful; otherwise 0.

See Also

Reference

CDC Class
Hierarchy Chart
DrawEdge

Other Resources

CDC Members

CDC::DrawEscape

Accesses drawing capabilities of a video display that are not directly available through the graphics device interface (GDI).

```
int DrawEscape(
   int nEscape,
   int nInputSize,
   LPCSTR lpszInputData
);
```

Parameters

nEscape

Specifies the escape function to be performed.

nInputSize

Specifies the number of bytes of data pointed to by the *lpszInputData* parameter.

lpszInputData

Points to the input structure required for the specified escape.

Return Value

Specifies the outcome of the function. Greater than zero if successful, except for the **QUERYESCSUPPORT** draw escape, which checks for implementation only; or zero if the escape is not implemented; or less than zero if an error occurred.

Remarks

When an application calls **DrawEscape**, the data identified by *nInputSize* and *lpszInputData* is passed directly to the specified display driver.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::Escape
DrawEscape

Other Resources

CDC Members

CDC::DrawFocusRect

Draws a rectangle in the style used to indicate that the rectangle has the focus.

```
void DrawFocusRect(
   LPCRECT lpRect
);
```

Parameters

lpRect

Points to a RECT structure or a CRect object that specifies the logical coordinates of the rectangle to be drawn.

Remarks

Since this is a Boolean XOR function, calling this function a second time with the same rectangle removes the rectangle from the display. The rectangle drawn by this function cannot be scrolled. To scroll an area containing a rectangle drawn by this function, first call **DrawFocusRect** to remove the rectangle from the display, then scroll the area, and then call **DrawFocusRect** again to draw the rectangle in the new position.

Caution
DrawFocusRect works only in **MM_TEXT** mode. In other modes, this function does not draw the focus rectangle correctly, but it does not return error values.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::FrameRect
DrawFocusRect
RECT Structure

Other Resources

CDC Members

CDC::DrawFrameControl

Call this member function to draw a frame control of the specified type and style.

```
BOOL DrawFrameControl(
   LPRECT lpRect,
   UINT nType,
   UINT nState
);
```

Parameters

lpRect

A pointer to a **RECT** structure that contains the logical coordinates of the rectangle.

nType

Specifies the type of frame control to draw. See the *uType* parameter in DrawFrameControl in the *Platform SDK* for a list of this parameter's possible values.

nState

Specifies the initial state of the frame control. Can be one or more of the values described for the *uState* parameter in **DrawFrameControl** in the *Platform SDK*. Use the *nState* value **DFCS_ADJUSTRECT** to adjust the bounding rectangle to exclude the surrounding edge of the push button.

Return Value

Nonzero if successful; otherwise 0.

Remarks

In several cases, *nState* depends on the *nType* parameter. The following list shows the relationship between the four *nType* values and *nState*:

- **DFC_BUTTON**
  - **DFCS_BUTTON3STATE** Three-state button
  - **DFCS_BUTTONCHECK** Check box
  - **DFCS_BUTTONPUSH** Push button
  - **DFCS_BUTTONRADIO** Radio button
  - **DFCS_BUTTONRADIOIMAGE** Image for radio button (nonsquare needs image)
  - **DFCS_BUTTONRADIOMASK** Mask for radio button (nonsquare needs mask)
- **DFC_CAPTION**
  - **DFCS_CAPTIONCLOSE** Close button
  - **DFCS_CAPTIONHELP** Help button
  - **DFCS_CAPTIONMAX** Maximize button
  - **DFCS_CAPTIONMIN** Minimize button
  - **DFCS_CAPTIONRESTORE** Restore button
- **DFC_MENU**
  - **DFCS_MENUARROW** Submenu arrow
  - **DFCS_MENUBULLET** Bullet
  - **DFCS_MENUCHECK** Check mark

- **DFC_SCROLL**
    - **DFCS_SCROLLCOMBOBOX** Combo box scroll bar
    - **DFCS_SCROLLDOWN** Down arrow of scroll bar
    - **DFCS_SCROLLLEFT** Left arrow of scroll bar
    - **DFCS_SCROLLRIGHT** Right arrow of scroll bar
    - **DFCS_SCROLLSIZEGRIP** Size grip in bottom-right corner of window
    - **DFCS_SCROLLUP** Up arrow of scroll bar

Example

This code draws the size gripper in the bottom-right corner of your window. It's appropriate for the **OnPaint** handler of a dialog box, which has no styles and normally doesn't contain other controls (like a status bar) that may give it a size gripper.

```
CRect rc;
GetClientRect(&rc);

rc.left = rc.right - ::GetSystemMetrics(SM_CXHSCROLL);
rc.top = rc.bottom - ::GetSystemMetrics(SM_CYVSCROLL);

dc.DrawFrameControl(rc, DFC_SCROLL, DFCS_SCROLLSIZEGRIP);
```

See Also

Reference

CDC Class

Hierarchy Chart

DrawFrameControl

Other Resources

CDC Members

CDC::DrawIcon

Draws an icon on the device represented by the current **CDC** object.

```
BOOL DrawIcon(
   int x,
   int y,
   HICON hIcon
);
BOOL DrawIcon(
   POINT point,
   HICON hIcon
);
```

Parameters

x

Specifies the logical x-coordinate of the upper-left corner of the icon.

y

Specifies the logical y-coordinate of the upper-left corner of the icon.

hIcon

Identifies the handle of the icon to be drawn.

point

Specifies the logical x- and y-coordinates of the upper-left corner of the icon. You can pass a POINT structure or a CPoint object for this parameter.

Return Value

Nonzero if the function completed successfully; otherwise 0.

Remarks

The function places the icon's upper-left corner at the location specified by *x* and *y*. The location is subject to the current mapping mode of the device context.

The icon resource must have been previously loaded by using the functions **CWinApp::LoadIcon**, **CWinApp::LoadStandardIcon**, or **CWinApp::LoadOEMIcon**. The **MM_TEXT** mapping mode must be selected prior to using this function.

Example

See the example for CWnd::IsIconic.

See Also

Reference

CDC Class
Hierarchy Chart
CWinApp::LoadIcon
CWinApp::LoadStandardIcon
CWinApp::LoadOEMIcon
CDC::GetMapMode
CDC::SetMapMode
DrawIcon
POINT Structure

Other Resources

CDC Members

CDC::DrawState

Call this member function to display an image and apply a visual effect to indicate a state, such as a disabled or default state.

Note
For all *nFlag* states except **DSS_NORMAL**, the image is converted to monochrome before the visual effect is applied.

```
BOOL DrawState(
   CPoint pt,
   CSize size,
   HBITMAP hBitmap,
   UINT nFlags,
   HBRUSH hBrush = NULL
);
BOOL DrawState(
   CPoint pt,
   CSize size,
   CBitmap* pBitmap,
   UINT nFlags,
   CBrush* pBrush = NULL
);
BOOL DrawState(
   CPoint pt,
   CSize size,
   HICON hIcon,
   UINT nFlags,
   HBRUSH hBrush = NULL
);
BOOL DrawState(
   CPoint pt,
   CSize size,
   HICON hIcon,
   UINT nFlags,
   CBrush* pBrush = NULL
);
BOOL DrawState(
   CPoint pt,
   CSize size,
   LPCTSTR lpszText,
   UINT nFlags,
   BOOL bPrefixText = TRUE,
   int nTextLen = 0,
   HBRUSH hBrush = NULL
);
BOOL DrawState(
   CPoint pt,
   CSize size,
   LPCTSTR lpszText,
   UINT nFlags,
   BOOL bPrefixText = TRUE,
   int nTextLen = 0,
   CBrush* pBrush = NULL
);
BOOL DrawState(
   CPoint pt,
   CSize size,
   DRAWSTATEPROC lpDrawProc,
   LPARAM lData,
   UINT nFlags,
   HBRUSH hBrush = NULL
);
BOOL DrawState(
   CPoint pt,
```

```
    CSize size,
    DRAWSTATEPROC lpDrawProc,
    LPARAM lData,
    UINT nFlags,
    CBrush* pBrush = NULL
  );
```

Parameters

pt

Specifies the location of the image.

size

Specifies the size of the image.

hBitmap

A handle to a bitmap.

nFlags

Flags that specify the image type and state. See DrawState in the *Platform SDK* for the possible *nFlags* types and states.

hBrush

A handle to a brush.

pBitmap

A pointer to a CBitmap object.

pBrush

A pointer to a CBrush object.

hIcon

A handle to an icon.

lpszText

A pointer to text.

bPrefixText

Text that may contain an accelerator mnemonic. The *lData* parameter specifies the address of the string, and the *nTextLen* parameter specifies the length. If *nTextLen* is 0, the string is assumed to be null-terminated.

nTextLen

Length of the text string pointed to by *lpszText*. If *nTextLen* is 0, the string is assumed to be null-terminated.

lpDrawProc

A pointer to a callback function used to render an image. This parameter is required if the image type in *nFlags* is **DST_COMPLEX**. It is optional and can be **NULL** if the image type is **DST_TEXT**. For all other image types, this parameter is ignored. For more information about the callback function, see the DrawStateProc function in the *Platform SDK*.

lData

Specifies information about the image. The meaning of this parameter depends on the image type.

Return Value

Nonzero if successful; otherwise 0.

See Also

Reference

CDC Class
Hierarchy Chart
DrawState
DrawStateProc

Other Resources

CDC::DrawText

Call this member function to format text in the given rectangle. To specify additional formatting options, use CDC::DrawTextEx.

```
virtual int DrawText(
   LPCTSTR lpszString,
   int nCount,
   LPRECT lpRect,
   UINT nFormat
);
int DrawText(
   const CString& str,
   LPRECT lpRect,
   UINT nFormat
);
```

Parameters

lpszString

Points to the string to be drawn. If *nCount* is –1, the string must be null-terminated.

nCount

Specifies the number of chars in the string. If *nCount* is –1, then *lpszString* is assumed to be a long pointer to a null-terminated string and **DrawText** computes the character count automatically.

lpRect

Points to a RECT structure or CRect object that contains the rectangle (in logical coordinates) in which the text is to be formatted.

str

A CString object that contains the specified characters to be drawn.

nFormat

Specifies the method of formatting the text. It can be any combination of the values described for the *uFormat* parameter in DrawText in the *Platform SDK*. (combine using the bitwise OR operator):

Note
Some *uFormat* flag combinations can cause the passed string to be modified. Using **DT_MODIFYSTRING** with either **DT_END_ELLIPSIS** or **DT_PATH_ELLIPSIS** may cause the string to be modified, causing an assertion in the **CString** override. The values **DT_CALCRECT**, **DT_EXTERNALLEADING**, **DT_INTERNAL**, **DT_NOCLIP**, and **DT_NOPREFIX** cannot be used with the **DT_TABSTOP** value.

Return Value

The height of the text if the function is successful.

Remarks

It formats text by expanding tabs into appropriate spaces, aligning text to the left, right, or center of the given rectangle, and breaking text into lines that fit within the given rectangle. The type of formatting is specified by *nFormat*.

This member function uses the device context's selected font, text color, and background color to draw the text. Unless the **DT_NOCLIP** format is used, **DrawText** clips the text so that the text does not appear outside the given rectangle. All formatting is assumed to have multiple lines unless the **DT_SINGLELINE** format is given.

If the selected font is too large for the specified rectangle, the **DrawText** member function does not attempt to substitute a smaller font.

If the **DT_CALCRECT** flag is specified, the rectangle specified by *lpRect* will be updated to reflect the width and height needed to draw the text.

If the **TA_UPDATECP** text-alignment flag has been set (see CDC::SetTextAlign), **DrawText** will display text starting at the current position, rather than at the left of the given rectangle. **DrawText** will not wrap text when the **TA_UPDATECP** flag has been set (that is, the **DT_WORDBREAK** flag will have no effect).

The text color may be set by CDC::SetTextColor.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetTextColor
CDC::ExtTextOut
CDC::TabbedTextOut
CDC::TextOut
DrawText
RECT Structure
CDC::SetTextAlign

Other Resources

CDC Members

CDC::DrawTextEx

Formats text in the given rectangle.

```
virtual int DrawTextEx(
   LPTSTR lpszString,
   int nCount,
   LPRECT lpRect,
   UINT nFormat,
      LPDRAWTEXTPARAMS lpDTParams
);
int DrawTextEx(
   const CString& str,
   LPRECT lpRect,
   UINT nFormat,
      LPDRAWTEXTPARAMS lpDTParams
);
```

Parameters

lpszString

Points to the string to be drawn. If *nCount* is –1, the string must be null terminated.

nCount

Specifies the number of chars in the string. If *nCount* is –1, then *lpszString* is assumed to be a long pointer to a null-terminated string and **DrawText** computes the character count automatically.

lpRect

Points to a RECT structure or CRect object that contains the rectangle (in logical coordinates) in which the text is to be formatted.

str

A CString object that contains the specified characters to be drawn.

nFormat

Specifies the method of formatting the text. It can be any combination of the values described for the *uFormat* parameter in DrawText in the *Platform SDK*. (Combine using the bitwise **OR** operator):

Note
Some *uFormat* flag combinations can cause the passed string to be modified. Using **DT_MODIFYSTRING** with either **DT_END_ELLIPSIS** or **DT_PATH_ELLIPSIS** may cause the string to be modified, causing an assertion in the **CString** override. The values **DT_CALCRECT**, **DT_EXTERNALLEADING**, **DT_INTERNAL**, **DT_NOCLIP**, and **DT_NOPREFIX** cannot be used with the **DT_TABSTOP** value.

lpDTParams

Pointer to a DRAWTEXTPARAMS structure that specifies additional formatting options. This parameter can be **NULL**.

Remarks

It formats text by expanding tabs into appropriate spaces, aligning text to the left, right, or center of the given rectangle, and breaking text into lines that fit within the given rectangle. The type of formatting is specified by *nFormat* and *lpDTParams*. For more information, see CDC::DrawText and DrawTextEx in the *Platform SDK*.

The text color may be set by CDC::SetTextColor.

See Also

Reference

CDC Class

Hierarchy Chart

CDC::DrawText
CDC::ExtTextOut
CDC::TabbedTextOut
CDC::TextOut
DrawText
RECT Structure
CDC::SetTextAlign

Other Resources

CDC Members

CDC::Ellipse

Draws an ellipse.

```
BOOL Ellipse(
   int x1,
   int y1,
   int x2,
   int y2
);
BOOL Ellipse(
   LPCRECT lpRect
);
```

Parameters

x1

Specifies the logical x-coordinate of the upper-left corner of the ellipse's bounding rectangle.

y1

Specifies the logical y-coordinate of the upper-left corner of the ellipse's bounding rectangle.

x2

Specifies the logical x-coordinate of the lower-right corner of the ellipse's bounding rectangle.

y2

Specifies the logical y-coordinate of the lower-right corner of the ellipse's bounding rectangle.

lpRect

Specifies the ellipse's bounding rectangle. You can also pass a CRect object for this parameter.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The center of the ellipse is the center of the bounding rectangle specified by *x1*, *y1*, *x2*, and *y2*, or *lpRect*. The ellipse is drawn with the current pen, and its interior is filled with the current brush.

The figure drawn by this function extends up to, but does not include, the right and bottom coordinates. This means that the height of the figure is *y2* – *y1* and the width of the figure is *x2* – *x1*.

If either the width or the height of the bounding rectangle is 0, no ellipse is drawn.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::Arc
CDC::Chord
Ellipse

Other Resources

CDC Members

CDC::EndDoc

Ends a print job started by a call to the StartDoc member function.

```
int EndDoc( );
```

Return Value

Greater than or equal to 0 if the function is successful, or a negative value if an error occurred.

Remarks

This member function replaces the **ENDDOC** printer escape, and should be called immediately after finishing a successful print job.

If an application encounters a printing error or a canceled print operation, it must not attempt to terminate the operation by using either **EndDoc** or AbortDoc. GDI automatically terminates the operation before returning the error value.

This function should not be used inside metafiles.

Example

See the example for CDC::StartDoc.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::AbortDoc
CDC::Escape
CDC::StartDoc

Other Resources

CDC Members

CDC::EndPage

Informs the device that the application has finished writing to a page.

```
int EndPage( );
```

Return Value

Greater than or equal to 0 if the function is successful, or a negative value if an error occurred.

Remarks

This member function is typically used to direct the device driver to advance to a new page.

This member function replaces the **NEWFRAME** printer escape. Unlike **NEWFRAME**, this function is always called after printing a page.

Example

See the example for CDC::StartDoc.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::StartPage
CDC::StartDoc
CDC::Escape

Other Resources

CDC Members

CDC::EndPath

Closes a path bracket and selects the path defined by the bracket into the device context.

```
BOOL EndPath( );
```

Return Value

Nonzero if the function is successful; otherwise 0.

Example

See the example for CDC::BeginPath.

See Also

Reference

CDC Class

Hierarchy Chart

CDC::BeginPath

Other Resources

CDC Members

CDC::EnumObjects

Enumerates the pens and brushes available in a device context.

```
int EnumObjects(
   int nObjectType,
   int ( CALLBACK * lpfn )( LPVOID, LPARAM ),
   LPARAM lpData
);
```

Parameters

nObjectType

Specifies the object type. It can have the values **OBJ_BRUSH** or **OBJ_PEN**.

lpfn

Is the procedure-instance address of the application-supplied callback function. See the "Remarks" section below.

lpData

Points to the application-supplied data. The data is passed to the callback function along with the object information.

Return Value

Specifies the last value returned by the callback function. Its meaning is user-defined.

Remarks

For each object of a given type, the callback function that you pass is called with the information for that object. The system calls the callback function until there are no more objects or the callback function returns 0.

Note that new features of Microsoft Visual C++ let you use an ordinary function as the function passed to **EnumObjects**. The address passed to **EnumObjects** is a pointer to a function exported with **EXPORT** and with the Pascal calling convention. In protect-mode applications, you do not have to create this function with the Windows MakeProcInstance function or free the function after use with the FreeProcInstance Windows function.

You also do not have to export the function name in an **EXPORTS** statement in your application's module-definition file. You can instead use the **EXPORT** function modifier, as in

int CALLBACK EXPORT AFunction**(LPSTR**, **LPSTR);**

to cause the compiler to emit the proper export record for export by name without aliasing. This works for most needs. For some special cases, such as exporting a function by ordinal or aliasing the export, you still need to use an **EXPORTS** statement in a module-definition file.

For compiling Microsoft Foundation programs, you will normally use the /GA and /GEs compiler options. The /Gw compiler option is not used with the Microsoft Foundation classes. (If you do use the Windows function **MakeProcInstance**, you will need to explicitly cast the returned function pointer from **FARPROC** to the type needed in this API.) Callback registration interfaces are now type-safe (you must pass in a function pointer that points to the right kind of function for the specific callback).

Also note that all callback functions must trap Microsoft Foundation exceptions before returning to Windows, since exceptions cannot be thrown across callback boundaries. For more information about exceptions, see the article Exceptions.

Example

```
// print some info about a pen we're ready to enumerate

BOOL CALLBACK EnumObjectHandler(LPVOID lpLogObject, LPARAM /* lpData */)
{
   LOGPEN* pPen = (LOGPEN*) lpLogObject;

   switch (pPen->lopnStyle)
   {
   case PS_SOLID:
      TRACE0("PS_SOLID:      ");
```

```
        break;
    case PS_DASH:
        TRACE0("PS_DASH:       ");
        break;
    case PS_DOT:
        TRACE0("PS_DOT:        ");
        break;
    case PS_DASHDOT:
        TRACE0("PS_DASHDOT:    ");
        break;
    case PS_DASHDOTDOT:
        TRACE0("PS_DASHDOTDOT: ");
        break;
    case PS_NULL:
        TRACE0("PS_NULL:       ");
        break;
    case PS_INSIDEFRAME:
        TRACE0("PS_INSIDEFRAME:");
        break;
    default:
        TRACE0("unk style:");
    }

    TRACE2("Color: 0x%8.8X, Width: %d\n", pPen->lopnColor, pPen->lopnWidth);
    return TRUE;
}

// get the default printer and enumerate the pens it has

void CMyView::OnEnumPens()
{
    CPrintDialog dlg(FALSE);
    dlg.GetDefaults();
    HDC hdc = dlg.GetPrinterDC();

    if (hdc != NULL)
    {
        CDC dc;
        dc.Attach(hdc);
        VERIFY(dc.EnumObjects(OBJ_PEN, EnumObjectHandler, 0));
    }
}
```

See Also

Reference

CDC Class

Hierarchy Chart

EnumObjects

Other Resources

CDC Members

CDC::Escape

This member function is practically obsolete for Win32 programming.

```
virtual int Escape(
   int nEscape,
   int nCount,
   LPCSTR lpszInData,
   LPVOID lpOutData
);
int Escape(
   int nEscape,
   int nInputSize,
   LPCSTR lpszInputData,
   int nOutputSize,
   LPSTR lpszOutputData
);
```

Parameters

nEscape

Specifies the escape function to be performed.

For a complete list of escape functions, see Escape in the *Platform SDK*.

nCount

Specifies the number of bytes of data pointed to by *lpszInData*.

lpszInData

Points to the input data structure required for this escape.

lpOutData

Points to the structure that is to receive output from this escape. The *lpOutData* parameter is **NULL** if no data is returned.

nInputSize

Specifies the number of bytes of data pointed to by the *lpszInputData* parameter.

lpszInputData

Points to the input structure required for the specified escape.

nOutputSize

Specifies the number of bytes of data pointed to by the *lpszOutputData* parameter.

lpszOutputData

Points to the structure that receives output from this escape. This parameter should be **NULL** if no data is returned.

Return Value

A positive value is returned if the function is successful, except for the **QUERYESCSUPPORT** escape, which only checks for implementation. Zero is returned if the escape is not implemented. A negative value is returned if an error occurred. The following are common error values:

- **SP_ERROR** General error.
- **SP_OUTOFDISK** Not enough disk space is currently available for spooling, and no more space will become available.
- **SP_OUTOFMEMORY** Not enough memory is available for spooling.
- **SP_USERABORT** User ended the job through the Print Manager.

Remarks

Of the original printer escapes, only **QUERYESCSUPPORT** is supported for Win32 applications. All other printer escapes are obsolete and are supported only for compatibility with 16-bit applications.

For Win32 programming, **CDC** now provides six member functions that supersede their corresponding printer escapes:

- CDC::AbortDoc
- CDC::EndDoc
- CDC::EndPage
- CDC::SetAbortProc
- CDC::StartDoc
- CDC::StartPage

In addition, CDC::GetDeviceCaps supports Win32 indexes that supersede other printer escapes. See GetDeviceCaps in the *Platform SDK* for more information.

This member function allows applications to access facilities of a particular device that are not directly available through GDI.

Use the first version if your application uses predefined escape values. Use the second version if your application defines private escape values. See ExtEscape in the *Platform SDK* for more information about the second version.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::ResetDC
EnumObjects

Other Resources

CDC Members

CDC::ExcludeClipRect

Creates a new clipping region that consists of the existing clipping region minus the specified rectangle.

```
int ExcludeClipRect(
   int x1,
   int y1,
   int x2,
   int y2
);
int ExcludeClipRect(
   LPCRECT lpRect
);
```

Parameters

x1

Specifies the logical x-coordinate of the upper-left corner of the rectangle.

y1

Specifies the logical y-coordinate of the upper-left corner of the rectangle.

x2

Specifies the logical x-coordinate of the lower-right corner of the rectangle.

y2

Specifies the logical y-coordinate of the lower-right corner of the rectangle.

lpRect

Specifies the rectangle. Can also be a **CRect** object.

Return Value

Specifies the new clipping region's type. It can be any of the following values:

- **COMPLEXREGION** The region has overlapping borders.
- **ERROR** No region was created.
- **NULLREGION** The region is empty.
- **SIMPLEREGION** The region has no overlapping borders.

Remarks

The width of the rectangle, specified by the absolute value of *x2* – *x1*, must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::ExcludeUpdateRgn
ExcludeClipRect

Other Resources

CDC Members

CDC::ExcludeUpdateRgn

Prevents drawing within invalid areas of a window by excluding an updated region in the window from the clipping region associated with the **CDC** object.

```
int ExcludeUpdateRgn(
   CWnd* pWnd
);
```

Parameters

pWnd

Points to the window object whose window is being updated.

Return Value

The type of excluded region. It can be any one of the following values:

- **COMPLEXREGION** The region has overlapping borders.
- **ERROR** No region was created.
- **NULLREGION** The region is empty.
- **SIMPLEREGION** The region has no overlapping borders.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::ExcludeClipRect
ExcludeUpdateRgn

Other Resources

CDC Members

CDC::ExtFloodFill

Fills an area of the display surface with the current brush.

```
BOOL ExtFloodFill(
   int x,
   int y,
   COLORREF crColor,
   UINT nFillType
);
```

Parameters

x

Specifies the logical x-coordinate of the point where filling begins.

y

Specifies the logical y-coordinate of the point where filling begins.

crColor

Specifies the color of the boundary or of the area to be filled. The interpretation of *crColor* depends on the value of *nFillType*.

nFillType

Specifies the type of flood fill to be performed. It must be either of the following values:

- **FLOODFILLBORDER** The fill area is bounded by the color specified by *crColor*. This style is identical to the filling performed by **FloodFill**.
- **FLOODFILLSURFACE** The fill area is defined by the color specified by *crColor*. Filling continues outward in all directions as long as the color is encountered. This style is useful for filling areas with multicolored boundaries.

Return Value

Nonzero if the function is successful; otherwise 0 if the filling could not be completed, if the given point has the boundary color specified by *crColor* (if **FLOODFILLBORDER** was requested), if the given point does not have the color specified by *crColor* (if **FLOODFILLSURFACE** was requested), or if the point is outside the clipping region.

Remarks

This member function offers more flexibility than **FloodFill** because you can specify a fill type in *nFillType*.

If *nFillType* is set to **FLOODFILLBORDER**, the area is assumed to be completely bounded by the color specified by *crColor*. The function begins at the point specified by *x* and *y* and fills in all directions to the color boundary.

If *nFillType* is set to **FLOODFILLSURFACE**, the function begins at the point specified by *x* and *y* and continues in all directions, filling all adjacent areas containing the color specified by *crColor*.

Only memory-device contexts and devices that support raster-display technology support **ExtFloodFill**. For more information, see the GetDeviceCaps member function.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::FloodFill
CDC::GetDeviceCaps
ExtFloodFill

Other Resources

CDC Members

CDC::ExtTextOut

Call this member function to write a character string within a rectangular region using the currently selected font.

```
virtual BOOL ExtTextOut(
   int x,
   int y,
   UINT nOptions,
   LPCRECT lpRect,
   LPCTSTR lpszString,
   UINT nCount,
   LPINT lpDxWidths
);
BOOL ExtTextOut(
   int x,
   int y,
   UINT nOptions,
   LPCRECT lpRect,
   const CString& str,
   LPINT lpDxWidths
);
```

Parameters

x

Specifies the logical x-coordinate of the character cell for the first character in the specified string.

y

Specifies the logical y-coordinate of the top of the character cell for the first character in the specified string.

nOptions

Specifies the rectangle type. This parameter can be one, both, or neither of the following values:

- **ETO_CLIPPED** Specifies that text is clipped to the rectangle.
- **ETO_OPAQUE** Specifies that the current background color fills the rectangle. (You can set and query the current background color with the SetBkColor and GetBkColor member functions.)

lpRect

Points to a RECT structure that determines the dimensions of the rectangle. This parameter can be **NULL**. You can also pass a CRect object for this parameter.

lpszString

Points to the specified character string to be drawn. You can also pass a CString object for this parameter.

nCount

Specifies the number of characters in the string.

lpDxWidths

Points to an array of values that indicate the distance between origins of adjacent character cells. For instance, *lpDxWidths*[*i*] logical units will separate the origins of character cell *i* and character cell *i* + 1. If *lpDxWidths* is **NULL**, **ExtTextOut** uses the default spacing between characters.

str

A **CString** object that contains the specified characters to be drawn.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The rectangular region can be opaque (filled with the current background color), and it can be a clipping region.

If *nOptions* is 0 and *lpRect* is **NULL**, the function writes text to the device context without using a rectangular region. By default, the current position is not used or updated by the function. If an application needs to update the current position when it calls **ExtTextOut**, the application can call the **CDC** member function SetTextAlign with *nFlags* set to **TA_UPDATECP**. When this flag is set, Windows ignores *x* and *y* on subsequent calls to **ExtTextOut** and uses the current position instead. When an application uses **TA_UPDATECP** to update the current position, **ExtTextOut** sets the current position either to the end of the previous line of text or to the position specified by the last element of the array pointed to by *lpDxWidths*, whichever is greater.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetTextAlign
CDC::TabbedTextOut
CDC::TextOut
CDC::GetBkColor
CDC::SetBkColor
CDC::SetTextColor
ExtTextOut
RECT Structure

Other Resources

CDC Members

CDC::FillPath

Closes any open figures in the current path and fills the path's interior by using the current brush and polygon-filling mode.

```
BOOL FillPath( );
```

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

After its interior is filled, the path is discarded from the device context.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::BeginPath
CDC::SetPolyFillMode
CDC::StrokeAndFillPath
CDC::StrokePath
FillPath

Other Resources

CDC Members

CDC::FillRect

Call this member function to fill a given rectangle using the specified brush.

```
void FillRect(
   LPCRECT lpRect,
   CBrush* pBrush
);
```

Parameters

lpRect

Points to a RECT structure that contains the logical coordinates of the rectangle to be filled. You can also pass a CRect object for this parameter.

pBrush

Identifies the brush used to fill the rectangle.

Remarks

The function fills the complete rectangle, including the left and top borders, but it does not fill the right and bottom borders.

The brush needs to either be created using the CBrush member functions CreateHatchBrush, CreatePatternBrush, and CreateSolidBrush, or retrieved by the GetStockObject Windows function.

When filling the specified rectangle, **FillRect** does not include the rectangle's right and bottom sides. GDI fills a rectangle up to, but does not include, the right column and bottom row, regardless of the current mapping mode. **FillRect** compares the values of the **top**, **bottom**, **left**, and **right** members of the specified rectangle. If **bottom** is less than or equal to **top**, or if **right** is less than or equal to **left**, the rectangle is not drawn.

FillRect is similar to CDC::FillSolidRect; however, **FillRect** takes a brush and therefore can be used to fill a rectangle with a solid color, a dithered color, hatched brushes, or a pattern. **FillSolidRect** uses only solid colors (indicated by a **COLORREF** parameter). **FillRect** usually is slower than **FillSolidRect**.

See Also

Reference

CDC Class
Hierarchy Chart
CBrush::CreateHatchBrush
CBrush::CreatePatternBrush
CBrush::CreateSolidBrush
FillRect
GetStockObject
RECT Structure
CBrush Class
CDC::FillSolidRect

Other Resources

CDC Members

CDC::FillRgn

Fills the region specified by *pRgn* with the brush specified by *pBrush*.

```
BOOL FillRgn(
   CRgn* pRgn,
   CBrush* pBrush
);
```

Parameters

pRgn

A pointer to the region to be filled. The coordinates for the given region are specified in logical units.

pBrush

Identifies the brush to be used to fill the region.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The brush must either be created using the **CBrush** member functions **CreateHatchBrush**, **CreatePatternBrush**, **CreateSolidBrush**, or be retrieved by **GetStockObject**.

Example

See the example for CRgn::CreateRoundRectRgn.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::PaintRgn
CDC::FillRect
CBrush Class
CRgn Class
FillRgn

Other Resources

CDC Members

CDC::FillSolidRect

Call this member function to fill the given rectangle with the specified solid color.

```
void FillSolidRect(
   LPCRECT lpRect,
   COLORREF clr
);
void FillSolidRect(
   int x,
   int y,
   int cx,
   int cy,
   COLORREF clr
);
```

Parameters

lpRect

Specifies the bounding rectangle (in logical units). You can pass either a pointer to a RECT data structure or a **CRect** object for this parameter.

clr Specifies the color to be used to fill the rectangle.

x

Specifies the logical x-coordinate of the upper-left corner of the rectangle.

y

Specifies the logical y-coordinate of the upper-left corner of the destination rectangle.

cx

Specifies the width of the rectangle.

cy

Specifies the height of the rectangle.

Remarks

FillSolidRect is very similar to CDC::FillRect; however, **FillSolidRect** uses only solid colors (indicated by the **COLORREF** parameter), while **FillRect** takes a brush and therefore can be used to fill a rectangle with a solid color, a dithered color, hatched brushes, or a pattern. **FillSolidRect** usually is faster than **FillRect**.

Note
When you call **FillSolidRect**, the background color, which was previously set using SetBkColor, is set to the color indicated by *clr*.

See Also

Reference

CDC Class
Hierarchy Chart
RECT Structure
CRect Class
CDC::FillRect

Other Resources

CDC Members

CDC::FlattenPath

Transforms any curves in the path selected into the current device context, and turns each curve into a sequence of lines.

```
BOOL FlattenPath( );
```

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::WidenPath

Other Resources

CDC Members

CDC::FloodFill

Fills an area of the display surface with the current brush.

```
BOOL FloodFill(
   int x,
   int y,
   COLORREF crColor
);
```

Parameters

x

Specifies the logical x-coordinate of the point where filling begins.

y

Specifies the logical y-coordinate of the point where filling begins.

crColor

Specifies the color of the boundary.

Return Value

Nonzero if the function is successful; otherwise 0 is returned if the filling could not be completed, the given point has the boundary color specified by *crColor*, or the point is outside the clipping region.

Remarks

The area is assumed to be bounded as specified by *crColor*. The **FloodFill** function begins at the point specified by *x* and *y* and continues in all directions to the color boundary.

Only memory-device contexts and devices that support raster-display technology support the **FloodFill** member function. For information about **RC_BITBLT** capability, see the **GetDeviceCaps** member function.

The **ExtFloodFill** function provides similar capability but greater flexibility.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::ExtFloodFill
CDC::GetDeviceCaps
FloodFill

Other Resources

CDC Members

CDC::FrameRect

Draws a border around the rectangle specified by *lpRect*.

```
void FrameRect(
   LPCRECT lpRect,
   CBrush* pBrush
);
```

Parameters

lpRect

Points to a RECT structure or CRect object that contains the logical coordinates of the upper-left and lower-right corners of the rectangle. You can also pass a **CRect** object for this parameter.

pBrush

Identifies the brush to be used for framing the rectangle.

Remarks

The function uses the given brush to draw the border. The width and height of the border is always 1 logical unit.

If the rectangle's **bottom** coordinate is less than or equal to **top**, or if **right** is less than or equal to **left**, the rectangle is not drawn.

The border drawn by **FrameRect** is in the same position as a border drawn by the **Rectangle** member function using the same coordinates (if **Rectangle** uses a pen that is 1 logical unit wide). The interior of the rectangle is not filled by **FrameRect**.

See Also

Reference

CDC Class
Hierarchy Chart
CBrush Class
FrameRect
CDC::Rectangle
CDC::FrameRgn
RECT Structure

Other Resources

CDC Members

CDC::FrameRgn

Draws a border around the region specified by *pRgn* using the brush specified by *pBrush*.

```
BOOL FrameRgn(
   CRgn* pRgn,
   CBrush* pBrush,
   int nWidth,
   int nHeight
);
```

Parameters

pRgn

Points to the **CRgn** object that identifies the region to be enclosed in a border. The coordinates for the given region are specified in logical units.

pBrush

Points to the **CBrush** object that identifies the brush to be used to draw the border.

nWidth

Specifies the width of the border in vertical brush strokes in device units.

nHeight

Specifies the height of the border in horizontal brush strokes in device units.

Return Value

Nonzero if the function is successful; otherwise 0.

Example

See the example for CRgn::CombineRgn.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::Rectangle
CDC::FrameRect
CBrush Class
CRgn Class
FrameRgn

Other Resources

CDC Members

CDC::FromHandle

Returns a pointer to a **CDC** object when given a handle to a device context.

```
static CDC* PASCAL FromHandle(
   HDC hDC
);
```

Parameters

hDC

Contains a handle to a Windows device context.

Return Value

The pointer may be temporary and should not be stored beyond immediate use.

Remarks

If a **CDC** object is not attached to the handle, a temporary **CDC** object is created and attached.

Example

See the example for CPrintDialog::GetPrinterDC.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::DeleteTempMap

Other Resources

CDC Members

CDC::GetArcDirection

Returns the current arc direction for the device context.

```
int GetArcDirection( ) const;
```

Return Value

Specifies the current arc direction, if successful. Following are the valid return values:

- **AD_COUNTERCLOCKWISE** Arcs and rectangles drawn counterclockwise.
- **AD_CLOCKWISE** Arcs and rectangles drawn clockwise.

If an error occurs, the return value is zero.

Remarks

Arc and rectangle functions use the arc direction.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetArcDirection
GetArcDirection

Other Resources

CDC Members

CDC::GetAspectRatioFilter

Retrieves the setting for the current aspect-ratio filter.

```
CSize GetAspectRatioFilter( ) const;
```

Return Value

A **CSize** object representing the aspect ratio used by the current aspect ratio filter.

Remarks

The aspect ratio is the ratio formed by a device's pixel width and height. Information about a device's aspect ratio is used in the creation, selection, and display of fonts. Windows provides a special filter, the aspect-ratio filter, to select fonts designed for a particular aspect ratio from all of the available fonts. The filter uses the aspect ratio specified by the **SetMapperFlags** member function.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetMapperFlags
CSize Class

Other Resources

CDC Members

CDC::GetBkColor

Returns the current background color.

```
COLORREF GetBkColor( ) const;
```

Return Value

An RGB color value.

Remarks

If the background mode is **OPAQUE**, the system uses the background color to fill the gaps in styled lines, the gaps between hatched lines in brushes, and the background in character cells. The system also uses the background color when converting bitmaps between color and monochrome device contexts.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetBkMode
CDC::SetBkColor
CDC::SetBkMode
GetBkColor

Other Resources

CDC Members

CDC::GetBkMode

Returns the background mode.

```
int GetBkMode( ) const;
```

Return Value

The current background mode, which can be **OPAQUE** or **TRANSPARENT**.

Remarks

The background mode defines whether the system removes existing background colors on the drawing surface before drawing text, hatched brushes, or any pen style that is not a solid line.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetBkColor
CDC::SetBkColor
CDC::SetBkMode
GetBkMode

Other Resources

CDC Members

CDC::GetBoundsRect

Returns the current accumulated bounding rectangle for the specified device context.

```
UINT GetBoundsRect(
   LPRECT lpRectBounds,
   UINT flags
);
```

Parameters

lpRectBounds

Points to a buffer that will receive the current bounding rectangle. The rectangle is returned in logical coordinates.

flags

Specifies whether the bounding rectangle is to be cleared after it is returned. This parameter can be either of the following values:

- **DCB_RESET** Forces the bounding rectangle to be cleared after it is returned.

Return Value

Specifies the current state of the bounding rectangle if the function is successful. It can be a combination of the following values:

- **DCB_ACCUMULATE** Bounding rectangle accumulation is occurring.
- **DCB_RESET** Bounding rectangle is empty.
- **DCB_SET** Bounding rectangle is not empty.
- **DCB_ENABLE** Bounding accumulation is on.
- **DCB_DISABLE** Bounding accumulation is off.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetBoundsRect
GetBoundsRect

Other Resources

CDC Members

CDC::GetBrushOrg

Retrieves the origin (in device units) of the brush currently selected for the device context.

```
CPoint GetBrushOrg( ) const;
```

Return Value

The current origin of the brush (in device units) as a CPoint object.

Remarks

The initial brush origin is at (0,0) of the client area. The return value specifies this point in device units relative to the origin of the desktop window.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetBrushOrg
CPoint Class

Other Resources

CDC Members

CDC::GetCharacterPlacement

Retrieves various types of information on a character string.

```
DWORD GetCharacterPlacement(
   LPCTSTR lpString,
   int nCount,
   int nMaxExtent,
   LPGCP_RESULTS lpResults,
   DWORD dwFlags
) const;
DWORD GetCharacterPlacement(
   CString& str,
   int nMaxExtent,
   LPGCP_RESULTS lpResults,
   DWORD dwFlags
) const;
```

Parameters

lpString

A pointer to the character string to process.

nCount

Specifies the length of the string. For the ANSI version, it is a **BYTE** count and for the Unicode function it is a **WORD** count. For more information, see GetCharacterPlacement.

nMaxExtent

Specifies the maximum extent (in logical units) to which the string is processed. Characters that, if processed, would exceed this extent are ignored. Computations for any required ordering or glyph arrays apply only to the included characters. This parameter is used only if the **GCP_MAXEXTENT** value is specified in the *dwFlags* parameter. As the function processes the input string, each character and its extent is added to the output, extent, and other arrays only if the total extent has not yet exceeded the maximum. Once the limit is reached, processing will stop.

lpResults

Pointer to a GCP_RESULTS structure that receives the results of the function.

dwFlags

Specifies how to process the string into the required arrays. This parameter can be one or more of the values listed in the *dwFlags* section of the GetCharacterPlacement topic.

str

A pointer to a CString object to process.

Return Value

If the function succeeds, the return value is the width and height of the string in logical units.

If the function fails, the return value is zero.

Remarks

This member function emulates the functionality of the function GetCharacterPlacement, as described in the *Platform SDK*.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::ExtTextOut
CDC::GetCharABCWidths
CDC::GetTextMetrics

Other Resources

CDC::GetCharABCWidths

Retrieves the widths of consecutive characters in a specified range from the current TrueType font.

```
BOOL GetCharABCWidths(
   UINT nFirstChar,
   UINT nLastChar,
   LPABC lpabc
) const;
BOOL GetCharABCWidths(
   UINT nFirstChar,
   UINT nLastChar,
   LPABCFLOAT lpABCF
) const;
```

Parameters

nFirstChar

Specifies the first character in the range of characters from the current font for which character widths are returned.

nLastChar

Specifies the last character in the range of characters from the current font for which character widths are returned.

lpabc

Points to an array of ABC structures that receive the character widths when the function returns. This array must contain at least as many **ABC** structures as there are characters in the range specified by the *nFirstChar* and *nLastChar* parameters.

lpABCF

Points to an application-supplied buffer with an array of ABCFLOAT structures to receive the character widths when the function returns. The widths returned by this function are in the IEEE floating-point format.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The widths are returned in logical units. This function succeeds only with TrueType fonts.

The TrueType rasterizer provides "ABC" character spacing after a specific point size has been selected. "A" spacing is the distance that is added to the current position before placing the glyph. "B" spacing is the width of the black part of the glyph. "C" spacing is added to the current position to account for the white space to the right of the glyph. The total advanced width is given by A + B + C.

When the **GetCharABCWidths** member function retrieves negative "A" or "C" widths for a character, that character includes underhangs or overhangs.

To convert the ABC widths to font design units, an application should create a font whose height (as specified in the **lfHeight** member of the LOGFONT structure) is equal to the value stored in the **ntmSizeEM** member of the NEWTEXTMETRIC structure. (The value of the **ntmSizeEM** member can be retrieved by calling the EnumFontFamilies Windows function.)

The ABC widths of the default character are used for characters that are outside the range of the currently selected font.

To retrieve the widths of characters in non-TrueType fonts, applications should use the GetCharWidth Windows function.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetCharWidth
GetCharABCWidths
GetCharABCWidthsFloat
GetCharWidthFloat

Other Resources

CDC::GetCharABCWidthsI

Retrieves the widths, in logical units, of consecutive glyph indices in a specified range from the current TrueType font.

```
BOOL GetCharABCWidthsI(
   UINT giFirst,
   UINT cgi,
   LPWORD pgi,
   LPABC lpabc
) const;
```

Parameters

giFirst

Specifies the first glyph index in the group of consecutive glyph indices from the current font. This parameter is only used if the *pgi* parameter is **NULL**.

cgi

Specifies the number of glyph indices.

pgi

A pointer to an array containing glyph indices. If the value is **NULL**, the *giFirst* parameter is used instead. The *cgi* parameter specifies the number of glyph indices in this array.

lpabc

Pointer to an array of ABC structures receiving the character widths. This array must contain at least as many **ABC** structures as there are glyph indices specified by the *cgi* parameter.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

This member function emulates the functionality of the function GetCharABCWidthsI, as described in the *Platform SDK*.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetCharWidth
CDC::GetOutlineTextMetrics

Other Resources

CDC Members

CDC::GetCharWidth

Retrieves the widths of individual characters in a consecutive group of characters from the current font, using **m_hAttribDC**, the input device context.

```
BOOL GetCharWidth(
   UINT nFirstChar,
   UINT nLastChar,
   LPINT lpBuffer
) const;
BOOL GetCharWidth(
   UINT nFirstChar,
   UINT nLastChar,
   float* lpFloatBuffer
) const;
```

Parameters

nFirstChar

Specifies the first character in a consecutive group of characters in the current font.

nLastChar

Specifies the last character in a consecutive group of characters in the current font.

lpBuffer

Points to a buffer that will receive the width values for a consecutive group of characters in the current font.

lpFloatBuffer

Points to a buffer to receive the character widths. The returned widths are in the 32-bit IEEE floating-point format. (The widths are measured along the base line of the characters.)

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

For example, if *nFirstChar* identifies the letter 'a' and *nLastChar* identifies the letter 'z', the function retrieves the widths of all lowercase characters.

The function stores the values in the buffer pointed to by *lpBuffer*. This buffer must be large enough to hold all of the widths. That is, there must be at least 26 entries in the example given.

If a character in the consecutive group of characters does not exist in a particular font, it will be assigned the width value of the default character.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetOutputCharWidth
CDC::m_hAttribDC
CDC::m_hDC
CDC::GetCharABCWidths
GetCharWidth
GetCharABCWidths
GetCharABCWidthsFloat
GetCharWidthFloat

Other Resources

CDC Members

CDC::GetCharWidthI

Retrieves the widths, in logical coordinates, of consecutive glyph indices in a specified range from the current font.

```
BOOL GetCharWidthI(
   UINT giFirst,
   UINT cgi,
   LPWORD pgi,
   LPINT lpBuffer
) const;
```

Parameters

giFirst

Specifies the first glyph index in the group of consecutive glyph indices from the current font. This parameter is only used if the *pgi* parameter is **NULL**.

cgi

Specifies the number of glyph indices.

pgi

A pointer to an array containing glyph indices. If the value is **NULL**, the *giFirst* parameter is used instead. The *cgi* parameter specifies the number of glyph indices in this array.

lpBuffer

A pointer to a buffer that receives the widths.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

This member function emulates the functionality of the function GetCharWidthI, as described in the *Platform SDK*.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetCharABCWidths

Other Resources

CDC Members

CDC::GetClipBox

Retrieves the dimensions of the tightest bounding rectangle around the current clipping boundary.

```
virtual int GetClipBox(
   LPRECT lpRect
) const;
```

Parameters

lpRect

Points to the RECT structure or CRect object that is to receive the rectangle dimensions.

Return Value

The clipping region's type. It can be any of the following values:

- **COMPLEXREGION** Clipping region has overlapping borders.
- **ERROR** Device context is not valid.
- **NULLREGION** Clipping region is empty.
- **SIMPLEREGION** Clipping region has no overlapping borders.

Remarks

The dimensions are copied to the buffer pointed to by *lpRect*.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SelectClipRgn
GetClipBox
RECT Structure

Other Resources

CDC Members

CDC::GetColorAdjustment

Retrieves the color adjustment values for the device context.

```
BOOL GetColorAdjustment(
   LPCOLORADJUSTMENT lpColorAdjust
) const;
```

Parameters

lpColorAdjust

Points to a COLORADJUSTMENT data structure to receive the color adjustment values.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

Reference

CDC Class

Hierarchy Chart

CDC::SetColorAdjustment

Other Resources

CDC Members

CDC::GetCurrentBitmap

Returns a pointer to the currently selected **CBitmap** object.

```
CBitmap* GetCurrentBitmap( ) const;
```

Return Value

Pointer to a **CBitmap** object, if successful; otherwise **NULL**.

Remarks

This member function may return temporary objects.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SelectObject
GetCurrentObject

Other Resources

CDC Members

CDC::GetCurrentBrush

Returns a pointer to the currently selected **CBrush** object.

```
CBrush* GetCurrentBrush( ) const;
```

Return Value

Pointer to a **CBrush** object, if successful; otherwise **NULL**.

Remarks

This member function may return temporary objects.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SelectObject
GetCurrentObject

Other Resources

CDC Members

CDC::GetCurrentFont

Returns a pointer to the currently selected **CFont** object.

```
CFont* GetCurrentFont( ) const;
```

Return Value

Pointer to a **CFont** object, if successful; otherwise **NULL**.

Remarks

This member function may return temporary objects.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SelectObject
GetCurrentObject

Other Resources

CDC Members

CDC::GetCurrentPalette

Returns a pointer to the currently selected **CPalette** object.

```
CPalette* GetCurrentPalette( ) const;
```

Return Value

Pointer to a **CPalette** object, if successful; otherwise **NULL**.

Remarks

This member function may return temporary objects.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SelectObject
GetCurrentObject

Other Resources

CDC Members

CDC::GetCurrentPen

Returns a pointer to the currently selected **CPen** object.

```
CPen* GetCurrentPen( ) const;
```

Return Value

Pointer to a **CPen** object, if successful; otherwise **NULL**.

Remarks

This member function may return temporary objects.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SelectObject
GetCurrentObject

Other Resources

CDC Members

CDC::GetCurrentPosition

Retrieves the current position (in logical coordinates).

```
CPoint GetCurrentPosition( ) const;
```

Return Value

The current position as a **CPoint** object.

Remarks

The current position can be set with the **MoveTo** member function.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::MoveTo
CPoint Class

Other Resources

CDC Members

CDC::GetDCBrushColor

Retrieves the current brush color.

```
COLORREF GetDCBrushColor( ) const;
```

Return Value

If the function succeeds, the return value is the COLORREF value for the current brush color.

If the function fails, the return value is **CLR_INVALID**.

Remarks

This member function emulates the functionality of the function GetDCBrushColor, as described in the *Platform SDK*.

See Also

Reference

CDC Class

Hierarchy Chart

CDC::SetDCBrushColor

Other Resources

CDC Members

CDC::GetDCPenColor

Retrieves the current pen color.

```
COLORREF GetDCPenColor( ) const;
```

Return Value

If the function succeeds, the return value is the COLORREF value for the current brush color.

If the function fails, the return value is **CLR_INVALID**.

Remarks

This member function utilizes the Win32 function GetDCPenColor, as described in the *Platform SDK*.

See Also

Reference

CDC Class

Hierarchy Chart

CDC::SetDCPenColor

Other Resources

CDC Members

CDC::GetDeviceCaps

Retrieves a wide range of device-specific information about the display device.

```
int GetDeviceCaps(
   int nIndex
) const;
```

Parameters

nIndex

Specifies the type of information to return. See GetDeviceCaps in the *Platform SDK* for a list of values.

Return Value

The value of the requested capability if the function is successful.

Example

See the example for CPrintDialog::GetDefaults.

See Also

Reference

CDC Class

Hierarchy Chart

Escape

Other Resources

CDC Members

CDC::GetFontData

Retrieves font-metric information from a scalable font file.

```
DWORD GetFontData(
   DWORD dwTable,
   DWORD dwOffset,
   LPVOID lpData,
   DWORD cbData
) const;
```

Parameters

dwTable

Specifies the name of the metric table to be returned. This parameter can be one of the metric tables documented in the TrueType Font Files specification published by Microsoft Corporation. If this parameter is 0, the information is retrieved starting at the beginning of the font file.

dwOffset

Specifies the offset from the beginning of the table at which to begin retrieving information. If this parameter is 0, the information is retrieved starting at the beginning of the table specified by the *dwTable* parameter. If this value is greater than or equal to the size of the table, **GetFontData** returns 0.

lpData

Points to a buffer that will receive the font information. If this value is **NULL**, the function returns the size of the buffer required for the font data specified in the *dwTable* parameter.

cbData

Specifies the length, in bytes, of the information to be retrieved. If this parameter is 0, **GetFontData** returns the size of the data specified in the *dwTable* parameter.

Return Value

Specifies the number of bytes returned in the buffer pointed to by *lpData* if the function is successful; otherwise –1.

Remarks

The information to retrieve is identified by specifying an offset into the font file and the length of the information to return.

An application can sometimes use the **GetFontData** member function to save a TrueType font with a document. To do this, the application determines whether the font can be embedded and then retrieves the entire font file, specifying 0 for the *dwTable*, *dwOffset*, and *cbData* parameters.

Applications can determine whether a font can be embedded by checking the **otmfsType** member of the OUTLINETEXTMETRIC structure. If bit 1 of **otmfsType** is set, embedding is not permitted for the font. If bit 1 is clear, the font can be embedded. If bit 2 is set, the embedding is read only.

If an application attempts to use this function to retrieve information for a non-TrueType font, the **GetFontData** member function returns –1.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetOutlineTextMetrics
GetFontData
OUTLINETEXTMETRIC

Other Resources

CDC Members

CDC::GetFontLanguageInfo

Returns information about the currently selected font for the specified display context.

```
DWORD GetFontLanguageInfo( ) const;
```

Return Value

The return value identifies characteristics of the currently selected font. For a complete listing of possible values, see GetFontLanguageInfo.

Remarks

This member function emulates the functionality of the function GetFontLanguageInfo, as described in the *Platform SDK*.

See Also

Reference

CDC Class

Hierarchy Chart

Other Resources

CDC Members

CDC::GetGlyphOutline

Retrieves the outline curve or bitmap for an outline character in the current font.

```
DWORD GetGlyphOutline(
   UINT nChar,
   UINT nFormat,
   LPGLYPHMETRICS lpgm,
   DWORD cbBuffer,
   LPVOID lpBuffer,
   const MAT2* lpmat2
) const;
```

Parameters

nChar

Specifies the character for which information is to be returned.

nFormat

Specifies the format in which the function is to return information. It can be one of the following values, or 0:

Value	Meaning
GGO_BITMAP	Returns the glyph bitmap. When the function returns, the buffer pointed to by *lpBuffer* contains a 1-bit-per-pixel bitmap whose rows start on doubleword boundaries.
GGO_NATIVE	Returns the curve data points in the rasterizer's native format, using device units. When this value is specified, any transformation specified in *lpmat2* is ignored.

When the value of *nFormat* is 0, the function fills in a GLYPHMETRICS structure but does not return glyph-outline data.

lpgm

Points to a **GLYPHMETRICS** structure that describes the placement of the glyph in the character cell.

cbBuffer

Specifies the size of the buffer into which the function copies information about the outline character. If this value is 0 and the *nFormat* parameter is either the **GGO_BITMAP** or **GGO_NATIVE** values, the function returns the required size of the buffer.

lpBuffer

Points to a buffer into which the function copies information about the outline character. If *nFormat* specifies the **GGO_NATIVE** value, the information is copied in the form of **TTPOLYGONHEADER** and **TTPOLYCURVE** structures. If this value is **NULL** and *nFormat* is either the **GGO_BITMAP** or **GGO_NATIVE** value, the function returns the required size of the buffer.

lpmat2

Points to a MAT2 structure that contains a transformation matrix for the character. This parameter cannot be **NULL**, even when the **GGO_NATIVE** value is specified for *nFormat*.

Return Value

The size, in bytes, of the buffer required for the retrieved information if *cbBuffer* is 0 or *lpBuffer* is **NULL**. Otherwise, it is a positive value if the function is successful, or –1 if there is an error.

Remarks

An application can rotate characters retrieved in bitmap format by specifying a 2-by-2 transformation matrix in the structure pointed to by *lpmat2*.

A glyph outline is returned as a series of contours. Each contour is defined by a TTPOLYGONHEADER structure followed by as many **TTPOLYCURVE** structures as are required to describe it. All points are returned as POINTFX structures and represent

absolute positions, not relative moves. The starting point given by the **pfxStart** member of the TTPOLYGONHEADER structure is the point at which the outline for a contour begins. The TTPOLYCURVE structures that follow can be either polyline records or spline records. Polyline records are a series of points; lines drawn between the points describe the outline of the character. Spline records represent the quadratic curves used by TrueType (that is, quadratic b-splines).

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetOutlineTextMetrics
GetGlyphOutline
GLYPHMETRICS
TTPOLYGONHEADER
TTPOLYCURVE

Other Resources

CDC Members

CDC::GetGraphicsMode

Retrieves the current graphics mode for the specified device context.

```
int GetGraphicsMode() const;
```

Return Value

Returns the current graphics mode on success. For a list of the values that this method can return, see GetGraphicsMode.

Returns 0 on failure.

If you are using Windows NT, Windows 2000, or Windows, XP, to get extended error information, call GetLastError.

Remarks

This method wraps the Windows GDI function GetGraphicsMode.

See Also

Reference

CDC Class

Hierarchy Chart

Other Resources

CDC Members

CDC::GetHalftoneBrush

Call this member function to retrieve a halftone brush.

```
static CBrush* PASCAL GetHalftoneBrush( );
```

Return Value

A pointer to a **CBrush** object if successful; otherwise **NULL**.

Remarks

A halftone brush shows pixels that are alternately foreground and background colors to create a dithered pattern. The following is an example of a dithered pattern created by a halftone brush.

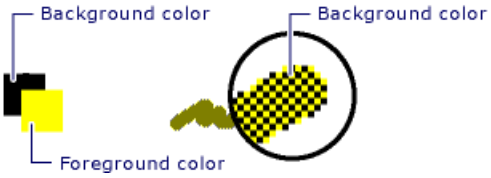

See Also

Reference

CDC Class

Hierarchy Chart

CBrush Class

Other Resources

CDC Members

CDC::GetKerningPairs

Retrieves the character kerning pairs for the font that is currently selected in the specified device context.

```
int GetKerningPairs(
   int nPairs,
   LPKERNINGPAIR lpkrnpair
) const;
```

Parameters

nPairs

Specifies the number of KERNINGPAIR structures pointed to by *lpkrnpair*. The function will not copy more kerning pairs than specified by *nPairs*.

lpkrnpair

Points to an array of **KERNINGPAIR** structures that receive the kerning pairs when the function returns. This array must contain at least as many structures as specified by *nPairs*. If this parameter is **NULL**, the function returns the total number of kerning pairs for the font.

Return Value

Specifies the number of kerning pairs retrieved or the total number of kerning pairs in the font, if the function is successful. Zero is returned if the function fails or there are no kerning pairs for the font.

See Also

Reference

CDC Class
Hierarchy Chart
GetKerningPairs
KERNINGPAIR

Other Resources

CDC Members

CDC::GetLayout

Call this member function to determine the layout of the text and graphics for a device context, such as a printer or a metafile.

```
DWORD GetLayout( ) const;
```

Return Value

If successful, the layout flags for the current device context. Otherwise, **GDI_ERROR**. For extended error information, call GetLastError. For a list of the layout flags, see CDC::SetLayout.

Remarks

The default layout is left to right.

See Also

Reference

CDC Class
Hierarchy Chart
GetLayout

Other Resources

CDC Members

CDC::GetMapMode

Retrieves the current mapping mode.

```
int GetMapMode( ) const;
```

Return Value

The mapping mode.

Remarks

For a description of the mapping modes, see the **SetMapMode** member function.

Note
If you call SetLayout to change the DC to right-to-left layout, **SetLayout** automatically changes the mapping mode to **MM_ISOTROPIC**. Consequently, any subsequent call to **GetMapMode** will return **MM_ISOTROPIC**.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetMapMode
CDC::GetLayout
GetMapMode

Other Resources

CDC Members

CDC::GetMiterLimit

Returns the miter limit for the device context.

```
float GetMiterLimit( ) const;
```

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The miter limit is used when drawing geometric lines that have miter joins.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetMiterLimit
GetMiterLimit

Other Resources

CDC Members

CDC::GetNearestColor

Returns the solid color that best matches a specified logical color.

```
COLORREF GetNearestColor(
   COLORREF crColor
) const;
```

Parameters

crColor

Specifies the color to be matched.

Return Value

An RGB (red, green, blue) color value that defines the solid color closest to the *crColor* value that the device can represent.

Remarks

The given device must be able to represent this color.

See Also

Reference

CDC Class
Hierarchy Chart
GetNearestColor
CPalette::GetNearestPaletteIndex

Other Resources

CDC Members

CDC::GetOutlineTextMetrics

Retrieves metric information for TrueType fonts.

```
UINT GetOutlineTextMetrics(
   UINT cbData,
   LPOUTLINETEXTMETRIC lpotm
) const;
```

Parameters

lpotm

Points to an array of OUTLINETEXTMETRIC structures. If this parameter is **NULL**, the function returns the size of the buffer required for the retrieved metric data.

cbData

Specifies the size, in bytes, of the buffer to which information is returned.

lpotm

Points to an **OUTLINETEXTMETRIC** structure. If this parameter is **NULL**, the function returns the size of the buffer required for the retrieved metric information.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The OUTLINETEXTMETRIC structure contains most of the font metric information provided with the TrueType format, including a TEXTMETRIC structure. The last four members of the **OUTLINETEXTMETRIC** structure are pointers to strings. Applications should allocate space for these strings in addition to the space required for the other members. Because there is no system-imposed limit to the size of the strings, the simplest method for allocating memory is to retrieve the required size by specifying **NULL** for *lpotm* in the first call to the **GetOutlineTextMetrics** function.

See Also

Reference

CDC Class
Hierarchy Chart
GetTextMetrics
GetOutlineTextMetrics
CDC::GetTextMetrics

Other Resources

CDC Members

CDC::GetOutputCharWidth

Uses the output device context, **m_hDC**, and retrieves the widths of individual characters in a consecutive group of characters from the current font.

```
BOOL GetOutputCharWidth(
   UINT nFirstChar,
   UINT nLastChar,
   LPINT lpBuffer
) const;
```

Parameters

nFirstChar

Specifies the first character in a consecutive group of characters in the current font.

nLastChar

Specifies the last character in a consecutive group of characters in the current font.

lpBuffer

Points to a buffer that will receive the width values for a consecutive group of characters in the current font.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

For example, if *nFirstChar* identifies the letter 'a' and *nLastChar* identifies the letter 'z', the function retrieves the widths of all lowercase characters.

The function stores the values in the buffer pointed to by *lpBuffer*. This buffer must be large enough to hold all of the widths; that is, there must be at least 26 entries in the example given.

If a character in the consecutive group of characters does not exist in a particular font, it will be assigned the width value of the default character.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetCharWidth
CDC::m_hAttribDC
CDC::m_hDC
GetCharWidth

Other Resources

CDC Members

CDC::GetOutputTabbedTextExtent

Call this member function to compute the width and height of a character string using m_hDC, the output device context.

```
CSize GetOutputTabbedTextExtent(
   LPCTSTR lpszString,
   int nCount,
   int nTabPositions,
   LPINT lpnTabStopPositions
) const;
CSize GetOutputTabbedTextExtent(
   const CString& str,
   int nTabPositions,
   LPINT lpnTabStopPositions
) const;
```

Parameters

lpszString

Points to a character string to be measured. You can also pass a CString object for this parameter.

nCount

Specifies the number of characters in the string. If *nCount* is –1, the length is calculated.

nTabPositions

Specifies the number of tab-stop positions in the array pointed to by *lpnTabStopPositions*.

lpnTabStopPositions

Points to an array of integers containing the tab-stop positions in logical units. The tab stops must be sorted in increasing order; the smallest x-value should be the first item in the array. Back tabs are not allowed.

str

A **CString** object that contains the specified characters to be measured.

Return Value

The dimensions of the string (in logical units) in a CSize object.

Remarks

If the string contains one or more tab characters, the width of the string is based upon the tab stops specified by *lpnTabStopPositions*. The function uses the currently selected font to compute the dimensions of the string.

The current clipping region does not offset the width and height returned by the **GetOutputTabbedTextExtent** function.

Since some devices do not place characters in regular cell arrays (that is, they kern the characters), the sum of the extents of the characters in a string may not be equal to the extent of the string.

If *nTabPositions* is 0 and *lpnTabStopPositions* is **NULL**, tabs are expanded to eight average character widths. If *nTabPositions* is 1, the tab stops will be separated by the distance specified by the first value in the array to which *lpnTabStopPositions* points. If *lpnTabStopPositions* points to more than a single value, a tab stop is set for each value in the array, up to the number specified by *nTabPositions*.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetTextExtent
CDC::m_hAttribDC
CDC::m_hDC
CDC::GetTabbedTextExtent
CDC::GetOutputTextExtent
CDC::TabbedTextOut

GetTabbedTextExtent
CSize Class
Other Resources
CDC Members

CDC::GetOutputTextExtent

Call this member function to use the output device context, m_hDC, and compute the width and height of a line of text, using the current font.

```
CSize GetOutputTextExtent(
   LPCTSTR lpszString,
   int nCount
) const;
CSize GetOutputTextExtent(
   const CString& str
) const;
```

Parameters

lpszString

Points to a string of characters. You can also pass a CString object for this parameter.

nCount

Specifies the number of characters in the string. If *nCount* is –1, the length is calculated.

str

A **CString** object that contains the specified characters to be measured.

Return Value

The dimensions of the string (in logical units) returned in a CSize object.

Remarks

The current clipping region does not affect the width and height returned by **GetOutputTextExtent**.

Since some devices do not place characters in regular cell arrays (that is, they carry out kerning), the sum of the extents of the characters in a string may not be equal to the extent of the string.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetTabbedTextExtent
CDC::GetOutputTabbedTextExtent
CDC::m_hAttribDC
CDC::m_hDC
CDC::GetTextExtent
CDC::SetTextJustification
CSize Class

Other Resources

CDC Members

CDC::GetOutputTextMetrics

Retrieves the metrics for the current font using **m_hDC**, the output device context.

```
BOOL GetOutputTextMetrics(
   LPTEXTMETRIC lpMetrics
) const;
```

Parameters

lpMetrics

Points to the TEXTMETRIC structure that receives the metrics.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetTextAlign
CDC::m_hAttribDC
CDC::m_hDC
CDC::GetTextMetrics
CDC::GetTextExtent
CDC::GetTextFace
CDC::SetTextJustification
GetTextMetrics

Other Resources

CDC Members

CDC::GetPath

Retrieves the coordinates defining the endpoints of lines and the control points of curves found in the path that is selected into the device context.

```
int GetPath(
   LPPOINT lpPoints,
   LPBYTE lpTypes,
   int nCount
) const;
```

Parameters

lpPoints

Points to an array of POINT data structures or **CPoint** objects where the line endpoints and curve control points are placed.

lpTypes

Points to an array of bytes where the vertex types are placed. Values are one of the following:

- **PT_MOVETO** Specifies that the corresponding point in *lpPoints* starts a disjoint figure.
- **PT_LINETO** Specifies that the previous point and the corresponding point in *lpPoints* are the endpoints of a line.
- **PT_BEZIERTO** Specifies that the corresponding point in *lpPoints* is a control point or ending point for a Bzier curve.

  PT_BEZIERTO types always occur in sets of three. The point in the path immediately preceding them defines the starting point for the Bzier curve. The first two **PT_BEZIERTO** points are the control points, and the third **PT_BEZIERTO** point is the end point (if hard-coded).

  A **PT_LINETO** or **PT_BEZIERTO** type may be combined with the following flag (by using the bitwise operator **OR**) to indicate that the corresponding point is the last point in a figure and that the figure should be closed:

- **PT_CLOSEFIGURE** Specifies that the figure is automatically closed after the corresponding line or curve is drawn. The figure is closed by drawing a line from the line or curve endpoint to the point corresponding to the last **PT_MOVETO**.

nCount

Specifies the total number of POINT data structures that may be placed in the *lpPoints* array. This value must be the same as the number of bytes that may be placed in the *lpTypes* array.

Return Value

If the *nCount* parameter is nonzero, the number of points enumerated. If *nCount* is 0, the total number of points in the path (and **GetPath** writes nothing to the buffers). If *nCount* is nonzero and is less than the number of points in the path, the return value is -1.

Remarks

The device context must contain a closed path. The points of the path are returned in logical coordinates. Points are stored in the path in device coordinates, so **GetPath** changes the points from device coordinates to logical coordinates by using the inverse of the current transformation. The **FlattenPath** member function may be called before **GetPath**, to convert all curves in the path into line segments.

Example

See the example for CDC::BeginPath.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::FlattenPath
CDC::PolyDraw
CDC::WidenPath

Other Resources

CDC::GetPixel

Retrieves the RGB color value of the pixel at the point specified by *x* and *y*.

```
COLORREF GetPixel(
   int x,
   int y
) const;
COLORREF GetPixel(
   POINT point
) const;
```

Parameters

x

Specifies the logical x-coordinate of the point to be examined.

y

Specifies the logical y-coordinate of the point to be examined.

point

Specifies the logical x- and y-coordinates of the point to be examined.

Return Value

For either version of the function, an RGB color value for the color of the given point. It is –1 if the coordinates do not specify a point in the clipping region.

Remarks

The point must be in the clipping region. If the point is not in the clipping region, the function has no effect and returns –1.

Not all devices support the **GetPixel** function. For more information, see the **RC_BITBLT** raster capability under the GetDeviceCaps member function.

The **GetPixel** member function has two forms. The first takes two coordinate values; the second takes either a POINT structure or a CPoint object.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetDeviceCaps
CDC::SetPixel
GetPixel
POINT Structure
CPoint Class

Other Resources

CDC Members

CDC::GetPolyFillMode

Retrieves the current polygon-filling mode.

```
int GetPolyFillMode( ) const;
```

Return Value

The current polygon-filled mode, **ALTERNATE** or **WINDING**, if the function is successful.

Remarks

See the **SetPolyFillMode** member function for a description of the polygon-filling modes.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetPolyFillMode
GetPolyFillMode

Other Resources

CDC Members

CDC::GetROP2

Retrieves the current drawing mode.

```
int GetROP2( ) const;
```

Return Value

The drawing mode. For a list of the drawing mode values, see the **SetROP2** member function.

Remarks

The drawing mode specifies how the colors of the pen and the interior of filled objects are combined with the color already on the display surface.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetDeviceCaps
CDC::SetROP2
GetROP2

Other Resources

CDC Members

CDC::GetSafeHdc

Call this member function to get m_hDC, the output device context.

```
HDC GetSafeHdc( ) const;
```

Return Value

A device context handle.

Remarks

This member function also works with null pointers.

See Also

Reference

CDC Class

Hierarchy Chart

Other Resources

CDC Members

CDC::GetStretchBltMode

Retrieves the current bitmap-stretching mode.

```
int GetStretchBltMode( ) const;
```

Return Value

The return value specifies the current bitmap-stretching mode — **STRETCH_ANDSCANS**, **STRETCH_DELETESCANS**, or **STRETCH_ORSCANS** — if the function is successful.

Remarks

The bitmap-stretching mode defines how information is removed from bitmaps that are stretched or compressed by the **StretchBlt** member function.

The **STRETCH_ANDSCANS** and **STRETCH_ORSCANS** modes are typically used to preserve foreground pixels in monochrome bitmaps. The **STRETCH_DELETESCANS** mode is typically used to preserve color in color bitmaps.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::StretchBlt
CDC::SetStretchBltMode
GetStretchBltMode

Other Resources

CDC Members

CDC::GetTabbedTextExtent

Call this member function to compute the width and height of a character string using m_hAttribDC, the attribute device context.

```
CSize GetTabbedTextExtent(
   LPCTSTR lpszString,
   int nCount,
   int nTabPositions,
   LPINT lpnTabStopPositions
) const;
CSize GetTabbedTextExtent(
   const CString& str,
   int nTabPositions,
   LPINT lpnTabStopPositions
) const;
```

Parameters

lpszString

Points to a character string. You can also pass a CString object for this parameter.

nCount

Specifies the number of characters in the string. If *nCount* is –1, the length is calculated.

nTabPositions

Specifies the number of tab-stop positions in the array pointed to by *lpnTabStopPositions*.

lpnTabStopPositions

Points to an array of integers containing the tab-stop positions in logical units. The tab stops must be sorted in increasing order; the smallest x-value should be the first item in the array. Back tabs are not allowed.

str

A **CString** object that contains the specified characters to be drawn.

Return Value

The dimensions of the string (in logical units) in a CSize object.

Remarks

If the string contains one or more tab characters, the width of the string is based upon the tab stops specified by *lpnTabStopPositions*. The function uses the currently selected font to compute the dimensions of the string.

The current clipping region does not offset the width and height returned by the **GetTabbedTextExtent** function.

Since some devices do not place characters in regular cell arrays (that is, they kern the characters), the sum of the extents of the characters in a string may not be equal to the extent of the string.

If *nTabPositions* is 0 and *lpnTabStopPositions* is **NULL**, tabs are expanded to eight times the average character width. If *nTabPositions* is 1, the tab stops will be separated by the distance specified by the first value in the array to which *lpnTabStopPositions* points. If *lpnTabStopPositions* points to more than a single value, a tab stop is set for each value in the array, up to the number specified by *nTabPositions*.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetTextExtent
CDC::GetOutputTabbedTextExtent
CDC::GetOutputTextExtent
CDC::TabbedTextOut
GetTabbedTextExtent

CSize Class

Other Resources

CDC Members

CDC::GetTextAlign

Retrieves the status of the text-alignment flags for the device context.

```
UINT GetTextAlign( ) const;
```

Return Value

The status of the text-alignment flags. The return value is one or more of the following values:

- **TA_BASELINE** Specifies alignment of the x-axis and the baseline of the chosen font within the bounding rectangle.
- **TA_BOTTOM** Specifies alignment of the x-axis and the bottom of the bounding rectangle.
- **TA_CENTER** Specifies alignment of the y-axis and the center of the bounding rectangle.
- **TA_LEFT** Specifies alignment of the y-axis and the left side of the bounding rectangle.
- **TA_NOUPDATECP** Specifies that the current position is not updated.
- **TA_RIGHT** Specifies alignment of the y-axis and the right side of the bounding rectangle.
- **TA_TOP** Specifies alignment of the x-axis and the top of the bounding rectangle.
- **TA_UPDATECP** Specifies that the current position is updated.

Remarks

The text-alignment flags determine how the **TextOut** and **ExtTextOut** member functions align a string of text in relation to the string's starting point. The text-alignment flags are not necessarily single-bit flags and may be equal to 0. To test whether a flag is set, an application should follow these steps:

1. Apply the bitwise OR operator to the flag and its related flags, grouped as follows:
   - **TA_LEFT**, **TA_CENTER**, and **TA_RIGHT**
   - **TA_BASELINE**, **TA_BOTTOM**, and **TA_TOP**
   - **TA_NOUPDATECP** and **TA_UPDATECP**
2. Apply the bitwise-AND operator to the result and the return value of **GetTextAlign**.
3. Test for the equality of this result and the flag.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::ExtTextOut
CDC::SetTextAlign
CDC::TextOut
GetTextAlign

Other Resources

CDC Members

CDC::GetTextCharacterExtra

Retrieves the current setting for the amount of intercharacter spacing.

```
int GetTextCharacterExtra( ) const;
```

Return Value

The amount of the intercharacter spacing.

Remarks

GDI adds this spacing to each character, including break characters, when it writes a line of text to the device context.

The default value for the amount of intercharacter spacing is 0.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetTextCharacterExtra
GetTextCharacterExtra

Other Resources

CDC Members

CDC::GetTextColor

Retrieves the current text color.

```
COLORREF GetTextColor( ) const;
```

Return Value

The current text color as an RGB color value.

Remarks

The text color is the foreground color of characters drawn by using the GDI text-output member functions TextOut, ExtTextOut, and TabbedTextOut.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetBkColor
CDC::GetBkMode
CDC::SetBkMode
CDC::SetTextColor
GetTextColor

Other Resources

CDC Members

CDC::GetTextExtent

Call this member function to compute the width and height of a line of text using the current font to determine the dimensions.

```
CSize GetTextExtent(
   LPCTSTR lpszString,
   int nCount
) const;
CSize GetTextExtent(
   const CString& str
) const;
```

Parameters

lpszString

Points to a string of characters. You can also pass a CString object for this parameter.

nCount

Specifies the number of characters in the string.

str

A **CString** object that contains the specified characters.

Return Value

The dimensions of the string (in logical units) in a CSize object.

Remarks

The information is retrieved from m_hAttribDC, the attribute device context.

By default, **GetTextExtent** assumes the text for which it retrieves the dimension is set along a horizontal line (that is, the escapement is 0). If you create a font specifying a non-zero escapement, you must convert the angle of the text explicitly to get the dimensions of the string.

The current clipping region does not affect the width and height returned by **GetTextExtent**.

Since some devices do not place characters in regular cell arrays (that is, they carry out kerning), the sum of the extents of the characters in a string may not be equal to the extent of the string.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetTabbedTextExtent
CDC::m_hAttribDC
CDC::m_hDC
CDC::GetOutputTextExtent
CDC::SetTextJustification
CSize Class

Other Resources

CDC Members

CDC::GetTextExtentExPointI

Retrieves the number of characters in a specified string that will fit within a specified space and fills an array with the text extent for each of those characters.

```
BOOL GetTextExtentExPointI(
   LPWORD pgiIn,
   int cgi,
   int nMaxExtent,
   LPINT lpnFit,
   LPINT alpDx,
   LPSIZE lpSize
) const;
```

Parameters

pgiIn

A pointer to an array of glyph indices for which extents are to be retrieved.

cgi

Specifies the number of glyphs in the array pointed to by *pgiIn*.

nMaxExtent

Specifies the maximum allowable width, in logical units, of the formatted string.

lpnFit

A pointer to an integer that receives a count of the maximum number of characters that will fit in the space specified by *nMaxExtent*. When *lpnFit* is **NULL**, *nMaxExtent* is ignored.

alpDx

A pointer to an array of integers that receives partial glyph extents. Each element in the array gives the distance, in logical units, between the beginning of the glyph indices array and one of the glyphs that fits in the space specified by *nMaxExtent*. Although this array should have at least as many elements as glyph indices specified by *cgi*, the function fills the array with extents only for as many glyph indices as are specified by *lpnFit*. If *lpnDx* is **NULL**, the function does not compute partial string widths.

lpSize

Pointer to a SIZE structure that receives the dimensions of the glyph indices array, in logical units. This value cannot be **NULL**.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

This member function emulates the functionality of the function GetTextExtentExPointI, as described in the *Platform SDK*.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetTextExtentPointI

Other Resources

CDC Members

CDC::GetTextExtentPointI

Retrieves the width and height of the specified array of glyph indices.

```
BOOL GetTextExtentPointI(
   LPWORD pgiIn,
   int cgi,
   LPSIZE lpSize
) const;
```

Parameters

pgiIn

A pointer to an array of glyph indices for which extents are to be retrieved.

cgi

Specifies the number of glyphs in the array pointed to by *pgiIn*.

lpSize

Pointer to a SIZE structure that receives the dimensions of the glyph indices array, in logical units. This value cannot be **NULL**.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

This member function emulates the functionality of the function GetTextExtentPointI, as described in the *Platform SDK*.

See Also

Reference

CDC Class

Hierarchy Chart

CDC::GetTextExtentExPointI

Other Resources

CDC Members

CDC::GetTextFace

Call this member function to copy the typeface name of the current font into a buffer.

```
int GetTextFace(
   int nCount,
   LPTSTR lpszFacename
) const;
int GetTextFace(
   CString& rString
) const;
```

Parameters

nCount

Specifies the size of the buffer (in bytes). If the typeface name is longer than the number of bytes specified by this parameter, the name is truncated.

lpszFacename

Points to the buffer for the typeface name.

rString

A reference to a CString object.

Return Value

The number of bytes copied to the buffer, not including the terminating null character. It is 0 if an error occurs.

Remarks

The typeface name is copied as a null-terminated string.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetTextMetrics
CDC::SetTextAlign
CDC::TextOut
GetTextFace

Other Resources

CDC Members

CDC::GetTextMetrics

Retrieves the metrics for the current font using the attribute device context.

```
BOOL GetTextMetrics(
   LPTEXTMETRIC lpMetrics
) const;
```

Parameters

lpMetrics

Points to the TEXTMETRIC structure that receives the metrics.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetTextAlign
CDC::m_hAttribDC
CDC::m_hDC
CDC::GetOutputTextMetrics
CDC::GetTextExtent
CDC::GetTextFace
CDC::SetTextJustification
GetTextMetrics

Other Resources

CDC Members

CDC::GetViewportExt

Retrieves the x- and y-extents of the device context's viewport.

```
CSize GetViewportExt( ) const;
```

Return Value

The x- and y-extents (in device units) as a **CSize** object.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetViewportExt
CSize Class
CDC::SetWindowExt

Other Resources

CDC Members

CDC::GetViewportOrg

Retrieves the x- and y-coordinates of the origin of the viewport associated with the device context.

```
CPoint GetViewportOrg( ) const;
```

Return Value

The origin of the viewport (in device coordinates) as a **CPoint** object.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetWindowOrg
CPoint Class
CDC::SetViewportOrg

Other Resources

CDC Members

CDC::GetWindow

Returns the window associated with the display device context.

```
CWnd* GetWindow( ) const;
```

Return Value

Pointer to a **CWnd** object if successful; otherwise **NULL**.

Remarks

This is an advanced function. For example, this member function may not return the view window when printing or in print preview. It always returns the window associated with output. Output functions that use the given DC draw into this window.

See Also

Reference

CDC Class
Hierarchy Chart
CWnd::GetDC
CWnd::GetWindowDC
GetWindow

Other Resources

CDC Members

CDC::GetWindowExt

Retrieves the x- and y-extents of the window associated with the device context.

```
CSize GetWindowExt( ) const;
```

Return Value

The x- and y-extents (in logical units) as a **CSize** object.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetWindowExt
CSize Class
CDC::GetViewportExt

Other Resources

CDC Members

CDC::GetWindowOrg

Retrieves the x- and y-coordinates of the origin of the window associated with the device context.

```
CPoint GetWindowOrg( ) const;
```

Return Value

The origin of the window (in logical coordinates) as a **CPoint** object.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetViewportOrg
CDC::SetWindowOrg
CPoint Class

Other Resources

CDC Members

CDC::GetWorldTransform

Retrieves the current world-space to page-space transformation.

```
BOOL GetWorldTransform(
   XFORM& rXform
) const;
```

Parameters

rXform

Reference to an XFORM structure that receives the current world-space to page-space transformation.

Return Value

Returns a nonzero value on success.

Returns 0 on failure.

In Windows NT, Windows 2000, and Windows XP, to get extended error information, call GetLastError.

Remarks

This method wraps the Windows GDI function GetWorldTransform.

See Also

Reference

CDC Class

Hierarchy Chart

Other Resources

CDC Members

CDC::GradientFill

Call this member function to fill rectangle and triangle structures with color that smoothly fades from one side to the other.

```
BOOL GradientFill(
   TRIVERTEX* pVertices,
   ULONG nVertices,
   void* pMesh,
   ULONG nMeshElements,
   DWORD dwMode
);
```

Parameters

pVertices

Pointer to an array of TRIVERTEX structures that each define a triangle vertex.

nVertices

The number of vertices.

pMesh

Array of GRADIENT_TRIANGLE structures in triangle mode, or an array of GRADIENT_RECT structures in rectangle mode.

nMeshElements

The number of elements (triangles or rectangles) in *pMesh*.

dwMode

Specifies gradient fill mode. For a list of possible values, see GradientFill in the *Platform SDK*.

Return Value

TRUE if successful; otherwise **FALSE**.

Remarks

For more information, see **GradientFill** in the *Platform SDK*.

See Also

Reference

CDC Class
Hierarchy Chart
EMRGRADIENTFILL

Other Resources

CDC Members

CDC::GrayString

Draws dimmed (gray) text at the given location by writing the text in a memory bitmap, dimming the bitmap, and then copying the bitmap to the display.

```
virtual BOOL GrayString(
   CBrush* pBrush,
   BOOL ( CALLBACK* lpfnOutput )( HDC, LPARAM, int ),
   LPARAM lpData,
   int nCount,
   int x,
   int y,
   int nWidth,
   int nHeight
);
```

Parameters

pBrush

Identifies the brush to be used for dimming (graying).

lpfnOutput

Specifies the procedure-instance address of the application-supplied callback function that will draw the string. For more information, see the description of the Windows **OutputFunc** callback function. If this parameter is **NULL**, the system uses the Windows **TextOut** function to draw the string, and *lpData* is assumed to be a long pointer to the character string to be output.

lpData

Specifies a far pointer to data to be passed to the output function. If *lpfnOutput* is **NULL**, *lpData* must be a long pointer to the string to be output.

nCount

Specifies the number of characters to be output. If this parameter is 0, **GrayString** calculates the length of the string (assuming that *lpData* is a pointer to the string). If *nCount* is –1 and the function pointed to by *lpfnOutput* returns 0, the image is shown but not dimmed.

x

Specifies the logical x-coordinate of the starting position of the rectangle that encloses the string.

y

Specifies the logical y-coordinate of the starting position of the rectangle that encloses the string.

nWidth

Specifies the width (in logical units) of the rectangle that encloses the string. If *nWidth* is 0, **GrayString** calculates the width of the area, assuming *lpData* is a pointer to the string.

nHeight

Specifies the height (in logical units) of the rectangle that encloses the string. If *nHeight* is 0, **GrayString** calculates the height of the area, assuming *lpData* is a pointer to the string.

Return Value

Nonzero if the string is drawn, or 0 if either the **TextOut** function or the application-supplied output function returned 0, or if there was insufficient memory to create a memory bitmap for dimming.

Remarks

The function dims the text regardless of the selected brush and background. The **GrayString** member function uses the currently selected font. The **MM_TEXT** mapping mode must be selected before using this function.

An application can draw dimmed (grayed) strings on devices that support a solid gray color without calling the **GrayString**

member function. The system color **COLOR_GRAYTEXT** is the solid-gray system color used to draw disabled text. The application can call the **GetSysColor** Windows function to retrieve the color value of **COLOR_GRAYTEXT**. If the color is other than 0 (black), the application can call the **SetTextColor** member function to set the text color to the color value and then draw the string directly. If the retrieved color is black, the application must call **GrayString** to dim (gray) the text.

If *lpfnOutput* is **NULL**, GDI uses the Windows TextOut function, and *lpData* is assumed to be a far pointer to the character to be output. If the characters to be output cannot be handled by the **TextOut** member function (for example, the string is stored as a bitmap), the application must supply its own output function.

Also note that all callback functions must trap Microsoft Foundation exceptions before returning to Windows, since exceptions cannot be thrown across callback boundaries. For more information about exceptions, see the article Exceptions.

The callback function passed to **GrayString** must use the **__stdcall** calling convention and must be exported with **__declspec**.

When the framework is in preview mode, a call to the **GrayString** member function is translated to a **TextOut** call, and the callback function is not called.

See Also

Reference

CDC Class
Hierarchy Chart
GetSysColor
CDC::SetTextColor
CDC::TextOut
GrayString

Other Resources

CDC Members

CDC::HIMETRICtoDP

Use this function when you convert **HIMETRIC** sizes from OLE to pixels.

```
void HIMETRICtoDP(
   LPSIZE lpSize
) const;
```

Parameters

lpSize

Points to a SIZE structure or CSize object.

Remarks

If the mapping mode of the device context object is **MM_LOENGLISH**, **MM_HIENGLISH**, **MM_LOMETRIC** or **MM_HIMETRIC**, then the conversion is based on the number of pixels in the physical inch. If the mapping mode is one of the other non-constrained modes (e.g., **MM_TEXT**), then the conversion is based on the number of pixels in the logical inch.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::LPtoDP
CDC::HIMETRICtoLP

Other Resources

CDC Members

CDC::HIMETRICtoLP

Call this function to convert **HIMETRIC** units into logical units.

```
void HIMETRICtoLP(
   LPSIZE lpSize
) const;
```

Parameters

lpSize

Points to a SIZE structure or CSize object.

Remarks

Use this function when you get **HIMETRIC** sizes from OLE and wish to convert them to your application's natural mapping mode.

The conversion is accomplished by first converting the **HIMETRIC** units into pixels and then converting these units into logical units using the device context's current mapping units. Note that the extents of the device's window and viewport will affect the result.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::HIMETRICtoDP
CDC::DPtoLP

Other Resources

CDC Members

CDC::IntersectClipRect

Creates a new clipping region by forming the intersection of the current region and the rectangle specified by *x1*, *y1*, *x2*, and *y2*.

```
int IntersectClipRect(
   int x1,
   int y1,
   int x2,
   int y2
);
int IntersectClipRect(
   LPCRECT lpRect
);
```

Parameters

x1

Specifies the logical x-coordinate of the upper-left corner of the rectangle.

y1

Specifies the logical y-coordinate of the upper-left corner of the rectangle.

x2

Specifies the logical x-coordinate of the lower-right corner of the rectangle.

y2

Specifies the logical y-coordinate of the lower-right corner of the rectangle.

lpRect

Specifies the rectangle. You can pass either a **CRect** object or a pointer to a **RECT** structure for this parameter.

Return Value

The new clipping region's type. It can be any one of the following values:

- **COMPLEXREGION** New clipping region has overlapping borders.
- **ERROR** Device context is not valid.
- **NULLREGION** New clipping region is empty.
- **SIMPLEREGION** New clipping region has no overlapping borders.

Remarks

GDI clips all subsequent output to fit within the new boundary. The width and height must not exceed 32,767.

See Also

Reference

CDC Class
Hierarchy Chart
IntersectClipRect
CRect Class
RECT Structure

Other Resources

CDC Members

CDC::InvertRect

Inverts the contents of the given rectangle.

```
void InvertRect(
   LPCRECT lpRect
);
```

Parameters

lpRect

Points to a **RECT** that contains the logical coordinates of the rectangle to be inverted. You can also pass a **CRect** object for this parameter.

Remarks

Inversion is a logical NOT operation and flips the bits of each pixel. On monochrome displays, the function makes white pixels black and black pixels white. On color displays, the inversion depends on how colors are generated for the display. Calling **InvertRect** twice with the same rectangle restores the display to its previous colors.

If the rectangle is empty, nothing is drawn.

Example

```
// invert rect from 20,20 to 50,50
CRect rect(20, 20, 50, 50);
pDC->InvertRect(rect);

// inverting again restores to normal
::Sleep(1000);
pDC->InvertRect(rect);
```

See Also

Reference

CDC Class
Hierarchy Chart
CDC::FillRect
InvertRect
CRect Class
RECT Structure

Other Resources

CDC Members

CDC::InvertRgn

Inverts the colors in the region specified by *pRgn*.

```
BOOL InvertRgn(
   CRgn* pRgn
);
```

Parameters

pRgn

Identifies the region to be inverted. The coordinates for the region are specified in logical units.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

On monochrome displays, the function makes white pixels black and black pixels white. On color displays, the inversion depends on how the colors are generated for the display.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::FillRgn
CDC::PaintRgn
CRgn Class
InvertRgn

Other Resources

CDC Members

CDC::IsPrinting

Determines whether the device context is being used for printing.

```
BOOL IsPrinting( ) const;
```

Return Value

Nonzero if the **CDC** object is a printer DC; otherwise 0.

See Also

Reference

CDC Class

Hierarchy Chart

Other Resources

CDC Members

CDC::LineTo

Draws a line from the current position up to, but not including, the point specified by *x* and *y* (or *point*).

```
BOOL LineTo(
   int x,
   int y
);
BOOL LineTo(
   POINT point
);
```

Parameters

x

Specifies the logical x-coordinate of the endpoint for the line.

y

Specifies the logical y-coordinate of the endpoint for the line.

point

Specifies the endpoint for the line. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Return Value

Nonzero if the line is drawn; otherwise 0.

Remarks

The line is drawn with the selected pen. The current position is set to *x,y* or to *point*.

Example

See the example for CRect::CenterPoint.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::MoveTo
CDC::GetCurrentPosition
LineTo
CPoint Class
POINT Structure

Other Resources

CDC Members

CDC::LPtoDP

Converts logical units into device units.

```
void LPtoDP(
   LPPOINT lpPoints,
   int nCount = 1
) const;
void LPtoDP(
   LPRECT lpRect
) const;
void LPtoDP(
   LPSIZE lpSize
) const;
```

Parameters

lpPoints

Points to an array of points. Each point in the array is a POINT structure or a CPoint object.

nCount

The number of points in the array.

lpRect

Points to a RECT structure or a CRect object. This parameter is used for the common case of mapping a rectangle from logical to device units.

lpSize

Points to a SIZE structure or a CSize object.

Remarks

The function maps the coordinates of each point, or dimensions of a size, from GDI's logical coordinate system into a device coordinate system. The conversion depends on the current mapping mode and the settings of the origins and extents of the device's window and viewport.

The x- and y-coordinates of points are 2-byte signed integers in the range –32,768 through 32,767. In cases where the mapping mode would result in values larger than these limits, the system sets the values to –32,768 and 32,767, respectively.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::DPtoLP
CDC::HIMETRICtoLP
LPtoDP
CDC::GetWindowOrg
CDC::GetWindowExt

Other Resources

CDC Members

CDC::LPtoHIMETRIC

Call this function to convert logical units into **HIMETRIC** units.

```
void LPtoHIMETRIC(
   LPSIZE lpSize
) const;
```

Parameters

lpSize

Points to a **SIZE** structure or a **CSize** object.

Remarks

Use this function when you give **HIMETRIC** sizes to OLE, converting from your application's natural mapping mode. Note that the extents of the device's window and viewport will affect the result.

The conversion is accomplished by first converting the logical units into pixels using the device context's current mapping units and then converting these units into **HIMETRIC** units.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::HIMETRICtoLP
CDC::LPtoDP
CDC::DPtoHIMETRIC

Other Resources

CDC Members

CDC::MaskBlt

Combines the color data for the source and destination bitmaps using the given mask and raster operation.

```
BOOL MaskBlt(
   int x,
   int y,
   int nWidth,
   int nHeight,
   CDC* pSrcDC,
   int xSrc,
   int ySrc,
   CBitmap& maskBitmap,
   int xMask,
   int yMask,
   DWORD dwRop
);
```

Parameters

x

Specifies the logical x-coordinate of the upper-left corner of the destination rectangle.

y

Specifies the logical y-coordinate of the upper-left corner of the destination rectangle.

nWidth

Specifies the width, in logical units, of the destination rectangle and source bitmap.

nHeight

Specifies the height, in logical units, of the destination rectangle and source bitmap.

pSrcDC

Identifies the device context from which the bitmap is to be copied. It must be zero if the *dwRop* parameter specifies a raster operation that does not include a source.

xSrc

Specifies the logical x-coordinate of the upper-left corner of the source bitmap.

ySrc

Specifies the logical y-coordinate of the upper-left corner of the source bitmap.

maskBitmap

Identifies the monochrome mask bitmap combined with the color bitmap in the source device context.

xMask

Specifies the horizontal pixel offset for the mask bitmap specified by the *maskBitmap* parameter.

yMask

Specifies the vertical pixel offset for the mask bitmap specified by the *maskBitmap* parameter.

dwRop

Specifies both foreground and background ternary raster operation codes, which the function uses to control the combination of source and destination data. The background raster operation code is stored in the high byte of the high word of this value; the foreground raster operation code is stored in the low byte of the high word of this value; the low word of this value is ignored, and should be zero. The macro **MAKEROP4** creates such combinations of foreground and background raster operation codes. See the Remarks section for a discussion of foreground and background in the context of this function. See the **BitBlt** member function for a list of common raster operation codes.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

A value of 1 in the mask specified by *maskBitmap* indicates that the foreground raster operation code specified by *dwRop* should be applied at that location. A value of 0 in the mask indicates that the background raster operation code specified by *dwRop* should be applied at that location. If the raster operations require a source, the mask rectangle must cover the source rectangle. If it does not, the function will fail. If the raster operations do not require a source, the mask rectangle must cover the destination rectangle. If it does not, the function will fail.

If a rotation or shear transformation is in effect for the source device context when this function is called, an error occurs. However, other types of transformations are allowed.

If the color formats of the source, pattern, and destination bitmaps differ, this function converts the pattern or source format, or both, to match the destination format. If the mask bitmap is not a monochrome bitmap, an error occurs. When an enhanced metafile is being recorded, an error occurs (and the function returns 0) if the source device context identifies an enhanced-metafile device context. Not all devices support **MaskBlt**. An application should call **GetDeviceCaps** to determine whether a device supports this function. If no mask bitmap is supplied, this function behaves exactly like **BitBlt**, using the foreground raster operation code. The pixel offsets in the mask bitmap map to the point (0,0) in the source device context's bitmap. This is useful for cases in which a mask bitmap contains a set of masks; an application can easily apply any one of them to a mask-blitting task by adjusting the pixel offsets and rectangle sizes sent to **MaskBlt**.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::BitBlt
CDC::GetDeviceCaps
CDC::PlgBlt
CDC::StretchBlt
MaskBlt

Other Resources

CDC Members

CDC::ModifyWorldTransform

Changes the world transformation for a device context using the specified mode.

```
BOOL ModifyWorldTransform(
   const XFORM& rXform,
   DWORD iMode
);
```

Parameters

rXform

Reference to an XFORM structure used to modify the world transformation for the given device context.

iMode

Specifies how the transformation data modifies the current world transformation. For a list of the values that this parameter can take, see ModifyWorldTransform.

Return Value

Returns a nonzero value on success.

Returns 0 on failure.

In Windows NT, Windows 2000, and Windows XP, to get extended error information, call GetLastError.

Remarks

This method wraps the Windows GDI function ModifyWorldTransform.

See Also

Reference

CDC Class

Hierarchy Chart

Other Resources

CDC Members

CDC::MoveTo

Moves the current position to the point specified by *x* and *y* (or by *point*).

```
CPoint MoveTo(
   int x,
   int y
);
CPoint MoveTo(
   POINT point
);
```

Parameters

x

Specifies the logical x-coordinate of the new position.

y

Specifies the logical y-coordinate of the new position.

point

Specifies the new position. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Return Value

The x- and y-coordinates of the previous position as a **CPoint** object.

Example

See the example for CRect::CenterPoint.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetCurrentPosition
CDC::LineTo
CPoint Class
POINT Structure

Other Resources

CDC Members

CDC::OffsetClipRgn

Moves the clipping region of the device context by the specified offsets.

```
int OffsetClipRgn(
   int x,
   int y
);
int OffsetClipRgn(
   SIZE size
);
```

Parameters

x

Specifies the number of logical units to move left or right.

y

Specifies the number of logical units to move up or down.

size

Specifies the amount to offset.

Return Value

The new region's type. It can be any one of the following values:

- **COMPLEXREGION** Clipping region has overlapping borders.
- **ERROR** Device context is not valid.
- **NULLREGION** Clipping region is empty.
- **SIMPLEREGION** Clipping region has no overlapping borders.

Remarks

The function moves the region *x* units along the x-axis and *y* units along the y-axis.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SelectClipRgn
OffsetClipRgn

Other Resources

CDC Members

CDC::OffsetViewportOrg

Modifies the coordinates of the viewport origin relative to the coordinates of the current viewport origin.

```
virtual CPoint OffsetViewportOrg(
   int nWidth,
   int nHeight
);
```

Parameters

nWidth

Specifies the number of device units to add to the current origin's x-coordinate.

nHeight

Specifies the number of device units to add to the current origin's y-coordinate.

Return Value

The previous viewport origin (in device coordinates) as a **CPoint** object.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetViewportOrg
CDC::OffsetWindowOrg
CDC::SetViewportOrg
CPoint Class

Other Resources

CDC Members

CDC::OffsetWindowOrg

Modifies the coordinates of the window origin relative to the coordinates of the current window origin.

```
CPoint OffsetWindowOrg(
   int nWidth,
   int nHeight
);
```

Parameters

nWidth

Specifies the number of logical units to add to the current origin's x-coordinate.

nHeight

Specifies the number of logical units to add to the current origin's y-coordinate.

Return Value

The previous window origin (in logical coordinates) as a **CPoint** object.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetWindowOrg
CDC::OffsetViewportOrg
CDC::SetWindowOrg
CPoint Class

Other Resources

CDC Members

CDC::PaintRgn

Fills the region specified by *pRgn* using the current brush.

```
BOOL PaintRgn(
   CRgn* pRgn
);
```

Parameters

pRgn

Identifies the region to be filled. The coordinates for the given region are specified in logical units.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

Reference

CDC Class
Hierarchy Chart
CBrush Class
CDC::SelectObject
CDC::FillRgn
PaintRgn
CRgn Class

Other Resources

CDC Members

CDC::PatBlt

Creates a bit pattern on the device.

```
BOOL PatBlt(
   int x,
   int y,
   int nWidth,
   int nHeight,
   DWORDdwRop
);
```

Parameters

x

Specifies the logical x-coordinate of the upper-left corner of the rectangle that is to receive the pattern.

y

Specifies the logical y-coordinate of the upper-left corner of the rectangle that is to receive the pattern.

nWidth

Specifies the width (in logical units) of the rectangle that is to receive the pattern.

nHeight

Specifies the height (in logical units) of the rectangle that is to receive the pattern.

dwRop

Specifies the raster-operation code. Raster-operation codes (ROPs) define how GDI combines colors in output operations that involve a current brush, a possible source bitmap, and a destination bitmap. This parameter can be one of the following values:

- **PATCOPY** Copies pattern to destination bitmap.
- **PATINVERT** Combines destination bitmap with pattern using the Boolean XOR operator.
- **DSTINVERT** Inverts the destination bitmap.
- **BLACKNESS** Turns all output black.
- **WHITENESS** Turns all output white.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The pattern is a combination of the selected brush and the pattern already on the device. The raster-operation code specified by *dwRop* defines how the patterns are to be combined. The raster operations listed for this function are a limited subset of the full 256 ternary raster-operation codes; in particular, a raster-operation code that refers to a source cannot be used.

Not all device contexts support the **PatBlt** function. To determine whether a device context supports **PatBlt**, call the **GetDeviceCaps** member function with the **RASTERCAPS** index and check the return value for the **RC_BITBLT** flag.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetDeviceCaps
PatBlt

Other Resources

CDC Members

CDC::Pie

Draws a pie-shaped wedge by drawing an elliptical arc whose center and two endpoints are joined by lines.

```
BOOL Pie(
   int x1,
   int y1,
   int x2,
   int y2,
   int x3,
   int y3,
   int x4,
   int y4
);
BOOL Pie(
   LPCRECT lpRect,
      POINT ptStart,
      POINT ptEnd
);
```

Parameters

x1

Specifies the x-coordinate of the upper-left corner of the bounding rectangle (in logical units).

y1

Specifies the y-coordinate of the upper-left corner of the bounding rectangle (in logical units).

x2

Specifies the x-coordinate of the lower-right corner of the bounding rectangle (in logical units).

y2

Specifies the y-coordinate of the lower-right corner of the bounding rectangle (in logical units).

x3

Specifies the x-coordinate of the arc's starting point (in logical units). This point does not have to lie exactly on the arc.

y3

Specifies the y-coordinate of the arc's starting point (in logical units). This point does not have to lie exactly on the arc.

x4

Specifies the x-coordinate of the arc's endpoint (in logical units). This point does not have to lie exactly on the arc.

y4

Specifies the y-coordinate of the arc's endpoint (in logical units). This point does not have to lie exactly on the arc.

lpRect

Specifies the bounding rectangle. You can pass either a **CRect** object or a pointer to a **RECT** structure for this parameter.

ptStart

Specifies the starting point of the arc. This point does not have to lie exactly on the arc. You can pass either a POINT structure or a CPoint object for this parameter.

ptEnd

Specifies the endpoint of the arc. This point does not have to lie exactly on the arc. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The center of the arc is the center of the bounding rectangle specified by *x1*, *y1*, *x2*, and *y2* (or by *lpRect*). The starting and ending points of the arc are specified by *x3*, *y3*, *x4*, and *y4* (or by *ptStart* and *ptEnd*).

The arc is drawn with the selected pen, moving in a counterclockwise direction. Two additional lines are drawn from each endpoint to the arc's center. The pie-shaped area is filled with the current brush. If *x3* equals *x4* and *y3* equals *y4*, the result is an ellipse with a single line from the center of the ellipse to the point (*x3*, *y3*) or (*x4*, *y4*).

The figure drawn by this function extends up to but does not include the right and bottom coordinates. This means that the height of the figure is *y2* – *y1* and the width of the figure is *x2* – *x1*. Both the width and the height of the bounding rectangle must be greater than 2 units and less than 32,767 units.

Example

```
void CCurvesView::OnDraw(CDC* pDC)
{
   // Fill the client area with a simple pie chart. A
   // big blue slice covers 75% of the pie, from
   // 6 o'clock to 3 o'clock. This portion is filled
   // with blue and has a blue edge. The remaining 25%
   // is filled with a red, diagonal hatch and has
   // a red edge.

   // Get the client area.
   CRect rectClient;
   GetClientRect(rectClient);

   // Make a couple of pens and similar brushes.
   CPen penBlue, penRed;
   CBrush brushBlue, brushRed;
   CBrush* pOldBrush;
   CPen* pOldPen;

   brushBlue.CreateSolidBrush(RGB(0, 0, 255));
   brushRed.CreateHatchBrush(HS_FDIAGONAL, RGB(255, 0, 0));
   penBlue.CreatePen(PS_SOLID | PS_COSMETIC, 1, RGB(0, 0, 255));
   penRed.CreatePen(PS_SOLID | PS_COSMETIC, 1, RGB(255, 0, 0));

   // Draw from 3 o'clock to 6 o'clock, counterclockwise,
   // in a blue pen with a solid blue fill.

   pOldPen = pDC->SelectObject(&penBlue);
   pOldBrush = pDC->SelectObject(&brushBlue);

   pDC->Pie(rectClient,
      CPoint(rectClient.right, rectClient.CenterPoint().y),
      CPoint(rectClient.CenterPoint().x, rectClient.right));

   // Draw the remaining quarter slice from 6 o'clock
   // to 3 o'clock, counterclockwise, in a red pen with
   // the hatched brush.
   pDC->SelectObject(&penRed);
   pDC->SelectObject(&brushRed);

   // Same parameters, but reverse start and end points.
   pDC->Pie(rectClient,
      CPoint(rectClient.CenterPoint().x, rectClient.right),
      CPoint(rectClient.right, rectClient.CenterPoint().y));

   // Restore the previous pen.
   pDC->SelectObject(pOldPen);
}
```

See Also

Reference

CDC Class

Hierarchy Chart

CDC::Chord
Pie
RECT Structure
POINT Structure
CRect Class
CPoint Class
Other Resources
CDC Members

CDC::PlayMetaFile

Plays the contents of the specified metafile on the device context.

```
BOOL PlayMetaFile(
   HMETAFILE hMF
);
BOOL PlayMetaFile(
   HENHMETAFILE hEnhMetaFile,
   LPCRECT lpBounds
);
```

Parameters

hMF

Identifies the metafile to be played.

hEnhMetaFile

Identifies the enhanced metafile.

lpBounds

Points to a **RECT** structure or a **CRect** object that contains the coordinates of the bounding rectangle used to display the picture. The coordinates are specified in logical units.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The metafile can be played any number of times.

The second version of **PlayMetaFile** displays the picture stored in the given enhanced-format metafile. When an application calls the second version of **PlayMetaFile**, Windows uses the picture frame in the enhanced-metafile header to map the picture onto the rectangle pointed to by the *lpBounds* parameter. (This picture may be sheared or rotated by setting the world transform in the output device before calling **PlayMetaFile**.) Points along the edges of the rectangle are included in the picture. An enhanced-metafile picture can be clipped by defining the clipping region in the output device before playing the enhanced metafile.

If an enhanced metafile contains an optional palette, an application can achieve consistent colors by setting up a color palette on the output device before calling the second version of **PlayMetaFile**. To retrieve the optional palette, use the **GetEnhMetaFilePaletteEntries** Windows function. An enhanced metafile can be embedded in a newly created enhanced metafile by calling the second version of **PlayMetaFile** and playing the source enhanced metafile into the device context for the new enhanced metafile.

The states of the output device context are preserved by this function. Any object created but not deleted in the enhanced metafile is deleted by this function. To stop this function, an application can call the **CancelDC** Windows function from another thread to terminate the operation. In this case, the function returns zero.

See Also

Reference

CDC Class
Hierarchy Chart
CancelDC
GetEnhMetaFileHeader
GetEnhMetaFilePaletteEntries
SetWorldTransform
PlayMetaFile
PlayEnhMetaFile
PlayMetaFile

Other Resources

CDC Members

CDC::PlgBlt

Performs a bit-block transfer of the bits of color data from the specified rectangle in the source device context to the specified parallelogram in the given device context.

```
BOOL PlgBlt(
   LPPOINT lpPoint,
   CDC* pSrcDC,
   int xSrc,
   int ySrc,
   int nWidth,
   int nHeight,
   CBitmap& maskBitmap,
   int xMask,
   int yMask
);
```

Parameters

lpPoint

Points to an array of three points in logical space that identifies three corners of the destination parallelogram. The upper-left corner of the source rectangle is mapped to the first point in this array, the upper-right corner to the second point in this array, and the lower-left corner to the third point. The lower-right corner of the source rectangle is mapped to the implicit fourth point in the parallelogram.

pSrcDC

Identifies the source device context.

xSrc

Specifies the x-coordinate, in logical units, of the upper-left corner of the source rectangle.

ySrc

Specifies the y-coordinate, in logical units, of the upper-left corner of the source rectangle.

nWidth

Specifies the width, in logical units, of the source rectangle.

nHeight

Specifies the height, in logical units, of the source rectangle.

maskBitmap

Identifies an optional monochrome bitmap that is used to mask the colors of the source rectangle.

xMask

Specifies the x-coordinate of the upper-left corner of the monochrome bitmap.

yMask

Specifies the y-coordinate of the upper-left corner of the monochrome bitmap.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

If the given bitmask handle identifies a valid monochrome bitmap, the function uses this bitmap to mask the bits of color data from the source rectangle.

The fourth vertex of the parallelogram (D) is defined by treating the first three points (A, B, and C) as vectors and computing D = B + C - A.

If the bitmask exists, a value of 1 in the mask indicates that the source pixel color should be copied to the destination. A value of 0 in the mask indicates that the destination pixel color is not to be changed.

If the mask rectangle is smaller than the source and destination rectangles, the function replicates the mask pattern.

Scaling, translation, and reflection transformations are allowed in the source device context; however, rotation and shear transformations are not. If the mask bitmap is not a monochrome bitmap, an error occurs. The stretching mode for the destination device context is used to determine how to stretch or compress the pixels, if that is necessary. When an enhanced metafile is being recorded, an error occurs if the source device context identifies an enhanced-metafile device context.

The destination coordinates are transformed according to the destination device context; the source coordinates are transformed according to the source device context. If the source transformation has a rotation or shear, an error is returned. If the destination and source rectangles do not have the same color format, **PlgBlt** converts the source rectangle to match the destination rectangle. Not all devices support **PlgBlt**. For more information, see the description of the **RC_BITBLT** raster capability in the **CDC::GetDeviceCaps** member function.

If the source and destination device contexts represent incompatible devices, **PlgBlt** returns an error.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::BitBlt
CDC::GetDeviceCaps
CDC::MaskBlt
CDC::StretchBlt
SetStretchBltMode
PlgBlt

Other Resources

CDC Members

CDC::PolyBezier

Draws one or more Bzier splines.

```
BOOL PolyBezier(
   const POINT* lpPoints,
   int nCount
);
```

Parameters

lpPoints

Points to an array of POINT data structures that contain the endpoints and control points of the spline(s).

nCount

Specifies the number of points in the *lpPoints* array. This value must be one more than three times the number of splines to be drawn, because each Bzier spline requires two control points and an endpoint, and the initial spline requires an additional starting point.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

This function draws cubic Bzier splines by using the endpoints and control points specified by the *lpPoints* parameter. The first spline is drawn from the first point to the fourth point by using the second and third points as control points. Each subsequent spline in the sequence needs exactly three more points: the end point of the previous spline is used as the starting point, the next two points in the sequence are control points, and the third is the end point.

The current position is neither used nor updated by the **PolyBezier** function. The figure is not filled. This function draws lines by using the current pen.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::PolyBezierTo
PolyBezier

Other Resources

CDC Members

CDC::PolyBezierTo

Draws one or more Bzier splines.

```
BOOL PolyBezierTo(
   const POINT* lpPoints,
   int nCount
);
```

Parameters

lpPoints

Points to an array of POINT data structures that contains the endpoints and control points.

nCount

Specifies the number of points in the *lpPoints* array. This value must be three times the number of splines to be drawn, because each Bzier spline requires two control points and an end point.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

This function draws cubic Bzier splines by using the control points specified by the *lpPoints* parameter. The first spline is drawn from the current position to the third point by using the first two points as control points. For each subsequent spline, the function needs exactly three more points, and uses the end point of the previous spline as the starting point for the next. **PolyBezierTo** moves the current position to the end point of the last Bzier spline. The figure is not filled. This function draws lines by using the current pen.

Example

See the example for CDC::BeginPath.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::MoveTo
CDC::PolyBezier
PolyBezierTo

Other Resources

CDC Members

CDC::PolyDraw

Draws a set of line segments and Bzier splines.

```
BOOL PolyDraw(
   const POINT* lpPoints,
   const BYTE* lpTypes,
   int nCount
);
```

Parameters

lpPoints

Points to an array of POINT data structures that contains the endpoints for each line segment and the endpoints and control points for each Bzier spline.

lpTypes

Points to an array that specifies how each point in the *lpPoints* array is used. Values can be one of the following:

- **PT_MOVETO** Specifies that this point starts a disjoint figure. This point becomes the new current position.
- **PT_LINETO** Specifies that a line is to be drawn from the current position to this point, which then becomes the new current position.
- **PT_BEZIERTO** Specifies that this point is a control point or ending point for a Bzier spline.

  PT_BEZIERTO types always occur in sets of three. The current position defines the starting point for the Bzier spline. The first two **PT_BEZIERTO** points are the control points, and the third **PT_BEZIERTO** point is the ending point. The ending point becomes the new current position. If there are not three consecutive **PT_BEZIERTO** points, an error results.

  A **PT_LINETO** or **PT_BEZIERTO** type can be combined with the following constant by using the bitwise operator OR to indicate that the corresponding point is the last point in a figure and the figure is closed:

- **PT_CLOSEFIGURE** Specifies that the figure is automatically closed after the **PT_LINETO** or **PT_BEZIERTO** type for this point is done. A line is drawn from this point to the most recent **PT_MOVETO** or **MoveTo** point.

  This flag is combined with the **PT_LINETO** type for a line, or with the **PT_BEZIERTO** type of ending point for a Bzier spline, by using the bitwise **OR** operator. The current position is set to the ending point of the closing line.

nCount

Specifies the total number of points in the *lpPoints* array, the same as the number of bytes in the *lpTypes* array.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

This function can be used to draw disjoint figures in place of consecutive calls to **CDC::MoveTo**, **CDC::LineTo**, and **CDC::PolyBezierTo** member functions. The lines and splines are drawn using the current pen, and figures are not filled. If there is an active path started by calling the **CDC::BeginPath** member function, **PolyDraw** adds to the path. The points contained in the *lpPoints* array and in *lpTypes* indicate whether each point is part of a **CDC::MoveTo**, a **CDC::LineTo**, or a **CDC::BezierTo** operation. It is also possible to close figures. This function updates the current position.

Example

See the example for CDC::BeginPath.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::BeginPath

CDC::EndPath
CDC::LineTo
CDC::MoveTo
CDC::PolyBezierTo
CDC::Polyline
PolyDraw

Other Resources

CDC Members

CDC::Polygon

Draws a polygon consisting of two or more points (vertices) connected by lines, using the current pen.

```
BOOL Polygon(
   LPPOINT lpPoints,
   int nCount
);
```

Parameters

lpPoints

Points to an array of points that specifies the vertices of the polygon. Each point in the array is a **POINT** structure or a **CPoint** object.

nCount

Specifies the number of vertices in the array.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The system closes the polygon automatically, if necessary, by drawing a line from the last vertex to the first.

The current polygon-filling mode can be retrieved or set by using the **GetPolyFillMode** and **SetPolyFillMode** member functions.

Example

```
void CMyView::OnDraw(CDC* pDC)
{
   // find the client area
   CRect rect;
   GetClientRect(rect);

   // draw with a thick blue pen
   CPen penBlue(PS_SOLID, 5, RGB(0, 0, 255));
   CPen* pOldPen = pDC->SelectObject(&penBlue);

   // and a solid red brush
   CBrush brushRed(RGB(255, 0, 0));
   CBrush* pOldBrush = pDC->SelectObject(&brushRed);

   // Find the midpoints of the top, right, left, and bottom
   // of the client area. They will be the vertices of our polygon.
   CPoint pts[4];
   pts[0].x = rect.left + rect.Width()/2;
   pts[0].y = rect.top;

   pts[1].x = rect.right;
   pts[1].y = rect.top + rect.Height()/2;

   pts[2].x = pts[0].x;
   pts[2].y = rect.bottom;

   pts[3].x = rect.left;
   pts[3].y = pts[1].y;

   // Calling Polygon() on that array will draw three lines
   // between the points, as well as an additional line to
   // close the shape--from the last point to the first point
   // we specified.
   pDC->Polygon(pts, 4);
```

```
    // Put back the old objects.
    pDC->SelectObject(pOldPen);
    pDC->SelectObject(pOldBrush);
}
```

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetPolyFillMode
CDC::Polyline
CDC::PolyPolygon
CDC::SetPolyFillMode
CPoint Class
Polygon

Other Resources

CDC Members

CDC::Polyline

Draws a set of line segments connecting the points specified by *lpPoints*.

```
BOOL Polyline(
   LPPOINT lpPoints,
   int nCount
);
```

Parameters

lpPoints

Points to an array of **POINT** structures or **CPoint** objects to be connected.

nCount

Specifies the number of points in the array. This value must be at least 2.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The lines are drawn from the first point through subsequent points using the current pen. Unlike the **LineTo** member function, the **Polyline** function neither uses nor updates the current position.

For more information, see PolyLine in the *Platform SDK*.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::LineTo
CDC::Polygon
POINT Structure
CPoint Class

Other Resources

CDC Members

CDC::PolylineTo

Draws one or more straight lines.

```
BOOL PolylineTo(
   const POINT* lpPoints,
   int nCount
);
```

Parameters

lpPoints

Points to an array of POINT data structures that contains the vertices of the line.

nCount

Specifies the number of points in the array.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

A line is drawn from the current position to the first point specified by the *lpPoints* parameter by using the current pen. For each additional line, the function draws from the ending point of the previous line to the next point specified by *lpPoints*. **PolylineTo** moves the current position to the ending point of the last line. If the line segments drawn by this function form a closed figure, the figure is not filled.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::LineTo
CDC::Polyline
CDC::MoveTo
PolylineTo

Other Resources

CDC Members

CDC::PolyPolygon

Creates two or more polygons that are filled using the current polygon-filling mode.

```
BOOL PolyPolygon(
   LPPOINT lpPoints,
   LPINT lpPolyCounts,
   int nCount
);
```

Parameters

lpPoints

Points to an array of **POINT** structures or **CPoint** objects that define the vertices of the polygons.

lpPolyCounts

Points to an array of integers, each of which specifies the number of points in one of the polygons in the *lpPoints* array.

nCount

The number of entries in the *lpPolyCounts* array. This number specifies the number of polygons to be drawn. This value must be at least 2.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The polygons may be disjoint or overlapping.

Each polygon specified in a call to the **PolyPolygon** function must be closed. Unlike polygons created by the **Polygon** member function, the polygons created by **PolyPolygon** are not closed automatically.

The function creates two or more polygons. To create a single polygon, an application should use the **Polygon** member function.

The current polygon-filling mode can be retrieved or set by using the **GetPolyFillMode** and **SetPolyFillMode** member functions.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetPolyFillMode
CDC::Polygon
CDC::Polyline
CDC::SetPolyFillMode
PolyPolygon
POINT Structure
CPoint Class

Other Resources

CDC Members

CDC::PolyPolyline

Draws multiple series of connected line segments.

```
BOOL PolyPolyline(
   const POINT* lpPoints,
   const DWORD* lpPolyPoints,
   int nCount
);
```

Parameters

lpPoints

Points to an array of structures that contains the vertices of the polylines. The polylines are specified consecutively.

lpPolyPoints

Points to an array of variables specifying the number of points in the *lpPoints* array for the corresponding polygon. Each entry must be greater than or equal to 2.

nCount

Specifies the total number of counts in the *lpPolyPoints* array.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The line segments are drawn by using the current pen. The figures formed by the segments are not filled. The current position is neither used nor updated by this function.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::Polyline
CDC::PolylineTo
PolyPolyline

Other Resources

CDC Members

CDC::PtVisible

Determines whether the given point is within the clipping region of the device context.

```
virtual BOOL PtVisible(
   int x,
   int y
) const;
BOOL PtVisible(
   POINT point
) const;
```

Parameters

x

Specifies the logical x-coordinate of the point.

y

Specifies the logical y-coordinate of the point.

point

Specifies the point to check in logical coordinates. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Return Value

Nonzero if the specified point is within the clipping region; otherwise 0.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::RectVisible
CDC::SelectClipRgn
CPoint Class
PtVisible
POINT Structure

Other Resources

CDC Members

CDC::QueryAbort

Calls the abort function installed by the SetAbortProc member function for a printing application and queries whether the printing should be terminated.

```
BOOL QueryAbort( ) const;
```

Return Value

The return value is nonzero if printing should continue or if there is no abort procedure. It is 0 if the print job should be terminated. The return value is supplied by the abort function.

See Also

Reference

CDC Class

Hierarchy Chart

CDC::SetAbortProc

Other Resources

CDC Members

CDC::RealizePalette

Maps entries from the current logical palette to the system palette.

```
UINT RealizePalette( );
```

Return Value

Indicates how many entries in the logical palette were mapped to different entries in the system palette. This represents the number of entries that this function remapped to accommodate changes in the system palette since the logical palette was last realized.

Remarks

A logical color palette acts as a buffer between color-intensive applications and the system, allowing an application to use as many colors as needed without interfering with its own displayed colors or with colors displayed by other windows.

When a window has the input focus and calls **RealizePalette**, Windows ensures that the window will display all the requested colors, up to the maximum number simultaneously available on the screen. Windows also displays colors not found in the window's palette by matching them to available colors.

In addition, Windows matches the colors requested by inactive windows that call the function as closely as possible to the available colors. This significantly reduces undesirable changes in the colors displayed in inactive windows.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SelectPalette
CPalette Class
RealizePalette

Other Resources

CDC Members

CDC::Rectangle

Draws a rectangle using the current pen.

```
BOOL Rectangle(
   int x1,
   int y1,
   int x2,
   int y2
);
BOOL Rectangle(
   LPCRECT lpRect
);
```

Parameters

x1

Specifies the x-coordinate of the upper-left corner of the rectangle (in logical units).

y1

Specifies the y-coordinate of the upper-left corner of the rectangle (in logical units).

x2

Specifies the x-coordinate of the lower-right corner of the rectangle (in logical units).

y2

Specifies the y-coordinate of the lower-right corner of the rectangle (in logical units).

lpRect

Specifies the rectangle in logical units. You can pass either a **CRect** object or a pointer to a **RECT** structure for this parameter.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The interior of the rectangle is filled using the current brush.

The rectangle extends up to, but does not include, the right and bottom coordinates. This means that the height of the rectangle is *y2* – *y1* and the width of the rectangle is *x2* – *x1*. Both the width and the height of a rectangle must be greater than 2 units and less than 32,767 units.

Example

```
void CMyView::OnDraw(CDC* pDC)
{
   // create and select a solid blue brush
   CBrush brushBlue(RGB(0, 0, 255));
   CBrush* pOldBrush = pDC->SelectObject(&brushBlue);

   // create and select a thick, black pen
   CPen penBlack;
   penBlack.CreatePen(PS_SOLID, 3, RGB(0, 0, 0));
   CPen* pOldPen = pDC->SelectObject(&penBlack);

   // get our client rectangle
   CRect rect;
   GetClientRect(rect);

   // shrink our rect 20 pixels in each direction
   rect.DeflateRect(20, 20);

   // draw a thick black rectangle filled with blue
```

```
    pDC->Rectangle(rect);

    // put back the old objects
    pDC->SelectObject(pOldBrush);
    pDC->SelectObject(pOldPen);
}
```

See Also

Reference

CDC Class
Hierarchy Chart
Rectangle
CDC::Polyline
CDC::RoundRect
RECT Structure
CRect Class

Other Resources

CDC Members

CDC::RectVisible

Determines whether any part of the given rectangle lies within the clipping region of the display context.

```
virtual BOOL RectVisible(
   LPCRECT lpRect
) const;
```

Parameters

lpRect

Points to a **RECT** structure or a **CRect** object that contains the logical coordinates of the specified rectangle.

Return Value

Nonzero if any portion of the given rectangle lies within the clipping region; otherwise 0.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::PtVisible
CDC::SelectClipRgn
CRect Class
RectVisible
RECT Structure

Other Resources

CDC Members

CDC::ReleaseAttribDC

Call this member function to set **m_hAttribDC** to **NULL**.

```
virtual void ReleaseAttribDC( );
```

Remarks

This does not cause a **Detach** to occur. Only the output device context is attached to the **CDC** object, and only it can be detached.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetOutputDC
CDC::SetAttribDC
CDC::ReleaseOutputDC
CDC::m_hAttribDC

Other Resources

CDC Members

CDC::ReleaseOutputDC

Call this member function to set the **m_hDC** member to **NULL**.

```
virtual void ReleaseOutputDC( );
```

Remarks

This member function cannot be called when the output device context is attached to the **CDC** object. Use the **Detach** member function to detach the output device context.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetAttribDC
CDC::SetOutputDC
CDC::ReleaseAttribDC
CDC::m_hDC

Other Resources

CDC Members

CDC::ResetDC

Call this member function to update the device context wrapped by the **CDC** object.

```
BOOL ResetDC(
   const DEVMODE* lpDevMode
);
```

Parameters

lpDevMode

A pointer to a Windows **DEVMODE** structure.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The device context is updated from the information specified in the Windows **DEVMODE** structure. This member function only resets the attribute device context.

An application will typically use the **ResetDC** member function when a window processes a **WM_DEVMODECHANGE** message. You can also use this member function to change the paper orientation or paper bins while printing a document.

You cannot use this member function to change the driver name, device name, or output port. When the user changes the port connection or device name, you must delete the original device context and create a new device context with the new information.

Before you call this member function, you must ensure that all objects (other than stock objects) that had been selected into the device context have been selected out.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::m_hAttribDC
ResetDC
WM_DEVMODECHANGE
DEVMODE

Other Resources

CDC Members

CDC::RestoreDC

Restores the device context to the previous state identified by *nSavedDC*.

```
virtual BOOL RestoreDC(
   int nSavedDC
);
```

Parameters

nSavedDC

Specifies the device context to be restored. It can be a value returned by a previous **SaveDC** function call. If *nSavedDC* is –1, the most recently saved device context is restored.

Return Value

Nonzero if the specified context was restored; otherwise 0.

Remarks

RestoreDC restores the device context by popping state information off a stack created by earlier calls to the **SaveDC** member function.

The stack can contain the state information for several device contexts. If the context specified by *nSavedDC* is not at the top of the stack, **RestoreDC** deletes all state information between the device context specified by *nSavedDC* and the top of the stack. The deleted information is lost.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SaveDC
RestoreDC

Other Resources

CDC Members

CDC::RoundRect

Draws a rectangle with rounded corners using the current pen.

```
BOOL RoundRect(
   int x1,
   int y1,
   int x2,
   int y2,
   int x3,
   int y3
);
BOOL RoundRect(
   LPCRECT lpRect,
      POINT point
);
```

Parameters

x1

Specifies the x-coordinate of the upper-left corner of the rectangle (in logical units).

y1

Specifies the y-coordinate of the upper-left corner of the rectangle (in logical units).

x2

Specifies the x-coordinate of the lower-right corner of the rectangle (in logical units).

y2

Specifies the y-coordinate of the lower-right corner of the rectangle (in logical units).

x3

Specifies the width of the ellipse used to draw the rounded corners (in logical units).

y3

Specifies the height of the ellipse used to draw the rounded corners (in logical units).

lpRect

Specifies the bounding rectangle in logical units. You can pass either a **CRect** object or a pointer to a **RECT** structure for this parameter.

point

The x-coordinate of *point* specifies the width of the ellipse to draw the rounded corners (in logical units). The y-coordinate of *point* specifies the height of the ellipse to draw the rounded corners (in logical units). You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The interior of the rectangle is filled using the current brush.

The figure this function draws extends up to but does not include the right and bottom coordinates. This means that the height of the figure is *y2* – *y1* and the width of the figure is *x2* – *x1*. Both the height and the width of the bounding rectangle must be greater than 2 units and less than 32,767 units.

Example

```
void CMyView::OnDraw(CDC* pDC)
{
```

```
    // create and select a solid blue brush
    CBrush brushBlue(RGB(0, 0, 255));
    CBrush* pOldBrush = pDC->SelectObject(&brushBlue);

    // create and select a thick, black pen
    CPen penBlack;
    penBlack.CreatePen(PS_SOLID, 3, RGB(0, 0, 0));
    CPen* pOldPen = pDC->SelectObject(&penBlack);

    // get our client rectangle
    CRect rect;
    GetClientRect(rect);

    // shrink our rect 20 pixels in each direction
    rect.DeflateRect(20, 20);

    // Draw a thick black rectangle filled with blue
    // corners rounded at a 17-unit radius. Note that
    // a radius of three or less is not noticable because
    // the pen is three units wide.
    pDC->RoundRect(rect, CPoint(17, 17));

    // put back the old objects
    pDC->SelectObject(pOldBrush);
    pDC->SelectObject(pOldPen);
}
```

See Also

Reference

CDC Class
Hierarchy Chart
CDC::Rectangle
RoundRect
CRect Class
RECT Structure
POINT Structure
CPoint Class

Other Resources

CDC Members

CDC::SaveDC

Saves the current state of the device context by copying state information (such as clipping region, selected objects, and mapping mode) to a context stack maintained by Windows.

```
virtual int SaveDC( );
```

Return Value

An integer identifying the saved device context. It is 0 if an error occurs. This return value can be used to restore the device context by calling **RestoreDC**.

Remarks

The saved device context can later be restored by using **RestoreDC**.

SaveDC can be used any number of times to save any number of device-context states.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::RestoreDC
SaveDC

Other Resources

CDC Members

CDC::ScaleViewportExt

Modifies the viewport extents relative to the current values.

```
virtual CSize ScaleViewportExt(
   int xNum,
   int xDenom,
   int yNum,
   int yDenom
);
```

Parameters

xNum

Specifies the amount by which to multiply the current x-extent.

xDenom

Specifies the amount by which to divide the result of multiplying the current x-extent by the value of the *xNum* parameter.

yNum

Specifies the amount by which to multiply the current y-extent.

yDenom

Specifies the amount by which to divide the result of multiplying the current y-extent by the value of the *yNum* parameter.

Return Value

The previous viewport extents (in device units) as a **CSize** object.

Remarks

The formulas are written as follows:

```
xNewVE = ( xOldVE * xNum ) / xDenom
yNewVE = ( yOldVE * yNum ) / yDenom
```

The new viewport extents are calculated by multiplying the current extents by the given numerator and then dividing by the given denominator.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetViewportExt
CSize Class

Other Resources

CDC Members

CDC::ScaleWindowExt

Modifies the window extents relative to the current values.

```
virtual CSize ScaleWindowExt(
   int xNum,
   int xDenom,
   int yNum,
   int
yDenom );
```

Parameters

xNum

Specifies the amount by which to multiply the current x-extent.

xDenom

Specifies the amount by which to divide the result of multiplying the current x-extent by the value of the *xNum* parameter.

yNum

Specifies the amount by which to multiply the current y-extent.

yDenom

Specifies the amount by which to divide the result of multiplying the current y-extent by the value of the *yNum* parameter.

Return Value

The previous window extents (in logical units) as a **CSize** object.

Remarks

The formulas are written as follows:

```
xNewWE = ( xOldWE * xNum ) / xDenom
yNewWE = ( yOldWE * yNum ) / yDenom
```

The new window extents are calculated by multiplying the current extents by the given numerator and then dividing by the given denominator.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetWindowExt
CSize Class

Other Resources

CDC Members

CDC::ScrollDC

Scrolls a rectangle of bits horizontally and vertically.

```
BOOL ScrollDC(
   int dx,
   int dy,
   LPCRECT lpRectScroll,
   LPCRECT lpRectClip,
   CRgn* pRgnUpdate,
   LPRECT lpRectUpdate
);
```

Parameters

dx

Specifies the number of horizontal scroll units.

dy

Specifies the number of vertical scroll units.

lpRectScroll

Points to the **RECT** structure or **CRect** object that contains the coordinates of the scrolling rectangle.

lpRectClip

Points to the **RECT** structure or **CRect** object that contains the coordinates of the clipping rectangle. When this rectangle is smaller than the original one pointed to by *lpRectScroll*, scrolling occurs only in the smaller rectangle.

pRgnUpdate

Identifies the region uncovered by the scrolling process. The **ScrollDC** function defines this region; it is not necessarily a rectangle.

lpRectUpdate

Points to the **RECT** structure or **CRect** object that receives the coordinates of the rectangle that bounds the scrolling update region. This is the largest rectangular area that requires repainting. The values in the structure or object when the function returns are in client coordinates, regardless of the mapping mode for the given device context.

Return Value

Nonzero if scrolling is executed; otherwise 0.

Remarks

If *lpRectUpdate* is **NULL**, Windows does not compute the update rectangle. If both *pRgnUpdate* and *lpRectUpdate* are **NULL**, Windows does not compute the update region. If *pRgnUpdate* is not **NULL**, Windows assumes that it contains a valid pointer to the region uncovered by the scrolling process (defined by the **ScrollDC** member function). The update region returned in *lpRectUpdate* can be passed to **CWnd::InvalidateRgn** if required.

An application should use the **ScrollWindow** member function of class **CWnd** when it is necessary to scroll the entire client area of a window. Otherwise, it should use **ScrollDC**.

See Also

Reference

CDC Class
Hierarchy Chart
CWnd::InvalidateRgn
CWnd::ScrollWindow
ScrollDC
CRgn Class
RECT Structure
CRect Class

Other Resources

CDC Members

CDC::SelectClipPath

Selects the current path as a clipping region for the device context, combining the new region with any existing clipping region by using the specified mode.

```
BOOL SelectClipPath(
   int nMode
);
```

Parameters

nMode

Specifies the way to use the path. The following values are allowed:

- **RGN_AND** The new clipping region includes the intersection (overlapping areas) of the current clipping region and the current path.
- **RGN_COPY** The new clipping region is the current path.
- **RGN_DIFF** The new clipping region includes the areas of the current clipping region, and those of the current path are excluded.
- **RGN_OR** The new clipping region includes the union (combined areas) of the current clipping region and the current path.
- **RGN_XOR** The new clipping region includes the union of the current clipping region and the current path, but without the overlapping areas.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The device context identified must contain a closed path.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::BeginPath
CDC::EndPath

Other Resources

CDC Members

CDC::SelectClipRgn

Selects the given region as the current clipping region for the device context.

```
int SelectClipRgn(
   CRgn* pRgn
);
int SelectClipRgn(
   CRgn* pRgn,
   int nMode
);
```

Parameters

pRgn

Identifies the region to be selected.

- For the first version of this function, if this value is **NULL**, the entire client area is selected and output is still clipped to the window.
- For the second version of this function, this handle can be **NULL** only when the **RGN_COPY** mode is specified.

nMode

Specifies the operation to be performed. It must be one of the following values:

- **RGN_AND** The new clipping region combines the overlapping areas of the current clipping region and the region identified by *pRgn*.
- **RGN_COPY** The new clipping region is a copy of the region identified by *pRgn*. This is functionality is identical to the first version of **SelectClipRgn**. If the region identified by *pRgn* is **NULL**, the new clipping region becomes the default clipping region (a null region).
- **RGN_DIFF** The new clipping region combines the areas of the current clipping region with those areas excluded from the region identified by *pRgn*.
- **RGN_OR** The new clipping region combines the current clipping region and the region identified by *pRgn*.
- **RGN_XOR** The new clipping region combines the current clipping region and the region identified by *pRgn* but excludes any overlapping areas.

Return Value

The region's type. It can be any of the following values:

- **COMPLEXREGION** New clipping region has overlapping borders.
- **ERROR** Device context or region is not valid.
- **NULLREGION** New clipping region is empty.
- **SIMPLEREGION** New clipping region has no overlapping borders.

Remarks

Only a copy of the selected region is used. The region itself can be selected for any number of other device contexts, or it can be deleted.

The function assumes that the coordinates for the given region are specified in device units. Some printer devices support text output at a higher resolution than graphics output in order to retain the precision needed to express text metrics. These devices report device units at the higher resolution, that is, in text units. These devices then scale coordinates for graphics so that several reported device units map to only 1 graphic unit. You should always call the **SelectClipRgn** function using text units.

Applications that must take the scaling of graphics objects in the GDI can use the **GETSCALINGFACTOR** printer escape to determine the scaling factor. This scaling factor affects clipping. If a region is used to clip graphics, GDI divides the coordinates by the scaling factor. If the region is used to clip text, GDI makes no scaling adjustment. A scaling factor of 1 causes the

coordinates to be divided by 2; a scaling factor of 2 causes the coordinates to be divided by 4; and so on.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetClipBox
CDC::Escape
CRgn Class
SelectClipRgn

Other Resources

CDC Members

CDC::SelectObject

Selects an object into the device context.

```
CPen* SelectObject(
   CPen* pPen
);
CBrush* SelectObject(
   CBrush* pBrush
);
virtual CFont* SelectObject(
   CFont* pFont
);
CBitmap* SelectObject(
   CBitmap* pBitmap
);
int SelectObject(
   CRgn* pRgn
);
CGdiObject* SelectObject(
   CGdiObject* pObject
);
```

Parameters

pPen

A pointer to a CPen object to be selected.

pBrush

A pointer to a CBrush object to be selected.

pFont

A pointer to a CFont object to be selected.

pBitmap

A pointer to a CBitmap object to be selected.

pRgn

A pointer to a CRgn object to be selected.

pObject

A pointer to a CGdiObject object to be selected.

Return Value

A pointer to the object being replaced. This is a pointer to an object of one of the classes derived from **CGdiObject**, such as **CPen**, depending on which version of the function is used. The return value is **NULL** if there is an error. This function may return a pointer to a temporary object. This temporary object is only valid during the processing of one Windows message. For more information, see **CGdiObject::FromHandle**.

The version of the member function that takes a region parameter performs the same task as the **SelectClipRgn** member function. Its return value can be any of the following:

- **COMPLEXREGION** New clipping region has overlapping borders.
- **ERROR** Device context or region is not valid.
- **NULLREGION** New clipping region is empty.
- **SIMPLEREGION** New clipping region has no overlapping borders.

Remarks

Class **CDC** provides five versions specialized for particular kinds of GDI objects, including pens, brushes, fonts, bitmaps, and regions. The newly selected object replaces the previous object of the same type. For example, if *pObject* of the general version of **SelectObject** points to a CPen object, the function replaces the current pen with the pen specified by *pObject*.

An application can select a bitmap into memory device contexts only and into only one memory device context at a time. The format of the bitmap must either be monochrome or compatible with the device context; if it is not, **SelectObject** returns an error.

For Windows 3.1 and later, the **SelectObject** function returns the same value whether it is used in a metafile or not. Under previous versions of Windows, **SelectObject** returned a nonzero value for success and 0 for failure when it was used in a metafile.

See Also

Reference

CDC Class
Hierarchy Chart
CGdiObject::DeleteObject
CGdiObject::FromHandle
CDC::SelectClipRgn
CDC::SelectPalette
SelectObject

Other Resources

CDC Members

CDC::SelectPalette

Selects the logical palette that is specified by *pPalette* as the selected palette object of the device context.

```
CPalette* SelectPalette(
   CPalette* pPalette,
   BOOL bForceBackground
);
```

Parameters

pPalette

Identifies the logical palette to be selected. This palette must already have been created with the **CPalette** member function CreatePalette.

bForceBackground

Specifies whether the logical palette is forced to be a background palette. If *bForceBackground* is nonzero, the selected palette is always a background palette, regardless of whether the window has the input focus. If *bForceBackground* is 0 and the device context is attached to a window, the logical palette is a foreground palette when the window has the input focus.

Return Value

A pointer to a **CPalette** object identifying the logical palette replaced by the palette specified by *pPalette*. It is **NULL** if there is an error.

Remarks

The new palette becomes the palette object used by GDI to control colors displayed in the device context and replaces the previous palette.

An application can select a logical palette into more than one device context. However, changes to a logical palette will affect all device contexts for which it is selected. If an application selects a palette into more than one device context, the device contexts must all belong to the same physical device.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::RealizePalette
CPalette Class
SelectPalette

Other Resources

CDC Members

CDC::SelectStockObject

Selects a CGdiObject object that corresponds to one of the predefined stock pens, brushes, or fonts.

```
virtual CGdiObject* SelectStockObject(
   int nIndex
);
```

Parameters

nIndex

Specifies the kind of stock object desired. It can be one of the following values:

- **BLACK_BRUSH** Black brush.
- **DKGRAY_BRUSH** Dark gray brush.
- **GRAY_BRUSH** Gray brush.
- **HOLLOW_BRUSH** Hollow brush.
- **LTGRAY_BRUSH** Light gray brush.
- **NULL_BRUSH** Null brush.
- **WHITE_BRUSH** White brush.
- **BLACK_PEN** Black pen.
- **NULL_PEN** Null pen.
- **WHITE_PEN** White pen.
- **ANSI_FIXED_FONT** ANSI fixed system font.
- **ANSI_VAR_FONT** ANSI variable system font.
- **DEVICE_DEFAULT_FONT** Device-dependent font.
- **OEM_FIXED_FONT** OEM-dependent fixed font.
- **SYSTEM_FONT** The system font. By default, Windows uses the system font to draw menus, dialog-box controls, and other text. In Windows versions 3.0 and later, the system font is proportional width; earlier versions of Windows use a fixed-width system font.
- **SYSTEM_FIXED_FONT** The fixed-width system font used in Windows prior to version 3.0. This object is available for compatibility with earlier versions of Windows.
- **DEFAULT_PALETTE** Default color palette. This palette consists of the 20 static colors in the system palette.

Return Value

A pointer to the **CGdiObject** object that was replaced if the function is successful. The actual object pointed to is a CPen, CBrush, or CFont object. If the call is unsuccessful, the return value is **NULL**.

See Also

Reference

CDC Class
Hierarchy Chart
CGdiObject::GetObject

Other Resources

CDC Members

CDC::SetAbortProc

Installs the abort procedure for the print job.

```
int SetAbortProc(
   BOOL ( CALLBACK* lpfn )( HDC, int )
);
```

Parameters

lpfn

A pointer to the abort function to install as the abort procedure. For more about the callback function, see Callback Function for CDC::SetAbortProc.

Return Value

Specifies the outcome of the **SetAbortProc** function. Some of the following values are more probable than others, but all are possible.

- **SP_ERROR** General error.
- **SP_OUTOFDISK** Not enough disk space is currently available for spooling, and no more space will become available.
- **SP_OUTOFMEMORY** Not enough memory is available for spooling.
- **SP_USERABORT** User ended the job through the Print Manager.

Remarks

If an application is to allow the print job to be canceled during spooling, it must set the abort function before the print job is started with the StartDoc member function. The Print Manager calls the abort function during spooling to allow the application to cancel the print job or to process out-of-disk-space conditions. If no abort function is set, the print job will fail if there is not enough disk space for spooling.

Note that the features of Microsoft Visual C++ simplify the creation of the callback function passed to **SetAbortProc**. The address passed to the **EnumObjects** member function is a pointer to a function exported with **__declspec(dllexport)** and with the **__stdcall** calling convention.

You also do not have to export the function name in an **EXPORTS** statement in your application's module-definition file. You can instead use the **EXPORT** function modifier, as in

BOOL CALLBACK EXPORT AFunction(**HDC**, **int);**

to cause the compiler to emit the proper export record for export by name without aliasing. This works for most needs. For some special cases, such as exporting a function by ordinal or aliasing the export, you still need to use an **EXPORTS** statement in a module-definition file.

Callback registration interfaces are now type-safe (you must pass in a function pointer that points to the right kind of function for the specific callback).

Also note that all callback functions must trap Microsoft Foundation exceptions before returning to Windows, since exceptions cannot be thrown across callback boundaries. For more information about exceptions, see the article Exceptions.

See Also

Reference

CDC Class

Hierarchy Chart

Other Resources

CDC Members

CDC::SetArcDirection

Sets the drawing direction to be used for arc and rectangle functions.

```
int SetArcDirection(
   int nArcDirection
);
```

Parameters

nArcDirection

Specifies the new arc direction. This parameter can be either of the following values:

- **AD_COUNTERCLOCKWISE** Figures drawn counterclockwise.
- **AD_CLOCKWISE** Figures drawn clockwise.

Return Value

Specifies the old arc direction, if successful; otherwise 0.

Remarks

The default direction is counterclockwise. The **SetArcDirection** function specifies the direction in which the following functions draw:

Arc	**Pie**
ArcTo	**Rectangle**
Chord	**RoundRect**
Ellipse	

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetArcDirection
SetArcDirection

Other Resources

CDC Members

CDC::SetAttribDC

Call this function to set the attribute device context, **m_hAttribDC**.

```
virtual void SetAttribDC(
   HDC hDC
);
```

Parameters

hDC

A Windows device context.

Remarks

This member function does not attach the device context to the **CDC** object. Only the output device context is attached to a **CDC** object.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetOutputDC
CDC::ReleaseAttribDC
CDC::ReleaseOutputDC

Other Resources

CDC Members

CDC::SetBkColor

Sets the current background color to the specified color.

```
virtual COLORREF SetBkColor(
   COLORREF crColor
);
```

Parameters

crColor

Specifies the new background color.

Return Value

The previous background color as an RGB color value. If an error occurs, the return value is 0x80000000.

Remarks

If the background mode is **OPAQUE**, the system uses the background color to fill the gaps in styled lines, the gaps between hatched lines in brushes, and the background in character cells. The system also uses the background color when converting bitmaps between color and monochrome device contexts.

If the device cannot display the specified color, the system sets the background color to the nearest physical color.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::BitBlt
CDC::GetBkColor
CDC::GetBkMode
CDC::SetBkMode
CDC::StretchBlt
SetBkColor

Other Resources

CDC Members

CDC::SetBkMode

Sets the background mode.

```
int SetBkMode(
   int nBkMode
);
```

Parameters

nBkMode

Specifies the mode to be set. This parameter can be either of the following values:

- **OPAQUE** Background is filled with the current background color before the text, hatched brush, or pen is drawn. This is the default background mode.
- **TRANSPARENT** Background is not changed before drawing.

Return Value

The previous background mode.

Remarks

The background mode defines whether the system removes existing background colors on the drawing surface before drawing text, hatched brushes, or any pen style that is not a solid line.

Example

See the example for CWnd::OnCtlColor.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetBkColor
CDC::GetBkMode
CDC::SetBkColor
SetBkMode

Other Resources

CDC Members

CDC::SetBoundsRect

Controls the accumulation of bounding-rectangle information for the specified device context.

```
UINT SetBoundsRect(
   LPCRECT lpRectBounds,
   UINT flags
);
```

Parameters

lpRectBounds

Points to a **RECT** structure or **CRect** object that is used to set the bounding rectangle. Rectangle dimensions are given in logical coordinates. This parameter can be **NULL**.

flags

Specifies how the new rectangle will be combined with the accumulated rectangle. This parameter can be a combination of the following values:

- **DCB_ACCUMULATE** Add the rectangle specified by *lpRectBounds* to the bounding rectangle (using a rectangle-union operation).
- **DCB_DISABLE** Turn off bounds accumulation.
- **DCB_ENABLE** Turn on bounds accumulation. (The default setting for bounds accumulation is disabled.)

Return Value

The current state of the bounding rectangle, if the function is successful. Like *flags*, the return value can be a combination of **DCB_** values:

- **DCB_ACCUMULATE** The bounding rectangle is not empty. This value will always be set.
- **DCB_DISABLE** Bounds accumulation is off.
- **DCB_ENABLE** Bounds accumulation is on.

Remarks

Windows can maintain a bounding rectangle for all drawing operations. This rectangle can be queried and reset by the application. The drawing bounds are useful for invalidating bitmap caches.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetBoundsRect
SetBoundsRect
RECT Structure
CRect Class

Other Resources

CDC Members

CDC::SetBrushOrg

Specifies the origin that GDI will assign to the next brush that the application selects into the device context.

```
CPoint SetBrushOrg(
   int x,
   int y
);
CPoint SetBrushOrg(
   POINT point
);
```

Parameters

x

Specifies the x-coordinate (in device units) of the new origin. This value must be in the range 0–7.

y

Specifies the y-coordinate (in device units) of the new origin. This value must be in the range 0–7.

point

Specifies the x- and y-coordinates of the new origin. Each value must be in the range 0–7. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Return Value

The previous origin of the brush in device units.

Remarks

The default coordinates for the brush origin are (0, 0). To alter the origin of a brush, call the **UnrealizeObject** function for the **CBrush** object, call **SetBrushOrg**, and then call the **SelectObject** member function to select the brush into the device context.

Do not use **SetBrushOrg** with stock **CBrush** objects.

See Also

Reference

CDC Class
Hierarchy Chart
CBrush Class
CDC::GetBrushOrg
CDC::SelectObject
CGdiObject::UnrealizeObject
POINT Structure
CPoint Class

Other Resources

CDC Members

CDC::SetColorAdjustment

Sets the color adjustment values for the device context using the specified values.

```
BOOL SetColorAdjustment(
   const COLORADJUSTMENT* lpColorAdjust
);
```

Parameters

lpColorAdjust

Points to a COLORADJUSTMENT data structure containing the color adjustment values.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The color adjustment values are used to adjust the input color of the source bitmap for calls to the **CDC::StretchBlt** member function when **HALFTONE** mode is set.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetStretchBltMode
CDC::StretchBlt
StretchDIBits

Other Resources

CDC Members

CDC::SetDCBrushColor

Sets the current device context (DC) brush color to the specified color value.

```
COLORREF SetDCBrushColor(
   COLORREF crColor
);
```

Parameters

crColor

Specifies the new brush color.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

This member function emulates the functionality of the function SetDCBrushColor, as described in the *Platform SDK*.

See Also

Reference

CDC Class

Hierarchy Chart

CDC::GetDCBrushColor

Other Resources

CDC Members

CDC::SetDCPenColor

Sets the current device context (DC) pen color to the specified color value.

```
COLORREF SetDCPenColor(
   COLORREF crColor
);
```

Parameters

crColor

Specifies the new pen color.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

This member function utilizes the Win32 function SetDCPenColor, as described in the *Platform SDK*.

See Also

Reference

CDC Class

Hierarchy Chart

CDC::GetDCPenColor

Other Resources

CDC Members

CDC::SetGraphicsMode

Sets the graphics mode for the specified device context.

```
int SetGraphicsMode(
   int iMode
);
```

Parameters

iMode

Specifies the graphics mode. For a list of the values that this parameter can take, see SetGraphicsMode.

Return Value

Returns the old graphics mode on success.

Returns 0 on failure. To get extended error information, call GetLastError.

Remarks

This method wraps the Windows GDI function SetGraphicsMode.

See Also

Reference

CDC Class

Hierarchy Chart

Other Resources

CDC Members

CDC::SetLayout

Call this member function to change the layout of the text and graphics for a device context to right to left, the standard layout for cultures such as Arabic and Hebrew.

```
DWORD SetLayout(
   DWORD dwLayout
);
```

Parameters

dwLayout

Device context layout and bitmap control flags. It can be a combination of the following values.

Value	Meaning
LAYOUT_BITMAPORIENTATIONPRESERVED	Disables any reflection for calls to CDC::BitBlt and CDC::StretchBlt.
LAYOUT_RTL	Sets the default horizontal layout to be right to left.
LAYOUT_LTR	Sets the default layout to be left to right.

Return Value

If successful, the previous layout of the device context.

If unsuccessful, **GDI_ERROR**. To get extended error information, call GetLastError.

Remarks

Normally, you would not call **SetLayout** for a window. Rather, you control the right-to-left layout in a window by setting the extended window styles such as **WS_EX_RTLREADING**. A device context, such as a printer or a metafile, does not inherit this layout. The only way to set the device context for a right-to-left layout is by calling **SetLayout**.

If you call **SetLayout(LAYOUT_RTL**), **SetLayout** automatically changes the mapping mode to **MM_ISOTROPIC**. As a result, a subsequent call to GetMapMode will return **MM_ISOTROPIC** instead of **MM_TEXT**.

In some cases, such as with many bitmaps, you may want to preserve the left-to-right layout. In these cases, render the image by calling **BitBlt** or **StretchBlt**, then set the bitmap control flag for *dwLayout* to **LAYOUT_BITMAPORIENTATIONPRESERVED**.

Once you change the layout with the **LAYOUT_RTL** flag, the flags normally specifying right or left are reversed. To avoid confusion, you may want to define alternate names for the standard flags. For a list of suggested alternate flag names, see SetLayout in the *Platform SDK*.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetLayout
CDC::SetMapMode

Other Resources

CDC Members

CDC::SetMapMode

Sets the mapping mode.

```
virtual int SetMapMode(
   int nMapMode
);
```

Parameters

nMapMode

Specifies the new mapping mode. It can be any one of the following values:

- **MM_ANISOTROPIC** Logical units are converted to arbitrary units with arbitrarily scaled axes. Setting the mapping mode to **MM_ANISOTROPIC** does not change the current window or viewport settings. To change the units, orientation, and scaling, call the SetWindowExt and SetViewportExt member functions.
- **MM_HIENGLISH** Each logical unit is converted to 0.001 inch. Positive x is to the right; positive y is up.
- **MM_HIMETRIC** Each logical unit is converted to 0.01 millimeter. Positive x is to the right; positive y is up.
- **MM_ISOTROPIC** Logical units are converted to arbitrary units with equally scaled axes; that is, 1 unit along the x-axis is equal to 1 unit along the y-axis. Use the **SetWindowExt** and **SetViewportExt** member functions to specify the desired units and the orientation of the axes. GDI makes adjustments as necessary to ensure that the x and y units remain the same size.
- **MM_LOENGLISH** Each logical unit is converted to 0.01 inch. Positive x is to the right; positive y is up.
- **MM_LOMETRIC** Each logical unit is converted to 0.1 millimeter. Positive x is to the right; positive y is up.
- **MM_TEXT** Each logical unit is converted to 1 device pixel. Positive x is to the right; positive y is down.
- **MM_TWIPS** Each logical unit is converted to 1/20 of a point. (Because a point is 1/72 inch, a twip is 1/1440 inch.) Positive x is to the right; positive y is up.

Return Value

The previous mapping mode.

Remarks

The mapping mode defines the unit of measure used to convert logical units to device units; it also defines the orientation of the device's x- and y-axes. GDI uses the mapping mode to convert logical coordinates into the appropriate device coordinates. The **MM_TEXT** mode allows applications to work in device pixels, where 1 unit is equal to 1 pixel. The physical size of a pixel varies from device to device.

The **MM_HIENGLISH**, **MM_HIMETRIC**, **MM_LOENGLISH**, **MM_LOMETRIC**, and **MM_TWIPS** modes are useful for applications that must draw in physically meaningful units (such as inches or millimeters). The **MM_ISOTROPIC** mode ensures a 1:1 aspect ratio, which is useful when it is important to preserve the exact shape of an image. The **MM_ANISOTROPIC** mode allows the x- and y-coordinates to be adjusted independently.

Note
If you call SetLayout to change the DC (device context) to right-to-left layout, **SetLayout** automatically changes the mapping mode to **MM_ISOTROPIC**.

Example

See the example for CView::OnPrepareDC.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetViewportExt

CDC::SetWindowExt
SetMapMode
Other Resources
CDC Members

CDC::SetMapperFlags

Changes the method used by the font mapper when it converts a logical font to a physical font.

```
DWORD SetMapperFlags(
   DWORD dwFlag
);
```

Parameters

dwFlag

Specifies whether the font mapper attempts to match a font's aspect height and width to the device. When this value is **ASPECT_FILTERING**, the mapper selects only fonts whose x-aspect and y-aspect exactly match those of the specified device.

Return Value

The previous value of the font-mapper flag.

Remarks

An application can use **SetMapperFlags** to cause the font mapper to attempt to choose only a physical font that exactly matches the aspect ratio of the specified device.

An application that uses only raster fonts can use the **SetMapperFlags** function to ensure that the font selected by the font mapper is attractive and readable on the specified device. Applications that use scalable (TrueType) fonts typically do not use **SetMapperFlags**.

If no physical font has an aspect ratio that matches the specification in the logical font, GDI chooses a new aspect ratio and selects a font that matches this new aspect ratio.

See Also

Reference

CDC Class
Hierarchy Chart
SetMapperFlags

Other Resources

CDC Members

CDC::SetMiterLimit

Sets the limit for the length of miter joins for the device context.

```
BOOL SetMiterLimit(
   float fMiterLimit
);
```

Parameters

fMiterLimit

Specifies the new miter limit for the device context.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The miter length is defined as the distance from the intersection of the line walls on the inside of the join to the intersection of the line walls on the outside of the join. The miter limit is the maximum allowed ratio of the miter length to the line width. The default miter limit is 10.0.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetMiterLimit
SetMiterLimit

Other Resources

CDC Members

CDC::SetOutputDC

Call this member function to set the output device context, **m_hDC**.

```
virtual void SetOutputDC(
   HDC hDC
);
```

Parameters

hDC

A Windows device context.

Remarks

This member function can only be called when a device context has not been attached to the **CDC** object. This member function sets **m_hDC** but does not attach the device context to the **CDC** object.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetAttribDC
CDC::ReleaseAttribDC
CDC::ReleaseOutputDC
CDC::m_hDC

Other Resources

CDC Members

CDC::SetPixel

Sets the pixel at the point specified to the closest approximation of the color specified by *crColor*.

```
COLORREF SetPixel(
   int x,
   int y,
   COLORREF crColor
);
COLORREF SetPixel(
   POINT point,
   COLORREF crColor
);
```

Parameters

x

Specifies the logical x-coordinate of the point to be set.

y

Specifies the logical y-coordinate of the point to be set.

crColor

A **COLORREF** RGB value that specifies the color used to paint the point. See COLORREF in the *Platform SDK* for a description of this value.

point

Specifies the logical x- and y-coordinates of the point to be set. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Return Value

An RGB value for the color that the point is actually painted. This value can be different from that specified by *crColor* if an approximation of that color is used. If the function fails (if the point is outside the clipping region), the return value is –1.

Remarks

The point must be in the clipping region. If the point is not in the clipping region, the function does nothing.

Not all devices support the **SetPixel** function. To determine whether a device supports **SetPixel**, call the **GetDeviceCaps** member function with the **RASTERCAPS** index and check the return value for the **RC_BITBLT** flag.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetDeviceCaps
CDC::GetPixel
SetPixel
POINT Structure
CPoint Class

Other Resources

CDC Members

CDC::SetPixelV

Sets the pixel at the specified coordinates to the closest approximation of the specified color.

```
BOOL SetPixelV(
      int x,
   int y,
   COLORREF crColor
);
BOOL SetPixelV(
   POINT point,
   COLORREF crColor
);
```

Parameters

x

Specifies the x-coordinate, in logical units, of the point to be set.

y

Specifies the y-coordinate, in logical units, of the point to be set.

crColor

Specifies the color to be used to paint the point.

point

Specifies the logical x- and y-coordinates of the point to be set. You can pass either a POINT data structure or a CPoint object for this parameter.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The point must be in both the clipping region and the visible part of the device surface. Not all devices support the member function. For more information, see the **RC_BITBLT** capability in the **CDC::GetDeviceCaps** member function. **SetPixelV** is faster than **SetPixel** because it does not need to return the color value of the point actually painted.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetDeviceCaps
CDC::SetPixel
SetPixelV

Other Resources

CDC Members

CDC::SetPolyFillMode

Sets the polygon-filling mode.

```
int SetPolyFillMode(
   int nPolyFillMode
);
```

Parameters

nPolyFillMode

Specifies the new filling mode. This value may be either **ALTERNATE** or **WINDING**. The default mode set in Windows is **ALTERNATE**.

Return Value

The previous filling mode, if successful; otherwise 0.

Remarks

When the polygon-filling mode is **ALTERNATE**, the system fills the area between odd-numbered and even-numbered polygon sides on each scan line. That is, the system fills the area between the first and second side, between the third and fourth side, and so on. This mode is the default.

When the polygon-filling mode is **WINDING**, the system uses the direction in which a figure was drawn to determine whether to fill an area. Each line segment in a polygon is drawn in either a clockwise or a counterclockwise direction. Whenever an imaginary line drawn from an enclosed area to the outside of a figure passes through a clockwise line segment, a count is incremented. When the line passes through a counterclockwise line segment, the count is decremented. The area is filled if the count is nonzero when the line reaches the outside of the figure.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetPolyFillMode
CDC::PolyPolygon
SetPolyFillMode

Other Resources

CDC Members

CDC::SetROP2

Sets the current drawing mode.

```
int SetROP2(
   int nDrawMode
);
```

Parameters

nDrawMode

Specifies the new drawing mode. It can be any of the following values:

- **R2_BLACK** Pixel is always black.
- **R2_WHITE** Pixel is always white.
- **R2_NOP** Pixel remains unchanged.
- **R2_NOT** Pixel is the inverse of the screen color.
- **R2_COPYPEN** Pixel is the pen color.
- **R2_NOTCOPYPEN** Pixel is the inverse of the pen color.
- **R2_MERGEPENNOT** Pixel is a combination of the pen color and the inverse of the screen color (final pixel = (NOT screen pixel) OR pen).
- **R2_MASKPENNOT** Pixel is a combination of the colors common to both the pen and the inverse of the screen (final pixel = (NOT screen pixel) AND pen).
- **R2_MERGENOTPEN** Pixel is a combination of the screen color and the inverse of the pen color (final pixel = (NOT pen) OR screen pixel).
- **R2_MASKNOTPEN** Pixel is a combination of the colors common to both the screen and the inverse of the pen (final pixel = (NOT pen) AND screen pixel).
- **R2_MERGEPEN** Pixel is a combination of the pen color and the screen color (final pixel = pen OR screen pixel).
- **R2_NOTMERGEPEN** Pixel is the inverse of the **R2_MERGEPEN** color (final pixel = NOT(pen OR screen pixel)).
- **R2_MASKPEN** Pixel is a combination of the colors common to both the pen and the screen (final pixel = pen AND screen pixel).
- **R2_NOTMASKPEN** Pixel is the inverse of the **R2_MASKPEN** color (final pixel = NOT(pen AND screen pixel)).
- **R2_XORPEN** Pixel is a combination of the colors that are in the pen or in the screen, but not in both (final pixel = pen XOR screen pixel).
- **R2_NOTXORPEN** Pixel is the inverse of the **R2_XORPEN** color (final pixel = NOT(pen XOR screen pixel)).

Return Value

The previous drawing mode.

It can be any of the values given in the *Platform SDK*.

Remarks

The drawing mode specifies how the colors of the pen and the interior of filled objects are combined with the color already on the display surface.

The drawing mode is for raster devices only; it does not apply to vector devices. Drawing modes are binary raster-operation codes representing all possible Boolean combinations of two variables, using the binary operators AND, OR, and XOR (exclusive OR), and the unary operation NOT.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetDeviceCaps
CDC::GetROP2
SetROP2

Other Resources

CDC Members

CDC::SetStretchBltMode

Sets the bitmap-stretching mode for the **StretchBlt** member function.

```
int SetStretchBltMode(
   int nStretchMode
);
```

Parameters

nStretchMode

Specifies the stretching mode. It can be any of the following values:

Value	Description
BLACKONWHITE	Performs a Boolean AND operation using the color values for the eliminated and existing pixels. If the bitmap is a monochrome bitmap, this mode preserves black pixels at the expense of white pixels.
COLORONCOLOR	Deletes the pixels. This mode deletes all eliminated lines of pixels without trying to preserve their information.
HALFTONE	Maps pixels from the source rectangle into blocks of pixels in the destination rectangle. The average color over the destination block of pixels approximates the color of the source pixels.
	After setting the **HALFTONE** stretching mode, an application must call the Win32 function SetBrushOrgEx to set the brush origin. If it fails to do so, brush misalignment occurs.
STRETCH_ANDSCANS	**Windows 95/98**: Same as **BLACKONWHITE**
STRETCH_DELETESCANS	**Windows 95/98**: Same as **COLORONCOLOR**
STRETCH_HALFTONE	**Windows 95/98**: Same as **HALFTONE**.
STRETCH_ORSCANS	**Windows 95/98**: Same as **WHITEONBLACK**
WHITEONBLACK	Performs a Boolean OR operation using the color values for the eliminated and existing pixels. If the bitmap is a monochrome bitmap, this mode preserves white pixels at the expense of black pixels.

Return Value

The previous stretching mode. It can be **STRETCH_ANDSCANS**, **STRETCH_DELETESCANS**, or **STRETCH_ORSCANS**.

Remarks

The bitmap-stretching mode defines how information is removed from bitmaps that are compressed by using the function.

The **BLACKONWHITE** (**STRETCH_ANDSCANS**) and **WHITEONBLACK** (**STRETCH_ORSCANS**) modes are typically used to preserve foreground pixels in monochrome bitmaps. The **COLORONCOLOR** (**STRETCH_DELETESCANS**) mode is typically used to preserve color in color bitmaps.

The **HALFTONE** mode requires more processing of the source image than the other three modes; it is slower than the others, but produces higher quality images. Also note that **SetBrushOrgEx** must be called after setting the **HALFTONE** mode to avoid brush misalignment.

Additional stretching modes might also be available depending on the capabilities of the device driver.

See Also

Reference

CDC Class

Hierarchy Chart

CDC::GetStretchBltMode

CDC::StretchBlt

SetStretchBltMode

Other Resources

CDC Members

CDC::SetTextAlign

Sets the text-alignment flags.

```
UINT SetTextAlign(
   UINT nFlags
);
```

Parameters

nFlags

Specifies text-alignment flags. The flags specify the relationship between a point and a rectangle that bounds the text. The point can be either the current position or coordinates specified by a text-output function. The rectangle that bounds the text is defined by the adjacent character cells in the text string. The *nFlags* parameter can be one or more flags from the following three categories. Choose only one flag from each category. The first category affects text alignment in the x-direction:

- **TA_CENTER** Aligns the point with the horizontal center of the bounding rectangle.
- **TA_LEFT** Aligns the point with the left side of the bounding rectangle. This is the default setting.
- **TA_RIGHT** Aligns the point with the right side of the bounding rectangle.

The second category affects text alignment in the y-direction:

- **TA_BASELINE** Aligns the point with the base line of the chosen font.
- **TA_BOTTOM** Aligns the point with the bottom of the bounding rectangle.
- **TA_TOP** Aligns the point with the top of the bounding rectangle. This is the default setting.

The third category determines whether the current position is updated when text is written:

- **TA_NOUPDATECP** Does not update the current position after each call to a text-output function. This is the default setting.
- **TA_UPDATECP** Updates the current x-position after each call to a text-output function. The new position is at the right side of the bounding rectangle for the text. When this flag is set, the coordinates specified in calls to the **TextOut** member function are ignored.

Return Value

The previous text-alignment setting, if successful. The low-order byte contains the horizontal setting and the high-order byte contains the vertical setting; otherwise 0.

Remarks

The **TextOut** and **ExtTextOut** member functions use these flags when positioning a string of text on a display or device. The flags specify the relationship between a specific point and a rectangle that bounds the text. The coordinates of this point are passed as parameters to the **TextOut** member function. The rectangle that bounds the text is formed by the adjacent character cells in the text string.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::ExtTextOut
CDC::GetTextAlign
CDC::TabbedTextOut
CDC::TextOut
SetTextAlign

Other Resources

CDC Members

CDC::SetTextCharacterExtra

Sets the amount of intercharacter spacing.

```
int SetTextCharacterExtra(
   int nCharExtra
);
```

Parameters

nCharExtra

Specifies the amount of extra space (in logical units) to be added to each character. If the current mapping mode is not **MM_TEXT**, *nCharExtra* is transformed and rounded to the nearest pixel.

Return Value

The amount of the previous intercharacter spacing.

Remarks

GDI adds this spacing to each character, including break characters, when it writes a line of text to the device context. The default value for the amount of intercharacter spacing is 0.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetTextCharacterExtra
SetTextCharacterExtra

Other Resources

CDC Members

CDC::SetTextColor

Sets the text color to the specified color.

```
virtual COLORREF SetTextColor(
   COLORREF crColor
);
```

Parameters

crColor

Specifies the color of the text as an RGB color value.

Return Value

An RGB value for the previous text color.

Remarks

The system will use this text color when writing text to this device context and also when converting bitmaps between color and monochrome device contexts.

If the device cannot represent the specified color, the system sets the text color to the nearest physical color. The background color for a character is specified by the **SetBkColor** and **SetBkMode** member functions.

Example

See the example for CWnd::OnCtlColor.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetTextColor
CDC::BitBlt
CDC::SetBkColor
CDC::SetBkMode
SetTextColor

Other Resources

CDC Members

CDC::SetTextJustification

Adds space to the break characters in a string.

```
int SetTextJustification(
   int nBreakExtra,
   int nBreakCount
);
```

Parameters

nBreakExtra

Specifies the total extra space to be added to the line of text (in logical units). If the current mapping mode is not **MM_TEXT**, the value given by this parameter is converted to the current mapping mode and rounded to the nearest device unit.

nBreakCount

Specifies the number of break characters in the line.

Return Value

One if the function is successful; otherwise 0.

Remarks

An application can use the **GetTextMetrics** member functions to retrieve a font's break character.

After the **SetTextJustification** member function is called, a call to a text-output function (such as **TextOut**) distributes the specified extra space evenly among the specified number of break characters. The break character is usually the space character (ASCII 32), but may be defined by a font as some other character.

The member function **GetTextExtent** is typically used with **SetTextJustification**. **GetTextExtent** computes the width of a given line before alignment. An application can determine how much space to specify in the *nBreakExtra* parameter by subtracting the value returned by **GetTextExtent** from the width of the string after alignment.

The **SetTextJustification** function can be used to align a line that contains multiple runs in different fonts. In this case, the line must be created piecemeal by aligning and writing each run separately.

Because rounding errors can occur during alignment, the system keeps a running error term that defines the current error. When aligning a line that contains multiple runs, **GetTextExtent** automatically uses this error term when it computes the extent of the next run. This allows the text-output function to blend the error into the new run.

After each line has been aligned, this error term must be cleared to prevent it from being incorporated into the next line. The term can be cleared by calling **SetTextJustification** with *nBreakExtra* set to 0.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetMapMode
CDC::GetTextExtent
CDC::GetTextMetrics
CDC::SetMapMode
CDC::TextOut
SetTextJustification

Other Resources

CDC Members

CDC::SetViewportExt

Sets the x- and y-extents of the viewport of the device context.

```
virtual CSize SetViewportExt(
   int cx,
   int cy
);
CSize SetViewportExt(
   SIZE size
);
```

Parameters

cx

Specifies the x-extent of the viewport (in device units).

cy

Specifies the y-extent of the viewport (in device units).

size

Specifies the x- and y-extents of the viewport (in device units).

Return Value

The previous extents of the viewport as a CSize object. When an error occurs, the x- and y-coordinates of the returned **CSize** object are both set to 0.

Remarks

The viewport, along with the device-context window, defines how GDI maps points in the logical coordinate system to points in the coordinate system of the actual device. In other words, they define how GDI converts logical coordinates into device coordinates.

When the following mapping modes are set, calls to **SetWindowExt** and **SetViewportExt** are ignored:

MM_HIENGLISH	**MM_LOMETRIC**
MM_HIMETRIC	**MM_TEXT**
MM_LOENGLISH	**MM_TWIPS**

When **MM_ISOTROPIC** mode is set, an application must call the **SetWindowExt** member function before it calls **SetViewportExt**.

Example

See the example for CView::OnPrepareDC.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetWindowExt
CSize Class
CDC::GetViewportExt

Other Resources

CDC Members

CDC::SetViewportOrg

Sets the viewport origin of the device context.

```
virtual CPoint SetViewportOrg(
   int x,
   int y
);
CPoint SetViewportOrg(
   POINT point
);
```

Parameters

x

Specifies the x-coordinate (in device units) of the origin of the viewport. The value must be within the range of the device coordinate system.

y

Specifies the y-coordinate (in device units) of the origin of the viewport. The value must be within the range of the device coordinate system.

point

Specifies the origin of the viewport. The values must be within the range of the device coordinate system. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Return Value

The previous origin of the viewport (in device coordinates) as a **CPoint** object.

Remarks

The viewport, along with the device-context window, defines how GDI maps points in the logical coordinate system to points in the coordinate system of the actual device. In other words, they define how GDI converts logical coordinates into device coordinates.

The viewport origin marks the point in the device coordinate system to which GDI maps the window origin, a point in the logical coordinate system specified by the **SetWindowOrg** member function. GDI maps all other points by following the same process required to map the window origin to the viewport origin. For example, all points in a circle around the point at the window origin will be in a circle around the point at the viewport origin. Similarly, all points in a line that passes through the window origin will be in a line that passes through the viewport origin.

Example

See the example for CView::OnPrepareDC.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::SetWindowOrg
CPoint Class
POINT Structure
CDC::GetViewportOrg

Other Resources

CDC Members

CDC::SetWindowExt

Sets the x- and y-extents of the window associated with the device context.

```
virtual CSize SetWindowExt(
   int cx,
   int cy
);
CSize SetWindowExt(
   SIZE size
);
```

Parameters

cx

Specifies the x-extent (in logical units) of the window.

cy

Specifies the y-extent (in logical units) of the window.

size

Specifies the x- and y-extents (in logical units) of the window.

Return Value

The previous extents of the window (in logical units) as a **CSize** object. If an error occurs, the x- and y-coordinates of the returned **CSize** object are both set to 0.

Remarks

The window, along with the device-context viewport, defines how GDI maps points in the logical coordinate system to points in the device coordinate system.

When the following mapping modes are set, calls to **SetWindowExt** and **SetViewportExt** functions are ignored:

- **MM_HIENGLISH**
- **MM_HIMETRIC**
- **MM_LOENGLISH**
- **MM_LOMETRIC**
- **MM_TEXT**
- **MM_TWIPS**

When **MM_ISOTROPIC** mode is set, an application must call the **SetWindowExt** member function before calling **SetViewportExt**.

Example

See the example for CView::OnPrepareDC.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetWindowExt
CDC::SetViewportExt
CSize Class

Other Resources

CDC Members

CDC::SetWindowOrg

Sets the window origin of the device context.

```
CPoint SetWindowOrg(
   int x,
   int y
);
CPoint SetWindowOrg(
   POINT point
);
```

Parameters

x

Specifies the logical x-coordinate of the new origin of the window.

y

Specifies the logical y-coordinate of the new origin of the window.

point

Specifies the logical coordinates of the new origin of the window. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Return Value

The previous origin of the window as a **CPoint** object.

Remarks

The window, along with the device-context viewport, defines how GDI maps points in the logical coordinate system to points in the device coordinate system.

The window origin marks the point in the logical coordinate system from which GDI maps the viewport origin, a point in the device coordinate system specified by the **SetWindowOrg** function. GDI maps all other points by following the same process required to map the window origin to the viewport origin. For example, all points in a circle around the point at the window origin will be in a circle around the point at the viewport origin. Similarly, all points in a line that passes through the window origin will be in a line that passes through the viewport origin.

See Also

Reference

CDC Class
Hierarchy Chart
CPoint Class
POINT Structure
CDC::GetWindowOrg

Other Resources

CDC Members

CDC::SetWorldTransform

Sets a two-dimensional linear transformation between world space and page space for the specified device context. This transformation can be used to scale, rotate, shear, or translate graphics output.

```
BOOL SetWorldTransform(
   const XFORM& rXform
);
```

Parameters

rXform

Reference to an XFORM structure that contains the transformation data.

Return Value

Returns a nonzero value on success.

Returns 0 on failure.

In Windows NT, Windows 2000, and Windows XP, to get extended error information, call GetLastError.

Remarks

This method wraps the Windows GDI function SetWorldTransform.

See Also

Reference

CDC Class

Hierarchy Chart

Other Resources

CDC Members

CDC::StartDoc

Informs the device driver that a new print job is starting and that all subsequent **StartPage** and **EndPage** calls should be spooled under the same job until an **EndDoc** call occurs.

```
int StartDoc(
   LPDOCINFO lpDocInfo
);
int StartDoc(
   LPCTSTR lpszDocName
);
```

Parameters

lpDocInfo

Points to a DOCINFO structure containing the name of the document file and the name of the output file.

lpszDocName

Pointer to a string containing the name of the document file.

Return Value

If the function succeeds, the return value is greater than zero. This value is the print job identifier for the document.

If the function fails, the return value is less than or equal to zero.

Remarks

This ensures that documents longer than one page will not be interspersed with other jobs.

For Windows versions 3.1 and later, this function replaces the **STARTDOC** printer escape. Using this function ensures that documents containing more than one page are not interspersed with other print jobs.

StartDoc should not be used inside metafiles.

Example

This code fragment gets the default printer, opens a print job, and spools one page with "Hello, World!" on it. Because the text printed by this code isn't scaled to the printer's logical units, the output text may be in such small letters that the result is unreadable. The CDC scaling functions, such as **SetMapMode**, **SetViewportOrg**, and **SetWindowExt**, can be used to fix the scaling.

```
// get the default printer
CPrintDialog dlg(FALSE);
dlg.GetDefaults();

// is a default printer set up?
HDC hdcPrinter = dlg.GetPrinterDC();
if (hdcPrinter == NULL)
{
   MessageBox(_T("Buy a printer!"));
}
else
{
   // create a CDC and attach it to the default printer
   CDC dcPrinter;
   dcPrinter.Attach(hdcPrinter);

   // call StartDoc() to begin printing
   DOCINFO docinfo;
   memset(&docinfo, 0, sizeof(docinfo));
   docinfo.cbSize = sizeof(docinfo);
   docinfo.lpszDocName = _T("CDC::StartDoc() Code Fragment");

   // if it fails, complain and exit gracefully
```

```
    if (dcPrinter.StartDoc(&docinfo) < 0)
    {
       MessageBox(_T("Printer wouldn't initalize"));
    }
    else
    {
       // start a page
       if (dcPrinter.StartPage() < 0)
       {
          MessageBox(_T("Could not start page"));
          dcPrinter.AbortDoc();
       }
       else
       {
          // actually do some printing
          CGdiObject* pOldFont = dcPrinter.SelectStockObject(SYSTEM_FONT);

          dcPrinter.TextOut(50, 50, _T("Hello World!"), 12);

          dcPrinter.EndPage();
          dcPrinter.EndDoc();
          dcPrinter.SelectObject(pOldFont);
       }
    }
 }
```

See Also

Reference

CDC Class

Hierarchy Chart

CDC::Escape

CDC::EndDoc

CDC::AbortDoc

Other Resources

CDC Members

CDC::StartPage

Call this member function to prepare the printer driver to receive data.

```
int StartPage( );
```

Return Value

Greater than or equal to 0 if the function is successful, or a negative value if an error occurred.

Remarks

StartPage supersedes the **NEWFRAME** and **BANDINFO** escapes.

For an overview of the sequence of printing calls, see the StartDoc member function.

The system disables the **ResetDC** member function between calls to **StartPage** and **EndPage**.

Example

See the example for CDC::StartDoc.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::Escape
CDC::EndPage

Other Resources

CDC Members

CDC::StretchBlt

Copies a bitmap from a source rectangle into a destination rectangle, stretching or compressing the bitmap if necessary to fit the dimensions of the destination rectangle.

```
BOOL StretchBlt(
   int x,
   int y,
   int nWidth,
   int nHeight,
   CDC* pSrcDC,
   int xSrc,
   int ySrc,
   int nSrcWidth,
   int nSrcHeight,
   DWORD dwRop
);
```

Parameters

x

Specifies the x-coordinate (in logical units) of the upper-left corner of the destination rectangle.

y

Specifies the y-coordinate (in logical units) of the upper-left corner of the destination rectangle.

nWidth

Specifies the width (in logical units) of the destination rectangle.

nHeight

Specifies the height (in logical units) of the destination rectangle.

pSrcDC

Specifies the source device context.

xSrc

Specifies the x-coordinate (in logical units) of the upper-left corner of the source rectangle.

ySrc

Specifies the x-coordinate (in logical units) of the upper-left corner of the source rectangle.

nSrcWidth

Specifies the width (in logical units) of the source rectangle.

nSrcHeight

Specifies the height (in logical units) of the source rectangle.

dwRop

Specifies the raster operation to be performed. Raster operation codes define how GDI combines colors in output operations that involve a current brush, a possible source bitmap, and a destination bitmap. This parameter may be one of the following values:

- **BLACKNESS** Turns all output black.
- **DSTINVERT** Inverts the destination bitmap.
- **MERGECOPY** Combines the pattern and the source bitmap using the Boolean AND operator.
- **MERGEPAINT** Combines the inverted source bitmap with the destination bitmap using the Boolean OR operator.

- **NOTSRCCOPY** Copies the inverted source bitmap to the destination.
- **NOTSRCERASE** Inverts the result of combining the destination and source bitmaps using the Boolean OR operator.
- **PATCOPY** Copies the pattern to the destination bitmap.
- **PATINVERT** Combines the destination bitmap with the pattern using the Boolean XOR operator.
- **PATPAINT** Combines the inverted source bitmap with the pattern using the Boolean OR operator. Combines the result of this operation with the destination bitmap using the Boolean OR operator.
- **SRCAND** Combines pixels of the destination and source bitmaps using the Boolean AND operator.
- **SRCCOPY** Copies the source bitmap to the destination bitmap.
- **SRCERASE** Inverts the destination bitmap and combines the result with the source bitmap using the Boolean AND operator.
- **SRCINVERT** Combines pixels of the destination and source bitmaps using the Boolean XOR operator.
- **SRCPAINT** Combines pixels of the destination and source bitmaps using the Boolean OR operator.
- **WHITENESS** Turns all output white.

Return Value

Nonzero if the bitmap is drawn; otherwise 0.

Remarks

The function uses the stretching mode of the destination device context (set by **SetStretchBltMode**) to determine how to stretch or compress the bitmap.

The **StretchBlt** function moves the bitmap from the source device given by *pSrcDC* to the destination device represented by the device-context object whose member function is being called. The *xSrc*, *ySrc*, *nSrcWidth*, and *nSrcHeight* parameters define the upper-left corner and dimensions of the source rectangle. The *x*, *y*, *nWidth*, and *nHeight* parameters give the upper-left corner and dimensions of the destination rectangle. The raster operation specified by *dwRop* defines how the source bitmap and the bits already on the destination device are combined.

The **StretchBlt** function creates a mirror image of a bitmap if the signs of the *nSrcWidth* and *nWidth* or *nSrcHeight* and *nHeight* parameters differ. If *nSrcWidth* and *nWidth* have different signs, the function creates a mirror image of the bitmap along the x-axis. If *nSrcHeight* and *nHeight* have different signs, the function creates a mirror image of the bitmap along the y-axis.

The **StretchBlt** function stretches or compresses the source bitmap in memory and then copies the result to the destination. If a pattern is to be merged with the result, it is not merged until the stretched source bitmap is copied to the destination. If a brush is used, it is the selected brush in the destination device context. The destination coordinates are transformed according to the destination device context; the source coordinates are transformed according to the source device context.

If the destination, source, and pattern bitmaps do not have the same color format, **StretchBlt** converts the source and pattern bitmaps to match the destination bitmaps. The foreground and background colors of the destination device context are used in the conversion.

If **StretchBlt** must convert a monochrome bitmap to color, it sets white bits (1) to the background color and black bits (0) to the foreground color. To convert color to monochrome, it sets pixels that match the background color to white (1) and sets all other pixels to black (0). The foreground and background colors of the device context with color are used.

Not all devices support the **StretchBlt** function. To determine whether a device supports **StretchBlt**, call the **GetDeviceCaps** member function with the **RASTERCAPS** index and check the return value for the **RC_STRETCHBLT** flag.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::BitBlt
CDC::GetDeviceCaps
CDC::SetStretchBltMode
StretchBlt

Other Resources

CDC Members

CDC::StrokeAndFillPath

Closes any open figures in a path, strokes the outline of the path by using the current pen, and fills its interior by using the current brush.

```
BOOL StrokeAndFillPath( );
```

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The device context must contain a closed path. The **StrokeAndFillPath** member function has the same effect as closing all the open figures in the path, and stroking and filling the path separately, except that the filled region will not overlap the stroked region even if the pen is wide.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::BeginPath
CDC::FillPath
CDC::SetPolyFillMode
CDC::StrokePath
StrokeAndFillPath

Other Resources

CDC Members

CDC::StrokePath

Renders the specified path by using the current pen.

```
BOOL StrokePath( );
```

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The device context must contain a closed path.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::BeginPath
CDC::EndPath
StrokePath

Other Resources

CDC Members

CDC::TabbedTextOut

Call this member function to write a character string at the specified location, expanding tabs to the values specified in the array of tab-stop positions.

```
virtual CSize TabbedTextOut(
   int x,
   int y,
   LPCTSTR lpszString,
   int nCount,
   int nTabPositions,
   LPINT lpnTabStopPositions,
   int nTabOrigin
);
CSize TabbedTextOut(
   int x,
   int y,
   const CString& str,
   int nTabPositions,
   LPINT lpnTabStopPositions,
   int nTabOrigin
);
```

Parameters

x

Specifies the logical x-coordinate of the starting point of the string.

y

Specifies the logical y-coordinate of the starting point of the string.

lpszString

Points to the character string to draw. You can pass either a pointer to an array of characters or a CString object for this parameter.

nCount

Specifies the number of characters in the string. If *nCount* is –1, the length is calculated.

nTabPositions

Specifies the number of values in the array of tab-stop positions.

lpnTabStopPositions

Points to an array containing the tab-stop positions (in logical units). The tab stops must be sorted in increasing order; the smallest x-value should be the first item in the array.

nTabOrigin

Specifies the x-coordinate of the starting position from which tabs are expanded (in logical units).

str

A **CString** object that contains the specified characters.

Return Value

The dimensions of the string (in logical units) as a **CSize** object.

Remarks

Text is written in the currently selected font. If *nTabPositions* is 0 and *lpnTabStopPositions* is **NULL**, tabs are expanded to eight times the average character width.

If *nTabPositions* is 1, the tab stops are separated by the distance specified by the first value in the *lpnTabStopPositions* array. If

the *lpnTabStopPositions* array contains more than one value, a tab stop is set for each value in the array, up to the number specified by *nTabPositions*. The *nTabOrigin* parameter allows an application to call the **TabbedTextOut** function several times for a single line. If the application calls the function more than once with the *nTabOrigin* set to the same value each time, the function expands all tabs relative to the position specified by *nTabOrigin*.

By default, the current position is not used or updated by the function. If an application needs to update the current position when it calls the function, the application can call the SetTextAlign member function with *nFlags* set to **TA_UPDATECP**. When this flag is set, Windows ignores the *x* and *y* parameters on subsequent calls to **TabbedTextOut**, using the current position instead.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::GetTabbedTextExtent
CDC::SetTextAlign
CDC::TextOut
CDC::SetTextColor
TabbedTextOut
CSize Class

Other Resources

CDC Members

CDC::TextOut

Writes a character string at the specified location using the currently selected font.

```
virtual BOOL TextOut(
   int x,
   int y,
   LPCTSTR lpszString,
   int nCount
);
BOOL TextOut(
   int x,
      int y,
      const CString& str
);
```

Parameters

x

Specifies the logical x-coordinate of the starting point of the text.

y

Specifies the logical y-coordinate of the starting point of the text.

lpszString

Points to the character string to be drawn.

nCount

Specifies the number of bytes in the string.

str

A **CString** object that contains the characters to be drawn.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

Character origins are at the upper-left corner of the character cell. By default, the current position is not used or updated by the function.

If an application needs to update the current position when it calls **TextOut**, the application can call the **SetTextAlign** member function with *nFlags* set to **TA_UPDATECP**. When this flag is set, Windows ignores the *x* and *y* parameters on subsequent calls to **TextOut**, using the current position instead.

Example

See the example for CDC::BeginPath.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::ExtTextOut
CDC::GetTextExtent
CDC::SetTextAlign
CDC::SetTextColor
CDC::TabbedTextOut
TextOut

Other Resources

CDC Members

CDC::TransparentBlt

Call this member function to transfer a bit-block of the color data, which corresponds to a rectangle of pixels from the specified source device context, into a destination device context.

```
BOOL TransparentBlt(
   int xDest,
   int yDest,
      int nDestWidth,
      int nDestHeight,
   CDC* pSrcDC,
   int xSrc,
   int ySrc,
   int nSrcWidth,
   int nSrcHeight,
   UINT clrTransparent
);
```

Parameters

xDest

Specifies the x-coordinate, in logical units, of the upper-left corner of the destination rectangle.

yDest

Specifies the y-coordinate, in logical units, of the upper-left corner of the destination rectangle.

nDestWidth

Specifies the width, in logical units, of the destination rectangle.

nDestHeight

Specifies the height, in logical units, of the destination rectangle.

pSrcDC

Pointer to the source device context.

xSrc

Specifies the x-coordinate, in logical units, of the source rectangle.

ySrc

Specifies the y-coordinate, in logical units, of the source rectangle.

nSrcWidth

Specifies the width, in logical units, of the source rectangle.

nSrcHeight

Specifies the height, in logical units, of the source rectangle.

clrTransparent

The RGB color in the source bitmap to treat as transparent.

Return Value

TRUE if successful; otherwise **FALSE**.

Remarks

TransparentBlt allows for transparency; that is, the RGB color indicated by *clrTransparent* is rendered transparent for the transfer.

For more information, see TransparentBlt in the *Platform SDK*.

See Also

Reference

CDC Class

Hierarchy Chart

CDC::AlphaBlend

CDC::SetStretchBltMode

Other Resources

CDC Members

CDC::UpdateColors

Updates the client area of the device context by matching the current colors in the client area to the system palette on a pixel-by-pixel basis.

```
void UpdateColors( );
```

Remarks

An inactive window with a realized logical palette may call **UpdateColors** as an alternative to redrawing its client area when the system palette changes.

For more information about using color palettes, see UpdateColors in the *Platform SDK*.

The **UpdateColors** member function typically updates a client area faster than redrawing the area. However, because the function performs the color translation based on the color of each pixel before the system palette changed, each call to this function results in the loss of some color accuracy.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::RealizePalette
CPalette Class
UpdateColors

Other Resources

CDC Members

CDC::WidenPath

Redefines the current path as the area that would be painted if the path were stroked using the pen currently selected into the device context.

```
BOOL WidenPath( );
```

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

This function is successful only if the current pen is a geometric pen created by the second version of **CreatePen** member function, or if the pen is created with the first version of **CreatePen** and has a width, in device units, of greater than 1. The device context must contain a closed path. Any Bzier curves in the path are converted to sequences of straight lines approximating the widened curves. As such, no Bzier curves remain in the path after **WidenPath** is called.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::BeginPath
CDC::EndPath
CDC::SetMiterLimit
WidenPath

Other Resources

CDC Members

CDC Data Members

For information about the data members in CDC, see CDC Members.

CDC::m_hAttribDC

The attribute device context for this **CDC** object.

```
HDC m_hAttribDC;
```

Remarks

By default, this device context is equal to **m_hDC**. In general, **CDC** GDI calls that request information from the device context are directed to **m_hAttribDC**. See the CDC class description for more on the use of these two device contexts.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::m_hDC
CDC::SetAttribDC
CDC::ReleaseAttribDC

Other Resources

CDC Members

CDC::m_hDC

The output device context for this **CDC** object.

```
HDC m_hDC;
```

Remarks

By default, **m_hDC** is equal to **m_hAttribDC**, the other device context wrapped by **CDC**. In general, **CDC** GDI calls that create output go to the **m_hDC** device context. You can initialize **m_hDC** and **m_hAttribDC** to point to different devices. See the CDC class description for more on the use of these two device contexts.

See Also

Reference

CDC Class
Hierarchy Chart
CDC::m_hAttribDC
CDC::SetOutputDC
CDC::ReleaseOutputDC

Other Resources

CDC Members

CDC Operators

For information about the operators in **CDC**, see CDC Members.

CDC::operator HDC

Use this operator to retrieve the device context handle of the **CDC** object.

```
operator HDC( ) const;
```

Return Value

If successful, the handle of the device context object; otherwise, **NULL**.

Remarks

You can use the handle to call Windows APIs directly.

See Also

Reference

CDC Class

Hierarchy Chart

Other Resources

CDC Members

CDHtmlDialog Class

Is used to create dialog boxes that use HTML rather than dialog resources to implement their user interface.

```
class CDHtmlDialog : public CDialog, public CDHtmlEventSink
```

Remarks

CDHtmlDialog can load the HTML to be displayed from either an HTML resource or a URL.

CDHtmlDialog can also do data exchange with HTML controls and handle events from HTML controls, such as button clicks.

Requirements

Header: afxdhtml.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

DHtmlExplore Sample: Demonstrates Using MFC DHtml Classes

Reference

DDX_DHtml Helper Macros

Hierarchy Chart

Other Resources

CDHtmlDialog Members

CDHtmlDialog Members

Constructor/Destructor

CDHtmlDialog	Constructs a CDHtmlDialog object.
~CDHtmlDialog	Destroys a CDHtmlDialog object.

Member Functions

CanAccessExternal	Overridable that is called as an access check to see whether scripting objects on the loaded page can access the external dispatch of the control site. Checks to make sure the dispatch is either safe for scripting or the current zone allows for objects that are not safe for scripting.
CreateControlSite	Overridable used to create a control site instance to host the WebBrowser control on the dialog.
DDX_DHtml_AxControl	Exchanges data between a member variable and the property value of an ActiveX control on an HTML page.
DDX_DHtml_CheckBox	Exchanges data between a member variable and a check box on an HTML page.
DDX_DHtml_ElementText	Exchanges data between a member variable and any HTML element property on an HTML page.
DDX_DHtml_Radio	Exchanges data between a member variable and a radio button on an HTML page.
DDX_DHtml_SelectIndex	Gets or sets the index of a list box on an HTML page.
DDX_DHtml_SelectString	Gets or sets the display text of a list box entry (based on the current index) on an HTML page.
DDX_DHtml_SelectValue	Gets or sets the value of a list box entry (based on the current index) on an HTML page.
DestroyModeless	Destroys a modeless dialog box.
EnableModeless	Enables modeless dialog boxes.
FilterDataObject	Allows the dialog to filter clipboard data objects created by the hosted browser.
GetControlDispatch	Retrieves the **IDispatch** interface on an ActiveX control embedded in the HTML document.
GetControlProperty	Retrieves the requested property of the specified ActiveX control.
GetCurrentUrl	Retrieves the Uniform Resource Locator (URL) associated with the current document.
GetDHtmlDocument	Retrieves the IHTMLDocument2 interface on the currently loaded HTML document.
GetDropTarget	Called by the contained WebBrowser control when it is being used as a drop target to allow the dialog to supply an alternative IDropTarget.
GetElement	Gets an interface on an HTML element.
GetElementHtml	Retrieves the **innerHTML** property of an HTML element.
GetElementInterface	Retrieves the requested interface pointer from an HTML element.

GetElementProperty	Retrieves the value of an HTML element's property.
GetElementText	Retrieves the **innerText** property of an HTML element.
GetEvent	Gets the **IHTMLEventObj** pointer to the current event object.
GetExternal	Gets the host's **IDispatch** interface.
GetHostInfo	Retrieves the host's UI capabilities.
GetOptionKeyPath	Retrieves the registry key under which user preferences are stored.
HideUI	Hides the host's UI.
IsExternalDispatchSafe	Indicates whether the host's **IDispatch** interface is safe for scripting.
LoadFromResource	Loads the specified resource into the WebBrowser control.
Navigate	Navigates to the specified URL.
OnBeforeNavigate	Called by the framework before a navigation event is fired.
OnDocumentComplete	Called by the framework to notify an application when a document has reached the **READYSTATE_COMPLETE** state.
OnDocWindowActivate	Called by the framework when the document window is activated or deactivated.
OnFrameWindowActivate	Called by the framework when the frame window is activated or deactivated.
OnInitDialog	Called in response to the **WM_INITDIALOG** message.
OnNavigateComplete	Called by the framework after a navigation event is completed.
ResizeBorder	Alerts the object that it needs to resize its border space.
SetControlProperty	Sets the property of an ActiveX control to a new value.
SetElementHtml	Sets the **innerHTML** property of an HTML element.
SetElementProperty	Sets a property of an HTML element.
SetElementText	Sets the **innerText** property of an HTML element.
SetExternalDispatch	Sets the host's **IDispatch** interface.
SetHostFlags	Sets the host's UI flags.
ShowContextMenu	Called when a context menu is about to be displayed.
ShowUI	Shows the host's UI.
TranslateAccelerator	Called to process menu accelerator-key messages.

TranslateUrl	Called to modify the URL to be loaded.
UpdateUI	Called to notify the host that the command state has changed.

Data Members

m_bUseHtmlTitle	Indicates whether to use the HTML document's title as the dialog caption.
m_nHtmlResID	Resource ID of HTML resource to be displayed.
m_pBrowserApp	A pointer to a Web browser application.
m_spHtmlDoc	A pointer to an HTML document.
m_strCurrentUrl	The current URL.
m_szHtmlResID	String version of the HTML resource ID.

See Also

Reference

DDX_DHtml Helper Macros
CDHtmlDialog Class

CDHtmlDialog Member Functions

For information about the member functions in CDHtmlDialog, see CDHtmlDialog Members.

CDHtmlDialog::CanAccessExternal

Overridable that is called as an access check to see whether scripting objects on the loaded page can access the external dispatch of the control site. Checks to make sure the dispatch is either safe for scripting or the current zone allows for objects that are not safe for scripting.

```
virtual BOOL CanAccessExternal( );
```

Return Value

Nonzero if successful; otherwise 0.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
CDHtmlDialog::IsExternalDispatchSafe

Other Resources

CDHtmlDialog Members

CDHtmlDialog::CDHtmlDialog

Constructs a resource-based dynamic HTML dialog box.

```
CDHtmlDialog( );
CDHtmlDialog(
   LPCTSTR lpszTemplateName,
   LPCTSTR szHtmlResID,
   CWnd *pParentWnd = NULL
);
CDHtmlDialog(
   UINT nIDTemplate,
   UINT nHtmlResID = 0,
   CWnd *pParentWnd = NULL
);
```

Parameters

lpszTemplateName

The null-terminated string that is the name of a dialog-box template resource.

szHtmlResID

The null-terminated string that is the name of an HTML resource.

pParentWnd

A pointer to the parent or owner window object (of type CWnd) to which the dialog object belongs. If it is **NULL**, the dialog object's parent window is set to the main application window.

nIDTemplate

Contains the ID number of a dialog-box template resource.

nHtmlResID

Contains the ID number of an HTML resource.

Remarks

The second form of the constructor provides access to the dialog resource through the template name. The third form of the constructor provides access to the dialog resource through the ID of the resource template. Usually, the ID begins with the **IDD_** prefix.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
CDHtmlDialog::LoadFromResource

Other Resources

CDHtmlDialog Members

CDHtmlDialog::~CDHtmlDialog

Destroys a CDHtmlDialog object.

```
virtual ~CDHtmlDialog( );
```

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

Other Resources

CDHtmlDialog Members

CDHtmlDialog::CreateControlSite

Overridable used to create a control site instance to host the WebBrowser control on the dialog.

```
virtual BOOL CreateControlSite(
   COleControlContainer* pContainer,
   COleControlSite** ppSite,
   UINT /* nID */,
   REFCLSID /* clsid */
);
```

Parameters

pContainer

A pointer to the COleControlContainer object

ppSite

A pointer to a pointer to a COleControlSite.

Return Value

Nonzero if successful; otherwise 0.

Remarks

You can override this member function to return an instance of your own control site class.

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

Other Resources

CDHtmlDialog Members

CDHtmlDialog::DDX_DHtml_AxControl

Exchanges data between a member variable and the property value of an ActiveX control on an HTML page.

```
void DDX_DHtml_AxControl(
   CDataExchange* pDX,
   LPCTSTR szId,
   DISPID dispid,
   VARIANT& var
);
void DDX_DHtml_AxControl(
   CDataExchange* pDX,
   LPCTSTR szId,
   LPCTSTR szPropName,
   VARIANT& var
);
```

Parameters

pDX

A pointer to a CDataExchange object.

szId

The value of the object tag's ID parameter in the HTML source for the ActiveX control.

dispid

The dispatch ID of the property with which you want to exchange data.

szPropName

The name of the property.

var

The data member, of type VARIANT, COleVariant, or CComVariant, that holds the value exchanged with the ActiveX control property.

Example

```
// COleVariant m_varSliderValue;
DDX_DHtml_AxControl( pDX, L"slider1", 0x0b /* Value */, m_varSliderValue );
```

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

Other Resources

CDHtmlDialog Members

CDHtmlDialog::DDX_DHtml_CheckBox

Exchanges data between a member variable and a check box on an HTML page.

```
void DDX_DHtml_CheckBox(
   CDataExchange* pDX,
   LPCTSTR szId,
   int& value
);
```

Parameters

pDX

A pointer to a CDataExchange object.

szId

The value that you specified for the HTML control's ID parameter.

value

The value being exchanged.

Example

```
// int m_nItalic;
DDX_DHtml_CheckBox( pDX, L"italic", m_nItalic );
```

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

Other Resources

CDHtmlDialog Members

CDHtmlDialog::DDX_DHtml_ElementText

Exchanges data between a member variable and any HTML element property on an HTML page.

```
void DDX_DHtml_ElementText(
   CDataExchange* pDX,
   LPCTSTR szId,
   DISPID dispid,
   CString& value
);
void DDX_DHtml_ElementText(
   CDataExchange* pDX,
   LPCTSTR szId,
   DISPID dispid,
   short& value
);
void DDX_DHtml_ElementText(
   CDataExchange* pDX,
   LPCTSTR szId,
   DISPID dispid,
   int& value
);
void DDX_DHtml_ElementText(
   CDataExchange* pDX,
   LPCTSTR szId,
   DISPID dispid,
   long& value
);
void DDX_DHtml_ElementText(
   CDataExchange* pDX,
   LPCTSTR szId,
   DISPID dispid,
   DWORD& value
);
void DDX_DHtml_ElementText(
   CDataExchange* pDX,
   LPCTSTR szId,
   DISPID dispid,
   float& value
);
void DDX_DHtml_ElementText(
   CDataExchange* pDX,
   LPCTSTR szId,
   DISPID dispid,
   double& value
);
```

Parameters

pDX

A pointer to a CDataExchange object.

szId

The value that you specified for the HTML control's ID parameter.

dispid

The dispatch ID of the HTML element with which you want to exchange data.

value

The value being exchanged.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
Other Resources
CDHtmlDialog Members

CDHtmlDialog::DDX_DHtml_Radio

Exchanges data between a member variable and a radio button on an HTML page.

```
void DDX_DHtml_Radio(
   CDataExchange* pDX,
   LPCTSTR szId,
   long& value
);
```

Parameters

pDX

A pointer to a CDataExchange object.

szId

The value that you specified for the HTML control's ID parameter.

value

The value being exchanged.

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

Other Resources

CDHtmlDialog Members

CDHtmlDialog::DDX_DHtml_SelectIndex

Gets or sets the index of a list box on an HTML page.

```
void DDX_DHtml_SelectIndex(
   CDataExchange* pDX,
   LPCTSTR szId,
   long& value
);
```

Parameters

pDX

A pointer to a CDataExchange object.

szId

The value that you specified for the HTML control's id parameter.

value

The value being exchanged.

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

Other Resources

CDHtmlDialog Members

CDHtmlDialog::DDX_DHtml_SelectString

Gets or sets the display text of a list box entry (based on the current index) on an HTML page.

```
void DDX_DHtml_SelectString(
   CDataExchange* pDX,
   LPCTSTR szId,
   CString& value
);
```

Parameters

pDX

A pointer to a CDataExchange object.

szId

The value that you specified for the HTML control's ID parameter.

value

The value being exchanged.

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

Other Resources

CDHtmlDialog Members

CDHtmlDialog::DDX_DHtml_SelectValue

Gets or sets the value of a list box entry (based on the current index) on an HTML page.

```
void DDX_DHtml_SelectValue(
   CDataExchange* pDX,
   LPCTSTR szId,
   CString& value
);
```

Parameters

pDX

A pointer to a CDataExchange object.

szId

The value that you specified for the HTML control's ID parameter.

value

The value being exchanged.

Example

```
// CString m_strBlurDir;
DDX_DHtml_SelectValue( pDX, L"blurDir", m_strBlurDir );
```

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

Other Resources

CDHtmlDialog Members

CDHtmlDialog::DestroyModeless

Detaches a modeless dialog box from the **CDHtmlDialog** object and destroys the object.

```
void DestroyModeless( );
```

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

Other Resources

CDHtmlDialog Members

CDHtmlDialog::EnableModeless

Enables modeless dialog boxes.

```
STDMETHOD(EnableModeless)(
   BOOL fEnable
);
```

Parameters

fEnable

See *fEnable* in IDocHostUIHandler::EnableModeless in the *Platform SDK*.

Return Value

Returns **E_NOTIMPL**.

Remarks

This member function is CDHtmlDialog's implementation of IDocHostUIHandler::EnableModeless, as described in the *Platform SDK*.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
IDocHostUIhandler Interface

Other Resources

CDHtmlDialog Members

CDHtmlDialog::FilterDataObject

Allows the dialog to filter clipboard data objects created by the hosted browser.

```
STDMETHOD(FilterDataObject)(
   IDataObject *pDO,
   IDataObject **ppDORet
);
```

Parameters

pDO

See *pDO* in IDocHostUIHandler::FilterDataObject in the *Platform SDK*.

ppDORet

See *ppDORet* in **IDocHostUIHandler::FilterDataObject** in the *Platform SDK*.

Return Value

Returns **S_FALSE**.

Remarks

This member function is CDHtmlDialog's implementation of IDocHostUIHandler::FilterDataObject, as described in the *Platform SDK*.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
IDocHostUIhandler Interface

Other Resources

CDHtmlDialog Members

CDHtmlDialog::GetControlDispatch

Retrieves the **IDispatch** interface on an ActiveX control embedded in the HTML document returned by GetDHtmlDocument.

```
HRESULT GetControlDispatch(
   LPCTSTR szId,
   IDispatch **ppdisp
);
```

Parameters

szId

The HTML ID of an ActiveX control.

ppdisp

The **IDispatch** interface of the control if found in the Web page.

Return Value

A standard **HRESULT** value.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
CDHtmlDialog::GetElement
CDHtmlDialog::GetElementInterface

Other Resources

CDHtmlDialog Members

CDHtmlDialog::GetControlProperty

Retrieves the requested property of the specified ActiveX control.

```
VARIANT GetControlProperty(
   LPCTSTR szId,
   LPCTSTR szPropName
);
VARIANT GetControlProperty(
   LPCTSTR szId,
   DISPID dispid
);
VARIANT GetControlProperty(
   IDispatch* pdispControl,
   DISPID dispid
);
```

Parameters

szId

The HTML ID of an ActiveX control.

szPropName

The name of a property in the default locale of the current user.

pdispControl

The **IDispatch** pointer of an ActiveX control.

dispid

The dispatch ID of a property.

Return Value

A variant containing the requested property or an empty variant if the control or property could not be found.

Remarks

The overloads are listed from least efficient at the top to most efficient at the bottom.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
CDHtmlDialog::GetControlDispatch

Other Resources

CDHtmlDialog Members

CDHtmlDialog::GetCurrentUrl

Retrieves the Uniform Resource Locator (URL) associated with the current document.

```
void GetCurrentUrl(
   CString& szUrl
);
```

Parameters

szUrl

A CString object containing the URL to retrieve.

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

Other Resources

CDHtmlDialog Members

CDHtmlDialog::GetDHtmlDocument

Retrieves the IHTMLDocument2 interface on the currently loaded HTML document.

```
HRESULT GetDHtmlDocument(
   IHTMLDocument2 **pphtmlDoc
);
```

Parameters

***pphtmlDoc*

A pointer to a pointer to an HTML document.

Return Value

A standard **HRESULT**. Returns **S_OK** if successful.

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

IHTMLDocument2

Other Resources

CDHtmlDialog Members

CDHtmlDialog::GetDropTarget

Called by the contained WebBrowser control when it is being used as a drop target to allow the dialog to supply an alternative IDropTarget.

```
STDMETHOD(GetDropTarget)(
   IDropTarget *pDropTarget,
   IDropTarget **ppDropTarget
);
```

Parameters

pDropTarget

See *pDropTarget* in IDocHostUIHandler::GetDropTarget in the *Platform SDK*.

ppDropTarget

See *ppDropTarget* in **IDocHostUIHandler::GetDropTarget** in the *Platform SDK*.

Return Value

Returns **E_NOTIMPL**.

Remarks

This member function is CDHtmlDialog's implementation of IDocHostUIHandler::GetDropTarget, as described in the *Platform SDK*.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
IDocHostUIhandler Interface

Other Resources

CDHtmlDialog Members

CDHtmlDialog::GetElement

Returns an interface on the HTML element specified by *szElementId*.

```
HRESULT GetElement(
   LPCTSTR szElementId,
   IDispatch **ppdisp,
   BOOL *pbCollection = NULL
);
HRESULT GetElement(
   LPCTSTR szElementId,
   IHTMLElement **pphtmlElement
);
```

Parameters

szElementId

The ID of an HTML element.

ppdisp

An **IDispatch** pointer to the requested element or collection of elements.

pbCollection

A **BOOL** indicating whether the object represented by *ppdisp* is a single element or a collection of elements.

pphtmlElement

An **IHTMLElement** pointer to the requested element.

Return Value

A standard **HRESULT** value.

Remarks

Use the first overload if you need to handle conditions in which there may be more than one element with the specified ID. You can use the last parameter to find out whether the returned interface pointer is to a collection or a single item. If the interface pointer is on a collection, you can query for the **IHTMLElementCollection** and use its **item** property to refer to the elements by ordinal position.

The second overload will fail if there is more than one element with the same ID in the page.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
CDHtmlDialog::GetElementInterface
CDHtmlDialog::GetControlDispatch

Other Resources

CDHtmlDialog Members

CDHtmlDialog::GetElementHtml

Retrieves the **innerHTML** property of the HTML element identified by *szElementId*.

```
BSTR GetElementHtml(
   LPCTSTR szElementId
);
```

Parameters

szElementId

The ID of an HTML element.

Return Value

The **innerHTML** property of the HTML element identified by *szElementId* or **NULL** if the element could not be found.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
CDHtmlDialog::GetElementText
CDHtmlDialog::GetElementProperty

Other Resources

CDHtmlDialog Members

CDHtmlDialog::GetElementInterface

Retrieves the requested interface pointer from the HTML element identified by *szElementId*.

```
template <class Q>
HRESULT GetElementInterface(
   LPCTSTR szElementId,
   Q** ppvObj
);
HRESULT GetElementInterface(
   LPCTSTR szElementId,
   REFIID riid,
   void** ppvObj
);
```

Parameters

szElementId

The ID of an HTML element.

ppvObj

Address of a pointer that will be filled with the requested interface pointer if the element is found and the query succeeds.

riid

The interface ID (IID) of the requested interface.

Return Value

A standard **HRESULT** value.

Example

```
CComPtr<IHTMLButtonElement> spBtn;
HRESULT hr = S_OK;

// Use the template overload
hr = GetElementInterface(L"Button1", &spBtn);

// Use the nontemplate overload
hr = GetElementInterface(L"Button1", IID_IHTMLButtonElement, reinterpret_cast<void**>(&spBt
n));
```

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
CDHtmlDialog::GetElement
CDHtmlDialog::GetControlDispatch

Other Resources

CDHtmlDialog Members

CDHtmlDialog::GetElementProperty

Retrieves the value of the property identified by *dispid* from the HTML element identified by *szElementId*.

```
VARIANT GetElementProperty(
   LPCTSTR szElementId,
   DISPID dispid
);
```

Parameters

szElementId

The ID of an HTML element.

dispid

The dispatch ID of a property.

Return Value

The value of the property or an empty variant if the property or element could not be found.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
CDHtmlDialog::GetElementHtml
CDHtmlDialog::GetElementText

Other Resources

CDHtmlDialog Members

CDHtmlDialog::GetElementText

Retrieves the **innerText** property of the HTML element identified by *szElementId*.

```
BSTR GetElementText(
   LPCTSTR szElementId
);
```

Parameters

szElementId

The ID of an HTML element.

Return Value

The **innerText** property of the HTML element identified by *szElementId* or **NULL** if the property or element could not be found.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
CDHtmlDialog::GetElementHtml
CDHtmlDialog::GetElementProperty

Other Resources

CDHtmlDialog Members

CDHtmlDialog::GetEvent

Returns the **IHTMLEventObj** pointer to the current event object.

```
HRESULT GetEvent(
   IHTMLEventObj **ppEventObj
);
```

Parameters

ppEventObj

Address of a pointer that will be filled with the **IHTMLEventObj** interface pointer.

Return Value

A standard **HRESULT** value.

Remarks

This function should only be called from within a DHTML event handler.

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

Other Resources

CDHtmlDialog Members

CDHtmlDialog::GetExternal

Gets the host's **IDispatch** interface.

```
STDMETHOD(GetExternal)(
   IDispatch **ppDispatch
);
```

Parameters

ppDispatch

See *ppDispatch* in IDocHostUIHandler::GetExternal in the *Platform SDK*.

Return Value

Returns **S_OK** on success or **E_NOTIMPL** on failure.

Remarks

This member function is CDHtmlDialog's implementation of IDocHostUIHandler::GetExternal, as described in the *Platform SDK*.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
IDocHostUIhandler Interface
CDHtmlDialog::CanAccessExternal
CDHtmlDialog::SetExternalDispatch

Other Resources

CDHtmlDialog Members

CDHtmlDialog::GetHostInfo

Retrieves the host's UI capabilities.

```
STDMETHOD(GetHostInfo)(
   DOCHOSTUIINFO *pInfo
);
```

Parameters

pInfo

See *pInfo* in IDocHostUIHandler::GetHostInfo in the *Platform SDK*.

Return Value

Returns **S_OK**.

Remarks

This member function is CDHtmlDialog's implementation of IDocHostUIHandler::GetHostInfo, as described in the *Platform SDK*.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
IDocHostUIhandler Interface
CDHtmlDialog::SetHostFlags

Other Resources

CDHtmlDialog Members

CDHtmlDialog::GetOptionKeyPath

Retrieves the registry key under which user preferences are stored.

```
STDMETHOD(GetOptionKeyPath)(
   LPOLESTR * pchKey,
   DWORD dw
);
```

Parameters

pchKey

See *pchKey* in IDocHostUIHandler::GetOptionKeyPath in the *Platform SDK*.

dw

See *dw* in **IDocHostUIHandler::GetOptionKeyPath** in the *Platform SDK*.

Return Value

Returns **E_NOTIMPL**.

Remarks

This member function is CDHtmlDialog's implementation of IDocHostUIHandler::GetOptionKeyPath, as described in the *Platform SDK*.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
IDocHostUIhandler Interface

Other Resources

CDHtmlDialog Members

CDHtmlDialog::HideUI

Hides the host's UI.

```
STDMETHOD(HideUI)( void );
```

Return Value

Returns **E_NOTIMPL**.

Remarks

This member function is CDHtmlDialog's implementation of IDocHostUIHandler::HideUI, as described in the *Platform SDK*.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
IDocHostUIhandler Interface
CDHtmlDialog::ShowUI

Other Resources

CDHtmlDialog Members

CDHtmlDialog::IsExternalDispatchSafe

Indicates whether the host's **IDispatch** interface is safe for scripting.

```
virtual BOOL IsExternalDispatchSafe( );
```

Return Value

Returns **FALSE**.

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

CDHtmlDialog::CanAccessExternal

Other Resources

CDHtmlDialog Members

CDHtmlDialog::LoadFromResource

Loads the specified resource into the WebBrowser control in the DHTML dialog.

```
BOOL LoadFromResource(
   LPCTSTR lpszResource
);
BOOL LoadFromResource(
   UINT nRes
);
```

Parameters

lpszResource

A pointer to a string containing the name of the resource to load.

nRes

The ID of the resource to load.

Return Value

TRUE if successful; otherwise **FALSE**.

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

Other Resources

CDHtmlDialog Members

CDHtmlDialog::Navigate

Navigates to the resource identified by the URL that is specified by *lpszURL*.

```
void Navigate(
   LPCTSTR lpszURL,
   DWORD dwFlags = 0,
   LPCTSTR lpszTargetFrameName = NULL,
   LPCTSTR lpszHeaders = NULL,
   LPVOID lpvPostData = NULL,
   DWORD dwPostDataLen = 0
);
```

Parameters

lpszURL

A pointer to a string containing the URL to be targeted.

dwFlags

The flags of a variable that specifies whether to add the resource to the history list, whether to read to the cache or write from the cache, and whether to display the resource in a new window. The variable can be a combination of the values defined by the BrowserNavConstants enumeration.

lpszTargetFrameName

A pointer to a string that contains the name of the frame in which to display the resource.

lpszHeaders

A pointer to a value that specifies the HTTP headers to send to the server. These headers are added to the default Internet Explorer headers. The headers can specify such information as the action required of the server, the type of data being passed to the server, or a status code. This parameter is ignored if the URL is not an HTTP URL.

lpvPostData

A pointer to the data to send with the HTTP POST transaction. For example, the POST transaction is used to send data gathered by an HTML form. If this parameter does not specify any post data, **Navigate** issues an HTTP GET transaction. This parameter is ignored if the URL is not an HTTP URL.

dwPostDataLen

Data to send with the HTTP POST transaction. For example, the POST transaction is used to send data gathered by an HTML form. If this parameter does not specify any post data, **Navigate** issues an HTTP GET transaction. This parameter is ignored if URL is not an HTTP URL.

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

Other Resources

CDHtmlDialog Members

CDHtmlDialog::OnBeforeNavigate

Called by the framework to cause an event to fire before a navigation occurs.

```
virtual void OnBeforeNavigate(
   LPDISPATCH pDisp,
   LPCTSTR szUrl
);
```

Parameters

pDisp

A pointer to an **IDispatch** object.

szUrl

A pointer to a string containing the URL to navigate to.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
CDHtmlDialog::OnNavigateComplete

Other Resources

CDHtmlDialog Members

CDHtmlDialog::OnDocumentComplete

Called by the framework to notify an application when a document has achieved the **READYSTATE_COMPLETE** state.

```
virtual void OnDocumentComplete(
   LPDISPATCH pDisp,
   LPCTSTR szUrl
);
```

Parameters

pDisp

A pointer to an **IDispatch** object.

szUrl

A pointer to a string containing the URL that was navigated to.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
CDHtmlDialog::OnNavigateComplete

Other Resources

CDHtmlDialog Members

CDHtmlDialog::OnDocWindowActivate

Called by the framework when the document window is activated or deactivated.

```
STDMETHOD(OnDocWindowActivate)(
   BOOL fActivate
);
```

Parameters
fActivate

See *fActivate* in IDocHostUIHandler::OnDocWindowActivate in the *Platform SDK*.

Return Value

Returns **E_NOTIMPL**.

Remarks

This member function is CDHtmlDialog's implemention of IDocHostUIHandler::OnDocWindowActivate, as described in the *Platform SDK*.

See Also
Reference
CDHtmlDialog Class
Hierarchy Chart
IDocHostUIhandler Interface
Other Resources
CDHtmlDialog Members

CDHtmlDialog::OnFrameWindowActivate

Called by the framework when the frame window is activated or deactivated.

```
STDMETHOD(OnFrameWindowActivate)(
   BOOL fActivate
);
```

Parameters

fActivate

See *fActivate* in IDocHostUIHandler::OnFrameWindowActivate in the *Platform SDK*.

Return Value

Returns **E_NOTIMPL**.

Remarks

This member function is CDHtmlDialog's implementation of IDocHostUIHandler::OnFrameWindowActivate, as described in the *Platform SDK*.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
IDocHostUIhandler Interface

Other Resources

CDHtmlDialog Members

CDHtmlDialog::OnInitDialog

Called in response to the **WM_INITDIALOG** message.

```
virtual BOOL OnInitDialog( );
```

Return Value

The default implementation returns **TRUE**.

Remarks

This message is sent to the dialog box during the **Create**, **CreateIndirect**, or **DoModal** calls, which occur immediately before the dialog box is displayed.

Override this member function if you need to perform special processing when the dialog box is initialized. In the overridden version, first call the base class **OnInitDialog** but disregard its return value. You will normally return **TRUE** from your overridden member function.

Windows calls the **OnInitDialog** function through the standard global dialog-box procedure common to all Microsoft Foundation Class Library dialog boxes, rather than through your message map, so you do not need a message-map entry for this member function.

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

Other Resources

CDHtmlDialog Members

CDHtmlDialog::OnNavigateComplete

Called by the framework after navigation to the specified URL is completed.

```
virtual void OnNavigateComplete(
   LPDISPATCH pDisp,
   LPCTSTR szUrl
);
```

Parameters

pDisp

A pointer to an **IDispatch** object.

szUrl

A pointer to a string containing the URL that was navigated to.

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

CDHtmlDialog::OnDocumentComplete

Other Resources

CDHtmlDialog Members

CDHtmlDialog::ResizeBorder

Alerts the object that it needs to resize its border space.

```
STDMETHOD(ResizeBorder)(
   LPCRECT prcBorder,
   IOleInPlaceUIWindow * pUIWindow,
   BOOL fRameWindow
);
```

Parameters

prcBorder

See *prcBorder* in IDocHostUIHandler::ResizeBorder in the *Platform SDK*.

pUIWindow

See *pUIWindow* in **IDocHostUIHandler::ResizeBorder** in the *Platform SDK*.

fFrameWindow

See *fFrameWindow* in **IDocHostUIHandler::ResizeBorder** in the *Platform SDK*.

Return Value

Returns **E_NOTIMPL**.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
IDocHostUIhandler Interface

Other Resources

CDHtmlDialog Members

CDHtmlDialog::SetControlProperty

Sets the property of an ActiveX control to a new value.

```
void SetControlProperty(
   LPCTSTR szElementId,
   DISPID dispid,
   VARIANT *pVar
);
void SetControlProperty(
   IDispatch *pdispControl,
   DISPID dispid,
   VARIANT *pVar
);
void SetControlProperty(
   LPCTSTR szElementId,
   LPCTSTR szPropName,
   VARIANT *pVar
);
```

Parameters

szElementId

The HTML ID of an ActiveX control.

dispid

The dispatch ID of the property to set.

pVar

Pointer to a **VARIANT** containing the new property value.

pdispControl

Pointer to an ActiveX control's **IDispatch** interface.

szPropName

String containing the name of the property to set.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
CDHtmlDialog::SetElementProperty

Other Resources

CDHtmlDialog Members

CDHtmlDialog::SetElementHtml

Sets the **innerHTML** property of an HTML element.

```
void SetElementHtml(
   LPCTSTR szElementId,
   BSTR bstrText
);
void SetElementHtml(
   IUnknown *punkElem,
   BSTR bstrText
);
```

Parameters

szElementId

The ID of an HTML element.

bstrText

The new value of the **innerHTML** property.

punkElem

The **IUnknown** pointer of an HTML element.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
CDHtmlDialog::SetElementText
CDHtmlDialog::SetElementProperty

Other Resources

CDHtmlDialog Members

CDHtmlDialog::SetElementProperty

Sets a property of an HTML element.

```
void SetElementProperty(
   LPCTSTR szElementId,
   DISPID dispid,
   VARIANT *pVar
);
```

Parameters

szElementId

The ID of an HTML element.

dispid

The dispatch ID of the property to set.

pVar

The new value of the property.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
CDHtmlDialog::SetControlProperty
CDHtmlDialog::SetElementText
CDHtmlDialog::SetElementHtml

Other Resources

CDHtmlDialog Members

CDHtmlDialog::SetElementText

Sets the **innerText** property of an HTML element.

```
void SetElementText(
   LPCTSTR szElementId,
   BSTR bstrText
);
void SetElementText(
   IUnknown *punkElem,
   BSTR bstrText
);
```

Parameters

szElementId

The ID of an HTML element.

bstrText

The new value of the **innerText** property.

punkElem

The **IUnknown** pointer of an HTML element.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
CDHtmlDialog::SetElementHtml
CDHtmlDialog::SetElementProperty

Other Resources

CDHtmlDialog Members

CDHtmlDialog::SetExternalDispatch

Sets the host's **IDispatch** interface.

```
void SetExternalDispatch(
   IDispatch * pdispExternal
);
```

Parameters

pdispExternal

The new **IDispatch** interface.

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

CDHtmlDialog::GetExternal

Other Resources

CDHtmlDialog Members

CDHtmlDialog::SetHostFlags

Sets the host UI flags.

```
void SetHostFlags(
   DWORD dwFlags
);
```

Parameters

dwFlags

For possible values, see DOCHOSTUIFLAG in the *Platform SDK*.

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

CDHtmlDialog::GetHostInfo

Other Resources

CDHtmlDialog Members

CDHtmlDialog::ShowContextMenu

Called when a context menu is about to be displayed.

```
STDMETHOD(ShowContextMenu)(
   DWORD dwID,
   POINT * ppt,
   IUnknown * pcmdtReserved,
   IDispatch * pdispReserved
);
```

Parameters

dwID

See *dwID* in IDocHostUIHandler::ShowContextMenu in the *Platform SDK*.

ppt

See *ppt* in **IDocHostUIHandler::ShowContextMenu** in the *Platform SDK*.

pcmdtReserved

See *pcmdtReserved* in **IDocHostUIHandler::ShowContextMenu** in the *Platform SDK*.

pdispReserved

See *pdispReserved* in **IDocHostUIHandler::ShowContextMenu** in the *Platform SDK*.

Return Value

Returns **S_FALSE**.

Remarks

This member function is CDHtmlDialog's implementation of IDocHostUIHandler::ShowContextMenu, as described in the *Platform SDK*.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
IDocHostUIhandler Interface

Other Resources

CDHtmlDialog Members

CDHtmlDialog::ShowUI

Shows the host's UI.

```
STDMETHOD(ShowUI)(
   DWORD dwID,
   IOleInPlaceActiveObject * pActiveObject,
   IOleCommandTarget * pCommandTarget,
   IOleInPlaceFrame * pFrame,
   IOleInPlaceUIWindow * pDoc
);
```

Parameters

dwID

See *dwID* in IDocHostUIHandler::ShowUI in the *Platform SDK*.

pActiveObject

See *d pActiveObject* in **IDocHostUIHandler::ShowUI** in the *Platform SDK*.

pCommandTarget

See *pCommandTarget* in **IDocHostUIHandler::ShowUI** in the *Platform SDK*.

pFrame

See *pFrame* in **IDocHostUIHandler::ShowUI** in the *Platform SDK*.

pDoc

See *pDoc* in **IDocHostUIHandler::ShowUI** in the *Platform SDK*.

Return Value

Returns **S_FALSE**.

Remarks

This member function is CDHtmlDialog's implementation of IDocHostUIHandler::ShowUI, as described in the *Platform SDK*.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
IDocHostUIhandler Interface
CDHtmlDialog::HideUI

Other Resources

CDHtmlDialog Members

CDHtmlDialog::TranslateAccelerator

Called to process menu accelerator-key messages.

```
STDMETHOD(TranslateAccelerator)(
   LPMSG lpMsg,
   const GUID * pguidCmdGroup,
   DWORD nCmdID
);
```

Parameters

lpMsg

See *lpMsg* in IDocHostUIHandler::TranslateAccelerator in the *Platform SDK*.

pguidCmdGroup

See *pguidCmdGroup* in **IDocHostUIHandler::TranslateAccelerator** in the *Platform SDK*.

nCmdID

See *nCmdID* in **IDocHostUIHandler::TranslateAccelerator** in the *Platform SDK*.

Return Value

Returns **S_FALSE**.

Remarks

This member function is CDHtmlDialog's implementation of IDocHostUIHandler::TranslateAccelerator, as described in the *Platform SDK*.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
IDocHostUIhandler Interface

Other Resources

CDHtmlDialog Members

CDHtmlDialog::TranslateUrl

Called to modify the URL to be loaded.

```
STDMETHOD(TranslateUrl)(
   DWORD dwTranslate,
   OLECHAR * pchURLIn,
   OLECHAR ** ppchURLOut
);
```

Parameters

dwTranslate

See *dwTranslate* in IDocHostUIHandler::TranslateUrl in the *Platform SDK*.

pchURLIn

See *pchURLIn* in **IDocHostUIHandler::TranslateUrl** in the *Platform SDK*.

ppchURLOut

See *ppchURLOut* in **IDocHostUIHandler::TranslateUrl** in the *Platform SDK*.

Return Value

Returns **S_FALSE**.

Remarks

This member function is CDHtmlDialog's implementation of IDocHostUIHandler::TranslateUrl, as described in the *Platform SDK*.

See Also

Reference

CDHtmlDialog Class
Hierarchy Chart
IDocHostUIhandler Interface

Other Resources

CDHtmlDialog Members

CDHtmlDialog::UpdateUI

Called to notify the host that the command state has changed.

```
STDMETHOD(UpdateUI)( void );
```

Return Value

Returns **E_NOTIMPL**.

Remarks

This member function is CDHtmlDialog's implementation of IDocHostUIHandler::UpdateUI, as described in the *Platform SDK*.

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

IDocHostUIhandler Interface

Other Resources

CDHtmlDialog Members

CDHtmlDialog Data Members

For information about the data members in CDHtmlDialog, see CDHtmlDialog Members.

CDHtmlDialog::m_bUseHtmlTitle

Indicates whether to use the HTML document's title as the dialog caption.

```
BOOL m_bUseHtmlTitle;
```

Remarks

If **m_bUseHtmlTitle** is **true**, the dialog caption is set equal to the title of the HTML document; otherwise, the caption in the dialog resource is used.

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

Other Resources

CDHtmlDialog Members

CDHtmlDialog::m_nHtmlResID

Resource ID of HTML resource to be displayed.

```
UINT m_nHtmlResID;
```

Example

```
CDHtmlDialog mydialog( IDD_DIALOG1 );
mydialog.m_nHtmlResID = IDR_HTML1;
mydialog.DoModal( );
```

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

Other Resources

CDHtmlDialog Members

CDHtmlDialog::m_pBrowserApp

A pointer to a Web browser application.

```
CComPtr <IWebBrowser2> m_pBrowserApp;
```

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

IWebBrowser2

Other Resources

CDHtmlDialog Members

CDHtmlDialog::m_spHtmlDoc

A pointer to an HTML document.

```
CComPtr<IHTMLDocument2> m_spHtmlDoc;
```

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

IHTMLDocument2

Other Resources

CDHtmlDialog Members

CDHtmlDialog::m_strCurrentUrl

The current URL.

```
CString m_strCurrentUrl;
```

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

Other Resources

CDHtmlDialog Members

CDHtmlDialog::m_szHtmlResID

String version of the HTML resource ID.

```
LPTSTR m_szHtmlResID;
```

Example

```
CDHtmlDialog mydialog( IDD_DIALOG1 );
TCHAR szResID[] = _T( "HTML_PAGE" );
mydialog.m_szHtmlResID = szResID;
mydialog.DoModal( );
```

See Also

Reference

CDHtmlDialog Class

Hierarchy Chart

Other Resources

CDHtmlDialog Members

DDX_DHtml Helper Macros

The DDX_DHtml helper macros allow easy access to the commonly used properties of controls on an HTML page.

Data Exchange Macros

DDX_DHtml_ElementValue	Sets or retrieves the Value property from the selected control.
DDX_DHtml_ElementInnerText	Sets or retrieves the text between the start and end tags of the current element.
DDX_DHtml_ElementInnerHtml	Sets or retrieves the HTML between the start and end tags of the current element.
DDX_DHtml_Anchor_Href	Sets or retrieves the destination URL or anchor point.
DDX_DHtml_Anchor_Target	Sets or retrieves the target window or frame.
DDX_DHtml_Img_Src	Sets or retrieves the name of an image or a video clip in the document.
DDX_DHtml_Frame_Src	Sets or retrieves the URL of the associated frame.
DDX_DHtml_IFrame_Src	Sets or retrieves the URL of the associated frame.

Requirements

Header: afxdhtml.h

See Also

Reference

CDHtmlDialog Class

DDX_DHtml_Anchor_Href

Sets or retrieves the destination URL or anchor point.

```
DDX_DHtml_Anchor_Href(
   CDataExchange* dx,
   LPCTSTR name,
   CString& var
)
```

Parameters

dx

A pointer to a CDataExchange object.

name

The value that you specified for the HTML control's ID parameter.

var

The value being exchanged.

Remarks

This macro calls the CDHtmlDialog::DDX_DHtml_ElementText function using the DISPID_IHTMLANCHORELEMENT_HREF dispatch ID.

See Also

Reference

DDX_DHtml Helper Macros

href Property (IHTMLAnchorElement)

DDX_DHtml_Anchor_Target

Sets or retrieves the target window or frame.

```
DDX_DHtml_Anchor_Target(
   CDataExchange* dx,
   LPCTSTR name,
   CString& var
)
```

Parameters

dx

A pointer to a CDataExchange object.

name

The value that you specified for the HTML control's ID parameter.

var

The value being exchanged.

Remarks

This macro calls the CDHtmlDialog::DDX_DHtml_ElementText function using the DISPID_IHTMLANCHORELEMENT_TARGET dispatch ID.

See Also

Reference

DDX_DHtml Helper Macros

target Property (IHTMLAnchorElement)

DDX_DHtml_ElementInnerHtml

Sets or retrieves the HTML between the start and end tags of the current element.

```
DDX_DHtml_ElementInnerHtml(
   CDataExchange* dx,
   LPCTSTR name,
   CString& var
)
```

Parameters

dx

A pointer to a CDataExchange object.

name

The value that you specified for the HTML control's ID parameter.

var

The value being exchanged.

Remarks

This macro calls the CDHtmlDialog::DDX_DHtml_ElementText function using the DISPID_IHTMLELEMENT_INNERHTML dispatch ID.

See Also

Reference

DDX_DHtml Helper Macros

innerHTML Property (IHTMLElement)

DDX_DHtml_ElementInnerText

Sets or retrieves the text between the start and end tags of the current element.

```
DDX_DHtml_ElementInnerText(
   CDataExchange* dx,
   LPCTSTR name,
   CString& var
)
```

Parameters

dx

A pointer to a CDataExchange object.

name

The value that you specified for the HTML control's ID parameter.

var

The value being exchanged.

Remarks

This macro calls the CDHtmlDialog::DDX_DHtml_ElementText function using the DISPID_IHTMLELEMENT_INNERTEXT dispatch ID.

See Also

Reference

DDX_DHtml Helper Macros

innerText Property (IHTMLElement)

DDX_DHtml_ElementValue

Sets or retrieves the Value property from the selected control.

```
DDX_DHtml_ElementValue(
   CDataExchange* dx,
   LPCTSTR name,
      var
)
```

Parameters

dx

A pointer to a CDataExchange object.

name

The value that you specified for the HTML control's ID parameter.

var

The value being exchanged. See *value* in CDHtmlDialog::DDX_DHtml_ElementText.

Remarks

This macro will only succeed when run on controls that have a Value property. Controls that have a Value property include edit boxes, list boxes, and combo boxes.

This macro calls the CDHtmlDialog::DDX_DHtml_ElementText function using the DISPID_A_VALUE dispatch ID.

See Also

Reference

DDX_DHtml Helper Macros

DDX_DHtml_Frame_Src

Sets or retrieves the URL of the associated frame.

```
DDX_DHtml_Frame_Src(
   CDataExchange* dx,
   LPCTSTR name,
   CString& var
)
```

Parameters

dx

A pointer to a CDataExchange object.

name

The value that you specified for the HTML control's ID parameter.

var

The value being exchanged.

Remarks

This macro calls the CDHtmlDialog::DDX_DHtml_ElementText function using the DISPID_IHTMLFRAMEBASE_SRC dispatch ID.

See Also

Reference

DDX_DHtml Helper Macros

src Property (IHTMLFrameBase)

DDX_DHtml_IFrame_Src

Sets or retrieves the URL of the associated frame.

```
DDX_DHtml_IFrame_Src(
   CDataExchange* dx,
   LPCTSTR name,
   CString& var
)
```

Parameters

dx

A pointer to a CDataExchange object.

name

The value that you specified for the HTML control's ID parameter.

var

The value being exchanged.

Remarks

This macro calls the CDHtmlDialog::DDX_DHtml_ElementText function using the DISPID_IHTMLFRAMEBASE_SRC dispatch ID.

See Also

Reference

DDX_DHtml Helper Macros

src Property (IHTMLFrameBase)

DDX_DHtml_Img_Src

Gets or retrieves the name of an image or a video clip in the document.

```
DDX_DHtml_Img_Src(
   CDataExchange* dx,
   LPCTSTR name,
   CString& var
)
```

Parameters

dx

A pointer to a CDataExchange object.

name

The value that you specified for the HTML control's ID parameter.

var

The value being exchanged.

Remarks

When using the **DDX_DHtml_Img_Src** macro to retrieve the src property for an IMAGE element, the Internet Explorer image object will return the fully escaped URL for the image source. For example, if you use the **DDX_DHtml_Img_Src** macro to set the src property of an IMAGE element to the string "some interesting picture," when you retrieve that property, Internet Explorer will return the string "res://d:\myapplication\myapp.exe/some%20interesting%20picture."

This macro calls the CDHtmlDialog::DDX_DHtml_ElementText function using the DISPID_IHTMLIMGELEMENT_SRC dispatch ID.

See Also

Reference

DDX_DHtml Helper Macros

src Property (IHTMLFrameBase)

CDialog Class

The base class used for displaying dialog boxes on the screen.

```
class CDialog : public CWnd
```

Remarks

Dialog boxes are of two types: modal and modeless. A modal dialog box must be closed by the user before the application continues. A modeless dialog box allows the user to display the dialog box and return to another task without canceling or removing the dialog box.

A **CDialog** object is a combination of a dialog template and a **CDialog**-derived class. Use the dialog editor to create the dialog template and store it in a resource, then use the Add Class wizard to create a class derived from **CDialog**.

A dialog box, like any other window, receives messages from Windows. In a dialog box, you are particularly interested in handling notification messages from the dialog box's controls since that is how the user interacts with your dialog box. Use the Properties window to select which messages you wish to handle and it will add the appropriate message-map entries and message-handler member functions to the class for you. You only need to write application-specific code in the handler member functions.

If you prefer, you can always write message-map entries and member functions manually.

In all but the most trivial dialog box, you add member variables to your derived dialog class to store data entered in the dialog box's controls by the user or to display data for the user. You can use the Add Variable wizard to create member variables and associate them with controls. At the same time, you choose a variable type and permissible range of values for each variable. The code wizard adds the member variables to your derived dialog class.

A data map is generated to automatically handle the exchange of data between the member variables and the dialog box's controls. The data map provides functions that initialize the controls in the dialog box with the proper values, retrieve the data, and validate the data.

To create a modal dialog box, construct an object on the stack using the constructor for your derived dialog class and then call **DoModal** to create the dialog window and its controls. If you wish to create a modeless dialog, call **Create** in the constructor of your dialog class.

You can also create a template in memory by using a DLGTEMPLATE data structure as described in the *Platform SDK*. After you construct a **CDialog** object, call CreateIndirect to create a modeless dialog box, or call InitModalIndirect and DoModal to create a modal dialog box.

The exchange and validation data map is written in an override of **CWnd::DoDataExchange** that is added to your new dialog class. See the DoDataExchange member function in **CWnd** for more on the exchange and validation functionality.

Both the programmer and the framework call **DoDataExchange** indirectly through a call to CWnd::UpdateData.

The framework calls **UpdateData** when the user clicks the OK button to close a modal dialog box. (The data is not retrieved if the Cancel button is clicked.) The default implementation of OnInitDialog also calls **UpdateData** to set the initial values of the controls. You typically override **OnInitDialog** to further initialize controls. **OnInitDialog** is called after all the dialog controls are created and just before the dialog box is displayed.

You can call **CWnd::UpdateData** at any time during the execution of a modal or modeless dialog box.

If you develop a dialog box by hand, you add the necessary member variables to the derived dialog-box class yourself, and you add member functions to set or get these values.

A modal dialog box closes automatically when the user presses the OK or Cancel buttons or when your code calls the **EndDialog** member function.

When you implement a modeless dialog box, always override the **OnCancel** member function and call **DestroyWindow** from within it. Don't call the base class **CDialog::OnCancel**, because it calls **EndDialog**, which will make the dialog box invisible but will not destroy it. You should also override **PostNcDestroy** for modeless dialog boxes in order to delete **this**, since modeless dialog boxes are usually allocated with **new**. Modal dialog boxes are usually constructed on the frame and do not need **PostNcDestroy** cleanup.

For more information on **CDialog**, see:

- Dialog Boxes
- Knowledge Base article Q262954 : HOWTO: Create a Resizeable Dialog Box with Scroll Bars

Requirements

Header: afxwin.h

See Also

Tasks

DLGCBR32 Sample: Demonstrates Adding a Status Bar and Toolbar to Dialog Boxes

DLGTEMPL Sample: Creates Dialog Templates Dynamically

Reference

CWnd Class

Hierarchy Chart

Other Resources

CDialog Members

CDialog Members

Base Class Members

COBJECT Members

CCmdTarget Members

CWnd Members

Construction

CDialog	Constructs a **CDialog** object.

Initialization

Create	Initializes the **CDialog** object. Creates a modeless dialog box and attaches it to the **CDialog** object.
CreateIndirect	Creates a modeless dialog box from a dialog-box template in memory (not resource-based).
InitModalIndirect	Creates a modal dialog box from a dialog-box template in memory (not resource-based). The parameters are stored until the function **DoModal** is called.

Operations

DoModal	Calls a modal dialog box and returns when done.
EndDialog	Closes a modal dialog box.
GetDefID	Gets the ID of the default pushbutton control for a dialog box.
GotoDlgCtrl	Moves the focus to a specified dialog-box control in the dialog box.
MapDialogRect	Converts the dialog-box units of a rectangle to screen units.
NextDlgCtrl	Moves the focus to the next dialog-box control in the dialog box.
PrevDlgCtrl	Moves the focus to the previous dialog-box control in the dialog box.
SetDefID	Changes the default pushbutton control for a dialog box to a specified pushbutton.
SetHelpID	Sets a context-sensitive help ID for the dialog box.

Overridables

OnCancel	Override to perform the Cancel button or ESC key action. The default closes the dialog box and **DoModal** returns **IDCANCEL**.
OnInitDialog	Override to augment dialog-box initialization.
OnOK	Override to perform the OK button action in a modal dialog box. The default closes the dialog box and **DoModal** returns **IDOK**.
OnSetFont	Override to specify the font that a dialog-box control is to use when it draws text.

See Also

Reference

CDialog Class

Hierarchy Chart

CDialog Member Functions

For information about the member functions in **CDialog**, see CDialog Members.

CDialog::CDialog

To construct a resource-based modal dialog box, call either public form of the constructor.

```
explicit CDialog(
   LPCTSTR lpszTemplateName,
   CWnd* pParentWnd = NULL
);
explicit CDialog(
   UINT nIDTemplate,
   CWnd* pParentWnd = NULL
);
CDialog( );
```

Parameters

lpszTemplateName

Contains a null-terminated string that is the name of a dialog-box template resource.

nIDTemplate

Contains the ID number of a dialog-box template resource.

pParentWnd

Points to the parent or owner window object (of type CWnd) to which the dialog object belongs. If it is **NULL**, the dialog object's parent window is set to the main application window.

Remarks

One form of the constructor provides access to the dialog resource by template name. The other constructor provides access by template ID number, usually with an **IDD_** prefix (for example, IDD_DIALOG1).

To construct a modal dialog box from a template in memory, first invoke the parameterless, protected constructor and then call **InitModalIndirect**.

After you construct a modal dialog box with one of the above methods, call **DoModal**.

To construct a modeless dialog box, use the protected form of the **CDialog** constructor. The constructor is protected because you must derive your own dialog-box class to implement a modeless dialog box. Construction of a modeless dialog box is a two-step process. First call the constructor; then call the **Create** member function to create a resource-based dialog box, or call **CreateIndirect** to create the dialog box from a template in memory.

See Also

Reference

CDialog Class
Hierarchy Chart
CDialog::Create
CWnd::DestroyWindow
CDialog::InitModalIndirect
CDialog::DoModal
CreateDialog

Other Resources

CDialog Members

CDialog::Create

Call **Create** to create a modeless dialog box using a dialog-box template from a resource.

```
virtual BOOL Create(
   LPCTSTR lpszTemplateName,
   CWnd* pParentWnd = NULL
);
virtual BOOL Create(
   UINT nIDTemplate,
   CWnd* pParentWnd = NULL
);
```

Parameters

lpszTemplateName

Contains a null-terminated string that is the name of a dialog-box template resource.

pParentWnd

Points to the parent window object (of type CWnd) to which the dialog object belongs. If it is **NULL**, the dialog object's parent window is set to the main application window.

nIDTemplate

Contains the ID number of a dialog-box template resource.

Return Value

Both forms return nonzero if dialog-box creation and initialization were successful; otherwise 0.

Remarks

You can put the call to **Create** inside the constructor or call it after the constructor is invoked.

Two forms of the **Create** member function are provided for access to the dialog-box template resource by either template name or template ID number (for example, IDD_DIALOG1).

For either form, pass a pointer to the parent window object. If *pParentWnd* is **NULL**, the dialog box will be created with its parent or owner window set to the main application window.

The **Create** member function returns immediately after it creates the dialog box.

Use the **WS_VISIBLE** style in the dialog-box template if the dialog box should appear when the parent window is created. Otherwise, you must call **ShowWindow**. For further dialog-box styles and their application, see the DLGTEMPLATE structure in the *Platform SDK* and Window Styles in the *MFC Reference*.

Use the **CWnd::DestroyWindow** function to destroy a dialog box created by the **Create** function.

Example

```
CMyDialog* pDialog;

void CMyWnd::OnSomeAction()
{
   //pDialog initialized to NULL in the constructor of CMyWnd class
   pDialog = new CMyDialog();
   //Check if new succeeded and we got a valid pointer to a dialog object
   if(pDialog != NULL)
   {
      BOOL ret = pDialog->Create(IDD_MYDIALOG,this);
      if(!ret)   //Create failed.
         AfxMessageBox("Error creating Dialog");
      pDialog->ShowWindow(SW_SHOW);
   }
   else
      AfxMessageBox("Error Creating Dialog Object");
```

```
}
```

See Also

Reference

CDialog Class
Hierarchy Chart
CDialog::CDialog
CWnd::DestroyWindow
CDialog::InitModalIndirect
CDialog::DoModal
CreateDialog

Other Resources

CDialog Members

CDialog::CreateIndirect

Call this member function to create a modeless dialog box from a dialog-box template in memory.

```
virtual BOOL CreateIndirect(
   LPCDLGTEMPLATE lpDialogTemplate,
   CWnd* pParentWnd = NULL,
   void* lpDialogInit = NULL
);
virtual BOOL CreateIndirect(
   HGLOBAL hDialogTemplate,
   CWnd* pParentWnd = NULL
);
```

Parameters

lpDialogTemplate

Points to memory that contains a dialog-box template used to create the dialog box. This template is in the form of a DLGTEMPLATE structure and control information, as described in the *Platform SDK*.

pParentWnd

Points to the dialog object's parent window object (of type CWnd). If it is **NULL**, the dialog object's parent window is set to the main application window.

lpDialogInit

Points to a **DLGINIT** resource.

hDialogTemplate

Contains a handle to global memory containing a dialog-box template. This template is in the form of a **DLGTEMPLATE** structure and data for each control in the dialog box.

Return Value

Nonzero if the dialog box was created and initialized successfully; otherwise 0.

Remarks

The **CreateIndirect** member function returns immediately after it creates the dialog box.

Use the **WS_VISIBLE** style in the dialog-box template if the dialog box should appear when the parent window is created. Otherwise, you must call **ShowWindow** to cause it to appear. For more information on how you can specify other dialog-box styles in the template, see the DLGTEMPLATE structure in the *Platform SDK*.

Use the **CWnd::DestroyWindow** function to destroy a dialog box created by the **CreateIndirect** function.

Dialog boxes that contain ActiveX controls require additional information provided in a **DLGINIT** resource. For more information, see Knowledge Base article Q231591, " HOWTO: Use a Dialog Template to Create a MFC Dialog with an ActiveX Control." Knowledge Base articles are available in the MSDN Library Visual Studio documentation or at http://support.microsoft.com.

See Also

Reference

CDialog Class
Hierarchy Chart
CDialog::CDialog
CWnd::DestroyWindow
CDialog::Create
CreateDialogIndirect

Other Resources

CDialog Members

CDialog::DoModal

Call this member function to invoke the modal dialog box and return the dialog-box result when done.

```
virtual INT_PTR DoModal( );
```

Return Value

An **int** value that specifies the value of the *nResult* parameter that was passed to the CDialog::EndDialog member function, which is used to close the dialog box. The return value is –1 if the function could not create the dialog box, or **IDABORT** if some other error occurred, in which case the Output window will contain error information from GetLastError.

Remarks

This member function handles all interaction with the user while the dialog box is active. This is what makes the dialog box modal; that is, the user cannot interact with other windows until the dialog box is closed.

If the user clicks one of the pushbuttons in the dialog box, such as OK or Cancel, a message-handler member function, such as OnOK or OnCancel, is called to attempt to close the dialog box. The default **OnOK** member function will validate and update the dialog-box data and close the dialog box with result **IDOK**, and the default **OnCancel** member function will close the dialog box with result **IDCANCEL** without validating or updating the dialog-box data. You can override these message-handler functions to alter their behavior.

Note

PreTranslateMessage is now called for modal dialog box message processing.

Example

```
void CTstApp::OnAppAbout()
{
   // Construct the dialog box passing the
   // ID of the dialog template resource
   CDialog aboutDlg(IDD_ABOUTBOX);

   // Create and show the dialog box
   INT_PTR nRet = -1;
   nRet = aboutDlg.DoModal();

   // Handle the return value from DoModal
   switch ( nRet )
   {
   case -1:
      AfxMessageBox("Dialog box could not be created!");
      break;
   case IDABORT:
      // Do something
      break;
   case IDOK:
      // Do something
      break;
   case IDCANCEL:
      // Do something
      break;
   default:
      // Do something
      break;
   };
}
```

See Also

Reference

CDialog Class

Hierarchy Chart
DialogBox
CWnd::IsDialogMessage
Other Resources
CDialog Members

CDialog::EndDialog

Call this member function to terminate a modal dialog box.

```
void EndDialog(
   int nResult
);
```

Parameters

nResult

Contains the value to be returned from the dialog box to the caller of **DoModal**.

Remarks

This member function returns *nResult* as the return value of **DoModal**. You must use the **EndDialog** function to complete processing whenever a modal dialog box is created.

You can call **EndDialog** at any time, even in OnInitDialog, in which case you should close the dialog box before it is shown or before the input focus is set.

EndDialog does not close the dialog box immediately. Instead, it sets a flag that directs the dialog box to close as soon as the current message handler returns.

Example

```
/* MyWnd.cpp */
#include "MyDialog.h"

void CMyWnd::ShowDialog()
{
   CMyDialog myDlg;
   int nRet = myDlg.DoModal();

   if ( nRet == IDOK || nRet == 5  )
      AfxMessageBox("Dialog closed successfully");
}

/* MyDialog.cpp */
void CMyDialog::OnSomeAction()
{
   // Do something

   int nRet = 5; // Just any value would do!
   EndDialog(nRet); // This value is returned by DoModal!

   // Do something

   return; // Dialog closed and DoModal returns only here!
}
```

See Also

Reference

CDialog Class
Hierarchy Chart
CDialog::DoModal
CDialog::OnOK
CDialog::OnCancel

Other Resources

CDialog Members

CDialog::GetDefID

Call the **GetDefID** member function to get the ID of the default pushbutton control for a dialog box.

```
DWORD GetDefID( ) const;
```

Return Value

A 32-bit value (**DWORD**). If the default pushbutton has an ID value, the high-order word contains **DC_HASDEFID** and the low-order word contains the ID value. If the default pushbutton does not have an ID value, the return value is 0.

Remarks

This is usually an OK button.

See Also

Reference

CDialog Class
Hierarchy Chart
CDialog::SetDefID
DM_GETDEFID

Other Resources

CDialog Members

CDialog::GotoDlgCtrl

Moves the focus to the specified control in the dialog box.

```
void GotoDlgCtrl(
   CWnd* pWndCtrl
);
```

Parameters

pWndCtrl

Identifies the window (control) that is to receive the focus.

Remarks

To get a pointer to the control (child window) to pass as *pWndCtrl*, call the **CWnd::GetDlgItem** member function, which returns a pointer to a CWnd object.

Example

See the example for CWnd::GetDlgItem.

See Also

Reference

CDialog Class
Hierarchy Chart
CWnd::GetDlgItem
CDialog::PrevDlgCtrl
CDialog::NextDlgCtrl

Other Resources

CDialog Members

CDialog::InitModalIndirect

Call this member function to initialize a modal dialog object using a dialog-box template that you construct in memory.

```
BOOL InitModalIndirect(
   LPCDLGTEMPLATE lpDialogTemplate,
   CWnd* pParentWnd = NULL,
   void* lpDialogInit = NULL
);
   BOOL InitModalIndirect(
   HGLOBAL hDialogTemplate,
   CWnd* pParentWnd = NULL
);
```

Parameters

lpDialogTemplate

Points to memory that contains a dialog-box template used to create the dialog box. This template is in the form of a DLGTEMPLATE structure and control information, as described in the *Platform SDK*.

hDialogTemplate

Contains a handle to global memory containing a dialog-box template. This template is in the form of a **DLGTEMPLATE** structure and data for each control in the dialog box.

pParentWnd

Points to the parent or owner window object (of type CWnd) to which the dialog object belongs. If it is **NULL**, the dialog object's parent window is set to the main application window.

lpDialogInit

Points to a **DLGINIT** resource.

Return Value

Nonzero if the dialog object was created and initialized successfully; otherwise 0.

Remarks

To create a modal dialog box indirectly, first allocate a global block of memory and fill it with the dialog box template. Then call the empty **CDialog** constructor to construct the dialog-box object. Next, call **InitModalIndirect** to store your handle to the in-memory dialog-box template. The Windows dialog box is created and displayed later, when the DoModal member function is called.

Dialog boxes that contain ActiveX controls require additional information provided in a **DLGINIT** resource. For more information, see Knowledge Base article Q231591, " HOWTO: Use a Dialog Template to Create a MFC Dialog with an ActiveX Control." Knowledge Base articles are available in the MSDN Library Visual Studio documentation or at http://support.microsoft.com.

See Also

Reference

CDialog Class
Hierarchy Chart
DialogBoxIndirect
CDialog::DoModal
CWnd::DestroyWindow
CDialog::CDialog

Other Resources

CDialog Members

CDialog::MapDialogRect

Call to convert the dialog-box units of a rectangle to screen units.

```
void MapDialogRect(
   LPRECT lpRect
) const;
```

Parameters

lpRect

Points to a RECT structure or CRect object that contains the dialog-box coordinates to be converted.

Remarks

Dialog-box units are stated in terms of the current dialog-box base unit derived from the average width and height of characters in the font used for dialog-box text. One horizontal unit is one-fourth of the dialog-box base-width unit, and one vertical unit is one-eighth of the dialog-box base height unit.

The **GetDialogBaseUnits** Windows function returns size information for the system font, but you can specify a different font for each dialog box if you use the **DS_SETFONT** style in the resource-definition file. The **MapDialogRect** Windows function uses the appropriate font for this dialog box.

The **MapDialogRect** member function replaces the dialog-box units in *lpRect* with screen units (pixels) so that the rectangle can be used to create a dialog box or position a control within a box.

See Also

Reference

CDialog Class
Hierarchy Chart
GetDialogBaseUnits
MapDialogRect
WM_SETFONT

Other Resources

CDialog Members

CDialog::NextDlgCtrl

Moves the focus to the next control in the dialog box.

```
void NextDlgCtrl( ) const;
```

Remarks

If the focus is at the last control in the dialog box, it moves to the first control.

See Also

Reference

CDialog Class
Hierarchy Chart
CDialog::PrevDlgCtrl
CDialog::GotoDlgCtrl

Other Resources

CDialog Members

CDialog::OnCancel

The framework calls this member function when the user clicks the Cancel button or presses the ESC key in a modal or modeless dialog box.

```
virtual void OnCancel( );
```

Remarks

Override this member function to perform Cancel button action. The default simply terminates a modal dialog box by calling **EndDialog** and causing **DoModal** to return **IDCANCEL**.

If you implement the Cancel button in a modeless dialog box, you must override the **OnCancel** member function and call **DestroyWindow** from within it. Don't call the base-class member function, because it calls **EndDialog**, which will make the dialog box invisible but not destroy it.

Example

```
/* MyDialog.cpp */
#include "MyDialog.h"

void CMyDialog::OnCancel()
{
   // TODO: Add extra cleanup here

   // Ensure that you reset all the values back to the
   // ones before modification. This handler is called
   // when the user doesn't want to save the changes.

   if ( AfxMessageBox("Are you sure you want to abort the changes?",
         MB_YESNO) == IDNO )
      return; // Give the user a chance if he has unknowingly hit the
              // Cancel button. If he says No, return. Don't reset. If
              // Yes, go ahead and reset the values and close the dialog.

   m_nMyValue = m_nPrevValue;
   m_pMyString = NULL;

   CDialog::OnCancel();
}
```

See Also

Reference

CDialog Class
Hierarchy Chart
CDialog::OnOK
CDialog::EndDialog

Other Resources

CDialog Members

CDialog::OnInitDialog

This member function is called in response to the **WM_INITDIALOG** message.

```
virtual  BOOL OnInitDialog( );
```

Return Value

Specifies whether the application has set the input focus to one of the controls in the dialog box. If **OnInitDialog** returns nonzero, Windows sets the input focus to the first control in the dialog box. The application can return 0 only if it has explicitly set the input focus to one of the controls in the dialog box.

Remarks

This message is sent to the dialog box during the **Create**, **CreateIndirect**, or **DoModal** calls, which occur immediately before the dialog box is displayed.

Override this member function if you need to perform special processing when the dialog box is initialized. In the overridden version, first call the base class **OnInitDialog** but disregard its return value. You will normally return **TRUE** from your overridden member function.

Windows calls the **OnInitDialog** function via the standard global dialog-box procedure common to all Microsoft Foundation Class Library dialog boxes, rather than through your message map, so you do not need a message-map entry for this member function.

Example

```
/* MyDialog.cpp */
#include "MyDialog.h"

BOOL CMyDialog::OnInitDialog()
{
   CDialog::OnInitDialog();

   // TODO: Add extra initialization here
   m_cMyEdit.SetWindowText("My Name"); // Initialize control values
   m_cMyList.ShowWindow(SW_HIDE);      // Show or hide a control, etc.

   return TRUE;   // return TRUE unless you set the focus to a control
                  // EXCEPTION: OCX Property Pages should return FALSE
}
```

See Also

Reference

CDialog Class
Hierarchy Chart
CDialog::Create
CDialog::CreateIndirect
WM_INITDIALOG

Other Resources

CDialog Members

CDialog::OnOK

Called when the user clicks the OK button (the button with an ID of **IDOK**).

```
virtual void OnOK( );
```

Remarks

Override this member function to perform the OK button action. If the dialog box includes automatic data validation and exchange, the default implementation of this member function validates the dialog-box data and updates the appropriate variables in your application.

If you implement the OK button in a modeless dialog box, you must override the **OnOK** member function and call **DestroyWindow** from within it. Don't call the base-class member function, because it calls **EndDialog**, which makes the dialog box invisible but does not destroy it.

Example

```
/* MyDialog.cpp */
#include "MyDialog.h"

void CMyDialog::OnOK()
{
   // TODO: Add extra validation here

   // Ensure that your UI got the necessary input
   // from the user before closing the dialog. The
   // default OnOK will close this.
   if ( m_nMyValue == 0 ) // Is a particular field still empty?
   {
      AfxMessageBox("Please enter a value for MyValue");
      return; // Inform the user that he can't close the dialog without
              // entering the necessary values and don't close the
              // dialog.
   }

   CDialog::OnOK(); // This will close the dialog and DoModal will return.
}
```

See Also

Reference

CDialog Class
Hierarchy Chart
CDialog::OnCancel
CDialog::EndDialog

Other Resources

CDialog Members

CDialog::OnSetFont

Specifies the font a dialog-box control will use when drawing text.

```
virtual  void OnSetFont(
   CFont* pFont
);
```

Parameters

pFont

Specifies a pointer to the font. Used as the default font for all controls in this dialog box.

Remarks

The dialog-box control will use the specified font as the default for all dialog-box controls.

The dialog editor typically sets the dialog-box font as part of the dialog-box template resource.

See Also

Reference

CDialog Class
Hierarchy Chart
WM_SETFONT
CWnd::SetFont

Other Resources

CDialog Members

CDialog::PrevDlgCtrl

Sets the focus to the previous control in the dialog box.

```
void PrevDlgCtrl( ) const;
```

Remarks

If the focus is at the first control in the dialog box, it moves to the last control in the box.

See Also

Reference

CDialog Class
Hierarchy Chart
CDialog::NextDlgCtrl
CDialog::GotoDlgCtrl

Other Resources

CDialog Members

CDialog::SetDefID

Changes the default pushbutton control for a dialog box.

```
void SetDefID(
   UINT nID
);
```

Parameters

nID

Specifies the ID of the pushbutton control that will become the default.

See Also

Reference

CDialog Class

Hierarchy Chart

CDialog::GetDefID

Other Resources

CDialog Members

CDialog::SetHelpID

Sets a context-sensitive help ID for the dialog box.

```
void SetHelpID(
   UINT nIDR
);
```

Parameters

nIDR

Specifies the context-sensitive help ID.

See Also

Reference

CDialog Class

Hierarchy Chart

Other Resources

CDialog Members

CDialogBar Class

This topic has been updated for Visual Studio 2005 SP1.

Provides the functionality of a Windows modeless dialog box in a control bar.

```
class CDialogBar : public CControlBar
```

Remarks

A dialog bar resembles a dialog box in that it contains standard Windows controls that the user can tab between. Another similarity is that you create a dialog template to represent the dialog bar.

Creating and using a dialog bar is similar to creating and using a **CFormView** object. First, use the dialog editor to define a dialog template with the style **WS_CHILD** and no other style. The template must not have the style **WS_VISIBLE**. In your application code, call the constructor to construct the **CDialogBar** object, then call **Create** to create the dialog-bar window and attach it to the **CDialogBar** object.

For more information on **CDialogBar**, see the article Dialog Bars and Technical Note 31, Control Bars.

Note
In the current release, a **CDialogBar** object cannot host Windows Forms controls. For more information about Windows Forms controls in Visual C++, see Using a Windows Form User Control in MFC.

Requirements

Header: afxext.h

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

See Also

Tasks

CTRLBARS Sample: Illustrates Custom Control Bars

Reference

CControlBar Class

Hierarchy Chart

CFormView Class

CControlBar Class

Other Resources

CDialogBar Members

CDialogBar Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CControlBar Members

Construction

CDialogBar	Constructs a **CDialogBar** object.
Create	Creates a Windows dialog bar and attaches it to the **CDialogBar** object.

See Also

Reference

CDialogBar Class

Hierarchy Chart

CDialogBar Member Functions

For information about the member functions in **CDialogBar**, see CDialogBar Members.

CDialogBar::CDialogBar

Constructs a **CDialogBar** object.

```
CDialogBar( );
```

See Also

Reference

CDialogBar Class

Hierarchy Chart

CControlBar Class

Other Resources

CDialogBar Members

CDialogBar::Create

Loads the dialog-box resource template specified by *lpszTemplateName* or *nIDTemplate*, creates the dialog-bar window, sets its style, and associates it with the **CDialogBar** object.

```
virtual BOOL Create(
   CWnd* pParentWnd,
   LPCTSTR lpszTemplateName,
   UINT nStyle,
   UINT nID
);
virtual BOOL Create(
   CWnd* pParentWnd,
   UINT nIDTemplate,
   UINT nStyle,
   UINT nID
);
```

Parameters

pParentWnd

A pointer to the parent **CWnd** object.

lpszTemplateName

A pointer to the name of the **CDialogBar** object's dialog-box resource template.

nStyle

The alignment style of the dialog bar. The following styles are supported:

- **CBRS_TOP** Control bar is at the top of the frame window.
- **CBRS_BOTTOM** Control bar is at the bottom of the frame window.
- **CBRS_NOALIGN** Control bar is not repositioned when the parent is resized.
- **CBRS_LEFT** Control bar is at the left of the frame window.
- **CBRS_RIGHT** Control bar is at the right of the frame window.

nID

The control ID of the dialog bar.

nIDTemplate

The resource ID of the **CDialogBar** object's dialog-box template.

Return Value

Nonzero if successful; otherwise 0.

Remarks

If you specify the **CBRS_TOP** or **CBRS_BOTTOM** alignment style, the dialog bar's width is that of the frame window and its height is that of the resource specified by *nIDTemplate*. If you specify the **CBRS_LEFT** or **CBRS_RIGHT** alignment style, the dialog bar's height is that of the frame window and its width is that of the resource specified by *nIDTemplate*.

Example

```
// Mainfrm.h.

class CMainFrame : public CFrameWnd
{
// Constructor.
public:
   CMainFrame();
```

```
   virtual ~CMainFrame();

protected:  // Control bar embedded members.
   CDialogBar  m_wndDlgBar;

   // Other data members and methods.

   DECLARE_MESSAGE_MAP()
};


// Mainfrm.cpp.

#include "MainFrm.h"

/////////////////////////////////////////////////////////////////////////////

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
      return -1;

   EnableDocking(CBRS_ALIGN_ANY);

   // IDD_VIEWSELECT - Resource ID of the dialog
   // template. This dialog template should be created
   // with  the  style  WS_CHILD and no other style.
   // The template must not have the style WS_VISIBLE.

   if (!m_wndDlgBar.Create(this, IDD_VIEWSELECT,
      CBRS_LEFT|CBRS_TOOLTIPS|CBRS_FLYBY, IDD_VIEWSELECT))
   {
      TRACE0("Failed to create DlgBar\n");
      return -1;      // Fail to create.
   }

   return 0;
}
```

See Also

Reference

CDialogBar Class

Hierarchy Chart

CDialogBar::CDialogBar

Other Resources

CDialogBar Members

CDocItem Class

The base class for document items, which are components of a document's data.

```
class CDocItem : public CCmdTarget
```

Remarks

CDocItem objects are used to represent OLE items in both client and server documents.

For more information, see the article Containers: Implementing a Container.

Requirements

Header: afxole.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CCmdTarget Class
Hierarchy Chart
COleDocument Class
COleServerItem Class
COleClientItem Class

Other Resources

CDocItem Members

CDocItem Members

Base Class Members

CObject Members

CCmdTarget Members

Operations

GetDocument	Returns the document that contains the item.

Overridables

IsBlank	Determines whether the item contains any information.

See Also

Reference

CDocItem Class

Hierarchy Chart

CDocItem Member Functions

For information about the member functions in **CDocItem**, see CDocItem Members.

CDocItem::GetDocument

Call this function to get the document that contains the item.

```
CDocument* GetDocument( ) const;
```

Return Value

A pointer to the document that contains the item; **NULL**, if the item is not part of a document.

Remarks

This function is overridden in the derived classes COleClientItem and COleServerItem, returning a pointer to either a COleDocument, a COleLinkingDoc, or a COleServerDoc object.

See Also

Reference

CDocItem Class
Hierarchy Chart
COleDocument Class
COleLinkingDoc Class
COleServerDoc Class
COleClientItem Class
COleServerItem Class

Other Resources

CDocItem Members

CDocItem::IsBlank

Called by the framework when default serialization occurs.

```
virtual BOOL IsBlank( ) const;
```

Return Value

Nonzero if the item contains no information; otherwise 0.

Remarks

By default, **CDocItem** objects are not blank. COleClientItem objects are sometimes blank because they derive directly from **CDocItem**. However, COleServerItem objects are always blank. By default, OLE applications containing **COleClientItem** objects that have no x or y extent are serialized. This is done by returning **TRUE** from an override of **IsBlank** when the item has no x or y extent.

Override this function if you want to implement other actions during serialization.

See Also

Reference

CDocItem Class
Hierarchy Chart
CObject::Serialize

Other Resources

CDocItem Members

CDockState Class

A serialized **CObject** class that loads, unloads, or clears the state of one or more docking control bars in persistent memory (a file).

```
class CDockState : public CObject
```

Remarks

The dock state includes the size and position of the bar and whether or not it is docked. When retrieving the stored dock state, **CDockState** checks the bar's position and, if the bar is not visible with the current screen settings, **CDockState** scales the bar's position so that it is visible. The main purpose of **CDockState** is to hold the entire state of a number of control bars and to allow that state to be saved and loaded either to the registry, the application's .INI file, or in binary form as part of a **CArchive** object's contents.

The bar can be any dockable control bar, including a toolbar, status bar, or dialog bar. **CDockState** objects are written and read to or from a file via a **CArchive** object.

CFrameWnd::GetDockState retrieves the state information of all the frame window's **CControlBar** objects and puts it into the **CDockState** object. You can then write the contents of the **CDockState** object to storage with Serialize or CDockState::SaveState. If you later want to restore the state of the control bars in the frame window, you can load the state with **Serialize** or CDockState::LoadState, then use CFrameWnd::SetDockState to apply the saved state to the frame window's control bars.

For more information on docking control bars, see the articles Control Bars, Toolbars: Docking and Floating, and Frame Windows.

Requirements

Header: afxadv.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class

Hierarchy Chart

Other Resources

CDockState Members

CDockState Members

Base Class Members

CObject Members

Data Members

m_arrBarInfo	Array of pointers to the stored dock state information with one entry for each control bar.

Construction

CDockState	Constructs a **CDockState** object.

Operations

Clear	Clears the dock state information.
GetVersion	Retrieves the version number of the stored bar state.
LoadState	Retrieves state information from the registry or .INI file.
SaveState	Saves state information to the registry or INI file.

See Also

Reference

CDockState Class
Hierarchy Chart

CDockState Member Functions

For information about the member functions in **CDockState**, see CDockState Members.

CDockState::CDockState

Constructs a **CDockState** object.

```
CDockState( );
```

See Also

Reference

CDockState Class
Hierarchy Chart
CDockState::Clear
CDockState::GetVersion
CDockState::LoadState
CDockState::SaveState
CFrameWnd::GetDockState
CFrameWnd::SetDockState

Other Resources

CDockState Members

CDockState::Clear

Call this function to clear all docking information stored in the **CDockState** object.

```
void Clear( );
```

Remarks

This includes not only whether the bar is docked or not, but the bar's size and position and whether or not it is visible.

See Also

Reference

CDockState Class
Hierarchy Chart
CDockState::LoadState
CDockState::SaveState
CDockState::GetVersion
CFrameWnd::GetDockState
CFrameWnd::SetDockState

Other Resources

CDockState Members

CDockState::GetVersion

Call this function to retrieve the version number of the stored bar state.

```
DWORD GetVersion( );
```

Return Value

1 if the stored bar information is older than current bar state; 2 if the stored bar information is the same as the current bar state.

Remarks

Version support enables a revised bar to add new persistent properties and still be able to detect and load the persistent state created by an earlier version of the bar.

See Also

Reference

CDockState Class
Hierarchy Chart
CDockState::LoadState
CDockState::SaveState
CDockState::Clear
CFrameWnd::GetDockState
CFrameWnd::SetDockState

Other Resources

CDockState Members

CDockState::LoadState

Call this function to retrieve state information from the registry or .INI file.

```
void LoadState(
   LPCTSTR lpszProfileName
);
```

Parameters

lpszProfileName

Points to a null-teminated string that specifies the name of a section in the initialization file or a key in the Windows registry where state information is stored.

Remarks

The profile name is the section of the application's .INI file or the registry that contains the bars' state information. You can save control bar state information to the registry or .INI file with **SaveState**.

See Also

Reference

CDockState Class
Hierarchy Chart
CDockState::SaveState
CDockState::GetVersion
CDockState::Clear
CFrameWnd::GetDockState
CFrameWnd::SetDockState

Other Resources

CDockState Members

CDockState::SaveState

Call this function to save the state information to the registry or .INI file.

```
void SaveState(
   LPCTSTR lpszProfileName
);
```

Parameters

lpszProfileName

Points to a null-teminated string that specifies the name of a section in the initialization file or a key in the Windows registry where state information is stored.

Remarks

The profile name is the section of the application's .INI file or the registry that contains the control bar's state information. **SaveState** also saves the current screen size. You can retrieve control bar information from the registry or .INI file with **LoadState**.

See Also

Reference

CDockState Class
Hierarchy Chart
CDockState::LoadState
CDockState::GetVersion
CDockState::Clear
CFrameWnd::GetDockState
CFrameWnd::SetDockState

Other Resources

CDockState Members

CDockState Data Members

For information about the data members in **CDockState**, see CDockState Members.

CDockState::m_arrBarInfo

A **CPtrArray** object that is an array of pointers to the stored control bar information for each control bar that has saved state information in the **CDockState** object.

```
CPtrArray m_arrBarInfo;
```

See Also

Reference

CDockState Class
Hierarchy Chart
CDockState::LoadState
CDockState::SaveState
CDockState::Clear
CFrameWnd::GetDockState
CFrameWnd::SetDockState

Other Resources

CDockState Members

CDocObjectServer Class

Implements the additional OLE interfaces needed to make a normal **COleDocument** server into a full DocObject server: **IOleDocument**, **IOleDocumentView**, **IOleCommandTarget**, and **IPrint**.

```
class CDocObjectServer : public CCmdTarget
```

Remarks

CDocObjectServer is derived from **CCmdTarget** and works closely with **COleServerDoc** to expose the interfaces.

A DocObject server document can contain CDocObjectServerItem objects, which represent the server interface to DocObject items.

To customize your DocObject server, derive your own class from **CDocObjectServer** and override its view setup functions, OnActivateView, OnApplyViewState, and OnSaveViewState. You will need to provide a new instance of your class in response to framework calls.

For further information on DocObjects, see CDocObjectServerItem and COleCmdUI in the *MFC Reference*. Also see Internet First Steps: Active Documents and Active Documents.

Also see the following Knowledge Base article:

- Q247382 : PRB: ToolTips for Controls in ActiveX Document Server Are Hidden by the ActiveX Document Container

Requirements

Header: afxdocob.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CCmdTarget Class
Hierarchy Chart
CDocObjectServerItem Class

Other Resources

CDocObjectServer Members

CDocObjectServer Members

Base Class Members

CObject Members

CCmdTarget Members

Constructors

CDocObjectServer	Constructs a **CDocObjectServer** object.

Operations

ActivateDocObject	Activates the document object server, but does not show it.

Overrideables

OnActivateView	Displays the DocObject view.
OnApplyViewState	Restores the state of the DocObject view.
OnSaveViewState	Saves the state of the DocObject view.

See Also

Reference

CDocObjectServer Class

Hierarchy Chart

CDocObjectServer Member Functions

For information about the member functions in **CDocObjectServer**, see CDocObjectServer Members.

CDocObjectServer::ActivateDocObject

Call this function to activate (but not show) the document object server.

```
void ActivateDocObject( );
```

Remarks

ActivateDocObject calls **IOleDocumentSite**'s **ActivateMe** method, but does not show the view because it waits for specific instructions on how to set up and display the view, given in the call to CDocObjectServer::OnActivateView.

Together, **ActivateDocObject** and **OnActivateView** activate and display the DocObject view. DocObject activation differs from other kinds of OLE in-place activation. DocObject activation bypasses displaying in-place hatch borders and object adornments (such as sizing handles), ignores object extent functions, and draws scroll bars within the view rectangle as opposed to drawing them outside that rectangle (as in normal in-place activation).

See Also

Reference

CDocObjectServer Class
Hierarchy Chart
CDocObjectServerItem Class

Other Resources

CDocObjectServer Members

CDocObjectServer::CDocObjectServer

Constructs and initializes a **CDocObjectServer** object.

```
explicit CDocObjectServer(
   COleServerDoc* pOwner,
   LPOLEDOCUMENTSITE pDocSite = NULL
);
```

Parameters

pOwner

A pointer to the client site document that is the client for the DocObject server.

pDocSite

A pointer to the **IOleDocumentSite** interface implemented by the container.

Remarks

When a DocObject is active, the client site OLE interface (**IOleDocumentSite**) is what allows the DocObject server to communicate with its client (the container). When a DocObject server is activated, it first checks that the container implements the **IOleDocumentSite** interface. If so, COleServerDoc::GetDocObjectServer is called to see if the container supports DocObjects. By default, **GetDocObjectServer** returns **NULL**. You must override **COleServerDoc::GetDocObjectServer** to construct a new **CDocObjectServer** object or a derived object of your own, with pointers to the **COleServerDoc** container and its **IOleDocumentSite** interface as arguments to the constructor.

See Also

Reference

CDocObjectServer Class
Hierarchy Chart
CDocObjectServerItem Class
COleServerDoc::GetDocObjectServer

Other Resources

CDocObjectServer Members

CDocObjectServer::OnActivateView

Call this function to display the DocObject view.

```
virtual HRESULT OnActivateView( );
```

Return Value

Returns an error or warning value. By default, returns **NOERROR** if successful; otherwise, **E_FAIL**.

Remarks

This function creates an in-place frame window, draws scrollbars within the view, sets up the menus the server shares with its container, adds frame controls, sets the active object, then finally shows the in-place frame window and sets the focus.

See Also

Reference

CDocObjectServer Class
Hierarchy Chart
CDocObjectServer::OnApplyViewState

Other Resources

CDocObjectServer Members

CDocObjectServer::OnApplyViewState

Override this function to restore the state of the DocObject view.

```
virtual void OnApplyViewState(
   CArchive& ar
);
```

Parameters

ar

A **CArchive** object from which to serialize the view state.

Remarks

This function is called when the view is being displayed for the first time after its instantiation. **OnApplyViewState** instructs a view to reinitialize itself according to the data in the **CArchive** object previously saved with OnSaveViewState. The view must validate the data in the **CArchive** object because the container does not attempt to interpret the view state data in any way.

You can use **OnSaveViewState** to store persistent information specific to your view's state. If you override **OnSaveViewState** to store information, you will want to override **OnApplyViewState** to read that information and apply it to your view when it is newly activated.

See Also

Reference

CDocObjectServer Class
Hierarchy Chart
CDocObjectServer::OnSaveViewState

Other Resources

CDocObjectServer Members

CDocObjectServer::OnSaveViewState

Override this function to save extra state information about your DocObject view.

```
virtual void OnSaveViewState(
   CArchive& ar
);
```

Parameters

ar

A **CArchive** object to which the view state is serialized.

Remarks

Your state might include properties like the view type, zoom factor, insertion and selection point, and so on. The container typically calls this function before deactivating the view. The saved state can later be restored through OnApplyViewState.

You can use **OnSaveViewState** to store persistent information specific to your view's state. If you override **OnSaveViewState** to store information, you will want to override **OnApplyViewState** to read that information and apply it to your view when it is newly activated.

See Also

Reference

CDocObjectServer Class
Hierarchy Chart
CDocObjectServer::OnApplyViewState

Other Resources

CDocObjectServer Members

CDocObjectServerItem Class

Implements OLE server verbs specifically for DocObject servers.

```
class CDocObjectServerItem : public COleServerItem
```

Remarks

CDocObjectServerItem defines overridable member functions: OnHide, OnOpen, and OnShow.

To use **CDocObjectServerItem**, assure that the OnGetEmbeddedItem override in your **COleServerDoc**-derived class returns a new **CDocObjectServerItem** object. If you need to change any functionality in your item, you can create a new instance of your own **CDocObjectServerItem**-derived class.

For further information on DocObjects, see CDocObjectServer and COleCmdUI in the *MFC Reference*. Also see Internet First Steps: Active Documents and Active Documents.

Requirements

Header: afxdocob.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

COleServerItem Class
Hierarchy Chart
CDocObjectServer Class
COleDocObjectItem Class

Other Resources

CDocObjectServerItem Members

CDocObjectServerItem Members

Base Class Members

CObject Members

CCmdTarget Members

CDocItem Members

COleServerItem Members

Methods

CDocObjectServerItem	Constructs a **CDocObjectServerItem** object.
GetDocument	Retrieves a pointer to the document that contains the item.

Overridables

OnHide	Throws an exception if the framework tries to hide a DocObject item.
OnOpen	Called by the framework to make the DocObject item in-place active. If the item is not a DocObject, calls COleServerItem::OnOpen.
OnShow	Called by the framework to make the DocObject item in-place active. If the item is not a DocObject, calls COleServerItem::OnShow.

See Also

Reference

CDocObjectServerItem Class

Hierarchy Chart

CDocObjectServerItem Member Functions

For information about the member functions in **CDocObjectServerItem**, see CDocObjectServerItem Members.

CDocObjectServerItem::CDocObjectServerItem

Constructs a **CDocObjectServerItem** object.

```
CDocObjectServerItem(
   COleServerDoc* pServerDoc,
      BOOL bAutoDelete
);
```

Parameters

pServerDoc

A pointer to the document that will contain the new DocObject item.

bAutoDelete

Indicates whether the object can be deleted when a link to it is released. Set the argument to **FALSE** if the **CDocObjectServerItem** object is an integral part of your document's data. Set it to **TRUE** if the object is a secondary structure used to identify a range in your document's data that can be deleted by the framework.

See Also

Reference

CDocObjectServerItem Class

Hierarchy Chart

CDocObjectServer Class

Other Resources

CDocObjectServerItem Members

CDocObjectServerItem::GetDocument

Retrieves a pointer to the document that contains the item.

```
COleServerDoc* GetDocument( ) const;
```

Return Value

A pointer to the document that contains the item; **NULL** if the item is not part of a document.

Remarks

This allows access to the server document that you passed as an argument to the CDocObjectServerItem constructor.

See Also

Reference

CDocObjectServerItem Class
Hierarchy Chart
CDocObjectServer Class

Other Resources

CDocObjectServerItem Members

CDocObjectServerItem::OnHide

Called by the framework to hide the item.

```
virtual void OnHide( );
```

Remarks

The default implementation throws an exception if the item is a DocObject. You cannot hide an active DocObject item because it takes the whole view. You must deactivate the DocObject item to make it disappear. If the item is not a DocObject, the default implementation calls COleServerItem::OnHide.

See Also

Reference

CDocObjectServerItem Class
Hierarchy Chart
CDocObjectServerItem::OnOpen
CDocObjectServerItem::OnShow

Other Resources

CDocObjectServerItem Members

CDocObjectServerItem::OnOpen

Called by the framework to instruct the server application to make the DocObject item in-place active.

```
virtual void OnOpen( );
```

Remarks

If the item is not a DocObject, the default implementation calls COleServerItem::OnOpen. Override this function if you want to perform special processing when opening a DocObject item.

See Also

Reference

CDocObjectServerItem Class
Hierarchy Chart
CDocObjectServerItem::OnHide
CDocObjectServerItem::OnShow

Other Resources

CDocObjectServerItem Members

CDocObjectServerItem::OnShow

Called by the framework to instruct the server application to make the DocObject item in-place active.

```
virtual void OnShow( );
```

Remarks

If the item is not a DocObject, the default implementation calls COleServerItem::OnShow. Override this function if you want to perform special processing when opening a DocObject item.

See Also

Reference

CDocObjectServerItem Class
Hierarchy Chart
CDocObjectServerItem::OnHide
CDocObjectServerItem::OnOpen

Other Resources

CDocObjectServerItem Members

CDocTemplate Class

An abstract base class that defines the basic functionality for document templates.

```
class CDocTemplate : public CCmdTarget
```

Remarks

You usually create one or more document templates in the implementation of your application's **InitInstance** function. A document template defines the relationships among three types of classes:

- A document class, which you derive from **CDocument**.
- A view class, which displays data from the document class listed above. You can derive this class from **CView**, **CScrollView**, **CFormView**, or **CEditView**. (You can also use **CEditView** directly.)
- A frame window class, which contains the view. For a single document interface (SDI) application, you derive this class from **CFrameWnd**. For a multiple document interface (MDI) application, you derive this class from **CMDIChildWnd**. If you don't need to customize the behavior of the frame window, you can use **CFrameWnd** or **CMDIChildWnd** directly without deriving your own class.

Your application has one document template for each type of document that it supports. For example, if your application supports both spreadsheets and text documents, the application has two document template objects. Each document template is responsible for creating and managing all the documents of its type.

The document template stores pointers to the **CRuntimeClass** objects for the document, view, and frame window classes. These **CRuntimeClass** objects are specified when constructing a document template.

The document template contains the ID of the resources used with the document type (such as menu, icon, or accelerator table resources). The document template also has strings containing additional information about its document type. These include the name of the document type (for example, "Worksheet") and the file extension (for example, ".xls"). Optionally, it can contain other strings used by the application's user interface, the Windows File Manager, and Object Linking and Embedding (OLE) support.

If your application is an OLE container and/or server, the document template also defines the ID of the menu used during in-place activation. If your application is an OLE server, the document template defines the ID of the toolbar and menu used during in-place activation. You specify these additional OLE resources by calling **SetContainerInfo** and **SetServerInfo**.

Because **CDocTemplate** is an abstract class, you cannot use the class directly. A typical application uses one of the two **CDocTemplate**-derived classes provided by the Microsoft Foundation Class Library: **CSingleDocTemplate**, which implements SDI, and **CMultiDocTemplate**, which implements MDI. See those classes for more information on using document templates.

If your application requires a user-interface paradigm that is fundamentally different from SDI or MDI, you can derive your own class from **CDocTemplate**.

For more information on **CDocTemplate**, see Document Templates and the Document/View Creation Process.

Requirements

Header: afxwin.h

See Also

Reference

CCmdTarget Class
Hierarchy Chart
CSingleDocTemplate Class
CMultiDocTemplate Class
CDocument Class
CView Class
CScrollView Class
CEditView Class
CFormView Class
CFrameWnd Class

CMDIChildWnd Class

Other Resources

CDocTemplate Members

CDocTemplate Members

Base Class Members

CObject Members

CCmdTarget Members

Constructors

CDocTemplate	Constructs a **CDocTemplate** object.

Attributes

SetContainerInfo	Determines the resources for OLE containers when editing an in-place OLE item.
SetServerInfo	Determines the resources and classes when the server document is embedded or edited in-place.
GetFirstDocPosition	Retrieves the position of the first document associated with this template.
GetNextDoc	Retrieves a document and the position of the next one.
LoadTemplate	Loads the resources for a given **CDocTemplate** or derived class.

Operations

AddDocument	Adds a document to a template.
RemoveDocument	Removes a document from a template.
GetDocString	Retrieves a string associated with the document type.
CreateOleFrame	Creates an OLE-enabled frame window.

Overridables

MatchDocType	Determines the degree of confidence in the match between a document type and this template.
CreateNewDocument	Creates a new document.
CreateNewFrame	Creates a new frame window containing a document and view.
InitialUpdateFrame	Initializes the frame window, and optionally makes it visible.
SaveAllModified	Saves all documents associated with this template which have been modified.
CloseAllDocuments	Closes all documents associated with this template.
OpenDocumentFile	Opens a file specified by a pathname.
SetDefaultTitle	Displays the default title in the document window's title bar.

See Also

Reference

CDocTemplate Class

Hierarchy Chart

CDocTemplate Member Functions

For information about the member functions in **CDocTemplate**, see CDocTemplate Members.

CDocTemplate::AddDocument

Use this function to add a document to a template.

```
virtual void AddDocument(
   CDocument* pDoc
);
```

Parameters

pDoc

A pointer to the document to be added.

Remarks

The derived classes CMultiDocTemplate and CSingleDocTemplate override this function. If you derive your own document-template class from **CDocTemplate**, your derived class must override this function.

See Also

Reference

CDocTemplate Class
Hierarchy Chart
CDocTemplate::RemoveDocument
CMultiDocTemplate Class
CSingleDocTemplate Class

Other Resources

CDocTemplate Members

CDocTemplate::CDocTemplate

Constructs a **CDocTemplate** object.

```
CDocTemplate (
   UINT nIDResource,
   CRuntimeClass* pDocClass,
   CRuntimeClass* pFrameClass,
   CRuntimeClass* pViewClass
);
```

Parameters

nIDResource

Specifies the ID of the resources used with the document type. This may include menu, icon, accelerator table, and string resources.

The string resource consists of up to seven substrings separated by the '\n' character (the '\n' character is needed as a place holder if a substring is not included; however, trailing '\n' characters are not necessary); these substrings describe the document type. For information on the substrings, see GetDocString. This string resource is found in the application's resource file. For example:

```
// MYCALC.RC
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    IDR_SHEETTYPE "\nSheet\nWorksheet\nWorksheets (*.myc)\n.myc\n MyCalcSheet\nMyCalc Wor
ksheet"
END
```

Note that the string begins with a '\n' character; this is because the first substring is not used for MDI applications and so is not included. You can edit this string using the string editor; the entire string appears as a single entry in the String Editor, not as seven separate entries.

pDocClass

Points to the **CRuntimeClass** object of the document class. This class is a **CDocument**-derived class you define to represent your documents.

pFrameClass

Points to the **CRuntimeClass** object of the frame window class. This class can be a **CFrameWnd**-derived class, or it can be **CFrameWnd** itself if you want default behavior for your main frame window.

pViewClass

Points to the **CRuntimeClass** object of the view class. This class is a **CView**-derived class you define to display your documents.

Remarks

Use this member function to construct a **CDocTemplate** object. Dynamically allocate a **CDocTemplate** object and pass it to CWinApp::AddDocTemplate from the `InitInstance` member function of your application class.

See Also

Reference

CDocTemplate Class
Hierarchy Chart
CDocTemplate::GetDocString
CMultiDocTemplate::CMultiDocTemplate
CSingleDocTemplate::CSingleDocTemplate
CWinApp::AddDocTemplate
CWinApp::InitInstance
CRuntimeClass Structure

Other Resources

CDocTemplate::CloseAllDocuments

Call this member function to close all open documents.

```
virtual void CloseAllDocuments(
   BOOL bEndSession
);
```

Parameters

bEndSession

Specifies whether or not the session is being ended. It is **TRUE** if the session is being ended; otherwise **FALSE**.

Remarks

This member function is typically used as part of the File Exit command. The default implementation of this function calls the CDocument::DeleteContents member function to delete the document's data and then closes the frame windows for all the views attached to the document.

Override this function if you want to require the user to perform special cleanup processing before the document is closed. For example, if the document represents a record in a database, you may want to override this function to close the database.

See Also

Reference

CDocTemplate Class
Hierarchy Chart
CDocTemplate::OpenDocumentFile
CDocTemplate::SaveAllModified

Other Resources

CDocTemplate Members

CDocTemplate::CreateNewDocument

Call this member function to create a new document of the type associated with this document template.

```
virtual CDocument* CreateNewDocument( );
```

Return Value

A pointer to the newly created document, or **NULL** if an error occurs.

See Also

Reference

CDocTemplate Class
Hierarchy Chart
CDocTemplate::CreateNewFrame

Other Resources

CDocTemplate Members

CDocTemplate::CreateNewFrame

Creates a new frame window containing a document and view.

```
virtual CFrameWnd* CreateNewFrame(
   CDocument* pDoc,
   CFrameWnd* pOther
);
```

Parameters

pDoc

The document to which the new frame window should refer. Can be **NULL**.

pOther

The frame window on which the new frame window is to be based. Can be **NULL**.

Return Value

A pointer to the newly created frame window, or **NULL** if an error occurs.

Remarks

CreateNewFrame uses the **CRuntimeClass** objects passed to the constructor to create a new frame window with a view and document attached. If the *pDoc* parameter is **NULL**, the framework outputs a TRACE message.

The *pOther* parameter is used to implement the Window New command. It provides a frame window on which to model the new frame window. The new frame window is usually created invisible. Call this function to create frame windows outside the standard framework implementation of File New and File Open.

See Also

Reference

CDocTemplate Class
Hierarchy Chart
CCreateContext Structure
CFrameWnd::LoadFrame
CDocTemplate::InitialUpdateFrame

Other Resources

CDocTemplate Members

CDocTemplate::CreateOleFrame

Creates an OLE frame window.

```
CFrameWnd* CreateOleFrame(
   CWnd* pParentWnd,
   CDocument* pDoc,
   BOOL bCreateView
);
```

Parameters

pParentWnd

A pointer to the frame's parent window.

pDoc

A pointer to the document to which the new OLE frame window should refer.

bCreateView

Determines whether a view is created along with the frame.

Return Value

A pointer to a frame window if successful; otherwise **NULL**.

Remarks

If *bCreateView* is zero, an empty frame is created.

See Also

Reference

CDocTemplate Class
Hierarchy Chart
CDocTemplate::CreateNewFrame
COleDocument Class
COleIPFrameWnd Class

Other Resources

CDocTemplate Members

CDocTemplate::GetDocString

Retrieves a string associated with the document type.

```
virtual BOOL GetDocString(
   CString& rString,
   enum DocStringIndex index
) const;
```

Parameters

rString

A reference to a **CString** object that will contain the string when the function returns.

index

An index of the substring being retrieved from the string that describes the document type. This parameter can have one of the following values:

- **CDocTemplate::windowTitle** Name that appears in the application window's title bar (for example, "Microsoft Excel"). Present only in the document template for SDI applications.
- **CDocTemplate::docName** Root for the default document name (for example, "Sheet"). This root, plus a number, is used for the default name of a new document of this type whenever the user chooses the New command from the File menu (for example, "Sheet1" or "Sheet2"). If not specified, "Untitled" is used as the default.
- **CDocTemplate::fileNewName** Name of this document type. If the application supports more than one type of document, this string is displayed in the File New dialog box (for example, "Worksheet"). If not specified, the document type is inaccessible using the File New command.
- **CDocTemplate::filterName** Description of the document type and a wildcard filter matching documents of this type. This string is displayed in the List Files Of Type drop-down list in the File Open dialog box (for example, "Worksheets (*.xls)"). If not specified, the document type is inaccessible using the File Open command.
- **CDocTemplate::filterExt** Extension for documents of this type (for example, ".xls"). If not specified, the document type is inaccessible using the File Open command.
- **CDocTemplate::regFileTypeId** Identifier for the document type to be stored in the registration database maintained by Windows. This string is for internal use only (for example, "ExcelWorksheet"). If not specified, the document type cannot be registered with the Windows File Manager.
- **CDocTemplate::regFileTypeName** Name of the document type to be stored in the registration database. This string may be displayed in dialog boxes of applications that access the registration database (for example, "Microsoft Excel Worksheet").

Return Value

Nonzero if the specified substring was found; otherwise 0.

Remarks

Call this function to retrieve a specific substring describing the document type. The string containing these substrings is stored in the document template and is derived from a string in the resource file for the application. The framework calls this function to get the strings it needs for the application's user interface. If you have specified a filename extension for your application's documents, the framework also calls this function when adding an entry to the Windows registration database; this allows documents to be opened from the Windows File Manager.

Call this function only if you are deriving your own class from **CDocTemplate**.

See Also

Reference

CDocTemplate Class
Hierarchy Chart
CMultiDocTemplate::CMultiDocTemplate

CSingleDocTemplate::CSingleDocTemplate
CWinApp::RegisterShellFileTypes

Other Resources

CDocTemplate Members

CDocTemplate::GetFirstDocPosition

Retrieves the position of the first document associated with this template.

```
virtual POSITION GetFirstDocPosition( ) const = 0;
```

Return Value

A **POSITION** value that can be used to iterate through the list of documents associated with this document template; or **NULL** if the list is empty.

Remarks

Use this function to get the position of the first document in the list of documents associated with this template. Use the **POSITION** value as an argument to CDocTemplate::GetNextDoc to iterate through the list of documents associated with the template.

CSingleDocTemplate and CMultiDocTemplate both override this pure virtual function. Any class you derive from **CDocTemplate** must also override this function.

See Also

Reference

CDocTemplate Class
Hierarchy Chart
CDocTemplate::GetNextDoc
CSingleDocTemplate Class
CMultiDocTemplate Class

Other Resources

CDocTemplate Members

CDocTemplate::GetNextDoc

Retrieves the list element identified by *rPos*, then sets *rPos* to the **POSITION** value of the next entry in the list.

```
virtual CDocument* GetNextDoc(
   POSITION& rPos
) const = 0;
```

Parameters

rPos

A reference to a **POSITION** value returned by a previous call to GetFirstDocPosition or **GetNextDoc**.

Return Value

A pointer to the next document in the list of documents associated with this template.

Remarks

If the retrieved element is the last in the list, then the new value of *rPos* is set to **NULL**.

You can use **GetNextDoc** in a forward iteration loop if you establish the initial position with a call to GetFirstDocPosition.

You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

See Also

Reference

CDocTemplate Class
Hierarchy Chart
CDocTemplate::GetFirstDocPosition

Other Resources

CDocTemplate Members

CDocTemplate::InitialUpdateFrame

Initializes the frame window, and optionally makes it visible.

```
virtual void InitialUpdateFrame(
   CFrameWnd* pFrame,
   CDocument* pDoc,
   BOOL bMakeVisible = TRUE
);
```

Parameters

pFrame

The frame window that needs the initial update.

pDoc

The document to which the frame is associated. Can be **NULL**.

bMakeVisible

Indicates whether the frame should become visible and active.

Remarks

Call **IntitialUpdateFrame** after creating a new frame with **CreateNewFrame**. Calling this function causes the views in that frame window to receive their **OnInitialUpdate** calls. Also, if there was not previously an active view, the primary view of the frame window is made active; the primary view is a view with a child ID of **AFX_IDW_PANE_FIRST**. Finally, the frame window is made visible if *bMakeVisible* is non-zero. If *bMakeVisible* is zero, the current focus and visible state of the frame window will remain unchanged.

It is not necessary to call this function when using the framework's implementation of File New and File Open.

See Also

Reference

CDocTemplate Class
Hierarchy Chart
CView::OnInitialUpdate
CFrameWnd::SetActiveView
CDocTemplate::CreateNewFrame

Other Resources

CDocTemplate Members

CDocTemplate::LoadTemplate

Loads the resources for a given **CDocTemplate** or derived class.

```
virtual void LoadTemplate( );
```

Remarks

This member function is called by the framework to load the resources for a given **CDocTemplate** or derived class. Normally it is called during construction, except when the template is being constructed globally. In that case, the call to **LoadTemplate** is delayed until CWinApp::AddDocTemplate is called.

See Also

Reference

CDocTemplate Class
Hierarchy Chart
CWinApp::AddDocTemplate

Other Resources

CDocTemplate Members

CDocTemplate::MatchDocType

Determines the degree of confidence in the match between a document type and this template.

```
virtual Confidence MatchDocType(
   LPCTSTR lpszPathName,
   CDocument*& rpDocMatch
);
```

Parameters

lpszPathName

Pathname of the file whose type is to be determined.

rpDocMatch

Pointer to a document that is assigned the matching document, if the file specified by *lpszPathName* is already open.

Return Value

A value from the **Confidence** enumeration, which is defined as follows:

```
enum Confidence
{
        noAttempt,
        maybeAttemptForeign,
        maybeAttemptNative,
        yesAttemptForeign,
        yesAttemptNative,
        yesAlreadyOpen
};
```

Remarks

Use this function to determine the type of document template to use for opening a file. If your application supports multiple file types, for example, you can use this function to determine which of the available document templates is appropriate for a given file by calling **MatchDocType** for each template in turn, and choosing a template according to the confidence value returned.

If the file specified by *lpszPathName* is already open, this function returns **CDocTemplate::yesAlreadyOpen** and copies the file's **CDocument** object into the object at *rpDocMatch*.

If the file is not open but the extension in *lpszPathName* matches the extension specified by **CDocTemplate::filterExt**, this function returns **CDocTemplate::yesAttemptNative** and sets *rpDocMatch* to **NULL**. For more information on **CDocTemplate::filterExt**, see CDocTemplate::GetDocString.

If neither case is true, the function returns **CDocTemplate::yesAttemptForeign**.

The default implementation does not return **CDocTemplate::maybeAttemptForeign** or **CDocTemplate::maybeAttemptNative**. Override this function to implement type-matching logic appropriate to your application, perhaps using these two values from the **Confidence** enumeration.

See Also

Reference

CDocTemplate Class
Hierarchy Chart
CDocTemplate::GetDocString

Other Resources

CDocTemplate Members

CDocTemplate::OpenDocumentFile

Opens a file specified by a pathname.

```
virtual CDocument* OpenDocumentFile(
   LPCTSTR lpszPathName,
   BOOL bMakeVisible = TRUE
) = 0;
```

Parameters

lpszPathName

Pointer to the pathname of the file containing the document to be opened.

bMakeVisible

Determines whether the window containing the document is to be made visible.

Return Value

A pointer to the document whose file is named by *lpszPathName*; **NULL** if unsuccessful.

Remarks

Opens the file whose pathname is specified by *lpszPathName*. If *lpszPathName* is **NULL**, a new file, containing a document of the type associated with this template, is created.

See Also

Reference

CDocTemplate Class
Hierarchy Chart
CDocTemplate::CloseAllDocuments

Other Resources

CDocTemplate Members

CDocTemplate::RemoveDocument

Removes the document pointed to by *pDoc* from the list of documents associated with this template.

```
virtual void RemoveDocument(
   CDocument* pDoc
);
```

Parameters

pDoc

Pointer to the document to be removed.

Remarks

The derived classes **CMultiDocTemplate** and **CSingleDocTemplate** override this function. If you derive your own document-template class from **CDocTemplate**, your derived class must override this function.

See Also

Reference

CDocTemplate Class
Hierarchy Chart
CDocTemplate::AddDocument
CMultiDocTemplate Class
CSingleDocTemplate Class

Other Resources

CDocTemplate Members

CDocTemplate::SaveAllModified

Saves all documents that have been modified.

```
virtual BOOL SaveAllModified( );
```

Return Value

Non-zero if successful; otherwise 0.

See Also

Reference

CDocTemplate Class
Hierarchy Chart
CDocTemplate::OpenDocumentFile
CDocTemplate::CloseAllDocuments

Other Resources

CDocTemplate Members

CDocTemplate::SetContainerInfo

Determines the resources for OLE containers when editing an in-place OLE item.

```
void SetContainerInfo(
   UINT nIDOleInPlaceContainer
);
```

Parameters

nIDOleInPlaceContainer

The ID of the resources used when an embedded object is activated.

Remarks

Call this function to set the resources to be used when an OLE object is in-place activated. These resources may include menus and accelerator tables. This function is usually called in the CWinApp::InitInstance function of your application.

The menu associated with *nIDOleInPlaceContainer* contains separators that allow the menu of the activated in-place item to merge with the menu of the container application. For more information about merging server and container menus, see the article Menus and Resources (OLE).

See Also

Reference

CDocTemplate Class
Hierarchy Chart
CDocTemplate::SetServerInfo
CWinApp::InitInstance
CMultiDocTemplate::CMultiDocTemplate

Other Resources

CDocTemplate Members

CDocTemplate::SetDefaultTitle

Call this function to load the document's default title and display it in the document's title bar.

```
virtual void SetDefaultTitle(
   CDocument* pDocument
) = 0;
```

Parameters

pDocument

Pointer to the document whose title is to be set.

Remarks

For information on the default title, see the description of **CDocTemplate::docName** in CDocTemplate::GetDocString.

See Also

Reference

CDocTemplate Class
Hierarchy Chart
CDocTemplate::GetDocString

Other Resources

CDocTemplate Members

CDocTemplate::SetServerInfo

Determines the resources and classes when the server document is embedded or edited in-place.

```
void SetServerInfo(
   UINT nIDOleEmbedding,
   UINT nIDOleInPlaceServer = 0,
   CRuntimeClass* pOleFrameClass = NULL,
   CRuntimeClass* pOleViewClass = NULL
);
```

Parameters

nIDOleEmbedding

The ID of the resources used when an embedded object is opened in a separate window.

nIDOleInPlaceServer

The ID of the resources used when an embedded object is activated in-place.

pOleFrameClass

Pointer to a CRuntimeClass structure containing class information for the frame window object created when in-place activation occurs.

pOleViewClass

Pointer to a **CRuntimeClass** structure containing class information for the view object created when in-place activation occurs.

Remarks

Call this member function to identify resources that will be used by the server application when the user requests activation of an embedded object. These resources consist of menus and accelerator tables. This function is usually called in the **InitInstance** of your application.

The menu associated with *nIDOleInPlaceServer* contains separators that allow the server menu to merge with the menu of the container. For more information about merging server and container menus, see the article Menus and Resources (OLE).

See Also

Reference

CDocTemplate Class
Hierarchy Chart
CMultiDocTemplate::CMultiDocTemplate
CDocTemplate::SetContainerInfo
CWinApp::InitInstance

Other Resources

CDocTemplate Members

CDocument Class

Provides the basic functionality for user-defined document classes.

```
class CDocument : public CCmdTarget
```

Remarks

A document represents the unit of data that the user typically opens with the File Open command and saves with the File Save command.

CDocument supports standard operations such as creating a document, loading it, and saving it. The framework manipulates documents using the interface defined by **CDocument**.

An application can support more than one type of document; for example, an application might support both spreadsheets and text documents. Each type of document has an associated document template; the document template specifies what resources (for example, menu, icon, or accelerator table) are used for that type of document. Each document contains a pointer to its associated **CDocTemplate** object.

Users interact with a document through the CView object(s) associated with it. A view renders an image of the document in a frame window and interprets user input as operations on the document. A document can have multiple views associated with it. When the user opens a window on a document, the framework creates a view and attaches it to the document. The document template specifies what type of view and frame window are used to display each type of document.

Documents are part of the framework's standard command routing and consequently receive commands from standard user-interface components (such as the File Save menu item). A document receives commands forwarded by the active view. If the document doesn't handle a given command, it forwards the command to the document template that manages it.

When a document's data is modified, each of its views must reflect those modifications. **CDocument** provides the UpdateAllViews member function for you to notify the views of such changes, so the views can repaint themselves as necessary. The framework also prompts the user to save a modified file before closing it.

To implement documents in a typical application, you must do the following:

- Derive a class from **CDocument** for each type of document.
- Add member variables to store each document's data.
- Implement member functions for reading and modifying the document's data. The document's views are the most important users of these member functions.
- Override the CObject::Serialize member function in your document class to write and read the document's data to and from disk.

CDocument supports sending your document via mail if mail support (MAPI) is present. See the articles MAPI and MAPI Support in MFC.

For more information on **CDocument**, see Serialization, Document/View Architecture Topics, and Document/View Creation.

Requirements

Header: afxwin.h

See Also

Tasks

MDIDOCVW Sample: Demonstrates MDI Using Doc/View Architecture
SNAPVW Sample: Uses Property Pages in a Form View Application
NPP Sample: Demonstrates the Windows Messaging API (MAPI)

Reference

CCmdTarget Class
Hierarchy Chart
CCmdTarget Class
CView Class
CDocTemplate Class

Other Resources

CDocument Members

Base Class Members

CObject Members

CCmdTarget Members

Construction

CDocument	Constructs a **CDocument** object.

Operations

AddView	Attaches a view to the document.
GetDocTemplate	Returns a pointer to the document template that describes the type of the document.
GetFirstViewPosition	Returns the position of the first in the list of views; used to begin iteration.
GetNextView	Iterates through the list of views associated with the document.
GetPathName	Returns the path of the document's data file.
GetTitle	Returns the document's title.
IsModified	Indicates whether the document has been modified since it was last saved.
RemoveView	Detaches a view from the document.
SetModifiedFlag	Sets a flag indicating that you have modified the document since it was last saved.
SetPathName	Sets the path of the data file used by the document.
SetTitle	Sets the document's title.
UpdateAllViews	Notifies all views that document has been modified.

Overridables

CanCloseFrame	Advanced overridable; called before closing a frame window viewing this document.
DeleteContents	Called to perform cleanup of the document.
GetFile	Returns a pointer to the desired **CFile** object.
OnChangedViewList	Called after a view is added to or removed from the document.
OnCloseDocument	Called to close the document.
OnNewDocument	Called to create a new document.
OnOpenDocument	Called to open an existing document.
OnSaveDocument	Called to save the document to disk.
PreCloseFrame	Called before the frame window is closed.

ReleaseFile	Releases a file to make it available for use by other applications.
ReportSaveLoadException	Advanced overridable; called when an open or save operation cannot be completed because of an e xception.
SaveModified	Advanced overridable; called to ask the user whether the document should be saved.

Mail Functions

OnFileSendMail	Sends a mail message with the document attached.
OnUpdateFileSendMail	Enables the Send Mail command if mail support is present.

See Also

Reference

CDocument Class

Hierarchy Chart

CDocument Member Functions

For information about the member functions in **CDocument**, see CDocument Members.

CDocument::AddView

Call this function to attach a view to the document.

```
void AddView(
   CView* pView
);
```

Parameters

pView

Points to the view being added.

Remarks

This function adds the specified view to the list of views associated with the document; the function also sets the view's document pointer to this document. The framework calls this function when attaching a newly created view object to a document; this occurs in response to a File New, File Open, or New Window command or when a splitter window is split.

Call this function only if you are manually creating and attaching a view. Typically you will let the framework connect documents and views by defining a CDocTemplate object to associate a document class, view class, and frame window class.

Example

```
// The following example toggles two views in an SDI (single document
// interface) frame window. A design decision must be made as to
// whether to leave the inactive view connected to the document,
// such that the inactive view continues to receive OnUpdate
// notifications from the document. It is usually desirable to
// keep the inactive view continuously in sync with the document, even
// though it is inactive. However, doing so incurs a performance cost,
// as well as the programming cost of implementing OnUpdate hints.
// It may be less expensive, in terms of performance and/or programming,
// to re-sync the inactive view with the document only with it is
// reactivated. This example illustrates this latter approach, by
// reconnecting the newly active view and disconnecting the newly
// inactive view, via calls to CDocument::AddView and RemoveView.

BOOL CMainFrame::OnViewChange(UINT nCmdID)
// There is an ON_COMMAND_RANGE message map entry associated with
// OnViewChange:
// ON_COMMAND_RANGE( ID_VIEW_VIEW1, ID_VIEW_VIEW2, OnViewChange)
{
 CView* pViewAdd;
 CView* pViewRemove;
 CDocument* pDoc = GetActiveDocument();

 if((nCmdID == ID_VIEW_VIEW1) && (m_currentView == 1))
    return;
 if((nCmdID == ID_VIEW_VIEW2) && (m_currentView == 2))
   return;

 if (nCmdID == ID_VIEW_VIEW2)
 {
  if (m_pView2 == NULL)
  {
   m_pView1 = GetActiveView();
   m_pView2 = new CMyView2;

//Note that if OnSize has been overridden in CMyView2
//and GetDocument() is used in this override it can
//cause assertions and, if the assertions are ignored,
//cause access violation.

   m_pView2->Create(NULL, NULL, AFX_WS_DEFAULT_VIEW,
```

```
        rectDefault, this, AFX_IDW_PANE_FIRST + 1, NULL);
    }
     pViewAdd = m_pView2;
     pViewRemove = m_pView1;
     m_currentView= 2;
   }
   else
   {
    pViewAdd = m_pView1;
    pViewRemove = m_pView2;
    m_currentView= 1;
   }

  // Set the child i.d. of the active view to AFX_IDW_PANE_FIRST,
  // so that CFrameWnd::RecalcLayout will allocate to this
  // "first pane" that portion of   the frame window's client area
  // not allocated to control   bars.  Set the child i.d. of the
  // other view to anything other than AFX_IDW_PANE_FIRST; this
  // examples switches the child id's of the two views.

   int nSwitchChildID = pViewAdd->GetDlgCtrlID();
   pViewAdd->SetDlgCtrlID(AFX_IDW_PANE_FIRST);
   pViewRemove->SetDlgCtrlID(nSwitchChildID);

   // Show the newly active view and hide the inactive view.

   pViewAdd->ShowWindow(SW_SHOW);
   pViewRemove->ShowWindow(SW_HIDE);

   // Connect the newly active view to the document, and
   // disconnect the inactive view.
   pDoc->AddView(pViewAdd);
   pDoc->RemoveView(pViewRemove);

   SetActiveView(pViewAdd);
   RecalcLayout();
  }
```

See Also

Reference

CDocument Class
Hierarchy Chart
CDocTemplate Class
CDocument::GetFirstViewPosition
CDocument::GetNextView
CDocument::RemoveView
CView::GetDocument

Other Resources

CDocument Members

CDocument::CanCloseFrame

Called by the framework before a frame window displaying the document is closed.

```
virtual BOOL CanCloseFrame(
   CFrameWnd* pFrame
);
```

Parameters

pFrame

Points to the frame window of a view attached to the document.

Return Value

Nonzero if it is safe to close the frame window; otherwise 0.

Remarks

The default implementation checks if there are other frame windows displaying the document. If the specified frame window is the last one that displays the document, the function prompts the user to save the document if it has been modified. Override this function if you want to perform special processing when a frame window is closed. This is an advanced overridable.

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::SaveModified

Other Resources

CDocument Members

CDocument::CDocument

Constructs a **CDocument** object.

```
CDocument( );
```

Remarks

The framework handles document creation for you. Override the OnNewDocument member function to perform initialization on a per-document basis; this is particularly important in single document interface (SDI) applications.

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::OnNewDocument
CDocument::OnOpenDocument

Other Resources

CDocument Members

CDocument::DeleteContents

Called by the framework to delete the document's data without destroying the **CDocument** object itself.

```
virtual void DeleteContents( );
```

Remarks

It is called just before the document is to be destroyed. It is also called to ensure that a document is empty before it is reused. This is particularly important for an SDI application, which uses only one document; the document is reused whenever the user creates or opens another document. Call this function to implement an "Edit Clear All" or similar command that deletes all of the document's data. The default implementation of this function does nothing. Override this function to delete the data in your document.

Example

```
// This example is the handler for an Edit Clear All command.

void CMyDoc::OnEditClearAll()
{
   DeleteContents();
   UpdateAllViews(NULL);
}

void CMyDoc::DeleteContents()
{
   // Re-initialize document data here.

}
```

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::OnCloseDocument
CDocument::OnNewDocument
CDocument::OnOpenDocument

Other Resources

CDocument Members

CDocument::GetDocTemplate

Call this function to get a pointer to the document template for this document type.

```
CDocTemplate* GetDocTemplate( ) const;
```

Return Value

A pointer to the document template for this document type, or **NULL** if the document is not managed by a document template.

Example

```
    // This example accesses the doc template object to construct
    // a default document name such as SHEET.XLS, where "sheet"
    // is the base document name and ".xls" is the file extension
    // for the document type.
    CString strDefaultDocName, strBaseName, strExt;
    CDocTemplate* pDocTemplate = GetDocTemplate();
    if (!pDocTemplate->GetDocString(strBaseName, CDocTemplate::docName)
       || !pDocTemplate->GetDocString(strExt, CDocTemplate::filterExt))
    {
       AfxThrowUserException(); // These doc template strings will
          // be available if you created the application using AppWizard
          // and specified the file extension as an option for
          // the document class produced by AppWizard.
    }
    strDefaultDocName = strBaseName + strExt;
```

See Also

Reference

CDocument Class

Hierarchy Chart

CDocTemplate Class

Other Resources

CDocument Members

CDocument::GetFile

Call this member function to get a pointer to a **CFile** object.

```
virtual CFile* GetFile(
   LPCTSTR lpszFileName,
   UINT nOpenFlags,
   CFileException* pError
);
```

Parameters

lpszFileName

A string that is the path to the desired file. The path may be relative or absolute.

pError

A pointer to an existing file-exception object that indicates the completion status of the operation.

nOpenFlags

Sharing and access mode. Specifies the action to take when opening the file. You can combine options listed in the CFile constructor CFile::CFile by using the bitwise OR (|) operator. One access permission and one share option are required; the **modeCreate** and **modeNoInherit** modes are optional.

Return Value

A pointer to a **CFile** object.

See Also

Reference

CDocument Class
Hierarchy Chart
CDocTemplate Class

Other Resources

CDocument Members

CDocument::GetFirstViewPosition

Call this function to get the position of the first view in the list of views associated with the document.

```
virtual POSITION GetFirstViewPosition( ) const;
```

Return Value

A **POSITION** value that can be used for iteration with the GetNextView member function.

Example

```
//To get the first view in the list of views:

POSITION pos = GetFirstViewPosition();
CView* pFirstView = GetNextView( pos );
// This example uses CDocument::GetFirstViewPosition
// and GetNextView to repaint each view.
void CMyDoc::OnRepaintAllViews()
{
   POSITION pos = GetFirstViewPosition();
   while (pos != NULL)
   {
      CView* pView = GetNextView(pos);
      pView->UpdateWindow();
   }
}

// An easier way to accomplish the same result is to call
// UpdateAllViews(NULL);
```

See Also

Reference

CDocument Class

Hierarchy Chart

CDocument::GetNextView

Other Resources

CDocument Members

CDocument::GetNextView

Call this function to iterate through all of the document's views.

```
virtual CView* GetNextView(
   POSITION& rPosition
) const;
```

Parameters

rPosition

A reference to a **POSITION** value returned by a previous call to the **GetNextView** or GetFirstViewPosition member functions. This value must not be **NULL**.

Return Value

A pointer to the view identified by *rPosition*.

Remarks

The function returns the view identified by *rPosition* and then sets *rPosition* to the **POSITION** value of the next view in the list. If the retrieved view is the last in the list, then *rPosition* is set to **NULL**.

Example

```
// This example uses CDocument::GetFirstViewPosition
// and GetNextView to repaint each view.
void CMyDoc::OnRepaintAllViews()
{
   POSITION pos = GetFirstViewPosition();
   while (pos != NULL)
   {
      CView* pView = GetNextView(pos);
      pView->UpdateWindow();
   }
}

// An easier way to accomplish the same result is to call
// UpdateAllViews(NULL);
```

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::AddView
CDocument::GetFirstViewPosition
CDocument::RemoveView
CDocument::UpdateAllViews

Other Resources

CDocument Members

CDocument::GetPathName

Call this function to get the fully qualified path of the document's disk file.

```
const CString& GetPathName( ) const;
```

Return Value

The document's fully qualified path. This string is empty if the document has not been saved or does not have a disk file associated with it.

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::SetPathName

Other Resources

CDocument Members

CDocument::GetTitle

Call this function to get the document's title, which is usually derived from the document's filename.

```
const CString& GetTitle( ) const;
```

Return Value

The document's title.

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::SetTitle

Other Resources

CDocument Members

CDocument::IsModified

Call this function to determine whether the document has been modified since it was last saved.

```
virtual BOOL IsModified( );
```

Return Value

Nonzero if the document has been modified since it was last saved; otherwise 0.

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::SetModifiedFlag
CDocument::SaveModified

Other Resources

CDocument Members

CDocument::OnChangedViewList

Called by the framework after a view is added to or removed from the document.

```
virtual void OnChangedViewList( );
```

Remarks

The default implementation of this function checks whether the last view is being removed and, if so, deletes the document. Override this function if you want to perform special processing when the framework adds or removes a view. For example, if you want a document to remain open even when there are no views attached to it, override this function.

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::AddView
CDocument::RemoveView

Other Resources

CDocument Members

CDocument::OnCloseDocument

Called by the framework when the document is closed, typically as part of the File Close command.

```
virtual void OnCloseDocument( );
```

Remarks

The default implementation of this function destroys all of the frames used for viewing the document, closes the view, cleans up the document's contents, and then calls the DeleteContents member function to delete the document's data.

Override this function if you want to perform special cleanup processing when the framework closes a document. For example, if the document represents a record in a database, you may want to override this function to close the database. You should call the base class version of this function from your override.

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::DeleteContents
CDocument::OnNewDocument
CDocument::OnOpenDocument

Other Resources

CDocument Members

CDocument::OnFileSendMail

Sends a message via the resident mail host (if any) with the document as an attachment.

```
void OnFileSendMail( );
```

Remarks

OnFileSendMail calls OnSaveDocument to serialize (save) untitled and modified documents to a temporary file, which is then sent via electronic mail. If the document has not been modified, a temporary file is not needed; the original is sent. **OnFileSendMail** loads MAPI32.DLL if it has not already been loaded.

A special implementation of **OnFileSendMail** for COleDocument handles compound files correctly.

CDocument supports sending your document via mail if mail support (MAPI) is present. See the articles MAPI Topics and MAPI Support in MFC.

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::OnUpdateFileSendMail
COleDocument::OnFileSendMail
CDocument::OnSaveDocument

Other Resources

CDocument Members

CDocument::OnNewDocument

Called by the framework as part of the File New command.

```
virtual BOOL OnNewDocument( );
```

Return Value

Nonzero if the document was successfully initialized; otherwise 0.

Remarks

The default implementation of this function calls the DeleteContents member function to ensure that the document is empty and then marks the new document as clean. Override this function to initialize the data structure for a new document. You should call the base class version of this function from your override.

If the user chooses the File New command in an SDI application, the framework uses this function to reinitialize the existing document, rather than creating a new one. If the user chooses File New in a multiple document interface (MDI) application, the framework creates a new document each time and then calls this function to initialize it. You must place your initialization code in this function instead of in the constructor for the File New command to be effective in SDI applications.

Note that there are cases where **OnNewDocument** is called twice. This occurs when the document is embedded as an ActiveX Document Server. The function is first called by the **CreateInstance** method (exposed by the **COleObjectFactory**-derived class) and a second time by the **InitNew** method (exposed by the **COleServerDoc**-derived class).

Example

```
// The following examples illustrate alternative methods of
// initializing a document object.

// Method 1: In an MDI application, the simplest place to do
// initialization is in the document constructor.  The framework
// always creates a new document object for File New or File Open.

CMyDoc::CMyDoc()
{
   // Do initialization of MDI document here.
   // ...
}

// Method 2: In an SDI or MDI application, do all initialization
// in an override of OnNewDocument, if you are certain that
// the initialization is effectively saved upon File Save
// and fully restored upon File Open, via serialization.

BOOL CMyDoc::OnNewDocument()
{
   if (!CDocument::OnNewDocument())
      return FALSE;

   // Do initialization of new document here.

   return TRUE;
}

// Method 3: If the initialization of your document is not
// effectively saved and restored by serialization (during File Save
// and File Open), then implement the initialization in single
// function (named InitMyDocument in this example).  Call the
// shared initialization function from overrides of both
// OnNewDocument and OnOpenDocument.

BOOL CMyDoc::OnNewDocument()
{
   if (!CDocument::OnNewDocument())
```

```
        return FALSE;

    InitMyDocument(); // call your shared initialization function

    // If your new document object requires additional initialization
    // not necessary when the document is deserialized via File Open,
    // then perform that additional initialization here.

    return TRUE;
}
```

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::CDocument
CDocument::DeleteContents
CDocument::OnCloseDocument
CDocument::OnOpenDocument
CDocument::OnSaveDocument

Other Resources

CDocument Members

CDocument::OnOpenDocument

Called by the framework as part of the File Open command.

```
virtual BOOL OnOpenDocument(
   LPCTSTR lpszPathName
);
```

Parameters

lpszPathName

Points to the path of the document to be opened.

Return Value

Nonzero if the document was successfully loaded; otherwise 0.

Remarks

The default implementation of this function opens the specified file, calls the DeleteContents member function to ensure that the document is empty, calls CObject::Serialize to read the file's contents, and then marks the document as clean. Override this function if you want to use something other than the archive mechanism or the file mechanism. For example, you might write an application where documents represent records in a database rather than separate files.

If the user chooses the File Open command in an SDI application, the framework uses this function to reinitialize the existing **CDocument** object, rather than creating a new one. If the user chooses File Open in an MDI application, the framework constructs a new **CDocument** object each time and then calls this function to initialize it. You must place your initialization code in this function instead of in the constructor for the File Open command to be effective in SDI applications.

Example

```
// The following examples illustrate alternative methods of
// initializing a document object.

// Method 1: In an MDI application, the simplest place to do
// initialization is in the document constructor.  The framework
// always creates a new document object for File New or File Open.

CMyDoc::CMyDoc()
{
   // Do initialization of MDI document here.
   // ...
}

// Method 2: In an SDI or MDI application, do all initialization
// in an override of OnNewDocument, if you are certain that
// the initialization is effectively saved upon File Save
// and fully restored upon File Open, via serialization.

BOOL CMyDoc::OnNewDocument()
{
   if (!CDocument::OnNewDocument())
      return FALSE;

   // Do initialization of new document here.

   return TRUE;
}

// Method 3: If the initialization of your document is not
// effectively saved and restored by serialization (during File Save
// and File Open), then implement the initialization in single
// function (named InitMyDocument in this example).  Call the
// shared initialization function from overrides of both
// OnNewDocument and OnOpenDocument.
```

```
BOOL CMyDoc::OnNewDocument()
{
   if (!CDocument::OnNewDocument())
      return FALSE;

   InitMyDocument(); // call your shared initialization function

   // If your new document object requires additional initialization
   // not necessary when the document is deserialized via File Open,
   // then perform that additional initialization here.

   return TRUE;
}
// Additional example of OnOpenDocument()
BOOL CMyDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
   if (!CDocument::OnOpenDocument(lpszPathName))
      return FALSE;

   // TODO: Add your specialized creation code here
   InitMyDocument(); // call your shared initialization function

   return TRUE;
}
```

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::DeleteContents
CDocument::OnCloseDocument
CDocument::OnNewDocument
CDocument::OnSaveDocument
CDocument::ReportSaveLoadException
CObject::Serialize

Other Resources

CDocument Members

CDocument::OnSaveDocument

Called by the framework as part of the File Save or File Save As command.

```
virtual BOOL OnSaveDocument(
   LPCTSTR lpszPathName
);
```

Parameters

lpszPathName

Points to the fully qualified path to which the file should be saved.

Return Value

Nonzero if the document was successfully saved; otherwise 0.

Remarks

The default implementation of this function opens the specified file, calls CObject::Serialize to write the document's data to the file, and then marks the document as clean. Override this function if you want to perform special processing when the framework saves a document. For example, you might write an application where documents represent records in a database rather than separate files.

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::OnCloseDocument
CDocument::OnNewDocument
CDocument::OnOpenDocument
CDocument::ReportSaveLoadException
CObject::Serialize

Other Resources

CDocument Members

CDocument::OnUpdateFileSendMail

Enables the **ID_FILE_SEND_MAIL** command if mail support (MAPI) is present.

```
void OnUpdateFileSendMail(
   CCmdUI* pCmdUI
);
```

Parameters

pCmdUI

A pointer to the CCmdUI object associated with the **ID_FILE_SEND_MAIL** command.

Remarks

Otherwise the function removes the **ID_FILE_SEND_MAIL** command from the menu, including separators above or below the menu item as appropriate. MAPI is enabled if MAPI32.DLL is present in the path and, in the [Mail] section of the WIN.INI file, MAPI=1. Most applications put this command on the File menu.

CDocument supports sending your document via mail if mail support (MAPI) is present. See the articles MAPI Topics and MAPI Support in MFC.

See Also

Reference

CDocument Class

Hierarchy Chart

CDocument::OnFileSendMail

Other Resources

CDocument Members

CDocument::PreCloseFrame

This member function is called by the framework before the frame window is destroyed.

```
virtual void PreCloseFrame(
   CFrameWnd* pFrame
);
```

Parameters

pFrame

Pointer to the CFrameWnd that holds the associated **CDocument** object.

Remarks

It can be overridden to provide custom cleanup, but be sure to call the base class as well.

The default of **PreCloseFrame** does nothing in **CDocument**. The **CDocument**-derived classes COleDocument and CRichEditDoc use this member function.

See Also

Reference

CDocument Class

Hierarchy Chart

Other Resources

CDocument Members

CDocument::ReleaseFile

This member function is called by the framework to release a file, making it available for use by other applications.

```
virtual void ReleaseFile(
   CFile* pFile,
   BOOL bAbort
);
```

Parameters

pFile

A pointer to the CFile object to be released.

bAbort

Specifies whether the file is to be released by using either **CFile::Close** or **CFile::Abort**. **FALSE** if the file is to be released using CFile::Close; **TRUE** if the file is to be released using CFile::Abort.

Remarks

If *bAbort* is **TRUE**, **ReleaseFile** calls **CFile::Abort**, and the file is released. **CFile::Abort** will not throw an exception.

If *bAbort* is **FALSE**, **ReleaseFile** calls **CFile::Close** and the file is released.

Override this member function to require an action by the user before the file is released.

See Also

Reference

CDocument Class
Hierarchy Chart
CDocTemplate Class
CFile::Close
CFile::Abort

Other Resources

CDocument Members

CDocument::RemoveView

Call this function to detach a view from a document.

```
void RemoveView(
   CView* pView
);
```

Parameters

pView

Points to the view being removed.

Remarks

This function removes the specified view from the list of views associated with the document; it also sets the view's document pointer to **NULL**. This function is called by the framework when a frame window is closed or a pane of a splitter window is closed.

Call this function only if you are manually detaching a view. Typically you will let the framework detach documents and views by defining a CDocTemplate object to associate a document class, view class, and frame window class.

See the example at AddView for a sample implementation.

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::AddView
CDocument::GetFirstViewPosition
CDocument::GetNextView

Other Resources

CDocument Members

CDocument::ReportSaveLoadException

Called if an exception is thrown (typically a CFileException or CArchiveException) while saving or loading the document.

```
virtual void ReportSaveLoadException(
   LPCTSTR lpszPathName,
   CException* e,
   BOOL bSaving,
   UINT nIDPDefault
);
```

Parameters

lpszPathName

Points to name of document that was being saved or loaded.

e

Points to the exception that was thrown. May be **NULL**.

bSaving

Flag indicating what operation was in progress; nonzero if the document was being saved, 0 if the document was being loaded.

nIDPDefault

Identifier of the error message to be displayed if the function does not specify a more specific one.

Remarks

The default implementation examines the exception object and looks for an error message that specifically describes the cause. If a specific message is not found or if *e* is **NULL**, the general message specified by the *nIDPDefault* parameter is used. The function then displays a message box containing the error message. Override this function if you want to provide additional, customized failure messages. This is an advanced overridable.

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::OnOpenDocument
CDocument::OnSaveDocument
CFileException Class
CArchiveException Class

Other Resources

CDocument Members

CDocument::SaveModified

Called by the framework before a modified document is to be closed.

```
virtual BOOL SaveModified( );
```

Return Value

Nonzero if it is safe to continue and close the document; 0 if the document should not be closed.

Remarks

The default implementation of this function displays a message box asking the user whether to save the changes to the document, if any have been made. Override this function if your program requires a different prompting procedure. This is an advanced overridable.

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::CanCloseFrame
CDocument::IsModified
CDocument::OnNewDocument
CDocument::OnOpenDocument
CDocument::OnSaveDocument

Other Resources

CDocument Members

CDocument::SetModifiedFlag

Call this function after you have made any modifications to the document.

```
virtual void SetModifiedFlag(
   BOOL bModified = TRUE
);
```

Parameters

bModified

Flag indicating whether the document has been modified.

Remarks

By calling this function consistently, you ensure that the framework prompts the user to save changes before closing a document. Typically you should use the default value of **TRUE** for the *bModified* parameter. To mark a document as clean (unmodified), call this function with a value of **FALSE**.

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::IsModified
CDocument::SaveModified

Other Resources

CDocument Members

CDocument::SetPathName

Call this function to specify the fully qualified path of the document's disk file.

```
virtual void SetPathName(
   LPCTSTR lpszPathName,
   BOOL bAddToMRU = TRUE
);
```

Parameters

lpszPathName

Points to the string to be used as the path for the document.

bAddToMRU

Determines whether the filename is added to the most recently used (MRU) file list. If **TRUE,** the filename is added; if **FALSE**, it is not added.

Remarks

Depending on the value of *bAddToMRU* the path is added, or not added, to the MRU list maintained by the application. Note that some documents are not associated with a disk file. Call this function only if you are overriding the default implementation for opening and saving files used by the framework.

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::GetPathName
CWinApp::AddToRecentFileList

Other Resources

CDocument Members

CDocument::SetTitle

Call this function to specify the document's title (the string displayed in the title bar of a frame window).

```
virtual void SetTitle(
   LPCTSTR lpszTitle
);
```

Parameters

lpszTitle

Points to the string to be used as the document's title.

Remarks

Calling this function updates the titles of all frame windows that display the document.

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::GetTitle

Other Resources

CDocument Members

CDocument::UpdateAllViews

Call this function after the document has been modified.

```
void UpdateAllViews(
   CView* pSender,
   LPARAM lHint = 0L,
   CObject* pHint = NULL
);
```

Parameters

pSender

Points to the view that modified the document, or **NULL** if all views are to be updated.

lHint

Contains information about the modification.

pHint

Points to an object storing information about the modification.

Remarks

You should call this function after you call the SetModifiedFlag member function. This function informs each view attached to the document, except for the view specified by *pSender*, that the document has been modified. You typically call this function from your view class after the user has changed the document through a view.

This function calls the CView::OnUpdate member function for each of the document's views except the sending view, passing *pHint* and *lHint*. Use these parameters to pass information to the views about the modifications made to the document. You can encode information using *lHint* and/or you can define a CObject-derived class to store information about the modifications and pass an object of that class using *pHint*. Override the **CView::OnUpdate** member function in your CView-derived class to optimize the updating of the view's display based on the information passed.

Example

```
void CMyDoc::OnRepaintAllViews()
{
   UpdateAllViews(NULL);
}
```

See Also

Reference

CDocument Class
Hierarchy Chart
CDocument::SetModifiedFlag
CDocument::GetFirstViewPosition
CDocument::GetNextView
CView::OnUpdate

Other Resources

CDocument Members

CDragListBox Class

In addition to providing the functionality of a Windows list box, the **CDragListBox** class allows the user to move list box items, such as filenames, within the list box.

```
class CDragListBox : public CListBox
```

Remarks

List boxes with this capability allow users to order the items in a list in whatever manner is most useful to them. By default, the list box will move the item to the new location in the list. However, **CDragListBox** objects can be customized to copy items instead of moving them.

The list box control associated with the **CDragListBox** class must not have the **LBS_SORT** or the **LBS_MULTIPLESELECT** style. For a description of list box styles, see List-Box Styles.

To use a drag list box in an existing dialog box of your application, add a list box control to your dialog template using the dialog editor and then assign a member variable (of Category `Control` and Variable Type `CDragListBox`) corresponding to the list box control in your dialog template.

For more information on assigning controls to member variables, see
Shortcut for Defining Member Variables for Dialog Controls.

Requirements

Header: afxcmn.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also
Tasks
TSTCON Sample: ActiveX Control Test Container
Reference
CListBox Class
Hierarchy Chart
CListBox Class
Other Resources
CDragListBox Members

CDragListBox Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CListBox Members

Attributes

ItemFromPt	Returns the coordinates of the item being dragged.

Construction

CDragListBox	Constructs a **CDragListBox** object.

Operations

DrawInsert	Draws the insertion guide of the drag list box.

Overridables

BeginDrag	Called by the framework when a drag operation starts.
CancelDrag	Called by the framework when a drag operation has been canceled.
Dragging	Called by the framework during a drag operation.
Dropped	Called by the framework after the item has been dropped.

See Also

Reference

CDragListBox Class

Hierarchy Chart

CDragListBox Member Functions

For information about the member functions in **CDragListBox**, see CDragListBox Members.

CDragListBox::BeginDrag

Called by the framework when an event occurs that could begin a drag operation, such as pressing the left mouse button.

```
virtual BOOL BeginDrag(
   CPoint pt
);
```

Parameters

pt

A CPoint object that contains the coordinates of the item being dragged.

Return Value

Nonzero if dragging is allowed, otherwise 0.

Remarks

Override this function if you want to control what happens when a drag operation begins. The default implementation captures the mouse and stays in drag mode until the user clicks the left or right mouse button or presses ESC, at which time the drag operation is canceled.

See Also

Reference

CDragListBox Class
Hierarchy Chart
CDragListBox::CancelDrag
CDragListBox::Dragging

Other Resources

CDragListBox Members

CDragListBox::CancelDrag

Called by the framework when a drag operation has been canceled.

```
virtual void CancelDrag(
   CPoint pt
);
```

Parameters

pt

A CPoint object that contains the coordinates of the item being dragged.

Remarks

Override this function to handle any special processing for your list box control.

See Also

Reference

CDragListBox Class
Hierarchy Chart
CDragListBox::BeginDrag
CDragListBox::Dragging

Other Resources

CDragListBox Members

CDragListBox::CDragListBox

Constructs a **CDragListBox** object.

```
CDragListBox( );
```

See Also

Reference

CDragListBox Class

Hierarchy Chart

CListBox::Create

Other Resources

CDragListBox Members

CDragListBox::Dragging

Called by the framework when a list box item is being dragged within the **CDragListBox** object.

```
virtual UINT Dragging(
   CPoint pt
);
```

Parameters

pt

A CPoint object that contains the x and y screen coordinates of the cursor.

Return Value

The resource ID of the cursor to be displayed. The following values are possible:

- **DL_COPYCURSOR** Indicates that the item will be copied.
- **DL_MOVECURSOR** Indicates that the item will be moved.
- **DL_STOPCURSOR** Indicates that the current drop target is not acceptable.

Remarks

The default behavior returns **DL_MOVECURSOR**. Override this function if you want to provide additional functionality.

See Also

Reference

CDragListBox Class
Hierarchy Chart
CDragListBox::BeginDrag
CDragListBox::CancelDrag

Other Resources

CDragListBox Members

CDragListBox::DrawInsert

Called by the framework to draw the insertion guide before the item with the indicated index.

```
virtual void DrawInsert(
   int nItem
);
```

Parameters

nItem

Zero-based index of the insertion point.

Remarks

A value of - 1 clears the insertion guide. Override this function to modify the appearance or behavior of the insertion guide.

See Also

Reference

CDragListBox Class

Hierarchy Chart

Other Resources

CDragListBox Members

CDragListBox::Dropped

Called by the framework when an item is dropped within a **CDragListBox** object.

```
virtual void Dropped(
   int nSrcIndex,
   CPoint pt
);
```

Parameters

nSrcIndex

Specifies the zero-based index of the dropped string.

pt

A CPoint object that contains the coordinates of the drop site.

Remarks

The default behavior copies the list box item and its data to the new location and then deletes the original item. Override this function to customize the default behavior, such as enabling copies of list box items to be dragged to other locations within the list.

See Also

Reference

CDragListBox Class
Hierarchy Chart
CDragListBox::BeginDrag

Other Resources

CDragListBox Members

CDragListBox::ItemFromPt

Call this function to retrieve the zero-based index of the list box item located at *pt*.

```
int ItemFromPt(
   CPoint pt,
   BOOL bAutoScroll = TRUE
) const;
```

Parameters

pt

A CPoint object containing the coordinates of a point within the list box.

bAutoScroll

Nonzero if scrolling is allowed, otherwise 0.

Return Value

Zero-based index of the drag list box item.

See Also

Reference

CDragListBox Class

Hierarchy Chart

Other Resources

CDragListBox Members

CDumpContext Class

Supports stream-oriented diagnostic output in the form of human-readable text.

```
class CDumpContext
```

Remarks

CDumpContext does not have a base class.

You can use afxDump, a predeclared **CDumpContext** object, for most of your dumping. The **afxDump** object is available only in the Debug version of the Microsoft Foundation Class Library.

Several of the memory diagnostic services use **afxDump** for their output.

Under the Windows environment, the output from the predefined **afxDump** object, conceptually similar to the **cerr** stream, is routed to the debugger via the Windows function **OutputDebugString**.

The **CDumpContext** class has an overloaded insertion (**<<**) operator for **CObject** pointers that dumps the object's data. If you need a custom dump format for a derived object, override CObject::Dump. Most Microsoft Foundation classes implement an overridden **Dump** member function.

Classes that are not derived from **CObject**, such as **CString**, **CTime**, and **CTimeSpan**, have their own overloaded **CDumpContext** insertion operators, as do often-used structures such as **CFileStatus**, **CPoint**, and **CRect**.

If you use the IMPLEMENT_DYNAMIC or IMPLEMENT_SERIAL macro in the implementation of your class, then **CObject::Dump** will print the name of your **CObject**-derived class. Otherwise, it will print `CObject`.

The **CDumpContext** class is available with both the Debug and Release versions of the library, but the **Dump** member function is defined only in the Debug version. Use **#ifdef _DEBUG** / **#endif** statements to bracket your diagnostic code, including your custom **Dump** member functions.

Before you create your own **CDumpContext** object, you must create a **CFile** object that serves as the dump destination.

For more information on **CDumpContext**, see Debugging MFC Applications.

#define _DEBUG

Requirements

Header: afx.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart
CFile Class
CObject Class

Other Resources

CDumpContext Members

CDumpContext Members

Construction

CDumpContext	Constructs a **CDumpContext** object.

Basic Input/Output

DumpAsHex	Dumps the indicated item in hexadecimal format.
Flush	Flushes any data in the dump context buffer.
HexDump	Dumps bytes contained in an array in hexadecimal format.
operator <<	Inserts variables and objects into the dump context.

Status

GetDepth	Gets an integer corresponding to the depth of the dump.
SetDepth	Sets the depth of the dump.

See Also

Reference

CDumpContext Class

Hierarchy Chart

CDumpContext Member Functions

For information about the member functions in **CDumpContext**, see CDumpContext Members.

CDumpContext::CDumpContext

Constructs an object of class **CDumpContext**.

```
CDumpContext(
   CFile* pFile = NULL
);
```

Parameters

pFile

A pointer to the **CFile** object that is the dump destination.

Remarks

The **afxDump** object is constructed automatically.

Do not write to the underlying **CFile** while the dump context is active; otherwise, you will interfere with the dump. Under the Windows environment, the output is routed to the debugger via the Windows function **OutputDebugString**.

Example

```
//example for CDumpContext::CDumpContext
CFile f;
if( !f.Open( "dump.txt", CFile::modeCreate | CFile::modeWrite ) ) {
   afxDump << "Unable to open file" << "\n";
   exit( 1 );
}
CDumpContext dc( &f );
```

See Also

Reference

CDumpContext Class

Hierarchy Chart

Other Resources

CDumpContext Members

CDumpContext::DumpAsHex

Dumps the specified type formatted as hexadecimal numbers.

```
CDumpContext& DumpAsHex(
   BYTE b
);
CDumpContext& DumpAsHex(
   DWORD dw
);
CDumpContext& DumpAsHex(
   int n
);
CDumpContext& DumpAsHex(
   LONG l
);
CDumpContext& DumpAsHex(
   LONGLONG n
);
CDumpContext& DumpAsHex(
   UINT u
);
CDumpContext& DumpAsHex(
   ULONGLONG n
);
CDumpContext& DumpAsHex(
   WORD w
);
```

Return Value

A reference to a **CDumpContext** object.

Remarks

Call this member function to dump the item of the specified type as a hexadecimal number. To dump an array, call CDumpContext::HexDump.

Example

```
//example for CDumpContext::DumpAsHex
afxDump.DumpAsHex(115);
```

The output from this function call is:

```
73
```

See Also

Reference

CDumpContext Class

Hierarchy Chart

CDumpContext::operator <<

Other Resources

CDumpContext Members

CDumpContext::Flush

Forces any data remaining in buffers to be written to the file attached to the dump context.

```
void Flush( );
```

Example

```
//example for CDumpContext::Flush
afxDump.Flush();
```

See Also

Reference

CDumpContext Class

Hierarchy Chart

Other Resources

CDumpContext Members

CDumpContext::GetDepth

Determines whether a deep or shallow dump is in process.

```
int GetDepth( ) const;
```

Return Value

The depth of the dump as set by **SetDepth**.

Example

See the example for SetDepth.

See Also

Reference

CDumpContext Class
Hierarchy Chart
CDumpContext::SetDepth

Other Resources

CDumpContext Members

CDumpContext::HexDump

Dumps an array of bytes formatted as hexadecimal numbers.

```
void HexDump(
   LPCTSTR lpszLine,
   BYTE* pby,
   int nBytes,
   int nWidth
);
```

Parameters

lpszLine

A string to output at the start of a new line.

pby

A pointer to a buffer containing the bytes to dump.

nBytes

The number of bytes to dump.

nWidth

Maximum number of bytes dumped per line (not the width of the output line).

Remarks

To dump a single, specific item type as a hexadecimal number, call CDumpContext::DumpAsHex.

Example

```
//example for CDumpContext::HexDump
char test[] = "This is a test of CDumpContext::HexDump\n";
afxDump.HexDump( ".", (BYTE*) test, sizeof test, 20 );
```

The output from this program is:

```
     . 54 68 69 73 20 69 73 20 61 20 74 65 73 74 20 6F 66 20 43 44
     . 75 6D 70 43 6F 6E 74 65 78 74 3A 3A 48 65 78 44 75 6D 70 0A
     . 00
```

See Also

Reference

CDumpContext Class

Hierarchy Chart

Other Resources

CDumpContext Members

CDumpContext::SetDepth

Sets the depth for the dump.

```
void SetDepth(
   int nNewDepth
);
```

Parameters

nNewDepth

The new depth value.

Remarks

If you are dumping a primitive type or simple **CObject** that contains no pointers to other objects, then a value of 0 is sufficient. A value greater than 0 specifies a deep dump where all objects are dumped recursively. For example, a deep dump of a collection will dump all elements of the collection. You may use other specific depth values in your derived classes.

Note
Circular references are not detected in deep dumps and can result in infinite loops.

Example

```
//example for CDumpContext::SetDepth
afxDump.SetDepth( 1 );  // Specifies deep dump
ASSERT( afxDump.GetDepth() == 1 );
```

See Also

Reference

CDumpContext Class
Hierarchy Chart
CObject::Dump

Other Resources

CDumpContext Members

CDumpContext Operators

For information about the operators in **CDumpContext**, see CDumpContext Members.

CDumpContext::operator <<

Outputs the specified data to the dump context.

```
CDumpContext& operator <<(
   const CObject* pOb
);
throw(
   CFileException*
);
CDumpContext& operator <<(
   const CObject& ob
);
throw(
   CFileException*
);
CDumpContext& operator <<(
   LPCTSTR lpsz
);
throw(
   CFileException*
);
CDumpContext& operator <<(
   const void* lp
);
throw(
   CFileException*
);
CDumpContext& operator <<(
   BYTE by
);
throw(
   CFileException*
);
CDumpContext& operator <<(
   WORD w
);
throw(
   CFileException*
);
CDumpContext& operator <<(
   DWORD dw
);
throw(
   CFileException*
);
CDumpContext& operator <<(
   int n
);
throw(
   CFileException*
);
CDumpContext& operator <<(
   double d
);
throw(
   CFileException*
);
CDumpContext& operator <<(
   float f
);
throw(
   CFileException*
);
```

```
CDumpContext& operator <<(
   LONG l
);
throw(
   CFileException*
);
CDumpContext& operator <<(
   UINT u
);
throw(
   CFileException*
);
CDumpContext& operator <<(
   LPCWSTR lpsz
);
throw(
   CFileException*
);
CDumpContext& operator <<(
   LPCSTR lpsz
);
throw(
   CFileException*
);
CDumpContext& operator <<(
   LONGLONG n
);
CDumpContext& operator <<(
   ULONGLONG n
);
CDumpContext& operator <<(
   HWND h
);
CDumpContext& operator <<(
HDC h
);
CDumpContext& operator <<(
HMENU h
);
CDumpContext& operator <<(
HACCEL h
);
CDumpContext& operator <<(
HFONT h
);
```

Return Value

A **CDumpContext** reference. Using the return value, you can write multiple insertions on a single line of source code.

Remarks

The insertion operator is overloaded for **CObject** pointers as well as for most primitive types. A pointer to character results in a dump of string contents; a pointer to **void** results in a hexadecimal dump of the address only. A **LONGLONG** results in a dump of a 64-bit signed integer; A **ULONGLONG** results in a dump of a 64-bit unsigned integer.

If you use the **IMPLEMENT_DYNAMIC** or **IMPLEMENT_SERIAL** macro in the implementation of your class, then the insertion operator, through **CObject::Dump**, will print the name of your **CObject**-derived class. Otherwise, it will print `CObject`. If you override the **Dump** function of the class, then you can provide a more meaningful output of the object's contents instead of a hexadecimal dump.

Example

```
//example for CDumpContext::operator <<
extern CObList li;
CString s = "test";
int i = 7;
long lo = 1000000000L;
LONGLONG lolo = 12345678901234i64;
```

```
afxDump << "list=" << &li << "string="
   << s << "int=" << i << "long=" << lo << "LONGLONG=" << lolo << "\n";
```

See Also

Reference

CDumpContext Class

Hierarchy Chart

Other Resources

CDumpContext Members

CDWordArray Class

Supports arrays of 32-bit doublewords.

```
class CDWordArray : public CObject
```

Remarks

The member functions of **CDWordArray** are similar to the member functions of class CObArray. Because of this similarity, you can use the **CObArray** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a function parameter or return value, substitute a **DWORD**.

```
CObject* CObArray::GetAt( int <nIndex> ) const;
```

for example, translates to

```
DWORD CDWordArray::GetAt( int <nIndex> ) const;
```

CDWordArray incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. If an array of doublewords is stored to an archive, either with the overloaded insertion (**<<**) operator or with the **Serialize** member function, each element is, in turn, serialized.

Note
Before using an array, use **SetSize** to establish its size and allocate memory for it. If you do not use **SetSize**, adding elements to your array causes it to be frequently reallocated and copied. Frequent reallocation and copying are inefficient and can fragment memory.

If you need debug output from individual elements in the array, you must set the depth of the **CDumpContext** object to 1 or greater.

For more information on using **CDWordArray**, see the article Collections.

Requirements

Header: afxcoll.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class
Hierarchy Chart
CObArray Class

Other Resources

CDWordArray Members

CDWordArray Members

Base Class Members

CObject Members

Construction

CDWordArray	Constructs an empty array for doublewords.

Bounds

GetCount	Gets the number of elements in this array.
GetSize	Gets the number of elements in this array.
IsEmpty	Determines if the array is empty.
GetUpperBound	Returns the largest valid index.
SetSize	Sets the number of elements to be contained in this array.

Operations

FreeExtra	Frees all unused memory above the current upper bound.
RemoveAll	Removes all the elements from this array.

Element Access

ElementAt	Returns a temporary reference to the doubleword within the array.
GetAt	Returns the value at a given index.
GetData	Allows access to elements in the array. Can be **NULL**.
SetAt	Sets the value for a given index; array not allowed to grow.

Growing the Array

Add	Adds an element to the end of the array; grows the array if necessary.
Append	Appends another array to the array; grows the array if necessary.
Copy	Copies another array to the array; grows the array if necessary.
SetAtGrow	Sets the value for a given index; grows the array if necessary.

Insertion/Removal

InsertAt	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt	Removes an element at a specific index.

Operators

operator []	Sets or gets the element at the specified index.

See Also

Reference

CDWordArray Class

Hierarchy Chart

CEdit Class

Provides the functionality of a Windows edit control.

```
class CEdit : public CWnd
```

Remarks

An edit control is a rectangular child window in which the user can enter text.

You can create an edit control either from a dialog template or directly in your code. In both cases, first call the constructor **CEdit** to construct the **CEdit** object, then call the Create member function to create the Windows edit control and attach it to the **CEdit** object.

Construction can be a one-step process in a class derived from **CEdit**. Write a constructor for the derived class and call **Create** from within the constructor.

CEdit inherits significant functionality from **CWnd**. To set and retrieve text from a **CEdit** object, use the **CWnd** member functions SetWindowText and GetWindowText, which set or get the entire contents of an edit control, even if it is a multiline control. Text lines in a multiline control are separated by '\r\n' character sequences. Also, if an edit control is multiline, get and set part of the control's text by calling the **CEdit** member functions GetLine, SetSel, GetSel, and ReplaceSel.

If you want to handle Windows notification messages sent by an edit control to its parent (usually a class derived from **CDialog**), add a message-map entry and message-handler member function to the parent class for each message.

Each message-map entry takes the following form:

ON_Notification**(** *id, memberFxn* **)**

where *id* specifies the child window ID of the edit control sending the notification, and *memberFxn* is the name of the parent member function you have written to handle the notification.

The parent's function prototype is as follows:

afx_msg void memberFxn**( );**

Following is a list of potential message-map entries and a description of the cases in which they would be sent to the parent:

- **ON_EN_CHANGE** The user has taken an action that may have altered text in an edit control. Unlike the **EN_UPDATE** notification message, this notification message is sent after Windows updates the display.
- **ON_EN_ERRSPACE** The edit control cannot allocate enough memory to meet a specific request.
- **ON_EN_HSCROLL** The user clicks an edit control's horizontal scroll bar. The parent window is notified before the screen is updated.
- **ON_EN_KILLFOCUS** The edit control loses the input focus.
- **ON_EN_MAXTEXT** The current insertion has exceeded the specified number of characters for the edit control and has been truncated. Also sent when an edit control does not have the **ES_AUTOHSCROLL** style and the number of characters to be inserted would exceed the width of the edit control. Also sent when an edit control does not have the **ES_AUTOVSCROLL** style and the total number of lines resulting from a text insertion would exceed the height of the edit control.
- **ON_EN_SETFOCUS** Sent when an edit control receives the input focus.
- **ON_EN_UPDATE** The edit control is about to display altered text. Sent after the control has formatted the text but before it screens the text so that the window size can be altered, if necessary.
- **ON_EN_VSCROLL** The user clicks an edit control's vertical scroll bar. The parent window is notified before the screen is updated.

If you create a **CEdit** object within a dialog box, the **CEdit** object is automatically destroyed when the user closes the dialog box.

If you create a **CEdit** object from a dialog resource using the dialog editor, the **CEdit** object is automatically destroyed when

the user closes the dialog box.

If you create a **CEdit** object within a window, you may also need to destroy it. If you create the **CEdit** object on the stack, it is destroyed automatically. If you create the **CEdit** object on the heap by using the **new** function, you must call **delete** on the object to destroy it when the user terminates the Windows edit control. If you allocate any memory in the **CEdit** object, override the **CEdit** destructor to dispose of the allocations.

To modify certain styles in an edit control (such as **ES_READONLY**) you must send specific messages to the control instead of using ModifyStyle. See Edit Control Styles in the *Platform SDK*.

For more information on **CEdit**, see:

- Controls
- Knowledge Base article Q259949 : INFO: SetCaretPos() Is Not Appropriate with CEdit or CRichEditCtrl Controls

Requirements

Header: afxwin.h

See Also

Tasks

CALCDRIV Sample: Demonstrates an Automation Client Application
CMNCTRL2 Sample: Demonstrates Common Control MFC Classes, Part 2

Reference

CWnd Class
Hierarchy Chart
CWnd Class
CButton Class
CComboBox Class
CListBox Class
CScrollBar Class
CStatic Class
CDialog Class

Other Resources

CEdit Members

CEdit Members

Base Class Members

Construction

Attributes

Operations

Clipboard Operations

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Construction

CEdit	Constructs a **CEdit** control object.
Create	Creates the Windows edit control and attaches it to the **CEdit** object.

CEdit Attributes

CanUndo	Determines whether an edit-control operation can be undone.
CharFromPos	Retrieves the line and character indices for the character closest to a specified position.
GetCueBanner	Retrieves the text that is displayed as the text cue, or tip, in an edit control when the control is empty and d oes not have focus.
GetFirstVisibleLine	Determines the topmost visible line in an edit control.
GetHandle	Retrieves a handle to the memory currently allocated for a multiple-line edit control.
GetLimitText	Gets the maximum amount of text this **CEdit** can contain.
GetLine	Retrieves a line of text from an edit control.
GetLineCount	Retrieves the number of lines in a multiple-line edit control.
GetMargins	Gets the left and right margins for this **CEdit**.
GetModify	Determines whether the contents of an edit control have been modified.
GetPasswordChar	Retrieves the password character displayed in an edit control when the user enters text.
GetRect	Gets the formatting rectangle of an edit control.
GetSel	Gets the starting and ending character positions of the current selection in an edit control.
PosFromChar	Retrieves the coordinates of the upper-left corner of a specified character index.
SetCueBanner	Sets the text that is displayed as the text cue, or tip, in an edit control when the control is empty and does n ot have focus.

SetHandle	Sets the handle to the local memory that will be used by a multiple-line edit control.
SetLimitText	Sets the maximum amount of text this **CEdit** can contain.
SetMargins	Sets the left and right margins for this **CEdit**.
SetModify	Sets or clears the modification flag for an edit control.

CEdit Operations

EmptyUndoBuffer	Resets (clears) the undo flag of an edit control.
FmtLines	Sets the inclusion of soft line-break characters on or off within a multiple-line edit control.
LimitText	Limits the length of the text that the user may enter into an edit control.
LineFromChar	Retrieves the line number of the line that contains the specified character index.
LineIndex	Retrieves the character index of a line within a multiple-line edit control.
LineLength	Retrieves the length of a line in an edit control.
LineScroll	Scrolls the text of a multiple-line edit control.
ReplaceSel	Replaces the current selection in an edit control with the specified text.
SetPasswordChar	Sets or removes a password character displayed in an edit control when the user enters text.
SetReadOnly	Sets the read-only state of an edit control.
SetRect	Sets the formatting rectangle of a multiple-line edit control and updates the control.
SetRectNP	Sets the formatting rectangle of a multiple-line edit control without redrawing the control window.
SetSel	Selects a range of characters in an edit control.
SetTabStops	Sets the tab stops in a multiple-line edit control.

CEdit Clipboard Operations

Clear	Deletes (clears) the current selection (if any) in the edit control.
Copy	Copies the current selection (if any) in the edit control to the Clipboard in **CF_TEXT** format.
Cut	Deletes (cuts) the current selection (if any) in the edit control and copies the deleted text to the Clipboard in **CF_TEXT** format.
Paste	Inserts the data from the Clipboard into the edit control at the current cursor position. Data is inserted only if the Clipboard contains data in **CF_TEXT** format.
Undo	Reverses the last edit-control operation.

See Also

Reference

CEdit Class

Hierarchy Chart

CEdit Member Functions

For information about the member functions in **CEdit**, see CEdit Members.

CEdit::CanUndo

Call this function to determine if the last edit operation can be undone.

```
BOOL CanUndo( ) const;
```

Return Value

Nonzero if the last edit operation can be undone by a call to the **Undo** member function; 0 if it cannot be undone.

Remarks

For more information, see EM_CANUNDO in the *Platform SDK*.

Example

See the example for CEdit::Undo.

See Also

Reference

CEdit Class
Hierarchy Chart
CEdit::Undo
CEdit::EmptyUndoBuffer

Other Resources

CEdit Members

CEdit::CEdit

Constructs a **CEdit** object.

```
CEdit( );
```

Remarks

Use Create to construct the Windows edit control.

Example

```
// Declare a local CEdit object.
CEdit myEdit;

// Declare a dynamic CEdit object.
CEdit* pmyEdit = new CEdit;
```

See Also

Reference

CEdit Class

Hierarchy Chart

CEdit::Create

Other Resources

CEdit Members

CEdit::CharFromPos

Call this function to retrieve the zero-based line and character indices of the character nearest the specified point in this **CEdit** control

```
int CharFromPos(
   CPoint pt
) const;
```

Parameters

pt

The coordinates of a point in the client area of this **CEdit** object.

Return Value

The character index in the low-order **WORD**, and the line index in the high-order **WORD**.

Remarks

Note
This member function is available beginning with Windows 95 and Windows NT 4.0.

For more information, see EM_CHARFROMPOS in the *Platform SDK*.

Example

```
#ifdef _DEBUG
   // The pointer to my edit.
   extern CEdit* pmyEdit;
   // The point where the user clicked the mouse.
   extern CPoint myPt;

   int n = pmyEdit->CharFromPos(myPt);
   int nLineIndex = HIWORD(n);
   int nCharIndex = LOWORD(n);
   TRACE(TEXT("nLineIndex = %d, nCharIndex = %d\r\n"), nLineIndex, nCharIndex);
#endif
```

See Also

Reference

CEdit Class

Hierarchy Chart

CEdit::PosFromChar

Other Resources

CEdit Members

CEdit::Clear

Call this function to delete (clear) the current selection (if any) in the edit control.

```
void Clear( );
```

Remarks

The deletion performed by **Clear** can be undone by calling the Undo member function.

To delete the current selection and place the deleted contents into the Clipboard, call the Cut member function.

For more information, see WM_CLEAR in the *Platform SDK*.

Example

```
// The pointer to my edit.
extern CEdit* pmyEdit;

// Delete all of the text.
pmyEdit->SetSel(0, -1);
pmyEdit->Clear();
```

See Also

Reference

CEdit Class
Hierarchy Chart
CEdit::Undo
CEdit::Copy
CEdit::Cut
CEdit::Paste

Other Resources

CEdit Members

CEdit::Copy

Call this function to coy the current selection (if any) in the edit control to the Clipboard in **CF_TEXT** format.

```
void Copy( );
```

Remarks

For more information, see WM_COPY in the *Platform SDK*.

Example

```
// The pointer to my edit.
extern CEdit* pmyEdit;

// Copy all of the text to the clipboard.
pmyEdit->SetSel(0, -1);
pmyEdit->Copy();
```

See Also

Reference

CEdit Class
Hierarchy Chart
CEdit::Clear
CEdit::Cut
CEdit::Paste

Other Resources

CEdit Members

CEdit::Create

Creates the Windows edit control and attaches it to the **CEdit** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the edit control's style. Apply any combination of edit styles to the control.

rect

Specifies the edit control's size and position. Can be a **CRect** object or **RECT** structure.

pParentWnd

Specifies the edit control's parent window (usually a **CDialog**). It must not be **NULL**.

nID

Specifies the edit control's ID.

Return Value

Nonzero if initialization is successful; otherwise 0.

Remarks

You construct a **CEdit** object in two steps. First, call the **CEdit** constructor and then call **Create**, which creates the Windows edit control and attaches it to the **CEdit** object.

When **Create** executes, Windows sends the WM_NCCREATE, WM_NCCALCSIZE, WM_CREATE, and WM_GETMINMAXINFO messages to the edit control.

These messages are handled by default by the OnNcCreate, OnNcCalcSize, OnCreate, and OnGetMinMaxInfo member functions in the **CWnd** base class. To extend the default message handling, derive a class from **CEdit**, add a message map to the new class, and override the above message-handler member functions. Override **OnCreate**, for example, to perform needed initialization for the new class.

Apply the following window styles to an edit control.

- **WS_CHILD** Always
- **WS_VISIBLE** Usually
- **WS_DISABLED** Rarely
- **WS_GROUP** To group controls
- **WS_TABSTOP** To include edit control in the tabbing order

Example

```
void CMyView::OnInitialUpdate()
{
   CView::OnInitialUpdate();

   // dynamically create an edit control on the view
   CEdit* pEdit = new CEdit;
   pEdit->Create(ES_MULTILINE | WS_CHILD | WS_VISIBLE | WS_TABSTOP | WS_BORDER,
      CRect(10, 10, 100, 100), this, 1);
}
```

See Also

Reference

CEdit Class

Hierarchy Chart

CEdit::CEdit

Other Resources

CEdit Members

CEdit::Cut

Call this function to delete (cut) the current selection (if any) in the edit control and copy the deleted text to the Clipboard in **CF_TEXT** format.

```
void Cut( );
```

Remarks

The deletion performed by **Cut** can be undone by calling the Undo member function.

To delete the current selection without placing the deleted text into the Clipboard, call the Clear member function.

For more information, see WM_CUT in the *Platform SDK*.

Example

```
// The pointer to my edit.
extern CEdit* pmyEdit;

// Delete all of the text and copy it to the clipboard.
pmyEdit->SetSel(0, -1);
pmyEdit->Cut();
```

See Also

Reference

CEdit Class
Hierarchy Chart
CEdit::Undo
CEdit::Clear
CEdit::Copy
CEdit::Paste

Other Resources

CEdit Members

CEdit::EmptyUndoBuffer

Call this function to reset (clear) the undo flag of an edit control.

```
void EmptyUndoBuffer( );
```

Remarks

The edit control will now be unable to undo the last operation. The undo flag is set whenever an operation within the edit control can be undone.

The undo flag is automatically cleared whenever the SetWindowText or SetHandle **CWnd** member functions are called.

For more information, see EM_EMPTYUNDOBUFFER in the *Platform SDK*.

Example

```
// The pointer to my edit.
extern CEdit* pmyEdit;

// Clear the undo buffer.
if (pmyEdit->CanUndo())
{
   pmyEdit->EmptyUndoBuffer();
   ASSERT(!pmyEdit->CanUndo());
}
```

See Also

Reference

CEdit Class
Hierarchy Chart
CEdit::CanUndo
CEdit::SetHandle
CEdit::Undo
CWnd::SetWindowText

Other Resources

CEdit Members

CEdit::FmtLines

Call this function to set the inclusion of soft line-break characters on or off within a multiple-line edit control.

```
BOOL FmtLines(
   BOOL bAddEOL
);
```

Parameters

bAddEOL

Specifies whether soft line-break characters are to be inserted. A value of **TRUE** inserts the characters; a value of **FALSE** removes them.

Return Value

Nonzero if any formatting occurs; otherwise 0.

Remarks

A soft line break consists of two carriage returns and a linefeed inserted at the end of a line that is broken because of word wrapping. A hard line break consists of one carriage return and a linefeed. Lines that end with a hard line break are not affected by **FmtLines**.

Windows will only respond if the **CEdit** object is a multiple-line edit control.

FmtLines only affects the buffer returned by GetHandle and the text returned by WM_GETTEXT. It has no impact on the display of the text within the edit control.

For more information, see EM_FMTLINES in the *Platform SDK*.

Example

```
#ifdef _DEBUG
   // The pointer to my edit.
   extern CEdit* pmyEdit;
   CString strText;

   // Add soft line-break breaks.
   pmyEdit->FmtLines(TRUE);

   // Dump the text of the edit control.
   pmyEdit->GetWindowText(strText);
   afxDump << strText;

   // Remove soft line-break breaks.
   pmyEdit->FmtLines(FALSE);
#endif
```

See Also

Reference

CEdit Class
Hierarchy Chart
CEdit::GetHandle
CWnd::GetWindowText

Other Resources

CEdit Members

CEdit::GetCueBanner

Retrieves the text that is displayed as the text cue, or tip, in an edit control when the control is empty and does not have focus.

```
BOOL GetCueBanner(
   LPWSTR lpwText,
   int cchText
);
```

Parameters

lpwText

A pointer to a string containing the cue text.

cchText

The number of characters that can be received for the cue text.

Return Value

Nonzero if successful; otherwise 0.

Remarks

See Edit_GetCueBannerText for more information.

This method is supported in Windows XP or later.

See Also

Reference

CEdit Class
Hierarchy Chart
CEdit::SetCueBanner

Other Resources

CEdit Members

CEdit::GetFirstVisibleLine

Call this function to determine the topmost visible line in an edit control.

```
int GetFirstVisibleLine( ) const;
```

Return Value

The zero-based index of the topmost visible line. For single-line edit controls, the return value is 0.

Remarks

For more information, see EM_GETFIRSTVISIBLELINE in the *Platform SDK*.

Example

```
// The pointer to my edit.
extern CEdit* pmyEdit;

int nFirstVisible = pmyEdit->GetFirstVisibleLine();

// Scroll the edit control so that the first visible line
// is the first line of text.
if (nFirstVisible > 0)
{
   pmyEdit->LineScroll(-nFirstVisible, 0);
}
```

See Also

Reference

CEdit Class

Hierarchy Chart

CEdit::GetLine

Other Resources

CEdit Members

CEdit::GetHandle

Call this function to retrieve a handle to the memory currently allocated for a multiple-line edit control.

```
HLOCAL GetHandle( ) const;
```

Return Value

A local memory handle that identifies the buffer holding the contents of the edit control. If an error occurs, such as sending the message to a single-line edit control, the return value is 0.

Remarks

The handle is a local memory handle and may be used by any of the **Local** Windows memory functions that take a local memory handle as a parameter.

GetHandle is processed only by multiple-line edit controls.

Call **GetHandle** for a multiple-line edit control in a dialog box only if the dialog box was created with the **DS_LOCALEDIT** style flag set. If the **DS_LOCALEDIT** style is not set, you will still get a nonzero return value, but you will not be able to use the returned value.

Note

GetHandle will not work with Windows 95/98. If you call **GetHandle** in Windows 95/98, it will return **NULL**. **GetHandle** will work as documented under Windows NT, versions 3.51 and later.

For more information, see EM_GETHANDLE in the *Platform SDK*.

Example

```
#ifdef _DEBUG
   // The pointer to my edit.
   extern CEdit* pmyEdit;

   HLOCAL h = pmyEdit->GetHandle();
   LPCTSTR lpszText = (LPCTSTR) ::LocalLock(h);

   // Dump the text of the edit control.
   afxDump << lpszText;

   ::LocalUnlock(h);
#endif
```

See Also

Reference

CEdit Class

Hierarchy Chart

CEdit::SetHandle

Other Resources

CEdit Members

CEdit::GetLimitText

Call this member function to get the text limit for this **CEdit** object.

```
UINT GetLimitText( ) const;
```

Return Value

The current text limit, in bytes, for this **CEdit** object.

Remarks

The text limit is the maximum amount of text, in bytes, that the edit control can accept.

Note
This member function is available beginning with Windows 95 and Windows NT 4.0.

For more information, see EM_GETLIMITTEXT in the *Platform SDK*.

Example

```
// The pointer to my edit control.
extern CEdit* pmyEdit;
// The new text of the edit control.
extern LPCTSTR lpszmyString;
int nLength = strlen(lpszmyString);

// Want the text limit to be at least the size of the new string.
if (pmyEdit->GetLimitText() < nLength)
   pmyEdit->SetLimitText(nLength);

pmyEdit->SetWindowText(lpszmyString);
```

See Also

Reference

CEdit Class
Hierarchy Chart
CEdit::SetLimitText
CEdit::LimitText

Other Resources

CEdit Members

CEdit::GetLine

Call this function to retrieve a line of text from an edit control and places it in *lpszBuffer*.

```
int GetLine(
   int nIndex,
   LPTSTR lpszBuffer
) const;
int GetLine(
   int nIndex,
   LPTSTR lpszBuffer,
   int nMaxLength
) const;
```

Parameters

nIndex

Specifies the line number to retrieve from a multiple-line edit control. Line numbers are zero-based; a value of 0 specifies the first line. This parameter is ignored by a single-line edit control.

lpszBuffer

Points to the buffer that receives a copy of the line. The first word of the buffer must specify the maximum number of bytes that can be copied to the buffer.

nMaxLength

Specifies the maximum number of bytes that can be copied to the buffer. **GetLine** places this value in the first word of *lpszBuffer* before making the call to Windows.

Return Value

The number of bytes actually copied. The return value is 0 if the line number specified by *nIndex* is greater than the number of lines in the edit control.

Remarks

The copied line does not contain a null-termination character.

For more information, see EM_GETLINE in the *Platform SDK*.

Example

See the example for CEdit::GetLineCount.

See Also

Reference

CEdit Class
Hierarchy Chart
CEdit::LineLength
CWnd::GetWindowText

Other Resources

CEdit Members

CEdit::GetLineCount

Call this function to retrieve the number of lines in a multiple-line edit control.

```
int GetLineCount( ) const;
```

Return Value

An integer containing the number of lines in the multiple-line edit control. If no text has been entered into the edit control, the return value is 1.

Remarks

GetLineCount is only processed by multiple-line edit controls.

For more information, see EM_GETLINECOUNT in the *Platform SDK*.

Example

```
#ifdef _DEBUG
   extern CEdit* pmyEdit;    // The pointer to my edit control:

   int i, nLineCount = pmyEdit->GetLineCount();
   CString strText, strLine;
   // Dump every line of text of the edit control.
   for (i=0;i < nLineCount;i++)
   {
      // length of line i:
      int len = pmyEdit->LineLength(pmyEdit->LineIndex(i));
      pmyEdit->GetLine(i, strText.GetBuffer(len), len);
      strText.ReleaseBuffer(len);
      strLine.Format(TEXT("line %d: '%s'\n"), i, strText);
      afxDump << strLine;
   }
#endif
```

See Also

Reference

CEdit Class

Hierarchy Chart

Other Resources

CEdit Members

CEdit::GetMargins

Call this member function to retrieve the left and right margins of this edit control.

```
DWORD GetMargins( ) const;
```

Return Value

The width of the left margin in the low-order **WORD** and the width of the right margin in the high-order **WORD**.

Remarks

Margins are measured in pixels.

Note
This member function is available beginning with Windows 95 and Windows NT 4.0.

For more information, see EM_GETMARGINS in the *Platform SDK*.

Example

See the example for CEditView::GetEditCtrl.

See Also

Reference

CEdit Class
Hierarchy Chart
CEdit::SetMargins

Other Resources

CEdit Members

CEdit::GetModify

Call this function to determine whether the contents of an edit control have been modified.

```
BOOL GetModify( ) const;
```

Return Value

Nonzero if the edit-control contents have been modified; 0 if they have remained unchanged.

Remarks

Windows maintains an internal flag indicating whether the contents of the edit control have been changed. This flag is cleared when the edit control is first created and may also be cleared by calling the SetModify member function.

For more information, see EM_GETMODIFY in the *Platform SDK*.

Example

```
// The pointer to my edit.
extern CEdit* pmyEdit;

// Reset the modified state only if my edit has been modified.
if (pmyEdit->GetModify())
   pmyEdit->SetModify(FALSE);
```

See Also

Reference

CEdit Class

Hierarchy Chart

CEdit::SetModify

Other Resources

CEdit Members

CEdit::GetPasswordChar

Call this function to retrieve the password character displayed in an edit control when the user enters text.

```
TCHAR GetPasswordChar( ) const;
```

Return Value

Specifies the character to be displayed in place of the character typed by the user. The return value is **NULL** if no password character exists.

Remarks

If the edit control is created with the **ES_PASSWORD** style, the default password character is set to an asterisk (*****).

For more information, see EM_GETPASSWORDCHAR in the *Platform SDK*.

Example

```
// The pointer to my edit.
extern CEdit* pmyEdit;

// Turn on the password mode.
pmyEdit->SetPasswordChar('*');
ASSERT(pmyEdit->GetStyle() & ES_PASSWORD);
ASSERT(pmyEdit->GetPasswordChar() == '*');
```

See Also

Reference

CEdit Class

Hierarchy Chart

CEdit::SetPasswordChar

Other Resources

CEdit Members

CEdit::GetRect

Call this function to get the formatting rectangle of an edit control.

```
void GetRect(
   LPRECT lpRect
) const;
```

Parameters

lpRect

Points to the **RECT** structure that receives the formatting rectangle.

Remarks

The formatting rectangle is the limiting rectangle of the text, which is independent of the size of the edit-control window.

The formatting rectangle of a multiple-line edit control can be modified by the SetRect and SetRectNP member functions.

For more information, see EM_GETRECT in the *Platform SDK*.

Example

See the example for CEdit::LimitText.

See Also

Reference

CEdit Class
Hierarchy Chart
CEdit::SetRect
CEdit::SetRectNP

Other Resources

CEdit Members

CEdit::GetSel

Call this function to get the starting and ending character positions of the current selection (if any) in an edit control, using either the return value or the parameters.

```
DWORD GetSel( ) const;
void GetSel(
   int& nStartChar,
   int& nEndChar
) const;
```

Parameters

nStartChar

Reference to an integer that will receive the position of the first character in the current selection.

nEndChar

Reference to an integer that will receive the position of the first nonselected character past the end of the current selection.

Return Value

The version that returns a **DWORD** returns a value that contains the starting position in the low-order word and the position of the first nonselected character after the end of the selection in the high-order word.

Remarks

For more information, see EM_GETSEL in the *Platform SDK*.

Example

```
// The pointer to my edit.
extern CEdit* pmyEdit;

// Set the selection to be all characters after the current selection.
DWORD dwSel = pmyEdit->GetSel();
pmyEdit->SetSel(HIWORD(dwSel), -1);
```

See Also

Reference

CEdit Class

Hierarchy Chart

CEdit::SetSel

Other Resources

CEdit Members

CEdit::LimitText

Call this function to limit the length of the text that the user may enter into an edit control.

```
void LimitText(
   int nChars = 0
);
```

Parameters

nChars

Specifies the length (in bytes) of the text that the user can enter. If this parameter is 0, the text length is set to **UINT_MAX** bytes. This is the default behavior.

Remarks

Changing the text limit restricts only the text the user can enter. It has no effect on any text already in the edit control, nor does it affect the length of the text copied to the edit control by the SetWindowText member function in **CWnd**. If an application uses the **SetWindowText** function to place more text into an edit control than is specified in the call to **LimitText**, the user can delete any of the text within the edit control. However, the text limit will prevent the user from replacing the existing text with new text, unless deleting the current selection causes the text to fall below the text limit.

Note

In Win32 (Windows NT and Windows 95/98), SetLimitText replaces this function.

For more information, see EM_LIMITTEXT in the *Platform SDK*.

Example

```
// The pointer to my edit.
extern CEdit* pmyEdit;

// Limit the number of characters to be the maximum number visible.

// Get the text metrics for the edit; needed for the
// average character width.
TEXTMETRIC tm;
CDC* pDC = pmyEdit->GetDC();
pDC->GetTextMetrics(&tm);
pmyEdit->ReleaseDC(pDC);

CRect r;
pmyEdit->GetRect(&r);
pmyEdit->LimitText(r.Width()/tm.tmAveCharWidth);
```

See Also

Reference

CEdit Class
Hierarchy Chart
CWnd::SetWindowText
CEdit::GetLimitText
CEdit::SetLimitText

Other Resources

CEdit Members

CEdit::LineFromChar

Call this function to retrieve the line number of the line that contains the specified character index.

```
int LineFromChar(
   int nIndex = -1
) const;
```

Parameters

nIndex

Contains the zero-based index value for the desired character in the text of the edit control, or contains –1. If *nIndex* is –1, it specifies the current line, that is, the line that contains the caret.

Return Value

The zero-based line number of the line containing the character index specified by *nIndex*. If *nIndex* is –1, the number of the line that contains the first character of the selection is returned. If there is no selection, the current line number is returned.

Remarks

A character index is the number of characters from the beginning of the edit control.

This member function is only used by multiple-line edit controls.

For more information, see EM_LINEFROMCHAR in the *Platform SDK*.

Example

```
// The pointer to my edit control.
extern CEdit* pmyEdit;
// The index of the char to get information on.
extern int nIndex;

CString strText;

pmyEdit->GetWindowText(strText);
strText = strText.Mid(nIndex, 1);

// Get the text extent of the character.
CDC* pDC = pmyEdit->GetDC();
CSize sz = pDC->GetTextExtent(strText);
pmyEdit->ReleaseDC(pDC);

CPoint pt = pmyEdit->PosFromChar(nIndex);

// Dump the index, character, line number, and character bounds.
TRACE("nIndex = %d, character = %c, line = %d, bounds = {%d, %d, %d, %d}\r\n",
   nIndex, strText[0], pmyEdit->LineFromChar(nIndex),
   pt.x /* left */, pt.y /* top */,
   pt.x+sz.cx /* right */, pt.y+sz.cy /* bottom */);
```

See Also

Reference

CEdit Class

Hierarchy Chart

CEdit::LineIndex

Other Resources

CEdit Members

CEdit::LineIndex

Call this function to retrieve the character index of a line within a multiple-line edit control.

```
int LineIndex(
   int nLine = -1
) const;
```

Parameters

nLine

Contains the index value for the desired line in the text of the edit control, or contains –1. If *nLine* is –1, it specifies the current line, that is, the line that contains the caret.

Return Value

The character index of the line specified in *nLine* or –1 if the specified line number is greater than the number of lines in the edit control.

Remarks

The character index is the number of characters from the beginning of the edit control to the specified line.

This member function is only processed by multiple-line edit controls.

For more information, see EM_LINEINDEX in the *Platform SDK*.

Example

```
// The pointer to my edit.
extern CEdit* pmyEdit;
// The string for replacing.
extern LPCTSTR lpszmyString;

int nBegin, nEnd;

// Replace the second line, if it exists, of the edit control
// with the text lpszmyString.
if ((nBegin=pmyEdit->LineIndex(1)) != -1)
{
   nEnd = nBegin + pmyEdit->LineLength(nBegin);
   pmyEdit->SetSel(nBegin, nEnd);
   pmyEdit->ReplaceSel(lpszmyString);
}
```

See Also

Reference

CEdit Class

Hierarchy Chart

CEdit::LineFromChar

Other Resources

CEdit Members

CEdit::LineLength

Call this function to retrieve the length of a line in an edit control.

```
int LineLength(
   int nLine = -1
) const;
```

Parameters

nLine

Specifies the character index of a character in the line whose length is to be retrieved. If this parameter is –1, the length of the current line (the line that contains the caret) is returned, not including the length of any selected text within the line. When **LineLength** is called for a single-line edit control, this parameter is ignored.

Return Value

When **LineLength** is called for a multiple-line edit control, the return value is the length (in bytes) of the line specified by *nLine*. When **LineLength** is called for a single-line edit control, the return value is the length (in bytes) of the text in the edit control.

Remarks

Use the LineIndex member function to retrieve a character index for a given line number within a multiple-line edit control.

For more information, see EM_LINELENGTH in the *Platform SDK*.

Example

See the example for CEdit::LineIndex.

See Also

Reference

CEdit Class

Hierarchy Chart

CEdit::LineIndex

Other Resources

CEdit Members

CEdit::LineScroll

Call this function to scroll the text of a multiple-line edit control.

```
void LineScroll(
   int nLines,
   int nChars = 0
);
```

Parameters

nLines

Specifies the number of lines to scroll vertically.

nChars

Specifies the number of character positions to scroll horizontally. This value is ignored if the edit control has either the **ES_RIGHT** or **ES_CENTER** style.

Remarks

This member function is processed only by multiple-line edit controls.

The edit control does not scroll vertically past the last line of text in the edit control. If the current line plus the number of lines specified by *nLines* exceeds the total number of lines in the edit control, the value is adjusted so that the last line of the edit control is scrolled to the top of the edit-control window.

LineScroll can be used to scroll horizontally past the last character of any line.

For more information, see EM_LINESCROLL in the *Platform SDK*.

Example

See the example for CEdit::GetFirstVisibleLine.

See Also

Reference

CEdit Class
Hierarchy Chart
CEdit::LineIndex

Other Resources

CEdit Members

CEdit::Paste

Call this function to insert the data from the Clipboard into the **CEdit** at the insertion point.

```
void Paste( );
```

Remarks

Data is inserted only if the Clipboard contains data in **CF_TEXT** format.

For more information, see WM_PASTE in the *Platform SDK*.

Example

```
// The pointer to my edit.
extern CEdit* pmyEdit;

// Replace all of the text with the text in the clipboard.
pmyEdit->SetSel(0, -1);
pmyEdit->Paste();
```

See Also

Reference

CEdit Class
Hierarchy Chart
CEdit::Clear
CEdit::Copy
CEdit::Cut

Other Resources

CEdit Members

CEdit::PosFromChar

Call this function to get the position (top-left corner) of a given character within this **CEdit** object.

```
CPoint PosFromChar(
   UINT nChar
) const;
```

Parameters

nChar

The zero-based index of the specified character.

Return Value

The coordinates of the top-left corner of the character specified by *nChar*.

Remarks

The character is specified by giving its zero-based index value. If *nChar* is greater than the index of the last character in this **CEdit** object, the return value specifies the coordinates of the character position just past the last character in this **CEdit** object.

Note
This member function is available beginning with Windows 95 and Windows NT 4.0.

For more information, see EM_POSFROMCHAR in the *Platform SDK*.

Example

See the example for CEdit::LineFromChar.

See Also

Reference

CEdit Class

Hierarchy Chart

CEdit::CharFromPos

Other Resources

CEdit Members

CEdit::ReplaceSel

Call this function to replace the current selection in an edit control with the text specified by *lpszNewText*.

```
void ReplaceSel(
   LPCTSTR lpszNewText,
      BOOL bCanUndo = FALSE
);
```

Parameters

lpszNewText

Points to a null-terminated string containing the replacement text.

bCanUndo

To specify that this function can be undone, set the value of this parameter to **TRUE** . The default value is **FALSE**.

Remarks

Replaces only a portion of the text in an edit control. If you want to replace all of the text, use the CWnd::SetWindowText member function.

If there is no current selection, the replacement text is inserted at the current cursor location.

For more information, see EM_REPLACESEL in the *Platform SDK*.

Example

See the example for CEdit::LineIndex.

See Also

Reference

CEdit Class
Hierarchy Chart
CWnd::SetWindowText

Other Resources

CEdit Members

CEdit::SetCueBanner

Sets the text that is displayed as the text cue, or tip, in an edit control when the control is empty and does not have focus.

```
BOOL SetCueBanner(
   LPCWSTR lpcwText
);
```

Parameters

lpcwText

A pointer to a string containing the text cue to display in the edit box.

Return Value

Nonzero if the cue text is set successfully; otherwise zero.

Remarks

See Edit_SetCueBannerText for more information.

This method is supported in Windows XP or later.

Example

```
// The pointer to my edit.
extern CEdit* pmyEdit;

// Set text to display in the edit box.
pmyEdit->SetCueBanner(L"Enter your text here");
```

See Also

Reference

CEdit Class

Hierarchy Chart

CEdit::GetCueBanner

Other Resources

CEdit Members

CEdit::SetHandle

Call this function to set the handle to the local memory that will be used by a multiple-line edit control.

```
void SetHandle(
   HLOCAL hBuffer
);
```

Parameters

hBuffer

Contains a handle to the local memory. This handle must have been created by a previous call to the LocalAlloc Windows function using the **LMEM_MOVEABLE** flag. The memory is assumed to contain a null-terminated string. If this is not the case, the first byte of the allocated memory should be set to 0.

Remarks

The edit control will then use this buffer to store the currently displayed text instead of allocating its own buffer.

This member function is processed only by multiple-line edit controls.

Before an application sets a new memory handle, it should use the GetHandle member function to get the handle to the current memory buffer and free that memory using the **LocalFree** Windows function.

SetHandle clears the undo buffer (the CanUndo member function then returns 0) and the internal modification flag (the GetModify member function then returns 0). The edit-control window is redrawn.

You can use this member function in a multiple-line edit control in a dialog box only if you have created the dialog box with the **DS_LOCALEDIT** style flag set.

Note

GetHandle will not work with Windows 95/98. If you call **GetHandle** in Windows 95/98, it will return **NULL**. **GetHandle** wil l work as documented under Windows NT, versions 3.51 and later.

For more information, see EM_SETHANDLE, LocalAlloc, and LocalFree in the *Platform SDK*.

Example

```
// The pointer to my edit control.
extern CEdit* pmyEdit;
// The string to set in the edit control.
extern LPCTSTR lpszmyString;

// Initialize the new local handle.
HLOCAL h = ::LocalAlloc(LHND, strlen(lpszmyString)+sizeof(TCHAR));
LPTSTR lpszText = (LPTSTR) ::LocalLock(h);
strcpy(lpszText, lpszmyString);
::LocalUnlock(h);

// Free the current text handle of the edit control.
::LocalFree(pmyEdit->GetHandle());

// Set the new text handle.
pmyEdit->SetHandle(h);
```

See Also

Reference

CEdit Class
Hierarchy Chart
CEdit::CanUndo
CEdit::GetHandle
CEdit::GetModify

Other Resources

CEdit Members

CEdit::SetLimitText

Call this member function to set the text limit for this **CEdit** object.

```
void SetLimitText(
   UINT nMax
);
```

Parameters

nMax

The new text limit, in characters.

Remarks

The text limit is the maximum amount of text, in characters, that the edit control can accept.

Changing the text limit restricts only the text the user can enter. It has no effect on any text already in the edit control, nor does it affect the length of the text copied to the edit control by the SetWindowText member function in **CWnd**. If an application uses the **SetWindowText** function to place more text into an edit control than is specified in the call to **LimitText**, the user can delete any of the text within the edit control. However, the text limit will prevent the user from replacing the existing text with new text, unless deleting the current selection causes the text to fall below the text limit.

This function replaces LimitText in Win32.

For more information, see EM_SETLIMITTEXT in the *Platform SDK*.

Example

See the example for CEditView::GetEditCtrl.

See Also

Reference

CEdit Class
Hierarchy Chart
CEdit::GetLimitText
CEdit::LimitText

Other Resources

CEdit Members

CEdit::SetMargins

Call this member function to set the left and right margins of this edit control.

```
void SetMargins(
   UINT nLeft,
   UINT nRight
);
```

Parameters

nLeft

The width of the new left margin, in pixels.

nRight

The width of the new right margin, in pixels.

Remarks

Note
This member function is available beginning with Windows 95 and Windows NT 4.0.

For more information, see EM_SETMARGINS in the *Platform SDK*.

Example

See the example for CEditView::GetEditCtrl.

See Also

Reference

CEdit Class
Hierarchy Chart
CEdit::GetMargins

Other Resources

CEdit Members

CEdit::SetModify

Call this function to set or clear the modified flag for an edit control.

```
void SetModify(
   BOOL bModified = TRUE
);
```

Parameters

bModified

A value of **TRUE** indicates that the text has been modified, and a value of **FALSE** indicates it is unmodified. By default, the modified flag is set.

Remarks

The modified flag indicates whether or not the text within the edit control has been modified. It is automatically set whenever the user changes the text. Its value may be retrieved with the GetModify member function.

For more information, see EM_SETMODIFY in the *Platform SDK*.

Example

See the example for CEdit::GetModify.

See Also

Reference

CEdit Class

Hierarchy Chart

CEdit::GetModify

Other Resources

CEdit Members

CEdit::SetPasswordChar

Call this function to set or remove a password character displayed in an edit control when the user types text.

```
void SetPasswordChar(
   TCHAR ch
);
```

Parameters

ch

Specifies the character to be displayed in place of the character typed by the user. If *ch* is 0, the actual characters typed by the user are displayed.

Remarks

When a password character is set, that character is displayed for each character the user types.

This member function has no effect on a multiple-line edit control.

When the **SetPasswordChar** member function is called, **CEdit** will redraw all visible characters using the character specified by *ch*.

If the edit control is created with the ES_PASSWORD style, the default password character is set to an asterisk (*). This style is removed if **SetPasswordChar** is called with *ch* set to 0.

For more information, see EM_SETPASSWORDCHAR in the *Platform SDK*.

Example

```
// The pointer to my edit.
extern CEdit* pmyEdit;

// Turn off the password mode.
pmyEdit->SetPasswordChar(0);
ASSERT(!(pmyEdit->GetStyle() & ES_PASSWORD));
```

See Also

Reference

CEdit Class

Hierarchy Chart

CEdit::GetPasswordChar

Other Resources

CEdit Members

CEdit::SetReadOnly

Calls this function to set the read-only state of an edit control.

```
BOOL SetReadOnly(
   BOOL bReadOnly = TRUE
);
```

Parameters

bReadOnly

Specifies whether to set or remove the read-only state of the edit control. A value of **TRUE** sets the state to read-only; a value of **FALSE** sets the state to read/write.

Return Value

Nonzero if the operation is successful, or 0 if an error occurs.

Remarks

The current setting can be found by testing the ES_READONLY flag in the return value of CWnd::GetStyle.

For more information, see EM_SETREADONLY in the *Platform SDK*.

Example

```
// The pointer to my edit.
extern CEdit* pmyEdit;

// Set the edit control to be read-only.
pmyEdit->SetReadOnly(TRUE);
ASSERT(pmyEdit->GetStyle() & ES_READONLY);
```

See Also

Reference

CEdit Class

Hierarchy Chart

CWnd::GetStyle

Other Resources

CEdit Members

CEdit::SetRect

Call this function to set the dimensions of a rectangle using the specified coordinates.

```
void SetRect(
   LPCRECT lpRect
);
```

Parameters

lpRect

Points to the **RECT** structure or **CRect** object that specifies the new dimensions of the formatting rectangle.

Remarks

This member is processed only by multiple-line edit controls.

Use **SetRect** to set the formatting rectangle of a multiple-line edit control. The formatting rectangle is the limiting rectangle of the text, which is independent of the size of the edit-control window. When the edit control is first created, the formatting rectangle is the same as the client area of the edit-control window. By using the **SetRect** member function, an application can make the formatting rectangle larger or smaller than the edit-control window.

If the edit control has no scroll bar, text will be clipped, not wrapped, if the formatting rectangle is made larger than the window. If the edit control contains a border, the formatting rectangle is reduced by the size of the border. If you adjust the rectangle returned by the **GetRect** member function, you must remove the size of the border before you pass the rectangle to **SetRect**.

When **SetRect** is called, the edit control's text is also reformatted and redisplayed.

For more information, see EM_SETRECT in the *Platform SDK*.

Example

```
// The pointer to my edit.
extern CEdit* pmyEdit;
// Flag indicating whether to redraw the edit control.
extern bool fRedraw;

CRect r;

pmyEdit->GetRect(&r);

// Reduce the formatting rect of the edit control by
// 10 pixels on each side.
if ((r.Width() > 20) && (r.Height() > 20))
{
   r.DeflateRect(10, 10);

   if (fRedraw)
      pmyEdit->SetRect(&r);
   else
      pmyEdit->SetRectNP(&r);
}
```

See Also

Reference

CEdit Class
Hierarchy Chart
CRect::CRect
CRect::CopyRect
CRect::operator =
CRect::SetRectEmpty
CEdit::GetRect
CEdit::SetRectNP

Other Resources

CEdit Members

CEdit::SetRectNP

Call this function to set the formatting rectangle of a multiple-line edit control.

```
void SetRectNP(
   LPCRECT lpRect
);
```

Parameters

lpRect

Points to a **RECT** structure or **CRect** object that specifies the new dimensions of the rectangle.

Remarks

The formatting rectangle is the limiting rectangle of the text, which is independent of the size of the edit-control window.

SetRectNP is identical to the **SetRect** member function except that the edit-control window is not redrawn.

When the edit control is first created, the formatting rectangle is the same as the client area of the edit-control window. By calling the **SetRectNP** member function, an application can make the formatting rectangle larger or smaller than the edit-control window.

If the edit control has no scroll bar, text will be clipped, not wrapped, if the formatting rectangle is made larger than the window.

This member is processed only by multiple-line edit controls.

For more information, see EM_SETRECTNP in the *Platform SDK*.

Example

See the example for CEdit::SetRect.

See Also

Reference

CEdit Class
Hierarchy Chart
CRect::CRect
CRect::CopyRect
CRect::operator =
CRect::SetRectEmpty
CEdit::GetRect
CEdit::SetRect

Other Resources

CEdit Members

CEdit::SetSel

Call this function to select a range of characters in an edit control.

```
void SetSel(
   DWORD dwSelection,
   BOOL bNoScroll = FALSE
);
void SetSel(
   int nStartChar,
   int nEndChar,
   BOOL bNoScroll = FALSE
);
```

Parameters

dwSelection

Specifies the starting position in the low-order word and the ending position in the high-order word. If the low-order word is 0 and the high-order word is –1, all the text in the edit control is selected. If the low-order word is –1, any current selection is removed.

bNoScroll

Indicates whether the caret should be scrolled into view. If **FALSE**, the caret is scrolled into view. If **TRUE**, the caret is not scrolled into view.

nStartChar

Specifies the starting position. If *nStartChar* is 0 and *nEndChar* is –1, all the text in the edit control is selected. If *nStartChar* is –1, any current selection is removed.

nEndChar

Specifies the ending position.

Remarks

For more information, see EM_SETSEL in the *Platform SDK*.

Example

See the example for CEdit::GetSel.

See Also

Reference

CEdit Class
Hierarchy Chart
CEdit::GetSel
CEdit::ReplaceSel

Other Resources

CEdit Members

CEdit::SetTabStops

Call this function to set the tab stops in a multiple-line edit control.

```
void SetTabStops( );
BOOL SetTabStops(
   const int& cxEachStop
);
BOOL SetTabStops(
   int nTabStops,
   LPINT rgTabStops
);
```

Parameters

cxEachStop

Specifies that tab stops are to be set at every *cxEachStop* dialog units.

nTabStops

Specifies the number of tab stops contained in *rgTabStops*. This number must be greater than 1.

rgTabStops

Points to an array of unsigned integers specifying the tab stops in dialog units. A dialog unit is a horizontal or vertical distance. One horizontal dialog unit is equal to one-fourth of the current dialog base width unit, and 1 vertical dialog unit is equal to one-eighth of the current dialog base height unit. The dialog base units are computed based on the height and width of the current system font. The **GetDialogBaseUnits** Windows function returns the current dialog base units in pixels.

Return Value

Nonzero if the tabs were set; otherwise 0.

Remarks

When text is copied to a multiple-line edit control, any tab character in the text will cause space to be generated up to the next tab stop.

To set tab stops to the default size of 32 dialog units, call the parameterless version of this member function. To set tab stops to a size other than 32, call the version with the *cxEachStop* parameter. To set tab stops to an array of sizes, use the version with two parameters.

This member function is only processed by multiple-line edit controls.

SetTabStops does not automatically redraw the edit window. If you change the tab stops for text already in the edit control, call CWnd::InvalidateRect to redraw the edit window.

For more information, see EM_SETTABSTOPS and GetDialogBaseUnits in the *Platform SDK*.

Example

See the example for CEditView::SetTabStops.

See Also

Reference

CEdit Class
Hierarchy Chart
CWnd::InvalidateRect

Other Resources

CEdit Members

CEdit::Undo

Call this function to undo the last edit-control operation.

```
BOOL Undo( );
```

Return Value

For a single-line edit control, the return value is always nonzero. For a multiple-line edit control, the return value is nonzero if the undo operation is successful, or 0 if the undo operation fails.

Remarks

An undo operation can also be undone. For example, you can restore deleted text with the first call to **Undo**. As long as there is no intervening edit operation, you can remove the text again with a second call to **Undo**.

For more information, see EM_UNDO in the *Platform SDK*.

Example

```
// The pointer to my edit.
extern CEdit* pmyEdit;

// Undo the last operation, if possible.
if (pmyEdit->CanUndo())
   pmyEdit->Undo();
```

See Also

Reference

CEdit Class

Hierarchy Chart

CEdit::CanUndo

Other Resources

CEdit Members

CEditView Class

This topic has been updated for Visual Studio 2005 SP1.

A type of view class that provides the functionality of a Windows edit control and can be used to implement simple text-editor functionality.

```
class CEditView : public CCtrlView
```

Remarks

The **CEditView** class provides the following additional functions:

- Print.
- Find and replace.

Because class **CEditView** is a derivative of class **CView**, objects of class **CEditView** can be used with documents and document templates.

Each **CEditView** control's text is kept in its own global memory object. Your application can have any number of **CEditView** objects.

Create objects of type **CEditView** if you want an edit window with the added functionality listed above, or if you want simple text-editor functionality. A **CEditView** object can occupy the entire client area of a window. Derive your own classes from **CEditView** to add or modify the basic functionality, or to declare classes that can be added to a document template.

The default implementation of class **CEditView** handles the following commands: **ID_EDIT_SELECT_ALL**, **ID_EDIT_FIND**, **ID_EDIT_REPLACE**, **ID_EDIT_REPEAT**, and **ID_FILE_PRINT**.

The default character limit for **CEditView** is (1024 * 1024 - 1 = 1048575). This can be changed by calling the **EM_LIMITTEXT** function of the underlying edit control. However, the limits are different depending on the operating system and the type of edit control (single or multiline). For more information on these limits, see EM_LIMITTEXT.

To change this limit in your control, override the `OnCreate()` function for your **CEditView** class and insert the following line of code:

```
GetEditCtrl().SetLimitText(nNewVal); //nNewVal, the new character limit
```

Objects of type **CEditView** (or of types derived from **CEditView**) have the following limitations:

- **CEditView** does not implement true what you see is what you get (WYSIWYG) editing. Where there is a choice between readability on the screen and matching printed output, **CEditView** opts for screen readability.
- **CEditView** can display text in only a single font. No special character formatting is supported. See class CRichEditView for greater capabilities.
- The amount of text a **CEditView** can contain is limited. The limits are the same as for the **CEdit** control.

For more information on **CEditView**, see Derived View Classes Available in MFC.

Requirements

Header: afxext.h

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

In Windows CE-based projects, CEditView::LockBuffer and CEditView::UnlockBuffer do not lock and unlock the text contents of CEdit Class. `LockBuffer` only refreshes the `m_pshadowbuffer` buffer variable by checking for changes and then calling the inherited function CWnd::GetWindowText. `Unlockbuffer` does not do anything and is required for code compatibility.

Search over selected text does not work on Smartphone platforms because text cannot be selected on Smartphone devices.

The Smart Device version of CEditView Class does not support the following print methods:

- CEditView::GetPrinterFont
- CEditView::SetPrinterFont
- CView::OnPrepareDC
- CView::OnPreparePrinting
- CView::OnBeginPrinting
- CView::OnPrint
- CView::OnEndPrinting

See Also

Tasks

SUPERPAD Sample: Demonstrates a Visual Editing Server That Edits Text Using CEditView

Reference

CCtrlView Class
Hierarchy Chart
CEdit Class
CDocument Class
CDocTemplate Class
CCtrlView Class
CRichEditView Class
CView Class

Concepts

CEditView Members

CEditView Members

This topic has been updated for Visual Studio 2005 SP1.

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CView Members

CCtrlView Members

Data Members

dwStyleDefault	Default style for objects of type **CEditView.**

Construction

CEditView	Constructs an object of type **CEditView**.

Attributes

GetBufferLength	Obtains the length of the character buffer.
GetEditCtrl	Provides access to the **CEdit** portion of a **CEditView** object (the Windows edit control).
GetPrinterFont	Retrieves the current printer font.
GetSelectedText	Retrieves the current text selection.
LockBuffer	Locks the buffer.
SetPrinterFont	Sets a new printer font.
SetTabStops	Sets tab stops for both screen display and printing.
UnlockBuffer	Unlocks the buffer.

Operations

FindText	Searches for a string within the text.
PrintInsideRect	Renders text inside a given rectangle.
SerializeRaw	Serializes a **CEditView** object to disk as raw text.

Overridables

OnFindNext	Finds next occurrence of a text string.
OnReplaceAll	Replaces all occurrences of a given string with a new string.
OnReplaceSel	Replaces current selection.
OnTextNotFound	Called when a find operation fails to match any further text.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

In Windows CE-based projects, CEditView::LockBuffer and CEditView::UnlockBuffer do not lock and unlock the text contents of CEdit Class. `LockBuffer` only refreshes the `m_pshadowbuffer` buffer variable by checking for changes and then calling the inherited function CWnd::GetWindowText. `Unlockbuffer` does not do anything and is required for code compatibility.

Search over selected text does not work on Smartphone platforms because text cannot be selected on Smartphone devices.

The Smart Device version of CEditView Class does not support the following print methods:

- CEditView::GetPrinterFont
- CEditView::SetPrinterFont
- CView::OnPrepareDC
- CView::OnPreparePrinting
- CView::OnBeginPrinting
- CView::OnPrint
- CView::OnEndPrinting

See Also

Reference

CEditView Class
Hierarchy Chart

CEditView Member Functions

For information about the member functions in **CEditView**, see CEditView Members.

CEditView::CEditView

Constructs an object of type **CEditView**.

```
CEditView( );
```

Remarks

After constructing the object, you must call the CWnd::Create function before the edit control is used. If you derive a class from **CEditView** and add it to the template using **CWinApp::AddDocTemplate**, the framework calls both this constructor and the **Create** function.

See Also

Reference

CEditView Class
Hierarchy Chart
CWnd::Create
CWinApp::AddDocTemplate

Concepts

CEditView Members

CEditView::FindText

Call the **FindText** function to search the **CEditView** object's text buffer.

```
BOOL FindText(
   LPCTSTR lpszFind,
   BOOL bNext = TRUE,
   BOOL bCase = TRUE
);
```

Parameters

lpszFind

The text to be found.

bNext

Specifies the direction of the search. If **TRUE**, the search direction is toward the end of the buffer. If **FALSE**, the search direction is toward the beginning of the buffer.

bCase

Specifies whether the search is case sensitive. If **TRUE**, the search is case sensitive. If **FALSE**, the search is not case sensitive.

Return Value

Nonzero if the search text is found; otherwise 0.

Remarks

This function searches the text in the buffer for the text specified by *lpszFind*, starting at the current selection, in the direction specified by *bNext*, and with case sensitivity specified by *bCase*. If the text is found, it sets the selection to the found text and returns a nonzero value. If the text is not found, the function returns 0.

You normally do not need to call the **FindText** function unless you override **OnFindNext**, which calls **FindText**.

See Also

Reference

CEditView Class
Hierarchy Chart
CEditView::OnFindNext
CEditView::OnReplaceAll
CEditView::OnReplaceSel
CEditView::OnTextNotFound

Concepts

CEditView Members

CEditView::GetBufferLength

Call this member function to obtain the number of characters currently in the edit control's buffer, not including the null terminator.

```
UINT GetBufferLength( ) const;
```

Return Value

The length of the string in the buffer.

See Also

Reference

CEditView Class
Hierarchy Chart
CEditView::LockBuffer
CEditView::UnlockBuffer

Concepts

CEditView Members

CEditView::GetEditCtrl

Call **GetEditCtrl** to get a reference to the edit control used by the edit view.

```
CEdit& GetEditCtrl( ) const;
```

Return Value

A reference to a **CEdit** object.

Remarks

This control is of type CEdit, so you can manipulate the Windows edit control directly using the **CEdit** member functions.

Caution

Using the **CEdit** object can change the state of the underlying Windows edit control. For example, you should not change the tab settings using the CEdit::SetTabStops function because **CEditView** caches these settings for use both in the edit control a nd in printing. Instead, use CEditView::SetTabStops.

Example

```
CMyEditView::OnInitialUpdate()
{
   // get the edit control and set some initial properties for it
   CEdit& theEdit = GetEditCtrl();

   // adjust the left margin without changing the right margin
   DWORD dwMargins = theEdit.GetMargins();
   theEdit.SetMargins(20, HIWORD(dwMargins));

   // only accept 10k of text
   theEdit.SetLimitText(10 * 1024);
}
```

See Also

Reference

CEditView Class
Hierarchy Chart
CEdit Class
CEditView::SetTabStops

Concepts

CEditView Members

CEditView::GetPrinterFont

This topic has been updated for Visual Studio 2005 SP1.

Call **GetPrinterFont** to get a pointer to a CFont object that describes the current printer font.

```
CFont* GetPrinterFont( ) const;
```

Return Value

A pointer to a **CFont** object that specifies the current printer font; **NULL** if the printer font has not been set. The pointer may be temporary and should not be stored for later use.

Remarks

If the printer font has not been set, the default printing behavior of the **CEditView** class is to print using the same font used for display.

Use this function to determine the current printer font. If it is not the desired printer font, use CEditView::SetPrinterFont to change it.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CEditView Class
Hierarchy Chart
CEditView::SetPrinterFont

Concepts

CEditView Members

CEditView::GetSelectedText

Call **GetSelectedText** to copy the selected text into a **CString** object, up to the end of the selection or the character preceding the first carriage-return character in the selection.

```
void GetSelectedText(
   CString& strResult
) const;
```

Parameters

strResult

A reference to the **CString** object that is to receive the selected text.

See Also

Reference

CEditView Class

Hierarchy Chart

CEditView::OnReplaceSel

Concepts

CEditView Members

CEditView::LockBuffer

Call this member function to obtain a pointer to the buffer. The buffer should not be modified.

```
LPCTSTR LockBuffer( ) const;
```

Return Value

A pointer to the edit control's buffer.

See Also

Reference

CEditView Class
Hierarchy Chart
CEditView::UnlockBuffer
CEditView::GetBufferLength

Concepts

CEditView Members

CEditView::OnFindNext

Searches the text in the buffer for the text specified by *lpszFind*, in the direction specified by *bNext*, with case sensitivity specified by *bCase*.

```
virtual void OnFindNext(
   LPCTSTR lpszFind,
   BOOL bNext,
   BOOL bCase
);
```

Parameters

lpszFind

The text to be found.

bNext

Specifies the direction of the search. If **TRUE**, the search direction is toward the end of the buffer. If **FALSE**, the search direction is toward the beginning of the buffer.

bCase

Specifies whether the search is case sensitive. If **TRUE**, the search is case sensitive. If **FALSE**, the search is not case sensitive.

Remarks

The search starts at the beginning of the current selection and is accomplished through a call to FindText. In the default implementation, **OnFindNext** calls OnTextNotFound if the text is not found.

Override **OnFindNext** to change the way a **CEditView**-derived object searches text. **CEditView** calls **OnFindNext** when the user chooses the Find Next button in the standard Find dialog box.

See Also

Reference

CEditView Class
Hierarchy Chart
CEditView::OnTextNotFound
CEditView::FindText
CEditView::OnReplaceAll
CEditView::OnReplaceSel

Concepts

CEditView Members

CEditView::OnReplaceAll

CEditView calls **OnReplaceAll** when the user selects the Replace All button in the standard Replace dialog box.

```
virtual void OnReplaceAll(
   LPCTSTR lpszFind,
   LPCTSTR lpszReplace,
   BOOL bCase
);
```

Parameters

lpszFind

The text to be found.

lpszReplace

The text to replace the search text.

bCase

Specifies whether search is case sensitive. If **TRUE**, the search is case sensitive. If **FALSE**, the search is not case sensitive.

Remarks

OnReplaceAll searches the text in the buffer for the text specified by *lpszFind*, with case sensitivity specified by *bCase*. The search starts at the beginning of the current selection. Each time the search text is found, this function replaces that occurrence of the text with the text specified by *lpszReplace*. The search is accomplished through a call to FindText. In the default implementation, OnTextNotFound is called if the text is not found.

If the current selection does not match *lpszFind*, the selection is updated to the first occurrence of the text specified by *lpszFind* and a replace is not performed. This allows the user to confirm that this is what they want to do when the selection does not match the text to be replaced.

Override **OnReplaceAll** to change the way a **CEditView**-derived object replaces text.

See Also

Reference

CEditView Class
Hierarchy Chart
CEditView::OnFindNext
CEditView::OnTextNotFound
CEditView::FindText
CEditView::OnReplaceSel

Concepts

CEditView Members

CEditView::OnReplaceSel

CEditView calls **OnReplaceSel** when the user selects the Replace button in the standard Replace dialog box.

```
virtual void OnReplaceSel(
   LPCTSTR lpszFind,
   BOOL bNext,
   BOOL bCase,
   LPCTSTR lpszReplace
);
```

Parameters

lpszFind

The text to be found.

bNext

Specifies the direction of the search. If **TRUE**, the search direction is toward the end of the buffer. If **FALSE**, the search direction is toward the beginning of the buffer.

bCase

Specifies whether the search is case sensitive. If **TRUE**, the search is case sensitive. If **FALSE**, the search is not case sensitive.

lpszReplace

The text to replace the found text.

Remarks

After replacing the selection, this function searches the text in the buffer for the next occurrence of the text specified by *lpszFind*, in the direction specified by *bNext*, with case sensitivity specified by *bCase*. The search is accomplished through a call to FindText. If the text is not found, OnTextNotFound is called.

Override **OnReplaceSel** to change the way a **CEditView**-derived object replaces the selected text.

See Also

Reference

CEditView Class
Hierarchy Chart
CEditView::OnFindNext
CEditView::OnTextNotFound
CEditView::FindText
CEditView::OnReplaceAll

Concepts

CEditView Members

CEditView::OnTextNotFound

Override this function to change the default implementation, which calls the Windows function **MessageBeep**.

```
virtual void OnTextNotFound(
   LPCTSTR lpszFind
);
```

Parameters

lpszFind

The text to be found.

See Also

Reference

CEditView Class
Hierarchy Chart
CEditView::FindText
CEditView::OnFindNext
CEditView::OnReplaceAll
CEditView::OnReplaceSel

Concepts

CEditView Members

CEditView::PrintInsideRect

Call **PrintInsideRect** to print text in the rectangle specified by *rectLayout*.

```
UINT PrintInsideRect(
   CDC *pDC,
   RECT& rectLayout,
   UINT nIndexStart,
   UINT nIndexStop
);
```

Parameters

pDC

Pointer to the printer device context.

rectLayout

Reference to a CRect object or RECT structure specifying the rectangle in which the text is to be rendered.

nIndexStart

Index within the buffer of the first character to be rendered.

nIndexStop

Index within the buffer of the character following the last character to be rendered.

Return Value

The index of the next character to be printed (that is, the character following the last character rendered).

Remarks

If the **CEditView** control does not have the style **ES_AUTOHSCROLL**, text is wrapped within the rendering rectangle. If the control does have the style **ES_AUTOHSCROLL**, the text is clipped at the right edge of the rectangle.

The **rect.bottom** element of the *rectLayout* object is changed so that the rectangle's dimensions define the part of the original rectangle that is occupied by the text.

See Also

Reference

CEditView Class
Hierarchy Chart
CEditView::SetPrinterFont
CEditView::GetPrinterFont

Concepts

CEditView Members

CEditView::SerializeRaw

Call **SerializeRaw** to have a **CArchive** object read or write the text in the **CEditView** object to a text file.

```
void SerializeRaw(
   CArchive& ar
);
```

Parameters

ar

Reference to the **CArchive** object that stores the serialized text.

Remarks

SerializeRaw differs from **CEditView**'s internal implementation of **Serialize** in that it reads and writes only the text, without preceding object-description data.

See Also

Reference

CEditView Class
Hierarchy Chart
CArchive Class
CObject::Serialize

Concepts

CEditView Members

CEditView::SetPrinterFont

This topic has been updated for Visual Studio 2005 SP1.

Call **SetPrinterFont** to set the printer font to the font specified by *pFont*.

```
void SetPrinterFont(
   CFont* pFont
);
```

Parameters

pFont

A pointer to an object of type **CFont**. If **NULL**, the font used for printing is based on the display font.

Remarks

If you want your view to always use a particular font for printing, include a call to **SetPrinterFont** in your class's **OnPreparePrinting** function. This virtual function is called before printing occurs, so the font change takes place before the view's contents are printed.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CEditView Class
Hierarchy Chart
CWnd::SetFont
CFont Class
CView::OnPreparePrinting

Concepts

CEditView Members

CEditView::SetTabStops

Call this function to set the tab stops used for display and printing.

```
void SetTabStops(
   int nTabStops
);
```

Parameters

nTabStops

Width of each tab stop, in dialog units.

Remarks

Only a single tab-stop width is supported. (**CEdit** objects support multiple tab widths.) Widths are in dialog units, which equal one-fourth of the average character width (based on uppercase and lowercase alphabetic characters only) of the font used at the time of printing or displaying. You should not use **CEdit::SetTabStops** because **CEditView** must cache the tab-stop value.

This function modifies only the tabs of the object for which it is called. To change the tab stops for each **CEditView** object in your application, call each object's **SetTabStops** function.

Example

This code fragment sets the tab stops in the control to every fourth character by carefully measuring the font the control uses.

```
// gain a pointer to the control
CEdit* pCtrl = (CEdit*) GetDlgItem(IDC_EDIT1);
ASSERT(pCtrl != NULL);

// get the font the control is using
CFont* pFont = pCtrl->GetFont();
TEXTMETRIC tm;

// get the control's DC, too
CDC* pDC = pCtrl->GetDC();

// Select the font that the control uses by default into the DC.
// We must do this because the control may or may not be using
// that font at this exact moment
CFont* pOldFont = pDC->SelectObject(pFont);

// Retreive text metrics for that font and return the previously
// selected font.
pDC->GetTextMetrics(&tm);
pDC->SelectObject(pOldFont);

// Get an identity rectangle and map it to dialog units
CRect rect(0, 0, 100, 1);
MapDialogRect(rect);

// We now know that 100 dialog units are rect.Width() screen units,
// so we can multiply screen units by 100 and divide by rect.Width()
// to find dialog units from screen units. tm.tmAveCharWidth is
// the width of _one_ character, so setting the tabs at every
// four characters means we also multiply by four.

pCtrl->SetTabStops((4 * tm.tmAveCharWidth * 100) / rect.Width());
```

See Also

Reference

CEditView Class

Hierarchy Chart

CWnd::SetFont

CEditView::SetPrinterFont

Concepts

CEditView Members

CEditView::UnlockBuffer

Call this member function to unlock the buffer.

```
void UnlockBuffer( ) const;
```

Remarks

Call **UnlockBuffer** after you have finished using the pointer returned by LockBuffer.

See Also

Reference

CEditView Class
Hierarchy Chart
CEditView::LockBuffer
CEditView::GetBufferLength

Concepts

CEditView Members

CEditView Data Members

For information about the data members in **CEditView**, see CEditView Members.

CEditView::dwStyleDefault

Contains the default style of the **CEditView** object.

```
static const DWORD dwStyleDefault;
```

Remarks

Pass this static member as the *dwStyle* parameter of the **Create** function to obtain the default style for the **CEditView** object.

See Also

Reference

CEditView Class

Hierarchy Chart

Concepts

CEditView Members

CEvent Class

Represents an "event" — a synchronization object that allows one thread to notify another that an event has occurred.

```
class CEvent : public CSyncObject
```

Remarks

Events are useful when a thread needs to know when to perform its task. For example, a thread that copies data to a data archive would need to be notified when new data is available. By using a **CEvent** object to notify the copy thread when new data is available, the thread can perform its task as soon as possible.

CEvent objects have two types: manual and automatic. A manual **CEvent** object stays in the state set by SetEvent or ResetEvent until the other function is called. An automatic **CEvent** object automatically returns to a nonsignaled (unavailable) state after at least one thread is released.

To use a **CEvent** object, construct the **CEvent** object when it is needed. Specify the name of the event you wish to wait on, and that your application should initially own it. You can then access the event when the constructor returns. Call SetEvent to signal (make available) the event object and then call Unlock when you are done accessing the controlled resource.

An alternative method for using **CEvent** objects is to add a variable of type **CEvent** as a data member to the class you wish to control. During construction of the controlled object, call the constructor of the **CEvent** data member specifying if the event is initially signaled, the type of event object you want, the name of the event (if it will be used across process boundaries), and desired security attributes.

To access a resource controlled by a **CEvent** object in this manner, first create a variable of either type CSingleLock or type CMultiLock in your resource's access member function. Then call the lock object's **Lock** member function (for example, CMultiLock::Lock). At this point, your thread will either gain access to the resource, wait for the resource to be released and gain access, or wait for the resource to be released and time out, failing to gain access to the resource. In any case, your resource has been accessed in a thread-safe manner. To release the resource, call **SetEvent** to signal the event object, and then use the lock object's **Unlock** member function (for example, CMultiLock::Unlock), or allow the lock object to fall out of scope.

For more information on using **CEvent** objects, see the article Multithreading: How to Use the Synchronization Classes.

Example

```
// The following demonstrates trivial usage of the CEvent class.
// A CEvent object is created and passed as a parameter to another
// thread.  The other thread will wait for the event to be signaled
// and then exit

void CEvent_Test()
{
   // Create the CEvent object that will be passed to the thread routine
   CEvent* pEvent = new CEvent( FALSE, FALSE );

   // Create a thread that will wait on the event
   CWinThread* pThread;
   pThread = ::AfxBeginThread( ThreadProc, pEvent, 0, 0, CREATE_SUSPENDED, NULL);
   pThread->m_bAutoDelete = FALSE;
   pThread->ResumeThread();

   // Signal the thread to do the next work item
   pEvent->SetEvent();

   // Wait for the thread to consume the event and return
   ::WaitForSingleObject( pThread->m_hThread, INFINITE );
   delete pThread;
}

UINT __cdecl ThreadProc( LPVOID lpParameter )
{
   CEvent* pEvent = (CEvent*)( lpParameter );
   VERIFY( pEvent != NULL );
```

```
   // Wait for the event to be signaled
   ::WaitForSingleObject( pEvent->m_hObject, INFINITE );

   // Terminate the thread
    ::AfxEndThread( 0, FALSE );
   return 0L;
}
```

```
// This example builds upon the previous one.
// A second thread is created to calculate prime numbers.
// The main thread will signal the second thread to calulate the next
// prime number in the series.  The second thread signals the first
// after each number is calculated. Finally, after several iterations
// the worker thread is signaled to terminate.

class CPrimeTest
{
public:
   CPrimeTest()
      : m_pCalcNext( new CEvent( FALSE, FALSE ) )
      , m_pCalcFinished( new CEvent( FALSE, FALSE ) )
      , m_pTerminateThread( new CEvent( FALSE, FALSE ) )
      , m_iCurrentPrime( 0 )
   {
      // Create a thread that will calculate the prime numbers
      CWinThread* pThread;
      pThread = ::AfxBeginThread( PrimeCalcProc, this, 0, 0, CREATE_SUSPENDED, NULL);
      pThread->m_bAutoDelete = FALSE;
      pThread->ResumeThread();

      // Calcuate the first 10 prime numbers in the series on the thread
      for( UINT i = 0; i < 10; i++ )
      {
         // Signal the thread to do the next work item
         m_pCalcNext->SetEvent();
         // Wait for the thread to complete the current task
         ::WaitForSingleObject( m_pCalcFinished->m_hObject, INFINITE );
         // Print the result
         TRACE( "The value of m_iCurrentPrime is: %d\n", m_iCurrentPrime );
      }

      // Notify the worker thread to exit and wait for it to complete
      m_pTerminateThread->SetEvent();
      ::WaitForSingleObject( pThread->m_hThread, INFINITE );
      delete pThread;
   }
   ~CPrimeTest()
   {
      delete m_pCalcNext;
      delete m_pCalcFinished;
      delete m_pTerminateThread;
   }

private:
   // Determines whether the given number is a prime number
   static BOOL IsPrime( INT ThisPrime )
   {
      if( ThisPrime < 2 )
         return FALSE;

      for( INT n = 2; n < ThisPrime; n++ )
      {
         if( ThisPrime % n == 0 )
            return FALSE;
      }
      return TRUE;
```

```
    }

    // Calculates the next prime number in the series
    static INT NextPrime( INT ThisPrime )
    {
       while( TRUE )
       {
          if( IsPrime( ++ThisPrime ) )
          {
             return ThisPrime;
          }
       }
    }

    // Worker thread responsible for calculating the next prime
    // number in the series
    static UINT __cdecl PrimeCalcProc( LPVOID lpParameter )
    {
       CPrimeTest* pThis = static_cast<CPrimeTest*>( lpParameter );
       VERIFY( pThis != NULL );

       VERIFY( pThis->m_pCalcNext != NULL );
       VERIFY( pThis->m_pCalcFinished != NULL );
       VERIFY( pThis->m_pTerminateThread != NULL );

       // Create a CMultiLock object to wait on the various events
       // WAIT_OBJECT_0 refers to the first event in the array, WAIT_OBJECT_0+1 refers to th
  e second
       CSyncObject* pWaitObjects[] = { pThis->m_pCalcNext, pThis->m_pTerminateThread };
       CMultiLock MultiLock( pWaitObjects, 2L );
       while( MultiLock.Lock( INFINITE, FALSE ) == WAIT_OBJECT_0 )
       {
          // Calculate next prime
          pThis->m_iCurrentPrime = NextPrime( pThis->m_iCurrentPrime );
          // Notify main thread calculation is complete
          pThis->m_pCalcFinished->SetEvent();
        }

       // Terminate the thread
        ::AfxEndThread( 0, FALSE );
       return 0L;
    }

    CEvent* m_pCalcNext;      // notifies worker thread to calculate next prime
    CEvent* m_pCalcFinished;   // notifies main thread current calculation is complete
    CEvent* m_pTerminateThread;   // notifies worker thread to terminate

    INT m_iCurrentPrime;   // current calculated prime number
  };
```

Requirements

Header: afxmt.h

See Also

Reference

CSyncObject Class

Hierarchy Chart

Other Resources

CEvent Members

CEvent Members

Base Class Members

CObject Members

CSyncObject Members

Construction

CEvent	Constructs a **CEvent** object.

Methods

PulseEvent	Sets the event to available (signaled), releases waiting threads, and sets the event to unavailable (nonsignaled).
ResetEvent	Sets the event to unavailable (nonsignaled).
SetEvent	Sets the event to available (signaled) and releases any waiting threads.
Unlock	Releases the event object.

See Also

Reference

CEvent Class

Hierarchy Chart

CEvent Member Functions

For information about the member functions in **CEvent**, see CEvent Members.

CEvent::CEvent

Constructs a named or unnamed **CEvent** object.

```
CEvent(
   BOOL bInitiallyOwn = FALSE,
   BOOL bManualReset = FALSE,
   LPCTSTR lpszName = NULL,
   LPSECURITY_ATTRIBUTES lpsaAttribute = NULL
);
```

Parameters

bInitiallyOwn

If **TRUE**, the thread for the **CMultilock** or **CSingleLock** object is enabled. Otherwise, all threads wanting to access the resource must wait.

bManualReset

If **TRUE**, specifies that the event object is a manual event, otherwise the event object is an automatic event.

lpszName

Name of the **CEvent** object. Must be supplied if the object will be used across process boundaries. If the name matches an existing event, the constructor builds a new **CEvent** object which references the event of that name. If the name matches an existing synchronization object that is not an event, the construction will fail. If **NULL**, the name will be null.

lpsaAttribute

Security attributes for the event object. For a full description of this structure, see SECURITY_ATTRIBUTES in the *Platform SDK*.

Remarks

To access or release a **CEvent** object, create a CMultiLock or CSingleLock object and call its Lock and Unlock member functions.

To change the state of a **CEvent** object to signaled (threads do not have to wait), call SetEvent or PulseEvent. To set the state of a **CEvent** object to nonsignaled (threads must wait), call ResetEvent.

Security Note
After creating the **CEvent** object, use GetLastError to ensure that the mutex didn't already exist. If the mutex did exist unexpectedly, it may indicate a rogue process is squatting and may be intending to use the mutex maliciously. In this case, the recommended security-conscious procedure is to close the handle and continue as if there was a failure in creating the object.

See Also

Reference

CEvent Class

Hierarchy Chart

Other Resources

CEvent Members

CEvent::PulseEvent

Sets the state of the event to signaled (available), releases any waiting threads, and resets it to nonsignaled (unavailable) automatically.

```
BOOL PulseEvent( );
```

Return Value

Nonzero if the function was successful; otherwise 0.

Remarks

If the event is manual, all waiting threads are released, the event is set to nonsignaled, and **PulseEvent** returns. If the event is automatic, a single thread is released, the event is set to nonsignaled, and **PulseEvent** returns.

If no threads are waiting, or no threads can be released immediately, **PulseEvent** sets the state of the event to nonsignaled and returns.

PulseEvent uses the underlying Win32 **PulseEvent** function, which can be momentarily removed from the wait state by a kernel-mode asynchronous procedure call. Therefore, **PulseEvent** is unreliable and should not be used by new applications. For more information, see the PulseEvent function.

See Also

Reference

CEvent Class

Hierarchy Chart

Other Resources

CEvent Members

CEvent::ResetEvent

Sets the state of the event to nonsignaled until explicitly set to signaled by the SetEvent member function.

```
BOOL ResetEvent( );
```

Return Value

Nonzero if the function was successful; otherwise 0.

Remarks

This causes all threads wishing to access this event to wait.

This member function is not used by automatic events.

See Also

Reference

CEvent Class

Hierarchy Chart

Other Resources

CEvent Members

CEvent::SetEvent

Sets the state of the event to signaled, releasing any waiting threads.

```
BOOL SetEvent( );
```

Return Value

Nonzero if the function was successful, otherwise 0.

Remarks

If the event is manual, the event will remain signaled until ResetEvent is called. More than one thread can be released in this case. If the event is automatic, the event will remain signaled until a single thread is released. The system will then set the state of the event to nonsignaled. If no threads are waiting, the state remains signaled until one thread is released.

See Also

Reference

CEvent Class

Hierarchy Chart

Other Resources

CEvent Members

CEvent::Unlock

Releases the event object.

```
BOOL Unlock( );
```

Return Value

Nonzero if the thread owned the event object and the event is an automatic event; otherwise 0.

Remarks

This member function is called by threads that currently own an automatic event to release it after they are done, if their lock object is to be reused. If the lock object is not to be reused, this function will be called by the lock object's destructor.

See Also

Reference

CEvent Class

Hierarchy Chart

Other Resources

CEvent Members

CException Class

The base class for all exceptions in the Microsoft Foundation Class Library.

```
class AFX_NOVTABLE CException : public CObject
```

Remarks

Because **CException** is an abstract base class you cannot create **CException** objects directly; you must create objects of derived classes. If you need to create your own **CException**-style class, use one of the derived classes listed above as a model. Make sure that your derived class also uses **IMPLEMENT_DYNAMIC**.

The derived classes and their descriptions are listed below:

CSimpleException	A base class for resource-critical MFC exceptions
CInvalidArgException	Invalid argument exception condition
CMemoryException	Out-of-memory exception
CNotSupportedException	Request for an unsupported operation
CArchiveException	Archive-specific exception
CFileException	File-specific exception
CResourceException	Windows resource not found or not createable
COleException	OLE exception
CDBException	Database exception (that is, exception conditions arising for MFC database classes based on Open D atabase Connectivity)
COleDispatchException	OLE dispatch (automation) exception
CUserException	Exception that indicates that a resource could not be found
CDaoException	Data access object exception (that is, exception conditions arising for DAO classes)
CInternetException	Internet exception (that is, exception conditions arising for Internet classes).

These exceptions are intended to be used with the THROW, THROW_LAST, TRY, CATCH, AND_CATCH, and END_CATCH macros. For more information on exceptions, see Exception Processing, or see the article Exception Handling (MFC).

To catch a specific exception, use the appropriate derived class. To catch all types of exceptions, use **CException**, and then use CObject::IsKindOf to differentiate among **CException**-derived classes. Note that **CObject::IsKindOf** works only for classes declared with the IMPLEMENT_DYNAMIC macro, in order to take advantage of dynamic type checking. Any **CException**-derived class that you create should use the **IMPLEMENT_DYNAMIC** macro, too.

You can report details about exceptions to the user by calling GetErrorMessage or ReportError, two member functions that work with any of **CException**'s derived classes.

If an exception is caught by one of the macros, the **CException** object is deleted automatically; do not delete it yourself. If an exception is caught by using a **catch** keyword, it is not automatically deleted. See the article Exception Handling (MFC) for more information about when to delete an exeption object.

Requirements

Header: afx.h

See Also

Reference

CObject Class

Hierarchy Chart

Other Resources

CException Members

Exception Processing

CException Members

Base Class Members

CObject Members

Operations

CException	Constructs a **CException** object.
Delete	Deletes a **CException** object.
GetErrorMessage	Retrieves the message describing an exception.
ReportError	Reports an error message in a message box to the user.

See Also

Reference

CException Class
Hierarchy Chart

CException Member Functions

For information about the member functions in **CException**, see CException Members.

CException::CException

This member function constructs a **CException** object.

```
explicit CException(
   BOOL bAutoDelete
);
```

Parameters

b_AutoDelete

Specify **TRUE** if the memory for the **CException** object has been allocated on the heap. This will cause the **CException** object to be deleted when the **Delete** member function is called to delete the exception. Specify **FALSE** if the **CException** object is on the stack or is a global object. In this case, the **CException** object will not be deleted when the **Delete** member function is called.

Remarks

You would normally never need to call this constructor directly. A function that throws an exception should create an instance of a **CException**-derived class and call its constructor, or it should use one of the MFC throw functions, such as AfxThrowFileException, to throw a predefined type. This documentation is provided only for completeness.

See Also

Reference

CException Class

Hierarchy Chart

Other Resources

CException Members

CException::Delete

This function checks to see if the **CException** object was created on the heap, and if so, it calls the **delete** operator on the object.

```
void Delete( );
```

Remarks

When deleting a **CException** object, use the **Delete** member function to delete the exception. Do not use the **delete** operator directly, because the **CException** object may be a global object or have been created on the stack.

You can specify whether the object should be deleted when the object is constructed. For more information, see CException::CException.

You only need to call **Delete** if you are using the C++ **try**-**catch** mechanism. If you are using the MFC macros **TRY** and **CATCH**, then these macros will automatically call this function.

Example

```
CFile* pFile = NULL;

   // Constructing a CFile object with this override may throw
   // a CFile exception, and won't throw any other exceptions.
   // Calling CString::Format() may throw a CMemoryException,
   // so we have a catch block for such exceptions, too. Any
   // other exception types this function throws will be
   // routed to the calling function.

   // Note that this example performs the same actions as the
   // example for CATCH, but uses C++ try/catch syntax instead
   // of using the MFC TRY/CATCH macros. This sample must use
   // CException::Delete() to delete the exception objects
   // before closing the catch block, while the CATCH example
   // implicitly performs the deletion via the macros.

   try
   {
      pFile = new CFile(_T("C:\\WINDOWS\\SYSTEM.INI"),
         CFile::modeRead | CFile::shareDenyNone);

      DWORD dwLength = pFile->GetLength();

      CString str;
      str.Format(_T("Your SYSTEM.INI file is %u bytes long."),
         dwLength);

      AfxMessageBox(str);
   }
   catch(CFileException* pEx)
   {
      // Simply show an error message to the user.

      pEx->ReportError();
      pEx->Delete();
   }
   catch(CMemoryException* pEx)
   {
      // We can't recover from this memory exception, so we'll
      // just terminate the app without any cleanup. Normally, an
      // an application should do everything it possibly can to
      // clean up properly and _not_ call AfxAbort().

      pEx->Delete();
      AfxAbort();
```

```
    }

    // If an exception occurrs in the CFile constructor,
    // the language will free the memory allocated by new
    // and will not complete the assignment to pFile.
    // Thus, our clean-up code needs to test for NULL.

    if (pFile != NULL)
    {
       pFile->Close();
       delete pFile;
    }
```

See Also

Reference

CException Class

Hierarchy Chart

Other Resources

CException Members

CException::GetErrorMessage

Call this member function to provide text about an error that has occurred.

```
virtual BOOL GetErrorMessage(
   LPTSTR lpszError,
   UINT nMaxError,
   PUINT pnHelpContext = NULL
);
```

Parameters

lpszError

A pointer to a buffer that will receive an error message.

nMaxError

The maximum number of characters the buffer can hold, including the **NULL** terminator.

pnHelpContext

The address of a **UINT** that will receive the help context ID. If **NULL**, no ID will be returned.

Return Value

Nonzero if the function is successful; otherwise 0 if no error message text is available.

Remarks

For example, call **GetErrorMessage** to retrieve a string describing the error which caused MFC to throw a **CFileException** when writing to a CFile object.

Note

GetErrorMessage will not copy more than *nMaxError -1* characters to the buffer, and it will always add a trailing null to end the string. If the buffer is too small, the error message may be truncated.

Example

Here is an example of the use of **CException::GetErrorMessage**.

```
    CFile fileInput;
    CFileException ex;

    // try to open a file for reading.
    // The file will certainly not
    // exist because there are too many explicit
    // directories in the name.

    // if the call to Open() fails, ex will be
    // initialized with exception
    // information.  the call to ex.GetErrorMessage()
    // will retrieve an appropriate message describing
    // the error, and we'll add our own text
    // to make sure the user is perfectly sure what
    // went wrong.

    if (!fileInput.Open("\\Too\\Many\\Bad\\Dirs.DAT", CFile::modeRead, &ex))
    {
       TCHAR   szCause[255];
       CString strFormatted;

       ex.GetErrorMessage(szCause, 255);

       // (in real life, it's probably more
```

```
        // appropriate to read this from
        //  a string resource so it would be easy to
        // localize)

        strFormatted = _T("The data file could not be opened because of this error: ");
        strFormatted += szCause;

        AfxMessageBox(strFormatted);
     }
     else
     {
        // the file was opened, so do whatever work
        // with fileInput
        // we were planning...
        // :

        fileInput.Close();
     }
```

See Also

Reference

CException Class

Hierarchy Chart

CException::ReportError

Other Resources

CException Members

CException::ReportError

Call this member function to report error text in a message box to the user.

```
virtual int ReportError(
   UINT nType = MB_OK,
   UINT nMessageID = 0
);
```

Parameters

nType

Specifies the style of the message box. Apply any combination of the message-box styles to the box. If you don't specify this parameter, the default is **MB_OK**.

nMessageID

Specifies the resource ID (string table entry) of a message to display if the exception object does not have an error message. If 0, the message "No error message is available" is displayed.

Return Value

An **AfxMessageBox** value; otherwise 0 if there is not enough memory to display the message box. See AfxMessageBox for the possible return values.

Example

Here is an example of the use of **CException::ReportError**. For another example, see the example for CATCH.

```
    CFile fileInput;
    CFileException ex;

    // try to open a file for reading.
    // The file will certainly not
    // exist because there are too many explicit
    // directories in the name.

    // if the call to Open() fails, ex will be
    // initialized with exception
    // information.  the call to ex.ReportError() will
    // display an appropriate
    // error message to the user, such as
    // "\Too\Many\Bad\Dirs.DAT contains an
    // invalid path."  The error message text will be
    // appropriate for the
    // file name and error condition.

    if (!fileInput.Open("\\Too\\Many\\Bad\\Dirs.DAT", CFile::modeRead, &ex))
    {
       ex.ReportError();
    }
    else
    {
       // the file was opened, so do whatever work
       // with fileInput we were planning...
       // :

       fileInput.Close();
    }
```

See Also

Reference

CException Class

Hierarchy Chart

AfxMessageBox
CException::GetErrorMessage

Other Resources

CException Members

CFieldExchange Class

Supports the record field exchange (RFX) and bulk record field exchange (Bulk RFX) routines used by the database classes.

```
class CFieldExchange
```

Remarks

CFieldExchange does not have a base class.

Use this class if you are writing data exchange routines for custom data types or when you are implementing bulk row fetching; otherwise, you will not directly use this class. RFX and Bulk RFX exchanges data between the field data members of your recordset object and the corresponding fields of the current record on the data source.

Note
If you are working with the Data Access Objects (DAO) classes rather than the Open Database Connectivity (ODBC) classes, use class CDaoFieldExchange instead. For more information, see the article Overview:Database Programming.

A **CFieldExchange** object provides the context information needed for record field exchange or bulk record field exchange to take place. **CFieldExchange** objects support a number of operations, including binding parameters and field data members and setting various flags on the fields of the current record. RFX and Bulk RFX operations are performed on recordset-class data members of types defined by the **enum FieldType** in **CFieldExchange**. Possible **FieldType** values are:

- **CFieldExchange::outputColumn** for field data members.
- **CFieldExchange::inputParam** or **CFieldExchange::param** for input parameter data members.
- **CFieldExchange::outputParam** for output parameter data members.
- **CFieldExchange::inoutParam** for input/output parameter data members.

Most of the class's member functions and data members are provided for writing your own custom RFX routines. You will use **SetFieldType** frequently. For more information, see the articles Record Field Exchange (RFX) and Recordset (ODBC). For information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC). For details about the RFX and Bulk RFX global functions, see Record Field Exchange Functions in the MFC Macros and Globals section of this reference.

Requirements

Header: afxdb.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart

CRecordset Class

Other Resources

CFieldExchange Members

CFieldExchange Members

Operations

IsFieldType	Returns nonzero if the current operation is appropriate for the type of field being updated.
SetFieldType	Specifies the type of recordset data member — column or parameter — represented by all following calls to RFX functions until the next call to **SetFieldType**.

See Also

Reference

CFieldExchange Class

Hierarchy Chart

CFieldExchange Member Functions

For information about the member functions in **CFieldExchange**, see CFieldExchange Members.

CFieldExchange::IsFieldType

If you write your own RFX function, call **IsFieldType** at the beginning of your function to determine whether the current operation can be performed on a particular field or parameter data member type (a **CFieldExchange::outputColumn**, **CFieldExchange::inputParam**, **CFieldExchange::param**, **CFieldExchange::outputParam**, or **CFieldExchange::inoutParam**).

```
BOOL IsFieldType(
   UINT* pnField
);
```

Parameters

pnField

The sequential number of the field or parameter data member is returned in this parameter. This number corresponds to the data member's order in the CRecordset::DoFieldExchange or CRecordset::DoBulkFieldExchange function.

Return Value

Nonzero if the current operation can be performed on the current field or parameter type.

Remarks

Follow the model of the existing RFX functions.

See Also

Reference

CFieldExchange Class

Hierarchy Chart

Other Resources

CFieldExchange Members

CFieldExchange::SetFieldType

You need a call to **SetFieldType** in your recordset class's DoFieldExchange or DoBulkFieldExchange override.

```
void SetFieldType(
   UINT nFieldType
);
```

Parameters

nFieldType

A value of the **enum FieldType**, declared in **CFieldExchange**, which can be one of the following:

- **CFieldExchange::outputColumn**
- **CFieldExchange::inputParam**
- **CFieldExchange::param**
- **CFieldExchange::outputParam**
- **CFieldExchange::inoutParam**

Remarks

For field data members, you must call **SetFieldType** with a parameter of **CFieldExchange::outputColumn**, followed by calls to the RFX or Bulk RFX functions. If you have not implemented bulk row fetching, then ClassWizard places this **SetFieldType** call for you in the field map section of **DoFieldExchange**.

If you parameterize your recordset class, you must call **SetFieldType** again, outside any field map section, followed by RFX calls for all the parameter data members. Each type of parameter data member must have its own **SetFieldType** call. The following table distinguishes the different values you can pass to **SetFieldType** to represent the parameter data members of your class:

SetFieldType parameter value	Type of parameter data member
CFieldExchange::inputParam	Input parameter. A value that is passed into the recordset's query or stored procedure.
CFieldExchange::param	Same as **CFieldExchange::inputParam**.
CFieldExchange::outputParam	Output parameter. A return value of the recordset's stored procedure.
CFieldExchange::inoutParam	Input/output parameter. A value that is passed into and returned from the recordset's stored procedure.

In general, each group of RFX function calls associated with field data members or parameter data members must be preceded by a call to **SetFieldType**. The *nFieldType* parameter of each **SetFieldType** call identifies the type of the data members represented by the RFX function calls that follow the **SetFieldType** call.

For more information about handling output and input/output parameters, see the **CRecordset** member function FlushResultSet. For more information about the RFX and Bulk RFX functions, see the topic Record Field Exchange Functions. For related information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

Example

This example shows several calls to RFX functions with accompanying calls to **SetFieldType**. Note that **SetFieldType** is called through the *pFX* pointer to a **CFieldExchange** object.

```
void CSections::DoFieldExchange( CFieldExchange* pFX )
{
```

```
    //{{AFX_FIELD_MAP(CSections)
    pFX->SetFieldType( CFieldExchange::outputColumn );
    RFX_Text( pFX, "CourseID", m_strCourseID );
    RFX_Text( pFX, "InstructorID", m_strInstructorID );
    RFX_Text( pFX, "RoomNo", m_strRoomNo );
    RFX_Text( pFX, "Schedule", m_strSchedule );
    //}}AFX_FIELD_MAP

    // output parameter
    pFX->SetFieldType( CFieldExchange::outputParam );
    RFX_Long( pFX, "Instructor_Count", m_nCountParam );

    // input parameter
    pFX->SetFieldType( CFieldExchange::inputParam );
    RFX_Text( pFX, "Department_Name", m_strNameParam );
}
```

See Also

Reference

CFieldExchange Class
Hierarchy Chart
CRecordset::DoFieldExchange
CRecordset::DoBulkFieldExchange
CRecordset::FlushResultSet

Other Resources

CFieldExchange Members
Record Field Exchange Functions

CFile Class

The base class for Microsoft Foundation Class file classes.

```
class CFile : public CObject
```

Remarks

It directly provides unbuffered, binary disk input/output services, and it indirectly supports text files and memory files through its derived classes. **CFile** works in conjunction with the **CArchive** class to support serialization of Microsoft Foundation Class objects.

The hierarchical relationship between this class and its derived classes allows your program to operate on all file objects through the polymorphic **CFile** interface. A memory file, for example, behaves like a disk file.

Use **CFile** and its derived classes for general-purpose disk I/O. Use **ofstream** or other Microsoft iostream classes for formatted text sent to a disk file.

Normally, a disk file is opened automatically on **CFile** construction and closed on destruction. Static member functions permit you to interrogate a file's status without opening the file.

For more information on using **CFile**, see the articles Files in MFC and File Handling in the *Run-Time Library Reference*.

Requirements

Header: afx.h

See Also

Tasks

DRAWCLI Sample: Illustrates Integrating Active Container Support with Application-Specific Features

Reference

CObject Class
Hierarchy Chart
CStdioFile Class
CMemFile Class

Other Resources

CFile Members

CFile Members

Base Class Members

CObject Members

Data Members

m_hFile	Usually contains the operating-system file handle.
CFile::hFileNull	Determines if the **CFile** object has a valid handle.

Operators

operator HANDLE	A handle to a **CFile** object.

Construction

Abort	Closes a file ignoring all warnings and errors.
CFile	Constructs a **CFile** object from a path or file handle.
Close	Closes a file and deletes the object.
Duplicate	Constructs a duplicate object based on this file.
Open	Safely opens a file with an error-testing option.

Input/Output

Flush	Flushes any data yet to be written.
Read	Reads (unbuffered) data from a file at the current file position.
Write	Writes (unbuffered) data in a file to the current file position.

Position

GetLength	Retrieves the length of the file.
Seek	Positions the current file pointer.
SeekToBegin	Positions the current file pointer at the beginning of the file.
SeekToEnd	Positions the current file pointer at the end of the file.
SetLength	Changes the length of the file.

Locking

LockRange	Locks a range of bytes in a file.
UnlockRange	Unlocks a range of bytes in a file.

Status

GetFileName	Retrieves the filename of the selected file.
GetFilePath	Retrieves the full file path of the selected file.

GetFileTitle	Retrieves the title of the selected file.
GetPosition	Retrieves the current file pointer.
GetStatus	Retrieves the status of this open file.
SetFilePath	Sets the full file path of the selected file.

Static

GetStatus	Retrieves the status of the specified file (static, virtual function).
Remove	Deletes the specified file (static function).
Rename	Renames the specified file (static function).
SetStatus	Sets the status of the specified file (static, virtual function).

See Also

Reference

CFile Class

Hierarchy Chart

CFile Member Functions

For information about the member functions in **CFile**, see CFile Members.

CFile::Abort

Closes the file associated with this object and makes the file unavailable for reading or writing.

```
virtual void Abort( );
```

Remarks

If you have not closed the file before destroying the object, the destructor closes it for you.

When handling exceptions, **CFile::Abort** differs from **CFile::Close** in two important ways. First, the **Abort** function will not throw an exception on failures because failures are ignored by **Abort**. Second, **Abort** will not **ASSERT** if the file has not been opened or was closed previously.

If you used **new** to allocate the **CFile** object on the heap, then you must delete it after closing the file. **Abort** sets **m_hFile** to **CFile::hFileNull**.

Example

```
//example for CFile::Abort
CStdioFile fileTest;
char* pFileName = "test.dat";
TRY
{
   // do stuff that may throw exceptions
   fileTest.Open( pFileName, CFile::modeWrite );
}
CATCH_ALL( e )
{
   fileTest.Abort();   // close file safely and quietly
   THROW_LAST();
}
END_CATCH_ALL
```

See Also

Reference

CFile Class
Hierarchy Chart
CFile::Close
CFile::Open

Other Resources

CFile Members

CFile::CFile

The default constructor does not open a file but rather sets **m_hFile** to **CFile::hFileNull**.

```
CFile( );
CFile(
   HANDLE hFile
);
CFile(
   LPCTSTR lpszFileName,
   UINT nOpenFlags
);
```

Parameters

hFile

The **HFILE** returned from a previous call to CreateFile, as described in the *Platform SDK*.

lpszFileName

A string that is the path to the desired file. The path can be relative or absolute.

nOpenFlags

Sharing and access mode. Specifies the action to take when opening the file. You can combine options listed below by using the bitwise-OR (**|**) operator. One access permission and one share option are required; the **modeCreate** and **modeNoInherit** modes are optional. The values are as follows:

- **CFile::modeCreate** Directs the constructor to create a new file. If the file exists already, it is truncated to 0 length.
- **CFile::modeNoTruncate** Combine this value with **modeCreate**. If the file being created already exists, it is not truncated to 0 length. Furthermore, if the file being created already exists, the underlying file pointer will point to the beginning of the file. This flag guarantees the file to open, either as a newly created file or as an existing file. This might be useful, for example, when opening a settings file that may or may not exist already. This option applies to **CStdioFile** as well.
- **CFile::modeRead** Opens the file for reading only.
- **CFile::modeReadWrite** Opens the file for reading and writing.
- **CFile::modeWrite** Opens the file for writing only.
- **CFile::modeNoInherit** Prevents the file from being inherited by child processes.
- **CFile::shareDenyNone** Opens the file without denying other processes read or write access to the file. **Create** fails if the file has been opened in compatibility mode by any other process.
- **CFile::shareDenyRead** Opens the file and denies other processes read access to the file. **Create** fails if the file has been opened in compatibility mode or for read access by any other process.
- **CFile::shareDenyWrite** Opens the file and denies other processes write access to the file. **Create** fails if the file has been opened in compatibility mode or for write access by any other process.
- **CFile::shareExclusive** Opens the file with exclusive mode, denying other processes both read and write access to the file. Construction fails if the file has been opened in any other mode for read or write access, even by the current process.
- **CFile::shareCompat** This flag is not available in 32-bit and 64-bit MFC. This flag maps to **CFile::shareExclusive** when used in **CFile::Open**.
- **CFile::typeText** Sets text mode with special processing for carriage return–linefeed pairs (used in derived classes only).
- **CFile::typeBinary** Sets binary mode (used in derived classes only).
- **CFile::osNoBuffer** The system opens a file with no system caching. For more information see

FILE_FLAG_NO_BUFFERING in CreateFile in the *Platform SDK*.

- **CFile::osWriteThrough** The system writes through any intermediate cache and goes directly to disk. For more information see **FILE_FLAG_WRITE_THROUGH** in **CreateFile** in the *Platform SDK*.
- **CFile::osRandomAccess** A file is accessed randomly. The system can use this as a hint to optimize file caching.
- **CFile::osSequentialScan** A file is accessed sequentially from beginning to end. The system can use this as a hint to optimize file caching. For more information see **FILE_FLAG_SEQUENTIAL_SCAN** in **CreateFile** in the *Platform SDK*.

Remarks

Because this constructor does not throw an exception, it does not make sense to use **TRY**/**CATCH** logic. Use the **Open** member function, then test directly for exception conditions. For a discussion of exception-processing strategy, see the article Exceptions.

The constructor with one argument creates a **CFile** object that corresponds to an existing operating-system file identified by *hFile*. No check is made on the access mode or file type. When the **CFile** object is destroyed, the operating-system file will not be closed. You must close the file yourself.

The constructor with two arguments creates a **CFile** object and opens the corresponding operating-system file with the given path. This constructor combines the functions of the first constructor and the **Open** member function. It throws an exception if there is an error while opening the file. Generally, this means that the error is unrecoverable and that the user should be alerted.

Example

For an example using the default constructor, see CFile::Open. For an example using the constructor which throws exceptions, see CATCH.

The HANDLE constructor can be used as in the following code, which creates a file named C:\MyFile.DAT and writes the ANSI characters "Hockey is best!":

```
HANDLE hFile = CreateFile(_T("C:\\MyFile.DAT"),
         GENERIC_WRITE, FILE_SHARE_READ,
         NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

   if (hFile == INVALID_HANDLE_VALUE)
      AfxMessageBox(_T("Couldn't create the file!"));
   else
   {
      // Attach a CFile object to the handle we have.
      CFile myFile(hFile);

      static const char sz[] = "Hockey is best!";

      // write string, without null-terminator
      myFile.Write(sz, lstrlen(sz));

      // We can call Close() explicitly, but the destructor would have
      // also closed the file for us. Note that there's no need to
      // call the CloseHandle() on the handle returned by the API because
      // MFC will close it for us.
      myFile.Close();
   }
```

See Also

Reference

CFile Class

Hierarchy Chart

Other Resources

CFile Members

CFile::Close

Closes the file associated with this object and makes the file unavailable for reading or writing.

```
virtual void Close( );
```

Remarks

If you have not closed the file before destroying the object, the destructor closes it for you.

If you used **new** to allocate the **CFile** object on the heap, then you must delete it after closing the file. **Close** sets **m_hFile** to **CFile::hFileNull**.

Example

See the example for CFile::CFile.

See Also

Reference

CFile Class

Hierarchy Chart

CFile::Open

Other Resources

CFile Members

CFile::Duplicate

Constructs a duplicate **CFile** object for a given file.

```
virtual CFile* Duplicate( ) const;
```

Return Value

A pointer to a duplicate **CFile** object.

Remarks

This is equivalent to the C run-time function **_dup**.

See Also

Reference

CFile Class

Hierarchy Chart

Other Resources

CFile Members

CFile::Flush

Forces any data remaining in the file buffer to be written to the file.

```
virtual void Flush( );
```

Remarks

The use of **Flush** does not guarantee flushing of **CArchive** buffers. If you are using an archive, call CArchive::Flush first.

Example

See the example for CFile::SetFilePath.

See Also

Reference

CFile Class

Hierarchy Chart

Other Resources

CFile Members

CFile::GetFileName

Call this member function to retrieve the name of a specified file.

```
virtual CString GetFileName( ) const;
```

Return Value

The name of the file.

Remarks

For example, when you call **GetFileName** to generate a message to the user about the file `c:\windows\write\myfile.wri`, the filename, `myfile.wri`, is returned.

To return the entire path of the file, including the name, call GetFilePath. To return the title of the file (`myfile`), call GetFileTitle.

Example

This code fragment opens the SYSTEM.INI file in your WINDOWS directory. If found, the example will print out the name and path and title, as shown under Output:

```
{
    // ...
    try
    {
        // try to open the file
        CFile sysFile(_T("C:\\WINDOWS\\SYSTEM.INI"), CFile::modeRead);

        // print out path name and title information
        _tprintf_s(_T("Path is : \"%s\"\n"),
                   (LPCTSTR) sysFile.GetFilePath());
        _tprintf_s(_T("Name is : \"%s\"\n"),
                   (LPCTSTR) sysFile.GetFileName());
        _tprintf_s(_T("Title is: \"%s\"\n"),
                   (LPCTSTR) sysFile.GetFileTitle());

        // close the file handle
        sysFile.Close();
    }
    catch (CFileException* pEx)
    {
        // if an error occurs, just make a message box
        pEx->ReportError();
        pEx->Delete();
    }

    // ...
}
```

Output

```
Path is : "C:\WINDOWS\SYSTEM.INI"
Name is : "SYSTEM.INI"
Title is: "System"
```

See Also

Reference

CFile Class

Hierarchy Chart

CFile::GetFilePath

CFile::GetFileTitle

Other Resources

CFile Members

CFile::GetFilePath

Call this member function to retrieve the full path of a specified file.

```
virtual CString GetFilePath( ) const;
```

Return Value

The full path of the specified file.

Remarks

For example, when you call **GetFilePath** to generate a message to the user about the file `c:\windows\write\myfile.wri`, the file path, `c:\windows\write\myfile.wri`, is returned.

To return just the name of the file (`myfile.wri`), call GetFileName. To return the title of the file (`myfile`), call GetFileTitle.

Example

See the example for GetFileName.

See Also

Reference

CFile Class
Hierarchy Chart
CFile::SetFilePath
CFile::GetFileTitle
CFile::GetFileName

Other Resources

CFile Members

CFile::GetFileTitle

Call this member function to retrieve the file title for a specified file.

```
virtual CString GetFileTitle( ) const;
```

Return Value

The title of the specified file.

Remarks

For example, when you call **GetFileTitle** to generate a message to the user about the file `c:\windows\write\myfile.wri`, the file title (`myfile`) is returned.

Note
In Windows 95/98, the file title typically does not include the extension. For a explanation of this, see GetFileTitle in the *Platform SDK*.

To return the entire path of the file, including the name, call GetFilePath. To return just the name of the file (`myfile.wri`), call GetFileName.

Example

See the example for GetFileName.

See Also

Reference

CFile Class
Hierarchy Chart
CFile::GetFileName
CFile::GetFilePath
GetFileTitle

Other Resources

CFile Members

CFile::GetLength

Obtains the current logical length of the file in bytes.

```
virtual ULONGLONG GetLength( ) const;
```

Return Value

The length of the file.

Example

```
CFile* pFile = NULL;
   // Constructing a CFile object with this override may throw
   // a CFile exception, and won't throw any other exceptions.
   // Calling CString::Format() may throw a CMemoryException,
   // so we have a catch block for such exceptions, too. Any
   // other exception types this function throws will be
   // routed to the calling function.
   TRY
   {
      pFile = new CFile(_T("C:\\WINDOWS\\SYSTEM.INI"),
         CFile::modeRead | CFile::shareDenyNone);
      ULONGLONG dwLength = pFile->GetLength();
      CString str;
      str.Format(_T("Your SYSTEM.INI file is %I64u bytes long."), dwLength);
      AfxMessageBox(str);
   }
   CATCH(CFileException, pEx)
   {
      // Simply show an error message to the user.
      pEx->ReportError();
   }
   AND_CATCH(CMemoryException, pEx)
   {
      // We can't recover from this memory exception, so we'll
      // just terminate the app without any cleanup. Normally, an
      // an application should do everything it possibly can to
      // clean up properly and _not_ call AfxAbort().
      AfxAbort();
   }
   END_CATCH
   // If an exception occurs in the CFile constructor,
   // the language will free the memory allocated by new
   // and will not complete the assignment to pFile.
   // Thus, our clean-up code needs to test for NULL.
   if (pFile != NULL)   {
      pFile->Close();
      delete pFile;
   }
```

See Also

Reference

CFile Class

Hierarchy Chart

CFile::SetLength

Other Resources

CFile Members

CFile::GetPosition

Obtains the current value of the file pointer, which can be used in subsequent calls to **Seek**.

```
virtual ULONGLONG GetPosition( ) const;
```

Return Value

The file pointer.

Example

```
//example for CFile::GetPosition
extern CFile cfile;
ULONGLONG dwPosition = cfile.GetPosition();
```

See Also

Reference

CFile Class

Hierarchy Chart

Other Resources

CFile Members

CFile::GetStatus

This method retrieves status information related to a given **CFile** object instance or a given file path.

```
BOOL GetStatus(
   CFileStatus& rStatus
) const;
static BOOL PASCAL GetStatus(
   LPCTSTR lpszFileName,
   CFileStatus& rStatus
);
```

Parameters

rStatus

A reference to a user-supplied **CFileStatus** structure that will receive the status information. The **CFileStatus** structure has the following fields:

- **CTime m_ctime** The date and time the file was created.
- **CTime m_mtime** The date and time the file was last modified.
- **CTime m_atime** The date and time the file was last accessed for reading.
- **ULONGLONG m_size** The logical size of the file in bytes, as reported by the DIR command.
- **BYTE m_attribute** The attribute byte of the file.
- **char m_szFullName[_MAX_PATH]** The absolute filename in the Windows character set.

lpszFileName

A string in the Windows character set that is the path to the desired file. The path can be relative or absolute, or it can contain a network path name.

Return Value

TRUE if the status information for the specified file is successfully obtained; otherwise, **FALSE**.

Remarks

The non-static version of **GetStatus** retrieves status information of the open file associated with the given **CFile** object. The static version of **GetStatus** obtains the file status from a given file path without actually opening the file. This is useful for testing the existence and access rights of a file.

The **m_attribute** member of the **CFileStatus** structure refers to the file attribute set. The **CFile** class provides the **Attribute** enumeration type so file attributes can be specified symbolically:

```
enum Attribute {
   normal =    0x00,
   readOnly =  0x01,
   hidden =    0x02,
   system =    0x04,
   volume =    0x08,
   directory = 0x10,
   archive =   0x20
   };
```

Example

```
//example for CFile::GetStatus
CFileStatus status;
extern CFile cfile;
if( cfile.GetStatus( status ) )    // non-static version
   {
```

```
      #ifdef _DEBUG
         afxDump << "File size = " << status.m_size << "\n";
      #endif
   }
char* pFileName = "test.dat";
if( CFile::GetStatus( pFileName, status ) )   // static version
   {
      #ifdef _DEBUG
         afxDump << "Full file name = " << status.m_szFullName << "\n";
      #endif
   }
```

See Also

Reference

CFile Class

Hierarchy Chart

CFile::SetStatus

CTime Class

Other Resources

CFile Members

CFile::LockRange

Locks a range of bytes in an open file, throwing an exception if the file is already locked.

```
virtual void LockRange(
   ULONGLONG dwPos,
   ULONGLONG dwCount
);
```

Parameters

dwPos

The byte offset of the start of the byte range to lock.

dwCount

The number of bytes in the range to lock.

Remarks

Locking bytes in a file prevents access to those bytes by other processes. You can lock more than one region of a file, but no overlapping regions are allowed.

When you unlock the region, using the **UnlockRange** member function, the byte range must correspond exactly to the region that was previously locked. The **LockRange** function does not merge adjacent regions; if two locked regions are adjacent, you must unlock each region separately.

Note
This function is not available for the **CMemFile**-derived class.

Example

```
//example for CFile::LockRange
extern ULONGLONG dwPos;
extern ULONGLONG dwCount;
extern CFile cfile;
cfile.LockRange( dwPos, dwCount );
```

See Also

Reference

CFile Class

Hierarchy Chart

CFile::UnlockRange

Other Resources

CFile Members

CFile::Open

Open is designed for use with the default **CFile** constructor.

```
virtual BOOL Open(
   LPCTSTR lpszFileName,
   UINT nOpenFlags,
   CFileException* pError = NULL
);
```

Parameters

lpszFileName

A string that is the path to the desired file. The path can be relative, absolute, or a network name (UNC).

nOpenFlags

A **UINT** that defines the file's sharing and access mode. It specifies the action to take when opening the file. You can combine options by using the bitwise-OR (**|**) operator. One access permission and one share option are required; the **modeCreate** and **modeNoInherit** modes are optional. See the CFile constructor for a list of mode options.

pError

A pointer to an existing file-exception object that will receive the status of a failed operation.

Return Value

Nonzero if the open was successful; otherwise 0. The *pError* parameter is meaningful only if 0 is returned.

Remarks

The two functions form a "safe" method for opening a file where a failure is a normal, expected condition.

While the **CFile** constructor will throw an exception in an error condition, **Open** will return **FALSE** for error conditions. **Open** can still initialize a CFileException object to describe the error, however. If you don't supply the *pError* parameter, or if you pass **NULL** for *pError*, **Open** will return **FALSE** and not throw a **CFileException**. If you pass a pointer to an existing **CFileException**, and **Open** encounters an error, the function will fill it with information describing that error. In neither case will **Open** throw an exception.

The following table describes the possible results of **Open**.

pError	**Error encountered?**	**Return value**	**CFileException content**
NULL	No	**TRUE**	n/a
ptr to **CFileException**	No	**TRUE**	unchanged
NULL	Yes	**FALSE**	n/a
ptr to **CFileException**	Yes	**FALSE**	initialized to describe error

Note

Under Windows NT, if the file *lpszFileName* exists and is hidden, and if *nOpenFlags* includes **CFile::modeCreate** but is not OR-ed with **CFile::modeNoTruncate**, **Open** fails. Under Windows 95/98 and later, the call succeeds and creates a new file that is not hidden.

Example

```
//example for CFile::Open
CFile f;
CFileException e;
char* pFileName = "test.dat";
if( !f.Open( pFileName, CFile::modeCreate | CFile::modeWrite, &e ) )
```

```
   {
#ifdef _DEBUG
   afxDump << "File could not be opened " << e.m_cause << "\n";
#endif
   }

//A second example for CFile::Open.
//This console program uses CFile to copy binary files.

#include <afx.h>
#include <afxwin.h>
#include <iostream>

using namespace std;

CWinApp theApp;

int main(int argc, char *argv[])
{
   if (!AfxWinInit(GetModuleHandle(NULL), NULL, GetCommandLine(), 0))
   {
      cout << "panic: MFC couldn't initialize!" << endl;
      return 1;
   }

   // constructing these file objects doesn't open them

   CFile sourceFile;
   CFile destFile;

   // see that we have a reasonable number of arguments

   if (argc != 3)
   {
      cout << "usage: " << argv[0];
      cout << " <source> <dest>" << endl;
      cout << endl;
      return 1;
   }

   // we'll use a CFileException object to get error information

   CFileException ex;

   // open the source file for reading

   if (!sourceFile.Open(argv[1],
      CFile::modeRead | CFile::shareDenyWrite, &ex))
   {
      // complain if an error happened
      // no need to delete the ex object

      TCHAR szError[1024];
      ex.GetErrorMessage(szError, 1024);
      cout << "Couldn't open source file: ";
      cout << szError;
      return 1;
   }
   else
   {
      if (!destFile.Open(argv[2], CFile::modeWrite |
            CFile::shareExclusive | CFile::modeCreate, &ex))
      {
         TCHAR szError[1024];
         ex.GetErrorMessage(szError, 1024);
         cout << "Couldn't open source file: ";
         cout << szError;
```

```
            sourceFile.Close();
            return 1;
         }

         BYTE buffer[4096];
         DWORD dwRead;

         // Read in 4096-byte blocks,
         // remember how many bytes were actually read,
         // and try to write that many out. This loop ends
         // when there are no more bytes to read.

         do
         {
            dwRead = sourceFile.Read(buffer, 4096);
            destFile.Write(buffer, dwRead);
         }
         while (dwRead > 0);

         // Close both files

         destFile.Close();
         sourceFile.Close();
      }

      return 0;
   }
```

See Also

Reference

CFile Class

Hierarchy Chart

CFile::CFile

CFile::Close

Other Resources

CFile Members

CFile::Read

Reads data into a buffer from the file associated with the **CFile** object.

```
virtual UINT Read(
   void* lpBuf,
   UINT nCount
);
```

Parameters

lpBuf

Pointer to the user-supplied buffer that is to receive the data read from the file.

nCount

The maximum number of bytes to be read from the file. For text-mode files, carriage return–linefeed pairs are counted as single characters.

Return Value

The number of bytes transferred to the buffer. Note that for all **CFile** classes, the return value may be less than *nCount* if the end of file was reached.

Example

```
//example for CFile::Read
extern CFile cfile;
char pbuf[100];
UINT nBytesRead = cfile.Read( pbuf, 100 );
```

For another example see CFile::Open.

See Also

Reference

CFile Class

Hierarchy Chart

Other Resources

CFile Members

CFile::Remove

This static function deletes the file specified by the path.

```
static void PASCAL Remove(
   LPCTSTR lpszFileName
);
```

Parameters

lpszFileName

A string that is the path to the desired file. The path can be relative or absolute, and can contain a network name.

Remarks

It will not remove a directory.

The **Remove** member function throws an exception if the connected file is open or if the file cannot be removed. This is equivalent to the DEL command.

Example

```
//example for CFile::Remove
TCHAR* pFileName = _T("test.dat");
try
{
    CFile::Remove(pFileName);
}
catch (CFileException* pEx)
{
#ifdef _DEBUG
    afxDump << "File " << pFileName << " cannot be removed\n";
#endif
   pEx->Delete();
}
```

See Also

Reference

CFile Class

Hierarchy Chart

Other Resources

CFile Members

CFile::Rename

This static function renames the specified file.

```
static void PASCAL Rename(
   LPCTSTR lpszOldName,
   LPCTSTR lpszNewName
);
```

Parameters

lpszOldName

The old path.

lpszNewName

The new path.

Remarks

Directories cannot be renamed. This is equivalent to the REN command.

Example

```
//example for CFile::Rename
extern TCHAR* pOldName;
extern TCHAR* pNewName;

try
{
    CFile::Rename( pOldName, pNewName );
}
catch(CFileException* pEx )
{
#ifdef _DEBUG
    afxDump << "File " << pOldName << " not found, cause = "
            << e->m_cause << "\n";
#endif
    pEx->Delete();
}
```

See Also

Reference

CFile Class

Hierarchy Chart

Other Resources

CFile Members

CFile::Seek

Repositions the pointer in a previously opened file.

```
virtual ULONGLONG Seek(
   LONGLONG lOff,
   UINT nFrom
);
```

Parameters

lOff

Number of bytes to move the pointer.

nFrom

Pointer movement mode. Must be one of the following values:

- **CFile::begin** Move the file pointer *lOff* bytes forward from the beginning of the file.
- **CFile::current** Move the file pointer *lOff* bytes from the current position in the file.
- **CFile::end** Move the file pointer *lOff* bytes from the end of the file. Note that *lOff* must be negative to seek into the existing file; positive values will seek past the end of the file.

Return Value

If the requested position is legal, **Seek** returns the new byte offset from the beginning of the file. Otherwise, the return value is undefined and a **CFileException** object is thrown.

Remarks

The **Seek** function permits random access to a file's contents by moving the pointer a specified amount, absolutely or relatively. No data is actually read during the seek. If the requested position is larger than the size of the file, the file length will be extended to that position, and no exception will be thrown.

When a file is opened, the file pointer is positioned at offset 0, the beginning of the file.

Example

```
//example for CFile::Seek
extern CFile cfile;
LONGLONG lOffset = 1000;
ULONGLONG lActual;
lActual = cfile.Seek( lOffset, CFile::begin );
```

See Also

Reference

CFile Class

Hierarchy Chart

Other Resources

CFile Members

CFile::SeekToBegin

Sets the value of the file pointer to the beginning of the file.

```
void SeekToBegin( );
```

Remarks

`SeekToBegin()` is equivalent to `Seek( 0L, CFile::begin )`.

Example

```
//example for CFile::SeekToBegin
extern CFile cfile;
cfile.SeekToBegin();
```

See Also

Reference

CFile Class

Hierarchy Chart

Other Resources

CFile Members

CFile::SeekToEnd

Sets the value of the file pointer to the logical end of the file.

```
ULONGLONG SeekToEnd( );
```

Return Value

The length of the file in bytes.

Remarks

`SeekToEnd()` is equivalent to `CFile::Seek( 0L, CFile::end )`.

Example

```
//example for CFile::SeekToEnd
extern CFile cfile;
ULONGLONG dwActual = cfile.SeekToEnd();
```

See Also

Reference

CFile Class
Hierarchy Chart
CFile::GetLength
CFile::Seek
CFile::SeekToBegin

Other Resources

CFile Members

CFile::SetFilePath

Call this function to specify the path of the file; for example, if the path of a file is not available when a CFile object is constructed, call **SetFilePath** to provide it.

```
virtual void SetFilePath(
   LPCTSTR lpszNewName
);
```

Parameters

lpszNewName

Pointer to a string specifying the new path.

Remarks

Note

SetFilePath does not open the file or create the file; it simply associates the **CFile** object with a path name, which can then be used.

Example

```
TCHAR* pstrName = _T("C:\\MyFile.DAT");

// open a file
HANDLE hFile = ::CreateFile(pstrName, GENERIC_WRITE, FILE_SHARE_READ,
   NULL, CREATE_ALWAYS, 0, NULL);

if (hFile != INVALID_HANDLE_VALUE)
{
   // attach a CFile object to it
   CFile myFile((int) hFile);

   // At this point, myFile doesn't know the path name for the file
   // it owns because Windows doesn't associate that information
   // with the handle. Any CFileExceptions thrown by this object
   // won't have complete information.

   // Calling SetFilePath() remedies that problem by letting CFile
   // know the name of the file that's associated with the object.

   myFile.SetFilePath(pstrName);

   // write something to the file and flush it immediately
   DWORD dwValue = 1234;
   myFile.Write(&dwValue, sizeof(dwValue));
   myFile.Flush();

   // destroying the CObject here will call ::CloseHandle() on the file
}
```

See Also

Reference

CFile Class

Hierarchy Chart

CFile::GetFilePath

CFile::CFile

Other Resources

CFile Members

CFile::SetLength

Call this function to change the length of the file.

```
virtual void SetLength(
   ULONGLONG dwNewLen
);
```

Parameters

dwNewLen

Desired length of the file in bytes. This value can be larger or smaller than the current length of the file. The file will be extended or truncated as appropriate.

Remarks

Note

With **CMemFile**, this function could throw a **CMemoryException** object.

Example

```
//example for CFile::SetLength
extern CFile cfile;
ULONGLONG dwNewLength = 10000;
cfile.SetLength( dwNewLength );
```

See Also

Reference

CFile Class

Hierarchy Chart

Other Resources

CFile Members

CFile::SetStatus

Sets the status of the file associated with this file location.

```
static void PASCAL SetStatus(
   LPCTSTR lpszFileName,
   const CFileStatus& status
);
```

Parameters

lpszFileName

A string that is the path to the desired file. The path can be relative or absolute, and can contain a network name.

status

The buffer containing the new status information. Call the **GetStatus** member function to prefill the **CFileStatus** structure with current values, then make changes as required. If a value is 0, then the corresponding status item is not updated. See the GetStatus member function for a description of the **CFileStatus** structure.

Remarks

To set the time, modify the **m_mtime** field of *status*.

Please note that when you make a call to **SetStatus** in an attempt to change only the attributes of the file, and the **m_mtime** member of the file status structure is nonzero, the attributes may also be affected (changing the time stamp may have side effects on the attributes). If you want to only change the attributes of the file, first set the **m_mtime** member of the file status structure to zero and then make a call to **SetStatus**.

Example

```
//example for CFile::SetStatus
char* pFileName = "test.dat";
extern BYTE newAttribute;
CFileStatus status;
CFile::GetStatus( pFileName, status );
status.m_attribute = newAttribute;
CFile::SetStatus( pFileName, status );
```

See Also

Reference

CFile Class
Hierarchy Chart
CFile::GetStatus

Other Resources

CFile Members

CFile::UnlockRange

Unlocks a range of bytes in an open file.

```
virtual void UnlockRange(
   ULONGLONG dwPos,
   ULONGLONG dwCount
);
```

Parameters

dwPos

The byte offset of the start of the byte range to unlock.

dwCount

The number of bytes in the range to unlock.

Remarks

See the description of the LockRange member function for details.

Note

This function is not available for the **CMemFile**-derived class.

Example

```
//example for CFile::UnlockRange
extern ULONGLONG dwPos;
extern ULONGLONG dwCount;
extern CFile cfile;
cfile.UnlockRange( dwPos, dwCount );
```

See Also

Reference

CFile Class

Hierarchy Chart

CFile::LockRange

Other Resources

CFile Members

CFile::Write

Writes data from a buffer to the file associated with the **CFile** object.

```
virtual void Write(
   const void* lpBuf,
   UINT nCount
);
```

Parameters

lpBuf

A pointer to the user-supplied buffer that contains the data to be written to the file.

nCount

The number of bytes to be transferred from the buffer. For text-mode files, carriage return–linefeed pairs are counted as single characters.

Remarks

Write throws an exception in response to several conditions, including the disk-full condition.

Example

```
//example for CFile::Write
extern CFile cfile;
char pbuf[100];
cfile.Write( pbuf, 100 );
```

In addition, see the examples for CFile::CFile and CFile::Open.

See Also

Reference

CFile Class
Hierarchy Chart
CFile::Read
CStdioFile::WriteString

Other Resources

CFile Members

CFile Data Members

For information about the data members in **CFile**, see CFile Members.

CFile::hFileNull

Determines the presence of a valid file handle for the **CFile** object.

```
static AFX_DATA const HANDLE hFileNull;
```

Remarks

This constant is used to determine if the **CFile** object has a valid file handle.

The following example demonstrates this operation:

```
if (myFile.m_hFile != CFile::hFileNull)
   //perform operations on the file
else
   //indicate the presence of an invalid handle
```

See Also

Reference

CFile Class

Hierarchy Chart

Other Resources

CFile Members

CFile::m_hFile

Contains the operating-system file handle for an open file.

```
HANDLE m_hFile;
```

Remarks

m_hFile is a public variable of type **UINT**. It contains **CFile::hFileNull** (an operating-system-independent empty file indicator) if the handle has not been assigned.

Use of **m_hFile** is not recommended because the member's meaning depends on the derived class. **m_hFile** is made a public member for convenience in supporting nonpolymorphic use of the class.

See Also

Reference

CFile Class

Hierarchy Chart

Other Resources

CFile Members

CFile Operators

For information about the operators in **CFile**, see CFile Members.

CFile::operator HANDLE

Use this operator to pass a handle to a **CFile** object to functions such as ReadFileEx and GetFileTime that expect a **HANDLE**.

```
operator HANDLE( ) const;
```

See Also

Reference

CFile Class

Hierarchy Chart

CFile::m_hFile

Other Resources

CFile Members

CFileDialog Class

Encapsulates the Windows common file dialog box.

```
class CFileDialog : public CCommonDialog
```

Remarks

Common file dialog boxes provide an easy way to implement File Open and File Save As dialog boxes (as well as other file-selection dialog boxes) in a manner consistent with Windows standards.

You can use **CFileDialog** "as is" with the constructor provided, or you can derive your own dialog class from **CFileDialog** and write a constructor to suit your needs. In either case, these dialog boxes will behave like standard Microsoft Foundation class dialog boxes because they are derived from the **CCommonDialog** class.

To use a **CFileDialog** object, first create the object using the **CFileDialog** constructor. After the dialog box has been constructed, you can set or modify any values in the **m_ofn** structure to initialize the values or states of the dialog box's controls. The **m_ofn** structure is of type **OPENFILENAME**. For more information, see the OPENFILENAME structure in the *Platform SDK*.

After initializing the dialog box's controls, call the **DoModal** member function to display the dialog box and allow the user to enter the path and file. **DoModal** returns whether the user selected the OK (**IDOK**) or the Cancel (**IDCANCEL**) button.

If **DoModal** returns **IDOK**, you can use one of **CFileDialog**'s public member functions to retrieve the information input by the user.

CFileDialog includes several protected members that enable you to do custom handling of share violations, filename validation, and list-box change notification. These protected members are callback functions that most applications do not need to use, since default handling is done automatically. Message-map entries for these functions are not necessary because they are standard virtual functions.

You can use the Windows **CommDlgExtendedError** function to determine whether an error occurred during initialization of the dialog box and to learn more about the error.

The destruction of **CFileDialog** objects is handled automatically. It is not necessary to call **CDialog::EndDialog**.

To allow the user to select multiple files, set the **OFN_ALLOWMULTISELECT** flag before calling **DoModal**. You need to supply your own filename buffer to accommodate the returned list of multiple filenames. Do this by replacing **m_ofn.lpstrFile** with a pointer to a buffer you have allocated, after constructing the **CFileDialog**, but before calling **DoModal**.

When the user allocates their own buffer to accommodate **OFN_ALLOWMULTISELECT**, the buffer can't be larger than 2048 or else everything gets corrupted (2048 is the maximum size).

Additionally, you must set **m_ofn.nMaxFile** with the number of characters in the buffer pointed to by **m_ofn.lpstrFile**. If you set the maximum number of files to be selected to n, the necessary buffer size is n*(_MAX_PATH + 1) + 1. For example:

```
CFileDialog dlgFile(...);
...
CString fileName;
dlgFile.GetOFN().lpstrFile = fileName.GetBuffer(<very large number>);
dlgFile.GetOFN().nMaxFile = <very large number>;

INT_PTR nResult = dlgFile.DoModal();
fileName.ReleaseBuffer();
```

CFileDialog relies on the COMMDLG.DLL file that ships with Windows versions 3.1 and later.

If you derive a new class from **CFileDialog**, you can use a message map to handle any messages. To extend the default message handling, derive a class from **CWnd**, add a message map to the new class, and provide member functions for the new messages. You do not need to provide a hook function to customize the dialog box.

To customize the dialog box, derive a class from **CFileDialog**, provide a custom dialog template, and add a message map to process the notification messages from the extended controls. Any unprocessed messages should be passed to the base class.

Customizing the hook function is not required.

For more information on using **CFileDialog**, see Common Dialog Classes.

Requirements

Header: afxdlgs.h

See Also

Reference

CCommonDialog Class

Hierarchy Chart

Other Resources

CFileDialog Members

CFileDialog Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CDialog Members

CCommonDialog Members

Data Members

m_ofn	The Windows **OPENFILENAME** structure. Provides access to basic file dialog box parameters.

Construction

CFileDialog	Constructs a **CFileDialog** object.

Operations

DoModal	Displays the dialog box and allows the user to make a selection.
GetFileExt	Returns the file extension of the selected file.
GetFileName	Returns the filename of the selected file.
GetFileTitle	Returns the title of the selected file.
GetFolderPath	Retrieves the path of the currently open folder or directory for an Explorer-style Open or Save As common dialog box.
GetNextPathName	Returns the full path of the next selected file.
GetOFN	Retrieves the **OPENFILENAME** structure of the **CFileDialog** object.
GetPathName	Returns the full path of the selected file.
GetReadOnlyPref	Returns the read-only status of the selected file.
GetStartPosition	Returns the position of the first element of the filename list.
HideControl	Hides the specified control in an Explorer-style Open or Save As common dialog box.
SetControlText	Sets the text for the specified control in an Explorer-style Open or Save As common dialog box.
SetDefExt	Sets the default file name extension for an Explorer-style Open or Save As common dialog box.
SetTemplate	Sets the dialog box template for the **CFileDialog** object.

Overridables

OnFileNameChange	Called to handle the **WM_NOTIFY CDN_SELCHANGE** message.
OnFileNameOK	Called to validate the filename entered in the dialog box.
OnFolderChange	Called to handle the **WM_NOTIFY CDN_FOLDERCHANGE** message.

OnInitDone	Called to handle the **WM_NOTIFY CDN_INITDONE** message.
OnLBSelChangedNotify	Called when the list box selection changes.
OnShareViolation	Called when a share violation occurs.
OnTypeChange	Called to handle the **WM_NOTIFY CDN_TYPECHANGE** message.

See Also

Reference

CFileDialog Class
Hierarchy Chart

CFileDialog Member Functions

For information about the member functions in **CFileDialog**, see CFileDialog Members.

CFileDialog::CFileDialog

Call this function to construct a standard Windows file dialog box-object.

```
explicit CFileDialog(
   BOOL bOpenFileDialog,
   LPCTSTR lpszDefExt = NULL,
   LPCTSTR lpszFileName = NULL,
   DWORD dwFlags = OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
   LPCTSTR lpszFilter = NULL,
   CWnd* pParentWnd = NULL,
   DWORD dwSize = 0
);
```

Parameters

bOpenFileDialog

Set to **TRUE** to construct a File Open dialog box or **FALSE** to construct a File Save As dialog box.

lpszDefExt

The default filename extension. If the user does not include an extension in the Filename edit box, the extension specified by *lpszDefExt* is automatically appended to the filename. If this parameter is **NULL**, no file extension is appended.

lpszFileName

The initial filename that appears in the filename edit box. If **NULL**, no filename initially appears.

dwFlags

A combination of one or more flags that allow you to customize the dialog box. For a description of these flags, see the OPENFILENAME structure in the *Platform SDK*. If you modify the **m_ofn.Flags** structure member, use a bitwise-OR operator in your changes to keep the default behavior intact.

lpszFilter

A series of string pairs that specify filters you can apply to the file. If you specify file filters, only selected files will appear in the Files list box. See the Remarks section for more information on how to work with file filters.

pParentWnd

A pointer to the file dialog-box object's parent or owner window.

dwSize

The size of the **OPENFILENAME** structure. This value is dependent on the operating system version, so MFC can determine the appropriate kind of dialog box to create (for example, new Windows 2000 dialogs as opposed to NT4 dialogs). The default size of 0 means that the MFC code will determine the proper dialog box size to use based on the operating system version on which the program is run.

Remarks

Either a File Open or File Save As dialog box is constructed, depending on the value of *bOpenFileDialog*.

To allow the user to select multiple files, set the **OFN_ALLOWMULTISELECT** flag before calling **DoModal**. You need to supply your own filename buffer to accommodate the returned list of multiple filenames. Do this by replacing **m_ofn.lpstrFile** with a pointer to a buffer you have allocated, after constructing the **CFileDialog**, but before calling **DoModal**. Additionally, you must set **m_ofn.nMaxFile** with the number of characters in the buffer pointed to by **m_ofn.lpstrFile**. If you set the maximum number of files to be selected to n, the necessary buffer size is n*(_MAX_PATH + 1) + 1. For example:

```
CFileDialog dlgFile(...);
...
CString fileName;
dlgFile.GetOFN().lpstrFile = fileName.GetBuffer(<very large number>);
dlgFile.GetOFN().nMaxFile = <very large number>;
```

```
INT_PTR nResult = dlgFile.DoModal();
fileName.ReleaseBuffer();
```

To allow the user to resize an Explorer-style dialog box using either the mouse or keyboard, set the **OFN_ENABLESIZING** flag. Setting this flag is necessary only if you provide a hook procedure or custom template. The flag works only with an Explorer-style dialog box; old-style dialog boxes do not permit resizing.

The *lpszFilter* parameter is used to determine the type of filename a file must have to be displayed in the file list box. The first string in the string pair describes the filter; the second string indicates the file extension to use. Multiple extensions may be specified using ';' as the delimiter. The string ends with two '|' characters, followed by a **NULL** character. You can also use a **CString** object for this parameter.

For example, Microsoft Excel permits users to open files with extensions .XLC (chart) or .XLS (worksheet), among others. The filter for Excel could be written as:

```
static char BASED_CODE szFilter[] = "Chart Files (*.xlc)|*.xlc|Worksheet Files (*.xls)|*.xl
s|Data Files (*.xlc;*.xls)|*.xlc; *.xls|All Files (*.*)|*.*||";
```

Note, however, that if you plan to use this string to directly update the OPENFILENAME structure, that you should delimit your strings with the null character, '\0', instead of the vertical bars (|).

Example

See the example for CFileDialog::DoModal.

See Also

Reference

CFileDialog Class
Hierarchy Chart
CFileDialog::DoModal
GetOpenFileName
GetSaveFileName
OPENFILENAME

Other Resources

CFileDialog Members

CFileDialog::DoModal

Call this function to display the Windows common file dialog box and allow the user to browse files and directories and enter a filename.

```
virtual INT_PTR DoModal( );
```

Return Value

IDOK or **IDCANCEL**. If **IDCANCEL** is returned, call the Windows CommDlgExtendedError function to determine whether an error occurred.

IDOK and **IDCANCEL** are constants that indicate whether the user selected the OK or Cancel button.

Remarks

If you want to initialize the various file dialog-box options by setting members of the **m_ofn** structure, you should do this before calling **DoModal**, but after the dialog object is constructed.

For example, if you want to allow the user to select multiple files, set the **OFN_ALLOWMULTISELECT** flag before calling **DoModal**, as shown in the code example in CFileDialog Class.

When the user clicks the dialog box's OK or Cancel buttons, or selects the Close option from the dialog box's control menu, control is returned to your application. You can then call other member functions to retrieve the settings or information the user inputs into the dialog box.

DoModal is a virtual function overridden from class **CDialog**.

Example

```
void CChildFrame::OnFileOpen()
{
   // szFilters is a text string that includes two file name filters:
   // "*.my" for "MyType Files" and "*.*' for "All Files."
   char CChildFrame::szFilters[]=
      "MyType Files (*.my)|*.my|All Files (*.*)|*.*||";

   // Create an Open dialog; the default file name extension is ".my".
   CFileDialog fileDlg (TRUE, "my", "*.my",
      OFN_FILEMUSTEXIST| OFN_HIDEREADONLY, szFilters, this);

   // Display the file dialog. When user clicks OK, fileDlg.DoModal()
   // returns IDOK.
   if( fileDlg.DoModal ()==IDOK )
   {
      CString pathName = fileDlg.GetPathName();

      // Implement opening and reading file in here.
      ...
      //Change the window's title to the opened file's title.
      CString fileName = fileDlg.GetFileTitle ();

      SetWindowText(fileName);
   }

}
```

See Also

Reference

CFileDialog Class
Hierarchy Chart
CDialog::DoModal
CFileDialog::CFileDialog

Other Resources

CFileDialog::GetFileExt

Call this function to retrieve the extension of the filename entered into the dialog box.

```
CString GetFileExt( ) const;
```

Return Value

The extension of the filename.

Remarks

For example, if the name of the file entered is DATA.TXT, **GetFileExt** returns "TXT".

If **m_ofn.Flags** has the **OFN_ALLOWMULTISELECT** flag set, this string contains a sequence of null-terminated strings, with the first string being the directory path of the file group selected, followed by the names of all files selected by the user. To retrieve file pathnames, use the GetStartPosition and GetNextPathName member functions.

See Also

Reference

CFileDialog Class
Hierarchy Chart
CFileDialog::GetPathName
CFileDialog::GetFileName
CFileDialog::GetFileTitle

Other Resources

CFileDialog Members

CFileDialog::GetFileName

Call this function to retrieve the name of the filename entered in the dialog box.

```
CString GetFileName( ) const;
```

Return Value

The name of the file.

Remarks

The name of the file includes both the prefix and the extension. For example, **GetFileName** will return "TEXT.DAT" for the file C:\FILES\TEXT.DAT.

If **m_ofn.Flags** has the **OFN_ALLOWMULTISELECT** flag set, you should call GetStartPosition and GetNextPathName to retrieve a file pathname.

See Also

Reference

CFileDialog Class
Hierarchy Chart
CFileDialog::GetPathName
CFileDialog::GetStartPosition
CFileDialog::GetFileTitle

Other Resources

CFileDialog Members

CFileDialog::GetFileTitle

Call this function to retrieve the title of the file entered in the dialog box.

```
CString GetFileTitle( ) const;
```

Return Value

The title of the file.

Remarks

The title of the file includes only its prefix, without the path or the extension. For example, **GetFileTitle** will return "TEXT" for the file C:\FILES\TEXT.DAT.

If **m_ofn.Flags** has the **OFN_ALLOWMULTISELECT** flag set, this string contains a sequence of null-terminated strings, with the first string being the directory path of the file group selected, followed by the names of all files selected by the user. For this reason, use the GetStartPosition and GetNextPathName member functions to retrieve the next file name in the list.

Example

See the example for CFileDialog::DoModal.

See Also

Reference

CFileDialog Class
Hierarchy Chart
CFileDialog::GetPathName
CFileDialog::GetFileName
CFileDialog::GetFileExt
GetFileTitle

Other Resources

CFileDialog Members

CFileDialog::GetFolderPath

Call this member function to retrieve the path of the currently open folder or directory for an Explorer-style Open or Save As common dialog box.

```
CString GetFolderPath( ) const;
```

Return Value

A CString object containing the currently open folder or directory.

Remarks

The dialog box must have been created with the **OFN_EXPLORER** style; otherwise, the method will fail with an assertion.

You can call this method only while the dialog box is being displayed. After the dialog box has been closed, this function will no longer work, and the method will fail with an assertion.

See Also

Reference

CFileDialog Class

Hierarchy Chart

CDM_GETFOLDERPATH

Other Resources

CFileDialog Members

CFileDialog::GetNextPathName

Call this function to retrieve the next filename from the group selected in the dialog box.

```
CString GetNextPathName(
   POSITION& pos
) const;
```

Parameters

pos

A reference to a **POSITION** value returned by a previous **GetNextPathName** or **GetStartPosition** function call. **NULL** if the end of the list has been reached.

Return Value

The full path of the file.

Remarks

The path of the filename includes the file's title plus the entire directory path. For example, **GetNextPathName** will return "C:\FILES\TEXT.DAT" for the file C:\FILES\TEXT.DAT. You can use **GetNextPathName** in a forward iteration loop if you establish the initial position with a call to **GetStartPosition**.

If the selection consists of only one file, that file name will be returned.

See Also

Reference

CFileDialog Class
Hierarchy Chart
CFileDialog::GetFileName
CFileDialog::GetStartPosition

Other Resources

CFileDialog Members

CFileDialog::GetOFN

Retrieves the associated **OPENFILENAME** structure.

```
const OPENFILENAME& GetOFN( ) const;
OPENFILENAME& GetOFN( );
```

Return Value

An OPENFILENAME structure.

Remarks

Use the second version of this function to initialize the appearance of a **File Open** or **File Save As** dialog box after it is constructed but before it is displayed with the **DoModal** member function. For example, you can set the **lpstrTitle** member of **m_ofn** to the caption you want the dialog box to have.

See Also

Reference

CFileDialog Class

Hierarchy Chart

CFileDialog::m_ofn

Other Resources

CFileDialog Members

CFileDialog::GetPathName

Call this function to retrieve the full path of the file entered in the dialog box.

```
CString GetPathName( ) const;
```

Return Value

The full path of the file.

Remarks

The path of the filename includes the file's title plus the entire directory path. For example, **GetPathName** will return "C:\FILES\TEXT.DAT" for the file C:\FILES\TEXT.DAT.

If **m_ofn.Flags** has the **OFN_ALLOWMULTISELECT** flag set, this string contains a sequence of null-teminated strings, with the first string being the directory path of the file group selected, followed by the names of all files selected by the user. For this reason, use the GetStartPosition and GetNextPathName member functions to retrieve the next file name in the list.

Example

See the example for CFileDialog::DoModal.

See Also

Reference

CFileDialog Class
Hierarchy Chart
CFileDialog::GetFileName
CFileDialog::GetFileExt
CFileDialog::GetFileTitle

Other Resources

CFileDialog Members

CFileDialog::GetReadOnlyPref

Call this function to determine whether the Read Only check box has been selected in the Windows standard File Open and File Save As dialog boxes.

```
BOOL GetReadOnlyPref( ) const;
```

Return Value

Non-zero if the Read Only check box in the dialog box is selected; otherwise 0.

Remarks

The Read Only check box can be hidden by setting the **OFN_HIDEREADONLY** style in the **CFileDialog** constructor.

See Also

Reference

CFileDialog Class
Hierarchy Chart
CFileDialog::CFileDialog
CFileDialog::GetPathName
CFileDialog::GetFileExt

Other Resources

CFileDialog Members

CFileDialog::GetStartPosition

Call this member function to retrieve the position of the first file pathname in the list, if **m_ofn.Flags** has the **OFN_ALLOWMULTISELECT** flag set.

```
POSITION GetStartPosition( ) const;
```

Return Value

A **POSITION** value that can be used for iteration; **NULL** if the list is empty.

See Also

Reference

CFileDialog Class
Hierarchy Chart
CFileDialog::GetFileName
CFileDialog::GetNextPathName

Other Resources

CFileDialog Members

CFileDialog::HideControl

Call this member function to hide the specified control in an Explorer-style Open or Save As common dialog box.

```
void HideControl(
   int nID
);
```

Parameters

nID

The ID of the control to hide.

Remarks

The dialog box must have been created with the **OFN_EXPLORER** style; otherwise, the function will fail with an assertion.

See Also

Reference

CFileDialog Class
Hierarchy Chart
CDM_HIDECONTROL
CFileDialog::SetControlText

Other Resources

CFileDialog Members

CFileDialog::OnFileNameChange

Override this function to handle the **WM_NOTIFY CDN_SELCHANGE** message.

```
virtual void OnFileNameChange( );
```

Remarks

The notification message is sent when the user selects a new file or folder in the file list of the Open or Save As dialog box.

Notification is sent only if the dialog box was created with the OFN_EXPLORER style. For more information about the notification, see CDN_SELCHANGE. For information about the OFN_EXPLORER style, see the OPENFILENAME structure and Open and Save As Dialog Boxes.

See Also

Reference

CFileDialog Class
Hierarchy Chart
CFileDialog::OnFolderChange

Other Resources

CFileDialog Members

CFileDialog::OnFileNameOK

Override this function only if you want to provide custom validation of filenames that are entered into a common file dialog box.

```
virtual BOOL OnFileNameOK( );
```

Return Value

1 if the filename is not a valid filename; otherwise 0.

Remarks

This function allows you to reject a filename for any application-specific reason. Normally, you do not need to use this function because the framework provides default validation of filenames and displays a message box if an invalid filename is entered.

If 1 is returned, the dialog box will remain displayed for the user to enter another filename. The dialog procedure dismisses the dialog if the return is 0. Other nonzero return values are currently reserved and should not be used.

See Also

Reference

CFileDialog Class
Hierarchy Chart
OPENFILENAME

Other Resources

CFileDialog Members

CFileDialog::OnFolderChange

Override this function to handle the **WM_NOTIFY CDN_FOLDERCHANGE** message.

```
virtual void OnFolderChange( );
```

Remarks

The notification message is sent when a new folder is opened in the Open or Save As dialog box.

Notification is sent only if the dialog box was created with the OFN_EXPLORER style. For more information about the notification, see CDN_FOLDERCHANGE. For information about the OFN_EXPLORER style, see the OPENFILENAME structure and Open and Save As Dialog Boxes.

See Also

Reference

CFileDialog Class
Hierarchy Chart
CFileDialog::OnFileChange

Other Resources

CFileDialog Members

CFileDialog::OnInitDone

Override this function to handle the **WM_NOTIFY CDN_INITDONE** message.

```
virtual void OnInitDone( );
```

Remarks

The notification message is sent when the system has finished arranging controls in the Open or Save As dialog box to make room for the controls of the child dialog box.

Notification is sent only if the dialog box was created with the OFN_EXPLORER style. For more information about the notification, see CDN_INITDONE. For information about the OFN_EXPLORER style, see the OPENFILENAME structure and Open and Save As Dialog Boxes.

See Also

Reference

CFileDialog Class

Hierarchy Chart

Other Resources

CFileDialog Members

CFileDialog::OnLBSelChangedNotify

This function is called whenever the current selection in a list box is about to change.

```
virtual void OnLBSelChangedNotify(
   UINT nIDBox,
   UINT iCurSel,
   UINT nCode
);
```

Parameters

nIDBox

The ID of the list box or combo box in which the selection occurred.

iCurSel

The index of the current selection.

nCode

The control notification code. This parameter must have one of the following values:

- **CD_LBSELCHANGE** Specifies *iCurSel* is the selected item in a single-selection list box.
- **CD_LBSELSUB** Specifies that *iCurSel* is no longer selected in a multiselection list box.
- **CD_LBSELADD** Specifies that *iCurSel* is selected in a multiselection list box.
- **CD_LBSELNOITEMS** Specifies that no selection exists in a multiselection list box.

Remarks

Override this function to provide custom handling of selection changes in the list box. For example, you can use this function to display the access rights or date-last-modified of each file the user selects.

See Also

Reference

CFileDialog Class

Hierarchy Chart

Other Resources

CFileDialog Members

CFileDialog::OnShareViolation

Override this function to provide custom handling of share violations.

```
virtual UINT OnShareViolation(
   LPCTSTR lpszPathName
);
```

Parameters

lpszPathName

The path of the file on which the share violation occurred.

Return Value

One of the following values:

- **OFN_SHAREFALLTHROUGH** The filename is returned from the dialog box.
- **OFN_SHARENOWARN** No further action needs to be taken.
- **OFN_SHAREWARN** The user receives the standard warning message for this error.

Remarks

Normally, you do not need to use this function because the framework provides default checking of share violations and displays a message box if a share violation occurs.

If you want to disable share violation checking, use the bitwise OR operator to combine the flag **OFN_SHAREAWARE** with **m_ofn.Flags**.

See Also

Reference

CFileDialog Class
Hierarchy Chart
CFileDialog::OnFileNameOK

Other Resources

CFileDialog Members

CFileDialog::OnTypeChange

Override this function to handle the **WM_NOTIFY CDN_TYPECHANGE** message.

```
virtual void OnTypeChange( );
```

Remarks

The notification message is sent when the user selects a new file type from the list of file types in the Open or Save As dialog box.

Notification is sent only if the dialog box was created with the OFN_EXPLORER style. For more information about the notification, see CDN_TYPECHANGE. For information about the OFN_EXPLORER style, see the OPENFILENAME structure and Open and Save As Dialog Boxes.

See Also

Reference

CFileDialog Class
Hierarchy Chart
CFileDialog::OnFolderChange

Other Resources

CFileDialog Members

CFileDialog::SetControlText

Call this member function to set the text for the specified control in an Explorer-style Open or Save As common dialog box.

```
void SetControlText(
   int nID,
   LPCSTR lpsz
);
```

Parameters

nID

The ID of the control for which to set the text.

lpsz

A pointer to the string containing the text to set for the control.

Remarks

The dialog box must have been created with the **OFN_EXPLORER** style; otherwise, the function will fail with an assertion.

See Also

Reference

CFileDialog Class
Hierarchy Chart
CDM_SETCONTROLTEXT
CFileDialog::HideControl

Other Resources

CFileDialog Members

CFileDialog::SetDefExt

Call this function to set the default file name extension for an Explorer-style Open or Save As common dialog box.

```
void SetDefExt(
   LPCSTR lpsz
);
```

Parameters

lpsz

A pointer to a string containing the default extension to use for the dialog box object. This string must not contain a period (.).

Remarks

The dialog box must have been created with the **OFN_EXPLORER** style; otherwise, the function will fail with an assertion.

See Also

Reference

CFileDialog Class

Hierarchy Chart

CDM_SETDEFEXT

Other Resources

CFileDialog Members

CFileDialog::SetTemplate

Call this function to set the dialog box template for the **CFileDialog** object.

```
void SetTemplate(
   UINT nWin3ID,
   UINT nWin4ID
);
void SetTemplate(
   LPCTSTR lpWin3ID,
   LPCTSTR lpWin4ID
);
```

Parameters

nWin3ID

Contains the ID number of the non-Explorer common file dialog template resource. This template is only used on Windows NT 3.51 or when the **OFN_EXPLORER** style is not present.

nWin4ID

Contains the ID number of the Explorer common file dialog template resource. This template is only used on Windows NT 4.0 or greater, Windows 95 or greater, or when the **OFN_EXPLORER** style is present.

lpWin3ID

Contains the name of the non-Explorer common file dialog template resource. This template is only used on Windows NT 3.51 or when the **OFN_EXPLORER** style is not present.

lpWin4ID

Contains the name of the Explorer common file dialog template resource. This template is only used on Windows NT 4.0 or greater, Windows 95 or greater, or when the **OFN_EXPLORER** style is present.

Remarks

Only one of the specified templates is used, depending on the presence of the **OFN_EXPLORER** style and depending on the platform that the application is running on. By specifying both a non-Explorer and Explorer-style template, it is easy to support Windows NT 3.51, Windows NT 4.0 and greater, and Windows 95 and greater.

See Also

Reference

CFileDialog Class

Hierarchy Chart

Other Resources

CFileDialog Members

CFileDialog Data Members

For information about the data members in **CFileDialog**, see CFileDialog Members.

CFileDialog::m_ofn

m_ofn is a structure of type **OPENFILENAME**.

Remarks

Use this structure to initialize the appearance of a File Open or File Save As dialog box after it is constructed but before it is displayed with the **DoModal** member function. For example, you can set the **lpstrTitle** member of **m_ofn** to the caption you want the dialog box to have.

For more information, see the OPENFILENAME structure in the *Platform SDK*.

See Also

Reference

CFileDialog Class

Hierarchy Chart

Other Resources

CFileDialog Members

CFileException Class

Represents a file-related exception condition.

```
class CFileException : public CException
```

Remarks

The **CFileException** class includes public data members that hold the portable cause code and the operating-system-specific error number. The class also provides static member functions for throwing file exceptions and for returning cause codes for both operating-system errors and C run-time errors.

CFileException objects are constructed and thrown in **CFile** member functions and in member functions of derived classes. You can access these objects within the scope of a **CATCH** expression. For portability, use only the cause code to get the reason for an exception. For more information about exceptions, see the article Exception Handling (MFC).

Requirements

Header: afx.h

See Also

Reference

CException Class

Hierarchy Chart

Other Resources

CFileException Members

Exception Processing

CFileException Members

Base Class Members

CObject Members

CException Members

Data Members

m_cause	Contains portable code corresponding to the exception cause.
m_lOsError	Contains the related operating-system error number.
m_strFileName	Contains the name of the file for this exception.

Construction

CFileException	Constructs a **CFileException** object.

Code Conversion

ErrnoToException	Returns cause code corresponding to a run-time error number.
OsErrorToException	Returns a cause code corresponding to an operating system error code.

Helper Functions

ThrowErrno	Throws a file exception based on a run-time error number.
ThrowOsError	Throws a file exception based on an operating-system error number.

See Also

Reference

CFileException Class

Hierarchy Chart

CFileException Member Functions

For information about the member functions in **CFileException**, see CFileException Members.

CFileException::CFileException

Constructs a **CFileException** object that stores the cause code and the operating-system code in the object.

```
CFileException(
   int cause = CFileException::none,
   LONG lOsError = -1,
   LPCTSTR lpszArchiveName = NULL
);
```

Parameters

cause

An enumerated type variable that indicates the reason for the exception. See CFileException::m_cause for a list of the possible values.

lOsError

An operating-system-specific reason for the exception, if available. The *lOsError* parameter provides more information than *cause* does.

lpszArchiveName

Points to a string containing the name of the **CFile** object causing the exception.

Remarks

Do not use this constructor directly, but rather call the global function AfxThrowFileException.

Note
The variable *lOsError* applies only to **CFile** and **CStdioFile** objects. The **CMemFile** class does not handle this error code.

See Also

Reference

CFileException Class

Hierarchy Chart

AfxThrowFileException

Other Resources

CFileException Members

CFileException::ErrnoToException

Converts a given run-time library error value to a **CFileException** enumerated error value.

```
static int PASCAL ErrnoToException(
   int nErrno
);
```

Parameters

nErrno

An integer error code as defined in the run-time include file ERRNO.H.

Return Value

Enumerated value that corresponds to a given run-time library error value.

Remarks

See CFileException::m_cause for a list of the possible enumerated values.

Example

```
//example for CFileException::ErrnoToException
#include <errno.h>
ASSERT( CFileException::ErrnoToException( EACCES ) ==
                  CFileException::accessDenied );
```

See Also

Reference

CFileException Class

Hierarchy Chart

CFileException::OsErrorToException

Other Resources

CFileException Members

CFileException::OsErrorToException

Returns an enumerator that corresponds to a given *lOsError* value. If the error code is unknown, then the function returns **CFileException::generic**.

```
static int PASCAL OsErrorToException(
   LONG lOsError
);
```

Parameters

lOsError

An operating-system-specific error code.

Return Value

Enumerated value that corresponds to a given operating-system error value.

Example

```
//example for CFileException::OsErrorToException
ASSERT( CFileException::OsErrorToException( 5 ) ==
                  CFileException::accessDenied );
```

See Also

Reference

CFileException Class

Hierarchy Chart

CFileException::ErrnoToException

Other Resources

CFileException Members

CFileException::ThrowErrno

Constructs a **CFileException** object corresponding to a given *nErrno* value, then throws the exception.

```
static void PASCAL ThrowErrno(
   int nErrno,
      LPCTSTR lpszFileName = NULL
);
```

Parameters

nErrno

An integer error code as defined in the run-time include file ERRNO.H.

lpszFileName

A pointer to the string containing the name of the file that caused the exception, if available.

Example

```
//example for CFileException::ThrowErrno
#include <errno.h>
CFileException::ThrowErrno( EACCES );  // "access denied"
```

See Also

Reference

CFileException Class

Hierarchy Chart

CFileException::ThrowOsError

Other Resources

CFileException Members

CFileException::ThrowOsError

Throws a **CFileException** corresponding to a given *lOsError* value. If the error code is unknown, then the function throws an exception coded as **CFileException::generic**.

```
static void PASCAL ThrowOsError(
   LONG lOsError,
      LPCTSTR lpszFileName = NULL
);
```

Parameters

lOsError

An operating-system-specific error code.

lpszFileName

A pointer to the string containing the name of the file that caused the exception, if available.

Example

```
//example for CFileException::ThrowOsError
CFileException::ThrowOsError( 5 );  // "access denied"
```

See Also

Reference

CFileException Class
Hierarchy Chart
CFileException::ThrowErrno

Other Resources

CFileException Members

CFileException Data Members

For information about the data members in **CFileException**, see CFileException Members.

CFileException::m_cause

Contains values defined by a **CFileException** enumerated type.

```
int m_cause;
```

Remarks

This data member is a public variable of type **int**. The enumerators and their meanings are as follows:

- **CFileException::none** No error occurred.
- **CFileException::genericException** An unspecified error occurred.
- **CFileException::fileNotFound** The file could not be located.
- **CFileException::badPath** All or part of the path is invalid.
- **CFileException::tooManyOpenFiles** The permitted number of open files was exceeded.
- **CFileException::accessDenied** The file could not be accessed.
- **CFileException::invalidFile** There was an attempt to use an invalid file handle.
- **CFileException::removeCurrentDir** The current working directory cannot be removed.
- **CFileException::directoryFull** There are no more directory entries.
- **CFileException::badSeek** There was an error trying to set the file pointer.
- **CFileException::hardIO** There was a hardware error.
- **CFileException::sharingViolation** SHARE.EXE was not loaded, or a shared region was locked.
- **CFileException::lockViolation** There was an attempt to lock a region that was already locked.
- **CFileException::diskFull** The disk is full.
- **CFileException::endOfFile** The end of file was reached.

Note

These **CFileException** cause enumerators are distinct from the **CArchiveException** cause enumerators.

Note

CArchiveException::generic is deprecated. Use **genericException** instead. If **generic** is used in an application and b uilt with /clr, there will be syntax errors that are not easy to decipher.

Example

```
//example for CFileException::m_cause
try
{
   CFile f( pFileName, CFile::modeCreate | CFile::modeWrite );
}

catch( CFileException* e )
{
   if( e->m_cause == CFileException::fileNotFound )
      printf( "ERROR: File not found\n")
   e->Delete();
}
```

See Also

Reference

CFileException Class

Hierarchy Chart

Other Resources

CFileException Members

CFileException::m_lOsError

Contains the operating-system error code for this exception.

```
LONG m_lOsError;
```

Remarks

See your operating-system technical manual for a listing of error codes. This data member is a public variable of type **LONG**.

See Also

Reference

CFileException Class

Hierarchy Chart

Other Resources

CFileException Members

CFileException::m_strFileName

Contains the name of the file for this exception condition.

```
CString m_strFileName;
```

See Also

Reference

CFileException Class

Hierarchy Chart

Other Resources

CFileException Members

CFileFind Class

Performs local file searches and is the base class for CGopherFileFind and CFtpFileFind, which perform Internet file searches.

```
class CFileFind : public CObject
```

Remarks

CFileFind includes member functions that begin a search, locate a file, and return the title, name, or path of the file. For Internet searches, the member function GetFileURL returns the file's URL.

CFileFind is the base class for two other MFC classes designed to search particular server types: **CGopherFileFind** works specifically with gopher servers, and **CFtpFileFind** works specifically with FTP servers. Together, these three classes provide a seamless mechanism for the client to find files, regardless of the server protocol, the file type, or location, on either a local machine or a remote server.

The following code will enumerate all the files in the current directory, printing the name of each file:

```
CFileFind finder;
BOOL bWorking = finder.FindFile("*.*");
while (bWorking)
{
   bWorking = finder.FindNextFile();
   cout << (LPCTSTR) finder.GetFileName() << endl;
}
```

To keep the example simple, this code uses the standard C++ library **cout** class. The **cout** line could be replaced with a call to **CListBox::AddString**, for example, in a program with a graphical user interface.

For more information about how to use **CFileFind** and the other WinInet classes, see the article Internet Programming with WinInet.

Requirements

Header: afx.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class
Hierarchy Chart
CFtpFileFind Class
CGopherFileFind Class
CInternetFile Class
CGopherFile Class
CHttpFile Class

Other Resources

CFileFind Members

CFileFind Members

Base Class Members

CObject Members

Construction

CFileFind	Constructs a **CFileFind** object.

Attributes

GetCreationTime	Gets the time the file was created.
GetFileName	Gets the name, including the extension, of the found file
GetFilePath	Gets the whole path of the found file.
GetFileTitle	Gets the title of the found file. The title does not include the extension.
GetFileURL	Gets the URL, including the file path, of the found file.
GetLastAccessTime	Gets the time that the file was last accessed.
GetLastWriteTime	Gets the time the file was last changed and saved.
GetLength	Gets the length of the found file, in bytes.
GetRoot	Gets the root directory of the found file.
IsArchived	Determines if the found file is archived.
IsCompressed	Determines if the found file is compressed.
IsDirectory	Determines if the found file is a directory.
IsDots	Determines if the name of the found file has the name "." or "..", indicating that is actually a directory.
IsHidden	Determines if the found file is hidden.
IsNormal	Determines if the found file is normal (in other words, has no other attributes).
IsReadOnly	Determines if the found file is read-only.
IsSystem	Determines if the found file is a system file.
IsTemporary	Determines if the found file is temporary.
MatchesMask	Indicates the desired file attributes of the file to be found.

Operations

Close	Closes the search request.
CloseContext	Closes the file specified by the current search handle.

FindFile	Searches a directory for a specified file name.
FindNextFile	Continues a file search from a previous call to FindFile.

See Also

Reference

CFileFind Class
Hierarchy Chart

CFileFind Member Functions

For information about the member functions in **CFileFind**, see CFileFind Members.

CFileFind::CFileFind

This member function is called when a **CFileFind** object is constructed.

```
CFileFind( );
```

Example

See the example for CFileFind::GetFileName.

See Also

Reference

CFileFind Class
Hierarchy Chart
CGopherFileFind Class
CFtpFileFind Class

Other Resources

CFileFind Members

CFileFind::Close

Call this member function to end the search, reset the context, and release all resources.

```
void Close( );
```

Remarks

After calling **Close**, you do not have to create a new **CFileFind** instance before calling FindFile to begin a new search.

Example

See the example for CFileFind::GetFileName.

See Also

Reference

CFileFind Class

Hierarchy Chart

Other Resources

CFileFind Members

CFileFind::CloseContext

Closes the file specified by the current search handle.

```
virtual void CloseContext( );
```

Remarks

Closes the file specified by the current value of the search handle. Override this function to change the default behavior.

You must call the FindFile or FindNextFile functions at least once to retrieve a valid search handle. The **FindFile** and **FindNextFile** functions use the search handle to locate files with names that match a given name.

See Also

Reference

CFileFind Class
Hierarchy Chart
CFileFind::FindNextFile
CFileFind::FindFile

Other Resources

CFileFind Members

CFileFind::FindFile

Call this member function to open a file search.

```
virtual BOOL FindFile(
   LPCTSTR pstrName = NULL,
   DWORD dwUnused = 0
);
```

Parameters

pstrName

A pointer to a string containing the name of the file to find. If you pass **NULL** for *pstrName*, **FindFile** does a wildcard (*.*) search.

dwUnused

Reserved to make **FindFile** polymorphic with derived classes. Must be 0.

Return Value

Nonzero if successful; otherwise 0. To get extended error information, call the Win32 function GetLastError.

Remarks

After calling **FindFile** to begin the file search, call FindNextFile to retrieve subsequent files. You must call **FindNextFile** at least once before calling any of the following attribute member functions:

- GetCreationTime
- GetFileName
- GetFileTitle
- GetFilePath
- GetFileURL
- GetLastAccessTime
- GetLastWriteTime
- GetLength
- GetRoot
- IsArchived
- IsCompressed
- IsDirectory
- IsDots
- IsHidden
- IsNormal
- IsReadOnly
- IsSystem
- IsTemporary
- MatchesMask

Example

See the example for CFileFind::IsDirectory.

See Also

Reference

CFileFind Class

Hierarchy Chart

CFileFind::FindNextFile

Other Resources

CFileFind Members

CFileFind::FindNextFile

Call this member function to continue a file search from a previous call to FindFile.

```
virtual BOOL FindNextFile( );
```

Return Value

Nonzero if there are more files; zero if the file found is the last one in the directory or if an error occurred. To get extended error information, call the Win32 function GetLastError. If the file found is the last file in the directory, or if no matching files can be found, the **GetLastError** function returns ERROR_NO_MORE_FILES.

Remarks

You must call **FindNextFile** at least once before calling any of the following attribute member functions:

- GetCreationTime
- GetFileName
- GetFileTitle
- GetFilePath
- GetFileURL
- GetLastAccessTime
- GetLastWriteTime
- GetLength
- GetRoot
- IsArchived
- IsCompressed
- IsDirectory
- IsDots
- IsHidden
- IsNormal
- IsReadOnly
- IsSystem
- IsTemporary
- MatchesMask

FindNextFile wraps the Win32 function FindNextFile.

Example

See the example for CFileFind::IsDirectory.

See Also

Reference

CFileFind Class

Hierarchy Chart

Other Resources

CFileFind Members

CFileFind::GetCreationTime

Call this member function to get the time the specified file was created.

```
virtual BOOL GetCreationTime(
   FILETIME* pTimeStamp
) const;
virtual BOOL GetCreationTime(
   CTime& refTime
) const;
```

Parameters

pTimeStamp

A pointer to a FILETIME structure containing the time the file was created.

refTime

A reference to a CTime object.

Return Value

Nonzero if successful; 0 if unsuccessful. **GetCreationTime** returns 0 only if FindNextFile has never been called on this **CFileFind** object.

Remarks

You must call FindNextFile at least once before calling **GetCreationTime**.

Note

Not all file systems use the same semantics to implement the time stamp returned by this function. This function may return the same value returned by other time stamp functions if the underlying file system or server does not support keeping the time attribute. See the Win32_FIND_DATA structure for information about time formats. On some operation systems, the returned time is in the time zone local to the machine were the file is located. See the Win32 FileTimeToLocalFileTime API for more information.

Example

See the example for CFileFind::GetLength.

See Also

Reference

CFileFind Class

Hierarchy Chart

Other Resources

CFileFind Members

CFileFind::GetFileName

Call this member function to get the name of the found file.

```
virtual CString GetFileName( ) const;
```

Return Value

The name of the most-recently-found file.

Remarks

You must call FindNextFile at least once before calling GetFileName.

GetFileName is one of three **CFileFind** member functions that return some form of the file name. The following list describes the three and how they vary:

- **GetFileName** returns the file name, including the extension. For example, calling **GetFileName** to generate a user message about the file `c:\myhtml\myfile.txt` returns the file name `myfile.txt`.
- GetFilePath returns the entire path for the file. For example, calling **GetFilePath** to generate a user message about the file `c:\myhtml\myfile.txt` returns the file path `c:\myhtml\myfile.txt`.
- GetFileTitle returns the file name, excluding the file extension. For example, calling **GetFileTitle** to generate a user message about the file `c:\myhtml\myfile.txt` returns the file title `myfile`.

Example

```
CFileFind finder;
static const TCHAR szFileToFind[] = _T("C:\\WINDOWS\\SYSTEM.INI");

BOOL bResult = finder.FindFile(szFileToFind);

if (bResult)
{
   finder.FindNextFile();

   cout << "Root of " << szFileToFind;
   cout << " is " << (LPCTSTR) finder.GetRoot();
   cout << endl;

   cout << "Title of " << szFileToFind;
   cout << " is " << (LPCTSTR) finder.GetFileTitle();
   cout << endl;

   cout << "Path of " << szFileToFind;
   cout << " is " << (LPCTSTR) finder.GetFilePath();
   cout << endl;

   cout << "URL of " << szFileToFind;
   cout << " is " << (LPCTSTR) finder.GetFileURL();
   cout << endl;

   cout << "Name of " << szFileToFind;
   cout << " is " << (LPCTSTR) finder.GetFileName();
   cout << endl;

   finder.Close();
}
else
   cout << "You have no " << szFileToFind << " file." << endl;
```

Output

Assumes that the file C:\WINDOWS\SYSTEM.INI exists:

```
Root of C:\WINDOWS\SYSTEM.INI is C:\WINDOWS
Title of C:\WINDOWS\SYSTEM.INI is SYSTEM
Path of C:\WINDOWS\SYSTEM.INI is C:\WINDOWS\SYSTEM.INI
URL of C:\WINDOWS\SYSTEM.INI is file://C:\WINDOWS\SYSTEM.INI
Name of C:\WINDOWS\SYSTEM.INI is SYSTEM.INI
```

See Also

Reference

CFileFind Class

Hierarchy Chart

CFileFind::FindFile

Other Resources

CFileFind Members

CFileFind::GetFilePath

Call this member function to get the full path of the specified file.

```
virtual CString GetFilePath( ) const;
```

Return Value

The path of the specified file.

Remarks

You must call FindNextFile at least once before calling **GetFilePath**.

GetFilePath is one of three **CFileFind** member functions that return some form of the file name. The following list describes the three and how they vary:

- GetFileName returns the file name, including the extension. For example, calling **GetFileName** to generate a user message about the file `c:\myhtml\myfile.txt` returns the file name `myfile.txt`.
- **GetFilePath** returns the entire path for the file. For example, calling **GetFilePath** to generate a user message about the file `c:\myhtml\myfile.txt` returns the file path `c:\myhtml\myfile.txt`.
- GetFileTitle returns the file name, excluding the file extension. For example, calling **GetFileTitle** to generate a user message about the file `c:\myhtml\myfile.txt` returns the file title `myfile`.

Example

See the example for CFileFind::GetFileName.

See Also

Reference

CFileFind Class
Hierarchy Chart
CFileFind::FindFile

Other Resources

CFileFind Members

CFileFind::GetFileTitle

Call this member function to get the title of the found file.

```
virtual CString GetFileTitle( ) const;
```

Return Value

The title of the file.

Remarks

You must call FindNextFile at least once before calling **GetFileTitle**.

GetFileTitle is one of three **CFileFind** member functions that return some form of the file name. The following list describes the three and how they vary:

- GetFileName returns the file name, including the extension. For example, calling **GetFileName** to generate a user message about the file `c:\myhtml\myfile.txt` returns the file name `myfile.txt`.
- GetFilePath returns the entire path for the file. For example, calling **GetFilePath** to generate a user message about the file `c:\myhtml\myfile.txt` returns the file path `c:\myhtml\myfile.txt`.
- **GetFileTitle** returns the file name, excluding the file extension. For example, calling **GetFileTitle** to generate a user message about the file `c:\myhtml\myfile.txt` returns the file title `myfile`.

Example

See the example for CFileFind::GetFileName.

See Also

Reference

CFileFind Class
Hierarchy Chart
CFileFind::FindFile

Other Resources

CFileFind Members

CFileFind::GetFileURL

Call this member function to retrieve the specified URL.

```
virtual CString GetFileURL( ) const;
```

Return Value

The complete URL.

Remarks

You must call FindNextFile at least once before calling **GetFileURL**.

GetFileURL is similar to the member function GetFilePath, except that it returns the URL in the form `file://path`. For example, calling **GetFileURL** to get the complete URL for `myfile.txt` returns the URL `file://c:\myhtml\myfile.txt`.

Example

See the example for CFileFind::GetFileName.

See Also

Reference

CFileFind Class

Hierarchy Chart

CFileFind::FindFile

Other Resources

CFileFind Members

CFileFind::GetLastAccessTime

Call this member function to get the time that the specified file was last accessed.

```
virtual BOOL GetLastAccessTime(
   CTime& refTime
) const;
virtual BOOL GetLastAccessTime(
   FILETIME* pTimeStamp
) const;
```

Parameters

refTime

A reference to a CTime object.

pTimeStamp

A pointer to a FILETIME structure containing the time the file was last accessed.

Return Value

Nonzero if successful; 0 if unsuccessful. **GetLastAccessTime** returns 0 only if FindNextFile has never been called on this **CFileFind** object.

Remarks

You must call FindNextFile at least once before calling **GetLastAccessTime**.

Note

Not all file systems use the same semantics to implement the time stamp returned by this function. This function may return the same value returned by other time stamp functions if the underlying file system or server does not support keeping the time attribute. See the Win32_FIND_DATA structure for information about time formats. On some operation systems, the returned time is in the time zone local to the machine were the file is located. See the Win32 FileTimeToLocalFileTime API for more information.

Example

See the example for CFileFind::GetLength.

See Also

Reference

CFileFind Class

Hierarchy Chart

Other Resources

CFileFind Members

CFileFind::GetLastWriteTime

Call this member function to get the last time the file was changed.

```
virtual BOOL GetLastWriteTime(
   FILETIME* pTimeStamp
) const;
virtual BOOL GetLastWriteTime(
   CTime& refTime
) const;
```

Parameters

pTimeStamp

A pointer to a FILETIME structure containing the time the file was last written to.

refTime

A reference to a CTime object.

Return Value

Nonzero if successful; 0 if unsuccessful. **GetLastWriteTime** returns 0 only if FindNextFile has never been called on this **CFileFind** object.

Remarks

You must call FindNextFile at least once before calling **GetLastWriteTime**.

Note

Not all file systems use the same semantics to implement the time stamp returned by this function. This function may return the same value returned by other time stamp functions if the underlying file system or server does not support keeping the time attribute. See the Win32_Find_Data structure for information about time formats. On some operation systems, the returned time is in the time zone local to the machine were the file is located. See the Win32 FileTimeToLocalFileTime API for more information.

Example

See the example for CFileFind::GetLength.

See Also

Reference

CFileFind Class

Hierarchy Chart

Other Resources

CFileFind Members

CFileFind::GetLength

Call this member function to get the length of the found file, in bytes.

```
ULONGLONG GetLength( ) const;
```

Return Value

The length of the found file, in bytes.

Remarks

You must call FindNextFile at least once before calling **GetLength**.

GetLength uses the Win32 structure WIN32_FIND_DATA to get and return the value of the file size, in bytes.

Note

As of MFC 7.0, **GetLength** supports 64-bit integer types. Previously existing code built with this newer version of the library may result in truncation warnings.

Example

```
// This code fragment prints out a very verbose directory
// listing for all the files in the root directory on the
// C: drive. After the file's name, each attribute of the
// file is printed, as are the creation, last access, and
// last write times.

   CFileFind finder;

   BOOL bWorking = finder.FindFile(_T("C:\\*.*"));

   while (bWorking)
   {
      bWorking = finder.FindNextFile();

      _tprintf_s(_T("%s\n\t"), (LPCTSTR) finder.GetFileName());
      _tprintf_s(_T("%c"), finder.IsArchived()   ? 'A' : 'a');
      _tprintf_s(_T("%c"), finder.IsCompressed() ? 'C' : 'c');
      _tprintf_s(_T("%c"), finder.IsHidden()     ? 'H' : 'h');
      _tprintf_s(_T("%c"), finder.IsNormal()     ? 'N' : 'n');
      _tprintf_s(_T("%c"), finder.IsReadOnly()   ? 'R' : 'r');
      _tprintf_s(_T("%c"), finder.IsSystem()     ? 'S' : 's');
      _tprintf_s(_T("%c"), finder.IsTemporary()  ? 'T' : 't');

      _tprintf_s(_T("\t%I64u byte(s)\n"), finder.GetLength());

      CTime tempTime;
      CString str;

      _tprintf_s(_T("\tCreated    : "));
      if (finder.GetCreationTime(tempTime))
      {
         str = tempTime.Format(_T("%c"));
         _tprintf_s(_T("%s\n"), (LPCTSTR) str);
      }
      else
         _tprintf_s(_T("(unavailable)\n"));

      _tprintf_s(_T("\tLast Access: "));
      if (finder.GetLastAccessTime(tempTime))
      {
         str = tempTime.Format(_T("%c"));
         _tprintf_s(_T("%s\n"), (LPCTSTR) str);
```

```
        }
        else
           _tprintf_s(_T("(unavailable)\n"));

        _tprintf_s(_T("\tLast Write : "));
        if (finder.GetLastWriteTime(tempTime))
        {
           str = tempTime.Format(_T("%c"));
           _tprintf_s(_T("%s\n"), (LPCTSTR) str);
        }
        else
           _tprintf_s(_T("(unavailable)\n"));

        _tprintf_s(_T("\n"));
     }
```

See Also

Reference

CFileFind Class

Hierarchy Chart

Other Resources

CFileFind Members

CFileFind::GetRoot

Call this member function to get the root of the found file.

```
virtual CString GetRoot( ) const;
```

Return Value

The root of the active search.

Remarks

You must call FindNextFile at least once before calling **GetRoot**.

This member function returns the drive specifier and path name used to start a search. For example, calling FindFile with `*.dat` results in **GetRoot** returning an empty string. Passing a path, such as `c:\windows\system\*.dll`, to **FindFile** results **GetRoot** returning `c:\windows\system\`.

Example

See the example for CFileFind::GetFileName.

See Also

Reference

CFileFind Class

Hierarchy Chart

Other Resources

CFileFind Members

CFileFind::IsArchived

Call this member function to determine if the found file is archived.

```
BOOL IsArchived( ) const;
```

Return Value

Nonzero if successful; otherwise 0.

Remarks

Applications mark an archive file, which is to be backed up or removed, with FILE_ATTRIBUTE_ARCHIVE, a file attribute identified in the WIN32_FIND_DATA structure.

You must call FindNextFile at least once before calling **IsArchived**.

See the member function MatchesMask for a complete list of file attributes.

Example

See the example for CFileFind::GetLength.

See Also

Reference

CFileFind Class

Hierarchy Chart

Other Resources

CFileFind Members

CFileFind::IsCompressed

Call this member function to determine if the found file is compressed.

```
BOOL IsCompressed( ) const;
```

Return Value

Nonzero if successful; otherwise 0.

Remarks

A compressed file is marked with FILE_ATTRIBUTE_COMPRESSED, a file attribute identified in the WIN32_FIND_DATA structure. For a file, this attribute indicates that all of the data in the file is compressed. For a directory, this attribute indicates that compression is the default for newly created files and subdirectories.

You must call FindNextFile at least once before calling **IsCompressed**.

See the member function MatchesMask for a complete list of file attributes.

Example

See the example for CFileFind::GetLength.

See Also

Reference

CFileFind Class

Hierarchy Chart

Other Resources

CFileFind Members

CFileFind::IsDirectory

Call this member function to determine if the found file is a directory.

```
BOOL IsDirectory( ) const;
```

Return Value

Nonzero if successful; otherwise 0.

Remarks

A file that is a directory is marked with FILE_ATTRIBUTE_DIRECTORY a file attribute identified in the WIN32_FIND_DATA structure.

You must call FindNextFile at least once before calling **IsDirectory**.

See the member function MatchesMask for a complete list of file attributes.

Example

This small program recurses every directory on the C:\ drive and prints the name of the directory.

```
#include <afxwin.h>
#include <iostream>

using namespace std;

void Recurse(LPCTSTR pstr)
{
   CFileFind finder;

   // build a string with wildcards
   CString strWildcard(pstr);
   strWildcard += _T("\\*.*");

   // start working for files
   BOOL bWorking = finder.FindFile(strWildcard);

   while (bWorking)
   {
      bWorking = finder.FindNextFile();

      // skip . and .. files; otherwise, we'd
      // recur infinitely!

      if (finder.IsDots())
         continue;

      // if it's a directory, recursively search it

      if (finder.IsDirectory())
      {
         CString str = finder.GetFilePath();
         cout << (LPCTSTR) str << endl;
         Recurse(str);
      }
   }

   finder.Close();
}

int main()
{
   if (!AfxWinInit(GetModuleHandle(NULL), NULL, GetCommandLine(), 0))
```

```
        cout << "panic!" << endl;
    else
        Recurse(_T("C:"));
}
```

See Also

Reference

CFileFind Class

Hierarchy Chart

Other Resources

CFileFind Members

CFileFind::IsDots

Call this member function to test for the current directory and parent directory markers while iterating through files.

```
virtual BOOL IsDots( ) const;
```

Return Value

Nonzero if the found file has the name "." or "..", which indicates that the found file is actually a directory. Otherwise 0.

Remarks

You must call FindNextFile at least once before calling **IsDots**.

Example

See the example for CFileFind::IsDirectory.

See Also

Reference

CFileFind Class
Hierarchy Chart
CFileFind::IsDirectory

Other Resources

CFileFind Members

CFileFind::IsHidden

Call this member function to determine if the found file is hidden.

```
BOOL IsHidden( ) const;
```

Return Value

Nonzero if successful; otherwise 0.

Remarks

Hidden files, which are marked with FILE_ATTRIBUTE_HIDDEN, a file attribute identified in the WIN32_FIND_DATA structure. A hidden file is not included in an ordinary directory listing.

You must call FindNextFile at least once before calling **IsHidden**.

See the member function MatchesMask for a complete list of file attributes.

Example

See the example for CFileFind::GetLength.

See Also

Reference

CFileFind Class

Hierarchy Chart

Other Resources

CFileFind Members

CFileFind::IsNormal

Call this member function to determine if the found file is a normal file.

```
BOOL IsNormal( ) const;
```

Return Value

Nonzero if successful; otherwise 0.

Remarks

Files marked with FILE_ATTRIBUTE_NORMAL, a file attribute identified in the WIN32_FIND_DATA structure. A normal file has no other attributes set. All other file attributes override this attribute.

You must call FindNextFile at least once before calling **IsNormal**.

See the member function MatchesMask for a complete list of file attributes.

Example

See the example for CFileFind::GetLength.

See Also

Reference

CFileFind Class

Hierarchy Chart

Other Resources

CFileFind Members

CFileFind::IsReadOnly

Call this member function to determine if the found file is read-only.

```
BOOL IsReadOnly( ) const;
```

Return Value

Nonzero if successful; otherwise 0.

Remarks

A read-only file is marked with FILE_ATTRIBUTE_READONLY, a file attribute identified in the WIN32_FIND_DATA structure. Applications can read such a file, but they cannot write to it or delete it.

You must call FindNextFile at least once before calling **IsReadOnly**.

See the member function MatchesMask for a complete list of file attributes.

Example

See the example for CFileFind::GetLength.

See Also

Reference

CFileFind Class

Hierarchy Chart

Other Resources

CFileFind Members

CFileFind::IsSystem

Call this member function to determine if the found file is a system file.

```
BOOL IsSystem( ) const;
```

Return Value

Nonzero if successful; otherwise 0.

Remarks

A system file is marked with FILE_ATTRIBUTE_SYSTEM, , a file attribute identified in the WIN32_FIND_DATA structure. A system file is part of, or is used exclusively by, the operating system.

You must call FindNextFile at least once before calling **IsSystem**.

See the member function MatchesMask for a complete list of file attributes.

Example

See the example for CFileFind::GetLength.

See Also

Reference

CFileFind Class

Hierarchy Chart

Other Resources

CFileFind Members

CFileFind::IsTemporary

Call this member function to determine if the found file is a temporary file.

```
BOOL IsTemporary( ) const;
```

Return Value

Nonzero if successful; otherwise 0.

Remarks

A temporary file is marked with FILE_ATTRIBUTE_TEMPORARY, a file attribute identified in the WIN32_FIND_DATA structure. A temporary file is used for temporary storage. Applications should write to the file only if absolutely necessary. Most of the file's data remains in memory without being flushed to the media because the file will soon be deleted.

You must call FindNextFile at least once before calling **IsTemporary**.

See the member function MatchesMask for a complete list of file attributes.

Example

See the example for CFileFind::GetLength.

See Also

Reference

CFileFind Class

Hierarchy Chart

Other Resources

CFileFind Members

CFileFind::MatchesMask

Call this member function to test the file attributes on the found file.

```
virtual BOOL MatchesMask(
   DWORD dwMask
) const;
```

Parameters

dwMask

Specifies one or more file attributes, identified in the WIN32_FIND_DATA structure, for the found file. To search for multiple attributes, use the bitwise OR (|) operator. Any combination of the following attributes is acceptable:

- FILE_ATTRIBUTE_ARCHIVE The file is an archive file. Applications use this attribute to mark files for backup or removal.
- FILE_ATTRIBUTE_COMPRESSED The file or directory is compressed. For a file, this means that all of the data in the file is compressed. For a directory, this means that compression is the default for newly created files and subdirectories.
- FILE_ATTRIBUTE_DIRECTORY The file is a directory.
- FILE_ATTRIBUTE_NORMAL The file has no other attributes set. This attribute is valid only if used alone. All other file attributes override this attribute.
- FILE_ATTRIBUTE_HIDDEN The file is hidden. It is not to be included in an ordinary directory listing.
- FILE_ATTRIBUTE_READONLY The file is read only. Applications can read the file but cannot write to it or delete it.
- FILE_ATTRIBUTE_SYSTEM The file is part of or is used exclusively by the operating system.
- FILE_ATTRIBUTE_TEMPORARY The file is being used for temporary storage. Applications should write to the file only if absolutely necessary. Most of the file's data remains in memory without being flushed to the media because the file will soon be deleted.

Return Value

Nonzero if successful; otherwise 0. To get extended error information, call the Win32 function GetLastError.

Remarks

You must call FindNextFile at least once before calling **MatchesMask**.

Example

```
// cfilefind_matchesmask.cpp
// compile with: /W0
// This code fragment shows all of the files in the root directory
// of drive C: which have either the hidden attribute or the system
// attribute, or both.
#include <string.h>
#include <afx.h>

int main(int argc, char* argv[])
{
    CFileFind finder;

    BOOL bWorking = finder.FindFile(_T("C:\\*.*"));

    while (bWorking)
    {
        bWorking = finder.FindNextFile();

        if (finder.MatchesMask(FILE_ATTRIBUTE_HIDDEN |
                               FILE_ATTRIBUTE_SYSTEM))
        {
            _tprintf_s(_T("%s\n"), (LPCTSTR) finder.GetFileName());
```

```
        }
    }


    return 0;
}
```

See Also

Reference

CFileFind Class
Hierarchy Chart
CFileFind::IsDots
CFileFind::IsReadOnly
CFileFind::IsDirectory
CFileFind::IsCompressed
CFileFind::IsSystem
CFileFind::IsHidden
CFileFind::IsTemporary
CFileFind::IsNormal
CFileFind::IsArchived

Other Resources

CFileFind Members

CFindReplaceDialog Class

This topic has been updated for Visual Studio 2005 SP1.

Allows you to implement standard string Find/Replace dialog boxes in your application.

```
class CFindReplaceDialog : public CCommonDialog
```

Remarks

Unlike the other Windows common dialog boxes, **CFindReplaceDialog** objects are modeless, allowing users to interact with other windows while they are on screen. There are two kinds of **CFindReplaceDialog** objects: Find dialog boxes and Find/Replace dialog boxes. Although the dialog boxes allow the user to input search and search/replace strings, they do not perform any of the searching or replacing functions. You must add these to the application.

To construct a **CFindReplaceDialog** object, use the provided constructor (which has no arguments). Since this is a modeless dialog box, allocate the object on the heap using the **new** operator, rather than on the stack.

Once a **CFindReplaceDialog** object has been constructed, you must call the Create member function to create and display the dialog box.

Use the m_fr structure to initialize the dialog box before calling **Create**. The **m_fr** structure is of type FINDREPLACE. For more information on this structure, see the *Platform SDK*.

In order for the parent window to be notified of find/replace requests, you must use the Windows RegisterWindowMessage function and use the ON_REGISTERED_MESSAGE message-map macro in your frame window that handles this registered message. You can call any of the member functions listed in the "Operations" section of the **See Also**

CFindReplaceDialog Class Members table from the frame window's callback function.

You can determine whether the user has decided to terminate the dialog box with the **IsTerminating** member function.

CFindReplaceDialog relies on the COMMDLG.DLL file that ships with Windows versions 3.1 and later.

To customize the dialog box, derive a class from **CFindReplaceDialog**, provide a custom dialog template, and add a message map to process the notification messages from the extended controls. Any unprocessed messages should be passed to the base class.

Customizing the hook function is not required.

For more information on using **CFindReplaceDialog**, see Common Dialog Classes.

Requirements

Header: afxdlgs.h

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

See Also

Reference

CCommonDialog Class

Hierarchy Chart

Other Resources

CFindReplaceDialog Members

CFindReplaceDialog Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CDialog Members

CCommonDialog Members

Data Members

m_fr	A structure used to customize a **CFindReplaceDialog** object.

Construction

CFindReplaceDialog	Call this function to construct a **CFindReplaceDialog** object.
Create	Creates and displays a **CFindReplaceDialog** dialog box.

Operations

FindNext	Call this function to determine whether the user wants to find the next occurrence of the find string.
GetFindString	Call this function to retrieve the current find string.
GetNotifier	Call this function to retrieve the **FINDREPLACE** structure in your registered message handler.
GetReplaceString	Call this function to retrieve the current replace string.
IsTerminating	Call this function to determine whether the dialog box is terminating.
MatchCase	Call this function to determine whether the user wants to match the case of the find string exactly.
MatchWholeWord	Call this function to determine whether the user wants to match entire words only.
ReplaceAll	Call this function to determine whether the user wants all occurrences of the string to be replaced.
ReplaceCurrent	Call this function to determine whether the user wants the current word to be replaced.
SearchDown	Call this function to determine whether the user wants the search to proceed in a downward direction.

See Also

Reference

CFindReplaceDialog Class

Hierarchy Chart

CFindReplaceDialog Member Functions

For information about the member functions in **CFindReplaceDialog**, see CFindReplaceDialog Members.

CFindReplaceDialog::CFindReplaceDialog

Constructs a **CFindReplaceDialog** object.

```
CFindReplaceDialog( );
```

Remarks

CFindReplaceDialog objects are constructed on the heap with the **new** operator. For more information on the construction of **CFindReplaceDialog** objects, see the CFindReplaceDialog overview. Use the **Create** member function to display the dialog box.

Example

```
// m_pFRDlg is a pointer to a class derived from CFindReplaceDialog
// which defines variables used by the FINDREPLACE structure.
// InitFindReplaceDlg creates a CFindReplaceDialog and initializes
// the m_fr with the data members from the derived class
void CRichEditView::InitFindReplaceDlg()
{
   if( NULL == m_pFRDlg )
   {
      m_pFRDlg = new CMyFindReplaceDlg();  // Must be created on the heap

      m_pFRDlg->Create( TRUE, "", "", FR_DOWN, this );

      m_pFRDlg->m_fr.lStructSize = sizeof(FINDREPLACE);
      m_pFRDlg->m_fr.hwndOwner = this->m_hWnd;
      m_pFRDlg->m_fr.lpstrFindWhat = m_pFRDlg->GetFindWhatStr();
      m_pFRDlg->m_fr.lpstrReplaceWith = m_pFRDlg->GetReplaceWithStr();
      m_pFRDlg->m_fr.wFindWhatLen =  m_pFRDlg->GetFindWhatStrLen();
      m_pFRDlg->m_fr.wReplaceWithLen =  m_pFRDlg->GetReplaceWithStrLen();
   }
}
```

See Also

Reference

CFindReplaceDialog Class
Hierarchy Chart
CFindReplaceDialog::Create

Other Resources

CFindReplaceDialog Members

CFindReplaceDialog::Create

Creates and displays either a Find or Find/Replace dialog box object, depending on the value of *bFindDialogOnly*.

```
virtual BOOL Create(
   BOOL bFindDialogOnly,
   LPCTSTR lpszFindWhat,
   LPCTSTR lpszReplaceWith = NULL,
   DWORD dwFlags = FR_DOWN,
   CWnd* pParentWnd = NULL
);
```

Parameters

bFindDialogOnly

Set this parameter to **TRUE** to display the standard Windows Find dialog box. Set it to **FALSE** to display the Windows Find/Replace dialog box.

lpszFindWhat

Specifies the string for which to search.

lpszReplaceWith

Specifies the default string with which to replace found strings.

dwFlags

One or more flags you can use to customize the settings of the dialog box, combined using the bitwise OR operator. The default value is **FR_DOWN**, which specifies that the search is to proceed in a downward direction. See the FINDREPLACE structure in the *Platform SDK* for more information on these flags.

pParentWnd

A pointer to the dialog box's parent or owner window. This is the window that will receive the special message indicating that a find/replace action is requested. If **NULL**, the application's main window is used.

Return Value

Nonzero if the dialog box object was successfully created; otherwise 0.

Remarks

In order for the parent window to be notified of find/replace requests, you must use the Windows RegisterWindowMessage function whose return value is a message number unique to the application's instance. Your frame window should have a message map entry that declares the callback function (**OnFindReplace** in the example that follows) that handles this registered message. The following code fragment is an example of how to do this for a frame window class named `CMyFrameWnd`:

```
class CMyFrameWnd : public CFrameWnd
{
protected:
    afx_msg LONG OnFindReplace(WPARAM wParam, LPARAM lParam);

    DECLARE_MESSAGE_MAP()
};
static UINT WM_FINDREPLACE = ::RegisterWindowMessage(FINDMSGSTRING);

BEGIN_MESSAGE_MAP( CMyFrameWnd, CFrameWnd )
   //Normal message map entries here.
   ON_REGISTERED_MESSAGE( WM_FINDREPLACE, OnFindReplace )
END_MESSAGE_MAP
```

Within your **OnFindReplace** function, you interpret the intentions of the user and create the code for the find/replace operations.

Example

See the example for CFindReplaceDialog::CFindReplaceDialog.

See Also

Reference

CFindReplaceDialog Class
Hierarchy Chart
CFindReplaceDialog::CFindReplaceDialog

Other Resources

CFindReplaceDialog Members

CFindReplaceDialog::FindNext

Call this function from your callback function to determine whether the user wants to find the next occurrence of the search string.

```
BOOL FindNext( ) const;
```

Return Value

Nonzero if the user wants to find the next occurrence of the search string; otherwise 0.

See Also

Reference

CFindReplaceDialog Class
Hierarchy Chart
CFindReplaceDialog::GetFindString
CFindReplaceDialog::SearchDown

Other Resources

CFindReplaceDialog Members

CFindReplaceDialog::GetFindString

Call this function from your callback function to retrieve the default string to find.

```
CString GetFindString( ) const;
```

Return Value

The default string to find.

Example

```
// m_pFRDlg is a pointer to a class derived from CFindReplaceDialog

if( m_pFRDlg->IsTerminating() )
{
   CString   csFindString;
   CString   csReplaceString;

   csFindString = m_pFRDlg->GetFindString();
   csReplaceString = m_pFRDlg->GetReplaceString();

   VERIFY( AfxGetApp()->WriteProfileString( AfxGetApp()->m_pszAppName,
      "FindString", csFindString ) );
   VERIFY( AfxGetApp()->WriteProfileString( AfxGetApp()->m_pszAppName,
      "ReplaceString", csReplaceString ) );

   VERIFY( m_pFRDlg->DestroyWindow() );
}
```

See Also

Reference

CFindReplaceDialog Class
Hierarchy Chart
CFindReplaceDialog::FindNext
CFindReplaceDialog::GetReplaceString

Other Resources

CFindReplaceDialog Members

CFindReplaceDialog::GetNotifier

Call this function to retrieve a pointer to the current Find Replace dialog box.

```
static CFindReplaceDialog* PASCAL GetNotifier(
   LPARAM lParam
);
```

Parameters

lParam

The **lparam** value passed to the frame window's **OnFindReplace** member function.

Return Value

A pointer to the current dialog box.

Remarks

It should be used within your callback function to access the current dialog box, call its member functions, and access the **m_fr** structure.

Example

```
// In mainframe.h

protected:
   //{{AFX_MSG(CMyRichEditTestView)
      ...

   //}}AFX_MSG
   afx_msg LONG OnFindReplace(WPARAM wParam, LPARAM lParam);
   DECLARE_MESSAGE_MAP()

// Register the FINDMSGSTRING message
static WM_FINDREPLACE = ::RegisterWindowMessage(FINDMSGSTRING);

// In mainframe.cpp

BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
//{{AFX_MSG_MAP
...

//}}AFX_MSG_MAP
ON_REGISTERED_MESSAGE(WM_FINDREPLACE, OnFindReplace)


LRESULT CMainFrame::OnFindReplace(WPARAM wparam, LPARAM lparam)
{
   CFindReplaceDialog *pDlg = CFindReplaceDialog::GetNotifier(lparam)

   if( NULL != pDlg )
   {
      // Use pDlg as a pointer to the existing FindReplace dlg to
      // call CFindReplaceDialog member functions
   }
}
```

See Also

Reference

CFindReplaceDialog Class

Hierarchy Chart

Other Resources

CFindReplaceDialog Members

CFindReplaceDialog::GetReplaceString

Call this function to retrieve the current replace string.

```
CString GetReplaceString( ) const;
```

Return Value

The default string with which to replace found strings.

Example

See the example for CFindReplaceDialog::GetFindString.

See Also

Reference

CFindReplaceDialog Class
Hierarchy Chart
CFindReplaceDialog::GetFindString

Other Resources

CFindReplaceDialog Members

CFindReplaceDialog::IsTerminating

Call this function within your callback function to determine whether the user has decided to terminate the dialog box.

```
BOOL IsTerminating( ) const;
```

Return Value

Nonzero if the user has decided to terminate the dialog box; otherwise 0.

Remarks

If this function returns nonzero, you should call the **DestroyWindow** member function of the current dialog box and set any dialog box pointer variable to **NULL**. Optionally, you can also store the find/replace text last entered and use it to initialize the next find/replace dialog box.

Example

See the example for CFindReplaceDialog::GetFindString.

See Also

Reference

CFindReplaceDialog Class

Hierarchy Chart

Other Resources

CFindReplaceDialog Members

CFindReplaceDialog::MatchCase

Call this function to determine whether the user wants to match the case of the find string exactly.

```
BOOL MatchCase( ) const;
```

Return Value

Nonzero if the user wants to find occurrences of the search string that exactly match the case of the search string; otherwise 0.

See Also

Reference

CFindReplaceDialog Class
Hierarchy Chart
CFindReplaceDialog::MatchWholeWord

Other Resources

CFindReplaceDialog Members

CFindReplaceDialog::MatchWholeWord

Call this function to determine whether the user wants to match entire words only.

```
BOOL MatchWholeWord( ) const;
```

Return Value

Nonzero if the user wants to match only the entire words of the search string; otherwise 0.

See Also

Reference

CFindReplaceDialog Class
Hierarchy Chart
CFindReplaceDialog::MatchCase

Other Resources

CFindReplaceDialog Members

CFindReplaceDialog::ReplaceAll

Call this function to determine whether the user wants all occurrences of the string to be replaced.

```
BOOL ReplaceAll( ) const;
```

Return Value

Nonzero if the user has requested that all strings matching the replace string be replaced; otherwise 0.

See Also

Reference

CFindReplaceDialog Class
Hierarchy Chart
CFindReplaceDialog::ReplaceCurrent

Other Resources

CFindReplaceDialog Members

CFindReplaceDialog::ReplaceCurrent

Call this function to determine whether the user wants the current word to be replaced.

```
BOOL ReplaceCurrent( ) const;
```

Return Value

Nonzero if the user has requested that the currently selected string be replaced with the replace string; otherwise 0.

See Also

Reference

CFindReplaceDialog Class
Hierarchy Chart
CFindReplaceDialog::ReplaceAll

Other Resources

CFindReplaceDialog Members

CFindReplaceDialog::SearchDown

Call this function to determine whether the user wants the search to proceed in a downward direction.

```
BOOL SearchDown( ) const;
```

Return Value

Nonzero if the user wants the search to proceed in a downward direction; 0 if the user wants the search to proceed in an upward direction.

See Also

Reference

CFindReplaceDialog Class

Hierarchy Chart

Other Resources

CFindReplaceDialog Members

CFindReplaceDialog Data Members

For information about the data members in **CFindReplaceDialog**, see CFindReplaceDialog Members.

CFindReplaceDialog::m_fr

Used to customize a **CFindReplaceDialog** object.

```
FINDREPLACE m_fr;
```

Remarks

m_fr is a structure of type FINDREPLACE. Its members store the characteristics of the dialog-box object. After constructing a **CFindReplaceDialog** object, you can use **m_fr** to modify various values in the dialog box.

For more information on this structure, see the **FINDREPLACE** structure in the *Platform SDK*.

Example

See the example for CFindReplaceDialog::CFindReplaceDialog.

See Also

Reference

CFindReplaceDialog Class

Hierarchy Chart

Other Resources

CFindReplaceDialog Members

CFont Class

Encapsulates a Windows graphics device interface (GDI) font and provides member functions for manipulating the font.

```
class CFont : public CGdiObject
```

Remarks

To use a **CFont** object, construct a **CFont** object and attach a Windows font to it with CreateFont, CreateFontIndirect, CreatePointFont, or CreatePointFontIndirect, and then use the object's member functions to manipulate the font.

The **CreatePointFont** and **CreatePointFontIndirect** functions are often easier to use than **CreateFont** or **CreateFontIndirect** since they do the conversion for the height of the font from a point size to logical units automatically.

For more information on **CFont**, see Graphic Objects.

Requirements

Header: afxwin.h

See Also

Tasks

HIERSVR Sample: Demonstrates a Server Application with OLE Drag and Drop

Reference

CGdiObject Class

Hierarchy Chart

Other Resources

CFont Members

CFont Members

Base Class Members

CObject Members

CGdiObject Members

Construction

CFont	Constructs a **CFont** object.

Initialization

CreateFont	Initializes a **CFont** with the specified characteristics.
CreateFontIndirect	Initializes a **CFont** object with the characteristics given in a **LOGFONT** structure.
CreatePointFont	Initializes a **CFont** with the specified height, measured in tenths of a point, and typeface.
CreatePointFontIndirect	Same as **CreateFontIndirect** except that the font height is measured in tenths of a point rather than l ogical units.

Operations

FromHandle	Returns a pointer to a **CFont** object when given a Windows **HFONT**.

Attributes

GetLogFont	Fills a **LOGFONT** with information about the logical font attached to the **CFont** object.
operator HFONT	Returns the Windows GDI font handle attached to the **CFont** object.

See Also

Reference

CFont Class

Hierarchy Chart

CFont Member Functions

For information about the member functions in **CFont**, see CFont Members.

CFont::CFont

Constructs a **CFont** object.

```
CFont( );
```

Remarks

The resulting object must be initialized with **CreateFont**, **CreateFontIndirect**, **CreatePointFont**, or **CreatePointFontIndirect** before it can be used.

Example

```
// Declare a CFont object.
CFont font;
```

See Also

Reference

CFont Class
Hierarchy Chart
CFont::CreateFontIndirect
CFont::CreateFont
CFont::CreatePointFont
CFont::CreatePointFontIndirect
EnumFonts

Other Resources

CFont Members

CFont::CreateFont

Initializes a **CFont** object with the specified characteristics.

```
BOOL CreateFont(
   int nHeight,
   int nWidth,
   int nEscapement,
   int nOrientation,
   int nWeight,
   BYTE bItalic,
   BYTE bUnderline,
   BYTE cStrikeOut,
   BYTE nCharSet,
   BYTE nOutPrecision,
   BYTE nClipPrecision,
   BYTE nQuality,
   BYTE nPitchAndFamily,
   LPCTSTR lpszFacename
);
```

Parameters

nHeight

Specifies the desired height (in logical units) of the font. See the **lfHeight** member of the LOGFONTstructure in the *Platform SDK* for a descrition. The absolute value of *nHeight* must not exceed 16,384 device units after it is converted. For all height comparisons, the font mapper looks for the largest font that does not exceed the requested size or the smallest font if all the fonts exceed the requested size.

nWidth

Specifies the average width (in logical units) of characters in the font. If *nWidth* is 0, the aspect ratio of the device will be matched against the digitization aspect ratio of the available fonts to find the closest match, which is determined by the absolute value of the difference.

nEscapement

Specifies the angle (in 0.1-degree units) between the escapement vector and the x-axis of the display surface. The escapement vector is the line through the origins of the first and last characters on a line. The angle is measured counterclockwise from the x-axis. See the **lfEscapement** member in the **LOGFONT** structure in the *Platform SDK* for more information.

nOrientation

Specifies the angle (in 0.1-degree units) between the baseline of a character and the x-axis. The angle is measured counterclockwise from the x-axis for coordinate systems in which the y-direction is down and clockwise from the x-axis for coordinate systems in which the y-direction is up.

nWeight

Specifies the font weight (in inked pixels per 1000). See the **lfWeight** member in the **LOGFONT** structure in the *Platform SDK* for more information. The described values are approximate; the actual appearance depends on the typeface. Some fonts have only **FW_NORMAL**, **FW_REGULAR**, and **FW_BOLD** weights. If **FW_DONTCARE** is specified, a default weight is used.

bItalic

Specifies whether the font is italic.

bUnderline

Specifies whether the font is underlined.

cStrikeOut

Specifies whether characters in the font are struck out. Specifies a strikeout font if set to a nonzero value.

nCharSet

Specifies the font's character setSee the **lfCharSet** member in the **LOGFONT** structure in the *Platform SDK* for a list of values.

The OEM character set is system-dependent.

Fonts with other character sets may exist in the system. An application that uses a font with an unknown character set must not attempt to translate or interpret strings that are to be rendered with that font. Instead, the strings should be passed directly to the output device driver.

The font mapper does not use the **DEFAULT_CHARSET** value. An application can use this value to allow the name and size of a font to fully describe the logical font. If a font with the specified name does not exist, a font from any character set can be substituted for the specified font. To avoid unexpected results, applications should use the **DEFAULT_CHARSET** value sparingly.

nOutPrecision

Specifies the desired output precision. The output precision defines how closely the output must match the requested font's height, width, character orientation, escapement, and pitch. See the **lfOutPrecision** member in the **LOGFONT** structure in the *Platform SDK* for a list of values and more information.

nClipPrecision

Specifies the desired clipping precision. The clipping precision defines how to clip characters that are partially outside the clipping region. See the **lfClipPrecision** member in the **LOGFONT** structure in the *Platform SDK* for a list of values.

To use an embedded read-only font, an application must specify **CLIP_ENCAPSULATE**.

To achieve consistent rotation of device, TrueType, and vector fonts, an application can use the OR operator to combine the **CLIP_LH_ANGLES** value with any of the other *nClipPrecision* values. If the **CLIP_LH_ANGLES** bit is set, the rotation for all fonts depends on whether the orientation of the coordinate system is left-handed or right-handed. (For more information about the orientation of coordinate systems, see the description of the *nOrientation* parameter.) If **CLIP_LH_ANGLES** is not set, device fonts always rotate counterclockwise, but the rotation of other fonts is dependent on the orientation of the coordinate system.

nQuality

Specifies the font's output quality, which defines how carefully the GDI must attempt to match the logical-font attributes to those of an actual physical font. See the **lfQuality** member in the **LOGFONT** structure in the *Platform SDK* for a list of values.

nPitchAndFamily

Specifies the pitch and family of the font. See the **lfPitchAndFamily** member in the **LOGFONT** structure in the *Platform SDK* for a list of values and more information.

lpszFacename

A **CString** or pointer to a null-terminated string that specifies the typeface name of the font. The length of this string must not exceed 30 characters. The Windows **EnumFontFamilies** function can be used to enumerate all currently available fonts. If *lpszFacename* is **NULL**, the GDI uses a device-independent typeface.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The font can subsequently be selected as the font for any device context.

The **CreateFont** function does not create a new Windows GDI font. It merely selects the closest match from the fonts available in the GDI's pool of physical fonts.

Applications can use the default settings for most of these parameters when creating a logical font. The parameters that should always be given specific values are *nHeight* and *lpszFacename*. If *nHeight* and *lpszFacename* are not set by the application, the logical font that is created is device-dependent.

When you finish with the **CFont** object created by the **CreateFont** function, first select the font out of the device context, then delete the **CFont** object.

Example

```
// The code fragment shows how to create a font object,
// select the font object into a DC (device context) for text
// drawing, and finally delete the font object.

// Initializes a CFont object with the specified characteristics.
CFont font;
VERIFY(font.CreateFont(
   12,                        // nHeight
   0,                         // nWidth
   0,                         // nEscapement
   0,                         // nOrientation
   FW_NORMAL,                 // nWeight
   FALSE,                     // bItalic
   FALSE,                     // bUnderline
   0,                         // cStrikeOut
   ANSI_CHARSET,              // nCharSet
   OUT_DEFAULT_PRECIS,        // nOutPrecision
   CLIP_DEFAULT_PRECIS,       // nClipPrecision
   DEFAULT_QUALITY,           // nQuality
   DEFAULT_PITCH | FF_SWISS,  // nPitchAndFamily
   "Arial"));                 // lpszFacename

// Do something with the font just created...
CClientDC dc(this);
CFont* def_font = dc.SelectObject(&font);
dc.TextOut(5, 5, "Hello", 5);
dc.SelectObject(def_font);

// Done with the font.  Delete the font object.
font.DeleteObject();
```

See Also

Reference

CFont Class
Hierarchy Chart
CFont::CreateFontIndirect
CFont::CreatePointFont
CreateFont
EnumFontFamilies
EnumFonts

Other Resources

CFont Members

CFont::CreateFontIndirect

Initializes a **CFont** object with the characteristics given in a **LOGFONT** structure pointed to by *lpLogFont*.

```
BOOL CreateFontIndirect(
   const LOGFONT* lpLogFont
);
```

Parameters

lpLogFont

Points to a **LOGFONT** structure that defines the characteristics of the logical font.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The font can subsequently be selected as the current font for any device.

This font has the characteristics specified in the LOGFONT structure. When the font is selected by using the CDC::SelectObject member function, the GDI's font mapper attempts to match the logical font with an existing physical font. If it fails to find an exact match for the logical font, it provides an alternative whose characteristics match as many of the requested characteristics as possible.

When you finish with the **CFont** object created by the **CreateFontIndirect** function, first select the font out of the device context, then delete the **CFont** object.

Example

```
// The code fragment shows how to create a font object,
// select the font object into a DC (device context) for text
// drawing, and finally delete the font object.

// Initializes a CFont object with the characteristics given
// in a LOGFONT structure.
CFont font;
LOGFONT lf;
memset(&lf, 0, sizeof(LOGFONT));       // zero out structure
lf.lfHeight = 12;                      // request a 12-pixel-height font
strcpy(lf.lfFaceName, "Arial");        // request a face name "Arial"
VERIFY(font.CreateFontIndirect(&lf));  // create the font

// Do something with the font just created...
CClientDC dc(this);
CFont* def_font = dc.SelectObject(&font);
dc.TextOut(5, 5, "Hello", 5);
dc.SelectObject(def_font);

// Done with the font. Delete the font object.
font.DeleteObject();
```

See Also

Reference

CFont Class
Hierarchy Chart
CFont::CreateFont
CFont::CreatePointFontIndirect
CDC::SelectObject
CGdiObject::DeleteObject
CreateFontIndirect

Other Resources

CFont Members

CFont::CreatePointFont

This function provides a simple way to create a font of a specified typeface and point size.

```
BOOL CreatePointFont(
   int nPointSize,
   LPCTSTR lpszFaceName,
   CDC* pDC = NULL
);
```

Parameters

nPointSize

Requested font height in tenths of a point. (For instance, pass 120 to request a 12-point font.)

lpszFaceName

A **CString** or pointer to a null-terminated string that specifies the typeface name of the font. The length of this string must not exceed 30 characters. The Windows **EnumFontFamilies** function can be used to enumerate all currently available fonts. If *lpszFaceName* is **NULL**, the GDI uses a device-independent typeface.

pDC

Pointer to the CDC object to be used to convert the height in *nPointSize* to logical units. If **NULL**, a screen device context is used for the conversion.

Return Value

Nonzero if successful, otherwise 0.

Remarks

It automatically converts the height in *nPointSize* to logical units using the **CDC** object pointed to by *pDC*.

When you finish with the **CFont** object created by the **CreatePointFont** function, first select the font out of the device context, then delete the **CFont** object.

Example

```
// The code fragment shows how to create a font object,
// select the font object into a DC (device context) for text
// drawing, and finally delete the font object.

CClientDC dc(this);

CFont font;
VERIFY(font.CreatePointFont(120, "Arial", &dc));

// Do something with the font just created...
CFont* def_font = dc.SelectObject(&font);
dc.TextOut(5, 5, "Hello", 5);
dc.SelectObject(def_font);

// Done with the font. Delete the font object.
font.DeleteObject();
```

See Also

Reference

CFont Class
Hierarchy Chart
CFont::CreatePointFontIndirect
CFont::CreateFont

Other Resources

CFont Members

CFont::CreatePointFontIndirect

This function is the same as CreateFontIndirect except that the **lfHeight** member of the **LOGFONT** is interpreted in tenths of a point rather than device units.

```
BOOL CreatePointFontIndirect(
   const LOGFONT* lpLogFont,
   CDC* pDC = NULL
);
```

Parameters

lpLogFont

Points to a LOGFONT structure that defines the characteristics of the logical font. The **lfHeight** member of the **LOGFONT** structure is measured in tenths of a point rather than logical units. (For instance, set **lfHeight** to 120 to request a 12-point font.)

pDC

Pointer to the CDC object to be used to convert the height in **lfHeight** to logical units. If **NULL**, a screen device context is used for the conversion.

Return Value

Nonzero if successful, otherwise 0.

Remarks

This function automatically converts the height in **lfHeight** to logical units using the **CDC** object pointed to by *pDC* before passing the **LOGFONT** structure on to Windows.

When you finish with the **CFont** object created by the **CreatePointFontIndirect** function, first select the font out of the device context, then delete the **CFont** object.

Example

```
// The code fragment shows how to create a font object,
// select the font object into a DC (device context) for text
// drawing, and finally delete the font object.
{
    // ...
    LOGFONT lf;

    // clear out structure.
    memset(&lf, 0, sizeof(LOGFONT));

    // request a 12-pixel-height font
    lf.lfHeight = 120;

    // request a face name "Arial".
    strcpy_s(lf.lfFaceName, LF_FACESIZE, "Arial");

    CClientDC dc(this);

    CFont font;
    VERIFY(font.CreatePointFontIndirect(&lf, &dc));

    // Do something with the font just created...
    CFont* def_font = dc.SelectObject(&font);
    dc.TextOut(5, 5, "Hello", 5);
    dc.SelectObject(def_font);

    // Done with the font. Delete the font object.
    font.DeleteObject();

    // ...
```

```
}
```

See Also

Reference

CFont Class

Hierarchy Chart

CFont::CreatePointFont

CFont::CreateFontIndirect

Other Resources

CFont Members

CFont::FromHandle

Returns a pointer to a **CFont** object when given an **HFONT** handle to a Windows GDI font object.

```
static CFont* PASCAL FromHandle(
   HFONT hFont
);
```

Parameters

hFont

An **HFONT** handle to a Windows font.

Return Value

A pointer to a **CFont** object if successful; otherwise **NULL**.

Remarks

If a **CFont** object is not already attached to the handle, a temporary **CFont** object is created and attached. This temporary **CFont** object is valid only until the next time the application has idle time in its event loop, at which time all temporary graphic objects are deleted. Another way of saying this is that the temporary object is valid only during the processing of one window message.

Example

```
// The code fragment shows how to create a font object using
// Windows API CreateFontIndirect(), convert the HFONT to a
// CFont* before selecting the font object into a DC (device
// context) for text drawing, and finally delete the font object.

// Initialize a CFont object with the characteristics given
// in a LOGFONT structure.
{
    ...
    LOGFONT lf;

    // clear out structure
    memset(&lf, 0, sizeof(LOGFONT));
    // request a 12-pixel-height font
    lf.lfHeight = 12;
    // request a face name "Arial"
    strcpy_s(lf.lfFaceName, LF_FACESIZE, "Arial");
    // create the font
    HFONT hfont = ::CreateFontIndirect(&lf);

    // Convert the HFONT to CFont*.
    CFont* pfont = CFont::FromHandle(hfont);

    // Do something with the font just created...
    CClientDC dc(this);
    CFont* def_font = dc.SelectObject(pfont);
    dc.TextOut(5, 5, "Hello", 5);
    dc.SelectObject(def_font);

    // Done with the font. Delete the font object.
    ::DeleteObject(hfont);
    ...
}
```

See Also

Reference

CFont Class

Hierarchy Chart

Other Resources

CFont Members

CFont::GetLogFont

Call this function to retrieve a copy of the **LOGFONT** structure for **CFont**.

```
int GetLogFont(
   LOGFONT * pLogFont
);
```

Parameters

pLogFont

Pointer to the LOGFONT structure to receive the font information.

Return Value

Nonzero if the function succeeds, otherwise 0.

Example

```
// The code fragment shows how to retrieve a copy of the
// LOGFONT structure for a currently selected font of a window.

CFont* font = pWnd->GetFont();
if (font)
{
   LOGFONT lf;
   font->GetLogFont(&lf);
   TRACE("Typeface name of font = %s\n", lf.lfFaceName);
}
```

See Also

Reference

CFont Class

Hierarchy Chart

LOGFONT

GetObject

Other Resources

CFont Members

CFont::operator HFONT

Use this operator to get the Windows GDI handle of the font attached to the **CFont** object.

```
operator HFONT( ) const;
```

Return Value

The handle of the Windows GDI font object attached to **CFont** if successful; otherwise **NULL**.

Remarks

Since this operator is automatically used for conversions from **CFont** to Fonts and Text, you can pass **CFont** objects to functions that expect **HFONT**s.

For more information about using graphic objects, see Graphic Objects in the *Platform SDK*.

Example

```
// The code fragment shows the usage of CFont::operator HFONT.
{
    // ...

    // Initialize a CFont object with the characteristics given
    // in a LOGFONT structure.
    LOGFONT lf;

    // clear out structure
    memset(&lf, 0, sizeof(LOGFONT));

    // request a 12-pixel-height font
    lf.lfHeight = 12;

    // request a face name "Arial"
    strcpy_s(lf.lfFaceName, LF_FACESIZE, "Arial");

    CFont font1;
    font1.CreateFontIndirect(&lf);  // create the font

    // CFont::operator HFONT automatically converts font1 from
    // CFont* to HFONT.
    CFont* font2 = CFont::FromHandle(font1);

    // Do something with the font just created...
    CClientDC dc(this);
    CFont* def_font = dc.SelectObject(font2);
    dc.TextOut(5, 5, "Hello", 5);
    dc.SelectObject(def_font);

    // Done with the font. Delete the font object.
    font1.DeleteObject();

    // ...
}
```

See Also

Reference

CFont Class

Hierarchy Chart

Other Resources

CFont Members

CFontDialog Class

Allows you to incorporate a font-selection dialog box into your application.

```
class CFontDialog : public CCommonDialog
```

Remarks

A **CFontDialog** object is a dialog box with a list of fonts that are currently installed in the system. The user can select a particular font from the list, and this selection is then reported back to the application.

To construct a **CFontDialog** object, use the provided constructor or derive a new subclass and use your own custom constructor.

Once a **CFontDialog** object has been constructed, you can use the **m_cf** structure to initialize the values or states of controls in the dialog box. The m_cf structure is of type CHOOSEFONT. For more information on this structure, see the *Platform SDK*.

After initializing the dialog object's controls, call the **DoModal** member function to display the dialog box and allow the user to select a font. **DoModal** returns whether the user selected the OK (**IDOK**) or Cancel (**IDCANCEL**) button.

If **DoModal** returns **IDOK**, you can use one of **CFontDialog**'s member functions to retrieve the information input by the user.

You can use the Windows CommDlgExtendedError function to determine whether an error occurred during initialization of the dialog box and to learn more about the error. For more information on this function, see the *Platform SDK*.

CFontDialog relies on the COMMDLG.DLL file that ships with Windows versions 3.1 and later.

To customize the dialog box, derive a class from **CFontDialog**, provide a custom dialog template, and add a message-map to process the notification messages from the extended controls. Any unprocessed messages should be passed to the base class.

Customizing the hook function is not required.

For more information on using **CFontDialog**, see Common Dialog Classes.

Requirements

Header: afxdlgs.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

HIERSVR Sample: Demonstrates a Server Application with OLE Drag and Drop

Reference

CCommonDialog Class

Hierarchy Chart

Other Resources

CFontDialog Members

CFontDialog Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CDialog Members

CCommonDialog Members

Data Members

m_cf	A structure used to customize a **CFontDialog** object.

Construction

CFontDialog	Constructs a **CFontDialog** object.

Operations

DoModal	Displays the dialog and allows the user to make a selection.
GetCharFormat	Retrieves the character formatting of the selected font.
GetColor	Returns the color of the selected font.
GetCurrentFont	Retrieves the name of the currently selected font.
GetFaceName	Returns the face name of the selected font.
GetSize	Returns the point size of the selected font.
GetStyleName	Returns the style name of the selected font.
GetWeight	Returns the weight of the selected font.
IsBold	Determines whether the font is bold.
IsItalic	Determines whether the font is italic.
IsStrikeOut	Determines whether the font is displayed with strikeout.
IsUnderline	Determines whether the font is underlined.

See Also

Reference

CFontDialog Class

Hierarchy Chart

CFontDialog Member Functions

For information about the member functions in **CFontDialog**, see CFontDialog Members.

CFontDialog::CFontDialog

Constructs a **CFontDialog** object.

```
CFontDialog(
   LPLOGFONT lplfInitial = NULL,
   DWORD dwFlags = CF_EFFECTS | CF_SCREENFONTS,
   CDC* pdcPrinter = NULL,
   CWnd* pParentWnd = NULL
);
CFontDialog(
   const CHARFORMAT& charformat,
   DWORD dwFlags = CF_SCREENFONTS,
   CDC* pdcPrinter = NULL,
   CWnd* pParentWnd = NULL
);
```

Parameters

lplfInitial

A pointer to a LOGFONT data structure that allows you to set some of the font's characteristics.

charFormat

A pointer to a CHARFORMAT data structure that allows you to set some of the font's characteristics in a rich edit control.

dwFlags

Specifies one or more choose-font flags. One or more preset values can be combined using the bitwise OR operator. If you modify the **m_cf.Flags** structure member, be sure to use a bitwise OR operator in your changes to keep the default behavior intact. For details on each of these flags, see the description of the CHOOSEFONT structure in the *Platform SDK*.

pdcPrinter

A pointer to a printer-device context. If supplied, this parameter points to a printer-device context for the printer on which the fonts are to be selected.

pParentWnd

A pointer to the font dialog box's parent or owner window.

Remarks

Note that the constructor automatically fills in the members of the **CHOOSEFONT** structure. You should only change these if you want a font dialog different than the default.

Note
The first version of this function only exists when there is no rich edit control support.

Example

```
// Show the font dialog with all the default settings.
CFontDialog dlg;
dlg.DoModal();

// Show the font dialog with 12 point "Times New Roman" as the
// selected font.
LOGFONT lf;
memset(&lf, 0, sizeof(LOGFONT));

CClientDC dc(this);
lf.lfHeight = -MulDiv(12, dc.GetDeviceCaps(LOGPIXELSY), 72);
strcpy(lf.lfFaceName, "Times New Roman");

CFontDialog dlg(&lf);
```

```
dlg.DoModal();
```

See Also

Reference

CFontDialog Class

Hierarchy Chart

CFontDialog::DoModal

Other Resources

CFontDialog Members

CFontDialog::DoModal

Call this function to display the Windows common font dialog box and allow the user to choose a font.

```
virtual INT_PTR DoModal( );
```

Return Value

IDOK or **IDCANCEL**. If **IDCANCEL** is returned, call the Windows CommDlgExtendedError function to determine whether an error occurred.

IDOK and **IDCANCEL** are constants that indicate whether the user selected the OK or Cancel button.

Remarks

If you want to initialize the various font dialog controls by setting members of the m_cf structure, you should do this before calling **DoModal**, but after the dialog object is constructed.

If **DoModal** returns **IDOK**, you can call other member functions to retrieve the settings or information input by the user into the dialog box.

Example

See the examples for CFontDialog::CFontDialog and CFontDialog::GetColor.

See Also

Reference

CFontDialog Class
Hierarchy Chart
CDialog::DoModal
CFontDialog::CFontDialog

Other Resources

CFontDialog Members

CFontDialog::GetCharFormat

Retrieves the character formatting of the selected font.

```
void GetCharFormat(
   CHARFORMAT& cf
) const;
```

Parameters

cf

A CHARFORMAT structure containing information about the character formatting of the selected font.

See Also

Reference

CFontDialog Class

Hierarchy Chart

Other Resources

CFontDialog Members

CFontDialog::GetColor

Call this function to retrieve the selected font color.

```
COLORREF GetColor( ) const;
```

Return Value

The color of the selected font.

Example

```
// Get the color of the selected font, if any.
CFontDialog dlg;
if (dlg.DoModal() == IDOK)
{
   COLORREF color = dlg.GetColor();
   TRACE("Color of the selected font = %8x\n", color);
}
```

See Also

Reference

CFontDialog Class
Hierarchy Chart
CFontDialog::GetCurrentFont

Other Resources

CFontDialog Members

CFontDialog::GetCurrentFont

Call this function to assign the characteristics of the currently selected font to the members of a LOGFONT structure.

```
void GetCurrentFont(
   LPLOGFONT lplf
);
```

Parameters

lplf

A pointer to a **LOGFONT** structure.

Remarks

Other **CFontDialog** member functions are provided to access individual characteristics of the current font.

If this function is called during a call to DoModal, it returns the current selection at the time (what the user sees or has changed in the dialog). If this function is called after a call to **DoModal** (only if **DoModal** returns **IDOK**), it returns what the user actually selected.

Example

```
// Get the characteristics of the currently selected font, if any.
CFontDialog dlg;
if (dlg.DoModal() == IDOK)
{
   LOGFONT lf;
   dlg.GetCurrentFont(&lf);
   TRACE("Face name of the selected font = %s\n", lf.lfFaceName);
}
```

See Also

Reference

CFontDialog Class
Hierarchy Chart
CFontDialog::GetFaceName
CFontDialog::GetStyleName

Other Resources

CFontDialog Members

CFontDialog::GetFaceName

Call this function to retrieve the face name of the selected font.

```
CString GetFaceName( ) const;
```

Return Value

The face name of the font selected in the **CFontDialog** dialog box.

Example

```
// Get the face name of the selected font, if any.
CFontDialog dlg
if (dlg.DoModal() == IDOK)
{
   CString facename = dlg.GetFaceName();
   TRACE("Face name of the selected font = %s\n", facename);
}
```

See Also

Reference

CFontDialog Class
Hierarchy Chart
CFontDialog::GetCurrentFont
CFontDialog::GetStyleName

Other Resources

CFontDialog Members

CFontDialog::GetSize

Call this function to retrieve the size of the selected font.

```
int GetSize( ) const;
```

Return Value

The font's size, in tenths of a point.

Example

```
// Get the size of the selected font, if any.
CFontDialog dlg;
if (dlg.DoModal() == IDOK)
{
   int size = dlg.GetSize();
   TRACE("The size of the selected font = %d\n", size);
}
```

See Also

Reference

CFontDialog Class
Hierarchy Chart
CFontDialog::GetWeight
CFontDialog::GetCurrentFont

Other Resources

CFontDialog Members

CFontDialog::GetStyleName

Call this function to retrieve the style name of the selected font.

```
CString GetStyleName( ) const;
```

Return Value

The style name of the font.

Example

```
// Get the style name of the selected font, if any.
CFontDialog dlg;
if (dlg.DoModal() == IDOK)
{
   CString stylename = dlg.GetStyleName();
   TRACE("Style name of the selected font = %s\n", stylename);
}
```

See Also

Reference

CFontDialog Class
Hierarchy Chart
CFontDialog::GetFaceName
CFontDialog::GetCurrentFont

Other Resources

CFontDialog Members

CFontDialog::GetWeight

Call this function to retrieve the weight of the selected font.

```
int GetWeight( ) const;
```

Return Value

The weight of the selected font.

Remarks

For more information on the weight of a font, see CFont::CreateFont.

Example

```
// Get the weight of the selected font, if any.
CFontDialog dlg;
if (dlg.DoModal() == IDOK)
{
   int weight = dlg.GetWeight();
   TRACE("Weight of the selected font = %d\n", weight);
}
```

See Also

Reference

CFontDialog Class
Hierarchy Chart
CFontDialog::GetCurrentFont
CFontDialog::IsBold

Other Resources

CFontDialog Members

CFontDialog::IsBold

Call this function to determine if the selected font is bold.

```
BOOL IsBold( ) const;
```

Return Value

Nonzero if the selected font has the Bold characteristic enabled; otherwise 0.

Example

```
// Is the selected font bold?
CFontDialog dlg;
if (dlg.DoModal() == IDOK)
{
   BOOL bold = dlg.IsBold();
   TRACE("Is the selected font bold? %d\n", bold);
}
```

See Also

Reference

CFontDialog Class
Hierarchy Chart
CFontDialog::GetCurrentFont

Other Resources

CFontDialog Members

CFontDialog::IsItalic

Call this function to determine if the selected font is italic.

```
BOOL IsItalic( ) const;
```

Return Value

Nonzero if the selected font has the Italic characteristic enabled; otherwise 0.

Example

```
// Is the selected font italic?
CFontDialog dlg;
if (dlg.DoModal() == IDOK)
{
   BOOL italic = dlg.IsItalic();
   TRACE("Is the selected font italic? %d\n", italic);
}
```

See Also

Reference

CFontDialog Class
Hierarchy Chart
CFontDialog::GetCurrentFont

Other Resources

CFontDialog Members

CFontDialog::IsStrikeOut

Call this function to determine if the selected font is displayed with strikeout.

```
BOOL IsStrikeOut( ) const;
```

Return Value

Nonzero if the selected font has the Strikeout characteristic enabled; otherwise 0.

Example

```
// Is the selected font displayed with strikeout?
CFontDialog dlg;
if (dlg.DoModal() == IDOK)
{
   BOOL strikeout = dlg.IsStrikeOut();
   TRACE("Is the selected font displayed with strikeout? %d\n", strikeout);
}
```

See Also

Reference

CFontDialog Class
Hierarchy Chart
CFontDialog::GetCurrentFont

Other Resources

CFontDialog Members

CFontDialog::IsUnderline

Call this function to determine if the selected font is underlined.

```
BOOL IsUnderline( ) const;
```

Return Value

Nonzero if the selected font has the Underline characteristic enabled; otherwise 0.

Example

```
// Is the selected font underlined?
CFontDialog dlg;
if (dlg.DoModal() == IDOK)
{
   BOOL underline = dlg.IsUnderline();
   TRACE("Is the selected font underlined? %d\n", underline);
}
```

See Also

Reference

CFontDialog Class
Hierarchy Chart
CFontDialog::GetCurrentFont

Other Resources

CFontDialog Members

CFontDialog Data Members

For information about the data members in **CFontDialog**, see CFontDialog Members.

CFontDialog::m_cf

A structure whose members store the characteristics of the dialog object.

```
CHOOSEFONT m_cf;
```

Remarks

After constructing a **CFontDialog** object, you can use **m_cf** to modify various aspects of the dialog box before calling the **DoModal** member function. For more information on this structure, see CHOOSEFONT in the *Platform SDK*.

Example

```
// The code fragment creates a font based on the information
// we got from CFontDialog::m_cf variable.

CFontDialog dlg;
if (dlg.DoModal() == IDOK)
{
   // Create the font using the selected font from CFontDialog.
   LOGFONT lf;
   memcpy(&lf, dlg.m_cf.lpLogFont, sizeof(LOGFONT));

   CFont font;
   VERIFY(font.CreateFontIndirect(&lf));

   // Do something with the font just created...
   CClientDC dc(this);
   CFont* def_font = dc.SelectObject(&font);
   dc.TextOut(5, 5, "Hello", 5);
   dc.SelectObject(def_font);

   // Done with the font. Delete the font object.
   font.DeleteObject();
}
```

See Also

Reference

CFontDialog Class

Hierarchy Chart

Other Resources

CFontDialog Members

CFontHolder Class

Implements the stock Font property and encapsulates the functionality of a Windows font object and the **IFont** interface.

```
class CFontHolder
```

Remarks

CFontHolder does not have a base class.

Use this class to implement custom font properties for your control. For information on creating such properties, see the article ActiveX Controls: Using Fonts.

Requirements

Header: afxctl.h

See Also

Reference

Hierarchy Chart

CPropExchange Class

Other Resources

CFontHolder Members

CFontHolder Members

Data Members

m_pFont	A pointer to the **CFontHolder** object's **IFont** interface.

Construction/Destruction

CFontHolder	Constructs a **CFontHolder** object.

Operations

GetDisplayString	Retrieves the string displayed in a container's property browser.
GetFontDispatch	Returns the font's **IDispatch** interface.
GetFontHandle	Returns a handle to a Windows font.
InitializeFont	Initializes a **CFontHolder** object.
QueryTextMetrics	Retrieves information for the related font.
ReleaseFont	Disconnects the **CFontHolder** object from the **IFont** and **IFontNotification** interfaces.
Select	Selects a font resource into a device context.
SetFont	Connects the **CFontHolder** object to an **IFont** interface.

See Also

Reference

CFontHolder Class

Hierarchy Chart

CFontHolder Member Functions

For information about the member functions in **CFontHolder**, see CFontHolder Members.

CFontHolder::CFontHolder

Constructs a **CFontHolder** object.

```
explicit CFontHolder(
   LPPROPERTYNOTIFYSINK pNotify
);
```

Parameters

pNotify

Pointer to the font's **IPropertyNotifySink** interface.

Remarks

You must call **InitializeFont** to initialize the resulting object before using it.

See Also

Reference

CFontHolder Class

Hierarchy Chart

CFontHolder::InitializeFont

Other Resources

CFontHolder Members

CFontHolder::GetDisplayString

Retrieves a string that can be displayed in a container's property browser.

```
BOOL GetDisplayString(
   CString& strValue
);
```

Parameters

strValue

Reference to the CString that is to hold the display string.

Return Value

Nonzero if the string is successfully retrieved; otherwise 0.

See Also

Reference

CFontHolder Class

Hierarchy Chart

Other Resources

CFontHolder Members

CFontHolder::GetFontDispatch

Call this function to retrieve a pointer to the font's dispatch interface.

```
LPFONTDISP GetFontDispatch( );
```

Return Value

A pointer to the **CFontHolder** object's **IFontDisp** interface. Note that the function that calls **GetFontDispatch** must call **IUnknown::Release** on this interface pointer when done with it.

Remarks

Call **InitializeFont** before calling **GetFontDispatch**.

See Also

Reference

CFontHolder Class
Hierarchy Chart
CFontHolder::InitializeFont

Other Resources

CFontHolder Members

CFontHolder::GetFontHandle

Call this function to get a handle to a Windows font.

```
HFONT GetFontHandle( );
HFONT GetFontHandle(
   long cyLogical,
   long cyHimetric
);
```

Parameters

cyLogical

Height, in logical units, of the rectangle in which the control is drawn.

cyHimetric

Height, in **MM_HIMETRIC** units, of the control.

Return Value

A handle to the Font object; otherwise **NULL**.

Remarks

The ratio of *cyLogical* and *cyHimetric* is used to calculate the proper display size, in logical units, for the font's point size expressed in **MM_HIMETRIC** units:

Display size = (*cyLogical* / *cyHimetric*) X font size

The version with no parameters returns a handle to a font sized correctly for the screen.

See Also

Reference

CFontHolder Class

Hierarchy Chart

Other Resources

CFontHolder Members

CFontHolder::InitializeFont

Initializes a **CFontHolder** object.

```
void InitializeFont(
   const FONTDESC* pFontDesc = NULL,
   LPDISPATCH pFontDispAmbient = NULL
);
```

Parameters

pFontDesc

Pointer to a font description structure (FONTDESC) that specifies the font's characteristics.

pFontDispAmbient

Pointer to the container's ambient Font property.

Remarks

If *pFontDispAmbient* is not **NULL**, the **CFontHolder** object is connected to a clone of the **IFont** interface used by the container's ambient Font property.

If *pFontDispAmbient* is **NULL**, a new Font object is created either from the font description pointed to by *pFontDesc* or, if *pFontDesc* is **NULL**, from a default description.

Call this function after constructing a **CFontHolder** object.

See Also

Reference

CFontHolder Class
Hierarchy Chart
CFontHolder::CFontHolder

Other Resources

CFontHolder Members

CFontHolder::QueryTextMetrics

Retrieves information on the physical font represented by the **CFontHolder** object.

```
void QueryTextMetrics(
   LPTEXTMETRIC lptm
);
```

Parameters

lptm

A pointer to a TEXTMETRIC structure that will receive the information.

See Also

Reference

CFontHolder Class

Hierarchy Chart

Other Resources

CFontHolder Members

CFontHolder::ReleaseFont

This function disconnects the **CFontHolder** object from its **IFont** interface.

```
void ReleaseFont( );
```

See Also

Reference

CFontHolder Class
Hierarchy Chart
CFontHolder::SetFont

Other Resources

CFontHolder Members

CFontHolder::Select

Call this function to select your control's font into the specified device context.

```
CFont* Select(
   CDC* pDC,
   long cyLogical,
   long cyHimetric
);
```

Parameters

pDC

Device context into which the font is selected.

cyLogical

Height, in logical units, of the rectangle in which the control is drawn.

cyHimetric

Height, in **MM_HIMETRIC** units, of the control.

Return Value

A pointer to the font that is being replaced.

Remarks

See GetFontHandle for a discussion of the *cyLogical* and *cyHimetric* parameters.

See Also

Reference

CFontHolder Class

Hierarchy Chart

Other Resources

CFontHolder Members

CFontHolder::SetFont

Releases any existing font and connects the **CFontHolder** object to an **IFont** interface.

```
void SetFont(
   LPFONT pNewFont
);
```

Parameters

pNewFont

Pointer to the new **IFont** interface.

See Also

Reference

CFontHolder Class

Hierarchy Chart

CFontHolder::ReleaseFont

Other Resources

CFontHolder Members

CFontHolder Data Members

For information about the data members in **CFontHolder**, see CFontHolder Members.

CFontHolder::m_pFont

A pointer to the **CFontHolder** object's **IFont** interface.

```
LPFONT m_pFont;
```

See Also

Reference

CFontHolder Class

Hierarchy Chart

CFontHolder::SetFont

Other Resources

CFontHolder Members

CFormView Class

The base class used for form views.

```
class CFormView : public CScrollView
```

Remarks

A form view is essentially a view that contains controls. These controls are laid out based on a dialog-template resource. Use **CFormView** if you want forms in your application. These views support scrolling, as needed, using the CScrollView functionality.

When you are Creating a Forms-Based Application, you can base its view class on **CFormView**, making it a forms-based application.

You can also insert new Form Topics into document-view-based applications. Even if your application did not initially support forms, Visual C++ will add this support when you insert a new form.

The MFC Application Wizard and the Add Class command are the preferred methods for creating forms-based applications. If you need to create a forms-based application without using these methods, see Creating a Forms-Based Application.

Requirements

Header: afxext.h

See Also

Tasks

SNAPVW Sample: Uses Property Pages in a Form View Application
VIEWEX Sample: Demonstrates Multiple Views, Scroll Views, and Splitter Windows

Reference

CScrollView Class
Hierarchy Chart
CDialog Class
CScrollView Class
CView::OnUpdate
CView::OnInitialUpdate
CView::OnPrint
CWnd::UpdateData
CScrollView::ResizeParentToFit

Other Resources

CFormView Members

CFormView Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CView Members

CScrollView Members

Construction

CFormView	Constructs a **CFormView** object.

See Also

Reference

CFormView Class

Hierarchy Chart

CFormView Member Functions

For information about the member functions in **CFormView**, see CFormView Members.

CFormView::CFormView

Constructs a **CFormView** object.

```
CFormView(
   LPCTSTR lpszTemplateName
);
CFormView(
   UINT nIDTemplate
);
```

Parameters

lpszTemplateName

Contains a null-terminated string that is the name of a dialog-template resource.

nIDTemplate

Contains the ID number of a dialog-template resource.

Remarks

When you create an object of a type derived from **CFormView**, invoke one of the constructors to create the view object and identify the dialog resource on which the view is based. You can identify the resource either by name (pass a string as the argument to the constructor) or by its ID (pass an unsigned integer as the argument).

The form-view window and child controls are not created until **CWnd::Create** is called. **CWnd::Create** is called by the framework as part of the document and view creation process, which is driven by the document template.

Note

Your derived class *must* supply its own constructor. In the constructor, invoke the constructor, **CFormView::CFormView**, wit h the resource name or ID as an argument as shown in the preceding class overview.

Example

```
// MyView.h.

/////////////////////////////////////////////////////
// CMyView form view.

class CMyView : public CFormView
{
   DECLARE_DYNCREATE(CMyView)

// Protected constructor used by dynamic creation.
protected:
   CMyView();

// Form Data.
public:
   //{{AFX_DATA(CMyView)
   enum { IDD = IDD_MYFORM };
   //}}AFX_DATA

   // Other members and methods ...

   // Generated message map functions.
   //{{AFX_MSG(CMyView)
   //}}AFX_MSG
   DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////
```

```
// MyView.cpp.

#include "MyView.h"

/////////////////////////////////////////////////////
// CMyView

IMPLEMENT_DYNCREATE(CMyView, CFormView)

CMyView::CMyView()
   : CFormView(CMyView::IDD)
{
   //{{AFX_DATA_INIT(CMyView)
   //}}AFX_DATA_INIT
}
/////////////////////////////////////////////////////

// Other methods and handlers ...
```

See Also

Reference

CFormView Class

Hierarchy Chart

CWnd::Create

Other Resources

CFormView Members

CFrameWnd Class

Provides the functionality of a Windows single document interface (SDI) overlapped or pop-up frame window, along with members for managing the window.

```
class CFrameWnd : public CWnd
```

Remarks

To create a useful frame window for your application, derive a class from **CFrameWnd**. Add member variables to the derived class to store data specific to your application. Implement message-handler member functions and a message map in the derived class to specify what happens when messages are directed to the window.

There are three ways to construct a frame window:

- Directly construct it using Create.
- Directly construct it using LoadFrame.
- Indirectly construct it using a document template.

Before you call either **Create** or **LoadFrame**, you must construct the frame-window object on the heap using the C++ **new** operator. Before calling **Create**, you can also register a window class with the AfxRegisterWndClass global function to set the icon and class styles for the frame.

Use the **Create** member function to pass the frame's creation parameters as immediate arguments.

LoadFrame requires fewer arguments than **Create**, and instead retrieves most of its default values from resources, including the frame's caption, icon, accelerator table, and menu. To be accessible by **LoadFrame**, all these resources must have the same resource ID (for example, **IDR_MAINFRAME**).

When a **CFrameWnd** object contains views and documents, they are created indirectly by the framework instead of directly by the programmer. The **CDocTemplate** object orchestrates the creation of the frame, the creation of the containing views, and the connection of the views to the appropriate document. The parameters of the **CDocTemplate** constructor specify the **CRuntimeClass** of the three classes involved (document, frame, and view). A **CRuntimeClass** object is used by the framework to dynamically create new frames when specified by the user (for example, by using the File New command or the multiple document interface (MDI) Window New command).

A frame-window class derived from **CFrameWnd** must be declared with **DECLARE_DYNCREATE** in order for the above **RUNTIME_CLASS** mechanism to work correctly.

A **CFrameWnd** contains default implementations to perform the following functions of a main window in a typical application for Windows:

- A **CFrameWnd** frame window keeps track of a currently active view that is independent of the Windows active window or the current input focus. When the frame is reactivated, the active view is notified by calling **CView::OnActivateView**.
- Command messages and many common frame-notification messages, including those handled by the **OnSetFocus**, **OnHScroll**, and **OnVScroll** functions of **CWnd**, are delegated by a **CFrameWnd** frame window to the currently active view.
- The currently active view (or currently active MDI child frame window in the case of an MDI frame) can determine the caption of the frame window. This feature can be disabled by turning off the **FWS_ADDTOTITLE** style bit of the frame window.
- A **CFrameWnd** frame window manages the positioning of the control bars, views, and other child windows inside the frame window's client area. A frame window also does idle-time updating of toolbar and other control-bar buttons. A **CFrameWnd** frame window also has default implementations of commands for toggling on and off the toolbar and status bar.
- A **CFrameWnd** frame window manages the main menu bar. When a pop-up menu is displayed, the frame window uses the **UPDATE_COMMAND_UI** mechanism to determine which menu items should be enabled, disabled, or checked. When the user selects a menu item, the frame window updates the status bar with the message string for that command.

- A **CFrameWnd** frame window has an optional accelerator table that automatically translates keyboard accelerators.
- A **CFrameWnd** frame window has an optional help ID set with **LoadFrame** that is used for context-sensitive help. A frame window is the main orchestrator of semimodal states such as context-sensitive help (SHIFT+F1) and print-preview modes.
- A **CFrameWnd** frame window will open a file dragged from the File Manager and dropped on the frame window. If a file extension is registered and associated with the application, the frame window responds to the dynamic data exchange (DDE) open request that occurs when the user opens a data file in the File Manager or when the **ShellExecute** Windows function is called.
- If the frame window is the main application window (that is, **CWinThread::m_pMainWnd**), when the user closes the application, the frame window prompts the user to save any modified documents (for **OnClose** and **OnQueryEndSession**).
- If the frame window is the main application window, the frame window is the context for running WinHelp. Closing the frame window will shut down WINHELP.EXE if it was launched for help for this application.

Do not use the C++ **delete** operator to destroy a frame window. Use **CWnd::DestroyWindow** instead. The **CFrameWnd** implementation of **PostNcDestroy** will delete the C++ object when the window is destroyed. When the user closes the frame window, the default **OnClose** handler will call **DestroyWindow**.

For more information on **CFrameWnd**, see Frame Windows.

Requirements

Header: afxwin.h

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd Class
CMDIFrameWnd Class
CMDIChildWnd Class
CView Class
CDocTemplate Class
CRuntimeClass Structure

Other Resources

CFrameWnd Members

CFrameWnd Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Data Members

m_bAutoMenuEnable	Controls automatic enable and disable functionality for menu items.
rectDefault	Pass this static **CRect** as a parameter when creating a **CFrameWnd** object to allow Windows to choose the window's initial size and position.

Construction

CFrameWnd	Constructs a **CFrameWnd** object.

Initialization

Create	Call to create and initialize the Windows frame window associated with the **CFrameWnd** object.
GetDockState	Retrieves the dock state of a frame window.
LoadAccelTable	Call to load an accelerator table.
LoadBarState	Call to restore control bar settings.
LoadFrame	Call to dynamically create a frame window from resource information.
SaveBarState	Call to save control bar settings.
SetDockState	Call to dock the frame window in the main window.
ShowControlBar	Call to show the control bar.

Operations

ActivateFrame	Makes the frame visible and available to the user.
BeginModalState	Sets the frame window to modal.
CreateView	Creates a view within a frame that is not derived from **CView**.
DockControlBar	Docks a control bar.
EnableDocking	Allows a control bar to be docked.
EndModalState	Ends the frame window's modal state. Enables all of the windows disabled by **BeginModalState**.
FloatControlBar	Floats a control bar.
GetActiveDocument	Returns the active **CDocument** object.
GetActiveFrame	Returns the active **CFrameWnd** object.

GetActiveView	Returns the active **CView** object.
GetControlBar	Retrieves the control bar.
GetMessageString	Retrieves message corresponding to a command ID.
GetTitle	Retrieves the title of the related control bar.
InitialUpdateFrame	Causes the **OnInitialUpdate** member function belonging to all views in the frame window to be called.
InModalState	Returns a value indicating whether or not a frame window is in a modal state.
IsTracking	Determines if splitter bar is currently being moved.
RecalcLayout	Repositions the control bars of the **CFrameWnd** object.
SetActiveView	Sets the active **CView** object.
SetMessageText	Sets the text of a standard status bar.
SetTitle	Sets the title of the related control bar.
ShowOwnedWindows	Shows all windows that are descendants of the **CFrameWnd** object.

Overridables

GetMessageBar	Returns a pointer to the status bar belonging to the frame window.
NegotiateBorderSpace	Negotiates border space in the frame window.
OnCreateClient	Creates a client window for the frame.
OnSetPreviewMode	Sets the application's main frame window into and out of print-preview mode.

Command Handlers

OnBarCheck	Called whenever an action is performed on the specified control bar.
OnContextHelp	Handles SHIFT+F1 Help for in-place items.
OnUpdateControlBarMenu	Called by the framework when the associated menu is updated.

See Also

Reference

CFrameWnd Class
Hierarchy Chart

CFrameWnd Member Functions

For information about the member functions in **CFrameWnd**, see CFrameWnd Members.

CFrameWnd::ActivateFrame

Call this member function to activate and restore the frame window so that it is visible and available to the user.

```
virtual void ActivateFrame(
   int nCmdShow = -1
);
```

Parameters

nCmdShow

Specifies the parameter to pass to CWnd::ShowWindow. By default, the frame is shown and correctly restored.

Remarks

This member function is usually called after a non-user interface event such as a DDE, OLE, or other event that may show the frame window or its contents to the user.

The default implementation activates the frame and brings it to the top of the Z-order and, if necessary, carries out the same steps for the application's main frame window.

Override this member function to change how a frame is activated. For example, you can force MDI child windows to be maximized. Add the appropriate functionality, then call the base class version with an explicit *nCmdShow*.

Example

```
void CChildFrame::ActivateFrame(int nCmdShow)
{
   // Create the child frame window maximized
   nCmdShow = SW_MAXIMIZE;

   CMDIChildWnd::ActivateFrame(nCmdShow);
}
```

See Also

Reference

CFrameWnd Class

Hierarchy Chart

Other Resources

CFrameWnd Members

CFrameWnd::BeginModalState

Call this member function to make a frame window modal.

```
virtual void BeginModalState( );
```

See Also

Reference

CFrameWnd Class

Hierarchy Chart

Other Resources

CFrameWnd Members

CFrameWnd::CFrameWnd

Constructs a **CFrameWnd** object, but does not create the visible frame window.

```
CFrameWnd( );
```

Remarks

Call **Create** to create the visible window.

See Also

Reference

CFrameWnd Class
Hierarchy Chart
CFrameWnd::Create
CFrameWnd::LoadFrame

Other Resources

CFrameWnd Members

CFrameWnd::Create

Call to create and initialize the Windows frame window associated with the **CFrameWnd** object.

```
virtual BOOL Create(
   LPCTSTR lpszClassName,
   LPCTSTR lpszWindowName,
   DWORD dwStyle = WS_OVERLAPPEDWINDOW,
   const RECT& rect = rectDefault,
   CWnd* pParentWnd = NULL,
   LPCTSTR lpszMenuName = NULL,
   DWORD dwExStyle = 0,
   CCreateContext* pContext = NULL
);
```

Parameters

lpszClassName

Points to a null-terminated character string that names the Windows class. The class name can be any name registered with the **AfxRegisterWndClass** global function or the **RegisterClass** Windows function. If **NULL**, uses the predefined default **CFrameWnd** attributes.

lpszWindowName

Points to a null-terminated character string that represents the window name. Used as text for the title bar.

dwStyle

Specifies the window style attributes. Include the **FWS_ADDTOTITLE** style if you want the title bar to automatically display the name of the document represented in the window.

rect

Specifies the size and position of the window. The **rectDefault** value allows Windows to specify the size and position of the new window.

pParentWnd

Specifies the parent window of this frame window. This parameter should be **NULL** for top-level frame windows.

lpszMenuName

Identifies the name of the menu resource to be used with the window. Use **MAKEINTRESOURCE** if the menu has an integer ID instead of a string. This parameter can be **NULL**.

dwExStyle

Specifies the window extended style attributes.

pContext

Specifies a pointer to a CCreateContext structure. This parameter can be **NULL**.

Return Value

Nonzero if initialization is successful; otherwise 0.

Remarks

Construct a **CFrameWnd** object in two steps. First, invoke the constructor, which constructs the **CFrameWnd** object, and then call **Create**, which creates the Windows frame window and attaches it to the **CFrameWnd** object. **Create** initializes the window's class name and window name and registers default values for its style, parent, and associated menu.

Use **LoadFrame** rather than **Create** to load the frame window from a resource instead of specifying its arguments.

See Also

Reference

CFrameWnd Class

Hierarchy Chart
CFrameWnd::CFrameWnd
CFrameWnd::LoadFrame
CCreateContext Structure
CWnd::Create
CWnd::PreCreateWindow
Other Resources
CFrameWnd Members

CFrameWnd::CreateView

Call **CreateView** to create a view within a frame.

```
CWnd* CreateView(
   CCreateContext* pContext,
   UINT nID = AFX_IDW_PANE_FIRST
);
```

Parameters

pContext

Specifies the type of view and document.

nID

The ID number of a view.

Return Value

Pointer to a **CWnd** object if successful; otherwise **NULL**.

Remarks

Use this member function to create "views" that are not **CView**-derived within a frame. After calling **CreateView**, you must manually set the view to active and set it to be visible; these tasks are not automatically performed by **CreateView**.

Note
The MFC Advanced Concepts sample COLLECT uses **CreateView** to get correct 3D effects in Windows 95/98.

See Also

Reference

CFrameWnd Class

Hierarchy Chart

Other Resources

CFrameWnd Members

CFrameWnd::DockControlBar

Causes a control bar to be docked to the frame window.

```
void DockControlBar(
   CControlBar* pBar,
   UINT nDockBarID = 0,
   LPCRECT lpRect = NULL
);
```

Parameters

pBar

Points to the control bar to be docked.

nDockBarID

Determines which sides of the frame window to consider for docking. It can be 0, or one or more of the following:

- **AFX_IDW_DOCKBAR_TOP** Dock to the top side of the frame window.
- **AFX_IDW_DOCKBAR_BOTTOM** Dock to the bottom side of the frame window.
- **AFX_IDW_DOCKBAR_LEFT** Dock to the left side of the frame window.
- **AFX_IDW_DOCKBAR_RIGHT** Dock to the right side of the frame window.

If 0, the control bar can be docked to any side enabled for docking in the destination frame window.

lpRect

Determines, in screen coordinates, where the control bar will be docked in the nonclient area of the destination frame window.

Remarks

The control bar will be docked to one of the sides of the frame window specified in the calls to both CControlBar::EnableDocking and CFrameWnd::EnableDocking. The side chosen is determined by *nDockBarID*.

See Also

Reference

CFrameWnd Class
Hierarchy Chart
CFrameWnd::FloatControlBar

Other Resources

CFrameWnd Members

CFrameWnd::EnableDocking

Call this function to enable dockable control bars in a frame window.

```
void EnableDocking(
   DWORD dwDockStyle
);
```

Parameters

dwDockStyle

Specifies which sides of the frame window can serve as docking sites for control bars. It can be one or more of the following:

- **CBRS_ALIGN_TOP** Allows docking at the top of the client area.
- **CBRS_ALIGN_BOTTOM** Allows docking at the bottom of the client area.
- **CBRS_ALIGN_LEFT** Allows docking on the left side of the client area.
- **CBRS_ALIGN_RIGHT** Allows docking on the right side of the client area.
- **CBRS_ALIGN_ANY** Allows docking on any side of the client area.

Remarks

By default, control bars will be docked to a side of the frame window in the following order: top, bottom, left, right.

Example

See the example for CToolBar::Create.

See Also

Reference

CFrameWnd Class
Hierarchy Chart
CControlBar::EnableDocking
CFrameWnd::DockControlBar
CFrameWnd::FloatControlBar

Other Resources

CFrameWnd Members

CFrameWnd::EndModalState

Call this member function to change a frame window from modal to modeless.

```
virtual void EndModalState( );
```

Remarks

EndModalState enables all of the windows disabled by BeginModalState.

See Also

Reference

CFrameWnd Class

Hierarchy Chart

Other Resources

CFrameWnd Members

CFrameWnd::FloatControlBar

Call this function to cause a control bar to not be docked to the frame window.

```
void FloatControlBar(
   CControlBar * pBar,
   CPoint point,
   DWORD dwStyle = CBRS_ALIGN_TOP
);
```

Parameters

pBar

Points to the control bar to be floated.

point

The location, in screen coordinates, where the top left corner of the control bar will be placed.

dwStyle

Specifies whether to align the control bar horizontally or vertically within its new frame window. It can be any one of the following:

- **CBRS_ALIGN_TOP** Orients the control bar vertically.
- **CBRS_ALIGN_BOTTOM** Orients the control bar vertically.
- **CBRS_ALIGN_LEFT** Orients the control bar horizontally.
- **CBRS_ALIGN_RIGHT** Orients the control bar horizontally.

If styles are passed specifying both horizontal and vertical orientation, the toolbar will be oriented horizontally.

Remarks

Typically, this is done at application startup when the program is restoring settings from the previous execution.

This function is called by the framework when the user causes a drop operation by releasing the left mouse button while dragging the control bar over a location that is not available for docking.

See Also

Reference

CFrameWnd Class
Hierarchy Chart
CFrameWnd::DockControlBar

Other Resources

CFrameWnd Members

CFrameWnd::GetActiveDocument

Call this member function to obtain a pointer to the current **CDocument** attached to the current active view.

```
virtual CDocument* GetActiveDocument( );
```

Return Value

A pointer to the current CDocument. If there is no current document, returns **NULL**.

See Also

Reference

CFrameWnd Class
Hierarchy Chart
CFrameWnd::GetActiveView

Other Resources

CFrameWnd Members

CFrameWnd::GetActiveFrame

Call this member function to obtain a pointer to the active multiple document interface (MDI) child window of an MDI frame window.

```
virtual CFrameWnd* GetActiveFrame( );
```

Return Value

A pointer to the active MDI child window. If the application is an SDI application, or the MDI frame window has no active document, the implicit **this** pointer will be returned.

Remarks

If there is no active MDI child or the application is a single document interface (SDI), the implicit **this** pointer is returned.

See Also

Reference

CFrameWnd Class
Hierarchy Chart
CFrameWnd::GetActiveView
CFrameWnd::GetActiveDocument
CMDIFrameWnd Class

Other Resources

CFrameWnd Members

CFrameWnd::GetActiveView

Call this member function to obtain a pointer to the active view (if any) attached to a frame window (**CFrameWnd**).

```
CView* GetActiveView( ) const;
```

Return Value

A pointer to the current CView. If there is no current view, returns **NULL**.

Remarks

This function returns **NULL** when called for an MDI main frame window (**CMDIFrameWnd**). In an MDI application, the MDI main frame window does not have a view associated with it. Instead, each individual child window (**CMDIChildWnd**) has one or more associated views. The active view in an MDI application can be obtained by first finding the active MDI child window and then finding the active view for that child window. The active MDI child window can be found by calling the function **MDIGetActive** or **GetActiveFrame** as demonstrated in the following:

```
CMDIFrameWnd *pFrame =
             (CMDIFrameWnd*)AfxGetApp()->m_pMainWnd;

// Get the active MDI child window.
CMDIChildWnd *pChild =
             (CMDIChildWnd *) pFrame->GetActiveFrame();

// or CMDIChildWnd *pChild = pFrame->MDIGetActive();

// Get the active view attached to the active MDI child
// window.
CMyView *pView = (CMyView *) pChild->GetActiveView();
```

See Also

Reference

CFrameWnd Class
Hierarchy Chart
CFrameWnd::SetActiveView
CFrameWnd::GetActiveDocument

Other Resources

CFrameWnd Members

CFrameWnd::GetControlBar

Call **GetControlBar** to gain access to the control bar that is associated with the ID.

```
CControlBar* GetControlBar(
   UINT nID
);
```

Parameters

nID

The ID number of a control bar.

Return Value

A pointer to the control bar that is associated with the ID.

Remarks

The *nID* parameter refers to the unique identifier passed to the **Create** method of the control bar. For more information on control bars, refer to the topic entitled Control Bars.

GetControlBar will return the control bar even if it is floating and thus is not currently a child window of the frame.

See Also

Reference

CFrameWnd Class

Hierarchy Chart

Other Resources

CFrameWnd Members

CFrameWnd::GetDockState

Call this member function to store state information about the frame window's control bars in a **CDockState** object.

```
void GetDockState(
   CDockState& state
) const;
```

Parameters

state

Contains the current state of the frame window's control bars upon return.

Remarks

You can then write the contents of **CDockState** to storage using **CDockState::SaveState** or **Serialize**. If you later want to restore the control bars to a previous state, load the state with **CDockState::LoadState** or **Serialize**, then call **SetDockState** to apply the previous state to the frame window's control bars.

See Also

Reference

CFrameWnd Class
Hierarchy Chart
CFrameWnd::SetDockState
CDockState Class
CDockState::SaveState
CObject::Serialize

Other Resources

CFrameWnd Members

CFrameWnd::GetMessageBar

Call this member function to get a pointer to the status bar.

```
virtual CWnd* GetMessageBar( );
```

Return Value

Pointer to the status-bar window.

See Also

Reference

CFrameWnd Class

Hierarchy Chart

Other Resources

CFrameWnd Members

CFrameWnd::GetMessageString

Override this function to provide custom strings for command IDs.

```
virtual void GetMessageString(
   UINT nID,
   CString& rMessage
) const;
```

Parameters

nID

Resource ID of the desired message.

rMessage

CString object into which to place the message.

Remarks

The default implementation simply loads the string specified by *nID* from the resource file. This function is called by the framework when the message string in the status bar needs updating.

See Also

Reference

CFrameWnd Class

Hierarchy Chart

CFrameWnd::SetMessageText

Other Resources

CFrameWnd Members

CFrameWnd::GetTitle

Retrieves the title of the window object.

```
CString GetTitle( ) const;
```

Return Value

A CString object containing the current title of the window object.

See Also

Reference

CFrameWnd Class
Hierarchy Chart
CFrameWnd::SetTitle

Other Resources

CFrameWnd Members

CFrameWnd::InitialUpdateFrame

Call **IntitialUpdateFrame** after creating a new frame with **Create**.

```
void InitialUpdateFrame(
   CDocument* pDoc,
   BOOL bMakeVisible
);
```

Parameters

pDoc

Points to the document to which the frame window is associated. Can be **NULL**.

bMakeVisible

If **TRUE**, indicates that the frame should become visible and active. If **FALSE**, no descendants are made visible.

Remarks

This causes all views in that frame window to receive their **OnInitialUpdate** calls.

Also, if there was not previously an active view, the primary view of the frame window is made active. The primary view is a view with a child ID of **AFX_IDW_PANE_FIRST**. Finally, the frame window is made visible if *bMakeVisible* is nonzero. If *bMakeVisible* is 0, the current focus and visible state of the frame window will remain unchanged. It is not necessary to call this function when using the framework's implementation of File New and File Open.

See Also

Reference

CFrameWnd Class
Hierarchy Chart
CView::OnInitialUpdate
CFrameWnd::SetActiveView
CDocTemplate::CreateNewFrame

Other Resources

CFrameWnd Members

CFrameWnd::InModalState

Call this member function to check if a frame window is modal or modeless.

```
BOOL InModalState( ) const;
```

Return Value

Nonzero if yes; otherwise 0.

See Also

Reference

CFrameWnd Class

Hierarchy Chart

Other Resources

CFrameWnd Members

CFrameWnd::IsTracking

Call this member function to determine if the splitter bar in the window is currently being moved.

```
BOOL IsTracking( ) const;
```

Return Value

Nonzero if a splitter operation is in progress; otherwise 0.

See Also

Reference

CFrameWnd Class

Hierarchy Chart

Other Resources

CFrameWnd Members

CFrameWnd::LoadAccelTable

Call to load the specified accelerator table.

```
BOOL LoadAccelTable(
   LPCTSTR lpszResourceName
);
```

Parameters

lpszResourceName

Identifies the name of the accelerator resource. Use **MAKEINTRESOURCE** if the resource is identified with an integer ID.

Return Value

Nonzero if the accelerator table was successfully loaded; otherwise 0.

Remarks

Only one table can be loaded at a time.

Accelerator tables loaded from resources are freed automatically when the application terminates.

If you call **LoadFrame** to create the frame window, the framework loads an accelerator table along with the menu and icon resources, and a subsequent call to this member function is then unnecessary.

See Also

Reference

CFrameWnd Class
Hierarchy Chart
CFrameWnd::LoadFrame
LoadAccelerators

Other Resources

CFrameWnd Members

CFrameWnd::LoadBarState

Call this function to restore the settings of each control bar owned by the frame window.

```
void LoadBarState(
   LPCTSTR lpszProfileName
);
```

Parameters

lpszProfileName

Name of a section in the initialization (INI) file or a key in the Windows registry where state information is stored.

Remarks

Information restored includes visibility, horizontal/vertical orientation, docking state, and control-bar position.

The settings you want to restore must be written to the registry before you call **LoadBarState**. Write the information to the registry by calling CWinApp::SetRegistryKey. Write the information to the INI file by calling SaveBarState.

See Also

Reference

CFrameWnd Class
Hierarchy Chart
CFrameWnd::SaveBarState
CWinApp::m_pszProfileName

Other Resources

CFrameWnd Members

CFrameWnd::LoadFrame

Call to dynamically create a frame window from resource information.

```
virtual BOOL LoadFrame(
   UINT nIDResource,
   DWORD dwDefaultStyle = WS_OVERLAPPEDWINDOW | FWS_ADDTOTITLE,
   CWnd* pParentWnd = NULL,
   CCreateContext* pContext = NULL
);
```

Parameters

nIDResource

The ID of shared resources associated with the frame window.

dwDefaultStyle

The frame's style. Include the **FWS_ADDTOTITLE** style if you want the title bar to automatically display the name of the document represented in the window.

pParentWnd

A pointer to the frame's parent.

pContext

A pointer to a CCreateContext structure. This parameter can be **NULL**.

Remarks

Construct a **CFrameWnd** object in two steps. First, invoke the constructor, which constructs the **CFrameWnd** object, and then call **LoadFrame**, which loads the Windows frame window and associated resources and attaches the frame window to the **CFrameWnd** object. The *nIDResource* parameter specifies the menu, the accelerator table, the icon, and the string resource of the title for the frame window.

Use the **Create** member function rather than **LoadFrame** when you want to specify all of the frame window's creation parameters.

The framework calls **LoadFrame** when it creates a frame window using a document template object.

The framework uses the *pContext* argument to specify the objects to be connected to the frame window, including any contained view objects. You can set the *pContext* argument to **NULL** when you call **LoadFrame**.

See Also

Reference

CFrameWnd Class
Hierarchy Chart
CDocTemplate Class
CFrameWnd::Create
CFrameWnd::CFrameWnd
CWnd::PreCreateWindow

Other Resources

CFrameWnd Members

CFrameWnd::NegotiateBorderSpace

Call this member function to negotiate border space in a frame window during OLE inplace activation.

```
virtual BOOL NegotiateBorderSpace(
   UINT nBorderCmd,
   LPRECT lpRectBorder
);
```

Parameters

nBorderCmd

Contains one of the following values from the **enum BorderCmd**:

- **borderGet** = 1
- **borderRequest** = 2
- **borderSet** = 3

lpRectBorder

Pointer to a RECT structure or a CRect object that specifies the coordinates of the border.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function is the **CFrameWnd** implementation of OLE border space negotiation.

See Also

Reference

CFrameWnd Class
Hierarchy Chart
IOleInPlaceUIWindow

Other Resources

CFrameWnd Members

CFrameWnd::OnBarCheck

Called whenever an action is performed on the specified control bar.

```
afx_msg BOOL OnBarCheck(
   UINT nID
);
```

Parameters

nID

The ID of the control bar being shown.

Return Value

Nonzero if the control bar existed; otherwise 0.

See Also

Reference

CFrameWnd Class

Hierarchy Chart

Other Resources

CFrameWnd Members

CFrameWnd::OnContextHelp

Handles SHIFT+F1 Help for in-place items.

```
afx_msg void OnContextHelp( );
```

Remarks

To enable context-sensitive help, you must add an

```
ON_COMMAND( ID_CONTEXT_HELP, OnContextHelp )
```

statement to your **CFrameWnd** class message map and also add an accelerator-table entry, typically SHIFT+F1, to enable this member function.

If your application is an OLE Container, **OnContextHelp** puts all in-place items contained within the frame window object into Help mode. The cursor changes to an arrow and a question mark, and the user can then move the mouse pointer and press the left mouse button to select a dialog box, window, menu, or command button. This member function calls the Windows function **WinHelp** with the Help context of the object under the cursor.

See Also

Reference

CFrameWnd Class
Hierarchy Chart
CWinApp::OnHelp
CWinApp::WinHelp

Other Resources

CFrameWnd Members

CFrameWnd::OnCreateClient

Called by the framework during the execution of **OnCreate**.

```
virtual BOOL OnCreateClient(
   LPCREATESTRUCT lpcs,
   CCreateContext* pContext
);
```

Parameters

lpcs

A pointer to a Windows CREATESTRUCT structure.

pContext

A pointer to a CCreateContext structure.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Never call this function.

The default implementation of this function creates a **CView** object from the information provided in *pContext*, if possible.

Override this function to override values passed in the **CCreateContext** object or to change the way controls in the main client area of the frame window are created. The **CCreateContext** members you can override are described in the CCreateContext class.

Note
Do not replace values passed in the **CREATESTRUCT** structure. They are for informational use only. If you want to override t he initial window rectangle, for example, override the **CWnd** member function PreCreateWindow.

See Also

Reference

CFrameWnd Class

Hierarchy Chart

Other Resources

CFrameWnd Members

CFrameWnd::OnSetPreviewMode

Call this member function to set the application's main frame window into and out of print-preview mode.

```
virtual void OnSetPreviewMode(
   BOOL bPreview,
   CPrintPreviewState* pState
);
```

Parameters

bPreview

Specifies whether or not to place the application in print-preview mode. Set to **TRUE** to place in print preview, **FALSE** to cancel preview mode.

pState

A pointer to a **CPrintPreviewState** structure.

Remarks

The default implementation disables all standard toolbars and hides the main menu and the main client window. This turns MDI frame windows into temporary SDI frame windows.

Override this member function to customize the hiding and showing of control bars and other frame window parts during print preview. Call the base class implementation from within the overridden version.

See Also

Reference

CFrameWnd Class

Hierarchy Chart

Other Resources

CFrameWnd Members

CFrameWnd::OnUpdateControlBarMenu

Called by the framework when the associated menu is updated.

```
afx_msg void OnUpdateControlBarMenu(
   CCmdUI* pCmdUI
);
```

Parameters

pCmdUI

A pointer to a CCmdUI object representing the menu that generated the update command. The update handler calls the Enable member function of the **CCmdUI** object through *pCmdUI* to update the user interface.

See Also

Reference

CFrameWnd Class

Hierarchy Chart

Other Resources

CFrameWnd Members

CFrameWnd::RecalcLayout

Called by the framework when the standard control bars are toggled on or off or when the frame window is resized.

```
virtual void RecalcLayout(
   BOOL bNotify = TRUE
);
```

Parameters

bNotify

Determines whether the active in-place item for the frame window receives notification of the layout change. If **TRUE**, the item is notified; otherwise **FALSE**.

Remarks

The default implementation of this member function calls the **CWnd** member function **RepositionBars** to reposition all the control bars in the frame as well as in the main client window (usually a **CView** or **MDICLIENT**).

Override this member function to control the appearance and behavior of control bars after the layout of the frame window has changed. For example, call it when you turn control bars on or off or add another control bar.

See Also

Reference

CFrameWnd Class

Hierarchy Chart

CWnd::RepositionBars

Other Resources

CFrameWnd Members

CFrameWnd::SaveBarState

Call this function to store information about each control bar owned by the frame window.

```
void SaveBarState(
   LPCTSTR lpszProfileName
) const;
```

Parameters

lpszProfileName

Name of a section in the initialization file or a key in the Windows registry where state information is stored.

Remarks

This information can be read from the initialization file using LoadBarState. Information stored includes visibility, horizontal/vertical orientation, docking state, and control bar position.

See Also

Reference

CFrameWnd Class
Hierarchy Chart
CFrameWnd::LoadBarState
CWinApp::SetRegistryKey
CWinApp::m_pszProfileName

Other Resources

CFrameWnd Members

CFrameWnd::SetActiveView

Call this member function to set the active view.

```
void SetActiveView(
   CView* pViewNew,
   BOOL bNotify = TRUE
);
```

Parameters

pViewNew

Specifies a pointer to a CView object, or **NULL** for no active view.

bNotify

Specifies whether the view is to be notified of activation. If **TRUE**, **OnActivateView** is called for the new view; if **FALSE**, it is not.

Remarks

The framework will call this function automatically as the user changes the focus to a view within the frame window. You can explicitly call **SetActiveView** to change the focus to the specified view.

See Also

Reference

CFrameWnd Class
Hierarchy Chart
CFrameWnd::GetActiveView
CView::OnActivateView
CFrameWnd::GetActiveDocument

Other Resources

CFrameWnd Members

CFrameWnd::SetDockState

Call this member function to apply state information stored in a **CDockState** object to the frame window's control bars.

```
void SetDockState(
   const CDockState& state
);
```

Parameters

state

Apply the stored state to the frame window's control bars.

Remarks

To restore a previous state of the control bars, you can load the stored state with **CDockState::LoadState** or **Serialize**, then use **SetDockState** to apply it to the frame window's control bars. The previous state is stored in the **CDockState** object with **GetDockState**

See Also

Reference

CFrameWnd Class
Hierarchy Chart
CFrameWnd::GetDockState
CDockState Class
CDockState::LoadState
CObject::Serialize

Other Resources

CFrameWnd Members

CFrameWnd::SetMessageText

Call this function to place a string in the status-bar pane that has an ID of 0.

```
void SetMessageText(
   LPCTSTR lpszText
);
void SetMessageText(
   UINT nID
);
```

Parameters

lpszText

Points to the string to be placed on the status bar.

nID

String resource ID of the string to be placed on the status bar.

Remarks

This is typically the leftmost, and longest, pane of the status bar.

See Also

Reference

CFrameWnd Class
Hierarchy Chart
CStatusBar Class

Other Resources

CFrameWnd Members

CFrameWnd::SetTitle

Sets the title of the window object.

```
void SetTitle(
   LPCTSTR lpszTitle
);
```

Parameters

lpszTitle

A pointer to a character string containing the title of the window object.

See Also

Reference

CFrameWnd Class

Hierarchy Chart

CFrameWnd::GetTitle

Other Resources

CFrameWnd Members

CFrameWnd::ShowControlBar

Call this member function to show or hide the control bar.

```
void ShowControlBar(
   CControlBar* pBar,
   BOOL bShow,
   BOOL bDelay
);
```

Parameters

pBar

Pointer to the control bar to be shown or hidden.

bShow

If **TRUE**, specifies that the control bar is to be shown. If **FALSE**, specifies that the control bar is to be hidden.

bDelay

If **TRUE**, delay showing the control bar. If **FALSE**, show the control bar immediately.

See Also

Reference

CFrameWnd Class

Hierarchy Chart

Other Resources

CFrameWnd Members

CFrameWnd::ShowOwnedWindows

Call this member function to show all windows that are descendants of the **CFrameWnd** object.

```
void ShowOwnedWindows(
   BOOL bShow
);
```

Parameters

bShow

Specifies whether the owned windows are to be shown or hidden.

See Also

Reference

CFrameWnd Class

Hierarchy Chart

Other Resources

CFrameWnd Members

CFrameWnd Data Members

For information about the data members in **CFrameWnd**, see CFrameWnd Members.

CFrameWnd::m_bAutoMenuEnable

When this data member is enabled (which is the default), menu items that do not have **ON_UPDATE_COMMAND_UI** or **ON_COMMAND** handlers will be automatically disabled when the user pulls down a menu.

```
BOOL m_bAutoMenuEnable;
```

Remarks

Menu items that have an **ON_COMMAND** handler but no **ON_UPDATE_COMMAND_UI** handler will be automatically enabled.

When this data member is set, menu items are automatically enabled in the same way that toolbar buttons are enabled.

Note
m_bAutoMenuEnable has no effect on top-level menu items.

This data member simplifies the implementation of optional commands based on the current selection and reduces the need to write **ON_UPDATE_COMMAND_UI** handlers for enabling and disabling menu items.

Example

```
CMainFrame::CMainFrame()
{
   // Set to FALSE so no ON_UPDATE_COMMAND_UI or
   // ON_COMMAND handlers are needed, and
   // CMenu::EnableMenuItem() will work as expected.
   m_bAutoMenuEnable  = FALSE;
}
```

See Also

Reference

CFrameWnd Class
Hierarchy Chart
CCmdUI Class
CCmdTarget Class

Other Resources

CFrameWnd Members

CFrameWnd::rectDefault

Pass this static **CRect** as a parameter when creating a window to allow Windows to choose the window's initial size and position.

```
static AFX_DATA const CRect rectDefault;
```

See Also

Reference

CFrameWnd Class

Hierarchy Chart

Other Resources

CFrameWnd Members

CFtpConnection Class

Manages your FTP connection to an Internet server and allows direct manipulation of directories and files on that server.

```
class CFtpConnection : public CInternetConnection
```

Remarks

FTP is one of the three Internet services recognized by the MFC WinInet classes.

To communicate with an FTP Internet server, you must first create an instance of CInternetSession, and then create a **CFtpConnection** object. You never create a **CFtpConnection** object directly; rather, call CInternetSession::GetFtpConnection, which creates the **CFtpConnection** object and returns a pointer to it.

To learn more about how **CFtpConnection** works with the other MFC Internet classes, see the article Internet Programming with WinInet. For more information about communicating with the other two supported services, HTTP and gopher, see the classes CHttpConnection and CGopherConnection.

Example

See the example in the CFtpFileFind class overview.

Requirements

Header: afxinet.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CInternetConnection Class
Hierarchy Chart
CInternetConnection Class
CInternetSession Class

Other Resources

CFtpConnection Members

CFtpConnection Members

Base Class Members

CObject Members

CInternetConnection Members

Construction

CFtpConnection	Constructs a **CFtpConnection** object.

Operations

Command	Sends a command directly to an FTP server.
CreateDirectory	Creates a directory on the server.
GetCurrentDirectory	Gets the current directory for this connection.
GetCurrentDirectoryAsURL	Gets the current directory for this connection as a URL.
GetFile	Gets a file from the connected server
OpenFile	Opens a file on the connected server.
PutFile	Places a file on the server.
Remove	Removes a file from the server.
RemoveDirectory	Removes the specified directory from the server.
Rename	Renames a file on the server.
SetCurrentDirectory	Sets the current FTP directory.

See Also

Reference

CFtpConnection Class
Hierarchy Chart
CInternetConnection Class

CFtpConnection Member Functions

For information about the member functions in **CFtpConnection**, see CFtpConnection Members.

CFtpConnection::CFtpConnection

This member function is called to construct a **CFtpConnection** object.

```
CFtpConnection(
   CInternetSession* pSession,
   HINTERNET hConnected,
   LPCTSTR pstrServer,
   DWORD_PTR dwContext
);
CFtpConnection(
   CInternetSession* pSession,
   LPCTSTR pstrServer,
   LPCTSTR pstrUserName = NULL,
   LPCTSTR pstrPassword = NULL,
   DWORD_PTR dwContext = 0,
   INTERNET_PORT nPort = INTERNET_INVALID_PORT_NUMBER,
   BOOL bPassive = FALSE
);
```

Parameters

pSession

A pointer to the related CInternetSession object.

hConnected

The Windows handle of the current Internet session.

pstrServer

A pointer to a string containing the FTP server name.

dwContext

The context identifier for the operation. *dwContext* identifies the operation's status information returned by CInternetSession::OnStatusCallback. The default is set to 1; however, you can explicitly assign a specific context ID for the operation. The object and any work it does will be associated with that context ID.

pstrUserName

Pointer to a null-terminated string that specifies the name of the user to log in. If **NULL**, the default is anonymous.

pstrPassword

A pointer to a null-terminated string that specifies the password to use to log in. If both *pstrPassword* and *pstrUserName* are **NULL**, the default anonymous password is the user's email name. If *pstrPassword* is **NULL** (or an empty string) but *pstrUserName* is not **NULL**, a blank password is used. The following table describes the behavior for the four possible settings of *pstrUserName* and *pstrPassword*:

pstrUserName	***pstrPassword***	**Username sent to FTP server**	**Password sent to FTP server**
NULL or " "	**NULL** or " "	"anonymous"	User's email name
Non-**NULL** String	**NULL** or " "	*pstrUserName*	" "
NULL Non-**NULL** String	**ERROR**	**ERROR**	
Non-**NULL** String	Non-**NULL** String	*pstrUserName*	*pstrPassword*

nPort

A number that identifies the TCP/IP port to use on the server.

bPassive

Specifies passive or active mode for this FTP session. If set to **TRUE**, it sets the Win32 API *dwFlag* to **INTERNET_FLAG_PASSIVE**.

Remarks

You never create a **CFtpConnection** object directly. Instead, call CInternetSession::GetFtpConnection, which creates the **CFptConnection** object.

Requirements

Header: afxinet.h

See Also

Reference

CFtpConnection Class
Hierarchy Chart
CInternetSession::GetFtpConnection
CFtpFileFind Class
CGopherConnection Class
CHttpConnection Class
CInternetConnection Class

Other Resources

CFtpConnection Members

CFtpConnection::Command

Sends a command directly to an FTP server.

```
CInternetFile* Command(
   LPCTSTR pszCommand,
   CmdResponseType eResponse = CmdRespNone,
   DWORD dwFlags = FTP_TRANSFER_TYPE_BINARY,
   DWORD_PTR dwContext = 1
);
```

Parameters

pszCommand

A pointer to a string containing the command to be sent.

eResponse

Determines whether a response is expected from the FTP server. Can be one of the following values:

- **CmdRespNone** No response is expected.
- **CmdRespRead** A response is expected.

dwFlags

A value containing the flags that control this function. For a complete list, see FTPCommand.

dwContext

A pointer to a value containing an application-defined value used to identify the application context in callbacks.

Return Value

Nonzero if successful; otherwise 0. If the call fails, the Windows function GetLastError may be called to determine the cause of the error.

Remarks

This member function emulates the functionality of the FTPCommand function, as described in the *Platform SDK*.

Requirements

Header: afxinet.h

See Also

Reference

CFtpConnection Class

Hierarchy Chart

Other Resources

CFtpConnection Members

CFtpConnection::CreateDirectory

Call this member function to create a directory on the connected server.

```
BOOL CreateDirectory(
   LPCTSTR pstrDirName
);
```

Parameters

pstrDirName

A pointer to a string containing the name of the directory to create.

Return Value

Nonzero if successful; otherwise 0. If the call fails, the Windows function GetLastError may be called to determine the cause of the error.

Remarks

Use **GetCurrentDirectory** to determine the current working directory for this connection to the server. Do not assume that the remote system has connected you to the root directory.

The *pstrDirName* parameter can be either a partially or a fully qualified filename relative to the current directory. A backslash (\) or forward slash (/) can be used as the directory separator for either name. **CreateDirectory** translates the directory name separators to the appropriate characters before they are used.

Requirements

Header: afxinet.h

See Also

Reference

CFtpConnection Class
Hierarchy Chart
CInternetConnection Class

Other Resources

CFtpConnection Members

CFtpConnection::GetCurrentDirectory

Call this member function to get the name of the current directory.

```
BOOL GetCurrentDirectory(
   CString& strDirName
) const;
BOOL GetCurrentDirectory(
   LPTSTR pstrDirName,
   LPDWORD lpdwLen
) const;
```

Parameters

strDirName

A reference to a string that will receive the name of the directory.

pstrDirName

A pointer to a string that will receive the name of the directory.

lpdwLen

A pointer to a DWORD that contains the following information:

On entry	The size of the buffer referenced by *pstrDirName*.
On return	The number of characters stored to *pstrDirName*. If the member function fails and ERROR_INSUFFICIENT_BUFFER is returned, then *lpdwLen* contains the number of bytes that the application must allocate in order to receive the string.

Return Value

Nonzero if successful; otherwise 0. If the call fails, the Win32 function GetLastError may be called to determine the cause of the error.

Remarks

To get the directory name as a URL instead, call GetCurrentDirectoryAsURL.

The parameters *pstrDirName* or *strDirName* can be either partially qualified filenames relative to the current directory or fully qualified. A backslash (\) or forward slash (/) can be used as the directory separator for either name. **GetCurrentDirectory** translates the directory name separators to the appropriate characters before they are used.

Requirements

Header: afxinet.h

See Also

Reference

CFtpConnection Class
Hierarchy Chart
CFtpConnection::GetCurrentDirectoryAsURL
CInternetConnection Class

Other Resources

CFtpConnection Members

CFtpConnection::GetCurrentDirectoryAsURL

Call this member function to get the current directory's name as a URL.

```
BOOL GetCurrentDirectoryAsURL(
   CString& strDirName
) const;
BOOL GetCurrentDirectoryAsURL(
   LPTSTR pstrName,
   LPDWORD lpdwLen
) const;
```

Parameters

strDirName

A reference to a string that will receive the name of the directory.

pstrDirName

A pointer to a string that will receive the name of the directory.

lpdwLen

A pointer to a DWORD that contains the following information:

On entry	The size of the buffer referenced by *pstrDirName*.
On return	The number of characters stored to *pstrDirName*. If the member function fails and ERROR_INSUFFICIENT_BUFFER is returned, then *lpdwLen* contains the number of bytes that the application must allocate in order to receive the string.

Return Value

Nonzero if successful; otherwise 0. If the call fails, the Win32 function GetLastError may be called to determine the cause of the error.

Remarks

GetCurrentDirectoryAsURL behaves the same as GetCurrentDirectory

The parameter *strDirName* can be either partially qualified filenames relative to the current directory or fully qualified. A backslash (\) or forward slash (/) can be used as the directory separator for either name. **GetCurrentDirectoryAsURL** translates the directory name separators to the appropriate characters before they are used.

Requirements

Header: afxinet.h

See Also

Reference

CFtpConnection Class
Hierarchy Chart
CFtpConnection::GetCurrentDirectory
CInternetConnection Class

Other Resources

CFtpConnection Members

CFtpConnection::GetFile

Call this member function to get a file from an FTP server and store it on the local machine.

```
BOOL GetFile(
   LPCTSTR pstrRemoteFile,
   LPCTSTR pstrLocalFile,
   BOOL bFailIfExists = TRUE,
   DWORD dwAttributes = FILE_ATTRIBUTE_NORMAL,
   DWORD dwFlags = FTP_TRANSFER_TYPE_BINARY,
   DWORD_PTR dwContext = 1
);
```

Parameters

pstrRemoteFile

A pointer to a null-terminated string containing the name of a file to retrieve from the FTP server.

pstrLocalFile

A pointer to a null-terminated string containing the name of the file to create on the local system.

bFailIfExists

Indicates whether the file name may already be used by an existing file. If the local file name already exists, and this parameter is **TRUE**, **GetFile** fails. Otherwise, **GetFile** will erase the existing copy of the file.

dwAttributes

Indicates the attributes of the file. This can be any combination of the following FILE_ATTRIBUTE_* flags.

- FILE_ATTRIBUTE_ARCHIVE The file is an archive file. Applications use this attribute to mark files for backup or removal.
- FILE_ATTRIBUTE_COMPRESSED The file or directory is compressed. For a file, compression means that all of the data in the file is compressed. For a directory, compression is the default for newly created files and subdirectories.
- FILE_ATTRIBUTE_DIRECTORY The file is a directory.
- FILE_ATTRIBUTE_NORMAL The file has no other attributes set. This attribute is valid only if used alone. All other file attributes override FILE_ATTRIBUTE_NORMAL:
- FILE_ATTRIBUTE_HIDDEN The file is hidden. It is not to be included in an ordinary directory listing.
- FILE_ATTRIBUTE_READONLY The file is read only. Applications can read the file but cannot write to it or delete it.
- FILE_ATTRIBUTE_SYSTEM The file is part of or is used exclusively by the operating system.
- FILE_ATTRIBUTE_TEMPORARY The file is being used for temporary storage. Applications should write to the file only if absolutely necessary. Most of the file's data remains in memory without being flushed to the media because the file will soon be deleted.

dwFlags

Specifies the conditions under which the transfer occurs. This parameter can be any of the *dwFlags* values described in FtpGetFile in the *Platform SDK*.

dwContext

The context identifier for the file retrieval. See **Remarks** for more information about *dwContext*.

Return Value

Nonzero if successful; otherwise 0. If the call fails, the Win32 function GetLastError may be called to determine the cause of the error.

Remarks

GetFile is a high-level routine that handles all of the overhead associated with reading a file from an FTP server and storing it

locally. Applications that only retrieve file data, or that require close control over the file transfer, should use **OpenFile** and CInternetFile::Read instead.

If *dwFlags* is FILE_TRANSFER_TYPE_ASCII, translation of file data also converts control and formatting characters to Windows equivalents. The default transfer is binary mode, where the file is downloaded in the same format as it is stored on the server.

Both *pstrRemoteFile* and *pstrLocalFile* can be either partially qualified filenames relative to the current directory or fully qualified. A backslash (\) or forward slash (/) can be used as the directory separator for either name. **GetFile** translates the directory name separators to the appropriate characters before they are used.

Override the *dwContext* default to set the context identifier to a value of your choosing. The context identifier is associated with this specific operation of the **CFtpConnection** object created by its CInternetSession object. The value is returned to CInternetSession::OnStatusCallback to provide status on the operation with which it is identified. See the article Internet First Steps: WinInet for more information about the context identifier.

Requirements

Header: afxinet.h

See Also

Reference

CFtpConnection Class
Hierarchy Chart
CInternetConnection Class

Other Resources

CFtpConnection Members

CFtpConnection::OpenFile

Call this member function to open a file located on an FTP server for reading or writing.

```
CInternetFile* OpenFile(
   LPCTSTR pstrFileName,
   DWORD dwAccess = GENERIC_READ,
   DWORD dwFlags = FTP_TRANSFER_TYPE_BINARY,
   DWORD_PTR dwContext = 1
);
```

Parameters

pstrFileName

A pointer to a string containing the name of the file to be opened.

dwAccess

Determines how the file will be accessed. Can be either GENERIC_READ or GENERIC_WRITE, but not both.

dwFlags

Specifies the conditions under which subsequent transfers occur. This can be any of the following FTP_TRANSFER_* constants:

- FTP_TRANSFER_TYPE_ASCII The file transfers using FTP ASCII (Type A) transfer method. Converts control and formatting information to local equivalents.
- FTP_TRANSFER_TYPE_BINARY The file transfers data using FTP's Image (Type I) transfer method. The file transfers data exactly as it exists, with no changes. This is the default transfer method.

dwContext

The context identifier for opening the file. See **Remarks** for more information about *dwContext*.

Return Value

A pointer to a CInternetFile object.

Remarks

OpenFile should be used in the following situations:

- An application has data that needs to be sent and created as a file on the FTP server, but that data is not in a local file. Once **OpenFile** opens a file, the application uses CInternetFile::Write to send the FTP file data to the server.
- An application must retrieve a file from the server and place it into application-controlled memory, instead of writing it to disk. The application uses CInternetFile::Read after using **OpenFile** to open the file.
- An application needs a fine level of control over a file transfer. For example, the application may want to display a progress control indicate the progress of the file transfer status while downloading a file.

After calling **OpenFile** and until calling **CInternetConnection::Close**, the application can only call CInternetFile::Read, CInternetFile::Write, **CInternetConnection::Close**, or CFtpFileFind::FindFile. Calls to other FTP functions for the same FTP session will fail and set the error code to FTP_ETRANSFER_IN_PROGRESS.

The *pstrFileName* parameter can be either a partially qualified filename relative to the current directory or fully qualified. A backslash (\) or forward slash (/) can be used as the directory separator for either name. **OpenFile** translates the directory name separators to the appropriate characters before using it.

Override the *dwContext* default to set the context identifier to a value of your choosing. The context identifier is associated with this specific operation of the **CFtpConnection** object created by its CInternetSession object. The value is returned to CInternetSession::OnStatusCallback to provide status on the operation with which it is identified. See the article Internet First Steps: WinInet for more information about the context identifier.

Requirements

Header: afxinet.h

See Also

Reference

CFtpConnection Class
Hierarchy Chart
CInternetConnection Class
CFtpConnection::GetFile
CGopherConnection::OpenFile
CInternetFile::Write
CInternetFile::Read

Other Resources

CFtpConnection Members

CFtpConnection::PutFile

Call this member function to store a file on an FTP server.

```
BOOL PutFile(
   LPCTSTR pstrLocalFile,
   LPCTSTR pstrRemoteFile,
   DWORD dwFlags = FTP_TRANSFER_TYPE_BINARY,
   DWORD_PTR dwContext = 1
);
```

Parameters

pstrLocalFile

A pointer to a string containing the name of the file to send from the local system.

pstrRemoteFile

A pointer to a string containing the name of the file to create on the FTP server.

dwFlags

Specifies the conditions under which the transfer of the file occurs. Can be any of the FTP_TRANSFER_* constants described in OpenFile.

dwContext

The context identifier for placing the file. See **Remarks** for more information about *dwContext*.

Return Value

Nonzero if successful; otherwise 0. If the call fails, the Win32 function GetLastError may be called to determine the cause of the error.

Remarks

PutFile is a high-level routine that handles all of the operations associated with storing a file on an FTP server. Applications that only send data, or that require closer control over the file transfer, should use OpenFile and CInternetFile::Write.

Override the *dwContext* default to set the context identifier to a value of your choosing. The context identifier is associated with this specific operation of the **CFtpConnection** object created by its CInternetSession object. The value is returned to CInternetSession::OnStatusCallback to provide status on the operation with which it is identified. See the article Internet First Steps: WinInet for more information about the context identifier.

Requirements

Header: afxinet.h

See Also

Reference

CFtpConnection Class
Hierarchy Chart
CInternetConnection Class

Other Resources

CFtpConnection Members

CFtpConnection::Remove

Call this member function to delete the specified file from the connected server.

```
BOOL Remove(
   LPCTSTR pstrFileName
);
```

Parameters

pstrFileName

A pointer to a string containing the file name to remove.

Return Value

Nonzero if successful; otherwise 0. If the call fails, the Win32 function GetLastError may be called to determine the cause of the error.

Remarks

The *pstrFileName* parameter can be either a partially qualified filename relative to the current directory or fully qualified. A backslash (\) or forward slash (/) can be used as the directory separator for either name. The **Remove** function translates the directory name separators to the appropriate characters before they are used.

Requirements

Header: afxinet.h

See Also

Reference

CFtpConnection Class
Hierarchy Chart
CInternetConnection Class

Other Resources

CFtpConnection Members

CFtpConnection::RemoveDirectory

Call this member function to remove the specified directory from the connected server.

```
BOOL RemoveDirectory(
   LPCTSTR pstrDirName
);
```

Parameters

pstrDirName

A pointer to a string containing the directory to be removed.

Return Value

Nonzero if successful; otherwise 0. If the call fails, the Win32 function GetLastError may be called to determine the cause of the error.

Remarks

Use GetCurrentDirectory to determine the server's current working directory. Do not assume that the remote system has connected you to the root directory.

The *pstrDirName* parameter can be either a partially or fully qualified filename relative to the current directory. A backslash (\) or forward slash (/) can be used as the directory separator for either name. **RemoveDirectory** translates the directory name separators to the appropriate characters before they are used.

Requirements

Header: afxinet.h

See Also

Reference

CFtpConnection Class
Hierarchy Chart
CInternetConnection Class

Other Resources

CFtpConnection Members

CFtpConnection::Rename

Call this member function to rename the specified file on the connected server.

```
BOOL Rename(
   LPCTSTR pstrExisting,
   LPCTSTR pstrNew
);
```

Parameters

pstrExisting

A pointer to a string containing the current name of the file to be renamed.

pstrNew

A pointer to a string containing the file's new name.

Return Value

Nonzero if successful; otherwise 0. If the call fails, the Win32 function GetLastError may be called to determine the cause of the error.

Remarks

The *pstrExisting* and *pstrNew* parameters can be either a partially qualified filename relative to the current directory or fully qualified. A backslash (\) or forward slash (/) can be used as the directory separator for either name. **Rename** translates the directory name separators to the appropriate characters before they are used.

Requirements

Header: afxinet.h

See Also

Reference

CFtpConnection Class
Hierarchy Chart
CInternetConnection Class

Other Resources

CFtpConnection Members

CFtpConnection::SetCurrentDirectory

Call this member function to change to a different directory on the FTP server.

```
BOOL SetCurrentDirectory(
   LPCTSTR pstrDirName
);
```

Parameters

pstrDirName

A pointer to a string containing the name of the directory.

Return Value

Nonzero if successful; otherwise 0. If the call fails, the Win32 function GetLastError may be called to determine the cause of the error.

Remarks

The *pstrDirName* parameter can be either a partially or fully qualified filename relative to the current directory. A backslash (\) or forward slash (/) can be used as the directory separator for either name. **SetCurrentDirectory** translates the directory name separators to the appropriate characters before they are used.

Use GetCurrentDirectory to determine an FTP server's current working directory. Do not assume that the remote system has connected you to the root directory.

Requirements

Header: afxinet.h

See Also

Reference

CFtpConnection Class
Hierarchy Chart
CInternetConnection Class

Other Resources

CFtpConnection Members

CFtpFileFind Class

Aids in Internet file searches of FTP servers.

```
class CFtpFileFind : public CFileFind
```

Remarks

CFtpFileFind includes member functions that begin a search, locate a file, and return the URL or other descriptive information about the file.

Other MFC classes designed for Internet and local file searched include CGopherFileFind and CFileFind. Together with **CFtpFileFind**, these classes provide a seamless mechanism for the client to find specific files, regardless of the server protocol or file type (either a local machine or a remote server). Note that there is no MFC class for searching on HTTP servers because HTTP does not support the direct file manipulation required for searches.

For more information about how to use **CFtpFileFind** and the other WinInet classes, see the article Internet Programming with WinInet.

Example

The following code demonstrates how to enumerate all files in the current directory of the FTP server.

```
// cftpfilefind_class.cpp
// Compile with: /MT /EHsc
#include <afx.h>
#include <afxwin.h>
#include <afxinet.h>
#include <stdio.h>

CWinApp theApp;

int main()
{
    if (!AfxWinInit(::GetModuleHandle(NULL), NULL,
                    ::GetCommandLine(), 0))
    {
        // catastropic error! MFC can't initialize
        return -1;
    }

    // create a session object to initialize WININET library
    // Default parameters mean the access method in the registry
    // (that is, set by the "Internet" icon in the Control Panel)
    // will be used.

    CInternetSession sess(_T("MyProgram/1.0"));

    CFtpConnection* pConnect = NULL;

    try
    {
        // Request a connection to ftp.microsoft.com. Default
        // parameters mean that we'll try with username = ANONYMOUS
        // and password set to the machine name @ domain name
        pConnect = sess.GetFtpConnection(_T("ftp.microsoft.com"));

        // use a file find object to enumerate files
        CFtpFileFind finder(pConnect);

        // start looping
        BOOL bWorking = finder.FindFile(_T("*"));

        while (bWorking)
```

```
        {
            bWorking = finder.FindNextFile();
            printf_s("%s\n", (LPCTSTR) finder.GetFileURL());
        }
    }
    catch (CInternetException* pEx)
    {
        TCHAR sz[1024];
        pEx->GetErrorMessage(sz, 1024);
        printf_s("ERROR!  %s\n", sz);
        pEx->Delete();
     }

    // if the connection is open, close it
    if (pConnect != NULL)
    {
        pConnect->Close();
        delete pConnect;
    }

   return 0;
}
```

Requirements

Header: afxinet.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CFileFind Class
Hierarchy Chart
CGopherFileFind Class
CInternetFile Class
CGopherFile Class
CHttpFile Class

Other Resources

CFtpFileFind Members

CFtpFileFind Members

Base Class Members

CObject Members

CFileFind Members

Construction

CFtpFileFind	Constructs a **CFtpFileFind** object.

Operations

FindFile	Finds a file on a FTP server.
FindNextFile	Continues a file search from a previous call to FindFile.
GetFileURL	Gets the URL, including path, of the found file.

See Also

Reference

CFtpFileFind Class
Hierarchy Chart
CFtpFileFind Class
CGopherFileFind Class
CInternetFile Class
CGopherFile Class
CHttpFile Class

CFtpFileFind Member Functions

For information about the member functions in **CFtpFileFind**, see CFtpFileFind Members.

CFtpFileFind::CFtpFileFind

This member function is called to construct a **CFtpFileFind** object.

```
explicit CFtpFileFind(
   CFtpConnection* pConnection,
   DWORD_PTR dwContext = 1
);
```

Parameters

pConnection

A pointer to a **CFtpConnection** object. You can obtain an FTP connection by calling CInternetSession::GetFtpConnection.

dwContext

The context identifier for the **CFtpFileFind** object. See **Remarks** for more information about this parameter.

Remarks

The default value for *dwContext* is sent by MFC to the **CFtpFileFind** object from the CInternetSession object that created the **CFtpFileFind** object. You can override the default to set the context identifier to a value of your choosing. The context identifier is returned to CInternetSession::OnStatusCallback to provide status on the object with which it is identified. See the article Internet First Steps: WinInet for more information about the context identifier.

Example

See the example in the CFtpFileFind class overview.

See Also

Reference

CFtpFileFind Class
Hierarchy Chart
CGopherFileFind Class
CFileFind Class

Other Resources

CFtpFileFind Members

CFtpFileFind::FindFile

Call this member function to find an FTP file.

```
virtual BOOL FindFile(
   LPCTSTR pstrName = NULL,
   DWORD dwFlags = INTERNET_FLAG_RELOAD
);
```

Parameters

pstrName

A pointer to a string containing the name of the file to find. If **NULL**, the call will perform a wildcard search (*).

dwFlags

The flags describing how to handle this session. These flags can be combined with the bitwise OR operator (|) and are as follows:

- INTERNET_FLAG_RELOAD Get the data from the wire even if it is locally cached. This is the default flag.
- INTERNET_FLAG_DONT_CACHE Do not cache the data, either locally or in any gateways.
- INTERNET_FLAG_RAW_DATA Override the default to return the raw data (WIN32_FIND_DATA structures for FTP).
- INTERNET_FLAG_SECURE Secures transactions on the wire with Secure Sockets Layer or PCT. This flag is applicable to HTTP requests only.
- INTERNET_FLAG_EXISTING_CONNECT If possible, reuse the existing connections to the server for new **FindFile** requests instead of creating a new session for each request.

Return Value

Nonzero if successful; otherwise 0. To get extended error information, call the Win32 function GetLastError.

Remarks

After calling **FindFile** to retrieve the first FTP file, you can call FindNextFile to retrieve subsequent FTP files.

Example

See the example in the CFtpFileFind class overview.

See Also

Reference

CFtpFileFind Class
Hierarchy Chart
CFtpFileFind::FindNextFile
CFileFind Class

Other Resources

CFtpFileFind Members

CFtpFileFind::FindNextFile

Call this member function to continue a file search begun with a call to the FindFile member function.

```
virtual BOOL FindNextFile( );
```

Return Value

Nonzero if there are more files; zero if the file found is the last one in the directory or if an error occurred. To get extended error information, call the Win32 function GetLastError. If the file found is the last file in the directory, or if no matching files can be found, the **GetLastError** function returns ERROR_NO_MORE_FILES.

Remarks

You must call this function at least once before calling any attribute function (see CFileFind::FindNextFile).

FindNextFile wraps the Win32 function FindNextFile.

Example

See the example in the CFtpFileFind class overview.

See Also

Reference

CFtpFileFind Class
Hierarchy Chart
CFileFind Class

Other Resources

CFtpFileFind Members

CFtpFileFind::GetFileURL

Call this member function to get the URL of the specified file.

```
CString GetFileURL( ) const;
```

Return Value

The file and path of the Universal Resource Locator (URL).

Remarks

GetFileURL is similar to the member function CFileFind::GetFilePath, except that it returns the URL in the form `ftp://moose/dir/file.txt`.

See Also

Reference

CFtpFileFind Class
Hierarchy Chart
CFtpFileFind::FindFile
CFileFind Class

Other Resources

CFtpFileFind Members

CGdiObject Class

Provides a base class for various kinds of Windows graphics device interface (GDI) objects such as bitmaps, regions, brushes, pens, palettes, and fonts.

```
class CGdiObject : public CObject
```

Remarks

You never create a **CGdiObject** directly. Rather, you create an object from one of its derived classes, such as **CPen** or **CBrush**.

For more information on **CGdiObject**, see Graphic Objects.

Requirements

Header: afxwin.h

See Also

Reference

Base Class
Hierarchy Chart
CBitmap Class
CBrush Class
CFont Class
CPalette Class
CPen Class
CRgn Class

Other Resources

CGdiObject Members

CGdiObject Members

Base Class Members

COBJECT Members

Data Members

m_hObject	A **HANDLE** containing the **HBITMAP**, **HPALETTE**, **HRGN**, **HBRUSH**, **HPEN**, or **HFONT** attached to this object.

Construction

CGdiObject	Constructs a **CGdiObject** object.

Operations

Attach	Attaches a Windows GDI object to a **CGdiObject** object.
CreateStockObject	Retrieves a handle to one of the Windows predefined stock pens, brushes, or fonts.
DeleteObject	Deletes the Windows GDI object attached to the **CGdiObject** object from memory by freeing all system storage associated with the object.
DeleteTempMap	Deletes any temporary **CGdiObject** objects created by **FromHandle**.
Detach	Detaches a Windows GDI object from a **CGdiObject** object and returns a handle to the Windows GDI object.
FromHandle	Returns a pointer to a **CGdiObject** object given a handle to a Windows GDI object.
GetObject	Fills a buffer with data that describes the Windows GDI object attached to the **CGdiObject** object.
GetObjectType	Retrieves the type of the GDI object.
GetSafeHandle	Returns **m_hObject** unless **this** is **NULL**, in which case **NULL** is returned.
UnrealizeObject	Resets the origin of a brush or resets a logical palette.

Operators

operator !=	Determines if two GDI objects are logically not equal.
operator ==	Determines if two GDI objects are logically equal.
operator HGDIOBJ	Retrieves a **HANDLE** to the attached Windows GDI object.

See Also

Reference

CGdiObject Class

Hierarchy Chart

CGdiObject Member Functions

For information about the member functions in **CGdiObject**, see CGdiObject Members.

CGdiObject::Attach

Attaches a Windows GDI object to a **CGdiObject** object.

```
BOOL Attach(
   HGDIOBJ hObject
);
```

Parameters

hObject

A **HANDLE** to a Windows GDI object (for example, **HPEN** or **HBRUSH**).

Return Value

Nonzero if attachment is successful; otherwise 0.

See Also

Reference

CGdiObject Class

Hierarchy Chart

CGdiObject::Detach

Other Resources

CGdiObject Members

CGdiObject::CGdiObject

Constructs a **CGdiObject** object.

```
CGdiObject( );
```

Remarks

You never create a **CGdiObject** directly. Rather, you create an object from one of its derived classes, such as **CPen** or **Cbrush**.

See Also

Reference

CGdiObject Class
Hierarchy Chart
CPen Class
CBrush Class
CFont Class
CBitmap Class
CRgn Class
CPalette Class

Other Resources

CGdiObject Members

CGdiObject::CreateStockObject

Retrieves a handle to one of the predefined stock Windows GDI pens, brushes, or fonts, and attaches the GDI object to the **CGdiObject** object.

```
BOOL CreateStockObject(
   int nIndex
);
```

Parameters

nIndex

A constant specifying the type of stock object desired. See the parameter *fnObject* for GetStockObject in the *Platform SDK* for a description of appropriate values.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

Call this function with one of the derived classes that corresponds to the Windows GDI object type, such as **CPen** for a stock pen.

See Also

Reference

CGdiObject Class
Hierarchy Chart
CPen::CPen
CBrush::CBrush
CFont::CFont
CPalette::CPalette

Other Resources

CGdiObject Members

CGdiObject::DeleteObject

Deletes the attached Windows GDI object from memory by freeing all system storage associated with the Windows GDI object.

```
BOOL DeleteObject( );
```

Return Value

Nonzero if the GDI object was successfully deleted; otherwise 0.

Remarks

The storage associated with the **CGdiObject** object is not affected by this call. An application should not call **DeleteObject** on a **CGdiObject** object that is currently selected into a device context.

When a pattern brush is deleted, the bitmap associated with the brush is not deleted. The bitmap must be deleted independently.

See Also

Reference

CGdiObject Class
Hierarchy Chart
CGdiObject::Detach

Other Resources

CGdiObject Members

CGdiObject::DeleteTempMap

Called automatically by the **CWinApp** idle-time handler, **DeleteTempMap** deletes any temporary **CGdiObject** objects created by **FromHandle**.

```
static void PASCAL DeleteTempMap( );
```

Remarks

DeleteTempMap detaches the Windows GDI object attached to a temporary **CGdiObject** object before deleting the **CGdiObject** object.

Example

```
// The following example is correct:
CGdiObject::DeleteTempMap();

// The following example is incorrect.
// DeleteTempMap() is a static member and cannot be called
// within the scope of an instantiated CGdiObject object.
pGdiObject->DeleteTempMap();  // Causes compiler error
```

See Also

Reference

CGdiObject Class
Hierarchy Chart
CGdiObject::Detach
CGdiObject::FromHandle

Other Resources

CGdiObject Members

CGdiObject::Detach

Detaches a Windows GDI object from a **CGdiObject** object and returns a handle to the Windows GDI object.

```
HGDIOBJ Detach( );
```

Return Value

A **HANDLE** to the Windows GDI object detached; otherwise **NULL** if no GDI object is attached.

See Also

Reference

CGdiObject Class
Hierarchy Chart
CGdiObject::Attach

Other Resources

CGdiObject Members

CGdiObject::FromHandle

Returns a pointer to a **CGdiObject** object given a handle to a Windows GDI object.

```
static CGdiObject* PASCAL FromHandle(
   HGDIOBJ hObject
);
```

Parameters

hObject

A **HANDLE** to a Windows GDI object.

Return Value

A pointer to a **CGdiObject** that may be temporary or permanent.

Remarks

If a **CGdiObject** object is not already attached to the Windows GDI object, a temporary **CGdiObject** object is created and attached.

This temporary **CGdiObject** object is only valid until the next time the application has idle time in its event loop, at which time all temporary graphic objects are deleted. Another way of saying this is that the temporary object is only valid during the processing of one window message.

See Also

Reference

CGdiObject Class
Hierarchy Chart
CGdiObject::DeleteTempMap

Other Resources

CGdiObject Members

CGdiObject::GetObject

Fills a buffer with data that defines a specified object.

```
int GetObject(
   int nCount,
   LPVOID lpObject
) const;
```

Parameters

nCount

Specifies the number of bytes to copy into the *lpObject* buffer.

lpObject

Points to a user-supplied buffer that is to receive the information.

Return Value

The number of bytes retrieved; otherwise 0 if an error occurs.

Remarks

The function retrieves a data structure whose type depends on the type of graphic object, as shown by the following list:

Object	Buffer type
CPen	LOGPEN
CBrush	LOGBRUSH
CFont	LOGFONT
CBitmap	BITMAP
CPalette	**WORD**
CRgn	Not supported

If the object is a **CBitmap** object, **GetObject** returns only the width, height, and color format information of the bitmap. The actual bits can be retrieved by using CBitmap::GetBitmapBits.

If the object is a **CPalette** object, **GetObject** retrieves a **WORD** that specifies the number of entries in the palette. The function does not retrieve the LOGPALETTE structure that defines the palette. An application can get information on palette entries by calling CPalette::GetPaletteEntries.

See Also

Reference

CGdiObject Class
Hierarchy Chart
CBitmap::GetBitmapBits
CPalette::GetPaletteEntries

Other Resources

CGdiObject Members

CGdiObject::GetObjectType

Retrieves the type of the GDI object.

```
UINT GetObjectType( ) const;
```

Return Value

The type of the object, if successful; otherwise 0. The value can be one of the following:

- **OBJ_BITMAP** Bitmap
- **OBJ_BRUSH** Brush
- **OBJ_FONT** Font
- **OBJ_PAL** Palette
- **OBJ_PEN** Pen
- **OBJ_EXTPEN** Extended pen
- **OBJ_REGION** Region
- **OBJ_DC** Device context
- **OBJ_MEMDC** Memory device context
- **OBJ_METAFILE** Metafile
- **OBJ_METADC** Metafile device context
- **OBJ_ENHMETAFILE** Enhanced metafile
- **OBJ_ENHMETADC** Enhanced-metafile device context

See Also

Reference

CGdiObject Class
Hierarchy Chart
CGdiObject::GetObject
CDC::SelectObject

Other Resources

CGdiObject Members

CGdiObject::GetSafeHandle

Returns **m_hObject** unless **this** is **NULL**, in which case **NULL** is returned.

```
HGDIOBJ GetSafeHandle( ) const;
```

Return Value

A **HANDLE** to the attached Windows GDI object; otherwise **NULL** if no object is attached.

Remarks

This is part of the general handle interface paradigm and is useful when **NULL** is a valid or special value for a handle.

Example

See the example for CWnd::IsWindowEnabled.

See Also

Reference

CGdiObject Class

Hierarchy Chart

Other Resources

CGdiObject Members

CGdiObject::UnrealizeObject

Resets the origin of a brush or resets a logical palette.

```
BOOL UnrealizeObject( );
```

Return Value

Nonzero if successful; otherwise 0.

Remarks

While **UnrealizeObject** is a member function of the **CGdiObject** class, it should be invoked only on **CBrush** or **CPalette** objects.

For **CBrush** objects, **UnrealizeObject** directs the system to reset the origin of the given brush the next time it is selected into a device context. If the object is a **CPalette** object, **UnrealizeObject** directs the system to realize the palette as though it had not previously been realized. The next time the application calls the CDC::RealizePalette function for the specified palette, the system completely remaps the logical palette to the system palette.

The **UnrealizeObject** function should not be used with stock objects. The **UnrealizeObject** function must be called whenever a new brush origin is set (by means of the CDC::SetBrushOrg function). The **UnrealizeObject** function must not be called for the currently selected brush or currently selected palette of any display context.

See Also

Reference

CGdiObject Class
Hierarchy Chart
CDC::RealizePalette
CDC::SetBrushOrg

Other Resources

CGdiObject Members

CGdiObject Operators

For information about the operators in **CGdiObject**, see CGdiObject Members.

CGdiObject::operator !=

Determines if two GDI objects are logically not equal.

```
BOOL operator!=(
   const CGdiObject& obj
) const;
```

Parameters

obj

A pointer to an existing **CGdiObject**.

Remarks

Determines if a GDI object on the left side is not equal to a GDI object on the right side.

See Also

Reference

CGdiObject Class

Hierarchy Chart

CGdiObject::operator ==

Other Resources

CGdiObject Members

CGdiObject::operator ==

Determines if two GDI objects are logically equal.

```
BOOL operator==(
   const CGdiObject& obj
) const;
```

Parameters

obj

A reference to an existing **CGdiObject**.

Remarks

Determines if a GDI object on the left side is equal to a GDI object on the right side.

See Also

Reference

CGdiObject Class

Hierarchy Chart

CGdiObject::operator !=

Other Resources

CGdiObject Members

CGdiObject::operator HGDIOBJ

Retrieves a **HANDLE** to the attached Windows GDI object; otherwise **NULL** if no object is attached.

```
operator HGDIOBJ() const;
```

See Also

Reference

CGdiObject Class

Hierarchy Chart

CGdiObject::GetSafeHandle

Other Resources

CGdiObject Members

CGdiObject Data Members

For information about the data members in **CGdiObject**, see CGdiObject Members.

CGdiObject::m_hObject

A **HANDLE** containing the **HBITMAP**, **HRGN**, **HBRUSH**, **HPEN**, **HPALETTE**, or **HFONT** attached to this object.

```
HGDIOBJ m_hObject;
```

See Also

Reference

CGdiObject Class

Hierarchy Chart

Other Resources

CGdiObject Members

CGopherConnection Class

Manages your connection to a gopher Internet server.

Note
The classes **CGopherConnection**, **CGopherFile**, **CGopherFileFind**, **CGopherLocator** and their members have been deprecated because they do not work on the Windows XP platform, but they will continue to work on earlier platforms.

```
class CGopherConnection : public CInternetConnection
```

Remarks

The gopher service is one of three Internet services recognized by the MFC WinInet classes.

The class **CGopherConnection** contains a constructor and three additional member functions that manage the gopher service: OpenFile, CreateLocator, and GetAttribute.

To communicate with a gopher Internet server, you must first create an instance of CInternetSession, and then call CInternetSession::GetGopherConnection, which creates the **CGopherConnection** object and returns a pointer to it. You never create a **CGopherConnection** object directly.

To learn more about how **CGopherConnection** works with the other MFC Internet classes, see the article Internet Programming with WinInet. For more information about using the other two supported Internet services, FTP and HTTP see the classes CHttpConnection and CFtpConnection.

Requirements

Header: afxinet.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CInternetConnection Class
Hierarchy Chart
CFtpConnection Class
CHttpConnection Class
CInternetConnection Class
CGopherLocator Class
CGopherFile Class
CInternetSession Class

Other Resources

CGopherConnection Members

CGopherConnection Members

Base Class Members

CObject Members

CInternetConnection Members

Construction

CGopherConnection	Constructs a **CGopherConnection** object.

Operations

CreateLocator	Creates a CGopherLocator object to find files on a gopher server.
GetAttribute	Retrieves attribute information about the gopher object.
OpenFile	Opens a gopher file.

See Also

Reference

CGopherConnection Class
Hierarchy Chart
CFtpConnection Class
CHttpConnection Class
CGopherConnection Class
CGopherFileFind Class

CGopherConnection Member Functions

For information about the member functions in **CGopherConnection**, see CGopherConnection Members.

CGopherConnection::CGopherConnection

This member function is called to construct a **CGopherConnection** object.

```
CGopherConnection(
   CInternetSession* pSession,
   HINTERNET hConnected,
   LPCTSTR pstrServer,
   DWORD_PTR dwContext
);
CGopherConnection(
   CInternetSession* pSession,
   LPCTSTR pstrServer,
   LPCTSTR pstrUserName = NULL,
   LPCTSTR pstrPassword = NULL,
   DWORD_PTR dwContext = 0,
   INTERNET_PORT nPort = INTERNET_INVALID_PORT_NUMBER
);
```

Parameters

pSession

A pointer to the related CInternetSession object.

hConnected

The Windows handle of the current Internet session.

pstrServer

A pointer to a string containing the FTP server name.

dwContext

The context identifier for the operation. *dwContext* identifies the operation's status information returned by CInternetSession::OnStatusCallback. The default is set to 1; however, you can explicitly assign a specific context ID for the operation. The object and any work it does will be associated with that context ID.

pstrUserName

Pointer to a null-terminated string that specifies the name of the user to log in. If **NULL**, the default is anonymous.

pstrPassword

A pointer to a null-terminated string that specifies the password to use to log in. If both *pstrPassword* and *pstrUserName* are **NULL**, the default anonymous password is the user's email name. If *pstrPassword* is **NULL** (or an empty string) but *pstrUserName* is not **NULL**, a blank password is used. The following table describes the behavior for the four possible settings of *pstrUserName* and *pstrPassword*:

pstrUserName	***pstrPassword***	**Username sent to FTP server**	**Password sent to FTP server**
NULL or " "	**NULL** or " "	"anonymous"	User's email name
Non-**NULL** String	**NULL** or " "	*pstrUserName*	" "
NULL Non-**NULL** String	**ERROR**	**ERROR**	
Non-**NULL** String	Non-**NULL** String	*pstrUserName*	*pstrPassword*

nPort

A number that identifies the TCP/IP port to use on the server.

Remarks

You never create a **CGopherConnection** directly. Rather, call CInternetSession::GetGopherConnection, which creates a **CGopherConnection** object and returns a pointer to it.

See Also

Reference

CGopherConnection Class
Hierarchy Chart
CFtpConnection Class
CHttpConnection Class
CInternetConnection Class

Other Resources

CGopherConnection Members

CGopherConnection::CreateLocator

Call this member function to create a gopher locator to find or identify a file on a gopher server.

```
CGopherLocator CreateLocator(
   LPCTSTR pstrDisplayString,
   LPCTSTR pstrSelectorString,
   DWORD dwGopherType
);
static CGopherLocator CreateLocator(
   LPCTSTR pstrLocator
);
static CGopherLocator CreateLocator(
   LPCTSTR pstrServerName,
   LPCTSTR pstrDisplayString,
   LPCTSTR pstrSelectorString,
   DWORD dwGopherType,
   INTERNET_PORT nPort = INTERNET_INVALID_PORT_NUMBER
);
```

Parameters

pstrDisplayString

A pointer to a string containing the name of the gopher document or directory to be retrieved. If the *pstrDisplayString* parameter is **NULL**, the default directory for the gopher server is returned.

pstrSelectorString

A pointer to the selector string to be sent to the gopher server in order to retrieve an item. *pstrSelectorString* can be **NULL**.

dwGopherType

This specifies whether *pstrSelectorString* refers to a directory or document, and whether the request is gopher or gopher+. See the attributes for the structure GOPHER_FIND_DATA in the *Platform SDK*.

pstrLocator

A pointer to a string identifying the file to open. Generally, this string is returned from a call to CGopherFileFind::GetLocator.

pstrServerName

A pointer to a string containing the gopher server name.

nPort

The number identifying the Internet port for this connection.

Return Value

A CGopherLocator object.

Remarks

The static version of the member function requires you to specify a server, while the non-static version uses the server name from the connection object.

In order to retrieve information from a gopher server, an application must first get a gopher locator. The application must then treat the locator as an opaque token (that is, the application can use the locator but not directly manipulate or compare it). Normally, the application uses the locator for calls to the CGopherFileFind::FindFile member function to retrieve a specific piece of information.

See Also

Reference

CGopherConnection Class
Hierarchy Chart
CFtpConnection Class
CHttpConnection Class

CInternetConnection Class
CGopherLocator Class
CGopherFileFind Class

Other Resources

CGopherConnection Members

CGopherConnection::GetAttribute

Call this member function to retrieve specific attribute information about an item from the gopher server.

```
BOOL GetAttribute(
   CGopherLocator& refLocator
   CString strRequestedAttributes,
   CString& strResult,
);
```

Parameters

refLocator

A reference to a CGopherLocator object.

strRequestedAttributes

A space-delimited string specifying the names of the requested attributes.

strResult

A reference to a CString that receives the locator type.

Return Value

Nonzero if successful; otherwise 0. If the call fails, the Win32 function GetLastError may be called to determine the cause of the error.

See Also

Reference

CGopherConnection Class
Hierarchy Chart
CFtpConnection Class
CHttpConnection Class
CInternetConnection Class
CGopherLocator Class

Other Resources

CGopherConnection Members

CGopherConnection::OpenFile

Call this member function to open a file on a gopher server.

```
CGopherFile* OpenFile(
   CGopherLocator& refLocator,
   DWORD dwFlags = 0,
   LPCTSTR pstrView = NULL,
   DWORD_PTR dwContext = 1
);
```

Parameters

refLocator

A reference to a CGopherLocator object.

dwFlags

Any combination of INTERNET_FLAG_* flags. See CInternetSession::OpenUrl for further information on INTERNET_FLAG_* flags.

pstrView

A pointer to a file-view string. If several views of the file exist at the server, this parameter specifies which file view to open. If *pstrView* is **NULL**, the default file view is used.

dwContext

The context ID for the file being opened. See **Remarks** for more information about *dwContext*.

Return Value

A pointer to the CGopherFile object to be opened.

Remarks

Override the *dwContext* default to set the context identifier to a value of your choosing. The context identifier is associated with this specific operation of the **CGopherConnection** object created by its CInternetSession object. The value is returned to CInternetSession::OnStatusCallback to provide status on the operation with which it is identified. See the article Internet First Steps: WinInet for more information about the context identifier.

See Also

Reference

CGopherConnection Class
Hierarchy Chart
CFtpConnection Class
CHttpConnection Class
CInternetConnection Class
CGopherFile Class
CGopherLocator Class
CInternetSession Class

Other Resources

CGopherConnection Members

CGopherFile Class

Provides the functionality to find and read files on a gopher server.

Note
The classes **CGopherConnection**, **CGopherFile**, **CGopherFileFind**, **CGopherLocator** and their members have been deprecated because they do not work on the Windows XP platform, but they will continue to work on earlier platforms.

```
class CGopherFile : public CInternetFile
```

Remarks

The gopher service does not allow users to write data to a gopher file because this service functions mainly as a menu-driven interface for finding information. The **CGopherFile** member functions **Write**, **WriteString**, and **Flush** are not implemented for **CGopherFile**. Calling these functions on a **CGopherFile** object, returns a CNotSupportedException.

To learn more about how **CGopherFile** works with the other MFC Internet classes, see the article Internet Programming with WinInet.

Requirements

Header: afxinet.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CInternetFile Class
Hierarchy Chart
CInternetFile Class
CGopherLocator Class
CGopherFileFind Class
CGopherConnection Class

Other Resources

CGopherFile Members

CGopherFile Members

Base Class Members

CObject Members

CFileFind Members

Construction

CGopherFile	Constructs a **CGopherFile** object.

See Also

Reference

CGopherFile Class
Hierarchy Chart
CInternetFile Class
CGopherLocator Class
CGopherFileFind Class
CGopherConnection Class

CGopherFile Member Functions

For information about the member functions in **CGopherFile**, see CGopherFile Members.

CGopherFile::CGopherFile

This member function is called to construct a **CGopherFile** object.

```
CGopherFile(
   HINTERNET hFile,
   CGopherLocator& refLocator,
   CGopherConnection* pConnection
);
CGopherFile(
   HINTERNET hFile,
   HINTERNET hSession,
   LPCTSTR pstrLocator,
   DWORD dwLocLen,
   DWORD_PTR dwContext
);
```

Parameters

hFile

A handle to an **HINTERNET** file.

refLocator

A reference to a CGopherLocator object.

pConnection

A pointer to a CGopherConnection object.

hSession

A handle to the current Internet session.

pstrLocator

A pointer to a string used to locate the gopher server. See Gopher Sessions for more information about gopher locators.

dwLocLen

A DWORD containing the number of bytes in *pstrLocator*.

dwContext

A pointer to the context identifier of the file being opened.

Remarks

You need a **CGopherFile** object to read from a file during a gopher Internet session.

You never create a **CGopherFile** object directly. Instead, call CGopherConnection::OpenFile to open a file on a gopher server.

See Also

Reference

CGopherFile Class
Hierarchy Chart
CInternetFile Class
CGopherLocator Class
CGopherFileFind Class
CGopherConnection Class

Other Resources

CGopherFile Members

CGopherFileFind Class

Aids in Internet file searches of gopher servers.

Note
The classes **CGopherConnection**, **CGopherFile**, **CGopherFileFind**, **CGopherLocator** and their members have been deprecated because they do not work on the Windows XP platform, but they will continue to work on earlier platforms.

```
class CGopherFileFind : public CFileFind
```

Remarks

CGopherFileFind includes member functions that begin a search, locate a file, and return a file's URL.

Other MFC classes designed for Internet and local file searched include CFtpFileFind and CFileFind. Together with **CGopherFileFind**, these classes provide a seamless mechanism for the user to find specific files, regardless of the server protocol, file type, or location (either a local machine or a remote server.) Note that there is no MFC class for searching on HTTP servers because HTTP does not support the direct file manipulation required by searches.

Note
CGopherFileFind does not support the following member functions of its base class CFileFind:

- GetRoot
- GetFileName
- GetFilePath
- GetFileTitle
- GetFileURL

In addition, when used with **CGopherFileFind**, the **CFileFind** member function IsDots is always **FALSE**.

For more information about how to use **CGopherFileFind** and the other WinInet classes, see the article Internet Programming with WinInet.

Requirements

Header: afxinet.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CFileFind Class
Hierarchy Chart
CFtpFileFind Class
CFileFind Class
CInternetFile Class
CGopherFile Class
CHttpFile Class

Other Resources

CGopherFileFind Members

CGopherFileFind Members

Base Class Members

CObject Members

CFileFind Members

Construction

CGopherFileFind	Constructs a **CGopherFileFind** object.

Attributes

FindFile	Finds a file on a gopher server.
FindNextFile	Continues a file search from a previous call to FindFile.
GetCreationTime	Gets the time the specified file was created.
GetLastAccessTime	Gets the time the specified file was last accessed.
GetLastWriteTime	Gets the time the specified file was last written to.
IsDots	Tests for the current directory and parent directory markers while iterating through files.
GetLength	Gets the length of the found file, in bytes.
GetLocator	Get a **CGopherLocator** object.
GetScreenName	Gets the name of a gopher screen.

See Also

Reference

CFileFind Class
Hierarchy Chart
CFtpFileFind Class
CFileFind Class
CInternetFile Class
CGopherFile Class
CHttpFile Class

CGopherFileFind Member Functions

For information about the member functions in **CGopherFileFind**, see CGopherFileFind Members.

CGopherFileFind::CGopherFileFind

This member function is called to construct a **CGopherFileFind** object.

```
explicit CGopherFileFind(
   CGopherConnection* pConnection,
   DWORD_PTR dwContext = 1
);
```

Parameters

pConnection

A pointer to a CGopherConnection object.

dwContext

The context identifier for the operation. See **Remarks** for more information about *dwContext*.

Remarks

The default value for *dwContext* is sent by MFC to the **CGopherFileFind** object from the CInternetSession object that created the **CGopherFileFind** object. When you construct a **CGopherFileFind** object, you can override the default to set the context identifier to a value of your choosing. The context identifier is returned to CInternetSession::OnStatusCallback to provide status on the object with which it is identified. See the article Internet First Steps: WinInet for more information about the context identifier.

See Also

Reference

CGopherFileFind Class
Hierarchy Chart
CFtpFileFind Class
CFileFind Class

Other Resources

CGopherFileFind Members

CGopherFileFind::FindFile

Call this member function to find a gopher file.

```
virtual BOOL FindFile(
   CGopherLocator& refLocator,
   LPCTSTR pstrString,
   DWORD dwFlags = INTERNET_FLAG_RELOAD
);
virtual BOOL FindFile(
   LPCTSTR pstrString,
   DWORD dwFlags = INTERNET_FLAG_RELOAD
);
```

Parameters

refLocator

A reference to a CGopherLocator object.

pstrString

A pointer to a string containing the file name.

dwFlags

The flags describing how to handle this session. The valid flags are:

- INTERNET_FLAG_RELOAD Get the data from the remote server even if it is locally cached.
- INTERNET_FLAG_DONT_CACHE Do not cache the data, either locally or in any gateways.
- INTERNET_FLAG_SECURE Request secure transactions on the wire with Secure Sockets Layer or PCT. This flag is applicable to HTTP requests only.
- INTERNET_FLAG_USE_EXISTING If possible, reuse the existing connections to the server for new **FindFile** requests, instead of creating a new session for each request.

Return Value

Nonzero if successful; otherwise 0. To get extended error information, call the Win32 function GetLastError.

Remarks

After calling **FindFile** to retrieve the first gopher object, you can call FindNextFile to retrieve subsequent gopher files.

See Also

Reference

CGopherFileFind Class
Hierarchy Chart
CFileFind::FindFile

Other Resources

CGopherFileFind Members

CGopherFileFind::FindNextFile

Call this member function to continue a file search begun with a call to CGopherFileFind::FindFile.

```
virtual BOOL FindNextFile( );
```

Return Value

Nonzero if there are more files; zero if the file found is the last one in the directory or if an error occurred. To get extended error information, call the Win32 function GetLastError. If the file found is the last file in the directory, or if no matching files can be found, the **GetLastError** function returns ERROR_NO_MORE_FILES.

See Also

Reference

CGopherFileFind Class
Hierarchy Chart
CFileFind::FindNextFile

Other Resources

CGopherFileFind Members

CGopherFileFind::GetCreationTime

Gets the creation time for the current file.

```
virtual BOOL GetCreationTime(
   FILETIME* pTimeStamp
) const;
virtual BOOL GetCreationTime(
   CTime& refTime
) const;
```

Parameters

pTimeStamp

A pointer to a FILETIME structure containing the time the file was created.

refTime

A reference to a CTime object.

Return Value

Nonzero if successful; 0 if unsuccessful. **GetCreationTime** returns 0 only if FindNextFile has never been called on this **CGopherFileFind** object.

Remarks

You must call FindNextFile at least once before calling **GetCreationTime**.

Note

Not all file systems use the same semantics to implement the time stamp returned by this function. This function may return the same value returned by other time stamp functions if the underlying file system or server does not support keeping the time attribute. See the Win32_FIND_DATA structure for information about time formats. On some operating systems, the returned time is in the time zone local to the machine were the file is located. See the Win32 FileTimeToLocalFileTime API for more information.

See Also

Reference

CGopherFileFind Class

Hierarchy Chart

Other Resources

CGopherFileFind Members

CGopherFileFind::GetLastAccessTime

Gets the time the specified file was last accessed.

```
virtual BOOL GetLastAccessTime(
   CTime& refTime
) const;
virtual BOOL GetLastAccessTime(
   FILETIME* pTimeStamp
) const;
```

Parameters

refTime

A reference to a CTime object.

pTimeStamp

A pointer to a FILETIME structure containing the time the file was last accessed.

Return Value

Nonzero if successful; 0 if unsuccessful. **GetLastAccessTime** returns 0 only if FindNextFile has never been called on this **CGopherFileFind** object.

Remarks

You must call FindNextFile at least once before calling **GetLastAccessTime**.

Note

Not all file systems use the same semantics to implement the time stamp returned by this function. This function may return the same value returned by other time stamp functions if the underlying file system or server does not support keeping the time attribute. See the Win32_FIND_DATA structure for information about time formats. On some operating systems, the returned time is in the time zone local to the machine were the file is located. See the Win32 FileTimeToLocalFileTime API for more information.

See Also

Reference

CGopherFileFind Class

Hierarchy Chart

Other Resources

CGopherFileFind Members

CGopherFileFind::GetLastWriteTime

Gets the last time the file was changed.

```
virtual BOOL GetLastWriteTime(
   FILETIME* pTimeStamp
) const;
virtual BOOL GetLastWriteTime(
   CTime& refTime
) const;
```

Parameters

pTimeStamp

A pointer to a FILETIME structure containing the time the file was last written to.

refTime

A reference to a CTime object.

Return Value

Nonzero if successful; 0 if unsuccessful. **GetLastWriteTime** returns 0 only if FindNextFile has never been called on this **CGopherFileFind** object.

Remarks

You must call FindNextFile at least once before calling **GetLastWriteTime**.

Note

Not all file systems use the same semantics to implement the time stamp returned by this function. This function may return the same value returned by other time stamp functions if the underlying file system or server does not support keeping the time attribute. See the Win32_FIND_DATA structure for information about time formats. On some operating systems, the returned time is in the time zone local to the machine were the file is located. See the Win32 FileTimeToLocalFileTime API for more information.

See Also

Reference

CGopherFileFind Class

Hierarchy Chart

Other Resources

CGopherFileFind Members

CGopherFileFind::GetLength

Call this member function to get the length, in bytes, of the found file.

```
virtual ULONGLONG GetLength( ) const;
```

Return Value

The length, in bytes, of the found file.

Remarks

GetLength uses the Win32 structure WIN32_FIND_DATA to get the value of the file size in bytes.

Note
As of MFC 7.0, **GetLength** supports 64-bit integer types. Previously-existing code built with this newer version of the library may result in truncation warnings.

Example

See the example for CFile::GetLength (the base class implementation).

See Also

Reference

CGopherFileFind Class
Hierarchy Chart
CFileFind Class

Other Resources

CGopherFileFind Members

CGopherFileFind::GetLocator

Call this member function to get the CGopherLocator object that FindFile uses to find the gopher file.

```
CGopherLocator GetLocator( ) const;
```

Return Value

A **CGopherLocator** object.

See Also

Reference

CGopherFileFind Class
Hierarchy Chart
CGopherConnection::CreateLocator

Other Resources

CGopherFileFind Members

CGopherFileFind::GetScreenName

Call this member function to get the name of the gopher screen.

```
CString GetScreenName( ) const;
```

Return Value

The name of the gopher screen.

See Also

Reference

CGopherFileFind Class

Hierarchy Chart

Other Resources

CGopherFileFind Members

CGopherFileFind::IsDots

Tests for the current directory and parent directory markers while iterating through files.

```
virtual BOOL IsDots( ) const;
```

Return Value

Nonzero if the found file has the name "." or "..", which indicates that the found file is actually a directory. Otherwise 0.

Remarks

You must call FindNextFile at least once before calling **IsDots**.

See Also

Reference

CGopherFileFind Class

Hierarchy Chart

Other Resources

CGopherFileFind Members

CGopherLocator Class

Gets a gopher "locator" from a gopher server, determines the locator's type, and makes the locator available to CGopherFileFind.

Note
The classes **CGopherConnection**, **CGopherFile**, **CGopherFileFind**, **CGopherLocator** and their members have been deprecated because they do not work on the Windows XP platform, but they will continue to work on earlier platforms.

```
class CGopherLocator : public CObject
```

Remarks

An application must get a gopher server's locator before it can retrieve information from that server. Once it has the locator, it must treat the locator as an opaque token.

Each gopher locator has attributes that determine the type of file or server found. See GetLocatorType for a list of types of gopher locators.

An application normally uses the locator for calls to CGopherFileFind::FindFile to retrieve a specific piece of information.

To learn more about how **CGopherLocator** works with the other MFC Internet classes, see the article Internet Programming with WinInet.

Requirements

Header: afxinet.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

COBJECT Class
Hierarchy Chart
CGopherFileFind Class

Other Resources

CGopherLocator Members

CGopherLocator Members

Base Class Members

CObject Members

Construction

CGopherLocator	Constructs a **CGopherLocator** object.

Attributes

GetLocatorType	Parses a gopher locator and determines its attributes.

Operators

operator LPCTSTR	Directly accesses characters stored in a **CGopherLocator** object as a C-style string.

See Also

Reference

CGopherLocator Class

Hierarchy Chart

CGopherLocator Member Functions

For information about the member functions in **CGopherLocator**, see CGopherLocator Members.

CGopherLocator::CGopherLocator

This member function is called to create a **CGopherLocator** object.

```
CGopherLocator(
   const CGopherLocator& ref
);
```

Parameters

ref

A reference to a constant **CGopherLocator** object.

Remarks

You never create a **CGopherLocator** object directly. Instead, call CGopherConnection::CreateLocator to create and return a pointer to the **CGopherLocator** object.

See Also

Reference

CGopherLocator Class
Hierarchy Chart
CGopherFileFind Class
CGopherConnection Class

Other Resources

CGopherLocator Members

CGopherLocator::GetLocatorType

Call this member function to get the locator type.

```
BOOL GetLocatorType(
   DWORD& dwRef
) const;
```

Parameters

dwRef

A reference to a **DWORD** that will receive the locator type. See **Remarks** for a table of locator types.

Return Value

Nonzero if successful; otherwise 0. If the call fails, the Win32 function GetLastError may be called to determine the cause of the error.

Remarks

The possible types are as follows:

Value	Meaning
GOPHER_TYPE_TEXT_FILE	An ASCII text file.
GOPHER_TYPE_DIRECTORY	A directory of additional Gopher items.
GOPHER_TYPE_CSO	A CSO phone book server.
GOPHER_TYPE_ERROR	Indicates an error condition.
GOPHER_TYPE_MAC_BINHEX	A Macintosh file in BINHEX format.
GOPHER_TYPE_DOS_ARCHIVE	A DOS archive file.
GOPHER_TYPE_UNIX_UUENCODED	A UUENCODED file.
GOPHER_TYPE_INDEX_SERVER	An index server.
GOPHER_TYPE_TELNET	A Telnet Server.
GOPHER_TYPE_BINARY	A binary file.
GOPHER_TYPE_REDUNDANT	A duplicated server. The information contained within is a duplicate of the primary server. The primary server is the last directory entry that did not have a GOPHER_TYPE_REDUNDANT type.

GOPHER_TYPE_TN3270	A TN3270 server.
GOPHER_TYPE_GIF	A GIF graphics file.
GOPHER_TYPE_IMAGE	An image file.
GOPHER_TYPE_BITMAP	A bitmap file.
GOPHER_TYPE_MOVIE	A movie file.
GOPHER_TYPE_SOUND	A sound file.
GOPHER_TYPE_HTML	An HTML document.
GOPHER_TYPE_PDF	A PDF file.
GOPHER_TYPE_CALENDAR	A calendar file.
GOPHER_TYPE_INLINE	An inline file.
GOPHER_TYPE_UNKNOWN	The item type is unknown.
GOPHER_TYPE_ASK	An Ask+ item.
GOPHER_TYPE_GOPHER_PLUS	A Gopher+ item.

See Also

Reference

CGopherLocator Class
Hierarchy Chart
CGopherFileFind Class
CGopherConnection Class

Other Resources

CGopherLocator Members

CGopherLocator Operators

For information about the operators in **CGopherLocator**, see CGopherLocator Members.

CGopherLocator::operator LPCTSTR

This useful casting operator provides an efficient method to access the null-terminated C string contained in a **CGopherLocator** object.

```
operator LPCTSTR ( ) const;
```

Return Value

A character pointer to the string's data.

Remarks

No characters are copied; only a pointer is returned.

See Also

Reference

CGopherLocator Class

Hierarchy Chart

Other Resources

CGopherLocator Members

CHeaderCtrl Class

Provides the functionality of the Windows common header control.

```
class CHeaderCtrl : public CWnd
```

Remarks

A "header control" is a window usually positioned above columns of text or numbers. It contains a title for each column, and it can be divided into parts. The user can drag the dividers that separate the parts to set the width of each column.

This control (and therefore the **CHeaderCtrl** class) is available only to programs running under Windows 95/98 and Windows NT version 3.51 and later.

Functionality added for Windows 95/Internet Explorer 4.0 common controls includes the following:

- Header item custom ordering.
- Header item drag and drop, for reordering of header items. Use the **HDS_DRAGDROP** style when you create the **CHeaderCtrl** object.
- Header column text constantly viewable during column resizing. Use the **HDS_FULLDRAG** style when you create a **CHeaderCtrl** object.
- Header hot tracking, which highlights the header item when the pointer is paused over it. Use the **HDS_HOTTRACK** style when you create the **CHeaderCtrl** object.
- Image list support. Header items can contain images stored in a **CImageList** object, as well as text.

For more information on using **CHeaderCtrl**, see Controls and Using CHeaderCtrl.

Requirements

Header: afxcmn.h

See Also

Reference

CWnd Class
Hierarchy Chart
CTabCtrl Class
CListCtrl Class
CImageList Class

Other Resources

CHeaderCtrl Members

CHeaderCtrl Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Construction

CHeaderCtrl	Constructs a **CHeaderCtrl** object.
Create	Creates a header control and attaches it to a **CHeaderCtrl** object.
CreateEx	Creates a header control with the specified Windows extended styles and attaches it to a **CListCtrl** object.

Attributes

GetBitmapMargin	Retrieves the width of the margin of a bitmap in a header control.
GetImageList	Retrieves the handle of an image list used for drawing header items in a header control.
GetItem	Retrieves information about an item in a header control.
GetItemCount	Retrieves a count of the items in a header control.
GetItemRect	Retrieves the bounding rectangle for a given item in a header control.
GetOrderArray	Retrieves the left-to-right order of items in a header control.
OrderToIndex	Retrieves the index value for an item based on its order in the header control.
SetBitmapMargin	Sets the width of the margin of a bitmap in a header control.
SetImageList	Assigns an image list to a header control.
SetItem	Sets the attributes of the specified item in a header control.
SetOrderArray	Sets the left-to-right order of items in a header control.

Operations

ClearAllFilters	Clears all filters for a header control.
ClearFilter	Clears the filter for a header control.
CreateDragImage	Creates a transparent version of an item's image within a header control.
DeleteItem	Deletes an item from a header control.
EditFilter	Starts editing the specified filter of a header control.
InsertItem	Inserts a new item into a header control.
Layout	Retrieves the size and position of a header control within a given rectangle.

SetFilterChangeTimeout	Sets the timeout interval between the time a change takes place in the filter attributes and the posting of an **HDN_FILTERCHANGE** notification.
SetHotDivider	Changes the divider between header items to indicate a manual drag and drop of a header item.

Overridables

DrawItem	Draws the specified item of a header control.

See Also

Reference

CHeaderCtrl Class
Hierarchy Chart

CHeaderCtrl Member Functions

For information about the member functions in **CHeaderCtrl**, see CHeaderCtrl Members.

CHeaderCtrl::CHeaderCtrl

Constructs a **CHeaderCtrl** object.

```
CHeaderCtrl( );
```

Example

```
// Declare a local CHeaderCtrl object.
CHeaderCtrl  myHeaderCtrl;

// Declare a dynamic CHeaderCtrl object.
CHeaderCtrl* pmyHeaderCtrl = new CHeaderCtrl;
```

See Also

Reference

CHeaderCtrl Class
Hierarchy Chart
CHeaderCtrl::Create
CHeaderCtrl::CreateEx

Other Resources

CHeaderCtrl Members

CHeaderCtrl::ClearAllFilters

Clears all filters for a header control.

```
int ClearAllFilters( );
```

Return Value

The index of the filter control being cleared.

Remarks

This member function implements the behavior of the Win32 message HDM_CLEARFILTER with a column value of –1, as described in the *Platform SDK*.

Example

```
// The pointer to my header control.
extern CHeaderCtrl* pmyHeaderCtrl;
pmyHeaderCtrl->ClearAllFilters( );
```

See Also

Reference

CHeaderCtrl Class
Hierarchy Chart
CHeaderCtrl::ClearFilter
CHeaderCtrl::EditFilter
CHeaderCtrl::SetFilterChangeTimeout

Other Resources

CHeaderCtrl Members

CHeaderCtrl::ClearFilter

Clears the filter for a header control.

```
int ClearFilter(
   int nColumn
);
```

Parameters

nColumn

Column value indicating which filter to clear.

Return Value

The index of the filter control being cleared.

Remarks

This member function implements the behavior of the Win32 message HDM_CLEARFILTER, as described in the *Platform SDK*.

Example

```
// The pointer to my header control.
extern CHeaderCtrl* pmyHeaderCtrl;
int i = pmyHeaderCtrl->ClearFilter( 1 );
```

See Also

Reference

CHeaderCtrl Class
Hierarchy Chart
CHeaderCtrl::ClearAllFilters
CHeaderCtrl::EditFilter
CHeaderCtrl::SetFilterChangeTimeout

Other Resources

CHeaderCtrl Members

CHeaderCtrl::Create

Creates a header control and attaches it to a **CHeaderCtrl** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the header control's style. For a description of header control styles, see Header Control Styles in the *Platform SDK*.

rect

Specifies the header control's size and position. It can be either a CRect object or a RECT structure.

pParentWnd

Specifies the header control's parent window, usually a **CDialog**. It must not be **NULL**.

nID

Specifies the header control's ID.

Return Value

Nonzero if initialization was successful; otherwise zero.

Remarks

You construct a **CHeaderCtrl** object in two steps. First, call the constructor and then call **Create**, which creates the header control and attaches it to the **CHeaderCtrl** object.

In addition to the header control styles, you can use the following common control styles to determine how the header control positions and resizes itself (see Common Control Styles for more information):

- **CCS_BOTTOM** Causes the control to position itself at the bottom of the parent window's client area and sets the width to be the same as the parent window's width.
- **CCS_NODIVIDER** Prevents a two-pixel highlight from being drawn at the top of the control.
- **CCS_NOMOVEY** Causes the control to resize and move itself horizontally, but not vertically, in response to a **WM_SIZE** message. If the **CCS_NORESIZE** style is used, this style does not apply. Header controls have this style by default.
- **CCS_NOPARENTALIGN** Prevents the control from automatically moving to the top or bottom of the parent window. Instead, the control keeps its position within the parent window despite changes to the size of the parent window. If the **CCS_TOP** or **CCS_BOTTOM** style is also used, the height is adjusted to the default, but the position and width remain unchanged.
- **CCS_NORESIZE** Prevents the control from using the default width and height when setting its initial size or a new size. Instead, the control uses the width and height specified in the request for creation or sizing.
- **CCS_TOP** Causes the control to position itself at the top of the parent window's client area and sets the width to be the same as the parent window's width.

You can also apply the following window styles to a header control (see Window Styles for more information):

- **WS_CHILD** Creates a child window. Cannot be used with the **WS_POPUP** style.
- **WS_VISIBLE** Creates a window that is initially visible.
- **WS_DISABLED** Creates a window that is initially disabled.

- **WS_GROUP** Specifies the first control of a group of controls in which the user can move from one control to the next with the arrow keys. All controls defined with the **WS_GROUP** style after the first control belong to the same group. The next control with the **WS_GROUP** style ends the style group and starts the next group (that is, one group ends where the next begins).
- **WS_TABSTOP** Specifies one of any number of controls through which the user can move by using the TAB key. The TAB key moves the user to the next control specified by the **WS_TABSTOP** style.

If you want to use extended windows styles with your control, call CreateEx instead of **Create**.

Example

```
// pParentWnd is a pointer to the parent window.
extern CWnd* pParentWnd;
// The pointer to my header control.
extern CHeaderCtrl* pmyHeaderCtrl;

pmyHeaderCtrl->Create(WS_CHILD|WS_VISIBLE|HDS_HORZ,
   CRect(10, 10, 600, 50), pParentWnd, 1);
```

See Also

Reference

CHeaderCtrl Class
Hierarchy Chart
CHeaderCtrl::CHeaderCtrl

Other Resources

CHeaderCtrl Members

CHeaderCtrl::CreateEx

Creates a control (a child window) and associate it with the **CHeaderCtrl** object.

```
virtual BOOL CreateEx(
   DWORD dwExStyle,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwExStyle

Specifies the extended style of the control being created. For a list of extended Windows styles, see the *dwExStyle* parameter for CreateWindowEx in the *Platform SDK*.

dwStyle

The header control's style. For a description of header control styles, see Header Control Styles in the *Platform SDK*. See Create for a list of additional styles.

rect

A reference to a RECT structure describing the size and position of the window to be created, in client coordinates of *pParentWnd*.

pParentWnd

A pointer to the window that is the control's parent.

nID

The control's child-window ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Use **CreateEx** instead of **Create** to apply extended Windows styles, specified by the Windows extended style preface **WS_EX_**.

See Also

Reference

CHeaderCtrl Class
Hierarchy Chart
CHeaderCtrl::CHeaderCtrl

Other Resources

CHeaderCtrl Members

CHeaderCtrl::CreateDragImage

Creates a transparent version of an item's image within a header control.

```
CImageList* CreateDragImage(
   int nIndex
);
```

Parameters

nIndex

The zero-based index of the item within the header control. The image assigned to this item is the basis for the transparent image.

Return Value

A pointer to a CImageList object if successful; otherwise **NULL**. The returned list contains only one image.

Remarks

This member function implements the behavior of the Win32 message HDM_CREATEDRAGIMAGE, as described in the *Platform SDK*. It is provided to support header item drag and drop.

The **CImageList** object to which the returned pointer points is a temporary object and is deleted in the next idle-time processing.

See Also

Reference

CHeaderCtrl Class
Hierarchy Chart
CHeaderCtrl::GetImageList
CHeaderCtrl::SetImageList

Other Resources

CHeaderCtrl Members

CHeaderCtrl::DeleteItem

Deletes an item from a header control.

```
BOOL DeleteItem(
   int nPos
);
```

Parameters

nPos

Specifies the zero-based index of the item to delete.

Return Value

Nonzero if successful; otherwise 0.

Example

```
// The pointer to my header control.
extern CHeaderCtrl* pmyHeaderCtrl;

int nCount = pmyHeaderCtrl->GetItemCount();

// Delete all of the items.
for (int i=0;i < nCount;i++)
{
   pmyHeaderCtrl->DeleteItem(0);
}
```

See Also

Reference

CHeaderCtrl Class

Hierarchy Chart

CHeaderCtrl::InsertItem

Other Resources

CHeaderCtrl Members

CHeaderCtrl::DrawItem

Called by the framework when a visual aspect of an owner-draw header control changes.

```
virtual void DrawItem(
   LPDRAWITEMSTRUCT lpDrawItemStruct
);
```

Parameters

lpDrawItemStruct

A pointer to a DRAWITEMSTRUCT structure describing the item to be painted.

Remarks

The **itemAction** member of the **DRAWITEMSTRUCT** structure defines the drawing action that is to be performed.

By default, this member function does nothing. Override this member function to implement drawing for an owner-draw **CHeaderCtrl** object.

The application should restore all graphics device interface (GDI) objects selected for the display context supplied in *lpDrawItemStruct* before this member function terminates.

Example

```
// NOTE: CMyHeaderCtrl is a class derived from CHeaderCtrl.
// The CMyHeaderCtrl object was created as follows:
//
//   CMyHeaderCtrl myHeader;
//   myHeader.Create(WS_CHILD|WS_VISIBLE|HDS_HORZ,
//       CRect(10, 10, 600, 50), pParentWnd, 1);


// This example implements the DrawItem method for a
// CHeaderCtrl-derived class that draws every item as a
// 3D button using the text color red.
void CMyHeaderCtrl::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
   // This code only works with header controls.
   ASSERT(lpDrawItemStruct->CtlType == ODT_HEADER);

   HDITEM hdi;
   TCHAR  lpBuffer[256];

   hdi.mask = HDI_TEXT;
   hdi.pszText = lpBuffer;
   hdi.cchTextMax = 256;

   GetItem(lpDrawItemStruct->itemID, &hdi);

   // Draw the button frame.
   ::DrawFrameControl(lpDrawItemStruct->hDC,
      &lpDrawItemStruct->rcItem, DFC_BUTTON, DFCS_BUTTONPUSH);

   // Draw the items text using the text color red.
   COLORREF crOldColor = ::SetTextColor(lpDrawItemStruct->hDC,
      RGB(255,0,0));
   ::DrawText(lpDrawItemStruct->hDC, lpBuffer, strlen(lpBuffer),
      &lpDrawItemStruct->rcItem, DT_SINGLELINE|DT_VCENTER|DT_CENTER);
   ::SetTextColor(lpDrawItemStruct->hDC, crOldColor);
}
```

See Also

Reference

CHeaderCtrl Class

Hierarchy Chart
CWnd::OnDrawItem
Other Resources
CHeaderCtrl Members

CHeaderCtrl::EditFilter

Starts editing the specified filter of a header control.

```
int EditFilter(
   int nColumn,
   BOOL bDiscardChanges
);
```

Parameters

nColumn

The column to edit.

bDiscardChanges

See *fDiscardChanges* in HDM_EDITFILTER in the *Platform SDK*.

Return Value

The index of the filter being edited.

Remarks

This member function implements the behavior of the Win32 message HDM_EDITFILTER, as described in the *Platform SDK*.

Example

```
// The pointer to my header control.
extern CHeaderCtrl* pmyHeaderCtrl;
int i = pmyHeaderCtrl->EditFilter( 1, TRUE );
```

See Also

Reference

CHeaderCtrl Class
Hierarchy Chart
CHeaderCtrl::ClearAllFilters
CHeaderCtrl::ClearFilter
CHeaderCtrl::SetFilterChangeTimeout

Other Resources

CHeaderCtrl Members

CHeaderCtrl::GetBitmapMargin

Retrieves the width of the margin of a bitmap in a header control.

```
int GetBitmapMargin( ) const;
```

Return Value

The width of the bitmap margin in pixels.

Remarks

This member function implements the behavior of the Win32 message HDM_GETBITMAPMARGIN, as described in the *Platform SDK*.

Example

```
// The pointer to my header control.
extern CHeaderCtrl* pmyHeaderCtrl;
int margin = pmyHeaderCtrl->GetBitmapMargin();
```

See Also

Reference

CHeaderCtrl Class
Hierarchy Chart
CHeaderCtrl::SetBitmapMargin

Other Resources

CHeaderCtrl Members

CHeaderCtrl::GetImageList

Retrieves the handle of an image list used for drawing header items in a header control.

```
CImageList* GetImageList( ) const;
```

Return Value

A pointer to a CImageList object.

Remarks

This member function implements the behavior of the Win32 message HDM_GETIMAGELIST, as described in the *Platform SDK*. The **CImageList** object to which the returned pointer points is a temporary object and is deleted in the next idle-time processing.

Example

```
// The pointer to my header control.
extern CHeaderCtrl* pmyHeaderCtrl;
// The new image list of the header control.
extern CImageList* pmyImageList;

ASSERT(pmyHeaderCtrl->GetImageList() == NULL);

pmyHeaderCtrl->SetImageList(pmyImageList);
ASSERT(pmyHeaderCtrl->GetImageList() == pmyImageList);
```

See Also

Reference

CHeaderCtrl Class
Hierarchy Chart
CHeaderCtrl::SetImageList
CHeaderCtrl::CreateDragImage

Other Resources

CHeaderCtrl Members

CHeaderCtrl::GetItem

Retrieves information about a header control item.

```
BOOL GetItem(
   int nPos,
   HDITEM* pHeaderItem
) const;
```

Parameters

nPos

Specifies the zero-based index of the item to retrieve.

pHeaderItem

Pointer to an HDITEM structure that receives the new item. This structure is used with the **InsertItem** and **SetItem** member functions. Any flags set in the **mask** element ensure that values in the corresponding elements are properly filled in upon return. If the **mask** element is set to zero, values in the other structure elements are meaningless.

Return Value

Nonzero if successful; otherwise 0.

Example

```
// The pointer to my header control.
extern CHeaderCtrl* pmyHeaderCtrl;
// External strings.
extern LPCTSTR lpszmyString, lpszmyString2;

{
    // ...
    // Find the item whose text matches lpszmyString, and
    // replace it with lpszmyString2.
    int    i, nCount = pmyHeaderCtrl->GetItemCount();
    HDITEM hdi;
    enum   { sizeOfBuffer = 256 };
    TCHAR  lpBuffer[sizeOfBuffer];
    bool   fFound = false;

    hdi.mask = HDI_TEXT;
    hdi.pszText = lpBuffer;
    hdi.cchTextMax = sizeOfBuffer;

    for (i=0;!fFound && (i < nCount);i++)
    {
        pmyHeaderCtrl->GetItem(i, &hdi);

        if (strcmp(hdi.pszText, lpszmyString) == 0)
        {
            strcpy_s(hdi.pszText, sizeOfBuffer, lpszmyString2);
            pmyHeaderCtrl->SetItem(i, &hdi);
            fFound = false;
        }
    }

    // ...
}
```

See Also

Reference

CHeaderCtrl Class

Hierarchy Chart

CHeaderCtrl::SetItem

Other Resources

CHeaderCtrl Members

CHeaderCtrl::GetItemCount

Retrieves a count of the items in a header control.

```
int GetItemCount( ) const;
```

Return Value

Number of header control items if successful; otherwise – 1.

Example

See the example for CHeaderCtrl::DeleteItem.

See Also

Reference

CHeaderCtrl Class
Hierarchy Chart
CHeaderCtrl::GetItem
CHeaderCtrl::SetItem

Other Resources

CHeaderCtrl Members

CHeaderCtrl::GetItemRect

Retrieves the bounding rectangle for a given item in a header control.

```
BOOL GetItemRect(
   int nIndex,
   LPRECT lpRect
) const;
```

Parameters

nIndex

The zero-based index of the header control item.

lpRect

A pointer to the address of a RECT structure that receives the bounding rectangle information.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function implements the behavior of the Win32 message HDM_GETITEMRECT, as described in the *Platform SDK*.

See Also

Reference

CHeaderCtrl Class

Hierarchy Chart

Other Resources

CHeaderCtrl Members

CHeaderCtrl::GetOrderArray

Retrieves the left-to-right order of items in a header control.

```
BOOL GetOrderArray(
   LPINT piArray,
   int iCount
);
```

Parameters

piArray

A pointer to the address of a buffer that receives the index values of the items in the header control, in the order in which they appear from left to right.

iCount

The number of header control items. Must be non-negative.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function implements the behavior of the Win32 message HDM_GETORDERARRAY, as described in the *Platform SDK*. It is provided to support header item ordering.

Example

```
// The pointer to my header control.
extern CHeaderCtrl* pmyHeaderCtrl;

// Reverse the order of the items in the header control.
// (i.e. make the first item the last one, the last item
// the first one, and so on ...).
int   nCount = pmyHeaderCtrl->GetItemCount();
LPINT pnOrder = (LPINT) malloc(nCount*sizeof(int));
ASSERT(pnOrder != NULL);

pmyHeaderCtrl->GetOrderArray(pnOrder, nCount);

int i, j, nTemp;
for (i=0,j=nCount-1;i < j;i++,j--)
{
   nTemp = pnOrder[i];
   pnOrder[i] = pnOrder[j];
   pnOrder[j] = nTemp;
}

pmyHeaderCtrl->SetOrderArray(nCount, pnOrder);
free(pnOrder);
```

See Also

Reference

CHeaderCtrl Class
Hierarchy Chart
CHeaderCtrl::SetOrderArray
CHeaderCtrl::OrderToIndex

Other Resources

CHeaderCtrl Members

CHeaderCtrl::InsertItem

Inserts a new item into a header control at the specified index.

```
int InsertItem(
   int nPos,
   HDITEM* phdi
);
```

Parameters

nPos

The zero-based index of the item to be inserted. If the value is zero, the item is inserted at the beginning of the header control. If the value is greater than the maximum value, the item is inserted at the end of the header control.

phdi

Pointer to an HDITEM structure that contains information about the item to be inserted.

Return Value

Index of the new item if successful; otherwise – 1.

Example

```
// The pointer to my header control.
extern CHeaderCtrl* pmyHeaderCtrl;

CString str;
HDITEM  hdi;

hdi.mask = HDI_TEXT | HDI_WIDTH | HDI_FORMAT;
hdi.cxy = 100; // Make all columns 100 pixels wide.
hdi.fmt = HDF_STRING | HDF_CENTER;

// Insert 6 columns in the header control.
for (int i=0;i < 6;i++)
{
   str.Format(TEXT("column %d"), i);
   hdi.pszText = str.GetBuffer(0);

   pmyHeaderCtrl->InsertItem(i, &hdi);
}
```

See Also

Reference

CHeaderCtrl Class
Hierarchy Chart
CHeaderCtrl::DeleteItem
CHeaderCtrl::GetItem

Other Resources

CHeaderCtrl Members

CHeaderCtrl::Layout

Retrieves the size and position of a header control within a given rectangle.

```
BOOL Layout(
   HDLAYOUT* pHeaderLayout
);
```

Parameters

pHeaderLayout

Pointer to an HDLAYOUT structure, which contains information used to set the size and position of a header control.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This function is used to determine the appropriate dimensions for a new header control that is to occupy the given rectangle.

Example

```
// The pointer to my header control.
extern CHeaderCtrl* pmyHeaderCtrl;
HDLAYOUT  hdl;
WINDOWPOS wpos;
RECT      rc;

// Reposition the header control so that it is placed at
// the top of its parent window's client area.
pmyHeaderCtrl->GetParent()->GetClientRect(&rc);

hdl.prc = &rc;
hdl.pwpos = &wpos;
pmyHeaderCtrl->Layout(&hdl);

pmyHeaderCtrl->SetWindowPos(
   CWnd::FromHandle(wpos.hwndInsertAfter),
   wpos.x,
   wpos.y,
   wpos.cx,
   wpos.cy,
   wpos.flags | SWP_SHOWWINDOW);
```

See Also

Reference

CHeaderCtrl Class

Hierarchy Chart

Other Resources

CHeaderCtrl Members

CHeaderCtrl::OrderToIndex

Retrieves the index value for an item based on its order in the header control.

```
int OrderToIndex(
   int nOrder
) const;
```

Parameters

nOrder

The zero-based order that the item appears in the header control, from left to right.

Return Value

The index of the item, based on its order in the header control. The index counts from left to right, beginning with 0.

Remarks

This member function implements the behavior of the Win32 macro HDM_ORDERTOINDEX, as described in the *Platform SDK*. It is provided to support header item ordering.

See Also

Reference

CHeaderCtrl Class
Hierarchy Chart
CHeaderCtrl::GetOrderArray
CHeaderCtrl::SetOrderArray

Other Resources

CHeaderCtrl Members

CHeaderCtrl::SetBitmapMargin

Sets the width of the margin of a bitmap in a header control.

```
int SetBitmapMargin(
   int nWidth
);
```

Parameters

nWidth

Width, specified in pixels, of the margin that surrounds a bitmap within an existing header control.

Return Value

The width of the bitmap margin in pixels.

Remarks

This member function implements the behavior of the Win32 message HDM_SETBITMAPMARGIN, as described in the *Platform SDK*.

Example

```
// The pointer to my header control.
extern CHeaderCtrl* pmyHeaderCtrl;
int margin = pmyHeaderCtrl->SetBitmapMargin( 15 );
```

See Also

Reference

CHeaderCtrl Class
Hierarchy Chart
CHeaderCtrl::GetBitmapMargin

Other Resources

CHeaderCtrl Members

CHeaderCtrl::SetFilterChangeTimeout

Sets the timeout interval between the time a change takes place in the filter attributes and the posting of an HDN_FILTERCHANGE notification.

```
int SetFilterChangeTimeout(
   DWORD dwTimeOut
);
```

Parameters

dwTimeOut

Timeout value, in milliseconds.

Return Value

The index of the filter control being modified.

Remarks

This member function implements the behavior of the Win32 message HDM_SETFILTERCHANGETIMEOUT, as described in the *Platform SDK*.

Example

```
// The pointer to my header control.
extern CHeaderCtrl* pmyHeaderCtrl;
int i = pmyHeaderCtrl->SetFilterChangeTimeout( 15 );
```

See Also

Reference

CHeaderCtrl Class
Hierarchy Chart
CHeaderCtrl::ClearAllFilters
CHeaderCtrl::ClearFilter
CHeaderCtrl::EditFilter

Other Resources

CHeaderCtrl Members

CHeaderCtrl::SetHotDivider

Changes the divider between header items to indicate a manual drag and drop of a header item.

```
int SetHotDivider(
   CPoint pt
);
int SetHotDivider(
   int nIndex
);
```

Parameters

pt

The position of the pointer. The header control highlights the appropriate divider based on the pointer's position.

nIndex

The index of the highlighted divider.

Return Value

The index of the highlighted divider.

Remarks

This member function implements the behavior of the Win32 message HDM_SETHOTDIVIDER, as described in the *Platform SDK*. It is provided to support header item drag and drop.

Example

```
// The pointer to my header control.
extern CHeaderCtrl* pmyHeaderCtrl;
// The point where the mouse cursor is at.
extern CPoint myPoint;

pmyHeaderCtrl->SetHotDivider(myPoint);
```

See Also

Reference

CHeaderCtrl Class

Hierarchy Chart

Other Resources

CHeaderCtrl Members

CHeaderCtrl::SetImageList

Assigns an image list to a header control.

```
CImageList* SetImageList(
   CImageList* pImageList
);
```

Parameters

pImageList

A pointer to a **CImageList** object containing the image list to be assigned to the header control.

Return Value

A pointer to the CImageList object previously assigned to the header control.

Remarks

This member function implements the behavior of the Win32 message HDM_SETIMAGELIST, as described in the *Platform SDK*. The **CImageList** object to which the returned pointer points is a temporary object and is deleted in the next idle-time processing.

Example

See the example for CHeaderCtrl::GetImageList.

See Also

Reference

CHeaderCtrl Class
Hierarchy Chart
CHeaderCtrl::GetImageList
CHeaderCtrl::CreateDragImage

Other Resources

CHeaderCtrl Members

CHeaderCtrl::SetItem

Sets the attributes of the specified item in a header control.

```
BOOL SetItem(
   int nPos,
   HDITEM* pHeaderItem
);
```

Parameters

nPos

The zero-based index of the item to be manipulated.

pHeaderItem

Pointer to an HDITEM structure that contains information about the new item.

Return Value

Nonzero if successful; otherwise 0.

Example

See the example for CHeaderCtrl::GetItem.

See Also

Reference

CHeaderCtrl Class
Hierarchy Chart
CHeaderCtrl::GetItem
CHeaderCtrl::GetItemCount

Other Resources

CHeaderCtrl Members

CHeaderCtrl::SetOrderArray

Sets the left-to-right order of items in a header control.

```
BOOL SetOrderArray(
   int iCount,
   LPINT piArray
);
```

Parameters

iCount

The number of header control items.

piArray

A pointer to the address of a buffer that receives the index values of the items in the header control, in the order in which they appear from left to right.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function implements the behavior of the Win32 macro HDM_SETORDERARRAY, as described in the *Platform SDK*. It is provided to support header item ordering.

Example

See the example for CHeaderCtrl::GetOrderArray.

See Also

Reference

CHeaderCtrl Class
Hierarchy Chart
CHeaderCtrl::GetOrderArray
CHeaderCtrl::OrderToIndex

Other Resources

CHeaderCtrl Members

CHotKeyCtrl Class

Provides the functionality of the Windows common hot key control.

```
class CHotKeyCtrl : public CWnd
```

Remarks

A "hot key control" is a window that enables the user to create a hot key. A "hot key" is a key combination that the user can press to perform an action quickly. (For example, a user can create a hot key that activates a given window and brings it to the top of the Z order.) The hot key control displays the user's choices and ensures that the user selects a valid key combination.

This control (and therefore the **CHotKeyCtrl** class) is available only to programs running under Windows 95/98 and Windows NT version 3.51 and later.

When the user has chosen a key combination, the application can retrieve the specified key combination from the control and use the **WM_SETHOTKEY** message to set up the hot key in the system. Whenever the user presses the hot key thereafter, from any part of the system, the window specified in the **WM_SETHOTKEY** message receives a **WM_SYSCOMMAND** message specifying **SC_HOTKEY**. This message activates the window that receives it. The hot key remains valid until the application that called **WM_SETHOTKEY** exits.

This mechanism is different from the hot key support that depends on the **WM_HOTKEY** message and the Windows RegisterHotKey and UnregisterHotKey functions.

For more information on using **CHotKeyCtrl**, see Controls and Using CHotKeyCtrl.

Requirements

Header: afxcmn.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CWnd Class

Hierarchy Chart

Other Resources

CHotKeyCtrl Members

CHotKeyCtrl Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Construction

CHotKeyCtrl	Constructs a **CHotKeyCtrl** object.
Create	Creates a hot key control and attaches it to a **CHotKeyCtrl** object.
CreateEx	Creates a hot key control with the specified Windows extended styles and attaches it to a **CHotKeyCtrl** object.

Attributes

GetHotKey	Retrieves the virtual key code and modifier flags of a hot key from a hot key control.
GetHotKeyName	Retrieves the key name, in the local character set, assigned to a hot key.
GetKeyName	Retrieves the key name, in the local character set, assigned to the specified virtual key code.
SetHotKey	Sets the hot key combination for a hot key control.

Operations

SetRules	Defines the invalid combinations and the default modifier combination for a hot key control.

See Also

Reference

CHotKeyCtrl Class

Hierarchy Chart

CHotKeyCtrl Member Functions

For information about the member functions in **CHotKeyCtrl**, see CHotKeyCtrl Members.

CHotKeyCtrl::CHotKeyCtrl

Constructs a **CHotKeyCtrl** object.

```
CHotKeyCtrl( );
```

See Also

Reference

CHotKeyCtrl Class
Hierarchy Chart
CHotKeyCtrl::CreateEx
CHotKeyCtrl::Create

Other Resources

CHotKeyCtrl Members

CHotKeyCtrl::Create

Creates a hot key control and attaches it to a **CHotKeyCtrl** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the hot key control's style. Apply any combination of control styles. See Common Control Styles in the *Platform SDK* for more information.

rect

Specifies the hot key control's size and position. It can be either a CRect object or a RECT structure.

pParentWnd

Specifies the hot key control's parent window, usually a CDialog. It must not be **NULL**.

nID

Specifies the hot key control's ID.

Return Value

Nonzero, if initialization was successful; otherwise 0.

Remarks

You construct a **CHotKeyCtrl** object in two steps. First, call the constructor and then call **Create**, which creates the hot key control and attaches it to the **CHotKeyCtrl** object.

If you want to use extended windows styles with your control, call CreateEx instead of **Create**.

See Also

Reference

CHotKeyCtrl Class
Hierarchy Chart
CHotKeyCtrl::CHotKeyCtrl

Other Resources

CHotKeyCtrl Members

CHotKeyCtrl::CreateEx

Call this function to create a control (a child window) and associate it with the **CHotKeyCtrl** object.

```
virtual BOOL CreateEx(
   DWORD dwExStyle,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwExStyle

Specifies the extended style of the control being created. For a list of extended Windows styles, see the *dwExStyle* parameter for CreateWindowEx in the *Platform SDK*.

dwStyle

Specifies the hot key control's style. Apply any combination of control styles. For more information, see Common Control Styles in the *Platform SDK*.

rect

A reference to a RECT structure describing the size and position of the window to be created, in client coordinates of *pParentWnd*.

pParentWnd

A pointer to the window that is the control's parent.

nID

The control's child-window ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Use **CreateEx** instead of Create to apply extended Windows styles, specified by the Windows extended style preface **WS_EX_**.

See Also

Reference

CHotKeyCtrl Class
Hierarchy Chart
CHotKeyCtrl::CHotKeyCtrl

Other Resources

CHotKeyCtrl Members

CHotKeyCtrl::GetHotKey

Call this function to retrieve the virtual key code and modifier flags of a hot key from a hot key control.

```
DWORD GetHotKey( ) const;
void GetHotKey(
   WORD &wVirtualKeyCode,
   WORD &wModifiers
) const;
```

Parameters

wVirtualKeyCode

Virtual key code of the hot key. For a list of standard virtual key codes, see Winuser.h.

wModifiers

Modifier flags indicating the keys that, when used in combination with *wVirtualKeyCode*, define a hot key combination.

Return Value

In the first usage above, a **DWORD** containing the virtual key code and modifier flags. The low-order word is the virtual key code, and the high-order word is the modifier flags. The HIWORD and LOWORD of this 32-bit value can be used as the parameters in the **SetHotKey** member function.

Remarks

The modifier flags can be a combination of the following values:

- **HOTKEYF_ALT** ALT key
- **HOTKEYF_CONTROL** CTRL key
- **HOTKEYF_EXT** Extended key
- **HOTKEYF_SHIFT** SHIFT key

See Also

Reference

CHotKeyCtrl Class
Hierarchy Chart
CHotKeyCtrl::SetHotKey

Other Resources

CHotKeyCtrl Members

CHotKeyCtrl::GetHotKeyName

Call this member function to get the localized name of the hot key.

```
CString GetHotKeyName( ) const;
```

Return Value

The localized name of the currently selected hot key. If there is no selected hot key, **GetHotKeyName** returns an empty string.

Remarks

The name that this member function returns comes from the keyboard driver. You can install a non-localized keyboard driver in a localized version of Windows, and vice versa.

See Also

Reference

CHotKeyCtrl Class
Hierarchy Chart
CHotKeyCtrl::GetKeyName

Other Resources

CHotKeyCtrl Members

CHotKeyCtrl::GetKeyName

Call this member function to get the localized name of the key assigned to a specified virtual key code.

```
static CString GetKeyName(
   UINT vk,
   BOOL fExtended
);
```

Parameters

vk

The virtual key code.

fExtended

If the virtual key code is an extended key, **TRUE**; otherwise **FALSE**.

Return Value

The localized name of the key specified by the *vk* parameter. If the key has no mapped name, **GetKeyName** returns an empty string.

Remarks

The key name that this function returns comes from the keyboard driver, so you can install a non-localized keyboard driver in a localized version of Windows, and vice versa.

Example

```
CString str;

str = CHotKeyCtrl::GetKeyName(VK_CONTROL, FALSE);

// str is now "Ctrl", or the localized equivalent.
```

See Also

Reference

CHotKeyCtrl Class
Hierarchy Chart
WM_KEYDOWN
GetKeyNameText
CHotKeyCtrl::GetHotKeyName

Other Resources

CHotKeyCtrl Members

CHotKeyCtrl::SetHotKey

Call this function to set the hot key combination for a hot key control.

```
void SetHotKey(
   WORD wVirtualKeyCode,
   WORD wModifiers
);
```

Parameters

wVirtualKeyCode

Virtual key code of the hot key. For a list of standard virtual key codes, see Winuser.h

wModifiers

Modifier flags indicating the keys that, when used in combination with *wVirtualKeyCode*, define a hot key combination. For more information on the modifier flags, see GetHotKey.

See Also

Reference

CHotKeyCtrl Class
Hierarchy Chart
CHotKeyCtrl::GetHotKey

Other Resources

CHotKeyCtrl Members

CHotKeyCtrl::SetRules

Call this function to define the invalid combinations and the default modifier combination for a hot key control.

```
void SetRules(
   WORD wInvalidComb,
   WORD wModifiers
);
```

Parameters

wInvalidComb

Array of flags that specifies invalid key combinations. It can be a combination of the following values:

- **HKCOMB_A** ALT
- **HKCOMB_C** CTRL
- **HKCOMB_CA** CTRL+ALT
- **HKCOMB_NONE** Unmodified keys
- **HKCOMB_S** SHIFT
- **HKCOMB_SA** SHIFT+ALT
- **HKCOMB_SC** SHIFT+CTRL
- **HKCOMB_SCA** SHIFT+CTRL+ALT

wModifiers

Array of flags that specifies the key combination to use when the user enters an invalid combination. For more information on the modifier flags, see GetHotKey.

Remarks

When a user enters an invalid key combination, as defined by flags specified in *wInvalidComb*, the system uses the OR operator to combine the keys entered by the user with the flags specified in *wModifiers*. The resulting key combination is converted into a string and then displayed in the hot key control.

See Also

Reference

CHotKeyCtrl Class
Hierarchy Chart
CHotKeyCtrl::GetHotKey
CHotKeyCtrl::SetHotKey

Other Resources

CHotKeyCtrl Members

CHtmlEditCtrl Class

Provides the functionality of the WebBrowser ActiveX control in an MFC window.

```
class CHtmlEditCtrl: public CWnd,
   public CHtmlEditCtrlBase< CHtmlEditCtrl >
```

Remarks

The hosted WebBrowser control is automatically put into edit mode after it is created.

Requirements

Header: afxhtml.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart

Other Resources

CHtmlEditCtrl Members

CHtmlEditCtrl Members

Member Functions

CHtmlEditCtrl	Constructs a **CHtmlEditCtrl** object.
Create	Creates a WebBrowser ActiveX control and attaches it to the **CHtmlEditCtrl** object. This function automatically puts the WebBrowser ActiveX control into edit mode.
GetDHtmlDocument	Retrieves the IHTMLDocument2 interface on the document currently loaded in the contained WebBrowser control.
GetStartDocument	Retrieves the URL to a default document to load in the contained WebBrowser control.

See Also

Reference

CHtmlEditCtrl Class

CHtmlEditCtrl Member Functions

For information about the member functions in **CHtmlEditCtrl**, see CHtmlEditCtrl Members.

CHtmlEditCtrl::CHtmlEditCtrl

Constructs a **CHtmlEditCtrl** object.

```
CHtmlEditCtrl( );
```

See Also

Reference

CHtmlEditCtrl Class

CHtmlEditCtrl::Create

Other Resources

CHtmlEditCtrl Members

CHtmlEditCtrl::Create

Creates a WebBrowser ActiveX control and attaches it to the **CHtmlEditCtrl** object. The WebBrowser ActiveX control automatically navigates to a default document and then is placed in edit mode by this function.

```
virtual BOOL Create(
   LPCTSTR lpszWindowName,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   int nID,
   CCreateContext *pContext = NULL
);
```

Parameters

lpszWindowName

This parameter is unused.

dwStyle

This parameter is unused.

rect

Specifies the control's size and position.

pParentWnd

Specifies the control's parent window. It must not be **NULL**.

nID

Specifies the control's ID.

pContext

This parameter is unused.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

See Also

Reference

CHtmlEditCtrl Class

CHtmlEditCtrl::CHtmlEditCtrl

Other Resources

CHtmlEditCtrl Members

CHtmlEditCtrl::GetDHtmlDocument

Retrieves the IHTMLDocument2 interface on the document currently loaded in the contained WebBrowser control

```
BOOL GetDHtmlDocument(
   IHTMLDocument2 **ppDocument
) const;
```

Parameters

ppDocument

The document interface.

See Also

Reference

CHtmlEditCtrl Class

Other Resources

CHtmlEditCtrl Members

CHtmlEditCtrl::GetStartDocument

Retrieves the URL to a default document to load in the contained WebBrowser control.

```
virtual LPCTSTR GetStartDocument();
```

See Also

Reference

CHtmlEditCtrl Class

Other Resources

CHtmlEditCtrl Members

CHtmlEditCtrlBase Class

Represents an HTML editing component.

```
template < class T > class CHtmlEditCtrlBase
```

Parameters

T

The name of the derived class.

Remarks

CHtmlEditCtrlBase provides member functions for the WebBrowser's HTML editing commands, such as Bold. (Alternately, you can call ExecCommand to execute the **IDM_BOLD** command.)

CHtmlEditCtrlBase is not intended to stand on its own. It is designed to be a base class for derived classes that expose the HTML editing functionality of the WebBrowser (see CHtmlEditCtrl and CHtmlEditView).

Requirements

Header: afxhtml.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

HTMLEdit Sample: Wraps the Internet Explorer MSHTML Editing Control

Reference

Hierarchy Chart

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase Members

Methods

AddToGlyphTable	Adds an entry to the glyph table, which specifies images to display for specific tags in design mode.
Bold	Toggles the bold state of the selected text.
Button	Overwrites a button control on the current selection.
CheckBox	Overwrites a check box control on the current selection.
ClearSelection	Clears the current selection.
Copy	Copies the current selection to the clipboard.
Cut	Copies the current selection to the clipboard and then deletes it.
Delete	Deletes the current selection.
DropDownBox	Overwrites a drop-down selection control on the current selection.
EmptyGlyphTable	Removes all entries from the glyph table, which hides all images displayed for tags in design mode.
ExecCommand	Executes a command.
Font	Opens a font dialog box to enable the user to change the text color, font, and font size of the current selection.
GetAbsolutePosition	Returns whether an element's position property is "absolute."
GetBackColor	Retrieves the background color of the current selection.
GetBlockFormat	Retrieves the current block format tag.
GetBlockFormatNames	Retrieves the strings corresponding to the available block format tags.
GetBookMark	Retrieves the name of a bookmark anchor.
GetDocument	Retrieves the document object.
GetDocumentHTML	Retrieves the HTML of the current document.
GetDocumentTitle	Retrieves the document's title.
GetEvent	Retrieves an interface pointer to the event object that contains information relevant to the most recent event.
GetEventSrcElement	Retrieves the object that fired the event.
GetFontFace	Retrieves the font name for the current selection.

GetFontSize	Retrieves the font size for the current selection.
GetForeColor	Retrieves the foreground (text) color of the current selection.
GetFrameZone	Returns the security zone of the current page in the web browser.
GetIsDirty	Indicates whether the HTML document has changed.
GetShowAlignedSiteTags	Returns whether a glyph is displayed for all elements that have a **styleFloat** property.
GetShowAllTags	Returns whether the WebBrowser displays glyphs to show the location of all tags in a document.
GetShowAreaTags	Retrieves whether the WebBrowser displays a glyph for area tags.
GetShowBRTags	Retrieves whether the WebBrowser displays a glyph for br tags.
GetShowCommentTags	Retrieves whether the WebBrowser displays a glyph for comment tags.
GetShowMiscTags	Retrieves whether the WebBrowser displays all the tags shown in Microsoft Internet Explorer 4.0.
GetShowScriptTags	Retrieves whether the WebBrowser displays a glyph for all the script tags.
GetShowStyleTags	Retrieves whether the WebBrowser displays a glyph for all the style tags.
GetShowUnknownTags	Retrieves whether the WebBrowser displays a glyph for all unknown tags.
HorizontalLine	Overwrites a horizontal line on the current selection.
HyperLink	Inserts a hyperlink on the current selection.
IE50Paste	Performs a paste operation compatible with Microsoft Internet Explorer 5.
Iframe	Overwrites an inline frame on the current selection.
Image	Overwrites an image on the current selection.
Indent	Increases the indent of the selected text by one indentation increment.
InsFieldSet	Overwrites a box on the current selection.
InsInputButton	Overwrites a button control on the current selection.
InsInputHidden	Inserts a hidden control on the current selection.
InsInputImage	Overwrites an image control on the current selection.
InsInputPassword	Overwrites a password control on the current selection.
InsInputReset	Overwrites a reset control on the current selection.
InsInputSubmit	Overwrites a submit control on the current selection.
InsInputUpload	Overwrites a file upload control on the current selection.

Is1DElement	Determines if an element is statically positioned.
Is2DElement	Determines if an element is absolutely positioned.
Italic	Toggles the current selection between italic and nonitalic.
JustifyCenter	Centers the format block in which the current selection is located.
JustifyLeft	Left-justifies the format block in which the current selection is located.
JustifyRight	Right-justifies the format block in which the current selection is located.
ListBox	Overwrites a list box selection control on the current selection.
Marquee	Overwrites an empty marquee on the current selection.
NewDocument	Creates a new document.
OrderList	Toggles the current selection between an ordered list and a normal format block.
Outdent	Decreases by one increment the indentation of the format block in which the current selection is l ocated.
Paragraph	Overwrites a line break on the current selection.
Paste	Overwrites the contents of the clipboard on the current selection.
PrintDocument	Prints the current document.
PrintPreview	Opens the Print Preview window for the current document using either the default print preview t emplate or a custom template.
QueryStatus	Call this method to query the status of commands.
RadioButton	Overwrites a radio control on the current selection.
RefreshDocument	Refreshes the current document.
RemoveFormat	Removes the formatting tags from the current selection.
SaveAs	Saves the current Web page to a file.
SelectAll	Selects the entire document.
Set2DPosition	Allows absolutely positioned elements to be moved by dragging.
SetAbsolutePosition	Sets an element's position property to "absolute" or "static."
SetAtomicSelection	Set atomic selection mode.
SetAutoURLDetectMode	Turns automatic URL detection on and off.
SetBackColor	Sets the background color of the current selection.

SetBlockFormat	Sets the current block format tag.
SetBookMark	Creates a bookmark anchor for the current selection or insertion point.
SetCSSEditingLevel	Selects which CSS level (CSS1 or CSS2) the editor will support, if any.
SetDefaultComposeSettings	Call this method to set the default compose settings.
SetDesignMode	Set design mode.
SetDisableEditFocusUI	Disables the hatched border and handles around an element that has edit focus.
SetDocumentHTML	Sets the HTML of the current document.
SetFontFace	Sets the font for the current selection.
SetFontSize	Sets the font size for the current selection.
SetForeColor	Sets the foreground (text) color of the current selection.
SetIE5PasteMode	Sets the paste operation to be compatible with Microsoft Internet Explorer 5.
SetLiveResize	Causes the WebBrowser to update an element's appearance continuously during a resizing or mo ving operation.
SetMultiSelect	Enables multiple selection.
SetOverrideCursor	Commands the WebBrowser never to change the mouse pointer.
SetOverwriteMode	Toggles the text-entry mode between insert and overwrite.
SetRespectVisInDesign	Hides invisible elements in design mode.
SetShowAlignedSiteTags	Displays a glyph for all elements that have a **styleFloat** property.
SetShowAllTags	Displays glyphs to show the location of all tags in a document.
SetShowAreaTags	Displays a glyph for all the area tags.
SetShowBRTags	Displays a glyph for all the br tags.
SetShowCommentTags	Displays a glyph for all the comment tags.
SetShowMiscTags	Displays all the tags shown in Microsoft Internet Explorer 4.0.
SetShowScriptTags	Displays a glyph for all the script tags.
SetShowStyleTags	Displays a glyph for all the style tags.
SetShowUnknownTags	Displays a glyph for all the unknown tags.
TextArea	Overwrites a multiline text input control on the current selection.

TextBox	Overwrites a text control on the current selection.
UnBookmark	Removes any bookmark from the current selection.
Underline	Toggles the current selection between underlined and not underlined.
Unlink	Removes any hyperlink from the current selection.
UnorderList	Toggles the current selection between an ordered list and a normal format block.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase Methods

For information about the methods in **CHtmlEditCtrlBase**, see CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::AddToGlyphTable

Adds an entry to the glyph table, which specifies images to display for specific tags in design mode.

```
HRESULT AddToGlyphTable(
   LPCTSTR szTag,
   LPCTSTR szImgUrl,
   unsigned short nTagType,
   unsigned short nAlignment,
   unsigned short nPosInfo,
   unsigned short nDirection,
   unsigned int nImgWidth,
   unsigned int nImgHeight
) const;
```

Parameters

szTag

The tag name (for example, "P" or "table").

szImgUrl

The image URL.

nTagType

Tag type: 0 means the image is for the opening tag only. 1 means the image is for the closing tag only. 2 means the image is for both the opening and closing tags. Single tags such as br and comment must be added with the tag type set to 0.

nAlignment

Alignment (rectangular elements only): This parameter indicates that the image is for an element with an alignment attribute. Left = 0, center = 1, right = 2, and undefined = 3. Left, right, or center attributes must be explicitly set on the element.

nPosInfo

Positioning information. Determines what cascading style sheets (CSS) positioning value the glyph applies to, where static positioning = 0, absolute positioning = 1, relative positioning = 2, and all = 3. This field enables you to specify one glyph for a tag when it is not positioned and another glyph to show an anchor point when the tag is positioned.

nDirection

The direction. This parameter specifies the image for a tag based on the reading order of the current language. 0 specifies left to right, 1 specifies right to left, 2 specifies top to bottom, 3 specifies bottom to top, and 4 specifies all. You normally set this field to 4.

nImgWidth

The image width in pixels.

nImgHeight

The image height in pixels.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information on the parameters, see "Glyph Table String Format" in Using Editing Glyphs.

This method sends the IDM_ADDTOGLYPHTABLE command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::EmptyGlyphTable

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::Bold

Toggles the bold state of the selected text.

```
HRESULT Bold( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_BOLD command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::Italic

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::Button

Overwrites a button control on the current selection.

```
HRESULT Button(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID of the button control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_BUTTON command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::CheckBox

Overwrites a check box control on the current selection.

```
HRESULT CheckBox(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID of the check box control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_CHECKBOX command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::ClearSelection

Clears the current selection.

```
HRESULT ClearSelection( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_CLEARSELECTION command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::Copy

Copies the current selection to the clipboard.

```
HRESULT Copy( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_COPY command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class
CHtmlEditCtrlBase::Cut
CHtmlEditCtrlBase::Paste

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::Cut

Copies the current selection to the clipboard and then deletes it.

```
HRESULT Cut( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_CUT command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class
CHtmlEditCtrlBase::Delete
CHtmlEditCtrlBase::Copy
CHtmlEditCtrlBase::Paste

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::Delete

Deletes the current selection.

```
HRESULT Delete( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_DELETE command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::Cut

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::DropDownBox

Overwrites a drop-down selection control on the current selection.

```
HRESULT DropDownBox(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID of the drop-down selection control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_DROPDOWNBOX command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::EmptyGlyphTable

Removes all entries from the glyph table, which hides all images displayed for tags in design mode.

```
HRESULT EmptyGlyphTable( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_EMPTYGLYPHTABLE command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::AddToGlyphTable

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::ExecCommand

Executes a command.

```
HRESULT ExecCommand(
   long cmdID,
   long cmdExecOpt,
   VARIANT* pInVar = NULL,
   VARIANT* pOutVar = NULL
) const;
HRESULT ExecCommand(
   const GUID* pGuid,
   long cmdID,
   long cmdExecOpt,
   VARIANT* pInVar = NULL,
   VARIANT* pOutVar = NULL
) const;
```

Parameters

cmdID

The command ID to be executed. For a list, see MSHTML Command Identifiers.

cmdExecOpt

Values taken from the OLECMDEXECOPT enumeration, which describe how the object should execute the command.

pInVar

The input arguments.

pOutVar

The command output.

pGuid

The GUID of the command group.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method provides the functionality of IOleCommandTarget::Exec.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::Font

Opens a font dialog box to enable the user to change the text color, font, and font size of the current selection.

```
HRESULT Font( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_FONT command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetAbsolutePosition

Returns whether an element's position property is "absolute."

```
HRESULT GetAbsolutePosition(
   bool& bCurValue
) const;
```

Parameters

bCurValue

True if the element's position property is set to "absolute."

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see IDM_ABSOLUTE_POSITION Command ID.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::SetAbsolutePosition

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetBackColor

Retrieves the background color of the current selection.

```
HRESULT GetBackColor(
   int& nColor
) const;
```

Parameters

nColor

The background color.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_BACKCOLOR Command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class
CHtmlEditCtrlBase::SetBackColor
CHtmlEditCtrlBase::GetForeColor

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetBlockFormat

Retrieves the current block format tag.

```
HRESULT GetBlockFormat(
   CString& strFormat
) const;
```

Parameters

strFormat

The current block format tag.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_BLOCKFMT command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::GetBlockFormatNames

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetBlockFormatNames

Retrieves the strings corresponding to the available block format tags.

```
HRESULT GetBlockFormatNames(
   CStringArray& sa
) const;
```

Parameters

sa

The available block format tags, as an array of strings.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_GETBLOCKFMTS command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::GetBlockFormat

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetBookMark

Retrieves the name of a bookmark anchor.

```
HRESULT GetBookMark(
   CString& strAnchor
) const;
```

Parameters

strAnchor

The name of a bookmark anchor.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see IDM_BOOKMARK Command ID.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::SetBookMark

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetDocument

Retrieves the document object.

```
HRESULT GetDocument(
   IHTMLDocument2** ppDoc
) const;
```

Parameters

ppDoc

The document object.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetDocumentHTML

Retrieves the HTML of the current document.

```
HRESULT GetDocumentHTML(
   CString& szHTML
) const;
```

Parameters

szHTML

The HTML.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::SetDocumentHTML

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetDocumentTitle

Retrieves the document's title.

```
HRESULT GetDocumentTitle(
   CString& szTitle
) const;
```

Parameters

szTitle

The document's title.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetEvent

Retrieves an interface pointer to the event object that contains information relevant to the most recent event.

```
HRESULT GetEvent(
   IHTMLEventObj ** ppEventObj
) const;
```

Parameters

ppEventObj

The event object.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlEditCtrlBase Class
IHTMLEventObj Interface
event Property (IHTMLWindow2)

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetEventSrcElement

Retrieves the object that fired the event.

```
HRESULT GetEventSrcElement(
   IHTMLElement ** ppSrcElement
) const;
```

Parameters

ppSrcElement

The element that fired the event.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlEditCtrlBase Class
IHTMLElement Interface
srcElement Property (IHTMLEventObj)

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetFontFace

Retrieves the font name for the current selection.

```
HRESULT GetFontFace(
   CString& strFace
) const;
```

Parameters

strFace

The font name.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

If the current selection uses more than one font, *strFace* will be an empty string.

This method sends the IDM_FONTNAME command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class
CHtmlEditCtrlBase::SetFontFace
CHtmlEditCtrlBase::GetFontSize

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetFontSize

Retrieves the font size for the current selection.

```
HRESULT GetFontSize(
   short& nSize
) const;
```

Parameters

nSize

The font size.

Return Value

Returns the HTML font size (1-7). Returns 0 if the selection contains multiple font sizes.

Remarks

This method sends the IDM_FONTSIZE command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class
CHtmlEditCtrlBase::SetFontSize
CHtmlEditCtrlBase::GetFontFace

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetForeColor

Retrieves the foreground (text) color of the current selection.

```
HRESULT GetForeColor(
   int & nColor
);
```

Parameters

nColor

The foreground color.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_FORECOLOR Command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class
CHtmlEditCtrlBase::SetForeColor
CHtmlEditCtrlBase::GetBackColor

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetFrameZone

Returns the security zone of the current page in the web browser.

```
HRESULT GetFrameZone(
   short& nZone
) const;
```

Parameters

nZone

The security zone.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_GETFRAMEZONE command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetIsDirty

Indicates whether the HTML document has changed.

```
HRESULT GetIsDirty( ) const;
```

Remarks

Indicates whether the document has changed. **GetIsDirty** returns an **HRESULT** from IPersistStorage::IsDirty.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetShowAlignedSiteTags

Returns whether a glyph is displayed for all elements that have a **styleFloat** property.

```
HRESULT GetShowAlignedSiteTags(
   bool& bCurValue
) const;
```

Parameters

bCurValue

True if a glyph is displayed for all elements that have a **styleFloat** property; false if no glyph is displayed.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see IDM_SHOWALIGNEDSITETAGS Command ID.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::SetShowAlignedSiteTags

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetShowAllTags

Returns whether the WebBrowser displays glyphs to show the location of all tags in a document.

```
HRESULT GetShowAllTags(
   bool& bCurValue
) const;
```

Parameters

bCurValue

True if the WebBrowser displays glyphs to show the location of all tags in a document; false if it does not.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see IDM_SHOWALLTAGS Command ID.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::SetShowAllTags

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetShowAreaTags

Retrieves whether the WebBrowser displays a glyph for area tags.

```
HRESULT GetShowAreaTags(
   bool& bCurValue
) const;
```

Parameters

bCurValue

True if the WebBrowser displays a glyph for area tags, false if it does not.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see IDM_SHOWAREATAGS Command ID.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::SetShowAreaTags

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetShowBRTags

Retrieves whether the WebBrowser displays a glyph for br tags.

```
HRESULT GetShowBRTags(
   bool& bCurValue
) const;
```

Parameters

bCurValue

True if the WebBrowser displays a glyph for br tags, false if it doesn't.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see IDM_SHOWWBRTAGS Command ID.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::SetShowBRTags

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetShowCommentTags

Retrieves whether the WebBrowser displays a glyph for comment tags.

```
HRESULT GetShowCommentTags(
   bool& bCurValue
) const;
```

Parameters

bCurValue

True if the WebBrowser displays a glyph for comment tags, false if it doesn't.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see IDM_SHOWCOMMENTTAGS Command ID.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::SetShowCommentTags

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetShowMiscTags

Retrieves whether the WebBrowser displays all the tags shown in Microsoft Internet Explorer 4.0.

```
HRESULT GetShowMiscTags(
   bool& bCurValue
) const;
```

Parameters

bCurValue

True if the WebBrowser displays all the tags shown in Microsoft Internet Explorer 4.0, false if it does not.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see IDM_SHOWMISCTAGS Command ID.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::SetShowMiscTags

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetShowScriptTags

Retrieves whether the WebBrowser displays a glyph for all the script tags.

```
HRESULT GetShowScriptTags(
   bool& bCurValue
) const;
```

Parameters

bCurValue

True if the WebBrowser displays a glyph for all the script tags, false if it does not.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see IDM_SHOWSCRIPTTAGS Command ID.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::SetShowScriptTags

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetShowStyleTags

Retrieves whether the WebBrowser displays a glyph for all the style tags.

```
HRESULT GetShowStyleTags(
   bool& bCurValue
) const;
```

Parameters

bCurValue

True if the WebBrowser displays a glyph for all the style tags, false if it does not

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see IDM_SHOWSTYLETAGS Command ID.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::SetShowStyleTags

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::GetShowUnknownTags

Retrieves whether the WebBrowser displays a glyph for all unknown tags.

```
HRESULT GetShowUnknownTags(
   bool& bCurValue
) const;
```

Parameters

bCurValue

True if the WebBrowser displays a glyph for all unknown tags, false if it does not.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

For more information, see IDM_SHOWUNKNOWNTAGS Command ID.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::SetShowUnknownTags

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::HorizontalLine

Overwrites a horizontal line on the current selection.

```
HRESULT HorizontalLine(
   LPCTSTR szId = NULL
) const;
```

Parameters

szID

The ID for the horizontal line.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_HORIZONTALLINE command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::HyperLink

Inserts a hyperlink on the current selection.

```
HRESULT HyperLink(
   LPCTSTR szUrl = NULL
) const;
```

Parameters

szUrl

The hyperlink URL.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_HYPERLINK command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::Unlink

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::IE50Paste

Performs a paste operation compatible with Microsoft® Internet Explorer 5.

```
HRESULT IE50Paste(
   LPCTSTR szData
) const;
```

Parameters

szData

The string to paste.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_IE50_PASTE command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::SetIE5PasteMode

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::Iframe

Overwrites an inline frame on the current selection.

```
HRESULT Iframe(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID for the inline frame.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_IFRAME command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::Image

Overwrites an image on the current selection.

```
HRESULT Image(
   LPCTSTR szUrl = NULL
) const;
```

Parameters

szUrl

The path and file name of the image to be inserted.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_IMAGE command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::Indent

Increases the indent of the selected text by one indentation increment.

```
HRESULT Indent( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_INDENT command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::Outdent

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::InsFieldSet

Overwrites a box on the current selection.

```
HRESULT InsFieldSet(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID for the box.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_INSFIELDSET command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::InsInputButton

Overwrites a button control on the current selection.

```
HRESULT InsInputButton(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID for the button control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_INSINPUTBUTTON command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::InsInputHidden

Inserts a hidden control on the current selection.

```
HRESULT InsInputHidden(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID for the hidden control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_INSINPUTHIDDEN command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::InsInputImage

Overwrites an image control on the current selection.

```
HRESULT InsInputImage(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID for the image control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_INSINPUTIMAGE command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::InsInputPassword

Overwrites a password control on the current selection.

```
HRESULT InsInputPassword(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID for the password control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_INSINPUTPASSWORD command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::InsInputReset

Overwrites a reset control on the current selection.

```
HRESULT InsInputReset(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID for the reset control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_INSINPUTRESET command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::InsInputSubmit

Overwrites a submit control on the current selection.

```
HRESULT InsInputSubmit(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID for the submit control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_INSINPUTSUBMIT command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::InsInputUpload

Overwrites a file upload control on the current selection.

```
HRESULT InsInputUpload(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID for the file upload control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_INSINPUTUPLOAD command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::Is1DElement

Determines if an element is statically positioned.

```
HRESULT Is1DElement(
   bool& bValue
) const;
```

Parameters

bValue

True if the element is statically positioned, false otherwise.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_1D_ELEMENT command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::Is2DElement

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::Is2DElement

Determines if an element is absolutely positioned.

```
HRESULT Is2DElement(
   bool& bValue
) const;
```

Parameters

bValue

True if the element is absolutely positioned, false otherwise.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_2D_ELEMENT command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::Is1DElement

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::Italic

Toggles the current selection between italic and nonitalic.

```
HRESULT Italic( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_ITALIC command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class
CHtmlEditCtrlBase::Bold
CHtmlEditCtrlBase::Underline

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::JustifyCenter

Centers the format block in which the current selection is located.

```
HRESULT JustifyCenter( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_JUSTIFYCENTER command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class
CHtmlEditCtrlBase::JustifyLeft
CHtmlEditCtrlBase::JustifyRight

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::JustifyLeft

Left-justifies the format block in which the current selection is located.

```
HRESULT JustifyLeft( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_JUSTIFYLEFT command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class
CHtmlEditCtrlBase::JustifyCenter
CHtmlEditCtrlBase::JustifyRight

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::JustifyRight

Right-justifies the format block in which the current selection is located.

```
HRESULT JustifyRight( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_JUSTIFYRIGHT command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class
CHtmlEditCtrlBase::JustifyCenter
CHtmlEditCtrlBase::JustifyLeft

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::ListBox

Overwrites a list box selection control on the current selection.

```
HRESULT ListBox(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID for the list box control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_LISTBOX command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::Marquee

Overwrites an empty marquee on the current selection.

```
HRESULT Marquee(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID for the marquee.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_MARQUEE command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::NewDocument

Creates a new document.

```
HRESULT NewDocument( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::OrderList

Toggles the current selection between an ordered list and a normal format block.

```
HRESULT OrderList(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID for the ordered list.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_ORDERLIST command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::UnorderList

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::Outdent

Decreases by one increment the indentation of the format block in which the current selection is located.

```
HRESULT Outdent( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_OUTDENT command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::Indent

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::Paragraph

Overwrites a line break on the current selection.

```
HRESULT Paragraph(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID for the paragraph.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_PARAGRAPH command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::Paste

Overwrites the contents of the clipboard on the current selection.

```
HRESULT Paste( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_PASTE command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class
CHtmlEditCtrlBase::Copy
CHtmlEditCtrlBase::Cut
CHtmlEditCtrlBase::IE50Paste

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::PrintDocument

Prints the current document.

```
HRESULT PrintDocument( ) const;
HRESULT PrintDocument(
   LPCTSTR szPrintTemplate
) const;
HRESULT PrintDocument(
   bool bShowPrintDialog
) const;
```

Parameters

szPrintTemplate

Path to a print template; if none is specified, the default print template is used.

bShowPrintDialog

If true, shows the Print dialog.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_PRINT command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::PrintPreview

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::PrintPreview

Opens the Print Preview window for the current document using either the default print preview template or a custom template.

```
HRESULT PrintPreview( ) const;
HRESULT PrintPreview(
   LPCTSTR szPrintTemplate
) const;
```

Parameters

szPrintTemplate

Path to a print template.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_PRINTPREVIEW command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::PrintDocument

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::QueryStatus

Call this method to query the status of commands.

```
long QueryStatus(
   long cmdID
) const;
```

Parameters

cmdID

The command ID. Command identifiers are taken from the **CGID_MSHTML** command group. These commands are defined in Mshtmcid.h.

Return Value

Returns an OLECMDF indicating the status for *cmdID*, or 0 on failure.

See Also

Reference

CHtmlEditCtrlBase Class

IOleCommandTarget::QueryStatus

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::RadioButton

Overwrites a radio control on the current selection.

```
HRESULT RadioButton(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID of the radio button.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_RADIOBUTTON command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::RefreshDocument

Refreshes the current document.

```
HRESULT RefreshDocument( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_REFRESH Command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::RemoveFormat

Removes the formatting tags from the current selection.

```
HRESULT RemoveFormat( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_REMOVEFORMAT command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SaveAs

Saves the current Web page to a file.

```
HRESULT SaveAs(
   LPCTSTR szPath = NULL
) const;
```

Parameters

szPath

The path and file name to which to save the Web page.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_SAVEAS command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SelectAll

Selects the entire document.

```
HRESULT SelectAll( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_SELECTALL command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::Set2DPosition

Allows absolutely positioned elements to be moved by dragging.

```
HRESULT Set2DPosition(
   bool bNewValue
) const;
```

Parameters

bNewValue

If true, absolutely positioned elements can be moved by dragging.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_2D_POSITION command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetAbsolutePosition

Sets an element's position property to "absolute" or "static."

```
HRESULT SetAbsolutePosition(
   bool bNewValue
) const;
```

Parameters

bNewValue

If true, the element's position property is "absolute"; if false, it is "static."

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_ABSOLUTE_POSITION command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::GetAbsolutePosition

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetAtomicSelection

Set atomic selection mode.

```
HRESULT SetAtomicSelection(
   bool bNewValue
) const;
```

Parameters

bNewValue

If true, any element that has an ATOMICSELECTION attribute set to TRUE will be selectable only as a unit.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_ATOMICSELECTION command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetAutoURLDetectMode

Turns automatic URL detection on and off.

```
HRESULT SetAutoURLDetectMode(
   bool bNewValue
) const;
```

Parameters

bNewValue

If true, automatic URL detection is enabled.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_AUTOURLDETECT_MODE command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetBackColor

Sets the background color of the current selection.

```
HRESULT SetBackColor(
   int nColor
) const;
HRESULT SetBackColor(
   LPCTSTR szColor
) const;
```

Parameters

nColor

The color. See *pvaIn* in IDM_BACKCOLOR Command ID.

szColor

The color. See *pvaIn* in IDM_BACKCOLOR Command ID.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_BACKCOLOR_ command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::GetBackColor

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetBlockFormat

Sets the current block format tag.

```
HRESULT SetBlockFormat(
   LPCTSTR szFormat
) const;
```

Parameters

szFormat

The format tag.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_BLOCKFMT_command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::GetBlockFormat

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetBookMark

Creates a bookmark anchor for the current selection or insertion point.

```
HRESULT SetBookMark(
   LPCTSTR szAnchorName
) const;
```

Parameters

szAnchorName

The anchor name.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_BOOKMARK command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class
CHtmlEditCtrlBase::GetBookMark
CHtmlEditCtrlBase::UnBookmark

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetCSSEditingLevel

Selects which CSS level (CSS1 or CSS2) the editor will support, if any.

```
HRESULT SetCSSEditingLevel(
   short nLevel
) const;
```

Parameters

nLevel

The CSS level. Pass 0 if you do not want CSS support.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_CSSEDITING_LEVEL command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetDefaultComposeSettings

Call this method to set the default compose settings.

```
HRESULT SetDefaultComposeSettings(
   LPCSTR szFontName = NULL,
   unsigned short nFontSize = 3,
   COLORREF crFontColor = 0xFF000000,
   COLORREF crFontBgColor = 0xFF000000,
   bool bBold = false,
   bool bItalic = false,
   bool bUnderline = false
) const;
```

Parameters

szFontName

The font name.

nFontSize

The font size.

crFontColor

The font color.

crFontBgColor

The font background color.

bBold

Pass true for bold text.

bItalic

Pass true for italic text.

bUnderline

Pass true for underlined text.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_COMPOSESETTINGS command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetDesignMode

Set design mode.

```
BOOL SetDesignMode(
   BOOL bMode
) const;
```

Parameters

bMode

If true, turns design mode on.

Return Value

Returns TRUE on success, FALSE on failure.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetDisableEditFocusUI

Disables the hatched border and handles around an element that has edit focus.

```
HRESULT SetDisableEditFocusUI(
   bool bNewValue
) const;
```

Parameters

bNewValue

If true, disables the hatched border and handles around a site selectable element when the element has "edit focus" in design mode; that is, when the text or contents of the element can be edited.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM_DISABLE_EDITFOCUS_UI command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetDocumentHTML

Sets the HTML of the current document.

```
HRESULT SetDocumentHTML(
   LPCTSTR szHTML
) const;
```

Parameters

szHTML

The HTML.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::GetDocumentHTML

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetFontFace

Sets the font for the current selection.

```
HRESULT SetFontFace(
   LPCTSTR szFace
) const;
```

Parameters

szFace

The font name.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM FONTNAME Command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class
CHtmlEditCtrlBase::GetFontFace
CHtmlEditCtrlBase::SetFontSize

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetFontSize

Sets the font size for the current selection.

```
HRESULT SetFontSize(
   unsigned short size
) const;
```

Parameters

size

The HTML font size (1-7). A value of 0 sets the font size to 1.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM FONTSIZE command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class
CHtmlEditCtrlBase::GetFontSize
CHtmlEditCtrlBase::GetFontFace

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetForeColor

Sets the foreground (text) color of the current selection.

```
HRESULT SetForeColor(
   LPCTSTR szColor
) const;
HRESULT SetForeColor(
   int nColor
) const;
```

Parameters

szColor

The color.

nColor

The color.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM FORECOLOR command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class
CHtmlEditCtrlBase::GetForeColor
CHtmlEditCtrlBase::SetBackColor

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetIE5PasteMode

Sets the paste operation to be compatible with Microsoft Internet Explorer 5.

```
HRESULT SetIE5PasteMode(
   bool bNewValue
) const;
```

Parameters

bNewValue

If true, all paste operations are compatible with Internet Explorer 5; if false, paste operations are compatible with Internet Explorer 5.5.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM IE50_PASTE_MODE command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::IE50Paste

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetLiveResize

Causes the WebBrowser to update an element's appearance continuously during a resizing or moving operation, rather than updating only at the completion of the move or resize.

```
HRESULT SetLiveResize(
   bool bNewValue
) const;
```

Parameters

bNewValue

If true, causes the WebBrowser to update an element's appearance continuously during a resizing or moving operation; if false, it updates only at the completion of the move or resize.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM LIVERESIZE command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetMultiSelect

Enables multiple selection.

```
HRESULT SetMultiSelect(
   bool bNewValue
) const;
```

Parameters

bNewValue

If true, allows for the selection of more than one site-selectable element at a time when the user holds down the SHIFT or CTRL keys.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM MULTIPLESELECTION command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetOverrideCursor

Commands the WebBrowser never to change the mouse pointer.

```
HRESULT SetOverrideCursor(
   bool bNewValue
) const;
```

Parameters

bNewValue

If true, the WebBrowser will not change the mouse pointer.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM OVERRIDE_CURSOR command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetOverwriteMode

Toggles the text-entry mode between insert and overwrite.

```
HRESULT SetOverwriteMode(
   bool bMode
) const;
```

Parameters

bMode

If true, text-entry mode is overwrite; if false, text-entry mode is insert.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM OVERWRITE command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetRespectVisInDesign

Hides invisible elements in design mode.

```
HRESULT SetRespectVisInDesign(
   bool bNewValue
) const;
```

Parameters

bNewValue

If true, any elements that have a visibility set to "hidden" or display property set to "none" will not be shown in both design mode and browse mode; if false, those elements will be displayed only in browse mode.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM RESPECTVISIBILITY_INDESIGN command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetShowAlignedSiteTags

Displays a glyph for all elements that have a **styleFloat** property.

```
HRESULT SetShowAlignedSiteTags(
   bool bNewValue
) const;
```

Parameters

bNewValue

If true, displays a glyph for all elements that have a **styleFloat** property.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM SHOWALIGNEDSITETAGS command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::GetShowAlignedSiteTags

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetShowAllTags

Displays glyphs to show the location of all tags in a document.

```
HRESULT SetShowAllTags(
   bool bNewValue
) const;
```

Parameters

bNewValue

If true, displays glyphs to show the location of all tags in a document.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM SHOWALLTAGS command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::GetShowAllTags

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetShowAreaTags

Displays a glyph for all the area tags.

```
HRESULT SetShowAreaTags(
   bool bNewValue
) const;
```

Parameters

bNewValue

If true, displays a glyph for all the area tags.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM SHOWAREATAGS command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::GetShowAreaTags

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetShowBRTags

Displays a glyph for all the br tags.

```
HRESULT SetShowBRTags(
   bool bNewValue
) const;
```

Parameters

bNewValue

If true, displays a glyph for all the br tags.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM SHOWWBRTAGS command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::GetShowBRTags

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetShowCommentTags

Displays a glyph for all the comment tags.

```
HRESULT SetShowCommentTags(
   bool bNewValue
) const;
```

Parameters

bNewValue

If true, displays a glyph for all the comment tags.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM SHOWCOMMENTTAGS command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::GetShowCommentTags

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetShowMiscTags

Displays all the tags shown in Microsoft Internet Explorer 4.0.

```
HRESULT SetShowMiscTags(
   bool bNewValue
) const;
```

Parameters

bNewValue

If true, displays all the tags shown in Microsoft Internet Explorer 4.0.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM SHOWMISCTAGS command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::GetShowMiscTags

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetShowScriptTags

Displays a glyph for all the script tags.

```
HRESULT SetShowScriptTags(
   bool bNewValue
) const;
```

Parameters

bNewValue

If true, displays a glyph for all the script tags.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM SHOWSCRIPTTAGS command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::GetShowScriptTags

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetShowStyleTags

Displays a glyph for all the style tags.

```
HRESULT SetShowStyleTags(
   bool bNewValue
) const;
```

Parameters

bNewValue

If true, displays a glyph for all the style tags.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM SHOWSTYLETAGS command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::GetShowStyleTags

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::SetShowUnknownTags

Displays a glyph for all the unknown tags.

```
HRESULT SetShowUnknownTags(
   bool bNewValue
) const;
```

Parameters

bNewValue

If true, displays a glyph for all the unknown tags.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM SHOWUNKNOWNTAGS command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::GetShowUnknownTags

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::TextArea

Overwrites a multiline text input control on the current selection.

```
HRESULT TextArea(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID of the multiline text input control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM TEXTAREA command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::TextBox

Overwrites a text control on the current selection.

```
HRESULT TextBox(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID of the text control.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM TEXTBOX command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::UnBookmark

Removes any bookmark from the current selection.

```
HRESULT UnBookmark( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM UNBOOKMARK command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class
CHtmlEditCtrlBase::SetBookMark
CHtmlEditCtrlBase::GetBookMark

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::Underline

Toggles the current selection between underlined and not underlined.

```
HRESULT Underline( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM UNDERLINE command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class
CHtmlEditCtrlBase::Bold
CHtmlEditCtrlBase::Italic

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::Unlink

Removes any hyperlink from the current selection.

```
HRESULT Unlink( ) const;
```

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM UNLINK command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::HyperLink

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditCtrlBase::UnorderList

Toggles the current selection between an ordered list and a normal format block.

```
HRESULT UnorderList(
   LPCTSTR szId = NULL
) const;
```

Parameters

szId

The ID of the unordered list.

Return Value

Returns S_OK on success, or an error HRESULT on failure.

Remarks

This method sends the IDM UNORDERLIST command ID to the WebBrowser control.

See Also

Reference

CHtmlEditCtrlBase Class

CHtmlEditCtrlBase::OrderList

Other Resources

CHtmlEditCtrlBase Members

CHtmlEditDoc Class

With CHtmlEditView, provides the functionality of the WebBrowser editing platform within the context of the MFC document-view architecture.

```
class AFX_NOVTABLE CHtmlEditDoc : public CDocument
```

Requirements

Header: afxhtml.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

HTMLEdit Sample: Wraps the Internet Explorer MSHTML Editing Control

Reference

Hierarchy Chart

Other Resources

CHtmlEditDoc Members

CHtmlEditDoc Members

Member Functions

CHtmlEditDoc	Constructs a **CHtmlEditDoc** object.
GetView	Retrieves the **CHtmlEditView** object attached to this document.
IsModified	Returns whether the associated view's WebBrowser control contains a document that has been modified by the user.
OpenURL	Opens a URL.

See Also

Reference

CHtmlEditDoc Class

CHtmlEditDoc Member Functions

For information about the member functions in **CHtmlEditDoc**, see CHtmlEditDoc Members.

CHtmlEditDoc::CHtmlEditDoc

Constructs a **CHtmlEditDoc** object.

```
CHtmlEditDoc( );
```

See Also

Reference

CHtmlEditDoc Class

Other Resources

CHtmlEditDoc Members

CHtmlEditDoc::GetView

Retrieves the CHtmlEditView object attached to this document.

```
virtual CHtmlEditView * GetView( ) const;
```

Return Value

Returns a pointer to the document's **CHtmlEditView** object.

See Also

Reference

CHtmlEditDoc Class

Other Resources

CHtmlEditDoc Members

CHtmlEditDoc::IsModified

Returns whether the associated view's WebBrowser control contains a document that has been modified by the user.

```
virtual BOOL IsModified();
```

See Also

Reference

CHtmlEditDoc Class

Other Resources

CHtmlEditDoc Members

CHtmlEditDoc::OpenURL

Opens a URL.

```
virtual BOOL OpenURL(
   LPCTSTR lpszURL
);
```

Parameters

lpszURL

The URL to open.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

See Also

Reference

CHtmlEditDoc Class

Other Resources

CHtmlEditDoc Members

CHtmlEditView Class

Provides the functionality of the WebBrowser editing platform within the context of MFC's document/view architecture.

```
class CHtmlEditView : public CHtmlView, public CHtmlEditCtrlBase< CHtmlEditView >
```

Requirements

Header: afxhtml.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

HTMLEdit Sample: Wraps the Internet Explorer MSHTML Editing Control

Reference

Hierarchy Chart

Other Resources

CHtmlEditView Members

CHtmlEditView Members

Member Functions

CHtmlEditView	Constructs a **CHtmlEditView** object.
Create	Creates a new window object.
GetDHtmlDocument	Returns the **IHTMLDocument2** interface on the current document.
GetStartDocument	Retrieves the name of the default document for this view.

See Also

Reference

CHtmlEditView Class

CHtmlEditView Member Functions

For information about the member functions in **CHtmlEditView**, see CHtmlEditView Members.

CHtmlEditView::CHtmlEditView

Constructs a **CHtmlEditView** object.

```
CHtmlEditView( );
```

See Also

Reference

CHtmlEditView Class

Other Resources

CHtmlEditView Members

CHtmlEditView::Create

Creates a new window object.

```
virtual BOOL Create(
   LPCTSTR lpszClassName,
   LPCTSTR lpszWindowName,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID,
   CCreateContext* pContext = NULL
);
```

Parameters

lpszClassName

Points to a null-terminated character string that names the Windows class. The class name can be any name registered with the AfxRegisterWndClass global function or the **RegisterClass** Windows function. If **NULL**, uses the predefined default CFrameWnd attributes.

lpszWindowName

Points to a null-terminated character string that represents the window name.

dwStyle

Specifies the window style attributes. By default, the **WS_VISIBLE** and **WS_CHILD** Windows styles are set.

rect

A reference to a RECT structure specifying the size and position of the window. The **rectDefault** value allows Windows to specify the size and position of the new window.

pParentWnd

A pointer to the parent window of the control.

nID

The ID number of the view. By default, set to **AFX_IDW_PANE_FIRST**.

pContext

A pointer to a CCreateContext. **NULL** by default.

Remarks

This method will also call the contained WebBrowser's **Navigate** method to load a default document (see CHtmlEditView::GetStartDocument).

See Also

Reference

CHtmlEditView Class

CHtmlView::Create

Other Resources

CHtmlEditView Members

CHtmlEditView::GetDHtmlDocument

Returns the **IHTMLDocument2** interface on the current document.

```
BOOL GetDHtmlDocument(
   IHTMLDocument2 **ppDocument
) const;
```

Parameters

ppDocument

The IHTMLDocument2 interface.

See Also

Reference

CHtmlEditView Class

Other Resources

CHtmlEditView Members

CHtmlEditView::GetStartDocument

Retrieves the name of the default document for this view.

```
virtual LPCTSTR GetStartDocument();
```

See Also

Reference

CHtmlEditView Class

Other Resources

CHtmlEditView Members

CHtmlStream Class

Manages in-memory HTML.

```
class CHtmlStream
```

Remarks

CHtmlStream does not have a base class.

HTML memory files are useful for temporarily storing raw bytes or serialized objects prior to their transmission. Although it is not derived from **CFile**, **CHtmlStream** behaves like the **CFile**–derived class **CMemFile**, except **CHtmlStream** is used to store data in a temporary buffer prior to sending it out, and the data stored in a **CHtmlStream** memory file cannot be read.

CHtmlStream objects usually are created automatically and handed to you by CHttpServer::ConstructStream; however, you can override **CHttpServer::ConstructStream** and provide your own special functionality.

CHtmlStream objects can automatically allocate their own memory or you can attach your own memory block to the **CHtmlStream** object by calling Attach. In either case, memory for growing the memory file automatically is allocated in *nGrowBytes*-sized increments if *nGrowBytes* is not zero. Set *nGrowBytes* with a parameter to the constructor.

The memory will automatically be deleted upon destruction of the **CHtmlStream** object if the memory was originally allocated by the **CHtmlStream** object; otherwise, you are responsible for deallocating the memory you attached to the object.

CHtmlStream uses the run-time library functions malloc, realloc, and free to allocate, reallocate, and deallocate memory; and the intrinsic memcpy to block copy memory when growing the buffer. To change this behavior or the behavior when **CHtmlStream** grows a file, derive your own class from **CHtmlStream** and override the appropriate functions.

Requirements

Header: afxisapi.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart
CHttpServer Class
CHttpFilter Class

Other Resources

CHtmlStream Members

CHtmlStream Members

Data Members

m_nStreamSize	Contains the size of the stream.

Construction

CHtmlStream	Constructs a **CHtmlStream** object.

Operations

Attach	Attaches a block of memory to **CHtmlStream**.
Close	Closes the stream and frees the buffer.
GetStreamSize	Gets the size of the **CHtmlStream**.
InitStream	Initializes a stream associated with a **CHtmlStream** object.

Overridables

Abort	Ends a stream and ignores all warnings and errors.
Alloc	allocates memory in a **CHtmlStream** object.
Detach	closes the **CHtmlStream**.
Free	frees memory in a **CHtmlStream** object.
GrowStream	Grows a **CHtmlStream** object.
Memcpy	Copies memory to grow a **CHtmlStream** object.
Realloc	reallocates memory in a **CHtmlStream** object.
Reset	empties a **CHtmlStream** object.
Write	Writes data from the buffer to the current stream.

Operators

operator <<	Writes data into a stream.

See Also

Reference

CHtmlStream Class

Hierarchy Chart

CHtmlStream Member Functions

For information about the member functions in **CHtmlStream**, see CHtmlStream Members.

CHtmlStream::Abort

Called by the framework to end the stream and make the **CHtmlStream** unavailable for writing.

```
virtual void Abort( );
```

Remarks

Use **Abort** to clean up the stream after a catastrophic error. Use Reset to erase the content of the stream if you plan to write to it again.

Override this member function to implement custom cleanup.

See Also

Reference

CHtmlStream Class
Hierarchy Chart
CHtmlStream::Reset
CHtmlStream::Close

Other Resources

CHtmlStream Members

CHtmlStream::Alloc

Called by the framework to allocate memory.

```
virtual BYTE* Alloc(
   DWORD nBytes
);
```

Parameters

nBytes

Number of bytes of memory to be allocated.

Return Value

A pointer to the memory block that was allocated, or **NULL** if the allocation failed.

Remarks

Override this function to implement custom memory allocation. If you override this function, override Free, too.

The default implementation uses the run-time library function malloc to allocate memory.

See Also

Reference

CHtmlStream Class
Hierarchy Chart
CHtmlStream::Realloc

Other Resources

CHtmlStream Members

CHtmlStream::Attach

Call this function to attach a block of memory to **CHtmlStream**.

```
void Attach(
   BYTE* lpBuffer,
   UINT nBufferSize,
   UINT nGrowBytes = 0
);
```

Parameters

lpBuffer

Pointer to the buffer to be attached to **CHtmlStream**.

nBufferSize

An integer that specifies the size of the buffer in bytes.

nGrowBytes

The memory allocation increment in bytes.

Remarks

This causes **CHtmlStream** to use the block of memory as the memory file.

If *nGrowBytes* is 0, **CHtmlStream** will set the file length to *nBufferSize*. This means that the data in the memory block before it was attached to **CHtmlStream** will be used as the file data. Memory files created in this manner cannot be grown.

Because the file cannot be grown, be careful not to cause **CHtmlStream** to attempt to grow the file. Don't use operator << to add data.

See Also

Reference

CHtmlStream Class
Hierarchy Chart
CHtmlStream::Detach

Other Resources

CHtmlStream Members

CHtmlStream::CHtmlStream

This member function is called by the framework during the construction of a **CHtmlStream** object.

```
CHtmlStream(
   UINT nGrowBytes = 4096
);
CHtmlStream(
   BYTE* lpBuffer,
   UINT nBufferSize,
   UINT nGrowBytes = 0
);
```

Parameters

nGrowBytes

The memory allocation increment in bytes.

lpBuffer

Pointer to a buffer that receives information of the size *nBufferSize*.

nBufferSize

An integer that specifies the size of the file buffer, in bytes.

Remarks

Normally, a **CHtmlStream** object is created automatically and handed to you by CHttpServer::ConstructStream. You can change the behavior of the **CHtmlStream** object associated with a **CHttpServerContext** object by overriding **CHttpServer::ConstructStream**. For example, you might want to set *nGrowBytes* to a specific value. Use caution if you set *nGrowBytes*, because it will affect the performance of your code. The *nGrowBytes* parameter tells MFC how rapidly to increase the memory block associated with the stream. If the value is large, your code will be faster, but it will waste memory. If the value is small, your code will use less memory, but it will waste time by allocating memory more frequently.

See Also

Reference

CHtmlStream Class
Hierarchy Chart
CHtmlStream::InitStream
CHtmlStream::Attach
CHtmlStream::Alloc

Other Resources

CHtmlStream Members

CHtmlStream::Close

Called by the framework to close the HTML stream and free the buffer.

```
virtual void Close( );
```

Remarks

Override this member function to perform an action before the HTML stream is closed.

See Also

Reference

CHtmlStream Class
Hierarchy Chart
CHtmlStream::Abort
CHtmlStream::Reset

Other Resources

CHtmlStream Members

CHtmlStream::Detach

Call this function to get a pointer to the memory block being used by **CHtmlStream**.

```
BYTE* Detach( );
```

Return Value

A pointer to the memory block that contains the contents of the HTML stream.

Remarks

Calling this function also closes the **CHtmlStream**. You can reattach the memory block to **CHtmlStream** by calling Attach. If you want to reattach the file and use the data in it, you should call GetStreamSize to get the length of the file before calling **Detach**. Note that if you attach a memory block to **CHtmlStream** so that you can use its data (*nGrowBytes* == 0), then you will not be able to grow the memory.

See Also

Reference

CHtmlStream Class
Hierarchy Chart
CHtmlStream::Attach

Other Resources

CHtmlStream Members

CHtmlStream::Free

Called by the framework to free memory.

```
virtual void Free(
   BYTE* lpMem
);
```

Parameters

lpMem

Pointer to the memory to be deallocated.

Remarks

Override this function to implement custom memory deallocation. If you override this function, you will probably want to override Alloc and Realloc as well.

See Also

Reference

CHtmlStream Class
Hierarchy Chart
CHtmlStream::Alloc
CHtmlStream::Realloc

Other Resources

CHtmlStream Members

CHtmlStream::GetStreamSize

Call this member function to obtain the size of the HTML stream in bytes.

```
DWORD GetStreamSize( ) const;
```

Return Value

The length of the file.

See Also

Reference

CHtmlStream Class
Hierarchy Chart
CHtmlStream::m_nStreamSize
CHtmlStream::GrowStream

Other Resources

CHtmlStream Members

CHtmlStream::GrowStream

Called by the framework to expand memory.

```
virtual void GrowStream(
   DWORD dwNewLen
);
```

Parameters

dwNewLen

New size of the memory file.

Remarks

You can override it if you want to change how **CHtmlStream** expands its memory. The default implementation calls Realloc to increase an existing block (or Alloc to create a memory block), allocating memory in multiples of the *nGrowBytes* value specified in the constructor or Attach call.

See Also

Reference

CHtmlStream Class
Hierarchy Chart
CHtmlStream::m_nStreamSize
CHtmlStream::GetStreamSize

Other Resources

CHtmlStream Members

CHtmlStream::InitStream

Called by the framework to initialize a **CHtmlStream**.

```
virtual void InitStream( );
```

Remarks

Override **InitStream** to implement per-instance initialization. For example, override this function to specify HTML headers that you always need.

See Also

Reference

CHtmlStream Class
Hierarchy Chart
CHtmlStream::CHtmlStream

Other Resources

CHtmlStream Members

CHtmlStream::Memcpy

Called by the framework to transfer data to and from the stream.

```
virtual BYTE* Memcpy(
   BYTE* lpMemTarget,
   const BYTE* lpMemSource,
   UINT nBytes
);
```

Parameters

lpMemTarget

Pointer to the memory block into which the source memory will be copied.

lpMemSource

Pointer to the source memory block.

nBytes

Number of bytes to be copied.

Return Value

A copy of *lpMemTarget*.

Remarks

Override this function if you want to change the way that **CHtmlStream** does these memory copies.

See Also

Reference

CHtmlStream Class

Hierarchy Chart

Other Resources

CHtmlStream Members

CHtmlStream::Realloc

Called by the framework to reallocate memory.

```
virtual BYTE* Realloc(
   BYTE* lpMem,
   DWORD nBytes
);
```

Parameters

lpMem

A pointer to the memory block to be reallocated.

nBytes

New size for the memory block.

Return Value

A pointer to the memory block that was reallocated (and possibly moved), or **NULL** if the reallocation failed.

Remarks

Override this function to implement custom memory reallocation. If you override this function, you'll probably want to override Alloc and Free as well.

See Also

Reference

CHtmlStream Class
Hierarchy Chart
CHtmlStream::Free
CHtmlStream::Alloc

Other Resources

CHtmlStream Members

CHtmlStream::Reset

Called by the framework to empty a previously initialized **CHtmlStream** object.

```
virtual void Reset( );
```

Remarks

Override this member function to require a special action before emptying a **CHtmlStream** object.

See Also

Reference

CHtmlStream Class
Hierarchy Chart
CHtmlStream::Close
CHtmlStream::Abort

Other Resources

CHtmlStream Members

CHtmlStream::Write

Called by the framework to write data from a buffer to the stream associated with the **CHtmlStream** object.

```
virtual void Write(
   const void* lpBuf,
   UINT nCount
);
```

Parameters

lpBuf

A pointer to the user-supplied buffer that contains the data to be written to the stream.

nCount

The number of bytes to be transferred from the buffer.

See Also

Reference

CHtmlStream Class

Hierarchy Chart

CHtmlStream::operator <<

Other Resources

CHtmlStream Members

CHtmlStream Operators

For information about the operators in **CHtmlStream**, see CHtmlStream Members.

CHtmlStream::operator <<

The **CHtmlStream** insertion (**<<**) operator writes the specified string or integer to the HTML stream.

```
CHtmlStream& operator<<(
   LPCTSTR psz
);
CHtmlStream& operator<<(
   short int w
);
CHtmlStream& operator<<(
   long int dw
);
CHtmlStream& operator<<(
   const CHtmlStream& stream
);
CHtmlStream& operator<<(
   double d
);
CHtmlStream& operator<<(
   float f
);
CHtmlStream& operator<<(
   const CByteArray& array
);
CHtmlStream& operator<<(
   const CLongBinary& blob
);
```

Remarks

The string version of the operator writes the string without modification. The integer override versions of the operator format the value as decimal text before writing it.

You can use the **CHtmlStream&** override of this function to append the content of one HTML stream to another.

See Also

Reference

CHtmlStream Class
Hierarchy Chart
CHtmlStream::Write

Other Resources

CHtmlStream Members

CHtmlStream Data Members

For information about the data members in **CHtmlStream**, see CHtmlStream Members.

CHtmlStream::m_nStreamSize

Contains the size for an HTML stream.

```
DWORD m_nStreamSize;
```

Remarks

m_nStreamSize is a protected variable of type **UINT**. Only reference this variable if you want to override functions like Alloc and Free and GrowStream.

See Also

Reference

CHtmlStream Class
Hierarchy Chart
CHtmlStream::GetStreamSize
CHtmlStream::GrowStream

Other Resources

CHtmlStream Members

CHtmlView Class

Provides the functionality of the WebBrowser control within the context of MFC's document/view architecture.

```
class CHtmlView : public CFormView
```

Remarks

The WebBrowser control is a window in which the user can browse sites on the World Wide Web, as well as folders in the local file system and on a network. The WebBrowser control supports hyperlinking, Uniform Resource Locator (URL) navigation, and maintains a history list.

Using the CHtmlView Class in an MFC Application

In the standard MFC framework application (either SDI or MDI based), the view object is commonly derived from a specialized set of classes. These classes, all derived from **CView**, provide specialized functionality beyond that provided by **CView**.

Basing the application's view class on **CHtmlView** provides the view with the WebBrowser control. This effectively makes the application a web browser. The preferred method of creating a web browser-style application is to use the MFC Application Wizard, and specify **CHtmlView** as the view class. For more information on implementing and using the WebBrowser control within MFC applications, see Creating a Web Browser-Style Application.

Note
The WebBrowser ActiveX control (and therefore **CHtmlView**) is available only to programs running under Windows NT versi ons 4.0 or later, in which Internet Explorer 4.0 or later has been installed.

CHtmlView is designed for applications that access the Web (and/or HTML documents). The following **CHtmlView** member functions apply to the Internet Explorer application only. These functions will succeed on the WebBrowser control, but they will have no visible effect.

- GetAddressBar
- GetFullName
- GetStatusBar
- SetAddressBar
- SetFullScreen
- SetMenuBar
- SetStatusBar
- SetToolBar

Requirements

Header: afxhtml.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

MFCIE Sample: Demonstrates the MFC CHtmlView and CReBar Classes

Reference

CFormView Class

Hierarchy Chart

IWebBrowser2

Other Resources

CHtmlView Members

CHtmlView Members

Base Class Members

COBJECT Members

CCmdTarget Members

CWnd Members

CView Members

CScrollView Members

CFormView Members

Attributes

GetAddressBar	Determines if the Internet Explorer object's address bar is visible. (WebBrowser control ignores; Internet Explorer only.)
GetApplication	Retrieves an application object representing the application that contains the current instance of the Internet Explorer application.
GetBusy	Retrieves a value indicating whether a download or other activity is still in progress.
GetContainer	Retrieves the container of the WebBrowser control.
GetFullName	Retrieves the full name, including the path, of the resource displayed in the web browser. (WebBrowser control ignores; Internet Explorer only.)
GetFullScreen	Indicates whether the WebBrowser control is operating in full-screen mode or in normal window mode.
GetHeight	Retrieves the height of the Internet Explorer main window.
GetHtmlDocument	Retrieves the active HTML document.
GetLeft	Retrieves the screen coordinate of the left edge of the Internet Explorer main window.
GetLocationName	Retrieves the name of the resource that WebBrowser is currently displaying
GetLocationURL	Retrieves the URL of the resource that WebBrowser is currently displaying.
GetMenuBar	Retrieves a value that determines whether the menu bar is visible.
GetOffline	Retrieves a value that determines whether the control is offline.
GetParentBrowser	Retrieves a pointer to the **IDispatch** interface. For more information, see Implementing the IDispatch Interface.
GetReadyState	Retrieves the ready state of the web browser object.
GetRegisterAsBrowser	Indicates whether the WebBrowser control is registered as a top-level browser for target name resolution.
GetRegisterAsDropTarget	Indicates whether the WebBrowser control is registered as a drop target for navigation.

GetSilent	Indicates whether any dialog boxes can be shown.
GetSource	The HTML source code of the web page.
GetStatusBar	Indicates whether the Internet Explorer's status bar is visible. (WebBrowser control ignores; Internet Explorer only.)
GetTheaterMode	Indicates whether the WebBrowser control is in theater mode.
GetToolBar	Retrieves a value that determines whether the toolbar is visible.
GetTopLevelContainer	Retrieves a value indicating whether the current object is the top-level container of the WebBrowser control.
GetTop	Retrieves the screen coordinate of the top edge of the Internet Explorer main window.
GetType	Retrieves the type name of the document object.
GetVisible	Retrieves a value indicating whether the object is visible or hidden.
GetWidth	Retrieves the width of the Internet Explorer main window.
QueryStatusWB	Queries the status of a command being processed by the WebBrowser control.
SetAddressBar	Shows or hides the Internet Explorer object's address bar. (WebBrowser control ignores; Internet Explorer only.)
SetFullScreen	Sets a value to determine whether the control is operating in full-screen mode or in normal window mode. (WebBrowser control ignores; Internet Explorer only.)
SetHeight	Sets the height of the Internet Explorer main window.
SetLeft	Sets the horizontal position of the Internet Explorer main window.
SetMenuBar	Sets a value to determine whether the control's menu bar is visible. (WebBrowser control ignores; Internet Explorer only.)
SetOffline	Sets a value to determine whether the control is offline.
SetRegisterAsBrowser	Sets a value indicating whether the WebBrowser control is registered as a top-level browser for target name resolution.
SetRegisterAsDropTarget	Sets a value indicating whether the WebBrowser control is registered as a drop target for navigation.
SetSilent	Sets a value to determine whether the control will display dialog boxes.
SetStatusBar	Sets a value to determine whether the Internet Explorer's status bar is visible. (WebBrowser control ignores; Internet Explorer only.)
SetTheaterMode	Sets a value indicating whether the WebBrowser control is in theater mode.
SetToolBar	Sets a value to determine whether the control's toolbar is visible. (WebBrowser control ignores; Internet Explorer only.)

SetTop	Sets the vertical position of the Internet Explorer main window.
SetVisible	Sets a value indicating whether the object is visible or hidden.
SetWidth	Sets the width of the Internet Explorer main window.

Operations

ExecFormsCommand	Executes the specified command using the **IOleCommandTarget::Exec** method.
ExecWB	Executes a command.
GetProperty	Retrieves the current value of a property associated with the given object.
GoBack	Navigates to the previous item in the history list.
GoForward	Navigates to the next item in the history list.
GoHome	Navigates to the current home or start page.
GoSearch	Navigates to the current search page.
LoadFromResource	Loads a resource in the WebBrowser control.
Navigate2	Navigates to the resource identified by a URL, or to the file identified by a full path.
Navigate	Navigates to the resource identified by a URL.
PutProperty	Sets the value of a property associated with the given object.
QueryFormsCommand	
Refresh2	Reloads the current file and optionally prevents the "pragma:nocache" header from being sent.
Refresh	Reloads the current file.
Stop	Stops opening a file.

Overrides

Create	Creates the WebBrowser control.

Events

OnBeforeNavigate2	Called before a navigation occurs in the given WebBrowser (on either a window or frameset eleme nt).
OnCommandStateChange	Called to notify an application that the enabled state of a web browser command has changed.
OnDocumentComplete	Called to notify an application that a document has reached the **READYSTATE_COMPLETE** state.
OnDownloadBegin	Called to notify an application that a navigation operation is beginning.
OnDownloadComplete	Called when a navigation operation finished, was halted, or failed.
OnFullScreen	Called when the FullScreen property has changed.

OnMenuBar	Called when the MenuBar property has changed.
OnNavigateComplete2	Called after a navigation to a hyperlink completes (on either a window or frameset element).
OnNavigateError	Called by the framework if navigation to a hyperlink fails.
OnNewWindow2	Called when a new window is to be created for displaying a resource.
OnProgressChange	Called to notify an application that the progress of a download operation has been updated.
OnPropertyChange	Called to notify an application that the PutProperty method has changed the value of a property.
OnQuit	Called to notify an application that the Internet Explorer application is ready to quit. (Applies to Internet Explorer only)
OnStatusBar	Called when the StatusBar property has changed.
OnStatusTextChange	Called to notify an application that the text of the status bar associated with the WebBrowser control has changed.
OnTheaterMode	Called when the TheaterMode property has changed.
OnTitleChange	Called to notify an application if the title of a document in the WebBrowser control becomes available or changes.
OnToolBar	Called when the ToolBar property has changed.
OnVisible	Called when the window for the WebBrowser control should be shown/hidden.

DocHostUIHandler Overridables

OnDocWindowActivate	Called from the Internet Explorer or MSHTML implementation of IOleInPlaceActiveObject::OnDocWindowActivate, which notifies the active in-place object when the container's document window is activated or deactivated.
OnEnableModeless	Called to enable or disable modeless dialog boxes when the container creates or destroys a modal dialog box.
OnFilterDataObject	Called on the host by Internet Explorer or MSHTML to allow the host to replace Internet Explorer or MSHTML's data object.
OnFrameWindowActivate	Called from IOleInPlaceActiveObject::OnFrameWindowActivate to notify the object when the container's top-level frame window is activated or deactivated.
OnGetDropTarget	Called by Internet Explorer or MSHTML when it is being used as a drop target to allow the host to supply an alternative IDropTarget.
OnGetExternal	Called by Internet Explorer or MSHTML to obtain the host's **IDispatch** interface.
OnGetHostInfo	Retrieves the UI capabilities of the Internet Explorer or MSHTML host.
OnGetOptionKeyPath	Returns the registry key under which Internet Explorer or MSHTML stores user preferences.
OnHideUI	Called when Internet Explorer or MSHTML removes its menus and toolbars.

OnResizeBorder	Called from the Internet Explorer or MSHTML implementation of IOleInPlaceActiveObject::ResizeBorder, which alerts the object that it needs to resize its border space.
OnShowContextMenu	Called from Internet Explorer or MSHTML when it is about to show its context menu.
OnShowUI	Called before Internet Explorer or MSHTML displays its menus and toolbars.
OnTranslateAccelerator	Called by Internet Explorer or MSHTML when IOleInPlaceActiveObject::TranslateAccelerator or IOleControlSite::TranslateAccelerator is called to process menu accelerator-key messages from the c ontainer's message queue.
OnTranslateUrl	Called by Internet Explorer or MSHTML to allow the host an opportunity to modify the URL to be loa ded.
OnUpdateUI	Notifies the host that the command state has changed.

See Also

Reference

CHtmlView Class

Hierarchy Chart

CHtmlView Member Functions

For information about the member functions in **CHtmlView**, see CHtmlView Members.

CHtmlView::Create

Call this member function to create a WebBrowser control or container for the Internet Explorer executable.

```
virtual BOOL Create(
   LPCTSTR lpszClassName,
   LPCTSTR lpszWindowName,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID,
   CCreateContext* pContext = NULL
);
```

Parameters

lpszClassName

Points to a null-terminated character string that names the Windows class. The class name can be any name registered with the AfxRegisterWndClass global function or the **RegisterClass** Windows function. If **NULL**, uses the predefined default CFrameWnd attributes.

lpszWindowName

Points to a null-terminated character string that represents the window name.

dwStyle

Specifies the window style attributes. By default, the **WS_VISIBLE** and **WS_CHILD** Windows styles are set.

rect

A reference to a RECT structure specifying the size and position of the window. The **rectDefault** value allows Windows to specify the size and position of the new window.

pParentWnd

A pointer to the parent window of the control.

nID

The ID number of the view. By default, set to **AFX_IDW_PANE_FIRST**.

pContext

A pointer to a CCreateContext. **NULL** by default.

See Also

Reference

CHtmlView Class

Hierarchy Chart

Other Resources

CHtmlView Members

CHtmlView::CreateControlSite

Overridable used to create a control site instance to host a control on the form.

```
virtual BOOL CreateControlSite(
   COleControlContainer* pContainer,
   COleControlSite** ppSite,
   UINT nID,
   REFCLSID clsid
);
```

Parameters

pContainer

A pointer to a COleControlContainer object containing the control.

ppSite

A pointer to a pointer to a COleControlSite object, providing the site for the control.

nID

The identifier of the control to be hosted.

clsid

The CLSID of the control to be hosted

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

You can override this member function to return an instance of your own control site class.

See Also

Reference

CHtmlView Class

Hierarchy Chart

Other Resources

CHtmlView Members

CHtmlView::ExecFormsCommand

Executes the specified command using the **IOleCommandTarget::Exec** method.

```
HRESULT ExecFormsCommand(
   DWORD dwCommandID,
   VARIANT* pVarIn,
   VARIANT* pVarOut
);
```

Parameters

dwCommandID

The command to be executed. This command must belong to the **CMDSETID3_Forms3** group.

pVarIn

Pointer to a **VARIANT** structure containing input arguments. Can be **NULL**.

pVarOut

Pointer to a **VARIANT** structure to receive command output. Can be **NULL**.

Return Value

A standard **HRESULT** value. For a complete listing of possible values, see IOleCommandTarget::Exec in the *Platform SDK*.

Remarks

ExecFormsCommand implements the behavior of the IOleCommandTarget::Exec method.

See Also

Reference

CHtmlView Class

Hierarchy Chart

CHtmlView::QueryFormsCommand

Other Resources

CHtmlView Members

CHtmlView::ExecWB

Call this member function to execute a command in the WebBrowser or Internet Explorer.

```
void ExecWB(
   OLECMDID cmdID,
   OLECMDEXECOPT cmdexecopt,
   VARIANT* pvaIn,
   VARIANT* pvaOut
);
```

Parameters

cmdID

The command to execute.

cmdexecopt

The options set for executing the command.

pvaIn

A variant used for specifying command input arguments.

pvaOut

A variant used for specifying command output arguments.

Remarks

See IWebBrowser2::ExecWB in the *Platform SDK*.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::Stop

Other Resources

CHtmlView Members

CHtmlView::GetAddressBar

Call this member function to retrieve Internet Explorer's address bar.

```
BOOL GetAddressBar( ) const;
```

Return Value

Nonzero if the address bar is visible; otherwise zero.

Remarks

Applies to Internet Explorer. If you use this call with a WebBrowser control, it will return no error, but it will ignore this call.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::SetAddressBar
IWebBrowser2::get_AddressBar

Other Resources

CHtmlView Members

CHtmlView::GetApplication

Call this member function to retrieve the automation object supported by the application that contains the WebBrowser control.

```
LPDISPATCH GetApplication( ) const;
```

Return Value

A pointer to the **IDispatch** interface of the active document object. For more information, see Implementing the IDispatch Interface.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
IWebBrowser2::get_Application

Other Resources

CHtmlView Members

CHtmlView::GetBusy

Call this member function to determine whether the WebBrowser control is engaged in a navigation or downloading operation.

```
BOOL GetBusy( ) const;
```

Return Value

Nonzero if the web browser is busy; otherwise zero.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
IWebBrowser2::get_Busy

Other Resources

CHtmlView Members

CHtmlView::GetContainer

Call this member function to retrieve an object that evaluates to the container of the web browser.

```
LPDISPATCH GetContainer( ) const;
```

Return Value

A pointer to the **IDispatch** interface of the active document object. For more information, see Implementing the IDispatch Interface.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetTopLevelContainer
IWebBrowser2::get_Container

Other Resources

CHtmlView Members

CHtmlView::GetFullName

Call this member function to retrieve the full path of the file that Internet Explorer is currently displaying.

```
CString GetFullName( ) const;
```

Return Value

A CString object containing the path and name of the currently displayed file. If no path and filename exist, **GetFullName** returns an empty **CString**.

Remarks

Applies to Internet Explorer. If you use this call with a WebBrowser control, it will return no error, but it will ignore this call.

See Also

Reference

CHtmlView Class
Hierarchy Chart
IWebBrowser2::get_FullName

Other Resources

CHtmlView Members

CHtmlView::GetFullScreen

Call this member function to determine whether the WebBrowser control is operating in full-screen mode or in normal window mode.

```
BOOL GetFullScreen( ) const;
```

Return Value

Nonzero if the WebBrowser is operating in full-screen mode; otherwise zero.

Remarks

In full-screen mode, the Internet Explorer main window is maximized and the status bar, toolbar, menu bar, and title bar are hidden.

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::SetFullScreen
IWebBrowser2::get_FullScreen

Other Resources

CHtmlView Members

CHtmlView::GetHeight

Call this member function to retrieve the height, in pixels, of the WebBrowser control's frame window.

```
long GetHeight( ) const;
```

Return Value

The control's frame window height, in pixels.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::SetHeight

Other Resources

CHtmlView Members

CHtmlView::GetHtmlDocument

Call this member function to retrieve the HTML document for the active document.

```
LPDISPATCH GetHtmlDocument( ) const;
```

Return Value

A pointer to the **IDispatch** interface of the active document object. For more information, see Implementing the IDispatch Interface.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
IWebBrowser2::get_Document

Other Resources

CHtmlView Members

CHtmlView::GetLeft

Call this member function to retrieve the distance between the internal left edge of the WebBrowser control and the left edge of its container.

```
long GetLeft( ) const;
```

Return Value

The left-edge distance, in pixels.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::SetLeft
IWebBrowser2::get_Left

Other Resources

CHtmlView Members

CHtmlView::GetLocationName

Call this member function to get the name of the resource being displayed in the WebBrowser.

```
CString GetLocationName( ) const;
```

Return Value

A CString object containing the name of the resource currently displayed in the WebBrowser.

Remarks

If the resource is an HTML page on the World Wide Web, the name is the title of that page. If the resource is a folder or file on the network or local computer, the name is the UNC or full path of the folder or file.

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetLocationURL
IWebBrowser2::get_LocationName

Other Resources

CHtmlView Members

CHtmlView::GetLocationURL

Call this member function to retrieve the URL of the resource that the WebBrowser control is currently displaying.

```
CString GetLocationURL( ) const;
```

Return Value

A CString object containing the URL of the resource currently displayed in the WebBrowser.

Remarks

If the resource is a folder or file on the network or local computer, the name is the UNC or full path of the folder or file.

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetLocationName
IWebBrowser2::get_LocationURL

Other Resources

CHtmlView Members

CHtmlView::GetMenuBar

Call this member function to determine whether the menu bar is visible.

```
BOOL GetMenuBar( ) const;
```

Return Value

Nonzero if the menu bar is visible; otherwise zero.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::SetMenuBar
IWebBrowser2::get_MenuBar

Other Resources

CHtmlView Members

CHtmlView::GetOffline

Call this member function to determine whether the web browser is operating offline.

```
BOOL GetOffline( ) const;
```

Return Value

Nonzero if the web browser is currently offline; otherwise zero.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::SetOffline
IWebBrowser2::get_Offline

Other Resources

CHtmlView Members

CHtmlView::GetParentBrowser

Call this member function to retrieve a pointer to the parent object of the WebBrowser control.

```
LPDISPATCH GetParentBrowser( ) const;
```

Return Value

A pointer to the **IDispatch** interface of the object that is the parent of the WebBrowser control. For more information, see Implementing the IDispatch Interface.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
IWebBrowser2::get_Parent

Other Resources

CHtmlView Members

CHtmlView::GetProperty

Call this member function to get the value of the property currently associated with the control.

```
BOOL GetProperty(
   LPCTSTR lpszProperty,
   CString& strValue
);
COleVariant GetProperty(
   LPCTSTR lpszProperty
);
```

Parameters

lpszProperty

A pointer to a string containing the property to retrieve.

strValue

A reference to a CString object that receives the current value of the property.

Return Value

In the first version, nonzero if completed successfully; otherwise zero. In the second version, a COleVariant object.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::PutProperty
IWebBrowser2::GetProperty

Other Resources

CHtmlView Members

CHtmlView::GetReadyState

Call this member function to retrieve the ready state of the WebBrowser object.

```
READYSTATE GetReadyState( ) const;
```

Return Value

A READYSTATE value, as described in the *Platform SDK*.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class

Hierarchy Chart

IWebBrowser2::get_ReadyState

Other Resources

CHtmlView Members

CHtmlView::GetRegisterAsBrowser

Call this member function to determine whether the WebBrowser object is registered as a top-level browser for target name resolution.

```
BOOL GetRegisterAsBrowser( ) const;
```

Return Value

Nonzero if the browser is registered as a top-level browser; otherwise zero.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::SetRegisterAsBrowser
IWebBrowser2::get_RegisterAsBrowser

Other Resources

CHtmlView Members

CHtmlView::GetRegisterAsDropTarget

Call this member function to determine whether the WebBrowser control is registered as a drop target for navigation.

```
BOOL GetRegisterAsDropTarget( ) const;
```

Return Value

Nonzero if the browser is registered as a drop target; otherwise zero.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::SetRegisterAsDropTarget
IWebBrowser2::get_RegisterAsDropTarget

Other Resources

CHtmlView Members

CHtmlView::GetSilent

Call this member function to determine whether any dialog boxes can be shown in the WebBrowser control.

```
BOOL GetSilent( ) const;
```

Return Value

Nonzero if dialog boxes cannot be displayed from the WebBrowser control; otherwise zero.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::SetSilent
IWebBrowser2::get_Silent

Other Resources

CHtmlView Members

CHtmlView::GetSource

Call this member function to retrieve the HTML source code for the web page.

```
BOOL GetSource(
   CString& strRef
);
```

Parameters

refString

A CString that will hold the source code.

Return Value

Nonzero if successful; otherwise zero.

Remarks

This function is equivalent to the "View Source" command in Internet Explorer, except that the source code is returned in a **CString**.

See Also

Reference

CHtmlView Class

Hierarchy Chart

Other Resources

CHtmlView Members

CHtmlView::GetStatusBar

Call this member function to determine whether the WebBrowser control displays a status bar.

```
BOOL GetStatusBar( ) const;
```

Return Value

Nonzero if the status bar can be displayed; otherwise zero.

Remarks

Applies to Internet Explorer. If you use this call with a WebBrowser control, it will return no error, but it will ignore this call.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::SetStatusBar
IWebBrowser2::get_StatusBar

Other Resources

CHtmlView Members

CHtmlView::GetTheaterMode

Call this member function to determine whether the web browser is in theater mode.

```
BOOL GetTheaterMode( ) const;
```

Return Value

Nonzero if the web browser is in theater mode; otherwise zero.

Remarks

When the web browser is in theater mode, the browser main window fills the entire screen, a toolbar with a minimal set of navigational tools appears, and the status bar appears in the upper right-hand corner of the screen.

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::SetTheaterMode
CHtmlView::SetFullScreen
IWebBrowser2::get_TheaterMode

Other Resources

CHtmlView Members

CHtmlView::GetToolBar

Call this member function to determine whether the toolbar is visible.

```
int GetToolBar( ) const;
```

Return Value

A value indicating whether the toolbar is visible. Nonzero if toolbar is visible; otherwise zero.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::SetToolBar
IWebBrowser2::get_ToolBar

Other Resources

CHtmlView Members

CHtmlView::GetTop

Call this member function to retrieve the screen coordinate of the top edge of the WebBrowser control's main window.

```
long GetTop( ) const;
```

Return Value

Address of a variable that receives the screen coordinate of the main window's top edge.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::SetTop

Other Resources

CHtmlView Members

CHtmlView::GetTopLevelContainer

Call this member function to determine whether Internet Explorer is the top-level container of the WebBrowser control.

```
BOOL GetTopLevelContainer( ) const;
```

Return Value

Nonzero the container is the top-level container; otherwise zero.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetContainer
IWebBrowser2::get_TopLevelContainer

Other Resources

CHtmlView Members

CHtmlView::GetType

Call this member function to retrieve the type name of the contained active document.

```
CString GetType( ) const;
```

Return Value

A CString object containing the type name of the contained active document.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
IWebBrowser2::get_Type

Other Resources

CHtmlView Members

CHtmlView::GetVisible

Call this member function to determine if the contained object is visible.

```
BOOL GetVisible( ) const;
```

Return Value

Nonzero if the object is visible; otherwise zero.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::SetVisible
IWebBrowser2::get_Visible

Other Resources

CHtmlView Members

CHtmlView::GetWidth

Retrieves the width of the Internet Explorer main window.

```
long GetWidth( ) const;
```

Return Value

The current width of the window, in pixels.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::SetWidth

Other Resources

CHtmlView Members

CHtmlView::GoBack

Navigates backward one item in the history list.

```
void GoBack( );
```

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GoForward
CHtmlView::GoHome
IWebBrowser2::GoBack

Other Resources

CHtmlView Members

CHtmlView::GoForward

Navigates forward one item in the history list.

```
void GoForward( );
```

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GoBack
CHtmlView::GoHome
IWebBrowser2::GoForward

Other Resources

CHtmlView Members

CHtmlView::GoHome

Navigates to the current home or start page specified in the Internet Explorer Internet Options dialog box or the Internet Properties dialog box, accessed from the Control Panel.

```
void GoHome( );
```

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GoForward
CHtmlView::GoBack
IWebBrowser2::GoHome

Other Resources

CHtmlView Members

CHtmlView::GoSearch

Navigates to the current search page, as specified in the Internet Explorer Internet Options dialog box or the Internet Properties dialog box, accessed from the Control Panel.

```
void GoSearch( );
```

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GoHome
IWebBrowser2::GoSearch

Other Resources

CHtmlView Members

CHtmlView::LoadFromResource

Call this member function to load the specified resource into the WebBrowser control.

```
BOOL LoadFromResource(
   LPCTSTR lpszResource
);
BOOL LoadFromResource(
   UINT nRes
);
```

Parameters

lpszResource

A pointer to a string containing the name of the resource to load.

nRes

The ID of the buffer containing the name of the resource to load.

Return Value

Nonzero if successful; otherwise zero.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class

Hierarchy Chart

Other Resources

CHtmlView Members

CHtmlView::Navigate

Call this member function to navigate to the resource identified by a URL.

```
void Navigate(
   LPCTSTR URL,
   DWORD dwFlags = 0,
   LPCTSTR lpszTargetFrameName = NULL,
   LPCTSTR lpszHeaders = NULL,
   LPVOID lpvPostData = NULL,
   DWORD dwPostDataLen = 0
);
```

Parameters

URL

A caller-allocated string that contains the URL to navigate to, or the full path of the file to display.

dwFlags

The flags of a variable that specifies whether to add the resource to the history list, whether to read to or write from the cache, and whether to display the resource in a new window. The variable can be a combination of the values defined by the BrowserNavConstants enumeration.

lpszTargetFrameName

A pointer to a string that contains the name of the frame in which to display the resource.

lpszHeaders

A pointer to a value that specifies the HTTP headers to send to the server. These headers are added to the default Internet Explorer headers. The headers can specify such things as the action required of the server, the type of data being passed to the server, or a status code. This parameter is ignored if *URL* is not an HTTP URL.

lpvPostData

A pointer to the data to send with the HTTP POST transaction. For example, the POST transaction is used to send data gathered by an HTML form. If this parameter does not specify any post data, **Navigate** issues an HTTP GET transaction. This parameter is ignored if *URL* is not an HTTP URL.

dwPostDataLen

Data to send with the HTTP POST transaction. For example, the POST transaction is used to send data gathered by an HTML form. If this parameter does not specify any post data, **Navigate** issues an HTTP GET transaction. This parameter is ignored if *URL* is not an HTTP URL.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::Navigate2
IWebBrowser2::Navigate

Other Resources

CHtmlView Members

CHtmlView::Navigate2

Call this member function to navigate to the resource identified by a URL, or to the file identified by a full path.

```
void Navigate2(
   LPITEMIDLIST pIDL,
   DWORD dwFlags = 0,
   LPCTSTR lpszTargetFrameName = NULL
);
void Navigate2(
   LPCTSTR lpszURL,
   DWORD dwFlags = 0,
   LPCTSTR lpszTargetFrameName = NULL,
   LPCTSTR lpszHeaders = NULL,
   LPVOID lpvPostData = NULL,
   DWORD dwPostDataLen = 0
);
void Navigate2(
   LPCTSTR lpszURL,
   DWORD dwFlags,
   CByteArray& baPostedData,
   LPCTSTR lpszTargetFrameName = NULL,
   LPCTSTR lpszHeader = NULL
);
```

Parameters

pIDL

A pointer to an ITEMIDLIST structure.

dwFlags

The flags of a variable that specifies whether to add the resource to the history list, whether to read to or write from the cache, and whether to display the resource in a new window. The variable can be a combination of the values defined by the BrowserNavConstants enumeration.

lpszTargetFrameName

A pointer to a string that contains the name of the frame in which to display the resource.

lpszURL

A pointer to a string containing the URL.

lpvPostData

Data to send with the HTTP POST transaction. For example, the POST transaction is used to send data gathered by an HTML form. If this parameter does not specify any post data, **Navigate2** issues an HTTP GET transaction. This parameter is ignored if *URL* is not an HTTP URL.

dwPostDataLen

Length in bytes of the data pointed to by the *lpvPostData* parameter.

lpszHeaders

A pointer to a value that specifies the HTTP headers to send to the server. These headers are added to the default Internet Explorer headers. The headers can specify such things as the action required of the server, the type of data being passed to the server, or a status code. This parameter is ignored if *URL* is not an HTTP URL.

baPostedData

A reference to a CByteArray object.

Remarks

This member function extends the **Navigate** member function by supporting browsing on special folders, such as Desktop

and My Computer, that are represented by the parameter *pIDL*.

Applies to Internet Explorer and WebBrowser.

Example

```
void CMyHtmlView::OnGoToMicrosoft()
{
   CString str;

   str.Format("http://home.microsoft.com");
   Navigate2(str, 0, NULL);
}
```

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::Navigate
IWebBrowser2::Navigate2

Other Resources

CHtmlView Members

CHtmlView::OnBeforeNavigate2

This member function is called by the framework to cause an event to fire before a navigation occurs in the web browser.

```
virtual void OnBeforeNavigate2(
   LPCTSTR lpszURL,
   DWORD nFlags,
   LPCTSTR lpszTargetFrameName,
   CByteArray& baPostedData,
   LPCTSTR lpszHeaders,
   BOOL* pbCancel
);
```

Parameters

lpszURL

Pointer to a string containing the URL to navigate to.

nFlags

Reserved for future use.

lpszTargetFrameName

A string that contains the name of the frame in which to display the resource, or **NULL** if no named frame is targeted for the resource.

baPostedData

A reference to a **CByteArray** object containing the data to send to the server if the HTTP POST transaction is being used.

lpszHeaders

A pointer to a string containing additional HTTP headers to send to the server (HTTP URLs only). The headers can specify such things as the action required of the server, the type of data being passed to the server, or a status code.

pbCancel

A pointer to a cancel flag. An application can set this parameter to nonzero to cancel the navigation operation, or to zero to allow it to proceed.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::Navigate
CHtmlView::Navigate2
DWebBrowserEvents2::BeforeNavigate2

Other Resources

CHtmlView Members

CHtmlView::OnCommandStateChange

This member function is called by the framework to notify an application that the enabled state of a web browser command has changed.

```
virtual void OnCommandStateChange(
   long nCommand,
   BOOL bEnable
);
```

Parameters

nCommand

Identifier of the command whose enabled state has changed.

bEnable

Enabled state. This parameter is nonzero if the command is enabled, or zero if it is disabled.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetReadyState
DWebBrowserEvents2::CommandStateChange

Other Resources

CHtmlView Members

CHtmlView::OnDocumentComplete

This member function is called by the framework to notify an application that a document has reached the **READYSTATE_COMPLETE** state.

```
virtual void OnDocumentComplete(
   LPCTSTR lpszURL
);
```

Parameters

lpszURL

A pointer to a string that evaluates to the URL, UNC file name, or a PIDL (a pointer to an item identifier list) that was navigated to.

Remarks

Not every frame will fire this event, but each frame that fires an OnDownloadBegin event will fire a corresponding **OnDocumentComplete** event.

The URL indicated by *lpszURL* can be different from the URL that the browser was told to navigate to, because this URL is the canonicalized and qualified URL. For example, if an application specifies a URL of "www.microsoft.com" in a call to Navigate or Navigate2, the URL passed by **OnNavigateComplete2** will be "http://www.microsoft.com/". Also, if the server has redirected the browser to a different URL, the redirected URL will be reflected here.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetHtmlDocument
DWebBrowserEvents2::DocumentComplete

Other Resources

CHtmlView Members

CHtmlView::OnDocWindowActivate

Called from the Internet Explorer or MSHTML implementation of **IOleInPlaceActiveObject::OnDocWindowActivate**, which notifies the active in-place object when the container's document window is activated or deactivated.

```
virtual HRESULT OnDocWindowActivate(
   BOOL fActivate
);
```

Parameters

fActivate

Indicates the state of the document window. If this value is nonzero, the window is being activated. If this value is zero, the window is being deactivated.

Return Value

S_OK if successful, or an OLE-defined error code otherwise.

Remarks

Override **OnDocWindowActivate** to react to the **OnDocWindowActivate** notification from the Microsoft Web Browser control. See IDocHostUIHandler::OnDocWindowActivate in the *Platform SDK* for more information.

See Also

Reference

CHtmlView Class
Hierarchy Chart
IOleInPlaceActiveObject::OnDocWindowActivate

Other Resources

CHtmlView Members

CHtmlView::OnDownloadBegin

This member function is called by the framework to begin downloading a document.

```
virtual void OnDownloadBegin( );
```

Remarks

This event is fired shortly after the OnBeforeNavigate2 event, unless the navigation is canceled. Any animation or "busy" indication that the container needs to display should be connected to this event.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::OnDownloadComplete
CHtmlView::Navigate
CHtmlView::Navigate2
DWebBrowserEvents2::DownloadBegin

Other Resources

CHtmlView Members

CHtmlView::OnDownloadComplete

This member function is called by the framework to indicate that a navigation operation finished, was halted, or failed.

```
virtual void OnDownloadComplete( );
```

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::OnDownloadBegin
DWebBrowserEvents2::DownloadComplete

Other Resources

CHtmlView Members

CHtmlView::OnEnableModeless

Called when Internet Explorer or MSHTML displays modal UI.

```
virtual HRESULT OnEnableModeless(
   BOOL fEnable
);
```

Parameters

fEnable

Indicates if the host's modeless dialog boxes are enabled or disabled. If this value is nonzero, modeless dialog boxes are enabled. If this value is zero, modeless dialog boxes are disabled.

Return Value

S_OK if successful, or an OLE-defined error code otherwise.

Remarks

Enables or disables modeless dialog boxes when the container creates or destroys a modal dialog box. Override **OnEnableModeless** to react to the **EnableModeless** notification from the Microsoft Web Browser control. See IDocHostUIHandler::EnableModeless in the *Platform SDK* for more information.

See Also

Reference

CHtmlView Class
Hierarchy Chart
IOleInPlaceActiveObject::EnableModeless

Other Resources

CHtmlView Members

CHtmlView::OnFilterDataObject

Called on the host by Internet Explorer or MSHTML to allow the host to replace Internet Explorer or MSHTML's data object.

```
virtual HRESULT OnFilterDataObject(
   LPDATAOBJECT pDataObject,
   LPDATAOBJECT* ppDataObject
);
```

Parameters

pDataObject

Address of the IDataObject interface supplied by Internet Explorer or MSHTML.

ppDataObject

Address that receives the **IDataObject** interface pointer supplied by the host. The contents of this parameter should always be initialized to **NULL**, even if the method fails.

Return Value

S_OK if the data object is replaced, **S_FALSE** if the data object is not replaced, or an OLE-defined error code if an error occurs.

Remarks

Override **OnFilterDataObject** to react to the **FilterDataObject** notification from the Microsoft Web Browser control. See IDocHostUIHandler::FilterDataObject in the *Platform SDK* for more information.

See Also

Reference

CHtmlView Class

Hierarchy Chart

Other Resources

CHtmlView Members

CHtmlView::OnFrameWindowActivate

Called from IOleInPlaceActiveObject::OnFrameWindowActivate to notify the object when the container's top-level frame window is activated or deactivated.

```
virtual HRESULT OnFrameWindowActivate(
   BOOL fActivate
);
```

Parameters

fActivate

Indicates the state of the container's top-level frame window. If this value is nonzero, the window is being activated. If this value is zero, the window is being deactivated.

Return Value

S_OK if successful, or an OLE-defined error code otherwise.

Remarks

Override **OnFrameWindowActivate** to react to the **OnFrameWindowActivate** notification from the Microsoft Web Browser control. See IDocHostUIHandler::OnFrameWindowActivate in the *Platform SDK* for more information.

See Also

Reference

CHtmlView Class

Hierarchy Chart

Other Resources

CHtmlView Members

CHtmlView::OnFullScreen

This member function is called by the framework when the FullScreen property has changed.

```
virtual void OnFullScreen(
   BOOL bFullScreen
);
```

Parameters

bFullScreen

Nonzero if Internet Explorer is in full screen mode; zero otherwise.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetFullScreen
CHtmlView::SetFullScreen
DWebBrowserEvents2::OnFullScreen

Other Resources

CHtmlView Members

CHtmlView::OnGetDropTarget

Called by Internet Explorer or MSHTML when it is being used as a drop target to allow the host to supply an alternative **IDropTarget**.

```
virtual HRESULT OnGetDropTarget(
   LPDROPTARGET pDropTarget,
   LPDROPTARGET* ppDropTarget
);
```

Parameters

pDropTarget

IDropTarget Internet Explorer or MSHTML proposes to use.

ppDropTarget

Address of the **IDropTarget** that receives the **IDropTarget** interface pointer the host wants to provide.

Return Value

See IDocHostUIHandler::GetDropTarget in the *Platform SDK* for a list of return codes.

Remarks

Override **OnGetDropTarget** to react to the **GetDropTarget** notification from the Microsoft Web Browser control. See IDocHostUIHandler::GetDropTarget in the *Platform SDK* for more information.

See Also

Reference

CHtmlView Class

Hierarchy Chart

Other Resources

CHtmlView Members

CHtmlView::OnGetExternal

Called by Internet Explorer or MSHTML to obtain the host's **IDispatch** interface.

```
virtual HRESULT OnGetExternal(
   LPDISPATCH *lppDispatch
);
```

Parameters

lppDispatch

A pointer to the address that receives the **IDispatch** interface pointer of the host application. If the host exposes an Automation interface, it can provide a reference to Internet Explorer or MSHTML through this parameter. The contents of this parameter should always be initialized to **NULL**, even if the method fails.

Return Value

S_OK if successful, or an OLE-defined error code otherwise.

Remarks

Override **OnGetExternal** to react to the **GetExternal** notification from the Microsoft Web Browser control. See IDocHostUIHandler::GetExternal in the *Platform SDK* for more information.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::OnGetHostInfo

Other Resources

CHtmlView Members

CHtmlView::OnGetHostInfo

Retrieves the UI capabilities of the Internet Explorer or MSHTML host.

```
virtual HRESULT OnGetHostInfo(
   DOCHOSTUIINFO *pInfo
);
```

Parameters

pInfo

Address of a DOCHOSTUIINFO structure that receives the host's UI capabilities.

Return Value

S_OK if successful, or an OLE-defined error code otherwise.

Remarks

Override **OnGetHostInfo** to react to the **GetHostInfo** notification from the Microsoft Web Browser control. See IDocHostUIHandler::GetHostInfo in the *Platform SDK* for more information.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::OnGetExternal

Other Resources

CHtmlView Members

CHtmlView::OnGetOptionKeyPath

Call this member function to get the registry key under which Internet Explorer or MSHTML stores user preferences.

```
virtual HRESULT OnGetOptionKeyPath(
   LPOLESTR* pchKey,
   DWORD dwReserved
);
```

Parameters

pchKey

Address of an **LPOLESTR** that receives the registry subkey string where the host stores its default options. This subkey will be under the HKEY_CURRENT_USER key. Allocate this memory using CoTaskMemAlloc. The calling application is responsible for freeing this memory using CoTaskMemFree. This parameter should always be initialized to **NULL**, even if the method fails.

dwReserved

Reserved for future use. Not currently used.

Return Value

S_OK if successful, or **S_FALSE** otherwise. If **S_FALSE**, Internet Explorer or MSHTML will default to its own user options.

Remarks

Override **OnGetOptionKeyPath** to react to the **GetOptionKeyPath** notification from the Microsoft Web Browser control. See IDocHostUIHandler::GetOptionKeyPath in the *Platform SDK* for more information.

See Also

Reference

CHtmlView Class

Hierarchy Chart

Other Resources

CHtmlView Members

CHtmlView::OnHideUI

This member function is called by the framework when Internet Explorer or MSHTML removes its menus and toolbars.

```
virtual HRESULT OnHideUI( );
```

Return Value

S_OK if successful, or an OLE-defined error code otherwise.

Remarks

Override **OnHideUI** to react to the **HideUI** notification from the Microsoft Web Browser control. See IDocHostUIHandler::HideUI in the *Platform SDK* for more information.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::OnUpdateUI
CHtmlView::OnShowUI
CHtmlView::OnShowContextMenu

Other Resources

CHtmlView Members

CHtmlView::OnMenuBar

This member function is called by the framework when the MenuBar property has changed.

```
virtual void OnMenuBar(
   BOOL bMenuBar
);
```

Parameters

bMenuBar

Nonzero if the Internet Explorer menu bar is visible; zero otherwise.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetMenuBar
CHtmlView::SetMenuBar
DWebBrowserEvents2::OnMenuBar

Other Resources

CHtmlView Members

CHtmlView::OnNavigateComplete2

This member function is called by the framework after a navigation to a hyperlink completes (on either a window or frameset element).

```
virtual void OnNavigateComplete2(
   LPCTSTR strURL
);
```

Parameters

strURL

A string expression that evaluates to the URL, UNC file name, or PIDL (a pointer to an item identifier list) that was navigated to.

Remarks

The URL parameter can be a PIDL in the case of a shell name space entity for which there is no URL representation.

Note that the URL contained in *strURL* can be different from the URL that the browser was told to navigate to, because this URL is the canonicalized and qualified URL. For example, if an application specifies a URL of "www.microsoft.com" in a call to Navigate or Navigate2, the URL passed by **OnNavigateComplete2** will be "http://www.microsoft.com/". Also, if the server has redirected the browser to a different URL, the redirected URL will be reflected here.

See Also

Reference

CHtmlView Class
Hierarchy Chart
DWebBrowserEvents2::NavigateComplete2

Other Resources

CHtmlView Members

CHtmlView::OnNavigateError

Called by the framework if navigation to a hyperlink fails.

```
virtual void OnNavigateError(
   LPCTSTR lpszURL,
   LPCTSTR lpszFrame,
   DWORD dwError,
   BOOL *pbCancel
);
```

Parameters

lpszURL

The URL for which navigation failed.

lpszFrame

The name of the frame in which the resource is to be displayed, or NULL if no named frame was targeted for the resource.

dwError

An error status code, if available. For a list of the possible HRESULT and HTTP status codes, see NavigateError Event Status Codes.

pbCancel

Specifies whether to cancel the navigation to an error page or any further autosearch. If **TRUE** (the default), continue with navigation to an error page or autosearch; if **FALSE**, cancel navigation to an error page or autosearch.

Remarks

Override this method to provide custom navigation error handling.

For more information, see DWebBrowserEvents2::NavigateError

See Also

Reference

CHtmlView Class

Hierarchy Chart

Other Resources

CHtmlView Members

CHtmlView::OnNewWindow2

This member function is called by the framework when a new window is to be created for displaying a resource.

```
virtual void OnNewWindow2(
   LPDISPATCH* ppDisp,
   BOOL* Cancel
);
```

Parameters

ppDisp

A pointer to an interface pointer that, optionally, receives the **IDispatch** interface pointer of a new WebBrowser or Internet Explorer object. For more information, see Implementing the IDispatch Interface.

Cancel

A pointer to a cancel flag. An application can set this parameter to nonzero to cancel the navigation operation, or to zero to allow it to proceed.

Remarks

This event precedes the creation of a new window from within the WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
DWebBrowserEvents2::NewWindow2

Other Resources

CHtmlView Members

CHtmlView::OnProgressChange

This member function is called by the framework to notify an application that the progress of a download operation has been updated.

```
virtual void OnProgressChange(
   long nProgress,
   long nProgressMax
);
```

Parameters

nProgress

Amount of total progress to show, or -1 when progress is complete.

nProgressMax

Maximum progress value.

Remarks

The container can use the information provided by this event to display the number of bytes downloaded so far or to update a progress indicator.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetBusy
DWebBrowserEvents2::ProgressChange
CProgressCtrl Class

Other Resources

CHtmlView Members

CHtmlView::OnPropertyChange

This member function is called by the framework to notify an application that PutProperty has changed the value of a property.

```
virtual void OnPropertyChange(
   LPCTSTR lpszProperty
);
```

Parameters

lpszProperty

A pointer to a string containing the name of the property.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetProperty
DWebBrowserEvents2::PropertyChange

Other Resources

CHtmlView Members

CHtmlView::OnQuit

This member function is called by the framework to notify an application that the Internet Explorer application is ready to quit.

```
virtual void OnQuit( );
```

See Also

Reference

CHtmlView Class

Hierarchy Chart

DWebBrowserEvents2::OnQuit

Other Resources

CHtmlView Members

CHtmlView::OnResizeBorder

Called from the Internet Explorer or MSHTML implementation of IOleInPlaceActiveObject::ResizeBorder, which alerts the object that it needs to resize its border space.

```
virtual HRESULT OnResizeBorder(
   LPCRECT prcBorder,
   LPOLEINPLACEUIWINDOW pUIWindow,
   BOOL fFrameWindow
);
```

Parameters

prcBorder

New outer rectangle for border space.

pUIWindow

A pointer to the interface for the frame or document window object whose border has changed.

fFrameWindow

TRUE if the frame window is calling IOleInPlaceActiveObject::ResizeBorder, otherwise **FALSE**.

Return Value

S_OK if successful, or an OLE-defined error code otherwise.

Remarks

Override **OnResizeBorder** to react to the **ResizeBorder** notification from the Microsoft Web Browser control. See IDocHostUIHandler::ResizeBorder in the *Platform SDK* for more information.

See Also

Reference

CHtmlView Class

Hierarchy Chart

Other Resources

CHtmlView Members

CHtmlView::OnShowContextMenu

Called from Internet Explorer or MSHTML when it is about to show its context menu.

```
virtual HRESULT OnShowContextMenu(
   DWORD dwID,
   LPPOINT ppt,
   LPUNKNOWN pcmdtReserved,
   LPDISPATCH pdispReserved
);
```

Parameters

dwID

Identifier of the context menu to be displayed. See **IDocHostUIHandler::ShowContextMenu** in the *Platform SDK* for a list of values.

ppt

Screen coordinates for the menu.

pcmdtReserved

IOleCommandTarget interface used to query command status and execute commands on this object.

pdispReserved

IDispatch interface of the object at the screen coordinates. This allows a host to differentiate particular objects to provide more specific context.

Return Value

See IDocHostUIHandler::ShowContextMenu in the *Platform SDK* for a list of values.

Remarks

Override **OnShowContextMenu** to react to the **ShowContextMenu** notification from the Microsoft Web Browser control. See IDocHostUIHandler::ShowContextMenu in the *Platform SDK* for more information.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::OnShowUI
CHtmlView::OnHideUI
CHtmlView::OnUpdateUI

Other Resources

CHtmlView Members

CHtmlView::OnShowUI

Called before Internet Explorer or MSHTML displays its menus and toolbars.

```
virtual HRESULT OnShowUI(
   DWORD dwID,
   LPOLEINPLACEACTIVEOBJECT pActiveObject,
   LPOLECOMMANDTARGET pCommandTarget,
   LPOLEINPLACEFRAME pFrame,
   LPOLEINPLACEUIWINDOW pDoc
);
```

Parameters

dwID

Reserved for future use.

pActiveObject

IOleInPlaceActiveObject interface of the currently active object.

pCommandTarget

IOleCommandTarget interface of the object.

pFrame

IOleInPlaceFrame interface of the object. This is needed for menus and toolbars.

pDoc

IOleInPlaceUIWindow interface for the object. This is needed for toolbars.

Return Value

See IDocHostUIHandler::ShowUI in the *Platform SDK* for a list of values.

Remarks

Override **OnShowUI** to react to the **ShowUI** notification from the Microsoft Web Browser control. See IDocHostUIHandler::ShowUI in the *Platform SDK* for more information.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::OnUpdateUI
CHtmlView::OnHideUI
CHtmlView::OnShowContextMenu

Other Resources

CHtmlView Members

CHtmlView::OnStatusBar

This member function is called by the framework when the StatusBar property has changed.

```
virtual void OnStatusBar(
   BOOL bStatusBar
);
```

Parameters

bStatusBar

Nonzero if Internet Explorer's status bar is visible or zero otherwise.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetStatusBar
CHtmlView::SetStatusBar
DWebBrowserEvents2::OnStatusBar

Other Resources

CHtmlView Members

CHtmlView::OnStatusTextChange

This member function is called by the framework to notify an application that the text of the status bar associated with the WebBrowser control has changed.

```
virtual void OnStatusTextChange(
   LPCTSTR lpszText
);
```

Parameters

lpszText

A string that contains the new status bar text.

See Also

Reference

CHtmlView Class

Hierarchy Chart

DWebBrowserEvents2::StatusTextChange

Other Resources

CHtmlView Members

CHtmlView::OnTheaterMode

This member function is called by the framework when the TheaterMode property has changed.

```
virtual void OnTheaterMode(
   BOOL bTheaterMode
);
```

Parameters

bTheaterMode

Nonzero if Internet Explorer is in theater mode; zero otherwise.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetTheaterMode
CHtmlView::SetTheaterMode
DWebBrowserEvents2::OnTheaterMode

Other Resources

CHtmlView Members

CHtmlView::OnTitleChange

This member function is called by the framework to notify an application if the title of a document in the WebBrowser control becomes available or changes.

```
virtual void OnTitleChange(
   LPCTSTR lpszText
);
```

Parameters

lpszText

The new document title.

Remarks

For HTML, the title might change; while HTML is still downloading, the URL of the document is set as the title. After the real title (if there is one) is parsed from the HTML, the title is changed to reflect the actual title.

See Also

Reference

CHtmlView Class

Hierarchy Chart

DWebBrowserEvents2::TitleChange

Other Resources

CHtmlView Members

CHtmlView::OnToolBar

This member function is called by the framework when the ToolBar property has changed.

```
virtual void OnToolBar(
   BOOL bToolBar
);
```

Parameters

bToolBar

Nonzero if Internet Explorer's toolbar is visible or zero otherwise.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetToolBar
CHtmlView::SetToolBar
DWebBrowserEvents2::OnToolBar

Other Resources

CHtmlView Members

CHtmlView::OnTranslateAccelerator

Called by Internet Explorer or MSHTML when IOleInPlaceActiveObject::TranslateAccelerator or IOleControlSite::TranslateAccelerator is called to process menu accelerator-key messages from the container's message queue.

```
virtual HRESULT OnTranslateAccelerator(
   LPMSG lpMsg,
   const GUID* pguidCmdGroup,
   DWORD nCmdID
);
```

Parameters

lpMsg

Points to the message that might need to be translated.

pguidCmdGroup

Command group identifier.

nCmdID

Command identifier.

Return Value

S_OK if successful, or **S_FALSE** otherwise.

Remarks

Override **OnTranslateAccelerator** to react to the **TranslateAccelerator** notification from the Microsoft Web Browser control. See IDocHostUIHandler::TranslateAccelerator in the *Platform SDK* for more information.

See Also

Reference

CHtmlView Class

Hierarchy Chart

Other Resources

CHtmlView Members

CHtmlView::OnTranslateUrl

Called by Internet Explorer or MSHTML to allow the host an opportunity to modify the URL to be loaded.

```
virtual HRESULT OnTranslateUrl(
   DWORD dwTranslate,
   OLECHAR* pchURLIn,
   OLECHAR** ppchURLOut
);
```

Parameters

dwTranslate

Reserved for future use.

pchURLIn

Address of a string supplied by Internet Explorer or MSHTML that represents the URL to be translated.

ppchURLOut

Address of a string pointer that receives the address of the translated URL. The host allocates the buffer using the task memory allocator. The contents of this parameter should always be initialized to **NULL**, even if the URL is not translated or the method fails.

Return Value

S_OK if the URL was translated, **S_FALSE** if the URL was not translated, or an OLE-defined error code if an error occurred.

Remarks

Override **OnTranslateUrl** to react to the **TranslateUrl** notification from the Microsoft Web Browser control. See IDocHostUIHandler::TranslateUrl in the *Platform SDK* for more information.

See Also

Reference

CHtmlView Class

Hierarchy Chart

Other Resources

CHtmlView Members

CHtmlView::OnUpdateUI

Notifies the host that the command state has changed.

```
virtual HRESULT OnUpdateUI( );
```

Return Value

S_OK if successful, or an OLE-defined error code otherwise.

Remarks

The host should update the state of toolbar buttons. This method is called regardless of the return value from **ShowUI**. Override **OnUpdateUI** to react to the **UpdateUI** notification from the Microsoft Web Browser control.

See Also

Reference

CHtmlView Class
Hierarchy Chart
IDocHostUIHandler::UpdateUI
CHtmlView::OnShowUI
CHtmlView::OnHideUI
CHtmlView::OnShowContextMenu

Other Resources

CHtmlView Members

CHtmlView::OnVisible

This member function is called by the framework when the window for the WebBrowser should be shown or hidden.

```
virtual void OnVisible(
   BOOL bVisible
);
```

Parameters

bVisible

Nonzero if the object is visible or zero otherwise.

Remarks

This allows the object control host window to behave the same way the Internet Explorer window would behave.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::SetVisible
CHtmlView::GetVisible
DWebBrowserEvents2::OnVisible

Other Resources

CHtmlView Members

CHtmlView::PutProperty

Call this member function to set the property associated with a given object.

```
void PutProperty(
   LPCTSTR lpszProperty,
   const VARIANT& vtValue
);
void PutProperty(
   LPCTSTR lpszPropertyName,
   double dValue
);
void PutProperty(
   LPCTSTR lpszPropertyName,
   long lValue
);
void PutProperty(
   LPCTSTR lpszPropertyName,
   LPCTSTR lpszValue
);
void PutProperty(
   LPCTSTR lpszPropertyName,
   short nValue
);
```

Parameters

lpszProperty

A string containing the property to set.

vtValue

The new value of the property indicated by *lpszProperty*.

lpszPropertyName

A pointer to a string containing the name of the property to set.

dValue

The new value of the property.

lValue

The new value of the property.

lpszValue

A pointer to a string containing the new value of the property.

nValue

The new value of the property.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetProperty
IWebBrowser2::PutProperty

Other Resources

CHtmlView Members

CHtmlView::QueryFormsCommand

Queries for the status of one or more commands generated by user interface events.

```
HRESULT QueryFormsCommand(
   DWORD dwCommandID,
   BOOL* pbSupported,
   BOOL* pbEnabled,
   BOOL* pbChecked
);
```

Parameters

dwCommandID

The identifier of the command being queried for.

pbSupported

A pointer to a **BOOL** specifying if the command (identified by *dwCommandID*) is supported. If TRUE, the command is supported; otherwise FALSE.

pbEnabled

A pointer to a **BOOL** specifying if the command (identified by *dwCommandID*) is enabled. If TRUE, the command is supported; otherwise FALSE.

pbChecked

A pointer to a **BOOL** specifying if the command (identified by *dwCommandID*) is checked. If TRUE, the command is supported; otherwise FALSE.

Return Value

A standard **HRESULT** value. For a complete listing of possible values, see IOleCommandTarget::QueryStatus in the *Platform SDK*.

Remarks

QueryFormsCommand implements the behavior of the IOleCommandTarget::QueryStatus method.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::QueryFormsCommand
CHtmlView::QueryStatusWB

Other Resources

CHtmlView Members

CHtmlView::QueryStatusWB

Call this member function to query a command status.

```
OLECMDF QueryStatusWB(
   OLECMDID cmdID
) const;
```

Parameters

cmdID

The OLECMDID value of the command for which the caller needs status information.

Return Value

The address of the OLECMDF value that receives the status of the command.

Remarks

QueryStatusWB implements the behavior of the IOleCommandTarget::QueryStatus method.

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::ExecWB
IWebBrowser2::QueryStatusWB

Other Resources

CHtmlView Members

CHtmlView::Refresh

Reloads the URL or file that the web browser is currently displaying.

```
void Refresh( );
```

Remarks

Refresh contains no parameters for setting the refresh level.

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::Refresh2
IWebBrowser2::Refresh

Other Resources

CHtmlView Members

CHtmlView::Refresh2

Reloads the file that Internet Explorer is currently displaying.

```
void Refresh2(
   int nLevel
);
```

Parameters

nLevel

The address of the variable specifying the refresh level. The possible variables are defined in RefreshConstants, in the *Platform SDK*.

Remarks

Unlike Refresh, **Refresh2** contains a parameter that specifies the refresh level.

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
IWebBrowser2::Refresh2

Other Resources

CHtmlView Members

CHtmlView::SetAddressBar

Call this member function to show or hide the Internet Explorer object's address bar.

```
void SetAddressBar(
   BOOL bNewValue
);
```

Parameters

bNewValue

Nonzero to show address bar; otherwise zero.

Remarks

Applies to Internet Explorer. If you use this call with a WebBrowser control, it will return no error, but it will ignore this call.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetAddressBar
IWebBrowser2::put_AddressBar

Other Resources

CHtmlView Members

CHtmlView::SetFullScreen

Call this member function to set Internet Explorer to either full-screen or normal window mode.

```
void SetFullScreen(
   BOOL bNewValue
);
```

Parameters

bNewValue

Nonzero for full-screen mode; otherwise zero.

Remarks

In full-screen mode, the Internet Explorer main window is maximized and the status bar, toolbar, menu bar, and title bar are hidden.

Applies to Internet Explorer. If you use this call with a WebBrowser control, it will return no error, but it will ignore this call.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetFullScreen
CHtmlView::SetTheaterMode
IWebBrowser2::put_FullScreen

Other Resources

CHtmlView Members

CHtmlView::SetHeight

Call this member function to set the height of the Internet Explorer main window.

```
void SetHeight(
   long nNewValue
);
```

Parameters

nNewValue

The height, in pixels, of the main window.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetHeight
IWebBrowser2::put_Height

Other Resources

CHtmlView Members

CHtmlView::SetLeft

Sets the horizontal position of the Internet Explorer main window.

```
void SetLeft(
   long nNewValue
);
```

Parameters

nNewValue

The screen coordinate of the left edge of the main window.

See Also

Reference

CHtmlView Class

Hierarchy Chart

CHtmlView::GetLeft

IWebBrowser2::put_Left

Other Resources

CHtmlView Members

CHtmlView::SetMenuBar

Call this member function to show or hide the Internet Explorer menu bar.

```
void SetMenuBar(
   BOOL bNewValue
);
```

Parameters

bNewValue

Nonzero to show menu bar; otherwise zero.

Remarks

Applies to Internet Explorer. If you use this call with a WebBrowser control, it will return no error, but it will ignore this call.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetMenuBar
IWebBrowser2::put_MenuBar

Other Resources

CHtmlView Members

CHtmlView::SetOffline

Call this member function to set a value indicating whether the WebBrowser control is currently operating in offline mode.

```
void SetOffline(
   BOOL bNewValue
);
```

Parameters

bNewValue

Nonzero to read from the local cache; otherwise zero.

Remarks

In offline mode, the browser reads HTML pages from the local cache rather than from the source document.

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetOffline
IWebBrowser2::put_Offline

Other Resources

CHtmlView Members

CHtmlView::SetRegisterAsBrowser

Call this member function to set a value indicating whether the WebBrowser control is registered as a top-level browser for target name resolution.

```
void SetRegisterAsBrowser(
   BOOL bNewValue
);
```

Parameters

bNewValue

Determines whether Internet Explorer is registered as a top-level browser. If nonzero, the web browser is registered as a top-level browser; if zero, it is not a top-level browser. The default value is zero.

Remarks

A top-level browser is the browser set in the registry as the default browser.

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetRegisterAsBrowser
IWebBrowser2::put_RegisterAsBrowser

Other Resources

CHtmlView Members

CHtmlView::SetRegisterAsDropTarget

Call this member function to set a value indicating whether the WebBrowser control is registered as a drop target for navigation.

```
void SetRegisterAsDropTarget(
   BOOL bNewValue
);
```

Parameters

bNewValue

Determines if the WebBrowser control is registered as a drop target for navigation. If nonzero, the object is registered as a drop target; if zero, it is not a drop target.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetRegisterAsDropTarget
IWebBrowser2::put_RegisterAsDropTarget

Other Resources

CHtmlView Members

CHtmlView::SetSilent

Call this member function to set a value indicating whether any dialog boxes can be shown.

```
void SetSilent(
   BOOL bNewValue
);
```

Parameters

bNewValue

If nonzero, dialog boxes will not be displayed; if zero, dialog boxes will be displayed. The default value is zero.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetSilent
IWebBrowser2::put_Silent

Other Resources

CHtmlView Members

CHtmlView::SetStatusBar

Call this member function to display the status bar.

```
void SetStatusBar(
   BOOL bNewValue
);
```

Parameters

bNewValue

Nonzero if the status bar is visible; otherwise zero.

Remarks

Applies to Internet Explorer. If you use this call with a WebBrowser control, it will return no error, but it will ignore this call.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetStatusBar
IWebBrowser2::put_StatusBar

Other Resources

CHtmlView Members

CHtmlView::SetTheaterMode

Call this member function to set a value indicating whether the WebBrowser control is in theater mode.

```
void SetTheaterMode(
   BOOL bNewValue
);
```

Parameters

bNewValue

Nonzero to set the WebBrowser control to theater mode; otherwise zero. The default value is zero.

Remarks

When the web browser is in theater mode, the browser main window fills the entire screen, a toolbar with a minimal set of navigational tools appears, and the status bar appears in the upper right-hand corner of the screen.

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetTheaterMode
CHtmlView::GetFullScreen
CHtmlView::SetFullScreen

Other Resources

CHtmlView Members

CHtmlView::SetToolBar

Call this member function to show or hide the Internet Explorer toolbar.

```
void SetToolBar(
   int nNewValue
);
```

Parameters

nNewValue

Indicates whether to display the toolbar. Nonzero if the toolbar is to be displayed; otherwise zero.

Remarks

Applies to Internet Explorer. If you use this call with a WebBrowser control, it will return no error, but it will ignore this call.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetToolBar
IWebBrowser2::put_ToolBar

Other Resources

CHtmlView Members

CHtmlView::SetTop

Call this member function to set the distance between the internal top edge of the WebBrowser control and the top edge of its container

```
void SetTop(
   long nNewValue
);
```

Parameters

nNewValue

The screen coordinate of the top edge of the main window.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetTop
IWebBrowser2::put_Top

Other Resources

CHtmlView Members

CHtmlView::SetVisible

Call this member function to set the visibility state of the WebBrowser control.

```
void SetVisible(
   BOOL bNewValue
);
```

Parameters

bNewValue

Nonzero if the control is visible; otherwise zero.

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::GetVisible
IWebBrowser2::put_Visible

Other Resources

CHtmlView Members

CHtmlView::SetWidth

Sets the width of the Internet Explorer main window.

```
void SetWidth(
   long nNewValue
);
```

Parameters

nNewValue

The width, in pixels, of the Internet Explorer main window.

See Also

Reference

CHtmlView Class

Hierarchy Chart

CHtmlView::GetWidth

Other Resources

CHtmlView Members

CHtmlView::Stop

Call this member function to cancel any pending navigation or download operation and stop any dynamic page elements, such as background sounds and animations.

```
void Stop( );
```

Remarks

Applies to Internet Explorer and WebBrowser.

See Also

Reference

CHtmlView Class
Hierarchy Chart
CHtmlView::ExecWB
IWebBrowser2::Stop

Other Resources

CHtmlView Members

CHttpArg Structure

Stores the information about a single parameter-value pair sent to your HTTP server extension.

```
struct CHttpArg {
   LPCTSTR m_pstrRaw;
   LPCTSTR m_pstrParam;
   LPCTSTR m_pstrValue;
};
```

Remarks

CHttpArg does not have a base class.

A **CHttpArg** structure has three data members that contain the unparsed raw data, the parsed parameter, and the parsed argument for the client's request.

See CHttpArgList for more detailed information.

Requirements

Header: afxisapi.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart

Concepts

How Do I Use the MFC ISAPI Classes Without Linking to the MFC DLL?

Other Resources

CHttpArg Members

CHttpArg Members

Data Members

m_pstrParam	A string containing a parameter from a URL.
m_pstrRaw	A string containing the raw data in a URL.
m_pstrValue	A string containing the value assigned to a parameter in a URL.

See Also

Reference

CHttpArg Structure

Hierarchy Chart

CHttpArg Data Members

For information about the data members in **CHttpArg**, see CHttpArg Members.

CHttpArg::m_pstrParam

A string containing the parameter of the argument.

```
LPCTSTR m_pstrParam;
```

Remarks

The parameter will be the entire argument if no equal sign (=) is present. If an equal sign is present, the parameter is everything to the left of the equal sign.

Example

```
// given the following request URL:
http://www.website/result.dll?Arg1=A&Arg2
//The m_pstrParam for the first parameter points to "Arg1".
//The m_pstrParam for the second parameter points to "Arg2".
```

See Also

Reference

CHttpArg Structure
Hierarchy Chart
CHttpArg::m_pstrRaw
CHttpArg::m_pstrValue
CHttpArgList Class

Other Resources

CHttpArg Members

CHttpArg::m_pstrRaw

A string containing the raw text extracted from the URL.

```
LPCTSTR m_pstrRaw;
```

Example

```
// given the following request URL:
http://www.website/result.dll?Arg1=A&Arg2
//The m_pstrRaw for the first parameter points to "Arg1=A".
//The m_pstrRaw for the second parameter points to "Arg2".
```

See Also

Reference

CHttpArg Structure
Hierarchy Chart
CHttpArg::m_pstrValue
CHttpArg::m_pstrParam
CHttpArgList Class

Other Resources

CHttpArg Members

CHttpArg::m_pstrValue

Contains a string with the value assigned to a parameter.

```
LPCTSTR m_pstrValue;
```

Example

```
// given the following request URL:
http://www.website/result.dll?Arg1=A&Arg2
//The m_pstrValue for the first parameter points to "A".
//The m_pstrValue for the second parameter points to the empty string, "".
```

See Also

Reference

CHttpArg Structure
Hierarchy Chart
CHttpArg::m_pstrRaw
CHttpArg::m_pstrParam
CHttpArgList Class

Other Resources

CHttpArg Members

CHttpArgList Class

Contains a collection of CHttpArg structures and provides functions for enumerating those **CHttpArg** structures.

```
class CHttpArgList
```

Remarks

CHttpArgList does not have a base class.

A **CHttpArgList** is sent to a parse map function, which uses the **ITS_ARGLIST** in the ON_PARSE_COMMAND macro for the parameter type.

CHttpServer constructs a single **CHttpArgList** containing zero, one, or many **CHttpArg** objects. The **CHttpArg** objects represent, in order, the individual parameters passed along in the URL which activated your **CHttpServer** command.

For example, the following URL:

```
http://mooseboy/yourext.dll&Arg1=hockey&Arg2&Arg3=beer+nuts
```

indicates the user's selections for three arguments: "hockey" for the first argument, nothing for the second argument, and a combination of "beer" and "nuts" for the third argument. The user's selection of nothing for the second argument results in CHttpArg::m_pstrValue containing an empty string. The user's two selections for the third argument results in an escaped string, using the RFC escape character +. See CHttpArgList::Unescape for more information about escaped strings.

The above URL results in a structure of **CHttpArgList** and **CHttpArg** objects diagrammed below:

CHttpArgList and CHttpArg Objects

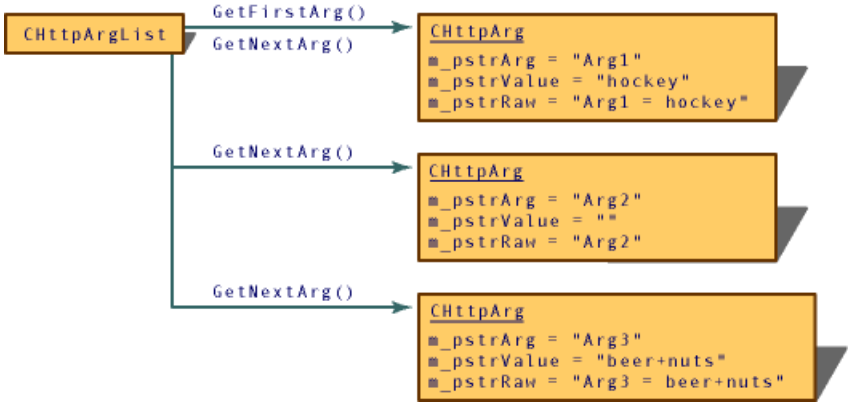

CHttpArg objects can be constructed from arguments having the same name, making **ITS_ARGLIST** an ideal candidate for server extensions that handle pages containing multiple selection list boxes.

Using **CHttpArg** structures and a **CHttpArgList** object improves the flexibility of your parse map macros. The class is well-suited for clients that offer a variable list of arguments, or situations where you want MFC to do some of the parsing but not all of it. See Managing Required and Optional Parameters for more information about parsing parameters.

Requirements

Header: afxisapi.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart

Concepts

How Do I Use the MFC ISAPI Classes Without Linking to the MFC DLL?

Other Resources

CHttpArgList Members

CHttpArgList Members

Operations

GetFirstArgPosition	Retrieves the position of the first **CHttpArg** structure.
GetNextArg	Retrieves the next **CHttpArg** structure. Used for iterating through the arguments.
Unescape	Removes RFC-compatible escape codes from the passed string.

See Also

Reference

CHttpArgList Class

Hierarchy Chart

CHttpArgList Member Functions

For information about the member functions in **CHttpArgList**, see CHttpArgList Members.

CHttpArgList::GetFirstArgPosition

Call this member function to get the first CHttpArg structure for the first argument in the URL.

```
POSITION GetFirstArgPosition( ) const;
```

Return Value

The position of the first element in the argument list, or **NULL** if there are no elements in the list.

Example

```
// this parse map handling function dumps an HTML table showing the raw
// argument, the parameter, and the value for each of the arguments it
// is passed.

void CYourExtension::TableCellStringOrNull(CHttpServerContext* pCtxt, LPCTSTR pstr)
{
   *pCtxt << "   <TD>";
   if (pstr == NULL)
      *pCtxt << "<i>null</i>";
   else
      *pCtxt << pstr;
   *pCtxt << "</TD>\r\n";
}

void CYourExtension::dumpArgList(CHttpServerContext* pCtxt, CHttpArgList* pArgList)
{
   StartContent(pCtxt);
   WriteTitle(pCtxt);

   POSITION pos = pArgList->GetFirstArgPosition();
   if (pos == NULL)
      *pCtxt << "No arguments!\n";
   else
   {
      *pCtxt << "<TABLE BORDER=3>\r\n";
      *pCtxt << "<TR><TD><B>Parameter</B></TD>";
      *pCtxt << "<TD><B>Value</B></TD>";
      *pCtxt << "<TD><B>Unescaped</B></TD>";
      *pCtxt << "<TD><B>Raw</B></TD></TR>\r\n";

      while (pos != NULL)
      {
         CHttpArg* pCurrentArg = pArgList->GetNextArg(pos);

         *pCtxt << " <TR>\r\n";

         TableCellStringOrNull(pCtxt, pCurrentArg->m_pstrParam);
         TableCellStringOrNull(pCtxt, pCurrentArg->m_pstrValue);
         TableCellStringOrNull(pCtxt, pCurrentArg->m_pstrRaw);

         char* pch = strdup(pCurrentArg->m_pstrRaw);
         CHttpArgList::Unescape(pch);
         TableCellStringOrNull(pCtxt, pch);
         free(pch);

         *pCtxt << " </TR>\r\n";
      }
      *pCtxt << "</TABLE>\r\n";
   }

   EndContent(pCtxt);
}
```

See Also

Reference

CHttpArgList Class

Hierarchy Chart

CHttpArgList::GetNextArg

Other Resources

CHttpArgList Members

CHttpArgList::GetNextArg

Call this member function to retrieve the **CHttpArg** structure of the current element.

```
CHttpArg* GetNextArg(
   POSITION& pos
) const;
```

Parameters

pos

The position of an element.

Return Value

A pointer to a CHttpArg structure containing the currently enumerated argument.

Remarks

By calling **GetNextArg** in a loop until it returns **NULL**, you can enumerate all of the **CHttpArg** objects in the list.

Example

See the example for CHttpArgList::GetFirstArgPosition.

See Also

Reference

CHttpArgList Class
Hierarchy Chart
CHttpArgList::GetFirstArgPosition

Other Resources

CHttpArgList Members

CHttpArgList::Unescape

Call this member function to decode a string encoded with escape characters according to the Internet Architecture Board's (IAB) "Request For Comment" (RFC) rules for encoding URLs.

```
static void Unescape(
   LPTSTR pstrChunk
);
```

Parameters

pstrChunk

A fragment of, or a complete, URL string.

Remarks

This member function will convert the following escaped string:

```
"some+string+with+a+tilde (%7F)"
```

to the following text:

```
"some string with a tilde (~)"
```

The conversion is done in-place because the unescaped string is always equal to or shorter in length than the escaped string.

Example

See the example for CHttpArgList::GetFirstArgPosition.

See Also

Reference

CHttpArgList Class

Hierarchy Chart

Other Resources

CHttpArgList Members

CHttpConnection Class

This topic has been updated for Visual Studio 2005 SP1.

Manages your connection to an HTTP server.

```
class CHttpConnection : public CInternetConnection
```

Remarks

HTTP is one of three Internet server protocols implemented by the MFC WinInet classes.

The class **CHttpConnection** contains a constructor and one member function, OpenRequest, that manages connections to a server with an HTTP protocol.

To communicate with an HTTP server, you must first create an instance of CInternetSession, and then create a CHttpConnection object. You never create a **CHttpConnection** object directly; rather, call CInternetSession::GetHttpConnection, which creates the **CHttpConnection** object and returns a pointer to it.

To learn more about how **CHttpConnection** works with the other MFC Internet classes, see the article Internet Programming with WinInet. For more information about connecting to servers using the other two supported Internet protocols, gopher and FTP, see the classes CGopherConnection and CFtpConnection.

Requirements

Header: afxinet.h

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

See Also

Reference

CInternetConnection Class
Hierarchy Chart
CInternetConnection Class
CHttpFile Class

Other Resources

CHttpConnection Members

CHttpConnection Members

Base Class Members

CObject Members

CInternetConnection Members

Construction

CHttpConnection	Creates a **CHttpConnection** object.

Operations

OpenRequest	Opens an HTTP request.

See Also

Reference

CHttpConnection Class
Hierarchy Chart
CFtpConnection Class
CGopherConnection Class
CInternetConnection Class
CHttpFile Class

CHttpConnection Member Functions

For information about the member functions in **CHttpConnection**, see CHttpConnection Members.

CHttpConnection::CHttpConnection

This member function is called to construct a **CHttpConnection** object.

```
CHttpConnection(
   CInternetSession* pSession,
   HINTERNET hConnected,
   LPCTSTR pstrServer,
   DWORD_PTR dwContext
);
CHttpConnection(
   CInternetSession* pSession,
   LPCTSTR pstrServer,
   INTERNET_PORT nPort = INTERNET_INVALID_PORT_NUMBER,
   LPCTSTR pstrUserName = NULL,
   LPCTSTR pstrPassword = NULL,
   DWORD_PTR dwContext = 1
);
CHttpConnection(
   CInternetSession* pSession,
   LPCTSTR pstrServer,
   DWORD dwFlags,
   INTERNET_PORT nPort = INTERNET_INVALID_PORT_NUMBER,
   LPCTSTR pstrUserName = NULL,
   LPCTSTR pstrPassword = NULL,
   DWORD_PTR dwContext = 1
);
```

Parameters

pSession

A pointer to a CInternetSession object.

hConnected

A handle to an Internet connection.

pstrServer

A pointer to a string containing the server name.

dwContext

The context identifier for the **CInternetConnection** object. See **Remarks** for more information about *dwContext*.

nPort

The number that identifies the Internet port for this connection.

pstrUserName

Pointer to a null-terminated string that specifies the name of the user to log in. If **NULL**, the default is anonymous.

pstrPassword

A pointer to a null-terminated string that specifies the password to use to log in. If both *pstrPassword* and *pstrUserName* are **NULL**, the default anonymous password is the user's email name. If *pstrPassword* is **NULL** (or an empty string) but *pstrUserName* is not **NULL**, a blank password is used. The following table describes the behavior for the four possible settings of *pstrUserName* and *pstrPassword*:

pstrUserName	***pstrPassword***	**Username sent to FTP server**	**Password sent to FTP server**
NULL or " "	**NULL** or " "	"anonymous"	User's email name
Non-**NULL** String	**NULL** or " "	*pstrUserName*	" "

NULL Non-**NULL** String	**ERROR**	**ERROR**	
Non-**NULL** String	Non-**NULL** String	*pstrUserName*	*pstrPassword*

dwFlags

Any combination of the **INTERNET_ FLAG_*** flags. See the table in the **Remarks** section of CHttpConnection::OpenRequest for a description of *dwFlags* values.

Remarks

You never create a **CHttpConnection** directly. Rather, you create an object by calling CInternetSession::GetHttpConnection.

See Also

Reference

CHttpConnection Class
Hierarchy Chart
CInternetSession::GetHttpConnection
CFtpConnection Class
CGopherConnection Class
CInternetConnection Class

Other Resources

CHttpConnection Members

CHttpConnection::OpenRequest

This topic has been updated for Visual Studio 2005 SP1.

Call this member function to open an HTTP connection.

```
CHttpFile* OpenRequest(
   LPCTSTR pstrVerb,
   LPCTSTR pstrObjectName,
   LPCTSTR pstrReferer = NULL,
   DWORD_PTR dwContext = 1,
   LPCTSTR* ppstrAcceptTypes = NULL,
   LPCTSTR pstrVersion = NULL,
   DWORD dwFlags = INTERNET_FLAG_EXISTING_CONNECT
);
CHttpFile* OpenRequest(
   int nVerb,
   LPCTSTR pstrObjectName,
   LPCTSTR pstrReferer = NULL,
   DWORD_PTR dwContext = 1,
   LPCTSTR* ppstrAcceptTypes = NULL,
   LPCTSTR pstrVersion = NULL,
   DWORD dwFlags = INTERNET_FLAG_EXISTING_CONNECT
);
```

Parameters

pstrVerb

A pointer to a string containing the verb to use in the request. If **NULL**, "GET" is used.

pstrObjectName

A pointer to a string containing the target object of the specified verb. This is generally a filename, an executable module, or a search specifier.

pstrReferer

A pointer to a string that specifies the address (URL) of the document from which the URL in the request (*pstrObjectName*) was obtained. If **NULL**, no HTTP header is specified.

dwContext

The context identifier for the **OpenRequest** operation. See **Remarks** for more information about *dwContext*.

ppstrAcceptTypes

A pointer to a null-terminated array of **LPCTSTR** pointers to strings indicating content types accepted by the client. If *ppstrAcceptTypes* is **NULL**, the servers interpret that the client only accepts documents of type "text/*" (that is, only text documents and not pictures or other binary files). The content type is equivalent to the CGI variable CONTENT_TYPE, which identifies the type of data for queries that have attached information, such as HTTP POST and PUT.

pstrVersion

A pointer to a string defining the HTTP version. If **NULL**, "HTTP/1.0" is used.

dwFlags

Any combination of the INTERNET_ FLAG_* flags. See the **Remarks** for a description of possible *dwFlags* values.

nVerb

A number associated with the HTTP request type. Can be one of the following:

HTTP request type	***nVerb* value**
HTTP_VERB_POST	0

HTTP_VERB_GET	1
HTTP_VERB_HEAD	2
HTTP_VERB_PUT	3
HTTP_VERB_LINK	4
HTTP_VERB_DELETE	5
HTTP_VERB_UNLINK	6

Return Value

A pointer to the CHttpFile object requested.

Remarks

dwFlags can be one of the following:

Internet flag	Description
INTERNET_FLAG_RELOAD	Forces a download of the requested file, object, or directory listing from the origin server, not from the cache.
INTERNET_FLAG_DONT_CACHE	Does not add the returned entity to the cache.
INTERNET_FLAG_MAKE_PERSISTENT	Adds the returned entity to the cache as a persistent entity. This means that standard cache cleanup, consistency checking, or garbage collection cannot remove this item from the cache.
INTERNET_FLAG_SECURE	Uses secure transaction semantics. This translates to using SSL/PCT and is only meaningful in HTTP requests
INTERNET_FLAG_NO_AUTO_REDIRECT	Used only with HTTP, specifies that redirections should not be automatically handled in CHttpFile::SendRequest.

Override the *dwContext* default to set the context identifier to a value of your choosing. The context identifier is associated with this specific operation of the **CHttpConnection** object created by its CInternetSession object. The value is returned to CInternetSession::OnStatusCallback to provide status on the operation with which it is identified. See the article Internet First Steps: WinInet for more information about the context identifier.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

In the Smart Device version of this method, the default value of **dwFlags** is 0. The desktop version's default value is **INTERNET_FLAG_EXISTING_CONNECT**.

See Also

Reference

CHttpConnection Class
Hierarchy Chart
CHttpFile Class
CInternetSession Class
CFtpConnection Class
CGopherConnection Class
CInternetConnection Class

Other Resources

CHttpConnection Members

CHttpFile Class

This topic has been updated for Visual Studio 2005 SP1.

Provides the functionality to request and read files on an HTTP server.

```
class CHttpFile : public CInternetFile
```

Remarks

If your Internet session reads data from an HTTP server, you must create an instance of **CHttpFile**.

To learn more about how **CHttpFile** works with the other MFC Internet classes, see the article Internet Programming with WinInet.

Requirements

Header: afxinet.h

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

The Smart Device version of CHttpFile Class does not support the following methods:

- CHttpFile::SendRequestEx
- CHttpFile::EndRequest

See Also

Reference

CInternetFile Class
Hierarchy Chart
CInternetFile Class
CGopherFile Class
CHttpConnection Class

Concepts

CHttpFile Members

CHttpFile Members

This topic has been updated for Visual Studio 2005 SP1.

Base Class Members

CObject Members

CFile Members

CStdioFile Members

CInternetFile Members

Construction

CHttpFile	Creates a **CHttpFile** object.

Operations

AddRequestHeaders	Adds headers to the request sent to an HTTP server.
EndRequest	Ends a request sent to an HTTP server with the SendRequestEx member function.
GetFileURL	Gets the URL for the specified file.
GetObject	Gets the target object of the verb in a request to an HTTP server.
GetVerb	Gets the verb that was used in a request to an HTTP server.
QueryInfo	Returns the response or request headers from the HTTP server.
QueryInfoStatusCode	Retrieves the status code associated with an HTTP request and places it in the supplied *dwStatusCode* parameter.
SendRequest	Sends a request to an HTTP server.
SendRequestEx	Sends a request to an HTTP server using the Write or WriteString methods of **CInternetFile**.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

The Smart Device version of CHttpFile Class does not support the following methods:

- CHttpFile::SendRequestEx
- CHttpFile::EndRequest

See Also

Reference

CHttpFile Class
Hierarchy Chart
CInternetFile Class
CGopherFile Class
CHttpConnection Class

CHttpFile Member Functions

For information about the member functions in **CHttpFile**, see CHttpFile Members.

CHttpFile::AddRequestHeaders

Call this member function to add one or more HTTP request headers to the HTTP request handle.

```
BOOL AddRequestHeaders(
   LPCTSTR pstrHeaders,
   DWORD dwFlags = HTTP_ADDREQ_FLAG_ADD_IF_NEW,
   int dwHeadersLen = -1
);
BOOL AddRequestHeaders(
   CString& str,
   DWORD dwFlags = HTTP_ADDREQ_FLAG_ADD_IF_NEW
);
```

Parameters

pstrHeaders

A pointer to a string containing the header or headers to append to the request. Each header must be terminated by a CR/LF pair.

dwFlags

Modifies the semantics of the new headers. Can be one of the following:

- **HTTP_ADDREQ_FLAG_COALESCE** Merges headers of the same name, using the flag to add the first header found to the subsequent header. For example, "Accept: text/*" followed by "Accept: audio/*" results in the formation of the single header "Accept: text/*, audio/*". It is up to the calling application to ensure a cohesive scheme with respect to data received by requests sent with coalesced or separate headers.
- **HTTP_ADDREQ_FLAG_REPLACE** Performs a remove and add to replace the current header. The header name will be used to remove the current header, and the full value will be used to add the new header. If the header-value is empty and the header is found, it is removed. If not empty, the header-value is replaced.
- **HTTP_ADDREQ_FLAG_ADD_IF_NEW** Only adds the header if it does not already exist. If one exists, an error is returned.
- **HTTP_ADDREQ_FLAG_ADD** Used with REPLACE. Adds the header if it doesn't exist.

dwHeadersLen

The length, in characters, of *pstrHeaders*. If this is -1L, then *pstrHeaders* is assumed to be zero-terminated and the length is computed.

str

A reference to a CString object containing the request header or headers to be added.

Return Value

Nonzero if successful; otherwise 0. If the call fails, the Win32 function GetLastError may be called to determine the cause of the error.

Remarks

AddRequestHeaders appends additional, free-format headers to the HTTP request handle. It is intended for use by sophisticated clients who need detailed control over the exact request sent to the HTTP server.

Note
The application can pass multiple headers in *pstrHeaders* or *str* for an **AddRequestHeaders** call using **HTTP_ADDREQ_FLAG_ADD** or **HTTP_ADDREQ_FLAG_ADD_IF_NEW**. If the application tries to remove or replace a header using **HTTP_ADDREQ_FLAG_REMOVE** or **HTTP_ADDREQ_FLAG_REPLACE**, only one header can be supplied in *lpszHeaders*.

See Also

Reference

CHttpFile Class
Hierarchy Chart
CInternetFile Class
Concepts
CHttpFile Members

CHttpFile::CHttpFile

This member function is called to construct a **CHttpFile** object.

```
CHttpFile(
   HINTERNET hFile,
   HINTERNET hSession,
   LPCTSTR pstrObject,
   LPCTSTR pstrServer,
   LPCTSTR pstrVerb,
   DWORD_PTR dwContext
);
CHttpFile(
   HINTERNET hFile,
   LPCTSTR pstrVerb,
   LPCTSTR pstrObject,
   CHttpConnection* pConnection
);
```

Parameters

hFile

A handle to an Internet file.

hSession

A handle to an Internet session.

pstrObject

A pointer to a string containing the **CHttpFile** object.

pstrServer

A pointer to a string containing the name of the server.

pstrVerb

A pointer to a string containing the method to be used when sending the request. Can be **POST**, **HEAD**, or **GET**.

dwContext

The context identifier for the **CHttpFile** object. See **Remarks** for more information about this parameter.

pConnection

A pointer to a CHttpConnection object.

Remarks

You never construct a **CHttpFile** object directly; rather call CInternetSession::OpenURL or CHttpConnection::OpenRequest instead.

The default value for *dwContext* is sent by MFC to the **CHttpFile** object from the CInternetSession object that created the **CHttpFile** object. When you call **CInternetSession::OpenURL** or **CHttpConnection** to construct a **CHttpFile** object, you can override the default to set the context identifier to a value of your choosing. The context identifier is returned to CInternetSession::OnStatusCallback to provide status on the object with which it is identified. See the article Internet First Steps: WinInet for more information about the context identifier.

See Also

Reference

CHttpFile Class
Hierarchy Chart
CInternetFile Class

Concepts

CHttpFile Members

CHttpFile::EndRequest

This topic has been updated for Visual Studio 2005 SP1.

Call this member function to end a request sent to an HTTP server with the SendRequestEx member function.

```
BOOL EndRequest(
   DWORD dwFlags = 0,
   LPINTERNET_BUFFERS lpBuffIn = NULL,
   DWORD_PTR dwContext = 1
);
```

Parameters

dwFlags

Flags describing the operation. For a list of the appropriate flags, see HttpEndRequest in the *Platform SDK*.

lpBuffIn

Pointer to an initialized INTERNET_BUFFERS that describes the input buffer used for the operation.

dwContext

The context identifier for the **CHttpFile** operation. See Remarks for more information about this parameter.

Return Value

Nonzero if successful; otherwise 0. If the call fails, determine the cause of the failure by examining the thrown CInternetException object.

Remarks

The default value for *dwContext* is sent by MFC to the **CHttpFile** object from the CInternetSession object that created the **CHttpFile** object. When you call CInternetSession::OpenURL or CHttpConnection to construct a **CHttpFile** object, you can override the default to set the context identifier to a value of your choosing. The context identifier is returned to CInternetSession::OnStatusCallback to provide status on the object with which it is identified. See article Internet First Steps: WinInet for more information about the context identifier.

Exceptions

This method can throw exceptions of type **CInternetException***.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CHttpFile Class
Hierarchy Chart
CInternetFile Class

Concepts

CHttpFile Members

CHttpFile::GetFileURL

Call this member function to get the name of the HTTP file as a URL.

```
virtual CString GetFileURL( ) const;
```

Return Value

A CString object containing a URL referencing the resource associated with this file.

Remarks

Use this member function only after a successful call to SendRequest or on a **CHttpFile** object successfully created by OpenURL.

See Also

Reference

CHttpFile Class
Hierarchy Chart
CInternetFile Class

Concepts

CHttpFile Members

CHttpFile::GetObject

Call this member function to get the name of the object associated with this **CHttpFile**.

```
CString GetObject( ) const;
```

Return Value

A CString object containing the name of the object.

Remarks

Use this member function only after a successful call to SendRequest or on a **CHttpFile** object successfully created by OpenURL.

See Also

Reference

CHttpFile Class
Hierarchy Chart
CInternetFile Class

Concepts

CHttpFile Members

CHttpFile::GetVerb

Call this member function to get the HTTP verb (or method) associated with this **CHttpFile**.

```
CString GetVerb( ) const;
```

Return Value

A CString object containing the name of the HTTP verb (or method).

Remarks

Use this member function only after a successful call to SendRequest or on a **CHttpFile** object successfully created by OpenURL.

See Also

Reference

CHttpFile Class
Hierarchy Chart
CInternetFile Class

Concepts

CHttpFile Members

CHttpFile::QueryInfo

Call this member function to return response or request headers from an HTTP request.

```
BOOL QueryInfo(
   DWORD dwInfoLevel,
   LPVOID lpvBuffer,
   LPDWORD lpdwBufferLength,
   LPDWORD lpdwIndex = NULL
) const;
BOOL QueryInfo(
   DWORD dwInfoLevel,
   CString& str,
   LPDWORD dwIndex = NULL
) const;
BOOL QueryInfo(
   DWORD dwInfoLevel,
   SYSTEMTIME* pSysTime,
   LPDWORD dwIndex = NULL
) const;
```

Parameters

dwInfoLevel

A combination of the attribute to query and the following flags that specify the type of information requested:

- **HTTP_QUERY_CUSTOM** Finds the header name and returns this value in *lpvBuffer* on output. **HTTP_QUERY_CUSTOM** throws an assertion if the header isn't found.
- **HTTP_QUERY_FLAG_REQUEST_HEADERS** Typically, the application queries the response headers, but an application can also query request headers by using this flag.
- **HTTP_QUERY_FLAG_SYSTEMTIME** For those headers whose value is a date/time string, such as "Last-Modified-Time," this flag returns the header value as a standard Win32 SYSTEMTIME structure that does not require the application to parse the data. If you use this flag, you may want to use the **SYSTEMTIME** override of the function.
- **HTTP_QUERY_FLAG_NUMBER** For those headers whose value is a number, such as the status code, this flag returns the data as a 32-bit number.

See the **Remarks** section for a list of the possible values.

lpvBuffer

A pointer to the buffer that receives the information.

lpdwBufferLength

On entry, this points to a value containing the length of the data buffer, in number of characters or bytes. See the **Remarks** section for more detailed information about this parameter.

lpdwIndex

A pointer to a zero-based header index. Can be **NULL**. Use this flag to enumerate multiple headers with the same name. On input, *lpdwIndex* indicates the index of the specified header to return. On output, *lpdwIndex* indicates the index of the next header. If the next index cannot be found, **ERROR_HTTP_HEADER_NOT_FOUND** is returned.

str

A reference to the CString object receiving the returned information.

dwIndex

An index value. See *lpdwIndex*.

pSysTime

A pointer to a Win32 SYSTEMTIME structure.

Return Value

Nonzero if successful; otherwise 0. If the call fails, the Win32 function GetLastError may be called to determine the cause of the error.

Remarks

Use this member function only after a successful call to SendRequest or on a **CHttpFile** object successfully created by OpenURL.

You can retrieve the following types of data from **QueryInfo**:

- strings (default)
- **SYSTEMTIME** (for "Data:" "Expires:" etc, headers)
- **DWORD** (for **STATUS_CODE**, **CONTENT_LENGTH**, etc.)

When a string is written to the buffer, and the member function succeeds, *lpdwBufferLength* contains the length of the string in characters minus 1 for the terminating **NULL** character.

The possible *dwInfoLevel* values include:

- **HTTP_QUERY_MIME_VERSION**
- **HTTP_QUERY_CONTENT_TYPE**
- **HTTP_QUERY_CONTENT_TRANSFER_ENCODING**
- **HTTP_QUERY_CONTENT_ID**
- **HTTP_QUERY_CONTENT_DESCRIPTION**
- **HTTP_QUERY_CONTENT_LENGTH**
- **HTTP_QUERY_ALLOWED_METHODS**
- **HTTP_QUERY_PUBLIC_METHODS**
- **HTTP_QUERY_DATE**
- **HTTP_QUERY_EXPIRES**
- **HTTP_QUERY_LAST_MODIFIED**
- **HTTP_QUERY_MESSAGE_ID**
- **HTTP_QUERY_URI**
- **HTTP_QUERY_DERIVED_FROM**
- **HTTP_QUERY_LANGUAGE**
- **HTTP_QUERY_COST**
- **HTTP_QUERY_WWW_LINK**
- **HTTP_QUERY_PRAGMA**
- **HTTP_QUERY_VERSION**
- **HTTP_QUERY_STATUS_CODE**
- **HTTP_QUERY_STATUS_TEXT**
- **HTTP_QUERY_RAW_HEADERS**
- **HTTP_QUERY_RAW_HEADERS_CRLF**

See Also

Reference

CHttpFile Class
Hierarchy Chart
CInternetFile Class

CHttpConnection::OpenRequest
CFtpConnection Class
CGopherConnection Class
CInternetConnection Class
Concepts
CHttpFile Members

CHttpFile::QueryInfoStatusCode

Call this member function to get the status code associated with an HTTP request and place it in the supplied *dwStatusCode* parameter.

```
BOOL QueryInfoStatusCode(
   DWORD& dwStatusCode
) const;
```

Parameters

dwStatusCode

A reference to a status code. Status codes indicate the success or failure of the requested event. See **Remarks** for a selection of status code descriptions.

Return Value

Nonzero if successful; otherwise 0. If the call fails, the Win32 function GetLastError may be called to determine the cause of the error.

Remarks

Use this member function only after a successful call to SendRequest or on a **CHttpFile** object successfully created by OpenURL.

HTTP status codes fall into groups indicating the success or failure of the request. The following tables outline the status code groups and the most common HTTP status codes.

Group	Meaning
200-299	Success
300-399	Information
400-499	Request error
500-599	Server error

Common HTTP Status Codes:

Status code	Meaning
200	URL located, transmission follows
400	Unintelligible request
404	Requested URL not found
405	Server does not support requested method
500	Unknown server error
503	Server capacity reached

See Also

Reference

CHttpFile Class

Hierarchy Chart

CInternetFile Class

Concepts

CHttpFile::SendRequest

Call this member function to send a request to an HTTP server.

```
BOOL SendRequest(
   LPCTSTR pstrHeaders = NULL,
   DWORD dwHeadersLen = 0,
   LPVOID lpOptional = NULL,
   DWORD dwOptionalLen = 0
);
BOOL SendRequest(
   CString& strHeaders,
   LPVOID lpOptional = NULL,
   DWORD dwOptionalLen = 0
);
```

Parameters

pstrHeaders

A pointer to a string containing the name of the headers to send.

dwHeadersLen

The length of the headers identified by *pstrHeaders*.

lpOptional

Any optional data to send immediately after the request headers. This is generally used for **POST** and **PUT** operations. This can be **NULL** if there is no optional data to send.

dwOptionalLen

The length of *lpOptional*.

strHeaders

A string containing the name of the headers for the request being sent.

Return Value

Nonzero if successful; otherwise 0. If the call fails, determine the cause of the failure by examining the thrown CInternetException object.

Exceptions

This method can throw exceptions of type **CInternetException**.

See Also

Reference

CHttpFile Class
Hierarchy Chart
CInternetFile Class
CHttpFile::SendRequestEx

Concepts

CHttpFile Members

CHttpFile::SendRequestEx

This topic has been updated for Visual Studio 2005 SP1.

Call this member function to send a request to an HTTP server.

```
BOOL SendRequestEx(
   DWORD dwTotalLen,
   DWORD dwFlags = HSR_INITIATE,
   DWORD_PTR dwContext = 1
);
BOOL SendRequestEx(
   LPINTERNET_BUFFERS lpBuffIn,
   LPINTERNET_BUFFERS lpBuffOut,
   DWORD dwFlags = HSR_INITIATE,
   DWORD_PTR dwContext = 1
);
```

Parameters

dwTotalLen

Number of bytes to be sent in the request.

dwFlags

Flags describing the operation. For a list of appropriate flags, see HttpSendRequestEx in the *Platform SDK*.

dwContext

The context identifier for the **CHttpFile** operation. See Remarks for more information about this parameter.

lpBuffIn

Pointer to an initialized INTERNET_BUFFERS that describes the input buffer used for the operation.

lpBuffOut

Pointer to an initialized **INTERNET_BUFFERS** that describes the output buffer used for the operation.

Return Value

Nonzero if successful; otherwise 0. If the call fails, determine the cause of the failure by examining the thrown CInternetException object.

Remarks

This function allows your application to send data using the Write and WriteString methods of **CInternetFile**. You must know the length of the data to send before calling either override of this function. The first override allows you to specify the length of data you'd like to send. The second override accepts pointers to **INTERNET_BUFFERS** structures, which can be used to describe the buffer in great detail.

After content is written to the file, call EndRequest to end the operation.

The default value for *dwContext* is sent by MFC to the **CHttpFile** object from the CInternetSession object that created the **CHttpFile** object. When you call CInternetSession::OpenURL or CHttpConnection to construct a **CHttpFile** object, you can override the default to set the context identifier to a value of your choosing. The context identifier is returned to CInternetSession::OnStatusCallback to provide status on the object with which it is identified. See the article Internet First Steps: WinInet for more information about the context identifier.

Exceptions

This method can throw exceptions of type **CInternetException***.

Example

This code fragment sends the content of a string to a DLL named ISAPI.DLL on a server named MOOSEBOY. While this example uses only one call to **WriteString**, using multiple calls to send data in blocks is acceptable.

```
CString strData = "Some very long data to be POSTed here!";
pServer = sess.GetHttpConnection("mooseboy");
pFile = pServer->OpenRequest(CHttpConnection::HTTP_VERB_POST, "/isapi.dll?");
pFile->SendRequestEx(strData.GetLength());

pFile->WriteString(strData);
pFile->EndRequest();
```

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CHttpFile Class
Hierarchy Chart
CInternetFile Class
CHttpFile::SendRequest

Concepts

CHttpFile Members

CHttpFilter Class

Creates and manages, with CHttpFilterContext, a Hypertext Transfer Protocol (HTTP) filter object.

```
class CHttpFilter
```

Remarks

CHttpFilter does not have a base class.

An HTTP filter is a replaceable dynamic link library (DLL) that the server calls on every HTTP request. When the filter is loaded, it tells the server what sort of notifications it is interested in. After that, whenever the selected events occur, the filter is called and given the opportunity to process that event.

ISAPI (Internet Server API) filters are powerful enough to allow for the following applications:

- Custom authentication schemes
- Compression
- Encryption
- Logging
- Traffic analysis or other request analysis

Multiple filters can be installed. The notification order is based on the priority specified by the filter and then the load order in the registry for any ties. Consult your filter's documentation to see exactly how to install your filter.

Note
Once a filter begins processing a request, it will receive the data regardless of whether the request is for a file or an ISAPI app lication.

The filter applications sit between the network connection to the client and the HTTP server. Depending on the options that the filter application chooses, it can act on several server actions, including reading raw data from the client, processing the headers, communications over a secure port (PCT — Personal Communications Technology, SSL — Secure Sockets Layer, and others), or several other stages in the processing of the HTTP request.

To set the filter notifications that your filter will use, see GetFilterVersion.

For more information on Internet filters, see ISAPI Extensions: Filters. For information about creating an Internet filter with ISAPI Extension Wizard, see Steps to Create a Typical ISAPI Filter.

Requirements

Header: afxisapi.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart
CHttpFilterContext Class
CHttpServer Class
CHttpServerContext Class

Concepts

How Do I Use the MFC ISAPI Classes Without Linking to the MFC DLL?

Other Resources

CHttpFilter Members

CHttpFilter Members

Construction

CHttpFilter	Constructs a **CHttpFilter** object.

Attributes

GetFilterVersion	Gets the version of the filter after the **CHttpFilter** object is constructed.

Overridables

OnAccessDenied	Notifies the client that access to a resource has been denied.
OnAuthComplete	Notifies the client that authentication is complete.
OnAuthentication	Authenticates the client.
OnEndOfNetSession	Notifies the client that the session is ending.
OnEndOfRequest	Notifies the client that the request has ended.
OnLog	Logs information to a server file.
OnPreprocHeaders	Notifies the client that the server has preprocessed the client headers.
OnReadRawData	Allows the application to see the raw data. The data returned will contain both headers and data.
OnSendRawData	Sends raw data from the server to the client.
OnSendResponse	Notifies the client that the requested headers are being sent to a client.
OnUrlMap	Notifies a client when a server is mapping a logical URL to a physical path.

Operation

HttpFilterProc	Returns a message indicating how an event that passed through the filter was processed. Called each time an event occors.

See Also

Reference

CHttpFilterContext Class
CHttpFilter Class
Hierarchy Chart

CHttpFilter Member Functions

For information about the member functions in **CHttpFilter**, see CHttpFilter Members.

CHttpFilter::CHttpFilter

This member function is called by the framework during the construction of a **CHttpFilter** object.

```
CHttpFilter( );
```

See Also

Reference

CHttpFilter Class

Hierarchy Chart

Other Resources

CHttpFilter Members

CHttpFilter::GetFilterVersion

This member function is called by the internet server to get the filter version indicated by *pVer*.

```
virtual BOOL GetFilterVersion(
   PHTTP_FILTER_VERSION pVer
);
```

Parameters

pVer

A pointer to the HTTP_FILTER_VERSION structure containing the server's version information and fields for the filter to indicate version number and notifications. The filter application also includes space to register a small description of itself. The following two flags are set in the structure by the default implementation:

- *dwFlags* The priority notification flag, **SF_NOTIFY_ORDER_DEFAULT**, is set by default. See **Remarks** for a list of the notification flags and their descriptions.
- *dwFilterVersion* **HTTP_FILTER_REVISION** is set by default. This flag indicates the version of the specification used by the server.

Return Value

Nonzero if the filter was properly loaded. If the filter returns 0, then the filter application will be unloaded and it will not receive any notifications.

Remarks

It is called only once, after the **CHttpFilter** object is constructed.

Use *dwFlags* to specify the notifications in the *pVer* member that interest your server. Here is a list of the valid flags for *dwFlags*:

SF_NOTIFY_ORDER_DEFAULT

Loads the filter at the default priority. This value is recommended because other priority notifications can have a strong impact on performance and scalability.

SF_NOTIFY_ORDER_LOW

Loads the filter at low priority.

SF_NOTIFY_ORDER_MEDIUM

Loads the filter at medium priority.

SF_NOTIFY_ORDER_HIGH

Loads the filter at high priority.

SF_NOTIFY_SECURE_PORT

Notifies the application that it is passing data through a secure port.

SF_NOTIFY_NONSECURE_PORT

Notifies the application that it is passing data through a nonsecure port.

Note
If you set neither **SF_NOTIFY_NONSECURE_PORT** nor **SF_NOTIFY_SECURE_PORT**, the server defaults to both, which allows processing data through any port.

SF_NOTIFY_READ_RAW_DATA

Allows the application to see the raw data. The data returned to the client will contain both headers and data.

SF_NOTIFY_PREPROC_HEADERS

The server has pre-processed the headers.

SF_NOTIFY_AUTHENTICATION

The server is authenticating the client.

SF_NOTIFY_URL_MAP

The server is mapping a logical URL to a physical path.

SF_NOTIFY_SEND_RAW_DATA

The server is sending raw data back to the client.

SF_NOTIFY_LOG

The server is writing information to the server log.

SF_NOTIFY_END_OF_NET_SESSION

The session with the client is ending.

See Also

Reference

CHttpFilter Class
Hierarchy Chart
HTTP_FILTER_VERSION Structure
CHttpFilter::HttpFilterProc

Other Resources

CHttpFilter Members

CHttpFilter::HttpFilterProc

This member function is called by the framework to process data every time it passes through the filter.

```
virtual DWORD HttpFilterProc(
   PHTTP_FILTER_CONTEXT pfc,
   DWORD dwNotificationType,
   LPVOID pvNotification
);
```

Parameters

pfc

A pointer to an HTTP_FILTER_CONTEXT structure. The **HTTP_FILTER_CONTEXT** structure pointed to by this parameter contains context information. The *pfc* structure member can be used by the filter to associate any context information with the HTTP request. The **SF_NOTIFY_END_OF_NET_SESSION** notification can be used to release any such context information.

NotificationType

Indicates the type of event being processed. Valid types are listed in GetFilterVersion.

pvNotification

A notification-specific structure.

Notification Type	***pvNotification* points to**	MFC Calls
SF_NOTIFY_READ_RAW_DATA	HTTP_FILTER_RAW_DATA	OnReadRawData
SF_NOTIFY_SEND_RAW_DATA	HTTP_FILTER_RAW_DATA	OnSendRawData
SF_NOTIFY_PREPROC_HEADERS	HTTP_FILTER_PREPROC_HEADERS	OnPreprocHeaders
SF_NOTIFY_AUTHENTICATION	HTTP_FILTER_AUTHENT	OnAuthentication
SF_NOTIFY_URL_MAP	HTTP_FILTER_URL_MAP	OnUrlMap
SF_NOTIFY_LOG	HTTP_FILTER_LOG	OnLog

Return Value

Indicates how the application handled the event. Indicated by a *dwFlags* value; see GetFilterVersion **Remarks** for a list of these values.

Remarks

HttpFilterProc will call the appropriate **CHttpFilter** member functions, depending on the notification types given. For example, **HttpFilterProc** will call OnPreprocHeaders if the notification type is **SF_NOTIFY_PREPROC_HEADERS**.

HttpFilterProc is where the core work of the ISAPI filter applications is done. The various structures pointed to by *pvNotification* (listed in the table above) contain data and function pointers specific to these operations. See the structure details for more information about how data is processed by **HttpFilterProc**.

You can override the individual handlers (listed in the third column, above) to change the way data in their associated structures is processed.

See Also

Reference

CHttpFilter Class
Hierarchy Chart
HTTP_FILTER_CONTEXT Structure
HTTP_FILTER_AUTHENT Structure
HTTP_FILTER_PREPROC_HEADERS Structure

HTTP_FILTER_RAW_DATA Structure
HTTP_FILTER_URL_MAP Structure
HTTP_FILTER_LOG Structure

Other Resources

CHttpFilter Members

CHttpFilter::OnAccessDenied

Called by the framework prior to denying access from a client to a resource.

```
virtual DWORD OnAccessDenied(
   CHttpFilterContext* pfc,
   PHTTP_FILTER_ACCESS_DENIED pAccess
);
```

Parameters

pfc

A CHttpFilterContext object that contains context information. The **CHttpFilterContext** object can be used by the filter to associate any context information with the HTTP request.

pAccess

A pointer to an HTTP_FILTER_ACCESS_DENIED structure.

Return Value

For a list of possible return values, see the **Return Value** section of the HttpFilterProc reference topic, located in the *Platform SDK*.

Remarks

Override this member function to implement your own response when access is denied to a client.

See Also

Reference

CHttpFilter Class
Hierarchy Chart
CHttpFilter::HttpFilterProc
CHttpFilterContext Class

Other Resources

CHttpFilter Members

CHttpFilter::OnAuthentication

This member function is called by the framework to authenticate the client.

```
virtual DWORD OnAuthentication(
   CHttpFilterContext* pfc,
   PHTTP_FILTER_AUTHENT pAuthent
);
```

Parameters

pfc

A CHttpFilterContext object, which contains context information. The **CHttpFilterContext** object can be used by the filter to associate any context information with the HTTP request.

pAuthent

A pointer to an HTTP_FILTER_AUTHENT structure.

Return Value

One of the following notification types:

SF_STATUS_REQ_FINISHED

The filter has handled the HTTP request. The server should disconnect the session.

SF_STATUS_REQ_FINISHED_KEEP_CONN

Same as **SF_STATUS_REQ_FINISHED** except the server should keep the TCP session open if the option was negotiated.

SF_STATUS_REQ_NEXT_NOTIFICATION

The next filter in the notification chain should be called.

SF_STATUS_REQ_HANDLED_NOTIFICATION

This filter handled the notification. No other handlers should be called for this particular notification.

SF_STATUS_REQ_ERROR

An error occurred. The server should use the Win32 API SetLastError to indicate the error to the client.

SF_STATUS_REQ_READ_NEXT

The filter is an opaque stream filter; Negotiate the session parameters. Only valid for raw read notification.

If unsuccessful, the notification type **SF_STATUS_REQ_ERROR** should be returned. In this case, the server should use the Windows function SetLastError and indicate the error to the client.

Remarks

Override this member function to implement your own authentication. The default implementation does nothing.

See Also

Reference

CHttpFilter Class
Hierarchy Chart
CHttpFilter::HttpFilterProc
HTTP_FILTER_AUTHENT Structure
CHttpFilterContext Class

Other Resources

CHttpFilter Members

CHttpFilter::OnAuthComplete

Called by the framework when authentication is complete.

```
virtual DWORD OnAuthComplete(
   CHttpFilterContext* pfc,
   PHTTP_FILTER_AUTH_COMPLETE_INFO pAuthComplInfo
);
```

Parameters

pfc

A CHttpFilterContext object, which contains context information and can be used by the filter to associate any context information with the HTTP request.

pAuthComplInfo

A pointer to an HTTP_FILTER_AUTH_COMPLETE_INFO structure.

Return Value

One of the following notification types:

SF_STATUS_REQ_FINISHED

The filter has handled the HTTP request. The server should disconnect the session.

SF_STATUS_REQ_FINISHED_KEEP_CONN

Same as **SF_STATUS_REQ_FINISHED**, except the server should keep the TCP session open if the option was negotiated.

SF_STATUS_REQ_NEXT_NOTIFICATION

The next filter in the notification chain should be called.

SF_STATUS_REQ_HANDLED_NOTIFICATION

This filter handled the notification. No other handlers should be called for this particular notification.

SF_STATUS_REQ_ERROR

An error occurred. The server should use the Win32 API SetLastError to indicate the error to the client.

SF_STATUS_REQ_READ_NEXT

The filter is an opaque stream filter; negotiate the session parameters. Only valid for raw read notification.

If unsuccessful, the notification type **SF_STATUS_REQ_ERROR** should be returned. In this case, the server should use the Windows function SetLastError and indicate the error to the client.

Remarks

Override this member function to provide your own authentication-complete implementation. The default implementation does nothing.

See Also

Reference

CHttpFilter Class
Hierarchy Chart
CHttpFilter::HttpFilterProc
HTTP_FILTER_AUTH_COMPLETE_INFO Structure
CHttpFilterContext Class

Other Resources

CHttpFilter Members

CHttpFilter::OnEndOfNetSession

This member function is called by the framework to notify the filter that the session is ending.

```
virtual DWORD OnEndOfNetSession(
   CHttpFilterContext* pfc
);
```

Parameters

pfc

A CHttpFilterContext object, which contains context information and can be used by the filter to associate any context information with the HTTP request.

Return Value

One of the following notification types:

SF_STATUS_REQ_FINISHED

The filter has handled the HTTP request. The server should disconnect the session.

SF_STATUS_REQ_FINISHED_KEEP_CONN

Same as **SF_STATUS_REQ_FINISHED** except the server should keep the TCP session open if the option was negotiated.

SF_STATUS_REQ_NEXT_NOTIFICATION

The next filter in the notification chain should be called.

SF_STATUS_REQ_HANDLED_NOTIFICATION

This filter handled the notification. No other handlers should be called for this particular notification.

SF_STATUS_REQ_ERROR

An error occurred. The server should use the Win32 API SetLastError to indicate the error to the client.

SF_STATUS_REQ_READ_NEXT

The filter is an opaque stream filter; Negotiate the session parameters. Only valid for raw read notification.

If unsuccessful, the notification type **SF_STATUS_REQ_ERROR** should be returned. In this case, the server should use the Windows function SetLastError and indicate the error to the client.

Remarks

Override this member function to provide your own end of session implementation. The default implementation does nothing.

See Also

Reference

CHttpFilter Class
Hierarchy Chart
CHttpFilter::HttpFilterProc
CHttpFilterContext Class

Other Resources

CHttpFilter Members

CHttpFilter::OnEndOfRequest

Called by the framework to inform the filter when the request ends.

```
virtual DWORD OnEndOfRequest(
   CHttpFilterContext* pfc
);
```

Parameters

pfc

A CHttpFilterContext object, which contains context information and can be used by the filter to associate any context information with the HTTP request.

Return Value

One of the following notification types:

SF_STATUS_REQ_FINISHED

The filter has handled the HTTP request. The server should disconnect the session.

SF_STATUS_REQ_FINISHED_KEEP_CONN

Same as **SF_STATUS_REQ_FINISHED**, except the server should keep the TCP session open if the option was negotiated.

SF_STATUS_REQ_NEXT_NOTIFICATION

The next filter in the notification chain should be called.

SF_STATUS_REQ_HANDLED_NOTIFICATION

This filter handled the notification. No other handlers should be called for this particular notification.

SF_STATUS_REQ_ERROR

An error occurred. The server should use the Win32 API SetLastError to indicate the error to the client.

SF_STATUS_REQ_READ_NEXT

The filter is an opaque stream filter; negotiate the session parameters. Only valid for raw read notification.

If unsuccessful, the notification type **SF_STATUS_REQ_ERROR** should be returned. In this case, the server should use the Windows function SetLastError and indicate the error to the client.

Remarks

Override this member function to provide your own end of request implementation. The default implementation does nothing.

See Also

Reference

CHttpFilter Class
Hierarchy Chart
CHttpFilter::HttpFilterProc
CHttpFilterContext Class

Other Resources

CHttpFilter Members

CHttpFilter::OnLog

This member function is called by the framework to inform the filter when the server is writing information to the server log.

```
virtual DWORD OnLog(
   CHttpFilterContext* pfc,
   PHTTP_FILTER_LOG pLog
);
```

Parameters

pfc

A CHttpFilterContext object, which contains context information, and can be used by the filter to associate any context information with the HTTP request.

pLog

A pointer to an HTTP_FILTER_LOG structure.

Return Value

One of the following notification types:

SF_STATUS_REQ_FINISHED

The filter has handled the HTTP request. The server should disconnect the session.

SF_STATUS_REQ_FINISHED_KEEP_CONN

Same as **SF_STATUS_REQ_FINISHED** except the server should keep the TCP session open if the option was negotiated.

SF_STATUS_REQ_NEXT_NOTIFICATION

The next filter in the notification chain should be called.

SF_STATUS_REQ_HANDLED_NOTIFICATION

This filter handled the notification. No other handlers should be called for this particular notification.

SF_STATUS_REQ_ERROR

An error occurred. The server should use the Win32 API SetLastError to indicate the error to the client.

SF_STATUS_REQ_READ_NEXT

The filter is an opaque stream filter; Negotiate the session parameters. Only valid for raw read notification.

If unsuccessful, the notification type **SF_STATUS_REQ_ERROR** should be returned. In this case, the server should use the Windows function SetLastError and indicate the error to the client.

Remarks

Override this member function to provide your own method for logging information to the server file. The default implementation does nothing.

Example

This example demonstrates how to modify values logged by the IIS for the specific request. For each request to the GIF files, this filter will replace corresponding values of the request log entry with empty values. Depending on the current log format, empty values may be shown as either "-" (hyphen) or as commas („).

```
DWORD CLogNotifyFilter::OnLog(CHttpFilterContext *pCtxt, PHTTP_FILTER_LOG pLog)
{
   // TODO: React to this notification accordingly and
   // return the appropriate status code

   ISAPITRACE1 ("Log notification: %s\n", pLog->pszTarget);
   // check if this request is for the file name and includes a gif
   // or GIF extension
```

```
    if ( strstr (pLog->pszTarget, ".gif") || strstr (pLog->pszTarget, ".GIF") )
    {
       ISAPITRACE ("Request for the GIF (or gif)\n");
       // This is one way of copying empty strings to the corresponding
       // log entry. Depending on the log format, empty values will be
       // represented as "-" or ,,
       strcpy ( (char*)pLog->pszClientHostName, "");
       // another way is to just set pointers to NULL
       *(char *)pLog->pszClientUserName = NULL;
       *(char *)pLog->pszServerName = NULL;
       *(char *)pLog->pszOperation = NULL;
       *(char *)pLog->pszTarget = NULL;
       *(char *)pLog->pszParameters = NULL;
    }
    return SF_STATUS_REQ_NEXT_NOTIFICATION;
 }
```

See Also

Reference

CHttpFilter Class

Hierarchy Chart

CHttpFilter::HttpFilterProc

HTTP_FILTER_LOG Structure

CHttpFilterContext Class

Other Resources

CHttpFilter Members

CHttpFilter::OnPreprocHeaders

This member function is called by the framework to notify the client that the server has preprocessed the client headers.

```
virtual DWORD OnPreprocHeaders(
   CHttpFilterContext* pfc,
   PHTTP_FILTER_PREPROC_HEADERS pHeaders
);
```

Parameters

pfc

A CHttpFilterContext object, which contains context information. The **CHttpFilterContext** object can be used by the filter to associate any context information with the HTTP request.

pHeaders

A pointer to a HTTP_FILTER_PREPROC_HEADERS structure.

Return Value

One of the following notification types:

SF_STATUS_REQ_FINISHED

The filter has handled the HTTP request. The server should disconnect the session.

SF_STATUS_REQ_FINISHED_KEEP_CONN

Same as **SF_STATUS_REQ_FINISHED** except the server should keep the TCP session open if the option was negotiated.

SF_STATUS_REQ_NEXT_NOTIFICATION

The next filter in the notification chain should be called.

SF_STATUS_REQ_HANDLED_NOTIFICATION

This filter handled the notification. No other handlers should be called for this particular notification.

SF_STATUS_REQ_ERROR

An error occurred. The server should use the Win32 API SetLastError to indicate the error to the client.

SF_STATUS_REQ_READ_NEXT

The filter is an opaque stream filter; Negotiate the session parameters. Only valid for raw read notification.

If unsuccessful, the notification type **SF_STATUS_REQ_ERROR** should be returned. In this case, the server should use the Windows function SetLastError and indicate the error to the client.

Remarks

Override this member function to provide your own method for processing client headers. The default does nothing.

See Also

Reference

CHttpFilter Class
Hierarchy Chart
CHttpFilter::HttpFilterProc
HTTP_FILTER_PREPROC_HEADERS Structure
CHttpFilterContext Class

Other Resources

CHttpFilter Members

CHttpFilter::OnReadRawData

This member function is called by the framework to allow the application to see the raw data.

```
virtual DWORD OnReadRawData(
   CHttpFilterContext* pfc,
   PHTTP_FILTER_RAW_DATA pRawData
);
```

Parameters

pfc

A CHttpFilterContext object, which contains context information. The **CHttpFilterContext** object can be used by the filter to associate any context information with the HTTP request.

pRawData

A pointer to an HTTP_FILTER_RAW_DATA structure.

Return Value

One of the following notification types:

SF_STATUS_REQ_FINISHED

The filter has handled the HTTP request. The server should disconnect the session.

SF_STATUS_REQ_FINISHED_KEEP_CONN

Same as **SF_STATUS_REQ_FINISHED** except the server should keep the TCP session open if the option was negotiated.

SF_STATUS_REQ_NEXT_NOTIFICATION

The next filter in the notification chain should be called.

SF_STATUS_REQ_HANDLED_NOTIFICATION

This filter handled the notification. No other handlers should be called for this particular notification.

SF_STATUS_REQ_ERROR

An error occurred. The server should use the Win32 API SetLastError to indicate the error to the client.

SF_STATUS_REQ_READ_NEXT

The filter is an opaque stream filter; Negotiate the session parameters. Only valid for raw read notification.

If unsuccessful, the notification type **SF_STATUS_REQ_ERROR** should be returned. In this case, the server should use the Windows function SetLastError and indicate the error to the client.

Remarks

The data returned will contain both headers and data.

Override this member function to process raw data differently.The default implementation does nothing.

See Also

Reference

CHttpFilter Class
Hierarchy Chart
CHttpFilter::HttpFilterProc
HTTP_FILTER_RAW_DATA Structure
CHttpFilterContext Class
CHttpFilter::OnSendRawData

Other Resources

CHttpFilter Members

CHttpFilter::OnSendRawData

This member function is called by the framework to notify the client that the server is sending raw data back to the client.

```
virtual DWORD OnSendRawData(
   CHttpFilterContext* pfc,
   PHTTP_FILTER_RAW_DATA pRawData
);
```

Parameters

pfc

A CHttpFilterContext object, which contains context information. The **CHttpFilterContext** object can be used by the filter to associate any context information with the HTTP request. The **SF_NOTIFY_END_OF_NET_SESSION** notification can be used to release any such context information.

pRawData

A pointer to an HTTP_FILTER_RAW_DATA structure.

Return Value

If successful, the notification type **SF_STATUS_REQ_NEXT_NOTIFICATION**. Call the next filter in the notification chain.

If unsuccessful, the notification type **SF_STATUS_REQ_ERROR** should be returned. In this case, the server should use the Windows function SetLastError and indicate the error to the client.

Remarks

Override this member function only to change the default notification handler used by HttpFilterProc and process raw data differently.

See Also

Reference

CHttpFilter Class
Hierarchy Chart
CHttpFilter::HttpFilterProc
HTTP_FILTER_RAW_DATA Structure
CHttpFilterContext Class
CHttpFilter::OnReadRawData

Other Resources

CHttpFilter Members

CHttpFilter::OnSendResponse

Called by the framework prior to sending requested headers to a client.

```
virtual DWORD OnSendResponse(
   CHttpFilterContext* pfc,
   PHTTP_FILTER_SEND_RESPONSE pResponse
);
```

Parameters

pfc

A CHttpFilterContext object that contains context information. The **CHttpFilterContext** object can be used by the filter to associate any context information with the HTTP request.

pResponse

A pointer to an **HTTP_FILTER_SEND_RESPONSE** structure.

Return Value

For a list of possible return values, see the **Return Value** section of the HttpFilterProc reference topic, located in the *Platform SDK*.

Remarks

Override this member function to implement your own response when access is denied to a client.

See Also

Reference

CHttpFilter Class
Hierarchy Chart
CHttpFilter::HttpFilterProc
CHttpFilterContext Class

Other Resources

CHttpFilter Members

CHttpFilter::OnUrlMap

This member function is called by the framework when the server is mapping a logical URL to a physical path.

```
virtual DWORD OnUrlMap(
   CHttpFilterContext* pfc,
   PHTTP_FILTER_URL_MAP pUrlMap
);
```

Parameters

pfc

A CHttpFilterContext object, which contains context information. The **CHttpFilterContext** object can be used by the filter to associate any context information with the HTTP request.

pUrlMap

A pointer to an HTTP_FILTER_URL_MAP structure.

Return Value

One of the following notification types:

SF_STATUS_REQ_FINISHED

The filter has handled the HTTP request. The server should disconnect the session.

SF_STATUS_REQ_FINISHED_KEEP_CONN

Same as **SF_STATUS_REQ_FINISHED** except the server should keep the TCP session open if the option was negotiated.

SF_STATUS_REQ_NEXT_NOTIFICATION

The next filter in the notification chain should be called.

SF_STATUS_REQ_HANDLED_NOTIFICATION

This filter handled the notification. No other handlers should be called for this particular notification.

SF_STATUS_REQ_ERROR

An error occurred. The server should use the Win32 API SetLastError to indicate the error to the client.

SF_STATUS_REQ_READ_NEXT

The filter is an opaque stream filter; Negotiate the session parameters. Only valid for raw read notification.

If unsuccessful, the notification type **SF_STATUS_REQ_ERROR** should be returned. In this case, the server should use the Windows function SetLastError and indicate the error to the client.

Remarks

Override this member function handle URL mapping differently. The default implementation does nothing.

Example

The following code demonstrates how to use an **OnUrlMap** notification in the filter to redirect a request to a different URL. If the URL contains a **DoRedirect** string (for example, http://server/doRedirect), the request will be redirected to http://www.microsoft.com.

```
DWORD CFiltRedirFilter::OnUrlMap(CHttpFilterContext* pCtxt,
   PHTTP_FILTER_URL_MAP pMapInfo)
{
   CHAR szRedirect [256];

   if (strstr (pMapInfo->pszURL, "DoRedirect"))
   {
      CHAR szRedirect [256];
      // replace www.microsoft.com with desired server
```

```
        sprintf(szRedirect, "Location: http://%s\r\n\r\n", "www.microsoft.com");
        pCtxt->ServerSupportFunction ( SF_REQ_SEND_RESPONSE_HEADER,
              (LPVOID) "302 Redirect",
              (DWORD *) szRedirect,
              0 );
        // Print a message to the debug window
        ISAPITRACE1 ("Redirecting to: %s\n", szRedirect);
        // we are done with this request
        return SF_STATUS_REQ_FINISHED_KEEP_CONN;
    }
    // URL did not contain a DoRedirect string.
    return SF_STATUS_REQ_NEXT_NOTIFICATION;
}
```

See Also

Reference

CHttpFilter Class

Hierarchy Chart

CHttpFilter::HttpFilterProc

HTTP_FILTER_URL_MAP Structure

CHttpFilterContext Class

Other Resources

CHttpFilter Members

CHttpFilterContext Class

Provides the tools that a CHttpFilter object needs to process data that passes through the filter.

```
class CHttpFilterContext
```

Remarks

CHttpFilterContext does not have a base class.

When the filter receives a request, a **CHttpFilter** object is created and initialized, and a **CHttpFilterContext** object is created. As the filter processes requests, it uses **CHttpFilterContext** member functions to perform tasks.

A **CHttpFilterContext** object exists separately from a **CHttpFilter** object in order to allow multi-threading. Only one **CHttpFilter** object exists in a module, but a filter might be required to process multiple client requests simultaneously.

CHttpFilter will create a **CHttpFilterContext** for each request to handle these multiple requests. A **CHttpFilter** uses multiple **CHttpFilterContext** objects to run in separate threads. This design allows simultaneous, multiple calls to the **CHttpFilter** object by different client connections.

When an extension DLL (ISA) is called, the member function ServerSupportFunction prompts the server to provide the general ISA information to the client.

If the filter must communicate something — for example, an error — back to the client immediately, call WriteClient.

Requirements

Header: afxisapi.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart
CHttpServer Class
CHttpFilter Class
HTTP_FILTER_CONTEXT Structure

Concepts

How Do I Use the MFC ISAPI Classes Without Linking to the MFC DLL?

Other Resources

CHttpFilterContext Members

CHttpFilterContext Members

Data Members

m_pFC	A pointer to an HTTP_FILTER_CONTEXT structure.

Construction

CHttpFilterContext	Constructs a **CHttpFilterContext** object.

Attributes

AddResponseHeaders	Adds a header to the HTTP response.
AllocMem	Allocates memory in a buffer.
GetServerVariable	Copies information relating to an HTTP connection, or to the server itself, into a buffer supplied by the caller.
ServerSupportFunction	Provides general ISA information to the client.
WriteClient	Writes raw data to the client immediately.

See Also

Reference

CHttpFilterContext Class
Hierarchy Chart
HTTP_FILTER_CONTEXT Structure

CHttpFilterContext Member Functions

For information about the member functions in **CHttpFilterContext**, see CHttpFilterContext Members.

CHttpFilterContext::AddResponseHeaders

Call this member function to add a header to an HTTP response.

```
BOOL AddResponseHeaders(
   LPTSTR lpszHeaders,
   DWORD dwReserved =0
);
```

Parameters

lpszHeaders

A pointer to a string containing headers to add.

dwReserved

Reserved for future use. Must be 0.

Return Value

Nonzero if successful, otherwise 0.

Remarks

The header string is contained in *lpszHeaders*. See the **HSE_REQ_SEND_RESPONSE_HEADER** value described in the CHttpServerContext::ServerSupportFunction topic for information about how a **CHttpServer** object delivers information about an HTTP server response header.

See Also

Reference

CHttpFilterContext Class

Hierarchy Chart

Other Resources

CHttpFilterContext Members

CHttpFilterContext::AllocMem

Call this member function to allocate memory that is automatically freed when the communication with the client is terminated.

```
LPVOID AllocMem(
   DWORD cbSize,
   DWORD dwReserved = 0
);
```

Parameters

cbSize

Specifies the size of the memory buffer to allocate, in bytes.

dwReserved

Reserved for future use.

Return Value

A pointer to a buffer, or **NULL** on error.

Remarks

When an HTTP filter is registered, usually it will register for the end-of-net-session event. This event is a good time to recycle any buffers used by that client request. For performance reasons, most filters will probably keep a pool of filter buffers and only allocate or free a buffer when the pool becomes empty or too large to save on the overhead of the memory management. Calling **AllocMem** can have a negative impact on performance, but with careful use, it can be a valuable tool.

Memory blocks allocated with **AllocMem** cannot be managed with the normal C run-time or Windows API memory management functions.

See Also

Reference

CHttpFilterContext Class

Hierarchy Chart

Other Resources

CHttpFilterContext Members

CHttpFilterContext::CHttpFilterContext

This member function is called by the framework during the construction of a **CHttpFilterContext** object.

```
CHttpFilterContext(
   PHTTP_FILTER_CONTEXT pfc
);
```

Parameters

pfc

A pointer to a HTTP_FILTER_CONTEXT structure.

See Also

Reference

CHttpFilterContext Class
Hierarchy Chart
HTTP_FILTER_CONTEXT Structure

Other Resources

CHttpFilterContext Members

CHttpFilterContext::GetServerVariable

This member function is called by the framework to copy information relating to an HTTP connection, or to the server itself, into a buffer supplied by the caller.

```
BOOL GetServerVariable(
   LPTSTR lpszVariableName,
   LPVOID lpvBuffer,
   LPDWORD lpdwSize
);
```

Parameters

lpszVariableName

Null-terminated string indicating which variable is being requested. See the Remarks section below for a selection of possible names. All variable names are as defined in the CGI specification located at **http://hoohoo.ncsa.uiuc.edu/cgi/env.html**.

lpvBuffer

Pointer to buffer to receive the requested information.

lpdwSize

Pointer to DWORD indicating the number of bytes available in the buffer. On successful completion the DWORD contains the number of bytes transferred into the buffer (including the null-terminating byte).

Return Value

Nonzero if successful, otherwise 0. The Win32 API call GetLastError can be used to determine why the call failed. Possible error values include:

Value	Meaning
ERROR_INVALID_PARAMETER	Bad connection handle.
ERROR_INVALID_INDEX	Bad or unsupported variable identifier.
ERROR_INSUFFICIENT_BUFFER	Buffer too small; the required size is returned in *lpdwSize*.
ERROR_MORE_DATA	Buffer too small, only part of data returned. The total size of the data is not returned.
ERROR_NO_DATA	The data requested is not available.

Remarks

Possible *lpszVariableNames* include:

Value	Meaning
ALL_HTTP	All HTTP headers that were not already parsed into one of the above variables. These variables are of the form HTTP_<header field name>.
AUTH_TYPE	Contains the type of authentication used. For example, if Basic authentication is used, the string will be "Basic". For Windows NT Challenge-response, it will be "NTLM". Other authentication schemes will have other strings. Because new authentication types can be added to Internet Server, it is not possible to list all possible strings. If the string is empty then no authentication is used.

CONTENT_LENGTH	The number of bytes which the script can expect to receive from the client.
CONTENT_TYPE	The content type of the information supplied in the body of a POST request.
GATEWAY_INTERFACE	The revision of the CGI specification to which this server complies. The current version is CGI/1.1.

HTTP_ACCEPT	Special case HTTP header. Values of the Accept: fields are concatenated, separated by ", ". For example, if the following lines are part of the HTTP header:
	`accept: */*; q=0.1`
	`accept: text/html`
	`accept: image/jpeg`
	then the **HTTP_ACCEPT** variable will have a value of:
	`*/*; q=0.1, text/html, image/jpeg`

PATH_INFO	Additional path information, as given by the client. This comprises the trailing part of the URL after the script name but before the query string (if any).
PATH_TRANSLATED	This is the value of **PATH_INFO**, but with any virtual path name expanded into a directory specification.
QUERY_STRING	The information which follows the **?** in the URL which referenced this script.
REMOTE_ADDR	The IP address of the client.
REMOTE_HOST	The hostname of the client.
REMOTE_USER	This contains the username supplied by the client and authenticated by the server.
REQUEST_METHOD	The HTTP request method.

SCRIPT_NAME	The name of the script program being executed.
SERVER_NAME	The server's hostname (or IP address) as it should appear in self-referencing URLs.
SERVER_PORT	The TCP/IP port on which the request was received.
SERVER_PROTOCOL	The name and version of the information retrieval protocol relating to this request. Normally HTTP/1.0.
SERVER_SOFTWARE	The name and version of the web server under which the CGI program is running.

See Also

Reference

CHttpFilterContext Class

Hierarchy Chart

Other Resources

CHttpFilterContext Members

CHttpFilterContext::ServerSupportFunction

Call this member function to extend the ISA APIs.

```
BOOL ServerSupportFunction(
   enum SF_REQ_TYPE sfReq,
   LPVOID lpvBuffer,
   LPDWORD lpdwSize,
   LPDWORD lpdwDataType
);
```

Parameters

sfReq

Server request type. See the Remarks section for a list of of the possible values.

pvData

A pointer to a zero-terminated string. Its value is specific to the *sfReq* extension. When used with **SF_REQ_SEND_RESPONSE_HEADER**, it is an optional, null-terminated status string (for example, "401 Access Denied") or **NULL** for the default response of "200 OK". When used with **SF_REQ_ADD_HEADERS_ON_DENIAL**, it is a null-terminated string pointing to one or more header lines with terminating "\r\n".

lpdwSize

Null-terminated string. Its value is specific to the extension. When used with **SF_REQ_SEND_RESPONSE_HEADER**, it is a null-terminated string pointing to optional data to be appended and set with the header. If **NULL**, the header will be terminated with an empty line. When used with **SF_REQ_ADD_HEADERS_ON_DENIAL**, it is the size in bytes for the next read.

lpdwDataType

A null-terminated string pointing to optional headers or data to be appended and sent with the header. If **NULL**, the header will be terminated by a "\r\n" pair.

Return Value

Nonzero if successful, otherwise 0.

Remarks

The HTTP Server Extension value represented by *sfReq,* can be one of the following:

SF_REQ_SEND_RESPONSE_HEADER

Sends a complete HTTP server response header including the status, server version, message time, and MIME (Multipurpose Internet Mail Extension) version. Server extensions should append other information at the end, such as Content-type, Content-length, and so forth, followed by an extra "\r\n".

SF_REQ_ADD_HEADERS_ON_DENIAL

If the server denies the HTTP request, add the specified headers to the server error response. This allows an authentication filter to advertise its services without filtering every request. Generally the headers will be WWW-Authenticate headers with custom authentication schemes, but no restriction is placed on what headers may be specified.

SF_REQ_SET_NEXT_READ_SIZE

Only used by raw data filters that return **SF_STATUS_READ_NEXT**.

See Also

Reference

CHttpFilterContext Class

Hierarchy Chart

Other Resources

CHttpFilterContext Members

CHttpFilterContext::WriteClient

Call this member function to send raw data back to the client immediately.

```
BOOL WriteClient(
   LPVOID lpvBuffer,
   LPDWORD lpdwBytes,
   DWORD dwReserved = 0
);
```

Parameters

lpvBuffer

A pointer to the buffer containing the data.

lpdwBytes

A pointer to a DWORD containing the number of bytes to write from the buffer.

dwReserved

Reserved for future use. Must be 0.

Return Value

Nonzero if successful, otherwise 0. If 0, use the Windows function GetLastError to determine the cause of the error.

See Also

Reference

CHttpFilterContext Class
Hierarchy Chart
HTTP_FILTER_CONTEXT Structure
CHttpServerContext::WriteClient

Other Resources

CHttpFilterContext Members

CHttpFilterContext Data Members

For information about the data members in **CHttpFilterContext**, see CHttpFilterContext Members.

CHttpFilterContext::m_pFC

The **m_pFC** data member is a pointer to an HTTP_FILTER_CONTEXT structure.

Remarks

The pointer points at the same structure passed to CHttpFilter::HttpFilterProc.

See Also

Reference

CHttpFilterContext Class
Hierarchy Chart
HTTP_FILTER_CONTEXT Structure

Other Resources

CHttpFilterContext Members

CHttpServer Class

Provides a means to extend the functionality of an ISAPI-compliant HTTP server.

```
class CHttpServer
```

Remarks

CHttpServer does not have a base class.

The class **CHttpServer**, with CHttpServerContext, wraps the Internet Server API (ISAPI) functionality and can process various types of client requests, including extension DLLs. **CHttpServer** cannot process client requests from Common Gateway Interface (CGI) executables. These extension DLLs are sometimes called Internet Server Applications; however, they are DLLs, rather than EXEs. For brevity's sake, we refer to an extension DLL as an ISA.

For more information on the difference between CGI and ISA, see Internet Server API (ISAPI) Extensions.

When an ISAPI HTTP server receives a request from a client browser, a **CHttpServer** object is created and initialized, and a **CHttpServerContext** object is created. Only one instance of **CHttpServer** may exist for each module; however, one **CHttpServerContext** object is created for each call to the server. A **CHttpServer** object uses multiple **CHttpServerContext** objects to run in separate threads. This design allows simultaneous, multiple calls to the **CHttpServer** object by different client connections. The **CHttpServer** object communicates with the client or server itself via the **CHttpServerContext** object.

When the server loads the ISA, it calls the ISA at the entry point GetExtensionVersion to get the version number of the specification on which the extension is based. For every client request, the HttpExtensionProc member function is called. The default (recommended) implementation of **HttpExtensionProc** will read client data and decide what action is to be taken. You can override this member function to customize the implementation.

Other **CHttpServer** member functions process the client request, format the responses, and correspond with the client.

When a client command is received by a **CHttpServer** object, the parse maps associate the command to its class member function and parameters. Only one parse map is created per **CHttpServer** object.

See Internet Server API (ISAPI) Parse Maps for general information on using the parse map macros. See BEGIN_PARSE_MAP and END_PARSE_MAP for information on how to create a parse map to handle client commands.

See the following macro descriptions for information about how the client commands are mapped to member functions and their arguments:

- ON_PARSE_COMMAND
- ON_PARSE_COMMAND_PARAMS
- DEFAULT_PARSE_COMMAND

For more information on using parse maps to handle client commands, see ISAPI Extensions: Parse Maps.

For information on debugging internet extension DLLs, see Technical Note 63.

Requirements

Header: afxisapi.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart

CHtmlStream Class

Concepts

How Do I Use the MFC ISAPI Classes Without Linking to the MFC DLL?

Other Resources

CHttpServer Members

CHttpServer Members

Constructor

CHttpServer	Constructs a **CHttpServer** object.

Overridables

CallFunction	Finds and executes the appropriate function associated with the command in the URL
ConstructStream	Constructs a CHtmlStream object.
GetExtensionVersion	Gets the version number that the DLL extension is based on.
HttpExtensionProc	Uses the callback functions to read client data and decide what action to take.
OnParseError	Constructs a description of the error to be returned to the client.
OnWriteBody	Writes data back to the client.
TerminateExtension	Provides a safe way to clean up threads and complete other shutdown type activities.

Attributes

AddHeader	Adds headers to a response before it is sent to the client.
EndContent	Inserts closing HTML tags into a CHtmlStream object to be returned to the client. Override to change or omit the default tags.
GetTitle	Gets the title of an HTML document to be sent to the client.
InitInstance	Initializes the **CHttpServer** object.
StartContent	Inserts opening HTML tags into a CHtmlStream object to be returned to the client. Override to change or omit the default tags.
WriteTitle	Inserts the title between the appropriate HTML tags in the CHtmlStream object to be returned to the client. Override to provide a different title.

See Also

Reference

CHttpServer Class
Hierarchy Chart
CHtmlStream Class
CHttpServerContext Class

CHttpServer Member Functions

For information about the member functions in **CHttpServer**, see CHttpServer Members.

CHttpServer::AddHeader

Call this member function to add a header to the response before the response is sent to the client.

```
void AddHeader(
   CHttpServerContext* pCtxt,
   LPCTSTR pszString
) const;
```

Parameters

pCtxt

A pointer to a **CHttpServerContext** object.

pszString

A pointer to a string.

Remarks

Use **AddHeader** to append your own headers to those the server supplies when it receives CHttpServerContext::ServerSupportFunction **HSE_REQ_SEND_RESPONSE_HEADERS**. The extra header provides the client with more information.

For example, call **AddHeader** to specify your own "content-type," then call it to specify an encoding, and then call it once more to insert the "content-length" header. After you have called **AddHeader** as many times as you need, use << to stream your output until you are done.

Note
Once you put data in the HTML stream in the server context, do not call **AddHeader** again. If you do, your HTML stream will not work properly.

Example

The following code demonstrates how to use **AddHeader** to add a customer header. In this instance customer headers represents a cookie. szCookie is supplied in the request in following fashion:

```
http://server/scripts/MfcCookie.dll?SetCookie?cookie=Hello

BEGIN_PARSE_MAP(CMfcCookieExtension, CHttpServer)
   ...
   ON_PARSE_COMMAND(SetCookie, CMfcCookieExtension, ITS_PSTR)
      ON_PARSE_COMMAND_PARAMS ("cookie")
   ...
END_PARSE_MAP(CMfcCookieExtension)


void CMfcCookieExtension::SetCookie(CHttpServerContext * pCtxt,
    LPCTSTR szCookie)
{
    enum {
        sizeOfHeader = 256
    };
    char szHeader [sizeOfHeader];
    StartContent(pCtxt);
    sprintf_s(szHeader, sizeOfHeader, "Set-Cookie: LeonBrCookie=%s;"
              " expires=Friday, 22-May-99 13:00:00 GMT;  path=/;\r\n",
              szCookie);
    AddHeader(pCtxt, szHeader);
    AddHeader(pCtxt, "Expires: 0\r\n");
    *pCtxt <<"Cookie name/value: <b>"<< szHeader <<"</b> is set <br>";
    EndContent(pCtxt);
}
```

Here is an example of a function that creates an on-the-fly web-page:

```
void CHelloExtension::Default(CHttpServerContext* pCtxt)
{
    AddHeader(pCtxt, "Content-type = text/plain\r\n");
    (*pCtxt) << "Hello world!\r\n";
}
```

See Also

Reference

CHttpServer Class

Hierarchy Chart

CHttpServerContext Class

CHttpServerContext::ServerSupportFunction

Other Resources

CHttpServer Members

CHttpServer::CallFunction

Called by the framework to find and execute the appropriate function associated with the command in the URL.

```
virtual int CallFunction(
   CHttpServerContext* pCtxt,
   LPTSTR pszQuery,
    LPTSTR pszCommand
);
```

Parameters

pCtxt

Pointer to a CHttpServerContext object.

pszQuery

A pointer to a query. Specific to the type of command received from the client. See **Remarks** for more information.

pszCommand

Either a pointer to a query or NULL. Specific to the type of command received from the client. See **Remarks** for more information.

Return Value

A value of one of the following enum types:

Enum value	Description
callOK	The function call was successful.
callParamRequired	A required parameter was missing.
callBadParamCount	There were too many or too few parameters.
callBadCommand	The command name was not found.
callNoStackSpace	No stack space was available.
callNoStream	No CHtmlStream was available.
callMissingQuote	A parameter had a bad format.
callMissingParams	No parameters were available.
callBadParam	A parameter had a bad format (i.e., only one quote).

Remarks

Below is a breakdown between the types of methods received and the parameters:

Method type	***pszQuery***	***pszCommand***
GET	A pointer to the **EXTENSION_CONTROL_BLOCK** structure query string.	**NULL**
POST	A pointer to a query sent in the body of the command.	Pointer to the **EXTENSION_CONTROL_BLOCK** structure query string.

Note

Fill-out forms authors are advised to use only the POST method because of browser inconsistencies, and because GET methods are limited to a 1024-byte buffer. When writing forms for ISAPI, either use only the POST method, or design the ISA so that only the default function handles the form.

For example, some browsers sending a form via GET with an action of:

```
TestLet.DLL?Command
```

will truncate `Command` and send:

```
TestLet.DLL?name=value
```

instead of the correct command:

```
TestLet.DLL?Command?name=value
```

By truncating `Command`, the browser removes the association to the ISA function needed to map the request. Unless the function `Command` is the default function, the form will not be handled correctly.

If you want to handle parsing of the **EXTENSION_CONTROL_BLOCK** structure function **lpszQueryString** yourself, override **CallFunction** and do not use the **PARSE_MAP** macros. See Internet Server API (ISAPI) Parse Maps for more information on using the parse map macros.

See Also

Reference

CHttpServer Class
Hierarchy Chart
CHttpServerContext Class

Other Resources

CHttpServer Members
Internet Server API (ISAPI) Parse Maps

CHttpServer::CHttpServer

The run-time calls this function when constructing a **CHttpServer** object.

```
        CHttpServer(
    TCHAR cDelimiter = '&'
);
```

Parameters
cDelimiter

A character identifying the token delimiter. By default, this delimiter is '&'.

Remarks

Only one instance of **CHttpServer** may exist for each module. Once a **CHttpServer** object is created, it can be initialized with InitInstance.

After the ISA has been initiated by a client command and acted upon by the server, the client receives a response page that reflects the *cDelimiter* parameter in the URL. The *cDelimiter* parameter separates the command's arguments that are parsed by the parse map macros ON_PARSE_COMMAND and ON_PARSE_COMMAND_PARAMS.

Example

If the client initiates an ISA to view a colorized JPEG image from the URL **http://www.contoso.com/**, the command sent to the server could look like this:

```
http://www.contoso.com/scripts/Books.dll?Colorize
```

where **Colorize** is the command initiating the **Colorize** function.

The URL that the server returns to the client would look like this:

```
http://www.contoso.com/scripts/Books.dll?Colorize?Target=Picture&Format=JPEG
```

The *cDelimiter* default delimiter *&* appears in the client's URL between the two parameters *Picture* and *Format* of the function **Colorize**.

See **ON_PARSE_COMMAND** and **ON_PARSE_COMMAND_PARAMS** for more information about parsing commands.

See Also
Reference
CHttpServer Class
Hierarchy Chart
Other Resources
CHttpServer Members

CHttpServer::ConstructStream

This member function is called by the framework to construct a **CHtmlStream** object.

```
virtual CHtmlStream* ConstructStream( );
```

Return Value

A pointer to a CHtmlStream object.

Remarks

Override this member function to create an instance of your own class to give it functionality other than the default.

See the constructor CHtmlStream::CHtmlStream for information about why you might override **ConstructStream** and provide special functionality for a **CHtmlStream** object.

See Also

Reference

CHttpServer Class

Hierarchy Chart

Other Resources

CHttpServer Members

CHttpServer::EndContent

This member function is called by the framework to insert the closing HTML tags "`</Body>`" and "`</HTML>`" into an HTML document to be returned to the client.

```
virtual void EndContent(
   CHttpServerContext* pCtxt
) const;
```

Parameters

pCtxt

A pointer to a **CHttpServerContext** object. Cannot be **NULL**.

Remarks

Override this member function to implement a behavior different from the default. For example, override if you are returning a stream type other than an HTML stream (like a JPEG image).

See Also

Reference

CHttpServer Class
Hierarchy Chart
CHttpServer::StartContent

Other Resources

CHttpServer Members

CHttpServer::GetExtensionVersion

This member function is called by the framework when it loads an ISA.

```
virtual BOOL GetExtensionVersion(
   HSE_VERSION_INFO *pVer
);
```

Parameters

pVer

A pointer to the HSE_VERSION_INFO structure containing version information for the server and fields for the client to indicate version number, notifications, and priority desired. There is also a space for the filter application to register a small description of itself.

Return Value

Nonzero if successful; otherwise zero.

Remarks

GetExtensionVersion gets the version number of the specification the DLL extension is based on. It also provides a short text description for server administrators.

GetExtensionVersion is one of two necessary entry points for an ISA. The second necessary entry point is the function HttpExtensionProc. Both of these are provided by MFC, with default implementation. Call the default implementation to set the version, and then override to replace the default text string with your own short description.

See Also

Reference

CHttpServer Class
Hierarchy Chart
HSE_VERSION_INFO Structure
CHttpServer::HttpExtensionProc

Other Resources

CHttpServer Members

CHttpServer::GetTitle

This member function is called by the framework to get the title of an HTML document to be sent to the client.

```
virtual LPCTSTR GetTitle( ) const;
```

Return Value

A pointer to a string containing the title.

Remarks

Override to supply your own title.

See Also

Reference

CHttpServer Class

Hierarchy Chart

Other Resources

CHttpServer Members

CHttpServer::HttpExtensionProc

This member function is called by the framework for each request for an ISA.

```
virtual DWORD HttpExtensionProc(
   EXTENSION_CONTROL_BLOCK *pECB
);
```

Parameters

pECB

A pointer to an **EXTENSION_CONTROL_BLOCK** structure.

Return Value

One of the following HTTP Server Extension messages:

HSE_STATUS_SUCCESS

The ISA has finished processing and the server can disconnect and free up allocated resources.

HSE_STATUS_SUCCESS_AND_KEEP_CONN

The ISA has finished processing and the server should wait for the next HTTP request if the client supports persistent connections. The application should only return this if it was able to send the correct content-length header to the client. The server is not required to keep the session open.

HSE_STATUS_PENDING

The ISA has queued the request for processing and will notify the server when it has finished. See **HSE_REQ_DONE_WITH_SESSION** under CHttpServerContext::ServerSupportFunction.

HSE_STATUS_ERROR

The ISA has encountered an error while processing the request and the server can disconnect and free up allocated resources.

Remarks

HttpExtensionProc uses the callback functions to read client data and decide what action to take. Before returning to the server, a properly formatted response must be sent to the client via either the CHttpServerContext::WriteClient or the CHttpServerContext::ServerSupportFunction member function.

The default implementation of **HttpExtensionProc** is recommended; however you can override this member function to customize the implementation.

See Also

Reference

CHttpServer Class
Hierarchy Chart
CHttpServerContext::WriteClient
CHttpServerContext::ServerSupportFunction

Other Resources

CHttpServer Members

CHttpServer::InitInstance

This member function is called by the framework to initialize a **CHttpServer** object.

```
virtual BOOL InitInstance(
   CHttpServerContext* pCtxt
);
```

Parameters

pCtxt

A pointer to a **CHttpServerContext** object.

Return Value

Nonzero if initialization is successful; otherwise 0.

Remarks

InitInstance is called in **CHttpServer::HttpExtensionProc**, which is called by the framework for each request for an ISA.

Override this member function to provide **CHttpServer** custom initialization.

See Also

Reference

CHttpServer Class

Hierarchy Chart

Other Resources

CHttpServer Members

CHttpServer::OnParseError

Called by the framework to parse errors.

```
virtual BOOL OnParseError(
   CHttpServerContext* pCtxt,
   int nCause
);
```

Parameters

pCtxt

A pointer to a CHttpServerContext object that contains an EXTENSION_CONTROL_BLOCK with its **dwHttpStatusCode** set to the code for the error that has occurred. These status codes are:

- **HTTP_STATUS_BAD_REQUEST**
- **HTTP_STATUS_AUTH_REQUIRED**
- **HTTP_STATUS_FORBIDDEN**
- **HTTP_STATUS_NOT_FOUND**
- **HTTP_STATUS_SERVER_ERROR**
- **HTTP_STATUS_NOT_IMPLEMENTED**

nCause

The cause of the error. Can be one of the following values:

Enum type	Description
callOK	**OnParseError** handled the error.
callParamRequired	A required parameter was missing.
callBadParamCount	There were too many or too few parameters.
callBadCommand	The command name was not found.
callNoStackSpace	No stack space was available.
callNoStream	No CHtmlStream was available.
callMissingQuote	A parameter is missing a quote mark.
callMissingParams	No parameters were available.
callBadParam	A parameter had a bad format.

Return Value

Nonzero error is successfully parsed; otherwise 0.

Remarks

Once the error is identified, the message associated with the cause of the error is returned to the client either in an HTML stream or in a CHttpServerContext::WriteClient message.

Override this member function to customize the error parsing.

Example

The following code demonstrates how to handle various parsing errors in the ISAPI and notify users about them.

```
BOOL CMfcCookieExtension::OnParseError(CHttpServerContext* pCtxt, int nCause)
{
    // TODO: Add your specialized code here and/or call the base class
    ISAPITRACE1 ("Parser: %d\n", nCause);
    switch (nCause)
    {
        case callOK:
            ISAPITRACE ("No errors\n");
        break;

        case callParamRequired:
            *pCtxt << "A required parameter was missing<br>";
        break;

        case callBadParamCount:
            *pCtxt <<"There were too many or too few parameters<br>";
        break;

        case callBadCommand:
            *pCtxt <<"The command name was not found<br>";
        break;

        case callNoStackSpace:
            *pCtxt <<"No stack space was available<br>";
        break;

        case callNoStream:
            *pCtxt <<"No CHtmlStream was available<br>";
        break;

        case callMissingQuote:
            *pCtxt <<"A parameter had a bad format <br>";
        break;

        case callMissingParams:
            *pCtxt <<"No parameters were available <br>";
        break;

        case callBadParam:
            *pCtxt <<"A parameter had a bad format (ie, only one quote) <br>";
        break;
    }

   // Return TRUE, because error was handled.
    return  TRUE;
}
```

See Also

Reference

CHttpServer Class

Hierarchy Chart

Other Resources

CHttpServer Members

CHttpServer::OnWriteBody

Override this function to write data back to the server.

```
virtual BOOL OnWriteBody(
   CHttpServerContext* pCtxt,
   LPBYTE pbContent,
   DWORD dwSize,
   DWORD dwReserved = 0
);
```

Parameters

pCtxt

A pointer to a CHttpServerContext object

pbContent

The content to be written to the client.

dwSize

The number of bytes to be written to the client.

dwReserved

Reserved for future use; must be 0.

Return Value

Zero for a failure, and nonzero for success.

Remarks

MFC will call this function only when all of these conditions are true:

- You have streamed data in the CHtmlStream object associated with the CHttpServerContext object.
- You have not set CHttpServerContext::m_bSendHeaders to **FALSE**.
- No other error has occurred in the processing of the request.

This function writes data back to the client by calling CHttpServerContext::WriteClient. It may make several calls to **CHttpServerContext::WriteClient** if the **CHttpServerContext** object has set a chunk size.

See Also

Reference

CHttpServer Class
Hierarchy Chart
CHttpServerContext::SetChunkSize
CHttpServerContext::GetChunkSize

Other Resources

CHttpServer Members

CHttpServer::StartContent

This member function is called by the framework to insert the starting HTML tags "`<Body>`" and "`<HTML>`" into an HTML document to be returned to the client.

```
virtual void StartContent(
   CHttpServerContext* pCtxt
) const;
```

Parameters

pCtxt

A pointer to a CHttpServerContext object.

Remarks

Override this member function to implement a behavior different from the default. For example, override if you are returning a stream type other than an HTML stream (like a JPEG image).

See Also

Reference

CHttpServer Class
Hierarchy Chart
CHttpServer::EndContent

Other Resources

CHttpServer Members

CHttpServer::TerminateExtension

TerminateExtension gives you a safe way to clean up threads and complete other shutdown activities.

```
virtual BOOL TerminateExtension(
   DWORD dwFlags
);
```

Parameters
dwFlags

A bitfield consisting of the following values:

- **HSE_TERM_ADVISORY_UNLOAD** The server wants to unload the extension. The extension can return **TRUE** if OK to unload or **FALSE** if the server should not unload the extension.
- **HSE_TERM_MUST_UNLOAD** The server is indicating the extension is about to be unloaded, the extension cannot refuse; the return value should be ignored.

Return Value

If *dwFlags* is set to HSE_TERM_ADVISORY_UNLOAD, your override of this function can return **TRUE** to indicate that it can safely exit or **FALSE** to indicate that it is not ready to unload. If *dwFlags* is set to HSE_TERM_MUST_UNLOAD, the server is forcing the extension out of memory and the return value is ignored.

See Also
Reference
CHttpServer Class
Hierarchy Chart
Other Resources
CHttpServer Members

CHttpServer::WriteTitle

This member function is called by the framework to write the title to insert between the appropriate HTML tags on the document to be transmitted back to the client.

```
virtual void WriteTitle(
   CHttpServerContext* pCtxt
) const;
```

Parameters

pCtxt

A pointer to a CHttpServerContext object.

Remarks

The default implementation writes the title returned from GetTitle between the HTML tags "`<Title>`" and "`</Title>`". Override this member function to provide a different title.

See Also

Reference

CHttpServer Class
Hierarchy Chart
CHttpServer::GetTitle

Other Resources

CHttpServer Members

CHttpServerContext Class

Provides the tools that a CHttpServer object needs to process data that a client has sent to the HTTP server.

```
class CHttpServerContext
```

Remarks

CHttpServerContext does not have a base class.

When Microsoft Internet Information Services receives a request from a client browser, a **CHttpServer** object is created and initialized, and a **CHttpServerContext** object is created. As the server extension DLL processes requests, it uses **CHttpServerContext** member functions to perform tasks.

A **CHttpServerContext** object exists separately from a **CHttpServer** object in order to allow multithreading. Only one **CHttpServer** exists in a module, but a server might be required to process multiple client requests simultaneously.

CHttpServer creates a **CHttpServerContext** for each request to handle these multiple requests. A **CHttpServer** uses multiple **CHttpServerContext** objects to run in separate threads. This design allows simultaneous, multiple calls to the **CHttpServer** object by different client connections.

When an extension DLL (ISA) is called, the member function ServerSupportFunction provides the ISA with some general-purpose functions as well as functions that are specific to HTTP server implementation.

If the server extension must communicate something — for example, an error — back to the client immediately, call WriteClient. Otherwise, the server should output a message to the client to the **m_pStream** data member owned by the *pCtxt* parameter passed to it.

Requirements

Header: afxisapi.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart

Concepts

How Do I Use the MFC ISAPI Classes Without Linking to the MFC DLL?

Other Resources

CHttpServerContext Members

CHttpServerContext Members

Data Members

m_bSendHeaders	A flag to control whether headers are automatically included in an HTTP response.
m_pECB	A pointer to an **EXTENSION_CONTROL_BLOCK** structure.
m_pStream	A pointer to a CHtmlStream.

Construction

CHttpServerContext	Constructs a **CHttpServerContext** object.

Operations

GetChunkSize	Retrieve the current chunk size.
GetServerVariable	Copies information relating to an HTTP connection, or to the server itself, into a supplied buffer.
ReadClient	Reads information from the body of the Web client's HTTP request into the buffer supplied by the caller.
ServerSupportFunction	Provides **ISAs** with some general-purpose functions as well as functions that are specific to HTTP server implementation.
SetChunkSize	Set the chunk size for the current context.
TransmitFile	Sends the data from a file from the server to the requester.
WriteClient	Sends information to the HTTP client immediately.

Operators

operator <<	Writes data into a stream.

See Also

Reference

CHttpServerContext Class
Hierarchy Chart
CHttpServer Class

CHttpServerContext Member Functions

For information about the member functions in **CHttpServerContext**, see CHttpServerContext Members.

CHttpServerContext::CHttpServerContext

This member function is called by the framework during the construction of a **CHttpServerContext** object.

```
explicit CHttpServerContext(
   EXTENSION_CONTROL_BLOCK* pECB
);
```

Parameters

pECB

A pointer to an **EXTENSION_CONTROL_BLOCK** data structure.

See Also

Reference

CHttpServerContext Class

Hierarchy Chart

Other Resources

CHttpServerContext Members

CHttpServerContext::GetChunkSize

Call this member function to retrieve the current chunk size.

```
DWORD GetChunkSize( ) const;
```

Return Value

The chunk size.

Example

See the example for CHttpServerContext::SetChunkSize.

See Also

Reference

CHttpServerContext Class
Hierarchy Chart
CHttpServerContext::SetChunkSize
CHttpServer::OnWriteBody

Other Resources

CHttpServerContext Members

CHttpServerContext::GetServerVariable

This function copies information relating to an HTTP connection, or to the server itself, into a buffer supplied by the caller.

```
BOOL GetServerVariable(
   LPTSTR lpszVariableName,
   LPVOID lpvBuffer,
   LPDWORD lpdwSize
);
```

Parameters

lpszVariableName

Null terminated string indicating which variable is being requested. See the Remarks section for a list of current variables.

lpvBuffer

Pointer to buffer to receive the requested information.

lpdwSize

Pointer to **DWORD** indicating the number of bytes available in the buffer. On successful completion the **DWORD** contains the number of bytes transferred into the buffer (including the null-terminating byte).

Return Value

Nonzero if successful, otherwise 0. If the call fails, the Windows function GetLastError may be called to determine the cause of the error. Possible error values include:

Value	Meaning
ERROR_INVALID_PARAMETER	Bad connection handle.
ERROR_INVALID_INDEX	Bad or unsupported variable identifier.
ERROR_INSUFFICIENT_BUFFER	Buffer too small, required size returned in **lpdwSize*.
ERROR_MORE_DATA	Buffer too small, only part of data returned. The total size of the data is not returned.
ERROR_NO_DATA	The data requested is not available.

Remarks

Possible values for *lpszVariableNames* include:

Value	Meaning
AUTH_TYPE	Contains the type of authentication used. For example, if Basic authentication is used, the string will be "Basic". For Windows NT Challenge-response, it will be "NTLM". Other authentication schemes will have other strings. Because new authentication types can be added to Internet Server, it is not possible to list all possible strings. If the string is empty, then no authentication is used.
CONTENT_LENGTH	The number of bytes which the script can expect to receive from the client.
CONTENT_TYPE	The content type of the information supplied in the body of a POST request.

GATEWAY_INTERFACE	The revision of the CGI specification to which this server complies. The current version is CGI/1.1.
PATH_INFO	Additional path information, as given by the client. This comprises the trailing part of the URL after the extension DLL (script) name but before the query string (if any).
PATH_TRANSLATED	This is the value of **PATH_INFO**, but with any virtual path name expanded into a directory specification.
QUERY_STRING	The information which follows the ? in the URL which referenced this extension DLL.
REMOTE_ADDR	The IP address of the client.
REMOTE_HOST	The hostname of the client.
REMOTE_USER	This contains the username supplied by the client and authenticated by the server.
REQUEST_METHOD	The HTTP request method.
SCRIPT_NAME	The name of the extension DLL that is being executed.
SERVER_NAME	The server's hostname (or IP address) as it should appear in self-referencing URLs.
SERVER_PORT	The TCP/IP port on which the request was received.
SERVER_PROTOCOL	The name and version of the information retrieval protocol relating to this request. Normally HTTP/1.0.
SERVER_SOFTWARE	The name and version of the web server under which the ISA or server extension DLL program is running.

ALL_HTTP	All HTTP headers that were not already parsed into one of the above variables. These variables are of the form HTTP_<header field name>.

HTTP_ACCEPT	Special case HTTP header. Values of the Accept: fields are concatenated, separated by ", ". For example, if the following lines are part of the HTTP header:
	`accept: */*; q=0.1`
	`accept: text/html`
	`accept: image/jpeg`
	then the **HTTP_ACCEPT** variable will have a value of:
	`*/*; q=0.1, text/html, image/jpeg`

See Also

Reference

CHttpServerContext Class

Hierarchy Chart

Other Resources

CHttpServerContext Members

CHttpServerContext::ReadClient

Call this member function to read information from the body of the Web client's HTTP request into the buffer supplied by the caller.

```
BOOL ReadClient(
   LPVOID lpvBuffer,
   LPDWORD lpdwSize
);
```

Parameters

lpvBuffer

Pointer to the buffer area to receive the requested information.

lpdwSize

Pointer to **DWORD** indicating the number of bytes available in the buffer. On return **lpdwSize* will contain the number of bytes actually transferred into the buffer.

Return Value

Nonzero if successful, otherwise 0. If the socket used by the server to listen to the client is closed, it will return nonzero, but with zero bytes read.

If the call fails, the Windows function GetLastError may be called to determine the cause of the error.

Remarks

ReadClient might be used to read data from an HTML form that uses the POST method. If more than **lpdwSize* bytes are immediately available to be read, **ReadClient** will return after transferring that amount of data into the buffer. Otherwise it will block incoming data and wait for buffer space to become available.

See Also

Reference

CHttpServerContext Class
Hierarchy Chart
CHttpServerContext::WriteClient

Other Resources

CHttpServerContext Members

CHttpServerContext::ServerSupportFunction

Call this member function to provide the ISA with some general purpose functions as well as functions that are specific to HTTP server implementation.

```
BOOL ServerSupportFunction(
   DWORD dwHSERRequest,
   LPVOID lpvBuffer,
   LPDWORD lpdwSize,
   LPDWORD lpdwDataType
);
```

Parameters

dwHSERRequest

An HTTP Server Extension value. See the Remarks section for a list of the supported values.

lpvBuffer

When used with **HSE_REQ_SEND_RESPONSE_HEADER**, it points to a null terminated optional status string (for example, "401 Access Denied"). If this buffer is null, a default response of "200 OK" will be sent by this function. When used with **HSE_REQ_DONE_WITH_SESSION**, it points to a **DWORD** indicating the status code of the request.

lpdwSize

When used with **HSE_REQ_SEND_RESPONSE_HEADER**, it points to the size of the buffer *lpdwDataType*.

lpdwDataType

When used with **HSE_REQ_SEND_RESPONSE_HEADER**, this is a null-terminated string pointing to optional headers or data to be appended and sent with the header. If this is **NULL**, the header will be terminated by a "\r\n" pair.

Note
General purpose functions should have a *dwHSERequest* value larger than **HSE_REQ_END_RESERVED**. Values up to **HSE_REQ_END_RESERVED** are reserved for mandatory ServerSupportFunctions and should not be used.

Return Value

Nonzero if successful, otherwise 0.

Remarks

The HTTP Server Extension value represented by *dwHSERRequest* can be one of the following:

HSE_REQ_SEND_URL_REDIRECT_RESP

Sends a `302`(`URL Redirect`) message to the client. No further processing is needed after the call. This operation is similar to specifying "URI: <URL>" in a CGI script header. The variable *lpvBuffer* should point to a null terminated URL string. Variable *lpdwSize* should have the size of *lpvBuffer*. Variable *lpdwDataType* is ignored.

HSE_REQ_SEND_URL

Sends the data specified by the URL to the client as if the client had requested that URL. The Null terminated URL pointed to by *lpvBuffer* MUST be on the server and must not specify protocol information (i.e. it must begin with a '/'). No further processing is required after this call. Variable *lpdwSize* points to a **DWORD** holding the size of *lpvBuffer*. Variable *lpdwDataType* is ignored.

HSE_REQ_SEND_RESPONSE_HEADER

Sends a complete HTTP server response header including the status, server version, message time and MIME version. The **ISA** or server extension should append other HTTP headers at the end such as the Content-Type, Content-Length, and so forth, followed by an extra "\r\n".

HSE_REQ_DONE_WITH_SESSION

If the ISA or server extension wants to hold onto the session because it has extended processing requirements, it needs to tell

the server when the session is finished so the server can close it and free the related structures. Variables *lpvBuffer*, *lpdwSize*, and *lpdwDataType* are all ignored.

HSE_REQ_END_RESERVED

Functions higher than this value are server specific and may not be available on all web servers that support ISAPI.

HSE_REQ_MAP_URL_TO_PATH

The *lpvBuffer* parameter is a pointer to the buffer that contains the logical path on entry and the physical path on exit. The *lpdwSize* parameter is a pointer to the **DWORD** containing the size of the buffer passed in *lpvBuffer* on entry, and the number of bytes placed in the buffer on exit. The *lpdwDataType* parameter is ignored). A Microsoft-specific extension.

HSE_REQ_GET_SSPI_INFO

The *lpvBuffer* is filled in with the context handle and **lpdwDataType* is filled in with the credential handle. A context handle specifies a pointer type or a type identifier. A credential handle specifies authentication and authorization.

HSE_REQ_TRANSMIT_FILE

See TransmitFile for an easier method of transmitting a file to a requester.

Note
The server does not ensure that the buffers are large enough before filling in the handles, and *lpdwSize* is not updated to r eflect the amount of data copied into the *lpvBuffer* buffer. Since these are fixed size structures, it is assumed the pointers p assed in are pointers to the structure and must be at least as large as the request structures.

See Also

Reference

CHttpServerContext Class
Hierarchy Chart
CHttpFilterContext::ServerSupportFunction
CHttpServerContext::TransmitFile

Other Resources

CHttpServerContext Members

CHttpServerContext::SetChunkSize

Call this member function to set the chunk size for the current context.

```
DWORD SetChunkSize(
   DWORD dwNewSize
);
```

Parameters

dwNewSize

New chunk size.

Return Value

The previous chunk size.

Remarks

The chunk size is the number of bytes that MFC will send back in an individual write operation when writing data from the body of the stream within the **CHttpServerContext** object back to the client. The chunk size is initially zero when the context is created. A chunk size of zero means that MFC will attempt to write the entire stream in one operation. Any other value causes the framework to divide the body of the response into chunks no larger than the specified size.

Setting a chunk size is not always necessary, but it can be required when you use HTTPS, because HTTPS is limited to writing 14 kilobytes per operation at a maximum. HTTPS is a variant of the HTTP protocol that uses the secure-sockets layer for its transport protocol.

Example

The following code demonstrates how to check chunk size used by MFC.

```
void CMfcCookieExtension::Default(CHttpServerContext* pCtxt)
{

    DWORD dwChunkOld = pCtxt->SetChunkSize (10);
    DWORD dwChunkNew = pCtxt->GetChunkSize();
    StartContent(pCtxt);
    WriteTitle(pCtxt);
    *pCtxt << "Chunk size was changed from: " << (long) dwChunkOld << "<br>";
    *pCtxt << "to: " << (long) dwChunkNew << "<br>";
    EndContent(pCtxt);
}
```

See Also

Reference

CHttpServerContext Class
Hierarchy Chart
CHttpServerContext::GetChunkSize
CHttpServer::OnWriteBody

Other Resources

CHttpServerContext Members

CHttpServerContext::TransmitFile

Call this member function to transmit a file to a requester.

```
BOOL TransmitFile(
   HANDLE hFile,
   DWORD dwFlags = HSE_IO_DISCONNECT_AFTER_SEND,
   LPVOID pstrHeader = NULL,
   DWORD dwHeaderLen = 0,
   LPVOID pstrTrailer = NULL,
   DWORD dwTrailerLen = 0
);
```

Parameters

hFile

A handle to the file containing the data to transmit. The file must have been opened with **FILE_FLAG_SEQUENTIAL_SCAN** and **FILE_FLAG_OVERLAPPED**.

dwFlags

The attributes associated with the file. By default, **HSE_IO_DISCONNECT_AFTER_SEND**. See **ServerSupportFunction** in the *Platform SDK* for a list of possible attributes.

pstrHeader

A pointer to the string containing the header to send before transmitting the file's data. The header conveys to the receiving browser the type of data being sent and how it should be displayed. If **NULL**, the header is not sent.

dwHeaderLen

The length of the header. By default, 0. If *pstrHeader* is not **NULL**, and *dwHeaderLen* is 0, the function will use the Win32 function lstrlen to find the length of the header. (In this case, *pstrHeader* must be a null-terminated string.)

pstrTrailer

A pointer to the string containing the trailer to send after transmitting the file's data. If **NULL**, the trailer is not sent.

dwTrailerLen

The length of the trailer. By default, 0. If *pstrTrailer* is not **NULL**, and *dwTrailerLen* is 0, the function will use the Win32 function **lstrlen** to find the length of the trailer. (In this case, *pstrTrailer* must be a null-terminated string.)

Return Value

TRUE if the file's data was sent; otherwise, **FALSE**. To get extended error information, call GetLastError. The function returns **FALSE** if an overlapped I/O operation is not complete before **TransmitFile** returns. In that case, **GetLastError** returns **ERROR_IO_PENDING**.

Remarks

This member function performs the operations of the **ServerSupportFunction** command flag **HSE_REQ_TRANSMIT_FILE**. The server performs this operation asynchronously. The server submits this operation to its internal asynchronous I/O queue and returns to the caller. MFC alters the context object to return **HSE_STATUS_PENDING**, and will use **HSE_DONE_WITH_SESSION** when the callback indicates completion of the I/O. MFC also sets m_bSendHeaders in the context object **FALSE**, because headers are sent automatically.

See Also

Reference

CHttpServerContext Class
Hierarchy Chart
CHttpServerContext::ServerSupportFunction

Other Resources

CHttpServerContext Members

CHttpServerContext::WriteClient

Call this member function to send information to the HTTP client immediately.

```
BOOL WriteClient(
   LPVOID lpvBuffer,
   LPDWORD lpdwBytes,
   DWORD dwReserved = 0
);
```

Parameters

lpvBuffer

Pointer to the buffer where the data is to be written.

lpdwBytes

Pointer to a DWORD that holds the number of characters to write from the buffer referenced by *Buffer*.

dwReserved

Reserved for future use.

Return Value

Nonzero if successful, otherwise 0. If the call fails, the Windows function GetLastError may be called to determine the cause of the error.

Remarks

For example, use **WriteClient** to send an error message.

Example

This example demonstrates how to use **WriteClient** to immediately send binary data from the ISAPI extension. The example opens a binary file, sets the appropriate Context-Type header, and sends it to the browser. **m_bSendHeaders** is set to FALSE so that MFC won't add any types of headers.

This ISAPI extension can be used to stream binary data directly to the browser on the HTML page.

```
<HTML>.
You will see image here:.
<IMG SRC="/scripts/myisapi.dll?header=image/gif&file=c:\temp\myfile.gif" >
</HTML>

// Parse map: indicates that the default function takes two name/value
// pairs: a content-type header that will be generated by the ISAPI
// extension and a path to the local binary file.
BEGIN_PARSE_MAP(CSendBin5Extension, CHttpServer)
   // TODO: insert your ON_PARSE_COMMAND() and
   // ON_PARSE_COMMAND_PARAMS() here to hook up your commands.
   // For example:

   ON_PARSE_COMMAND(Default, CSendBin5Extension, ITS_PSTR ITS_PSTR)
   ON_PARSE_COMMAND_PARAMS("file header")

   DEFAULT_PARSE_COMMAND(Default, CSendBin5Extension)
END_PARSE_MAP(CSendBin5Extension)

void CSendBin5Extension::Default(CHttpServerContext* pCtxt,
   LPTSTR szFile, LPTSTR szHeader)
{
   CHAR szHeaders [256];
   // Use headers would be supplied via name/value pair to the default.
   // They would be in this form: image/gif.
   wsprintf (szHeaders, "Content-Type: %s\r\n\r\n", szHeader);
```

```
    DWORD dwRead ;
    DWORD dwSize = lstrlen (szHeaders);
    VOID * pBuff;

    // Turn off sending header by MFC.
    pCtxt->m_bSendHeaders = FALSE;

    // Send our own headers.
    if (!pCtxt->ServerSupportFunction (HSE_REQ_SEND_RESPONSE_HEADER,
       NULL, &dwSize, (LPDWORD ) szHeaders))
    {
       // Report an error.
       ISAPITRACE1 ("ServerSupportFunction failed: %d\n", GetLastError());
       return;
    }

    // Open local file. File name would be supplied as a parameter to
    // the default in this form: c:\temp\myfile.gif
    HANDLE hFile = CreateFile (szFile, GENERIC_READ, FILE_SHARE_READ,
       (LPSECURITY_ATTRIBUTES) NULL,
       OPEN_EXISTING, FILE_ATTRIBUTE_READONLY, (HANDLE) NULL);
    if (hFile == INVALID_HANDLE_VALUE)
    {
       ISAPITRACE1 ("CreateFile error: %d\n", GetLastError());
       return;
    }

    CHAR szBuffer [2048];
    do
    {
       // read chunk of the file
       if (!ReadFile (hFile, szBuffer, 2048, &dwRead, NULL))
       {
          ISAPITRACE1 ("ReadFile error: %d\n", GetLastError());
          return;
       }
       if (!dwRead)
          // EOF reached, bail out
          break;

       // Send binary chunk to the browser
       if (!pCtxt->WriteClient( szBuffer, &dwRead, 0))
       {
          ISAPITRACE1 ("WriteClient error: %d\n", GetLastError());
          return;
       }
       ISAPITRACE ("Buffer sent\n");
    }
    while (1);
    CloseHandle (hFile);
    return;
 }
```

See Also

Reference

CHttpServerContext Class

Hierarchy Chart

CHttpServerContext::ReadClient

Other Resources

CHttpServerContext Members

CHttpServerContext Operators

For information about the operators in **CHttpServerContext**, see CHttpServerContext Members.

CHttpServerContext::operator <<

The **CHttpServerContext** insertion (**<<**) operator writes the specified string or integer to the HTML stream owned by the **CHttpServerContext** object.

```
CHttpServerContext& operator<<(
   LPCTSTR psz
);
CHttpServerContext& operator<<(
   long int dw
);
CHttpServerContext& operator<<(
   short int w
);
CHttpServerContext& operator<<(
   const CHtmlStream& stream
);
CHttpServerContext& operator<<(
   double d
);
CHttpServerContext& operator<<(
   float f
);
CHttpServerContext& operator<<(
   const CLongBinary& blob
);
CHttpServerContext& operator<<(
   const CByteArray& array
);
```

Remarks

The string version of the operator writes the string without modification. The integer overrides format the value as decimal text before writing it.

The operator parameters correspond directly to the **ITS_*** types that you can use in the ISAPI parse map.

See Also

Reference

CHttpServerContext Class

Hierarchy Chart

Concepts

Parse Maps

Other Resources

CHttpServerContext Members

Internet Server API (ISAPI) Parse Maps

CHttpServerContext Data Members

For information about the data members in **CHttpServerContext**, see CHttpServerContext Members.

CHttpServerContext::m_bSendHeaders

A boolean flag that controls whether MFC automatically sends headers in an HTTP response.

Remarks

When MFC constructs a **CHttpServerContext**, it sets **m_bSendHeaders** to **TRUE**. If you set **m_bSendHeaders** to **FALSE**, no headers are sent. Suppressing automatic headers is useful if you want to keep a connection alive or you want to construct your own headers using CHttpServer::AddHeader.

Example

See the example for CHttpServerContext::WriteClient.

See Also

Reference

CHttpServerContext Class
Hierarchy Chart
CHttpServer::AddHeader

Other Resources

CHttpServerContext Members

CHttpServerContext::m_pECB

A pointer to an **EXTENSION_CONTROL_BLOCK** structure.

Remarks

This structure contains information describing the connection between the client which issued this server extension command and the server. See the **EXTENSION_CONTROL_BLOCK** structure for a description of the individual members.

See Also

Reference

CHttpServerContext Class

Hierarchy Chart

Other Resources

CHttpServerContext Members

CHttpServerContext::m_pStream

The **m_pStream** data member is the pointer to the initialized CHtmlStream, which your server can use to communicate with the client.

Remarks

Most extensions will write data to this stream as they do their work. MFC will write all of the data in this stream to the client when your function returns. If your function takes a long time to execute, you can use the WriteClient function to send data to the client immediately, even before your function ends.

See Also

Reference

CHttpServerContext Class
Hierarchy Chart
CHtmlStream Class

Other Resources

CHttpServerContext Members

CImageList Class

Provides the functionality of the Windows common image list control.

```
class CImageList : public CObject
```

Remarks

An "image list" is a collection of same-sized images, each of which can be referred to by its zero-based index. Image lists are used to efficiently manage large sets of icons or bitmaps. All images in an image list are contained in a single, wide bitmap in screen device format. An image list may also include a monochrome bitmap that contains masks used to draw images transparently (icon style). The Microsoft Win32 application programming interface (API) provides image list functions that enable you to draw images, create and destroy image lists, add and remove images, replace images, merge images, and drag images.

This control (and therefore the **CImageList** class) is available only to programs running under Windows 95/98 and Windows NT version 3.51 and later.

For more information on using **CImageList**, see Controls and Using CImageList.

Requirements

Header: afxcmn.h

See Also

Reference

COBJECT Class
Hierarchy Chart
CListCtrl Class
CTabCtrl Class

Other Resources

CImageList Members

CImageList Members

Base Class Members

Data Members

Construction

Attributes

Operations

Base Class Members

CObject Members

Data Members

m_hImageList	A handle containing the image list attached to this object.

Construction

CImageList	Constructs a **CImageList** object.
Create	Initializes an image list and attaches it to a **CImageList** object.

Attributes

DeleteTempMap	Called by the CWinApp idle-time handler to delete any temporary **CImageList** object created by **From Handle**.
FromHandle	Returns a pointer to a **CImageList** object when given a handle to an image list. If a **CImageList** object is not attached to the handle, a temporary **CImageList** object is created and attached.
FromHandlePermanent	Returns a pointer to a **CImageList** object when given a handle to an image list. If a **CImageList** object is not attached to the handle, **NULL** is returned.
GetBkColor	Retrieves the current background color for an image list.
GetImageCount	Retrieves the number of images in an image list.
GetImageInfo	Retrieves information about an image.
GetSafeHandle	Retrieves **m_hImageList**.
operator HIMAGELIST	Returns the **HIMAGELIST** attached to the **CImageList**.
SetBkColor	Sets the background color for an image list.

Operations

Add	Adds an image or images to an image list.
Attach	Attaches an image list to a **CImageList** object.
BeginDrag	Begins dragging an image.
Copy	Copies an image within a **CImageList** object.
DeleteImageList	Deletes an image list.

Detach	Detaches an image list object from a **CImageList** object and returns a handle to an image list.
DragEnter	Locks updates during a drag operation and displays the drag image at a specified position.
DragLeave	Unlocks the window and hides the drag image so that the window can be updated.
DragMove	Moves the image that is being dragged during a drag-and-drop operation.
DragShowNolock	Shows or hides the drag image during a drag operation, without locking the window.
Draw	Draws the image that is being dragged during a drag-and-drop operation.
DrawEx	Draws an image list item in the specified device context. The function uses the specified drawing style an d blends the image with the specified color.
DrawIndirect	Draws an image from an image list.
EndDrag	Ends a drag operation.
ExtractIcon	Creates an icon based on an image and mask in an image list.
GetDragImage	Gets the temporary image list that is used for dragging.
Read	Reads an image list from an archive.
Remove	Removes an image from an image list.
Replace	Replaces an image in an image list with a new image.
SetDragCursorImage	Creates a new drag image.
SetImageCount	Resets the count of images in an image list.
SetOverlayImage	Adds the zero-based index of an image to the list of images to be used as overlay masks.
Write	Writes an image list to an archive.

See Also

Reference

CImageList Class

Hierarchy Chart

Other Resources

CObject Members

CImageList Member Functions

For information about the member functions in **CImageList**, see CImageList Members.

CImageList::Add

Call this function to add one or more images or an icon to an image list.

```
int Add(
   CBitmap* pbmImage,
   CBitmap* pbmMask
);
int Add(
   CBitmap* pbmImage,
   COLORREF crMask
);
int Add(
   HICON hIcon
);
```

Parameters

pbmImage

Pointer to the bitmap containing the image or images. The number of images is inferred from the width of the bitmap.

pbmMask

Pointer to the bitmap containing the mask. If no mask is used with the image list, this parameter is ignored.

crMask

Color used to generate the mask. Each pixel of this color in the given bitmap is changed to black and the corresponding bit in the mask is set to one.

hIcon

Handle of the icon that contains the bitmap and mask for the new image.

Return Value

Zero-based index of the first new image if successful; otherwise – 1.

Example

```
// The pointer to my image list.
extern CImageList* pmyImageList;

// Add my icons.
pmyImageList->Add(AfxGetApp()->LoadIcon(IDI_ICON1));
pmyImageList->Add(AfxGetApp()->LoadIcon(IDI_ICON2));

// Add my bitmap, make all black pixels transparent.
CBitmap bm;
bm.LoadBitmap(IDB_BITMAP1);
pmyImageList->Add(&bm, RGB(0, 0, 0));
```

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::Remove
CImageList::Replace
COLORREF

Other Resources

CImageList Members

CImageList::Attach

Call this function to attach an image list to a **CImageList** object.

```
BOOL Attach(
   HIMAGELIST hImageList
);
```

Parameters

hImageList

A handle to an image list object.

Return Value

Nonzero if the attachment was successful; otherwise 0.

Example

```
// The pointer to my image list.
extern CImageList* pmyImageList;
// A handle to an image list that will be attached.
extern HIMAGELIST hmyImageList;

// Attach the image list handle to the CImageList object.
pmyImageList->Attach(hmyImageList);

// Add a new icon to the image list.
pmyImageList->Add(AfxGetApp()->LoadStandardIcon(IDI_QUESTION));

// Detach the handle from the CImageList object.
pmyImageList->Detach();
```

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::Detach
CImageList::GetSafeHandle

Other Resources

CImageList Members

CImageList::BeginDrag

Call this function to begin dragging an image.

```
BOOL BeginDrag(
   int nImage,
   CPoint ptHotSpot
);
```

Parameters

nImage

Zero-based index of the image to drag.

ptHotSpot

Coordinates of the starting drag position (typically, the cursor position). The coordinates are relative to the upper left corner of the image.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This function creates a temporary image list that is used for dragging. The image combines the specified image and its mask with the current cursor. In response to subsequent **WM_MOUSEMOVE** messages, you can move the drag image by using the **DragMove** member function. To end the drag operation, you can use the **EndDrag** member function.

Example

```
// The pointer to my image list.
extern CImageList* pmyImageList;
// A pointer to the window in which to drag the image.
extern CWnd* pmyWnd;
// The position of the cursor.
extern CPoint myPoint;

// Initialize the drag image (usually called from WM_LBUTTONDOWN).
pmyImageList->BeginDrag(0, CPoint(0, 0));
pmyImageList->DragEnter(pmyWnd, myPoint);
```

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::Draw
CImageList::EndDrag
CImageList::DragMove

Other Resources

CImageList Members

CImageList::CImageList

Constructs a **CImageList** object.

```
CImageList( );
```

See Also

Reference

CImageList Class

Hierarchy Chart

CImageList::Create

Other Resources

CImageList Members

CImageList::Copy

This member function implements the behavior of the Win32 function ImageList_Copy, as described in the *Platform SDK*.

```
BOOL Copy(
   int iDst,
   int iSrc,
   UINT uFlags = ILCF_MOVE
);
BOOL Copy(
   int iDst,
   CImageList* pSrc,
   int iSrc,
   UINT uFlags = ILCF_MOVE
);
```

Parameters

iDst

The zero-based index of the image to be used as the destination of the copy operation.

iSrc

The zero-based index of the image to be used as the source of the copy operation.

uFlags

The bit flag value that specifies the type of copy operation to be made. This parameter can be one of the following values:

Value	Meaning
ILCF_MOVE	The source image is copied to the destination image's index. This operation results in multiple instances of a given image. **ILCF_MOVE** is the default.
ILCF_SWAP	The source and destination images exchange positions within the image list.

pSrc

A pointer to a **CImageList** object that is the target of the copy operation.

Return Value

Nonzero if successful; otherwise zero.

Example

```
// The pointer to my image list.
extern CImageList* pmyImageList;
// The pointer to another image list.
extern CImageList* pmyImageList2;

// Copy the first image from pmyImageList2 and make it
// the first image of pmyImageList.
pmyImageList->Copy(0, pmyImageList2, 0, ILCF_MOVE);

// Recopy the new image of pmyImageList to make it also
// the last image in pmyImageList.
// Recopy the image to make it also the last image in pmyImageList.
pmyImageList->Copy(pmyImageList->GetImageCount()-1,
    (UINT) 0, ILCF_MOVE);
```

See Also

Reference

CImageList Class

Hierarchy Chart

Other Resources

CImageList Members

CImageList::Create

Initializes an image list and attaches it to a **CImageList** object.

```
BOOL Create(
   int cx,
   int cy,
   UINT nFlags,
   int nInitial,
   int nGrow
);
BOOL Create(
   UINT nBitmapID,
   int cx,
   int nGrow,
   COLORREF crMask
);
BOOL Create(
   LPCTSTR lpszBitmapID,
   int cx,
   int nGrow,
   COLORREF crMask
);
BOOL Create(
   CImageList& imagelist1,
   int nImage1,
   CImageList& imagelist2,
   int nImage2,
   int dx,
   int dy
);
BOOL Create(
   CImageList* pImageList
);
```

Parameters

cx

Dimensions of each image, in pixels.

cy

Dimensions of each image, in pixels.

nFlags

Specifies the type of image list to create. This parameter can be a combination of the following values, but it can include only one of the **ILC_COLOR** values.

Value	Meaning
ILC_COLOR	Use the default behavior if none of the other **ILC_COLOR*** flags is specified. Typically, the default is **ILC_COLOR4**; but for older display drivers, the default is **ILC_COLORDDB**.
ILC_COLOR4	Use a 4-bit (16 color) device-independent bitmap (DIB) section as the bitmap for the image list.
ILC_COLOR8	Use an 8-bit DIB section. The colors used for the color table are the same colors as the halftone palette.
ILC_COLOR16	Use a 16-bit (32/64k color) DIB section.

ILC_COLOR24	Use a 24-bit DIB section.
ILC_COLOR32	Use a 32-bit DIB section.
ILC_COLORDDB	Use a device-dependent bitmap.
ILC_MASK	Uses a mask. The image list contains two bitmaps, one of which is a monochrome bitmap used as a mask. If this value is not included, the image list contains only one bitmap.

nInitial

Number of images that the image list initially contains.

nGrow

Number of images by which the image list can grow when the system needs to resize the list to make room for new images. This parameter represents the number of new images the resized image list can contain.

nBitmapID

Resource IDs of the bitmap to be associated with the image list.

crMask

Color used to generate a mask. Each pixel of this color in the specified bitmap is changed to black, and the corresponding bit in the mask is set to one.

lpszBitmapID

A string containing the resource IDs of the images.

imagelist1

A reference to a **CImageList** object.

nImage1

Index of the first existing image.

imagelist2

A reference to a **CImageList** object.

nImage2

Index of the second existing image.

dx

Offset of the x-axis of the second image in relationship to the first image, in pixels.

dy

Offset of the y-axis of the second image in relationship to the first image, in pixels.

pImageList

A pointer to a **CImageList** object.

Return Value

Nonzero if successful; otherwise 0.

Remarks

You construct a **CImageList** in two steps. First, call the constructor and then call **Create**, which creates the image list and attaches it to the **CImageList** object.

Example

```
// The pointer to my image list.
extern CImageList* pmyImageList;

pmyImageList->Create(32, 32, ILC_COLOR8, 0, 4);
```

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::CImageList
COLORREF

Other Resources

CImageList Members

CImageList::DeleteImageList

Call this function to delete an image list.

```
BOOL DeleteImageList( );
```

Return Value

Nonzero if successful; otherwise 0.

Example

```
// The pointer to my image list.
extern CImageList* pmyImageList;

// Delete the image list and verify.
pmyImageList->DeleteImageList();
ASSERT(pmyImageList->GetSafeHandle() == NULL);
```

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::Detach

Other Resources

CImageList Members

CImageList::DeleteTempMap

Called automatically by the **CWinApp** idle-time handler, **DeleteTempMap** deletes any temporary **CImageList** objects created by FromHandle, but does not destroy any handles (*hImageList*) temporarily associated with the **ImageList** objects.

```
static void PASCAL DeleteTempMap( );
```

Example

```
// The following example is correct:
CImageList::DeleteTempMap();

// The following example is incorrect.
// DeleteTempMap() is a static member and cannot be called
// within the scope of an instantiated CImageList object.
pImageList->DeleteTempMap();  // Causes compiler error
```

See Also

Reference

CImageList Class

Hierarchy Chart

Other Resources

CImageList Members

CImageList::Detach

Call this function to detach an image list object from a **CImageList** object.

```
HIMAGELIST Detach( );
```

Return Value

A handle to an image list object.

Remarks

This function returns a handle to the image list object.

Example

See the example for CImageList::Attach.

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::Attach
CImageList::DeleteImageList

Other Resources

CImageList Members

CImageList::DragEnter

During a drag operation, locks updates to the window specified by *pWndLock* and displays the drag image at the position specified by *point*.

```
static BOOL PASCAL DragEnter(
   CWnd* pWndLock,
   CPoint point
);
```

Parameters

pWndLock

Pointer to the window that owns the drag image.

point

Position at which to display the drag image. Coordinates are relative to the upper left corner of the window (not the client area).

Return Value

Nonzero if successful; otherwise 0.

Remarks

The coordinates are relative to the window's upper left corner, so you must compensate for the widths of window elements, such as the border, title bar, and menu bar, when specifying the coordinates.

If *pWndLock* is **NULL**, this function draws the image in the display context associated with the desktop window, and coordinates are relative to the upper left corner of the screen.

This function locks all other updates to the given window during the drag operation. If you need to do any drawing during a drag operation, such as highlighting the target of a drag-and-drop operation, you can temporarily hide the dragged image by using the CImageList::DragLeave function.

Example

See the example for CImageList::BeginDrag.

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::BeginDrag
CImageList::EndDrag
CImageList::DragMove
CImageList::DragLeave

Other Resources

CImageList Members

CImageList::DragLeave

Unlocks the window specified by *pWndLock* and hides the drag image, allowing the window to be updated.

```
static BOOL PASCAL DragLeave(
   CWnd* pWndLock
);
```

Parameters

pWndLock

Pointer to the window that owns the drag image.

Return Value

Nonzero if successful; otherwise 0.

Example

See the example for CImageList::EndDrag.

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::BeginDrag
CImageList::EndDrag
CImageList::DragMove
CImageList::DragEnter

Other Resources

CImageList Members

CImageList::DragMove

Call this function to move the image that is being dragged during a drag-and-drop operation.

```
static BOOL PASCAL DragMove(
   CPoint pt
);
```

Parameters

pt

New drag position.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This function is typically called in response to a **WM_MOUSEMOVE** message. To begin a drag operation, use the **BeginDrag** member function.

Example

```
// The pointer to my image list.
extern CImageList* pmyImageList;
// The position of the cursor.
extern CPoint myPoint;

// Move the drag image to the next position (usually called
// from WM_MOUSEMOVE).
pmyImageList->DragMove(myPoint);
```

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::BeginDrag
CImageList::EndDrag
CImageList::Draw

Other Resources

CImageList Members

CImageList::DragShowNolock

Shows or hides the drag image during a drag operation, without locking the window.

```
static BOOL PASCAL DragShowNolock(
   BOOL bShow
);
```

Parameters

bShow

Specifies whether the drag image is to be shown.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The CImageList::DragEnter function locks all updates to the window during a drag operation. This function, however, does not lock the window.

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::BeginDrag
CImageList::EndDrag
CImageList::DragEnter
CImageList::DragLeave
CImageList::Draw

Other Resources

CImageList Members

CImageList::Draw

Call this function to draw the image that is being dragged during a drag-and-drop operation.

```
BOOL Draw(
   CDC* pDC,
   int nImage,
   POINT pt,
   UINT nStyle
);
```

Parameters

pDC

Pointer to the destination device context.

nImage

Zero-based index of the image to draw.

pt

Location at which to draw within the specified device context.

nStyle

Flag specifying the drawing style. It can be one or more of these values:

Value	Meaning
ILD_BLEND25, **ILD_FOCUS**	Draws the image, blending 25 percent with the system highlight color. This value has no effect if the image list does not contain a mask.
ILD_BLEND50, **ILD_SELECTED**, **ILD_BLEND**	Draws the image, blending 50 percent with the system highlight color. This value has no effect if the image list does not contain a mask.
ILD_MASK	Draws the mask.
ILD_NORMAL	Draws the image using the background color for the image list. If the background color is the **CLR_NONE** value, the image is drawn transparently using the mask.
ILD_TRANSPARENT	Draws the image transparently using the mask, regardless of the background color.

Return Value

Nonzero if successful; otherwise 0.

Example

See the example for CImageList::SetOverlayImage.

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::BeginDrag
CImageList::EndDrag
CImageList::DragMove
CImageList::DrawEx

Other Resources

CImageList Members

CImageList::DrawEx

Draws an image list item in the specified device context.

```
BOOL DrawEx(
   CDC* pDC,
   int nImage,
   POINT pt,
   SIZE sz,
   COLORREF clrBk,
   COLORREF clrFg,
   UINT nStyle
);
```

Parameters

pDC

Pointer to the destination device context.

nImage

Zero-based index of the image to draw.

pt

Location at which to draw within the specified device context.

sz

Size of the portion of the image to draw relative to the upper-left corner of the image. See *dx* and *dy* in ImageList_DrawEx in the *Platform SDK*.

clrBk

Background color of the image. See *rgbBk* in ImageList_DrawEx in the *Platform SDK*.

clrFg

Foreground color of the image. See *rgbFg* in ImageList_DrawEx in the *Platform SDK*.

nStyle

Flag specifying the drawing style. See *fStyle* in ImageList_DrawEx in the *Platform SDK*.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The function uses the specified drawing style and blends the image with the specified color.

Example

```
pmyImageList->DrawEx(pmyDC, 0, myPoint, mySize, CLR_DEFAULT, CLR_DEFAULT, ILD_IMAGE);
```

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::BeginDrag
CImageList::EndDrag
CImageList::DragMove
CImageList::Draw

Other Resources

CImageList Members

CImageList::DrawIndirect

Call this member function to draw an image from an image list.

```
BOOL DrawIndirect(
   IMAGELISTDRAWPARAMS* pimldp
);
BOOL DrawIndirect(
   CDC* pDC,
   int nImage,
   POINT pt,
   SIZE sz,
   POINT ptOrigin,
   UINT fStyle = ILD_NORMAL,
   DWORD dwRop = SRCCOPY,
   COLORREF rgbBack = CLR_DEFAULT,
   COLORREF rgbFore = CLR_DEFAULT,
   DWORD fState = ILS_NORMAL,
   DWORD Frame = 0,
   COLORREF crEffect = CLR_DEFAULT
);
```

Parameters

pimldp

A pointer to an IMAGELISTDRAWPARAMS structure that contains information about the draw operation.

pDC

A pointer to the destination device context. You must delete this CDC object when you are done with it.

nImage

The zero-based index of the image to be drawn.

pt

A POINT structure containing the x– and y– coordinates where the image will be drawn.

sz

A SIZE structure indicating the size of the image to be drawn.

ptOrigin

A POINT structure containing the x– and y–coordinates specifying the upper left corner of the drawing operation with respect to the image itself. Pixels of the image that are to the left of the x–coordinate and above the y–coordinate are not drawn.

fStyle

Flag specifying the drawing style and, optionally, the overlay image. See the Remarks section for information on the overlay image. The MFC default implementation, **ILD_NORMAL**, draws the image using the background color for the image list. If the background color is the **CLR_NONE** value, the image is drawn transparently using a mask.

Other possible styles are described under the **fStyle** member of the IMAGELISTDRAWPARAMS structure.

dwRop

Value specifying a raster-operation code. These codes define how the color data for the source rectangle will be combined with the color data for the destination rectangle to achieve the final color. MFC's default implementation, **SRCCOPY**, copies the source rectangle directly to the destination rectangle. This parameter is ignored if the *fStyle* parameter does not include the **ILD_ROP** flag.

Other possible values are described under the **dwRop** member of the IMAGELISTDRAWPARAMS structure.

rgbBack

The image background color, by default **CLR_DEFAULT**. This parameter can be an application-defined RGB value or one of

the following values:

Value	Meaning
CLR_DEFAULT	Default background color. The image is drawn using the image list background color.
CLR_NONE	No background color. The image is drawn transparently.

rgbFore

Image foreground color, by default **CLR_DEFAULT**. This parameter can be an application-defined RGB value or one of the following values:

Value	Meaning
CLR_DEFAULT	Default foreground color. The image is drawn using the system highlight color as the foreground color.
CLR_NONE	No blend color. The image is blended with the color of the destination device context.

This parameter is used only if *fStyle* includes the **ILD_BLEND25** or **ILD_BLEND50** flag.

fState

Flag specifying the drawing state. This member can contain one or more image list state flags.

Frame

Affects the behavior of saturate and alpha-blending effects.

When used with **ILS_SATURATE**, this member holds the value that is added to each color component of the RGB triplet for each pixel in the icon.

When used with **ILS_APLHA**, this member holds the value for the alpha channel. This value can be from 0 to 255, with 0 being completely transparent, and 255 being completely opaque.

crEffect

A COLORREF value used for glow and shadow effects.

Return Value

TRUE if the image is successfully drawn; otherwise **FALSE**.

Remarks

Use the first version if you want to fill the Win32 structure yourself. Use the second version if you want to take advantage of one or more of MFC's default arguments, or avoid managing the structure.

An overlay image is an image that is drawn on top of the primary image, specified in this member function by the *nImage* parameter. Draw an overlay mask by using the Draw member function with the one-based index of the overlay mask specified by using the INDEXTOOVERLAYMASK macro.

Example

```
// The pointer to my image list.
extern CImageList* pmyImageList;
// The pointer to the DC for drawing.
extern CDC* pmyDC;

int i, dx, cx, cy, nCount = pmyImageList->GetImageCount();

::ImageList_GetIconSize(*pmyImageList, &cx, &cy);

// Draw the images of the image list on the DC.
for (dx=0,i=0;i < nCount;i++)
{
   pmyImageList->DrawIndirect(pmyDC, i, CPoint(dx, 0),
      CSize(cx, cy), CPoint(0, 0));
   dx += cx;
}
```

See Also

Reference

CImageList Class

Hierarchy Chart

CImageList::SetOverlayImage

Other Resources

CImageList Members

CImageList::EndDrag

Call this function to end a drag operation.

```
static void PASCAL EndDrag( );
```

Remarks

To begin a drag operation, use the **BeginDrag** member function.

Example

```
// The pointer to my image list.
extern CImageList* pmyImageList;
// A pointer to the window in which to drag the image.
extern CWnd* pmyWnd;

// Terminate the drag image (usually called from WM_LBUTTONUP).
pmyImageList->DragLeave(pmyWnd);
pmyImageList->EndDrag();
```

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::BeginDrag
CImageList::Draw
CImageList::DragMove

Other Resources

CImageList Members

CImageList::ExtractIcon

Call this function to create an icon based on an image and its related mask in an image list.

```
HICON ExtractIcon(
   int nImage
);
```

Parameters

nImage

Zero-based index of the image.

Return Value

Handle of the icon if successful; otherwise **NULL**.

Remarks

This method relies on the behavior of the ImageList_ExtractIcon macro to create the icon. Refer to the ImageList_ExtractIcon macro for more information on icon creation and cleanup.

Example

```
// The pointer to my image list.
extern CImageList* pmyImageList;
extern CDC* pmyDC;

int   i, dx, cx, cy, nCount = pmyImageList->GetImageCount();
HICON hIcon;

::ImageList_GetIconSize(*pmyImageList, &cx, &cy);

// Draw the images of the image list on the DC.
for (dx=0,i=0;i < nCount;i++)
{
   hIcon = pmyImageList->ExtractIcon(i);

   pmyDC->DrawIcon(dx, 0, hIcon);
   dx += cx;
}
```

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::Replace

Other Resources

CImageList Members

CImageList::FromHandle

Returns a pointer to a **CImageList** object when given a handle to an image list.

```
static CImageList* PASCAL FromHandle(
   HIMAGELIST hImageList
);
```

Parameters

hImageList

Specifies the image list.

Return Value

A pointer to a **CImageList** object if successful; otherwise **NULL**.

Remarks

If a **CImageList** is not already attached to the handle, a temporary **CImageList** object is created and attached. This temporary **CImageList** object is valid only until the next time the application has idle time in its event loop, at which time all temporary objects are deleted.

Example

```
// The handle of my image list.
extern HIMAGELIST hmyImageList;

// Convert the HIMAGELIST to a CImageList*.
ASSERT(hmyImageList != NULL);
CImageList* pmyImageList = CImageList::FromHandle(hmyImageList);
ASSERT(pmyImageList != NULL);
```

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::FromHandlePermanent

Other Resources

CImageList Members

CImageList::FromHandlePermanent

Returns a pointer to a **CImageList** object when given a handle to an image list.

```
static CImageList* PASCAL FromHandlePermanent(
   HIMAGELIST hImageList
);
```

Parameters

hImageList

Specifies the image list.

Return Value

A pointer to a **CImageList** object if successful; otherwise **NULL**.

Remarks

If a **CImageList** object is not attached to the handle, **NULL** is returned.

Example

```
// The handle of my image list.
extern HIMAGELIST hmyImageList;

// Convert the HIMAGELIST to a CImageList*.
ASSERT(hmyImageList != NULL);
CImageList* pmyImageList = CImageList::FromHandlePermanent(hmyImageList);
ASSERT(pmyImageList != NULL);
```

See Also

Reference

CImageList Class

Hierarchy Chart

CImageList::FromHandle

Other Resources

CImageList Members

CImageList::GetBkColor

Call this function to retrieve the current background color for an image list.

```
COLORREF GetBkColor( ) const;
```

Return Value

The RGB color value of the **CImageList** object background color.

Example

See the example for CImageList::SetBkColor.

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::SetBkColor
COLORREF

Other Resources

CImageList Members

CImageList::GetDragImage

Gets the temporary image list that is used for dragging.

```
static CImageList* PASCAL GetDragImage(
   LPPOINT lpPoint,
   LPPOINT lpPointHotSpot
);
```

Parameters

lpPoint

Address of a POINT structure that receives the current drag position.

lpPointHotSpot

Address of a **POINT** structure that receives the offset of the drag image relative to the drag position.

Return Value

If successful, a pointer to the temporary image list that is used for dragging; otherwise, **NULL**.

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::SetDragCursorImage

Other Resources

CImageList Members

CImageList::GetImageCount

Call this function to retrieve the number of images in an image list.

```
int GetImageCount( ) const;
```

Return Value

The number of images.

Example

See the example for CImageList::ExtractIcon.

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::GetImageInfo

Other Resources

CImageList Members

CImageList::GetImageInfo

Call this function to retrieve information about an image.

```
BOOL GetImageInfo(
   int nImage,
   IMAGEINFO* pImageInfo
) const;
```

Parameters

nImage

Zero-based index of the image.

pImageInfo

Pointer to an IMAGEINFO structure that receives information about the image. The information in this structure can be used to directly manipulate the bitmaps for the image.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The **IMAGEINFO** structure contains information about an image in an image list.

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::GetImageCount

Other Resources

CImageList Members

CImageList::GetSafeHandle

Call this function to retrieve the **m_hImageList** data member.

```
HIMAGELIST GetSafeHandle( ) const;
```

Return Value

A handle to the attached image list; otherwise **NULL** if no object is attached.

Example

```
// The pointer to my image list.
extern CImageList* pmyImageList;

// Get the safe handle to the image list.
HIMAGELIST hImageList = pmyImageList->GetSafeHandle();
```

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::Attach
CImageList::Detach
CImageList::m_hImageList

Other Resources

CImageList Members

CImageList::operator HIMAGELIST

Use this operator to get the attached handle of the **CImageList** object.

```
operator HIMAGELIST( ) const;
```

Return Value

If successful, a handle to the image list represented by the **CImageList** object; otherwise **NULL**.

Remarks

This operator is a casting operator, which supports direct use of an **HIMAGELIST** object.

Example

```
// The pointer to my image list.
extern CImageList* pmyImageList;

// Get the safe handle to the image list.
HIMAGELIST hImageList = *pmyImageList;
```

See Also

Reference

CImageList Class

Hierarchy Chart

Other Resources

CImageList Members

CImageList::Read

Call this function to read an image list from an archive.

```
BOOL Read(
   CArchive* pArchive
);
```

Parameters

pArchive

A pointer to a **CArchive** object from which the image list is to be read.

Return Value

Nonzero if successful; otherwise 0.

Example

```
// The pointer to my image list.
extern CImageList* pmyImageList;

// Open the archive to load the image list from.
CFile   myFile(TEXT("myfile.data"), CFile::modeRead);
CArchive ar(&myFile, CArchive::load);

// Load the image list from the archive.
pmyImageList->Read(&ar);
```

See Also

Reference

CImageList Class

Hierarchy Chart

CImageList::Write

Other Resources

CImageList Members

CImageList::Remove

Call this function to remove an image from an image list object.

```
BOOL Remove(
   int nImage
);
```

Parameters

nImage

Zero-based index of the image to remove.

Return Value

Nonzero if successful; otherwise 0.

Remarks

All items following *nImage* now move down one position. For example, if an image list contains two items, deleting the first item will cause the remaining item to now be in the first position. *nImage*=0 for the item in the first position.

Example

```
// The pointer to my image list.
extern CImageList* pmyImageList;

// Remove every other image from the image list.
for (int i=0;i < pmyImageList->GetImageCount();i++)
{
   pmyImageList->Remove(i);
}
```

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::DeleteImageList

Other Resources

CImageList Members

CImageList::Replace

Call this function to replace an image in an image list with a new image.

```
BOOL Replace(
   int nImage,
   CBitmap* pbmImage,
   CBitmap* pbmMask
);
int Replace(
   int nImage,
   HICON hIcon
);
```

Parameters

nImage

Zero-based index of the image to replace.

pbmImage

A pointer to the bitmap containing the image.

pbmMask

A pointer to the bitmap containing the mask. If no mask is used with the image list, this parameter is ignored.

hIcon

A handle to the icon that contains the bitmap and mask for the new image.

Return Value

The version returning **BOOL** returns nonzero if successful; otherwise 0.

The version returning **int** returns the zero-based index of the image if successful; otherwise – 1.

Remarks

Call this member function after calling SetImageCount to assign the new, valid images to the placeholder image index numbers.

Example

See the example for CImageList::SetImageCount.

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::Remove

Other Resources

CImageList Members

CImageList::SetBkColor

Call this function to set the background color for an image list.

```
COLORREF SetBkColor(
   COLORREF cr
);
```

Parameters

cr

Background color to set. It can be **CLR_NONE**. In that case, images are drawn transparently using the mask.

Return Value

The previous background color if successful; otherwise **CLR_NONE**.

Example

```
// The pointer to my image list.
extern CImageList* pmyImageList;

// Set the background color to white.
pmyImageList->SetBkColor(RGB(255, 255, 255));
ASSERT(pmyImageList->GetBkColor() == RGB(255, 255, 255));
```

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::GetBkColor
COLORREF

Other Resources

CImageList Members

CImageList::SetDragCursorImage

Creates a new drag image by combining the given image (typically a mouse cursor image) with the current drag image.

```
BOOL SetDragCursorImage(
   int nDrag,
   CPoint ptHotSpot
);
```

Parameters

nDrag

Index of the new image to be combined with the drag image.

ptHotSpot

Position of the hot spot within the new image.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Because the dragging functions use the new image during a drag operation, you should use the Windows ShowCursor function to hide the actual mouse cursor after calling **CImageList::SetDragCursorImage**. Otherwise, the system may appear to have two mouse cursors for the duration of the drag operation.

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::BeginDrag
CImageList::EndDrag
CImageList::GetDragImage

Other Resources

CImageList Members

CImageList::SetImageCount

Call this member function to reset the number of images in a **CImageList** object.

```
BOOL SetImageCount(
   UINT uNewCount
);
```

Parameters

uNewCount

The value specifying the new total number of images in the image list.

Return Value

Nonzero if successful; otherwise zero.

Remarks

If you call this member function to increase the number of images in the image list, then call Replace for each additional image to assign the new indexes to valid images. If you fail to assign the indexes to valid images, draw operations that create the new images will be unpredictable.

If you decrease the size of an image list by using this function, the truncated images are freed.

Example

```
// The pointer to my image list.
extern CImageList* pmyImageList;

// Set the image count of the image list to be 10 with
// all images being the system question mark icon.
pmyImageList->SetImageCount(10);
HICON hIcon = AfxGetApp()->LoadStandardIcon(IDI_QUESTION);

for (int i=0;i < 10;i++)
{
   pmyImageList->Replace(i, hIcon);
}
```

See Also

Reference

CImageList Class

Hierarchy Chart

Other Resources

CImageList Members

CImageList::SetOverlayImage

Call this function to add the zero-based index of an image to the list of images to be used as overlay masks.

```
BOOL SetOverlayImage(
   int nImage,
   int nOverlay
);
```

Parameters

nImage

Zero-based index of the image to use as an overlay mask.

nOverlay

One-based index of the overlay mask.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Up to four indices can be added to the list.

An overlay mask is an image drawn transparently over another image. Draw an overlay mask over an image by using the CImageList::Draw member function with the one-based index of the overlay mask specified by using the **INDEXTOOVERLAYMASK** macro.

Example

```
// The pointer to my image list.
extern CImageList* pmyImageList;
// The position of where to draw the image.
extern CPoint myPoint;
// The DC to draw the image on.
extern CDC* pmyDC;

// Add a new image to the image list.
int nIndex = pmyImageList->Add(AfxGetApp()->LoadStandardIcon(IDI_QUESTION));

if (nIndex != -1)
{
   // Make the new image an overlay image.
   pmyImageList->SetOverlayImage(nIndex, 1);

   // Draw the first image in the image list with an overlay image.
   pmyImageList->Draw(pmyDC, 0, myPoint, INDEXTOOVERLAYMASK(1));
}
```

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::Add

Other Resources

CImageList Members

CImageList::Write

Call this function to write an image list object to an archive.

```
BOOL Write(
   CArchive* pArchive
);
```

Parameters

pArchive

A pointer to a **CArchive** object in which the image list is to be stored.

Return Value

Nonzero if successful; otherwise 0.

Example

```
// The pointer to my image list.
extern CImageList* pmyImageList;

// Open the archive to store the image list in.
CFile   myFile(TEXT("myfile.data"), CFile::modeCreate | CFile::modeWrite);
CArchive ar(&myFile, CArchive::store);

// Store the image list in the archive.
pmyImageList->Write(&ar);
```

See Also

Reference

CImageList Class

Hierarchy Chart

CImageList::Read

Other Resources

CImageList Members

CImageList Data Members

For information about the data members in **CImageList**, see CImageList Members.

CImageList::m_hImageList

A handle of the image list attached to this object.

HIMAGELIST m_hImageList;

Remarks

The **m_hImageList** data member is a public variable of type **HIMAGELIST**.

Example

```
// The pointer to my image list.
extern CImageList* pmyImageList;

// Get the safe handle to the image list.
HIMAGELIST hImageList = pmyImageList->m_hImageList;
```

See Also

Reference

CImageList Class
Hierarchy Chart
CImageList::Attach
CImageList::Detach

Other Resources

CImageList Members

CInternetConnection Class

This topic has been updated for Visual Studio 2005 SP1.

Manages your connection to an Internet server.

```
class CInternetConnection : public CObject
```

Remarks

It is the base class for MFC classes CFtpConnection, CHttpConnection, and CGopherConnection. Each of these classes provides additional functionality for communicating with the respective FTP, HTTP, or gopher server.

To communicate directly with an Internet server, you must have a CInternetSession object and a **CInternetConnection** object.

To learn more about how the WinInet classes work, see the article Internet Programming with WinInet.

Requirements

Header: afxinet.h

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

See Also

Reference

CObject Class

Hierarchy Chart

Other Resources

CInternetConnection Members

CInternetConnection Members

Base Class Members

CObject Members

Construction

CInternetConnection	Constructs a **CInternetConnection** object.

Operations

GetContext	Gets the context ID for this connection object.
GetServerName	Gets the name of the server associated with the connection.
GetSession	Gets a pointer to the CInternetSession object associated with the connection.

Operators

operator HINTERNET	A handle to an Internet session.

See Also

Reference

CInternetConnection Class

Hierarchy Chart

CInternetConnection Member Functions

For information about the member functions in **CInternetConnection**, see CInternetConnection Members.

CInternetConnection::CInternetConnection

This member function is called when a **CInternetConnection** object is created.

```
CInternetConnection(
   CInternetSession* pSession,
   LPCTSTR pstrServer,
   INTERNET_PORT nPort = INTERNET_INVALID_PORT_NUMBER,
   DWORD_PTR dwContext = 1
);
```

Parameters

pSession

A pointer to a CInternetSession object.

pstrServer

A pointer to a string containing the server name.

nPort

The number that identifies the Internet port for this connection.

dwContext

The context identifier for the **CInternetConnection** object. See **Remarks** for more information about *dwContext*.

Remarks

You never call **CInternetConnection** yourself; instead, call the CInternetSession member function for the type of connection you want to establish:

- CInternetSession::GetFtpConnection
- CInternetSession::GetHttpConnection
- CInternetSession::GetGopherConnection

The default value for *dwContext* is sent by MFC to the **CInternetConnection**-derived object from the CInternetSession object that created the **InternetConnection**-derived object. The default is set to 1; however, you can explicitly assign a specific context identifier in the CInternetSession constructor for the connection. The object and any work it does will be associated with that context ID. The context identifier is returned to CInternetSession::OnStatusCallback to provide status on the object with which it is identified. See the article Internet First Steps: WinInet for more information about the context identifier.

See Also

Reference

CInternetConnection Class
Hierarchy Chart
CInternetSession Class
CGopherConnection Class
CFtpConnection Class
CHttpConnection Class

Other Resources

CInternetConnection Members

CInternetConnection::GetContext

Call this member function to get the context ID for this session.

```
DWORD_PTR GetContext( ) const;
```

Return Value

The application-assigned context ID.

Remarks

The context ID is originally specified in CInternetSession and propagates to **CInternetConnection**- and CInternetFile-derived classes, unless specified differently in the call to a function that opens the connection. The context ID is associated with any operation of the given object and identifies the operation's status information returned by CInternetSession::OnStatusCallback.

For more information about how **GetContext** works with other WinInet classes to give the user status information, see the article Internet First Steps: WinInet for more information about the context identifier.

See Also

Reference

CInternetConnection Class
Hierarchy Chart
CInternetSession::EnableStatusCallback

Other Resources

CInternetConnection Members

CInternetConnection::GetServerName

Call this member function to get the name of the server associated with this Internet connection.

```
CString GetServerName( ) const;
```

Return Value

The name of the server this connection object is working with.

See Also

Reference

CInternetConnection Class
Hierarchy Chart
CInternetSession Class
CGopherConnection Class
CFtpConnection Class
CHttpConnection Class

Other Resources

CInternetConnection Members

CInternetConnection::GetSession

Call this member function to get a pointer to the **CInternetSession** object that's associated with this connection.

```
CInternetSession* GetSession( ) const;
```

Return Value

A pointer to a CInternetSession object associated with this Internet connection object.

See Also

Reference

CInternetConnection Class
Hierarchy Chart
CInternetSession Class
CGopherConnection Class
CFtpConnection Class
CHttpConnection Class

Other Resources

CInternetConnection Members

CInternetConnection Operators

For information about the operators in **CInternetConnection**, see CInternetConnection Members.

CInternetConnection::operator HINTERNET

Use this operator to get the API-level handle for the current Internet session.

```
operator HINTERNET( ) const;
```

See Also

Reference

CInternetConnection Class
Hierarchy Chart
CInternetSession Class
CGopherConnection Class
CFtpConnection Class
CHttpConnection Class

Other Resources

CInternetConnection Members

CInternetException Class

This topic has been updated for Visual Studio 2005 SP1.

Represents an exception condition related to an Internet operation.

```
class CInternetException : public CException
```

Remarks

The **CInternetException** class includes two public data members: one holds the error code associated with the exception, and the other holds the context identifier of the Internet application associated with the error.

For more information about context identifiers for Internet applications, see the article Internet Programming with WinInet.

Requirements

Header: afxinet.h

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

See Also

Reference

CException Class

Hierarchy Chart

CException Class

Other Resources

CInternetException Members

CInternetException Members

Base Class Members

CObject Members

CException Members

Construction

CInternetException	Constructs a **CInternetException** object.

Data Members

m_dwContext	The context value associated with the operation that caused the exception.
m_dwError	The error that caused the exception.

See Also

Reference

CInternetException Class
Hierarchy Chart
CException Class

CInternetException Member Functions

For information about the member functions in **CInternetException**, see CInternetException Members.

CInternetException::CInternetException

This member function is called when a **CInternetException** object is created.

```
CInternetException(
   DWORD dwError
);
```

Parameters

dwError

The error that caused the exception.

Remarks

To throw a CInternetException, call the MFC global function AfxThrowInternetException.

See Also

Reference

CInternetException Class
Hierarchy Chart
CException Class

Other Resources

CInternetException Members

CInternetException Data Members

For information about the data members in **CInternetException**, see CInternetException Members.

CInternetException::m_dwContext

The context value associated with the related Internet operation.

```
DWORD_PTR m_dwContext;
```

Remarks

The context identifier is originally specified in CInternetSession and passed by MFC to CInternetConnection- and CInternetFile-derived classes. You can override this default and assign any *dwContext* parameter a value of your choosing. *dwContext* is associated with any operation of the given object. *dwContext* identifies the operation's status information returned by CInternetSession::OnStatusCallback.

See Also

Reference

CInternetException Class
Hierarchy Chart
CException Class

Other Resources

CInternetException Members

CInternetException::m_dwError

The error that caused the exception.

```
DWORD m_dwError;
```

Remarks

This error value may be a system error code, found in WINERROR.H, or an error value from WININET.H.

For a list of Win32 error codes, see Error Codes. For a list of Internet-specific error messages, see Error Messages and Error Pages. Both topics are in the *Platform SDK*.

See Also

Reference

CInternetException Class
Hierarchy Chart
CException Class

Other Resources

CInternetException Members

CInternetFile Class

This topic has been updated for Visual Studio 2005 SP1.

Allows access to files on remote systems that use Internet protocols.

```
class CInternetFile : public CStdioFile
```

Remarks

Provides a base class for the CHttpFile and CGopherFile file classes. You never create a **CInternetFile** object directly. Instead, create an object of one of its derived classes by calling CGopherConnection::OpenFile or CHttpConnection::OpenRequest. You also can create a **CInternetFile** object by calling CFtpConnection::OpenFile.

The **CInternetFile** member functions **Open**, **LockRange**, **UnlockRange**, and **Duplicate** are not implemented for **CInternetFile**. If you call these functions on a **CInternetFile** object, you will get a CNotSupportedException.

To learn more about how **CInternetFile** works with the other MFC Internet classes, see the article Internet Programming with WinInet.

Requirements

Header: afxinet.h

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

The Smart Device version of CInternetFile Class does not support the following methods:

- CInternetFile::SetWriteBufferSize
- CInternetFile::Seek
- CInternetFile::Write
- CInternetFile::Flush
- CInternetFile::WriteString

See Also

Reference

CStdioFile Class
Hierarchy Chart
CInternetConnection Class

Concepts

CInternetFile Members

CInternetFile Members

This topic has been updated for Visual Studio 2005 SP1.

Base Class Members

CObject Members

CFile Members

CStdioFile Members

Construction

CInternetFile	Constructs a **CInternetFile** object.

Operations

SetReadBufferSize	Sets the size of the buffer where data will be read.
SetWriteBufferSize	Sets the size of the buffer where data will be written.

Overridables

Abort	Closes the file, ignoring all warnings and errors.
Close	Closes a **CInternetFile** and frees its resources.
Flush	Flushes the contents of the write buffer and makes sure the data in memory is written to the target machine.
GetLength	Returns the size of the file.
Read	Reads the number of specified bytes.
ReadString	Reads a stream of characters.
Seek	Repositions the pointer in an open file.
Write	Writes the number of specified bytes.
WriteString	Writes a null-terminated string to a file.

Data Members

m_hFile	A handle to a file.

Operators

operator HINTERNET	A casting operator for an Internet handle.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

The Smart Device version of CInternetFile Class does not support the following methods:

- CInternetFile::SetWriteBufferSize
- CInternetFile::Seek
- CInternetFile::Write

- CInternetFile::Flush
- CInternetFile::WriteString

See Also

Reference

CInternetFile Class
Hierarchy Chart

CInternetFile Member Functions

For information about the member functions in **CInternetFile**, see CInternetFile Members.

CInternetFile::Abort

Closes the file associated with this object and makes the file unavailable for reading or writing.

```
virtual void Abort( );
```

Remarks

If you have not closed the file before destroying the object, the destructor closes it for you.

When handling exceptions, **Abort** differs from Close in two important ways. First, the **Abort** function does not throw an exception on failures because it ignores failures. Second, **Abort** does not **ASSERT** if the file has not been opened or was closed previously.

See Also

Reference

CInternetFile Class

Hierarchy Chart

Concepts

CInternetFile Members

CInternetFile::CInternetFile

This member function is called when a **CInternetFile** object is created.

```
CInternetFile(
   HINTERNET hFile,
   LPCTSTR pstrFileName,
   CInternetConnection* pConnection,
   BOOL bReadMode
);
CInternetFile(
   HINTERNET hFile,
   HINTERNET hSession,
   LPCTSTR pstrFileName,
   LPCTSTR pstrServer,
   DWORD_PTR dwContext,
   BOOL bReadMode
);
```

Parameters

hFile

A handle to an Internet file.

pstrFileName

A pointer to a string containing the file name.

pConnection

A pointer to a CInternetConnection object.

bReadMode

Indicates whether the file is read-only.

hSession

A handle to an Internet session.

pstrServer

A pointer to a string containing the name of the server.

dwContext

The context identifier for the **CInternetFile** object. See WinInet Basics for more information about the context identifier.

Remarks

You never create a **CInternetFile** object directly. Instead, create an object of one of its derived classes by calling CGopherConnection::OpenFile or CHttpConnection::OpenRequest. You also can create a **CInternetFile** object by calling CFtpConnection::OpenFile.

See Also

Reference

CInternetFile Class
Hierarchy Chart
CInternetConnection Class
CHttpFile Class
CGopherFile Class

Concepts

CInternetFile Members

CInternetFile::Close

Closes a **CInternetFile** and frees any of its resources.

```
virtual void Close( );
```

Remarks

If the file was opened for writing, there is an implicit call to Flush to assure that all buffered data is written to the host. You should call **Close** when you are finished using a file.

Exceptions

This method can throw exceptions of type **CInternetException***.

See Also

Reference

CInternetFile Class

Hierarchy Chart

Concepts

CInternetFile Members

CInternetFile::Flush

This topic has been updated for Visual Studio 2005 SP1.

Call this member function to flush the contents of the write buffer.

```
virtual void Flush( );
```

Remarks

Use **Flush** to assure that all data in memory has actually been written to the target machine and to assure your transaction with the host machine has been completed. **Flush** is only effective on **CInternetFile** objects opened for writing.

Exceptions

This method can throw exceptions of type **CInternetException***.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CInternetFile Class

Hierarchy Chart

Concepts

CInternetFile Members

CInternetFile::GetLength

Returns the size of the file.

```
virtual ULONGLONG GetLength( ) const;
```

See Also

Reference

CInternetFile Class

Hierarchy Chart

Concepts

CInternetFile Members

CInternetFile::Read

Call this member function to read into the given memory, starting at *lpvBuf*, the specified number of bytes, *nCount*.

```
virtual UINT Read(
   void* lpBuf,
   UINT nCount
);
```

Parameters

lpBuf

A pointer to a memory address to which file data is read.

nCount

The number of bytes to be written.

Return Value

The number of bytes transferred to the buffer. The return value may be less than *nCount* if the end of file was reached.

Remarks

The function returns the number of bytes actually read — a number that may be less than *nCount* if the file ends. If an error occurs while reading the file, the function throws a CInternetException object that describes the error. Note that reading past the end of the file is not considered an error and no exception will be thrown.

Exceptions

This method can throw exceptions of type **CInternetException***.

See Also

Reference

CInternetFile Class

Hierarchy Chart

Concepts

CInternetFile Members

CInternetFile::ReadString

Call this member function to read a stream of characters until it finds a newline character.

```
virtual BOOL ReadString(
   CString& rString
);
virtual LPTSTR ReadString(
   LPTSTR pstr,
   UINT nMax
);
```

Parameters

pstr

A pointer to a string which will receive the line being read.

nMax

The maximum number of characters to be read.

rString

A reference to the CString object that receives the read line.

Return Value

A pointer to the buffer containing plain data retrieved from the CInternetFile object. Regardless of the data type of the buffer passed to this method, it does not perform any manipulations on the data (for example, conversion to Unicode), so you must map the returned data to the structure you expect, as if the **void*** type were returned.

NULL if end-of-file was reached without reading any data; or, if boolean, **FALSE** if end-of-file was reached without reading any data.

Remarks

The function places the resulting line into the memory referenced by the *pstr* parameter. It stops reading characters when it reaches the maximum number of characters, specified by *nMax*. The buffer always receives a terminating null character.

If you call **ReadString** without first calling SetReadBufferSize, you will get a buffer of 4096 bytes.

Exceptions

This method can throw exceptions of type **CInternetException***.

See Also

Reference

CInternetFile Class

Hierarchy Chart

Concepts

CInternetFile Members

CInternetFile::Seek

This topic has been updated for Visual Studio 2005 SP1.

Call this member function to reposition the pointer in a previously opened file.

```
virtual ULONGLONG Seek(
   LONGLONG lOffset,
   UINT nFrom
);
```

Parameters

lOffset

Offset in bytes to move the read/write pointer in the file.

nFrom

Relative reference for the offset. Must be one of the following values:

- **CFile::begin** Move the file pointer *lOff* bytes forward from the beginning of the file.
- **CFile::current** Move the file pointer *lOff* bytes from the current position in the file.
- **CFile::end** Move the file pointer *lOff* bytes from the end of the file. *lOff* must be negative to seek into the existing file; positive values will seek past the end of the file.

Return Value

The new byte offset from the beginning of the file if the requested position is legal; otherwise, the value is undefined and a CInternetException object is thrown.

Remarks

The **Seek** function permits random access to a file's contents by moving the pointer a specified amount, absolutely or relatively. No data is actually read during the seek.

At this time, a call to this member function is only supported for data associated with **CHttpFile** objects. It is not supported for FTP or gopher requests. If you call **Seek** for one of these unsupported services, it will pass back you to the Win32 error code **ERROR_INTERNET_INVALID_OPERATION**.

When a file is opened, the file pointer is at offset 0, the beginning of the file.

Note
Using **Seek** may cause an implicit call to Flush.

Exceptions

This method can throw exceptions of type **CInternetException***.

Example

See the example for the base class implementation (CFile::Seek).

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CInternetFile Class

Hierarchy Chart

Concepts

CInternetFile Members

CInternetFile::SetReadBufferSize

Call this member function to set the size of the temporary read buffer used by a **CInternetFile**-derived object.

```
BOOL SetReadBufferSize(
  UINT nReadSize
);
```

Parameters

nReadSize

The desired buffer size in bytes.

Return Value

Nonzero if successful; otherwise 0. If the call fails, the Win32 function GetLastError may be called to determine the cause of the error.

Remarks

The underlying WinInet APIs do not perform buffering, so choose a buffer size that allows your application to read data efficiently, regardless of the amount of data to be read. If each call to Read normally involves a large aount of data (for example, four or more kilobytes), you should not need a buffer. However, if you call **Read** to get small chunks of data, or if you use ReadString to read individual lines at a time, then a read buffer improves application performance.

By default, a **CInternetFile** object does not provide any buffering for reading. If you call this member function, you must be sure that the file has been opened for read access.

You can increase the buffer size at any time, but shrinking the buffer will have no effect. If you call ReadString without first calling **SetReadBufferSize**, you will get a buffer of 4096 bytes.

See Also

Reference

CInternetFile Class

Hierarchy Chart

Concepts

CInternetFile Members

CInternetFile::SetWriteBufferSize

This topic has been updated for Visual Studio 2005 SP1.

Call this member function to set the size of the temporary write buffer used by a **CInternetFile**-derived object.

```
BOOL SetWriteBufferSize(
  UINT nWriteSize
);
```

Parameters

nWriteSize

The size of the buffer in bytes.

Return Value

Nonzero if successful; otherwise 0. If the call fails, the Win32 function GetLastError may be called to determine the cause of the error.

Remarks

The underlying WinInet APIs don't perform buffering, so choose a buffer size that allows your application to write data efficiently regardless of the amount of data to be written. If each call to Write normally involves a large amount of data (for example, four or more kilobytes at a time), you should not need a buffer. However, if you call Write to write small chunks of data, a write buffer improves your application's performance.

By default, a **CInternetFile** object does not provide any buffering for writing. If you call this member function, you must be sure that the file has been opened for write access. You can change the size of the write buffer at any time, but doing so causes an implicit call to Flush.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CInternetFile Class

Hierarchy Chart

Concepts

CInternetFile Members

CInternetFile::Write

This topic has been updated for Visual Studio 2005 SP1.

Call this member function to write into the given memory, *lpvBuf*, the specified number of bytes, *nCount*.

```
virtual void Write(
   const void* lpBuf,
   UINT nCount
);
```

Parameters

lpBuf

A pointer to the first byte to be written.

nCount

Specifies the number of bytes to be written.

Remarks

If any error occurs while writing the data, the function throws a CInternetException object describing the error.

Exceptions

This method can throw exceptions of type **CInternetException**.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CInternetFile Class

Hierarchy Chart

Concepts

CInternetFile Members

CInternetFile::WriteString

This topic has been updated for Visual Studio 2005 SP1.

This function writes a null-terminated string to the associated file.

```
virtual void WriteString(
   LPCTSTR pstr
);
```

Parameters

pstr

A pointer to a string containing the contents to be written.

Remarks

If any error occurs while writing the data, the function throws a CInternetException object describing the error.

Exceptions

This method can throw exceptions of type **CInternetException***.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CInternetFile Class

Hierarchy Chart

Concepts

CInternetFile Members

CInternetFile Data Members

For information about the data members in **CInternetFile**, see CInternetFile Members.

CInternetFile::m_hFile

A handle to the file associated with this object.

```
HINTERNET m_hFile;
```

See Also

Reference

CInternetFile Class

Hierarchy Chart

Concepts

CInternetFile Members

CInternetFile Operators

For information about the operators in **CInternetFile**, see CInternetFile Members.

CInternetFile::operator HINTERNET

Use this operator to get the Windows handle for the current Internet session.

```
operator HINTERNET( ) const;
```

See Also

Reference

CInternetFile Class

Hierarchy Chart

Concepts

CInternetFile Members

CInternetSession Class

This topic has been updated for Visual Studio 2005 SP1.

Creates and initializes a single or several simultaneous Internet sessions and, if necessary, describes your connection to a proxy server.

```
class CInternetSession : public CObject
```

Remarks

If your Internet connection must be maintained for the duration of an application, you can create a **CInternetSession** member of the class CWinApp.

Once you have established an Internet session, you can call OpenURL. **CInternetSession** then parses the URL for you by calling the global function AfxParseURL. Regardless of its protocol type, **CInternetSession** interprets the URL and manages it for you. It can handle requests for local files identified with the URL resource "file://". **OpenURL** will return a pointer to a CStdioFile object if the name you pass it is a local file.

If you open a URL on an Internet server using **OpenURL**, you can read information from the site. If you want to perform service-specific (for example, HTTP, FTP, or gopher) actions on files located on a server, you must establish the appropriate connection with that server. To open a particular kind of connection directly to a particular service, use one of the following member functions:

- GetGopherConnection to open a connection to a gopher service.
- GetHttpConnection to open a connection to an HTTP service.
- GetFtpConnection to open a connection to an FTP service.

QueryOption and SetOption allow you to set the query options of your session, such as time-out values, number of retries, and so on.

CInternetSession member functions SetCookie, GetCookie, and GetCookieLength provide the means to manage a Win32 cookie database, through which servers and scripts maintain state information about the client workstation.

For more information about basic Internet programming tasks, see the article Internet First Steps: WinInet. For general information about using the MFC WinInet classes, see the article Internet Programming with WinInet.

Note
CInternetSession will throw an AfxThrowNotSupportedException for unsupported service types. Only the following service types are currently supported: FTP, HTTP, gopher, and file.

Requirements

Header: afxinet.h

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

The Smart Device version of CInternetSession Class does not support the following methods:

- CInternetSession::GetFtpConnection
- CInternetSession::GetGopherConnection
- CInternetSession::SetCookie
- CInternetSession::GetCookie
- CInternetSession::GetCookieLength

Smart Device MFC does not support FTP or Gopher classes. Also, the WinInet API for Windows CE does not support cookies or

Gopher.

See Also

Reference

CObject Class
Hierarchy Chart
CInternetConnection Class
CHttpConnection Class
CFtpConnection Class
CGopherConnection Class

Concepts

CInternetSession Members

CInternetSession Members

This topic has been updated for Visual Studio 2005 SP1.

Base Class Members

CObject Members

Construction

CInternetSession	Constructs a **CInternetSession** object.

Attributes

EnableStatusCallback	Establishes a status callback routine.
GetFtpConnection	Opens an FTP session with a server. Logs on the user.
GetGopherConnection	Opens a gopher server for an application that is trying to open a connection.
GetHttpConnection	Opens an HTTP server for an application that is trying to open a connection.
OpenURL	Parses and opens a URL.
QueryOption	Provides possible asserts for error checking.
ServiceTypeFromHandle	Gets the type of service from the Internet handle.
SetOption	Sets options for the Internet session.

Operations

Close	Closes the Internet connection when the Internet session is terminated.
GetContext	Gets the context value for an Internet or application session.
GetCookie	Returns cookies for the specified URL and all its parent URLs.
GetCookieLength	Retrieves the variable specifying the length of the cookie stored in the buffer.
SetCookie	Sets a cookie for the specified URL.

Overridables

OnStatusCallback	Updates the status of an operation when status callback is enabled.

Operators

operator HINTERNET	A handle to the current Internet session.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

The Smart Device version of CInternetSession Class does not support the following methods:

- CInternetSession::GetFtpConnection
- CInternetSession::GetGopherConnection
- CInternetSession::SetCookie

- CInternetSession::GetCookie
- CInternetSession::GetCookieLength

Smart Device MFC does not support FTP or Gopher classes. Also, the WinInet API for Windows CE does not support Cookies or Gopher.

See Also

Reference

CInternetSession Class
Hierarchy Chart

CInternetSession Member Functions

For information about the member functions in **CInternetSession**, see CInternetSession Members.

CInternetSession::CInternetSession

This member function is called when a **CInternetSession** object is created.

```
CInternetSession(
   LPCTSTR pstrAgent = NULL,
   DWORD_PTR dwContext = 1,
   DWORD dwAccessType = PRE_CONFIG_INTERNET_ACCESS,
   LPCTSTR pstrProxyName = NULL,
   LPCTSTR pstrProxyBypass = NULL,
   DWORD dwFlags = 0
);
```

Parameters

pstrAgent

A pointer to a string that identifies the name of the application or entity calling the Internet functions (for example, "Microsoft Internet Browser"). If *pstrAgent* is **NULL** (the default), the framework calls the global function AfxGetAppName, which returns a null-terminated string containing an application's name. Some protocols use this string to identify your application to the server.

dwContext

The context identifier for the operation. *dwContext* identifies the operation's status information returned by CInternetSession::OnStatusCallback. The default is set to 1; however, you can explicitly assign a specific context ID for the operation. The object and any work it does will be associated with that context ID.

dwAccessType

The type of access required. The following are valid values, exactly one of which may be supplied:

- **INTERNET_OPEN_TYPE_PRECONFIG** Connect using preconfigured settings in the registry. This access type is set as the default. To connect through a TIS proxy, set *dwAccessType* to this value; you then set the registry appropriately.
- **INTERNET_OPEN_TYPE_DIRECT** Connect directly to Internet.
- **INTERNET_OPEN_TYPE_PROXY** Connect through a CERN proxy.

For information on connecting with different types of proxies, see Steps in a Typical FTP Client Application.

pstrProxyName

The name of the preferred CERN proxy if *dwAccessType* is set as **INTERNET_OPEN_TYPE_PROXY**. The default is **NULL**.

pstrProxyBypass

A pointer to a string containing an optional list of server addresses. These addresses may be bypassed when using proxy access. If a **NULL** value is supplied, the bypass list will be read from the registry. This parameter is meaningful only if *dwAccessType* is set to **INTERNET_OPEN_TYPE_PROXY**.

dwFlags

Indicates various caching options. The default is set to 0. The possible values include:

- **INTERNET_FLAG_DONT_CACHE** Do not cache the data, either locally or in any gateway servers.
- **INTERNET_FLAG_OFFLINE** Download operations are satisfied through the persistent cache only. If the item does not exist in the cache, an appropriate error code is returned. This flag may be combined with the bitwise **OR** (**|**) operator.

Remarks

CInternetSession is the first Internet function called by an application. It initializes internal data structures and prepares for future calls from the application.

If no Internet connection can be opened, **CInternetSession** throws an AfxThrowInternetException.

Example

See the example for CFtpFileFind.

See Also

Reference

CInternetSession Class
Hierarchy Chart
CInternetSession::Close
CInternetSession::EnableStatusCallback
CInternetSession::GetContext

Concepts

CInternetSession Members

CInternetSession::Close

Call this member function when your application has finished using the **CInternetSession** object.

```
virtual void Close( );
```

Example

See the example for CFtpFileFind.

See Also

Reference

CInternetSession Class
Hierarchy Chart
CInternetSession::CInternetSession

Concepts

CInternetSession Members

CInternetSession::EnableStatusCallback

Call this member function to enable status callback.

```
BOOL EnableStatusCallback(
   BOOL bEnable = TRUE
);
```

Parameters

bEnable

Specifies whether callback is enabled or disabled. The default is **TRUE**.

Return Value

Nonzero if successful; otherwise 0. If the call fails, determine the cause of the failure by examining the thrown CInternetException object.

Remarks

When handling status callback, you can provide status about the progress of the operation (such as resolving name, connecting to server, and so on) in the status bar of the application. Displaying operation status is especially desirable during a long-term operation.

Because callbacks occur during the request's processing, the application should spend as little time as possible in the callback to prevent degradation of data throughput to the network. For example, putting up a dialog box in a callback may be such a lengthy operation that the server terminates the request.

The status callback cannot be removed as long as any callbacks are pending.

To handle any operations asynchronously, you must either create your own thread or use the WinInet functions without MFC.

Exceptions

This method can throw exceptions of type **CInternetException***.

See Also

Reference

CInternetSession Class
Hierarchy Chart
CInternetSession::CInternetSession

Concepts

CInternetSession Members

CInternetSession::GetContext

Call this member function to get the context value for a particular application session.

```
DWORD_PTR GetContext( ) const;
```

Return Value

The application-defined context Identifier.

Remarks

OnStatusCallback uses the context ID returned by **GetContext** to report the status of a particular application. For example, when a user activates an Internet request that involves returning status information, the status callback uses the context ID to report status on that particular request. If the user activates two separate Internet requests that both involve returning status information, **OnStatusCallback** uses the context identifiers to return status about their corresponding requests. Consequently, the context identifier is used for all status callback operations, and it is associated with the session until the session is ended.

For more information about asynchronous operations, see the article Internet First Steps: WinInet.

See Also

Reference

CInternetSession Class
Hierarchy Chart
CInternetConnection Class
CInternetSession::EnableStatusCallback
CInternetSession::OnStatusCallback

Concepts

CInternetSession Members

CInternetSession::GetCookie

This topic has been updated for Visual Studio 2005 SP1.

This member function implements the behavior of the Win32 function InternetGetCookie, as described in the *Platform SDK*.

```
static BOOL GetCookie(
   LPCTSTR pstrUrl,
   LPCTSTR pstrCookieName,
   LPTSTR pstrCookieData,
   DWORD dwBufLen
);
static BOOL GetCookie(
   LPCTSTR pstrUrl,
   LPCTSTR pstrCookieName,
   CString& strCookieData
);
```

Parameters

pstrUrl

A pointer to a string containing the URL.

pstrCookieName

A pointer to a string containing the name of the cookie to get for the specified URL.

pstrCookieData

In the first overload, a pointer to a string containing the address of the buffer that receives the cookie data. This value can be **NULL**. In the second overload, a reference to a CString object to receive the cookie data.

dwBufLen

The variable specifying the size of the *pstrCookieData* buffer. If the function succeeds, the buffer receives the amount of data copied to the *pstrCookieData* buffer. If *pstrCookieData* is **NULL**, this parameter receives a value that specifies the size of the buffer necessary to copy all the cookie data.

Return Value

Returns **TRUE** if successful, or **FALSE** otherwise. If the call fails, call the Win32 function GetLastError to determine the cause of the error. The following error values apply:

- **ERROR_NO_MORE_ITEMS** There is no cookie for the specified URL and all its parents.
- **ERROR_INSUFFICIENT_BUFFER** The value passed in *dwBufLen* is insufficient to copy all the cookie data. The value returned in *dwBufLen* is the size of the buffer necessary to get all the data.

Remarks

In the second overload, MFC retrieves the cookie data into the supplied **CString** object.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CInternetSession::GetCookieLength
CInternetSession::SetCookie

CInternetSession::GetCookieLength

This topic has been updated for Visual Studio 2005 SP1.

Call this member function to get the length of the cookie stored in the buffer.

```
static DWORD GetCookieLength(
   LPCTSTR pstrUrl,
   LPCTSTR pstrCookieName
);
```

Parameters

pstrUrl

A pointer to a string containing the URL

pstrCookieName

A pointer to a string containing the name of the cookie.

Return Value

A **DWORD** value indicating the length of the cookie, stored in the buffer. Zero if no cookie with the name indicated by *pstrCookieName* exists.

Remarks

This value is used by GetCookie.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CInternetSession Class
Hierarchy Chart
CInternetSession::SetCookie

Concepts

CInternetSession Members

CInternetSession::GetFtpConnection

This topic has been updated for Visual Studio 2005 SP1.

Call this member function to establish an FTP connection and get a pointer to a **CFtpConnection** object.

```
CFtpConnection* GetFtpConnection(
   LPCTSTR pstrServer,
   LPCTSTR pstrUserName = NULL,
   LPCTSTR pstrPassword = NULL,
   INTERNET_PORT nPort = INTERNET_INVALID_PORT_NUMBER,
   BOOL bPassive = FALSE
);
```

Parameters

pstrServer

A pointer to a string containing the FTP server name.

pstrUserName

Pointer to a null-terminated string that specifies the name of the user to log in. If **NULL**, the default is anonymous.

pstrPassword

A pointer to a null-terminated string that specifies the password to use to log in. If both *pstrPassword* and *pstrUserName* are **NULL**, the default anonymous password is the user's email name. If *pstrPassword* is **NULL** (or an empty string) but *pstrUserName* is not **NULL**, a blank password is used. The following table describes the behavior for the four possible settings of *pstrUserName* and *pstrPassword*:

pstrUserName	***pstrPassword***	**Username sent to FTP server**	**Password sent to FTP server**
NULL or " "	**NULL** or " "	"anonymous"	User's email name
Non-**NULL** String	**NULL** or " "	*pstrUserName*	" "
NULL Non-**NULL** String	**ERROR**	**ERROR**	
Non-**NULL** String	Non-**NULL** String	*pstrUserName*	*pstrPassword*

nPort

A number that identifies the TCP/IP port to use on the server.

bPassive

Specifies passive or active mode for this FTP session. If set to **TRUE**, it sets the Win32 API *dwFlag* to **INTERNET_FLAG_PASSIVE**.

Return Value

A pointer to a CFtpConnection object. If the call fails, determine the cause of the failure by examining the thrown CInternetException object.

Remarks

GetFtpConnection connects to an FTP server, and creates and returns a pointer to a **CFTPConnection** object. It does not perform any specific operation on the server. If you intend to read or write to files, for example, you must perform those operations as separate steps. See the classes CFtpConnection and CFtpFileFind for information about searching for files, opening files, and reading or writing to files. See the article Internet Programming with WinInet for steps in performing common FTP connection tasks.

Exceptions

This method can throw exceptions of type **CInternetException***.

Example

See the example for CFtpFileFind.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CInternetSession Class
Hierarchy Chart
CFtpConnection Class
CInternetSession::GetGopherConnection
CInternetSession::GetHttpConnection
CInternetSession::OpenURL

Concepts

CInternetSession Members

CInternetSession::GetGopherConnection

This topic has been updated for Visual Studio 2005 SP1.

Call this member function to establish a new gopher connection and get a pointer to a **CGopherConnection** object.

```
CGopherConnection* GetGopherConnection(
   LPCTSTR pstrServer,
   LPCTSTR pstrUserName = NULL,
   LPCTSTR pstrPassword = NULL,
   INTERNET_PORT nPort = INTERNET_INVALID_PORT_NUMBER
);
```

Parameters

pstrServer

A pointer to a string containing the gopher server name.

pstrUserName

A pointer to a string containing the user name.

pstrPassword

A pointer to a string containing the access password.

nPort

A number that identifies the TCP/IP port to use on the server.

Return Value

A pointer to a CGopherConnection object. If the call fails, determine the cause of the failure by examining the thrown CInternetException object.

Remarks

GetGopherConnection connects to a gopher server, and creates and returns a pointer to a **CGopherConnection** object. It does not perform any specific operation on the server. If you intend to read or write data, for example, you must perform those operations as separate steps. See the classes CGopherConnection, CGopherFile, and CGopherFileFind for information about searching for files, opening files, and reading or writing to files. For information about browsing an FTP site, see the member function OpenURL. See the article Internet Programming with WinInet for steps in performing common gopher connection tasks.

Exceptions

This method can throw exceptions of type **CInternetException***.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CInternetSession Class
Hierarchy Chart
CGopherConnection Class
CInternetSession::GetFtpConnection
CInternetSession::GetHttpConnection
CInternetSession::OpenURL

Concepts

CInternetSession Members

CInternetSession::GetHttpConnection

Call this member function to establish an HTTP connection and get a pointer to a **CHttpConnection** object.

```
CHttpConnection* GetHttpConnection(
   LPCTSTR pstrServer,
   INTERNET_PORT nPort = INTERNET_INVALID_PORT_NUMBER,
   LPCTSTR pstrUserName = NULL,
   LPCTSTR pstrPassword = NULL
);
CHttpConnection* GetHttpConnection(
   LPCTSTR pstrServer,
   DWORD dwFlags,
   INTERNET_PORT nPort = INTERNET_INVALID_PORT_NUMBER,
   LPCTSTR pstrUserName = NULL,
   LPCTSTR pstrPassword = NULL
);
```

Parameters

pstrServer

A pointer to a string containing the HTTP server name.

nPort

A number that identifies the TCP/IP port to use on the server.

pstrUserName

A pointer to a string containing the user name.

pstrPassword

A pointer to a string containing the access password.

dwflags

Any combination of the **INTERNET_ FLAG_*** flags. See the table in the **Remarks** section of CHttpConnection::OpenRequest for a description of *dwFlags* values.

Return Value

A pointer to a CHttpConnection object. If the call fails, determine the cause of the failure by examining the thrown CInternetException object.

Remarks

GetHttpConnection connects to an HTTP server, and creates and returns a pointer to a **CHttpConnection** object. It does not perform any specific operation on the server. If you intend to query an HTTP header, for example, you must perform this operation as a separate step. See the classes CHttpConnection and CHttpFile for information about operations you can perform by using a connection to an HTTP server. For information about browsing an HTTP site, see the member function OpenURL. See the article Internet Programming with WinInet for steps in performing common HTTP connection tasks.

Exceptions

This method can throw exceptions of type **CInternetException***.

See Also

Reference

CInternetSession Class
Hierarchy Chart
CHttpConnection Class
CInternetSession::GetGopherConnection
CInternetSession::GetFtpConnection
CInternetSession::OpenURL

Concepts

CInternetSession Members

CInternetSession::OnStatusCallback

This member function is called by the framework to update the status when status callback is enabled and an operation is pending.

```
virtual void OnStatusCallback(
   DWORD_PTR dwContext,
   DWORD dwInternetStatus,
   LPVOID lpvStatusInformation,
   DWORD dwStatusInformationLength
);
```

Parameters

dwContext

The context value supplied by the application.

dwInternetStatus

A status code which indicates why the callback is being made. See **Remarks** for a table of possible values.

lpvStatusInformation

A pointer to a buffer containing information pertinent to this callback.

dwStatusInformationLength

The size of *lpvStatusInformation*.

Remarks

You must first call EnableStatusCallback to take advantage of status callback.

The *dwInternetStatus* parameter indicates the operation being performed and determines what the contents of *lpvStatusInformation* will be. *dwStatusInformationLength* indicates the length of the data included in *lpvStatusInformation*. The following status values for *dwInternetStatus* are defined as follows:

Value	Meaning
INTERNET_STATUS_RESOLVING_NAME	Looking up the IP address of the name contained in *lpvStatusInformation*.
INTERNET_STATUS_NAME_RESOLVED	Successfully found the IP address of the name contained in *lpvStatusInformation*.
INTERNET_STATUS_CONNECTING_TO_SERVER	Connecting to the socket address (SOCKADDR) pointed to by *lpvStatusInformation*.
INTERNET_STATUS_CONNECTED_TO_SERVER	Successfully connected to the socket address (**SOCKADDR**) pointed to by *lpvStatusInformation*.
INTERNET_STATUS_SENDING_REQUEST	Sending the information request to the server. The *lpvStatusInformation* parameter is **NULL**.
INTERNET_STATUS_ REQUEST_SENT	Successfully sent the information request to the server. The *lpvStatusInformation* parameter is **NULL**.

INTERNET_STATUS_RECEIVING_RESPONSE	Waiting for the server to respond to a request. The *lpvStatusInformation* parameter is **NULL**.
INTERNET_STATUS_RESPONSE_RECEIVED	Successfully received a response from the server. The *lpvStatusInformation* parameter is **NULL**.
INTERNET_STATUS_CLOSING_CONNECTION	Closing the connection to the server. The *lpvStatusInformation* parameter is **NULL**.
INTERNET_STATUS_CONNECTION_CLOSED	Successfully closed the connection to the server. The *lpvStatusInformation* parameter is **NULL**.
INTERNET_STATUS_HANDLE_CREATED	Used by the Win32 API function InternetConnect to indicate that it has created the new handle. This lets the application call the Win32 function InternetCloseHandle from another thread if the connect is taking too long. See the *Platform SDK* for more information about these functions.
INTERNET_STATUS_HANDLE_CLOSING	Successfully terminated this handle value.

Override this member function to require some action before a status callback routine is performed.

Note
Status callbacks need thread-state protection. If you are using MFC in a shared library, add the following line to the beginning of your override:

```
AFX_MANAGE_STATE( AfxGetAppModuleState( ) );
```

For more information about asynchronous operations, see the article Internet First Steps: WinInet.

See Also

Reference

CInternetSession Class
Hierarchy Chart
CInternetSession::EnableStatusCallback
CInternetSession::GetContext

Concepts

CInternetSession Members

CInternetSession::OpenURL

Call this member function to send the specified request to the HTTP server and allow the client to specify additional RFC822, MIME, or HTTP headers to send along with the request.

```
CStdioFile* OpenURL(
   LPCTSTR pstrURL,
   DWORD_PTR dwContext = 1,
   DWORD dwFlags = INTERNET_FLAG_TRANSFER_ASCII,
   LPCTSTR pstrHeaders = NULL,
   DWORD dwHeadersLength = 0
);
```

Parameters

pstrURL

A pointer to the name of the URL to begin reading. Only URLs beginning with file:, ftp:, gopher:, or http: are supported. **ASSERTS** if *pszURL* is **NULL**.

dwContext

An application-defined value passed with the returned handle in callback.

dwFlags

The flags describing how to handle this connection. See **Remarks** for more information about the valid flags. The valid flags are:

- **INTERNET_FLAG_TRANSFER_ASCII** The default. Transfer the file as ASCII text.
- **INTERNET_FLAG_TRANSFER_BINARY** Transfer the file as a binary file.
- **INTERNET_FLAG_RELOAD** Get the data from the wire even if it is locally cached.
- **INTERNET_FLAG_DONT_CACHE** Do not cache the data, either locally or in any gateways.
- **INTERNET_FLAG_SECURE** This flag is applicable to HTTP requests only. It requests secure transactions on the wire with Secure Sockets Layer or PCT.
- **INTERNET_OPEN_FLAG_USE_EXISTING_CONNECT** If possible, reuse the existing connections to the server for new requests generated by **OpenUrl** instead of creating a new session for each connection request.
- **INTERNET_FLAG_PASSIVE** Used for an FTP site. Uses passive FTP semantics. Used with CInternetConnection of **OpenURL**.

pstrHeaders

A pointer to a string containing the headers to be sent to the HTTP server.

dwHeadersLength

The length, in characters, of the additional headers. If this is -1L and *pstrHeaders* is non-**NULL**, then *pstrHeaders* is assumed to be zero terminated and the length is calculated.

Return Value

Returns a file handle for FTP, GOPHER, HTTP, and FILE-type Internet services only. Returns **NULL** if parsing was unsuccessful.

The pointer that **OpenURL** returns depends on *pszURL*'s type of service. The table below illustrates the possible pointers **OpenURL** can return.

URL type	Returns
file://	**CStdioFile***
http://	**CHttpFile***

gopher://	**CGopherFile***
ftp://	**CInternetFile***

Remarks

The parameter *dwFlags* must include either **INTERNET_FLAG_TRANSFER_ASCII** or **INTERNET_FLAG_TRANSFER_BINARY**, but not both. The remaining flags can be combined with the bitwise **OR** operator (**|**).

OpenURL, which wraps the Win32 function **InternetOpenURL**, allows only downloading, retrieving, and reading the data from an Internet server. **OpenURL** allows no file manipulation on a remote location, so it requires no CInternetConnection object.

To use connection-specific (that is, protocol-specific) functions, such as writing to a file, you must open a session, then open a particular kind of connection, then use that connection to open a file in the desired mode. See **CInternetConnection** for more information about connection-specific functions.

Exceptions

This method can throw exceptions of type **CInternetException***.

See Also

Reference

CInternetSession Class
Hierarchy Chart
CInternetConnection Class
CGopherConnection Class
CInternetSession::GetFtpConnection
CInternetSession::GetHttpConnection

Concepts

CInternetSession Members

CInternetSession::QueryOption

Provides five possible asserts for basic error-checking.

```
BOOL QueryOption(
   DWORD dwOption,
   LPVOID lpBuffer,
   LPDWORD lpdwBufLen
) const;
BOOL QueryOption(
   DWORD dwOption,
   DWORD& dwValue
) const;
```

Parameters

dwOption

The Internet option to query. See Option Flags in the *Platform SDK* for a list of the possible options.

lpBuffer

A buffer that receives the option setting.

lpdwBufLen

A pointer to a DWORD containing the length of *lpBuffer*. On return, this contains the length of the data placed into *lpBuffer*.

dwValue

Sent to **QueryOption** in place of *lpBuffer*.

Return Value

If the operation was successful, a value of **TRUE** is returned. If an error occurred, a value of **FALSE** is returned. If the call fails, the Win32 function GetLastError may be called to determine the cause of the error.

Remarks

See CInternetSession::SetOption to select and set the specific option to query.

The possible settings for **INTERNET_OPTION_HANDLE_TYPE** include the following:

- **INTERNET_HANDLE_TYPE_INTERNET**
- **INTERNET_HANDLE_TYPE_CONNECT_FTP**
- **INTERNET_HANDLE_TYPE_CONNECT_GOPHER**
- **INTERNET_HANDLE_TYPE_CONNECT_HTTP**
- **INTERNET_HANDLE_TYPE_FTP_FIND**
- **INTERNET_HANDLE_TYPE_FTP_FIND_HTML**
- **INTERNET_HANDLE_TYPE_FTP_FILE**
- **INTERNET_HANDLE_TYPE_FTP_FILE_HTML**
- **INTERNET_HANDLE_TYPE_GOPHER_FIND**
- **INTERNET_HANDLE_TYPE_GOPHER_FIND_HTML**
- **INTERNET_HANDLE_TYPE_GOPHER_FILE**
- **INTERNET_HANDLE_TYPE_GOPHER_FILE_HTML**
- **INTERNET_HANDLE_TYPE_HTTP_REQUEST**

Note that the following form of this method has been deprecated (the two valid forms are shown in the syntax section above):

```
CInternetSession::QueryOption(DWORD dwOption, CString& refString);
```

See Also

Reference

CInternetSession Class

Hierarchy Chart

CInternetSession::ServiceTypeFromHandle

CInternetSession::SetOption

Concepts

CInternetSession Members

CInternetSession::ServiceTypeFromHandle

Call this member function to get the type of service from the Internet handle.

```
DWORD ServiceTypeFromHandle(
   HINTERNET hQuery
);
```

Parameters

hQuery

A handle to an Internet query.

Return Value

The Internet service type. See **Remarks** for a list of recognized service types.

Remarks

The following service types are recognized by MFC:

Service type	Return value
FTP	**INTERNET_SERVICE_FTP**
HTTP	**INTERNET_SERVICE_HTTP**
Gopher	**INTERNET_SERVICE_GOPHER**
File	**AFX_INET_SERVICE_FILE**

CInternetSession will throw an AfxThrowNotSupportedException object for unsupported service types.

Note
The return value **AFX_INET_SERVICE_FILE** is used only by MFC and is not recognized by Win32. This feature allows the client to access local files in the same way he or she would access Internet services.

Exceptions

This method can throw exceptions of type **AfxThrowNotSupportedException**.

See Also

Reference

CInternetSession Class
Hierarchy Chart
CInternetSession::QueryOption
CInternetSession::operator HINTERNET

Concepts

CInternetSession Members

CInternetSession::SetCookie

This topic has been updated for Visual Studio 2005 SP1.

Sets a cookie for the specified URL.

```
static BOOL SetCookie(
   LPCTSTR pstrUrl,
   LPCTSTR pstrCookieName,
   LPCTSTR pstrCookieData
);
```

Parameters

pstrUrl

A pointer to a null-terminated string that specifies the URL for which the cookie should be set.

pstrCookieName

A pointer to a string containing the name of the cookie.

pstrCookieData

A pointer to a string containing the actual string data to associate with the URL.

Return Value

Returns **TRUE** if successful, or **FALSE** otherwise. To get the specific error code, call **GetLastError.**

Remarks

This member function implements the behavior of the Win32 message InternetSetCookie, as described in the *Platform SDK*.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CInternetSession::GetCookieLength
CInternetSession::GetCookie

CInternetSession::SetOption

Call this member function to set options for the Internet session.

```
BOOL SetOption(
   DWORD dwOption,
   LPVOID lpBuffer,
   DWORD dwBufferLength,
   DWORD dwFlags = 0
);
BOOL SetOption(
   DWORD dwOption,
   DWORD dwValue,
   DWORD dwFlags = 0
);
```

Parameters

dwOption

The Internet option to set. See Option Flags in the *Platform SDK* for a list of the possible options.

lpBuffer

A buffer that contains the option setting.

dwBufferLength

The length of *lpBuffer* or the size of *dwValue*.

dwValue

A **DWORD** that contains the option setting.

dwFlags

Indicates various caching options. The default is set to 0. The possible values include:

- **INTERNET_FLAG_DONT_CACHE** Do not cache the data, either locally or in any gateway servers.
- **INTERNET_FLAG_OFFLINE** Download operations are satisfied through the persistent cache only. If the item does not exist in the cache, an appropriate error code is returned. This flag may be combined with the bitwise **OR** (**|**) operator.

Return Value

If the operation was successful, a value of **TRUE** is returned. If an error occurred, a value of **FALSE** is returned. If the call fails, the Win32 function GetLastError may be called to determine the cause of the error.

See Also

Reference

CInternetSession Class
Hierarchy Chart
CInternetSession::ServiceTypeFromHandle
CInternetSession::QueryOption

Concepts

CInternetSession Members

CInternetSession Operators

For information about the operators in **CInternetSession**, see CInternetSession Members.

CInternetSession::operator HINTERNET

Use this operator to get the Windows handle for the current Internet session.

```
operator HINTERNET( ) const;
```

See Also

Reference

CInternetSession Class

Hierarchy Chart

CInternetSession::ServiceTypeFromHandle

Concepts

CInternetSession Members

CInvalidArgException Class

This class represents an invalid argument exception condition.

```
class CInvalidArgException : public CSimpleException
```

Remarks

A **CInvalidArgException** object represents an invalid argument exception condition.

For more information on Exception Handling, see the CException Class topic and Exception Handling (MFC).

Requirements

Header: afx.h

See Also

Reference

Hierarchy Chart

CSimpleException Class

Other Resources

CInvalidArgException Members

CInvalidArgException Members

Methods

CInvalidArgException	The constructor.

See Also

Reference

CInvalidArgException Class

CInvalidArgException Methods

For information about the methods in **CInvalidArgException**, see CInvalidArgException Members.

CInvalidArgException::CInvalidArgException

The constructor.

```
CInvalidArgException( );
```

Remarks

Do not use this constructor directly; call the global function **AfxThrowInvalidArgException**.

See Also

Reference

CInvalidArgException Class

Other Resources

CInvalidArgException Members

CIPAddressCtrl Class

Provides the functionality of the Windows common IP Address control.

```
class CIPAddressCtrl : public CWnd
```

Remarks

An IP Address control, a control similar to an edit control, allows you to enter and manipulate a numerical address in Internet Protocol (IP) format.

This control (and therefore the **CIPAddressCtrl** class) is available only to programs running under Microsoft Internet Explorer 4.0 and later. They will also be available under future versions of Windows and Windows NT.

For more general information about the IP Address Control, see IP Address Controls in the *Platform SDK*.

Requirements

Header: afxcmn.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CWnd Class

Hierarchy Chart

Other Resources

CIPAddressCtrl Members

CIPAddressCtrl Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Construction

CIPAddressCtrl	Constructs a **CIPAddressCtrl** object.
Create	Creates an IP Address Control and attaches it to a **CIPAddressCtrl** object.
CreateEx	Creates an IP Address control with the specified Windows extended styles and attaches it to a **CIPAddressCtrl** object.

Attributes

ClearAddress	Clears the contents of the IP Address Control.
GetAddress	Retrieves the address values for all four fields in the IP Address Control.
IsBlank	Determines if all fields in the IP Address Control are empty.
SetAddress	Sets the address values for all four fields in the IP Address Control.
SetFieldFocus	Sets the keyboard focus to the specified field in the IP Address Control.
SetFieldRange	Sets the range in the specified field in the IP Address Control.

See Also

Reference

CIPAddressCtrl Class

Hierarchy Chart

CIPAddressCtrl Member Functions

For information about the member functions in **CIPAddressCtrl**, see CIPAddressCtrl Members.

CIPAddressCtrl::CIPAddressCtrl

Creates a **CIPAddressCtrl** object.

```
CIPAddressCtrl( );
```

See Also

Reference

CIPAddressCtrl Class
Hierarchy Chart
CIPAddressCtrl::Create
CIPAddressCtrl::CreateEx

Other Resources

CIPAddressCtrl Members

CIPAddressCtrl::ClearAddress

Clears the contents of the IP Address Control.

```
void ClearAddress( );
```

Remarks

This member function implements the behavior of the Win32 message IPM_CLEARADDRESS, as described in the *Platform SDK*.

See Also

Reference

CIPAddressCtrl Class
Hierarchy Chart
CIPAddressCtrl::GetAddress
CIPAddressCtrl::SetAddress

Other Resources

CIPAddressCtrl Members

CIPAddressCtrl::Create

Creates an IP Address Control and attaches it to a **CIPAddressCtrl** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

The IP Address control's style. Apply a combination of window styles. You must include the **WS_CHILD** style because the control must be a child window. See CreateWindow in the *Platform SDK* for a list of windows styles.

rect

A reference to the IP Address Control's size and position. It can be either a CRect object or a RECT structure.

pParentWnd

A pointer to the IP Address Control's parent window. It must not be **NULL.**

nID

The IP Address Control's ID.

Return Value

Nonzero if initialization was successful; otherwise 0.

Remarks

You construct a **CIPAddressCtrl** object in two steps.

1. Call the constructor, which creates the **CIPAddressCtrl** object.
2. Call **Create**, which creates the IP Address Control.

If you want to use extended windows styles with your control, call CreateEx instead of **Create**.

See Also

Reference

CIPAddressCtrl Class
Hierarchy Chart
CIPAddressCtrl::CIPAddressCtrl

Other Resources

CIPAddressCtrl Members

CIPAddressCtrl::CreateEx

Call this function to create a control (a child window) and associate it with the **CIPAddressCtrl** object.

```
virtual BOOL CreateEx(
   DWORD dwExStyle,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwExStyle

Specifies the extended style of the control being created. For a list of extended Windows styles, see the *dwExStyle* parameter for CreateWindowEx in the *Platform SDK*.

dwStyle

The IP Address control's style. Apply a combination of window styles. You must include the **WS_CHILD** style because the control must be a child window. See CreateWindow in the *Platform SDK* for a list of windows styles.

rect

A reference to a RECT structure describing the size and position of the window to be created, in client coordinates of *pParentWnd*.

pParentWnd

A pointer to the window that is the control's parent.

nID

The control's child-window ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Use **CreateEx** instead of Create to apply extended Windows styles, specified by the Windows extended style preface **WS_EX_**.

See Also

Reference

CIPAddressCtrl Class
Hierarchy Chart
CIPAddressCtrl::CIPAddressCtrl

Other Resources

CIPAddressCtrl Members

CIPAddressCtrl::GetAddress

Retrieves the address values for all four fields in the IP Address Control.

```
int GetAddress(
   BYTE& nField0,
   BYTE& nField1,
   BYTE& nField2,
   BYTE& nField3
);
int GetAddress(
   DWORD& dwAddress
);
```

Parameters

nField0

A reference to the field 0 value from a packed IP address.

nField1

A reference to the field 1 value from a packed IP address.

nField2

A reference to the field 2 value from a packed IP address.

nField3

A reference to the field 3 value from a packed IP address.

dwAddress

A reference to the address of a **DWORD** value that receives the IP address. See **Remarks** for a table that shows how *dwAddress* is filled.

Return Value

The number of non-blank fields in the IP Address Control.

Remarks

This member function implements the behavior of the Win32 message IPM_GETADDRESS, as described in the *Platform SDK*. In the first prototype above, the numbers in fields 0 through 3 of the control, read left to right respectively, populate the four parameters. In the second prototype above, *dwAddress* is populated as follows.

Field	Bits containing the field value
0	24 through 31
1	16 through 23
2	8 through 15
3	0 through 7

See Also

Reference

CIPAddressCtrl Class
Hierarchy Chart
CIPAddressCtrl::SetAddress

Other Resources

CIPAddressCtrl Members

CIPAddressCtrl::IsBlank

Determines if all fields in the IP Address Control are empty.

```
BOOL IsBlank( ) const;
```

Return Value

Nonzero if all of the IP Address Control fields are empty; otherwise 0.

Remarks

This member function implements the behavior of the Win32 message IPM_ISBLANK, as described in the *Platform SDK*.

See Also

Reference

CIPAddressCtrl Class

Hierarchy Chart

Other Resources

CIPAddressCtrl Members

CIPAddressCtrl::SetAddress

Sets the address values for all four fields in the IP Address Control.

```
void SetAddress(
   BYTE nField0,
   BYTE nField1,
   BYTE nField2,
   BYTE nField3
);
void SetAddress(
   DWORD dwAddress
);
```

Parameters

nField0

The field 0 value from a packed IP address.

nField1

The field 1 value from a packed IP address.

nField2

The field 2 value from a packed IP address.

nField3

The field 3 value from a packed IP address.

dwAddress

A **DWORD** value that contains the new IP address. See **Remarks** for a table that shows how the **DWORD** value is filled.

Remarks

This member function implements the behavior of the Win32 message IPM_SETADDRESS, as described in the *Platform SDK*. In the first prototype above, the numbers in fields 0 through 3 of the control, read left to right respectively, populate the four parameters. In the second prototype above, *dwAddress* is populated as follows.

Field	Bits containing the field value
0	24 through 31
1	16 through 23
2	8 through 15
3	0 through 7

See Also

Reference

CIPAddressCtrl Class
Hierarchy Chart
CIPAddressCtrl::GetAddress
CIPAddressCtrl::ClearAddress

Other Resources

CIPAddressCtrl Members

CIPAddressCtrl::SetFieldFocus

Sets the keyboard focus to the specified field in the IP Address Control.

```
void SetFieldFocus(
   WORD nField
);
```

Parameters

nField

Zero-based field index to which the focus should be set. If this value is greater than the number of fields, focus is set to the first blank field. If all fields are non-blank, focus is set to the first field.

Remarks

This member function implements the behavior of the Win32 message IPM_SETFOCUS, as described in the *Platform SDK*.

See Also

Reference

CIPAddressCtrl Class
Hierarchy Chart
CIPAddressCtrl::SetFieldRange

Other Resources

CIPAddressCtrl Members

CIPAddressCtrl::SetFieldRange

Sets the range in the specified field in the IP Address Control.

```
void SetFieldRange(
   int nField,
   BYTE nLower,
   BYTE nUpper
);
```

Parameters

nField

Zero-based field index to which the range will be applied.

nLower

A reference to an integer receiving the lower limit of the specified field in this IP Address Control.

nUpper

A reference to an integer receiving the upper limit of the specified field in this IP Address Control.

Remarks

This member function implements the behavior of the Win32 message IPM_SETRANGE, as described in the *Platform SDK*. Use the two parameters, *nLower* and *nUpper*, to indicate the lower and upper limits of the field, instead of the *wRange* parameter used with the Win32 message.

See Also

Reference

CIPAddressCtrl Class
Hierarchy Chart
CIPAddressCtrl::SetFieldFocus

Other Resources

CIPAddressCtrl Members

CLinkCtrl Class

Provides the functionality of the Windows common SysLink control.

```
class CLinkCtrl : public CWnd
```

Remarks

A "link control" provides a convenient way to embed hypertext links in a window. The actual control is a window that renders marked-up text and launches appropriate applications when the user clicks an embedded link. Multiple links are supported within one control and can be accessed by a zero-based index.

This control (and therefore the **CLinkCtrl** class) is available only to programs running under Windows XP and later.

For more information, see SysLink Control in the *Platform SDK*.

Requirements

Header: afxcmn.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart

CWnd Class

Other Resources

CLinkCtrl Members

CLinkCtrl Members

Member Functions

CLinkCtrl	Constructs a **CLinkCtrl** object.
Create	Creates a link control and attaches it to a **CLinkCtrl** object.
CreateEx	Creates a link control (with extended styles) and attaches it to a **CLinkCtrl** object.
GetIdealHeight	Retrieves the ideal height of the link control.
GetItem	Retrieves the states and attributes of a link control item.
GetItemID	Retrieves the ID of a link control item.
GetItemState	Retrieves the state of the link control item.
GetItemUrl	Retrieves the URL represented by the link control item.
HitTest	Determines if the user clicked the specified link.
SetItem	Sets the states and attributes of a link control item.
SetItemID	Sets the ID of a link control item.
SetItemState	Sets the state of the link control item.
SetItemUrl	Sets the URL represented by the link control item.

See Also

Reference

CLinkCtrl Class

Hierarchy Chart

CLinkCtrl Member Functions

For information about the members in **CLinkCtrl**, see CLinkCtrl Members

CLinkCtrl::CLinkCtrl

Constructs a **CLinkCtrl** object.

```
CLinkCtrl( );
```

See Also

Reference

CLinkCtrl Class
Hierarchy Chart
CLinkCtrl::Create
CLinkCtrl::CreateEx

Other Resources

CLinkCtrl Members

CLinkCtrl::Create

Creates a link control and attaches it to a **CLinkCtrl** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the link control's style. Apply any combination of control styles. See Common Control Styles in the *Platform SDK* for more information.

rect

Specifies the link control's size and position. It can be either a CRect object or a RECT structure.

pParentWnd

Specifies the link control's parent window. It must not be **NULL**.

nID

Specifies the link control's ID.

Return Value

TRUE if initialization was successful; otherwise **FALSE**.

Remarks

You construct a **CLinkCtrl** object in two steps. First, call the constructor and then call **Create**, which creates the link control and attaches it to the **CLinkCtrl** object.

If you want to use extended windows styles with your control, call CreateEx instead of **Create**.

See Also

Reference

CLinkCtrl Class
Hierarchy Chart
CLinkCtrl::CreateEx

Other Resources

CLinkCtrl Members

CLinkCtrl::CreateEx

Creates a link control (with extended styles) and attaches it to a **CLinkCtrl** object.

```
virtual BOOL CreateEx(
   DWORD dwExStyle,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwExStyle

Specifies the extended style of the link control. For a list of extended Windows styles, see the *dwExStyle* parameter for CreateWindowEx in the *Platform SDK*.

dwStyle

Specifies the link control's style. Apply any combination of control styles. See Common Control Styles in the *Platform SDK* for more information.

rect

Specifies the link control's size and position. It can be either a CRect object or a RECT structure.

pParentWnd

Specifies the link control's parent window. It must not be **NULL**.

nID

Specifies the link control's ID.

Return Value

TRUE if initialization was successful; otherwise **FALSE**.

Remarks

Use **CreateEx** instead of Create to apply extended Windows styles, specified by the Windows extended style preface **WS_EX_**.

See Also

Reference

CLinkCtrl Class
Hierarchy Chart
CLinkCtrl::Create

Other Resources

CLinkCtrl Members

CLinkCtrl::GetIdealHeight

Retrieves the ideal height of the link control.

```
int GetIdealHeight( ) const;
```

Return Value

The ideal height of the control, in pixels.

Remarks

This member function implements the behavior of the Win32 message LM_GETIDEALHEIGHT, as described in the *Platform SDK*.

See Also

Reference

CLinkCtrl Class

Hierarchy Chart

Other Resources

CLinkCtrl Members

CLinkCtrl::GetItem

Retrieves the states and attributes of a link control item.

```
BOOL GetItem(
   PLITEM pItem
) const;
```

Parameters

pItem

A pointer to a LITEM structure to receive item information.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This member function implements the behavior of the Win32 message LM_GETITEM, as described in the *Platform SDK*.

See Also

Reference

CLinkCtrl Class
Hierarchy Chart
CLinkCtrl::SetItem
CLinkCtrl::GetItemID
CLinkCtrl::GetItemState
CLinkCtrl::GetItemUrl

Other Resources

CLinkCtrl Members

CLinkCtrl::GetItemID

Retrieves the ID of a link control item.

```
BOOL GetItemID(
   int iLink,
   CString& strID
) const;
BOOL GetItemID(
   int iLink,
   LPWSTR szID,
   UINT cchID
) const;
```

Parameters

iLink

The index of a link control item.

strID

A CStringT object containing the ID of the specified item.

szID

A null-terminated string containing the ID of the specified item.

cchID

The size in characters of the *szID* buffer.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Note
This function also returns **FALSE** if the buffer of *szID or strID* is smaller than **MAX_LINKID_TEXT**.

Remarks

Retrieves the ID of a specific link control item. For more information, see the Win32 message LM_GETITEM in the *Platform SDK*.

See Also

Reference

CLinkCtrl Class
Hierarchy Chart
CLinkCtrl::SetItemID
CLinkCtrl::GetItem

Other Resources

CLinkCtrl Members

CLinkCtrl::GetItemState

Retrieves the state of the link control item.

```
BOOL GetItemState(
   int iLink,
   UINT * pnState,
   UINT stateMask = LIS_FOCUSED | LIS_ENABLED | LIS_VISITED
) const;
```

Parameters

iLink

The index of a link control item.

pnState

The value of the specified state item.

stateMask

Combination of flags describing which state item to get. For a list of values, see the description of the **state** member in the LITEM structure. Allowable items are identical to those allowed in **state**.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

Retrieves the value of the specified state item of a specific link control item. For more information, see the Win32 message LM_GETITEM in the *Platform SDK*.

See Also

Reference

CLinkCtrl Class
Hierarchy Chart
CLinkCtrl::SetItemState
CLinkCtrl::GetItem

Other Resources

CLinkCtrl Members

CLinkCtrl::GetItemUrl

Retrieves the URL represented by the link control item.

```
BOOL GetItemUrl(
   int iLink,
   CString& strUrl
) const;
BOOL GetItemUrl(
   int iLink,
   LPWSTR szUrl,
   UINT cchUrl
) const;
```

Parameters

iLink

The index of a link control item.

strUrl

A CStringT object containing the URL represented by the specified item

szUrl

A null-terminated string containing the URL represented by the specified item

cchUrl

The size in characters of the *szURL* buffer.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Note
This function also returns **FALSE** if the buffer of *szUrl or strUrl* is smaller than **MAX_LINKID_TEXT**.

Remarks

Retrieves the URL represented by the specified link control item. For more information, see the Win32 message LM_GETITEM in the *Platform SDK*.

See Also

Reference

CLinkCtrl Class
Hierarchy Chart
CLinkCtrl::SetItemUrl
CLinkCtrl::GetItem

Other Resources

CLinkCtrl Members

CLinkCtrl::HitTest

Determines if the user clicked the specified link.

```
BOOL HitTest(
   PLHITTESTINFO phti
) const;
```

Parameters

phti

Pointer to a **LHITTESTINFO** structure containing any information about the link the user clicked.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This member function implements the behavior of the Win32 message LM_HITTEST, as described in the *Platform SDK*.

See Also

Reference

CLinkCtrl Class

Hierarchy Chart

Other Resources

CLinkCtrl Members

CLinkCtrl::SetItem

Sets the states and attributes of a link control item.

```
BOOL SetItem(
   PLITEM pItem
);
```

Parameters

pItem

A pointer to a LITEM structure containing the information to set.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This member function implements the behavior of the Win32 message LM_SETITEM, as described in the *Platform SDK*.

See Also

Reference

CLinkCtrl Class
Hierarchy Chart
CLinkCtrl::GetItem
CLinkCtrl::SetItemID
CLinkCtrl::SetItemState
CLinkCtrl::SetItemUrl

Other Resources

CLinkCtrl Members

CLinkCtrl::SetItemID

Retrieves the ID of a link control item.

```
BOOL SetItemID(
   int iLink,
   LPCWSTR szID
);
```

Parameters

iLink

The index of a link control item.

szID

A null-terminated string containing the ID of the specified item.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

Sets the ID of a specific link control item. For more information, see the Win32 message LM_SETITEM in the *Platform SDK*.

See Also

Reference

CLinkCtrl Class
Hierarchy Chart
CLinkCtrl::GetItemID
CLinkCtrl::GetItem

Other Resources

CLinkCtrl Members

CLinkCtrl::SetItemState

Retrieves the state of the link control item.

```
BOOL SetItemState(
   int iLink,
   UINT state,
   UINT stateMask = LIS_FOCUSED | LIS_ENABLED | LIS_VISITED
);
```

Parameters

iLink

The index of a link control item.

pnState

The value of the specified state item being set.

stateMask

Combination of flags describing the state item being set. For a list of values, see the description of the **state** member in the LITEM structure. Allowable items are identical to those allowed in **state**.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

Sets the value of the specified state item of a specific link control item. For more information, see the Win32 message LM_SETITEM in the *Platform SDK*.

See Also

Reference

CLinkCtrl Class
Hierarchy Chart
CLinkCtrl::GetItemState
CLinkCtrl::GetItem

Other Resources

CLinkCtrl Members

CLinkCtrl::SetItemUrl

Sets the URL represented by the link control item.

```
BOOL SetItemUrl(
   int iLink,
   LPCWSTR szUrl
);
```

Parameters

iLink

The index of a link control item.

szUrl

A null-terminated string containing the URL represented by the specified item

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

Sets the URL represented by the specified link control item. For more information, see the Win32 message LM_SETITEM in the *Platform SDK*.

See Also

Reference

CLinkCtrl Class

Hierarchy Chart

Other Resources

CLinkCtrl Members

CList Class

Supports ordered lists of nonunique objects accessible sequentially or by value.

```
template< class TYPE, class ARG_TYPE = const TYPE& >
class CList : public CObject
```

Parameters

TYPE

Type of object stored in the list.

ARG _ TYPE

Type used to reference objects stored in the list. Can be a reference.

Remarks

CList lists behave like doubly-linked lists.

A variable of type **POSITION** is a key for the list. You can use a **POSITION** variable as an iterator to traverse a list sequentially and as a bookmark to hold a place. A position is not the same as an index, however.

Element insertion is very fast at the list head, at the tail, and at a known **POSITION**. A sequential search is necessary to look up an element by value or index. This search can be slow if the list is long.

If you need a dump of individual elements in the list, you must set the depth of the dump context to 1 or greater.

Certain member functions of this class call global helper functions that must be customized for most uses of the **CList** class. See Collection Class Helpers in the "Macros and Globals" section.

For more information on using **CList**, see the article Collections.

Example

```
// CList is a template class that takes two template arguments.
// The first argument is type stored internally by the list, the
// second argument is the type used in the arguments for the
// CList methods.

// This code defines a list of ints.
CList<int,int> myList;

// This code defines a list of CStrings
CList<CString,CString&> myList;

// This code defines a list of MYTYPEs,
//NOTE: MYTYPE could be any struct, class or type definition
CList<MYTYPE,MYTYPE&> myList;
```

Requirements

Header: afxtempl.h

See Also

Tasks

COLLECT Sample: Illustrates MFC Collection Classes

Reference

CObject Class

Hierarchy Chart

CMap Class

CArray Class

Other Resources

CList Members

CList Members

Base Class Members

COBJECT Members

CList Members

Construction

CList	Constructs an empty ordered list.

Head/Tail Access

GetHead	Returns the head element of the list (cannot be empty).
GetTail	Returns the tail element of the list (cannot be empty).

Operations

AddHead	Adds an element (or all the elements in another list) to the head of the list (makes a new head).
AddTail	Adds an element (or all the elements in another list) to the tail of the list (makes a new tail).
RemoveAll	Removes all the elements from this list.
RemoveHead	Removes the element from the head of the list.
RemoveTail	Removes the element from the tail of the list.

Iteration

GetHeadPosition	Returns the position of the head element of the list.
GetNext	Gets the next element for iterating.
GetPrev	Gets the previous element for iterating.
GetTailPosition	Returns the position of the tail element of the list.

Retrieval/Modification

GetAt	Gets the element at a given position.
RemoveAt	Removes an element from this list, specified by position.
SetAt	Sets the element at a given position.

Insertion

InsertAfter	Inserts a new element after a given position.
InsertBefore	Inserts a new element before a given position.

Searching

Find	Gets the position of an element specified by pointer value.
FindIndex	Gets the position of an element specified by a zero-based index.

Status

GetCount	Returns the number of elements in this list.
GetSize	Returns the number of elements in this list.
IsEmpty	Tests for the empty list condition (no elements).

See Also

Reference

CList Class

Hierarchy Chart

CList Member Functions

For information about the member functions in **CList**, see CList Members.

CList::AddHead

Adds a new element or list of elements to the head of this list.

```
POSITION AddHead(
      ARG_TYPE newElement
);
void AddHead(
   CList* pNewList
);
```

Parameters

ARG_TYPE

Template parameter specifying the type of the list element (can be a reference).

newElement

The new element.

pNewList

A pointer to another **CList** list. The elements in *pNewList* will be added to this list.

Return Value

The first version returns the **POSITION** value of the newly inserted element.

Remarks

The list can be empty before the operation.

Example

```
// Declarations of the variables used in the example
CList<CString,CString&> myList;
CList<CString,CString&> myList2;

// There are two versions of CList::AddHead: one adds a single
// element to the front of the list, the second adds another list
// to the front.

// This adds the string "ABC" to the front of myList.
// myList is a list of CStrings (ie defined as CList<CString,CString&>).
myList.AddHead(CString("ABC"));

// This adds the elements of myList2 to the front of myList.
myList.AddHead(&myList2);
```

See Also

Reference

CList Class
Hierarchy Chart
CList::GetHead
CList::RemoveHead

Other Resources

CList Members

CList::AddTail

Adds a new element or list of elements to the tail of this list.

```
POSITION AddTail(
      ARG_TYPE newElement
);
void AddTail(
   CList* pNewList
);
```

Parameters

ARG_TYPE

Template parameter specifying the type of the list element (can be a reference).

newElement

The element to be added to this list.

pNewList

A pointer to another **CList** list. The elements in *pNewList* will be added to this list.

Return Value

The first version returns the **POSITION** value of the newly inserted element.

Remarks

The list can be empty before the operation.

Example

```
// Define myList and myList2.
CList<CString,CString&> myList;
CList<CString,CString&> myList2;

// Add elements to the end of myList and myList2.
myList.AddTail(CString("A"));
myList.AddTail(CString("B"));
myList2.AddTail(CString("C"));
myList2.AddTail(CString("D"));

// There are two versions of CList::AddTail: one adds a single
// element to the end of the list, the second adds another list
// to the end.

// This adds the string "ABC" to the end of myList.
// myList is a list of CStrings (ie defined as CList<CString,CString&>).
myList.AddTail(CString("ABC"));
ASSERT(CString("ABC") == myList.GetTail());

// The adds the elements of myList2 to the end of myList.
myList.AddTail(&myList2);
```

See Also

Reference

CList Class
Hierarchy Chart
CObList::GetTail
CObList::RemoveTail

Other Resources

CList Members

CList::CList

Constructs an empty ordered list.

```
CList(
   INT_PTR nBlockSize = 10
);
```

Parameters

nBlockSize

The memory-allocation granularity for extending the list.

Remarks

As the list grows, memory is allocated in units of *nBlockSize* entries.

Example

```
// This code defines myList as a list of strings
// such that memory gets allocated in chunks of
// 16 strings.
CList<CString,CString&> myList(16);

// This code defines myList2 as a list of ints
// such that memory gets allocated in chunks of
// 128 ints.
CList<int,int> myList2(128);
```

See Also

Reference

CList Class

Hierarchy Chart

Other Resources

CList Members

CList::Find

Searches the list sequentially to find the first element matching the specified *searchValue*.

```
POSITION Find(
      ARG_TYPE searchValue,
   POSITION startAfter = NULL
) const;
```

Parameters

ARG_TYPE

Template parameter specifying the type of the list element (can be a reference).

searchValue

The value to be found in the list.

startAfter

The start position for the search. If no value is specified, the search begins with the head element.

Return Value

A **POSITION** value that can be used for iteration or object pointer retrieval; **NULL** if the object is not found.

Example

```
// Define myList.
CList<CString,CString&> myList;

// Add three elements to the list.
myList.AddHead(CString("XYZ"));
myList.AddHead(CString("ABC"));
myList.AddHead(CString("123"));

// Find a specific element.
POSITION pos = myList.Find(CString("XYZ"));
ASSERT(CString("XYZ") == myList.GetAt(pos));
```

See Also

Reference

CList Class
Hierarchy Chart
CList::GetNext
CList::GetPrev

Other Resources

CList Members

CList::FindIndex

Uses the value of *nIndex* as an index into the list.

```
POSITION FindIndex(
   INT_PTR nIndex
) const;
```

Parameters

nIndex

The zero-based index of the list element to be found.

Return Value

A **POSITION** value that can be used for iteration or object pointer retrieval; **NULL** if *nIndex* is negative or too large.

Remarks

It starts a sequential scan from the head of the list, stopping on the *n*th element.

Example

```
// Define myList.
CList<CString,CString&> myList;

// Add three elements to the list.
myList.AddTail(CString("XYZ"));
myList.AddTail(CString("ABC"));
myList.AddTail(CString("123"));

// Verify the first element (index 0).
ASSERT(CString("XYZ") == myList.GetAt(myList.FindIndex(0)));

// Verify the third element (index 2).
ASSERT(CString("123") == myList.GetAt(myList.FindIndex(2)));
```

See Also

Reference

CList Class
Hierarchy Chart
CObList::Find
CObList::GetNext
CObList::GetPrev

Other Resources

CList Members

CList::GetAt

Gets the list element at a given position.

```
TYPE& GetAt(
   POSITION position
);
const TYPE& GetAt(
   POSITION position
) const;
```

Parameters

TYPE

Template parameter specifying the type of object in the list.

position

The position in the list at which the element is to be returned.

Return Value

See the return value description for **GetHead**.

Remarks

GetAt returns the element (or a reference to the element) associated with a given position. It is not the same as an index, and you cannot operate on a **POSITION** value yourself. A variable of type **POSITION** is a key for the list.

You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

Example

See the example for CList::GetHeadPosition.

See Also

Reference

CList Class
Hierarchy Chart
CList::Find
CList::SetAt
CList::GetNext
CList::GetPrev
CList::GetHead

Other Resources

CList Members

CList::GetCount

Gets the number of elements in this list.

```
INT_PTR GetCount() const;
```

Return Value

An integer value containing the element count.

Remarks

Calling this method will generate the same result as the CList::GetSize method.

Example

See the example for CList::RemoveHead.

See Also

Reference

CList Class
Hierarchy Chart
CList::IsEmpty
CList::GetSize

Other Resources

CList Members

CList::GetHead

Gets the head element (or a reference to the head element) of this list.

```
const TYPE& GetHead( ) const;
TYPE& GetHead( );
```

Parameters

TYPE

Template parameter specifying the type of object in the list.

Return Value

If the list is **const**, **GetHead** returns a copy of the element at the head of the list. This allows the function to be used only on the right side of an assignment statement and protects the list from modification.

If the list is not **const**, **GetHead** returns a reference to the element at the head of the list. This allows the function to be used on either side of an assignment statement and thus allows the list entries to be modified.

Remarks

You must ensure that the list is not empty before calling **GetHead**. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use IsEmpty to verify that the list contains elements.

Example

```
// Define myList.
CList<CString,CString&> myList;

// Add an element to the front of the list.
myList.AddHead(CString("ABC"));

// Verify the element was added to the front of the list.
ASSERT(CString("ABC") == myList.GetHead());
```

See Also

Reference

CList Class
Hierarchy Chart
CList::GetTail
CList::GetTailPosition
CList::AddHead
CList::RemoveHead

Other Resources

CList Members

CList::GetHeadPosition

Gets the position of the head element of this list.

```
POSITION GetHeadPosition( ) const;
```

Return Value

A **POSITION** value that can be used for iteration or object pointer retrieval; **NULL** if the list is empty.

Example

```
// Define myList.
CList<CString,CString&> myList;

// Add an element to the front of the list.
myList.AddHead(CString("ABC"));

// Verify the element at the head position
// is the one added.
POSITION pos = myList.GetHeadPosition();
ASSERT(CString("ABC") == myList.GetAt(pos));
```

See Also

Reference

CList Class

Hierarchy Chart

CList::GetTailPosition

Other Resources

CList Members

CList::GetNext

Gets the list element identified by *rPosition*, then sets *rPosition* to the **POSITION** value of the next entry in the list.

```
TYPE& GetNext(
   POSITION& rPosition
);
const TYPE& GetNext(
   POSITION& rPosition
) const;
```

Parameters

TYPE

Template parameter specifying the type of the elements in the list.

rPosition

A reference to a **POSITION** value returned by a previous **GetNext**, GetHeadPosition, or other member function call.

Return Value

If the list is **const**, **GetNext** returns a copy of an element of the list. This allows the function to be used only on the right side of an assignment statement and protects the list from modification.

If the list is not **const**, **GetNext** returns a reference to an element of the list. This allows the function to be used on either side of an assignment statement and thus allows the list entries to be modified.

Remarks

You can use **GetNext** in a forward iteration loop if you establish the initial position with a call to **GetHeadPosition** or **Find**.

You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

If the retrieved element is the last in the list, then the new value of *rPosition* is set to **NULL**.

Example

```
// Define myList.
CList<CString,CString&> myList;

// Add two elements to the list.
myList.AddHead(CString("ABC"));
myList.AddHead(CString("123"));

// Dump the list elements to the debug window.
POSITION pos = myList.GetHeadPosition();
for (int i=0;i < myList.GetCount();i++)
{
   TRACE("%s\r\n", (LPCSTR) myList.GetNext(pos));
}
```

See Also

Reference

CList Class
Hierarchy Chart
CList::Find
CList::GetHeadPosition
CList::GetTailPosition
CList::GetPrev
CList::GetHead

Other Resources

CList Members

CList::GetPrev

Gets the list element identified by *rPosition*, then sets *rPosition* to the **POSITION** value of the previous entry in the list.

```
TYPE& GetPrev(
   POSITION& rPosition
);
const TYPE& GetPrev(
   POSITION& rPosition
) const;
```

Parameters

TYPE

Template parameter specifying the type of the elements in the list.

rPosition

A reference to a **POSITION** value returned by a previous **GetPrev** or other member function call.

Return Value

If the list is **const**, **GetPrev** returns a copy of the element at the head of the list. This allows the function to be used only on the right side of an assignment statement and protects the list from modification.

If the list is not **const**, **GetPrev** returns a reference to an element of the list. This allows the function to be used on either side of an assignment statement and thus allows the list entries to be modified.

Remarks

You can use **GetPrev** in a reverse iteration loop if you establish the initial position with a call to **GetTailPosition** or **Find**.

You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

If the retrieved element is the first in the list, then the new value of *rPosition* is set to **NULL**.

Example

```
// Define myList.
CList<CString,CString&> myList;

// Add two elements to the list.
myList.AddHead(CString("ABC"));
myList.AddHead(CString("123"));

// Dump the list elements to the debug window,
// in reverse order.
POSITION pos = myList.GetTailPosition();
for (int i=0;i < myList.GetCount();i++)
{
   TRACE("%s\r\n", (LPCSTR) myList.GetPrev(pos));
}
```

See Also

Reference

CList Class
Hierarchy Chart
CList::Find
CList::GetTailPosition
CList::GetHeadPosition
CList::GetNext
CList::GetHead

Other Resources

CList Members

CList::GetSize

Returns the number of list elements.

```
INT_PTR GetSize( ) const;
```

Return Value

The number of items in the list.

Remarks

Call this method to retrieve the number of elements in the list. Calling this method will generate the same result as the CList::GetCount method.

Example

```
// Define myList.
CList<CString,CString&> myList;

// Add two elements to the list.
myList.AddHead(CString("ABC"));
myList.AddHead(CString("123"));

// Remove the head element and verify the list.
// NOTE: once the head is removed, the number of
// elements in the list will be one.
CString strHead = myList.RemoveHead();
ASSERT((CString("123") == strHead) && (myList.GetSize() == 1) && (CString("ABC") == myList.
GetHead()));
```

See Also

Reference

CList Class

Hierarchy Chart

CList::GetCount

Other Resources

CList Members

CList::GetTail

Gets the **CObject** pointer that represents the tail element of this list.

```
TYPE& GetTail( );
const TYPE& GetTail() const;
```

Parameters

TYPE

Template parameter specifying the type of elements in the list.

Return Value

See the return value description for GetHead.

Remarks

You must ensure that the list is not empty before calling **GetTail**. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use IsEmpty to verify that the list contains elements.

Example

```
// Define myList.
CList<CString,CString&> myList;

// Add an element to the end of the list.
myList.AddTail(CString("ABC"));

// Verify the element was added to the end of the list.
ASSERT(CString("ABC") == myList.GetTail());
```

See Also

Reference

CList Class
Hierarchy Chart
CList::AddTail
CList::AddHead
CList::RemoveHead
CList::GetHead

Other Resources

CList Members

CList::GetTailPosition

Gets the position of the tail element of this list; **NULL** if the list is empty.

```
POSITION GetTailPosition( ) const;
```

Return Value

A **POSITION** value that can be used for iteration or object pointer retrieval; **NULL** if the list is empty.

Example

```
// Define myList.
CList<CString,CString&> myList;

// Add an element to the end of the list.
myList.AddTail(CString("ABC"));

// Verify the element at the end position
// is the one added.
POSITION pos = myList.GetTailPosition();
ASSERT(CString("ABC") == myList.GetAt(pos));
```

See Also

Reference

CList Class

Hierarchy Chart

CList::GetHeadPosition

CList::GetTail

Other Resources

CList Members

CList::InsertAfter

Adds an element to this list after the element at the specified position.

```
POSITION InsertAfter(
   POSITION position,
      ARG_TYPE newElement
);
```

Parameters

position

A **POSITION** value returned by a previous **GetNext**, **GetPrev**, or **Find** member function call.

ARG_TYPE

Template parameter specifying the type of the list element.

newElement

The element to be added to this list.

Return Value

A **POSITION** value that can be used for iteration or list element retrieval.

Example

```
// Define myList.
CList<CString,CString&> myList;

// Add three elements to the list.
POSITION pos = myList.AddHead(CString("XYZ"));
pos = myList.InsertAfter(pos, CString("ABC"));
pos = myList.InsertAfter(pos, CString("123"));

// Verify the tail element is what's expected.
ASSERT(CString("123") == myList.GetTail());
```

See Also

Reference

CList Class
Hierarchy Chart
CList::Find
CList::InsertBefore

Other Resources

CList Members

CList::InsertBefore

Adds an element to this list before the element at the specified position.

```
POSITION InsertBefore(
   POSITION position,
      ARG_TYPE newElement
);
```

Parameters

position

A **POSITION** value returned by a previous **GetNext**, **GetPrev**, or **Find** member function call.

ARG_TYPE

Template parameter specifying the type of the list element (can be a reference).

newElement

The element to be added to this list.

Return Value

A **POSITION** value that can be used for iteration or list element retrieval.

Remarks

If *position* is **NULL**, the element is inserted at the head of the list.

Example

```
// Define myList.
CList<CString,CString&> myList;

// Add three elements to the list.
POSITION pos = myList.AddHead(CString("XYZ"));
pos = myList.InsertBefore(pos, CString("ABC"));
pos = myList.InsertBefore(pos, CString("123"));

// Verify the head element is what's expected.
ASSERT(CString("123") == myList.GetHead());
```

See Also

Reference

CList Class
Hierarchy Chart
CList::Find
CList::InsertAfter

Other Resources

CList Members

CList::IsEmpty

Indicates whether this list contains no elements.

```
BOOL IsEmpty( ) const;
```

Return Value

Nonzero if this list is empty; otherwise 0.

Example

```
// Define myList.
CList<CString,CString&> myList;

// Add three elements to the list.
myList.AddTail(CString("XYZ"));
myList.AddTail(CString("ABC"));
myList.AddTail(CString("123"));

// Remove the head element until the list is empty.
CString str;
while (!myList.IsEmpty())
{
   str = myList.RemoveHead();
   TRACE("%s\r\n", (LPCSTR) str);
}
```

See Also

Reference

CList Class

Hierarchy Chart

CList::GetCount

Other Resources

CList Members

CList::RemoveAll

Removes all the elements from this list and frees the associated memory.

```
void RemoveAll( );
```

Remarks

No error is generated if the list is already empty.

Example

```
// Define myList.
CList<CString,CString&> myList;

// Add three elements to the list.
myList.AddTail(CString("XYZ"));
myList.AddTail(CString("ABC"));
myList.AddTail(CString("123"));

// Remove all of the elements in the list.
myList.RemoveAll();

// Verify the list is empty.
ASSERT(myList.IsEmpty());
```

See Also

Reference

CList Class

Hierarchy Chart

CList::RemoveAt

Other Resources

CList Members

CList::RemoveAt

Removes the specified element from this list.

```
void RemoveAt(
   POSITION position
);
```

Parameters

position

The position of the element to be removed from the list.

Remarks

You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

Example

```
// Define myList.
CList<CString,CString&> myList;

// Add three elements to the list.
myList.AddTail(CString("XYZ"));
myList.AddTail(CString("ABC"));
myList.AddTail(CString("123"));

// Remove CString("ABC") from the list.
myList.RemoveAt(myList.FindIndex(1));

// Verify CString("ABC") is not in the list.
ASSERT(myList.Find(CString("ABC")) == NULL);
```

See Also

Reference

CList Class

Hierarchy Chart

CList::RemoveAll

Other Resources

CList Members

CList::RemoveHead

Removes the element from the head of the list and returns a pointer to it.

```
TYPE RemoveHead( );
```

Parameters

TYPE

Template parameter specifying the type of elements in the list.

Return Value

The element previously at the head of the list.

Remarks

You must ensure that the list is not empty before calling **RemoveHead**. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use IsEmpty to verify that the list contains elements.

Example

```
// Define myList.
CList<CString,CString&> myList;

// Add two elements to the list.
myList.AddHead(CString("ABC"));
myList.AddHead(CString("123"));

// Remove the head element and verify the list.
// NOTE: once the head is removed, the number of
// elements in the list will be one.
CString strHead = myList.RemoveHead();
ASSERT((CString("123") == strHead) && (myList.GetCount() == 1) && (CString("ABC") == myList
.GetHead()));
```

See Also

Reference

CList Class
Hierarchy Chart
CList::GetHead
CList::AddHead

Other Resources

CList Members

CList::RemoveTail

Removes the element from the tail of the list and returns a pointer to it.

```
TYPE RemoveTail( );
```

Parameters

TYPE

Template parameter specifying the type of elements in the list.

Return Value

The element that was at the tail of the list.

Remarks

You must ensure that the list is not empty before calling **RemoveTail**. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use IsEmpty to verify that the list contains elements.

Example

```
// Define myList.
CList<CString,CString&> myList;

// Add two elements to the list.
myList.AddTail(CString("ABC"));
myList.AddTail(CString("123"));

// Remove the tail element and verify the list.
// NOTE: once the tail is removed, the number of
// elements in the list will be one.
CString strTail = myList.RemoveTail();
ASSERT((CString("123") == strTail) && (myList.GetCount() == 1) && (CString("ABC") == myList
.GetTail()));
```

See Also

Reference

CList Class
Hierarchy Chart
CList::GetTail
CList::AddTail

Other Resources

CList Members

CList::SetAt

A variable of type **POSITION** is a key for the list.

```
void SetAt(
   POSITION pos,
      ARG_TYPE newElement
);
```

Parameters

pos

The **POSITION** of the element to be set.

ARG_TYPE

Template parameter specifying the type of the list element (can be a reference).

newElement

The element to be added to the list.

Remarks

It is not the same as an index, and you cannot operate on a **POSITION** value yourself. **SetAt** writes the element to the specified position in the list.

You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

Example

```
// Define myList.
CList<CString,CString&> myList;

// Add three elements to the list.
myList.AddTail(CString("XYZ"));
myList.AddTail(CString("ABC"));
myList.AddTail(CString("123"));

// Replace CString("ABC") with CString("CBA")
POSITION pos = myList.Find(CString("ABC"));
myList.SetAt(pos, CString("CBA"));

// Verify CString("ABC") is not in the list.
ASSERT(myList.Find(CString("ABC")) == NULL);
```

See Also

Reference

CList Class
Hierarchy Chart
CList::Find
CList::GetAt
CList::GetNext
CList::GetPrev

Other Resources

CList Members

CListBox Class

Provides the functionality of a Windows list box.

```
class CListBox : public CWnd
```

Remarks

A list box displays a list of items, such as filenames, that the user can view and select.

In a single-selection list box, the user can select only one item. In a multiple-selection list box, a range of items can be selected. When the user selects an item, it is highlighted and the list box sends a notification message to the parent window.

You can create a list box either from a dialog template or directly in your code. To create it directly, construct the **CListBox** object, then call the Create member function to create the Windows list-box control and attach it to the **CListBox** object. To use a list box in a dialog template, declare a list-box variable in your dialog box class, then use **DDX_Control** in your dialog box class's **DoDataExchange** function to connect the member variable to the control. (this is done for you automatically when you add a control variable to your dialog box class.)

Construction can be a one-step process in a class derived from **CListBox**. Write a constructor for the derived class and call **Create** from within the constructor.

If you want to handle Windows notification messages sent by a list box to its parent (usually a class derived from CDialog), add a message-map entry and message-handler member function to the parent class for each message.

Each message-map entry takes the following form:

```
ON_Notification( id, memberFxn )
```

where *id* specifies the child window ID of the list-box control sending the notification and *memberFxn* is the name of the parent member function you have written to handle the notification.

The parent's function prototype is as follows:

```
afx_msg void memberFxn( );
```

Following is a list of potential message-map entries and a description of the cases in which they would be sent to the parent:

- **ON_LBN_DBLCLK** The user double-clicks a string in a list box. Only a list box that has the LBS_NOTIFY style will send this notification message.
- **ON_LBN_ERRSPACE** The list box cannot allocate enough memory to meet the request.
- **ON_LBN_KILLFOCUS** The list box is losing the input focus.
- **ON_LBN_SELCANCEL** The current list-box selection is canceled. This message is only sent when a list box has the **LBS_NOTIFY** style.
- **ON_LBN_SELCHANGE** The selection in the list box has changed. This notification is not sent if the selection is changed by the CListBox::SetCurSel member function. This notification applies only to a list box that has the **LBS_NOTIFY** style. The **LBN_SELCHANGE** notification message is sent for a multiple-selection list box whenever the user presses an arrow key, even if the selection does not change.
- **ON_LBN_SETFOCUS** The list box is receiving the input focus.
- **ON_WM_CHARTOITEM** An owner-draw list box that has no strings receives a **WM_CHAR** message.
- **ON_WM_VKEYTOITEM** A list box with the **LBS_WANTKEYBOARDINPUT** style receives a **WM_KEYDOWN** message.

If you create a **CListBox** object within a dialog box (through a dialog resource), the **CListBox** object is automatically destroyed when the user closes the dialog box.

If you create a **CListBox** object within a window, you may need to destroy the **CListBox** object. If you create the **CListBox** object on the stack, it is destroyed automatically. If you create the **CListBox** object on the heap by using the **new** function, you must call **delete** on the object to destroy it when the user closes the parent window.

If you allocate any memory in the **CListBox** object, override the **CListBox** destructor to dispose of the allocation.

Requirements

Header: afxwin.h

See Also

Tasks

CTRLTEST Sample: Implements Custom Controls

Reference

CWnd Class

Hierarchy Chart

CWnd Class

CButton Class

CComboBox Class

CEdit Class

CScrollBar Class

CStatic Class

Other Resources

CListBox Members

CListBox Members

Base Class Members

Construction Initialization

General Operations Single-Selection Operations

Multiple-Selection Operations

String Operations

Overridables

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Construction

CListBox	Constructs a **CListBox** object.

Initialization

Create	Creates the Windows list box and attaches it to the **CListBox** object.
InitStorage	Preallocates blocks of memory for list box items and strings.

General Operations

GetCount	Returns the number of strings in a list box.
GetHorizontalExtent	Returns the width in pixels that a list box can be scrolled horizontally.
GetItemData	Returns the 32-bit value associated with the list-box item.
GetItemDataPtr	Returns a pointer to a list-box item.
GetItemHeight	Determines the height of items in a list box.
GetItemRect	Returns the bounding rectangle of the list-box item as it is currently displayed.
GetListBoxInfo	Retrieves the number of items per column.
GetLocale	Retrieves the locale identifier for a list box.
GetSel	Returns the selection state of a list-box item.
GetText	Copies a list-box item into a buffer.
GetTextLen	Returns the length in bytes of a list-box item.
GetTopIndex	Returns the index of the first visible string in a list box.
ItemFromPoint	Returns the index of the list-box item nearest a point.
SetColumnWidth	Sets the column width of a multicolumn list box.

SetHorizontalExtent	Sets the width in pixels that a list box can be scrolled horizontally.
SetItemData	Sets the 32-bit value associated with the list-box item.
SetItemDataPtr	Sets a pointer to the list-box item.
SetItemHeight	Sets the height of items in a list box.
SetLocale	Sets the locale identifier for a list box.
SetTabStops	Sets the tab-stop positions in a list box.
SetTopIndex	Sets the zero-based index of the first visible string in a list box.

Single-Selection Operations

GetCurSel	Returns the zero-based index of the currently selected string in a list box.
SetCurSel	Selects a list-box string.

Multiple-Selection Operations

GetAnchorIndex	Retrieves the zero-based index of the current anchor item in a list box.
GetCaretIndex	Determines the index of the item that has the focus rectangle in a multiple-selection list box.
GetSelCount	Returns the number of strings currently selected in a multiple-selection list box.
GetSelItems	Returns the indices of the strings currently selected in a list box.
SelItemRange	Selects or deselects a range of strings in a multiple-selection list box.
SetAnchorIndex	Sets the anchor in a multiple-selection list box to begin an extended selection.
SetCaretIndex	Sets the focus rectangle to the item at the specified index in a multiple-selection list box.
SetSel	Selects or deselects a list-box item in a multiple-selection list box.

String Operations

AddString	Adds a string to a list box.
DeleteString	Deletes a string from a list box.
Dir	Adds filenames, drives, or both from the current directory to a list box.
FindString	Searches for a string in a list box.
FindStringExact	Finds the first list-box string that matches a specified string.
InsertString	Inserts a string at a specific location in a list box.
ResetContent	Clears all the entries from a list box.
SelectString	Searches for and selects a string in a single-selection list box.

Overridables

CharToItem	Override to provide custom **WM_CHAR** handling for owner-draw list boxes which don't have strings.
CompareItem	Called by the framework to determine the position of a new item in a sorted owner-draw list box.
DeleteItem	Called by the framework when the user deletes an item from an owner-draw list box.
DrawItem	Called by the framework when a visual aspect of an owner-draw list box changes.
MeasureItem	Called by the framework when an owner-draw list box is created to determine list-box dimensions.
VKeyToItem	Override to provide custom **WM_KEYDOWN** handling for list boxes with the **LBS_WANTKEYBOARDINPUT** style set.

See Also

Reference

CListBox Class

Hierarchy Chart

CListBox Member Functions

For information about the member functions in **CListBox**, see CListBox Members.

CListBox::AddString

Adds a string to a list box.

```
int AddString(
   LPCTSTR lpszItem
);
```

Parameters

lpszItem

Points to the null-terminated string that is to be added.

Return Value

The zero-based index to the string in the list box. The return value is **LB_ERR** if an error occurs; the return value is **LB_ERRSPACE** if insufficient space is available to store the new string.

Remarks

If the list box was not created with the LBS_SORT style, the string is added to the end of the list. Otherwise, the string is inserted into the list, and the list is sorted. If the list box was created with the **LBS_SORT** style but not the LBS_HASSTRINGS style, the framework sorts the list by one or more calls to the **CompareItem** member function.

Use InsertString to insert a string into a specific location within the list box.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Add 10 items to the list box.
CString str;
for (int i=0;i < 10;i++)
{
   str.Format(_T("item string %d"), i);
   pmyListBox->AddString( str );
}
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::InsertString
CListBox::CompareItem
LB_ADDSTRING

Other Resources

CListBox Members

CListBox::CharToItem

Called by the framework when the list box's parent window receives a **WM_CHARTOITEM** message from the list box.

```
virtual int CharToItem(
   UINT nKey,
   UINT nIndex
);
```

Parameters

nKey

The ANSI code of the character the user typed.

nIndex

The current position of the list-box caret.

Return Value

Returns – 1 or – 2 for no further action or a nonnegative number to specify an index of a list-box item on which to perform the default action for the keystroke. The default implementation returns – 1.

Remarks

The **WM_CHARTOITEM** message is sent by the list box when it receives a **WM_CHAR** message, but only if the list box meets all of these criteria:

- Is an owner-draw list box.
- Does not have the LBS_HASSTRINGS style set.
- Has at least one item.

You should never call this function yourself. Override this function to provide your own custom handling of keyboard messages.

In your override, you must return a value to tell the framework what action you performed. A return value of – 1 or – 2 indicates that you handled all aspects of selecting the item and requires no further action by the list box. Before returning – 1 or – 2, you could set the selection or move the caret or both. To set the selection, use SetCurSel or SetSel. To move the caret, use SetCaretIndex.

A return value of 0 or greater specifies the index of an item in the list box and indicates that the list box should perform the default action for the keystroke on the given item.

Example

```
// CMyListBox is my owner-drawn list box derived from CListBox. This
// example moves the caret down one item on a numeric key and up one item
// on an alphabetic key. The list box control was created with the
// following code:
//   pmyListBox->Create(
//      WS_CHILD|WS_VISIBLE|WS_BORDER|WS_HSCROLL|WS_VSCROLL|
//      LBS_SORT|LBS_MULTIPLESEL|LBS_OWNERDRAWVARIABLE|LBS_WANTKEYBOARDINPUT,
//      CRect(10,10,200,200), pParentWnd, 1);
//
int CMyListBox::CharToItem(UINT nKey, UINT nIndex)
{
   // On a numeric key, move the caret up one item.
   if (isdigit(nKey) && (nIndex > 0))
      SetCaretIndex(nIndex-1);

   // On an alphabetic key, move the caret down one item.
   else if (isalpha(nKey) && (nIndex < GetCount()))
      SetCaretIndex(nIndex+1);
```

```
    // Do not perform any default processing.
    return -1;
}
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::VKeyToItem
CListBox::SetCurSel
CListBox::SetSel
CListBox::SetCaretIndex
WM_CHARTOITEM

Other Resources

CListBox Members

CListBox::CListBox

Constructs a **CListBox** object.

```
CListBox( );
```

Remarks

You construct a **CListBox** object in two steps. First, call the constructor **ClistBox** and then call **Create**, which initializes the Windows list box and attaches it to the **CListBox**.

Example

```
// Declare a local CListBox object.
CListBox myListBox;

// Declare a dynamic CListBox object.
CListBox* pmyListBox = new CListBox;
```

See Also

Reference

CListBox Class

Hierarchy Chart

CListBox::Create

Other Resources

CListBox Members

CListBox::CompareItem

Called by the framework to determine the relative position of a new item in a sorted owner-draw list box.

```
virtual int CompareItem(
   LPCOMPAREITEMSTRUCT lpCompareItemStruct
);
```

Parameters

lpCompareItemStruct

A long pointer to a **COMPAREITEMSTRUCT** structure.

Return Value

Indicates the relative position of the two items described in the COMPAREITEMSTRUCT structure. It may be any of the following values:

Value	Meaning
–1	Item 1 sorts before item 2.
0	Item 1 and item 2 sort the same.
1	Item 1 sorts after item 2.

See CWnd::OnCompareItem for a description of the **COMPAREITEMSTRUCT** structure.

Remarks

By default, this member function does nothing. If you create an owner-draw list box with the **LBS_SORT** style, you must override this member function to assist the framework in sorting new items added to the list box.

Example

```
// CMyListBox is my owner-drawn list box derived from CListBox. This
// example compares two items using strcmp to sort items in reverse
// alphabetical order. The list box control was created with the
// following code:
//   pmyListBox->Create(
//      WS_CHILD|WS_VISIBLE|WS_BORDER|WS_HSCROLL|WS_VSCROLL|
//      LBS_SORT|LBS_MULTIPLESEL|LBS_OWNERDRAWVARIABLE,
//      myRect, pParentWnd, 1);
//
int CMyListBox::CompareItem(LPCOMPAREITEMSTRUCT lpCompareItemStruct)
{
   ASSERT(lpCompareItemStruct->CtlType == ODT_LISTBOX);
   LPCTSTR lpszText1 = (LPCTSTR) lpCompareItemStruct->itemData1;
   ASSERT(lpszText1 != NULL);
   LPCTSTR lpszText2 = (LPCTSTR) lpCompareItemStruct->itemData2;
   ASSERT(lpszText2 != NULL);

   return strcmp(lpszText2, lpszText1);
}
```

See Also

Reference

CListBox Class
Hierarchy Chart
WM_COMPAREITEM
CWnd::OnCompareItem
CListBox::DrawItem
CListBox::MeasureItem

CListBox::DeleteItem

Other Resources

CListBox Members

CListBox::Create

Creates the Windows list box and attaches it to the **CListBox** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the style of the list box. Apply any combination of list-box styles to the box.

rect

Specifies the list-box size and position. Can be either a **CRect** object or a **RECT** structure.

pParentWnd

Specifies the list box's parent window (usually a **CDialog** object). It must not be **NULL**.

nID

Specifies the list box's control ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

You construct a **CListBox** object in two steps. First, call the constructor and then call **Create**, which initializes the Windows list box and attaches it to the **CListBox** object.

When **Create** executes, Windows sends the WM_NCCREATE, WM_CREATE, WM_NCCALCSIZE, and WM_GETMINMAXINFO messages to the list-box control.

These messages are handled by default by the OnNcCreate, OnCreate, OnNcCalcSize, and OnGetMinMaxInfo member functions in the **CWnd** base class. To extend the default message handling, derive a class from **CListBox**, add a message map to the new class, and override the preceding message-handler member functions. Override **OnCreate**, for example, to perform needed initialization for a new class.

Apply the following window styles to a list-box control.

- **WS_CHILD** Always
- **WS_VISIBLE** Usually
- **WS_DISABLED** Rarely
- **WS_VSCROLL** To add a vertical scroll bar
- **WS_HSCROLL** To add a horizontal scroll bar
- **WS_GROUP** To group controls
- **WS_TABSTOP** To allow tabbing to this control

Example

```
// pParentWnd is an external pointer to the parent window.
extern CWnd* pParentWnd;
// The pointer to my list box.
extern CListBox* pmyListBox;
```

```
pmyListBox->Create(WS_CHILD|WS_VISIBLE|LBS_STANDARD|WS_HSCROLL,
   CRect(10,10,200,200), pParentWnd, 1);
```

See Also

Reference

CListBox Class

Hierarchy Chart

CListBox::CListBox

Other Resources

CListBox Members

CListBox::DeleteItem

Called by the framework when the user deletes an item from an owner-draw **CListBox** object or destroys the list box.

```
virtual void DeleteItem(
   LPDELETEITEMSTRUCT lpDeleteItemStruct
);
```

Parameters

lpDeleteItemStruct

A long pointer to a Windows DELETEITEMSTRUCT structure that contains information about the deleted item.

Remarks

The default implementation of this function does nothing. Override this function to redraw an owner-draw list box as needed.

See CWnd::OnDeleteItem for a description of the **DELETEITEMSTRUCT** structure.

Example

```
// CMyListBox is my owner-drawn list box derived from CListBox. This
// example simply frees the item's text. The list box control was created
// with the following code:
//   pmyListBox->Create(
//      WS_CHILD|WS_VISIBLE|WS_BORDER|WS_HSCROLL|WS_VSCROLL|
//      LBS_SORT|LBS_MULTIPLESEL|LBS_OWNERDRAWVARIABLE,
//      myRect, pParentWnd, 1);
//
void CMyListBox::DeleteItem(LPDELETEITEMSTRUCT lpDeleteItemStruct)
{
   ASSERT(lpDeleteItemStruct->CtlType == ODT_LISTBOX);
   LPVOID lpszText = (LPVOID) lpDeleteItemStruct->itemData;
   ASSERT(lpszText != NULL);

   free(lpszText);
}
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::CompareItem
CWnd::OnDeleteItem
CListBox::DrawItem
CListBox::MeasureItem
DeleteItem

Other Resources

CListBox Members

CListBox::DeleteString

Deletes the item in position *nIndex* from the list box.

```
int DeleteString(
   UINT nIndex
);
```

Parameters

nIndex

Specifies the zero-based index of the string to be deleted.

Return Value

A count of the strings remaining in the list. The return value is **LB_ERR** if *nIndex* specifies an index greater than the number of items in the list.

Remarks

All items following *nIndex* now move down one position. For example, if a list box contains two items, deleting the first item will cause the remaining item to now be in the first position. *nIndex*=0 for the item in the first position.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Delete every other item from the list box.
for (int i=0;i < pmyListBox->GetCount();i++)
{
   pmyListBox->DeleteString( i );
}
```

See Also

Reference

CListBox Class
Hierarchy Chart
LB_DELETESTRING
CListBox::AddString
CListBox::InsertString

Other Resources

CListBox Members

CListBox::Dir

Adds a list of filenames, drives, or both to a list box.

```
int Dir(
   UINT attr,
   LPCTSTR lpszWildCard
);
```

Parameters

attr

Can be any combination of the **enum** values described in **CFile::GetStatus**, or any combination of the following values:

Value	Meaning
0x0000	File can be read from or written to.
0x0001	File can be read from but not written to.
0x0002	File is hidden and does not appear in a directory listing.
0x0004	File is a system file.
0x0010	The name specified by *lpszWildCard* specifies a directory.
0x0020	File has been archived.
0x4000	Include all drives that match the name specified by *lpszWildCard*.
0x8000	Exclusive flag. If the exclusive flag is set, only files of the specified type are listed. Otherwise, files of the specified type are listed in addition to "normal" files.

lpszWildCard

Points to a file-specification string. The string can contain wildcards (for example, *.*).

Return Value

The zero-based index of the last filename added to the list. The return value is **LB_ERR** if an error occurs; the return value is **LB_ERRSPACE** if insufficient space is available to store the new strings.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Add all the files and directories in the windows directory.
TCHAR lpszWinPath[MAX_PATH], lpszOldPath[MAX_PATH];
::GetWindowsDirectory(lpszWinPath, MAX_PATH);

::GetCurrentDirectory(MAX_PATH, lpszOldPath);
```

```
::SetCurrentDirectory(lpszWinPath);

pmyListBox->ResetContent();
pmyListBox->Dir(DDL_READWRITE|DDL_DIRECTORY, _T("*.*"));

::SetCurrentDirectory(lpszOldPath);
```

See Also

Reference

CListBox Class
Hierarchy Chart
CWnd::DlgDirList
LB_DIR
CFile::GetStatus

Other Resources

CListBox Members

CListBox::DrawItem

Called by the framework when a visual aspect of an owner-draw list box changes.

```
virtual void DrawItem(
   LPDRAWITEMSTRUCT lpDrawItemStruct
);
```

Parameters

lpDrawItemStruct

A long pointer to a DRAWITEMSTRUCT structure that contains information about the type of drawing required.

Remarks

The **itemAction** and **itemState** members of the **DRAWITEMSTRUCT** structure define the drawing action that is to be performed.

By default, this member function does nothing. Override this member function to implement drawing for an owner-draw **CListBox** object. The application should restore all graphics device interface (GDI) objects selected for the display context supplied in *lpDrawItemStruct* before this member function terminates.

See CWnd::OnDrawItem for a description of the **DRAWITEMSTRUCT** structure.

Example

```
// CMyListBox is my owner-drawn list box derived from CListBox. This
// example draws an item's text centered vertically and horizontally. The
// list box control was created with the following code:
//   pmyListBox->Create(
//      WS_CHILD|WS_VISIBLE|WS_BORDER|WS_HSCROLL|WS_VSCROLL|
//      LBS_SORT|LBS_MULTIPLESEL|LBS_OWNERDRAWVARIABLE,
//      myRect, pParentWnd, 1);
//
void CMyListBox::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
   ASSERT(lpDrawItemStruct->CtlType == ODT_LISTBOX);
   LPCTSTR lpszText = (LPCTSTR) lpDrawItemStruct->itemData;
   ASSERT(lpszText != NULL);
   CDC dc;

   dc.Attach(lpDrawItemStruct->hDC);

   // Save these value to restore them when done drawing.
   COLORREF crOldTextColor = dc.GetTextColor();
   COLORREF crOldBkColor = dc.GetBkColor();

   // If this item is selected, set the background color
   // and the text color to appropriate values. Also, erase
   // rect by filling it with the background color.
   if ((lpDrawItemStruct->itemAction | ODA_SELECT) &&
      (lpDrawItemStruct->itemState & ODS_SELECTED))
   {
      dc.SetTextColor(::GetSysColor(COLOR_HIGHLIGHTTEXT));
      dc.SetBkColor(::GetSysColor(COLOR_HIGHLIGHT));
      dc.FillSolidRect(&lpDrawItemStruct->rcItem,
         ::GetSysColor(COLOR_HIGHLIGHT));
   }
   else
      dc.FillSolidRect(&lpDrawItemStruct->rcItem, crOldBkColor);

   // If this item has the focus, draw a red frame around the
   // item's rect.
   if ((lpDrawItemStruct->itemAction | ODA_FOCUS) &&
      (lpDrawItemStruct->itemState & ODS_FOCUS))
```

```
    {
       CBrush br(RGB(255, 0, 0));
       dc.FrameRect(&lpDrawItemStruct->rcItem, &br);
    }

    // Draw the text.
    dc.DrawText(
       lpszText,
       strlen(lpszText),
       &lpDrawItemStruct->rcItem,
       DT_CENTER|DT_SINGLELINE|DT_VCENTER);

    // Reset the background color and the text color back to their
    // original values.
    dc.SetTextColor(crOldTextColor);
    dc.SetBkColor(crOldBkColor);

    dc.Detach();
 }
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::CompareItem
CWnd::OnDrawItem
WM_DRAWITEM
CListBox::MeasureItem
CListBox::DeleteItem

Other Resources

CListBox Members

CListBox::FindString

Finds the first string in a list box that contains the specified prefix without changing the list-box selection.

```
int FindString(
   int nStartAfter,
   LPCTSTR lpszItem
) const;
```

Parameters

nStartAfter

Contains the zero-based index of the item before the first item to be searched. When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by *nStartAfter*. If *nStartAfter* is –1, the entire list box is searched from the beginning.

lpszItem

Points to the null-terminated string that contains the prefix to search for. The search is case independent, so this string may contain any combination of uppercase and lowercase letters.

Return Value

The zero-based index of the matching item, or **LB_ERR** if the search was unsuccessful.

Remarks

Use the SelectString member function to both find and select a string.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;
// The string to match.
extern LPCTSTR lpszmyString;

// Delete all items that begin with the specified string.
int nIndex = 0;
while ((nIndex=pmyListBox->FindString(nIndex, lpszmyString)) != LB_ERR)
{
   pmyListBox->DeleteString( nIndex );
}
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::SelectString
CListBox::AddString
CListBox::InsertString
LB_FINDSTRING

Other Resources

CListBox Members

CListBox::FindStringExact

Finds the first list-box string that matches the string specified in *lpszFind*.

```
int FindStringExact(
   int nIndexStart,
   LPCTSTR lpszFind
) const;
```

Parameters

nIndexStart

Specifies the zero-based index of the item before the first item to be searched. When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by *nIndexStart*. If *nIndexStart* is –1, the entire list box is searched from the beginning.

lpszFind

Points to the null-terminated string to search for. This string can contain a complete filename, including the extension. The search is not case sensitive, so the string can contain any combination of uppercase and lowercase letters.

Return Value

The index of the matching item, or **LB_ERR** if the search was unsuccessful.

Remarks

If the list box was created with an owner-draw style but without the LBS_HASSTRINGS style, the **FindStringExact** member function attempts to match the doubleword value against the value of *lpszFind*.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;
// The string to match.
extern LPCTSTR lpszmyString;

// Delete all items that exactly match the specified string.
int nIndex = 0;
while ((nIndex=pmyListBox->FindStringExact(nIndex, lpszmyString)) != LB_ERR)
{
   pmyListBox->DeleteString( nIndex );
}
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::FindString
LB_FINDSTRING
LB_FINDSTRINGEXACT

Other Resources

CListBox Members

CListBox::GetAnchorIndex

Retrieves the zero-based index of the current anchor item in the list box.

```
int GetAnchorIndex( ) const;
```

Return Value

The index of the current anchor item, if successful; otherwise **LB_ERR**.

Remarks

In a multiple-selection list box, the anchor item is the first or last item in a block of contiguous selected items.

Example

See the example for CListBox::SetAnchorIndex.

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::SetAnchorIndex

Other Resources

CListBox Members

CListBox::GetCaretIndex

Determines the index of the item that has the focus rectangle in a multiple-selection list box.

```
int GetCaretIndex( ) const;
```

Return Value

The zero-based index of the item that has the focus rectangle in a list box. If the list box is a single-selection list box, the return value is the index of the item that is selected, if any.

Remarks

The item may or may not be selected.

Example

See the example for CListBox::SetCaretIndex.

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::SetCaretIndex
LB_GETCARETINDEX

Other Resources

CListBox Members

CListBox::GetCount

Retrieves the number of items in a list box.

```
int GetCount( ) const;
```

Return Value

The number of items in the list box, or **LB_ERR** if an error occurs.

Remarks

The returned count is one greater than the index value of the last item (the index is zero-based).

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Add 10 items to the list box.
CString str;
for (int i=0;i < 10;i++)
{
   str.Format(_T("item %d"), i);
   pmyListBox->AddString( str );
}

// Verify that 10 items were added to the list box.
ASSERT(pmyListBox->GetCount() == 10);
```

See Also

Reference

CListBox Class

Hierarchy Chart

LB_GETCOUNT

Other Resources

CListBox Members

CListBox::GetCurSel

Retrieves the zero-based index of the currently selected item, if any, in a single-selection list box.

```
int GetCurSel( ) const;
```

Return Value

The zero-based index of the currently selected item. It is **LB_ERR** if no item is currently selected or if the list box is a multiple-selection list box.

Remarks

GetCurSel should not be called for a multiple-selection list box.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Select the next item of the currently selected one.
int nIndex = pmyListBox->GetCurSel();
int nCount = pmyListBox->GetCount();
if ((nIndex != LB_ERR) && (nCount > 1))
{
   if (++nIndex < nCount)
      pmyListBox->SetCurSel(nIndex);
   else
      pmyListBox->SetCurSel(0);
}
```

See Also

Reference

CListBox Class
Hierarchy Chart
LB_GETCURSEL
CListBox::SetCurSel

Other Resources

CListBox Members

CListBox::GetHorizontalExtent

Retrieves from the list box the width in pixels by which it can be scrolled horizontally.

```
int GetHorizontalExtent( ) const;
```

Return Value

The scrollable width of the list box, in pixels.

Remarks

This is applicable only if the list box has a horizontal scroll bar.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Find the longest string in the list box.
CString str;
CSize   sz;
int     dx=0;
CDC*    pDC = pmyListBox->GetDC();
for (int i=0;i < pmyListBox->GetCount();i++)
{
   pmyListBox->GetText( i, str );
   sz = pDC->GetTextExtent(str);

   if (sz.cx > dx)
      dx = sz.cx;
}
pmyListBox->ReleaseDC(pDC);

// Set the horizontal extent only if the current extent is not large enough.
if (pmyListBox->GetHorizontalExtent() < dx)
{
   pmyListBox->SetHorizontalExtent(dx);
   ASSERT(pmyListBox->GetHorizontalExtent() == dx);
}
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::SetHorizontalExtent
LB_GETHORIZONTALEXTENT

Other Resources

CListBox Members

CListBox::GetItemData

Retrieves the application-supplied doubleword value associated with the specified list-box item.

```
DWORD_PTR GetItemData(
   int nIndex
) const;
```

Parameters

nIndex

Specifies the zero-based index of the item in the list box.

Return Value

The 32-bit value associated with the item, or **LB_ERR** if an error occurs.

Remarks

The doubleword value was the *dwItemData* parameter of a SetItemData call.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// If any item's data is equal to zero then reset it to -1.
for (int i=0;i < pmyListBox->GetCount();i++)
{
   if (pmyListBox->GetItemData(i) == 0)
   {
      pmyListBox->SetItemData(i, (DWORD) -1);
   }
}
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::AddString
CListBox::GetItemDataPtr
CListBox::SetItemDataPtr
CListBox::InsertString
CListBox::SetItemData
LB_GETITEMDATA

Other Resources

CListBox Members

CListBox::GetItemDataPtr

Retrieves the application-supplied 32-bit value associated with the specified list-box item as a pointer (**void***).

```
void* GetItemDataPtr(
   int nIndex
) const;
```

Parameters

nIndex

Specifies the zero-based index of the item in the list box.

Return Value

Retrieves a pointer, or –1 if an error occurs.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;
extern LPVOID    lpmyPtr;

// Check all the items in the list box; if an item's
// data pointer is equal to my pointer then reset it to NULL.
for (int i=0;i < pmyListBox->GetCount();i++)
{
   if (pmyListBox->GetItemDataPtr(i) == lpmyPtr)
   {
      pmyListBox->SetItemDataPtr(i, NULL);
   }
}
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::AddString
CListBox::GetItemData
CListBox::InsertString
CListBox::SetItemData
LB_GETITEMDATA

Other Resources

CListBox Members

CListBox::GetItemHeight

Determines the height of items in a list box.

```
int GetItemHeight(
   int nIndex
) const;
```

Parameters

nIndex

Specifies the zero-based index of the item in the list box. This parameter is used only if the list box has the **LBS_OWNERDRAWVARIABLE** style; otherwise, it should be set to 0.

Return Value

The height, in pixels, of the items in the list box. If the list box has the LBS_OWNERDRAWVARIABLE style, the return value is the height of the item specified by *nIndex*. If an error occurs, the return value is **LB_ERR**.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Set the height of every item so the item
// is completely visible.
CString str;
CSize   sz;
int     dx=0;
CDC*    pDC = pmyListBox->GetDC();
for (int i=0;i < pmyListBox->GetCount();i++)
{
   pmyListBox->GetText( i, str );
   sz = pDC->GetTextExtent(str);

   // Only want to set the item height if the current height
   // is not big enough.
   if (pmyListBox->GetItemHeight(i) < sz.cy)
      pmyListBox->SetItemHeight( i, sz.cy );
}
pmyListBox->ReleaseDC(pDC);
```

See Also

Reference

CListBox Class
Hierarchy Chart
LB_GETITEMHEIGHT
CListBox::SetItemHeight

Other Resources

CListBox Members

CListBox::GetItemRect

Retrieves the dimensions of the rectangle that bounds a list-box item as it is currently displayed in the list-box window.

```
int GetItemRect(
   int nIndex,
   LPRECT lpRect
) const;
```

Parameters

nIndex

Specifies the zero-based index of the item.

lpRect

Specifies a long pointer to a RECT structure that receives the list-box client coordinates of the item.

Return Value

LB_ERR if an error occurs.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Dump all of the items bounds.
#ifdef _DEBUG
   CString str;
   RECT r;
   for (int i=0;i < pmyListBox->GetCount();i++)
   {
      pmyListBox->GetItemRect( i, &r );

      str.Format(_T("item %d: left = %d, top = %d, right = %d,
         bottom = %d\r\n"),
         i,
         r.left,
         r.top,
         r.right,
         r.bottom);
      afxDump << str;
   }
#endif
```

See Also

Reference

CListBox Class

Hierarchy Chart

LB_GETITEMRECT

Other Resources

CListBox Members

CListBox::GetListBoxInfo

Retrieves the number of items per column.

```
DWORD GetListBoxInfo( ) const;
```

Return Value

Number of items per column of the **CListBox** object.

Remarks

This member function emulates the functionality of the LB_GETLISTBOXINFO message, as described in the *Platform SDK*.

See Also

Reference

CListBox Class

Hierarchy Chart

Other Resources

CListBox Members

CListBox::GetLocale

Retrieves the locale used by the list box.

```
LCID GetLocale( ) const;
```

Return Value

The locale identifier (LCID) value for the strings in the list box.

Remarks

The locale is used, for example, to determine the sort order of the strings in a sorted list box.

Example

See the example for CListBox::SetLocale.

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::SetLocale
GetStringTypeW
GetSystemDefaultLCID
GetUserDefaultLCID

Other Resources

CListBox Members

CListBox::GetSel

Retrieves the selection state of an item.

```
int GetSel(
   int nIndex
) const;
```

Parameters

nIndex

Specifies the zero-based index of the item.

Return Value

A positive number if the specified item is selected; otherwise, it is 0. The return value is **LB_ERR** if an error occurs.

Remarks

This member function works with both single- and multiple-selection list boxes.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Dump all of the items select state.
#ifdef _DEBUG
   CString str;
   for (int i=0;i < pmyListBox->GetCount();i++)
   {
      str.Format(_T("item %d: select state is %s\r\n"),
      i,
      pmyListBox->GetSel( i ) > 0 ? _T("true") : _T("false"));
      afxDump << str;
   }
#endif
```

See Also

Reference

CListBox Class

Hierarchy Chart

LB_GETSEL

CListBox::SetSel

Other Resources

CListBox Members

CListBox::GetSelCount

Retrieves the total number of selected items in a multiple-selection list box.

```
int GetSelCount( ) const;
```

Return Value

The count of selected items in a list box. If the list box is a single-selection list box, the return value is **LB_ERR**.

Example

See the example for CListBox::GetSelItems.

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::SetSel
LB_GETSELCOUNT

Other Resources

CListBox Members

CListBox::GetSelItems

Fills a buffer with an array of integers that specifies the item numbers of selected items in a multiple-selection list box.

```
int GetSelItems(
   int nMaxItems,
   LPINT rgIndex
) const;
```

Parameters

nMaxItems

Specifies the maximum number of selected items whose item numbers are to be placed in the buffer.

rgIndex

Specifies a long pointer to a buffer large enough for the number of integers specified by *nMaxItems*.

Return Value

The actual number of items placed in the buffer. If the list box is a single-selection list box, the return value is **LB_ERR**.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Get the indexes of all the selected items.
int nCount = pmyListBox->GetSelCount();
CArray<int,int> aryListBoxSel;

aryListBoxSel.SetSize(nCount);
pmyListBox->GetSelItems(nCount, aryListBoxSel.GetData());

// Dump the selection array.
#ifdef _DEBUG
   afxDump << aryListBoxSel;
#endif
```

See Also

Reference

CListBox Class
Hierarchy Chart
LB_GETSELITEMS

Other Resources

CListBox Members

CListBox::GetText

Gets a string from a list box.

```
int GetText(
   int nIndex,
   LPTSTR lpszBuffer
) const;
void GetText(
   int nIndex,
   CString& rString
) const;
```

Parameters

nIndex

Specifies the zero-based index of the string to be retrieved.

lpszBuffer

Points to the buffer that receives the string. The buffer must have sufficient space for the string and a terminating null character. The size of the string can be determined ahead of time by calling the **GetTextLen** member function.

rString

A reference to a **CString** object.

Return Value

The length (in bytes) of the string, excluding the terminating null character. If *nIndex* does not specify a valid index, the return value is **LB_ERR**.

Remarks

The second form of this member function fills a **CString** object with the string text.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Dump all of the items in the list box.
#ifdef _DEBUG
   CString str, str2;
   int n;
   for (int i=0;i < pmyListBox->GetCount();i++)
   {
      n = pmyListBox->GetTextLen( i );
      pmyListBox->GetText( i, str.GetBuffer(n) );
      str.ReleaseBuffer();

      str2.Format(_T("item %d: %s\r\n"), i, str.GetBuffer(0));
      afxDump << str2;
   }
#endif
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::GetTextLen
LB_GETTEXT

Other Resources

CListBox Members

CListBox::GetTextLen

Gets the length of a string in a list-box item.

```
int GetTextLen(
   int nIndex
) const;
```

Parameters

nIndex

Specifies the zero-based index of the string.

Return Value

The length of the string in characters, excluding the terminating null character. If *nIndex* does not specify a valid index, the return value is **LB_ERR**.

Example

See the example for CListBox::GetText.

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::GetText
LB_GETTEXTLEN

Other Resources

CListBox Members

CListBox::GetTopIndex

Retrieves the zero-based index of the first visible item in a list box.

```
int GetTopIndex( ) const;
```

Return Value

The zero-based index of the first visible item in a list box if successful, **LB_ERR** otherwise.

Remarks

Initially, item 0 is at the top of the list box, but if the list box is scrolled, another item may be at the top.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Want an item in the bottom half to be the first visible item.
int n = pmyListBox->GetCount()/2;
if (pmyListBox->GetTopIndex() < n)
{
   pmyListBox->SetTopIndex(n);
   ASSERT(pmyListBox->GetTopIndex() == n);
}
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::SetTopIndex
LB_GETTOPINDEX

Other Resources

CListBox Members

CListBox::InitStorage

Allocates memory for storing list-box items.

```
int InitStorage(
   int nItems,
   UINT nBytes
);
```

Parameters

nItems

Specifies the number of items to add.

nBytes

Specifies the amount of memory, in bytes, to allocate for item strings.

Return Value

If successful, the maximum number of items that the list box can store before a memory reallocation is needed, otherwise **LB_ERRSPACE**, meaning not enough memory is available.

Remarks

Call this function before adding a large number of items to a **CListBox**.

This function helps speed up the initialization of list boxes that have a large number of items (more than 100). It preallocates the specified amount of memory so that subsequent AddString, InsertString, and Dir functions take the shortest possible time. You can use estimates for the parameters. If you overestimate, some extra memory is allocated; if you underestimate, the normal allocation is used for items that exceed the preallocated amount.

Windows 95/98 only: The *nItems* parameter is limited to 16-bit values. This means list boxes cannot contain more than 32,767 items. Although the number of items is restricted, the total size of the items in a list box is limited only by available memory.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Initialize the storage of the list box to be 256 strings with
// about 10 characters per string, performance improvement.
int n = pmyListBox->InitStorage(256, 10);
ASSERT(n != LB_ERRSPACE);

// Add 256 items to the list box.
CString str;
for (int i=0;i < 256;i++)
{
   str.Format(_T("item string %d"), i);
   pmyListBox->AddString( str );
}
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::CListBox
CListBox::Create
CListBox::ResetContent
LB_INITSTORAGE

Other Resources

CListBox Members

CListBox::InsertString

Inserts a string into the list box.

```
int InsertString(
   int nIndex,
   LPCTSTR lpszItem
);
```

Parameters

nIndex

Specifies the zero-based index of the position to insert the string. If this parameter is –1, the string is added to the end of the list.

lpszItem

Points to the null-terminated string that is to be inserted.

Return Value

The zero-based index of the position at which the string was inserted. The return value is **LB_ERR** if an error occurs; the return value is **LB_ERRSPACE** if insufficient space is available to store the new string.

Remarks

Unlike the AddString member function, **InsertString** does not cause a list with the LBS_SORT style to be sorted.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Insert items in between existing items.
CString str;
int n = pmyListBox->GetCount();
for (int i=0;i < n;i++)
{
   str.Format(_T("item string %c"), (char)('A'+i));
   pmyListBox->InsertString( 2*i, str );
}
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::AddString
LB_INSERTSTRING

Other Resources

CListBox Members

CListBox::ItemFromPoint

Determines the list-box item nearest the point specified in *pt*.

```
UINT ItemFromPoint(
   CPoint pt,
   BOOL& bOutside
) const;
```

Parameters

pt

Point for which to find the nearest item, specified relative to the upper-left corner of the client area of the list box.

bOutside

Reference to a **BOOL** variable which will be set to **TRUE** if *pt* is outside the client area of the list box, **FALSE** if *pt* is inside the client area of the list box.

Return Value

The index of the nearest item to the point specified in *pt*.

Remarks

You could use this function to determine which list-box item the mouse cursor moves over.

Example

See the example for CListBox::SetAnchorIndex.

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::GetItemRect
LB_ITEMFROMPOINT

Other Resources

CListBox Members

CListBox::MeasureItem

Called by the framework when a list box with an owner-draw style is created.

```
virtual void MeasureItem(
   LPMEASUREITEMSTRUCT lpMeasureItemStruct
);
```

Parameters

lpMeasureItemStruct

A long pointer to a MEASUREITEMSTRUCT structure.

Remarks

By default, this member function does nothing. Override this member function and fill in the **MEASUREITEMSTRUCT** structure to inform Windows of the list-box dimensions. If the list box is created with the LBS_OWNERDRAWVARIABLE style, the framework calls this member function for each item in the list box. Otherwise, this member is called only once.

For further information about using the LBS_OWNERDRAWFIXED style in an owner-draw list box created with the **SubclassDlgItem** member function of **CWnd**, see the discussion in Technical Note 14.

See CWnd::OnMeasureItem for a description of the **MEASUREITEMSTRUCT** structure**.**

Example

```
// CMyListBox is my owner-drawn list box derived from CListBox. This
// example measures an item and sets the height of the item to twice the
// vertical extent of its text. The list box control was created with the
// following code:
//   pmyListBox->Create(
//      WS_CHILD|WS_VISIBLE|WS_BORDER|WS_HSCROLL|WS_VSCROLL|
//      LBS_SORT|LBS_MULTIPLESEL|LBS_OWNERDRAWVARIABLE,
//      myRect, pParentWnd, 1);
//
void CMyListBox::MeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct)
{
   ASSERT(lpMeasureItemStruct->CtlType == ODT_LISTBOX);
   LPCTSTR lpszText = (LPCTSTR) lpMeasureItemStruct->itemData;
   ASSERT(lpszText != NULL);
   CSize   sz;
   CDC*    pDC = GetDC();

   sz = pDC->GetTextExtent(lpszText);

   ReleaseDC(pDC);

   lpMeasureItemStruct->itemHeight = 2*sz.cy;
}
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::CompareItem
CWnd::OnMeasureItem
CListBox::DrawItem
CListBox::DeleteItem

Other Resources

CListBox Members

CListBox::ResetContent

Removes all items from a list box.

```
void ResetContent( );
```

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Delete all the items from the list box.
pmyListBox->ResetContent();
ASSERT(pmyListBox->GetCount() == 0);
```

See Also

Reference

CListBox Class

Hierarchy Chart

LB_RESETCONTENT

Other Resources

CListBox Members

CListBox::SelectString

Searches for a list-box item that matches the specified string, and if a matching item is found, it selects the item.

```
int SelectString(
   int nStartAfter,
   LPCTSTR lpszItem
);
```

Parameters

nStartAfter

Contains the zero-based index of the item before the first item to be searched. When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by *nStartAfter*. If *nStartAfter* is –1, the entire list box is searched from the beginning.

lpszItem

Points to the null-terminated string that contains the prefix to search for. The search is case independent, so this string may contain any combination of uppercase and lowercase letters.

Return Value

The index of the selected item if the search was successful. If the search was unsuccessful, the return value is **LB_ERR** and the current selection is not changed.

Remarks

The list box is scrolled, if necessary, to bring the selected item into view.

This member function cannot be used with a list box that has the LBS_MULTIPLESEL style.

An item is selected only if its initial characters (from the starting point) match the characters in the string specified by *lpszItem*.

Use the **FindString** member function to find a string without selecting the item.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;
// The string to match.
extern LPCTSTR lpszmyString;

// Select the item that begins with the specified string.
int nIndex = pmyListBox->SelectString(0, lpszmyString);
ASSERT(nIndex != LB_ERR);
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::FindString
LB_SELECTSTRING

Other Resources

CListBox Members

CListBox::SelItemRange

Selects multiple consecutive items in a multiple-selection list box.

```
int SelItemRange(
   BOOL bSelect,
   int nFirstItem,
   int nLastItem
);
```

Parameters

bSelect

Specifies how to set the selection. If *bSelect* is **TRUE**, the string is selected and highlighted; if **FALSE**, the highlight is removed and the string is no longer selected.

nFirstItem

Specifies the zero-based index of the first item to set.

nLastItem

Specifies the zero-based index of the last item to set.

Return Value

LB_ERR if an error occurs.

Remarks

Use this member function only with multiple-selection list boxes. If you need to select only one item in a multiple-selection list box — that is, if *nFirstItem* is equal to *nLastItem* — call the SetSel member function instead.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Select half of the items.
pmyListBox->SelItemRange(TRUE, 0, pmyListBox->GetCount()/2);
```

See Also

Reference

CListBox Class
Hierarchy Chart
LB_SELITEMRANGE
CListBox::GetSelItems

Other Resources

CListBox Members

CListBox::SetAnchorIndex

Sets the anchor in a multiple-selection list box to begin an extended selection.

```
void SetAnchorIndex(
   int nIndex
);
```

Parameters

nIndex

Specifies the zero-based index of the list-box item that will be the anchor.

Remarks

In a multiple-selection list box, the anchor item is the first or last item in a block of contiguous selected items.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;
// Point where mouse was clicked.
extern CPoint myPoint;

BOOL bOutside;
UINT uItem = pmyListBox->ItemFromPoint(myPoint, bOutside);

if (!bOutside)
{
   // Set the anchor to be the middle item.
   pmyListBox->SetAnchorIndex(uItem);
   ASSERT(pmyListBox->GetAnchorIndex() == uItem);
}
```

See Also

Reference

CListBox Class

Hierarchy Chart

CListBox::GetAnchorIndex

Other Resources

CListBox Members

CListBox::SetCaretIndex

Sets the focus rectangle to the item at the specified index in a multiple-selection list box.

```
int SetCaretIndex(
   int nIndex,
   BOOL bScroll = TRUE
);
```

Parameters

nIndex

Specifies the zero-based index of the item to receive the focus rectangle in the list box.

bScroll

If this value is 0, the item is scrolled until it is fully visible. If this value is not 0, the item is scrolled until it is at least partially visible.

Return Value

LB_ERR if an error occurs.

Remarks

If the item is not visible, it is scrolled into view.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Set the caret to be the middle item.
pmyListBox->SetCaretIndex(pmyListBox->GetCount()/2);
ASSERT(pmyListBox->GetCaretIndex() == pmyListBox->GetCount()/2);
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::GetCaretIndex
LB_SETCARETINDEX

Other Resources

CListBox Members

CListBox::SetColumnWidth

Sets the width in pixels of all columns in a multicolumn list box (created with the LBS_MULTICOLUMN style).

```
void SetColumnWidth(
   int cxWidth
);
```

Parameters

cxWidth

Specifies the width in pixels of all columns.

Example

```
// The pointer to my multicolumn list box.
extern CListBox* pmyListBox;

// Find the pixel width of the largest item.
CString str;
CSize   sz;
int     dx=0;
CDC*    pDC = pmyListBox->GetDC();
for (int i=0;i < pmyListBox->GetCount();i++)
{
   pmyListBox->GetText( i, str );
   sz = pDC->GetTextExtent(str);

   if (sz.cx > dx)
      dx = sz.cx;
}
pmyListBox->ReleaseDC(pDC);

// Set the column width of the first column to be one and 1/3 units
// of the largest string.
pmyListBox->SetColumnWidth(dx*4/3);
```

See Also

Reference

CListBox Class

Hierarchy Chart

LB_SETCOLUMNWIDTH

Other Resources

CListBox Members

CListBox::SetCurSel

Selects a string and scrolls it into view, if necessary.

```
int SetCurSel(
   int nSelect
);
```

Parameters

nSelect

Specifies the zero-based index of the string to be selected. If *nSelect* is –1, the list box is set to have no selection.

Return Value

LB_ERR if an error occurs.

Remarks

When the new string is selected, the list box removes the highlight from the previously selected string.

Use this member function only with single-selection list boxes. It cannot be used to set or remove a selection in a multiple-selection list box.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Select the last item in the list box.
int nCount = pmyListBox->GetCount();
if (nCount > 0)
   pmyListBox->SetCurSel(nCount-1);
```

See Also

Reference

CListBox Class
Hierarchy Chart
LB_SETCURSEL
CListBox::GetCurSel

Other Resources

CListBox Members

CListBox::SetHorizontalExtent

Sets the width, in pixels, by which a list box can be scrolled horizontally.

```
void SetHorizontalExtent(
   int cxExtent
);
```

Parameters

cxExtent

Specifies the number of pixels by which the list box can be scrolled horizontally.

Remarks

If the size of the list box is smaller than this value, the horizontal scroll bar will horizontally scroll items in the list box. If the list box is as large or larger than this value, the horizontal scroll bar is hidden.

To respond to a call to **SetHorizontalExtent**, the list box must have been defined with the WS_HSCROLL style.

This member function is not useful for multicolumn list boxes. For multicolumn list boxes, call the **SetColumnWidth** member function.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Find the longest string in the list box.
CString      str;
CSize      sz;
int      dx = 0;
TEXTMETRIC   tm;
CDC*      pDC = pmyListBox->GetDC();
CFont*      pFont = pmyListBox->GetFont();

// Select the listbox font, save the old font
CFont* pOldFont = pDC->SelectObject(pFont);
// Get the text metrics for avg char width
pDC->GetTextMetrics(&tm);

for (int i = 0; i < pmyListBox->GetCount(); i++)
{
      pmyListBox->GetText(i, str);
   sz = pDC->GetTextExtent(str);

   // Add the avg width to prevent clipping
   sz.cx += tm.tmAveCharWidth;

   if (sz.cx > dx)
            dx = sz.cx;
}
// Select the old font back into the DC
pDC->SelectObject(pOldFont);
pmyListBox->ReleaseDC(pDC);

// Set the horizontal extent so every character of all strings
// can be scrolled to.
pmyListBox->SetHorizontalExtent(dx);
```

See Also

Reference

CListBox Class

Hierarchy Chart

CListBox::GetHorizontalExtent
CListBox::SetColumnWidth
LB_SETHORIZONTALEXTENT

Other Resources

CListBox Members

CListBox::SetItemData

Sets a 32-bit value associated with the specified item in a list box.

```
int SetItemData(
   int nIndex,
   DWORD_PTR dwItemData
);
```

Parameters

nIndex

Specifies the zero-based index of the item.

dwItemData

Specifies the value to be associated with the item.

Return Value

LB_ERR if an error occurs.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Set the data of each item to be equal to its index.
for (int i=0;i < pmyListBox->GetCount();i++)
{
   pmyListBox->SetItemData(i, i);
}
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::SetItemDataPtr
CListBox::GetItemData
LB_SETITEMDATA

Other Resources

CListBox Members

CListBox::SetItemDataPtr

Sets the 32-bit value associated with the specified item in a list box to be the specified pointer (**void***).

```
int SetItemDataPtr(
   int nIndex,
   void* pData
);
```

Parameters

nIndex

Specifies the zero-based index of the item.

pData

Specifies the pointer to be associated with the item.

Return Value

LB_ERR if an error occurs.

Remarks

This pointer remains valid for the life of the list box, even though the item's relative position within the list box might change as items are added or removed. Hence, the item's index within the box can change, but the pointer remains reliable.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Set the data pointer of each item to be NULL.
for (int i=0;i < pmyListBox->GetCount();i++)
{
   pmyListBox->SetItemDataPtr(i, NULL);
}
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::SetItemData
CListBox::GetItemData
CListBox::GetItemDataPtr
LB_SETITEMDATA

Other Resources

CListBox Members

CListBox::SetItemHeight

Sets the height of items in a list box.

```
int SetItemHeight(
   int nIndex,
   UINT cyItemHeight
);
```

Parameters

nIndex

Specifies the zero-based index of the item in the list box. This parameter is used only if the list box has the **LBS_OWNERDRAWVARIABLE** style; otherwise, it should be set to 0.

cyItemHeight

Specifies the height, in pixels, of the item.

Return Value

LB_ERR if the index or height is invalid.

Remarks

If the list box has the LBS_OWNERDRAWVARIABLE style, this function sets the height of the item specified by *nIndex*. Otherwise, this function sets the height of all items in the list box.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Set the height of every item to be the
// vertical size of the item's text extent.
CString str;
CSize   sz;
int     dx=0;
CDC*    pDC = pmyListBox->GetDC();
for (int i=0;i < pmyListBox->GetCount();i++)
{
   pmyListBox->GetText( i, str );
   sz = pDC->GetTextExtent(str);

   pmyListBox->SetItemHeight( i, sz.cy );
}
pmyListBox->ReleaseDC(pDC);
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::GetItemHeight
LB_SETITEMHEIGHT

Other Resources

CListBox Members

CListBox::SetLocale

Sets the locale identifier for this list box.

```
LCID SetLocale(
   LCID nNewLocale
);
```

Parameters

nNewLocale

The new locale identifier (LCID) value to set for the list box.

Return Value

The previous locale identifier (LCID) value for this list box.

Remarks

If **SetLocale** is not called, the default locale is obtained from the system. This system default locale can be modified by using Control Panel's Regional (or International) application.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;
// My LCID to use.
LCID mylcid = MAKELCID(MAKELANGID(LANG_SPANISH, SUBLANG_SPANISH_MEXICAN),
   SORT_DEFAULT);

// Force the list box to use my locale.
pmyListBox->SetLocale(mylcid);
ASSERT(pmyListBox->GetLocale() == mylcid);
```

See Also

Reference

CListBox Class

Hierarchy Chart

CListBox::GetLocale

Other Resources

CListBox Members

CListBox::SetSel

Selects a string in a multiple-selection list box.

```
int SetSel(
   int nIndex,
   BOOL bSelect = TRUE
);
```

Parameters

nIndex

Contains the zero-based index of the string to be set. If –1, the selection is added to or removed from all strings, depending on the value of *bSelect*.

bSelect

Specifies how to set the selection. If *bSelect* is **TRUE**, the string is selected and highlighted; if **FALSE**, the highlight is removed and the string is no longer selected. The specified string is selected and highlighted by default.

Return Value

LB_ERR if an error occurs.

Remarks

Use this member function only with multiple-selection list boxes.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Select all of the items with an even index and
// deselect all others.
for (int i=0;i < pmyListBox->GetCount();i++)
{
   pmyListBox->SetSel(i, ((i%2) == 0));
}
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::GetSel
LB_SETSEL

Other Resources

CListBox Members

CListBox::SetTabStops

Sets the tab-stop positions in a list box.

```
void SetTabStops( );
BOOL SetTabStops(
   const int& cxEachStop
);
BOOL SetTabStops(
   int nTabStops,
   LPINT rgTabStops
);
```

Parameters

cxEachStop

Tab stops are set at every *cxEachStop* dialog units. See *rgTabStops* for a description of a dialog unit.

nTabStops

Specifies the number of tab stops to have in the list box.

rgTabStops

Points to the first member of an array of integers containing the tab-stop positions in dialog units. A dialog unit is a horizontal or vertical distance. One horizontal dialog unit is equal to one-fourth of the current dialog base width unit, and one vertical dialog unit is equal to one-eighth of the current dialog base height unit. The dialog base units are computed based on the height and width of the current system font. The **GetDialogBaseUnits** Windows function returns the current dialog base units in pixels. The tab stops must be sorted in increasing order; back tabs are not allowed.

Return Value

Nonzero if all the tabs were set; otherwise 0.

Remarks

To set tab stops to the default size of 2 dialog units, call the parameterless version of this member function. To set tab stops to a size other than 2, call the version with the *cxEachStop* argument.

To set tab stops to an array of sizes, use the version with the *rgTabStops* and *nTabStops* arguments. A tab stop will be set for each value in *rgTabStops*, up to the number specified by *nTabStops*.

To respond to a call to the **SetTabStops** member function, the list box must have been created with the LBS_USETABSTOPS style.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Find the pixel width of the largest first substring.
CString str;
CSize   sz;
int     nIndex, dx=0;
CDC*    pDC = pmyListBox->GetDC();
for (int i=0;i < pmyListBox->GetCount();i++)
{
   pmyListBox->GetText( i, str );

   if ((nIndex=str.Find('\t')) != -1)
      str = str.Right(nIndex);

   sz = pDC->GetTextExtent(str);

   if (sz.cx > dx)
      dx = sz.cx;
```

```
}
pmyListBox->ReleaseDC(pDC);

// Set tab stops at every one and 1/3 units
// of the largest string.
// NOTE: Convert pixels to dialog units.
pmyListBox->SetTabStops((dx*4/3 * 4) / LOWORD(::GetDialogBaseUnits()));
```

See Also

Reference

CListBox Class

Hierarchy Chart

LB_SETTABSTOPS

GetDialogBaseUnits

Other Resources

CListBox Members

CListBox::SetTopIndex

Ensures that a particular list-box item is visible.

```
int SetTopIndex(
   int nIndex
);
```

Parameters

nIndex

Specifies the zero-based index of the list-box item.

Return Value

Zero if successful, or **LB_ERR** if an error occurs.

Remarks

The system scrolls the list box until either the item specified by *nIndex* appears at the top of the list box or the maximum scroll range has been reached.

Example

```
// The pointer to my list box.
extern CListBox* pmyListBox;

// Set the first visible item in the list box to be the middle item
pmyListBox->SetTopIndex(pmyListBox->GetCount()/2);
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::GetTopIndex
LB_SETTOPINDEX

Other Resources

CListBox Members

CListBox::VKeyToItem

Called by the framework when the list box's parent window receives a **WM_VKEYTOITEM** message from the list box.

```
virtual int VKeyToItem(
   UINT nKey,
   UINT nIndex
);
```

Parameters

nKey

The virtual key code of the key the user pressed. For a list of of standard virtual key codes, see Winuser.h

nIndex

The current position of the list-box caret.

Return Value

Returns – 2 for no further action, – 1 for default action, or a nonnegative number to specify an index of a list box item on which to perform the default action for the keystroke.

Remarks

The **WM_VKEYTOITEM** message is sent by the list box when it receives a **WM_KEYDOWN** message, but only if the list box meets both of the following:

- Has the LBS_WANTKEYBOARDINPUT style set.
- Has at least one item.

You should never call this function yourself. Override this function to provide your own custom handling of keyboard messages.

You must return a value to tell the framework what action your override performed. A return value of – 2 indicates that the application handled all aspects of selecting the item and requires no further action by the list box. Before returning – 2, you could set the selection or move the caret or both. To set the selection, use SetCurSel or SetSel. To move the caret, use SetCaretIndex.

A return value of – 1 indicates that the list box should perform the default action in response to the keystroke.The default implementation returns – 1.

A return value of 0 or greater specifies the index of an item in the list box and indicates that the list box should perform the default action for the keystroke on the given item.

Example

```
// CMyListBox is my owner-drawn list box derived from CListBox. This
// example moves the caret down one item on the down key and up one item
// on the up key. The list box control was created with the following
// code:
//   pmyListBox->Create(
//      WS_CHILD|WS_VISIBLE|WS_BORDER|WS_HSCROLL|WS_VSCROLL|
//      LBS_SORT|LBS_MULTIPLESEL|LBS_OWNERDRAWVARIABLE|LBS_WANTKEYBOARDINPUT,
//      CRect(10,10,200,200), pParentWnd, 1);
//
int CMyListBox::VKeyToItem(UINT nKey, UINT nIndex)
{
   // On key up, move the caret up one item.
   if ((nKey == VK_UP) && (nIndex > 0))
      SetCaretIndex(nIndex-1);

   // On key down, move the caret down one item.
   else if ((nKey == VK_DOWN) && (nIndex < GetCount()))
      SetCaretIndex(nIndex+1);
```

```
    // Do not perform any default processing.
    return -2;
}
```

See Also

Reference

CListBox Class
Hierarchy Chart
CListBox::CharToItem
CListBox::SetCurSel
CListBox::SetSel
CListBox::SetCaretIndex

Other Resources

CListBox Members

CListCtrl Class

Encapsulates the functionality of a "list view control," which displays a collection of items each consisting of an icon (from an image list) and a label.

```
class CListCtrl : public CWnd
```

Remarks

In addition to an icon and label, each item can have information displayed in columns to the right of the icon and label. This control (and therefore the **CListCtrl** class) is available only to programs running under Windows 95/98 and Windows NT version 3.51 and later.

The following is a brief overview of the **CListCtrl** class. For a detailed, conceptual discussion, see Using CListCtrl and Controls.

Views

List view controls can display their contents in four different ways, called "views."

- Icon view

  Each item appears as a full-sized icon (32 x 32 pixels) with a label below it. The user can drag the items to any location in the list view window.

- Small icon view

  Each item appears as a small icon (16 x 16 pixels) with the label to the right of it. The user can drag the items to any location in the list view window.

- List view

  Each item appears as a small icon with a label to the right of it. Items are arranged in columns and cannot be dragged to any location in the list view window.

- Report view

  Each item appears on its own line, with additional information arranged in columns to the right. The leftmost column contains the small icon and label, and subsequent columns contain subitems as specified by the application. An embedded header control (class CHeaderCtrl) implements these columns. For more information on the header control and columns in a report view, see Using CListCtrl: Adding Columns to the Control (Report View).

Also see:

- Knowledge Base article Q250614: HOWTO: Sort Items in a CListCtrl in Report View
- Knowledge Base article Q200054: PRB: OnTimer() Is Not Called Repeatedly for a List Control

The style of the control's current list view determines the current view. For more information on these styles and their usage, see Using CListCtrl: Changing List Control Styles.

Extended Styles

In addition to the standard list styles, class **CListCtrl** supports a large set of extended styles, providing enriched functionality. Some examples of this functionality include:

- Hover selection

  When enabled, allows automatic selection of an item when the cursor remains over the item for a certain period of time.

- Virtual list views

  When enabled, allows the control to support up to **DWORD** items. This is possible by placing the overhead of managing item data on the application. Except for the item selection and focus information, all item information must be managed by the application. For more information, see Using CListCtrl: Virtual List Controls.

- One– and two– click activation

When enabled, allows hot tracking (automatic highlighting of the item text) and one– or two– click activation of the highlighted item.

- Drag and drop column ordering

  When enabled, allows drag-and-drop reordering of columns in a list view control. Only available in report view.

For information on using these new extended styles, see Using CListCtrl: Changing List Control Styles.

Items and Subitems

Each item in a list view control consists of an icon (from an image list), a label, a current state, and an application-defined value (referred to as "item data"). One or more subitems can also be associated with each item. A "subitem" is a string that, in report view, can be displayed in a column to the right of an item's icon and label. All items in a list view control must have the same number of subitems.

Class **CListCtrl** provides several functions for inserting, deleting, finding, and modifying these items. For more information, see CListCtrl::GetItem, CListCtrl::InsertItem, and CListCtrl::FindItem, Using CListCtrl: Adding Items to the Control, and Using CListCtrl: Scrolling, Arranging, Sorting, and Finding in List Controls.

By default, the list view control is responsible for storing an item's icon and text attributes. However, in addition to these item types, class **CListCtrl** supports "callback items." A "callback item" is a list view item for which the application — rather than the control — stores the text, icon, or both. A callback mask is used to specify which item attributes (text and/or icon) are supplied by the application. If an application uses callback items, it must be able to supply the text and/or icon attributes on demand. Callback items are helpful when your application already maintains some of this information. For more information, see Using CListCtrl: Callback Items and the Callback Mask.

Image Lists

The icons, header item images, and application– defined states for list view items are contained in several image lists (implemented by class CImageList), which you create and assign to the list view control. Each list view control can have up to four different types of image lists:

- Large icon

  Used in the icon view for full-sized icons.

- Small icon

  Used in the small icon, list, and report views for smaller versions of the icons used in the icon view.

- Application-defined state

  Contains state images, which are displayed next to an item's icon to indicate an application-defined state.

- Header item

  Used in the report view for small images that appear in each header control item.

By default, a list view control destroys the image lists assigned to it when it is destroyed; however, the developer can customize this behavior by destroying each image list when it is no longer used, as determined by the application. For more information, see Using CListCtrl: List Items and Image Lists.

Requirements

Header: afxcmn.h

See Also

Tasks

ROWLIST Sample: Demonstrates Selecting Full Rows in List Views

Reference

CWnd Class

Hierarchy Chart

CImageList Class

Other Resources

CListCtrl Members

CListCtrl Members

Construction

Attributes

Operations

Overridables

Construction

CListCtrl	Constructs a **CListCtrl** object.
Create	Creates a list control and attaches it to a **CListCtrl** object.
CreateEx	Creates a list control with the specified Windows extended styles and attaches it to a **CListCtrl** object.

Attributes

ApproximateViewRect	Determines the width and height required to display the items of a list view control.
CancelEditLabel	Cancels item text editing operation.
GetInsertMark	Retrieves the current position of the insertion mark.
InsertMarkHitTest	Retrieves the insertion point closest to a specified point.
GetInsertMarkColor	Retrieves the current color of the insertion mark.
GetInsertMarkRect	Retrieves the rectangle that bounds the insertion point.
GetOutlineColor	Retrieves the color of the border of a list view control.
SetOutlineColor	Sets the color of the border of a list view control.
GetTileInfo	Retrieves information about a tile in a list view control.
GetTileViewInfo	Retrieves information about a list view control in tile view.
GetSelectedColumn	Retrieves the index of the currently selected column in the list control.
IsGroupViewEnabled	Determines whether group view is enabled for a list view control.
SetInfoTip	Sets the tooltip text.
SetInsertMark	Sets the insertion point to the defined position.
SetInsertMarkColor	Sets the color of the insertion point.
SetTileInfo	Sets the information for a tile of the list view control.
SetTileViewInfo	Sets information that a list view control uses in tile view.
EnableGroupView	Enables or disables whether the items in a list view control display as a group.

GetGroupInfo	Gets the information for a specified group of the list view control.
GetGroupMetrics	Retrieves the metrics of a group.
GetView	Gets the view of the list view control.
HasGroup	Determines if the list view control has the specified group.
InsertGroup	Inserts a group into the list view control.
InsertGroupSorted	Inserts the specified group into an ordered list of groups.
MoveGroup	Moves the specified group.
MoveItemToGroup	Moves the specified group to the specified zero based index of the list view control.
RemoveAllGroups	Removes all groups from a list view control.
RemoveGroup	Removes the specified group from the list view control.
SetGroupInfo	Sets the information for the specified group of a list view control.
SetGroupMetrics	Sets the group metrics of a list view control.
SetSelectedColumn	Sets the selected column of the list view control.
SetView	Sets the view of the list view control.
SortGroups	Sorts the groups of a list view control with a user-defined function.
GetBkColor	Retrieves the background color of a list view control.
GetBkImage	Retrieves the current background image of a list view control.
GetCallbackMask	Retrieves the callback mask for a list view control.
GetCheck	Retrieves the current display status of the state image associated with an item.
GetColumn	Retrieves the attributes of a control's column.
GetColumnOrderArray	Retrieves the column order (left to right) of a list view control.
GetColumnWidth	Retrieves the width of a column in report view or list view.
GetCountPerPage	Calculates the number of items that can fit vertically in a list view control.
GetEditControl	Retrieves the handle of the edit control used to edit an item's text.
GetExtendedStyle	Retrieves the current extended styles of a list view control.
GetFirstSelectedItemPosition	Retrieves the position of the first selected list view item in a list view control.
GetHeaderCtrl	Retrieves the header control of a list view control.

GetHotCursor	Retrieves the cursor used when hot tracking is enabled for a list view control.
GetHotItem	Retrieves the list view item currently under the cursor.
GetHoverTime	Retrieves the current hover time of a list view control.
GetImageList	Retrieves the handle of an image list used for drawing list view items.
GetItem	Retrieves a list view item's attributes.
GetItemCount	Retrieves the number of items in a list view control.
GetItemData	Retrieves the application-specific value associated with an item.
GetItemPosition	Retrieves the position of a list view item.
GetItemRect	Retrieves the bounding rectangle for an item.
GetItemState	Retrieves the state of a list view item.
GetItemText	Retrieves the text of a list view item or subitem.
GetNextItem	Searches for a list view item with specified properties and with specified relationship to a given it em.
GetNextSelectedItem	Retrieves the index of a list view item position, and the position of the next selected list view item for iterating.
GetNumberOfWorkAreas	Retrieves the current number of working areas for a list view control.
GetOrigin	Retrieves the current view origin for a list view control.
GetSelectedCount	Retrieves the number of selected items in the list view control.
GetSelectionMark	Retrieves the selection mark of a list view control.
GetStringWidth	Determines the minimum column width necessary to display all of a given string.
GetSubItemRect	Retrieves the bounding rectangle of an item in a list view control.
GetTextBkColor	Retrieves the text background color of a list view control.
GetTextColor	Retrieves the text color of a list view control.
GetToolTips	Retrieves the tooltip control that the list view control uses to display tooltips.
GetTopIndex	Retrieves the index of the topmost visible item.
GetViewRect	Retrieves the bounding rectangle of all items in the list view control.
GetWorkAreas	Retrieves the current working areas of a list view control.
SetBkColor	Sets the background color of the list view control.

SetBkImage	Sets the current background image of a list view control.
SetCallbackMask	Sets the callback mask for a list view control.
SetCheck	Sets the current display status of the state image associated with an item.
SetColumn	Sets the attributes of a list view column.
SetColumnOrderArray	Sets the column order (left to right) of a list view control.
SetColumnWidth	Changes the width of a column in report view or list view.
SetExtendedStyle	Sets the current extended styles of a list view control.
SetHotCursor	Sets the cursor used when hot tracking is enabled for a list view control.
SetHotItem	Sets the current hot item of a list view control.
SetHoverTime	Sets the current hover time of a list view control.
SetIconSpacing	Sets the spacing between icons in a list view control.
SetImageList	Assigns an image list to a list view control.
SetItem	Sets some or all of a list view item's attributes.
SetItemCount	Prepares a list view control for adding a large number of items.
SetItemCountEx	Sets the item count for a virtual list view control.
SetItemData	Sets the item's application-specific value.
SetItemPosition	Moves an item to a specified position in a list view control.
SetItemState	Changes the state of an item in a list view control.
SetItemText	Changes the text of a list view item or subitem.
SetSelectionMark	Sets the selection mark of a list view control.
SetTextBkColor	Sets the background color of text in a list view control.
SetTextColor	Sets the text color of a list view control.
SetToolTips	Sets the tooltip control that the list view control will use to display tooltips.
SetWorkAreas	Sets the area where icons can be displayed in a list view control.
SubItemHitTest	Determines which list view item, if any, is at a given position.

Operations

Arrange	Aligns items on a grid.

CreateDragImage	Creates a drag image list for a specified item.
DeleteAllItems	Deletes all items from the control.
DeleteColumn	Deletes a column from the list view control.
DeleteItem	Deletes an item from the control.
EditLabel	Begins in-place editing of an item's text.
EnsureVisible	Ensures that an item is visible.
FindItem	Searches for a list view item having specified characteristics.
HitTest	Determines which list view item is at a specified position.
InsertColumn	Inserts a new column in a list view control.
InsertItem	Inserts a new item in a list view control.
RedrawItems	Forces a list view control to repaint a range of items.
Scroll	Scrolls the content of a list view control.
SortItems	Sorts list view items using an application-defined comparison function.
Update	Forces the control to repaint a specified item.

Overridables

DrawItem	Called when a visual aspect of an owner-draw control changes.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Concepts

CWnd Members

CListCtrl Member Functions

For information about the member functions in **CListCtrl**, see CListCtrl Members.

CListCtrl::ApproximateViewRect

Determines the width and height required to display the items of a list view control.

```
CSize ApproximateViewRect(
   CSize sz = CSize(-1,
   -1
),
   int iCount = -1
) const;
```

Parameters

sz

The proposed dimensions of the control, in pixels. If dimensions are not specified, the framework uses the current width or height values of the control.

iCount

Number of items to be displayed in the control. If this parameter is -1, the framework uses the total number of items currently in the control.

Return Value

A **CSize** object that contains the approximate width and height needed to display the items, in pixels.

Remarks

This member function implements the behavior of the Win32 macro, ListView_ApproximateViewRect, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetViewRect

Other Resources

CListCtrl Members

CListCtrl::Arrange

Repositions items in an icon view so that they align on a grid.

```
BOOL Arrange(
   UINT nCode
);
```

Parameters

nCode

Specifies the alignment style for the items. It can be one of the following values:

- **LVA_ALIGNLEFT** Aligns items along the left edge of the window.
- **LVA_ALIGNTOP** Aligns items along the top edge of the window.
- **LVA_DEFAULT** Aligns items according to the list view's current alignment styles (the default value).
- **LVA_SNAPTOGRID** Snaps all icons to the nearest grid position.

Return Value

Nonzero if successful; otherwise zero.

Remarks

The *nCode* parameter specifies the alignment style.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// Align all of the list view control items along the top
// of the window (the list view control must be in icon or
// small icon mode).
pmyListCtrl->Arrange(LVA_ALIGNTOP);
```

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::EnsureVisible

Other Resources

CListCtrl Members

CListCtrl::CancelEditLabel

Cancels item text editing operation.

```
void CancelEditLabel( );
```

Remarks

This member function emulates the functionality of the LVM_CANCELEDITLABEL message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::CListCtrl

Constructs a **CListCtrl** object.

```
CListCtrl( );
```

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::Create
CListCtrl::CreateEx

Other Resources

CListCtrl Members

CListCtrl::Create

Creates a list control and attaches it to a **CListCtrl** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the list control's style. Apply any combination of list control styles to the control. See List view window styles in the *Platform SDK* for a complete list of these styles. Set extended styles specific to a control using SetExtendedStyle.

rect

Specifies the list control's size and position. It can be either a **CRect** object or a RECT structure.

pParentWnd

Specifies the list control's parent window, usually a **CDialog**. It must not be **NULL**.

nID

Specifies the list control's ID.

Return Value

Nonzero if successful; otherwise zero.

Remarks

You construct a **CListCtrl** in two steps. First, call the constructor and then call **Create**, which creates the list view control and attaches it to the **CListCtrl** object.

To apply extended Windows styles to the list control object, call CreateEx instead of **Create**.

Example

```
// pParentWnd is a pointer to the parent window.
extern CWnd* pParentWnd;
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

pmyListCtrl->Create(
   WS_CHILD|WS_VISIBLE|WS_BORDER|LVS_REPORT,
   CRect(10,10,400,200), pParentWnd, 1);
```

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::CListCtrl
CListCtrl::CreateEx

Other Resources

CListCtrl Members

CListCtrl::CreateEx

Creates a control (a child window) and associates it with the **CListCtrl** object.

```
virtual BOOL CreateEx(
   DWORD dwExStyle,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwExStyle

Specifies the extended style of the control being created. For a list of extended Windows styles, see the *dwExStyle* parameter for CreateWindowEx in the *Platform SDK*.

dwStyle

Specifies the list control's style. Apply any combination of list control styles to the control. For a complete list of these styles, see List view window styles in the *Platform SDK*.

rect

A reference to a RECT structure describing the size and position of the window to be created, in client coordinates of *pParentWnd*.

pParentWnd

A pointer to the window that is the control's parent.

nID

The control's child-window ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Use **CreateEx** instead of Create to apply extended Windows styles, specified by the Windows extended style preface **WS_EX_**.

CreateEx creates the control with the extended Windows styles specified by *dwExStyle*. To set extended styles specific to a control, call SetExtendedStyle. For example, use **CreateEx** to set such styles as **WS_EX_CONTEXTHELP**, but use **SetExtendedStyle** to set such styles as **LVS_EX_FULLROWSELECT**. For more information, see the styles described in the topic Extended List View Styles in the *Platform SDK*.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::CListCtrl

Other Resources

CListCtrl Members

CListCtrl::CreateDragImage

Creates a drag image list for the item specified by *nItem*.

```
CImageList* CreateDragImage(
   int nItem,
   LPPOINT lpPoint
);
```

Parameters

nItem

Index of the item whose drag image list is to be created.

lpPoint

Address of a POINT structure that receives the initial location of the upper-left corner of the image, in view coordinates.

Return Value

A pointer to the drag image list if successful; otherwise **NULL**.

Remarks

The **CImageList** object is permanent, and you must delete it when finished. For example:

```
CImageList* pImageList = MyListCtrl.CreateDragImage(nItem, &point);
   ...
   ...
   delete pImageList;
```

See Also

Reference

CListCtrl Class
Hierarchy Chart
CImageList Class
CListCtrl::GetImageList

Other Resources

CListCtrl Members

CListCtrl::DeleteAllItems

Deletes all items from the list view control.

```
BOOL DeleteAllItems( );
```

Return Value

Nonzero if successful; otherwise zero.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// Delete all of the items from the list view control.
pmyListCtrl->DeleteAllItems();
ASSERT(pmyListCtrl->GetItemCount() == 0);
```

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::InsertItem
CListCtrl::DeleteItem

Other Resources

CListCtrl Members

CListCtrl::DeleteColumn

Deletes a column from the list view control.

```
BOOL DeleteColumn(
   int nCol
);
```

Parameters

nCol

Index of the column to be deleted.

Return Value

Nonzero if successful; otherwise zero.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

int nColumnCount = pmyListCtrl->GetHeaderCtrl()->GetItemCount();

// Delete all of the columns.
for (int i=0;i < nColumnCount;i++)
{
   pmyListCtrl->DeleteColumn(0);
}
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::InsertColumn

CListCtrl::DeleteAllItems

Other Resources

CListCtrl Members

CListCtrl::DeleteItem

Deletes an item from a list view control.

```
BOOL DeleteItem(
   int nItem
);
```

Parameters

nItem

Specifies the index of the item to be deleted.

Return Value

Nonzero if successful; otherwise zero.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

int nCount = pmyListCtrl->GetItemCount();

// Delete all of the items from the list view control.
for (int i=0;i < nCount;i++)
{
   pmyListCtrl->DeleteItem(0);
}
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::InsertItem

CListCtrl::DeleteAllItems

Other Resources

CListCtrl Members

CListCtrl::DrawItem

Called by the framework when a visual aspect of an owner-draw list view control changes.

```
virtual void DrawItem(
   LPDRAWITEMSTRUCT lpDrawItemStruct
);
```

Parameters

lpDrawItemStruct

A long pointer to a **DRAWITEMSTRUCT** structure that contains information about the type of drawing required.

Remarks

The **itemAction** member of the DRAWITEMSTRUCT structure defines the drawing action that is to be performed.

By default, this member function does nothing. Override this member function to implement drawing for an owner-draw **CListCtrl** object.

The application should restore all graphics device interface (GDI) objects selected for the display context supplied in *lpDrawItemStruct* before this member function terminates.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CWnd::OnDrawItem

Other Resources

CListCtrl Members

CListCtrl::EditLabel

Begins in-place editing of an item's text.

```
CEdit* EditLabel(
   int nItem
);
```

Parameters

nItem

Index of the list view item that is to be edited.

Return Value

If successful, a pointer to the **CEdit** object that is used to edit the item text; otherwise **NULL**.

Remarks

A list view control that has the **LVS_EDITLABELS** window style enables a user to edit item labels in place. The user begins editing by clicking the label of an item that has the focus.

Use this function to begin in-place editing of the specified list view item's text.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// Make sure the focus is set to the list view control.
pmyListCtrl->SetFocus();

// Show the edit control on the label of the first
// item in the list view control.
CEdit* pmyEdit = pmyListCtrl->EditLabel(1);
ASSERT(pmyEdit != NULL);
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::GetEditControl

Other Resources

CListCtrl Members

CListCtrl::EnableGroupView

Enables or disables whether the items in a list view control display as a group.

```
LRESULT EnableGroupView(
   BOOL fEnable
);
```

Parameters

fEnable

Indicates whether to enable a listview control to group displayed items. **TRUE** to enable grouping; **FALSE** to disable it.

Return Value

Returns one of the following values:

- **0** The ability to display list view items as a group is already enabled or disabled.
- **1** The state of the control was successfully changed.
- **-1** The operation failed.

Remarks

This member function emulates the functionality of the LVM_ENABLEGROUPVIEW message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::EnsureVisible

Ensures that a list view item is at least partially visible.

```
BOOL EnsureVisible(
   int nItem,
   BOOL bPartialOK
);
```

Parameters

nItem

Index of the list view item that is to be visible.

bPartialOK

Specifies whether partial visibility is acceptable.

Return Value

Nonzero if successful; otherwise zero.

Remarks

The list view control is scrolled if necessary. If the *bPartialOK* parameter is nonzero, no scrolling occurs if the item is partially visible.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// Ensure that the last item is visible.
int nCount = pmyListCtrl->GetItemCount();
if (nCount > 0)
   pmyListCtrl->EnsureVisible(nCount-1, FALSE);
```

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::Scroll

Other Resources

CListCtrl Members

CListCtrl::FindItem

Searches for a list view item having specified characteristics.

```
int FindItem(
   LVFINDINFO* pFindInfo,
   int nStart = -1
) const;
```

Parameters

pFindInfo

A pointer to an LVFINDINFO structure containing information about the item to be searched for.

nStart

Index of the item to begin the search with, or -1 to start from the beginning. The item at *nStart* is excluded from the search if *nStart* is not equal to -1.

Return Value

The index of the item if successful or -1 otherwise.

Remarks

The *pFindInfo* parameter points to an **LVFINDINFO** structure, which contains information used to search for a list view item.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;
// The string to match.
extern LPCTSTR lpszmyString;

LVFINDINFO info;
int nIndex;

info.flags = LVFI_PARTIAL|LVFI_STRING;
info.psz = lpszmyString;

// Delete all of the items that begin with the string lpszmyString.
while ((nIndex=pmyListCtrl->FindItem(&info)) != -1)
{
   pmyListCtrl->DeleteItem(nIndex);
}
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::SortItems

Other Resources

CListCtrl Members

CListCtrl::GetBkColor

Retrieves the background color of a list view control.

```
COLORREF GetBkColor( ) const;
```

Return Value

A 32-bit value used to specify an RGB color.

Example

See the example for CListCtrl::SetBkColor.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::SetBkColor
COLORREF

Other Resources

CListCtrl Members

CListCtrl::GetBkImage

Retrieves the current background image of a list view control.

```
BOOL GetBkImage(
   LVBKIMAGE* plvbkImage
) const;
```

Parameters

plvbkImage

A pointer to an **LVBKIMAGE** structure containing the current background image of the list view.

Return Value

Returns nonzero if successful, or zero otherwise.

Remarks

This member function implements the behavior of the Win32 macro, ListView_GetBkImage, as described in the *Platform SDK*.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;
LVBKIMAGE bki;

// If no background image is set for the list view control use
// the Microsoft homepage image as the background image.
if (pmyListCtrl->GetBkImage(&bki) && (bki.ulFlags == LVBKIF_SOURCE_NONE))
{
   pmyListCtrl->SetBkImage(
   TEXT("http://www.microsoft.com/library/images/gifs/homepage/microsoft.gif"),
      TRUE);
}
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::SetBkImage

Other Resources

CListCtrl Members

CListCtrl::GetCallbackMask

Retrieves the callback mask for a list view control.

```
UINT GetCallbackMask( ) const;
```

Return Value

The list view control's callback mask.

Remarks

A "callback item" is a list view item for which the application — rather than the control — stores the text, icon, or both. Although a list view control can store these attributes for you, you may want to use callback items if your application already maintains some of this information. The callback mask specifies which item state bits are maintained by the application, and it applies to the whole control rather than to a specific item. The callback mask is zero by default, meaning that the control tracks all item states. If an application uses callback items or specifies a nonzero callback mask, it must be able to supply list view item attributes on demand.

Example

See the example for CListCtrl::SetCallbackMask.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::SetCallbackMask

Other Resources

CListCtrl Members

CListCtrl::GetCheck

Retrieves the current display status of the state image that is associated with an item.

```
BOOL GetCheck(
   int nItem
) const;
```

Parameters

nItem

The zero-based index of a list control item.

Return Value

Nonzero if the item is selected, otherwise 0.

Remarks

This member function implements the behavior of the Win32 macro, ListView_GetCheckState, as described in the *Platform SDK*.

Example

See the example for CListCtrl::SetCheck.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::SetCheck

Other Resources

CListCtrl Members

CListCtrl::GetColumn

Retrieves the attributes of a list view control's column.

```
BOOL GetColumn(
   int nCol,
   LVCOLUMN* pColumn
) const;
```

Parameters

nCol

Index of the column whose attributes are to be retrieved.

pColumn

Address of an LVCOLUMN structure that specifies the information to retrieve and receives information about the column. The **mask** member specifies which column attributes to retrieve. If the **mask** member specifies the **LVCF_TEXT** value, the **pszText** member must contain the address of the buffer that receives the item text and the **cchTextMax** member must specify the size of the buffer.

Return Value

Nonzero if successful; otherwise zero.

Remarks

The **LVCOLUMN** structure contains information about a column in report view.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;
LVCOLUMN col;

col.mask = LVCF_WIDTH;

// Double the column width of the first column.
if (pmyListCtrl->GetColumn(0, &col))
{
   col.cx *= 2;
   pmyListCtrl->SetColumn(0, &col);
}
```

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::SetColumn
CListCtrl::GetColumnWidth

Other Resources

CListCtrl Members

CListCtrl::GetColumnOrderArray

Retrieves the column order (left to right) of a list view control.

```
BOOL GetColumnOrderArray(
   LPINT piArray,
   int iCount = -1
);
```

Parameters

piArray

A pointer to a buffer that will contain the index values of the columns in the list view control. The buffer must be large enough to contain the total number of columns in the list view control.

iCount

Number of columns in the list view control. If this parameter is -1, the number of columns is automatically retrieved by the framework.

Return Value

Nonzero if successful; otherwise zero.

Remarks

This member function implements the behavior of the Win32 macro, ListView_GetColumnOrderArray, as described in the *Platform SDK*.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// Reverse the order of the columns in the list view control
// (i.e. make the first column the last, the last column
// the first, and so on...).
CHeaderCtrl* pHeaderCtrl = pmyListCtrl->GetHeaderCtrl();

if (pHeaderCtrl != NULL)
{
   int  nColumnCount = pHeaderCtrl->GetItemCount();
   LPINT pnOrder = (LPINT) malloc(nColumnCount*sizeof(int));
   ASSERT(pnOrder != NULL);

   pmyListCtrl->GetColumnOrderArray(pnOrder, nColumnCount);

   int i, j, nTemp;
   for (i=0,j=nColumnCount-1;i < j;i++,j--)
   {
      nTemp = pnOrder[i];
      pnOrder[i] = pnOrder[j];
      pnOrder[j] = nTemp;
   }

   pmyListCtrl->SetColumnOrderArray(nColumnCount, pnOrder);
   free(pnOrder);
}
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::SetColumnOrderArray

Other Resources

CListCtrl Members

CListCtrl::GetColumnWidth

Retrieves the width of a column in report view or list view.

```
int GetColumnWidth(
   int nCol
) const;
```

Parameters

nCol

Specifies the index of the column whose width is to be retrieved.

Return Value

The width, in pixels, of the column specified by *nCol*.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// Double the column width of the first column.
int nWidth = pmyListCtrl->GetColumnWidth(0);
pmyListCtrl->SetColumnWidth(0, 2*nWidth);
```

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::SetColumnWidth
CListCtrl::GetColumn

Other Resources

CListCtrl Members

CListCtrl::GetCountPerPage

Calculates the number of items that can fit vertically in the visible area of a list view control when in list view or report view.

```
int GetCountPerPage( ) const;
```

Return Value

The number of items that can fit vertically in the visible area of a list view control when in list view or report view.

Example

See the example for CListCtrl::GetTopIndex.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetTopIndex

Other Resources

CListCtrl Members

CListCtrl::GetEditControl

Retrieves the handle of the edit control used to edit a list view item's text.

```
CEdit* GetEditControl( ) const;
```

Return Value

If successful, a pointer to the CEdit object that is used to edit the item text; otherwise **NULL**.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;
// The string replacing the text in the edit control.
extern LPCTSTR lpszmyString;

// If possible, replace the text in the label edit control.
CEdit* pEdit = pmyListCtrl->GetEditControl();

if (pEdit != NULL)
{
   pEdit->SetWindowText(lpszmyString);
}
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::EditLabel

Other Resources

CListCtrl Members

CListCtrl::GetExtendedStyle

Retrieves the current extended styles of a list view control.

```
DWORD GetExtendedStyle( );
```

Return Value

A combination of the extended styles currently in use by the list view control. For a descriptive list of these extended styles, see the Extended List View Styles topic in the *Platform SDK*.

Remarks

This member function implements the behavior of the Win32 macro, ListView_GetExtendedListViewStyle, as described in the *Platform SDK*.

Example

See the example for CListCtrl::SetExtendedStyle.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::SetExtendedStyle

Other Resources

CListCtrl Members

CListCtrl::GetFirstSelectedItemPosition

Gets the position of the first selected item in the list view control.

```
POSITION GetFirstSelectedItemPosition( ) const;
```

Return Value

A **POSITION** value that can be used for iteration or object pointer retrieval; **NULL** if no items are selected.

Example

The following code sample demonstrates the usage of this function.

```
CListCtrl* pListCtrl = (CListCtrl*) GetDlgItem(IDC_YOURLISTCONTROL);
ASSERT(pListCtrl != NULL);

POSITION pos = pList->GetFirstSelectedItemPosition();
if (pos == NULL)
   TRACE0("No items were selected!\n");
else
{
   while (pos)
   {
      int nItem = pList->GetNextSelectedItem(pos);
      TRACE1("Item %d was selected!\n", nItem);
      // you could do your own processing on nItem here
   }
}
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CImageList Class

CListCtrl::GetNextSelectedItem

Other Resources

CListCtrl Members

CListCtrl::GetGroupInfo

Gets the information for a specified group of the list view control.

```
int GetGroupInfo(
   int iGroupId,
   PLVGROUP pgrp
) const;
```

Parameters

iGroupId

The identifier of the group whose information is to be retrieved.

pgrp

A pointer to the LVGROUP containing information on the group specified.

Return Value

Returns the ID of the group if successful, or -1 otherwise.

Remarks

This member function emulates the functionality of the LVM_GETGROUPINFO message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::SetGroupInfo

Other Resources

CListCtrl Members

CListCtrl::GetGroupMetrics

Retrieves the metrics of a group.

```
void GetGroupMetrics(
   PLVGROUPMETRICS pGroupMetrics
) const;
```

Parameters

pGroupMetrics

A pointer to a LVGROUPMETRICS containing the group metrics information.

Remarks

This member function emulates the functionality of the LVM_GETGROUPMETRICS message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::SetGroupMetrics

Other Resources

CListCtrl Members

CListCtrl::GetHeaderCtrl

Retrieves the header control of a list view control.

```
CHeaderCtrl* GetHeaderCtrl( );
```

Return Value

A pointer to the header control, used by the list view control.

Remarks

This member function implements the behavior of the Win32 macro, ListView_GetHeader, as described in the *Platform SDK*.

Example

See the example for CListCtrl::GetColumnOrderArray.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CHeaderCtrl Class

Other Resources

CListCtrl Members

CListCtrl::GetHotCursor

Retrieves the cursor used when hot tracking is enabled for a list view control.

```
HCURSOR GetHotCursor( );
```

Return Value

The handle to the current hot cursor resource being used by the list view control.

Remarks

This member function implements the behavior of the Win32 macro, ListView_GetHotCursor, as described in the *Platform SDK*. The hot cursor, only visible when hover selection is enabled, appears when the cursor passes over any list view item. Hover selection is enabled by setting the **LVS_EX_TRACKSELECT** extended style.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// Set the hot cursor to be the system app starting cursor.
HCURSOR hCursor = ::LoadCursor(NULL, IDC_APPSTARTING);
pmyListCtrl->SetHotCursor(hCursor);
ASSERT(pmyListCtrl->GetHotCursor() == hCursor);
```

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::SetHotCursor
CListCtrl::GetHotItem

Other Resources

CListCtrl Members

CListCtrl::GetHotItem

Retrieves the list view item currently under the cursor.

```
int GetHotItem( );
```

Return Value

The index of the current hot item of the list view control.

Remarks

This member function implements the behavior of the Win32 macro, ListView_GetHotItem, as described in the *Platform SDK*. The hot item is defined as the currently selected item when hot tracking (and hover selection) is enabled.

If hot tracking is enabled, when a user pauses over a list view item, the item label is automatically highlighted without the use of a mouse button.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// Set the hot item to the first item only if no other item is
// highlighted.
if (pmyListCtrl->GetHotItem() == -1)
   pmyListCtrl->SetHotItem(0);
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::GetHotCursor

Other Resources

CListCtrl Members

CListCtrl::GetHoverTime

Retrieves the current hover time of a list view control.

```
DWORD GetHoverTime( ) const;
```

Return Value

Returns the delay, in milliseconds, which the mouse cursor must hover over an item before it is selected. If the return value is -1, then the hover time is the default hover time.

Remarks

This member function implements the behavior of the Win32 macro, ListView_GetHoverTime, as described in the *Platform SDK*.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// Double the hover time, if possible.
DWORD dwTime = pmyListCtrl->GetHoverTime();
if (dwTime != -1)
   pmyListCtrl->SetHoverTime(2*dwTime);
```

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::SetHoverTime
CListCtrl::GetHotCursor

Other Resources

CListCtrl Members

CListCtrl::GetImageList

Retrieves the handle of an image list used for drawing list view items.

```
CImageList* GetImageList(
   int nImageList
) const;
```

Parameters

nImageList

Value specifying which image list to retrieve. It can be one of these values:

- **LVSIL_NORMAL** Image list with large icons.
- **LVSIL_SMALL** Image list with small icons.
- **LVSIL_STATE** Image list with state images.

Return Value

A pointer to the image list used for drawing list view items.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;
// The new image list of the list view control.
extern CImageList* pmyImageList;

ASSERT(pmyListCtrl->GetImageList(LVSIL_NORMAL) == NULL);

pmyListCtrl->SetImageList(pmyImageList, LVSIL_NORMAL);
ASSERT(pmyListCtrl->GetImageList(LVSIL_NORMAL) == pmyImageList);
```

See Also

Reference

CListCtrl Class
Hierarchy Chart
CImageList Class
CListCtrl::SetImageList

Other Resources

CListCtrl Members

CListCtrl::GetInsertMark

Retrieves the current position of the insertion mark.

```
BOOL GetInsertMark(
   LPLVINSERTMARK lvim
) const;
```

Parameters

lvim

A pointer to an LVINSERTMARK structure containing the information for the insert mark.

Return Value

Returns **TRUE** if successful, or **FALSE** otherwise. **FALSE** is returned if the size in the **cbSize** member of the **LVINSERTMARK** structure does not equal the actual size of the structure.

Remarks

This member function emulates the functionality of the LVM_GETINSERTMARK message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Other Resources

CListCtrl Members

CListCtrl::GetInsertMarkColor

Retrieves the current color of the insertion mark.

```
COLORREF GetInsertMarkColor( ) const;
```

Return Value

Returns a COLORREF structure that contains the color of the insertion point.

Remarks

This member function emulates the functionality of the LVM_GETINSERTMARKCOLOR message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Other Resources

CListCtrl Members

CListCtrl::GetInsertMarkRect

Retrieves the rectangle that bounds the insertion point.

```
int GetInsertMarkRect(
   LPRECT pRect
) const;
```

Parameters

pRect

Pointer to a **RECT** structure that contains the coordinates of a rectangle that bounds the insertion point.

Return Value

Returns one of the following values:

- **0** No insertion point found.
- **1** Insertion point found.

Remarks

This member function emulates the functionality of the LVM_GETINSERTMARKRECT message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Other Resources

CListCtrl Members

CListCtrl::GetItem

Retrieves some or all of a list view item's attributes.

```
BOOL GetItem(
   LVITEM* pItem
) const;
```

Parameters

pItem

Pointer to an LVITEM structure that receives the item's attributes.

Return Value

Nonzero if successful; otherwise zero.

Remarks

The **LVITEM** structure specifies or receives the attributes of a list view item.

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::SetItem

Other Resources

CListCtrl Members

CListCtrl::GetItemCount

Retrieves the number of items in a list view control.

```
int GetItemCount( ) const;
```

Return Value

The number of items in the list view control.

Example

See the example for CListCtrl::DeleteItem.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::SetItemCount
CListCtrl::GetSelectedCount

Other Resources

CListCtrl Members

CListCtrl::GetItemData

Retrieves the 32-bit application-specific value associated with the item specified by *nItem*.

```
DWORD_PTR GetItemData(
   int nItem
) const;
```

Parameters

nItem

Index of the list item whose data is to be retrieved.

Return Value

A 32-bit application-specific value associated with the specified item.

Remarks

This value is the **lParam** member of the LVITEM structure, as described in the *Platform SDK*

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// If any item's data is equal to zero then reset it to -1.
for (int i=0;i < pmyListCtrl->GetItemCount();i++)
{
   if (pmyListCtrl->GetItemData(i) == 0)
   {
      pmyListCtrl->SetItemData(i, (DWORD) -1);
   }
}
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::SetItemData

Other Resources

CListCtrl Members

CListCtrl::GetItemPosition

Retrieves the position of a list view item.

```
BOOL GetItemPosition(
   int nItem,
   LPPOINT lpPoint
) const;
```

Parameters

nItem

The index of the item whose position is to be retrieved.

lpPoint

Address of a POINT structure that receives the position of the item's upper-left corner, in view coordinates.

Return Value

Nonzero if successful; otherwise zero.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;
POINT pt;

// Move all items in the list control 100 pixels to the right.
UINT i, nCount = pmyListCtrl->GetItemCount();

for (i=0;i < nCount;i++)
{
   pmyListCtrl->GetItemPosition(i, &pt);
   pt.x += 100;
   pmyListCtrl->SetItemPosition(i, pt);
}
```

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::SetItemPosition
CListCtrl::GetOrigin

Other Resources

CListCtrl Members

CListCtrl::GetItemRect

Retrieves the bounding rectangle for all or part of an item in the current view.

```
BOOL GetItemRect(
   int nItem,
   LPRECT lpRect,
   UINT nCode
) const;
```

Parameters

nItem

The index of the item whose position is to be retrieved.

lpRect

Address of a RECT structure that receives the bounding rectangle.

nCode

Portion of the list view item for which to retrieve the bounding rectangle. It can be one of these values:

- **LVIR_BOUNDS** Returns the bounding rectangle of the entire item, including the icon and label.
- **LVIR_ICON** Returns the bounding rectangle of the icon or small icon.
- **LVIR_LABEL** Returns the bounding rectangle of the item text.

Return Value

Nonzero if successful; otherwise zero.

Example

```
// OnClick in the handler for the NM_CLICK notification
void CMyListCtrl::OnClick(NMHDR* pNMHDR, LRESULT* pResult)
{
   // Get the current mouse location and convert it to client
   // coordinates.
   CPoint pos( ::GetMessagePos() );
   ScreenToClient(&pos);

   // Get indexes of the first and last visible items in
   // the listview control.
   int index = GetTopIndex();
   int last_visible_index = index + GetCountPerPage();
   if (last_visible_index > GetItemCount())
       last_visible_index = GetItemCount();

   // Loop until number visible items has been reached.
   while (index <= last_visible_index)
   {
       // Get the bounding rectangle of an item. If the mouse
       // location is within the bounding rectangle of the item,
       // you know you have found the item that was being clicked.
       CRect r;
       GetItemRect(index, &r, LVIR_BOUNDS);
       if (r.PtInRect(pt))
           {
               UINT flag = LVIS_SELECTED | LVIS_FOCUSED;
               SetItemState(index, flag, flag);
               break;
          }

       // Get the next item in listview control.
```

```
        index++;
    }

    *pResult = 0;
}
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::GetItemPosition

CListCtrl::SetItemPosition

CListCtrl::GetOrigin

Other Resources

CListCtrl Members

CListCtrl::GetItemState

Retrieves the state of a list view item.

```
UINT GetItemState(
   int nItem,
   UINT nMask
) const;
```

Parameters

nItem

The index of the item whose state is to be retrieved.

nMask

Mask specifying which of the item's state flags to return.

Return Value

The state flags for the specified list view item.

Remarks

An item's state is specified by the **state** member of the LVITEM structure, as described in the *Platform SDK*. When you specify or change an item's state, the **stateMask** member specifies which state bits you want to change.

Example

See the example for CListCtrl::GetTopIndex.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::SetItemState
CListCtrl::GetItem

Other Resources

CListCtrl Members

CListCtrl::GetItemText

Retrieves the text of a list view item or subitem.

```
int GetItemText(
   int nItem,
   int nSubItem,
   LPTSTR lpszText,
   int nLen
) const;
CString GetItemText(
   int nItem,
   int nSubItem
) const;
```

Parameters

nItem

The index of the item whose text is to be retrieved.

nSubItem

Specifies the subitem whose text is to be retrieved.

lpszText

Pointer to a string that is to receive the item text.

nLen

Length of the buffer pointed to by *lpszText*.

Return Value

The version returning **int** returns the length of the retrieved string.

The version returning a **CString** returns the item text.

Remarks

If *nSubItem* is zero, this function retrieves the item label; if *nSubItem* is nonzero, it retrieves the text of the subitem. For more information on the subitem argument, see the discussion of the LVITEM structure in the *Platform SDK*.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetItem

Other Resources

CListCtrl Members

CListCtrl::GetNextItem

Searches for a list view item that has the specified properties and that bears the specified relationship to a given item.

```
int GetNextItem(
   int nItem,
   int nFlags
) const;
```

Parameters

nItem

Index of the item to begin the searching with, or -1 to find the first item that matches the specified flags. The specified item itself is excluded from the search.

nFlags

Geometric relation of the requested item to the specified item, and the state of the requested item. The geometric relation can be one of these values:

- **LVNI_ABOVE** Searches for an item that is above the specified item.
- **LVNI_ALL** Searches for a subsequent item by index (the default value).
- **LVNI_BELOW** Searches for an item that is below the specified item.
- **LVNI_TOLEFT** Searches for an item to the left of the specified item.
- **LVNI_TORIGHT** Searches for an item to the right of the specified item.

The state can be zero, or it can be one or more of these values:

- **LVNI_DROPHILITED** The item has the **LVIS_DROPHILITED** state flag set.
- **LVNI_FOCUSED** The item has the **LVIS_FOCUSED** state flag set.
- **LVNI_SELECTED** The item has the **LVIS_SELECTED** state flag set.

If an item does not have all of the specified state flags set, the search continues with the next item.

Return Value

The index of the next item if successful, or -1 otherwise.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetItem

Other Resources

CListCtrl Members

CListCtrl::GetNextSelectedItem

Gets the index of the list item identified by *pos*, then sets *pos* to the **POSITION** value.

```
int GetNextSelectedItem(
   POSITION& pos
) const;
```

Parameters

pos

A reference to a **POSITION** value returned by a previous call to **GetNextSelectedItem** or **GetFirstSelectedItemPosition**. The value is updated to the next position by this call.

Return Value

The index of the list item identified by *pos*.

Remarks

You can use **GetNextSelectedItem** in a forward iteration loop if you establish the initial position with a call to **GetFirstSelectedItemPosition**.

You must ensure that your **POSITION** value is valid. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

Example

The following code sample demonstrates the usage of this function.

```
CListCtrl* pListCtrl = (CListCtrl*) GetDlgItem(IDC_YOURLISTCONTROL);
ASSERT(pListCtrl != NULL);

POSITION pos = pList->GetFirstSelectedItemPosition();
if (pos == NULL)
   TRACE0("No items were selected!\n");
else
{
   while (pos)
   {
      int nItem = pList->GetNextSelectedItem(pos);
      TRACE1("Item %d was selected!\n", nItem);
      // you could do your own processing on nItem here
   }
}
```

See Also

Reference

CListCtrl Class
Hierarchy Chart
CImageList Class
CListCtrl::GetFirstSelectedItemPosition

Other Resources

CListCtrl Members

CListCtrl::GetNumberOfWorkAreas

Retrieves the current number of working areas for a list view control.

```
UINT GetNumberOfWorkAreas( ) const;
```

Return Value

Not used at this time.

Remarks

This member function implements the behavior of the Win32 macro, ListView_GetNumberOfWorkAreas, as described in the *Platform SDK*.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

UINT i, uCount = pmyListCtrl->GetNumberOfWorkAreas();
LPRECT lpRects = (LPRECT) malloc(uCount*sizeof(RECT));
ASSERT(lpRects != NULL);

// Dump all of the work area dimensions.
pmyListCtrl->GetWorkAreas(uCount, lpRects);

for (i=0;i < uCount;i++)
{
   TRACE (TEXT("Work area %d; left = %d, top = %d, right = %d,
      bottom = %d\r\n"),
      i, lpRects[i].left, lpRects[i].top, lpRects[i].right,
      lpRects[i].bottom);
}

free(lpRects);
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::GetWorkAreas

Other Resources

CListCtrl Members

CListCtrl::GetOutlineColor

Retrieves the color of the border of a list view control.

```
COLORREF GetOutlineColor( ) const;
```

Return Value

Returns a COLORREF structure containing the outline color.

Remarks

This member function emulates the functionality of the LVM_GETOUTLINECOLOR message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::GetOrigin

Retrieves the current view origin for a list view control.

```
BOOL GetOrigin(
   LPPOINT lpPoint
) const;
```

Parameters

lpPoint

Address of a POINT structure that receives the view origin.

Return Value

Nonzero if successful; otherwise zero. However, if the control is in report view, the return value is always zero.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetItemPosition
CListCtrl::SetItemPosition

Other Resources

CListCtrl Members

CListCtrl::GetSelectedColumn

Retrieves the index of the currently-selected column in the list control.

```
UINT GetSelectedColumn( ) const;
```

Return Value

The index of the selected column.

Remarks

This member function emulates the functionality of the LVM_GETSELECTEDCOLUMN message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl::SetSelectedColumn
CListCtrl Class
Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::GetSelectedCount

Retrieves the number of selected items in the list view control.

```
UINT GetSelectedCount( ) const;
```

Return Value

The number of selected items in the list view control.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

UINT i, uSelectedCount = pmyListCtrl->GetSelectedCount();
int  nItem = -1;

// Update all of the selected items.
if (uSelectedCount > 0)
{
   for (i=0;i < uSelectedCount;i++)
   {
      nItem = pmyListCtrl->GetNextItem(nItem, LVNI_SELECTED);
      ASSERT(nItem != -1);
      pmyListCtrl->Update(nItem);
   }
}
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::SetItemCount

CListCtrl::GetItemCount

Other Resources

CListCtrl Members

CListCtrl::GetSelectionMark

Retrieves the selection mark of a list view control.

```
int GetSelectionMark( );
```

Return Value

The zero-based selection mark, or -1 if there is no selection mark.

Remarks

This member function implements the behavior of the Win32 macro, ListView_GetSelectionMark, as described in the *Platform SDK*.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// Set the selection mark to the first item only if no other item is
// selected.
if (pmyListCtrl->GetSelectionMark() == -1)
      pmyListCtrl->SetSelectionMark(0);
```

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::SetSelectionMark

Other Resources

CListCtrl Members

CListCtrl::GetStringWidth

Determines the minimum column width necessary to display all of a given string.

```
int GetStringWidth(
   LPCTSTR lpsz
) const;
```

Parameters

lpsz

Address of a null-terminated string whose width is to be determined.

Return Value

The width, in pixels, of the string pointed to by *lpsz*.

Remarks

The returned width takes into account the control's current font and column margins, but not the width of a small icon.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

CString strColumn;
int nWidth;

// Insert six columns in the list view control. Make the width of
// the column be the width of the column header plus 50%.
for (int i=0;i < 6;i++)
{
   strColumn.Format(TEXT("column %d"), i);
   nWidth = 3*pmyListCtrl->GetStringWidth(strColumn)/2;
   pmyListCtrl->InsertColumn(i, strColumn, LVCFMT_LEFT, nWidth);
}
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::GetColumnWidth

Other Resources

CListCtrl Members

CListCtrl::GetSubItemRect

Retrieves the bounding rectangle of an item in a list view control.

```
BOOL GetSubItemRect(
   int iItem,
   int iSubItem,
   int nArea,
   CRect& ref
);
```

Parameters

iItem

Index of the subitem's parent item.

iSubItem

The one-based index of the subitem.

nArea

Determines the portion of the bounding rectangle (of the list view subitem) to be retrieved. The portion (icon, label, or both) of the bounding rectangle is specified by applying the bitwise OR operator to one or more of the following values:

- **LVIR_BOUNDS** Returns the bounding rectangle of the entire item, including the icon and label.
- **LVIR_ICON** Returns the bounding rectangle of the icon or small icon.
- **LVIR_LABEL** Returns the bounding rectangle of the entire item, including the icon and label. This is identical to **LVIR_BOUNDS**.

ref

Reference to a CRect object that contains the coordinates of the subitem's bounding rectangle.

Return Value

Nonzero if successful; otherwise zero.

Remarks

This member function implements the behavior of the Win32 macro, ListView_GetSubItemRect, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::GetTextBkColor

Retrieves the text background color of a list view control.

```
COLORREF GetTextBkColor( ) const;
```

Return Value

A 32-bit value used to specify an RGB color.

Example

See the example for CListCtrl::SetTextBkColor.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::SetTextBkColor
CListCtrl::GetTextColor
COLORREF

Other Resources

CListCtrl Members

CListCtrl::GetTextColor

Retrieves the text color of a list view control.

```
COLORREF GetTextColor( ) const;
```

Return Value

A 32-bit value used to specify an RGB color.

Example

See the example for CListCtrl::SetTextColor.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::SetTextColor
CListCtrl::GetTextBkColor
COLORREF

Other Resources

CListCtrl Members

CListCtrl::GetTileInfo

Retrieves information about a tile in a list view control.

```
BOOL GetTileInfo(
   PLVTILEINFO pti
) const;
```

Parameters

pti

A pointer to an LVTILEINFO structure that receives the tile information.

Return Value

The return value is not used.

Remarks

This member function emulates the functionality of the LVM_GETTILEINFO message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::GetTileViewInfo

Retrieves information about a list view control in tile view.

```
BOOL GetTileViewInfo(
   PLVTILEVIEWINFO ptvi
) const;
```

Parameters

ptvi

A pointer to an LVTILEVIEWINFO structure that receives the retrieved information.

Return Value

The return value is not used.

Remarks

This member function emulates the functionality of the LVM_GETTILEVIEWINFO message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::GetToolTips

Retrieves the tooltip control that the list view control uses to display tooltips.

```
CToolTipCtrl* GetToolTips( ) const;
```

Return Value

A pointer to a CToolTipCtrl object to be used by the list control. If the Create member function uses the style **LVS_NOTOOLTIPS**, no tooltips are used, and **NULL** is returned.

Remarks

This member function implements the behavior of the Win32 message LVM_GETTOOLTIPS, as described in the *Platform SDK*. The MFC implementation of **GetToolTips** returns a **CToolTipCtrl** object, which is used by the list control, rather than a handle to a tooltip control.

Example

```
// The pointer to my list control.
extern CListCtrl* pmyListCtrl;
// A pointer to a tooltips control.
extern CToolTipCtrl* pmyToolTip;

// If the list control does not have a tooltips control,
// then use pmyToolTip as the tootips for the list control.
if (pmyListCtrl->GetToolTips() == NULL)
{
   pmyListCtrl->SetToolTips(pmyToolTip);
}
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::SetToolTips

Other Resources

CListCtrl Members

CListCtrl::GetTopIndex

Retrieves the index of the topmost visible item when in list view or report view.

```
int GetTopIndex( ) const;
```

Return Value

The index of the topmost visible item.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// Make sure the focus is set to the list view control.
pmyListCtrl->SetFocus();

// Select all of the items that are completely visible.
int n = pmyListCtrl->GetTopIndex();
int nLast = n + pmyListCtrl->GetCountPerPage();

for (;n < nLast;n++)
{
   pmyListCtrl->SetItemState(n, LVIS_SELECTED, LVIS_SELECTED);
   ASSERT(pmyListCtrl->GetItemState(n, LVIS_SELECTED) == LVIS_SELECTED);
}
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::GetCountPerPage

Other Resources

CListCtrl Members

CListCtrl::GetView

Gets the view of the list view control.

```
DWORD GetView( ) const;
```

Return Value

The current view of the list view control.

Remarks

This member function emulates the functionality of the LVM_GETVIEW message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::SetView

Other Resources

CListCtrl Members

CListCtrl::GetViewRect

Retrieves the bounding rectangle of all items in the list view control.

```
BOOL GetViewRect(
   LPRECT lpRect
) const;
```

Parameters

lpRect

Address of a RECT structure.

Return Value

Nonzero if successful; otherwise zero.

Remarks

The list view must be in icon view or small icon view.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetTopIndex

Other Resources

CListCtrl Members

CListCtrl::GetWorkAreas

Retrieves the current working areas of a list view control.

```
void GetWorkAreas(
   int nWorkAreas,
   LPRECT prc
) const;
```

Parameters

nWorkAreas

The number of **RECT** structures contained in the *prc* array.

prc

A pointer to an array of **RECT** structures (or CRect objects) that receive the working areas of the list view control. Values in these structures are in client coordinates.

Remarks

This member function implements the behavior of the Win32 macro, ListView_GetWorkAreas, as described in the *Platform SDK*.

Example

See the example for CListCtrl::GetNumberOfWorkAreas.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetNumberOfWorkAreas

Other Resources

CListCtrl Members

CListCtrl::HasGroup

Determines if the list view control has the specified group.

```
BOOL HasGroup(
   int iGroupId
) const;
```

Parameters

iGroupId

The identifier of the group being requested.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This member function emulates the functionality of the LVM_HASGROUP message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::HitTest

Determines which list view item, if any, is at a specified position.

```
int HitTest(
   LVHITTESTINFO* pHitTestInfo
) const;
int HitTest(
   CPoint pt,
   UINT* pFlags = NULL
) const;
```

Parameters

pHitTestInfo

Address of an **LVHITTESTINFO** structure that contains the position to hit test and that receives information about the results of the hit test.

pt

Point to be tested.

pFlags

Pointer to an integer that receives information about the results of the test. See the explanation of the **flags** member of the LVHITTESTINFO structure in the *Platform SDK*.

Return Value

The index of the item at the position specified by *pHitTestInfo*, if any, or -1 otherwise.

Remarks

You can use the **LVHT_ABOVE**, **LVHT_BELOW**, **LVHT_TOLEFT**, and **LVHT_TORIGHT** values of the structure's **flag** member to determine whether to scroll the contents of a list view control. Two of these flags can be combined, for example, if the position is above and to the left of the client area.

You can test for the **LVHT_ONITEM** value of the structure's **flag** member to determine whether a given position is over a list view item. This value is a bitwise-OR operation on the **LVHT_ONITEMICON**, **LVHT_ONITEMLABEL**, and **LVHT_ONITEMSTATEICON** values of the structure's **flag** member.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;
// The pointer where the mouse was clicked.
extern CPoint myPoint;

// Select the item the user clicked on.
UINT uFlags;
int nItem = pmyListCtrl->HitTest(myPoint, &uFlags);

if (uFlags & LVHT_ONITEMLABEL)
{
   pmyListCtrl->SetItem(nItem, 0, LVIF_STATE, NULL, 0, LVIS_SELECTED,
      LVIS_SELECTED, 0);
}
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::GetItemPosition

Other Resources

CListCtrl Members

CListCtrl::InsertColumn

Inserts a new column in a list view control.

```
int InsertColumn(
   int nCol,
   const LVCOLUMN* pColumn
);
int InsertColumn(
   int nCol,
   LPCTSTR lpszColumnHeading,
   int nFormat = LVCFMT_LEFT,
   int nWidth = -1,
   int nSubItem = -1
);
```

Parameters

nCol

The index of the new column.

pColumn

Address of an **LVCOLUMN** structure that contains the attributes of the new column.

lpszColumnHeading

Address of a string containing the column's heading.

nFormat

Integer specifying the alignment of the column. It can be one of these values: **LVCFMT_LEFT**, **LVCFMT_RIGHT**, or **LVCFMT_CENTER**.

nWidth

Width of the column, in pixels. If this parameter is -1, the column width is not set.

nSubItem

Index of the subitem associated with the column. If this parameter is -1, no subitem is associated with the column.

Return Value

The index of the new column if successful or -1 otherwise.

Remarks

The leftmost column in a list view control must be left-aligned.

The LVCOLUMN structure contains the attributes of a column in report view. It is also used to receive information about a column. This structure is described in the *Platform SDK*.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::DeleteColumn

Other Resources

CListCtrl Members

CListCtrl::InsertGroup

Inserts a group into the list view control.

```
LRESULT InsertGroup(
   int index,
   PLVGROUP pgrp
);
```

Parameters

index

The index of the item where the group is to be inserted.

pgrp

A pointer to an LVGROUP structure containing the group to be added.

Return Value

Returns the index of the item that the group was added to, or -1 if the operation failed.

Remarks

This member function emulates the functionality of the LVM_INSERTGROUP message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::InsertGroupSorted

Inserts the specified group into an ordered list of groups.

```
LRESULT InsertGroupSorted(
   PLVINSERTGROUPSORTED pStructInsert
);
```

Parameters

pStructInsert

A pointer to an LVINSERTGROUPSORTED structure that contains the group to insert.

Return Value

The return value is not used.

Remarks

This member function emulates the functionality of the LVM_INSERTGROUPSORTED message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::InsertItem

Inserts an item into the list view control.

```
int InsertItem(
   const LVITEM* pItem
);
int InsertItem(
   int nItem,
   LPCTSTR lpszItem
);
int InsertItem(
   int nItem,
   LPCTSTR lpszItem,
   int nImage
);
int InsertItem(
   UINT nMask,
   int nItem,
   LPCTSTR lpszItem,
   UINT nState,
   UINT nStateMask,
   int nImage,
   LPARAM lParam
);
```

Parameters

pItem

Pointer to an LVITEM structure that specifies the item's attributes, as described in the *Platform SDK*.

nItem

Index of the item to be inserted.

lpszItem

Address of a string containing the item's label, or **LPSTR_TEXTCALLBACK** if the item is a callback item. For information on callback items, see CListCtrl::GetCallbackMask.

nImage

Index of the item's image, or **I_IMAGECALLBACK** if the item is a callback item. For information on callback items, see CListCtrl::GetCallbackMask.

nMask

The *nMask* parameter specifies which item attributes passed as parameters are valid. It can be one or more of the mask values described in LVITEM Structure in the *Platform SDK*. The valid values can be combined with the bitwise OR operator.

nState

Indicates the item's state, state image, and overlay image. See the *Platform SDK* topics LVITEM Structure for more information and List-View Item States for a list of valid flags.

nStateMask

Indicates which bits of the state member will be retrieved or modified. See LVITEM Structure in the *Platform SDK* for more information.

lParam

A 32-bit application-specific value associated with the item. If this parameter is specified, you must set the *nMask* attribute **LVIF_PARAM**.

Return Value

The index of the new item if successful or -1 otherwise.

Remarks

Calling this method may cause the **LVM_INSERTITEM** message to be sent to your control window. The associated message handler for the control may fail to set the item text under certain conditions (such as using window styles such as **LVS_OWNERDRAW**). For more information on these conditions, refer to LVM_INSERTITEM in the *Platform SDK*.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

CString strText;
int nColumnCount = pmyListCtrl->GetHeaderCtrl()->GetItemCount();

// Insert 10 items in the list view control.
for (int i=0;i < 10;i++)
{
   strText.Format(TEXT("item %d"), i);

   // Insert the item, select every other item.
   pmyListCtrl->InsertItem(
      LVIF_TEXT|LVIF_STATE, i, strText,
      (i%2)==0 ? LVIS_SELECTED : 0, LVIS_SELECTED,
      0, 0);

   // Initialize the text of the subitems.
   for (int j=1;j < nColumnCount;j++)
   {
      strText.Format(TEXT("sub-item %d %d"), i, j);
      pmyListCtrl->SetItemText(i, j, strText);
   }
}
```

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::DeleteItem
CListCtrl::DeleteAllItems
LVM_INSERTITEM

Other Resources

CListCtrl Members

CListCtrl::InsertMarkHitTest

Retrieves the insertion point closest to a specified point.

```
int InsertMarkHitTest(
   LPPOINT pPoint,
   LPLVINSERTMARK lvim
) const;
```

Parameters

pPoint

A pointer to a POINT structure that contains the hit test coordinates, relative to the client area of the list control.

lvim

A pointer to an LVINSERTMARK structure that specifies the insertion point closest to the coordinates defined by the point parameter.

Return Value

The insertion point closest to the specified point.

Remarks

This member function emulates the functionality of the LVM_INSERTMARKHITTEST message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetInsertMark
CListCtrl::SetInsertMark

Other Resources

CListCtrl Members

CListCtrl::IsGroupViewEnabled

Determines whether group view is enabled for a list view control.

```
BOOL IsGroupViewEnabled( ) const;
```

Return Value

Returns **TRUE** if group view is enabled, or **FALSE** otherwise.

Remarks

This member function emulates the functionality of the LVM_ISGROUPVIEWENABLED message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::MoveGroup

Moves the specified group to the specified zero based index of the list view control.

```
LRESULT MoveGroup(
   int iGroupId,
   int toIndex
);
```

Parameters

iGroupId

The identifier of the group to be moved.

toIndex

The zero-based index where the group is to be moved.

Return Value

The return value is not used.

Remarks

This member function emulates the functionality of the LVM_MOVEGROUP message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::MoveItemToGroup

Moves the specified item into the specified group.

```
LRESULT MoveItemToGroup(
   int idItemFrom,
   int idGroupTo
);
```

Parameters

idItemFrom

The index of the item to be moved.

idGroupTo

The identifier of the group the item will be moved to.

Return Value

The return value is not used.

Remarks

This member function emulates the functionality of the LVM_MOVEITEMTOGROUP message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::RedrawItems

Forces a list view control to repaint a range of items.

```
BOOL RedrawItems(
   int nFirst,
   int nLast
);
```

Parameters

nFirst

Index of the first item to be repainted.

nLast

Index of the last item to be repainted.

Return Value

Nonzero if successful; otherwise zero.

Remarks

The specified items are not actually repainted until the list view window receives a **WM_PAINT** message. To repaint immediately, call the Windows UpdateWindow function after using this function.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::DrawItem

Other Resources

CListCtrl Members

CListCtrl::RemoveAllGroups

Removes all groups from a list view control.

```
void RemoveAllGroups( );
```

Remarks

This member function emulates the functionality of the LVM_REMOVEALLGROUPS message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::RemoveGroup

Removes the specified group from the list view control.

```
LRESULT RemoveGroup(
   int iGroupId
);
```

Parameters

iGroupId

The identifier of the group to be removed.

Return Value

Returns the index of the group if successful, or -1 otherwise.

Remarks

This member function emulates the functionality of the LVM_REMOVEGROUP message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::Scroll

Scrolls the content of a list view control.

```
BOOL Scroll(
   CSize size
);
```

Parameters

size

A **CSize** object specifying the amount of horizontal and vertical scrolling, in pixels. The **y** member of *size* is divided by the height, in pixels, of the list view control's line, and the control is scrolled by the resulting number of lines.

Return Value

Nonzero if successful; otherwise zero.

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::EnsureVisible

Other Resources

CListCtrl Members

CListCtrl::SetBkColor

Sets the background color of the list view control.

```
BOOL SetBkColor(
   COLORREF cr
);
```

Parameters

cr

Background color to set, or the **CLR_NONE** value for no background color. List view controls with background colors redraw themselves significantly faster than those without background colors. For information, see COLORREF in the *Platform SDK*.

Return Value

Nonzero if successful; otherwise zero.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// Use the 3D button face color for the background.
COLORREF crBkColor = ::GetSysColor(COLOR_3DFACE);
pmyListCtrl->SetBkColor(crBkColor);
ASSERT(pmyListCtrl->GetBkColor() == crBkColor);
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::GetBkColor

Other Resources

CListCtrl Members

CListCtrl::SetBkImage

Sets the background image of a list view control.

```
BOOL SetBkImage(
   LVBKIMAGE* plvbkImage
);
BOOL SetBkImage(
   HBITMAP hbm,
   BOOL fTile = TRUE,
   int xOffsetPercent = 0,
   int yOffsetPercent = 0
);
BOOL SetBkImage(
   LPTSTR pszUrl,
   BOOL fTile = TRUE,
   int xOffsetPercent = 0,
   int yOffsetPercent = 0
);
```

Parameters

plvbkImage

Address of an **LVBKIMAGE** structure, containing the new background image information.

hbm

Handle to a bitmap.

pszUrl

A **NULL**-terminated string that contains the URL of the background image.

fTile

Nonzero if the image is to be tiled in the background of the list view control; otherwise 0.

xOffsetPercent

The offset, in pixels, of the image's left edge, from origin of the list view control.

yOffsetPercent

The offset, in pixels, of the image's top edge, from origin of the list view control.

Return Value

Returns nonzero if successful, or zero otherwise.

Remarks

Note

Because **CListCtrl::SetBkImage** makes use of OLE COM functionality, the OLE libraries must be initialized before using **SetBkImage**. It is best to initialize the COM libraries when the application is initialized and uninitialize the libraries when the application terminates. This is automatically done in MFC applications that make use of ActiveX technology, OLE Automation, OLE Linking/Embedding, or ODBC/DAO operations.

Example

See the example for CListCtrl::GetBkImage.

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::GetBkImage

Other Resources

CListCtrl Members

CListCtrl::SetCallbackMask

Sets the callback mask for a list view control.

```
BOOL SetCallbackMask(
   UINT nMask
);
```

Parameters

nMask

New value of the callback mask.

Return Value

Nonzero if successful; otherwise zero.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// Set the callback mask so that only the selected and focused states
// are stored for each item.
pmyListCtrl->SetCallbackMask(LVIS_SELECTED|LVIS_FOCUSED);
ASSERT(pmyListCtrl->GetCallbackMask() ==
   (LVIS_SELECTED|LVIS_FOCUSED));
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::GetCallbackMask

Other Resources

CListCtrl Members

CListCtrl::SetCheck

Determines if the state image of a list control item is visible.

```
BOOL SetCheck(
   int nItem,
   BOOL fCheck = TRUE
);
```

Parameters

nItem

The zero-based index of a list control item.

fCheck

Specifies whether the state image of the item should be visible or not. By default, *fCheck* is **TRUE** and the state image is visible. If *fCheck* is **FALSE**, it is not visible.

Return Value

Nonzero if the item is checked, otherwise 0.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

int nCount = pmyListCtrl->GetItemCount();
BOOL fCheck = FALSE;

// Set the check state of every other item to TRUE and
// all others to FALSE.
for (int i=0;i < nCount;i++)
{
   pmyListCtrl->SetCheck(i, fCheck);
   ASSERT((pmyListCtrl->GetCheck(i) && fCheck) ||
      (!pmyListCtrl->GetCheck(i) && !fCheck));
   fCheck = !fCheck;
}
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::GetCheck

Other Resources

CListCtrl Members

CListCtrl::SetColumn

Sets the attributes of a list view column.

```
BOOL SetColumn(
   int nCol,
   const LVCOLUMN* pColumn
);
```

Parameters

nCol

Index of the column whose attributes are to be set.

pColumn

Address of an LVCOLUMN structure that contains the new column attributes, as described in the *Platform SDK*. The structure's **mask** member specifies which column attributes to set. If the **mask** member specifies the **LVCF_TEXT** value, the structure's **pszText** member is the address of a null-terminated string and the structure's **cchTextMax** member is ignored.

Return Value

Nonzero if successful; otherwise zero.

Example

See the example for CListCtrl::GetColumn.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetColumn

Other Resources

CListCtrl Members

CListCtrl::SetColumnOrderArray

Sets the column order (left to right) of a list view control.

```
BOOL SetColumnOrderArray(
   int iCount,
   LPINT piArray
);
```

Parameters

piArray

A pointer to a buffer containing the index values of the columns in the list view control (from left to right). The buffer must be large enough to contain the total number of columns in the list view control.

iCount

Number of columns in the list view control.

Return Value

Nonzero if successful; otherwise zero.

Remarks

This member function implements the behavior of the Win32 macro, ListView_SetColumnOrderArray, as described in the *Platform SDK*.

Example

See the example for CListCtrl::GetColumnOrderArray.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetColumnOrderArray

Other Resources

CListCtrl Members

CListCtrl::SetColumnWidth

Changes the width of a column in report view or list view.

```
BOOL SetColumnWidth(
   int nCol,
   int cx
);
```

Parameters

nCol

Index of the column for which the width is to be set. In list view, this parameter must be 0.

cx

The new width of the column. Can be either **LVSCW_AUTOSIZE** or **LVSCW_AUTOSIZE_USEHEADER**, as described in LVM_SETCOLUMNWIDTH in the *Platform SDK*.

Return Value

Nonzero if successful; otherwise zero.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetColumnWidth
CListCtrl::GetStringWidth

Other Resources

CListCtrl Members

CListCtrl::SetExtendedStyle

Sets the current extended styles of a list view control.

```
DWORD SetExtendedStyle(
   DWORD dwNewStyle
);
```

Parameters

dwNewStyle

A combination of extended styles to be used by the list view control. For a descriptive list of these styles, see the Extended List View Styles topic in the *Platform SDK*.

Return Value

A combination of the previous extended styles used by the list view control.

Remarks

This member function implements the behavior of the Win32 macro, ListView_SetExtendedListViewStyle, as described in the *Platform SDK*.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// Allow the header controls item to be movable by the user.
pmyListCtrl->SetExtendedStyle
   (pmyListCtrl->GetExtendedStyle()|LVS_EX_HEADERDRAGDROP);
```

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::CreateEx
CListCtrl::GetExtendedStyle

Other Resources

CListCtrl Members

CListCtrl::SetGroupInfo

Sets the information for the specified group of a list view control.

```
int SetGroupInfo(
   int iGroupId,
   PLVGROUP pgrp
);
```

Parameters

iGroupId

The identifier of the group whose information is to be set.

pgrp

A pointer to the LVGROUP containing information on the group specified.

Return Value

Returns the ID of the group if successful, or -1 otherwise.

Remarks

This member function emulates the functionality of the LVM_SETGROUPINFO message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetGroupInfo

Other Resources

CListCtrl Members

CListCtrl::SetGroupMetrics

Sets the group metrics of a list view control.

```
void SetGroupMetrics(
   PLVGROUPMETRICS pGroupMetrics
);
```

Parameters

pGroupMetrics

A pointer to an LVGROUPMETRICS structure containing the group metrics information to be set.

Remarks

This member function emulates the functionality of the LVM_SETGROUPMETRICS message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::GetGroupMetrics

Other Resources

CListCtrl Members

CListCtrl::SetHotCursor

Sets the cursor used when hot tracking is enabled for a list view control.

```
HCURSOR SetHotCursor(
   HCURSOR hc
);
```

Parameters

hc

A handle to a cursor resource, used to represent the hot cursor.

Return Value

The handle to the previous hot cursor resource being used by the list view control.

Remarks

This member function implements the behavior of the Win32 macro, ListView_SetHotCursor, as described in the *Platform SDK*.

The hot cursor, only visible when hover selection is enabled, appears as the cursor passes over any list view item. Hover selection is enabled by setting the **LVS_EX_TRACKSELECT** extended style.

Example

See the example for CListCtrl::GetHotCursor.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetHotCursor
CListCtrl::GetHotItem

Other Resources

CListCtrl Members

CListCtrl::SetHotItem

Sets the current hot item of a list view control.

```
int SetHotItem(
   int iIndex
);
```

Parameters

iIndex

Zero-based index of the item to be set as the hot item.

Return Value

The zero-based index of the previously hot item.

Remarks

This member function implements the behavior of the Win32 macro, ListView_SetHotItem, as described in the *Platform SDK*.

Example

See the example for CListCtrl::GetHotItem.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetHotCursor
CListCtrl::GetHotItem

Other Resources

CListCtrl Members

CListCtrl::SetHoverTime

Sets the current hover time of a list view control.

```
DWORD SetHoverTime(
   DWORD dwHoverTime = (DWORD
)-1
);
```

Parameters

dwHoverTime

The new delay, in milliseconds, which the mouse cursor must hover over an item before it is selected. If the default value is passed, the time is set to the default hover time.

Return Value

The previous hover time, in milliseconds.

Remarks

This member function implements the behavior of the Win32 macro, ListView_SetHoverTime, as described in the *Platform SDK*.

Example

See the example for CListCtrl::GetHoverTime.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetHoverTime

Other Resources

CListCtrl Members

CListCtrl::SetIconSpacing

Sets the spacing between icons in a list view control.

```
CSize SetIconSpacing(
   int cx,
   int cy
);
CSize SetIconSpacing(
   CSize size
);
```

Parameters

cx

The distance (in pixels) between icons on the x-axis.

cy

The distance (in pixels) between icons on the y-axis.

size

A **CSize** object specifying the distance (in pixels) between icons on the x- and y-axes.

Return Value

A CSize object containing the previous values for icon spacing.

Remarks

This member function implements the behavior of the Win32 macro, ListView_SetIconSpacing, as described in the *Platform SDK*.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// Leave lots of space between icons.
pmyListCtrl->SetIconSpacing(CSize(100, 100));
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::SetImageList

Assigns an image list to a list view control.

```
CImageList* SetImageList(
   CImageList* pImageList,
   int nImageListType
);
```

Parameters

pImageList

Pointer to the image list to assign.

nImageListType

Type of image list. It can be one of these values:

- **LVSIL_NORMAL** Image list with large icons.
- **LVSIL_SMALL** Image list with small icons.
- **LVSIL_STATE** Image list with state images.

Return Value

A pointer to the previous image list.

Example

See the example for CListCtrl::GetImageList.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CImageList Class
CListCtrl::GetImageList

Other Resources

CListCtrl Members

CListCtrl::SetInfoTip

Sets the tooltip text.

```
BOOL SetInfoTip(
   PLVSETINFOTIP plvInfoTip
);
```

Parameters

plvInfoTip

A pointer to an LVFSETINFOTIP structure containing the information to be set.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This member function emulates the functionality of the LVM_SETINFOTIP message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::SetInsertMark

Sets the insertion point to the defined position.

```
BOOL SetInsertMark(
   LPLVINSERTMARK lvim
);
```

Parameters

lvim

A pointer to an LVINSERTMARK structure specifying where to set the insertion point.

Return Value

Returns **TRUE** if successful, or **FALSE** otherwise. **FALSE** is returned if the size in the **cbSize** member of the **LVINSERTMARK** structure does not equal the actual size of the structure, or when an insertion point does not apply in the current view.

Remarks

This member function emulates the functionality of the LVM_SETINSERTMARK message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Other Resources

CListCtrl Members

CListCtrl::SetInsertMarkColor

Sets the color of the insertion point.

```
COLORREF SetInsertMarkColor(
   COLORREF color
);
```

Parameters

color

A COLORREF structure specifying the color to set the insertion point.

Return Value

Returns a **COLORREF** structure containing the previous color.

Remarks

This member function emulates the functionality of the LVM_SETINSERTMARKCOLOR message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::SetItem

Sets some or all of a list view item's attributes.

```
BOOL SetItem(
   const LVITEM* pItem
);
BOOL SetItem(
   int nItem,
   int nSubItem,
   UINT nMask,
   LPCTSTR lpszItem,
   int nImage,
   UINT nState,
   UINT nStateMask,
   LPARAM lParam
);
BOOL SetItem(
   int nItem,
   int nSubItem,
   UINT nMask,
   LPCTSTR lpszItem,
   int nImage,
   UINT nState,
   UINT nStateMask,
   LPARAM lParam,
   int nIndent
);
```

Parameters

pItem

Address of an LVITEM structure that contains the new item attributes, as described in the *Platform SDK*. The structure's **iItem** and **iSubItem** members identify the item or subitem, and the structure's **mask** member specifies which attributes to set. For more information on the **mask** member, see the **Remarks**.

nItem

Index of the item whose attributes are to be set.

nSubItem

Index of the subitem whose attributes are to be set.

nMask

Specifies which attributes are to be set (see the Remarks).

lpszItem

Address of a null-terminated string specifying the item's label.

nImage

Index of the item's image within the image list.

nState

Specifies values for states to be changed (see the Remarks).

nStateMask

Specifies which states are to be changed (see the Remarks).

lParam

A 32-bit application-specific value to be associated with the item.

nIndent

Width, in pixels, of the indentation. If *nIndent* is less than the system-defined minimum width, the new width is set to the system-defined minimum

Return Value

Nonzero if successful; otherwise zero.

Remarks

The **iItem** and **iSubItem** members of the **LVITEM** structure and the *nItem* and *nSubItem* parameters identify the item and subitem whose attributes are to be set.

The **mask** member of the **LVITEM** structure and the *nMask* parameter specify which item attributes are to be set:

- **LVIF_TEXT** The **pszText** member or the *lpszItem* parameter is the address of a null-terminated string; the **cchTextMax** member is ignored.
- **LVIF_STATE** The **stateMask** member or *nStateMask* parameter specifies which item states to change and the **state** member or *nState* parameter contains the values for those states.

Example

See the example for CListCtrl::HitTest.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetItem

Other Resources

CListCtrl Members

CListCtrl::SetItemCount

Prepares a list view control for adding a large number of items.

```
void SetItemCount(
   int nItems
);
```

Parameters

nItems

Number of items that the control will ultimately contain.

Remarks

To set the item count for a virtual list view control, see CListCtrl::SetItemCountEx.

Remarks

This member function implements the behavior of the Win32 macro, ListView_SetItemCount, as described in the *Platform SDK*.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;
CString str;

// Add 1024 items to the list view control.
pmyListCtrl->SetItemCount(1024);

for (int i=0;i < 1024;i++)
{
   str.Format(TEXT("item %d"), i);
   pmyListCtrl->InsertItem(i, str);
}
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::GetItemCount

CListCtrl::GetSelectedCount

Other Resources

CListCtrl Members

CListCtrl::SetItemCountEx

Sets the item count for a virtual list view control.

```
BOOL SetItemCountEx(
   int iCount,
   DWORD dwFlags = LVSICF_NOINVALIDATEALL
);
```

Parameters

iCount

Number of items that the control will ultimately contain.

dwFlags

Specifies the behavior of the list view control after resetting the item count. This value can be a combination of the following:

- **LVSICF_NOINVALIDATEALL** The list view control will not repaint unless affected items are currently in view. This is the default value.
- **LVSICF_NOSCROLL** The list view control will not change the scroll position when the item count changes.

Return Value

Nonzero if successful; otherwise zero.

Remarks

This member function implements the behavior of the Win32 macro, ListView_SetItemCountEx, as described in the *Platform SDK* and should only be called for virtual list views.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;
CString str;

// Add 1024 items to the list view control.

// Force my virtual list view control to allocate
// enough memory for my 1024 items.
pmyListCtrl->SetItemCountEx(1024, LVSICF_NOSCROLL|LVSICF_NOINVALIDATEALL);

for (int i=0;i < 1024;i++)
{
   str.Format(TEXT("item %d"), i);
   pmyListCtrl->InsertItem(i, str);
}
```

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::SetItemCount

Other Resources

CListCtrl Members

CListCtrl::SetItemData

Sets the 32-bit application-specific value associated with the item specified by *nItem*.

```
BOOL SetItemData(
   int nItem,
      DWORD_PTR dwData
);
```

Parameters

nItem

Index of the list item whose data is to be set.

dwData

A 32-bit value to be associated with the item.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This value is the **lParam** member of the LVITEM structure, as described in the *Platform SDK*.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// Set the data of each item to be equal to its index.
for (int i=0;i < pmyListCtrl->GetItemCount();i++)
{
   pmyListCtrl->SetItemData(i, i);
}
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::GetItemData

Other Resources

CListCtrl Members

CListCtrl::SetItemPosition

Moves an item to a specified position in a list view control.

```
BOOL SetItemPosition(
   int nItem,
   POINT pt
);
```

Parameters

nItem

Index of the item whose position is to be set.

pt

A POINT structure specifying the new position, in view coordinates, of the item's upper-left corner.

Return Value

Nonzero if successful; otherwise zero.

Remarks

The control must be in icon or small icon view.

If the list view control has the **LVS_AUTOARRANGE** style, the list view is arranged after the position of the item is set.

Example

See the example for CListCtrl::GetItemPosition.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetItemPosition
CListCtrl::GetOrigin

Other Resources

CListCtrl Members

CListCtrl::SetItemState

Changes the state of an item in a list view control.

```
BOOL SetItemState(
   int nItem,
   LVITEM* pItem
);
BOOL SetItemState(
   int nItem,
   UINT nState,
   UINT nMask
);
```

Parameters

nItem

Index of the item whose state is to be set.

pItem

Address of an LVITEM structure, as described in the *Platform SDK*. The structure's **stateMask** member specifies which state bits to change, and the structure's **state** member contains the new values for those bits. The other members are ignored.

nState

New values for the state bits.

nMask

Mask specifying which state bits to change.

Return Value

Nonzero if successful; otherwise zero.

Remarks

An item's "state" is a value that specifies the item's availability, indicates user actions, or otherwise reflects the item's status. A list view control changes some state bits, such as when the user selects an item. An application might change other state bits to disable or hide the item, or to specify an overlay image or state image.

Example

See the example for CListCtrl::GetTopIndex.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetItemState

Other Resources

CListCtrl Members

CListCtrl::SetItemText

Changes the text of a list view item or subitem.

```
BOOL SetItemText(
   int nItem,
   int nSubItem,
   LPCTSTR lpszText
);
```

Parameters

nItem

Index of the item whose text is to be set.

nSubItem

Index of the subitem, or zero to set the item label.

lpszText

Pointer to a string that contains the new item text.

Return Value

Nonzero if successful; otherwise zero.

Remarks

This method is not intended for use with controls containing the LVS_OWNERDATA window style (in fact, this will cause an assertion in Debug builds). For more information about this list control style, see List-View Controls Overview.

Example

See the example for CListCtrl::InsertItem.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetItemText

Other Resources

CListCtrl Members

CListCtrl::SetOutlineColor

Sets the color of the border of a list-view control if the LVS_EX_BORDERSELECT extended window style is set.

```
COLORREF SetOutlineColor(
   COLORREF color
);
```

Parameters

color

The new COLORREF structure containing the outline color.

Return Value

The previous **COLORREF** structure containing the outline color

Remarks

This member function emulates the functionality of the LVM_SETOUTLINECOLOR message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::SetSelectedColumn

Sets the selected column of the list view control.

```
LRESULT SetSelectedColumn(
   int iCol
);
```

Parameters

iCol

The index of the column to be selected.

Return Value

The return value is not used.

Remarks

This member function emulates the functionality of the LVM_SETSELECTEDCOLUMN message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl::GetSelectedColumn
CListCtrl Class
Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::SetSelectionMark

Sets the selection mark of a list view control.

```
int SetSelectionMark(
   int iIndex
);
```

Parameters

iIndex

The zero-based index of the first item in a multiple selection.

Return Value

The previous selection mark, or -1 if there was no selection mark.

Remarks

This member function implements the behavior of the Win32 macro, ListView_SetSelectionMark, as described in the *Platform SDK*.

Example

See the example for CListCtrl::GetSelectionMark.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetSelectionMark

Other Resources

CListCtrl Members

CListCtrl::SetTextBkColor

Sets the background color of text in a list view control.

```
BOOL SetTextBkColor(
   COLORREF cr
);
```

Parameters

cr

A **COLORREF** specifying the new text background color. For information, see COLORREF in the *Platform SDK*.

Return Value

Nonzero if successful; otherwise zero.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// Use the 3D button face color for the text background.
COLORREF crBkColor = ::GetSysColor(COLOR_3DFACE);
pmyListCtrl->SetTextBkColor(crBkColor);
ASSERT(pmyListCtrl->GetTextBkColor() == crBkColor);
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::GetTextBkColor

Other Resources

CListCtrl Members

CListCtrl::SetTextColor

Sets the text color of a list view control.

```
BOOL SetTextColor(
   COLORREF cr
);
```

Parameters

cr

A **COLORREF** specifying the new text color. For information, see COLORREF in the *Platform SDK*.

Return Value

Nonzero if successful; otherwise zero.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// Use the window text color for
// the item text of the list view control.
COLORREF crTextColor = ::GetSysColor(COLOR_WINDOWTEXT);
pmyListCtrl->SetTextColor(crTextColor);
ASSERT(pmyListCtrl->GetTextColor() == crTextColor);
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::SetTextBkColor

Other Resources

CListCtrl Members

CListCtrl::SetTileInfo

Sets the information for a tile of the list view control.

```
BOOL SetTileInfo(
   PLVTILEINFO pti
);
```

Parameters

pti

A pointer to an LVTILEINFO structure containing the information to be set.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This member function emulates the functionality of the LVM_SETTILEINFO message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetTileInfo

Other Resources

CListCtrl Members

CListCtrl::SetTileViewInfo

Sets information that a list view control uses in tile view.

```
BOOL SetTileViewInfo(
   PLVTILEVIEWINFO ptvi
);
```

Parameters

ptvi

A pointer to an LVTILEVIEWINFO structure containing the information to set.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This member function emulates the functionality of the LVM_SETTILEVIEWINFO message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::SetToolTips

Sets the tooltip control that the list view control will use to display tooltips.

```
CToolTipCtrl* SetToolTips(
   CToolTipCtrl* pWndTip
);
```

Parameters

pWndTip

A pointer to a **CToolTipCtrl** object that the list control will use.

Return Value

A pointer to a CToolTipCtrl object containing the tooltip previously used by the control, or **NULL** if no tooltips were used previously.

Remarks

This member function implements the behavior of the Win32 message LVM_SETTOOLTIPS, as described in the *Platform SDK*.

To use tooltips, indicate the **LVS_NOTOOLTIPS** style when you create the **CListCtrl** object.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::GetToolTips

Other Resources

CListCtrl Members

CListCtrl::SetView

Sets the view of the list view control.

```
DWORD SetView(
   int iView
);
```

Parameters

iView

The view to be selected.

Return Value

Returns 1 if successful, or -1 otherwise. For example, -1 is returned if the view is invalid.

Remarks

This member function emulates the functionality of the LVM_SETVIEW message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::SetWorkAreas

Sets the area where icons can be displayed in a list view control.

```
void SetWorkAreas(
   int nWorkAreas,
   LPRECT lpRect
);
```

Parameters

nWorkAreas

The number of **RECT** structures (or CRect objects) in the array pointed to by *lpRect*.

lpRect

The address of an array of **RECT** structures (or **CRect** objects) that specify the new work areas of the list view control. These areas must be specified in client coordinates. If this parameter is **NULL**, the working area will be set to the client area of the control.

Remarks

This member function implements the behavior of the Win32 macro, ListView_SetWorkAreas, as described in the *Platform SDK*.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;

// Remove all working areas.
pmyListCtrl->SetWorkAreas(0, NULL);
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::SortGroups

Uses an application-defined comparison function to sort groups by ID within a list view control.

```
BOOL SortGroups(
   PFNLVGROUPCOMPARE _pfnGroupCompare,
   LPVOID _plv
);
```

Parameters

_pfnGroupCompare

A pointer to the group comparison function.

_plv

A void pointer.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This member function emulates the functionality of the LVM_SORTGROUPS message, as described in the *Platform SDK*.

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::SortItems

Sorts list view items using an application-defined comparison function.

```
BOOL SortItems(
   PFNLVCOMPARE pfnCompare,
   DWORD_PTR dwData
);
```

Parameters

pfnCompare

Address of the application-defined comparison function. The comparison function is called during the sort operation each time the relative order of two list items needs to be compared. The comparison function must be either a static member of a class or a stand-alone function that is not a member of any class.

dwData

Application-defined value that is passed to the comparison function.

Return Value

Nonzero if successful; otherwise zero.

Remarks

The index of each item changes to reflect the new sequence.

The comparison function has the following form:

```
int CALLBACK CompareFunc(LPARAM lParam1, LPARAM lParam2,
   LPARAM lParamSort);
```

The comparison function must return a negative value if the first item should precede the second, a positive value if the first item should follow the second, or zero if the two items are equivalent.

The *lParam1* parameter is the 32-bit value associated with the first item being compared, and the *lParam2* parameter is the value associated with the second item. These are the values that were specified in the **lParam** member of the items' LVITEM structure when they were inserted into the list. The *lParamSort* parameter is the same as the *dwData* value.

Example

The following is a simple comparison function that results in items being sorted by their **lParam** values.

```
// Sort items by associated lParam
static int CALLBACK
MyCompareProc(LPARAM lParam1, LPARAM lParam2, LPARAM lParamSort)
{
   return (int)(lParam1 - lParam2);
}
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

CListCtrl::FindItem

Other Resources

CListCtrl Members

CListCtrl::SubItemHitTest

Determines which list view item, if any, is at a given position.

```
int SubItemHitTest(
   LPLVHITTESTINFO pInfo
);
```

Parameters

pInfo

A pointer to the LVHITTESTINFO structure.

Return Value

The one-based index of the item, or subitem, being tested (if any), or -1 otherwise.

Remarks

This member function implements the behavior of the Win32 macro, ListView_SubItemHitTest, as described in the *Platform SDK*.

Example

```
// The pointer to my list view control.
extern CListCtrl* pmyListCtrl;
// The pointer where the mouse was clicked.
extern CPoint myPoint;

LVHITTESTINFO lvhti;

// Clear the subitem text the user clicked on.
lvhti.pt = myPoint;
pmyListCtrl->SubItemHitTest(&lvhti);

if (lvhti.flags & LVHT_ONITEMLABEL)
{
   pmyListCtrl->SetItemText(lvhti.iItem, lvhti.iSubItem, NULL);
}
```

See Also

Reference

CListCtrl Class

Hierarchy Chart

Other Resources

CListCtrl Members

CListCtrl::Update

Forces the list view control to repaint the item specified by *nItem*.

```
BOOL Update(
   int nItem
);
```

Parameters

nItem

Index of the item to be updated.

Return Value

Nonzero if successful; otherwise zero.

Remarks

This function also arranges the list view control if it has the **LVS_AUTOARRANGE** style.

Example

See the example for CListCtrl::GetSelectedCount.

See Also

Reference

CListCtrl Class
Hierarchy Chart
CListCtrl::DrawItem

Other Resources

CListCtrl Members

CListView Class

Simplifies use of the list control and of CListCtrl, the class that encapsulates list-control functionality, with MFC's document-view architecture.

```
class CListView : public CCtrlView
```

Remarks

For more information on this architecture, see the overview for the CView class and the cross-references cited there.

Requirements

Header: afxcview.h

See Also

Tasks

ROWLIST Sample: Demonstrates Selecting Full Rows in List Views

Reference

CCtrlView Class

Hierarchy Chart

CCtrlView Class

Other Resources

CListView Members

CListView Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CView Members

CCtrlView Members

Construction

CListView	Constructs a **CListView** object.

Attributes

GetListCtrl	Returns the list control associated with the view.
RemoveImageList	Removes the specified image list from the list view.

See Also

Reference

CListView Class

Hierarchy Chart

CListView Member Functions

For information about the member functions in **CListView**, see CListView Members.

CListView::CListView

Constructs a **CListView** object.

```
CListView( );
```

See Also

Reference

CListView Class

Hierarchy Chart

Other Resources

CListView Members

CListView::GetListCtrl

Call this member function to get a reference to the list control associated with the view.

```
CListCtrl& GetListCtrl( ) const;
```

Return Value

A reference to the list control associated with the view.

Example

```
void CMyListView::OnInitialUpdate()
{
   CListView::OnInitialUpdate();

   // this code only works for a report-mode list view
   ASSERT(GetStyle() & LVS_REPORT);

   // Gain a reference to the list control itself
   CListCtrl& theCtrl = GetListCtrl();

   // Insert a column. This override is the most convenient.
   theCtrl.InsertColumn(0, _T("Player Name"), LVCFMT_LEFT);

   // The other InsertColumn() override requires an initialized
   // LVCOLUMN structure.
   LVCOLUMN col;
   col.mask = LVCF_FMT | LVCF_TEXT;
   col.pszText = _T("Jersey Number");
   col.fmt = LVCFMT_LEFT;
   theCtrl.InsertColumn(1, &col);

   // Set reasonable widths for our columns
   theCtrl.SetColumnWidth(0, LVSCW_AUTOSIZE_USEHEADER);
   theCtrl.SetColumnWidth(1, LVSCW_AUTOSIZE_USEHEADER);
}
```

See Also

Reference

CListView Class

Hierarchy Chart

CListCtrl Class

Other Resources

CListView Members

CListView::RemoveImageList

Removes the specified image list from the list view.

```
void RemoveImageList(
   int nImageList
);
```

Parameters

nImageList

The zero-based index of the image to remove.

See Also

Reference

CListView Class
Hierarchy Chart
CListCtrl Class
LVM_GETIMAGELIST
LVM_SETIMAGELIST

Other Resources

CListView Members

CLongBinary Class

Simplifies working with very large binary data objects (often called BLOBs, or "binary large objects") in a database.

```
class CLongBinary : public CObject
```

Remarks

For example, a record field in a SQL table might contain a bitmap representing a picture. A **CLongBinary** object stores such an object and keeps track of its size.

Note

In general, it is better practice now to use CByteArray in conjunction with the DFX_Binary function. You can still use **CLongBinary**, but in general **CByteArray** provides more functionality under Win32, since there is no longer the size limitation encountered with 16-bit **CByteArray**. This advice applies to programming with Data Access Objects (DAO) as well as Open Database Connectivity (ODBC).

To use a **CLongBinary** object, declare a field data member of type **CLongBinary** in your recordset class. This member will be an embedded member of the recordset class and will be constructed when the recordset is constructed. After the **CLongBinary** object is constructed, the record field exchange (RFX) mechanism loads the data object from a field in the current record on the data source and stores it back to the record when the record is updated. RFX queries the data source for the size of the binary large object, allocates storage for it (via the **CLongBinary** object's **m_hData** data member), and stores an **HGLOBAL** handle to the data in **m_hData**. RFX also stores the actual size of the data object in the **m_dwDataLength** data member. Work with the data in the object through **m_hData**, using the same techniques you would normally use to manipulate the data stored in a Windows **HGLOBAL** handle.

When you destroy your recordset, the embedded **CLongBinary** object is also destroyed, and its destructor deallocates the **HGLOBAL** data handle.

For more information about large objects and the use of **CLongBinary**, see the articles Recordset (ODBC) and Recordset: Working with Large Data Items (ODBC).

Requirements

Header: afxdb_.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class
Hierarchy Chart
CRecordset Class

Other Resources

CLongBinary Members

CLongBinary Members

Base Class Members

CObject Members

Data Members

m_dwDataLength	Contains the actual size in bytes of the data object whose handle is stored in **m_hData**.
m_hData	Contains a Windows **HGLOBAL** handle to the actual image object.

Construction

CLongBinary	Constructs a **CLongBinary** object.

See Also

Reference

CLongBinary Class
Hierarchy Chart

CLongBinary Member Functions

For information about the member functions in **CLongBinary**, see CLongBinary Members.

CLongBinary::CLongBinary

Constructs a **CLongBinary** object.

```
CLongBinary( );
```

See Also

Reference

CLongBinary Class

Hierarchy Chart

Other Resources

CLongBinary Members

CLongBinary Data Members

For information about the data members in **CLongBinary**, see CLongBinary Members.

CLongBinary::m_dwDataLength

Stores the actual size in bytes of the data stored in the **HGLOBAL** handle in **m_hData**.

```
SQLULEN m_dwDataLength;
```

Remarks

This size may be smaller than the size of the memory block allocated for the data. Call the Win32 GLobalSize function to get the allocated size.

See Also

Reference

CLongBinary Class

Hierarchy Chart

Other Resources

CLongBinary Members

CLongBinary::m_hData

Stores a Windows **HGLOBAL** handle to the actual binary large object data.

```
HGLOBAL m_hData;
```

See Also

Reference

CLongBinary Class

Hierarchy Chart

Other Resources

CLongBinary Members

CMap Class

A dictionary collection class that maps unique keys to values.

```
template< class KEY, class ARG_KEY, class VALUE, class ARG_VALUE >class CMap : public CObje
ct
```

Parameters

KEY

Class of the object used as the key to the map.

ARG _ KEY

Data type used for *KEY* arguments; usually a reference to *KEY*.

VALUE

Class of the object stored in the map.

ARG _ VALUE

Data type used for *VALUE* arguments; usually a reference to *VALUE*.

Remarks

Once you have inserted a key-value pair (element) into the map, you can efficiently retrieve or delete the pair using the key to access it. You can also iterate over all the elements in the map.

A variable of type **POSITION** is used for alternate access to entries. You can use a **POSITION** to "remember" an entry and to iterate through the map. You might think that this iteration is sequential by key value; it is not. The sequence of retrieved elements is indeterminate.

Certain member functions of this class call global helper functions that must be customized for most uses of the **CMap** class. See Collection Class Helpers in the Macros and Globals section of the *MFC Reference*.

CMap overrides CObject::Serialize to support serialization and dumping of its elements. If a map is stored to an archive using **Serialize**, each map element is serialized in turn. The default implementation of the **SerializeElements** helper function does a bitwise write. For information about serialization of pointer collection items derived from **CObject** or other user defined types, see How to: Make a Type-Safe Collection.

If you need a diagnostic dump of the individual elements in the map (the keys and values), you must set the depth of the dump context to 1 or greater.

When a **CMap** object is deleted, or when its elements are removed, the keys and values both are removed.

Map class derivation is similar to list derivation. See the article Collections for an illustration of the derivation of a special-purpose list class.

Requirements

Header: afxtempl.h

See Also

Tasks

COLLECT Sample: Illustrates MFC Collection Classes

Reference

CObject Class

Hierarchy Chart

Other Resources

CMap Members

CMap Members

Base Class Members

CObject Members

Construction

CMap	Constructs a collection that maps keys to values.

Operations

GetHashTableSize	Returns the size (number of elements) of the hash table.
GetNextAssoc	Gets the next element for iterating.
PGetNextAssoc	Gets a pointer to the next element for iterating.
GetStartPosition	Returns the position of the first element.
PGetFirstAssoc	Returns a pointer to the first element.
InitHashTable	Initializes the hash table and specifies its size.
Lookup	Looks up the value mapped to a given key.
PLookup	Returns a pointer to a key whose value matches the specified value.
operator []	Inserts an element into the map — operator substitution for **SetAt**.
RemoveAll	Removes all the elements from this map.
RemoveKey	Removes an element specified by a key.
SetAt	Inserts an element into the map; replaces an existing element if a matching key is found.

Status

GetCount	Returns the number of elements in this map.
GetSize	Returns the number of elements in this map.
IsEmpty	Tests for the empty-map condition (no elements).

Data Members

CMap::CPair	A nested structure containing a key value and the value of the associated object.

See Also

Reference

CMap Class

Hierarchy Chart

CMap Member Functions

For information about the member functions in **CMap**, see CMap Members.

CMap::CMap

Constructs an empty map.

```
CMap(
   INT_PTR nBlockSize = 10
);
```

Parameters

nBlockSize

Specifies the memory-allocation granularity for extending the map.

Remarks

As the map grows, memory is allocated in units of *nBlockSize* entries.

Example

```
// declares a map of ints to points
CMap<int,int,CPoint,CPoint> myMap(16);
```

See Also

Reference

CMap Class

Hierarchy Chart

Other Resources

CMap Members

CMap::GetCount

Retrieves the number of elements in the map.

```
INT_PTR GetCount( ) const;
```

Return Value

The number of elements.

Example

See the example for CMap::Lookup.

See Also

Reference

CMap Class

Hierarchy Chart

CMap::IsEmpty

Other Resources

CMap Members

CMap::GetHashTableSize

Determines the number of elements in the hash table for the map.

```
UINT GetHashTableSize( ) const;
```

Return Value

The number of elements in the hash table.

Example

```
CMap<int,int,CPoint,CPoint> myMap;

UINT uTableSize = myMap.GetHashTableSize();
```

See Also

Reference

CMap Class
Hierarchy Chart
CMap::InitHashTable

Other Resources

CMap Members

CMap::GetNextAssoc

Retrieves the map element at *rNextPosition*, then updates *rNextPosition* to refer to the next element in the map.

```
void GetNextAssoc(
   POSITION& rNextPosition,
      KEY& rKey,
      VALUE& rValue
) const;
```

Parameters

rNextPosition

Specifies a reference to a **POSITION** value returned by a previous **GetNextAssoc** or **GetStartPosition** call.

KEY

Template parameter specifying the type of the map's key.

rKey

Specifies the returned key of the retrieved element.

VALUE

Template parameter specifying the type of the map's value.

rValue

Specifies the returned value of the retrieved element.

Remarks

This function is most useful for iterating through all the elements in the map. Note that the position sequence is not necessarily the same as the key value sequence.

If the retrieved element is the last in the map, then the new value of *rNextPosition* is set to **NULL**.

Example

See the example for CMap::SetAt.

See Also

Reference

CMap Class
Hierarchy Chart
CMap::GetStartPosition

Other Resources

CMap Members

CMap::GetSize

Returns the number of map elements.

```
INT_PTR GetSize( ) const;
```

Return Value

The number of items in the map.

Remarks

Call this method to retrieve the number of elements in the map.

Example

```
CMap<int,int,CPoint,CPoint> myMap;
   int i;

   myMap.InitHashTable( 257 );

   // Add 10 elements to the map.
   for (i=0;i < 200;i++)
      myMap[i] = CPoint(i, i);

   // Remove the elements with even key values.
   CPoint pt;
   for (i=0; myMap.Lookup( i, pt ) ;i+=2)
   {
      myMap.RemoveKey( i );
   }
#ifdef _DEBUG
   ASSERT(myMap.GetSize() == 100);
    afxDump.SetDepth( 1 );
    afxDump << "myMap: " << &myMap << "\n";
#endif
```

See Also

Reference

CMap Class

Hierarchy Chart

CMap::GetCount

Other Resources

CMap Members

CMap::GetStartPosition

Starts a map iteration by returning a **POSITION** value that can be passed to a **GetNextAssoc** call.

```
POSITION GetStartPosition( ) const;
```

Return Value

A **POSITION** value that indicates a starting position for iterating the map; or **NULL** if the map is empty.

Remarks

The iteration sequence is not predictable; therefore, the "first element in the map" has no special significance.

Example

See the example for CMap::SetAt.

See Also

Reference

CMap Class
Hierarchy Chart
CMap::GetNextAssoc

Other Resources

CMap Members

CMap::InitHashTable

Initializes the hash table.

```
void InitHashTable(
   UINT hashSize,
      BOOL bAllocNow = TRUE
);
```

Parameters

hashSize

Number of entries in the hash table.

bAllocNow

If **TRUE**, allocates the hash table upon initialization; otherwise the table is allocated when needed.

Remarks

For best performance, the hash table size should be a prime number. To minimize collisions, the size should be roughly 20 percent larger than the largest anticipated data set.

Example

See the example for CMap::Lookup.

See Also

Reference

CMap Class
Hierarchy Chart
CMap::GetHashTableSize

Other Resources

CMap Members

CMap::IsEmpty

Determines whether the map is empty.

```
BOOL IsEmpty( ) const;
```

Return Value

Nonzero if this map contains no elements; otherwise 0.

Example

See the example for CMap::RemoveAll.

See Also

Reference

CMap Class
Hierarchy Chart
CMap::GetCount

Other Resources

CMap Members

CMap::Lookup

Looks up the value mapped to a given key.

```
BOOL Lookup(
      ARG_KEY key,
      VALUE& rValue
) const;
```

Parameters

ARG_KEY

Template parameter specifying the type of the *key* value.

key

Specifies the key that identifies the element to be looked up.

VALUE

Specifies the type of the value to be looked up.

rValue

Receives the looked-up value.

Return Value

Nonzero if the element was found; otherwise 0.

Remarks

Lookup uses a hashing algorithm to quickly find the map element with a key that exactly matches the given key.

Example

```
CMap<int,int,CPoint,CPoint> myMap;
   int i;

   myMap.InitHashTable( 257 );

   // Add 10 elements to the map.
   for (i=0;i < 200;i++)
      myMap[i] = CPoint(i, i);

   // Remove the elements with even key values.
   CPoint pt;
   for (i=0; myMap.Lookup( i, pt ) ;i+=2)
   {
      myMap.RemoveKey( i );
   }

#ifdef _DEBUG
   ASSERT(myMap.GetCount() == 100);
    afxDump.SetDepth( 1 );
    afxDump << "myMap: " << &myMap << "\n";
#endif
```

See Also

Reference

CMap Class

Hierarchy Chart

CMap::operator [ ]

Other Resources

CMap Members

CMap::PGetFirstAssoc

Returns the first entry of the map object.

```
const CPair* PGetFirstAssoc( ) const;
CPair* PGetFirstAssoc( );
```

Return Value

A pointer to the first entry in the map; see CMap::CPair. If the map contains no entries, the value is **NULL**.

Remarks

Call this function to return a pointer the first element in the map object.

Example

```
typedef CMap<int, int, CPoint, CPoint> CMyMap;
CMyMap myMap;
int i;

myMap.InitHashTable( 257 );

// Add 10 elements to the map.
for (i=0;i <= 10;i++)
    myMap.SetAt( i, CPoint(i, i) );

// Print the element value with even key values.
int nKey= 0;
CPoint pt;
CMyMap::CPair* pCurVal;

pCurVal= myMap.PGetFirstAssoc( );
while (pCurVal != NULL)
{
    if ((nKey%2) == 0)
        printf_s("Current key value at %d: %d,%d\n",
                 pCurVal->key, pCurVal->value.x, pCurVal->value.y);
    pCurVal= myMap.PGetNextAssoc(pCurVal);
    nKey++
}
```

See Also

Reference

CMap Class
Hierarchy Chart
CMap::PGetNextAssoc
CMap::PLookup

Other Resources

CMap Members

CMap::PGetNextAssoc

Retrieves the map element pointed to by *pAssocRec*.

```
const CPair *PGetNextAssoc(
   const CPair* pAssocRet
) const;
CPair *PGetNextAssoc(
   const CPair* pAssocRet
);
```

Parameters

pAssocRet

Points to a map entry returned by a previous PGetNextAssoc or PGetFirstAssoc call.

Return Value

A pointer to the next entry in the map; see CMap::CPair. If the element is the last in the map, the value is **NULL**.

Remarks

Call this method to iterate through all the elements in the map. Retrieve the first element with a call to **PGetFirstAssoc** and then iterate through the map with successive calls to **PGetNextAssoc**.

Example

See the example for CMap::PGetFirstAssoc.

See Also

Reference

CMap Class
Hierarchy Chart
CMap::PGetFirstAssoc
CMap::PLookup

Other Resources

CMap Members

CMap::PLookup

Finds the value mapped to a given key.

```
const CPair* PLookup(
   ARG_KEY key
) const;
CPair* PLookup(
   ARG_KEY key
);
```

Parameters

key

Key for the element to be searched for.

Return Value

A pointer to a key structure; see CMap::CPair.

Remarks

Call this method to search for a map element with a key that exactly matches the given key.

Example

```
typedef CMap<int, int, CPoint, CPoint> CMyMap;
CMyMap myMap;
int i;

myMap.InitHashTable( 257 );

// Add 10 elements to the map.
for (i=0;i <= 10;i++)
   myMap[i] = CPoint(i, i);

// Print the element values with even key values.
CPoint pt;
CMyMap::CPair *pCurVal;

for (i=0; i <= 10 ;i+=2)
{
   pCurVal= myMap.PLookup(i);
   printf("Current key value at %d: %d,%d\n", pCurVal->key, pCurVal->value.x, pCurVal->valu
e.y);
}
```

See Also

Reference

CMap Class
Hierarchy Chart
CMap::PGetNextAssoc
CMap::PGetFirstAssoc

Other Resources

CMap Members

CMap::RemoveAll

Removes all the values from this map by calling the global helper function **DestructElements**.

```
void RemoveAll( );
```

Remarks

The function works correctly if the map is already empty.

Example

```
CMap<int,int,CPoint,CPoint> myMap;

// Add 10 elements to the map.
for (int i=0;i < 10;i++)
   myMap.SetAt( i, CPoint(i, i) );

myMap.RemoveAll();

ASSERT(myMap.IsEmpty());
```

See Also

Reference

CMap Class

Hierarchy Chart

CMap::RemoveKey

Other Resources

CMap Members

CMap::RemoveKey

Looks up the map entry corresponding to the supplied key; then, if the key is found, removes the entry.

```
BOOL RemoveKey(
      ARG_KEY key
);
```

Parameters

ARG_KEY

Template parameter specifying the type of the key.

key

Key for the element to be removed.

Return Value

Nonzero if the entry was found and successfully removed; otherwise 0.

Remarks

The **DestructElements** helper function is used to remove the entry.

Example

See the example for CMap::SetAt.

See Also

Reference

CMap Class
Hierarchy Chart
CMap::RemoveAll

Other Resources

CMap Members

CMap::SetAt

The primary means to insert an element in a map.

```
void SetAt(
      ARG_KEY key,
      ARG_VALUE newValue
);
```

Parameters

ARG_KEY

Template parameter specifying the type of the *key* parameter.

key

Specifies the key of the new element.

ARG_VALUE

Template parameter specifying the type of the *newValue* parameter.

newValue

Specifies the value of the new element.

Remarks

First, the key is looked up. If the key is found, then the corresponding value is changed; otherwise a new key-value pair is created.

Example

```
CMap<int,int,CPoint,CPoint> myMap;

   // Add 10 elements to the map.
   for (int i=0;i < 10;i++)
      myMap.SetAt( i, CPoint(i, i) );

   // Remove the elements with even key values.
   POSITION pos = myMap.GetStartPosition();
   int    nKey;
   CPoint pt;
   while (pos != NULL)
   {
      myMap.GetNextAssoc( pos, nKey, pt );

      if ((nKey%2) == 0)
         myMap.RemoveKey( nKey );
   }

#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "myMap: " << &myMap << "\n";
#endif
```

See Also

Reference

CMap Class
Hierarchy Chart
CMap::Lookup
CMap::operator [ ]

Other Resources

CMap Members

CMap Data Members

For information about the data members in **CMap**, see CMap Members

CMap::CPair

Contains a key value and the value of the associated object.

Remarks

This is a nested structure within class CMap.

The structure is composed of two fields:

- **key** The actual value of the key type.
- **value** The value of the associated object.

It is used to store the return values from CMap::PLookup, CMap::PGetFirstAssoc, and CMap::PGetNextAssoc.

Example

For an example of usage, see the example for CMap::PLookup.

See Also

Reference

CMap Class

Hierarchy Chart

Other Resources

CMap Members

CMap Operators

For information about the operators in **CMap**, see CMap Members.

CMap::operator [ ]

A convenient substitute for the **SetAt** member function.

```
VALUE& operator[](
      ARG_KEY key
);
```

Parameters

VALUE

Template parameter specifying the type of the map value.

ARG_KEY

Template parameter specifying the type of the key value.

key

The key used to retrieve the value from the map.

Remarks

Thus it can be used only on the left side of an assignment statement (an l-value). If there is no map element with the specified key, then a new element is created.

There is no "right side" (r-value) equivalent to this operator because there is a possibility that a key may not be found in the map. Use the **Lookup** member function for element retrieval.

Example

See the example for CMap::Lookup.

See Also

Reference

CMap Class
Hierarchy Chart
CMap::SetAt
CMap::Lookup

Other Resources

CMap Members

CMapPtrToPtr Class

Supports maps of void pointers keyed by void pointers.

```
class CMapPtrToPtr : public CObject
```

Remarks

The member functions of **CMapPtrToPtr** are similar to the member functions of class CMapStringToOb. Because of this similarity, you can use the **CMapStringToOb** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a function parameter or return value, substitute a pointer to **void**. Wherever you see a **CString** or a **const** pointer to **char** as a function parameter or return value, substitute a pointer to **void**.

```
BOOL CMapStringToOb::Lookup( const char* <key>,
                             CObject*& <rValue> ) const;
```

for example, translates to

```
BOOL CMapPtrToPtr::Lookup( void* <key>, void*& <rValue> ) const;
```

CMapPtrToPtr incorporates the **IMPLEMENT_DYNAMIC** macro to support run-time type access and dumping to a **CDumpContext** object. If you need a dump of individual map elements (pointer values), you must set the depth of the dump context to 1 or greater.

Pointer-to-pointer maps may not be serialized.

When a **CMapPtrToPtr** object is deleted, or when its elements are removed, only the pointers are removed, not the entities they reference.

For more information on **CMapPtrToPtr**, see the article Collections.

Requirements

Header: afxcoll.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class

Hierarchy Chart

Other Resources

CMapPtrToPtr Members

CMapPtrToPtr Members

Base Class Members

CObject Members

Construction

CMapPtrToPtr	Constructs a collection that maps void pointers to void pointers.

Operations

GetHashTableSize	Determines the current number of elements in the hash table.
HashKey	Calculates the hash value of a specified key.
InitHashTable	Initializes the hash table.
GetNextAssoc	Gets the next element for iterating.
GetStartPosition	Returns the position of the first element.
Lookup	Looks up a void pointer based on the void pointer key. The pointer value, not the entity it points to, is used f or the key comparison.
operator [ ]	Inserts an element into the map — operator substitution for **SetAt**.
RemoveAll	Removes all the elements from this map.
RemoveKey	Removes an element specified by a key.
SetAt	Inserts an element into the map; replaces an existing element if a matching key is found.

Status

GetSize	Returns the number of elements in this map.
GetCount	Returns the number of elements in this map.
IsEmpty	Tests for the empty-map condition (no elements).

See Also

Reference

CMapPtrToPtr Class

Hierarchy Chart

CMapPtrToWord Class

Supports maps of 16-bit words keyed by void pointers.

```
class CMapPtrToWord : public CObject
```

Remarks

The member functions of **CMapPtrToWord** are similar to the member functions of class CMapStringToOb. Because of this similarity, you can use the **CMapStringToOb** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a function parameter or return value, substitute **WORD**. Wherever you see a **CString** or a **const** pointer to **char** as a function parameter or return value, substitute a pointer to **void**.

```
BOOL CMapStringToOb::Lookup( const char* <key>,
                             CObject*& <rValue> ) const;
```

for example, translates to

```
BOOL CMapPtrToWord::Lookup( const void* <key>, WORD& <rValue> ) const;
```

CMapWordToPtr incorporates the **IMPLEMENT_DYNAMIC** macro to support run-time type access and dumping to a **CDumpContext** object. If you need a dump of individual map elements, you must set the depth of the dump context to 1 or greater.

Pointer-to-word maps may not be serialized.

When a **CMapPtrToWord** object is deleted, or when its elements are removed, the pointers and the words are removed. The entities referenced by the key pointers are not removed.

For more information on **CMapPtrToWord**, see the article Collections.

Requirements

Header: afxcoll.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class

Hierarchy Chart

Other Resources

CMapPtrToWord Members

CMapPtrToWord Members

Base Class Members

COBJECT Members

Construction

CMapPtrToWord	Constructs a collection that maps void pointers to 16-bit words.

Operations

GetHashTableSize	Determines the current number of elements in the hash table.
HashKey	Calculates the hash value of a specified key.
InitHashTable	Initializes the hash table.
GetNextAssoc	Gets the next element for iterating.
GetStartPosition	Returns the position of the first element.
Lookup	Returns a **WORD** using a void pointer as a key. The pointer value, not the entity it points to, is used for the key comparison.
operator []	Inserts an element into the map — operator substitution for **SetAt**.
RemoveAll	Removes all the elements from this map.
RemoveKey	Removes an element specified by a key.
SetAt	Inserts an element into the map; replaces an existing element if a matching key is found.

Status

GetSize	Returns the number of elements in this map.
GetCount	Returns the number of elements in this map.
IsEmpty	Tests for the empty-map condition (no elements).

See Also

Reference

CMapPtrToWord Class

Hierarchy Chart

CMapStringToOb Class

A dictionary collection class that maps unique **CString** objects to **CObject** pointers.

```
class CMapStringToOb : public CObject
```

Remarks

Once you have inserted a **CString**-**CObject*** pair (element) into the map, you can efficiently retrieve or delete the pair using a string or a **CString** value as a key. You can also iterate over all the elements in the map.

A variable of type **POSITION** is used for alternate entry access in all map variations. You can use a **POSITION** to "remember" an entry and to iterate through the map. You might think that this iteration is sequential by key value; it is not. The sequence of retrieved elements is indeterminate.

CMapStringToOb incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. Each element is serialized in turn if a map is stored to an archive, either with the overloaded insertion (**<<**) operator or with the **Serialize** member function.

If you need a diagnostic dump of the individual elements in the map (the **CString** value and the **CObject** contents), you must set the depth of the dump context to 1 or greater.

When a **CMapStringToOb** object is deleted, or when its elements are removed, the **CString** objects and the **CObject** pointers are removed. The objects referenced by the **CObject** pointers are not destroyed.

Map class derivation is similar to list derivation. See the article Collections for an illustration of the derivation of a special-purpose list class.

Requirements

Header: afxcoll.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class
Hierarchy Chart
CMapPtrToPtr Class
CMapPtrToWord Class
CMapStringToPtr Class
CMapStringToString Class
CMapWordToOb Class
CMapWordToPtr Class

Other Resources

CMapStringToOb Members

CMapStringToOb Members

Base Class Members

CObject Members

Construction

CMapStringToOb	Constructs a collection that maps **CString** values to **CObject** pointers.

Operations

GetNextAssoc	Gets the next element for iterating.
GetStartPosition	Returns the position of the first element.
GetHashTableSize	Determines the current number of elements in the hash table.
HashKey	Calculates the hash value of a specified key.
InitHashTable	Initializes the hash table.
LookupKey	Returns a key value associated with the specified key value.
Lookup	Returns a **CObject** pointer based on a **CString** value.
RemoveAll	Removes all the elements from this map.
RemoveKey	Removes an element specified by a key.
SetAt	Inserts an element into the map; replaces an existing element if a matching key is found.

Status

GetCount	Returns the number of elements in this map.
GetSize	Returns the number of elements in this map.
IsEmpty	Tests for the empty-map condition (no elements).

Operators

operator []	Inserts an element into the map — operator substitution for **SetAt**.

See Also

Reference

CMapStringToOb Class
Hierarchy Chart

CMapStringToOb Member Functions

For information about the member functions in **CMapStringToOb**, see CMapStringToOb Members.

CMapStringToOb::CMapStringToOb

Constructs an empty **CString**-to-**CObject*** map.

```
CMapStringToOb(
   INT_PTR nBlockSize = 10
);
```

Parameters

nBlockSize

Specifies the memory-allocation granularity for extending the map.

Remarks

As the map grows, memory is allocated in units of *nBlockSize* entries.

The following table shows other member functions that are similar to **CMapStringToOb:: CMapStringToOb**.

Class	Member Function
CMapPtrToPtr	**CMapPtrToPtr(INT_PTR** *nBlockSize* **= 10);**
CMapPtrToWord	**CMapPtrToWord(INT_PTR** *nBlockSize* **= 10);**
CMapStringToPtr	**CMapStringToPtr(INT_PTR** *nBlockSize* **= 10);**
CMapStringToString	**CMapStringToString(INT_PTR** *nBlockSize* **= 10);**
CMapWordToOb	**CMapWordToOb(INT_PTR** *nBlockSize* **= 10);**
CMapWordToPtr	**MapWordToPtr(INT_PTR** *nBlockSize* **= 10);**

Example

```
// example for CMapStringToOb::CMapStringToOb
CMapStringToOb map(20);  // Map on the stack with blocksize of 20

CMapStringToOb* pm = new CMapStringToOb;  // Map on the heap
                                          // with default blocksize
```

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

See Also

Reference

CMapStringToOb Class

Hierarchy Chart

Other Resources

CMapStringToOb Members

CMapStringToOb::GetCount

Determines how many elements are in the map.

```
INT_PTR GetCount( ) const;
```

Return Value

The number of elements in this map.

Remarks

The following table shows other member functions that are similar to **CMapStringToOb::GetCount**.

Class	Member Function
CMapPtrToPtr	**INT_PTR GetCount( ) const;**
CMapPtrToWord	**INT_PTR GetCount( ) const;**
CMapStringToPtr	**INT_PTR GetCount( ) const;**
CMapStringToString	**INT_PTR GetCount( ) const;**
CMapWordToOb	**INT_PTR GetCount( ) const;**
CMapWordToPtr	**INT_PTR GetCount( ) const;**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
// example for CMapStringToOb::GetCount
CMapStringToOb map;

map.SetAt( "Bart", new CAge( 13 ) );
map.SetAt( "Homer", new CAge( 36 ) );
ASSERT( map.GetCount() == 2 );
```

See Also

Reference

CMapStringToOb Class
Hierarchy Chart
CMapStringToOb::IsEmpty

Other Resources

CMapStringToOb Members

CMapStringToOb::GetHashTableSize

Determines the current number of elements in the hash table.

```
UINT GetHashTableSize( ) const;
```

Return Value

Returns the number of elements in the hash table.

Remarks

The following table shows other member functions that are similar to **CMapStringToOb::GetHashTableSize**.

Class	Member Function
CMapPtrToPtr	**UINT GetHashTableSize( ) const;**
CMapPtrToWord	**UINT GetHashTableSize( ) const;**
CMapStringToPtr	**UINT GetHashTableSize( ) const;**
CMapStringToString	**UINT GetHashTableSize( ) const;**
CMapWordToOb	**UINT GetHashTableSize( ) const;**
CMapWordToPtr	**UINT GetHashTableSize( ) const;**

See Also

Reference

CMapStringToOb Class

Hierarchy Chart

Other Resources

CMapStringToOb Members

CMapStringToOb::GetNextAssoc

Retrieves the map element at *rNextPosition*, then updates *rNextPosition* to refer to the next element in the map.

```
void GetNextAssoc(
   POSITION& rNextPosition,
   CString& rKey,
   CObject*& rValue
) const;
```

Parameters

rNextPosition

Specifies a reference to a **POSITION** value returned by a previous **GetNextAssoc** or **GetStartPosition** call.

rKey

Specifies the returned key of the retrieved element (a string).

rValue

Specifies the returned value of the retrieved element (a **CObject** pointer). See Remarks for more about this parameter.

Remarks

This function is most useful for iterating through all the elements in the map. Note that the position sequence is not necessarily the same as the key value sequence.

If the retrieved element is the last in the map, then the new value of *rNextPosition* is set to **NULL**.

For the *rValue* parameter, be sure to cast your object type to **CObject*&**, which is what the compiler requires, as shown in the following example:

```
CMyObject* ob;
map.GetNextAssoc(pos, key, (CObject*&)ob);
```

This is not true of **GetNextAssoc** for maps based on templates.

The following table shows other member functions that are similar to **CMapStringToOb::GetNextAssoc**.

Class	Member Function
CMapPtrToPtr	**void GetNextAssoc(POSITION&** *rNextPosition* **, void*&** *rKey* **, void*&** *rValue* **) const;**
CMapPtrToWord	**void GetNextAssoc(POSITION&** *rNextPosition* **, void*&** *rKey* **, WORD&** *rValue* **) const;**
CMapStringToPtr	**void GetNextAssoc(POSITION&** *rNextPosition* **, CString&** *rKey* **, void*&** *rValue* **) const;**
CMapStringToString	**void GetNextAssoc(POSITION&** *rNextPosition* **, CString&** *rKey* **, CString&** *rValue* **) const;**
CMapWordToOb	**void GetNextAssoc(POSITION&** *rNextPosition* **, WORD&** *rKey* **, CObject*&** *rValue* **) const;**
CMapWordToPtr	**void GetNextAssoc(POSITION&** *rNextPosition* **, WORD&** *rKey* **, void*&** *rValue* **) const;**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
// example for CMapStringToOb::GetNextAssoc and CMapStringToOb::GetStartPosition
   CMapStringToOb map;
   POSITION pos;
   CString key;
   CAge* pa;
```

```
   map.SetAt( "Bart", new CAge( 13 ) );
   map.SetAt( "Lisa", new CAge( 11 ) );
   map.SetAt( "Homer", new CAge( 36 ) );
   map.SetAt( "Marge", new CAge( 35 ) );
   // Iterate through the entire map, dumping both name and age.
   for( pos = map.GetStartPosition(); pos != NULL; )
   {
   map.GetNextAssoc( pos, key, (CObject*&)pa );
#ifdef _DEBUG
      afxDump << key << " : " << pa << "\n";
#endif
   }
```

The results from this program are as follows:

```
Lisa : a CAge at $4724 11
Marge : a CAge at $47A8 35
Homer : a CAge at $4766 36
Bart : a CAge at $45D4 13
```

See Also

Reference

CMapStringToOb Class

Hierarchy Chart

CMapStringToOb::GetStartPosition

Other Resources

CMapStringToOb Members

CMapStringToOb::GetSize

Returns the number of map elements.

```
INT_PTR GetSize( ) const;
```

Return Value

The number of items in the map.

Remarks

Call this method to retrieve the number of elements in the map.

The following table shows other member functions that are similar to **CMapStringToOb::GetSize**.

Class	Member Function
CMapPtrToPtr	**INT_PTR GetSize( ) const;**
CMapPtrToWord	**INT_PTR GetSize( ) const;**
CMapStringToPtr	**INT_PTR GetSize( ) const;**
CMapStringToString	**INT_PTR GetSize( ) const;**
CMapWordToOb	**INT_PTR GetSize( ) const;**
CMapWordToPtr	**INT_PTR GetSize( ) const;**

Example

```
// example for CMapStringToOb::GetSize
CMapStringToOb map;

map.SetAt( "Bart", new CAge( 13 ) );
map.SetAt( "Homer", new CAge( 36 ) );
ASSERT( map.GetSize() == 2 );
```

See Also

Reference

CMapStringToOb Class

Hierarchy Chart

Other Resources

CMapStringToOb Members

CMapStringToOb::GetStartPosition

Starts a map iteration by returning a **POSITION** value that can be passed to a **GetNextAssoc** call.

```
POSITION GetStartPosition( ) const;
```

Return Value

A **POSITION** value that indicates a starting position for iterating the map; or **NULL** if the map is empty.

Remarks

The iteration sequence is not predictable; therefore, the "first element in the map" has no special significance.

The following table shows other member functions that are similar to **CMapStringToOb::GetStartPosition**.

Class	Member Function
CMapPtrToPtr	**POSITION GetStartPosition( ) const;**
CMapPtrToWord	**POSITION GetStartPosition( ) const;**
CMapStringToPtr	**POSITION GetStartPosition( ) const;**
CMapStringToString	**POSITION GetStartPosition( ) const;**
CMapWordToOb	**POSITION GetStartPosition( ) const;**
CMapWordToPtr	**POSITION GetStartPosition( ) const;**

Example

See the example for CMapStringToOb::GetNextAssoc.

See Also

Reference

CMapStringToOb Class

Hierarchy Chart

Other Resources

CMapStringToOb Members

CMapStringToOb::HashKey

Calculates the hash value of a specified key.

```
UINT HashKey(
   LPCTSTR key
) const;
```

Parameters

key

The key whose hash value is to be calculated.

Return Value

The Key's hash value

Remarks

The following table shows other member functions that are similar to **CMapStringToOb::HashKey**.

Class	Member Function
CMapPtrToPtr	**UINT HashKey(void* *key*) const;**
CMapPtrToWord	**UINT HashKey(void* *key*) const;**
CMapStringToString	**UINT HashKey(LPCTSTR *key*) const;**
CMapStringToPtr	**UINT HashKey(LPCTSTR *key*) const;**
CMapWordToOb	**UINT HashKey(WORD *key*) const;**
CMapWordToPtr	**UINT HashKey(WORD *key*) const;**

See Also

Reference

CMapStringToOb Class

Hierarchy Chart

Other Resources

CMapStringToOb Members

CMapStringToOb::InitHashTable

Initializes the hash table.

```
void InitHashTable(
   UINT hashSize,
   BOOL bAllocNow = TRUE
);
```

Parameters

hashSize

Number of entries in the hash table.

bAllocNow

If **TRUE**, allocates the hash table upon initialization; otherwise the table is allocated when needed.

Remarks

For best performance, the hash table size should be a prime number. To minimize collisions, the size should be roughly 20 percent larger than the largest anticipated data set.

The following table shows other member functions that are similar to **CMapStringToOb::InitHashTable**.

Class	Member Function
CMapPtrToPtr	**void InitHashTable(UINT** *hashSize* **, BOOL** *bAllocNow* **= TRUE);**
CMapPtrToWord	**void InitHashTable(UINT** *hashSize* **, BOOL** *bAllocNow* **= TRUE);**
CMapStringToString	**void InitHashTable(UINT** *hashSize* **, BOOL** *bAllocNow* **= TRUE);**
CMapStringToPtr	**void InitHashTable(UINT** *hashSize* **, BOOL** *bAllocNow* **= TRUE);**
CMapWordToOb	**void InitHashTable(UINT** *hashSize* **, BOOL** *bAllocNow* **= TRUE);**
CMapWordToPtr	**void InitHashTable(UINT** *hashSize* **, BOOL** *bAllocNow* **= TRUE);**

See Also

Reference

CMapStringToOb Class

Hierarchy Chart

Other Resources

CMapStringToOb Members

CMapStringToOb::IsEmpty

Determines whether the map is empty.

```
BOOL IsEmpty( ) const;
```

Return Value

Nonzero if this map contains no elements; otherwise 0.

Example

See the example for RemoveAll.

Remarks

The following table shows other member functions that are similar to **CMapStringToOb:: IsEmpty**.

Class	Member Function
CMapPtrToPtr	**BOOL IsEmpty( ) const;**
CMapPtrToWord	**BOOL IsEmpty( ) const;**
CMapStringToPtr	**BOOL IsEmpty( ) const;**
CMapStringToString	**BOOL IsEmpty( ) const;**
CMapWordToOb	**BOOL IsEmpty( ) const;**
CMapWordToPtr	**BOOL IsEmpty( ) const;**

See Also

Reference

CMapStringToOb Class

Hierarchy Chart

Other Resources

CMapStringToOb Members

CMapStringToOb::Lookup

Returns a **CObject** pointer based on a **CString** value.

```
BOOL Lookup(
   LPCTSTR key,
   CObject*& rValue
) const;
```

Parameters

key

Specifies the string key that identifies the element to be looked up.

rValue

Specifies the returned value from the looked-up element.

Return Value

Nonzero if the element was found; otherwise 0.

Remarks

Lookup uses a hashing algorithm to quickly find the map element with a key that matches exactly (**CString** value).

The following table shows other member functions that are similar to **CMapStringToOb::LookUp**.

Class	Member Function
CMapPtrToPtr	**BOOL Lookup(void*** *key* **, void*&** *rValue* **) const;**
CMapPtrToWord	**BOOL Lookup(void*** *key* **, WORD&** *rValue* **) const;**
CMapStringToPtr	**BOOL Lookup(LPCTSTR** *key* **, void*&** *rValue* **) const;**
CMapStringToString	**BOOL Lookup(LPCTSTR** *key* **, CString&** *rValue* **) const;**
CMapWordToOb	**BOOL Lookup(WORD** *key* **, CObject*&** *rValue* **) const;**
CMapWordToPtr	**BOOL Lookup(WORD** *key* **, void*&** *rValue* **) const;**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
// example for CMapStringToOb::LookUp

CMapStringToOb map;
CAge* pa;

map.SetAt( "Bart", new CAge( 13 ) );
map.SetAt( "Lisa", new CAge( 11 ) );
map.SetAt( "Homer", new CAge( 36 ) );
map.SetAt( "Marge", new CAge( 35 ) );
ASSERT( map.Lookup( "Lisa", ( CObject*& ) pa ) ); // Is "Lisa" in the map?
ASSERT( *pa ==  CAge( 11 ) ); // Is she 11?
```

See Also

Reference

CMapStringToOb Class

Hierarchy Chart

CMapStringToOb::operator [ ]

Other Resources

CMapStringToOb Members

CMapStringToOb::LookupKey

Returns a reference to the key associated with the specified key value.

```
BOOL LookupKey(
   LPCTSTR key,
   LPCTSTR& rKey
) const;
```

Parameters

key

Specifies the string key that identifies the element to be looked up.

rKey

The reference to the associated key.

Return Value

Nonzero if the key was found; otherwise 0.

Remarks

Using a reference to a key is unsafe if used after the associated element was removed from the map or after the map was destroyed.

The following table shows other member functions that are similar to **CMapStringToOb:: LookupKey**.

Class	Member Function
CMapStringToPtr	**BOOL LookupKey(LPCTSTR** *key* **, LPCTSTR&** *rKey* **) const;**
CMapStringToString	**BOOL LookupKey(LPCTSTR** *key* **, LPCTSTR&** *rKey* **) const;**

See Also

Reference

CMapStringToOb Class

Hierarchy Chart

Other Resources

CMapStringToOb Members

CMapStringToOb::RemoveAll

Removes all the elements from this map and destroys the **CString** key objects.

```
void RemoveAll( );
```

Remarks

The **CObject** objects referenced by each key are not destroyed. The **RemoveAll** function can cause memory leaks if you do not ensure that the referenced **CObject** objects are destroyed.

The function works correctly if the map is already empty.

The following table shows other member functions that are similar to **CMapStringToOb::RemoveAll**.

Class	Member Function
CMapPtrToPtr	**void RemoveAll( );**
CMapPtrToWord	**void RemoveAll( );**
CMapStringToPtr	**void RemoveAll( );**
CMapStringToString	**void RemoveAll( );**
CMapWordToOb	**void RemoveAll( );**
CMapWordToPtr	**void RemoveAll( );**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
// example for CMapStringToOb::RemoveAll
{
   CMapStringToOb map;

   CAge age1( 13 ); // Two objects on the stack
   CAge age2( 36 );
   map.SetAt( "Bart", &age1 );
   map.SetAt( "Homer", &age2 );
   ASSERT( map.GetCount() == 2 );
   map.RemoveAll(); // CObject pointers removed; objects not removed.
   ASSERT( map.GetCount() == 0 );
   ASSERT( map.IsEmpty() );
} // The two CAge objects are deleted when they go out of scope.
```

See Also

Reference

CMapStringToOb Class
Hierarchy Chart
CMapStringToOb::RemoveKey

Other Resources

CMapStringToOb Members

CMapStringToOb::RemoveKey

Looks up the map entry corresponding to the supplied key; then, if the key is found, removes the entry.

```
BOOL RemoveKey(
   LPCTSTR key
);
```

Parameters

key

Specifies the string used for map lookup.

Return Value

Nonzero if the entry was found and successfully removed; otherwise 0.

Remarks

This can cause memory leaks if the **CObject** object is not deleted elsewhere.

The following table shows other member functions that are similar to **CMapStringToOb::RemoveKey**.

Class	Member Function
CMapPtrToPtr	**BOOL RemoveKey(void*** *key* **);**
CMapPtrToWord	**BOOL RemoveKey(void*** *key* **);**
CMapStringToPtr	**BOOL RemoveKey(LPCTSTR** *key* **);**
CMapStringToString	**BOOL RemoveKey(LPCTSTR** *key* **);**
CMapWordToOb	**BOOL RemoveKey(WORD** *key* **);**
CMapWordToPtr	**BOOL RemoveKey(WORD** *key* **);**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
// example for CMapStringToOb::RemoveKey
   CMapStringToOb map;

   map.SetAt( "Bart", new CAge( 13 ) );
   map.SetAt( "Lisa", new CAge( 11 ) );
   map.SetAt( "Homer", new CAge( 36 ) );
   map.SetAt( "Marge", new CAge( 35 ) );
   map.RemoveKey( "Lisa" ); // Memory leak: CAge object not
                            // deleted.
#ifdef _DEBUG
   afxDump.SetDepth( 1 );
   afxDump << "RemoveKey example: " << &map << "\n";
#endif
```

The results from this program are as follows:

```
RemoveKey example: A CMapStringToOb with 3 elements
    [Marge] = a CAge at $49A0 35
    [Homer] = a CAge at $495E 36
    [Bart] = a CAge at $4634 13
```

See Also

Reference

CMapStringToOb Class

Hierarchy Chart

CMapStringToOb::RemoveAll

Other Resources

CMapStringToOb Members

CMapStringToOb::SetAt

The primary means to insert an element in a map.

```
void SetAt(
   LPCTSTR key,
   CObject* newValue
);
```

Parameters

key

Specifies the string that is the key of the new element.

newValue

Specifies the **CObject** pointer that is the value of the new element.

Remarks

First, the key is looked up. If the key is found, then the corresponding value is changed; otherwise a new key-value element is created.

The following table shows other member functions that are similar to **CMapStringToOb::SetAt**.

Class	Member Function
CMapPtrToPtr	**void SetAt(void*** *key* **, void*** *newValue* **);**
CMapPtrToWord	**void SetAt(void*** *key* **, WORD** *newValue* **);**
CMapStringToPtr	**void SetAt(LPCTSTR** *key* **, void*** *newValue* **);**
CMapStringToString	**void SetAt(LPCTSTR** *key* **, LPCTSTR** *newValue* **);**
CMapWordToOb	**void SetAt(WORD** *key* **, CObject*** *newValue* **);**
CMapWordToPtr	**void SetAt(WORD** *key* **, void*** *newValue* **);**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
// example for CMapStringToOb::SetAt
   CMapStringToOb map;
   CAge* pa;

   map.SetAt( "Bart", new CAge( 13 ) );
   map.SetAt( "Lisa", new CAge( 11 ) ); // Map contains 2
                                        // elements.
#ifdef _DEBUG
   afxDump.SetDepth( 1 );
   afxDump << "before Lisa's birthday: " << &map << "\n";
#endif
   if( map.Lookup( "Lisa", (CObject *&)pa ) )
   { // CAge 12 pointer replaces CAge 11 pointer.
      map.SetAt( "Lisa", new CAge( 12 ) );
      delete pa;  // Must delete CAge 11 to avoid memory leak.
   }
#ifdef _DEBUG
   afxDump << "after Lisa's birthday: " << &map << "\n";
#endif
```

The results from this program are as follows:

```
before Lisa's birthday: A CMapStringToOb with 2 elements
    [Lisa] = a CAge at $493C 11
    [Bart] = a CAge at $4654 13
after Lisa's birthday: A CMapStringToOb with 2 elements
    [Lisa] = a CAge at $49C0 12
    [Bart] = a CAge at $4654 13
```

See Also

Reference

CMapStringToOb Class

Hierarchy Chart

CMapStringToOb::Lookup

CMapStringToOb::operator [ ]

Other Resources

CMapStringToOb Members

CMapStringToOb Operators

For information about the operators in **CMapStringToOb**, see CMapStringToOb Members.

CMapStringToOb::operator [ ]

A convenient substitute for the **SetAt** member function.

```
COBJECT*& operator [ ](
   LPCTSTR key
);
```

Return Value

A reference to a pointer to a **CObject** object; or **NULL** if the map is empty or *key* is out of range.

Remarks

Thus it can be used only on the left side of an assignment statement (an l-value). If there is no map element with the specified key, then a new element is created.

There is no "right side" (r-value) equivalent to this operator because there is a possibility that a key may not be found in the map. Use the **Lookup** member function for element retrieval.

The following table shows other member functions that are similar to **CMapStringToOb::operator []**.

Class	Member Function
CMapPtrToPtr	**void*& operator[](void*** *key* **);**
CMapPtrToWord	**WORD& operator[](void*** *key* **);**
CMapStringToPtr	**void*& operator[](LPCTSTR** *key* **);**
CMapStringToString	**CString& operator[](LPCTSTR** *key* **);**
CMapWordToOb	**CObject*& operator[](WORD** *key* **);**
CMapWordToPtr	**void*& operator[](WORD** *key* **);**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
// example for CMapStringToOb::operator[]
   CMapStringToOb map;

   map["Bart"] = new CAge( 13 );
   map["Lisa"] = new CAge( 11 );
#ifdef _DEBUG
   afxDump.SetDepth( 1 );
   afxDump << "Operator [] example: " << &map << "\n";
#endif
```

The results from this program are as follows:

```
Operator [] example: A CMapStringToOb with 2 elements
    [Lisa] = a CAge at $4A02 11
    [Bart] = a CAge at $497E 13
```

See Also

Reference

CMapStringToOb Class

Hierarchy Chart

CMapStringToOb::SetAt
CMapStringToOb::Lookup

Other Resources

CMapStringToOb Members

CMapStringToPtr Class

Supports maps of void pointers keyed by **CString** objects.

```
class CMapStringToPtr : public CObject
```

Remarks

The member functions of **CMapStringToPtr** are similar to the member functions of class CMapStringToOb. Because of this similarity, you can use the **CMapStringToOb** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a function parameter or return value, substitute a pointer to **void**.

```
BOOL CMapStringToOb::Lookup( const char* <key>,
                             CObject*& <rValue> ) const;
```

for example, translates to

```
BOOL CMapStringToPtr::Lookup( LPCTSTR <key>, void*& <rValue> )
                              const;
```

CMapStringToPtr incorporates the **IMPLEMENT_DYNAMIC** macro to support run-time type access and dumping to a **CDumpContext** object. If you need a dump of individual map elements, you must set the depth of the dump context to 1 or greater.

String-to-pointer maps may not be serialized.

When a **CMapStringToPtr** object is deleted, or when its elements are removed, the **CString** key objects and the words are removed.

Requirements

Header: afxcoll.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class

Hierarchy Chart

Other Resources

CMapStringToPtr Members

CMapStringToPtr Members

Base Class Members

CObject Members

Construction

CMapStringToPtr	Constructs a collection that maps **CString** objects to void pointers.

Operations

GetHashTableSize	Determines the current number of elements in the hash table.
HashKey	Calculates the hash value of a specified key.
InitHashTable	Initializes the hash table.
GetNextAssoc	Gets the next element for iterating.
GetStartPosition	Returns the position of the first element.
Lookup	Returns a void pointer based on a **CString** value.
LookupKey	Returns the associated key of the specified key.
operator []	Inserts an element into the map — operator substitution for **SetAt**.
RemoveAll	Removes all the elements from this map.
RemoveKey	Removes an element specified by a key.
SetAt	Inserts an element into the map; replaces an existing element if a matching key is found.

Status

GetSize	Returns the number of elements in this map.
GetCount	Returns the number of elements in this map.
IsEmpty	Tests for the empty-map condition (no elements).

See Also

Reference

CMapStringToPtr Class
Hierarchy Chart

CMapStringToString Class

Supports maps of **CString** objects keyed by **CString** objects.

```
class CMapStringToString : public CObject
```

Remarks

The member functions of **CMapStringToString** are similar to the member functions of class CMapStringToOb. Because of this similarity, you can use the **CMapStringToOb** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a return value or "output" function parameter, substitute a pointer to **char**. Wherever you see a **CObject** pointer as an "input" function parameter, substitute a pointer to **char**.

```
BOOL CMapStringToOb::Lookup( const char* <key>,
                             CObject*& <rValue> ) const;
```

for example, translates to

```
BOOL CMapStringToString::Lookup( LPCTSTR <key>,
                                 CString& <rValue> ) const;
```

CMapStringToString incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. Each element is serialized in turn if a map is stored to an archive, either with the overloaded insertion (**<<**) operator or with the **Serialize** member function.

If you need a dump of individual **CString**-**CString** elements, you must set the depth of the dump context to 1 or greater.

When a **CMapStringToString** object is deleted, or when its elements are removed, the **CString** objects are removed as appropriate.

For more information on **CMapStringToString**, see the article Collections.

Requirements

Header: afxcoll.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

COLLECT Sample: Illustrates MFC Collection Classes

Reference

CObject Class

Hierarchy Chart

Other Resources

CMapStringToString Members

CMapStringToString Members

Base Class Members

CObject Members

Construction

CMapStringToString	Constructs a collection that maps **CString** objects to **CString** objects.

Operations

GetNextAssoc	Gets the next element for iterating.
PGetNextAssoc	Gets a pointer to the next **CString** for iterating.
GetHashTableSize	Determines the number of elements in the hash table for the map.
GetStartPosition	Returns the position of the first element.
HashKey	Calculates the hash value of a specified key.
InitHashTable	Initializes the hash table.
PGetFirstAssoc	Gets a pointer to the first **CString** in the map.
Lookup	Returns a **CString** using a **CString** value as a key.
LookupKey	Returns the associated key of the specified key.
PLookup	Returns a pointer to a **CString** whose value matches the specified value.
operator []	Inserts an element into the map — operator substitution for **SetAt**.
RemoveAll	Removes all the elements from this map.
RemoveKey	Removes an element specified by a key.
SetAt	Inserts an element into the map; replaces an existing element if a matching key is found.

Status

GetSize	Returns the number of elements in this map.
GetCount	Returns the number of elements in this map.
IsEmpty	Tests for the empty-map condition (no elements).

Data Members

CMapStringToString::CPair	A nested structure containing a key value and the value of the associated string object.

See Also

Reference

CMapStringToString Class

Hierarchy Chart

CMapStringToString Member Functions

For information about the member functions in **CMapStringToString**, see CMapStringToString Members.

CMapStringToString::PGetFirstAssoc

Returns the first entry of the map object.

```
const CPair* PGetFirstAssoc( ) const;
CPair* PGetFirstAssoc( );
```

Return Value

A pointer to the first entry in the map; see CMapStringToString::CPair. If the map is empty, the value is **NULL**.

Remarks

Call this function to return a pointer the first element in the map object.

Example

```
{
    // ...

    typedef CMapStringToString CMyMap;

    CMyMap myMap;
    CString myStr[4]={"One", "Two", "Three", "Four"};
    CMyMap::CPair* pCurVal;

    myMap.InitHashTable( 257 );

    // Add 4 elements to the map.
    myMap.SetAt(myStr[0], "Odd");
    myMap.SetAt(myStr[1], "Even");
    myMap.SetAt(myStr[2], "Odd");
    myMap.SetAt(myStr[3], "Even");

    pCurVal= myMap.PGetFirstAssoc( );
    while (pCurVal != NULL)
    {
           printf_s("Current key value at %s: %s\n",
                    pCurVal->key, pCurVal->value);
           pCurVal= myMap.PGetNextAssoc(pCurVal);
    }

    // ...
}
```

See Also

Reference

CMapStringToString Class
Hierarchy Chart
CMapStringToString::PGetNextAssoc
CMapStringToString::PLookup

Other Resources

CMapStringToString Members

CMapStringToString::PGetNextAssoc

Retrieves the map element pointed to by *pAssocRec*.

```
const CPair *PGetNextAssoc(
   const CPair* pAssoc
) const;
CPair *PGetNextAssoc(
   const CPair* pAssoc
);
```

Parameters

pAssoc

Points to a map entry returned by a previous PGetNextAssoc or PGetFirstAssoc call.

Return Value

A pointer to the next entry in the map; see CMapStringToString::CPair. If the element is the last in the map, the value is **NULL**.

Remarks

Call this method to iterate through all the elements in the map. Retrieve the first element with a call to **PGetFirstAssoc** and then iterate through the map with successive calls to **PGetNextAssoc**.

Example

See the example for CMapStringToString::PGetFirstAssoc.

See Also

Reference

CMapStringToString Class
Hierarchy Chart
CMapStringToString::PGetFirstAssoc
CMapStringToString::PLookup

Other Resources

CMapStringToString Members

CMapStringToString::PLookup

Looks up the value mapped to a given key.

```
const CPair* PLookup(
   LPCTSTR key
) const;
CPair* PLookup(
   LPCTSTR key
);
```

Parameters

key

A pointer to the key for the element to be searched for.

Return Value

A pointer to the specified key.

Remarks

Call this method to search for a map element with a key that exactly matches the given key.

Example

```
typedef CMapStringToString CMyMap;

CMyMap myMap;
CString myStr[4]={"One", "Two", "Three", "Four"};
int i;

myMap.InitHashTable( 257 );

// Add 4 elements to the map.
myMap.SetAt(myStr[0], "Odd");
myMap.SetAt(myStr[1], "Even");
myMap.SetAt(myStr[2], "Odd");
myMap.SetAt(myStr[3], "Even");

// Print the element values with odd key values.
CMyMap::CPair *pCurVal;

for (i=0; i < 4 ;i+=2)
{
    pCurVal= myMap.PLookup(myStr[i]);
    printf_s("Current key value at %s: %s\n",
             pCurVal->key, pCurVal->value);
}
```

See Also

Reference

CMapStringToString Class
Hierarchy Chart
CMapStringToString::PGetNextAssoc
CMapStringToString::PGetFirstAssoc

Other Resources

CMapStringToString Members

CMapStringToString Data Members

For information about the data members in **CMapStringToString**, see CMapStringToString Members

CMapStringToString::CPair

Contains a key value and the value of the associated string object.

Remarks

This is a nested structure within class CMapStringToString.

The structure is composed of two fields:

- **key** The actual value of the key type.
- **value** The value of the associated object.

It is used to store the return values from CMapStringToString::PLookup, CMapStringToString::PGetFirstAssoc, and CMapStringToString::PGetNextAssoc.

Example

For an example of usage, see the example for CMapStringToString::PLookup.

See Also

Reference

CMapStringToString Class

Hierarchy Chart

Other Resources

CMapStringToString Members

CMapWordToOb Class

Supports maps of **CObject** pointers keyed by 16-bit words.

```
class CMapWordToOb : public CObject
```

Remarks

The member functions of **CMapWordToOb** are similar to the member functions of class CMapStringToOb. Because of this similarity, you can use the **CMapStringToOb** reference documentation for member function specifics. Wherever you see a **CString** or a **const** pointer to **char** as a function parameter or return value, substitute **WORD**.

```
BOOL CMapStringToOb::Lookup( const char* <key>,
                             CObject*& <rValue> ) const;
```

for example, translates to

```
BOOL CMapWordToOb::Lookup( WORD <key>, CObject*& <rValue> ) const;
```

CMapWordToOb incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. Each element is serialized in turn if a map is stored to an archive, either with the overloaded insertion (**<<**) operator or with the **Serialize** member function.

If you need a dump of individual **WORD**-**CObject** elements, you must set the depth of the dump context to 1 or greater.

When a **CMapWordToOb** object is deleted, or when its elements are removed, the **CObject** pointers are removed. The objects referenced by the **CObject** pointers are not destroyed.

For more information on **CMapWordToOb**, see the article Collections.

Requirements

Header: afxcoll.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class

Hierarchy Chart

Other Resources

CMapWordToOb Members

CMapWordToOb Members

Base Class Members

CObject Members

Construction

CMapWordToOb	Constructs a collection that maps words to **CObject** pointers.

Operations

GetHashTableSize	Determines the current number of elements in the hash table.
HashKey	Calculates the hash value of a specified key.
InitHashTable	Initializes the hash table.
GetNextAssoc	Gets the next element for iterating.
GetStartPosition	Returns the position of the first element.
Lookup	Returns a **CObject** pointer using a word value as a key.
operator []	Inserts an element into the map — operator substitution for **SetAt**.
RemoveAll	Removes all the elements from this map.
RemoveKey	Removes an element specified by a key.
SetAt	Inserts an element into the map; replaces an existing element if a matching key is found.

Status

GetSize	Returns the number of elements in this map.
GetCount	Returns the number of elements in this map.
IsEmpty	Tests for the empty-map condition (no elements).

See Also

Reference

CMapWordToOb Class
Hierarchy Chart

CMapWordToPtr Class

Supports maps of void pointers keyed by 16-bit words.

```
class CMapWordToPtr : public CObject
```

Remarks

The member functions of **CMapWordToPtr** are similar to the member functions of class CMapStringToOb. Because of this similarity, you can use the **CMapStringToOb** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a function parameter or return value, substitute a pointer to **void**. Wherever you see a **CString** or a **const** pointer to **char** as a function parameter or return value, substitute **WORD**.

```
BOOL CMapStringToOb::Lookup( const char* <key>,
                             CObject*& <rValue> ) const;
```

for example, translates to

```
BOOL CMapWordToPtr::Lookup( WORD <key>, void*& <rValue> ) const;
```

CMapWordToPtr incorporates the **IMPLEMENT_DYNAMIC** macro to support run-time type access and dumping to a **CDumpContext** object. If you need a dump of individual map elements, you must set the depth of the dump context to 1 or greater.

Word-to-pointer maps may not be serialized.

When a **CMapWordToPtr** object is deleted, or when its elements are removed, the words and the pointers are removed. The entities referenced by the pointers are not removed.

For more information on **CMapWordToPtr**, see the article Collections.

Requirements

Header: afxcoll.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class

Hierarchy Chart

Other Resources

CMapWordToPtr Members

CMapWordToPtr Members

Base Class Members

CObject Members

Construction

CMapWordToPtr	Constructs a collection that maps words to void pointers.

Operations

GetHashTableSize	Determines the current number of elements in the hash table.
HashKey	Calculates the hash value of a specified key.
InitHashTable	Initializes the hash table.
GetNextAssoc	Gets the next element for iterating.
GetStartPosition	Returns the position of the first element.
Lookup	Returns a void pointer using a word value as a key.
operator []	Inserts an element into the map — operator substitution for **SetAt**.
RemoveAll	Removes all the elements from this map.
RemoveKey	Removes an element specified by a key.
SetAt	Inserts an element into the map; replaces an existing element if a matching key is found.

Status

GetSize	Returns the number of elements in this map.
GetCount	Returns the number of elements in this map.
IsEmpty	Tests for the empty-map condition (no elements).

See Also

Reference

CMapWordToPtr Class
Hierarchy Chart

CMDIChildWnd Class

Provides the functionality of a Windows multiple document interface (MDI) child window, along with members for managing the window.

```
class CMDIChildWnd : public CFrameWnd
```

Remarks

An MDI child window looks much like a typical frame window, except that the MDI child window appears inside an MDI frame window rather than on the desktop. An MDI child window does not have a menu bar of its own, but instead shares the menu of the MDI frame window. The framework automatically changes the MDI frame menu to represent the currently active MDI child window.

To create a useful MDI child window for your application, derive a class from **CMDIChildWnd**. Add member variables to the derived class to store data specific to your application. Implement message-handler member functions and a message map in the derived class to specify what happens when messages are directed to the window.

There are three ways to construct an MDI child window:

- Directly construct it using **Create**.
- Directly construct it using **LoadFrame**.
- Indirectly construct it through a document template.

Before you call **Create** or **LoadFrame**, you must construct the frame-window object on the heap using the C++ **new** operator. Before calling **Create** you can also register a window class with the AfxRegisterWndClass global function to set the icon and class styles for the frame.

Use the **Create** member function to pass the frame's creation parameters as immediate arguments.

LoadFrame requires fewer arguments than **Create**, and instead retrieves most of its default values from resources, including the frame's caption, icon, accelerator table, and menu. To be accessible by **LoadFrame**, all these resources must have the same resource ID (for example, **IDR_MAINFRAME**).

When a **CMDIChildWnd** object contains views and documents, they are created indirectly by the framework instead of directly by the programmer. The **CDocTemplate** object orchestrates the creation of the frame, the creation of the containing views, and the connection of the views to the appropriate document. The parameters of the **CDocTemplate** constructor specify the **CRuntimeClass** of the three classes involved (document, frame, and view). A **CRuntimeClass** object is used by the framework to dynamically create new frames when specified by the user (for example, by using the File New command or the MDI Window New command).

A frame-window class derived from **CMDIChildWnd** must be declared with **DECLARE_DYNCREATE** in order for the above **RUNTIME_CLASS** mechanism to work correctly.

The **CMDIChildWnd** class inherits much of its default implementation from **CFrameWnd**. For a detailed list of these features, please refer to the CFrameWnd class description. The **CMDIChildWnd** class has the following additional features:

- In conjunction with the **CMultiDocTemplate** class, multiple **CMDIChildWnd** objects from the same document template share the same menu, saving Windows system resources.
- The currently active MDI child window menu entirely replaces the MDI frame window's menu, and the caption of the currently active MDI child window is added to the MDI frame window's caption. For further examples of MDI child window functions that are implemented in conjunction with an MDI frame window, see the **CMDIFrameWnd** class description.

Do not use the C++ **delete** operator to destroy a frame window. Use **CWnd::DestroyWindow** instead. The **CFrameWnd** implementation of **PostNcDestroy** will delete the C++ object when the window is destroyed. When the user closes the frame window, the default **OnClose** handler will call **DestroyWindow**.

For more information on **CMDIChildWnd**, see Frame Windows.

Requirements

Header: afxwin.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

MDI Sample: Demonstrates MDI Without Using Doc/View Architecture
MDIDOCVW Sample: Demonstrates MDI Using Doc/View Architecture
SNAPVW Sample: Uses Property Pages in a Form View Application

Reference

CFrameWnd Class
Hierarchy Chart
CWnd Class
CMDIFrameWnd Class

Other Resources

CMDIChildWnd Members

CMDIChildWnd Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CFrameWnd Members

Construction

CMDIChildWnd	Constructs a **CMDIChildWnd** object.

Initialization

Create	Creates the Windows MDI child window associated with the **CMDIChildWnd** object.

Operations

GetMDIFrame	Returns the parent MDI frame of the MDI client window.
MDIActivate	Activates this MDI child window.
MDIDestroy	Destroys this MDI child window.
MDIMaximize	Maximizes this MDI child window.
MDIRestore	Restores this MDI child window from maximized or minimized size.
SetHandles	Sets the handles for menu and accelerator resources.

See Also

Reference

CMDIChildWnd Class

Hierarchy Chart

CMDIChildWnd Member Functions

For information about the member functions in **CMDIChildWnd**, see CMDIChildWnd Members.

CMDIChildWnd::CMDIChildWnd

Call to construct a **CMDIChildWnd** object.

```
CMDIChildWnd( );
```

Remarks

Call **Create** to create the visible window.

Example

See the example for CMDIChildWnd::Create.

See Also

Reference

CMDIChildWnd Class
Hierarchy Chart
CMDIChildWnd::Create

Other Resources

CMDIChildWnd Members

CMDIChildWnd::Create

Call this member function to create a Windows MDI child window and attach it to the **CMDIChildWnd** object.

```
virtual BOOL Create(
   LPCTSTR lpszClassName,
   LPCTSTR lpszWindowName,
   DWORD dwStyle = WS_CHILD | WS_VISIBLE | WS_OVERLAPPEDWINDOW,
   const RECT& rect = rectDefault,
   CMDIFrameWnd* pParentWnd = NULL,
   CCreateContext* pContext = NULL
);
```

Parameters

lpszClassName

Points to a null-terminated character string that names the Windows class (a WNDCLASS structure). The class name can be any name registered with the AfxRegisterWndClass global function. Should be **NULL** for a standard **CMDIChildWnd**.

lpszWindowName

Points to a null-terminated character string that represents the window name. Used as text for the title bar.

dwStyle

Specifies the window style attributes. The **WS_CHILD** style is required.

rect

Contains the size and position of the window. The **rectDefault** value allows Windows to specify the size and position of the new **CMDIChildWnd**.

pParentWnd

Specifies the window's parent. If **NULL**, the main application window is used.

pContext

Specifies a CCreateContext structure. This parameter can be **NULL**.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The currently active MDI child frame window can determine the caption of the parent frame window. This feature is disabled by turning off the **FWS_ADDTOTITLE** style bit of the child frame window.

The framework calls this member function in response to a user command to create a child window, and the framework uses the *pContext* parameter to properly connect the child window to the application. When you call **Create**, *pContext* can be **NULL**.

Example

```
// Example 1:
// CMainFrame::OnFileNewCMdiChildWnd() is a menu command handler for the
// CMainFrame class, which in turn is a CMDIFrameWnd-derived class.
// It shows the creation of a standard Windows MDI child window using
// the registered CMDIChildWnd class.
void CMainFrame::OnFileNewCMdiChildWnd()
{
   CMDIChildWnd* pMDIChildWnd = new CMDIChildWnd;
   VERIFY(pMDIChildWnd->Create(
      NULL,                                       // standard CMDIChildWnd class
      _T("My MDIChildWnd"),                       // caption of MDI child window
      WS_CHILD | WS_VISIBLE | WS_OVERLAPPEDWINDOW, // window styles
      rectDefault,                                // default rectangle size
      this));                                     // parent window; can be NULL
```

```
   // the default PostNcDestroy handler will delete this object when destroyed
}

// Example 2:
// CMainFrame::OnHello() is a menu command handler for the CMainFrame
// class, which in turn is a CMDIFrameWnd-derived class.
// It shows the creation of a Windows MDI child window using a custom
// window class. The custom window class is registered in
// CHelloWnd::Create(). CHelloWnd is a CMDIChildWnd-derived class.
void CMainFrame::OnHello()
{
   CHelloWnd *pHelloWnd = new CHelloWnd;
   if (!pHelloWnd->Create(_T("Hello"),
      WS_CHILD | WS_VISIBLE | WS_OVERLAPPEDWINDOW,
      rectDefault, this))
      return;

   // the default PostNcDestroy handler will delete this object when destroyed
}

BOOL CHelloWnd::Create(
   LPCTSTR szTitle,
   LONG style              /* = 0 */,
   const RECT& rect        /* = rectDefault */,
   CMDIFrameWnd* parent    /* = NULL */)
{
   // Setup the shared menu
   if (menu.m_hMenu == NULL)
      menu.LoadMenu(IDR_HELLO);
   m_hMenuShared = menu.m_hMenu;

   // Register a custom WndClass and create a window.
   // This must be done because CHelloWnd has a custom icon.
   LPCTSTR lpszHelloClass =
      AfxRegisterWndClass(CS_HREDRAW | CS_VREDRAW,
      LoadCursor(NULL, IDC_ARROW),
      (HBRUSH) (COLOR_WINDOW+1),
      LoadIcon(AfxGetInstanceHandle(), MAKEINTRESOURCE(IDI_HELLO)));

   return CMDIChildWnd::Create(lpszHelloClass, szTitle, style, rect, parent);
}
```

See Also

Reference

CMDIChildWnd Class
Hierarchy Chart
CMDIChildWnd::CMDIChildWnd
CWnd::PreCreateWindow

Other Resources

CMDIChildWnd Members

CMDIChildWnd::GetMDIFrame

Call this function to return the MDI parent frame.

```
CMDIFrameWnd* GetMDIFrame( );
```

Return Value

A pointer to the MDI parent frame window.

Remarks

The frame returned is two parents removed from the **CMDIChildWnd** and is the parent of the window of type **MDICLIENT** that manages the **CMDIChildWnd** object. Call the GetParent member function to return the **CMDIChildWnd** object's immediate **MDICLIENT** parent as a temporary **CWnd** pointer.

Example

See the example for CMDIFrameWnd::MDISetMenu.

See Also

Reference

CMDIChildWnd Class
Hierarchy Chart
CWnd::GetParent

Other Resources

CMDIChildWnd Members

CMDIChildWnd::MDIActivate

Call this member function to activate an MDI child window independently of the MDI frame window.

```
void MDIActivate( );
```

Remarks

When the frame becomes active, the child window that was last activated will be activated as well.

Example

See the example for CMDIFrameWnd::GetWindowMenuPopup.

See Also

Reference

CMDIChildWnd Class
Hierarchy Chart
CMDIFrameWnd::MDIGetActive
CWnd::OnNcActivate
CMDIFrameWnd::MDINext
WM_MDIACTIVATE

Other Resources

CMDIChildWnd Members

CMDIChildWnd::MDIDestroy

Call this member function to destroy an MDI child window.

```
void MDIDestroy( );
```

Remarks

The member function removes the title of the child window from the frame window and deactivates the child window.

Example

```
// CMainFrame::OnCloseWindow() is a menu command handler for
// CMainFrame class, which in turn is a CMDIFrameWnd-derived
// class. It closes and destroys the current active MDI child window.
void CMainFrame::OnCloseWindow()
{
   CMDIChildWnd* child = MDIGetActive();
   if (child)
      child->MDIDestroy();
}
```

See Also

Reference

CMDIChildWnd Class
Hierarchy Chart
WM_MDIDESTROY
CMDIChildWnd::Create

Other Resources

CMDIChildWnd Members

CMDIChildWnd::MDIMaximize

Call this member function to maximize an MDI child window.

```
void MDIMaximize( );
```

Remarks

When a child window is maximized, Windows resizes it to make its client area fill the client area of the frame window. Windows places the child window's Control menu in the frame's menu bar so that the user can restore or close the child window and adds the title of the child window to the frame-window title.

Example

```
// CMainFrame::OnMaximizeWindow() is a menu command handler for
// CMainFrame class, which in turn is a CMDIFrameWnd-derived
// class. It maximizes the current active MDI child window.
void CMainFrame::OnMaximizeWindow()
{
   BOOL maximized;
   CMDIChildWnd* child = MDIGetActive(&maximized);
   if (child && (!maximized))
      child->MDIMaximize();   // or MDIMaximize(child);
}
```

See Also

Reference

CMDIChildWnd Class
Hierarchy Chart
WM_MDIMAXIMIZE
CMDIChildWnd::MDIRestore

Other Resources

CMDIChildWnd Members

CMDIChildWnd::MDIRestore

Call this member function to restore an MDI child window from maximized or minimized size.

```
void MDIRestore( );
```

Example

```
// CMainFrame::OnRestoreWindow() is a menu command handler for
// CMainFrame class, which in turn is a CMDIFrameWnd-derived class.
// It restores the current active MDI child window from maximized
// or minimized size.
void CMainFrame::OnRestoreWindow()
{
   BOOL maximized;
   CMDIChildWnd* child = MDIGetActive(&maximized);
   if (child && (maximized || child->IsIconic()))
     child->MDIRestore();     // or MDIRestore(child);
}
```

See Also

Reference

CMDIChildWnd Class
Hierarchy Chart
CMDIChildWnd::MDIMaximize
WM_MDIRESTORE

Other Resources

CMDIChildWnd Members

CMDIChildWnd::SetHandles

Sets the handles for menu and accelerator resources.

```
void SetHandles(
   HMENU hMenu,
   HACCEL hAccel
);
```

Parameters

hMenu

The handle of a menu resource.

hAccel

The handle of an accelerator resource.

Remarks

Call this function to set the menu and accelerator resources used by the MDI child window object.

See Also

Reference

CMDIChildWnd Class

Hierarchy Chart

Other Resources

CMDIChildWnd Members

CMDIFrameWnd Class

Provides the functionality of a Windows multiple document interface (MDI) frame window, along with members for managing the window.

```
class CMDIFrameWnd : public CFrameWnd
```

Remarks

To create a useful MDI frame window for your application, derive a class from **CMDIFrameWnd**. Add member variables to the derived class to store data specific to your application. Implement message-handler member functions and a message map in the derived class to specify what happens when messages are directed to the window.

You can construct an MDI frame window by calling the Create or LoadFrame member function of **CFrameWnd**.

Before you call **Create** or **LoadFrame**, you must construct the frame window object on the heap using the C++ **new** operator. Before calling **Create** you can also register a window class with the AfxRegisterWndClass global function to set the icon and class styles for the frame.

Use the **Create** member function to pass the frame's creation parameters as immediate arguments.

LoadFrame requires fewer arguments than **Create**, and instead retrieves most of its default values from resources, including the frame's caption, icon, accelerator table, and menu. To be accessed by **LoadFrame**, all these resources must have the same resource ID (for example, **IDR_MAINFRAME**).

Though **MDIFrameWnd** is derived from **CFrameWnd**, a frame window class derived from **CMDIFrameWnd** need not be declared with **DECLARE_DYNCREATE**.

The **CMDIFrameWnd** class inherits much of its default implementation from **CFrameWnd**. For a detailed list of these features, refer to the CFrameWnd class description. The **CMDIFrameWnd** class has the following additional features:

- An MDI frame window manages the **MDICLIENT** window, repositioning it in conjunction with control bars. The MDI client window is the direct parent of MDI child frame windows. The **WS_HSCROLL** and **WS_VSCROLL** window styles specified on a **CMDIFrameWnd** apply to the MDI client window rather than the main frame window so the user can scroll the MDI client area (as in the Windows Program Manager, for example).
- An MDI frame window owns a default menu that is used as the menu bar when there is no active MDI child window. When there is an active MDI child, the MDI frame window's menu bar is automatically replaced by the MDI child window menu.
- An MDI frame window works in conjunction with the current MDI child window, if there is one. For instance, command messages are delegated to the currently active MDI child before the MDI frame window.
- An MDI frame window has default handlers for the following standard Window menu commands:
    - **ID_WINDOW_TILE_VERT**
    - **ID_WINDOW_TILE_HORZ**
    - **ID_WINDOW_CASCADE**
    - **ID_WINDOW_ARRANGE**
- An MDI frame window also has an implementation of **ID_WINDOW_NEW**, which creates a new frame and view on the current document. An application can override these default command implementations to customize MDI window handling.

Do not use the C++ **delete** operator to destroy a frame window. Use **CWnd::DestroyWindow** instead. The **CFrameWnd** implementation of **PostNcDestroy** will delete the C++ object when the window is destroyed. When the user closes the frame window, the default **OnClose** handler will call **DestroyWindow**.

For more information on **CMDIFrameWnd**, see Frame Windows.

Requirements

Header: afxwin.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

MDI Sample: Demonstrates MDI Without Using Doc/View Architecture
MDIDOCVW Sample: Demonstrates MDI Using Doc/View Architecture
SNAPVW Sample: Uses Property Pages in a Form View Application

Reference

CFrameWnd Class
Hierarchy Chart
CWnd Class
CMDIChildWnd Class

Other Resources

CMDIFrameWnd Members

CMDIFrameWnd Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CFrameWnd Members

Construction

CMDIFrameWnd	Constructs a **CMDIFrameWnd**.

Operations

CreateNewChild	Creates a new child window.
MDIActivate	Activates a different MDI child window.
MDICascade	Arranges all child windows in a cascaded format.
MDIGetActive	Retrieves the currently active MDI child window, along with a flag indicating whether or not the child is maximized.
MDIIconArrange	Arranges all minimized document child windows.
MDIMaximize	Maximizes an MDI child window.
MDINext	Activates the child window immediately behind the currently active child window and places the currently active child window behind all other child windows.
MDIPrev	Activates the previous child window and places the currently active child window immediately behind it.
MDIRestore	Restores an MDI child window from maximized or minimized size.
MDISetMenu	Replaces the menu of an MDI frame window, the Window pop-up menu, or both.
MDITile	Arranges all child windows in a tiled format.

Overridables

CreateClient	Creates a Windows **MDICLIENT** window for this **CMDIFrameWnd**. Called by the **OnCreate** member function of **CWnd**.
GetWindowMenuPopup	Returns the Window pop-up menu.

See Also

Reference

CMDIFrameWnd Class

Hierarchy Chart

CMDIFrameWnd Member Functions

For information about the member functions in **CMDIFrameWnd**, see CMDIFrameWnd Members.

CMDIFrameWnd::CMDIFrameWnd

Constructs a **CMDIFrameWnd** object.

```
CMDIFrameWnd( );
```

Remarks

Call the **Create** or **LoadFrame** member function to create the visible MDI frame window.

Example

```
// Create main MDI Frame window. CMainFrame is a CMDIFrameWnd-derived
// class. The default CFrameWnd::PostNcDestroy() handler will delete this
// object when destroyed.
CMainFrame* pMainFrame = new CMainFrame;
```

See Also

Reference

CMDIFrameWnd Class
Hierarchy Chart
CFrameWnd::Create
CFrameWnd::LoadFrame

Other Resources

CMDIFrameWnd Members

CMDIFrameWnd::CreateClient

Creates the MDI client window that manages the **CMDIChildWnd** objects.

```
virtual BOOL CreateClient(
   LPCREATESTRUCT lpCreateStruct,
   CMenu* pWindowMenu
);
```

Parameters

lpCreateStruct

A long pointer to a CREATESTRUCT structure.

pWindowMenu

A pointer to the Window pop-up menu.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function should be called if you override the **OnCreate** member function directly.

Example

```
// The code below is from winmdi.cpp. It shows how to
// call CMDIFrameWnd::CreateClient(). CMainFrame is a
// CMDIFrameWnd-derived class.
BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext* pContext)
{
   CMenu* pMenu = NULL;
   if (m_hMenuDefault == NULL)
   {
      // default implementation for MFC V1 backward compatibility
      pMenu = GetMenu();
      ASSERT(pMenu != NULL);
      // This is attempting to guess which sub-menu is the Window menu.
      // The Windows user interface guidelines say that the right-most
      // menu on the menu bar should be Help and Window should be one
      // to the left of that.
      int iMenu = pMenu->GetMenuItemCount() - 2;

      // If this assertion fails, your menu bar does not follow the guidelines
      // so you will have to override this function and call CreateClient
      // appropriately or use the MFC V2 MDI functionality.
      ASSERT(iMenu >= 0);
      pMenu = pMenu->GetSubMenu(iMenu);
      ASSERT(pMenu != NULL);
   }

   return CreateClient(lpcs, pMenu);
}
```

See Also

Reference

CMDIFrameWnd Class

Hierarchy Chart

CMDIFrameWnd::CMDIFrameWnd

Other Resources

CMDIFrameWnd Members

CMDIFrameWnd::CreateNewChild

Creates a new child window.

```
CMDIChildWnd* CreateNewChild(
   CRuntimeClass* pClass,
   UINT nResource,
   HMENU hMenu = NULL,
   HACCEL hAccel = NULL
);
```

Parameters

pClass

The run-time class of the child window to be created.

nResource

The ID of shared resources associated with the child window.

hMenu

The child window's menu.

hAccel

The child window's accelerator.

Remarks

Use this function to create child windows of an MDI frame window.

Example

```
void CNDVWApp::OnFileNewDraw()
{
   CMainFrame* pFrame = STATIC_DOWNCAST(CMainFrame, m_pMainWnd);
   pFrame->CreateNewChild(
      RUNTIME_CLASS(CDrawFrame), IDR_DRAWTYPE,
      m_hDrawMenu, m_hDrawAccelerator );
}
```

This example is an excerpt from Knowledge Base article Q201045, "HOWTO: Add Multiple Window Types to a Non-Document/View MDI App." Knowledge Base articles are available in the MSDN Library Visual Studio documentation or at http://support.microsoft.com.

See Also

Reference

CMDIFrameWnd Class

Hierarchy Chart

Other Resources

CMDIFrameWnd Members

CMDIFrameWnd::GetWindowMenuPopup

Call this member function to obtain a handle to the current pop-up menu named "Window" (the pop-up menu with menu items for MDI window management).

```
virtual HMENU GetWindowMenuPopup(
   HMENU hMenuBar
);
```

Parameters

hMenuBar

The current menu bar.

Return Value

The Window pop-up menu if one exists; otherwise **NULL**.

Remarks

The default implementation looks for a pop-up menu containing standard Window menu commands such as **ID_WINDOW_NEW** and **ID_WINDOW_TILE_HORZ**.

Override this member function if you have a Window menu that does not use the standard menu command IDs.

Example

```
// CMainFrame::OnActivateFirstMDIChild() is a menu command handler for
// CMainFrame class, which in turn is a CMDIFrameWnd-derived class.
// It looks for the caption of the first created MDI child window from
// the Window popup menu, and then activate the child window.
void CMainFrame::OnActivateFirstMDIChild()
{
   // Get handle to the Window pop-up menu.
   CMenu* menubar = GetMenu();
   CMenu* wmenu = CMenu::FromHandle(GetWindowMenuPopup(menubar->GetSafeHmenu()));
   if (wmenu == NULL)
      return;

   // Get the caption of the first created MDI child window.
   CString caption;
   if (!wmenu->GetMenuString(AFX_IDM_FIRST_MDICHILD, caption, MF_BYCOMMAND))
      return;

   // Get the actual name of the first created MDI child window by
   // getting rid of the number and space, e.g. "&1 MDI 1".
   int pos = caption.FindOneOf(" ");
   if (pos == -1)
      return;

   caption = caption.Right(caption.GetLength() - (pos + 1));

   // Get the CWnd* of the first created MDI child window by comparing
   // the caption of each MDI child window in the MDI application.
   // Activate the first created MDI child window if found.
   CMDIChildWnd* child = MDIGetActive();
   do
   {
      CString str;
      child->GetWindowText(str);
      if (str == caption)
      {
         child->MDIActivate();        // or MDIActivate(child);
         break;
      }
```

```
        child = (CMDIChildWnd*) child->GetWindow(GW_HWNDNEXT);
    }
    while (child);
}
```

See Also

Reference

CMDIFrameWnd Class

Hierarchy Chart

CMDIFrameWnd::MDIGetActive

Other Resources

CMDIFrameWnd Members

CMDIFrameWnd::MDIActivate

Activates a different MDI child window.

```
void MDIActivate(
   CWnd* pWndActivate
);
```

Parameters

pWndActivate

Points to the MDI child window to be activated.

Remarks

This member function sends the WM_MDIACTIVATE message to both the child window being activated and the child window being deactivated.

This is the same message that is sent if the user changes the focus to an MDI child window by using the mouse or keyboard.

Note

An MDI child window is activated independently of the MDI frame window. When the frame becomes active, the child window that was last activated is sent a WM_NCACTIVATE message to draw an active window frame and caption bar, but it does not receive another **WM_MDIACTIVATE** message.

Example

See the example for CMDIFrameWnd::GetWindowMenuPopup.

See Also

Reference

CMDIFrameWnd Class
Hierarchy Chart
CMDIFrameWnd::MDIGetActive
CMDIFrameWnd::MDINext
CWnd::OnActivate
WM_NCACTIVATE

Other Resources

CMDIFrameWnd Members

CMDIFrameWnd::MDICascade

Arranges all the MDI child windows in a cascade format.

```
void MDICascade( );
void MDICascade(
   int nType
);
```

Parameters

nType

Specifies a cascade flag. Only the following flag can be specified: **MDITILE_SKIPDISABLED**, which prevents disabled MDI child windows from being cascaded.

Remarks

The first version of **MDICascade**, with no parameters, cascades all MDI child windows, including disabled ones. The second version optionally does not cascade disabled MDI child windows if you specify **MDITILE_SKIPDISABLED** for the *nType* parameter.

Example

```
// CMainFrame::OnWindowCommand() is a menu command handler for
// CMainFrame class, which is a CMDIFrameWnd-derived
// class. It handles menu commands for the Windows pop-up menu.
void CMainFrame::OnWindowCommand()
{
   // Get the message the window is currently processing. Since we are
   // in the OnMessage handler, the message we get back must be a
   // WM_COMMAND message, and its low-order word of wParam is the ID of
   // menu item being selected.
   const MSG* msg = GetCurrentMessage();

   switch (LOWORD(msg->wParam))
   {
   case ID_WINDOW_ARRANGE   // For Window\Arrange Icons menu item, arrange
      MDIIconArrange();     // all minimized document child windows.
      break;

   case ID_WINDOW_CASCADE:  // For Window\Cascade menu item, arrange
      MDICascade();         // all the MDI child windows in a cascade format.
      break;

   case ID_WINDOW_TILE_HORZ:       // For Window\Tile Horizontal menu item,
      MDITile(MDITILE_HORIZONTAL); // tile MDI child windows so that
      break;                       // one window appears above another.

   case ID_WINDOW_TILE_VERT:       // For Window\Tile Vertical menu item,
      MDITile(MDITILE_VERTICAL);   // tile MDI child windows so that
      break;                       // one window appears beside another.

   }
}
```

See Also

Reference

CMDIFrameWnd Class
Hierarchy Chart
CMDIFrameWnd::MDIIconArrange
CMDIFrameWnd::MDITile
WM_MDICASCADE

Other Resources

CMDIFrameWnd Members

CMDIFrameWnd::MDIGetActive

Retrieves the current active MDI child window, along with a flag indicating whether the child window is maximized.

```
CMDIChildWnd* MDIGetActive(
   BOOL* pbMaximized = NULL
) const;
```

Parameters

pbMaximized

A pointer to a **BOOL** return value. Set to **TRUE** on return if the window is maximized; otherwise **FALSE**.

Return Value

A pointer to the active MDI child window.

Example

See the example for CMDIChildWnd::MDIMaximize.

See Also

Reference

CMDIFrameWnd Class
Hierarchy Chart
CMDIFrameWnd::MDIActivate
WM_MDIGETACTIVE

Other Resources

CMDIFrameWnd Members

CMDIFrameWnd::MDIIconArrange

Arranges all minimized document child windows.

```
void MDIIconArrange( );
```

Remarks

It does not affect child windows that are not minimized.

Example

See the example for CMDIFrameWnd::MDICascade.

See Also

Reference

CMDIFrameWnd Class
Hierarchy Chart
CMDIFrameWnd::MDICascade
CMDIFrameWnd::MDITile
WM_MDIICONARRANGE

Other Resources

CMDIFrameWnd Members

CMDIFrameWnd::MDIMaximize

Maximizes the specified MDI child window.

```
void MDIMaximize(
   CWnd* pWnd
);
```

Parameters

pWnd

Points to the window to maximize.

Remarks

When a child window is maximized, Windows resizes it to make its client area fill the client window. Windows places the child window's Control menu in the frame's menu bar so the user can restore or close the child window. It also adds the title of the child window to the frame-window title.

If another MDI child window is activated when the currently active MDI child window is maximized, Windows restores the currently active child and maximizes the newly activated child window.

Example

See the example for CMDIChildWnd::MDIMaximize.

See Also

Reference

CMDIFrameWnd Class
Hierarchy Chart
WM_MDIMAXIMIZE
CMDIFrameWnd::MDIRestore

Other Resources

CMDIFrameWnd Members

CMDIFrameWnd::MDINext

Activates the child window immediately behind the currently active child window and places the currently active child window behind all other child windows.

```
void MDINext( );
```

Remarks

If the currently active MDI child window is maximized, the member function restores the currently active child and maximizes the newly activated child.

Example

```
// CMainFrame::OnActivateNextWindow() is a menu command handler for
// CMainFrame class, which in turn is a CMDIFrameWnd-derived class.
// It activates the child window immediately behind the currently
// active child window and places the currently active child window
// behind all other child windows.
void CMainFrame::OnActivateNextWindow()
{
   MDINext();
}
```

See Also

Reference

CMDIFrameWnd Class
Hierarchy Chart
CMDIFrameWnd::MDIPrev
CMDIFrameWnd::MDIActivate
CMDIFrameWnd::MDIGetActive
WM_MDINEXT

Other Resources

CMDIFrameWnd Members

CMDIFrameWnd::MDIPrev

Activates the previous child window and places the currently active child window immediately behind it.

```
void MDIPrev( );
```

Remarks

If the currently active MDI child window is maximized, the member function restores the currently active child and maximizes the newly activated child.

See Also

Reference

CMDIFrameWnd Class
Hierarchy Chart
CMDIFrameWnd::MDINext
CMDIFrameWnd::MDIActivate
CMDIFrameWnd::MDIGetActive
WM_MDINEXT

Other Resources

CMDIFrameWnd Members

CMDIFrameWnd::MDIRestore

Restores an MDI child window from maximized or minimized size.

```
void MDIRestore(
   CWnd* pWnd
);
```

Parameters

pWnd

Points to the window to restore.

Example

See the example for CMDIChildWnd::MDIRestore.

See Also

Reference

CMDIFrameWnd Class
Hierarchy Chart
CMDIFrameWnd::MDIMaximize
WM_MDIRESTORE

Other Resources

CMDIFrameWnd Members

CMDIFrameWnd::MDISetMenu

Replaces the menu of an MDI frame window, the Window pop-up menu, or both.

```
CMenu* MDISetMenu(
   CMenu* pFrameMenu,
   CMenu* pWindowMenu
);
```

Parameters

pFrameMenu

Specifies the menu of the new frame-window menu. If **NULL**, the menu is not changed.

pWindowMenu

Specifies the menu of the new Window pop-up menu. If **NULL**, the menu is not changed.

Return Value

A pointer to the frame-window menu replaced by this message. The pointer may be temporary and should not be stored for later use.

Remarks

After calling **MDISetMenu**, an application must call the DrawMenuBar member function of **CWnd** to update the menu bar.

If this call replaces the Window pop-up menu, MDI child-window menu items are removed from the previous Window menu and added to the new Window pop-up menu.

If an MDI child window is maximized and this call replaces the MDI frame-window menu, the Control menu and restore controls are removed from the previous frame-window menu and added to the new menu.

Do not call this member function if you use the framework to manage your MDI child windows.

Example

```
// CMdiView::OnReplaceMenu() is a menu command handler for CMdiView
// class, which in turn is a CView-derived class. It loads a new
// menu resource and replaces the main application window's menu
// bar with this new menu.
void CMdiView::OnReplaceMenu()
{
   // Load a new menu resource named IDR_SHORT_MENU. m_DefaultMenu is
   // a member variable of CMdiDoc class (a CDocument-derived class).
   // Its type is HMENU.
   CMdiDoc* pdoc = GetDocument();
   pdoc->m_DefaultMenu =
      ::LoadMenu(AfxGetResourceHandle(), MAKEINTRESOURCE(IDR_SHORT_MENU));
   if (pdoc->m_DefaultMenu == NULL)
      return;

   // Get the parent window of this view window. The parent window is
   // a CMDIChildWnd-derived class. We can then obtain the MDI parent
   // frame window using the CMDIChildWnd*. Then, replace the current
   // menu bar with the new loaded menu resource.
   CMDIFrameWnd* frame = ((CMDIChildWnd *) GetParent())->GetMDIFrame();
   frame->MDISetMenu(CMenu::FromHandle(pdoc->m_DefaultMenu), NULL);
   frame->DrawMenuBar();
}

// GetDefaultMenu() is an undocumented virtual function for
// CDocument class. It allows the document to determine which
// menu to display. m_DefaultMenu is of type HMENU. Its value
// is initialized to NULL either in the constructor or
// CDocument::OnNewDocument(). And the menu resource is destroyed
```

```
// in the destructor to avoid having too many menus loaded at once.
HMENU CMdiDoc::GetDefaultMenu()  // get menu depending on state
{
   if (m_DefaultMenu)
      return m_DefaultMenu;

   return COleServerDoc::GetDefaultMenu();
}

// Initialize member variable(s) in the constructor. CMdiDoc is
// a CDocument-derived class.
CMdiDoc::CMdiDoc()
{
   // Use OLE compound files
   EnableCompoundFile();

   m_DefaultMenu = NULL; // initialize to NULL
}

// Destroy menu resource in CMdiDoc's destructor. CMdiDoc is
// a CDocument-derived class.
CMdiDoc::~CMdiDoc()
{
   if (m_DefaultMenu)
      ::DestroyMenu(m_DefaultMenu);
}
```

See Also

Reference

CMDIFrameWnd Class

Hierarchy Chart

CWnd::DrawMenuBar

WM_MDISETMENU

Other Resources

CMDIFrameWnd Members

CMDIFrameWnd::MDITile

Arranges all child windows in a tiled format.

```
void MDITile( );
void MDITile(
   int nType
);
```

Parameters

nType

Specifies a tiling flag. This parameter can be any one of the following flags:

- **MDITILE_HORIZONTAL** Tiles MDI child windows so that one window appears above another.
- **MDITILE_SKIPDISABLED** Prevents disabled MDI child windows from being tiled.
- **MDITILE_VERTICAL** Tiles MDI child windows so that one window appears beside another.

Remarks

The first version of **MDITile**, without parameters, tiles the windows vertically under Windows versions 3.1 and later. The second version tiles windows vertically or horizontally, depending on the value of the *nType* parameter.

Example

See the example for CMDIFrameWnd::MDICascade.

See Also

Reference

CMDIFrameWnd Class
Hierarchy Chart
CMDIFrameWnd::MDICascade
CMDIFrameWnd::MDIIconArrange
WM_MDITILE

Other Resources

CMDIFrameWnd Members

CMemFile Class

The CFile-derived class that supports memory files.

```
class CMemFile : public CFile
```

Remarks

These memory files behave like disk files except that the file is stored in RAM rather than on disk. A memory file is useful for fast temporary storage or for transferring raw bytes or serialized objects between independent processes.

CMemFile objects can automatically allocate their own memory or you can attach your own memory block to the **CMemFile** object by calling Attach. In either case, memory for growing the memory file automatically is allocated in *nGrowBytes*-sized increments if *nGrowBytes* is not zero.

The memory block will automatically be deleted upon destruction of the **CMemFile** object if the memory was originally allocated by the **CMemFile** object; otherwise, you are responsible for deallocating the memory you attached to the object.

You can access the memory block through the pointer supplied when you detach it from the **CMemFile** object by calling Detach.

The most common use of **CMemFile** is to create a **CMemFile** object and use it by calling CFile member functions. Note that creating a **CMemFile** automatically opens it: you do not call CFile::Open, which is only used for disk files. Because **CMemFile** doesn't use a disk file, the data member **CFile::m_hFile** is not used and has no meaning.

The **CFile** member functions Duplicate, LockRange, and UnlockRange are not implemented for **CMemFile**. If you call these functions on a **CMemFile** object, you will get a CNotSupportedException.

CMemFile uses the run-time library functions malloc, realloc, and free to allocate, reallocate, and deallocate memory; and the intrinsic memcpy to block copy memory when reading and writing. If you'd like to change this behavior or the behavior when **CMemFile** grows a file, derive your own class from **CMemFile** and override the appropriate functions.

For more information on **CMemFile**, see the articles Files in MFC and Memory Management (MFC) and see File Handling in the *Run-Time Library Reference*.

Requirements

Header: afx.h

See Also

Reference

CFile Class

Hierarchy Chart

Other Resources

CMemFile Members

CMemFile Members

Base Class Members

CObject Members

CFile Members

Construction

CMemFile	Constructs a memory file object.

Operations

Attach	Attaches a block of memory to **CMemFile**.
Detach	Detaches the block of memory from **CMemFile** and returns a pointer to the block of memory detached.

Advanced Overridables

Alloc	Override to modify memory allocation behavior.
Free	Override to modify memory deallocation behavior.
GrowFile	Override to modify behavior when growing a file.
Memcpy	Override to modify memory copy behavior when reading and writing files.
Realloc	Override to modify memory reallocation behavior.

See Also

Reference

CMemFile Class

Hierarchy Chart

CMemFile Member Functions

For information about the member functions in **CMemFile**, see CMemFile Members.

CMemFile::Alloc

This function is called by **CMemFile** member functions.

```
virtual BYTE* Alloc(
   SIZE_T nBytes
);
```

Parameters

nBytes

Number of bytes of memory to be allocated.

Return Value

A pointer to the memory block that was allocated, or **NULL** if the allocation failed.

Remarks

Override this function to implement custom memory allocation. If you override this function, you'll probably want to override Free and Realloc as well.

The default implementation uses the run-time library function malloc to allocate memory.

See Also

Reference

CMemFile Class
Hierarchy Chart
CMemFile::Free
CMemFile::Realloc
malloc

Other Resources

CMemFile Members

CMemFile::Attach

Call this function to attach a block of memory to **CMemFile**.

```
void Attach(
   BYTE* lpBuffer,
   UINT nBufferSize,
   UINT nGrowBytes = 0
);
```

Parameters

lpBuffer

Pointer to the buffer to be attached to **CMemFile**.

nBufferSize

An integer that specifies the size of the buffer in bytes.

nGrowBytes

The memory allocation increment in bytes.

Remarks

This causes **CMemFile** to use the block of memory as the memory file.

If *nGrowBytes* is 0, **CMemFile** will set the file length to *nBufferSize*. This means that the data in the memory block before it was attached to **CMemFile** will be used as the file. Memory files created in this manner cannot be grown.

Since the file cannot be grown, be careful not to cause **CMemFile** to attempt to grow the file. For example, don't call the **CMemFile** overrides of CFile:Write to write past the end or don't call CFile:SetLength with a length longer than *nBufferSize*.

If *nGrowBytes* is greater than 0, **CMemFile** will ignore the contents of the memory block you've attached. You'll have to write the contents of the memory file from scratch using the **CMemFile** override of **CFile::Write**. If you attempt to write past the end of the file or grow the file by calling the **CMemFile** override of **CFile::SetLength**, **CMemFile** will grow the memory allocation in increments of *nGrowBytes*. Growing the memory allocation will fail if the memory block you pass to **Attach** wasn't allocated with a method compatible with Alloc. To be compatible with the default implementation of **Alloc**, you must allocate the memory with the run-time library function malloc or calloc.

See Also

Reference

CMemFile Class
Hierarchy Chart
CMemFile::CMemFile
CMemFile::Detach
CMemFile::Alloc
CFile::Write
CFile::SetLength

Other Resources

CMemFile Members

CMemFile::CMemFile

The first overload opens an empty memory file.

```
CMemFile(
   UINT nGrowBytes = 1024
);
CMemFile(
   BYTE* lpBuffer,
   UINT nBufferSize,
   UINT nGrowBytes = 0
);
```

Parameters

nGrowBytes

The memory allocation increment in bytes.

lpBuffer

Pointer to a buffer that receives information of the size *nBufferSize*.

nBufferSize

An integer that specifies the size of the file buffer, in bytes.

Remarks

Note that the file is opened by the constructor and that you should not call CFile::Open.

The second overload acts the same as if you used the first constructor and immediately called Attach with the same parameters. See **Attach** for details.

Example

```
// example for CMemFile::CMemFile
CMemFile f; // Ready to use - no Open necessary.

BYTE * pBuf = (BYTE *)new char [1024];
CMemFile g( pBuf, 1024, 256 );
// same as CMemFile g; g.Attach( pBuf, 1024, 256 );
```

See Also

Reference

CMemFile Class
Hierarchy Chart
CMemFile::Attach

Other Resources

CMemFile Members

CMemFile::Detach

Call this function to get a pointer to the memory block being used by **CMemFile**.

```
BYTE * Detach( );
```

Return Value

A pointer to the memory block that contains the contents of the memory file.

Remarks

Calling this function also closes the **CMemFile**. You can reattach the memory block to **CMemFile** by calling Attach. If you want to reattach the file and use the data in it, you should call CFile::GetLength to get the length of the file before calling **Detach**. Note that if you attach a memory block to **CMemFile** so that you can use its data (*nGrowBytes* == 0), then you won't be able to grow the memory file.

See Also

Reference

CMemFile Class
Hierarchy Chart
CMemFile::Attach
CFile::GetLength

Other Resources

CMemFile Members

CMemFile::Free

This function is called by **CMemFile** member functions.

```
virtual void Free(
   BYTE * lpMem
);
```

Parameters

lpMem

Pointer to the memory to be deallocated.

Remarks

Override this function to implement custom memory deallocation. If you override this function, you'll probably want to override Alloc and Realloc as well.

See Also

Reference

CMemFile Class
Hierarchy Chart
CMemFile::Alloc
CMemFile::Realloc

Other Resources

CMemFile Members

CMemFile::GrowFile

This function is called by several of the **CMemFile** member functions.

```
virtual void GrowFile(
   SIZE_T dwNewLen
);
```

Parameters

dwNewLen

New size of the memory file.

Remarks

You can override it if you want to change how **CMemFile** grows its file. The default implementation calls Realloc to grow an existing block (or Alloc to create a memory block), allocating memory in multiples of the *nGrowBytes* value specified in the constructor or Attach call.

See Also

Reference

CMemFile Class
Hierarchy Chart
CMemFile::Alloc
CMemFile::Realloc
CMemFile::CMemFile
CMemFile::Attach

Other Resources

CMemFile Members

CMemFile::Memcpy

This function is called by the **CMemFile** overrides of CFile::Read and CFile::Write to transfer data to and from the memory file.

```
virtual BYTE* Memcpy(
   BYTE* lpMemTarget,
   const BYTE* lpMemSource,
   SIZE_T nBytes);
```

Parameters

lpMemTarget

Pointer to the memory block into which the source memory will be copied.

lpMemSource

Pointer to the source memory block.

nBytes

Number of bytes to be copied.

Return Value

A copy of *lpMemTarget*.

Remarks

Override this function if you want to change the way that **CMemFile** does these memory copies.

See Also

Reference

CMemFile Class
Hierarchy Chart
CFile::Read
CFile::Write

Other Resources

CMemFile Members

CMemFile::Realloc

This function is called by **CMemFile** member functions.

```
virtual BYTE* Realloc(
   BYTE* lpMem,
   SIZE_T nBytes
);
```

Parameters

lpMem

A pointer to the memory block to be reallocated.

nBytes

New size for the memory block.

Return Value

A pointer to the memory block that was reallocated (and possibly moved), or **NULL** if the reallocation failed.

Remarks

Override this function to implement custom memory reallocation. If you override this function, you'll probably want to override Alloc and Free as well.

See Also

Reference

CMemFile Class
Hierarchy Chart
CMemFile::Alloc
CMemFile::Free

Other Resources

CMemFile Members

CMemoryException Class

Represents an out-of-memory exception condition.

```
class CMemoryException : public CSimpleException
```

Remarks

No further qualification is necessary or possible. Memory exceptions are thrown automatically by **new**. If you write your own memory functions, using **malloc**, for example, then you are responsible for throwing memory exceptions.

For more information on **CMemoryException**, see the article Exception Handling (MFC).

Requirements

Header: afx.h

See Also

Reference

CException Class

Hierarchy Chart

Other Resources

CMemoryException Members

CMemoryException Members

Base Class Members

CObject Members

CException Members

Construction

CMemoryException	Constructs a **CMemoryException** object.

See Also

Reference

CMemoryException Class

Hierarchy Chart

CMemoryException Member Functions

For information about the member functions in **CMemoryException**, see CMemoryException Members.

CMemoryException::CMemoryException

Constructs a **CMemoryException** object.

```
CMemoryException( );
```

Remarks

Do not use this constructor directly, but rather call the global function AfxThrowMemoryException. This global function can succeed in an out-of-memory situation because it constructs the exception object in previously allocated memory. For more information about exception processing, see the article Exceptions.

See Also

Reference

CMemoryException Class

Hierarchy Chart

Other Resources

CMemoryException Members

Exception Processing

CMemoryState Structure

Provides a convenient way to detect memory leaks in your program.

```
struct CMemoryState
```

Remarks

CMemoryState is a structure and does not have a base class.

A "memory leak" occurs when memory for an object is allocated on the heap but not deallocated when it is no longer required. Such memory leaks can eventually lead to out-of-memory errors. There are several ways to allocate and deallocate memory in your program:

- Using the **malloc**/**free** family of functions from the run-time library.
- Using the Windows API memory management functions, **LocalAlloc**/**LocalFree** and **GlobalAlloc**/**GlobalFree**.
- Using the C++ **new** and **delete** operators.

The **CMemoryState** diagnostics only help detect memory leaks caused when memory allocated using the **new** operator is not deallocated using **delete**. The other two groups of memory-management functions are for non-C++ programs, and mixing them with **new** and **delete** in the same program is not recommended. An additional macro, **DEBUG_NEW**, is provided to replace the **new** operator when you need file and line-number tracking of memory allocations. **DEBUG_NEW** is used whenever you would normally use the **new** operator.

As with other diagnostics, the **CMemoryState** diagnostics are only available in debug versions of your program. A debug version must have the **_DEBUG** constant defined.

If you suspect your program has a memory leak, you can use the **Checkpoint**, **Difference**, and **DumpStatistics** functions to discover the difference between the memory state (objects allocated) at two different points in program execution. This information can be useful in determining whether a function is cleaning up all the objects it allocates.

If simply knowing where the imbalance in allocation and deallocation occurs does not provide enough information, you can use the **DumpAllObjectsSince** function to dump all objects allocated since the previous call to **Checkpoint**. This dump shows the order of allocation, the source file and line where the object was allocated (if you are using **DEBUG_NEW** for allocation), and the derivation of the object, its address, and its size. **DumpAllObjectsSince** also calls each object's **Dump** function to provide information about its current state.

For more information about how to use **CMemoryState** and other diagnostics, see Debugging MFC Applications.

Note

Declarations of objects of type **CMemoryState** and calls to member functions should be bracketed by `#if defined(_DEBUG)` /`#endif` directives. This causes memory diagnostics to be included only in debugging builds of your program.

Requirements

Header: afx.h

See Also

Reference

Hierarchy Chart

Other Resources

CMemoryState Members

CMemoryState Members

Construction

Checkpoint	Obtains a snapshot or "checkpoint" of the current memory state.
CMemoryState	Constructs a class-like structure that controls memory checkpoints.

Operations

Difference	Computes the difference between two objects of type **CMemoryState**.
DumpAllObjectsSince	Dumps a summary of all currently allocated objects since a previous checkpoint.
DumpStatistics	Prints memory allocation statistics for a **CMemoryState** object.

See Also

Reference

CMemoryState Structure

Hierarchy Chart

CMemoryState Member Functions

For information about the member functions in **CMemoryState**, see CMemoryState Members.

CMemoryState::Checkpoint

Takes a snapshot summary of memory and stores it in this **CMemoryState** object.

```
void Checkpoint( );
```

Remarks

The **CMemoryState** member functions Difference and DumpAllObjectsSince use this snapshot data.

Example

See the example for the CMemoryState constructor.

See Also

Reference

CMemoryState Structure

Hierarchy Chart

Other Resources

CMemoryState Members

CMemoryState::CMemoryState

Constructs an empty **CMemoryState** object that must be filled in by the Checkpoint or Difference member function.

```
CMemoryState( );
```

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
// example for CMemoryState::CMemoryState
// Includes all CMemoryState functions
CMemoryState msOld;
msOld.Checkpoint();
CAge* page1 = new CAge( 21 );
CAge* page2 = new CAge( 22 );
msOld.DumpAllObjectsSince();
```

The results from this program are as follows:

```
Dumping objects ->
{2} a CObject at $190A
{1} a CObject at $18EA
Object dump complete.
```

See Also

Reference

CMemoryState Structure

Hierarchy Chart

Other Resources

CMemoryState Members

CMemoryState::Difference

Compares two **CMemoryState** objects, then stores the difference into this **CMemoryState** object.

```
BOOL Difference(
  const CMemoryState& oldState,
  const CMemoryState& newState
);
```

Parameters

oldState

The initial memory state as defined by a **CMemoryState** checkpoint.

newState

The new memory state as defined by a **CMemoryState** checkpoint.

Return Value

Nonzero if the two memory states are different; otherwise 0.

Remarks

Checkpoint must have been called for each of the two memory-state parameters.

Example

See the example for the CMemoryState constructor.

See Also

Reference

CMemoryState Structure

Hierarchy Chart

Other Resources

CMemoryState Members

CMemoryState::DumpAllObjectsSince

Calls the **Dump** function for all objects of a type derived from class **CObject** that were allocated (and are still allocated) since the last Checkpoint call for this **CMemoryState** object.

```
void DumpAllObjectsSince( ) const;
```

Remarks

Calling **DumpAllObjectsSince** with an uninitialized **CMemoryState** object will dump out all objects currently in memory.

Example

See the example for the CMemoryState constructor.

See Also

Reference

CMemoryState Structure

Hierarchy Chart

Other Resources

CMemoryState Members

CMemoryState::DumpStatistics

Prints a concise memory statistics report from a **CMemoryState** object that is filled by the Difference member function.

```
void DumpStatistics( ) const;
```

Remarks

The report, which is printed on the afxDump device, shows the following:

A sample report gives information on the number (or amount) of:

- free blocks
- normal blocks
- CRT blocks
- ignore blocks
- client blocks
- maximum memory used by the program at any one time (in bytes)
- total memory currently used by the program (in bytes)

Free blocks are the number of blocks whose deallocation was delayed if **afxMemDF** was set to **delayFreeMemDF**. For more information, see afxMemDF, in the "MFC Macros and Globals" section. See Types of Blocks on the Debug Heap for more information on these block types.

Example

The following code should be placed in *projname*App.cpp. Define the following global variables:

```
static CMemoryState oldstate, newstate, diffstate;
```

In the **InitInstance** function, add the line:

```
oldstate.Checkpoint();
```

Add a handler for the **ExitInstance** function and use the following code:

```
    newstate.Checkpoint();
    if (diffstate.Difference(oldstate, newstate))
       {
       TRACE("Memory leaked\n");
       diffstate.DumpStatistics();
       }
    return 0;
```

You can now run the program in Debug mode to see the output of the **DumpStatistics** function.

Output

```
Memory leaked
0 bytes in 0 Free Blocks.
-626 bytes in -13 Normal Blocks.
0 bytes in 0 CRT Blocks.
0 bytes in 0 Ignore Blocks.
-1212 bytes in -6 Client Blocks.
Largest number used: 178 bytes.
Total allocations: 988 bytes.
```

See Also

Reference

CMemoryState Structure

Hierarchy Chart

Other Resources

CMemoryState Members

CMenu Class

An encapsulation of the Windows **HMENU**.

```
class CMenu : public CObject
```

Remarks

It provides member functions for creating, tracking, updating, and destroying a menu.

Create a **CMenu** object on the stack frame as a local, then call **CMenu**'s member functions to manipulate the new menu as needed. Next, call CWnd::SetMenu to set the menu to a window, followed immediately by a call to the **CMenu** object's Detach member function. The **CWnd::SetMenu** member function sets the window's menu to the new menu, causes the window to be redrawn to reflect the menu change, and also passes ownership of the menu to the window. The call to **Detach** detaches the **HMENU** from the **CMenu** object, so that when the local **CMenu** variable passes out of scope, the **CMenu** object destructor does not attempt to destroy a menu it no longer owns. The menu itself is automatically destroyed when the window is destroyed.

You can use the LoadMenuIndirect member function to create a menu from a template in memory, but a menu created from a resource by a call to LoadMenu is more easily maintained, and the menu resource itself can be created and modified by the menu editor.

Requirements

Header: afxwin.h

See Also

Tasks

CTRLTEST Sample: Implements Custom Controls

DYNAMENU Sample: Dynamically Updates Menus

Reference

CObject Class

Hierarchy Chart

CObject Class

Other Resources

CMenu Members

CMenu Members

Base Class Members

Data Members

Construction/Destruction

Initialization

Menu Operations

Menu Item Operations

Overridables

Base Class Members

CObject Members

Data Members

m_hMenu	Specifies the handle to the Windows menu attached to the **CMenu** object.

Construction

CMenu	Constructs a **CMenu** object.

Initialization

Attach	Attaches a Windows menu handle to a **CMenu** object.
CreateMenu	Creates an empty menu and attaches it to a **CMenu** object.
CreatePopupMenu	Creates an empty pop-up menu and attaches it to a **CMenu** object.
DeleteTempMap	Deletes any temporary **CMenu** objects created by the **FromHandle** member function.
DestroyMenu	Destroys the menu attached to a **CMenu** object and frees any memory that the menu occupied.
Detach	Detaches a Windows menu handle from a **CMenu** object and returns the handle.
FromHandle	Returns a pointer to a **CMenu** object given a Windows menu handle.
GetSafeHmenu	Returns the **m_hMenu** wrapped by this **CMenu** object.
LoadMenu	Loads a menu resource from the executable file and attaches it to a **CMenu** object.
LoadMenuIndirect	Loads a menu from a menu template in memory and attaches it to a **CMenu** object.

Menu Operations

DeleteMenu	Deletes a specified item from the menu. If the menu item has an associated pop-up menu, destroys the ha ndle to the pop-up menu and frees the memory used by it.
TrackPopupMenu	Displays a floating pop-up menu at the specified location and tracks the selection of items on the pop-up menu.
TrackPopupMenuEx	Displays a floating pop-up menu at the specified location and tracks the selection of items on the pop-up menu.

Menu Item Operations

AppendMenu	Appends a new item to the end of this menu.
CheckMenuItem	Places a check mark next to or removes a check mark from a menu item in the pop-up menu.
CheckMenuRadioItem	Places a radio button next to a menu item and removes the radio button from all of the other menu ite ms in the group.
EnableMenuItem	Enables, disables, or dims (grays) a menu item.
GetDefaultItem	Determines the default menu item on the specified menu.
GetMenuContextHelpId	Retrieves the help context ID associated with the menu.
GetMenuInfo	Retrieves information on a specific menu.
SetMenuInfo	Sets information on a specific menu.
GetMenuItemCount	Determines the number of items in a pop-up or top-level menu.
GetMenuItemID	Obtains the menu-item identifier for a menu item located at the specified position.
GetMenuItemInfo	Retrieves information about a menu item.
GetMenuState	Returns the status of the specified menu item or the number of items in a pop-up menu.
GetMenuString	Retrieves the label of the specified menu item.
GetSubMenu	Retrieves a pointer to a pop-up menu.
InsertMenu	Inserts a new menu item at the specified position, moving other items down the menu.
InsertMenuItem	Inserts a new menu item at the specified position in a menu.
ModifyMenu	Changes an existing menu item at the specified position.
RemoveMenu	Deletes a menu item with an associated pop-up menu from the specified menu.
SetDefaultItem	Sets the default menu item for the specified menu.
SetMenuContextHelpId	Sets the help context ID to be associated with the menu.
SetMenuItemBitmaps	Associates the specified check-mark bitmaps with a menu item.
SetMenuItemInfo	Changes information about a menu item.

Overridables

DrawItem	Called by the framework when a visual aspect of an owner-drawn menu changes.
MeasureItem	Called by the framework to determine menu dimensions when an owner-drawn menu is created.

Operators

operator !=	Determines if two menu objects are not equal.

operator ==	Determines if two menu objects are not equal.
operator HMENU	Retrieves the handle of the menu object.

See Also

Reference

CMenu Class

Hierarchy Chart

CMenu Member Functions

For information about the member functions in **CMenu**, see CMenu Members.

CMenu::AppendMenu

Appends a new item to the end of a menu.

```
BOOL AppendMenu(
   UINT nFlags,
   UINT_PTR nIDNewItem = 0,
   LPCTSTR lpszNewItem = NULL
);
BOOL AppendMenu(
   UINT nFlags,
   UINT_PTR nIDNewItem,
   const CBitmap* pBmp
);
```

Parameters

nFlags

Specifies information about the state of the new menu item when it is added to the menu. It consists of one or more of the values listed in the Remarks section.

nIDNewItem

Specifies either the command ID of the new menu item or, if *nFlags* is set to **MF_POPUP**, the menu handle (**HMENU**) of a pop-up menu. The *nIDNewItem* parameter is ignored (not needed) if *nFlags* is set to **MF_SEPARATOR**.

lpszNewItem

Specifies the content of the new menu item. The *nFlags* parameter is used to interpret *lpszNewItem* in the following way:

nFlags	Interpretation of lpszNewItem
MF_OWNERDRAW	Contains an application-supplied 32-bit value that the application can use to maintain additional data associated with the menu item. This 32-bit value is available to the application when it processes **WM_MEASUREITEM** and **WM_DRAWITEM** messages. The value is stored in the **itemData** member of the structure supplied with those messages.
MF_STRING	Contains a pointer to a null-terminated string. This is the default interpretation.
MF_SEPARATOR	The *lpszNewItem* parameter is ignored (not needed).

pBmp

Points to a **CBitmap** object that will be used as the menu item.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The application can specify the state of the menu item by setting values in *nFlags*. When *nIDNewItem* specifies a pop-up menu, it becomes part of the menu to which it is appended. If that menu is destroyed, the appended menu will also be destroyed. An appended menu should be detached from a **CMenu** object to avoid conflict. Note that **MF_STRING** and **MF_OWNERDRAW** are not valid for the bitmap version of **AppendMenu**.

The following list describes the flags that may be set in *nFlags*:

- **MF_CHECKED** Acts as a toggle with **MF_UNCHECKED** to place the default check mark next to the item. When the

application supplies check-mark bitmaps (see the SetMenuItemBitmaps member function), the "check mark on" bitmap is displayed.

- **MF_UNCHECKED** Acts as a toggle with **MF_CHECKED** to remove a check mark next to the item. When the application supplies check-mark bitmaps (see the **SetMenuItemBitmaps** member function), the "check mark off" bitmap is displayed.
- **MF_DISABLED** Disables the menu item so that it cannot be selected but does not dim it.
- **MF_ENABLED** Enables the menu item so that it can be selected and restores it from its dimmed state.
- **MF_GRAYED** Disables the menu item so that it cannot be selected and dims it.
- **MF_MENUBARBREAK** Places the item on a new line in static menus or in a new column in pop-up menus. The new pop-up menu column will be separated from the old column by a vertical dividing line.
- **MF_MENUBREAK** Places the item on a new line in static menus or in a new column in pop-up menus. No dividing line is placed between the columns.
- **MF_OWNERDRAW** Specifies that the item is an owner-draw item. When the menu is displayed for the first time, the window that owns the menu receives a **WM_MEASUREITEM** message, which retrieves the height and width of the menu item. The **WM_DRAWITEM** message is the one sent whenever the owner must update the visual appearance of the menu item. This option is not valid for a top-level menu item.
- **MF_POPUP** Specifies that the menu item has a pop-up menu associated with it. The ID parameter specifies a handle to a pop-up menu that is to be associated with the item. This is used for adding either a top-level pop-up menu or a hierarchical pop-up menu to a pop-up menu item.
- **MF_SEPARATOR** Draws a horizontal dividing line. Can only be used in a pop-up menu. This line cannot be dimmed, disabled, or highlighted. Other parameters are ignored.
- **MF_STRING** Specifies that the menu item is a character string.

Each of the following groups lists flags that are mutually exclusive and cannot be used together:

- **MF_DISABLED**, **MF_ENABLED**, and **MF_GRAYED**
- **MF_STRING**, **MF_OWNERDRAW**, **MF_SEPARATOR**, and the bitmap version
- **MF_MENUBARBREAK** and **MF_MENUBREAK**
- **MF_CHECKED** and **MF_UNCHECKED**

Whenever a menu that resides in a window is changed (whether or not the window is displayed), the application should call CWnd::DrawMenuBar.

Example

See the example for CMenu::CreateMenu.

See Also

Reference

CMenu Class
Hierarchy Chart
CWnd::DrawMenuBar
CMenu::InsertMenu
CMenu::RemoveMenu
CMenu::SetMenuItemBitmaps
CMenu::Detach
AppendMenu

Other Resources

CMenu Members

CMenu::Attach

Attaches an existing Windows menu to a **CMenu** object.

```
BOOL Attach(
   HMENU hMenu
);
```

Parameters

hMenu

Specifies a handle to a Windows menu.

Return Value

Nonzero if the operation was successful; otherwise 0.

Remarks

This function should not be called if a menu is already attached to the **CMenu** object. The menu handle is stored in the **m_hMenu** data member.

If the menu you want to manipulate is already associated with a window, you can use the CWnd::GetMenu function to get a handle to the menu.

Example

```
CMenu mnu;
HMENU hmnu = AfxGetMainWnd()->GetMenu()->GetSafeHmenu();
mnu.Attach( hmnu );
// Now you can manipulate the window's menu as a CMenu
// object...
```

See Also

Reference

CMenu Class
Hierarchy Chart
CMenu::Detach
CMenu::CMenu
CWnd::GetMenu

Other Resources

CMenu Members

CMenu::CheckMenuItem

Adds check marks to or removes check marks from menu items in the pop-up menu.

```
UINT CheckMenuItem(
   UINT nIDCheckItem,
   UINT nCheck
);
```

Parameters

nIDCheckItem

Specifies the menu item to be checked, as determined by *nCheck*.

nCheck

Specifies how to check the menu item and how to determine the item's position in the menu. The *nCheck* parameter can be a combination of **MF_CHECKED** or **MF_UNCHECKED** with **MF_BYPOSITION** or **MF_BYCOMMAND** flags. These flags can be combined by using the bitwise OR operator. They have the following meanings:

- **MF_BYCOMMAND** Specifies that the parameter gives the command ID of the existing menu item. This is the default.
- **MF_BYPOSITION** Specifies that the parameter gives the position of the existing menu item. The first item is at position 0.
- **MF_CHECKED** Acts as a toggle with **MF_UNCHECKED** to place the default check mark next to the item.
- **MF_UNCHECKED** Acts as a toggle with **MF_CHECKED** to remove a check mark next to the item.

Return Value

The previous state of the item: **MF_CHECKED** or **MF_UNCHECKED**, or 0xFFFFFFFF if the menu item did not exist.

Remarks

The *nIDCheckItem* parameter specifies the item to be modified.

The *nIDCheckItem* parameter may identify a pop-up menu item as well as a menu item. No special steps are required to check a pop-up menu item. Top-level menu items cannot be checked. A pop-up menu item must be checked by position since it does not have a menu-item identifier associated with it.

Example

See the example for CMenu::GetMenuState.

See Also

Reference

CMenu Class
Hierarchy Chart
CMenu::GetMenuState
CheckMenuItem
CMenu::CheckMenuRadioItem

Other Resources

CMenu Members

CMenu::CheckMenuRadioItem

Checks a specified menu item and makes it a radio item.

```
BOOL CheckMenuRadioItem(
   UINT nIDFirst,
   UINT nIDLast,
   UINT nIDItem,
   UINT nFlags
);
```

Parameters

nIDFirst

Specifies (as an ID or offset, depending on the value of *nFlags*) the first menu item in the radio button group.

nIDLast

Specifies (as an ID or offset, depending on the value of *nFlags*) the last menu item in the radio button group.

nIDItem

Specifies (as an ID or offset, depending on the value of *nFlags*) the item in the group which will be checked with a radio button.

nFlags

Specifies interpretation of *nIDFirst*, *nIDLast*, and *nIDItem* in the following way:

nFlags	Interpretation
MF_BYCOMMAND	Specifies that the parameter gives the command ID of the existing menu item. This is the default if neither **MF_BYCOMMAND** nor **MF_BYPOSITION** is set.
MF_BYPOSITION	Specifies that the parameter gives the position of the existing menu item. The first item is at position 0.

Return Value

Nonzero if successful; otherwise 0

Remarks

At the same time, the function unchecks all other menu items in the associated group and clears the radio-item type flag for those items. The checked item is displayed using a radio button (or bullet) bitmap instead of a check mark bitmap.

Example

See the example for ON_COMMAND_RANGE.

See Also

Reference

CMenu Class
Hierarchy Chart
CMenu::CheckMenuItem
CMenu::GetMenuState
CheckMenuRadioItem

Other Resources

CMenu Members

CMenu::CMenu

Creates an empty menu and attaches it to a **CMenu** object.

```
CMenu( );
```

Remarks

The menu is not created until you call one of the create or load member functions of **CMenu:**

- CreateMenu
- CreatePopupMenu
- LoadMenu
- LoadMenuIndirect
- Attach

See Also

Reference

CMenu Class
Hierarchy Chart
CMenu::CreateMenu
CMenu::CreatePopupMenu
CMenu::LoadMenu
CMenu::LoadMenuIndirect
CMenu::Attach

Other Resources

CMenu Members

CMenu::CreateMenu

Creates a menu and attaches it to the **CMenu** object.

```
BOOL CreateMenu( );
```

Return Value

Nonzero if the menu was created successfully; otherwise 0.

Remarks

The menu is initially empty. Menu items can be added by using the **AppendMenu** or **InsertMenu** member function.

If the menu is assigned to a window, it is automatically destroyed when the window is destroyed.

Before exiting, an application must free system resources associated with a menu if the menu is not assigned to a window. An application frees a menu by calling the DestroyMenu member function.

Example

```
// The code fragment below shows how to create a new menu for the
// application window using CreateMenu() and CreatePopupMenu().
// Then, the created menu will replace the current menu of the
// application. The old menu will be destroyed with DestroyMenu().
// NOTE: The code fragment below is done in a CFrameWnd-derived class.

// Create a new menu for the application window.
VERIFY(m_NewMenu.CreateMenu());

// Create a "File" popup menu and insert this popup menu to the
// new menu of the application window. The "File" menu has only
// one menu item, i.e. "Exit".
VERIFY(m_FileMenu.CreatePopupMenu());
m_FileMenu.AppendMenu(MF_STRING, ID_APP_EXIT, (LPCTSTR)"E&xit");
m_NewMenu.AppendMenu(MF_POPUP, (UINT) m_FileMenu.m_hMenu, "&File");

// Remove and destroy old menu
SetMenu(NULL);
CMenu* old_menu = CMenu::FromHandle(m_hMenuDefault);
old_menu->DestroyMenu();

// Add new menu.
SetMenu(&m_NewMenu);

// Assign default menu
m_hMenuDefault = m_NewMenu.m_hMenu;
```

See Also

Reference

CMenu Class
Hierarchy Chart
CMenu::CMenu
CMenu::DestroyMenu
CMenu::InsertMenu
CWnd::SetMenu
CreateMenu
CMenu::AppendMenu

Other Resources

CMenu Members

CMenu::CreatePopupMenu

Creates a pop-up menu and attaches it to the **CMenu** object.

```
BOOL CreatePopupMenu( );
```

Return Value

Nonzero if the pop-up menu was successfully created; otherwise 0.

Remarks

The menu is initially empty. Menu items can be added by using the **AppendMenu** or **InsertMenu** member function. The application can add the pop-up menu to an existing menu or pop-up menu. The **TrackPopupMenu** member function may be used to display this menu as a floating pop-up menu and to track selections on the pop-up menu.

If the menu is assigned to a window, it is automatically destroyed when the window is destroyed. If the menu is added to an existing menu, it is automatically destroyed when that menu is destroyed.

Before exiting, an application must free system resources associated with a pop-up menu if the menu is not assigned to a window. An application frees a menu by calling the DestroyMenu member function.

Example

See the example for CMenu::CreateMenu.

See Also

Reference

CMenu Class
Hierarchy Chart
CMenu::CreateMenu
CMenu::InsertMenu
CWnd::SetMenu
CMenu::TrackPopupMenu
CreatePopupMenu
CMenu::AppendMenu

Other Resources

CMenu Members

CMenu::DeleteMenu

Deletes an item from the menu.

```
BOOL DeleteMenu(
   UINT nPosition,
   UINT nFlags
);
```

Parameters

nPosition

Specifies the menu item that is to be deleted, as determined by *nFlags*.

nFlags

Is used to interpret *nPosition* in the following way:

nFlags	Interpretation of nPosition
MF_BYCOMMAND	Specifies that the parameter gives the command ID of the existing menu item. This is the default if neither **MF_BYCOMMAND** nor **MF_BYPOSITION** is set.
MF_BYPOSITION	Specifies that the parameter gives the position of the existing menu item. The first item is at position 0.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

If the menu item has an associated pop-up menu, **DeleteMenu** destroys the handle to the pop-up menu and frees the memory used by the pop-up menu.

Whenever a menu that resides in a window is changed (whether or not the window is displayed), the application must call CWnd::DrawMenuBar.

Example

See the example for CWnd::GetMenu.

See Also

Reference

CMenu Class
Hierarchy Chart
CWnd::DrawMenuBar
DeleteMenu

Other Resources

CMenu Members

CMenu::DeleteTempMap

Called automatically by the **CWinApp** idle-time handler, deletes any temporary **CMenu** objects created by the FromHandle member function.

```
static void PASCAL DeleteTempMap( );
```

Remarks

DeleteTempMap detaches the Windows menu object attached to a temporary **CMenu** object before deleting the **CMenu** object.

Example

```
// The following example is correct:
CMenu::DeleteTempMap();

// The following example is incorrect.
// DeleteTempMap() is a static member and cannot be called
// within the scope of an instantiated CMenu object.
pMenu->DeleteTempMap();  // Causes compiler error
```

See Also

Reference

CMenu Class

Hierarchy Chart

Other Resources

CMenu Members

CMenu::DestroyMenu

Destroys the menu and any Windows resources that were used.

```
BOOL DestroyMenu( );
```

Return Value

Nonzero if the menu is destroyed; otherwise 0.

Remarks

The menu is detached from the **CMenu** object before it is destroyed. The Windows **DestroyMenu** function is automatically called in the **CMenu** destructor.

Example

See the example for CMenu::CreateMenu.

See Also

Reference

CMenu Class
Hierarchy Chart
DestroyMenu

Other Resources

CMenu Members

CMenu::Detach

Detaches a Windows menu from a **CMenu** object and returns the handle.

```
HMENU Detach( );
```

Return Value

The handle, of type **HMENU**, to a Windows menu, if successful; otherwise **NULL**.

Remarks

The **m_hMenu** data member is set to **NULL**.

Example

```
CMenu mnu;
HMENU hmnu = AfxGetMainWnd()->GetMenu()->GetSafeHmenu();
mnu.Attach( hmnu );

// Now you can manipulate the window's menu as a CMenu
// object...

hmnu = mnu.Detach();
```

See Also

Reference

CMenu Class
Hierarchy Chart
CMenu::Attach

Other Resources

CMenu Members

CMenu::DrawItem

Called by the framework when a visual aspect of an owner-drawn menu changes.

```
virtual void DrawItem(
   LPDRAWITEMSTRUCT lpDrawItemStruct
);
```

Parameters

lpDrawItemStruct

A pointer to a DRAWITEMSTRUCT structure that contains information about the type of drawing required.

Remarks

The *itemAction* member of the **DRAWITEMSTRUCT** structure defines the drawing action that is to be performed. Override this member function to implement drawing for an owner-draw **CMenu** object. The application should restore all graphics device interface (GDI) objects selected for the display context supplied in *lpDrawItemStruct* before the termination of this member function.

See CWnd::OnDrawItem for a description of the **DRAWITEMSTRUCT** structure.

Example

The following code is from the MFC CTRLTEST sample:

```
// Override DrawItem() to implement drawing for an owner-draw CMenu object.
// CColorMenu is a CMenu-derived class.
void CColorMenu::DrawItem(LPDRAWITEMSTRUCT lpDIS)
{
   CDC* pDC = CDC::FromHandle(lpDIS->hDC);
   COLORREF cr = (COLORREF)lpDIS->itemData; // RGB in item data

   if (lpDIS->itemAction & ODA_DRAWENTIRE)
   {
      // Paint the color item in the color requested
      CBrush br(cr);
      pDC->FillRect(&lpDIS->rcItem, &br);
   }

   if ((lpDIS->itemState & ODS_SELECTED) &&
      (lpDIS->itemAction & (ODA_SELECT | ODA_DRAWENTIRE)))
   {
      // item has been selected - hilite frame
      COLORREF crHilite = RGB(255-GetRValue(cr),
                  255-GetGValue(cr), 255-GetBValue(cr));
      CBrush br(crHilite);
      pDC->FrameRect(&lpDIS->rcItem, &br);
   }

   if (!(lpDIS->itemState & ODS_SELECTED) &&
      (lpDIS->itemAction & ODA_SELECT))
   {
      // Item has been de-selected -- remove frame
      CBrush br(cr);
      pDC->FrameRect(&lpDIS->rcItem, &br);
   }
}
```

See Also

Reference

CMenu Class

Hierarchy Chart

Other Resources

CMenu::EnableMenuItem

Enables, disables, or dims a menu item.

```
UINT EnableMenuItem(
   UINT nIDEnableItem,
   UINT nEnable
);
```

Parameters

nIDEnableItem

Specifies the menu item to be enabled, as determined by *nEnable*. This parameter can specify pop-up menu items as well as standard menu items.

nEnable

Specifies the action to take. It can be a combination of **MF_DISABLED**, **MF_ENABLED**, or **MF_GRAYED**, with **MF_BYCOMMAND** or **MF_BYPOSITION**. These values can be combined by using the bitwise OR operator. These values have the following meanings:

- **MF_BYCOMMAND** Specifies that the parameter gives the command ID of the existing menu item. This is the default.
- **MF_BYPOSITION** Specifies that the parameter gives the position of the existing menu item. The first item is at position 0.
- **MF_DISABLED** Disables the menu item so that it cannot be selected but does not dim it.
- **MF_ENABLED** Enables the menu item so that it can be selected and restores it from its dimmed state.
- **MF_GRAYED** Disables the menu item so that it cannot be selected and dims it.

Return Value

Previous state (**MF_DISABLED**, **MF_ENABLED**, or **MF_GRAYED**) or –1 if not valid.

Remarks

The CreateMenu, InsertMenu, ModifyMenu, and LoadMenuIndirect member functions can also set the state (enabled, disabled, or dimmed) of a menu item.

Using the **MF_BYPOSITION** value requires an application to use the correct **CMenu**. If the **CMenu** of the menu bar is used, a top-level menu item (an item in the menu bar) is affected. To set the state of an item in a pop-up or nested pop-up menu by position, an application must specify the **CMenu** of the pop-up menu.

When an application specifies the **MF_BYCOMMAND** flag, Windows checks all pop-up menu items that are subordinate to the **CMenu**; therefore, unless duplicate menu items are present, using the **CMenu** of the menu bar is sufficient.

Example

```
// The code fragment below shows how to disable (and gray out) the
// File\New menu item.
// NOTE: m_bAutoMenuEnable is set to FALSE in the constructor of
// CMainFrame so no ON_UPDATE_COMMAND_UI or ON_COMMAND handlers are
// needed, and CMenu::EnableMenuItem() will work as expected.

CMenu* mmenu = GetMenu();
CMenu* submenu = mmenu->GetSubMenu(0);
submenu->EnableMenuItem(ID_FILE_NEW, MF_BYCOMMAND | MF_DISABLED | MF_GRAYED);
```

See Also

Reference

CMenu Class
Hierarchy Chart
CMenu::GetMenuState

EnableMenuItem

Other Resources

CMenu Members

CMenu::FromHandle

Returns a pointer to a **CMenu** object given a Windows handle to a menu.

```
static CMenu* PASCAL FromHandle(
   HMENU hMenu
);
```

Parameters

hMenu

A Windows handle to a menu.

Return Value

A pointer to a **CMenu** that may be temporary or permanent.

Remarks

If a **CMenu** object is not already attached to the Windows menu object, a temporary **CMenu** object is created and attached.

This temporary **CMenu** object is only valid until the next time the application has idle time in its event loop, at which time all temporary objects are deleted.

Example

See the example for CMenu::CreateMenu.

See Also

Reference

CMenu Class

Hierarchy Chart

Other Resources

CMenu Members

CMenu::GetDefaultItem

Determines the default menu item on the specified menu.

```
UINT GetDefaultItem(
   UINT gmdiFlags,
   BOOL fByPos = FALSE
);
```

Parameters

gmdiFlags

Value specifying how the function searches for menu items. This parameter can be none, one, or a combination of the following values:

Value	Meaning
GMDI_GOINTOPOPUPS	Specifies that, if the default item is one that opens a submenu, the function is to search in the corresponding submenu recursively. If the submenu has no default item, the return value identifies the item that opens the submenu. By default, the function returns the first default item on the specified menu, regardless of whether it is an item that opens a submenu.
GMDI_USEDISABLED	Specifies that the function is to return a default item, even if it is disabled. By default, the function skips disabled or grayed items.

fByPos

Value specifying whether to retrieve the menu item's identifier or its position. If this parameter is **FALSE**, the identifier is returned. Otherwise, the position is returned.

Return Value

If the function succeeds, the return value is the identifier or position of the menu item. If the function fails, the return value is -1.

Remarks

This member function implements the behavior of the Win32 function GetMenuDefaultItem, as described in the *Platform SDK*.

Example

See the example for CMenu::InsertMenu.

See Also

Reference

CMenu Class
Hierarchy Chart
CMenu::SetDefaultItem

Other Resources

CMenu Members

CMenu::GetMenuContextHelpId

Retrieves the context help ID associated with **CMenu**.

```
DWORD GetMenuContextHelpId( ) const;
```

Return Value

The context help ID currently associated with **CMenu** if it has one; zero otherwise.

Example

See the example for CMenu::InsertMenu.

See Also

Reference

CMenu Class
Hierarchy Chart
CMenu::SetMenuContextHelpId
GetMenuContextHelpId

Other Resources

CMenu Members

CMenu::GetMenuInfo

Retrieves information for a menu.

```
BOOL GetMenuInfo(
   LPMENUINFO lpcmi
) const;
```

Parameters

lpcmi

A pointer to a MENUINFO structure containing information for the menu.

Return Value

If the function succeeds, the return value is nonzero; otherwise, the return value is zero.

Remarks

Call this function to retrieve information about the menu.

See Also

Reference

CMenu Class

Hierarchy Chart

CMenu::SetMenuInfo

Other Resources

CMenu Members

CMenu::GetMenuItemCount

Determines the number of items in a pop-up or top-level menu.

```
UINT GetMenuItemCount( ) const;
```

Return Value

The number of items in the menu if the function is successful; otherwise –1.

Example

See the example for CWnd::GetMenu.

See Also

Reference

CMenu Class
Hierarchy Chart
CWnd::GetMenu
CMenu::GetMenuItemID
CMenu::GetSubMenu
GetMenuItemCount

Other Resources

CMenu Members

CMenu::GetMenuItemID

Obtains the menu-item identifier for a menu item located at the position defined by *nPos*.

```
UINT GetMenuItemID(
   int nPos
) const;
```

Parameters

nPos

Specifies the position (zero-based) of the menu item whose ID is being retrieved.

Return Value

The item ID for the specified item in a pop-up menu if the function is successful. If the specified item is a pop-up menu (as opposed to an item within the pop-up menu), the return value is –1. If *nPos* corresponds to a **SEPARATOR** menu item, the return value is 0.

Example

See the example for CMenu::InsertMenu.

See Also

Reference

CMenu Class
Hierarchy Chart
CWnd::GetMenu
CMenu::GetMenuItemCount
CMenu::GetSubMenu
GetMenuItemID

Other Resources

CMenu Members

CMenu::GetMenuItemInfo

Retrieves information about a menu item.

```
BOOL GetMenuItemInfo(
   UINT uItem,
   LPMENUITEMINFO lpMenuItemInfo,
   BOOL fByPos = FALSE
);
```

Parameters

uItem

Identifier or position of the menu item to get information about. The meaning of this parameter depends on the value of *ByPos*.

lpMenuItemInfo

A pointer to a MENUITEMINFO, as described in the *Platform SDK*, that contains information about the menu.

fByPos

Value specifying the meaning of *nIDItem*. By default, *ByPos* is **FALSE**, which indicates that uItem is a menu item identifier. If *ByPos* is not set to **FALSE**, it indicates a menu item position.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero. To get extended error information, use the Win32 function GetLastError, as described in the *Platform SDK*.

Remarks

This member function implements the behavior of the of the Win32 function GetMenuItemInfo, as described in the *Platform SDK*. Note that in the MFC implementation of **GetMenuItemInfo**, you do not use a handle to a menu.

Example

```
// CMainFrame::OnToggleTestMenuItem() is a menu command handler for
// "Test" menu item (whose resource id is ID_HELP_TEST). It toggles
// the checked or unchecked state of the "Test" menu item.
// CMainFrame is a CFrameWnd-derived class.

void CMainFrame::OnToggleTestMenuItem()
{
   // Get the popup menu which contains the "Test" menu item.
   CMenu* mmenu = GetMenu();
   CMenu* submenu = mmenu->GetSubMenu(3);

   // Check the state of the "Test" menu item. Check the menu item
   // if it is currently unchecked. Otherwise, uncheck the menu item
   // if it is not currently checked.
   MENUITEMINFO info;
   info.cbSize = sizeof (MENUITEMINFO); // must fill up this field
   info.fMask = MIIM_STATE;             // get the state of the menu item
   VERIFY(submenu->GetMenuItemInfo(ID_HELP_TEST, &info));

   if (info.fState & MF_CHECKED)
      submenu->CheckMenuItem(ID_HELP_TEST, MF_UNCHECKED | MF_BYCOMMAND);
   else
      submenu->CheckMenuItem(ID_HELP_TEST, MF_CHECKED | MF_BYCOMMAND);
}
```

See Also

Reference

CMenu Class

Hierarchy Chart
CWnd::GetMenu
CMenu::GetMenuItemCount
CMenu::GetMenuItemID
Other Resources
CMenu Members

CMenu::GetMenuState

Returns the status of the specified menu item or the number of items in a pop-up menu.

```
UINT GetMenuState(
   UINT nID,
   UINT nFlags
) const;
```

Parameters

nID

Specifies the menu item ID, as determined by *nFlags*.

nFlags

Specifies the nature of *nID*. It can be one of the following values:

- **MF_BYCOMMAND** Specifies that the parameter gives the command ID of the existing menu item. This is the default.
- **MF_BYPOSITION** Specifies that the parameter gives the position of the existing menu item. The first item is at position 0.

Return Value

The value 0xFFFFFFFF if the specified item does not exist. If *nId* identifies a pop-up menu, the high-order byte contains the number of items in the pop-up menu and the low-order byte contains the menu flags associated with the pop-up menu. Otherwise the return value is a mask (Boolean OR) of the values from the following list (this mask describes the status of the menu item that *nId* identifies):

- **MF_CHECKED** Acts as a toggle with **MF_UNCHECKED** to place the default check mark next to the item. When the application supplies check-mark bitmaps (see the **SetMenuItemBitmaps** member function), the "check mark on" bitmap is displayed.
- **MF_DISABLED** Disables the menu item so that it cannot be selected but does not dim it.
- **MF_ENABLED** Enables the menu item so that it can be selected and restores it from its dimmed state. Note that the value of this constant is 0; an application should not test against 0 for failure when using this value.
- **MF_GRAYED** Disables the menu item so that it cannot be selected and dims it.
- **MF_MENUBARBREAK** Places the item on a new line in static menus or in a new column in pop-up menus. The new pop-up menu column will be separated from the old column by a vertical dividing line.
- **MF_MENUBREAK** Places the item on a new line in static menus or in a new column in pop-up menus. No dividing line is placed between the columns.
- **MF_SEPARATOR** Draws a horizontal dividing line. Can only be used in a pop-up menu. This line cannot be dimmed, disabled, or highlighted. Other parameters are ignored.
- **MF_UNCHECKED** Acts as a toggle with **MF_CHECKED** to remove a check mark next to the item. When the application supplies check-mark bitmaps (see the **SetMenuItemBitmaps** member function), the "check mark off" bitmap is displayed. Note that the value of this constant is 0; an application should not test against 0 for failure when using this value.

Example

```
// CMainFrame::OnToggleTestMenuItem() is a menu command handler for
// "Test" menu item (whose resource id is ID_HELP_TEST). It toggles
// the checked or unchecked state of the "Test" menu item.
// CMainFrame is a CFrameWnd-derived class.

void CMainFrame::OnToggleTestMenuItem()
{
   // Get the popup menu which contains the "Test" menu item.
```

```
    CMenu* mmenu = GetMenu();
    CMenu* submenu = mmenu->GetSubMenu(3);

    // Check the state of the "Test" menu item. Check the menu item
    // if it is currently unchecked. Otherwise, uncheck the menu item
    // if it is not currently checked.
    UINT state = submenu->GetMenuState(ID_HELP_TEST, MF_BYCOMMAND);
    ASSERT(state != 0xFFFFFFFF);

    if (state & MF_CHECKED)
       submenu->CheckMenuItem(ID_HELP_TEST, MF_UNCHECKED | MF_BYCOMMAND);
    else
       submenu->CheckMenuItem(ID_HELP_TEST, MF_CHECKED | MF_BYCOMMAND);
 }
```

See Also

Reference

CMenu Class

Hierarchy Chart

GetMenuState

CMenu::CheckMenuItem

CMenu::EnableMenuItem

Other Resources

CMenu Members

CMenu::GetMenuString

Copies the label of the specified menu item to the specified buffer.

```
int GetMenuString(
   UINT nIDItem,
   LPTSTR lpString,
   int nMaxCount,
   UINT nFlags
) const;
int GetMenuString(
   UINT nIDItem,
   CString& rString,
   UINT nFlags
) const;
```

Parameters

nIDItem

Specifies the integer identifier of the menu item or the offset of the menu item in the menu, depending on the value of *nFlags*.

lpString

Points to the buffer that is to receive the label.

rString

A reference to a **CString** object that is to receive the copied menu string.

nMaxCount

Specifies the maximum length (in characters) of the label to be copied. If the label is longer than the maximum specified in *nMaxCount*, the extra characters are truncated.

nFlags

Specifies the interpretation of the *nIDItem* parameter. It can be one of the following values:

nFlags	Interpretation of nIDItem
MF_BYCOMMAND	Specifies that the parameter gives the command ID of the existing menu item. This is the default if neither **MF_BYCOMMAND** nor **MF_BYPOSITION** is set.
MF_BYPOSITION	Specifies that the parameter gives the position of the existing menu item. The first item is at position 0.

Return Value

Specifies the actual number of characters copied to the buffer, not including the null terminator.

Remarks

The *nMaxCount* parameter should be one larger than the number of characters in the label to accommodate the null character that terminates a string.

Example

See the example for CMenu::InsertMenu.

See Also

Reference

CMenu Class
Hierarchy Chart
CMenu::GetMenuState
CMenu::ModifyMenu

GetMenuString

Other Resources

CMenu Members

CMenu::GetSafeHmenu

Returns the **HMENU** wrapped by this **CMenu** object, or a **NULL CMenu** pointer.

```
HMENU GetSafeHmenu( ) const;
```

Example

See the example for CMenu::LoadMenu.

See Also

Reference

CMenu Class
Hierarchy Chart
GetSubMenu

Other Resources

CMenu Members

CMenu::GetSubMenu

Retrieves the **CMenu** object of a pop-up menu.

```
CMenu* GetSubMenu(
   int nPos
) const;
```

Parameters

nPos

Specifies the position of the pop-up menu contained in the menu. Position values start at 0 for the first menu item. The pop-up menu's identifier cannot be used in this function.

Return Value

A pointer to a **CMenu** object whose **m_hMenu** member contains a handle to the pop-up menu if a pop-up menu exists at the given position; otherwise **NULL**. If a **CMenu** object does not exist, then a temporary one is created. The **CMenu** pointer returned should not be stored.

Example

See the example for CMenu::TrackPopupMenu.

See Also

Reference

CMenu Class
Hierarchy Chart
CWnd::GetMenu
CMenu::GetMenuItemID
GetSubMenu

Other Resources

CMenu Members

CMenu::InsertMenu

Inserts a new menu item at the position specified by *nPosition* and moves other items down the menu.

```
BOOL InsertMenu(
   UINT nPosition,
   UINT nFlags,
   UINT_PTR nIDNewItem = 0,
   LPCTSTR lpszNewItem = NULL
);
BOOL InsertMenu(
   UINT nPosition,
   UINT nFlags,
   UINT_PTR nIDNewItem,
   const CBitmap* pBmp
);
```

Parameters

nPosition

Specifies the menu item before which the new menu item is to be inserted. The *nFlags* parameter can be used to interpret *nPosition* in the following ways:

nFlags	Interpretation of nPosition
MF_BYCOMMAND	Specifies that the parameter gives the command ID of the existing menu item. This is the default if neither **MF_BYCOMMAND** nor **MF_BYPOSITION** is set.
MF_BYPOSITION	Specifies that the parameter gives the position of the existing menu item. The first item is at position 0. If *nPosition* is –1, the new menu item is appended to the end of the menu.

nFlags

Specifies how *nPosition* is interpreted and specifies information about the state of the new menu item when it is added to the menu. For a list of the flags that may be set, see the AppendMenu member function. To specify more than one value, use the bitwise OR operator to combine them with the **MF_BYCOMMAND** or **MF_BYPOSITION** flag.

nIDNewItem

Specifies either the command ID of the new menu item or, if *nFlags* is set to **MF_POPUP**, the menu handle (**HMENU**) of the pop-up menu. The *nIDNewItem* parameter is ignored (not needed) if *nFlags* is set to **MF_SEPARATOR**.

lpszNewItem

Specifies the content of the new menu item. *nFlags* can be used to interpret *lpszNewItem* in the following ways:

nFlags	Interpretation of lpszNewItem
MF_OWNERDRAW	Contains an application-supplied 32-bit value that the application can use to maintain additional data associated with the menu item. This 32-bit value is available to the application in the **itemData** member of the structure supplied by the WM_MEASUREITEM and WM_DRAWITEM messages. These messages are sent when the menu item is initially displayed or is changed.
MF_STRING	Contains a long pointer to a null-terminated string. This is the default interpretation.
MF_SEPARATOR	The *lpszNewItem* parameter is ignored (not needed).

pBmp

Points to a **CBitmap** object that will be used as the menu item.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The application can specify the state of the menu item by setting values in *nFlags*.

Whenever a menu that resides in a window is changed (whether or not the window is displayed), the application should call **CWnd::DrawMenuBar**.

When *nIDNewItem* specifies a pop-up menu, it becomes part of the menu in which it is inserted. If that menu is destroyed, the inserted menu will also be destroyed. An inserted menu should be detached from a **CMenu** object to avoid conflict.

If the active multiple document interface (MDI) child window is maximized and an application inserts a pop-up menu into the MDI application's menu by calling this function and specifying the **MF_BYPOSITION** flag, the menu is inserted one position farther left than expected. This happens because the Control menu of the active MDI child window is inserted into the first position of the MDI frame window's menu bar. To position the menu properly, the application must add 1 to the position value that would otherwise be used. An application can use the **WM_MDIGETACTIVE** message to determine whether the currently active child window is maximized.

Example

```
// CMainFrame::OnChangeFileMenu() is a menu command handler for
// CMainFrame class, which in turn is a CFrameWnd-derived class.
// It modifies the File menu by inserting, removing and renaming
// some menu items. Other operations include associating a context
// help id and setting default menu item to the File menu.
// CMainFrame is a CFrameWnd-derived class.
void CMainFrame::OnChangeFileMenu()
{
   // Get the menu from the application window.
   CMenu* mmenu = GetMenu();

   // Look for "File" menu.
   int pos = FindMenuItem(mmenu, "&File");
   if (pos == -1)
      return;

   // Remove "New" menu item from the File menu.
   CMenu* submenu = mmenu->GetSubMenu(pos);
   pos = FindMenuItem(submenu, "&New\tCtrl+N");
   if (pos > -1)
      submenu->RemoveMenu(pos, MF_BYPOSITION);

   // Look for "Open" menu item from the File menu. Insert a new
   // menu item called "Close" right after the "Open" menu item.
   // ID_CLOSEFILE is the command id for the "Close" menu item.
   pos = FindMenuItem(submenu, "&Open...\tCtrl+O");
   if (pos > -1)
      submenu->InsertMenu(pos + 1, MF_BYPOSITION, ID_CLOSEFILE, "&Close");

   // Rename menu item "Save" to "Save Selection".
   pos = FindMenuItem(submenu, "&Save\tCtrl+S");
   if (pos > -1)
   {
      UINT id = submenu->GetMenuItemID(pos);
      submenu->ModifyMenu(id, MF_BYCOMMAND, id, "&Save Selection");
   }

   // Associate a context help ID with File menu, if one is not found.
   // ID_FILE_CONTEXT_HELPID is the context help ID for the File menu
   // that is defined in resource file.
   if (submenu->GetMenuContextHelpId() == 0)
      submenu->SetMenuContextHelpId(ID_FILE_CONTEXT_HELPID);
```

```
    // Set "Open" menu item as the default menu item for the File menu,
    // if one is not found. So, when a user double-clicks the File
    // menu, the system sends a command message to the menu's owner
    // window and closes the menu as if the File\Open command item had
    // been chosen.
    if (submenu->GetDefaultItem(GMDI_GOINTOPOPUPS, TRUE) == -1)
    {
       pos = FindMenuItem(submenu, "&Open...\tCtrl+O");
       submenu->SetDefaultItem(pos, TRUE);
    }
 }

 // FindMenuItem() will find a menu item string from the specified
 // popup menu and returns its position (0-based) in the specified
 // popup menu. It returns -1 if no such menu item string is found.
 int FindMenuItem(CMenu* Menu, LPCTSTR MenuString)
 {
    ASSERT(Menu);
    ASSERT(::IsMenu(Menu->GetSafeHmenu()));

    int count = Menu->GetMenuItemCount();
    for (int i = 0; i < count; i++)
    {
       CString str;
       if (Menu->GetMenuString(i, str, MF_BYPOSITION) &&
          (strcmp(str, MenuString) == 0))
          return i;
    }

    return -1;
 }
```

See Also

Reference

CMenu Class
Hierarchy Chart
CMenu::AppendMenu
CWnd::DrawMenuBar
CMenu::SetMenuItemBitmaps
CMenu::Detach
InsertMenu

Other Resources

CMenu Members

CMenu::InsertMenuItem

Inserts a new menu item at the specified position in a menu.

```
BOOL InsertMenuItem(
   UINT uItem,
   LPMENUITEMINFO lpMenuItemInfo,
   BOOL fByPos = FALSE
);
```

Parameters

uItem

See description of *uItem* in InsertMenuItem in the *Platform SDK*.

lpMenuItemInfo

See description of *lpmii* in **InsertMenuItem** in the *Platform SDK*.

fByPos

See description of *fByPosition* in **InsertMenuItem** in the *Platform SDK*.

Remarks

This function wraps InsertMenuItem, described in the *Platform SDK*.

See Also

Reference

CMenu Class

Hierarchy Chart

Other Resources

CMenu Members

CMenu::LoadMenu

Loads a menu resource from the application's executable file and attaches it to the **CMenu** object.

```
BOOL LoadMenu(
   LPCTSTR lpszResourceName
);
BOOL LoadMenu(
   UINT nIDResource
);
```

Parameters

lpszResourceName

Points to a null-terminated string that contains the name of the menu resource to load.

nIDResource

Specifies the menu ID of the menu resource to load.

Return Value

Nonzero if the menu resource was loaded successfully; otherwise 0.

Remarks

Before exiting, an application must free system resources associated with a menu if the menu is not assigned to a window. An application frees a menu by calling the DestroyMenu member function.

Example

```
// CMainFrame::OnReplaceMenu() is a menu command handler for CMainFrame
// class, which in turn is a CFrameWnd-derived class. It loads a new
// menu resource and replaces the SDI application window's menu bar with
// this new menu. CMainFrame is a CFrameWnd-derived class.
void CMainFrame::OnReplaceMenu()
{
   // Load the new menu.
   m_NewMenu.LoadMenu(IDR_SHORT_MENU);
   ASSERT(m_NewMenu);

   // Remove and destroy the old menu
   SetMenu(NULL);
   ::DestroyMenu(m_hMenuDefault);

   // Add the new menu
   SetMenu(&m_NewMenu);

   // Assign default menu
   m_hMenuDefault = m_NewMenu.GetSafeHmenu();  // or m_NewMenu.m_hMenu;
}
```

See Also

Reference

CMenu Class
Hierarchy Chart
CMenu::AppendMenu
CMenu::DestroyMenu
CMenu::LoadMenuIndirect
LoadMenu

Other Resources

CMenu Members

CMenu::LoadMenuIndirect

Loads a resource from a menu template in memory and attaches it to the **CMenu** object.

```
BOOL LoadMenuIndirect(
   const void* lpMenuTemplate
);
```

Parameters

lpMenuTemplate

Points to a menu template (which is a single MENUITEMTEMPLATEHEADER structure and a collection of one or more MENUITEMTEMPLATE structures). For more information on these two structures, see the *Platform SDK*.

Return Value

Nonzero if the menu resource was loaded successfully; otherwise 0.

Remarks

A menu template is a header followed by a collection of one or more MENUITEMTEMPLATE structures, each of which may contain one or more menu items and pop-up menus.

The version number should be 0.

The **mtOption** flags should include **MF_END** for the last item in a pop-up list and for the last item in the main list. See the **AppendMenu** member function for other flags. The **mtId** member must be omitted from the **MENUITEMTEMPLATE** structure when **MF_POPUP** is specified in **mtOption**.

The space allocated for the **MENUITEMTEMPLATE** structure must be large enough for **mtString** to contain the name of the menu item as a null-terminated string.

Before exiting, an application must free system resources associated with a menu if the menu is not assigned to a window. An application frees a menu by calling the DestroyMenu member function.

Example

```
// CMainFrame::OnLoadMenuIndirect() is a menu command handler for
// CMainFrame class, which in turn is a CFrameWnd-derived class. It
// shows how to use LoadMenuIndirect() to load a resource from a
// menu template in memory.
void CMainFrame::OnLoadMenuIndirect()
{
   // For simplicity, allocate 500 bytes from stack. May use
   // GlobalAlloc() to allocate memory bytes from heap.
   BYTE milist[500];
   memset(milist, 0, 500);

   // Fill up the MENUITEMTEMPLATEHEADER structure.
   MENUITEMTEMPLATEHEADER* mheader = (MENUITEMTEMPLATEHEADER*) milist;
   mheader->versionNumber = 0;
   mheader->offset = 0;

   int bytes_used = sizeof(MENUITEMTEMPLATEHEADER);

   // Add the following menu items to menu bar:
   // File     Edit
   //   Exit     Copy
   //            Paste
   bytes_used += AddMenuItem(milist + bytes_used, L"&File", 0, TRUE, FALSE);
   bytes_used += AddMenuItem(milist + bytes_used, L"E&xit", ID_APP_EXIT, FALSE, TRUE);
   bytes_used += AddMenuItem(milist + bytes_used, L"&Edit", 0, TRUE, TRUE);
   bytes_used += AddMenuItem(milist + bytes_used, L"&Copy", ID_EDIT_COPY, FALSE, FALSE);
   bytes_used += AddMenuItem(milist + bytes_used, L"&Paste", ID_EDIT_PASTE, FALSE, TRUE);

   // Load resource from a menu template in memory.
```

```
    ASSERT(m_NewMenu.LoadMenuIndirect(milist));

    // Remove and destroy old menu
    SetMenu(NULL);
    ::DestroyMenu(m_hMenuDefault);

    // Add new menu.
    SetMenu(&m_NewMenu);

    // Assign default menu
    m_hMenuDefault = m_NewMenu.m_hMenu;
}

// This is a helper function for adding a menu item (either a popup
// or command item) to the specified menu template.
//
//    MenuTemplate - pointer to a menu template
//    MenuString   - string for the menu item to be added
//    MenuID       - id for the command item. Its value is ignored if
                     IsPopup is TRUE.
//    IsPopup      - TRUE for popup menu (or submenu); FALSE for command
                     item
//    LastItem     - TRUE if MenuString is the last item for the popup;
                     FALSE otherwise.
UINT AddMenuItem(LPVOID MenuTemplate, WCHAR* MenuString, WORD MenuID, BOOL IsPopup, BOOL La
stItem)
{
    MENUITEMTEMPLATE* mitem = (MENUITEMTEMPLATE*) MenuTemplate;

    UINT  bytes_used = 0;
    if (IsPopup)         // for popup menu
    {
        if (LastItem)
            mitem->mtOption = MF_POPUP | MF_END;
        else
            mitem->mtOption = MF_POPUP;
        bytes_used += sizeof (mitem->mtOption);

        mitem = (MENUITEMTEMPLATE*) ((BYTE*) MenuTemplate + bytes_used);
        // a popup doesn't have mtID!!!

        wcscpy((WCHAR*) mitem, MenuString);
        bytes_used += sizeof (WCHAR) * (wcslen(MenuString) + 1); // include '\0'
    }
    else      // for command item
    {
        mitem->mtOption = LastItem ? MF_END : 0;
        mitem->mtID = MenuID;
        wcscpy(mitem->mtString, MenuString);
        bytes_used += sizeof (mitem->mtOption ) + sizeof (mitem->mtID) +
            sizeof (WCHAR) * (wcslen(MenuString) + 1);   // include '\0'
    }

    return bytes_used;
}
```

See Also

Reference

CMenu Class
Hierarchy Chart
CMenu::DestroyMenu
CMenu::LoadMenu
LoadMenuIndirect
CMenu::AppendMenu

Other Resources

CMenu Members

CMenu::MeasureItem

Called by the framework when a menu with the owner-draw style is created.

```
virtual void MeasureItem(
   LPMEASUREITEMSTRUCT lpMeasureItemStruct
);
```

Parameters

lpMeasureItemStruct

A pointer to a **MEASUREITEMSTRUCT** structure.

Remarks

By default, this member function does nothing. Override this member function and fill in the **MEASUREITEMSTRUCT** structure to inform Windows of the menu's dimensions.

See CWnd::OnMeasureItem for a description of the **MEASUREITEMSTRUCT** structure.

Example

The following code is from the MFC CTRLTEST sample:

```
// Override MeasureItem() to return the size of the menu item.
// CColorMenu is a CMenu-derived class.

#define COLOR_BOX_WIDTH     20
#define COLOR_BOX_HEIGHT    20

void CColorMenu::MeasureItem(LPMEASUREITEMSTRUCT lpMIS)
{
   // all items are of fixed size
   lpMIS->itemWidth = COLOR_BOX_WIDTH;
   lpMIS->itemHeight = COLOR_BOX_HEIGHT;
}
```

See Also

Reference

CMenu Class

Hierarchy Chart

Other Resources

CMenu Members

CMenu::ModifyMenu

Changes an existing menu item at the position specified by *nPosition*.

```
BOOL ModifyMenu(
   UINT nPosition,
   UINT nFlags,
   UINT_PTR nIDNewItem = 0,
   LPCTSTR lpszNewItem = NULL
);
BOOL ModifyMenu(
   UINT nPosition,
   UINT nFlags,
   UINT_PTR nIDNewItem,
   const CBitmap* pBmp
);
```

Parameters

nPosition

Specifies the menu item to be changed. The *nFlags* parameter can be used to interpret *nPosition* in the following ways:

nFlags	Interpretation of nPosition
MF_BYCOMMAND	Specifies that the parameter gives the command ID of the existing menu item. This is the default if neither **MF_BYCOMMAND** nor **MF_BYPOSITION** is set.
MF_BYPOSITION	Specifies that the parameter gives the position of the existing menu item. The first item is at position 0.

nFlags

Specifies how *nPosition* is interpreted and gives information about the changes to be made to the menu item. For a list of flags that may be set, see the AppendMenu member function.

nIDNewItem

Specifies either the command ID of the modified menu item or, if *nFlags* is set to **MF_POPUP**, the menu handle (**HMENU**) of a pop-up menu. The *nIDNewItem* parameter is ignored (not needed) if *nFlags* is set to **MF_SEPARATOR**.

lpszNewItem

Specifies the content of the new menu item. The *nFlags* parameter can be used to interpret *lpszNewItem* in the following ways:

nFlags	Interpretation of lpszNewItem
MF_OWNERDRAW	Contains an application-supplied 32-bit value that the application can use to maintain additional data associated with the menu item. This 32-bit value is available to the application when it processes **MF_MEASUREITEM** and **MF_DRAWITEM**.
MF_STRING	Contains a long pointer to a null-terminated string or to a **CString**.
MF_SEPARATOR	The *lpszNewItem* parameter is ignored (not needed).

pBmp

Points to a **CBitmap** object that will be used as the menu item.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The application specifies the new state of the menu item by setting values in *nFlags*. If this function replaces a pop-up menu associated with the menu item, it destroys the old pop-up menu and frees the memory used by the pop-up menu.

When *nIDNewItem* specifies a pop-up menu, it becomes part of the menu in which it is inserted. If that menu is destroyed, the inserted menu will also be destroyed. An inserted menu should be detached from a **CMenu** object to avoid conflict.

Whenever a menu that resides in a window is changed (whether or not the window is displayed), the application should call **CWnd::DrawMenuBar**. To change the attributes of existing menu items, it is much faster to use the **CheckMenuItem** and **EnableMenuItem** member functions.

Example

See the example for CMenu::InsertMenu.

See Also

Reference

CMenu Class
Hierarchy Chart
CMenu::AppendMenu
CMenu::InsertMenu
CMenu::CheckMenuItem
CWnd::DrawMenuBar
CMenu::EnableMenuItem
CMenu::SetMenuItemBitmaps
CMenu::Detach
ModifyMenu

Other Resources

CMenu Members

CMenu::RemoveMenu

Deletes a menu item with an associated pop-up menu from the menu.

```
BOOL RemoveMenu(
   UINT nPosition,
   UINT nFlags
);
```

Parameters

nPosition

Specifies the menu item to be removed. The *nFlags* parameter can be used to interpret *nPosition* in the following ways:

nFlags	Interpretation of nPosition
MF_BYCOMMAND	Specifies that the parameter gives the command ID of the existing menu item. This is the default if neither **MF_BYCOMMAND** nor **MF_BYPOSITION** is set.
MF_BYPOSITION	Specifies that the parameter gives the position of the existing menu item. The first item is at position 0.

nFlags

Specifies how *nPosition* is interpreted.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

It does not destroy the handle for a pop-up menu, so the menu can be reused. Before calling this function, the application may call the **GetSubMenu** member function to retrieve the pop-up **CMenu** object for reuse.

Whenever a menu that resides in a window is changed (whether or not the window is displayed), the application must call **CWnd::DrawMenuBar**.

Example

See the example for CMenu::InsertMenu.

See Also

Reference

CMenu Class
Hierarchy Chart
CWnd::DrawMenuBar
CMenu::GetSubMenu
RemoveMenu

Other Resources

CMenu Members

CMenu::SetDefaultItem

Sets the default menu item for the specified menu.

```
BOOL SetDefaultItem(
   UINT uItem,
   BOOL fByPos = FALSE
);
```

Parameters

uItem

Identifier or position of the new default menu item or - 1 for no default item. The meaning of this parameter depends on the value of *fByPos*.

fByPos

Value specifying the meaning of *uItem*. If this parameter is **FALSE**, *uItem* is a menu item identifier. Otherwise, it is a menu item position.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero. To get extended error information, use the Win32 function GetLastError, as described in the *Platform SDK*.

Remarks

This member function implements the behavior of the Win32 function SetMenuDefaultItem, as described in the *Platform SDK*.

Example

See the example for CMenu::InsertMenu.

See Also

Reference

CMenu Class
Hierarchy Chart
CMenu::GetDefaultItem

Other Resources

CMenu Members

CMenu::SetMenuContextHelpId

Associates a context help ID with **CMenu**.

```
BOOL SetMenuContextHelpId(
   DWORD dwContextHelpId
);
```

Parameters

dwContextHelpId

Context help ID to associate with **CMenu**.

Return Value

Nonzero if successful; otherwise 0

Remarks

All items in the menu share this identifier — it is not possible to attach a help context identifier to the individual menu items.

Example

See the example for CMenu::InsertMenu.

See Also

Reference

CMenu Class
Hierarchy Chart
CMenu::GetMenuContextHelpId
SetMenuContextHelpId

Other Resources

CMenu Members

CMenu::SetMenuInfo

Sets information for a menu.

```
BOOL SetMenuInfo(
   LPCMENUINFO lpcmi
);
```

Parameters

lpcmi

A pointer to a MENUINFO structure containing information for the menu.

Return Value

If the function succeeds, the return value is nonzero; otherwise, the return value is zero.

Remarks

Call this function to set specific information about the menu.

See Also

Reference

CMenu Class

Hierarchy Chart

CMenu::GetMenuInfo

Other Resources

CMenu Members

CMenu::SetMenuItemBitmaps

Associates the specified bitmaps with a menu item.

```
BOOL SetMenuItemBitmaps(
   UINT nPosition,
   UINT nFlags,
   const CBitmap* pBmpUnchecked,
   const CBitmap* pBmpChecked
);
```

Parameters

nPosition

Specifies the menu item to be changed. The *nFlags* parameter can be used to interpret *nPosition* in the following ways:

nFlags	Interpretation of nPosition
MF_BYCOMMAND	Specifies that the parameter gives the command ID of the existing menu item. This is the default if neither **MF_BYCOMMAND** nor **MF_BYPOSITION** is set.
MF_BYPOSITION	Specifies that the parameter gives the position of the existing menu item. The first item is at position 0.

nFlags

Specifies how *nPosition* is interpreted.

pBmpUnchecked

Specifies the bitmap to use for menu items that are not checked.

pBmpChecked

Specifies the bitmap to use for menu items that are checked.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

Whether the menu item is checked or unchecked, Windows displays the appropriate bitmap next to the menu item.

If either *pBmpUnchecked* or *pBmpChecked* is **NULL**, then Windows displays nothing next to the menu item for the corresponding attribute. If both parameters are **NULL**, Windows uses the default check mark when the item is checked and removes the check mark when the item is unchecked.

When the menu is destroyed, these bitmaps are not destroyed; the application must destroy them.

The Windows **GetMenuCheckMarkDimensions** function retrieves the dimensions of the default check mark used for menu items. The application uses these values to determine the appropriate size for the bitmaps supplied with this function. Get the size, create your bitmaps, and then set them.

Example

```
// The code fragment below shows how to associate bitmaps with the
// "Test" menu item. Whether the "Test" menu item is checked or
// unchecked, Windows displays the appropriate bitmap next to the menu
// item. Both IDB_CHECKBITMAP and IDB_UNCHECKBITMAP bitmaps are loaded
// in OnCreate() and destroyed in the destructor of CMainFrame class.
// CMainFrame is a CFrameWnd-derived class.

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
```

```
        return -1;

    // Load bitmaps from resource. Both m_CheckBitmap and m_UnCheckBitmap
    // are member variables of CMainFrame class of type CBitmap.
    ASSERT(m_CheckBitmap.LoadBitmap(IDB_CHECKBITMAP));
    ASSERT(m_UnCheckBitmap.LoadBitmap(IDB_UNCHECKBITMAP));

    // Associate bitmaps with the "Test" menu item.
    CMenu* mmenu = GetMenu();
    CMenu* submenu = mmenu->GetSubMenu(3);
    ASSERT(submenu->SetMenuItemBitmaps(ID_HELP_TEST, MF_BYCOMMAND,
        &m_CheckBitmap, &m_UnCheckBitmap));

    // ...
}

CMainFrame::~CMainFrame()
{
    // Destroy the bitmap objects if they are loaded successfully
    // in OnCreate().
    if (m_CheckBitmap.m_hObject)
        m_CheckBitmap.DeleteObject();

    if (m_UnCheckBitmap.m_hObject)
        m_UnCheckBitmap.DeleteObject();
}
```

See Also

Reference

CMenu Class

Hierarchy Chart

GetMenuCheckMarkDimensions

SetMenuItemBitmaps

Other Resources

CMenu Members

CMenu::SetMenuItemInfo

Changes information about a menu item.

```
BOOL SetMenuItemInfo(
   UINT uItem,
   LPMENUITEMINFO lpMenuItemInfo,
   BOOL fByPos = FALSE
);
```

Parameters

uItem

See description of *uItem* in SetMenuItemInfo in the *Platform SDK*.

lpMenuItemInfo

See description of *lpmii* in **SetMenuItemInfo** in the *Platform SDK*.

fByPos

See description of *fByPosition* in **SetMenuItemInfo** in the *Platform SDK*.

Remarks

This function wraps SetMenuItemInfo, described in the *Platform SDK*.

See Also

Reference

CMenu Class

Hierarchy Chart

Other Resources

CMenu Members

CMenu::TrackPopupMenu

Displays a floating pop-up menu at the specified location and tracks the selection of items on the pop-up menu.

```
BOOL TrackPopupMenu(
   UINT nFlags,
   int x,
   int y,
   CWnd* pWnd,
   LPCRECT lpRect = 0
);
```

Parameters

nFlags

Specifies screen-position and mouse-position flags. See TrackPopupMenu for a list of available flags.

x

Specifies the horizontal position in screen coordinates of the pop-up menu. Depending on the value of the *nFlags* parameter, the menu can be left-aligned, right-aligned, or centered relative to this position.

y

Specifies the vertical position in screen coordinates of the top of the menu on the screen.

pWnd

Identifies the window that owns the pop-up menu. This parameter cannot be **NULL**, even if the **TPM_NONOTIFY** flag is specified. This window receives all **WM_COMMAND** messages from the menu. In Windows versions 3.1 and later, the window does not receive **WM_COMMAND** messages until **TrackPopupMenu** returns. In Windows 3.0, the window receives **WM_COMMAND** messages before **TrackPopupMenu** returns.

lpRect

Ignored.

Return Value

This method returns the result of calling TrackPopupMenu in the *Platform SDK*.

Remarks

A floating pop-up menu can appear anywhere on the screen.

Example

```
// The code fragment shows how to get the File menu from the
// application window and displays it as a floating popup menu
// when the right mouse button is clicked in view.
// CMyView is a CView-derived class.
void CMyView::OnRButtonDown(UINT nFlags, CPoint point)
{
   CView::OnRButtonDown(nFlags, point);

   CMenu* menu_bar = AfxGetMainWnd()->GetMenu();
   CMenu* file_menu = menu_bar->GetSubMenu(0);
   ASSERT(file_menu);

   file_menu->TrackPopupMenu(TPM_LEFTALIGN |TPM_RIGHTBUTTON, point.x,
      point.y, this);
}
```

See Also

Reference

CMenu Class

Hierarchy Chart
CMenu::CreatePopupMenu
CMenu::GetSubMenu
TrackPopupMenu
Other Resources
CMenu Members

CMenu::TrackPopupMenuEx

Displays a floating pop-up menu at the specified location and tracks the selection of items on the pop-up menu.

```
BOOL TrackPopupMenuEx(
   UINT fuFlags,
   int x,
   int y,
   CWnd* pWnd,
   LPTPMPARAMS lptpm
);
```

Parameters

fuFlags

Specifies various functions for the extended menu. For a listing of all values and their meaning, see TrackPopupMenuEx.

x

Specifies the horizontal position in screen coordinates of the pop-up menu.

y

Specifies the vertical position in screen coordinates of the top of the menu on the screen.

pWnd

A pointer to the window owning the pop-up menu and receiving the messages from the created menu. This window can be any window from the current application but cannot be **NULL**. If you specify **TPM_NONOTIFY** in the *fuFlags* parameter, the function does not send any messages to *pWnd*. The function must return for the window pointed to by *pWnd* to receive the **WM_COMMAND** message.

lptpm

Pointer to a TPMPARAMS structure that specifies an area of the screen the menu should not overlap. This parameter can be **NULL**.

Return Value

If you specify **TPM_RETURNCMD** in the *fuFlags* parameter, the return value is the menu-item identifier of the item that the user selected. If the user cancels the menu without making a selection, or if an error occurs, then the return value is 0.

If you do not specify **TPM_RETURNCMD** in the *fuFlags* parameter, the return value is nonzero if the function succeeds and 0 if it fails. To get extended error information, call GetLastError.

Remarks

A floating pop-up menu can appear anywhere on the screen. For more information on handling errors when creating the pop-up menu, see TrackPopupMenuEx.

See Also

Reference

CMenu Class
Hierarchy Chart
CMenu::CreatePopupMenu
CMenu::GetSubMenu

Other Resources

CMenu Members

CMenu Data Members

For information about the data members in **CMenu**, see CMenu Members.

CMenu::m_hMenu

Specifies the **HMENU** handle of the Windows menu attached to the **CMenu** object.

```
HMENU m_hMenu;
```

Example

See the example for CMenu::LoadMenu.

See Also

Reference

CMenu Class

Hierarchy Chart

Other Resources

CMenu Members

CMenu Operators

For information about the operators in **CMenu**, see CMenu Members.

CMenu::operator HMENU

Use this operator to retrieve the handle of the **CMenu** object.

```
operator HMENU( ) const;
```

Return Value

If successful, the handle of the **CMenu** object; otherwise, **NULL**.

Remarks

You can use the handle to call Windows APIs directly.

See Also

Reference

CMenu Class

Hierarchy Chart

Other Resources

CMenu Members

CMenu::operator !=

Determines if two menus are logically not equal.

```
BOOL operator!=(
   const CMenu& menu
) const;
```

Parameters

menu

A **CMenu** object for comparison.

Remarks

Tests if a menu object on the left side is not equal to a menu object on the right side.

See Also

Reference

CMenu Class

Hierarchy Chart

Other Resources

CMenu Members

CMenu::operator ==

Determines if two menus are logically equal.

```
BOOL operator==(
   const CMenu& menu
) const;
```

Parameters

menu

A **CMenu** object for comparison.

Remarks

Tests if a menu object on the left side is equal (in terms of the **HMENU** value) to a menu object on the right side.

See Also

Reference

CMenu Class

Hierarchy Chart

Other Resources

CMenu Members

CMetaFileDC Class

Implements a Windows metafile, which contains a sequence of graphics device interface (GDI) commands that you can replay to create a desired image or text.

```
class CMetaFileDC : public CDC
```

Remarks

To implement a Windows metafile, first create a **CMetaFileDC** object. Invoke the **CMetaFileDC** constructor, then call the Create member function, which creates a Windows metafile device context and attaches it to the **CMetaFileDC** object.

Next send the **CMetaFileDC** object the sequence of **CDC** GDI commands that you intend for it to replay. Only those GDI commands that create output, such as **MoveTo** and **LineTo**, can be used.

After you have sent the desired commands to the metafile, call the **Close** member function, which closes the metafile device contexts and returns a metafile handle. Then dispose of the **CMetaFileDC** object.

CDC::PlayMetaFile can then use the metafile handle to play the metafile repeatedly. The metafile can also be manipulated by Windows functions such as CopyMetaFile, which copies a metafile to disk.

When the metafile is no longer needed, delete it from memory with the DeleteMetaFile Windows function.

You can also implement the **CMetaFileDC** object so that it can handle both output calls and attribute GDI calls such as **GetTextExtent**. Such a metafile is more flexible and can more easily reuse general GDI code, which often consists of a mix of output and attribute calls. The **CMetaFileDC** class inherits two device contexts, **m_hDC** and **m_hAttribDC**, from **CDC**. The **m_hDC** device context handles all CDC GDI output calls and the **m_hAttribDC** device context handles all **CDC** GDI attribute calls. Normally, these two device contexts refer to the same device. In the case of **CMetaFileDC**, the attribute DC is set to **NULL** by default.

Create a second device context that points to the screen, a printer, or device other than a metafile, then call the **SetAttribDC** member function to associate the new device context with **m_hAttribDC**. GDI calls for information will now be directed to the new **m_hAttribDC**. Output GDI calls will go to **m_hDC**, which represents the metafile.

For more information on **CMetaFileDC**, see Device Contexts.

Requirements

Header: afxext.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CDC Class

Hierarchy Chart

Other Resources

CMetaFileDC Members

CMetaFileDC Members

Base Class Members

CObject Members

CDC Members

Construction

CMetaFileDC	Constructs a **CMetaFileDC** object.

Initialization

Create	Creates the Windows metafile device context and attaches it to the **CMetaFileDC** object.
CreateEnhanced	Creates a metafile device context for an enhanced-format metafile.

Operations

Close	Closes the device context and creates a metafile handle.
CloseEnhanced	Closes an enhanced-metafile device context and creates an enhanced-metafile handle.

See Also

Reference

CMetaFileDC Class

Hierarchy Chart

CMetaFileDC Member Functions

For information about the member functions in **CMetaFileDC**, see CMetaFileDC Members.

CMetaFileDC::Close

Closes the metafile device context and creates a Windows metafile handle that can be used to play the metafile by using the CDC::PlayMetaFile member function.

```
HMETAFILE Close( );
```

Return Value

A valid **HMETAFILE** if the function is successful; otherwise **NULL**.

Remarks

The Windows metafile handle can also be used to manipulate the metafile with Windows functions such as CopyMetaFile.

Delete the metafile after use by calling the Windows DeleteMetaFile function.

See Also

Reference

CMetaFileDC Class
Hierarchy Chart
CDC::PlayMetaFile
CloseMetaFile
CopyMetaFile
DeleteMetaFile

Other Resources

CMetaFileDC Members

CMetaFileDC::CloseEnhanced

Closes an enhanced-metafile device context and returns a handle that identifies an enhanced-format metafile.

```
HENHMETAFILE CloseEnhanced( );
```

Return Value

A handle of an enhanced metafile, if successful; otherwise **NULL**.

Remarks

An application can use the enhanced-metafile handle returned by this function to perform the following tasks:

- Display a picture stored in an enhanced metafile
- Create copies of the enhanced metafile
- Enumerate, edit, or copy individual records in the enhanced metafile
- Retrieve an optional description of the metafile contents from the enhanced-metafile header
- Retrieve a copy of the enhanced-metafile header
- Retrieve a binary copy of the enhanced metafile
- Enumerate the colors in the optional palette
- Convert an enhanced-format metafile into a Windows-format metafile

When the application no longer needs the enhanced metafile handle, it should release the handle by calling the Win32 **DeleteEnhMetaFile** function.

See Also

Reference

CMetaFileDC Class
Hierarchy Chart
CDC::PlayMetaFile
CMetaFileDC::CreateEnhanced
DeleteEnhMetaFile

Other Resources

CMetaFileDC Members

CMetaFileDC::CMetaFileDC

Construct a **CMetaFileDC** object in two steps.

```
CMetaFileDC( );
```

Remarks

First, call **CMetaFileDC**, then call **Create**, which creates the Windows metafile device context and attaches it to the **CMetaFileDC** object.

See Also

Reference

CMetaFileDC Class
Hierarchy Chart
CMetaFileDC::Create

Other Resources

CMetaFileDC Members

CMetaFileDC::Create

Construct a **CMetaFileDC** object in two steps.

```
BOOL Create(
   LPCTSTR lpszFilename = NULL
);
```

Parameters

lpszFilename

Points to a null-terminated character string. Specifies the filename of the metafile to create. If *lpszFilename* is **NULL**, a new in-memory metafile is created.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

First, call the constructor **CMetaFileDC**, then call **Create**, which creates the Windows metafile device context and attaches it to the **CMetaFileDC** object.

See Also

Reference

CMetaFileDC Class
Hierarchy Chart
CMetaFileDC::CMetaFileDC
CDC::SetAttribDC
CreateMetaFile

Other Resources

CMetaFileDC Members

CMetaFileDC::CreateEnhanced

Creates a device context for an enhanced-format metafile.

```
BOOL CreateEnhanced(
   CDC* pDCRef,
   LPCTSTR lpszFileName,
   LPCRECT lpBounds,
   LPCTSTR lpszDescription
);
```

Parameters

pDCRef

Identifies a reference device for the enhanced metafile.

lpszFileName

Points to a null-terminated character string. Specifies the filename for the enhanced metafile to be created. If this parameter is **NULL**, the enhanced metafile is memory based and its contents lost when the object is destroyed or when the Win32 **DeleteEnhMetaFile** function is called.

lpBounds

Points to a RECT data structure or a CRect object that specifies the dimensions in **HIMETRIC** units (in .01-millimeter increments) of the picture to be stored in the enhanced metafile.

lpszDescription

Points to a zero-terminated string that specifies the name of the application that created the picture, as well as the picture's title.

Return Value

A handle of the device context for the enhanced metafile, if successful; otherwise **NULL**.

Remarks

This DC can be used to store a device-independent picture.

Windows uses the reference device identified by the *pDCRef* parameter to record the resolution and units of the device on which a picture originally appeared. If the *pDCRef* parameter is **NULL**, it uses the current display device for reference.

The left and top members of the **RECT** data structure pointed to by the *lpBounds* parameter must be smaller than the right and bottom members, respectively. Points along the edges of the rectangle are included in the picture. If *lpBounds* is **NULL**, the graphics device interface (GDI) computes the dimensions of the smallest rectangle that can enclose the picture drawn by the application. The *lpBounds* parameter should be supplied where possible.

The string pointed to by the *lpszDescription* parameter must contain a null character between the application name and the picture name and must terminate with two null characters —for example, "XYZ Graphics Editor\0Bald Eagle\0\0," where \0 represents the null character. If *lpszDescription* is **NULL**, there is no corresponding entry in the enhanced-metafile header.

Applications use the DC created by this function to store a graphics picture in an enhanced metafile. The handle identifying this DC can be passed to any GDI function.

After an application stores a picture in an enhanced metafile, it can display the picture on any output device by calling the **CDC::PlayMetaFile** function. When displaying the picture, Windows uses the rectangle pointed to by the *lpBounds* parameter and the resolution data from the reference device to position and scale the picture. The device context returned by this function contains the same default attributes associated with any new DC.

Applications must use the Win32 **GetWinMetaFileBits** function to convert an enhanced metafile to the older Windows metafile format.

The filename for the enhanced metafile should use the .EMF extension.

See Also

Reference

CMetaFileDC Class
Hierarchy Chart
CMetaFileDC::CloseEnhanced
CDC::PlayMetaFile
CloseEnhMetaFile
DeleteEnhMetaFile
GetEnhMetaFileDescription
GetEnhMetaFileHeader
GetWinMetaFileBits
PlayEnhMetaFile

Other Resources

CMetaFileDC Members

CMiniFrameWnd Class

Represents a half-height frame window typically seen around floating toolbars.

```
class CMiniFrameWnd : public CFrameWnd
```

Remarks

These mini-frame windows behave like normal frame windows, except that they do not have minimize/maximize buttons or menus and you only have to single-click on the system menu to dismiss them.

To use a **CMiniFrameWnd** object, first define the object. Then call the Create member function to display the mini-frame window.

For more information on how to use **CMiniFrameWnd** objects, see the article Docking and Floating Toolbars.

Requirements

Header: afxwin.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CFrameWnd Class

Hierarchy Chart

CFrameWnd Class

Other Resources

CMiniFrameWnd Members

CMiniFrameWnd Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CFrameWnd Members

Construction

CMiniFrameWnd	Constructs a **CMiniFrameWnd** object.
Create	Creates a **CMiniFrameWnd** object after construction.
CreateEx	Creates a **CMiniFrameWnd** object (with additional options) after construction.

See Also

Reference

CMiniFrameWnd Class

Hierarchy Chart

CMiniFrameWnd Member Functions

For information about the member functions in **CMiniFrameWnd**, see CMiniFrameWnd Members.

CMiniFrameWnd::CMiniFrameWnd

Constructs a **CMiniFrameWnd** object, but does not create the window.

```
CMiniFrameWnd( );
```

Remarks

To create the window, call CMiniFrameWnd::Create.

See Also

Reference

CMiniFrameWnd Class
Hierarchy Chart
CFrameWnd Class

Other Resources

CMiniFrameWnd Members

CMiniFrameWnd::Create

Creates the Windows mini-frame window and attaches it to the **CMiniFrameWnd** object.

```
virtual BOOL Create(
   LPCTSTR lpClassName,
   LPCTSTR lpWindowName,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd = NULL,
   UINT nID = 0
);
```

Parameters

lpClassName

Points to a null-terminated character string that names the Windows class. The class name can be any name registered with the global AfxRegisterWndClass function. If **NULL**, the window class will be registered for you by the framework. MFC gives the default class the following styles and attributes:

- Sets style bit **CS_DBLCLKS**, which sends double-click messages to the window procedure when the user double-clicks the mouse.
- Sets style bits **CS_HREDRAW** and **CS_VREDRAW**, which direct the contents of the client area to be redrawn when the window changes size.
- Sets the class cursor to the Windows standard **IDC_ARROW**.
- Sets the class background brush to **NULL**, so the window will not erase its background.
- Sets the class icon to the standard, waving-flag Windows logo icon.
- Sets the window to the default size and position, as indicated by Windows.

lpWindowName

Points to a null-terminated character string that contains the window name.

dwStyle

Specifies the window style attributes. These can include standard window styles and one or more of the following special styles:

- **MFS_MOVEFRAME** Allows the mini-frame window to be moved by clicking on any edge of the window, not just the caption.
- **MFS_4THICKFRAME** Disables resizing of the mini-frame window.
- **MFS_SYNCACTIVE** Synchronizes the activation of the mini-frame window to the activation of its parent window.
- **MFS_THICKFRAME** Allows the mini-frame window to be sized as small as the contents of the client area allow.
- **MFS_BLOCKSYSMENU** Disables access to the system menu and the control menu, and converts them to part of the caption (title bar).

See CWnd::Create for a description of possible window style values. The typical combination used for mini-frame windows is **WS_POPUP|WS_CAPTION|WS_SYSMENU**.

rect

A **RECT** structure specifying the desired dimensions of the window.

pParentWnd

Points to the parent window. Use **NULL** for top-level windows.

nID

If the mini-frame window is created as a child window, this is the identifier of the child control; otherwise 0.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Create initializes the window's class name and window name and registers default values for its style and parent.

See Also

Reference

CMiniFrameWnd Class
Hierarchy Chart
CFrameWnd::Create
CWnd::Create
CWnd::CreateEx
CFrameWnd Class

Other Resources

CMiniFrameWnd Members

CMiniFrameWnd::CreateEx

Creates a **CMiniFrameWnd** object (with additional options) after construction.

```
virtual BOOL CreateEx(
   DWORD dwExStyle,
   LPCTSTR lpClassName,
   LPCTSTR lpWindowName,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd = NULL,
   UINT nID = 0
);
```

Parameters

dwExStyle

Specifies the extended style of the **CMiniFrameWnd** being created. Apply any of the extended window styles to the window.

lpClassName

Points to a null-terminated character string that names the Windows class (a WNDCLASS structure). The class name can be any name registered with the global AfxRegisterWndClass function or any of the predefined control-class names. It must not be **NULL**.

lpWindowName

Points to a null-terminated character string that contains the window name.

dwStyle

Specifies the window style attributes. See Window Styles and CWnd::Create for a description of the possible values.

rect

The size and position of the window, in client coordinates of *pParentWnd*.

pParentWnd

Points to the parent window object.

nID

The identifier of the child window.

Return Value

Returns TRUE on success, FALSE on failure.

Remarks

The **CreateEx** parameters specify the **WNDCLASS**, window style, and (optionally) initial position and size of the window. **CreateEx** also specifies the window's parent (if any) and ID.

When **CreateEx** executes, Windows sends the WM_GETMINMAXINFO, WM_NCCREATE, WM_NCCALCSIZE, and WM_CREATE messages to the window.

To extend the default message handling, derive a class from **CMiniFrameWnd**, add a message map to the new class, and provide member functions for the above messages. Override **OnCreate**, for example, to perform needed initialization for a new class.

Override further **On***Message* message handlers to add further functionality to your derived class.

If the **WS_VISIBLE** style is given, Windows sends the window all the messages required to activate and show the window. If the window style specifies a title bar, the window title pointed to by the *lpszWindowName* parameter is displayed in the title bar.

The *dwStyle* parameter can be any combination of window styles.

The old style Palette toolbox windows are no longer supported. The old style, which did not have an "X" Close button, was

supported when running an MFC application on previous versions of Windows, but is no longer supported in Visual C++.NET. Only the new **WS_EX_TOOLWINDOW** style is now supported; for a description of this style, see Extended Window Styles.

See Also

Reference

CMiniFrameWnd Class
Hierarchy Chart
CFrameWnd::Create
CWnd::Create
CWnd::CreateEx
CFrameWnd Class

Other Resources

CMiniFrameWnd Members

CMonikerFile Class

Represents a stream of data (IStream) named by an IMoniker.

```
class CMonikerFile : public COleStreamFile
```

Remarks

A moniker contains information much like a pathname to a file. If you have a pointer to a moniker object's **IMoniker** interface, you can get access to the identified file without having any other specific information about where the file is actually located.

Derived from **COleStreamFile**, **CMonikerFile** takes a moniker or a string representation it can make into a moniker and binds to the stream for which the moniker is a name. You can then read and write to that stream. The real purpose of **CMonikerFile** is to provide simple access to **IStream**s named by **IMoniker**s so that you do not have to bind to a stream yourself, yet have **CFile** functionality to the stream.

CMonikerFile cannot be used to bind to anything other than a stream. If you want to bind to storage or an object, you must use the **IMoniker** interface directly.

For more information on streams and monikers, see COleStreamFile in the *MFC Reference* and IStream and IMoniker in the *Platform SDK*.

Requirements

Header: afxole.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

COleStreamFile Class
Hierarchy Chart
CAsyncMonikerFile Class

Other Resources

CMonikerFile Members

CMonikerFile Members

Base Class Members

CObject Members

CFile Members

COleStreamFile Members

Construction

CMonikerFile	Constructs a **CMonikerFile** object.

Operations

Close	Detaches and releases the stream and releases the moniker.
Detach	Detaches the **IMoniker** from this **CMonikerFile** object.
GetMoniker	Returns the current moniker.

Overridables

CreateBindContext	Obtains the bind context or creates a default initialized bind context.
Open	Opens the specified file to obtain a stream.

See Also

Reference

CMonikerFile Class

Hierarchy Chart

CMonikerFile Member Functions

For information about the member functions in **CMonikerFile**, see CMonikerFile Members.

CMonikerFile::Close

Call this function to detach and release the stream and to release the moniker.

```
virtual void Close( );
```

Remarks

Can be called on unopened or already closed streams.

See Also

Reference

CMonikerFile Class
Hierarchy Chart
CMonikerFile::Open

Other Resources

CMonikerFile Members

CMonikerFile::CMonikerFile

Constructs a **CMonikerFile** object.

```
CMonikerFile( );
```

See Also

Reference

CMonikerFile Class
Hierarchy Chart
CAsyncMonikerFile Class
CMonikerFile::Open

Other Resources

CMonikerFile Members

CMonikerFile::CreateBindContext

Call this function to create a default initialized bind context.

```
IBindCtx* CreateBindContext(
   CFileException* pError
);
```

Parameters

pError

A pointer to a file exception. In the event of an error, it will be set to the cause.

Return Value

A pointer to the bind context IBindCtx to bind with if successful; otherwise **NULL**. If the instance was opened with an **IBindHost** interface, the bind context is retrieved from the **IBindHost**. If there is no **IBindHost** interface or the interface fails to return a bind context, a bind context is created. For a description of the IBindHost interface, see the *Platform SDK*.

Remarks

A bind context is an object that stores information about a particular moniker binding operation. You can override this function to provide a custom bind context.

See Also

Reference

CMonikerFile Class

Hierarchy Chart

Other Resources

CMonikerFile Members

CMonikerFile::Detach

Call this function to close the stream.

```
BOOL Detach(
   CFileException* pError = NULL
);
```

Parameters

pError

A pointer to a file exception. In the event of an error, it will be set to the cause.

Return Value

Nonzero if successful; otherwise 0.

See Also

Reference

CMonikerFile Class
Hierarchy Chart
CMonikerFile::Close
CMonikerFile::Open

Other Resources

CMonikerFile Members

CMonikerFile::GetMoniker

Call this function to retrieve a pointer to the current moniker.

```
IMoniker* GetMoniker( ) const;
```

Return Value

A pointer to the current moniker interface (IMoniker).

Remarks

Since **CMonikerFile** is not an interface, the pointer returned does not increment the reference count (through AddRef), and the moniker is released when the **CMonikerFile** object is released. If you want to hold onto the moniker or release it yourself, you must **AddRef** it.

See Also

Reference

CMonikerFile Class

Hierarchy Chart

Other Resources

CMonikerFile Members

CMonikerFile::Open

Call this member function to open a file or moniker object.

```
virtual BOOL Open(
   LPCTSTR lpszURL,
   CFileException* pError = NULL
);
virtual BOOL Open(
   IMoniker* pMoniker,
   CFileException* pError = NULL
);
```

Parameters

lpszURL

A URL or filename of the file to be opened.

pError

A pointer to a file exception. In the event of an error, it will be set to the cause.

pMoniker

A pointer to the moniker interface **IMoniker** to be used to obtain a stream.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The *lpszURL* parameter cannot be used on a Macintosh. Only the *pMoniker* form of **Open** can be used on a Macintosh.

You can use a URL or a filename for the *lpszURL* parameter. For example:

```
CMyMonFile  mymonf;
mymonf.Open(_T("http://www.microsoft.com"));
```

– or –

```
CMyMonFile  mymonf;
mymonf.Open(_T("file:c:\mydata.dat"));
```

See Also

Reference

CMonikerFile Class
Hierarchy Chart
CMonikerFile::CMonikerFile
CAsyncMonikerFile::Open

Other Resources

CMonikerFile Members

CMonthCalCtrl Class

Encapsulates the functionality of a month calendar control.

```
class CMonthCalCtrl : public CWnd
```

Remarks

The month calendar control provides the user with a simple calendar interface, from which the user can select a date. The user can change the display by:

- Scrolling backward and forward, from month to month.
- Clicking the Today text to display the current day (if the **MCS_NOTODAY** style is not used).
- Picking a month or a year from a pop-up menu.

You can customize the month calendar control by applying a variety of styles to the object when you create it. These styles are described in Month Calendar Control Styles in the *Platform SDK*.

The month calendar control can display more than one month, and it can indicate special days (such as holidays) by bolding the date.

For more information on using the month calendar control, see Using CMonthCalCtrl.

Requirements

Header: afxdtctl.h

See Also

Tasks

CMNCTRL1 Sample: Demonstrates Common Control MFC Classes, Part 1

Reference

CWnd Class

Hierarchy Chart

CDateTimeCtrl Class

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Construction

CMonthCalCtrl	Constructs a **CMonthCalCtrl** object.
Create	Creates a month calendar control and attaches it to the **CMonthCalCtrl** object.

Attributes

GetColor	Gets the color of a specified area of a month calendar control.
GetFirstDayOfWeek	Gets the first day of the week to be displayed in the leftmost column of the calendar.
GetMinReqRect	Retrieves the minimum size required to show a full month in a month calendar control.
GetMonthDelta	Retrieves the scroll rate for a month calendar control.
SetColor	Sets the color of a specified area of a month calendar control.
SetFirstDayOfWeek	Sets the day of week to be displayed in the leftmost column of the calendar.
SetMonthDelta	Sets the scroll rate for a month calendar control.

Operations

GetCurSel	Retrieves the system time as indicated by the currently-selected date.
GetMaxSelCount	Retrieves the current maximum number of days that can be selected in a month calendar control.
GetMonthRange	Retrieves date information representing the high and low limits of a month calendar control's display.
GetRange	Retrieves the current minimum and maximum dates set in a month calendar control.
GetSelRange	Retrieves date information representing the upper and lower limits of the date range currently selected by th e user.
GetToday	Retrieves the date information for the date specified as "today" for a month calendar control.
HitTest	Determines which portion of a month calendar control is at a given point on the screen.
SetCurSel	Sets the currently selected date for a month calendar control.
SetDayState	Sets the display for days in a month calendar control.
SetMaxSelCount	Sets the maximum number of days that can be selected in a month calendar control.
SetRange	Sets the minimum and maximum allowable dates for a month calendar control.

SetSelRange	Sets the selection for a month calendar control to a given date range.
SetToday	Sets the calendar control for the current day.
SizeMinReq	Repaints the month calendar control to its minimum, one-month size.

See Also

Reference

CMonthCalCtrl Class

Hierarchy Chart

CMonthCalCtrl Member Functions

For information about the member functions in **CMonthCalCtrl**, see CMonthCalCtrl Members.

CMonthCalCtrl::CMonthCalCtrl

Constructs a **CMonthCalCtrl** object.

```
CMonthCalCtrl( );
```

Remarks

You must call **Create** after you construct the object.

See Also

Reference

CMonthCalCtrl Class
Hierarchy Chart
CMonthCalCtrl::Create

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::Create

Creates a month calendar control and attaches it to the **CMonthCalCtrl** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
virtual BOOL Create(
   DWORD dwStyle,
   const POINT& pt,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the combination of Windows styles applied to the month calendar control. See Month Calendar Control Styles in the *Platform SDK* for more information about the styles.

rect

A reference to a RECT structure. Contains the position and size of the month calendar control.

pt

A reference to a POINT structure that identifies the location of the month calendar control.

pParentWnd

A pointer to a CWnd object that is the parent window of the month calendar control. It must not be **NULL**.

nID

Specifies the month calendar control's control ID.

Return Value

Nonzero if initialization was successful; otherwise 0.

Remarks

Create a month calendar control in two steps:

1. Call CMonthCalCtrl to construct a **CMonthCalCtrl** object.
2. Call this member function, which creates a month calendar control and attaches it to the **CMonthCalCtrl** object.

When you call **Create**, the common controls are initialized. The version of **Create** you call determines how it is sized:

- To have MFC automatically size the control to one month, call the override that uses the *pt* parameter.
- To size the control yourself, call the override of this function that uses the *rect* parameter.

Example

```
void CMyDialog::OnCreateControls()
{
   // Given two member objects m_calCtrl1 and m_calCtrl2, we can
   // create them in one of two ways.

   // Providing a point has the control with its top-left corner
   // at that point and sized automatically to show one month
   // page.
```

```
    CPoint pt(10, 10);
    VERIFY(m_calCtrl1.Create(WS_TABSTOP | WS_CHILD | WS_VISIBLE | WS_BORDER,
       pt, this, 1005));

    // Providing a rectangle lets us completely control the size.
    // The control will paint as many complete month pages in the
    // control's area as possible.

    CRect rect(10, 100, 170, 200);
    VERIFY(m_calCtrl2.Create(WS_TABSTOP | WS_CHILD | WS_VISIBLE | WS_BORDER,
       rect, this, 1006));
 }
```

See Also

Reference

CMonthCalCtrl Class

Hierarchy Chart

CMonthCalCtrl::CMonthCalCtrl

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::GetColor

Retrieves the color of an area of the month calendar control specified by *nRegion*.

```
COLORREF GetColor(
   int nRegion
) const;
```

Parameters

nRegion

The region of the month calendar control from which the color is retrieved. For a list of values, see the *nRegion* parameter of SetColor.

Return Value

A COLORREF value specifying the color associated with the portion of the month calendar control, if successful. Otherwise, this member function returns -1.

See Also

Reference

CMonthCalCtrl Class
Hierarchy Chart
CMonthCalCtrl::SetColor

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::GetCurSel

Retrieves the system time as indicated by the currently-selected date.

```
BOOL GetCurSel(
   COleDateTime& refDateTime
) const;
BOOL GetCurSel(
   CTime& refDateTime
) const;
BOOL GetCurSel(
   LPSYSTEMTIME pDateTime
) const;
```

Parameters

refDateTime

A reference to a COleDateTime object or a CTime object. Receives the current time.

pDateTime

A pointer to a SYSTEMTIME structure that will receive the currently-selected date information. This parameter must be a valid address and cannot be **NULL**.

Return Value

Nonzero if successful; otherwize 0.

Remarks

This member function implements the behavior of the Win32 message MCM_GETCURSEL, as described in the *Platform SDK*.

Note
This member function fails if the style **MCS_MULTISELECT** is set.

In MFC's implementation of **GetCurSel**, you can specify a **COleDateTime** usage, a **CTime** usage, or a **SYSTEMTIME** structure usage.

See Also

Reference

CMonthCalCtrl Class
Hierarchy Chart
CMonthCalCtrl::SetCurSel

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::GetFirstDayOfWeek

Gets the first day of the week to be displayed in the leftmost column of the calendar.

```
int GetFirstDayOfWeek(
   BOOL* pbLocal = NULL
) const;
```

Parameters

pbLocal

A pointer to a **BOOL** value. If the value is non-zero, the control's setting does not match the setting in the control panel.

Return Value

An integer value that represents the first day of the week. See **Remarks** for more information on what these integers represent.

Remarks

This member function implements the behavior of the Win32 message MCM_GETFIRSTDAYOFWEEK, as described in the *Platform SDK*. The days of the week are represented as integers, as follows.

Value	Day of the week
0	Monday
1	Tuesday
2	Wednesday
3	Thursday
4	Friday
5	Saturday
6	Sunday

Example

See the example for CMonthCalCtrl::SetFirstDayOfWeek.

See Also

Reference

CMonthCalCtrl Class

Hierarchy Chart

CMonthCalCtrl::SetFirstDayOfWeek

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::GetMaxSelCount

Retrieves the current maximum number of days that can be selected in a month calendar control.

```
int GetMaxSelCount( ) const;
```

Return Value

An integer value that represents the total number of days that can be selected for the control.

Remarks

This member function implements the behavior of the Win32 message MCM_GETMAXSELCOUNT, as described in the *Platform SDK*. Use this member function for controls with the **MCS_MULTISELECT** style set.

Example

See the example for CMonthCalCtrl::SetMaxSelCount.

See Also

Reference

CMonthCalCtrl Class
Hierarchy Chart
CMonthCalCtrl::SetMaxSelCount

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::GetMinReqRect

Retrieves the minimum size required to show a full month in a month calendar control.

```
BOOL GetMinReqRect(
   RECT* pRect
) const;
```

Parameters

pRect

A pointer to a RECT structure that will receive bounding rectangle information. This parameter must be a valid address and cannot be **NULL**.

Return Value

If successful, this member function returns nonzero and *lpRect* receives the applicable bounding information. If unsuccessful, the member function returns 0.

Remarks

This member function implements the behavior of the Win32 message MCM_GETMINREQRECT, as described in the *Platform SDK*.

See Also

Reference

CMonthCalCtrl Class
Hierarchy Chart
CMonthCalCtrl::SizeMinReq

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::GetMonthDelta

Retrieves the scroll rate for a month calendar control.

```
int GetMonthDelta( ) const;
```

Return Value

The scroll rate for the month calendar control. The scroll rate is the number of months that the control moves its display when the user clicks a scroll button once.

Remarks

This member function implements the behavior of the Win32 message MCM_GETMONTHDELTA, as described in the *Platform SDK*.

See Also

Reference

CMonthCalCtrl Class
Hierarchy Chart
CMonthCalCtrl::SetMonthDelta

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::GetMonthRange

Retrieves date information representing the high and low limits of a month calendar control's display.

```
int GetMonthRange(
   COleDateTime& refMinRange,
   COleDateTime& refMaxRange,
   DWORD dwFlags
) const;
int GetMonthRange(
   CTime& refMinRange,
   CTime& refMaxRange,
   DWORD dwFlags
) const;
int GetMonthRange(
   LPSYSTEMTIME pMinRange,
   LPSYSTEMTIME pMaxRange,
   DWORD dwFlags
) const;
```

Parameters

refMinRange

A reference to a COleDateTime or CTime object containing the minimum date allowed.

refMaxRange

A reference to a **COleDateTime** or **CTime** object containing the maximum date allowed.

pMinRange

A pointer to a SYSTEMTIME structure containing the date at the lowest end of the range.

pMaxRange

A pointer to a **SYSTEMTIME** structure containing the date at the highest end of the range.

dwFlags

Value specifying the scope of the range limits to be retrieved. This value must be one of the following.

Value	Meaning
GMR_DAYSTATE	Include preceding and trailing months of visible range that are only partially displayed.
GMR_VISIBLE	Include only those months that are entirely displayed.

Return Value

An integer that represents the range, in months, spanned by the two limits indicated by *refMinRange* and *refMaxRange* in the first and second versions, or *pMinRange* and *pMaxRange* in the third version.

Remarks

This member function implements the behavior of the Win32 message MCM_GETMONTHRANGE, as described in the *Platform SDK*. In MFC's implementation of **GetMonthRange**, you can specify **COleDateTime** usage, a **CTime** usage, or a **SYSTEMTIME** structure usage.

Example

See the example for CMonthCalCtrl::SetDayState.

See Also

Reference

CMonthCalCtrl Class

Hierarchy Chart

CMonthCalCtrl::GetRange

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::GetRange

Retrieves the current minimum and maximum dates set in a month calendar control.

```
DWORD GetRange(
   COleDateTime* pMinRange,
   COleDateTime* pMaxRange
) const;
DWORD GetRange(
   CTime* pMinRange,
   CTime* pMaxRange
) const;
DWORD GetRange(
   LPSYSTEMTIME pMinRange,
   LPSYSTEMTIME pMaxRange
) const;
```

Parameters

pMinRange

A pointer to a **COleDateTime** object, a **CTime** object, or SYSTEMTIME structure containing the date at the lowest end of the range.

pMaxRange

A pointer to a **COleDateTime** object, a **CTime** object, or SYSTEMTIME structure containing the date at the highest end of the range.

Return Value

A **DWORD** that can be zero (no limits are set) or a combination of the following values that specify limit information.

Value	Meaning
GDTR_MAX	A maximum limit is set for the control; *pMaxRange* is valid and contains the applicable date information.
GDTR_MIN	A minimum limit is set for the control; *pMinRange* is valid and contains the applicable date information.

Remarks

This member function implements the behavior of the Win32 message MCM_GETRANGE, as described in the *Platform SDK*. In MFC's implementation of **GetRange**, you can specify a **COleDateTime** usage, a **CTime** usage, or a **SYSTEMTIME** structure usage.

Example

```
// This code fragment sets a variety of ranges in the
// control, and calls a separate function to show the
// set range to the user.

void CDatesDlg::OnButton2()
{
   // Gain a pointer to the control
   CMonthCalCtrl* pCtrl = (CMonthCalCtrl*) GetDlgItem(IDC_MONTHCALENDAR1);
   ASSERT(pCtrl != NULL);

   // set minimum of January 1st, 1995 with no maximum
   COleDateTime dtMin;
   COleDateTime dtMax;

   dtMin = COleDateTime(1995, 1, 1, 0, 0, 0);
   dtMax.SetStatus(COleDateTime::null);
   pCtrl->SetRange(&dtMin, &dtMax);
   ShowRange(pCtrl);
```

```
    // set no minimum and a maximum of September 30th, 1997
    dtMin.SetStatus(COleDateTime::null);
    dtMax = COleDateTime(1997, 9, 30, 0, 0, 0);
    pCtrl->SetRange(&dtMin, &dtMax);
    ShowRange(pCtrl);

    // set minimum of April 15, 1992 and maximum of June 5, 2002
    dtMin = COleDateTime(1992, 4, 15, 0, 0, 0);
    dtMax = COleDateTime(2002, 6, 5, 0, 0, 0);
    pCtrl->SetRange(&dtMin, &dtMax);
    ShowRange(pCtrl);
}

void CDatesDlg::ShowRange(CMonthCalCtrl* pCtrl)
{
    ASSERT(pCtrl != NULL);
    CString strMessage;
    COleDateTime dtMinimum;
    COleDateTime dtMaximum;

    // Get the range
    DWORD dwResult = pCtrl->GetRange(&dtMinimum, &dtMaximum);

    // If a minimum was specified, format it
    // otherwise, indicate that there is no lower bound
    if (dwResult & GDTR_MIN)
        strMessage += dtMinimum.Format(_T("Minimum range is %x %X.\r\n"));
    else
        strMessage += _T("No minimum range.\r\n");

    // Treat maximum similarly
    if (dwResult & GDTR_MAX)
        strMessage += dtMaximum.Format(_T("Maximum range is %x %X.\r\n"));
    else
        strMessage += _T("No maximum range.\r\n");

    // Show the user
    AfxMessageBox(strMessage);
}
```

See Also

Reference

CMonthCalCtrl Class

Hierarchy Chart

CMonthCalCtrl::SetRange

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::GetSelRange

Retrieves date information representing the upper and lower limits of the date range currently selected by the user.

```
BOOL GetSelRange(
   COleDateTime& refMinRange,
   COleDateTime& refMaxRange
) const;
BOOL GetSelRange(
   CTime& refMinRange,
   CTime& refMaxRange
) const;
BOOL GetSelRange(
   LPSYSTEMTIME pMinRange,
   LPSYSTEMTIME pMaxRange
) const;
```

Parameters

refMinRange

A reference to a COleDateTime or CTime object containing the minimum date allowed.

refMaxRange

A reference to a **COleDateTime** or **CTime** object containing the maximum date allowed.

pMinRange

A pointer to a SYSTEMTIME structure containing the date at the lowest end of the range.

pMaxRange

A pointer to a **SYSTEMTIME** structure containing the date at the highest end of the range.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function implements the behavior of the Win32 message MCM_GETSELRANGE, as described in the *Platform SDK*. **GetSelRange** will fail if applied to a month calendar control that does not use the **MCS_MULTISELECT** style.

In MFC's implementation of **GetSelRange**, you can specify **COleDateTime** usage, a **CTime** usage, or a **SYSTEMTIME** structure usage.

See Also

Reference

CMonthCalCtrl Class
Hierarchy Chart
CMonthCalCtrl::SetSelRange

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::GetToday

Retrieves the date information for the date specified as "today" for a month calendar control.

```
BOOL GetToday(
   COleDateTime& refDateTime
) const;
BOOL GetToday(
   COleDateTime& refDateTime
) const;
BOOL GetToday(
   LPSYSTEMTIME pDateTime
) const;
```

Parameters

refDateTime

A reference to a COleDateTime or CTime object indicating the current day.

pDateTime

A pointer to a SYSTEMTIME structure that will receive the date information. This parameter must be a valid address and cannot be **NULL**.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function implements the behavior of the Win32 message MCM_GETTODAY, as described in the *Platform SDK*. In MFC's implementation of **GetToday**, you can specify a **COleDateTime** usage, a **CTime** usage, or a **SYSTEMTIME** structure usage.

Example

```
void CDatesDlg::OnButton1()
{
   // Gain a pointer to the control
   CMonthCalCtrl* pCtrl = (CMonthCalCtrl*) GetDlgItem(IDC_MONTHCALENDAR1);

   COleDateTime timeToday;
   if (pCtrl->GetToday(timeToday))
   {
      // Format the date information from the value we received
      // and post a message box about it.
      CString str = timeToday.Format(VAR_DATEVALUEONLY);
      AfxMessageBox(str);

      // Set the control's "today" indicator to be five
      // days previous.
      timeToday -= 5;
      pCtrl->SetToday(timeToday);
   }
   else
   {
      // Something is wrong!
      ASSERT(FALSE);
   }
}
```

See Also

Reference

CMonthCalCtrl Class

Hierarchy Chart

CMonthCalCtrl::SetToday

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::HitTest

Determines which month calendar control, if any, is at a specified position.

```
DWORD HitTest(
   PMCHITTESTINFO pMCHitTest
);
```

Parameters

pMCHitTest

A pointer to a MCHITTESTINFO structure containing hit testing points for the month calendar control.

Return Value

A **DWORD** value. Equal to the **uHit** member of the **MCHITTESTINFO** structure.

Remarks

HitTest uses the **MCHITTESTINFO** structure, which contains information about the hit test.

See Also

Reference

CMonthCalCtrl Class

Hierarchy Chart

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::SetColor

Sets the color of a specified area of a month calendar control.

```
COLORREF SetColor(
   int nRegion,
   COLORREF ref
);
```

Parameters

nRegion

An integer value specifying which month calendar color to set. This value can be one of the following.

Value	Meaning
MCSC_BACKGROUND	The background color displayed between months.
MCSC_MONTHBK	The background color displayed within the month.
MCSC_TEXT	The color used to display text within a month.
MCSC_TITLEBK	The background color displayed in the calendar's title.
MCSC_TITLETEXT	The color used to display text within the calendar's title.
MCSC_TRAILINGTEXT	The color used to display header and trailing-day text. Header and trailing days are the days from the previous and following months that appear on the current calendar.

ref

A **COLORREF** value for the new color setting for the specified portion of the month calendar control.

Return Value

A **COLORREF** value that represents the previous color setting for the specified portion of the month calendar control, if successful. Otherwise this message returns -1.

Remarks

This member function implements the behavior of the Win32 message MCM_SETCOLOR, as described in the *Platform SDK*.

Example

```
void CDatesDlg::OnButton2()
{
   // Gain a pointer to the control
   CMonthCalCtrl* pCtrl = (CMonthCalCtrl*) GetDlgItem(IDC_MONTHCALENDAR1);

   // Set colors for title text and title background to match
   // the Control Panel settings for inactive window captions.
   pCtrl->SetColor(MCSC_TITLETEXT, ::GetSysColor(COLOR_INACTIVECAPTIONTEXT));
   pCtrl->SetColor(MCSC_TITLEBK, ::GetSysColor(COLOR_INACTIVECAPTION));
}
```

See Also

Reference

CMonthCalCtrl Class

Hierarchy Chart
CMonthCalCtrl::GetColor
Other Resources
CMonthCalCtrl Members

CMonthCalCtrl::SetCurSel

Sets the currently selected date for a month calendar control.

```
BOOL SetCurSel(
   const COleDateTime& refDateTime
);
BOOL SetCurSel(
   const CTime& refDateTime
);
BOOL SetCurSel(
   const LPSYSTEMTIME pDateTime
);
```

Parameters

refDateTime

A reference to a COleDateTime or CTime object indicating the currently-selected month calendar control.

pDateTime

Pointer to a SYSTEMTIME structure that contains the date to be set as the current selection.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function implements the behavior of the Win32 message MCM_SETCURSEL, as described in the *Platform SDK*. In MFC's implementation of **SetCurSel**, you can specify a **COleDateTime** usage, a **CTime** usage, or a **SYSTEMTIME** structure usage.

Example

```
// All of these calls set the current selection to March 15, 1998.

void CDatesDlg::OnButton2()
{
   // Gain a pointer to the control
   CMonthCalCtrl* pCtrl = (CMonthCalCtrl*) GetDlgItem(IDC_MONTHCALENDAR1);

   // with a COleDateTime
   COleDateTime dt1(1998, 3, 15, 0, 0, 0);
   pCtrl->SetCurSel(dt1);

   // with a CTime
   CTime dt2(1998, 3, 15, 0, 0, 0);
   pCtrl->SetCurSel(dt2);

   // with a SYSTEMTIME structure

   SYSTEMTIME sysTime;

   // set everything to zero
   memset(&sysTime, 0, sizeof(sysTime));

   // except for the date we want
   sysTime.wYear = 1998;
   sysTime.wMonth = 3;
   sysTime.wDay = 15;

   pCtrl->SetCurSel(&sysTime);
}
```

See Also

Reference

CMonthCalCtrl Class

Hierarchy Chart

CMonthCalCtrl::GetCurSel

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::SetDayState

Sets the display for days in a month calendar control.

```
BOOL SetDayState(
   int nMonths,
   LPMONTHDAYSTATE pStates
);
```

Parameters

nMonths

Value indicating how many elements are in the array that *pStates* points to.

pStates

A pointer to a MONTHDAYSTATE array of values that define how the month calendar control will draw each day in its display. The **MONTHDAYSTATE** data type is a bit field, where each bit (1 through 31) represents the state of a day in a month. If a bit is on, the corresponding day will be displayed in bold; otherwise it will be displayed with no emphasis.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function implements the behavior of the Win32 message MCM_SETDAYSTATE, as described in the *Platform SDK*.

Example

```
void CDatesDlg::OnButton2()
{
   // Gain a pointer to the control

   CMonthCalCtrl* pCtrl = (CMonthCalCtrl*) GetDlgItem(IDC_MONTHCALENDAR1);
   ASSERT(pCtrl != NULL);

   // First, we must find the visible range. The array we pass to the
   // SetDayState() function must be large enough to hold days for all
   // of the visible months. Even if a month is _partially_ visible,
   // we must have MONTHDAYSTATE data for it in the array we pass.
   // GetMonthRange() returns the range of days currently visible in
   // the control, along with a count of visible months. This array
   // will be up to 2 months larger than the number of "pages" visible
   // in the control.

   SYSTEMTIME timeFrom;
   SYSTEMTIME timeUntil;
   int nCount = pCtrl->GetMonthRange(&timeFrom, &timeUntil, GMR_DAYSTATE);

   // Allocate the state array based on the return value.

   LPMONTHDAYSTATE pDayState;
   pDayState = new MONTHDAYSTATE[nCount];
   memset(pDayState, 0, sizeof(MONTHDAYSTATE) * nCount);

   // Find the first fully visible month.

   int nIndex = (timeFrom.wDay == 1) ? 0 : 1;

   // Set the 4th day, 19th day, and 26th day of the first
   // _fully_ visible month as bold.

   pDayState[nIndex] |= 1 << 3;   // 4th day
   pDayState[nIndex] |= 1 << 18;   // 19th day
   pDayState[nIndex] |= 1 << 25;   // 25th day
```

```
    // Set state and clean up

    VERIFY(pCtrl->SetDayState(nCount, pDayState));
    delete [] pDayState;
}
```

See Also

Reference

CMonthCalCtrl Class

Hierarchy Chart

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::SetFirstDayOfWeek

Sets the day of week to be displayed in the leftmost column of the calendar.

```
BOOL SetFirstDayOfWeek(
   int iDay,
   int* lpnOld = NULL
);
```

Parameters

iDay

An integer value representing which day is to be set as the first day of the week. This value must be one of the day numbers. See GetFirstDayOfWeek for a description of the day numbers.

lpnOld

A pointer to an integer indicating the first day of the week previously set.

Return Value

Nonzero if the previous first day of the week is set to a value other than that of **LOCALE_IFIRSTDAYOFWEEK**, which is the day indicated in the control panel setting. Otherwise, this function returns 0.

Remarks

This member function implements the behavior of the Win32 message MCM_SETFIRSTDAYOFWEEK, as described in the *Platform SDK*.

Example

```
void CDatesDlg::OnButton2()
{
   // This work isn't normally necessary, since the control will set
   // the day of the week to match the system locale by itself.

   // Ask the system for the first day of the week
   TCHAR sz[2];
   BOOL bResult = GetLocaleInfo(LOCALE_SYSTEM_DEFAULT,
      LOCALE_IFIRSTDAYOFWEEK, sz, 2);

   // Convert from string result
   int nFirstDay = _ttoi(sz);

   // Gain a pointer to the control
   CMonthCalCtrl* pCtrl = (CMonthCalCtrl*) GetDlgItem(IDC_MONTHCALENDAR1);

   // Set it and assert that it was successful.
   pCtrl->SetFirstDayOfWeek(nFirstDay);
   ASSERT(pCtrl->GetFirstDayOfWeek() == nFirstDay);
}
```

See Also

Reference

CMonthCalCtrl Class

Hierarchy Chart

CMonthCalCtrl::GetFirstDayOfWeek

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::SetMaxSelCount

Sets the maximum number of days that can be selected in a month calendar control.

```
BOOL SetMaxSelCount(
   int nMax
);
```

Parameters

nMax

The value that will be set to represent the maximum number of selectable days.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function implements the behavior of the Win32 message MCM_SETMAXSELCOUNT, as described in the *Platform SDK*.

Example

```
// The control needs to have the MCS_MULTISELECT style
// for the following code to work.
void CDatesDlg::OnButton2()
{
   // Gain a pointer to the control
   CMonthCalCtrl* pCtrl = (CMonthCalCtrl*) GetDlgItem(IDC_MONTHCALENDAR1);

   // change the maximum selection count
   pCtrl->SetMaxSelCount(10);

   // check that the change was really made
   ASSERT(pCtrl->GetMaxSelCount() == 10);
}
```

See Also

Reference

CMonthCalCtrl Class
Hierarchy Chart
CMonthCalCtrl::GetMaxSelCount

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::SetMonthDelta

Sets the scroll rate for a month calendar control.

```
int SetMonthDelta(
   int iDelta
);
```

Parameters

iDelta

The number of months to be set as the control's scroll rate. If this value is zero, the month delta is reset to the default, which is the number of months displayed in the control.

Return Value

The previous scroll rate. If the scroll rate has not been previously set, the return value is 0.

Remarks

This member function implements the behavior of the Win32 message MCM_SETMONTHDELTA, as described in the *Platform SDK*.

See Also

Reference

CMonthCalCtrl Class
Hierarchy Chart
CMonthCalCtrl::GetMonthDelta

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::SetRange

Sets the minimum and maximum allowable dates for a month calendar control.

```
BOOL SetRange(
   const COleDateTime* pMinRange,
   const COleDateTime* pMaxRange
);
BOOL SetRange(
   const CTime* pMinRange,
   const CTime* pMaxRange
);
BOOL SetRange(
   const LPSYSTEMTIME pMinRange,
   const LPSYSTEMTIME pMaxRange
);
```

Parameters

pMinRange

A pointer to a **COleDateTime** object, a **CTime** object, or SYSTEMTIME structure containing the date at the lowest end of the range.

pMaxRange

A pointer to a **COleDateTime** object, a **CTime** object, or **SYSTEMTIME** structure containing the date at the highest end of the range.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function implements the behavior of the Win32 message MCM_SETRANGE, as described in the *Platform SDK*. In MFC's implementation of **SetRange**, you can specify **COleDateTime** usage, a **CTime** usage, or a **SYSTEMTIME** structure usage.

Example

See the example for CMonthCalCtrl::GetRange.

See Also

Reference

CMonthCalCtrl Class
Hierarchy Chart
CMonthCalCtrl::GetRange

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::SetSelRange

Sets the selection for a month calendar control to a given date range.

```
BOOL SetSelRange(
   const COleDateTime& pMinRange,
   const COleDateTime& pMaxRange
);
BOOL SetSelRange(
   const CTime& pMinRange,
   const CTime& pMaxRange
);
BOOL SetSelRange(
   const LPSYSTEMTIME pMinRange,
   const LPSYSTEMTIME pMaxRange
);
```

Parameters

pMinRange

A pointer to a **COleDateTime** object, a **CTime** object, or SYSTEMTIME structure containing the date at the lowest end of the range.

pMaxRange

A pointer to a **COleDateTime** object, a **CTime** object, or **SYSTEMTIME** structure containing the date at the highest end of the range.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function implements the behavior of the Win32 message MCM_SETSELRANGE, as described in the *Platform SDK*. In MFC's implementation of **SetSelRange**, you can specify **COleDateTime** usage, a **CTime** usage, or a **SYSTEMTIME** structure usage.

See Also

Reference

CMonthCalCtrl Class
Hierarchy Chart
CMonthCalCtrl::GetSelRange

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::SetToday

Sets the calendar control for the current day.

```
void SetToday(
   const COleDateTime& refDateTime
);
void SetToday(
   const CTime* pDateTime
);
void SetToday(
   const LPSYSTEMTIME pDateTime
);
```

Parameters

refDateTime

A reference to a COleDateTime object that contains the current date.

pDateTime

In the second version, a pointer to a CTime object containing the current date information. In the third version, a pointer to a SYSTEMTIME structure that contains the current date information.

Remarks

This member function implements the behavior of the Win32 message MCM_SETTODAY, as described in the *Platform SDK*.

Example

See the example for CMonthCalCtrl::GetToday.

See Also

Reference

CMonthCalCtrl Class
Hierarchy Chart
CMonthCalCtrl::GetToday

Other Resources

CMonthCalCtrl Members

CMonthCalCtrl::SizeMinReq

Displays the month calendar control to the minimum size that displays one month.

```
BOOL SizeMinReq(
   BOOL bRepaint = TRUE
);
```

Parameters

bRepaint

Specifies whether the control is to be repainted. By default, **TRUE**. If **FALSE**, no repainting occurs.

Return Value

Nonzero if the month calendar control is sized to its minimum; otherwise 0.

Remarks

Calling **SizeMinReq** successfully displays the entire month calendar control for one month's calendar.

See Also

Reference

CMonthCalCtrl Class
Hierarchy Chart
CMonthCalCtrl::GetMinReqRect

Other Resources

CMonthCalCtrl Members

CMultiDocTemplate Class

Defines a document template that implements the multiple document interface (MDI).

```
class CMultiDocTemplate : public CDocTemplate
```

Remarks

An MDI application uses the main frame window as a workspace in which the user can open zero or more document frame windows, each of which displays a document. For a more detailed description of the MDI, see *Windows Interface Guidelines for Software Design*.

A document template defines the relationships among three types of classes:

- A document class, which you derive from CDocument.
- A view class, which displays data from the document class listed above. You can derive this class from CView, **CScrollView**, **CFormView**, or **CEditView**. (You can also use **CEditView** directly.)
- A frame window class, which contains the view. For an MDI document template, you can derive this class from **CMDIChildWnd**, or, if you don't need to customize the behavior of the document frame windows, you can use CMDIChildWnd directly without deriving your own class.

An MDI application can support more than one type of document, and documents of different types can be open at the same time. Your application has one document template for each document type that it supports. For example, if your MDI application supports both spreadsheets and text documents, the application has two **CMultiDocTemplate** objects.

The application uses the document template(s) when the user creates a new document. If the application supports more than one type of document, then the framework gets the names of the supported document types from the document templates and displays them in a list in the File New dialog box. Once the user has selected a document type, the application creates a document class object, a frame window object, and a view object and attaches them to each other.

You do not need to call any member functions of **CMultiDocTemplate** except the constructor. The framework handles **CMultiDocTemplate** objects internally.

For more information on **CMultiDocTemplate**, see Document Templates and the Document/View Creation Process.

Requirements

Header: afxwin.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CDocTemplate Class
Hierarchy Chart
CDocTemplate Class
CSingleDocTemplate Class
CWinApp Class

Other Resources

CMultiDocTemplate Members

CMultiDocTemplate Members

Base Class Members

CObject Members

CCmdTarget Members

CDocTemplate Members

Construction

CMultiDocTemplate	Constructs a **CMultiDocTemplate** object.

See Also

Reference

CMultiDocTemplate Class

Hierarchy Chart

CMultiDocTemplate Member Functions

For information about the member functions in **CMultiDocTemplate**, see CMultiDocTemplate Members.

CMultiDocTemplate::CMultiDocTemplate

Constructs a **CMultiDocTemplate** object.

```
CMultiDocTemplate(
   UINT nIDResource,
   CRuntimeClass* pDocClass,
   CRuntimeClass* pFrameClass,
   CRuntimeClass* pViewClass
);
```

Parameters

nIDResource

Specifies the ID of the resources used with the document type. This may include menu, icon, accelerator table, and string resources.

The string resource consists of up to seven substrings separated by the '\n' character (the '\n' character is needed as a place holder if a substring is not included; however, trailing '\n' characters are not necessary); these substrings describe the document type. For information on the substrings, see CDocTemplate::GetDocString. This string resource is found in the application's resource file. For example:

```
// MYCALC.RC
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    IDR_SHEETTYPE "\nSheet\nWorksheet\nWorksheets (*.myc)\n.myc\n MyCalcSheet\nMyCalc Wor
ksheet"
END
```

Note that the string begins with a '\n' character; this is because the first substring is not used for MDI applications and so is not included. You can edit this string using the string editor; the entire string appears as a single entry in the String Editor, not as seven separate entries.

For more information about these resource types, see Resource Editors.

pDocClass

Points to the **CRuntimeClass** object of the document class. This class is a **CDocument**-derived class you define to represent your documents.

pFrameClass

Points to the **CRuntimeClass** object of the frame-window class. This class can be a **CMDIChildWnd**-derived class, or it can be **CMDIChildWnd** itself if you want default behavior for your document frame windows.

pViewClass

Points to the **CRuntimeClass** object of the view class. This class is a **CView**-derived class you define to display your documents.

Remarks

Dynamically allocate one **CMultiDocTemplate** object for each document type that your application supports and pass each one to **CWinApp::AddDocTemplate** from the `InitInstance` member function of your application class.

Example

```
//example for CMultiDocTemplate
BOOL CMyApp::InitInstance()
{
        // ...
        // Establish all of the document types
        // supported by the application

        AddDocTemplate( new CMultiDocTemplate( IDR_SHEETTYPE,
```

```
                              RUNTIME_CLASS( CSheetDoc ),
                              RUNTIME_CLASS( CMDIChildWnd ),
                              RUNTIME_CLASS( CSheetView ) ) );

         AddDocTemplate( new CMultiDocTemplate( IDR_NOTETYPE,
                              RUNTIME_CLASS( CNoteDoc ),
                              RUNTIME_CLASS( CMDIChildWnd ),
                              RUNTIME_CLASS( CNoteView ) ) );
         // ...
}
```

Here is a second example.

```
BOOL CYourApp::InitInstance()
{
   // Normally, an application creates a document
   // template and registers it with MFC as a part
   // of its initialization.

   // IDR_SAMPLERESOURCE is a resource ID string; see
   // the CDocTemplate class overview documentation
   // for more information on its format.

   // The next three parameters use the RUNTIME_CLASS()
   // macro to get runtime type information for the doc,
   // frame, and view classes that will be associated
   // by the template.

   CMultiDocTemplate* pDocTemplate;
   pDocTemplate = new CMultiDocTemplate(
      IDR_SAMPLERESOURCE,
      RUNTIME_CLASS(CYourDoc),
      RUNTIME_CLASS(CChildFrame),
      RUNTIME_CLASS(CYourViewClass));

   // After the following call, MFC is aware of the doc
   // template and will free it when the application is
   // shut down. The doc templates known to MFC will
   // automatically be used when CWinApp:OnFileOpen()
   // or CWinApp::OnFileNew() are called.

   AddDocTemplate(pDocTemplate);

   // You might have other initialization code ...
   // ...

   return TRUE;
}
```

See Also

Reference

CMultiDocTemplate Class
Hierarchy Chart
CDocTemplate::GetDocString
CWinApp::AddDocTemplate
CWinApp::InitInstance
CRuntimeClass Structure

Other Resources

CMultiDocTemplate Members

CMultiLock Class

Represents the access-control mechanism used in controlling access to resources in a multithreaded program.

```
class CMultiLock
```

Remarks

CMultiLock does not have a base class.

To use the synchronization classes CSemaphore, CMutex, and CEvent, you can create either a **CMultiLock** or CSingleLock object to wait on and release the synchronization object. Use **CMultiLock** when there are multiple objects that you could use at a particular time. Use **CSingleLock** when you only need to wait on one object at a time.

To use a **CMultiLock** object, first create an array of the synchronization objects that you wish to wait on. Next, call the **CMultiLock** object's constructor inside a member function in the controlled resource's class. Then call the Lock member function to determine if a resource is available (signaled). If one is, continue with the remainder of the member function. If no resource is available, either wait for a specified amount of time for a resource to be released, or return failure. After use of a resource is complete, either call the Unlock function if the **CMultiLock** object is to be used again, or allow the **CMultiLock** object to be destroyed.

CMultiLock objects are most useful when a thread has a large number of **CEvent** objects it can respond to. Create an array containing all the **CEvent** pointers, and call **Lock**. This will cause the thread to wait until one of the events is signaled.

For more information on how to use **CMultiLock** objects, see the article Multithreading: How to Use the Synchronization Classes.

Requirements

Header: afxmt.h

See Also

Reference

Hierarchy Chart

Other Resources

CMultiLock Members

CMultiLock Members

Construction

CMultiLock	Constructs a **CMultiLock** object.

Methods

IsLocked	Determines if a specific synchronization object in the array is locked.
Lock	Waits on the array of synchronization objects.
Unlock	Releases any owned synchronization objects.

See Also

Reference

CMultiLock Class

Hierarchy Chart

CMultiLock Member Functions

For information about the member functions in **CMultiLock**, see CMultiLock Members.

CMultiLock::CMultiLock

Constructs a **CMultiLock** object.

```
CMultiLock(
   CSyncObject* ppObjects[ ],
   DWORD dwCount,
   BOOL bInitialLock = FALSE
);
```

Parameters

ppObjects

Array of pointers to the synchronization objects to be waited on. Cannot be **NULL**.

dwCount

Number of objects in *ppObjects*. Must be greater than 0.

bInitialLock

Specifies whether to initially attempt to access any of the supplied objects.

Remarks

This function is called after creating the array of synchronization objects to be waited on. It is usually called from within the thread that must wait for one of the synchronization objects to become available.

See Also

Reference

CMultiLock Class

Hierarchy Chart

Other Resources

CMultiLock Members

CMultiLock::IsLocked

Determines if the specified object is nonsignaled (unavailable).

```
BOOL IsLocked(
   DWORD dwItem
);
```

Parameters

dwItem

The index in the array of objects corresponding to the object whose state is being queried.

Return Value

Nonzero if the specified object is locked; otherwise 0.

See Also

Reference

CMultiLock Class

Hierarchy Chart

Other Resources

CMultiLock Members

CMultiLock::Lock

Call this function to gain access to one or more of the resources controlled by the synchronization objects supplied to the **CMultiLock** constructor.

```
DWORD Lock(
   DWORD dwTimeOut = INFINITE,
   BOOL bWaitForAll = TRUE,
   DWORD dwWakeMask = 0
);
```

Parameters

dwTimeOut

Specifies the amount of time to wait for the synchronization object to be available (signaled). If **INFINITE**, **Lock** will wait until the object is signaled before returning.

bWaitForAll

Specifies whether all objects waited on must become signaled at the same time before returning. If **FALSE**, **Lock** will return when any one of the objects waited on is signaled.

dwWakeMask

Specifies other conditions that are allowed to abort the wait. For a full list of the available options for this parameter, see MsgWaitForMultipleObjects in the *Platform SDK*.

Return Value

If **Lock** fails, it returns – 1. If successful, it returns one of the following values:

- Between **WAIT_OBJECT_0** and **WAIT_OBJECT_0** + (number of objects – 1)

  If *bWaitForAll* is **TRUE**, all objects are signaled (available). If *bWaitForAll* is **FALSE**, the return value – **WAIT_OBJECT_0** is the index in the array of objects of the object that is signaled (available).

- **WAIT_OBJECT_0** + (number of objects)

  An event specified in *dwWakeMask* is available in the thread's input queue.

- Between **WAIT_ABANDONED_0** and **WAIT_ABANDONED_0** + (number of objects – 1)

  If *bWaitForAll* is **TRUE**, all objects are signaled, and at least one of the objects is an abandoned mutex object. If *bWaitForAll* is **FALSE**, the return value – **WAIT_ABANDONED_0** is the index in the array of objects of the abandoned mutex object that satisfied the wait.

- **WAIT_TIMEOUT**

  The timeout interval specified in *dwTimeOut* expired without the wait succeeding.

Remarks

If *bWaitForAll* is **TRUE**, **Lock** will return successfully as soon as all the synchronization objects become signaled simultaneously. If *bWaitForAll* is **FALSE**, **Lock** will return as soon as one or more of the synchronization objects becomes signaled.

If **Lock** is not able to return immediately, it will wait for no more than the number of milliseconds specified in the *dwTimeOut* parameter before returning. If *dwTimeOut* is **INFINITE**, **Lock** will not return until access to an object is gained or a condition specified in *dwWakeMask* was met. Otherwise, if **Lock** was able to acquire a synchronization object, it will return successfully; if not, it will return failure.

See Also

Reference

CMultiLock Class

Hierarchy Chart

Other Resources

CMultiLock::Unlock

Releases the synchronization object owned by **CMultiLock**.

```
BOOL Unlock( );
BOOL Unlock(
   LONG lCount,
   LPLONG lPrevCount = NULL
);
```

Parameters

lCount

Number of reference counts to release. Must be greater than 0. If the specified amount would cause the object's count to exceed its maximum, the count is not changed and the function returns **FALSE**.

lPrevCount

Points to a variable to receive the previous count for the synchronization object. If **NULL**, the previous count is not returned.

Return Value

Nonzero if the function was successful; otherwise 0.

Remarks

This function is called by **CMultiLock**'s destructor.

The first form of **Unlock** tries to unlock the synchronization object managed by **CMultiLock**. The second form of **Unlock** tries to unlock the **CSemaphore** objects owned by **CMultiLock**. If **CMultiLock** does not own any locked **CSemaphore** object, the function returns **FALSE**; otherwise, it returns **TRUE**. *lCount* and *lpPrevCount* are exactly the same as the parameters of CSingleLock::Unlock. The second form of **Unlock** is rarely applicable to multilock situations.

See Also

Reference

CMultiLock Class

Hierarchy Chart

Other Resources

CMultiLock Members

CMultiPageDHtmlDialog Class

A multipage dialog displays multiple HTML pages sequentially and handles the events from each page.

```
class CMultiPageDHtmlDialog : public CDHtmlDialog
```

Remarks

The mechanism for doing this is a DHTML and URL event map, which contains embedded event maps for each page.

Example

This multipage dialog assumes three HTML resources that define simple wizard-like functionality. The first page has a **Next** button, the second a **Prev** and **Next** button, and the third a **Prev** button. When one of the buttons is pressed, a handler function calls CDHtmlDialog::LoadFromResource to load the appropriate new page.

The pertinent parts of the class declaration (in CMyMultiPageDlg.h):

```
class CMyMultiPageDlg : public CMultiPageDHtmlDialog
{
public:
   // Declare the DHTML event handlers:
   HRESULT OnPage1Next(IHTMLElement *pElement);
   HRESULT OnPage2Next(IHTMLElement *pElement);
   HRESULT OnPage2Back(IHTMLElement *pElement);
   HRESULT OnPage3Back(IHTMLElement *pElement);
...
   DECLARE_DHTML_URL_EVENT_MAP()
}
```

The pertinent parts of the class implementation (in CMyMultipageDlg.cpp):

```
BEGIN_DHTML_URL_EVENT_MAP(CMyMultiPageDlg)

   BEGIN_EMBED_DHTML_EVENT_MAP(CMyMultiPageDlg, Page1)
      DHTML_EVENT_ONCLICK(_T("Next"), OnPage1Next)
   END_EMBED_DHTML_EVENT_MAP()

   BEGIN_EMBED_DHTML_EVENT_MAP(CMyMultiPageDlg, Page2)
      DHTML_EVENT_ONCLICK(_T("Back"), OnPage2Back)
      DHTML_EVENT_ONCLICK(_T("Next"), OnPage2Next)
   END_EMBED_DHTML_EVENT_MAP()

   BEGIN_EMBED_DHTML_EVENT_MAP(CMyMultiPageDlg, Page3)
      DHTML_EVENT_ONCLICK(_T("Back"), OnPage3Back)
   END_EMBED_DHTML_EVENT_MAP()

   BEGIN_URL_ENTRIES(CMyMultiPageDlg)
      URL_EVENT_ENTRY(CMyMultiPageDlg, _T("131"), Page1)
      URL_EVENT_ENTRY(CMyMultiPageDlg, _T("132"), Page2)
      URL_EVENT_ENTRY(CMyMultiPageDlg, _T("133"), Page3)
      // Note: IDR_PAGE1 = 131, IDR_PAGE2 = 132, IDR_PAGE3 = 133
   END_URL_ENTRIES()

END_DHTML_URL_EVENT_MAP(CMyMultiPageDlg)

HRESULT CMyMultiPageDlg::OnPage1Next(IHTMLElement *pElement)
{
   LoadFromResource(IDR_PAGE2);
   return S_OK;
}

HRESULT CMyMultiPageDlg::OnPage2Next(IHTMLElement *pElement)
```

```
{
   LoadFromResource(IDR_PAGE3);
   return S_OK;
}

HRESULT CMyMultiPageDlg::OnPage2Back(IHTMLElement *pElement)
{
   LoadFromResource(IDR_PAGE1);
   return S_OK;
}

HRESULT CMyMultiPageDlg::OnPage3Back(IHTMLElement *pElement)
{
   LoadFromResource(IDR_PAGE2);
   return S_OK;
}
```

Requirements

Header: afxdhtml.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CDHtmlDialog Class

Other Resources

CMultiPageDHtmlDialog Members

Multipage DHTML and URL Event Maps

CMultiPageDHtmlDialog Members

Member Functions

CMultiPageDHtmlDialog	Constructs a multipage (wizard-style) DHTML dialog object.
~CMultiPageDHtmlDialog	Destroys a multipage DHTML dialog object.

See Also

Reference

CMultiPageDHtmlDialog Class

CMultiPageDHtmlDialog Member Functions

For information about the member functions in **CMultiPageDHtmlDialog**, see CMultiPageDHtmlDialog Members.

CMultiPageDHtmlDialog::CMultiPageDHtmlDialog

Constructs a multipage (wizard-style) DHTML dialog object.

```
CMultiPageDHtmlDialog(
   LPCTSTR lpszTemplateName,
   LPCTSTR szHtmlResID = NULL,
   CWnd * pParentWnd = NULL
);
CMultiPageDHtmlDialog(
   UINT nIDTemplate,
   UINT nHtmlResID = 0,
   CWnd * pParentWnd = NULL
);
CMultiPageDHtmlDialog( );
```

Parameters

lpszTemplateName

The null-terminated string that is the name of a dialog-box template resource.

szHtmlResID

The null-terminated string that is the name of an HTML resource.

pParentWnd

A pointer to the parent or owner window object (of type CWnd) to which the dialog object belongs. If it is **NULL**, the dialog object's parent window is set to the main application window.

nIDTemplate

Contains the ID number of a dialog-box template resource.

nHtmlResID

Contains the ID number of an HTML resource.

See Also

Reference

CMultiPageDHtmlDialog Class

Other Resources

CMultiPageDHtmlDialog Members

CMultiPageDHtmlDialog::~CMultiPageDHtmlDialog

Destroys a multipage DHTML dialog object.

```
virtual ~CMultiPageDHtmlDialog( );
```

See Also

Reference

CMultiPageDHtmlDialog Class

Other Resources

CMultiPageDHtmlDialog Members

CMutex Class

Represents a "mutex" — a synchronization object that allows one thread mutually exclusive access to a resource.

```
class CMutex : public CSyncObject
```

Remarks

Mutexes are useful when only one thread at a time can be allowed to modify data or some other controlled resource. For example, adding nodes to a linked list is a process that should only be allowed by one thread at a time. By using a **CMutex** object to control the linked list, only one thread at a time can gain access to the list.

To use a **CMutex** object, construct the **CMutex** object when it is needed. Specify the name of the mutex you wish to wait on, and that your application should initially own it. You can then access the mutex when the constructor returns. Call CSyncObject::Unlock when you are done accessing the controlled resource.

An alternative method for using **CMutex** objects is to add a variable of type **CMutex** as a data member to the class you wish to control. During construction of the controlled object, call the constructor of the **CMutex** data member specifying if the mutex is initially owned, the name of the mutex (if it will be used across process boundaries), and desired security attributes.

To access resources controlled by **CMutex** objects in this manner, first create a variable of either type CSingleLock or type CMultiLock in your resource's access member function. Then call the lock object's **Lock** member function (for example, CSingleLock::Lock). At this point, your thread will either gain access to the resource, wait for the resource to be released and gain access, or wait for the resource to be released and time out, failing to gain access to the resource. In any case, your resource has been accessed in a thread-safe manner. To release the resource, use the lock object's **Unlock** member function (for example, CSingleLock::Unlock), or allow the lock object to fall out of scope.

For more information on using **CMutex** objects, see the article Multithreading: How to Use the Synchronization Classes.

Requirements

Header: afxmt.h

See Also

Reference

CSyncObject Class

Hierarchy Chart

Other Resources

CMutex Members

CMutex Members

Base Class Members

CObject Members

CSyncObject Members

Construction

CMutex	Constructs a **CMutex** object.

See Also

Reference

CMutex Class

Hierarchy Chart

CMutex Member Functions

For information about the member functions in **CMutex**, see CMutex Members.

CMutex::CMutex

Constructs a named or unnamed **CMutex** object.

```
CMutex(
   BOOL bInitiallyOwn = FALSE,
   LPCTSTR lpszName = NULL,
   LPSECURITY_ATTRIBUTES lpsaAttribute = NULL
);
```

Parameters

bInitiallyOwn

Specifies if the thread creating the **CMutex** object initially has access to the resource controlled by the mutex.

lpszName

Name of the **CMutex** object. If another mutex with the same name exists, *lpszName* must be supplied if the object will be used across process boundaries. If **NULL**, the mutex will be unnamed. If the name matches an existing mutex, the constructor builds a new **CMutex** object which references the mutex of that name. If the name matches an existing synchronization object that is not a mutex, the construction will fail.

lpsaAttribute

Security attributes for the mutex object. For a full description of this structure, see SECURITY_ATTRIBUTES in the *Platform SDK*.

Remarks

To access or release a **CMutex** object, create a CMultiLock or CSingleLock object and call its Lock and Unlock member functions. If the **CMutex** object is being used stand-alone, call its **Unlock** member function to release it.

Security Note
After creating the **CMutex** object, use GetLastError to ensure that the mutex did not already exist. If the mutex did exist unexpectedly, it may indicate a rogue process is squatting and may be intending to use the mutex maliciously. In this case, the recommended security-conscious procedure is to close the handle and continue as if there was a failure in creating the object.

See Also

Reference

CMutex Class

Hierarchy Chart

Other Resources

CMutex Members

CNotSupportedException Class

Represents an exception that is the result of a request for an unsupported feature.

```
class CNotSupportedException : public CSimpleException
```

Remarks

No further qualification is necessary or possible.

For more information on using **CNotSupportedException**, see the article Exception Handling (MFC).

Requirements

Header: afx.h

See Also

Reference

CException Class

Hierarchy Chart

Other Resources

CNotSupportedException Members

CNotSupportedException Members

Base Class Members

CObject Members

CException Members

Construction

CNotSupportedException	Constructs a **CNotSupportedException** object.

See Also

Reference

CNotSupportedException Class

Hierarchy Chart

CNotSupportedException Member Functions

For information about the member functions in **CNotSupportedException**, see CNotSupportedException Members.

CNotSupportedException::CNotSupportedException

Constructs a **CNotSupportedException** object.

```
CNotSupportedException( );
```

Remarks

Do not use this constructor directly, but rather call the global function AfxThrowNotSupportedException. For more information about exception processing, see the article Exceptions.

See Also

Reference

CNotSupportedException Class
Hierarchy Chart
AfxThrowNotSupportedException

Other Resources

CNotSupportedException Members

CObArray Class

Supports arrays of **CObject** pointers.

```
class CObArray : public CObject
```

Remarks

These object arrays are similar to C arrays, but they can dynamically shrink and grow as necessary.

Array indexes always start at position 0. You can decide whether to fix the upper bound or allow the array to expand when you add elements past the current bound. Memory is allocated contiguously to the upper bound, even if some elements are null.

Under Win32, the size of a **CObArray** object is limited only to available memory.

As with a C array, the access time for a **CObArray** indexed element is constant and is independent of the array size.

CObArray incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. If an array of **CObject** pointers is stored to an archive, either with the overloaded insertion operator or with the **Serialize** member function, each **CObject** element is, in turn, serialized along with its array index.

If you need a dump of individual **CObject** elements in an array, you must set the depth of the **CDumpContext** object to 1 or greater.

When a **CObArray** object is deleted, or when its elements are removed, only the **CObject** pointers are removed, not the objects they reference.

Note
Before using an array, use **SetSize** to establish its size and allocate memory for it. If you do not use **SetSize**, adding elements to your array causes it to be frequently reallocated and copied. Frequent reallocation and copying are inefficient and can fragment memory.

Array class derivation is similar to list derivation. For details on the derivation of a special-purpose list class, see the article Collections.

Note
You must use the **IMPLEMENT_SERIAL** macro in the implementation of your derived class if you intend to serialize the array.

Requirements

Header: afxcoll.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class
Hierarchy Chart
CStringArray Class
CPtrArray Class
CByteArray Class
CWordArray Class
CDWordArray Class

Other Resources

CObArray Members

CObArray Members

Base Class Members

CObject Members

Construction

CObArray	Constructs an empty array for **CObject** pointers.

Bounds

GetCount	Gets the number of elements in this array.
GetSize	Gets the number of elements in this array.
GetUpperBound	Returns the largest valid index.
SetSize	Sets the number of elements to be contained in this array.

Operations

FreeExtra	Frees all unused memory above the current upper bound.
IsEmpty	Determines if the array is empty.
RemoveAll	Removes all the elements from this array.

Element Access

ElementAt	Returns a temporary reference to the element pointer within the array.
GetAt	Returns the value at a given index.
GetData	Allows access to elements in the array. Can be **NULL**.
SetAt	Sets the value for a given index; array not allowed to grow.

Growing the Array

Add	Adds an element to the end of the array; grows the array if necessary.
Append	Appends another array to the array; grows the array if necessary.
Copy	Copies another array to the array; grows the array if necessary.
SetAtGrow	Sets the value for a given index; grows the array if necessary.

Insertion/Removal

InsertAt	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt	Removes an element at a specific index.

Operators

operator []	Sets or gets the element at the specified index.

See Also

Reference

CObArray Class

Hierarchy Chart

CObArray Member Functions

For information about the member functions in **CObArray**, see CObArray Members.

CObArray::Add

Adds a new element to the end of an array, growing the array by 1.

```
INT_PTR Add(
   CObject* newElement
);
```

Parameters

newElement

The **CObject** pointer to be added to this array.

Return Value

The index of the added element.

Remarks

If SetSize has been used with an *nGrowBy* value greater than 1, then extra memory may be allocated. However, the upper bound will increase by only 1.

The following table shows other member functions that are similar to **CObArray::Add**.

Class	Member Function
CByteArray	**INT_PTR Add(BYTE** *newElement* **);** **throw(CMemoryException*);**
CDWordArray	**INT_PTR Add(DWORD** *newElement* **);** **throw(CMemoryException*);**
CPtrArray	**INT_PTR Add(void*** *newElement* **);** **throw(CMemoryException*);**
CStringArray	**INT_PTR Add(LPCTSTR** *newElement* **);** **throw(CMemoryException*);** **INT_PTR Add(const CString&** *newElement* **);**
CUIntArray	**INT_PTR Add(UINT** *newElement* **);** **throw(CMemoryException*);**
CWordArray	**INT_PTR Add(WORD** *newElement* **);** **throw(CMemoryException*);**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
// example for CObArray::Add

    CObArray array;

    array.Add( new CAge( 21 ) ); // Element 0
    array.Add( new CAge( 40 ) ); // Element 1
#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "Add example: " << &array << "\n";
#endif
```

The results from this program are as follows:

```
Add example: A CObArray with 2 elements
    [0] = a CAge at $442A 21
    [1] = a CAge at $4468 40
```

See Also

Reference

CObArray Class
Hierarchy Chart
CObArray::SetAt
CObArray::SetAtGrow
CObArray::InsertAt
CObArray::operator [ ]

Other Resources

CObArray Members

CObArray::Append

Call this member function to add the contents of another array to the end of the given array.

```
INT_PTR Append(
   const CObArray& src
);
```

Parameters

src

Source of the elements to be appended to the array.

Return Value

The index of the first appended element.

Remarks

The arrays must be of the same type.

If necessary, **Append** may allocate extra memory to accommodate the elements appended to the array.

The following table shows other member functions that are similar to **CObArray::Append**.

Class	Member Function
CByteArray	**INT_PTR Append(const CByteArray&** *src* **);**
CDWordArray	**INT_PTR Append(const CDWordArray&** *src* **);**
CPtrArray	**INT_PTR Append(const CPtrArray&** *src* **);**
CStringArray	**INT_PTR Append(const CStringArray&** *src* **);**
CUIntArray	**INT_PTR Append(const CUIntArray&** *src* **);**
CWordArray	**INT_PTR Append(const CWordArray&** *src* **);**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
CObArray myArray1, myArray2;

   // Add elements to the second array.
   myArray2.Add( new CAge( 21 ) );
   myArray2.Add( new CAge( 42 ) );

   // Add elements to the first array and also append the second array.
   myArray1.Add( new CAge( 3 ) );
   myArray1.Append( myArray2 );

#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "myArray1: " << &myArray1 << "\n";
    afxDump << "myArray2: " << &myArray2 << "\n";
#endif
```

See Also

Reference

CObArray Class

Hierarchy Chart
CObArray::Copy
Other Resources
CObArray Members

CObArray::Copy

Call this member function to overwrite the elements of the given array with the elements of another array of the same type.

```
void Copy(
   const CObArray& src
);
```

Parameters

src

Source of the elements to be copied to the array.

Remarks

Copy does not free memory; however, if necessary, **Copy** may allocate extra memory to accommodate the elements copied to the array.

The following table shows other member functions that are similar to **CObArray::Copy**.

Class	Member Function
CByteArray	**void Copy(const CByteArray&** *src* **);**
CDWordArray	**void Copy(const CDWordArray&** *src* **);**
CPtrArray	**void Copy(const CPtrArray&** *src* **);**
CStringArray	**void Copy(const CStringArray&** *src* **);**
CUIntArray	**void Copy(const CUIntArray&** *src* **);**
CWordArray	**void Copy(const CWordArray&** *src* **);**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
CObArray myArray1, myArray2;

   // Add elements to the second array.
   myArray2.Add( new CAge( 21 ) );
   myArray2.Add( new CAge( 42 ) );

   // Copy the elements from the second array to the first.
   myArray1.Copy( myArray2 );

#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "myArray1: " << &myArray1 << "\n";
    afxDump << "myArray2: " << &myArray2 << "\n";
#endif
```

See Also

Reference

CObArray Class

Hierarchy Chart

CObArray::Append

Other Resources

CObArray Members

CObArray::CObArray

Constructs an empty **CObject** pointer array.

```
CObArray( );
```

Remarks

The array grows one element at a time.

The following table shows other constructors that are similar to **CObArray::CObArray**.

Class	Constructor
CByteArray	**CByteArray( );**
CDWordArray	**CDWordArray( );**
CPtrArray	**CPtrArray( );**
CStringArray	**CStringArray( );**
CUIntArray	**CUIntArray( );**
CWordArray	**CWordArray( );**

Example

```
CObArray array; //Array with default blocksize
CObArray* pArray = new CObArray; //Array on the heap with default blocksize
```

See Also

Reference

CObArray Class

Hierarchy Chart

CObList::CObList

Other Resources

CObArray Members

CObArray::ElementAt

Returns a temporary reference to the element pointer within the array.

```
CObject*& ElementAt(
   INT_PTR nIndex
);
```

Parameters

nIndex

An integer index that is greater than or equal to 0 and less than or equal to the value returned by **GetUpperBound**.

Return Value

A reference to a **CObject** pointer.

Remarks

It is used to implement the left-side assignment operator for arrays. Note that this is an advanced function that should be used only to implement special array operators.

The following table shows other member functions that are similar to **CObArray::ElementAt**.

Class	Member Function
CByteArray	**BYTE& ElementAt(INT_PTR** *nIndex* **);**
CDWordArray	**DWORD& ElementAt(INT_PTR** *nIndex* **);**
CPtrArray	**void*& ElementAt(INT_PTR** *nIndex* **);**
CStringArray	**CString& ElementAt(INT_PTR** *nIndex* **);**
CUIntArray	**UINT& ElementAt(INT_PTR** *nIndex* **);**
CWordArray	**WORD& ElementAt(INT_PTR** *nIndex* **);**

Example

See the example for CObArray::GetSize.

See Also

Reference

CObArray Class

Hierarchy Chart

CObArray::operator [ ]

Other Resources

CObArray Members

CObArray::FreeExtra

Frees any extra memory that was allocated while the array was grown.

```
void FreeExtra( );
```

Remarks

This function has no effect on the size or upper bound of the array.

The following table shows other member functions that are similar to **CObArray::FreeExtra**.

Class	Member Function
CByteArray	**void FreeExtra( );**
CDWordArray	**void FreeExtra( );**
CPtrArray	**void FreeExtra( );**
CStringArray	**void FreeExtra( );**
CUIntArray	**void FreeExtra( );**
CWordArray	**void FreeExtra( );**

Example

See the example for CObArray::GetData.

See Also

Reference

CObArray Class

Hierarchy Chart

Other Resources

CObArray Members

CObArray::GetAt

Returns the array element at the specified index.

```
CObject* GetAt(
   INT_PTR nIndex
) const;
```

Parameters

nIndex

An integer index that is greater than or equal to 0 and less than or equal to the value returned by **GetUpperBound**.

Return Value

The **CObject** pointer element currently at this index.

Remarks

Note
Passing a negative value or a value greater than the value returned by **GetUpperBound** will result in a failed assertion.

The following table shows other member functions that are similar to **CObArray::GetAt**.

Class	Member Function
CByteArray	**BYTE GetAt(INT_PTR** *nIndex* **) const;**
CDWordArray	**DWORD GetAt(INT_PTR** *nIndex* **) const;**
CPtrArray	**void* GetAt(INT_PTR** *nIndex* **) const;**
CStringArray	**CString GetAt(INT_PTR** *nIndex* **) const;**
CUIntArray	**UINT GetAt(INT_PTR** *nIndex* **) const;**
CWordArray	**WORD GetAt(INT_PTR** *nIndex* **) const;**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
// example for CObArray::GetAt

CObArray array;

array.Add( new CAge( 21 ) ); // Element 0
array.Add( new CAge( 40 ) ); // Element 1
ASSERT( *(CAge*) array.GetAt( 0 ) == CAge( 21 ) );
```

See Also

Reference

CObArray Class
Hierarchy Chart
CObArray::SetAt
CObArray::operator [ ]

Other Resources

CObArray Members

CObArray::GetCount

Returns the number of array elements.

```
INT_PTR GetCount( ) const;
```

Return Value

The number of items in the array.

Remarks

Call this method to retrieve the number of elements in the array. Because indexes are zero-based, the size is 1 greater than the largest index.

The following table shows other member functions that are similar to **CObArray::GetCount**.

Class	Member Function
CByteArray	**INT_PTR GetCount( ) const;**
CDWordArray	**INT_PTR GetCount( ) const;**
CPtrArray	**INT_PTR GetCount( ) const;**
CStringArray	**INT_PTR GetCount( ) const;**
CUIntArray	**INT_PTR GetCount( ) const;**
CWordArray	**INT_PTR GetCount( ) const;**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
CObArray myArray;

   // Add elements to the array.
   for (int i=0;i < 10;i++)
      myArray.Add( new CAge( i ) );

   // Add 100 to all the elements of the array.
   for (i=0;i < myArray.GetCount();i++)
   {
      CAge*& pAge = (CAge*&) myArray.ElementAt(i);
      delete pAge;
      pAge = new CAge( 100+i );
   }
```

See Also

Reference

CObArray Class
Hierarchy Chart
CObArray::GetUpperBound
CObArray::SetSize
CObArray::GetSize

Other Resources

CObArray Members

CObArray::GetData

Use this member function to gain direct access to the elements in the array.

```
const CObject** GetData( ) const;
CObject** GetData( );
```

Return Value

A pointer to the array of **CObject** pointers.

Remarks

If no elements are available, **GetData** returns a null value.

While direct access to the elements of an array can help you work more quickly, use caution when calling **GetData**; any errors you make directly affect the elements of your array.

The following table shows other member functions that are similar to **CObArray::GetData**.

Class	Member Function
CByteArray	**const BYTE* GetData( ) const;BYTE* GetData( );**
CDWordArray	**const DWORD* GetData( ) const;DWORD* GetData( );**
CPtrArray	**const void** GetData( ) const;void** GetData( );**
CStringArray	**const CString* GetData( ) const;CString* GetData( );**
CUIntArray	**const UINT* GetData( ) const;UINT* GetData( );**
CWordArray	**const WORD* GetData( ) const;WORD* GetData( );**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
CObArray myArray;
   int i;

   // Allocate memory for at least 32 elements.
   myArray.SetSize(32, 128);

   // Add elements to the array.
   CAge** ppAge = (CAge**) myArray.GetData();
   for (i=0;i < 32;i++,ppAge++)
      *ppAge = new CAge( i );

   // Only keep first 5 elements and free extra (unused) bytes.
   myArray.SetSize(5, 128);
   myArray.FreeExtra();

#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "myArray: " << &myArray << "\n";
#endif
```

See Also

Reference

CObArray Class

Hierarchy Chart

COb Array::GetAt
COb Array::SetAt
COb Array::ElementAt

Other Resources

COb Array Members

CObArray::GetSize

Returns the size of the array.

```
INT_PTR GetSize( ) const;
```

Remarks

Since indexes are zero-based, the size is 1 greater than the largest index.

The following table shows other member functions that are similar to **CObArray::GetSize**.

Class	Member Function
CByteArray	**INT_PTR GetSize( ) const;**
CDWordArray	**INT_PTR GetSize( ) const;**
CPtrArray	**INT_PTR GetSize( ) const;**
CStringArray	**INT_PTR GetSize( ) const;**
CUIntArray	**INT_PTR GetSize( ) const;**
CWordArray	**INT_PTR GetSize( ) const;**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
CObArray myArray;

   // Add elements to the array.
   for (int i=0;i < 10;i++)
      myArray.Add( new CAge( i ) );

   // Add 100 to all the elements of the array.
   for (i=0;i < myArray.GetSize();i++)
   {
      CAge*& pAge = (CAge*&) myArray.ElementAt(i);
      delete pAge;
      pAge = new CAge( 100+i );
   }

#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "myArray: " << &myArray << "\n";
#endif
```

See Also

Reference

CObArray Class
Hierarchy Chart
CObArray::GetUpperBound
CObArray::SetSize
CObArray::GetCount

Other Resources

CObArray Members

CObArray::GetUpperBound

Returns the current upper bound of this array.

```
INT_PTR GetUpperBound( ) const;
```

Return Value

The index of the upper bound (zero-based).

Remarks

Because array indexes are zero-based, this function returns a value 1 less than **GetSize**.

The condition **GetUpperBound( )** = –1 indicates that the array contains no elements.

The following table shows other member functions that are similar to **CObArray::GetUpperBound**.

Class	Member Function
CByteArray	**INT_PTR GetUpperBound( ) const;**
CDWordArray	**INT_PTR GetUpperBound( ) const;**
CPtrArray	**INT_PTR GetUpperBound( ) const;**
CStringArray	**INT_PTR GetUpperBound( ) const;**
CUIntArray	**INT_PTR GetUpperBound( ) const;**
CWordArray	**INT_PTR GetUpperBound( ) const;**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
// example for CObArray::GetUpperBound

CObArray array;

array.Add( new CAge( 21 ) ); // Element 0
array.Add( new CAge( 40 ) ); // Element 1
ASSERT( array.GetUpperBound() == 1 ); // Largest index
```

See Also

Reference

CObArray Class
Hierarchy Chart
CObArray::GetSize
CObArray::SetSize

Other Resources

CObArray Members

CObArray::InsertAt

Inserts an element (or all the elements in another array) at a specified index.

```
void InsertAt(
   INT_PTR nIndex,
   CObject* newElement,
   INT_PTR nCount = 1
);
void InsertAt(
   INT_PTR nStartIndex,
   CObArray* pNewArray
);
```

Parameters

nIndex

An integer index that may be greater than the value returned by **GetUpperBound**.

newElement

The **CObject** pointer to be placed in this array. A *newElement* of value **NULL** is allowed.

nCount

The number of times this element should be inserted (defaults to 1).

nStartIndex

An integer index that may be greater than the value returned by **GetUpperBound**.

pNewArray

Another array that contains elements to be added to this array.

Remarks

The first version of **InsertAt** inserts one element (or multiple copies of an element) at a specified index in an array. In the process, it shifts up (by incrementing the index) the existing element at this index, and it shifts up all the elements above it.

The second version inserts all the elements from another **CObArray** collection, starting at the *nStartIndex* position.

The **SetAt** function, in contrast, replaces one specified array element and does not shift any elements.

The following table shows other member functions that are similar to **CObArray::InsertAt**.

Class	Member Function
CByteArray	**void InsertAt(INT_PTR** *nIndex* **, BYTE** *newElement* **, int** *nCount* **= 1);** **throw(CMemoryException*);** **void InsertAt(INT_PTR** *nStartIndex* **, CByteArray*** *pNewArray* **);** **throw(CMemoryException*);**
CDWordArray	**void InsertAt(INT_PTR** *nIndex* **, DWORD** *newElement* **, int** *nCount* **= 1);** **throw(CMemoryException*);** **void InsertAt(INT_PTR** *nStartIndex* **, CDWordArray*** *pNewArray* **);** **throw(CMemoryException*);**

CPtrArray	**void InsertAt(INT_PTR** *nIndex* **, void*** *newElement* **, int** *nCount* **= 1);** **throw(CMemoryException*);** **void InsertAt(INT_PTR** *nStartIndex* **, CPtrArray*** *pNewArray* **);** **throw(CMemoryException*);**
CStringArray	**void InsertAt(INT_PTR** *nIndex* **, LPCTSTR** *newElement* **, int** *nCount* **= 1);** **throw(CMemoryException*);** **void InsertAt(INT_PTR** *nStartIndex* **, CStringArray*** *pNewArray* **);** **throw(CMemoryException*);**
CUIntArray	**void InsertAt(INT_PTR** *nIndex* **, UINT** *newElement* **, int** *nCount* **= 1);** **throw(CMemoryException*);** **void InsertAt(INT_PTR** *nStartIndex* **, CUIntArray*** *pNewArray* **);** **throw(CMemoryException*);**
CWordArray	**void InsertAt(INT_PTR** *nIndex* **, WORD** *newElement* **, int** *nCount* **= 1);** **throw(CMemoryException*);** **void InsertAt(INT_PTR** *nStartIndex* **, CWordArray*** *pNewArray* **);** **throw(CMemoryException*);**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
// example for CObArray::InsertAt

    CObArray array;

    array.Add( new CAge( 21 ) ); // Element 0
    array.Add( new CAge( 40 ) ); // Element 1 (will become 2).
    array.InsertAt( 1, new CAge( 30 ) );  // New element 1
#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "InsertAt example: " << &array << "\n";
#endif
```

The results from this program are as follows:

```
InsertAt example: A CObArray with 3 elements
    [0] = a CAge at $45C8 21
    [1] = a CAge at $4646 30
    [2] = a CAge at $4606 40
```

See Also

Reference

CObArray Class
Hierarchy Chart
CObArray::SetAt
CObArray::RemoveAt

Other Resources

CObArray Members

CObArray::IsEmpty

Determines if the array is empty.

```
BOOL IsEmpty( ) const;
```

Return Value

Nonzero if the array is empty; otherwise 0.

See Also

Reference

CObArray Class

Hierarchy Chart

Other Resources

CObArray Members

CObArray::RemoveAll

Removes all the pointers from this array but does not actually delete the **CObject** objects.

```
void RemoveAll( );
```

Remarks

If the array is already empty, the function still works.

The **RemoveAll** function frees all memory used for pointer storage.

The following table shows other member functions that are similar to **CObArray::RemoveAll**.

Class	Member Function
CByteArray	**void RemoveAll( );**
CDWordArray	**void RemoveAll( );**
CPtrArray	**void RemoveAll( );**
CStringArray	**void RemoveAll( );**
CUIntArray	**void RemoveAll( );**
CWordArray	**void RemoveAll( );**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
// example for CObArray::RemoveAll

CObArray array;
CAge* pa1;
CAge* pa2;

array.Add( pa1 = new CAge( 21 ) ); // Element 0
array.Add( pa2 = new CAge( 40 ) ); // Element 1
ASSERT( array.GetSize() == 2 );
array.RemoveAll(); // Pointers removed but objects not deleted.
ASSERT( array.GetSize() == 0 );
delete pa1;
delete pa2;  // Cleans up memory.
```

See Also

Reference

CObArray Class

Hierarchy Chart

Other Resources

CObArray Members

CObArray::RemoveAt

Removes one or more elements starting at a specified index in an array.

```
void RemoveAt(
   INT_PTR nIndex,
   INT_PTR nCount = 1
);
```

Parameters

nIndex

An integer index that is greater than or equal to 0 and less than or equal to the value returned by **GetUpperBound**.

nCount

The number of elements to remove.

Remarks

In the process, it shifts down all the elements above the removed element(s). It decrements the upper bound of the array but does not free memory.

If you try to remove more elements than are contained in the array above the removal point, then the Debug version of the library asserts.

The **RemoveAt** function removes the **CObject** pointer from the array, but it does not delete the object itself.

The following table shows other member functions that are similar to **CObArray::RemoveAt**.

Class	Member Function
CByteArray	**void RemoveAt(INT_PTR** *nIndex* **, INT_PTR** *nCount* **= 1);**
CDWordArray	**void RemoveAt(INT_PTR** *nIndex* **, INT_PTR** *nCount* **= 1);**
CPtrArray	**void RemoveAt(INT_PTR** *nIndex* **, INT_PTR** *nCount* **= 1);**
CStringArray	**void RemoveAt(INT_PTR** *nIndex* **, INT_PTR** *nCount* **= 1);**
CUIntArray	**void RemoveAt(INT_PTR** *nIndex* **, INT_PTR** *nCount* **= 1);**
CWordArray	**void RemoveAt(INT_PTR** *nIndex*, **INT_PTR** *nCount* **= 1);**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
// example for CObArray::RemoveAt

   CObArray array;
   CObject* pa;

   array.Add( new CAge( 21 ) ); // Element 0
   array.Add( new CAge( 40 ) ); // Element 1
   if( ( pa = array.GetAt( 0 ) ) != NULL )
   {
       array.RemoveAt( 0 );  // Element 1 moves to 0.
       delete pa; // Delete the original element at 0.
   }
#ifdef _DEBUG
   afxDump.SetDepth( 1 );
   afxDump << "RemoveAt example: " << &array << "\n";
```

```
#endif
```

The results from this program are as follows:

```
RemoveAt example: A CObArray with 1 elements
    [0] = a CAge at $4606 40
```

See Also

Reference

CObArray Class
Hierarchy Chart
CObArray::SetAt
CObArray::SetAtGrow
CObArray::InsertAt

Other Resources

CObArray Members

CObArray::SetAt

Sets the array element at the specified index.

```
void SetAt(
   INT_PTR nIndex,
   CObject* newElement
);
```

Parameters

nIndex

An integer index that is greater than or equal to 0 and less than or equal to the value returned by **GetUpperBound**.

newElement

The object pointer to be inserted in this array. A **NULL** value is allowed.

Remarks

SetAt will not cause the array to grow. Use **SetAtGrow** if you want the array to grow automatically.

You must ensure that your index value represents a valid position in the array. If it is out of bounds, then the Debug version of the library asserts.

The following table shows other member functions that are similar to **CObArray::SetAt**.

Class	Member Function
CByteArray	**void SetAt(INT_PTR** *nIndex*, **BYTE** *newElement* **);**
CDWordArray	**void SetAt(INT_PTR** *nIndex*, **DWORD** *newElement* **);**
CPtrArray	**void SetAt(INT_PTR** *nIndex*, **void*** *newElement* **);**
CStringArray	**void SetAt(INT_PTR** *nIndex*, **LPCTSTR** *newElement* **);**
CUIntArray	**void SetAt(INT_PTR** *nIndex*, **UINT** *newElement* **);**
CWordArray	**void SetAt(INT_PTR** *nIndex*, **WORD** *newElement* **);**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
// example for CObArray::SetAt

   CObArray array;
   CObject* pa;

   array.Add( new CAge( 21 ) ); // Element 0
   array.Add( new CAge( 40 ) ); // Element 1
   if( ( pa = array.GetAt( 0 ) ) != NULL )
   {
       array.SetAt( 0, new CAge( 30 ) );  // Replace element 0.
       delete pa; // Delete the original element at 0.
   }
#ifdef _DEBUG
   afxDump.SetDepth( 1 );
   afxDump << "SetAt example: " << &array << "\n";
#endif
```

The results from this program are as follows:

```
SetAt example: A CObArray with 2 elements
    [0] = a CAge at $47E0 30
    [1] = a CAge at $47A0 40
```

See Also

Reference

CObArray Class
Hierarchy Chart
CObArray::GetAt
CObArray::SetAtGrow
CObArray::ElementAt
CObArray::operator [ ]

Other Resources

CObArray Members

CObArray::SetAtGrow

Sets the array element at the specified index.

```
void SetAtGrow(
   INT_PTR nIndex,
   CObject* newElement
);
```

Parameters

nIndex

An integer index that is greater than or equal to 0.

newElement

The object pointer to be added to this array. A **NULL** value is allowed.

Remarks

The array grows automatically if necessary (that is, the upper bound is adjusted to accommodate the new element).

The following table shows other member functions that are similar to **CObArray::SetAtGrow**.

Class	Member Function
CByteArray	**void SetAtGrow(INT_PTR** *nIndex* **, BYTE** *newElement* **);** **throw(CMemoryException*);**
CDWordArray	**void SetAtGrow(INT_PTR** *nIndex* **, DWORD** *newElement* **);** **throw(CMemoryException*);**
CPtrArray	**void SetAtGrow(INT_PTR** *nIndex* **, void*** *newElement* **);** **throw(CMemoryException*);**
CStringArray	**void SetAtGrow(INT_PTR** *nIndex* **, LPCTSTR** *newElement* **);** **throw(CMemoryException*);**
CUIntArray	**void SetAtGrow(INT_PTR** *nIndex* **, UINT** *newElement* **);** **throw(CMemoryException*);**
CWordArray	**void SetAtGrow(INT_PTR** *nIndex* **, WORD** *newElement* **);** **throw(CMemoryException*);**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
// example for CObArray::SetAtGrow

   CObArray array;

   array.Add( new CAge( 21 ) ); // Element 0
   array.Add( new CAge( 40 ) ); // Element 1
   array.SetAtGrow( 3, new CAge( 65 ) ); // Element 2 deliberately
                                         // skipped.
#ifdef _DEBUG
   afxDump.SetDepth( 1 );
```

```
    afxDump << "SetAtGrow example: " << &array << "\n";
#endif
```

The results from this program are as follows:

```
SetAtGrow example: A CObArray with 4 elements
    [0] = a CAge at $47C0 21
    [1] = a CAge at $4800 40
    [2] = NULL
    [3] = a CAge at $4840 65
```

See Also

Reference

CObArray Class
Hierarchy Chart
CObArray::GetAt
CObArray::SetAt
CObArray::ElementAt
CObArray::operator [ ]

Other Resources

CObArray Members

CObArray::SetSize

Establishes the size of an empty or existing array; allocates memory if necessary.

```
void SetSize(
   INT_PTR nNewSize,
   INT_PTR nGrowBy = -1
);
```

Parameters

nNewSize

The new array size (number of elements). Must be greater than or equal to 0.

nGrowBy

The minimum number of element slots to allocate if a size increase is necessary.

Remarks

If the new size is smaller than the old size, then the array is truncated and all unused memory is released. For efficiency, call **SetSize** to set the size of the array before using it. This prevents the need to reallocate and copy the array each time an item is added.

The *nGrowBy* parameter affects internal memory allocation while the array is growing. Its use never affects the array size as reported by **GetSize** and **GetUpperBound**.

If the size of the array has grown, all newly allocated **CObject *** pointers are set to NULL.

The following table shows other member functions that are similar to **CObArray::SetSize**.

Class	Member Function
CByteArray	**void SetSize(INT_PTR** *nNewSize* **, int** *nGrowBy* **= -1);** **throw(CMemoryException*);**
CDWordArray	**void SetSize(INT_PTR** *nNewSize* **, int** *nGrowBy* **= -1);** **throw(CMemoryException*);**
CPtrArray	**void SetSize(INT_PTR** *nNewSize* **, int** *nGrowBy* **= -1);** **throw(CMemoryException*);**
CStringArray	**void SetSize(INT_PTR** *nNewSize* **, int** *nGrowBy* **= -1);** **throw(CMemoryException*);**
CUIntArray	**void SetSize(INT_PTR** *nNewSize* **, int** *nGrowBy* **= -1);** **throw(CMemoryException*);**
CWordArray	**void SetSize(INT_PTR** *nNewSize* **, int** *nGrowBy* **= -1);** **throw(CMemoryException*);**

Example

See the example for CObArray::GetData.

See Also

Reference

CObArray Class

Hierarchy Chart

Other Resources

CObArray Members

CObArray Operators

For information about the operators in **CObArray**, see CObArray Members.

CObArray::operator [ ]

These subscript operators are a convenient substitute for the **SetAt** and **GetAt** functions.

```
CObject*& operator [](
   INT_PTR nIndex
);
CObject* operator [](
   INT_PTR nIndex
) const;
```

Remarks

The first operator, called for arrays that are not **const**, may be used on either the right (r-value) or the left (l-value) of an assignment statement. The second, called for **const** arrays, may be used only on the right.

The Debug version of the library asserts if the subscript (either on the left or right side of an assignment statement) is out of bounds.

The following table shows other operators that are similar to **CObArray::operator []**.

Class	Operator
CByteArray	**BYTE& operator [](** **INT_PTR** *nIndex* **);** **BYTE operator [](** **INT_PTR** *nIndex* **) const;**
CDWordArray	**DWORD& operator [](** **INT_PTR** *nIndex* **);** **DWORD operator [](** **INT_PTR** *nIndex* **) const;**
CPtrArray	**void*& operator [](** **INT_PTR** *nIndex* **);** **void* operator [](** **INT_PTR** *nIndex* **) const;**
CStringArray	**CString& operator [](** **INT_PTR** *nIndex* **);** **CString operator [](** **INT_PTR** *nIndex* **) const;**
CUIntArray	**UINT& operator [](** **INT_PTR** *nIndex* **);** **UINT operator [](** **INT_PTR** *nIndex* **) const;**
CWordArray	**WORD& operator [](** **INT_PTR** *nIndex* **);** **WORD operator [](** **INT_PTR** *nIndex* **) const;**

Example

See CObList::CObList for a listing of the `CAge` class used in all collection examples.

```
// example for CObArray::operator []

CObArray array;
CAge* pa;

array.Add( new CAge( 21 ) ); // Element 0
array.Add( new CAge( 40 ) ); // Element 1
pa = (CAge*)array[0]; // Get element 0
ASSERT( *pa == CAge( 21 ) ); // Get element 0
array[0] = new CAge( 30 ); // Replace element 0
delete pa;
ASSERT( *(CAge*) array[0] == CAge( 30 ) ); // Get new element 0
```

See Also

Reference

CObArray Class
Hierarchy Chart
CObArray::GetAt
CObArray::SetAt

Other Resources

CObArray Members

CObject Class

The principal base class for the Microsoft Foundation Class Library.

```
class AFX_NOVTABLE CObject
```

Remarks

It serves as the root not only for library classes such as **CFile** and **CObList**, but also for the classes that you write. **CObject** provides basic services, including

- Serialization support
- Run-time class information
- Object diagnostic output
- Compatibility with collection classes

Note that **CObject** does not support multiple inheritance. Your derived classes can have only one **CObject** base class, and that **CObject** must be leftmost in the hierarchy. It is permissible, however, to have structures and non-**CObject**-derived classes in right-hand multiple-inheritance branches.

You will realize major benefits from **CObject** derivation if you use some of the optional macros in your class implementation and declarations.

The first-level macros, DECLARE_DYNAMIC and IMPLEMENT_DYNAMIC, permit run-time access to the class name and its position in the hierarchy. This, in turn, allows meaningful diagnostic dumping.

The second-level macros, DECLARE_SERIAL and IMPLEMENT_SERIAL, include all the functionality of the first-level macros, and they enable an object to be "serialized" to and from an "archive."

For information about deriving Microsoft Foundation classes and C++ classes in general and using **CObject**, see Using CObject and Serialization.

Requirements

Header: afx.h

Smart Device Developer Notes

CObject::Dump is not supported in smart device projects because CDumpContext Class is not supported in smart device projects.

See Also
Reference
Hierarchy Chart
Other Resources
CObject Members

CObject Members

Construction

CObject	Default constructor.
CObject	Copy constructor.
operator delete	Special **delete** operator.
operator new	Special **new** operator.

Diagnostics

AssertValid	Validates this object's integrity.
Dump	Produces a diagnostic dump of this object.

Serialization

IsSerializable	Tests to see whether this object can be serialized.
Serialize	Loads or stores an object from/to an archive.

Miscellaneous

GetRuntimeClass	Returns the **CRuntimeClass** structure corresponding to this object's class.
IsKindOf	Tests this object's relationship to a given class.

See Also

Reference

CObject Class

Hierarchy Chart

CObject Member Functions

For information about the member functions in **CObject**, see CObject Members.

CObject::AssertValid

Validates this object's integrity.

```
virtual void AssertValid( ) const;
```

Remarks

AssertValid performs a validity check on this object by checking its internal state. In the Debug version of the library, **AssertValid** may assert and thus terminate the program with a message that lists the line number and filename where the assertion failed.

When you write your own class, you should override the **AssertValid** function to provide diagnostic services for yourself and other users of your class. The overridden **AssertValid** usually calls the **AssertValid** function of its base class before checking data members unique to the derived class.

Because **AssertValid** is a **const** function, you are not permitted to change the object state during the test. Your own derived class **AssertValid** functions should not throw exceptions but rather should assert whether they detect invalid object data.

The definition of "validity" depends on the object's class. As a rule, the function should perform a "shallow check." That is, if an object contains pointers to other objects, it should check to see whether the pointers are not null, but it should not perform validity testing on the objects referred to by the pointers.

Example

See CObList::CObList for a listing of the `CAge` class used in all **CObject** examples.

```
// example for CObject::AssertValid
void CAge::AssertValid() const
{
    CObject::AssertValid();
    ASSERT( m_years > 0 );
    ASSERT( m_years < 105 );
}
```

For another example, see AfxDoForAllObjects.

See Also

Reference

CObject Class

Hierarchy Chart

Other Resources

CObject Members

CObject::CObject

These functions are the standard **CObject** constructors.

```
CObject( );
CObject( const CObject& objectSrc );
```

Parameters

objectSrc

A reference to another **CObject**

Remarks

The default version is automatically called by the constructor of your derived class.

If your class is serializable (it incorporates the **IMPLEMENT_SERIAL** macro), then you must have a default constructor (a constructor with no arguments) in your class declaration. If you do not need a default constructor, declare a private or protected "empty" constructor. For more information, see Using CObject.

The standard C++ default class copy constructor does a member-by-member copy. The presence of the private **CObject** copy constructor guarantees a compiler error message if the copy constructor of your class is needed but not available. You must therefore provide a copy constructor if your class requires this capability.

Example

See CObList::CObList for a listing of the `CAge` class used in the **CObject** examples.

```
// Create a CAge object using the default constructor.
CAge age1;

// Create a CAge object using the copy constructor.
CAge age2( age1 );
```

See Also

Reference

CObject Class

Hierarchy Chart

Other Resources

CObject Members

CObject::Dump

Dumps the contents of your object to a CDumpContext object.

```
virtual void Dump(
   CDumpContext& dc
) const;
```

Parameters

dc

The diagnostic dump context for dumping, usually **afxDump**.

Remarks

When you write your own class, you should override the **Dump** function to provide diagnostic services for yourself and other users of your class. The overridden **Dump** usually calls the **Dump** function of its base class before printing data members unique to the derived class. **CObject::Dump** prints the class name if your class uses the **IMPLEMENT_DYNAMIC** or **IMPLEMENT_SERIAL** macro.

Note
Your **Dump** function should not print a newline character at the end of its output.

Dump calls make sense only in the Debug version of the Microsoft Foundation Class Library. You should bracket calls, function declarations, and function implementations with **#ifdef _DEBUG**/**#endif** statements for conditional compilation.

Since **Dump** is a **const** function, you are not permitted to change the object state during the dump.

The CDumpContext insertion (<<) operator calls **Dump** when a **CObject** pointer is inserted.

Dump permits only "acyclic" dumping of objects. You can dump a list of objects, for example, but if one of the objects is the list itself, you will eventually overflow the stack.

Example

See CObList::CObList for a listing of the `CAge` class used in all **CObject** examples.

```
// example for CObject::Dump
void CAge::Dump( CDumpContext &dc ) const
 {
 CObject::Dump( dc );
 dc << "Age = " << m_years;
 }
```

Smart Device Developer Notes

This method is not supported in smart device projects because CDumpContext Class is not supported in smart device projects.

See Also

Reference

CObject Class

Hierarchy Chart

Other Resources

CObject Members

CObject::GetRuntimeClass

Returns the **CRuntimeClass** structure corresponding to this object's class.

```
virtual CRuntimeClass* GetRuntimeClass( ) const;
```

Return Value

A pointer to the CRuntimeClass structure corresponding to this object's class; never **NULL**.

Remarks

There is one **CRuntimeClass** structure for each **CObject**-derived class. The structure members are as follows:

- **LPCSTR m_lpszClassName** A null-terminated string containing the ASCII class name.
- **int m_nObjectSize** The size of the object, in bytes. If the object has data members that point to allocated memory, the size of that memory is not included.
- **UINT m_wSchema** The schema number (– 1 for nonserializable classes). See the IMPLEMENT_SERIAL macro for a description of schema number.
- **CObject* (PASCAL* m_pfnCreateObject)( )** A function pointer to the default constructor that creates an object of your class (valid only if the class supports dynamic creation; otherwise, returns **NULL**).
- **CRuntimeClass* (PASCAL* m_pfn_GetBaseClass)( )** If your application is dynamically linked to the AFXDLL version of MFC, a pointer to a function that returns the **CRuntimeClass** structure of the base class.
- **CRuntimeClass* m_pBaseClass** If your application is statically linked to MFC, a pointer to the **CRuntimeClass** structure of the base class.

This function requires use of the IMPLEMENT_DYNAMIC, IMPLEMENT_DYNCREATE, or IMPLEMENT_SERIAL macro in the class implementation. You will get incorrect results otherwise.

Example

See CObList::CObList for a listing of the `CAge` class used in all **CObject** examples.

```
// example for CObject::GetRuntimeClass
CAge a(21);
CRuntimeClass* prt = a.GetRuntimeClass();
ASSERT( strcmp( prt->m_lpszClassName, "CAge" )  == 0 );
```

See Also

Reference

CObject Class
Hierarchy Chart
CObject::IsKindOf
RUNTIME_CLASS

Other Resources

CObject Members

CObject::IsKindOf

Tests this object's relationship to a given class.

```
BOOL IsKindOf(
   const CRuntimeClass* pClass
) const;
```

Parameters

pClass

A pointer to a CRuntimeClass structure associated with your **CObject**-derived class.

Return Value

Nonzero if the object corresponds to the class; otherwise 0.

Remarks

This function tests *pClass* to see if (1) it is an object of the specified class or (2) it is an object of a class derived from the specified class. This function works only for classes declared with the DECLARE_DYNAMIC, DECLARE_DYNCREATE, or DECLARE_SERIAL macro.

Do not use this function extensively because it defeats the C++ polymorphism feature. Use virtual functions instead.

Example

See CObList::CObList for a listing of the `CAge` class used in all **CObject** examples.

```
// example for CObject::IsKindOf
CAge a(21);  // Must use IMPLEMENT_DYNAMIC, IMPLEMENT _DYNCREATE, or
             // IMPLEMENT_SERIAL
ASSERT( a.IsKindOf( RUNTIME_CLASS( CAge ) ) );
ASSERT( a.IsKindOf( RUNTIME_CLASS( CObject ) ) );
```

See Also

Tasks

Accessing Run-Time Class Information

Reference

CObject Class
Hierarchy Chart
CObject::GetRuntimeClass
RUNTIME_CLASS

Other Resources

CObject Members

CObject::IsSerializable

Tests whether this object is eligible for serialization.

```
BOOL IsSerializable( ) const;
```

Return Value

Nonzero if this object can be serialized; otherwise 0.

Remarks

For a class to be serializable, its declaration must contain the DECLARE_SERIAL macro, and the implementation must contain the IMPLEMENT_SERIAL macro.

Note
Do not override this function.

Example

See CObList::CObList for a listing of the `CAge` class used in all **CObject** examples.

```
// example for CObject::IsSerializable
CAge a(21);
ASSERT( a.IsSerializable() );
```

See Also

Reference

CObject Class
Hierarchy Chart
CObject::Serialize

Other Resources

CObject Members

CObject::Serialize

Reads or writes this object from or to an archive.

```
virtual void Serialize(
   CArchive& ar
);
```

Parameters

ar

A **CArchive** object to serialize to or from.

Remarks

You must override **Serialize** for each class that you intend to serialize. The overridden **Serialize** must first call the **Serialize** function of its base class.

You must also use the DECLARE_SERIAL macro in your class declaration, and you must use the IMPLEMENT_SERIAL macro in the implementation.

Use CArchive::IsLoading or CArchive::IsStoring to determine whether the archive is loading or storing.

Serialize is called by CArchive::ReadObject and CArchive::WriteObject. These functions are associated with the **CArchive** insertion operator (**<<**) and extraction operator (**>>**).

For serialization examples, see the article Serialization: Serializing an Object.

Example

See CObList::CObList for a listing of the `CAge` class used in all **CObject** examples.

```
// example for CObject::Serialize
void CAge::Serialize( CArchive& ar )
 {
 CObject::Serialize( ar );
     if( ar.IsStoring() )
     ar << m_years;
     else
     ar >> m_years;
 }
```

See Also

Reference

CObject Class

Hierarchy Chart

Other Resources

CObject Members

CObject Operators

For information about the operators in **CObject**, see CObject Members.

CObject::operator delete

For the Release version of the library, operator **delete** frees the memory allocated by operator **new**.

```
void PASCAL operator delete(
   void* p
);
void PASCAL operator delete(
   void* p,
      void* pPlace
);
void PASCAL operator delete(
   void* p,
   LPCSTR lpszFileName,
   int nLine
);
```

Remarks

In the Debug version, operator **delete** participates in an allocation-monitoring scheme designed to detect memory leaks.

If you use the code line

```
#define new DEBUG_NEW
```

before any of your implementations in a .CPP file, then the third version of **delete** will be used, storing the filename and line number in the allocated block for later reporting. You do not have to worry about supplying the extra parameters; a macro takes care of that for you.

Even if you do not use **DEBUG_NEW** in Debug mode, you still get leak detection, but without the source-file line-number reporting described above.

If you override operators **new** and **delete**, you forfeit this diagnostic capability.

Example

See CObList::CObList for a listing of the `CAge` class used in the **CObject** examples.

```
void CAge::operator delete(void* p)
{
   free(p);
}

void CAge::operator delete(void *p, LPCSTR lpszFileName, int nLine)
{
   free(p);
}
```

See Also

Reference

CObject Class

Hierarchy Chart

CObject::operator new

Other Resources

CObject Members

CObject::operator new

For the Release version of the library, operator **new** performs an optimal memory allocation in a manner similar to **malloc**.

```
void* PASCAL operator new(
   size_t nSize
);
void* PASCAL operator new(
   size_t,
   void* p
);
void* PASCAL operator new(
   size_t nSize,
   LPCSTR lpszFileName,
   int nLine
);
```

Remarks

In the Debug version, operator **new** participates in an allocation-monitoring scheme designed to detect memory leaks.

If you use the code line

```
#define new DEBUG_NEW
```

before any of your implementations in a .CPP file, then the second version of **new** will be used, storing the filename and line number in the allocated block for later reporting. You do not have to worry about supplying the extra parameters; a macro takes care of that for you.

Even if you do not use **DEBUG_NEW** in Debug mode, you still get leak detection, but without the source-file line-number reporting described above.

Note
If you override this operator, you must also override **delete**. Do not use the standard library **_new_handler** function.

Example

See CObList::CObList for a listing of the `CAge` class used in the **CObject** examples.

```
void* CAge::operator new(size_t nSize)
{
   return malloc(nSize);
}

void* CAge::operator new(size_t nSize, LPCSTR lpszFileName, int nLine)
{
   return malloc(nSize);
}
```

See Also

Reference

CObject Class

Hierarchy Chart

CObject::operator delete

Other Resources

CObject Members

CObList Class

Supports ordered lists of nonunique **CObject** pointers accessible sequentially or by pointer value.

```
class CObList : public CObject
```

Remarks

CObList lists behave like doubly-linked lists.

A variable of type **POSITION** is a key for the list. You can use a **POSITION** variable both as an iterator to traverse a list sequentially and as a bookmark to hold a place. A position is not the same as an index, however.

Element insertion is very fast at the list head, at the tail, and at a known **POSITION**. A sequential search is necessary to look up an element by value or index. This search can be slow if the list is long.

CObList incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. If a list of **CObject** pointers is stored to an archive, either with an overloaded insertion operator or with the **Serialize** member function, each **CObject** element is serialized in turn.

If you need a dump of individual **CObject** elements in the list, you must set the depth of the dump context to 1 or greater.

When a **CObList** object is deleted, or when its elements are removed, only the **CObject** pointers are removed, not the objects they reference.

You can derive your own classes from **CObList**. Your new list class, designed to hold pointers to objects derived from **CObject**, adds new data members and new member functions. Note that the resulting list is not strictly type safe, because it allows insertion of any **CObject** pointer.

Note
You must use the IMPLEMENT_SERIAL macro in the implementation of your derived class if you intend to serialize the list.

For more information on using **CObList**, see the article Collections.

Requirements

Header: afxcoll.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class
Hierarchy Chart
CStringList Class
CPtrList Class

Other Resources

CObList Members

CObList Members

Base Class Members

CObject Members

Construction

CObList	Constructs an empty list for **CObject** pointers.

Head/Tail Access

GetHead	Returns the head element of the list (cannot be empty).
GetTail	Returns the tail element of the list (cannot be empty).

Operations

AddHead	Adds an element (or all the elements in another list) to the head of the list (makes a new head).
AddTail	Adds an element (or all the elements in another list) to the tail of the list (makes a new tail).
RemoveAll	Removes all the elements from this list.
RemoveHead	Removes the element from the head of the list.
RemoveTail	Removes the element from the tail of the list.

Iteration

GetHeadPosition	Returns the position of the head element of the list.
GetNext	Gets the next element for iterating.
GetPrev	Gets the previous element for iterating.
GetTailPosition	Returns the position of the tail element of the list.

Retrieval/Modification

GetAt	Gets the element at a given position.
RemoveAt	Removes an element from this list, specified by position.
SetAt	Sets the element at a given position.

Insertion

InsertAfter	Inserts a new element after a given position.
InsertBefore	Inserts a new element before a given position.

Searching

Find	Gets the position of an element specified by pointer value.
FindIndex	Gets the position of an element specified by a zero-based index.

Status

GetSize	Returns the number of elements in this list.

GetCount	Returns the number of elements in this list.
IsEmpty	Tests for the empty list condition (no elements).

See Also

Reference

CObList Class

Hierarchy Chart

CObList Member Functions

For information about the member functions in **CObList**, see CObList Members.

CObList::AddHead

Adds a new element or list of elements to the head of this list.

```
POSITION AddHead(
   CObject* newElement
);
void AddHead(
   CObList* pNewList
);
```

Parameters

newElement

The **CObject** pointer to be added to this list.

pNewList

A pointer to another **CObList** list. The elements in *pNewList* will be added to this list.

Return Value

The first version returns the **POSITION** value of the newly inserted element.

The following table shows other member functions that are similar to **CObList::AddHead**.

Class	Member Function
CPtrList	**POSITION AddHead(void*** *newElement* **);** **void AddHead(CPtrList*** *pNewList* **);**
CStringList	**POSITION AddHead(const CString&** *newElement* **);** **POSITION AddHead(LPCTSTR** *newElement* **);** **void AddHead(CStringList*** *pNewList* **);**

Remarks

The list can be empty before the operation.

Example

See CObList::CObList for a listing of the `CAge` class.

```
   CObList list;
   list.AddHead( new CAge( 21 ) ); // 21 is now at head.
   list.AddHead( new CAge( 40 ) ); // 40 replaces 21 at head.
#ifdef _DEBUG
   afxDump.SetDepth( 1 );
   afxDump << "AddHead example: " << &list << "\n";
#endif
```

The results from this program are as follows:

```
AddHead example: A CObList with 2 elements
    a CAge at $44A8 40
    a CAge at $442A 21
```

See Also

Reference

CObList Class

Hierarchy Chart

CObList::GetHead
CObList::RemoveHead

Other Resources

CObList Members

CObList::AddTail

Adds a new element or list of elements to the tail of this list.

```
POSITION AddTail(
   CObject* newElement
);
void AddTail(
   CObList* pNewList
);
```

Parameters

newElement

The **CObject** pointer to be added to this list.

pNewList

A pointer to another **CObList** list. The elements in *pNewList* will be added to this list.

Return Value

The first version returns the **POSITION** value of the newly inserted element.

Remarks

The list can be empty before the operation.

The following table shows other member functions that are similar to **CObList::AddTail**.

Class	Member Function
CPtrList	**POSITION AddTail(void*** *newElement* **);** **void AddTail(CPtrList*** *pNewList* **);**
CStringList	**POSITION AddTail(const CString&** *newElement* **);** **POSITION AddTail(LPCTSTR** *newElement* **);** **void AddTail(CStringList*** *pNewList* **);**

Example

See CObList::CObList for a listing of the `CAge` class.

```
   CObList list;
   list.AddTail( new CAge( 21 ) );
   list.AddTail( new CAge( 40 ) ); // List now contains (21, 40).
#ifdef _DEBUG
   afxDump.SetDepth( 1 );
   afxDump << "AddTail example: " << &list << "\n";
#endif
```

The results from this program are as follows:

```
AddTail example: A CObList with 2 elements
     a CAge at $444A 21
     a CAge at $4526 40
```

See Also

Reference

CObList Class

Hierarchy Chart

CObList::GetTail
CObList::RemoveTail

Other Resources

CObList Members

CObList::CObList

Constructs an empty **CObject** pointer list.

```
CObList(
   INT_PTR nBlockSize = 10
);
```

Parameters

nBlockSize

The memory-allocation granularity for extending the list.

Remarks

As the list grows, memory is allocated in units of *nBlockSize* entries. If a memory allocation fails, a **CMemoryException** is thrown.

The following table shows other member functions that are similar to **CObList::CObList**.

Class	Member Function
CPtrList	**CPtrList(INT_PTR** *nBlockSize* **= 10);**
CStringList	**CStringList(INT_PTR** *nBlockSize* **= 10);**

Example

Below is a listing of the **CObject**-derived class `CAge` used in all the collection examples:

```
// Simple CObject-derived class for CObList and other examples
class CAge : public CObject
{
    DECLARE_SERIAL( CAge )
private:
    int   m_years;
public:
    CAge() { m_years = 0; }
    CAge( int age ) { m_years = age; }
    CAge( const CAge& a ) { m_years = a.m_years; } // Copy constructor
    void Serialize( CArchive& ar);
    void AssertValid() const;
    const CAge& operator=( const CAge& a )
    {
        m_years = a.m_years; return *this;
    }
    BOOL operator==(CAge a)
    {
        return m_years == a.m_years;
    }
    void* operator new(size_t nSize);
    void* operator new(size_t nSize, LPCSTR lpszFileName, int nLine);
    void operator delete(void* p);
    void operator delete(void* p, LPCSTR lpszFileName, int nLine);
#ifdef _DEBUG
    void Dump( CDumpContext& dc ) const
    {
        CObject::Dump( dc );
        dc << m_years;
    }
#endif
};
```

Below is an example of **CObList** constructor usage:

```
CObList list( 20 );  // List on the stack with blocksize = 20.

CObList* plist = new CObList; // List on the heap with default
                              // blocksize.
```

See Also

Reference

CObList Class

Hierarchy Chart

Other Resources

CObList Members

CObList::Find

Searches the list sequentially to find the first **CObject** pointer matching the specified **CObject** pointer.

```
POSITION Find(
   CObject* searchValue,
   POSITION startAfter = NULL
) const;
```

Parameters

searchValue

The object pointer to be found in this list.

startAfter

The start position for the search.

Return Value

A **POSITION** value that can be used for iteration or object pointer retrieval; **NULL** if the object is not found.

Remarks

Note that the pointer values are compared, not the contents of the objects.

The following table shows other member functions that are similar to **CObList::Find**.

Class	Member Function
CPtrList	**POSITION Find(void*** *searchValue* **, POSITION** *startAfter* **= NULL) const;**
CStringList	**POSITION Find(LPCTSTR** *searchValue* **, POSITION** *startAfter* **= NULL) const;**

Example

See CObList::CObList for a listing of the `CAge` class.

```
CObList list;
CAge* pa1;
CAge* pa2;
POSITION pos;
list.AddHead( pa1 = new CAge( 21 ) );
list.AddHead( pa2 = new CAge( 40 ) );    // List now contains (40, 21).
if( ( pos = list.Find( pa1 ) ) != NULL ) // Hunt for pa1
{                                        // starting at head by default.
    ASSERT( *(CAge*) list.GetAt( pos ) == CAge( 21 ) );
}
```

See Also

Reference

CObList Class
Hierarchy Chart
CObList::GetNext
CObList::GetPrev

Other Resources

CObList Members

CObList::FindIndex

Uses the value of *nIndex* as an index into the list.

```
POSITION FindIndex(
   INT_PTR nIndex
) const;
```

Parameters

nIndex

The zero-based index of the list element to be found.

Return Value

A **POSITION** value that can be used for iteration or object pointer retrieval; **NULL** if *nIndex* is too large. (The framework generates an assertion if *nIndex* is negative.)

Remarks

It starts a sequential scan from the head of the list, stopping on the *n*th element.

The following table shows other member functions that are similar to **CObList::FindIndex**.

Class	Member Function
CPtrList	**POSITION FindIndex(INT_PTR** *nIndex* **) const;**
CStringList	**POSITION FindIndex(INT_PTR** *nIndex* **) const;**

Example

See CObList::CObList for a listing of the `CAge` class.

```
CObList list;
POSITION pos;

list.AddHead( new CAge( 21 ) );
list.AddHead( new CAge( 40 ) ); // List now contains (40, 21).
if( ( pos = list.FindIndex( 0 )) != NULL )
{
    ASSERT( *(CAge*) list.GetAt( pos ) == CAge( 40 ) );
}
```

See Also

Reference

CObList Class
Hierarchy Chart
CObList::Find
CObList::GetNext
CObList::GetPrev

Other Resources

CObList Members

CObList::GetAt

A variable of type **POSITION** is a key for the list.

```
CObject*& GetAt(
   POSITION position
);
const CObject*& GetAt(
   POSITION position
) const;
```

Parameters

position

A **POSITION** value returned by a previous **GetHeadPosition** or **Find** member function call.

Return Value

See the return value description for GetHead.

Remarks

It is not the same as an index, and you cannot operate on a **POSITION** value yourself. **GetAt** retrieves the **CObject** pointer associated with a given position.

You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

The following table shows other member functions that are similar to **CObList::GetAt**.

Class	Member Function
CPtrList	**const void*& GetAt(POSITION** *position* **) const;** **void*& GetAt(POSITION** *position* **);**
CStringList	**const CString& GetAt(POSITION** *position* **) const;** **CString& GetAt(POSITION** *position* **);**

Example

See the example for FindIndex.

See Also

Reference

CObList Class
Hierarchy Chart
CObList::Find
CObList::SetAt
CObList::GetNext
CObList::GetPrev
CObList::GetHead

Other Resources

CObList Members

CObList::GetCount

Gets the number of elements in this list.

```
INT_PTR GetCount( ) const;
```

Return Value

An integer value containing the element count.

The following table shows other member functions that are similar to **CObList::GetCount**.

Class	Member Function
CPtrList	**INT_PTR GetCount( ) const;**
CStringList	**INT_PTR GetCount( ) const;**

Example

See CObList::CObList for a listing of the `CAge` class.

```
COblist list;

list.AddHead( new CAge( 21 ) );
list.AddHead( new CAge( 40 ) ); // List now contains (40, 21).
ASSERT( list.GetCount() == 2 );
```

See Also

Reference

CObList Class

Hierarchy Chart

CObList::IsEmpty

Other Resources

CObList Members

CObList::GetHead

Gets the **CObject** pointer that represents the head element of this list.

```
CObject*& GetHead( );
const CObject*& GetHead( ) const;
```

Return Value

If the list is accessed through a pointer to a **const CObList**, then **GetHead** returns a **CObject** pointer. This allows the function to be used only on the right side of an assignment statement and thus protects the list from modification.

If the list is accessed directly or through a pointer to a **CObList**, then **GetHead** returns a reference to a **CObject** pointer. This allows the function to be used on either side of an assignment statement and thus allows the list entries to be modified.

Remarks

You must ensure that the list is not empty before calling **GetHead**. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use IsEmpty to verify that the list contains elements.

The following table shows other member functions that are similar to **CObList::GetHead**.

Class	Member Function
CPtrList	**const void*& GetHead( ) const; void*& GetHead( );**
CStringList	**const CString& GetHead( ) const; CString& GetHead( );**

Example

See CObList::CObList for a listing of the `CAge` class.

The following example illustrates the use of **GetHead** on the left side of an assignment statement.

```
    const CObList* cplist;

    CObList* plist = new CObList;
    CAge* page1 = new CAge( 21 );
    CAge* page2 = new CAge( 30 );
    CAge* page3 = new CAge( 40 );
    plist->AddHead( page1 );
    plist->AddHead( page2 );  // List now contains (30, 21).
    // The following statement REPLACES the head element.
    plist->GetHead() = page3; // List now contains (40, 21).
    ASSERT( *(CAge*) plist->GetHead() == CAge( 40 ) );
    cplist = plist;  // cplist is a pointer to a const list.
// cplist->GetHead() = page3; // Error: can't assign a pointer to a const list
    ASSERT( *(CAge*) plist->GetHead() == CAge( 40 ) ); // OK

    delete page1;
    delete page2;
    delete page3;
    delete plist; // Cleans up memory.
```

See Also

Reference

CObList Class
Hierarchy Chart
CObList::GetTail
CObList::GetTailPosition
CObList::AddHead
CObList::RemoveHead

Other Resources

CObList::GetHeadPosition

Gets the position of the head element of this list.

```
POSITION GetHeadPosition( ) const;
```

Return Value

A **POSITION** value that can be used for iteration or object pointer retrieval; **NULL** if the list is empty.

The following table shows other member functions that are similar to **CObList::GetHeadPosition**.

Class	Member Function
CPtrList	**POSITION GetHeadPosition( ) const;**
CStringList	**POSITION GetHeadPosition( ) const;**

Example

See CObList::CObList for a listing of the `CAge` class.

```
CObList list;
POSITION pos;

list.AddHead( new CAge( 21 ) );
list.AddHead( new CAge( 40 ) ); // List now contains (40, 21).
if( ( pos = list.GetHeadPosition() ) != NULL )
{
    ASSERT( *(CAge*) list.GetAt( pos ) == CAge( 40 ) );
}
```

See Also

Reference

CObList Class

Hierarchy Chart

CObList::GetTailPosition

Other Resources

CObList Members

CObList::GetNext

Gets the list element identified by *rPosition*, then sets *rPosition* to the **POSITION** value of the next entry in the list.

```
CObject*& GetNext(
   POSITION& rPosition
);
const CObject* GetNext(
   POSITION& rPosition
) const;
```

Parameters

rPosition

A reference to a **POSITION** value returned by a previous **GetNext**, **GetHeadPosition**, or other member function call.

Return Value

See the return value description for GetHead.

Remarks

You can use **GetNext** in a forward iteration loop if you establish the initial position with a call to **GetHeadPosition** or `Find`.

You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

If the retrieved element is the last in the list, then the new value of *rPosition* is set to **NULL**.

It is possible to remove an element during an iteration. See the example for RemoveAt.

Note

As of MFC 8.0 the const version of this method has changed to return `const CObject*` instead of `const CObject*&`. This change was made to bring the compiler into conformance with the C++ standard.

The following table shows other member functions that are similar to **CObList::GetNext**.

Class	Member Function
CPtrList	`void*& GetNext( POSITION& rPosition );` `const void* GetNext( POSITION& rPosition ) const;`
CStringList	`CString& GetNext( POSITION& rPosition );` `const CString& GetNext( POSITION& rPosition ) const;`

Example

See CObList::CObList for a listing of the `CAge` class.

```
   CObList list;
   POSITION pos;
   list.AddHead( new CAge( 21 ) );
   list.AddHead( new CAge( 40 ) ); // List now contains (40, 21).
   // Iterate through the list in head-to-tail order.
#ifdef _DEBUG
   for( pos = list.GetHeadPosition(); pos != NULL; )
   {
    afxDump << list.GetNext( pos ) << "\n";
   }
#endif
```

The results from this program are as follows:

```
a CAge at $479C 40
a CAge at $46C0 21
```

See Also

Reference

CObList Class
Hierarchy Chart
CObList::Find
CObList::GetHeadPosition
CObList::GetTailPosition
CObList::GetPrev
CObList::GetHead

Other Resources

CObList Members

CObList::GetPrev

Gets the list element identified by *rPosition*, then sets *rPosition* to the **POSITION** value of the previous entry in the list.

```
COBJECT*& GetPrev(
   POSITION& rPosition
);
const CObject* GetPrev(
   POSITION& rPosition
) const;
```

Parameters

rPosition

A reference to a **POSITION** value returned by a previous **GetPrev** or other member function call.

Return Value

See the return value description for GetHead.

Remarks

You can use **GetPrev** in a reverse iteration loop if you establish the initial position with a call to **GetTailPosition** or `Find`.

You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

If the retrieved element is the first in the list, then the new value of *rPosition* is set to **NULL**.

Note
As of MFC 8.0 the const version of this method has changed to return `const CObject*` instead of `const CObject*&`. This change was made to bring the compiler into conformance with the C++ standard.

The following table shows other member functions that are similar to **CObList::GetPrev**.

Class	Member Function
CPtrList	`void*& GetPrev( POSITION& rPosition );` `const void* GetPrev( POSITION& rPosition ) const;`
CStringList	`CString& GetPrev( POSITION& rPosition );` `const CString& GetPrev( POSITION& rPosition ) const;`

Example

See CObList::CObList for a listing of the `CAge` class.

```
    CObList list;
    POSITION pos;

    list.AddHead( new CAge(21) );
    list.AddHead( new CAge(40) ); // List now contains (40, 21).
    // Iterate through the list in tail-to-head order.
    for( pos = list.GetTailPosition(); pos != NULL; )
    {
#ifdef _DEBUG
     afxDump << list.GetPrev( pos ) << "\n";
#endif
    }
```

The results from this program are as follows:

```
a CAge at $421C 21
a CAge at $421C 40
```

See Also

Reference

CObList Class
Hierarchy Chart
CObList::Find
CObList::GetTailPosition
CObList::GetHeadPosition
CObList::GetNext
CObList::GetHead

Other Resources

CObList Members

CObList::GetSize

Returns the number of list elements.

```
INT_PTR GetSize( ) const;
```

Return Value

The number of items in the list.

Remarks

Call this method to retrieve the number of elements in the list.

The following table shows other member functions that are similar to **CObList::GetSize**.

Class	Member Function
CPtrList	**INT_PTR GetSize( ) const;**
CStringList	**INT_PTR GetSize( ) const;**

Example

See CObList::CObList for a listing of the `CAge` class.

```
CObList list;

list.AddHead( new CAge( 21 ) );
list.AddHead( new CAge( 40 ) ); // List now contains (40, 21).
ASSERT( list.GetSize() == 2 );
```

See Also

Reference

CObList Class
Hierarchy Chart
CObList::GetCount

Other Resources

CObList Members

CObList::GetTail

Gets the **CObject** pointer that represents the tail element of this list.

```
CObject*& GetTail( );
const CObject*& GetTail( ) const;
```

Return Value

See the return value description for GetHead.

Remarks

You must ensure that the list is not empty before calling **GetTail**. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use IsEmpty to verify that the list contains elements.

The following table shows other member functions that are similar to **CObList::GetTail**.

Class	Member Function
CPtrList	**const void*& GetTail( ) const; void*& GetTail( );**
CStringList	**const CString& GetTail( ) const; CString& GetTail( );**

Example

See CObList::CObList for a listing of the `CAge` class.

```
CObList list;

list.AddHead( new CAge( 21 ) );
list.AddHead( new CAge( 40 ) ); // List now contains (40, 21).
ASSERT( *(CAge*) list.GetTail() == CAge( 21 ) );
```

See Also

Reference

CObList Class
Hierarchy Chart
CObList::AddTail
CObList::AddHead
CObList::RemoveHead
CObList::GetHead

Other Resources

CObList Members

CObList::GetTailPosition

Gets the position of the tail element of this list; **NULL** if the list is empty.

```
POSITION GetTailPosition( ) const;
```

Return Value

A **POSITION** value that can be used for iteration or object pointer retrieval; **NULL** if the list is empty.

The following table shows other member functions that are similar to **CObList::GetTailPosition**.

Class	Member Function
CPtrList	**POSITION GetTailPosition( ) const;**
CStringList	**POSITION GetTailPosition( ) const;**

Example

See CObList::CObList for a listing of the `CAge` class.

```
COblist list;
POSITION pos;

list.AddHead( new CAge( 21 ) );
list.AddHead( new CAge( 40 ) ); // List now contains (40, 21).
if( ( pos = list.GetTailPosition() ) != NULL )
{
    ASSERT( *(CAge*) list.GetAt( pos ) == CAge( 21 ) );
}
```

See Also

Reference

CObList Class
Hierarchy Chart
CObList::GetHeadPosition
CObList::GetTail

Other Resources

CObList Members

CObList::InsertAfter

Adds an element to this list after the element at the specified position.

```
POSITION InsertAfter(
   POSITION position,
   CObject* newElement
);
```

Parameters

position

A **POSITION** value returned by a previous **GetNext**, **GetPrev**, or **Find** member function call.

newElement

The object pointer to be added to this list.

The following table shows other member functions that are similar to **CObList::InsertAfter**.

Class	Member Function
CPtrList	**POSITION InsertAfter(POSITION** *position* **, void*** *newElement* **);**
CStringList	**POSITION InsertAfter(POSITION** *position* **, const CString&** *newElement* **);** **POSITION InsertAfter(POSITION** *position* **, LPCTSTR** *newElement* **);**

Return Value

A **POSITION** value which is the same as the *position* parameter.

Example

See CObList::CObList for a listing of the `CAge` class.

```
   CObList list;
   POSITION pos1, pos2;
   list.AddHead( new CAge( 21 ) );
   list.AddHead( new CAge( 40 ) ); // List now contains (40, 21).
   if( ( pos1 = list.GetHeadPosition() ) != NULL )
   {
       pos2 = list.InsertAfter( pos1, new CAge( 65 ) );
   }
#ifdef _DEBUG
   afxDump.SetDepth( 1 );
   afxDump << "InsertAfter example: " << &list << "\n";
#endif
```

The results from this program are as follows:

```
InsertAfter example: A CObList with 3 elements
     a CAge at $4A44 40
     a CAge at $4A64 65
     a CAge at $4968 21
```

See Also

Reference

CObList Class

Hierarchy Chart

CObList::Find

CObList::InsertBefore

Other Resources

CObList::InsertBefore

Adds an element to this list before the element at the specified position.

```
POSITION InsertBefore(
   POSITION position,
   CObject* newElement
);
```

Parameters

position

A **POSITION** value returned by a previous **GetNext**, **GetPrev**, or **Find** member function call.

newElement

The object pointer to be added to this list.

Return Value

A **POSITION** value that can be used for iteration or object pointer retrieval; **NULL** if the list is empty.

The following table shows other member functions that are similar to **CObList::InsertBefore**.

Class	Member Function
CPtrList	**POSITION InsertBefore(POSITION** *position* **, void*** *newElement* **);**
CStringList	**POSITION InsertBefore(POSITION** *position* **, const CString&** *newElement* **);** **POSITION InsertBefore(POSITION** *position* **, LPCTSTR** *newElement* **);**

Example

See CObList::CObList for a listing of the `CAge` class.

```
   CObList list;
   POSITION pos1, pos2;
   list.AddHead( new CAge( 21 ) );
   list.AddHead( new CAge( 40 ) ); // List now contains (40, 21).
   if( ( pos1 = list.GetTailPosition() ) != NULL )
   {
       pos2 = list.InsertBefore( pos1, new CAge( 65 ) );
   }
#ifdef _DEBUG
   afxDump.SetDepth( 1 );
   afxDump << "InsertBefore example: " << &list << "\n";
#endif
```

The results from this program are as follows:

```
InsertBefore example: A CObList with 3 elements
    a CAge at $4AE2 40
    a CAge at $4B02 65
    a CAge at $49E6 21
```

See Also

Reference

CObList Class

Hierarchy Chart

CObList::Find

CObList::InsertAfter

Other Resources

CObList::IsEmpty

Indicates whether this list contains no elements.

```
BOOL IsEmpty( ) const;
```

Return Value

Nonzero if this list is empty; otherwise 0.

The following table shows other member functions that are similar to **CObList::IsEmpty**.

Class	Member Function
CPtrList	**BOOL IsEmpty( ) const;**
CStringList	**BOOL IsEmpty( ) const;**

Example

See the example for RemoveAll.

See Also

Reference

CObList Class
Hierarchy Chart
CObList::GetCount

Other Resources

CObList Members

CObList::RemoveAll

Removes all the elements from this list and frees the associated **CObList** memory.

```
void RemoveAll( );
```

Remarks

No error is generated if the list is already empty.

When you remove elements from a **CObList**, you remove the object pointers from the list. It is your responsibility to delete the objects themselves.

The following table shows other member functions that are similar to **CObList::RemoveAll**.

Class	Member Function
CPtrList	**void RemoveAll( );**
CStringList	**void RemoveAll( );**

Example

See CObList::CObList for a listing of the `CAge` class.

```
CObList list;
CAge* pa1;
CAge* pa2;
ASSERT( list.IsEmpty()); // Yes it is.
list.AddHead( pa1 = new CAge( 21 ) );
list.AddHead( pa2 = new CAge( 40 ) ); // List now contains (40, 21).
ASSERT( !list.IsEmpty()); // No it isn't.
list.RemoveAll(); // CAge's aren't destroyed.
ASSERT( list.IsEmpty()); // Yes it is.
delete pa1;     // Now delete the CAge objects.
delete pa2;
```

See Also

Reference

CObList Class

Hierarchy Chart

Other Resources

CObList Members

CObList::RemoveAt

Removes the specified element from this list.

```
void RemoveAt(
   POSITION position
);
```

Parameters

position

The position of the element to be removed from the list.

Remarks

When you remove an element from a **CObList**, you remove the object pointer from the list. It is your responsibility to delete the objects themselves.

You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

The following table shows other member functions that are similar to **CObList::RemoveAt**.

Class	Member Function
CPtrList	**void RemoveAt(POSITION** *position* **);**
CStringList	**void RemoveAt(POSITION** *position* **);**

Example

Be careful when removing an element during a list iteration. The following example shows a removal technique that guarantees a valid **POSITION** value for GetNext.

See CObList::CObList for a listing of the `CAge` class.

```
    CObList list;
    POSITION pos1, pos2;
    CObject* pa;

    list.AddHead( new CAge( 21 ) );
    list.AddHead( new CAge( 40 ) );
    list.AddHead( new CAge( 65 ) ); // List now contains (65 40, 21).
    for( pos1 = list.GetHeadPosition(); ( pos2 = pos1 ) != NULL; )
    {
        if( *(CAge*) list.GetNext( pos1 ) == CAge( 40 ) )
        {
            pa = list.GetAt( pos2 ); // Save the old pointer for
                                     //deletion.
            list.RemoveAt( pos2 );
            delete pa; // Deletion avoids memory leak.
        }
    }
#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "RemoveAt example: " << &list << "\n";
#endif
```

The results from this program are as follows:

```
RemoveAt example: A CObList with 2 elements
    a CAge at $4C1E 65
    a CAge at $4B22 21
```

See Also

Reference

CObList Class

Hierarchy Chart

Other Resources

CObList Members

CObList::RemoveHead

Removes the element from the head of the list and returns a pointer to it.

```
CObject* RemoveHead( );
```

Return Value

The **CObject** pointer previously at the head of the list.

Remarks

You must ensure that the list is not empty before calling **RemoveHead**. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use IsEmpty to verify that the list contains elements.

The following table shows other member functions that are similar to **CObList::RemoveHead**.

Class	Member Function
CPtrList	**void* RemoveHead( );**
CStringList	**CString RemoveHead( );**

Example

See CObList::CObList for a listing of the `CAge` class.

```
CObList list;
CAge* pa1;
CAge* pa2;

list.AddHead( pa1 = new CAge( 21 ) );
list.AddHead( pa2 = new CAge( 40 ) ); // List now contains (40, 21).
ASSERT( *(CAge*) list.RemoveHead() == CAge( 40 ) );  // Old head
ASSERT( *(CAge*) list.GetHead() == CAge( 21 ) );  // New head
delete pa1;
delete pa2;
```

See Also

Reference

CObList Class
Hierarchy Chart
CObList::GetHead
CObList::AddHead

Other Resources

CObList Members

CObList::RemoveTail

Removes the element from the tail of the list and returns a pointer to it.

```
CObject* RemoveTail( );
```

Return Value

A pointer to the object that was at the tail of the list.

Remarks

You must ensure that the list is not empty before calling **RemoveTail**. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use IsEmpty to verify that the list contains elements.

The following table shows other member functions that are similar to **CObList::RemoveTail**.

Class	Member Function
CPtrList	**void* RemoveTail( );**
CStringList	**CString RemoveTail( );**

Example

See CObList::CObList for a listing of the `CAge` class.

```
CObList list;
CAge* pa1;
CAge* pa2;

list.AddHead( pa1 = new CAge( 21 ) );
list.AddHead( pa2 = new CAge( 40 ) ); // List now contains (40, 21).
ASSERT( *(CAge*) list.RemoveTail() == CAge( 21 ) );  // Old tail
ASSERT( *(CAge*) list.GetTail() == CAge( 40 ) );  // New tail
delete pa1;
delete pa2; // Clean up memory.
```

See Also

Reference

CObList Class
Hierarchy Chart
CObList::GetTail
CObList::AddTail

Other Resources

CObList Members

CObList::SetAt

Sets the element at a given position.

```
void SetAt(
   POSITION pos,
   CObject* newElement
);
```

Parameters

pos

The **POSITION** of the element to be set.

newElement

The **CObject** pointer to be written to the list.

Remarks

A variable of type **POSITION** is a key for the list. It is not the same as an index, and you cannot operate on a **POSITION** value yourself. **SetAt** writes the **CObject** pointer to the specified position in the list.

You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

The following table shows other member functions that are similar to **CObList::SetAt**.

Class	Member Function
CPtrList	**void SetAt(POSITION** *pos* **, const CString&** *newElement* **);**
CStringList	**void SetAt(POSITION** *pos* **, LPCTSTR** *newElement* **);**

Example

See CObList::CObList for a listing of the `CAge` class.

```
    CObList list;
    CObject* pa;
    POSITION pos;

    list.AddHead( new CAge( 21 ) );
    list.AddHead( new CAge( 40 ) ); // List now contains (40, 21).
    if( ( pos = list.GetTailPosition()) != NULL )
    {
        pa = list.GetAt( pos ); // Save the old pointer for
                                //deletion.
        list.SetAt( pos, new CAge( 65 ) );  // Replace the tail
                                            //element.
        delete pa;  // Deletion avoids memory leak.
    }
#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "SetAt example: " << &list << "\n";
#endif
```

The results from this program are as follows:

```
SetAt example: A CObList with 2 elements
    a CAge at $4D98 40
    a CAge at $4DB8 65
```

See Also

Reference

CObList Class
Hierarchy Chart
CObList::Find
CObList::GetAt
CObList::GetNext
CObList::GetPrev

Other Resources

CObList Members

COccManager Class

Manages various custom control sites; implemented by **COleControlContainer** and **COleControlSite** objects.

```
class COccManager : public CNoTrackObject
```

Remarks

The base class, **CNoTrackObject**, is an undocumented base class (located in AFXTLS.H). Designed for use by the MFC framework, classes derived from the **CNoTrackObject** class are exempt from memory leak detection. It is not recommended that you derive directly from **CNoTrackObject**.

Requirements

Header: afxocc.h

See Also

Reference

Hierarchy Chart
COleControlSite Class
COleControlContainer Class

Other Resources

COccManager Members

COccManager Members

Operations

GetDefBtnCode	Retrieves the code of the default button.
IsDialogMessage	Determines the target of a dialog message.
IsLabelControl	Determines if the specified control is a label control.
IsMatchingMnemonic	Determines if the current mnemonic matches the mnemonic of the specified control.
SetDefaultButton	Toggles the default state of the specified control.

Overridables

CreateContainer	Creates a **COleContainer** object.
CreateDlgControls	Creates ActiveX controls, hosted by the associated **COleContainer** object.
CreateSite	Creates a **COleClientSite** object.
OnEvent	Attempts to handle the specified event.
PostCreateDialog	Frees resources allocated during dialog creation.
PreCreateDialog	Processes a dialog template for ActiveX controls.
SplitDialogTemplate	Separates any existing ActiveX controls from common controls in the specified dialog template.

See Also

Reference

COccManager Class

Hierarchy Chart

COccManager Member Functions

For information about the member functions in **COccManager**, see COccManager Members.

COccManager::CreateContainer

Called by the framework to create a control container.

```
virtual COleControlContainer* CreateContainer(
   CWnd* pWnd
);
```

Parameters

pWnd

A pointer to the window object associated with the custom site container.

Return Value

A pointer to the newly created container; otherwise **NULL**.

Remarks

For more information on creating custom sites, see COleControlContainer::AttachControlSite.

See Also

Reference

COccManager Class

Hierarchy Chart

Other Resources

COccManager Members

COccManager::CreateDlgControls

Call this function to create ActiveX controls specified by the *pOccDialogInfo* parameter.

```
virtual BOOL CreateDlgControls(
   CWnd* pWndParent,
   LPCTSTR lpszResourceName,
   _AFX_OCC_DIALOG_INFO* pOccDialogInfo
);
virtual BOOL CreateDlgControls(
   CWnd* pWndParent,
   void* lpResource,
   _AFX_OCC_DIALOG_INFO* pOccDialogInfo
);
```

Parameters

pWndParent

A pointer to the parent of the dialog object.

lpszResourceName

The name of the resource being created.

pOccDialogInfo

A pointer to the dialog template used to create the dialog object.

lpResource

A pointer to a resource.

Return Value

Nonzero if the control was created successfully; otherwise zero.

See Also

Reference

COccManager Class

Hierarchy Chart

Other Resources

COccManager Members

COccManager::CreateSite

Called by the framework to create a control site, hosted by the container pointed to by *pCtrlCont*.

```
virtual COleControlSite* CreateSite(
   COleControlContainer* pCtrlCont
);
```

Parameters

pCtrlCont

A pointer to the control container hosting the new control site.

Return Value

A pointer to the newly created control site.

Remarks

Override this function to create a custom control site, using your COleControlSite-derived class.

Each control container can host multiple sites. Create additional sites with multiple calls to **CreateSite**.

See Also

Reference

COccManager Class

Hierarchy Chart

Other Resources

COccManager Members

COccManager::GetDefBtnCode

Call this function to determine if the control is a default push button.

```
static DWORD AFX_CDECL GetDefBtnCode(
   CWnd* pWnd
);
```

Parameters

pWnd

The window object containing the button control.

Return Value

One of the following values:

- **DLGC_DEFPUSHBUTTON** Control is the default button in the dialog.
- **DLGC_UNDEFPUSHBUTTON** Control is not the default button in the dialog.
- **0** Control is not a button.

See Also

Reference

COccManager Class

Hierarchy Chart

Other Resources

COccManager Members

COccManager::IsDialogMessage

Called by the framework to determine whether a message is intended for the specified dialog box and, if it is, processes the message.

```
virtual BOOL IsDialogMessage(
   CWnd* pWndDlg,
   LPMSG lpMsg
);
```

Parameters

pWndDlg

A pointer to the intended target dialog of the message.

lpMsg

A pointer to an **MSG** structure that contains the message to be checked.

Return Value

Nonzero if the message is processed; otherwise zero.

Remarks

The default behavior of **IsDialogMessage** is to check for keyboard messages and convert them into selections for the corresponding dialog box. For example, the TAB key, when pressed, selects the next control or group of controls.

Override this function to provide custom behavior for messages sent to the specified dialog.

See Also

Reference

COccManager Class

Hierarchy Chart

Other Resources

COccManager Members

COccManager::IsLabelControl

Call this function to determine if the specified control is a label control.

```
static BOOL AFX_CDECL IsLabelControl(
   CWnd* pWnd
);
static BOOL AFX_CDECL IsLabelControl(
   COleControlSiteOrWnd* pWnd
);
```

Parameters

pWnd

A pointer to the window containing the control.

Return Value

Nonzero if the control is a label; otherwise zero

Remarks

A label control is one that acts like a label for whatever control is next in the ordering.

See Also

Reference

COccManager Class
Hierarchy Chart
OLEMISC

Other Resources

COccManager Members

COccManager::IsMatchingMnemonic

Call this function to determine if the current mnemonic matches that represented by the control.

```
static BOOL AFX_CDECL IsMatchingMnemonic(
   CWnd* pWnd,
   LPMSG lpMsg
);
static BOOL AFX_CDECL IsMatchingMnemonic(
COleControlSiteOrWnd* pWnd,
   LPMSG lpMsg
);
```

Parameters

pWnd

A pointer to the window containing the control.

lpMsg

A pointer to the message containing the mnemonic to match.

Return Value

Nonzero if the mnemonic matches the control; otherwise zero

Remarks

See Also

Reference

COccManager Class

Hierarchy Chart

Other Resources

COccManager Members

COccManager::OnEvent

Called by the framework to handle the specified event.

```
virtual BOOL OnEvent(
   CCmdTarget* pCmdTarget,
   UINT idCtrl,
   AFX_EVENT* pEvent,
   AFX_CMDHANDLERINFO* pHandlerInfo
);
```

Parameters

pCmdTarget

A pointer to the **CCmdTarget** object attempting to handle the event

idCtrl

The resource ID of the control.

pEvent

The event being handled.

pHandlerInfo

If not **NULL**, **OnEvent** fills in the **pTarget** and **pmf** members of the **AFX_CMDHANDLERINFO** structure instead of dispatching the command. Typically, this parameter should be **NULL**.

Return Value

Nonzero if the event was handled, otherwise zero.

Remarks

Override this function to customize the default event-handling process.

See Also

Reference

COccManager Class

Hierarchy Chart

Other Resources

COccManager Members

COccManager::PreCreateDialog

Called by the framework to process a dialog template for ActiveX controls before creating the actual dialog box.

```
virtual const DLGTEMPLATE* PreCreateDialog(
   _AFX_OCC_DIALOG_INFO* pOccDialogInfo,
   const DLGTEMPLATE* pOrigTemplate
);
```

Parameters

pOccDialogInfo

An **_AFX_OCC_DIALOG_INFO** structure containing information on the dialog template and any ActiveX controls hosted by the dialog.

pOrigTemplate

A pointer to the dialog template to be used in creating the dialog box.

Return Value

A pointer to a dialog template structure used to create the dialog box.

Remarks

The default behavior makes a call to **SplitDialogTemplate**, determining if there are any ActiveX controls present and then returns the resultant dialog template.

Override this function to customize the process of creating a dialog box hosting ActiveX controls.

See Also

Reference

COccManager Class
Hierarchy Chart
COccManager::SplitDialogTemplate
COccManager::PostCreateDialog

Other Resources

COccManager Members

COccManager::PostCreateDialog

Called by the framework to free memory allocated for the dialog template.

```
virtual void PostCreateDialog(
   _AFX_OCC_DIALOG_INFO* pOccDialogInfo
);
```

Parameters

pOccDialogInfo

An **_AFX_OCC_DIALOG_INFO** structure containing information on the dialog template and any ActiveX controls hosted by the dialog.

Remarks

This memory was allocated by a call to **SplitDialogTemplate**, and was used for any hosted ActiveX controls in the dialog box.

Override this function to customize the process of cleaning up any resources used by the dialog box object.

See Also

Reference

COccManager Class
Hierarchy Chart
COccManager::SplitDialogTemplate
COccManager::PreCreateDialog

Other Resources

COccManager Members

COccManager::SetDefaultButton

Call this function to set the control as the default button.

```
static void AFX_CDECL SetDefaultButton(
   CWnd* pWnd,
   BOOL bDefault
);
```

Parameters

pWnd

A pointer to the window containing the control.

bDefault

Nonzero if the control should become the default button; otherwise zero.

Return Value

Nonzero if successful; otherwise zero.

Remarks

Note
The control must have the **OLEMISC_ACTSLIKEBUTTON** status bit set. For more information on **OLEMISC** flags, see the OLEMISC topic in the *Platform SDK*.

See Also

Reference

COccManager Class
Hierarchy Chart
COleControlSite::SetDefaultButton

Other Resources

COccManager Members

COccManager::SplitDialogTemplate

Called by the framework to split the ActiveX controls from common dialog controls.

```
virtual DLGTEMPLATE* SplitDialogTemplate(
   const DLGTEMPLATE* pTemplate,
   DLGITEMTEMPLATE** ppOleDlgItems
);
```

Parameters

pTemplate

A pointer to the dialog template to be examined.

ppOleDlgItems

A list of pointers to dialog box items that are ActiveX controls.

Return Value

A pointer to a dialog template structure containing only non-ActiveX controls. If no ActiveX controls are present, **NULL** is returned.

Remarks

If any ActiveX controls are found, the template is analyzed and a new template, containing only non-ActiveX controls, is created. Any ActiveX controls found during this process are added to *ppOleDlgItems*.

If there are no ActiveX controls in the template, **NULL** is returned.

Note
Memory allocated for the new template is freed in the **PostCreateDialog** function.

Override this function to customize this process.

See Also

Reference

COccManager Class
Hierarchy Chart
COccManager::PostCreateDialog
COccManager::PreCreateDialog

Other Resources

COccManager Members

COleBusyDialog Class

Used for the OLE Server Not Responding or Server Busy dialog boxes.

```
class COleBusyDialog : public COleDialog
```

Remarks

Create an object of class **COleBusyDialog** when you want to call these dialog boxes. After a **COleBusyDialog** object has been constructed, you can use the m_bz structure to initialize the values or states of controls in the dialog box. The **m_bz** structure is of type **OLEUIBUSY**. For more information about using this dialog class, see the DoModal member function.

Note
Application Wizard-generated container code uses this class.

For more information, see the OLEUIBUSY structure in the *Platform SDK*.

For more information on OLE-specific dialog boxes, see the article Dialog Boxes in OLE.

Requirements

Header: afxodlgs.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

COleDialog Class
Hierarchy Chart
COleDialog Class

Other Resources

COleBusyDialog Members

COleBusyDialog Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CDialog Members

CCommonDialog Members

COleDialog Members

Data Members

m_bz	Structure of type **OLEUIBUSY** that controls the behavior of the dialog box.

Construction

COleBusyDialog	Constructs a **COleBusyDialog** object.

Operations

DoModal	Displays the OLE Server Busy dialog box.
GetSelectionType	Determines the choice made in the dialog box.

See Also

Reference

COleBusyDialog Class

Hierarchy Chart

COleBusyDialog Member Functions

For information about the member functions in **COleBusyDialog**, see COleBusyDialog Members.

COleBusyDialog::COleBusyDialog

This function only constructs a **COleBusyDialog** object.

```
explicit COleBusyDialog(
   HTASK htaskBusy,
   BOOL bNotResponding = FALSE,
   DWORD dwFlags = 0,
   CWnd* pParentWnd = NULL
);
```

Parameters

htaskBusy

Handle to the server task that is busy.

bNotResponding

If **TRUE**, call the Not Responding dialog box instead of the Server Busy dialog box. The wording in the Not Responding dialog box is slightly different than the wording in the Server Busy dialog box, and the Cancel button is disabled.

dwFlags

Creation flag. Can contain zero or more of the following values combined with the bitwise-OR operator:

- **BZ_DISABLECANCELBUTTON** Disable the Cancel button when calling the dialog box.
- **BZ_DISABLESWITCHTOBUTTON** Disable the Switch To button when calling the dialog box.
- **BZ_DISABLERETRYBUTTON** Disable the Retry button when calling the dialog box.

pParentWnd

Points to the parent or owner window object (of type **CWnd**) to which the dialog object belongs. If it is **NULL**, the parent window of the dialog object is set to the main application window.

Remarks

To display the dialog box, call DoModal.

For more information, see the OLEUIBUSY structure in the *Platform SDK*.

See Also

Reference

COleBusyDialog Class
Hierarchy Chart
COleBusyDialog::DoModal

Other Resources

COleBusyDialog Members

COleBusyDialog::DoModal

Call this function to display the OLE Server Busy or Server Not Responding dialog box.

```
virtual INT_PTR DoModal( );
```

Return Value

Completion status for the dialog box. One of the following values:

- **IDOK** if the dialog box was successfully displayed.
- **IDCANCEL** if the user canceled the dialog box.
- **IDABORT** if an error occurred. If **IDABORT** is returned, call the **COleDialog::GetLastError** member function to get more information about the type of error that occurred. For a listing of possible errors, see the OleUIBusy function in the *Platform SDK*.

Remarks

If you want to initialize the various dialog box controls by setting members of the m_bz structure, you should do this before calling **DoModal**, but after the dialog object is constructed.

If **DoModal** returns **IDOK**, you can call other member functions to retrieve the settings or information that was input by the user into the dialog box.

See Also

Reference

COleBusyDialog Class
Hierarchy Chart
COleDialog::GetLastError
CDialog::DoModal
COleBusyDialog::GetSelectionType
COleBusyDialog::m_bz

Other Resources

COleBusyDialog Members

COleBusyDialog::GetSelectionType

Call this function to get the selection type chosen by the user in the Server Busy dialog box.

```
UINT GetSelectionType( ) const;
```

Return Value

Type of selection made.

Remarks

The return type values are specified by the **Selection** enumeration type declared in the **COleBusyDialog** class.

```
enum Selection
{
   switchTo,
   retry,
   callUnblocked
};
```

Brief descriptions of these values follow:

- **COleBusyDialog::switchTo** Switch To button was pressed.
- **COleBusyDialog::retry** Retry button was pressed.
- **COleBusyDialog::callUnblocked** Call to activate the server is now unblocked.

See Also

Reference

COleBusyDialog Class
Hierarchy Chart
COleBusyDialog::DoModal

Other Resources

COleBusyDialog Members

COleBusyDialog Data Members

For information about the data members in **COleBusyDialog**, see COleBusyDialog Members.

COleBusyDialog::m_bz

Structure of type **OLEUIBUSY** used to control the behavior of the Server Busy dialog box.

```
OLEUIBUSY m_bz;
```

Remarks

Members of this structure can be modified directly or through member functions.

For more information, see the OLEUIBUSY structure in the *Platform SDK*.

See Also

Reference

COleBusyDialog Class
Hierarchy Chart
COleBusyDialog::COleBusyDialog
COleBusyDialog::DoModal

Other Resources

COleBusyDialog Members

COleChangeIconDialog Class

Used for the OLE Change Icon dialog box.

```
class COleChangeIconDialog : public COleDialog
```

Remarks

Create an object of class **COleChangeIconDialog** when you want to call this dialog box. After a **COleChangeIconDialog** object has been constructed, you can use the m_ci structure to initialize the values or states of controls in the dialog box. The **m_ci** structure is of type **OLEUICHANGEICON**. For more information about using this dialog class, see the DoModal member function.

For more information, see the OLEUICHANGEICON structure in the *Platform SDK*.

For more information about OLE-specific dialog boxes, see the article Dialog Boxes in OLE.

Requirements

Header: afxodlgs.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

COleDialog Class
Hierarchy Chart
COleDialog Class

Other Resources

COleChangeIconDialog Members

COleChangeIconDialog Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CDialog Members

CCommonDialog Members

COleDialog Members

Data Members

m_ci	A structure that controls the behavior of the dialog box.

Construction

COleChangeIconDialog	Constructs a **COleChangeIconDialog** object.

Operations and Attributes

DoChangeIcon	Performs the change specified in the dialog box.
DoModal	Displays the OLE 2 Change Icon dialog box.
GetIconicMetafile	Gets a handle to the metafile associated with the iconic form of this item.

See Also

Reference

COleChangeIconDialog Class

Hierarchy Chart

COleChangeIconDialog Member Functions

For information about the member functions in **COleChangeIconDialog**, see COleChangeIconDialog Members.

COleChangeIconDialog::COleChangeIconDialog

This function constructs only a **COleChangeIconDialog** object.

```
explicit COleChangeIconDialog(
   COleClientItem* pItem,
   DWORD dwFlags = CIF_SELECTCURRENT,
   CWnd* pParentWnd = NULL
);
```

Parameters

pItem

Points to the item to be converted.

dwFlags

Creation flag, which contains any number of the following values combined using the bitwise-or operator:

- **CIF_SELECTCURRENT** Specifies that the Current radio button will be selected initially when the dialog box is called. This is the default.
- **CIF_SELECTDEFAULT** Specifies that the Default radio button will be selected initially when the dialog box is called.
- **CIF_SELECTFROMFILE** Specifies that the From File radio button will be selected initially when the dialog box is called.
- **CIF_SHOWHELP** Specifies that the Help button will be displayed when the dialog box is called.
- **CIF_USEICONEXE** Specifies that the icon should be extracted from the executable specified in the **szIconExe** field of m_ci instead of retrieved from the type. This is useful for embedding or linking to non-OLE files.

pParentWnd

Points to the parent or owner window object (of type **CWnd**) to which the dialog object belongs. If it is **NULL**, the parent window of the dialog box will be set to the main application window.

Remarks

To display the dialog box, call the DoModal function.

For more information, see the OLEUICHANGEICON structure in the *Platform SDK*.

See Also

Reference

COleChangeIconDialog Class
Hierarchy Chart
COleClientItem Class
COleChangeIconDialog::DoModal

Other Resources

COleChangeIconDialog Members

COleChangeIconDialog::DoChangeIcon

Call this function to change the icon representing the item to the one selected in the dialog box after DoModal returns **IDOK**.

```
BOOL DoChangeIcon(
   COleClientItem* pItem
);
```

Parameters

pItem

Points to the item whose icon is changing.

Return Value

Nonzero if change is successful; otherwise 0.

See Also

Reference

COleChangeIconDialog Class

Hierarchy Chart

COleChangeIconDialog::DoModal

Other Resources

COleChangeIconDialog Members

COleChangeIconDialog::DoModal

Call this function to display the OLE Change Icon dialog box.

```
virtual INT_PTR DoModal( );
```

Return Value

Completion status for the dialog box. One of the following values:

- **IDOK** if the dialog box was successfully displayed.
- **IDCANCEL** if the user canceled the dialog box.
- **IDABORT** if an error occurred. If **IDABORT** is returned, call the **COleDialog::GetLastError** member function to get more information about the type of error that occurred. For a listing of possible errors, see the OleUIChangeIcon function in the *Platform SDK*.

Remarks

If you want to initialize the various dialog box controls by setting members of the m_ci structure, you should do this before calling **DoModal**, but after the dialog object is constructed.

If **DoModal** returns **IDOK**, you can call other member functions to retrieve the settings or information that was input by the user into the dialog box.

See Also

Reference

COleChangeIconDialog Class
Hierarchy Chart
COleDialog::GetLastError
CDialog::DoModal
COleChangeIconDialog::m_ci
COleChangeIconDialog::DoChangeIcon
COleChangeIconDialog::GetIconicMetafile

Other Resources

COleChangeIconDialog Members

COleChangeIconDialog::GetIconicMetafile

Call this function to get a handle to the metafile that contains the iconic aspect of the selected item.

```
HGLOBAL GetIconicMetafile( ) const;
```

Return Value

The handle to the metafile containing the iconic aspect of the new icon, if the dialog box was dismissed by choosing **OK**; otherwise, the icon as it was before the dialog was displayed.

See Also

Reference

COleChangeIconDialog Class
Hierarchy Chart
COleChangeIconDialog::DoModal
COleChangeIconDialog::COleChangeIconDialog
COleChangeIconDialog::DoChangeIcon

Other Resources

COleChangeIconDialog Members

COleChangeIconDialog Data Members

For information about the data members in **COleChangeIconDialog**, see COleChangeIconDialog Members.

COleChangeIconDialog::m_ci

Structure of type **OLEUICHANGEICON** used to control the behavior of the Change Icon dialog box.

```
OLEUICHANGEICON m_ci;
```

Remarks

Members of this structure can be modified either directly or through member functions.

For more information, see the OLEUICHANGEICON structure in the *Platform SDK*.

See Also

Reference

COleChangeIconDialog Class
Hierarchy Chart
COleChangeIconDialog::COleChangeIconDialog

Other Resources

COleChangeIconDialog Members

COleChangeSourceDialog Class

Used for the OLE Change Source dialog box.

```
class COleChangeSourceDialog : public COleDialog
```

Remarks

Create an object of class **COleChangeSourceDialog** when you want to call this dialog box. After a **COleChangeSourceDialog** object has been constructed, you can use the m_cs structure to initialize the values or states of controls in the dialog box. The **m_cs** structure is of type OLEUICHANGESOURCE. For more information about using this dialog class, see the DoModal member function.

For more information, see the OLEUICHANGESOURCE structure in *Platform SDK*.

For more information about OLE-specific dialog boxes, see the article Dialog Boxes in OLE.

Requirements

Header: afxodlgs.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

COleDialog Class
Hierarchy Chart
COleDialog Class

Other Resources

COleChangeSourceDialog Members

COleChangeSourceDialog Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CDialog Members

CCommonDialog Members

COleDialog Members

Constructor

COleChangeSourceDialog	Constructs a **COleChangeSourceDialog** object.

Operations

DoModal	Displays the OLE Change Source dialog box.

Attributes

GetDisplayName	Gets the complete source display name.
GetFileName	Gets the filename from the source name.
GetFromPrefix	Gets the prefix of the previous source.
GetItemName	Gets the item name from the source name.
GetToPrefix	Gets the prefix of the new source
IsValidSource	Indicates if the source is valid.

Data Member

m_cs	A structure that controls the behavior of the dialog box.

See Also

Reference

COleChangeSourceDialog Class

Hierarchy Chart

COleChangeSourceDialog Member Functions

For information about the member functions in **COleChangeSourceDialog**, see COleChangeSourceDialog Members.

COleChangeSourceDialog::COleChangeSourceDialog

This function constructs a **COleChangeSourceDialog** object.

```
explicit COleChangeSourceDialog(
   COleClientItem* pItem,
   CWnd* pParentWnd = NULL
);
```

Parameters

pItem

Pointer to the linked COleClientItem whose source is to be updated.

pParentWnd

Points to the parent or owner window object (of type **CWnd**) to which the dialog object belongs. If it is **NULL**, the parent window of the dialog box will be set to the main application window.

Remarks

To display the dialog box, call the DoModal function.

For more information, see the OLEUICHANGESOURCE structure and OleUIChangeSource function in *Platform SDK*.

See Also

Reference

COleChangeSourceDialog Class

Hierarchy Chart

Other Resources

COleChangeSourceDialog Members

COleChangeSourceDialog::DoModal

Call this function to display the OLE Change Source dialog box.

```
virtual INT_PTR DoModal( );
```

Return Value

Completion status for the dialog box. One of the following values:

- **IDOK** if the dialog box was successfully displayed.
- **IDCANCEL** if the user canceled the dialog box.
- **IDABORT** if an error occurred. If **IDABORT** is returned, call the COleDialog::GetLastError member function to get more information about the type of error that occurred. For a listing of possible errors, see the OleUIChangeSource function in *Platform SDK*.

Remarks

If you want to initialize the various dialog box controls by setting members of the m_cs structure, you should do this before calling **DoModal**, but after the dialog object is constructed.

If **DoModal** returns **IDOK**, you can call member functions to retrieve user-entered settings or information from the dialog box. The following list names typical query functions:

- GetFileName
- GetDisplayName
- GetItemName

See Also

Reference

COleChangeSourceDialog Class
Hierarchy Chart
COleChangeSourceDialog::COleChangeSourceDialog

Other Resources

COleChangeSourceDialog Members

COleChangeSourceDialog::GetDisplayName

Call this function to retrieve the complete display name for the linked client item.

```
CString GetDisplayName( );
```

Return Value

The complete source display name (moniker) for the COleClientItem specified in the constructor.

See Also

Reference

COleChangeSourceDialog Class
Hierarchy Chart
COleChangeSourceDialog::GetFileName
COleChangeSourceDialog::GetItemName

Other Resources

COleChangeSourceDialog Members

COleChangeSourceDialog::GetFileName

Call this function to retrieve the file moniker portion of the display name for the linked client item.

```
CString GetFileName( );
```

Return Value

The file moniker portion of the source display name for the COleClientItem specified in the constructor.

Remarks

The file moniker together with the item moniker gives the complete display name.

See Also

Reference

COleChangeSourceDialog Class
Hierarchy Chart
COleChangeSourceDialog::GetDisplayName
COleChangeSourceDialog::GetItemName

Other Resources

COleChangeSourceDialog Members

COleChangeSourceDialog::GetFromPrefix

Call this function to get the previous prefix string for the source.

```
CString GetFromPrefix( );
```

Return Value

The previous prefix string of the source.

Remarks

Call this function only after DoModal returns **IDOK**.

This value comes directly from the **lpszFrom** member of the OLEUICHANGESOURCE structure.

For more information, see the OLEUICHANGESOURCE structure in *Platform SDK*.

See Also

Reference

COleChangeSourceDialog Class
Hierarchy Chart
COleChangeSourceDialog::GetToPrefix

Other Resources

COleChangeSourceDialog Members

COleChangeSourceDialog::GetItemName

Call this function to retrieve the item moniker portion of the display name for the linked client item.

```
CString GetItemName( );
```

Return Value

The item moniker portion of the source display name for the COleClientItem specified in the constructor.

Remarks

The file moniker together with the item moniker gives the complete display name.

See Also

Reference

COleChangeSourceDialog Class
Hierarchy Chart
COleChangeSourceDialog::GetFileName
COleChangeSourceDialog::GetDisplayName

Other Resources

COleChangeSourceDialog Members

COleChangeSourceDialog::GetToPrefix

Call this function to get the new prefix string for the source.

```
CString GetToPrefix( );
```

Return Value

The new prefix string of the source.

Remarks

Call this function only after DoModal returns **IDOK**.

This value comes directly from the **lpszTo** member of the OLEUICHANGESOURCE structure.

For more information, see the OLEUICHANGESOURCE structure in *Platform SDK*.

See Also

Reference

COleChangeSourceDialog Class
Hierarchy Chart
COleChangeSourceDialog::GetFromPrefix

Other Resources

COleChangeSourceDialog Members

COleChangeSourceDialog::IsValidSource

Call this function to determine if the new source is valid.

```
BOOL IsValidSource( );
```

Return Value

Nonzero if the new source is valid, otherwise 0.

Remarks

Call this function only after DoModal returns **IDOK**.

For more information, see the OLEUICHANGESOURCE structure in *Platform SDK*.

See Also

Reference

COleChangeSourceDialog Class
Hierarchy Chart
COleChangeSourceDialog::DoModal

Other Resources

COleChangeSourceDialog Members

COleChangeSourceDialog Data Members

For information about the data members in **COleChangeSourceDialog**, see COleChangeSourceDialog Members.

COleChangeSourceDialog::m_cs

This data member is a structure of type OLEUICHANGESOURCE.

```
OLEUICHANGESOURCE m_cs;
```

Remarks

OLEUICHANGESOURCE is used to control the behavior of the OLE Change Source dialog box. Members of this structure can be modified directly.

For more information, see the OLEUICHANGESOURCE structure in *Platform SDK*.

See Also

Reference

COleChangeSourceDialog Class
Hierarchy Chart
COleChangeSourceDialog::COleChangeSourceDialog

Other Resources

COleChangeSourceDialog Members

COleClientItem Class

Defines the container interface to OLE items.

```
class COleClientItem : public CDocItem
```

Remarks

An OLE item represents data, created and maintained by a server application, which can be "seamlessly" incorporated into a document so that it appears to the user to be a single document. The result is a "compound document" made up of the OLE item and a containing document.

An OLE item can be either embedded or linked. If it is embedded, its data is stored as part of the compound document. If it is linked, its data is stored as part of a separate file created by the server application, and only a link to that file is stored in the compound document. All OLE items contain information specifying the server application that should be called to edit them.

COleClientItem defines several overridable functions that are called in response to requests from the server application; these overridables usually act as notifications. This allows the server application to inform the container of changes the user makes when editing the OLE item, or to retrieve information needed during editing.

COleClientItem can be used with either the COleDocument, COleLinkingDoc, or COleServerDoc class. To use **COleClientItem**, derive a class from it and implement the OnChange member function, which defines how the container responds to changes made to the item. To support in-place activation, override the OnGetItemPosition member function. This function provides information about the displayed position of the OLE item.

For more information about using the container interface, see the articles Containers: Implementing a Container and Activation.

Note
The *Platform SDK* refers to embedded and linked items as "objects" and refers to types of items as "classes." This reference uses the term "item" to distinguish the OLE entity from the corresponding C++ object and the term "type" to distinguish the OLE category from the C++ class.

Requirements

Header: afxole.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

MFCBIND Sample: Active Document Container

OCLIENT Sample: Illustrates a Visual Editing Container Application

Reference

CDocItem Class

Hierarchy Chart

COleServerItem Class

Other Resources

COleClientItem Members

COleClientItem Members

Base Class Members

Creation

Status

Data Access

Object Conversion

Clipboard Operations

General Operations

Activation

Embedded Object Operations

Linked Object Operations and Status

Overridables

Base Class Members

CObject Members

CCmdTarget Members

CDocItem Members

Construction

COleClientItem	Constructs a **COleClientItem** object.

Creation

CanCreateFromData	Indicates whether a container application can create an embedded object.
CanCreateLinkFromData	Indicates whether a container application can create a linked object.
CreateCloneFrom	Creates a duplicate of an existing item.
CreateFromClipboard	Creates an embedded item from the Clipboard.
CreateFromData	Creates an embedded item from a data object.
CreateFromFile	Creates an embedded item from a file.
CreateLinkFromClipboard	Creates a linked item from the Clipboard.
CreateLinkFromData	Creates a linked item from a data object.
CreateLinkFromFile	Creates a linked item from a file.
CreateNewItem	Creates a new embedded item by launching the server application.
CreateStaticFromClipboard	Creates a static item from the Clipboard.
CreateStaticFromData	Creates a static item from a data object.

Status

GetActiveView	Gets the view on which the item is activated in place.
GetCachedExtent	Returns the bounds of the OLE item's rectangle.
GetClassID	Gets the present item's class ID.
GetDrawAspect	Gets the item's current view for rendering.
GetExtent	Returns the bounds of the OLE item's rectangle.
GetIconFromRegistry	Retrives a handle to an icon associated with the server of a particular CLSID.
GetIconicMetafile	Gets the metafile used for drawing the item's icon.
GetItemState	Gets the item's current state.
GetLastStatus	Returns the status of the last OLE operation.
GetType	Returns the type (embedded, linked, or static) of the OLE item.
GetUserType	Gets a string describing the item's type.
IsInPlaceActive	Returns **TRUE** if the item is in-place active.
IsModified	Returns **TRUE** if the item has been modified since it was last saved.
IsOpen	Returns **TRUE** if the item is currently open in the server application.
IsRunning	Returns **TRUE** if the item's server application is running.
SetDrawAspect	Sets the item's current view for rendering.
SetIconicMetafile	Caches the metafile used for drawing the item's icon.

Data Access

AttachDataObject	Accesses the data in the OLE object.
GetDocument	Returns the **COleDocument** object that contains the present item.

Object Conversion

ActivateAs	Activates the item as another type.
ConvertTo	Converts the item to another type.
Reload	Reloads the item after a call to **ActivateAs**.

Clipboard Operations

CanPaste	Indicates whether the Clipboard contains an embeddable or static OLE item.
CanPasteLink	Indicates whether the Clipboard contains a linkable OLE item.
CopyToClipboard	Copies the OLE item to the Clipboard.
DoDragDrop	Performs a drag-and-drop operation.

GetClipboardData	Gets the data that would be placed on the Clipboard by calling the **CopyToClipboard** member function.

General Operations

Close	Closes a link to a server but does not destroy the OLE item.
Delete	Deletes or closes the OLE item if it was a linked item.
Draw	Draws the OLE item.
Release	Releases the connection to an OLE linked item and closes it if it was open. Does not destroy the client item.
Run	Runs the application associated with the item.
SetPrintDevice	Sets the print-target device for this client item.

Activation

Activate	Opens the OLE item for an operation and then executes the specified verb.
Deactivate	Deactivates the item.
DeactivateUI	Restores the container application's user interface to its original state.
DoVerb	Executes the specified verb.
GetInPlaceWindow	Returns a pointer to the item's in-place editing window.
ReactivateAndUndo	Reactivates the item and undoes the last in-place editing operation.
SetItemRects	Sets the item's bounding rectangle.

Embedded Object Operations

SetExtent	Sets the bounding rectangle of the OLE item.
SetHostNames	Sets the names the server displays when editing the OLE item.

Linked Object Operations and Status

GetLinkUpdateOptions	Returns the update mode for a linked item (advanced feature).
IsLinkUpToDate	Returns **TRUE** if a linked item is up to date with its source document.
SetLinkUpdateOptions	Sets the update mode for a linked item (advanced feature).
UpdateLink	Updates the presentation cache of an item.

Overridables

CanActivate	Called by the framework to determine whether in-place activation is allowed.
OnActivate	Called by the framework to notify the item that it is activated.
OnActivateUI	Called by the framework to notify the item that it is activated and should show its user interface.
OnChange	Called when the server changes the OLE item. Implementation required.

OnChangeItemPosition	Called by the framework when an item's position changes.
OnDeactivate	Called by the framework when an item is deactivated.
OnDeactivateAndUndo	Called by the framework to undo after activation.
OnDeactivateUI	Called by the framework when the server has removed its in-place user interface.
OnDiscardUndoState	Called by the framework to discard the item's undo state information.
OnGetClipboardData	Called by the framework to get the data to be copied to the Clipboard.
OnGetClipRect	Called by the framework to get the item's clipping-rectangle coordinates.
OnGetItemPosition	Called by the framework to get the item's position relative to the view.
OnGetWindowContext	Called by the framework when an item is activated in place.
OnInsertMenus	Called by the framework to create a composite menu.
OnRemoveMenus	Called by the framework to remove the container's menus from a composite menu.
OnScrollBy	Called by the framework to scroll the item into view.
OnSetMenu	Called by the framework to install and remove a composite menu.
OnShowControlBars	Called by the framework to show and hide control bars.
OnShowItem	Called by the framework to display the OLE item.
OnUpdateFrameTitle	Called by the framework to update the frame window's title bar.

See Also

Reference

COleClientItem Class
Hierarchy Chart

COleClientItem Member Functions

For information about the member functions in **COleClientItem**, see COleClientItem Members.

COleClientItem::Activate

Call this function to execute the specified verb instead of DoVerb so that you can do your own processing when an exception is thrown.

```
void Activate(
   LONG nVerb,
   CView* pView,
   LPMSG lpMsg = NULL
);
```

Parameters

nVerb

Specifies the verb to execute. It can be one of the following:

Value	Meaning	Symbol
– 0	Primary verb	**OLEIVERB_PRIMARY**
– 1	Secondary verb	(None)
– 1	Display item for editing	**OLEIVERB_SHOW**
– 2	Edit item in separate window	**OLEIVERB_OPEN**
– 3	Hide item	**OLEIVERB_HIDE**

The –1 value is typically an alias for another verb. If open editing is not supported, –2 has the same effect as –1. For additional values, see IOleObject::DoVerb in the *Platform SDK*.

pView

Pointer to the container view window that contains the OLE item; this is used by the server application for in-place activation. This parameter should be **NULL** if the container does not support in-place activation.

lpMsg

Pointer to the message that caused the item to be activated.

Remarks

If the server application was written using the Microsoft Foundation Class Library, this function causes the OnDoVerb member function of the corresponding **COleServerItem** object to be executed.

If the primary verb is Edit and zero is specified in the *nVerb* parameter, the server application is launched to allow the OLE item to be edited. If the container application supports in-place activation, editing can be done in place. If the container does not support in-place activation (or if the Open verb is specified), the server is launched in a separate window and editing can be done there. Typically, when the user of the container application double-clicks the OLE item, the value for the primary verb in the *nVerb* parameter determines which action the user can take. However, if the server supports only one action, it takes that action, no matter which value is specified in the *nVerb* parameter.

For more information, see IOleObject::DoVerb in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::DoVerb
COleServerItem::OnDoVerb

Other Resources

COleClientItem Members

COleClientItem::ActivateAs

Uses OLE's object conversion facilities to activate the item as though it were an item of the type specified by *clsidNew*.

```
virtual BOOL ActivateAs(
   LPCTSTR lpszUserType,
   REFCLSID clsidOld,
   REFCLSID clsidNew
);
```

Parameters

lpszUserType

Pointer to a string representing the target user type, such as "Word Document."

clsidOld

A reference to the item's current class ID. The class ID should represent the type of the actual object, as stored, unless it is a link. In that case, it should be the CLSID of the item to which the link refers. The COleConvertDialog automatically provides the correct class ID for the item.

clsidNew

A reference to the target class ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This is called automatically by COleConvertDialog::DoConvert. It is not usually called directly.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleConvertDialog Class
COleClientItem::ConvertTo
COleClientItem::Reload

Other Resources

COleClientItem Members

COleClientItem::AttachDataObject

Call this function to initialize a COleDataObject for accessing the data in the OLE item.

```
void AttachDataObject(
   COleDataObject& rDataObject
) const;
```

Parameters

rDataObject

Reference to a **COleDataObject** object that will be initialized to allow access to the data in the OLE item.

See Also

Reference

COleClientItem Class

Hierarchy Chart

COleDataObject Class

Other Resources

COleClientItem Members

COleClientItem::CanActivate

Called by the framework when the user requests in-place activation of the OLE item; this function's return value determines whether in-place activation is allowed.

```
virtual BOOL CanActivate( );
```

Return Value

Nonzero if in-place activation is allowed; otherwise 0.

Remarks

The default implementation allows in-place activation if the container has a valid window. Override this function to implement special logic for accepting or refusing the activation request. For example, an activation request can be refused if the OLE item is too small or not currently visible.

For more information, see IOleInPlaceSite::CanInPlaceActivate in the *Platform SDK*.

See Also

Reference

COleClientItem Class

Hierarchy Chart

Other Resources

COleClientItem Members

COleClientItem::CanCreateFromData

Checks whether a container application can create an embedded object from the given **COleDataObject** object.

```
static BOOL PASCAL CanCreateFromData(
   const COleDataObject* pDataObject
);
```

Parameters

pDataObject

Pointer to the COleDataObject object from which the OLE item is to be created.

Return Value

Nonzero if the container can create an embedded object from the **COleDataObject** object; otherwise 0.

Remarks

The **COleDataObject** class is used in data transfers for retrieving data in various formats from the Clipboard, through drag and drop, or from an embedded OLE item.

Containers can use this function to decide to enable or disable their Edit Paste and Edit Paste Special commands.

For more information, see the article Data Objects and Data Sources (OLE).

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleDataObject Class

Other Resources

COleClientItem Members

COleClientItem::CanCreateLinkFromData

Checks whether a container application can create a linked object from the given **COleDataObject** object.

```
static BOOL PASCAL CanCreateLinkFromData(
   const COleDataObject* pDataObject
);
```

Parameters

pDataObject

Pointer to the COleDataObject object from which the OLE item is to be created.

Return Value

Nonzero if the container can create a linked object from the **COleDataObject** object.

Remarks

The **COleDataObject** class is used in data transfers for retrieving data in various formats from the Clipboard, through drag and drop, or from an embedded OLE item.

Containers can use this function to decide to enable or disable their Edit Paste Special and Edit Paste Link commands.

For more information, see the article Data Objects and Data Sources (OLE).

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleDataObject Class

Other Resources

COleClientItem Members

COleClientItem::CanPaste

Call this function to see whether an embedded OLE item can be pasted from the Clipboard.

```
static BOOL PASCAL CanPaste( );
```

Return Value

Nonzero if an embedded OLE item can be pasted from the Clipboard; otherwise 0.

Remarks

For more information, see OleGetClipboard and OleQueryCreateFromData in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::CanPasteLink
COleClientItem::CreateFromClipboard
COleClientItem::CreateStaticFromClipboard
COleDocument Class

Other Resources

COleClientItem Members

COleClientItem::CanPasteLink

Call this function to see whether a linked OLE item can be pasted from the Clipboard.

```
static BOOL PASCAL CanPasteLink( );
```

Return Value

Nonzero if a linked OLE item can be pasted from the Clipboard; otherwise 0.

Remarks

For more information, see OleGetClipboard and OleQueryLinkFromData in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::CanPaste
COleClientItem::CreateLinkFromClipboard

Other Resources

COleClientItem Members

COleClientItem::Close

Call this function to change the state of an OLE item from the running state to the loaded state, that is, loaded with its handler in memory but with the server not running.

```
void Close(
   OLECLOSE dwCloseOption = OLECLOSE_SAVEIFDIRTY
);
```

Parameters

dwCloseOption

Flag specifying under what circumstances the OLE item is saved when it returns to the loaded state. It can have one of the following values:

- **OLECLOSE_SAVEIFDIRTY** Save the OLE item.
- **OLECLOSE_NOSAVE** Do not save the OLE item.
- **OLECLOSE_PROMPTSAVE** Prompt the user on whether to save the OLE item.

Remarks

This function has no effect when the OLE item is not running.

For more information, see IOleObject::Close in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::UpdateLink

Other Resources

COleClientItem Members

COleClientItem::COleClientItem

Constructs a **COleClientItem** object and adds it to the container document's collection of document items, which constructs only the C++ object and does not perform any OLE initialization.

```
COleClientItem(
   COleDocument* pContainerDoc = NULL
);
```

Parameters

pContainerDoc

Pointer to the container document that will contain this item. This can be any COleDocument derivative.

Remarks

If you pass a **NULL** pointer, no addition is made to the container document. You must explicitly call COleDocument::AddItem.

You must call one of the following creation member functions before you use the OLE item:

- CreateFromClipboard
- CreateFromData
- CreateFromFile
- CreateStaticFromClipboard
- CreateStaticFromData
- CreateLinkFromClipboard
- CreateLinkFromData
- CreateLinkFromFile
- CreateNewItem
- CreateCloneFrom

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleDocument Class
COleDocument::AddItem

Other Resources

COleClientItem Members

COleClientItem::ConvertTo

Call this member function to convert the item to the type specified by *clsidNew*.

```
virtual BOOL ConvertTo(
   REFCLSID clsidNew
);
```

Parameters

clsidNew

The class ID of the target type.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This is called automatically by COleConvertDialog. It is not necessary to call it directly.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::ActivateAs
COleConvertDialog Class

Other Resources

COleClientItem Members

COleClientItem::CopyToClipboard

Call this function to copy the OLE item to the Clipboard.

```
void CopyToClipboard(
   BOOL bIncludeLink = FALSE
);
```

Parameters

bIncludeLink

TRUE if link information should be copied to the Clipboard, allowing a linked item to be pasted; otherwise **FALSE**.

Remarks

Typically, you call this function when writing message handlers for the Copy or Cut commands from the Edit menu. You must implement item selection in your container application if you want to implement the Copy or Cut commands.

For more information, see OleSetClipboard in the *Platform SDK*.

See Also

Reference

COleClientItem Class

Hierarchy Chart

Other Resources

COleClientItem Members

COleClientItem::CreateCloneFrom

Call this function to create a copy of the specified OLE item.

```
BOOL CreateCloneFrom(
   const COleClientItem* pSrcItem
);
```

Parameters

pSrcItem

Pointer to the OLE item to be duplicated.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The copy is identical to the source item. You can use this function to support undo operations.

See Also

Reference

COleClientItem Class

Hierarchy Chart

COleClientItem::CreateNewItem

Other Resources

COleClientItem Members

COleClientItem::CreateFromClipboard

Call this function to create an embedded item from the contents of the Clipboard.

```
BOOL CreateFromClipboard(
   OLERENDER render = OLERENDER_DRAW,
   CLIPFORMAT cfFormat = 0,
   LPFORMATETC lpFormatEtc = NULL
);
```

Parameters

render

Flag specifying how the server will render the OLE item. For the possible values, see OLERENDER in the *Platform SDK*.

cfFormat

Specifies the Clipboard data format to be cached when creating the OLE item.

lpFormatEtc

Pointer to a FORMATETC structure used if *render* is **OLERENDER_FORMAT** or **OLERENDER_DRAW**. Provide a value for this parameter only if you want to specify additional format information beyond the Clipboard format specified by *cfFormat*. If you omit this parameter, default values are used for the other fields in the **FORMATETC** structure.

Return Value

Nonzero if successful; otherwise 0.

Remarks

You typically call this function from the message handler for the Paste command on the Edit menu. (The Paste command is enabled by the framework if the CanPaste member function returns nonzero.)

For more information, see OLERENDER and FORMATETC in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleDataObject::AttachClipboard
COleClientItem::CreateFromData
COleClientItem::CanPaste

Other Resources

COleClientItem Members

COleClientItem::CreateFromData

Call this function to create an embedded item from a **COleDataObject** object.

```
BOOL CreateFromData(
   COleDataObject* pDataObject,
   OLERENDER render = OLERENDER_DRAW,
   CLIPFORMAT cfFormat = 0,
   LPFORMATETC lpFormatEtc = NULL
);
```

Parameters

pDataObject

Pointer to the COleDataObject object from which the OLE item is to be created.

render

Flag specifying how the server will render the OLE item. For the possible values, see OLERENDER in the *Platform SDK*.

cfFormat

Specifies the Clipboard data format to be cached when creating the OLE item.

lpFormatEtc

Pointer to a FORMATETC structure used if *render* is **OLERENDER_FORMAT** or **OLERENDER_DRAW**. Provide a value for this parameter only if you want to specify additional format information beyond the Clipboard format specified by *cfFormat*. If you omit this parameter, default values are used for the other fields in the **FORMATETC** structure.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Data transfer operations, such as pasting from the Clipboard or drag-and-drop operations, provide **COleDataObject** objects containing the information offered by a server application. It is usually used in your override of CView::OnDrop.

For more information, see OleCreateFromData, OLERENDER, and FORMATETC in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleDataObject::AttachClipboard
COleClientItem::CreateFromClipboard
COleDataObject Class

Other Resources

COleClientItem Members

COleClientItem::CreateFromFile

Call this function to create an embedded OLE item from a file.

```
BOOL CreateFromFile(
   LPCTSTR lpszFileName,
   REFCLSID clsid = CLSID_NULL,
   OLERENDER render = OLERENDER_DRAW,
   CLIPFORMAT cfFormat = 0,
   LPFORMATETC lpFormatEtc = NULL
);
```

Parameters

lpszFileName

Pointer to the name of the file from which the OLE item is to be created.

clsid

Reserved for future use.

render

Flag specifying how the server will render the OLE item. For the possible values, see OLERENDER in the *Platform SDK*.

cfFormat

Specifies the Clipboard data format to be cached when creating the OLE item.

lpFormatEtc

Pointer to a FORMATETC structure used if *render* is **OLERENDER_FORMAT** or **OLERENDER_DRAW**. Provide a value for this parameter only if you want to specify additional format information beyond the Clipboard format specified by *cfFormat*. If you omit this parameter, default values are used for the other fields in the **FORMATETC** structure.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The framework calls this function from COleInsertDialog::CreateItem if the user chooses OK from the Insert Object dialog box when the Create from File button is selected.

For more information, see OleCreateFromFile, OLERENDER, and FORMATETC in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleInsertDialog::CreateItem

Other Resources

COleClientItem Members

COleClientItem::CreateLinkFromClipboard

Call this function to create a linked item from the contents of the Clipboard.

```
BOOL CreateLinkFromClipboard(
   OLERENDER render = OLERENDER_DRAW,
   CLIPFORMAT cfFormat = 0,
   LPFORMATETC lpFormatEtc = NULL
);
```

Parameters

render

Flag specifying how the server will render the OLE item. For the possible values, see OLERENDER in the *Platform SDK*.

cfFormat

Specifies the Clipboard data format to be cached when creating the OLE item.

lpFormatEtc

Pointer to a FORMATETC structure used if *render* is **OLERENDER_FORMAT** or **OLERENDER_DRAW**. Provide a value for this parameter only if you want to specify additional format information beyond the Clipboard format specified by *cfFormat*. If you omit this parameter, default values are used for the other fields in the **FORMATETC** structure.

Return Value

Nonzero if successful; otherwise 0.

Remarks

You typically call this function from the message handler for the Paste Link command on the Edit menu. (The Paste Link command is enabled in the default implementation of COleDocument if the Clipboard contains an OLE item that can be linked to.)

For more information, see OLERENDER and FORMATETC in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::CanPasteLink
COleClientItem::CreateLinkFromData
COleDataObject::AttachClipboard

Other Resources

COleClientItem Members

COleClientItem::CreateLinkFromData

Call this function to create a linked item from a **COleDataObject** object.

```
BOOL CreateLinkFromData(
   COleDataObject* pDataObject,
   OLERENDER render = OLERENDER_DRAW,
   CLIPFORMAT cfFormat = 0,
   LPFORMATETC lpFormatEtc = NULL
);
```

Parameters

pDataObject

Pointer to the COleDataObject object from which the OLE item is to be created.

render

Flag specifying how the server will render the OLE item. For the possible values, see OLERENDER in the *Platform SDK*.

cfFormat

Specifies the Clipboard data format to be cached when creating the OLE item.

lpFormatEtc

Pointer to a FORMATETC structure used if *render* is **OLERENDER_FORMAT** or **OLERENDER_DRAW**. Provide a value for this parameter only if you want to specify additional format information beyond the Clipboard format specified by *cfFormat*. If you omit this parameter, default values are used for the other fields in the **FORMATETC** structure.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Call this during a drop operation when the user indicates a link should be created. It can also be used to handle the Edit Paste command. It is called by the framework in **COleClientItem::CreateLinkFromClipboard** and in COlePasteSpecialDialog::CreateItem when the Link option has been selected.

For more information, see OleCreateLinkFromData, OLERENDER, and FORMATETC in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleDataObject::AttachClipboard
COleDataObject Class
COleClientItem::CreateLinkFromClipboard

Other Resources

COleClientItem Members

COleClientItem::CreateLinkFromFile

Call this function to create a linked OLE item from a file.

```
BOOL CreateLinkFromFile(
   LPCTSTR lpszFileName,
   OLERENDER render = OLERENDER_DRAW,
   CLIPFORMAT cfFormat = 0,
   LPFORMATETC lpFormatEtc = NULL
);
```

Parameters

lpszFileName

Pointer to the name of the file from which the OLE item is to be created.

render

Flag specifying how the server will render the OLE item. For the possible values, see OLERENDER in the *Platform SDK*.

cfFormat

Specifies the Clipboard data format to be cached when creating the OLE item.

lpFormatEtc

Pointer to a FORMATETC structure used if *render* is **OLERENDER_FORMAT** or **OLERENDER_DRAW**. Provide a value for this parameter only if you want to specify additional format information beyond the Clipboard format specified by *cfFormat*. If you omit this parameter, default values are used for the other fields in the **FORMATETC** structure.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The framework calls this function if the user chooses OK from the Insert Object dialog box when the Create from File button is selected and the Link check box is checked. It is called from COleInsertDialog::CreateItem.

For more information, see OleCreateLinkToFile, OLERENDER, and FORMATETC in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleInsertDialog::CreateItem

Other Resources

COleClientItem Members

COleClientItem::CreateNewItem

Call this function to create an embedded item; this function launches the server application that allows the user to create the OLE item.

```
BOOL CreateNewItem(
   REFCLSID clsid,
   OLERENDER render = OLERENDER_DRAW,
   CLIPFORMAT cfFormat = 0,
   LPFORMATETC lpFormatEtc = NULL
);
```

Parameters

clsid

ID that uniquely identifies the type of OLE item to create.

render

Flag specifying how the server will render the OLE item. For the possible values, see OLERENDER in the *Platform SDK*.

cfFormat

Specifies the Clipboard data format to be cached when creating the OLE item.

lpFormatEtc

Pointer to a FORMATETC structure used if *render* is **OLERENDER_FORMAT** or **OLERENDER_DRAW**. Provide a value for this parameter only if you want to specify additional format information beyond the Clipboard format specified by *cfFormat*. If you omit this parameter, default values are used for the other fields in the **FORMATETC** structure.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The framework calls this function if the user chooses OK from the Insert Object dialog box when the Create New button is selected.

For more information, see OleCreate, OLERENDER, and FORMATETC in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleInsertDialog::CreateItem

Other Resources

COleClientItem Members

COleClientItem::CreateStaticFromClipboard

Call this function to create a static item from the contents of the Clipboard.

```
BOOL CreateStaticFromClipboard(
   OLERENDER render = OLERENDER_DRAW,
   CLIPFORMAT cfFormat = 0,
   LPFORMATETC lpFormatEtc = NULL
);
```

Parameters

render

Flag specifying how the server will render the OLE item. For the possible values, see OLERENDER in the *Platform SDK*.

cfFormat

Specifies the Clipboard data format to be cached when creating the OLE item.

lpFormatEtc

Pointer to a FORMATETC structure used if *render* is **OLERENDER_FORMAT** or **OLERENDER_DRAW**. Provide a value for this parameter only if you want to specify additional format information beyond the Clipboard format specified by *cfFormat*. If you omit this parameter, default values are used for the other fields in the **FORMATETC** structure.

Return Value

Nonzero if successful; otherwise 0.

Remarks

A static item contains the presentation data but not the native data; consequently it cannot be edited. You typically call this function if the CreateFromClipboard member function fails.

For more information, see OLERENDER and FORMATETC in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleDataObject::AttachClipboard
COleClientItem::CanPaste
COleClientItem::CreateStaticFromData

Other Resources

COleClientItem Members

COleClientItem::CreateStaticFromData

Call this function to create a static item from a **COleDataObject** object.

```
BOOL CreateStaticFromData(
   COleDataObject* pDataObject,
   OLERENDER render = OLERENDER_DRAW,
   CLIPFORMAT cfFormat = 0,
   LPFORMATETC lpFormatEtc = NULL
);
```

Parameters

pDataObject

Pointer to the COleDataObject object from which the OLE item is to be created.

render

Flag specifying how the server will render the OLE item. For the possible values, see OLERENDER in the *Platform SDK*.

cfFormat

Specifies the Clipboard data format to be cached when creating the OLE item.

lpFormatEtc

Pointer to a FORMATETC structure used if *render* is **OLERENDER_FORMAT** or **OLERENDER_DRAW**. Provide a value for this parameter only if you want to specify additional format information beyond the Clipboard format specified by *cfFormat*. If you omit this parameter, default values are used for the other fields in the **FORMATETC** structure.

Return Value

Nonzero if successful; otherwise 0.

Remarks

A static item contains the presentation data but not the native data; consequently, it cannot be edited. This is essentially the same as CreateStaticFromClipboard except that a static item can be created from an arbitrary **COleDataObject**, not just from the Clipboard.

Used in COlePasteSpecialDialog::CreateItem when Static is selected.

For more information, see OleCreateStaticFromData, OLERENDER, and FORMATETC in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleDataObject::AttachClipboard
COleDataObject Class

Other Resources

COleClientItem Members

COleClientItem::Deactivate

Call this function to deactivate the OLE item and free any associated resources.

```
void Deactivate( );
```

Remarks

You typically deactivate an in-place active OLE item when the user clicks the mouse on the client area outside the bounds of the item. Note that deactivating the OLE item will discard its undo state, making it impossible to call the ReactivateAndUndo member function.

If your application supports undo, do not call **Deactivate**; instead, call DeactivateUI.

For more information, see IOleInPlaceObject::InPlaceDeactivate in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::ReactivateAndUndo
COleClientItem::DeactivateUI

Other Resources

COleClientItem Members

COleClientItem::DeactivateUI

Call this function when the user deactivates an item that was activated in place.

```
void DeactivateUI( );
```

Remarks

This function restores the container application's user interface to its original state, hiding any menus and other controls that were created for in-place activation.

This function does not flush the undo state information for the item. That information is retained so that ReactivateAndUndo can later be used to execute an undo command in the server application, in case the container's undo command is chosen immediately after deactivating the item.

For more information, see IOleInPlaceObject::InPlaceDeactivate in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::ReactivateAndUndo
COleClientItem::Activate

Other Resources

COleClientItem Members

COleClientItem::Delete

Call this function to delete the OLE item from the container document.

```
void Delete(
   BOOL bAutoDelete = TRUE
);
```

Parameters

bAutoDelete

Specifies whether the item is to be removed from the document.

Remarks

This function calls the Release member function, which in turn deletes the C++ object for the item, permanently removing the OLE item from the document. If the OLE item is embedded, the native data for the item is deleted. It always closes a running server; therefore, if the item is an open link, this function closes it.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::Release

Other Resources

COleClientItem Members

COleClientItem::DoDragDrop

Call the **DoDragDrop** member function to perform a drag-and-drop operation.

```
DROPEFFECT DoDragDrop(
   LPCRECT lpItemRect,
   CPoint ptOffset,
   BOOL bIncludeLink = FALSE,
   DWORD dwEffects = DROPEFFECT_COPY | DROPEFFECT_MOVE,
   LPCRECT lpRectStartDrag = NULL
);
```

Parameters

lpItemRect

The item's rectangle on screen in client coordinates (pixels).

ptOffset

The offset from *lpItemRect* where the mouse position was at the time of the drag.

bIncludeLink

Set this to **TRUE** if the link data should be copied to the Clipboard. Set it to **FALSE** if your server application does not support links.

dwEffects

Determines the effects that the drag source will allow in the drag operation.

lpRectStartDrag

Pointer to the rectangle that defines where the drag actually starts. For more information, see the following Remarks section.

Return Value

A **DROPEFFECT** value. If it is **DROPEFFECT_MOVE**, the original data should be removed.

Remarks

The drag-and-drop operation does not start immediately. It waits until the mouse cursor leaves the rectangle specified by *lpRectStartDrag* or until a specified number of milliseconds have passed. If *lpRectStartDrag* is **NULL**, the size of the rectangle is one pixel.

The delay time is specified by a registry key setting. You can change the delay time by calling CWinApp::WriteProfileString or CWinApp::WriteProfileInt. If you do not specify the delay time, a default value of 200 milliseconds is used. Drag delay time is stored as follows:

- Windows NT Drag delay time is stored in HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\NT\CurrentVersion\IniFileMapping\win.ini\Windows\DragDelay.
- Windows 3.x Drag delay time is stored in the WIN.INI file, under the [Windows} section.
- Windows 95/98 Drag delay time is stored in a cached version of WIN.INI.

For more information about how drag delay information is stored in either the registry or the .INI file, see WriteProfileString in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleDataSource::DoDragDrop
COleClientItem::CopyToClipboard

Other Resources

COleClientItem Members

COleClientItem::DoVerb

Call **DoVerb** to execute the specified verb.

```
virtual BOOL DoVerb(
   LONG nVerb,
   CView* pView,
      LPMSG lpMsg = NULL
);
```

Parameters

nVerb

Specifies the verb to execute. It can include one of the following:

Value	Meaning	Symbol
– 0	Primary verb	**OLEIVERB_PRIMARY**
– 1	Secondary verb	(None)
– 1	Display item for editing	**OLEIVERB_SHOW**
– 2	Edit item in separate window	**OLEIVERB_OPEN**
– 3	Hide item	**OLEIVERB_HIDE**

The –1 value is typically an alias for another verb. If open editing is not supported, –2 has the same effect as –1. For additional values, see IOleObject::DoVerb in the *Platform SDK*.

pView

Pointer to the view window; this is used by the server for in-place activation. This parameter should be **NULL** if the container application does not allow in-place activation.

lpMsg

Pointer to the message that caused the item to be activated.

Return Value

Nonzero if the verb was successfully executed; otherwise 0.

Remarks

This function calls the Activate member function to execute the verb. It also catches exceptions and displays a message box to the user if one is thrown.

If the primary verb is Edit and zero is specified in the *nVerb* parameter, the server application is launched to allow the OLE item to be edited. If the container application supports in-place activation, editing can be done in place. If the container does not support in-place activation (or if the Open verb is specified), the server is launched in a separate window and editing can be done there. Typically, when the user of the container application double-clicks the OLE item, the value for the primary verb in the *nVerb* parameter determines which action the user can take. However, if the server supports only one action, it takes that action, no matter which value is specified in the *nVerb* parameter.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::Activate

Other Resources

COleClientItem Members

COleClientItem::Draw

Call this function to draw the OLE item into the specified bounding rectangle using the specified device context.

```
BOOL Draw(
   CDC* pDC,
   LPCRECT lpBounds,
   DVASPECT nDrawAspect = (DVASPECT
)-1
);
```

Parameters

pDC

Pointer to a CDC object used for drawing the OLE item.

lpBounds

Pointer to a CRect object or **RECT** structure that defines the bounding rectangle in which to draw the OLE item (in logical units determined by the device context).

nDrawAspect

Specifies the aspect of the OLE item, that is, how it should be displayed. If *nDrawAspect* is –1, the last aspect set by using SetDrawAspect is used. For more information about possible values for this flag, see SetDrawAspect.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The function may use the metafile representation of the OLE item created by the OnDraw member function of **COleServerItem**.

Typically you use **Draw** for screen display, passing the screen device context as *pDC*. In this case, you need to specify only the first two parameters.

The *lpBounds* parameter identifies the rectangle in the target device context (relative to its current mapping mode). Rendering may involve scaling the picture and can be used by container applications to impose a view that scales between the displayed view and the final printed image.

For more information, see IViewObject::Draw in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::SetExtent
COleServerItem::OnDraw

Other Resources

COleClientItem Members

COleClientItem::GetActiveView

Returns the view on which the item is in-place activated.

```
CView* GetActiveView( ) const;
```

Return Value

A pointer to the view; otherwise **NULL** if the item is not in-place activated.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::IsInPlaceActive
COleClientItem::GetDocument

Other Resources

COleClientItem Members

COleClientItem::GetCachedExtent

Call this function to retrieve the OLE item's size.

```
BOOL GetCachedExtent(
   LPSIZE lpSize,
   DVASPECT nDrawAspect = (DVASPECT
)-1
);
```

Parameters

lpSize

Pointer to a **SIZE** structure or a CSize object that will receive the size information.

nDrawAspect

Specifies the aspect of the OLE item whose bounds are to be retrieved. For possible values, see SetDrawAspect.

Return Value

Nonzero if successful; 0 if the OLE item is blank.

Remarks

This function provides the same information as GetExtent. However, you can call **GetCachedExtent** to get extent information during the processing of other OLE handlers, such as OnChange. The dimensions are in **MM_HIMETRIC** units.

This is possible because **GetCachedExtent** uses the IViewObject2 interface rather than use the IOleObject interface to get the extent of this item. The **IViewObject2** COM object caches the extent information used in the previous call to IViewObject::Draw.

For more information, see IViewObject2::GetExtent in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::GetExtent
COleClientItem::SetExtent
COleServerItem::OnGetExtent

Other Resources

COleClientItem Members

COleClientItem::GetClassID

Returns the class ID of the item into the memory pointed to by *pClassID*.

```
void GetClassID(
   CLSID* pClassID
) const;
```

Parameters

pClassID

Pointer to an identifier of type CLSID to retrieve the class ID. For information on **CLSID**, see the *Platform SDK*.

Remarks

The class ID is a 128-bit number that uniquely identifies the application that edits the item.

For more information, see IPersist::GetClassID in the *Platform SDK*.

See Also

Reference

COleClientItem Class

Hierarchy Chart

Other Resources

COleClientItem Members

COleClientItem::GetClipboardData

Call this function to get a **COleDataSource** object containing all the data that would be placed on the Clipboard by a call to the CopyToClipboard member function.

```
void GetClipboardData(
   COleDataSource* pDataSource,
   BOOL bIncludeLink = FALSE,
   LPPOINT lpOffset = NULL,
   LPSIZE lpSize = NULL
);
```

Parameters

pDataSource

Pointer to a COleDataSource object that will receive the data contained in the OLE item.

bIncludeLink

TRUE if link data should be included; otherwise **FALSE**.

lpOffset

The offset of the mouse cursor from the origin of the object in pixels.

lpSize

The size of the object in pixels.

Remarks

GetClipboardData is called as the default implementation of OnGetClipboardData. Override **OnGetClipboardData** only if you want to offer data formats in addition to those offered by **CopyToClipboard**. Place those formats in the **COleDataSource** object before or after calling **CopyToClipboard**, and then pass the **COleDataSource** object to the COleDataSource::SetClipboard function. For example, if you want the OLE item's position in its container document to accompany it on the Clipboard, you would define your own format for passing that information and place it in the **COleDataSource** before calling **CopyToClipboard**.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleDataSource Class
COleClientItem::CopyToClipboard
COleDataSource::SetClipboard

Other Resources

COleClientItem Members

COleClientItem::GetDocument

Call this function to get a pointer to the document that contains the OLE item.

```
COleDocument* GetDocument( ) const;
```

Return Value

A pointer to the document that contains the OLE item. **NULL** if the item is not part of a document.

Remarks

This pointer allows access to the **COleDocument** object that you passed as an argument to the **COleClientItem** constructor.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::COleClientItem
COleDocument Class
COleLinkingDoc Class

Other Resources

COleClientItem Members

COleClientItem::GetDrawAspect

Call the **GetDrawAspect** member function to determine the current "aspect," or view, of the item.

```
DVASPECT GetDrawAspect( ) const;
```

Return Value

A value from the **DVASPECT** enumeration, whose values are listed in the reference for SetDrawAspect.

Remarks

The aspect specifies how the item is to be rendered.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::SetDrawAspect
COleClientItem::Draw

Other Resources

COleClientItem Members

COleClientItem::GetExtent

Call this function to retrieve the OLE item's size.

```
BOOL GetExtent(
   LPSIZE lpSize,
   DVASPECT nDrawAspect = (DVASPECT
)-1
);
```

Parameters

lpSize

Pointer to a **SIZE** structure or a **CSize** object that will receive the size information.

nDrawAspect

Specifies the aspect of the OLE item whose bounds are to be retrieved. For possible values, see SetDrawAspect.

Return Value

Nonzero if successful; 0 if the OLE item is blank.

Remarks

If the server application was written using the Microsoft Foundation Class Library, this function causes the OnGetExtent member function of the corresponding **COleServerItem** object to be called. Note that the retrieved size may differ from the size last set by the SetExtent member function; the size specified by **SetExtent** is treated as a suggestion. The dimensions are in **MM_HIMETRIC** units.

Note
Do not call **GetExtent** during the processing of an OLE handler, such as OnChange. Call GetCachedExtent instead.

For more information, see IOleObject::GetExtent in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::SetExtent
COleClientItem::GetCachedExtent
COleServerItem::OnGetExtent

Other Resources

COleClientItem Members

COleClientItem::GetIconFromRegistry

Call this member function to retrieve a handle to an icon resource associated with the server of a particular CLSID.

```
HICON GetIconFromRegistry( ) const;
static HICON GetIconFromRegistry(
   CLSID& clsid
);
```

Parameters

clsid

A reference to the CLSID for the server associated with the icon.

Return Value

A valid handle to the icon resource, or **NULL** if the server's icon, or a default icon, can't be found.

Remarks

This member function will not start the server or obtain an icon dynamically, even if the server is already running. Instead, this member function opens the server's executable image and retrieves the static icon associated with the server as it was registered.

See Also

Reference

COleClientItem Class

Hierarchy Chart

Other Resources

COleClientItem Members

COleClientItem::GetIconicMetafile

Retrieves the metafile used for drawing the item's icon.

```
HGLOBAL GetIconicMetafile( );
```

Return Value

A handle to the metafile if successful; otherwise **NULL**.

Remarks

If there is no current icon, a default icon is returned. This is called automatically by the MFC/OLE dialogs and is usually not called directly.

This function also calls SetIconicMetafile to cache the metafile for later use.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::SetIconicMetafile

Other Resources

COleClientItem Members

COleClientItem::GetInPlaceWindow

Call the **GetInPlaceWindow** member function to get a pointer to the window in which the item has been opened for in-place editing.

```
CWnd* GetInPlaceWindow( );
```

Return Value

A pointer to the item's in-place editing window; **NULL** if the item is not active or if its server is unavailable.

Remarks

This function should be called only for items that are in-place active.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::Activate
COleClientItem::Deactivate
COleClientItem::SetItemRects

Other Resources

COleClientItem Members

COleClientItem::GetItemState

Call this function to get the OLE item's current state.

```
UINT GetItemState( ) const;
```

Return Value

A **COleClientItem::ItemState** enumerated value, which can be one of the following: **emptyState**, **loadedState**, **openState**, **activeState**, **activeUIState**. For information about these states, see the article Containers: Client-Item States.

Remarks

To be notified when the OLE item's state changes, use the OnChange member function.

For more information, see the article Containers: Client-Item States.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::OnChange

Other Resources

COleClientItem Members

COleClientItem::GetLastStatus

Returns the status code of the last OLE operation.

```
SCODE GetLastStatus( ) const;
```

Return Value

An **SCODE** value.

Remarks

For member functions that return a **BOOL** value of **FALSE**, or other member functions that return **NULL**, **GetLastStatus** returns more detailed failure information. Be aware that most OLE member functions throw exceptions for more serious errors. The specific information on the interpretation of the **SCODE** depends on the underlying OLE call that last returned an **SCODE** value.

For more information on **SCODE**, see Structure of COM Error Codes in the *Platform SDK* documentation.

See Also

Reference

COleClientItem Class

Hierarchy Chart

Other Resources

COleClientItem Members

COleClientItem::GetLinkUpdateOptions

Call this function to get the current value of the link-update option for the OLE item.

```
OLEUPDATE GetLinkUpdateOptions( );
```

Return Value

One of the following values:

- **OLEUPDATE_ALWAYS** Update the linked item whenever possible. This option supports the Automatic link-update radio button in the Links dialog box.
- **OLEUPDATE_ONCALL** Update the linked item only on request from the container application (when the UpdateLink member function is called). This option supports the Manual link-update radio button in the Links dialog box.

Remarks

This is an advanced operation.

This function is called automatically by the COleLinksDialog class.

For more information, see IOleLink::GetUpdateOptions in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::SetLinkUpdateOptions
COleLinksDialog Class

Other Resources

COleClientItem Members

COleClientItem::GetType

Call this function to determine whether the OLE item is embedded or linked, or static.

```
OLE_OBJTYPE GetType( ) const;
```

Return Value

An unsigned integer with one of the following values:

- **OT_LINK** The OLE item is a link.
- **OT_EMBEDDED** The OLE item is embedded.
- **OT_STATIC** The OLE item is static, that is, it contains only presentation data, not native data, and thus cannot be edited.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::GetUserType

Other Resources

COleClientItem Members

COleClientItem::GetUserType

Call this function to get the user-visible string describing the OLE item's type, such as "Word document."

```
void GetUserType(
   USERCLASSTYPE nUserClassType,
   CString& rString
);
```

Parameters

nUserClassType

A value indicating the desired variant of the string describing the OLE item's type. This can have one of the following values:

- **USERCLASSTYPE_FULL** The full type name displayed to the user.
- **USERCLASSTYPE_SHORT** A short name (15 characters maximum) for use in pop-up menus and the Edit Links dialog box.
- **USERCLASSTYPE_APPNAME** Name of the application servicing the class.

rString

A reference to a CString object to which the string describing the OLE item's type is to be returned.

Remarks

This is often the entry in the system registration database.

If the full type name is requested but not available, the short name is used instead. If no entry for the type of OLE item is found in the registration database, or if there are no user types registered for the type of OLE item, then the user type currently stored in the OLE item is used. If that user type name is an empty string, "Unknown Object" is used.

For more information, see IOleObject::GetUserType in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::GetType

Other Resources

COleClientItem Members

COleClientItem::IsInPlaceActive

Call this function to see whether the OLE item is in-place active.

```
BOOL IsInPlaceActive( ) const;
```

Return Value

Nonzero if the OLE item is in-place active; otherwise 0.

Remarks

It is common to execute different logic depending on whether the item is being edited in place. The function checks whether the current item state is equal to either the **activeState** or the **activeUIState**.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::GetItemState

Other Resources

COleClientItem Members

COleClientItem::IsLinkUpToDate

Call this function to see whether the OLE item is up to date.

```
BOOL IsLinkUpToDate( ) const;
```

Return Value

Nonzero if the OLE item is up to date; otherwise 0.

Remarks

A linked item can be out of date if its source document has been updated. An embedded item that contains links within it can similarly become out of date. The function does a recursive check of the OLE item. Note that determining whether an OLE item is out of date can be as expensive as actually performing an update.

This is called automatically by the COleLinksDialog implementation.

For more information, see IOleObject::IsUpToDate in the *Platform SDK*.

See Also

Reference

COleClientItem Class

Hierarchy Chart

Other Resources

COleClientItem Members

COleClientItem::IsModified

Call this function to see whether the OLE item is dirty (modified since it was last saved).

```
BOOL IsModified( ) const;
```

Return Value

Nonzero if the OLE item is dirty; otherwise 0.

Remarks

For more information, see IPersistStorage::IsDirty in the *Platform SDK*.

See Also

Reference

COleClientItem Class

Hierarchy Chart

Other Resources

COleClientItem Members

COleClientItem::IsOpen

Call this function to see whether the OLE item is open; that is, opened in an instance of the server application running in a separate window.

```
BOOL IsOpen( ) const;
```

Return Value

Nonzero if the OLE item is open; otherwise 0.

Remarks

It is used to determine when to draw the object with a hatching pattern. An open object should have a hatch pattern drawn on top of the object. You can use a CRectTracker object to accomplish this.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::GetItemState
CRectTracker Class

Other Resources

COleClientItem Members

COleClientItem::IsRunning

Call this function to see whether the OLE item is running; that is, whether the item is loaded and running in the server application.

```
BOOL IsRunning( ) const;
```

Return Value

Nonzero if the OLE item is running; otherwise 0.

Remarks

For more information, see OleIsRunning in the *Platform SDK*.

See Also

Reference

COleClientItem Class

Hierarchy Chart

Other Resources

COleClientItem Members

COleClientItem::OnActivate

Called by the framework to notify the item that it has just been activated in place.

```
virtual void OnActivate( );
```

Remarks

Note that this function is called to indicate that the server is running, not to indicate that its user interface has been installed in the container application. At this point, the object does not have an active user interface (is not **activeUIState**). It has not installed its menus or toolbar. The OnActivateUI member function is called when that happens.

The default implementation calls the OnChange member function with **OLE_CHANGEDSTATE** as a parameter. Override this function to perform custom processing when an item becomes in-place active.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::OnDeactivate
COleClientItem::OnDeactivateUI
COleClientItem::OnActivateUI
COleClientItem::CanActivate

Other Resources

COleClientItem Members

COleClientItem::OnActivateUI

The framework calls **OnActivateUI** when the object has entered the active UI state.

```
virtual void OnActivateUI( );
```

Remarks

The object has now installed its tool bar and menus.

The default implementation remembers the server's **HWND** for later **GetServerWindow** calls.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::OnDeactivate
COleClientItem::OnDeactivateUI
COleClientItem::OnActivate
COleClientItem::CanActivate

Other Resources

COleClientItem Members

COleClientItem::OnChange

Called by the framework when the user modifies, saves, or closes the OLE item.

```
virtual void OnChange(
   OLE_NOTIFICATION nCode,
   DWORD dwParam
);
```

Parameters

nCode

The reason the server changed this item. It can have one of the following values:

- **OLE_CHANGED** The OLE item's appearance has changed.
- **OLE_SAVED** The OLE item has been saved.
- **OLE_CLOSED** The OLE item has been closed.
- **OLE_CHANGED_STATE** The OLE item has changed from one state to another.

dwParam

If *nCode* is **OLE_SAVED** or **OLE_CLOSED**, this parameter is not used. If *nCode* is **OLE_CHANGED**, this parameter specifies the aspect of the OLE item that has changed. For possible values, see the *dwParam* parameter of COleClientItem::Draw. If *nCode* is **OLE_CHANGED_STATE**, this parameter is a **COleClientItem::ItemState** enumerated value and describes the state being entered. It can have one of the following values: **emptyState**, **loadedState**, **openState**, **activeState**, or **activeUIState**.

Remarks

(If the server application is written using the Microsoft Foundation Class Library, this function is called in response to the **Notify** member functions of **COleServerDoc** or **COleServerItem**.) The default implementation marks the container document as modified if *nCode* is **OLE_CHANGED** or **OLE_SAVED**.

For **OLE_CHANGED_STATE**, the current state returned from GetItemState will still be the old state, meaning the state that was current prior to this state change.

Override this function to respond to changes in the OLE item's state. Typically you update the item's appearance by invalidating the area in which the item is displayed. Call the base class implementation at the beginning of your override.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::GetItemState
COleServerItem::NotifyChanged
COleServerDoc::NotifyChanged
COleServerDoc::NotifyClosed
COleServerDoc::NotifySaved

Other Resources

COleClientItem Members

COleClientItem::OnChangeItemPosition

Called by the framework to notify the container that the OLE item's extent has changed during in-place activation.

```
virtual BOOL OnChangeItemPosition(
   const CRect& rectPos
);
```

Parameters

rectPos

Indicates the item's position relative to the container application's client area.

Return Value

Nonzero if the item's position is successfully changed; otherwise 0.

Remarks

The default implementation determines the new visible rectangle of the OLE item and calls SetItemRects with the new values. The default implementation calculates the visible rectangle for the item and passes that information to the server.

Override this function to apply special rules to the resize/move operation. If the application is written in MFC, this call results because the server called COleServerDoc::RequestPositionChange.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleServerDoc::RequestPositionChange

Other Resources

COleClientItem Members

COleClientItem::OnDeactivate

Called by the framework when the OLE item transitions from the in-place active state (**activeState**) to the loaded state, meaning that it is deactivated after an in-place activation.

```
virtual void OnDeactivate( );
```

Remarks

Note that this function is called to indicate that the OLE item is closed, not that its user interface has been removed from the container application. When that happens, the OnDeactivateUI member function is called.

The default implementation calls the OnChange member function with **OLE_CHANGEDSTATE** as a parameter. Override this function to perform custom processing when an in-place active item is deactivated. For example, if you support the undo command in your container application, you can override this function to discard the undo state, indicating that the last operation performed on the OLE item cannot be undone once the item is deactivated.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::OnGetWindowContext
COleClientItem::OnDeactivateUI
COleClientItem::OnActivateUI
COleClientItem::OnActivate
COleClientItem::CanActivate
CDocTemplate::SetContainerInfo

Other Resources

COleClientItem Members

COleClientItem::OnDeactivateAndUndo

Called by the framework when the user invokes the undo command after activating the OLE item in place.

```
virtual void OnDeactivateAndUndo( );
```

Remarks

The default implementation calls DeactivateUI to deactivate the server's user interface. Override this function if you are implementing the undo command in your container application. In your override, call the base class version of the function and then undo the last command executed in your application.

For more information, see IOleInPlaceSite::DeactivateAndUndo in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::DeactivateUI

Other Resources

COleClientItem Members

COleClientItem::OnDeactivateUI

Called when the user deactivates an item that was activated in place.

```
virtual void OnDeactivateUI(
   BOOL bUndoable
);
```

Parameters

bUndoable

Specifies whether the editing changes are undoable.

Remarks

This function restores the container application's user interface to its original state, hiding any menus and other controls that were created for in-place activation.

If *bUndoable* is **FALSE**, the container should disable the undo command, in effect discarding the undo state of the container, because it indicates that the last operation performed by the server is not undoable.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::OnActivateUI
COleClientItem::OnDeactivateAndUndo
COleClientItem::OnDeactivate

Other Resources

COleClientItem Members

COleClientItem::OnDiscardUndoState

Called by the framework when the user performs an action that discards the undo state while editing the OLE item.

```
virtual void OnDiscardUndoState( );
```

Remarks

The default implementation does nothing. Override this function if you are implementing the undo command in your container application. In your override, discard the container application's undo state.

If the server was written with the Microsoft Foundation Class Library, the server can cause this function to be called by calling COleServerDoc::DiscardUndoState.

For more information, see IOleInPlaceSite::DiscardUndoState in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleServerDoc::DiscardUndoState

Other Resources

COleClientItem Members

COleClientItem::OnGetClipboardData

Called by the framework to get a **COleDataSource** object containing all the data that would be placed on the Clipboard by a call to either the CopyToClipboard or the DoDragDrop member function.

```
virtual COleDataSource* OnGetClipboardData(
   BOOL bIncludeLink,
   LPPOINT lpOffset,
   LPSIZE lpSize
);
```

Parameters

bIncludeLink

Set this to **TRUE** if link data should be copied to the Clipboard. Set this to **FALSE** if your server application does not support links.

lpOffset

Pointer to the offset of the mouse cursor from the origin of the object in pixels.

lpSize

Pointer to the size of the object in pixels.

Return Value

A pointer to a COleDataSource object containing the Clipboard data.

Remarks

The default implementation of this function calls GetClipboardData.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleDataSource Class
COleClientItem::CopyToClipboard
COleClientItem::GetClipboardData
COleDataSource::SetClipboard

Other Resources

COleClientItem Members

COleClientItem::OnGetClipRect

The framework calls the **OnGetClipRect** member function to get the clipping-rectangle coordinates of the item that is being edited in place.

```
virtual void OnGetClipRect(
   CRect& rClipRect
);
```

Parameters

rClipRect

Pointer to an object of class CRect that will hold the clipping-rectangle coordinates of the item.

Remarks

Coordinates are in pixels relative to the container application window's client area.

The default implementation simply returns the client rectangle of the view on which the item is in-place active.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::OnActivate

Other Resources

COleClientItem Members

COleClientItem::OnGetItemPosition

The framework calls the **OnGetItemPosition** member function to get the coordinates of the item that is being edited in place.

```
virtual void OnGetItemPosition(
   CRect& rPosition
);
```

Parameters

rPosition

Reference to the CRect object that will contain the item's position coordinates.

Remarks

Coordinates are in pixels relative to the container application window's client area.

The default implementation of this function does nothing. Applications that support in-place editing require its implementation.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::OnActivate
COleClientItem::OnActivateUI

Other Resources

COleClientItem Members

COleClientItem::OnGetWindowContext

Called by the framework when an item is activated in place.

```
virtual BOOL OnGetWindowContext(
   CFrameWnd** ppMainFrame,
   CFrameWnd** ppDocFrame,
   LPOLEINPLACEFRAMEINFO lpFrameInfo
);
```

Parameters

ppMainFrame

Pointer to a pointer to the main frame window.

ppDocFrame

Pointer to a pointer to the document frame window.

lpFrameInfo

Pointer to an OLEINPLACEFRAMEINFO structure that will receive frame window information.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This function is used to retrieve information about the OLE item's parent window.

If the container is an MDI application, the default implementation returns a pointer to the CMDIFrameWnd object in *ppMainFrame* and a pointer to the active CMDIChildWnd object in *ppDocFrame*. If the container is an SDI application, the default implementation returns a pointer to the CFrameWnd object in *ppMainFrame* and returns **NULL** in *ppDocFrame*. The default implementation also fills in the members of *lpFrameInfo*.

Override this function only if the default implementation does not suit your application; for example, if your application has a user-interface paradigm that differs from SDI or MDI. This is an advanced overridable.

For more information, see IOleInPlaceSite::GetWindowContext and the OLEINPLACEFRAMEINFO structure in the *Platform SDK*.

See Also

Reference

COleClientItem Class

Hierarchy Chart

Other Resources

COleClientItem Members

COleClientItem::OnInsertMenus

Called by the framework during in-place activation to insert the container application's menus into an empty menu.

```
virtual void OnInsertMenus(
   CMenu* pMenuShared,
   LPOLEMENUGROUPWIDTHS lpMenuWidths
);
```

Parameters

pMenuShared

Points to an empty menu.

lpMenuWidths

Points to an array of six **LONG** values indicating how many menus are in each of the following menu groups: File, Edit, Container, Object, Window, Help. The container application is responsible for the File, Container, and Window menu groups, corresponding to elements 0, 2, and 4 of this array.

Remarks

This menu is then passed to the server, which inserts its own menus, creating a composite menu. This function can be called repeatedly to build several composite menus.

The default implementation inserts into *pMenuShared* the in-place container menus; that is, the File, Container, and Window menu groups. CDocTemplate::SetContainerInfo is used to set this menu resource. The default implementation also assigns the appropriate values to elements 0, 2, and 4 in *lpMenuWidths*, depending on the menu resource. Override this function if the default implementation is not appropriate for your application; for example, if your application does not use document templates for associating resources with document types. If you override this function, you should also override OnSetMenu and OnRemoveMenus. This is an advanced overridable.

For more information, see IOleInPlaceFrame::InsertMenus in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::OnRemoveMenus
COleClientItem::OnSetMenu

Other Resources

COleClientItem Members

COleClientItem::OnRemoveMenus

Called by the framework to remove the container's menus from the specified composite menu when in-place activation ends.

```
virtual void OnRemoveMenus(
   CMenu* pMenuShared
);
```

Parameters

pMenuShared

Points to the composite menu constructed by calls to the OnInsertMenus member function.

Remarks

The default implementation removes from *pMenuShared* the in-place container menus, that is, the File, Container, and Window menu groups. Override this function if the default implementation is not appropriate for your application; for example, if your application does not use document templates for associating resources with document types. If you override this function, you should probably override OnInsertMenus and OnSetMenu as well. This is an advanced overridable.

The submenus on *pMenuShared* may be shared by more than one composite menu if the server has repeatedly called **OnInsertMenus**. Therefore you should not delete any submenus in your override of **OnRemoveMenus**; you should only detach them.

For more information, see IOleInPlaceFrame::RemoveMenus in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::OnInsertMenus
COleClientItem::OnSetMenu

Other Resources

COleClientItem Members

COleClientItem::OnScrollBy

Called by the framework to scroll the OLE item in response to requests from the server.

```
virtual BOOL OnScrollBy(
   CSize sizeExtent
);
```

Parameters

sizeExtent

Specifies the distances, in pixels, to scroll in the x and y directions.

Return Value

Nonzero if the item was scrolled; 0 if the item could not be scrolled.

Remarks

For example, if the OLE item is partially visible and the user moves outside the visible region while performing in-place editing, this function is called to keep the cursor visible. The default implementation does nothing. Override this function to scroll the item by the specified amount. Note that as a result of scrolling, the visible portion of the OLE item can change. Call SetItemRects to update the item's visible rectangle.

For more information, see IOleInPlaceSite::Scroll in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::SetItemRects

Other Resources

COleClientItem Members

COleClientItem::OnSetMenu

Called by the framework two times when in-place activation begins and ends; the first time to install the composite menu and the second time (with *holemenu* equal to **NULL**) to remove it.

```
virtual void OnSetMenu(
   CMenu* pMenuShared,
   HOLEMENU holemenu,
   HWND hwndActiveObject
);
```

Parameters

pMenuShared

Pointer to the composite menu constructed by calls to the OnInsertMenus member function and the **InsertMenu** function.

holemenu

Handle to the menu descriptor returned by the **OleCreateMenuDescriptor** function, or **NULL** if the dispatching code is to be removed.

hwndActiveObject

Handle to the editing window for the OLE item. This is the window that will receive editing commands from OLE.

Remarks

The default implementation installs or removes the composite menu and then calls the OleSetMenuDescriptor function to install or remove the dispatching code. Override this function if the default implementation is not appropriate for your application. If you override this function, you should probably override OnInsertMenus and OnRemoveMenus as well. This is an advanced overridable.

For more information, see OleCreateMenuDescriptor, OleSetMenuDescriptor, and IOleInPlaceFrame::SetMenu in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::OnInsertMenus
COleClientItem::OnRemoveMenus

Other Resources

COleClientItem Members

COleClientItem::OnShowControlBars

Called by the framework to show and hide the container application's control bars.

```
virtual BOOL OnShowControlBars(
   CFrameWnd* pFrameWnd,
   BOOL bShow
);
```

Parameters

pFrameWnd

Pointer to the container application's frame window. This can be either a main frame window or an MDI child window.

bShow

Specifies whether control bars are to be shown or hidden.

Return Value

Nonzero if the function call causes a change in the control bars' state; 0 if the call causes no change, or if *pFrameWnd* does not point to the container's frame window.

Remarks

This function returns 0 if the control bars are already in the state specified by *bShow*. This would occur, for example, if the control bars are hidden and *bShow* is **FALSE**.

The default implementation removes the toolbar from the top-level frame window.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::OnInsertMenus
COleClientItem::OnSetMenu
COleClientItem::OnRemoveMenus
COleClientItem::OnUpdateFrameTitle

Other Resources

COleClientItem Members

COleClientItem::OnShowItem

Called by the framework to display the OLE item, making it totally visible during editing.

```
virtual void OnShowItem( );
```

Remarks

It is used when your container application supports links to embedded items (that is, if you have derived your document class from COleLinkingDoc). This function is called during in-place activation or when the OLE item is a link source and the user wants to edit it. The default implementation activates the first view on the container document. Override this function to scroll the document so that the OLE item is visible.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleLinkingDoc Class

Other Resources

COleClientItem Members

COleClientItem::OnUpdateFrameTitle

Called by the framework during in-place activation to update the frame window's title bar.

```
virtual BOOL OnUpdateFrameTitle( );
```

Return Value

Nonzero if this function successfully updated the frame title, otherwise zero.

Remarks

The default implementation does not change the frame window title. Override this function if you want a different frame title for your application, for example "*server app* - *item* in *docname*" (as in, "Microsoft Excel - spreadsheet in REPORT.DOC"). This is an advanced overridable.

See Also

Reference

COleClientItem Class

Hierarchy Chart

Other Resources

COleClientItem Members

COleClientItem::ReactivateAndUndo

Call this function to reactivate the OLE item and undo the last operation performed by the user during in-place editing.

```
BOOL ReactivateAndUndo( );
```

Return Value

Nonzero if successful; otherwise 0.

Remarks

If your container application supports the undo command, call this function if the user chooses the undo command immediately after deactivating the OLE item.

If the server application is written with the Microsoft Foundation Class Libraries, this function causes the server to call COleServerDoc::OnReactivateAndUndo.

For more information, see IOleInPlaceObject::ReactivateAndUndo in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleServerDoc::OnReactivateAndUndo
COleClientItem::OnDeactivateAndUndo

Other Resources

COleClientItem Members

COleClientItem::Release

Call this function to clean up resources used by the OLE item.

```
virtual void Release(
   OLECLOSE dwCloseOption = OLECLOSE_NOSAVE
);
```

Parameters

dwCloseOption

Flag specifying under what circumstances the OLE item is saved when it returns to the loaded state. For a list of possible values, see COleClientItem::Close.

Remarks

Release is called by the **COleClientItem** destructor.

For more information, see IUnknown::Release in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::Close
COleClientItem::Delete

Other Resources

COleClientItem Members

COleClientItem::Reload

Closes and reloads the item.

```
BOOL Reload( );
```

Return Value

Nonzero if successful; otherwise 0.

Remarks

Call the **Reload** function after activating the item as an item of another type by a call to ActivateAs.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::ActivateAs

Other Resources

COleClientItem Members

COleClientItem::Run

Runs the application associated with this item.

```
void Run( );
```

Remarks

Call the **Run** member function to launch the server application before activating the item. This is done automatically by Activate and DoVerb, so it is usually not necessary to call this function. Call this function if it is necessary to run the server in order to set an item attribute, such as SetExtent, before executing DoVerb.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::IsRunning

Other Resources

COleClientItem Members

COleClientItem::SetDrawAspect

Call the **SetDrawAspect** member function to set the "aspect," or view, of the item.

```
virtual void SetDrawAspect(
   DVASPECT nDrawAspect
);
```

Parameters

nDrawAspect

A value from the **DVASPECT** enumeration. This parameter can have one of the following values:

- **DVASPECT_CONTENT** Item is represented in such a way that it can be displayed as an embedded object inside its container.
- **DVASPECT_THUMBNAIL** Item is rendered in a "thumbnail" representation so that it can be displayed in a browsing tool.
- **DVASPECT_ICON** Item is represented by an icon.
- **DVASPECT_DOCPRINT** Item is represented as if it were printed using the Print command from the File menu.

Remarks

The aspect specifies how the item is to be rendered by Draw when the default value for that function's *nDrawAspect* argument is used.

This function is called automatically by the Change Icon (and other dialogs that call the Change Icon dialog directly) to enable the iconic display aspect when requested by the user.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::GetDrawAspect
COleClientItem::Draw

Other Resources

COleClientItem Members

COleClientItem::SetExtent

Call this function to specify how much space is available to the OLE item.

```
void SetExtent(
   const CSize& size,
   DVASPECT nDrawAspect = DVASPECT_CONTENT
);
```

Parameters

size

A CSize object that contains the size information.

nDrawAspect

Specifies the aspect of the OLE item whose bounds are to be set. For possible values, see SetDrawAspect.

Remarks

If the server application was written using the Microsoft Foundation Class Library, this causes the OnSetExtent member function of the corresponding **COleServerItem** object to be called. The OLE item can then adjust its display accordingly. The dimensions must be in **MM_HIMETRIC** units. Call this function when the user resizes the OLE item or if you support some form of layout negotiation.

For more information, see IOleObject::SetExtent in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::GetExtent
COleClientItem::GetCachedExtent
COleServerItem::OnSetExtent

Other Resources

COleClientItem Members

COleClientItem::SetHostNames

Call this function to specify the name of the container application and the container's name for an embedded OLE item.

```
void SetHostNames(
   LPCTSTR lpszHost,
   LPCTSTR lpszHostObj
);
```

Parameters

lpszHost

Pointer to the user-visible name of the container application.

lpszHostObj

Pointer to an identifying string of the container that contains the OLE item.

Remarks

If the server application was written using the Microsoft Foundation Class Library, this function calls the OnSetHostNames member function of the **COleServerDoc** document that contains the OLE item. This information is used in window titles when the OLE item is being edited. Each time a container document is loaded, the framework calls this function for all the OLE items in the document. **SetHostNames** is applicable only to embedded items. It is not necessary to call this function each time an embedded OLE item is activated for editing.

This is also called automatically with the application name and document name when an object is loaded or when a file is saved under a different name. Accordingly, it is not usually necessary to call this function directly.

For more information, see IOleObject::SetHostNames in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleServerDoc::OnSetHostNames

Other Resources

COleClientItem Members

COleClientItem::SetIconicMetafile

Caches the metafile used for drawing the item's icon.

```
BOOL SetIconicMetafile(
   HGLOBAL hMetaPict
);
```

Parameters

hMetaPict

A handle to the metafile used for drawing the item's icon.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Use GetIconicMetafile to retrieve the metafile.

The *hMetaPict* parameter is copied into the item; therefore, *hMetaPict* must be freed by the caller.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::GetIconicMetafile

Other Resources

COleClientItem Members

COleClientItem::SetItemRects

Call this function to set the bounding rectangle or the visible rectangle of the OLE item.

```
BOOL SetItemRects(
   LPCRECT lpPosRect = NULL,
   LPCRECT lpClipRect = NULL
);
```

Parameters

lprcPosRect

Pointer to the rectangle containing the bounds of the OLE item relative to its parent window, in client coordinates.

lprcClipRect

Pointer to the rectangle containing the bounds of the visible portion of the OLE item relative to its parent window, in client coordinates.

Return Value

Nonzero if successful; otherwise, 0.

Remarks

This function is called by the default implementation of the OnChangeItemPosition member function. You should call this function whenever the position or visible portion of the OLE item changes. Usually this means that you call it from your view's OnSize and OnScrollBy member functions.

For more information, see IOleInPlaceObject::SetObjectRects in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::OnChangeItemPosition
COleClientItem::OnGetItemPosition

Other Resources

COleClientItem Members

COleClientItem::SetLinkUpdateOptions

Call this function to set the link-update option for the presentation of the specified linked item.

```
void SetLinkUpdateOptions(
   OLEUPDATE dwUpdateOpt
);
```

Parameters

dwUpdateOpt

The value of the link-update option for this item. This value must be one of the following:

- **OLEUPDATE_ALWAYS** Update the linked item whenever possible. This option supports the Automatic link-update radio button in the Links dialog box.
- **OLEUPDATE_ONCALL** Update the linked item only on request from the container application (when the UpdateLink member function is called). This option supports the Manual link-update radio button in the Links dialog box.

Remarks

Typically, you should not change the update options chosen by the user in the Links dialog box.

For more information, see IOleLink::SetUpdateOptions in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::GetLinkUpdateOptions
COleLinksDialog Class

Other Resources

COleClientItem Members

COleClientItem::SetPrintDevice

Call this function to change the print-target device for this item.

```
BOOL SetPrintDevice(
   const DVTARGETDEVICE* ptd
);
BOOL SetPrintDevice(
   const PRINTDLG* ppd
);
```

Parameters

ptd

Pointer to a DVTARGETDEVICE data structure, which contains information about the new print-target device. Can be **NULL**.

ppd

Pointer to a PRINTDLG data structure, which contains information about the new print-target device. Can be **NULL**.

Return Value

Nonzero if the function was successful; otherwise 0.

Remarks

This function updates the print-target device for the item but does not refresh the presentation cache. To update the presentation cache for an item, call UpdateLink.

The arguments to this function contain information that the OLE system uses to identify the target device. The **PRINTDLG** structure contains information that Windows uses to initialize the common Print dialog box. After the user closes the dialog box, Windows returns information about the user's selections in this structure. The **m_pd** member of a CPrintDialog object is a **PRINTDLG** structure.

For more information about this structure, see PRINTDLG in the *Platform SDK*.

For more information, see DVTARGETDEVICE in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleClientItem::UpdateLink
CPrintDialog Class

Other Resources

COleClientItem Members

COleClientItem::UpdateLink

Call this function to update the presentation data of the OLE item immediately.

```
BOOL UpdateLink( );
```

Return Value

Nonzero on success; otherwise 0.

Remarks

For linked items, the function finds the link source to obtain a new presentation for the OLE item. This process may involve running one or more server applications, which could be time-consuming. For embedded items, the function operates recursively, checking whether the embedded item contains links that might be out of date and updating them. The user can also manually update individual links using the Links dialog box.

For more information, see IOleLink::Update in the *Platform SDK*.

See Also

Reference

COleClientItem Class
Hierarchy Chart
COleLinksDialog Class

Other Resources

COleClientItem Members

COleCmdUI Class

Implements a method for MFC to update the state of user-interface objects related to the **IOleCommandTarget**-driven features of your application.

```
class COleCmdUI : public CCmdUI
```

Remarks

In an application that is not enabled for DocObjects, when the user views a menu in the application, MFC processes **UPDATE_COMMAND_UI** notifcations. Each notification is given a CCmdUI object that can be manipulated to reflect the state of a particular command. However, when your application is enabled for DocObjects, MFC processes **UPDATE_OLE_COMMAND_UI** notifications and assigns **COleCmdUI** objects.

COleCmdUI allows a DocObject to receive commands that originate in its container's user interface (such as FileNew, Open, Print, and so on), and allows a container to receive commands that originate in the DocObject's user interface. Although **IDispatch** could be used to dispatch the same commands, **IOleCommandTarget** provides a simpler way to query and execute because it relies on a standard set of commands, usually without arguments, and no type information is involved. **COleCmdUI** can be used to enable, update, and set other properties of DocObject user interface commands. When you want to invoke the command, call COleServerDoc::OnExecOleCmd.

For further information on DocObjects, see CDocObjectServer and CDocObjectServerItem. Also see Internet First Steps: Active Documents and Active Documents.

Requirements

Header: afxdocobj.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CCmdUI Class

Hierarchy Chart

Other Resources

COleCmdUI Members

COleCmdUI Members

Base Class Members

CCmdUI Members

Constructors

COleCmdUI	Constructs a **COleCmdUI** object.

Overridables

Enable	Sets or clears the enable command flag.
SetCheck	Sets the state of an on/off toggle command.
SetText	Returns a text name or status string for a command.

See Also

Reference

COleCmdUI Class

Hierarchy Chart

COleCmdUI Member Functions

For information about the member functions in **COleCmdUI**, see COleCmdUI Members.

COleCmdUI::COleCmdUI

Constructs a **COleCmdUI** object associated with a particular user-interface command.

```
COleCmdUI(
   OLECMD* rgCmds,
   ULONG cCmds,
   const GUID* m_pGroup
);
```

Parameters

rgCmds

A list of supported commands associated with the given GUID. The **OLECMD** structure associates commands with command flags.

cCmds

The count of commands in *rgCmds*.

pGroup

A pointer to a GUID that identifies a set of commands.

Remarks

The **COleCmdUI** object provides a programmatic interface for updating DocObject user-interface objects such as menu items or control-bar buttons. The user-interface objects can be enabled, disabled, checked, and/or cleared through the **COleCmdUI** object.

See Also

Reference

COleCmdUI Class

Hierarchy Chart

Other Resources

COleCmdUI Members

COleCmdUI::Enable

Call this function to set the command flag of the **COleCmdUI** object to **OLECOMDF_ENABLED**, which tells the interface the command is available and enabled, or to clear the command flag.

```
virtual void Enable(
   BOOL bOn
);
```

Parameters

bOn

Indicates whether the command associated with the **COleCmdUI** object should be enabled or disabled. Nonzero enables the command; 0 disables the command.

See Also

Reference

COleCmdUI Class

Hierarchy Chart

Other Resources

COleCmdUI Members

COleCmdUI::SetCheck

Call this function to set the state of an on/off toggle command.

```
virtual void SetCheck(
   int nCheck
);
```

Parameters

nCheck

A value determining the state to set an on/off toggle command. Values are:

Value	Description
1	Sets the command to on.
2	Sets the command to indeterminate; the state cannot be determined because the attribute of this command is in both on and off states in the relevant selection.
any other value	Sets the command to off.

See Also

Reference

COleCmdUI Class

Hierarchy Chart

COleCmdUI::SetText

Other Resources

COleCmdUI Members

COleCmdUI::SetText

Call this function to return a text name or status string for a command.

```
virtual void SetText(
   LPCTSTR lpszText
);
```

Parameters

lpszText

A pointer to the text to be used with the command.

See Also

Reference

COleCmdUI Class

Hierarchy Chart

COleCmdUI::SetCheck

Other Resources

COleCmdUI Members

COleControl Class

A powerful base class for developing OLE controls.

```
class COleControl : public CWnd
```

Remarks

Derived from **CWnd**, this class inherits all the functionality of a Windows window object plus additional functionality specific to OLE, such as event firing and the ability to support methods and properties.

OLE controls can be inserted into OLE container applications and communicate with the container by using a two-way system of event firing and exposing methods and properties to the container. Note that standard OLE containers only support the basic functionality of an OLE control. They are unable to support extended features of an OLE control. Event firing occurs when events are sent to the container as a result of certain actions taking place in the control. In turn, the container communicates with the control by using an exposed set of methods and properties analogous to the member functions and data members of a C++ class. This approach allows the developer to control the appearance of the control and notify the container when certain actions occur.

Windowless Controls

OLE controls can be used in-place active without a window. Windowless controls have significant advantages:

- Windowless controls can be transparent and non-rectangular
- Windowless controls reduce instance size and creation time of the object

Controls do not need a window. Services that a window offers can easily be provided via a single shared window (usually the container's) and a bit of dispatching code. Having a window is mostly an unnecessary complication on the object.

When windowless activation is used, the container (which does have a window) is responsible for providing services that would otherwise have been provided by the control's own window. For example, if your control needs to query the keyboard focus, query the mouse capture, or obtain a device context, these operations are managed by the container. The **COleControl** windowless-operation member functions invoke these operations on the container.

When windowless activation is enabled, the container delegates input messages to the control's **IOleInPlaceObjectWindowless** interface (an extension of IOleInPlaceObject for windowless support). **COleControl**'s implementation of this interface will dispatch these messages through your control's message map, after adjusting the mouse coordinates appropriately. You can process these messages like ordinary window messages, by adding the corresponding entries to the message map.

In a windowless control, you should always use the **COleControl** member functions instead of the corresponding **CWnd** member functions or their related Windows API functions.

OLE control objects can also create a window only when they become active, but the amount of work needed for the inactive-active transition goes up and the speed of the transition goes down. There are cases when this is a problem: as an example, consider a grid of text boxes. When cursoring up and down through the column, each control must be in-place activated and then deactivated. The speed of the inactive/active transition will directly affect the scrolling speed.

For more information on developing an OLE control framework, see the articles MFC ActiveX Controls and Overview: Creating an MFC ActiveX Control Program. For information on optimizing OLE controls, including windowless and flicker-free controls, see MFC ActiveX Controls: Optimization.

Requirements

Header: afxctl.h

See Also

Tasks

CIRC Sample: ActiveX Control

TESTHELP Sample: ActiveX Control with Tooltips and Help

Reference

COlePropertyPage Class

CWnd Class

Hierarchy Chart
CFontHolder Class
CPictureHolder Class
Other Resources
COleControl Members

COleControl Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Construction/Destruction

COleControl	Creates a **COleControl** object.
RecreateControlWindow	Destroys and re-creates the control's window.

Initialization

InitializeIIDs	Informs the base class of the IIDs the control will use.
ReparentControlWindow	Resets the parent of the control window.
ResetStockProps	Initializes **COleControl** stock properties to their default values.
ResetVersion	Initializes the version number to a given value.
SetInitialSize	Sets the size of an OLE control when first displayed in a container.

Control Modification Functions

GetControlFlags	Retrieves the control flag settings.
IsModified	Determines if the control state has changed.
SetModifiedFlag	Changes the modified state of a control.

Persistence

ExchangeExtent	Serializes the control's width and height.
ExchangeStockProps	Serializes the control's stock properties.
ExchangeVersion	Serializes the control's version number.
IsConvertingVBX	Allows specialized loading of an OLE control.
SerializeExtent	Serializes or initializes the display space for the control.
SerializeStockProps	Serializes or initializes the **COleControl** stock properties.
SerializeVersion	Serializes or initializes the control's version information.
SetModifiedFlag	Changes the modified state of a control.
WillAmbientsBeValidDuringLoad	Determines whether ambient properties will be available the next time the control is loaded.

Update/Painting Functions

DoSuperclassPaint	Redraws an OLE control that has been subclassed from a Windows control.

InvalidateControl	Invalidates an area of the displayed control, causing it to be redrawn.
IsOptimizedDraw	Indicates whether the container supports optimized drawing for the current drawing operation.
SelectFontObject	Selects a custom Font property into a device context.
SelectStockFont	Selects the stock Font property into a device context.
TranslateColor	Converts an **OLE_COLOR** value to a **COLORREF** value.

Dispatch Exceptions

GetNotSupported	Prevents access to a control's property value by the user.
SetNotPermitted	Indicates that an edit request has failed.
SetNotSupported	Prevents modification to a control's property value by the user.
ThrowError	Signals that an error has occurred in an OLE control.
WindowProc	Provides a Windows procedure for a **COleControl** object.

Ambient Property Functions

AmbientAppearance	Retrieves the current appearance of the control.
AmbientBackColor	Returns the value of the ambient BackColor property.
AmbientDisplayName	Returns the name of the control as specified by the container.
AmbientFont	Returns the value of the ambient Font property.
AmbientForeColor	Returns the value of the ambient ForeColor property.
AmbientLocaleID	Returns the container's locale ID.
AmbientScaleUnits	Returns the type of units used by the container.
AmbientShowGrabHandles	Determines if grab handles should be displayed.
AmbientShowHatching	Determines if hatching should be displayed.
AmbientTextAlign	Returns the type of text alignment specified by the container.
AmbientUIDead	Determines if the control should respond to user-interface actions.
AmbientUserMode	Determines the mode of the container.
GetAmbientProperty	Returns the value of the specified ambient property.

Event Firing Functions

FireClick	Fires the stock Click event.
FireDblClick	Fires the stock DblClick event.

FireError	Fires the stock Error event.
FireEvent	Fires a custom event.
FireKeyDown	Fires the stock KeyDown event.
FireKeyPress	Fires the stock KeyPress event.
FireKeyUp	Fires the stock KeyUp event.
FireMouseDown	Fires the stock MouseDown event.
FireMouseMove	Fires the stock MouseMove event.
FireMouseUp	Fires the stock MouseUp event.
FireReadyStateChange	Fires an event when the control's ready state changes.

Stock Methods/Properties

DoClick	Implementation of the stock DoClick method.
GetAppearance	Returns the value of the stock Appearance property.
GetBackColor	Returns the value of the stock BackColor property.
GetBorderStyle	Returns the value of the stock BorderStyle property.
GetEnabled	Returns the value of the stock Enabled property.
GetFont	Returns the value of the stock Font property.
GetFontTextMetrics	Returns the metrics of a **CFontHolder** object.
GetForeColor	Returns the value of the stock ForeColor property.
GetHwnd	Returns the value of the stock hWnd property.
GetStockTextMetrics	Returns the metrics of the stock Font property.
GetText	Returns the value of the stock Text or Caption property.
InternalGetFont	Returns a **CFontHolder** object for the stock Font property.
InternalGetText	Retrieves the stock Caption or Text property.
Refresh	Forces a repaint of a control's appearance.
SelectStockFont	Selects the control's stock Font property into a device context.
SetAppearance	Sets the value of the stock Appearance property.
SetBackColor	Sets the value of the stock BackColor property.

SetBorderStyle	Sets the value of the stock BorderStyle property.
SetEnabled	Sets the value of the stock Enabled property.
SetFont	Sets the value of the stock Font property.
SetForeColor	Sets the value of the stock ForeColor property.
SetText	Sets the value of the stock Text or Caption property.

OLE Control Sizing Functions

GetControlSize	Returns the position and size of the OLE control.
GetRectInContainer	Returns the control's rectangle relative to its container.
SetControlSize	Sets the position and size of the OLE control.
SetRectInContainer	Sets the control's rectangle relative to its container.

OLE Data Binding Functions

BoundPropertyChanged	Notifies the container that a bound property has been changed.
BoundPropertyRequestEdit	Requests permission to edit the property value.

Simple Frame Functions

EnableSimpleFrame	Enables simple frame support for a control.

OLE Control Site Functions

ControlInfoChanged	Call this function after the set of mnemonics handled by the control has changed.
GetClientSite	Queries an object for the pointer to its current client site within its container.
GetExtendedControl	Retrieves a pointer to an extended control object belonging to the container.
LockInPlaceActive	Determines if your control can be deactivated by the container.
TransformCoords	Transforms coordinate values between a container and the control.

Modal Dialog Functions

PostModalDialog	Notifies the container that a modal dialog box has been closed.
PreModalDialog	Notifies the container that a modal dialog box is about to be displayed.

Windowless Operations

ClipCaretRect	Adjusts a caret rectangle if it is overlapped by a control.
GetCapture	Determines whether a windowless, activated control object has the mouse capture.
GetClientRect	Retrieves the size of the control's client area.
GetDC	Provides a means for a windowless control to get a device context from its container.
GetFocus	Determines whether the control has the focus.

GetWindowlessDropTarget	Override to allow a windowless control to be the target of drag and drop operations.
InvalidateRgn	Invalidates the container window's client area within the given region. Can be used to redraw windowless controls in the region.
OnWindowlessMessage	Processes window messages (other than mouse and keyboard messages) for windowless controls.
ReleaseCapture	Releases mouse capture.
ReleaseDC	Releases the display device context of a container of a windowless control.
ScrollWindow	Allows a windowless control to scroll an area within its in-place active image on the display.
SetCapture	Causes the control's container window to take possession of the mouse capture on the control's behalf.
SetFocus	Causes the control's container window to take possession of the input focus on the control's behalf.

Inactive Pointer Handling Functions

ClientToParent	Translates a point relative to the control's origin to a point relative to its container's origin.
GetActivationPolicy	Alters the default activation behavior of a control that supports the **IPointerInactive** interface.
GetClientOffset	Retrieves the difference between the upper left corner of the control's rectangular area and the upper left corner of its client area.
OnInactiveMouseMove	Override to have the container for the inactive control under the mouse pointer dispatch **WM_MOUSEMOVE** messages to the control.
OnInactiveSetCursor	Override to have the container for the inactive control under the mouse pointer dispatch **WM_SETCURSOR** messages to the control.
ParentToClient	Translates a point relative to the container's origin to a point relative to the control's origin.

Asynchronous Control Functions

GetReadyState	Returns the control's readiness state.
InternalSetReadyState	Sets the control's readiness state and fires the ready-state-change event.
Load	Resets any previous asynchronous data and initiates a new load of the control's asynchronous property.

Overridables

DisplayError	Displays stock Error events to the control's user.
DoPropExchange	Serializes the properties of a **COleControl** object.
GetClassID	Retrieves the OLE class ID of the control.
GetMessageString	Provides status bar text for a menu item.
IsInvokeAllowed	Enables automation method invocation.
IsSubclassedControl	Called to determine if the control subclasses a Windows control.

OnClick	Called to fire the stock Click event.
OnDoVerb	Called after a control verb has been executed.
OnDraw	Called when a control is requested to redraw itself.
OnDrawMetafile	Called by the container when a control is requested to redraw itself using a metafile device context.
OnEdit	Called by the container to UI Activate an OLE control.
OnEnumVerbs	Called by the container to enumerate a control's verbs.
OnEventAdvise	Called when event handlers are connected or disconnected from a control.
OnKeyDownEvent	Called after the stock KeyDown event has been fired.
OnKeyPressEvent	Called after the stock KeyPress event has been fired.
OnKeyUpEvent	Called after the stock KeyUp event has been fired.
OnProperties	Called when the control's "Properties" verb has been invoked.
OnResetState	Resets a control's properties to the default values.

Change Notification Functions

OnAppearanceChanged	Called when the stock Appearance property is changed.
OnBackColorChanged	Called when the stock BackColor property is changed.
OnBorderStyleChanged	Called when the stock BorderStyle property is changed.
OnEnabledChanged	Called when the stock Enabled property is changed.
OnFontChanged	Called when the stock Font property is changed.
OnForeColorChanged	Called when the stock ForeColor property is changed.
OnTextChanged	Called when the stock Text or Caption property is changed.

OLE Interface Notification Functions

OnAmbientPropertyChange	Called when an ambient property is changed.
OnClose	Notifies the control that **IOleControl::Close** has been called.
OnFreezeEvents	Called when a control's events are frozen or unfrozen.
OnGetControlInfo	Provides mnemonic information to the container.
OnMnemonic	Called when a mnemonic key of the control has been pressed.
OnRenderData	Called by the framework to retrieve data in the specified format.
OnRenderFileData	Called by the framework to retrieve data from a file in the specified format.

OnRenderGlobalData	Called by the framework to retrieve data from global memory in the specified format.
OnSetClientSite	Notifies the control that **IOleControl::SetClientSite** has been called.
OnSetData	Replaces the control's data with another value.
OnSetExtent	Called after the control's extent has changed.
OnSetObjectRects	Called after the control's dimensions have been changed.

IViewObject Interface Notification Overridables

OnGetColorSet	Notifies the control that **IOleObject::GetColorSet** has been called.
OnGetNaturalExtent	Override to retrieve the control's display size closest to the proposed size and extent mode.
OnGetViewExtent	Override to retrieve the size of the control's display areas (can be used to enable two-pass drawing).
OnGetViewRect	Override to convert control's size into a rectangle starting at a specific position.
OnGetViewStatus	Override to retrieve the control's view status.
OnQueryHitPoint	Override to query whether a control's display overlaps a given point.
OnQueryHitRect	Override to query whether a control's display overlaps any point in a given rectangle.

In-Place Activation Functions

OnGetInPlaceMenu	Requests the handle of the control's menu that will be merged with the container menu.
OnHideToolBars	Called by the container when the control is UI deactivated.
OnShowToolBars	Called when the control has been UI activated.

Property Browsing Functions

OnGetDisplayString	Called to obtain a string to represent a property value.
OnGetPredefinedStrings	Returns strings representing possible values for a property.
OnGetPredefinedValue	Returns the value corresponding to a predefined string.
OnMapPropertyToPage	Indicates which property page to use for editing a property.

See Also

Reference

COleControl Class

Hierarchy Chart

COleControl Member Functions

For information about the member functions in **COleControl**, see COleControl Members.

COleControl::AmbientBackColor

Returns the value of the ambient BackColor property.

```
OLE_COLOR AmbientBackColor( );
```

Return Value

The current value of the container's ambient BackColor property, if any. If the property is not supported, this function returns the system-defined Windows background color.

Remarks

The ambient BackColor property is available to all controls and is defined by the container. Note that the container is not required to support this property.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::TranslateColor
COleControl::GetBackColor
COleControl::AmbientForeColor

Other Resources

COleControl Members

COleControl::AmbientDisplayName

The name the container has assigned to the control can be used in error messages displayed to the user.

```
CString AmbientDisplayName( );
```

Return Value

The name of the OLE control. The default is a zero-length string.

Remarks

Note that the container is not required to support this property.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::AmbientFont

Returns the value of the ambient Font property.

```
LPFONTDISP AmbientFont( );
```

Return Value

A pointer to the container's ambient Font dispatch interface. The default value is **NULL**. If the return is not equal to **NULL**, you are responsible for releasing the font by calling its IUnknown::Release member function.

Remarks

The ambient Font property is defined by the container and available to all controls.Note that the container is not required to support this property.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetFont
COleControl::SetFont

Other Resources

COleControl Members

COleControl::AmbientForeColor

Returns the value of the ambient ForeColor property.

```
OLE_COLOR AmbientForeColor( );
```

Return Value

The current value of the container's ambient ForeColor property, if any. If not supported, this function returns the system-defined Windows text color.

Remarks

The ambient ForeColor property is available to all controls and is defined by the container. Note that the container is not required to support this property.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::AmbientBackColor
COleControl::GetForeColor
COleControl::TranslateColor

Other Resources

COleControl Members

COleControl::AmbientLocaleID

Returns the container's locale ID.

```
LCID AmbientLocaleID( );
```

Return Value

The value of the container's LocaleID property, if any. If this property is not supported, this function returns 0.

Remarks

The control can use the LocaleID to adapt its user interface for specific locales. Note that the container is not required to support this property.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::AmbientAppearance

Retrieves the current appearance setting for the control object.

```
short AmbientAppearance( );
```

Return Value

The appearance of the control:

- **0** Flat appearance
- **1** 3D appearance

Remarks

Call this function to retrieve the current value of the **DISPID_AMBIENT_APPEARANCE** property for the control.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::AmbientScaleUnits

Returns the type of units used by the container.

```
CString AmbientScaleUnits( );
```

Return Value

A string containing the ambient ScaleUnits of the container. If this property is not supported, this function returns a zero-length string.

Remarks

The container's ambient ScaleUnits property can be used to display positions or dimensions, labeled with the chosen unit, such as twips or centimeters. Note that the container is not required to support this property.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::TransformCoords

Other Resources

COleControl Members

COleControl::AmbientShowGrabHandles

Determines whether the container allows the control to display grab handles for itself when active.

```
BOOL AmbientShowGrabHandles( );
```

Return Value

Nonzero if grab handles should be displayed; otherwise 0. If this property is not supported, this function returns nonzero.

Remarks

Note that the container is not required to support this property.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::AmbientShowHatching

Other Resources

COleControl Members

COleControl::AmbientShowHatching

Determines whether the container allows the control to display itself with a hatched pattern when UI active.

```
BOOL AmbientShowHatching( );
```

Return Value

Nonzero if the hatched pattern should be shown; otherwise 0. If this property is not supported, this function returns nonzero.

Remarks

Note that the container is not required to support this property.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::AmbientShowGrabHandles

Other Resources

COleControl Members

COleControl::AmbientTextAlign

Determines the ambient text alignment preferred by the control container.

```
short AmbientTextAlign( );
```

Return Value

The status of the container's ambient TextAlign property. If this property is not supported, this function returns 0.

The following is a list of valid return values:

Return value	Meaning
0	General alignment (numbers to the right, text to the left).
1	Left justify
2	Center
3	Right justify

Remarks

This property is available to all embedded controls and is defined by the container. Note that the container is not required to support this property.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::AmbientUIDead

Determines if the container wants the control to respond to user-interface actions.

```
BOOL AmbientUIDead( );
```

Return Value

Nonzero if the control should respond to user-interface actions; otherwise 0. If this property is not supported, this function returns 0.

Remarks

For example, a container might set this to **TRUE** in design mode.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::AmbientUserMode

Other Resources

COleControl Members

COleControl::AmbientUserMode

Determines if the container is in design mode or user mode.

```
BOOL AmbientUserMode( );
```

Return Value

Nonzero if the container is in user mode; otherwise 0 (in design mode). If this property is not supported, this function returns TRUE.

Remarks

For example, a container might set this to **FALSE** in design mode.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::AmbientUIDead

Other Resources

COleControl Members

COleControl::BoundPropertyChanged

Signals that the bound property value has changed.

```
void BoundPropertyChanged(
   DISPID dispid
);
```

Parameters

dispid

The dispatch ID of a bound property of the control.

Remarks

This must be called every time the value of the property changes, even in cases where the change was not made through the property Set method. Be particularly aware of bound properties that are mapped to member variables. Any time such a member variable changes, **BoundPropertyChanged** must be called.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::BoundPropertyRequestEdit

Other Resources

COleControl Members

COleControl::BoundPropertyRequestEdit

Requests permission from the **IPropertyNotifySink** interface to change a bound property value provided by the control.

```
BOOL BoundPropertyRequestEdit(
   DISPID dispid
);
```

Parameters

dispid

The dispatch ID of a bound property of the control.

Return Value

Nonzero if the change is permitted; otherwise 0. The default value is nonzero.

Remarks

If permission is denied, the control must not let the value of the property change. This can be done by ignoring or failing the action that attempted to change the property value.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::BoundPropertyChanged

Other Resources

COleControl Members

COleControl::ClientToParent

Translates the coordinates of *pPoint* into parent coordinates.

```
virtual void ClientToParent(
   LPCRECT lprcBounds,
      LPPOINT pPoint
) const;
```

Parameters

lprcBounds

Pointer to the bounds of the OLE control within the container. Not the client area but the area of the entire control including borders and scroll bars.

pPoint

Pointer to the OLE client area point to be translated into the coordinates of the parent (container).

Remarks

On input *pPoint* is relative to the origin of the client area of the OLE control (upper left corner of the client area of the control). On output *pPoint* is relative to the origin of the parent (upper left corner of the container).

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::ParentToClient
COleControl::GetClientOffset

Other Resources

COleControl Members

COleControl::ClipCaretRect

Adjusts a caret rectangle if it is entirely or partially covered by overlapping, opaque objects.

```
BOOL ClipCaretRect(
   LPRECT lpRect
);
```

Parameters

lpRect

On input, a pointer to a RECT structure that contains the caret area to be adjusted. On output, the adjusted caret area, or **NULL** if the caret rectangle is completely covered.

Return Value

Nonzero if successful; otherwise 0.

Remarks

A caret is a flashing line, block, or bitmap that typically indicates where text or graphics will be inserted.

A windowless object cannot safely show a caret without first checking whether the caret is partially or totally hidden by overlapping objects. In order to make that possible, an object can use **ClipCaretRect** to get the caret adjusted (reduced) to ensure it fits in the clipping region.

Objects creating a caret should submit the caret rectangle to **ClipCaretRect** and use the adjusted rectangle for the caret. If the caret is entirely hidden, this method will return **FALSE** and the caret should not be shown at all in this case.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::COleControl

Constructs a **COleControl** object.

```
COleControl( );
```

Remarks

This function is normally not called directly. Instead the OLE control is usually created by its class factory.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::ControlInfoChanged

Call this function when the set of mnemonics supported by the control has changed.

```
void ControlInfoChanged( );
```

Remarks

Upon receiving this notification, the control's container obtains the new set of mnemonics by making a call to IOleControl::GetControlInfo. Note that the container is not required to respond to this notification.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::DisplayError

Called by the framework after the stock Error event has been handled (unless the event handler has suppressed the display of the error).

```
virtual void DisplayError(
   SCODE scode,
   LPCTSTR lpszDescription,
   LPCTSTR lpszSource,
   LPCTSTR lpszHelpFile,
   UINT nHelpID
);
```

Parameters

scode

The status code value to be reported. For a complete list of possible codes, see the article ActiveX Controls: Advanced Topics.

lpszDescription

The description of the error being reported.

lpszSource

The name of the module generating the error (typically, the name of the OLE control module).

lpszHelpFile

The name of the help file containing a description of the error.

nHelpID

The Help Context ID of the error being reported.

Remarks

The default behavior displays a message box containing the description of the error, contained in *lpszDescription*.

Override this function to customize how errors are displayed.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::FireError

Other Resources

COleControl Members

COleControl::DoClick

Simulates a mouse click action on the control.

```
void DoClick( );
```

Remarks

The overridable **COleControl::OnClick** member function will be called, and a stock Click event will be fired, if supported by the control.

This function is supported by the **COleControl** base class as a stock method, called DoClick. For more information, see the article ActiveX Controls: Methods.

See Also

Reference

COleControl Class

Hierarchy Chart

COleControl::OnClick

Other Resources

COleControl Members

COleControl::DoPropExchange

Called by the framework when loading or storing a control from a persistent storage representation, such as a stream or property set.

```
virtual void DoPropExchange(
   CPropExchange* pPX
);
```

Parameters

pPX

A pointer to a **CPropExchange** object. The framework supplies this object to establish the context of the property exchange, including its direction.

Remarks

This function normally makes calls to the **PX_** family of functions to load or store specific user-defined properties of an OLE control.

If Control Wizard has been used to create the OLE control project, the overridden version of this function will serialize the stock properties supported by **COleControl** with a call to the base class function, **COleControl::DoPropExchange**. As you add user-defined properties to your OLE control you will need to modify this function to serialize your new properties. For more information on serialization, see the article ActiveX Controls: Serializing.

See Also

Reference

COleControl Class
Hierarchy Chart
PX_Bool
PX_Short

Other Resources

COleControl Members

COleControl::DoSuperclassPaint

Redraws an OLE control that has been subclassed from a Windows control.

```
void DoSuperclassPaint(
   CDC* pDC,
   const CRect& rcBounds
);
```

Parameters

pDC

A pointer to the device context of the control container.

rcBounds

The area in which the control is to be drawn.

Remarks

Call this function to properly handle the painting of a nonactive OLE control. This function should only be used if the OLE control subclasses a Windows control and should be called in the `OnDraw` function of your control.

For more information on this function and subclassing a Windows control, see the article ActiveX Controls: Subclassing a Windows Control.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::OnDraw

Other Resources

COleControl Members

COleControl::DrawContent

Called by the framework when the control's appearance needs to be updated.

```
void DrawContent(
   CDC* pDC,
   CRect& rc
);
```

Parameters

pDC

Pointer to the device context.

rc

Rectangular area to be drawn in.

Remarks

This function directly calls the overridable **OnDraw** function.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::OnDraw
COleControl::DrawMetafile
COleControl::OnDrawMetafile

Other Resources

COleControl Members

COleControl::DrawMetafile

Called by the framework when the metafile device context is being used.

```
void DrawMetafile(
   CDC* pDC,
   CRect& rc
);
```

Parameters

pDC

Pointer to the metafile device context.

rc

Rectangular area to be drawn in.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::OnDraw
COleControl::DrawContent
COleControl::OnDrawMetafile

Other Resources

COleControl Members

COleControl::EnableSimpleFrame

Enables the simple frame characteristic for an OLE control.

```
void EnableSimpleFrame( );
```

Remarks

This characteristic allows a control to support visual containment of other controls, but not true OLE containment. An example would be a group box with several controls inside. These controls are not OLE contained, but they are in the same group box.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::ExchangeExtent

Serializes or initializes the state of the control's extent (its dimensions in **HIMETRIC** units).

```
BOOL ExchangeExtent(
   CPropExchange* pPX
);
```

Parameters

pPX

A pointer to a CPropExchange object. The framework supplies this object to establish the context of the property exchange, including its direction.

Return Value

Nonzero if the function succeeded; 0 otherwise.

Remarks

This function is normally called by the default implementation of **COleControl::DoPropExchange**.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::DoPropExchange

Other Resources

COleControl Members

COleControl::ExchangeStockProps

Serializes or initializes the state of the control's stock properties.

```
void ExchangeStockProps(
   CPropExchange* pPX
);
```

Parameters

pPX

A pointer to a CPropExchange object. The framework supplies this object to establish the context of the property exchange, including its direction.

Remarks

This function is normally called by the default implementation of **COleControl::DoPropExchange**.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::DoPropExchange

Other Resources

COleControl Members

COleControl::ExchangeVersion

Serializes or initializes the state of a control's version information.

```
BOOL ExchangeVersion(
   CPropExchange* pPX,
   DWORD dwVersionDefault,
   BOOL bConvert = TRUE
);
```

Parameters

pPX

A pointer to a **CPropExchange** object. The framework supplies this object to establish the context of the property exchange, including its direction.

dwVersionDefault

The current version number of the control.

bConvert

Indicates whether persistent data should be converted to the latest format when saved, or maintained in the same format that was loaded.

Return Value

Nonzero of the function succeeded; 0 otherwise.

Remarks

Typically, this will be the first function called by a control's override of **COleControl::DoPropExchange**. When loading, this function reads the version number of the persistent data, and sets the version attribute of the CPropExchange object accordingly. When saving, this function writes the version number of the persistent data.

For more information on persistence and versioning, see the article ActiveX Controls: Serializing.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::DoPropExchange

Other Resources

COleControl Members

COleControl::FireClick

Called by the framework when the mouse is clicked over an active control.

```
void FireClick( );
```

Remarks

If this event is defined as a custom event, you determine when the event is fired.

For automatic firing of a Click event to occur, the control's Event map must have a stock Click event defined.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::FireDblClick
COleControl::FireMouseDown
COleControl::FireMouseUp

Other Resources

COleControl Members

COleControl::FireDblClick

Called by the framework when the mouse is double-clicked over an active control.

```
void FireDblClick( );
```

Remarks

If this event is defined as a custom event, you determine when the event is fired.

For automatic firing of a DblClick event to occur, the control's Event map must have a stock DblClick event defined.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::FireClick
COleControl::FireMouseDown
COleControl::FireMouseUp

Other Resources

COleControl Members

COleControl::FireError

Fires the stock Error event.

```
void FireError(
   SCODE scode,
   LPCTSTR lpszDescription,
   UINT nHelpID = 0
);
```

Parameters

scode

The status code value to be reported. For a complete list of possible codes, see the article ActiveX Controls: Advanced Topics.

lpszDescription

The description of the error being reported.

nHelpID

The Help ID of the error being reported.

Remarks

This event provides a way of signalling, at appropriate places in your code, that an error has occurred within your control. Unlike other stock events, such as Click or MouseMove, Error is never fired by the framework.

To report an error that occurs during a property get function, property set function, or automation method, call COleControl::ThrowError.

The implementation of an OLE control's Stock Error event uses an **SCODE** value. If your control uses this event, and is intended to be used in Visual Basic 4.0, you will receive errors because the **SCODE** value is not supported in Visual Basic.

To fix this, manually change the **SCODE** parameter in the control's .ODL file to a **long**. In addition, any custom event, method, or property that uses an **SCODE** parameter also causes the same problem.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::DisplayError

Other Resources

COleControl Members

COleControl::FireEvent

Fires a user-defined event from your control with any number of optional arguments,.

```
void AFX_CDECL FireEvent(
   DISPID dispid,
   BYTE* pbParams,
   ...
);
```

Parameters

dispid

The dispatch ID of the event to be fired.

pbParams

A descriptor for the event's parameter types.

Remarks

Usually this function should not be called directly. Instead you will call the event-firing functions in the event map section of your control's class declaration.

The *pbParams* argument is a space-separated list of **VTS_**. One or more of these values, separated by spaces (not commas), specifies the function's parameter list. Possible values are as follows:

Symbol	Parameter type
VTS_COLOR	**OLE_COLOR**
VTS_FONT	**IFontDisp***
VTS_HANDLE	**HWND**
VTS_PICTURE	**IPictureDisp***
VTS_OPTEXCLUSIVE	**OLE_OPTEXCLUSIVE***
VTS_TRISTATE	**OLE_TRISTATE**
VTS_XPOS_HIMETRIC	**OLE_XPOS_HIMETRIC**
VTS_YPOS_HIMETRIC	**OLE_YPOS_HIMETRIC**
VTS_XPOS_PIXELS	**OLE_XPOS_PIXELS**
VTS_YPOS_PIXELS	**OLE_YPOS_PIXELS**
VTS_XSIZE_PIXELS	**OLE_XSIZE_PIXELS**
VTS_YSIZE_PIXELS	**OLE_XSIZE_PIXELS**
VTS_XSIZE_HIMETRIC	**OLE_XSIZE_HIMETRIC**
VTS_YSIZE_HIMETRIC	**OLE_XSIZE_HIMETRIC**

Note

Additional variant constants have been defined for all variant types, with the exception of **VTS_FONT** and **VTS_PICTURE**, tha t provide a pointer to the variant data constant. These constants are named using the **VTS_P***constantname* convention. For e xample, **VTS_PCOLOR** is a pointer to a **VTS_COLOR** constant.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::FireKeyDown

Called by the framework when a key is pressed while the control is UI active.

```
void FireKeyDown(
   USHORT* pnChar,
   short nShiftState
);
```

Parameters

pnChar

Pointer to the virtual key code value of the pressed key. For a list of of standard virtual key codes, see Winuser.h

nShiftState

Contains a combination of the following flags:

- **SHIFT_MASK** The SHIFT key was pressed during the action.
- **CTRL_MASK** The CTRL key was pressed during the action.
- **ALT_MASK** The ALT key was pressed during the action.

Remarks

If this event is defined as a custom event, you determine when the event is fired.

For automatic firing of a KeyDown event to occur, the control's Event map must have a stock KeyDown event defined.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::FireKeyUp
COleControl::FireKeyPress
COleControl::OnKeyPressEvent

Other Resources

COleControl Members

COleControl::FireKeyPress

Called by the framework when a key is pressed and released while the custom control is UI Active within the container.

```
void FireKeyPress(
   USHORT* pnChar
);
```

Parameters

pnChar

A pointer to the character value of the key pressed.

Remarks

If this event is defined as a custom event, you determine when the event is fired.

The recipient of the event may modify *pnChar*, for example, convert all lowercase characters to uppercase. If you want to examine the modified character, override **OnKeyPressEvent**.

For automatic firing of a KeyPress event to occur, the control's Event map must have a stock KeyPress event defined.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::OnKeyPressEvent
COleControl::FireKeyDown
COleControl::FireKeyUp

Other Resources

COleControl Members

COleControl::FireKeyUp

Called by the framework when a key is released while the custom control is UI Active within the container.

```
void FireKeyUp(
   USHORT* pnChar,
   short nShiftState
);
```

Parameters

pnChar

Pointer to the virtual key code value of the released key. For a list of of standard virtual key codes, see Winuser.h

nShiftState

Contains a combination of the following flags:

- **SHIFT_MASK** The SHIFT key was pressed during the action.
- **CTRL_MASK** The CTRL key was pressed during the action.
- **ALT_MASK** The ALT key was pressed during the action.

Remarks

If this event is defined as a custom event, you determine when the event is fired.

For automatic firing of a KeyUp event to occur, the control's Event map must have a stock KeyUp event defined.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::FireKeyDown
COleControl::FireKeyPress
COleControl::OnKeyUpEvent

Other Resources

COleControl Members

COleControl::FireMouseDown

Called by the framework when a mouse button is pressed over an active custom control.

```
void FireMouseDown(
   short nButton,
   short nShiftState,
   OLE_XPOS_PIXELS x,
   OLE_YPOS_PIXELS y
);
```

Parameters

nButton

The numeric value of the mouse button pressed. It can contain one of the following values:

- **LEFT_BUTTON** The left mouse button was pressed down.
- **MIDDLE_BUTTON** The middle mouse button was pressed down.
- **RIGHT_BUTTON** The right mouse button was pressed down.

nShiftState

Contains a combination of the following flags:

- **SHIFT_MASK** The SHIFT key was pressed during the action.
- **CTRL_MASK** The CTRL key was pressed during the action.
- **ALT_MASK** The ALT key was pressed during the action.

x

The x-coordinate of the cursor when a mouse button was pressed down. The coordinate is relative to the upper-left corner of the control window.

y

The y-coordinate of the cursor when a mouse button was pressed down. The coordinate is relative to the upper-left corner of the control window.

Remarks

If this event is defined as a custom event, you determine when the event is fired.

For automatic firing of a MouseDown event to occur, the control's Event map must have a stock MouseDown event defined.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::FireMouseUp
COleControl::FireMouseMove
COleControl::FireClick

Other Resources

COleControl Members

COleControl::FireMouseMove

Called by the framework when the cursor is moved over an active custom control.

```
void FireMouseMove(
   short nButton,
   short nShiftState,
   OLE_XPOS_PIXELS x,
   OLE_YPOS_PIXELS y
);
```

Parameters

nButton

The numeric value of the mouse buttons pressed. Contains a combination of the following values:

- **LEFT_BUTTON** The left mouse button was pressed down during the action.
- **MIDDLE_BUTTON** The middle mouse button was pressed down during the action.
- **RIGHT_BUTTON** The right mouse button was pressed down during the action.

nShiftState

Contains a combination of the following flags:

- **SHIFT_MASK** The SHIFT key was pressed during the action.
- **CTRL_MASK** The CTRL key was pressed during the action.
- **ALT_MASK** The ALT key was pressed during the action.

x

The x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the control window.

y

The y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the control window.

Remarks

If this event is defined as a custom event, you determine when the event is fired.

For automatic firing of a MouseMove event to occur, the control's Event map must have a stock MouseMove event defined.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::FireMouseUp

Called by the framework when a mouse button is released over an active custom control.

```
void FireMouseUp(
   short nButton,
   short nShiftState,
   OLE_XPOS_PIXELS x,
   OLE_YPOS_PIXELS y
);
```

Parameters

nButton

The numeric value of the mouse button released. It can have one of the following values:

- **LEFT_BUTTON** The left mouse button was released.
- **MIDDLE_BUTTON** The middle mouse button was released.
- **RIGHT_BUTTON** The right mouse button was released.

nShiftState

Contains a combination of the following flags:

- **SHIFT_MASK** The SHIFT key was pressed during the action.
- **CTRL_MASK** The CTRL key was pressed during the action.
- **ALT_MASK** The ALT key was pressed during the action.

x

The x-coordinate of the cursor when a mouse button was released. The coordinate is relative to the upper-left corner of the control window.

y

The y-coordinate of a cursor when a mouse button was released. The coordinate is relative to the upper-left corner of the control window.

Remarks

If this event is defined as a custom event, you determine when the event is fired.

For automatic firing of a MouseUp event to occur, the control's Event map must have a stock MouseUp event defined.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::FireMouseDown
COleControl::FireClick
COleControl::FireDblClick

Other Resources

COleControl Members

COleControl::FireReadyStateChange

Fires an event with the current value of the ready state of control.

```
void FireReadyStateChange( );
```

Remarks

The ready state can be one of the following values:

READYSTATE_UNINITIALIZED

Default initialization state

READYSTATE_LOADING

Control is currently loading its properties

READYSTATE_LOADED

Control has been initialized

READYSTATE_INTERACTIVE

Control has enough data to be interactive but not all asynchronous data is yet loaded

READYSTATE_COMPLETE

Control has all its data

Use GetReadyState to determine the control's current readiness.

InternalSetReadyState changes the ready state to the value supplied, then calls **FireReadyStateChange**.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetReadyState
COleControl::InternalSetReadyState

Other Resources

COleControl Members

COleControl::GetActivationPolicy

Alters the default activation behavior of a control that supports the **IPointerInactive** interface.

```
virtual DWORD GetActivationPolicy( );
```

Return Value

A combination of flags from the **POINTERINACTIVE** enumeration. Possible flags are:

POINTERINACTIVE_ACTIVATEONENTRY

The object should be in-place activated when the mouse enters it during a mouse move operation.

POINTERINACTIVE_DEACTIVATEONLEAVE

The object should be deactivated when the mouse leaves the object during a mouse move operation.

POINTERINACTIVE_ACTIVATEONDRAG

The object should be in-place activated when the mouse is dragged over it during a drag and drop operation.

Remarks

When the **IPointerInactive** interface is enabled, the container will delegate **WM_SETCURSOR** and **WM_MOUSEMOVE** messages to it. **COleControl**'s implementation of this interface will dispatch these messages through your control's message map, after adjusting the mouse coordinates appropriately.

Whenever the container receives a **WM_SETCURSOR** or **WM_MOUSEMOVE** message with the mouse pointer over an inactive object supporting **IPointerInactive**, it should call **GetActivationPolicy** on the interface and return flags from the **POINTERINACTIVE** enumeration.

You can process these messages just like ordinary window messages, by adding the corresponding entries to the message map. In your handlers, avoid using the **m_hWnd** member variable (or any member functions that uses it) without first checking that its value is non-**NULL**.

Any object intended to do more than set the mouse cursor and/or fire a mouse move event, such as give special visual feedback, should return the **POINTERINACTIVE_ACTIVATEONENTRY** flag and draw the feedback only when active. If the object returns this flag, the container should activate it in-place immediately and then forward it the same message that triggered the call to **GetActivationPolicy**.

If both the **POINTERINACTIVE_ACTIVATEONENTRY** and **POINTERINACTIVE_DEACTIVATEONLEAVE** flags are returned, then the object will only be activated when the mouse is over the object. If only the **POINTERINACTIVE_ACTIVATEONENTRY** flag is returned, then the object will only be activated once when the mouse first enters the object.

You may also want an inactive control to be the target of an OLE drag and drop operation. This requires activating the control at the moment the user drags an object over it, so that the control's window can be registered as a drop target. To cause activation to occur during a drag, return the **POINTERINACTIVE_ACTIVATEONDRAG** flag:

```
DWORD CMyCtrl::GetActivationPolicy()
{
    return POINTERINACTIVE_ACTIVATEONDRAG;
}
```

The information communicated by **GetActivationPolicy** should not be cached by a container. Instead, this method should be called every time the mouse enters an inactive object.

If an inactive object does not request to be in-place activated when the mouse enters it, its container should dispatch subsequent **WM_SETCURSOR** messages to this object by calling OnInactiveSetCursor as long as the mouse pointer stays over the object.

Enabling the **IPointerInactive** interface typically means that you want the control to be capable of processing mouse messages at all times. To get this behaviour in a container that doesn't support the **IPointerInactive** interface, you will need to have your control always activated when visible, which means the control should have the **OLEMISC_ACTIVATEWHENVISIBLE** flag among its miscellaneous flags. However, to prevent this flag from taking effect in a

container that does support **IPointerInactive**, you can also specify the **OLEMISC_IGNOREACTIVATEWHENVISIBLE** flag:

```
static const DWORD BASED_CODE _dwMyOleMisc =
    OLEMISC_ACTIVATEWHENVISIBLE |
    OLEMISC_IGNOREACTIVATEWHENVISIBLE |
    OLEMISC_SETCLIENTSITEFIRST |
    OLEMISC_INSIDEOUT |
    OLEMISC_CANTLINKINSIDE |
    OLEMISC_RECOMPOSEONRESIZE;
```

See Also

Reference

COleControl Class

Hierarchy Chart

COleControl::OnInactiveSetCursor

COleControl::OnInactiveMouseMove

Other Resources

COleControl Members

COleControl::GetAmbientProperty

Gets the value of an ambient property of the container.

```
BOOL GetAmbientProperty(
   DISPID dispid,
   VARTYPE vtProp,
   void* pvProp
);
```

Parameters

dwDispid

The dispatch ID of the desired ambient property.

vtProp

A variant type tag that specifies the type of the value to be returned in *pvProp*.

pvProp

A pointer to the address of the variable that will receive the property value or return value. The actual type of this pointer must match the type specified by *vtProp*.

vtProp	Type of pvProp
VT_BOOL	**BOOL***
VT_BSTR	**CString***
VT_I2	**short***
VT_I4	**long***
VT_R4	**float***
VT_R8	**double***
VT_CY	**CY***
VT_COLOR	**OLE_COLOR***
VT_DISPATCH	**LPDISPATCH***
VT_FONT	**LPFONTDISP***

Return Value

Nonzero if the ambient property is supported; otherwise 0.

Remarks

If you use **GetAmbientProperty** to retrieve the ambient DisplayName and ScaleUnits properties, set *vtProp* to **VT_BSTR** and *pvProp* to **CString***. If you are retrieving the ambient Font property, set *vtProp* to **VT_FONT** and *pvProp* to **LPFONTDISP***.

Note that functions have already been provided for common ambient properties, such as AmbientBackColor and AmbientFont.

See Also

Reference

COleControl Class

Hierarchy Chart

COleControl::AmbientForeColor
COleControl::AmbientScaleUnits
COleControl::AmbientShowGrabHandles

Other Resources

COleControl Members

COleControl::GetAppearance

Implements the Get function of your control's stock Appearance property.

```
short GetAppearance ( );
```

Return Value

The return value specifies the current appearance setting as a **short** (**VT_I2**) value, if successful. This value is zero if the control's appearance is flat and 1 if the control's appearance is 3D.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SetAppearance
COleControl::OnAppearanceChanged

Other Resources

COleControl Members

COleControl::GetBackColor

Implements the Get function of your control's stock BackColor property.

```
OLE_COLOR GetBackColor( );
```

Return Value

The return value specifies the current background color as a **OLE_COLOR** value, if successful. This value can be translated to a **COLORREF** value with a call to **TranslateColor**.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::AmbientBackColor
COleControl::TranslateColor
COleControl::SetBackColor
COleControl::GetForeColor

Other Resources

COleControl Members

COleControl::GetBorderStyle

Implements the Get function of your control's stock BorderStyle property.

```
short GetBorderStyle( );
```

Return Value

1 if the control has a normal border; 0 if the control has no border.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SetBorderStyle
COleControl::OnBorderStyleChanged

Other Resources

COleControl Members

COleControl::GetCapture

Determines whether the **COleControl** object has the mouse capture.

```
CWnd* GetCapture( );
```

Return Value

If the control is activated and windowless, returns **this** if the control currently has the mouse capture (as determined by the control's container), or **NULL** if it does not have the capture.

Otherwise, returns the **CWnd** object that has the mouse capture (same as **CWnd::GetCapture**).

Remarks

An activated windowless control receives the mouse capture when SetCapture is called.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SetCapture
COleControl::ReleaseCapture

Other Resources

COleControl Members

COleControl::GetClassID

Called by the framework to retrieve the OLE class ID of the control.

```
virtual HRESULT GetClassID(
   LPCLSID pclsid
) = 0;
```

Parameters

pclsid

Pointer to the location of the class ID.

Return Value

Nonzero if the call was not successful; otherwise 0.

Remarks

Usually implemented by the IMPLEMENT_OLECREATE_EX macro.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::GetClientOffset

Retrieves the difference between the upper left corner of the control's rectangular area and the upper left corner of its client area.

```
virtual void GetClientOffset(
   long* pdxOffset,
      long* pdyOffset
) const;
```

Parameters

pdxOffset

Pointer to the horizontal offset of the OLE control's client area.

pdyOffset

Pointer to the vertical offset of the OLE control's client area.

Remarks

The OLE control has a rectangular area within its container. The client area of the control is the control area excluding borders and scroll bars. The offset retrieved by **GetClientOffset** is the difference between the upper left corner of the control's rectangular area and the upper left corner of its client area. If your control has non-client elements other than the standard borders and scrollbars, override this member function to specify the offset.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::ParentToClient
COleControl::ClientToParent

Other Resources

COleControl Members

COleControl::GetClientRect

Retrieves the size of the control's client area.

```
virtual void GetClientRect(
   LPRECT lpRect
) const;
```

Parameters

lpRect

Pointer to a **RECT** structure containing the dimensions of the windowless control's client area; that is, the control's size minus window borders, frames, scroll bars, and so on. The *lpRect* parameter indicates the size of the control's client rectangle, not its position.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::GetClientSite

Queries an object for the pointer to its current client site within its container.

```
LPOLECLIENTSITE GetClientSite( );
```

Return Value

A pointer to the control's current client site in its container.

Remarks

The returned pointer points to an instance of **IOleClientSite**. The **IOleClientSite** interface, implemented by containers, is the object's view of its context: where it is anchored in the document, where it gets its storage, user interface, and other resources.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::GetControlFlags

Retrieves the control flag settings.

```
virtual DWORD GetControlFlags( );
```

Return Value

An ORed combination of the flags in the ControlFlags enumeration:

```
enum ControlFlags {
   fastBeginPaint = 0x0001,
   clipPaintDC = 0x0002,
   pointerInactive = 0x0004,
   noFlickerActivate = 0x0008,
   windowlessActivate = 0x0010,
   canOptimizeDraw = 0x0020,
};
```

Remarks

By default, **GetControlFlags** returns `fastBeginPaint | clipPaintDC`.

`fastBeginPaint`

If set, uses a begin-paint function tailored for OLE controls instead of the BeginPaint API (set by default).

`clipPaintDC`

If not set, disables the call to **IntersectClipRect** made by **COleControl** and gains a small speed advantage. If you are using windowless activation, the flag has no effect.

`pointerInactive`

If set, provides mouse interaction while your control is inactive by enabling **COleControl**'s implementation of the **IPointerInactive** interface, which is disabled by default.

`noFlickerActivate`

If set, eliminates extra drawing operations and the accompanying visual flicker. Use when your control draws itself identically in the inactive and active states. If you are using windowless activation, the flag has no effect.

`windowlessActivate`

If set, indicates your control uses windowless activation.

`canOptimizeDraw`

If set, indicates that the control will perform optimized drawing, if the container supports it.

For more information about **GetControlFlags** and other optimizations of OLE controls, see ActiveX Controls: Optimization.

See Also

Reference

COleControl Class
Hierarchy Chart
CDC::IntersectClipRect
COleControl::SetControlSize

Other Resources

COleControl Members

COleControl::GetControlSize

Retrieves the size of the OLE control window.

```
void GetControlSize(
   int* pcx,
   int* pcy
);
```

Parameters

pcx

Specifies the width of the control in pixels.

pcy

Specifies the height of the control in pixels.

Remarks

Note that all coordinates for control windows are relative to the upper-left corner of the control.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetRectInContainer
COleControl::SetControlSize

Other Resources

COleControl Members

COleControl::GetDC

Provides for a windowless object to get a screen (or compatible) device context from its container.

```
CDC* GetDC(
   LPCRECT lprcRect = NULL,
      DWORD dwFlags = OLEDC_PAINTBKGND
);
```

Parameters

lprcRect

A pointer to the rectangle the windowless control wants to redraw, in client coordinates of the control. **NULL** means the full object's extent.

dwFlags

Drawing attributes of the device context. Choices are:

- **OLEDC_NODRAW** Indicates that the object won't use the device context to perform any drawing but merely to get information about the display device. The container should simply pass the window's DC without further processing.
- **OLEDC_PAINTBKGND** Requests that the container paint the background before returning the DC. An object should use this flag if it is requesting a DC for redrawing an area with transparent background.
- **OLEDC_OFFSCREEN** Informs the container that the object wishes to render into an off-screen bitmap that should then be copied to the screen. An object should use this flag when the drawing operation it is about to perform generates a lot of flicker. The container is free to honor this request or not. However, if this flag is not set, the container must hand back an on-screen DC. This allows objects to perform direct screen operations such as showing a selection (via an **XOR** operation).

Return Value

Pointer to the display device context for the container **CWnd** client area if successful; otherwise, the return value is **NULL**. The display device context can be used in subsequent GDI functions to draw in the client area of the container's window.

Remarks

The ReleaseDC member function must be called to release the context after painting. When calling **GetDC**, objects pass the rectangle they wish to draw into in their own client coordinates. **GetDC** translates these to coordinates of the container client area. The object should not request a desired drawing rectangle larger than its own client area rectangle, the size of which can be retrieved with GetClientRect. This prevents objects from inadvertently drawing where they are not supposed to.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::ReleaseDC

Other Resources

COleControl Members

COleControl::GetEnabled

Implements the Get function of your control's stock Enabled property.

```
BOOL GetEnabled( );
```

Return Value

Nonzero if the control is enabled; otherwise 0.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SetEnabled
COleControl::OnEnabledChanged

Other Resources

COleControl Members

COleControl::GetExtendedControl

Obtains a pointer to an object maintained by the container that represents the control with an extended set of properties.

```
LPDISPATCH GetExtendedControl( );
```

Return Value

A pointer to the container's extended control object. If there is no object available, the value is **NULL**.

This object may be manipulated through its **IDispatch** interface. For more information, see Implementing the IDispatch Interface.You can also use **QueryInterface** to obtain other available interfaces provided by the object. However, the object is not required to support a specific set of interfaces. Note that relying on the specific features of a container's extended control object limits the portability of your control to other arbitrary containers.

Remarks

The function that calls this function is responsible for releasing the pointer when finished with the object. Note that the container is not required to support this object.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::GetFocus

Determines whether the **COleControl** object has the focus.

```
CWnd* GetFocus( );
```

Return Value

If the control is activated and windowless, returns **this** if the control currently has the keyboard focus (as determined by the control's container), or **NULL** if it does not have the focus.

Otherwise, returns the **CWnd** object that has the focus (same as **CWnd::GetFocus**).

Remarks

An activated windowless control receives the focus when SetFocus is called.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SetFocus

Other Resources

COleControl Members

COleControl::GetFont

Implements the Get function of the stock Font property.

```
LPFONTDISP GetFont( );
```

Return Value

A pointer to the font dispatch interface of the control's stock Font property.

Remarks

Note that the caller must release the object when finished. Within the implementation of the control, use **InternalGetFont** to access the control's stock Font object. For more information on using fonts in your control, see the article ActiveX Controls: Using Fonts in an ActiveX Control.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SetFont
COleControl::AmbientFont
COleControl::InternalGetFont

Other Resources

COleControl Members

COleControl::GetFontTextMetrics

Measures the text metrics for any **CFontHolder** object owned by the control.

```
void GetFontTextMetrics(
   LPTEXTMETRIC lptm,
   CFontHolder& fontHolder
);
```

Parameters

lptm

Pointer to a TEXTMETRIC structure.

fontHolder

Reference to a CFontHolder object.

Remarks

Such a font can be selected with the COleControl::SelectFontObject function. **GetFontTextMetrics** will initialize the **TEXTMETRIC** structure pointed to by *lptm* with valid metrics information about *fontHolder*'s font if successful, or fill the structure with zeros if not successful. You should use this function instead of GetTextMetrics when painting your control because controls, like any embedded OLE object, may be required to render themselves into a metafile.

The **TEXTMETRIC** structure for the default font is refreshed when the SelectFontObject function is called. You should call **GetFontTextMetrics** only after selecting the stock Font property to assure the information it provides is valid.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::GetForeColor

Implements the Get function of the stock ForeColor property.

```
OLE_COLOR GetForeColor( );
```

Return Value

The return value specifies the current foreground color as a **OLE_COLOR** value, if successful. This value can be translated to a COLORREF value with a call to **TranslateColor**.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::AmbientForeColor
COleControl::TranslateColor
COleControl::GetBackColor
COleControl::SetForeColor

Other Resources

COleControl Members

COleControl::GetHwnd

Implements the Get function of the stock hWnd property.

```
OLE_HANDLE GetHwnd( );
```

Return Value

The OLE control's window handle, if any; otherwise **NULL**.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::GetMessageString

Called by the framework to obtain a short string that describes the purpose of the menu item identified by *nID*.

```
virtual void GetMessageString(
   UINT nID,
   CString& rMessage
) const;
```

Parameters

nID

A menu item ID.

rMessage

A reference to a CString object through which a string will be returned.

Remarks

This can be used to obtain a message for display in a status bar while the menu item is highlighted. The default implementation attempts to load a string resource identified by *nID*.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::GetNotSupported

Prevents access to a control's property value by the user.

```
void GetNotSupported( );
```

Remarks

Call this function in place of the Get function of any property where retrieval of the property by the control's user is not supported. One example would be a property that is write only.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SetNotSupported

Other Resources

COleControl Members

COleControl::GetReadyState

Returns the readiness state of the control.

```
long GetReadyState( );
```

Return Value

The readiness state of the control, one of the following values:

READYSTATE_UNINITIALIZED

Default initialization state

READYSTATE_LOADING

Control is currently loading its properties

READYSTATE_LOADED

Control has been initialized

READYSTATE_INTERACTIVE

Control has enough data to be interactive but not all asynchronous data is yet loaded

READYSTATE_COMPLETE

Control has all its data

Remarks

Most simple controls never need to differentiate between **LOADED** and **INTERACTIVE**. However, controls that support data path properties may not be ready to be interactive until at least some data is received asynchronously. A control should attempt to become interactive as soon as possible.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::FireReadyStateChange
COleControl::InternalSetReadyState

Other Resources

COleControl Members

COleControl::GetRectInContainer

Obtains the coordinates of the control's rectangle relative to the container, expressed in device units.

```
BOOL GetRectInContainer(
   LPRECT lpRect
);
```

Parameters

lpRect

A pointer to the rectangle structure into which the control's coordinates will be copied.

Return Value

Nonzero if the control is in-place active; otherwise 0.

Remarks

The rectangle is only valid if the control is in-place active.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SetRectInContainer
COleControl::GetControlSize

Other Resources

COleControl Members

COleControl::GetStockTextMetrics

Measures the text metrics for the control's stock Font property, which can be selected with the SelectStockFont function.

```
void GetStockTextMetrics(
   LPTEXTMETRIC lptm
);
```

Parameters

lptm

A pointer to a TEXTMETRIC structure.

Remarks

The **GetStockTextMetrics** function will initialize the **TEXTMETRIC** structure pointed to by *lptm* with valid metrics information if successful, or fill the structure with zeros if not successful. Use this function instead of GetTextMetrics when painting your control because controls, like any embedded OLE object, may be required to render themselves into a metafile.

The **TEXTMETRIC** structure for the default font is refreshed when the **SelectStockFont** function is called. You should call this function only after selecting the stock font to assure the information it provides is valid.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::GetText

Implements the Get function of the stock Text or Caption property.

```
BSTR GetText( );
```

Return Value

The current value of the control text string or a zero-length string if no string is present.

Note
For more information on the **BSTR** data type, see Data Types in the Macros and Globals section.

Remarks

Note that the caller of this function must call SysFreeString on the string returned in order to free the resource. Within the implementation of the control, use **InternalGetText** to access the control's stock Text or Caption property.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::InternalGetText
COleControl::SetText

Other Resources

COleControl Members

COleControl::GetWindowlessDropTarget

Override **GetWindowlessDropTarget** when you want a windowless control to be the target of an OLE drag and drop operation.

```
virtual IDropTarget* GetWindowlessDropTarget( );
```

Return Value

Pointer to the object's **IDropTarget** interface. Since it does not have a window, a windowless object cannot register an **IDropTarget** interface. However, to participate in drag and drop, a windowless object can still implement the interface and return it in **GetWindowlessDropTarget**.

Remarks

Normally, this would require that the control's window be registered as a drop target. But since the control has no window of its own, the container will use its own window as a drop target. The control simply needs to provide an implementation of the **IDropTarget** interface to which the container can delegate calls at the appropriate time. For example:

```
IDropTarget* CMyCtrl::GetWindowlessDropTarget()
{
    m_xDropTarget.AddRef();
    return &m_xDropTarget;
}
```

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::InitializeIIDs

Informs the base class of the IIDs the control will use.

```
void InitializeIIDs(
   const IID* piidPrimary,
      const IID* piidEvents
);
```

Parameters

piidPrimary

Pointer to the interface ID of the control's primary dispatch interface.

piidEvents

Pointer to the interface ID of the control's event interface.

Remarks

Call this function in the control's constructor to inform the base class of the interface IDs your control will be using.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::InternalGetFont

Accesses the stock Font property of your control

```
CFontHolder& InternalGetFont( );
```

Return Value

A reference to a CFontHolder object that contains the stock Font object.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetFont
COleControl::SetFont

Other Resources

COleControl Members

COleControl::InternalGetText

Accesses the stock Text or Caption property of your control.

```
const CString& InternalGetText( );
```

Return Value

A reference to the control text string.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetText
COleControl::SetText

Other Resources

COleControl Members

COleControl::InternalSetReadyState

Sets the readiness state of the control.

```
void InternalSetReadyState(
   long lNewReadyState
);
```

Parameters

lNewReadyState

The readiness state to set for the control, one of the following values:

READYSTATE_UNINITIALIZED

Default initialization state

READYSTATE_LOADING

Control is currently loading its properties

READYSTATE_LOADED

Control has been initialized

READYSTATE_INTERACTIVE

Control has enough data to be interactive but not all asynchronous data is yet loaded

READYSTATE_COMPLETE

Control has all its data

Remarks

Most simple controls never need to differentiate between **LOADED** and **INTERACTIVE**. However, controls that support data path properties may not be ready to be interactive until at least some data is received asynchronously. A control should attempt to become interactive as soon as possible.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::FireReadyStateChange
COleControl::GetReadyState

Other Resources

COleControl Members

COleControl::InvalidateControl

Forces the control to redraw itself.

```
void InvalidateControl(
   LPCRECT lpRect = NULL,
   BOOL bErase = TRUE
);
```

Parameters

lpRect

A pointer to the region of the control to be invalidated.

bErase

Specifies whether the background within the update region is to be erased when the update region is processed.

Remarks

If *lpRect* has a **NULL** value, the entire control will be redrawn. If *lpRect* is not **NULL**, this indicates the portion of the control's rectangle that is to be invalidated. In cases where the control has no window, or is currently not active, the rectangle is ignored, and a call is made to the client site's IAdviseSink::OnViewChange member function. Use this function instead of **CWnd::InvalidateRect** or **InvalidateRect**.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::Refresh

Other Resources

COleControl Members

COleControl::InvalidateRgn

Invalidates the container window's client area within the given region.

```
void InvalidateRgn(
   CRgn* pRgn,
      BOOL bErase = TRUE
);
```

Parameters

pRgn

A pointer to a CRgn object that identifies the display region of the OLE object to invalidate, in client coordinates of the containing window. If this parameter is **NULL**, the extent is the entire object.

bErase

Specifies whether the background within the invalidated region is to be erased. If **TRUE**, the background is erased. If **FALSE**, the background remains unchanged.

Remarks

This can be used to redraw windowless controls within the container. The invalidated region, along with all other areas in the update region, is marked for painting when the next WM_PAINT message is sent.

If *bErase* is **TRUE** for any part of the update region, the background in the entire region, not just in the given part, is erased.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::IsConvertingVBX

Allows specialized loading of an OLE control.

```
BOOL IsConvertingVBX( );
```

Return Value

Nonzero if the control is being converted; otherwise 0.

Remarks

When converting a form that uses VBX controls to one that uses OLE controls, special loading code for the OLE controls may be required. For example, if you are loading an instance of your OLE control, you might have a call to PX_Font in your **DoPropExchange**:

```
PX_Font(pPx, "Font", m_MyFont, pDefaultFont);
```

However, VBX controls did not have a Font object; each font property was saved individually. In this case, you would use **IsConvertingVBX** to distinguish between these two cases:

```
if (IsConvertingVBX()==FALSE)
    PX_Font(pPX, "Font", m_MyFont, pDefaultFont);
else
{
    PX_String(pPX, "FontName", tempString, DefaultName);
    m_MyFont->put_Name(tempString);
    PX_Bool(pPX, "FontUnderline", tempBool, DefaultValue);
    m_MyFont->put_Underline(tempBool);
...
}
```

Another case would be if your VBX control saved proprietary binary data (in its **VBM_SAVEPROPERTY** message handler), and your OLE control saves its binary data in a different format. If you want your OLE control to be backward-compatible with the VBX control, you could read both the old and new formats using the **IsConvertingVBX** function by distinguishing whether the VBX control or the OLE control was being loaded.

In your control's **DoPropExchange** function, you can check for this condition and if true, execute load code specific to this conversion (such as the previous examples). If the control is not being converted, you can execute normal load code. This ability is only applicable to controls being converted from VBX counterparts.

See Also

Reference

COleControl Class

Hierarchy Chart

COleControl::DoPropExchange

Other Resources

COleControl Members

COleControl::IsInvokeAllowed

Enables automation method invocation.

```
BOOL IsInvokeAllowed(
   DISPID dispid
);
```

Return Value

Nonzero if the control has been initialized; otherwise 0.

Remarks

The framework's implementation of **IDispatch::Invoke** calls **IsInvokeAllowed** to determine if a given function (identified by *dispid*) may be invoked. The default behavior for an OLE control is to allow automation methods to be invoked only if the control has been initialized; however, **IsInvokeAllowed** is a virtual function and may be overridden if necessary (for example, when the control is being used as an automation server). For more information, see Knowledge Base article Q166472, "HOWTO: Use an OLE Control as an Automation Server." Knowledge Base articles are available in the MSDN Library Visual Studio documentation or at http://support.microsoft.com.

See Also

Reference

COleControl Class
Hierarchy Chart
CCmdTarget::IsInvokeAllowed

Other Resources

COleControl Members

COleControl::IsModified

Determines if the control's state has been modified.

```
BOOL IsModified( );
```

Return Value

Nonzero if the control's state has been modified since it was last saved; otherwise 0.

Remarks

The state of a control is modified when a property changes value.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SetModifiedFlag

Other Resources

COleControl Members

COleControl::IsOptimizedDraw

Determines whether the container supports optimized drawing for the current drawing operation.

```
BOOL IsOptimizedDraw( );
```

Return Value

TRUE if the container supports optimized drawing for the current drawing operation; otherwise **FALSE**.

Remarks

If optimized drawing is supported, then the control need not select old objects (pens, brushes, fonts, etc.) into the device context when drawing is finished.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::IsSubclassedControl

Called by the framework to determine if the control subclasses a Windows control.

```
virtual BOOL IsSubclassedControl( );
```

Return Value

Nonzero if the control is subclassed; otherwise 0.

Remarks

You must override this function and return **TRUE** if your OLE control subclasses a Windows control.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::Load

Resets any previous data loaded asynchronously and initiates a new loading of the control's asynchronous property.

```
void Load(
   LPCTSTR strNewPath,
      CDataPathProperty& prop
);
```

Parameters

strNewPath

A pointer to a string containing the path that references the absolute location of the asynchronous control property.

prop

A CDataPathProperty object implementing an asynchronous control property.

See Also

Reference

COleControl Class
Hierarchy Chart
CDataPathProperty Class

Other Resources

COleControl Members

COleControl::LockInPlaceActive

Prevents the container from deactivating your control.

```
BOOL LockInPlaceActive(
   BOOL bLock
);
```

Parameters

bLock

TRUE if the in-place active state of the control is to be locked; **FALSE** if it is to be unlocked.

Return Value

Nonzero if the lock was successful; otherwise 0.

Remarks

Note that every locking of the control must be paired with an unlocking of the control when finished. You should only lock your control for short periods, such as while firing an event.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::OnAmbientPropertyChange

Called by the framework when an ambient property of the container has changed value.

```
virtual void OnAmbientPropertyChange(
   DISPID dispid
);
```

Parameters

dispID

The dispatch ID of the ambient property that changed, or **DISPID_UNKNOWN** if multiple properties have changed.

See Also

Reference

COleControl Class

Hierarchy Chart

COleControl::GetAmbientProperty

Other Resources

COleControl Members

COleControl::OnAppearanceChanged

Called by the framework when the stock Appearance property value has changed.

```
virtual void OnAppearanceChanged ( );
```

Remarks

Override this function if you want notification after this property changes. The default implementation calls **InvalidateControl**.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetAppearance
COleControl::SetAppearance
COleControl::InvalidateControl

Other Resources

COleControl Members

COleControl::OnBackColorChanged

Called by the framework when the stock BackColor property value has changed.

```
virtual void OnBackColorChanged( );
```

Remarks

Override this function if you want notification after this property changes. The default implementation calls **InvalidateControl**.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetBackColor
COleControl::InvalidateControl

Other Resources

COleControl Members

COleControl::OnBorderStyleChanged

Called by the framework when the stock BorderStyle property value has changed.

```
virtual void OnBorderStyleChanged( );
```

Remarks

The default implementation calls **InvalidateControl**.

Override this function if you want notification after this property changes.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SetBorderStyle
COleControl::InvalidateControl

Other Resources

COleControl Members

COleControl::OnClick

Called by the framework when a mouse button has been clicked or the DoClick stock method has been invoked.

```
virtual void OnClick(
   USHORT iButton
);
```

Parameters

iButton

Index of a mouse button. Can have one of the following values:

- **LEFT_BUTTON** The left mouse button was clicked.
- **MIDDLE_BUTTON** The middle mouse button was clicked.
- **RIGHT_BUTTON** The right mouse button was clicked.

Remarks

The default implementation calls **COleControl::FireClick**.

Override this member function to modify or extend the default handling.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::DoClick
COleControl::FireClick

Other Resources

COleControl Members

COleControl::OnClose

Called by the framework when the container has called the control's **IOleControl::Close** function.

```
virtual void OnClose(
   DWORD dwSaveOption
);
```

Parameters

dwSaveOption

Flag that indicates whether the object should be saved before loading. Valid values are:

- **OLECLOSE_SAVEIFDIRTY**
- **OLECLOSE_NOSAVE**
- **OLECLOSE_PROMPTSAVE**

Remarks

By default, **OnClose** saves the control object if it has been modified and *dwSaveOption* is either **OLECLOSE_SAVEIFDIRTY** or **OLECLOSE_PROMPTSAVE**.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::OnDoVerb

Called by the framework when the container calls the **IOleObject::DoVerb** member function.

```
virtual BOOL OnDoVerb(
   LONG iVerb,
   LPMSG lpMsg,
   HWND hWndParent,
   LPCRECT lpRect
);
```

Parameters

iVerb

The index of the control verb to be invoked.

lpMsg

A pointer to the Windows message that caused the verb to be invoked.

hWndParent

The handle to the parent window of the control. If the execution of the verb creates a window (or windows), *hWndParent* should be used as the parent.

lpRect

A pointer to a RECT structure into which the coordinates of the control, relative to the container, will be copied.

Return Value

Nonzero if call was successful; otherwise 0.

Remarks

The default implementation uses the **ON_OLEVERB** and **ON_STDOLEVERB** message map entries to determine the proper function to invoke.

Override this function to change the default handling of verb.

See Also

Reference

COleControl Class
Hierarchy Chart
ON_OLEVERB
ON_STDOLEVERB
COleControl::OnEnumVerbs

Other Resources

COleControl Members

COleControl::OnDraw

Called by the framework to draw the OLE control in the specified bounding rectangle using the specified device context.

```
virtual void OnDraw(
   CDC* pDC,
   const CRect& rcBounds,
   const CRect& rcInvalid
);
```

Parameters

pDC

The device context in which the drawing occurs.

rcBounds

The rectangular area of the control, including the border.

rcInvalid

The rectangular area of the control that is invalid.

Remarks

OnDraw is typically called for screen display, passing a screen device context as *pDC*. The *rcBounds* parameter identifies the rectangle in the target device context (relative to its current mapping mode). The *rcInvalid* parameter is the actual rectangle that is invalid. In some cases this will be a smaller area than *rcBounds*.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::OnDrawMetafile
COleControl::DrawContent
COleControl::DrawMetafile

Other Resources

COleControl Members

COleControl::OnDrawMetafile

Called by the framework to draw the OLE control in the specified bounding rectangle using the specified metafile device context.

```
virtual void OnDrawMetafile(
   CDC* pDC,
   const CRect& rcBounds
);
```

Parameters

pDC

The device context in which the drawing occurs.

rcBounds

The rectangular area of the control, including the border.

Remarks

The default implementation calls the OnDraw function.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::OnDraw
COleControl::DrawContent
COleControl::DrawMetafile

Other Resources

COleControl Members

COleControl::OnEdit

Causes the control to be UI activated.

```
virtual BOOL OnEdit(
   LPMSG lpMsg,
   HWND hWndParent,
   LPCRECT lpRect
);
```

Parameters

lpMsg

A pointer to the Windows message that invoked the verb.

hWndParent

A handle to the parent window of the control.

lpRect

A pointer to the rectangle used by the control in the container.

Return Value

Nonzero if the call is successful; otherwise 0.

Remarks

This has the same effect as invoking the control's **OLEIVERB_UIACTIVATE** verb.

This function is typically used as the handler function for an **ON_OLEVERB** message map entry. This makes an "Edit" verb available on the control's "Object" menu. For example:

```
ON_OLEVERB(AFX_IDS_VERB_EDIT, OnEdit)
```

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::OnEnabledChanged

Called by the framework when the stock Enabled property value has changed.

```
virtual void OnEnabledChanged( );
```

Remarks

Override this function if you want notification after this property changes. The default implementation calls InvalidateControl.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SetEnabled
COleControl::GetEnabled

Other Resources

COleControl Members

COleControl::OnEnumVerbs

Called by the framework when the container calls the **IOleObject::EnumVerbs** member function.

```
virtual BOOL OnEnumVerbs(
   LPENUMOLEVERB* ppenumOleVerb
);
```

Parameters

ppenumOleVerb

A pointer to the **IEnumOLEVERB** object that enumerates the control's verbs.

Return Value

Nonzero if verbs are available; otherwise 0.

Remarks

The default implementation enumerates the **ON_OLEVERB** entries in the message map.

Override this function to change the default way of enumerating verbs.

See Also

Reference

COleControl Class
Hierarchy Chart
ON_OLEVERB
ON_STDOLEVERB

Other Resources

COleControl Members

COleControl::OnEventAdvise

Called by the framework when an event handler is connected to or disconnected from an OLE control.

```
virtual void OnEventAdvise(
   BOOL bAdvise
);
```

Parameters

bAdvise

TRUE indicates that an event handler has been connected to the control. **FALSE** indicates that an event handler has been disconnected from the control.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::OnFontChanged

Called by the framework when the stock Font property value has changed.

```
virtual void OnFontChanged( );
```

Remarks

The default implementation calls **COleControl::InvalidateControl**. If the control is subclassing a Windows control, the default implementation also sends a **WM_SETFONT** message to the control's window.

Override this function if you want notification after this property changes.

Example

```
/* SampleActCtrl.cpp */

void CSampleActCtrl::OnFontChanged()
{
   ///////////////////////////////////////////////////////////
   // Always set it to the container's font
   if (m_MyEdit.m_hWnd != NULL)
   {
      IFontDisp* pFontDisp = NULL;
      IFont *pFont = NULL;
      HRESULT hr;

      // Get the container's FontDisp interface
      pFontDisp = AmbientFont();
      if (pFontDisp)
      {
         hr = pFontDisp->QueryInterface(IID_IFont, (LPVOID *) &pFont);
         if (FAILED(hr))
         {
            pFontDisp->Release();
            return;
         }
      }

      HFONT hFont = NULL;
      if (pFont)
      {
         pFont->get_hFont(&hFont);
         m_MyEdit.SendMessage(WM_SETFONT, (WPARAM)hFont, 0L);
      }

      pFontDisp->Release();
   }

   // Invalidate the control
   m_MyEdit.Invalidate();
   m_MyEdit.UpdateWindow();
   ///////////////////////////////////////////////////////////

   COleControl::OnFontChanged();
}
```

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetFont
COleControl::InternalGetFont

COleControl::InvalidateControl

Other Resources

COleControl Members

COleControl::OnForeColorChanged

Called by the framework when the stock ForeColor property value has changed.

```
virtual void OnForeColorChanged( );
```

Remarks

The default implementation calls **InvalidateControl**.

Override this function if you want notification after this property changes.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SetForeColor
COleControl::InvalidateControl

Other Resources

COleControl Members

COleControl::OnFreezeEvents

Called by the framework after the container calls **IOleControl::FreezeEvents**.

```
virtual void OnFreezeEvents(
   BOOL bFreeze
);
```

Parameters

bFreeze

TRUE if the control's event handling is frozen; otherwise **FALSE**.

Remarks

The default implementation does nothing.

Override this function if you want additional behavior when event handling is frozen or unfrozen.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::OnGetColorSet

Called by the framework when the container calls the **IViewObject::GetColorSet** member function.

```
virtual BOOL OnGetColorSet(
   DVTARGETDEVICE* ptd,
   HDC hicTargetDev,
   LPLOGPALETTE* ppColorSet
);
```

Parameters

ptd

Points to the target device for which the picture should be rendered. If this value is **NULL**, the picture should be rendered for a default target device, usually a display device.

hicTargetDev

Specifies the information context on the target device indicated by *ptd*. This parameter can be a device context, but is not one necessarily. If *ptd* is **NULL**, *hicTargetDev* should also be **NULL**.

ppColorSet

A pointer to the location into which the set of colors that would be used should be copied. If the function does not return the color set, **NULL** is returned.

Return Value

Nonzero if a valid color set is returned; otherwise 0.

Remarks

The container calls this function to obtain all the colors needed to draw the OLE control. The container can use the color sets obtained in conjunction with the colors it needs to set the overall color palette. The default implementation returns **FALSE**.

Override this function to do any special processing of this request.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::OnGetControlInfo

Called by the framework when the control's container has requested information about the control.

```
virtual void OnGetControlInfo(
   LPCONTROLINFO pControlInfo
);
```

Parameters

pControlInfo

Pointer to a CONTROLINFO structure to be filled in.

Remarks

This information consists primarily of a description of the control's mnemonic keys. The default implementation fills *pControlInfo* with default information.

Override this function if your control needs to process mnemonic keys.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::OnGetDisplayString

Called by the framework to obtain a string that represents the current value of the property identified by *dispid*.

```
virtual BOOL OnGetDisplayString(
   DISPID dispid,
   CString& strValue
);
```

Parameters

dispid

The dispatch ID of a property of the control.

strValue

A reference to a CString object through which a string will be returned.

Return Value

Nonzero if a string has been returned in *strValue;* otherwise 0.

Remarks

Override this function if your control has a property whose value cannot be directly converted to a string and you want the property's value to be displayed in a container-supplied property browser.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::OnMapPropertyToPage

Other Resources

COleControl Members

COleControl::OnGetInPlaceMenu

Called by the framework when the control is UI activated to obtain the menu to be merged into the container's existing menu.

```
virtual HMENU OnGetInPlaceMenu( );
```

Return Value

The handle of the control's menu, or **NULL** if the control has none. The default implementation returns **NULL**.

Remarks

For more information on merging OLE resources, see the article Menus and Resources (OLE).

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::OnGetNaturalExtent

Called by the framework in response to a container's **IViewObjectEx::GetNaturalExtent** request.

```
virtual BOOL OnGetNaturalExtent(
   DWORD dwAspect,
      LONG lindex,
      DVTARGETDEVICE* ptd,
      HDC hicTargetDev,
      DVEXTENTINFO* pExtentInfo,
      LPSIZEL psizel
);
```

Parameters

dwAspect

Specifies how the object is to be represented. Representations include content, an icon, a thumbnail, or a printed document. Valid values are taken from the enumeration DVASPECT or **DVASPECT2**.

lindex

The portion of the object that is of interest. Currently only -1 is valid.

ptd

Points to the DVTARGETDEVICE structure defining the target device for which the object's size should be returned.

hicTargetDev

Specifies the information context for the target device indicated by the *ptd* parameter from which the object can extract device metrics and test the device's capabilities. If *ptd* is **NULL**, the object should ignore the value in the *hicTargetDev* parameter.

pExtentInfo

Points to the **DVEXTENTINFO** structure that specifies sizing data. The **DVEXTENTINFO** structure is:

```
typedef struct  tagExtentInfo
    {
    UINT cb;
    DWORD dwExtentMode;
    SIZEL sizelProposed;
    }   DVEXTENTINFO;
```

The structure member `dwExtentMode` can take one of two values:

- **DVEXTENT_CONTENT** Inquire how big the control should be to exactly fit content (snap-to-size)
- **DVEXTENT_INTEGRAL** When resizing, pass proposed size to control

psizel

Points to sizing data returned by control. The returned sizing data is set to -1 for any dimension that was not adjusted.

Return Value

Nonzero if it successfully returns or adjusts the size; otherwise 0.

Remarks

Override this function to return the object's display size closest to the proposed size and extent mode in the **DVEXTENTINFO** structure. The default implementation returns **FALSE** and makes no adjustments to the size.

See Also

Reference

COleControl Class

Hierarchy Chart
COleControl::OnGetViewExtent
Other Resources
COleControl Members

COleControl::OnGetPredefinedStrings

Called by the framework to obtain a set of predefined strings representing the possible values for a property.

```
virtual BOOL OnGetPredefinedStrings(
   DISPID dispid,
   CStringArray* pStringArray,
   CDWordArray* pCookieArray
);
```

Parameters

dispid

The dispatch ID of a property of the control.

pStringArray

A string array to be filled in with return values.

pCookieArray

A **DWORD** array to be filled in with return values.

Return Value

Nonzero if elements have been added to *pStringArray* and *pCookieArray*.

Remarks

Override this function if your control has a property with a set of possible values that can be represented by strings. For each element added to *pStringArray*, you should add a corresponding "cookie" element to *pCookieArray*. These "cookie" values may later be passed by the framework to the **COleControl::OnGetPredefinedValue** function.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::OnGetPredefinedValue
COleControl::OnGetDisplayString

Other Resources

COleControl Members

COleControl::OnGetPredefinedValue

Called by the framework to obtain the value corresponding to one of the predefined strings previously returned by an override of **COleControl::OnGetPredefinedStrings**.

```
virtual BOOL OnGetPredefinedValue(
   DISPID dispid,
   DWORD dwCookie,
   VARIANT* lpvarOut
);
```

Parameters

dispid

The dispatch ID of a property of the control.

dwCookie

A cookie value previously returned by an override of **COleControl::OnGetPredefinedStrings**.

lpvarOut

Pointer to a **VARIANT** structure through which a property value will be returned.

Return Value

Nonzero if a value has been returned in *lpvarOut*; otherwise 0.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::OnGetPredefinedStrings
COleControl::OnGetDisplayString

Other Resources

COleControl Members

COleControl::OnGetViewExtent

Called by the framework in response to a container's IViewObject2::GetExtent request.

```
virtual BOOL OnGetViewExtent(
   DWORD dwDrawAspect,
      LONG lindex,
      DVTARGETDEVICE* ptd,
      LPSIZEL lpsizel
);
```

Parameters

dwDrawAspect

DWORD describing which form, or aspect, of an object is to be displayed. Valid values are taken from the enumeration DVASPECT or **DVASPECT2**.

lindex

The portion of the object that is of interest. Currently only –1 is valid.

ptd

Points to the DVTARGETDEVICE structure defining the target device for which the object's size should be returned.

lpsizel

Points to the location where the object's size is returned.

Return Value

Nonzero if extent information is successfully returned; otherwise 0.

Remarks

Override this function if your control uses two-pass drawing, and its opaque and transparent parts have different dimensions.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::OnGetViewRect

Other Resources

COleControl Members

COleControl::OnGetViewRect

Called by the framework in response to a container's **IViewObjectEx::GetRect** request.

```
virtual BOOL OnGetViewRect(
   DWORD dwAspect,
      LPRECTL pRect
);
```

Parameters

dwAspect

DWORD describing which form, or aspect, of an object is to be displayed. Valid values are taken from the enumeration DVASPECT or **DVASPECT2**:

- **DVASPECT_CONTENT** Bounding rectangle of the whole object. Top-left corner at the object's origin and size equal to the extent returned by **GetViewExtent**.
- **DVASPECT_OPAQUE** Objects with a rectangular opaque region return that rectangle. Others fail.
- **DVASPECT_TRANSPARENT** Rectangle covering all transparent or irregular parts.

pRect

Points to the RECTL structure specifying the rectangle in which the object should be drawn. This parameter controls the positioning and stretching of the object.

Return Value

Nonzero if the rectangle sized to the object is successfully returned; otherwise 0.

Remarks

The object's size is converted by **OnGetViewRect** into a rectangle starting at a specific position (the default is the upper left corner of the display). Override this function if your control uses two-pass drawing, and its opaque and transparent parts have different dimensions.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::OnGetViewExtent

Other Resources

COleControl Members

COleControl::OnGetViewStatus

Called by the framework in response to a container's **IViewObjectEx::GetViewStatus** request.

```
virtual DWORD OnGetViewStatus( );
```

Return Value

One of the values of the **VIEWSTATUS** enumeration if successful; otherwise 0. Possible values are any combination of the following:

VIEWSTATUS_OPAQUE

Object is completely opaque. If this bit is not set, the object contains transparent parts. This bit applies only to content-related aspects and not to **DVASPECT_ICON** or **DVASPECT_DOCPRINT**.

VIEWSTATUS_SOLIDBKGND

Object has a solid background (consisting in a solid color, not a brush pattern). This bit is meaningful only if **VIEWSTATUS_OPAQUE** is set and applies only to content-related aspects and not to **DVASPECT_ICON** or **DVASPECT_DOCPRINT**.

VIEWSTATUS_DVASPECTOPAQUE

Object supports **DVASPECT_OPAQUE**. All **IViewObjectEx** methods that take a drawing aspect as a parameter can be called with this aspect.

VIEWSTATUS_DVASPECTTRANSPARENT

Object supports **DVASPECT_TRANSPARENT**. All **IViewObjectEx** methods that take a drawing aspect as a parameter can be called with this aspect.

Remarks

Override this function if your control uses two-pass drawing. The default implementation returns **VIEWSTATUS_OPAQUE**.

See Also

Reference

COleControl Class
Hierarchy Chart
DVASPECT

Other Resources

COleControl Members

COleControl::OnHideToolBars

Called by the framework when the control is UI deactivated.

```
virtual void OnHideToolBars( );
```

Remarks

The implementation should hide all toolbars displayed by **OnShowToolbars**.

See Also

Reference

COleControl Class

Hierarchy Chart

COleControl::OnShowToolBars

Other Resources

COleControl Members

COleControl::OnInactiveMouseMove

Called by the container for the inactive object under the mouse pointer on receipt of a **WM_MOUSEMOVE** message.

```
virtual void OnInactiveMouseMove(
   LPCRECT lprcBounds,
      long x,
      long y,
      DWORD dwKeyState
);
```

Parameters

lprcBounds

The object bounding rectangle, in client coordinates of the containing window. Tells the object its exact position and size on the screen when the **WM_MOUSEMOVE** message was received.

x

The x coordinate of the mouse location in client coordinates of the containing window.

y

The y coordinate of the mouse location in client coordinates of the containing window.

dwKeyState

Identifies the current state of the keyboard modifier keys on the keyboard. Valid values can be a combination of any of the flags **MK_CONTROL**, **MK_SHIFT**, **MK_ALT**, **MK_BUTTON**, **MK_LBUTTON**, **MK_MBUTTON**, and **MK_RBUTTON**.

Remarks

Note that window client coordinates (pixels) are used to pass the mouse cursor position. This is made possible by also passing the bounding rectangle of the object in the same coordinate system.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetActivationPolicy
COleControl::OnInactiveSetCursor

Other Resources

COleControl Members

COleControl::OnInactiveSetCursor

Called by the container for the inactive object under the mouse pointer on receipt of a **WM_SETCURSOR** message.

```
virtual BOOL OnInactiveSetCursor(
   LPCRECT lprcBounds,
      long x,
      long y,
      DWORD dwMouseMsg,
      BOOL bSetAlways
);
```

Parameters

lprcBounds

The object bounding rectangle, in client coordinates of the containing window. Tells the object its exact position and size on the screen when the **WM_SETCURSOR** message was received.

x

The x coordinate of the mouse location in client coordinates of the containing window.

y

The y coordinate of the mouse location in client coordinates of the containing window.

dwMouseMsg

The identifier of the mouse message for which a **WM_SETCURSOR** occurred.

bSetAlways

Specifies whether or not the object must set the cursor. If **TRUE**, the object must set the cursor; if **FALSE**, the cursor is not obligated to set the cursor, and should return **S_FALSE** in that case.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Note that window client coordinates (pixels) are used to pass the mouse cursor position. This is made possible by also passing the bounding rectangle of the object in the same coordinate system.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetActivationPolicy
COleControl::OnInactiveMouseMove

Other Resources

COleControl Members

COleControl::OnKeyDownEvent

Called by the framework after a stock KeyDown event has been processed.

```
virtual void OnKeyDownEvent(
   USHORT nChar,
   USHORT nShiftState
);
```

Parameters

nChar

The virtual key code value of the pressed key. For a list of standard virtual key codes, see Winuser.h

nShiftState

Contains a combination of the following flags:

- **SHIFT_MASK** The SHIFT key was pressed during the action.
- **CTRL_MASK** The CTRL key was pressed during the action.
- **ALT_MASK** The ALT key was pressed during the action.

Remarks

Override this function if your control needs access to the key information after the event has been fired.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::OnKeyUpEvent
COleControl::OnKeyPressEvent

Other Resources

COleControl Members

COleControl::OnKeyPressEvent

Called by the framework after the stock KeyPress event has been fired.

```
virtual void OnKeyPressEvent(
   USHORT nChar
);
```

Parameters

nChar

Contains the virtual key code value of the key pressed. For a list of standard virtual key codes, see Winuser.h

Remarks

Note that the *nChar* value may have been modified by the container.

Override this function if you want notification after this event occurs.

See Also

Reference

COleControl Class

Hierarchy Chart

COleControl::FireKeyPress

Other Resources

COleControl Members

COleControl::OnKeyUpEvent

Called by the framework after a stock KeyDown event has been processed.

```
virtual void OnKeyUpEvent(
   USHORT nChar,
   USHORT nShiftState
);
```

Parameters

nChar

The virtual key code value of the pressed key. For a list of standard virtual key codes, see Winuser.h

nShiftState

Contains a combination of the following flags:

- **SHIFT_MASK** The SHIFT key was pressed during the action.
- **CTRL_MASK** The CTRL key was pressed during the action.
- **ALT_MASK** The ALT key was pressed during the action.

Remarks

Override this function if your control needs access to the key information after the event has been fired.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::OnKeyDownEvent
COleControl::OnKeyPressEvent

Other Resources

COleControl Members

COleControl::OnMapPropertyToPage

Called by the framework to obtain the class ID of a property page that implements editing of the specified property.

```
virtual BOOL OnMapPropertyToPage(
   DISPID dispid,
   LPCLSID lpclsid,
   BOOL* pbPageOptional
);
```

Parameters

dispid

The dispatch ID of a property of the control.

lpclsid

Pointer to a **CLSID** structure through which a class ID will be returned.

pbPageOptional

Returns an indicator of whether use of the specified property page is optional.

Return Value

Nonzero if a class ID has been returned in *lpclsid*; otherwise 0.

Remarks

Override this function to provide a way to invoke your control's property pages from the container's property browser.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::OnGetDisplayString

Other Resources

COleControl Members

COleControl::OnMnemonic

Called by the framework when the container has detected that a mnemonic key of the OLE control has been pressed.

```
virtual void OnMnemonic(
   LPMSG pMsg
);
```

Parameters

pMsg

Pointer to the Windows message generated by a mnemonic key press.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::OnProperties

Called by the framework when the control's properties verb has been invoked by the container.

```
virtual BOOL OnProperties(
   LPMSG lpMsg,
   HWND hWndParent,
   LPCRECT lpRect
);
```

Parameters

lpMsg

A pointer to the Windows message that invoked the verb.

hWndParent

A handle to the parent window of the control.

lpRect

A pointer to the rectangle used by the control in the container.

Return Value

Nonzero if the call is successful; otherwise 0.

Remarks

The default implementation displays a modal property dialog box.

You can also use this function to cause the display of your control's property pages. Make a call to the **OnProperties** function, passing the handle of your control's parent in the *hWndParent* parameter. In this case, the values of the *lpMsg* and *lpRect* parameters are ignored.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::OnQueryHitPoint

Called by the framework in response to a container's **IViewObjectEx::QueryHitPoint** request.

```
virtual BOOL OnQueryHitPoint(
   DWORD dwAspect,
      LPCRECT pRectBounds,
      POINT ptlLoc,
      LONG lCloseHint,
      DWORD* pHitResult
);
```

Parameters

dwAspect

Specifies how the object is represented. Valid values are taken from the enumeration DVASPECT or **DVASPECT2**.

pRectBounds

Pointer to a **RECT** structure specifying the bounding rectangle of the OLE control client area.

ptlLoc

Pointer to the **POINT** structure specifying the point to be checked for a hit. The point is specified in OLE client area coordinates.

lCloseHint

The distance that defines "close" to the point checked for a hit.

pHitResult

Pointer to the result of the hit query. One of the following values:

- **HITRESULT_OUTSIDE** *ptlLoc* is outside the OLE object and not close.
- **HITRESULT_TRANSPARENT** *ptlLoc* is within the bounds of the OLE object, but not close to the image. For example, a point in the middle of a transparent circle could be **HITRESULT_TRANSPARENT**.
- **HITRESULT_CLOSE** *ptlLoc* is inside or outside the OLE object but close enough to the object to be considered inside. Small, thin, or detailed objects may use this value. Even if a point is outside the bounding rectangle of an object it may still be close (this is needed for hitting small objects).
- **HITRESULT_HIT** *ptlLoc* is within the image of the object.

Return Value

Nonzero if a hit result is successfully returned; otherwise 0. A hit is an overlap with the OLE control display area.

Remarks

Queries whether an object's display rectangle overlaps the given point (hits the point). **QueryHitPoint** can be overridden to test hits for non-rectangular objects.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::OnQueryHitRect

Other Resources

COleControl Members

COleControl::OnQueryHitRect

Called by the framework in response to a container's **IViewObjectEx::QueryHitRect** request.

```
virtual BOOL OnQueryHitRect(
   DWORD dwAspect,
   LPCRECT pRectBounds,
   LPCRECT prcLoc,
   LONG lCloseHint,
   DWORD* pHitResult
);
```

Parameters

dwAspect

Specifies how the object is to be represented. Valid values are taken from the enumeration DVASPECT or **DVASPECT2**.

pRectBounds

Pointer to a **RECT** structure specifying the bounding rectangle of the OLE control client area.

prcLoc

Pointer to the **RECT** structure specifying the rectangle to be checked for a hit (overlap with the object rectangle), relative to the upper left corner of the object.

lCloseHint

Not used.

pHitResult

Pointer to the result of the hit query. One of the following values:

- **HITRESULT_OUTSIDE** no point in the rectangle is hit by the OLE object.
- **HITRESULT_HIT** at least one point in the rectangle would be a hit on the object.

Return Value

Nonzero if a hit result is successfully returned; otherwise 0.

Remarks

Queries whether an object's display rectangle overlaps any point in the given rectangle (hits the rectangle). **QueryHitRect** can be overridden to test hits for non-rectangular objects.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::OnQueryHitPoint

Other Resources

COleControl Members

COleControl::OnRenderData

Called by the framework to retrieve data in the specified format.

```
virtual BOOL OnRenderData(
   LPFORMATETC lpFormatEtc,
   LPSTGMEDIUM lpStgMedium
);
```

Parameters

lpFormatEtc

Points to the FORMATETC structure specifying the format in which information is requested.

lpStgMedium

Points to a STGMEDIUM structure in which the data is to be returned.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The specified format is one previously placed in the control object using the DelayRenderData or DelayRenderFileData member functions for delayed rendering. The default implementation of this function calls **OnRenderFileData** or **OnRenderGlobalData**, respectively, if the supplied storage medium is either a file or memory. If the requested format is **CF_METAFILEPICT** or the persistent property set format, the default implementation renders the appropriate data and returns nonzero. Otherwise, it returns 0 and does nothing.

If *lpStgMedium->tymed* is **TYMED_NULL**, the **STGMEDIUM** should be allocated and filled as specified by *lpFormatEtc->tymed*. If not **TYMED_NULL**, the **STGMEDIUM** should be filled in place with the data.

Override this function to provide your data in the requested format and medium. Depending on your data, you may want to override one of the other versions of this function instead. If your data is small and fixed in size, override **OnRenderGlobalData**. If your data is in a file, or is of variable size, override **OnRenderFileData**.

For more information, see the **FORMATETC** and **STGMEDIUM** structures in the *Platform SDK*.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::OnRenderFileData
COleControl::OnRenderGlobalData

Other Resources

COleControl Members

COleControl::OnRenderFileData

Called by the framework to retrieve data in the specified format when the storage medium is a file.

```
virtual BOOL OnRenderFileData(
   LPFORMATETC lpFormatEtc,
   CFile* pFile
);
```

Parameters

lpFormatEtc

Points to the FORMATETC structure specifying the format in which information is requested.

pFile

Points to a CFile object in which the data is to be rendered.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The specified format is one previously placed in the control object using the DelayRenderData member function for delayed rendering. The default implementation of this function simply returns **FALSE**.

Override this function to provide your data in the requested format and medium. Depending on your data, you might want to override one of the other versions of this function instead. If you want to handle multiple storage mediums, override **OnRenderData**. If your data is in a file, or is of variable size, override **OnRenderFileData**.

For more information, see the **FORMATETC** structure in the *Platform SDK*.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::OnRenderData
COleControl::OnRenderGlobalData

Other Resources

COleControl Members

COleControl::OnRenderGlobalData

Called by the framework to retrieve data in the specified format when the specified storage medium is global memory.

```
virtual BOOL OnRenderGlobalData(
   LPFORMATETC lpFormatEtc,
   HGLOBAL* phGlobal
);
```

Parameters

lpFormatEtc

Points to the FORMATETC structure specifying the format in which information is requested.

phGlobal

Points to a handle to global memory in which the data is to be returned. If no memory has been allocated, this parameter can be **NULL**.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The specified format is one previously placed in the control object using the DelayRenderData member function for delayed rendering. The default implementation of this function simply returns **FALSE**.

If *phGlobal* is **NULL**, then a new **HGLOBAL** should be allocated and returned in *phGlobal*. Otherwise, the **HGLOBAL** specified by *phGlobal* should be filled with the data. The amount of data placed in the **HGLOBAL** must not exceed the current size of the memory block. Also, the block cannot be reallocated to a larger size.

Override this function to provide your data in the requested format and medium. Depending on your data, you may want to override one of the other versions of this function instead. If you want to handle multiple storage mediums, override **OnRenderData**. If your data is in a file, or is of variable size, override **OnRenderFileData**.

For more information, see the **FORMATETC** structure in the *Platform SDK*.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::OnRenderFileData
COleControl::OnRenderData

Other Resources

COleControl Members

COleControl::OnResetState

Called by the framework when the control's properties should be set to their default values.

```
virtual void OnResetState( );
```

Remarks

The default implementation calls DoPropExchange, passing a **CPropExchange** object that causes properties to be set to their default values.

The control writer can insert initialization code for the OLE control in this overridable. This function is called when IPersistStream::Load or IPersistStorage::Load fails, or IPersistStreamInit::InitNew or IPersistStorage::InitNew is called, without first calling either **IPersistStream::Load** or **IPersistStorage::Load**.

See Also

Reference

COleControl Class

Hierarchy Chart

COleControl::OnSetClientSite

Other Resources

COleControl Members

COleControl::OnSetClientSite

Called by the framework when the container has called the control's **IOleControl::SetClientSite** function.

```
virtual void OnSetClientSite( );
```

Remarks

By default, **OnSetClientSite** checks whether data path properties are loaded and, if they are, calls `DoDataPathPropExchange`.

Override this function to do any special processing of this notification. In particular, overrides of this function should call the base class.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::OnSetData

Called by the framework to replace the control's data with the specified data.

```
virtual BOOL OnSetData(
      LPFORMATETC lpFormatEtc,
      LPSTGMEDIUM lpStgMedium,
      BOOL bRelease
);
```

Parameters

lpFormatEtc

Pointer to a FORMATETC structure specifying the format of the data.

lpStgMedium

Pointer to a STGMEDIUM structure in which the data resides.

bRelease

TRUE if the control should free the storage medium; **FALSE** if the control should not free the storage medium.

Return Value

Nonzero if successful; otherwise 0.

Remarks

If the data is in the persistent property set format, the default implementation modifies the control's state accordingly. Otherwise, the default implementation does nothing. If *bRelease* is **TRUE**, then a call to **ReleaseStgMedium** is made; otherwise not.

Override this function to replace the control's data with the specified data.

For more information, see the **FORMATETC** and **STGMEDIUM** structures in the *Platform SDK*.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::DoPropExchange

Other Resources

COleControl Members

COleControl::OnSetExtent

Called by the framework when the control's extent needs to be changed, as a result of a call to IOleObject::SetExtent.

```
virtual BOOL OnSetExtent(
   LPSIZEL lpSizeL
);
```

Parameters

lpSizeL

A pointer to the **SIZEL** structure that uses long integers to represent the width and height of the control, expressed in **HIMETRIC** units.

Return Value

Nonzero if the size change was accepted; otherwise 0.

Remarks

The default implementation handles the resizing of the control's extent. If the control is in-place active, a call to the container's **OnPosRectChanged** is then made.

Override this function to alter the default resizing of your control.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::OnSetObjectRects

Called by the framework to implement a call to IOleInPlaceObject::SetObjectRects.

```
virtual BOOL OnSetObjectRects(
   LPCRECT lpRectPos,
   LPCRECT lpRectClip
);
```

Parameters

lpRectPos

A pointer to a RECT structure indicating the control's new position and size relative to the container.

lpRectClip

A pointer to a **RECT** structure indicating a rectangular area to which the control is to be clipped.

Return Value

Nonzero if the repositioning was accepted; otherwise 0.

Remarks

The default implementation automatically handles the repositioning and resizing of the control window and returns **TRUE**.

Override this function to alter the default behavior of this function.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::OnShowToolBars

Called by the framework when the control has been UI activated.

```
virtual void OnShowToolBars( );
```

Remarks

The default implementation does nothing.

See Also

Reference

COleControl Class

Hierarchy Chart

COleControl::OnHideToolBars

Other Resources

COleControl Members

COleControl::OnTextChanged

Called by the framework when the stock Caption or Text property value has changed.

```
virtual void OnTextChanged( );
```

Remarks

The default implementation calls **InvalidateControl**.

Override this function if you want notification after this property changes.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SetText
COleControl::InternalGetText
COleControl::InvalidateControl

Other Resources

COleControl Members

COleControl::OnWindowlessMessage

Called by the framework in response to a container's **IOleInPlaceObjectWindowless::OnWindowMessage** request.

```
virtual BOOL OnWindowlessMessage(
   UINT msg,
      WPARAM wParam,
      LPARAM lParam,
      LRESULT* plResult
);
```

Parameters

msg

Message identifier as passed by Windows.

wParam

As passed by Windows. Specifies additional message-specific information. The contents of this parameter depend on the value of the *msg* parameter.

lParam

As passed by Windows. Specifies additional message-specific information. The contents of this parameter depend on the value of the *msg* parameter.

plResult

Windows result code. Specifies the result of the message processing and depends on the message sent.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Processes window messages for windowless controls. **COleControl**'s **OnWindowlessMessage** should be used for window messages other than mouse messages and keyboard messages. **COleControl** provides SetCapture and SetFocus specifically to get mouse capture and keyboard focus for windowless OLE objects.

Because windowless objects do not have a window, they need a mechanism to let the container dispatch messages to them. A windowless OLE object gets messages from its container, through the **OnWindowMessage** method on the **IOleInPlaceObjectWindowless** interface (an extension of IOleInPlaceObject for windowless support). **OnWindowMessage** does not take an **HWND** parameter.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SetCapture
COleControl::SetFocus
COleControl::GetWindowlessDropTarget

Other Resources

COleControl Members

COleControl::ParentToClient

Translates the coordinates of *pPoint* into client coordinates.

```
virtual UINT ParentToClient(
   LPCRECT lprcBounds,
      LPPOINT pPoint,
      BOOL bHitTest = FALSE
) const;
```

Parameters

lprcBounds

Pointer to the bounds of the OLE control within the container. Not the client area but the area of the entire control including borders and scroll bars.

pPoint

Pointer to the parent (container) point to be translated into the coordinates of the client area of the control.

bHitTest

Specifies whether or not hit testing is to be done on the point.

Return Value

If *bHitTest* is **FALSE**, returns **HTNOWHERE**. If *bHitTest* is **TRUE**, returns the location in which the parent (container) point landed in the client area of the OLE control and is one of the following mouse hit-test values:

- **HTBORDER** In the border of a window that does not have a sizing border.
- **HTBOTTOM** In the lower horizontal border of the window.
- **HTBOTTOMLEFT** In the lower-left corner of the window border.
- **HTBOTTOMRIGHT** In the lower-right corner of the window border.
- **HTCAPTION** In a title-bar area.
- **HTCLIENT** In a client area.
- **HTERROR** On the screen background or on a dividing line between windows (same as **HTNOWHERE** except that the **DefWndProc** Windows function produces a system beep to indicate an error).
- **HTGROWBOX** In a size box.
- **HTHSCROLL** In the horizontal scroll bar.
- **HTLEFT** In the left border of the window.
- **HTMAXBUTTON** In a Maximize button.
- **HTMENU** In a menu area.
- **HTMINBUTTON** In a Minimize button.
- **HTNOWHERE** On the screen background or on a dividing line between windows.
- **HTREDUCE** In a Minimize button.
- **HTRIGHT** In the right border of the window.
- **HTSIZE** In a size box (same as **HTGROWBOX**).
- **HTSYSMENU** In a Control menu or in a Close button in a child window.
- **HTTOP** In the upper horizontal border of the window.
- **HTTOPLEFT** In the upper-left corner of the window border.

- **HTTOPRIGHT** In the upper-right corner of the window border.
- **HTTRANSPARENT** In a window currently covered by another window.
- **HTVSCROLL** In the vertical scroll bar.
- **HTZOOM** In a Maximize button.

Remarks

On input *pPoint* is relative to the origin of the parent (upper left corner of the container). On output *pPoint* is relative to the origin of the client area of the OLE control (upper left corner of the client area of the control).

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::ClientToParent
COleControl::GetClientOffset

Other Resources

COleControl Members

COleControl::PostModalDialog

Notifies the container that a modal dialog box has been closed.

```
void PostModalDialog(
   HWND hWndParent = NULL
);
```

Parameters

hWndParent

Handle to the parent window of the modal dialog box.

Remarks

Call this function after displaying any modal dialog box. You must call this function so that the container can enable any top-level windows disabled by **PreModalDialog**. This function should be paired with a call to **PreModalDialog**.

See Also

Reference

COleControl Class

Hierarchy Chart

COleControl::PreModalDialog

Other Resources

COleControl Members

COleControl::PreModalDialog

Notifies the container that a modal dialog box is about to be displayed.

```
void PreModalDialog(
   HWND hWndParent = NULL
);
```

Parameters

hWndParent

Handle to the parent window of the modal dialog box.

Remarks

Call this function before displaying any modal dialog box. You must call this function so that the container can disable all its top-level windows. After the modal dialog box has been displayed, you must then call **PostModalDialog**.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::PostModalDialog

Other Resources

COleControl Members

COleControl::RecreateControlWindow

Destroys and re-creates the control's window.

```
void RecreateControlWindow( );
```

Remarks

This may be necessary if you need to change the window's style bits.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::Refresh

Forces a repaint of the OLE control.

```
void Refresh( );
```

Remarks

This function is supported by the **COleControl** base class as a stock method, called Refresh. This allows users of your OLE control to repaint the control at a specific time. For more information on this method, see the article ActiveX Controls: Methods.

See Also

Reference

COleControl Class

Hierarchy Chart

COleControl::InvalidateControl

Other Resources

COleControl Members

COleControl::ReleaseCapture

Releases mouse capture.

```
BOOL ReleaseCapture( );
```

Return Value

Nonzero if successful; otherwise 0.

Remarks

If the control currently has the mouse capture, the capture is released. Otherwise, this function has no effect.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SetCapture
COleControl::GetCapture

Other Resources

COleControl Members

COleControl::ReleaseDC

Releases the display device context of a container of a windowless control, freeing the device context for use by other applications.

```
int ReleaseDC(
   CDC* pDC
);
```

Parameters

pDC

Identifies the container device context to be released.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The application must call **ReleaseDC** for each call to GetDC.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetDC

Other Resources

COleControl Members

COleControl::ReparentControlWindow

Sets the parent of the control.

```
virtual void ReparentControlWindow(
   HWND hWndOuter,
   HWND hWndParent
);
```

Parameters

hWndOuter

The handle of the control window.

hWndParent

The handle of the new parent window.

Remarks

Call this function to reset the parent of the control window.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::ResetStockProps

Initializes the state of the **COleControl** stock properties to their default values.

```
void ResetStockProps( );
```

Remarks

The properties are: Appearance, BackColor, BorderStyle, Caption, Enabled, Font, ForeColor, hWnd, and Text. For a description of stock properties, see ActiveX Controls: Adding Stock Properties.

You can improve a control's binary initialization performance by using **ResetStockProps** and **ResetVersion** to override **COleControl::OnResetState**. See the example below. For further information on optimizing initialization, see ActiveX Controls: Optimization.

Example

```
void CMyCtrl::OnResetState()
{
    ResetVersion(MAKELONG(_wVerMinor, _wVerMajor));
    ResetStockProps();

    // initialize custom properties here
}
```

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::ResetVersion
COleControl::SerializeStockProps

Other Resources

COleControl Members

COleControl::ResetVersion

Initializes the version number to specified value.

```
void ResetVersion(
   DWORD dwVersionDefault
);
```

Parameters

dwVersionDefault

The version number to be assigned to the control.

Remarks

You can improve a control's binary initialization performance by using **ResetVersion** and **ResetStockProps** to override **COleControl::OnResetState**. See the example at ResetStockProps. For further information on optimizing initialization, see ActiveX Controls: Optimization.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::ResetStockProps
COleControl::SerializeVersion

Other Resources

COleControl Members

COleControl::ScrollWindow

Allows a windowless OLE object to scroll an area within its in-place active image on the screen.

```
void ScrollWindow(
   int xAmount,
      int yAmount,
      LPCRECT lpRect = NULL,
      LPCRECT lpClipRect = NULL
);
```

Parameters

xAmount

Specifies the amount, in device units, of horizontal scrolling. This parameter must be a negative value to scroll to the left.

yAmount

Specifies the amount, in device units, of vertical scrolling. This parameter must be a negative value to scroll upward.

lpRect

Points to a CRect object or RECT structure that specifies the portion of the OLE object's client area to scroll, in client coordinates of the containing window. If *lpRect* is **NULL**, the entire OLE object's client area is scrolled.

lpClipRect

Points to a **CRect** object or **RECT** structure that specifies the rectangle to clip to. Only pixels inside the rectangle are scrolled. Bits outside the rectangle are not affected even if they are in the *lpRect* rectangle. If *lpClipRect* is **NULL**, no clipping is performed on the scroll rectangle.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::SelectFontObject

Selects a font into a device context.

```
CFont* SelectFontObject(
   CDC* pDC,
   CFontHolder& fontHolder
);
```

Parameters

pDC

Pointer to a device context object.

fontHolder

Reference to the CFontHolder object representing the font to be selected.

Return Value

A pointer to the previously selected font. When the caller has finished all drawing operations that use *fontHolder,* it should reselect the previously selected font by passing it as a parameter to CDC::SelectObject.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::SelectStockFont

Selects the stock Font property into a device context.

```
CFont* SelectStockFont(
   CDC* pDC
);
```

Parameters

pDC

The device context into which the font will be selected.

Return Value

A pointer to the previously selected **CFont** object. You should use CDC::SelectObject to select this font back into the device context when you are finished.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetFont
COleControl::SetFont

Other Resources

COleControl Members

COleControl::SerializeExtent

Serializes or initializes the state of the display space allotted to the control.

```
void SerializeExtent(
   CArchive& ar
);
```

Parameters

ar

A **CArchive** object to serialize to or from.

Remarks

You can improve a control's binary persistence performance by using **SerializeExtent**, **SerializeStockProps**, and **SerializeVersion** to override **COleControl::Serialize**. See the example below. For further information on optimizing initialization, see ActiveX Controls: Optimization.

Example

```
void CMyCtrl::Serialize(CArchive& ar)
{
    DWORD dwVersion =
        SerializeVersion(ar, MAKELONG(_wVerMinor, _wVerMajor));
    SerializeExtent(ar);
    SerializeStockProps(ar);

    if (ar.IsLoading())
    {
        // load custom properties here
    }
    else
    {
        // save custom properties here
    }
}
```

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SerializeStockProps
COleControl::SerializeVersion

Other Resources

COleControl Members

COleControl::SerializeStockProps

Serializes or initializes the state of the **COleControl** stock properties: Appearance, BackColor, BorderStyle, Caption, Enabled, Font, ForeColor, and Text.

```
void SerializeStockProps(
   CArchive& ar
);
```

Parameters

ar

A **CArchive** object to serialize to or from.

Remarks

For a description of stock properties, see ActiveX Controls: Adding Stock Properties.

You can improve a control's binary persistence performance by using **SerializeStockProps**, **SerializeExtent**, and **SerializeVersion** to override **COleControl::Serialize**. For an example, see the code at SerializeExtent. For further information on optimizing initialization, see ActiveX Controls: Optimization.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SerializeExtent
COleControl::SerializeVersion
COleControl::ResetStockProps

Other Resources

COleControl Members

COleControl::SerializeVersion

Serializes or initializes the state of a control's version information.

```
DWORD SerializeVersion(
   CArchive& ar,
      DWORD dwVersionDefault,
      BOOL bConvert = TRUE
);
```

Parameters

ar

A **CArchive** object to serialize to or from.

dwVersionDefault

The current version number of the control.

bConvert

Indicates whether persistent data should be converted to the latest format when it is saved, or maintained in the same format it had when it was loaded.

Return Value

The version number of the control. If the specified archive is loading, **SerializeVersion** returns the version loaded from that archive. Otherwise, it returns the currently loaded version.

Remarks

You can improve a control's binary persistence performance by using **SerializeVersion**, **SerializeExtent**, and **SerializeStockProps** to override **COleControl::Serialize**. For an example, see the code at SerializeExtent. For further information on optimizing initialization, see ActiveX Controls: Optimization.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SerializeExtent
COleControl::SerializeStockProps
COleControl::ResetVersion

Other Resources

COleControl Members

COleControl::SetAppearance

Sets the stock Appearance property value of your control.

```
void SetAppearance (
   short sAppearance
);
```

Parameters

sAppearance

A **short** (**VT_I2**) value to be used for the appearance of your control. A value of zero sets the control's appearance to flat and a value of 1 sets the control's appearance to 3D.

Remarks

For more about stock properties, see ActiveX Controls: Properties.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetAppearance
COleControl::OnAppearanceChanged

Other Resources

COleControl Members

COleControl::SetBackColor

Sets the stock BackColor property value of your control.

```
void SetBackColor(
   OLE_COLOR dwBackColor
);
```

Parameters

dwBackColor

An **OLE_COLOR** value to be used for background drawing of your control.

Remarks

For more information on using this property and other related properties, see the article ActiveX Controls: Properties.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SetForeColor
COleControl::GetBackColor
COleControl::OnBackColorChanged

Other Resources

COleControl Members

COleControl::SetBorderStyle

Sets the stock BorderStyle property value of your control.

```
void SetBorderStyle(
   short sBorderStyle
);
```

Parameters

sBorderStyle

The new border style for the control; 0 indicates no border and 1 indicates a normal border.

Remarks

The control window will then be re-created and **OnBorderStyleChanged** called.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetBorderStyle
COleControl::OnBorderStyleChanged

Other Resources

COleControl Members

COleControl::SetCapture

Causes the control's container window to take possession of the mouse capture on the control's behalf.

```
CWnd* SetCapture( );
```

Return Value

A pointer to the **CWnd** window object that previously received mouse input.

Remarks

If the control is activated and windowless, this function causes the control's container window to take possession of the mouse capture, on the control's behalf. Otherwise, this function causes the control itself to take possession of the mouse capture (same as **CWnd::SetCapture**).

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetCapture
COleControl::ReleaseCapture

Other Resources

COleControl Members

COleControl::SetControlSize

Sets the size of the OLE control window and notifies the container that the control site is changing.

```
BOOL SetControlSize(
   int cx,
      int cy
);
```

Parameters

cx

Specifies the new width of the control in pixels.

cy

Specifies the new height of the control in pixels.

Return Value

Nonzero if the call was successful; otherwise 0.

Remarks

This function should not be used in your control's constructor.

Note that all coordinates for control windows are relative to the upper-left corner of the control.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetControlSize
COleControl::GetRectInContainer

Other Resources

COleControl Members

COleControl::SetEnabled

Sets the stock Enabled property value of your control.

```
void SetEnabled(
   BOOL bEnabled
);
```

Parameters

bEnabled

TRUE if the control is to be enabled; otherwise **FALSE**.

Remarks

After setting this property, **OnEnabledChange** is called.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetEnabled
COleControl::OnEnabledChanged

Other Resources

COleControl Members

COleControl::SetFocus

Causes the control's container window to take possession of the input focus on the control's behalf.

```
CWnd* SetFocus( );
```

Return Value

A pointer to the **CWnd** window object that previously had the input focus, or **NULL** if there is no such window.

Remarks

If the control is activated and windowless, this function causes the control's container window to take possession of the input focus, on the control's behalf. The input focus directs keyboard input to the container's window, and the container dispatches all subsequent keyboard messages to the OLE object that calls **SetFocus**. Any window that previously had the input focus loses it.

If the control is not windowless, this function causes the control itself to take possession of the input focus (same as **CWnd::SetFocus**).

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetFocus

Other Resources

COleControl Members

COleControl::SetFont

Sets the stock Font property of your control.

```
void SetFont(
   LPFONTDISP pFontDisp
);
```

Parameters

pFontDisp

A pointer to a Font dispatch interface.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetFont
COleControl::InternalGetText
COleControl::OnFontChanged

Other Resources

COleControl Members

COleControl::SetForeColor

Sets the stock ForeColor property value of your control.

```
void SetForeColor(
   OLE_COLOR dwForeColor
);
```

Parameters

dwForeColor

An **OLE_COLOR** value to be used for foreground drawing of your control.

Remarks

For more information on using this property and other related properties, see the article ActiveX Controls: Properties.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::SetBackColor
COleControl::GetForeColor
COleControl::OnForeColorChanged

Other Resources

COleControl Members

COleControl::SetInitialDataFormats

Called by the framework to initialize the list of data formats supported by the control.

```
virtual void SetInitialDataFormats( );
```

Remarks

The default implementation specifies two formats: **CF_METAFILEPICT** and the persistent property set.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::SetInitialSize

Sets the size of an OLE control when first displayed in a container.

```
void SetInitialSize(
   int cx,
   int cy
);
```

Parameters

cx

The initial width of the OLE control in pixels.

cy

The initial height of the OLE control in pixels.

Remarks

Call this function in your constructor to set the initial size of your control. The initial size is measured in device units, or pixels. It is recommended that this call be made in your control's constructor.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControl::SetModifiedFlag

Changes the modified state of a control.

```
void SetModifiedFlag(
   BOOL bModified = TRUE
);
```

Parameters

bModified

The new value for the control's modified flag. **TRUE** indicates that the control's state has been modified; **FALSE** indicates that the control's state has just been saved.

Remarks

Call this function whenever a change occurs that would affect your control's persistent state. For example, if the value of a persistent property changes, call this function with *bModified* **TRUE**.

See Also

Reference

COleControl Class

Hierarchy Chart

COleControl::IsModified

Other Resources

COleControl Members

COleControl::SetNotPermitted

Indicates that an edit request has failed.

```
void SetNotPermitted( );
```

Remarks

Call this function when **BoundPropertyRequestEdit** fails. This function throws an exception of type **COleDispScodeException** to indicate that the set operation was not permitted.

See Also

Reference

COleControl Class

Hierarchy Chart

COleControl::BoundPropertyRequestEdit

Other Resources

COleControl Members

COleControl::SetNotSupported

Prevents modification to a control's property value by the user.

```
void SetNotSupported( );
```

Remarks

Call this function in place of the Set function of any property where modification of the property value by the control's user is not supported. One example would be a property that is read only.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetNotSupported

Other Resources

COleControl Members

COleControl::SetRectInContainer

Sets the coordinates of the control's rectangle relative to the container, expressed in device units.

```
BOOL SetRectInContainer(
   LPCRECT lpRect
);
```

Parameters

lpRect

A pointer to a rectangle containing the control's new coordinates relative to the container.

Return Value

Nonzero if the call was successful; otherwise 0.

Remarks

If the control is open, it is resized; otherwise the container's **OnPosRectChanged** function is called.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetRectInContainer
COleControl::GetControlSize

Other Resources

COleControl Members

COleControl::SetText

Sets the value of your control's stock Caption or Text property.

```
void SetText(
   LPCTSTR pszText
);
```

Parameters

pszText

A pointer to a character string.

Remarks

Note that the stock Caption and Text properties are both mapped to the same value. This means that any changes made to either property will automatically change both properties. In general, a control should support either the stock Caption or Text property, but not both.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetText
COleControl::InternalGetText
COleControl::OnTextChanged

Other Resources

COleControl Members

COleControl::ThrowError

Signals the occurrence of an error in your control.

```
void ThrowError(
   SCODE sc,
   UINT nDescriptionID,
   UINT nHelpID = -1
);
void ThrowError(
   SCODE sc,
   LPCTSTR pszDescription = NULL,
   UINT nHelpID = 0
);
```

Parameters

sc

The status code value to be reported. For a complete list of possible codes, see the article ActiveX Controls: Advanced Topics.

nDescriptionID

The string resource ID of the exception to be reported.

nHelpID

The help ID of the topic to be reported on.

pszDescription

A string containing an explanation of the exception to be reported.

Remarks

This function should only be called from within a Get or Set function for an OLE property, or the implementation of an OLE automation method. If you need to signal errors that occur at other times, you should fire the stock Error event.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::FireError
COleControl::DisplayError

Other Resources

COleControl Members

COleControl::TransformCoords

Transforms coordinate values between **HIMETRIC** units and the container's native units.

```
void TransformCoords(
   POINTL* lpptlHimetric,
   POINTF* lpptfContainer,
   DWORD flags
);
```

Parameters

lpptlHimetric

Pointer to a **POINTL** structure containing coordinates in **HIMETRIC** units.

lpptfContainer

Pointer to a **POINTF** structure containing coordinates in the container's unit size.

flags

A combination of the following values:

- **XFORMCOORDS_POSITION** A position in the container.
- **XFORMCOORDS_SIZE** A size in the container.
- **XFORMCOORDS_HIMETRICTOCONTAINER** Transform **HIMETRIC** units to the container's units.
- **XFORMCOORDS_CONTAINERTOHIMETRIC** Transform the container's units to **HIMETRIC** units.

Remarks

The first two flags, **XFORMCOORDS_POSITION** and **XFORMCOORDS_SIZE**, indicate whether the coordinates should be treated as a position or a size. The remaining two flags indicate the direction of transformation.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::AmbientScaleUnits

Other Resources

COleControl Members

COleControl::TranslateColor

Converts a color value from the **OLE_COLOR** data type to the COLORREF data type.

```
COLORREF TranslateColor(
   OLE_COLOR clrColor,
   HPALETTE hpal = NULL
);
```

Parameters

clrColor

A **OLE_COLOR** data type. For more information, see the Windows OleTranslateColor function.

hpal

A handle to an optional palette; can be **NULL**.

Return Value

An RGB (red, green, blue) 32-bit color value that defines the solid color closest to the *clrColor* value that the device can represent.

Remarks

This function is useful to translate the stock ForeColor and BackColor properties to **COLORREF** types used by CDC member functions.

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::GetForeColor
COleControl::GetBackColor

Other Resources

COleControl Members

COleControl::WillAmbientsBeValidDuringLoad

Determines whether your control should use the values of ambient properties as default values, when it is subsequently loaded from its persistent state.

```
BOOL WillAmbientsBeValidDuringLoad( );
```

Return Value

Nonzero indicates that ambient properties will be valid; otherwise ambient properties will not be valid.

Remarks

In some containers, your control may not have access to its ambient properties during the initial call to the override of **COleControl::DoPropExchange**. This is the case if the container calls IPersistStreamInit::Load or IPersistStorage::Load prior to calling IOleObject::SetClientSite (that is, if it does not honor the **OLEMISC_SETCLIENTSITEFIRST** status bit).

See Also

Reference

COleControl Class
Hierarchy Chart
COleControl::DoPropExchange
COleControl::GetAmbientProperty

Other Resources

COleControl Members

COleControl::WindowProc

Provides a Windows procedure for a **COleControl** object.

```
virtual LRESULT WindowProc(
   UINT message,
   WPARAM wParam,
   LPARAM lParam
);
```

Parameters

message

Specifies the Windows message to be processed.

wParam

Provides additional information used in processing the message. The parameter value depends on the message.

lParam

Provides additional information used in processing the message. The parameter value depends on the message.

Return Value

The return value of the message dispatched.

Remarks

Call this function to dispatch specific messages through the control's message map.

See Also

Reference

COleControl Class

Hierarchy Chart

Other Resources

COleControl Members

COleControlContainer Class

Acts as a control container for ActiveX controls.

```
class COleControlContainer : public CCmdTarget
```

Remarks

This is done by providing support for one or more ActiveX control sites (implemented by **COleControlSite**). **COleControlContainer** fully implements the IOleInPlaceFrame and IOleContainer interfaces, allowing the contained ActiveX controls to fulfill their qualifications as in-place items.

Commonly, this class is used in conjunction with **COccManager** and **COleControlSite** to implement a custom ActiveX control container, with custom sites for one or more ActiveX controls.

Requirements

Header: afxocc.h

See Also

Reference

CCmdTarget Class
Hierarchy Chart
COleControlSite Class
COccManager Class

Other Resources

COleControlContainer Members

COleControlContainer Members

Base Class Members

CObject Members

CCmdTarget Members

Construction

COleControlContainer	Constructs a **COleControlContainer** object.

Operations

CreateControl	Creates a hosted ActiveX control.

Overridables

AttachControlSite	Creates a control site, hosted by the container.
BroadcastAmbientPropertyChange	Informs all hosted controls that an ambient property has changed.
CheckDlgButton	Modifies the specified button control.
CheckRadioButton	Selects the specified radio button of a group.
CreateOleFont	Creates an OLE font.
FindItem	Returns the custom site of the specified control.
FreezeAllEvents	Determines if the control site is accepting events.
GetAmbientProp	Retrieves the specified ambient property.
GetDlgItemInt	Retrieves the value of the specified dialog control.
GetDlgItemText	Retrieves the caption of the specified dialog control.
GetDlgItem	Retrieves the specified dialog control.
HandleSetFocus	Determines if the container handles **WM_SETFOCUS** messages.
HandleWindowlessMessage	Handles messages sent to a windowless control.
IsDlgButtonChecked	Determines the state of the specified button.
OnPaint	Called to repaint a portion of the container.
OnUIActivate	Called when a control is about to be in-place activated.
OnUIDeactivate	Called when a control is about to be deactivated.
ScrollChildren	Called by the framework when scroll messages are received from a child window.
SendDlgItemMessage	Sends a message to the specified control.
SetDlgItemInt	Sets the value of the specified control.

SetDlgItemText	Sets the text of the specified control.

Data Members

m_crBack	The background color of the container.
m_crFore	The foreground color of the container.
m_listSitesOrWnds	A list of the supported control sites.
m_nWindowlessControls	The number of hosted windowless controls.
m_pOleFont	A pointer to the OLE font of the custom control site.
m_pSiteCapture	Pointer to the capture control site.
m_pSiteFocus	Pointer to the control that currently has input focus.
m_pSiteUIActive	Pointer to the control that is currently in-place activated.
m_pWnd	Pointer to the window implementing the control container.
m_siteMap	The site map.

See Also

Reference

COleControlContainer Class
Hierarchy Chart

COleControlContainer Member Functions

For information about the member functions in **COleControlContainer**, see COleControlContainer Members.

COleControlContainer::AttachControlSite

Called by the framework to create and attach a control site.

```
virtual void AttachControlSite(
   CWnd* pWnd,
   UINT nIDC = 0
);
void AttachControlSite(
   CWnd* pWnd,
   UINT nIDC = 0
);
```

Parameters

pWnd

A pointer to a **CWnd** object.

nIDC

The ID of the control to be attached.

Remarks

Override this function if you want to customize this process.

Note
Use the first form of this function if you are statically linking to the MFC library. Use the second form if you are dynamically linking to the MFC library.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::BroadcastAmbientPropertyChange

Informs all hosted controls that an ambient property has changed.

```
virtual void BroadcastAmbientPropertyChange(
DISPID dispid
);
```

Parameters

dispid

The dispatch ID of the ambient property being changed.

Remarks

This function is called by the framework when an ambient property has changed value. Override this function to customize this behavior.

See Also

Reference

COleControlContainer Class
Hierarchy Chart
COleControl::OnAmbientPropertyChange

Other Resources

COleControlContainer Members

COleControlContainer::CheckDlgButton

Modifies the current state of the button.

```
virtual void CheckDlgButton(
   int nIDButton,
   UINT nCheck
);
```

Parameters

nIDButton

The ID of the button to be modified.

nCheck

Specifies the state of the button. Can be one of the following:

- **BST_CHECKED** Sets the button state to checked.
- **BST_INDETERMINATE** Sets the button state to grayed, indicating an indeterminate state. Use this value only if the button has the **BS_3STATE** or **BS_AUTO3STATE** style.
- **BST_UNCHECKED** Sets the button state to cleared.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::CheckRadioButton

Selects a specified radio button in a group and clears the remaining buttons in the group.

```
virtual void CheckRadioButton(
   int nIDFirstButton,
   int nIDLastButton,
   int nIDCheckButton
);
```

Parameters

nIDFirstButton

Specifies the identifier of the first radio button in the group.

nIDLastButton

Specifies the identifier of the last radio button in the group.

nIDCheckButton

Specifies the identifier of the radio button to be checked.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::COleControlContainer

Constructs a **COleControlContainer** object.

```
explicit COleControlContainer(
   CWnd* pWnd
);
```

Parameters

pWnd

A pointer to the parent window of the control container.

Remarks

Once the object has been successfully created, add a custom control site with a call to **AttachControlSite**.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::CreateControl

Creates an ActiveX control, hosted by the specified **COleControlSite** object.

```
BOOL CreateControl(
   CWnd* pWndCtrl,
   REFCLSID clsid,
   LPCTSTR lpszWindowName,
   DWORD dwStyle,
   const RECT& rect,
   UINT nID,
   CFile* pPersist=NULL,
   BOOL bStorage=FALSE,
   BSTR bstrLicKey=NULL,
   COleControlSite** ppNewSite=NULL
);
BOOL CreateControl(
   CWnd* pWndCtrl,
   REFCLSID clsid,
   LPCTSTR lpszWindowName,
   DWORD dwStyle,
   const POINT* ppt,
   const SIZE* psize,
   UINT nID,
   CFile* pPersist=NULL,
   BOOL bStorage=FALSE,
   BSTR bstrLicKey=NULL,
   COleControlSite** ppNewSite=NULL
);
```

Parameters

pWndCtrl

A pointer to the window object representing the control.

clsid

The unique class ID of the control.

lpszWindowName

A pointer to the text to be displayed in the control. Sets the value of the control's Caption or Text property (if any). If **NULL**, the control's Caption or Text property is not changed.

dwStyle

Windows styles. The available styles are listed under the **Remarks** section.

rect

Specifies the control's size and position. It can be either a **CRect** object or a **RECT** structure.

nID

Specifies the control's child window ID.

pPersist

A pointer to a **CFile** containing the persistent state for the control. The default value is **NULL**, indicating that the control initializes itself without restoring its state from any persistent storage. If not **NULL**, it should be a pointer to a **CFile**-derived object that contains the control's persistent data, in the form of either a stream or a storage. This data could have been saved in a previous activation of the client. The **CFile** can contain other data, but must have its read-write pointer set to the first byte of persistent data at the time of the call to **CreateControl**.

bStorage

Indicates whether the data in *pPersist* should be interpreted as **IStorage** or **IStream** data. If the data in *pPersist* is a storage,

bStorage should be **TRUE**. If the data in *pPersist* is a stream, *bStorage* should be **FALSE**. The default value is **FALSE**.

bstrLicKey

Optional license key data. This data is needed only for creating controls that require a run-time license key. If the control supports licensing, you must provide a license key for the creation of the control to succeed. The default value is **NULL**.

ppNewSite

A pointer to the existing control site that will host the control being created. The default value is **NULL**, indicating that a new control site will be automatically created and attached to the new control.

ppt

A pointer to a **POINT** structure that contains the upper-left corner of the control. The size of the control is determined by the value of *psize*. The *ppt* and *psize* values are an optional method of specifying the size and position of the control.

psize

A pointer to a **SIZE** structure that contains the size of the control. The upper-left corner is determined by the value of *ppt*. The *ppt* and *psize* values are an optional method of specifying the size and position of the control.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Only a subset of the Windows *dwStyle* flags are supported by **CreateControl**:

- **WS_VISIBLE** Creates a window that is initially visible. Required if you want the control to be visible immediately, like ordinary windows.
- **WS_DISABLED** Creates a window that is initially disabled. A disabled window cannot receive input from the user. Can be set if the control has an Enabled property.
- **WS_BORDER** Creates a window with a thin-line border. Can be set if control has a BorderStyle property.
- **WS_GROUP** Specifies the first control of a group of controls. The user can change the keyboard focus from one control in the group to the next by using the direction keys. All controls defined with the **WS_GROUP** style after the first control belong to the same group. The next control with the **WS_GROUP** style ends the group and starts the next group.
- **WS_TABSTOP** Specifies a control that can receive the keyboard focus when the user presses the TAB key. Pressing the TAB key changes the keyboard focus to the next control of the **WS_TABSTOP** style.

Use the second overload to create default-sized controls.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::CreateOleFont

Creates an OLE font.

```
void CreateOleFont(
   CFont* pFont
);
```

Parameters

pFont

A pointer to the font to be used by the control container.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::FindItem

Finds the custom site that hosts the specified item.

```
virtual COleControlSite* FindItem(
   UINT nID
) const;
```

Parameters

nID

The identifier of the item to be found.

Return Value

A pointer to the custom site of the specified item.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::FreezeAllEvents

Determines if the container will ignore events from the attached control sites or accept them.

```
void FreezeAllEvents(
   BOOL bFreeze
);
```

Parameters

bFreeze

Nonzero if events will be processed; otherwise 0.

Remarks

Note
The control is not required to stop firing events if requested by the control container. It can continue firing but all subsequent events will be ignored by the control container.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::GetAmbientProp

Retrieves the value of a specified ambient property.

```
virtual BOOL GetAmbientProp(
   COleControlSite* pSite,
   DISPID dispid,
   VARIANT* pvarResult
);
```

Parameters

pSite

A pointer to a control site from which the ambient property will be retrieved.

dispid

The dispatch ID of the desired ambient property.

pVarResult

A pointer to the value of the ambient property.

Return Value

Nonzero if successful; otherwise 0.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::GetDlgItem

Retrieves a pointer to the specified control or child window in a dialog box or other window.

```
virtual CWnd* GetDlgItem(
   int nID
) const;
virtual void GetDlgItem(
   int nID,
   HWND* phWnd
) const;
```

Parameters

nID

Identifier of the dialog item to retrieve.

phWnd

A pointer to the handle of the specified dialog item's window object.

Return Value

A pointer to the dialog item's window.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::GetDlgItemInt

Retrieves the value of the translated text of the given control.

```
virtual UINT GetDlgItemInt(
   int nID,
   BOOL* lpTrans,
   BOOL bSigned
) const;
```

Parameters

nID

The identifier of the control.

lpTrans

Pointer to a Boolean variable that receives a function success/failure value (**TRUE** indicates success, **FALSE** indicates failure).

bSigned

Specifies whether the function should examine the text for a minus sign at the beginning and return a signed integer value if it finds one. If the *bSigned* parameter is **TRUE**, specifying that the value to be retrieved is a signed integer value, cast the return value to an **int** type. To get extended error information, call GetLastError.

Return Value

If successful, the variable pointed to by *lpTrans* is set to **TRUE**, and the return value is the translated value of the control text.

If the function fails, the variable pointed to by *lpTrans* is set to **FALSE**, and the return value is zero. Note that, since zero is a possible translated value, a return value of zero does not by itself indicate failure.

If *lpTrans* is **NULL**, the function returns no information about success or failure.

Remarks

The function translates the retrieved text by stripping any extra spaces at the beginning of the text and then converting the decimal digits. The function stops translating when it reaches the end of the text or encounters a nonnumeric character.

This function returns zero if the translated value is greater than **INT_MAX** (for signed numbers) or **UINT_MAX** (for unsigned numbers).

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::GetDlgItemText

Retrieves the text of the given control.

```
virtual int GetDlgItemText(
   int nID,
   LPTSTR lpStr,
   int nMaxCount
) const;
```

Parameters

nID

The identifier of the control.

lpStr

Pointer to the text of the control.

nMaxCount

Specifies the maximum length, in characters, of the string to be copied to the buffer pointed to by *lpStr*. If the length of the string exceeds the limit, the string is truncated.

Return Value

If the function succeeds, the return value specifies the number of characters copied to the buffer, not including the terminating null character.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::HandleSetFocus

Determines if the container handles **WM_SETFOCUS** messages.

```
virtual BOOL HandleSetFocus( );
```

Return Value

Nonzero if the container handles **WM_SETFOCUS** messages; otherwise zero.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::HandleWindowlessMessage

Processes window messages for windowless controls.

```
virtual BOOL HandleWindowlessMessage(
   UINT message,
   WPARAM wParam,
   LPARAM lParam,
   LRESULT* plResult
);
```

Parameters

message

The identifier for the window message, provided by Windows.

wParam

Parameter of the message; provided by Windows. Specifies additional message-specific information. The contents of this parameter depend on the value of the *message* parameter.

lParam

Parameter of the message; provided by Windows. Specifies additional message-specific information. The contents of this parameter depend on the value of the *message* parameter.

plResult

Windows result code. Specifies the result of the message processing and depends on the message sent.

Return Value

Nonzero if successful; otherwise zero.

Remarks

Override this function to customize the handling of windowless control messages.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::IsDlgButtonChecked

Determines the state of the specified button.

```
virtual UINT IsDlgButtonChecked(
   int nIDButton
) const;
```

Parameters

nIDButton

The identifier of the button control.

Return Value

The return value, from a button created with the **BS_AUTOCHECKBOX**, **BS_AUTORADIOBUTTON**, **BS_AUTO3STATE**, **BS_CHECKBOX**, **BS_RADIOBUTTON**, or **BS_3STATE** style. Can be one of the following:

- **BST_CHECKED** Button is checked.
- **BST_INDETERMINATE** Button is grayed, indicating an indeterminate state (applies only if the button has the **BS_3STATE** or **BS_AUTO3STATE** style).
- **BST_UNCHECKED** Button is cleared.

Remarks

If the button is a three-state control, the member function determines whether it is dimmed, checked, or neither.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::OnPaint

Called by the framework to handle **WM_PAINT** requests.

```
virtual BOOL OnPaint(
   CDC* pDC
);
```

Parameters

pDC

A pointer to the device context used by the container.

Return Value

Nonzero if the message was handled; otherwise zero.

Remarks

Override this function to customize the painting process.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::OnUIActivate

Called by the framework when the control site, pointed to by *pSite*, is about to be activated in-place.

```
virtual void OnUIActivate(
   COleControlSite* pSite
);
```

Parameters

pSite

A pointer to the control site about to be in-place activated.

Remarks

In-place activation means that the container's main menu is replaced with an in-place composite menu.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::OnUIDeactivate

Called by the framework when the control site, pointed to by *pSite*, is about to be deactivated.

```
virtual void OnUIDeactivate(
   COleControlSite* pSite
);
```

Parameters

pSite

A pointer to the control site about to be deactivated.

Remarks

When this notification is received, the container should reinstall its user interface and take focus.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::ScrollChildren

Called by the framework when scroll messages are received from a child window.

```
virtual void ScrollChildren(
   int dx,
   int dy
);
```

Parameters

dx

The amount, in pixels, of scrolling along the x-axis.

dy

The amount, in pixels, of scrolling along the y-axis.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::SendDlgItemMessage

Sends a message to the specified control.

```
virtual LRESULT SendDlgItemMessage(
   int nID,
   UINT message,
   WPARAM wParam,
   LPARAM lParam
);
```

Parameters

nID

Specifies the identifier of the control that receives the message.

message

Specifies the message to be sent.

wParam

Specifies additional message-specific information.

lParam

Specifies additional message-specific information.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::SetDlgItemInt

Sets the text of a control in a dialog box to the string representation of a specified integer value.

```
virtual void SetDlgItemInt(
   int nID,
   UINT nValue,
   BOOL bSigned
);
```

Parameters

nID

The identifier of the control.

nValue

The integer value to be displayed.

bSigned

Specifies whether the *nValue* parameter is signed or unsigned. If this parameter is **TRUE**, *nValue* is signed. If this parameter is **TRUE** and *nValue* is less than zero, a minus sign is placed before the first digit in the string. If this parameter is **FALSE**, *nValue* is unsigned.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::SetDlgItemText

Sets the text of the specified control, using the text contained in *lpszString*.

```
virtual void SetDlgItemText(
   int nID,
   LPCTSTR lpszString
);
```

Parameters

nID

The identifier of the control.

lpszString

Pointer to the text of the control.

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer Data Members

For information about the data members in **COleControlContainer**, see COleControlContainer Members.

COleControlContainer::m_crBack

The background color of the container.

```
COLORREF m_crBack;
```

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::m_crFore

The foreground color of the container.

```
COLORREF m_crFore;
```

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::m_listSitesOrWnds

A list of the control sites hosted by the container.

```
CTypedPtrList< CPtrList, COleControlSiteOrWnd* > m_listSitesOrWnds;
```

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::m_nWindowlessControls

The number of windowless controls hosted by the control container.

```
int m_nWindowlessControls;
```

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::m_pOleFont

A pointer to the OLE font of the custom control site.

```
LPFONTDISP m_pOleFont;
```

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::m_pSiteCapture

Pointer to the capture control site.

```
COleControlSite* m_pSiteCapture;
```

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::m_pSiteFocus

A pointer to the control site that currently has input focus.

```
COleControlSite* m_pSiteFocus;
```

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::m_pSiteUIActive

A pointer to the control site that is in-place activated.

```
COleControlSite* m_pSiteUIActive;
```

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::m_pWnd

A pointer to the window object associated with the container.

```
CWnd* m_pWnd;
```

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlContainer::m_siteMap

The site map.

```
CMapPtrToPtr m_siteMap;
```

See Also

Reference

COleControlContainer Class

Hierarchy Chart

Other Resources

COleControlContainer Members

COleControlModule Class

The base class from which you derive an OLE control module object.

```
class COleControlModule : public CWinApp
```

Remarks

This class provides member functions for initializing your control module. Each OLE control module that uses the Microsoft Foundation classes can only contain one object derived from **COleControlModule**. This object is constructed when other C++ global objects are constructed. Declare your derived **COleControlModule** object at the global level.

For more information on using the **COleControlModule** class, see the CWinApp class and the article ActiveX Controls.

Requirements

Header: afxctl.h

See Also

Tasks

TESTHELP Sample: ActiveX Control with Tooltips and Help

Reference

Hierarchy Chart

Other Resources

CWinApp Members

COleControlSite Class

Provides support for custom client-side control interfaces.

```
class COleControlSite : public CCmdTarget
```

Remarks

This support is the primary means by which an embedded ActiveX control obtains information about the location and extent of its display site, its moniker, its user interface, its ambient properties, and other resources provided by its container. **COleControlSite** fully implements the IOleControlSite, IOleInPlaceSite, IOleClientSite, IPropertyNotifySink, **IBoundObjectSite**, **INotifyDBEvents**, IRowSetNotify interfaces. In addition, the IDispatch interface (providing support for ambient properties and event sinks) is also implemented.

To create an ActiveX control site using **COleControlSite**, derive a class from **COleControlSite**. In your **CWnd**-derived class for the container (for instance, your dialog box) override the **CWnd::CreateControlSite** function.

Requirements

Header: afxocc.h

See Also

Reference

CCmdTarget Class
Hierarchy Chart
COleControlContainer Class

Other Resources

COleControlSite Members

COleControlSite Members

Base Class Members

CObject Members

CCmdTarget Members

Construction

COleControlSite	Constructs a **COleControlSite** object.

Operations

CreateControl	Creates a hosted ActiveX control.
DestroyControl	Destroys the hosted control.
DoVerb	Executes a specific verb of the hosted control.
FreezeEvents	Specifies if the control site is accepting events.
GetControlInfo	Retrieves keyboard information and mnemonics for the hosted control.
GetDefBtnCode	Retrieves the default button code for the hosted control.
GetEventIID	Retrieves the ID of an event interface for a hosted control.
IsDefaultButton	Determines if the control is the default button in the window.
SetDefaultButton	Sets the default button in the window.

Overridables

BindDefaultProperty	Binds the default property of the hosted control to a data source.
BindProperty	Binds a property of the hosted control to a data source.
EnableDSC	Enables data sourcing for a control site.
EnableWindow	Enables the control site.
GetDlgCtrlID	Retrieves the identifier of the control.
GetExStyle	Retrieves the extended styles of the control site.
GetProperty	Retrieves a specific property of the hosted control.
GetStyle	Retrieves the styles of the control site.
GetWindowText	Retrieves the text of the hosted control.
InvokeHelper	Invoke a specific method of the hosted control.
InvokeHelperV	Invoke a specific method of the hosted control with a variable list of arguments.
IsWindowEnabled	Checks the visible state of the control site.

ModifyStyleEx	Modifies the current extended styles of the control site.
ModifyStyle	Modifies the current styles of the control site.
MoveWindow	Changes the position of the control site.
QuickActivate	Quick activates the hosted control.
SafeSetProperty	Sets a property or method of the control without chance of throwing an exception.
SetDlgCtrlID	Retrieves the identifier of the control.
SetFocus	Sets the focus to the control site.
SetProperty	Sets a specific property of the hosted control.
SetPropertyV	Sets a specific property of the hosted control with a variable list of arguments.
SetWindowPos	Sets the position of the control site.
SetWindowText	Sets the text of the hosted control.
ShowWindow	Shows or hides the control site.

Attributes

m_bIsWindowless	Determines if the hosted control is a windowless control.
m_ctlInfo	Contains information on keyboard handling for the control.
m_dwEventSink	The cookie of the control's connection point.
m_dwMiscStatus	The miscellaneous states for the hosted control.
m_dwPropNotifySink	The **IPropertyNotifySink** cookie of the control.
m_dwStyle	The styles of the hosted control.
m_hWnd	The handle of the control site.
m_iidEvents	The ID of the event interface for the hosted control.
m_nID	The ID of the hosted control.
m_pActiveObject	A pointer to the **IOleInPlaceActiveObject** object of the hosted control.
m_pCtrlCont	The container of the hosted control.
m_pInPlaceObject	A pointer to the **IOleInPlaceObject** object of the hosted control.
m_pObject	A pointer to the **IOleObjectInterface** interface of the control.
m_pWindowlessObject	A pointer to the **IOleInPlaceObjectWindowless** interface of the control.

m_pWndCtrl	A pointer to the window object for the hosted control.
m_rect	The dimensions of the control site.

See Also

Reference

COleControlSite Class
Hierarchy Chart

COleControlSite Member Functions

For information about the member functions in **COleControlSite**, see COleControlSite Members.

COleControlSite::BindDefaultProperty

Binds the calling object's default simple bound property, as marked in the type library, to the underlying cursor that is defined by the DataSource, UserName, Password, and SQL properties of the data-source control.

```
virtual void BindDefaultProperty(
   DISPID dwDispID,
   VARTYPE vtProp,
   LPCTSTR szFieldName,
   CWnd* pDSCWnd
);
```

Parameters

dwDispID

Specifies the **DISPID** of a property on a data-bound control that is to be bound to a data-source control.

vtProp

Specifies the type of the property to be bound — for example, **VT_BSTR**, **VT_VARIANT**, and so on.

szFieldName

Specifies the name of the column, in the cursor provided by the data-source control, to which the property will be bound.

pDSCWnd

A pointer to the **CWnd**-derived object that hosts the data-source control to which the property will be bound.

Remarks

The **CWnd** object on which you call this function must be a data-bound control.

See Also

Reference

COleControlSite Class
Hierarchy Chart
COleControlSite::BindProperty

Other Resources

COleControlSite Members

COleControlSite::BindProperty

Binds the calling object's simple bound property, as marked in the type library, to the underlying cursor that is defined by the DataSource, UserName, Password, and SQL properties of the data-source control.

```
virtual void BindProperty(
   DISPID dwDispId,
   CWnd* pWndDSC
);
```

Parameters

dwDispId

Specifies the **DISPID** of a property on a data-bound control that is to be bound to a data-source control.

pWndDSC

A pointer to the **CWnd**-derived object that hosts the data-source control to which the property will be bound.

Remarks

The **CWnd** object on which you call this function must be a data-bound control.

See Also

Reference

COleControlSite Class
Hierarchy Chart
COleControlSite::BindDefaultProperty

Other Resources

COleControlSite Members

COleControlSite::COleControlSite

Constructs a new **COleControlSite** object.

```
explicit COleControlSite(
   COleControlContainer* pCtrlCont
);
```

Parameters

pCtrlCont

A pointer to the control's container (which represents the window that hosts the AtiveX control).

Remarks

This function is called by the COccManager::CreateContainer function. For more information on customizing the creation of containers, see COccManager::CreateSite.

See Also

Reference

COleControlSite Class
Hierarchy Chart
COleControlSite::CreateControl

Other Resources

COleControlSite Members

COleControlSite::CreateControl

Creates an ActiveX control, hosted by the **COleControlSite** object.

```
virtual HRESULT CreateControl(
   CWnd* pWndCtrl,
   REFCLSID clsid,
   LPCTSTR lpszWindowName,
   DWORD dwStyle,
   const RECT& rect,
   UINT nID,
   CFile* pPersist = NULL,
   BOOL bStorage = FALSE,
   BSTR bstrLicKey = NULL
);
virtual HRESULT CreateControl(
   CWnd* pWndCtrl,
   REFCLSID clsid,
   LPCTSTR lpszWindowName,
   DWORD dwStyle,
   const POINT* ppt,
   const SIZE* psize,
   UINT nID,
   CFile* pPersist = NULL,
   BOOL bStorage = FALSE,
   BSTR bstrLicKey = NULL
);
```

Parameters

pWndCtrl

A pointer to the window object representing the control.

clsid

The unique class ID of the control.

lpszWindowName

A pointer to the text to be displayed in the control. Sets the value of the winodw's Caption or Text property (if any).

dwStyle

Windows styles. The available styles are listed under the **Remarks** section.

rect

Specifies the control's size and position. It can be either a **CRect** object or a **RECT** structure.

nID

Specifies the control's child window ID.

pPersist

A pointer to a **CFile** containing the persistent state for the control. The default value is **NULL**, indicating that the control initializes itself without restoring its state from any persistent storage. If not **NULL**, it should be a pointer to a **CFile**-derived object that contains the control's persistent data, in the form of either a stream or a storage. This data could have been saved in a previous activation of the client. The **CFile** can contain other data, but must have its read-write pointer set to the first byte of persistent data at the time of the call to **CreateControl**.

bStorage

Indicates whether the data in *pPersist* should be interpreted as **IStorage** or **IStream** data. If the data in *pPersist* is a storage, *bStorage* should be **TRUE**. If the data in *pPersist* is a stream, *bStorage* should be **FALSE**. The default value is **FALSE**.

bstrLicKey

Optional license key data. This data is needed only for creating controls that require a run-time license key. If the control supports licensing, you must provide a license key for the creation of the control to succeed. The default value is **NULL**.

ppt

A pointer to a **POINT** structure that contains the upper-left corner of the control. The size of the control is determined by the value of *psize*. The *ppt* and *psize* values are an optional method of specifying the size and position opf the control.

psize

A pointer to a **SIZE** structure that contains the size of the control. The upper-left corner is determined by the value of *ppt*. The *ppt* and *psize* values are an optional method of specifying the size and position opf the control.

Return Value

A standard **HRESULT** value.

Remarks

Only a subset of the Windows *dwStyle* flags are supported by **CreateControl**:

- **WS_VISIBLE** Creates a window that is initially visible. Required if you want the control to be visible immediately, like ordinary windows.
- **WS_DISABLED** Creates a window that is initially disabled. A disabled window cannot receive input from the user. Can be set if the control has an Enabled property.
- **WS_BORDER** Creates a window with a thin-line border. Can be set if control has a BorderStyle property.
- **WS_GROUP** Specifies the first control of a group of controls. The user can change the keyboard focus from one control in the group to the next by using the direction keys. All controls defined with the **WS_GROUP** style after the first control belong to the same group. The next control with the **WS_GROUP** style ends the group and starts the next group.
- **WS_TABSTOP** Specifies a control that can receive the keyboard focus when the user presses the TAB key. Pressing the TAB key changes the keyboard focus to the next control of the **WS_TABSTOP** style.

Use the second overload to create default-sized controls.

See Also

Reference

COleControlSite Class
Hierarchy Chart
COleControlContainer::CreateControl

Other Resources

COleControlSite Members

COleControlSite::DestroyControl

Destroys the **COleControlSite** object.

```
virtual BOOL DestroyControl( );
```

Return Value

Nonzero if successful, otherwise 0.

Remarks

Once completed, the object is freed from memory and any pointers to the object are no longer valid.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::DoVerb

Executes the specified verb.

```
virtual HRESULT DoVerb(
   LONG nVerb,
   LPMSG lpMsg = NULL
);
```

Parameters

nVerb

Specifies the verb to execute. It can include one of the following:

Value	Meaning	Symbol
0	Primary verb	**OLEIVERB_PRIMARY**
-1	Secondary verb	(None)
1	Displays the object for editing.	**OLEIVERB_SHOW**
-2	Edits the item in a separate window.	**OLEIVERB_OPEN**
-3	Hides the object.	**OLEIVERB_HIDE**
-4	Activates a control in-place.	**OLEIVERB_UIACTIVATE**
-5	Activates a control in-place, without additional user interface elements.	**OLEIVERB_INPLACEACTIVATE**
-7	Display the control's properties.	**OLEIVERB_PROPERTIES**

lpMsg

Pointer to the message that caused the item to be activated.

Return Value

A standard **HRESULT** value.

Remarks

This function directly calls through the control's **IOleObject** interface to execute the specified verb. If an exception is thrown as a result of this function call, an **HRESULT** error code is returned.

For more information, see IOleObject::DoVerb in the Platform SDK.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::EnableDSC

Enables data sourcing for the control site.

```
virtual void EnableDSC( );
```

Remarks

Called by the framework to enable and initialize data sourcing for the control site. Override this function to provide customized behavior.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::EnableWindow

Enables or disables mouse and keyboard input to the control site.

```
virtual BOOL EnableWindow(
   BOOL bEnable
);
```

Parameters

bEnable

Specifies whether to enable or disable the window: **TRUE** if window input is to be enabled, otherwise **FALSE**.

Return Value

Nonzero if the window was previously disabled, otherwise 0.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::FreezeEvents

Specifies whether the control site will handle or ignore events fired from a control.

```
void FreezeEvents(
   BOOL bFreeze
);
```

Parameters

bFreeze

Specifies whether the control site wishes to stop accepting events. Nonzero if the control is not accepting events; otherwise zero.

Remarks

If *bFreeze* is **TRUE**, the control site requests the control to stop fring events. If *bFreeze* is **FALSE**, the control site requests the control to continue firing events.

Note
The control is not required to stop firing events if requested by the control site. It can continue firing but all subsequent event s will be ignored by the control site.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::GetControlInfo

Retrieves information about a control's keyboard mnemonics and keyboard behavior.

```
void GetControlInfo( );
```

Remarks

The information is stored in COleControlSite::m_ctlInfo.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::GetDefBtnCode

Determines if the control is a default push button.

```
DWORD GetDefBtnCode( );
```

Return Value

Can be one of the following values:

- **DLGC_DEFPUSHBUTTON** Control is the default button in the dialog.
- **DLGC_UNDEFPUSHBUTTON** Control is not the default button in the dialog.
- **0** Control is not a button.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::GetDlgCtrlID

Retrieves the identifier of the control.

```
virtual int GetDlgCtrlID( ) const;
```

Return Value

The dialog item identifier of the control.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::GetEventIID

Retrieves a pointer to the control's default event interface.

```
BOOL GetEventIID(
   IID* piid
);
```

Parameters

piid

A pointer to an interface ID.

Return Value

Nonzero if successful, otherwise 0. If successful, *piid* contains the interface ID for the control's default event interface.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::GetExStyle

Retrieves the window's extended styles.

```
virtual DWORD GetExStyle( ) const;
```

Return Value

The control window's extended styles.

Remarks

To retrieve the regular styles, call COleControlSite::GetStyle.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::GetProperty

Gets the control property specified by *dwDispID*.

```
virtual void GetProperty(
   DISPID dwDispID,
   VARTYPE vtProp,
   void* pvProp
) const;
```

Parameters

dwDispID

Identifies the dispatch ID of the property, found on the control's default **IDispatch** interface, to be retrieved.

vtProp

Specifies the type of the property to be retrieved. For possible values, see the Remarks section for COleDispatchDriver::InvokeHelper.

pvProp

Address of the variable that will receive the property value. It must match the type specified by *vtProp*.

Remarks

The value is returned through *pvProp*.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::GetStyle

Retrieves the styles of the control site.

```
virtual DWORD GetStyle( ) const;
```

Return Value

The window's styles.

Remarks

For a list of possible values, see Windows Styles. To retrieve the extended styles of the control site, call COleControlSite::GetExStyle.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::GetWindowText

Retrieves the current text of the control.

```
virtual void GetWindowText(
   CString& str
) const;
```

Parameters

str

A reference to a **CString** object that contains the current text of the control.

Remarks

If the control supports the Caption stock property, this value is returned. If the Caption stock property is not supported, the value for the Text property is returned.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::InvokeHelper

Invokes the method or property specified by *dwDispID*, in the context specified by *wFlags*.

```
virtual void AFX_CDECL InvokeHelper(
   DISPID dwDispID,
   WORD wFlags,
   VARTYPE vtRet,
   void* pvRet,
      const BYTE* pbParamInfo,
      ...
);
```

Parameters

dwDispID

Identifies the dispatch ID of the property or method, found on the control's **IDispatch** interface, to be invoked.

wFlags

Flags describing the context of the call to IDispatch::Invoke. For possible *wFlags* values, see IDispatch::Invoke in the *Platform SDK*.

vtRet

Specifies the type of the return value. For possible values, see the Remarks section for COleDispatchDriver::InvokeHelper.

pvRet

Address of the variable that will receive the property value or return value. It must match the type specified by *vtRet*.

pbParamInfo

Pointer to a null-terminated string of bytes specifying the types of the parameters following *pbParamInfo*. For possible values, see the Remarks section for COleDispatchDriver::InvokeHelper.

...

Variable list of parameters, of types specified in *pbParamInfo*.

Remarks

The *pbParamInfo* parameter specifies the types of the parameters passed to the method or property. The variable list of arguments is represented by ... in the syntax declaration.

This function converts the parameters to **VARIANTARG** values, then invokes the **IDispatch::Invoke** method on the control. If the call to **IDispatch::Invoke** fails, this function will throw an exception. If the status code returned by **IDispatch::Invoke** is **DISP_E_EXCEPTION**, this function throws a **COleDispatchException** object, otherwise it throws a **COleException**.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::InvokeHelperV

Invokes the method or property specified by *dwDispID*, in the context specified by *wFlags*.

```
virtual void InvokeHelperV(
   DISPID dwDispID,
   WORD wFlags,
   VARTYPE vtRet,
   void* pvRet,
   const BYTE* pbParamInfo,
   va_list argList
);
```

Parameters

dwDispID

Identifies the dispatch ID of the property or method, found on the control's **IDispatch** interface, to be invoked.

wFlags

Flags describing the context of the call to IDispatch::Invoke.

vtRet

Specifies the type of the return value. For possible values, see the Remarks section for COleDispatchDriver::InvokeHelper.

pvRet

Address of the variable that will receive the property value or return value. It must match the type specified by *vtRet*.

pbParamInfo

Pointer to a null-terminated string of bytes specifying the types of the parameters following *pbParamInfo*. For possible values, see the Remarks section for COleDispatchDriver::InvokeHelper.

argList

Pointer to a variable argument list.

Remarks

The *pbParamInfo* parameter specifies the types of the parameters passed to the method or property. Extra parameters for the method or property being invoked can be passed using the *va_list* parameter.

Typically, this function is called by **COleControlSite::InvokeHelper**.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::IsDefaultButton

Determines if the control is the default button.

```
BOOL IsDefaultButton( );
```

Return Value

Nonzero if the control is the default button on the window, otherwise zero.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::IsWindowEnabled

Determines if the control site is enabled.

```
virtual BOOL IsWindowEnabled( ) const;
```

Return Value

Nonzero if the control is enabled, otherwise zero.

Remarks

The value is retrieved from the control's Enabled stock property.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::ModifyStyle

Modifies the styles of the control.

```
virtual BOOL ModifyStyle(
   DWORD dwRemove,
   DWORD dwAdd,
   UINT nFlags
);
```

Parameters

dwRemove

The styles to be removed from the current window styles.

dwAdd

The styles to be added from the current window styles.

nFlags

Window positioning flags. For a list of possible values, see the SetWindowPos function in the *Platform SDK*.

Return Value

Nonzero if the styles are changed, otherwise zero.

Remarks

The control's stock Enabled property will be modified to match the setting for **WS_DISABLED**. The control's stock Border Style property will be modified to match the requested setting for **WS_BORDER**. All other styles are applied directly to the control's window handle, if one is present.

Modifies the window styles of the control. Styles to be added or removed can be combined by using the bitwise OR (|) operator. See the CreateWindow function in the *Platform SDK* for information about the available window styles.

If *nFlags* is nonzero, **ModifyStyle** calls the Win32 function **SetWindowPos**, and redraws the window by combining *nFlags* with the following four flags:

- **SWP_NOSIZE** Retains the current size.
- **SWP_NOMOVE** Retains the current position.
- **SWP_NOZORDER** Retains the current Z order.
- **SWP_NOACTIVATE** Does not activate the window.

To modify a window's extended styles, call ModifyStyleEx.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::ModifyStyleEx

Modifies the extended styles of the control.

```
virtual BOOL ModifyStyleEx(
   DWORD dwRemove,
   DWORD dwAdd,
   UINT nFlags
);
```

Parameters

dwRemove

The extended styles to be removed from the current window styles.

dwAdd

The extended styles to be added from the current window styles.

nFlags

Window positioning flags. For a list of possible values, see the SetWindowPos function in the *Platform SDK*.

Return Value

Nonzero if the styles are changed, otherwise zero.

Remarks

The control's stock Appearance property will be modified to match the setting for **WS_EX_CLIENTEDGE**. All other extended window styles are applied directly to the control's window handle, if one is present.

Modifies the window extended styles of the control site object. Styles to be added or removed can be combined by using the bitwise OR (|) operator. See the CreateWindowEx function in the *Platform SDK* for information about the available window styles.

If *nFlags* is nonzero, **ModifyStyleEx** calls the Win32 function **SetWindowPos**, and redraws the window by combining *nFlags* with the following four flags:

- **SWP_NOSIZE** Retains the current size.
- **SWP_NOMOVE** Retains the current position.
- **SWP_NOZORDER** Retains the current Z order.
- **SWP_NOACTIVATE** Does not activate the window.

To modify a window's extended styles, call ModifyStyle.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::MoveWindow

Changes the position of the control.

```
virtual void MoveWindow(
   int x,
   int y,
   int nWidth,
   int nHeight
);
```

Parameters

x

The new position of the left side of the window.

y

The new position of the top of the window.

nWidth

The new width of the window

nHeight

The new height of the window.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::QuickActivate

Quick activates the contained control.

```
virtual BOOL QuickActivate( );
```

Return Value

Nonzero if the control site was activated, otherwise zero.

Remarks

This function should be called only if the user is overriding the creation process of the control.

The **IPersist*::Load** and **IPersist*::InitNew** methods should be called after quick activation occurs. The control should establish its connections to the container's sinks during quick activation. However, these connections are not live until **IPersist*::Load** or **IPersist*::InitNew** has been called.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::SafeSetProperty

Sets the control property specified by *dwDispID*.

```
virtual BOOL AFX_CDECL SafeSetProperty(
   DISPID dwDispID,
   VARTYPE vtProp,
      ...
);
```

Parameters

dwDispID

Identifies the dispatch ID of the property or method, found on the control's **IDispatch** interface, to be set.

vtProp

Specifies the type of property to be set. For possible values, see the Remarks section for COleDispatchDriver::InvokeHelper.

...

A single parameter of the type specified by *vtProp*.

Return Value

Nonzero if successful; otherwise zero.

Remarks

Note
Unlike **SetProperty** and **SetPropertyV**, if an error is encountered (such as trying to set a nonexistent property), no exception is thrown.

See Also

Reference

COleControlSite Class

Hierarchy Chart

COleControlSite::SetProperty

Other Resources

COleControlSite Members

COleControlSite::SetDefaultButton

Sets the control as the default button.

```
void SetDefaultButton(
   BOOL bDefault
);
```

Parameters

bDefault

Nonzero if the control should become the default button; otherwise zero.

Remarks

Note
The control must have the **OLEMISC_ACTSLIKEBUTTON** status bit set.

See Also

Reference

COleControlSite Class
Hierarchy Chart
COleControlSite::IsDefaultButton
DECLARE_OLECTLTYPE

Other Resources

COleControlSite Members

COleControlSite::SetDlgCtrlID

Changes the value of the control's dialog item identifier.

```
virtual int SetDlgCtrlID(
   int nID
);
```

Parameters

nID

The new identifier value.

Return Value

If successful, the previous dialog item identifier of the window; otherwise 0.

Remarks

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::SetFocus

Sets focus to the control.

```
virtual CWnd* SetFocus( );
virtual CWnd* SetFocus(
   LPMSG lpmsg
);
```

Parameters

lpmsg

A pointer to a MSG structure. This structure contains the Windows message triggering the **SetFocus** request for the control contained in the current control site.

Return Value

A pointer to the window that previously had focus.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::SetProperty

Sets the control property specified by *dwDispID*.

```
virtual void AFX_CDECL SetProperty(
   DISPID dwDispID,
   VARTYPE vtProp,
      ...
);
```

Parameters

dwDispID

Identifies the dispatch ID of the property or method, found on the control's **IDispatch** interface, to be set.

vtProp

Specifies the type of property to be set. For possible values, see the Remarks section for COleDispatchDriver::InvokeHelper.

...

A single parameter of the type specified by *vtProp*.

Remarks

If **SetProperty** encounters an error, an exception is thrown.

The type of exception is determined by the return value of the attempt to set the property or method. If the return value is **DISP_E_EXCEPTION**, a **COleDispatchExcpetion** is thrown; otherwise a **COleException**.

See Also

Reference

COleControlSite Class
Hierarchy Chart
COleControlSite::SetPropertyV

Other Resources

COleControlSite Members

COleControlSite::SetPropertyV

Sets the control property specified by *dwDispID*.

```
virtual void SetPropertyV(
   DISPID dwDispID,
   VARTYPE vtProp,
   va_list argList
);
```

Parameters

dwDispID

Identifies the dispatch ID of the property or method, found on the control's **IDispatch** interface, to be set.

vtProp

Specifies the type of property to be set. For possible values, see the Remarks section for COleDispatchDriver::InvokeHelper.

argList

Pointer to the list of arguments.

Remarks

Extra parameters for the method or property being invoked can be passeed using the *arg_list* parameter. If **SetProperty** encounters an error, an exception is thrown.

The type of exception is determined by the return value of the attempt to set the property or method. If the return value is **DISP_E_EXCEPTION**, a **COleDispatchExcpetion** is thrown; otherwise a **COleException**.

See Also

Reference

COleControlSite Class
Hierarchy Chart
COleControlSite::SetProperty

Other Resources

COleControlSite Members

COleControlSite::SetWindowPos

Sets the size, position, and Z order of the control site.

```
virtual BOOL SetWindowPos(
   const CWnd* pWndInsertAfter,
   int x,
   int y,
   int cx,
   int cy,
   UINT nFlags
);
```

Parameters

pWndInsertAfter

A pointer to the window.

x

The new position of the left side of the window.

y

The new position of the top of the window.

cx

The new width of the window

cy

The new height of the window.

nFlags

Specifies the window sizing and positioning flags. For possible values, see the Remarks section for SetWindowPos in the *Platform SDK*.

Return Value

Nonzero if successful, otherwise zero.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::SetWindowText

Setsthe text for the control site.

```
virtual void SetWindowText(
   LPCTSTR lpszString
);
```

Parameters

lpszString

Pointer to a null-terminated string to be used as the new title or control text.

Remarks

This function first attempts to set the Caption stock property. If the Caption stock property is not supported, the Text property is set instead.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::ShowWindow

Sets the window's show state.

```
virtual BOOL ShowWindow(
   int nCmdShow
);
```

Parameters

nCmdShow

Specifies how the control site is to be shown. It must be one of the following values:

- **SW_HIDE** Hides this window and passes activation to another window.
- **SW_MINIMIZE** Minimizes the window and activates the top-level window in the system's list.
- **SW_RESTORE** Activates and displays the window. If the window is minimized or maximized, Windows restores it to its original size and position.
- **SW_SHOW** Activates the window and displays it in its current size and position.
- **SW_SHOWMAXIMIZED** Activates the window and displays it as a maximized window.
- **SW_SHOWMINIMIZED** Activates the window and displays it as an icon.
- **SW_SHOWMINNOACTIVE** Displays the window as an icon. The window that is currently active remains active.
- **SW_SHOWNA** Displays the window in its current state. The window that is currently active remains active.
- **SW_SHOWNOACTIVATE** Displays the window in its most recent size and position. The window that is currently active remains active.
- **SW_SHOWNORMAL** Activates and displays the window. If the window is minimized or maximized, Windows restores it to its original size and position.

Return Value

Nonzero if the function was successful, otherwise zero.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite Data Members

For information about the data members in **COleControlSite**, see COleControlSite Members.

COleControlSite::m_bIsWindowless

Determines if the object is a windowless control.

```
BOOL m_bIsWindowless;
```

Remarks

Nonzero if the control has no window, otherwise zero.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::m_ctlInfo

Information on how keyboard input is handled by the control.

```
CONTROLINFO m_ctlInfo;
```

Remarks

This information is stored in a CONTROLINFO structure.

See Also

Reference

COleControlSite Class
Hierarchy Chart
IOleControl::GetInfo

Other Resources

COleControlSite Members

COleControlSite::m_dwEventSink

Contains the connection point's cookie from the control's event sink.

```
DWORD m_dwEventSink;
```

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::m_dwMiscStatus

Contains miscellaneous information about the control.

```
DWORD m_dwMiscStatus;
```

Remarks

For more information, see OLEMISCin the Platform SDK.

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::m_dwPropNotifySink

Contains the IPropertyNotifySink cookie.

```
DWORD m_dwPropNotifySink;
```

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::m_dwStyle

Contains the Window styles of the control.

```
DWORD m_dwStyle;
```

See Also

Reference

COleControlSite Class

Hierarchy Chart

COleControlSite::ModifyStyle

Other Resources

COleControlSite Members

COleControlSite::m_hWnd

Contains the **HWND** of the control, or **NULL** if the control is windowless.

```
HWND m_hWnd;
```

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::m_iidEvents

Contains the interface ID of the control's default event sink interface.

```
IID m_iidEvents;
```

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::m_nID

Contains the control's dialog item ID.

```
UINT m_nID;
```

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::m_pActiveObject

Contains the IOleInPlaceActiveObject interface of the control.

```
LPOLEINPLACEACTIVEOBJECT m_pActiveObject;
```

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::m_pCtrlCont

Contains the control's container (representing the form).

```
COleControlContainer* m_pCtrlCont;
```

See Also

Reference

COleControlSite Class

Hierarchy Chart

COleControlContainer Class

Other Resources

COleControlSite Members

COleControlSite::m_pInPlaceObject

Contains the **IOleInPlaceObject** IOleInPlaceObject interface of the control.

```
LPOLEINPLACEOBJECT m_pInPlaceObject;
```

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::m_pObject

Contains the **IOleObjectInterface** interface of the control.

```
LPOLEOBJECT m_pObject;
```

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::m_pWindowlessObject

Contains the **IOleInPlaceObjectWindowless**IOleInPlaceObjectWindowless interface of the control.

```
IOleInPlaceObjectWindowless* m_pWindowlessObject;
```

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::m_pWndCtrl

Contains a pointer to the **CWnd** object that represents the control itself.

```
CWnd* m_pWndCtrl;
```

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleControlSite::m_rect

Contains the bounds of the control, relative to the container's window.

```
CRect m_rect;
```

See Also

Reference

COleControlSite Class

Hierarchy Chart

Other Resources

COleControlSite Members

COleConvertDialog Class

For more information, see the OLEUICONVERT structure in the Platform SDK.

```
class COleConvertDialog : public COleDialog
```

Remarks

Note
Application Wizard-generated container code uses this class.

For more information about OLE-specific dialog boxes, see the article Dialog Boxes in OLE.

Requirements

Header: afxodlgs.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

COleDialog Class
Hierarchy Chart
COleDialog Class

Other Resources

COleConvertDialog Members

COleConvertDialog Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CDialog Members

CCommonDialog Members

COleDialog Members

Data Members

m_cv	A structure that controls the behavior of the dialog box.

Construction

COleConvertDialog	Constructs a **COleConvertDialog** object.

Operations and Attributes

DoConvert	Performs the conversion specified in the dialog box.
DoModal	Displays the OLE Change Item dialog box.
GetClassID	Gets the **CLSID** associated with the chosen item.
GetDrawAspect	Specifies whether to draw item as an icon.
GetIconicMetafile	Gets a handle to the metafile associated with the iconic form of this item.
GetSelectionType	Gets the type of selection chosen.

See Also

Reference

COleConvertDialog Class

Hierarchy Chart

COleConvertDialog Member Functions

For information about the member functions in **COleConvertDialog**, see COleConvertDialog Members.

COleConvertDialog::COleConvertDialog

Constructs only a **COleConvertDialog** object.

```
explicit COleConvertDialog (
  COleClientItem* pItem,
  DWORD dwFlags = CF_SELECTCONVERTTO,
  CLSID* pClassID = NULL,
  CWnd* pParentWnd = NULL
);
```

Parameters

pItem

Points to the item to be converted or activated.

dwFlags

Creation flag, which contains any number of the following values combined using the bitwise-or operator:

- **CF_SELECTCONVERTTO** Specifies that the Convert To radio button will be selected initially when the dialog box is called. This is the default.
- **CF_SELECTACTIVATEAS** Specifies that the Activate As radio button will be selected initially when the dialog box is called.
- **CF_SETCONVERTDEFAULT** Specifies that the class whose **CLSID** is specified by the **clsidConvertDefault** member of the **m_cv** structure will be used as the default selection in the class list box when the Convert To radio button is selected.
- **CF_SETACTIVATEDEFAULT** Specifies that the class whose **CLSID** is specified by the **clsidActivateDefault** member of the **m_cv** structure will be used as the default selection in the class list box when the Activate As radio button is selected.
- **CF_SHOWHELPBUTTON** Specifies that the Help button will be displayed when the dialog box is called.

pClassID

Points to the CLSID of the item to be converted or activated. If **NULL**, the **CLSID** associated with *pItem* will be used.

pParentWnd

Points to the parent or owner window object (of type **CWnd**) to which the dialog object belongs. If it is **NULL**, the parent window of the dialog box is set to the main application window.

Remarks

To display the dialog box, call the DoModal function.

For more information, see CLSID Key and the OLEUICONVERT structure.

See Also

Reference

COleConvertDialog Class
Hierarchy Chart
COleConvertDialog::DoModal
COleConvertDialog::m_cv

Other Resources

COleConvertDialog Members

COleConvertDialog::DoConvert

Call this function, after returning successfully from DoModal, either to convert or to activate an object of type COleClientItem.

```
BOOL DoConvert(
  COleClientItem* pItem
);
```

Parameters

pItem

Points to the item to be converted or activated. Cannot be **NULL**.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The item is converted or activated according to the information selected by the user in the Convert dialog box.

See Also

Reference

COleConvertDialog Class
Hierarchy Chart
COleClientItem Class
COleConvertDialog::DoModal
COleConvertDialog::GetSelectionType
COleClientItem::ConvertTo
COleClientItem::ActivateAs

Other Resources

COleConvertDialog Members

COleConvertDialog::DoModal

Call this function to display the OLE Convert dialog box.

```
virtual INT_PTR DoModal( );
```

Return Value

Completion status for the dialog box. One of the following values:

- **IDOK** if the dialog box was successfully displayed.
- **IDCANCEL** if the user canceled the dialog box.
- **IDABORT** if an error occurred. If **IDABORT** is returned, call the COleDialog::GetLastError member function to get more information about the type of error that occurred. For a listing of possible errors, see the OleUIConvert function in the *Platform SDK*.

Remarks

If you want to initialize the various dialog box controls by setting members of the m_cv structure, you should do this before calling **DoModal**, but after the dialog object is constructed.

If **DoModal** returns **IDOK**, you can call other member functions to retrieve the settings or information that was input by the user into the dialog box.

See Also

Reference

COleConvertDialog Class
Hierarchy Chart
COleDialog::GetLastError
CDialog::DoModal
COleConvertDialog::m_cv
COleConvertDialog::DoConvert
COleConvertDialog::GetSelectionType
COleConvertDialog::GetClassID
COleConvertDialog::GetDrawAspect
COleConvertDialog::GetIconicMetafile

Other Resources

COleConvertDialog Members

COleConvertDialog::GetClassID

Call this function to get the **CLSID** associated with the item the user selected in the Convert dialog box.

```
REFCLSID GetClassID( ) const;
```

Return Value

The **CLSID** associated with the item that was selected in the Convert dialog box.

Remarks

Call this function only after DoModal returns **IDOK**.

For more information, see CLSID Key in the *Platform SDK*.

See Also

Reference

COleConvertDialog Class
Hierarchy Chart
COleConvertDialog::DoModal

Other Resources

COleConvertDialog Members

COleConvertDialog::GetDrawAspect

Call this function to determine whether the user chose to display the selected item as an icon.

```
DVASPECT GetDrawAspect( ) const;
```

Return Value

The method needed to render the object.

- **DVASPECT_CONTENT** Returned if the Display As Icon check box was not checked.
- **DVASPECT_ICON** Returned if the Display As Icon check box was checked.

Remarks

Call this function only after DoModal returns **IDOK**.

For more information on drawing aspect, see the FORMATETC data structure in the *Platform SDK*.

See Also

Reference

COleConvertDialog Class
Hierarchy Chart
COleConvertDialog::DoModal
COleConvertDialog::COleConvertDialog

Other Resources

COleConvertDialog Members

COleConvertDialog::GetIconicMetafile

Call this function to get a handle to the metafile that contains the iconic aspect of the selected item.

```
HGLOBAL GetIconicMetafile( ) const;
```

Return Value

The handle to the metafile containing the iconic aspect of the selected item, if the Display As Icon check box was checked when the dialog was dismissed by choosing **OK**; otherwise **NULL**.

See Also

Reference

COleConvertDialog Class
Hierarchy Chart
COleConvertDialog::DoModal
COleConvertDialog::COleConvertDialog
COleConvertDialog::GetDrawAspect

Other Resources

COleConvertDialog Members

COleConvertDialog::GetSelectionType

Call this function to determine the type of conversion selected in the Convert dialog box.

```
UINT GetSelectionType( ) const;
```

Return Value

Type of selection made.

Remarks

The return type values are specified by the **Selection** enumeration type declared in the **COleConvertDialog** class.

```
enum Selection
{
   noConversion,
   convertItem,
   activateAs
};
```

Brief descriptions of these values follow:

- **COleConvertDialog::noConversion** Returned if either the dialog box was canceled or the user selected no conversion. If **COleConvertDialog::DoModal** returned **IDOK**, it is possible that the user selected a different icon than the one previously selected.
- **COleConvertDialog::convertItem** Returned if the Convert To radio button was checked, the user selected a different item to convert to, and **DoModal** returned **IDOK**.
- **COleConvertDialog::activateAs** Returned if the Activate As radio button was checked, the user selected a different item to activate, and **DoModal** returned **IDOK**.

See Also

Reference

COleConvertDialog Class
Hierarchy Chart
COleConvertDialog::DoModal
COleConvertDialog::COleConvertDialog

Other Resources

COleConvertDialog Members

COleConvertDialog Data Members

For information about the data members in **COleConvertDialog**, see COleConvertDialog Members.

COleConvertDialog::m_cv

Structure of type **OLEUICONVERT** used to control the behavior of the Convert dialog box.

```
OLEUICONVERT m_cv;
```

Remarks

Members of this structure can be modified either directly or through member functions.

For more information, see the OLEUICONVERT structure in the *Platform SDK*.

See Also

Reference

COleConvertDialog Class
Hierarchy Chart
COleConvertDialog::COleConvertDialog
COleConvertDialog::DoModal

Other Resources

COleConvertDialog Members

COleCurrency Class

Encapsulates the CURRENCY data type of OLE automation.

```
class COleCurrency
```

Remarks

COleCurrency does not have a base class.

CURRENCY is implemented as an 8-byte, two's-complement integer value scaled by 10,000. This gives a fixed-point number with 15 digits to the left of the decimal point and 4 digits to the right. The **CURRENCY** data type is extremely useful for calculations involving money, or for any fixed-point calculation where accuracy is important. It is one of the possible types for the VARIANT data type of OLE automation.

COleCurrency also implements some basic arithmetic operations for this fixed-point type. The supported operations have been selected to control the rounding errors which occur during fixed-point calculations.

For more information, see the CURRENCY and VARIANT entries in the *Platform SDK*.

Requirements

Header: afxdisp.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart

COleVariant Class

Other Resources

COleCurrency Members

COleCurrency Members

Construction

COleCurrency	Constructs a **COleCurrency** object.

Attributes

GetStatus	Gets the status (validity) of this **COleCurrency** object.
SetStatus	Sets the status (validity) for this **COleCurrency** object.

Operations

Format	Generates a formatted string representation of a **COleCurrency** object.
ParseCurrency	Reads a **CURRENCY** value from a string and sets the value of **COleCurrency**.
SetCurrency	Sets the value of this **COleCurrency** object.

Operators

operator *, /	Scales a **COleCurrency** value by an integer value.
operator *=, /=	Scales this **COleCurrency** value by an integer value.
operator +, -	Adds, subtracts, and changes sign of **COleCurrency** values.
operator +=, -=	Adds and subtracts a **COleCurrency** value from this **COleCurrency** object.
operator =	Copies a **COleCurrency** value.
operator ==, <, <=, etc.	Compares two **COleCurrency** values.
operator CURRENCY	Converts a **COleCurrency** value into a **CURRENCY**.

Data Members

m_cur	Contains the underlying **CURRENCY** for this **COleCurrency** object.
m_status	Contains the status of this **COleCurrency** object.

Archive/Dump

operator <<	Outputs a **COleCurrency** value to **CArchive** or **CDumpContext**.
operator >>	Inputs a **COleCurrency** object from **CArchive**.

See Also

Reference

COleCurrency Class

Hierarchy Chart

COleCurrency Member Functions

For information about the member functions in **COleCurrency**, see COleCurrency Members.

COleCurrency::COleCurrency

Constructs a **COleCurrency** object.

```
COleCurrency( );
COleCurrency(
   CURRENCY cySrc
);
COleCurrency(
   const COleCurrency& curSrc
);
COleCurrency(
   const VARIANT& varSrc
);
COleCurrency(
   long nUnits,
   long nFractionalUnits
);
```

Parameters

cySrc

A **CURRENCY** value to be copied into the new **COleCurrency** object.

curSrc

An existing **COleCurrency** object to be copied into the new **COleCurrency** object.

varSrc

An existing **VARIANT** data structure (possibly a **COleVariant** object) to be converted to a currency value (**VT_CY**) and copied into the new **COleCurrency** object.

nUnits, *nFractionalUnits*

Indicate the units and fractional part (in 1/10,000's) of the value to be copied into the new **COleCurrency** object.

Remarks

All of these constructors create new **COleCurrency** objects initialized to the specified value. A brief description of each of these constructors follows. Unless otherwise noted, the status of the new **COleCurrency** item is set to valid.

- **COleCurrency( )** Constructs a **COleCurrency** object initialized to 0 (zero).
- **COleCurrency(** *cySrc* **)** Constructs a **COleCurrency** object from a CURRENCY value.
- **COleCurrency(** *curSrc* **)** Constructs a **COleCurrency** object from an existing **COleCurrency** object. The new object has the same status as the source object.
- **COleCurrency(** *varSrc* **)** Constructs a **COleCurrency** object. Attempts to convert a VARIANT structure or **COleVariant** object to a currency (**VT_CY**) value. If this conversion is successful, the converted value is copied into the new **COleCurrency** object. If it is not, the value of the **COleCurrency** object is set to zero (0) and its status to invalid.
- **COleCurrency(** *nUnits*, *nFractionalUnits* **)** Constructs a **COleCurrency** object from the specified numerical components. If the absolute value of the fractional part is greater than 10,000, the appropriate adjustment is made to the units. Note that the units and fractional part are specified by signed long values.

For more information, see the CURRENCY and VARIANT entries in the *Platform SDK*.

Example

The following examples show the effects of the zero-parameter and two-parameter constructors:

```
COleCurrency curZero;         // value: 0.0000
COleCurrency curA(4, 500);    // value: 4.0500
COleCurrency curB(2, 11000);  // value: 3.1000
```

```
COleCurrency curC(2, -50);    // value: 1.9950
```

See Also

Reference

COleCurrency Class
Hierarchy Chart
COleCurrency::SetCurrency
COleCurrency::operator =
COleCurrency::GetStatus
COleCurrency::m_cur
COleCurrency::m_status

Other Resources

COleCurrency Members

COleCurrency::Format

Call this member function to create a formatted representation of the currency value.

```
CString Format(
   DWORD dwFlags = 0,
   LCID lcid = LANG_USER_DEFAULT
) const;
```

Parameters

dwFlags

Indicates flags for locale settings, possibly the following flag:

- **LOCALE_NOUSEROVERRIDE** Use the system default locale settings, rather than custom user settings.

lcid

Indicates locale ID to use for the conversion.

Return Value

A **CString** that contains the formatted currency value.

Remarks

It formats the value using the national language specifications (locale IDs). A currency symbol is not included in the value returned. If the status of this **COleCurrency** object is null, the return value is an empty string. If the status is invalid, the return string is specified by the string resource **IDS_INVALID_CURRENCY**.

Example

```
COleCurrency curA;           // value: 0.0000
curA.SetCurrency(4, 500);    // value: 4.0500

// value returned: 4.05
curA.Format(0, MAKELCID(MAKELANGID(LANG_CHINESE,
     SUBLANG_CHINESE_SINGAPORE), SORT_DEFAULT));
// value returned: 4,05
curA.Format(0, MAKELCID(MAKELANGID(LANG_GERMAN,
     SUBLANG_GERMAN_AUSTRIAN), SORT_DEFAULT));
```

Note

For a discussion of locale ID values, see the section Supporting Multiple National Languages in the *Platform SDK*.

See Also

Reference

COleCurrency Class
Hierarchy Chart
COleCurrency::ParseCurrency
COleCurrency::GetStatus

Other Resources

COleCurrency Members

COleCurrency::GetStatus

Call this member function to get the status (validity) of a given **COleCurrency** object.

```
CurrencyStatus GetStatus( ) const;
```

Return Value

Returns the status of this **COleCurrency** value.

Remarks

The return value is defined by the **CurrencyStatus** enumerated type that is defined within the **COleCurrency** class.

```
enum CurrencyStatus{
   valid = 0,
   invalid = 1,
   null = 2,
};
```

For a brief description of these status values, see the following list:

- **COleCurrency::valid** Indicates that this **COleCurrency** object is valid.
- **COleCurrency::invalid** Indicates that this **COleCurrency** object is invalid; that is, its value may be incorrect.
- **COleCurrency::null** Indicates that this **COleCurrency** object is null, that is, that no value has been supplied for this object. (This is "null" in the database sense of "having no value," as opposed to the C++ **NULL**.)

The status of a **COleCurrency** object is invalid in the following cases:

- If its value is set from a **VARIANT** or **COleVariant** value that could not be converted to a currency value.
- If this object has experienced an overflow or underflow during an arithmetic assignment operation, for example **+=** or ***=**.
- If an invalid value was assigned to this object.
- If the status of this object was explicitly set to invalid using SetStatus.

For more information on operations that may set the status to invalid, see the following member functions:

- COleCurrency
- operator =
- operator +, -
- operator +=, -=
- operator *, /
- operator *=, /=

Example

```
// even an empty COleCurrency is valid
COleCurrency cy;
ASSERT(cy.GetStatus() == COleCurrency::valid);

// always valid after being set
cy.SetCurrency(4, 500);
ASSERT(cy.GetStatus() == COleCurrency::valid);

// some conversions aren't possible and will
// cause an invalid state, like this:
```

```
CByteArray array;
COleVariant varBogus(array);
cy = varBogus;
ASSERT(cy.GetStatus() == COleCurrency::invalid);
```

See Also

Reference

COleCurrency Class

Hierarchy Chart

COleCurrency::SetStatus

COleCurrency::m_status

Other Resources

COleCurrency Members

COleCurrency::ParseCurrency

Call this member function to parse a string to read a currency value.

```
BOOL ParseCurrency(
   LPCTSTR lpszCurrency,
   DWORD dwFlags = 0,
   LCID lcid = LANG_USER_DEFAULT
);
throw(
   CMemoryException*
);
throw(
   COleException*
);
```

Parameters

lpszCurrency

A pointer to the null-terminated string which is to be parsed.

dwFlags

Indicates flags for locale settings, possibly the following flag:

- **LOCALE_NOUSEROVERRIDE** Use the system default locale settings, rather than custom user settings.

lcid

Indicates locale ID to use for the conversion.

Return Value

Nonzero if the string was successfully converted to a currency value, otherwise 0.

Remarks

It uses national language specifications (locale IDs) for the meaning of nonnumeric characters in the source string.

For a discussion of locale ID values, see the section Supporting Multiple National Languages in the *Platform SDK*.

If the string was successfully converted to a currency value, the value of this **COleCurrency** object is set to that value and its status to valid.

If the string could not be converted to a currency value or if there was a numerical overflow, the status of this **COleCurrency** object is invalid.

If the string conversion failed due to memory allocation errors, this function throws a CMemoryException. In any other error state, this function throws a COleException.

Example

```
// works if default locale has dot decimal point
COleCurrency cur;
cur.ParseCurrency("$135.95", 0);
ASSERT(cur == COleCurrency(135, 9500));
```

See Also

Reference

COleCurrency Class
Hierarchy Chart
COleCurrency::Format
COleCurrency::GetStatus

Other Resources

COleCurrency Members

COleCurrency::SetCurrency

Call this member function to set the units and fractional part of this **COleCurrency** object.

```
void SetCurrency(
   long nUnits,
   long nFractionalUnits
);
```

Parameters

nUnits, *nFractionalUnits*

Indicate the units and fractional part (in 1/10,000's) of the value to be copied into this **COleCurrency** object.

Remarks

If the absolute value of the fractional part is greater than 10,000, the appropriate adjustment is made to the units, as shown in the third of the following examples.

Note that the units and fractional part are specified by signed long values. The fourth of the following examples shows what happens when the parameters have different signs.

Example

```
COleCurrency curA;           // value: 0.0000
curA.SetCurrency(4, 500);    // value: 4.0500
curA.SetCurrency(2, 11000);  // value: 3.1000
curA.SetCurrency(2, -50);    // value: 1.9950
```

See Also

Reference

COleCurrency Class
Hierarchy Chart
COleCurrency::COleCurrency
COleCurrency::operator =
COleCurrency::m_cur

Other Resources

COleCurrency Members

COleCurrency::SetStatus

Call this member function to set the status (validity) of this **COleCurrency** object.

```
void SetStatus(
   CurrencyStatus status
);
```

Parameters

status

The new status for this **COleCurrency** object.

Remarks

The *status* parameter value is defined by the **CurrencyStatus** enumerated type, which is defined within the **COleCurrency** class.

```
enum CurrencyStatus{
   valid = 0,
   invalid = 1,
   null = 2,
};
```

For a brief description of these status values, see the following list:

- **COleCurrency::valid** Indicates that this **COleCurrency** object is valid.
- **COleCurrency::invalid** Indicates that this **COleCurrency** object is invalid; that is, its value may be incorrect.
- **COleCurrency::null** Indicates that this **COleCurrency** object is null, that is, that no value has been supplied for this object. (This is "null" in the database sense of "having no value," as opposed to the C++ **NULL**.)

Caution
This function is for advanced programming situations. This function does not alter the data in this object. It will most of ten be used to set the status to null or invalid. Note that the assignment operator (operator =) and SetCurrency do set t he status to of the object based on the source value(s).

See Also

Reference

COleCurrency Class
Hierarchy Chart
COleCurrency::GetStatus
COleCurrency::operator =
COleCurrency::SetCurrency
COleCurrency::m_status

Other Resources

COleCurrency Members

COleCurrency Operators

For information about the operators in **COleCurrency**, see COleCurrency Members.

COleCurrency::operator =

These overloaded assignment operators copy the source currency value into this **COleCurrency** object.

```
const COleCurrency& operator =(
   CURRENCY cySrc
);
const COleCurrency& operator =(
   const COleCurrency& curSrc
);
const COleCurrency& operator =(
   const VARIANT& varSrc
);
```

Remarks

A brief description of each operator follows:

- **operator =(** *cySrc* **)** The CURRENCY value is copied into the **COleCurrency** object and its status is set to valid.
- **operator =(** *curSrc* **)** The value and status of the operand, an existing **COleCurrency** object are copied into this **COleCurrency** object.
- **operator =(** *varSrc* **)** If the conversion of the VARIANT value (or COleVariant object) to a currency (**VT_CY**) is successful, the converted value is copied into this **COleCurrency** object and its status is set to valid. If the conversion is not successful, the value of the **COleCurrency** object is set to 0 and its status to invalid.

For more information, see the CURRENCY and VARIANT entries in the *Platform SDK*.

Example

```
// set to 35.0050
COleCurrency cur1(35, 50);
COleCurrency cur2;

// operator= copies COleCurrency types
cur2 = cur1;
ASSERT(cur1 == cur2);

// can be used to assign a CURRENCY type, as well
CURRENCY cy;
cy.Hi = 0;
cy.Lo = 350050;
cy.int64 = 350050;

// perform assignment
COleCurrency cur3;
cur3 = cy;
ASSERT(cur3 == cur1);
```

See Also

Reference

COleCurrency Class
Hierarchy Chart
COleCurrency::COleCurrency
COleCurrency::SetCurrency
COleCurrency::GetStatus

Other Resources

COleCurrency Members

COleCurrency::operator +, -

These operators allow you to add and subtract two **COleCurrency** values to and from each other and to change the sign of a **COleCurrency** value.

```
COleCurrency operator +(
   const COleCurrency& cur
) const;
COleCurrency operator -(
   const COleCurrency& cur
) const;
COleCurrency operator -( ) const;
```

Remarks

If either of the operands is null, the status of the resulting **COleCurrency** value is null.

If the arithmetic operation overflows, the resulting **COleCurrency** value is invalid.

If the operand is invalid and the other is not null, the status of the resulting **COleCurrency** value is invalid.

For more information on the valid, invalid, and null status values, see the m_status member variable.

Example

```
// 35.0050
COleCurrency cur1(35, 50);
// 2.0075
COleCurrency cur2(2, 75);
COleCurrency cur3;

// sum is 37.0125
cur3 = cur1 + cur2;
ASSERT(cur3 == COleCurrency(37, 125));

// difference is 32.9975
cur3 = cur1 - cur2;
ASSERT(cur3 == COleCurrency(32, 9975));
```

See Also

Reference

COleCurrency Class
Hierarchy Chart
COleCurrency::operator +=, -=
COleCurrency::GetStatus

Other Resources

COleCurrency Members

COleCurrency::operator +=, -=

Allow you to add and subtract a **COleCurrency** value to and from this **COleCurrency** object.

```
const COleCurrency& operator +=(
   const COleCurrency& cur
);
const COleCurrency& operator -=(
   const COleCurrency& cur
);
```

Remarks

If either of the operands is null, the status of this **COleCurrency** object is set to null.

If the arithmetic operation overflows, the status of this **COleCurrency** object is set to invalid.

If either of the operands is invalid and the other is not null, the status of this **COleCurrency** object is set to invalid.

For more information on the valid, invalid, and null status values, see the m_status member variable.

Example

```
// both set to 35.0050
COleCurrency cur1(35, 50);
COleCurrency cur2(35, 50);

// adding 2.0075 results in is 37.0125
cur1 += COleCurrency(2, 75);
ASSERT(cur1 == COleCurrency(37, 125));

// subtracting 2.0075 results in 32.9975
cur2 -= COleCurrency(2, 75);
ASSERT(cur2 == COleCurrency(32, 9975));
```

See Also

Reference

COleCurrency Class
Hierarchy Chart
COleCurrency::operator +, -
COleCurrency::GetStatus

Other Resources

COleCurrency Members

COleCurrency::operator *, /

Allow you to scale a **COleCurrency** value by an integral value.

```
COleCurrency operator *(
   long nOperand
) const;
COleCurrency operator /(
   long nOperand
) const;
```

Remarks

If the **COleCurrency** operand is null, the status of the resulting **COleCurrency** value is null.

If the arithmetic operation overflows or underflows, the status of the resulting **COleCurrency** value is invalid.

If the **COleCurrency** operand is invalid, the status of the resulting **COleCurrency** value is invalid.

For more information on the valid, invalid, and null status values, see the m_status member variable.

Example

```
// 35 units and 50/10000, or 35.0050
COleCurrency cur1(35, 50);
COleCurrency cur2;

// divided by two is 17.5025
cur2 = cur1 / 2;
ASSERT(cur2 == COleCurrency(17, 5025));

// multiplied by two is 70.0100
cur2 = cur1 * 2;
ASSERT(cur2 == COleCurrency(70, 100));
```

See Also

Reference

COleCurrency Class
Hierarchy Chart
COleCurrency::operator *=, /=
COleCurrency::GetStatus

Other Resources

COleCurrency Members

COleCurrency::operator *=, /=

Allow you to scale this **COleCurrency** value by an integral value.

```
const COleCurrency& operator *=(
   long nOperand
);
const COleCurrency& operator /=(
   long nOperand
);
```

Remarks

If the **COleCurrency** operand is null, the status of this **COleCurrency** object is set to null.

If the arithmetic operation overflows, the status of this **COleCurrency** object is set to invalid.

If the **COleCurrency** operand is invalid, the status of this **COleCurrency** object is set to invalid.

For more information on the valid, invalid, and null status values, see the m_status member variable.

Example

```
// both set to 35.0050
COleCurrency cur1(35, 50);
COleCurrency cur2(35, 50);

// divide in half
cur1 /= 2;
ASSERT(cur1 == COleCurrency(17, 5025));

// multiply by two
cur2 *= 2;
ASSERT(cur2 == COleCurrency(70, 100));
```

See Also

Reference

COleCurrency Class
Hierarchy Chart
COleCurrency::operator *, /
COleCurrency::GetStatus

Other Resources

COleCurrency Members

COleCurrency::operator CURRENCY

Returns a CURRENCY structure whose value is copied from this **COleCurrency** object.

```
operator CURRENCY( ) const;
```

Remarks

For more information, see the CURRENCY entry in the *Platform SDK*.

See Also

Reference

COleCurrency Class
Hierarchy Chart
COleCurrency::m_cur
COleCurrency::SetCurrency

Other Resources

COleCurrency Members

COleCurrency Relational Operators

Compare two currency values and return nonzero if the condition is true; otherwise 0.

```
BOOL operator ==(
   const COleCurrency& cur
) const;
BOOL operator !=(
   const COleCurrency& cur
) const;
BOOL operator <(
   const COleCurrency& cur
) const;
BOOL operator >(
   const COleCurrency& cur
) const;
BOOL operator <=(
   const COleCurrency& cur
) const;
BOOL operator >=(
   const COleCurrency& cur
) const;
```

Remarks

Note

The return value of the ordering operations (**<**, **<=**, **>**, **>=**) is undefined if the status of either operand is null or invalid. The equality operators (**==**, **!=**) consider the status of the operands.

Example

```
COleCurrency curOne(3, 5000);             // 3.5
COleCurrency curTwo(curOne);              // 3.5
BOOL b;
b = curOne == curTwo;                     // TRUE

curTwo.SetStatus(COleCurrency::invalid);
b = curOne == curTwo;                     // FALSE, different status
b = curOne != curTwo;                     // TRUE, different status
b = curOne < curTwo;                      // FALSE, same value
b = curOne > curTwo;                      // FALSE, same value
b = curOne <= curTwo;                     // TRUE, same value
b = curOne >= curTwo;                     // TRUE, same value
```

Note

The last four lines of the preceding example will **ASSERT** in debug mode.

See Also

Reference

COleCurrency Class

Hierarchy Chart

COleCurrency::GetStatus

Other Resources

COleCurrency Members

COleCurrency::operator <<, >>

Supports diagnostic dumping and storing to an archive.

```
friend CDumpContext& operator <<(
   CDumpContext& dc,
   COleCurrency curSrc
);
friend CArchive& operator <<(
   CArchive& ar,
   COleCurrency curSrc
);
friend CArchive& operator >>(
   CArchive& ar,
   COleCurrency& curSrc
);
```

Remarks

The extraction (**>>**) operator supports loading from an archive.

See Also

Reference

COleCurrency Class
Hierarchy Chart
CDumpContext Class
CArchive Class

Other Resources

COleCurrency Members

COleCurrency Data Members

For information about the data members in **COleCurrency**, see COleCurrency Members.

COleCurrency::m_cur

The underlying CURRENCY structure for this **COleCurrency** object.

Remarks

Caution
Changing the value in the **CURRENCY** structure accessed by the pointer returned by this function will change the value of this **COleCurrency** object. It does not change the status of this **COleCurrency** object.

For more information, see the CURRENCY entry in the *Platform SDK*.

See Also

Reference

COleCurrency Class
Hierarchy Chart
COleCurrency::COleCurrency
COleCurrency::operator CURRENCY
COleCurrency::SetCurrency

Other Resources

COleCurrency Members

COleCurrency::m_status

The type of this data member is the enumerated type **CurrencyStatus**, which is defined within the **COleCurrency** class.

```
enum CurrencyStatus{
   valid = 0,
   invalid = 1,
   null = 2,
};
```

Remarks

For a brief description of these status values, see the following list:

- **COleCurrency::valid** Indicates that this **COleCurrency** object is valid.
- **COleCurrency::invalid** Indicates that this **COleCurrency** object is invalid; that is, its value may be incorrect.
- **COleCurrency::null** Indicates that this **COleCurrency** object is null, that is, that no value has been supplied for this object. (This is "null" in the database sense of "having no value," as opposed to the C++ **NULL**.)

The status of a **COleCurrency** object is invalid in the following cases:

- If its value is set from a **VARIANT** or **COleVariant** value that could not be converted to a currency value.
- If this object has experienced an overflow or underflow during an arithmetic assignment operation, for example **+=** or ***=**.
- If an invalid value was assigned to this object.
- If the status of this object was explicitly set to invalid using SetStatus.

For more information on operations that may set the status to invalid, see the following member functions:

- COleCurrency
- operator =
- operator +, -
- operator +=, -=
- operator *, /
- operator *=, /=

Caution
This data member is for advanced programming situations. You should use the inline member functions GetStatus and SetStatus. See **SetStatus** for further cautions regarding explicitly setting this data member.

See Also

Reference

COleCurrency Class
Hierarchy Chart
COleCurrency::GetStatus
COleCurrency::SetStatus

Other Resources

COleCurrency Members

COleDataObject Class

Used in data transfers for retrieving data in various formats from the Clipboard, through drag and drop, or from an embedded OLE item.

```
class COleDataObject
```

Remarks

COleDataObject does not have a base class.

These kinds of data transfers include a source and a destination. The data source is implemented as an object of the COleDataSource class. Whenever a destination application has data dropped in it or is asked to perform a paste operation from the Clipboard, an object of the **COleDataObject** class must be created.

This class enables you to determine whether the data exists in a specified format. You can also enumerate the available data formats or check whether a given format is available and then retrieve the data in the preferred format. Object retrieval can be accomplished in several different ways, including the use of a CFile, an **HGLOBAL**, or an **STGMEDIUM** structure.

For more information, see the STGMEDIUM structure in the *Platform SDK*.

For more information about using data objects in your application, see the article Data Objects and Data Sources (OLE).

Requirements

Header: afxole.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

HIERSVR Sample: Demonstrates a Server Application with OLE Drag and Drop
OCLIENT Sample: Illustrates a Visual Editing Container Application

Reference

Hierarchy Chart
COleDataSource Class
COleClientItem Class
COleServerItem Class
COleDataSource::DoDragDrop
CView::OnDrop

Other Resources

COleDataObject Members

COleDataObject Members

Construction

COleDataObject	Constructs a **COleDataObject** object.

Operations

Attach	Attaches the specified OLE data object to the **COleDataObject**.
AttachClipboard	Attaches the data object that is on the Clipboard.
BeginEnumFormats	Prepares for one or more subsequent **GetNextFormat** calls.
Detach	Detaches the associated **IDataObject** object.
GetData	Copies data from the attached OLE data object in a specified format.
GetFileData	Copies data from the attached OLE data object into a **CFile** pointer in the specified format.
GetGlobalData	Copies data from the attached OLE data object into an **HGLOBAL** in the specified format.
GetNextFormat	Returns the next data format available.
IsDataAvailable	Checks whether data is available in a specified format.
Release	Detaches and releases the associated **IDataObject** object.

See Also

Reference

COleDataObject Class

Hierarchy Chart

COleDataObject Member Functions

For information about the member functions in COleDataObject, see COleDataObject Members.

COleDataObject::Attach

Call this function to associate the **COleDataObject** object with an OLE data object.

```
void Attach(
   LPDATAOBJECT lpDataObject,
   BOOL bAutoRelease = TRUE
);
```

Parameters

lpDataObject

Points to an OLE data object.

bAutoRelease

TRUE if the OLE data object should be released when the **COleDataObject** object is destroyed; otherwise **FALSE**.

Remarks

For more information, see IDataObject in the *Platform SDK*.

See Also

Reference

COleDataObject Class
Hierarchy Chart
COleDataObject::AttachClipboard
COleDataObject::Detach
COleDataObject::Release

Other Resources

COleDataObject Members

COleDataObject::AttachClipboard

Call this function to attach the data object that is currently on the Clipboard to the **COleDataObject** object.

```
BOOL AttachClipboard( );
```

Return Value

Nonzero if successful; otherwise 0.

Remarks

Note
Calling this function locks the Clipboard until this data object is released. The data object is released in the destructor for the **COleDataObject**. For more information, see OpenClipboard and CloseClipboard in the Win32 documention.

See Also

Reference

COleDataObject Class
Hierarchy Chart
COleDataObject::Attach
COleDataObject::Detach
COleDataObject::Release

Other Resources

COleDataObject Members

COleDataObject::BeginEnumFormats

Call this function to prepare for subsequent calls to **GetNextFormat** for retrieving a list of data formats from the item.

```
void BeginEnumFormats( );
```

Remarks

After a call to **BeginEnumFormats**, the position of the first format supported by this data object is stored. Successive calls to **GetNextFormat** will enumerate the list of available formats in the data object.

To check on the availability of data in a given format, use COleDataObject::IsDataAvailable.

For more information, see IDataObject::EnumFormatEtc in the *Platform SDK*.

See Also

Reference

COleDataObject Class
Hierarchy Chart
COleDataObject::GetNextFormat
COleDataObject::IsDataAvailable

Other Resources

COleDataObject Members

COleDataObject::COleDataObject

Constructs a **COleDataObject** object.

```
COleDataObject( );
```

Remarks

A call to COleDataObject::Attach or COleDataObject::AttachClipboard must be made before calling other **COleDataObject** functions.

Note
Since one of the parameters to the drag-and-drop handlers is a pointer to a **COleDataObject**, there is no need to call this constructor to support drag and drop.

See Also

Reference

COleDataObject Class
Hierarchy Chart
COleDataObject::Attach
COleDataObject::AttachClipboard
COleDataObject::Release

Other Resources

COleDataObject Members

COleDataObject::Detach

Call this function to detach the **COleDataObject** object from its associated OLE data object without releasing the data object.

```
LPDATAOBJECT Detach( );
```

Return Value

A pointer to the OLE data object that was detached.

Remarks

See Also

Reference

COleDataObject Class

Hierarchy Chart

COleDataObject::Attach

COleDataObject::Release

Other Resources

COleDataObject Members

COleDataObject::GetData

Call this function to retrieve data from the item in the specified format.

```
BOOL GetData(
   CLIPFORMAT cfFormat,
   LPSTGMEDIUM lpStgMedium,
   LPFORMATETC lpFormatEtc = NULL
);
```

Parameters

cfFormat

The format in which data is to be returned. This parameter can be one of the predefined Clipboard formats or the value returned by the native Windows RegisterClipboardFormat function.

lpStgMedium

Points to a STGMEDIUM structure that will receive data.

lpFormatEtc

Points to a FORMATETC structure describing the format in which data is to be returned. Provide a value for this parameter if you want to specify additional format information beyond the Clipboard format specified by *cfFormat*. If it is **NULL**, the default values are used for the other fields in the **FORMATETC** structure.

Return Value

Nonzero if successful; otherwise 0.

Remarks

For more information, see IDataObject::GetData, STGMEDIUM, and FORMATETC in the *Platform SDK*.

For more information, see RegisterClipboardFormat in the *Platform SDK*.

See Also

Reference

COleDataObject Class
Hierarchy Chart
COleDataObject::GetFileData
COleDataObject::GetGlobalData
COleDataObject::IsDataAvailable

Other Resources

COleDataObject Members

COleDataObject::GetFileData

Call this function to create a **CFile** or **CFile**-derived object and to retrieve data in the specified format into a **CFile** pointer.

```
CFile* GetFileData(
   CLIPFORMAT cfFormat,
   LPFORMATETC lpFormatEtc = NULL
);
```

Parameters

cfFormat

The format in which data is to be returned. This parameter can be one of the predefined Clipboard formats or the value returned by the native Windows RegisterClipboardFormat function.

lpFormatEtc

Points to a FORMATETC structure describing the format in which data is to be returned. Provide a value for this parameter if you want to specify additional format information beyond the Clipboard format specified by *cfFormat*. If it is **NULL**, the default values are used for the other fields in the **FORMATETC** structure.

Return Value

Pointer to the new **CFile** or **CFile**-derived object containing the data if successful; otherwise **NULL**.

Remarks

Depending on the medium the data is stored in, the actual type pointed to by the return value may be **CFile**, **CSharedFile**, or **COleStreamFile**.

Note
The **CFile** object accessed by the return value of this function is owned by the caller. It is the responsibility of the caller to **delete** the **CFile** object, thereby closing the file.

For more information, see FORMATETC in the *Platform SDK*.

For more information, see RegisterClipboardFormat in the *Platform SDK*.

See Also

Reference

COleDataObject Class
Hierarchy Chart
COleDataObject::GetData
COleDataObject::GetGlobalData
COleDataObject::IsDataAvailable

Other Resources

COleDataObject Members

COleDataObject::GetGlobalData

Call this function to allocate a global memory block and to retrieve data in the specified format into an **HGLOBAL**.

```
HGLOBAL GetGlobalData(
   CLIPFORMAT cfFormat,
   LPFORMATETC lpFormatEtc = NULL
);
```

Parameters

cfFormat

The format in which data is to be returned. This parameter can be one of the predefined Clipboard formats or the value returned by the native Windows RegisterClipboardFormat function.

lpFormatEtc

Points to a FORMATETC structure describing the format in which data is to be returned. Provide a value for this parameter if you want to specify additional format information beyond the Clipboard format specified by *cfFormat*. If it is **NULL**, the default values are used for the other fields in the **FORMATETC** structure.

Return Value

The handle of the global memory block containing the data if successful; otherwise **NULL**.

Remarks

For more information, see FORMATETC in the *Platform SDK*.

For more information, see RegisterClipboardFormat in the *Platform SDK*.

See Also

Reference

COleDataObject Class
Hierarchy Chart
COleDataObject::GetData
COleDataObject::GetFileData
COleDataObject::IsDataAvailable

Other Resources

COleDataObject Members

COleDataObject::GetNextFormat

Call this function repeatedly to obtain all the formats available for retrieving data from the item.

```
BOOL GetNextFormat(
   LPFORMATETC lpFormatEtc
);
```

Parameters

lpFormatEtc

Points to the FORMATETC structure that receives the format information when the function call returns.

Return Value

Nonzero if another format is available; otherwise 0.

Remarks

After a call to COleDataObject::BeginEnumFormats, the position of the first format supported by this data object is stored. Successive calls to **GetNextFormat** will enumerate the list of available formats in the data object. Use these functions to list the available formats.

To check for the availability of a given format, call COleDataObject::IsDataAvailable.

For more information, see IEnumXXXX::Next in the *Platform SDK*.

See Also

Reference

COleDataObject Class
Hierarchy Chart
COleDataObject::BeginEnumFormats
COleDataObject::GetData
COleDataObject::GetFileData
COleDataObject::GetGlobalData

Other Resources

COleDataObject Members

COleDataObject::IsDataAvailable

Call this function to determine if a particular format is available for retrieving data from the OLE item.

```
BOOL IsDataAvailable(
   CLIPFORMAT cfFormat,
   LPFORMATETC lpFormatEtc = NULL
);
```

Parameters

cfFormat

The Clipboard data format to be used in the structure pointed to by *lpFormatEtc*. This parameter can be one of the predefined Clipboard formats or the value returned by the native Windows RegisterClipboardFormat function.

lpFormatEtc

Points to a FORMATETC structure describing the format desired. Provide a value for this parameter only if you want to specify additional format information beyond the Clipboard format specified by *cfFormat*. If it is **NULL**, the default values are used for the other fields in the **FORMATETC** structure.

Return Value

Nonzero if data is available in the specified format; otherwise 0.

Remarks

This function is useful before calling **GetData**, **GetFileData**, or **GetGlobalData**.

For more information, see IDataObject::QueryGetData and FORMATETC in the *Platform SDK*.

For more information, see RegisterClipboardFormat in the *Platform SDK*.

Example

See the example for CRichEditView::QueryAcceptData.

See Also

Reference

COleDataObject Class
Hierarchy Chart
COleDataObject::BeginEnumFormats
COleDataObject::GetData
COleDataObject::GetFileData
COleDataObject::GetGlobalData
COleDataObject::GetNextFormat

Other Resources

COleDataObject Members

COleDataObject::Release

Call this function to release ownership of the IDataObject object that was previously associated with the **COleDataObject** object.

```
void Release( );
```

Remarks

The **IDataObject** was associated with the **COleDataObject** by calling **Attach** or **AttachClipboard** explicitly or by the framework. If the *bAutoRelease* parameter of **Attach** is **FALSE**, the **IDataObject** object will not be released. In this case, the caller is responsible for releasing the **IDataObject** by calling IUnknown::Release.

See Also

Reference

COleDataObject Class
Hierarchy Chart
COleDataObject::Attach
COleDataObject::COleDataObject
COleDataObject::Detach

Other Resources

COleDataObject Members

COleDataSource Class

Acts as a cache into which an application places the data that it will offer during data transfer operations, such as Clipboard or drag-and-drop operations.

```
class COleDataSource : public CCmdTarget
```

Remarks

You can create OLE data sources directly. Alternately, the COleClientItem and COleServerItem classes create OLE data sources in response to their **CopyToClipboard** and **DoDragDrop** member functions. See COleServerItem::CopyToClipboard for a brief description. Override the **OnGetClipboardData** member function of your client item or server item class to add additional Clipboard formats to the data in the OLE data source created for the **CopyToClipboard** or **DoDragDrop** member function.

Whenever you want to prepare data for a transfer, you should create an object of this class and fill it with your data using the most appropriate method for your data. The way it is inserted into a data source is directly affected by whether the data is supplied immediately (immediate rendering) or on demand (delayed rendering). For every Clipboard format in which you are providing data by passing the Clipboard format to be used (and an optional FORMATETC structure), call DelayRenderData.

For more information about data sources and data transfer, see the article Data Objects and Data Sources (OLE). In addition, the article Clipboard Topics describes the OLE Clipboard mechanism.

Requirements

Header: afxole.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

HIERSVR Sample: Demonstrates a Server Application with OLE Drag and Drop

OCLIENT Sample: Illustrates a Visual Editing Container Application

Reference

CCmdTarget Class

Hierarchy Chart

COleDataObject Class

Other Resources

COleDataSource Members

COleDataSource Members

Base Class Members

CObject Members

CCmdTarget Members

Construction

COleDataSource	Constructs a **COleDataSource** object.

Operations

CacheData	Offers data in a specified format using a **STGMEDIUM** structure.
CacheGlobalData	Offers data in a specified format using an **HGLOBAL**.
DelayRenderData	Offers data in a specified format using delayed rendering.
DelayRenderFileData	Offers data in a specified format in a **CFile** pointer.
DelaySetData	Called for every format that is supported in **OnSetData**.
DoDragDrop	Performs drag-and-drop operations with a data source.
Empty	Empties the **COleDataSource** object of data.
FlushClipboard	Renders all data to the Clipboard.
GetClipboardOwner	Verifies that the data placed on the Clipboard is still there.
OnRenderData	Retrieves data as part of delayed rendering.
OnRenderFileData	Retrieves data into a **CFile** as part of delayed rendering.
OnRenderGlobalData	Retrieves data into an **HGLOBAL** as part of delayed rendering.
OnSetData	Called to replace the data in the **COleDataSource** object.
SetClipboard	Places a **COleDataSource** object on the Clipboard.

See Also

Reference

COleDataSource Class

Hierarchy Chart

COleDataSource Member Functions

For information about the member functions in **COleDataSource**, see COleDataSource Members.

COleDataSource::CacheData

Call this function to specify a format in which data is offered during data transfer operations.

```
void CacheData(
   CLIPFORMAT cfFormat,
   LPSTGMEDIUM lpStgMedium,
   LPFORMATETC lpFormatEtc = NULL
);
```

Parameters

cfFormat

The Clipboard format in which the data is to be offered. This parameter can be one of the predefined Clipboard formats or the value returned by the native Windows RegisterClipboardFormat function.

lpStgMedium

Points to a STGMEDIUM structure containing the data in the format specified.

lpFormatEtc

Points to a FORMATETC structure describing the format in which the data is to be offered. Provide a value for this parameter if you want to specify additional format information beyond the Clipboard format specified by *cfFormat*. If it is **NULL**, default values are used for the other fields in the **FORMATETC** structure.

Remarks

You must supply the data, because this function provides it by using immediate rendering. The data is cached until needed.

Supply the data using a STGMEDIUM structure. You can also use the **CacheGlobalData** member function if the amount of data you are supplying is small enough to be transferred efficiently using an **HGLOBAL**.

After the call to **CacheData** the **ptd** member of *lpFormatEtc* and the contents of *lpStgMedium* are owned by the data object, not by the caller.

To use delayed rendering, call the DelayRenderData or DelayRenderFileData member function. For more information on delayed rendering as handled by MFC, see the article Data Objects and Data Sources: Manipulation.

For more information, see the STGMEDIUM and FORMATETC structures in the *Platform SDK*.

For more information, see RegisterClipboardFormat in the *Platform SDK*.

See Also

Reference

COleDataSource Class
Hierarchy Chart
COleDataSource::CacheGlobalData
COleDataSource::DelayRenderData
COleDataSource::DelayRenderFileData
COleDataSource::SetClipboard
COleDataSource::DoDragDrop

Other Resources

COleDataSource Members

COleDataSource::CacheGlobalData

Call this function to specify a format in which data is offered during data transfer operations.

```
void CacheGlobalData(
   CLIPFORMAT cfFormat,
   HGLOBAL hGlobal,
   LPFORMATETC lpFormatEtc = NULL
);
```

Parameters

cfFormat

The Clipboard format in which the data is to be offered. This parameter can be one of the predefined Clipboard formats or the value returned by the native Windows RegisterClipboardFormat function.

hGlobal

Handle to the global memory block containing the data in the format specified.

lpFormatEtc

Points to a FORMATETC structure describing the format in which the data is to be offered. Provide a value for this parameter if you want to specify additional format information beyond the Clipboard format specified by *cfFormat*. If it is **NULL**, default values are used for the other fields in the **FORMATETC** structure.

Remarks

This function provides the data using immediate rendering, so you must supply the data when calling the function; the data is cached until needed. Use the **CacheData** member function if you are supplying a large amount of data or if you require a structured storage medium.

To use delayed rendering, call the DelayRenderData or DelayRenderFileData member function. For more information on delayed rendering as handled by MFC, see the article Data Objects and Data Sources: Manipulation.

For more information, see the FORMATETC structure in the *Platform SDK*.

For more information, see RegisterClipboardFormat in the *Platform SDK*.

See Also

Reference

COleDataSource Class
Hierarchy Chart
COleDataSource::CacheData
COleDataSource::DelayRenderData
COleDataSource::DelayRenderFileData

Other Resources

COleDataSource Members

COleDataSource::COleDataSource

Constructs a **COleDataSource** object.

```
COleDataSource( );
```

See Also

Reference

COleDataSource Class

Hierarchy Chart

Other Resources

COleDataSource Members

COleDataSource::DelayRenderData

Call this function to specify a format in which data is offered during data transfer operations.

```
void DelayRenderData(
   CLIPFORMAT cfFormat,
   LPFORMATETC lpFormatEtc = NULL
);
```

Parameters

cfFormat

The Clipboard format in which the data is to be offered. This parameter can be one of the predefined Clipboard formats or the value returned by the native Windows RegisterClipboardFormat function.

lpFormatEtc

Points to a FORMATETC structure describing the format in which the data is to be offered. Provide a value for this parameter if you want to specify additional format information beyond the Clipboard format specified by *cfFormat*. If it is **NULL**, default values are used for the other fields in the **FORMATETC** structure.

Remarks

This function provides the data using delayed rendering, so the data is not supplied immediately. The OnRenderData or OnRenderGlobalData member function is called to request the data.

Use this function if you are not going to supply your data through a **CFile** object. If you are going to supply the data through a **CFile** object, call the DelayRenderFileData member function. For more information on delayed rendering as handled by MFC, see the article Data Objects and Data Sources: Manipulation.

To use immediate rendering, call the CacheData or CacheGlobalData member function.

For more information, see the FORMATETC structure in the *Platform SDK*.

For more information, see RegisterClipboardFormat in the *Platform SDK*.

See Also

Reference

COleDataSource Class
Hierarchy Chart
COleDataSource::CacheData
COleDataSource::CacheGlobalData
COleDataSource::DelayRenderFileData
COleDataSource::OnRenderData
COleDataSource::OnRenderGlobalData

Other Resources

COleDataSource Members

COleDataSource::DelayRenderFileData

Call this function to specify a format in which data is offered during data transfer operations.

```
void DelayRenderFileData(
   CLIPFORMAT cfFormat,
   LPFORMATETC lpFormatEtc = NULL
);
```

Parameters

cfFormat

The Clipboard format in which the data is to be offered. This parameter can be one of the predefined Clipboard formats or the value returned by the native Windows RegisterClipboardFormat function.

lpFormatEtc

Points to a FORMATETC structure describing the format in which the data is to be offered. Provide a value for this parameter if you want to specify additional format information beyond the Clipboard format specified by *cfFormat*. If it is **NULL**, default values are used for the other fields in the **FORMATETC** structure.

Remarks

This function provides the data using delayed rendering, so the data is not supplied immediately. The OnRenderFileData member function is called to request the data.

Use this function if you are going to use a **CFile** object to supply the data. If you are not going to use a **CFile** object, call the DelayRenderData member function. For more information on delayed rendering as handled by MFC, see the article Data Objects and Data Sources: Manipulation.

To use immediate rendering, call the CacheData or CacheGlobalData member function.

For more information, see the FORMATETC structure in the *Platform SDK*.

For more information, see RegisterClipboardFormat in the *Platform SDK*.

See Also

Reference

COleDataSource Class
Hierarchy Chart
COleDataSource::CacheData
COleDataSource::CacheGlobalData
COleDataSource::DelayRenderData
COleDataSource::OnRenderFileData

Other Resources

COleDataSource Members

COleDataSource::DelaySetData

Call this function to support changing the contents of the data source.

```
void DelaySetData(
   CLIPFORMAT cfFormat,
   LPFORMATETC lpFormatEtc = NULL
);
```

Parameters

cfFormat

The Clipboard format in which the data is to be placed. This parameter can be one of the predefined Clipboard formats or the value returned by the native Windows RegisterClipboardFormat function.

lpFormatEtc

Points to a FORMATETC structure describing the format in which the data is to be replaced. Provide a value for this parameter if you want to specify additional format information beyond the Clipboard format specified by *cfFormat*. If it is **NULL**, default values are used for the other fields in the **FORMATETC** structure.

Remarks

OnSetData will be called by the framework when this happens. This is only used when the framework returns the data source from COleServerItem::GetDataSource. If **DelaySetData** is not called, your **OnSetData** function will never be called. **DelaySetData** should be called for each Clipboard or **FORMATETC** format you support.

For more information, see the FORMATETC structure in the *Platform SDK*.

For more information, see RegisterClipboardFormat in the *Platform SDK*.

See Also

Reference

COleDataSource Class
Hierarchy Chart
COleServerItem::GetDataSource
COleDataSource::OnSetData

Other Resources

COleDataSource Members

COleDataSource::DoDragDrop

Call the **DoDragDrop** member function to perform a drag-and-drop operation for this data source, typically in an CWnd::OnLButtonDown handler.

```
DROPEFFECT DoDragDrop(
   DWORD dwEffects = DROPEFFECT_COPY|DROPEFFECT_MOVE|DROPEFFECT_LINK,
   LPCRECT lpRectStartDrag = NULL,
   COleDropSource* pDropSource = NULL
);
```

Parameters

dwEffects

Drag-and-drop operations that are allowed on this data source. Can be one or more of the following:

- **DROPEFFECT_COPY** A copy operation could be performed.
- **DROPEFFECT_MOVE** A move operation could be performed.
- **DROPEFFECT_LINK** A link from the dropped data to the original data could be established.
- **DROPEFFECT_SCROLL** Indicates that a drag scroll operation could occur.

lpRectStartDrag

Pointer to the rectangle that defines where the drag actually starts. For more information, see the following Remarks section.

pDropSource

Points to a drop source. If **NULL** then a default implementation of COleDropSource will be used.

Return Value

Drop effect generated by the drag-and-drop operation; otherwise **DROPEFFECT_NONE** if the operation never begins because the user released the mouse button before leaving the supplied rectangle.

Remarks

The drag-and-drop operation does not start immediately. It waits until the mouse cursor leaves the rectangle specified by *lpRectStartDrag* or until a specified number of milliseconds have passed. If *lpRectStartDrag* is **NULL**, the size of the rectangle is one pixel.

The delay time is specified by a registry key setting. You can change the delay time by calling CWinApp::WriteProfileString or CWinApp::WriteProfileInt. If you do not specify the delay time, a default value of 200 milliseconds is used. Drag delay time is stored as follows:

- Windows NT Drag delay time is stored in HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\NT\CurrentVersion\IniFileMapping\win.ini\Windows\DragDelay.
- Windows 3.x Drag delay time is stored in the WIN.INI file, under the [Windows} section.
- Windows 95/98 Drag delay time is stored in a cached version of WIN.INI.

For more information about how drag delay information is stored in either the registry or the .INI file, see WriteProfileString in the *Platform SDK*.

For more information, see the article Drag and Drop: Implementing a Drop Source.

See Also

Reference

COleDataSource Class
Hierarchy Chart
COleDropSource::OnBeginDrag
COleDropSource Class

Other Resources

COleDataSource Members

COleDataSource::Empty

Call this function to empty the **COleDataSource** object of data.

```
void Empty( );
```

Remarks

Both cached and delay render formats are emptied so they can be reused.

For more information, see ReleaseStgMedium in the *Platform SDK*.

See Also

Reference

COleDataSource Class

Hierarchy Chart

Other Resources

COleDataSource Members

COleDataSource::FlushClipboard

Removes data from the Clipboard that was placed there by a previous call to SetClipboard.

```
static void PASCAL FlushClipboard( );
```

Remarks

This function also causes any data still on the Clipboard to be immediately rendered. Call this function when it is necessary to delete the data object last placed on the Clipboard from memory. Calling this function ensures that OLE will not require the original data source to perform Clipboard rendering.

See Also

Reference

COleDataSource Class
Hierarchy Chart
COleDataSource::GetClipboardOwner
COleDataSource::SetClipboard

Other Resources

COleDataSource Members

COleDataSource::GetClipboardOwner

Determines whether the data on the Clipboard has changed since SetClipboard was last called and, if so, identifies the current owner.

```
static COleDataSource* PASCAL GetClipboardOwner( );
```

Return Value

The data source currently on the Clipboard, or **NULL** if there is nothing on the Clipboard or if the Clipboard is not owned by the calling application.

See Also

Reference

COleDataSource Class
Hierarchy Chart
COleDataSource::FlushClipboard
COleDataSource::SetClipboard

Other Resources

COleDataSource Members

COleDataSource::OnRenderData

Called by the framework to retrieve data in the specified format.

```
virtual BOOL OnRenderData(
   LPFORMATETC lpFormatEtc,
   LPSTGMEDIUM lpStgMedium
);
```

Parameters

lpFormatEtc

Points to the FORMATETC structure specifying the format in which information is requested.

lpStgMedium

Points to a STGMEDIUM structure in which the data is to be returned.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The specified format is one previously placed in the **COleDataSource** object using the DelayRenderData or DelayRenderFileData member function for delayed rendering. The default implementation of this function will call OnRenderFileData or OnRenderGlobalData if the supplied storage medium is either a file or memory, respectively. If neither of these formats are supplied, then the default implementation will return 0 and do nothing. For more information on delayed rendering as handled by MFC, see the article Data Objects and Data Sources: Manipulation.

If *lpStgMedium->tymed* is **TYMED_NULL**, the **STGMEDIUM** should be allocated and filled as specified by *lpFormatEtc->tymed*. If it is not **TYMED_NULL**, the **STGMEDIUM** should be filled in place with the data.

This is an advanced overridable. Override this function to supply your data in the requested format and medium. Depending on your data, you may want to override one of the other versions of this function instead. If your data is small and fixed in size, override **OnRenderGlobalData**. If your data is in a file, or is of variable size, override **OnRenderFileData**.

For more information, see the STGMEDIUM and FORMATETC structures, the TYMED enumeration type, and IDataObject::GetData in the *Platform SDK*.

See Also

Reference

COleDataSource Class
Hierarchy Chart
COleDataSource::DelayRenderData
COleDataSource::DelayRenderFileData
COleDataSource::OnRenderFileData
COleDataSource::OnRenderGlobalData
COleDataSource::OnSetData

Other Resources

COleDataSource Members

COleDataSource::OnRenderFileData

Called by the framework to retrieve data in the specified format when the specified storage medium is a file.

```
virtual BOOL OnRenderFileData(
   LPFORMATETC lpFormatEtc,
   CFile* pFile
);
```

Parameters

lpFormatEtc

Points to the FORMATETC structure specifying the format in which information is requested.

pFile

Points to a CFile object in which the data is to be rendered.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The specified format is one previously placed in the **COleDataSource** object using the DelayRenderData member function for delayed rendering. The default implementation of this function simply returns **FALSE**.

This is an advanced overridable. Override this function to supply your data in the requested format and medium. Depending on your data, you might want to override one of the other versions of this function instead. If you want to handle multiple storage media, override OnRenderData. If your data is in a file, or is of variable size, override **OnRenderFileData**. For more information on delayed rendering as handled by MFC, see the article Data Objects and Data Sources: Manipulation.

For more information, see the FORMATETC structure and IDataObject::GetData in the *Platform SDK*.

See Also

Reference

COleDataSource Class
Hierarchy Chart
COleDataSource::DelayRenderData
COleDataSource::DelayRenderFileData
COleDataSource::OnRenderData
COleDataSource::OnRenderGlobalData
COleDataSource::OnSetData
CFile Class

Other Resources

COleDataSource Members

COleDataSource::OnRenderGlobalData

Called by the framework to retrieve data in the specified format when the specified storage medium is global memory.

```
virtual BOOL OnRenderGlobalData(
   LPFORMATETC lpFormatEtc,
   HGLOBAL* phGlobal
);
```

Parameters

lpFormatEtc

Points to the FORMATETC structure specifying the format in which information is requested.

phGlobal

Points to a handle to global memory in which the data is to be returned. If one has not yet been allocated, this parameter can be **NULL**.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The specified format is one previously placed in the **COleDataSource** object using the DelayRenderData member function for delayed rendering. The default implementation of this function simply returns **FALSE**.

If *phGlobal* is **NULL**, then a new **HGLOBAL** should be allocated and returned in *phGlobal*. Otherwise, the **HGLOBAL** specified by *phGlobal* should be filled with the data. The amount of data placed in the **HGLOBAL** must not exceed the current size of the memory block. Also, the block cannot be reallocated to a larger size.

This is an advanced overridable. Override this function to supply your data in the requested format and medium. Depending on your data, you may want to override one of the other versions of this function instead. If you want to handle multiple storage media, override OnRenderData. If your data is in a file, or is of variable size, override OnRenderFileData. For more information on delayed rendering as handled by MFC, see the article Data Objects and Data Sources: Manipulation.

For more information, see the FORMATETC structure and IDataObject::GetData in the *Platform SDK*.

See Also

Reference

COleDataSource Class
Hierarchy Chart
COleDataSource::DelayRenderData
COleDataSource::DelayRenderFileData
COleDataSource::OnRenderData
COleDataSource::OnRenderFileData
COleDataSource::OnSetData

Other Resources

COleDataSource Members

COleDataSource::OnSetData

Called by the framework to set or replace the data in the **COleDataSource** object in the specified format.

```
virtual BOOL OnSetData(
   LPFORMATETC lpFormatEtc,
   LPSTGMEDIUM lpStgMedium,
   BOOL bRelease
);
```

Parameters

lpFormatEtc

Points to the FORMATETC structure specifying the format in which data is being replaced.

lpStgMedium

Points to the STGMEDIUM structure containing the data that will replace the current contents of the **COleDataSource** object.

bRelease

Indicates who has ownership of the storage medium after completing the function call. The caller decides who is responsible for releasing the resources allocated on behalf of the storage medium. The caller does this by setting *bRelease*. If *bRelease* is nonzero, the data source takes ownership, freeing the medium when it has finished using it. When *bRelease* is 0, the caller retains ownership and the data source can use the storage medium only for the duration of the call.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The data source does not take ownership of the data until it has successfully obtained it. That is, it does not take ownership if **OnSetData** returns 0. If the data source takes ownership, it frees the storage medium by calling the ReleaseStgMedium function.

The default implementation does nothing. Override this function to replace the data in the specified format. This is an advanced overridable.

For more information, see the STGMEDIUM and FORMATETC structures and the ReleaseStgMedium and IDataObject::GetData functions in the *Platform SDK*.

See Also

Reference

COleDataSource Class
Hierarchy Chart
COleDataSource::DelaySetData
COleDataSource::OnRenderData
COleDataSource::OnRenderFileData
COleDataSource::OnRenderGlobalData
COleServerItem::OnSetData

Other Resources

COleDataSource Members

COleDataSource::SetClipboard

Puts the data contained in the **COleDataSource** object on the Clipboard after calling one of the following functions: CacheData, CacheGlobalData, DelayRenderData, or DelayRenderFileData.

```
void SetClipboard( );
```

See Also

Reference

COleDataSource Class
Hierarchy Chart
COleDataSource::GetClipboardOwner
COleDataSource::FlushClipboard

Other Resources

COleDataSource Members

COleDBRecordView Class

A view that displays database records in controls.

```
class COleDBRecordView : public CFormView
```

Remarks

The view is a form view directly connected to a **CRowset** object. The view is created from a dialog template resource and displays the fields of the **CRowset** object in the dialog template's controls. The **COleDBRecordView** object uses dialog data exchange (DDX), and the navigational functionality built into **CRowset**, to automate the movement of data between the controls on the form and the fields of the rowset. **COleDBRecordView** also supplies a default implementation for moving to the first, next, previous, or last record and an interface for updating the record currently on view.

You can use DDX functions with **COleDbRecordView** to get data directly from the database recordset and display it in a dialog control. You should use the **DDX_*** methods (such as **DDX_Text**), not the **DDX_Field*** functions (such as **DDX_FieldText**) with **COleDbRecordView**. **DDX_FieldText** will not work with **COleDbRecordView** because **DDX_FieldText** takes an additional argument of type **CRecordset*** (for **CRecordView**) or **CDaoRecordset*** (for **CDaoRecordView**).

Note
If you are working with the Data Access Objects (DAO) classes rather than the OLE DB Consumer Template classes, use class CDaoRecordView instead. For more information, see the article Overview: Database Programming.

COleDBRecordView keeps track of the user's position in the rowset so that the record view can update the user interface. When the user moves to either end of the rowset, the record view disables user interface objects — such as menu items or toolbar buttons — for moving further in the same direction.

For more information about rowset classes, see the Using OLE DB Consumer Templates article.

Requirements

Header: afxoledb.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart

Other Resources

COleDBRecordView Members

COleDBRecordView Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CView Members

CScrollView Members

CFormView Members

Construction

COleDBRecordView	Constructs a **COleDBRecordView** object.

Attributes

OnGetRowset	Returns a standard **HRESULT** value.

Operations

OnMove	Updates the current record (if dirty) on the data source and then moves to the specified record (next, previous, first, or last).

See Also

Reference

COleDBRecordView Class

Hierarchy Chart

COleDBRecordView Member Functions

For information about the member functions in **COleDBRecordView**, see COleDBRecordView Members.

COleDBRecordView::COleDBRecordView

Constructs a **COleDBRecordView** object.

```
COleDBRecordView(
   LPCTSTR lpszTemplateName
);
COleDBRecordView(
   UINT nIDTemplate
);
```

Parameters

lpszTemplateName

Contains a null-terminated string that is the name of a dialog-template resource.

nIDTemplate

Contains the ID number of a dialog-template resource.

Remarks

When you create an object of a type derived from **COleDBRecordView**, invoke one of the constructors to create the view object and identify the dialog resource on which the view is based. You can identify the resource either by name (pass a string as the argument to the constructor) or by its ID (pass an unsigned integer as the argument).

Note

Your derived class *must* supply its own constructor. In the constructor, invoke the constructor, **COleDBRecordView::COleDBRecordView**, with the resource name or ID as an argument.

See Also

Reference

COleDBRecordView Class

Hierarchy Chart

Other Resources

COleDBRecordView Members

COleDBRecordView::OnGetRowset

Returns a handle for the **CRowset<>** object associated with the record view.

```
virtual CRowset< >* OnGetRowset(
   ) = 0;
```

Return Value

A standard **HRESULT** value.

Remarks

You must override this member function to construct or obtain a rowset object and return a handle to it. If you declare your record view class with ClassWizard, the wizard writes a default override for you. ClassWizard's default implementation returns the rowset handle stored in the record view if one exists. If not, it constructs a rowset object of the type you specified with ClassWizard and calls its **Open** member function to open the table or run the query, and then returns a handle to the object.

Note

Previous to MFC 7.0, **OnGetRowset** returned a pointer to **CRowset**. If you have code that calls **OnGetRowset**, you need to c hange the return type to the templatized class **CRowset<>**.

Example

```
COleDBRecordView myRecordView;
...
// CProductAccessor is a user-defined accessor class
CRowset<CAccessor<CProductAccessor>> myRowSet = myRecordView.OnGetRowset();
```

For more information and examples, see the article Record Views: Using a Record View.

See Also

Reference

COleDBRecordView Class

Hierarchy Chart

Other Resources

COleDBRecordView Members

COleDBRecordView::OnMove

Moves to a different record in the rowset and display its fields in the controls of the record view.

```
virtual BOOL OnMove(
   UINT nIDMoveCommand
);
```

Parameters

nIDMoveCommand

One of the following standard command ID values:

- **ID_RECORD_FIRST** Move to the first record in the recordset.
- **ID_RECORD_LAST** Move to the last record in the recordset.
- **ID_RECORD_NEXT** Move to the next record in the recordset.
- **ID_RECORD_PREV** Move to the previous record in the recordset.

Return Value

Nonzero if the move was successful; otherwise 0 if the move request was denied.

Remarks

The default implementation calls the appropriate **Move** member function of the **CRowset** object associated with the record view.

By default, **OnMove** updates the current record on the data source if the user has changed it in the record view.

The Application Wizard creates a menu resource with First Record, Last Record, Next Record, and Previous Record menu items. If you select the Dockable Toolbar option, The Application Wizard also creates a toolbar with buttons corresponding to these commands.

If you move past the last record in the recordset, the record view continues to display the last record. If you move backward past the first record, the record view continues to display the first record.

See Also

Reference

COleDBRecordView Class

Hierarchy Chart

Other Resources

COleDBRecordView Members

COleDialog Class

Provides functionality common to dialog boxes for OLE.

```
class COleDialog : public CCommonDialog
```

Remarks

The Microsoft Foundation Class Library provides several classes derived from **COleDialog**:

- COleInsertDialog
- COleConvertDialog
- COleChangeIconDialog
- COleLinksDialog
- COleBusyDialog
- COleUpdateDialog
- COlePasteSpecialDialog
- COlePropertiesDialog
- COleChangeSourceDialog

For more information about OLE-specific dialog boxes, see the article Dialog Boxes in OLE.

Requirements

Header: afxodlgs.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CCommonDialog Class

Hierarchy Chart

Other Resources

COleDialog Members

COleDialog Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CDialog Members

CCommonDialog Members

Operations

GetLastError	Gets the error code returned by the dialog box.

See Also

Reference

COleDialog Class

Hierarchy Chart

COleDialog Member Functions

For information about the member functions in **COleDialog**, see COleDialog Members.

COleDialog::GetLastError

Call the **GetLastError** member function to get additional error information when **DoModal** returns **IDABORT**.

```
UINT GetLastError( ) const;
```

Return Value

The error codes returned by **GetLastError** depend on the specific dialog box displayed.

Remarks

See the **DoModal** member function in the derived classes for information about specific error messages.

See Also

Reference

COleDialog Class
Hierarchy Chart
COleBusyDialog::DoModal
COleChangeIconDialog::DoModal
COleChangeSourceDialog::DoModal
COleConvertDialog::DoModal
COleInsertDialog::DoModal
COleLinksDialog::DoModal
COlePasteSpecialDialog::DoModal
COlePropertiesDialog::DoModal
COleUpdateDialog::DoModal

Other Resources

COleDialog Members

COleDispatchDriver Class

Implements the client side of OLE automation.

```
class COleDispatchDriver
```

Remarks

COleDispatchDriver does not have a base class.

OLE dispatch interfaces provide access to an object's methods and properties. Member functions of **COleDispatchDriver** attach, detach, create, and release a dispatch connection of type **IDispatch**. Other member functions use variable argument lists to simplify calling **IDispatch::Invoke**.

This class can be used directly, but it is generally used only by classes created by the Add Class wizard. When you create new C++ classes by importing a type library, the new classes are derived from **COleDispatchDriver**.

For more information on using **COleDispatchDriver**, see the following articles:

- Automation Clients
- Automation Servers

Requirements

Header: afxdisp.h

See Also

Tasks

CALCDRIV Sample: Demonstrates an Automation Client Application
ACDUAL Sample: Adds Dual Interfaces to an Automation Application

Reference

Hierarchy Chart
CCmdTarget Class

Other Resources

COleDispatchDriver Members

COleDispatchDriver Members

Data Members

m_bAutoRelease	Specifies whether to release the **IDispatch** during **ReleaseDispatch** or object destruction.
m_lpDispatch	Indicates the pointer to the **IDispatch** interface attached to this **COleDispatchDriver**.

Construction

COleDispatchDriver	Constructs a **COleDispatchDriver** object.

Operations

AttachDispatch	Attaches an **IDispatch** connection to the **COleDispatchDriver** object.
CreateDispatch	Creates an **IDispatch** connection and attaches it to the **COleDispatchDriver** object.
DetachDispatch	Detaches an **IDispatch** connection, without releasing it.
GetProperty	Gets an automation property.
InvokeHelper	Helper for calling automation methods.
ReleaseDispatch	Releases an **IDispatch** connection.
SetProperty	Sets an automation property.

Operators

operator =	Copies the source value into the **COleDispatchDriver** object.
operator LPDISPATCH	Accesses the underlying **IDispatch** pointer.

See Also

Reference

COleDispatchDriver Class

Hierarchy Chart

COleDispatchDriver Member Functions

For information about the member functions in **COleDispatchDriver**, see COleDispatchDriver Members.

COleDispatchDriver::AttachDispatch

Call the **AttachDispatch** member function to attach an **IDispatch** pointer to the **COleDispatchDriver** object. For more information, see Implementing the IDispatch Interface.

```
void AttachDispatch(
   LPDISPATCH lpDispatch,
   BOOL bAutoRelease = TRUE
);
```

Parameters

lpDispatch

Pointer to an OLE **IDispatch** object to be attached to the **COleDispatchDriver** object.

bAutoRelease

Specifies whether the dispatch is to be released when this object goes out of scope.

Remarks

This function releases any **IDispatch** pointer that is already attached to the **COleDispatchDriver** object.

Example

```
// VC++ Client for the VJ++ ComCallingJava sample

// (33B0ECE2-E706-11CF-A0C2-00AA00A71DD8)
static const CLSID GCDclsid =
   { 0x33B0ECE2, 0xE706, 0x11CF, { 0xa0, 0xC2, 0x0, 0xaa, 0x0,
   0xa7, 0x1d, 0xd8 }};

// (33B0ECE1-E706-11cf-A0C2-00AA00A71DD8)
static const CLSID IGCDclsid =
   { 0x33B0ECE1, 0xE706, 0x11CF, { 0xa0, 0xC2, 0x0, 0xaa, 0x0,
   0xa7, 0x1d, 0xd8 }};

void CComjavaView::OnLButtonDown(UINT nFlags, CPoint point)
{
   CString str;
   LPDISPATCH pDisp;

   IGCD VJCom;   // wizard generated COleDispatchDriver
               // derived class created from the Euclid typelibrary.
   VJCom.CreateDispatch(GCDclsid);

   pDisp = VJCom.m_lpDispatch;
   HRESULT hr = pDisp->QueryInterface( IGCDclsid,
      (void**)&VJCom.m_lpDispatch);
   if (hr == S_OK)
      pDisp->Release();
   else
      VJCom.AttachDispatch(pDisp, TRUE);

   // Call the wizard-generated helper function GCD. This
   // function will in turn call COleDispatchDriver::InvokeHelper.
   str.Format("The least common denominator of 40 and 24 is %d",
      VJCom.GCD(40,24));
   AfxMessageBox(str);
   VJCom.ReleaseDispatch();

   CView::OnLButtonDown(nFlags, point);
}
```

See Also

Reference

COleDispatchDriver Class
Hierarchy Chart
COleDispatchDriver::DetachDispatch
COleDispatchDriver::ReleaseDispatch
COleDispatchDriver::CreateDispatch
COleDispatchDriver::m_lpDispatch
COleDispatchDriver::m_bAutoRelease

Other Resources

COleDispatchDriver Members

COleDispatchDriver::COleDispatchDriver

Constructs a **COleDispatchDriver** object.

```
COleDispatchDriver( );
COleDispatchDriver(
      LPDISPATCH lpDispatch,
      BOOL bAutoRelease = TRUE
);
COleDispatchDriver(
      const COleDispatchDriver& dispatchSrc
);
```

Parameters

lpDispatch

Pointer to an OLE **IDispatch** object to be attached to the **COleDispatchDriver** object.

bAutoRelease

Specifies whether the dispatch is to be released when this object goes out of scope.

dispatchSrc

Reference to an existing **COleDispatchDriver** object.

Remarks

The form **COleDispatchDriver**(**LPDISPATCH** *lpDispatch*, **BOOL** *bAutoRelease* = **TRUE**) connects the IDispatch interface.

The form **COleDispatchDriver**(**const COleDispatchDriver**& *dispatchSrc*) copies an existing **COleDispatchDriver** object and increments the reference count.

The form **COleDispatchDriver**( ) creates a **COleDispatchDriver** object but does not connect the **IDispatch** interface. Before using **COleDispatchDriver**( ) without arguments, you should connect an **IDispatch** to it using either COleDispatchDriver::CreateDispatch or COleDispatchDriver::AttachDispatch. For more information, see Implementing the IDispatch Interface.

Example

See the example for COleDispatchDriver::CreateDispatch.

See Also

Reference

COleDispatchDriver Class
Hierarchy Chart
COleDispatchDriver::AttachDispatch
COleDispatchDriver::CreateDispatch

Other Resources

COleDispatchDriver Members

COleDispatchDriver::CreateDispatch

Creates an IDispatch interface object and attaches it to the **COleDispatchDriver** object.

```
BOOL CreateDispatch(
   REFCLSID clsid,
   COleException* pError = NULL
);
BOOL CreateDispatch(
   LPCTSTR lpszProgID,
   COleException* pError = NULL
);
```

Parameters

clsid

Class ID of the **IDispatch** connection object to be created.

pError

Pointer to an OLE exception object, which will hold the status code resulting from the creation.

lpszProgID

Pointer to the programmatic identifier, such as "Excel.Document.5", of the automation object for which the dispatch object is to be created.

Return Value

Nonzero on success; otherwise 0.

Example

```
void CTestView::OnBatterylife()
{
   COleDispatchDriver disp;
   COleException *e = new COleException;

   try {
      // Create instance of Microsoft System Information Control
      // by using ProgID.
      if (disp.CreateDispatch("SYSINFO.Sysinfo.1", e))
      {
         //Call BatteryLifePercent.
         short nBatteryLifePercent;

         disp.InvokeHelper(0x03, DISPATCH_PROPERTYGET, VT_I2,
            (void*)&nBatteryLifePercent, NULL);

         if (nBatteryLifePercent == 255)
            AfxMessageBox("Battery Life % unknown");
         else
         {
            CString cStr;
            cStr.Format("Battery Life is at %d%%",
                nBatteryLifePercent);
            AfxMessageBox(cStr);
         }
      }
      else
         throw e;

   }

   //Catch control-specific exceptions.
    catch (COleDispatchException * e)
```

```
    {
       CString cStr;

       if (!e->m_strSource.IsEmpty())
          cStr = e->m_strSource + " - ";
       if (!e->m_strDescription.IsEmpty())
          cStr += e->m_strDescription;
       else
          cStr += "unknown error";

       AfxMessageBox(cStr, MB_OK,
          (e->m_strHelpFile.IsEmpty())? 0:e->m_dwHelpContext);

       e->Delete();
    }
    //Catch all MFC exceptions, including COleExceptions.
    // OS exceptions will not be caught.
     catch (CException *e)
    {
       TRACE("%s(%d): OLE Execption caught: SCODE = %x",
          __FILE__, __LINE__, COleException::Process(e));
       e->Delete();
    }
 }
```

See Also

Reference

COleDispatchDriver Class
Hierarchy Chart
COleDispatchDriver::DetachDispatch
COleDispatchDriver::ReleaseDispatch
COleDispatchDriver::AttachDispatch
COleException Class
COleDispatchDriver::m_lpDispatch

Other Resources

COleDispatchDriver Members

COleDispatchDriver::DetachDispatch

Detaches the current **IDispatch** connection from this object.

```
LPDISPATCH DetachDispatch( );
```

Return Value

A pointer to the previously attached OLE **IDispatch** object.

Remarks

The **IDispatch** is not released.

For more information about the **LPDISPATCH** type, see Implementing the IDispatch Interface in the *Platform SDK*.

Example

```
LPDISPATCH CreateLPDispatch(LPCTSTR lpszProgId)
{
    COleDispatchDriver disp;

    disp.CreateDispatch(lpszProgId);

    return disp.DetachDispatch();
}
```

See Also

Reference

COleDispatchDriver Class
Hierarchy Chart
COleDispatchDriver::ReleaseDispatch
COleDispatchDriver::CreateDispatch
COleDispatchDriver::AttachDispatch
COleDispatchDriver::m_lpDispatch

Other Resources

COleDispatchDriver Members

COleDispatchDriver::GetProperty

Gets the object property specified by *dwDispID*.

```
void GetProperty(
   DISPID dwDispID,
   VARTYPE vtProp,
   void* pvProp
) const;
```

Parameters

dwDispID

Identifies the property to be retrieved.

vtProp

Specifies the property to be retrieved. For possible values, see the Remarks section for COleDispatchDriver::InvokeHelper.

pvProp

Address of the variable that will receive the property value. It must match the type specified by *vtProp*.

Example

```
CString IMyComObject::GetString()
{
    CString result;
    GetProperty(0x1, VT_BSTR, (void*)&result);
    return result;
}
```

See Also

Reference

COleDispatchDriver Class
Hierarchy Chart
COleDispatchDriver::InvokeHelper
COleDispatchDriver::SetProperty

Other Resources

COleDispatchDriver Members

COleDispatchDriver::InvokeHelper

Calls the object method or property specified by *dwDispID*, in the context specified by *wFlags*.

```
void AFX_CDECL InvokeHelper(
   DISPID dwDispID,
   WORD wFlags,
   VARTYPE vtRet,
   void* pvRet,
   const BYTE* pbParamInfo,
      ...
);
```

Parameters

dwDispID

Identifies the method or property to be invoked.

wFlags

Flags describing the context of the call to **IDispatch::Invoke**. . For a list of possible values, see the *wFlags* parameter in IDispatch::Invoke in the *Platform SDK*.

vtRet

Specifies the type of the return value. For possible values, see the Remarks section.

pvRet

Address of the variable that will receive the property value or return value. It must match the type specified by *vtRet*.

pbParamInfo

Pointer to a null-terminated string of bytes specifying the types of the parameters following *pbParamInfo*.

...

Variable list of parameters, of types specified in *pbParamInfo*.

Remarks

The *pbParamInfo* parameter specifies the types of the parameters passed to the method or property. The variable list of arguments is represented by **...** in the syntax declaration.

Possible values for the *vtRet* argument are taken from the **VARENUM** enumeration. Possible values are as follows:

Symbol	Return Type
VT_EMPTY	**void**
VT_I2	**short**
VT_I4	**long**
VT_R4	**float**
VT_R8	**double**
VT_CY	**CY**
VT_DATE	**DATE**
VT_BSTR	**BSTR**

VT_DISPATCH	**LPDISPATCH**
VT_ERROR	**SCODE**
VT_BOOL	**BOOL**
VT_VARIANT	**VARIANT**
VT_UNKNOWN	**LPUNKNOWN**

The *pbParamInfo* argument is a space-separated list of **VTS_** constants. One or more of these values, separated by spaces (not commas), specifies the function's parameter list. Possible values are listed with the EVENT_CUSTOM macro.

This function converts the parameters to **VARIANTARG** values, then invokes the IDispatch::Invoke method. If the call to **Invoke** fails, this function will throw an exception. If the **SCODE** (status code) returned by **IDispatch::Invoke** is **DISP_E_EXCEPTION**, this function throws a COleException object; otherwise it throws a COleDispatchException.

For more information, see VARIANTARG, Implementing the IDispatch Interface, IDispatch::Invoke, and Structure of COM Error Codes in the *Platform SDK*.

Example

See the example for COleDispatchDriver::CreateDispatch.

See Also

Reference

COleDispatchDriver Class
Hierarchy Chart
COleException Class
COleDispatchException Class

Other Resources

COleDispatchDriver Members

COleDispatchDriver::ReleaseDispatch

Releases the **IDispatch** connection. For more information, see Implementing the IDispatch Interface

```
void ReleaseDispatch( );
```

Remarks

If auto release has been set for this connection, this function calls **IDispatch::Release** before releasing the interface.

Example

See the example for COleDispatchDriver::AttachDispatch.

See Also

Reference

COleDispatchDriver Class
Hierarchy Chart
COleDispatchDriver::DetachDispatch
COleDispatchDriver::CreateDispatch
COleDispatchDriver::AttachDispatch
COleDispatchDriver::m_lpDispatch
COleDispatchDriver::m_bAutoRelease

Other Resources

COleDispatchDriver Members

COleDispatchDriver::SetProperty

Sets the OLE object property specified by *dwDispID*.

```
void AFX_CDECL SetProperty(
   DISPID dwDispID,
   VARTYPE vtProp,
      ...
);
```

Parameters

dwDispID

Identifies the property to be set.

vtProp

Specifies the type of the property to be set. For possible values, see the Remarks section for COleDispatchDriver::InvokeHelper.

...

A single parameter of the type specified by *vtProp*.

Example

```
void IMyComObject::SetString(LPCTSTR propVal)
{
    SetProperty(0x1, VT_BSTR, propVal);
}
```

See Also

Reference

COleDispatchDriver Class
Hierarchy Chart
COleDispatchDriver::InvokeHelper
COleDispatchDriver::GetProperty

Other Resources

COleDispatchDriver Members

COleDispatchDriver Operators

For information about the operators in **COleDispatchDriver**, see COleDispatchDriver Members.

COleDispatchDriver::operator =

Copies the source value into the **COleDispatchDriver** object.

```
const COleDispatchDriver& operator=(
   const COleDispatchDriver& dispatchSrc
);
```

Parameters

dispatchSrc

A pointer to an existing **COleDispatchDriver** object.

See Also

Reference

COleDispatchDriver Class

Hierarchy Chart

Other Resources

COleDispatchDriver Members

COleDispatchDriver::operator LPDISPATCH

Accesses the underlying **IDispatch** pointer of the **COleDispatchDriver** object.

```
operator LPDISPATCH( );
```

Example

```
COleDispatchDriver disp;
if (disp.CreateDispatch("SAFRCFileDlg.FileOpen"))
{
   IDispatch* pDispatch=disp; //COleDispatchDriver::operator
                              //LPDISPATCH is called here
   IUnknown* pUnkn=NULL;
   HRESULT hr=pDispatch->QueryInterface(
      IID_IUnknown ,(void**)&pUnkn);
   if ( SUCCEEDED(hr) )
   {
      //Do something...
      pUnkn->Release();
   }
}
```

See Also

Reference

COleDispatchDriver Class

Hierarchy Chart

COleDispatchDriver::m_lpDispatch

Other Resources

COleDispatchDriver Members

COleDispatchDriver Data Members

For information about the data members in **COleDispatchDriver**, see COleDispatchDriver Members.

COleDispatchDriver::m_bAutoRelease

If **TRUE**, the COM object accessed by m_lpDispatch will be automatically released when ReleaseDispatch is called or when this **COleDispatchDriver** object is destroyed.

```
BOOL m_bAutoRelease;
```

Remarks

By default, **m_bAutoRelease** is set to **TRUE** in the constructor.

For more information on releasing COM objects, see Implementing Reference Counting and IUnknown::Release in the *Platform SDK*.

Example

```
// Clean up by forcing Release to be called
// on COleDispatchDriver object and delete
if (bError)
{
   pDisp.m_bAutoRelease = TRUE;
   delete pDisp;
   pDisp = NULL;
}
```

See Also

Reference

COleDispatchDriver Class
Hierarchy Chart
COleDispatchDriver::AttachDispatch
COleDispatchDriver::ReleaseDispatch
COleDispatchDriver::m_lpDispatch

Other Resources

COleDispatchDriver Members

COleDispatchDriver::m_lpDispatch

The pointer to the **IDispatch** interface attached to this **COleDispatchDriver**. For more information, see Implementing the IDispatch Interface.

```
LPDISPATCH m_lpDispatch;
```

Remarks

The **m_lpDispatch** data member is a public variable of type **LPDISPATCH**.

For more information, see IDispatch in the *Platform SDK*.

Example

See the example for COleDispatchDriver::AttachDispatch.

See Also

Reference

COleDispatchDriver Class
Hierarchy Chart
COleDispatchDriver::AttachDispatch
COleDispatchDriver::ReleaseDispatch
COleDispatchDriver::CreateDispatch
COleDispatchDriver::DetachDispatch

Other Resources

COleDispatchDriver Members

COleDispatchException Class

Handles exceptions specific to the OLE **IDispatch** interface, which is a key part of OLE automation.

```
class COleDispatchException : public CException
```

Remarks

Like the other exception classes derived from the **CException** base class, **COleDispatchException** can be used with the **THROW**, **THROW_LAST**, **TRY**, **CATCH**, **AND_CATCH**, and **END_CATCH** macros.

In general, you should call AfxThrowOleDispatchException to create and throw a **COleDispatchException** object.

For more information on exceptions, see the articles Exception Handling (MFC) and Exceptions: OLE Exceptions.

Requirements

Header: afxdisp.h

See Also

Tasks

CALCDRIV Sample: Demonstrates an Automation Client Application

Reference

CException Class

Hierarchy Chart

COleDispatchDriver Class

COleException Class

Other Resources

COleDispatchException Members

COleDispatchException Members

Base Class Members

CObject Members

CException Members

Data Members

m_dwHelpContext	Help context for error.
m_strDescription	Verbal error description.
m_strHelpFile	Help file to use with **m_dwHelpContext**.
m_strSource	Application that generated the exception.
m_wCode	**IDispatch**-specific error code.

See Also

Reference

COleDispatchException Class

Hierarchy Chart

COleDispatchException Data Members

For information about the data members in **COleDispatchException**, see COleDispatchException Members.

COleDispatchException::m_dwHelpContext

Identifies a help context in your application's help (.HLP) file.

```
DWORD m_dwHelpContext;
```

Remarks

This member is set by the function AfxThrowOleDispatchException when an exception is thrown.

Example

See the example for COleDispatchDriver::CreateDispatch.

See Also

Reference

COleDispatchException Class
Hierarchy Chart
COleDispatchException::m_strDescription
COleDispatchException::m_wCode
AfxThrowOleDispatchException

Other Resources

COleDispatchException Members

COleDispatchException::m_strDescription

Contains a verbal error description, such as "Disk full."

```
CString m_strDescription;
```

Remarks

This member is set by the function AfxThrowOleDispatchException when an exception is thrown.

Example

See the example for COleDispatchDriver::CreateDispatch.

See Also

Reference

COleDispatchException Class
Hierarchy Chart
COleDispatchException::m_dwHelpContext
COleDispatchException::m_wCode
AfxThrowOleDispatchException

Other Resources

COleDispatchException Members

COleDispatchException::m_strHelpFile

The framework fills in this string with the name of the application's help file.

```
CString m_strHelpFile;
```

See Also

Reference

COleDispatchException Class

Hierarchy Chart

AfxThrowOleDispatchException

Other Resources

COleDispatchException Members

COleDispatchException::m_strSource

The framework fills in this string with the name of the application that generated the exception.

```
CString m_strSource;
```

Example

See the example for COleDispatchDriver::CreateDispatch.

See Also

Reference

COleDispatchException Class
Hierarchy Chart
AfxThrowOleDispatchException

Other Resources

COleDispatchException Members

COleDispatchException::m_wCode

Contains an error code specific to your application.

```
WORD m_wCode;
```

Remarks

This member is set by the function AfxThrowOleDispatchException when an exception is thrown.

See Also

Reference

COleDispatchException Class
Hierarchy Chart
COleDispatchException::m_strDescription
COleDispatchException::m_dwHelpContext
AfxThrowOleDispatchException

Other Resources

COleDispatchException Members

COleDocObjectItem Class

Implements Active document containment.

```
class COleDocObjectItem : public COleClientItem
```

Remarks

In MFC, an Active document is handled similarly to a regular, in-place editable embedding, with the following differences:

- The **COleDocument**-derived class still maintains a list of the currently embedded items; however, these items may be **COleDocObjectItem**-derived items.
- When an active document is active, it occupies the entire client area of the view when it is in-place active.
- An Active document container has full control of the **Help** menu.
- The **Help** menu contains menu items for both the Active document container and server.

Because the Active document container owns the **Help** menu, the container is responsible for forwarding server **Help** menu messages to the server. This integration is handled by **COleDocObjectItem**.

For more information on menu merging and Active document activation, see Overview of Active Document Containment.

Requirements

Header: afxole.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

MFCBIND Sample: Active Document Container

Reference

COleClientItem Class

Hierarchy Chart

COleClientItem Class

CDocObjectServerItem Class

Other Resources

COleDocObjectItem Members

COleDocObjectItem Members

Base Class Members

CObject Members

CCmdTarget Members

CDocItem Members

COleClientItem Members

Constructors

COleDocObjectItem	Constructs a **COleDocObject** item.

Operations

DoDefaultPrinting	Prints the container application's document using the default printer settings.
ExecCommand	Executes the command specified by the user.
GetActiveView	Retrieves the document's active view.
GetPageCount	Retrieves the number of pages in the container application's document.
OnPreparePrinting	Prepares the container application's document for printing.
OnPrint	Prints the container application's document.
QueryCommand	Queries for the status of one or more commands generated by user interface events.
Release	Releases the connection to an OLE linked item and closes it if it was open. Does not destroy the client item.

See Also

Reference

COleDocObjectItem Class

Hierarchy Chart

COleDocObjectItem Member Functions

For information about the member functions in **COleDocObjectItem**, see COleDocObjectItem Members.

COleDocObjectItem::COleDocObjectItem

Call this member function to initialize the **COleDocObjectItem** object.

```
COleDocObjectItem(
   COleDocument* pContainerDoc = NULL
);
```

Parameters

pContainerDoc

A pointer to the **COleDocument** object acting as the active document container. This parameter must be **NULL** to enable **IMPLEMENT_SERIALIZE**. Normally OLE items are constructed with a non-**NULL** document pointer.

See Also

Reference

COleDocObjectItem Class

Hierarchy Chart

Other Resources

COleDocObjectItem Members

COleDocObjectItem::DoDefaultPrinting

Called by the framework to a document using the default settings.

```
static HRESULT DoDefaultPrinting(
   CView *pCaller,
   CPrintInfo *pInfo
);
```

Parameters

pCaller

A pointer to a CView object that is sending the print command.

pInfo

A pointer to a CPrintInfo object that describes the job to be printed.

See Also

Reference

COleDocObjectItem Class

Hierarchy Chart

Other Resources

COleDocObjectItem Members

COleDocObjectItem::ExecCommand

Call this member function to execute the command specified by the user.

```
HRESULT ExecCommand(
   DWORD nCmdID,
   DWORD nCmdExecOpt = OLECMDEXECOPT_DONTPROMPTUSER,
   const GUID* pguidCmdGroup = NULL
);
```

Parameters

nCmdID

The identifier of the command to execute. Must be in the group identified by *pguidCmdGroup*.

nCmdExecOpt

Specifies command-execution options. By default, set to execute the command without prompting the user. See OLECMDEXECOPT for a list of values.

pguidCmdGroup

Unique identifier of the command group. By default, **NULL**, which specifies the standard group. The command passed in *nCmdID* must belong to the group.

Return Value

Returns **S_OK** if successful; otherwise,returns one of the following error codes.

Value	Description
E_UNEXPECTED	Unexpected error occurred.
E_FAIL	Error occurred.
E_NOTIMPL	Indicates MFC itself should attempt to translate and dispatch the command.
OLECMDERR_E_UNKNOWNGROUP	*pguidCmdGroup* is non-**NULL** but does not specify a recognized command group.
OLECMDERR_E_NOTSUPPORTED	*nCmdID* is not recognized as a valid command in the group pGroup.
OLECMDERR_DISABLED	The command identified by *nCmdID* is disabled and cannot be executed.
OLECMDERR_NOHELP	Caller asked for help on the command identified by *nCmdID* but no help is available.
OLECMDERR_CANCELLED	User canceled the execution.

Remarks

The *pguidCmdGroup* and the *nCmdID* parameters together uniquely identify the command to invoke. The *nCmdExecOpt* parameter specifies the exact action to take.

See Also

Reference

COleDocObjectItem Class
Hierarchy Chart
IOleCommandTarget::Exec

Other Resources

COleDocObjectItem Members

COleDocObjectItem::GetActiveView

Call this member function to get a pointer to the **IOleDocumentView** interface of the currently active view.

```
LPOLEDOCUMENTVIEW GetActiveView( ) const;
```

Return Value

A pointer to the IOleDocumentView interface of the currently active view. If there is no current view, it returns **NULL**.

Remarks

The reference count on the returned **IOleDocumentView** pointer is not incremented before it is returned by this function.

See Also

Reference

COleDocObjectItem Class

Hierarchy Chart

Other Resources

COleDocObjectItem Members

COleDocObjectItem::GetPageCount

Call this member function to retrieve the number of pages in the document.

```
BOOL GetPageCount(
   LPLONG pnFirstPage,
   LPLONG pcPages
);
```

Parameters

pnFirstPage

A pointer to the number of the document's first page. Can be **NULL**, which indicates the caller doesn't need this number.

pcPages

A pointer to the total number of pages in the document. Can be **NULL**, which indicates the caller doesn't need this number.

Return Value

Nonzero if successful; otherwise 0.

See Also

Reference

COleDocObjectItem Class
Hierarchy Chart
IPrint::GetPageInfo

Other Resources

COleDocObjectItem Members

COleDocObjectItem::OnPreparePrinting

This member function is called by the framework to prepare a document for printing.

```
static BOOL OnPreparePrinting(
   CView* pCaller,
   CPrintInfo* pInfo,
   BOOL bPrintAll = TRUE
);
```

Parameters

pCaller

A pointer to a CView object that is sending the print command.

pInfo

A pointer to a CPrintInfo object that describes the job to be printed.

bPrintAll

Specifies whether the entire document is to be printed.

Return Value

Nonzero if successful; otherwise 0.

See Also

Reference

COleDocObjectItem Class
Hierarchy Chart
COleDocObjectItem::OnPrint

Other Resources

COleDocObjectItem Members

COleDocObjectItem::OnPrint

This member function is called by the framework to print a document.

```
static void OnPrint(
   CView* pCaller,
   CPrintInfo* pInfo,
   BOOL bPrintAll = TRUE
);
```

Parameters

pCaller

A pointer to a CView object that is sending the print command.

pInfo

A pointer to a CPrintInfo object that describes the job to be printed.

bPrintAll

Specifies whether the entire document is to be printed.

See Also

Reference

COleDocObjectItem Class
Hierarchy Chart
COleDocObjectItem::OnPreparePrinting

Other Resources

COleDocObjectItem Members

COleDocObjectItem::QueryCommand

Queries for the status of one or more commands generated by user interface events.

```
HRESULT QueryCommand(
   ULONG nCmdID,
   DWORD* pdwStatus,
   OLECMDTEXT* pCmdText=NULL,
   const GUID* pguidCmdGroup=NULL
);
```

Parameters

nCmdID

identifier of the command being queried for.

pdwStatus

A pointer to the flags returned as a result of the query. For a list of possible values, see OLECMDF.

pCmdText

Pointer to an OLECMDTEXT structure in which to return name and status information for a single command. Can be **NULL** to indicate that the caller does not need this information.

pguidCmdGroup

Unique identifier of the command group; can be **NULL** to specify the standard group.

Return Value

For a complete listing of return values, see IOleCommandTarget::QueryStatus in the *Platform SDK*.

Remarks

This member function emulates the functionality of the IOleCommandTarget::QueryStatus method, as described in the *Platform SDK*.

See Also

Reference

COleDocObjectItem Class

Hierarchy Chart

Other Resources

COleDocObjectItem Members

COleDocObjectItem::Release

Releases the connection to an OLE linked item and closes it if it was open. Does not destroy the client item.

```
virtual void Release(
   OLECLOSE dwCloseOption = OLECLOSE_NOSAVE
);
```

Parameters

dwCloseOption

Flag specifying under what circumstances the OLE item is saved when it returns to the loaded state. For a list of possible values, see COleClientItem::Close.

Remarks

Does not destroy the client item.

See Also

Reference

COleDocObjectItem Class

Hierarchy Chart

Other Resources

COleDocObjectItem Members

COleDocument Class

The base class for OLE documents that support visual editing.

```
class COleDocument : public CDocument
```

Remarks

COleDocument is derived from **CDocument**, which allows your OLE applications to use the document/view architecture provided by the Microsoft Foundation Class Library.

COleDocument treats a document as a collection of CDocItem objects to handle OLE items. Both container and server applications require such an architecture because their documents must be able to contain OLE items. The COleServerItem and COleClientItem classes, both derived from **CDocItem**, manage the interactions between applications and OLE items.

If you are writing a simple container application, derive your document class from **COleDocument**. If you are writing a container application that supports linking to the embedded items contained by its documents, derive your document class from COleLinkingDoc. If you are writing a server application or combination container/server, derive your document class from COleServerDoc. **COleLinkingDoc** and **COleServerDoc** are derived from **COleDocument**, so these classes inherit all the services available in **COleDocument** and **CDocument**.

To use **COleDocument**, derive a class from it and add functionality to manage the application's non-OLE data as well as embedded or linked items. If you define **CDocItem**-derived classes to store the application's native data, you can use the default implementation defined by **COleDocument** to store both your OLE and non-OLE data. You can also design your own data structures for storing your non-OLE data separately from the OLE items. For more information, see the article Containers: Compound Files..

CDocument supports sending your document via mail if mail support (MAPI) is present. **COleDocument** has updated OnFileSendMail to handle compound documents correctly. For more information, see the articles MAPI and MAPI Support in MFC..

Requirements

Header: afxole.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

CONTAINER Sample: Visual Editing Container Application

MFCBIND Sample: Active Document Container

Reference

CDocument Class

Hierarchy Chart

Other Resources

COleDocument Members

COleDocument Members

Base Class Members

CObject Members

CCmdTarget Members

CDocument Members

Construction

COleDocument	Constructs a **COleDocument** object.

Operations

AddItem	Adds an item to the list of items maintained by the document.
ApplyPrintDevice	Sets the print-target device for all client items in the document.
EnableCompoundFile	Causes documents to be stored using the OLE Structured Storage file format.
GetInPlaceActiveItem	Returns the OLE item that is currently in-place active.
GetNextClientItem	Gets the next client item for iterating.
GetNextItem	Gets the next document item for iterating.
GetNextServerItem	Gets the next server item for iterating.
GetStartPosition	Gets the initial position to begin iteration.
HasBlankItems	Checks for blank items in the document.
RemoveItem	Removes an item from the list of items maintained by the document.
UpdateModifiedFlag	Marks the document as modified if any of the contained OLE items have been modified.

Overridables

GetPrimarySelectedItem	Returns the primary selected OLE item in the document.
OnShowViews	Called when the document becomes visible or invisible.

Mail Functions

OnFileSendMail	Sends a mail message with the document attached.

Message Handlers

OnEditChangeIcon	Handles events in the Change Icon menu command.
OnEditConvert	Handles the conversion of an embedded or linked object from one type to another.
OnEditLinks	Handles events in the Links command on the Edit menu.
OnUpdateEditChangeIcon	Called by the framework to update the command UI for the Edit/Change Icon menu option.
OnUpdateEditLinksMenu	Called by the framework to update the command UI for the Edit/Links menu option.

OnUpdateObjectVerbMenu	Called by the framework to update the command UI for the Edit/*ObjectName* menu option and the Verb submenu accessed from Edit/*ObjectName*.
OnUpdatePasteLinkMenu	Called by the framework to update the command UI for the Paste Special menu option.
OnUpdatePasteMenu	Called by the framework to update the command UI for the Paste menu option.

See Also

Reference

COleDocument Class
Hierarchy Chart

COleDocument Member Functions

For information about the member functions in **COleDocument**, see COleDocument Members.

COleDocument::AddItem

Call this function to add an item to the document.

```
virtual void AddItem(
   CDocItem* pItem
);
```

Parameters

pItem

Pointer to the document item being added.

Remarks

You do not need to call this function explicitly when it is called by the **COleClientItem** or **COleServerItem** constructor that accepts a pointer to a document.

See Also

Reference

COleDocument Class
Hierarchy Chart
CDocItem Class
COleDocument::RemoveItem
COleServerItem::COleServerItem
COleClientItem::COleClientItem

Other Resources

COleDocument Members

COleDocument::ApplyPrintDevice

Call this function to change the print-target device for all embedded COleClientItem items in your application's container document.

```
BOOL ApplyPrintDevice(
   const DVTARGETDEVICE* ptd
);
BOOL ApplyPrintDevice(
   const PRINTDLG* ppd
);
```

Parameters

ptd

Pointer to a **DVTARGETDEVICE** data structure, which contains information about the new print-target device. Can be **NULL**.

ppd

Pointer to a **PRINTDLG** data structure, which contains information about the new print-target device. Can be **NULL**.

Return Value

Nonzero if the function was successful; otherwise 0.

Remarks

This function updates the print-target device for all items but does not refresh the presentation cache for those items. To update the presentation cache for an item, call COleClientItem::UpdateLink.

The arguments to this function contain information that OLE uses to identify the target device. The PRINTDLG structure contains information that Windows uses to initialize the common Print dialog box. After the user closes the dialog box, Windows returns information about the user's selections in this structure. The **m_pd** member of a CPrintDialog object is a **PRINTDLG** structure.

For more information, see the PRINTDLG structure in the *Platform SDK*.

For more information, see the DVTARGETDEVICE structure in the *Platform SDK*.

See Also

Reference

COleDocument Class
Hierarchy Chart
CPrintDialog Class

Other Resources

COleDocument Members

COleDocument::COleDocument

Constructs a **COleDocument** object.

```
COleDocument( );
```

See Also

Reference

COleDocument Class

Hierarchy Chart

Other Resources

COleDocument Members

COleDocument::EnableCompoundFile

Call this function if you want to store the document using the compound-file format.

```
void EnableCompoundFile(
   BOOL bEnable = TRUE
);
```

Parameters

bEnable

Specifies whether compound file support is enabled or disabled.

Remarks

This is also called structured storage. You typically call this function from the constructor of your **COleDocument**-derived class. For more information about compound documents, see the article Containers: Compound Files..

If you do not call this member function, documents will be stored in a nonstructured ("flat") file format.

After compound file support is enabled or disabled for a document, the setting should not be changed during the document's lifetime.

See Also

Reference

COleDocument Class
Hierarchy Chart
COleClientItem Class

Other Resources

COleDocument Members

COleDocument::GetInPlaceActiveItem

Call this function to get the OLE item that is currently activated in place in the frame window containing the view identified by *pWnd*.

```
virtual COleClientItem* GetInPlaceActiveItem(
   CWnd* pWnd
);
```

Parameters

pWnd

Pointer to the window that displays the container document.

Return Value

A pointer to the single, in-place active OLE item; **NULL** if there is no OLE item currently in the "in-place active" state.

See Also

Reference

COleDocument Class

Hierarchy Chart

COleClientItem Class

Other Resources

COleDocument Members

COleDocument::GetNextClientItem

Call this function repeatedly to access each of the client items in your document.

```
COleClientItem* GetNextClientItem(
   POSITION& pos
) const;
```

Parameters

pos

A reference to a **POSITION** value set by a previous call to **GetNextClientItem**; the initial value is returned by the **GetStartPosition** member function.

Return Value

A pointer to the next client item in the document, or **NULL** if there are no more client items.

Remarks

After each call, the value of *pos* is set for the next item in the document, which might or might not be a client item.

Example

```
// Example for COleDocument::GetNextClientItem
// pDoc points to a COleDocument object
POSITION pos = pDoc->GetStartPosition();
COleClientItem *pItem;
while ( ( pItem = pDoc->GetNextClientItem( pos ) ) != NULL )
{
   // Use pItem
}
```

See Also

Reference

COleDocument Class
Hierarchy Chart
COleClientItem Class
COleDocument::GetStartPosition
COleDocument::GetNextServerItem
COleDocument::GetNextItem

Other Resources

COleDocument Members

COleDocument::GetNextItem

Call this function repeatedly to access each of the items in your document.

```
virtual CDocItem* GetNextItem(
   POSITION& pos
) const;
```

Parameters

pos

A reference to a **POSITION** value set by a previous call to **GetNextItem**; the initial value is returned by the **GetStartPosition** member function.

Return Value

A pointer to the document item at the specified position.

Remarks

After each call, the value of *pos* is set to the **POSITION** value of the next item in the document. If the retrieved element is the last element in the document, the new value of *pos* is **NULL**.

Example

```
// Example for COleDocument::GetNextItem
// pDoc points to a COleDocument object
POSITION pos = pDoc->GetStartPosition();
CDocItem *pItem;
while( pos != NULL )
{
   pItem = pDoc->GetNextItem( pos );
   // Use pItem
}
```

See Also

Reference

COleDocument Class
Hierarchy Chart
COleDocument::GetStartPosition
COleDocument::GetNextClientItem
COleDocument::GetNextServerItem

Other Resources

COleDocument Members

COleDocument::GetNextServerItem

Call this function repeatedly to access each of the server items in your document.

```
COleServerItem* GetNextServerItem(
   POSITION& pos
) const;
```

Parameters

pos

A reference to a **POSITION** value set by a previous call to **GetNextServerItem**; the initial value is returned by the **GetStartPosition** member function.

Return Value

A pointer to the next server item in the document, or **NULL** if there are no more server items.

Remarks

After each call, the value of *pos* is set for the next item in the document, which might or might not be a server item.

Example

```
// Example for COleDocument::GetNextServerItem
// pDoc points to a COleDocument object
POSITION pos = pDoc->GetStartPosition();
COleServerItem *pItem;
while ( ( pItem = pDoc->GetNextServerItem( pos ) ) != NULL )
{
   // Use pItem
}
```

See Also

Reference

COleDocument Class
Hierarchy Chart
COleServerItem Class
COleDocument::GetStartPosition
COleDocument::GetNextClientItem
COleDocument::GetNextItem

Other Resources

COleDocument Members

COleDocument::GetPrimarySelectedItem

Called by the framework to retrieve the currently selected OLE item in the specified view.

```
virtual COleClientItem* GetPrimarySelectedItem(
   CView* pView
);
```

Parameters

pView

Pointer to the active view object displaying the document.

Return Value

A pointer to the single, selected OLE item; **NULL** if no OLE items are selected or if more than one is selected.

Remarks

The default implementation searches the list of contained OLE items for a single selected item and returns a pointer to it. If there is no item selected, or if there is more than one item selected, the function returns **NULL**. You must override the **CView::IsSelected** member function in your view class for this function to work. Override this function if you have your own method of storing contained OLE items.

See Also

Reference

COleDocument Class
Hierarchy Chart
CView::IsSelected

Other Resources

COleDocument Members

COleDocument::GetStartPosition

Call this function to get the position of the first item in the document.

```
virtual POSITION GetStartPosition( ) const;
```

Return Value

A **POSITION** value that can be used to begin iterating through the document's items; **NULL** if the document has no items.

Remarks

Pass the value returned to **GetNextItem**, **GetNextClientItem**, or **GetNextServerItem**.

See Also

Reference

COleDocument Class
Hierarchy Chart
COleDocument::GetNextItem
COleDocument::GetNextClientItem
COleDocument::GetNextServerItem

Other Resources

COleDocument Members

COleDocument::HasBlankItems

Call this function to determine whether the document contains any blank items.

```
BOOL HasBlankItems( ) const;
```

Return Value

Nonzero if the document contains any blank items; otherwise 0.

Remarks

A blank item is one whose rectangle is empty.

See Also

Reference

COleDocument Class

Hierarchy Chart

CDocItem::IsBlank

Other Resources

COleDocument Members

COleDocument::OnEditChangeIcon

Displays the OLE Change Icon dialog box and changes the icon representing the currently selected OLE item to the icon the user selects in the dialog box.

```
afx_msg void OnEditChangeIcon( );
```

Remarks

OnEditChangeIcon creates and launches a **COleChangeIconDialog** Change Icon dialog box.

See Also

Reference

COleDocument Class
Hierarchy Chart
COleDocument::OnUpdateEditChangeIcon
COleChangeIconDialog Class

Other Resources

COleDocument Members

COleDocument::OnEditConvert

Displays the OLE Convert dialog box and converts or activates the currently selected OLE item according to user selections in the dialog box.

```
afx_msg void OnEditConvert( );
```

Remarks

OnEditConvert creates and launches a **COleConvertDialog** Convert dialog box.

An example of conversion is converting a Microsoft Word document into a WordPad document.

See Also

Reference

COleDocument Class
Hierarchy Chart
COleDocument::OnUpdateObjectVerbMenu
COleConvertDialog Class

Other Resources

COleDocument Members

COleDocument::OnEditLinks

Displays the OLE Edit/Links dialog box.

```
afx_msg void OnEditLinks( );
```

Remarks

OnEditLinks creates and launches a **COleLinksDialog** Links dialog box that allows the user to change the linked objects.

See Also

Reference

COleDocument Class
Hierarchy Chart
COleDocument::OnUpdateEditLinksMenu
COleLinksDialog Class

Other Resources

COleDocument Members

COleDocument::OnFileSendMail

Sends a message via the resident mail host (if any) with the document as an attachment.

```
afx_msg void OnFileSendMail( );
```

Remarks

OnFileSendMail calls **OnSaveDocument** to serialize (save) untitled and modified documents to a temporary file, which is then sent via electronic mail. If the document has not been modified, a temporary file is not needed; the original is sent. **OnFileSendMail** loads MAPI32.DLL if it has not already been loaded.

Unlike the implementation of **OnFileSendMail** for **CDocument**, this function handles compound files correctly.

For more information, see the MAPI Topics and MAPI Support in MFC articles..

See Also

Reference

COleDocument Class
Hierarchy Chart
CDocument::OnFileSendMail
CDocument::OnUpdateFileSendMail
CDocument::OnSaveDocument

Other Resources

COleDocument Members

COleDocument::OnShowViews

The framework calls this function after the document's visibility state changes.

```
virtual void OnShowViews(
   BOOL bVisible
);
```

Parameters

bVisible

Indicates whether the document has become visible or invisible.

Remarks

The default version of this function does nothing. Override it if your application must perform any special processing when the document's visibility changes.

See Also

Reference

COleDocument Class

Hierarchy Chart

Other Resources

COleDocument Members

COleDocument::OnUpdateEditChangeIcon

Called by the framework to update the Change Icon command on the Edit menu.

```
afx_msg void OnUpdateEditChangeIcon(
   CCmdUI* pCmdUI
);
```

Parameters

pCmdUI

A pointer to a **CCmdUI** structure that represents the menu that generated the update command. The update handler calls the **Enable** member function of the **CCmdUI** structure through *pCmdUI* to update the user interface.

Remarks

OnUpdateEditChangeIcon updates the command's user interface depending on whether or not a valid icon exists in the document. Override this function to change the behavior.

See Also

Reference

COleDocument Class
Hierarchy Chart
COleDocument::OnEditChangeIcon
CCmdUI Class

Other Resources

COleDocument Members

COleDocument::OnUpdateEditLinksMenu

Called by the framework to update the Links command on the Edit menu.

```
afx_msg void OnUpdateEditLinksMenu(
   CCmdUI* pCmdUI
);
```

Parameters

pCmdUI

A pointer to a **CCmdUI** structure that represents the menu that generated the update command. The update handler calls the **Enable** member function of the **CCmdUI** structure through *pCmdUI* to update the user interface.

Remarks

Starting with the first OLE item in the document, **OnUpdateEditLinksMenu** accesses each item, tests whether the item is a link, and, if it is a link, enables the Links command. Override this function to change the behavior.

See Also

Reference

COleDocument Class
Hierarchy Chart
COleDocument::OnEditLinks
COleDocument::GetStartPosition
COleDocument::GetNextClientItem
CCmdUI Class

Other Resources

COleDocument Members

COleDocument::OnUpdateObjectVerbMenu

Called by the framework to update the *ObjectName* command on the Edit menu and the Verb submenu accessed from the *ObjectName* command, where *ObjectName* is the name of the OLE object embedded in the document.

```
afx_msg void OnUpdateObjectVerbMenu(
   CCmdUI* pCmdUI
);
```

Parameters

pCmdUI

A pointer to a **CCmdUI** structure that represents the menu that generated the update command. The update handler calls the **Enable** member function of the **CCmdUI** structure through *pCmdUI* to update the user interface.

Remarks

OnUpdateObjectVerbMenu updates the *ObjectName* command's user interface depending on whether or not a valid object exists in the document. If an object exists, the *ObjectName* command on the Edit menu is enabled. When this menu command is selected, the Verb submenu is displayed. The Verb submenu contains all the verb commands available for the object, such as Edit, Properties, and so on. Override this function to change the behavior.

See Also

Reference

COleDocument Class
Hierarchy Chart
COleDocument::OnEditConvert
CCmdUI Class

Other Resources

COleDocument Members

COleDocument::OnUpdatePasteLinkMenu

Called by the framework to determine whether a linked OLE item can be pasted from the Clipboard.

```
afx_msg void OnUpdatePasteLinkMenu(
   CCmdUI* pCmdUI
);
```

Parameters

pCmdUI

A pointer to a **CCmdUI** structure that represents the menu that generated the update command. The update handler calls the **Enable** member function of the **CCmdUI** structure through *pCmdUI* to update the user interface.

Remarks

The Paste Special menu command is enabled or disabled depending on whether the item can be pasted into the document or not.

See Also

Reference

COleDocument Class
Hierarchy Chart
COleDocument::OnUpdatePasteMenu
CCmdUI Class

Other Resources

COleDocument Members

COleDocument::OnUpdatePasteMenu

Called by the framework to determine whether an embedded OLE item can be pasted from the Clipboard.

```
afx_msg void OnUpdatePasteMenu(
   CCmdUI* pCmdUI
);
```

Parameters

pCmdUI

A pointer to a **CCmdUI** structure that represents the menu that generated the update command. The update handler calls the **Enable** member function of the **CCmdUI** structure through *pCmdUI* to update the user interface.

Remarks

The Paste menu command and button are enabled or disabled depending on whether the item can be pasted into the document or not.

See Also

Reference

COleDocument Class
Hierarchy Chart
COleDocument::OnUpdatePasteLinkMenu
CCmdUI Class

Other Resources

COleDocument Members

COleDocument::RemoveItem

Call this function to remove an item from the document.

```
virtual void RemoveItem(
   CDocItem* pItem
);
```

Parameters

pItem

Pointer to the document item to be removed.

Remarks

You typically do not need to call this function explicitly; it is called by the destructors for **COleClientItem** and **COleServerItem**.

See Also

Reference

COleDocument Class
Hierarchy Chart
COleServerItem Class
COleClientItem Class
COleDocument::AddItem
CDocItem Class

Other Resources

COleDocument Members

COleDocument::UpdateModifiedFlag

Call this function to mark the document as modified if any of the contained OLE items have been modified.

```
virtual void UpdateModifiedFlag( );
```

Remarks

This allows the framework to prompt the user to save the document before closing, even if the native data in the document has not been modified.

See Also

Reference

COleDocument Class
Hierarchy Chart
CDocument::SetModifiedFlag
COleClientItem::IsModified

Other Resources

COleDocument Members

COleDropSource Class

Allows data to be dragged to a drop target.

```
class COleDropSource : public CCmdTarget
```

Remarks

The COleDropTarget class handles the receiving portion of the drag-and-drop operation. The **COleDropSource** object is responsible for determining when a drag operation begins, providing feedback during the drag operation, and determining when the drag operation ends.

To use a **COleDropSource** object, just call the constructor. This simplifies the process of determining what events, such as a mouse click, begin a drag operation using COleDataSource::DoDragDrop, COleClientItem::DoDragDrop, or COleServerItem::DoDragDrop function. These functions will create a **COleDropSource** object for you. You might want to modify the default behavior of the **COleDropSource** overridable functions. These member functions will be called at the appropriate times by the framework.

For more information on drag-and-drop operations using OLE, see the article Drag and Drop (OLE).

For more information, see IDropSource in the *Platform SDK*.

Requirements

Header: afxole.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

HIERSVR Sample: Demonstrates a Server Application with OLE Drag and Drop

OCLIENT Sample: Illustrates a Visual Editing Container Application

Reference

CCmdTarget Class

Hierarchy Chart

Other Resources

COleDropSource Members

COleDropSource Members

Base Class Members

CObject Members

CCmdTarget Members

COleDropSource Members

Construction

COleDropSource	Constructs a **COleDropSource** object.

Overridables

GiveFeedback	Changes the cursor during a drag-and-drop operation.
OnBeginDrag	Handles mouse capture during a drag-and-drop operation.
QueryContinueDrag	Checks to see whether dragging should continue.

See Also

Reference

COleDropSource Class

Hierarchy Chart

COleDropSource Member Functions

For information about the member functions in **COleDropSource**, see COleDropSource Members.

COleDropSource::COleDropSource

Constructs a **COleDropSource** object.

```
COleDropSource( );
```

See Also

Reference

COleDropSource Class

Hierarchy Chart

COleDropTarget Class

Other Resources

COleDropSource Members

COleDropSource::GiveFeedback

Called by the framework after calling COleDropTarget::OnDragOver or COleDropTarget::DragEnter.

```
virtual SCODE GiveFeedback(
   DROPEFFECT dropEffect
);
```

Parameters

dropEffect

The effect you would like to display to the user, usually indicating what would happen if a drop occurred at this point with the selected data. Typically, this is the value returned by the most recent call to CView::OnDragEnter or CView::OnDragOver. It can be one or more of the following:

- **DROPEFFECT_NONE** A drop would not be allowed.
- **DROPEFFECT_COPY** A copy operation would be performed.
- **DROPEFFECT_MOVE** A move operation would be performed.
- **DROPEFFECT_LINK** A link from the dropped data to the original data would be established.
- **DROPEFFECT_SCROLL** A drag scroll operation is about to occur or is occurring in the target.

Return Value

Returns **DRAGDROP_S_USEDEFAULTCURSORS** if dragging is in progress, **NOERROR** if it is not.

Remarks

Override this function to provide feedback to the user about what would happen if a drop occurred at this point. The default implementation uses the OLE default cursors. For more information on drag-and-drop operations using OLE, see the article Drag and Drop (OLE).

For more information, see IDropSource::GiveFeedback, IDropTarget::DragOver, and IDropTarget::DragEnter in the *Platform SDK*.

See Also

Reference

COleDropSource Class
Hierarchy Chart
CView::OnDragEnter
CView::OnDragOver

Other Resources

COleDropSource Members

COleDropSource::OnBeginDrag

Called by the framework when an event occurs that could begin a drag operation, such as pressing the left mouse button.

```
virtual BOOL OnBeginDrag(
   CWnd* pWnd
);
```

Parameters

pWnd

Points to the window that contains the selected data.

Return Value

Nonzero if dragging is allowed, otherwise 0.

Remarks

Override this function if you want to modify the way the dragging process is started. The default implementation captures the mouse and stays in drag mode until the user clicks the left or right mouse button or hits ESC, at which time it releases the mouse.

See Also

Reference

COleDropSource Class
Hierarchy Chart
COleDropSource::GiveFeedback

Other Resources

COleDropSource Members

COleDropSource::QueryContinueDrag

After dragging has begun, this function is called repeatedly by the framework until the drag operation is either canceled or completed.

```
virtual SCODE QueryContinueDrag(
   BOOL bEscapePressed,
      DWORD dwKeyState
);
```

Parameters

bEscapePressed

States whether the ESC key has been pressed since the last call to **COleDropSource::QueryContinueDrag**.

dwKeyState

Contains the state of the modifier keys on the keyboard. This is a combination of any number of the following: **MK_CONTROL**, **MK_SHIFT**, **MK_ALT**, **MK_LBUTTON**, **MK_MBUTTON**, and **MK_RBUTTON**.

Return Value

DRAGDROP_S_CANCEL if the ESC key or right button is pressed, or left button is raised before dragging starts. **DRAGDROP_S_DROP** if a drop operation should occur. Otherwise **S_OK**.

Remarks

Override this function if you want to change the point at which dragging is canceled or a drop occurs.

The default implementation initiates the drop or cancels the drag as follows. It cancels a drag operation when the ESC key or the right mouse button is pressed. It initiates a drop operation when the left mouse button is raised after dragging has started. Otherwise, it returns **S_OK** and performs no further operations.

Because this function is called frequently, it should be optimized as much as possible.

See Also

Reference

COleDropSource Class
Hierarchy Chart
COleDropSource::OnBeginDrag
COleDropTarget::OnDrop

Other Resources

COleDropSource Members

COleDropTarget Class

Provides the communication mechanism between a window and the OLE libraries.

```
class COleDropTarget : public CCmdTarget
```

Remarks

Creating an object of this class allows a window to accept data through the OLE drag-and-drop mechanism.

To get a window to accept drop commands, you should first create an object of the **COleDropTarget** class, and then call the Register function with a pointer to the desired **CWnd** object as the only parameter.

For more information on drag-and-drop operations using OLE, see the article Drag and Drop (OLE).

Requirements

Header: afxole.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

HIERSVR Sample: Demonstrates a Server Application with OLE Drag and Drop

OCLIENT Sample: Illustrates a Visual Editing Container Application

Reference

CCmdTarget Class

Hierarchy Chart

COleDropSource Class

Other Resources

COleDropTarget Members

COleDropTarget Members

Base Class Members

CObject Members

CCmdTarget Members

Construction

COleDropTarget	Constructs a **COleDropTarget** object.

Operations

Register	Registers the window as a valid drop target.
Revoke	Causes the window to cease being a valid drop target.

Overridables

OnDragEnter	Called when the cursor first enters the window.
OnDragLeave	Called when the cursor is dragged out of the window.
OnDragOver	Called repeatedly when the cursor is dragged over the window.
OnDragScroll	Called to determine whether the cursor is dragged into the scroll region of the window.
OnDrop	Called when data is dropped into the window, default handler.
OnDropEx	Called when data is dropped into the window, initial handler.

See Also

Reference

COleDropTarget Class

Hierarchy Chart

COleDropTarget Member Functions

For information about the member functions in **COleDropTarget**, see COleDropTarget Members.

COleDropTarget::COleDropTarget

Constructs an object of class **COleDropTarget**.

```
COleDropTarget( );
```

Remarks

Call Register to associate this object with a window.

See Also

Reference

COleDropTarget Class
Hierarchy Chart
COleDropSource Class
COleDropTarget::Register
COleDropTarget::Revoke

Other Resources

COleDropTarget Members

COleDropTarget::OnDragEnter

Called by the framework when the cursor is first dragged into the window.

```
virtual DROPEFFECT OnDragEnter(
   CWnd* pWnd,
   COleDataObject* pDataObject,
   DWORD dwKeyState,
   CPoint point
);
```

Parameters

pWnd

Points to the window the cursor is entering.

pDataObject

Points to the data object containing the data that can be dropped.

dwKeyState

Contains the state of the modifier keys. This is a combination of any number of the following: **MK_CONTROL**, **MK_SHIFT**, **MK_ALT**, **MK_LBUTTON**, **MK_MBUTTON**, and **MK_RBUTTON**.

point

Contains the current location of the cursor in client coordinates.

Return Value

The effect that would result if a drop were attempted at the location specified by *point*. It can be one or more of the following:

- **DROPEFFECT_NONE** A drop would not be allowed.
- **DROPEFFECT_COPY** A copy operation would be performed.
- **DROPEFFECT_MOVE** A move operation would be performed.
- **DROPEFFECT_LINK** A link from the dropped data to the original data would be established.
- **DROPEFFECT_SCROLL** A drag scroll operation is about to occur or is occurring in the target.

Remarks

Override this function to allow drop operations to occur in the window. The default implementation calls CView::OnDragEnter, which simply returns **DROPEFFECT_NONE** by default.

For more information, see IDropTarget::DragEnter in the *Platform SDK*.

See Also

Reference

COleDropTarget Class
Hierarchy Chart
COleDropTarget::OnDragOver
COleDropTarget::OnDragLeave
COleDropTarget::OnDrop
COleDropTarget::OnDropEx
CView::OnDragEnter

Other Resources

COleDropTarget Members

COleDropTarget::OnDragLeave

Called by the framework when the cursor leaves the window while a dragging operation is in effect.

```
virtual void OnDragLeave(
   CWnd* pWnd
);
```

Parameters

pWnd

Points to the window the cursor is leaving.

Remarks

Override this function if you want special behavior when the drag operation leaves the specified window. The default implementation of this function calls CView::OnDragLeave.

For more information, see IDropTarget::DragLeave in the *Platform SDK*.

See Also

Reference

COleDropTarget Class
Hierarchy Chart
COleDropTarget::OnDragEnter
COleDropTarget::OnDragOver
COleDropTarget::OnDrop
COleDropTarget::OnDropEx

Other Resources

COleDropTarget Members

COleDropTarget::OnDragOver

Called by the framework when the cursor is dragged over the window.

```
virtual DROPEFFECT OnDragOver(
   CWnd* pWnd,
   COleDataObject* pDataObject,
   DWORD dwKeyState,
   CPoint point
);
```

Parameters

pWnd

Points to the window that the cursor is over.

pDataObject

Points to the data object that contains the data to be dropped.

dwKeyState

Contains the state of the modifier keys. This is a combination of any number of the following: **MK_CONTROL**, **MK_SHIFT**, **MK_ALT**, **MK_LBUTTON**, **MK_MBUTTON**, and **MK_RBUTTON**.

point

Contains the current location of the cursor in client coordinates.

Return Value

The effect that would result if a drop were attempted at the location specified by *point*. It can be one or more of the following:

- **DROPEFFECT_NONE** A drop would not be allowed.
- **DROPEFFECT_COPY** A copy operation would be performed.
- **DROPEFFECT_MOVE** A move operation would be performed.
- **DROPEFFECT_LINK** A link from the dropped data to the original data would be established.
- **DROPEFFECT_SCROLL** Indicates that a drag scroll operation is about to occur or is occurring in the target.

Remarks

This function should be overridden to allow drop operations to occur in the window. The default implementation of this function calls CView::OnDragOver, which returns **DROPEFFECT_NONE** by default. Because this function is called frequently during a drag-and-drop operation, it should be optimized as much as possible.

For more information, see IDropTarget::DragOver in the *Platform SDK*.

Example

```
DROPEFFECT CMyView::OnDragOver(COleDataObject* pDataObj , DWORD dwKey, CPoint pt)
{
   DROPEFFECT de = DROPEFFECT_NONE;
   //Determine the type of operation
   if(dwKey & (MK_SHIFT | MK_CONTROL))
      de =  DROPEFFECT_LINK;
   if(dwKey &  MK_CONTROL)
      de =  DROPEFFECT_COPY;
   if(dwKey & MK_SHIFT)
      de =  DROPEFFECT_MOVE;
   return de;
}
```

See Also

Reference

COleDropTarget Class
Hierarchy Chart
COleDropTarget::OnDragEnter
COleDropTarget::OnDragLeave
COleDropTarget::OnDrop
COleDropTarget::OnDropEx

Other Resources

COleDropTarget Members

COleDropTarget::OnDragScroll

Called by the framework before calling OnDragEnter or OnDragOver to determine whether *point* is in the scrolling region.

```
virtual DROPEFFECT OnDragScroll(
   CWnd* pWnd,
   DWORD dwKeyState,
   CPoint point
);
```

Parameters

pWnd

Points to the window the cursor is currently over.

dwKeyState

Contains the state of the modifier keys. This is a combination of any number of the following: **MK_CONTROL**, **MK_SHIFT**, **MK_ALT**, **MK_LBUTTON**, **MK_MBUTTON**, and **MK_RBUTTON**.

point

Contains the location of the cursor, in pixels, relative to the screen.

Return Value

The effect that would result if a drop were attempted at the location specified by *point*. It can be one or more of the following:

- **DROPEFFECT_NONE** A drop would not be allowed.
- **DROPEFFECT_COPY** A copy operation would be performed.
- **DROPEFFECT_MOVE** A move operation would be performed.
- **DROPEFFECT_LINK** A link from the dropped data to the original data would be established.
- **DROPEFFECT_SCROLL** Indicates that a drag scroll operation is about to occur or is occurring in the target.

Remarks

Override this function when you want to provide special behavior for this event. The default implementation of this function calls CView::OnDragScroll, which returns **DROPEFFECT_NONE** and scrolls the window when the cursor is dragged into the default scroll region inside the border of the window.

See Also

Reference

COleDropTarget Class

Hierarchy Chart

Other Resources

COleDropTarget Members

COleDropTarget::OnDrop

Called by the framework when a drop operation is to occur.

```
virtual BOOL OnDrop(
   CWnd* pWnd,
   COleDataObject* pDataObject,
   DROPEFFECT dropEffect,
   CPoint point
);
```

Parameters

pWnd

Points to the window the cursor is currently over.

pDataObject

Points to the data object that contains the data to be dropped.

dropEffect

The effect that the user chose for the drop operation. It can be one or more of the following:

- **DROPEFFECT_COPY** A copy operation would be performed.
- **DROPEFFECT_MOVE** A move operation would be performed.
- **DROPEFFECT_LINK** A link from the dropped data to the original data would be established.

point

Contains the location of the cursor, in pixels, relative to the screen.

Return Value

Nonzero if the drop is successful; otherwise 0.

Remarks

The framework first calls OnDropEx. If the **OnDropEx** function does not handle the drop, the framework then calls this member function, **OnDrop**. Typically, the application overrides OnDropEx in the view class to handle right mouse-button drag and drop. Typically, the view class OnDrop is used to handle simple drag and drop.

The default implementation of **COleDropTarget::OnDrop** calls CView::OnDrop, which simply returns **FALSE** by default.

For more information, see IDropTarget::Drop in the *Platform SDK*.

See Also

Reference

COleDropTarget Class
Hierarchy Chart
COleDropTarget::OnDragOver
COleDropTarget::OnDragEnter
COleDropTarget::OnDropEx

Other Resources

COleDropTarget Members

COleDropTarget::OnDropEx

Called by the framework when a drop operation is to occur.

```
virtual DROPEFFECT OnDropEx(
   CWnd* pWnd,
   COleDataObject* pDataObject,
   DROPEFFECT dropDefault,
   DROPEFFECT dropList,
   CPoint point
);
```

Parameters

pWnd

Points to the window the cursor is currently over.

pDataObject

Points to the data object that contains the data to be dropped.

dropDefault

The effect that the user chose for the default drop operation based on the current key state. It can be **DROPEFFECT_NONE**. Drop effects are discussed in the Remarks section.

dropList

A list of the drop effects that the drop source supports. Drop effect values can be combined using the bitwise OR (**|**) operation. Drop effects are discussed in the Remarks section.

point

Contains the location of the cursor, in pixels, relative to the screen.

Return Value

The drop effect that resulted from the drop attempt at the location specified by *point*. Drop effects are discussed in the Remarks section.

Remarks

The framework first calls this function. If it does not handle the drop, the framework then calls OnDrop. Typically, you will override OnDropEx in the view class to support right mouse-button drag and drop. Typically, the view class OnDrop is used to handle the case of support for simple drag and drop.

The default implementation of **COleDropTarget::OnDropEx** calls CView::OnDropEx. By default, CView::OnDropEx simply returns a dummy value to indicate the OnDrop member function should be called.

Drop effects describe the action associated with a drop operation. See the following list of drop effects:

- **DROPEFFECT_NONE** A drop would not be allowed.
- **DROPEFFECT_COPY** A copy operation would be performed.
- **DROPEFFECT_MOVE** A move operation would be performed.
- **DROPEFFECT_LINK** A link from the dropped data to the original data would be established.
- **DROPEFFECT_SCROLL** Indicates that a drag scroll operation is about to occur or is occurring in the target.

For more information, see IDropTarget::Drop in the *Platform SDK*.

See Also

Reference

COleDropTarget Class
Hierarchy Chart
COleDropTarget::OnDragOver

COleDropTarget::OnDragEnter

Other Resources

COleDropTarget Members

COleDropTarget::Register

Call this function to register your window with the OLE DLLs as a valid drop target.

```
BOOL Register(
   CWnd* pWnd
);
```

Parameters

pWnd

Points to the window that is to be registered as a drop target.

Return Value

Nonzero if registration is successful; otherwise 0.

Remarks

This function must be called for drop operations to be accepted.

For more information, see RegisterDragDrop in the *Platform SDK*.

See Also

Reference

COleDropTarget Class
Hierarchy Chart
COleDropTarget::Revoke
COleDropTarget::COleDropTarget

Other Resources

COleDropTarget Members

COleDropTarget::Revoke

Call this function before destroying any window that has been registered as a drop target through a call to Register to remove it from the list of drop targets.

```
virtual void Revoke( );
```

Remarks

This function is called automatically from the OnDestroy handler for the window that was registered, so it is usually not necessary to call this function explicitly.

For more information, see RevokeDragDrop in the *Platform SDK*.

See Also

Reference

COleDropTarget Class

Hierarchy Chart

Other Resources

COleDropTarget Members

COleException Class

Represents an exception condition related to an OLE operation.

```
class COleException : public CException
```

Remarks

The **COleException** class includes a public data member that holds the status code indicating the reason for the exception.

In general, you should not create a **COleException** object directly; instead, you should call AfxThrowOleException.

For more information on exceptions, see the articles Exception Handling (MFC) and Exceptions: OLE Exceptions.

Requirements

Header: afxdisp.h

See Also

Tasks

CALCDRIV Sample: Demonstrates an Automation Client Application

Reference

CException Class

Hierarchy Chart

Other Resources

COleException Members

COleException Members

Base Class Members

CObject Members

CException Members

Data Members

m_sc	Contains the status code that indicates the reason for the exception.

Operations

Process	Translates a caught exception into an OLE return code.

See Also

Reference

COleException Class

Hierarchy Chart

COleException Member Functions

For information about the member functions in **COleException**, see COleException Members.

COleException::Process

Call the **Process** member function to translate a caught exception into an OLE status code.

```
static SCODE PASCAL Process(
   const CException* pAnyException
);
```

Parameters

pAnyException

Pointer to a caught exception.

Return Value

An OLE status code.

Remarks

Note
This function is **static**.

For more information on **SCODE**, see Structure of COM Error Codes in the *Platform SDK*.

Example

See the example for COleDispatchDriver::CreateDispatch.

See Also

Reference

COleException Class
Hierarchy Chart
CException Class

Other Resources

COleException Members

COleException Data Members

For information about the data members in **COleException**, see COleException Members.

COleException::m_sc

This data member holds the OLE status code that indicates the reason for the exception.

```
SCODE m_sc;
```

Remarks

This variable's value is set by AfxThrowOleException.

For more information on **SCODE**, see Structure of COM Error Codes in the *Platform SDK*.

Example

```
try {
   // ...
   // Some OLE call
   // ...
}
catch (COleException *e) {

   TRACE ("COleException caught. SCODE = %x\n", e->m_sc);
   e->Delete();
}
```

See Also

Reference

COleException Class
Hierarchy Chart
AfxThrowOleException

Other Resources

COleException Members

COleInsertDialog Class

Used for the OLE Insert Object dialog box.

```
class COleInsertDialog : public COleDialog
```

Remarks

Create an object of class **COleInsertDialog** when you want to call this dialog box. After a **COleInsertDialog** object has been constructed, you can use the m_io structure to initialize the values or states of controls in the dialog box. The **m_io** structure is of type **OLEUIINSERTOBJECT**. For more information about using this dialog class, see the DoModal member function.

Note
Application Wizard-generated container code uses this class.

For more information, see the OLEUIINSERTOBJECT structure in the *Platform SDK*.

For more information regarding OLE-specific dialog boxes, see the article Dialog Boxes in OLE.

Requirements

Header: afxodlgs.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

OCLIENT Sample: Illustrates a Visual Editing Container Application

Reference

COleDialog Class

Hierarchy Chart

COleDialog Class

Other Resources

COleInsertDialog Members

COleInsertDialog Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CDialog Members

CCommonDialog Members

COleDialog Members

Data Members

m_io	A structure of type **OLEUIINSERTOBJECT** that controls the behavior of the dialog box.

Construction

COleInsertDialog	Constructs a **COleInsertDialog** object.

Operations and Attributes

CreateItem	Creates the item selected in the dialog box.
DoModal	Displays the OLE Insert Object dialog box.
GetClassID	Gets the **CLSID** associated with the chosen item.
GetDrawAspect	Tells whether to draw the item as an icon.
GetIconicMetafile	Gets a handle to the metafile associated with the iconic form of this item.
GetPathName	Gets the full path to the file chosen in the dialog box.
GetSelectionType	Gets the type of object selected.

See Also

Reference

COleInsertDialog Class

Hierarchy Chart

COleInsertDialog Member Functions

For information about the member functions in **COleInsertDialog**, see COleInsertDialog Members.

COleInsertDialog::COleInsertDialog

This function constructs only a **COleInsertDialog** object.

```
COleInsertDialog (
   DWORD dwFlags = IOF_SELECTCREATENEW,
   CWnd* pParentWnd = NULL
);
```

Parameters

dwFlags

Creation flag that contains any number of the following values to be combined using the bitwise-OR operator:

- **IOF_SHOWHELP** Specifies that the Help button will be displayed when the dialog box is called.
- **IOF_SELECTCREATENEW** Specifies that the Create New radio button will be selected initially when the dialog box is called. This is the default and cannot be used with **IOF_SELECTCREATEFROMFILE**.
- **IOF_SELECTCREATEFROMFILE** Specifies that the Create From File radio button will be selected initially when the dialog box is called. Cannot be used with **IOF_SELECTCREATENEW**.
- **IOF_CHECKLINK** Specifies that the Link check box will be checked initially when the dialog box is called.
- **IOF_DISABLELINK** Specifies that the Link check box will be disabled when the dialog box is called.
- **IOF_CHECKDISPLAYASICON** Specifies that the Display As Icon check box will be checked initially, the current icon will be displayed, and the Change Icon button will be enabled when the dialog box is called.
- **IOF_VERIFYSERVERSEXIST** Specifies that the dialog box should validate the classes it adds to the list box by ensuring that the servers specified in the registration database exist before the dialog box is displayed. Setting this flag can significantly impair performance.

pParentWnd

Points to the parent or owner window object (of type **CWnd**) to which the dialog object belongs. If it is **NULL**, the parent window of the dialog object is set to the main application window.

Remarks

To display the dialog box, call the DoModal function.

See Also

Reference

COleInsertDialog Class
Hierarchy Chart
COleInsertDialog::DoModal

Other Resources

COleInsertDialog Members

COleInsertDialog::CreateItem

Call this function to create an object of type COleClientItem only if DoModal returns **IDOK**.

```
BOOL CreateItem(
   COleClientItem* pItem
);
```

Parameters

pItem

Points to the item to be created.

Return Value

Nonzero if item was created; otherwise 0.

Remarks

You must allocate the **COleClientItem** object before you can call this function.

See Also

Reference

COleInsertDialog Class
Hierarchy Chart
COleClientItem::CreateLinkFromFile
COleClientItem::CreateFromFile
COleClientItem::CreateNewItem
COleClientItem::SetDrawAspect
COleInsertDialog::GetSelectionType
COleInsertDialog::DoModal

Other Resources

COleInsertDialog Members

COleInsertDialog::DoModal

Call this function to display the OLE Insert Object dialog box.

```
virtual INT_PTR DoModal( );
INT_PTR DoModal(
   DWORD dwFlags
);
```

Parameters

dwFlags

One of the following values:

COleInsertDialog::DocObjectOnly will insert only DocObjects.

COleInsertDialog::ControlsOnly will insert only ActiveX controls.

If zero, **DoModal** will insert neither a DocObject or an ActiveX control; it results in the same implementation as the first prototype listed above.

Return Value

Completion status for the dialog box. One of the following values:

- **IDOK** if the dialog box was successfully displayed.
- **IDCANCEL** if the user canceled the dialog box.
- **IDABORT** if an error occurred. If **IDABORT** is returned, call the COleDialog::GetLastError member function to get more information about the type of error that occurred. For a listing of possible errors, see the OleUIInsertObject function in the *Platform SDK*.

Remarks

If you want to initialize the various dialog box controls by setting members of the m_io structure, you should do this before calling **DoModal**, but after the dialog object is constructed.

If **DoModal** returns **IDOK**, you can call other member functions to retrieve the settings or information input into the dialog box by the user.

See Also

Reference

COleInsertDialog Class
Hierarchy Chart
COleDialog::GetLastError
CDialog::DoModal
COleInsertDialog::GetSelectionType
COleInsertDialog::GetClassID
COleInsertDialog::GetDrawAspect
COleInsertDialog::GetIconicMetafile
COleInsertDialog::GetPathName
COleInsertDialog::m_io

Other Resources

COleInsertDialog Members

COleInsertDialog::GetClassID

Call this function to get the **CLSID** associated with the selected item only if DoModal returns **IDOK** and the selection type is **COleInsertDialog::createNewItem**.

```
REFCLSID GetClassID( ) const;
```

Return Value

Returns the **CLSID** associated with the selected item.

Remarks

For more information, see CLSID Key in the *Platform SDK*.

See Also

Reference

COleInsertDialog Class
Hierarchy Chart
COleInsertDialog::DoModal
COleInsertDialog::GetSelectionType

Other Resources

COleInsertDialog Members

COleInsertDialog::GetDrawAspect

Call this function to determine if the user chose to display the selected item as an icon.

```
DVASPECT GetDrawAspect( ) const;
```

Return Value

The method needed to render the object.

- **DVASPECT_CONTENT** Returned if the Display As Icon check box was not checked.
- **DVASPECT_ICON** Returned if the Display As Icon check box was checked.

Remarks

Call this function only if DoModal returns **IDOK**.

For more information on drawing aspect, see FORMATETC data structure in the *Platform SDK*.

See Also

Reference

COleInsertDialog Class
Hierarchy Chart
COleInsertDialog::DoModal
COleInsertDialog::COleInsertDialog

Other Resources

COleInsertDialog Members

COleInsertDialog::GetIconicMetafile

Call this function to get a handle to the metafile that contains the iconic aspect of the selected item.

```
HGLOBAL GetIconicMetafile( ) const;
```

Return Value

The handle to the metafile containing the iconic aspect of the selected item, if the Display As Icon check box was checked when the dialog was dismissed by choosing **OK**; otherwise **NULL**.

See Also

Reference

COleInsertDialog Class
Hierarchy Chart
COleInsertDialog::DoModal
COleInsertDialog::GetDrawAspect

Other Resources

COleInsertDialog Members

COleInsertDialog::GetPathName

Call this function to get the full path of the selected file only if DoModal returns **IDOK** and the selection type is not **COleInsertDialog::createNewItem**.

```
CString GetPathName( ) const;
```

Return Value

The full path to the file selected in the dialog box. If the selection type is **createNewItem**, this function returns a meaningless **CString** in release mode or causes an assertion in debug mode.

See Also

Reference

COleInsertDialog Class
Hierarchy Chart
COleInsertDialog::GetSelectionType
COleInsertDialog::DoModal

Other Resources

COleInsertDialog Members

COleInsertDialog::GetSelectionType

Call this function to get the selection type chosen when the Insert Object dialog box was dismissed by choosing **OK**.

```
UINT GetSelectionType( ) const;
```

Return Value

Type of selection made.

Remarks

The return type values are specified by the **Selection** enumeration type declared in the **COleInsertDialog** class.

```
enum Selection
{
   createNewItem,
   insertFromFile,
   linkToFile
};
```

Brief descriptions of these values follow:

- **COleInsertDialog::createNewItem** The Create New radio button was selected.
- **COleInsertDialog::insertFromFile** The Create From File radio button was selected and the Link check box was not checked.
- **COleInsertDialog::linkToFile** The Create From File radio button was selected and the Link check box was checked.

See Also

Reference

COleInsertDialog Class
Hierarchy Chart
COleInsertDialog::DoModal
COleInsertDialog::COleInsertDialog

Other Resources

COleInsertDialog Members

COleInsertDialog Data Members

For information about the data members in **COleInsertDialog**, see COleInsertDialog Members.

COleInsertDialog::m_io

Structure of type **OLEUIINSERTOBJECT** used to control the behavior of the Insert Object dialog box.

```
OLEUIINSERTOBJECT m_io;
```

Remarks

Members of this structure can be modified either directly or through member functions.

For more information, see the OLEUIINSERTOBJECT structure in the *Platform SDK*.

See Also

Reference

COleInsertDialog Class
Hierarchy Chart
COleInsertDialog::COleInsertDialog
COleInsertDialog::DoModal

Other Resources

COleInsertDialog Members

COleIPFrameWnd Class

The base for your application's in-place editing window.

```
class COleIPFrameWnd : public CFrameWnd
```

Remarks

This class creates and positions control bars within the container application's document window. It also handles notifications generated by an embedded COleResizeBar object when the user resizes the in-place editing window.

For more information on using **COleIPFrameWnd**, see the article Activation.

Requirements

Header: afxole.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

HIERSVR Sample: Demonstrates a Server Application with OLE Drag and Drop

Reference

CFrameWnd Class

Hierarchy Chart

CFrameWnd Class

Other Resources

COleIPFrameWnd Members

COleIPFrameWnd Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CFrameWnd Members

Construction

COleIPFrameWnd	Constructs a **COleIPFrameWnd** object.

Overridables

OnCreateControlBars	Called by the framework when an item is activated for in-place editing.
RepositionFrame	Called by the framework to reposition the in-place editing window.

See Also

Reference

COleIPFrameWnd Class

Hierarchy Chart

COleIPFrameWnd Member Functions

For information about the member functions in **COleIPFrameWnd**, see COleIPFrameWnd Members.

COleIPFrameWnd::COleIPFrameWnd

Constructs a **COleIPFrameWnd** object and initializes its in-place state information, which is stored in a structure of type **OLEINPLACEFRAMEINFO**.

```
COleIPFrameWnd( );
```

Remarks

For more information, see OLEINPLACEFRAMEINFO in the *Platform SDK*.

See Also

Reference

COleIPFrameWnd Class

Hierarchy Chart

COleServerDoc::ActivateInPlace

Other Resources

COleIPFrameWnd Members

COleIPFrameWnd::OnCreateControlBars

The framework calls the **OnCreateControlBars** function when an item is activated for in-place editing.

```
virtual BOOL OnCreateControlBars(
   CWnd* pWndFrame,
   CWnd* pWndDoc
);
virtual BOOL OnCreateControlBars(
   CFrameWnd* pWndFrame,
   CFrameWnd* pWndDoc
);
```

Parameters

pWndFrame

Pointer to the container application's frame window.

pWndDoc

Pointer to the container's document-level window. Can be **NULL** if the container is an SDI application.

Return Value

Nonzero on success; otherwise, 0.

Remarks

The default implementation does nothing. Override this function to perform any special processing required when control bars are created.

See Also

Reference

COleIPFrameWnd Class
Hierarchy Chart
COleServerDoc::ActivateInPlace

Other Resources

COleIPFrameWnd Members

COleIPFrameWnd::RepositionFrame

The framework calls the **RepositionFrame** member function to lay out control bars and reposition the in-place editing window so all of it is visible.

```
virtual void RepositionFrame(
   LPCRECT lpPosRect,
   LPCRECT lpClipRect
);
```

Parameters

lpPosRect

Pointer to a **RECT** structure or a **CRect** object containing the in-place frame window's current position coordinates, in pixels, relative to the client area.

lpClipRect

Pointer to a **RECT** structure or a **CRect** object containing the in-place frame window's current clipping-rectangle coordinates, in pixels, relative to the client area.

Remarks

Layout of control bars in the container window differs from that performed by a non-OLE frame window. The non-OLE frame window calculates the positions of control bars and other objects from a given frame-window size, as in a call to CFrameWnd::RecalcLayout. The client area is what remains after space for control bars and other objects is subtracted. A **COleIPFrameWnd** window, on the other hand, positions toolbars in accordance with a given client area. In other words, **CFrameWnd::RecalcLayout** works "from the outside in," whereas **COleIPFrameWnd::RepositionFrame** works "from the inside out."

See Also

Reference

COleIPFrameWnd Class
Hierarchy Chart
CFrameWnd::RecalcLayout

Other Resources

COleIPFrameWnd Members

COleLinkingDoc Class

The base class for OLE container documents that support linking to the embedded items they contain.

```
class COleLinkingDoc : public COleDocument
```

Remarks

A container application that supports linking to embedded items is called a "link container." The OCLIENT sample application is an example of a link container.

When a linked item's source is an embedded item in another document, that containing document must be loaded in order for the embedded item to be edited. For this reason, a link container must be able to be launched by another container application when the user wants to edit the source of a linked item. Your application must also use the COleTemplateServer class so that it can create documents when launched programmatically.

To make your container a link container, derive your document class from **COleLinkingDoc** instead of COleDocument. As with any other OLE container, you must design your class for storing the application's native data as well as embedded or linked items. Also, you must design data structures for storing your native data. If you define a **CDocItem**-derived class for your application's native data, you can use the interface defined by **COleDocument** to store your native data as well as your OLE data.

To allow your application to be launched programmatically by another container, declare a **COleTemplateServer** object as a member of your application's **CWinApp**-derived class:

```
class COleClientApp : public CWinApp
{
// ...
protected:
   COleTemplateServer m_server;
// ...
};
```

In the **InitInstance** member function of your **CWinApp**-derived class, create a document template and specify your **COleLinkingDoc**-derived class as the document class:

```
// CMainDoc is derived from COleLinkingDoc
CMultiDocTemplate* pDocTemplate = new CMultiDocTemplate(IDR_OCLIENTTYPE,
   RUNTIME_CLASS(CMainDoc),
   RUNTIME_CLASS(CSplitFrame),
   RUNTIME_CLASS(CMainView));
pDocTemplate->SetContainerInfo(
   IDR_OCLIENTTYPE_CNTR_IP);
AddDocTemplate(pDocTemplate);
```

Connect your **COleTemplateServer** object to your document templates by calling the object's **ConnectTemplate** member function, and register all class objects with the OLE system by calling **COleTemplateServer::RegisterAll**:

```
m_server.ConnectTemplate(clsid, pDocTemplate, FALSE);
COleTemplateServer::RegisterAll();
```

For a sample **CWinApp**-derived class definition and **InitInstance** function, see OCLIENT.H and OCLIENT.CPP in the MFC sample OCLIENT.

For more information on using **COleLinkingDoc**, see the articles Containers: Implementing a Container and Containers: Advanced Features.

Requirements

Header: afxole.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

OCLIENT Sample: Illustrates a Visual Editing Container Application

Reference

COleDocument Class

Hierarchy Chart

CDocTemplate Class

Other Resources

COleLinkingDoc Members

COleLinkingDoc Members

Base Class Members

CObject Members

CCmdTarget Members

CDocument Members

COleDocument Members

Construction

COleLinkingDoc	Constructs a **COleLinkingDoc** object.

Operations

Register	Registers the document with the OLE system DLLs.
Revoke	Revokes the document's registration.

Overridables

OnFindEmbeddedItem	Finds the specified embedded item.
OnGetLinkedItem	Finds the specified linked item.

See Also

Reference

COleLinkingDoc Class

Hierarchy Chart

COleLinkingDoc Member Functions

For information about the member functions in **COleLinkingDoc**, see COleLinkingDoc Members.

COleLinkingDoc::COleLinkingDoc

Constructs a **COleLinkingDoc** object without beginning communications with the OLE system DLLs.

```
COleLinkingDoc( );
```

Remarks

You must call the **Register** member function to inform OLE that the document is open.

See Also

Reference

COleLinkingDoc Class
Hierarchy Chart
COleLinkingDoc::Register

Other Resources

COleLinkingDoc Members

COleLinkingDoc::OnFindEmbeddedItem

Called by the framework to determine whether the document contains an embedded OLE item with the specified name.

```
virtual COleClientItem* OnFindEmbeddedItem(
   LPCTSTR lpszItemName
);
```

Parameters

lpszItemName

Pointer to the name of the embedded OLE item requested.

Return Value

A pointer to the specified item; **NULL** if the item is not found.

Remarks

The default implementation searches the list of embedded items for an item with the specified name (the name comparison is case sensitive). Override this function if you have your own method of storing or naming embedded OLE items.

See Also

Reference

COleLinkingDoc Class
Hierarchy Chart
COleClientItem Class
COleLinkingDoc::OnGetLinkedItem

Other Resources

COleLinkingDoc Members

COleLinkingDoc::OnGetLinkedItem

Called by the framework to check whether the document contains a linked server item with the specified name.

```
virtual COleServerItem* OnGetLinkedItem(
   LPCTSTR lpszItemName
);
```

Parameters

lpszItemName

Pointer to the name of the linked OLE item requested.

Return Value

A pointer to the specified item; **NULL** if the item is not found.

Remarks

The default **COleLinkingDoc** implementation always returns **NULL**. This function is overriden in the derived class **COleServerDoc** to search the list of OLE server items for a linked item with the specified name (the name comparison is case sensitive). Override this function if you have implemented your own method of storing or retrieving linked server items.

See Also

Reference

COleLinkingDoc Class
Hierarchy Chart
COleServerItem::GetItemName
COleServerItem::SetItemName
COleLinkingDoc::OnFindEmbeddedItem

Other Resources

COleLinkingDoc Members

COleLinkingDoc::Register

Informs the OLE system DLLs that the document is open.

```
BOOL Register(
   COleObjectFactory* pFactory,
   LPCTSTR lpszPathName
);
```

Parameters

pFactory

Pointer to an OLE factory object (can be **NULL**).

lpszPathName

Pointer to the fully qualified path of the container document.

Return Value

Nonzero if the document is successfully registered; otherwise 0.

Remarks

Call this function when creating or opening a named file to register the document with the OLE system DLLs. There is no need to call this function if the document represents an embedded item.

If you are using **COleTemplateServer** in your application, **Register** is called for you by **COleLinkingDoc**'s implementation of **OnNewDocument**, **OnOpenDocument**, and **OnSaveDocument**.

See Also

Reference

COleLinkingDoc Class
Hierarchy Chart
COleTemplateServer Class
COleObjectFactory Class
CDocument::OnNewDocument
CDocument::OnOpenDocument

Other Resources

COleLinkingDoc Members

COleLinkingDoc::Revoke

Informs the OLE system DLLs that the document is no longer open.

```
void Revoke( );
```

Remarks

Call this function to revoke the document's registration with the OLE system DLLs.

You should call this function when closing a named file, but you usually do not need to call it directly. **Revoke** is called for you by **COleLinkingDoc**'s implementation of **OnCloseDocument**, **OnNewDocument**, **OnOpenDocument**, and **OnSaveDocument**.

See Also

Reference

COleLinkingDoc Class
Hierarchy Chart
COleTemplateServer Class
CDocument::OnCloseDocument
CDocument::OnNewDocument
CDocument::OnOpenDocument
CDocument::OnSaveDocument

Other Resources

COleLinkingDoc Members

COleLinksDialog Class

Used for the OLE Edit Links dialog box.

```
class COleLinksDialog : public COleDialog
```

Remarks

Create an object of class **COleLinksDialog** when you want to call this dialog box. After a **COleLinksDialog** object has been constructed, you can use the m_el structure to initialize the values or states of controls in the dialog box. The **m_el** structure is of type **OLEUIEDITLINKS**. For more information about using this dialog class, see the DoModal member function.

Note
Application Wizard-generated container code uses this class.

For more information, see the OLEUIEDITLINKS structure in the *Platform SDK*.

For more information regarding OLE-specific dialog boxes, see the article Dialog Boxes in OLE.

Requirements

Header: afxodlgs.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

COleDialog Class

Hierarchy Chart

COleDialog Class

Other Resources

COleLinksDialog Members

COleLinksDialog Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CDialog Members

CCommonDialog Members

COleDialog Members

Data Members

m_el	A structure of type **OLEUIEDITLINKS** that controls the behavior of the dialog box.

Construction

COleLinksDialog	Constructs a **COleLinksDialog** object.

Operations

DoModal	Displays the OLE Edit Links dialog box.

See Also

Reference

COleLinksDialog Class

Hierarchy Chart

COleLinksDialog Member Functions

For information about the member functions in **COleLinksDialog**, see COleLinksDialog Members.

COleLinksDialog::COleLinksDialog

Constructs a **COleLinksDialog** object.

```
COleLinksDialog (
   COleDocument* pDoc,
   CView* pView,
   DWORD dwFlags = 0,
   CWnd* pParentWnd = NULL
);
```

Parameters

pDoc

Points to the OLE document that contains the links to be edited.

pView

Points to the current view on *pDoc*.

dwFlags

Creation flag, which contains either 0 or **ELF_SHOWHELP** to specify whether the Help button will be displayed when the dialog box is displayed.

pParentWnd

Points to the parent or owner window object (of type **CWnd**) to which the dialog object belongs. If it is **NULL**, the parent window of the dialog box is set to the main application window.

Remarks

This function constructs only a **COleLinksDialog** object. To display the dialog box, call the DoModal function.

See Also

Reference

COleLinksDialog Class
Hierarchy Chart
COleDocument Class
COleLinksDialog::DoModal
CView Class
CWnd Class

Other Resources

COleLinksDialog Members

COleLinksDialog::DoModal

Displays the OLE Edit Links dialog box.

```
virtual INT_PTR DoModal( );
```

Return Value

Completion status for the dialog box. One of the following values:

- **IDOK** if the dialog box was successfully displayed.
- **IDCANCEL** if the user canceled the dialog box.
- **IDABORT** if an error occurred. If **IDABORT** is returned, call the **COleDialog::GetLastError** member function to get more information about the type of error that occurred. For a listing of possible errors, see the OleUIEditLinks function in the *Platform SDK*.

Remarks

If you want to initialize the various dialog box controls by setting members of the m_el structure, you should do it before calling **DoModal**, but after the dialog object is constructed.

See Also

Reference

COleLinksDialog Class
Hierarchy Chart
COleDialog::GetLastError
CDialog::DoModal
COleLinksDialog::m_el

Other Resources

COleLinksDialog Members

COleLinksDialog Data Members

For information about the data members in **COleLinksDialog**, see COleLinksDialog Members.

COleLinksDialog::m_el

Structure of type **OLEUIEDITLINKS** used to control the behavior of the Edit Links dialog box.

```
OLEUIEDITLINKS m_el;
```

Remarks

Members of this structure can be modified either directly or through member functions.

For more information, see the OLEUIEDITLINKS structure in the *Platform SDK*.

See Also

Reference

COleLinksDialog Class
Hierarchy Chart
COleLinksDialog::COleLinksDialog
COleLinksDialog::DoModal

Other Resources

COleLinksDialog Members

COleMessageFilter Class

Manages the concurrency required by the interaction of OLE applications.

```
class COleMessageFilter : public CCmdTarget
```

Remarks

The **COleMessageFilter** class is useful in visual editing server and container applications, as well as OLE automation applications. For server applications that are being called, this class can be used to make the application "busy" so that incoming calls from other container applications are either canceled or retried later. This class can also be used to determine the action to be taken by a calling application when the called application is busy.

Common usage is for a server application to call BeginBusyState and EndBusyState when it would be dangerous for a document or other OLE accessible object to be destroyed. These calls are made in CWinApp::OnIdle during user-interface updates.

By default, a **COleMessageFilter** object is allocated when the application is initialized. It can be retrieved with AfxOleGetMessageFilter.

This is an advanced class; you seldom need to work with it directly.

For more information, see the article Servers: Implementing a Server.

Requirements

Header: afxole.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CCmdTarget Class

Hierarchy Chart

CCmdTarget Class

Other Resources

COleMessageFilter Members

COleMessageFilter Members

Base Class Members

CObject Members

CCmdTarget Members

Construction

COleMessageFilter	Constructs a **COleMessageFilter** object.

Operations

BeginBusyState	Puts the application in the busy state.
EnableBusyDialog	Enables and disables the dialog box that appears when a called application is busy.
EnableNotRespondingDialog	Enables and disables the dialog box that appears when a called application is not responding.
EndBusyState	Terminates the application's busy state.
Register	Registers the message filter with the OLE system DLLs.
Revoke	Revokes the message filter's registration with the OLE system DLLs.
SetBusyReply	Determines the busy application's reply to an OLE call.
SetMessagePendingDelay	Determines how long the application waits for a response to an OLE call.
SetRetryReply	Determines the calling application's reply to a busy application.

Overridables

OnMessagePending	Called by the framework to process messages while an OLE call is in progress.

See Also

Reference

COleMessageFilter Class

Hierarchy Chart

COleMessageFilter Member Functions

For information about the member functions in **COleMessageFilter**, see COleMessageFilter Members.

COleMessageFilter::BeginBusyState

Call this function to begin a busy state.

```
virtual void BeginBusyState( );
```

Remarks

It works in conjunction with EndBusyState to control the application's busy state. The function SetBusyReply determines the application's reply to calling applications when it is busy.

The **BeginBusyState** and **EndBusyState** calls increment and decrement, respectively, a counter that determines whether the application is busy. For example, two calls to **BeginBusyState** and one call to **EndBusyState** still result in a busy state. To cancel a busy state it is necessary to call **EndBusyState** the same number of times **BeginBusyState** has been called.

By default, the framework enters the busy state during idle processing, which is performed by CWinApp::OnIdle. While the application is handling **ON_COMMANDUPDATEUI** notifications, incoming calls are handled later, after idle processing is complete.

See Also

Reference

COleMessageFilter Class
Hierarchy Chart
COleMessageFilter::EndBusyState
COleMessageFilter::SetBusyReply
CWinApp::OnIdle

Other Resources

COleMessageFilter Members

COleMessageFilter::COleMessageFilter

Creates a **COleMessageFilter** object.

```
COleMessageFilter( );
```

See Also

Reference

COleMessageFilter Class
Hierarchy Chart
COleMessageFilter::Register
COleMessageFilter::Revoke

Other Resources

COleMessageFilter Members

COleMessageFilter::EnableBusyDialog

Enables and disables the busy dialog box, which is displayed when the message-pending delay expires (see SetRetryReply) during an OLE call.

```
void EnableBusyDialog(
   BOOL bEnableBusy = TRUE
);
```

Parameters

bEnableBusy

Specifies whether the "busy" dialog box is enabled or disabled.

See Also

Reference

COleMessageFilter Class
Hierarchy Chart
COleMessageFilter::EnableNotRespondingDialog
COleMessageFilter::BeginBusyState
COleMessageFilter::SetBusyReply
COleMessageFilter::SetRetryReply
COleBusyDialog Class

Other Resources

COleMessageFilter Members

COleMessageFilter::EnableNotRespondingDialog

Enables and disables the "not responding" dialog box, which is displayed if a keyboard or mouse message is pending during an OLE call and the call has timed out.

```
void EnableNotRespondingDialog(
   BOOL bEnableNotResponding = TRUE
);
```

Parameters

bEnableNotResponding

Specifies whether the "not responding" dialog box is enabled or disabled.

See Also

Reference

COleMessageFilter Class
Hierarchy Chart
COleMessageFilter::EnableBusyDialog
COleMessageFilter::BeginBusyState
COleMessageFilter::SetBusyReply
COleBusyDialog Class

Other Resources

COleMessageFilter Members

COleMessageFilter::EndBusyState

Call this function to end a busy state.

```
virtual void EndBusyState( );
```

Remarks

It works in conjunction with BeginBusyState to control the application's busy state. The function SetBusyReply determines the application's reply to calling applications when it is busy.

The **BeginBusyState** and **EndBusyState** calls increment and decrement, respectively, a counter that determines whether the application is busy. For example, two calls to **BeginBusyState** and one call to **EndBusyState** still result in a busy state. To cancel a busy state it is necessary to call **EndBusyState** the same number of times **BeginBusyState** has been called.

By default, the framework enters the busy state during idle processing, which is performed by CWinApp::OnIdle. While the application is handling **ON_UPDATE_COMMAND_UI** notifications, incoming calls are handled after idle processing is complete.

See Also

Reference

COleMessageFilter Class
Hierarchy Chart
COleMessageFilter::BeginBusyState
COleMessageFilter::SetBusyReply
CWinApp::OnIdle

Other Resources

COleMessageFilter Members

COleMessageFilter::OnMessagePending

Called by the framework to process messages while an OLE call is in progress.

```
virtual BOOL OnMessagePending(
   const MSG* pMsg
);
```

Parameters

pMsg

Pointer to the pending message.

Return Value

Nonzero on success; otherwise 0.

Remarks

When a calling application is waiting for a call to be completed, the framework calls **OnMessagePending** with a pointer to the pending message. By default, the framework dispatches **WM_PAINT** messages, so that window updates can occur during a call that is taking a long time.

You must register your message filter by means of a call to Register before it can become active.

See Also

Reference

COleMessageFilter Class
Hierarchy Chart
COleMessageFilter::Register
AfxOleInit
CWinApp::InitInstance

Other Resources

COleMessageFilter Members

COleMessageFilter::Register

Registers the message filter with the OLE system DLLs.

```
BOOL Register( );
```

Return Value

Nonzero on success; otherwise 0.

Remarks

A message filter has no effect unless it is registered with the system DLLs. Usually your application's initialization code registers the application's message filter. Any other message filter registered by your application should be revoked before the program terminates by a call to Revoke.

The framework's default message filter is automatically registered during initialization and revoked at termination.

See Also

Reference

COleMessageFilter Class
Hierarchy Chart
COleMessageFilter::Revoke

Other Resources

COleMessageFilter Members

COleMessageFilter::Revoke

Revokes a previous registration performed by a call to Register.

```
void Revoke( );
```

Remarks

A message filter should be revoked before the program terminates.

The default message filter, which is created and registered automatically by the framework, is also automatically revoked.

See Also

Reference

COleMessageFilter Class
Hierarchy Chart
COleMessageFilter::Register

Other Resources

COleMessageFilter Members

COleMessageFilter::SetBusyReply

This function sets the application's "busy reply."

```
void SetBusyReply(
   SERVERCALL nBusyReply
);
```

Parameters

nBusyReply

A value from the **SERVERCALL** enumeration, which is defined in COMPOBJ.H. It can have any one of the following values:

- **SERVERCALL_ISHANDLED** The application can accept calls but may fail in processing a particular call.
- **SERVERCALL_REJECTED** The application probably will never be able to process a call.
- **SERVERCALL_RETRYLATER** The application is temporarily in a state in which it cannot process a call.

Remarks

The BeginBusyState and EndBusyState functions control the application's busy state.

When an application has been made busy with a call to **BeginBusyState**, it responds to calls from the OLE system DLLs with a value determined by the last setting of **SetBusyReply**. The calling application uses this busy reply to determine what action to take.

By default, the busy reply is **SERVERCALL_RETRYLATER**. This reply causes the calling application to retry the call as soon as possible.

See Also

Reference

COleMessageFilter Class
Hierarchy Chart
COleMessageFilter::BeginBusyState
COleMessageFilter::EndBusyState

Other Resources

COleMessageFilter Members

COleMessageFilter::SetMessagePendingDelay

Determines how long the calling application waits for a response from the called application before taking further action.

```
void SetMessagePendingDelay(
   DWORD nTimeout = 5000
);
```

Parameters

nTimeout

Number of milliseconds for the message-pending delay.

Remarks

This function works in concert with SetRetryReply.

See Also

Reference

COleMessageFilter Class
Hierarchy Chart
COleMessageFilter::SetRetryReply

Other Resources

COleMessageFilter Members

COleMessageFilter::SetRetryReply

Determines the calling application's action when it receives a busy response from a called application.

```
void SetRetryReply(
   DWORD nRetryReply = 0
);
```

Parameters

nRetryReply

Number of milliseconds between retries.

Remarks

When a called application indicates that it is busy, the calling application may decide to wait until the server is no longer busy, to retry right away, or to retry after a specified interval. It may also decide to cancel the call altogether.

The caller's response is controlled by the functions **SetRetryReply** and SetMessagePendingDelay. **SetRetryReply** determines how long the calling application should wait between retries for a given call. **SetMessagePendingDelay** determines how long the calling application waits for a response from the server before taking further action.

Usually the defaults are acceptable and do not need to be changed. The framework retries the call every *nRetryReply* milliseconds until the call goes through or the message-pending delay has expired. A value of 0 for *nRetryReply* specifies an immediate retry, and – 1 specifies cancellation of the call.

When the message-pending delay has expired, the OLE "busy dialog box" (see COleBusyDialog) is displayed so that the user can choose to cancel or retry the call. Call EnableBusyDialog to enable or disable this dialog box.

When a keyboard or mouse message is pending during a call and the call has timed out (exceeded the message-pending delay), the "not responding" dialog box is displayed. Call EnableNotRespondingDialog to enable or disable this dialog box. Usually this state of affairs indicates that something has gone wrong and the user is getting impatient.

When the dialogs are disabled, the current "retry reply" is always used for calls to busy applications.

See Also

Reference

COleMessageFilter Class
Hierarchy Chart
COleBusyDialog Class
COleMessageFilter::EnableNotRespondingDialog
COleMessageFilter::EnableBusyDialog
COleMessageFilter::SetMessagePendingDelay

Other Resources

COleMessageFilter Members

COleObjectFactory Class

Implements the OLE class factory, which creates OLE objects such as servers, automation objects, and documents.

```
class COleObjectFactory : public CCmdTarget
```

Remarks

The **COleObjectFactory** class has member functions for performing the following functions:

- Managing the registration of objects.
- Updating the OLE system register, as well as the run-time registration that informs OLE that objects are running and ready to receive messages.
- Enforcing licensing by limiting use of the control to licensed developers at design time and to licensed applications at run time.
- Registering control object factories with the OLE system registry.

For more information about object creation, see the articles Data Objects and Data Sources (OLE) and Data Objects and Data Sources: Creation and Destruction. For more about registration, see the article Registration.

Requirements

Header: afxdisp.h

See Also

Reference

CCmdTarget Class
Hierarchy Chart
COleTemplateServer Class

Other Resources

COleObjectFactory Members

COleObjectFactory Members

Base Class Members

CObject Members

CCmdTarget Members

Construction

COleObjectFactory	Constructs a **COleObjectFactory** object.

Operations

Register	Registers this object factory with the OLE system DLLs.
RegisterAll	Registers all of the application's object factories with OLE system DLLs.
Revoke	Revokes this object factory's registration with the OLE system DLLs.
RevokeAll	Revokes an application's object factories' registrations with the OLE system DLLs.
UnregisterAll	Unregisters all of an application's object factories.
UpdateRegistryAll	Registers all of the application's object factories with the OLE system registry.

Attributes

GetClassID	Returns the OLE class ID of the objects this factory creates.
IsLicenseValid	Determines if the license of the control is valid.
IsRegistered	Indicates whether the object factory is registered with the OLE system DLLs.

Overridables

GetLicenseKey	Requests a unique key from the control's DLL.
OnCreateObject	Called by the framework to create a new object of this factory's type.
UpdateRegistry	Registers this object factory with the OLE system registry.
VerifyLicenseKey	Verifies that the key embedded in the control matches the key embedded in the container.
VerifyUserLicense	Verifies that the control is licensed for design-time use.

See Also

Reference

COleObjectFactory Class

Hierarchy Chart

COleObjectFactory Member Functions

For information about the member functions in **COleObjectFactory**, see COleObjectFactory Members.

COleObjectFactory::COleObjectFactory

Constructs a **COleObjectFactory** object, initializes it as an unregistered object factory, and adds it to the list of factories.

```
COleObjectFactory(
   REFCLSID clsid,
   CRuntimeClass* pRuntimeClass,
   BOOL bMultiInstance,
   LPCTSTR lpszProgID
);
COleObjectFactory(
   REFCLSID clsid,
   CRuntimeClass* pRuntimeClass,
   BOOL bMultiInstance,
   int nFlags,
   LPCTSTR lpszProgID
);
```

Parameters

clsid

Reference to the OLE class ID this object factory represents.

pRuntimeClass

Pointer to the run-time class of the C++ objects this factory can create.

bMultiInstance

Indicates whether a single instance of the application can support multiple instantiations. If **TRUE**, multiple instances of the application are launched for each request to create an object.

nFlags

Contains one or more of the following flags:

- **afxRegDefault** Sets the threading model to ThreadingModel=Apartment.
- **afxRegInsertable** Allows the control to appear in the **Insert Object** dialog box for OLE objects.
- **afxRegApartmentThreading** Sets the threading model in the registry to ThreadingModel=Apartment.
- **afxRegFreeThreading** Sets the threading model in the registry to ThreadingModel=Free.

  You can combine the two flags **afxRegApartmentThreading** and **afxRegFreeThreading** to set ThreadingModel=Both. See InprocServer32 in the *Platform SDK* for more information on threading model registration.

lpszProgID

Pointer to a string containing a verbal program identifier, such as "Microsoft Excel."

Remarks

To use the object, however, you must register it.

For more information, see CLSID Key in the *Platform SDK*.

See Also

Reference

COleObjectFactory Class
Hierarchy Chart
CRuntimeClass Structure

Other Resources

COleObjectFactory Members

COleObjectFactory::GetClassID

Returns a reference to the OLE class ID this factory represents.

```
REFCLSID GetClassID( ) const;
```

Return Value

Reference to the OLE class ID this factory represents.

Remarks

For more information, see CLSID Key in the *Platform SDK*.

See Also

Reference

COleObjectFactory Class
Hierarchy Chart
COleObjectFactory::COleObjectFactory

Other Resources

COleObjectFactory Members

COleObjectFactory::GetLicenseKey

Requests a unique license key from the control's DLL and stores it in the **BSTR** pointed to by *pbstrKey*.

```
virtual BOOL GetLicenseKey(
   DWORD dwReserved,
   BSTR *pbstrKey
);
```

Parameters

dwReserved

Reserved for future use.

pbstrKey

Pointer to a **BSTR** that will store the license key.

Return Value

Nonzero if the license-key string is not **NULL**; otherwise 0.

Remarks

The default implementation of this function returns 0 and stores nothing in the **BSTR**. If you use MFC ActiveX ControlWizard to create your project, ControlWizard supplies an override that retrieves the control's license key.

See Also

Reference

COleObjectFactory Class
Hierarchy Chart
COleObjectFactory::VerifyUserLicense
COleObjectFactory::VerifyLicenseKey

Other Resources

COleObjectFactory Members

COleObjectFactory::IsLicenseValid

Determines if the license of the control is valid.

```
BOOL IsLicenseValid( );
```

Return Value

TRUE if successul; otherwise false.

See Also

Reference

COleObjectFactory Class
Hierarchy Chart
COleObjectFactory::VerifyUserLicense

Other Resources

COleObjectFactory Members

COleObjectFactory::IsRegistered

Returns a nonzero value if the factory is registered with the OLE system DLLs.

```
virtual BOOL IsRegistered( ) const;
```

Return Value

Nonzero if the factory is registered; otherwise 0.

See Also

Reference

COleObjectFactory Class
Hierarchy Chart
COleObjectFactory::Register
COleObjectFactory::Revoke

Other Resources

COleObjectFactory Members

COleObjectFactory::OnCreateObject

Called by the framework to create a new object.

```
virtual CCmdTarget* OnCreateObject( );
```

Return Value

A pointer to the created object. It can throw a memory exception if it fails.

Remarks

Override this function to create the object from something other than the CRuntimeClass passed to the constructor.

See Also

Reference

COleObjectFactory Class
Hierarchy Chart
COleObjectFactory::COleObjectFactory
CRuntimeClass Structure

Other Resources

COleObjectFactory Members

COleObjectFactory::Register

Registers this object factory with the OLE system DLLs.

```
virtual BOOL Register( );
```

Return Value

Nonzero if the factory is successfully registered; otherwise 0.

Remarks

This function is usually called by CWinApp::InitInstance when the application is launched.

See Also

Reference

COleObjectFactory Class
Hierarchy Chart
COleObjectFactory::Revoke
COleObjectFactory::RegisterAll
CWinApp::InitInstance

Other Resources

COleObjectFactory Members

COleObjectFactory::RegisterAll

Registers all of the application's object factories with the OLE system DLLs.

```
static BOOL PASCAL RegisterAll( );
```

Return Value

Nonzero if the factories are successfully registered; otherwise 0.

Remarks

This function is usually called by CWinApp::InitInstance when the application is launched.

See Also

Reference

COleObjectFactory Class
Hierarchy Chart
COleObjectFactory::Revoke
COleObjectFactory::Register
CWinApp::InitInstance

Other Resources

COleObjectFactory Members

COleObjectFactory::Revoke

Revokes this object factory's registration with the OLE system DLLs.

```
void Revoke( );
```

Remarks

The framework calls this function automatically before the application terminates. If necessary, call it from an override of CWinApp::ExitInstance.

See Also

Reference

COleObjectFactory Class
Hierarchy Chart
COleObjectFactory::RevokeAll
COleObjectFactory::Register
CWinApp::ExitInstance

Other Resources

COleObjectFactory Members

COleObjectFactory::RevokeAll

Revokes all of the application's object factories' registrations with the OLE system DLLs.

```
static void PASCAL RevokeAll( );
```

Remarks

The framework calls this function automatically before the application terminates. If necessary, call it from an override of CWinApp::ExitInstance.

See Also

Reference

COleObjectFactory Class
Hierarchy Chart
COleObjectFactory::Revoke
COleObjectFactory::Register
CWinApp::ExitInstance

Other Resources

COleObjectFactory Members

COleObjectFactory::UnregisterAll

Unregisters all of an application's object factories.

```
static BOOL PASCAL UnregisterAll( );
```

Return Value

TRUE if successful; otherwise FALSE.

See Also

Reference

COleObjectFactory Class
Hierarchy Chart
COleObjectFactory::Register
COleObjectFactory::Register

Other Resources

COleObjectFactory Members

COleObjectFactory::UpdateRegistry

Registers all of the application's object factories with the OLE system registry.

```
void UpdateRegistry(
   LPCTSTR lpszProgID = NULL
);
virtual BOOL UpdateRegistry(
   BOOL bRegister
);
```

Parameters

lpszProgID

Pointer to a string containing the human-readable program identifier, such as "Excel.Document.5."

bRegister

Determines whether the control class's object factory is to be registered.

Remarks

Brief discussions of the two forms for this function follow:

- **UpdateRegistry(** *lpszProgID* **)** Registers this object factory with the OLE system registry. This function is usually called by CWinApp::InitInstance when the application is launched.
- **UpdateRegistry(** *bRegister* **)** This form of the function is overridable. If *bRegister* is **TRUE**, this function registers the control class with the system registry. Otherwise, it unregisters the class.

  If you use MFC ActiveX ControlWizard to create your project, ControlWizard supplies an override to this pure virtual function.

See Also

Reference

COleObjectFactory Class
Hierarchy Chart
COleObjectFactory::Revoke
COleObjectFactory::Register
COleObjectFactory::UpdateRegistryAll
CWinApp::InitInstance

Other Resources

COleObjectFactory Members

COleObjectFactory::UpdateRegistryAll

Registers all of the application's object factories with the OLE system registry.

```
static BOOL PASCAL UpdateRegistryAll(
   BOOL bRegister = TRUE
);
```

Parameters

bRegister

Determines whether the control class's object factory is to be registered.

Return Value

Nonzero if the factories are successfully updated; otherwise 0.

Remarks

This function is usually called by CWinApp::InitInstance when the application is launched.

See Also

Reference

COleObjectFactory Class
Hierarchy Chart
COleObjectFactory::Revoke
COleObjectFactory::Register
COleObjectFactory::UpdateRegistry
CWinApp::InitInstance

Other Resources

COleObjectFactory Members

COleObjectFactory::VerifyLicenseKey

Verifies that the container is licensed to use the OLE control.

```
virtual BOOL VerifyLicenseKey(
   BSTR bstrKey
);
```

Parameters

bstrKey

A **BSTR** storing the container's version of the license string.

Return Value

Nonzero if the run-time license is valid; otherwise 0.

Remarks

The default version calls GetLicenseKey to get a copy of the control's license string and compares it with the string in *bstrKey*. If the two strings match, the function returns a nonzero value; otherwise it returns 0.

You can override this function to provide customized verification of the license.

The function VerifyUserLicense verifies the design-time license.

See Also

Reference

COleObjectFactory Class
Hierarchy Chart
COleObjectFactory::VerifyUserLicense
COleObjectFactory::GetLicenseKey

Other Resources

COleObjectFactory Members

COleObjectFactory::VerifyUserLicense

Verifies the design-time license for the OLE control.

```
virtual BOOL VerifyUserLicense( );
```

Return Value

Nonzero if the design-time license is valid; otherwise 0.

See Also

Reference

COleObjectFactory Class
Hierarchy Chart
COleObjectFactory::VerifyLicenseKey
COleObjectFactory::GetLicenseKey

Other Resources

COleObjectFactory Members

COlePasteSpecialDialog Class

Used for the OLE Paste Special dialog box.

```
class COlePasteSpecialDialog : public COleDialog
```

Remarks

Create an object of class **COlePasteSpecialDialog** when you want to call this dialog box. After a **COlePasteSpecialDialog** object has been constructed, you can use the AddFormat and AddStandardFormats member functions to add Clipboard formats to the dialog box. You can also use the m_ps structure to initialize the values or states of controls in the dialog box. The **m_ps** structure is of type **OLEUIPASTESPECIAL**.

For more information, see the OLEUIPASTESPECIAL structure in the *Platform SDK*.

For more information regarding OLE-specific dialog boxes, see the article Dialog Boxes in OLE.

Requirements

Header: afxodlgs.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

OCLIENT Sample: Illustrates a Visual Editing Container Application

Reference

COleDialog Class

Hierarchy Chart

COleDialog Class

Other Resources

COlePasteSpecialDialog Members

COlePasteSpecialDialog Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CDialog Members

CCommonDialog Members

COleDialog Members

Data Members

m_ps	A structure of type **OLEUIPASTESPECIAL** that controls the function of the dialog box.

Construction

COlePasteSpecialDialog	Constructs a **COlePasteSpecialDialog** object.

Operations and Attributes

AddFormat	Adds custom formats to the list of formats your application can paste.
AddLinkEntry	Adds a new entry to the list of supported Clipboard formats.
AddStandardFormats	Adds **CF_BITMAP**, **CF_DIB**, **CF_METAFILEPICT**, and optionally **CF_LINKSOURCE** to the list of formats yo ur application can paste.
CreateItem	Creates the item in the container document using the specified format.
DoModal	Displays the OLE Paste Special dialog box.
GetDrawAspect	Tells whether to draw item as an icon or not.
GetIconicMetafile	Gets a handle to the metafile associated with the iconic form of this item.
GetPasteIndex	Gets the index of available paste options that was chosen by the user.
GetSelectionType	Gets the type of selection chosen.

See Also

Reference

COlePasteSpecialDialog Class

Hierarchy Chart

COlePasteSpecialDialog Member Functions

For information about the member functions in **COlePasteSpecialDialog**, see COlePasteSpecialDialog Members.

COlePasteSpecialDialog::AddFormat

Call this function to add new formats to the list of formats your application can support in a Paste Special operation.

```
void AddFormat(
   const FORMATETC& formatEtc,
   LPTSTR lpszFormat,
   LPTSTR lpszResult,
   DWORD flags
);
void AddFormat(
   UINT cf,
   DWORD tymed,
   UINT nFormatID,
   BOOL bEnableIcon,
   BOOL bLink
);
```

Parameters

fmt

Reference to the data type to add.

lpszFormat

String that describes the format to the user.

lpszResult

String that describes the result if this format is chosen in the dialog box.

flags

The different linking and embedding options available for this format. This flag is a bitwise combination of one or more of the different values in the **OLEUIPASTEFLAG** enumerated type.

cf

The clipboard format to add.

tymed

The types of media available in this format. This is a bitwise combination of one or more of the values in the **TYMED** enumerated type.

nFormatID

The ID of the string that identifies this format. The format of this string is two separate strings separated by a '\n' character. The first string is the same that would be passed in the *lpstrFormat* parameter, and the second is the same as the *lpstrResult* parameter.

bEnableIcon

Flag that determines whether the Display As Icon check box is enabled when this format is chosen in the list box.

bLink

Flag that determines whether the Paste Link radio button is enabled when this format is chosen in the list box.

Remarks

This function can be called to add either standard formats such as **CF_TEXT** or **CF_TIFF** or custom formats that your application has registered with the system. For more information about pasting data objects into your application, see the article Data Objects and Data Sources: Manipulation.

For more information, see the TYMED enumeration type and the FORMATETC structure in the *Platform SDK*.

For more information, see the OLEUIPASTEFLAG enumerated type in the *Platform SDK*.

See Also

Reference

COlePasteSpecialDialog Class

Hierarchy Chart

COlePasteSpecialDialog::AddStandardFormats

Other Resources

COlePasteSpecialDialog Members

COlePasteSpecialDialog::AddLinkEntry

Adds a new entry to the list of supported Clipboard formats.

```
OLEUIPASTEFLAG AddLinkEntry(
   UINT cf
);
```

Parameters

cf

The clipboard format to add.

Return Value

An OLEUIPASTEFLAG structure containing the information for the new link entry.

See Also

Reference

COlePasteSpecialDialog Class

Hierarchy Chart

Other Resources

COlePasteSpecialDialog Members

COlePasteSpecialDialog::AddStandardFormats

Call this function to add the following Clipboard formats to the list of formats your application can support in a Paste Special operation:

```
void AddStandardFormats(
   BOOL bEnableLink = TRUE
);
```

Parameters

bEnableLink

Flag that determines whether to add **CF_LINKSOURCE** to the list of formats your application can paste.

Remarks

- **CF_BITMAP**
- **CF_DIB**
- **CF_METAFILEPICT**
- **"Embedded Object"**
- (optionally) **"Link Source"**

These formats are used to support embedding and linking.

See Also

Reference

COlePasteSpecialDialog Class
Hierarchy Chart
COlePasteSpecialDialog::AddFormat

Other Resources

COlePasteSpecialDialog Members

COlePasteSpecialDialog::COlePasteSpecialDialog

Constructs a **COlePasteSpecialDialog** object.

```
COlePasteSpecialDialog(
   DWORD dwFlags = PSF_SELECTPASTE,
   COleDataObject* pDataObject = NULL,
   CWnd* pParentWnd = NULL
);
```

Parameters

dwFlags

Creation flag, contains any number of the following flags combined using the bitwise-OR operator:

- **PSF_SELECTPASTE** Specifies that the Paste radio button will be checked initially when the dialog box is called. Cannot be used in combination with **PSF_SELECTPASTELINK**. This is the default.
- **PSF_SELECTPASTELINK** Specifies that the Paste Link radio button will be checked initially when the dialog box is called. Cannot be used in combination with **PSF_SELECTPASTE**.
- **PSF_CHECKDISPLAYASICON** Specifies that the Display As Icon check box will be checked initially when the dialog box is called.
- **PSF_SHOWHELP** Specifies that the Help button will be displayed when the dialog box is called.

pDataObject

Points to the COleDataObject for pasting. If this value is **NULL**, it gets the **COleDataObject** from the Clipboard.

pParentWnd

Points to the parent or owner window object (of type **CWnd**) to which the dialog object belongs. If it is **NULL**, the parent window of the dialog box is set to the main application window.

Remarks

This function only constructs a **COlePasteSpecialDialog** object. To display the dialog box, call the DoModal function.

For more information, see the OLEUIPASTEFLAG enumerated type in the *Platform SDK*.

See Also

Reference

COlePasteSpecialDialog Class
Hierarchy Chart
COleDataObject Class
COlePasteSpecialDialog::DoModal

Other Resources

COlePasteSpecialDialog Members

COlePasteSpecialDialog::CreateItem

Creates the new item that was chosen in the Paste Special dialog box.

```
BOOL CreateItem(
   COleClientItem* pNewItem
);
```

Parameters

pNewItem

Points to a **COleClientItem** instance. Cannot be **NULL**.

Return Value

Nonzero if the item was created successfully; otherwise 0.

Remarks

This function should only be called after DoModal returns **IDOK**.

See Also

Reference

COlePasteSpecialDialog Class
Hierarchy Chart
COleClientItem Class
COlePasteSpecialDialog::DoModal
COlePasteSpecialDialog::GetSelectionType
COlePasteSpecialDialog::COlePasteSpecialDialog

Other Resources

COlePasteSpecialDialog Members

COlePasteSpecialDialog::DoModal

Displays the OLE Paste Special dialog box.

```
virtual INT_PTR DoModal( );
```

Return Value

Completion status for the dialog box. One of the following values:

- **IDOK** if the dialog box was successfully displayed.
- **IDCANCEL** if the user canceled the dialog box.
- **IDABORT** if an error occurred. If **IDABORT** is returned, call the **COleDialog::GetLastError** member function to get more information about the type of error that occurred. For a listing of possible errors, see the OleUIPasteSpecial function in the *Platform SDK*.

Remarks

If you want to initialize the various dialog box controls by setting members of the m_ps structure, you should do this before calling **DoModal**, but after the dialog object is constructed.

If **DoModal** returns **IDOK**, you can call other member functions to retrieve the settings or information input by the user into the dialog box.

See Also

Reference

COlePasteSpecialDialog Class
Hierarchy Chart
COleDataObject Class
COleDialog::GetLastError
CDialog::DoModal
COlePasteSpecialDialog::COlePasteSpecialDialog
COlePasteSpecialDialog::GetDrawAspect
COlePasteSpecialDialog::GetIconicMetafile
COlePasteSpecialDialog::GetPasteIndex
COlePasteSpecialDialog::GetSelectionType

Other Resources

COlePasteSpecialDialog Members

COlePasteSpecialDialog::GetDrawAspect

Determines if the user chose to display the selected item as an icon.

```
DVASPECT GetDrawAspect( ) const;
```

Return Value

The method needed to render the object.

- **DVASPECT_CONTENT** Returned if the Display As Icon check box was not checked when the dialog box was dismissed.
- **DVASPECT_ICON** Returned if the Display As Icon check box was checked when the dialog box was dismissed.

Remarks

Only call this function after DoModal returns **IDOK**.

For more information on drawing aspect, see the FORMATETC structure in the *Platform SDK*.

See Also

Reference

COlePasteSpecialDialog Class
Hierarchy Chart
COlePasteSpecialDialog::DoModal

Other Resources

COlePasteSpecialDialog Members

COlePasteSpecialDialog::GetIconicMetafile

Gets the metafile associated with the item selected by the user.

```
HGLOBAL GetIconicMetafile( ) const;
```

Return Value

The handle to the metafile containing the iconic aspect of the selected item, if the Display As Icon check box was selected when the dialog box was dismissed by choosing **OK**; otherwise **NULL**.

See Also

Reference

COlePasteSpecialDialog Class
Hierarchy Chart
COlePasteSpecialDialog::GetDrawAspect
COlePasteSpecialDialog::DoModal

Other Resources

COlePasteSpecialDialog Members

COlePasteSpecialDialog::GetPasteIndex

Gets the index value associated with the entry the user selected.

```
int GetPasteIndex( ) const;
```

Return Value

The index into the array of **OLEUIPASTEENTRY** structures that was selected by the user. The format that corresponds to the selected index should be used when performing the paste operation.

Remarks

For more information, see the OLEUIPASTEENTRY structure in the *Platform SDK*.

See Also

Reference

COlePasteSpecialDialog Class
Hierarchy Chart
COlePasteSpecialDialog::DoModal

Other Resources

COlePasteSpecialDialog Members

COlePasteSpecialDialog::GetSelectionType

Determines the type of selection the user made.

```
UINT GetSelectionType( ) const;
```

Return Value

Returns type of selection made.

Remarks

The return type values are specified by the **Selection** enumeration type declared in the **COlePasteSpecialDialog** class.

```
enum Selection
{
   pasteLink,
   pasteNormal,
   pasteOther,
   pasteStatic
};
```

Brief desccriptions of these values follow:

- **COlePasteSpecialDialog::pasteLink** The Paste Link radio button was checked and the chosen format was a standard OLE format.
- **COlePasteSpecialDialog::pasteNormal** The Paste radio button was checked and the chosen format was a standard OLE format.
- **COlePasteSpecialDialog::pasteOther** The selected format is not a standard OLE format.
- **COlePasteSpecialDialog::pasteStatic** The chosen format was a metafile.

See Also

Reference

COlePasteSpecialDialog Class
Hierarchy Chart
COlePasteSpecialDialog::DoModal

Other Resources

COlePasteSpecialDialog Members

COlePasteSpecialDialog Data Members

For information about the data members in **COlePasteSpecialDialog**, see COlePasteSpecialDialog Members.

COlePasteSpecialDialog::m_ps

Structure of type **OLEUIPASTESPECIAL** used to control the behavior of the Paste Special dialog box.

```
OLEUIPASTESPECIAL m_ps;
```

Remarks

Members of this structure can be modified directly or through member functions.

For more information, see the OLEUIPASTESPECIAL structure in the *Platform SDK*.

See Also

Reference

COlePasteSpecialDialog Class
Hierarchy Chart
COlePasteSpecialDialog::COlePasteSpecialDialog
COlePasteSpecialDialog::DoModal

Other Resources

COlePasteSpecialDialog Members

COlePropertiesDialog Class

Encapsulates the Windows common OLE Object Properties dialog box.

```
class COlePropertiesDialog : public COleDialog
```

Remarks

Common OLE Object Properties dialog boxes provide an easy way to display and modify the properties of an OLE document item in a manner consistent with Windows standards. These properties include, among others, information on the file represented by the document item, options for displaying the icon and image scaling, and information on the item's link (if the item is linked).

To use a **COlePropertiesDialog** object, first create the object using the **COlePropertiesDialog** constructor. After the dialog box has been constructed, call the **DoModal** member function to display the dialog box and allow the user to modify any properties of the item. **DoModal** returns whether the user selected the OK (**IDOK**) or the Cancel (**IDCANCEL**) button. In addition to the OK and Cancel buttons, there is an Apply button. When the user selects Apply, any changes made to the properties of the document item are applied to the item and its image is automatically updated, but remains active.

The m_psh data member is a pointer to a **PROPSHEETHEADER** structure, and in most cases you will not need to access it explicitly. One exception is when you need additional property pages beyond the default General, View, and Link pages. In this case, you can modify the **m_psh** data member to include your custom pages before calling the **DoModal** member function.

For more information on OLE dialog boxes, see the article Dialog Boxes in OLE..

Requirements

Header: afxodlgs.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

CIRC Sample: ActiveX Control

Reference

COleDialog Class

Hierarchy Chart

COleDialog Class

CPropertyPage Class

Other Resources

COlePropertiesDialog Members

COlePropertiesDialog Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CDialog Members

CCommonDialog Members

COleDialog Members

Construction

COlePropertiesDialog	Constructs a **COlePropertiesDialog** object.

Data Members

m_gp	A structure used to initialize the "General" page of a **COlePropertiesDialog** object.
m_lp	A structure used to initialize the "Link" page of a **COlePropertiesDialog** object.
m_op	A structure used to initialize the **COlePropertiesDialog** object.
m_psh	A structure used to add additional custom property pages.
m_vp	A structure used to customize the "View" page of a **COlePropertiesDialog** object.

Operations

DoModal	Displays the dialog box and allows the user to make a selection.

Overridables

OnApplyScale	Called by the framework when the scaling of the document item has changed.

See Also

Reference

COlePropertiesDialog Class

Hierarchy Chart

COlePropertiesDialog Member Functions

For information about the member functions in **COlePropertiesDialog**, see COlePropertiesDialog Members.

COlePropertiesDialog::COlePropertiesDialog

Creates a **COlePropertiesDialog** object.

```
COlePropertiesDialog(
   COleClientItem* pItem,
   UINT nScaleMin = 10,
   UINT nScaleMax = 500,
   CWnd* pParentWnd = NULL
);
```

Parameters

pItem

Pointer to the document item whose properties are being accessed.

nScaleMin

Minimum scaling percentage for the document item image.

nScaleMax

Maximum scaling percentage for the document item image.

pParentWnd

Pointer to the dialog box's parent or owner.

Remarks

Derive your common OLE Object Properties dialog class from **COlePropertiesDialog** in order to implement scaling for your document items. Any dialog boxes implemented by an instance of this class will not support scaling of the document item.

By default, the common OLE Object Properties dialog box has three default pages:

- General

  This page contains system information for the file represented by the selected document item. From this page, the user can convert the selected item to another type.

- View

  This page contains options for displaying the item, changing the icon, and changing the scaling of the image.

- Link

  This page contains options for changing the location of the linked item and updating the linked item. From this page, the user can break the link of the selected item.

To add pages beyond those provided by default, modify the m_psh member variable before exiting the constructor of your **COlePropertiesDialog**-derived class. This is an advanced implementation of the **COlePropertiesDialog** constructor.

See Also

Reference

COlePropertiesDialog Class
Hierarchy Chart
COlePropertiesDialog::OnApplyScale

Other Resources

COlePropertiesDialog Members

COlePropertiesDialog::DoModal

Call this member function to display the Windows common OLE Object Properties dialog box and allow the user to view and/or change the various properties of the document item.

```
virtual INT_PTR DoModal( );
```

Return Value

IDOK or **IDCANCEL** if successful; otherwise 0. **IDOK** and **IDCANCEL** are constants that indicate whether the user selected the OK or Cancel button.

If **IDCANCEL** is returned, you can call the Windows CommDlgExtendedError function to determine whether an error occurred.

See Also

Reference

COlePropertiesDialog Class
Hierarchy Chart
COlePropertiesDialog::OnApplyScale
COlePropertiesDialog::m_psh

Other Resources

COlePropertiesDialog Members

COlePropertiesDialog::OnApplyScale

Called by the framework when the scaling value has changed and either OK or Apply was selected.

```
virtual BOOL OnApplyScale(
   COleClientItem* pItem,
   int nCurrentScale,
   BOOL bRelativeToOrig
);
```

Parameters

pItem

Pointer to the document item whose properties are being accessed.

nCurrentScale

Numerical value of the dialog scale.

bRelativeToOrig

Indicates whether scaling applies to the original size of the document item.

Return Value

Nonzero if handled; otherwise 0.

Remarks

The default implementation does nothing. You must override this function to enable the scaling controls.

Note
Before the common OLE Object Properties dialog box is displayed, the framework calls this function with a **NULL** for *pItem* and a – 1 for *nCurrentScale*. This is done to determine if the scaling controls should be enabled.

See Also

Reference

COlePropertiesDialog Class
Hierarchy Chart
COlePropertiesDialog::DoModal

Other Resources

COlePropertiesDialog Members

COlePropertiesDialog Data Members

For information about the data members in **COlePropertiesDialog**, see COlePropertiesDialog Members.

COlePropertiesDialog::m_gp

A structure of type OLEUIGNRLPROPS, used to initialize the General page of the OLE Object Properties dialog box.

```
OLEUIGNRLPROPS m_gp;
```

Remarks

This page shows the type and size of an embedding and allows the user access to the Convert dialog box. This page also shows the link destination if the object is a link.

For more information on the **OLEUIGNRLPROPS** structure, see the *Platform SDK*.

See Also

Reference

COlePropertiesDialog Class

Hierarchy Chart

Other Resources

COlePropertiesDialog Members

COlePropertiesDialog::m_lp

A structure of type OLEUILINKPROPS, used to initialize the Link page of the OLE Object Properties dialog box.

```
OLEUILINKPROPS m_lp;
```

Remarks

This page shows the location of the linked item and allows the user to update, or break, the link to the item.

For more information on the **OLEUILINKPROPS** structure, see the *Platform SDK*.

See Also

Reference

COlePropertiesDialog Class

Hierarchy Chart

Other Resources

COlePropertiesDialog Members

COlePropertiesDialog::m_op

A structure of type OLEUIOBJECTPROPS, used to initialize the common OLE Object Properties dialog box.

```
OLEUIOBJECTPROPS m_op;
```

Remarks

This structure contains members used to initialize the General, Link, and View pages.

For more information, see the **OLEUIOBJECTPROPS** and OLEUILINKPROPS structures in the *Platform SDK*.

See Also

Reference

COlePropertiesDialog Class

Hierarchy Chart

Other Resources

COlePropertiesDialog Members

COlePropertiesDialog::m_psh

A structure of type PROPSHEETHEADER, whose members store the characteristics of the dialog object.

```
PROPSHEETHEADER m_psh;
```

Remarks

After constructing a **COlePropertiesDialog** object, you can use **m_psh** to set various aspects of the dialog box before calling the **DoModal** member function.

If you modify the **m_psh** data member directly, you will override any default behavior.

For more information on the **PROPSHEETHEADER** structure, see the *Platform SDK*.

See Also

Reference

COlePropertiesDialog Class
Hierarchy Chart
COlePropertiesDialog::DoModal

Other Resources

COlePropertiesDialog Members

COlePropertiesDialog::m_vp

A structure of type OLEUIVIEWPROPS, used to initialize the View page of the OLE Object Properties dialog box.

```
OLEUIVIEWPROPS m_vp;
```

Remarks

This page allows the user to toggle between "content" and "iconic" views of the object, and change its scaling within the container. It also allows the user access to the Change Icon dialog box when the object is being displayed as an icon.

For more information on the **OLEUIVIEWPROPS** structure, see the *Platform SDK*.

See Also

Reference

COlePropertiesDialog Class

Hierarchy Chart

Other Resources

COlePropertiesDialog Members

COlePropertyPage Class

Used to display the properties of a custom control in a graphical interface, similar to a dialog box.

```
class AFX_NOVTABLE COlePropertyPage : public CDialog
```

Remarks

For instance, a property page may include an edit control that allows the user to view and modify the control's caption property.

Each custom or stock control property can have a dialog control that allows the control's user to view the current property value and modify that value if needed.

For more information on using **COlePropertyPage**, see the article ActiveX Controls: Property Pages.

Requirements

Header: afxctl.h

See Also

Tasks

CIRC Sample: ActiveX Control

TESTHELP Sample: ActiveX Control with Tooltips and Help

Reference

CDialog Class

Hierarchy Chart

CDialog Class

Other Resources

COlePropertyPage Members

COlePropertyPage Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CDialog Members

Construction

COlePropertyPage	Constructs a **COlePropertyPage** object.

Operations

GetControlStatus	Indicates whether the user has modified the value in the control.
GetObjectArray	Returns the array of objects being edited by the property page.
GetPageSite	Returns a pointer to the property page's **IPropertyPageSite** interface.
IgnoreApply	Determines which controls do not enable the Apply button.
IsModified	Indicates whether the user has modified the property page.
SetControlStatus	Sets a flag indicating whether the user has modified the value in the control.
SetDialogResource	Sets the property page's dialog resource.
SetHelpInfo	Sets the property page's brief help text, the name of its help file, and its help context.
SetModifiedFlag	Sets a flag indicating whether the user has modified the property page.
SetPageName	Sets the property page's name (caption).

Overridables

OnEditProperty	Called by the framework when the user edits a property.
OnHelp	Called by the framework when the user invokes help.
OnInitDialog	Called by the framework when the property page is initialized.
OnObjectsChanged	Called by the framework when another OLE control, with new properties, is chosen.
OnSetPageSite	Called by the framework when the property frame provides the page's site.

See Also

Reference

COlePropertyPage Class

Hierarchy Chart

COlePropertyPage Member Functions

For information about the member functions in **COlePropertyPage**, see COlePropertyPage Members.

COlePropertyPage::COlePropertyPage

Constructs a **COlePropertyPage** object.

```
COlePropertyPage(
   UINT idDlg,
   UINT idCaption
);
```

Parameters

idDlg

Resource ID of the dialog template.

idCaption

Resource ID of the property page's caption.

Remarks

When you implement a subclass of **COlePropertyPage**, your subclass's constructor should use the **COlePropertyPage** constructor to identify the dialog-template resource on which the property page is based and the string resource containing its caption.

See Also

Reference

COlePropertyPage Class

Hierarchy Chart

Other Resources

COlePropertyPage Members

COlePropertyPage::GetControlStatus

Determines whether the user has modified the value of the property page control with the specified resource ID.

```
BOOL GetControlStatus(
   UINT nID
);
```

Parameters

nID

Resource ID of a property page control.

Return Value

TRUE if the control value has been modified; otherwise **FALSE**.

See Also

Reference

COlePropertyPage Class

Hierarchy Chart

COlePropertyPage::SetControlStatus

Other Resources

COlePropertyPage Members

COlePropertyPage::GetObjectArray

Returns the array of objects being edited by the property page.

```
LPDISPATCH* GetObjectArray(
   ULONG* pnObjects
);
```

Parameters

pnObjects

Pointer to an unsigned long integer that will receive the number of objects being edited by the page.

Return Value

Pointer to an array of **IDispatch** pointers, which are used to access the properties of each control on the property page. The caller must not release these interface pointers.

Remarks

Each property page object maintains an array of pointers to the **IDispatch** interfaces of the objects being edited by the page. This function sets its *pnObjects* argument to the number of elements in that array and returns a pointer to the first element of the array.

See Also

Reference

COlePropertyPage Class

Hierarchy Chart

Other Resources

COlePropertyPage Members

COlePropertyPage::GetPageSite

Gets a pointer to the property page's **IPropertyPageSite** interface.

```
LPPROPERTYPAGESITE GetPageSite( );
```

Return Value

A pointer to the property page's **IPropertyPageSite** interface.

Remarks

Controls and containers cooperate so that users can browse and edit control properties. The control provides property pages, each of which is an OLE object that allows the user to edit a related set of properties. The container provides a property frame that displays the property pages. For each page, the property frame provides a page site, which supports the **IPropertyPageSite** interface.

See Also

Reference

COlePropertyPage Class
Hierarchy Chart
COlePropertyPage::OnSetPageSite

Other Resources

COlePropertyPage Members

COlePropertyPage::IgnoreApply

Determines which controls do not enable the Apply button.

```
void IgnoreApply(
   UINT nID
);
```

Parameters

nID

ID of the control to be ignored.

Remarks

The property page's Apply button is enabled only when values of property page controls have been changed. Use this function to specify controls that do not cause the Apply button to be enabled when their values change.

See Also

Reference

COlePropertyPage Class
Hierarchy Chart
COlePropertyPage::GetControlStatus

Other Resources

COlePropertyPage Members

COlePropertyPage::IsModified

Determines whether the user has changed any values on the property page.

```
BOOL IsModified( );
```

Return Value

TRUE if the property page has been modified.

See Also

Reference

COlePropertyPage Class
Hierarchy Chart
COlePropertyPage::SetModifiedFlag

Other Resources

COlePropertyPage Members

COlePropertyPage::OnEditProperty

The framework calls this function when a specific property is to be edited.

```
virtual BOOL OnEditProperty(
   DISPID dispid
);
```

Parameters

dispid

Dispatch ID of the property being edited.

Return Value

The default implementation returns **FALSE**. Overrides of this function should return **TRUE**.

Remarks

You can override it to set the focus to the appropriate control on the page. The default implementation does nothing and returns **FALSE**.

See Also

Reference

COlePropertyPage Class

Hierarchy Chart

Other Resources

COlePropertyPage Members

COlePropertyPage::OnHelp

The framework calls this function when the user requests online help.

```
virtual BOOL OnHelp(
   LPCTSTR lpszHelpDir
);
```

Parameters

lpszHelpDir

Directory containing the property page's help file.

Return Value

The default implementation returns **FALSE**.

Remarks

Override it if your property page must perform any special action when the user accesses help. The default implementation does nothing and returns **FALSE**, which instructs the framework to call WinHelp.

See Also

Reference

COlePropertyPage Class

Hierarchy Chart

Other Resources

COlePropertyPage Members

COlePropertyPage::OnInitDialog

The framework calls this function when the property page's dialog is initialized.

```
virtual BOOL OnInitDialog( );
```

Return Value

The default implementation returns **FALSE**.

Remarks

Override it if any special action is required when the dialog is initialized. The default implementation calls **CDialog::OnInitDialog** and returns **FALSE**.

See Also

Reference

COlePropertyPage Class
Hierarchy Chart
CDialog::OnInitDialog

Other Resources

COlePropertyPage Members

COlePropertyPage::OnObjectsChanged

Called by the framework when another OLE control, with new properties, is chosen.

```
virtual void OnObjectsChanged( );
```

Remarks

When viewing the properties of an OLE control in the developer environment, a modeless dialog box is used to display its property pages. If another control is selected, a different set of property pages must be displayed for the new set of properties. The framework calls this function to notify the property page of the change.

Override this function to receive notification of this action and perform any special actions.

See Also

Reference

COlePropertyPage Class

Hierarchy Chart

Other Resources

COlePropertyPage Members

COlePropertyPage::OnSetPageSite

The framework calls this function when the property frame provides the property page's page site.

```
virtual void OnSetPageSite( );
```

Remarks

The default implementation loads the page's caption and attempts to determine the page's size from the dialog resource. Override this function if your property page requires any further action; your override should call the base-class implementation.

See Also

Reference

COlePropertyPage Class
Hierarchy Chart
COlePropertyPage::GetPageSite

Other Resources

COlePropertyPage Members

COlePropertyPage::SetControlStatus

Changes the status of a property page control.

```
BOOL SetControlStatus(
   UINT nID,
   BOOL bDirty
);
```

Parameters

nID

Contains the ID of a property page control.

bDirty

Specifies if a field of the property page has been modified. Set to **TRUE** if the field has been modified, **FALSE** if it has not been modified.

Return Value

TRUE, if the specified control was set; otherwise **FALSE**.

Remarks

If the status of a property page control is dirty when the property page is closed or the Apply button is chosen, the control's property will be updated with the appropriate value.

See Also

Reference

COlePropertyPage Class
Hierarchy Chart
COlePropertyPage::GetControlStatus

Other Resources

COlePropertyPage Members

COlePropertyPage::SetDialogResource

Sets the property page's dialog resource.

```
void SetDialogResource(
   HGLOBAL hDialog
);
```

Parameters

hDialog

Handle to the property page's dialog resource.

See Also

Reference

COlePropertyPage Class

Hierarchy Chart

Other Resources

COlePropertyPage Members

COlePropertyPage::SetHelpInfo

Specifies tooltip information, the help filename, and the help context for your property page.

```
void SetHelpInfo(
   LPCTSTR lpszDocString,
   LPCTSTR lpszHelpFile = NULL,
   DWORD dwHelpContext = 0
);
```

Parameters

lpszDocString

A string containing brief help information for display in a status bar or other location.

lpszHelpFile

Name of the property page's help file.

dwHelpContext

Help context for the property page.

See Also

Reference

COlePropertyPage Class
Hierarchy Chart
COlePropertyPage::OnHelp

Other Resources

COlePropertyPage Members

COlePropertyPage::SetModifiedFlag

Indicates whether the user has modified the property page.

```
void SetModifiedFlag(
   BOOL bModified = TRUE
);
```

Parameters

bModified

Specifies the new value for the property page's modified flag.

See Also

Reference

COlePropertyPage Class

Hierarchy Chart

COlePropertyPage::IsModified

Other Resources

COlePropertyPage Members

COlePropertyPage::SetPageName

Sets the property page's name, which the property frame will typically display on the page's tab.

```
void SetPageName(
   LPCTSTR lpszPageName
);
```

Parameters

lpszPageName

Pointer to a string containing the property page's name.

See Also

Reference

COlePropertyPage Class

Hierarchy Chart

Other Resources

COlePropertyPage Members

COleResizeBar Class

A type of control bar that supports resizing of in-place OLE items.

```
class COleResizeBar : public CControlBar
```

Remarks

COleResizeBar objects appear as a CRectTracker with a hatched border and outer resize handles.

COleResizeBar objects are usually embedded members of frame-window objects derived from the COleIPFrameWnd class.

For more information, see the article Activation..

Requirements

Header: afxole.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

SUPERPAD Sample: Demonstrates a Visual Editing Server That Edits Text Using CEditView

Reference

CControlBar Class

Hierarchy Chart

COleServerDoc Class

Other Resources

COleResizeBar Members

COleResizeBar Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CControlBar Members

Construction

COleResizeBar	Constructs a **COleResizeBar** object.
Create	Creates and initializes a Windows child window and associates it to the **COleResizeBar** object.

See Also

Reference

COleResizeBar Class

Hierarchy Chart

COleResizeBar Member Functions

For information about the member functions in **COleResizeBar**, see COleResizeBar Members.

COleResizeBar::COleResizeBar

Constructs a **COleResizeBar** object.

```
COleResizeBar( );
```

Remarks

Call **Create** to create the resize bar object.

See Also

Reference

COleResizeBar Class
Hierarchy Chart
COleResizeBar::Create

Other Resources

COleResizeBar Members

COleResizeBar::Create

Creates a child window and associates it with the **COleResizeBar** object.

```
virtual BOOL Create(
   CWnd* pParentWnd,
   DWORD dwStyle = WS_CHILD | WS_VISIBLE,
   UINT nID = AFX_IDW_RESIZE_BAR
);
```

Parameters

pParentWnd

Pointer to the parent window of the resize bar.

dwStyle

Specifies the window style attributes.

nID

The resize bar's child window ID.

Return Value

Nonzero if the resize bar was created; otherwise 0.

See Also

Reference

COleResizeBar Class
Hierarchy Chart
CWnd::Create
CControlBar Class

Other Resources

COleResizeBar Members

COleSafeArray Class

This topic has been updated for Visual Studio 2005 SP1.

A class for working with arrays of arbitrary type and dimension.

```
class COleSafeArray : public tagVARIANT
```

Remarks

COleSafeArray derives from the OLE **VARIANT** structure. The OLE **SAFEARRAY** member functions are available through **COleSafeArray**, as well as a set of member functions specifically designed for one-dimensional arrays of bytes.

Requirements

Header: afxdisp.h

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

The Smart Device version of COleSafeArray Class does not support the following operator:

- COleSafeArray::operator <<

See Also

Reference

Hierarchy Chart
COleVariant Class
CRecordset Class
CDatabase Class

Other Resources

COleSafeArray Members

COleSafeArray Members

Construction

COleSafeArray	Constructs a **COleSafeArray** object.

Operations

Attach	Gives control of the existing **VARIANT** array to the **COleSafeArray** object.
Clear	Frees all data in the underlying **VARIANT**.
Detach	Detaches the **VARIANT** array from the **COleSafeArray** object (so that the data will not be freed).

Win32 API Wrappers

AccessData	Retrieves a pointer to the array data.
AllocData	Allocates memory for the array.
AllocDescriptor	Allocates memory for the safe array descriptor.
Copy	Creates a copy of an existing array.
Create	Creates a safe array.
Destroy	Destroys an existing array.
DestroyData	Destroys data in a safe array.
DestroyDescriptor	Destroys a descriptor of a safe array.
GetDim	Returns the number of dimensions in the array.
GetByteArray	Copies the contents of the safe array into a CByteArray.
GetElement	Retrieves a single element of the safe array.
GetElemSize	Returns the size, in bytes, of one element in a safe array.
GetLBound	Returns the lower bound for any dimension of a safe array.
GetUBound	Returns the upper bound for any dimension of a safe array.
Lock	Increments the lock count of an array and places a pointer to the array data in the array descriptor.
PtrOfIndex	Returns a pointer to the indexed element.
PutElement	Assigns a single element into the array.
Redim	Changes the least significant (rightmost) bound of a safe array.
UnaccessData	Decrements the lock count of an array and invalidates the pointer retrieved by **AccessData**.

Unlock	Decrements the lock count of an array so it can be freed or resized.

One-Dimensional Array Operations

CreateOneDim	Creates a one-dimensional **COleSafeArray** object.
GetOneDimSize	Returns the number of elements in the one-dimensional **COleSafeArray** object.
ResizeOneDim	Changes the number of elements in a one-dimensional **COleSafeArray** object.

Operators

operator <<	Outputs the contents of a **COleSafeArray** object to the dump context.
operator =	Copies values into a **COleSafeArray** object (**SAFEARRAY**, **VARIANT**, **COleVariant**, or **COleSafeArray** array).
operator ==	Compares two variant arrays (**SAFEARRAY**, **VARIANT**, **COleVariant**, or **COleSafeArray** arrays).
operator LPCVARIANT	Accesses the underlying **VARIANT** structure of the **COleSafeArray** object.
operator LPVARIANT	Accesses the underlying **VARIANT** structure of the **COleSafeArray** object.

See Also

Reference

COleSafeArray Class

Hierarchy Chart

COleSafeArray Member Functions

For information about the member functions in **COleSafeArray**, see COleSafeArray Members.

COleSafeArray::AccessData

Retrieves a pointer to the array data.

```
void AccessData(
   void** ppvData
);
```

Parameters

ppvData

A pointer to a pointer to the array data.

Remarks

On error, the function throws a CMemoryException or COleException.

Example

```
//Sort is an automation method by the CCmdTarget-derived class
// CSAProjDlgAutoProxy
void CSAProjDlgAutoProxy::Sort(VARIANT FAR* vArray)
{
   COleSafeArray sa;
   BSTR *pbstr;
   TCHAR buf[1024];
   LONG cElements, lLBound, lUBound;

   //needed for OLE2T macro below, include afxpriv.h
   USES_CONVERSION;

   // Type check VARIANT parameter. It should contain a BSTR array
   // passed by reference. The array must be passed by reference it is
   // an in-out-parameter.
   if (V_VT(vArray) != (VT_ARRAY | VT_BSTR))
      AfxThrowOleDispatchException(1001,
         "Type Mismatch in Parameter. Pass a string array by reference");

   // clears data in sa and copies the variant data into sa
   sa.Attach(*vArray);

   // Check that array is 1 dimensional
   if (sa.GetDim() != 1)
      AfxThrowOleDispatchException(1002,
         "Type Mismatch in Parameter. Pass a one-dimensional array");

   try
   {
      // Get array bounds.
      sa.GetLBound(1, &lLBound);
      sa.GetUBound(1, &lUBound);

      // Get a pointer to the elements of the array
      // and increments the lock count on the array
      sa.AccessData((LPVOID*)&pbstr);

      //get no. of elements in array
      cElements = lUBound-lLBound+1;
      for (int i = 0; i < cElements-1; i++)
      {
         //output the elements of the array
         wsprintf(buf, "%s", OLE2T(pbstr[i]));
         OutputDebugString(buf);
      }
```

```
        //decrement lock count
        sa.UnaccessData();
    }
    catch (COleException *pEx)
    {
        AfxThrowOleDispatchException(1003,
            "Unexpected Failure in FastSort method");
        pEx->Delete();
    }
}
```

See Also

Reference

COleSafeArray Class

Hierarchy Chart

COleSafeArray::UnaccessData

SafeArrayAccessData

Other Resources

COleSafeArray Members

COleSafeArray::AllocData

Allocates memory for a safe array.

```
void AllocData( );
```

Remarks

On error, the function throws a CMemoryException or COleException.

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::AllocDescriptor
SafeArrayAllocData

Other Resources

COleSafeArray Members

COleSafeArray::AllocDescriptor

Allocates memory for the descriptor of a safe array.

```
void AllocDescriptor(
   DWORD dwDims
);
```

Parameters

dwDims

Number of dimensions in the safe array.

Remarks

On error, the function throws a CMemoryException or COleException.

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::AllocData
SafeArrayAllocDescriptor

Other Resources

COleSafeArray Members

COleSafeArray::Attach

Gives control of the data in an existing **VARIANT** array to the **COleSafeArray** object.

```
void Attach(
   VARIANT& varSrc
);
```

Parameters

varSrc

A **VARIANT** object. The *varSrc* parameter must have the VARTYPE **VT_ARRAY**.

Remarks

The source **VARIANT**'s type is set to **VT_EMPTY**. This function clears the current array data, if any.

Example

See the example for COleSafeArray::AccessData.

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::Detach

Other Resources

COleSafeArray Members

COleSafeArray::Clear

Clears the safe array.

```
void Clear( );
```

Remarks

The function clears a safe array by setting the VARTYPE of the object to **VT_EMPTY**. The current contents are released and the array is freed.

See Also

Reference

COleSafeArray Class
Hierarchy Chart
VariantClear

Other Resources

COleSafeArray Members

COleSafeArray::COleSafeArray

Constructs a **COleSafeArray** object.

```
COleSafeArray( );
COleSafeArray(
   const SAFEARRAY& saSrc,
      VARTYPE vtSrc
);
COleSafeArray(
   LPCSAFEARRAY pSrc,
      VARTYPE vtSrc
);
COleSafeArray(
   const COleSafeArray& saSrc
);
COleSafeArray(
   const VARIANT& varSrc
);
COleSafeArray(
   LPCVARIANT pSrc
);
COleSafeArray(
   const COleVariant& varSrc
);
```

Parameters

saSrc

An existing **COleSafeArray** object or **SAFEARRAY** to be copied into the new **COleSafeArray** object.

vtSrc

The **VARTYPE** of the new **COleSafeArray** object.

psaSrc

A pointer to a **SAFEARRAY** to be copied into the new **COleSafeArray** object.

varSrc

An existing **VARIANT** or **COleVariant** object to be copied into the new **COleSafeArray** object.

pSrc

A pointer to a **VARIANT** object to be copied into the new **COleSafeArray** object.

Remarks

All of these constructors create new **COleSafeArray** objects. If there is no parameter, an empty **COleSafeArray** object is created (**VT_EMPTY**). If the **COleSafeArray** is copied from another array whose VARTYPE is known implicitly (a **COleSafeArray**, **COleVariant**, or **VARIANT**), the **VARTYPE** of the source array is retained and need not be specified. If the **COleSafeArray** is copied from another array whose **VARTYPE** is not known (**SAFEARRAY**), the **VARTYPE** must be specified in the *vtSrc* parameter.

On error, the function throws a CMemoryException or COleException.

See Also

Reference

COleSafeArray Class
Hierarchy Chart
VariantCopy

Other Resources

COleSafeArray Members

COleSafeArray::Copy

Creates a copy of an existing safe array.

```
void Copy(
   LPSAFEARRAY* ppsa
);
```

Parameters

ppsa

Pointer to a location in which to return the new array descriptor.

Remarks

On error, the function throws a CMemoryException or COleException.

See Also

Reference

COleSafeArray Class
Hierarchy Chart
SafeArrayCopy

Other Resources

COleSafeArray Members

COleSafeArray::Create

Allocates and initializes the data for the array.

```
void Create(
   VARTYPE vtSrc,
   DWORD dwDims,
   DWORD* rgElements
);
void Create(
   VARTYPE vtSrc,
   DWORD dwDims,
   SAFEARRAYBOUND* rgsabounds
);
```

Parameters

vtSrc

The base type of the array (that is, the **VARTYPE** of each element of the array). The **VARTYPE** is restricted to a subset of the variant types. Neither the **VT_ARRAY** nor the **VT_BYREF** flag can be set. **VT_EMPTY** and **VT_NULL** are not valid base types for the array. All other types are legal.

dwDims

Number of dimensions in the array. This can be changed after the array is created with Redim.

rgElements

Pointer to an array of the number of elements for each dimension in the array.

rgsabounds

Pointer to a vector of bounds (one for each dimension) to allocate for the array.

Remarks

This function will clear the current array data if necessary. On error, the function throws a CMemoryException.

Example

```
COleSafeArray saMatrix;
DWORD numElements[] = {10, 5};

// creates a 2 dimensional safearray of type VT_I2
// with size 10x5 elements, with all indices starting at 0(default)
saMatrix.Create(VT_I2, 2, numElements);

ASSERT(saMatrix.GetDim() == 2);

COleSafeArray saVector;
SAFEARRAYBOUND rgsabounds[] = { {5, 2} };

// creates a 1 dimensional safearray of type VT_I1
// with size 5 elements, with the index starting at 2
saVector.Create(VT_I1, 1, rgsabounds);

ASSERT(saVector.GetDim() == 1);
```

See Also

Reference

COleSafeArray Class
Hierarchy Chart
SafeArrayCreate

Other Resources

COleSafeArray Members

COleSafeArray::CreateOneDim

Creates a new one-dimensional **COleSafeArray** object.

```
void CreateOneDim(
   VARTYPE vtSrc,
   DWORD dwElements,
   const void* pvSrcData = NULL,
   long nLBound = 0
);
```

Parameters

vtSrc

The base type of the array (that is, the **VARTYPE** of each element of the array).

dwElements

Number of elements in the array. This can be changed after the array is created with ResizeOneDim.

pvSrcData

Pointer to the data to copy into the array.

nLBound

The lower bound of the array.

Remarks

The function allocates and initializes the data for the array, copying the specified data if the pointer *pvSrcData* is not **NULL**.

On error, the function throws a CMemoryException.

Example

```
VARIANT varColInfo[3];

//initialize VARIANTs
for (int i = 0; i < 3; i++)
   VariantInit(&varColInfo[i]);

// Column Name
varColInfo[0].vt = VT_BSTR;
varColInfo[0].bstrVal = ::SysAllocString(L"Name");

// Column Type
varColInfo[1].vt = VT_UI4;
varColInfo[1].lVal = 1;

COleSafeArray sa;
//create a 1 dimensional safearray of VARIANTs
//& initialize it with varColInfo VARIANT array
sa.CreateOneDim(VT_VARIANT, 2, varColInfo);

//check that the dimension is 2
ASSERT(sa.GetOneDimSize() == 2);

//increase safearray size by 1
sa.ResizeOneDim(3);

// populate the last element of the safearray, (Column Size)
varColInfo[2].vt = VT_I4;
varColInfo[2].lVal = 30;
```

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::GetOneDimSize
COleSafeArray::ResizeOneDim
COleSafeArray::Create

Other Resources

COleSafeArray Members

COleSafeArray::Destroy

Destroys an existing array descriptor and all the data in the array.

```
void Destroy( );
```

Remarks

If objects are stored in the array, each object is released. On error, the function throws a CMemoryException or COleException.

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::DestroyData
COleSafeArray::DestroyDescriptor
SafeArrayDestroy

Other Resources

COleSafeArray Members

COleSafeArray::DestroyData

Destroys all the data in a safe array.

```
void DestroyData( );
```

Remarks

If objects are stored in the array, each object is released. On error, the function throws a CMemoryException or COleException.

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::Destroy
COleSafeArray::DestroyDescriptor
SafeArrayDestroyData

Other Resources

COleSafeArray Members

COleSafeArray::DestroyDescriptor

Destroys a descriptor of a safe array.

```
void DestroyDescriptor( );
```

Remarks

On error, the function throws a CMemoryException or COleException.

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::Destroy
COleSafeArray::DestroyData
SafeArrayDestroyDescriptor

Other Resources

COleSafeArray Members

COleSafeArray::Detach

Detaches the **VARIANT** data from the **COleSafeArray** object.

```
VARIANT Detach( );
```

Return Value

The underlying **VARIANT** value in the **COleSafeArray** object.

Remarks

The function detaches the data in a safe array by setting the VARTYPE of the object to **VT_EMPTY**. It is the caller's responsibility to free the array by calling the Windows function VariantClear.

On error, the function throws a COleException.

Example

See the example for COleSafeArray::PutElement.

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::Attach
VariantClear

Other Resources

COleSafeArray Members

COleSafeArray::GetByteArray

Copies the contents of the safe array into a **CByteArray**.

```
void GetByteArray(
   CByteArray& bytes
);
```

Parameters

bytes

A reference to a CByteArray object.

See Also

Reference

COleSafeArray Class

Hierarchy Chart

Other Resources

COleSafeArray Members

COleSafeArray::GetDim

Returns the number of dimensions in the **COleSafeArray** object.

```
DWORD GetDim( );
```

Return Value

The number of dimensions in the safe array.

Example

```
COleSafeArray saMatrix;
DWORD numElements[] = {10, 5};

// creates a 2 dimensional safearray of type VT_I2
// with size 10x5 elements, with all indices starting at 0(default)
saMatrix.Create(VT_I2, 2, numElements);

ASSERT(saMatrix.GetDim() == 2);

COleSafeArray saVector;
SAFEARRAYBOUND rgsabounds[] = { {5, 1} };

// creates a 1 dimensional safearray of type VT_I1
// with size 5 elements, with the index starting at 1
saVector.Create(VT_I1, 1, rgsabounds);

ASSERT(saVector.GetDim() == 1);
```

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::Create
COleSafeArray::Redim
SafeArrayGetDim

Other Resources

COleSafeArray Members

COleSafeArray::GetElement

Retrieves a single element of the safe array.

```
void GetElement(
   long* rgIndices,
   void* pvData
);
```

Parameters

rgIndices

Pointer to an array of indexes for each dimension of the array.

pvData

Pointer to the location to place the element of the array.

Remarks

This function automatically calls the windows functions SafeArrayLock and SafeArrayUnlock before and after retrieving the element. If the data element is a string, object, or variant, the function copies the element in the correct way. The parameter *pvData* should point to a large enough buffer to contain the element.

On error, the function throws a CMemoryException or COleException.

Example

```
//sa is of type COleSafeArray with 2 dimensions
COleSafeArray sa;

//Determine upper bounds for both dimensions
long lNumRows;
long lNumCols;
sa.GetUBound(1, &lNumRows);
sa.GetUBound(2, &lNumCols);

//Display the elements in the SAFEARRAY.
long index[2];
VARIANT val;

//Determine upper bounds for both dimensions
long r, c;
sa.GetLBound(1, &r);
sa.GetLBound(2, &c);

for(;r <= lNumRows; r++ )
{
   for(; c <= lNumCols; c++ )
   {
      index[0]=r;
      index[1]=c;

      //retrieve each element of the safearray
      sa.GetElement(index, &val);

      switch(val.vt)
      {
      case VT_R8:
         TRACE("%1.2f\n", val.dblVal);
         break;

      case VT_BSTR:
         TRACE("%s\n",(CString)val.bstrVal);
         break;
```

```
        ...
        ...

        case VT_EMPTY:
           TRACE("<empty>\n");
           break;
        }
     }
  }
```

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::PutElement
SafeArrayGetElement

Other Resources

COleSafeArray Members

COleSafeArray::GetElemSize

Retrieves the size of an element in a **COleSafeArray** object.

```
DWORD GetElemSize( );
```

Return Value

The size, in bytes, of the elements of a safe array.

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::GetDim
SafeArrayGetElemsize

Other Resources

COleSafeArray Members

COleSafeArray::GetLBound

Returns the lower bound for any dimension of a **COleSafeArray** object.

```
void GetLBound(
   DWORD dwDim,
   long* pLBound
);
```

Parameters

dwDim

The array dimension for which to get the lower bound.

pLBound

Pointer to the location to return the lower bound.

Remarks

On error, the function throws a COleException.

Example

```
COleSafeArray saMatrix;
DWORD numElements[] = {10, 5};

// creates a 2 dimensional safearray of type VT_I2
// with size 10x5 elements, with all indices starting at 0(default)
saMatrix.Create(VT_I2, 2, numElements);

long lLBound;

//get lower bound for 1st dimension
saMatrix.GetLBound(1, &lLBound);

ASSERT(lLBound == 0);

//get lower for 2nd dimension
saMatrix.GetLBound(2, &lLBound);

ASSERT(lLBound == 0);

COleSafeArray saVector;
SAFEARRAYBOUND rgsabounds[] = { {5, 1} };

// creates a 1 dimensional safearray of type VT_I1
// with size 5 elements, with the index starting at 1
saVector.Create(VT_I1, 1, rgsabounds);

//get lower bound for 1st dimension
saVector.GetLBound(1, &lLBound);

ASSERT(lLBound == 1);
```

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::GetUBound
SafeArrayGetLBound

Other Resources

COleSafeArray Members

COleSafeArray::GetOneDimSize

Returns the number of elements in the one-dimensional **COleSafeArray** object.

```
DWORD GetOneDimSize( );
```

Return Value

The number of elements in the one-dimensional safe array.

Example

See the example for COleSafeArray::CreateOneDim.

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::CreateOneDim
COleSafeArray::ResizeOneDim
SafeArrayRedim

Other Resources

COleSafeArray Members

COleSafeArray::GetUBound

Returns the upper bound for any dimension of a safe array.

```
void GetUBound(
   DWORD dwDim,
   long* pUBound
);
```

Parameters

dwDim

The array dimension for which to get the upper bound.

pUBound

Pointer to the location to return the upper bound.

Remarks

On error, the function throws a COleException.

Example

```
COleSafeArray saMatrix;
DWORD numElements[] = {10, 5};

// creates a 2 dimensional safearray of type VT_I2
// with size 10x5 elements, with all indices starting at 0(default)
saMatrix.Create(VT_I2, 2, numElements);

long lUBound;
ASSERT(saMatrix.GetDim() == 2);

//get upper bound for 1st dimension
saMatrix.GetUBound(1, &lUBound);

ASSERT(lUBound == 9);

//get upper bound for 2nd dimension
saMatrix.GetUBound(2, &lUBound);

ASSERT(lUBound == 4);

COleSafeArray saVector;
SAFEARRAYBOUND rgsabounds[] = { {5, 1} };

// creates a 1 dimensional safearray of type VT_I1
// with size 5 elements, with the index starting at 1
saVector.Create(VT_I1, 1, rgsabounds);

//get upper bound for 1st dimension
saVector.GetUBound(1, &lUBound);

ASSERT(lUBound == 5);
```

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::GetLBound
SafeArrayGetUBound

Other Resources

COleSafeArray Members

COleSafeArray::Lock

Increments the lock count of an array and place a pointer to the array data in the array descriptor.

```
void Lock( );
```

Remarks

On error, it throws a COleException.

The pointer in the array descriptor is valid until **Unlock** is called. Calls to **Lock** can be nested; an equal number of calls to **Unlock** are required.

An array cannot be deleted while it is locked.

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::Unlock
SafeArrayLock

Other Resources

COleSafeArray Members

COleSafeArray::PtrOfIndex

Returns a pointer to the element specified by the index values.

```
void PtrOfIndex(
   long* rgIndices,
   void** ppvData
);
```

Parameters

rgIndices

An array of index values that identify an element of the array. All indexes for the element must be specified.

ppvData

On return, pointer to the element identified by the values in *rgIndices*.

See Also

Reference

COleSafeArray Class

Hierarchy Chart

SafeArrayPtrOfIndex

Other Resources

COleSafeArray Members

COleSafeArray::PutElement

Assigns a single element into the array.

```
void PutElement(
   long* rgIndices,
   void* pvData
);
```

Parameters

rgIndices

Pointer to an array of indexes for each dimension of the array.

pvData

Pointer to the data to assign to the array. **VT_DISPATCH**, **VT_UNKNOWN**, and **VT_BSTR** variant types are pointers and do not require another level of indirection.

Remarks

This function automatically calls the Windows functions SafeArrayLock and SafeArrayUnlock before and after assigning the element. If the data element is a string, object, or variant, the function copies it correctly, and if the existing element is a string, object, or variant, it is cleared correctly.

Note that you can have multiple locks on an array, so you can put elements into an array while the array is locked by other operations.

On error, the function throws a CMemoryException or COleException.

Example

```
VARIANT _stdcall retVariantArray(void)
{
   COleSafeArray saRet;
   DWORD numElements[] = {10, 10}; // 10x10

   // Create the 2 dimensional safe-array of type VT_R8 with size 10x10
   saRet.Create(VT_R8, 2, numElements);

   // Initialize safearray  with values...
   long index[2];
   for(index[0]=0; index[0]<10; index[0]++)
   {
      for(index[1]=0; index[1]<10; index[1]++)
       {
         double val = index[0] + index[1]*10;
         //populate the safearray elements with double values
         saRet.PutElement(index, &val);
      }
   }
   // Return the safe-array encapsulated in a VARIANT...
   return saRet.Detach();
}
```

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::GetElement
SafeArrayPutElement

Other Resources

COleSafeArray Members

COleSafeArray::Redim

Changes the least significant (rightmost) bound of a safe array.

```
void Redim(
   SAFEARRAYBOUND* psaboundNew
);
```

Parameters

psaboundNew

Pointer to a new safe array bound structure containing the new array bound. Only the least significant dimension of an array may be changed.

Remarks

On error, the function throws a COleException.

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::Create
COleSafeArray::GetDim
COleSafeArray::ResizeOneDim
SafeArrayRedim

Other Resources

COleSafeArray Members

COleSafeArray::ResizeOneDim

Changes the number of elements in a one-dimensional **COleSafeArray** object.

```
void ResizeOneDim(
   DWORD dwElements
);
```

Parameters

dwElements

Number of elements in the one-dimensional safe array.

Remarks

On error, the function throws a COleException.

Example

See the example for COleSafeArray::CreateOneDim.

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::Redim
COleSafeArray::GetOneDimSize
COleSafeArray::CreateOneDim
SafeArrayRedim

Other Resources

COleSafeArray Members

COleSafeArray::UnaccessData

Decrements the lock count of an array and invalidates the pointer retrieved by **AccessData**.

```
void UnaccessData( );
```

Remarks

On error, the function throws a COleException.

Example

See the example for COleSafeArray::AccessData.

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::AccessData
SafeArrayUnaccessData

Other Resources

COleSafeArray Members

COleSafeArray::Unlock

Decrements the lock count of an array so it can be freed or resized.

```
void Unlock( );
```

Remarks

This function is called after access to the data in an array is finished. On error, it throws a COleException.

See Also

Reference

COleSafeArray Class
Hierarchy Chart
COleSafeArray::Lock
SafeArrayUnlock

Other Resources

COleSafeArray Members

COleSafeArray Operators

For information about the operators in **COleSafeArray**, see COleSafeArray Members.

COleSafeArray::operator =

These overloaded assignment operators copy the source value into this **COleSafeArray** object.

```
COleSafeArray& operator =(
   const COleSafeArray& saSrc
);
COleSafeArray& operator =(
   const VARIANT& varSrc
);
COleSafeArray& operator =(
   LPCVARIANT pSrc
);
COleSafeArray& operator =(
   const COleVariant& varSrc
);
```

Remarks

A brief description of each operator follows:

- **operator =(** *saSrc* **)** Copies an existing **COleSafeArray** object into this object.
- **operator =(** *varSrc* **)** Copies an existing **VARIANT** or **COleVariant** array into this object.
- **operator =(** *pSrc* **)** Copies the **VARIANT** array object accessed by *pSrc* into this object.

See Also

Reference

COleSafeArray Class

Hierarchy Chart

Other Resources

COleSafeArray Members

COleSafeArray::operator ==

This operator compares two arrays (**SAFEARRAY**, **VARIANT**, **COleVariant**, or **COleSafeArray** arrays) and returns nonzero if they are equal; otherwise 0.

```
BOOL operator ==(
   const SAFEARRAY& saSrc
) const;
BOOL operator ==(
   LPCSAFEARRAY pSrc
) const;
BOOL operator ==(
   const COleSafeArray& saSrc
) const;
BOOL operator ==(
   const VARIANT& varSrc
) const;
BOOL operator ==(
   LPCVARIANT pSrc
) const;
BOOL operator ==(
   const COleVariant& varSrc
) const;
```

Remarks

Two arrays are equal if they have an equal number of dimensions, equal size in each dimension, and equal element values.

See Also

Reference

COleSafeArray Class

Hierarchy Chart

Other Resources

COleSafeArray Members

COleSafeArray::operator <<

This topic has been updated for Visual Studio 2005 SP1.

The **COleSafeArray** insertion (<<) operator supports diagnostic dumping and storing of a **COleSafeArray** object to an archive.

```
CDumpContext& AFXAPI operator<<(
   CDumpContext& dc,
   COleSafeArray& saSrc
);
```

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This operator is not supported in Smart Device projects.

See Also

Reference

COleSafeArray Class

Hierarchy Chart

Other Resources

COleSafeArray Members

COleSafeArray::operator LPCVARIANT

Call this casting operator to access the underlying **VARIANT** structure for this **COleSafeArray** object.

```
operator LPCVARIANT( ) const;
```

See Also

Reference

COleSafeArray Class

Hierarchy Chart

Other Resources

COleSafeArray Members

COleSafeArray::operator LPVARIANT

Call this casting operator to access the underlying **VARIANT** structure for this **COleSafeArray** object.

```
operator LPVARIANT( );
```

Remarks

Note that changing the value in the **VARIANT** structure accessed by the pointer returned by this function will change the value of this **COleSafeArray** object.

See Also

Reference

COleSafeArray Class

Hierarchy Chart

Other Resources

COleSafeArray Members

COleServerDoc Class

The base class for OLE server documents.

```
class AFX_NOVTABLE COleServerDoc : public COleLinkingDoc
```

Remarks

A server document can contain COleServerItem objects, which represent the server interface to embedded or linked items. When a server application is launched by a container to edit an embedded item, the item is loaded as its own server document; the **COleServerDoc** object contains just one **COleServerItem** object, consisting of the entire document. When a server application is launched by a container to edit a linked item, an existing document is loaded from disk; a portion of the document's contents is highlighted to indicate the linked item.

COleServerDoc objects can also contain items of the COleClientItem class. This allows you to create container-server applications. The framework provides functions to properly store the **COleClientItem** items while servicing the **COleServerItem** objects.

If your server application does not support links, a server document will always contain only one server item, which represents the entire embedded object as a document. If your server application does support links, it must create a server item each time a selection is copied to the Clipboard.

To use **COleServerDoc**, derive a class from it and implement the OnGetEmbeddedItem member function, which allows your server to support embedded items. Derive a class from **COleServerItem** to implement the items in your documents, and return objects of that class from **OnGetEmbeddedItem**.

To support linked items, **COleServerDoc** provides the OnGetLinkedItem member function. You can use the default implementation or override it if you have your own way of managing document items.

You need one **COleServerDoc**-derived class for each type of server document your application supports. For example, if your server application supports worksheets and charts, you need two **COleServerDoc**-derived classes.

For more information on servers, see the article Servers: Implementing a Server..

Requirements

Header: afxole.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

HIERSVR Sample: Demonstrates a Server Application with OLE Drag and Drop

Reference

COleLinkingDoc Class

Hierarchy Chart

COleDocument Class

COleLinkingDoc Class

COleTemplateServer Class

Other Resources

COleServerDoc Members

COleServerDoc Members

Base Class Members

Construction

Attributes

Operations

Overridables

Base Class Members

CObject Members

CCmdTarget Members

CDocument Members

COleDocument Members

COleLinkingDoc Members

Construction

COleServerDoc	Constructs a **COleServerDoc** object.

Attributes

GetClientSite	Retrieves a pointer to the underlying **IOleClientSite** interface.
GetEmbeddedItem	Returns a pointer to an item representing the entire document.
GetItemClipRect	Returns the current clipping rectangle for in-place editing.
GetItemPosition	Returns the current position rectangle, relative to the container application's client area, for in-place editing.
GetZoomFactor	Returns the zoom factor in pixels.
IsDocObject	Determines if the document is a DocObject.
IsEmbedded	Indicates whether the document is embedded in a container document or running stand-alone.
IsInPlaceActive	Returns **TRUE** if the item is currently activated in place.

Operations

ActivateDocObject	Activates the associated DocObject document.
ActivateInPlace	Activates the document for in-place editing.
DeactivateAndUndo	Deactivates the server's user interface.
DiscardUndoState	Discards undo-state information.
NotifyChanged	Notifies containers that the user has changed the document.
NotifyClosed	Notifies containers that the user has closed the document.
NotifyRename	Notifies containers that the user has renamed the document.

NotifySaved	Notifies containers that the user has saved the document.
OnExecOleCmd	Executes a specified command or displays help for the command.
RequestPositionChange	Changes the position of the in-place editing frame.
SaveEmbedding	Tells the container application to save the document.
ScrollContainerBy	Scrolls the container document.
UpdateAllItems	Notifies containers that the user has changed the document.

Overridables

CreateInPlaceFrame	Called by the framework to create a frame window for in-place editing.
DestroyInPlaceFrame	Called by the framework to destroy a frame window for in-place editing.
GetDocObjectServer	Override this function to create a new **CDocObjectServer** object and indicate that this document is a DocObject container.
OnClose	Called by the framework when a container requests to close the document.
OnDeactivate	Called by the framework when the user deactivates an item that was activated in place.
OnDeactivateUI	Called by the framework to destroy controls and other user-interface elements created for in-place activation.
OnDocWindowActivate	Called by the framework when the container's document frame window is activated or deactivated.
OnFrameWindowActivate	Called by the framework when the container's frame window is activated or deactivated.
OnGetEmbeddedItem	Called to get a **COleServerItem** that represents the entire document; used to get an embedded ite m. Implementation required.
OnReactivateAndUndo	Called by the framework to undo changes made during in-place editing.
OnResizeBorder	Called by the framework when the container application's frame window or document window is re sized.
OnSetHostNames	Called by the framework when a container sets the window title for an embedded object.
OnSetItemRects	Called by the framework to position the in-place editing frame window within the container applicat ion's window.
OnShowControlBars	Called by the framework to show or hide control bars for in-place editing.
OnShowDocument	Called by the framework to show or hide the document.
OnUpdateDocument	Called by the framework when a server document that is an embedded item is saved, updating the c ontainer's copy of the item.

See Also

Reference

COleServerDoc Class

Hierarchy Chart

COleServerDoc Member Functions

For information about the member functions in **COleServerDoc**, see COleServerDoc Members.

COleServerDoc::ActivateDocObject

Activates the associated DocObject document.

```
void ActivateDocObject( );
```

Remarks

By default, **COleServerDoc** does not support Active documents (also referred to as DocObjects). To enable this support, see GetDocObjectServer and class CDocObjectServer.

See Also

Reference

COleServerDoc Class

Hierarchy Chart

COleServerDoc::IsDocObject

Other Resources

COleServerDoc Members

COleServerDoc::ActivateInPlace

Activates the item for in-place editing.

```
BOOL ActivateInPlace( );
```

Return Value

Nonzero if successful; otherwise 0, which indicates that the item is fully open.

Remarks

This function performs all operations necessary for in-place activation. It creates an in-place frame window, activates it and sizes it to the item, sets up shared menus and other controls, scrolls the item into view, and sets the focus to the in-place frame window.

This function is called by the default implementation of COleServerItem::OnShow. Call this function if your application supports another verb for in-place activation (such as Play).

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerItem::OnShow

Other Resources

COleServerDoc Members

COleServerDoc::COleServerDoc

Constructs a **COleServerDoc** object without connecting with the OLE system DLLs.

```
COleServerDoc( );
```

Remarks

You must call COleLinkingDoc::Register to open communications with OLE. If you are using COleTemplateServer in your application, **COleLinkingDoc::Register** is called for you by **COleLinkingDoc**'s implementation of **OnNewDocument**, **OnOpenDocument**, and **OnSaveDocument**.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleLinkingDoc::Register

Other Resources

COleServerDoc Members

COleServerDoc::CreateInPlaceFrame

The framework calls this function to create a frame window for in-place editing.

```
virtual COleIPFrameWnd* CreateInPlaceFrame(
   CWnd* pParentWnd
);
```

Parameters

pParentWnd

Pointer to the container application's parent window.

Return Value

A pointer to the in-place frame window, or **NULL** if unsuccessful.

Remarks

The default implementation uses information specified in the document template to create the frame. The view used is the first view created for the document. This view is temporarily detached from the original frame and attached to the newly created frame.

This is an advanced overridable.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::DestroyInPlaceFrame

Other Resources

COleServerDoc Members

COleServerDoc::DeactivateAndUndo

Call this function if your application supports Undo and the user chooses Undo after activating an item but before editing it.

```
BOOL DeactivateAndUndo( );
```

Return Value

Nonzero on success; otherwise 0.

Remarks

If the container application is written using the Microsoft Foundation Class Library, calling this function causes COleClientItem::OnDeactivateAndUndo to be called, which deactivates the server's user interface.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleClientItem::OnDeactivateAndUndo

Other Resources

COleServerDoc Members

COleServerDoc::DestroyInPlaceFrame

The framework calls this function to destroy an in-place frame window and return the server application's document window to its state before in-place activation.

```
virtual void DestroyInPlaceFrame(
   COleIPFrameWnd* pFrameWnd
);
```

Parameters

pFrameWnd

Pointer to the in-place frame window to be destroyed.

Remarks

This is an advanced overridable.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::CreateInPlaceFrame

Other Resources

COleServerDoc Members

COleServerDoc::DiscardUndoState

If the user performs an editing operation that cannot be undone, call this function to force the container application to discard its undo-state information.

```
BOOL DiscardUndoState( );
```

Return Value

Nonzero on success; otherwise 0.

Remarks

This function is provided so that servers that support Undo can free resources that would otherwise be consumed by undo-state information that cannot be used.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::OnReactivateAndUndo

Other Resources

COleServerDoc Members

COleServerDoc::GetClientSite

Retrieves a pointer to the underlying **IOleClientSite** interface.

```
LPOLECLIENTSITE GetClientSite( ) const;
```

Return Value

Retrieves a pointer to the underlying IOleClientSite interface.

See Also

Reference

COleServerDoc Class

Hierarchy Chart

Other Resources

COleServerDoc Members

COleServerDoc::GetDocObjectServer

Override this function to create a new **CDocObjectServer** item and return a pointer to it.

```
virtual CDocObjectServer* GetDocObjectServer(
   LPOLEDOCUMENTSITE pDocSite
);
```

Parameters

pDocSite

Pointer to the **IOleDocumentSite** interface that will connect this document to the server.

Return Value

A pointer to a **CDocObjectServer**; **NULL** if the operation failed.

Remarks

When a DocObject server is activated, the return of a non-**NULL** pointer shows that the client can support DocObjects. The default implementation returns **NULL**.

A typical implementation for a document that supports DocObjects will simply allocate a new **CDocObjectServer** object and return it to the caller. For example:

```
CDocObjectServer* COleServerDoc::GetDocObjectServer(LPOLEDOCUMENTSITE pSite)
{
   return new CDocObjectServer(this, pSite);
}
```

See Also

Reference

COleServerDoc Class

Hierarchy Chart

CDocObjectServer::CDocObjectServer

Other Resources

COleServerDoc Members

COleServerDoc::GetEmbeddedItem

Call this function to get a pointer to an item representing the entire document.

```
COleServerItem* GetEmbeddedItem( );
```

Return Value

A pointer to an item representing the entire document; **NULL** if the operation failed.

Remarks

It calls COleServerDoc::OnGetEmbeddedItem, a virtual function with no default implementation.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::OnGetEmbeddedItem

Other Resources

COleServerDoc Members

COleServerDoc::GetItemClipRect

Call the **GetItemClipRect** member function to get the clipping-rectangle coordinates of the item that is being edited in place.

```
void GetItemClipRect(
   LPRECT lpClipRect
) const;
```

Parameters

lpClipRect

Pointer to a **RECT** structure or a **CRect** object to receive the clipping-rectangle coordinates of the item.

Remarks

Coordinates are in pixels relative to the container application window's client area.

Drawing should not occur outside the clipping rectangle. Usually, drawing is automatically restricted. Use this function to determine whether the user has scrolled outside the visible portion of the document; if so, scroll the container document as needed by means of a call to ScrollContainerBy.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::GetItemPosition
COleServerDoc::ScrollContainerBy

Other Resources

COleServerDoc Members

COleServerDoc::GetItemPosition

Call the **GetItemPosition** member function to get the coordinates of the item being edited in place.

```
void GetItemPosition(
   LPRECT lpPosRect
) const;
```

Parameters

lpPosRect

Pointer to a **RECT** structure or a **CRect** object to receive the coordinates of the item.

Remarks

Coordinates are in pixels relative to the container application window's client area.

The item's position can be compared with the current clipping rectangle to determine the extent to which the item is visible (or not visible) on the screen.

See Also

Reference

COleServerDoc Class

Hierarchy Chart

COleServerDoc::GetItemClipRect

Other Resources

COleServerDoc Members

COleServerDoc::GetZoomFactor

The **GetZoomFactor** member function determines the "zoom factor" of an item that has been activated for in-place editing.

```
BOOL GetZoomFactor(
   LPSIZE lpSizeNum = NULL,
   LPSIZE lpSizeDenom = NULL,
   LPCRECT lpPosRect = NULL
) const;
```

Parameters

lpSizeNum

Pointer to an object of class **CSize** that will hold the zoom factor's numerator. Can be **NULL**.

lpSizeDenom

Pointer to an object of class **CSize** that will hold the zoom factor's denominator. Can be **NULL**.

lpPosRect

Pointer to an object of class **CRect** that describes the item's new position. If this argument is **NULL**, the function uses the item's current position.

Return Value

Nonzero if the item is activated for in-place editing and its zoom factor is other than 100% (1:1); otherwise 0.

Remarks

The zoom factor, in pixels, is the proportion of the item's size to its current extent. If the container application has not set the item's extent, its natural extent (as determined by COleServerItem::OnGetExtent) is used.

The function sets its first two arguments to the numerator and denominator of the item's "zoom factor." If the item is not being edited in place, the function sets these arguments to a default value of 100% (or 1:1) and returns zero. For further information, see Technical Note 40, MFC/OLE In-Place Resizing and Zooming.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::GetItemPosition
COleServerDoc::GetItemClipRect
COleServerDoc::OnSetItemRects

Other Resources

COleServerDoc Members

COleServerDoc::IsDocObject

Determines if the document is a DocObject.

```
BOOL IsDocObject( ) const;
```

Return Value

TRUE if the document is a DocObject; otherwise **FALSE**.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::ActivateDocObject

Other Resources

COleServerDoc Members

COleServerDoc::IsEmbedded

Call the **IsEmbedded** member function to determine whether the document represents an object embedded in a container.

```
BOOL IsEmbedded( ) const;
```

Return Value

Nonzero if the **COleServerDoc** object is a document that represents an object embedded in a container; otherwise 0.

Remarks

A document loaded from a file is not embedded although it may be manipulated by a container application as a link. A document that is embedded in a container document is considered to be embedded.

See Also

Reference

COleServerDoc Class

Hierarchy Chart

Other Resources

COleServerDoc Members

COleServerDoc::IsInPlaceActive

Call the **IsInPlaceActive** member function to determine whether the item is currently in the in-place active state.

```
BOOL IsInPlaceActive( ) const;
```

Return Value

Nonzero if the **COleServerDoc** object is active in place; otherwise 0.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleClientItem::OnActivate
COleServerDoc::OnReactivateAndUndo
COleServerDoc::ActivateInPlace

Other Resources

COleServerDoc Members

COleServerDoc::NotifyChanged

Call this function to notify all linked items connected to the document that the document has changed.

```
void NotifyChanged( );
```

Remarks

Typically, you call this function after the user changes some global attribute such as the dimensions of the server document. If an OLE item is linked to the document with an automatic link, the item is updated to reflect the changes. In container applications written with the Microsoft Foundation Class Library, the OnChange member function of **COleClientItem** is called.

Note
This function is included for compatibility with OLE 1. New applications should use UpdateAllItems.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::NotifyClosed
COleServerDoc::NotifySaved
COleClientItem::OnChange

Other Resources

COleServerDoc Members

COleServerDoc::NotifyClosed

Call this function to notify the container(s) that the document has been closed.

```
void NotifyClosed( );
```

Remarks

When the user chooses the Close command from the File menu, **NotifyClosed** is called by **COleServerDoc**'s implementation of the OnCloseDocument member function. In container applications written with the Microsoft Foundation Class Library, the OnChange member function of **COleClientItem** is called.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::NotifyChanged
COleServerDoc::NotifySaved
COleClientItem::OnChange
CDocument::OnCloseDocument

Other Resources

COleServerDoc Members

COleServerDoc::NotifyRename

Call this function after the user renames the server document.

```
void NotifyRename(
   LPCTSTR lpszNewName
);
```

Parameters

lpszNewName

Pointer to a string specifying the new name of the server document; this is typically a fully qualified path.

Remarks

When the user chooses the Save As command from the File menu, **NotifyRename** is called by **COleServerDoc**'s implementation of the OnSaveDocument member function. This function notifies the OLE system DLLs, which in turn notify the containers. In container applications written with the Microsoft Foundation Class Library, the OnChange member function of **COleClientItem** is called.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::NotifySaved
CDocument::OnSaveDocument

Other Resources

COleServerDoc Members

COleServerDoc::NotifySaved

Call this function after the user saves the server document.

```
void NotifySaved( );
```

Remarks

When the user chooses the Save command from the File menu, **NotifySaved** is called for you by **COleServerDoc**'s implementation of OnSaveDocument. This function notifies the OLE system DLLs, which in turn notify the containers. In container applications written with the Microsoft Foundation Class Library, the OnChange member function of **COleClientItem** is called.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::NotifyChanged
COleServerDoc::NotifyClosed
COleClientItem::OnChange
CDocument::OnSaveDocument

Other Resources

COleServerDoc Members

COleServerDoc::OnClose

Called by the framework when a container requests that the server document be closed.

```
virtual void OnClose(
   OLECLOSE dwCloseOption
);
```

Parameters

dwCloseOption

A value from the enumeration **OLECLOSE**. This parameter can have one of the following values:

- **OLECLOSE_SAVEIFDIRTY** The file is saved if it has been modified.
- **OLECLOSE_NOSAVE** The file is closed without being saved.
- **OLECLOSE_PROMPTSAVE** If the file has been modified, the user is prompted about saving it.

Remarks

The default implementation calls **CDocument::OnCloseDocument**.

For more information and additional values, see OLECLOSE in the *Platform SDK*.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleException Class
CDocument::OnCloseDocument

Other Resources

COleServerDoc Members

COleServerDoc::OnDeactivate

Called by the framework when the user deactivates an embedded or linked item that is currently in-place active.

```
virtual void OnDeactivate( );
```

Remarks

This function restores the container application's user interface to its original state and destroys any menus and other controls that were created for in-place activation.

The undo state information should be unconditionally released at this point.

For more information, see the article Activation..

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::ActivateInPlace
COleServerDoc::OnDeactivateUI
COleServerDoc::DestroyInPlaceFrame

Other Resources

COleServerDoc Members

COleServerDoc::OnDeactivateUI

Called when the user deactivates an item that was activated in place.

```
virtual void OnDeactivateUI(
   BOOL bUndoable
);
```

Parameters

bUndoable

Specifies whether the editing changes can be undone.

Remarks

This function restores the container application's user interface to its original state, hiding any menus and other controls that were created for in-place activation.

The framework always sets *bUndoable* to **FALSE**. If the server supports undo and there is an operation that can be undone, call the base-class implementation with *bUndoable* set to **TRUE**.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::OnDeactivate

Other Resources

COleServerDoc Members

COleServerDoc::OnDocWindowActivate

The framework calls this function to activate or deactivate a document window for in-place editing.

```
virtual void OnDocWindowActivate(
   BOOL bActivate
);
```

Parameters

bActivate

Specifies whether the document window is to be activated or deactivated.

Remarks

The default implementation removes or adds the frame-level user interface elements as appropriate. Override this function if you want to perform additional actions when the document containing your item is activated or deactivated.

For more information, see the article Activation..

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::ActivateInPlace
COleServerDoc::OnReactivateAndUndo
COleServerDoc::OnShowControlBars
COleServerDoc::OnDeactivateUI
COleServerDoc::OnFrameWindowActivate
COleIPFrameWnd Class

Other Resources

COleServerDoc Members

COleServerDoc::OnExecOleCmd

The framework calls this function to execute a specified command or display help for the command.

```
virtual HRESULT OnExecOleCmd(
   const GUID* pguidCmdGroup,
   DWORD nCmdID,
   DWORD nCmdExecOpt,
   VARIANTARG* pvarargIn,
   VARIANTARG* pvarargOut
);
```

Parameters

pguidCmdGroup

A pointer to a GUID that identifies a set of commands. Can be **NULL** to indicate the default command group.

nCmdID

The command to execute. Must be in the group identified by *pguidCmdGroup*.

nCmdExecOut

The way the object should execute the command, one or more of the following values from the **OLECMDEXECOPT** enumeration:

OLECMDEXECOPT_DODEFAULT

OLECMDEXECOPT_PROMPTUSER

OLECMDEXECOPT_DONTPROMPTUSER

OLECMDEXECOPT_SHOWHELP

pvarargIn

Pointer to a **VARIANTARG** containing input arguments for the command. Can be **NULL**.

pvarargOut

Pointer to a **VARIANTARG** to receive the output return values from the command. Can be **NULL**.

Return Value

Returns **S_OK** if successful; otherwise, one of the following error codes:

Value	Description
E_UNEXPECTED	Unexpected error occurred
E_FAIL	Error occurred
E_NOTIMPL	Indicates MFC itself should attempt to translate and dispatch the command
OLECMDERR_E_UNKNOWNGROUP	*pguidCmdGroup* is non-**NULL** but does not specify a recognized command group
OLECMDERR_E_NOTSUPPORTED	*nCmdID* is not recognized as a valid command in the group *pguidCmdGroup*
OLECMDERR_DISABLED	The command identified by *nCmdID* is disabled and cannot be executed
OLECMDERR_NOHELP	Caller asked for help on the command identified by *nCmdID* but no help is available
OLECMDERR_CANCELED	User canceled the execution

Remarks

COleCmdUI can be used to enable, update, and set other properties of DocObject user interface commands. After the commands are initialized, you can execute them with **OnExecOleCmd**.

The framework calls the function before attempting to translate and dispatch an OLE document command. You don't need to override this function to handle standard OLE document commands, but you must supply an override to this function if you want to handle your own custom commands or handle commands that accept parameters or return results.

Most of the commands do not take arguments or return values. For a majority of commands the caller can pass **NULL**s for *pvarargIn* and *pvarargOut*. For commands that expect input values, the caller can declare and initialize a **VARIANTARG** variable and pass a pointer to the variable in *pvarargIn*. For commands that require a single value, the argument can be stored directly in the **VARIANTARG** and passed to the function. Multiple arguments must be packaged within the **VARIANTARG** using one of the supported types (such as **IDispatch** and **SAFEARRAY**).

Similarly, if a command returns arguments the caller is expected to declare a **VARIANTARG**, initialize it to **VT_EMPTY**, and pass its address in *pvarargOut*. If a command returns a single value, the object can store that value directly in *pvarargOut*. Multiple output values must be packaged in some way appropriate for the **VARIANTARG**.

The base-class implementation of this function will walk the **OLE_COMMAND_MAP** structures associated with the command target and try to dispatch the command to an appropriate handler. The base-class implementation works only with commands that do not accept arguments or return values. If you need to handle commands that do accept arguments or return values, you must override this function and work with the *pvarargIn* and *pvarargOut* parameters yourself.

See Also

Reference

COleServerDoc Class

Hierarchy Chart

COleCmdUI Class

Other Resources

COleServerDoc Members

COleServerDoc::OnFrameWindowActivate

The framework calls this function when the container application's frame window is activated or deactivated.

```
virtual void OnFrameWindowActivate(
   BOOL bActivate
);
```

Parameters

bActivate

Specifies whether the frame window is to be activated or deactivated.

Remarks

The default implementation cancels any help modes the frame window might be in. Override this function if you want to perform special processing when the frame window is activated or deactivated.

For more information, see the article Activation..

See Also

Reference

COleServerDoc Class

Hierarchy Chart

COleServerDoc::OnDocWindowActivate

Other Resources

COleServerDoc Members

COleServerDoc::OnGetEmbeddedItem

Called by the framework when a container application calls the server application to create or edit an embedded item.

```
virtual COleServerItem* OnGetEmbeddedItem( ) = 0;
```

Return Value

A pointer to an item representing the entire document; **NULL** if the operation failed.

Remarks

There is no default implementation. You must override this function to return an item that represents the entire document. This return value should be an object of a **COleServerItem**-derived class.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleLinkingDoc::OnGetLinkedItem
COleServerItem Class

Other Resources

COleServerDoc Members

COleServerDoc::OnReactivateAndUndo

The framework calls this function when the user chooses to undo changes made to an item that has been activated in place, changed, and subsequently deactivated.

```
virtual BOOL OnReactivateAndUndo( );
```

Return Value

Nonzero if successful; otherwise 0.

Remarks

The default implementation does nothing except return **FALSE** to indicate failure.

Override this function if your application supports undo. Usually you would perform the undo operation, then activate the item by calling **ActivateInPlace**. If the container application is written with the Microsoft Foundation Class Library, calling **COleClientItem::ReactivateAndUndo** causes this function to be called.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::ActivateInPlace
COleServerDoc::IsInPlaceActive
COleClientItem::ReactivateAndUndo

Other Resources

COleServerDoc Members

COleServerDoc::OnResizeBorder

The framework calls this function when the container application's frame windows change size.

```
virtual void OnResizeBorder(
   LPCRECT lpRectBorder,
   LPOLEINPLACEUIWINDOW lpUIWindow,
   BOOL bFrame
);
```

Parameters

lpRectBorder

Pointer to a **RECT** structure or a **CRect** object that specifies the coordinates of the border.

lpUIWindow

Pointer to an object of class **IOleInPlaceUIWindow** that owns the current in-place editing session.

bFrame

TRUE if *lpUIWindow* points to the container application's top-level frame window, or **FALSE** if *lpUIWindow* points to the container application's document-level frame window.

Remarks

This function resizes and adjusts toolbars and other user-interface elements in accordance with the new window size.

For more information, see IOleInPlaceUIWindow in the *Platform SDK*.

This is an advanced overridable.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::OnShowControlBars

Other Resources

COleServerDoc Members

COleServerDoc::OnSetHostNames

Called by the framework when the container sets or changes the host names for this document.

```
virtual void OnSetHostNames(
   LPCTSTR lpszHost,
   LPCTSTR lpszHostObj
);
```

Parameters

lpszHost

Pointer to a string that specifies the name of the container application.

lpszHostObj

Pointer to a string that specifies the container's name for the document.

Remarks

The default implementation changes the document title for all views referring to this document.

Override this function if your application sets the titles through a different mechanism.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleClientItem::SetHostNames

Other Resources

COleServerDoc Members

COleServerDoc::OnSetItemRects

The framework calls this function to position the in-place editing frame window within the container application's frame window.

```
virtual void OnSetItemRects(
   LPCRECT lpPosRect,
   LPCRECT lpClipRect
);
```

Parameters

lpPosRect

Pointer to a **RECT** structure or a **CRect** object that specifies the in-place frame window's position relative to the container application's client area.

lpClipRect

Pointer to a **RECT** structure or a **CRect** object that specifies the in-place frame window's clipping rectangle relative to the container application's client area.

Remarks

Override this function to update the view's zoom factor, if necessary.

This function is usually called in response to a **RequestPositionChange** call, although it can be called at any time by the container to request a position change for the in-place item.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::RequestPositionChange
COleIPFrameWnd::RepositionFrame
COleClientItem::SetItemRects
COleServerDoc::GetZoomFactor

Other Resources

COleServerDoc Members

COleServerDoc::OnShowControlBars

The framework calls this function to show or hide the server application's control bars associated with the frame window identified by *pFrameWnd*.

```
virtual void OnShowControlBars(
   CFrameWnd *pFrameWnd,
   BOOL bShow
);
```

Parameters

pFrameWnd

Pointer to the frame window whose control bars should be hidden or shown.

bShow

Determines whether control bars are shown or hidden.

Remarks

The default implementation enumerates all control bars owned by that frame window and hides or shows them.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::ActivateInPlace
COleServerDoc::OnReactivateAndUndo
COleServerDoc::OnFrameWindowActivate
COleServerDoc::IsInPlaceActive

Other Resources

COleServerDoc Members

COleServerDoc::OnShowDocument

The framework calls the **OnShowDocument** function when the server document must be hidden or shown.

```
virtual void OnShowDocument(
   BOOL bShow
);
```

Parameters

bShow

Specifies whether the user interface to the document is to be shown or hidden.

Remarks

If *bShow* is **TRUE**, the default implementation activates the server application, if necessary, and causes the container application to scroll its window so that the item is visible. If *bShow* is **FALSE**, the default implementation deactivates the item through a call to **OnDeactivate**, then destroys or hides all frame windows that have been created for the document, except the first one. If no visible documents remain, the default implementation hides the server application.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::ActivateInPlace
COleServerItem::OnDoVerb
COleServerDoc::IsInPlaceActive
COleServerDoc::OnDeactivateUI

Other Resources

COleServerDoc Members

COleServerDoc::OnUpdateDocument

Called by the framework when saving a document that is an embedded item in a compound document.

```
virtual BOOL OnUpdateDocument( );
```

Return Value

Nonzero if the document was successfully updated; otherwise 0.

Remarks

The default implementation calls the COleServerDoc::NotifySaved and COleServerDoc::SaveEmbedding member functions and then marks the document as clean. Override this function if you want to perform special processing when updating an embedded item.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::NotifySaved
COleServerDoc::SaveEmbedding
CDocument::OnSaveDocument

Other Resources

COleServerDoc Members

COleServerDoc::RequestPositionChange

Call this member function to have the container application change the item's position.

```
void RequestPositionChange(
   LPCRECT lpPosRect
);
```

Parameters

lpPosRect

Pointer to a **RECT** structure or a **CRect** object containing the item's new position.

Remarks

This function is usually called (in conjunction with **UpdateAllItems**) when the data in an in-place active item has changed. Following this call, the container might or might not perform the change by calling **OnSetItemRects**. The resulting position might be different from the one requested.

See Also

Reference

COleServerDoc Class

Hierarchy Chart

COleServerDoc::ScrollContainerBy

Other Resources

COleServerDoc Members

COleServerDoc::SaveEmbedding

Call this function to tell the container application to save the embedded object.

```
void SaveEmbedding( );
```

Remarks

This function is called automatically from **OnUpdateDocument**. Note that this function causes the item to be updated on disk, so it is usually called only as a result of a specific user action.

See Also

Reference

COleServerDoc Class

Hierarchy Chart

COleServerDoc::NotifyClosed

Other Resources

COleServerDoc Members

COleServerDoc::ScrollContainerBy

Call the **ScrollContainerBy** member function to scroll the container document by the amount, in pixels, indicated by *sizeScroll*.

```
BOOL ScrollContainerBy(
   CSize sizeScroll
);
```

Parameters

sizeScroll

Indicates how far the container document is to scroll.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Positive values indicate scrolling down and to the right; negative values indicate scrolling up and to the left.

See Also

Reference

COleServerDoc Class

Hierarchy Chart

COleClientItem::OnScrollBy

Other Resources

COleServerDoc Members

COleServerDoc::UpdateAllItems

Call this function to notify all linked items connected to the document that the document has changed.

```
void UpdateAllItems(
   COleServerItem* pSender,
   LPARAM lHint = 0L,
   CObject* pHint = NULL,
   DVASPECT nDrawAspect = DVASPECT_CONTENT
);
```

Parameters

pSender

Pointer to the item that modified the document, or **NULL** if all items are to be updated.

lHint

Contains information about the modification.

pHint

Pointer to an object storing information about the modification.

nDrawAspect

Determines how the item is to be drawn. This is a value from the **DVASPECT** enumeration. This parameter can have one of the following values:

- **DVASPECT_CONTENT** Item is represented in such a way that it can be displayed as an embedded object inside its container.
- **DVASPECT_THUMBNAIL** Item is rendered in a "thumbnail" representation so that it can be displayed in a browsing tool.
- **DVASPECT_ICON** Item is represented by an icon.
- **DVASPECT_DOCPRINT** Item is represented as if it were printed using the Print command from the File menu.

Remarks

You typically call this function after the user changes the server document. If an OLE item is linked to the document with an automatic link, the item is updated to reflect the changes. In container applications written with the Microsoft Foundation Class Library, the OnChange member function of **COleClientItem** is called.

This function calls the **OnUpdate** member function for each of the document's items except the sending item, passing *pHint*, *lHint*, and *nDrawAspect*. Use these parameters to pass information to the items about the modifications made to the document. You can encode information using *lHint* or you can define a **CObject**-derived class to store information about the modifications and pass an object of that class using *pHint*. Override the **OnUpdate** member function in your **COleServerItem**-derived class to optimize the updating of each item depending on whether its presentation has changed.

See Also

Reference

COleServerDoc Class
Hierarchy Chart
COleServerDoc::NotifyChanged
COleServerItem::OnUpdate
COleServerDoc::NotifySaved
COleClientItem::OnChange

Other Resources

COleServerDoc Members

COleServerItem Class

Provides the server interface to OLE items.

```
class COleServerItem : public CDocItem
```

Remarks

A linked item can represent some or all of a server document. An embedded item always represents an entire server document.

The **COleServerItem** class defines several overridable member functions that are called by the OLE system dynamic-link libraries (DLLs), usually in response to requests from the container application. These member functions allow the container application to manipulate the item indirectly in various ways, such as by displaying it, executing its verbs, or retrieving its data in various formats.

To use **COleServerItem**, derive a class from it and implement the OnDraw and Serialize member functions. The **OnDraw** function provides the metafile representation of an item, allowing it to be displayed when a container application opens a compound document. The **Serialize** function of **CObject** provides the native representation of an item, allowing an embedded item to be transferred between the server and container applications. OnGetExtent provides the natural size of the item to the container, enabling the container to size the item.

For more information about servers and related topics, see the article Servers: Implementing a Server and "Creating a Container/Server Application" in the article Containers: Advanced Features.

Requirements

Header: afxole.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

HIERSVR Sample: Demonstrates a Server Application with OLE Drag and Drop

Reference

CDocItem Class
Hierarchy Chart
COleClientItem Class
COleServerDoc Class
COleTemplateServer Class

Other Resources

COleServerItem Members

COleServerItem Members

Base Class Members

Status

Operations

Construction

Overridables

Data Members

Base Class Members

CObject Members

CCmdTarget Members

CDocItem Members

Status

GetDocument	Returns the server document that contains the item.
GetItemName	Returns the name of the item. Used for linked items only.
IsConnected	Indicates whether the item is currently attached to an active container.
IsLinkedItem	Indicates whether the item represents a linked OLE item.
SetItemName	Sets the name of the item. Used for linked items only.

Operations

AddOtherClipboardData	Places presentation and conversion formats in a **COleDataSource** object.
CopyToClipboard	Copies the item to the Clipboard.
DoDragDrop	Performs a drag-and-drop operation.
GetClipboardData	Gets the data source for use in data transfer (drag and drop or Clipboard).
GetEmbedSourceData	Gets the **CF_EMBEDSOURCE** data for an OLE item.
GetLinkSourceData	Gets the **CF_LINKSOURCE** data for an OLE item.
GetObjectDescriptorData	Gets the **CF_OBJECTDESCRIPTOR** data for an OLE item.
NotifyChanged	Updates all containers with automatic link update.

Construction

COleServerItem	Constructs a **COleServerItem** object.
GetDataSource	Gets the object used to store conversion formats.

Overridables

OnDoVerb	Called to execute a verb.

OnDraw	Called when the container requests to draw the item; implementation required.
OnDrawEx	Called for specialized item drawing.
OnGetClipboardData	Called by the framework to get the data that would be copied to the Clipboard.
OnGetExtent	Called by the framework to retrieve the size of the OLE item.
OnHide	Called by the framework to hide the OLE item.
OnInitFromData	Called by the framework to initialize an OLE item using the contents of the data transfer object specified .
OnOpen	Called by the framework to display the OLE item in its own top-level window.
OnQueryUpdateItems	Called to determine whether any linked items require updating.
OnRenderData	Retrieves data as part of delayed rendering.
OnRenderFileData	Retrieves data into a **CFile** object as part of delayed rendering.
OnRenderGlobalData	Retrieves data into an **HGLOBAL** as part of delayed rendering.
OnSetColorScheme	Called to set the item's color scheme.
OnSetData	Called to set the item's data.
OnSetExtent	Called by the framework to set the size of the OLE item.
OnShow	Called when the container requests to show the item.
OnUpdate	Called when some portion of the document the item belongs in is changed.
OnUpdateItems	Called to update the presentation cache of all items in the server document.

Data Members

m_sizeExtent	Informs the server about how much of the OLE item is visible.

See Also

Reference

COleServerItem Class

Hierarchy Chart

COleServerItem Member Functions

For information about the member functions in **COleServerItem**, see COleServerItem Members.

COleServerItem::AddOtherClipboardData

Call this function to place the presentation and conversion formats for the OLE item in the specified **COleDataSource** object.

```
void AddOtherClipboardData(
   COleDataSource* pDataSource
);
```

Parameters

pDataSource

Pointer to the **COleDataSource** object in which the data should be placed.

Remarks

You must have implemented the OnDraw member function to provide the presentation format (a metafile picture) for the item. To support other conversion formats, register them using the COleDataSource object returned by GetDataSource and override the OnRenderData member function to provide data in the formats you want to support.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleDataSource Class
COleServerItem::GetDataSource
COleServerItem::GetEmbedSourceData
COleServerItem::OnDraw

Other Resources

COleServerItem Members

COleServerItem::COleServerItem

Constructs a **COleServerItem** object and adds it to the server document's collection of document items.

```
COleServerItem(
   COleServerDoc* pServerDoc,
   BOOL bAutoDelete
);
```

Parameters

pServerDoc

Pointer to the document that will contain the new item.

bAutoDelete

Flag indicating whether the object can be deleted when a link to it is released. Set this to **FALSE** if the **COleServerItem** object is an integral part of your document's data which you must delete. Set this to **TRUE** if the object is a secondary structure used to identify a range in your document's data that can be deleted by the framework.

See Also

Reference

COleServerItem Class

Hierarchy Chart

COleDocument::AddItem

Other Resources

COleServerItem Members

COleServerItem::CopyToClipboard

Call this function to copy the OLE item to the Clipboard.

```
void CopyToClipboard(
   BOOL bIncludeLink = FALSE
);
```

Parameters

bIncludeLink

Set this to **TRUE** if link data should be copied to the Clipboard. Set this to **FALSE** if your server application does not support links.

Remarks

The function uses the OnGetClipboardData member function to create a COleDataSource object containing the OLE item's data in the formats supported. The function then places the **COleDataSource** object on the Clipboard by using the COleDataSource::SetClipboard function. The **COleDataSource** object includes the item's native data and its representation in **CF_METAFILEPICT** format, as well as data in any conversion formats you choose to support. You must have implemented Serialize and OnDraw for this member function to work.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleDataSource::SetClipboard
COleDataSource Class
COleServerItem::AddOtherClipboardData
COleServerItem::GetClipboardData
COleServerItem::OnDraw
CObject::Serialize

Other Resources

COleServerItem Members

COleServerItem::DoDragDrop

Call the **DoDragDrop** member function to perform a drag-and-drop operation.

```
DROPEFFECT DoDragDrop(
   LPCRECT lpRectItem,
   CPoint ptOffset,
   BOOL bIncludeLink = FALSE,
   DWORD dwEffects = DROPEFFECT_COPY | DROPEFFECT_MOVE,
   LPCRECT lpRectStartDrag = NULL
);
```

Parameters

lpRectItem

The item's rectangle on screen, in pixels, relative to the client area.

ptOffset

The offset from *lpItemRect* where the mouse position was at the time of the drag.

bIncludeLink

Set this to **TRUE** if link data should be copied to the Clipboard. Set it to **FALSE** if your application does not support links.

dwEffects

Determines the effects that the drag source will allow in the drag operation (a combination of Copy, Move, and Link).

lpRectStartDrag

Pointer to the rectangle that defines where the drag actually starts. For more information, see the following Remarks section.

Return Value

A value from the **DROPEFFECT** enumeration. If it is **DROPEFFECT_MOVE**, the original data should be removed.

Remarks

The drag-and-drop operation does not start immediately. It waits until the mouse cursor leaves the rectangle specified by *lpRectStartDrag* or until a specified number of milliseconds have passed. If *lpRectStartDrag* is **NULL**, a default rectangle is used so that the drag starts when the mouse cursor moves one pixel.

The delay time is specified by a registry key setting. You can change the delay time by calling CWinApp::WriteProfileString or CWinApp::WriteProfileInt. If you do not specify the delay time, a default value of 200 milliseconds is used. Drag delay time is stored as follows:

- Windows NT Drag delay time is stored in HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\NT\CurrentVersion\IniFileMapping\win.ini\Windows\DragDelay.
- Windows 3.x Drag delay time is stored in the WIN.INI file, under the [Windows} section.
- Windows 95/98 Drag delay time is stored in a cached version of WIN.INI.

For more information about how drag delay information is stored in either the registry or the .INI file, see WriteProfileString in the *Platform SDK*.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleDataSource::DoDragDrop
COleServerItem::CopyToClipboard

Other Resources

COleServerItem Members

COleServerItem::GetClipboardData

Call this function to fill the specified COleDataSource object with all the data that would be copied to the Clipboard if you called CopyToClipboard (the same data would also be transferred if you called DoDragDrop).

```
void GetClipboardData(
   COleDataSource* pDataSource,
   BOOL bIncludeLink = FALSE,
   LPPOINT lpOffset = NULL,
   LPSIZE lpSize = NULL
);
```

Parameters

pDataSource

Pointer to the **COleDataSource** object that will receive the OLE item's data in all supported formats.

bIncludeLink

TRUE if link data should be copied to the Clipboard. **FALSE** if your server application does not support links.

lpOffset

The offset, in pixels, of the mouse cursor from the origin of the object.

lpSize

The size of the object in pixels.

Remarks

This function calls the GetEmbedSourceData member function to get the native data for the OLE item and calls the AddOtherClipboardData member function to get the presentation format and any supported conversion formats. If *bIncludeLink* is **TRUE**, the function also calls GetLinkSourceData to get the link data for the item.

Override this function if you want to put formats in a **COleDataSource** object before or after those formats supplied by **CopyToClipboard**.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleDataSource Class
COleServerItem::AddOtherClipboardData
COleServerItem::CopyToClipboard
COleServerItem::DoDragDrop
COleServerItem::GetEmbedSourceData
COleServerItem::GetLinkSourceData

Other Resources

COleServerItem Members

COleServerItem::GetDataSource

Call this function to get the COleDataSource object used to store the conversion formats that the server application supports.

```
COleDataSource* GetDataSource( );
```

Return Value

A pointer to the **COleDataSource** object used to store the conversion formats.

Remarks

If you want your server application to offer data in a variety of formats during data transfer operations, register those formats with the **COleDataSource** object returned by this function. For example, if you want to supply a **CF_TEXT** representation of the OLE item for Clipboard or drag-and-drop operations, you would register the format with the **COleDataSource** object this function returns, and then override the **OnRenderXxxData** member function to provide the data.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleDataSource Class
COleDataSource::DelayRenderData
COleServerItem::CopyToClipboard
COleServerItem::DoDragDrop
COleServerItem::OnRenderData
COleServerItem::OnRenderFileData
COleServerItem::OnRenderGlobalData

Other Resources

COleServerItem Members

COleServerItem::GetDocument

Call this function to get a pointer to the document that contains the item.

```
COleServerDoc* GetDocument( ) const;
```

Return Value

A pointer to the document that contains the item; **NULL** if the item is not part of a document.

Remarks

This allows access to the server document that you passed as an argument to the **COleServerItem** constructor.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleServerItem::COleServerItem
COleServerDoc Class

Other Resources

COleServerItem Members

COleServerItem::GetEmbedSourceData

Call this function to get the **CF_EMBEDSOURCE** data for an OLE item.

```
void GetEmbedSourceData(
   LPSTGMEDIUM lpStgMedium
);
```

Parameters

lpStgMedium

Pointer to the STGMEDIUM structure that will receive the **CF_EMBEDSOURCE** data for the OLE item.

Remarks

This format includes the item's native data. You must have implemented the **Serialize** member function for this function to work properly.

The result can then be added to a data source by using COleDataSource::CacheData. This function is called automatically by COleServerItem::OnGetClipboardData.

For more information, see STGMEDIUM in the *Platform SDK*.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleServerItem::GetLinkSourceData
COleServerItem::GetObjectDescriptorData
COleDataSource::CacheData
CObject::Serialize

Other Resources

COleServerItem Members

COleServerItem::GetItemName

Call this function to get the name of the item.

```
const CString& GetItemName( ) const;
```

Return Value

The name of the item.

Remarks

You typically call this function only for linked items.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleServerItem::SetItemName
COleLinkingDoc::OnGetLinkedItem

Other Resources

COleServerItem Members

COleServerItem::GetLinkSourceData

Call this function to get the **CF_LINKSOURCE** data for an OLE item.

```
BOOL GetLinkSourceData(
   LPSTGMEDIUM lpStgMedium
);
```

Parameters

lpStgMedium

Pointer to the STGMEDIUM structure that will receive the **CF_LINKSOURCE** data for the OLE item.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This format includes the CLSID describing the type of the OLE item and the information needed to locate the document containing the OLE item.

The result can then be added to a data source with COleDataSource::CacheData. This function is called automatically by OnGetClipboardData.

For more information, see STGMEDIUM in the *Platform SDK*.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleServerItem::GetEmbedSourceData
COleServerItem::GetObjectDescriptorData

Other Resources

COleServerItem Members

COleServerItem::GetObjectDescriptorData

Call this function to get the **CF_OBJECTDESCRIPTOR** data for an OLE item.

```
void GetObjectDescriptorData(
   LPPOINT lpOffset,
   LPSIZE lpSize,
   LPSTGMEDIUM lpStgMedium
);
```

Parameters

lpOffset

Offset of the mouse click from the upper-left corner of the OLE item. Can be **NULL**.

lpSize

Size of the OLE item. Can be **NULL**.

lpStgMedium

Pointer to the STGMEDIUM structure that will receive the **CF_OBJECTDESCRIPTOR** data for the OLE item.

Remarks

The information is copied into the **STGMEDIUM** structure pointed to by *lpStgMedium*. This format includes the information needed for the Paste Special dialog.

For more information, see STGMEDIUM in the *Platform SDK*.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleServerItem::AddOtherClipboardData
COleServerItem::GetEmbedSourceData
COleServerItem::GetLinkSourceData
COlePasteSpecialDialog Class

Other Resources

COleServerItem Members

COleServerItem::IsConnected

Call this function to see if the OLE item is connected.

```
BOOL IsConnected( ) const;
```

Return Value

Nonzero if the item is connected; otherwise 0.

Remarks

An OLE item is considered connected if one or more containers have references to the item. An item is connected if its reference count is greater than 0 or if it is an embedded item.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleServerItem::IsLinkedItem
COleLinkingDoc::OnGetLinkedItem

Other Resources

COleServerItem Members

COleServerItem::IsLinkedItem

Call this function to see if the OLE item is a linked item.

```
BOOL IsLinkedItem( ) const;
```

Return Value

Nonzero if the item is a linked item; otherwise 0.

Remarks

An item is linked if the item is valid and is not returned in the document's list of embedded items. A linked item might or might not be connected to a container.

It is common to use the same class for both linked and embedded items. **IsLinkedItem** allows you to make linked items behave differently than embedded items, although many times the code is common.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleServerItem::IsConnected
COleLinkingDoc::OnGetLinkedItem

Other Resources

COleServerItem Members

COleServerItem::NotifyChanged

Call this function after the linked item has been changed.

```
void NotifyChanged(
   DVASPECT nDrawAspect = DVASPECT_CONTENT
);
```

Parameters

nDrawAspect

A value from the **DVASPECT** enumeration that indicates which aspect of the OLE item has changed. This parameter can have any of the following values:

- **DVASPECT_CONTENT** Item is represented in such a way that it can be displayed as an embedded object inside its container.
- **DVASPECT_THUMBNAIL** Item is rendered in a "thumbnail" representation so that it can be displayed in a browsing tool.
- **DVASPECT_ICON** Item is represented by an icon.
- **DVASPECT_DOCPRINT** Item is represented as if it were printed using the Print command from the File menu.

Remarks

If a container item is linked to the document with an automatic link, the item is updated to reflect the changes. In container applications written using the Microsoft Foundation Class Library, COleClientItem::OnChange is called in response.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleClientItem::OnChange
COleServerItem::OnUpdate
COleServerDoc::NotifyChanged

Other Resources

COleServerItem Members

COleServerItem::OnDoVerb

Called by the framework to execute the specified verb.

```
virtual void OnDoVerb(
   LONG iVerb
);
```

Parameters

iVerb

Specifies the verb to execute. It can be any one of the following:

Value	Meaning	Symbol
0	Primary verb	**OLEIVERB_PRIMARY**
1	Secondary verb	(None)
– 1	Display item for editing	**OLEIVERB_SHOW**
– 2	Edit item in separate window	**OLEIVERB_OPEN**
– 3	Hide item	**OLEIVERB_HIDE**

The –1 value is typically an alias for another verb. If open editing is not supported, –2 has the same effect as –1. For additional values, see IOleObject::DoVerb in the *Platform SDK*.

Remarks

If the container application was written with the Microsoft Foundation Class Library, this function is called when the COleClientItem::Activate member function of the corresponding **COleClientItem** object is called. The default implementation calls the OnShow member function if the primary verb or **OLEIVERB_SHOW** is specified, OnOpen if the secondary verb or **OLEIVERB_OPEN** is specified, and OnHide if **OLEIVERB_HIDE** is specified. The default implementation calls **OnShow** if *iVerb* is not one of the verbs listed above.

Override this function if your primary verb does not show the item. For example, if the item is a sound recording and its primary verb is Play, you would not have to display the server application to play the item.

For more information, see IOleObject::DoVerb in the *Platform SDK*.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleClientItem::Activate
COleServerItem::OnShow
COleServerItem::OnOpen
COleServerItem::OnHide

Other Resources

COleServerItem Members

COleServerItem::OnDraw

Called by the framework to render the OLE item into a metafile.

```
virtual BOOL OnDraw(
   CDC* pDC,
   CSize& rSize
) = 0;
```

Parameters

pDC

A pointer to the CDC object on which to draw the item. The display context is automatically connected to the attribute display context so you can call attribute functions, although doing so would make the metafile device-specific.

rSize

Size, in **HIMETRIC** units, in which to draw the metafile.

Return Value

Nonzero if the item was successfully drawn; otherwise 0.

Remarks

The metafile representation of the OLE item is used to display the item in the container application. If the container application was written with the Microsoft Foundation Class Library, the metafile is used by the Draw member function of the corresponding COleClientItem object. There is no default implementation. You must override this function to draw the item into the device context specified.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleClientItem::Draw

Other Resources

COleServerItem Members

COleServerItem::OnDrawEx

Called by the framework for all drawing.

```
virtual BOOL OnDrawEx(
   CDC* pDC,
   DVASPECT nDrawAspect,
   CSize& rSize
);
```

Parameters

pDC

A pointer to the CDC object on which to draw the item. The DC is automatically connected to the attribute DC so you can call attribute functions, although doing so would make the metafile device-specific.

nDrawAspect

A value from the **DVASPECT** enumeration. This parameter can have any of the following values:

- **DVASPECT_CONTENT** Item is represented in such a way that it can be displayed as an embedded object inside its container.
- **DVASPECT_THUMBNAIL** Item is rendered in a "thumbnail" representation so that it can be displayed in a browsing tool.
- **DVASPECT_ICON** Item is represented by an icon.
- **DVASPECT_DOCPRINT** Item is represented as if it were printed using the Print command from the File menu.

rSize

Size of the item in **HIMETRIC** units.

Return Value

Nonzero if the item was successfully drawn; otherwise 0.

Remarks

The default implementation calls **OnDraw** when **DVASPECT** is equal to **DVASPECT_CONTENT**; otherwise it fails.

Override this function to provide presentation data for aspects other than **DVASPECT_CONTENT**, such as **DVASPECT_ICON** or **DVASPECT_THUMBNAIL**.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleServerItem::OnDraw

Other Resources

COleServerItem Members

COleServerItem::OnGetClipboardData

Called by the framework to get a **COleDataSource** object containing all the data that would be placed on the Clipboard by a call to the CopyToClipboard member function.

```
virtual COleDataSource* OnGetClipboardData(
   BOOL bIncludeLink,
   LPPOINT lpOffset,
   LPSIZE lpSize
);
```

Parameters

bIncludeLink

Set this to **TRUE** if link data should be copied to the Clipboard. Set this to **FALSE** if your server application does not support links.

lpOffset

The offset of the mouse cursor from the origin of the object in pixels.

lpSize

The size of the object in pixels.

Return Value

A pointer to a COleDataSource object containing the Clipboard data.

Remarks

The default implementation of this function calls GetClipboardData.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleDataSource Class
COleDataSource::SetClipboard
COleServerItem::CopyToClipboard
COleServerItem::GetClipboardData

Other Resources

COleServerItem Members

COleServerItem::OnGetExtent

Called by the framework to retrieve the size, in **HIMETRIC** units, of the OLE item.

```
virtual BOOL OnGetExtent(
   DVASPECT nDrawAspect,
   CSize& rSize
);
```

Parameters

nDrawAspect

Specifies the aspect of the OLE item whose bounds are to be retrieved. This parameter can have any of the following values:

- **DVASPECT_CONTENT** Item is represented in such a way that it can be displayed as an embedded object inside its container.
- **DVASPECT_THUMBNAIL** Item is rendered in a "thumbnail" representation so that it can be displayed in a browsing tool.
- **DVASPECT_ICON** Item is represented by an icon.
- **DVASPECT_DOCPRINT** Item is represented as if it were printed using the Print command from the File menu.

rSize

Reference to a **CSize** object that will receive the size of the OLE item.

Return Value

Nonzero if successful; otherwise 0.

Remarks

If the container application was written with the Microsoft Foundation Class Library, this function is called when the GetExtent member function of the corresponding **COleClientItem** object is called. The default implementation does nothing. You must implement it yourself. Override this function if you want to perform special processing when handling a request for the size of the OLE item.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleClientItem::Draw
COleClientItem::GetExtent

Other Resources

COleServerItem Members

COleServerItem::OnHide

Called by the framework to hide the OLE item.

```
virtual void OnHide( );
```

Remarks

The default calls **COleServerDoc::OnShowDocument(FALSE)**. The function also notifies the container that the OLE item has been hidden. Override this function if you want to perform special processing when hiding an OLE item.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleServerItem::OnOpen
COleServerItem::OnShow
COleServerDoc::OnShowDocument

Other Resources

COleServerItem Members

COleServerItem::OnInitFromData

Called by the framework to initialize an OLE item using the contents of *pDataObject*.

```
virtual BOOL OnInitFromData(
   COleDataObject* pDataObject,
   BOOL bCreation
);
```

Parameters

pDataObject

Pointer to an OLE data object containing data in various formats for initializing the OLE item.

bCreation

TRUE if the function is called to initialize an OLE item being newly created by a container application. **FALSE** if the function is called to replace the contents of an already existing OLE item.

Return Value

Nonzero if successful; otherwise 0.

Remarks

If *bCreation* is **TRUE**, this function is called if a container implements Insert New Object based on the current selection. The data selected is used when creating the new OLE item. For example, when selecting a range of cells in a spreadsheet program and then using the Insert New Object to create a chart based on the values in the selected range. The default implementation does nothing. Override this function to choose an acceptable format from those offered by *pDataObject* and initialize the OLE item based on the data provided. This is an advanced overridable.

For more information, see IOleObject::InitFromData in the *Platform SDK*.

See Also

Reference

COleServerItem Class

Hierarchy Chart

Other Resources

COleServerItem Members

COleServerItem::OnOpen

Called by the framework to display the OLE item in a separate instance of the server application, rather than in place.

```
virtual void OnOpen( );
```

Remarks

The default implementation activates the first frame window displaying the document that contains the OLE item; if the application is a mini-server, the default implementation shows the main window. The function also notifies the container that the OLE item has been opened.

Override this function if you want to perform special processing when opening an OLE item. This is especially common with linked items where you want to set the selection to the link when it is opened.

For more information, see IOleClientSite::OnShowWindow in the *Platform SDK*.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleServerItem::OnShow

Other Resources

COleServerItem Members

COleServerItem::OnQueryUpdateItems

Called by the framework to determine whether any linked items in the current server document are out of date.

```
virtual BOOL OnQueryUpdateItems( );
```

Return Value

Nonzero if the document has items needing updates; 0 if all items are up to date.

Remarks

An item is out of date if its source document has been changed but the linked item has not been updated to reflect the changes in the document.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleServerItem::OnUpdate
COleServerItem::OnUpdateItems

Other Resources

COleServerItem Members

COleServerItem::OnRenderData

Called by the framework to retrieve data in the specified format.

```
virtual BOOL OnRenderData(
   LPFORMATETC lpFormatEtc,
   LPSTGMEDIUM lpStgMedium
);
```

Parameters

lpFormatEtc

Points to the FORMATETC structure specifying the format in which information is requested.

lpStgMedium

Points to a STGMEDIUM structure in which the data is to be returned.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The specified format is one previously placed in the **COleDataSource** object using the DelayRenderData or DelayRenderFileData member function for delayed rendering. The default implementation of this function calls OnRenderFileData or OnRenderGlobalData, respectively, if the supplied storage medium is either a file or memory. If neither of these formats is supplied, the default implementation returns 0 and does nothing.

If *lpStgMedium->tymed* is **TYMED_NULL**, the **STGMEDIUM** should allocated and filled as specified by *lpFormatEtc->tymed*. If not **TYMED_NULL**, the **STGMEDIUM** should be filled in place with the data.

This is an advanced overridable. Override this function to provide your data in the requested format and medium. Depending on your data, you may want to override one of the other versions of this function instead. If your data is small and fixed in size, override **OnRenderGlobalData**. If your data is in a file, or is of variable size, override **OnRenderFileData**.

For more information, see IDataObject::GetData, STGMEDIUM, FORMATETC, and TYMED in the *Platform SDK*.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleServerItem::OnRenderFileData

Other Resources

COleServerItem Members

COleServerItem::OnRenderFileData

Called by the framework to retrieve data in the specified format when the storage medium is a file.

```
virtual BOOL OnRenderFileData(
   LPFORMATETC lpFormatEtc,
   CFile* pFile
);
```

Parameters

lpFormatEtc

Points to the FORMATETC structure specifying the format in which information is requested.

pFile

Points to a **CFile** object in which the data is to be rendered.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The specified format is one previously placed in the **COleDataSource** object using the DelayRenderData member function for delayed rendering. The default implementation of this function simply returns **FALSE**.

This is an advanced overridable. Override this function to provide your data in the requested format and medium. Depending on your data, you might want to override one of the other versions of this function instead. If you want to handle multiple storage mediums, override OnRenderData. If your data is in a file, or is of variable size, override OnRenderFileData.

For more information, see IDataObject::GetData and FORMATETC in the *Platform SDK*.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleServerItem::OnRenderData

Other Resources

COleServerItem Members

COleServerItem::OnRenderGlobalData

Called by the framework to retrieve data in the specified format when the specified storage medium is global memory.

```
virtual BOOL OnRenderGlobalData(
   LPFORMATETC lpFormatEtc,
   HGLOBAL* phGlobal
);
```

Parameters

lpFormatEtc

Points to the FORMATETC structure specifying the format in which information is requested.

phGlobal

Points to a handle to global memory in which the data is to be returned. If no memory has been allocated, this parameter can be **NULL**.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The specified format is one previously placed in the **COleDataSource** object using the DelayRenderData member function for delayed rendering. The default implementation of this function simply returns **FALSE**.

If *phGlobal* is **NULL**, then a new **HGLOBAL** should be allocated and returned in *phGlobal*. Otherwise, the **HGLOBAL** specified by *phGlobal* should be filled with the data. The amount of data placed in the **HGLOBAL** must not exceed the current size of the memory block. Also, the block cannot be reallocated to a larger size.

This is an advanced overridable. Override this function to provide your data in the requested format and medium. Depending on your data, you may want to override one of the other versions of this function instead. If you want to handle multiple storage mediums, override OnRenderData. If your data is in a file, or is of variable size, override OnRenderFileData.

For more information, see IDataObject::GetData and FORMATETC in the *Platform SDK*.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleServerItem::OnRenderData

Other Resources

COleServerItem Members

COleServerItem::OnSetColorScheme

Called by the framework to specify a color palette to be used when editing the OLE item.

```
virtual BOOL OnSetColorScheme(
   const LOGPALETTE* lpLogPalette
);
```

Parameters

lpLogPalette

Pointer to a Windows LOGPALETTE structure.

Return Value

Nonzero if the color palette is used; otherwise 0.

Remarks

If the container application was written using the Microsoft Foundation Class Library, this function is called when the IOleObject::SetColorScheme function of the corresponding **COleClientItem** object is called. The default implementation returns **FALSE**. Override this function if you want to use the recommended palette. The server application is not required to use the suggested palette.

For more information, see IOleObject::SetColorScheme in the *Platform SDK*.

See Also

Reference

COleServerItem Class

Hierarchy Chart

Other Resources

COleServerItem Members

COleServerItem::OnSetData

Called by the framework to replace the OLE item's data with the specified data.

```
virtual BOOL OnSetData(
   LPFORMATETC lpFormatEtc,
   LPSTGMEDIUM lpStgMedium,
   BOOL bRelease
);
```

Parameters

lpFormatEtc

Pointer to a FORMATETC structure specifying the format of the data.

lpStgMedium

Pointer to a STGMEDIUM structure in which the data resides.

bRelease

Indicates who has ownership of the storage medium after completing the function call. The caller decides who is responsible for releasing the resources allocated on behalf of the storage medium. The caller does this by setting *bRelease*. If *bRelease* is nonzero, the server item takes ownership, freeing the medium when it has finished using it. When *bRelease* is 0, the caller retains ownership and the server item can use the storage medium only for the duration of the call.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The server item does not take ownership of the data until it has successfully obtained it. That is, it does not take ownership if it returns 0. If the data source takes ownership, it frees the storage medium by calling the ReleaseStgMedium function.

The default implementation does nothing. Override this function to replace the OLE item's data with the specified data. This is an advanced overridable.

For more information, see STGMEDIUM, FORMATETC, and ReleaseStgMedium in the *Platform SDK*.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleDataSource::OnSetData

Other Resources

COleServerItem Members

COleServerItem::OnSetExtent

Called by the framework to tell the OLE item how much space is available to it in the container document.

```
virtual BOOL OnSetExtent(
   DVASPECT nDrawAspect,
   const CSize& size
);
```

Parameters

nDrawAspect

Specifies the aspect of the OLE item whose bounds are being specified. This parameter can have any of the following values:

- **DVASPECT_CONTENT** Item is represented in such a way that it can be displayed as an embedded object inside its container.
- **DVASPECT_THUMBNAIL** Item is rendered in a "thumbnail" representation so that it can be displayed in a browsing tool.
- **DVASPECT_ICON** Item is represented by an icon.
- **DVASPECT_DOCPRINT** Item is represented as if it were printed using the Print command from the File menu.

size

A CSize structure specifying the new size of the OLE item.

Return Value

Nonzero if successful; otherwise 0.

Remarks

If the container application was written with the Microsoft Foundation Class Library, this function is called when the SetExtent member function of the corresponding **COleClientItem** object is called. The default implementation sets the m_sizeExtent member to the specified size if *nDrawAspect* is **DVASPECT_CONTENT**; otherwise it returns 0. Override this function to perform special processing when you change the size of the item.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleClientItem::SetExtent
COleServerItem::OnGetExtent
COleServerItem::m_sizeExtent

Other Resources

COleServerItem Members

COleServerItem::OnShow

Called by the framework to instruct the server application to display the OLE item in place.

```
virtual void OnShow( );
```

Remarks

This function is typically called when the user of the container application creates an item or executes a verb, such as Edit, that requires the item to be shown. The default implementation attempts in-place activation. If this fails, the function calls the **OnOpen** member function to display the OLE item in a separate window.

Override this function if you want to perform special processing when an OLE item is shown.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleServerItem::OnOpen
COleClientItem::Activate

Other Resources

COleServerItem Members

COleServerItem::OnUpdate

Called by the framework when an item has been modified.

```
virtual void OnUpdate(
   COleServerItem* pSender,
   LPARAM lHint,
   CObject* pHint,
   DVASPECT nDrawAspect
);
```

Parameters

pSender

Pointer to the item that modified the document. Can be **NULL**.

lHint

Contains information about the modification.

pHint

Pointer to an object storing information about the modification.

nDrawAspect

A value from the **DVASPECT** enumeration. This parameter can have any one of the following values:

- **DVASPECT_CONTENT** Item is represented in such a way that it can be displayed as an embedded object inside its container.
- **DVASPECT_THUMBNAIL** Item is rendered in a "thumbnail" representation so that it can be displayed in a browsing tool.
- **DVASPECT_ICON** Item is represented by an icon.
- **DVASPECT_DOCPRINT** Item is represented as if it were printed using the Print command from the File menu.

Remarks

The default implementation calls NotifyChanged, regardless of the hint or sender.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleServerItem::NotifyChanged

Other Resources

COleServerItem Members

COleServerItem::OnUpdateItems

Called by the framework to update all items in the server document.

```
virtual void OnUpdateItems( );
```

Remarks

The default implementation calls UpdateLink for all **COleClientItem** objects in the document.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleServerItem::OnUpdate
COleServerItem::OnQueryUpdateItems

Other Resources

COleServerItem Members

COleServerItem::SetItemName

Call this function when you create a linked item to set its name.

```
void SetItemName(
   LPCTSTR lpszItemName
);
```

Parameters

lpszItemName

Pointer to the new name of the item.

Remarks

The name must be unique within the document. When a server application is called to edit a linked item, the application uses this name to find the item. You do not need to call this function for embedded items.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleServerItem::GetItemName
COleLinkingDoc::OnGetLinkedItem

Other Resources

COleServerItem Members

COleServerItem Data Members

For information about the data members in **COleServerItem**, see COleServerItem Members.

COleServerItem::m_sizeExtent

This member tells the server how much of the object is visible in the container document.

```
CSize m_sizeExtent;
```

Remarks

The default implementation of OnSetExtent sets this member.

See Also

Reference

COleServerItem Class
Hierarchy Chart
COleServerItem::OnSetExtent

Other Resources

COleServerItem Members

COleStreamFile Class

Represents a stream of data (**IStream**) in a compound file as part of OLE Structured Storage.

```
class COleStreamFile : public CFile
```

Remarks

An **IStorage** object must exist before the stream can be opened or created unless it is a memory stream.

COleStreamFile objects are manipulated exactly like CFile objects.

For more information about manipulating streams and storages, see the article Containers: Compound Files..

For more information, see IStream and IStorage in the *Platform SDK*.

Requirements

Header: afxole.h

See Also

Reference

CFile Class

Hierarchy Chart

Other Resources

COleStreamFile Members

COleStreamFile Members

Base Class Members

CObject Members

CFile Members

Construction

COleStreamFile	Constructs a **COleStreamFile** object.

Attributes and Operations

Attach	Associates a stream with the object.
CreateMemoryStream	Creates a stream from global memory and associates it with the object.
CreateStream	Creates a stream and associates it with the object.
Detach	Disassociates the stream from the object.
GetStream	Returns the current stream.
OpenStream	Safely opens a stream and associates it with the object.

See Also

Reference

COleStreamFile Class

Hierarchy Chart

COleStreamFile Member Functions

For information about the member functions in **COleStreamFile**, see COleStreamFile Members.

COleStreamFile::Attach

Associates the supplied OLE stream with the **COleStreamFile** object.

```
void Attach(
   LPSTREAM lpStream
);
```

Parameters

lpStream

Points to the OLE stream (**IStream**) to be associated with the object. Cannot be **NULL**.

Remarks

The object must not already be associated with an OLE stream.

For more information, see IStream in the *Platform SDK*.

See Also

Reference

COleStreamFile Class
Hierarchy Chart
COleStreamFile::Detach

Other Resources

COleStreamFile Members

COleStreamFile::COleStreamFile

Creates a **COleStreamFile** object.

```
COleStreamFile(
   LPSTREAM lpStream = NULL
);
```

Parameters

lpStream

Pointer to the OLE stream to be associated with the object.

Remarks

If *lpStream* is **NULL**, the object is not associated with an OLE stream, otherwise, the object is associated with the supplied OLE stream.

For more information, see IStream in the *Platform SDK*.

See Also

Reference

COleStreamFile Class
Hierarchy Chart
COleStreamFile::Attach
CFile Class

Other Resources

COleStreamFile Members

COleStreamFile::CreateMemoryStream

Safely creates a new stream out of global, shared memory where a failure is a normal, expected condition.

```
BOOL CreateMemoryStream(
   CFileException* pError = NULL
);
```

Parameters

pError

Points to a CFileException object or **NULL** that indicates the completion status of the create operation. Supply this parameter if you want to monitor possible exceptions generated by attempting to create the stream.

Return Value

Nonzero if the stream is created successfully; otherwise 0.

Remarks

The memory is allocated by the OLE subsystem.

For more information, see CreateStreamOnHGlobal in the *Platform SDK*.

See Also

Reference

COleStreamFile Class
Hierarchy Chart
COleStreamFile::OpenStream
COleStreamFile::CreateStream
CFileException Class

Other Resources

COleStreamFile Members

COleStreamFile::CreateStream

Safely creates a new stream in the supplied storage object where a failure is a normal, expected condition.

```
BOOL CreateStream(
   LPSTORAGE lpStorage,
   LPCTSTR lpszStreamName,
   DWORD nOpenFlags = modeReadWrite|shareExclusive|modeCreate,
   CFileException* pError = NULL
);
```

Parameters

lpStorage

Points to the OLE storage object that contains the stream to be created. Cannot be **NULL**.

lpszStreamName

Name of the stream to be created. Cannot be **NULL**.

nOpenFlags

Access mode to use when opening the stream. Exclusive, read/write, and create modes are used by default. For a complete list of the available modes, see CFile::CFile.

pError

Points to a CFileException object or **NULL**. Supply this parameter if you want to monitor possible exceptions generated by attempting to create the stream.

Return Value

Nonzero if the stream is created successfully; otherwise 0.

Remarks

A file exception will be thrown if the open fails and *pError* is not **NULL**.

For more information, see IStorage::CreateStream in the *Platform SDK*.

See Also

Reference

COleStreamFile Class
Hierarchy Chart
COleStreamFile::OpenStream
COleStreamFile::CreateMemoryStream
CFileException Class

Other Resources

COleStreamFile Members

COleStreamFile::Detach

Disassociates the stream from the object without closing the stream.

```
LPSTREAM Detach( );
```

Return Value

A pointer to the stream (**IStream**) that was associated with the object.

Remarks

The stream must be closed in some other fashion before the program terminates.

For more information, see IStream in the *Platform SDK*.

See Also

Reference

COleStreamFile Class
Hierarchy Chart
COleStreamFile::Attach

Other Resources

COleStreamFile Members

COleStreamFile::GetStream

Call this function to return a pointer to current stream.

```
IStream* GetStream( ) const;
```

Return Value

A pointer to the current stream interface (IStream).

See Also

Reference

COleStreamFile Class

Hierarchy Chart

Other Resources

COleStreamFile Members

COleStreamFile::OpenStream

Opens an existing stream.

```
BOOL OpenStream(
   LPSTORAGE lpStorage,
   LPCTSTR lpszStreamName,
   DWORD nOpenFlags = modeReadWrite|shareExclusive,
   CFileException* pError = NULL
);
```

Parameters

lpStorage

Points to the OLE storage object that contains the stream to be opened. Cannot be **NULL**.

lpszStreamName

Name of the stream to be opened. Cannot be **NULL**.

nOpenFlags

Access mode to use when opening the stream. Exclusive and read/write modes are used by default. For the complete list of the available modes, see CFile::CFile.

pError

Points to a CFileException object or **NULL**. Supply this parameter if you want to monitor possible exceptions generated by attempting to open the stream.

Return Value

Nonzero if the stream is opened successfully; otherwise 0.

Remarks

A file exception will be thrown if the open fails and *pError* is not **NULL**.

For more information, see IStorage::OpenStream in the *Platform SDK*.

See Also

Reference

COleStreamFile Class
Hierarchy Chart
COleStreamFile::CreateStream
COleStreamFile::CreateMemoryStream
CFileException Class

Other Resources

COleStreamFile Members

COleTemplateServer Class

Used for OLE visual editing servers, automation servers, and link containers (applications that support links to embeddings).

```
class COleTemplateServer : public COleObjectFactory
```

Remarks

This class is derived from the class COleObjectFactory; usually, you can use **COleTemplateServer** directly rather than deriving your own class. **COleTemplateServer** uses a CDocTemplate object to manage the server documents. Use **COleTemplateServer** when implementing a full server, that is, a server that can be run as a standalone application. Full servers are typically multiple document interface (MDI) applications, although single document interface (SDI) applications are supported. One **COleTemplateServer** object is needed for each type of server document an application supports; that is, if your server application supports both worksheets and charts, you must have two **COleTemplateServer** objects.

COleTemplateServer overrides the **OnCreateInstance** member function defined by **COleObjectFactory**. This member function is called by the framework to create a C++ object of the proper type.

For more information about servers, see the article Servers: Implementing a Server..

Requirements

Header: afxdisp.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

HIERSVR Sample: Demonstrates a Server Application with OLE Drag and Drop

Reference

COleObjectFactory Class
Hierarchy Chart
COleServerDoc Class
COleServerItem Class

Other Resources

COleTemplateServer Members

COleTemplateServer Members

Base Class Members

CObject Members

CCmdTarget Members

COleObjectFactory Members

Construction

COleTemplateServer	Constructs a **COleTemplateServer** object.

Operations

ConnectTemplate	Connects a document template to the underlying **COleObjectFactory** object.
Unregister	Unregisters the associated document template.
UpdateRegistry	Registers the document type with the OLE system registry.

See Also

Reference

COleTemplateServer Class

Hierarchy Chart

COleTemplateServer Member Functions

For information about the member functions in **COleTemplateServer**, see COleTemplateServer Members.

COleTemplateServer::COleTemplateServer

Constructs a **COleTemplateServer** object.

```
COleTemplateServer( );
```

Remarks

For a brief description of the use of the **COleTemplateServer** class, see the COleLinkingDoc class overview.

See Also

Reference

COleTemplateServer Class

Hierarchy Chart

Other Resources

COleTemplateServer Members

COleTemplateServer::ConnectTemplate

Connects the document template pointed to by *pDocTemplate* to the underlying COleObjectFactory object.

```
void ConnectTemplate(
   REFCLSID clsid,
   CDocTemplate* pDocTemplate,
   BOOL bMultiInstance
);
```

Parameters

clsid

Reference to the OLE class ID that the template requests.

pDocTemplate

Pointer to the document template.

bMultiInstance

Indicates whether a single instance of the application can support multiple instantiations. If **TRUE**, multiple instances of the application are launched for each request to create an object.

Remarks

For more information, see CLSID Key in the *Platform SDK*.

See Also

Reference

COleTemplateServer Class
Hierarchy Chart
CDocTemplate Class

Other Resources

COleTemplateServer Members

COleTemplateServer::Unregister

Unregisters the associated document template.

```
BOOL Unregister( );
```

Return Value

TRUE if successful; otherwise FALSE.

Remarks

EnterRemarks

See Also

Reference

COleTemplateServer Class

Hierarchy Chart

CDocTemplate Class

Other Resources

COleTemplateServer Members

COleTemplateServer::UpdateRegistry

Loads file-type information from the document-template string and places that information in the OLE system registry.

```
void UpdateRegistry(
   OLE_APPTYPE nAppType = OAT_INPLACE_SERVER,
   LPCTSTR* rglpszRegister = NULL,
   LPCTSTR* rglpszOverwrite = NULL,
   BOOL bRegister = TRUE
);
```

Parameters

nAppType

A value from the **OLE_APPTYPE** enumeration, which is defined in AFXDISP.H. It can have any one of the following values:

- **OAT_INPLACE_SERVER** Server has full server user-interface.
- **OAT_SERVER** Server supports only embedding.
- **OAT_CONTAINER** Container supports links to embedded objects.
- **OAT_DISPATCH_OBJECT** Object is **IDispatch**-capable.
- **OAT_DOC_OBJECT_SERVER** Server supports both embedding and the Document Object component model.

rglpszRegister

A list of entries that is written into the registry only if no entries exist.

rglpszOverwrite

A list of entries that is written into the registry regardless of whether any preceding entries exist.

bRegister

Determines whether the class is to be registered. If *bRegister* is **TRUE**, the class is registered with the system registry. Otherwise, it unregisters the class.

Remarks

The registration information is loaded by means of a call to CDocTemplate::GetDocString. The substrings retrieved are those identified by the indexes **regFileTypeId**, **regFileTypeName**, and **fileNewName**, as described in the **GetDocString** reference pages.

If the **regFileTypeId** substring is empty or if the call to **GetDocString** fails for any other reason, this function fails and the file information is not entered in the registry.

The information in the arguments *rglpszRegister* and *rglpszOverwrite* is written to the registry through a call to AfxOleRegisterServerClass. The default information, which is registered when the two arguments are **NULL**, is suitable for most applications. For information on the structure of the information in these arguments, see **AfxOleRegisterServerClass**.

For more information, see Implementing the IDispatch Interface.

See Also

Reference

COleTemplateServer Class
Hierarchy Chart
CDocTemplate::GetDocString
AfxOleRegisterServerClass

Other Resources

COleTemplateServer Members

COleUpdateDialog Class

Used for a special case of the OLE Edit Links dialog box, which should be used when you need to update only existing linked or embedded objects in a document.

```
class COleUpdateDialog : public COleLinksDialog
```

Remarks

For more information regarding OLE-specific dialog boxes, see the article Dialog Boxes in OLE.

Requirements

Header: afxodlgs.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

OCLIENT Sample: Illustrates a Visual Editing Container Application

Reference

COleLinksDialog Class

Hierarchy Chart

COleLinksDialog Class

Other Resources

COleUpdateDialog Members

COleUpdateDialog Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CDialog Members

CCommonDialog Members

COleDialog Members

COleLinksDialog Members

Construction

COleUpdateDialog	Constructs a **COleUpdateDialog** object.

Operations

DoModal	Displays the Edit Links dialog box in an update mode.

See Also

Reference

COleUpdateDialog Class

Hierarchy Chart

COleUpdateDialog Member Functions

For information about the member functions in **COleUpdateDialog**, see COleUpdateDialog Members.

COleUpdateDialog::COleUpdateDialog

Constructs a **COleUpdateDialog** object.

```
explicit COleUpdateDialog(
   COleDocument* pDoc,
   BOOL bUpdateLinks = TRUE,
   BOOL bUpdateEmbeddings = FALSE,
   CWnd* pParentWnd = NULL
);
```

Parameters

pDoc

Points to the document containing the links that may need updating.

bUpdateLinks

Flag that determines whether linked objects are to be updated.

bUpdateEmbeddings

Flag that determines whether embedded objects are to be updated.

pParentWnd

Points to the parent or owner window object (of type **CWnd**) to which the dialog object belongs. If it is **NULL**, the parent window of the dialog box will be set to the main application window.

Remarks

This function constructs only a **COleUpdateDialog** object. To display the dialog box, call DoModal. This class should be used instead of **COleLinksDialog** when you want to update only existing linked or embedded items.

See Also

Reference

COleUpdateDialog Class
Hierarchy Chart
COleDialog Class
COleLinksDialog Class
COleDocument Class
CWnd Class
CDialog Class
COleUpdateDialog::DoModal

Other Resources

COleUpdateDialog Members

COleUpdateDialog::DoModal

Displays the Edit Links dialog box in update mode.

```
virtual INT_PTR DoModal( );
```

Return Value

Completion status for the dialog box. One of the following values:

- **IDOK** if the dialog box returned successfully.
- **IDCANCEL** if none of the linked or embedded items in the current document need updating.
- **IDABORT** if an error occurred. If **IDABORT** is returned, call the COleDialog::GetLastError member function to get more information about the type of error that occurred. For a listing of possible errors, see the OleUIEditLinks function in the *Platform SDK*.

Remarks

All links and/or embeddings are updated unless the user selects the Cancel button.

See Also

Reference

COleUpdateDialog Class
Hierarchy Chart
COleDialog::GetLastError
COleLinksDialog::DoModal

Other Resources

COleUpdateDialog Members

COleVariant Class

Encapsulates the VARIANT data type.

```
class COleVariant : public tagVARIANT
```

Remarks

This data type is used in OLE automation. Specifically, the DISPPARAMS structure contains a pointer to an array of **VARIANT** structures. A **DISPPARAMS** structure is used to pass parameters to IDispatch::Invoke.

Note
This class is derived from the **VARIANT** structure. This means you can pass a **COleVariant** in a parameter that calls for a **VARIANT** and that the data members of the **VARIANT** structure are accessible data members of **COleVariant**.

The two related MFC classes COleCurrency and COleDateTime encapsulate the variant data types **CURRENCY** (**VT_CY**) and **DATE** (**VT_DATE**). The **COleVariant** class is used extensively in the DAO classes; see these classes for typical usage of this class, for example CDaoQueryDef and CDaoRecordset.

For more information, see the VARIANT, CURRENCY, DISPPARAMS, and IDispatch::Invoke entries in the *Platform SDK*.

For more information on the **COleVariant** class and its use in OLE automation, see "Passing Parameters in OLE Automation" in the article Automation.

Requirements

Header: afxdisp.h

See Also

Reference

Hierarchy Chart

Other Resources

COleVariant Members

COleVariant Members

Construction

COleVariant	Constructs a **COleVariant** object.

Operations

Attach	Attaches a **VARIANT** to a **COleVariant**.
ChangeType	Changes the variant type of this **COleVariant** object.
Clear	Clears this **COleVariant** object.
Detach	Detaches a **VARIANT** from a **COleVariant** and returns the **VARIANT**.
GetByteArrayFromVariantArray	Retrieves a byte array from an existing variant array.
SetString	Sets the string to a particular type, typically ANSI.

Operators

operator =	Copies a **COleVariant** value.
operator ==	Compares two **COleVariant** values.
operator LPCVARIANT	Converts a **COleVariant** value into an **LPCVARIANT**.
operator LPVARIANT	Converts a **COleVariant** object into an **LPVARIANT**.

Archive/Dump

operator <<	Outputs a **COleVariant** value to **CArchive** or **CDumpContext**.
operator >>	Inputs a **COleVariant** object from **CArchive**.

See Also

Reference

COleVariant Class

Hierarchy Chart

COleVariant Member Functions

For information about the member functions in **COleVariant**, see COleVariant Members.

COleVariant::Attach

Call this function to attach the given VARIANT object to the current **COleVariant** object.

```
void Attach(
   VARIANT& varSrc
);
```

Parameters

varSrc

An existing **VARIANT** object to be attached to the current **COleVariant** object.

Remarks

This function sets the VARTYPE of *varSrc* to **VT_EMPTY**.

For more information, see the VARIANT and VARTYPE entries in the *Platform SDK*.

See Also

Reference

COleVariant Class
Hierarchy Chart
COleVariant::operator LPCVARIANT
COleVariant::operator LPVARIANT

Other Resources

COleVariant Members

COleVariant::COleVariant

Constructs a **COleVariant** object.

```
COleVariant( );
COleVariant(
   const VARIANT& varSrc
);
COleVariant(
   const COleVariant& varSrc
);
COleVariant(
   LPCVARIANT pSrc
);
COleVariant(
   LPCTSTR lpszSrc
);
COleVariant(
   LPCTSTR lpszSrc,
   VARTYPE vtSrc
);
COleVariant(
   CString& strSrc
);
COleVariant(
   BYTE nSrc
);
COleVariant(
   short nSrc,
   VARTYPE vtSrc = VT_I2
);
COleVariant(
   long lSrc,
   VARTYPE vtSrc = VT_I4
);
COleVariant(
   const COleCurrency& curSrc
);
COleVariant(
   float fltSrc
);
COleVariant(
   double dblSrc
);
COleVariant(
   const COleDateTime& timeSrc
);
COleVariant(
   const CByteArray& arrSrc
);
COleVariant(
   const CLongBinary& lbSrc
);
COleVariant(
   LPCITEMIDLIST pidl
);
```

Parameters

varSrc

An existing **COleVariant** or **VARIANT** object to be copied into the new **COleVariant** object.

pSrc

A pointer to a **VARIANT** object that will be copied into the new **COleVariant** object.

lpszSrc

A null-terminated string to be copied into the new **COleVariant** object.

vtSrc

The VARTYPE for the new **COleVariant** object.

strSrc

A CString object to be copied into the new **COleVariant** object.

nSrc, *lSrc*

A numerical value to be copied into the new **COleVariant** object.

vtSrc

The VARTYPE for the new **COleVariant** object.

curSrc

A COleCurrency object to be copied into the new **COleVariant** object.

fltSrc, *dblSrc*

A numerical value to be copied into the new **COleVariant** object.

timeSrc

A COleDateTime object to be copied into the new **COleVariant** object.

arrSrc

A CByteArray object to be copied into the new **COleVariant** object.

lbSrc

A CLongBinary object to be copied into the new **COleVariant** object.

pidl

A pointer to a ITEMIDLIST structure to be copied into the new **COleVariant** object.

Remarks

All these constructors create new **COleVariant** objects initialized to the specified value. A brief description of each of these constructors follows.

- **COleVariant( )** Creates an empty **COleVariant** object, **VT_EMPTY**.
- **COleVariant(** *varSrc* **)** Copies an existing **VARIANT** or **COleVariant** object. The variant type is retained.
- **COleVariant(** *pSrc* **)** Copies an existing **VARIANT** or **COleVariant** object. The variant type is retained.
- **COleVariant(** *lpszSrc* **)** Copies a string into the new object, **VT_BSTR** (UNICODE).
- **COleVariant(** *lpszSrc* **,** *vtSrc* **)** Copies a string into the new object. The parameter *vtSrc* must be **VT_BSTR** (UNICODE) or **VT_BSTRT** (ANSI).
- **COleVariant(** *strSrc* **)** Copies a string into the new object, **VT_BSTR** (UNICODE).
- **COleVariant(** *nSrc* **)** Copies an 8-bit integer into the new object, **VT_UI1**.
- **COleVariant(** *nSrc* **,** *vtSrc* **)** Copies a 16-bit integer (or Boolean value) into the new object. The parameter *vtSrc* must be **VT_I2** or **VT_BOOL**.
- **COleVariant(** *lSrc* **,** *vtSrc* **)** Copies a 32-bit integer (or **SCODE** value) into the new object. The parameter *vtSrc* must be **VT_I4**, **VT_ERROR**, or **VT_BOOL**.
- **COleVariant(** *curSrc* **)** Copies a **COleCurrency** value into the new object, **VT_CY**.
- **COleVariant(** *fltSrc* **)** Copies a 32-bit floating-point value into the new object, **VT_R4**.

- **COleVariant(** *dblSrc* **)** Copies a 64-bit floating-point value into the new object, **VT_R8**.
- **COleVariant(** *timeSrc* **)** Copies a **COleDateTime** value into the new object, **VT_DATE**.
- **COleVariant(** *arrSrc* **)** Copies a **CByteArray** object into the new object, **VT_EMPTY**.
- **COleVariant(** *lbSrc* **)** Copies a **CLongBinary** object into the new object, **VT_EMPTY**.

For more information, see the VARIANT and VARTYPE entries in the *Platform SDK*.

For more information on **SCODE**, see Structure of COM Error Codes in the *Platform SDK*.

See Also

Reference

COleVariant Class
Hierarchy Chart
COleVariant::operator =
CStringT Class
COleCurrency Class
COleDateTime Class

Other Resources

COleVariant Members

COleVariant::ChangeType

Converts the type of variant value in this **COleVariant** object.

```
void ChangeType(
   VARTYPE vartype,
   LPVARIANT pSrc = NULL
);
```

Parameters

vartype

The VARTYPE for this **COleVariant** object.

pSrc

A pointer to the VARIANT object to be converted. If this value is **NULL**, this **COleVariant** object is used as the source for the conversion.

Remarks

For more information, see the VARIANT, VARTYPE, and VariantChangeType entries in the *Platform SDK*.

See Also

Reference

COleVariant Class

Hierarchy Chart

COleVariant::operator =

Other Resources

COleVariant Members

COleVariant::Clear

Clears the **VARIANT**.

```
void Clear( );
```

Remarks

This sets the **VARTYPE** for this object to **VT_EMPTY**. The **COleVariant** destructor calls this function.

For more information, see the VARIANT, VARTYPE, and VariantClear entries in the *Platform SDK*.

See Also

Reference

COleVariant Class

Hierarchy Chart

Other Resources

COleVariant Members

COleVariant::Detach

Detaches the underlying VARIANT object from this **COleVariant** object.

```
VARIANT Detach( );
```

Return Type

The underlying **VARIANT** value of this **COleVariant** object.

Remarks

This function sets the VARTYPE for this **COleVariant** object to **VT_EMPTY**.

Note
After calling **Detach**, it is the caller's responsibility to call **VariantClear** on the resulting **VARIANT** structure.

For more information, see the VARIANT, VARTYPE, and VariantClear entries in the *Platform SDK*.

See Also

Reference

COleVariant Class
Hierarchy Chart
COleVariant::operator LPCVARIANT
COleVariant::operator LPVARIANT

Other Resources

COleVariant Members

COleVariant::GetByteArrayFromVariantArray

Retrieves a byte array from an existing variant array

```
void GetByteArrayFromVariantArray(
   CByteArray& bytes
);
```

Parameters

bytes

A reference to an existing CByteArray object.

See Also

Reference

COleVariant Class

Hierarchy Chart

Other Resources

COleVariant Members

COleVariant::SetString

Sets the string to a particular type.

```
void SetString(
   LPCTSTR lpszSrc,
   VARTYPE vtSrc
);
```

Parameters

lpszSrc

A null-terminated string to be copied into the new **COleVariant** object.

VtSrc

The VARTYPE for the new **COleVariant** object.

Remarks

The parameter *vtSrc* must be **VT_BSTR** (UNICODE) or **VT_BSTRT** (ANSI). **SetString** is typically used to set strings to ANSI, since the default for the COleVariant::COleVariant constructor with a string or string pointer parameter and no **VARTYPE** is UNICODE.

A DAO recordset in a non-UNICODE build expects strings to be ANSI. Thus, for DAO functions that use **COleVariant** objects, if you are not creating a UNICODE recordset, you must use the **COleVariant::COleVariant(** *lpszSrc***,** *vtSrc* **)** form of constructor with *vtSrc* set to **VT_BSTRT** (ANSI) or use **SetString** with *vtSrc* set to **VT_BSTRT** to make ANSI strings. For example, the **CDaoRecordset** functions CDaoRecordset::Seek and CDaoRecordset::SetFieldValue use **COleVariant** objects as parameters. These objects must be ANSI if the DAO recordset is not UNICODE.

See Also

Reference

COleVariant Class
Hierarchy Chart
COleVariant::COleVariant
CDaoRecordset::Seek
CDaoRecordset::SetFieldValue

Other Resources

COleVariant Members

COleVariant Operators

For information about the operators in **COleVariant**, see COleVariant Members.

COleVariant::operator =

These overloaded assignment operators copy the source value into this **COleVariant** object.

```
const COleVariant& operator =(
   const VARIANT& varSrc
);
const COleVariant& operator =(
   LPCVARIANT pSrc
);
const COleVariant& operator =(
   const COleVariant& varSrc
);
const COleVariant& operator =(
   const LPCTSTR lpszSrc
);
const COleVariant& operator =(
   const CString& strSrc
);
const COleVariant& operator =(
   BYTE nSrc
);
const COleVariant& operator =(
   short nSrc
);
const COleVariant& operator =(
   long lSrc
);
const COleVariant& operator =(
   const COleCurrency& curSrc
);
const COleVariant& operator =(
   float fltSrc
);
const COleVariant& operator =(
   double dblSrc
);
const COleVariant& operator =(
   const COleDateTime& dateSrc
);
const COleVariant& operator =(
   const CByteArray& arrSrc
);
const COleVariant& operator =(
   const CLongBinary& lbSrc
);
```

Remarks

A brief description of each operator follows:

- **operator =(** *varSrc* **)** Copies an existing **VARIANT** or **COleVariant** object into this object.
- **operator =(** *pSrc* **)** Copies the **VARIANT** object accessed by *pSrc* into this object.
- **operator =(** *lpszSrc* **)** Copies a null-terminated string into this object and sets the **VARTYPE** to **VT_BSTR**.
- **operator =(** *strSrc* **)** Copies a CString object into this object and sets the **VARTYPE** to **VT_BSTR**.
- **operator =(** *nSrc* **)** Copies an 8- or 16-bit integer value into this object. If *nSrc* is an 8-bit value, the **VARTYPE** of this is set to **VT_UI1**. If *nSrc* is a 16-bit value and the **VARTYPE** of this is **VT_BOOL**, it is kept; otherwise, it is set to **VT_I2**.
- **operator =(** *lSrc* **)** Copies a 32-bit integer value into this object. If the **VARTYPE** of this is **VT_ERROR**, it is kept; otherwise, it is set to **VT_I4**.

- **operator =(** *curSrc* **)** Copies a COleCurrency object into this object and sets the **VARTYPE** to **VT_CY**.
- **operator =(** *fltSrc* **)** Copies a 32-bit floating-point value into this object and sets the **VARTYPE** to **VT_R4**.
- **operator =(** *dblSrc* **)** Copies a 64-bit floating-point value into this object and sets the **VARTYPE** to **VT_R8**.
- **operator =(** *dateSrc* **)** Copies a COleDateTime object into this object and sets the **VARTYPE** to **VT_DATE**.
- **operator =(** *arrSrc* **)** Copies a CByteArray object into this **COleVariant** object.
- **operator =(** *lbSrc* **)** Copies a CLongBinary object into this **COleVariant** object.

For more information, see the VARIANT and VARTYPE entries in the *Platform SDK*.

See Also

Reference

COleVariant Class
Hierarchy Chart
COleVariant::COleVariant
COleCurrency Class
COleDateTime Class

Other Resources

COleVariant Members

COleVariant::operator ==

This operator compares two variant values and returns nonzero if they are equal; otherwise 0.

```
BOOL operator ==(
   const VARIANT& varSrc
) const;
BOOL operator ==(
   LPCVARIANT pSrc
) const;
```

See Also

Reference

COleVariant Class

Hierarchy Chart

COleVariant::operator =

Other Resources

COleVariant Members

COleVariant::operator LPCVARIANT

This casting operator returns a VARIANT structure whose value is copied from this **COleVariant** object.

```
operator LPCVARIANT( ) const;
```

Remarks

For more information, see the VARIANT entry in the *Platform SDK*.

See Also

Reference

COleVariant Class
Hierarchy Chart
COleVariant::operator LPVARIANT

Other Resources

COleVariant Members

COleVariant::operator LPVARIANT

Call this casting operator to access the underlying VARIANT structure for this **COleVariant** object.

```
operator LPVARIANT( );
```

Remarks

Caution
Changing the value in the **VARIANT** structure accessed by the pointer returned by this function will change the value of this **COleVariant** object.

For more information, see the VARIANT entry in the *Platform SDK*.

See Also

Reference

COleVariant Class

Hierarchy Chart

COleVariant::operator LPCVARIANT

Other Resources

COleVariant Members

COleVariant::operator <<, >>

Outputs a **COleVariant** value to **CArchive** or **CdumpContext** and inputs a **COleVariant** object from **CArchive**.

```
friend CDumpContext& AFXAPI operator <<(
   CDumpContext& dc,
   OleVariant varSrc
);
friend CArchive& AFXAPI operator <<(
   CArchive& ar,
   COleVariant varSrc
);
friend CArchive& AFXAPI operator >>(
   CArchive& ar,
   COleVariant& varSrc
);
```

Remarks

The **COleVariant** insertion (**<<**) operator supports diagnostic dumping and storing to an archive. The extraction (**>>**) operator supports loading from an archive.

See Also

Reference

COleVariant Class

Hierarchy Chart

CDumpContext Class

CArchive Class

Other Resources

COleVariant Members

CPageSetupDialog Class

Encapsulates the services provided by the Windows common OLE Page Setup dialog box with additional support for setting and modifying print margins.

```
class CPageSetupDialog : public CCommonDialog
```

Remarks

This class is designed to take the place of the Print Setup dialog box.

To use a **CPageSetupDialog** object, first create the object using the **CPageSetupDialog** constructor. Once the dialog box has been constructed, you can set or modify any values in the **m_psd** data member to initialize the values of the dialog box's controls. The m_psd structure is of type **PAGESETUPDLG**.

After initializing the dialog box controls, call the **DoModal** member function to display the dialog box and allow the user to select print options. **DoModal** returns whether the user selected the OK (**IDOK**) or Cancel (**IDCANCEL**) button.

If **DoModal** returns **IDOK**, you can use several of **CPageSetupDialog**'s member functions, or access the **m_psd** data member, to retrieve information input by the user.

Note
After the common OLE Page Setup dialog box is dismissed, any changes made by the user will not be saved by the framework. It is up to the application itself to save any values from this dialog box to a permanent location, such as member of the application's document or application class.

Requirements

Header: afxdlgs.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

WORDPAD Sample: The Windows Application

Reference

CCommonDialog Class

Hierarchy Chart

Other Resources

CPageSetupDialog Members

CPageSetupDialog Members

Base Class Members

COBJECT Members

CCmdTarget Members

CWnd Members

CDialog Members

CCommonDialog Members

Attributes

CreatePrinterDC	Creates a device context for printing.
GetDeviceName	Returns the device name of the printer.
GetDevMode	Returns the current **DEVMODE** of the printer.
GetDriverName	Returns the driver used by the printer.
GetMargins	Returns the current margin settings of the printer.
GetPaperSize	Returns the paper size of the printer.
GetPortName	Returns the output port name.

Construction

CPageSetupDialog	Constructs a **CPageSetupDialog** object.

Data Members

m_psd	A structure used to customize a **CPageSetupDialog** object.

Operations

DoModal	Displays the dialog box and allows the user make a selection.

Overridables

OnDrawPage	Called by the framework to render a screen image of a printed page.
PreDrawPage	Called by the framework before rendering a screen image of a printed page.

See Also

Reference

CPageSetupDialog Class
Hierarchy Chart

CPageSetupDialog Member Functions

For information about the member functions in **CPageSetupDialog**, see CPageSetupDialog Members.

CPageSetupDialog::CPageSetupDialog

Call this function to construct a **CPageSetupDialog** object.

```
CPageSetupDialog(
   DWORD dwFlags = PSD_MARGINS | PSD_INWININIINTLMEASURE,
   CWnd* pParentWnd = NULL
);
```

Parameters

dwFlags

One or more flags you can use to customize the settings of the dialog box. The values can be combined using the bitwise-OR operator. These values have the following meanings:

- **PSD_DEFAULTMINMARGINS** Sets the minimum allowable widths for the page margins to be the same as the printer's minimums. This flag is ignored if the **PSD_MARGINS** and **PSD_MINMARGINS** flags are also specified.
- **PSD_INWININIINTLMEASURE** Not implemented.
- **PSD_MINMARGINS** Causes the system to use the values specified in the **rtMinMargin** member as the minimum allowable widths for the left, top, right, and bottom margins. The system prevents the user from entering a width that is less than the specified minimum. If **PSD_MINMARGINS** is not specified, the system sets the minimum allowable widths to those allowed by the printer.
- **PSD_MARGINS** Activates the margin control area.
- **PSD_INTHOUSANDTHSOFINCHES** Causes the units of the dialog box to be measured in 1/1000 of an inch.
- **PSD_INHUNDREDTHSOFMILLIMETERS** Causes the units of the dialog box to be measured in 1/100 of a millimeter.
- **PSD_DISABLEMARGINS** Disables the margin dialog box controls.
- **PSD_DISABLEPRINTER** Disables the Printer button.
- **PSD_NOWARNING** Prevents the warning message from being displayed when there is no default printer.
- **PSD_DISABLEORIENTATION** Disables the page orientation dialog control.
- **PSD_RETURNDEFAULT** Causes **CPageSetupDialog** to return DEVMODE and DEVNAMES structures that are initialized for the system default printer without displaying a dialog box. It is assumed that both **hDevNames** and **hDevMode** are **NULL**; otherwise, the function returns an error. If the system default printer is supported by an old printer driver (earlier than Windows version 3.0), only **hDevNames** is returned; **hDevMode** is **NULL**.
- **PSD_DISABLEPAPER** Disables the paper selection control.
- **PSD_SHOWHELP** Causes the dialog box to show the Help button. The **hwndOwner** member must not be **NULL** if this flag is specified.
- **PSD_ENABLEPAGESETUPHOOK** Enables the hook function specified in **lpfnSetupHook**.
- **PSD_ENABLEPAGESETUPTEMPLATE** Causes the operating system to create the dialog box by using the dialog template box identified by **hInstance** and **lpSetupTemplateName**.
- **PSD_ENABLEPAGESETUPTEMPLATEHANDLE** Indicates that **hInstance** identifies a data block that contains a preloaded dialog box template. The system ignores **lpSetupTemplateName** if this flag is specified.
- **PSD_ENABLEPAGEPAINTHOOK** Enables the hook function specified in **lpfnPagePaintHook**.
- **PSD_DISABLEPAGEPAINTING** Disables the draw area of the dialog box.

pParentWnd

Pointer to the dialog box's parent or owner.

Remarks

Use the DoModal function to display the dialog box.

Example

```
void CMyRichEditView::OnPageSetupDlg()
{
   CPageSetupDialog psd( PSD_INTHOUSANDTHSOFINCHES | PSD_MARGINS |
          PSD_ENABLEPAGEPAINTHOOK, this );

   // Initialize margins
   psd.m_psd.rtMargin.top = 1000;
   psd.m_psd.rtMargin.left = 1250;
   psd.m_psd.rtMargin.right = 1250;
   psd.m_psd.rtMargin.bottom = 1000;
   psd.m_psd.lpfnPagePaintHook = (LPPAGEPAINTHOOK)PaintHook;

   if( IDOK == psd.DoModal() )
   {
      // Propagate changes to the app
      AfxGetApp()->SelectPrinter(psd.m_psd.hDevNames, psd.m_psd.hDevMode);
   }
   else
      TRACE("CommDlgExtendedError returned error %d from
         CPageSetupDialog::DoModal().\n",
         (int)CommDlgExtendedError() );
}
```

See Also

Reference

CPageSetupDialog Class
Hierarchy Chart
CPrintDialog Class
CPageSetupDialog Class

Other Resources

CPageSetupDialog Members

CPageSetupDialog::CreatePrinterDC

Creates a printer device context from the DEVMODE and DEVNAMES structures.

```
HDC CreatePrinterDC( );
```

Return Value

Handle to the newly created printer device context (DC).

See Also

Reference

CPageSetupDialog Class
Hierarchy Chart
CPageSetupDialog::GetDevMode
CPageSetupDialog::GetDeviceName
CPageSetupDialog::GetDriverName

Other Resources

CPageSetupDialog Members

CPageSetupDialog::DoModal

Call this function to display the Windows common OLE Page Setup dialog box and allow the user to select various print setup options such as the printing margins, size and orientation of the paper, and destination printer.

```
virtual INT_PTR DoModal( );
```

Return Value

IDOK or **IDCANCEL**. If **IDCANCEL** is returned, call the Windows CommDlgExtendedError function to determine whether an error occurred.

IDOK and **IDCANCEL** are constants that indicate whether the user selected the OK or Cancel button.

Remarks

In addition, the user can access the printer setup options such as network location and properties specific to the selected printer.

If you want to initialize the various Page Setup dialog options by setting members of the **m_psd** structure, you should do so before calling **DoModal**, and after the dialog object is constructed. After calling **DoModal**, call other member functions to retrieve the settings or information input by the user into the dialog box.

If you want to propagate the current settings entered by the user, make a call to CWinApp::SelectPrinter. This function takes the information from the **CPageSetupDialog** object and initializes and selects a new printer DC with the proper attributes.

```
AfxGetApp()->SelectPrinter(dlg.m_psd.hDevNames, dlg.m_psd.hDevMode);
```

Example

See the example for CPageSetupDialog::CPageSetupDialog.

See Also

Reference

CPageSetupDialog Class
Hierarchy Chart
CPageSetupDialog::m_psd

Other Resources

CPageSetupDialog Members

CPageSetupDialog::GetDeviceName

Call this function after **DoModal** to retrieve the name of the currently selected printer.

```
CString GetDeviceName( ) const;
```

Return Value

The device name used by the **CPageSetupDialog** object.

See Also

Reference

CPageSetupDialog Class

Hierarchy Chart

Other Resources

CPageSetupDialog Members

CPageSetupDialog::GetDevMode

Call this function after calling **DoModal** to retrieve information about the printer device context of the **CPageSetupDialog** object.

```
LPDEVMODE GetDevMode( ) const;
```

Return Value

The DEVMODE data structure, which contains information about the device initialization and environment of a print driver. You must unlock the memory taken by this structure with the Windows GlobalUnlock function, which is described in the *Platform SDK*.

See Also

Reference

CPageSetupDialog Class

Hierarchy Chart

Other Resources

CPageSetupDialog Members

CPageSetupDialog::GetDriverName

Call this function after calling DoModal to retrieve the name of the system-defined printer device driver.

```
CString GetDriverName( ) const;
```

Return Value

A **CString** specifying the system-defined driver name.

Remarks

Use a pointer to the **CString** object returned by **GetDriverName** as the value of *lpszDriverName* in a call to CDC::CreateDC.

See Also

Reference

CPageSetupDialog Class
Hierarchy Chart
CPageSetupDialog::GetDeviceName
CPageSetupDialog::GetDevMode
CPageSetupDialog::GetPortName

Other Resources

CPageSetupDialog Members

CPageSetupDialog::GetMargins

Call this function after a call to **DoModal** to retrieve the margins of the printer device driver.

```
void GetMargins(
   LPRECT lpRectMargins,
   LPRECT lpRectMinMargins
) const;
```

Parameters

lpRectMargins

Pointer to a RECT structure or CRect object that describes (in 1/1000 inches or 1/100 mm) the print margins for the currently selected printer. Pass **NULL** for this parameter, if you are not interested in this rectangle.

lpRectMinMargins

Pointer to a **RECT** structure or **CRect** object that describes (in 1/1000 inches or 1/100 mm) the minimum print margins for the currently selected printer. Pass **NULL** for this parameter, if you are not interested in this rectangle.

See Also

Reference

CPageSetupDialog Class

Hierarchy Chart

Other Resources

CPageSetupDialog Members

CPageSetupDialog::GetPaperSize

Call this function to retrieve the size of the paper selected for printing.

```
CSize GetPaperSize( ) const;
```

Return Value

A CSize object containing the size of the paper (in 1/1000 inches or 1/100 mm) selected for printing.

See Also

Reference

CPageSetupDialog Class

Hierarchy Chart

Other Resources

CPageSetupDialog Members

CPageSetupDialog::GetPortName

Call this function after calling **DoModal** to retrieve the name of the currently selected printer port.

```
CString GetPortName( ) const;
```

Return Value

The name of the currently selected printer port.

See Also

Reference

CPageSetupDialog Class
Hierarchy Chart
CPageSetupDialog::GetDeviceName
CPageSetupDialog::GetDriverName

Other Resources

CPageSetupDialog Members

CPageSetupDialog::OnDrawPage

Called by the framework to draw a screen image of a printed page.

```
virtual UINT OnDrawPage(
   CDC* pDC,
   UINT nMessage,
   LPRECT lpRect
);
```

Parameters

pDC

Pointer to the printer device context.

nMessage

Specifies a message, indicating the area of the page currently being drawn. Can be one of the following:

- **WM_PSD_FULLPAGERECT** The entire page area.
- **WM_PSD_MINMARGINRECT** Current minimum margins.
- **WM_PSD_MARGINRECT** Current margins.
- **WM_PSD_GREEKTEXTRECT** Contents of the page.
- **WM_PSD_ENVSTAMPRECT** Area reserved for a postage stamp representation.
- **WM_PSD_YAFULLPAGERECT** Area for a return address representation. This area extends to the edges of the sample page area.

lpRect

Pointer to a CRect or RECT object containing the coordinates of the drawing area.

Return Value

Nonzero value if handled; otherwise 0.

Remarks

This image is then displayed as part of the common OLE Page Setup dialog box. The default implementation draws an image of a page of text.

Override this function to customize the drawing of a specific area of the image, or the entire image. You can do this by using a **switch** statement with **case** statements checking the value of *nMessage*. For example, to customize the rendering of the contents of the page image, you could use the following example code:

```
switch( nType )
{
    case WM_PSD_GREEKTEXTRECT:
      DrawMyImage(pDC, lpRect);    //draws my special graphic
      return 1;
    default:
      return ::Draw(CDC* pDC, UINT nDrawType, LPRECT lpRect);
};
```

Note that you do not need to handle every case of *nMessage*. You can choose to handle one component of the image, several components of the image, or the whole area.

See Also

Reference

CPageSetupDialog Class
Hierarchy Chart
CPageSetupDialog::PreDrawPage

Other Resources

CPageSetupDialog Members

CPageSetupDialog::PreDrawPage

Called by the framework before drawing the screen image of a printed page.

```
virtual UINT PreDrawPage(
   WORD wPaper,
   WORD wFlags,
   LPPAGESETUPDLG pPSD
);
```

Parameters

wPaper

Specifies a value that indicates the paper size. This value can be one of the **DMPAPER_** values listed in the description of the DEVMODE structure.

wFlags

Indicates the orientation of the paper or envelope, and whether the printer is a dot-matrix or HPPCL (Hewlett Packard Printer Control Language) device. This parameter can have one of the following values:

- 0x001 Paper in landscape mode (dot matrix)
- 0x003 Paper in landscape mode (HPPCL)
- 0x005 Paper in portrait mode (dot matrix)
- 0x007 Paper in portrait mode (HPPCL)
- 0x00b Envelope in landscape mode (HPPCL)
- 0x00d Envelope in portrait mode (dot matrix)
- 0x019 Envelope in landscape mode (dot matrix)
- 0x01f Envelope in portrait mode (dot matrix)

pPSD

Pointer to a **PAGESETUPDLG** structure. For more information on PAGESETUPDLG, see the *Platform SDK*.

Return Value

Nonzero value if handled; otherwise 0.

Remarks

Override this function to customize the drawing of the image. If you override this function and return **TRUE**, you must draw the entire image. If you override this function and return **FALSE**, the entire default image is drawn by the framework.

See Also

Reference

CPageSetupDialog Class
Hierarchy Chart
CPageSetupDialog::OnDrawPage

Other Resources

CPageSetupDialog Members

CPageSetupDialog Data Members

For information about the data members in **CPageSetupDialog**, see CPageSetupDialog Members.

CPageSetupDialog::m_psd

A structure of type **PAGESETUPDLG**, whose members store the characteristics of the dialog object.

```
PAGESETUPDLG m_psd;
```

Remarks

After constructing a **CPageSetupDialog** object, you can use **m_psd** to set various aspects of the dialog box before calling the **DoModal** member function.

If you modify the **m_psd** data member directly, you will override any default behavior.

For more information on the PAGESETUPDLG structure, see the *Platform SDK*.

See the example for CPageSetupDialog::CPageSetupDialog.

See Also

Reference

CPageSetupDialog Class

Hierarchy Chart

Other Resources

CPageSetupDialog Members

CPaintDC Class

A device-context class derived from CDC.

```
class CPaintDC : public CDC
```

Remarks

It performs a CWnd::BeginPaint at construction time and CWnd::EndPaint at destruction time.

A **CPaintDC** object can only be used when responding to a WM_PAINT message, usually in your **OnPaint** message-handler member function.

For more information on using **CPaintDC**, see Device Contexts.

Requirements

Header: afxwin.h

See Also

Tasks

MDI Sample: Demonstrates MDI Without Using Doc/View Architecture

Reference

CDC Class

Hierarchy Chart

Other Resources

CPaintDC Members

CPaintDC Members

Base Class Members

CObject Members

CDC Members

Data Members

m_hWnd	The **HWND** to which this **CPaintDC** object is attached.
m_ps	Contains the PAINTSTRUCT used to paint the client area.

Construction

CPaintDC	Constructs a **CPaintDC** connected to the specified CWnd.

See Also

Reference

CPaintDC Class

Hierarchy Chart

CPaintDC Member Functions

For information about the member functions in **CPaintDC**, see CPaintDC Members.

CPaintDC::CPaintDC

Constructs a **CPaintDC** object, prepares the application window for painting, and stores the PAINTSTRUCT structure in the m_ps member variable.

```
explicit CPaintDC(
   CWnd* pWnd
);
```

Parameters

pWnd

Points to the **CWnd** object to which the **CPaintDC** object belongs.

Remarks

An exception (of type **CResourceException**) is thrown if the Windows GetDC call fails. A device context may not be available if Windows has already allocated all of its available device contexts. Your application competes for the five common display contexts available at any given time under Windows.

Example

```
// Get a dc for a CWnd pointer.
CPaintDC dc(pWnd);

// Get a dc for a HWND.
CPaintDC dc2(CWnd::FromHandle(hWnd));
```

See Also

Reference

CPaintDC Class

Hierarchy Chart

Other Resources

CPaintDC Members

CPaintDC Data Members

For information about the data members in **CPaintDC**, see CPaintDC Members.

CPaintDC::m_hWnd

The **HWND** to which this **CPaintDC** object is attached.

```
HWND m_hWnd;
```

Remarks

m_hWnd is a protected variable of type **HWND**.

Example

```
// Get a dc for a CWnd object pointer.
CPaintDC dc(pWnd);

// Send my private massage.
::SendMessage(pWnd->m_hWnd, WM_MYMESSAGE, (LPARAM) &dc.m_ps, 0);
```

See Also

Reference

CPaintDC Class

Hierarchy Chart

Other Resources

CPaintDC Members

CPaintDC::m_ps

m_ps is a public member variable of type PAINTSTRUCT.

```
PAINTSTRUCT m_ps;
```

Remarks

It is the **PAINTSTRUCT** that is passed to and filled out by CWnd::BeginPaint.

The **PAINTSTRUCT** contains information that the application uses to paint the client area of the window associated with a **CPaintDC** object.

Note that you can access the device-context handle through the **PAINTSTRUCT**. However, you can access the handle more directly through the **m_hDC** member variable that **CPaintDC** inherits from **CDC**.

Example

See the example for CPaintDC::m_hWnd.

See Also

Reference

CPaintDC Class

Hierarchy Chart

Other Resources

CPaintDC Members

CPalette Class

Encapsulates a Windows color palette.

```
class CPalette : public CGdiObject
```

Remarks

A palette provides an interface between an application and a color output device (such as a display device). The interface allows the application to take full advantage of the color capabilities of the output device without severely interfering with the colors displayed by other applications. Windows uses the application's logical palette (a list of needed colors) and the system palette (which defines available colors) to determine the colors used.

A **CPalette** object provides member functions for manipulating the palette referred to by the object. Construct a **CPalette** object and use its member functions to create the actual palette, a graphics device interface (GDI) object, and to manipulate its entries and other properties.

For more information on using **CPalette**, see Graphic Objects.

Requirements

Header: afxwin.h

See Also

Tasks

DIBLOOK Sample: Illustrates the Use of DIBs and Color Palettes

Reference

CGdiObject Class

Hierarchy Chart

CPalette::GetPaletteEntries

CPalette::SetPaletteEntries

Other Resources

CPalette Members

CPalette Members

Base Class Members

CObject Members

CGdiObject Members

Construction

CPalette	Constructs a **CPalette** object with no attached Windows palette. You must initialize the **CPalette** object with one of the initialization member functions before it can be used.

Initialization

CreateHalftonePalette	Creates a halftone palette for the device context and attaches it to the **CPalette** object.
CreatePalette	Creates a Windows color palette and attaches it to the **CPalette** object.

Operations

AnimatePalette	Replaces entries in the logical palette identified by the **CPalette** object. The application does not have to update its client area, because Windows maps the new entries into the system palette immediately.
FromHandle	Returns a pointer to a **CPalette** object when given a handle to a Windows palette object.
GetNearestPaletteIndex	Returns the index of the entry in the logical palette that most closely matches a color value.
ResizePalette	Changes the size of the logical palette specified by the **CPalette** object to the specified number of entries.

Attributes

GetEntryCount	Retrieves the number of palette entries in a logical palette.
GetPaletteEntries	Retrieves a range of palette entries in a logical palette.
operator HPALETTE	Returns the **HPALETTE** attached to the **CPalette**.
SetPaletteEntries	Sets RGB color values and flags in a range of entries in a logical palette.

See Also

Reference

CPalette Class

Hierarchy Chart

CPalette Member Functions

For information about the member functions in **CPalette**, see CPalette Members.

CPalette::AnimatePalette

Replaces entries in the logical palette attached to the **CPalette** object.

```
void AnimatePalette(
   UINT nStartIndex,
   UINT nNumEntries,
   LPPALETTEENTRY lpPaletteColors
);
```

Parameters

nStartIndex

Specifies the first entry in the palette to be animated.

nNumEntries

Specifies the number of entries in the palette to be animated.

lpPaletteColors

Points to the first member of an array of PALETTEENTRY structures to replace the palette entries identified by *nStartIndex* and *nNumEntries*.

Remarks

When an application calls **AnimatePalette**, it does not have to update its client area, because Windows maps the new entries into the system palette immediately.

The **AnimatePalette** function will only change entries with the **PC_RESERVED** flag set in the corresponding **palPaletteEntry** member of the LOGPALETTE structure that is attached to the **CPalette** object. See **LOGPALETTE** in the *Platform SDK* for more information about this structure.

See Also

Reference

CPalette Class
Hierarchy Chart
CPalette::CreatePalette
AnimatePalette

Other Resources

CPalette Members

CPalette::CPalette

Constructs a **CPalette** object.

```
CPalette( );
```

Remarks

The object has no attached palette until you call **CreatePalette** to attach one.

See Also

Reference

CPalette Class
Hierarchy Chart
CPalette::CreatePalette

Other Resources

CPalette Members

CPalette::CreateHalftonePalette

Creates a halftone palette for the device context.

```
BOOL CreateHalftonePalette(
   CDC* pDC
);
```

Parameters

pDC

Identifies the device context.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

An application should create a halftone palette when the stretching mode of a device context is set to **HALFTONE**. The logical halftone palette returned by the CreateHalftonePalette member function should then be selected and realized into the device context before the CDC::StretchBlt or StretchDIBits function is called.

See the *Platform SDK* for more information about **CreateHalftonePalette** and **StretchDIBits**.

See Also

Reference

CPalette Class
Hierarchy Chart
CDC::RealizePalette
CDC::SelectPalette
CDC::SetStretchBltMode
CreateHalftonePalette
StretchDIBits

Other Resources

CPalette Members

CPalette::CreatePalette

Initializes a **CPalette** object by creating a Windows logical color palette and attaching it to the **CPalette** object.

```
BOOL CreatePalette(
   LPLOGPALETTE lpLogPalette
);
```

Parameters

lpLogPalette

Points to a LOGPALETTE structure that contains information about the colors in the logical palette.

Return Value

Nonzero if successful; otherwise 0.

Remarks

See the *Platform SDK* for more information about the **LOGPALETTE** structure.

See Also

Reference

CPalette Class
Hierarchy Chart
CreatePalette
LOGPALETTE

Other Resources

CPalette Members

CPalette::FromHandle

Returns a pointer to a **CPalette** object when given a handle to a Windows palette object.

```
static CPalette* PASCAL FromHandle(
   HPALETTE hPalette
);
```

Parameters

hPalette

A handle to a Windows GDI color palette.

Return Value

A pointer to a **CPalette** object if successful; otherwise **NULL**.

Remarks

If a **CPalette** object is not already attached to the Windows palette, a temporary **CPalette** object is created and attached. This temporary **CPalette** object is valid only until the next time the application has idle time in its event loop, at which time all temporary graphic objects are deleted. In other words, the temporary object is valid only during the processing of one window message.

See Also

Reference

CPalette Class

Hierarchy Chart

Other Resources

CPalette Members

CPalette::GetEntryCount

Call this member function to retrieve the number of entries in a given logical palette.

```
int GetEntryCount( );
```

Return Value

Number of entries in a logical palette.

See Also

Reference

CPalette Class
Hierarchy Chart
CPalette::GetPaletteEntries
CPalette::SetPaletteEntries

Other Resources

CPalette Members

CPalette::GetNearestPaletteIndex

Returns the index of the entry in the logical palette that most closely matches the specified color value.

```
UINT GetNearestPaletteIndex(
   COLORREF crColor
) const;
```

Parameters

crColor

Specifies the color to be matched.

Return Value

The index of an entry in a logical palette. The entry contains the color that most nearly matches the specified color.

See Also

Reference

CPalette Class

Hierarchy Chart

GetNearestPaletteIndex

Other Resources

CPalette Members

CPalette::GetPaletteEntries

Retrieves a range of palette entries in a logical palette.

```
UINT GetPaletteEntries(
   UINT nStartIndex,
   UINT nNumEntries,
   LPPALETTEENTRY lpPaletteColors
) const;
```

Parameters

nStartIndex

Specifies the first entry in the logical palette to be retrieved.

nNumEntries

Specifies the number of entries in the logical palette to be retrieved.

lpPaletteColors

Points to an array of PALETTEENTRY data structures to receive the palette entries. The array must contain at least as many data structures as specified by *nNumEntries*.

Return Value

The number of entries retrieved from the logical palette; 0 if the function failed.

See Also

Reference

CPalette Class
Hierarchy Chart
GetPaletteEntries
CPalette::SetPaletteEntries

Other Resources

CPalette Members

CPalette::operator HPALETTE

Use this operator to get the attached Windows GDI handle of the **CPalette** object.

```
operator HPALETTE( ) const;
```

Return Value

If successful, a handle to the Windows GDI object represented by the **CPalette** object; otherwise **NULL**.

Remarks

This operator is a casting operator, which supports direct use of an **HPALETTE** object.

For more information about using graphic objects, see the article Graphic Objects in the *Platform SDK*.

See Also

Reference

CPalette Class

Hierarchy Chart

Other Resources

CPalette Members

CPalette::ResizePalette

Changes the size of the logical palette attached to the **CPalette** object to the number of entries specified by *nNumEntries*.

```
BOOL ResizePalette(
   UINT nNumEntries
);
```

Parameters

nNumEntries

Specifies the number of entries in the palette after it has been resized.

Return Value

Nonzero if the palette was successfully resized; otherwise 0.

Remarks

If an application calls **ResizePalette** to reduce the size of the palette, the entries remaining in the resized palette are unchanged. If the application calls **ResizePalette** to enlarge the palette, the additional palette entries are set to black (the red, green, and blue values are all 0), and the flags for all additional entries are set to 0.

For more information on the Windows API **ResizePalette**, see ResizePalette in the *Platform SDK*.

See Also

Reference

CPalette Class

Hierarchy Chart

ResizePalette

Other Resources

CPalette Members

CPalette::SetPaletteEntries

Sets RGB color values and flags in a range of entries in a logical palette.

```
UINT SetPaletteEntries(
   UINT nStartIndex,
   UINT nNumEntries,
   LPPALETTEENTRY lpPaletteColors
);
```

Parameters

nStartIndex

Specifies the first entry in the logical palette to be set.

nNumEntries

Specifies the number of entries in the logical palette to be set.

lpPaletteColors

Points to an array of PALETTEENTRY data structures to receive the palette entries. The array must contain at least as many data structures as specified by *nNumEntries*.

Return Value

The number of entries set in the logical palette; 0 if the function failed.

Remarks

If the logical palette is selected into a device context when the application calls **SetPaletteEntries**, the changes will not take effect until the application calls CDC::RealizePalette.

For more information on the Windows structure **PALETTEENTRY**, see PALETTEENTRY in the *Platform SDK*.

See Also

Reference

CPalette Class
Hierarchy Chart
CDC::RealizePalette
CPalette::GetPaletteEntries
SetPaletteEntries

Other Resources

CPalette Members

CPen Class

Encapsulates a Windows graphics device interface (GDI) pen.

```
class CPen : public CGdiObject
```

Remarks

For more information on using **CPen**, see Graphic Objects.

Requirements

Header: afxwin.h

See Also

Reference

CGdiObject Class

Hierarchy Chart

CBrush Class

Other Resources

CPen Members

CPen Members

Base Class Members

CObject Members

CGdiObject Members

Construction

CPen	Constructs a **CPen** object.

Initialization

CreatePen	Creates a logical cosmetic or geometric pen with the specified style, width, and brush attributes, and attaches it to the **CPen** object.
CreatePenIndirect	Creates a pen with the style, width, and color given in a LOGPEN structure, and attaches it to the **CPen** object.

Operations

FromHandle	Returns a pointer to a **CPen** object when given a Windows **HPEN**.

Attributes

GetExtLogPen	Gets an EXTLOGPEN underlying structure.
GetLogPen	Gets a LOGPEN underlying structure.
operator HPEN	Returns the Windows handle attached to the **CPen** object.

See Also

Reference

CPen Class

Hierarchy Chart

CPen Member Functions

For information about the member functions in **CPen**, see CPen Members.

CPen::CPen

Constructs a **CPen** object.

```
CPen( );
CPen(
   int nPenStyle,
   int nWidth,
   COLORREF crColor
);
CPen(
   int nPenStyle,
   int nWidth,
   const LOGBRUSH* pLogBrush,
   int nStyleCount = 0,
   const DWORD* lpStyle = NULL
);
```

Parameters

nPenStyle

Specifies the pen style. This parameter in the first version of the constructor can be one of the following values:

- **PS_SOLID** Creates a solid pen.
- **PS_DASH** Creates a dashed pen. Valid only when the pen width is 1 or less, in device units.
- **PS_DOT** Creates a dotted pen. Valid only when the pen width is 1 or less, in device units.
- **PS_DASHDOT** Creates a pen with alternating dashes and dots. Valid only when the pen width is 1 or less, in device units.
- **PS_DASHDOTDOT** Creates a pen with alternating dashes and double dots. Valid only when the pen width is 1 or less, in device units.
- **PS_NULL** Creates a null pen.
- **PS_INSIDEFRAME** Creates a pen that draws a line inside the frame of closed shapes produced by the Windows GDI output functions that specify a bounding rectangle (for example, the **Ellipse**, **Rectangle**, **RoundRect**, **Pie**, and **Chord** member functions). When this style is used with Windows GDI output functions that do not specify a bounding rectangle (for example, the **LineTo** member function), the drawing area of the pen is not limited by a frame.

The second version of the **CPen** constructor specifies a combination of type, style, end cap, and join attributes. The values from each category should be combined by using the bitwise OR operator (|). The pen type can be one of the following values:

- **PS_GEOMETRIC** Creates a geometric pen.
- **PS_COSMETIC** Creates a cosmetic pen.

  The second version of the **CPen** constructor adds the following pen styles for *nPenStyle*:

- **PS_ALTERNATE** Creates a pen that sets every other pixel. (This style is applicable only for cosmetic pens.)
- **PS_USERSTYLE** Creates a pen that uses a styling array supplied by the user.

  The end cap can be one of the following values:

- **PS_ENDCAP_ROUND** End caps are round.
- **PS_ENDCAP_SQUARE** End caps are square.
- **PS_ENDCAP_FLAT** End caps are flat.

  The join can be one of the following values:

- **PS_JOIN_BEVEL** Joins are beveled.
- **PS_JOIN_MITER** Joins are mitered when they are within the current limit set by the SetMiterLimit function. If the join exceeds this limit, it is beveled.
- **PS_JOIN_ROUND** Joins are round.

nWidth

Specifies the width of the pen.

- For the first version of the constructor, if this value is 0, the width in device units is always 1 pixel, regardless of the mapping mode.
- For the second version of the constructor, if *nPenStyle* is **PS_GEOMETRIC**, the width is given in logical units. If *nPenStyle* is **PS_COSMETIC**, the width must be set to 1.

crColor

Contains an RGB color for the pen.

pLogBrush

Points to a **LOGBRUSH** structure. If *nPenStyle* is **PS_COSMETIC**, the *lbColor* member of the **LOGBRUSH** structure specifies the color of the pen and the *lbStyle* member of the **LOGBRUSH** structure must be set to **BS_SOLID**. If *nPenStyle* is **PS_GEOMETRIC**, all members must be used to specify the brush attributes of the pen.

nStyleCount

Specifies the length, in doubleword units, of the *lpStyle* array. This value must be zero if *nPenStyle* is not **PS_USERSTYLE**.

lpStyle

Points to an array of doubleword values. The first value specifies the length of the first dash in a user-defined style, the second value specifies the length of the first space, and so on. This pointer must be **NULL** if *nPenStyle* is not **PS_USERSTYLE**.

Remarks

If you use the constructor with no arguments, you must initialize the resulting **CPen** object with the **CreatePen**, **CreatePenIndirect**, or **CreateStockObject** member functions.

If you use the constructor that takes arguments, then no further initialization is necessary. The constructor with arguments can throw an exception if errors are encountered, while the constructor with no arguments will always succeed.

Example

```
// Create a solid red pen of width 2.
CPen myPen1(PS_SOLID, 2, RGB(255,0,0));

// Create a geometric pen.
LOGBRUSH logBrush;
logBrush.lbStyle = BS_SOLID;
logBrush.lbColor = RGB(0,255,0);
CPen myPen2(PS_DOT|PS_GEOMETRIC|PS_ENDCAP_ROUND, 2, &logBrush);
```

See Also

Reference

CPen Class
Hierarchy Chart
CPen::CreatePen
CPen::CreatePenIndirect
CGdiObject::CreateStockObject

Other Resources

CPen Members

CPen::CreatePen

Creates a logical cosmetic or geometric pen with the specified style, width, and brush attributes, and attaches it to the **CPen** object.

```
BOOL CreatePen(
   int nPenStyle,
   int nWidth,
   COLORREF crColor
);
BOOL CreatePen(
   int nPenStyle,
   int nWidth,
   const LOGBRUSH* pLogBrush,
   int nStyleCount = 0,
   const DWORD* lpStyle = NULL
);
```

Parameters

nPenStyle

Specifies the style for the pen. For a list of possible values, see the *nPenStyle* parameter in the CPen constructor.

nWidth

Specifies the width of the pen.

- For the first version of **CreatePen**, if this value is 0, the width in device units is always 1 pixel, regardless of the mapping mode.
- For the second version of **CreatePen**, if *nPenStyle* is **PS_GEOMETRIC**, the width is given in logical units. If *nPenStyle* is **PS_COSMETIC**, the width must be set to 1.

crColor

Contains an RGB color for the pen.

pLogBrush

Points to a LOGBRUSH structure. If *nPenStyle* is **PS_COSMETIC**, the **lbColor** member of the **LOGBRUSH** structure specifies the color of the pen and the **lbStyle** member of the **LOGBRUSH** structure must be set to **BS_SOLID**. If **nPenStyle** is **PS_GEOMETRIC**, all members must be used to specify the brush attributes of the pen.

nStyleCount

Specifies the length, in doubleword units, of the *lpStyle* array. This value must be zero if *nPenStyle* is not **PS_USERSTYLE**.

lpStyle

Points to an array of doubleword values. The first value specifies the length of the first dash in a user-defined style, the second value specifies the length of the first space, and so on. This pointer must be **NULL** if *nPenStyle* is not **PS_USERSTYLE**.

Return Value

Nonzero, or the handle of a logical pen, if successful; otherwise 0.

Remarks

The first version of **CreatePen** initializes a pen with the specified style, width, and color. The pen can be subsequently selected as the current pen for any device context.

Pens that have a width greater than 1 pixel should always have either the **PS_NULL**, **PS_SOLID**, or **PS_INSIDEFRAME** style.

If a pen has the **PS_INSIDEFRAME** style and a color that does not match a color in the logical color table, the pen is drawn with a dithered color. The **PS_SOLID** pen style cannot be used to create a pen with a dithered color. The style **PS_INSIDEFRAME** is identical to **PS_SOLID** if the pen width is less than or equal to 1.

The second version of **CreatePen** initializes a logical cosmetic or geometric pen that has the specified style, width, and brush attributes. The width of a cosmetic pen is always 1; the width of a geometric pen is always specified in world units. After an application creates a logical pen, it can select that pen into a device context by calling the CDC::SelectObject function. After a pen is selected into a device context, it can be used to draw lines and curves.

- If *nPenStyle* is **PS_COSMETIC** and **PS_USERSTYLE**, the entries in the *lpStyle* array specify lengths of dashes and spaces in style units. A style unit is defined by the device in which the pen is used to draw a line.
- If *nPenStyle* is **PS_GEOMETRIC** and **PS_USERSTYLE**, the entries in the *lpStyle* array specify lengths of dashes and spaces in logical units.
- If *nPenStyle* is **PS_ALTERNATE**, the style unit is ignored and every other pixel is set.

When an application no longer requires a given pen, it should call the CGdiObject::DeleteObject member function or destroy the **CPen** object so the resource is no longer in use. An application should not delete a pen when the pen is selected in a device context.

Example

```
CPen myPen1, myPen2;

// Create a solid red pen of width 2.
myPen1.CreatePen(PS_SOLID, 2, RGB(255,0,0));

// Create a geometric pen.
LOGBRUSH logBrush;
logBrush.lbStyle = BS_SOLID;
logBrush.lbColor = RGB(0,255,0);
myPen2.CreatePen(PS_DOT|PS_GEOMETRIC|PS_ENDCAP_ROUND, 2, &logBrush);
```

See Also

Reference

CPen Class
Hierarchy Chart
CPen::CreatePenIndirect
CPen::CPen
CGdiObject::DeleteObject
LOGBRUSH

Other Resources

CPen Members

CPen::CreatePenIndirect

Initializes a pen that has the style, width, and color given in the structure pointed to by *lpLogPen*.

```
BOOL CreatePenIndirect(
   LPLOGPEN lpLogPen
);
```

Parameters

lpLogPen

Points to the Windows LOGPEN structure that contains information about the pen.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

Pens that have a width greater than 1 pixel should always have either the **PS_NULL**, **PS_SOLID**, or **PS_INSIDEFRAME** style.

If a pen has the **PS_INSIDEFRAME** style and a color that does not match a color in the logical color table, the pen is drawn with a dithered color. The **PS_INSIDEFRAME** style is identical to **PS_SOLID** if the pen width is less than or equal to 1.

Example

```
LOGPEN logpen;
CPen   cMyPen;

// Get the LOGPEN of an existing pen.
pExistingPen->GetLogPen( &logpen );

// Change the color to red and the width to 2.
logpen.lopnWidth.x = 2;
logpen.lopnColor = RGB(255, 0, 0);

// Create my pen using the new settings.
cMyPen.CreatePenIndirect( &logpen );
```

See Also

Reference

CPen Class
Hierarchy Chart
CPen::CreatePen
CPen::CPen

Other Resources

CPen Members

CPen::FromHandle

Returns a pointer to a **CPen** object given a handle to a Windows GDI pen object.

```
static CPen* PASCAL FromHandle(
   HPEN hPen
);
```

Parameters

hPen

HPEN handle to Windows GDI pen.

Return Value

A pointer to a **CPen** object if successful; otherwise **NULL**.

Remarks

If a **CPen** object is not attached to the handle, a temporary **CPen** object is created and attached. This temporary **CPen** object is valid only until the next time the application has idle time in its event loop, at which time all temporary graphic objects are deleted. In other words, the temporary object is only valid during the processing of one window message.

Example

```
// Convert an HPEN to a CPen*.
// NOTE: hPen is a valid pen handle.
CPen* pPen = CPen::FromHandle(hPen);
```

See Also

Reference

CPen Class

Hierarchy Chart

Other Resources

CPen Members

CPen::GetExtLogPen

Gets an **EXTLOGPEN** underlying structure.

```
int GetExtLogPen(
   EXTLOGPEN* pLogPen
);
```

Parameters

pLogPen

Points to an EXTLOGPEN structure that contains information about the pen.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The **EXTLOGPEN** structure defines the style, width, and brush attributes of a pen. For example, call **GetExtLogPen** to match the particular style of a pen.

See the following topics in the *Platform SDK* for information about pen attributes:

- GetObject
- EXTLOGPEN
- LOGPEN
- ExtCreatePen

Example

The following code example demonstrates calling **GetExtLogPen** to retrieve a pen's attributes, and then create a new, cosmetic pen with the same color.

```
EXTLOGPEN extlogpen;
penExisting.GetExtLogPen( &extlogpen );
CPen penOther;
LOGBRUSH LogBrush={ extlogpen.elpBrushStyle, extlogpen.elpColor, extlogpen.elpHatch };
penOther.CreatePen( PS_COSMETIC, 1, &LogBrush );
```

See Also

Reference

CPen Class
Hierarchy Chart
CPen::GetLogPen
CPen::CreatePen

Other Resources

CPen Members

CPen::GetLogPen

Gets a **LOGPEN** underlying structure.

```
int GetLogPen(
   LOGPEN* pLogPen
);
```

Parameters

pLogPen

Points to a LOGPEN structure to contain information about the pen.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The **LOGPEN** structure defines the style, color, and pattern of a pen.

For example, call **GetLogPen** to match the particular style of pen.

See the following topics in the *Platform SDK* for information about pen attributes:

- GetObject
- LOGPEN

Example

The following code example demonstrates calling **GetLogPen** to retrieve a pen character, and then create a new, solid pen with the same color.

```
LOGPEN logpen;
penExisting.GetLogPen( &logpen );
CPen penOther( PS_SOLID, 0, logpen.lopnColor);
```

See Also

Reference

CPen Class

Hierarchy Chart

CPen::GetExtLogPen

Other Resources

CPen Members

CPen::operator HPEN

Gets the attached Windows GDI handle of the **CPen** object.

```
operator HPEN( ) const;
```

Return Value

If successful, a handle to the Windows GDI object represented by the **CPen** object; otherwise **NULL**.

Remarks

This operator is a casting operator, which supports direct use of an **HPEN** object.

For more information about using graphic objects, see the article Graphic Objects in *Platform SDK*.

Example

```
// Create a solid red pen of width 2.
CPen myPen(PS_SOLID, 2, RGB(255,0,0));

// Get the handle of the pen object.
HPEN hPen = (HPEN) myPen;
```

See Also

Reference

CPen Class

Hierarchy Chart

Other Resources

CPen Members

CPictureHolder Class

Implements a Picture property, which allows the user to display a picture in your control.

```
class CPictureHolder
```

Remarks

CPictureHolder does not have a base class.

With the stock Picture property, the developer can specify a bitmap, icon, or metafile for display.

For information on creating custom picture properties, see the article
MFC ActiveX Controls: Using Pictures in an ActiveX Control.

Requirements

Header: afxctl.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart
CFontHolder Class

Other Resources

CPictureHolder Members

CPictureHolder Members

Data Members

m_pPict	A pointer to a picture object.

Construction

CPictureHolder	Constructs a **CPictureHolder** object.

Operations

CreateEmpty	Creates an empty **CPictureHolder** object.
CreateFromBitmap	Creates a **CPictureHolder** object from a bitmap.
CreateFromIcon	Creates a **CPictureHolder** object from an icon.
CreateFromMetafile	Creates a **CPictureHolder** object from a metafile.
GetDisplayString	Retrieves the string displayed in a control container's property browser.
GetPictureDispatch	Returns the **CPictureHolder** object's **IDispatch** interface.
GetType	Tells whether the **CPictureHolder** object is a bitmap, a metafile, or an icon.
Render	Renders the picture.
SetPictureDispatch	Sets the **CPictureHolder** object's **IDispatch** interface.

See Also

Reference

CPictureHolder Class

Hierarchy Chart

CPictureHolder Member Functions

For information about the member functions in **CPictureHolder**, see CPictureHolder Members.

CPictureHolder::CPictureHolder

Constructs a **CPictureHolder** object.

```
CPictureHolder( );
```

See Also

Reference

CPictureHolder Class

Hierarchy Chart

CPictureHolder::CreateEmpty

Other Resources

CPictureHolder Members

CPictureHolder::CreateEmpty

Creates an empty **CPictureHolder** object and connects it to an **IPicture** interface.

```
BOOL CreateEmpty( );
```

Return Value

Nonzero if the object is successfully created; otherwise 0.

See Also

Reference

CPictureHolder Class
Hierarchy Chart
CPictureHolder::CreateFromBitmap
CPictureHolder::CreateFromIcon
CPictureHolder::CreateFromMetafile

Other Resources

CPictureHolder Members

CPictureHolder::CreateFromBitmap

Uses a bitmap to initialize the picture object in a **CPictureHolder**.

```
BOOL CreateFromBitmap(
   UINT idResource
);
BOOL CreateFromBitmap(
   CBitmap* pBitmap,
   CPalette* pPal = NULL,
   BOOL bTransferOwnership = TRUE
);
BOOL CreateFromBitmap(
   HBITMAP hbm,
   HPALETTE hpal = NULL,
   BOOL bTransferOwnership = FALSE
);
```

Parameters

idResource

Resource ID of a bitmap resource.

pBitmap

Pointer to a CBitmap object.

pPal

Pointer to a CPalette object.

bTransferOwnership

Indicates whether the picture object will take ownership of the bitmap and palette objects.

hbm

Handle to the bitmap from which the **CPictureHolder** object is created.

hpal

Handle to the palette used for rendering the bitmap.

Return Value

Nonzero if the object is successfully created; otherwise 0.

Remarks

If *bTransferOwnership* is **TRUE**, the caller should not use the bitmap or palette object in any way after this call returns. If *bTransferOwnership* is **FALSE**, the caller is responsible for ensuring that the bitmap and palette objects remain valid for the lifetime of the picture object.

See Also

Reference

CPictureHolder Class
Hierarchy Chart
CPictureHolder::CreateEmpty
CPictureHolder::CreateFromIcon
CPictureHolder::CreateFromMetafile

Other Resources

CPictureHolder Members

CPictureHolder::CreateFromIcon

Uses an icon to initialize the picture object in a **CPictureHolder**.

```
BOOL CreateFromIcon(
   UINT idResource
);
BOOL CreateFromIcon(
   HICON hIcon,
   BOOL bTransferOwnership = FALSE
);
```

Parameters

idResource

Resource ID of a bitmap resource.

hIcon

Handle to the icon from which the **CPictureHolder** object is created.

bTransferOwnership

Indicates whether the picture object will take ownership of the icon object.

Return Value

Nonzero if the object is successfully created; otherwise 0.

Remarks

If *bTransferOwnership* is **TRUE**, the caller should not use the icon object in any way after this call returns. If *bTransferOwnership* is **FALSE**, the caller is responsible for ensuring that the icon object remains valid for the lifetime of the picture object.

See Also

Reference

CPictureHolder Class
Hierarchy Chart
CPictureHolder::CreateEmpty
CPictureHolder::CreateFromBitmap
CPictureHolder::CreateFromMetafile

Other Resources

CPictureHolder Members

CPictureHolder::CreateFromMetafile

Uses a metafile to initialize the picture object in a **CPictureHolder**.

```
BOOL CreateFromMetafile(
   HMETAFILE hmf,
   int xExt,
   int yExt,
   BOOL bTransferOwnership = FALSE
);
```

Parameters

hmf

Handle to the metafile used to create the **CPictureHolder** object.

xExt

X extent of the picture.

yExt

Y extent of the picture.

bTransferOwnership

Indicates whether the picture object will take ownership of the metafile object.

Return Value

Nonzero if the object is successfully created; otherwise 0.

Remarks

If *bTransferOwnership* is **TRUE**, the caller should not use the metafile object in any way after this call returns. If *bTransferOwnership* is **FALSE**, the caller is responsible for ensuring that the metafile object remains valid for the lifetime of the picture object.

See Also

Reference

CPictureHolder Class
Hierarchy Chart
CPictureHolder::CreateEmpty
CPictureHolder::CreateFromBitmap
CPictureHolder::CreateFromIcon

Other Resources

CPictureHolder Members

CPictureHolder::GetDisplayString

Retrieves the string that is displayed in a container's property browser.

```
BOOL GetDisplayString(
   CString& strValue
);
```

Parameters

strValue

Reference to the CString that is to hold the display string.

Return Value

Nonzero if the string is successfully retrieved; otherwise 0.

See Also

Reference

CPictureHolder Class

Hierarchy Chart

Other Resources

CPictureHolder Members

CPictureHolder::GetPictureDispatch

This function returns a pointer to the **CPictureHolder** object's **IPictureDisp** interface.

```
LPPICTUREDISP GetPictureDispatch( );
```

Return Value

A pointer to the **CPictureHolder** object's **IPictureDisp** interface.

Remarks

The caller must call **Release** on this pointer when finished with it.

See Also

Reference

CPictureHolder Class
Hierarchy Chart
CPictureHolder::SetPictureDispatch

Other Resources

CPictureHolder Members

CPictureHolder::GetType

Indicates whether the picture is a bitmap, metafile, or icon.

```
short GetType( );
```

Return Value

A value indicating the type of the picture. Possible values and their meanings are as follows:

Value	Meaning
PICTYPE_UNINITIALIZED	**CPictureHolder** object is unititialized.
PICTYPE_NONE	**CPictureHolder** object is empty.
PICTYPE_BITMAP	Picture is a bitmap.
PICTYPE_METAFILE	Picture is a metafile.
PICTYPE_ICON	Picture is an icon.

See Also

Reference

CPictureHolder Class

Hierarchy Chart

Other Resources

CPictureHolder Members

CPictureHolder::Render

Renders the picture in the rectangle referenced by *rcRender*.

```
void Render(
   CDC* pDC,
   const CRect& rcRender,
   const CRect& rcWBounds
);
```

Parameters

pDC

Pointer to the display context in which the picture is to be rendered.

rcRender

Rectangle in which the picture is to be rendered.

rcWBounds

A rectangle representing the bounding rectangle of the object rendering the picture. For a control, this rectangle is the *rcBounds* parameter passed to an override of COleControl::OnDraw.

See Also

Reference

CPictureHolder Class

Hierarchy Chart

Other Resources

CPictureHolder Members

CPictureHolder::SetPictureDispatch

Connects the **CPictureHolder** object to a **IPictureDisp** interface.

```
void SetPictureDispatch(
   LPPICTUREDISP pDisp
);
```

Parameters

pDisp

Pointer to the new **IPictureDisp** interface.

See Also

Reference

CPictureHolder Class

Hierarchy Chart

CPictureHolder::GetPictureDispatch

Other Resources

CPictureHolder Members

CPictureHolder Data Members

For information about the data members in **CPictureHolder**, see CPictureHolder Members.

CPictureHolder::m_pPict

A pointer to the **CPictureHolder** object's **IPicture** interface.

```
LPPICTURE m_pPict;
```

See Also

Reference

CPictureHolder Class

Hierarchy Chart

Other Resources

CPictureHolder Members

CPrintDialog Class

Encapsulates the services provided by the Windows common dialog box for printing.

```
class CPrintDialog : public CCommonDialog
```

Remarks

Common print dialog boxes provide an easy way to implement Print and Print Setup dialog boxes in a manner consistent with Windows standards.

Note
The **CPrintDialogEx** class encapsulates the services provided by the Windows 2000 Print property sheet. For more information see the CPrintDialogEx overview.

CPrintDialog's functionality is superceded by that of CPageSetupDialog, which is designed to provide you with a common dialog box for both print setup and page setup.

You can rely on the framework to handle many aspects of the printing process for your application. In this case, the framework automatically displays the Windows common dialog box for printing. You can also have the framework handle printing for your application but override the common Print dialog box with your own print dialog box. For more information about using the framework to handle printing tasks, see the article Printing.

If you want your application to handle printing without the framework's involvement, you can use the **CPrintDialog** class "as is" with the constructor provided, or you can derive your own dialog class from **CPrintDialog** and write a constructor to suit your needs. In either case, these dialog boxes will behave like standard MFC dialog boxes because they are derived from class **CCommonDialog**.

To use a **CPrintDialog** object, first create the object using the **CPrintDialog** constructor. Once the dialog box has been constructed, you can set or modify any values in the m_pd structure to initialize the values of the dialog box's controls. The **m_pd** structure is of type PRINTDLG. For more information on this structure, see the *Platform SDK*.

If you do not supply your own handles in **m_pd** for the **hDevMode** and **hDevNames** members, be sure to call the Windows function **GlobalFree** for these handles when you are done with the dialog box. When using the framework's Print Setup implementation provided by **CWinApp::OnFilePrintSetup**, you do not have to free these handles. The handles are maintained by **CWinApp** and are freed in **CWinApp**'s destructor. It is only necessary to free these handles when using **CPrintDialog** stand-alone.

After initializing the dialog box controls, call the **DoModal** member function to display the dialog box and allow the user to select print options. **DoModal** returns whether the user selected the OK (**IDOK**) or Cancel (**IDCANCEL**) button.

If **DoModal** returns **IDOK**, you can use one of **CPrintDialog**'s member functions to retrieve the information input by the user.

The **CPrintDialog::GetDefaults** member function is useful for retrieving the current printer defaults without displaying a dialog box. This member function requires no user interaction.

You can use the Windows **CommDlgExtendedError** function to determine whether an error occurred during initialization of the dialog box and to learn more about the error. For more information on this function, see the *Platform SDK*.

CPrintDialog relies on the COMMDLG.DLL file that ships with Windows versions 3.1 and later.

To customize the dialog box, derive a class from **CPrintDialog**, provide a custom dialog template, and add a message map to process the notification messages from the extended controls. Any unprocessed messages should be passed on to the base class. Customizing the hook function is not required.

To process the same message differently depending on whether the dialog box is Print or Print Setup, you must derive a class for each dialog box. You must also override the Windows **AttachOnSetup** function, which handles the creation of a new dialog box when the Print Setup button is selected within a Print dialog box.

For more information on using **CPrintDialog**, see Common Dialog Classes.

Requirements

Header: afxdlgs.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

DIBLOOK Sample: Illustrates the Use of DIBs and Color Palettes

Reference

CCommonDialog Class

Hierarchy Chart

CPrintInfo Structure

Other Resources

CPrintDialog Members

CPrintDialog Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CDialog Members

CCommonDialog Members

Data Members

m_pd	A structure used to customize a **CPrintDialog** object.

Construction

CPrintDialog	Constructs a **CPrintDialog** object.

Operations

CreatePrinterDC	Creates a printer device context without displaying the Print dialog box.
DoModal	Displays the dialog box and allows the user to make a selection.
GetCopies	Retrieves the number of copies requested.
GetDefaults	Retrieves device defaults without displaying a dialog box.
GetDeviceName	Retrieves the name of the currently selected printer device.
GetDevMode	Retrieves the **DEVMODE** structure.
GetDriverName	Retrieves the name of the currently selected printer driver.
GetFromPage	Retrieves the starting page of the print range.
GetPortName	Retrieves the name of the currently selected printer port.
GetPrinterDC	Retrieves a handle to the printer device context.
GetToPage	Retrieves the ending page of the print range.
PrintAll	Determines whether to print all pages of the document.
PrintCollate	Determines whether collated copies are requested.
PrintRange	Determines whether to print only a specified range of pages.
PrintSelection	Determines whether to print only the currently selected items.

See Also

Reference

CPrintDialog Class

Hierarchy Chart

CPrintDialog Member Functions

For information about the member functions in **CPrintDialog**, see CPrintDialog Members.

CPrintDialog::CPrintDialog

Constructs either a Windows Print or Print Setup dialog object.

```
CPrintDialog(
   BOOL bPrintSetupOnly,
   DWORD dwFlags = PD_ALLPAGES | PD_USEDEVMODECOPIES | PD_NOPAGENUMS | PD_HIDEPRINTTOFILE |
PD_NOSELECTION,
   CWnd* pParentWnd = NULL
);
```

Parameters

bPrintSetupOnly

Specifies whether the standard Windows Print dialog box or Print Setup dialog box is displayed. Set this parameter to **TRUE** to display the standard Windows Print Setup dialog box. Set it to **FALSE** to display the Windows Print dialog box. If *bPrintSetupOnly* is **FALSE**, a Print Setup option button is still displayed in the Print dialog box.

dwFlags

One or more flags you can use to customize the settings of the dialog box, combined using the bitwise OR operator. For example, the **PD_ALLPAGES** flag sets the default print range to all pages of the document. See the PRINTDLG structure in the *Platform SDK* for more information on these flags.

pParentWnd

A pointer to the dialog box's parent or owner window.

Remarks

This member function only constructs the object. Use the **DoModal** member function to display the dialog box.

Note that when you call the constructor with *bPrintSetupOnly* set to **FALSE**, the **PD_RETURNDC** flag is automatically used. After calling **DoModal**, **GetDefaults**, or **GetPrinterDC**, a printer DC will be returned in `m_pd.hDC`. This DC must be freed with a call to DeleteDC by the caller of **CPrintDialog**.

Example

```
// Display the Windows Print dialog box with "All" radio button
// initially selected. All other radio buttons are disabled.
CPrintDialog dlg(FALSE);

// Display the Windows Print dialog box with Collate check box checked.
CPrintDialog dlg(FALSE, PD_ALLPAGES | PD_COLLATE | PD_NOPAGENUMS | PD_HIDEPRINTTOFILE);

// Display the Windows Print dialog box with "Selection" radio
// button initially selected. "All" radio button is enabled
// but "Pages" radio button is disabled.
CPrintDialog dlg(FALSE, PD_SELECTION | PD_USEDEVMODECOPIES);
```

See Also

Reference

CPrintDialog Class
Hierarchy Chart
CPrintDialog::DoModal
PrintDlg

Other Resources

CPrintDialog Members

CPrintDialog::CreatePrinterDC

Creates a printer device context (DC) from the DEVMODE and DEVNAMES structures.

```
HDC CreatePrinterDC( );
```

Return Value

Handle to the newly created printer device context.

Remarks

This DC is assumed to be the current printer DC, and any other previously obtained printer DCs must be deleted by the user. This function can be called, and the resulting DC used, without ever displaying the Print dialog box.

Example

```
// Display the Windows Print dialog box with "All" radio button
// initially selected. All other radio buttons are disabled.
CPrintDialog dlg(FALSE);
if (dlg.DoModal() == IDOK)
{
   // Create a printer device context (DC) based on the information
   // selected from the Print dialog.
   HDC hdc = dlg.CreatePrinterDC();
   ASSERT(hdc);
}
```

See Also

Reference

CPrintDialog Class
Hierarchy Chart
CPrintDialog::GetDevMode

Other Resources

CPrintDialog Members

CPrintDialog::DoModal

Displays the Windows common print dialog box and allows the user to select various printing options such as the number of copies, page range, and whether copies should be collated.

```
virtual INT_PTR DoModal( );
```

Return Value

IDOK or **IDCANCEL**. If **IDCANCEL** is returned, call the Windows CommDlgExtendedError function to determine whether an error occurred.

IDOK and **IDCANCEL** are constants that indicate whether the user selected the OK or Cancel button.

Remarks

If you want to initialize the various print dialog options by setting members of the **m_pd** structure, you should do this before calling **DoModal**, but after the dialog object is constructed.

After calling **DoModal**, you can call other member functions to retrieve the settings or information input by the user into the dialog box.

Note that when you call the constructor with *bPrintSetupOnly* set to **FALSE**, the **PD_RETURNDC** flag is automatically used. After calling **DoModal**, **GetDefaults**, or **GetPrinterDC**, a printer DC will be returned in `m_pd.hDC`. This DC must be freed with a call to DeleteDC by the caller of **CPrintDialog**.

Example

See the example for CPrintDialog::CreatePrinterDC.

See Also

Reference

CPrintDialog Class
Hierarchy Chart
CPrintDialog::CPrintDialog
CDialog::DoModal

Other Resources

CPrintDialog Members

CPrintDialog::GetCopies

Retrieves the number of copies requested.

```
int GetCopies( ) const;
```

Return Value

The number of copies requested.

Remarks

Call this function after calling **DoModal** to retrieve the number of copies requested.

Example

See the example for CPrintDialog::PrintCollate.

See Also

Reference

CPrintDialog Class
Hierarchy Chart
CPrintDialog::PrintCollate

Other Resources

CPrintDialog Members

CPrintDialog::GetDefaults

Retrieves the device defaults of the default printer without displaying a dialog box.

```
BOOL GetDefaults( );
```

Return Value

Nonzero if the function was successful; otherwise 0.

Remarks

The retrieved values are placed in the **m_pd** structure.

In some cases, a call to this function will call the constructor for **CPrintDialog** with *bPrintSetupOnly* set to **FALSE**. In these cases, a printer DC and **hDevNames** and **hDevMode** (two handles located in the **m_pd** data member) are automatically allocated.

If the constructor for **CPrintDialog** was called with *bPrintSetupOnly* set to **FALSE**, this function will not only return **hDevNames** and **hDevMode** (located in **m_pd.hDevNames** and **m_pd.hDevMode**) to the caller, but will also return a printer DC in **m_pd.hDC**. It is the responsibility of the caller to delete the printer DC and call the Windows GlobalFree function on the handles when you are finished with the **CPrintDialog** object.

Example

This code fragment gets the default printer's device context and reports to the user the resolution of the printer in dots per inch. (This attribute of the printer's capabilities is often referred to as DPI.)

```
CPrintDialog dlg(FALSE);

if (!dlg.GetDefaults())
   AfxMessageBox(_T("You have no default printer!"));
else
{
   // attach to the DC we were given
   CDC dc;
   dc.Attach(dlg.m_pd.hDC);

   // ask for the measurements
   int nHorz = dc.GetDeviceCaps(LOGPIXELSX);
   int nVert = dc.GetDeviceCaps(LOGPIXELSY);

   // almost always the same in both directions, but sometimes not!
   CString str;
   if (nHorz == nVert)
      str.Format(_T("Your printer supports %d pixels per inch"), nHorz);
   else
      str.Format(_T("Your printer supports %d pixels per inch ")
         _T("horizontal resolution, and %d pixels per inch vertical ")
         _T("resolution"), nHorz, nVert);

   // tell the user
   AfxMessageBox(str);

   // Note: no need to call Detach() because we want the CDC destructor
   // to call FreeDC() on the DC we borrowed from the common dialog
}
```

See Also

Reference

CPrintDialog Class

Hierarchy Chart

CPrintDialog::m_pd

CPrintDialog::GetDeviceName
CPrintDialog::GetDriverName
CPrintDialog::GetPortName

Other Resources

CPrintDialog Members

CPrintDialog::GetDeviceName

Retrieves the name of the currently selected printer device.

```
CString GetDeviceName( ) const;
```

Return Value

The name of the currently selected printer.

Remarks

Call this function after calling DoModal to retrieve the name of the currently selected printer, or after calling GetDefaults to retrieve the current device defaults of the default printer. Use a pointer to the **CString** object returned by **GetDeviceName** as the value of *lpszDeviceName* in a call to CDC::CreateDC.

Example

This code fragment shows the user's default printer name and the port it is connected to, along with the spooler name the printer uses. The code might show a message box that says, "Your default printer is HP LaserJet IIIP on \\server\share using winspool.", for example.

```
CPrintDialog dlg(FALSE);

if (!dlg.GetDefaults())
   AfxMessageBox(_T("You have no default printer!"));
else
{
   CString strDescription;

   strDescription.Format(_T("Your default printer is %s on %s using %s."),
      (LPCTSTR) dlg.GetDeviceName(),
      (LPCTSTR) dlg.GetPortName(),
      (LPCTSTR) dlg.GetDriverName());

   AfxMessageBox(strDescription);
}
```

See Also

Reference

CPrintDialog Class
Hierarchy Chart
CPrintDialog::GetDriverName
CPrintDialog::GetDevMode
CPrintDialog::GetPortName

Other Resources

CPrintDialog Members

CPrintDialog::GetDevMode

Retrieves the **DEVMODE** structure.

```
LPDEVMODE GetDevMode( ) const;
```

Return Value

The DEVMODE data structure, which contains information about the device initialization and environment of a print driver. You must unlock the memory taken by this structure with the Windows GlobalUnlock function, which is described in the *Platform SDK*.

Remarks

Call this function after calling DoModal or GetDefaults to retrieve information about the printing device.

Example

See the example for CPrintDialog::PrintCollate.

See Also

Reference

CPrintDialog Class
Hierarchy Chart
CDC::GetDeviceCaps

Other Resources

CPrintDialog Members

CPrintDialog::GetDriverName

Retrieves the name of the currently selected printer driver.

```
CString GetDriverName( ) const;
```

Return Value

A **CString** specifying the system-defined driver name.

Remarks

Call this function after calling DoModal or GetDefaults to retrieve the name of the system-defined printer device driver. Use a pointer to the **CString** object returned by **GetDriverName** as the value of *lpszDriverName* in a call to CDC::CreateDC.

Example

See the example for CPrintDialog::GetDeviceName.

See Also

Reference

CPrintDialog Class
Hierarchy Chart
CPrintDialog::GetDeviceName
CPrintDialog::GetDevMode
CPrintDialog::GetPortName

Other Resources

CPrintDialog Members

CPrintDialog::GetFromPage

Retrieves the starting page of the print range.

```
int GetFromPage( ) const;
```

Return Value

The starting page number in the range of pages to be printed.

Remarks

Call this function after calling **DoModal** to retrieve the starting page number in the range of pages to be printed.

Example

See the example for CPrintDialog::m_pd.

See Also

Reference

CPrintDialog Class
Hierarchy Chart
CPrintDialog::GetToPage
CPrintDialog::PrintRange

Other Resources

CPrintDialog Members

CPrintDialog::GetPortName

Retrieves the name of the currently selected printer port.

```
CString GetPortName( ) const;
```

Return Value

The name of the currently selected printer port.

Remarks

Call this function after calling DoModal or GetDefaults to retrieve the name of the currently selected printer port.

Example

See the example for CPrintDialog::GetDeviceName.

See Also

Reference

CPrintDialog Class
Hierarchy Chart
CPrintDialog::GetDriverName
CPrintDialog::GetDeviceName

Other Resources

CPrintDialog Members

CPrintDialog::GetPrinterDC

Retrieves a handle to the printer device context.

```
HDC GetPrinterDC( ) const;
```

Return Value

A handle to the printer device context if successful; otherwise **NULL**.

Remarks

If the *bPrintSetupOnly* parameter of the **CPrintDialog** constructor was **FALSE** (indicating that the Print dialog box is displayed), then **GetPrinterDC** returns a handle to the printer device context. You must call the Windows DeleteDC function to delete the device context when you are done using it.

Example

```
CPrintDialog dlg(FALSE);
if (dlg.DoModal() == IDOK)
{
   // Get a handle to the printer device context (DC).
   HDC hdc = dlg.GetPrinterDC();
   ASSERT(hdc);

   // Do something with the HDC...

   // Clean up.
   CDC::FromHandle(hdc)->DeleteDC();
}
```

See Also

Reference

CPrintDialog Class

Hierarchy Chart

Other Resources

CPrintDialog Members

CPrintDialog::GetToPage

Retrieves the ending page of the print range.

```
int GetToPage( ) const;
```

Return Value

The ending page number in the range of pages to be printed.

Remarks

Call this function after calling **DoModal** to retrieve the ending page number in the range of pages to be printed.

Example

See the example for CPrintDialog::m_pd.

See Also

Reference

CPrintDialog Class
Hierarchy Chart
CPrintDialog::GetFromPage
CPrintDialog::PrintRange

Other Resources

CPrintDialog Members

CPrintDialog::PrintAll

Determines whether to print all pages of the document.

```
BOOL PrintAll( ) const;
```

Return Value

Nonzero if all pages in the document are to be printed; otherwise 0.

Remarks

Call this function after calling **DoModal** to determine whether to print all pages in the document.

Example

See the example for CPrintDialog::m_pd.

See Also

Reference

CPrintDialog Class
Hierarchy Chart
CPrintDialog::PrintRange
CPrintDialog::PrintSelection

Other Resources

CPrintDialog Members

CPrintDialog::PrintCollate

Determines whether collated copies are requested.

```
BOOL PrintCollate( ) const;
```

Return Value

Nonzero if the user selects the collate check box in the dialog box; otherwise 0.

Remarks

Call this function after calling **DoModal** to determine whether the printer should collate all printed copies of the document.

Example

```
// Display the Windows Print dialog box with Collate check box checked.
CPrintDialog dlg(FALSE, PD_ALLPAGES | PD_COLLATE | PD_NOPAGENUMS | PD_HIDEPRINTTOFILE);
if (dlg.DoModal() == IDOK)
{
   // If the collate check box is selected, then GetCopies() will return
   // the number of copies printed. Otherwise, GetCopies() always
   // returns 1. Then, the number of copies printed can be found from the
   // DEVMODE structure of the printing device.
   if (dlg.PrintCollate())
   {
      int num = dlg.GetCopies();
      TRACE("Number of copies printed = %d\n", num);
   }
   else
   {
      LPDEVMODE devmode = dlg.GetDevMode();
      TRACE("Number of copies printed = %d\n", devmode->dmCopies);
   }
}
```

See Also

Reference

CPrintDialog Class

Hierarchy Chart

CPrintDialog::GetCopies

Other Resources

CPrintDialog Members

CPrintDialog::PrintRange

Determines whether to print only a specified range of pages.

```
BOOL PrintRange( ) const;
```

Return Value

Nonzero if only a range of pages in the document are to be printed; otherwise 0.

Remarks

Call this function after calling **DoModal** to determine whether to print only a range of pages in the document.

Example

See the example for CPrintDialog::m_pd.

See Also

Reference

CPrintDialog Class
Hierarchy Chart
CPrintDialog::PrintAll
CPrintDialog::PrintSelection
CPrintDialog::GetFromPage
CPrintDialog::GetToPage

Other Resources

CPrintDialog Members

CPrintDialog::PrintSelection

Determines whether to print only the currently selected items.

```
BOOL PrintSelection( ) const;
```

Return Value

Nonzero if only the selected items are to be printed; otherwise 0.

Remarks

Call this function after calling **DoModal** to determine whether to print only the currently selected items.

Example

See the example for CPrintDialog::m_pd.

See Also

Reference

CPrintDialog Class
Hierarchy Chart
CPrintDialog::PrintRange
CPrintDialog::PrintAll

Other Resources

CPrintDialog Members

CPrintDialog Data Members

For information about the data members in **CPrintDialog**, see CPrintDialog Members.

CPrintDialog::m_pd

A structure whose members store the characteristics of the dialog object.

```
PRINTDLG& m_pd;
```

Remarks

After constructing a **CPrintDialog** object, you can use **m_pd** to set various aspects of the dialog box before calling the DoModal member function. For more information on the **m_pd** structure, see PRINTDLG in the *Platform SDK*.

If you modify the **m_pd** data member directly, you will override any default behavior.

Example

```
// Display the Windows Print dialog box with "Pages" radio button
// initially selected. "All" and "Pages" radio buttons are
// enabled as well.
CPrintDialog dlg(FALSE, PD_PAGENUMS | PD_USEDEVMODECOPIES);
dlg.m_pd.nMinPage = dlg.m_pd.nFromPage =1;
dlg.m_pd.nMaxPage = dlg.m_pd.nToPage = 10;
if (dlg.DoModal() == IDOK)
{
   // Determine the starting and ending page numbers for the range
   // of pages to be printed.

   int from_page, to_page;
   if (dlg.PrintAll())              // print all pages in the document
   {
      from_page = dlg.m_pd.nMinPage;
      to_page = dlg.m_pd.nMaxPage;
   }
   else if (dlg.PrintRange())       // print only a range of pages
   {                                // in the document
      from_page = dlg.GetFromPage();
      to_page = dlg.GetToPage();
   }
   else if (dlg.PrintSelection())   // print only the currently selected
                                    // items
   {
      from_page = to_page = -1;     // -1 to denote unknown yet
   }

   TRACE("Print from %d to %d\n", from_page, to_page);
}
```

See Also

Reference

CPrintDialog Class

Hierarchy Chart

Other Resources

CPrintDialog Members

CPrintDialogEx Class

Encapsulates the services provided by the Windows 2000 Print property sheet.

```
class CPrintDialogEx : public CCommonDialog
```

Remarks

You can rely on the framework to handle many aspects of the printing process for your application. For more information about using the framework to handle printing tasks, see the article Printing.

If you want your application to handle printing without the framework's involvement, you can use the **CPrintDialogEx** class "as is" with the constructor provided, or you can derive your own dialog class from **CPrintDialogEx** and write a constructor to suit your needs. In either case, these dialog boxes will behave like standard MFC dialog boxes because they are derived from class **CCommonDialog**.

To use a **CPrintDialogEx** object, first create the object using the **CPrintDialogEx** constructor. Once the dialog box has been constructed, you can set or modify any values in the m_pdex structure to initialize the values of the dialog box's controls. The **m_pdex** structure is of type PRINTDLGEX. For more information on this structure, see the *Platform SDK*.

If you do not supply your own handles in **m_pdex** for the **hDevMode** and **hDevNames** members, be sure to call the Windows function **GlobalFree** for these handles when you are done with the dialog box.

After initializing the dialog box controls, call the **DoModal** member function to display the dialog box and allow the user to select print options. When **DoModal** returns, you can determine whether the user selected the OK, Apply, or Cancel button.

If the user pressed OK, you can use **CPrintDialogEx**'s member functions to retrieve the information input by the user.

The **CPrintDialogEx::GetDefaults** member function is useful for retrieving the current printer defaults without displaying a dialog box. This member function requires no user interaction.

You can use the Windows **CommDlgExtendedError** function to determine whether an error occurred during initialization of the dialog box and to learn more about the error. For more information on this function, see the *Platform SDK*.

For more information on using **CPrintDialogEx**, see Common Dialog Classes.

Requirements

Header: afxdlgs.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CCommonDialog Class
Hierarchy Chart
CPrintInfo Structure

Other Resources

CPrintDialog Members

CPrintDialogEx Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CDialog Members

CCommonDialog Members

Data Members

m_pdex	A structure used to customize a **CPrintDialogEx** object.

Construction

CPrintDialogEx	Constructs a **CPrintDialogEx** object.

Operations

CreatePrinterDC	Creates a printer device context without displaying the Print dialog box.
DoModal	Displays the dialog box and allows the user to make selections.
GetCopies	Retrieves the number of copies requested.
GetDefaults	Retrieves device defaults without displaying a dialog box.
GetDeviceName	Retrieves the name of the currently selected printer device.
GetDevMode	Retrieves the **DEVMODE** structure.
GetDriverName	Retrieves the name of the system-defined printer device driver.
GetPortName	Retrieves the name of the currently selected printer port.
GetPrinterDC	Retrieves a handle to the printer device context.
PrintAll	Determines whether to print all pages of the document.
PrintCollate	Determines whether collated copies are requested.
PrintCurrentPage	Determines whether to print the current page of the document.
PrintRange	Determines whether to print only a specified range of pages.
PrintSelection	Determines whether to print only the currently selected items.

See Also

Reference

CPrintDialog Class

Hierarchy Chart

CPrintDialogEx Member Functions

For information about the member functions in **CPrintDialogEx**, see CPrintDialogEx Members.

CPrintDialogEx::CPrintDialogEx

Constructs a Windows 2000 Print property sheet.

```
CPrintDialogEx(
   DWORD dwFlags = PD_ALLPAGES | PD_USEDEVMODECOPIES | PD_NOPAGENUMS
      | PD_HIDEPRINTTOFILE | PD_NOSELECTION | PD_NOCURRENTPAGE,
   CWnd* pParentWnd = NULL
);
```

Parameters

dwFlags

One or more flags you can use to customize the settings of the dialog box, combined using the bitwise OR operator. For example, the **PD_ALLPAGES** flag sets the default print range to all pages of the document. See the PRINTDLGEX structure in the *Platform SDK* for more information on these flags.

pParentWnd

A pointer to the dialog box's parent or owner window.

Remarks

This member function only constructs the object. Use the **DoModal** member function to display the dialog box.

See Also

Reference

CPrintDialogEx Class
Hierarchy Chart
CPrintDialogEx::DoModal
Print Property Sheet

Other Resources

CPrintDialogEx Members

CPrintDialogEx::CreatePrinterDC

Creates a printer device context (DC) from the DEVMODE and DEVNAMES structures.

```
HDC CreatePrinterDC( );
```

Return Value

Handle to the newly created printer device context.

Remarks

The returned DC is also stored in the **hDC** member of m_pdex.

This DC is assumed to be the current printer DC, and any other previously obtained printer DCs must be deleted. This function can be called, and the resulting DC used, without ever displaying the Print dialog box.

See Also

Reference

CPrintDialogEx Class
Hierarchy Chart
CPrintDialogEx::GetPrinterDC
CPrintDialogEx::GetDevMode

Other Resources

CPrintDialogEx Members

CPrintDialogEx::DoModal

Call this function to display the Windows 2000 common Print property sheet and allow the user to select various printing options such as the number of copies, page range, and whether copies should be collated.

```
virtual INT_PTR DoModal( );
```

Return Value

The INT_PTR return value is actually an HRESULT. See the Return Values section in PrintDlgEx in the *Platform SDK*.

Remarks

If you want to initialize the various print dialog options by setting members of the **m_pdex** structure, you should do this before calling **DoModal**, but after the dialog object is constructed.

After calling **DoModal**, you can call other member functions to retrieve the settings or information input by the user into the dialog box.

If the **PD_RETURNDC** flag is used when calling **DoModal**, a printer DC will be returned in the **hDC** member of m_pdex. This DC must be freed with a call to DeleteDC by the caller of **CPrintDialogEx**.

See Also

Reference

CPrintDialogEx Class
Hierarchy Chart
CPrintDialogEx::CPrintDialogEx
CDialog::DoModal

Other Resources

CPrintDialogEx Members

CPrintDialogEx::GetCopies

Call this function after calling **DoModal** to retrieve the number of copies requested.

```
int GetCopies( ) const;
```

Return Value

The number of copies requested.

See Also

Reference

CPrintDialogEx Class
Hierarchy Chart
CPrintDialogEx::PrintCollate

Other Resources

CPrintDialogEx Members

CPrintDialogEx::GetDefaults

Call this function to retrieve the device defaults of the default printer without displaying a dialog box.

```
BOOL GetDefaults( );
```

Return Value

TRUE if successful, otherwise **FALSE**.

Remarks

Creates a printer device context (DC) from the DEVMODE and DEVNAMES structures.

GetDefaults does not display the Print property sheet. Instead, it sets the **hDevNames** and **hDevMode** members of m_pdex to handles to the DEVMODE and DEVNAMES structures that are initialized for the system default printer. Both **hDevNames** and **hDevMode** must be NULL, or **GetDefaults** fails.

If the **PD_RETURNDC** flag is set, this function will not only return **hDevNames** and **hDevMode** (located in **m_pdex.hDevNames** and **m_pdex.hDevMode**) to the caller, but will also return a printer DC in **m_pdex.hDC**. It is the responsibility of the caller to delete the printer DC and call the Windows GlobalFree function on the handles when you are finished with the **CPrintDialogEx** object.

See Also

Reference

CPrintDialogEx Class
Hierarchy Chart
CPrintDialogEx::m_pdex
CPrintDialog::GetDeviceName
CPrintDialog::GetDriverName
CPrintDialog::GetPortName

Other Resources

CPrintDialogEx Members

CPrintDialogEx::GetDeviceName

Call this function after calling DoModal to retrieve the name of the currently selected printer, or after calling GetDefaults to retrieve the name of the default printer.

```
CString GetDeviceName( ) const;
```

Return Value

The name of the currently selected printer.

Remarks

Use a pointer to the **CString** object returned by **GetDeviceName** as the value of *lpszDeviceName* in a call to CDC::CreateDC.

See Also

Reference

CPrintDialogEx Class
Hierarchy Chart
CPrintDialogEx::GetDriverName
CPrintDialogEx::GetDevMode
CPrintDialogEx::GetPortName

Other Resources

CPrintDialogEx Members

CPrintDialogEx::GetDevMode

Call this function after calling DoModal or GetDefaults to retrieve information about the printing device.

```
LPDEVMODE GetDevMode( ) const;
```

Return Value

The DEVMODE data structure, which contains information about the device initialization and environment of a print driver. You must unlock the memory taken by this structure with the Windows GlobalUnlock function, which is described in the *Platform SDK*.

See Also

Reference

CPrintDialogEx Class
Hierarchy Chart
CDC::GetDeviceCaps

Other Resources

CPrintDialogEx Members

CPrintDialogEx::GetDriverName

Call this function after calling DoModal or GetDefaults to retrieve the name of the system-defined printer device driver.

```
CString GetDriverName( ) const;
```

Return Value

A **CString** specifying the system-defined driver name.

Remarks

Use a pointer to the **CString** object returned by **GetDriverName** as the value of *lpszDriverName* in a call to CDC::CreateDC.

See Also

Reference

CPrintDialogEx Class
Hierarchy Chart
CPrintDialogEx::GetDeviceName
CPrintDialogEx::GetDevMode
CPrintDialogEx::GetPortName

Other Resources

CPrintDialogEx Members

CPrintDialogEx::GetPortName

Call this function after calling DoModal or GetDefaults to retrieve the name of the currently selected printer port.

```
CString GetPortName( ) const;
```

Return Value

The name of the currently selected printer port.

See Also

Reference

CPrintDialogEx Class
Hierarchy Chart
CPrintDialogEx::GetDriverName
CPrintDialogEx::GetDeviceName

Other Resources

CPrintDialogEx Members

CPrintDialogEx::GetPrinterDC

Returns a handle to the printer device context.

```
HDC GetPrinterDC( ) const;
```

Return Value

A handle to the printer device context.

Remarks

You must call the Windows DeleteDC function to delete the device context when you are done using it.

See Also

Reference

CPrintDialogEx Class
Hierarchy Chart
CPrintDialogEx::CreatePrinterDC

Other Resources

CPrintDialogEx Members

CPrintDialogEx::PrintAll

Call this function after calling **DoModal** to determine whether to print all pages in the document.

```
BOOL PrintAll( ) const;
```

Return Value

TRUE if all pages in the document are to be printed; otherwise **FALSE**.

See Also

Reference

CPrintDialogEx Class
Hierarchy Chart
CPrintDialogEx::PrintCurrentPage
CPrintDialogEx::PrintRange
CPrintDialogEx::PrintSelection

Other Resources

CPrintDialogEx Members

CPrintDialogEx::PrintCollate

Call this function after calling **DoModal** to determine whether the printer should collate all printed copies of the document.

```
BOOL PrintCollate( ) const;
```

Return Value

TRUE if the user selects the collate check box in the dialog box; otherwise **FALSE**.

See Also

Reference

CPrintDialogEx Class
Hierarchy Chart
CPrintDialogEx::GetCopies

Other Resources

CPrintDialogEx Members

CPrintDialogEx::PrintCurrentPage

Call this function after calling **DoModal** to determine whether to print the current page in the document.

```
BOOL PrintCurrentPage( ) const;
```

Return Value

TRUE if **Print Current Page** is selected in the print dialog; otherwise **FALSE**.

See Also

Reference

CPrintDialogEx Class
Hierarchy Chart
CPrintDialogEx::PrintAll
CPrintDialogEx::PrintSelection

Other Resources

CPrintDialogEx Members

CPrintDialogEx::PrintRange

Call this function after calling **DoModal** to determine whether to print only a range of pages in the document.

```
BOOL PrintRange( ) const;
```

Return Value

TRUE if only a range of pages in the document are to be printed; otherwise **FALSE**.

Remarks

The specified page ranges can be determined from m_pdex (see **nPageRanges**, **nMaxPageRanges**, and **lpPageRanges** in the PRINTDLGEX structure in the *Platform SDK*).

See Also

Reference

CPrintDialogEx Class
Hierarchy Chart
CPrintDialogEx::PrintAll
CPrintDialogEx::PrintCurrentPage
CPrintDialogEx::PrintSelection

Other Resources

CPrintDialogEx Members

CPrintDialogEx::PrintSelection

Call this function after calling **DoModal** to determine whether to print only the currently selected items.

```
BOOL PrintSelection( ) const;
```

Return Value

TRUE if only the selected items are to be printed; otherwise **FALSE**.

See Also

Reference

CPrintDialogEx Class
Hierarchy Chart
CPrintDialogEx::PrintAll
CPrintDialogEx::PrintRange

Other Resources

CPrintDialogEx Members

CPrintDialogEx Data Members

For information about the data members in **CPrintDialogEx**, see CPrintDialogEx Members.

CPrintDialogEx::m_pdex

A PRINTDLGEX structure whose members store the characteristics of the dialog object.

```
PRINTDLGEX m_pdex;
```

Remarks

After constructing a **CPrintDialogEx** object, you can use **m_pdex** to set various aspects of the dialog box before calling the DoModal member function. For more information on the **m_pdex** structure, see PRINTDLGEX in the *Platform SDK*.

If you modify the **m_pdex** data member directly, you will override any default behavior.

See Also

Reference

CPrintDialogEx Class

Hierarchy Chart

Other Resources

CPrintDialogEx Members

CPrintInfo Structure

Stores information about a print or print-preview job.

```
struct CPrintInfo
```

Remarks

CPrintInfo is a structure and does not have a base class.

The framework creates an object of **CPrintInfo** each time the Print or Print Preview command is chosen and destroys it when the command is completed.

CPrintInfo contains information about both the print job as a whole, such as the range of pages to be printed, and the current status of the print job, such as the page currently being printed. Some information is stored in an associated CPrintDialog object; this object contains the values entered by the user in the Print dialog box.

A **CPrintInfo** object is passed between the framework and your view class during the printing process and is used to exchange information between the two. For example, the framework informs the view class which page of the document to print by assigning a value to the **m_nCurPage** member of **CPrintInfo**; the view class retrieves the value and performs the actual printing of the specified page.

Another example is the case in which the length of the document is not known until it is printed. In this situation, the view class tests for the end of the document each time a page is printed. When the end is reached, the view class sets the **m_bContinuePrinting** member of **CPrintInfo** to **FALSE**; this informs the framework to stop the print loop.

CPrintInfo is used by the member functions of **CView** listed under "See Also." For more information about the printing architecture provided by the Microsoft Foundation Class Library, see Frame Windows and Document/View Architecture and the articles Printing and Printing: Multipage Documents.

Requirements

Header: afxext.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

DIBLOOK Sample: Illustrates the Use of DIBs and Color Palettes

Reference

Hierarchy Chart
CView::OnBeginPrinting
CView::OnEndPrinting
CView::OnEndPrintPreview
CView::OnPrepareDC
CView::OnPreparePrinting
CView::OnPrint

Other Resources

CPrintInfo Members

CPrintInfo Members

Data Members

m_bContinuePrinting	Contains a flag indicating whether the framework should continue the print loop.
m_bDirect	Contains a flag indicating whether the document is being printed directly (without displaying the Print dialog box).
m_bDocObject	Contains a flag indicating whether the document being printed is a DocObject.
m_bPreview	Contains a flag indicating whether the document is being previewed.
m_dwFlags	Specifies DocObject printing operations.
m_lpUserData	Contains a pointer to a user-created structure.
m_nCurPage	Identifies the number of the page currently being printed.
m_nJobNumber	Specifies the job number assigned by the operating system for the current print job
m_nNumPreviewPages	Identifies the number of pages displayed in the preview window; either 1 or 2.
m_nOffsetPage	Specifies offset of a particular DocObject's first page in a combined DocObject print job.
m_pPD	Contains a pointer to the **CPrintDialog** object used for the Print dialog box.
m_rectDraw	Specifies a rectangle defining the current usable page area.
m_strPageDesc	Contains a format string for page-number display.

Attributes

GetFromPage	Returns the number of the first page being printed.
GetMaxPage	Returns the number of the last page of the document.
GetMinPage	Returns the number of the first page of the document.
GetOffsetPage	Returns the number of the pages preceding the first page of a DocObject item being printed in a combined Doc Object print job.
GetToPage	Returns the number of the last page being printed.
SetMaxPage	Sets the number of the last page of the document.
SetMinPage	Sets the number of the first page of the document.

See Also

Reference

CPrintInfo Structure

Hierarchy Chart

CPrintInfo Member Functions

For information about the member functions in **CPrintInfo**, see CPrintInfo Members.

CPrintInfo::GetFromPage

Call this function to retrieve the number of the first page to be printed.

```
UINT GetFromPage( ) const;
```

Return Value

The number of the first page to be printed.

Remarks

This is the value specified by the user in the Print dialog box, and it is stored in the **CPrintDialog** object referenced by the **m_pPD** member. If the user has not specified a value, the default is the first page of the document.

See Also

Reference

CPrintInfo Structure
Hierarchy Chart
CPrintInfo::m_nCurPage
CPrintInfo::m_pPD
CPrintInfo::GetToPage

Other Resources

CPrintInfo Members

CPrintInfo::GetMaxPage

Call this function to retrieve the number of the last page of the document.

```
UINT GetMaxPage( ) const;
```

Return Value

The number of the last page of the document.

Remarks

This value is stored in the **CPrintDialog** object referenced by the **m_pPD** member.

See Also

Reference

CPrintInfo Structure
Hierarchy Chart
CPrintInfo::m_nCurPage
CPrintInfo::m_pPD
CPrintInfo::GetMinPage
CPrintInfo::SetMaxPage
CPrintInfo::SetMinPage

Other Resources

CPrintInfo Members

CPrintInfo::GetMinPage

Call this function to retrieve the number of the first page of the document.

```
UINT GetMinPage( ) const;
```

Return Value

The number of the first page of the document.

Remarks

This value is stored in the **CPrintDialog** object referenced by the **m_pPD** member.

See Also

Reference

CPrintInfo Structure
Hierarchy Chart
CPrintInfo::m_nCurPage
CPrintInfo::m_pPD
CPrintInfo::GetMaxPage
CPrintInfo::SetMaxPage
CPrintInfo::SetMinPage

Other Resources

CPrintInfo Members

CPrintInfo::GetOffsetPage

Call this function to retrieve the offset when printing multiple DocObject items from a DocObject client.

```
UINT GetOffsetPage( ) const;
```

Return Value

The number of pages preceding the first page of a DocObject item being printed in a combined DocObject print job.

Remarks

This value is referenced by the **m_nOffsetPage** member. The first page of your document will be numbered the **m_nOffsetPage** value + 1 when printed as a DocObject with other active documents. The **m_nOffsetPage** member is valid only if the **m_bDocObject** value is **TRUE**.

See Also

Reference

CPrintInfo Structure
Hierarchy Chart
CPrintInfo::m_nOffsetPage
CPrintInfo::m_bDocObject

Other Resources

CPrintInfo Members

CPrintInfo::GetToPage

Call this function to retrieve the number of the last page to be printed.

```
UINT GetToPage( ) const;
```

Return Value

The number of the last page to be printed.

Remarks

This is the value specified by the user in the Print dialog box, and it is stored in the **CPrintDialog** object referenced by the **m_pPD** member. If the user has not specified a value, the default is the last page of the document.

See Also

Reference

CPrintInfo Structure
Hierarchy Chart
CPrintInfo::m_nCurPage
CPrintInfo::m_pPD
CPrintInfo::GetFromPage

Other Resources

CPrintInfo Members

CPrintInfo::SetMaxPage

Call this function to specify the number of the last page of the document.

```
void SetMaxPage(
   UINT nMaxPage
);
```

Parameters

nMaxPage

Number of the last page of the document.

Remarks

This value is stored in the **CPrintDialog** object referenced by the **m_pPD** member. If the length of the document is known before it is printed, call this function from your override of **CView::OnPreparePrinting**. If the length of the document depends on a setting specified by the user in the Print dialog box, call this function from your override of **CView::OnBeginPrinting**. If the length of the document is not known until it is printed, use the **m_bContinuePrinting** member to control the print loop.

Example

See the example for CView::OnPreparePrinting.

See Also

Reference

CPrintInfo Structure
Hierarchy Chart
CPrintInfo::m_bContinuePrinting
CPrintInfo::m_nCurPage
CPrintInfo::m_pPD
CPrintInfo::GetMinPage
CPrintInfo::GetToPage
CPrintInfo::SetMinPage
CView::OnBeginPrinting
CView::OnPreparePrinting

Other Resources

CPrintInfo Members

CPrintInfo::SetMinPage

Call this function to specify the number of the first page of the document.

```
void SetMinPage(
   UINT nMinPage
);
```

Parameters

nMinPage

Number of the first page of the document.

Remarks

Page numbers normally start at 1. This value is stored in the **CPrintDialog** object referenced by the **m_pPD** member.

See Also

Reference

CPrintInfo Structure
Hierarchy Chart
CPrintInfo::m_nCurPage
CPrintInfo::m_pPD
CPrintInfo::GetMaxPage
CPrintInfo::GetMinPage
CPrintInfo::SetMaxPage

Other Resources

CPrintInfo Members

CPrintInfo Data Members

For information about the data members in **CPrintInfo**, see CPrintInfo Members.

CPrintInfo::m_bContinuePrinting

Contains a flag indicating whether the framework should continue the print loop.

Remarks

If you are doing print-time pagination, you can set this member to **FALSE** in your override of **CView::OnPrepareDC** once the end of the document has been reached. You do not have to modify this variable if you have specified the length of the document at the beginning of the print job using the **SetMaxPage** member function. The **m_bContinuePrinting** member is a public variable of type **BOOL**.

See Also

Reference

CPrintInfo Structure
Hierarchy Chart
CPrintInfo::SetMaxPage
CView::OnPrepareDC

Other Resources

CPrintInfo Members

CPrintInfo::m_bDirect

The framework sets this member to **TRUE** if the Print dialog box will be bypassed for direct printing; **FALSE** otherwise.

Remarks

The Print dialog is normally bypassed when you print from the shell or when printing is done using the command ID **ID_FILE_PRINT_DIRECT**.

You normally don't change this member, but if you do change it, change it before you call CView::DoPreparePrinting in your override of CView::OnPreparePrinting.

See Also

Reference

CPrintInfo Structure
Hierarchy Chart
CView::DoPreparePrinting
CView::OnPreparePrinting

Other Resources

CPrintInfo Members

CPrintInfo::m_bDocObject

Contains a flag indicating whether the document being printed is a DocObject.

Remarks

Data members **m_dwFlags** and **m_nOffsetPage** are invalid unless this flag is **TRUE**.

See Also

Reference

CPrintInfo Structure
Hierarchy Chart
CPrintInfo::m_dwFlags
CPrintInfo::m_nOffsetPage

Other Resources

CPrintInfo Members

CPrintInfo::m_bPreview

Contains a flag indicating whether the document is being previewed.

Remarks

This is set by the framework depending on which command the user executed. The Print dialog box is not displayed for a print-preview job. The **m_bPreview** member is a public variable of type **BOOL**.

See Also

Reference

CPrintInfo Structure
Hierarchy Chart
CView::DoPreparePrinting
CView::OnPreparePrinting

Other Resources

CPrintInfo Members

CPrintInfo::m_dwFlags

Contains a combination of flags specifying DocObject printing operations.

Remarks

Valid only if data member **m_bDocObject** is **TRUE**.

The flags can be one or more of the following values:

- **PRINTFLAG_MAYBOTHERUSER**
- **PRINTFLAG_PROMPTUSER**
- **PRINTFLAG_USERMAYCHANGEPRINTER**
- **PRINTFLAG_RECOMPOSETODEVICE**
- **PRINTFLAG_DONTACTUALLYPRINT**
- **PRINTFLAG_FORCEPROPERTIES**
- **PRINTFLAG_PRINTTOFILE**

See Also

Reference

CPrintInfo Structure
Hierarchy Chart
CPrintInfo::m_bDocObject
CPrintInfo::m_nOffsetPage

Other Resources

CPrintInfo Members

CPrintInfo::m_lpUserData

Contains a pointer to a user-created structure.

Remarks

You can use this to store printing-specific data that you do not want to store in your view class. The **m_lpUserData** member is a public variable of type **LPVOID**.

See Also

Reference

CPrintInfo Structure

Hierarchy Chart

Other Resources

CPrintInfo Members

CPrintInfo::m_nCurPage

Contains the number of the current page.

Remarks

The framework calls **CView::OnPrepareDC** and **CView::OnPrint** once for each page of the document, specifying a different value for this member each time; its values range from the value returned by **GetFromPage** to that returned by **GetToPage**. Use this member in your overrides of **CView::OnPrepareDC** and **CView::OnPrint** to print the specified page of the document.

When preview mode is first invoked, the framework reads the value of this member to determine which page of the document should be previewed initially. You can set the value of this member in your override of **CView::OnPreparePrinting** to maintain the user's current position in the document when entering preview mode. The **m_nCurPage** member is a public variable of type **UINT**.

See Also

Reference

CPrintInfo Structure
Hierarchy Chart
CPrintInfo::GetFromPage
CPrintInfo::GetToPage
CView::OnPrepareDC
CView::OnPreparePrinting
CView::OnPrint

Other Resources

CPrintInfo Members

CPrintInfo::m_nJobNumber

Indicates the job number assigned by the operating system for the current print job.

Remarks

This value may be **SP_ERROR** if the job hasn't yet printed (that is, if the **CPrintInfo** object is newly constructed and has not yet been used to print), or if there was an error in starting the job.

See Also

Reference

CPrintInfo Structure

Hierarchy Chart

Other Resources

CPrintInfo Members

CPrintInfo::m_nNumPreviewPages

Contains the number of pages displayed in preview mode; it can be either 1 or 2.

Remarks

The **m_nNumPreviewPages** member is a public variable of type **UINT**.

See Also

Reference

CPrintInfo Structure

Hierarchy Chart

CPrintInfo::m_strPageDesc

Other Resources

CPrintInfo Members

CPrintInfo::m_nOffsetPage

Contains the number of pages preceding the first page of a particular DocObject in a combined DocObject print job.

See Also

Reference

CPrintInfo Structure
Hierarchy Chart
CPrintInfo::m_bDocObject
CPrintInfo::m_dwFlags

Other Resources

CPrintInfo Members

CPrintInfo::m_pPD

Contains a pointer to the **CPrintDialog** object used to display the Print dialog box for the print job.

Remarks

The **m_pPD** member is a public variable declared as a pointer to **CPrintDialog**.

See Also

Reference

CPrintInfo Structure

Hierarchy Chart

CPrintDialog Class

Other Resources

CPrintInfo Members

CPrintInfo::m_rectDraw

Specifies the usable drawing area of the page in logical coordinates.

Remarks

You may want to refer to this in your override of **CView::OnPrint**. You can use this member to keep track of what area remains usable after you print headers, footers, and so on. The **m_rectDraw** member is a public variable of type **CRect**.

See Also

Reference

CPrintInfo Structure

Hierarchy Chart

CView::OnPrint

Other Resources

CPrintInfo Members

CPrintInfo::m_strPageDesc

Contains a format string used to display the page numbers during print preview; this string consists of two substrings, one for single-page display and one for double-page display, each terminated by a '\n' character.

Remarks

The framework uses "Page %u\nPages %u-%u\n" as the default value. If you want a different format for the page numbers, specify a format string in your override of **CView::OnPreparePrinting**. The **m_strPageDesc** member is a public variable of type **CString**.

See Also

Reference

CPrintInfo Structure
Hierarchy Chart
CView::OnPreparePrinting

Other Resources

CPrintInfo Members

CProgressCtrl Class

Provides the functionality of the Windows common progress bar control.

```
class CProgressCtrl : public CWnd
```

Remarks

A "progress bar control" is a window that an application can use to indicate the progress of a lengthy operation. It consists of a rectangle that is gradually filled, from left to right, with the system highlight color as an operation progresses.

This control (and therefore the **CProgressCtrl** class) is available only to programs running under Windows 95/98 and Windows NT version 3.51 and later.

A progress bar control has a range and a current position. The range represents the entire duration of the operation, and the current position represents the progress the application has made toward completing the operation. The window procedure uses the range and the current position to determine the percentage of the progress bar to fill with the highlight color and to determine the text, if any, to display within the progress bar. Because the range and current position values are expressed as signed integers, the possible range of current position values is from -217483648 to 217483648 inclusive.

For more information on using **CProgressCtrl**, see Controls and Using CProgressCtrl.

Requirements

Header: afxcmn.h

See Also

Tasks

CMNCTRL2 Sample: Demonstrates Common Control MFC Classes, Part 2

Reference

CWnd Class

Hierarchy Chart

Other Resources

CProgressCtrl Members

CProgressCtrl Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Construction

CProgressCtrl	Constructs a **CProgressCtrl** object.
Create	Creates a progress bar control and attaches it to a **CProgressCtrl** object.
CreateEx	Creates a progress control with the specified Windows extended styles and attaches it to a **CProgressCtrl** object.

Attributes

GetPos	Gets the current position of the progress bar.
GetRange	Gets the lower and upper limits of the range of the progress bar control.
OffsetPos	Advances the current position of a progress bar control by a specified increment and redraws the bar to reflect the new position.
SetBkColor	Sets the background color for the progress bar.
SetPos	Sets the current position for a progress bar control and redraws the bar to reflect the new position.
SetRange	Sets the minimum and maximum ranges for a progress bar control and redraws the bar to reflect the new ranges.
SetRange32	Sets the minimum and maximum ranges for a progress bar control and redraws the bar to reflect the new ranges.
SetStep	Specifies the step increment for a progress bar control.

Operations

StepIt	Advances the current position for a progress bar control by the step increment (see SetStep) and redraws the bar to reflect the new position.

See Also

Reference

CProgressCtrl Class

Hierarchy Chart

CProgressCtrl Member Functions

For information about the member functions in **CProgressCtrl**, see CProgressCtrl Members.

CProgressCtrl::CProgressCtrl

Constructs a **CProgressCtrl** object.

```
CProgressCtrl( );
```

Remarks

After constructing the **CProgressCtrl** object, call **CProgressCtrl::Create** to create the progress bar control.

Example

```
// Create a progress control object on the stack.
CProgressCtrl myCtrl;

// Create a progress control object on the heap.
CProgressCtrl* pmyCtrl = new CProgressCtrl;
```

See Also

Reference

CProgressCtrl Class
Hierarchy Chart
CProgressCtrl::Create
CProgressCtrl::CreateEx

Other Resources

CProgressCtrl Members

CProgressCtrl::Create

Creates a progress bar control and attaches it to a **CProgressCtrl** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the progress bar control's style. Apply any combination of window stylesdescribed in CreateWindow in the *Platform SDK*, in addition to the following progress bar control styles, to the control:

- **PBS_VERTICAL** Displays progress information vertically, top to bottom. Without this flag, the progress bar control displays horizontally, left to right.
- **PBS_SMOOTH** Displays gradual, smooth filling in the progress bar control. Without this flag, the control will fill with blocks.

rect

Specifies the progress bar control's size and position. It can be either a CRect object or a RECT structure. Because the control must be a child window, the specified coordinates are relative to the client area of the *pParentWnd*.

pParentWnd

Specifies the progress bar control's parent window, usually a **CDialog**. It must not be **NULL.**

nID

Specifies the progress bar control's ID.

Return Value

TRUE if the **CProgressCtrl** object is successfully created; otherwise **FALSE**.

Remarks

You construct a **CProgressCtrl** object in two steps. First, call the constructor, which creates the **CProgressCtrl** object, and then call **Create**, which creates the progress bar control.

Example

```
CProgressCtrl myCtrl;

// Create a smooth child progress control.
myCtrl.Create(WS_CHILD|WS_VISIBLE|PBS_SMOOTH, CRect(10,10,200,30),
   pParentWnd, 1);
```

See Also

Reference

CProgressCtrl Class
Hierarchy Chart
CProgressCtrl::CProgressCtrl
CProgressCtrl::CreateEx

Other Resources

CProgressCtrl Members

CProgressCtrl::CreateEx

Creates a control (a child window) and associates it with the **CProgressCtrl** object.

```
virtual BOOL CreateEx(
   DWORD dwExStyle,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwExStyle

Specifies the extended style of the control being created. For a list of extended Windows styles, see the *dwExStyle* parameter for CreateWindowEx in the *Platform SDK*.

dwStyle

Specifies the progress bar control's style. Apply any combination of window styles described in CreateWindow in the *Platform SDK*.

rect

A reference to a RECT structure describing the size and position of the window to be created, in client coordinates of *pParentWnd*.

pParentWnd

A pointer to the window that is the control's parent.

nID

The control's child-window ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Use **CreateEx** instead of Create to apply extended Windows styles, specified by the Windows extended style preface **WS_EX_**.

See Also

Reference

CProgressCtrl::CProgressCtrl
CProgressCtrl Class
Hierarchy Chart

Other Resources

CProgressCtrl Members

CProgressCtrl::GetPos

Retrieves the current position of the progress bar.

```
int GetPos( );
```

Return Value

The position of the progress bar control.

Remarks

The position of the progress bar control is not the physical location on the screen, but rather is between the upper and lower range indicated in SetRange.

Example

```
CProgressCtrl myCtrl;

// Create a child progress control.
myCtrl.Create(WS_CHILD|WS_VISIBLE, CRect(10,10,200,30), pParentWnd, 1);

// Set the new position to half of the current position.
myCtrl.SetPos( myCtrl.GetPos()/2 );
```

See Also

Reference

CProgressCtrl Class
Hierarchy Chart
CProgressCtrl::SetPos

Other Resources

CProgressCtrl Members

CProgressCtrl::GetRange

Gets the current lower and upper limits, or range, of the progress bar control.

```
void GetRange(
   int& nLower,
   int& nUpper
);
```

Parameters

nLower

A reference to an integer receiving the lower limit of the progress bar control.

nUpper

A reference to an integer receiving the upper limit of the progress bar control.

Remarks

This function copies the values of the lower and upper limits to the integers referenced by *nLower* and *nUpper*, respectively.

Example

```
CProgressCtrl myCtrl;

// Create a child progress control.
myCtrl.Create(WS_CHILD|WS_VISIBLE, CRect(10,10,200,30), pParentWnd, 1);

// Set the position to be one-fourth of the total range.
int nLower, nUpper;
myCtrl.GetRange( nLower, nUpper );
myCtrl.SetPos( (nUpper-nLower)/4 );
```

See Also

Reference

CProgressCtrl Class
Hierarchy Chart
PBRANGE
PBM_GETRANGE

Other Resources

CProgressCtrl Members

CProgressCtrl::OffsetPos

Advances the progress bar control's current position by the increment specified by *nPos* and redraws the bar to reflect the new position.

```
int OffsetPos(
   int nPos
);
```

Parameters

nPos

Amount to advance the position.

Return Value

The previous position of the progress bar control.

Example

```
CProgressCtrl myCtrl;

// Create a child progress control.
myCtrl.Create(WS_CHILD|WS_VISIBLE, CRect(10,10,200,30), pParentWnd, 1);

// Offset the position by one-fourth of the total range.
int nLower, nUpper;
myCtrl.GetRange( nLower, nUpper );
myCtrl.OffsetPos( (nUpper-nLower)/4 );
```

See Also

Reference

CProgressCtrl Class
Hierarchy Chart
CProgressCtrl::SetPos
CProgressCtrl::SetRange
CProgressCtrl::StepIt

Other Resources

CProgressCtrl Members

CProgressCtrl::SetBkColor

Sets the background color for the progress bar.

```
COLORREF SetBkColor(
   COLORREF clrNew
);
```

Parameters

clrNew

A **COLORREF** value that specifies the new background color. Specify the **CLR_DEFAULT** value to use the default background color for the progress bar.

Return Value

The COLORREF value indicating the previous background color, or **CLR_DEFAULT** if the background color is the default color.

Example

```
CProgressCtrl myCtrl;

// Create a smooth child progress control.
myCtrl.Create(WS_CHILD|WS_VISIBLE|PBS_SMOOTH, CRect(10,10,200,30),
   pParentWnd, 1);

// Set the background color to red.
myCtrl.SetBkColor(RGB(255, 0, 0));
```

See Also

Reference

CProgressCtrl Class

Hierarchy Chart

PBM_SETBKCOLOR

Other Resources

CProgressCtrl Members

CProgressCtrl::SetPos

Sets the progress bar control's current position as specified by *nPos* and redraws the bar to reflect the new position.

```
int SetPos(
   int nPos
);
```

Parameters

nPos

New position of the progress bar control.

Return Value

The previous position of the progress bar control.

Remarks

The position of the progress bar control is not the physical location on the screen, but rather is between the upper and lower range indicated in SetRange.

Example

```
CProgressCtrl myCtrl;

// Create a child progress control.
myCtrl.Create(WS_CHILD|WS_VISIBLE, CRect(10,10,200,30), pParentWnd, 1);

// Set the range to be 0 to 100.
myCtrl.SetRange( 0, 100 );

// Set the position to be half, 50.
myCtrl.SetPos(50);
```

See Also

Reference

CProgressCtrl Class
Hierarchy Chart
CProgressCtrl::OffsetPos
CProgressCtrl::SetRange
CProgressCtrl::StepIt

Other Resources

CProgressCtrl Members

CProgressCtrl::SetRange

Sets the upper and lower limits of the progress bar control's range and redraws the bar to reflect the new ranges.

```
void SetRange(
   short nLower,
   short nUpper
);
void SetRange32(
   int nLower,
   int nUpper
);
```

Parameters

nLower

Specifies the lower limit of the range (default is zero).

nUpper

Specifies the upper limit of the range (default is 100).

Remarks

The member function **SetRange32** sets the 32-bit range for the progress control.

Example

```
CProgressCtrl myCtrl;

// Create a smooth child progress control.
myCtrl.Create(WS_CHILD|WS_VISIBLE|PBS_SMOOTH, CRect(10,10,200,30),
   pParentWnd, 1);

// Set the range to be 0 to 100.
myCtrl.SetRange( 0, 100 );
```

See Also

Reference

CProgressCtrl Class
Hierarchy Chart
CProgressCtrl::OffsetPos
CProgressCtrl::SetPos
CProgressCtrl::StepIt

Other Resources

CProgressCtrl Members

CProgressCtrl::SetStep

Specifies the step increment for a progress bar control.

```
int SetStep(
   int nStep
);
```

Parameters

nStep

New step increment.

Return Value

The previous step increment.

Remarks

The step increment is the amount by which a call to **CProgressCtrl::StepIt** increases the progress bar's current position.

The default step increment is 10.

Example

```
CProgressCtrl myCtrl;

// Create a child progress control.
myCtrl.Create(WS_CHILD|WS_VISIBLE, CRect(10,10,200,30), pParentWnd, 1);

// Set the size to be 1/10 of the total range.
int nLower, nUpper;
myCtrl.GetRange( nLower, nUpper );
myCtrl.SetStep( (nUpper-nLower)/10 );
```

See Also

Reference

CProgressCtrl Class
Hierarchy Chart
CProgressCtrl::OffsetPos
CProgressCtrl::SetPos
CProgressCtrl::StepIt

Other Resources

CProgressCtrl Members

CProgressCtrl::StepIt

Advances the current position for a progress bar control by the step increment and redraws the bar to reflect the new position.

```
int StepIt( );
```

Return Value

The previous position of the progress bar control.

Remarks

The step increment is set by the **CProgressCtrl::SetStep** member function.

Example

```
CProgressCtrl myCtrl;

// Create a child progress control.
myCtrl.Create(WS_CHILD|WS_VISIBLE, CRect(10,10,200,30), pParentWnd, 1);

// Advance the position to the next step.
myCtrl.StepIt();
```

See Also

Reference

CProgressCtrl::SetPos
CProgressCtrl::SetRange
CProgressCtrl::SetStep

CPropertyPage Class

Represents individual pages of a property sheet, otherwise known as a tab dialog box.

```
class CPropertyPage : public CDialog
```

Remarks

As with standard dialog boxes, you derive a class from **CPropertyPage** for each page in your property sheet. To use **CPropertyPage**-derived objects, first create a CPropertySheet object, and then create an object for each page that goes in the property sheet. Call CPropertySheet::AddPage for each page in the sheet, and then display the property sheet by calling CPropertySheet::DoModal for a modal property sheet, or CPropertySheet::Create for a modeless property sheet.

You can create a type of tab dialog box called a wizard, which consists of a property sheet with a sequence of property pages that guide the user through the steps of an operation, such as setting up a device or creating a newsletter. In a wizard-type tab dialog box, the property pages do not have tabs, and only one property page is visible at a time. Also, instead of having OK and Apply Now buttons, a wizard-type tab dialog box has a Back button, a Next or Finish button, and a Cancel button.

For more information on establishing a property sheet as a wizard, see CPropertySheet::SetWizardMode. For more information on using **CPropertyPage** objects, see the article Property Sheets and Property Pages.

Requirements

Header: afxdlgs.h

Smart Device Developer Notes

Not supported for Smartphone projects.

See Also

Tasks

CMNCTRL1 Sample: Demonstrates Common Control MFC Classes, Part 1
CMNCTRL2 Sample: Demonstrates Common Control MFC Classes, Part 2
PROPDLG Sample: Demonstrates Property Sheet Support
SNAPVW Sample: Uses Property Pages in a Form View Application

Reference

CDialog Class
Hierarchy Chart
CPropertySheet Class
CDialog Class
CPropertySheet::SetWizardMode

Other Resources

CPropertyPage Members

CPropertyPage Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CDialog Members

Data Members

m_psp	The Windows PROPSHEETPAGE structure. Provides access to basic property page parameters.

Attributes

GetPSP	Retrieves the Windows PROPSHEETPAGE structure associated with the **CPropertyPage** object.

Construction

Construct	Constructs a **CPropertyPage** object. Use **Construct** if you want to specify your parameters at run time, or if y ou are using arrays.
CPropertyPage	Constructs a **CPropertyPage** object.

Operations

CancelToClose	Changes the OK button to read Close, and disables the Cancel button, after an unrecoverable change in the pag e of a modal property sheet.
QuerySiblings	Forwards the message to each page of the property sheet.
SetModified	Call to activate or deactivate the Apply Now button.

Overridables

OnApply	Called by the framework when the Apply Now button is clicked.
OnCancel	Called by the framework when the Cancel button is clicked.
OnKillActive	Called by the framework when the current page is no longer the active page. Perform data validation here.
OnOK	Called by the framework when the OK, Apply Now, or Close button is clicked.
OnQueryCancel	Called by the framework when the Cancel button is clicked, and before the cancel has taken place.
OnReset	Called by the framework when the Cancel button is clicked.
OnSetActive	Called by the framework when the page is made the active page.
OnWizardBack	Called by the framework when the Back button is clicked while using a wizard-type property sheet.
OnWizardFinish	Called by the framework when the Finish button is clicked while using a wizard-type property sheet.
OnWizardNext	Called by the framework when the Next button is clicked while using a wizard-type property sheet.

See Also

Reference

CPropertyPage Class

Hierarchy Chart

CPropertyPage Member Functions

For information about the member functions in **CPropertyPage**, see CPropertyPage Members.

CPropertyPage::CancelToClose

Call this function after an unrecoverable change has been made to the data in a page of a modal property sheet.

```
void CancelToClose( );
```

Remarks

This function will change the OK button to Close and disable the Cancel button. This change alerts the user that a change is permanent and the modifications cannot be cancelled.

The **CancelToClose** member function does nothing in a modeless property sheet, because a modeless property sheet does not have a Cancel button by default.

Example

See the example for CPropertyPage::QuerySiblings.

See Also

Reference

CPropertyPage Class
Hierarchy Chart
CPropertyPage::OnKillActive
CPropertyPage::SetModified

Other Resources

CPropertyPage Members

CPropertyPage::Construct

Call this member function to construct a **CPropertyPage** object.

```
void Construct(
   UINT nIDTemplate,
   UINT nIDCaption = 0
);
void Construct(
   LPCTSTR lpszTemplateName,
   UINT nIDCaption = 0
);
void Construct(
   UINT nIDTemplate,
   UINT nIDCaption,
   UINT nIDHeaderTitle,
   UINT nIDHeaderSubTitle = 0
);
void Construct(
   LPCTSTR lpszTemplateName,
   UINT nIDCaption,
   UINT nIDHeaderTitle,
   UINT nIDHeaderSubTitle = 0
);
```

Parameters

nIDTemplate

ID of the template used for this page.

nIDCaption

ID of the name to be placed in the tab for this page. If 0, the name will be taken from the dialog template for this page.

lpszTemplateName

Contains a null-terminated string that is the name of a template resource.

nIDHeaderTitle

ID of the name to be placed in the title location of the property page header. By default, 0.

nIDHeaderSubTitle

ID of the name to be placed in the subtitle location of the property page header. By default, 0.

Remarks

The object is displayed after all of the following conditions are met:

- The page has been added to a property sheet using CPropertySheet::AddPage.
- The property sheet's DoModal or Create function has been called.
- The user has selected (tabbed to) this page.

Call **Construct** if one of the other class constructors has not been called. The **Construct** member function is flexible because you can leave the parameter statement blank and then specify multiple parameters and construction at any point in your code.

You must use **Construct** when you work with arrays, and you must call **Construct** for each member of the array so that the data members are assigned proper values.

Example

```
// Declare a CPropertySheet object.
CPropertySheet sheet("Simple PropertySheet");

// Create three CPropertyPage objects whose template IDs are specified
```

```
// in rgID array, and add each page to the CPropertySheet object.
CPropertyPage   pages[3];
UINT rgID[4] = {IDD_STYLE, IDD_COLOR, IDD_SHAPE};
for (int i = 0; i < 3; i++)
{
   pages[i].Construct(rgID[i]);
   sheet.AddPage(&pages[i]);
}

// Display a modal CPropertySheet dialog.
sheet.DoModal();
```

See Also

Reference

CPropertyPage Class

Hierarchy Chart

CPropertyPage::CPropertyPage

CPropertySheet::DoModal

CPropertySheet::AddPage

Other Resources

CPropertyPage Members

CPropertyPage::CPropertyPage

Constructs a **CPropertyPage** object.

```
CPropertyPage( );
explicit CPropertyPage(
   UINT nIDTemplate,
   UINT nIDCaption = 0,
   DWORD dwSize = sizeof(PROPSHEETPAGE)
);
explicit CPropertyPage(
   LPCTSTR lpszTemplateName,
   UINT nIDCaption = 0,
   DWORD dwSize = sizeof(PROPSHEETPAGE)
);
CPropertyPage(
   UINT nIDTemplate,
   UINT nIDCaption,
   UINT nIDHeaderTitle,
   UINT nIDHeaderSubTitle = 0,
   DWORD dwSize = sizeof(PROPSHEETPAGE)
);
CPropertyPage(
   LPCTSTR lpszTemplateName,
   UINT nIDCaption,
   UINT nIDHeaderTitle,
   UINT nIDHeaderSubTitle = 0,
   DWORD dwSize = sizeof(PROPSHEETPAGE)
);
```

Parameters

nIDTemplate

ID of the template used for this page.

nIDCaption

ID of the name to be placed in the tab for this page. If 0, the name will be taken from the dialog template for this page.

dwSize

lpszTemplateName

Points to a string containing the name of the template for this page. Cannot be **NULL**.

nIDHeaderTitle

ID of the name to be placed in the title location of the property page header.

nIDHeaderSubTitle

ID of the name to be placed in the subtitle location of the property page header.

Remarks

The object is displayed after all of the following conditions are met:

- The page has been added to a property sheet using CPropertySheet::AddPage.
- The property sheet's DoModal or Create function has been called.
- The user has selected (tabbed to) this page.

If you have multiple parameters (for example, if you are using an array), use CPropertySheet::Construct instead of **CPropertyPage**.

Example

```
// Declare a CStylePage object, which is a CPropertyPage-derived class.
CStylePage stylePage;

// Declare a CPropertyPage object with IDD_SHAPE, the ID of the
// template used for this page.
CPropertyPage shapePage(IDD_SHAPE);
```

See Also

Reference

CPropertyPage Class
Hierarchy Chart
CPropertySheet::Create
CPropertySheet::DoModal
CPropertySheet::AddPage
CPropertyPage::Construct

Other Resources

CPropertyPage Members

CPropertyPage::GetPSP

Retrieves the Windows PROPSHEETPAGE structure associated with the **CPropertyPage** object.

```
const PROPSHEETPAGE & GetPSP( ) const;
PROPSHEETPAGE & GetPSP( );
```

Return Value

A reference to the **PROPSHEETPAGE** structure.

See Also

Reference

CPropertyPage Class

Hierarchy Chart

Other Resources

CPropertyPage Members

CPropertyPage::OnApply

This member function is called by the framework when the user chooses the OK or the Apply Now button.

```
virtual BOOL OnApply( );
```

Return Value

Nonzero if the changes are accepted; otherwise 0.

Remarks

When the framework calls this function, changes made on all property pages in the property sheet are accepted, the property sheet retains focus, and **OnApply** returns **TRUE** (the value 1). Before **OnApply** can be called by the framework, you must have called SetModified and set its parameter to **TRUE**. This will activate the Apply Now button as soon as the user makes a change on the property page.

Override this member function to specify what action your program takes when the user clicks the Apply Now button. When overriding, the function should return **TRUE** to accept changes and **FALSE** to prevent changes from taking effect.

The default implementation of **OnApply** calls **OnOK**.

For more information about notification messages sent when the user presses the Apply Now or OK button in a property sheet, see PSN_APPLY in the *Platform SDK*.

Example

See the example for CPropertyPage::OnOK.

See Also

Reference

CPropertyPage Class
Hierarchy Chart
CPropertyPage::SetModified
CPropertyPage::OnOK

Other Resources

CPropertyPage Members

CPropertyPage::OnCancel

This member function is called by the framework when the Cancel button is selected.

```
virtual void OnCancel( );
```

Remarks

Override this member function to perform Cancel button actions. The default negates any changes that have been made.

Example

```
// Discard any selection the user made to this page.  The object
// in the view will be painted with the initial color when the
// CPropertySheet dialog is first shown. CColorPage is a
// CPropertyPage-derived class.
void CColorPage::OnCancel()
{
   // Reset the color saved in the document class.  m_InitialColor
   // is a member variable of CColorPage and it is the color shown
   // in the view before CPropertySheet is shown.
   // doc->m_Color is the color saved in the document class, and
   // this is the color to be used by the view class.
   CFrameWnd* frame = (CFrameWnd*) AfxGetMainWnd();
   CPSheetDoc* doc = (CPSheetDoc*) frame->GetActiveDocument();
   doc->m_Color = m_InitialColor;

   // Tell the view to paint with the initial color.
   CView* view = frame->GetActiveView();
   view->Invalidate();

   CPropertyPage::OnCancel();
}

// The default MFC implementation of OnReset() would call OnCancel().
void CColorPage::OnReset()
{
   CPropertyPage::OnReset();
}
```

See Also

Reference

CPropertyPage Class
Hierarchy Chart
CPropertyPage::OnApply
CDialog::OnCancel
CPropertyPage::OnOK

Other Resources

CPropertyPage Members

CPropertyPage::OnKillActive

This member function is called by the framework when the page is no longer the active page.

```
virtual BOOL OnKillActive( );
```

Return Value

Nonzero if data was updated successfully, otherwise 0.

Remarks

Override this member function to perform special data validation tasks.

The default implementation of this member function copies settings from the controls in the property page to the member variables of the property page. If the data was not updated successfully due to a dialog data validation (DDV) error, the page retains focus.

After this member function returns successfully, the framework will call the page's OnOK function.

Example

```
// Validate the value entered to the "Number" edit control. Its
// value must be at least one. If not, tell the user and set the
// focus to the "Number" edit control. CStylePage is a
// CPropertyPage-derived class.
BOOL CStylePage::OnKillActive()
{
   int num = GetDlgItemInt(IDC_NUMOBJECTS);
   if (num <= 0)
   {
      AfxMessageBox("Number of objects must be at least 1.");
      CEdit* edit = (CEdit*) GetDlgItem(IDC_NUMOBJECTS);
      edit->SetFocus();
      edit->SetSel(0, -1);
      return 0;
   }

   return CPropertyPage::OnKillActive();
}
```

See Also

Reference

CPropertyPage Class
Hierarchy Chart
CWnd::UpdateData
CPropertyPage::OnOK
CPropertyPage::OnSetActive

Other Resources

CPropertyPage Members

CPropertyPage::OnOK

This member function is called by the framework when the user chooses either the OK or the Apply Now button, immediately after the framework calls OnKillActive.

```
virtual void OnOK( );
```

Remarks

When the user chooses either the OK or the Apply Now button, the framework receives the PSN_APPLY notification from the property page. The call to **OnOK** won't be made if you call CPropertySheet::PressButton because the property page does not send the notification in that case.

Override this member function to implement additional behavior specific to the currently active page when user dismisses the entire property sheet.

The default implementation of this member function marks the page as "clean" to reflect that the data was updated in the **OnKillActive** function.

Example

```
// Accept the new color selection and dismiss the CPropertySheet
// dialog. The view's object will be painted with the new selected
// color. CColorPage is a CPropertyPage-derived class.
void CColorPage::OnOK()
{
   // Store the new selected color to a member variable of
   // document class.  m_Color is a member varible of CColorPage
   // and it stores the new selected color.  doc->m_Color is
   // the color saved in the document class and it is the color
   // used by the view class.
   CFrameWnd* frame = (CFrameWnd*) AfxGetMainWnd();
   CPSheetDoc* doc = (CPSheetDoc*) frame->GetActiveDocument();
   doc->m_Color = m_Color;

   // Tell the view to paint with the new selected color.
   CView* view = frame->GetActiveView();
   view->Invalidate();

   CPropertyPage::OnOK();
}

// The default MFC implementation of OnApply() would call OnOK().
BOOL CColorPage::OnApply()
{
   return CPropertyPage::OnApply();
}
```

See Also

Reference

CPropertyPage Class
Hierarchy Chart
CDialog::OnOK
CPropertyPage::OnKillActive

Other Resources

CPropertyPage Members

CPropertyPage::OnQueryCancel

This member function is called by the framework when the user clicks the Cancel button and before the cancel action has taken place.

```
virtual BOOL OnQueryCancel( );
```

Return Value

Returns **FALSE** to prevent the cancel operation or TRUE to allow it.

Remarks

Override this member function to specify an action the program takes when the user clicks the Cancel button.

The default implementation of **OnQueryCancel** returns **TRUE**.

Example

```
// Query the user whether to abort the changes if the new selected
// color (m_Color) is different from the initial color
// (m_InitialColor) when the CPropertySheet dialog is first shown.
// CColorPage is a CPropertyPage-derived class.
BOOL CColorPage::OnQueryCancel()
{
   if (m_InitialColor != m_Color)
   {
      if (AfxMessageBox("Abort the changes?", MB_YESNO) == IDNO)
         return FALSE;
   }

   return CPropertyPage::OnQueryCancel();
}
```

See Also

Reference

CPropertyPage Class

Hierarchy Chart

Other Resources

CPropertyPage Members

CPropertyPage::OnReset

This member function is called by the framework when the user chooses the Cancel button.

```
virtual void OnReset( );
```

Remarks

When the framework calls this function, changes to all property pages that were made by the user previously choosing the Apply Now button are discarded, and the property sheet retains focus.

Override this member function to specify what action the program takes when the user clicks the Cancel button.

The default implementation of **OnReset** does nothing.

Example

See the example for CPropertyPage::OnCancel.

See Also

Reference

CPropertyPage Class
Hierarchy Chart
CPropertyPage::OnCancel
CPropertyPage::OnApply

Other Resources

CPropertyPage Members

CPropertyPage::OnSetActive

This member function is called by the framework when the page is chosen by the user and becomes the active page.

```
virtual BOOL OnSetActive( );
```

Return Value

Nonzero if the page was successfully set active; otherwise 0.

Remarks

Override this member function to perform tasks when a page is activated. Your override of this member function would typically call the default version after updating data members, to allow it to update the page controls with the new data.

The default implementation creates the window for the page, if not previously created, and makes it the active page.

Example

See the example for CPropertySheet::SetFinishText.

See Also

Reference

CPropertyPage Class
Hierarchy Chart
CPropertyPage::OnKillActive

Other Resources

CPropertyPage Members

CPropertyPage::OnWizardBack

This member function is called by the framework when the user clicks on the Back button in a wizard.

```
virtual LRESULT OnWizardBack();
```

Return Value

0 to automatically advance to the next page; –1 to prevent the page from changing. To jump to a page other than the next one, return the identifier of the dialog to be displayed.

Remarks

Override this member function to specify some action the user must take when the Back button is pressed.

For more information on how to make a wizard-type property sheet, see CPropertySheet::SetWizardMode.

Example

```
// The Back button is selected from the propertysheet.  Get the selected
// radio button of the page by looping through all buttons on the
// pages.  m_SelectedRadioButtonID is a member variable of
// CColorPage (a CPropertyPage-derived class).  Its initial value
// is initialized in OnInitDialog().
LRESULT CColorPage::OnWizardBack()
{
   for (int id = IDC_BLACK; id <= IDC_BLUE; id++)
   {
      CButton* button = (CButton*) GetDlgItem(id);
      if (button->GetCheck() == 1)
      {
         m_SelectedRadioButtonID = id;
         break;
      }
   }

   return CPropertyPage::OnWizardBack();
}
```

See Also

Reference

CPropertyPage Class
Hierarchy Chart
CPropertySheet::SetWizardMode

Other Resources

CPropertyPage Members

CPropertyPage::OnWizardFinish

This member function is called by the framework when the user clicks on the Finish button in a wizard.

```
virtual BOOL OnWizardFinish( );
```

Return Value

Nonzero if the property sheet is destroyed when the wizard finishes; otherwise zero.

Remarks

When a user clicks the **Finish** button in a wizard, the framework calls this function; when **OnWizardFinish** returns **TRUE** (a nonzero value), the property sheet is able to be destroyed (but is not actually destroyed). Call **DestroyWindow** to destroy the property sheet. Do not call **DestroyWindow** from **OnWizardFinish**; doing so will cause heap corruption or other errors.

You can override this member function to specify some action the user must take when the Finish button is pressed. When overriding this function, return **FALSE** to prevent the property sheet from being destroyed.

For more information about notification messages sent when the user presses the Finish button in a wizard property sheet, see PSN_WIZFINISH in the *Platform SDK*.

For more information on how to make a wizard-type property sheet, see CPropertySheet::SetWizardMode.

Example

```
// Inform users regarding the selections they have made by
// navigating the pages in propertysheet.
BOOL CColorPage::OnWizardFinish()
{
   CString report = "You have selected the following options:\n";

   // Get the number of property pages from CPropertySheet.
   CPropertySheet* sheet = (CPropertySheet*) GetParent();
   int count = sheet->GetPageCount();

   // Get the formatted string from each page. This formatted string
   // will be shown in a message box. Each page knows about the
   // string to be displayed. For simplicity, we derive a class
   // from CPropertyPage called CMyPropertyPage. CMyPropertyPage
   // has a pure virtual member function called GetPageSelections().
   // All pages in the property sheet must be derived from
   // CMyPropertyPage so we loop through each page to get the
   // formatted string by calling the GetPageSelections() function.
   for (int i = 0; i < count; i++)
   {
      CMyPropertyPage* page = (CMyPropertyPage*) sheet->GetPage(i);

      CString str;
      page->GetPageSelections(str);
      report += "\n" + str;
   }

   AfxMessageBox(report);

   return CPropertyPage::OnWizardFinish();
}

// An example of implementing the GetPageSelections() for CStylePage.
// CStylePage is a CMyProperty-derived class, which in turn is a
// CPropertyPage-derived class.
void CStylePage::GetPageSelections(CString &S)
{
   CString shapename;
   switch (m_Selection)
```

```
    {
    case IDC_RECTANGLE:
       shapename = "Rectangle";
       break;

    case IDC_ROUND_RECTANGLE:
       shapename = "Round Rectangle";
       break;

    case IDC_ELLIPSE:
       shapename = "Ellipse";
       break;
    }

    S.Format("Number of %s to be created = %d", shapename, m_NumObjects);
 }

 // An example of implementing the GetPageSelections() for CColorPage.
 // CColorPage is a CMyProperty-derived class, which in turn is a
 // CPropertyPage-derived class.
 void CColorPage::GetPageSelections(CString &S)
 {
    S = "Color selected is ";
    switch (m_Color)
    {
    case RGB(0, 0, 0):
       S += "Black";
       break;

    case RGB(255, 0, 0):
       S += "Red";
       break;

    case RGB(0, 255, 0):
       S += "Green";
       break;

    case RGB(0, 0, 255):
       S += "Blue";
       break;
    }
 }
```

See Also

Reference

CPropertyPage Class

Hierarchy Chart

CPropertySheet::SetWizardMode

Other Resources

CPropertyPage Members

CPropertyPage::OnWizardNext

This member function is called by the framework when the user clicks on the Next button in a wizard.

```
virtual LRESULT OnWizardNext();
```

Return Value

0 to automatically advance to the next page; –1 to prevent the page from changing. To jump to a page other than the next one, return the identifier of the dialog to be displayed.

Remarks

Override this member function to specify some action the user must take when the Next button is pressed.

For more information on how to make a wizard-type property sheet, see CPropertySheet::SetWizardMode.

Example

```
// The Next button is selected from the propertysheet. Show the
// second page of the propertysheet ONLY if a non-zero value is
// entered to the Number edit control of the CStylePage. Otherwise
// display a message to the user and stay on the current page.

LRESULT CStylePage::OnWizardNext()
{
   // Get the number from the edit control
   int num = GetDlgItemInt(IDC_NUMOBJECTS);

   if (num == 0)
   {
      // Get a pointer to the parent property sheet
      CPropertySheet* sheet = (CPropertySheet*)GetParent();

         // Display a message to the user
         ::MessageBox
            (
            sheet->m_hWnd,
            _T("Supply a non-zero number on the edit control"),
            _T("CStylePage Property Page"),
            MB_OK
            );

         // Stay on the current page
         return -1;
   }

   // CPropertyPage::OnWizardNext returns zero and causes
   // the property sheet to display the next page
   return CPropertyPage::OnWizardNext();
}
```

See Also

Reference

CPropertyPage Class

Hierarchy Chart

CPropertySheet::SetWizardMode

Other Resources

CPropertyPage Members

CPropertyPage::QuerySiblings

Call this member function to forward a message to each page in the property sheet.

```
LRESULT QuerySiblings(
   WPARAM wParam,
   LPARAM lParam
);
```

Parameters

wParam

Specifies additional message-dependent information.

lParam

Specifies additional message-dependent information

Return Value

The nonzero value from a page in the property sheet, or 0 if all pages return a value of 0.

Remarks

If a page returns a nonzero value, the property sheet does not send the message to subsequent pages.

Example

```
// Validate the value entered in the Number edit control. If its
// value is not a natural number, request CPropertySheet (i.e. parent
// window of the page) to send a PSM_QUERYSIBLINGS message to each
// LOADED page (a page won't be loaded in memory until it is shown).
// If one of the pages returns a nonzero value in response to the
// PSM_QUERYSIBLINGS message, then inform the user and change the OK
// to Close and disable the Cancel button. CStylePage is a
// CPropertyPage-derived class.
BOOL CStylePage::OnKillActive()
{
   int num = GetDlgItemInt(IDC_NUMOBJECTS);
   if (num <= 0)
   {
      if (QuerySiblings(num, 0L))
      {
         AfxMessageBox("An invalid data is entered.  Choose Close
            button to close the dialog.");
         CancelToClose();
      }
   }

   return CPropertyPage::OnKillActive();
}

// This is an example of trapping the PSM_QUERYSIBLINGS in one of
// the pages. CColorPage is a CPropertyPage-derived class.  Upon
// receiving this message, wParam contains the value passed to
// QuerySiblings() call. CColorPage will check this value and return
// FALSE only if the value is greater than 1.
BEGIN_MESSAGE_MAP(CColorPage, CPropertyPage)
   // ...
   ON_MESSAGE(PSM_QUERYSIBLINGS, OnQuerySiblings)
END_MESSAGE_MAP()

LRESULT CColorPage::OnQuerySiblings( WPARAM wParam, LPARAM lParam )
{
   return (wParam <= 0);
}
```

See Also

Reference

CPropertyPage Class

Hierarchy Chart

Other Resources

CPropertyPage Members

CPropertyPage::SetModified

Call this member function to enable or disable the Apply Now button, based on whether the settings in the property page should be applied to the appropriate external object.

```
void SetModified(
   BOOL bChanged = TRUE
);
```

Parameters

bChanged

TRUE to indicate that the property page settings have been modified since the last time they were applied; **FALSE** to indicate that the property page settings have been applied, or should be ignored.

Remarks

The framework keeps track of which pages are "dirty," that is, property pages for which you have called **SetModified(TRUE)**. The Apply Now button will always be enabled if you call **SetModified(TRUE)** for one of the pages. The Apply Now button will be disabled when you call **SetModified(FALSE)** for one of the pages, but only if none of the other pages is "dirty."

Example

```
// OnColorClicked() is a member function of CColorPage (a
// CPropertyPage-derived class). It is called whenever a radio button
// is selected on the page. Call SetModified() to enable the Apply
// button whenever a new selection is made. m_Color is a member
// variable of CColorPage and it is to store the selected RGB color.

BEGIN_MESSAGE_MAP(CColorPage, CPropertyPage)
   ON_CONTROL_RANGE(BN_CLICKED, IDC_BLACK, IDC_BLUE, OnColorClicked)
   //{{AFX_MSG_MAP(CColorPage)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

void CColorPage::OnColorClicked(UINT nCmdID)
{
   COLORREF color;
   switch (nCmdID)
   {
   case IDC_BLACK:
      color = RGB(0, 0, 0);
      break;

   case IDC_RED:
      color = RGB(255, 0, 0);
      break;

   case IDC_GREEN:
      color = RGB(0, 255, 0);
      break;

   case IDC_BLUE:
      color = RGB(0, 0, 255);
      break;
   }

   if (color != m_Color)
   {
      m_Color = color;
      SetModified();    // Enable Apply Now button.
   }
}
```

See Also

Reference

CPropertyPage Class

Hierarchy Chart

CPropertyPage::CancelToClose

Other Resources

CPropertyPage Members

CPropertyPage Data Members

For information about the data members in **CPropertyPage**, see CPropertyPage Members.

CPropertyPage::m_psp

m_psp is a structure whose members store the characteristics of PROPSHEETPAGE.

```
PROPSHEETPAGE m_psp;
```

Remarks

Use this structure to initialize the appearance of a property page after it is constructed.

For more information on this structure, including a listing of its members, see **PROPSHEETPAGE** in the *Platform SDK*.

Example

```
// Declare a CPropertySheet object.
CPropertySheet sheet("Simple PropertySheet");

// Change the settings of the two pages to enable property sheet's
// Help button when the page is active.  Both CStylePage and
// CColorPage are CPropertyPage-derived classes.
CStylePage style;
style.m_psp.dwFlags |= PSP_HASHELP;

CColorPage color;
color.m_psp.dwFlags  |= PSP_HASHELP;

sheet.AddPage(&style);
sheet.AddPage(&color);

// Display a modal CPropertySheet dialog.
sheet.DoModal();
```

See Also

Reference

CPropertyPage Class
Hierarchy Chart
CPropertySheet Class
PROPSHEETPAGE

Other Resources

CPropertyPage Members

CPropertySheet Class

Represents property sheets, otherwise known as tab dialog boxes.

```
class CPropertySheet : public CWnd
```

Remarks

A property sheet consists of a **CPropertySheet** object and one or more CPropertyPage objects. A property sheet is displayed by the framework as a window with a set of tab indices, with which the user selects the current page, and an area for the currently selected page.

CPropertySheet provides support for the expanded PROPSHEETHEADER structure introduced in Windows 98 and Windows NT 2000. The structure contains additional flags and members that support using a "watermark" background bitmap.

To display these new images automatically in your property sheet object, pass valid values for the bitmap and palette images in the call to CPropertySheet::Construct or CPropertySheet::CPropertySheet.

Even though **CPropertySheet** is not derived from CDialog, managing a **CPropertySheet** object is similar to managing a **CDialog** object. For example, creation of a property sheet requires two-part construction: call the constructor, and then call DoModal for a modal property sheet or Create for a modeless property sheet. **CPropertySheet** has two types of constructors: CPropertySheet::Construct and CPropertySheet::CPropertySheet.

Exchanging data between a **CPropertySheet** object and some external object is similar to exchanging data with a **CDialog** object. The important difference is that the settings of a property sheet are normally member variables of the **CPropertyPage** objects rather than of the **CPropertySheet** object itself.

You can create a type of tab dialog box called a wizard, which consists of a property sheet with a sequence of property pages that guide the user through the steps of an operation, such as setting up a device or creating a newsletter. In a wizard-type tab dialog box, the property pages do not have tabs, and only one property page is visible at a time. Also, instead of having OK and Apply Now buttons, a wizard-type tab dialog box has a Back button, a Next or Finish button, a Cancel button, and a Help button.

To create a wizard-type dialog box, follow the same steps you would follow to create a standard property sheet, but call SetWizardMode before you call DoModal. To enable the wizard buttons, call SetWizardButtons, using flags to customize their function and appearance. To enable the Finish button, call SetFinishText after the user has taken action on the last page of the wizard.

For more information on how to use **CPropertySheet** objects, see the article Property Sheets and Property Pages. Also, see Knowledge Base article Q146916 : HOWTO: Create a Modeless CPropertySheet with Standard Buttons and article Q300606 : HOWTO: Design a Resizable MFC Property Sheet.

Requirements

Header: afxdlgs.h

Smart Device Developer Notes

Not supported for Smartphone projects.

See Also

Tasks

CMNCTRL1 Sample: Demonstrates Common Control MFC Classes, Part 1
CMNCTRL2 Sample: Demonstrates Common Control MFC Classes, Part 2
PROPDLG Sample: Demonstrates Property Sheet Support
SNAPVW Sample: Uses Property Pages in a Form View Application

Reference

CWnd Class
Hierarchy Chart

Other Resources

CPropertySheet Members

CPropertySheet Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Data Members

m_psh	The Windows PROPSHEETHEADER structure. Provides access to basic property sheet parameters.

Construction

Construct	Constructs a **CPropertySheet** object.
CPropertySheet	Constructs a **CPropertySheet** object.

Attributes

EnableStackedTabs	Indicates whether the property sheet uses stacked or scrolling tabs.
GetActiveIndex	Retrieves the index of the active page of the property sheet.
GetActivePage	Returns the active page object.
GetPage	Retrieves a pointer to the specified page.
GetPageCount	Retrieves the number of pages in the property sheet.
GetPageIndex	Retrieves the index of the specified page of the property sheet.
GetTabControl	Retrieves a pointer to a tab control.
SetActivePage	Programmatically sets the active page object.
SetFinishText	Sets the text for the Finish button.
SetTitle	Sets the caption of the property sheet.
SetWizardButtons	Enables the wizard buttons.
SetWizardMode	Enables the wizard mode.

Operations

AddPage	Adds a page to the property sheet.
Create	Displays a modeless property sheet.
DoModal	Displays a modal property sheet.
EndDialog	Terminates the property sheet.
MapDialogRect	Converts the dialog-box units of a rectangle to screen units.

PressButton	Simulates the choice of the specified button in a property sheet.
RemovePage	Removes a page from the property sheet.

Overridables

OnInitDialog	Override to augment property sheet initialization.

See Also

Reference

CPropertySheet Class
Hierarchy Chart

CPropertySheet Member Functions

For information about the member functions in **CPropertySheet**, see CPropertySheet Members.

CPropertySheet::AddPage

Adds the supplied page with the rightmost tab in the property sheet.

```
void AddPage(
   CPropertyPage *pPage
);
```

Parameters

pPage

Points to the page to be added to the property sheet. Cannot be **NULL**.

Remarks

Add pages to the property sheet in the left-to-right order you want them to appear.

AddPage adds the CPropertyPage object to the **CPropertySheet** object's list of pages but does not actually create the window for the page. The framework postpones creation of the window for the page until the user selects that page.

When you add a property page using **AddPage**, the **CPropertySheet** is the parent of the **CPropertyPage**. To gain access to the property sheet from the property page, call CWnd::GetParent.

It is not necessary to wait until creation of the property sheet window to call **AddPage**. Typically, you will call **AddPage** before calling DoModal or Create.

If you call **AddPage** after displaying the property page, the tab row will reflect the newly added page.

Example

```
// Add two pages to a CPropertySheet object, then show the
// CPropertySheet object as a modal dialog.  Both CStylePage
// and CColorPage are CPropertyPage-derived classes created
// by the Add Class wizard.

CPropertySheet dlgPropertySheet("Simple PropertySheet");

CStylePage     stylePage;
CColorPage     colorPage;
dlgPropertySheet.AddPage(&stylePage);
dlgPropertySheet.AddPage(&colorPage);

dlgPropertySheet.DoModal();
```

See Also

Reference

CPropertySheet Class

Hierarchy Chart

CPropertySheet::RemovePage

Other Resources

CPropertySheet Members

CPropertySheet::Construct

Constructs a **CPropertySheet** object.

```
void Construct(
   UINT nIDCaption,
   CWnd* pParentWnd = NULL,
   UINT iSelectPage = 0
);
void Construct(
   LPCTSTR pszCaption,
   CWnd* pParentWnd = NULL,
   UINT iSelectPage = 0
);
void Construct(
   UINT nIDCaption,
   CWnd* pParentWnd,
   UINT iSelectPage,
   HBITMAP hbmWatermark,
   HPALETTE hpalWatermark = NULL,
   HBITMAP hbmHeader = NULL
);
void Construct(
   LPCTSTR pszCaption,
   CWnd* pParentWnd,
   UINT iSelectPage,
   HBITMAP hbmWatermark,
   HPALETTE hpalWatermark = NULL,
   HBITMAP hbmHeader = NULL
);
```

Parameters

nIDCaption

ID of the caption to be used for the property sheet.

pParentWnd

Pointer to the parent window of the property sheet. If **NULL**, the parent window will be the main window of the application.

iSelectPage

The index of the page that will initially be on top. Default is the first page added to the sheet.

pszCaption

Pointer to a string containing the caption to be used for the property sheet. Cannot be **NULL**.

hbmWatermark

Handle to the watermark bitmap of the property page.

hpalWatermark

Handle to the palette of the watermark bitmap and/or header bitmap.

hbmHeader

Handle to the header bitmap of the property page.

Remarks

Call this member function if one of the class constructors has not already been called. For example, call **Construct** when you declare or allocate arrays of **CPropertySheet** objects. In the case of arrays, you must call **Construct** for each member in the array.

To display the property sheet, call DoModal or Create. The string contained in the first parameter will be placed in the caption bar for the property sheet.

You can display watermark and/or header images automatically if you use the third or fourth prototypes of **Construct**, listed above, and you pass valid values for the *hbmWatermark*, *hpalWatermark*, and/or *hbmHeader* parameters.

Example

The following example demonstrates under what circumstances you would call **Construct**.

```
int i;
CPropertySheet   grpropsheet[4];
CPropertySheet   someSheet;   // no need to call Construct for this one

UINT rgID[4] = {IDD_SHEET1, IDD_SHEET2, IDD_SHEET3, IDD_SHEET4};

for (i = 0; i < 4; i++)
    grpropsheet[i].Construct(rgID[i]);
```

See Also

Reference

CPropertySheet Class
Hierarchy Chart
CPropertySheet::CPropertySheet
CPropertySheet::DoModal
CPropertySheet::Create

Other Resources

CPropertySheet Members

CPropertySheet::CPropertySheet

Constructs a **CPropertySheet** object.

```
CPropertySheet( );
explicit CPropertySheet(
   UINT nIDCaption,
   CWnd* pParentWnd = NULL,
   UINT iSelectPage = 0
);
explicit CPropertySheet(
   LPCTSTR pszCaption,
   CWnd* pParentWnd = NULL,
   UINT iSelectPage = 0
);
CPropertySheet(
   UINT nIDCaption,
   CWnd* pParentWnd,
   UINT iSelectPage,
   HBITMAP hbmWatermark,
   HPALETTE hpalWatermark = NULL,
   HBITMAP hbmHeader = NULL
);
CPropertySheet(
   LPCTSTR pszCaption,
   CWnd* pParentWnd,
   UINT iSelectPage,
   HBITMAP hbmWatermark,
   HPALETTE hpalWatermark = NULL,
   HBITMAP hbmHeader = NULL
);
```

Parameters

nIDCaption

ID of the caption to be used for the property sheet.

pParentWnd

Points to the parent window of the property sheet. If **NULL**, the parent window will be the main window of the application.

iSelectPage

The index of the page that will initially be on top. Default is the first page added to the sheet.

pszCaption

Points to a string containing the caption to be used for the property sheet. Cannot be **NULL**.

hbmWatermark

A handle to the background bitmap of the property sheet.

hpalWatermark

A handle to the palette of the watermark bitmap and/or header bitmap.

hbmHeader

A handle to the header bitmap of the property page.

Remarks

To display the property sheet, call DoModal or Create. The string contained in the first parameter will be placed in the caption bar for the property sheet.

If you have multiple parameters (for example, if you are using an array), use Construct instead of **CPropertySheet**.

You can display watermark and/or header images automatically if you use the third or fourth prototypes of **CPropertySheet**, above, and you pass valid values for the *hbmWatermark*, *hpalWatermark*, and/or *hbmHeader* parameters.

Example

```
// Declare a CPropertySheet object titled "Simple PropertySheet".
CPropertySheet dlgPropertySheet("Simple PropertySheet");

// Declare a CPropertySheet object whose title is specified in the
// IDS_PROPERTYSHEET_TITLE string resource, and the second page is
// initially on top.
CPropertySheet dlgPropertySheet(IDS_PROPERTYSHEET_TITLE, this, 1);
```

See Also

Reference

CPropertySheet Class
Hierarchy Chart
CPropertySheet::Construct
CPropertySheet::DoModal
CPropertySheet::Create
CPropertyPage::CPropertyPage

Other Resources

CPropertySheet Members

CPropertySheet::Create

Displays a modeless property sheet.

```
virtual BOOL Create(
   CWnd* pParentWnd = NULL,
   DWORD dwStyle = (DWORD)-1,
   DWORD dwExStyle = 0
);
```

Parameters

pParentWnd

Points to parent window. If **NULL**, parent is the desktop.

dwStyle

Window styles for property sheet. For a complete list of available styles, see Window Styles.

dwExStyle

Extended window styles for property sheet. For a complete list of available styles, see Extended Window Styles

Return Value

Nonzero if the property sheet is created successfully; otherwise 0.

Remarks

The call to **Create** can be inside the constructor, or you can call it after the constructor is invoked.

The default style, expressed by passing –1 as *dwStyle*, is actually **WS_SYSMENU | WS_POPUP | WS_CAPTION | DS_MODALFRAME | DS_CONTEXTHELP | WS_VISIBLE**. The default extended window style, expressed by passing 0 as *dwExStyle*, is actually **WS_EX_DLGMODALFRAME**.

The **Create** member function returns immediately after creating the property sheet. To destroy the property sheet, call CWnd::DestroyWindow.

Modeless property sheets displayed with a call to **Create** do not have OK, Cancel, Apply Now, and Help buttons as modal property sheets do. Desired buttons must be created by the user.

To display a modal property sheet, call DoModal instead.

Example

```
// This code fragment shows how to create a modeless property sheet
// dialog in a command message handler (OnModelessPropertySheet())
// of a CView-derived class.

void CMyView::OnModelessPropertySheet()
{
   // Declare a CPropertySheet object.  m_dlgPropertySheet is a data
   // member of type CPropertySheet in CView-derived class.
   m_dlgPropertySheet = new CPropertySheet("Simple PropertySheet");
   ASSERT(m_dlgPropertySheet);

   // Add two pages to the CPropertySheet object.  Both m_stylePage and
   // m_colorPage are data members of type CPropertyPage-derived classes
   // in CView-derived class.
   m_stylePage = new CStylePage;
   m_colorPage = new CColorPage;
   m_dlgPropertySheet->AddPage(m_stylePage);
   m_dlgPropertySheet->AddPage(m_colorPage);

   // Create a modeless CPropertySheet dialog.
   m_dlgPropertySheet->Create();
}
```

```
// The code fragment below shows how to destroy the C++ objects for
// propertysheet and propertypage in the destructor of CView-derived
// class.
// NOTE:  DestroyWindow() is called in CPropertySheet::OnClose() so
// you do not need to call it here.  Property pages are children
// of the CPropertySheet, they will be destroyed by their parents.

CMyView::~CMyView()
{
   if (m_dlgPropertySheet)
   {
      delete m_stylePage;
      delete m_colorPage;
      delete m_dlgPropertySheet;
   }
}
```

See Also

Reference

CPropertySheet Class

Hierarchy Chart

CDialog::Create

CPropertySheet::DoModal

Other Resources

CPropertySheet Members

CPropertySheet::DoModal

Displays a modal property sheet.

```
virtual INT_PTR DoModal( );
```

Return Value

IDOK or **IDCANCEL** if the function was successful; otherwise 0 or -1. If the property sheet has been established as a wizard (see SetWizardMode), **DoModal** returns either **ID_WIZFINISH** or **IDCANCEL**.

Remarks

The return value corresponds to the ID of the control that closed the property sheet. After this function returns, the windows corresponding to the property sheet and all the pages will have been destroyed. The objects themselves will still exist. Typically, you will retrieve data from the CPropertyPage objects after **DoModal** returns **IDOK**.

To display a modeless property sheet, call Create instead.

Note
The first time a property page is created from its corresponding dialog resource, it may cause a first-chance exception. This is a result of the property page changing the style of the dialog resource to the required style prior to creating the page. Because resources are generally read-only, this causes an exception. The exception is handled by the system, and a copy of the modified resource is made automatically by the system. The first-chance exception can thus be ignored.

Note
Because this exception must be handled by the operating system, do not wrap calls to **CPropertySheet::DoModal** with a C++ **try**/**catch** block in which the catch handles all exceptions, for example, `catch (...)`. This will handle the exception intended for the operating system, causing unpredictable behavior. Using C++ exception handling with specific exception types or using structured exception handling where the Access Violation exception is passed through to the operating system is safe, however.

Example

See the example for CPropertySheet::AddPage.

See Also

Reference

CPropertySheet Class
Hierarchy Chart
CDialog::DoModal
CPropertySheet::Create

Other Resources

CPropertySheet Members

CPropertySheet::EnableStackedTabs

Indicates whether to stack rows of tabs in a property sheet.

```
void EnableStackedTabs(
   BOOL bStacked
);
```

Parameters

bStacked

Indicates whether stacked tabs are enabled in the property sheet. Disable stacked rows of tags by setting *bStacked* to **FALSE**.

Remarks

By default, if a property sheet has more tabs than will fit in a single row in the width of the property sheet, the tabs will stack in multiple rows. To use scrolling tabs instead of stacking tabs, call **EnableStackedTabs** with *bStacked* set to **FALSE** before calling DoModal or Create.

You must call **EnableStackedTabs** when you create a modal or a modeless property sheet. To incorporate this style in a **CPropertySheet**-derived class, write a message handler for **WM_CREATE**. In the overridden version of CWnd::OnCreate, call **EnableStackedTabs(FALSE)** before calling the base class implementation.

Example

```
int CMyPropertySheet::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
      // Set for Scrolling Tabs style
      EnableStackedTabs(FALSE);
      // Call the base class
      if (CPropertySheet::OnCreate(lpCreateStruct) == -1)
            return -1;
      // TODO: Add your specialized creation code here
      return 0;
}
```

See Also

Reference

CPropertySheet Class

Hierarchy Chart

Other Resources

CPropertySheet Members

CPropertySheet::EndDialog

Terminates the property sheet.

```
void EndDialog(
   int nEndID
);
```

Parameters

nEndID

Identifier to be used as return value of the property sheet.

Remarks

This member function is called by the framework when the OK, Cancel, or Close button is pressed. Call this member function if an event occurs that should close the property sheet.

This member function is only used with a modal dialog box.

Example

See the example for CPropertySheet::PressButton.

See Also

Reference

CPropertySheet Class
Hierarchy Chart
CPropertyPage::OnOK
CPropertyPage::OnCancel
CWnd::DestroyWindow

Other Resources

CPropertySheet Members

CPropertySheet::GetActiveIndex

Gets the index number of the property sheet window's active page and then uses the returned index number as the parameter for **GetPage**.

```
int GetActiveIndex( ) const;
```

Return Value

The index number of the active page.

Example

See the example for CPropertySheet::GetActivePage.

See Also

Reference

CPropertySheet Class
Hierarchy Chart
CPropertySheet::GetPage
CPropertySheet::GetActivePage

Other Resources

CPropertySheet Members

CPropertySheet::GetActivePage

Retrieves the property sheet window's active page.

```
CPropertyPage* GetActivePage( ) const;
```

Return Value

The pointer to the active page.

Remarks

Use this member function to perform some action on the active page.

Example

```
// The code fragment below sets the last active page (i.e. the
// active page when the propertysheet was closed) to be the first
// visible page when the propertysheet is shown. The last active
// page was saved in m_LastActivePage, (a member variable of
// CDocument-derived class) when OK was selected from the
// propertysheet. CMyPropertySheet is a CPropertySheet-derived class.
BOOL CMyPropertySheet::OnInitDialog()
{
   BOOL bResult = CPropertySheet::OnInitDialog();

   CFrameWnd* frame = (CFrameWnd*) AfxGetMainWnd();
   CPSheetDoc* doc = (CPSheetDoc*) frame->GetActiveDocument();
   SetActivePage(doc->m_LastActivePage);
   return bResult;
}

BOOL CMyPropertySheet::OnCommand(WPARAM wParam, LPARAM lParam)
{
   if (LOWORD(wParam) == IDOK)
   {
      CFrameWnd* frame = (CFrameWnd*) AfxGetMainWnd();
      CPSheetDoc* doc = (CPSheetDoc*) frame->GetActiveDocument();
      doc->m_LastActivePage = GetPageIndex(GetActivePage());
         // or GetActiveIndex()
   }

   return CPropertySheet::OnCommand(wParam, lParam);
}
```

See Also

Reference

CPropertySheet Class

Hierarchy Chart

CPropertySheet::GetPage

Other Resources

CPropertySheet Members

CPropertySheet::GetPage

Returns a pointer to the specified page in this property sheet.

```
CPropertyPage* GetPage(
   int nPage
) const;
```

Parameters

nPage

Index of the desired page, starting at 0. Must be between 0 and one less than the number of pages in the property sheet, inclusive.

Return Value

The pointer to the page corresponding to the *nPage* parameter.

Example

See the example for CPropertyPage::OnWizardFinish.

See Also

Reference

CPropertySheet Class
Hierarchy Chart
CPropertySheet::AddPage
CPropertySheet::GetActivePage
CPropertySheet::GetPageCount
CPropertySheet::RemovePage
CPropertySheet::SetTitle

Other Resources

CPropertySheet Members

CPropertySheet::GetPageCount

Determines the number of pages currently in the property sheet.

```
int GetPageCount( ) const;
```

Return Value

The number of pages in the property sheet.

Example

See the example for CPropertyPage::OnWizardFinish.

See Also

Reference

CPropertySheet Class
Hierarchy Chart
CPropertySheet::GetPage
CPropertySheet::AddPage
CPropertySheet::RemovePage

Other Resources

CPropertySheet Members

CPropertySheet::GetPageIndex

Retrieves the index number of the specified page in the property sheet.

```
int GetPageIndex(
   CPropertyPage* pPage
);
```

Parameters

pPage

Points to the page with the index to be found. Cannot be **NULL**.

Return Value

The index number of a page.

Remarks

For example, you would use **GetPageIndex** to get the page index in order to use SetActivePage or GetPage.

Example

See the example for CPropertySheet::GetActivePage.

See Also

Reference

CPropertySheet Class
Hierarchy Chart
CPropertySheet::SetActivePage
CPropertySheet::GetPage

Other Resources

CPropertySheet Members

CPropertySheet::GetTabControl

Retrieves a pointer to a tab control to do something specific to the tab control (that is, to use any of the APIs in CTabCtrl).

```
CTabCtrl* GetTabControl( ) const;
```

Return Value

A pointer to a tab control.

Remarks

For example, call this member function if you want to add bitmaps to each of the tabs during initialization.

Example

```
// Create and associate a tooltip control to the tab control of
// CMyPropertySheet.  CMyPropertySheet is a CPropertySheet-derived
// class.
BOOL CMyPropertySheet::OnInitDialog()
{
   BOOL bResult = CPropertySheet::OnInitDialog();

   // Create a tooltip control.  m_ToolTipCtrl is a member variable
   // of type CToolTipCtrl* in CMyPropertySheet class.  It is
   // initialized to NULL in the constructor, and destroyed in the
   // destructor of CMyPropertySheet class.
   m_ToolTipCtrl = new CToolTipCtrl;
   if (!m_ToolTipCtrl->Create(this))
   {
      TRACE("Unable To create ToolTip\n");
      return bResult;
   }

   // Associate the tooltip control to the tab control
   // of CMyPropertySheet.
   CTabCtrl* tab = GetTabControl();
   tab->SetToolTips(m_ToolTipCtrl);

   // Get the bounding rectangle of each tab in the tab control of the
   // property sheet. Use this rectangle when registering a tool with
   // the tool tip control.  IDS_FIRST_TOOLTIP is the first ID string
   // resource that contains the text for the tool.
   int count = tab->GetItemCount();
   int id = IDS_FIRST_TOOLTIP;
   for (int i = 0; i < count; i++)
   {
      CRect r;
      tab->GetItemRect(i, &r);
      VERIFY(m_ToolTipCtrl->AddTool(tab, id, &r, id));
      id++;
   }

   // Activate the tooltip control.
   m_ToolTipCtrl->Activate(TRUE);

   return bResult;
}

// Override PreTranslateMessage() so RelayEvent() can be
// called to pass a mouse message to CMyPropertySheet's
// tooltip control for processing.
BOOL CMyPropertySheet::PreTranslateMessage(MSG* pMsg)
{
   if (NULL != m_ToolTipCtrl)
```

```
        m_ToolTipCtrl->RelayEvent(pMsg);

    return CPropertySheet::PreTranslateMessage(pMsg);
}
```

See Also

Reference

CPropertySheet Class

Hierarchy Chart

CTabCtrl::CTabCtrl

Other Resources

CPropertySheet Members

CPropertySheet::MapDialogRect

Converts the dialog-box units of a rectangle to screen units.

```
void MapDialogRect(
   LPRECT lpRect
) const;
```

Parameters

lpRect

Points to a RECT structure or CRect object that contains the dialog-box coordinates to be converted.

Remarks

Dialog-box units are stated in terms of the current dialog-box base unit derived from the average width and height of characters in the font used for dialog-box text. One horizontal unit is one-fourth of the dialog-box base-width unit, and one vertical unit is one-eighth of the dialog-box base height unit.

The GetDialogBaseUnits Windows function returns size information for the system font, but you can specify a different font for each property sheet if you use the **DS_SETFONT** style in the resource-definition file. The MapDialogRect Windows function, described in the *Platform SDK*, uses the appropriate font for this dialog box.

The **MapDialogRect** member function replaces the dialog-box units in *lpRect* with screen units (pixels) so that the rectangle can be used to create a dialog box or position a control within a box.

See Also

Reference

CPropertySheet Class
Hierarchy Chart
WM_SETFONT

Other Resources

CPropertySheet Members

CPropertySheet::OnInitDialog

Overrides to augment property sheet initialization.

```
virtual BOOL OnInitDialog( );
```

Return Value

Specifies whether the application has set the input focus to one of the controls in the property sheet. If **OnInitDialog** returns nonzero, Windows sets the input focus to the first control in the property sheet. The application can return 0 only if it has explicitly set the input focus to one of the controls in the property sheet.

Remarks

This member function is called in response to the **WM_INITDIALOG** message. This message is sent to the property sheet during the Create or DoModal calls, which occur immediately before the property sheet is displayed.

Override this member function if you need to perform special processing when the property sheet is initialized. In the overridden version, first call the base class **OnInitDialog** but disregard its return value. You will normally return **TRUE** from your overridden member function.

You do not need a message-map entry for this member function.

See Also

Reference

CPropertySheet Class
Hierarchy Chart
CDialog::OnInitDialog

Other Resources

CPropertySheet Members

CPropertySheet::PressButton

Simulates the choice of the specified button in a property sheet.

```
void PressButton(
   int nButton
);
```

Parameters

nButton

nButton : Identifies the button to be pressed. This parameter can be one of the following values:

- **PSBTN_BACK** Chooses the Back button.
- **PSBTN_NEXT** Chooses the Next button.
- **PSBTN_FINISH** Chooses the Finish button.
- **PSBTN_OK** Chooses the OK button.
- **PSBTN_APPLYNOW** Chooses the Apply Now button.
- **PSBTN_CANCEL** Chooses the Cancel button.
- **PSBTN_HELP** Chooses the Help button.

Remarks

See PSM_PRESSBUTTON for more information about the Windows SDK Pressbutton message.

A call to **PressButton** will not send the PSN_APPLY notification from a property page to the framework. To send this notification, call CPropertyPage::OnOK.

Example

```
// Simulate the selection of OK and Cancel buttons when Alt+K and
// Alt+C are pressed.  CMyPropertySheet is a CPropertySheet-derived
// class.
BOOL CMyPropertySheet::PreTranslateMessage(MSG* pMsg)
{
   if (pMsg->message >= WM_KEYFIRST && pMsg->message <= WM_KEYLAST)
   {
      BOOL altkey = GetKeyState(VK_MENU) < 0;
      if (altkey)
      {
         BOOL handled = TRUE;
         switch(toupper(pMsg->wParam))
         {
         case 'C':                     // for Alt+C - Cancel button
            PressButton(PSBTN_CANCEL);   // or EndDialog(IDCANCEL);
            break;

         case 'K':                     // for Alt+K - OK button
            PressButton(PSBTN_OK);      // or EndDialog(IDOK);
            break;

         default:
            handled = FALSE;
         }

         if (handled)
            return TRUE;
      }
   }
```

```
    return CPropertySheet::PreTranslateMessage(pMsg);
}
```

See Also

Reference

CPropertySheet Class

Hierarchy Chart

Other Resources

CPropertySheet Members

CPropertySheet::RemovePage

Removes a page from the property sheet and destroys the associated window.

```
void RemovePage(
   CPropertyPage *pPage
);
void RemovePage(
   int nPage
);
```

Parameters

pPage

Points to the page to be removed from the property sheet. Cannot be **NULL**.

nPage

Index of the page to be removed. Must be between 0 and one less than the number of pages in the property sheet, inclusive.

Remarks

The CPropertyPage object itself is not destroyed until the owner of the **CPropertySheet** window is closed.

Example

See the example for CPropertyPage::OnWizardNext.

See Also

Reference

CPropertySheet Class
Hierarchy Chart
CPropertySheet::AddPage

Other Resources

CPropertySheet Members

CPropertySheet::SetActivePage

Changes the active page.

```
BOOL SetActivePage(
   int nPage
);
BOOL SetActivePage(
   CPropertyPage* pPage
);
```

Parameters

nPage

Index of the page to set. It must be between 0 and one less than the number of pages in the property sheet, inclusive.

pPage

Points to the page to set in the property sheet. It cannot be **NULL**.

Return Value

Nonzero if the property sheet is activated successfully; otherwise 0.

Remarks

For example, use **SetActivePage** if a user's action on one page should cause another page to become the active page.

Example

See the example for CPropertySheet::GetActivePage.

See Also

Reference

CPropertySheet Class

Hierarchy Chart

Other Resources

CPropertySheet Members

CPropertySheet::SetFinishText

Sets the text in the Finish command button.

```
void SetFinishText(
   LPCTSTR lpszText
);
```

Parameters

lpszText

Points to the text to be displayed on the Finish command button.

Remarks

Call **SetFinishText** to display the text on the Finish command button and hide the Next and Back buttons after the user completes action on the last page of the wizard.

Example

```
// CShapePage is the last page of the wizard.  The "Finish" button
// will have "Done" as its caption, and both "Back" and "Next"
// buttons are hidden.
BOOL CShapePage::OnSetActive()
{
   CPropertySheet* psheet = (CPropertySheet*) GetParent();
   psheet->SetWizardButtons(PSWIZB_FINISH);
   psheet->SetFinishText("Done");

   return CPropertyPage::OnSetActive();
}
```

See Also

Reference

CPropertySheet Class

Hierarchy Chart

Other Resources

CPropertySheet Members

CPropertySheet::SetTitle

Specifies the property sheet's caption (the text displayed in the title bar of a frame window).

```
void SetTitle(
   LPCTSTR lpszText,
   UINT nStyle = 0
);
```

Parameters

nStyle

Specifies the style of the property sheet title. The style must be specified at 0 or as **PSH_PROPTITLE**. If the style is set as **PSH_PROPTITLE**, the word "Properties" appears after the text specified as the caption. For example, calling **SetTitle**("Simple", **PSH_PROPTITLE**) will result in a property sheet caption of "Simple Properties."

lpszText

Points to the text to be used as the caption in the title bar of the property sheet.

Remarks

By default, a property sheet uses the caption parameter in the property sheet constructor.

Example

```
// Declare a CPropertySheet object with a caption "Simple PropertySheet".
CPropertySheet dlgPropertySheet("Simple PropertySheet");

// Add two pages to the CPropertySheet object. Both CStylePage and
// CColorPage are CPropertyPage-derived classes created
// by the Add Class wizard.
CStylePage     stylePage;
CColorPage     colorPage;
dlgPropertySheet.AddPage(&stylePage);
dlgPropertySheet.AddPage(&colorPage);

// Change the caption of the CPropertySheet object
// from "Simple PropertySheet" to "Simple Properties".
dlgPropertySheet.SetTitle("Simple", PSH_PROPTITLE);

// Show the CPropertySheet object as MODAL.
dlgPropertySheet.DoModal();
```

See Also

Reference

CPropertySheet Class
Hierarchy Chart
CPropertySheet::GetPage
CPropertySheet::GetActivePage

Other Resources

CPropertySheet Members

CPropertySheet::SetWizardButtons

Enables or disables the Back, Next, or Finish button in a wizard property sheet.

```
void SetWizardButtons(
   DWORD dwFlags
);
```

Parameters

dwFlags

A set of flags that customize the function and appearance of the wizard buttons. This parameter can be a combination of the following values:

- **PSWIZB_BACK** Back button
- **PSWIZB_NEXT** Next button
- **PSWIZB_FINISH** Finish button
- **PSWIZB_DISABLEDFINISH** Disabled Finish button

Remarks

Call **SetWizardButtons** only after the dialog is open; you can't call **SetWizardButtons** before you call DoModal. Typically, you should call **SetWizardButtons** from CPropertyPage::OnSetActive.

If you want to change the text on the Finish button or hide the Next and Back buttons once the user has completed the wizard, call SetFinishText. Note that the same button is shared for Finish and Next. You can display a Finish or a Next button at one time, but not both.

Example

```
// A CPropertySheet has three wizard property pages: CStylePage,
// CColorPage, and CShapePage.  The code fragment below shows how to
// enable and disable the Back and Next buttons on the wizard
// property page.

// CStylePage is the first wizard property page.  Disable the Back
// button but enable the Next button.
BOOL CStylePage::OnSetActive()
{
   CPropertySheet* psheet = (CPropertySheet*) GetParent();
   psheet->SetWizardButtons(PSWIZB_NEXT);

   return CPropertyPage::OnSetActive();
}

// CColorPage is the second wizard property page. Enable both the
// Back button and the Next button.
BOOL CColorPage::OnSetActive()
{
   CPropertySheet* psheet = (CPropertySheet*) GetParent();
   psheet->SetWizardButtons(PSWIZB_BACK | PSWIZB_NEXT);

   return CPropertyPage::OnSetActive();
}

// CShapePage is the third wizard property page. Enable the Back
// button and change the Next button to Finish. If dwFlags is equal
// to PSWIZB_BACK | PSWIZB_DISABLEDFINISH, then the Back
// button is enabled but the Next button is disabled.
BOOL CShapePage::OnSetActive()
{
   CPropertySheet* psheet = (CPropertySheet*) GetParent();
```

```
    psheet->SetWizardButtons(PSWIZB_BACK | PSWIZB_FINISH);

    return CPropertyPage::OnSetActive();
}
```

See Also

Reference

CPropertySheet Class

Hierarchy Chart

Other Resources

CPropertySheet Members

CPropertySheet::SetWizardMode

Establishes a property page as a wizard.

```
void SetWizardMode( );
```

Remarks

A key characteristic of a wizard property page is that the user navigates using Next or Finish, Back, and Cancel buttons instead of tabs.

Call **SetWizardMode** before calling DoModal. After you call **SetWizardMode**, **DoModal** will return either **ID_WIZFINISH** (if the user closes with the Finish button) or **IDCANCEL**.

SetWizardMode sets the PSH_WIZARD flag.

Example

```
CPropertySheet dlg;
CPropertyPage page1, page2;

dlg.AddPage(&page1);
dlg.AddPage(&page2);
dlg.SetWizardMode();
dlg.DoModal();
```

See Also

Reference

CPropertySheet Class
Hierarchy Chart
CPropertySheet::DoModal

Other Resources

CPropertySheet Members

CPropertySheet Data Members

For information about the data members in **CPropertySheet**, see CPropertySheet Members.

CPropertySheet::m_psh

A structure whose members store the characteristics of PROPSHEETHEADER.

Remarks

Use this structure to initialize the appearance of the property sheet after it is constructed but before it is displayed with the DoModal member function. For example, set the *dwSize* member of **m_psh** to the size you want the property sheet to have.

For more information on this structure, including a listing of its members, see **PROPSHEETHEADER** in the *Platform SDK*.

Example

```
// This code fragment shows how to change CPropertySheet's settings
// before it is shown.  After the changes, CPropertySheet has the
// caption "Simple Properties", no "Apply" button, and the
// second page (CColorPage) initially on top.

CPropertySheet dlgPropertySheet("Simple PropertySheet");

CStylePage stylePage;
CColorPage colorPage;
dlgPropertySheet.AddPage(&stylePage);
dlgPropertySheet.AddPage(&colorPage);

dlgPropertySheet.m_psh.dwFlags |= PSH_NOAPPLYNOW | PSH_PROPTITLE;
dlgPropertySheet.m_psh.pszCaption = "Simple";
dlgPropertySheet.m_psh.nStartPage = 1;

dlgPropertySheet.DoModal();
```

See Also

Reference

CPropertySheet Class
Hierarchy Chart
CPropertySheet::DoModal

Other Resources

CPropertySheet Members

CPropExchange Class

Supports the implementation of persistence for your OLE controls.

```
class AFX_NOVTABLE CPropExchange
```

Remarks

CPropExchange does not have a base class.

Establishes the context and direction of a property exchange.

Persistence is the exchange of the control's state information, usually represented by its properties, between the control itself and a medium.

The framework constructs an object derived from **CPropExchange** when it is notified that an OLE control's properties are to be loaded from or stored to persistent storage.

The framework passes a pointer to this **CPropExchange** object to your control's `DoPropExchange` function. If you used a wizard to create the starter files for your control, your control's `DoPropExchange` function calls **COleControl::DoPropExchange**. The base-class version exchanges the control's stock properties; you modify your derived class's version to exchange properties you have added to your control.

CPropExchange can be used to serialize a control's properties or initialize a control's properties upon the load or creation of a control. The **ExchangeProp** and **ExchangeFontProp** member functions of **CPropExchange** are able to store properties to and load them from different media.

For more information on using **CPropExchange**, see the article MFC ActiveX Controls: Property Pages.

Requirements

Header: afxctl.h

See Also

Reference

Hierarchy Chart

COleControl::DoPropExchange

Other Resources

CPropExchange Members

CPropExchange Members

Operations

ExchangeBlobProp	Exchanges a binary large object (BLOB) property.
ExchangeFontProp	Exchanges a font property.
ExchangePersistentProp	Exchanges a property between a control and a file.
ExchangeProp	Exchanges properties of any built-in type.
ExchangeVersion	Exchanges the version number of an OLE control.
GetVersion	Retrieves the version number of an OLE control.
IsAsynchronous	Determines if property exchanges are done asynchronously.
IsLoading	Indicates whether properties are being loaded into the control or saved from it.

See Also

Reference

CPropExchange Class
Hierarchy Chart

CPropExchange Member Functions

For information about the member functions in **CPropExchange**, see CPropExchange Members.

CPropExchange::ExchangeBlobProp

Serializes a property that stores binary large object (BLOB) data.

```
virtual BOOL ExchangeBlobProp(
   LPCTSTR pszPropName,
   HGLOBAL* phBlob,
   HGLOBAL hBlobDefault = NULL
) = 0;
```

Parameters

pszPropName

The name of the property being exchanged.

phBlob

Pointer to a variable pointing to where the property is stored (variable is typically a member of your class).

hBlobDefault

Default value for the property.

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

The property's value is read from or written to, as appropriate, the variable referenced by *phBlob*. If *hBlobDefault* is specified, it will be used as the property's default value. This value is used if, for any reason, the control's serialization fails.

The functions **CArchivePropExchange::ExchangeBlobProp**, **CResetPropExchange::ExchangeBlobProp**, and **CPropsetPropExchange::ExchangeBlobProp** override this pure virtual function.

See Also

Reference

CPropExchange Class
Hierarchy Chart
COleControl::DoPropExchange
CPropExchange::ExchangeFontProp
CPropExchange::ExchangePersistentProp
CPropExchange::ExchangeProp

Other Resources

CPropExchange Members

CPropExchange::ExchangeFontProp

Exchanges a font property between a storage medium and the control.

```
virtual BOOL ExchangeFontProp(
   LPCTSTR pszPropName,
   CFontHolder& font,
   const FONTDESC* pFontDesc,
   LPFONTDISP pFontDispAmbient
) = 0;
```

Parameters

pszPropName

The name of the property being exchanged.

font

A reference to a CFontHolder object that contains the font property.

pFontDesc

A pointer to a FONTDESC structure containing values for initializing the default state of the font property when *pFontDispAmbient* is **NULL**.

pFontDispAmbient

A pointer to the **IFontDisp** interface of a font to be used for initializing the default state of the font property.

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

If the font property is being loaded from the medium to the control, the font's characteristics are retrieved from the medium and the **CFontHolder** object referenced by *font* is initialized with them. If the font property is being stored, the characteristics in the font object are written to the medium.

The functions **CArchivePropExchange::ExchangeFontProp**, **CResetPropExchange::ExchangeFontProp**, and **CPropsetPropExchange::ExchangeFontProp** override this pure virtual function.

See Also

Reference

CPropExchange Class
Hierarchy Chart
COleControl::DoPropExchange
CPropExchange::ExchangeBlobProp
CPropExchange::ExchangePersistentProp
CPropExchange::ExchangeProp

Other Resources

CPropExchange Members

CPropExchange::ExchangePersistentProp

Exchanges a property between the control and a file.

```
virtual BOOL ExchangePersistentProp(
   LPCTSTR pszPropName,
   LPUNKNOWN* ppUnk,
   REFIID iid,
   LPUNKNOWN pUnkDefault
) = 0;
```

Parameters

pszPropName

The name of the property being exchanged.

ppUnk

A pointer to a variable containing a pointer to the property's **IUnknown** interface (this variable is typically a member of your class).

iid

Interface ID of the interface on the property that the control will use.

pUnkDefault

Default value for the property.

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

If the property is being loaded from the file to the control, the property is created and initialized from the file. If the property is being stored, its value is written to the file.

The functions **CArchivePropExchange::ExchangePersistentProp**, **CResetPropExchange::ExchangePersistentProp**, and **CPropsetPropExchange::ExchangePersistentProp** override this pure virtual function.

See Also

Reference

CPropExchange Class
Hierarchy Chart
COleControl::DoPropExchange
CPropExchange::ExchangeBlobProp
CPropExchange::ExchangeFontProp
CPropExchange::ExchangeProp

Other Resources

CPropExchange Members

CPropExchange::ExchangeProp

Exchanges a property between a storage medium and the control.

```
virtual BOOL ExchangeProp(
   LPCTSTR pszPropName,
   VARTYPE vtProp,
   void* pvProp,
   const void* pvDefault = NULL
) = 0;
```

Parameters

pszPropName

The name of the property being exchanged.

vtProp

A symbol specifying the type of the property being exchanged. Possible values are:

Symbol	Property Type
VT_I2	**short**
VT_I4	**long**
VT_BOOL	**BOOL**
VT_BSTR	**CString**
VT_CY	**CY**
VT_R4	**float**
VT_R8	**double**

pvProp

A pointer to the property's value.

pvDefault

Pointer to a default value for the property.

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

If the property is being loaded from the medium to the control, the property's value is retrieved from the medium and stored in the object pointed to by *pvProp*. If the property is being stored to the medium, the value of the object pointed to by *pvProp* is written to the medium.

The functions **CArchivePropExchange::ExchangeProp**, **CResetPropExchange::ExchangeProp**, and **CPropsetPropExchange::ExchangeProp** override this pure virtual function.

See Also

Reference

CPropExchange Class
Hierarchy Chart
COleControl::DoPropExchange
CPropExchange::ExchangeBlobProp

CPropExchange::ExchangeFontProp
CPropExchange::ExchangePersistentProp

Other Resources

CPropExchange Members

CPropExchange::ExchangeVersion

Called by the framework to handle persistence of a version number.

```
virtual BOOL ExchangeVersion(
   DWORD& dwVersionLoaded,
   DWORD dwVersionDefault,
   BOOL bConvert
);
```

Parameters

dwVersionLoaded

Reference to a variable where the version number of the persistent data being loaded will be stored.

dwVersionDefault

The current version number of the control.

bConvert

Indicates whether to convert persistent data to the current version or keep it at the same version that was loaded.

Return Value

Nonzero if the function succeeded; 0 otherwise.

See Also

Reference

CPropExchange Class
Hierarchy Chart
COleControl::ExchangeVersion

Other Resources

CPropExchange Members

CPropExchange::GetVersion

Call this function to retrieve the version number of the control.

```
DWORD GetVersion( );
```

Return Value

The version number of the control.

See Also

Reference

CPropExchange Class

Hierarchy Chart

Other Resources

CPropExchange Members

CPropExchange::IsAsynchronous

Determines if property exchanges are done asynchronously.

```
BOOL IsAsynchronous( );
```

Return Value

Returns TRUE if properties are exchanged asynchronously, otherwise FALSE.

See Also

Reference

CPropExchange Class

Hierarchy Chart

Other Resources

CPropExchange Members

CPropExchange::IsLoading

Call this function to determine whether properties are being loaded to the control or saved from it.

```
BOOL IsLoading( );
```

Return Value

Nonzero if properties are being loaded; otherwise 0.

See Also

Reference

CPropExchange Class
Hierarchy Chart
COleControl::DoPropExchange

Other Resources

CPropExchange Members

CPtrArray Class

Supports arrays of void pointers.

```
class CPtrArray : public CObject
```

Remarks

The member functions of **CPtrArray** are similar to the member functions of class CObArray. Because of this similarity, you can use the **CObArray** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a function parameter or return value, substitute a pointer to **void**.

```
CObject* CObArray::GetAt( int <nIndex> ) const;
```

for example, translates to

```
void* CPtrArray::GetAt( int <nIndex> ) const;
```

CPtrArray incorporates the **IMPLEMENT_DYNAMIC** macro to support run-time type access and dumping to a **CDumpContext** object. If you need a dump of individual pointer array elements, you must set the depth of the dump context to 1 or greater.

Note
Before using an array, use **SetSize** to establish its size and allocate memory for it. If you do not use **SetSize**, adding elements to your array causes it to be frequently reallocated and copied. Frequent reallocation and copying are inefficient and can fragment memory.

Pointer arrays cannot be serialized.

When a pointer array is deleted, or when its elements are removed, only the pointers are removed, not the entities they reference.

For more information on using **CPtrArray**, see the article Collections.

Requirements

Header: afxcoll.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class
Hierarchy Chart
CObArray Class

Other Resources

CPtrArray Members

CPtrArray Members

Base Class Members

CObject Members

Construction

CPtrArray	Constructs an empty array for void pointers.

Bounds

GetCount	Gets number of elements in this array.
GetSize	Gets number of elements in this array.
GetUpperBound	Returns the largest valid index.
SetSize	Sets the number of elements to be contained in this array.

Operations

FreeExtra	Frees all unused memory above the current upper bound.
IsEmpty	Determines of the array is empty.
RemoveAll	Removes all the elements from this array.

Element Access

ElementAt	Returns a temporary reference to the element pointer within the array.
GetAt	Returns the value at a given index.
GetData	Allows access to elements in the array. Can be **NULL**.
SetAt	Sets the value for a given index; array is not allowed to grow.

Growing the Array

Add	Adds an element to the end of the array; grows the array if necessary.
Append	Appends another array to the array; grows the array if necessary.
Copy	Copies another array to the array; grows the array if necessary.
SetAtGrow	Sets the value for a given index; grows the array if necessary.

Insertion/Removal

InsertAt	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt	Removes an element at a specific index.

Operators

operator []	Sets or gets the element at the specified index.

See Also

Reference

CPtrArray Class

Hierarchy Chart

CPtrList Class

Supports lists of void pointers.

```
class CPtrList : public CObject
```

Remarks

The member functions of **CPtrList** are similar to the member functions of class CObList. Because of this similarity, you can use the **CObList** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a function parameter or return value, substitute a pointer to **void**.

```
CObject*& CObList::GetHead() const;
```

for example, translates to

```
void*& CPtrList::GetHead() const;
```

CPtrList incorporates the **IMPLEMENT_DYNAMIC** macro to support run-time type access and dumping to a **CDumpContext** object. If you need a dump of individual pointer list elements, you must set the depth of the dump context to 1 or greater.

Pointer lists cannot be serialized.

When a **CPtrList** object is deleted, or when its elements are removed, only the pointers are removed, not the entities they reference.

For more information on using **CPtrList**, see the article Collections.

Requirements

Header: afxcoll.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class
Hierarchy Chart
CObList Class

Other Resources

CPtrList Members

CPtrList Members

Base Class Members

COBJECT Members

Construction

CPtrList	Constructs an empty list for void pointers.

Head/Tail Access

GetHead	Returns the head element of the list (cannot be empty).
GetTail	Returns the tail element of the list (cannot be empty).

Operations

AddHead	Adds an element (or all the elements in another list) to the head of the list (makes a new head).
AddTail	Adds an element (or all the elements in another list) to the tail of the list (makes a new tail).
RemoveAll	Removes all the elements from this list.
RemoveHead	Removes the element from the head of the list.
RemoveTail	Removes the element from the tail of the list.

Iteration

GetHeadPosition	Returns the position of the head element of the list.
GetNext	Gets the next element for iterating.
GetPrev	Gets the previous element for iterating.
GetTailPosition	Returns the position of the tail element of the list.

Retrieval/Modification

GetAt	Gets the element at a given position.
RemoveAt	Removes an element from this list, specified by position.
SetAt	Sets the element at a given position.

Insertion

InsertAfter	Inserts a new element after a given position.
InsertBefore	Inserts a new element before a given position.

Searching

Find	Gets the position of an element specified by pointer value.
FindIndex	Gets the position of an element specified by a zero-based index.

Status

GetSize	Returns the number of elements in this list.

GetCount	Returns the number of elements in this list.
IsEmpty	Tests for the empty list condition (no elements).

See Also

Reference

CPtrList Class

Hierarchy Chart

CReBar Class

This topic has been updated for Visual Studio 2005 SP1.

A control bar that provides layout, persistence, and state information for rebar controls.

```
class CReBar : public CControlBar
```

Remarks

A rebar object can contain a variety of child windows, usually other controls, including edit boxes, toolbars, and list boxes. A rebar object can display its child windows over a specified bitmap. Your application can automatically resize the rebar, or the user can manually resize the rebar by clicking or dragging its gripper bar.

Rebar Control

A rebar object behaves similarly to a toolbar object. A rebar uses the click-and-drag mechanism to resize its bands. A rebar control can contain one or more bands, with each band having any combination of a gripper bar, a bitmap, a text label, and a child window. However, bands cannot contain more than one child window.

CReBar uses the CReBarCtrl class to provide its implementation. You can access the rebar control through GetReBarCtrl to take advantage of the control's customization options. For more information about rebar controls, see **CReBarCtrl**. For more information about using rebar controls, see Using CReBarCtrl.

Caution
Rebar and rebar control objects do not support MFC control bar docking. If **CRebar::EnableDocking** is called, your application will assert.

Requirements

Header: afxext.h

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

See Also

Tasks

MFCIE Sample: Demonstrates the MFC CHtmlView and CReBar Classes

Reference

CControlBar Class

Hierarchy Chart

Other Resources

CReBar Members

CReBar Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CControlBar Members

Construction

Create	Creates the rebar control and attaches it to the **CReBar** object.

Attributes

AddBar	Adds a band to a rebar.
GetReBarCtrl	Allows direct access to the underlying common control.

See Also

Reference

CReBar Class

Hierarchy Chart

CReBar Member Functions

For information about the member functions in **CReBar**, see CReBar Members.

CReBar::AddBar

Call this member function to add a band to the rebar.

```
BOOL AddBar(
   CWnd* pBar,
   LPCTSTR pszText = NULL,
   CBitmap* pbmp = NULL,
   DWORD dwStyle = RBBS_GRIPPERALWAYS | RBBS_FIXEDBMP
);
BOOL AddBar(
   CWnd* pBar,
   COLORREF clrFore,
   COLORREF clrBack,
   LPCTSTR pszText = NULL,
   DWORD dwStyle = RBBS_GRIPPERALWAYS
);
```

Parameters

pBar

A pointer to a **CWnd** object that is the child window to be inserted into the rebar. The referenced object must have a **WS_CHILD**.

lpszText

A pointer to a string containing the text to appear on the rebar. **NULL** by default. The text contained in *lpszText* is not part of the child window; it is on the rebar itself.

pbmp

A pointer to a **CBitmap** object to be displayed on the rebar background. **NULL** by default.

dwStyle

A **DWORD** containing the style to apply to the rebar. See the **fStyle** function description in the Win32 structure REBARBANDINFO for a complete list of band styles.

clrFore

A **COLORREF** value that represents the foreground color of the rebar.

clrBack

A **COLORREF** value that represents the background color of the rebar.

Return Value

Nonzero if successful; otherwise 0.

Example

```
// Define a pointer to a CRebar in your class definition,
// such as: CReBar* m_pReBar; More often, however, you
// would probably specify an instance in your class
// definition, such as: CReBar m_ReBar;
m_pReBar = new CReBar();
m_pReBar->Create(this);
m_wndDlgBar.Create(this, IDR_MAINFRAME, CBRS_ALIGN_TOP,
    AFX_IDW_DIALOGBAR);
m_pReBar->AddBar(&m_wndDlgBar);
```

See Also

Reference

CReBar Class

Hierarchy Chart

CReBarCtrl Class

Other Resources

CReBar Members

CReBar::Create

Call this member function to create a rebar.

```
virtual BOOL Create(
   CWnd* pParentWnd,
   DWORD dwCtrlStyle = RBS_BANDBORDERS,
   DWORD dwStyle = WS_CHILD | WS_VISIBLE | WS_CLIPSIBLINGS | WS_CLIPCHILDREN | CBRS_TOP,
   UINT nID = AFX_IDW_REBAR
);
```

Parameters

pParentWnd

Pointer to the **CWnd** object whose Windows window is the parent of the status bar. Normally your frame window.

dwCtrlStyle

The rebar control style. By default, **RBS_BANDBORDERS**, which displays narrow lines to separate adjacent bands within the rebar control. See Rebar Control Styles in the *Platform SDK* for a list of styles.

dwStyle

The rebar window styles.

nID

The rebar's child-window ID.

Return Value

Nonzero if successful; otherwise 0.

Example

See the example for CReBar::AddBar.

See Also

Reference

CReBar Class
Hierarchy Chart
CReBarCtrl Class

Other Resources

CReBar Members

CReBar::GetReBarCtrl

This member function allows direct access to the underlying common control.

```
CReBarCtrl& GetReBarCtrl( ) const;
```

Return Value

A reference to a CReBarCtrl object.

Remarks

Call this member function to take advantage of the functionality of the Windows rebar common control in customizing your rebar. When you call **GetReBarCtrl**, it returns a reference object to the **CReBarCtrl** object so you can use either set of member functions.

For more information about using **CReBarCtrl** to customize your rebar, see Using CReBarCtrl.

Example

```
CReBarCtrl& refReBarCtrl = m_wndReBar.GetReBarCtrl();

UINT nBandCount = refReBarCtrl.GetBandCount();
CString msg;
msg.Format("Band Count is: %u", nBandCount);
AfxMessageBox(msg);
```

See Also

Reference

CReBar Class

Hierarchy Chart

Other Resources

CReBar Members

CReBarCtrl Class

This topic has been updated for Visual Studio 2005 SP1.

Encapsulates the functionality of a rebar control, which is a container for a child window.

```
class CReBarCtrl : public CWnd
```

Remarks

The application in which the rebar control resides assigns the child window contained by the rebar control to the rebar band. The child window is usually another common control.

Rebar controls contain one or more bands. Each band can contain a combination of a gripper bar, a bitmap, a text label, and a child window. The band can contain only one of each of these items.

The rebar control can display the child window over a specified background bitmap. All rebar control bands can be resized, except those that use the **RBBS_FIXEDSIZE** style. As you reposition or resize a rebar control band, the rebar control manages the size and position of the child window assigned to that band. To resize or change the order of bands within the control, click and drag a band's gripper bar.

The following illustration shows a rebar control that has three bands:

- Band 0 contains a flat, transparent toolbar control.
- Band 1 contains both transparent standard and transparent dropdown buttons.
- Band 2 contains a combo box and four standard buttons.

Rebar control

Rebar controls support:

- Image lists.
- Message-handling.
- Custom draw functionality.
- A variety of control styles in addition to standard window styles. For a list of these styles, see Rebar Control Styles in the *Platform SDK*.

For more information, see Using CReBarCtrl.

Requirements

Header: afxcmn.h

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

The Smart Device version of CReBarCtrl Class does not support the following methods:

- CReBarCtrl::GetColorScheme
- CReBarCtrl::SetColorScheme - Windows CE does not support RB_GETCOLORSCHEME/RB_SETCOLORSCHEME messages.
- CReBarCtrl::GetDropTarget

- CReBarCtrl::GetPalette
- CReBarCtrl::SetPalette - Windows CE does not support RB_GETPALETTE/RB_SETPALETTE messages.
- CReBarCtrl::GetToolTips
- CReBarCtrl::SetToolTips - MFC for devices does not support tool tips.
- CReBarCtrl::MoveBand Windows CE does not support RB_MOVEBAND messages.
- CReBarCtrl::GetBandMargins
- CReBarCtrl::SetWindowTheme

In Smartphone and Pocket PC platforms, the control bar always aligns to the bottom. The alignment is not configurable.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CReBarCtrl Members

CReBarCtrl Members

This topic has been updated for Visual Studio 2005 SP1.

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Constructors

CReBarCtrl	Constructs a **CReBarCtrl** object.

Initialization

Create	Creates the rebar control and attaches it to the **CReBarCtrl** object.
CreateEx	Creates a rebar control with the specified Windows extended styles and attaches it to a **CReBarCtrl** object.

Attributes

GetBandBorders	Retrieves the borders of a band.
GetBandCount	Retrieves the count of bands currently in the rebar control.
GetBandInfo	Retrieves information about a specified band in a rebar control.
GetBandMargins	Retrieves the margins of a band.
GetBarHeight	Retrieves the height of the rebar control.
GetBarInfo	Retrieves information about the rebar control and the image list it uses.
GetBkColor	Retrieves a rebar control's default background color.
GetDropTarget	Retrieves a rebar control's **IDropTarget** interface pointer.
GetImageList	Retrieves the image list associated with a rebar control.
GetPalette	Retrieves the rebar control's current palette.
GetColorScheme	Retrieves the COLORSCHEME structure associated with the rebar control.
GetRect	Retrieves the bounding rectangle for a given band in a rebar control.
GetRowCount	Retrieves the number of band rows in a rebar control.
GetRowHeight	Retrieves the height of a specified row in a rebar control.
GetTextColor	Retrieves a rebar control's default text color.
GetToolTips	Retrieves the handle to any tool tip control associated with the rebar control.
IDToIndex	Converts a band identifier (ID) to a band index in a rebar control.

SetBandInfo	Sets characteristics of an existing band in a rebar control.
SetBarInfo	Sets the characteristics of a rebar control.
SetBkColor	Sets a rebar control's default background color.
SetImageList	Sets a rebar control's image list.
SetOwner	Sets a rebar control's owner window.
SetPalette	Sets the rebar control's current palette.
SetColorScheme	Sets the color scheme for the buttons on a rebar control.
SetTextColor	Sets a rebar control's default text color.
SetToolTips	Associates a tool tip control with the rebar control.
SetWindowTheme	Sets the visual style of the rebar control.

Operations

BeginDrag	Places the rebar control into drag-and-drop mode.
DeleteBand	Deletes a band from a rebar control.
DragMove	Updates the drag position in the rebar control after a call to **BeginDrag**.
EndDrag	Terminates the rebar control's drag-and-drop operation.
HitTest	Determines which portion of a rebar band is at a given point on the screen, if a rebar band exists at that point.
InsertBand	Inserts a new band in a rebar control.
MaximizeBand	Resizes a band in a rebar control to its largest size.
MinimizeBand	Resizes a band in a rebar control to its smallest size.
MoveBand	Moves a band from one index to another.
PushChevron	Programmatically pushes a chevron.
RestoreBand	Resizes a band in a rebar control to its ideal size.
ShowBand	Shows or hides a given band in a rebar control.
SizeToRect	Fits a rebar control to a specified rectangle.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

The Smart Device version of CReBarCtrl Class does not support the following methods:

- CReBarCtrl::GetColorScheme

- CReBarCtrl::SetColorScheme - Windows CE does not support RB_GETCOLORSCHEME/RB_SETCOLORSCHEME messages.
- CReBarCtrl::GetDropTarget
- CReBarCtrl::GetPalette
- CReBarCtrl::SetPalette - Windows CE does not support RB_GETPALETTE/RB_SETPALETTE messages.
- CReBarCtrl::GetToolTips
- CReBarCtrl::SetToolTips - MFC for devices does not support tool tips.
- CReBarCtrl::MoveBand Windows CE does not support RB_MOVEBAND messages.
- CReBarCtrl::GetBandMargins
- CReBarCtrl::SetWindowTheme

In Smartphone and Pocket PC platforms, the control bar always aligns to the bottom. The alignment is not configurable.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart

CReBarCtrl Member Functions

For information about the member functions in **CReBarCtrl**, see CReBarCtrl Members.

CReBarCtrl::BeginDrag

Implements the behavior of the Win32 message RB_BEGINDRAG, as described in the *Platform SDK*.

```
void BeginDrag(
   UINT uBand,
   DWORD dwPos = (DWORD
)-1
);
```

Parameters

uBand

Zero-based index of the band that the drag-and-drop operation will affect.

dwPos

A **DWORD** value that contains the starting mouse coordinates. The horizontal coordinate is contained in the LOWORD and the vertical coordinate is contained in the HIWORD. If you pass `(DWORD)-1`, the rebar control will use the position of the mouse the last time the control's thread called **GetMessage** or **PeekMessage**.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
GetMessage
PeekMessage

Concepts

CReBarCtrl Members

CReBarCtrl::Create

Creates the rebar control and attaches it to the **CReBarCtrl** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the combination of rebar control styles applied to the control. See Rebar Control Styles in the *Platform SDK* for a list of supported styles.

rect

A reference to a CRect object or RECT structure, which is the position and size of the rebar control.

pParentWnd

A pointer to a CWnd object that is the parent window of the rebar control. It must not be **NULL**.

nID

Specifies the rebar control's control ID.

Return Value

Nonzero if the object was created successfully; otherwise 0.

Remarks

Create a rebar control in two steps:

1. Call CReBarCtrl to construct a **CReBarCtrl** object.
2. Call this member function, which creates the Windows rebar control and attaches it to the **CReBarCtrl** object.

When you call **Create**, the common controls are initialized.

Example

```
CReBarCtrl* pReBarCtrl = new CReBarCtrl();
CRect rect;
GetWindowRect(rect);
pReBarCtrl->Create(RBS_BANDBORDERS, rect, this, AFX_IDW_REBAR);
// Use ReBar Control.
...
delete pReBarCtrl;
```

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::CReBarCtrl
CReBarCtrl::CreateEx

Concepts

CReBarCtrl Members

CReBarCtrl::CreateEx

Creates a control (a child window) and associates it with the **CReBarCtrl** object.

```
virtual BOOL CreateEx(
   DWORD dwExStyle,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwExStyle

Specifies the extended style of the control being created. For a list of extended Windows styles, see the *dwExStyle* parameter for CreateWindowEx in the *Platform SDK*.

dwStyle

Specifies the combination of rebar control styles applied to the control. For a list of supported styles, see Rebar Control Styles in the *Platform SDK*.

rect

A reference to a RECT structure describing the size and position of the window to be created, in client coordinates of *pParentWnd*.

pParentWnd

A pointer to the window that is the control's parent.

nID

The control's child-window ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Use **CreateEx** instead of Create to apply extended Windows styles, specified by the Windows extended style preface **WS_EX_**.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::CReBarCtrl

Concepts

CReBarCtrl Members

CReBarCtrl::CReBarCtrl

Creates a **CReBarCtrl** object.

```
CReBarCtrl( );
```

Example

See the example for CReBarCtrl::Create.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::Create
CReBarCtrl::CreateEx

Concepts

CReBarCtrl Members

CReBarCtrl::DeleteBand

Implements the behavior of the Win32 message RB_DELETEBAND, as described in the *Platform SDK*.

```
BOOL DeleteBand(
   UINT uBand
);
```

Parameters

uBand

Zero-based index of the band to be deleted.

Return Value

Nonzero if the band deleted successfully; otherwise zero.

Example

```
    UINT nCount = m_wndReBar.GetReBarCtrl().GetBandCount();

    if (nCount > 0)
       m_wndReBar.GetReBarCtrl().DeleteBand(nCount - 1);
```

See Also

Reference

CReBarCtrl Class

Hierarchy Chart

Concepts

CReBarCtrl Members

CReBarCtrl::DragMove

Implements the behavior of the Win32 message RB_DRAGMOVE, as described in the *Platform SDK*.

```
void DragMove(
   DWORD dwPos = (DWORD
)-1
);
```

Parameters

dwPos

A **DWORD** value that contains the new mouse coordinates. The horizontal coordinate is contained in the LOWORD and the vertical coordinate is contained in the HIWORD. If you pass `(DWORD)-1`, the rebar control will use the position of the mouse the last time the control's thread called **GetMessage** or **PeekMessage**.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::BeginDrag
CReBarCtrl::EndDrag
GetMessage
PeekMessage

Concepts

CReBarCtrl Members

CReBarCtrl::EndDrag

Implements the behavior of the Win32 message RB_ENDDRAG, as described in the *Platform SDK*.

```
void EndDrag( );
```

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::BeginDrag
CReBarCtrl::DragMove

Concepts

CReBarCtrl Members

CReBarCtrl::GetBandBorders

Implements the behavior of the Win32 message RB_GETBANDBORDERS, as described in the *Platform SDK*.

```
void GetBandBorders(
   UINT uBand,
   LPRECT prc
) const;
```

Parameters

uBand

Zero-based index of the band for which the borders will be retrieved.

prc

A pointer to a RECT structure that will receive the band borders. If the rebar control has the **RBS_BANDBORDERS** style, each member of this structure will receive the number of pixels, on the corresponding side of the band, that constitute the border. If the rebar control does not have the **RBS_BANDBORDERS** style, only the left member of this structure receives valid information. For a description of rebar control styles, see Rebar Control Styles in the *Platform SDK*.

See Also

Reference

CReBarCtrl Class

Hierarchy Chart

Concepts

CReBarCtrl Members

CReBarCtrl::GetBandCount

Implements the behavior of the Win32 message RB_GETBANDCOUNT, as described in the *Platform SDK*.

```
UINT GetBandCount( ) const;
```

Return Value

The number of bands assigned to the control.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::GetBandInfo
CReBarCtrl::SetBandInfo
CReBarCtrl::DeleteBand
CReBarCtrl::InsertBand
CReBarCtrl::ShowBand

Concepts

CReBarCtrl Members

CReBarCtrl::GetBandInfo

Implements the behavior of the Win32 message RB_GETBANDINFO as described in the *Platform SDK*.

```
BOOL GetBandInfo(
   UINT uBand,
   REBARBANDINFO* prbbi
) const;
```

Parameters

uBand

Zero-based index of the band for which the information will be retrieved.

prbbi

A pointer to a REBARBANDINFO structure to receive the band information. You must set the **cbSize** member of this structure to `sizeof(REBARBANDINFO)` and set the **fMask** member to the items you want to retrieve before sending this message.

Return Value

Nonzero if successful; otherwise zero.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::SetBandInfo
CReBarCtrl::GetBandCount
CReBarCtrl::DeleteBand
CReBarCtrl::InsertBand
CReBarCtrl::ShowBand

Concepts

CReBarCtrl Members

CReBarCtrl::GetBandMargins

This topic has been updated for Visual Studio 2005 SP1.

Retrieves the margins of the band.

```
void GetBandMargins(
   PMARGINS pMargins
);
```

Parameters

pMargins

A pointer to a MARGINSstructure that will receive the information.

Remarks

This member function emulates the functionality of the RB_GETBANDMARGINS message, as described in the *Platform SDK*.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CReBarCtrl Class

Hierarchy Chart

Concepts

CReBarCtrl Members

CReBarCtrl::GetBarHeight

Retrieves the height of the rebar bar.

```
UINT GetBarHeight( ) const;
```

Return Value

Value that represents the height, in pixels, of the control.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::GetBarInfo
CReBarCtrl::SetBarInfo

Concepts

CReBarCtrl Members

CReBarCtrl::GetBarInfo

Implements the behavior of the Win32 message RB_GETBARINFO, as described in the *Platform SDK*.

```
BOOL GetBarInfo(
   REBARINFO* prbi
) const;
```

Parameters

prbi

A pointer to a REBARINFO structure that will receive the rebar control information. You must set the **cbSize** member of this structure to `sizeof(REBARINFO)` before sending this message.

Return Value

Nonzero if successful; otherwise zero.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::SetBarInfo

Concepts

CReBarCtrl Members

CReBarCtrl::GetBkColor

Implements the behavior of the Win32 message RB_GETBKCOLOR, as described in the *Platform SDK*.

```
COLORREF GetBkColor( ) const;
```

Return Value

A **COLORREF** value that represent the current default background color.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::SetBkColor

Concepts

CReBarCtrl Members

CReBarCtrl::GetColorScheme

This topic has been updated for Visual Studio 2005 SP1.

Retrieves the COLORSCHEME structure for the rebar control.

```
BOOL GetColorScheme( COLORSCHEME* lpcs );
```

Parameters

lpcs

A pointer to a COLORSCHEME structure, as described in the *Platform SDK*.

Return Value

Nonzero if successful; otherwise zero.

Remarks

The **COLORSCHEME** structure includes the button highlight color and the button shadow color.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::SetColorScheme

Concepts

CReBarCtrl Members

CReBarCtrl::GetDropTarget

This topic has been updated for Visual Studio 2005 SP1.

Implements the behavior of the Win32 message RB_GETDROPTARGET, as described in the *Platform SDK*.

```
IDropTarget* GetDropTarget( ) const;
```

Return Value

A pointer to an IDropTarget interface.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CReBarCtrl Class

Hierarchy Chart

Concepts

CReBarCtrl Members

CReBarCtrl::GetImageList

Gets the **CImageList** object associated with a rebar control.

```
CImageList* GetImageList( ) const;
```

Return Value

A pointer to a CImageList object. Returns **NULL** if no image list is set for the control.

Remarks

This member function uses size and mask information stored in the REBARINFO structure, as described in the *Platform SDK*.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::SetImageList

Concepts

CReBarCtrl Members

CReBarCtrl::GetPalette

This topic has been updated for Visual Studio 2005 SP1.

Retrieves the rebar control's current palette.

```
CPalette* GetPalette( ) const;
```

Return Value

A pointer to a CPalette object specifying the rebar control's current palette.

Remarks

Note that this member function uses a **CPalette** object as its return value, rather than an **HPALETTE**.

Example

```
CPalette* pPalette = m_wndReBar.GetReBarCtrl().GetPalette();
if (pPalette)
{
   int nEntries = pPalette->GetEntryCount();
   CString msg;
   msg.Format("Number of palette entries: %d", nEntries);
   AfxMessageBox(msg);
}
else
   AfxMessageBox("No palette!");
```

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::SetPalette

Concepts

CReBarCtrl Members

CReBarCtrl::GetRect

Implements the behavior of the Win32 message RB_GETRECT, as described in the *Platform SDK*.

```
BOOL GetRect(
   UINT uBand,
   LPRECT prc
) const;
```

Parameters

uBand

Zero-based index of a band in the rebar control.

prc

A pointer to a RECT structure that will receive the bounds of the rebar band.

Return Value

Nonzero if successful; otherwise zero.

Example

```
CRect rc;
m_wndReBar.GetReBarCtrl().GetRect(0, &rc);
CString msg;
msg.Format("rect.left = %d, rect.top = %d,
    rect.right = %d, rect.bottom = %d", rc.left,
    rc.top, rc.right, rc.bottom);
AfxMessageBox(msg);
```

See Also

Reference

CReBarCtrl Class

Hierarchy Chart

CReBarCtrl::SizeToRect

Concepts

CReBarCtrl Members

CReBarCtrl::GetRowCount

Implements the behavior of the Win32 message RB_GETROWCOUNT, as described in the *Platform SDK*.

```
UINT GetRowCount( ) const;
```

Return Value

A **UINT** value that represents the number of band rows in the control.

Example

```
UINT nRowCount = m_wndReBar.GetReBarCtrl().GetRowCount();
CString msg;
msg.Format("Row Count is %d", nRowCount);
AfxMessageBox(msg);
```

See Also

Reference

CReBarCtrl Class

Hierarchy Chart

CReBarCtrl::GetRowHeight

Concepts

CReBarCtrl Members

CReBarCtrl::GetRowHeight

Implements the behavior of the Win32 message RB_GETROWHEIGHT, as described in the *Platform SDK*.

```
UINT GetRowHeight(
   UINT uRow
) const;
```

Parameters

uRow

Zero-based index of the band that will have its height retrieved.

Return Value

A **UINT** value that represents the row height, in pixels.

Example

```
int nCount = m_wndReBar.GetReBarCtrl().GetRowCount();
for (int i=0; i<nCount; i++)
{
   UINT nHeight = m_wndReBar.GetReBarCtrl().GetRowHeight(i);
   CString msg;
   msg.Format("Height of row %d is %u", i, nHeight);
   AfxMessageBox(msg);
}
```

See Also

Reference

CReBarCtrl Class

Hierarchy Chart

CReBarCtrl::GetRowCount

Concepts

CReBarCtrl Members

CReBarCtrl::GetTextColor

Implements the behavior of the Win32 message RB_GETTEXTCOLOR, as described in the *Platform SDK*.

```
COLORREF GetTextColor( ) const;
```

Return Value

A **COLORREF** value that represent the current default text color.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::SetTextColor
CReBarCtrl::GetBkColor
CReBarCtrl::SetBkColor

Concepts

CReBarCtrl Members

CReBarCtrl::GetToolTips

This topic has been updated for Visual Studio 2005 SP1.

Implements the behavior of the Win32 message RB_GETTOOLTIPS, as described in the *Platform SDK*.

```
CToolTipCtrl* GetToolTips( ) const;
```

Return Value

A pointer to a CToolTipCtrl object.

Remarks

Note that the MFC implementation of **GetToolTips** returns a pointer to a **CToolTipCtrl**, rather than an **HWND**.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::SetToolTips

Concepts

CReBarCtrl Members

CReBarCtrl::HitTest

Implements the behavior of the Win32 message RB_HITTEST, as described in the *Platform SDK*.

```
int HitTest(
   RBHITTESTINFO* prbht
);
```

Parameters

prbht

A pointer to a RBHITTESTINFO structure. Before sending the message, the **pt** member of this structure must be initialized to the point that will be tested, in client coordinates.

Return Value

The zero-based index of the band at the given point, or -1 if no rebar band was at the point.

See Also

Reference

CReBarCtrl Class

Hierarchy Chart

Concepts

CReBarCtrl Members

CReBarCtrl::IDToIndex

Implements the behavior of the Win32 message RB_IDTOINDEX, as described in the *Platform SDK*.

```
int IDToIndex(
   UINT uBandID
) const;
```

Parameters

uBandID

The application-defined identifier of the specified band, passed in the **wID** member of the REBARBANDINFO structure when the band is inserted.

Return Value

The zero-based band index if successful, or -1 otherwise. If duplicate band indices exist, the first one is returned.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::InsertBand

Concepts

CReBarCtrl Members

CReBarCtrl::InsertBand

Implements the behavior of the Win32 message RB_INSERTBAND, as described in the *Platform SDK*.

```
BOOL InsertBand(
   UINT uIndex,
   REBARBANDINFO* prbbi
);
```

Parameters

uIndex

Zero-based index of the location where the band will be inserted. If you set this parameter to -1, the control will add the new band at the last location.

prbbi

A pointer to a REBARBANDINFO structure that defines the band to be inserted. You must set the **cbSize** member of this structure to `sizeof(REBARBANDINFO)` before calling this function.

Return Value

Nonzero if successful; otherwise zero.

Example

```
LPREBARBANDINFO prbbi =
    (LPREBARBANDINFO)alloca(sizeof(REBARBANDINFO));
prbbi->cbSize = sizeof(REBARBANDINFO);

LPTSTR lpszText = (LPTSTR)alloca(80);
prbbi->lpText = lpszText;
prbbi->cch = 80;

prbbi->fMask = RBBIM_BACKGROUND | RBBIM_CHILD |
    RBBIM_CHILDSIZE | RBBIM_COLORS | RBBIM_HEADERSIZE |
    RBBIM_IDEALSIZE | RBBIM_ID | RBBIM_IMAGE |
    RBBIM_LPARAM | RBBIM_SIZE | RBBIM_STYLE | RBBIM_TEXT;

m_wndReBar.GetReBarCtrl().GetBandInfo(1, prbbi);

m_wndReBar.GetReBarCtrl().InsertBand(2, prbbi);
```

See Also

Reference

CReBarCtrl Class

Hierarchy Chart

Concepts

CReBarCtrl Members

CReBarCtrl::MaximizeBand

Resizes a band in a rebar control to its largest size.

```
void MaximizeBand(
   UINT uBand
);
```

Parameters

uBand

Zero-based index of the band to be maximized.

Remarks

Implements the behavior of the Win32 message RB_MAXIMIZEBAND with *fIdeal* set to 0, as described in the *Platform SDK*.

Example

```
UINT nCount = m_pReBarCtrl->GetBandCount();

for (UINT i=0; i<nCount; i++)
   m_pReBarCtrl->MaximizeBand(i);
```

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::MinimizeBand
CReBarCtrl::RestoreBand

Concepts

CReBarCtrl Members

CReBarCtrl::MinimizeBand

Resizes a band in a rebar control to its smallest size.

```
void MinimizeBand(
   UINT uBand
);
```

Parameters

uBand

Zero-based index of the band to be minimized.

Remarks

Implements the behavior of the Win32 message RB_MINIMIZEBAND, as described in the *Platform SDK*.

Example

```
UINT nCount = m_pReBarCtrl->GetBandCount();

for (UINT i=0; i<nCount; i++)
   m_pReBarCtrl->MinimizeBand(i);
```

See Also

Reference

CReBarCtrl Class

Hierarchy Chart

CReBarCtrl::MaximizeBand

Concepts

CReBarCtrl Members

CReBarCtrl::MoveBand

This topic has been updated for Visual Studio 2005 SP1.

Implements the behavior of the Win32 message RB_MOVEBAND, as described in the *Platform SDK*.

```
BOOL MoveBand(
   UINT uFrom,
   UINT uTo
);
```

Parameters

uFrom

Zero-based index of the band to be moved.

uTo

Zero-based index of the new band position. This parameter value must never be greater than the number of bands minus one. To obtain the number of bands, call GetBandCount.

Return Value

Nonzero if successful; otherwise zero.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CReBarCtrl Class

Hierarchy Chart

Concepts

CReBarCtrl Members

CReBarCtrl::PushChevron

Implements the behavior of the Win32 message RB_PUSHCHEVRON, as described in the *Platform SDK*.

```
void PushChevron(
   UINT uBand,
   LPARAM lAppValue
);
```

Parameters

uBand

Zero-based index of the band whose chevron is to be pushed.

lAppValue

An application defined 32-bit value. See *lAppValue* in RB_PUSHCHEVRON in the *Platform SDK*.

See Also

Reference

CReBarCtrl Class

Hierarchy Chart

Concepts

CReBarCtrl Members

CReBarCtrl::RestoreBand

Resizes a band in a rebar control to its ideal size.

```
void RestoreBand(
   UINT uBand
);
```

Parameters

uBand

Zero-based index of the band to be maximized.

Remarks

Implements the behavior of the Win32 message RB_MAXIMIZEBAND with *fIdeal* set to 1, as described in the *Platform SDK*.

Example

```
UINT nCount = m_pReBarCtrl->GetBandCount();

for (UINT i=0; i<nCount; i++)
   m_pReBarCtrl->RestoreBand(i);
```

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::MaximizeBand
CReBarCtrl::MinimizeBand

Concepts

CReBarCtrl Members

CReBarCtrl::SetBandInfo

Implements the behavior of the Win32 message RB_SETBANDINFO, as described in the *Platform SDK*.

```
BOOL SetBandInfo(
   UINT uBand,
   REBARBANDINFO* prbbi
);
```

Parameters

uBand

Zero-based index of the band to receive the new settings.

prbbi

Pointer to a REBARBANDINFO structure that defines the band to be inserted. You must set the **cbSize** member of this structure to `sizeof(REBARBANDINFO)` before sending this message.

Return Value

Nonzero if successful; otherwise zero.

Example

```
int nCount = m_wndReBar.GetReBarCtrl().GetBandCount();
CString strText = "Band #:";

for (int i=0; i<nCount; i++)
{
   LPREBARBANDINFO prbbi =
      (LPREBARBANDINFO)alloca(sizeof(REBARBANDINFO));
   prbbi->cbSize = sizeof(REBARBANDINFO);
   CString strText;
   strText.Format("Band #: %d", i);
   LPTSTR lpszText = strText.GetBuffer(strText.GetLength());
   prbbi->lpText = lpszText;
   prbbi->cch = strlen(lpszText) + 1;
   prbbi->fMask = RBBIM_TEXT;

   m_wndReBar.GetReBarCtrl().SetBandInfo(i, prbbi);

   strText.ReleaseBuffer();
}
```

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::GetBandInfo

Concepts

CReBarCtrl Members

CReBarCtrl::SetBarInfo

Implements the behavior of the Win32 message RB_SETBARINFO, as described in the *Platform SDK*.

```
BOOL SetBarInfo(
   REBARINFO* prbi
);
```

Parameters

prbi

A pointer to a REBARINFO structure that contains the information to be set. You must set the **cbSize** member of this structure to `sizeof(REBARINFO)` before sending this message

Return Value

Nonzero if successful; otherwise zero.

Example

```
LPREBARINFO prbi = 0;
prbi = (LPREBARINFO)alloca(sizeof(REBARINFO));
if (!prbi)
{
   AfxMessageBox("Couldn't allocate memory for
       REBARINFO structure!");
   return;
}

prbi->cbSize = sizeof(REBARINFO);
prbi->fMask = 0;
prbi->himl = 0;
m_wndReBar.GetReBarCtrl().SetBarInfo(prbi);
```

See Also

Reference

CReBarCtrl Class

Hierarchy Chart

CReBarCtrl::GetBarInfo

Concepts

CReBarCtrl Members

CReBarCtrl::SetBkColor

Implements the behavior of the Win32 message RB_SETBKCOLOR, as described in the *Platform SDK*.

```
COLORREF SetBkColor(
   COLORREF clr
);
```

Parameters

clr

The **COLORREF** value that represents the new default background color.

Return Value

A COLORREF value that represents the previous default background color.

Remarks

See this topic for more information about when to set the background color, and how to set the default.

See Also

Reference

CReBarCtrl Class

Hierarchy Chart

CReBarCtrl::GetBkColor

Concepts

CReBarCtrl Members

CReBarCtrl::SetColorScheme

This topic has been updated for Visual Studio 2005 SP1.

Sets the color scheme for the buttons on a rebar control.

```
void SetColorScheme( const COLORSCHEME* lpcs );
```

Parameters

lpcs

A pointer to a COLORSCHEME structure, as described in the *Platform SDK*.

Remarks

The **COLORSCHEME** structure includes both the button highlight color and the button shadow color.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::GetColorScheme

Concepts

CReBarCtrl Members

CReBarCtrl::SetImageList

Assigns an image list to a rebar control.

```
BOOL SetImageList(
   CImageList* pImageList
);
```

Parameters

pImageList

A pointer to a CImageList object containing the image list to be assigned to the rebar control.

Return Value

Nonzero if successful; otherwise zero.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::GetImageList

Concepts

CReBarCtrl Members

CReBarCtrl::SetOwner

Implements the behavior of the Win32 message RB_SETPARENT, as described in the *Platform SDK*.

```
CWnd* SetOwner(
   CWnd* pWnd
);
```

Parameters

pWnd

A pointer to a **CWnd** object to set as the owner of the rebar control.

Return Value

A pointer to a CWnd object that is the current owner of the rebar control.

Remarks

Note that this member function uses pointers to **CWnd** objects for both the current and selected owner of the rebar control, rather than handles to windows.

Note
This member function does not change the actual parent that was set when the control was created; rather it sends notification messages to the window you specify.

See Also

Reference

CReBarCtrl Class

Hierarchy Chart

Concepts

CReBarCtrl Members

CReBarCtrl::SetPalette

This topic has been updated for Visual Studio 2005 SP1.

Implements the behavior of the Win32 message RB_SETPALETTE, as described in the *Platform SDK*.

```
CPalette* SetPalette(
   HPALETTE hPal
);
```

Parameters

hPal

An **HPALETTE** that specifies the new palette that the rebar control will use.

Return Value

A pointer to a CPalette object specifying the rebar control's previous palette.

Remarks

Note that this member function uses a **CPalette** object as its return value, rather than an **HPALETTE**.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::GetPalette

Concepts

CReBarCtrl Members

CReBarCtrl::SetTextColor

Implements the behavior of the Win32 message RB_SETTEXTCOLOR, as described in the *Platform SDK*.

```
COLORREF SetTextColor(
   COLORREF clr
);
```

Parameters

clr

A **COLORREF** value that represents the new text color in the **CReBarCtrl** object.

Return Value

The COLORREF value representing the previous text color associated with the **CReBarCtrl** object.

Remarks

It is provided to support text color flexibility in a rebar control.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::GetTextColor

Concepts

CReBarCtrl Members

CReBarCtrl::SetToolTips

This topic has been updated for Visual Studio 2005 SP1.

Associates a tool tip control with a rebar control.

```
void SetToolTips(
   CToolTipCtrl* pToolTip
);
```

Parameters

pToolTip

A pointer to a CToolTipCtrl object

Remarks

You must destroy the **CToolTipCtrl** object when you are done with it.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::GetToolTips

Concepts

CReBarCtrl Members

CReBarCtrl::SetWindowTheme

This topic has been updated for Visual Studio 2005 SP1.

Sets the visual style of the rebar control.

```
HRESULT SetWindowTheme(
   LPCWSTR pszSubAppName
);
```

Parameters

pszSubAppName

A pointer to a Unicode string that contains the rebar visual style to set.

Return Value

The return value is not used.

Remarks

This member function emulates the functionality of the RB_SETWINDOWTHEME message, as described in the *Platform SDK*.

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

This method is not supported in Smart Device projects.

See Also

Reference

CReBarCtrl Class

Hierarchy Chart

Concepts

CReBarCtrl Members

CReBarCtrl::ShowBand

Implements the behavior of the Win32 message RB_SHOWBAND, as described in the *Platform SDK*.

```
BOOL ShowBand(
   UINT uBand,
   BOOL fShow = TRUE
);
```

Parameters

uBand

Zero-based index of a band in the rebar control.

fShow

Indicates if the band should be shown or hidden. If this value is **TRUE**, the band will be shown. Otherwise, the band will be hidden.

Return Value

Nonzero if successful; otherwise zero.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::SetBandInfo
CReBarCtrl::GetBandCount
CReBarCtrl::DeleteBand
CReBarCtrl::InsertBand
CReBarCtrl::GetBandInfo

Concepts

CReBarCtrl Members

CReBarCtrl::SizeToRect

Implements the behavior of the Win32 message RB_SIZETORECT, as described in the *Platform SDK*.

```
BOOL SizeToRect(
   CRect& rect
);
```

Parameters

rect

A reference to a CRect object that specifies the rectangle that the rebar control should be sized to.

Return Value

Nonzero if successful; otherwise zero.

Remarks

Note that this member function uses a **CRect** object as a parameter, rather than a **RECT** structure.

See Also

Reference

CReBarCtrl Class
Hierarchy Chart
CReBarCtrl::GetRect

Concepts

CReBarCtrl Members

CRecentFileList Class

This topic has been updated for Visual Studio 2005 SP1.

Supports control of the most recently used (MRU) file list.

```
class CRecentFileList
```

Remarks

Files can be added to or deleted from the MRU file list, the file list can be read from or written to the registry or an .INI file, and the menu displaying the MRU file list can be updated.

For more information about MRU menu items, see

- Knowledge Base article Q243751 : HOWTO: Add Command Handlers for MRU Menu Items in MFC Application

Requirements

Header: afxadv.h

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

To use CRecentFileList in smart device doc/view architecture projects, your project must utilize macros defined in the header file MRUMacros.h.

Make the following modifications in various files generated by the MFC device application wizard in order to use CRecentFileList.

- In the file <ProjectName>Doc.h
  - Include the header file by adding `#include "MRUMacros.h"` at the top
  - Add the macro `DECLARE_MRU_DOC_SUPPORT(C<ProjectName>Doc)` to the C<ProjectName> class.
- In the file <ProjectName>.h
  - Include the header file by adding `#include "MRUMacros.h"` at the top
  - Add the macro `DECLARE_MRU_APP_SUPPORT(C<ProjectName>App)` to the <ProjectName>App class
- In the file <ProjectName>.cpp
  - Add the macro `MRU_INIT()` in constructor of C<ProjectName>App class
  - Add the macro `MRU_DESTROY()` in the destructor of C<ProjectName>App class.
  - Add the macro `IMPLEMENT_MRU_APP_SUPPORT(C<ProjectName>App,CWinApp)` at the bottom.
  - Add the macro `MRU_MESSAGE_MAP_ENTRIES(C<ProjectName>App)` between BEGIN_MESSAGE_MAP(C<ProjectName>App, CWinApp) and END_MESSAGE_MAP()

See Also

Reference

Hierarchy Chart

Other Resources

CRecentFileList Members

CRecentFileList Members

Construction

CRecentFileList	Constructs a **CRecentFileList** object.

Attributes

GetSize	Retrieves the number of files in the MRU file list.

Operations

Add	Adds a file to the MRU file list.
GetDisplayName	Provides a display name for menu display of an MRU filename.
ReadList	Reads the MRU file list from the registry or .INI file.
Remove	Removes a file from the MRU file list.
UpdateMenu	Updates the menu display of the MRU file list.
WriteList	Writes the MRU file list from the registry or .INI file.

Operators

operator[]	Returns a **CString** object at a given position.

See Also

Reference

CRecentFileList Class

Hierarchy Chart

CRecentFileList Member Functions

For information about the member functions in **CRecentFileList**, see CRecentFileList Members.

CRecentFileList::Add

Adds the file whose path is given in *lpszPathName* to the most recently used (MRU) file list.

```
virtual void Add(
   LPCTSTR lpszPathName
);
```

Parameters

lpszPathName

Pathname to be added to the list.

Remarks

The file name will be added to the top of the MRU list. If the file name already exists in the MRU list, it will be moved to the top.

See Also

Reference

CRecentFileList Class
Hierarchy Chart
CRecentFileList::Remove
CRecentFileList::UpdateMenu

Other Resources

CRecentFileList Members

CRecentFileList::CRecentFileList

Constructs a **CRecentFileList** object.

```
CRecentFileList(
   UINT nStart,
   LPCTSTR lpszSection,
   LPCTSTR lpszEntryFormat,
   int nSize,
   int nMaxDispLen = AFX_ABBREV_FILENAME_LEN
);
```

Parameters

nStart

Offset for the numbering in the menu display of the MRU (most recently used) file list.

lpszSection

Points to the name of the section of the registry or the application's .INI file where the MRU file list is read and/or written.

lpszEntryFormat

Points to a format string to be used for the names of the entries stored in the registry or the application's .INI file.

nSize

Maximum number of files in the MRU file list.

nMaxDispLen

Maximum length, in characters, available for the menu display of a filename in the MRU file list.

Remarks

The format string pointed to by *lpszEntryFormat* should contain "%d", which will be used for substituting the index of each MRU item. For example, if the format string is `"file%d"` then the entries will be named `file0`, `file1`, and so on.

See Also

Reference

CRecentFileList Class

Hierarchy Chart

Other Resources

CRecentFileList Members

CRecentFileList::GetDisplayName

Obtains a display name for a file in the MRU file list, for use in the menu display of the MRU list.

```
virtual BOOL GetDisplayName(
   CString& strName,
   int nIndex,
   LPCTSTR lpszCurDir,
   int nCurDir,
   BOOL bAtLeastName = TRUE
) const;
```

Parameters

strName

Full path of the file whose name is to be displayed in the menu list of MRU files.

nIndex

Zero-based index of the file in the MRU file list.

lpszCurDir

String holding the current directory.

nCurDir

Length of the current directory string.

bAtLeastName

If nonzero, indicates that the base name of the file should be returned, even if it exceeds the maximum display length (passed as the *nMaxDispLen* parameter to the **CRecentFileList** constructor).

Return Value

FALSE if there is no filename at the specified index in the most recently used (MRU) file list.

Remarks

If the file is in the current directory, the function leaves the directory off the display. If the filename is too long, the directory and extension are stripped. If the filename is still too long, the display name is set to an empty string unless *bAtLeastName* is nonzero.

See Also

Reference

CRecentFileList Class
Hierarchy Chart
CRecentFileList::ReadList
CRecentFileList::WriteList

Other Resources

CRecentFileList Members

CRecentFileList::GetSize

Retrieves the number of files in the MRU file list.

```
int GetSize( ) const;
```

Return Value

The number of files in the current most recently used (MRU) file list.

See Also

Reference

CRecentFileList Class
Hierarchy Chart
CRecentFileList::Add
CRecentFileList::Remove

Other Resources

CRecentFileList Members

CRecentFileList::ReadList

Reads the most recently used (MRU) file list from the registry or the application's .INI file.

```
virtual void ReadList( );
```

See Also

Reference

CRecentFileList Class

Hierarchy Chart

CRecentFileList::WriteList

Other Resources

CRecentFileList Members

CRecentFileList::Remove

Removes a file from the MRU file list.

```
virtual void Remove(
   int nIndex
);
```

Parameters

nIndex

Zero-based index of the file to be removed from the most recently used (MRU) file list.

See Also

Reference

CRecentFileList Class
Hierarchy Chart
CRecentFileList::Add
CRecentFileList::UpdateMenu

Other Resources

CRecentFileList Members

CRecentFileList::UpdateMenu

Updates the menu display of the MRU file list.

```
virtual void UpdateMenu(
   CCmdUI* pCmdUI
);
```

Parameters

pCmdUI

A pointer to the CCmdUI object for the most recently used (MRU) file list menu.

See Also

Reference

CRecentFileList Class
Hierarchy Chart
CRecentFileList::Add
CRecentFileList::Remove

Other Resources

CRecentFileList Members

CRecentFileList::WriteList

Writes the most recently used (MRU) file list into the registry or the application's .INI file.

```
virtual void WriteList( );
```

See Also

Reference

CRecentFileList Class

Hierarchy Chart

CRecentFileList::ReadList

Other Resources

CRecentFileList Members

CRecentFileList Operators

For information about the operators in **CRecentFileList**, see CRecentFileList Members.

CRecentFileList::operator [ ]

The overloaded subscript (**[]**) operator returns a single **CString** specified by the zero-based index in *nIndex*.

```
CString& operator[ ](
   int nIndex
);
```

Parameters

nIndex

Zero-based index of a **CString** in a set of **CString**s.

See Also

Reference

CRecentFileList Class

Hierarchy Chart

Other Resources

CRecentFileList Members

CRecordset Class

Represents a set of records selected from a data source.

```
class CRecordset : public CObject
```

Remarks

Known as "recordsets," **CRecordset** objects are typically used in two forms: dynasets and snapshots. A dynaset stays synchronized with data updates made by other users. A snapshot is a static view of the data. Each form represents a set of records fixed at the time the recordset is opened, but when you scroll to a record in a dynaset, it reflects changes subsequently made to the record, either by other users or by other recordsets in your application.

Note

If you are working with the Data Access Objects (DAO) classes rather than the Open Database Connectivity (ODBC) classes, use class CDaoRecordset instead. For more information, see the article Overview: Database Programming.

To work with either kind of recordset, you typically derive an application-specific recordset class from **CRecordset**. Recordsets select records from a data source, and you can then:

- Scroll through the records.
- Update the records and specify a locking mode.
- Filter the recordset to constrain which records it selects from those available on the data source.
- Sort the recordset.
- Parameterize the recordset to customize its selection with information not known until run time.

To use your class, open a database and construct a recordset object, passing the constructor a pointer to your **CDatabase** object. Then call the recordset's **Open** member function, where you can specify whether the object is a dynaset or a snapshot. Calling **Open** selects data from the data source. After the recordset object is opened, use its member functions and data members to scroll through the records and operate on them. The operations available depend on whether the object is a dynaset or a snapshot, whether it is updatable or read-only (this depends on the capability of the Open Database Connectivity (ODBC) data source), and whether you have implemented bulk row fetching. To refresh records that may have been changed or added since the **Open** call, call the object's **Requery** member function. Call the object's **Close** member function and destroy the object when you finish with it.

In a derived **CRecordset** class, record field exchange (RFX) or bulk record field exchange (Bulk RFX) is used to support reading and updating of record fields.

For more information about recordsets and record field exchange, see the articles Overview: Database Programming, Recordset (ODBC), Recordset: Fetching Records in Bulk (ODBC), and Record Field Exchange (RFX). For a focus on dynasets and snapshots, see the articles Dynaset and Snapshot.

Requirements

Header: afxdb.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class
Hierarchy Chart
CDatabase Class
CRecordView Class

Other Resources

CRecordset Members

CRecordset Members

Base Class Members

Data Members

Construction

Recordset Attributes

Recordset Update Operations

Recordset Navigation Operations

Other Recordset Operations

Recordset Overridables

Base Class Members

CObject Members

Data Members

m_hstmt	Contains the ODBC statement handle for the recordset. Type **HSTMT**.
m_nFields	Contains the number of field data members in the recordset. Type **UINT**.
m_nParams	Contains the number of parameter data members in the recordset. Type **UINT**.
m_pDatabase	Contains a pointer to the **CDatabase** object through which the recordset is connected to a data source.
m_strFilter	Contains a **CString** that specifies a Structured Query Language (SQL) **WHERE** clause. Used as a filter to select o nly those records that meet certain criteria.
m_strSort	Contains a **CString** that specifies a SQL **ORDER BY** clause. Used to control how the records are sorted.

Construction

Close	Closes the recordset and the ODBC **HSTMT** associated with it.
CRecordset	Constructs a **CRecordset** object. Your derived class must provide a constructor that calls this one.
Open	Opens the recordset by retrieving the table or performing the query that the recordset represents.

Recordset Attributes

CanAppend	Returns nonzero if new records can be added to the recordset via the **AddNew** member function.
CanBookmark	Returns nonzero if the recordset supports bookmarks.
CanRestart	Returns nonzero if **Requery** can be called to run the recordset's query again.
CanScroll	Returns nonzero if you can scroll through the records.
CanTransact	Returns nonzero if the data source supports transactions.
CanUpdate	Returns nonzero if the recordset can be updated (you can add, update, or delete records).
GetODBCFieldCount	Returns the number of fields in the recordset.

GetRecordCount	Returns the number of records in the recordset.
GetSQL	Gets the SQL string used to select records for the recordset.
GetStatus	Gets the status of the recordset: the index of the current record and whether a final count of the records h as been obtained.
GetTableName	Gets the name of the table on which the recordset is based.
IsBOF	Returns nonzero if the recordset has been positioned before the first record. There is no current record.
IsDeleted	Returns nonzero if the recordset is positioned on a deleted record.
IsEOF	Returns nonzero if the recordset has been positioned after the last record. There is no current record.
IsOpen	Returns nonzero if **Open** has been called previously.

Recordset Update Operations

AddNew	Prepares for adding a new record. Call **Update** to complete the addition.
CancelUpdate	Cancels any pending updates due to an **AddNew** or **Edit** operation.
Delete	Deletes the current record from the recordset. You must explicitly scroll to another record after the deletion.
Edit	Prepares for changes to the current record. Call **Update** to complete the edit.
Update	Completes an **AddNew** or **Edit** operation by saving the new or edited data on the data source.

Recordset Navigation Operations

GetBookmark	Assigns the bookmark value of a record to the parameter object.
Move	Positions the recordset to a specified number of records from the current record in either direction.
MoveFirst	Positions the current record on the first record in the recordset. Test for **IsBOF** first.
MoveLast	Positions the current record on the last record or on the last rowset. Test for **IsEOF** first.
MoveNext	Positions the current record on the next record or on the next rowset. Test for **IsEOF** first.
MovePrev	Positions the current record on the previous record or on the previous rowset. Test for **IsBOF** first.
SetAbsolutePosition	Positions the recordset on the record corresponding to the specified record number.
SetBookmark	Positions the recordset on the record specified by the bookmark.

Other Recordset Operations

Cancel	Cancels an asynchronous operation or a process from a second thread.
FlushResultSet	Returns nonzero if there is another result set to be retrieved, when using a predefined query.
GetFieldValue	Returns the value of a field in a recordset.
GetODBCFieldInfo	Returns specific kinds of information about the fields in a recordset.

GetRowsetSize	Returns the number of records you wish to retrieve during a single fetch.
GetRowsFetched	Returns the actual number of rows retrieved during a fetch.
GetRowStatus	Returns the status of the row after a fetch.
IsFieldDirty	Returns nonzero if the specified field in the current record has been changed.
IsFieldNull	Returns nonzero if the specified field in the current record is Null (has no value).
IsFieldNullable	Returns nonzero if the specified field in the current record can be set to Null (having no value).
RefreshRowset	Refreshes the data and status of the specified row(s).
Requery	Runs the recordset's query again to refresh the selected records.
SetFieldDirty	Marks the specified field in the current record as changed.
SetFieldNull	Sets the value of the specified field in the current record to Null (having no value).
SetLockingMode	Sets the locking mode to "optimistic" locking (the default) or "pessimistic" locking. Determines how r ecords are locked for updates.
SetParamNull	Sets the specified parameter to Null (having no value).
SetRowsetCursorPosition	Positions the cursor on the specified row within the rowset.

Recordset Overridables

CheckRowsetError	Called to handle errors generated during record fetching.
DoBulkFieldExchange	Called to exchange bulk rows of data from the data source to the recordset. Implements bulk record field exchange (Bulk RFX).
DoFieldExchange	Called to exchange data (in both directions) between the field data members of the recordset and the cor responding record on the data source. Implements record field exchange (RFX).
GetDefaultConnect	Called to get the default connection string.
GetDefaultSQL	Called to get the default SQL string to execute.
OnSetOptions	Called to set options (used on selection) for the specified ODBC statement.
OnSetUpdateOptions	Called to set options (used on update) for the specified ODBC statement.
SetRowsetSize	Specifies the number of records you wish to retrieve during a fetch.

See Also

Reference

CRecordset Class

Hierarchy Chart

CRecordset Member Functions

For information about the member functions in **CRecordset**, see CRecordset Members.

CRecordset::AddNew

Prepares for adding a new record to the table.

```
virtual void AddNew( );
```

Remarks

You must call the Requery member function to see the newly added record. The record's fields are initially Null. (In database terminology, Null means "having no value" and is not the same as **NULL** in C++.) To complete the operation, you must call the Update member function. **Update** saves your changes to the data source.

Note
If you have implemented bulk row fetching, you cannot call **AddNew**. This will result in a failed assertion. Although class **CRecordset** does not provide a mechanism for updating bulk rows of data, you can write your own functions by using the ODBC API function **SQLSetPos**. For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

AddNew prepares a new, empty record using the recordset's field data members. After you call **AddNew**, set the values you want in the recordset's field data members. (You do not have to call the Edit member function for this purpose; use **Edit** only for existing records.) When you subsequently call **Update**, changed values in the field data members are saved on the data source.

Caution
If you scroll to a new record before you call **Update**, the new record is lost, and no warning is given.

If the data source supports transactions, you can make your **AddNew** call part of a transaction. For more information about transactions, see class CDatabase. Note that you should call CDatabase::BeginTrans before calling **AddNew**.

Note
For dynasets, new records are added to the recordset as the last record. Added records are not added to snapshots; you must call **Requery** to refresh the recordset.

It is illegal to call **AddNew** for a recordset whose **Open** member function has not been called. A **CDBException** is thrown if you call **AddNew** for a recordset that cannot be appended to. You can determine whether the recordset is updatable by calling CanAppend.

For more information, see the following articles: Recordset: How Recordsets Update Records (ODBC), Recordset: Adding, Updating, and Deleting Records (ODBC), and Transaction (ODBC).

Exceptions

This method can throw exceptions of type **CDBException***.

Example

See the article Transaction: Performing a Transaction in a Recordset (ODBC).

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::Edit
CRecordset::Delete
CRecordset::Update
CRecordset::Requery
CDatabase::BeginTrans
CDBException Class

Other Resources

CRecordset Members

CRecordset::CanAppend

Determines whether the previously opened recordset allows you to add new records.

```
BOOL CanAppend( ) const;
```

Return Value

Nonzero if the recordset allows adding new records; otherwise 0. **CanAppend** will return 0 if you opened the recordset as read-only.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::AddNew
CRecordset::Requery

Other Resources

CRecordset Members

CRecordset::CanBookmark

Determines whether the recordset allows you to mark records using bookmarks.

```
BOOL CanBookmark( ) const;
```

Return Value

Nonzero if the recordset supports bookmarks; otherwise 0.

Remarks

This function is independent of the **CRecordset::useBookmarks** option in the *dwOptions* parameter of the Open member function. **CanBookmark** indicates whether the given ODBC driver and cursor type support bookmarks. **CRecordset::useBookmarks** indicates whether bookmarks will be available, provided they are supported.

Note
Bookmarks are not supported on forward-only recordsets.

For more information about bookmarks and recordset navigation, see the articles Recordset: Bookmarks and Absolute Positions (ODBC) and Recordset: Scrolling (ODBC).

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::GetBookmark
CRecordset::SetBookmark

Other Resources

CRecordset Members

CRecordset::Cancel

Requests that the data source cancel either an asynchronous operation in progress or a process from a second thread.

```
void Cancel( );
```

Remarks

Note that the MFC ODBC classes no longer use asynchronous processing; to perform an asychronous operation, you must directly call the ODBC API function **SQLSetConnectOption**. For more information, see the topic "Executing Functions Asynchronously" in the *ODBC SDK Programmer's Guide*.

See Also

Reference

CRecordset Class

Hierarchy Chart

Other Resources

CRecordset Members

CRecordset::CancelUpdate

Cancels any pending updates, caused by an Edit or AddNew operation, before Update is called.

```
void CancelUpdate( );
```

Remarks

Note

This member function is not applicable on recordsets that are using bulk row fetching, since such recordsets cannot call **Edit**, **AddNew**, or **Update**. For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

If automatic dirty field checking is enabled, **CancelUpdate** will restore the member variables to the values they had before **Edit** or **AddNew** was called; otherwise, any value changes will remain. By default, automatic field checking is enabled when the recordset is opened. To disable it, you must specify the **CRecordset::noDirtyFieldCheck** in the *dwOptions* parameter of the Open member function.

For more information about updating data, see the article Recordset: Adding, Updating, and Deleting Records (ODBC).

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::AddNew
CRecordset::Edit
CRecordset::Update

Other Resources

CRecordset Members

CRecordset::CanRestart

Determines whether the recordset allows restarting its query (to refresh its records) by calling the **Requery** member function.

```
BOOL CanRestart( ) const;
```

Return Value

Nonzero if requery is allowed; otherwise 0.

See Also

Reference

CRecordset Class

Hierarchy Chart

CRecordset::Requery

Other Resources

CRecordset Members

CRecordset::CanScroll

Determines whether the recordset allows scrolling.

```
BOOL CanScroll( ) const;
```

Return Value

Nonzero if the recordset allows scrolling; otherwise 0.

Remarks

For more information about scrolling, see the article Recordset: Scrolling (ODBC).

See Also

Reference

CRecordset Class

Hierarchy Chart

Other Resources

CRecordset Members

CRecordset::CanTransact

Determines whether the recordset allows transactions.

```
BOOL CanTransact( ) const;
```

Return Value

Nonzero if the recordset allows transactions; otherwise 0.

Remarks

For more information, see the article Transaction (ODBC).

See Also

Reference

CRecordset Class
Hierarchy Chart
CDatabase::BeginTrans
CDatabase::CommitTrans
CDatabase::Rollback

Other Resources

CRecordset Members

CRecordset::CanUpdate

Determines whether the recordset can be updated.

```
BOOL CanUpdate( ) const;
```

Return Value

Nonzero if the recordset can be updated; otherwise 0.

Remarks

A recordset might be read-only if the underlying data source is read-only or if you specified **CRecordset::readOnly** in the *dwOptions* parameter when you opened the recordset.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::Open
CRecordset::AddNew
CRecordset::Edit
CRecordset::Delete
CRecordset::Update

Other Resources

CRecordset Members

CRecordset::CheckRowsetError

Called to handle errors generated during record fetching.

```
virtual void CheckRowsetError(
   RETCODE nRetCode
);
```

Parameters

nRetCode

An ODBC API function return code. For details, see Remarks.

Remarks

This virtual member function handles errors that occur when records are fetched, and is useful during bulk row fetching. You may want to consider overriding **CheckRowsetError** to implement your own error handling.

CheckRowsetError is called automatically in a cursor navigation operation, such as **Open**, **Requery**, or any **Move** operation. It is passed the return value of the ODBC API function **SQLExtendedFetch**. The following table lists the possible values for the *nRetCode* parameter.

nRetCode	Description
SQL_SUCCESS	Function completed successfully; no additional information is available.
SQL_SUCCESS_WITH_INFO	Function completed successfully, possibly with a nonfatal error. Additional information can be obtained by calling **SQLError**.
SQL_NO_DATA_FOUND	All rows from the result set have been fetched.
SQL_ERROR	Function failed. Additional information can be obtained by calling **SQLError**.
SQL_INVALID_HANDLE	Function failed due to an invalid environment handle, connection handle, or statement handle. This indicates a programming error. No additional information is available from **SQLError**.
SQL_STILL_EXECUTING	A function that was started asynchronously is still executing. Note that by default, MFC will never pass this value to **CheckRowsetError**; MFC will continue calling **SQLExtendedFetch** until it no longer returns **SQL_STILL_EXECUTING**.

For more information about **SQLError**, see the *Platform SDK*. For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

Exceptions

This method can throw exceptions of type **CDBException***.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::DoBulkFieldExchange
CRecordset::GetRowsetSize
CRecordset::SetRowsetSize
CRecordset::Move

Other Resources

CRecordset Members

CRecordset::Close

Closes the recordset.

```
virtual void Close( );
```

Remarks

The ODBC **HSTMT** and all memory the framework allocated for the recordset are deallocated. Usually after calling **Close**, you delete the C++ recordset object if it was allocated with **new**.

You can call **Open** again after calling **Close**. This lets you reuse the recordset object. The alternative is to call **Requery**.

Example

```
// Example for CRecordset::Close

// Construct a snapshot object
CCustSet rsCustSet( NULL );

if( !rsCustSet.Open( ) )
   return FALSE;

// Use the snapshot ...

// Close the snapshot
rsCustSet.Close( );

// Destructor is called when the function exits
```

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::CRecordset
CRecordset::Open
CRecordset::Requery

Other Resources

CRecordset Members

CRecordset::CRecordset

Constructs a **CRecordset** object.

```
CRecordset(
   CDatabase* pDatabase = NULL
);
```

Parameters

pDatabase

Contains a pointer to a **CDatabase** object or the value **NULL**. If not **NULL** and the **CDatabase** object's **Open** member function has not been called to connect it to the data source, the recordset attempts to open it for you during its own **Open** call. If you pass **NULL**, a **CDatabase** object is constructed and connected for you using the data source information you specified when you derived your recordset class with ClassWizard.

Remarks

You can either use **CRecordset** directly or derive an application-specific class from **CRecordset**. You can use ClassWizard to derive your recordset classes.

Note
A derived class *must* supply its own constructor. In the constructor of your derived class, call the constructor **CRecordset::CRecordset**, passing the appropriate parameters along to it.

Pass **NULL** to your recordset constructor to have a **CDatabase** object constructed and connected for you automatically. This is a useful shorthand that does not require you to construct and connect a **CDatabase** object prior to constructing your recordset.

Example

For more information, see the article Recordset: Declaring a Class for a Table (ODBC).

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::Open
CRecordset::Close

Other Resources

CRecordset Members

CRecordset::Delete

Deletes the current record.

```
virtual void Delete( );
```

Remarks

After a successful deletion, the recordset's field data members are set to a Null value, and you must explicitly call one of the **Move** functions in order to move off the deleted record. Once you move off the deleted record, it is not possible to return to it. If the data source supports transactions, you can make the **Delete** call part of a transaction. For more information, see the article Transaction (ODBC).

Note

If you have implemented bulk row fetching, you cannot call **Delete**. This will result in a failed assertion. Although class **CRecordset** does not provide a mechanism for updating bulk rows of data, you can write your own functions by using the ODBC API function **SQLSetPos**. For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

Caution

The recordset must be updatable and there must be a valid record current in the recordset when you call **Delete**; otherwise, an error occurs. For example, if you delete a record but do not scroll to a new record before you call **Delete** again, **Delete** throws a CDBException.

Unlike AddNew and Edit, a call to **Delete** is not followed by a call to Update. If a **Delete** call fails, the field data members are left unchanged.

Exceptions

This method can throw exceptions of type **CDBException***.

Example

This example shows a recordset created on the frame of a function. The example assumes the existence of `m_dbCust`, a member variable of type **CDatabase** already connected to the data source.

```
// Create a derived CRecordset object
CCustSet rsCustSet( &m_dbCust );
rsCustSet.Open( );

if( rsCustSet.IsEOF( ) || !rsCustSet.CanUpdate( ) ||
    !rsCustSet.CanTransact( ) )
   return;

if( !m_dbCust.BeginTrans( ) )
{
   // Do something to handle a failure
}
else
{
   // Perhaps scroll to a new record...
   // Delete the current record
   rsCustSet.Delete( );
   // ...

   // Finished commands for this transaction
   if( <the user confirms the transaction> )
      m_dbCust.CommitTrans( );
   else // User changed mind
      m_dbCust.Rollback( );
```

```
}
// ...
```

See Also

Reference

CRecordset Class
Hierarchy Chart
CDatabase::BeginTrans
CDatabase::CommitTrans
CDatabase::Rollback
CDBException Class

Other Resources

CRecordset Members

CRecordset::DoBulkFieldExchange

Called to exchange bulk rows of data from the data source to the recordset. Implements bulk record field exchange (Bulk RFX).

```
virtual void DoBulkFieldExchange(
   CFieldExchange* pFX
);
```

Parameters

pFX

A pointer to a CFieldExchange object. The framework will already have set up this object to specify a context for the field exchange operation.

Remarks

When bulk row fetching is implemented, the framework calls this member function to automatically transfer data from the data source to your recordset object. **DoBulkFieldExchange** also binds your parameter data members, if any, to parameter placeholders in the SQL statement string for the recordset's selection.

If bulk row fetching is not implemented, the framework calls DoFieldExchange. To implement bulk row fetching, you must specify the **CRecordset::useMultiRowFetch** option of the *dwOptions* parameter in the Open member function.

Note
DoBulkFieldExchange is available only if you are using a class derived from **CRecordset**. If you have created a recordset object directly from **CRecordset**, you must call the GetFieldValue member function to retrieve data.

Bulk record field exchange (Bulk RFX) is similar to record field exchange (RFX). Data is automatically transferred from the data source to the recordset object. However, you cannot call **AddNew**, **Edit**, **Delete**, or **Update** to transfer changes back to the data source. Class **CRecordset** currently does not provide a mechanism for updating bulk rows of data; however, you can write your own functions by using the ODBC API function **SQLSetPos**.

Note that ClassWizard does not support bulk record field exchange; therefore, you must override **DoBulkFieldExchange** manually by writing calls to the Bulk RFX functions. For more information about these functions, see the topic Record Field Exchange Functions.

For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC). For related information, see the article Record Field Exchange (RFX).

Exceptions

This method can throw exceptions of type **CDBException***.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::m_nFields
CRecordset::m_nParams
CRecordset::DoFieldExchange
CRecordset::GetFieldValue
CFieldExchange Class

Other Resources

CRecordset Members
Record Field Exchange Functions

CRecordset::DoFieldExchange

Called to exchange data (in both directions) between the field data members of the recordset and the corresponding record on the data source. Implements record field exchange (RFX).

```
virtual void DoFieldExchange(
   CFieldExchange* pFX
);
```

Parameters

pFX

A pointer to a CFieldExchange object. The framework will already have set up this object to specify a context for the field exchange operation.

Remarks

When bulk row fetching is not implemented, the framework calls this member function to automatically exchange data between the field data members of your recordset object and the corresponding columns of the current record on the data source. **DoFieldExchange** also binds your parameter data members, if any, to parameter placeholders in the SQL statement string for the recordset's selection.

If bulk row fetching is implemented, the framework calls DoBulkFieldExchange. To implement bulk row fetching, you must specify the **CRecordset::useMultiRowFetch** option of the *dwOptions* parameter in the Open member function.

Note
DoFieldExchange is available only if you are using a class derived from **CRecordset**. If you have created a recordset object directly from **CRecordset**, you must call the GetFieldValue member function to retrieve data.

The exchange of field data, called record field exchange (RFX), works in both directions: from the recordset object's field data members to the fields of the record on the data source, and from the record on the data source to the recordset object.

The only action you must normally take to implement **DoFieldExchange** for your derived recordset class is to create the class with ClassWizard and specify the names and data types of the field data members. You might also add code to what ClassWizard writes to specify parameter data members or to deal with any columns you bind dynamically. For more information, see the article Recordset: Dynamically Binding Data Columns (ODBC).

When you declare your derived recordset class with ClassWizard, the wizard writes an override of **DoFieldExchange** for you, which resembles the following example:

```
void CCustSet::DoFieldExchange(CFieldExchange* pFX)
{
   //{{AFX_FIELD_MAP(CCustSet)
   pFX->SetFieldType(CFieldExchange::outputColumn);
   RFX_Text(pFX, "Name", m_strName);
   RFX_Int(pFX, "Age", m_wAge);
   //}}AFX_FIELD_MAP
}
```

For more information about the RFX functions, see the topic Record Field Exchange Functions.

For further examples and details about **DoFieldExchange**, see the article Record Field Exchange: How RFX Works. For general information about RFX, see the article Record Field Exchange.

Exceptions

This method can throw exceptions of type **CDBException***.

See Also

Reference

CRecordset Class

Hierarchy Chart

CRecordset::m_nFields
CRecordset::m_nParams
CRecordset::DoBulkFieldExchange
CRecordset::GetFieldValue
CFieldExchange Class

Other Resources

CRecordset Members
Record Field Exchange Functions

CRecordset::Edit

Allows changes to the current record.

```
virtual void Edit( );
```

Remarks

After you call **Edit**, you can change the field data members by directly resetting their values. The operation is completed when you subsequently call the Update member function to save your changes on the data source.

Note

If you have implemented bulk row fetching, you cannot call **Edit**. This will result in a failed assertion. Although class **CRecordset** does not provide a mechanism for updating bulk rows of data, you can write your own functions by using the ODBC API function **SQLSetPos**. For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

Edit saves the values of the recordset's data members. If you call **Edit**, make changes, then call **Edit** again, the record's values are restored to what they were before the first **Edit** call.

In some cases, you may want to update a column by making it Null (containing no data). To do so, call SetFieldNull with a parameter of **TRUE** to mark the field Null; this also causes the column to be updated. If you want a field to be written to the data source even though its value has not changed, call SetFieldDirty with a parameter of **TRUE**. This works even if the field had the value Null.

If the data source supports transactions, you can make the **Edit** call part of a transaction. Note that you should call CDatabase::BeginTrans before calling **Edit** and after the recordset has been opened. Also note that calling CDatabase::CommitTrans is not a substitute for calling **Update** to complete the **Edit** operation. For more information about transactions, see class CDatabase.

Depending on the current locking mode, the record being updated may be locked by **Edit** until you call **Update** or scroll to another record, or it may be locked only during the **Edit** call. You can change the locking mode with SetLockingMode.

The previous value of the current record is restored if you scroll to a new record before calling **Update**. A **CDBException** is thrown if you call **Edit** for a recordset that cannot be updated or if there is no current record.

For more information, see the articles Transaction (ODBC) and Recordset: Locking Records (ODBC).

Exceptions

This method can throw exceptions of type **CDBException*** and **CMemoryException***.

Example

```
// Example for CRecordset::Edit
// To edit a record,
// First set up the edit buffer
rsCustSet.Edit( );

// Then edit field data members for the record
rsCustSet.m_dwCustID = 2795;
rsCustSet.m_strCustomer = "Jones Mfg";

// Finally, complete the operation
if( !rsCustSet.Update( ) )
    // Handle the failure to update
```

See Also

Reference

CRecordset Class

Hierarchy Chart

CRecordset::Update

CRecordset::AddNew
CRecordset::Delete
CRecordset::SetFieldDirty
CRecordset::SetFieldNull
CRecordset::CanUpdate
CRecordset::CanTransact
CRecordset::SetLockingMode

Other Resources

CRecordset Members

CRecordset::FlushResultSet

Retrieves the next result set of a predefined query (stored procedure), if there are multiple result sets.

```
BOOL FlushResultSet( );
```

Return Value

Nonzero if there are more result sets to be retrieved; otherwise 0.

Remarks

You should call **FlushResultSet** only when you are completely finished with the cursor on the current result set. Note that when you retrieve the next result set by calling **FlushResultSet**, your cursor is not valid on that result set; you should call the MoveNext member function after calling **FlushResultSet**.

If a predefined query uses an output parameter or input/output parameters, you must call **FlushResultSet** until it returns **FALSE** (the value 0), in order to obtain these parameter values.

FlushResultSet calls the ODBC API function **SQLMoreResults**. If **SQLMoreResults** returns **SQL_ERROR** or **SQL_INVALID_HANDLE**, then **FlushResultSet** will throw an exception. For more information about **SQLMoreResults**, see the *Platform SDK*.

In Visual C++ 2005 your stored procedure needs to have bound fields if you want to call **FlushResultSet**.

Exceptions

This method can throw exceptions of type **CDBException***.

Example

The following code assumes that `COutParamRecordset` is a **CRecordset**-derived object based on a predefined query with an input parameter and an output parameter, and having multiple result sets. Note the structure of the DoFieldExchange override.

```
// DoFieldExchange override
//
// Only necessary to handle parameter bindings.
// Don't use CRecordset-derived class with bound
// fields unless all result sets have same schema
// OR there is conditional binding code.

void COutParamRecordset::DoFieldExchange( CFieldExchange* pFX )
{
   pFX->SetFieldType( CFieldExchange::outputParam );
   RFX_Long( pFX, "Param1", m_nOutParamInstructorCount );
         // The "Param1" name here is a dummy name
         // that is never used

   pFX->SetFieldType( CFieldExchange::inputParam );
   RFX_Text( pFX, "Param2", m_strInParamName );
         // The "Param2" name here is a dummy name
         // that is never used

}


// Now implement COurParamRecordset.

// Assume db is an already open CDatabase object
COutParamRecordset rs( &db );
rs.m_strInParamName = _T("Some_Input_Param_Value");

// Get the first result set
// NOTE: SQL Server requires forwardOnly cursor
//       type for multiple rowset returning stored
```

```
//       procedures
rs.Open( CRecordset::forwardOnly,
         "{? = CALL GetCourses( ? )}",
         CRecordset::readOnly);

// Loop through all the data in the first result set
while ( !rs.IsEOF( ) )
{
   CString strFieldValue;
   for( int nIndex = 0;
        nIndex < rs.GetODBCFieldCount( );
        nIndex++ )
   {
      rs.GetFieldValue( nIndex, strFieldValue );

      // TO DO: Use field value string.
   }
   rs.MoveNext( );
}

// Retrieve other result sets...
while( rs.FlushResultSet( ) )
{
   // must call MoveNext because cursor is invalid
   rs.MoveNext( );

   while ( !rs.IsEOF( ) )
   {
      CString strFieldValue;
      for( int nIndex = 0;
           nIndex < rs.GetODBCFieldCount( );
           nIndex++ )
      {
         rs.GetFieldValue( nIndex, strFieldValue );

         // TO DO: Use field value string.
      }
      rs.MoveNext( );
   }
}


// All result sets have been flushed. Cannot
// use the cursor, but the output parameter,
// m_nOutParamInstructorCount, has now been written.
// Note that m_nOutParamInstructorCount not valid until
// CRecordset::FlushResultSet has returned FALSE,
// indicating no more result sets will be returned.

// TO DO: Use m_nOutParamInstructorCount

// Cleanup
rs.Close( );
db.Close( );
```

See Also

Reference

CRecordset Class

Hierarchy Chart

CFieldExchange::SetFieldType

Other Resources

CRecordset Members

CRecordset::GetBookmark

Obtains the bookmark value for the current record.

```
void GetBookmark(
   CDBVariant& varBookmark
);
```

Parameters

varBookmark

A reference to a CDBVariant object representing the bookmark on the current record.

Remarks

To determine if bookmarks are supported on the recordset, call CanBookmark. To make bookmarks available if they are supported, you must set the **CRecordset::useBookmarks** option in the *dwOptions* parameter of the Open member function.

Note
If bookmarks are unsupported or unavailable, calling **GetBookmark** will result in an exception being thrown. Bookmarks are not supported on forward-only recordsets.

GetBookmark assigns the value of the bookmark for the current record to a **CDBVariant** object. To return to that record at any time after moving to a different record, call SetBookmark with the corresponding **CDBVariant** object.

Note
After certain recordset operations, bookmarks may no longer be valid. For example, if you call **GetBookmark** followed by **Requery**, you may not be able to return to the record with **SetBookmark**. Call CDatabase::GetBookmarkPersistence to check whether you can safely call **SetBookmark**.

For more information about bookmarks and recordset navigation, see the articles Recordset: Bookmarks and Absolute Positions (ODBC) and Recordset: Scrolling (ODBC).

Exceptions

This method can throw exceptions of type **CDBException*** and **CMemoryException***.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::CanBookmark
CRecordset::SetBookmark
CDatabase::GetBookmarkPersistence

Other Resources

CRecordset Members

CRecordset::GetDefaultConnect

Called to get the default connection string.

```
virtual CString GetDefaultConnect( );
```

Return Value

A **CString** that contains the default connection string.

Remarks

The framework calls this member function to get the default connection string for the data source on which the recordset is based. ClassWizard implements this function for you by identifying the same data source you use in ClassWizard to get information about tables and columns. You will probably find it convenient to rely on this default connection while developing your application. But the default connection may not be appropriate for users of your application. If that is the case, you should reimplement this function, discarding ClassWizard's version. For more information about connection strings, see the article Data Source (ODBC).

See Also

Reference

CRecordset Class

Hierarchy Chart

Other Resources

CRecordset Members

CRecordset::GetDefaultSQL

Called to get the default SQL string to execute.

```
virtual CString GetDefaultSQL( );
```

Return Value

A **CString** that contains the default SQL statement.

Remarks

The framework calls this member function to get the default SQL statement on which the recordset is based. This might be a table name or a SQL **SELECT** statement.

You indirectly define the default SQL statement by declaring your recordset class with ClassWizard, and ClassWizard performs this task for you.

If you need the SQL statement string for your own use, call **GetSQL**, which returns the SQL statement used to select the recordset's records when it was opened. You can edit the default SQL string in your class's override of **GetDefaultSQL**. For example, you could specify a call to a predefined query using a **CALL** statement. (Note, however, that if you edit **GetDefaultSQL**, you also need to modify *m_nFields* to match the number of columns in the data source.)

For more information, see the article Recordset: Declaring a Class for a Table (ODBC).

Caution
The table name will be empty if the framework could not identify a table name, if multiple table names were supplied, or if a **CALL** statement could not be interpreted. Note that when using a **CALL** statement, you must not insert whitespace between the curly brace and the **CALL** keyword, nor should you insert whitespace before the curly brace or before the **SELECT** keyword in a **SELECT** statement.

See Also

Reference

CRecordset Class

Hierarchy Chart

CRecordset::GetSQL

Other Resources

CRecordset Members

CRecordset::GetFieldValue

Retrieves field data in the current record.

```
void GetFieldValue(
   LPCTSTR lpszName,
   CDBVariant& varValue,
   short nFieldType = DEFAULT_FIELD_TYPE
);
void GetFieldValue(
   short nIndex,
   CDBVariant& varValue,
   short nFieldType = DEFAULT_FIELD_TYPE
);
void GetFieldValue(
   short nIndex,
   CStringA& strValue
);
void GetFieldValue(
   short nIndex,
   CStringW& strValue
);
```

Parameters

lpszName

The name of a field.

varValue

A reference to a CDBVariant object that will store the field's value.

nFieldType

The ODBC C data type of the field. Using the default value, **DEFAULT_FIELD_TYPE**, forces **GetFieldValue** to determine the C data type from the SQL data type, based on the following table. Otherwise, you can specify the data type directly or choose a compatible data type; for example, you can store any data type into **SQL_C_CHAR**.

C data type	SQL data type
SQL_C_BIT	**SQL_BIT**
SQL_C_UTINYINT	**SQL_TINYINT**
SQL_C_SSHORT	**SQL_SMALLINT**
SQL_C_SLONG	**SQL_INTEGER**
SQL_C_FLOAT	**SQL_REAL**
SQL_C_DOUBLE	**SQL_FLOATSQL_DOUBLE**
SQL_C_TIMESTAMP	**SQL_DATESQL_TIMESQL_TIMESTAMP**
SQL_C_CHAR	**SQL_NUMERICSQL_DECIMALSQL_BIGINTSQL_CHARSQL_VARCHARSQL_LONGVARCHAR**
SQL_C_BINARY	**SQL_BINARYSQL_VARBINARYSQL_LONGVARBINARY**

For more information about ODBC data types, see the topics "SQL Data Types" and "C Data Types" in Appendix D of the *Platform SDK*.

nIndex

The zero-based index of the field.

strValue

A reference to a CString object that will store the field's value converted to text, regardless of the field's data type.

Remarks

You can look up a field either by name or by index. You can store the field value in either a **CDBVariant** object or a **CString** object.

If you have implemented bulk row fetching, the current record is always positioned on the first record in a rowset. To use **GetFieldValue** on a record within a given rowset, you must first call the SetRowsetCursorPosition member function to move the cursor to the desired row within that rowset. Then call **GetFieldValue** for that row. To implement bulk row fetching, you must specify the **CRecordset::useMultiRowFetch** option of the *dwOptions* parameter in the Open member function.

You can use **GetFieldValue** to dynamically fetch fields at run time rather than statically binding them at design time. For example, if you have declared a recordset object directly from **CRecordset**, you must use **GetFieldValue** to retrieve the field data; record field exchange (RFX), or bulk record field exchange (Bulk RFX), is not implemented.

Note
If you declare a recordset object without deriving from **CRecordset**, do not have the ODBC Cursor Library loaded. The cursor library requires that the recordset have at least one bound column; however, when you use **CRecordset** directly, none of the columns are bound. The member functions CDatabase::OpenEx and CDatabase::Open control whether the cursor library will be loaded.

GetFieldValue calls the ODBC API function **SQLGetData**. If your driver outputs the value **SQL_NO_TOTAL** for the actual length of the field value, **GetFieldValue** throws an exception. For more information about **SQLGetData**, see the *Platform SDK*.

Exceptions

This method can throw exceptions of type **CDBException*** and **CMemoryException***.

Example

The following sample code illustrates calls to **GetFieldValue** for a recordset object declared directly from **CRecordset**.

```
// Create and open a database object;
// do not load the cursor library
CDatabase db;
db.OpenEx( NULL, CDatabase::forceOdbcDialog );

// Create and open a recordset object
// directly from CRecordset. Note that a
// table must exist in a connected database.
// Use forwardOnly type recordset for best
// performance, since only MoveNext is required
CRecordset rs( &db );
rs.Open( CRecordset::forwardOnly,
         _T( "SELECT * FROM SomeTable" ) );

// Create a CDBVariant object to
// store field data
CDBVariant varValue;

// Loop through the recordset,
// using GetFieldValue and
// GetODBCFieldCount to retrieve
// data in all columns
short nFields = rs.GetODBCFieldCount( );
while( !rs.IsEOF( ) )
{
   for( short index = 0; index < nFields; index++ )
   {
      rs.GetFieldValue( index, varValue );
      // do something with varValue
```

```
    }
    rs.MoveNext( );
}

rs.Close( );
db.Close( );
```

Note

Unlike the DAO class **CDaoRecordset**, **CRecordset** does not have a **SetFieldValue** member function. If you create an object directly from **CRecordset**, it is effectively read-only.

For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::DoFieldExchange
CRecordset::DoBulkFieldExchange
CRecordset::GetODBCFieldCount
CRecordset::GetODBCFieldInfo
CRecordset::SetRowsetCursorPosition

Other Resources

CRecordset Members

CRecordset::GetODBCFieldCount

Retrieves the total number of fields in your recordset object.

```
short GetODBCFieldCount( ) const;
```

Return Value

The number of fields in the recordset.

Remarks

For more information about creating recordsets, see the article Recordset: Creating and Closing Recordsets (ODBC).

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::GetFieldValue

Other Resources

CRecordset Members

CRecordset::GetODBCFieldInfo

Obtains information about the fields in the recordset.

```
void GetODBCFieldInfo(
   LPCTSTR lpszName,
   CODBCFieldInfo& fieldinfo
);
void GetODBCFieldInfo(
   short nIndex,
   CODBCFieldInfo& fieldinfo
);
```

Parameters

lpszName

The name of a field.

fieldinfo

A reference to a **CODBCFieldInfo** structure.

nIndex

The zero-based index of the field.

Remarks

One version of the function lets you look up a field by name. The other version lets you look up a field by index.

For a description about the information returned, see the CODBCFieldInfo structure.

For more information about creating recordsets, see the article Recordset: Creating and Closing Recordsets (ODBC).

Exceptions

This method can throw exceptions of type **CDBException***.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::GetFieldValue
CODBCFieldInfo Structure

Other Resources

CRecordset Members

CRecordset::GetRecordCount

Determines the size of the recordset.

```
long GetRecordCount( ) const;
```

Return Value

The number of records in the recordset; 0 if the recordset contains no records; or –1 if the record count cannot be determined.

Remarks

Caution

The record count is maintained as a "high water mark," the highest-numbered record yet seen as the user moves through the records. The total number of records is only known after the user has moved beyond the last record. For performance reasons, the count is not updated when you call **MoveLast**. To count the records yourself, call **MoveNext** repeatedly until **IsEOF** returns nonzero. Adding a record via **CRecordset:AddNew** and **Update** increases the count; deleting a record via **CRecordset::Delete** decreases the count.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::MoveLast
CRecordset::MoveNext
CRecordset::IsEOF
CRecordset::GetStatus

Other Resources

CRecordset Members

CRecordset::GetRowsetSize

Obtains the current setting for the number of rows you wish to retrieve during a given fetch.

```
DWORD GetRowsetSize( ) const;
```

Return Value

The number of rows to retrieve during a given fetch.

Remarks

If you are using bulk row fetching, the default rowset size when the recordset is opened is 25; otherwise, it is 1.

To implement bulk row fetching, you must specify the **CRecordset::useMultiRowFetch** option in the *dwOptions* parameter of the Open member function. To change the setting for the rowset size, call SetRowsetSize.

For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::Open
CRecordset::SetRowsetSize
CRecordset::CheckRowsetError
CRecordset::DoBulkFieldExchange

Other Resources

CRecordset Members

CRecordset::GetRowsFetched

Determines how many records were actually retrieved after a fetch.

```
DWORD GetRowsFetched( ) const;
```

Return Value

The number of rows retrieved from the data source after a given fetch.

Remarks

This is useful when you have implemented bulk row fetching. The rowset size normally indicates how many rows will be retrieved from a fetch; however, the total number of rows in the recordset also affects how many rows will be retrieved in a rowset. For example, if your recordset has 10 records with a rowset size setting of 4, then looping through the recordset by calling **MoveNext** will result in the final rowset having only 2 records.

To implement bulk row fetching, you must specify the **CRecordset::useMultiRowFetch** option in the *dwOptions* parameter of the Open member function. To specify the rowset size, call SetRowsetSize.

For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

Example

```
MultiRowSet rs;

// Set the rowset size
rs.SetRowsetSize( 5 );

// Open the recordset
rs.Open( CRecordset::dynaset, NULL,
         CRecordset::useMultiRowFetch );

// loop through the recordset by rowsets
while( !rs.IsEOF( ) )
{
    for( int rowCount = 0;
         rowCount < (int)rs.GetRowsFetched( );
         rowCount++ )
    {
       // do something
    }

   rs.MoveNext( );
}

rs.Close( );
```

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::SetRowsetSize
CRecordset::CheckRowsetError

Other Resources

CRecordset Members

CRecordset::GetRowStatus

Obtains the status for a row in the current rowset.

```
WORD GetRowStatus(
   WORD wRow
) const;
```

Parameters

wRow

The one-based position of a row in the current rowset. This value can range from 1 to the size of the rowset.

Return Value

A status value for the row. For details, see Remarks.

Remarks

GetRowStatus returns a value that indicates either any change in status to the row since it was last retrieved from the data source, or that no row corresponding to *wRow* was fetched. The following table lists the possible return values.

Status value	Description
SQL_ROW_SUCCESS	The row is unchanged.
SQL_ROW_UPDATED	The row has been updated.
SQL_ROW_DELETED	The row has been deleted.
SQL_ROW_ADDED	The row has been added.
SQL_ROW_ERROR	The row is unretrievable due to an error.
SQL_ROW_NOROW	There is no row that corresponds to *wRow*.

For more information, see the ODBC API function **SQLExtendedFetch** in the *Platform SDK*.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::CheckRowsetError
CRecordset::GetRowsFetched
CRecordset::RefreshRowset

Other Resources

CRecordset Members

CRecordset::GetStatus

Determines the index of the current record in the recordset and whether the last record has been seen.

```
void GetStatus(
   CRecordsetStatus& rStatus
) const;
```

Parameters

rStatus

A reference to a **CRecordsetStatus** object. See the Remarks section for more information.

Remarks

CRecordset attempts to track the index, but under some circumstances this may not be possible. See GetRecordCount for an explanation.

The **CRecordsetStatus** structure has the following form:

```
struct CRecordsetStatus
{
    long m_lCurrentRecord;
    BOOL m_bRecordCountFinal;
};
```

The two members of **CRecordsetStatus** have the following meanings:

- **m_lCurrentRecord** Contains the zero-based index of the current record in the recordset, if known. If the index cannot be determined, this member contains **AFX_CURRENT_RECORD_UNDEFINED** (–2). If **IsBOF** is **TRUE** (empty recordset or attempt to scroll before first record), then **m_lCurrentRecord** is set to **AFX_CURRENT_RECORD_BOF** (–1). If on the first record, then it is set to 0, second record 1, and so on.
- **m_bRecordCountFinal** Nonzero if the total number of records in the recordset has been determined. Generally this must be accomplished by starting at the beginning of the recordset and calling **MoveNext** until **IsEOF** returns nonzero. If this member is zero, the record count as returned by **GetRecordCount**, if not –1, is only a "high water mark" count of the records.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::GetRecordCount

Other Resources

CRecordset Members

CRecordset::GetSQL

Call this member function to get the SQL statement that was used to select the recordset's records when it was opened.

```
const CString& GetSQL( ) const;
```

Return Value

A **const** reference to a **CString** that contains the SQL statement.

Remarks

This will generally be a SQL **SELECT** statement. The string returned by **GetSQL** is read-only.

The string returned by **GetSQL** is typically different from any string you may have passed to the recordset in the *lpszSQL* parameter to the **Open** member function. This is because the recordset constructs a full SQL statement based on what you passed to **Open**, what you specified with ClassWizard, what you may have specified in the **m_strFilter** and **m_strSort** data members, and any parameters you may have specified. For details about how the recordset constructs this SQL statement, see the article Recordset: How Recordsets Select Records (ODBC).

Note
Call this member function only after calling Open.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::GetDefaultSQL
CRecordset::Open
CRecordset::m_strFilter
CRecordset::m_strSort

Other Resources

CRecordset Members

CRecordset::GetTableName

Gets the name of the SQL table on which the recordset's query is based.

```
const CString& GetTableName( ) const;
```

Return Value

A **const** reference to a **CString** that contains the table name, if the recordset is based on a table; otherwise, an empty string.

Remarks

GetTableName is only valid if the recordset is based on a table, not a join of multiple tables or a predefined query (stored procedure). The name is read-only.

Note
Call this member function only after calling Open.

See Also

Reference

CRecordset Class

Hierarchy Chart

Other Resources

CRecordset Members

CRecordset::IsBOF

Returns nonzero if the recordset has been positioned before the first record. There is no current record.

```
BOOL IsBOF( ) const;
```

Return Value

Nonzero if the recordset contains no records or if you have scrolled backward before the first record; otherwise 0.

Remarks

Call this member function before you scroll from record to record to learn whether you have gone before the first record of the recordset. You can also use **IsBOF** along with **IsEOF** to determine whether the recordset contains any records or is empty. Immediately after you call **Open**, if the recordset contains no records, **IsBOF** returns nonzero.When you open a recordset that has at least one record, the first record is the current record and **IsBOF** returns 0.

If the first record is the current record and you call **MovePrev**, **IsBOF** will subsequently return nonzero. If **IsBOF** returns nonzero and you call **MovePrev**, an error occurs. If **IsBOF** returns nonzero, the current record is undefined, and any action that requires a current record will result in an error.

Example

This example uses **IsBOF** and **IsEOF** to detect the limits of a recordset as the code scrolls through the recordset in both directions.

```
// Open a recordset; first record is current
CCustSet rsCustSet( NULL );
rsCustSet.Open( );

if( rsCustSet.IsBOF( ) )
    return;
    // The recordset is empty

// Scroll to the end of the recordset, past
// the last record, so no record is current
while ( !rsCustSet.IsEOF( ) )
    rsCustSet.MoveNext( );

// Move to the last record
rsCustSet.MoveLast( );

// Scroll to beginning of the recordset, before
// the first record, so no record is current
while( !rsCustSet.IsBOF( ) )
    rsCustSet.MovePrev( );

// First record is current again
rsCustSet.MoveFirst( );
```

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::IsEOF
CRecordset::MoveFirst
CRecordset::MovePrev

Other Resources

CRecordset Members

CRecordset::IsDeleted

Determines whether the current record has been deleted.

```
BOOL IsDeleted( ) const;
```

Return Value

Nonzero if the recordset is positioned on a deleted record; otherwise 0.

Remarks

If you scroll to a record and **IsDeleted** returns **TRUE** (nonzero), then you must scroll to another record before you can perform any other recordset operations.

The result of **IsDeleted** depends on many factors, such as your recordset type, whether your recordset is updatable, whether you specified the **CRecordset::skipDeletedRecords** option when you opened the recordset, whether your driver packs deleted records, and whether there are multiple users.

For more information about **CRecordset::skipDeletedRecords** and driver packing, see the Open member function.

Note

If you have implemented bulk row fetching, you should not call **IsDeleted**. Instead, call the GetRowStatus member function. For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::Delete
CRecordset::IsBOF
CRecordset::IsEOF
CRecordset::Move

Other Resources

CRecordset Members

CRecordset::IsEOF

Returns nonzero if the recordset has been positioned after the last record. There is no current record.

```
BOOL IsEOF( ) const;
```

Return Value

Nonzero if the recordset contains no records or if you have scrolled beyond the last record; otherwise 0.

Remarks

Call this member function as you scroll from record to record to learn whether you have gone beyond the last record of the recordset. You can also use **IsEOF** to determine whether the recordset contains any records or is empty. Immediately after you call **Open**, if the recordset contains no records, **IsEOF** returns nonzero. When you open a recordset that has at least one record, the first record is the current record and **IsEOF** returns 0.

If the last record is the current record when you call **MoveNext**, **IsEOF** will subsequently return nonzero. If **IsEOF** returns nonzero and you call **MoveNext**, an error occurs. If **IsEOF** returns nonzero, the current record is undefined, and any action that requires a current record will result in an error.

Example

See the example for IsBOF.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::IsBOF
CRecordset::MoveLast
CRecordset::MoveNext

Other Resources

CRecordset Members

CRecordset::IsFieldDirty

Determines whether the specified field data member has been changed since Edit or AddNew was called.

```
BOOL IsFieldDirty(
   void * pv
);
```

Parameters

pv

A pointer to the field data member whose status you want to check, or **NULL** to determine if any of the fields are dirty.

Return Value

Nonzero if the specified field data member has changed since calling **AddNew** or **Edit**; otherwise 0.

Remarks

The data in all dirty field data members will be transferred to the record on the data source when the current record is updated by a call to the Update member function of **CRecordset** (following a call to **Edit** or **AddNew**).

Note
This member function is not applicable on recordsets that are using bulk row fetching. If you have implemented bulk row fetching, then **IsFieldDirty** will always return **FALSE** and will result in a failed assertion. For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

Calling **IsFieldDirty** will reset the effects of preceding calls to SetFieldDirty since the dirty status of the field is re-evaluated. In the **AddNew** case, if the current field value differs from the pseudo null value, the field status is set dirty. In the **Edit** case, if the field value differs from the cached value, then the field status is set dirty.

IsFieldDirty is implemented through DoFieldExchange.

For more information on the dirty flag, see the article Recordset: How Recordsets Select Records (ODBC).

Exceptions

This method can throw exceptions of type **CMemoryException***.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::SetFieldDirty
CRecordset::IsFieldNull

Other Resources

CRecordset Members

CRecordset::IsFieldNull

Returns nonzero if the specified field in the current record is Null (has no value).

```
BOOL IsFieldNull(
   void * pv
);
```

Parameters

pv

A pointer to the field data member whose status you want to check, or **NULL** to determine if any of the fields are Null.

Return Value

Nonzero if the specified field data member is flagged as Null; otherwise 0.

Remarks

Call this member function to determine whether the specified field data member of a recordset has been flagged as Null. (In database terminology, Null means "having no value" and is not the same as **NULL** in C++.) If a field data member is flagged as Null, it is interpreted as a column of the current record for which there is no value.

Note
This member function is not applicable on recordsets that are using bulk row fetching. If you have implemented bulk row fetching, then **IsFieldNull** will always return **FALSE** and will result in a failed assertion. For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

IsFieldNull is implemented through DoFieldExchange.

Exceptions

This method can throw exceptions of type **CMemoryException***.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::SetFieldNull
CRecordset::IsFieldDirty

Other Resources

CRecordset Members

CRecordset::IsFieldNullable

Returns nonzero if the specified field in the current record can be set to Null (having no value).

```
BOOL IsFieldNullable(
   void * pv
);
```

Parameters

pv

A pointer to the field data member whose status you want to check, or **NULL** to determine if any of the fields can be set to a Null value.

Remarks

Call this member function to determine whether the specified field data member is "nullable" (can be set to a Null value; C++ **NULL** is not the same as Null, which, in database terminology, means "having no value").

Note

If you have implemented bulk row fetching, you cannot call **IsFieldNullable**. Instead, call the GetODBCFieldInfo member function to determine whether a field can be set to a Null value. Note that you can always call **GetODBCFieldInfo**, regardless of whether you have implemented bulk row fetching. For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

A field that cannot be Null must have a value. If you attempt to set a such a field to Null when adding or updating a record, the data source rejects the addition or update, and Update will throw an exception. The exception occurs when you call **Update**, not when you call SetFieldNull.

Using **NULL** for the first argument of the function will apply the function only to **outputColumn** fields, not **param** fields. For instance, the call

```
SetFieldNull( NULL );
```

will set only **outputColumn** fields to **NULL**; **param** fields will be unaffected.

To work on **param** fields, you must supply the actual address of the individual **param** you want to work on, such as:

```
SetFieldNull( &m_strParam );
```

This means you cannot set all **param** fields to **NULL**, as you can with **outputColumn** fields.

IsFieldNullable is implemented through DoFieldExchange.

Exceptions

This method can throw exceptions of type **CDBException***.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::IsFieldNull
CRecordset::SetFieldNull

Other Resources

CRecordset Members

CRecordset::IsOpen

Determines if the recordset is already open.

```
BOOL IsOpen( ) const;
```

Return Value

Nonzero if the recordset object's Open or Requery member function has previously been called and the recordset has not been closed; otherwise 0.

See Also

Reference

CRecordset Class

Hierarchy Chart

Other Resources

CRecordset Members

CRecordset::Move

Moves the current record pointer within the recordset, either forward or backward.

```
virtual void Move(
   long nRows,
   WORD wFetchType = SQL_FETCH_RELATIVE
);
```

Parameters

nRows

The number of rows to move forward or backward. Positive values move forward, toward the end of the recordset. Negative values move backward, toward the beginning.

wFetchType

Determines the rowset that **Move** will fetch. For details, see Remarks.

Remarks

If you pass a value of 0 for *nRows*, **Move** refreshes the current record; **Move** will end any current **AddNew** or **Edit** mode, and will restore the current record's value before **AddNew** or **Edit** was called.

Note
When you move through a recordset, you cannot skip deleted records. See CRecordset::IsDeleted for more information. When you open a **CRecordset** with the **skipDeletedRecords** option set, **Move** asserts if the *nRows* parameter is 0. This behavior prevents the refresh of rows that are deleted by other client applications using the same data. See the *dwOption* parameter in Open for a description of **skipDeletedRecords**.

Move repositions the recordset by rowsets. Based on the values for *nRows* and *wFetchType*, **Move** fetches the appropriate rowset and then makes the first record in that rowset the current record. If you have not implemented bulk row fetching, then the rowset size is always 1. When fetching a rowset, **Move** directly calls the CheckRowsetError member function to handle any errors resulting from the fetch.

Depending on the values you pass, **Move** is equivalent to other **CRecordset** member functions. In particular, the value of *wFetchType* may indicate a member function that is more intuitive and often the preferred method for moving the current record.

The following table lists the possible values for *wFetchType*, the rowset that **Move** will fetch based on *wFetchType* and *nRows*, and any equivalent member function corresponding to *wFetchType*.

wFetchType	Fetched rowset	Equivalent member function
SQL_FETCH_RELATIVE (the default value)	The rowset starting *nRows* row(s) from the first row in the current rowset.	
SQL_FETCH_NEXT	The next rowset; *nRows* is ignored.	MoveNext
SQL_FETCH_PRIOR	The previous rowset; *nRows* is ignored.	MovePrev
SQL_FETCH_FIRST	The first rowset in the recordset; *nRows* is ignored.	MoveFirst

SQL_FETCH_LAST	The last complete rowset in the recordset; *nRows* is ignored.	MoveLast
SQL_FETCH_ABSOLUTE	If *nRows* > 0, the rowset starting *nRows* row(s) from the beginning of the recordset. If *nRows* < 0, the rowset starting *nRows* row(s) from the end of the recordset. If *nRows* = 0, then a beginning-of-file (BOF) condition is returned.	SetAbsolutePosition
SQL_FETCH_BOOKMARK	The rowset starting at the row whose bookmark value corresponds to *nRows*.	SetBookmark

Note

For foward-only recordsets, **Move** is only valid with a value of **SQL_FETCH_NEXT** for *wFetchType*.

Caution

Calling **Move** throws an exception if the recordset has no records. To determine whether the recordset has any records, call IsBOF and IsEOF.

Note

If you have scrolled past the beginning or end of the recordset (**IsBOF** or **IsEOF** returns nonzero), calling a **Move** function will possibly throw a **CDBException**. For example, if **IsEOF** returns nonzero and **IsBOF** does not, then **MoveNext** will throw an exception, but **MovePrev** will not.

Note

If you call **Move** while the current record is being updated or added, the updates are lost without warning.

For more information about recordset navigation, see the articles Recordset: Scrolling (ODBC) and Recordset: Bookmarks and Absolute Positions (ODBC). For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC). For related information, see the ODBC API function **SQLExtendedFetch** in the *Platform SDK*.

Exceptions

This method can throw exceptions of type **CDBException*** and **CMemoryException***.

Example

```
// rs is a CRecordset or a
// CRecordset-derived object

// Change the rowset size to 5
rs.SetRowsetSize( 5 );

// Move to the first record
// in the recordset
rs.MoveFirst( );

// Move to the sixth record
rs.Move( 5 );
// Other equivalent ways to
// move to the sixth record:
// rs.Move( 6, SQL_FETCH_ABSOLUTE );
// rs.SetAbsolutePosition( 6 );
// In this case, the sixth record is
// the first record in the next rowset,
// so the following are also equivalent:
// rs.Move( 1, SQL_FETCH_NEXT );
// rs.MoveNext( );
```

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::MoveNext
CRecordset::MovePrev
CRecordset::MoveFirst
CRecordset::MoveLast
CRecordset::SetAbsolutePosition
CRecordset::SetBookmark
CRecordset::IsBOF
CRecordset::IsEOF
CRecordset::CheckRowsetError

Other Resources

CRecordset Members

CRecordset::MoveFirst

Makes the first record in the first rowset the current record.

```
void MoveFirst( );
```

Remarks

Regardless of whether bulk row fetching has been implemented, this will always be the first record in the recordset.

You do not have to call **MoveFirst** immediately after you open the recordset. At that time, the first record (if any) is automatically the current record.

Note
This member function is not valid for forward-only recordsets.

Note
When you move through a recordset, you cannot skip deleted records. See the IsDeleted member function for details.

Caution
Calling any of the **Move** functions throws an exception if the recordset has no records. To determine whether the recordset has any records, call **IsBOF** and **IsEOF**.

Note
If you call any of the **Move** functions while the current record is being updated or added, the updates are lost without warning.

For more information about recordset navigation, see the articles Recordset: Scrolling (ODBC) and Recordset: Bookmarks and Absolute Positions (ODBC). For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

Exceptions

This method can throw exceptions of type **CDBException*** and **CMemoryException***.

Example

See the example for IsBOF.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::Move
CRecordset::MoveLast
CRecordset::MoveNext
CRecordset::MovePrev
CRecordset::IsBOF
CRecordset::IsEOF

Other Resources

CRecordset Members

CRecordset::MoveLast

Makes the first record in the last complete rowset the current record.

```
void MoveLast( );
```

Remarks

If you have not implemented bulk row fetching, your recordset has a rowset size of 1, so **MoveLast** simply moves to the last record in the recordset.

Note

This member function is not valid for forward-only recordsets.

Note

When you move through a recordset, you cannot skip deleted records. See the IsDeleted member function for details.

Caution

Calling any of the **Move** functions throws an exception if the recordset has no records. To determine whether the recordset has any records, call **IsBOF** and **IsEOF**.

Note

If you call any of the **Move** functions while the current record is being updated or added, the updates are lost without warning.

For more information about recordset navigation, see the articles Recordset: Scrolling (ODBC) and Recordset: Bookmarks and Absolute Positions (ODBC). For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

Exceptions

This method can throw exceptions of type **CDBException*** and **CMemoryException***.

Example

See the example for IsBOF.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::Move
CRecordset::MoveFirst
CRecordset::MoveNext
CRecordset::MovePrev
CRecordset::IsBOF
CRecordset::IsEOF

Other Resources

CRecordset Members

CRecordset::MoveNext

Makes the first record in the next rowset the current record.

```
void MoveNext( );
```

Remarks

If you have not implemented bulk row fetching, your recordset has a rowset size of 1, so **MoveNext** simply moves to the next record.

Note

When you move through a recordset, you cannot skip deleted records. See the IsDeleted member function for details.

Caution

Calling any of the **Move** functions throws an exception if the recordset has no records. To determine whether the recordset has any records, call **IsBOF** and **IsEOF**.

Note

It is also recommended that you call **IsEOF** before calling **MoveNext**. For example, if you have scrolled past the end of the recordset, **IsEOF** will return nonzero; a subsequent call to **MoveNext** would throw an exception.

Note

If you call any of the **Move** functions while the current record is being updated or added, the updates are lost without warning.

For more information about recordset navigation, see the articles Recordset: Scrolling (ODBC) and Recordset: Bookmarks and Absolute Positions (ODBC). For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

Exceptions

This method can throw exceptions of type **CDBException*** and **CMemoryException***.

Example

See the example for IsBOF.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::Move
CRecordset::MovePrev
CRecordset::MoveFirst
CRecordset::MoveLast
CRecordset::IsBOF
CRecordset::IsEOF

Other Resources

CRecordset Members

CRecordset::MovePrev

Makes the first record in the previous rowset the current record.

```
void MovePrev( );
```

Remarks

If you have not implemented bulk row fetching, your recordset has a rowset size of 1, so **MovePrev** simply moves to the previous record.

Note

This member function is not valid for forward-only recordsets.

Note

When you move through a recordset, you cannot skip deleted records. See the IsDeleted member function for details.

Caution

Calling any of the **Move** functions throws an exception if the recordset has no records. To determine whether the recordset has any records, call **IsBOF** and **IsEOF**.

Note

It is also recommended that you call **IsBOF** before calling **MovePrev**. For example, if you have scrolled ahead of the beginning of the recordset, **IsBOF** will return nonzero; a subsequent call to **MovePrev** would throw an exception.

Note

If you call any of the **Move** functions while the current record is being updated or added, the updates are lost without warning.

For more information about recordset navigation, see the articles Recordset: Scrolling (ODBC) and Recordset: Bookmarks and Absolute Positions (ODBC). For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

Exceptions

This method can throw exceptions of type **CDBException*** and **CMemoryException***.

Example

See the example for IsBOF.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::Move
CRecordset::MoveNext
CRecordset::MoveFirst
CRecordset::MoveLast
CRecordset::IsBOF
CRecordset::IsEOF

Other Resources

CRecordset Members

CRecordset::OnSetOptions

Called to set options (used on selection) for the specified ODBC statement.

```
virtual void OnSetOptions(
   HSTMT hstmt
);
```

Parameters

hstmt

The **HSTMT** of the ODBC statement whose options are to be set.

Remarks

Call **OnSetOptions** to set options (used on selection) for the specified ODBC statement. The framework calls this member function to set initial options for the recordset. **OnSetOptions** determines the data source's support for scrollable cursors and for cursor concurrency and sets the recordset's options accordingly. (Whereas **OnSetOptions** is used for selection operations, **OnSetUpdateOptions** is used for update operations.)

Override **OnSetOptions** to set options specific to the driver or the data source. For example, if your data source supports opening for exclusive access, you might override **OnSetOptions** to take advantage of that ability.

For more information about cursors, see the article ODBC.

See Also

Reference

CRecordset Class
Hierarchy Chart
CDatabase::OnSetOptions
CRecordset::OnSetUpdateOptions

Other Resources

CRecordset Members

CRecordset::OnSetUpdateOptions

Called to set options (used on update) for the specified ODBC statement.

```
virtual void OnSetUpdateOptions(
   HSTMT hstmt
);
```

Parameters

hstmt

The **HSTMT** of the ODBC statement whose options are to be set.

Remarks

Call **OnSetUpdateOptions** to set options (used on update) for the specified ODBC statement. The framework calls this member function after it creates an HSTMT to update records in a recordset. (Whereas **OnSetOptions** is used for selection operations, **OnSetUpdateOptions** is used for update operations.) **OnSetUpdateOptions** determines the data source's support for scrollable cursors and for cursor concurrency and sets the recordset's options accordingly.

Override **OnSetUpdateOptions** to set options of an ODBC statement before that statement is used to access a database.

For more information about cursors, see the article ODBC.

See Also

Reference

CRecordset Class
Hierarchy Chart
CDatabase::OnSetOptions
CRecordset::OnSetOptions

Other Resources

CRecordset Members

CRecordset::Open

Opens the recordset by retrieving the table or performing the query that the recordset represents.

```
virtual BOOL Open(
   UINT nOpenType = AFX_DB_USE_DEFAULT_TYPE,
   LPCTSTR lpszSQL = NULL,
   DWORD dwOptions = none
);
```

Parameters

nOpenType

Accept the default value, **AFX_DB_USE_DEFAULT_TYPE**, or use one of the following values from the **enum OpenType**:

- **CRecordset::dynaset** A recordset with bi-directional scrolling. The membership and ordering of the records are determined when the recordset is opened, but changes made by other users to the data values are visible following a fetch operation. Dynasets are also known as keyset-driven recordsets.
- **CRecordset::snapshot** A static recordset with bi-directional scrolling. The membership and ordering of the records are determined when the recordset is opened; the data values are determined when the records are fetched. Changes made by other users are not visible until the recordset is closed and then reopened.
- **CRecordset::dynamic** A recordset with bi-directional scrolling. Changes made by other users to the membership, ordering, and data values are visible following a fetch operation. Note that many ODBC drivers do not support this type of recordset.
- **CRecordset::forwardOnly** A read-only recordset with only forward scrolling.

  For **CRecordset**, the default value is **CRecordset::snapshot**. The default-value mechanism allows the Visual C++ wizards to interact with both ODBC **CRecordset** and DAO **CDaoRecordset**, which have different defaults.

For more information about these recordset types, see the article Recordset (ODBC). For related information, see the article "Using Block and Scrollable Cursors" in the *Platform SDK*.

Caution
If the requested type is not supported, the framework throws an exception.

lpszSQL

A string pointer containing one of the following:

- A **NULL** pointer.
- The name of a table.
- A SQL **SELECT** statement (optionally with a SQL **WHERE** or **ORDER BY** clause).
- A **CALL** statement specifying the name of a predefined query (stored procedure). Be careful that you do not insert whitespace between the curly brace and the **CALL** keyword.

For more information about this string, see the table and the discussion of ClassWizard's role under Remarks.

Note
The order of the columns in your result set must match the order of the RFX or Bulk RFX function calls in your DoFieldExchange or DoBulkFieldExchange function override.

dwOptions

A bitmask which can specify a combination of the values listed below. Some of these are mutually exclusive. The default value is **none**.

- **CRecordset::none** No options set. This parameter value is mutually exclusive with all other values. By default, the recordset can be updated with Edit or Delete and allows appending new records with AddNew. Updatability depends on the data source as well as on the *nOpenType* option you specify. Optimization for bulk additions is not available. Bulk row fetching will not be implemented. Deleted records will not be skipped during recordset navigation. Bookmarks are not available. Automatic dirty field checking is implemented.
- **CRecordset::appendOnly** Do not allow **Edit** or **Delete** on the recordset. Allow **AddNew** only. This option is mutually exclusive with **CRecordset::readOnly**.
- **CRecordset::readOnly** Open the recordset as read-only. This option is mutually exclusive with **CRecordset::appendOnly**.
- **CRecordset::optimizeBulkAdd** Use a prepared SQL statement to optimize adding many records at one time. Applies only if you are not using the ODBC API function **SQLSetPos** to update the recordset. The first update determines which fields are marked dirty. This option is mutually exclusive with **CRecordset::useMultiRowFetch**.
- **CRecordset::useMultiRowFetch** Implement bulk row fetching to allow multiple rows to be retrieved in a single fetch operation. This is an advanced feature designed to improve performance; however, bulk record field exchange is not supported by ClassWizard. This option is mutually exclusive with **CRecordset::optimizeBulkAdd**. Note that if you specify **CRecordset::useMultiRowFetch**, then the option **CRecordset::noDirtyFieldCheck** will be turned on automatically (double buffering will not be available); on forward-only recordsets, the option **CRecordset::useExtendedFetch** will be turned on automatically. For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).
- **CRecordset::skipDeletedRecords** Skip all deleted records when navigating through the recordset. This will slow performance in certain relative fetches. This option is not valid on forward-only recordsets. If you call Move with the *nRows* parameter set to 0, and the **CRecordset::skipDeletedRecords** option set, **Move** will assert. Note that **CRecordset::skipDeletedRecords** is similar to *driver packing*, which means that deleted rows are removed from the recordset. However, if your driver packs records, then it will skip only those records that you delete; it will not skip records deleted by other users while the recordset is open. **CRecordset::skipDeletedRecords** will skip rows deleted by other users.
- **CRecordset::useBookmarks** May use bookmarks on the recordset, if supported. Bookmarks slow data retrieval but improve performance for data navigation. Not valid on forward-only recordsets. For more information, see the article Recordset: Bookmarks and Absolute Positions (ODBC).
- **CRecordset::noDirtyFieldCheck** Turn off automatic dirty field checking (double buffering). This will improve performance; however, you must manually mark fields as dirty by calling the **SetFieldDirty** and **SetFieldNull** member functions.Note that double buffering in class **CRecordset** is similar to double buffering in class **CDaoRecordset**. However, in **CRecordset**, you cannot enable double buffering on individual fields; you either enable it for all fields or disable it for all fields. Note that if you specify the option **CRecordset::useMultiRowFetch**, then **CRecordset::noDirtyFieldCheck** will be turned on automatically; however, **SetFieldDirty** and **SetFieldNull** cannot be used on recordsets that implement bulk row fetching.
- **CRecordset::executeDirect** Do not use a prepared SQL statement. For improved performance, specify this option if the **Requery** member function will never be called.
- **CRecordset::useExtendedFetch** Implement **SQLExtendedFetch** instead of **SQLFetch**. This is designed for implementing bulk row fetching on forward-only recordsets. If you specify the option **CRecordset::useMultiRowFetch** on a forward-only recordset, then **CRecordset::useExtendedFetch** will be turned on automatically.
- **CRecordset::userAllocMultiRowBuffers** The user will allocate storage buffers for the data. Use this option in conjunction with **CRecordset::useMultiRowFetch** if you want to allocate your own storage; otherwise, the framework will automatically allocate the necessary storage. For more information, see the article Recordset: Fetching Records in Bulk (ODBC). Note that specifying **CRecordset::userAllocMultiRowBuffers** without specifying **CRecordset::useMultiRowFetch** will result in a failed assertion.

Return Value

Nonzero if the **CRecordset** object was successfully opened; otherwise 0 if CDatabase::Open (if called) returns 0.

Remarks

You must call this member function to run the query defined by the recordset. Before calling **Open**, you must construct the

recordset object.

This recordset's connection to the data source depends on how you construct the recordset before calling **Open**. If you pass a CDatabase object to the recordset constructor that has not been connected to the data source, this member function uses GetDefaultConnect to attempt to open the database object. If you pass **NULL** to the recordset constructor, the constructor constructs a **CDatabase** object for you, and **Open** attempts to connect the database object. For details on closing the recordset and the connection under these varying circumstances, see Close.

Note
Access to a data source through a **CRecordset** object is always shared. Unlike the **CDaoRecordset** class, you cannot use a **CRecordset** object to open a data source with exclusive access.

When you call **Open**, a query, usually a SQL **SELECT** statement, selects records based on criteria shown in the following table.

Value of the lpszSQL parameter	**Records selected are determined by**	**Example**
NULL	The string returned by **GetDefaultSQL**.	
SQL table name	All columns of the table-list in **DoFieldExchange** or **DoBulkFieldExchange**.	`"Customer"`
Predefined query (stored procedure) name	The columns the query is defined to return.	`"{call OverDueAccts}"`
SELECT column-list **FROM** table-list	The specified columns from the specified table(s).	`"SELECT CustId, CustName FROM Customer"`

Caution
Be careful that you do not insert extra whitespace in your SQL string. For example, if you insert whitespace between the curly brace and the **CALL** keyword, MFC will misinterpret the SQL string as a table name and incorporate it into a **SELECT** statement, which will result in an exception being thrown. Similarly, if your predefined query uses an output parameter, do not insert whitespace between the curly brace and the '?' symbol. Finally, you must not insert whitespace before the curly brace in a **CALL** statement or before the **SELECT** keyword in a **SELECT** statment.

The usual procedure is to pass **NULL** to **Open**; in this case, **Open** calls GetDefaultSQL. If you are using a derived **CRecordset** class, **GetDefualtSQL** gives the table name(s) you specified in ClassWizard. You can instead specify other information in the *lpszSQL* parameter.

Whatever you pass, **Open** constructs a final SQL string for the query (the string may have SQL **WHERE** and **ORDER BY** clauses appended to the *lpszSQL* string you passed) and then executes the query. You can examine the constructed string by calling GetSQL after calling **Open**. For additional details about how the recordset constructs a SQL statement and selects records, see the article Recordset: How Recordsets Select Records (ODBC).

The field data members of your recordset class are bound to the columns of the data selected. If any records are returned, the first record becomes the current record.

If you want to set options for the recordset, such as a filter or sort, specify these after you construct the recordset object but before you call **Open**. If you want to refresh the records in the recordset after the recordset is already open, call Requery.

For more information, including additional examples, see the articles Recordset (ODBC), Recordset: How Recordsets Select Records (ODBC), and Recordset: Creating and Closing Recordsets (ODBC).

Exceptions

This method can throw exceptions of type **CDBException*** and **CMemoryException***.

Example

The following code examples show different forms of the **Open** call.

```
// rs is a CRecordset or
// CRecordset-derived object

// Open rs using the default SQL statement,
// implement bookmarks, and turn off
// automatic dirty field checking
rs.Open( CRecordset::snapshot, NULL,
         CRecordset::useBookmarks |
         CRecordset::noDirtyFieldCheck );

// Pass a complete SELECT statement
// and open as a dynaset
rs.Open( CRecordset::dynaset,
         _T( "Select L_Name from Customer" ) );

// Accept all defaults
rs.Open( );
```

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::CRecordset
CRecordset::Close
CRecordset::GetDefaultSQL
CRecordset::GetSQL
CRecordset::m_strFilter
CRecordset::m_strSort
CRecordset::Requery

Other Resources

CRecordset Members

CRecordset::RefreshRowset

Updates the data and the status for a row in the current rowset.

```
void RefreshRowset(
   WORD wRow,
   WORD wLockType = SQL_LOCK_NO_CHANGE
);
```

Parameters

wRow

The one-based position of a row in the current rowset. This value can range from zero to the size of the rowset.

wLockType

A value indicating how to lock the row after it has been refreshed. For details, see Remarks.

Remarks

If you pass a value of zero for *wRow*, then every row in the rowset will be refreshed.

To use **RefreshRowset**, you must have implemented bulk row fetching by specifying the **CRecordset::useMulitRowFetch** option in the Open member function.

RefreshRowset calls the ODBC API function **SQLSetPos**. The *wLockType* parameter specifies the lock state of the row after **SQLSetPos** has executed. The following table describes the possible values for *wLockTyp*e.

wLockType	Description
SQL_LOCK_NO_CHANGE (the default value)	The driver or data source ensures that the row is in the same locked or unlocked state as it was before **RefreshRowset** was called.
SQL_LOCK_EXCLUSIVE	The driver or data source locks the row exclusively. Not all data sources support this type of lock.
SQL_LOCK_UNLOCK	The driver or data source unlocks the row. Not all data sources support this type of lock.

For more information about **SQLSetPos**, see the *Platform SDK*. For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::SetRowsetCursorPosition
CRecordset::SetRowsetSize

Other Resources

CRecordset Members

CRecordset::Requery

Rebuilds (refreshes) a recordset.

```
virtual BOOL Requery( );
```

Return Value

Nonzero if the recordset was successfully rebuilt; otherwise 0.

Remarks

If any records are returned, the first record becomes the current record.

In order for the recordset to reflect the additions and deletions that you or other users are making to the data source, you must rebuild the recordset by calling **Requery**. If the recordset is a dynaset, it automatically reflects updates that you or other users make to its existing records (but not additions). If the recordset is a snapshot, you must call **Requery** to reflect edits by other users as well as additions and deletions.

For either a dynaset or a snapshot, call **Requery** any time you want to rebuild the recordset using a new filter or sort, or new parameter values. Set the new filter or sort property by assigning new values to **m_strFilter** and **m_strSort** before calling **Requery**. Set new parameters by assigning new values to parameter data members before calling **Requery**. If the filter and sort strings are unchanged, you can reuse the query, which improves performance.

If the attempt to rebuild the recordset fails, the recordset is closed. Before you call **Requery**, you can determine whether the recordset can be requeried by calling the **CanRestart** member function. **CanRestart** does not guarantee that **Requery** will succeed.

Caution
Call **Requery** only after you have called Open.

Exceptions

This method can throw exceptions of type **CDBException*** and **CMemoryException***.

Example

This example rebuilds a recordset to apply a different sort order.

```
// Example for CRecordset::Requery

CCustSet rsCustSet( NULL );

// Open the recordset
rsCustSet.Open( );

// Use the recordset ...

// Set the sort order and Requery the recordset
rsCustSet.m_strSort = "District, Last_Name";
if( !rsCustSet.CanRestart( ) )
   return;    // Unable to requery

if( !rsCustSet.Requery( ) )
   // Requery failed, so take action
```

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::CanRestart
CRecordset::m_strFilter

CRecordset::m_strSort

Other Resources

CRecordset Members

CRecordset::SetAbsolutePosition

Positions the recordset on the record corresponding to the specified record number.

```
void SetAbsolutePosition(
   long nRows
);
```

Parameters

nRows

The one-based ordinal position for the current record in the recordset.

Remarks

SetAbsolutePosition moves the current record pointer based on this ordinal position.

Note
This member function is not valid on forward-only recordsets.

For ODBC recordsets, an absolute position setting of 1 refers to the first record in the recordset; a setting of 0 refers to the beginning-of-file (BOF) position.

You can also pass negative values to **SetAbsolutePosition**. In this case the recordset's position is evaluated from the end of the recordset. For example, `SetAbsolutePosition( -1 )` moves the current record pointer to the last record in the recordset.

Note
Absolute position is not intended to be used as a surrogate record number. Bookmarks are still the recommended way of ret aining and returning to a given position, since a record's position changes when preceding records are deleted. In addition, y ou cannot be assured that a given record will have the same absolute position if the recordset is re-created again because th e order of individual records within a recordset is not guaranteed unless it is created with a SQL statement using an **ORDER BY** clause.

For more information about recordset navigation and bookmarks, see the articles Recordset: Scrolling (ODBC) and Recordset: Bookmarks and Absolute Positions (ODBC).

Exceptions

This method can throw exceptions of type **CDBException*** and **CMemoryException***.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::SetBookmark

Other Resources

CRecordset Members

CRecordset::SetBookmark

Positions the recordset on the record containing the specified bookmark.

```
void SetBookmark(
   const CDBVariant& varBookmark
);
```

Parameters

varBookmark

A reference to a CDBVariant object containing the bookmark value for a specific record.

Remarks

To determine if bookmarks are supported on the recordset, call CanBookmark. To make bookmarks available if they are supported, you must set the **CRecordset::useBookmarks** option in the *dwOptions* parameter of the Open member function.

Note
If bookmarks are unsupported or unavailable, calling **SetBookmark** will result in an exception being thrown. Bookmarks are not supported on forward-only recordsets.

To first retrieve the bookmark for the current record, call GetBookmark, which saves the bookmark value to a **CDBVariant** object. Later, you can return to that record by calling **SetBookmark** using the saved bookmark value.

Note
After certain recordset operations, you should check the bookmark persistence before calling **SetBookmark**. For example, if you retrieve a bookmark with **GetBookmark** and then call **Requery**, the bookmark may no longer be valid. Call CDatabase::GetBookmarkPersistence to check whether you can safely call **SetBookmark**.

For more information about bookmarks and recordset navigation, see the articles Recordset: Bookmarks and Absolute Positions (ODBC) and Recordset: Scrolling (ODBC).

Exceptions

This method can throw exceptions of type **CDBException*** and **CMemoryException***.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::CanBookmark
CRecordset::GetBookmark
CRecordset::SetAbsolutePosition
CDatabase::GetBookmarkPersistence

Other Resources

CRecordset Members

CRecordset::SetFieldDirty

Flags a field data member of the recordset as changed or as unchanged.

```
void SetFieldDirty(
   void* pv,
      BOOL bDirty = TRUE
);
```

Parameters

pv

Contains the address of a field data member in the recordset or **NULL**. If **NULL**, all field data members in the recordset are flagged. (C++ **NULL** is not the same as Null in database terminology, which means "having no value.")

bDirty

TRUE if the field data member is to be flagged as "dirty" (changed). Otherwise **FALSE** if the field data member is to be flagged as "clean" (unchanged).

Remarks

Marking fields as unchanged ensures the field is not updated and results in less SQL traffic.

Note
This member function is not applicable on recordsets that are using bulk row fetching. If you have implemented bulk row fetching, then **SetFieldDirty** will result in a failed assertion. For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

The framework marks changed field data members to ensure they will be written to the record on the data source by the record field exchange (RFX) mechanism. Changing the value of a field generally sets the field dirty automatically, so you will seldom need to call **SetFieldDirty** yourself, but you might sometimes want to ensure that columns will be explicitly updated or inserted regardless of what value is in the field data member.

Caution
Call this member function only after you have called Edit or AddNew.

Using **NULL** for the first argument of the function will apply the function only to **outputColumn** fields, not **param** fields. For instance, the call

```
SetFieldNull( NULL );
```

will set only **outputColumn** fields to **NULL**; **param** fields will be unaffected.

To work on **param** fields, you must supply the actual address of the individual **param** you want to work on, such as:

```
SetFieldNull( &m_strParam );
```

This means you cannot set all **param** fields to **NULL**, as you can with **outputColumn** fields.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::IsFieldDirty
CRecordset::SetFieldNull
CRecordset::Edit
CRecordset::Update

Other Resources

CRecordset Members

CRecordset::SetFieldNull

Flags a field data member of the recordset as Null (specifically having no value) or as non-Null.

```
void SetFieldNull(
   void* pv,
      BOOL bNull = TRUE
);
```

Parameters

pv

Contains the address of a field data member in the recordset or **NULL**. If **NULL**, all field data members in the recordset are flagged. (C++ **NULL** is not the same as Null in database terminology, which means "having no value.")

bNull

Nonzero if the field data member is to be flagged as having no value (Null). Otherwise 0 if the field data member is to be flagged as non-Null.

Remarks

When you add a new record to a recordset, all field data members are initially set to a Null value and flagged as "dirty" (changed). When you retrieve a record from a data source, its columns either already have values or are Null.

Note

Do not call this member function on recordsets that are using bulk row fetching. If you have implemented bulk row fetching, calling **SetFieldNull** results in a failed assertion. For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

If you specifically wish to designate a field of the current record as not having a value, call **SetFieldNull** with *bNull* set to **TRUE** to flag it as Null. If a field was previously marked Null and you now want to give it a value, simply set its new value. You do not have to remove the Null flag with **SetFieldNull**. To determine whether the field is allowed to be Null, call **IsFieldNullable**.

Caution

Call this member function only after you have called Edit or AddNew.

Using **NULL** for the first argument of the function will apply the function only to **outputColumn** fields, not **param** fields. For instance, the call

```
SetFieldNull( NULL );
```

will set only **outputColumn** fields to **NULL**; **param** fields will be unaffected.

To work on **param** fields, you must supply the actual address of the individual **param** you want to work on, such as:

```
SetFieldNull( &m_strParam );
```

This means you cannot set all **param** fields to **NULL**, as you can with **outputColumn** fields.

Note

When setting parameters to Null, a call to **SetFieldNull** before the recordset is opened results in an assertion. In this case, ca ll SetParamNull.

SetFieldNull is implemented through DoFieldExchange.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::IsFieldNull
CRecordset::SetFieldDirty
CRecordset::Edit
CRecordset::Update
CRecordset::IsFieldNullable

Other Resources

CRecordset Members

CRecordset::SetLockingMode

Sets the locking mode to "optimistic" locking (the default) or "pessimistic" locking. Determines how records are locked for updates.

```
void SetLockingMode(
   UINT nMode
);
```

Parameters

nMode

Contains one of the following values from the **enum LockMode**:

- **optimistic** Optimistic locking locks the record being updated only during the call to **Update**.
- **pessimistic** Pessimistic locking locks the record as soon as **Edit** is called and keeps it locked until the **Update** call completes or you move to a new record.

Remarks

Call this member function if you need to specify which of two record-locking strategies the recordset is using for updates. By default, the locking mode of a recordset is **optimistic**. You can change that to a more cautious **pessimistic** locking strategy. Call **SetLockingMode** after you construct and open the recordset object but before you call **Edit**.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::Edit
CRecordset::Update

Other Resources

CRecordset Members

CRecordset::SetParamNull

Flags a parameter as Null (specifically having no value) or as non-Null.

```
void SetParamNull(
   int nIndex,
   BOOL bNull = TRUE
);
```

Parameters

nIndex

The zero-based index of the parameter.

bNull

If **TRUE** (the default value), the parameter is flagged as Null. Otherwise, the parameter is flagged as non-Null.

Remarks

Unlike SetFieldNull, you can call **SetParamNull** before you have opened the recordset.

SetParamNull is typically used with predefined queries (stored procedures).

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::FlushResultSet

Other Resources

CRecordset Members

CRecordset::SetRowsetCursorPosition

Moves the cursor to a row within the current rowset.

```
void SetRowsetCursorPosition(
   WORD wRow,
      WORD wLockType = SQL_LOCK_NO_CHANGE
);
```

Parameters

wRow

The one-based position of a row in the current rowset. This value can range from 1 to the size of the rowset.

wLockType

Value indicating how to lock the row after it has been refreshed. For details, see Remarks.

Remarks

When implementing bulk row fetching, records are retrieved by rowsets, where the first record in the fetched rowset is the current record. In order to make another record within the rowset the current record, call **SetRowsetCursorPosition**. For example, you can combine **SetRowsetCursorPosition** with the GetFieldValue member function to dynamically retrieve the data from any record of your recordset.

To use **SetRowsetCursorPosition**, you must have implemented bulk row fetching by specifying the **CRecordset::useMultiRowFetch** option of the *dwOptions* parameter in the Open member function.

SetRowsetCursorPosition calls the ODBC API function **SQLSetPos**. The *wLockType* parameter specifies the lock state of the row after **SQLSetPos** has executed. The following table describes the possible values for *wLockType*.

wLockType	Description
SQL_LOCK_NO_CHANGE (the default value)	The driver or data source ensures that the row is in the same locked or unlocked state as it was before **SetRowsetCursorPosition** was called.
SQL_LOCK_EXCLUSIVE	The driver or data source locks the row exclusively. Not all data sources support this type of lock.
SQL_LOCK_UNLOCK	The driver or data source unlocks the row. Not all data sources support this type of lock.

For more information about **SQLSetPos**, see the *Platform SDK*. For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::RefreshRowset
CRecordset::SetRowsetSize

Other Resources

CRecordset Members

CRecordset::SetRowsetSize

Specifies the number of records you wish to retrieve during a fetch.

```
virtual void SetRowsetSize(
   DWORD dwNewRowsetSize
);
```

Parameters

dwNewRowsetSize

The number of rows to retrieve during a given fetch.

Remarks

This virtual member function specifies how many rows you wish to retrieve during a single fetch when using bulk row fetching. To implement bulk row fetching, you must set the **CRecordset::useMultiRowFetch** option in the *dwOptions* parameter of the Open member function.

Note
Calling **SetRowsetSize** without implementing bulk row fetching will result in a failed assertion.

Call **SetRowsetSize** before calling **Open** to initially set the rowset size for the recordset. The default rowset size when implementing bulk row fetching is 25.

Note
Use caution when calling **SetRowsetSize**. If you are manually allocating storage for the data (as specified by the **CRecordset::userAllocMultiRowBuffers** option of the dwOptions parameter in **Open**), you should check whether you need to reallocate these storage buffers after you call **SetRowsetSize**, but before you perform any cursor navigation operation.

To obtain the current setting for the rowset size, call GetRowsetSize.

For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::Open
CRecordset::GetRowsetSize
CRecordset::CheckRowsetError
CRecordset::DoBulkFieldExchange

Other Resources

CRecordset Members

CRecordset::Update

Completes an **AddNew** or **Edit** operation by saving the new or edited data on the data source.

```
virtual BOOL Update( );
```

Return Value

Nonzero if one record was successfully updated; otherwise 0 if no columns have changed. If no records were updated, or if more than one record was updated, an exception is thrown. An exception is also thrown for any other failure on the data source.

Remarks

Call this member function after a call to the AddNew or Edit member function. This call is required to complete the **AddNew** or **Edit** operation.

Note

If you have implemented bulk row fetching, you cannot call **Update**. This will result in a failed assertion. Although class **CRecordset** does not provide a mechanism for updating bulk rows of data, you can write your own functions by using the ODBC API function **SQLSetPos**. For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

Both **AddNew** and **Edit** prepare an edit buffer in which the added or edited data is placed for saving to the data source. **Update** saves the data. Only those fields marked or detected as changed are updated.

If the data source supports transactions, you can make the **Update** call (and its corresponding **AddNew** or **Edit** call) part of a transaction. For more information about transactions, see the article Transaction (ODBC).

Caution

If you call **Update** without first calling either **AddNew** or **Edit**, **Update** throws a **CDBException**. If you call **AddNew** or **Edit**, you must call **Update** before you call a **Move** operation or before you close either the recordset or the data source connection. Otherwise, your changes are lost without notification.

For details on handling **Update** failures, see the article Recordset: How Recordsets Update Records (ODBC).

Exceptions

This method can throw exceptions of type **CDBException***.

Example

See the article Transaction: Performing a Transaction in a Recordset (ODBC).

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::Edit
CRecordset::AddNew
CRecordset::SetFieldDirty
CDBException Class

Other Resources

CRecordset Members

CRecordset Data Members

For information about the data members in **CRecordset**, see CRecordset Members.

CRecordset::m_hstmt

Contains a handle to the ODBC statement data structure, of type **HSTMT**, associated with the recordset.

Remarks

Each query to an ODBC data source is associated with an **HSTMT**.

Caution
Do not use **m_hstmt** before Open has been called.

Normally you do not need to access the **HSTMT** directly, but you might need it for direct execution of SQL statements. The **ExecuteSQL** member function of class **CDatabase** provides an example of using **m_hstmt**.

See Also

Reference

CRecordset Class

Hierarchy Chart

CDatabase::ExecuteSQL

Other Resources

CRecordset Members

CRecordset::m_nFields

Contains the number of field data members in the recordset class; that is, the number of columns selected by the recordset from the data source.

Remarks

The constructor for the recordset class must initialize **m_nFields** with the correct number. If you have not implemented bulk row fetching, ClassWizard writes this initialization for you when you use it to declare your recordset class. You can also write it manually.

The framework uses this number to manage interaction between the field data members and the corresponding columns of the current record on the data source.

Caution
This number must correspond to the number of "output columns" registered in **DoFieldExchange** or **DoBulkFieldExchange** after a call to SetFieldType with the parameter **CFieldExchange::outputColumn**.

You can bind columns dynamically, as explained in the article "Recordset: Dynamically Binding Data Columns." If you do so, you must increase the count in **m_nFields** to reflect the number of RFX or Bulk RFX function calls in your **DoFieldExchange** or **DoBulkFieldExchange** member function for the dynamically bound columns.

For more information, see the articles Recordset: Dynamically Binding Data Columns (ODBC) and Recordset: Fetching Records in Bulk (ODBC).

Example

See the article Record Field Exchange: Using RFX.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::DoFieldExchange
CRecordset::DoBulkFieldExchange
CRecordset::m_nParams
CFieldExchange::SetFieldType

Other Resources

CRecordset Members

CRecordset::m_nParams

Contains the number of parameter data members in the recordset class; that is, the number of parameters passed with the recordset's query.

Remarks

If your recordset class has any parameter data members, the constructor for the class must initialize **m_nParams** with the correct number. The value of **m_nParams** defaults to 0. If you add parameter data members (which you must do manually) you must also manually add an initialization in the class constructor to reflect the number of parameters (which must be at least as large as the number of '?' placeholders in your **m_strFilter** or **m_strSort** string).

The framework uses this number when it parameterizes the recordset's query.

Caution
This number must correspond to the number of "params" registered in **DoFieldExchange** or **DoBulkFieldExchange** after a call to SetFieldType with a parameter value of **CFieldExchange::inputParam**, **CFieldExchange::param**, **CFieldExchange::outputParam**, or **CFieldExchange::inoutParam**.

Example

See the articles Recordset: Parameterizing a Recordset (ODBC) and Record Field Exchange: Using RFX.

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::DoFieldExchange
CRecordset::DoBulkFieldExchange
CRecordset::m_nFields
CFieldExchange::SetFieldType

Other Resources

CRecordset Members

CRecordset::m_pDatabase

Contains a pointer to the **CDatabase** object through which the recordset is connected to a data source.

Remarks

This variable is set in two ways. Typically, you pass a pointer to an already connected **CDatabase** object when you construct the recordset object. If you pass **NULL** instead, **CRecordset** creates a **CDatabase** object for you and connects it. In either case, **CRecordset** stores the pointer in this variable.

Normally you will not directly need to use the pointer stored in **m_pDatabase**. If you write your own extensions to **CRecordset**, however, you might need to use the pointer. For example, you might need the pointer if you throw your own **CDBException**s. Or you might need it if you need to do something using the same **CDatabase** object, such as running transactions, setting timeouts, or calling the **ExecuteSQL** member function of class **CDatabase** to execute SQL statements directly.

See Also

Reference

CRecordset Class

Hierarchy Chart

Other Resources

CRecordset Members

CRecordset::m_strFilter

After you construct the recordset object, but before you call its **Open** member function, use this data member to store a **CString** containing a SQL **WHERE** clause.

Remarks

The recordset uses this string to constrain (or filter) the records it selects during the **Open** or **Requery** call. This is useful for selecting a subset of records, such as "all salespersons based in California" ("state = CA"). The ODBC SQL syntax for a **WHERE** clause is

```
WHERE search-condition
```

Note that you do not include the **WHERE** keyword in your string. The framework supplies it.

You can also parameterize your filter string by placing '?' placeholders in it, declaring a parameter data member in your class for each placeholder, and passing parameters to the recordset at run time. This lets you construct the filter at run time. For more information, see the article Recordset: Parameterizing a Recordset (ODBC).

For more information about SQL **WHERE** clauses, see the article SQL. For more information about selecting and filtering records, see the article Recordset: Filtering Records (ODBC).

Example

```
// Example for CRecordset::m_strFilter

CCustSet rsCustSet( NULL );

// Set the filter
rsCustSet.m_strFilter = "state = 'CA'";

// Run the filtered query
rsCustSet.Open( CRecordset::snapshot, "Customers" );
```

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::m_strSort
CRecordset::Requery

Other Resources

CRecordset Members

CRecordset::m_strSort

After you construct the recordset object, but before you call its **Open** member function, use this data member to store a **CString** containing a SQL **ORDER BY** clause.

Remarks

The recordset uses this string to sort the records it selects during the **Open** or **Requery** call. You can use this feature to sort a recordset on one or more columns. The ODBC SQL syntax for an **ORDER BY** clause is

```
ORDER BY sort-specification [, sort-specification]...
```

where a sort-specification is an integer or a column name. You can also specify ascending or descending order (the order is ascending by default) by appending "ASC" or "DESC" to the column list in the sort string. The selected records are sorted first by the first column listed, then by the second, and so on. For example, you might order a "Customers" recordset by last name, then first name. The number of columns you can list depends on the data source. For more information, see the *Platform SDK*.

Note that you do not include the **ORDER BY** keyword in your string. The framework supplies it.

For more information about SQL clauses, see the article SQL. For more information about sorting records, see the article Recordset: Sorting Records (ODBC).

Example

```
// Example for CRecordset::m_strSort

CCustSet rsCustSet( NULL );

// Set the sort string
rsCustSet.m_strSort = "District, Last_Name";

// Run the sorted query
rsCustSet.Open( CRecordset::snapshot, "Customers" );
```

See Also

Reference

CRecordset Class
Hierarchy Chart
CRecordset::m_strFilter
CRecordset::Requery

Other Resources

CRecordset Members

CRecordView Class

A view that displays database records in controls.

```
class AFX_NOVTABLE CRecordView : public CFormView
```

Remarks

The view is a form view directly connected to a **CRecordset** object. The view is created from a dialog template resource and displays the fields of the **CRecordset** object in the dialog template's controls. The **CRecordView** object uses dialog data exchange (DDX) and record field exchange (RFX) to automate the movement of data between the controls on the form and the fields of the recordset. **CRecordView** also supplies a default implementation for moving to the first, next, previous, or last record and an interface for updating the record currently on view.

Note

If you are working with the Data Access Objects (DAO) classes rather than the Open Database Connectivity (ODBC) classes, u se class CDaoRecordView instead. For more information, see the article Overview: Database Programming.

The most common way to create your record view is with the Application Wizard. Tge Application Wizard creates both the record view class and its associated recordset class as part of your skeleton starter application. If you don't create the record view class with the Application Wizard, you can create it later with ClassWizard. If you simply need a single form, the Application Wizard approach is easier. ClassWizard lets you decide to use a record view later in the development process. Using ClassWizard to create a record view and a recordset separately and then connect them is the most flexible approach because it gives you more control in naming the recordset class and its .H/.CPP files. This approach also lets you have multiple record views on the same recordset class.

To make it easy for end-users to move from record to record in the record view, the Application Wizard creates menu (and optionally toolbar) resources for moving to the first, next, previous, or last record. If you create a record view class with ClassWizard, you need to create these resources yourself with the menu and bitmap editors.

For information about the default implementation for moving from record to record, see **IsOnFirstRecord** and **IsOnLastRecord** and the article Using a Record View.

CRecordView keeps track of the user's position in the recordset so that the record view can update the user interface. When the user moves to either end of the recordset, the record view disables user interface objects — such as menu items or toolbar buttons — for moving further in the same direction.

For more information about declaring and using your record view and recordset classes, see "Designing and Creating a Record View" in the article Record Views. For more information about how record views work and how to use them, see the article Using a Record View.

Requirements

Header: afxdb.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CFormView Class
Hierarchy Chart
CRecordset Class
CFormView Class

Other Resources

CRecordView Members

CRecordView Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CView Members

CScrollView Members

CFormView Members

Construction

CRecordView	Constructs a **CRecordView** object.

Attributes

IsOnFirstRecord	Returns nonzero if the current record is the first record in the associated recordset.
IsOnLastRecord	Returns nonzero if the current record is the last record in the associated recordset.
OnGetRecordset	Returns a pointer to an object of a class derived from **CRecordset**. ClassWizard overrides this function for you and creates the recordset if necessary.

Operations

OnMove	If the current record has changed, updates it on the data source, then moves to the specified record (next, previous, first, or last).

See Also

Reference

CRecordView Class

Hierarchy Chart

CRecordView Member Functions

For information about the member functions in **CRecordView**, see CRecordView Members.

CRecordView::CRecordView

When you create an object of a type derived from **CRecordView**, call either form of the constructor to initialize the view object and identify the dialog resource on which the view is based.

```
explicit CRecordView(
   LPCTSTR lpszTemplateName
);
explicit CRecordView(
   UINT nIDTemplate
);
```

Parameters

lpszTemplateName

Contains a null-terminated string that is the name of a dialog template resource.

nIDTemplate

Contains the ID number of a dialog template resource.

Remarks

You can either identify the resource by name (pass a string as the argument to the constructor) or by its ID (pass an unsigned integer as the argument). Using a resource ID is recommended.

Note

Your derived class *must* supply its own constructor. In the constructor of your derived class, call the constructor **CRecordView::CRecordView** with the resource name or ID as an argument, as shown in the example below.

CRecordView::OnInitialUpdate calls **UpdateData**, which calls **DoDataExchange**. This initial call to **DoDataExchange** connects **CRecordView** controls (indirectly) to **CRecordset** field data members created by ClassWizard. These data members cannot be used until after you call the base class **CFormView::OnInitialUpdate** member function.

Note

If you use ClassWizard, the wizard defines an **enum** value `CRecordView::IDD` and specifies it in the member initialization list for the constructor where you see `IDD_MYFORM` in the example. The example shows how you can specify the dialog template resource ID if you write the code yourself without the wizard.

Example

```
CMyRecordView::CMyRecordView()
   : CRecordView( IDD_MYFORM )
{
   //{{AFX_DATA_INIT( CMyRecordView )
      // NOTE: the ClassWizard will add member initialization here
   //}}AFX_DATA_INIT
   // Other construction code, such as data initialization
}
```

See Also

Reference

CRecordView Class
Hierarchy Chart
CRecordset::DoFieldExchange
CView::OnInitialUpdate
CWnd::UpdateData

Other Resources

CRecordView Members

CRecordView::IsOnFirstRecord

Call this member function to determine whether the current record is the first record in the recordset object associated with this record view.

```
BOOL IsOnFirstRecord( );
```

Return Value

Nonzero if the current record is the first record in the recordset; otherwise 0.

Remarks

This function is useful for writing your own implementations of default command update handlers written by ClassWizard.

If the user moves to the first record, the framework disables any user interface objects you have for moving to the first or the previous record.

See Also

Reference

CRecordView Class
Hierarchy Chart
CRecordView::OnMove
CRecordView::IsOnLastRecord
CRecordset::IsBOF
CRecordset::GetRecordCount

Other Resources

CRecordView Members

CRecordView::IsOnLastRecord

Call this member function to determine whether the current record is the last record in the recordset object associated with this record view.

```
BOOL IsOnLastRecord( );
```

Return Value

Nonzero if the current record is the last record in the recordset; otherwise 0.

Remarks

This function is useful for writing your own implementations of the default command update handlers that ClassWizard writes to support a user interface for moving from record to record.

Caution

The result of this function is reliable except that the view cannot detect the end of the recordset until the user has moved past it. The user must move beyond the last record before the record view can tell that it must disable any user interface objects for moving to the next or last record. If the user moves past the last record and then moves back to the last record (or before it), the record view can track the user's position in the recordset and disable user interface objects correctly. **IsOnLastRecord** is also unreliable after a call to the implementation function **OnRecordLast**, which handles the **ID_RECORD_LAST** command, or **CRecordset::MoveLast**.

See Also

Reference

CRecordView Class
Hierarchy Chart
CRecordView::OnMove
CRecordView::IsOnFirstRecord
CRecordset::IsEOF
CRecordset::GetRecordCount

Other Resources

CRecordView Members

CRecordView::OnGetRecordset

Returns a pointer to the **CRecordset**-derived object associated with the record view.

```
virtual CRecordset* OnGetRecordset( ) = 0;
```

Return Value

A pointer to a **CRecordset**-derived object if the object was successfully created; otherwise a **NULL** pointer.

Remarks

You must override this member function to construct or obtain a recordset object and return a pointer to it. If you declare your record view class with ClassWizard, the wizard writes a default override for you. ClassWizard's default implementation returns the recordset pointer stored in the record view if one exists. If not, it constructs a recordset object of the type you specified with ClassWizard and calls its **Open** member function to open the table or run the query, and then returns a pointer to the object.

For more information and examples, see the article Record Views: Using a Record View.

See Also

Reference

CRecordView Class
Hierarchy Chart
CRecordset Class
CRecordset::Open

Other Resources

CRecordView Members

CRecordView::OnMove

Call this member function to move to a different record in the recordset and display its fields in the controls of the record view.

```
virtual BOOL OnMove(
   UINT nIDMoveCommand
);
```

Parameters

nIDMoveCommand

One of the following standard command ID values:

- **ID_RECORD_FIRST** Move to the first record in the recordset.
- **ID_RECORD_LAST** Move to the last record in the recordset.
- **ID_RECORD_NEXT** Move to the next record in the recordset.
- **ID_RECORD_PREV** Move to the previous record in the recordset.

Return Value

Nonzero if the move was successful; otherwise 0 if the move request was denied.

Remarks

The default implementation calls the appropriate **Move** member function of the **CRecordset** object associated with the record view.

By default, **OnMove** updates the current record on the data source if the user has changed it in the record view.

The Application Wizard creates a menu resource with First Record, Last Record, Next Record, and Previous Record menu items. If you select the Dockable Toolbar option, the Application Wizard also creates a toolbar with buttons corresponding to these commands.

If you move past the last record in the recordset, the record view continues to display the last record. If you move backward past the first record, the record view continues to display the first record.

Caution
Calling **OnMove** throws an exception if the recordset has no records. Call the appropriate user interface update handler function — **OnUpdateRecordFirst**, **OnUpdateRecordLast**, **OnUpdateRecordNext**, or **OnUpdateRecordPrev** — before the corresponding move operation to determine whether the recordset has any records.

Exceptions

This method can throw exceptions of type **CDBException***.

See Also

Reference

CRecordView Class
Hierarchy Chart
CRecordset::Move

Other Resources

CRecordView Members

CRectTracker Class

Allows an item to be displayed, moved, and resized in different fashions.

```
class CRectTracker
```

Remarks

CRectTracker does not have a base class.

Although the **CRectTracker** class is designed to allow the user to interact with OLE items by using a graphical interface, its use is not restricted to OLE-enabled applications. It can be used anywhere such a user interface is required.

CRectTracker borders can be solid or dotted lines. The item can be given a hatched border or overlaid with a hatched pattern to indicate different states of the item. You can place eight resize handles on either the outside or the inside border of the item. (For an explanation of the resize handles, see GetHandleMask.) Finally, a **CRectTracker** allows you to change the orientation of an item during resizing.

To use **CRectTracker**, construct a **CRectTracker** object and specify which display states are initialized. You can then use this interface to give the user visual feedback on the current status of the OLE item associated with the **CRectTracker** object.

For more information on using **CRectTracker**, see the article Trackers.

Requirements

Header: afxext.h

See Also

Tasks

TRACKER Sample: Illustrates Various CRectTracker Styles and Options

DRAWCLI Sample: Illustrates Integrating Active Container Support with Application-Specific Features

Reference

Hierarchy Chart

COleResizeBar Class

CRect Class

CRectTracker::GetHandleMask

Other Resources

CRectTracker Members

CRectTracker Members

Data Members

m_nHandleSize	Determines size of resize handles.
m_nStyle	Current style(s) of the tracker.
m_rect	Current position (in pixels) of the rectangle.
m_sizeMin	Determines minimum rectangle width and height.

Construction

CRectTracker	Constructs a **CRectTracker** object.

Operations

Draw	Renders the rectangle.
GetTrueRect	Returns width and height of rectangle, including resize handles.
HitTest	Returns the current position of the cursor related to the **CRectTracker** object.
NormalizeHit	Normalizes a hit-test code.
SetCursor	Sets the cursor, depending on its position over the rectangle.
Track	Allows the user to manipulate the rectangle.
TrackRubberBand	Allows the user to "rubber-band" the selection.

Overridables

AdjustRect	Called when the rectangle is resized.
DrawTrackerRect	Called when drawing the border of a **CRectTracker** object.
GetHandleMask	Called to get the mask of a **CRectTracker** item's resize handles.
OnChangedRect	Called when the rectangle has been resized or moved.

See Also

Reference

CRectTracker Class

Hierarchy Chart

CRectTracker Member Functions

For information about the member functions in **CRectTracker**, see CRectTracker Members.

CRectTracker::AdjustRect

Called by the framework when the tracking rectangle is resized by using a resize handle.

```
virtual void AdjustRect(
   int nHandle,
   LPRECT lpRect
);
```

Parameters

nHandle

Index of handle used.

lpRect

Pointer to the current size of the rectangle. (The size of a rectangle is given by its height and width.)

Remarks

The default behavior of this function allows the rectangle's orientation to change only when **Track** and **TrackRubberBand** are called with inverting allowed.

Override this function to control the adjustment of the tracking rectangle during a dragging operation. One method is to adjust the coordinates specified by *lpRect* before returning.

Special features that are not directly supported by **CRectTracker**, such as snap-to-grid or keep-aspect-ratio, can be implemented by overriding this function.

See Also

Reference

CRectTracker Class
Hierarchy Chart
CRectTracker::Track
CRectTracker::TrackRubberBand
CRectTracker::OnChangedRect

Other Resources

CRectTracker Members

CRectTracker::CRectTracker

Creates and initializes a **CRectTracker** object.

```
CRectTracker( );
CRectTracker(
   LPCRECT lpSrcRect,
   UINT nStyle
);
```

Parameters

lpSrcRect

The coordinates of the rectangle object.

nStyle

Specifies the style of the **CRectTracker** object. The following styles are supported:

- **CRectTracker::solidLine** Use a solid line for the rectangle border.
- **CRectTracker::dottedLine** Use a dotted line for the rectangle border.
- **CRectTracker::hatchedBorder** Use a hatched pattern for the rectangle border.
- **CRectTracker::resizeInside** Resize handles located inside the rectangle.
- **CRectTracker::resizeOutside** Resize handles located outside the rectangle.
- **CRectTracker::hatchInside** Hatched pattern covers the entire rectangle.

Remarks

The default constructor initializes the **CRectTracker** object with the values from *lpSrcRect* and initializes other sizes to system defaults. If the object is created with no parameters, the **m_rect** and **m_nStyle** data members are uninitialized.

See Also

Reference

CRectTracker Class
Hierarchy Chart
CRect::CRect

Other Resources

CRectTracker Members

CRectTracker::Draw

Call this function to draw the rectangle's outer lines and inner region.

```
void Draw(
   CDC* pDC
) const;
```

Parameters

pDC

Pointer to the device context on which to draw.

Remarks

The style of the tracker determines how the drawing is done. See the constructor for **CRectTracker** for more information on the styles available.

See Also

Reference

CRectTracker Class
Hierarchy Chart
CRectTracker::DrawTrackerRect
CRectTracker::CRectTracker
CRect::NormalizeRect

Other Resources

CRectTracker Members

CRectTracker::DrawTrackerRect

Called by the framework whenever the position of the tracker has changed while inside the **Track** or **TrackRubberBand** member function.

```
virtual void DrawTrackerRect(
   LPCRECT lpRect,
   CWnd* pWndClipTo,
   CDC* pDC,
   CWnd* pWnd
);
```

Parameters

lpRect

Pointer to the **RECT** that contains the rectangle to draw.

pWndClipTo

Pointer to the window to use in clipping the rectangle.

pDC

Pointer to the device context on which to draw.

pWnd

Pointer to the window on which the drawing will occur.

Remarks

The default implementation makes a call to **CDC::DrawFocusRect**, which draws a dotted rectangle.

Override this function to provide different feedback during the tracking operation.

See Also

Reference

CRectTracker Class
Hierarchy Chart
CRectTracker::Track
CRectTracker::TrackRubberBand
CDC::DrawFocusRect

Other Resources

CRectTracker Members

CRectTracker::GetHandleMask

The framework calls this member function to retrieve the mask for a rectangle's resize handles.

```
virtual UINT GetHandleMask( ) const;
```

Return Value

The mask of a **CRectTracker** item's resize handles.

Remarks

The resize handles appear on the sides and corners of the rectangle and allow the user to control the shape and size of the rectangle.

A rectangle has 8 resize handles numbered 0–7. Each resize handle is represented by a bit in the mask; the value of that bit is 2^*n*, where *n* is the resize handle number. Bits 0–3 correspond to the corner resize handles, starting at the top left moving clockwise. Bits 4–7 correspond to the side resize handles starting at the top moving clockwise. The following illustration shows a rectangle's resize handles and their corresponding resize handle numbers and values:

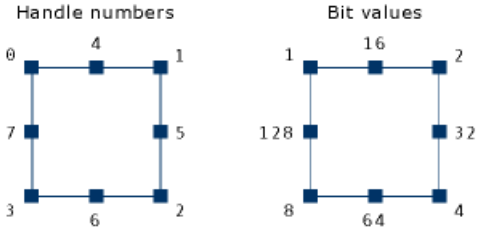

The default implementation of **GetHandleMask** returns the mask of the bits so that the resize handles appear. If the single bit is on, the corresponding resize handle will be drawn.

Override this member function to hide or show the indicated resize handles.

See Also

Reference

CRectTracker Class
Hierarchy Chart
CRectTracker::AdjustRect

Other Resources

CRectTracker Members

CRectTracker::GetTrueRect

Call this function to retrieve the coordinates of the rectangle.

```
void GetTrueRect(
   LPRECT lpTrueRect
) const;
```

Parameters

lpTrueRect

Pointer to the **RECT** structure that will contain the device coordinates of the **CRectTracker** object.

Remarks

The dimensions of the rectangle include the height and width of any resize handles located on the outer border. Upon returning, *lpTrueRect* is always a normalized rectangle in device coordinates.

See Also

Reference

CRectTracker Class

Hierarchy Chart

CRect::NormalizeRect

Other Resources

CRectTracker Members

CRectTracker::HitTest

Call this function to find out whether the user has grabbed a resize handle.

```
int HitTest(
   CPoint point
) const;
```

Parameters

point

The point, in device coordinates, to test.

Return Value

The value returned is based on the enumerated type **CRectTracker::TrackerHit** and can have one of the following values:

- **CRectTracker::hitNothing** –1
- **CRectTracker::hitTopLeft** 0
- **CRectTracker::hitTopRight** 1
- **CRectTracker::hitBottomRight** 2
- **CRectTracker::hitBottomLeft** 3
- **CRectTracker::hitTop** 4
- **CRectTracker::hitRight** 5
- **CRectTracker::hitBottom** 6
- **CRectTracker::hitLeft** 7
- **CRectTracker::hitMiddle** 8

See Also

Reference

CRectTracker Class
Hierarchy Chart
CRectTracker::NormalizeHit
CRectTracker::SetCursor

Other Resources

CRectTracker Members

CRectTracker::NormalizeHit

Call this function to convert a potentially inverted handle.

```
int NormalizeHit(
   int nHandle
) const;
```

Parameters

nHandle

Handle selected by the user.

Return Value

The index of the normalized handle.

Remarks

When **CRectTracker::Track** or **CRectTracker::TrackRubberBand** is called with inverting allowed, it is possible for the rectangle to be inverted on the x-axis, the y-axis, or both. When this happens, **HitTest** will return handles that are also inverted with respect to the rectangle. This is inappropriate for drawing cursor feedback because the feedback depends on the screen position of the rectangle, not the portion of the rectangle data structure that will be modified.

See Also

Reference

CRectTracker Class
Hierarchy Chart
CRectTracker::HitTest
CRectTracker::Track
CRectTracker::TrackRubberBand

Other Resources

CRectTracker Members

CRectTracker::OnChangedRect

Called by the framework whenever the tracker rectangle has changed during a call to **Track**.

```
virtual void OnChangedRect(
   const CRect& rectOld
);
```

Parameters

rectOld

Contains the old device coordinates of the **CRectTracker** object.

Remarks

At the time this function is called, all feedback drawn with **DrawTrackerRect** has been removed. The default implementation of this function does nothing.

Override this function when you want to perform any actions after the rectangle has been resized.

See Also

Reference

CRectTracker Class
Hierarchy Chart
CRectTracker::AdjustRect
CRectTracker::Track
CRectTracker::TrackRubberBand

Other Resources

CRectTracker Members

CRectTracker::SetCursor

Call this function to change the cursor shape while it is over the **CRectTracker** object's region.

```
BOOL SetCursor(
   CWnd* pWnd,
   UINT nHitTest
) const;
```

Parameters

pWnd

Points to the window that currently contains the cursor.

nHitTest

Results of the previous hit test, from the **WM_SETCURSOR** message.

Return Value

Nonzero if the previous hit was over the tracker rectangle; otherwise 0.

Remarks

Call this function from inside the function of your window that handles the **WM_SETCURSOR** message (typically **OnSetCursor**).

See Also

Reference

CRectTracker Class
Hierarchy Chart
CRectTracker::NormalizeHit
CRectTracker::HitTest
CWinApp::LoadCursor
CWnd::OnSetCursor

Other Resources

CRectTracker Members

CRectTracker::Track

Call this function to display the user interface for resizing the rectangle.

```
BOOL Track(
   CWnd* pWnd,
   CPoint point,
   BOOL bAllowInvert = FALSE,
   CWnd* pWndClipTo = NULL
);
```

Parameters

pWnd

The window object that contains the rectangle.

point

Device coordinates of the current mouse position relative to the client area.

bAllowInvert

If **TRUE**, the rectangle can be inverted along the x-axis or y-axis; otherwise **FALSE**.

pWndClipTo

The window that drawing operations will be clipped to. If **NULL**, *pWnd* is used as the clipping rectangle.

Return Value

If the ESC key is pressed, the tracking process is halted, the rectangle stored in the tracker is not altered, and 0 is returned. If the change is committed, by moving the mouse and releasing the left mouse button, the new position and/or size is recorded in the tracker's rectangle and nonzero is returned.

Remarks

This is usually called from inside the function of your application that handles the **WM_LBUTTONDOWN** message (typically **OnLButtonDown**).

This function will capture the mouse until the user releases the left mouse button, presses the ESC key, or presses the right mouse button. As the user moves the mouse cursor, the feedback is updated by calling **DrawTrackerRect** and **OnChangedRect**.

If *bAllowInvert* is **TRUE**, the tracking rectangle can be inverted on either the x-axis or y-axis.

See Also

Reference

CRectTracker Class
Hierarchy Chart
CRectTracker::DrawTrackerRect
CRectTracker::OnChangedRect
CRectTracker::CRectTracker
CRectTracker::TrackRubberBand

Other Resources

CRectTracker Members

CRectTracker::TrackRubberBand

Call this function to do rubber-band selection.

```
BOOL TrackRubberBand(
   CWnd* pWnd,
   CPoint point,
   BOOL bAllowInvert = TRUE
);
```

Parameters

pWnd

The window object that contains the rectangle.

point

Device coordinates of the current mouse position relative to the client area.

bAllowInvert

If **TRUE,** the rectangle can be inverted along the x-axis or y-axis; otherwise **FALSE**.

Return Value

Nonzero if the mouse has moved and the rectangle is not empty; otherwise 0.

Remarks

It is usually called from inside the function of your application that handles the **WM_LBUTTONDOWN** message (typically **OnLButtonDown**).

This function will capture the mouse until the user releases the left mouse button, presses the ESC key, or presses the right mouse button. As the user moves the mouse cursor, the feedback is updated by calling **DrawTrackerRect** and **OnChangedRect**.

Tracking is performed with a rubber-band-type selection from the lower-right handle. If inverting is allowed, the rectangle can be sized by dragging either up and to the left or down and to the right.

See Also

Reference

CRectTracker Class
Hierarchy Chart
CRectTracker::DrawTrackerRect
CRectTracker::OnChangedRect
CRectTracker::CRectTracker

Other Resources

CRectTracker Members

CRectTracker Data Members

For information about the data members in **CRectTracker**, see CRectTracker Members.

CRectTracker::m_nHandleSize

The size, in pixels, of the **CRectTracker** resize handles.

```
int m_nHandleSize;
```

Remarks

Initialized with the default system value.

See Also

Reference

CRectTracker Class

Hierarchy Chart

Other Resources

CRectTracker Members

CRectTracker::m_rect

The current position of the rectangle in client coordinates (pixels).

```
CRect m_rect;
```

See Also

Reference

CRectTracker Class
Hierarchy Chart
CRectTracker::CRectTracker
CRectTracker::Track
CRectTracker::TrackRubberBand

Other Resources

CRectTracker Members

CRectTracker::m_sizeMin

The minimum size of the rectangle.

```
CSize m_sizeMin;
```

Remarks

Both default values, **cx** and **cy**, are calculated from the default system value for the border width. This data member is used only by the **AdjustRect** member function.

See Also

Reference

CRectTracker Class
Hierarchy Chart
CRectTracker::Track
CRectTracker::TrackRubberBand
CRectTracker::AdjustRect

Other Resources

CRectTracker Members

CRectTracker::m_nStyle

Current style of the rectangle.

```
UINT m_nStyle;
```

Remarks

See CRectTracker::CRectTracker for a list of possible styles.

See Also

Reference

CRectTracker Class
Hierarchy Chart
CRectTracker::CRectTracker
CRectTracker::Draw

Other Resources

CRectTracker Members

CResourceException Class

Generated when Windows cannot find or allocate a requested resource.

```
class CResourceException : public CSimpleException
```

Remarks

No further qualification is necessary or possible.

For more information on using **CResourceException**, see the article Exception Handling (MFC).

Requirements

Header: afxwin.h

See Also

Reference

CException Class

Hierarchy Chart

Other Resources

CResourceException Members

CResourceException Members

Base Class Members

CObject Members

CException Members

Construction

CResourceException	Constructs a **CResourceException** object.

See Also

Reference

CResourceException Class

Hierarchy Chart

CResourceException Member Functions

For information about the member functions in **CResourceException**, see CResourceException Members.

CResourceException::CResourceException

Constructs a **CResourceException** object.

```
CResourceException( );
```

Remarks

Do not use this constructor directly, but rather call the global function AfxThrowResourceException. For more information about exceptions, see the article Exception Handling (MFC).

See Also

Reference

CResourceException Class
Hierarchy Chart
AfxThrowResourceException

Other Resources

CResourceException Members
Exception Processing

CRgn Class

Encapsulates a Windows graphics device interface (GDI) region.

```
class CRgn : public CGdiObject
```

Remarks

A region is an elliptical or polygonal area within a window. To use regions, you use the member functions of class **CRgn** with the clipping functions defined as members of class **CDC**.

The member functions of **CRgn** create, alter, and retrieve information about the region object for which they are called.

For more information on using **CRgn**, see Graphic Objects.

Requirements

Header: afxwin.h

See Also

Reference

CWnd Class

Hierarchy Chart

Other Resources

CRgn Members

CRgn Members

Base Class Members

CObject Members

CGdiObject Members

Construction

CRgn	Constructs a **CRgn** object.

Initialization

CombineRgn	Sets a **CRgn** object so that it is equivalent to the union of two specified **CRgn** objects.
CopyRgn	Sets a **CRgn** object so that it is a copy of a specified **CRgn** object.
CreateEllipticRgn	Initializes a **CRgn** object with an elliptical region.
CreateEllipticRgnIndirect	Initializes a **CRgn** object with an elliptical region defined by a RECT structure.
CreateFromData	Creates a region from the given region and transformation data.
CreateFromPath	Creates a region from the path that is selected into the given device context.
CreatePolygonRgn	Initializes a **CRgn** object with a polygonal region. The system closes the polygon automatically, if necessary, by drawing a line from the last vertex to the first.
CreatePolyPolygonRgn	Initializes a **CRgn** object with a region consisting of a series of closed polygons. The polygons may be disjoint, or they may overlap.
CreateRectRgn	Initializes a **CRgn** object with a rectangular region.
CreateRectRgnIndirect	Initializes a **CRgn** object with a rectangular region defined by a RECT structure.
CreateRoundRectRgn	Initializes a **CRgn** object with a rectangular region with rounded corners.

Operations

EqualRgn	Checks two **CRgn** objects to determine whether they are equivalent.
FromHandle	Returns a pointer to a **CRgn** object when given a handle to a Windows region.
GetRegionData	Fills the specified buffer with data describing the given region.
GetRgnBox	Retrieves the coordinates of the bounding rectangle of a **CRgn** object.
OffsetRgn	Moves a **CRgn** object by the specified offsets.
PtInRegion	Determines whether a specified point is in the region.
RectInRegion	Determines whether any part of a specified rectangle is within the boundaries of the region.
SetRectRgn	Sets the **CRgn** object to the specified rectangular region.

Operators

operator HRGN	Returns the Windows handle contained in the **CRgn** object.

See Also

Reference

CRgn Class

Hierarchy Chart

CRgn Member Functions

For information about the member functions in **CRgn**, see CRgn Members.

CRgn::CombineRgn

Creates a new GDI region by combining two existing regions.

```
int CombineRgn(
   CRgn* pRgn1,
   CRgn* pRgn2,
   int nCombineMode
);
```

Parameters

pRgn1

Identifies an existing region.

pRgn2

Identifies an existing region.

nCombineMode

Specifies the operation to be performed when combining the two source regions. It can be any one of the following values:

- **RGN_AND** Uses overlapping areas of both regions (intersection).
- **RGN_COPY** Creates a copy of region 1 (identified by *pRgn1*).
- **RGN_DIFF** Creates a region consisting of the areas of region 1 (identified by *pRgn1*) that are not part of region 2 (identified by *pRgn2*).
- **RGN_OR** Combines both regions in their entirety (union).
- **RGN_XOR** Combines both regions but removes overlapping areas.

Return Value

Specifies the type of the resulting region. It can be one of the following values:

- **COMPLEXREGION** New region has overlapping borders.
- **ERROR** No new region created.
- **NULLREGION** New region is empty.
- **SIMPLEREGION** New region has no overlapping borders.

Remarks

The regions are combined as specified by *nCombineMode*.

The two specified regions are combined, and the resulting region handle is stored in the **CRgn** object. Thus, whatever region is stored in the **CRgn** object is replaced by the combined region.

The size of a region is limited to 32,767 by 32,767 logical units or 64K of memory, whichever is smaller.

Use CopyRgn to simply copy one region into another region.

Example

```
CRgn   rgnA, rgnB, rgnC;

VERIFY(rgnA.CreateRectRgn( 50, 50, 150, 150 ));
VERIFY(rgnB.CreateRectRgn( 100, 100, 200, 200 ));
VERIFY(rgnC.CreateRectRgn( 0, 0, 50, 50 ));

int nCombineResult = rgnC.CombineRgn( &rgnA, &rgnB, RGN_OR );
ASSERT( nCombineResult != ERROR || nCombineResult != NULLREGION );
```

```
CBrush br1, br2, br3;
VERIFY(br1.CreateSolidBrush( RGB(255, 0, 0) ));  // rgnA Red
VERIFY(pDC->FrameRgn( &rgnA, &br1, 2, 2 ));
VERIFY(br2.CreateSolidBrush( RGB(0, 255, 0) ));  // rgnB Green
VERIFY(pDC->FrameRgn( &rgnB, &br2, 2, 2 ));
VERIFY(br3.CreateSolidBrush( RGB(0, 0, 255) ));  // rgnC Blue
VERIFY(pDC->FrameRgn( &rgnC, &br3, 2, 2 ));
```

See Also

Reference

CRgn Class
Hierarchy Chart
CRgn::CopyRgn
CombineRgn

Other Resources

CRgn Members

CRgn::CopyRgn

Copies the region defined by *pRgnSrc* into the **CRgn** object.

```
int CopyRgn(
   CRgn* pRgnSrc
);
```

Parameters

pRgnSrc

Identifies an existing region.

Return Value

Specifies the type of the resulting region. It can be one of the following values:

- **COMPLEXREGION** New region has overlapping borders.
- **ERROR** No new region created.
- **NULLREGION** New region is empty.
- **SIMPLEREGION** New region has no overlapping borders.

Remarks

The new region replaces the region formerly stored in the **CRgn** object. This function is a special case of the CombineRgn member function.

Example

See the example for CRgn::CreateEllipticRgn.

See Also

Reference

CRgn Class
Hierarchy Chart
CRgn::CombineRgn
CombineRgn

Other Resources

CRgn Members

CRgn::CreateEllipticRgn

Creates an elliptical region.

```
BOOL CreateEllipticRgn(
   int x1,
   int y1,
   int x2,
   int y2
);
```

Parameters

x1

Specifies the logical x-coordinate of the upper-left corner of the bounding rectangle of the ellipse.

y1

Specifies the logical y-coordinate of the upper-left corner of the bounding rectangle of the ellipse.

x2

Specifies the logical x-coordinate of the lower-right corner of the bounding rectangle of the ellipse.

y2

Specifies the logical y-coordinate of the lower-right corner of the bounding rectangle of the ellipse.

Return Value

Nonzero if the operation succeeded; otherwise 0.

Remarks

The region is defined by the bounding rectangle specified by *x1*, *y1*, *x2*, and *y2*. The region is stored in the **CRgn** object.

The size of a region is limited to 32,767 by 32,767 logical units or 64K of memory, whichever is smaller.

When it has finished using a region created with the **CreateEllipticRgn** function, an application should select the region out of the device context and use the **DeleteObject** function to remove it.

Example

```
CRgn   rgnA, rgnB, rgnC;

VERIFY(rgnA.CreateEllipticRgn(200, 100, 350, 250));
VERIFY(rgnB.CreateRectRgn( 0, 0, 50, 50 ));
VERIFY(rgnB.CopyRgn( &rgnA ));
int nOffsetResult = rgnB.OffsetRgn( -75, 75 );
ASSERT( nOffsetResult != ERROR && nOffsetResult != NULLREGION );

VERIFY(rgnC.CreateRectRgn( 0, 0, 1, 1));
int nCombineResult = rgnC.CombineRgn( &rgnA, &rgnB, RGN_AND );
ASSERT( nCombineResult != ERROR && nOffsetResult != NULLREGION );

CBrush brA, brB, brC;
VERIFY(brC.CreateHatchBrush( HS_FDIAGONAL, RGB(0, 0, 255) )); // Blue
VERIFY(pDC->FillRgn( &rgnC, &brC ));
VERIFY(brA.CreateSolidBrush( RGB(255, 0, 0) ));  // rgnA Red
VERIFY(pDC->FrameRgn( &rgnA, &brA, 2, 2 ));
VERIFY(brB.CreateSolidBrush( RGB(0, 255, 0) ));  // rgnB Green
VERIFY(pDC->FrameRgn( &rgnB, &brB, 2, 2 ));
```

See Also

Reference

CRgn Class

Hierarchy Chart

CRgn::CreateEllipticRgnIndirect
CreateEllipticRgn

Other Resources

CRgn Members

CRgn::CreateEllipticRgnIndirect

Creates an elliptical region.

```
BOOL CreateEllipticRgnIndirect(
   LPCRECT lpRect
);
```

Parameters

lpRect

Points to a **RECT** structure or a **CRect** object that contains the logical coordinates of the upper-left and lower-right corners of the bounding rectangle of the ellipse.

Return Value

Nonzero if the operation succeeded; otherwise 0.

Remarks

The region is defined by the structure or object pointed to by *lpRect* and is stored in the **CRgn** object.

The size of a region is limited to 32,767 by 32,767 logical units or 64K of memory, whichever is smaller.

When it has finished using a region created with the **CreateEllipticRgnIndirect** function, an application should select the region out of the device context and use the **DeleteObject** function to remove it.

Example

See the example for CRgn::CreateRectRgnIndirect.

See Also

Reference

CRgn Class
Hierarchy Chart
CRgn::CreateEllipticRgn
CreateEllipticRgnIndirect

Other Resources

CRgn Members

CRgn::CreateFromData

Creates a region from the given region and transformation data.

```
BOOL CreateFromData(
   const XFORM* lpXForm,
   int nCount,
   const RGNDATA* pRgnData
);
```

Parameters

lpXForm

Points to an XFORM data structure that defines the transformation to be performed on the region. If this pointer is **NULL**, the identity transformation is used.

nCount

Specifies the number of bytes pointed to by *pRgnData*.

pRgnData

Points to a RGNDATA data structure that contains the region data.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

An application can retrieve data for a region by calling the **CRgn::GetRegionData** function.

See Also

Reference

CRgn Class
Hierarchy Chart
CRgn::GetRegionData
ExtCreateRegion

Other Resources

CRgn Members

CRgn::CreateFromPath

Creates a region from the path that is selected into the given device context.

```
BOOL CreateFromPath(
   CDC* pDC
);
```

Parameters

pDC

Identifies a device context that contains a closed path.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The device context identified by the *pDC* parameter must contain a closed path. After **CreateFromPath** converts a path into a region, Windows discards the closed path from the device context.

See Also

Reference

CRgn Class
Hierarchy Chart
CDC::BeginPath
CDC::EndPath
CDC::SetPolyFillMode

Other Resources

CRgn Members

CRgn::CreatePolygonRgn

Creates a polygonal region.

```
BOOL CreatePolygonRgn(
   LPPOINT lpPoints,
   int nCount,
   int nMode
);
```

Parameters

lpPoints

Points to an array of **POINT** structures or an array of **CPoint** objects. Each structure specifies the x-coordinate and y-coordinate of one vertex of the polygon. The **POINT** structure has the following form:

```
typedef struct tagPOINT {
   int x;
   int y;
} POINT;
```

nCount

Specifies the number of **POINT** structures or **CPoint** objects in the array pointed to by *lpPoints*.

nMode

Specifies the filling mode for the region. This parameter may be either **ALTERNATE** or **WINDING**.

Return Value

Nonzero if the operation succeeded; otherwise 0.

Remarks

The system closes the polygon automatically, if necessary, by drawing a line from the last vertex to the first. The resulting region is stored in the **CRgn** object.

The size of a region is limited to 32,767 by 32,767 logical units or 64K of memory, whichever is smaller.

When the polygon-filling mode is **ALTERNATE**, the system fills the area between odd-numbered and even-numbered polygon sides on each scan line. That is, the system fills the area between the first and second side, between the third and fourth side, and so on.

When the polygon-filling mode is **WINDING**, the system uses the direction in which a figure was drawn to determine whether to fill an area. Each line segment in a polygon is drawn in either a clockwise or a counterclockwise direction. Whenever an imaginary line drawn from an enclosed area to the outside of a figure passes through a clockwise line segment, a count is incremented. When the line passes through a counterclockwise line segment, the count is decremented. The area is filled if the count is nonzero when the line reaches the outside of the figure.

When an application has finished using a region created with the **CreatePolygonRgn** function, it should select the region out of the device context and use the **DeleteObject** function to remove it.

Example

```
CRgn   rgnA, rgnB;

CPoint ptVertex[5];

ptVertex[0].x = 180;
ptVertex[0].y = 80;
ptVertex[1].x = 100;
ptVertex[1].y = 160;
ptVertex[2].x = 120;
ptVertex[2].y = 260;
```

```
ptVertex[3].x = 240;
ptVertex[3].y = 260;
ptVertex[4].x = 260;
ptVertex[4].y = 160;

VERIFY(rgnA.CreatePolygonRgn( ptVertex, 5, ALTERNATE));

CRect rectRgnBox;
int nRgnBoxResult = rgnA.GetRgnBox( &rectRgnBox );
ASSERT( nRgnBoxResult != ERROR || nRgnBoxResult != NULLREGION );

CBrush brA, brB;
VERIFY(brA.CreateSolidBrush( RGB(255, 0, 0) ));  // rgnA Red
VERIFY(pDC->FrameRgn( &rgnA, &brA, 2, 2 ));
VERIFY(brB.CreateSolidBrush( RGB(0, 0, 255) ));  // Blue
rectRgnBox.InflateRect(3,3);
pDC->FrameRect( &rectRgnBox, &brB );
```

See Also

Reference

CRgn Class

Hierarchy Chart

CRgn::CreatePolyPolygonRgn

CreatePolygonRgn

Other Resources

CRgn Members

CRgn::CreatePolyPolygonRgn

Creates a region consisting of a series of closed polygons.

```
BOOL CreatePolyPolygonRgn(
   LPPOINT lpPoints,
   LPINT lpPolyCounts,
   int nCount,
   int nPolyFillMode
);
```

Parameters

lpPoints

Points to an array of **POINT** structures or an array of **CPoint** objects that defines the vertices of the polygons. Each polygon must be explicitly closed because the system does not close them automatically. The polygons are specified consecutively. The **POINT** structure has the following form:

```
typedef struct tagPOINT {
   int x;
   int y;
} POINT;
```

lpPolyCounts

Points to an array of integers. The first integer specifies the number of vertices in the first polygon in the *lpPoints* array, the second integer specifies the number of vertices in the second polygon, and so on.

nCount

Specifies the total number of integers in the *lpPolyCounts* array.

nPolyFillMode

Specifies the polygon-filling mode. This value may be either **ALTERNATE** or **WINDING**.

Return Value

Nonzero if the operation succeeded; otherwise 0.

Remarks

The resulting region is stored in the **CRgn** object.

The polygons may be disjoint, or they may overlap.

The size of a region is limited to 32,767 by 32,767 logical units or 64K of memory, whichever is smaller.

When the polygon-filling mode is **ALTERNATE**, the system fills the area between odd-numbered and even-numbered polygon sides on each scan line. That is, the system fills the area between the first and second side, between the third and fourth side, and so on.

When the polygon-filling mode is **WINDING**, the system uses the direction in which a figure was drawn to determine whether to fill an area. Each line segment in a polygon is drawn in either a clockwise or a counterclockwise direction. Whenever an imaginary line drawn from an enclosed area to the outside of a figure passes through a clockwise line segment, a count is incremented. When the line passes through a counterclockwise line segment, the count is decremented. The area is filled if the count is nonzero when the line reaches the outside of the figure.

When an application has finished using a region created with the **CreatePolyPolygonRgn** function, it should select the region out of the device context and use the CGDIObject::DeleteObject member function to remove it.

See Also

Reference

CRgn Class

Hierarchy Chart

CRgn::CreatePolygonRgn
CDC::SetPolyFillMode
CreatePolyPolygonRgn

Other Resources

CRgn Members

CRgn::CreateRectRgn

Creates a rectangular region that is stored in the **CRgn** object.

```
BOOL CreateRectRgn(
   int x1,
   int y1,
   int x2,
   int y2
);
```

Parameters

x1

Specifies the logical x-coordinate of the upper-left corner of the region.

y1

Specifies the logical y-coordinate of the upper-left corner of the region.

x2

Specifies the logical x-coordinate of the lower-right corner of the region.

y2

Specifies the logical y-coordinate of the lower-right corner of the region.

Return Value

Nonzero if the operation succeeded; otherwise 0.

Remarks

The size of a region is limited to 32,767 by 32,767 logical units or 64K of memory, whichever is smaller.

When it has finished using a region created by **CreateRectRgn**, an application should use the CGDIObject::DeleteObject member function to remove the region.

Example

```
CRgn   rgn;

BOOL bSucceeded = rgn.CreateRectRgn( 50, 20, 150, 120 );
ASSERT( bSucceeded == TRUE );
```

For an additional example, see CRgn::CombineRgn.

See Also

Reference

CRgn Class
Hierarchy Chart
CRgn::CreateRectRgnIndirect
CRgn::CreateRoundRectRgn
CreateRectRgn

Other Resources

CRgn Members

CRgn::CreateRectRgnIndirect

Creates a rectangular region that is stored in the **CRgn** object.

```
BOOL CreateRectRgnIndirect(
   LPCRECT lpRect
);
```

Parameters

lpRect

Points to a **RECT** structure or **CRect** object that contains the logical coordinates of the upper-left and lower-right corners of the region. The **RECT** structure has the following form:

```
typedef struct tagRECT {
   int left;
   int top;
   int right;
   int bottom;
} RECT;
```

Return Value

Nonzero if the operation succeeded; otherwise 0.

Remarks

The size of a region is limited to 32,767 by 32,767 logical units or 64K of memory, whichever is smaller.

When it has finished using a region created by **CreateRectRgnIndirect**, an application should use the CGDIObject::DeleteObject member function to remove the region.

Example

```
CRgn   rgnA, rgnB, rgnC;

CRect rectA(50, 50, 150, 150);
CRect rectB(100, 50, 200, 150);

VERIFY(rgnA.CreateRectRgnIndirect(&rectA));
VERIFY(rgnB.CreateEllipticRgnIndirect(&rectB));
VERIFY(rgnC.CreateRectRgn( 0, 0, 50, 50 ));

int nCombineResult = rgnC.CombineRgn( &rgnA, &rgnB, RGN_AND );
ASSERT( nCombineResult != ERROR || nCombineResult != NULLREGION );

CBrush brA, brB, brC;
VERIFY(brA.CreateSolidBrush( RGB(255, 0, 0) ));
VERIFY(pDC->FrameRgn( &rgnA, &brA, 2, 2 ));      // rgnA Red

VERIFY(brB.CreateSolidBrush( RGB(0, 255, 0) ));
VERIFY(pDC->FrameRgn( &rgnB, &brB, 2, 2 ));      // rgnB Green
VERIFY(brC.CreateSolidBrush( RGB(0, 0, 255) ));  // rgnC Blue
VERIFY(pDC->FrameRgn( &rgnC, &brC, 2, 2 ));
```

See Also

Reference

CRgn Class
Hierarchy Chart
CRgn::CreateRectRgn
CRgn::CreateRoundRectRgn
CreateRectRgnIndirect

Other Resources

CRgn::CreateRoundRectRgn

Creates a rectangular region with rounded corners that is stored in the **CRgn** object.

```
BOOL CreateRoundRectRgn(
   int x1,
   int y1,
   int x2,
   int y2,
   int x3,
   int y3
);
```

Parameters

x1

Specifies the logical x-coordinate of the upper-left corner of the region.

y1

Specifies the logical y-coordinate of the upper-left corner of the region.

x2

Specifies the logical x-coordinate of the lower-right corner of the region.

y2

Specifies the logical y-coordinate of the lower-right corner of the region.

x3

Specifies the width of the ellipse used to create the rounded corners.

y3

Specifies the height of the ellipse used to create the rounded corners.

Return Value

Nonzero if the operation succeeded; otherwise 0.

Remarks

The size of a region is limited to 32,767 by 32,767 logical units or 64K of memory, whichever is smaller.

When an application has finished using a region created with the **CreateRoundRectRgn** function, it should select the region out of the device context and use the CGDIObject::DeleteObject member function to remove it.

Example

```
CRgn   rgnA, rgnB, rgnC;

VERIFY(rgnA.CreateRoundRectRgn( 50, 50, 150, 150, 30, 30 ));
VERIFY(rgnB.CreateRoundRectRgn( 200, 75, 250, 125, 50, 50 ));
VERIFY(rgnC.CreateRectRgn( 0, 0, 50, 50 ));

int nCombineResult = rgnC.CombineRgn( &rgnA, &rgnB, RGN_OR );
ASSERT( nCombineResult != ERROR || nCombineResult != NULLREGION );

CBrush brA, brB, brC;
VERIFY(brA.CreateSolidBrush( RGB(255, 0, 0) ));
VERIFY(pDC->FillRgn( &rgnA, &brA));      // rgnA Red Filled

VERIFY(brB.CreateSolidBrush( RGB(0, 255, 0) ));
VERIFY(pDC->FillRgn( &rgnB, &brB));      // rgnB Green Filled
VERIFY(brC.CreateSolidBrush( RGB(0, 0, 255) ));  // rgnC Blue
VERIFY(pDC->FrameRgn( &rgnC, &brC, 2, 2 ));
```

See Also

Reference

CRgn Class

Hierarchy Chart

CRgn::CreateRectRgn

CRgn::CreateRectRgnIndirect

CreateRoundRectRgn

Other Resources

CRgn Members

CRgn::CRgn

Constructs a **CRgn** object.

```
CRgn( );
```

Remarks

The **m_hObject** data member does not contain a valid Windows GDI region until the object is initialized with one or more of the other **CRgn** member functions.

Example

See the example for CRgn::CreateRoundRectRgn.

See Also

Reference

CRgn Class

Hierarchy Chart

Other Resources

CRgn Members

CRgn::EqualRgn

Determines whether the given region is equivalent to the region stored in the **CRgn** object.

```
BOOL EqualRgn(
   CRgn* pRgn
) const;
```

Parameters

pRgn

Identifies a region.

Return Value

Nonzero if the two regions are equivalent; otherwise 0.

Example

```
CRgn   rgnA, rgnB;

VERIFY(rgnA.CreateEllipticRgn(200, 100, 350, 250));
VERIFY(rgnB.CreateRectRgn( 0, 0, 50, 50 ));
VERIFY(rgnB.CopyRgn( &rgnA ));
int nOffsetResult = rgnB.OffsetRgn( -75, 75 );
ASSERT( nOffsetResult != ERROR || nOffsetResult != NULLREGION );
ASSERT( FALSE == rgnB.EqualRgn( &rgnA ) );
```

See Also

Reference

CRgn Class

Hierarchy Chart

EqualRgn

Other Resources

CRgn Members

CRgn::FromHandle

Returns a pointer to a **CRgn** object when given a handle to a Windows region.

```
static CRgn* PASCAL FromHandle(
   HRGN hRgn
);
```

Parameters

hRgn

Specifies a handle to a Windows region.

Return Value

A pointer to a **CRgn** object. If the function was not successful, the return value is **NULL**.

Remarks

If a **CRgn** object is not already attached to the handle, a temporary **CRgn** object is created and attached. This temporary **CRgn** object is valid only until the next time the application has idle time in its event loop, at which time all temporary graphic objects are deleted. Another way of saying this is that the temporary object is only valid during the processing of one window message.

See Also

Reference

CRgn Class

Hierarchy Chart

Other Resources

CRgn Members

CRgn::GetRegionData

Fills the specified buffer with data describing the region.

```
int GetRegionData(
   LPRGNDATA lpRgnData,
   int nCount
) const;
```

Parameters

lpRgnData

Points to a RGNDATA data structure that receives the information. If this parameter is **NULL**, the return value contains the number of bytes needed for the region data.

nCount

Specifies the size, in bytes, of the *lpRgnData* buffer.

Return Value

If the function succeeds and *nCount* specifies an adequate number of bytes, the return value is always *nCount*. If the function fails, or if *nCount* specifies less than adequate number of bytes, the return value is 0 (error).

Remarks

This data includes the dimensions of the rectangles that make up the region. This function is used in conjunction with the **CRgn::CreateFromData** function.

See Also

Reference

CRgn Class
Hierarchy Chart
CRgn::CreateFromData

Other Resources

CRgn Members

CRgn::GetRgnBox

Retrieves the coordinates of the bounding rectangle of the **CRgn** object.

```
int GetRgnBox(
   LPRECT lpRect
) const;
```

Parameters

lpRect

Points to a **RECT** structure or **CRect** object to receive the coordinates of the bounding rectangle. The **RECT** structure has the following form:

```
typedef struct tagRECT {
   int left;
   int top;
   int right;
   int bottom;
} RECT;
```

Return Value

Specifies the region's type. It can be any of the following values:

- **COMPLEXREGION** Region has overlapping borders.
- **NULLREGION** Region is empty.
- **ERROR** **CRgn** object does not specify a valid region.
- **SIMPLEREGION** Region has no overlapping borders.

Example

See the example for CRgn::CreatePolygonRgn.

See Also

Reference

CRgn Class
Hierarchy Chart
GetRgnBox

Other Resources

CRgn Members

CRgn::OffsetRgn

Moves the region stored in the **CRgn** object by the specified offsets.

```
int OffsetRgn(
   int x,
   int y
);
int OffsetRgn(
   POINT point
);
```

Parameters

x

Specifies the number of units to move left or right.

y

Specifies the number of units to move up or down.

point

The x-coordinate of *point* specifies the number of units to move left or right. The y-coordinate of *point* specifies the number of units to move up or down. The *point* parameter may be either a **POINT** structure or a **CPoint** object.

Return Value

The new region's type. It can be any one of the following values:

- **COMPLEXREGION** Region has overlapping borders.
- **ERROR** Region handle is not valid.
- **NULLREGION** Region is empty.
- **SIMPLEREGION** Region has no overlapping borders.

Remarks

The function moves the region *x* units along the x-axis and *y* units along the y-axis.

The coordinate values of a region must be less than or equal to 32,767 and greater than or equal to –32,768. The *x* and *y* parameters must be carefully chosen to prevent invalid region coordinates.

Example

See the example for CRgn::CreateEllipticRgn.

See Also

Reference

CRgn Class
Hierarchy Chart
OffsetRgn

Other Resources

CRgn Members

CRgn::PtInRegion

Checks whether the point given by *x* and *y* is in the region stored in the **CRgn** object.

```
BOOL PtInRegion(
   int x,
   int y
) const;
BOOL PtInRegion(
   POINT point
) const;
```

Parameters

x

Specifies the logical x-coordinate of the point to test.

y

Specifies the logical y-coordinate of the point to test.

point

The x- and y-coordinates of *point* specify the x- and y-coordinates of the point to test the value of. The *point* parameter can either be a **POINT** structure or a **CPoint** object.

Return Value

Nonzero if the point is in the region; otherwise 0.

See Also

Reference

CRgn Class

Hierarchy Chart

PtInRegion

Other Resources

CRgn Members

CRgn::RectInRegion

Determines whether any part of the rectangle specified by *lpRect* is within the boundaries of the region stored in the **CRgn** object.

```
BOOL RectInRegion(
   LPCRECT lpRect
) const;
```

Parameters

lpRect

Points to a **RECT** structure or **CRect** object. The **RECT** structure has the following form:

```
typedef struct tagRECT {
   int left;
   int top;
   int right;
   int bottom;
} RECT;
```

Return Value

Nonzero if any part of the specified rectangle lies within the boundaries of the region; otherwise 0.

See Also

Reference

CRgn Class

Hierarchy Chart

RectInRegion

Other Resources

CRgn Members

CRgn::SetRectRgn

Creates a rectangular region.

```
void SetRectRgn(
   int x1,
   int y1,
   int x2,
   int y2
);
void SetRectRgn(
   LPCRECT lpRect
);
```

Parameters

x1

Specifies the x-coordinate of the upper-left corner of the rectangular region.

y1

Specifies the y-coordinate of the upper-left corner of the rectangular region.

x2

Specifies the x-coordinate of the lower-right corner of the rectangular region.

y2

Specifies the y-coordinate of the lower-right corner of the rectangular region.

lpRect

Specifies the rectangular region. Can be either a pointer to a **RECT** structure or a **CRect** object.

Remarks

Unlike CreateRectRgn, however, it does not allocate any additional memory from the local Windows application heap. Instead, it uses the space allocated for the region stored in the **CRgn** object. This means that the **CRgn** object must already have been initialized with a valid Windows region before calling **SetRectRgn**. The points given by *x1*, *y1*, *x2*, and *y2* specify the minimum size of the allocated space.

Use this function instead of the **CreateRectRgn** member function to avoid calls to the local memory manager.

See Also

Reference

CRgn Class
Hierarchy Chart
CRgn::CreateRectRgn
SetRectRgn

Other Resources

CRgn Members

CRgn Operators

For information about the operators in **CRgn**, see CRgn Members.

CRgn::operator HRGN

Use this operator to get the attached Windows GDI handle of the **CRgn** object.

```
operator HRGN( ) const;
```

Return Value

If successful, a handle to the Windows GDI object represented by the **CRgn** object; otherwise **NULL**.

Remarks

This operator is a casting operator, which supports direct use of an **HRGN** object.

For more information about using graphic objects, see the article Graphic Objects in the *Platform SDK*.

See Also

Reference

CRgn Class

Hierarchy Chart

Other Resources

CRgn Members

CRichEditCntrItem Class

With CRichEditView and CRichEditDoc, provides the functionality of the rich edit control within the context of MFC's document view architecture.

```
class CRichEditCntrItem : public COleClientItem
```

Remarks

A "rich edit control" is a window in which the user can enter and edit text. The text can be assigned character and paragraph formatting, and can include embedded OLE objects. Rich edit controls provide a programming interface for formatting text. However, an application must implement any user interface components necessary to make formatting operations available to the user.

CRichEditView maintains the text and formatting characteristic of text. **CRichEditDoc** maintains the list of OLE client items which are in the view. **CRichEditCntrItem** provides container-side access to the OLE client item.

This Windows Common control (and therefore the CRichEditCtrl and related classes) is available only to programs running under Windows 95/98 and Windows NT versions 3.51 and later.

For an example of using rich edit container items in an MFC application, see the WORDPAD sample application.

Requirements

Header: afxrich.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

WORDPAD Sample: The Windows Application

Reference

COleClientItem Class

Hierarchy Chart

CRichEditDoc Class

CRichEditView Class

Other Resources

CRichEditCntrItem Members

CRichEditCntrItem Members

Base Class Members

CObject Members

CCmdTarget Members

CDocItem Members

COleClientItem Members

Constructor

CRichEditCntrItem	Constructs a **CRichEditCntrItem** object.

Operations

SyncToRichEditObject	Activates the item as another type.

See Also

Reference

CRichEditCntrItem Class
Hierarchy Chart

CRichEditCntrItem Member Functions

For information about the member functions in **CRichEditCntrItem**, see CRichEditCntrItem Members.

CRichEditCntrItem::CRichEditCntrItem

Call this function to create a **CRichEditCntrItem** object and add it to the container document.

```
CRichEditCntrItem(
   REOBJECT* preo = NULL,
   CRichEditDoc* pContainer = NULL
);
```

Parameters

preo

Pointer to an REOBJECT structure which describes an OLE item. The new **CRichEditCntrItem** object is constructed around this OLE item. If *preo* is **NULL**, the client item is empty.

pContainer

Pointer to the container document that will contain this item. If *pContainer* is **NULL**, you must explicitly call COleDocument::AddItem to add this client item to a document.

Remarks

This function does not perform any OLE initialization.

For more information, see the REOBJECT structure in the *Platform SDK*.

See Also

Reference

CRichEditCntrItem Class
Hierarchy Chart
COleDocument::AddItem
CRichEditDoc Class

Other Resources

CRichEditCntrItem Members

CRichEditCntrItem::SyncToRichEditObject

Call this function to synchronize the device aspect, DVASPECT, of this **CRichEditCntrItem** to that specified by *reo*.

```
void SyncToRichEditObject(
   REOBJECT& reo
);
```

Parameters

reo

Reference to an REOBJECT structure which describes an OLE item.

Remarks

For more information, see DVASPECT in the *Platform SDK*.

See Also

Reference

CRichEditCntrItem Class

Hierarchy Chart

Other Resources

CRichEditCntrItem Members

CRichEditCtrl Class

Provides the functionality of the rich edit control.

```
class CRichEditCtrl : public CWnd
```

Remarks

A "rich edit control" is a window in which the user can enter and edit text. The text can be assigned character and paragraph formatting, and can include embedded OLE objects. Rich edit controls provide a programming interface for formatting text. However, an application must implement any user interface components necessary to make formatting operations available to the user.

This Windows Common control (and therefore the **CRichEditCtrl** class) is available only to programs running under Windows 95/98 and Windows NT versions 3.51 and later. The **CRichEditCtrl** class supports versions 2.0 and 3.0 of the Platform SDK rich edit control.

Caution
If you are using a rich edit control in a dialog box (regardless whether your application is SDI, MDI, or dialog-based), you must call AfxInitRichEdit once before the dialog box is displayed. A typical place to call this function is in your program's **InitInstance** member function. You do not need to call it for each time you display the dialog box, only the first time. You do not have to call **AfxInitRichEdit** if you are working with **CRichEditView**.

For more information on using **CRichEditCtrl**, see:

- Controls
- Using CRichEditCtrl
- Knowledge Base article Q259949 : INFO: SetCaretPos() Is Not Appropriate with CEdit or CRichEditCtrl Controls

For an example of using a rich edit control in an MFC application, see the WORDPAD sample application.

Requirements

Header: afxcmn.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

WORDPAD Sample: The Windows Application

Reference

CWnd Class

Hierarchy Chart

CEdit Class

CRichEditView Class

Other Resources

CRichEditCtrl Members

CRichEditCtrl Members

Base Class Members

Construction

Line Operations

Selection Operations

Formatting Operations

Editing Operations

General Operations

Clipboard Operations

OLE Operations

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Construction

Create	Creates the Windows rich edit control and associates it with this **CRichEditCtrl** object.
CreateEx	Creates the Windows rich edit control with the specified extended Windows styles and associates it with this **CRichEditCtrl** object.
CRichEditCtrl	Constructs a **CRichEditCtrl** object.

Line Operations

CharFromPos	Retrieves information about the character closest to a specified point in the client area of an edit control.
GetFirstVisibleLine	Determines the topmost visible line in this **CRichEditCtrl** object.
GetLine	Retrieves a line of text from this **CRichEditCtrl** object.
GetLineCount	Retrieves the number of lines in this **CRichEditCtrl** object.
LineFromChar	Determines which line contains the given character.
LineIndex	Retrieves the character index of a given line in this **CRichEditCtrl** object.
LineLength	Retrieves the length of a given line in this **CRichEditCtrl** object.
LineScroll	Scrolls the text in this **CRichEditCtrl** object.
PosFromChar	Retrieves the client area coordinates of a specified character in an edit control.

Selection Operations

Clear	Clears the current selection.
GetSel	Gets the starting and ending positions of the current selection in this **CRichEditCtrl** object.

GetSelectionType	Retrieves the type of contents in the current selection in this **CRichEditCtrl** object.
GetSelText	Gets the text of the current selection in this **CRichEditCtrl** object
GetTextRange	Retrieves the specified range of text.
HideSelection	Shows or hides the current selection.
ReplaceSel	Replaces the current selection in this **CRichEditCtrl** object with specified text.
SetSel	Sets the selection in this **CRichEditCtrl** object.

Formatting Operations

GetDefaultCharFormat	Retrieves the current default character formatting attributes in this **CRichEditCtrl** object.
GetParaFormat	Retrieves the paragraph formatting attributes in the current selection in this **CRichEditCtrl** object.
GetSelectionCharFormat	Retrieves the character formatting attributes in the current selection in this **CRichEditCtrl** object.
SetDefaultCharFormat	Sets the current default character formatting attributes in this **CRichEditCtrl** object.
SetParaFormat	Sets the paragraph formatting attributes in the current selection in this **CRichEditCtrl** object.
SetSelectionCharFormat	Sets the character formatting attributes in the current selection in this **CRichEditCtrl** object.
SetWordCharFormat	Sets the character formatting attributes in the current word in this **CRichEditCtrl** object.

Editing Operations

CanRedo	Determines whether there are any actions in the control's redo queue.
CanUndo	Determines if an editing operation can be undone.
EmptyUndoBuffer	Resets (clears) the undo flag of this **CRichEditCtrl** object.
GetRedoName	Retrieves the type of the next action, if any, in the control's redo queue.
GetUndoName	Retrieves the type of the next undo action, if any.
Redo	Redoes the next action in the control's redo queue.
SetUndoLimit	Sets the maximum number of actions that can stored in the undo queue.
StopGroupTyping	Stops the control from collecting additional typing actions into the current undo action. The control stores t he next typing action, if any, into a new action in the undo queue.
StreamIn	Inserts text from an input stream into this **CRichEditCtrl** object.
StreamOut	Stores text from this **CRichEditCtrl** object into an output stream.
Undo	Reverses the last editing operation.

General Operations

DisplayBand	Displays a portion of the contents of this **CRichEditCtrl** object.

FindText	Locates text within this **CRichEditCtrl** object.
FindWordBreak	Finds the next word break before or after the specified character position, or retrieves information about the character at that position.
FormatRange	Formats a range of text for the target output device.
GetCharPos	Determines the location of a given character within this **CRichEditCtrl** object.
GetEventMask	Retrieves the event mask for this **CRichEditCtrl** object.
GetLimitText	Gets the limit on the amount of text a user can enter into this **CRichEditCtrl** object.
GetModify	Determines if the contents of this **CRichEditCtrl** object have changed since the last save.
GetOptions	Retrieves the rich edit control options.
GetPunctuation	Retrieves the current punctuation characters for the rich edit control. This message is available only in Asian-language versions of the operating system.
GetRect	Retrieves the formatting rectangle for this **CRichEditCtrl** object.
GetTextLength	Retrieves the length of the text, in characters, in this **CRichEditCtrl** object. Does not include the terminating null character.
GetTextLengthEx	Retrieves the number of characters or bytes in the rich edit view. Accepts a list of flags to indicate the method of determining length of the text in a rich edit control
GetTextMode	Retrieves the current text mode and undo level of a rich edit control.
GetWordWrapMode	Retrieves the current word wrapping and word breaking options for the rich edit control. This message is available only in Asian-language versions of the operating system.
LimitText	Limits the amount of text a user can enter into the **CRichEditCtrl** object.
RequestResize	Forces this **CRichEditCtrl** object to send request resize notifications.
SetAutoURLDetect	Indicates if the auto URL detection is active in a rich edit control.
SetBackgroundColor	Sets the background color in this **CRichEditCtrl** object.
SetEventMask	Sets the event mask for this **CRichEditCtrl** object.
SetModify	Sets or clears the modification flag for this **CRichEditCtrl** object.
SetOptions	Sets the options for this **CRichEditCtrl** object.
SetPunctuation	Sets the punctuation characters for a rich edit control. This message is available only in Asian-language versions of the operating system.
SetReadOnly	Sets the read-only option for this **CRichEditCtrl** object.
SetRect	Sets the formatting rectangle for this **CRichEditCtrl** object.

SetTargetDevice	Sets the target output device for this **CRichEditCtrl** object.
SetTextMode	Sets the text mode or undo level of a rich edit control. The message fails if the control contains text.
SetWordWrapMode	Sets the word-wrapping and word-breaking options for the rich edit control. This message is available only in Asian-language versions of the operating system.

Clipboard Operations

CanPaste	Determines if the contents of the Clipboard can be pasted into this rich edit control.
Copy	Copies the current selection to the Clipboard.
Cut	Cuts the current selection to the Clipboard.
Paste	Inserts the contents of the Clipboard into this rich edit control.
PasteSpecial	Inserts the contents of the Clipboard into this rich edit control in the specified data format.

OLE Operations

GetIRichEditOle	Retrieves a pointer to the **IRichEditOle** interface for this rich edit control.
SetOLECallback	Sets the **IRichEditOleCallback** COM object for this rich edit control.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart

CRichEditCtrl Member Functions

For information about the member functions in **CRichEditCtrl**, see CRichEditCtrl Members.

CRichEditCtrl::CanPaste

Determines if the rich edit control can paste the specified Clipboard format.

```
BOOL CanPaste(
   UINT nFormat = 0
) const;
```

Parameters

nFormat

The Clipboard data format to query. This parameter can be one of the predefined Clipboard formats or the value returned by RegisterClipboardFormat.

Return Value

Nonzero if the Clipboard format can be pasted; otherwise 0.

Remarks

If *nFormat* is 0, **CanPaste** will try any format currently on the Clipboard.

For more information, see EM_CANPASTE message and RegisterClipboardFormat function in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;

// Paste the clipboard data if possible.
if (pmyRichEditCtrl->CanPaste())
{
   pmyRichEditCtrl->Paste();
}
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::Paste
CRichEditCtrl::PasteSpecial

Other Resources

CRichEditCtrl Members

CRichEditCtrl::CanRedo

Determines if the redo queue contains any actions.

```
BOOL CanRedo( ) const;
```

Return Value

Nonzero if the redo queue contains actions, otherwise 0.

Remarks

To discover the name of the operation in the redo queue, call CRichEditCtrl::GetRedoName. To redo the most recent Undo operation, call Redo.

For more information, see EM_CANREDO in the *Platform SDK*.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::Undo
CRichEditCtrl::CanUndo

Other Resources

CRichEditCtrl Members

CRichEditCtrl::CanUndo

Determines if the last editing operation can be undone.

```
BOOL CanUndo( ) const;
```

Return Value

Nonzero if the last edit operation can be undone by a call to the Undo member function; 0 if it cannot be undone.

Remarks

For more information, see EM_CANUNDO in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;

// Undo the last operation, if possible.
if (pmyRichEditCtrl->CanUndo())
   pmyRichEditCtrl->Undo();
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::Undo
CRichEditCtrl::EmptyUndoBuffer
CRichEditCtrl::CanRedo
CRichEditCtrl::GetRedoName

Other Resources

CRichEditCtrl Members

CRichEditCtrl::CharFromPos

Retrieves information about the character at the point specified by the parameter *pt*.

```
int CharFromPos(
   CPoint pt
) const;
```

Parameters

pt

A CPoint object containing the coordinates of the specified point.

Return Value

The zero-based character index of the character nearest the specified point. If the specified point is beyond the last character in the control, the return value indicates the last character in the control.

Remarks

This member function works with a rich edit control. To get the information for an edit control, call CEdit::CharFromPos.

For more information, see EM_CHARFROMPOS in the *Platform SDK*.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::PosFromChar

Other Resources

CRichEditCtrl Members

CRichEditCtrl::Clear

Deletes (clears) the current selection (if any) in the rich edit control.

```
void Clear( );
```

Remarks

The deletion performed by **Clear** can be undone by calling the Undo member function.

To delete the current selection and place the deleted contents onto the Clipboard, call the Cut member function.

For more information, see WM_CLEAR in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;

// Delete all of the text.
pmyRichEditCtrl->SetSel(0, -1);
pmyRichEditCtrl->Clear();
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::Undo
CRichEditCtrl::Cut
CRichEditCtrl::Copy
CRichEditCtrl::Paste

Other Resources

CRichEditCtrl Members

CRichEditCtrl::Copy

Copies the current selection (if any) in the rich edit control to the Clipboard.

```
void Copy( );
```

Remarks

For more information, see WM_COPY in the *Platform SDK*.

Example

```
void snip_CRichEditCtrl_Copy()
{
   // The pointer to my rich edit control.
   extern CRichEditCtrl* pmyRichEditCtrl;

   // Copy all of the text to the clipboard.
   pmyRichEditCtrl->SetSel(0, -1);
   pmyRichEditCtrl->Copy();
}
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::Paste
CRichEditCtrl::Cut

Other Resources

CRichEditCtrl Members

CRichEditCtrl::Create

Creates the Windows rich edit control and associates it with this **CRichEditCtrl** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the edit control's style. Apply a combination of the window styles listed in the **Remarks** section below, and edit control styles, described in the *Platform SDK*.

rect

Specifies the edit control's size and position. Can be a CRect object or RECT structure.

pParentWnd

Specifies the edit control's parent window (often a CDialog). It must not be **NULL**.

nID

Specifies the edit control's ID.

Return Value

Nonzero if initialization is successful; otherwise, 0.

Remarks

You construct a **CRichEditCtrl** object in two steps. First, call the CRichEditCtrl constructor, then call **Create**, which creates the Windows edit control and attaches it to the **CRichEditCtrl** object.

When you create a rich edit control with this function, first you must load the necessary common controls library. To load the libary, call the global function AfxInitRichEdit, which in turn initializes the common controls library. You need to call **AfxInitRichEdit** only once in your process.

When **Create** executes, Windows sends the WM_NCCREATE, WM_NCCALCSIZE, WM_CREATE, and WM_GETMINMAXINFO messages to the edit control.

These messages are handled by default by the OnNcCreate, OnNcCalcSize, OnCreate, and OnGetMinMaxInfo member functions in the **CWnd** base class. To extend the default message handling, derive a class from **CRichEditCtrl**, add a message map to the new class, and override the above message-handler member functions. Override **OnCreate**, for example, to perform needed initialization for the new class.

Apply the following window styles to an edit control.

- **WS_CHILD** Always.
- **WS_VISIBLE** Usually.
- **WS_DISABLED** Rarely.
- **WS_GROUP** To group controls.
- **WS_TABSTOP** To include edit control in the tabbing order.

For more information about window styles, see CreateWindow in the Platform SDK.

Example

```
// pParentWnd is an external pointer to the parent window.
extern CWnd* pParentWnd;
```

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;

pmyRichEditCtrl->Create(
   WS_CHILD|WS_VISIBLE|WS_BORDER|ES_MULTILINE,
   CRect(10,10,100,200), pParentWnd, 1);
```

See Also

Reference

CRichEditCtrl Class

Hierarchy Chart

CRichEditCtrl::CreateEx

CRichEditCtrl::CRichEditCtrl

Other Resources

CRichEditCtrl Members

CRichEditCtrl::CreateEx

Creates a control (a child window) and associates it with the **CRichEditCtrl** object.

```
virtual BOOL CreateEx(
   DWORD dwExStyle,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwExStyle

Specifies the extended style of the control being created. For a list of extended Windows styles, see the *dwExStyle* parameter for CreateWindowEx in the *Platform SDK*.

dwStyle

Specifies the edit control's style. Apply a combination of the window styles listed in the **Remarks** section of Create and edit control styles, described in the *Platform SDK*.

rect

A reference to a RECT structure describing the size and position of the window to be created, in client coordinates of *pParentWnd*.

pParentWnd

A pointer to the window that is the control's parent.

nID

The control's child-window ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Use **CreateEx** instead of **Create** to apply extended Windows styles, specified by the Windows extended style preface **WS_EX_**.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::Create

Other Resources

CRichEditCtrl Members

CRichEditCtrl::CRichEditCtrl

Constructs a **CRichEditCtrl** object.

```
CRichEditCtrl( );
```

Remarks

Use Create to construct the Windows rich edit control.

Example

```
// Declare a local CRichEditCtrl object.
CRichEditCtrl myRichEditCtrl;

// Declare a dynamic CRichEditCtrl object.
CRichEditCtrl* pmyRichEditCtrl = new CRichEditCtrl;
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::Create
CRichEditCtrl::CreateEx

Other Resources

CRichEditCtrl Members

CRichEditCtrl::Cut

Delete (cuts) the current selection (if any) in the rich edit control and copies the deleted text to the Clipboard.

```
void Cut( );
```

Remarks

The deletion performed by **Cut** can be undone by calling the Undo member function.

To delete the current selection without placing the deleted text into the Clipboard, call the Clear member function.

For more information, see WM_CUT in the *Platform SDK*.

Example

```
void snip_CRichEditCtrl_Cut()
{
   // The pointer to my rich edit control.
   extern CRichEditCtrl* pmyRichEditCtrl;

   // Delete all of the text and copy it to the clipboard.
   pmyRichEditCtrl->SetSel(0, -1);
   pmyRichEditCtrl->Cut();
}
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::Copy
CRichEditCtrl::Undo
CRichEditCtrl::Clear

Other Resources

CRichEditCtrl Members

CRichEditCtrl::DisplayBand

Displays a portion of the contents of the rich edit control (text and OLE items), as previously formatted by FormatRange.

```
BOOL DisplayBand(
   LPRECT pDisplayRect
);
```

Parameters

pDisplayRect

Pointer to a RECT or CRect object specifying the area of the device to display the text.

Return Value

Nonzero if the display of the formatted text succeeds, otherwise, 0.

Remarks

The text and OLE items are clipped to the area specified by the pointer *pDisplayRect*.

For more information, see EM_DISPLAYBAND in the *Platform SDK*.

Example

See the example for CRichEditCtrl::FormatRange.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::FormatRange

Other Resources

CRichEditCtrl Members

CRichEditCtrl::EmptyUndoBuffer

Resets (clear) the undo flag of this rich edit control.

```
void EmptyUndoBuffer( );
```

Remarks

The control will now be unable to undo the last editing operation. The undo flag is set whenever an operation within the rich edit control can be undone.

The undo flag is automatically cleared whenever you call the CWnd member function SetWindowText.

For more information, see EM_EMPTYUNDOBUFFER in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;

// Clear the undo buffer.
if (pmyRichEditCtrl->CanUndo())
{
   pmyRichEditCtrl->EmptyUndoBuffer();
   ASSERT(!pmyRichEditCtrl->CanUndo());
}
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::CanUndo
CRichEditCtrl::Undo
CWnd::SetWindowText

Other Resources

CRichEditCtrl Members

CRichEditCtrl::FindText

Finds text within the rich edit control.

```
long FindText(
   DWORD dwFlags,
   FINDTEXTEX* pFindText
) const;
```

Parameters

dwFlags

For a list of possible values, see *wParam* in EM_FINDTEXTEXT in the *Platform SDK*.

pFindText

Pointer to the FINDTEXTEX structure giving the parameters for the search and returning the range where the match was found.

Return Value

Zero-based character position of the next match; – 1 if there are no more matches.

Remarks

You can search either up or down by setting the proper range parameters in the CHARRANGE structure within the **FINDTEXTEX** structure.

For more information, see EM_FINDTEXTEX message and FINDTEXTEX structure in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;
// The string to search for.
extern LPCTSTR lpszmyString;

// Set the selection to be the first occurrence of the
// string lpszmyString, if it is found.
FINDTEXTEX ft;
ft.chrg.cpMin = 0;
ft.chrg.cpMax = -1;
ft.lpstrText = (LPSTR) lpszmyString;
long n = pmyRichEditCtrl->FindText(FR_MATCHCASE|FR_WHOLEWORD, &ft);
if (n != -1)
   pmyRichEditCtrl->SetSel(ft.chrgText);
```

See Also

Reference

CRichEditCtrl Class

Hierarchy Chart

CRichEditCtrl::SetSel

Other Resources

CRichEditCtrl Members

CRichEditCtrl::FindWordBreak

Finds the next word break before or after the position specified by *nStart*.

```
DWORD FindWordBreak(
   UINT nCode,
   DWORD nStart
) const;
```

Parameters

nCode

Indicates the action to take. For a list of possible values, see the description for the parameter *code* in **EM_FINDWORDBREAK** in the *Platform SDK*.

nStart

The zero-based character position from which to start.

Return Value

Based on the parameter *nCode*. For more information, see EM_FINDWORDBREAK in the *Platform SDK*.

Remarks

You can use this member function to retrieve information about a character at a given position.

See Also

Reference

CRichEditCtrl Class

Hierarchy Chart

Other Resources

CRichEditCtrl Members

CRichEditCtrl::FormatRange

Formats a range of text in a rich edit control for a specific device.

```
long FormatRange(
   FORMATRANGE* pfr,
   BOOL bDisplay = TRUE
);
```

Parameters

pfr

Pointer to the FORMATRANGE structure which contains information about the output device. **NULL** indicates that cached information within the rich edit control can be freed.

bDisplay

Indicates if the text should be rendered. If **FALSE**, the text is just measured.

Return Value

The index of the last character that fits in the region plus one.

Remarks

Typically, this call is followed by a call to DisplayBand.

For more information, see EM_FORMATRANGE message and FORMATRANGE structure in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;
// A pointer to a printer DC.
extern CDC* pMyPrinterDC;

FORMATRANGE fr;

// Get the page width and height from the printer.
long lPageWidth = ::MulDiv(pMyPrinterDC->GetDeviceCaps(PHYSICALWIDTH),
    1440, pMyPrinterDC->GetDeviceCaps(LOGPIXELSX));
long lPageHeight = ::MulDiv(pMyPrinterDC->GetDeviceCaps(PHYSICALHEIGHT),
    1440, pMyPrinterDC->GetDeviceCaps(LOGPIXELSY));
CRect rcPage(0, 0, lPageWidth, lPageHeight);

// Format the text and render it to the printer.
fr.hdc = pMyPrinterDC->m_hDC;
fr.hdcTarget = pMyPrinterDC->m_hDC;
fr.rc = rcPage;
fr.rcPage = rcPage;
fr.chrg.cpMin = 0;
fr.chrg.cpMax = -1;
pmyRichEditCtrl->FormatRange(&fr, TRUE);

// Update the display with the new formatting.
RECT rcClient;
pmyRichEditCtrl->GetClientRect(&rcClient);
pmyRichEditCtrl->DisplayBand(&rcClient);
```

See Also

Reference

CRichEditCtrl Class

Hierarchy Chart

CRichEditCtrl::DisplayBand

Other Resources

CRichEditCtrl::GetCharPos

Gets the position (top-left corner) of a given character within this **CRichEditCtrl** object.

```
CPoint GetCharPos(
   long lChar
) const;
```

Parameters

lChar

Zero-based index of the character.

Return Value

The location of the top-left corner of the character specified by *lChar*.

Remarks

The character is specified by giving its zero-based index value. If *lChar* is greater than the index of the last character in this **CRichEditCtrl** object, the return value specifies the coordinates of the character position just past the last character in this **CRichEditCtrl** object.

For more information, see EM_POSFROMCHAR in the *Platform SDK*.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::FindText

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetDefaultCharFormat

Gets the default character formatting attributes of this **CRichEditCtrl** object.

```
DWORD GetDefaultCharFormat(
   CHARFORMAT& cf
) const;
DWORD GetDefaultCharFormat(
   CHARFORMAT2& cf
) const;
```

Parameters

cf

In the first version, a pointer to a CHARFORMAT structure holding the default character formatting attributes.

In the second version, a pointer to a CHARFORMAT2 structure, which is a Rich Edit 2.0 extension to the **CHARFORMAT** structure, holding the default character formatting attributes.

Return Value

The **dwMask** data member of *cf*. It specified the default character formatting attributes.

Remarks

For more information, see the EM_GETCHARFORMAT message and the **CHARFORMAT** and **CHARFORMAT2** structures in the *Platform SDK*.

Example

See the example for SetDefaultCharFormat.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::SetDefaultCharFormat
CRichEditCtrl::GetSelectionCharFormat
CRichEditCtrl::GetParaFormat

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetEventMask

Gets the event mask for this **CRichEditCtrl** object.

```
long GetEventMask( ) const;
```

Return Value

The event mask for this **CRichEditCtrl** object.

Remarks

The event mask specifies which notification messages the **CRichEditCtrl** object sends to its parent window.

For more information, see EM_GETEVENTMASK in the *Platform SDK*.

Example

See the example for CRichEditCtrl::SetEventMask.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::SetEventMask
CRichEditCtrl::CRichEditCtrl

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetFirstVisibleLine

Determines the topmost visible line in this **CRichEditCtrl** object.

```
int GetFirstVisibleLine( ) const;
```

Return Value

Zero-based index of the uppermost visible line in this **CRichEditCtrl** object.

Remarks

For more information, see EM_GETFIRSTVISIBLELINE in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;

int nFirstVisible = pmyRichEditCtrl->GetFirstVisibleLine();

// Scroll the rich edit control so that the first visible line
// is the first line of text.
if (nFirstVisible > 0)
{
   pmyRichEditCtrl->LineScroll(-nFirstVisible, 0);
}
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::GetLine
CRichEditCtrl::GetLineCount

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetIRichEditOle

Accesses the **IRichEditOle** interface for this **CRichEditCtrl** object.

```
IRichEditOle* GetIRichEditOle( ) const;
```

Return Value

Pointer to the IRichEditOle interface that can be used to access this **CRichEditCtrl** object's OLE functionality; **NULL** if the interface is not accessible.

Remarks

Use this interface to access this **CRichEditCtrl** object's OLE functionality.

For more information, see EM_GETOLEINTERFACE message and IRichEditOle interface in the *Platform SDK*.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::SetOLECallback

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetLimitText

Gets the text limit for this **CRichEditCtrl** object.

```
long GetLimitText( ) const;
```

Return Value

The current text limit, in bytes, for this **CRichEditCtrl** object.

Remarks

The text limit is the maximum amount of text, in bytes, the rich edit control can accept.

For more information, see EM_GETLIMITTEXT in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;
// The new text of the rich edit control.
extern LPCTSTR lpszmyString;
int nLength = strlen(lpszmyString);

// Want the text limit to be at least the size of the new string.
if (pmyRichEditCtrl->GetLimitText() < nLength)
   pmyRichEditCtrl->LimitText(nLength);

pmyRichEditCtrl->SetWindowText(lpszmyString);
```

See Also

Reference

CRichEditCtrl Class

Hierarchy Chart

CRichEditCtrl::LimitText

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetLine

Retrieves a line of text from this **CRichEditCtrl** object.

```
int GetLine(
   int nIndex,
   LPTSTR lpszBuffer
) const;
int GetLine(
   int nIndex,
   LPTSTR lpszBuffer,
   int nMaxLength
) const;
```

Parameters

nIndex

Zero-based index of the line to retrieve.

lpszBuffer

Points to the buffer to receive the text. The first word of the buffer must specify the maximum number of bytes that can be copied into the buffer.

nMaxLength

Maximum number of characters that can be copied into *lpszBuffer*. The second form of **GetLine** places this value into the first word of the buffer specified by *lpszBuffer*.

Return Value

The number of characters copied into *lpszBuffer*.

Remarks

The copied line does not contain a terminating null character.

Note
Because the first word of the buffer stores the number of characters to be copied, make sure that your buffer is at least 4 byt es long.

For more information, see EM_GETLINE in the *Platform SDK*.

Example

See the example for GetLineCount.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::LineLength

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetLineCount

Retrieves the number of lines in the **CRichEditCtrl** object.

```
int GetLineCount( ) const;
```

Return Value

The number of lines in this **CRichEditCtrl** object.

Remarks

For more information, see EM_GETLINECOUNT in the *Platform SDK*.

Example

```
#ifdef _DEBUG
   // The pointer to my rich edit control.
   extern CRichEditCtrl* pmyRichEditCtrl;

   int i, nLineLength, nLineCount = pmyRichEditCtrl->GetLineCount();
   CString strText, strLine;

   // Dump every line of text of the rich edit control.
   for (i=0;i < nLineCount;i++)
   {
      nLineLength = pmyRichEditCtrl->LineLength(i);
      pmyRichEditCtrl->GetLine(i, strText.GetBuffer(nLineLength));
      strText.ReleaseBuffer(nLineLength);

      strLine.Format(TEXT("line %d: '%s'\r\n"), i, strText.GetBuffer(0));
      afxDump << strLine;
   }
#endif
```

See Also

Reference

CRichEditCtrl Class

Hierarchy Chart

CRichEditCtrl::GetLine

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetModify

Determines if the contents of this **CRichEditCtrl** object have been modified.

```
BOOL GetModify( ) const;
```

Return Value

Nonzero if the text in this **CRichEditCtrl** object has been modified; otherwise 0.

Remarks

Windows maintains an internal flag indicating whether the contents of the rich edit control have been changed. This flag is cleared when the edit control is first created and can also be cleared by calling the SetModify member function.

For more information, see EM_GETMODIFY in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;

// Reset the modified state only if the text has been modified.
if (pmyRichEditCtrl->GetModify())
   pmyRichEditCtrl->SetModify(FALSE);
```

See Also

Reference

CRichEditCtrl Class

Hierarchy Chart

CRichEditCtrl::SetModify

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetOptions

Retrieves the options currently set for the rich edit control.

```
UINT GetOptions( ) const;
```

Return Value

A combination of the current option flag values. For a list of these values, see the *fOptions* parameter in the EM_SETOPTIONS message, as described in the *Platform SDK*.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::SetOptions

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetParaFormat

Gets the paragraph formatting attributes of the current selection.

```
DWORD GetParaFormat(
   PARAFORMAT& pf
) const;
DWORD GetParaFormat(
   PARAFORMAT2& pf
) const;
```

Parameters

pf

In the first version, a pointer to a PARAFORMAT structure to hold the paragraph formatting attributes of the current selection.

In the second version, a pointer to a PARAFORMAT2 structure, which is a Rich Edit 2.0 extension to the **PARAFORMAT** structure, holding the default character formatting attributes.

Return Value

The **dwMask** data member of *pf*. It specifies the paragraph formatting attributes that are consistent throughout the current selection.

Remarks

If more than one paragraph is selected, *pf* receives the attributes of the first selected paragraph. The return value specifies which attributes are consistent throughout the selection.

For more information, see the EM_GETPARAFORMAT message and the **PARAFORMAT** and **PARAFORMAT2** structures in the *Platform SDK*.

Example

See the example for CRichEditCtrl::SetParaFormat.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::SetParaFormat
CRichEditCtrl::GetSelectionCharFormat

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetPunctuation

Gets the current punctuation characters in a rich edit control.

```
BOOL GetPunctuation(
   UINT fType,
   PUNCTUATION* lpPunc
) const;
```

Parameters

fType

The punctuation type flag, as described in the *fType* parameter of EM_GETPUNCTUATION in the *Platform SDK*.

lpPunc

A pointer to a PUNCTUATION structure, as described in the *Platform SDK*.

Return Value

Nonzero if the operation succeeded, otherwise 0.

Remarks

This member function is available with only the Asian-language versions of the operating system.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::SetPunctuation

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetRect

Retrieves the formatting rectangle for this **CRichEditCtrl** object.

```
void GetRect(
   LPRECT lpRect
) const;
```

Parameters

lpRect

CRect or pointer to a RECT to receive the formatting rectangle of this **CRichEditCtrl** object.

Remarks

The formatting rectangle is the bounding rectangle for the text. This value is independent of the size of the **CRichEditCtrl** object.

For more information, see EM_GETRECT in the *Platform SDK*.

Example

See the example for LimitText.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::SetRect

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetRedoName

Retrieves the type of the next available action in the redo queue, if any.

```
UNDONAMEID GetRedoName( ) const;
```

Return Value

If successful, **GetRedoName** returns the UNDONAMEID enumeration type indicating the type of the next action in the control's redo queue. If the redo queue is empty, or if the redo action in the queue is of an unknown type, **GetRedoName** returns 0.

Remarks

The types of actions that can be undone or redone include typing, delete, drag-drop, cut, and paste operations. This information can be useful for applications that provide an extended user interface for Undo and Redo operations, such as a drop-down list box of redoable actions.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::Redo
CRichEditCtrl::Undo
CRichEditCtrl::GetUndoName

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetSel

Retrieves the bounds of the current selection in this **CRichEditCtrl** object.

```
void GetSel(
   CHARRANGE& cr
) const;
void GetSel(
   long& nStartChar,
   long& nEndChar
) const;
```

Parameters

cr

Reference to a CHARRANGE structure to receive the bounds of the current selection.

nStartChar

Zero-based index of the first character in the current selection.

nEndChar

Zero-based index of the last character in the current selection.

Remarks

The two forms of this function provide alternate ways to get the bounds for the selection. Brief descriptions of these forms follow:

- **GetSel(** *cr* **)** This form uses the **CHARRANGE** structure with its **cpMin** and **cpMax** members to return the bounds.
- **GetSel(** *nStartChar* **,** *nEndChar* **)** This form returns the bounds in the parameters *nStartChar* and *nEndChar*.

The selection includes everything if the beginning (**cpMin** or *nStartChar*) is 0 and the end (**cpMax** or *nEndChar*) is – 1.

For more information, see EM_EXGETSEL message and CHARRANGE structure in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;

// Set the selection to be all characters after the current selection.
long nStartChar, nEndChar;

pmyRichEditCtrl->GetSel(nStartChar, nEndChar);
pmyRichEditCtrl->SetSel(nEndChar, -1);
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::SetSel
CRichEditCtrl::GetSelText
CRichEditCtrl::GetParaFormat
CRichEditCtrl::GetSelectionCharFormat

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetSelectionCharFormat

Gets the character formatting attributes of the current selection.

```
DWORD GetSelectionCharFormat(
   CHARFORMAT& cf
) const;
DWORD GetSelectionCharFormat(
   CHARFORMAT2& cf
) const;
```

Parameters

cf

In the first version, a pointer to a CHARFORMAT structure to receive the character formatting attributes of the current selection.

In the second version, a pointer to a CHARFORMAT2 structure, which is a Rich Edit 2.0 extension to the **CHARFORMAT** structure to receive the character formatting attributes of the current selection.

Return Value

The **dwMask** data member of *cf*. It specifies the character formatting attributes that are consistent throughout the current selection.

Remarks

The *cf* parameter receives the attributes of the first character in the current selection. The return value specifies which attributes are consistent throughout the selection.

For more information, see the EM_GETCHARFORMAT message and the **CHARFORMAT** and **CHARFORMAT2** structures in the *Platform SDK*.

Example

See the example for SetSelectionCharFormat.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::GetDefaultCharFormat
CRichEditCtrl::GetParaFormat
CRichEditCtrl::SetSelectionCharFormat
CRichEditCtrl::GetSelText

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetSelectionType

Determines the selection type in this **CRichEditCtrl** object.

```
WORD GetSelectionType( ) const;
```

Return Value

Flags indicating the contents of the current selection. A combination of the following flags:

- **SEL_EMPTY** Indicates that there is no current selection.
- **SEL_TEXT** Indicates that the current selection contains text.
- **SEL_OBJECT** Indicates that the current selection contains at least one OLE item.
- **SEL_MULTICHAR** Indicates that the current selection contains more than one character of text.
- **SEL_MULTIOBJECT** Indicates that the current selection contains more than one OLE object.

Remarks

For more information, see EM_SELECTIONTYPE in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;

// Dump the selection text only if it contains at least one text character.
if (pmyRichEditCtrl->GetSelectionType() & (SEL_TEXT | SEL_MULTICHAR))
{
   CString strText = pmyRichEditCtrl->GetSelText();

   TRACE(TEXT("selection text is '%s'.\r\n"), (LPCSTR) strText);
}
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::GetSel
CRichEditCtrl::GetSelText

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetSelText

Retrieves the text from the current selection in this **CRichEditCtrl** object.

```
long GetSelText(
   LPSTR lpBuf
) const;
CString GetSelText( ) const;
```

Parameters

lpBuf

Pointer to the buffer to receive the text in the current selection.

Return Value

Depends on the form:

- **GetSelText(** *lpBuf* **)** The number of characters copied into *lpBuf*, not including the null termination.
- **GetSelText( )** The string containing the current selection.

Remarks

If you use the first form, **GetSelText(** *lpBuf* **)**, you must ensure that the buffer is large enough for the text it will receive. Call GetSel to determine the number of characters in the current selection.

For more information, see EM_GETSELTEXT in the *Platform SDK*.

Example

See the example for CRichEditCtrl::GetSelectionType.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::GetSel
CRichEditCtrl::GetSelectionType

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetTextLength

Retrieves the length of the text, in characters, in this **CRichEditCtrl** object, not including the terminating null character.

```
long GetTextLength( ) const;
```

Return Value

The length of the text in this **CRichEditCtrl** object.

Remarks

For more information, see WM_GETTEXTLENGTH in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;

// Limit the rich edit controls text to the number of
// characters currently in it.
pmyRichEditCtrl->LimitText(pmyRichEditCtrl->GetTextLength());
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::LimitText
CRichEditCtrl::GetLimitText

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetTextLengthEx

Calculates the length of the text in the rich edit control.

```
long GetTextLengthEx(
   DWORD dwFlags,
   UINT uCodePage = -1
) const;
```

Parameters

dwFlags

Value specifying the method to be used in determining the text length. This member can be one or more of the values listed in the flags member of GETTEXTLENGTHEX described in the *Platform SDK*.

uCodePage

Code page for translation (CP_ACP for ANSI Code Page, 1200 for Unicode).

Return Value

The number of characters or bytes in the edit control. If incompatible flags were set in *dwFlags*, this member function returns **E_INVALIDARG**.

Remarks

GetTextLengthEx provides additional ways of determining the length of the text. It supports the Rich Edit 2.0 functionality. See About Rich Edit Controls in the *Platform SDK* for more information.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::GetTextRange

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetTextMode

Retrieves the current text mode and undo level of a rich edit control.

```
UINT GetTextMode( ) const;
```

Return Value

A set of bit flags from the TEXTMODE enumeration type, as described in the *Platform SDK*. The flags indicate the current text mode and undo level of the control.

See Also

Reference

CRichEditCtrl Class

Hierarchy Chart

CRichEditCtrl::SetTextMode

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetTextRange

Gets the specified range of characters.

```
int GetTextRange(
   int nFirst,
   int nLast,
   CString& refString
) const;
```

Parameters

nFirst

The character position index immediately preceding the first character in the range.

nLast

The character position immediately following the last character in the range.

refString

A reference to a CString object that will receive the text.

Return Value

The number of characters copied, not including the terminating null character.

Remarks

For more information, see EM_GETTEXTRANGE in the *Platform SDK*.

GetTextRange supports the Rich Edit 2.0 functionality. See About Rich Edit Controls in the *Platform SDK* for more information.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::GetTextLength
CRichEditCtrl::GetTextLengthEx

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetUndoName

Retrieves the type of the next available action in the undo queue, if any.

```
UNDONAMEID GetUndoName( ) const;
```

Return Value

If an undo action is in the control's undo queue, **GetUndoName** returns the UNDONAMEID enumeration type indicating the type of the next action in the queue. If the undo queue is empty, or if the undo action in the queue is of an unknown type, **GetUndoName** returns 0.

Remarks

The types of actions that can be undone or redone include typing, delete, drag-drop, cut, and paste operations. This information can be useful for applications that provide an extended user interface for Undo and Redo operations, such as a drop-down list box of actions that can be undone.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::Undo
CRichEditCtrl::Redo
CRichEditCtrl::GetRedoName

Other Resources

CRichEditCtrl Members

CRichEditCtrl::GetWordWrapMode

Retrieves the current word wrapping and word breaking options for the rich edit control.

```
UINT GetWordWrapMode( ) const;
```

Return Value

The current word wrapping and word breaking options. These options are described in EM_SETWORDWRAPMODE in the *Platform SDK*.

Remarks

This member function is available only for Asian-language versions of the operating system.

See Also

Reference

CRichEditCtrl Class

Hierarchy Chart

Other Resources

CRichEditCtrl Members

CRichEditCtrl::HideSelection

Changes the visibility of the selection.

```
void HideSelection(
   BOOL bHide,
   BOOL bPerm
);
```

Parameters

bHide

Indicates if the selection should be shown or hidden, **TRUE** to hide the selection.

bPerm

Indicates if this change in visibility for the selection should be permanent.

Remarks

When *bPerm* is **TRUE**, it changes the **ECO_NOHIDESEL** option for this **CRichEditCtrl** object. For a brief description of this option, see SetOptions. You can use this function to set all the options for this **CRichEditCtrl** object.

For more information, see EM_HIDESELECTION in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;

// Show the selection and make it permanent.
pmyRichEditCtrl->HideSelection(FALSE, TRUE);
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::SetSel
CRichEditCtrl::GetSelectionType

Other Resources

CRichEditCtrl Members

CRichEditCtrl::LimitText

Limits the length of the text that the user can enter into an edit control.

```
void LimitText(
   long nChars = 0
);
```

Parameters

nChars

Specifies the length (in bytes) of the text that the user can enter. If this parameter is 0 (the default value), the text length is set to 64K bytes.

Remarks

Changing the text limit restricts only the text the user can enter. It has no effect on any text already in the edit control, nor does it affect the length of the text copied to the edit control by the SetWindowText member function in **CWnd**. If an application uses the **SetWindowText** function to place more text into an edit control than is specified in the call to **LimitText**, the user can delete any of the text within the edit control. However, the text limit will prevent the user from replacing the existing text with new text, unless deleting the current selection causes the text to fall below the text limit.

Note
For the text limit, each OLE item counts as a single character.

For more information, see EM_EXLIMITTEXT in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;

// Limit the number of characters to be the maximum number visible.

// Get the text metrics for the edit; needed for the
// average character width.
TEXTMETRIC tm;
CDC* pDC = pmyRichEditCtrl->GetDC();
pDC->GetTextMetrics(&tm);
pmyRichEditCtrl->ReleaseDC(pDC);

CRect r;
pmyRichEditCtrl->GetRect(&r);
pmyRichEditCtrl->LimitText(r.Width()/tm.tmAveCharWidth);
```

See Also

Reference

CRichEditCtrl Class

Hierarchy Chart

CRichEditCtrl::GetLimitText

Other Resources

CRichEditCtrl Members

CRichEditCtrl::LineFromChar

Retrieves the line number of the line that contains the specified character index.

```
long LineFromChar(
   long nIndex
) const;
```

Parameters

nIndex

Contains the zero-based index value for the desired character in the text of the edit control, or contains –1. If *nIndex* is –1, it specifies the current line, that is, the line that contains the caret.

Return Value

The zero-based line number of the line containing the character index specified by *nIndex*. If *nIndex* is –1, the number of the line that contains the first character of the selection is returned. If there is no selection, the current line number is returned.

Remarks

A character index is the number of characters from the beginning of the rich edit control. For character counting, an OLE item is counted as a single character.

For more information, see EM_EXLINEFROMCHAR in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;
// The index of the char to get information on.
extern int nIndex;

CString strText;

pmyRichEditCtrl->GetWindowText(strText);
strText = strText.Mid(nIndex, 1);

// Dump the index, character and line number.
TRACE("nIndex = %d, character = %c, line = %d\r\n",
   nIndex, strText[0], pmyRichEditCtrl->LineFromChar(nIndex));
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::GetLineCount
CRichEditCtrl::GetLine
CRichEditCtrl::LineIndex

Other Resources

CRichEditCtrl Members

CRichEditCtrl::LineIndex

Retrieves the character index of a line within this **CRichEditCtrl** object.

```
int LineIndex(
   int nLine = -1
) const;
```

Parameters

nLine

Contains the index value for the desired line in the text of the edit control, or contains –1. If *nLine* is –1, it specifies the current line, that is, the line that contains the caret.

Return Value

The character index of the line specified in *nLine* or –1 if the specified line number is greater then the number of lines in the edit control.

Remarks

The character index is the number of characters from the beginning of the rich edit control to the specified line.

For more information, see EM_LINEINDEX in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;
// The string for replacing.
extern LPCTSTR lpszmyString;

int nBegin, nEnd;

// Replace the second line, if it exists, of the rich edit control
// with the text lpszmyString.
if ((nBegin=pmyRichEditCtrl->LineIndex(1)) != -1)
{
   nEnd = nBegin + pmyRichEditCtrl->LineLength(1);
   pmyRichEditCtrl->SetSel(nBegin, nEnd);
   pmyRichEditCtrl->ReplaceSel(lpszmyString);
}
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::LineFromChar
CRichEditCtrl::GetLineCount

Other Resources

CRichEditCtrl Members

CRichEditCtrl::LineLength

Retrieves the length of a line in a rich edit control.

```
int LineLength(
   int nLine = -1
) const;
```

Parameters

nLine

Specifies the character index of a character in the line whose length is to be retrieved. If this parameter is –1, the length of the current line (the line that contains the caret) is returned, not including the length of any selected text within the line. When **LineLength** is called for a single-line edit control, this parameter is ignored.

Return Value

When **LineLength** is called for a multiple-line edit control, the return value is the length (in bytes) of the line specified by *nLine*. When **LineLength** is called for a single-line edit control, the return value is the length (in bytes) of the text in the edit control.

Remarks

Use the LineIndex member function to retrieve a character index for a given line number within this **CRichEditCtrl** object.

For more information, see EM_LINELENGTH in the *Platform SDK*.

Example

See the example for LineIndex.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::LineIndex

Other Resources

CRichEditCtrl Members

CRichEditCtrl::LineScroll

Scrolls the text of a multiple-line edit control.

```
void LineScroll(
   int nLines,
   int nChars = 0
);
```

Parameters

nLines

Specifies the number of lines to scroll vertically.

nChars

Specifies the number of character positions to scroll horizontally. This value is ignored if the rich edit control has either the **ES_RIGHT** or **ES_CENTER** style. Edit styles are specified in Create.

Remarks

The edit control does not scroll vertically past the last line of text in the edit control. If the current line plus the number of lines specified by *nLines* exceeds the total number of lines in the edit control, the value is adjusted so that the last line of the edit control is scrolled to the top of the edit-control window.

LineScroll can be used to scroll horizontally past the last character of any line.

For more information, see EM_LINESCROLL in the *Platform SDK*.

Example

See the example for GetFirstVisibleLine.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::LineIndex

Other Resources

CRichEditCtrl Members

CRichEditCtrl::Paste

Inserts the data from the Clipboard into the **CRichEditCtrl** at the insertion point, the location of the caret.

```
void Paste( );
```

Remarks

Data is inserted only if the Clipboard contains data in a recognized format.

For more information, see WM_PASTE in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;

// Replace all of the text with the text in the clipboard.
pmyRichEditCtrl->SetSel(0, -1);
pmyRichEditCtrl->Paste();
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::Copy
CRichEditCtrl::Cut
CRichEditCtrl::PasteSpecial

Other Resources

CRichEditCtrl Members

CRichEditCtrl::PasteSpecial

Pastes data in a specific Clipboard format into this **CRichEditCtrl** object.

```
void PasteSpecial(
   UINT nClipFormat,
   DWORD dvAspect = 0,
   HMETAFILE hMF = 0
);
```

Parameters

nClipFormat

Clipboard format to paste into this **CRichEditCtrl** object.

dvAspect

Device aspect for the data to be retrieved from the Clipboard.

hMF

Handle to the metafile containing the iconic view of the object to be pasted.

Remarks

The new material is inserted at the insertion point, the location of the caret.

For more information, see EM_PASTESPECIAL in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;

// Paste the data from the clipboard as text.
pmyRichEditCtrl->PasteSpecial(CF_TEXT);
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::Paste
CRichEditCtrl::Copy
CRichEditCtrl::Cut

Other Resources

CRichEditCtrl Members

CRichEditCtrl::PosFromChar

Retrieves the client area coordinates of a specified character in an edit control.

```
CPoint PosFromChar(
   UINT nChar
) const;
```

Parameters

nChar

The zero-based index of the character.

Return Value

The position of the character, (x, y). For a single-line edit control, the y-coordinate is always zero.

Remarks

For more information, see EM_POSFROMCHAR in the *Platform SDK*.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::GetCharPos

Other Resources

CRichEditCtrl Members

CRichEditCtrl::Redo

Redoes the next action in the control's redo queue.

```
BOOL Redo( );
```

Return Value

Nonzero if successful; otherwise, 0.

Remarks

For more information, see EM_REDO in the *Platform SDK*.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::Undo
CRichEditCtrl::CanRedo
CRichEditCtrl::GetRedoName

Other Resources

CRichEditCtrl Members

CRichEditCtrl::ReplaceSel

Replaces the current selection in this **CRichEditCtrl** object with the specified text.

```
void ReplaceSel(
   LPCTSTR lpszNewText,
   BOOL bCanUndo = FALSE
);
```

Parameters

lpszNewText

Pointer to a null-terminated string containing the replacement text.

bCanUndo

To specify that this function can be undone, set the value of this parameter to **TRUE**. The default value is **FALSE**.

Remarks

To replace all the text in this **CRichEditCtrl** object, use CWnd::SetWindowText.

If there is no current selection, the replacement text is inserted at the insertion point, that is, the current caret location.

This function will format the inserted text with the existing character formatting. When replacing the entire range of text (by calling **SetSel**(0,-1) before calling **ReplaceSel**), there is an end of paragraph character that retains the previous paragraph's formatting, which in inherited by the newly inserted text.

For more information, see EM_REPLACESEL in the *Platform SDK*.

Example

See the example for LineIndex.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::CanUndo
CRichEditCtrl::Undo
CWnd::SetWindowText

Other Resources

CRichEditCtrl Members

CRichEditCtrl::RequestResize

Forces this **CRichEditCtrl** object to send **EN_REQUESTRESIZE** notification messages to its parent window.

```
void RequestResize( );
```

Remarks

This function is useful during CWnd::OnSize processing for a bottomless **CRichEditCtrl** object.

For more information, see the EM_REQUESTRESIZE message and the **Bottomless Rich Edit Controls** section of About Rich Edit Controls in the *Platform SDK*.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CWnd::OnSize
CRichEditCtrl::Create

Other Resources

CRichEditCtrl Members

CRichEditCtrl::SetAutoURLDetect

Sets the rich edit control to automatically detect a URL.

```
BOOL SetAutoURLDetect(
   BOOL bEnable = TRUE
);
```

Parameters

bEnable

Specifies whether the control is set to automatically detect a URL. If **TRUE**, it is enabled. If **FALSE**, it is disabled.

Return Value

Zero if successful, otherwise nonzero. For example, the message may fail due to insufficient memory.

Remarks

If enabled, the rich edit control will scan the text to determine if it matches a standard URL format. For a list of these URL formats, see EM_AUTOURLDETECT in the *Platform SDK*.

Note

Do not set **SetAutoURLDetect** to **TRUE** if your edit control uses the **CFE_LINK** effect for text other than URLs. **SetAutoURLDetect** enables this effect for URLs and disables it for all other text. See EN_LINK for more information about the **CFE_LINK** effect.

See Also

Reference

CRichEditCtrl Class

Hierarchy Chart

Other Resources

CRichEditCtrl Members

CRichEditCtrl::SetBackgroundColor

Sets the background color for this **CRichEditCtrl** object.

```
COLORREF SetBackgroundColor(
   BOOL bSysColor,
   COLORREF cr
);
```

Parameters

bSysColor

Indicates if the background color should be set to the system value. If this value is **TRUE**, *cr* is ignored.

cr

The requested background color. Used only if *bSysColor* is **FALSE**.

Return Value

The previous background color for this **CRichEditCtrl** object.

Remarks

The background color can be set to the system value or to a specified COLORREF value.

For more information, see EM_SETBKGNDCOLOR message and COLORREF structure in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;

// Use red as the background color.
pmyRichEditCtrl->SetBackgroundColor(FALSE, RGB(255,0, 0));
```

See Also

Reference

CRichEditCtrl Class

Hierarchy Chart

CDC::SetBkColor

Other Resources

CRichEditCtrl Members

CRichEditCtrl::SetDefaultCharFormat

Sets the character formatting attributes for new text in this **CRichEditCtrl** object.

```
BOOL SetDefaultCharFormat(
   CHARFORMAT& cf
);
BOOL SetDefaultCharFormat(
   CHARFORMAT2& cf
);
```

Parameters

cf

In the first version, a pointer to a CHARFORMAT structure containing the new default character formatting attributes.

In the second version, a pointer to a CHARFORMAT2 structure, which is a Rich Edit 2.0 extension to the **CHARFORMAT** structure, containing the default character formatting attributes.

Return Value

Nonzero if successful; otherwise, 0.

Remarks

Only the attributes specified by the **dwMask** member of *cf* are changed by this function.

For more information, see the EM_SETCHARFORMAT message and the **CHARFORMAT** and **CHARFORMAT2** structures in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;
CHARFORMAT cf;

// Modify the default character format so that all new
// text is striked out and not bold.
cf.dwMask = CFM_STRIKEOUT|CFM_BOLD;
cf.dwEffects = CFE_STRIKEOUT;
pmyRichEditCtrl->SetDefaultCharFormat(cf);

// Verify the settings are what is expected.
#ifdef _DEBUG
   pmyRichEditCtrl->GetDefaultCharFormat(cf);
   ASSERT((cf.dwMask&(CFM_STRIKEOUT|CFM_BOLD)) ==
      (CFM_STRIKEOUT|CFM_BOLD));
   ASSERT((cf.dwEffects&(CFE_STRIKEOUT|CFE_BOLD)) == CFE_STRIKEOUT);
#endif
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::GetDefaultCharFormat
CRichEditCtrl::SetSelectionCharFormat

Other Resources

CRichEditCtrl Members

CRichEditCtrl::SetEventMask

Sets the event mask for this **CRichEditCtrl** object.

```
DWORD SetEventMask(
   DWORD dwEventMask
);
```

Parameters

dwEventMask

The new event mask for this **CRichEditCtrl** object.

Return Value

The previous event mask.

Remarks

The event mask specifies which notification messages the **CRichEditCtrl** object sends to its parent window.

For more information, see EM_SETEVENTMASK in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;

// Set the event mask so that the parent gets notified when the text
// of the rich edit control changes.
pmyRichEditCtrl->SetEventMask(pmyRichEditCtrl->GetEventMask() |
    ENM_CHANGE);
```

See Also

Reference

CRichEditCtrl Class

Hierarchy Chart

CRichEditCtrl::GetEventMask

Other Resources

CRichEditCtrl Members

CRichEditCtrl::SetModify

Sets or clears the modified flag for an edit control.

```
void SetModify(
   BOOL bModified = TRUE
);
```

Parameters

bModified

A value of **TRUE** indicates that the text has been modified, and a value of **FALSE** indicates it is unmodified. By default, the modified flag is set.

Remarks

The modified flag indicates whether or not the text within the edit control has been modified. It is automatically set whenever the user changes the text. Its value can be retrieved with the GetModify member function.

For more information, see EM_SETMODIFY in the *Platform SDK*.

Example

See the example for GetModify.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::GetModify

Other Resources

CRichEditCtrl Members

CRichEditCtrl::SetOLECallback

Gives this **CRichEditCtrl** object an **IRichEditOleCallback** object to use to access OLE-related resources and information.

```
BOOL SetOLECallback(
   IRichEditOleCallback* pCallback
);
```

Parameters

pCallback

Pointer to an IRichEditOleCallback object that this **CRichEditCtrl** object will use to get OLE-related resources and information.

Return Value

Nonzero if successful; otherwise, 0.

Remarks

This **CRichEditCtrl** object will call IUnknown::AddRef to increment the usage count for the COM object specified by *pCallback*.

For more information, see EM_SETOLECALLBACK message and IRichEditOleCallback interface in the *Platform SDK*.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::GetIRichEditOle

Other Resources

CRichEditCtrl Members

CRichEditCtrl::SetOptions

Sets the options for this **CRichEditCtrl** object.

```
void SetOptions(
   WORD wOp,
   DWORD dwFlags
);
```

Parameters

wOp

Indicates the type of operation. One of the following values:

- **ECOOP_SET** Set the options to those specified by *dwFlags*.
- **ECOOP_OR** Combine the current options with those specified by *dwFlags*.
- **ECOOP_AND** Retain only those current options that are also specified by *dwFlags*.
- **ECOOP_XOR** Retain only those current options that are *not* specified by *dwFlags*.

dwFlags

Rich edit options. The flag values are listed in the Remarks section.

Remarks

The options can be a combination of the following values:

- **ECO_AUTOWORDSELECTION** Automatic word selection on double-click.
- **ECO_AUTOVSCROLL** Automatically scrolls text to the right by 10 characters when the user types a character at the end of the line. When the user presses the ENTER key, the control scrolls all text back to position zero.
- **ECO_AUTOHSCROLL** Automatically scrolls text up one page when the user presses the ENTER key on the last line.
- **ECO_NOHIDESEL** Negates the default behavior for an edit control. The default behavior hides the selection when the control loses the input focus and shows the selection when the control receives the input focus. If you specify **ECO_NOHIDESEL**, the selected text is inverted, even if the control does not have the focus.
- **ECO_READONLY** Prevents the user from typing or editing text in the edit control.
- **ECO_WANTRETURN** Specifies that a carriage return be inserted when the user presses the ENTER key while entering text into a multiple-line rich edit control in a dialog box. If you do not specify this style, pressing the ENTER key sends a command to the rich edit control's parent window, which mimics clicking the parent window's default button (for example, the OK button in a dialog box). This style has no effect on a single-line edit control.
- **ECO_SAVESEL** Preserves the selection when the control loses the focus. By default, the entire contents of the control are selected when it regains the focus.
- **ECO_VERTICAL** Draws text and objects in a vertical direction. Available for Asian languages only.

For more information, see EM_SETOPTIONS in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;

// Add auto horizontal and vertical scrolling.
pmyRichEditCtrl->SetOptions(ECOOP_OR, ECO_AUTOVSCROLL |
   ECO_AUTOHSCROLL);
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::HideSelection
CRichEditCtrl::SetReadOnly

Other Resources

CRichEditCtrl Members

CRichEditCtrl::SetParaFormat

Sets the paragraph formatting attributes for the current selection in this **CRichEditCtrl** object.

```
BOOL SetParaFormat(
   PARAFORMAT& pf
);
BOOL SetParaFormat(
   PARAFORMAT2& pf
);
```

Parameters

pf

In the first version, a pointer to a PARAFORMAT structure containing the new default paragraph formatting attributes.

In the second version, a pointer to a PARAFORMAT2 structure, which is a Rich Edit 2.0 extension to the **PARAFORMAT** structure, holding the default character formatting attributes.

Return Value

Nonzero if successful; otherwise, 0.

Remarks

Only the attributes specified by the **dwMask** member of *pf* are changed by this function.

For more information, see the EM_SETPARAFORMAT message and the **PARAFORMAT** and **PARAFORMAT2** structures in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;
PARAFORMAT pf;

// Modify the paragraph format so that the text is centered.
pf.cbSize = sizeof(PARAFORMAT);
pf.dwMask = PFM_ALIGNMENT;
pf.wAlignment = PFA_CENTER;
pmyRichEditCtrl->SetParaFormat(pf);

// Verify the settings are what is expected.
#ifdef _DEBUG
   pmyRichEditCtrl->GetParaFormat(pf);
   ASSERT(pf.dwMask&PFM_ALIGNMENT);
   ASSERT(pf.wAlignment == PFA_CENTER);
#endif
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::GetParaFormat
CRichEditCtrl::SetSelectionCharFormat

Other Resources

CRichEditCtrl Members

CRichEditCtrl::SetPunctuation

Sets the punctuation in a rich edit control.

```
BOOL SetPunctuation(
   UINT fType,
   PUNCTUATION* lpPunc
);
```

Parameters

fType

The punctuation flag. For a list of possible values, see the *fType* parameter for EM_SETPUNCTUATION in the *Platform SDK*.

lpPunc

A pointer to a PUNCTUATION structure, as described in the *Platform SDK*.

Return Value

Nonzero if successful, otherwise 0.

Remarks

This member function is available for only Asian-language versions of the operating system.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::GetPunctuation

Other Resources

CRichEditCtrl Members

CRichEditCtrl::SetReadOnly

Changes the **ECO_READONLY** option for this **CRichEditCtrl** object.

```
BOOL SetReadOnly(
   BOOL bReadOnly = TRUE
);
```

Parameters

bReadOnly

Indicates if this **CRichEditCtrl** object should be read only.

Return Value

Nonzero if successful; otherwise, 0.

Remarks

For a brief description of this option, see SetOptions. You can use this function to set all the options for this **CRichEditCtrl** object.

For more information, see EM_SETREADONLY in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;

// Set the rich edit control to be read-only.
pmyRichEditCtrl->SetReadOnly(TRUE);
ASSERT(pmyRichEditCtrl->GetStyle() & ES_READONLY);
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::Create
CRichEditCtrl::SetOptions

Other Resources

CRichEditCtrl Members

CRichEditCtrl::SetRect

Sets the formatting rectangle for this **CRichEditCtrl** object.

```
void SetRect(
   LPCRECT lpRect
);
```

Parameters

lpRect

CRect or pointer to a RECT that indicates the new bounds for the formatting rectangle.

Remarks

The formatting rectangle is the limiting rectangle for the text. The limiting rectangle is independent of the size of the rich edit control window. When this **CRichEditCtrl** object is first created, the formatting rectangle is the same size as the client area of the window. Use **SetRect** to make the formatting rectangle larger or smaller than the rich edit window.

For more information, see EM_SETRECT in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;

CRect r;

pmyRichEditCtrl->GetRect(&r);

// Reduce the formatting rect of the rich edit control by
// 10 pixels on each side.
if ((r.Width() > 20) && (r.Height() > 20))
{
   r.DeflateRect(0, 20);
   pmyRichEditCtrl->SetRect(&r);
}
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::GetRect

Other Resources

CRichEditCtrl Members

CRichEditCtrl::SetSel

Sets the selection within this **CRichEditCtrl** object.

```
void SetSel(
   long nStartChar,
   long nEndChar
);
void SetSel(
   CHARRANGE& cr
);
```

Parameters

nStartChar

Zero-based index of the first character for the selection.

nEndChar

Zero-based index of the last character for the selection.

cr

CHARRANGE structure which holds the bounds of the current selection.

Remarks

The two forms of this function provide alternate ways to set the bounds for the selection. Brief descriptions of these forms follow:

- **SetSel(** *cr* **)** This form uses the **CHARRANGE** structure with its **cpMin** and **cpMax** members to set the bounds.
- **SetSel(** *nStartChar* **,** *nEndChar* **)** This form use the parameters *nStartChar* and *nEndChar* to set the bounds.

The caret is placed at the end of the selection indicated by the greater of the start (**cpMin** or *nStartChar*) and end (**cpMax** or *nEndChar*) indices. This function scrolls the contents of the **CRichEditCtrl** so that the caret is visible.

To select all the text in this **CRichEditCtrl** object, call **SetSel** with a start index of 0 and an end index of – 1.

For more information, see EM_EXSETSEL message and CHARRANGE structure in the *Platform SDK*.

Example

See the example for GetSel.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::GetSel
CRichEditCtrl::GetSelectionType

Other Resources

CRichEditCtrl Members

CRichEditCtrl::SetSelectionCharFormat

Sets the character formatting attributes for the text in the current selection in this **CRichEditCtrl** object.

```
BOOL SetSelectionCharFormat(
   CHARFORMAT& cf
);
BOOL SetSelectionCharFormat(
   CHARFORMAT2& cf
);
```

Parameters

cf

In the first version, a pointer to a CHARFORMAT structure containing the new character formatting attributes for the current selection.

In the second version, a pointer to a CHARFORMAT2 structure, which is a Rich Edit 2.0 extension to the **CHARFORMAT** structure, containing the new character formatting attributes for the current selection.

Return Value

Nonzero if successful; otherwise, 0.

Remarks

Only the attributes specified by the **dwMask** member of *cf* are changed by this function.

For more information, see the EM_SETCHARFORMAT and the **CHARFORMAT** and **CHARFORMAT2** structures in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;
CHARFORMAT cf;

// Modify the selection format so that the selected text is
// displayed in bold and not striked out.
cf.dwMask = CFM_STRIKEOUT|CFM_BOLD;
cf.dwEffects = CFE_BOLD;
pmyRichEditCtrl->SetSelectionCharFormat(cf);

// Verify the settings are what is expected.
#ifdef _DEBUG
   pmyRichEditCtrl->GetSelectionCharFormat(cf);
   ASSERT((cf.dwMask&(CFM_STRIKEOUT|CFM_BOLD)) ==
      (CFM_STRIKEOUT|CFM_BOLD));
   ASSERT((cf.dwEffects&(CFE_STRIKEOUT|CFE_BOLD)) == CFE_BOLD);
#endif
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::GetSelectionCharFormat
CRichEditCtrl::SetDefaultCharFormat

Other Resources

CRichEditCtrl Members

CRichEditCtrl::SetTargetDevice

Sets the target device and line width used for WYSIWYG (what you see is what you get) formatting in this **CRichEditCtrl** object.

```
BOOL SetTargetDevice(
   HDC hDC,
   long lLineWidth
);
BOOL SetTargetDevice(
   CDC& dc,
   long lLineWidth
);
```

Parameters

hDC

Handle to the device context for the new target device.

lLineWidth

Line width to use for formatting.

dc

CDC for the new target device.

Return Value

Nonzero if successful; otherwise, 0.

Remarks

If this function is successful, the rich edit control owns the device context passed as a parameter. In that case, the calling function should not destroy the device context.

For more information, see EM_SETTARGETDEVICE in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;
// A pointer to a printer DC.
extern CDC* pMyPrinterDC;

// Get line width information from the printer.
long lLineWidth = ::MulDiv(pMyPrinterDC->GetDeviceCaps(PHYSICALWIDTH),
    1440, pMyPrinterDC->GetDeviceCaps(LOGPIXELSX));

// Set the printer as the target device.
pmyRichEditCtrl->SetTargetDevice(*pMyPrinterDC, lLineWidth);
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::FormatRange
CRichEditCtrl::DisplayBand

Other Resources

CRichEditCtrl Members

CRichEditCtrl::SetTextMode

Sets the text mode or undo and redo level for a rich edit control.

```
BOOL SetTextMode(
   UINT fMode
);
```

Parameters

fMode

Specifies the new settings for the control's text mode and undo level parameters. For a list of the possible values, see the mode parameter for EM_SETTEXTMODE in the *Platform SDK*.

Return Value

Zero if successful, otherwise nonzero.

Remarks

For a description of the text modes, see **EM_SETTEXTMODE** in the *Platform SDK*.

This member function fails if the control contains text. To make sure the control is empty, send a WM_SETTEXT message with an empty string.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::GetTextMode

Other Resources

CRichEditCtrl Members

CRichEditCtrl::SetUndoLimit

Sets the maximum number of actions that can stored in the undo queue.

```
UINT SetUndoLimit(
   UINT nLimit
);
```

Parameters

nLimit

Specifies the maximum number of actions that can be stored in the undo queue. Set to zero to disable Undo.

Return Value

The new maximum number of undo actions for the rich edit control.

Remarks

By default, the maximum number of actions in the undo queue is 100. If you increase this number, there must be enough available memory to accommodate the new number. For better performance, set the limit to the smallest possible value.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::EmptyUndoBuffer
CRichEditCtrl::Undo

Other Resources

CRichEditCtrl Members

CRichEditCtrl::SetWordCharFormat

Sets the character formatting attributes for the currently selected word in this **CRichEditCtrl** object.

```
BOOL SetWordCharFormat(
   CHARFORMAT& cf
);
BOOL SetWordCharFormat(
   CHARFORMAT2& cf
);
```

Parameters

cf

In the first version, a pointer to a CHARFORMAT structure containing the new character formatting attributes for the currently selected word.

In the second version, a pointer to a CHARFORMAT2 structure, which is a Rich Edit 2.0 extension to the **CHARFORMAT** structure, containing the new character formatting attributes for the currently selected word.

Return Value

Nonzero if successful; otherwise, 0.

Remarks

Only the attributes specified by the **dwMask** member of *cf* are changed by this function.

For more information, see the EM_SETCHARFORMAT message and the **CHARFORMAT** and **CHARFORMAT2** structures in the *Platform SDK*.

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;
CHARFORMAT cf;

// Modify the word format so that the selected word is
// displayed in bold and not striked out.
cf.dwMask = CFM_STRIKEOUT|CFM_BOLD;
cf.dwEffects = CFE_BOLD;
pmyRichEditCtrl->SetWordCharFormat(cf);
```

See Also

Reference

CRichEditCtrl Class

Hierarchy Chart

CRichEditCtrl::SetSelectionCharFormat

Other Resources

CRichEditCtrl Members

CRichEditCtrl::SetWordWrapMode

Sets the word-wrapping and word-breaking options for the rich edit control.

```
UINT SetWordWrapMode(
   UINT uFlags
) const;
```

Parameters

uFlags

The options to set for word wrapping and word breaking. For a list of possible options, see EM_SETWORDWRAPMODE in the *Platform SDK*.

Return Value

The current word-wrapping and word-breaking options.

Remarks

This message is available only in Asian-language versions of the operating system.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::GetWordWrapMode

Other Resources

CRichEditCtrl Members

CRichEditCtrl::StopGroupTyping

Stops the control from collecting additional typing actions into the current undo action.

```
void StopGroupTyping( );
```

Remarks

The control stores the next typing action, if any, into a new action in the undo queue.

For more information, see EM_STOPGROUPTYPING in the *Platform SDK*.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::Undo

Other Resources

CRichEditCtrl Members

CRichEditCtrl::StreamIn

Replaces text in this **CRichEditCtrl** object with text from the specified input stream.

```
long StreamIn(
   int nFormat,
   EDITSTREAM& es
);
```

Parameters

nFormat

Flags specifying the input data formats. See the Remarks section for more information.

es

EDITSTREAM structure specifying the input stream. See the Remarks section for more information.

Return Value

Number of characters read from the input stream.

Remarks

The value of *nFormat* must be one of the following:

- **SF_TEXT** Indicates reading text only.
- **SF_RTF** Indicates reading text and formatting.

Either of these values can be combined with **SFF_SELECTION**. If **SFF_SELECTION** is specified, **StreamIn** replaces the current selection with the contents of the input stream. If it is not specified, **StreamIn** replaces the entire contents of this **CRichEditCtrl** object.

In the **EDITSTREAM** parameter *es*, you specify a callback function that fills a buffer with text. This callback function is called repeatedly, until the input stream is exhausted.

For more information, see EM_STREAMIN message and EDITSTREAM structure in the *Platform SDK*.

Example

```
// My callback procedure that writes the rich edit control contents
// to a file.
static DWORD CALLBACK
MyStreamInCallback(DWORD dwCookie, LPBYTE pbBuff, LONG cb, LONG *pcb)
{
   CFile* pFile = (CFile*) dwCookie;

   *pcb = pFile->Read(pbBuff, cb);

   return 0;
}

// The example code.
   // The pointer to my rich edit control.
   extern CRichEditCtrl* pmyRichEditCtrl;
   // The file from which to load the contents of the rich edit control.
   CFile cFile(TEXT("myfile.rtf"), CFile::modeRead);
   EDITSTREAM es;

   es.dwCookie = (DWORD) &cFile;
   es.pfnCallback = MyStreamInCallback;
   pmyRichEditCtrl->StreamIn(SF_RTF, es);
```

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::StreamOut
Other Resources
CRichEditCtrl Members

CRichEditCtrl::StreamOut

Writes out the contents of this **CRichEditCtrl** object to the specified output stream.

```
long StreamOut(
   int nFormat,
   EDITSTREAM& es
);
```

Parameters

nFormat

Flags specifying the output data formats. See the Remarks section for more information.

es

EDITSTREAM structure specifying the output stream. See the Remarks section for more information.

Return Value

Number of characters written to the output stream.

Remarks

The value of *nFormat* must be one of the following:

- **SF_TEXT** Indicates writing text only.
- **SF_RTF** Indicates writing text and formatting.
- **SF_RTFNOOBJS** Indicates writing text and formatting, replacing OLE items with spaces.
- **SF_TEXTIZED** Indicates writing text and formatting, with textual representations of OLE items.

Any of these values can be combined with **SFF_SELECTION**. If **SFF_SELECTION** is specified, **StreamOut** writes out the current selection into the output stream. If it is not specified, **StreamOut** writes out the entire contents of this **CRichEditCtrl** object.

In the **EDITSTREAM** parameter *es*, you specify a callback function which fills a buffer with text. This callback function is called repeatedly, until the output stream is exhausted.

For more information, see EM_STREAMOUT message and EDITSTREAM structure in the *Platform SDK*.

Example

```
// My callback procedure that reads the rich edit control contents
// from a file.
static DWORD CALLBACK
MyStreamOutCallback(DWORD dwCookie, LPBYTE pbBuff, LONG cb, LONG *pcb)
{
   CFile* pFile = (CFile*) dwCookie;

   pFile->Write(pbBuff, cb);
   *pcb = cb;

   return 0;
}

// The example code.
   // The pointer to my rich edit control.
   extern CRichEditCtrl* pmyRichEditCtrl;
   // The file to store the contents of the rich edit control.
   CFile cFile(TEXT("myfile.rtf"), CFile::modeCreate|CFile::modeWrite);
   EDITSTREAM es;

   es.dwCookie = (DWORD) &cFile;
   es.pfnCallback = MyStreamOutCallback;
   pmyRichEditCtrl->StreamOut(SF_RTF, es);
```

See Also

Reference

CRichEditCtrl Class

Hierarchy Chart

CRichEditCtrl::StreamIn

Other Resources

CRichEditCtrl Members

CRichEditCtrl::Undo

Undoes the last operation in the rich edit control.

```
BOOL Undo( );
```

Return Value

Nonzero if the undo operation is successful; otherwise, 0.

Remarks

An undo operation can also be undone. For example, you can restore deleted text with the first call to **Undo**. As long as there is no intervening edit operation, you can remove the text again with a second call to **Undo**.

For more information, see EM_UNDO in the *Platform SDK*.

Example

See the example for CanUndo.

See Also

Reference

CRichEditCtrl Class
Hierarchy Chart
CRichEditCtrl::CanUndo
CRichEditCtrl::EmptyUndoBuffer

Other Resources

CRichEditCtrl Members

CRichEditDoc Class

With CRichEditView and CRichEditCntrItem, provides the functionality of the rich edit control within the context of MFC's document view architecture.

```
class CRichEditDoc : public COleServerDoc
```

Remarks

A "rich edit control" is a window in which the user can enter and edit text. The text can be assigned character and paragraph formatting, and can include embedded OLE objects. Rich edit controls provide a programming interface for formatting text. However, an application must implement any user interface components necessary to make formatting operations available to the user.

CRichEditView maintains the text and formatting characteristic of text. **CRichEditDoc** maintains the list of client items which are in the view. **CRichEditCntrItem** provides container-side access to the OLE client items.

This Windows Common control (and therefore the CRichEditCtrl and related classes) is available only to programs running under Windows 95/98 and Windows NT versions 3.51 and later.

For an example of using a rich edit document in an MFC application, see the WORDPAD sample application.

Requirements

Header: afxrich.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

WORDPAD Sample: The Windows Application

Reference

COleServerDoc Class
Hierarchy Chart
CRichEditView Class
CRichEditCntrItem Class
COleDocument Class
CRichEditCtrl Class

Other Resources

CRichEditDoc Members

CRichEditDoc Members

Base Class Members

CObject Members

CCmdTarget Members

CDocument Members

COleDocument Members

COleLinkingDoc Members

COleServerDoc Members

Attributes

GetStreamFormat	Indicates whether stream input and output should include formatting information.
GetView	Retrieves the asssociated CRichEditView object.

Data Members

m_bRTF	Indicates whether stream I/O should include formatting.

Overridables

CreateClientItem	Called to perform cleanup of the document.

See Also

Reference

CRichEditDoc Class

Hierarchy Chart

CRichEditDoc Member Functions

For information about the member functions in **CRichEditDoc**, see CRichEditDoc Members.

CRichEditDoc::CreateClientItem

Call this function to create a **CRichEditCntrItem** object and add it to this document.

```
virtual CRichEditCntrItem* CreateClientItem(
   REOBJECT* preo = NULL
) const = 0;
```

Parameters

preo

Pointer to an REOBJECT structure which describes an OLE item. The new **CRichEditCntrItem** object is constructed around this OLE item. If *preo* is **NULL**, the new client item is empty.

Return Value

Pointer to a new CRichEditCntrItem object which has been added to this document.

Remarks

This function does not perform any OLE initialization.

For more information, see the REOBJECT structure in the *Platform SDK*.

See Also

Reference

CRichEditDoc Class
Hierarchy Chart
CRichEditCntrItem::CRichEditCntrItem
COleDocument::AddItem

Other Resources

CRichEditDoc Members

CRichEditDoc::GetStreamFormat

Call this function to determine the text format for streaming the contents of the rich edit.

```
int GetStreamFormat( ) const;
```

Return Value

One of the following flags:

- **SF_TEXT** Indicates that the rich edit control does not maintain formatting information.
- **SF_RTF** Indicates that the rich edit control does maintain formatting information.

Remarks

The return value is based on the m_bRTF data member. This function returns **SF_RTF** if **m_bRTF** is **TRUE**; otherwise, **SF_TEXT**.

See Also

Reference

CRichEditDoc Class
Hierarchy Chart
CRichEditDoc::m_bRTF
CRichEditCtrl::StreamIn
CRichEditCtrl::StreamOut

Other Resources

CRichEditDoc Members

CRichEditDoc::GetView

Call this function to access the CRichEditView object associated with this **CRichEditDoc** object.

```
virtual CRichEditView* GetView( ) const;
```

Return Value

Pointer to the **CRichEditView** object associated with the document.

Remarks

The text and formatting information are contained within the **CRichEditView** object. The **CRichEditDoc** object maintains the OLE items for serialization. There should be only one **CRichEditView** for each **CRichEditDoc**.

See Also

Reference

CRichEditDoc Class
Hierarchy Chart
CRichEditView Class
CDocument::GetNextView

Other Resources

CRichEditDoc Members

CRichEditDoc Data Members

For information about the data members in **CRichEditDoc**, see CRichEditDoc Members.

CRichEditDoc::m_bRTF

When **TRUE**, indicates that CRichEditCtrl::StreamIn and CRichEditCtrl::StreamOut should store paragraph and character-formatting characteristics.

```
BOOL m_bRTF;
```

See Also

Reference

CRichEditView Class

Hierarchy Chart

CRichEditDoc::GetStreamFormat

Other Resources

CRichEditView Members

CRichEditView Class

With CRichEditDoc and CRichEditCntrItem, provides the functionality of the rich edit control within the context of MFC's document view architecture.

```
class CRichEditView : public CCtrlView
```

Remarks

A "rich edit control" is a window in which the user can enter and edit text. The text can be assigned character and paragraph formatting, and can include embedded OLE objects. Rich edit controls provide a programming interface for formatting text. However, an application must implement any user interface components necessary to make formatting operations available to the user.

CRichEditView maintains the text and formatting characteristic of text. **CRichEditDoc** maintains the list of OLE client items which are in the view. **CRichEditCntrItem** provides container-side access to the OLE client item.

This Windows Common control (and therefore the CRichEditCtrl and related classes) is available only to programs running under Windows 95/98 and Windows NT versions 3.51 and later.

For an example of using a rich edit view in an MFC application, see the WORDPAD sample application.

Requirements

Header: afxrich.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

WORDPAD Sample: The Windows Application

Reference

CCtrlView Class
Hierarchy Chart
CRichEditDoc Class
CRichEditCntrItem Class

Other Resources

CRichEditView Members

CRichEditView Members

Base Class Members

COBJECT Members

CCmdTarget Members

CWnd Members

CView Members

CCtrlView Members

Constructor

CRichEditView	Constructs a **CRichEditView** object.

Attributes

GetCharFormatSelection	Retrieves the character formatting attributes for the current selection.
GetDocument	Retrieves a pointer to the related CRichEditDoc.
GetInPlaceActiveItem	Retrieves the OLE item that is currently in-place active in the rich edit view.
GetMargins	Retrieves the margins for this rich edit view.
GetPageRect	Retrieves the page rectangle for this rich edit view.
GetPaperSize	Retrieves the paper size for this rich edit view.
GetParaFormatSelection	Retrieves the paragraph formatting attributes for the current selection.
GetPrintRect	Retrieves the print rectangle for this rich edit view.
GetPrintWidth	Retrieves the print width for this rich edit view.
GetRichEditCtrl	Retrieves the rich edit control.
GetSelectedItem	Retrieves the selected item from the rich edit view.
GetTextLength	Retrieves the length of the text in the rich edit view.
GetTextLengthEx	Retrieves the number of characters or bytes in the rich edit view. Expanded flag list for method of det ermining the length.
SetCharFormat	Sets the character formatting attributes for the current selection.
SetMargins	Sets the margins for this rich edit view.
SetPaperSize	Sets the paper size for this rich edit view.
SetParaFormat	Sets the paragraph formatting attributes for the current selection.

Data Members

m_nBulletIndent	Indicates the amount of indent for bullet lists.

m_nWordWrap	Indicates the word wrap constraints.

Operations

AdjustDialogPosition	Moves a dialog box so that it doesn't obscure the current selection.
CanPaste	Tells whether the Clipboard contains data that can be pasted into the rich edit view.
DoPaste	Pastes an OLE item into this rich edit view.
FindText	Finds the specified text, invoking the wait cursor.
FindTextSimple	Finds the specified text.
InsertFileAsObject	Inserts a file as an OLE item.
InsertItem	Inserts a new item as an OLE item.
IsRichEditFormat	Tells whether the Clipboard contains data in a rich edit or text format.
OnCharEffect	Toggles the character formatting for the current selection.
OnParaAlign	Changes the alignment of paragraphs.
OnUpdateCharEffect	Updates the Command UI for character public member functions.
OnUpdateParaAlign	Updates the Command UI for paragraph public member functions.
PrintInsideRect	Formats the specified text within the given rectangle.
PrintPage	Formats the specified text within the given page.
TextNotFound	Resets the internal search state of the control.

Overridables

GetClipboardData	Retrieves a Clipboard object for a range in this rich edit view.
GetContextMenu	Retrieves a context menu to use on a right mouse-button down.
IsSelected	Indicates if the given OLE item is selected or not.
OnFindNext	Finds the next occurrence of a substring.
OnInitialUpdate	Refreshes a view when it is first attached to a document.
OnPasteNativeObject	Retrieves native data from an OLE item.
OnPrinterChanged	Sets the print characteristics to the given device.
OnReplaceAll	Replaces all occurrences of a given string with a new string.
OnReplaceSel	Replaces the current selection.
OnTextNotFound	Handles user notification that the requested text was not found.

QueryAcceptData	Queries to see about the data on the **IDataObject**.
WrapChanged	Adjusts the target output device for this rich edit view, based on the value of **m_nWordWrap**.

See Also

Reference

CRichEditView Class

Hierarchy Chart

CRichEditView Member Functions

For information about the member functions in **CRichEditView**, see CRichEditView Members.

CRichEditView::AdjustDialogPosition

Call this function to move the given dialog box so that it does not obscure the current selection.

```
void AdjustDialogPosition(
   CDialog* pDlg
);
```

Parameters

pDlg

Pointer to a **CDialog** object.

See Also

Reference

CRichEditView Class

Hierarchy Chart

CRichEditCtrl::GetSel

Other Resources

CRichEditView Members

CRichEditView::CanPaste

Call this function to determine if the Clipboard contains information that can be pasted into this rich edit view.

```
BOOL CanPaste( ) const;
```

Return Value

Nonzero if the Clipboard contains data in a format which this rich edit view can accept; otherwise, 0.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditCtrl::Paste
CRichEditView::DoPaste
CRichEditView::IsRichEditFormat

Other Resources

CRichEditView Members

CRichEditView::CRichEditView

Call this function to create a **CRichEditView** object.

```
CRichEditView( );
```

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditDoc Class
CRichEditCtrl Class

Other Resources

CRichEditView Members

CRichEditView::DoPaste

Call this function to paste the OLE item in *dataobj* into this rich edit document/view.

```
void DoPaste(
   COleDataObject& dataobj,
   CLIPFORMAT cf,
   HMETAFILEPICT hMetaPict
);
```

Parameters

dataobj

The COleDataObject containing the data to paste.

cf

The desired Clipboard format.

hMetaPict

The metafile that represents the item to be pasted.

Remarks

The framework calls this function as part of the default implementation of QueryAcceptData.

This function determines the type of paste based on the results of the handler for Paste Special. If *cf* is 0, the new item uses the current iconic representation. If *cf* is nonzero and *hMetaPict* is not **NULL**, the new item uses *hMetaPict* for its representation.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditCtrl::Paste
CRichEditView::IsRichEditFormat
CRichEditView::InsertItem

Other Resources

CRichEditView Members

CRichEditView::FindText

Call this function to find the specified text and set it to be the current selection.

```
BOOL FindText(
   LPCTSTR lpszFind,
   BOOL bCase = TRUE,
   BOOL bWord = TRUE,
   BOOL bNext = TRUE
);
```

Parameters

lpszFind

Contains the string to search for.

bCase

Indicates if the search is case sensitive.

bWord

Indicates if the search should match whole words only, not parts of words.

bNext

Indicates the direction of the search. If **TRUE**, the search direction is toward the end of the buffer. If **FALSE**, the search direction is toward the beginning of the buffer.

Return Value

Nonzero if the *lpszFind* text is found; otherwise 0.

Remarks

This function displays the wait cursor during the find operation.

Example

```
void CMyRichEditView::OnReplaceAll( LPCTSTR lpszFind,
            LPCTSTR lpszReplace, BOOL bCase, BOOL bWord )
{
   CWaitCursor wait;
   // No selection or different than what we are looking for
   if (!FindText(lpszFind, bCase, bWord))
   {
      // Call the TextNotFound method to reset the internal search
      // state of the control
      TextNotFound(lpszFind);
      return;
   }

   GetRichEditCtrl().HideSelection(TRUE, FALSE);
   m_nNumReplaced = 0;
   do
   {
      GetRichEditCtrl().ReplaceSel(lpszReplace);
      // Record the number of replacements
      m_nNumReplaced++;

   } while (FindTextSimple(lpszFind));
   GetRichEditCtrl().HideSelection(FALSE, FALSE);
}
```

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditCtrl::FindText
CRichEditCtrl::SetSel
CRichEditView::FindTextSimple
CWaitCursor Class

Other Resources

CRichEditView Members

CRichEditView::FindTextSimple

Call this function to find the specified text and set it to be the current selection.

```
BOOL FindTextSimple(
   LPCTSTR lpszFind,
   BOOL bCase = TRUE,
   BOOL bWord = TRUE,
   BOOL bNext = TRUE
);
```

Parameters

lpszFind

Contains the string to search for.

bCase

Indicates if the search is case sensitive.

bWord

Indicates if the search should match whole words only, not parts of words.

bNext

Indicates the direction of the search. If **TRUE**, the search direction is toward the end of the buffer. If **FALSE**, the search direction is toward the beginning of the buffer.

Return Value

Nonzero if the *lpszFind* text is found; otherwise 0.

Example

See the example for CRichEditView::FindText.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditCtrl::FindText
CRichEditCtrl::SetSel
CRichEditView::FindText

Other Resources

CRichEditView Members

CRichEditView::GetCharFormatSelection

Call this function to get the character formatting attributes of the current selection.

```
CHARFORMAT2& GetCharFormatSelection( );
```

Return Value

A CHARFORMAT2 structure which contains the character formatting attributes of the current selection.

Remarks

For more information, see the EM_GETCHARFORMAT message and the CHARFORMAT2 structure in the *Platform SDK*.

Example

```
void CMyRichEditView::OnCharUnderline ()
{
    CHARFORMAT2 cf;
    cf = GetCharFormatSelection ();

    if (!(cf.dwMask & CFM_UNDERLINE) || !(cf.dwEffects & CFE_UNDERLINE))
        cf.dwEffects = CFE_UNDERLINE;
    else
        cf.dwEffects = 0;

    cf.dwMask = CFM_UNDERLINE;
    SetCharFormat (cf);
}
```

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::SetCharFormat
CRichEditView::GetParaFormatSelection
CRichEditCtrl::GetSelectionCharFormat

Other Resources

CRichEditView Members

CRichEditView::GetClipboardData

The framework calls this function as part of the processing of IRichEditOleCallback::GetClipboardData.

```
virtual HRESULT GetClipboardData(
   CHARRANGE* lpchrg,
   DWORD dwReco,
   LPDATAOBJECT lpRichDataObj,
   LPDATAOBJECT* lplpdataobj
);
```

Parameters

lpchrg

Pointer to the CHARRANGE structure specifying the range of characters (and OLE items) to copy to the data object specified by *lplpdataobj*.

dwReco

Clipboard operation flag. Can be one of these values.

- **RECO_COPY** Copy to the Clipboard.
- **RECO_CUT** Cut to the Clipboard.
- **RECO_DRAG** Drag operation (drag and drop).
- **RECO_DROP** Drop operation (drag and drop).
- **RECO_PASTE** Paste from the Clipboard.

lpRichDataObj

Pointer to an IDataObject object containing the Clipboard data from the rich edit control (IRichEditOle::GetClipboardData).

lplpdataobj

Pointer to the pointer variable that receives the address of the **IDataObject** object representing the range specified in the *lpchrg* parameter. The value of *lplpdataobj* is ignored if an error is returned.

Return Value

An **HRESULT** value reporting the success of the operation. For more information on **HRESULT**, see Structure of COM Error Codes in the *Platform SDK*.

Remarks

If the return value indicates success, **IRichEditOleCallback::GetClipboardData** returns the **IDataObject** accessed by *lplpdataobj*; otherwise, it returns the one accessed by *lpRichDataObj*. Override this function to supply your own Clipboard data. The default implementation of this function returns **E_NOTIMPL**.

This is an advanced overridable.

For more information, see IRichEditOle::GetClipboardData, IRichEditOleCallback::GetClipboardData, and CHARRANGE in the *Platform SDK* and see IDataObject in the *Platform SDK*.

See Also

Reference

CRichEditView Class
Hierarchy Chart
COleServerItem::GetClipboardData

Other Resources

CRichEditView Members

CRichEditView::GetContextMenu

The framework calls this function as part of the processing of IRichEditOleCallback::GetContextMenu.

```
virtual HMENU GetContextMenu(
   WORD seltyp,
   LPOLEOBJECT lpoleobj,
   CHARRANGE* lpchrg
);
```

Parameters

seltyp

The selection type. The selection type values are described in the Remarks section.

lpoleobj

Pointer to a **OLEOBJECT** structure specifying the first selected OLE object if the selection contains one or more OLE items. If the selection contains no items, *lpoleobj* is **NULL**. The **OLEOBJECT** structure holds a pointer to an OLE object v-table.

lpchrg

Pointer to a CHARRANGE structure containing the current selection.

Return Value

Handle to the context menu.

Remarks

This function is a typical part of right mouse-button down processing.

The selection type can be any combination of the following flags:

- **SEL_EMPTY** Indicates that there is no current selection.
- **SEL_TEXT** Indicates that the current selection contains text.
- **SEL_OBJECT** Indicates that the current selection contains at least one OLE item.
- **SEL_MULTICHAR** Indicates that the current selection contains more than one character of text.
- **SEL_MULTIOBJECT** Indicates that the current selection contains more than one OLE object.

The default implementation returns **NULL**. This is an advanced overridable.

For more information, see IRichEditOleCallback::GetContextMenu and CHARRANGE in the *Platform SDK*.

For more information on the **OLEOBJECT** type, see the OLE Data Structures and Structure Allocation article in the *OLE Knowledge Base*.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditCtrl::GetSelectionType

Other Resources

CRichEditView Members

CRichEditView::GetDocument

Call this function to get a pointer to the **CRichEditDoc** associated with this view.

```
CRichEditDoc* GetDocument( ) const;
```

Return Value

Pointer to a CRichEditDoc object associated with your **CRichEditView** object.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditDoc Class
CView::GetDocument
COleClientItem::GetDocument

Other Resources

CRichEditView Members

CRichEditView::GetInPlaceActiveItem

Call this function to get the OLE item that is currently activated in place in this **CRichEditView** object.

```
CRichEditCntrItem* GetInPlaceActiveItem( ) const;
```

Return Value

A pointer to the single, in-place active CRichEditCntrItem object in this rich edit view; **NULL** if there is no OLE item currently in the in-place active state.

See Also

Reference

CRichEditView Class
Hierarchy Chart
COleDocument::GetInPlaceActiveItem
CRichEditCntrItem Class
CRichEditView::GetSelectedItem

Other Resources

CRichEditView Members

CRichEditView::GetMargins

Call this function to retrieve the current margins used in printing.

```
CRect GetMargins( ) const;
```

Return Value

The margins used in printing, measured in **MM_TWIPS**.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::SetMargins
CRichEditView::GetPrintWidth
CRichEditView::GetPrintRect
CRichEditView::GetPaperSize
CRichEditView::PrintPage
CRichEditView::WrapChanged

Other Resources

CRichEditView Members

CRichEditView::GetPageRect

Call this function to get the dimensions of the page used in printing.

```
CRect GetPageRect( ) const;
```

Return Value

The bounds of the page used in printing, measured in **MM_TWIPS**.

Remarks

This value is based on the paper size.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::GetMargins
CRichEditView::GetPrintWidth
CRichEditView::GetPrintRect
CRichEditView::GetPaperSize
CRichEditView::PrintPage

Other Resources

CRichEditView Members

CRichEditView::GetPaperSize

Call this function to retrieve the current paper size.

```
CSize GetPaperSize( ) const;
```

Return Value

The size of the paper used in printing, measured in **MM_TWIPS**.

Example

```
void CMyRichEditView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
   // Get the current paper size and construct an actual printing
   // rectangle by leaving out one half inch margin from each side.
   CSize sizePaper = GetPaperSize();
   CRect rectMargins(720, 720, sizePaper.cx - 720,
      sizePaper.cy - 720);

   // Need to set the margins when printing from CRichEditView
   SetMargins(rectMargins);

   // Set up three rectangular regions spaced an inch apart
   CRect rectHeader(0, 0, rectMargins.right, 1440);
   CRect rectBody(0, 1440, rectMargins.right, 1440 * 2);
   CRect rectFooter(0, 1440 * 2, rectMargins.right, 1440 * 3);

   // Format the first 10 characters in the buffer.
   int nSavedDC = pDC->SaveDC();
   PrintInsideRect(pDC, rectHeader, 0, 10, TRUE); // characters 0-10
   pDC->RestoreDC(nSavedDC);

   // Format the second 10 characters in the buffer.
   nSavedDC = pDC->SaveDC();
   PrintInsideRect(pDC, rectBody, 10, 20, TRUE); // characters 10-20
   pDC->RestoreDC(nSavedDC);

   // Format the third 10 characters in the buffer.
   nSavedDC = pDC->SaveDC();
   PrintInsideRect(pDC, rectFooter, 20, 30, TRUE); // characters 20-30
   pDC->RestoreDC(nSavedDC);

//   CRichEditView::OnPrint(pDC, pInfo);
}
```

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::SetPaperSize
CRichEditView::GetMargins
CRichEditView::GetPrintWidth
CRichEditView::GetPrintRect
CRichEditView::GetPageRect
CRichEditView::PrintPage

Other Resources

CRichEditView Members

CRichEditView::GetParaFormatSelection

Call this function to get the paragraph formatting attributes of the current selection.

```
PARAFORMAT2& GetParaFormatSelection( );
```

Return Value

A PARAFORMAT2 structure which contains the paragraph formatting attributes of the current selection.

Remarks

For more information, see EM_GETPARAFORMAT message and PARAFORMAT2 structure in the *Platform SDK*.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::GetCharFormatSelection
CRichEditView::SetParaFormat
CRichEditCtrl::GetParaFormat

Other Resources

CRichEditView Members

CRichEditView::GetPrintRect

Call this function to retrieve the bounds of the printing area within the page rectangle.

```
CRect GetPrintRect( ) const;
```

Return Value

The bounds of the image area used in printing, measured in **MM_TWIPS**.

Example

```
BOOL CMyRichEditView::OnPreparePrinting(CPrintInfo* pInfo)
{

   CRect rectPrintPage = GetPrintRect();  // Measured in MM_TWIPS

   SetMaxPage( (m_nDocSizeInInches * 1440)/rectPrintPage.Height() );
   return DoPreparePrinting(pInfo);
}
```

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::GetMargins
CRichEditView::GetPrintWidth
CRichEditView::GetPaperSize
CRichEditView::GetPageRect
CRichEditView::PrintPage

Other Resources

CRichEditView Members

CRichEditView::GetPrintWidth

Call this function to determine the width of the printing area.

```
int GetPrintWidth( ) const;
```

Return Value

The width of the printing area, measured in **MM_TWIPS**.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::GetMargins
CRichEditView::GetPrintRect
CRichEditView::GetPaperSize
CRichEditView::GetPageRect
CRichEditView::PrintPage
CRichEditView::WrapChanged

Other Resources

CRichEditView Members

CRichEditView::GetRichEditCtrl

Call this function to retrieve the CRichEditCtrl object associated with the **CRichEditView** object.

```
CRichEditCtrl& GetRichEditCtrl( ) const;
```

Return Value

The **CRichEditCtrl** object for this view.

Example

See the example for CRichEditView::FindText.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditCtrl Class
CEditView::GetEditCtrl
CTreeView::GetTreeCtrl
CListView::GetListCtrl

Other Resources

CRichEditView Members

CRichEditView::GetSelectedItem

Call this function to retrieve the OLE item (a **CRichEditCntrItem** object) currently selected in this **CRichEditView** object.

```
CRichEditCntrItem* GetSelectedItem( ) const;
```

Return Value

Pointer to a CRichEditCntrItem object selected in the **CRichEditView** object; **NULL** if no item is selected in this view.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditCntrItem Class
CRichEditView::GetInPlaceActiveItem

Other Resources

CRichEditView Members

CRichEditView::GetTextLength

Call this function to retrieve the length of the text in this **CRichEditView** object.

```
long GetTextLength( ) const;
```

Return Value

The length of the text in this **CRichEditView** object.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::GetTextLengthEx
CRichEditCtrl::GetTextLength

Other Resources

CRichEditView Members

CRichEditView::GetTextLengthEx

Call this member function to calculate the length of the text in this **CRichEditView** object.

```
long GetTextLengthEx(
   DWORD dwFlags,
   UINT uCodePage = -1
) const;
```

Parameters

dwFlags

Value specifying the method to be used in determining the text length. This member can be one or more of the values listed in the flags member of GETTEXTLENGTHEX described in the *Platform SDK*.

uCodePage

Code page for translation (CP_ACP for ANSI Code Page, 1200 for Unicode).

Return Value

The number of characters or bytes in the edit control. If incompatible flags were set in *dwFlags*, this member function returns **E_INVALIDARG**.

Remarks

GetTextLengthEx provides additional ways of determining the length of the text. It supports the Rich Edit 2.0 functionality. For more information, see About Rich Edit Controls in the *Platform SDK*.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditCtrl::GetTextLengthEx
CRichEditCtrl::GetTextLength

Other Resources

CRichEditView Members

CRichEditView::InsertFileAsObject

Call this function to insert the specified file (as a CRichEditCntrItem object) into a rich edit view.

```
void InsertFileAsObject(
   LPCTSTR lpszFileName
);
```

Parameters

lpszFileName

String containing the name of the file to be inserted.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::InsertItem
CRichEditCntrItem Class

Other Resources

CRichEditView Members

CRichEditView::InsertItem

Call this function to insert a CRichEditCntrItem object into a rich edit view.

```
HRESULT InsertItem(
   CRichEditCntrItem* pItem
);
```

Parameters

pItem

Pointer to the item to be inserted.

Return Value

An **HRESULT** value indicating the success of the insertion.

Remarks

For more information on **HRESULT**, see Structure of COM Error Codes in the *Platform SDK*.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::InsertFileAsObject
CRichEditCntrItem Class

Other Resources

CRichEditView Members

CRichEditView::IsRichEditFormat

Call this function to determine if *cf* is a Clipboard format which is text, rich text, or rich text with OLE items.

```
static BOOL AFX_CDECL IsRichEditFormat(
   CLIPFORMAT cf
);
```

Parameters

cf

The Clipboard format of interest.

Return Value

Nonzero if *cf* is a rich edit or text Clipboard format.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditCtrl::CanPaste
CRichEditCtrl::Paste
CRichEditView::DoPaste

Other Resources

CRichEditView Members

CRichEditView::IsSelected

Call this function to determine if the specified OLE item is currently selected in this view.

```
virtual BOOL IsSelected(
   const CObject* pDocItem
) const;
```

Parameters

pDocItem

Pointer to an object in the view.

Return Value

Nonzero if the object is selected; otherwise 0.

Remarks

Override this function if your derived view class has a different method for handling selection of OLE items.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::GetSelectedItem
CRichEditView::GetInPlaceActiveItem

Other Resources

CRichEditView Members

CRichEditView::OnCharEffect

Call this function to toggle the character formatting effects for the current selection.

```
void OnCharEffect(
   DWORD dwMask,
   DWORD dwEffect
);
```

Parameters

dwMask

The character formatting effects to modify in the current selection.

dwEffect

The desired list of character formatting effects to toggle.

Remarks

Each call to this function toggles the specified formatting effects for the current selection.

For more information on the *dwMask* and *dwEffect* parameters and their potential values, see the corresponding data members of CHARFORMAT in the *Platform SDK*.

Example

```
void CMyRichEditView::OnItalic()
{
   OnCharEffect( CFM_ITALIC, CFE_ITALIC );
}
```

See Also

Reference

CRichEditView Class

Hierarchy Chart

CRichEditView::SetCharFormat

Other Resources

CRichEditView Members

CRichEditView::OnFindNext

Called by the framework when processing commands from the Find/Replace dialog box.

```
virtual void OnFindNext(
   LPCTSTR lpszFind,
   BOOL bNext,
   BOOL bCase,
   BOOL bWord
);
```

Parameters

lpszFind

The string to find.

bNext

The direction to search: **TRUE** indicates down; **FALSE**, up.

bCase

Indicates whether the search is to be case sensitive.

bWord

Indicates whether the search is to match whole words only or not.

Remarks

Call this function to find text within the **CRichEditView**. Override this function to alter search characteristics for your derived view class.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::FindText
CRichEditView::FindTextSimple

Other Resources

CRichEditView Members

CRichEditView::OnInitialUpdate

Called by the framework after the view is first attached to the document, but before the view is initially displayed.

```
virtual void OnInitialUpdate( );
```

Remarks

The default implementation of this function calls the CView::OnUpdate member function with no hint information (that is, using the default values of 0 for the *lHint* parameter and **NULL** for the *pHint* parameter). Override this function to perform any one-time initialization that requires information about the document. For example, if your application has fixed-sized documents, you can use this function to initialize a view's scrolling limits based on the document size. If your application supports variable-sized documents, use **OnUpdate** to update the scrolling limits every time the document changes.

Example

See the example for CRichEditView::m_nWordWrap.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CView::OnUpdate

Other Resources

CRichEditView Members

CRichEditView::OnPasteNativeObject

Use this function to load native data from an embedded item.

```
virtual BOOL OnPasteNativeObject(
   LPSTORAGE lpStg
);
```

Parameters

lpStg

Pointer to an IStorage object.

Return Value

Nonzero if successful; otherwise, 0;

Remarks

Typically, you would do this by creating a COleStreamFile around the **IStorage**. The **COleStreamFile** can be attached to an archive and CObject::Serialize called to load the data.

This is an advanced overridable.

For more information, see IStorage in the *Platform SDK*.

See Also

Reference

CRichEditView Class
Hierarchy Chart
COleStreamFile Class
CObject::Serialize
CArchive Class

Other Resources

CRichEditView Members

CRichEditView::OnParaAlign

Call this function to change the paragraph alignment for the selected paragraphs.

```
void OnParaAlign(
   WORD wAlign
);
```

Parameters

wAlign

Desired paragraph alignment. One of the following values:

- **PFA_LEFT** Align the paragraphs with the left margin.
- **PFA_RIGHT** Align the paragraphs with the right margin.
- **PFA_CENTER** Center the paragraphs between the margins.

Example

```
void CMyRichEditView::OnParaCenter()
{
   OnParaAlign(PFA_CENTER);
}
```

See Also

Reference

CRichEditView Class

Hierarchy Chart

CRichEditView::OnUpdateParaAlign

Other Resources

CRichEditView Members

CRichEditView::OnPrinterChanged

Override this function to change characteristics for this rich edit view when the printer changes.

```
virtual void OnPrinterChanged(
   const CDC& dcPrinter
);
```

Parameters

dcPrinter

A CDC object for the new printer.

Remarks

The default implementation sets the paper size to the physical height and width for the output device (printer). If there is no device context associated with *dcPrinter*, the default implementation sets the paper size to 8.5 by 11 inches.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::SetPaperSize
CRichEditView::WrapChanged

Other Resources

CRichEditView Members

CRichEditView::OnReplaceAll

Called by the framework when processing Replace All commands from the Replace dialog box.

```
virtual void OnReplaceAll(
   LPCTSTR lpszFind,
   LPCTSTR lpszReplace,
   BOOL bCase,
   BOOL bWord
);
```

Parameters

lpszFind

The text to be replaced.

lpszReplace

The replacement text.

bCase

Indicates if the search is case sensitive.

bWord

Indicates if the search must select whole words or not.

Remarks

Call this function to replace all occurrences of some given text with another string. Override this function to alter search characteristics for this view.

Example

See the example for CRichEditView::FindText.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::OnReplaceSel
CRichEditView::OnFindNext

Other Resources

CRichEditView Members

CRichEditView::OnReplaceSel

Called by the framework when processing Replace commands from the Replace dialog box.

```
virtual void OnReplaceSel(
   LPCTSTR lpszFind,
   BOOL bNext,
   BOOL bCase,
   BOOL bWord,
   LPCTSTR lpszReplace
);
```

Parameters

lpszFind

The text to be replaced.

bNext

Indicates the direction of the search: **TRUE** is down; **FALSE**, up.

bCase

Indicates if the search is case sensitive.

bWord

Indicates if the search must select whole words or not.

lpszReplace

The replacement text.

Remarks

Call this function to replace one occurrence of some given text with another string. Override this function to alter search characteristics for this view.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::OnReplaceAll

Other Resources

CRichEditView Members

CRichEditView::OnTextNotFound

Called by the framework whenever a search fails.

```
virtual void OnTextNotFound(
   LPCTSTR lpszFind
);
```

Parameters

lpszFind

The text which was not found.

Remarks

Override this function to change the output notification from a MessageBeep.

For more information, see MessageBeep in the *Platform SDK*.

Example

```
void CMyRichEditViewTestView::OnTextNotFound( LPCTSTR lpszFind )
{
   // Replace the beep with a message box
   TCHAR *lpszMsg = " was not found.";
   TCHAR *lpszBuf = new TCHAR[_tcslen(lpszMsg) + _tcslen(lpszFind)];
   wsprintf( lpszBuf, "%s %s", lpszFind, lpszMsg );
   AfxMessageBox( lpszBuf );
}
```

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::FindText
CRichEditView::FindTextSimple
CRichEditView::OnFindNext

Other Resources

CRichEditView Members

CRichEditView::OnUpdateCharEffect

The framework calls this function to update the command UI for character effect commands.

```
void OnUpdateCharEffect(
   CCmdUI* pCmdUI,
   DWORD dwMask,
   DWORD dwEffect
);
```

Parameters

pCmdUI

Pointer to a CCmdUI object.

dwMask

Indicates the character formatting mask.

dwEffect

Indicates the character formatting effect.

Remarks

The mask *dwMask* specifies which character formatting attributes to check. The flags *dwEffect* list the character formatting attributes to set/clear.

For more information on the *dwMask* and *dwEffect* parameters and their potential values, see the corresponding data members of CHARFORMAT in the *Platform SDK*.

Example

```
void CMyRichEditView::OnUpdateCharItalicUI(CCmdUI* pCmdUI)
{
   OnUpdateCharEffect(pCmdUI, CFM_ITALIC, CFE_ITALIC);
}
```

See Also

Reference

CRichEditView Class

Hierarchy Chart

Other Resources

CRichEditView Members

CRichEditView::OnUpdateParaAlign

The framework calls this function to update the command UI for paragraph effect commands.

```
void OnUpdateParaAlign(
   CCmdUI* pCmdUI,
   WORD wAlign
);
```

Parameters

pCmdUI

Pointer to a CCmdUI object.

wAlign

The paragraph alignment to check. One of the following values:

- **PFA_LEFT** Align the paragraphs with the left margin.
- **PFA_RIGHT** Align the paragraphs with the right margin.
- **PFA_CENTER** Center the paragraphs between the margins.

Example

```
void CMyRichEditView::OnUpdateParaCenterUI(CCmdUI* pCmdUI)
{
   OnUpdateParaAlign(pCmdUI, PFA_CENTER);
}
```

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::GetParaFormatSelection
CRichEditView::OnParaAlign
CRichEditView::SetParaFormat

Other Resources

CRichEditView Members

CRichEditView::PrintInsideRect

Call this function to format a range of text in a rich edit control to fit within *rectLayout* for the device specified by *pDC*.

```
long PrintInsideRect(
   CDC* pDC,
   RECT& rectLayout,
   long nIndexStart,
   long nIndexStop,
   BOOL bOutput
);
```

Parameters

pDC

Pointer to a device context for the output area.

rectLayout

RECT or CRect which defines the output area.

nIndexStart

Zero-based index of the first character to be formatted.

nIndexStop

Zero-based index of the last character to be formatted.

bOutput

Indicates if the text should be rendered. If **FALSE**, the text is just measured.

Return Value

The index of the last character that fits in the output area plus one.

Remarks

Typically, this call is followed by a call to CRichEditCtrl::DisplayBand which generates the output.

Example

See the example for CRichEditView::GetPaperSize.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditCtrl::FormatRange
CRichEditView::PrintPage
CRichEditCtrl::DisplayBand

Other Resources

CRichEditView Members

CRichEditView::PrintPage

Call this function to format a range of text in a rich edit control for the output device specified by *pDC*.

```
long PrintPage(
   CDC* pDC,
   long nIndexStart,
   long nIndexStop
);
```

Parameters

pDC

Pointer to a device context for page output.

nIndexStart

Zero-based index of the first character to be formatted.

nIndexStop

Zero-based index of the last character to be formatted.

Return Value

The index of the last character that fits on the page plus one.

Remarks

The layout of each page is controlled by GetPageRect and GetPrintRect. Typically, this call is followed by a call to CRichEditCtrl::DisplayBand which generates the output.

Note that margins are relative to the physical page, not the logical page. Thus, margins of zero will often clip the text since many printers have unprintable areas on the page. To avoid clipping your text, you should call SetMargins and set reasonable margins before printing.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::PrintInsideRect
CRichEditView::GetPageRect
CRichEditView::GetPrintRect
CRichEditView::SetMargins

Other Resources

CRichEditView Members

CRichEditView::QueryAcceptData

Called by the framework to paste an object into the rich edit.

```
virtual HRESULT QueryAcceptData(
   LPDATAOBJECT lpdataobj,
   CLIPFORMAT* lpcfFormat,
   DWORD dwReco,
   BOOL bReally,
   HGLOBAL hMetaFile
);
```

Parameters

lpdataobj

Pointer to the IDataObject to query.

lpcfFormat

Pointer to the acceptable data format.

dwReco

Not used.

bReally

Indicates if the paste operation should continue or not.

hMetaFile

A handle to the metafile used for drawing the item's icon.

Return Value

An **HRESULT** value reporting the success of the operation.

Remarks

Override this function to handle different organization of COM items in your derived document class. This is an advanced overridable.

For more information on **HRESULT** and **IDataObject**, see Structure of COM Error Codes and IDataObject, respectively, in the *Platform SDK*.

Example

```
// From the WordPad sample, wordpvw.cpp
HRESULT CWordPadView::QueryAcceptData(LPDATAOBJECT lpdataobj,
   CLIPFORMAT* lpcfFormat, DWORD reco, BOOL bReally,
   HGLOBAL hMetaPict)
{
   if (bReally && *lpcfFormat == 0 && (m_nPasteType == 0))
   {
      COleDataObject dataobj;
      dataobj.Attach(lpdataobj, FALSE);
      if (!dataobj.IsDataAvailable(cfRTO)) // native avail, let
                                           // richedit do as it wants
      {
         if (dataobj.IsDataAvailable(cfEmbeddedObject))
         {
            if (PasteNative(lpdataobj)) // See WordPad for info
                                        // on PasteNative
               return S_FALSE;
         }
      }
   }
   return CRichEditView::QueryAcceptData(lpdataobj, lpcfFormat, reco,
```

```
        bReally, hMetaPict);
}
```

See Also

Reference

CRichEditView Class

Hierarchy Chart

Other Resources

CRichEditView Members

CRichEditView::SetCharFormat

Call this function to set the character formatting attributes for new text in this **CRichEditView** object.

```
void SetCharFormat(
   CHARFORMAT2 cf
);
```

Parameters

cf

CHARFORMAT2 structure containing the new default character formatting attributes.

Remarks

Only the attributes specified by the **dwMask** member of *cf* are changed by this function.

For more information, see EM_SETCHARFORMAT message and CHARFORMAT2 structure in the *Platform SDK*.

Example

```
void CMyRichEditView::OnCharUnderline ()
{
    CHARFORMAT2 cf;
    cf = GetCharFormatSelection ();

    if (!(cf.dwMask & CFM_UNDERLINE) || !(cf.dwEffects & CFE_UNDERLINE))
        cf.dwEffects = CFE_UNDERLINE;
    else
        cf.dwEffects = 0;

    cf.dwMask = CFM_UNDERLINE;
    SetCharFormat (cf);
}
```

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::GetCharFormatSelection
CRichEditView::SetParaFormat

Other Resources

CRichEditView Members

CRichEditView::SetMargins

Call this function to set the printing margins for this rich edit view.

```
void SetMargins(
   const CRect& rectMargin
);
```

Parameters

rectMargin

The new margin values for printing, measured in **MM_TWIPS**.

Remarks

If m_nWordWrap is **WrapToTargetDevice**, you should call WrapChanged after using this function to adjust printing characteristics.

Note that the margins used by PrintPage are relative to the physical page, not the logical page. Thus, margins of zero will often clip the text since many printers have unprintable areas on the page. To avoid clipping your text, you should call use **SetMargins** to set reasonable printer margins before printing.

Example

See the example for CRichEditView::GetPaperSize.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::GetMargins
CRichEditView::GetPrintWidth
CRichEditView::GetPrintRect
CRichEditView::GetPaperSize
CRichEditView::GetPageRect
CRichEditView::PrintPage
CRichEditView::WrapChanged

Other Resources

CRichEditView Members

CRichEditView::SetPaperSize

Call this function to set the paper size for printing this rich edit view.

```
void SetPaperSize(
   CSize sizePaper
);
```

Parameters

sizePaper

The new paper size values for printing, measured in **MM_TWIPS**.

Remarks

If m_nWordWrap is **WrapToTargetDevice**, you should call WrapChanged after using this function to adjust printing characteristics.

Example

```
BOOL CMyRichEditView:OnPreparePrinting(CPrintInfo* pInfo)
{
   // Set the printing margins (720 twips = 1/2 inch).
   SetMargins(CRect(720, 720, 720, 720));

   // Change the paper orientation to landscape mode
   // See the example for CWinApp::GetPrinterDeviceDefaults
   ((CMyApp*)AfxGetApp())->SetLandscapeMode();

   // Change the paper size in the CRichEditView to
   // reflect landscape mode
   CSize csPaper = GetPaperSize();
   int temp;
   temp = csPaper.cx; csPaper.cx = csPaper.cy; csPaper.cy = temp;
   SetPaperSize(csPaper);

   return DoPreparePrinting(pInfo);
}
```

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::GetPaperSize
CRichEditView::GetMargins
CRichEditView::GetPrintWidth
CRichEditView::GetPrintRect
CRichEditView::GetPageRect
CRichEditView::PrintPage
CRichEditView::WrapChanged

Other Resources

CRichEditView Members

CRichEditView::SetParaFormat

Call this function to set the paragraph formatting attributes for the current selection in this **CRichEditView** object.

```
BOOL SetParaFormat(
   PARAFORMAT2& pf
);
```

Parameters

pf

PARAFORMAT2 structure containing the new default paragraph formatting attributes.

Return Value

Nonzero if successful; otherwise, 0.

Remarks

Only the attributes specified by the **dwMask** member of *pf* are changed by this function.

For more information, see EM_SETPARAFORMAT message and PARAFORMAT2 structure in the *Platform SDK*.

Example

```
void CMyRichEditView::AddBullets()
{
   PARAFORMAT2 pf;

   pf.cbSize = sizeof(PARAFORMAT2);
   pf.dwMask = PFM_NUMBERING | PFM_OFFSET;
   pf.wNumbering = PFN_BULLET;
   pf.dxOffset = 10;

   VERIFY( SetParaFormat( pf ) );
}
```

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::GetParaFormatSelection
CRichEditView::SetCharFormat

Other Resources

CRichEditView Members

CRichEditView::TextNotFound

Call this function to reset the internal search state of the CRichEditView control after a failed call to FindText.

```
void TextNotFound(
   LPCTSTR lpszFind
);
```

Parameters

lpszFind

Contains the text string that was not found.

Remarks

It is recommended that this method be called immediately after failed calls to FindText so that the internal search state of the control is properly reset.

The *lpszFind* parameter should include the same content as the string provided to FindText. After resetting the internal search state, this method will call the OnTextNotFound method with the provided search string.

Example

See the example for CRichEditView::FindText.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::FindText
CRichEditView::OnTextNotFound

Other Resources

CRichEditView Members

CRichEditView::WrapChanged

Call this function when the printing characteristics have changed (SetMargins or SetPaperSize).

```
virtual void WrapChanged( );
```

Remarks

Override this function to modify the way the rich edit view responds to changes in m_nWordWrap or the printing characteristics (OnPrinterChanged).

Example

```
void CMyRichEditViewTestView::OnInitialUpdate()
{
   CRichEditView::OnInitialUpdate();

   // Turn on the horizontal scroll bar
   m_nWordWrap = WrapNone;
   WrapChanged();
}
```

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::m_nWordWrap
CRichEditView::OnPrinterChanged
CRichEditView::SetMargins
CRichEditView::SetPaperSize

Other Resources

CRichEditView Members

CRichEditView Data Members

For information about the data members in **CRichEditView**, see CRichEditView Members.

CRichEditView::m_nBulletIndent

The indentation for bullet items in a list; by default, 720 units, which is 1/2 inch.

```
int m_nBulletIndent;
```

See Also

Reference

CRichEditView Class

Hierarchy Chart

Other Resources

CRichEditView Members

CRichEditView::m_nWordWrap

Indicates the type of word wrap for this rich edit view.

```
int m_nWordWrap;
```

Remarks

One of the following values:

- **WrapNone** Indicates no automatic word wrapping.
- **WrapToWindow** Indicates word wrapping based on the width of the window.
- **WrapToTargetDevice** Indicates word wrapping based on the characteristics of the target device.

Example

See the example for CRichEditView::WrapChanged.

See Also

Reference

CRichEditView Class
Hierarchy Chart
CRichEditView::WrapChanged

Other Resources

CRichEditView Members

CRuntimeClass Structure

Each class derived from **CObject** is associated with a **CRuntimeClass** structure that you can use to obtain information about an object or its base class at run time.

```
struct CRuntimeClass
```

Remarks

CRuntimeClass is a structure and therefore does not have a base class.

The ability to determine the class of an object at run time is useful when extra type checking of function arguments is needed, or when you must write special-purpose code based on the class of an object. Run-time class information is not supported directly by the C++ language.

CRuntimeClass provides information on the related C++ object, such as a pointer to the **CRuntimeClass** of the base class and the ASCII class name of the related class. This structure also implements various functions that can be used to dynamically create objects, specifying the type of object by using a familiar name, and determining if the related class is derived from a specific class.

For more information on using **CRuntimeClass**, see the article Accessing Run-Time Class Information.

Requirements

Header: afx.h

See Also

Reference

Hierarchy Chart
CObject::GetRuntimeClass
CObject::IsKindOf
RUNTIME_CLASS
IMPLEMENT_DYNAMIC
IMPLEMENT_DYNCREATE
IMPLEMENT_SERIAL

Other Resources

CRuntimeClass Members

CRuntimeClass Members

Data Members

m_lpszClassName	The name of the class.
m_nObjectSize	The size of the object in bytes.
m_pBaseClass	A pointer to the **CRuntimeClass** structure of the base class.
m_pfnCreateObject	A pointer to the function that dynamically creates the object.
m_pfnGetBaseClass	Returns the **CRuntimeClass** structure (only available when dynamically linked).
m_wSchema	The schema number of the class.

Operations

CreateObject	Creates an object during run time.
FromName	Creates an object during run time using the familiar class name.
IsDerivedFrom	Determines if the class is derived from the specified class.

See Also

Reference

CRuntimeClass Structure
Hierarchy Chart

CRuntimeClass Member Functions

For information about the member functions in **CRuntimeClass**, see CRuntimeStructure Members.

CRuntimeClass::CreateObject

Call this function to dynamically create the specified class during run time.

```
CObject* CreateObject( );
static CObject* PASCAL CreateObject(
   LPCSTR lpszClassName
);
static CObject* PASCAL CreateObject(
   LPCWSTR lpszClassName
);
```

Parameters

lpszClassName

The familiar name of the class to be created.

Return Value

A pointer to the newly created object, or **NULL** if the class name is not found or there is insufficient memory to create the object.

Remarks

Classes derived from **CObject** can support dynamic creation, which is the ability to create an object of a specified class at run time. Document, view, and frame classes, for example, should support dynamic creation. For more information on dynamic creation and the **CreateObject** member, see CObject Class and CObject Class: Specifying Levels of Functionality.

Example

See the example for IsDerivedFrom.

See Also

Reference

CRuntimeClass Structure

Hierarchy Chart

Other Resources

CRuntimeClass Members

CRuntimeClass::FromName

Call this function to retrieve the **CRuntimeClass** structure associated with the familiar name.

```
static CRuntimeClass* PASCAL FromName(
   LPCSTR lpszClassName
);
static CRuntimeClass* PASCAL FromName(
   LPCWSTR lpszClassName
);
```

Parameters

lpszClassName

The familiar name of a class derived from **CObject**.

Return Value

A pointer to a **CRuntimeClass** object, corresponding to the name as passed in *lpszClassName*. The function returns **NULL** if no matching class name was found.

Example

```
// This example creates an object if CMyClass is defined.

CRuntimeClass* pMyRTClass= pMyObject->GetRuntimeClass();

CRuntimeClass* pClass = pMyRTClass->FromName("CMyClass");
if (pClass == NULL)
{
   // not found, display a warning for diagnostic purposes
   AfxMessageBox("Warning: CMyClass not defined");
   return NULL;
}

// attempt to create the object with the found CRuntimeClass
CObject* pObject = pClass->CreateObject();
```

Remarks

This method is not supported on Smart Devices because, in order for this method to return a class object, the returned class must use the DECLARE_SERIAL, IMPLEMENT_SERIAL macros, which are not supported for devices.

See Also

Reference

CRuntimeClass Structure
Hierarchy Chart
CRuntimeClass::m_lpszClassName

Other Resources

CRuntimeClass Members

CRuntimeClass::IsDerivedFrom

Call this function to determine if the calling class is derived from the class specified in the *pBaseClass* parameter.

```
BOOL IsDerivedFrom(
   const CRuntimeClass* pBaseClass
) const;
```

Parameters

pBaseClass

The familiar name of a class derived from **CObject**.

Return Value

TRUE if the class calling **IsDerivedFrom** is derived from the base class whose **CRuntimeClass** structure is given as a parameter; otherwise **FALSE**.

Remarks

The relationship is determined by "walking" from the member's class up the chain of derived classes all the way to the top. This function only returns **FALSE** if no match is found for the base class.

Note

To use the **CRuntimeClass** structure, you must include the **IMPLEMENT_DYNAMIC**, **IMPLEMENT_DYNCREATE**, or **IMPLEMENT_SERIAL** macro in the implementation of the class for which you want to retrieve run-time object information.

For more information on using **CRuntimeClass**, see the article CObject Class: Accessing Run-Time Class Information.

Example

```
// This example creates an object from the run-time class. It only
// creates objects derived from CWnd.

// We only want to create an object derived from CWnd.
if (!pClass->IsDerivedFrom(RUNTIME_CLASS(CWnd)))
{
   TRACE(_T("Error; Object %s is not derived from CWnd\n"));
   return;
}

// Get a pointer to the base class CRuntimeClass.
#ifdef _AFXDLL
   CRuntimeClass* pBaseClass = pClass->m_pfnGetBaseClass();
#else
   CRuntimeClass* pBaseClass = pClass->m_pBaseClass;
#endif
ASSERT(pBaseClass != NULL);

TRACE(_T("Creating object %s derived from %s, with object size %d
   and schema %d\n"), pClass->m_lpszClassName,
   pBaseClass->m_lpszClassName, pClass->m_nObjectSize,
   pClass->m_wSchema);

// Create the object.
CObject* pObject = pClass->CreateObject();
```

See Also

Reference

CRuntimeClass Structure

Hierarchy Chart

Other Resources

CRuntimeClass Data Members

For information about the data members in **CRuntimeClass**, see CRuntimeClass Members.

CRuntimeClass::m_lpszClassName

A null-terminated string containing the ASCII class name.

Remarks

This name can be used to create an instance of the class using the **FromName** member function.

Example

See the example for IsDerivedFrom.

See Also

Reference

CRuntimeClass Structure
Hierarchy Chart
CRuntimeClass::FromName

Other Resources

CRuntimeClass Members

CRuntimeClass::m_nObjectSize

The size of the object, in bytes.

Remarks

If the object has data members that point to allocated memory, the size of that memory is not included.

Example

See the example for IsDerivedFrom.

See Also

Reference

CRuntimeClass Structure

Hierarchy Chart

Other Resources

CRuntimeClass Members

CRuntimeClass::m_pBaseClass

If your application statically links to MFC, this data member contains a pointer to the **CRuntimeClass** structure of the base class.

Remarks

If your application dynamically links to the MFC library, see m_pfnGetBaseClass.

Example

See the example for IsDerivedFrom.

See Also

Reference

CRuntimeClass Structure

Hierarchy Chart

Other Resources

CRuntimeClass Members

CRuntimeClass::m_pfnCreateObject

A function pointer to the default constructor that creates an object of your class.

Remarks

This pointer is only valid if the class supports dynamic creation; otherwise, the function returns **NULL**.

See Also

Reference

CRuntimeClass Structure

Hierarchy Chart

Other Resources

CRuntimeClass Members

CRuntimeClass::m_pfnGetBaseClass

If your application uses the MFC library as a shared DLL, this data member points to a function that returns the **CRuntimeClass** structure of the base class.

Remarks

If your application statically links to the MFC library, see m_pBaseClass.

Example

See the example for IsDerivedFrom.

See Also

Reference

CRuntimeClass Structure

Hierarchy Chart

Other Resources

CRuntimeClass Members

CRuntimeClass::m_wSchema

The schema number (-1 for nonserializable classes).

Remarks

For more information on schema numbers, see the IMPLEMENT_SERIAL macro.

Example

See the example for IsDerivedFrom.

See Also

Reference

CRuntimeClass Structure

Hierarchy Chart

Other Resources

CRuntimeClass Members

CScrollBar Class

Provides the functionality of a Windows scroll-bar control.

```
class CScrollBar : public CWnd
```

Remarks

You create a scroll-bar control in two steps. First, call the constructor **CScrollBar** to construct the **CScrollBar** object, then call the Create member function to create the Windows scroll-bar control and attach it to the **CScrollBar** object.

If you create a **CScrollBar** object within a dialog box (through a dialog resource), the **CScrollBar** is automatically destroyed when the user closes the dialog box.

If you create a **CScrollBar** object within a window, you may also need to destroy it.

If you create the **CScrollBar** object on the stack, it is destroyed automatically. If you create the **CScrollBar** object on the heap by using the **new** function, you must call **delete** on the object to destroy it when the user terminates the Windows scroll bar.

If you allocate any memory in the **CScrollBar** object, override the **CScrollBar** destructor to dispose of the allocations.

For related information about using **CScrollBar**, see Controls.

Requirements

Header: afxwin.h

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd Class
CButton Class
CComboBox Class
CEdit Class
CListBox Class
CStatic Class
CDialog Class

Other Resources

CScrollBar Members

CScrollBar Members

Base Class Members

COBJECT Members

CCmdTarget Members

CWnd Members

Construction

CScrollBar	Constructs a **CScrollBar** object.

Initialization

Create	Creates the Windows scroll bar and attaches it to the **CScrollBar** object.

Operations

EnableScrollBar	Enables or disables one or both arrows of a scroll bar.
GetScrollBarInfo	Retrieves information about the scroll bar using a **SCROLLBARINFO** structure.
GetScrollInfo	Retrieves information about the scroll bar.
GetScrollLimit	Retrieves the limit of the scroll bar
GetScrollPos	Retrieves the current position of a scroll box.
GetScrollRange	Retrieves the current minimum and maximum scroll-bar positions for the given scroll bar.
SetScrollInfo	Sets information about the scroll bar.
SetScrollPos	Sets the current position of a scroll box.
SetScrollRange	Sets minimum and maximum position values for the given scroll bar.
ShowScrollBar	Shows or hides a scroll bar.

See Also

Reference

CScrollBar Class

Hierarchy Chart

CScrollBar Member Functions

For information about the member functions in **CScrollBar**, see CScrollBar Members.

CScrollBar::Create

Creates the Windows scroll bar and attaches it to the **CScrollBar** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the scroll bar's style. Apply any combination of scroll-bar styles to the scroll bar.

rect

Specifies the scroll bar's size and position. Can be either a **RECT** structure or a **CRect** object.

pParentWnd

Specifies the scroll bar's parent window, usually a **CDialog** object. It must not be **NULL**.

nID

The scroll bar's control ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

You construct a **CScrollBar** object in two steps. First, call the constructor, which constructs the **CScrollBar** object; then call **Create**, which creates and initializes the associated Windows scroll bar and attaches it to the **CScrollBar** object.

Apply the following window styles to a scroll bar:

- **WS_CHILD** Always
- **WS_VISIBLE** Usually
- **WS_DISABLED** Rarely
- **WS_GROUP** To group controls

Example

```
// Example 1:
// Create a horizontal CScrollBar control as a child window of CMyView
// class (a CView-derived class). The scroll bar is NOT visible until the
// call ShowScrollBar() is made. m_ScrollBar is of type CScrollBar class,
// and it is a member variable in CMyView class.
int CMyView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CView::OnCreate(lpCreateStruct) == -1)
      return -1;

   VERIFY(m_ScrollBar.Create(
      SBS_HORZ | SBS_TOPALIGN | WS_CHILD, CRect(5, 5, 100, 30), this, 100));

   m_ScrollBar.ShowScrollBar();

   return 0;
}

// Example 2:
```

```
// Create a horizontal CScrollBar control as a child window of CMyView
// class (a CView-derived class). m_ScrollBar is of type CScrollBar
// class, and it is a member variable in CMyView class.
int CMyView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CView::OnCreate(lpCreateStruct) == -1)
      return -1;

   VERIFY(m_ScrollBar.Create(
      SBS_HORZ | SBS_TOPALIGN | WS_CHILD | WS_VISIBLE,
      CRect(5, 5, 100, 30), this, 100));

   return 0;
}
```

See Also

Reference

CScrollBar Class

Hierarchy Chart

CScrollBar::CScrollBar

Other Resources

CScrollBar Members

CScrollBar::CScrollBar

Constructs a **CScrollBar** object.

```
CScrollBar( );
```

Remarks

After constructing the object, call the **Create** member function to create and initialize the Windows scroll bar.

Example

```
// Declare a CScrollBar control.
CScrollBar m_ScrollBar;
```

See Also

Reference

CScrollBar Class

Hierarchy Chart

CScrollBar::Create

Other Resources

CScrollBar Members

CScrollBar::EnableScrollBar

Enables or disables one or both arrows of a scroll bar.

```
BOOL EnableScrollBar(
   UINT nArrowFlags = ESB_ENABLE_BOTH
);
```

Parameters

nArrowFlags

Specifies whether the scroll arrows are enabled or disabled and which arrows are enabled or disabled. This parameter can be one of the following values:

- **ESB_ENABLE_BOTH** Enables both arrows of a scroll bar.
- **ESB_DISABLE_LTUP** Disables the left arrow of a horizontal scroll bar or the up arrow of a vertical scroll bar.
- **ESB_DISABLE_RTDN** Disables the right arrow of a horizontal scroll bar or the down arrow of a vertical scroll bar.
- **ESB_DISABLE_BOTH** Disables both arrows of a scroll bar.

Return Value

Nonzero if the arrows are enabled or disabled as specified; otherwise 0, which indicates that the arrows are already in the requested state or that an error occurred.

Example

See the example for CScrollBar::SetScrollRange.

See Also

Reference

CScrollBar Class
Hierarchy Chart
CWnd::EnableScrollBar
EnableScrollBar

Other Resources

CScrollBar Members

CScrollBar::GetScrollBarInfo

Retrieves the information that the **SCROLLBARINFO** structure maintains about a scroll bar.

```
BOOL GetScrollBarInfo(
   PSCROLLBARINFO pScrollInfo
) const;
```

Parameters

pScrollInfo

A pointer to the SCROLLBARINFO structure.

Return Value

Returns **TRUE** on success, **FALSE** on failure.

Remarks

This member function emulates the functionality of the SBM_SCROLLBARINFO message, as described in the *Platform SDK*.

See Also

Reference

CScrollBar Class

Hierarchy Chart

Other Resources

CScrollBar Members

CScrollBar::GetScrollInfo

Retrieves the information that the **SCROLLINFO** structure maintains about a scroll bar.

```
BOOL GetScrollInfo(
   LPSCROLLINFO lpScrollInfo,
   UINT nMask = SIF_ALL
);
```

Parameters

lpScrollInfo

A pointer to a SCROLLINFO structure. See the *Platform SDK* for more information about this structure.

nMask

Specifies the scroll bar parameters to retrieve. Typical usage, SIF_ALL, specifies a combination of SIF_PAGE, SIF_POS, SIF_TRACKPOS, and SIF_RANGE. See **SCROLLINFO** for more information on the nMask values.

Return Value

If the message retrieved any values, the return is **TRUE**. Otherwise, it is **FALSE**.

Remarks

GetScrollInfo enables applications to use 32-bit scroll positions.

The SCROLLINFO structure contains information about a scroll bar, including the minimum and maximum scrolling positions, the page size, and the position of the scroll box (the thumb). See the **SCROLLINFO** structure topic in the *Platform SDK* for more information about changing the structure defaults.

The MFC Windows message handlers that indicate scroll bar position, CWnd::OnHScroll and CWnd::OnVScroll, provide only 16 bits of position data. **GetScrollInfo** and **SetScrollInfo** provide 32 bits of scroll bar position data. Thus, an application can call **GetScrollInfo** while processing either **CWnd::OnHScroll** or **CWnd::OnVScroll** to obtain 32-bit scroll bar position data.

Example

See the example for CWnd::OnHScroll.

See Also

Reference

CScrollBar Class
Hierarchy Chart
CScrollBar::SetScrollInfo
CWnd::SetScrollInfo
CWnd::SetScrollPos
CWnd::OnVScroll
CWnd::OnHScroll
SCROLLINFO

Other Resources

CScrollBar Members

CScrollBar::GetScrollLimit

Retrieves the maximum scrolling position of the scroll bar.

```
int GetScrollLimit( );
```

Return Value

Specifies the maximum position of a scroll bar if successful; otherwise 0.

Example

See the example for CWnd::OnHScroll.

See Also

Reference

CScrollBar Class
Hierarchy Chart
CWnd::GetScrollLimit

Other Resources

CScrollBar Members

CScrollBar::GetScrollPos

Retrieves the current position of a scroll box.

```
int GetScrollPos( ) const;
```

Return Value

Specifies the current position of the scroll box if successful; otherwise 0.

Remarks

The current position is a relative value that depends on the current scrolling range. For example, if the scrolling range is 100 to 200 and the scroll box is in the middle of the bar, the current position is 150.

Example

See the example for CWnd::OnHScroll.

See Also

Reference

CScrollBar Class
Hierarchy Chart
CScrollBar::SetScrollPos
CScrollBar::GetScrollRange
CScrollBar::SetScrollRange
GetScrollPos

Other Resources

CScrollBar Members

CScrollBar::GetScrollRange

Copies the current minimum and maximum scroll-bar positions for the given scroll bar to the locations specified by *lpMinPos* and *lpMaxPos*.

```
void GetScrollRange(
   LPINT lpMinPos,
   LPINT lpMaxPos
) const;
```

Parameters

lpMinPos

Points to the integer variable that is to receive the minimum position.

lpMaxPos

Points to the integer variable that is to receive the maximum position.

Remarks

The default range for a scroll-bar control is empty (both values are 0).

Example

See the example for CWnd::OnHScroll.

See Also

Reference

CScrollBar Class
Hierarchy Chart
GetScrollRange
CScrollBar::SetScrollRange
CScrollBar::GetScrollPos
CScrollBar::SetScrollPos

Other Resources

CScrollBar Members

CScrollBar::SetScrollInfo

Sets the information that the **SCROLLINFO** structure maintains about a scroll bar.

```
BOOL SetScrollInfo(
   LPSCROLLINFO lpScrollInfo,
   BOOL bRedraw = TRUE
);
```

Parameters

lpScrollInfo

A pointer to a SCROLLINFO structure.

bRedraw

Specifies whether the scroll bar should be redrawn to reflect the new information. If *bRedraw* is **TRUE**, the scroll bar is redrawn. If it is **FALSE**, it is not redrawn. The scroll bar is redrawn by default.

Return Value

If successful, the return is **TRUE**. Otherwise, it is **FALSE**.

Remarks

You must provide the values required by the **SCROLLINFO** structure parameters, including the flag values.

The **SCROLLINFO** structure contains information about a scroll bar, including the minimum and maximum scrolling positions, the page size, and the position of the scroll box (the thumb). See the SCROLLINFO structure topic in the *Platform SDK* for more information about changing the structure defaults.

Example

```
// CMyView is a CView-derived class.
void CMyView::OnInitialUpdate()
{
   CView::OnInitialUpdate();

   // Set SCROLLINFO for the scroll bar. m_ScrollBar is of type
   // CScrollBar class, and it is a member variable in CMyView class.
   SCROLLINFO info;
   info.cbSize = sizeof(SCROLLINFO);
   info.fMask = SIF_ALL;
   info.nMin = 0;
   info.nMax = 10;
   info.nPage = 2;
   info.nPos = 5;
   info.nTrackPos = 2;
   m_ScrollBar.SetScrollInfo(&info);
}
```

See Also

Reference

CScrollBar Class
Hierarchy Chart
CScrollBar::GetScrollInfo
CWnd::SetScrollInfo
CWnd::SetScrollPos
CWnd::OnVScroll
CWnd::OnHScroll
CWnd::GetScrollInfo
SCROLLINFO

Other Resources

CScrollBar Members

CScrollBar::SetScrollPos

Sets the current position of a scroll box to that specified by *nPos* and, if specified, redraws the scroll bar to reflect the new position.

```
int SetScrollPos(
   int nPos,
   BOOL bRedraw = TRUE
);
```

Parameters

nPos

Specifies the new position for the scroll box. It must be within the scrolling range.

bRedraw

Specifies whether the scroll bar should be redrawn to reflect the new position. If *bRedraw* is **TRUE**, the scroll bar is redrawn. If it is **FALSE**, it is not redrawn. The scroll bar is redrawn by default.

Return Value

Specifies the previous position of the scroll box if successful; otherwise 0.

Remarks

Set *bRedraw* to **FALSE** whenever the scroll bar will be redrawn by a subsequent call to another function to avoid having the scroll bar redrawn twice within a short interval.

Example

See the example for CScrollBar::SetScrollRange.

See Also

Reference

CScrollBar Class
Hierarchy Chart
CScrollBar::GetScrollPos
CScrollBar::GetScrollRange
CScrollBar::SetScrollRange
SetScrollPos

Other Resources

CScrollBar Members

CScrollBar::SetScrollRange

Sets minimum and maximum position values for the given scroll bar.

```
void SetScrollRange(
   int nMinPos,
   int nMaxPos,
   BOOL bRedraw = TRUE
);
```

Parameters

nMinPos

Specifies the minimum scrolling position.

nMaxPos

Specifies the maximum scrolling position.

bRedraw

Specifies whether the scroll bar should be redrawn to reflect the change. If *bRedraw* is **TRUE**, the scroll bar is redrawn; if **FALSE**, it is not redrawn. It is redrawn by default.

Remarks

Set *nMinPos* and *nMaxPos* to 0 to hide standard scroll bars.

Do not call this function to hide a scroll bar while processing a scroll-bar notification message.

If a call to **SetScrollRange** immediately follows a call to the **SetScrollPos** member function, set *bRedraw* in **SetScrollPos** to 0 to prevent the scroll bar from being redrawn twice.

The difference between the values specified by *nMinPos* and *nMaxPos* must not be greater than 32,767. The default range for a scroll-bar control is empty (both *nMinPos* and *nMaxPos* are 0).

Example

```
// CMyView is a CView-derived class.
void CMyView::OnInitialUpdate()
{
   CView::OnInitialUpdate();

   // Sets minimum (0) and maximum (10) position values for the
   // CScrollBar control. m_ScrollBar is of type CScrollBar class,
   // and it is a member variable in CMyView class.
   m_ScrollBar.SetScrollRange(0, 10);

   // Set the position of the scroll box.
   m_ScrollBar.SetScrollPos(5);

   // Disable the right arrow of the scroll bar. By default, both arrows
   // are enabled.
   m_ScrollBar.EnableScrollBar(ESB_DISABLE_RTDN);
}
```

See Also

Reference

CScrollBar Class
Hierarchy Chart
CScrollBar::GetScrollPos
CScrollBar::SetScrollPos
CScrollBar::GetScrollRange
SetScrollRange

Other Resources

CScrollBar::ShowScrollBar

Shows or hides a scroll bar.

```
void ShowScrollBar(
   BOOL bShow = TRUE
);
```

Parameters

bShow

Specifies whether the scroll bar is shown or hidden. If this parameter is **TRUE**, the scroll bar is shown; otherwise it is hidden.

Remarks

An application should not call this function to hide a scroll bar while processing a scroll-bar notification message.

Example

See the example for CScrollBar::Create.

See Also

Reference

CScrollBar Class
Hierarchy Chart
CScrollBar::GetScrollPos
CScrollBar::GetScrollRange
CWnd::ScrollWindow
CScrollBar::SetScrollPos
CScrollBar::SetScrollRange

Other Resources

CScrollBar Members

CScrollView Class

A CView with scrolling capabilities.

```
class CScrollView : public CView
```

Remarks

You can handle standard scrolling yourself in any class derived from **CView** by overriding the message-mapped OnHScroll and OnVScroll member functions. But **CScrollView** adds the following features to its **CView** capabilities:

- It manages window and viewport sizes and mapping modes.
- It scrolls automatically in response to scroll-bar messages.
- It scrolls automatically in response to messages from the keyboard, a non-scrolling mouse, or the IntelliMouse wheel.

To scroll automatically in response to messages from the keyboard, add a WM_KEYDOWN message, and test for VK_DOWN, VK_PREV and call SetScrollPos.

You can handle mouse wheel scrolling yourself by overriding the message-mapped OnMouseWheel and OnRegisteredMouseWheel member functions. As they are for **CScrollView**, these member functions support the recommended behaviour for WM_MOUSEWHEEL, the wheel rotation message.

To take advantage of automatic scrolling, derive your view class from **CScrollView** instead of from **CView**. When the view is first created, if you want to calculate the size of the scrollable view based on the size of the document, call the **SetScrollSizes** member function from your override of either CView::OnInitialUpdate or CView::OnUpdate. (You must write your own code to query the size of the document. For an example, see the Scribble sample.)

The call to the **SetScrollSizes** member function sets the view's mapping mode, the total dimensions of the scroll view, and the amounts to scroll horizontally and vertically. All sizes are in logical units. The logical size of the view is usually calculated from data stored in the document, but in some cases you may want to specify a fixed size. For examples of both approaches, see CScrollView::SetScrollSizes.

You specify the amounts to scroll horizontally and vertically in logical units. By default, if the user clicks a scroll bar shaft outside of the scroll box, **CScrollView** scrolls a "page." If the user clicks a scroll arrow at either end of a scroll bar, **CScrollView** scrolls a "line." By default, a page is 1/10 of the total size of the view; a line is 1/10 of the page size. Override these default values by passing custom sizes in the **SetScrollSizes** member function. For example, you might set the horizontal size to some fraction of the width of the total size and the vertical size to the height of a line in the current font.

Instead of scrolling, **CScrollView** can automatically scale the view to the current window size. In this mode, the view has no scroll bars and the logical view is stretched or shrunk to exactly fit the window's client area. To use this scale-to-fit capability, call CScrollView::SetScaleToFitSize. (Call either **SetScaleToFitSize** or **SetScrollSizes**, but not both.)

Before the `OnDraw` member function of your derived view class is called, **CScrollView** automatically adjusts the viewport origin for the **CPaintDC** device-context object that it passes to `OnDraw`.

To adjust the viewport origin for the scrolling window, **CScrollView** overrides CView::OnPrepareDC. This adjustment is automatic for the **CPaintDC** device context that **CScrollView** passes to `OnDraw`, but you must call **CScrollView::OnPrepareDC** yourself for any other device contexts you use, such as a **CClientDC**. You can override **CScrollView::OnPrepareDC** to set the pen, background color, and other drawing attributes, but call the base class to do scaling.

Scroll bars can appear in three places relative to a view, as shown in the following cases:

- Standard window-style scroll bars can be set for the view using the **WS_HSCROLL** and **WS_VSCROLL** Windows Styles.
- Scroll-bar controls can also be added to the frame containing the view, in which case the framework forwards **WM_HSCROLL** and **WM_VSCROLL** messages from the frame window to the currently active view.
- The framework also forwards scroll messages from a **CSplitterWnd** splitter control to the currently active splitter pane (a view). When placed in a CSplitterWnd with shared scroll bars, a **CScrollView** object will use the shared ones rather than creating its own.

For more information on using **CScrollView**, see Document/View Architecture and Derived View Classes Available in MFC.

Requirements

Header: afxwin.h

See Also

Tasks

DIBLOOK Sample: Illustrates the Use of DIBs and Color Palettes

Reference

CView Class

Hierarchy Chart

CView Class

CSplitterWnd Class

Other Resources

CScrollView Members

CScrollView Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CView Members

Operations

CheckScrollBars	Indicates whether the scroll view has horizontal and vertical scroll bars.
FillOutsideRect	Fills the area of a view outside the scrolling area.
GetDeviceScrollPosition	Gets the current scroll position in device units.
GetDeviceScrollSizes	Gets the current mapping mode, the total size, and the line and page sizes of the scrollable view. Sizes are in device units.
GetScrollPosition	Gets the current scroll position in logical units.
GetTotalSize	Gets the total size of the scroll view in logical units.
ResizeParentToFit	Causes the size of the view to dictate the size of its frame.
ScrollToPosition	Scrolls the view to a given point, specified in logical units.
SetScaleToFitSize	Puts the scroll view into scale-to-fit mode.
SetScrollSizes	Sets the scroll view's mapping mode, total size, and horizontal and vertical scroll amounts.

Construction

CScrollView	Constructs a **CScrollView** object.

See Also

Reference

CScrollView Class

Hierarchy Chart

CScrollView Member Functions

For information about the member functions in **CScrollView**, see CScrollView Members.

CScrollView::CheckScrollBars

Call this member function to determine if the scroll view has horizontal and vertical bars.

```
void CheckScrollBars(
   BOOL& bHasHorzBar,
   BOOL& bHasVertBar
) const;
```

Parameters

bHasHorzBar

Indicates the application has a horizontal scroll bar.

bHasVertBar

Indicates the application has a vertical scroll bar.

See Also

Reference

CScrollView::GetDeviceScrollPosition

CScrollView::GetDeviceScrollPosition

CScrollView::CScrollView

Constructs a **CScrollView** object.

```
CScrollView( );
```

Remarks

You must call either **SetScrollSizes** or **SetScaleToFitSize** before the scroll view is usable.

See Also

Reference

CScrollView Class
Hierarchy Chart
CScrollView::SetScrollSizes
CScrollView::SetScaleToFitSize

Other Resources

CScrollView Members

CScrollView::FillOutsideRect

Call **FillOutsideRect** to fill the area of the view that appears outside of the scrolling area.

```
void FillOutsideRect(
   CDC* pDC,
   CBrush* pBrush
);
```

Parameters

pDC

Device context in which the filling is to be done.

pBrush

Brush with which the area is to be filled.

Remarks

Use **FillOutsideRect** in your scroll view's **OnEraseBkgnd** handler function to prevent excessive background repainting.

Example

```
BOOL CScaleView::OnEraseBkgnd( CDC* pDC )
{
    CBrush br( GetSysColor( COLOR_WINDOW ) );
    FillOutsideRect( pDC, &br );
    return TRUE;                    // Erased
}
```

See Also

Reference

CScrollView Class

Hierarchy Chart

CWnd::OnEraseBkgnd

Other Resources

CScrollView Members

CScrollView::GetDeviceScrollPosition

Call **GetDeviceScrollPosition** when you need the current horizontal and vertical positions of the scroll boxes in the scroll bars.

```
CPoint GetDeviceScrollPosition( ) const;
```

Return Value

The horizontal and vertical positions (in device units) of the scroll boxes as a **CPoint** object.

Remarks

This coordinate pair corresponds to the location in the document to which the upper-left corner of the view has been scrolled. This is useful for offsetting mouse-device positions to scroll-view device positions.

GetDeviceScrollPosition returns values in device units. If you want logical units, use **GetScrollPosition** instead.

See Also

Reference

CScrollView Class
Hierarchy Chart
CScrollView::GetScrollPosition

Other Resources

CScrollView Members

CScrollView::GetDeviceScrollSizes

GetDeviceScrollSizes gets the current mapping mode, the total size, and the line and page sizes of the scrollable view.

```
void GetDeviceScrollSizes(
   int& nMapMode,
   SIZE& sizeTotal,
   SIZE& sizePage,
   SIZE& sizeLine
) const;
```

Parameters

nMapMode

Returns the current mapping mode for this view. For a list of possible values, see **SetScrollSizes**.

sizeTotal

Returns the current total size of the scroll view in device units.

sizePage

Returns the current horizontal and vertical amounts to scroll in each direction in response to a mouse click in a scroll-bar shaft. The **cx** member contains the horizontal amount. The **cy** member contains the vertical amount.

sizeLine

Returns the current horizontal and vertical amounts to scroll in each direction in response to a mouse click in a scroll arrow. The **cx** member contains the horizontal amount. The **cy** member contains the vertical amount.

Remarks

Sizes are in device units. This member function is rarely called.

See Also

Reference

CScrollView Class
Hierarchy Chart
CScrollView::SetScrollSizes
CScrollView::GetTotalSize

Other Resources

CScrollView Members

CScrollView::GetScrollPosition

Call **GetScrollPosition** when you need the current horizontal and vertical positions of the scroll boxes in the scroll bars.

```
CPoint GetScrollPosition( ) const;
```

Return Value

The horizontal and vertical positions (in logical units) of the scroll boxes as a **CPoint** object.

Remarks

This coordinate pair corresponds to the location in the document to which the upper-left corner of the view has been scrolled.

GetScrollPosition returns values in logical units. If you want device units, use **GetDeviceScrollPosition** instead.

See Also

Reference

CScrollView Class
Hierarchy Chart
CScrollView::GetDeviceScrollPosition

Other Resources

CScrollView Members

CScrollView::GetTotalSize

Call **GetTotalSize** to retrieve the current horizontal and vertical sizes of the scroll view.

```
CSize GetTotalSize( ) const;
```

Return Value

The total size of the scroll view in logical units. The horizontal size is in the **cx** member of the **CSize** return value. The vertical size is in the **cy** member.

See Also

Reference

CScrollView Class
Hierarchy Chart
CScrollView::GetDeviceScrollSizes
CScrollView::SetScrollSizes

Other Resources

CScrollView Members

CScrollView::ResizeParentToFit

Call **ResizeParentToFit** to let the size of your view dictate the size of its frame window.

```
void ResizeParentToFit(
   BOOL bShrinkOnly = TRUE
);
```

Parameters

bShrinkOnly

The kind of resizing to perform. The default value, **TRUE**, shrinks the frame window if appropriate. Scroll bars will still appear for large views or small frame windows. A value of **FALSE** causes the view always to resize the frame window exactly. This can be somewhat dangerous since the frame window could get too big to fit inside the multiple document interface (MDI) frame window or the screen.

Remarks

This is recommended only for views in MDI child frame windows. Use **ResizeParentToFit** in the **OnInitialUpdate** handler function of your derived **CScrollView** class. For an example of this member function, see CScrollView::SetScrollSizes.

ResizeParentToFit assumes that the size of the view window has been set. If the view window size has not been set when **ResizeParentToFit** is called, you will get an assertion. To ensure that this does not happen, make the following call before calling **ResizeParentToFit**:

```
GetParentFrame()->RecalcLayout();
```

See Also

Reference

CScrollView Class
Hierarchy Chart
CView::OnInitialUpdate
CScrollView::SetScrollSizes

Other Resources

CScrollView Members

CScrollView::ScrollToPosition

Call **ScrollToPosition** to scroll to a given point in the view.

```
void ScrollToPosition(
   POINT pt
);
```

Parameters

pt

The point to scroll to, in logical units. The **x** member must be a positive value (greater than or equal to 0, up to the total size of the view). The same is true for the **y** member when the mapping mode is **MM_TEXT**. The **y** member is negative in mapping modes other than **MM_TEXT**.

Remarks

The view will be scrolled so that this point is at the upper-left corner of the window. This member function must not be called if the view is scaled to fit.

See Also

Reference

CScrollView Class
Hierarchy Chart
CScrollView::GetDeviceScrollPosition
CScrollView::SetScaleToFitSize
CScrollView::SetScrollSizes

Other Resources

CScrollView Members

CScrollView::SetScaleToFitSize

Call **SetScaleToFitSize** when you want to scale the viewport size to the current window size automatically.

```
void SetScaleToFitSize(
   SIZE sizeTotal
);
```

Parameters

sizeTotal

The horizontal and vertical sizes to which the view is to be scaled. The scroll view's size is measured in logical units. The horizontal size is contained in the **cx** member. The vertical size is contained in the **cy** member. Both **cx** and **cy** must be greater than or equal to 0.

Remarks

With scroll bars, only a portion of the logical view may be visible at any time. But with the scale-to-fit capability, the view has no scroll bars and the logical view is stretched or shrunk to exactly fit the window's client area. When the window is resized, the view draws its data at a new scale based on the size of the window.

You'll typically place the call to **SetScaleToFitSize** in your override of the view's **OnInitialUpdate** member function. If you do not want automatic scaling, call the **SetScrollSizes** member function instead.

SetScaleToFitSize can be used to implement a "Zoom to Fit" operation. Use **SetScrollSizes** to reinitialize scrolling.

SetScaleToFitSize assumes that the size of the view window has been set. If the view window size has not been set when **SetScaleToFitSize** is called, you will get an assertion. To ensure that this does not happen, make the following call before calling **SetScaleToFitSize**:

```
GetParentFrame()->RecalcLayout();
```

See Also

Reference

CScrollView Class
Hierarchy Chart
CScrollView::SetScrollSizes
CView::OnInitialUpdate

Other Resources

CScrollView Members

CScrollView::SetScrollSizes

Call **SetScrollSizes** when the view is about to be updated.

```
void SetScrollSizes(
   int nMapMode,
   SIZE sizeTotal,
   const SIZE& sizePage = sizeDefault,
   const SIZE& sizeLine = sizeDefault
);
```

Parameters

nMapMode

The mapping mode to set for this view. Possible values include:

Mapping Mode	Logical Unit	Positive y-axis Extends...
MM_TEXT	1 pixel	Downward
MM_HIMETRIC	0.01 mm	Upward
MM_TWIPS	1/1440 in	Upward
MM_HIENGLISH	0.001 in	Upward
MM_LOMETRIC	0.1 mm	Upward
MM_LOENGLISH	0.01 in	Upward

All of these modes are defined by Windows. Two standard mapping modes, **MM_ISOTROPIC** and **MM_ANISOTROPIC**, are not used for **CScrollView**. The class library provides the **SetScaleToFitSize** member function for scaling the view to window size. Column three in the table above describes the coordinate orientation.

sizeTotal

The total size of the scroll view. The **cx** member contains the horizontal extent. The **cy** member contains the vertical extent. Sizes are in logical units. Both **cx** and **cy** must be greater than or equal to 0.

sizePage

The horizontal and vertical amounts to scroll in each direction in response to a mouse click in a scroll-bar shaft. The **cx** member contains the horizontal amount. The **cy** member contains the vertical amount.

sizeLine

The horizontal and vertical amounts to scroll in each direction in response to a mouse click in a scroll arrow. The **cx** member contains the horizontal amount. The **cy** member contains the vertical amount.

Remarks

Call it in your override of the **OnUpdate** member function to adjust scrolling characteristics when, for example, the document is initially displayed or when it changes size.

You will typically obtain size information from the view's associated document by calling a document member function, perhaps called `GetMyDocSize`, that you supply with your derived document class. The following code shows this approach:

```
SetScrollSizes( nMapMode, GetDocument( )->GetMyDocSize( ) );
```

Alternatively, you might sometimes need to set a fixed size, as in the following code:

```
SetScrollSizes( nMapMode, CSize(100, 100) );
```

You must set the mapping mode to any of the Windows mapping modes except **MM_ISOTROPIC** or **MM_ANISOTROPIC**. If you want to use an unconstrained mapping mode, call the **SetScaleToFitSize** member function instead of **SetScrollSizes**.

Example

```
void CScaleView::OnUpdate( )
{
   // ...
   // Implement a GetDocSize( ) member function in
   // your document class; it returns a CSize.
   SetScrollSizes( MM_LOENGLISH, GetDocument( )->GetDocSize( ) );
   ResizeParentToFit( );   // Default bShrinkOnly argument
   // ...
}

// another example
void CMyView::OnInitialUpdate()
{
   // The GetDocSize( ) member function is implemented in
   // your document class. The return type is CSize.
   SetScrollSizes(MM_TEXT, GetDocument( )->GetDocSize( ) );
   CScrollView::OnInitialUpdate();
}
```

See Also

Reference

CScrollView Class
Hierarchy Chart
CScrollView::SetScaleToFitSize
CScrollView::GetDeviceScrollSizes
CScrollView::GetTotalSize

Other Resources

CScrollView Members

CSemaphore Class

An object of class **CSemaphore** represents a "semaphore" — a synchronization object that allows a limited number of threads in one or more processes to access a Maintains a count of the number of threads currently accessing a specified resource.

```
class CSemaphore : public CSyncObject
```

Remarks

Semaphores are useful in controlling access to a shared resource that can only support a limited number of users. The current count of the **CSemaphore** object is the number of additional users allowed. When the count reaches zero, all attempts to use the resource controlled by the **CSemaphore** object will be inserted into a system queue and wait until they either time out or the count rises above 0. The maximum number of users who can access the controlled resource at one time is specified during construction of the **CSemaphore** object.

To use a **CSemaphore** object, construct the **CSemaphore** object when it is needed. Specify the name of the semaphore you wish to wait on, and that your application should initially own it. You can then access the semaphore when the constructor returns. Call CSyncObject::Unlock when you are done accessing the controlled resource.

An alternative method for using **CSemaphore** objects is to add a variable of type **CSemaphore** as a data member to the class you wish to control. During construction of the controlled object, call the constructor of the **CSemaphore** data member specifying the initial access count, maximum access count, name of the semaphore (if it will be used across process boundaries), and desired security attributes.

To access resources controlled by **CSemaphore** objects in this manner, first create a variable of either type CSingleLock or type CMultiLock in your resource's access member function. Then call the lock object's **Lock** member function (for example, CSingleLock::Lock). At this point, your thread will either gain access to the resource, wait for the resource to be released and gain access, or wait for the resource to be released and time out, failing to gain access to the resource. In any case, your resource has been accessed in a thread-safe manner. To release the resource, use the lock object's **Unlock** member function (for example, CSingleLock::Unlock), or allow the lock object to fall out of scope.

Alternatively, you can create a **CSemaphore** object stand-alone, and access it explicitly before attempting to access the controlled resource. This method, while clearer to someone reading your source code, is more prone to error.

For more information on how to use **CSemaphore** objects, see the article Multithreading: How to Use the Synchronization Classes.

Requirements

Header: afxmt.h

See Also

Reference

CSyncObject Class

Hierarchy Chart

Other Resources

CSemaphore Members

CSemaphore Members

Base Class Members

CObject Members

CSyncObject Members

Construction

CSemaphore	Constructs a **CSemaphore** object.

See Also

Reference

CSemaphore Class

Hierarchy Chart

CSemaphore Member Functions

For information about the member functions in **CSemaphore**, see CSemaphore Members.

CSemaphore::CSemaphore

Constructs a named or unnamed **CSemaphore** object.

```
CSemaphore(
   LONG lInitialCount = 1,
   LONG lMaxCount = 1,
   LPCTSTR pstrName = NULL,
   LPSECURITY_ATTRIBUTES lpsaAttributes = NULL
);
```

Parameters

lInitialCount

The initial usage count for the semaphore. Must be greater than or equal to 0, and less than or equal to *lMaxCount*.

lMaxCount

The maximum usage count for the semaphore. Must be greater than 0.

pstrName

The name of the semaphore. Must be supplied if the semaphore will be accessed across process boundaries. If **NULL,** the object will be unnamed. If the name matches an existing semaphore, the constructor builds a new **CSemaphore** object which references the semaphore of that name. If the name matches an existing synchronization object that is not a semaphore, the construction will fail.

lpsaAttributes

Security attributes for the semaphore object. For a full description of this structure, see SECURITY_ATTRIBUTES in the *Platform SDK*.

Remarks

To access or release a **CSemaphore** object, create a CMultiLock or CSingleLock object and call its Lock and Unlock member functions.

Security Note
After creating the **CSemaphore** object, use GetLastError to ensure that the mutex did not already exist. If the mutex did exist unexpectedly, it may indicate a rogue process is squatting and may be intending to use the mutex maliciously. In this case, th e recommended security-conscious procedure is to close the handle and continue as if there was a failure in creating the obj ect.

See Also

Reference

CSemaphore Class
Hierarchy Chart
CMutex Class
CEvent Class
CMultiLock Class
CSingleLock Class

Other Resources

CSemaphore Members

CSharedFile Class

The CMemFile-derived class that supports shared memory files.

```
class CSharedFile : public CMemFile
```

Remarks

Memory files behave like disk files except that the file is stored in RAM rather than on disk. A memory file is useful for fast temporary storage or for transferring raw bytes or serialized objects between independent processes.

Shared memory files differ from other memory files in that memory for them is allocated with the GlobalAlloc Windows function. The **CSharedFile** class stores data in a globally allocated memory block (created using **GlobalAlloc**), and this memory block can be shared using DDE, the Clipboard, or other OLE/COM uniform data transfer operations, for example, using **IDataObject**.

GlobalAlloc returns an **HGLOBAL** handle rather than a pointer to memory, such as the pointer returned by malloc. The **HGLOBAL** handle is needed in certain applications. For example, to put data on the Clipboard you need an **HGLOBAL** handle.

Please note that **CSharedFile** does not use memory-mapped files, and the data cannot be directly shared between processes.

CSharedFile objects can automatically allocate their own memory or you can attach your own memory block to the **CSharedFile** object by calling CSharedFile::SetHandle. In either case, memory for growing the memory file automatically is allocated in *nGrowBytes*-sized increments if *nGrowBytes* is not zero.

For more information, see the article Files in MFC and File Handling in the *Run-Time Library Reference*.

Requirements

Header: afxadv.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CMemFile Class
Hierarchy Chart
CMemFile Class
GlobalAlloc
GlobalFree
GlobalRealloc

Other Resources

CSharedFile Members

CSharedFile Members

Base Class Members

CObject Members

CFile Members

CMemFile Members

Construction

CSharedFile	Constructs a **CSharedFile** object.

Operations

Detach	Closes the shared memory file and returns the handle of its memory block.
SetHandle	Attaches the shared memory file to a memory block.

See Also

Reference

CSharedFile Class

Hierarchy Chart

CSharedFile Member Functions

For information about the member functions in **CSharedFile**, see CSharedFile Members.

CSharedFile::CSharedFile

Constructs a **CSharedFile** object and allocates memory for it.

```
CSharedFile(
   UINT nAllocFlags = GMEM_DDESHARE | GMEM_MOVEABLE,
   UINT nGrowBytes = 4096
);
```

Parameters

nAllocFlags

Flags indicating how memory is to be allocated. See GlobalAlloc for a list of valid flag values.

nGrowBytes

The memory allocation increment in bytes.

See Also

Reference

CSharedFile Class
Hierarchy Chart
CSharedFile::Detach
CSharedFile::SetHandle

Other Resources

CSharedFile Members

CSharedFile::Detach

Call this function to close the memory file and detach it from the memory block.

```
HGLOBAL Detach( );
```

Return Value

The handle of the memory block that contains the contents of the memory file.

Remarks

You can reopen it by calling SetHandle, using the handle returned by **Detach**.

See Also

Reference

CSharedFile Class
Hierarchy Chart
CSharedFile::CSharedFile
CSharedFile::SetHandle

Other Resources

CSharedFile Members

CSharedFile::SetHandle

Call this function to attach a block of global memory to the **CSharedFile** object.

```
void SetHandle(
   HGLOBAL hGlobalMemory,
   BOOL bAllowGrow = TRUE
);
```

Parameters

hGlobalMemory

Handle to the global memory to be attached to the **CSharedFile**.

bAllowGrow

Specifies whether the memory block is allowed to grow.

Remarks

If *bAllowGrow* is nonzero, the size of the memory block is increased as necessary, for example, if an attempt is made to write more bytes to the file than were allocated for the memory block.

See Also

Reference

CSharedFile Class
Hierarchy Chart
CSharedFile::CSharedFile
CSharedFile::Detach

Other Resources

CSharedFile Members

CSimpleException Class

This class is a base class for resource-critical MFC exceptions.

```
class AFX_NOVTABLE CSimpleException : public CException
```

Remarks

CSimpleException is the base class for resource-critical MFC exceptions and handles the ownership and initialization of an error message. The following classes use **CSimpleException** as their base class:

CMemoryException Class	Out-of-memory exception
CNotSupportedException Class	Requests for an unsupported operation
CResourceException Class	Windows resource not found or not creatable
CUserException Class	Exception that indicates a resource could not be found
CInvalidArgException Class	Exception that indicates an invalid argument

Because **CSimpleException** is an abstract base class, you cannot declare a **CSimpleException** object directly. Instead, you must declare derived objects such as those in the previous table. If you are declaring your own derived class, use the previous classes as a model.

For more information, see the CException Class topic and Exception Handling (MFC).

Requirements

Header: afx.h

See Also

Reference

Hierarchy Chart

CException Class

Concepts

Exception Handling in MFC

Other Resources

CSimpleException Members

CSimpleException Members

Methods

CSimpleException	The constructor.
GetErrorMessage	Provides text about an error that has occurred.

See Also

Reference

CSimpleException Class

CSimpleException Methods

For information about the methods in **CSimpleException**, see CSimpleException Members.

CSimpleException::CSimpleException

The constructor.

```
CSimpleException( );
explicit CSimpleException(
   BOOL bAutoDelete
);
```

Parameters

bAutoDelete

Specify **TRUE** if the memory for the **CSimpleException** object has been allocated on the heap. This will cause the **CSimpleException** object to be deleted when the **Delete** member function is called to delete the exception. Specify **FALSE** if the **CSimpleException** object is on the stack or is a global object. In this case, the **CSimpleException** object will not be deleted when the **Delete** member function is called.

Remarks

You would normally never need to call this constructor directly. A function that throws an exception should create an instance of a **CException**-derived class and call its constructor, or it should use one of the MFC throw functions, such as AfxThrowFileException, to throw a predefined type.

See Also

Reference

CSimpleException Class

Hierarchy Chart

Other Resources

CSimpleException Members

CSimpleException::GetErrorMessage

Call this member function to provide text about an error that has occurred.

```
virtual BOOL GetErrorMessage(
   LPTSTR lpszError,
      UINT nMaxError,
      PUNIT pnHelpContext = NULL
);
```

Parameters

lpszError

A pointer to a buffer that will receive an error message.

nMaxError

The maximum number of characters the buffer can hold, including the **NULL** terminator.

pnHelpContext

The address of a **UINT** that will receive the help context ID. If **NULL**, no ID will be returned.

Return Value

Nonzero if the function is successful; otherwise 0 if no error message text is available.

Remarks

For more information, see CException::GetErrorMessage.

See Also

Reference

CSimpleException Class

Other Resources

CSimpleException Members

CSingleDocTemplate Class

Defines a document template that implements the single document interface (SDI).

```
class CSingleDocTemplate : public CDocTemplate
```

Remarks

An SDI application uses the main frame window to display a document; only one document can be open at a time.

A document template defines the relationship between three types of classes:

- A document class, which you derive from **CDocument**.
- A view class, which displays data from the document class listed above. You can derive this class from **CView**, **CScrollView**, **CFormView**, or **CEditView**. (You can also use **CEditView** directly.)
- A frame window class, which contains the view. For an SDI document template, you can derive this class from **CFrameWnd**; if you do not need to customize the behavior of the main frame window, you can use **CFrameWnd** directly without deriving your own class.

An SDI application typically supports one type of document, so it has only one **CSingleDocTemplate** object. Only one document can be open at a time.

You don't need to call any member functions of **CSingleDocTemplate** except the constructor. The framework handles **CSingleDocTemplate** objects internally.

For more information on using **CSingleDocTemplate**, see Document Templates and the Document/View Creation Process.

Requirements

Header: afxwin.h

See Also

Tasks

DOCKTOOL Sample: Demonstrates Dockable Toolbars

Reference

CDocTemplate Class
Hierarchy Chart
CDocTemplate Class
CDocument Class
CFrameWnd Class
CMultiDocTemplate Class
CView Class
CWinApp Class

Other Resources

CSingleDocTemplate Members

CSingleDocTemplate Members

Base Class Members

CObject Members

CCmdTarget Members

CDocTemplate Members

Construction

CSingleDocTemplate	Constructs a **CSingleDocTemplate** object.

See Also

Reference

CSingleDocTemplate Class
Hierarchy Chart

CSingleDocTemplate Member Functions

For information about the member functions in **CSingleDocTemplate**, see CSingleDocTemplate Members.

CSingleDocTemplate::CSingleDocTemplate

Constructs a **CSingleDocTemplate** object.

```
CSingleDocTemplate(
   UINT nIDResource,
   CRuntimeClass* pDocClass,
   CRuntimeClass* pFrameClass,
   CRuntimeClass* pViewClass
);
```

Parameters

nIDResource

Specifies the ID of the resources used with the document type. This may include menu, icon, accelerator table, and string resources.

The string resource consists of up to seven substrings separated by the '\n' character (the '\n' character is needed as a placeholder if a substring is not included; however, trailing '\n' characters are not necessary); these substrings describe the document type. For information about the substrings, see CDocTemplate::GetDocString. This string resource is found in the application's resource file. For example:

```
// MYCALC.RC
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    IDR_MAINFRAME "MyCalc Windows Application\nSheet\nWorksheet\n Worksheets (*.myc)\n.my
c\nMyCalcSheet\n MyCalc Worksheet"
END
```

You can edit this string using the string editor; the entire string appears as a single entry in the String Editor, not as seven separate entries.

For more information about these resource types, see the String Editor.

pDocClass

Points to the **CRuntimeClass** object of the document class. This class is a **CDocument**-derived class you define to represent your documents.

pFrameClass

Points to the **CRuntimeClass** object of the frame window class. This class can be a **CFrameWnd**-derived class, or it can be **CFrameWnd** itself if you want default behavior for your main frame window.

pViewClass

Points to the **CRuntimeClass** object of the view class. This class is a **CView**-derived class you define to display your documents.

Remarks

Dynamically allocate a **CSingleDocTemplate** object and pass it to **CWinApp::AddDocTemplate** from the `InitInstance` member function of your application class.

Example

```
// Example for CSingleDocTemplate::CSingleDocTemplate.
BOOL CMyApp::InitInstance()
{
   // ...
   // Establish the document type
   // supported by the application

   AddDocTemplate( new CSingleDocTemplate( IDR_MAINFRAME,
      RUNTIME_CLASS( CSheetDoc ),
```

```
      RUNTIME_CLASS( CFrameWnd ),
      RUNTIME_CLASS( CSheetView ) ) );

   // ...
}

// Second example for CSingleDocTemplate::CSingleDocTemplate.

BOOL CYourApp::InitInstance()
{
   // Normally, an application creates a document
   // template and registers it with MFC as a part
   // of its initialization.

   // IDR_SAMPLERESOURCE is a resource ID string;
   // see the CDocTemplate class overview documentation
   // for more information on its format.

   // The next three parameters use the RUNTIME_CLASS()
   // macro to get runtime type information for the doc,
   // frame, and view classes that will be associated by
   // the template.

   CSingleDocTemplate* pDocTemplate;
   pDocTemplate = new CSingleDocTemplate(
      IDR_SAMPLERESOURCE,
      RUNTIME_CLASS(CYourDoc),
      RUNTIME_CLASS(CChildFrame),
      RUNTIME_CLASS(CYourViewClass));

   // After the following call, MFC is aware of the doc
   // template and will free it when the application is
   // shut down. The doc templates known to MFC will
   // automatically be used when CWinApp:OnFileOpen() or
   // CWinApp::OnFileNew() are called.

   AddDocTemplate(pDocTemplate);

   // You might have other initialization code ...
   // ...

   return TRUE;
}
```

See Also

Reference

CSingleDocTemplate Class
Hierarchy Chart
CDocTemplate::GetDocString
CWinApp::AddDocTemplate
CWinApp::InitInstance
CRuntimeClass Structure
RUNTIME_CLASS

Other Resources

CSingleDocTemplate Members

CSingleLock Class

Represents the access-control mechanism used in controlling access to a resource in a multithreaded program.

```
class CSingleLock
```

Remarks

CSingleLock does not have a base class.

In order to use the synchronization classes CSemaphore, CMutex, CCriticalSection, and CEvent, you must create either a **CSingleLock** or CMultiLock object to wait on and release the synchronization object. Use **CSingleLock** when you only need to wait on one object at a time. Use **CMultiLock** when there are multiple objects that you could use at a particular time.

To use a **CSingleLock** object, call its constructor inside a member function in the controlled resource's class. Then call the IsLocked member function to determine if the resource is available. If it is, continue with the remainder of the member function. If the resource is unavailable, either wait for a specified amount of time for the resource to be released, or return failure. After use of the resource is complete, either call the Unlock function if the **CSingleLock** object is to be used again, or allow the **CSingleLock** object to be destroyed.

CSingleLock objects require the presence of an object derived from CSyncObject. This is usually a data member of the controlled resource's class. For more information on how to use **CSingleLock** objects, see the article Multithreading: How to Use the Synchronization Classes.

Requirements

Header: afxmt.h

See Also

Reference

Hierarchy Chart

CMultiLock Class

Other Resources

CSingleLock Members

CSingleLock Members

Construction

CSingleLock	Constructs a **CSingleLock** object.

Methods

IsLocked	Determines if the object is locked.
Lock	Waits on a synchronization object.
Unlock	Releases a synchronization object.

See Also

Reference

CSingleLock Class

Hierarchy Chart

CSingleLock Member Functions

For information about the member functions in **CSingleLock**, see CSingleLock Members.

CSingleLock::CSingleLock

Constructs a **CSingleLock** object.

```
explicit CSingleLock(
   CSyncObject* pObject,
   BOOL bInitialLock = FALSE
);
```

Parameters

pObject

Points to the synchronization object to be accessed. Cannot be **NULL**.

bInitialLock

Specifies whether to initially attempt to access the supplied object.

Remarks

This function is generally called from within an access member function of the controlled resource.

Example

```
// m_CritSection is a data member (of type CCriticalSection)
// of an existing class that implements the resource being shared.

// Relate the synchronization object (m_CritSection) with
// our CSingleLock object.
CSingleLock singleLock(&m_CritSection);
singleLock.Lock();  // Attempt to lock the shared resource
if (singleLock.IsLocked())  // Resource has been locked
{
//...use the shared resource...

// Now that we are finished,
// unlock the resource for others.
singleLock.Unlock();
}
```

See Also

Reference

CSingleLock Class

Hierarchy Chart

Other Resources

CSingleLock Members

CSingleLock::IsLocked

Determines if the object associated with the **CSingleLock** object is nonsignaled (unavailable).

```
BOOL IsLocked( );
```

Return Value

Nonzero if the object is locked; otherwise 0.

Example

```
// m_Mutex is a data member (of type CMutex)
// of an existing class that implements the resource being shared.

// Relate the synchronization object (m_Mutex) with
// our CSingleLock object.
CSingleLock singleLock(&m_Mutex);

// Attempt to lock the shared resource
singleLock.Lock(100);    // Wait 100 ms...

// Has the resource been successfully locked?
if (singleLock.IsLocked())
{
    // We were able to lock the resource;
    // we may now work with the data associated with the mutex...

    // Now that we are finished, unlock the resource for others.
    singleLock.Unlock();
}
```

See Also

Reference

CSingleLock Class

Hierarchy Chart

Other Resources

CSingleLock Members

CSingleLock::Lock

Call this function to gain access to the resource controlled by the synchronization object supplied to the **CSingleLock** constructor.

```
BOOL Lock(
   DWORD dwTimeOut = INFINITE
);
```

Parameters

dwTimeOut

Specifies the amount of time to wait for the synchronization object to be available (signaled). If **INFINITE**, **Lock** will wait until the object is signaled before returning.

Return Value

Nonzero if the function was successful; otherwise 0.

Remarks

If the synchronization object is signaled, **Lock** will return successfully and the thread now owns the object. If the synchronization object is nonsignaled (unavailable), **Lock** will wait for the synchronization object to become signaled up to the number of milliseconds specified in the *dwTimeOut* parameter. If the synchronization object did not become signaled in the specified amount of time, **Lock** returns failure.

Example

```
// m_Mutex is a data member (of type CMutex)
// of an existing class that implements the resource being shared.

// Relate the synchronization object (m_Mutex) with
// our CSingleLock object.
CSingleLock singleLock(&m_Mutex);

// Attempt to lock the shared resource
if (singleLock.Lock(100))    // Wait 100 ms...
{
    // We were able to lock the resource;
    // we may now work with the data associated with the mutex...

    // Now that we are finished, unlock the resource for others.
    singleLock.Unlock();
}
```

See Also

Reference

CSingleLock Class

Hierarchy Chart

Other Resources

CSingleLock Members

CSingleLock::Unlock

Releases the synchronization object owned by **CSingleLock**.

```
BOOL Unlock( );
BOOL Unlock(
   LONG lCount,
   LPLONG lPrevCount = NULL
);
```

Parameters

lCount

Number of accesses to release. Must be greater than 0. If the specified amount would cause the object's count to exceed its maximum, the count is not changed and the function returns **FALSE**.

lPrevCount

Points to a variable to receive the previous count of the synchronization object. If **NULL**, the previous count is not returned.

Return Value

Nonzero if the function was successful; otherwise 0.

Remarks

This function is called by **CSingleLock**'s destructor.

If you need to release more than one access count of a semaphore, use the second form of **Unlock** and specify the number of accesses to release.

Example

```
// m_Mutex is a data member (of type CMutex)
// of an existing class that implements the resource being shared.

// Relate the synchronization object (m_Mutex) with
// our CSingleLock object.
CSingleLock singleLock(&m_Mutex);

// Attempt to lock the shared resource
if (singleLock.Lock(100))    // Wait 100 ms...
{
    // We were able to lock the resource;
    // we may now work with the data associated with the mutex...

    // Now that we are finished, unlock the resource for others.
    singleLock.Unlock();
}
```

See Also

Reference

CSingleLock Class

Hierarchy Chart

Other Resources

CSingleLock Members

CSliderCtrl Class

Provides the functionality of the Windows common slider control.

```
class CSliderCtrl : public CWnd
```

Remarks

A "slider control" (also known as a trackbar) is a window containing a slider and optional tick marks. When the user moves the slider, using either the mouse or the direction keys, the control sends notification messages to indicate the change.

Slider controls are useful when you want the user to select a discrete value or a set of consecutive values in a range. For example, you might use a slider control to allow the user to set the repeat rate of the keyboard by moving the slider to a given tick mark.

This control (and therefore the **CSliderCtrl** class) is available only to programs running under Windows 95/98 and Windows NT version 3.51 and later.

The slider moves in increments that you specify when you create it. For example, if you specify that the slider should have a range of five, the slider can only occupy six positions: a position at the left side of the slider control and one position for each increment in the range. Typically, each of these positions is identified by a tick mark.

You create a slider by using the constructor and the **Create** member function of **CSliderCtrl**. Once you have created a slider control, you can use member functions in **CSliderCtrl** to change many of its properties. Changes that you can make include setting the minimum and maximum positions for the slider, drawing tick marks, setting a selection range, and repositioning the slider.

For more information on using **CSliderCtrl**, see Controls and Using CSliderCtrl.

Requirements

Header: afxcmn.h

See Also

Tasks

CMNCTRL2 Sample: Demonstrates Common Control MFC Classes, Part 2

Reference

CWnd Class

Hierarchy Chart

CProgressCtrl Class

Other Resources

CSliderCtrl Members

CSliderCtrl Members

Base Class Members

Construction

Attributes

Operations

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Construction

Create	Creates a slider control and attaches it to a **CSliderCtrl** object.
CreatEx	Creates a slider control with the specified Windows extended styles and attaches it to a **CSliderCtrl** object.
CSliderCtrl	Constructs a **CSliderCtrl** object.

Attributes

GetBuddy	Retrieves the handle to a slider control buddy window at a given location.
GetChannelRect	Retrieves the size of the slider control's channel.
GetLineSize	Retrieves the line size of a slider control.
GetNumTics	Retrieves the number of tick marks in a slider control.
GetPageSize	Retrieves the page size of a slider control.
GetPos	Retrieves the current position of the slider.
GetRange	Retrieves the minimum and maximum positions for a slider.
GetRangeMax	Retrieves the maximum position for a slider.
GetRangeMin	Retrieves the minimum position for a slider.
GetSelection	Retrieves the range of the current selection.
GetThumbRect	Retrieves the size of the slider control's thumb.
GetTic	Retrieves the position of the specified tick mark.
GetTicArray	Retrieves the array of tick mark positions for a slider control.
GetTicPos	Retrieves the position of the specified tick mark, in client coordinates.
GetToolTips	Retrieves the handle to the tooltip control assigned to the slider control, if any.
SetBuddy	Assigns a window as the buddy window for a slider control.

SetLineSize	Sets the line size of a slider control.
SetPageSize	Sets the page size of a slider control.
SetPos	Sets the current position of the slider.
SetRange	Sets the minimum and maximum positions for a slider.
SetRangeMax	Sets the maximum position for a slider.
SetRangeMin	Sets the minimum position for a slider.
SetSelection	Sets the range of the current selection.
SetTic	Sets the position of the specified tick mark.
SetTicFreq	Sets the frequency of tick marks per slider control increment.
SetTipSide	Positions a tooltip control used by a trackbar control.
SetToolTips	Assigns a tooltip control to a slider control.

Operations

ClearSel	Clears the current selection in a slider control.
ClearTics	Removes the current tick marks from a slider control.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart

CSliderCtrl Member Functions

For information about the member functions in **CSliderCtrl**, see CSliderCtrl Members.

CSliderCtrl::ClearSel

Clears the current selection in a slider control.

```
void ClearSel(
   BOOL bRedraw = FALSE
);
```

Parameters

bRedraw

Redraw flag. If this parameter is **TRUE**, the slider is redrawn after the selection is cleared; otherwise the slider is not redrawn.

See Also

Reference

CSliderCtrl Class

Hierarchy Chart

CSliderCtrl::GetSelection

CSliderCtrl::SetSelection

Other Resources

CSliderCtrl Members

CSliderCtrl::ClearTics

Removes the current tick marks from a slider control.

```
void ClearTics(
   BOOL bRedraw = FALSE
);
```

Parameters

bRedraw

Redraw flag. If this parameter is **TRUE**, the slider is redrawn after the tick marks are cleared; otherwise the slider is not redrawn.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::GetTicArray
CSliderCtrl::GetTic
CSliderCtrl::GetNumTics

Other Resources

CSliderCtrl Members

CSliderCtrl::Create

Creates a slider control and attaches it to a **CSliderCtrl** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the slider control's style. Apply any combination of slider control styles, described in the *Platform SDK*, to the control.

rect

Specifies the slider control's size and position. It can be either a CRect object or a RECT structure.

pParentWnd

Specifies the slider control's parent window, usually a **CDialog**. It must not be **NULL**.

nID

Specifies the slider control's ID.

Return Value

Nonzero if initialization was successful; otherwise 0.

Remarks

You construct a **CSliderCtrl** in two steps. First, call the constructor, and then call **Create**, which creates the slider control and attaches it to the **CSliderCtrl** object.

Depending on the values set for *dwStyle*, the slider control can have either a vertical or horizontal orientation. It can have tick marks on either side, both sides, or neither. It can also be used to specify a range of consecutive values.

To apply extended window styles to the slider control, call CreateEx instead of **Create**.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::CSliderCtrl

Other Resources

CSliderCtrl Members

CSliderCtrl::CreateEx

Creates a control (a child window) and associates it with the **CSliderCtrl** object.

```
virtual BOOL CreateEx(
   DWORD dwExStyle,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwExStyle

Specifies the extended style of the control being created. For a list of extended Windows styles, see the *dwExStyle* parameter for CreateWindowEx in the *Platform SDK*.

dwStyle

Specifies the slider control's style. Apply any combination of slider control styles, described in the *Platform SDK*, to the control.

rect

A reference to a RECT structure describing the size and position of the window to be created, in client coordinates of *pParentWnd*.

pParentWnd

A pointer to the window that is the control's parent.

nID

The control's child-window ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Use **CreateEx** instead of Create to apply extended Windows styles, specified by the Windows extended style preface **WS_EX_**.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::CSliderCtrl

Other Resources

CSliderCtrl Members

CSliderCtrl::CSliderCtrl

Constructs a **CSliderCtrl** object.

```
CSliderCtrl( );
```

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::Create
CSliderCtrl::CreateEx

Other Resources

CSliderCtrl Members

CSliderCtrl::GetBuddy

Retrieves the handle to a slider control buddy window at a given location.

```
CWnd* GetBuddy(
   BOOL fLocation = TRUE
) const;
```

Parameters

fLocation

Value indicating which buddy window handle will be retrieved, by relative location. This value can be one of the following:

- **TRUE** Retrieves the handle to the buddy to the left of the slider. If the slider control uses the **TBS_VERT** style, the message will retrieve the buddy above the slider.
- **FALSE** Retrieves the handle to the buddy to the right of the slider. If the slider control uses the **TBS_VERT** style, the message will retrieve the buddy below the slider.

Return Value

A pointer to a CWnd object that is the buddy window at the location specified by *fLocation*, or **NULL** if no buddy window exists at that location.

Remarks

This member function implements the behavior of the Win32 message TBM_GETBUDDY, as described in the *Platform SDK*. For a description of the slider control styles, see Trackbar Control Styles in the *Platform SDK*.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::SetBuddy

Other Resources

CSliderCtrl Members

CSliderCtrl::GetChannelRect

Retrieves the size and position of the bounding rectangle for a slider control's channel.

```
void GetChannelRect(
   LPRECT lprc
) const;
```

Parameters

lprc

A pointer to a CRect object that contains the size and position of the channel's bounding rectangle when the function returns.

Remarks

The channel is the area over which the slider moves and which contains the highlight when a range is selected.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::GetThumbRect

Other Resources

CSliderCtrl Members

CSliderCtrl::GetLineSize

Retrieves the size of the line for a slider control.

```
int GetLineSize( ) const;
```

Return Value

The size of a line for the slider control.

Remarks

The line size affects how much the slider moves for the **TB_LINEUP** and **TB_LINEDOWN** notifications. The default setting for the line size is 1.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::SetLineSize
CSliderCtrl::GetPageSize

Other Resources

CSliderCtrl Members

CSliderCtrl::GetNumTics

Retrieves the number of tick marks in a slider control.

```
UINT GetNumTics( ) const;
```

Return Value

The number of tick marks in the slider control.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::GetTicArray
CSliderCtrl::GetTic
CSliderCtrl::GetTicPos
CSliderCtrl::SetTicFreq
CSliderCtrl::ClearTics

Other Resources

CSliderCtrl Members

CSliderCtrl::GetPageSize

Retrieves the size of the page for a slider control.

```
int GetPageSize( ) const;
```

Return Value

The size of a page for the slider control.

Remarks

The page size affects how much the slider moves for the **TB_PAGEUP** and **TB_PAGEDOWN** notifications.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::GetLineSize
CSliderCtrl::SetPageSize

Other Resources

CSliderCtrl Members

CSliderCtrl::GetPos

Retrieves the current position of the slider in a slider control.

```
int GetPos( ) const;
```

Return Value

The current position.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::SetPos
CSliderCtrl::GetTicPos

Other Resources

CSliderCtrl Members

CSliderCtrl::GetRange

Retrieves the maximum and minimum positions for the slider in a slider control.

```
void GetRange(
   int& nMin,
   int& nMax
) const;
```

Parameters

nMin

Reference to an integer that receives the minimum position.

nMax

Reference to an integer that receives the maximum position.

Remarks

This function copies the values into the integers referenced by *nMin* and *nMax*.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::GetRangeMin
CSliderCtrl::GetRangeMax
CSliderCtrl::SetRange

Other Resources

CSliderCtrl Members

CSliderCtrl::GetRangeMax

Retrieves the maximum position for the slider in a slider control.

```
int GetRangeMax( ) const;
```

Return Value

The control's maximum position.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::GetRangeMin
CSliderCtrl::GetRange
CSliderCtrl::SetRange

Other Resources

CSliderCtrl Members

CSliderCtrl::GetRangeMin

Retrieves the minimum position for the slider in a slider control.

```
int GetRangeMin( ) const;
```

Return Value

The control's minimum position.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::GetRange
CSliderCtrl::GetRangeMax
CSliderCtrl::SetRange

Other Resources

CSliderCtrl Members

CSliderCtrl::GetSelection

Retrieves the starting and ending positions of the current selection in a slider control.

```
void GetSelection(
   int& nMin,
   int& nMax
) const;
```

Parameters

nMin

Reference to an integer that receives the starting position of the current selection.

nMax

Reference to an integer that receives the ending position of the current selection.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::SetSelection
CSliderCtrl::ClearSel

Other Resources

CSliderCtrl Members

CSliderCtrl::GetThumbRect

Retrieves the size and position of the bounding rectangle for the slider (thumb) in a slider control.

```
void GetThumbRect(
   LPRECT lprc
) const;
```

Parameters

lprc

A pointer to a **CRect** object that contains the bounding rectangle for the slider when the function returns.

See Also

Reference

CSliderCtrl Class

Hierarchy Chart

CSliderCtrl::GetChannelRect

Other Resources

CSliderCtrl Members

CSliderCtrl::GetTic

Retrieves the position of a tick mark in a slider control.

```
int GetTic(
   int nTic
) const;
```

Parameters

nTic

Zero-based index identifying a tick mark.

Return Value

The position of the specified tick mark or – 1 if *nTic* does not specify a valid index.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::SetTic
CSliderCtrl::GetTicArray
CSliderCtrl::GetTicPos
CSliderCtrl::SetTicFreq
CSliderCtrl::ClearTics

Other Resources

CSliderCtrl Members

CSliderCtrl::GetTicArray

Retrieves the address of the array containing the positions of tick marks for a slider control.

```
DWORD* GetTicArray( ) const;
```

Return Value

The address of the array containing tick mark positions for the slider control.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::SetTic
CSliderCtrl::GetTic
CSliderCtrl::GetTicPos
CSliderCtrl::SetTicFreq
CSliderCtrl::ClearTics

Other Resources

CSliderCtrl Members

CSliderCtrl::GetTicPos

Retrieves the current physical position of a tick mark in a slider control.

```
int GetTicPos(
   int nTic
) const;
```

Parameters

nTic

Zero-based index identifying a tick mark.

Return Value

The physical position, in client coordinates, of the specified tick mark or – 1 if *nTic* does not specify a valid index.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::SetTic
CSliderCtrl::GetTic
CSliderCtrl::SetTicFreq
CSliderCtrl::ClearTics

Other Resources

CSliderCtrl Members

CSliderCtrl::GetToolTips

Retrieves the handle to the tooltip control assigned to the slider control, if any.

```
CToolTipCtrl* GetToolTips( ) const;
```

Return Value

A pointer to a CToolTipCtrl object, or **NULL** if tooltips are not in use. If the slider control does not use the **TBS_TOOLTIPS** style, the return value is **NULL**.

Remarks

This member function implements the behavior of the Win32 message TBM_GETTOOLTIPS, as described in the *Platform SDK*. Note that this member function returns a **CToolTipCtrl** object instead of a handle to a control.

For a description of the slider control styles, see Trackbar Control Styles in the *Platform SDK*.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::SetToolTips

Other Resources

CSliderCtrl Members

CSliderCtrl::SetBuddy

Assigns a window as the buddy window for a slider control.

```
CWnd* SetBuddy(
   CWnd* pWndBuddy,
   BOOL fLocation = TRUE
);
```

Parameters

pWndBuddy

A pointer to a **CWnd** object that will be set as the slider control's buddy.

fLocation

Value specifying the location at which to display the buddy window. This value can be one of the following:

- **TRUE** The buddy will appear to the left of the trackbar if the trackbar control uses the **TBS_HORZ** style. If the trackbar uses the **TBS_VERT** style, the buddy appears above the trackbar control.
- **FALSE** The buddy will appear to the right of the trackbar if the trackbar control uses the **TBS_HORZ** style. If the trackbar uses the **TBS_VERT** style, the buddy appears below the trackbar control.

Return Value

A pointer to a CWnd object that was previously assigned to the slider control at that location.

Remarks

This member function implements the behavior of the Win32 message TBM_SETBUDDY, as described in the *Platform SDK*. Note that this member function uses pointers to **CWnd** objects, rather than window handles for both its return value and parameter.

For a description of the slider control styles, see Trackbar Control Styles in the *Platform SDK*.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::GetBuddy

Other Resources

CSliderCtrl Members

CSliderCtrl::SetLineSize

Sets the size of the line for a slider control.

```
int SetLineSize(
   int nSize
);
```

Parameters

nSize

The new line size of the slider control.

Return Value

The previous line size.

Remarks

The line size affects how much the slider moves for the **TB_LINEUP** and **TB_LINEDOWN** notifications.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::GetLineSize
CSliderCtrl::SetPageSize

Other Resources

CSliderCtrl Members

CSliderCtrl::SetPageSize

Sets the size of the page for a slider control.

```
int SetPageSize(
   int nSize
);
```

Parameters

nSize

The new page size of the slider control.

Return Value

The previous page size.

Remarks

The page size affects how much the slider moves for the **TB_PAGEUP** and **TB_PAGEDOWN** notifications.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::GetPageSize
CSliderCtrl::GetLineSize

Other Resources

CSliderCtrl Members

CSliderCtrl::SetPos

Sets the current position of the slider in a slider control.

```
void SetPos(
   int nPos
);
```

Parameters

nPos

Specifies the new slider position.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::GetPos
CSliderCtrl::SetTic

Other Resources

CSliderCtrl Members

CSliderCtrl::SetRange

Sets the range (minimum and maximum positions) for the slider in a slider control.

```
void SetRange(
   int nMin,
   int nMax,
   BOOL bRedraw = FALSE
);
```

Parameters

nMin

Minimum position for the slider.

nMax

Maximum position for the slider.

bRedraw

The redraw flag. If this parameter is **TRUE**, the slider is redrawn after the range is set; otherwise the slider is not redrawn.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::GetRange
CSliderCtrl::SetRangeMax
CSliderCtrl::SetRangeMin

Other Resources

CSliderCtrl Members

CSliderCtrl::SetRangeMax

Sets the maximum range for the slider in a slider control.

```
void SetRangeMax(
   int nMax,
   BOOL bRedraw = FALSE
);
```

Parameters

nMax

Maximum position for the slider.

bRedraw

The redraw flag. If this parameter is **TRUE**, the slider is redrawn after the range is set; otherwise the slider is not redrawn.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::SetRange
CSliderCtrl::GetRangeMax
CSliderCtrl::SetRangeMin

Other Resources

CSliderCtrl Members

CSliderCtrl::SetRangeMin

Sets the minimum range for the slider in a slider control.

```
void SetRangeMin(
   int nMin,
   BOOL bRedraw = FALSE
);
```

Parameters

nMin

Minimum position for the slider.

bRedraw

The redraw flag. If this parameter is **TRUE**, the slider is redrawn after the range is set; otherwise the slider is not redrawn.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::SetRange
CSliderCtrl::GetRangeMin
CSliderCtrl::SetRangeMax

Other Resources

CSliderCtrl Members

CSliderCtrl::SetSelection

Sets the starting and ending positions for the current selection in a slider control.

```
void SetSelection(
   int nMin,
   int nMax
);
```

Parameters

nMin

Starting position for the slider.

nMax

Ending position for the slider.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::GetSelection
CSliderCtrl::ClearSel

Other Resources

CSliderCtrl Members

CSliderCtrl::SetTic

Sets the position of a tick mark in a slider control.

```
BOOL SetTic(
   int nTic
);
```

Parameters

nTic

Position of the tick mark. This parameter must specify a positive value.

Return Value

Nonzero if the tick mark is set; otherwise 0.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::GetTic
CSliderCtrl::GetTicArray
CSliderCtrl::GetTicPos
CSliderCtrl::SetTicFreq
CSliderCtrl::ClearTics

Other Resources

CSliderCtrl Members

CSliderCtrl::SetTicFreq

Sets the frequency with which tick marks are displayed in a slider.

```
void SetTicFreq(
   int nFreq
);
```

Parameters

nFreq

Frequency of the tick marks.

Remarks

For example, if the frequency is set to 2, a tick mark is displayed for every other increment in the slider's range. The default setting for the frequency is 1 (that is, every increment in the range is associated with a tick mark).

You must create the control with the **TBS_AUTOTICKS** style to use this function. For more information, see CSliderCtrl::Create.

See Also

Reference

CSliderCtrl Class
Hierarchy Chart
CSliderCtrl::Create
CSliderCtrl::SetTic
CSliderCtrl::GetTicArray

Other Resources

CSliderCtrl Members

CSliderCtrl::SetTipSide

Positions a tooltip control used by a trackbar control.

```
int SetTipSide(
   int nLocation
);
```

Parameters

nLocation

Value representing the location at which to display the tooltip control. For a list of possible values, see the Win32 message TBM_SETTIPSIDE, as described in the *Platform SDK*.

Return Value

A value that represents the tooltip control's previous location. The return value equals one of the possible values for *nLocation*.

Remarks

This member function implements the behavior of the Win32 message **TBM_SETTIPSIDE**, as described in the *Platform SDK*. Slider controls that use the **TBS_TOOLTIPS** style display tooltips. For a description of the slider control styles, see Trackbar Control Styles in the *Platform SDK*.

See Also

Reference

CSliderCtrl Class

Hierarchy Chart

Other Resources

CSliderCtrl Members

CSliderCtrl::SetToolTips

Assigns a tooltip control to a slider control.

```
void SetToolTips(
   CToolTipCtrl* pWndTip
);
```

Parameters

pWndTip

A pointer to a CToolTipCtrl object containing the tooltips to use with the slider control.

Remarks

This member function implements the behavior of the Win32 message TBM_SETTOOLTIPS, as described in the *Platform SDK*. When a slider control is created with the **TBS_TOOLTIPS** style, it creates a default tooltip control that appears next to the slider, displaying the slider's current position. For a description of the slider control styles, see Trackbar Control Styles in the *Platform SDK*.

See Also

Reference

CSliderCtrl Class

Hierarchy Chart

CSliderCtrl::GetToolTips

Other Resources

CSliderCtrl Members

CSocket Class

Derives from **CAsyncSocket** and inherits its encapsulation of the Represents a higher level of abstraction of the Windows Sockets API than that of a **CAsyncSocket** object.

```
class CSocket : public CAsyncSocket
```

Remarks

CSocket works with classes **CSocketFile** and **CArchive** to manage the sending and receiving of data.

A **CSocket** object also provides blocking, which is essential to the synchronous operation of **CArchive**. Blocking functions, such as **Receive**, **Send**, **ReceiveFrom**, **SendTo**, and **Accept** (all inherited from **CAsyncSocket**), do not return a **WSAEWOULDBLOCK** error in **CSocket**. Instead, these functions wait until the operation completes. Additionally, the original call will terminate with the error **WSAEINTR** if **CancelBlockingCall** is called while one of these functions is blocking.

To use a **CSocket** object, call the constructor, then call **Create** to create the underlying **SOCKET** handle (type **SOCKET**). The default parameters of **Create** create a stream socket, but if you are not using the socket with a **CArchive** object, you can specify a parameter to create a datagram socket instead, or bind to a specific port to create a server socket. Connect to a client socket using **Connect** on the client side and **Accept** on the server side. Then create a **CSocketFile** object and associate it to the **CSocket** object in the **CSocketFile** constructor. Next, create a **CArchive** object for sending and one for receiving data (as needed), then associate them with the **CSocketFile** object in the **CArchive** constructor. When communications are complete, destroy the **CArchive**, **CSocketFile**, and **CSocket** objects. The **SOCKET** data type is described in the article Windows Sockets: Background.

When you use **CArchive** with **CSocketFile** and **CSocket**, you might encounter a situation where **CSocket::Receive** enters a loop (by **PumpMessages(FD_READ)**) waiting for the requested amount of bytes. This is because Windows sockets allow only one recv call per FD_READ notification, but **CSocketFile** and **CSocket** allow multiple recv calls per FD_READ. If you get an FD_READ when there is no data to read, the application hangs. If you never get another FD_READ, the application stops communicating over the socket.

You can resolve this problem as follows. In the **OnReceive** method of your socket class, call **CAsyncSocket::IOCtl(FIONREAD, ...)** before you call the **Serialize** method of your message class when the expected data to be read from the socket exceeds the size of one TCP packet (maximum transmission unit of the network medium, usually at least 1096 bytes). If the size of the available data is less than needed, wait for all the data to be received and only then start the read operation.

In the following example, *dwExpected* is the approximate number of bytes that the user expects to receive. It is assumed that you declare it elsewhere in your code.

```
void CChatSocket::OnReceive(int nErrorCode)
{
   CSocket::OnReceive(nErrorCode);

   DWORD dwReceived;

   if (IOCtl(FIONREAD, &dwReceived))
   {
      if (dwReceived >= dwExpected)   // Process only if you have enough data
         m_pDoc->ProcessPendingRead();
   }
   else
   {
      // Error handling here
   }
}
```

For more information, see Windows Sockets in MFC, Windows Sockets: Using Sockets with Archives, Windows Sockets: How Sockets with Archives Work, Windows Sockets: Sequence of Operations, Windows Sockets: Example of Sockets Using Archives.

Requirements

Header: afxsock.h

See Also

Reference

CAsyncSocket Class

Hierarchy Chart

CAsyncSocket Class

CSocketFile Class

Other Resources

CSocket Members

CSocket Members

Base Class Members

CObject Members

CAsyncSocket Members

Construction

Create	Creates a socket.
CSocket	Constructs a **CSocket** object.

Attributes

Attach	Attaches a **SOCKET** handle to a **CSocket** object.
FromHandle	Returns a pointer to a **CSocket** object, given a **SOCKET** handle.
IsBlocking	Determines whether a blocking call is in progress.

Operations

CancelBlockingCall	Cancels a blocking call that is currently in progress.

Overridables

OnMessagePending	Called to process pending messages while waiting for a blocking call to complete.

Operators

operator =	Copies the source data into an existing **CSocket** object.

See Also

Reference

CSocket Class

Hierarchy Chart

CSocket Member Functions

For information about the member functions in **CSocket**, see CSocket Members.

CSocket::Attach

Call this member function to attach the *hSocket* handle to a **CSocket** object.

```
BOOL Attach(
   SOCKET hSocket
);
```

Parameters

hSocket

Contains a handle to a socket.

Return Value

Nonzero if the function is successful.

Remarks

The **SOCKET** handle is stored in the object's m_hSocket data member.

For more information, see Windows Sockets: Using Sockets with Archives.

Example

```
// ...
class CSockThread : public CWinThread
{
// ... Other function and member declarations
protected:
  CSocket m_sConnected;
};

SOCKET hConnected;

BOOL CSockThread::InitInstance()
{
   // Attach the socket object to the socket handle
   // in the context of this thread.
   m_sConnected.Attach(hConnected);

   return TRUE;
}

// This listening socket has been constructed
// in the primary thread.
void CListeningSocket::OnAccept(int nErrorCode)
{
   // This CSocket object is used just temporarily
   // to accept the incoming connection.
   CSocket sConnected;
   Accept(sConnected);

   // Detach the newly accepted socket and save
   // the SOCKET handle.
   hConnected = sConnected.Detach();

   // After detaching it, it should no longer be
   // used in the context of this thread.

   // Start the other thread.
   AfxBeginThread(RUNTIME_CLASS(CSockThread));
}
```

See Also

Reference

CSocket Class

Hierarchy Chart

CAsyncSocket::Attach

Other Resources

CSocket Members

CSocket::CancelBlockingCall

Call this member function to cancel a blocking call currently in progress.

```
void CancelBlockingCall( );
```

Remarks

This function cancels any outstanding blocking operation for this socket. The original blocking call will terminate as soon as possible with the error **WSAEINTR**.

In the case of a blocking **Connect** operation, the Windows Sockets implementation will terminate the blocking call as soon as possible, but it may not be possible for the socket resources to be released until the connection has completed (and then been reset) or timed out. This is likely to be noticeable only if the application immediately tries to open a new socket (if no sockets are available), or to connect to the same peer.

Canceling any operation other than **Accept** can leave the socket in an indeterminate state. If an application cancels a blocking operation on a socket, the only operation that the application can depend on being able to perform on the socket is a call to **Close**, although other operations may work on some Windows Sockets implementations. If you desire maximum portability for your application, you must be careful not to depend on performing operations after a cancel.

For more information, see Windows Sockets: Using Sockets with Archives.

See Also

Reference

CSocket Class
Hierarchy Chart
CAsyncSocket::Accept
CAsyncSocket::Close
CAsyncSocket::Connect
CSocket::IsBlocking
WSASetBlockingHook

Other Resources

CSocket Members

CSocket::Create

Call the **Create** member function after constructing a socket object to create the Windows socket and attach it.

```
BOOL Create(
   UINT nSocketPort = 0,
   int nSocketType = SOCK_STREAM,
   LPCTSTR lpszSocketAddress = NULL
);
```

Parameters

nSocketPort

A particular port to be used with the socket, or 0 if you want MFC to select a port.

nSocketType

SOCK_STREAM or **SOCK_DGRAM**.

lpszSocketAddress

A pointer to a string containing the network address of the connected socket, a dotted number such as "128.56.22.8". Passing the **NULL** string for this parameter indicates the **CSocket** instance should listen for client activity on all network interfaces.

Return Value

Nonzero if the function is successful; otherwise 0, and a specific error code can be retrieved by calling **GetLastError**.

Remarks

Create then calls **Bind** to bind the socket to the specified address. The following socket types are supported:

- **SOCK_STREAM** Provides sequenced, reliable, two-way, connection-based byte streams. Uses Transmission Control Protocol (TCP) for the Internet address family.
- **SOCK_DGRAM** Supports datagrams, which are connectionless, unreliable buffers of a fixed (typically small) maximum length. Uses User Datagram Protocol (UDP) for the Internet address family. To use this option, you must not use the socket with a **CArchive** object.

Note
The **Accept** member function takes a reference to a new, empty **CSocket** object as its parameter. You must construct t his object before you call **Accept**. Keep in mind that if this socket object goes out of scope, the connection closes. Do n ot call **Create** for this new socket object.

For more information about stream and datagram sockets, see the articles Windows Sockets: Background, Windows Sockets: Ports and Socket Addresses, and Windows Sockets: Using Sockets with Archives.

See Also

Reference

CSocket Class
Hierarchy Chart
CAsyncSocket::Create
CAsyncSocket::Bind

Other Resources

CSocket Members

CSocket::CSocket

Constructs a **CSocket** object.

```
CSocket( );
```

Remarks

After construction, you must call the **Create** member function.

For more information, see Windows Sockets: Using Sockets with Archives.

See Also

Reference

CSocket Class
Hierarchy Chart
CAsyncSocket::Create

Other Resources

CSocket Members

CSocket::FromHandle

Returns a pointer to a **CSocket** object.

```
static CSocket* PASCAL FromHandle(
   SOCKET hSocket
);
```

Parameters

hSocket

Contains a handle to a socket.

Return Value

A pointer to a **CSocket** object, or **NULL** if there is no **CSocket** object attached to *hSocket*.

Remarks

When given a **SOCKET** handle, if a **CSocket** object is not attached to the handle, the member function returns **NULL** and does not create a temporary object.

For more information, see Windows Sockets: Using Sockets with Archives.

See Also

Reference

CSocket Class
Hierarchy Chart
CAsyncSocket::FromHandle

Other Resources

CSocket Members

CSocket::IsBlocking

Call this member function to determine if a blocking call is in progress.

```
BOOL IsBlocking( );
```

Return Value

Nonzero if the socket is blocking; otherwise 0.

Remarks

For more information, see Windows Sockets: Using Sockets with Archives.

See Also

Reference

CSocket Class
Hierarchy Chart
CSocket::CancelBlockingCall

Other Resources

CSocket Members

CSocket::OnMessagePending

Override this member function to look for particular messages from Windows and respond to them in your socket.

```
virtual BOOL OnMessagePending( );
```

Return Value

Nonzero if the message was handled; otherwise 0.

Remarks

This is an advanced overridable.

The framework calls **OnMessagePending** while the socket is pumping Windows messages to give you an opportunity to deal with messages of interest to your application. For examples of how you might use **OnMessagePending**, see the article Windows Sockets: Deriving from Socket Classes.

For more information, see Windows Sockets: Using Sockets with Archives.

See Also

Reference

CSocket Class
Hierarchy Chart
CSocket::CancelBlockingCall
CSocket::IsBlocking

Other Resources

CSocket Members

CSocket Operators

For information about the operators in **CSocket**, see CSocket Members.

CSocket::operator =

Copies the source data into an existing **CSocket** object.

```
void operator=(
   const CSocket& rSrc
);
```

Parameters

rSrc

A pointer to an existing **CSocket** object.

See Also

Reference

CSocket Class

Hierarchy Chart

Other Resources

CSocket Members

CSocketFile Class

A **CFile** object used for sending and receiving data across a network via Windows Sockets.

```
class CSocketFile : public CFile
```

Remarks

You can attach the **CSocketFile** object to a **CSocket** object for this purpose. You also can, and usually do, attach the **CSocketFile** object to a **CArchive** object to simplify sending and receiving data using MFC serialization.

To serialize (send) data, you insert it into the archive, which calls **CSocketFile** member functions to write data to the **CSocket** object. To deserialize (receive) data, you extract from the archive. This causes the archive to call **CSocketFile** member functions to read data from the **CSocket** object.

Tip

Besides using **CSocketFile** as described here, you can use it as a stand-alone file object, just as you can with **CFile**, its base class. You can also use **CSocketFile** with any archive-based MFC serialization functions. Because **CSocketFile** does not support all of **CFile**'s functionality, some default MFC serialize functions are not compatible with **CSocketFile**. This is particularly true of the **CEditView** class. You should not try to serialize **CEditView** data through a **CArchive** object attached to a **CSocketFile** object using **CEditView::SerializeRaw**; use **CEditView::Serialize** instead. The **SerializeRaw** function expects the file object to have functions, such as **Seek**, that **CSocketFile** does not have.

When you use **CArchive** with **CSocketFile** and **CSocket**, you might encounter a situation where **CSocket::Receive** enters a loop (by **PumpMessages(FD_READ)**) waiting for the requested amount of bytes. This is because Windows sockets allow only one recv call per FD_READ notification, but **CSocketFile** and **CSocket** allow multiple recv calls per FD_READ. If you get an FD_READ when there is no data to read, the application hangs. If you never get another FD_READ, the application stops communicating over the socket.

You can resolve this problem as follows. In the **OnReceive** method of your socket class, call **CAsyncSocket::IOCtl(FIONREAD, ...)** before you call the **Serialize** method of your message class when the expected data to be read from the socket exceeds the size of one TCP packet (maximum transmission unit of the network medium, usually at least 1096 bytes). If the size of the available data is less than needed, wait for all the data to be received and only then start the read operation.

In the following example, *dwExpected* is the approximate number of bytes that the user expects to receive. It is assumed that you declare it elsewhere in your code.

```
void CChatSocket::OnReceive(int nErrorCode)
{
   CSocket::OnReceive(nErrorCode);

   DWORD dwReceived;

   if (IOCtl(FIONREAD, &dwReceived))
   {
      if (dwReceived >= dwExpected)   // Process only if you have enough data
         m_pDoc->ProcessPendingRead();
   }
   else
   {
      // Error handling here
   }
}
```

For more information, see Windows Sockets in MFC, Windows Sockets: Using Sockets with Archives, as well as Windows Sockets 2 API.

Requirements

Header: afxsock.h

See Also

Reference

CFile Class

Hierarchy Chart

CAsyncSocket Class

CSocket Class

Other Resources

CSocketFile Members

CSocketFile Members

Base Class Members

CObject Members

CFile Members

Construction

CSocketFile	Constructs a **CSocketFile** object.

See Also

Reference

CSocketFile Class

Hierarchy Chart

CSocketFile Member Functions

For information about the member functions in **CSocketFile**, see CSocketFile Members.

CSocketFile::CSocketFile

Constructs a **CSocketFile** object.

```
explicit CSocketFile(
   CSocket* pSocket,
   BOOL bArchiveCompatible = TRUE
);
```

Parameters

pSocket

The socket to attach to the **CSocketFile** object.

bArchiveCompatible

Specifies whether the file object is for use with a **CArchive** object. Pass **FALSE** only if you want to use the **CSocketFile** object in a stand-alone manner as you would a stand-alone **CFile** object, with certain limitations. This flag changes how the **CArchive** object attached to the **CSocketFile** object manages its buffer for reading.

Remarks

The object's destructor disassociates itself from the socket object when the object goes out of scope or is deleted.

Note

A **CSocketFile** can also be used as a (limited) file without a **CArchive** object. By default, the **CSocketFile** constructor's *bArchiveCompatible* parameter is **TRUE**. This specifies that the file object is for use with an archive. To use the file object without an archive, pass **FALSE** in the *bArchiveCompatible* parameter.

In its "archive compatible" mode, a **CSocketFile** object provides better performance and reduces the danger of a "deadlock." A deadlock occurs when both the sending and receiving sockets are waiting on each other, or for a common resource. This situation might occur if the **CArchive** object worked with the **CSocketFile** the way it does with a **CFile** object. With **CFile**, the archive can assume that if it receives fewer bytes than it requested, the end of file has been reached.

With **CSocketFile**, however, data is message based; the buffer can contain multiple messages, so receiving fewer than the number of bytes requested does not imply end of file. The application does not block in this case as it might with **CFile**, and it can continue reading messages from the buffer until the buffer is empty. The CArchive::IsBufferEmpty function is useful for monitoring the state of the archive's buffer in such a case.

For more information on the use of **CSocketFile**, see the articles Windows Sockets: Using Sockets with Archives and Windows Sockets: Example of Sockets Using Archives.

See Also

Reference

CSocketFile Class
Hierarchy Chart
CFile::CFile
CFile::Read

Other Resources

CSocketFile Members

CSpinButtonCtrl Class

Provides the functionality of the Windows common spin button control.

```
class CSpinButtonCtrl : public CWnd
```

Remarks

A "spin button control" (also known as an up-down control) is a pair of arrow buttons that the user can click to increment or decrement a value, such as a scroll position or a number displayed in a companion control. The value associated with a spin button control is called its current position. A spin button control is most often used with a companion control, called a "buddy window."

This control (and therefore the **CSpinButtonCtrl** class) is available only to programs running under Windows 95/98 and Windows NT version 3.51 and later.

To the user, a spin button control and its buddy window often look like a single control. You can specify that a spin button control automatically position itself next to its buddy window, and that it automatically set the caption of the buddy window to its current position. You can use a spin button control with an edit control to prompt the user for numeric input.

Clicking the up arrow moves the current position toward the maximum, and clicking the down arrow moves the current position toward the minimum. By default, the minimum is 100 and the maximum is 0. Any time the minimum setting is greater than the maximum setting (for example, when the default settings are used), clicking the up arrow decreases the position value and clicking the down arrow increases it.

A spin button control without a buddy window functions as a sort of simplified scroll bar. For example, a tab control sometimes displays a spin button control to enable the user to scroll additional tabs into view.

For more information on using **CSpinButtonCtrl**, see Controls and Using CSpinButtonCtrl.

Requirements

Header: afxcmn.h

See Also

Tasks

CMNCTRL2 Sample: Demonstrates Common Control MFC Classes, Part 2

Reference

CWnd Class

Hierarchy Chart

CSliderCtrl Class

Other Resources

CSpinButtonCtrl Members

CSpinButtonCtrl Members

Base Class Members

COBJECT Members

CCmdTarget Members

CWnd Members

Construction

Create	Creates a spin button control and attaches it to a **CSpinButtonCtrl** object.
CreateEx	Creates a spin button control with the specified Windows extended styles and attaches it to a **CSpinButtonCtrl** object.
CSpinButtonCtrl	Constructs a **CSpinButtonCtrl** object.

Attributes

GetAccel	Retrieves acceleration information for a spin button control.
GetBase	Retrieves the current base for a spin button control.
GetBuddy	Retrieves a pointer to the current buddy window.
GetPos	Retrieves the current position of a spin button control.
GetPos32	Retrieves the 32-bit position of a spin button control.
GetRange	Retrieves the upper and lower limits (range) for a spin button control.
GetRange32	Retrieves the 32-bit range for a spin button control.
SetAccel	Sets the acceleration for a spin button control.
SetBase	Sets the base for a spin button control.
SetBuddy	Sets the buddy window for a spin button control.
SetPos	Sets the current position for the control.
SetPos32	Sets the 32-bit position for a spin button control.
SetRange	Sets the upper and lower limits (range) for a spin button control.
SetRange32	Sets the 32-bit range for a spin button control.

See Also

Reference

CSpinButtonCtrl Class

Hierarchy Chart

CSpinButtonCtrl Member Functions

For information about the member functions in **CSpinButtonCtrl**, see CSpinButtonCtrl Members.

CSpinButtonCtrl::Create

Creates a spin button control and attaches it to a **CSpinButtonCtrl** object..

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the spin button control's style. Apply any combination of spin button control styles to the control. These styles are described in Up-Down Control Styles in the *Platform SDK*.

rect

Specifies the spin button control's size and position. It can be either a CRect object or a RECT structure

pParentWnd

A pointer to the spin button control's parent window, usually a **CDialog**. It must not be **NULL.**

nID

Specifies the spin button control's ID.

Return Value

Nonzero if initialization was successful; otherwise 0.

Remarks

You construct a **CSpinButtonCtrl** object in two steps First, call the constructor, and then call **Create**, which creates the spin button control and attaches it to the **CSpinButtonCtrl** object.

To create a spin button control with extended window styles, call CSpinButtonCtrl::CreateEx instead of **Create**.

See Also

Reference

CSpinButtonCtrl Class
Hierarchy Chart
CSpinButtonCtrl::CSpinButtonCtrl

Other Resources

CSpinButtonCtrl Members

CSpinButtonCtrl::CreateEx

Creates a control (a child window) and associates it with the **CSpinButtonCtrl** object.

```
virtual BOOL CreateEx(
   DWORD dwExStyle,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwExStyle

Specifies the extended style of the control being created. For a list of extended windows styles, see the *dwExStyle* parameter for CreateWindowEx in the *Platform SDK*.

dwStyle

Specifies the spin button control's style. Apply any combination of spin button control styles to the control. These styles are described in Up-Down Control Styles in the *Platform SDK*.

rect

A reference to a RECT structure describing the size and position of the window to be created, in client coordinates of *pParentWnd*.

pParentWnd

A pointer to the window that is the control's parent.

nID

The control's child-window ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Use **CreateEx** instead of Create to apply extended Windows styles, specified by the Windows extended style preface **WS_EX_**.

See Also

Reference

CSpinButtonCtrl Class
Hierarchy Chart
CSpinButtonCtrl::CSpinButtonCtrl

Other Resources

CSpinButtonCtrl Members

CSpinButtonCtrl::CSpinButtonCtrl

Constructs a **CSpinButtonCtrl** object.

```
CSpinButtonCtrl( );
```

See Also

Reference

CSpinButtonCtrl Class
Hierarchy Chart
CSpinButtonCtrl::Create
CSpinButtonCtrl::CreateEx

Other Resources

CSpinButtonCtrl Members

CSpinButtonCtrl::GetAccel

Retrieves acceleration information for a spin button control.

```
UINT GetAccel(
   int nAccel,
   UDACCEL* pAccel
) const;
```

Parameters

nAccel

Number of elements in the array specified by *pAccel*.

pAccel

Pointer to an array of UDACCEL structures that receives acceleration information.

Return Value

Number of accelerator structures retrieved.

See Also

Reference

CSpinButtonCtrl Class
Hierarchy Chart
CSpinButtonCtrl::SetAccel

Other Resources

CSpinButtonCtrl Members

CSpinButtonCtrl::GetBase

Retrieves the current base for a spin button control.

```
UINT GetBase( ) const;
```

Return Value

The current base value.

See Also

Reference

CSpinButtonCtrl Class
Hierarchy Chart
CSpinButtonCtrl::SetBase

Other Resources

CSpinButtonCtrl Members

CSpinButtonCtrl::GetBuddy

Retrieves a pointer to the current buddy window.

```
CWnd* GetBuddy( ) const;
```

Return Value

A pointer to the current buddy window.

See Also

Reference

CSpinButtonCtrl Class
Hierarchy Chart
CSpinButtonCtrl::SetBuddy

Other Resources

CSpinButtonCtrl Members

CSpinButtonCtrl::GetPos

Retrieves the current position of a spin button control.

```
int GetPos( ) const;
int GetPos32(
   LPBOOL lpbError = NULL
) const;
```

Parameters

lpbError

A pointer to a boolean value that is set to zero if the value is successfully retrieved or non-zero if an error occurs. If this parameter is set to **NULL**, errors are not reported.

Return Value

The first version returns the 16-bit current position in the low-order word. The high-order word is nonzero if an error occurred.

The second version returns the 32-bit position.

Remarks

When it processes the value returned, the control updates its current position based on the caption of the buddy window. The control returns an error if there is no buddy window or if the caption specifies an invalid or out-of-range value.

See Also

Reference

CSpinButtonCtrl Class
Hierarchy Chart
CSpinButtonCtrl::SetPos

Other Resources

CSpinButtonCtrl Members

CSpinButtonCtrl::GetRange

Retrieves the upper and lower limits (range) for a spin button control.

```
DWORD GetRange( ) const;
void GetRange(
   int &lower,
   int& upper
) const;
void GetRange32(
   int &lower,
   int &upper
) const;
```

Parameters

lower

Reference to an integer that receives the lower limit for the control.

upper

Reference to an integer that receives the upper limit for the control.

Return Value

The first version returns a 32-bit value containing the upper and lower limits. The low-order word is the upper limit for the control, and the high-order word is the lower limit.

Remarks

The member function **GetRange32** retrieves the spin button control's range as a 32-bit integer.

See Also

Reference

CSpinButtonCtrl Class
Hierarchy Chart
CSpinButtonCtrl::SetRange

Other Resources

CSpinButtonCtrl Members

CSpinButtonCtrl::SetAccel

Sets the acceleration for a spin button control.

```
BOOL SetAccel(
   int nAccel,
   UDACCEL* pAccel
);
```

Parameters

nAccel

Number of UDACCEL structures specified by *pAccel*.

pAccel

Pointer to an array of **UDACCEL** structures, which contain acceleration information. Elements should be sorted in ascending order based on the **nSec** member.

Return Value

Nonzero if successful; otherwise 0.

See Also

Reference

CSpinButtonCtrl Class
Hierarchy Chart
CSpinButtonCtrl::GetAccel

Other Resources

CSpinButtonCtrl Members

CSpinButtonCtrl::SetBase

Sets the base for a spin button control.

```
int SetBase(
   int nBase
);
```

Parameters

nBase

New base value for the control. It can be 10 for decimal or 16 for hexadecimal.

Return Value

The previous base value if successful, or zero if an invalid base is given.

Remarks

The base value determines whether the buddy window displays numbers in decimal or hexadecimal digits. Hexadecimal numbers are always unsigned; decimal numbers are signed.

See Also

Reference

CSpinButtonCtrl Class
Hierarchy Chart
CSpinButtonCtrl::GetBase

Other Resources

CSpinButtonCtrl Members

CSpinButtonCtrl::SetBuddy

Sets the buddy window for a spin button control.

```
CWnd* SetBuddy(
   CWnd* pWndBuddy
);
```

Parameters

pWndBuddy

Pointer to the new buddy window.

Return Value

A pointer to the previous buddy window.

See Also

Reference

CSpinButtonCtrl Class

Hierarchy Chart

CSpinButtonCtrl::GetBuddy

Other Resources

CSpinButtonCtrl Members

CSpinButtonCtrl::SetPos

Sets the current position for a spin button control.

```
int SetPos(
   int nPos
);
int SetPos32(
   int nPos
);
```

Parameters

nPos

New position for the control. This value must be in the range specified by the upper and lower limits for the control.

Return Value

The previous position (16-bit precision for **SetPos**, 32-bit precision for **SetPos32**).

Remarks

SetPos32 sets the 32-bit position.

See Also

Reference

CSpinButtonCtrl Class
Hierarchy Chart
CSpinButtonCtrl::SetRange
CSpinButtonCtrl::GetPos

Other Resources

CSpinButtonCtrl Members

CSpinButtonCtrl::SetRange

Sets the upper and lower limits (range) for a spin button control.

```
void SetRange(
   short nLower,
   short nUpper
);
void SetRange32(
   int nLower,
   int nUpper
);
```

Parameters

*nLower*and *nUpper*

Upper and lower limits for the control. For **SetRange**, neither limit can be greater than **UD_MAXVAL** or less than **UD_MINVAL**; in addition, the difference between the two limits cannot exceed **UD_MAXVAL**. **SetRange32** places no restrictions on the limits; use any integers.

Remarks

The member function **SetRange32** sets the 32-bit range for the spin button control.

Note
The default range for the spin button has the maximum set to zero (0) and the minimum set to 100. Because the maximum value is less than the minimum value, clicking the up arrow will decrease the position and clicking the down arrow will increase it. Use **CSpinButtonCtrl::SetRange** to adjust these values.

See Also

Reference

CSpinButtonCtrl Class
Hierarchy Chart
CSpinButtonCtrl::GetRange
CSpinButtonCtrl::GetPos
Using CSpinButtonCtrl

Other Resources

CSpinButtonCtrl Members

CSplitterWnd Class

This topic has been updated for Visual Studio 2005 SP1.

Provides the functionality of a splitter window, which is a window that contains multiple panes.

```
class CSplitterWnd : public CWnd
```

Remarks

A pane is usually an application-specific object derived from CView, but it can be any CWnd object that has the appropriate child window ID.

A **CSplitterWnd** object is usually embedded in a parent CFrameWnd or CMDIChildWnd object. Create a **CSplitterWnd** object using the following steps:

1. Embed a **CSplitterWnd** member variable in the parent frame.
2. Override the parent frame's CFrameWnd::OnCreateClient member function.
3. From within the overridden **OnCreateClient**, call the Create or CreateStatic member function of **CSplitterWnd**.

Call the **Create** member function to create a dynamic splitter window. A dynamic splitter window typically is used to create and scroll a number of individual panes, or views, of the same document. The framework automatically creates an initial pane for the splitter; then the framework creates, resizes, and disposes of additional panes as the user operates the splitter window's controls.

When you call **Create**, you specify a minimum row height and column width that determine when the panes are too small to be fully displayed. After you call **Create**, you can adjust these minimums by calling the SetColumnInfo and SetRowInfo member functions.

Also use the **SetColumnInfo** and **SetRowInfo** member functions to set an "ideal" width for a column and "ideal" height for a row. When the framework displays a splitter window, it first displays the parent frame, then the splitter window. The framework then lays out the panes in columns and rows according to their ideal dimensions, working from the upper-left to the lower-right corner of the splitter window's client area.

All panes in a dynamic splitter window must be of the same class. Familiar applications that support dynamic splitter windows include Microsoft Word and Microsoft Excel.

Use the **CreateStatic** member function to create a static splitter window. The user can change only the size of the panes in a static splitter window, not their number or order.

You must specifically create all the static splitter's panes when you create the static splitter. Make sure you create all the panes before the parent frame's **OnCreateClient** member function returns, or the framework will not display the window correctly.

The **CreateStatic** member function automatically initializes a static splitter with a minimum row height and column width of 0. After you call **Create**, adjust these minimums by calling the SetColumnInfo and SetRowInfo member functions. Also use **SetColumnInfo** and **SetRowInfo** after you call **CreateStatic** to indicate desired ideal pane dimensions.

The individual panes of a static splitter often belong to different classes. For examples of static splitter windows, see the graphics editor and the Windows File Manager.

A splitter window supports special scroll bars (apart from the scroll bars that panes may have). These scroll bars are children of the **CSplitterWnd** object and are shared with the panes.

You create these special scroll bars when you create the splitter window. For example, a **CSplitterWnd** that has one row, two columns, and the **WS_VSCROLL** style will display a vertical scroll bar that is shared by the two panes. When the user moves the scroll bar, **WM_VSCROLL** messages are sent to both panes. When the panes set the scroll-bar position, the shared scroll bar is set.

For further information on splitter windows, see:

- Technical Note 29
- Knowledge Base article Q262024: HOWTO: Use CPropertySheet as a Child of CSplitterWnd

For more information on how to create dynamic splitter windows, see:

- MFC sample Scribble
- MFC sample VIEWEX.

Requirements

Header: afxext.h

Smart Device Developer Notes

This section has been updated for Visual Studio 2005 SP1.

Starting in Visual Studio 2005 SP1, this class is supported in Smart Device projects.

See Also

Tasks

VIEWEX Sample: Demonstrates Multiple Views, Scroll Views, and Splitter Windows

Reference

CWnd Class

Hierarchy Chart

CView Class

CWnd Class

Other Resources

CSplitterWnd Members

CSplitterWnd Members

Base Class Members

COBJECT Members

CCmdTarget Members

CWnd Members

Construction

Create	Call to create a dynamic splitter window and attach it to the **CSplitterWnd** object.
CreateStatic	Call to create a static splitter window and attach it to the **CSplitterWnd** object.
CreateView	Call to create a pane in a splitter window.
CSplitterWnd	Call to construct a **CSplitterWnd** object.

Operations

GetColumnCount	Returns the current pane column count.
GetColumnInfo	Returns information on the specified column.
GetPane	Returns the pane at the specified row and column.
GetRowCount	Returns the current pane row count.
GetRowInfo	Returns information on the specified row.
GetScrollStyle	Returns the shared scroll-bar style.
IdFromRowCol	Returns the child window ID of the pane at the specified row and column.
IsTracking	Determines if splitter bar is currently being moved.
IsChildPane	Call to determine whether the window is currently a child pane of this splitter window.
RecalcLayout	Call to redisplay the splitter window after adjusting row or column size.
SetColumnInfo	Call to set the specified column information.
SetRowInfo	Call to set the specified row information.
SetScrollStyle	Specifies the new scroll-bar style for the splitter window's shared scroll-bar support.

Overridables

ActivateNext	Performs the Next Pane or Previous Pane command.
CanActivateNext	Checks to see if the Next Pane or Previous Pane command is currently possible.
CreateScrollBarCtrl	Creates a shared scroll bar control.
DeleteColumn	Deletes a column from the splitter window.

DeleteRow	Deletes a row from the splitter window.
DeleteView	Deletes a view from the splitter window.
DoKeyboardSplit	Performs the keyboard split command, usually "Window Split."
DoScroll	Performs synchronized scrolling of split windows.
DoScrollBy	Scrolls split windows by a given number of pixels.
GetActivePane	Determines the active pane from the focus or active view in the frame.
OnDrawSplitter	Renders an image of a split window.
OnInvertTracker	Renders the image of a split window to be the same size and shape as the frame window.
SetActivePane	Sets a pane to be the active one in the frame.
SplitColumn	Indicates where a frame window splits vertically.
SplitRow	Indicates where a frame window splits horizontally.

See Also

Reference

CSplitterWnd Class

Hierarchy Chart

CSplitterWnd Member Functions

For information about the member functions in **CSplitterWnd**, see CSplitterWnd Members.

CSplitterWnd::ActivateNext

Called by the framework to perform the Next Pane or Previous Pane command.

```
virtual void ActivateNext(
   BOOL bPrev = FALSE
);
```

Parameters

bPrev

Indicates which window to activate. **TRUE** for previous; **FALSE** for next.

Remarks

This member function is a high level command that is used by the CView class to delegate to the **CSplitterWnd** implementation.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CView Class
CSplitterWnd::CanActivateNext
CSplitterWnd::SetActivePane

Other Resources

CSplitterWnd Members

CSplitterWnd::CanActivateNext

Called by the framework to check to see if the Next Pane or Previous Pane command is currently possible.

```
virtual BOOL CanActivateNext(
   BOOL bPrev = FALSE
);
```

Parameters

bPrev

Indicates which window to activate. **TRUE** for previous; **FALSE** for next.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function is a high level command that is used by the CView class to delegate to the **CSplitterWnd** implementation.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::ActivateNext
CSplitterWnd::SetActivePane

Other Resources

CSplitterWnd Members

CSplitterWnd::Create

To create a dynamic splitter window, call the **Create** member function.

```
virtual BOOL Create(
   CWnd* pParentWnd,
   int nMaxRows,
   int nMaxCols,
   SIZE sizeMin,
   CCreateContext* pContext,
   DWORD dwStyle = WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL | SPLS_DYNAMIC_SPLIT,
   UINT nID = AFX_IDW_PANE_FIRST
);
```

Parameters

pParentWnd

The parent frame window of the splitter window.

nMaxRows

The maximum number of rows in the splitter window. This value must not exceed 2.

nMaxCols

The maximum number of columns in the splitter window. This value must not exceed 2.

sizeMin

Specifies the minimum size at which a pane may be displayed.

pContext

A pointer to a CCreateContext structure. In most cases, this can be the *pContext* passed to the parent frame window.

dwStyle

Specifies the window style.

nID

The child window ID of the window. The ID can be **AFX_IDW_PANE_FIRST** unless the splitter window is nested inside another splitter window.

Return Value

Nonzero if successful; otherwise 0.

Remarks

You can embed a **CSplitterWnd** in a parent CFrameWnd or CMDIChildWnd object by taking the following steps:

1. Embed a **CSplitterWnd** member variable in the parent frame.
2. Override the parent frame's CFrameWnd::OnCreateClient member function.
3. Call the **Create** member function from within the overridden **OnCreateClient**.

When you create a splitter window from within a parent frame, pass the parent frame's *pContext* parameter to the splitter window. Otherwise, this parameter can be **NULL**.

The initial minimum row height and column width of a dynamic splitter window are set by the *sizeMin* parameter. These minimums, which determine whether a pane is too small to be shown in its entirety, can be changed with the SetRowInfo and SetColumnInfo member functions.

For more on dynamic splitter windows, see "Splitter Windows" in the article Multiple Document Types, Views, and Frame Windows, Technical Note 29, and the CSplitterWnd class overview.

Example

```
// the following function is created by the MFC Application Wizard
// when you select Split window from the User Interface Features tab:

BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT /*lpcs*/,
      CCreateContext* pContext)
{
   return m_wndSplitter.Create(this,
         2, 2,               // TODO: adjust the number of rows, columns
         CSize(10, 10),      // TODO: adjust the minimum pane size
         pContext);
}
```

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::CreateStatic
CFrameWnd::OnCreateClient
CSplitterWnd::SetRowInfo
CSplitterWnd::SetColumnInfo
CSplitterWnd::CreateView

Other Resources

CSplitterWnd Members

CSplitterWnd::CreateScrollBarCtrl

Called by the framework to create a shared scroll bar control.

```
virtual BOOL CreateScrollBarCtrl(
   DWORD dwStyle,
   UINT nID
);
```

Parameters

dwStyle

Specifies the window style.

nID

The child window ID of the window. The ID can be **AFX_IDW_PANE_FIRST** unless the splitter window is nested inside another splitter window.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Override **CreateScrollBarCtrl** to include extra controls next to a scroll bar. The default behavior is to create normal Windows scroll bar controls.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
AfxGetInstanceHandle

Other Resources

CSplitterWnd Members

CSplitterWnd::CreateStatic

To create a static splitter window, call the **CreateStatic** member function.

```
virtual BOOL CreateStatic(
   CWnd* pParentWnd,
   int nRows,
   int nCols,
   DWORD dwStyle = WS_CHILD | WS_VISIBLE,
   UINT nID = AFX_IDW_PANE_FIRST
);
```

Parameters

pParentWnd

The parent frame window of the splitter window.

nRows

The number of rows. This value must not exceed 16.

nCols

The number of columns. This value must not exceed 16.

dwStyle

Specifies the window style.

nID

The child window ID of the window. The ID can be **AFX_IDW_PANE_FIRST** unless the splitter window is nested inside another splitter window.

Return Value

Nonzero if successful; otherwise 0.

Remarks

A **CSplitterWnd** is usually embedded in a parent **CFrameWnd** or CMDIChildWnd object by taking the following steps:

1. Embed a **CSplitterWnd** member variable in the parent frame.
2. Override the parent frame's **OnCreateClient** member function.
3. Call the **CreateStatic** member function from within the overridden CFrameWnd::OnCreateClient.

A static splitter window contains a fixed number of panes, often from different classes.

When you create a static splitter window, you must at the same time create all its panes. The CreateView member function is usually used for this purpose, but you can create other nonview classes as well.

The initial minimum row height and column width for a static splitter window is 0. These minimums, which determine when a pane is too small to be shown in its entirety, can be changed with the SetRowInfo and SetColumnInfo member functions.

To add scroll bars to a static splitter window, add the **WS_HSCROLL** and **WS_VSCROLL** styles to *dwStyle*.

See "Splitter Windows" in the article Multiple Document Types, Views, and Frame Windows, Technical Note 29, and the CSplitterWnd class overview for more on static splitter windows.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::Create
CFrameWnd::OnCreateClient
CSplitterWnd::SetRowInfo

CSplitterWnd::SetColumnInfo
CSplitterWnd::CreateView

Other Resources

CSplitterWnd Members

CSplitterWnd::CreateView

Creates the panes for a static splitter window.

```
virtual BOOL CreateView(
   int row,
   int col,
   CRuntimeClass* pViewClass,
   SIZE sizeInit,
   CCreateContext* pContext
);
```

Parameters

row

Specifies the splitter window row in which to place the new view.

col

Specifies the splitter window column in which to place the new view.

pViewClass

Specifies the CRuntimeClass of the new view.

sizeInit

Specifies the initial size of the new view.

pContext

A pointer to a creation context used to create the view (usually the *pContext* passed into the parent frame's overridden CFrameWnd::OnCreateClient member function in which the splitter window is being created).

Return Value

Nonzero if successful; otherwise 0.

Remarks

All panes of a static splitter window must be created before the framework displays the splitter.

The framework also calls this member function to create new panes when the user of a dynamic splitter window splits a pane, row, or column.

Example

```
// this function creates the panes for a static splitter window
BOOL CChildFrame::OnCreateClient( LPCREATESTRUCT lpcs,
   CCreateContext* pContext)
{

   BOOL bCreateSpltr = m_wndSplitter.CreateStatic( this, 2, 1);
   // COneView and CAnotherView are user-defined views derived from CMDIView
   m_wndSplitter.CreateView(0,0,RUNTIME_CLASS(COneView), CSize(0,0),
      pContext);
   m_wndSplitter.CreateView(1,0,RUNTIME_CLASS(CAnotherView), CSize(0,0),
      pContext);

   return (bCreateSpltr);
}
```

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::Create

Other Resources

CSplitterWnd Members

CSplitterWnd::CSplitterWnd

Call to construct a **CSplitterWnd** object.

```
CSplitterWnd( );
```

Remarks

Construct a **CSplitterWnd** object in two steps. First, call the constructor, which creates the **CSplitterWnd** object, and then call the Create member function, which creates the splitter window and attaches it to the **CSplitterWnd** object.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::Create

Other Resources

CSplitterWnd Members

CSplitterWnd::DeleteColumn

Deletes a column from the splitter window.

```
virtual void DeleteColumn(
   int colDelete
);
```

Parameters

colDelete

Specifies the column to be deleted.

Remarks

This member function is called by the framework to implement the logic of the dynamic splitter window (that is, if the splitter window has the **SPLS_DYNAMIC_SPLIT** style). It can be customized, along with the virtual function CreateView, to implement more advanced dynamic splitters.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::DeleteRow
CSplitterWnd::CreateView
CSplitterWnd::DeleteView

Other Resources

CSplitterWnd Members

CSplitterWnd::DeleteRow

Deletes a row from the splitter window.

```
virtual void DeleteRow(
   int rowDelete
);
```

Parameters

rowDelete

Specifies the row to be deleted.

Remarks

This member function is called by the framework to implement the logic of the dynamic splitter window (that is, if the splitter window has the **SPLS_DYNAMIC_SPLIT** style). It can be customized, along with the virtual function CreateView, to implement more advanced dynamic splitters.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::DeleteColumn
CSplitterWnd::CreateView
CSplitterWnd::DeleteView

Other Resources

CSplitterWnd Members

CSplitterWnd::DeleteView

Deletes a view from the splitter window.

```
virtual void DeleteView(
   int row,
   int col
);
```

Parameters

row

Specifies the splitter window row at which to delete the view.

col

Specifies the splitter window column at which to delete the view.

Remarks

If the active view is being deleted, the next view will become active. The default implementation assumes the view will auto delete in PostNcDestroy.

This member function is called by the framework to implement the logic of the dynamic splitter window (that is, if the splitter window has the **SPLS_DYNAMIC_SPLIT** style). It can be customized, along with the virtual function CreateView, to implement more advanced dynamic splitters.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CWnd::PostNcDestroy
CSplitterWnd::CreateView
CSplitterWnd::DeleteColumn
CSplitterWnd::DeleteRow

Other Resources

CSplitterWnd Members

CSplitterWnd::DoKeyboardSplit

Performs the keyboard split command, usually "Window Split."

```
virtual BOOL DoKeyboardSplit( );
```

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function is a high level command that is used by the CView class to delegate to the **CSplitterWnd** implementation.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CView Class

Other Resources

CSplitterWnd Members

CSplitterWnd::DoScroll

Performs synchronized scrolling of split windows.

```
virtual BOOL DoScroll(
   CView* pViewFrom,
   UINT nScrollCode,
   BOOL bDoScroll = TRUE
);
```

Parameters

pViewFrom

A pointer to the view from which the scrolling message originates.

nScrollCode

A scroll-bar code that indicates the user's scrolling request. This parameter is composed of two parts: a low-order byte, which determines the type of scrolling occurring horizontally, and a high-order byte, which determines the type of scrolling occurring vertically:

- **SB_BOTTOM** Scrolls to bottom.
- **SB_LINEDOWN** Scrolls one line down.
- **SB_LINEUP** Scrolls one line up.
- **SB_PAGEDOWN** Scrolls one page down.
- **SB_PAGEUP** Scrolls one page up.
- **SB_TOP** Scrolls to top.

bDoScroll

Determines whether the specified scrolling action occurs. If *bDoScroll* is **TRUE** (that is, if a child window exists, and if the split windows have a scroll range), then the specified scrolling action can take place; if *bDoScroll* is **FALSE** (that is, if no child window exists, or the split views have no scroll range), then scrolling does not occur.

Return Value

Nonzero if synchronized scrolling occurs; otherwise 0.

Remarks

This member function is called by the framework to perform synchronized scrolling of split windows when the view receives a scroll message. Override to require an action by the user before synchronized scrolling is allowed.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::DoScrollBy
CView::OnScroll

Other Resources

CSplitterWnd Members

CSplitterWnd::DoScrollBy

Scrolls split windows by a given number of pixels.

```
virtual BOOL DoScrollBy(
   CView* pViewFrom,
   CSize sizeScroll,
   BOOL bDoScroll = TRUE
);
```

Parameters

pViewFrom

A pointer to the view from which the scrolling message originates.

sizeScroll

Number of pixels to be scrolled horizontally and vertically.

bDoScroll

Determines whether the specified scrolling action occurs. If *bDoScroll* is **TRUE** (that is, if a child window exists, and if the split windows have a scroll range), then the specified scrolling action can take place; if *bDoScroll* is **FALSE** (that is, if no child window exists, or the split views have no scroll range), then scrolling does not occur.

Return Value

Nonzero if synchronized scrolling occurs; otherwise 0.

Remarks

This member function is called by the framework in response to a scroll message, to perform synchronized scrolling of the split windows by the amount, in pixels, indicated by *sizeScroll*. Positive values indicate scrolling down and to the right; negative values indicate scrolling up and to the left.

Override to require an action by the user before allowing scroll.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::DoScroll
CView::OnScroll

Other Resources

CSplitterWnd Members

CSplitterWnd::GetActivePane

Determines the active pane from the focus or active view in the frame.

```
virtual CWnd* GetActivePane(
   int* pRow = NULL,
   int* pCol = NULL
);
```

Parameters

pRow

A pointer to an **int** to retrieve the row number of the active pane.

pCol

A pointer to an **int** to retrieve the column number of the active pane.

Return Value

Pointer to the active pane. **NULL** if no active pane exists.

Remarks

This member function is called by the framework to determine the active pane in a splitter window. Override to require an action by the user before getting the active pane.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::SetActivePane
CFrameWnd::GetActiveView
CWnd::GetParentFrame
CWnd::GetFocus

Other Resources

CSplitterWnd Members

CSplitterWnd::GetColumnCount

Returns the current pane column count.

```
int GetColumnCount( ) const;
```

Return Value

Returns the current number of columns in the splitter. For a static splitter, this will also be the maximum number of columns.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::GetRowCount

Other Resources

CSplitterWnd Members

CSplitterWnd::GetColumnInfo

Returns information on the specified column.

```
void GetColumnInfo(
   int col,
   int& cxCur,
   int& cxMin
) const;
```

Parameters

col

Specifies a column.

cxCur

A reference to an **int** to be set to the current width of the column.

cxMin

A reference to an **int** to be set to the current minimum width of the column.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::SetColumnInfo
CSplitterWnd::GetRowInfo

Other Resources

CSplitterWnd Members

CSplitterWnd::GetPane

Returns the pane at the specified row and column.

```
CWnd* GetPane(
   int row,
   int col
) const;
```

Parameters

row

Specifies a row.

col

Specifies a column.

Return Value

Returns the pane at the specified row and column. The returned pane is usually a CView-derived class.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::GetActivePane
CSplitterWnd::IdFromRowCol
CSplitterWnd::IsChildPane

Other Resources

CSplitterWnd Members

CSplitterWnd::GetRowCount

Returns the current pane row count.

```
int GetRowCount( ) const;
```

Return Value

Returns the current number of rows in the splitter window. For a static splitter window, this will also be the maximum number of rows.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::GetColumnCount

Other Resources

CSplitterWnd Members

CSplitterWnd::GetRowInfo

Returns information on the specified row.

```
void GetRowInfo(
   int row,
   int& cyCur,
   int& cyMin
) const;
```

Parameters

row

Specifies a row.

cyCur

Reference to **int** to be set to the current height of the row in pixels.

cyMin

Reference to **int** to be set to the current minimum height of the row in pixels.

Remarks

Call this member function to obtain information about the specified row. The *cyCur* parameter is filled with the current height of the specified row, and *cyMin* is filled with the minimum height of the row.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::SetRowInfo
CSplitterWnd::GetColumnInfo

Other Resources

CSplitterWnd Members

CSplitterWnd::GetScrollStyle

Returns the shared scroll-bar style for the splitter window.

```
DWORD GetScrollStyle( ) const;
```

Return Value

One or more of the following windows style flags, if successful:

- **WS_HSCROLL** If the splitter currently manages shared horizontal scroll bars.
- **WS_VSCROLL** If the splitter currently manages shared vertical scroll bars.

If zero, the splitter window does not currently manage any shared scroll bars.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::SetScrollStyle

Other Resources

CSplitterWnd Members

CSplitterWnd::IdFromRowCol

Obtains the child window ID for the pane at the specified row and column.

```
int IdFromRowCol(
   int row,
   int col
) const;
```

Parameters

row

Specifies the splitter window row.

col

Specifies the splitter window column.

Return Value

The child window ID for the pane.

Remarks

This member function is used for creating nonviews as panes and may be called before the pane exists.

Example

```
HBRUSH CMySplitter::OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor)
{
   HBRUSH hbr = CSplitterWnd::OnCtlColor(pDC, pWnd, nCtlColor);

   if( nCtlColor == CTLCOLOR_LISTBOX &&
      pWnd->GetDlgCtrlID() == IdFromRowCol(1,0) )
      {
         // Pane 1,0 is a list box. Set the color of the text to be blue.
         pDC->SetBkColor(m_BkColor);
         pDC->SetTextColor(RGB(0,0,255));
         return ( HBRUSH ) m_hbrListBoxBkgnd.GetSafeHandle();
      }
   // TODO: Return a different brush if the default is not desired
   return hbr;
}
```

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::GetPane
CSplitterWnd::IsChildPane

Other Resources

CSplitterWnd Members

CSplitterWnd::IsChildPane

Determines whether *pWnd* is currently a child pane of this splitter window.

```
BOOL IsChildPane(
   CWnd* pWnd,
   int* pRow,
   int* pCol
);
```

Parameters

pWnd

A pointer to a CWnd object to be tested.

pRow

A pointer to an **int** in which to store row number.

pCol

A pointer to an **int** in which to store a column number.

Return Value

If nonzero, *pWnd* is currently a child pane of this splitter window, and *pRow* and *pCol* are filled in with the position of the pane in the splitter window. If *pWnd* is not a child pane of this splitter window, 0 is returned.

Remarks

In Visual C++ versions prior to 6.0, this function was defined as

```
BOOL IsChildPane(CWnd* pWnd, int& row, int& col);
```

This version is now obsolete and should not be used.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::GetPane

Other Resources

CSplitterWnd Members

CSplitterWnd::IsTracking

Call this member function to determine if the splitter bar in the window is currently being moved.

```
BOOL IsTracking( );
```

Return Value

Nonzero if a splitter operation is in progress; otherwise 0.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::GetPane

Other Resources

CSplitterWnd Members

CSplitterWnd::OnDrawSplitter

Renders an image of a split window.

```
virtual void OnDrawSplitter(
   CDC* pDC,
   ESplitType nType,
   const CRect& rect
);
```

Parameters

pDC

A pointer to the device context in which to draw. If *pDC* is **NULL**, then CWnd::RedrawWindow is called by the framework and no split window is drawn.

nType

A value of the **enum ESplitType**, which can be one of the following:

- **splitBox** The splitter drag box.
- **splitBar** The bar that appears between the two split windows.
- **splitIntersection** The intersection of the split windows. This element will not be called when running on Windows 95/98.
- **splitBorder** The split window borders.

rect

A reference to a CRect object specifying the size and shape of the split windows.

Remarks

This member function is called by the framework to draw and specify the exact characteristics of a splitter window. Override **OnDrawSplitter** for advanced customization of the imagery for the various graphical components of a splitter window. The default imagery is similar to the splitter in Microsoft Works for Windows or Microsoft Windows 95/98, in that the intersections of the splitter bars are blended together.

For more on dynamic splitter windows, see "Splitter Windows" in the article Multiple Document Types, Views, and Frame Windows, Technical Note 29, and the CSplitterWnd class overview.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::OnInvertTracker

Other Resources

CSplitterWnd Members

CSplitterWnd::OnInvertTracker

Renders the image of a split window to be the same size and shape as the frame window.

```
virtual void OnInvertTracker(
   const CRect& rect
);
```

Parameters

rect

Reference to a **CRect** object specifying the tracking rectangle.

Remarks

This member function is called by the framework during resizing of splitters. Override **OnInvertTracker** for advanced customization of the imagery of the splitter window. The default imagery is similar to the splitter in Microsoft Works for Windows or Microsoft Windows 95/98, in that the intersections of the splitter bars are blended together.

For more on dynamic splitter windows, see "Splitter Windows" in the article Multiple Document Types, Views, and Frame Windows, Technical Note 29, and the CSplitterWnd class overview.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::OnDrawSplitter

Other Resources

CSplitterWnd Members

CSplitterWnd::RecalcLayout

Call to redisplay the splitter window after adjusting row or column size.

```
virtual void RecalcLayout( );
```

Remarks

Call this member function to correctly redisplay the splitter window after you have adjusted row and column sizes with the SetRowInfo and SetColumnInfo member functions. If you change row and column sizes as part of the creation process before the splitter window is visible, it is not necessary to call this member function.

The framework calls this member function whenever the user resizes the splitter window or moves a split.

Example

See the example for CSplitterWnd::SetColumnInfo.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::SetRowInfo
CSplitterWnd::SetColumnInfo

Other Resources

CSplitterWnd Members

CSplitterWnd::SetActivePane

Sets a pane to be the active one in the frame.

```
virtual void SetActivePane(
   int row,
   int col,
   CWnd* pWnd = NULL
);
```

Parameters

row

If *pWnd* is **NULL**, specifies the row in the pane that will be active.

col

If *pWnd* is **NULL**, specifies the column in the pane that will be active.

pWnd

A pointer to a **CWnd** object. If **NULL**, the pane specified by *row* and *col* is set active. If not **NULL**, specifies the pane that is set active.

Remarks

This member function is called by the framework to set a pane as active when the user changes the focus to a pane within the frame window. You can explicitly call **SetActivePane** to change the focus to the specified view.

Specify pane by providing either row and column, **or** by providing *pWnd*.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::GetActivePane
CSplitterWnd::GetPane
CFrameWnd::SetActiveView

Other Resources

CSplitterWnd Members

CSplitterWnd::SetColumnInfo

Call to set the specified column information.

```
void SetColumnInfo(
   int col,
   int cxIdeal,
   int cxMin
);
```

Parameters

col

Specifies a splitter window column.

cxIdeal

Specifies an ideal width for the splitter window column in pixels.

cxMin

Specifies a minimum width for the splitter window column in pixels.

Remarks

Call this member function to set a new minimum width and ideal width for a column. The column minimum value determines when the column will be too small to be fully displayed.

When the framework displays the splitter window, it lays out the panes in columns and rows according to their ideal dimensions, working from the upper-left to the lower-right corner of the splitter window's client area.

Example

```
void CChildFrame::OnSize(UINT nType, int cx, int cy)
{
   CMDIChildWnd::OnSize(nType, cx, cy);

   CRect rect;
   GetWindowRect( &rect );
   if( m_bSplitterCreated )  // m_bSplitterCreated set in OnCreateClient
   {
      m_wndSplitter.SetColumnInfo(0, rect.Width()/2, 10);
      m_wndSplitter.SetColumnInfo(1, rect.Width()/2, 10);
      m_wndSplitter.RecalcLayout();
   }
}
```

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::GetRowInfo
CSplitterWnd::RecalcLayout

Other Resources

CSplitterWnd Members

CSplitterWnd::SetRowInfo

Call to set the specified row information.

```
void SetRowInfo(
   int row,
   int cyIdeal,
   int cyMin
);
```

Parameters

row

Specifies a splitter window row.

cyIdeal

Specifies an ideal height for the splitter window row in pixels.

cyMin

Specifies a minimum height for the splitter window row in pixels.

Remarks

Call this member function to set a new minimum height and ideal height for a row. The row minimum value determines when the row will be too small to be fully displayed.

When the framework displays the splitter window, it lays out the panes in columns and rows according to their ideal dimensions, working from the upper-left to the lower-right corner of the splitter window's client area.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::GetRowInfo
CSplitterWnd::SetColumnInfo
CSplitterWnd::RecalcLayout

Other Resources

CSplitterWnd Members

CSplitterWnd::SetScrollStyle

Specifies the new scroll style for the splitter window's shared scroll-bar support.

```
void SetScrollStyle(
   DWORD dwStyle
);
```

Parameters

dwStyle

The new scroll style for the splitter window's shared scroll-bar support, which can be one of the following values:

- **WS_HSCROLL** Create/show horizontal shared scroll bars.
- **WS_VSCROLL** Create/show vertical shared scroll bars.

Remarks

Once a scroll bar is created it will not be destroyed even if **SetScrollStyle** is called without that style; instead those scroll bars are hidden. This allows the scroll bars to retain their state even though they are hidden. After calling **SetScrollStyle** it is necessary to call RecalcLayout for all the changes to take effect.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::GetScrollStyle

Other Resources

CSplitterWnd Members

CSplitterWnd::SplitColumn

Indicates where a frame window splits vertically.

```
virtual BOOL SplitColumn(
   int cxBefore
);
```

Parameters

cxBefore

The position, in pixels, before which the split occurs.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function is called when a vertical splitter window is created. **SplitColumn** indicates the default location where the split occurs.

SplitColumn is called by the framework to implement the logic of the dynamic splitter window (that is, if the splitter window has the **SPLS_DYNAMIC_SPLIT** style). It can be customized, along with the virtual function CreateView, to implement more advanced dynamic splitters.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::CreateView
CSplitterWnd::SplitRow
CSplitterWnd::RecalcLayout

Other Resources

CSplitterWnd Members

CSplitterWnd::SplitRow

Indicates where a frame window splits horizontally.

```
virtual BOOL SplitRow(
   int cyBefore
);
```

Parameters

cyBefore

The position, in pixels, before which the split occurs.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function is called when a horizontal splitter window is created. **SplitRow** indicates the default location where the split occurs.

SplitRow is called by the framework to implement the logic of the dynamic splitter window (that is, if the splitter window has the **SPLS_DYNAMIC_SPLIT** style). It can be customized, along with the virtual function CreateView, to implement more advanced dynamic splitters.

See Also

Reference

CSplitterWnd Class
Hierarchy Chart
CSplitterWnd::SplitColumn
CSplitterWnd::CreateView
CSplitterWnd::RecalcLayout

Other Resources

CSplitterWnd Members

CStatic Class

Provides the functionality of a Windows static control.

```
class CStatic : public CWnd
```

Remarks

A static control displays a text string, box, rectangle, icon, cursor, bitmap, or enhanced metafile. It can be used to label, box, or separate other controls. A static control normally takes no input and provides no output; however, it can notify its parent of mouse clicks if it's created with **SS_NOTIFY** style.

Create a static control in two steps. First, call the constructor to construct the **CStatic** object, then call the Create member function to create the static control and attach it to the **CStatic** object.

If you create a **CStatic** object within a dialog box (through a dialog resource), the **CStatic** object is automatically destroyed when the user closes the dialog box.

If you create a **CStatic** object within a window, you may also need to destroy it. A **CStatic** object created on the stack within a window is automatically destroyed. If you create the **CStatic** object on the heap by using the **new** function, you must call **delete** on the object to destroy it when the you are done with it.

Requirements

Header: afxwin.h

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd Class
CButton Class
CComboBox Class
CEdit Class
CListBox Class
CScrollBar Class
CDialog Class

Other Resources

CStatic Members

CStatic Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Construction

CStatic	Constructs a **CStatic** object.

Initialization

Create	Creates the Windows static control and attaches it to the **CStatic** object.

Operations

GetBitmap	Retrieves the handle of the bitmap previously set with SetBitmap.
GetCursor	Retrieves the handle of the cursor image previously set with SetCursor.
GetEnhMetaFile	Retrieves the handle of the enhanced metafile previously set with SetEnhMetaFile.
GetIcon	Retrieves the handle of the icon previously set with SetIcon.
SetBitmap	Specifies a bitmap to be displayed in the static control.
SetCursor	Specifies a cursor image to be displayed in the static control.
SetEnhMetaFile	Specifies an enhanced metafile to be displayed in the static control.
SetIcon	Specifies an icon to be displayed in the static control.

Overridables

DrawItem	Override to draw an owner-drawn static control.

See Also

Reference

CStatic Class

Hierarchy Chart

CStatic Member Functions

For information about the member functions in **CStatic**, see CStatic Members.

CStatic::Create

Creates the Windows static control and attaches it to the **CStatic** object.

```
virtual BOOL Create(
   LPCTSTR lpszText,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID = 0xffff
);
```

Parameters

lpszText

Specifies the text to place in the control. If **NULL**, no text will be visible.

dwStyle

Specifies the static control's window style. Apply any combination of static control styles to the control.

rect

Specifies the position and size of the static control. It can be either a **RECT** structure or a **CRect** object.

pParentWnd

Specifies the **CStatic** parent window, usually a **CDialog** object. It must not be **NULL**.

nID

Specifies the static control's control ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Construct a **CStatic** object in two steps. First, call the constructor **CStatic**, and then call **Create**, which creates the Windows static control and attaches it to the **CStatic** object.

Apply the following window styles to a static control:

- **WS_CHILD** Always
- **WS_VISIBLE** Usually
- **WS_DISABLED** Rarely

If you're going to display a bitmap, cursor, icon, or metafile in the static control, you'll need to apply one of the following static styles:

- **SS_BITMAP** Use this style for bitmaps.
- **SS_ICON** Use this style for cursors and icons.
- **SS_ENHMETAFILE** Use this style for enhanced metafiles.

For cursors, bitmaps, or icons, you may also want to use the following style:

- **SS_CENTERIMAGE** Use to center the image in the static control.

Example

```
CStatic myStatic;

// Create a child static control that centers its text horizontally.
myStatic.Create(_T("my static"), WS_CHILD|WS_VISIBLE|SS_CENTER,
```

```
    CRect(10,10,150,50), pParentWnd);
```

See Also

Reference

CStatic Class

Hierarchy Chart

CStatic::CStatic

Other Resources

CStatic Members

CStatic::CStatic

Constructs a **CStatic** object.

```
CStatic( );
```

Example

```
// Create a static object on the stack.
CStatic myStatic;

// Create a static object on the heap.
CStatic* pmyStatic = new CStatic;
```

See Also

Reference

CStatic Class

Hierarchy Chart

CStatic::Create

Other Resources

CStatic Members

CStatic::DrawItem

Called by the framework to draw an owner-drawn static control.

```
virtual void DrawItem(
   LPDRAWITEMSTRUCT lpDrawItemStruct
);
```

Parameters

lpDrawItemStruct

A pointer to a DRAWITEMSTRUCT structure. The structure contains information about the item to be drawn and the type of drawing required.

Remarks

Override this function to implement drawing for an owner-drawn **CStatic** object (the control has the style **SS_OWNERDRAW**).

See Also

Reference

CStatic Class

Hierarchy Chart

Other Resources

CStatic Members

CStatic::GetBitmap

Gets the handle of the bitmap, previously set with SetBitmap, that is associated with **CStatic**.

```
HBITMAP GetBitmap( ) const;
```

Return Value

A handle to the current bitmap, or **NULL** if no bitmap has been set.

Example

```
CStatic myStatic;

// Create a child bitmap static control.
myStatic.Create(_T("my static"),
   WS_CHILD|WS_VISIBLE|SS_BITMAP|SS_CENTERIMAGE, CRect(10,10,150,50),
   pParentWnd);

// If no bitmap is defined for the static control, define the bitmap
// to the system close bitmap.
if (myStatic.GetBitmap() == NULL)
   myStatic.SetBitmap( ::LoadBitmap(NULL, MAKEINTRESOURCE(OBM_CLOSE)) );
```

See Also

Reference

CStatic Class
Hierarchy Chart
CStatic::SetBitmap
STM_GETIMAGE
Bitmaps

Other Resources

CStatic Members

CStatic::GetCursor

Gets the handle of the cursor, previously set with SetCursor, that is associated with **CStatic**.

```
HCURSOR GetCursor( );
```

Return Value

A handle to the current cursor, or **NULL** if no cursor has been set.

Example

```
CStatic myStatic;

// Create a child icon static control.
myStatic.Create(_T("my static"),
   WS_CHILD|WS_VISIBLE|SS_ICON|SS_CENTERIMAGE, CRect(10,10,150,50),
   pParentWnd);

// If no image is defined for the static control, define the image
// to the system arrow and question mark cursor.
if (myStatic.GetCursor() == NULL)
   myStatic.SetCursor( ::LoadCursor(NULL, IDC_HELP) );
```

See Also

Reference

CStatic Class
Hierarchy Chart
CStatic::SetCursor
STM_GETIMAGE
Cursors

Other Resources

CStatic Members

CStatic::GetEnhMetaFile

Gets the handle of the enhanced metafile, previously set with SetEnhMetafile, that is associated with **CStatic**.

```
HENHMETAFILE GetEnhMetaFile( ) const;
```

Return Value

A handle to the current enhanced metafile, or **NULL** if no enhanced metafile has been set.

Example

```
CStatic myStatic;

// Create a child enhanced metafile static control.
myStatic.Create(_T("my static"),
   WS_CHILD|WS_VISIBLE|SS_ENHMETAFILE|SS_CENTERIMAGE,
   CRect(10,10,150,50), pParentWnd);

// If no image is defined for the static control, define the image
// to be "myemf.emf."
if (myStatic.GetEnhMetaFile() == NULL)
   myStatic.SetEnhMetaFile( ::GetEnhMetaFile(_T("myemf.emf")) );
```

See Also

Reference

CStatic Class
Hierarchy Chart
CStatic::SetEnhMetaFile
STM_GETIMAGE

Other Resources

CStatic Members

CStatic::GetIcon

Gets the handle of the icon, previously set with SetIcon, that is associated with **CStatic**.

```
HICON GetIcon( ) const;
```

Return Value

A handle to the current icon, or **NULL** if no icon has been set.

Example

```
CStatic myStatic;

// Create a child icon static control.
myStatic.Create(_T("my static"),
   WS_CHILD|WS_VISIBLE|SS_ICON|SS_CENTERIMAGE, CRect(10,10,150,50),
   pParentWnd);

// If no icon is defined for the static control, define the icon
// to the system error icon.
if (myStatic.GetIcon() == NULL)
   myStatic.SetIcon( ::LoadIcon(NULL, IDI_ERROR) );
```

See Also

Reference

CStatic Class
Hierarchy Chart
CStatic::SetIcon
STM_GETICON
Icons

Other Resources

CStatic Members

CStatic::SetBitmap

Associates a new bitmap with the static control.

```
HBITMAP SetBitmap(
   HBITMAP hBitmap
);
```

Parameters

hBitmap

Handle of the bitmap to be drawn in the static control.

Return Value

The handle of the bitmap previously associated with the static control, or **NULL** if no bitmap was associated with the static control.

Remarks

The bitmap will be automatically drawn in the static control. By default, it will be drawn in the upper-left corner and the static control will be resized to the size of the bitmap.

You can use various window and static control styles, including the following:

- **SS_BITMAP** Use this style always for bitmaps.
- **SS_CENTERIMAGE** Use to center in the static control. If the image is larger than the static control, it will be clipped. If it is smaller than the static control, the empty space around the image will be filled by the color of the pixel in the upper left corner of the bitmap.

Example

```
CStatic myStatic;

// Create a child bitmap static control.
myStatic.Create(_T("my static"),
   WS_CHILD|WS_VISIBLE|SS_BITMAP|SS_CENTERIMAGE, CRect(10,10,150,50),
   pParentWnd);

// Set the bitmap of the static control to be the
// system check-mark bitmap.
myStatic.SetBitmap( ::LoadBitmap(NULL, MAKEINTRESOURCE(OBM_CHECK)) );
```

See Also

Reference

CStatic Class
Hierarchy Chart
CStatic::GetBitmap
STM_SETIMAGE
Bitmaps

Other Resources

CStatic Members

CStatic::SetCursor

Associates a new cursor image with the static control.

```
HCURSOR SetCursor(
   HCURSOR hCursor
);
```

Parameters

hCursor

Handle of the cursor to be drawn in the static control.

Return Value

The handle of the cursor previously associated with the static control, or **NULL** if no cursor was associated with the static control.

Remarks

The cursor will be automatically drawn in the static control. By default, it will be drawn in the upper-left corner and the static control will be resized to the size of the cursor.

You can use various window and static control styles, including the following:

- **SS_ICON** Use this style always for cursors and icons.
- **SS_CENTERIMAGE** Use to center in the static control. If the image is larger than the static control, it will be clipped. If it is smaller than the static control, the empty space around the image will be filled with the background color of the static control.

Example

```
CStatic myStatic;

// Create a child icon static control.
myStatic.Create(_T("my static"),
   WS_CHILD|WS_VISIBLE|SS_ICON|SS_CENTERIMAGE, CRect(10,10,150,50),
   pParentWnd);

// Set the image of the static control to be the system arrow
// and small hourglass cursor.
myStatic.SetCursor( ::LoadCursor(NULL, IDC_APPSTARTING) );
```

See Also

Reference

CStatic Class
Hierarchy Chart
CStatic::GetCursor
STM_SETIMAGE
Cursors

Other Resources

CStatic Members

CStatic::SetEnhMetaFile

Associates a new enhanced metafile image with the static control.

```
HENHMETAFILE SetEnhMetaFile(
   HENHMETAFILE hMetaFile
);
```

Parameters

hMetaFile

Handle of the enhanced metafile to be drawn in the static control.

Return Value

The handle of the enhanced metafile previously associated with the static control, or **NULL** if no enhanced metafile was associated with the static control.

Remarks

The enhanced metafile will be automatically drawn in the static control. The enhanced metafile is scaled to fit the size of the static control.

You can use various window and static control styles, including the following:

- **SS_ENHMETAFILE** Use this style always for enhanced metafiles.

Example

```
CStatic myStatic;

// Create a child enhanced metafile static control.
myStatic.Create(_T("my static"),
   WS_CHILD|WS_VISIBLE|SS_ENHMETAFILE|SS_CENTERIMAGE,
   CRect(10,10,150,50), pParentWnd);

// Set the image of the static control to be "myemf.emf."
myStatic.SetEnhMetaFile( ::GetEnhMetaFile(_T("myemf.emf")) );
```

See Also

Reference

CStatic Class
Hierarchy Chart
CStatic::GetEnhMetaFile
STM_SETIMAGE

Other Resources

CStatic Members

CStatic::SetIcon

Associates a new icon image with the static control.

```
HICON SetIcon(
   HICON hIcon
);
```

Parameters

hIcon

Handle of the icon to be drawn in the static control.

Return Value

The handle of the icon previously associated with the static control, or **NULL** if no icon was associated with the static control.

Remarks

The icon will be automatically drawn in the static control. By default, it will be drawn in the upper-left corner and the static control will be resized to the size of the icon.

You can use various window and static control styles, including the following:

- **SS_ICON** Use this style always for cursors and icons.
- **SS_CENTERIMAGE** Use to center in the static control. If the image is larger than the static control, it will be clipped. If it is smaller than the static control, the empty space around the image will be filled with the background color of the static control.

Example

```
CStatic myStatic;

// Create a child icon static control.
myStatic.Create(_T("my static"),
   WS_CHILD|WS_VISIBLE|SS_ICON|SS_CENTERIMAGE, CRect(10,10,150,50),
   pParentWnd);

// Set the icon of the static control to be the system
// question mark icon.
myStatic.SetIcon( ::LoadIcon(NULL, IDI_QUESTION) );
```

See Also

Reference

CStatic Class
Hierarchy Chart
CStatic::GetIcon
STM_SETICON
Icons

Other Resources

CStatic Members

CStatusBar Class

A control bar with a row of text output panes, or "indicators."

```
class CStatusBar : public CControlBar
```

Remarks

The output panes commonly are used as message lines and as status indicators. Examples include the menu help-message lines that briefly explain the selected menu command and the indicators that show the status of the SCROLL LOCK, NUM LOCK, and other keys.

CStatusBar::GetStatusBarCtrl, a member function new to MFC 4.0, allows you to take advantage of the Windows common control's support for status bar customization and additional functionality. **CStatusBar** member functions give you most of the functionality of the Windows common controls; however, when you call **GetStatusBarCtrl**, you can give your status bars even more of the characteristics of a Windows 95/98 status bar. When you call **GetStatusBarCtrl**, it will return a reference to a **CStatusBarCtrl** object. See CStatusBarCtrl for more information about designing toolbars using Windows common controls. For more general information about common controls, see Common Controls in the *Platform SDK*.

The framework stores indicator information in an array with the leftmost indicator at position 0. When you create a status bar, you use an array of string IDs that the framework associates with the corresponding indicators. You can then use either a string ID or an index to access an indicator.

By default, the first indicator is "elastic": it takes up the status-bar length not used by the other indicator panes, so that the other panes are right-aligned.

To create a status bar, follow these steps:

1. Construct the **CStatusBar** object.
2. Call the Create (or CreateEx) function to create the status-bar window and attach it to the **CStatusBar** object.
3. Call SetIndicators to associate a string ID with each indicator.

There are three ways to update the text in a status-bar pane:

1. Call CWnd::SetWindowText to update the text in pane 0 only.
2. Call CCmdUI::SetText in the status bar's **ON_UPDATE_COMMAND_UI** handler.
3. Call SetPaneText to update the text for any pane.

Call SetPaneStyle to update the style of a status-bar pane.

For more information on using **CStatusBar**, see the article Status Bar Implementation in MFC and Technical Note 31 : Control Bars.

Requirements

Header: afxext.h

See Also

Tasks

CTRLBARS Sample: Illustrates Custom Control Bars
DLGCBR32 Sample: Demonstrates Adding a Status Bar and Toolbar to Dialog Boxes

Reference

CControlBar Class
Hierarchy Chart
CStatusBarCtrl Class
CControlBar Class
CWnd::SetWindowText
CStatusBar::SetIndicators

Other Resources

CStatusBar Members

CStatusBar Members

Base Class Members

COBJECT Members

CCmdTarget Members

CWnd Members

CControlBar Members

Construction

Create	Creates the status bar, attaches it to the **CStatusBar** object, and sets the initial font and bar height.
CreateEx	Creates a **CStatusBar** object with additional styles for the embedded **CStatusBarCtrl** object.
CStatusBar	Constructs a **CStatusBar** object.
SetIndicators	Sets indicator IDs.

Attributes

CommandToIndex	Gets index for a given indicator ID.
GetItemID	Gets indicator ID for a given index.
GetItemRect	Gets display rectangle for a given index.
GetPaneInfo	Gets indicator ID, style, and width for a given index.
GetPaneStyle	Gets indicator style for a given index.
GetPaneText	Gets indicator text for a given index.
GetStatusBarCtrl	Allows direct access to the underlying common control.
SetPaneInfo	Sets indicator ID, style, and width for a given index.
SetPaneStyle	Sets indicator style for a given index.
SetPaneText	Sets indicator text for a given index.

Overridables

DrawItem	Called when a visual aspect of an owner-draw status bar control changes.

See Also

Reference

CStatusBar Class

Hierarchy Chart

CStatusBar Member Functions

For information about the member functions in **CStatusBar**, see CStatusBar Members.

CStatusBar::CommandToIndex

Gets the indicator index for a given ID.

```
int CommandToIndex(
   UINT nIDFind
) const;
```

Parameters

nIDFind

String ID of the indicator whose index is to be retrieved.

Return Value

The index of the indicator if successful; –1 if not successful.

Remarks

The index of the first indicator is 0.

See Also

Reference

CStatusBar Class
Hierarchy Chart
CStatusBar::GetItemID

Other Resources

CStatusBar Members

CStatusBar::Create

Creates a status bar (a child window) and associates it with the **CStatusBar** object.

```
virtual BOOL Create(
   CWnd* pParentWnd,
   DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_BOTTOM,
   UINT nID = AFX_IDW_STATUS_BAR
);
```

Parameters

pParentWnd

Pointer to the CWnd object whose Windows window is the parent of the status bar.

dwStyle

The status-bar style. In addition to the standard Windows styles, these styles are supported.

- **CBRS_TOP** Control bar is at top of frame window.
- **CBRS_BOTTOM** Control bar is at bottom of frame window.
- **CBRS_NOALIGN** Control bar is not repositioned when the parent is resized.

nID

The toolbar's child-window ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Also sets the initial font and sets the status bar's height to a default value.

See Also

Reference

CStatusBar Class
Hierarchy Chart
CStatusBar::SetIndicators

Other Resources

CStatusBar Members

CStatusBar::CreateEx

Call this function to create a status bar (a child window) and associate it with the **CStatusBar** object.

```
virtual BOOL CreateEx(
   CWnd* pParentWnd,
   DWORD dwCtrlStyle = 0 ,
   DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_BOTTOM,
   UINT nID = AFX_IDW_STATUS_BAR );
```

Parameters

pParentWnd

Pointer to the CWnd object whose Windows window is the parent of the status bar.

dwCtrlStyle

Additional styles for the creation of the embedded CStatusBarCtrl object. The default specifies a status bar without a sizing grip or tooltip support. Status bar styles supported are:

- **SBARS_SIZEGRIP** The status bar control includes a sizing grip at the right end of the status bar. A sizing grip is similar to a sizing border; it is a rectangular area that the user can click and drag to resize the parent window.
- **SBT_TOOLTIPS** The status bar supports tooltips.

For details on these styles, see Settings for the CStatusBarCtrl.

dwStyle

The status bar style. The default specifies that a visible status bar be created at the bottom of the frame window. Apply any combination of status bar control styles listed in Window Styles and CDialogBar::Create. However, this parameter should always include the WS_CHILD and WS_VISIBLE styles.

nID

The status bar's child-window ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This function also sets the initial font and sets the status bar's height to a default value.

Use **CreateEx**, instead of Create, when certain styles need to be present during the creation of the embedded status bar control. For example, set *dwCtrlStyle* to **SBT_TOOLTIPS** to display tooltips in a status bar object.

See Also

Reference

CStatusBar Class

Hierarchy Chart

Other Resources

CStatusBar Members

CStatusBar::CStatusBar

Constructs a **CStatusBar** object, creates a default status-bar font if necessary, and sets the font characteristics to default values.

```
CStatusBar( );
```

See Also

Reference

CStatusBar Class

Hierarchy Chart

CStatusBar::Create

Other Resources

CStatusBar Members

CStatusBar::DrawItem

This member function is called by the framework when a visual aspect of an owner-drawn status bar changes.

```
virtual void DrawItem(
   LPDRAWITEMSTRUCT lpDrawItemStruct
);
```

Parameters

lpDrawItemStruct

A pointer to a DRAWITEMSTRUCT structure that contains information about the type of drawing required.

Remarks

The **itemAction** member of the **DRAWITEMSTRUCT** structure defines the drawing action that is to be performed. Override this member function to implement drawing for an owner-draw **CStatusBar** object. The application should restore all graphics device interface (GDI) objects selected for the display context supplied in *lpDrawItemStruct* before the termination of this member function.

See Also

Reference

CStatusBar Class

Hierarchy Chart

CWnd::OnDrawItem

Other Resources

CStatusBar Members

CStatusBar::GetItemID

Returns the ID of the indicator specified by *nIndex*.

```
UINT GetItemID(
   int nIndex
) const;
```

Parameters

nIndex

Index of the indicator whose ID is to be retrieved.

Return Value

The ID of the indicator specified by *nIndex*.

See Also

Reference

CStatusBar Class

Hierarchy Chart

CStatusBar::CommandToIndex

Other Resources

CStatusBar Members

CStatusBar::GetItemRect

Copies the coordinates of the indicator specified by *nIndex* into the structure pointed to by *lpRect*.

```
void GetItemRect(
   int nIndex,
   LPRECT lpRect
) const;
```

Parameters

nIndex

Index of the indicator whose rectangle coordinates are to be retrieved.

lpRect

Points to a RECT structure or a CRect object that will receive the coordinates of the indicator specified by *nIndex*.

Remarks

Coordinates are in pixels relative to the upper-left corner of the status bar.

See Also

Reference

CStatusBar Class
Hierarchy Chart
CStatusBar::CommandToIndex
CStatusBar::GetPaneInfo

Other Resources

CStatusBar Members

CStatusBar::GetPaneInfo

Sets *nID*, *nStyle*, and *cxWidth* to the ID, style, and width of the indicator pane at the location specified by *nIndex*.

```
void GetPaneInfo(
   int nIndex,
   UINT& nID,
   UINT& nStyle,
   int& cxWidth
) const;
```

Parameters

nIndex

Index of the pane whose information is to be retrieved.

nID

Reference to a **UINT** that is set to the ID of the pane.

nStyle

Reference to a **UINT** that is set to the style of the pane.

cxWidth

Reference to an integer that is set to the width of the pane.

See Also

Reference

CStatusBar Class
Hierarchy Chart
CStatusBar::SetPaneInfo
CStatusBar::GetItemID
CStatusBar::GetItemRect

Other Resources

CStatusBar Members

CStatusBar::GetPaneStyle

Call this member function to retrieve the style of a status bar's pane.

```
UINT GetPaneStyle(
   int nIndex
) const;
```

Parameters

nIndex

Index of the pane whose style is to be retrieved.

Return Value

The style of the status-bar pane specified by *nIndex*.

Remarks

A pane's style determines how the pane appears.

For a list of styles available for status bars, see Create.

See Also

Reference

CStatusBar Class
Hierarchy Chart
CStatusBar::Create
CStatusBar::SetPaneStyle

Other Resources

CStatusBar Members

CStatusBar::GetPaneText

Call this member function to retrieve the text that appears in a status-bar pane.

```
CString GetPaneText(
   int nIndex
) const;
void GetPaneText(
   int nIndex,
   CString& rString
) const;
```

Parameters

nIndex

Index of the pane whose text is to be retrieved.

rString

A reference to a CString object that contains the text to be retrieved.

Return Value

A **CString** object containing the pane's text.

Remarks

The second form of this member function fills a **CString** object with the string text.

See Also

Reference

CStatusBar Class
Hierarchy Chart
CStatusBar::SetPaneText

Other Resources

CStatusBar Members

CStatusBar::GetStatusBarCtrl

This member function allows direct access to the underlying common control.

```
CStatusBarCtrl& GetStatusBarCtrl( ) const;
```

Return Value

Contains a reference to a CStatusBarCtrl object.

Remarks

Use **GetStatusBarCtrl** to take advantage of the functionality of the Windows status-bar common control, and to take advantage of the support CStatusBarCtrl provides for status-bar customization. For example, by using the common control, you can specify a style that includes a sizing grip on the status bar, or you can specify a style to have the status bar appear at the top of the parent window's client area.

For more general information about common controls, See Common Controls in the *Platform SDK*.

See Also

Reference

CStatusBar Class

Hierarchy Chart

Other Resources

CStatusBar Members

CStatusBar::SetIndicators

Sets each indicator's ID to the value specified by the corresponding element of the array *lpIDArray*, loads the string resource specified by each ID, and sets the indicator's text to the string.

```
BOOL SetIndicators(
   const UINT* lpIDArray,
   int nIDCount
);
```

Parameters

lpIDArray

Pointer to an array of IDs.

nIDCount

Number of elements in the array pointed to by *lpIDArray*.

Return Value

Nonzero if successful; otherwise 0.

See Also

Reference

CStatusBar Class
Hierarchy Chart
CStatusBar::CStatusBar
CStatusBar::Create
CStatusBar::SetPaneInfo
CStatusBar::SetPaneText

Other Resources

CStatusBar Members

CStatusBar::SetPaneInfo

Sets the specified indicator pane to a new ID, style, and width.

```
void SetPaneInfo(
   int nIndex,
   UINT nID,
   UINT nStyle,
   int cxWidth
);
```

Parameters

nIndex

Index of the indicator pane whose style is to be set.

nID

New ID for the indicator pane.

nStyle

New style for the indicator pane.

cxWidth

New width for the indicator pane.

Remarks

The following indicator styles are supported:

- **SBPS_NOBORDERS** No 3-D border around the pane.
- **SBPS_POPOUT** Reverse border so that text "pops out."
- **SBPS_DISABLED** Do not draw text.
- **SBPS_STRETCH** Stretch pane to fill unused space. Only one pane per status bar can have this style.
- **SBPS_NORMAL** No stretch, borders, or pop-out.

See Also

Reference

CStatusBar Class
Hierarchy Chart
CStatusBar::GetPaneInfo

Other Resources

CStatusBar Members

CStatusBar::SetPaneStyle

Call this member function to set the style of a status bar's pane.

```
void SetPaneStyle(
   int nIndex,
   UINT nStyle
);
```

Parameters

nIndex

Index of the pane whose style is to be set.

nStyle

Style of the pane whose style is to be set.

Remarks

A pane's style determines how the pane appears.

For a list of styles available for status bars, see SetPaneInfo.

See Also

Reference

CStatusBar Class
Hierarchy Chart
CStatusBar::Create
CStatusBar::GetPaneStyle

Other Resources

CStatusBar Members

CStatusBar::SetPaneText

Call this member function to set the pane text to the string pointed to by *lpszNewText*.

```
BOOL SetPaneText(
   int nIndex,
   LPCTSTR lpszNewText,
   BOOL bUpdate = TRUE
);
```

Parameters

nIndex

Index of the pane whose text is to be set.

lpszNewText

Pointer to the new pane text.

bUpdate

If **TRUE**, the pane is invalidated after the text is set.

Return Value

Nonzero if successful; otherwise 0.

Remarks

After you call **SetPaneText**, you must add a UI update handler to display the new text in the status bar.

Example

```
//Sets and displays text for pane index 4 and id ID_PANE_FOUR
SetPaneText (4, "My New Status Bar Text", TRUE)

//UI handler in the message map updates the status bar text:
ON_UPDATE_COMMAND_UI ( ID_PANE_FOUR, OnUpdatePane )

//In the appropriate .cpp file add:
void CMyClass::OnUpdatePane (CCmdUI *pCmdUI)  {     pCmdUI->Enable ();  }
```

See Also

Reference

CStatusBar Class
Hierarchy Chart
CStatusBar::GetPaneText

Other Resources

CStatusBar Members

CStatusBarCtrl Class

Provides the functionality of the Windows common status bar control.

```
class CStatusBarCtrl : public CWnd
```

Remarks

A "status bar control" is a horizontal window, usually displayed at the bottom of a parent window, in which an application can display various kinds of status information. The status bar control can be divided into parts to display more than one type of information.

This control (and therefore the **CStatusBarCtrl** class) is available only to programs running under Windows 95/98 and Windows NT version 3.51 and later.

For more information on using **CStatusBarCtrl**, see Controls and Using CStatusBarCtrl.

Requirements

Header: afxcmn.h

See Also

Reference

CWnd Class
Hierarchy Chart
CToolBarCtrl Class

Other Resources

CStatusBarCtrl Members

CStatusBarCtrl Members

Base Class Members

COBJECT Members

CCmdTarget Members

CWnd Members

Construction

Create	Creates a status bar control and attaches it to a **CStatusBarCtrl** object.
CreateEx	Creates a status bar control with the specified Windows extended styles and attaches it to a **CStatusBarCtrl** object.
CStatusBarCtrl	Constructs a **CStatusBarCtrl** object.

Attributes

GetBorders	Retrieves the current widths of the horizontal and vertical borders of a status bar control.
GetParts	Retrieves a count of the parts in a status bar control.
GetRect	Retrieves the bounding rectangle of a part in a status bar control.
GetText	Retrieves the text from the given part of a status bar control.
GetTextLength	Retrieve the length, in characters, of the text from the given part of a status bar control.
GetTipText	Retrieves the tooltip text for a pane in a status bar.
IsSimple	Checks a status window control to determine if it is in simple mode.
SetBkColor	Sets the background color in a status bar.
SetIcon	Sets the icon for a pane in a status bar.
SetMinHeight	Sets the minimum height of a status bar control's drawing area.
SetParts	Sets the number of parts in a status bar control and the coordinate of the right edge of each part.
SetSimple	Specifies whether a status bar control displays simple text or displays all control parts set by a previous call to **SetParts**.
SetText	Sets the text in the given part of a status bar control.
SetTipText	Sets the tooltip text for a pane in a status bar.

Overridables

DrawItem	Called when a visual aspect of an owner-draw status bar control changes.

See Also

Reference

CStatusBarCtrl Class

Hierarchy Chart

CStatusBarCtrl Member Functions

For information about the member functions in **CStatusBarCtrl**, see CStatusBarCtrl Members.

CStatusBarCtrl::Create

Creates a status bar control and attaches it to a **CStatusBarCtrl** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the status bar control's style. Apply any combination of status bar control styles listed in Common Control Styles in the *Platform SDK*. This parameter must include the **WS_CHILD** style. It should also include the **WS_VISIBLE** style.

rect

Specifies the status bar control's size and position. It can be either a CRect object or a RECT structure.

pParentWnd

Specifies the status bar control's parent window, usually a **CDialog**. It must not be **NULL.**

nID

Specifies the status bar control's ID.

Return Value

Nonzero if successful; otherwise zero.

Remarks

You construct a **CStatusBarCtrl** in two steps. First, call the constructor, and then call **Create**, which creates the status bar control and attaches it to the **CStatusBarCtrl** object.

The default position of a status window is along the bottom of the parent window, but you can specify the **CCS_TOP** style to have it appear at the top of the parent window's client area. You can specify the **SBARS_SIZEGRIP** style to include a sizing grip at the right end of the status window. Combining the **CCS_TOP** and **SBARS_SIZEGRIP** styles is not recommended, because the resulting sizing grip is not functional even though the system draws it in the status window.

To create a status bar with extended window styles, call CStatusBarCtrl::CreateEx instead of **Create**.

Example

```
VERIFY( m_wndSBC.Create(WS_CHILD|WS_VISIBLE|CCS_BOTTOM|SBARS_SIZEGRIP,
          CRect(0,0,0,0), this, IDC_STATUS_BAR) );
```

See Also

Reference

CStatusBarCtrl Class

Hierarchy Chart

CStatusBarCtrl::CStatusBarCtrl

Other Resources

CStatusBarCtrl Members

CStatusBarCtrl::CreateEx

Creates a control (a child window) and associates it with the **CStatusBarCtrl** object.

```
virtual BOOL CreateEx(
   DWORD dwExStyle,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwExStyle

Specifies the extended style of the control being created. For a list of extended Windows styles, see the *dwExStyle* parameter for CreateWindowEx in the *Platform SDK*.

dwStyle

Specifies the status bar control's style. Apply any combination of status bar control styles listed in Common Control Styles in the *Platform SDK*. This parameter must include the **WS_CHILD** style. It should also include the **WS_VISIBLE** style.

rect

A reference to a RECT structure describing the size and position of the window to be created, in client coordinates of *pParentWnd*.

pParentWnd

A pointer to the window that is the control's parent.

nID

The control's child-window ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Use **CreateEx** instead of Create to apply extended Windows styles, specified by the Windows extended style preface **WS_EX_**.

See Also

Reference

CStatusBarCtrl Class
Hierarchy Chart
CStatusBarCtrl::CStatusBarCtrl

Other Resources

CStatusBarCtrl Members

CStatusBarCtrl::CStatusBarCtrl

Constructs a **CStatusBarCtrl** object.

```
CStatusBarCtrl( );
```

See Also

Reference

CStatusBarCtrl Class
Hierarchy Chart
CStatusBarCtrl::Create
CStatusBarCtrl::CreateEx

Other Resources

CStatusBarCtrl Members

CStatusBarCtrl::DrawItem

Called by the framework when a visual aspect of an owner-draw status bar control changes.

```
virtual void DrawItem(
   LPDRAWITEMSTRUCT lpDrawItemStruct
);
```

Parameters

lpDrawItemStruct

A long pointer to a DRAWITEMSTRUCT structure that contains information about the type of drawing required.

Remarks

The **itemAction** member of the **DRAWITEMSTRUCT** structure defines the drawing action that is to be performed.

By default, this member function does nothing. Override this member function to implement drawing for an owner-draw **CStatusBarCtrl** object.

The application should restore all graphics device interface (GDI) objects selected for the display context supplied in *lpDrawItemStruct* before this member function terminates.

See Also

Reference

CStatusBarCtrl Class
Hierarchy Chart
CWnd::OnDrawItem

Other Resources

CStatusBarCtrl Members

CStatusBarCtrl::GetBorders

Retrieves the status bar control's current widths of the horizontal and vertical borders and of the space between rectangles.

```
BOOL GetBorders(
   int* pBorders
) const;
BOOL GetBorders(
   int& nHorz,
   int& nVert,
   int& nSpacing
) const;
```

Parameters

pBorders

Address of an integer array having three elements. The first element receives the width of the horizontal border, the second receives the width of the vertical border, and the third receives the width of the border between rectangles.

nHorz

Reference to an integer that receives the width of the horizontal border.

nVert

Reference to an integer that receives the width of the vertical border.

nSpacing

Reference to an integer that receives the width of the border between rectangles.

Return Value

Nonzero if successful; otherwise zero.

Remarks

These borders determine the spacing between the outside edge of the control and the rectangles within the control that contain text.

Example

```
VERIFY( GetRect(1, &rectPane1) );

int borderArray[3];

VERIFY( GetBorders(borderArray) );

int nHorz, nVert, nSpacing;

VERIFY( GetBorders(nHorz, nVert, nSpacing) );
```

See Also

Reference

CStatusBarCtrl Class
Hierarchy Chart
CStatusBarCtrl::GetParts
CStatusBarCtrl::SetParts

Other Resources

CStatusBarCtrl Members

CStatusBarCtrl::GetParts

Retrieves a count of the parts in a status bar control.

```
int GetParts(
   int nParts,
   int* pParts
) const;
```

Parameters

nParts

Number of parts for which to retrieve coordinates. If this parameter is greater than the number of parts in the control, the message retrieves coordinates for existing parts only.

pParts

Address of an integer array having the same number of elements as the number of parts specified by *nParts*. Each element in the array receives the client coordinate of the right edge of the corresponding part. If an element is set to – 1, the position of the right edge for that part extends to the right edge of the status bar.

Return Value

The number of parts in the control if successful, or zero otherwise.

Remarks

This member function also retrieves the coordinate of the right edge of the given number of parts.

Example

```
int pParts[2];

int nParts = GetParts( 2, pParts );
```

See Also

Reference

CStatusBarCtrl Class
Hierarchy Chart
CStatusBarCtrl::GetBorders
CStatusBarCtrl::SetParts

Other Resources

CStatusBarCtrl Members

CStatusBarCtrl::GetRect

Retrieves the bounding rectangle of a part in a status bar control.

```
BOOL GetRect(
   int nPane,
   LPRECT lpRect
) const;
```

Parameters

nPane

Zero-based index of the part whose bounding rectangle is to be retrieved.

lpRect

Address of a RECT structure that receives the bounding rectangle.

Return Value

Nonzero if successful; otherwise zero.

Example

```
CRect rectPane1;

VERIFY( GetRect(1, &rectPane1) );
```

See Also

Reference

CStatusBarCtrl Class
Hierarchy Chart
CStatusBarCtrl::GetParts

Other Resources

CStatusBarCtrl Members

CStatusBarCtrl::GetText

Retrieves the text from the given part of a status bar control.

```
CString GetText(
   int nPane,
   int* pType = NULL
) const;
int GetText(
   LPCTSTR lpszText,
   int nPane,
   int* pType = NULL
) const;
```

Parameters

lpszText

Address of the buffer that receives the text. This parameter is a null-terminated string.

nPane

Zero-based index of the part from which to retrieve text.

pType

Pointer to an integer that receives the type information. The type can be one of these values:

- **0** The text is drawn with a border to appear lower than the plane of the status bar.
- **SBT_NOBORDERS** The text is drawn without borders.
- **SBT_POPOUT** The text is drawn with a border to appear higher than the plane of the status bar.
- **SBT_OWNERDRAW** If the text has the **SBT_OWNERDRAW** drawing type, *pType* receives this message and returns the 32-bit value associated with the text instead of the length and operation type.

Return Value

The length, in characters, of the text or a CString containing the current text.

Example

```
int nType;
char* szPaneOneText;

szPaneOneText = new char[ GetTextLength( 1, &nType ) ];
int nTextLength = GetText( szPaneOneText, 1, &nType );

switch( nType )
{
   case 0:
      // Text is drawn with a border to appear lower than the
      // plane of the status bar
      break;
   case SBT_NOBORDERS:
      // text is drawn without borders
      break;
   case SBT_OWNERDRAW:
      // Text is drawn by the parent window
      break;
   case SBT_POPOUT:
      // Text is drawn with a border to appear higher than the
      // plane of the status bar
      break;
}
```

See Also

Reference

CStatusBarCtrl Class
Hierarchy Chart
CStatusBarCtrl::SetText
CStatusBarCtrl::GetTextLength

Other Resources

CStatusBarCtrl Members

CStatusBarCtrl::GetTextLength

Retrieves the length, in characters, of the text from the given part of a status bar control.

```
int GetTextLength(
   int nPane,
   int* pType = NULL
) const;
```

Parameters

nPane

Zero-based index of the part from which to retrieve text.

pType

Pointer to an integer that receives the type information. The type can be one of these values:

- **0** The text is drawn with a border to appear lower than the plane of the status bar.
- **SBT_NOBORDERS** The text is drawn without borders.
- **SBT_OWNERDRAW** The text is drawn by the parent window.
- **SBT_POPOUT** The text is drawn with a border to appear higher than the plane of the status bar.

Return Value

The length, in characters, of the text.

Example

```
int nType;
int nLength = GetTextLength( 0, &nType );

switch( nType )
{
   case 0:
      // Text is drawn with a border to appear lower than the
      // plane of the status bar
      break;
   case SBT_NOBORDERS:
      // text is drawn without borders
      break;
   case SBT_OWNERDRAW:
      // Text is drawn by the parent window
      break;
   case SBT_POPOUT:
      // Text is drawn with a border to appear higher than the
      // plane of the status bar
      break;
}
```

See Also

Reference

CStatusBarCtrl Class
Hierarchy Chart
CStatusBarCtrl::GetText
CStatusBarCtrl::SetText

Other Resources

CStatusBarCtrl Members

CStatusBarCtrl::GetTipText

Retrieves the tooltip text for a pane in a status bar.

```
CString GetTipText(
   int nPane
) const;
```

Parameters

nPane

The zero-based index of status bar pane to receive the tooltip text.

Return Value

A CString object containing the text to be used in the tooltip.

Remarks

This member function implements the behavior of the Win32 message SB_GETTIPTEXT, as described in the *Platform SDK*.

Example

```
CString csPane0TipText = GetTipText(0);
```

See Also

Reference

CStatusBar Class
Hierarchy Chart
CStatusBarCtrl::SetTipText

Other Resources

CStatusBar Members

CStatusBarCtrl::IsSimple

Checks a status window control to determine if it is in simple mode.

```
BOOL IsSimple( ) const;
```

Return Value

Nonzero if the status window control is in simple mode; otherwise zero.

Remarks

This member function implements the behavior of the Win32 message SB_ISSIMPLE, as described in the *Platform SDK*.

See Also

Reference

CStatusBarCtrl Class
Hierarchy Chart
CStatusBarCtrl::SetSimple

Other Resources

CStatusBarCtrl Members

CStatusBarCtrl::SetBkColor

Sets the background color in a status bar.

```
COLORREF SetBkColor(
   COLORREF cr
);
```

Parameters

cr

COLORREF value that specifies the new background color. Specify the **CLR_DEFAULT** value to cause the status bar to use its default background color.

Return Value

A COLORREF value that represents the previous default background color.

Remarks

This member function implements the behavior of the Win32 message SB_SETBKCOLOR, as described in the *Platform SDK*.

Example

```
int CMyStatusBarCtrl::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CStatusBarCtrl::OnCreate(lpCreateStruct) == -1)
      return -1;

   SetBkColor( RGB(0,0,250) );

   HICON hIcon;
   VERIFY( hIcon = AfxGetApp()->LoadIcon(IDR_PANE_0_ICON) );
   VERIFY( SetIcon(0, hIcon) );

   return 0;
}
```

See Also

Reference

CStatusBar Class

Hierarchy Chart

Other Resources

CStatusBar Members

CStatusBarCtrl::SetIcon

Sets the icon for a pane in a status bar.

```
BOOL SetIcon(
   int nPane,
   HICON hIcon
);
```

Parameters

nPane

The zero-based index of the pane that will receive the icon. If this parameter is -1, the status bar is assumed to be a simple status bar.

hIcon

Handle to the icon to be set. If this value is **NULL**, the icon is removed from the part.

Return Value

Nonzero if successful; otherwise zero.

Remarks

This member function implements the behavior of the Win32 message SB_SETICON, as described in the *Platform SDK*.

Example

See the example for CStatusBarCtrl::SetBkColor.

See Also

Reference

CStatusBar Class

Hierarchy Chart

Other Resources

CStatusBar Members

CStatusBarCtrl::SetMinHeight

Sets the minimum height of a status bar control's drawing area.

```
void SetMinHeight(
   int nMin
);
```

Parameters

nMin

Minimum height, in pixels, of the control.

Remarks

The minimum height is the sum of *nMin* and twice the width, in pixels, of the vertical border of the status bar control.

Example

```
int CMyCStatusBarCtrl::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CStatusBarCtrl::OnCreate(lpCreateStruct) == -1)
      return -1;

   SetMinHeight(40);

   HICON hIcon;
   VERIFY(hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME));
   VERIFY(SetIcon(0, hIcon));

   return 0;
}
```

See Also

Reference

CStatusBarCtrl Class
Hierarchy Chart
CStatusBarCtrl::GetRect
CStatusBarCtrl::GetBorders

Other Resources

CStatusBarCtrl Members

CStatusBarCtrl::SetParts

Sets the number of parts in a status bar control and the coordinate of the right edge of each part.

```
BOOL SetParts(
   int nParts,
   int* pWidths
);
```

Parameters

nParts

Number of parts to set. The number of parts cannot be greater than 255.

pWidths

Address of an integer array having the same number of elements as parts specified by *nParts*. Each element in the array specifies the position, in client coordinates, of the right edge of the corresponding part. If an element is – 1, the position of the right edge for that part extends to the right edge of the control.

Return Value

Nonzero if successful; otherwise zero.

Example

```
const int nParts = 4;
CRect rect;

m_wndStatusBarCtrl.GetClientRect(&rect);
int widths[nParts] = { rect.right-300, rect.right-200, rect.right-100, -1 };

VERIFY( m_wndStatusBarCtrl.SetParts(nParts, widths) );
```

See Also

Reference

CStatusBarCtrl Class
Hierarchy Chart
CStatusBarCtrl::GetBorders
CStatusBarCtrl::GetParts

Other Resources

CStatusBarCtrl Members

CStatusBarCtrl::SetSimple

Specifies whether a status bar control displays simple text or displays all control parts set by a previous call to SetParts.

```
BOOL SetSimple(
   BOOL bSimple = TRUE
);
```

Parameters

bSimple

Display-type flag. If this parameter is **TRUE**, the control displays simple text; if it is **FALSE**, it displays multiple parts.

Return Value

Zero if an error occurs.

Remarks

If the status bar control is being changed from nonsimple to simple, or vice versa, the control is immediately redrawn.

See Also

Reference

CStatusBarCtrl Class
Hierarchy Chart
CStatusBarCtrl::SetParts

Other Resources

CStatusBarCtrl Members

CStatusBarCtrl::SetText

Sets the text in the given part of a status bar control.

```
BOOL SetText(
   LPCTSTR lpszText,
   int nPane,
   int nType
);
```

Parameters

lpszText

Address of a null-terminated string specifying the text to set. If *nType* is **SBT_OWNERDRAW**, *lpszText* represents 32 bits of data.

nPane

Zero-based index of the part to set. If this value is 255, the status bar control is assumed to be a simple control having only one part.

nType

Type of drawing operation. It can be one of these values:

- **0** The text is drawn with a border to appear lower than the plane of the status bar.
- **SBT_NOBORDERS** The text is drawn without borders.
- **SBT_OWNERDRAW** The text is drawn by the parent window.
- **SBT_POPOUT** The text is drawn with a border to appear higher than the plane of the status bar.

Return Value

Nonzero if successful; otherwise zero.

Remarks

The message invalidates the portion of the control that has changed, causing it to display the new text when the control next receives the **WM_PAINT** message.

Example

```
VERIFY( SetText("Text For Pane 1", 1, 0) );
```

See Also

Reference

CStatusBarCtrl Class
Hierarchy Chart
CStatusBarCtrl::GetText
CStatusBarCtrl::GetTextLength

Other Resources

CStatusBarCtrl Members

CStatusBarCtrl::SetTipText

Sets the tooltip text for a pane in a status bar.

```
void SetTipText(
   int nPane,
   LPCTSTR pszTipText
);
```

Parameters

nPane

The zero-based index of status bar pane to receive the tooltip text.

pszTipText

A pointer to a string containing the tooltip text.

Remarks

This member function implements the behavior of the Win32 message SB_SETTIPTEXT, as described in the *Platform SDK*.

Example

```
SetTipText(0, "This is Pane 0");
```

See Also

Reference

CStatusBar Class
Hierarchy Chart
CStatusBarCtrl::GetTipText

Other Resources

CStatusBar Members

CStdioFile Class

Represents a C run-time stream file as opened by the run-time function fopen.

```
class CStdioFile : public CFile
```

Remarks

Stream files are buffered and can be opened in either text mode (the default) or binary mode.

Text mode provides special processing for carriage return–linefeed pairs. When you write a newline character (0x0A) to a text-mode **CStdioFile** object, the byte pair (0x0D, 0x0A) is sent to the file. When you read, the byte pair (0x0D, 0x0A) is translated to a single 0x0A byte.

The CFile functions Duplicate, LockRange, and UnlockRange are not supported for **CStdioFile**.

If you call these functions on a **CStdioFile**, you will get a CNotSupportedException.

For more information on using **CStdioFile**, see the articles Files in MFC and File Handling in the *Run-Time Library Reference*.

Requirements

Header: afx.h

See Also

Reference

CFile Class
Hierarchy Chart
CFile Class
CFile::Duplicate
CFile::LockRange
CFile::UnlockRange
CNotSupportedException Class

Other Resources

CStdioFile Members

CStdioFile Members

Base Class Members

CObject Members

CFile Members

Data Members

m_pStream	Contains a pointer to an open file.

Construction

CStdioFile	Constructs a **CStdioFile** object from a path or file pointer.

Text Read/Write

ReadString	Reads a single line of text.
WriteString	Writes a single line of text.

Position

Seek	Positions the current file pointer.

See Also

Reference

CStdioFile Class

Hierarchy Chart

CStdioFile Member Functions

For information about the member functions in **CStdioFile**, see CStdioFile Members.

CStdioFile::CStdioFile

The default version of the constructor works in conjunction with the CFile::Open member function to test errors.

```
CStdioFile( );
CStdioFile(
   FILE* pOpenStream
);
CStdioFile(
   LPCTSTR lpszFileName,
   UINT nOpenFlags
);
```

Parameters

pOpenStream

Specifies the file pointer returned by a call to the C run-time function fopen.

lpszFileName

Specifies a string that is the path to the desired file. The path can be relative or absolute.

nOpenFlags

Sharing and access mode. Specifies the action to take when the file is opened. You can combine options by using the bitwise OR (**|**) operator. One access permission and a text-binary specifier are required; the **create** and **noInherit** modes are optional. See CFile::CFile for a list of mode options and other flags. In MFC version 3.0 and later, share flags are allowed.

Remarks

The one-parameter version constructs a **CStdioFile** object from a pointer to a file that is already open. Allowed pointer values include the predefined input/output file pointers stdin, **stdout**, or **stderr**.

The two-parameter version constructs a **CStdioFile** object and opens the corresponding operating-system file with the given path.

CFileException is thrown if the file cannot be opened or created.

Example

```
// example for CStdioFile::CStdioFile
char* pFileName = "test.dat";
CStdioFile f1;
if( !f1.Open( pFileName, CFile::modeCreate
       | CFile::modeWrite | CFile::typeText ) ) {
   #ifdef _DEBUG
      afxDump << "Unable to open file" << "\n";
   #endif
   exit( 1 );
}
CStdioFile f2( stdout );
TRY
{
   CStdioFile f3( pFileName,
      CFile::modeCreate | CFile::modeWrite | CFile::typeText );
}
CATCH( CFileException, e )
{
   #ifdef _DEBUG
      afxDump << "File could not be opened "
              << e->m_cause << "\n";
   #endif
}
END_CATCH
```

See Also

Reference

CStdioFile Class

Hierarchy Chart

Other Resources

CStdioFile Members

CStdioFile::ReadString

Reads text data into a buffer, up to a limit of *nMax*–1 characters, from the file associated with the **CStdioFile** object.

```
virtual LPTSTR ReadString(
   LPTSTR lpsz,
   UINT nMax
);
virtual BOOL ReadString(
   CString& rString
);
```

Parameters

lpsz

Specifies a pointer to a user-supplied buffer that will receive a null-terminated text string.

nMax

Specifies the maximum number of characters to read, not counting the terminating null character.

rString

A reference to a **CString** object that will contain the string when the function returns.

Return Value

A pointer to the buffer containing the text data. **NULL** if end-of-file was reached without reading any data; or if boolean, **FALSE** if end-of-file was reached without reading any data.

Remarks

Reading is stopped by the first newline character. If, in that case, fewer than *nMax*–1 characters have been read, a newline character is stored in the buffer. A null character ('\0') is appended in either case.

CFile::Read is also available for text-mode input, but it does not terminate on a carriage return–linefeed pair.

Note

The **CString** version of this function removes the `'\n'` if present; the **LPTSTR** version does not.

Example

```
// example for CStdioFile::ReadString
extern CStdioFile f;
char buf[100];

f.ReadString( buf, 99 );
```

See Also

Reference

CStdioFile Class
Hierarchy Chart
CStdioFile::WriteString
CFile::Read

Other Resources

CStdioFile Members

CStdioFile::Seek

Repositions the pointer in a previously opened file.

```
virtual ULONGLONG Seek(
   LONGLONG lOff,
   UINT nFrom
);
```

Parameters

lOff

Number of bytes to move the pointer.

nFrom

Pointer movement mode. Must be one of the following values:

- **CFile::begin**: Move the file pointer *lOff* bytes forward from the beginning of the file.
- **CFile::current**: Move the file pointer *lOff* bytes from the current position in the file.
- **CFile::end**: Move the file pointer *lOff* bytes from the end of the file. Note that *lOff* must be negative to seek into the existing file; positive values will seek past the end of the file.

Return Value

If the requested position is legal, **Seek** returns the new byte offset from the beginning of the file. Otherwise, the return value is undefined and a **CFileException** object is thrown.

Remarks

The **Seek** function permits random access to a file's contents by moving the pointer a specified amount, absolutely or relatively. No data is actually read during the seek. If the requested position is larger than the size of the file, the file length will be extended to that position, and no exception will be thrown.

When a file is opened, the file pointer is positioned at offset 0, the beginning of the file.

This implementation of **Seek** is based on the Run-Time Library (CRT) function **fseek**. There are several limits on the usage of **Seek** on streams opened in text mode. For more information, see fseek, _fseeki64.

Example

The following example shows how to use **Seek** to move the pointer 1000 bytes from the beginning of the `cfile` file. Note that **Seek** does not read data, so you must subsequently call CStdioFile::ReadString to read data.

```
extern CStdioFile cfile;
LONGLONG lOff = 1000;
ULONGLONG lActual;
lActual = cfile.Seek( lOff, CFile::begin );
```

See Also

Reference

CStdioFile Class
Hierarchy Chart
CFileException Class
CStdioFile::ReadString
CFile::Read

Other Resources

CStdioFile Members

CStdioFile::WriteString

Writes data from a buffer to the file associated with the **CStdioFile** object.

```
virtual void WriteString(
   LPCTSTR lpsz
);
```

Parameters

lpsz

Specifies a pointer to a buffer containing a null-terminated text string.

Remarks

The terminating null character ('\0') is not written to the file. Any newline character in *lpsz* is written to the file as a carriage return–linefeed pair.

WriteString throws an exception in response to several conditions, including the disk-full condition.

This is a text-oriented write function available to **CStdioFile** and its descendents, and to CArchive. CFile::Write is also available, but rather than terminating on a null character, it writes the requested number of bytes to the file.

Example

```
// example for CStdioFile::WriteString
extern CStdioFile f;
char buf[] = "test string";

f.WriteString( buf );
```

See Also

Reference

CStdioFile Class
Hierarchy Chart
CArchive::ReadString
CFile::Write

Other Resources

CStdioFile Members

CStdioFile Data Members

For information about the data members in **CStdioFile**, see CStdioFile Members.

CStdioFile::m_pStream

The **m_pStream** data member is the pointer to an open file as returned by the C run-time function **fopen**.

```
FILE* m_pStream;
```

Remarks

It is **NULL** if the file has never been opened or has been closed.

See Also

Reference

CStdioFile Class

Hierarchy Chart

Other Resources

CStdioFile Members

CStringArray Class

Supports arrays of **CString** objects.

```
class CStringArray : public CObject
```

Remarks

The member functions of **CStringArray** are similar to the member functions of class CObArray. Because of this similarity, you can use the **CObArray** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a return value, substitute a **CString** (not a **CString** pointer). Wherever you see a **CObject** pointer as a function parameter, substitute a **LPCTSTR**.

```
CObject* CObArray::GetAt( int <nIndex> ) const;
```

for example, translates to

```
CString CStringArray::GetAt( int <nIndex> ) const;
```

and

```
void SetAt( int <nIndex>, CObject* <newElement> )
```

translates to

```
void SetAt( int <nIndex>, LPCTSTR <newElement> )
```

CStringArray incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. If an array of **CString** objects is stored to an archive, either with an overloaded insertion operator or with the **Serialize** member function, each element is serialized in turn.

Note
Before using an array, use **SetSize** to establish its size and allocate memory for it. If you do not use **SetSize**, adding elements to your array causes it to be frequently reallocated and copied. Frequent reallocation and copying are inefficient and can fragment memory.

If you need a dump of individual string elements in the array, you must set the depth of the dump context to 1 or greater.

When a **CString** array is deleted, or when its elements are removed, string memory is freed as appropriate.

For more information on using **CStringArray**, see the article Collections.

Requirements

Header: afxcoll.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class

Hierarchy Chart

Other Resources

CStringArray Members

CStringArray Members

Base Class Members

CObject Members

CStringArray Members

Construction

CStringArray	Constructs an empty array for **CString** objects.

Bounds

GetCount	Gets number of elements in this array.
GetSize	Gets number of elements in this array.
GetUpperBound	Returns the largest valid index.
SetSize	Sets the number of elements to be contained in this array.

Operations

FreeExtra	Frees all unused memory above the current upper bound.
RemoveAll	Removes all the elements from this array.

Element Access

ElementAt	Returns a temporary reference to the element pointer within the array.
GetAt	Returns the value at a given index.
GetData	Allows access to elements in the array. Can be **NULL**.
SetAt	Sets the value for a given index; array not allowed to grow.

Growing the Array

Add	Adds an element to the end of the array; grows the array if necessary.
Append	Appends another array to the array; grows the array if necessary.
Copy	Copies anolther array to the array; grows the array if necessary.
SetAtGrow	Sets the value for a given index; grows the array if necessary.

Insertion/Removal

InsertAt	Inserts an element (or all the elements in another array) at a specified index.
IsEmpty	Determines if the array is empty
RemoveAt	Removes an element at a specific index.

Operators

operator []	Sets or gets the element at the specified index.

See Also

Reference

CStringArray Class

Hierarchy Chart

CStringList Class

Supports lists of **CString** objects.

```
class CStringList : public CObject
```

Remarks

All comparisons are done by value, meaning that the characters in the string are compared instead of the addresses of the strings.

The member functions of **CStringList** are similar to the member functions of class CObList. Because of this similarity, you can use the **CObList** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a return value, substitute a **CString** (not a **CString** pointer). Wherever you see a **CObject** pointer as a function parameter, substitute an **LPCTSTR**.

```
CObject*& CObList::GetHead() const;
```

for example, translates to

```
CString& CStringList::GetHead() const;
```

and

```
POSITION AddHead( CObject* <newElement> );
```

translates to

```
POSITION AddHead( LPCTSTR <newElement> );
```

CStringList incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. If a list of **CString** objects is stored to an archive, either with an overloaded insertion operator or with the **Serialize** member function, each **CString** element is serialized in turn.

If you need a dump of individual **CString** elements, you must set the depth of the dump context to 1 or greater.

When a **CStringList** object is deleted, or when its elements are removed, the **CString** objects are deleted as appropriate.

For more information on using **CStringList**, see the article Collections.

Requirements

Header: afxcoll.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

COLLECT Sample: Illustrates MFC Collection Classes

Reference

CObject Class

Hierarchy Chart

Other Resources

CStringList Members

CStringList Members

Base Class Members

CObject Members

CStringList Members

Note
The member functions of **CStringList** are similar to the member functions of class CObList. Because of this similarity, you can use the **CObList** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a return value, substitute a **CString** (not a **CString** pointer). Wherever you see a **CObject** pointer as a function parameter, substitute an **LPCTSTR**.

Construction

CStringList	Constructs an empty list for **CString** objects.

Head/Tail Access

GetHead	Returns the head element of the list (cannot be empty).
GetTail	Returns the tail element of the list (cannot be empty).

Operations

AddHead	Adds an element (or all the elements in another list) to the head of the list (makes a new head).
AddTail	Adds an element (or all the elements in another list) to the tail of the list (makes a new tail).
RemoveAll	Removes all the elements from this list.
RemoveHead	Removes the element from the head of the list.
RemoveTail	Removes the element from the tail of the list.

Iteration

GetHeadPosition	Returns the position of the head element of the list.
GetNext	Gets the next element for iterating.
GetPrev	Gets the previous element for iterating.
GetTailPosition	Returns the position of the tail element of the list.

Retrieval/Modification

GetAt	Gets the element at a given position.
RemoveAt	Removes an element from this list as specified by position.
SetAt	Sets the element at a given position.

Insertion

InsertAfter	Inserts a new element after a given position.
InsertBefore	Inserts a new element before a given position.

Searching

Find	Gets the position of an element specified by string value.
FindIndex	Gets the position of an element specified by a zero-based index.

Status

GetSize	Returns the number of elements in this list.
GetCount	Returns the number of elements in this list.
IsEmpty	Tests for the empty list condition (no elements).

See Also

Reference

CStringList Class
Hierarchy Chart

CSyncObject Class

A pure virtual class that provides functionality common to the synchronization objects in Win32.

```
class CSyncObject : public CObject
```

Remarks

The Microsoft Foundation Class Library provides several classes derived from **CSyncObject**. These are CEvent, CMutex, CCriticalSection, and CSemaphore.

For information on how to use the synchronization objects, see the article Multithreading: How to Use the Synchronization Classes.

Requirements

Header: afxmt.h

See Also

Reference

CObject Class

Hierarchy Chart

Other Resources

CSyncObject Members

CSyncObject Members

Base Class Members

CObject Members

CSyncObject Members

Construction

CSyncObject	Constructs a **CSyncObject** object.

Methods

Lock	Gains access to the synchronization object.
Unlock	Releases access to the synchronization object.

Attributes

operator HANDLE	Gains access to the synchronization object.

Data Members

m_hObject	The handle to the underlying synchronization object.

See Also

Reference

CSyncObject Class

Hierarchy Chart

CSyncObject Member Functions

For information about the member functions in **CSyncObject**, see CSyncObject Members.

CSyncObject::CSyncObject

Constructs a synchronization object with the supplied name.

```
explicit CSyncObject(
   LPCTSTR pstrName
);
virtual ~CSyncObject( );
```

Parameters

pstrName

The name of the object. If **NULL**, *pstrName* will be null.

See Also

Reference

CSyncObject Class

Hierarchy Chart

Other Resources

CSyncObject Members

CSyncObject::Lock

Call this function to gain access to the resource controlled by the synchronization object.

```
virtual BOOL Lock(
   DWORD dwTimeout = INFINITE
);
```

Parameters

dwTimeout

Specifies the amount of time in milliseconds to wait for the synchronization object to be available (signaled). If **INFINITE**, **Lock** will wait until the object is signaled before returning.

Return Value

Nonzero if the function was successful; otherwise 0.

Remarks

If the synchronization object is signaled, **Lock** will return successfully and the thread now owns the object. If the synchronization object is nonsignaled (unavailable), **Lock** will wait for the synchronization object to become signaled up to the number of milliseconds specified in the *dwTimeOut* parameter. If the synchronization object did not become signaled in the specified amount of time, **Lock** returns failure.

See Also

Reference

CSyncObject Class

Hierarchy Chart

Other Resources

CSyncObject Members

CSyncObject::m_hObject

The handle to the underlying synchronization object.

```
HANDLE m_hObject;
```

See Also

Reference

CSyncObject Class

Hierarchy Chart

Other Resources

CSyncObject Members

CSyncObject::operator HANDLE

Use this operator to get the handle of the **CSyncObject** object.

```
operator HANDLE( ) const;
```

Return Value

If successful, the handle of the synchronization object; otherwise, **NULL**.

Remarks

You can use the handle to call Windows APIs directly.

See Also

Reference

CSyncObject Class

Hierarchy Chart

Other Resources

CSyncObject Members

CSyncObject::Unlock

The declaration of **Unlock** with no parameters is a pure virtual function, and must be overridden by all classes deriving from **CSyncObject**.

```
virtual BOOL Unlock( ) = 0;
virtual BOOL Unlock(
   LONG lCount,
   LPLONG lpPrevCount = NULL
);
```

Parameters

lCount

Not used by default implementation.

lpPrevCount

Not used by default implementation.

Return Value

Default implementation always returns **TRUE**.

Remarks

The default implementation of the declaration with two parameters always returns **TRUE**. This function is called to release access to the synchronization object owned by the calling thread. The second declaration is provided for synchronization objects such as semaphores that allow more than one access of a controlled resource.

See Also

Reference

CSyncObject Class

Hierarchy Chart

Other Resources

CSyncObject Members

CTabCtrl Class

Provides the functionality of the Windows common tab control.

```
class CTabCtrl : public CWnd
```

Remarks

A "tab control" is analogous to the dividers in a notebook or the labels in a file cabinet. By using a tab control, an application can define multiple pages for the same area of a window or dialog box. Each page consists of a set of information or a group of controls that the application displays when the user selects the corresponding tab. A special type of tab control displays tabs that look like buttons. Clicking a button should immediately perform a command instead of displaying a page.

This control (and therefore the **CTabCtrl** class) is available only to programs running under Windows 95/98 and Windows NT version 3.51 and later.

For more information on using **CTabCtrl**, see Controls and Using CTabCtrl.

Requirements

Header: afxcmn.h

See Also

Reference

CWnd Class
Hierarchy Chart
CHeaderCtrl Class
CListCtrl Class

Other Resources

CTabCtrl Members

CTabCtrl Members

Base Class Members

Construction

Attributes

Operations

Overridables

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Construction

Create	Creates a tab control and attaches it to an instance of a **CTabCtrl** object.
CreateEx	Creates a tab control with the specified Windows extended styles and attaches it to an instance of a **CTabCtrl** object.
CTabCtrl	Constructs a **CTabCtrl** object.

Attributes

GetCurFocus	Retrieves the tab with the current focus of a tab control.
GetCurSel	Determines the currently selected tab in a tab control.
GetExtendedStyle	Retrieves the extended styles that are currently in use for the tab control.
GetImageList	Retrieves the image list associated with a tab control.
GetItem	Retrieves information about a tab in a tab control.
GetItemCount	Retrieves the number of tabs in the tab control.
GetItemRect	Retrieves the bounding rectangle for a tab in a tab control.
GetItemState	Retrieves the state of the indicated tab control item.
GetRowCount	Retrieves the current number of rows of tabs in a tab control.
GetToolTips	Retrieves the handle of the tool tip control associated with a tab control.
SetCurFocus	Sets the focus to a specified tab in a tab control.
SetCurSel	Selects a tab in a tab control.
SetExtendedStyle	Sets the extended styles for a tab control.
SetImageList	Assigns an image list to a tab control.
SetItem	Sets some or all of a tab's attributes.

SetItemExtra	Sets the number of bytes per tab reserved for application-defined data in a tab control.
SetItemSize	Sets the width and height of an item.
SetItemState	Sets the state of the indicated tab control item.
SetMinTabWidth	Sets the minimum width of items in a tab control.
SetPadding	Sets the amount of space (padding) around each tab's icon and label in a tab control.
SetToolTips	Assigns a tool tip control to a tab control.

Operations

AdjustRect	Calculates a tab control's display area given a window rectangle, or calculates the window rectangle that would correspond to a given display area.
DeleteAllItems	Removes all items from a tab control.
DeleteItem	Removes an item from a tab control.
DeselectAll	Resets items in a tab control, clearing any that were pressed.
HighlightItem	Sets the highlight state of a tab item.
HitTest	Determines which tab, if any, is at a specified screen position.
InsertItem	Inserts a new tab in a tab control.
RemoveImage	Removes an image from a tab control's image list.

Overridables

DrawItem	Draws a specified item of a tab control.

See Also

Reference

CTabCtrl Class
Hierarchy Chart

CTabCtrl Member Functions

For information about the member functions in **CTabCtrl**, see CTabCtrl Members.

CTabCtrl::AdjustRect

Calculates a tab control's display area given a window rectangle, or calculates the window rectangle that would correspond to a given display area.

```
void AdjustRect(
  BOOL bLarger,
  LPRECT lpRect
);
```

Parameters

bLarger

Indicates which operation to perform. If this parameter is **TRUE**, *lpRect* specifies a display rectangle and receives the corresponding window rectangle. If this parameter is **FALSE**, *lpRect* specifies a window rectangle and receives the corresponding display rectangle.

lpRect

Pointer to a RECT structure that specifies the given rectangle and receives the calculated rectangle.

Example

```
void CDialogDlg::OnSize(UINT nType, int cx, int cy)
{
   CDialog::OnSize(nType, cx, cy);

   if(m_tabCtrl.m_hWnd == NULL)
      return;      // Return if window is not created yet.

   RECT lpRect;

   // Get size of dialog window.
   GetClientRect(&lpRect);


   // Adjust the rectangle to fit the tab control into the
   // dialog's client rectangle.
   m_tabCtrl.AdjustRect(FALSE, &lpRect);

   // Move the tab control to the new position and size.
   m_tabCtrl.MoveWindow(&lpRect, TRUE);
}
```

See Also

Reference

CTabCtrl Class
Hierarchy Chart
CTabCtrl::SetItemSize
CTabCtrl::GetItemRect
CTabCtrl::AdjustRect

Other Resources

CTabCtrl Members

CTabCtrl::Create

Creates a tab control and attaches it to an instance of a **CTabCtrl** object.

```
virtual BOOL Create(
  DWORD dwStyle,
  const RECT& rect,
  CWnd* pParentWnd,
  UINT nID
);
```

Parameters

dwStyle

Specifies the tab control's style. Apply any combination of tab control styles, described in the *Platform SDK*. See **Remarks** for a list of window styles that you can also apply to the control.

rect

Specifies the tab control's size and position. It can be either a CRect object or a RECT structure.

pParentWnd

Specifies the tab control's parent window, usually a **CDialog**. It must not be **NULL**.

nID

Specifies the tab control's ID.

Return Value

TRUE if initialization of the object was successful; otherwise **FALSE**.

Remarks

You construct a **CTabCtrl** object in two steps. First, call the constructor, and then call **Create**, which creates the tab control and attaches it to the **CTabCtrl** object.

In addition to tab control styles, you can apply the following window styles to a tab control:

- **WS_CHILD** Creates a child window that represents the tab control. Cannot be used with the **WS_POPUP** style.
- **WS_VISIBLE** Creates a tab control that is initially visible.
- **WS_DISABLED** Creates a window that is initially disabled.
- **WS_GROUP** Specifies the first control of a group of controls in which the user can move from one control to the next with the arrow keys. All controls defined with the **WS_GROUP** style after the first control belong to the same group. The next control with the **WS_GROUP** style ends the style group and starts the next group (that is, one group ends where the next begins).
- **WS_TABSTOP** Specifies one of any number of controls through which the user can move by using the TAB key. The TAB key moves the user to the next control specified by the **WS_TABSTOP** style.

To create a tab control with extended window styles, call CTabCtrl::CreateEx instead of **Create**.

Example

```
// Assuming you have a member variable m_pTabCtrl, that is a CTabCtrl
// pointer, you can use the following to create a tab control.

m_pTabCtrl->Create(TCS_TABS | TCS_FIXEDWIDTH | WS_CHILD | WS_VISIBLE,
   &rect, this, 0x1006);

// This creates a tab control with the given styles, and with
// an ID of 0x1006.
```

See Also

Reference

CTabCtrl Class

Hierarchy Chart

CTabCtrl::CTabCtrl

Other Resources

CTabCtrl Members

CTabCtrl::CreateEx

Creates a control (a child window) and associates it with the **CTabCtrl** object.

```
virtual BOOL CreateEx(
  DWORD dwExStyle,
  DWORD dwStyle,
  const RECT& rect,
  CWnd* pParentWnd,
  UINT nID
);
```

Parameters

dwExStyle

Specifies the extended style of the control being created. For a list of extended Windows styles, see the *dwExStyle* parameter for CreateWindowEx in the *Platform SDK*.

dwStyle

Specifies the tab control's style. Apply any combination of tab control styles, described in the *Platform SDK*. See **Remarks** in Create for a list of window styles that you can also apply to the control.

rect

A reference to a RECT structure describing the size and position of the window to be created, in client coordinates of *pParentWnd*.

pParentWnd

A pointer to the window that is the control's parent.

nID

The control's child-window ID.

Return Value

Nonzero if successful otherwise 0.

Remarks

Use **CreateEx** instead of Create to apply extended Windows styles, specified by the Windows extended style preface **WS_EX_**.

CreateEx creates the control with the extended Windows styles specified by *dwExStyle*. Set extended styles specific to a control using SetExtendedStyle. For example, use **CreateEx** to set such styles as **WS_EX_CONTEXTHELP**, but use **SetExtendedStyle** to set such styles as **TCS_EX_FLATSEPARATORS**. For more information, see the styles described in Tab Control Extended Styles in the *Platform SDK*.

See Also

Reference

CTabCtrl Class
Hierarchy Chart
CTabCtrl::CTabCtrl

Other Resources

CTabCtrl Members

CTabCtrl::CTabCtrl

Constructs a **CTabCtrl** object.

```
CTabCtrl( );
```

See Also

Reference

CTabCtrl Class
Hierarchy Chart
CTabCtrl::Create
CTabCtrl::CreateEx

Other Resources

CTabCtrl Members

CTabCtrl::DeleteAllItems

Removes all items from a tab control.

```
BOOL DeleteAllItems( );
```

Return Value

Nonzero if successful; otherwise 0.

See Also

Reference

CTabCtrl Class
Hierarchy Chart
CTabCtrl::DeleteItem

Other Resources

CTabCtrl Members

CTabCtrl::DeleteItem

Removes the specified item from a tab control.

```
BOOL DeleteItem(
  int nItem
);
```

Parameters

nItem

Zero-based value of the item to delete.

Return Value

Nonzero if successful; otherwise 0.

Example

```
// This example assumes that there is a CTabCtrl member of the
// CDialogDlg class named m_tabCtrl.  On a button handler
// called OnDeleteItem of the dialog box the tab control will
// delete the 0 indexed item.

void CDialogDlg::OnDeleteItem()
{
   // Delete the first item in the tab control.
   m_tabCtrl.DeleteItem(0);
}
```

See Also

Reference

CTabCtrl Class

Hierarchy Chart

CTabCtrl::DeleteAllItems

Other Resources

CTabCtrl Members

CTabCtrl::DeselectAll

Resets items in a tab control, clearing any that were pressed.

```
void DeselectAll(
  BOOL fExcludeFocus
);
```

Parameters

fExcludeFocus

Flag that specifies the scope of the item deselection. If this parameter is set to **FALSE**, all tab buttons will be reset. If it is set to **TRUE**, then all tab items except for the one currently selected will be reset.

Remarks

This member function implements the behavior of the Win32 message, TCM_DESELECTALL, as described in the *Platform SDK*.

See Also

Reference

CTabCtrl Class

Hierarchy Chart

Other Resources

CTabCtrl Members

CTabCtrl::DrawItem

Called by the framework when a visual aspect of an owner-draw tab control changes.

```
virtual void DrawItem(
  LPDRAWITEMSTRUCT lpDrawItemStruct
);
```

Parameters

lpDrawItemStruct

A pointer to a DRAWITEMSTRUCT structure describing the item to be painted.

Remarks

The **itemAction** member of the **DRAWITEMSTRUCT** structure defines the drawing action that is to be performed.

By default, this member function does nothing. Override this member function to implement drawing for an owner-draw **CTabCtrl** object.

The application should restore all graphics device interface (GDI) objects selected for the display context supplied in *lpDrawItemStruct* before this member function terminates.

See Also

Reference

CTabCtrl Class
Hierarchy Chart
CWnd::OnDrawItem

Other Resources

CTabCtrl Members

CTabCtrl::GetCurFocus

Retrieves the index of the tab with the current focus.

```
int GetCurFocus( ) const;
```

Return Value

The zero-based index of the tab with the current focus.

See Also

Reference

CTabCtrl Class
Hierarchy Chart
CTabCtrl::GetCurSel

Other Resources

CTabCtrl Members

CTabCtrl::GetCurSel

Retrieves the currently selected tab in a tab control.

```
int GetCurSel( ) const;
```

Return Value

Zero-based index of the selected tab if successful or – 1 if no tab is selected.

See Also

Reference

CTabCtrl Class
Hierarchy Chart
CTabCtrl::SetCurSel
CTabCtrl::GetCurFocus

Other Resources

CTabCtrl Members

CTabCtrl::GetExtendedStyle

Retrieves the extended styles that are currently in use for the tab control.

```
DWORD GetExtendedStyle( );
```

Return Value

Represents the extended styles currently in use for the tab control. This value is a combination of tab control extended styles, as described in the *Platform SDK*.

Remarks

This member function implements the behavior of the Win32 message TCM_GETEXTENDEDSTYLE, as described in the *Platform SDK*.

See Also

Reference

CTabCtrl Class
Hierarchy Chart
CTabCtrl::SetExtendedStyle

Other Resources

CTabCtrl Members

CTabCtrl::GetImageList

Retrieves the image list associated with a tab control.

```
CImageList* GetImageList( ) const;
```

Return Value

A pointer tof the image list of the tab control if successful; otherwise **NULL**.

See Also

Reference

CTabCtrl Class
Hierarchy Chart
CTabCtrl::SetImageList
CImageList Class

Other Resources

CTabCtrl Members

CTabCtrl::GetItem

Retrieves information about a tab in a tab control.

```
BOOL GetItem(
  int nItem,
  TCITEM* pTabCtrlItem
) const;
```

Parameters

nItem

Zero-based index of the tab.

pTabCtrlItem

Pointer to a TCITEM structure, used to specify the information to retrieve. Also used to receive information about the tab. This structure is used with the **InsertItem**, **GetItem**, and **SetItem** member functions.

Return Value

Returns **TRUE** if successful; **FALSE** otherwise.

Remarks

When the message is sent, the **mask** member specifies which attributes to return. If the **mask** member specifies the **TCIF_TEXT** value, the **pszText** member must contain the address of the buffer that receives the item text and the **cchTextMax** member must specify the size of the buffer.

mask

Value specifying which **TCITEM** structure members to retrieve or set. This member can be zero or a combination of the following values:

- **TCIF_TEXT** The **pszText** member is valid.
- **TCIF_IMAGE** The **iImage** member is valid.
- **TCIF_PARAM** The **lParam** member is valid.
- **TCIF_RTLREADING** The text of **pszText** is displayed using right-to-left reading order on Hebrew or Arabic systems.
- **TCIF_STATE** The **dwState** member is valid.

pszText

Pointer to a null-terminated string containing the tab text if the structure contains information about a tab. If the structure is receiving information, this member specifies the address of the buffer that receives the tab text.

cchTextMax

Size of the buffer pointed to by **pszText**. This member is ignored if the structure is not receiving information.

iImage

Index into the tab control's image list, or – 1 if there is no image for the tab.

lParam

Application-defined data associated with the tab. If there are more than four bytes of application-defined data per tab, an application must define a structure and use it instead of the **TCITEM** structure. The first member of the application-defined structure must be a TCITEMHEADERstructure. The **TCITEMHEADER** structure is identical to the **TCITEM** structure, but without the **lParam** member. The difference between the size of your structure and the size of the **TCITEMHEADER** structure should equal the number of extra bytes per tab.

Example

```
// In this example a CTabCtrl data member, m_tabCtrl, changes the
```

```
// text of the tabs in the tab control.  A call to GetItem is used
// to get the current text, and then the text is changed.  A call
// to SetItem is used to update the tab with the new text.

void CDialogDlg::OnChangeItem()
{
   TCITEM tcItem;
   CString pszString;

   //  Get text for the tab item.
   GetDlgItemText(IDC_ITEM_TEXT, pszString);

   //  Get the current tab item text.
   char buffer[256] = {0};
   tcItem.pszText = buffer;
   tcItem.cchTextMax = 256;
   tcItem.mask = TCIF_TEXT;
   m_tabCtrl.GetItem(0, &tcItem);

   //  Set the new text for the item.
   tcItem.pszText = pszString.GetBuffer(256);

   //  Set the item in the tab control.
   m_tabCtrl.SetItem(0, &tcItem);

}
```

See Also

Reference

CTabCtrl Class

Hierarchy Chart

CTabCtrl::InsertItem

CTabCtrl::SetItem

Other Resources

CTabCtrl Members

CTabCtrl::GetItemCount

Retrieves the number of tabs in the tab control.

```
int GetItemCount( ) const;
```

Return Value

Number of items in the tab control.

Example

See the example for CPropertySheet::GetTabControl.

See Also

Reference

CTabCtrl Class
Hierarchy Chart
CTabCtrl::GetItem
CTabCtrl::SetItem

Other Resources

CTabCtrl Members

CTabCtrl::GetItemRect

Retrieves the bounding rectangle for the specified tab in a tab control.

```
BOOL GetItemRect(
  int nItem,
  LPRECT lpRect
) const;
```

Parameters

nItem

Zero-based index of the tab item.

lpRect

Pointer to a RECT structure that receives the bounding rectangle of the tab. These coordinates use the viewport's current mapping mode.

Return Value

Nonzero if successful; otherwise 0.

Example

See the example for CPropertySheet::GetTabControl.

See Also

Reference

CTabCtrl Class
Hierarchy Chart
CTabCtrl::SetItemSize
CTabCtrl::AdjustRect

Other Resources

CTabCtrl Members

CTabCtrl::GetItemState

Retrieves the state of the tab control item identified by *nItem*.

```
DWORD GetItemState(
   int nItem,
   DWORD dwMask
) const;
```

Parameters

nItem

The zero-based index number of the item for which to retrieve state information.

dwMask

Mask specifying which of the item's state flags to return. For a list of values, see the mask member of the TCITEM structure, as described in the *Platform SDK*.

Return Value

A reference to a **DWORD** value receiving the state information. Can be one of the following values:

Value	Description
TCIS_BUTTONPRESSED	The tab control item is selected.
TCIS_HIGHLIGHTED	The tab control item is highlighted, and the tab and text are drawn using the current highlight color. When using highlight color, this will be a true interpolation, not a dithered color.

Remarks

An item's state is specified by the **dwState** member of the **TCITEM** structure.

See Also

Reference

CTabCtrl Class

Hierarchy Chart

CTabCtrl::SetItemState

Other Resources

CTabCtrl Members

CTabCtrl::GetRowCount

Retrieves the current number of rows in a tab control.

```
int GetRowCount( ) const;
```

Return Value

The number of rows of tabs in the tab control.

Remarks

Only tab controls that have the **TCS_MULTILINE** style can have multiple rows of tabs.

See Also

Reference

CTabCtrl Class

Hierarchy Chart

CTabCtrl::Create

Other Resources

CTabCtrl Members

CTabCtrl::GetToolTips

Retrieves the handle of the tool tip control associated with a tab control.

```
CToolTipCtrl* GetToolTips( ) const;
```

Return Value

Handle of the tool tip control if successful; otherwise **NULL**.

Remarks

A tab control creates a tool tip control if it has the **TCS_TOOLTIPS** style. You can also assign a tool tip control to a tab control by using the **SetToolTips** member function.

See Also

Reference

CTabCtrl Class
Hierarchy Chart
CTabCtrl::SetToolTips
CTabCtrl::Create

Other Resources

CTabCtrl Members

CTabCtrl::HighlightItem

Sets the highlight state of a tab item.

```
BOOL HighlightItem(
  int idItem,
  BOOL fHighlight = TRUE
);
```

Parameters

idItem

Zero-based index of a tab control item.

fHighlight

Value specifying the highlight state to be set. If this value is **TRUE**, the tab is highlighted; if **FALSE**, the tab is set to its default state.

Return Value

Nonzero if successful; otherwise zero.

Remarks

This member function implements the Win32 message TCM_HIGHLIGHTITEM, as described in the *Platform SDK*.

See Also

Reference

CTabCtrl Class

Hierarchy Chart

Other Resources

CTabCtrl Members

CTabCtrl::HitTest

Determines which tab, if any, is at the specified screen position.

```
int HitTest(
  TCHITTESTINFO* pHitTestInfo
) const;
```

Parameters

pHitTestInfo

Pointer to a TCHITTESTINFO structure, as described in the *Platform SDK*, which specifies the screen position to test.

Return Value

Returns the zero-based index of the tab or – 1 if no tab is at the specified position.

See Also

Reference

CTabCtrl Class

Hierarchy Chart

Other Resources

CTabCtrl Members

CTabCtrl::InsertItem

Inserts a new tab in an existing tab control.

```
LONG InsertItem(
  int nItem,
  TCITEM* pTabCtrlItem
);
LONG InsertItem(
  int nItem,
  LPCTSTR lpszItem
);
LONG InsertItem(
  int nItem,
  LPCTSTR lpszItem,
  int nImage
);
LONG InsertItem(
  UINT nMask,
  int nItem,
  LPCTSTR lpszItem,
  int nImage,
  LPARAM lParam
);
LONG InsertItem(
   UINT nMask,
   int nItem,
   LPCTSTR lpszItem,
   int nImage,
   LPARAM lParam,
   DWORD dwState,
   DWORD dwStateMask
);
```

Parameters

nItem

Zero-based index of the new tab.

pTabCtrlItem

Pointer to a TCITEM structure that specifies the attributes of the tab.

lpszItem

Address of a null-terminated string that contains the text of the tab.

nImage

The zero-based index of an image to insert from an image list.

nMask

Specifies which **TCITEM** structure attributes to set. Can be zero or a combination of the following values:

- **TCIF_TEXT** The **pszText** member is valid.
- **TCIF_IMAGE** The **iImage** member is valid.
- **TCIF_PARAM** The **lParam** member is valid.
- **TCIF_RTLREADING** The text of **pszText** is displayed using right-to-left reading order on Hebrew or Arabic systems.
- **TCIF_STATE** The **dwState** member is valid.

lParam

Application-defined data associated with the tab.

dwState

Specifies values for the item's states. For more information, see TCITEM in the *Platform SDK*.

dwStateMask

Specifies which states are to be set. For more information, see TCITEM in the *Platform SDK*.

Return Value

Zero-based index of the new tab if successful; otherwise – 1.

Example

```
BOOL CDialogDlg::OnInitDialog()
{
   // Other initialization.

   TCITEM tcItem;
   tcItem.mask = TCIF_TEXT;
   tcItem.pszText = _T("Tab #1");

   m_tabCtrl.InsertItem(0, &tcItem);

   return TRUE;  // Return TRUE unless you set the focus to a control.
}
```

See Also

Reference

CTabCtrl Class
Hierarchy Chart
CTabCtrl::GetItem
CTabCtrl::SetItem

Other Resources

CTabCtrl Members

CTabCtrl::RemoveImage

Removes the specified image from a tab control's image list.

```
void RemoveImage(
  int nImage
);
```

Parameters

nImage

Zero-based index of the image to remove.

Remarks

The tab control updates each tab's image index so that each tab remains associated with the same image.

See Also

Reference

CTabCtrl Class
Hierarchy Chart
CTabCtrl::GetImageList
CTabCtrl::SetImageList

Other Resources

CTabCtrl Members

CTabCtrl::SetCurFocus

Sets the focus to a specified tab in a tab control.

```
void SetCurFocus(
  int nItem
);
```

Parameters

nItem

Specifies the index of the tab that gets the focus.

Remarks

This member function implements the behavior of the Win32 message TCM_SETCURFOCUS, as described in the *Platform SDK*.

See Also

Reference

CTabCtrl Class

Hierarchy Chart

CTabCtrl::SetCurSel

Other Resources

CTabCtrl Members

CTabCtrl::SetCurSel

Selects a tab in a tab control.

```
int SetCurSel(
  int nItem
);
```

Parameters

nItem

The zero-based index of the item to be selected.

Return Value

Zero-based index of the previously selected tab if successful, otherwise – 1.

Remarks

A tab control does not send a **TCN_SELCHANGING** or **TCN_SELCHANGE** notification message when a tab is selected using this function. These notifications are sent, using **WM_NOTIFY**, when the user clicks or uses the keyboard to change tabs.

See Also

Reference

CTabCtrl Class
Hierarchy Chart
CTabCtrl::GetCurSel
CTabCtrl::GetCurFocus

Other Resources

CTabCtrl Members

CTabCtrl::SetExtendedStyle

Sets the extended styles for a tab control.

```
DWORD SetExtendedStyle(
  DWORD dwNewStyle,
  DWORD dwExMask = 0
);
```

Parameters

dwNewStyle

Value specifying a combination of tab control extended styles.

dwExMask

A **DWORD** value that indicates which styles in *dwNewStyle* are to be affected. Only the extended styles in *dwExMask* will be changed. All other styles will be maintained as is. If this parameter is zero, then all of the styles in *dwNewStyle* will be affected.

Return Value

A **DWORD** value that contains the previous tab control extended styles, as described in the *Platform SDK*.

Return Value

This member function implements the behavior of the Win32 message TCM_SETEXTENDEDSTYLE, as described in the *Platform SDK*.

See Also

Reference

CTabCtrl Class
Hierarchy Chart
CTabCtrl::GetExtendedStyle
CTabCtrl::CreateEx

Other Resources

CTabCtrl Members

CTabCtrl::SetImageList

Assigns an image list to a tab control.

```
CImageList * SetImageList(
  CImageList * pImageList
);
```

Parameters

pImageList

Pointer to the image list to be assigned to the tab control.

Return Value

Returns a pointer to the previous image list or **NULL** if there is no previous image list.

See Also

Reference

CTabCtrl Class
Hierarchy Chart
CTabCtrl::GetImageList
CImageList Class

Other Resources

CTabCtrl Members

CTabCtrl::SetItem

Sets some or all of a tab's attributes.

```
BOOL SetItem(
  int nItem,
  TCITEM* pTabCtrlItem
);
```

Parameters

nItem

Zero-based index of the item.

pTabCtrlItem

Pointer to a TCITEM structure that contains the new item attributes. The **mask** member specifies which attributes to set. If the **mask** member specifies the **TCIF_TEXT** value, the **pszText** member is the address of a null-terminated string and the **cchTextMax** member is ignored.

Return Value

Nonzero if successful; otherwise 0.

Example

See the example for GetItem.

See Also

Reference

CTabCtrl Class
Hierarchy Chart
CTabCtrl::InsertItem
CTabCtrl::GetItem

Other Resources

CTabCtrl Members

CTabCtrl::SetItemExtra

Sets the number of bytes per tab reserved for application-defined data in a tab control.

```
BOOL SetItemExtra(
  int nBytes
);
```

Parameters

nBytes

The number of extra bytes to set.

Return Value

Nonzero if successful; otherwise zero.

Remarks

This member function implements the behavior of the Win32 message TCM_SETITEMEXTRA, as described in the *Platform SDK*.

See Also

Reference

CTabCtrl Class

Hierarchy Chart

CTabCtrl::SetItem

Other Resources

CTabCtrl Members

CTabCtrl::SetItemSize

Sets the width and height of the tab control items.

```
CSize SetItemSize(
  CSize size
);
```

Parameters

size

The new width and height, in pixels, of the tab control items.

Return Value

Returns the old width and height of the tab control items.

See Also

Reference

CTabCtrl Class
Hierarchy Chart
CTabCtrl::AdjustRect
CTabCtrl::GetItemRect
CTabCtrl::SetItemSize

Other Resources

CTabCtrl Members

CTabCtrl::SetItemState

Sets the state of the tab control item identified by *nItem*.

```
BOOL SetItemState(
  int nItem,
  DWORD dwMask,
  DWORD dwState
);
```

Parameters

nItem

The zero-based index number of the item for which to set state information.

dwMask

Mask specifying which of the item's state flags to set. For a list of values, see the mask member of the TCITEM structure, as described in the *Platform SDK*.

dwState

A reference to a **DWORD** value containing the state information. Can be one of the following values:

Value	Description
TCIS_BUTTONPRESSED	The tab control item is selected.
TCIS_HIGHLIGHTED	The tab control item is highlighted, and the tab and text are drawn using the current highlight color. When using highlight color, this will be a true interpolation, not a dithered color.

Return Value

Nonzero if successful; otherwise 0.

See Also

Reference

CTabCtrl Class

Hierarchy Chart

CTabCtrl::GetItemState

Other Resources

CTabCtrl Members

CTabCtrl::SetMinTabWidth

Sets the minimum width of items in a tab control.

```
int SetMinTabWidth(
  int cx
);
```

Parameters

cx

Minimum width to be set for a tab control item. If this parameter is set to -1, the control will use the default tab width.

Return Value

The previous minimum tab width.

Return Value

This member function implements the behavior of the Win32 message TCM_SETMINTABWIDTH, as described in the *Platform SDK*.

See Also

Reference

CTabCtrl Class

Hierarchy Chart

Other Resources

CTabCtrl Members

CTabCtrl::SetPadding

Sets the amount of space (padding) around each tab's icon and label in a tab control.

```
void SetPadding(
  CSize size
);
```

Parameters

size

Sets the amount of space (padding) around each tab's icon and label in a tab control.

See Also

Reference

CTabCtrl Class

Hierarchy Chart

Other Resources

CTabCtrl Members

CTabCtrl::SetToolTips

Assigns a tool tip control to a tab control.

```
void SetToolTips(
  CToolTipCtrl* pWndTip
);
```

Parameters

pWndTip

Handle of the tool tip control.

Remarks

You can get the tool tip control associated with a tab control by making a call to **GetToolTips**.

Example

See the example for CPropertySheet::GetTabControl.

See Also

Reference

CTabCtrl Class

Hierarchy Chart

CTabCtrl::GetToolTips

Other Resources

CTabCtrl Members

CToolBar Class

Control bars that have a row of bitmapped buttons and optional separators.

```
class CToolBar : public CControlBar
```

Remarks

The buttons can act like pushbuttons, check-box buttons, or radio buttons. **CToolBar** objects are usually embedded members of frame-window objects derived from the class CFrameWnd or CMDIFrameWnd.

CToolBar::GetToolBarCtrl, a member function new to MFC 4.0, allows you to take advantage of the Windows common control's support for toolbar customization and additional functionality. **CToolBar** member functions give you most of the functionality of the Windows common controls; however, when you call **GetToolBarCtrl**, you can give your toolbars even more of the characteristics of Windows 95/98 toolbars. When you call **GetToolBarCtrl**, it will return a reference to a **CToolBarCtrl** object. See CToolBarCtrl for more information about designing toolbars using Windows common controls. For more general information about common controls, see Common Controls in the *Platform SDK*.

Visual C++ provides you with two methods to create a toolbar. To create a toolbar resource using the Resource Editor, follow these steps:

1. Create a toolbar resource.
2. Construct the **CToolBar** object.
3. Call the Create (or CreateEx) function to create the Windows toolbar and attach it to the **CToolBar** object.
4. Call LoadToolBar to load the toolbar resource.

Otherwise, follow these steps:

1. Construct the **CToolBar** object.
2. Call the Create (or CreateEx) function to create the Windows toolbar and attach it to the **CToolBar** object.
3. Call LoadBitmap to load the bitmap that contains the toolbar button images.
4. Call SetButtons to set the button style and associate each button with an image in the bitmap.

All the button images in the toolbar are taken from one bitmap, which must contain one image for each button. All images must be the same size; the default is 16 pixels wide and 15 pixels high. Images must be side by side in the bitmap.

The **SetButtons** function takes a pointer to an array of control IDs and an integer that specifies the number of elements in the array. The function sets each button's ID to the value of the corresponding element of the array and assigns each button an image index, which specifies the position of the button's image in the bitmap. If an array element has the value **ID_SEPARATOR**, no image index is assigned.

The order of the images in the bitmap is typically the order in which they are drawn on the screen, but you can use the SetButtonInfo function to change the relationship between image order and drawing order.

All buttons in a toolbar are the same size. The default is 24 x 22 pixels, in accordance with *Windows Interface Guidelines for Software Design*. Any additional space between the image and button dimensions is used to form a border around the image.

Each button has one image. The various button states and styles (pressed, up, down, disabled, disabled down, and indeterminate) are generated from that one image. Although bitmaps can be any color, you can achieve the best results with images in black and shades of gray.

Toolbar buttons imitate pushbuttons by default. However, toolbar buttons can also imitate check-box buttons or radio buttons. Check-box buttons have three states: checked, cleared, and indeterminate. Radio buttons have only two states: checked and cleared.

To set an individual button or separator style without pointing to an array, call GetButtonStyle to retrieve the style, and then call SetButtonStyle instead of **SetButtons**. **SetButtonStyle** is most useful when you want to change a button's style at run time.

To assign text to appear on a button, call GetButtonText to retrieve the text to appear on the button, and then call SetButtonText

to set the text.

To create a check-box button, assign it the style **TBBS_CHECKBOX** or use a **CCmdUI** object's **SetCheck** member function in an **ON_UPDATE_COMMAND_UI** handler. Calling **SetCheck** turns a pushbutton into a check-box button. Pass **SetCheck** an argument of 0 for unchecked, 1 for checked, or 2 for indeterminate.

To create a radio button, call a CCmdUI object's SetRadio member function from an **ON_UPDATE_COMMAND_UI** handler. Pass **SetRadio** an argument of 0 for unchecked or nonzero for checked. In order to provide a radio group's mutually exclusive behavior, you must have **ON_UPDATE_COMMAND_UI** handlers for all of the buttons in the group.

For more information on using **CToolBar**, see the article MFC Toolbar Implementation and Technical Note 31: Control Bars.

Requirements

Header: afxext.h

See Also

Tasks

CTRLBARS Sample: Illustrates Custom Control Bars
DLGCBR32 Sample: Demonstrates Adding a Status Bar and Toolbar to Dialog Boxes
DOCKTOOL Sample: Demonstrates Dockable Toolbars

Reference

CControlBar Class
Hierarchy Chart
CToolBarCtrl Class
CControlBar Class
CToolBar::Create
CToolBar::LoadBitmap
CToolBar::SetButtons
CCmdUI::SetCheck
CCmdUI::SetRadio

Other Resources

CToolBar Members

CToolBar Members

Base Class Members

COBJECT Members

CCmdTarget Members

CWnd Members

CControlBar Members

Construction

Create	Creates the Windows toolbar and attaches it to the **CToolBar** object.
CreateEx	Creates a **CToolBar** object with additional styles for the embedded **CToolBarCtrl** object.
CToolBar	Constructs a **CToolBar** object.
LoadBitmap	Loads the bitmap containing bitmap-button images.
LoadToolBar	Loads a toolbar resource created with the resource editor.
SetBitmap	Sets a bitmapped image.
SetButtons	Sets button styles and an index of button images within the bitmap.
SetHeight	Sets the height of the toolbar.
SetSizes	Sets the sizes of buttons and their bitmaps.

Attributes

CommandToIndex	Returns the index of a button with the given command ID.
GetButtonInfo	Retrieves the ID, style, and image number of a button.
GetButtonStyle	Retrieves the style for a button.
GetButtonText	Retrieves the text that will appear on a button.
GetItemID	Returns the command ID of a button or separator at the given index.
GetItemRect	Retrieves the display rectangle for the item at the given index.
GetToolBarCtrl	Allows direct access to the underlying common control.
SetButtonInfo	Sets the ID, style, and image number of a button.
SetButtonStyle	Sets the style for a button.
SetButtonText	Sets the text that will appear on a button.

See Also

Reference

CToolBar Class

Hierarchy Chart

CToolBar Member Functions

For information about the member functions in **CToolBar**, see CToolBar Members.

CToolBar::CommandToIndex

This member function returns the index of the first toolbar button, starting at position 0, whose command ID matches *nIDFind*.

```
int CommandToIndex(
   UINT nIDFind
) const;
```

Parameters

nIDFind

Command ID of a toolbar button.

Return Value

The index of the button, or –1 if no button has the given command ID.

See Also

Reference

CToolBar Class

Hierarchy Chart

CToolBar::GetItemID

Other Resources

CToolBar Members

CToolBar::Create

This member function creates a Windows toolbar (a child window) and associates it with the **CToolBar** object.

```
virtual BOOL Create(
   CWnd* pParentWnd,
   DWORD dwStyle = WS_CHILD |  WS_VISIBLE | CBRS_TOP,
   UINT nID = AFX_IDW_TOOLBAR
);
```

Parameters

pParentWnd

Pointer to the window that is the toolbar's parent.

dwStyle

The toolbar style. Additional toolbar styles supported are:

- **CBRS_TOP** Control bar is at top of the frame window.
- **CBRS_BOTTOM** Control bar is at bottom of the frame window.
- **CBRS_NOALIGN** Control bar is not repositioned when the parent is resized.
- **CBRS_TOOLTIPS** Control bar displays tool tips.
- **CBRS_SIZE_DYNAMIC** Control bar is dynamic.
- **CBRS_SIZE_FIXED** Control bar is fixed.
- **CBRS_FLOATING** Control bar is floating.
- **CBRS_FLYBY** Status bar displays information about the button.
- **CBRS_HIDE_INPLACE** Control bar is not displayed to the user.

nID

The toolbar's child-window ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

It also sets the toolbar height to a default value.

Example

```
//This example creates a dockable toolbar.
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
      return -1;

   if (!m_wndToolBar.Create(this) ||
      !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
   {
      TRACE0("Failed to create toolbar\n");
      return -1;      // fail to create
   }

   //Make the toolbar dockable
   m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
   EnableDocking(CBRS_ALIGN_ANY);
   DockControlBar(&m_wndToolBar);
```

```
    return 0;
}
```

See Also

Reference

CToolBar Class
Hierarchy Chart
CToolBar::CToolBar
CToolBar::LoadBitmap
CToolBar::SetButtons
CToolBar::LoadToolBar
CControlBar::CalcDynamicLayout
CControlBar::CalcFixedLayout

Other Resources

CToolBar Members

CToolBar::CreateEx

Call this function to create a Windows toolbar (a child window) and associate it with the **CToolBar** object.

```
virtual BOOL CreateEx(
   CWnd* pParentWnd,
   DWORD dwCtrlStyle = TBSTYLE_FLAT,
   DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_ALIGN_TOP,
   CRect rcBorders = CRect(
   0,
   0,
   0,
   0
),
   UINT nID = AFX_IDW_TOOLBAR
);
```

Parameters

pParentWnd

Pointer to the window that is the toolbar's parent.

dwCtrlStyle

Additional styles for the creation of the embedded CToolBarCtrl object. By default, this value is set to **TBSTYLE_FLAT**. For a complete list of toolbar styles, see *dwStyle*.

dwStyle

The toolbar style. See Toolbar Control and Button Styles in the *Platform SDK* for a list of appropriate styles.

rcBorders

A CRect object that defines the widths of the toolbar window borders. These borders are set to 0,0,0,0 by default, thereby resulting in a toolbar window with no borders.

nID

The toolbar's child-window ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

It also sets the toolbar height to a default value.

Use **CreateEx**, instead of Create, when certain styles need to be present during the creation of the embedded tool bar control. For example, set *dwCtrlStyle* to **TBSTYLE_FLAT | TBSTYLE_TRANSPARENT** to create a toolbar that resembles the Internet Explorer 4 toolbars.

Example

```
// This example demonstrates CToolBar::CreateEx by creating a
// toolbar as part of a child frame window. It also calls the
// LoadToolbar and EnableDocking functions

int CChildFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CMDIChildWnd::OnCreate(lpCreateStruct) == -1)
      return -1;

   if (!m_wndMyToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE | CBRS_TOP
      | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||
      !m_wndMyToolBar.LoadToolBar(IDR_MYTOOLBAR))
   {
```

```
        TRACE0("Failed to create toolbar\n");
        return -1;      // fail to create
    }

    m_wndMyToolBar.EnableDocking(CBRS_ALIGN_ANY);
    EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_wndMyToolBar);

    return 0;
}
```

See Also

Reference

CToolBar Class

Hierarchy Chart

Other Resources

CToolBar Members

CToolBar::CToolBar

This member function constructs a **CToolBar** object and sets the default sizes.

```
CToolBar( );
```

Remarks

Call the Create member function to create the toolbar window.

See Also

Reference

CToolBar Class
Hierarchy Chart
CToolBar::Create

Other Resources

CToolBar Members

CToolBar::GetButtonInfo

This member function retrieves the control ID, style, and image index of the toolbar button or separator at the location specified by *nIndex*.

```
void GetButtonInfo(
   int nIndex,
   UINT& nID,
   UINT& nStyle,
   int& iImage
) const;
```

Parameters

nIndex

Index of the toolbar button or separator whose information is to be retrieved.

nID

Reference to a **UINT** that is set to the command ID of the button.

nStyle

Reference to a **UINT** that is set to the style of the button.

iImage

Reference to an integer that is set to the index of the button's image within the bitmap.

Remarks

Those values are assigned to the variables referenced by *nID*, *nStyle*, and *iImage*. The image index is the position of the image within the bitmap that contains images for all the toolbar buttons. The first image is at position 0.

If *nIndex* specifies a separator, *iImage* is set to the separator width in pixels.

See Also

Reference

CToolBar Class
Hierarchy Chart
CToolBar::SetButtonInfo
CToolBar::GetItemID

Other Resources

CToolBar Members

CToolBar::GetButtonStyle

Call this member function to retrieve the style of a button or separator on the toolbar.

```
UINT GetButtonStyle(
   int nIndex
) const;
```

Parameters

nIndex

The index of the toolbar button or separator style to be retrieved.

Return Value

The style of the button or separator specified by *nIndex*.

Remarks

A button's style determines how the button appears and how it responds to user input. See SetButtonStyle for examples of button styles.

See Also

Reference

CToolBar Class
Hierarchy Chart
CToolBar::SetButtonStyle

Other Resources

CToolBar Members

CToolBar::GetButtonText

Call this member function to retrieve the text that appears on a button.

```
CString GetButtonText(
   int nIndex
) const;
void GetButtonText(
   int nIndex,
   CString& rString
) const;
```

Parameters

nIndex

Index of the text to be retrieved.

rString

A reference to a CString object that will contain the text to be retrieved.

Return Value

A **CString** object containing the button text.

Remarks

The second form of this member function fills a **CString** object with the string text.

See Also

Reference

CToolBar Class
Hierarchy Chart
CToolBar::SetButtonText

Other Resources

CToolBar Members

CToolBar::GetItemID

This member function returns the command ID of the button or separator specified by *nIndex*.

```
UINT GetItemID(
   int nIndex
) const;
```

Parameters

nIndex

Index of the item whose ID is to be retrieved.

Return Value

The command ID of the button or separator specified by *nIndex*.

Remarks

Separators return **ID_SEPARATOR**.

See Also

Reference

CToolBar Class
Hierarchy Chart
CToolBar::CommandToIndex
CControlBar::GetCount

Other Resources

CToolBar Members

CToolBar::GetItemRect

This member function fills the **RECT** structure whose address is contained in *lpRect* with the coordinates of the button or separator specified by *nIndex*.

```
virtual void GetItemRect(
   int nIndex,
   LPRECT lpRect
) const;
```

Parameters

nIndex

Index of the item (button or separator) whose rectangle coordinates are to be retrieved.

lpRect

Address of the RECT structure that will contain the item's coordinates.

Remarks

Coordinates are in pixels relative to the upper-left corner of the toolbar.

Use **GetItemRect** to get the coordinates of a separator you want to replace with a combo box or other control.

Example

See the example for CToolBar::SetSizes.

See Also

Reference

CToolBar Class
Hierarchy Chart
CToolBar::CommandToIndex

Other Resources

CToolBar Members

CToolBar::GetToolBarCtrl

This member function allows direct access to the underlying common control.

```
CToolBarCtrl& GetToolBarCtrl( ) const;
```

Return Value

A reference to a **CToolBarCtrl** object.

Remarks

Use **GetToolBarCtrl** to take advantage of the functionality of the Windows toolbar common control, and to take advantage of the support CToolBarCtrl provides for toolbar customization.

For more information about using common controls, see the article Controls and Common Controls in the *Platform SDK*.

Example

```
//This example shows how to add text to toolbar buttons.
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CMDIFrameWnd::OnCreate(lpCreateStruct) == -1)
      return -1;

   //Create a toolbar. Resource ID of the toolbar to be loaded
   //is IDR_MAINFRAME.

   if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE
      | CBRS_TOP) || !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
   {
      TRACE0("Failed to create toolbar\n");
      return -1;      // fail to create
   }

   //Show text on toolbar buttons.
   VERIFY(m_wndToolBar.SetButtonText(0,"New"));
   VERIFY(m_wndToolBar.SetButtonText(1,"Open"));
   VERIFY(m_wndToolBar.SetButtonText(2,"Save"));
   VERIFY(m_wndToolBar.SetButtonText(4,"Cut"));
   VERIFY(m_wndToolBar.SetButtonText(5,"Copy"));
   VERIFY(m_wndToolBar.SetButtonText(6,"Paste"));
   VERIFY(m_wndToolBar.SetButtonText(8,"Print"));
   VERIFY(m_wndToolBar.SetButtonText(9,"About"));

   CRect temp;
   m_wndToolBar.GetItemRect(0,&temp);
   m_wndToolBar.GetToolBarCtrl()->SetButtonSize(CSize(temp.Width(),
      temp.Height()));

   return 0;
}
```

See Also

Reference

CToolBar Class

Hierarchy Chart

CToolBarCtrl Class

Other Resources

CToolBar Members

CToolBar::LoadBitmap

Call this member function to load the bitmap specified by *lpszResourceName* or *nIDResource*.

```
BOOL LoadBitmap(
   LPCTSTR lpszResourceName
);
BOOL LoadBitmap(
   UINT nIDResource
);
```

Parameters

lpszResourceName

Pointer to the resource name of the bitmap to be loaded.

nIDResource

Resource ID of the bitmap to be loaded.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The bitmap should contain one image for each toolbar button. If the images are not of the standard size (16 pixels wide and 15 pixels high), call SetSizes to set the button sizes and their images.

See Also

Reference

CToolBar Class
Hierarchy Chart
CToolBar::Create
CToolBar::SetButtons
CToolBar::SetSizes
CToolBar::LoadToolBar

Other Resources

CToolBar Members

CToolBar::LoadToolBar

Call this member function to load the toolbar specified by *lpszResourceName* or *nIDResource*.

```
BOOL LoadToolBar(
   LPCTSTR lpszResourceName
);
BOOL LoadToolBar(
   UINT nIDResource
);
```

Parameters

lpszResourceName

Pointer to the resource name of the toolbar to be loaded.

nIDResource

Resource ID of the toolbar to be loaded.

Return Value

Nonzero if successful; otherwise 0.

Remarks

See toolbar editor in for more information about creating a toolbar resource.

Example

See the example for CToolBar::CreateEx.

See Also

Reference

CToolBar Class
Hierarchy Chart
CToolBar::Create
CToolBar::LoadBitmap
CToolBar::SetButtons

Other Resources

CToolBar Members

CToolBar::SetBitmap

Call this member function to set the bitmap image for the toolbar.

```
BOOL SetBitmap(
   HBITMAP hbmImageWell
);
```

Parameters

hbmImageWell

Handle of a bitmap image that is associated with a toolbar.

Return Value

Nonzero if successful; otherwise 0.

Remarks

For example, call **SetBitmap** to change the bitmapped image after the user takes an action on a document that changes the action of a button.

See Also

Reference

CToolBar Class

Hierarchy Chart

Other Resources

CToolBar Members

CToolBar::SetButtonInfo

Call this member function to set the button's command ID, style, and image number.

```
void SetButtonInfo(
   int nIndex,
   UINT nID,
   UINT nStyle,
   int iImage
);
```

Parameters

nIndex

Zero-based index of the button or separator for which information is to be set.

nID

The value to which the button's command ID is set.

nStyle

The new button style. The following button styles are supported:

- **TBBS_BUTTON** Standard pushbutton (default)
- **TBBS_SEPARATOR** Separator
- **TBBS_CHECKBOX** Auto check-box button
- **TBBS_GROUP** Marks the start of a group of buttons
- **TBBS_CHECKGROUP** Marks the start of a group of check-box buttons
- **TBBS_DROPDOWN** Creates a drop-down list button.
- **TBBS_AUTOSIZE** The button's width will be calculated based on the text of the button, not on the size of the image.
- **TBBS_NOPREFIX** The button text will not have an accelerator prefix associated with it.

iImage

New index for the button's image within the bitmap.

Remarks

For separators, which have the style **TBBS_SEPARATOR**, this function sets the separator's width in pixels to the value stored in *iImage*.

Note

You can also set button states using the *nStyle* parameter; however, because button states are controlled by the ON_UPDATE_COMMAND_UI handler, any state you set using **SetButtonInfo** will be lost during the next idle processing. See How to Update User-Interface Objects and TN031: Control Bars for more information.

For information on bitmap images and buttons, see the CToolBar Overview and CToolBar::LoadBitmap.

See Also

Reference

CToolBar Class
Hierarchy Chart
CToolBar::GetButtonInfo

Other Resources

CToolBar Members

CToolBar::SetButtons

This member function sets each toolbar button's command ID to the value specified by the corresponding element of the array *lpIDArray*.

```
BOOL SetButtons(
   const UINT* lpIDArray,
   int nIDCount
);
```

Parameters

lpIDArray

Pointer to an array of command Ids. It can be **NULL** to allocate empty buttons.

nIDCount

Number of elements in the array pointed to by *lpIDArray*.

Return Value

Nonzero if successful; otherwise 0.

Remarks

If an element of the array has the value **ID_SEPARATOR**, a separator is created in the corresponding position of the toolbar. This function also sets each button's style to **TBBS_BUTTON** and each separator's style to **TBBS_SEPARATOR**, and assigns an image index to each button. The image index specifies the position of the button's image within the bitmap.

You do not need to account for separators in the bitmap because this function does not assign image indexes for separators. If your toolbar has buttons at positions 0, 1, and 3 and a separator at position 2, the images at positions 0, 1, and 2 in your bitmap are assigned to the buttons at positions 0, 1, and 3, respectively.

If *lpIDArray* is **NULL**, this function allocates space for the number of items specified by *nIDCount*. Use SetButtonInfo to set each item's attributes.

See Also

Reference

CToolBar Class
Hierarchy Chart
CToolBar::Create
CToolBar::SetButtonInfo
CToolBar::SetButtonStyle
CToolBar::LoadToolBar

Other Resources

CToolBar Members

CToolBar::SetButtonStyle

Call this member function to set the style of a button or separator, or to group buttons.

```
void SetButtonStyle(
   int nIndex,
   UINT nStyle
);
```

Parameters

nIndex

Index of the button or separator whose information is to be set.

nStyle

The button style. The following button styles are supported:

- **TBBS_BUTTON** Standard pushbutton (default)
- **TBBS_SEPARATOR** Separator
- **TBBS_CHECKBOX** Auto check-box button
- **TBBS_GROUP** Marks the start of a group of buttons
- **TBBS_CHECKGROUP** Marks the start of a group of check-box buttons
- **TBBS_DROPDOWN** Creates a drop-down list button
- **TBBS_AUTOSIZE** The button's width will be calculated based on the text of the button, not on the size of the image
- **TBBS_NOPREFIX** The button text will not have an accelerator prefix associated with it

Remarks

A button's style determines how the button appears and how it responds to user input.

Before calling **SetButtonStyle**, call the GetButtonStyle member function to retrieve the button or separator style.

Note

You can also set button states using the *nStyle* parameter; however, because button states are controlled by the ON_UPDATE_COMMAND_UI handler, any state you set using **SetButtonStyle** will be lost during the next idle processing. Se e How to Update User-Interface Objects and TN031: Control Bars for more information.

See Also

Reference

CToolBar Class
Hierarchy Chart
CToolBar::GetButtonStyle

Other Resources

CToolBar Members

CToolBar::SetButtonText

Call this function to set the text on a button.

```
BOOL SetButtonText(
   int nIndex,
   LPCTSTR lpszText
);
```

Parameters

nIndex

Index of the button whose text is to be set.

lpszText

Points to the text to be set on a button.

Return Value

Nonzero if successful; otherwise 0.

Example

See the example for CToolBar::GetToolBarCtrl.

See Also

Reference

CToolBar Class
Hierarchy Chart
CToolBar::GetButtonText

Other Resources

CToolBar Members

CToolBar::SetHeight

This member function sets the toolbar's height to the value, in pixels, specified in *cyHeight*.

```
void SetHeight(
   int cyHeight
);
```

Parameters

cyHeight

The height in pixels of the toolbar.

Remarks

After calling SetSizes, use this member function to override the standard toolbar height. If the height is too small, the buttons will be clipped at the bottom.

If this function is not called, the framework uses the size of the button to determine the toolbar height.

See Also

Reference

CToolBar Class
Hierarchy Chart
CToolBar::SetSizes
CToolBar::SetButtonInfo
CToolBar::SetButtons

Other Resources

CToolBar Members

CToolBar::SetSizes

Call this member function to set the toolbar's buttons to the size, in pixels, specified in *sizeButton*.

```
void SetSizes(
   SIZE sizeButton,
   SIZE sizeImage
);
```

Parameters

sizeButton

The size in pixels of each button.

sizeImage

The size in pixels of each image.

Remarks

The *sizeImage* parameter must contain the size, in pixels, of the images in the toolbar's bitmap. The dimensions in *sizeButton* must be sufficient to hold the image plus 7 pixels extra in width and 6 pixels extra in height. This function also sets the toolbar height to fit the buttons.

Call this member function only for toolbars that do not follow *Windows Interface Guidelines for Software Design* recommendations for button and image sizes.

Example

```
// This example shows how to add text to toolbar buttons.
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CMDIFrameWnd::OnCreate(lpCreateStruct) == -1)
      return -1;

   //Create a toolbar. Resource ID of the toolbar to be loaded
   //is IDR_MAINFRAME.

   if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE
      | CBRS_TOP) || !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
   {
      TRACE0("Failed to create toolbar\n");
      return -1;  // fail to create
   }

   //Show text on toolbar buttons.
   VERIFY(m_wndToolBar.SetButtonText(0,"New"));
   VERIFY(m_wndToolBar.SetButtonText(1,"Open"));
   VERIFY(m_wndToolBar.SetButtonText(2,"Save"));
   VERIFY(m_wndToolBar.SetButtonText(4,"Cut"));
   VERIFY(m_wndToolBar.SetButtonText(5,"Copy"));
   VERIFY(m_wndToolBar.SetButtonText(6,"Paste"));
   VERIFY(m_wndToolBar.SetButtonText(8,"Print"));
   VERIFY(m_wndToolBar.SetButtonText(9,"About"));

   CRect temp;
   m_wndToolBar.GetItemRect(0,&temp);
   m_wndToolBar.SetSizes(CSize(temp.Width(),
      temp.Height()),CSize(16,15));

   return 0;
}
```

See Also

Reference

CToolBar Class
Hierarchy Chart
CToolBar::LoadBitmap
CToolBar::SetButtonInfo
CToolBar::SetButtons
Other Resources
CToolBar Members

CToolBarCtrl Class

Provides the functionality of the Windows toolbar common control.

```
class CToolBarCtrl : public CWnd
```

Remarks

This control (and therefore the **CToolBarCtrl** class) is available only to programs running under Windows 95/98 and Windows NT version 3.51 and later.

A Windows toolbar common control is a rectangular child window that contains one or more buttons. These buttons can display a bitmap image, a string, or both. When the user chooses a button, it sends a command message to the toolbar's owner window. Typically, the buttons in a toolbar correspond to items in the application's menu; they provide a more direct way for the user to access an application's commands.

CToolBarCtrl objects contain several important internal data structures: a list of button image bitmaps or an image list, a list of button label strings, and a list of **TBBUTTON** structures which associate an image and/or string with the position, style, state, and command ID of the button. Each of the elements of these data structures is referred to by a zero-based index. Before you can use a **CToolBarCtrl** object, you must set up these data structures. The list of strings can only be used for button labels; you cannot retrieve strings from the toolbar.

To use a **CToolBarCtrl** object, you will typically follow these steps:

1. Construct the **CToolBarCtrl** object.
2. Call Create to create the Windows toolbar common control and attach it to the **CToolBarCtrl** object. Indicate the style of toolbar by using styles, such as **TBSTYLE_TRANSPARENT** for a transparent toolbar or **TBSTYLE_DROPDOWN** for a toolbar that supports drop-down style buttons.
3. Identify how you want the buttons on the toolbar displayed:
   - To use bitmap images for buttons, add the button bitmaps to the toolbar by calling AddBitmap.
   - To use images displayed from an image list for buttons, specify the image list by calling SetImageList, SetHotImageList, or SetDisabledImageList.
   - To use string labels for buttons, add the strings to the toolbar by calling AddString and/or AddStrings.
4. Add button structures to the toolbar by calling AddButtons.
5. If you want tool tips for a toolbar button in an owner window that is not a **CFrameWnd**, you need to handle the **TTN_NEEDTEXT** messages in the toolbar's owner window as described in Handling Tool Tip Notifications. If the parent window of the toolbar is derived from **CFrameWnd**, tool tips are displayed without any extra effort from you because **CFrameWnd** provides a default handler.
6. If you want your user to be able to customize the toolbar, handle customization notification messages in the owner window as described in Handling Customization Notifications.

You can use SaveState to save the current state of a toolbar control in the registry and RestoreState to restore the state based on information previously stored in the registry. In addition to saving the toolbar state between uses of the application, applications typically store the state before the user begins customizing the toolbar in case the user later wants to restore the toolbar to its original state.

Support for Internet Explorer Version 4.0 and Later

To support functionality introduced in Internet Explorer, version 4.0 and later, MFC provides image list support and transparent and flat styles for toolbar controls.

A transparent toolbar allows the client under the toolbar to show through. To create a transparent toolbar, use both **TBSTYLE_FLAT** and **TBSTYLE_TRANSPARENT** styles. Transparent toolbars feature hot tracking; that is, when the mouse pointer moves over a hot button on the toolbar, the button's appearance changes. Toolbars created with just the **TBSTYLE_FLAT** style will contain buttons that are not transparent.

Image list support allows a control greater flexibility for default behavior, hot images, and disabled images. Use GetImageList,

GetHotImageList, and GetDisabledImageList with the transparent toolbar to manipulate the image according to its state:

For more information on using **CToolBarCtrl**, see Controls and Using CToolBarCtrl.

Requirements

Header: afxcmn.h

See Also

Tasks

CMNCTRL1 Sample: Demonstrates Common Control MFC Classes, Part 1

MFCIE Sample: Demonstrates the MFC CHtmlView and CReBar Classes

Reference

CWnd Class

Hierarchy Chart

CToolBar Class

Other Resources

CToolBarCtrl Members

CToolBarCtrl Members

Base Class Members

Construction

Attributes

Operations

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Construction

Create	Creates a toolbar control and attaches it to a **CToolBarCtrl** object.
CreateEx	Creates a toolbar control with the specified Windows extended styles and attaches it to a **CToolBarCtrl** object.
CToolBarCtrl	Constructs a **CToolBarCtrl** object.

Attributes

GetAnchorHighlight	Retrieves the anchor highlight setting for a toolbar.
GetBitmap	Retrieves the index of the bitmap associated with a button in a toolbar.
GetBitmapFlags	Gets flags associated with the toolbar's bitmap.
GetButton	Retrieves information about the specified button in a toolbar control.
GetButtonCount	Retrieves a count of the buttons currently in the toolbar control.
GetButtonSize	Retrieves the current width and height of toolbar buttons, in pixels.
GetDisabledImageList	Retrieves the image list that a toolbar control uses to display disabled buttons.
GetDropTarget	Retrieves the IDropTarget interface for a toolbar control.
GetExtendedStyle	Retrieves the extended styles for a toolbar control.
GetHotImageList	Retrieves the image list that a toolbar control uses to display "hot" buttons. A hot button appears highlighted when the mouse pointer is above it.
GetHotItem	Retrieves the index of the hot item in a toolbar.
GetImageList	Retrieves the image list that a toolbar control uses to display buttons in their default state.
GetInsertMark	Retrieves the current insertion mark for the toolbar.
GetInsertMarkColor	Retrieves the color used to draw the insertion mark for the toolbar.
GetItemRect	Retrieves the bounding rectangle of a button in a toolbar control.

GetMaxSize	Retrieves the total size of all of the visible buttons and separators in the toolbar.
GetMaxTextRows	Retrieves the maximum number of text rows displayed on a toolbar button.
GetMetrics	Retrieves the metrics of a toolbar control.
GetRect	Retrieves the bounding rectangle for a specified toolbar button.
GetRows	Retrieves the number of rows of buttons currently displayed in the toolbar.
GetState	Retrieves information about the state of the specified button in a toolbar control, such as whether it is e nabled, pressed, or checked.
GetStyle	Retrieves the styles currently in use for a toolbar control.
GetToolTips	Retrieves the handle of the tool tip control, if any, associated with the toolbar control.
HitTest	Determines where a point lies in a toolbar control.
InsertMarkHitTest	Retrieves the insertion mark information for a point in a toolbar.
IsButtonChecked	Tells whether the specified button in a toolbar control is checked.
IsButtonEnabled	Tells whether the specified button in a toolbar control is enabled.
IsButtonHidden	Tells whether the specified button in a toolbar control is hidden.
IsButtonHighlighted	Checks the highlight state of the toolbar button.
IsButtonIndeterminate	Tells whether the state of the specified button in a toolbar control is indeterminate (gray).
IsButtonPressed	Tells whether the specified button in a toolbar control is pressed.
MapAccelerator	Maps an accelerator character to a toolbar button.
MoveButton	Moves a button from one index to another.
SetAnchorHighlight	Sets the anchor highlight setting for a toolbar.
SetBitmapSize	Sets the size of the bitmapped images to be added to a toolbar control.
SetButtonSize	Sets the size of the buttons to be added to a toolbar control.
SetButtonStructSize	Specifies the size of the **TBBUTTON** structure.
SetButtonWidth	Sets the minimum and maximum button widths in the toolbar control.
SetCmdID	Sets the command identifier to be sent to the owner window when the specified button is pressed.
SetDisabledImageList	Sets the image list that the toolbar control will use to display disabled buttons.
SetExtendedStyle	Sets the extended styles for a toolbar control.

SetHotImageList	Sets the image list that the toolbar control will use to display "hot" buttons.
SetHotItem	Sets the hot item in a toolbar.
SetImageList	Sets the image list that the toolbar will use to display buttons that are in their default state.
SetIndent	Sets the indentation for the first button in a toolbar control.
SetInsertMark	Sets the current insertion mark for the toolbar.
SetInsertMarkColor	Sets the color used to draw the insertion mark for the toolbar.
SetMaxTextRows	Sets the maximum number of text rows displayed on a toolbar button.
SetMetrics	Sets the metrics of a toolbar control.
SetWindowTheme	Sets the visual style of a toolbar control.
SetOwner	Sets the window to receive notification messages from the toolbar control.
SetRows	Sets the number of rows of buttons displayed in the toolbar.
SetState	Sets the state for the specified button in a toolbar control.
SetStyle	Sets the styles for a toolbar control.
SetToolTips	Associates a tool tip control with the toolbar control.

Operations

AddBitmap	Adds one or more bitmap button images to the list of button images available for a toolbar control.
AddButtons	Adds one or more buttons to a toolbar control.
AddString	Adds a new string, passed as a resource ID, to the toolbar's internal list of strings.
AddStrings	Adds a new string or strings, passed as a pointer to a buffer of null-separated strings, to the toolbar's intern al list of strings.
AutoSize	Resizes a toolbar control.
CheckButton	Checks or clears a given button in a toolbar control.
CommandToIndex	Retrieves the zero-based index for the button associated with the specified command identifier.
Customize	Displays the Customize Toolbar dialog box.
DeleteButton	Deletes a button from the toolbar control.
EnableButton	Enables or disables the specified button in a toolbar control.
GetButtonInfo	Retrieves the information for a button in a toolbar.

GetString	Retrieves a toolbar string.
HideButton	Hides or shows the specified button in a toolbar control.
Indeterminate	Sets or clears the indeterminate (gray) state of the specified button in a toolbar control.
InsertButton	Inserts a button in a toolbar control.
LoadImages	Loads bitmaps into a toolbar control's image list.
MarkButton	Sets the highlight state of a given button in a toolbar control.
PressButton	Presses or releases the specified button in a toolbar control.
RestoreState	Restores the state of the toolbar control.
SaveState	Saves the state of the toolbar control.
SetButtonInfo	Sets the information for an existing button in a toolbar.
SetDrawTextFlags	Sets the flags in the Win32 function DrawText, which is used to draw the text in the specified rectangle, for matted according to how the flags are set.

See Also

Reference

CToolBarCtrl Class

Hierarchy Chart

CToolBarCtrl Member Functions

For information about the member functions in **CToolBarCtrl**, see CToolBarCtrl Members.

CToolBarCtrl::AddBitmap

Adds one or more button images to the list of button images stored in the toolbar control.

```
int AddBitmap(
   int nNumButtons,
   UINT nBitmapID
);
int AddBitmap(
   int nNumButtons,
   CBitmap* pBitmap
);
```

Parameters

nNumButtons

Number of button images in the bitmap.

nBitmapID

Resource identifier of the bitmap that contains the button image or images to add.

pBitmap

Pointer to the **CBitmap** object that contains the button image or images to add.

Return Value

Zero-based index of the first new image if successful; otherwise – 1.

Remarks

You can use the Windows API CreateMappedBitmap to map colors before adding the bitmap to the toolbar. If you pass a pointer to a **CBitMap** object, you must ensure that the bitmap is not destroyed until after the toolbar is destroyed.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::AddButtons
CToolBarCtrl::InsertButton
CToolBarCtrl::AddString
CToolBarCtrl::AddStrings

Other Resources

CToolBarCtrl Members

CToolBarCtrl::AddButtons

Adds one or more buttons to a toolbar control.

```
BOOL AddButtons(
   int nNumButtons,
   LPTBBUTTON lpButtons
);
```

Parameters

nNumButtons

Number of buttons to add.

lpButtons

Address of an array of **TBBUTTON** structures that contains information about the buttons to add. There must be the same number of elements in the array as buttons specified by *nNumButtons*.

Return Value

Nonzero if successful; otherwise zero.

Remarks

The *lpButtons* pointer points to an array of **TBBUTTON** structures. Each **TBBUTTON** structure associates the button being added with the button's style, image and/or string, command ID, state, and user-defined data:

```
typedef struct _TBBUTTON {
int iBitmap;// zero-based index of button image
int idCommand;  // command to be sent when button pressed
BYTE fsState;   // button state--see below
BYTE fsStyle;   // button style--see below
DWORD dwData;   // application-defined value
int iString;// zero-based index of button label string
} TBBUTTON;
```

The members are as follows:

iBitmap

Zero-based index of button image, -1 if no image for this button.

idCommand

Command identifier associated with the button. This identifier is sent in a **WM_COMMAND** message when the button is chosen. If the **fsStyle** member has the **TBSTYLE_SEP** value, this member must be zero.

fsState

Button state flags. It can be a combination of the values listed below:

- **TBSTATE_CHECKED** The button has the **TBSTYLE_CHECKED** style and is being pressed.
- **TBSTATE_ENABLED** The button accepts user input. A button that does not have this state does not accept user input and is grayed.
- **TBSTATE_HIDDEN** The button is not visible and cannot receive user input.
- **TBSTATE_INDETERMINATE** The button is grayed.
- **TBSTATE_PRESSED** The button is being pressed.
- **TBSTATE_WRAP** A line break follows the button. The button must also have the **TBSTATE_ENABLED** state.

fsStyle

Button style. It can be a combination of the values listed below:

- **TBSTYLE_BUTTON** Creates a standard push button.
- **TBSTYLE_CHECK** Creates a button that toggles between the pressed and unpressed states each time the user clicks it. The button has a different background color when it is in the pressed state.
- **TBSTYLE_CHECKGROUP** Creates a check button that stays pressed until another button in the group is pressed.
- **TBSTYLE_GROUP** Creates a button that stays pressed until another button in the group is pressed.
- **TBSTYLE_SEP** Creates a separator, providing a small gap between button groups. A button that has this style does not receive user input.

dwData

User-defined data.

iString

Zero-based index of the string to use as the button's label, -1 if there is no string for this button.

The image and/or string whose index you provide must have previously been added to the toolbar control's list using AddBitmap, AddString, and/or AddStrings.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::InsertButton
CToolBarCtrl::DeleteButton
CToolBarCtrl::AddBitmap
CToolBarCtrl::AddString
CToolBarCtrl::AddStrings

Other Resources

CToolBarCtrl Members

CToolBarCtrl::AddString

Adds a new string, passed as a resource ID, to the toolbar's internal list of strings.

```
int AddString(
   UINT nStringID
);
```

Parameters

nStringID

Resource identifier of the string resource to add to the toolbar control's string list.

Return Value

The zero-based index of the first new string added if successful; otherwise –1.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::AddStrings
CToolBarCtrl::AddButtons
CToolBarCtrl::InsertButton
CToolBarCtrl::AddBitmap

Other Resources

CToolBarCtrl Members

CToolBarCtrl::AddStrings

Adds a new string or strings to the list of strings available for a toolbar control.

```
int AddStrings(
   LPCTSTR lpszStrings
);
```

Parameters

lpszStrings

Address of a buffer that contains one or more null-terminated strings to add to the toolbar's string list. The last string must be terminated with two null characters.

Return Value

The zero-based index of the first new string added if successful; otherwise –1.

Remarks

Strings in the buffer must be separated by a null character. You must ensure that the last string has two null terminators. To properly format a constant string, you might write it as:

```
// one null added automatically
lpszStrings = "Only one string to add\0";
```

or:

```
// adds three strings with one call
lpszStrings = "String 1\0String 2\0String 3\0";
```

You should not pass a **CString** object to this function since it is not possible to have more than one null character in a **CString**.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::AddString
CToolBarCtrl::AddButtons
CToolBarCtrl::InsertButton
CToolBarCtrl::AddBitmap

Other Resources

CToolBarCtrl Members

CToolBarCtrl::AutoSize

Resizes the entire toolbar control.

```
void AutoSize( );
```

Remarks

You should call this function when the size of the parent window changes or when the size of the toolbar changes (such as when you set the button or bitmap size, or add strings).

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::SetBitmapSize
CToolBarCtrl::SetButtonSize
CToolBarCtrl::AddString
CToolBarCtrl::AddStrings

Other Resources

CToolBarCtrl Members

CToolBarCtrl::CheckButton

Checks or clears a given button in a toolbar control.

```
BOOL CheckButton(
   int nID,
   BOOL bCheck = TRUE
);
```

Parameters

nID

Command identifier of the button to check or clear.

bCheck

TRUE to check the button, **FALSE** to clear it.

Return Value

Nonzero if successful; otherwise zero.

Remarks

When a button has been checked, it appears to have been pressed. If you want to change more than one button state, consider calling SetState instead.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::IsButtonChecked
CToolBarCtrl::EnableButton
CToolBarCtrl::PressButton
CToolBarCtrl::HideButton
CToolBarCtrl::Indeterminate
CToolBarCtrl::GetState
CToolBarCtrl::SetState

Other Resources

CToolBarCtrl Members

CToolBarCtrl::CommandToIndex

Retrieves the zero-based index for the button associated with the specified command identifier.

```
UINT CommandToIndex(
   UINT nID
) const;
```

Parameters

nID

Command ID whose button index you want to find.

Return Value

The zero-based index for the button associated with the command ID.

Remarks

CToolBarCtrl Overview | Class Members | Hierarchy Chart | CToolBarCtrl::SetCmdID | CToolBarCtrl::GetButton | CToolBarCtrl::AddButtons | CToolBarCtrl::InsertButton

See Also

Other Resources

CToolBarCtrl Member Functions

CToolBarCtrl::Create

Creates a toolbar control and attaches it to a **CToolBarCtrl** object.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the toolbar control's style. Toolbars must always have the **WS_CHILD** style. In addition, you can specify any combination of toolbar styles and window styles as described under **Remarks**.

rect

Optionally specifies the toolbar control's size and position. It can be either a CRect object or a RECT structure.

pParentWnd

Specifies the toolbar control's parent window. It must not be **NULL**.

nID

Specifies the toolbar control's ID.

Return Value

Nonzero if successful; otherwise zero.

Remarks

You construct a **CToolBarCtrl** in two steps. First, call the constructor, and then call **Create**, which creates the toolbar control and attaches it to the **CToolBarCtrl** object. Apply the following window styles to a toolbar control.

- **WS_CHILD** Always
- **WS_VISIBLE** Usually
- **WS_DISABLED** Rarely

See CreateWindow in the *Platform SDK* for a description of window styles.

Optionally, apply a combination of common control styles, as described in the *Platform SDK*.

Apply a combination of toolbar styles to either the control or the buttons themselves. The styles are described in the topic Toolbar Control and Button Styles in the *Platform SDK*.

To use extended toolbar styles, call SetExtendedStyle after you call **Create**. To create a toolbar with extended window styles, call CToolBarCtrl::CreateEx instead of **Create**.

The toolbar control automatically sets the size and position of the toolbar window. The height is based on the height of the buttons in the toolbar. The width is the same as the width of the parent window's client area. The **CCS_TOP** and **CCS_BOTTOM** styles determine whether the toolbar is positioned along the top or bottom of the client area. By default, a toolbar has the **CCS_TOP** style.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::CToolBarCtrl
CToolBarCtrl::SetButtonStructSize

Other Resources

CToolBarCtrl Members

CToolBarCtrl::CreateEx

Creates a control (a child window) and associates it with the **CToolBarCtrl** object.

```
virtual BOOL CreateEx(
   DWORD dwExStyle,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwExStyle

Specifies the extended style of the control being created. For a list of extended Windows styles, see the *dwExStyle* parameter for CreateWindowEx in the *Platform SDK*.

dwStyle

Specifies the toolbar control's style. Toolbars must always have the **WS_CHILD** style. In addition, you can specify any combination of toolbar styles and window styles as described in the **Remarks** section of Create.

rect

A reference to a RECT structure describing the size and position of the window to be created, in client coordinates of *pParentWnd*.

pParentWnd

A pointer to the window that is the control's parent.

nID

The control's child-window ID.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Use **CreateEx** instead of Create to apply extended Windows styles, specified by the Windows extended style preface **WS_EX_**. **CreateEx** creates the control with the extended Windows styles specified by *dwExStyle*. Set extended styles specific to a control using SetExtendedStyle. For example, use **CreateEx** to set such styles as **WS_EX_CONTEXTHELP**, but use **SetExtendedStyle** to set such styles as **TBSTYLE_EX_DRAWDDARROWS**. For more information, see the styles described in Toolbar Extended Styles in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::CToolBarCtrl

Other Resources

CToolBarCtrl Members

CToolBarCtrl::CToolBarCtrl

Constructs a **CToolBarCtrl** object.

```
CToolBarCtrl( );
```

Remarks

You must call Create to make the toolbar usable.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::Create
CToolBarCtrl::CreateEx

Other Resources

CToolBarCtrl Members

CToolBarCtrl::Customize

Displays the Customize Toolbar dialog box.

```
void Customize( );
```

Remarks

This dialog box allows the user to customize the toolbar by adding and deleting buttons. To support customization, your toolbar's parent window must handle the customization notification messages as described in Handling Customization Notifications. Your toolbar must also have been created with the **CCS_ADJUSTABLE** style, as described in CToolBarCtrl::Create.

For more information, see Knowledge Base article Q241850 : PRB: Call to CToolBarCtrl::Customize Does Not Keep the Customize Dialog Visible.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
Handling Customization Notifications

Other Resources

CToolBarCtrl Members

CToolBarCtrl::DeleteButton

Deletes a button from the toolbar control.

```
BOOL DeleteButton(
   int nIndex
);
```

Parameters

nIndex

Zero-based index of the button to delete.

Return Value

Nonzero if successful; otherwise zero.

Remarks

CToolBarCtrl Overview | Class Members | Hierarchy Chart | CToolBarCtrl::AddButtons | CToolBarCtrl::AutoSize | CToolBarCtrl::InsertButton

See Also

Other Resources

CToolBarCtrl Member Functions

CToolBarCtrl::EnableButton

Enables or disables the specified button in a toolbar control.

```
BOOL EnableButton(
   int nID,
   BOOL bEnable = TRUE
);
```

Parameters

nID

Command identifier of the button to enable or disable.

bEnable

TRUE to enable the button; **FALSE** to disable the button.

Return Value

Nonzero if successful; otherwise zero.

Remarks

When a button has been enabled, it can be pressed and checked. If you want to change more than one button state, consider calling SetState instead.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::IsButtonEnabled
CToolBarCtrl::CheckButton
CToolBarCtrl::PressButton
CToolBarCtrl::HideButton
CToolBarCtrl::Indeterminate
CToolBarCtrl::GetState
CToolBarCtrl::SetState

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetAnchorHighlight

Retrieves the anchor highlight setting for a toolbar.

```
BOOL GetAnchorHighlight( ) const;
```

Return Value

If nonzero, anchor highlighting is enabled. If zero, anchor highlighting is disabled.

Remarks

This member function implements the behavior of the Win32 message TB_GETANCHORHIGHLIGHT, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::SetAnchorHighlight

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetBitmap

Retrieves the index of the bitmap associated with a button in a toolbar.

```
int GetBitmap(
   int nID
) const;
```

Parameters

nID

Command identifier of the button whose bitmap index is to be retrieved.

Return Value

Returns the index of the bitmap if successful, or zero otherwise.

Remarks

Implements the functionality of TB_GETBITMAP in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class

Hierarchy Chart

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetBitmapFlags

Retrieves the bitmap flags from the toolbar.

```
UINT GetBitmapFlags( ) const;
```

Return Value

A **UINT** that has the **TBBF_LARGE** flag set if the display can support large toolbar bitmaps, clear otherwise.

Remarks

You should call it after creating the toolbar but before adding bitmaps to the toolbar. The return value indicates whether the display supports large bitmaps or not. If the display supports large bitmaps and if you choose to use them, call SetBitmapSize and SetButtonSize before adding your large bitmap using AddBitmap.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::AddBitmap
CToolBarCtrl::SetBitmapSize
CToolBarCtrl::SetButtonSize

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetButton

Retrieves information about the specified button in a toolbar control.

```
BOOL GetButton(
   int nIndex,
   LPTBBUTTON lpButton
) const;
```

Parameters

nIndex

Zero-based index of the button for which to retrieve information.

lpButton

Address of the **TBBUTTON** structure that is to receive a copy of the button information. See CToolBarCtrl::AddButtons for information about the **TBBUTTON** structure.

Return Value

Nonzero if successful; otherwise zero.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::GetState
CToolBarCtrl::SetState
CToolBarCtrl::GetButtonCount
CToolBarCtrl::GetItemRect
CToolBarCtrl::CommandToIndex
CToolBarCtrl::AddButtons
CToolBarCtrl::InsertButton

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetButtonCount

Retrieves a count of the buttons currently in the toolbar control.

```
int GetButtonCount( ) const;
```

Return Value

The count of the buttons.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::GetButton
CToolBarCtrl::GetState
CToolBarCtrl::GetItemRect
CToolBarCtrl::AddButtons
CToolBarCtrl::InsertButton
CToolBarCtrl::DeleteButton

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetButtonInfo

Retrieves the information for a button in a toolbar.

```
BOOL GetButtonInfo(
   int nID,
   TBBUTTONINFO* ptbbi
) const;
```

Parameters

nID

The button identifier.

ptbbi

A pointer to a TBBUTTONINFO structure that receives the button information.

Return Value

Nonzero if successful; otherwise zero.

Remarks

This member function implements the behavior of the Win32 message TB_GETBUTTONINFO, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::SetButtonInfo

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetButtonSize

Gets the size of a toolbar button.

```
DWORD GetButtonSize( ) const;
```

Return Value

A **DWORD** value that contains the width and height values in the LOWORD and HIWORD, respectively.

See Also

Reference

CToolBarCtrl Class

Hierarchy Chart

CToolBarCtrl::GetButtonInfo

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetDisabledImageList

Retrieves the image list that a toolbar control uses to display disabled buttons.

```
CImageList* GetDisabledImageList( ) const;
```

Return Value

A pointer to a CImageList object, or **NULL** if no disabled image list is set.

Remarks

This member function implements the behavior of the Win32 message TB_GETDISABLEDIMAGELIST, as described in the *Platform SDK*. The MFC implementation of **GetDisabledImageList** uses a **CImageList** object containing the toolbar control's button images, rather than a handle to an image list.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::SetDisabledImageList
CToolBarCtrl::GetHotImageList
CToolBarCtrl::GetImageList

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetDropTarget

Retrieves the IDropTarget interface for a toolbar control.

```
HRESULT GetDropTarget(
   IDropTarget** ppDropTarget
) const;
```

Parameters

ppDropTarget

A pointer to an IDropTarget interface pointer. If an error occurs, a **NULL** pointer is placed in this address.

Return Value

Returns an **HRESULT** value indicating success or failure of the operation.

Remarks

This member function implements the behavior of the Win32 message TB_GETOBJECT, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class

Hierarchy Chart

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetExtendedStyle

Retrieves the extended styles for a toolbar control.

```
DWORD GetExtendedStyle( ) const;
```

Return Value

A **DWORD** that represents the extended styles currently in use for the toolbar control. For a list of styles, see Toolbar Extended Styles, in the *Platform SDK*.

Remarks

This member function implements the behavior of the Win32 message TB_GETEXTENDEDSTYLE, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::SetExtendedStyle

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetHotImageList

Retrieves the image list that a toolbar control uses to display "hot" buttons. A hot button appears highlighted when the mouse pointer is above it.

```
CImageList* GetHotImageList( ) const;
```

Return Value

A pointer to a CImageList object, or **NULL** if no disabled image list is set.

Remarks

This member function implements the behavior of the Win32 message TB_GETHOTIMAGELIST, as described in the *Platform SDK*. A hot button appears highlighted when the mouse pointer is above it.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::GetDisabledImageList
CToolBarCtrl::GetImageList

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetHotItem

Retrieves the index of the hot item in a toolbar.

```
int GetHotItem( ) const;
```

Return Value

The zero-based index of the hot item in a toolbar.

Remarks

This member function implements the behavior of the Win32 message TB_GETHOTITEM, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::SetHotItem

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetImageList

Retrieves the image list that a toolbar control uses to display buttons in their default state.

```
CImageList* GetImageList( ) const;
```

Return Value

A pointer to a CImageList object, or **NULL** if no image list is set.

Remarks

This member function implements the behavior of the Win32 message TB_GETIMAGELIST, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::GetDisabledImageList
CToolBarCtrl::GetHotImageList

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetInsertMark

Retrieves the current insertion mark for the toolbar.

```
void GetInsertMark(
   TBINSERTMARK* ptbim
) const;
```

Parameters

ptbim

A pointer to a TBINSERTMARK structure that receives the insertion mark.

Remarks

This member function implements the behavior of the Win32 message TB_GETINSERTMARK, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::SetInsertMark

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetInsertMarkColor

Retrieves the color used to draw the insertion mark for the toolbar.

```
COLORREF GetInsertMarkColor( ) const;
```

Return Value

A **COLORREF** value that contains the current insertion mark color.

Remarks

This member function implements the behavior of the Win32 message TB_GETINSERTMARKCOLOR, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::SetInsertMarkColor

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetItemRect

Retrieves the bounding rectangle of a button in a toolbar control.

```
BOOL GetItemRect(
   int nIndex,
   LPRECT lpRect
) const;
```

Parameters

nIndex

Zero-based index of the button for which to retrieve information.

lpRect

Address of a RECT structure or a CRect object that receives the coordinates of the bounding rectangle.

Return Value

Nonzero if successful; otherwise zero.

Remarks

This function does not retrieve the bounding rectangle for buttons whose state is set to **TBSTATE_HIDDEN**.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::GetButton
CToolBarCtrl::GetButtonCount
CToolBarCtrl::GetState
CToolBarCtrl::SetButtonSize
CToolBarCtrl::SetBitmapSize

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetMaxSize

Retrieves the total size of all of the visible buttons and separators in the toolbar.

```
BOOL GetMaxSize(
   LPSIZE pSize
) const;
```

Parameters

pSize

A pointer to a SIZE structure that receives the size of the items.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function implements the behavior of the Win32 message TB_GETMAXSIZE, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class

Hierarchy Chart

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetMaxTextRows

Retrieves the maximum number of text rows displayed on a toolbar button.

```
int GetMaxTextRows( ) const;
```

Return Value

The maximum number of text rows displayed on a toolbar button.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::SetMaxTextRows

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetMetrics

Retrieves the metrics of the **CToolBarCtrl** object.

```
void GetMetrics(
   LPTBMETRICS ptbm
) const;
```

Parameters

ptbm

A pointer to the TBMETRICS structure of the **CToolBarCtrl** object.

Remarks

This member function emulates the functionality of the TB_GETMETRICS message, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class

Hierarchy Chart

CToolBarCtrl::SetMetrics

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetRect

Retrieves the bounding rectangle for a specified toolbar button.

```
BOOL GetRect(
   int nID,
   LPRECT lpRect
) const;
```

Parameters

nID

The button identifier.

lpRect

A pointer to a RECT structure to receive the bounding rectangle information.

Return Value

TRUE if successful; otherwise **FALSE**.

Remarks

This member function implements the behavior of the Win32 message TB_GETRECT, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class

Hierarchy Chart

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetRows

Retrieves the number of rows of buttons currently displayed by the toolbar control.

```
int GetRows( ) const;
```

Return Value

Number of rows of buttons currently displayed on the toolbar.

Remarks

Note that the number of rows will always be one unless the toolbar was created with the **TBSTYLE_WRAPABLE** style.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::Create
CToolBarCtrl::SetRows

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetState

Retrieves information about the state of the specified button in a toolbar control, such as whether it is enabled, pressed, or checked.

```
int GetState(
   int nID
) const;
```

Parameters

nID

Command identifier of the button for which to retrieve information.

Return Value

The button state information if successful or – 1 otherwise. The button state information can be a combination of the values listed in CToolBarCtrl::AddButtons.

Remarks

This function is especially handy if you want to retrieve more than one of the button states. To just retrieve one state, use one of the following member functions: IsButtonEnabled, IsButtonChecked, IsButtonPressed, IsButtonHidden, or IsButtonIndeterminate. However, the **GetState** member function is the only way to detect the **TBSTATE_WRAP** button state.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::SetState
CToolBarCtrl::GetItemRect
CToolBarCtrl::IsButtonEnabled
CToolBarCtrl::IsButtonChecked
CToolBarCtrl::IsButtonPressed
CToolBarCtrl::IsButtonHidden
CToolBarCtrl::IsButtonIndeterminate

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetString

Retrieves a toolbar string.

```
int GetString(
   int nString,
   LPTSTR lpstrString,
   int cchMaxLen
) const;
int GetString(
   int nString,
   CString& str
) const;
```

Parameters

nString

Index of the string.

lpstrString

Pointer to a buffer used to return the string.

cchMaxLen

Length of the buffer in bytes.

str

The string.

Return Value

The length of the string if successful, -1 if not.

Remarks

This member function implements the behavior of the Win32 message TB_GETSTRING, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::AddString

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetStyle

Gets the styles currently applied to a toolbar control.

```
DWORD GetStyle( ) const;
```

Return Value

A **DWORD** containing a combination of toolbar control styles, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::SetStyle

Other Resources

CToolBarCtrl Members

CToolBarCtrl::GetToolTips

Retrieves the handle of the tool tip control, if any, associated with the toolbar control.

```
CToolTipCtrl* GetToolTips( ) const;
```

Return Value

A pointer to the CToolTipCtrl object associated with this toolbar or **NULL** if the toolbar has no associated tool tip control.

Remarks

Since the toolbar control normally creates and maintains its own tool tip control, most programs don't need to call this function.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::SetToolTips
Handling Tool Tip Notifications
CToolTipCtrl Class

Other Resources

CToolBarCtrl Members

CToolBarCtrl::HitTest

Determines where a point lies in a toolbar control.

```
int HitTest(
   LPPOINT ppt
) const;
```

Parameters

ppt

A pointer to a POINT structure that contains the x-coordinate of the hit test in the **x** member and the y-coordinate of the hit test in the **y** member. The coordinates are relative to the toolbar's client area.

Return Value

An integer value indicating the location of a point on a toolbar. If the value is zero or a positive value, this return value is the zero-based index of the nonseparator item in which the point lies.

If the return value is negative, the point does not lie within a button. The absolute value of the return value is the index of a separator item or the nearest nonseparator item.

Remarks

This member function implements the behavior of the Win32 message TB_HITTEST, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class

Hierarchy Chart

Other Resources

CToolBarCtrl Members

CToolBarCtrl::HideButton

Hides or shows the specified button in a toolbar control.

```
BOOL HideButton(
   int nID,
   BOOL bHide = TRUE
);
```

Parameters

nID

Command identifier of the button to hide or show.

bHide

TRUE to hide the button, **FALSE** to show it.

Return Value

Nonzero if successful; otherwise zero.

Remarks

If you want to change more than one button state, consider calling SetState instead.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::IsButtonHidden
CToolBarCtrl::EnableButton
CToolBarCtrl::CheckButton
CToolBarCtrl::PressButton
CToolBarCtrl::Indeterminate
CToolBarCtrl::GetState
CToolBarCtrl::SetState

Other Resources

CToolBarCtrl Members

CToolBarCtrl::Indeterminate

Sets or clears the indeterminate state of the specified button in a toolbar control.

```
BOOL Indeterminate(
   int nID,
   BOOL bIndeterminate = TRUE
);
```

Parameters

nID

Command identifier of the button whose indeterminate state is to be set or cleared.

bIndeterminate

TRUE to set the indeterminate state for the specified button, **FALSE** to clear it.

Return Value

Nonzero if successful; otherwise zero.

Remarks

Indeterminate buttons are displayed grayed, such as the way the bold button on the toolbar of a word processor would look when the text selected contains both bold and regular characters. If you want to change more than one button state, consider calling SetState instead.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::AddButtons
CToolBarCtrl::IsButtonIndeterminate
CToolBarCtrl::EnableButton
CToolBarCtrl::CheckButton
CToolBarCtrl::PressButton
CToolBarCtrl::HideButton
CToolBarCtrl::GetState
CToolBarCtrl::SetState

Other Resources

CToolBarCtrl Members

CToolBarCtrl::InsertButton

Inserts a button in a toolbar control.

```
BOOL InsertButton(
   int nIndex,
   LPTBBUTTON lpButton
);
```

Parameters

nIndex

Zero-based index of a button. This function inserts the new button to the left of this button.

lpButton

Address of a **TBBUTTON** structure containing information about the button to insert. See CToolBarCtrl::AddButtons for a description of the **TBBUTTON** structure.

Return Value

Nonzero if successful; otherwise zero.

Remarks

The image and/or string whose index you provide must have previously been added to the toolbar control's list using AddBitmap, AddString, and/or AddStrings.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::AddButtons
CToolBarCtrl::DeleteButton
CToolBarCtrl::AddBitmap
CToolBarCtrl::AddString
CToolBarCtrl::AddStrings

Other Resources

CToolBarCtrl Members

CToolBarCtrl::InsertMarkHitTest

Retrieves the insertion mark information for a point in a toolbar.

```
BOOL InsertMarkHitTest(
   LPPOINT ppt,
   LPTBINSERTMARK ptbim
) const;
```

Parameters

ppt

A pointer to a POINT structure that contains the hit test coordinates, relative to the client area of the toolbar.

ptbim

A pointer to a TBINSERTMARK structure that receives the insertion mark information.

Return Value

Nonzero if successful; otherwise zero.

Remarks

This member function implements the behavior of the Win32 message TB_INSERTMARKHITTEST, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::GetInsertMark
CToolBarCtrl::SetInsertMark

Other Resources

CToolBarCtrl Members

CToolBarCtrl::IsButtonChecked

Determines whether the specified button in a toolbar control is checked.

```
BOOL IsButtonChecked(
   int nID
) const;
```

Parameters

nID

Command identifier of the button in the toolbar.

Return Value

Nonzero if the button is checked; otherwise zero.

Remarks

Consider calling GetState if you want to retrieve more than one button state.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::CheckButton
CToolBarCtrl::GetState
CToolBarCtrl::SetState
CToolBarCtrl::IsButtonEnabled
CToolBarCtrl::IsButtonPressed
CToolBarCtrl::IsButtonHidden
CToolBarCtrl::IsButtonIndeterminate
CToolBarCtrl::IsButtonHighlighted

Other Resources

CToolBarCtrl Members

CToolBarCtrl::IsButtonEnabled

Determines whether the specified button in a toolbar control is enabled.

```
BOOL IsButtonEnabled(
   int nID
) const;
```

Parameters

nID

Command identifier of the button in the toolbar.

Return Value

Nonzero if the button is enabled; otherwise zero.

Remarks

Consider calling GetState if you want to retrieve more than one button state.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::EnableButton
CToolBarCtrl::GetState
CToolBarCtrl::SetState
CToolBarCtrl::IsButtonChecked
CToolBarCtrl::IsButtonPressed
CToolBarCtrl::IsButtonHidden
CToolBarCtrl::IsButtonIndeterminate
CToolBarCtrl::IsButtonHighlighted

Other Resources

CToolBarCtrl Members

CToolBarCtrl::IsButtonHidden

Determines whether the specified button in a toolbar control is hidden.

```
BOOL IsButtonHidden(
   int nID
) const;
```

Parameters

nID

Command identifier of the button in the toolbar.

Return Value

Nonzero if the button is hidden; otherwise zero.

Remarks

Consider calling GetState if you want to retrieve more than one button state.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::HideButton
CToolBarCtrl::GetState
CToolBarCtrl::SetState
CToolBarCtrl::IsButtonEnabled
CToolBarCtrl::IsButtonChecked
CToolBarCtrl::IsButtonPressed
CToolBarCtrl::IsButtonIndeterminate
CToolBarCtrl::IsButtonHighlighted

Other Resources

CToolBarCtrl Members

CToolBarCtrl::IsButtonHighlighted

Checks the highlight state of a toolbar button.

```
BOOL IsButtonHighlighted(
   int nID
) const;
```

Parameters

nID

The command ID for the toolbar button.

Return Value

Nonzero if the button is highlighted; otherwise 0.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::IsButtonChecked
CToolBarCtrl::IsButtonIndeterminate
CToolBarCtrl::IsButtonHidden
CToolBarCtrl::IsButtonEnabled
CToolBarCtrl::IsButtonPressed
CToolBarCtrl::MarkButton

Other Resources

CToolBarCtrl Members

CToolBarCtrl::IsButtonIndeterminate

Determines whether the specified button in a toolbar control is indeterminate.

```
BOOL IsButtonIndeterminate(
   int nID
) const;
```

Parameters

nID

Command identifier of the button in the toolbar.

Return Value

Nonzero if the button is indeterminate; otherwise zero.

Remarks

Indeterminate buttons are displayed grayed, such as the way the bold button on the toolbar of a word processor would look when the text selected contains both bold and regular characters. Consider calling GetState if you want to retrieve more than one button state.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::Indeterminate
CToolBarCtrl::GetState
CToolBarCtrl::SetState
CToolBarCtrl::IsButtonEnabled
CToolBarCtrl::IsButtonChecked
CToolBarCtrl::IsButtonPressed
CToolBarCtrl::IsButtonHidden
CToolBarCtrl::IsButtonHighlighted

Other Resources

CToolBarCtrl Members

CToolBarCtrl::IsButtonPressed

Determines whether the specified button in a toolbar control is pressed.

```
BOOL IsButtonPressed(
   int nID
) const;
```

Parameters

nID

Command identifier of the button in the toolbar.

Return Value

Nonzero if the button is pressed, otherwise zero.

Remarks

Consider calling GetState if you want to retrieve more than one button state.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::PressButton
CToolBarCtrl::GetState
CToolBarCtrl::SetState
CToolBarCtrl::IsButtonEnabled
CToolBarCtrl::IsButtonChecked
CToolBarCtrl::IsButtonHidden
CToolBarCtrl::IsButtonIndeterminate
CToolBarCtrl::IsButtonHighlighted

Other Resources

CToolBarCtrl Members

CToolBarCtrl::LoadImages

Loads bitmaps into a toolbar control's image list.

```
void LoadImages(
   int iBitmapID,
   HINSTANCE hinst
);
```

Parameters

iBitmapID

ID of a bitmap that contains the images to be loaded. To specify your own bitmap resource, set this parameter to the ID of a bitmap resource and set *hInst* to **NULL**. Your bitmap resource will be added to the image list as a single image. You can add standard, system-defined bitmaps by setting *hinst* to **HINST_COMMCTRL** and setting this parameter to one of the following IDs:

Bitmap ID	Description
IDB_HIST_LARGE_COLOR	Explorer bitmaps in large size
IDB_HIST_SMALL_COLOR	Explorer bitmaps in small size
IDB_STD_LARGE_COLOR	Standard bitmaps in large size
IDB_STD_SMALL_COLOR	Standard bitmaps in small size
IDB_VIEW_LARGE_COLOR	View bitmaps in large size
IDB_VIEW_SMALL_COLOR	View bitmaps in small size

hinst

Program instance handle to the calling application. This parameter can be **HINST_COMMCTRL** to load a standard image list.

Remarks

This member function implements the behavior of the Win32 message TB_LOADIMAGES, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::SetImageList
CToolBarCtrl::GetImageList

Other Resources

CToolBarCtrl Members

CToolBarCtrl::MapAccelerator

Maps an accelerator character to a toolbar button.

```
BOOL MapAccelerator(
   TCHAR chAccel,
   UINT* pIDBtn
);
```

Parameters

chAccel

Accelerator character to be mapped. This character is the same character that is underlined in the button's text.

pIDBtn

A pointer to a **UINT** that receives the command identifier of the button that corresponds to the accelerator specified in *chAccel*.

Return Value

Nonzero if successful; otherwise zero.

Remarks

This member function implements the behavior of the Win32 message TB_MAPACCELERATOR, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class

Hierarchy Chart

Other Resources

CToolBarCtrl Members

CToolBarCtrl::MarkButton

Sets the highlight state of a given button in a toolbar control.

```
BOOL MarkButton(
   int nID,
   BOOL fHighlight = TRUE
);
```

Parameters

nID

The button identifier.

fHighlight

Specifies the highlight state to be set. By default, **TRUE**. If set to **FALSE**, the button is set to its default state.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function implements the behavior of the Win32 message TB_MARKBUTTON, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::IsButtonHighlighted

Other Resources

CToolBarCtrl Members

CToolBarCtrl::MoveButton

Moves a button from one index to another.

```
BOOL MoveButton(
   UINT nOldPos,
   UINT nNewPos
);
```

Parameters

nOldPos

The zero-based index of the button to be moved.

nNewPos

The zero-based index of the button's destination.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function implements the behavior of the Win32 message TB_MOVEBUTTON, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class

Hierarchy Chart

Other Resources

CToolBarCtrl Members

CToolBarCtrl::PressButton

Presses or releases the specified button in a toolbar control.

```
BOOL PressButton(
   int nID,
   BOOL bPress = TRUE
);
```

Parameters

nID

Command identifier of the button to press or release.

bPress

TRUE to press the specified button; **FALSE** to release the specified button.

Return Value

Nonzero if successful; otherwise zero.

Remarks

If you want to change more than one button state, consider calling SetState instead.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::IsButtonPressed
CToolBarCtrl::EnableButton
CToolBarCtrl::CheckButton
CToolBarCtrl::HideButton
CToolBarCtrl::Indeterminate
CToolBarCtrl::GetState
CToolBarCtrl::SetState

Other Resources

CToolBarCtrl Members

CToolBarCtrl::RestoreState

Restores the state of the toolbar control from the location in the registry specified by the parameters.

```
void RestoreState(
   HKEY hKeyRoot,
   LPCTSTR lpszSubKey,
   LPCTSTR lpszValueName
);
```

Parameters

hKeyRoot

Identifies a currently open key in the registry or any of the following predefined reserved handle values:

- **HKEY_CLASSES_ROOT**
- **HKEY_CURRENT_USER**
- **HKEY_LOCAL_MACHINE**
- **HKEY_USERS**

lpszSubKey

Points to a null-terminated string containing the name of the subkey with which a value is associated. This parameter can be null or a pointer to an empty string. If the parameter is **NULL**, the value will be added to the key identified by the *hKeyRoot* parameter.

lpszValueName

Points to a string containing the name of the value to retrieve. If a value with this name is not already present in the key, the function adds it to the key.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::SaveState

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SaveState

Saves the state of the toolbar control in the location in the registry specified by the parameters.

```
void SaveState(
   HKEY hKeyRoot,
   LPCTSTR lpszSubKey,
   LPCTSTR lpszValueName
);
```

Parameters

hKeyRoot

Identifies a currently open key in the registry or any of the following predefined reserved handle values:

- **HKEY_CLASSES_ROOT**
- **HKEY_CURRENT_USER**
- **HKEY_LOCAL_MACHINE**
- **HKEY_USERS**

lpszSubKey

Points to a null-terminated string containing the name of the subkey with which a value is associated. This parameter can be null or a pointer to an empty string. If the parameter is **NULL**, the value will be added to the key identified by the *hKeyRoot* parameter.

lpszValueName

Points to a string containing the name of the value to set. If a value with this name is not already present in the key, the function adds it to the key.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::RestoreState

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetAnchorHighlight

Sets the anchor highlight setting for a toolbar.

```
BOOL SetAnchorHighlight(
   BOOL fAnchor = TRUE
);
```

Parameters

fAnchor

Specifies if anchor highlighting is enabled or disabled. If this value is nonzero, anchor highlighting will be enabled. If this value is zero, anchor highlighting will be disabled

Return Value

Nonzero if successful; otherwise zero.

Remarks

This member function implements the behavior of the Win32 message TB_SETANCHORHIGHLIGHT, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::GetAnchorHighlight

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetBitmapSize

Sets the size of the actual bitmapped images to be added to a toolbar control.

```
BOOL SetBitmapSize(
   CSize size
);
```

Parameters

size

Width and height, in pixels, of the bitmapped images.

Return Value

Nonzero if successful; otherwise zero.

Remarks

This function must be called only before adding any bitmaps to the toolbar. If the application does not explicitly set the bitmap size, it defaults to 16 by 15 pixels.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::SetButtonSize
CToolBarCtrl::GetItemRect

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetButtonInfo

Sets the information for an existing button in a toolbar.

```
BOOL SetButtonInfo(
   int nID,
   TBBUTTONINFO* ptbbi
);
```

Parameters

nID

The button identifier.

ptbbi

A pointer to a TBBUTTONINFO structure that receives the button information.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The member function implements the behavior of the Win32 message TB_SETBUTTONINFO, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::GetButtonInfo

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetButtonSize

Sets the size of the buttons in the toolbar control.

```
BOOL SetButtonSize(
   CSize size
);
```

Parameters

size

Width and height, in pixels, of the buttons.

Return Value

Nonzero if successful; otherwise zero.

Remarks

The button size must always be at least as large as the bitmap size it encloses. This function must be called only before adding any bitmaps to the toolbar. If the application does not explicitly set the button size, it defaults to 24 by 22 pixels.

Example

See the example for CToolBar::GetToolBarCtrl.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::SetBitmapSize
CToolBarCtrl::GetItemRect

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetButtonStructSize

Specifies the size of the **TBBUTTON** structure.

```
void SetButtonStructSize(
   int nSize
);
```

Parameters

nSize

Size, in bytes, of the **TBBUTTON** structure.

Remarks

If you wanted to store extra data in the **TBBUTTON** structure, you could either derive a new structure from **TBBUTTON**, adding the members you needed, or create a new structure that contains a **TBBUTTON** structure as its first member. You would then call this function to tell the toolbar control the size of the new structure.

See CToolBarCtrl::AddButtons for more information on the **TBBUTTON** structure.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::Create
CToolBarCtrl::AddButtons
CToolBarCtrl::InsertButton
CToolBarCtrl::GetButton

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetButtonWidth

Sets the minimum and maximum button widths in the toolbar control.

```
BOOL SetButtonWidth(
   int cxMin,
   int cxMax
);
```

Parameters

cxMin

Minimum button width, in pixels. Toolbar buttons will never be narrower than this value.

cxMax

Maximum button width, in pixels. If button text is too wide, the control displays it with ellipsis points.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function implements the behavior of the Win32 message TB_SETBUTTONWIDTH, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::SetButtonInfo

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetCmdID

Sets the command identifier that will be sent to the owner window when the specified button is pressed.

```
BOOL SetCmdID(
   int nIndex,
   UINT nID
);
```

Parameters

nIndex

The zero-based index of the button whose command ID is to be set.

nID

The command ID to set the selected button to.

Return Value

Returns nonzero if successful; otherwise zero.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::CommandToIndex
CToolBarCtrl::GetButton
CToolBarCtrl::AddButtons
CToolBarCtrl::InsertButton

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetDisabledImageList

Sets the image list that the toolbar control will use to display disabled buttons.

```
CImageList* SetDisabledImageList(
   CImageList* pImageList
);
```

Parameters

pImageList

A pointer to a **CImageList** object containing the images to be used by the toolbar control to display disabled button images.

Return Value

A pointer to a CImageList object that was previously used by the toolbar control to display disabled button images.

Remarks

This member function implements the behavior of the Win32 message TB_SETDISABLEDIMAGELIST, as described in the *Platform SDK*. The MFC implementation of **SetDisabledImageList** uses a **CImageList** object containing the toolbar control's disabled button images, rather than a handle to an image list.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::GetDisabledImageList
CToolBarCtrl::SetHotImageList
CToolBarCtrl::SetImageList

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetDrawTextFlags

Sets the flags in the Win32 function DrawText, which is used to draw the text in the specified rectangle, formatted according to how the flags are set.

```
DWORD SetDrawTextFlags(
   DWORD dwMask,
   DWORD dwDTFlags
);
```

Parameters

dwMask

A combination of one or more of the DT_ flags, specified in the Win32 function DrawText, that indicates which bits in *dwDTFlags* will be used when drawing the text.

dwDTFlags

A combination of one or more of the DT_ flags, specified in the Win32 function **DrawText**, that indicate how the button text will be drawn. This value is passed to **DrawText** when the button text is drawn.

Return Value

A **DWORD** containing the previous text drawing flags.

Remarks

This member function implements the behavior of the Win32 message TB_SETDRAWTEXTFLAGS, as described in the *Platform SDK*. This member function sets the flags in the Win32 function **DrawText**, which draws text in the specified rectangle, formatted according to how the flags are set.

See Also

Reference

CToolBarCtrl Class

Hierarchy Chart

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetExtendedStyle

Sets the extended styles for a toolbar control.

```
DWORD SetExtendedStyle(
   DWORD dwExStyle
);
```

Parameters

dwExStyle

A value specifying the new extended styles. This parameter can be a combination of the toolbar extended styles.

Return Value

A **DWORD** that represents the previous extended styles. For a list of styles, see Toolbar Extended Styles, in the *Platform SDK*.

Remarks

This member function implements the behavior of the Win32 message TB_SETEXTENDEDSTYLE, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::GetExtendedStyle

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetHotImageList

Sets the image list that the toolbar control will use to display "hot" buttons.

```
CImageList* SetHotImageList(
   CImageList* pImageList
);
```

Parameters

pImageList

A pointer to a **CImageList** object containing the images to be used by the toolbar control to display hot button images.

Return Value

A pointer to a CImageList object that was previously used by the toolbar control to display hot button images.

Remarks

This member function implements the behavior of the Win32 message TB_SETHOTIMAGELIST, as described in the *Platform SDK*.

The MFC implementation of **SetHotImageList** uses a **CImageList** object containing the toolbar control's hot button images, rather than a handle to an image list. A hot button appears highlighted when the pointer is above it.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::GetHotImageList
CToolBarCtrl::SetDisabledImageList
CToolBarCtrl::SetImageList

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetHotItem

Sets the hot item in a toolbar.

```
int SetHotItem(
   int nHot
);
```

Parameters

nHot

The zero-based index number of the item that will be made hot. If this value is -1, none of the items will be hot.

Return Value

The index of the previous hot item, or -1 if there was no hot item.

Remarks

This member function implements the behavior of the Win32 message TB_SETHOTITEM, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::GetHotItem

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetImageList

Sets the image list that the toolbar will use to display buttons that are in their default state.

```
CImageList* SetImageList(
   CImageList* pImageList
);
```

Parameters

pImageList

A pointer to a **CImageList** object containing the images to be used by the toolbar control to display button images in their default state.

Return Value

A pointer to a CImageList object that was previously used by the toolbar control to display button images in their default state.

Remarks

This member function implements the behavior of the Win32 message TB_SETIMAGELIST, as described in the *Platform SDK*.

The MFC implementation of **SetImageList** uses a **CImageList** object containing the toolbar control's button images, rather than a handle to an image list.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::GetImageList
CToolBarCtrl::SetDisabledImageList
CToolBarCtrl::SetHotImageList

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetIndent

Sets the indentation for the first button in a toolbar control.

```
BOOL SetIndent(
   int iIndent
);
```

Parameters

iIndent

The value specifying the indentation, in pixels.

Return Value

Nonzero if successful; otherwise zero.

See Also

Reference

CToolBarCtrl Class

Hierarchy Chart

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetInsertMark

Sets the current insertion mark for the toolbar.

```
void SetInsertMark(
   TBINSERTMARK* ptbim
);
```

Parameters

ptbim

A pointer to the TBINSERTMARK structure that contains the insertion mark.

Remarks

This member function implements the behavior of the Win32 message TB_SETINSERTMARK, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::GetInsertMark

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetInsertMarkColor

Sets the color used to draw the insertion mark for the toolbar.

```
COLORREF SetInsertMarkColor(
   COLORREF clrNew
);
```

Parameters

clrNew

A **COLORREF** value that contains the new insertion mark color.

Return Value

A **COLORREF** value that contains the previous insertion mark color.

Remarks

This member function implements the behavior of the Win32 message TB_SETINSERTMARKCOLOR, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::GetInsertMarkColor

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetMaxTextRows

Sets the maximum number of text rows displayed on a toolbar button.

```
BOOL SetMaxTextRows(
   int iMaxRows
);
```

Parameters

iMaxRows

Maximum number of rows to be set.

Return Value

Nonzero if successful; otherwise zero.

See Also

Reference

CToolBarCtrl Class

Hierarchy Chart

CToolBarCtrl::GetMaxTextRows

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetMetrics

Sets the metrics of the **CToolBarCtrl** object.

```
void SetMetrics(
   LPTBMETRICS ptbm
);
```

Parameters

ptbm

A pointer to the TBMETRICS structure of the **CToolBarCtrl** object.

Remarks

This member function emulates the functionality of the TB_SETMETRICS message, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::GetMetrics

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetOwner

Sets the owner window for the toolbar control.

```
void SetOwner(
   CWnd* pWnd
);
```

Parameters

pWnd

Pointer to the **CWnd** or **CWnd**-derived object that will be the new owner window for the toolbar control.

Remarks

The owner window is the window that receives notifications from the toolbar.

See Also

Reference

CToolBarCtrl Class

Hierarchy Chart

CToolBarCtrl::Create

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetRows

Asks the toolbar control to resize itself to the requested number of rows.

```
void SetRows(
   int nRows,
   BOOL bLarger,
   LPRECT lpRect
);
```

Parameters

nRows

Requested number of rows.

bLarger

Tells whether to use more rows or fewer rows if the toolbar cannot be resized to the requested number of rows.

lpRect

Points to the CRect object or RECT structure that will receive the new bounding rectangle of the toolbar.

Remarks

If the toolbar cannot resize itself to the requested number or rows, it will resize itself to either the next larger or next smaller valid size, depending on the value of *bLarger*. If *bLarger* is **TRUE**, the new number of rows will be larger than the number requested. If *bLarger* is **FALSE**, the new number of rows will be smaller than the number requested.

A given number of rows is valid for the toolbar if the buttons can be arranged such that all of the rows have the same number of buttons (except perhaps the last row). For example, a toolbar that contains four buttons could not be sized to three rows because the last two rows would have to be shorter. If you attempted to size it to three rows, you would get four rows if *bLarger* was **TRUE** and two rows if *bLarger* was **FALSE**.

If there are separators in the toolbar, the rules for when a given number of rows is valid are more complicated. The layout is computed such that button groups (buttons with a separator before the first and the last button in the group) are never broken up on several rows unless the group cannot fit on one row.

If a group does not fit on one row, the next group will start on the next row even if it would fit on the row where the large group ended. The purpose of this rule is to make the separation between large groups more noticeable. The resulting vertical separators are counted as rows.

Note also that the **SetRows** member function will always chose the layout that results in the smallest toolbar size. Creating a toolbar with the **TBSTYLE_WRAPABLE** style and then resizing the control will simply apply the method outlined above given the width of the control.

This function can only be called for toolbars that were created with the **TBSTYLE_WRAPABLE** style.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::Create
CToolBarCtrl::GetRows

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetState

Sets the state for the specified button in a toolbar control.

```
BOOL SetState(
   int nID,
   UINT nState
);
```

Parameters

nID

Command identifier of the button.

nState

State flags. It can be a combination of the values listed for button states in CToolBarCtrl::AddButtons.

Return Value

Nonzero if successful; otherwise zero.

Remarks

This function is especially handy if you want to set more than one of the button states. To just set one state, use one of the following member functions: EnableButton, CheckButton, HideButton, Indeterminate, or PressButton.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::GetState
CToolBarCtrl::AddButtons
CToolBarCtrl::EnableButton
CToolBarCtrl::CheckButton
CToolBarCtrl::HideButton
CToolBarCtrl::Indeterminate
CToolBarCtrl::PressButton

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetStyle

Sets the styles for a toolbar control.

```
void SetStyle(
   DWORD dwStyle
);
```

Parameters

dwStyle

A **DWORD** containing a combination of toolbar control styles, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class

Hierarchy Chart

CToolBarCtrl::GetStyle

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetToolTips

Associates a tool tip control with a toolbar control.

```
void SetToolTips(
   CToolTipCtrl* pTip
);
```

Parameters

pTip

Pointer to the CToolTipCtrl object.

See Also

Reference

CToolBarCtrl Class
Hierarchy Chart
CToolBarCtrl::GetToolTips
Handling Tool Tip Notifications
CToolTipCtrl Class

Other Resources

CToolBarCtrl Members

CToolBarCtrl::SetWindowTheme

Sets the visual style of the **CToolBarCtrl** object.

```
HRESULT SetWindowTheme(
   LPCWSTR pszSubAppName
);
```

Parameters

pszSubAppName

A pointer to a Unicode string that contains the toolbar visual style to set.

Return Value

The return value is not used.

Remarks

This member function emulates the functionality of the TB_SETWINDOWTHEME message, as described in the *Platform SDK*.

See Also

Reference

CToolBarCtrl Class

Hierarchy Chart

Other Resources

CToolBarCtrl Members

CToolTipCtrl Class

Encapsulates the functionality of a "tool tip control," a small pop-up window that displays a single line of text describing the purpose of a tool in an application.

```
class CToolTipCtrl : public CWnd
```

Remarks

A "tool" is either a window, such as a child window or control, or an application-defined rectangular area within a window's client area. A tool tip is hidden most of the time, appearing only when the user puts the cursor on a tool and leaves it there for approximately one-half second. The tool tip appears near the cursor and disappears when the user clicks a mouse button or moves the cursor off the tool.

CToolTipCtrl provides the functionality to control the initial time and duration of the tool tip, the margin widths surrounding the tool tip text, the width of the tool tip window itself, and the background and text color of the tool tip. A single tool tip control can provide information for more than one tool.

The **CToolTipCtrl** class provides the functionality of the Windows common tool tip control. This control (and therefore the **CToolTipCtrl** class) is available only to programs running under Windows 95/98 and Windows NT versions 3.51 and later.

For more information about enabling tool tips, see Tool Tips in Windows not Derived from CFrameWnd.

For more information on using **CToolTipCtrl**, see Controls and Using CToolTipCtrl.

Requirements

Header: afxcmn.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CWnd Class
Hierarchy Chart
CToolBar Class

Other Resources

CToolTipCtrl Members

CToolTipCtrl Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Construction

Create	Creates a tool tip control and attaches it to a **CToolTipCtrl** object.
CreateEx	Creates a tool tip control with the specified Windows extended styles and attaches it to a **CToolTipCtrl** object.
CToolTipCtrl	Constructs a **CToolTipCtrl** object.

Attributes

GetBubbleSize	Retrieves the size of the tool tip.
GetDelayTime	Retrieves the initial, pop-up, and reshow durations currently set for a tool tip control.
GetMargin	Retrieves the top, left, bottom, and right margins set for a tool tip window.
GetMaxTipWidth	Retrieves the maximum width for a tool tip window.
GetText	Retrieves the text that a tool tip control maintains for a tool.
GetTipBkColor	Retrieves the background color in a tool tip window.
GetTipTextColor	Retrieves the text color in a tool tip window.
GetToolCount	Retrieves a count of the tools maintained by a tool tip control.
GetToolInfo	Retrieves the information that a tool tip control maintains about a tool.
SetDelayTime	Sets the initial, pop-up, and reshow durations for a tool tip control.
SetMargin	Sets the top, left, bottom, and right margins for a tool tip window.
SetMaxTipWidth	Sets the maximum width for a tool tip window.
SetTipBkColor	Sets the background color in a tool tip window.
SetTipTextColor	Sets the text color in a tool tip window.
SetToolInfo	Sets the information that a tool tip maintains for a tool.
SetWindowTheme	Sets the visual style of the tool tip window.

Operations

Activate	Activates and deactivates the tool tip control.
AddTool	Registers a tool with the tool tip control.

AdjustRect	Converts between a tool tip control's text display rectangle and its window rectangle.
DelTool	Removes a tool from the tool tip control.
HitTest	Tests a point to determine whether it is within the bounding rectangle of the given tool and, if so, retrieves information about the tool.
Pop	Removes a displayed tool tip window from view.
RelayEvent	Passes a mouse message to a tool tip control for processing.
SetTitle	Adds a standard icon and title string to a tool tip.
SetToolRect	Sets a new bounding rectangle for a tool.
Update	Forces the current tool to be redrawn.
UpdateTipText	Sets the tool tip text for a tool.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart

CToolTipCtrl Member Functions

For information about the member functions in **CToolTipCtrl**, see CToolTipCtrl Members.

CToolTipCtrl::Activate

Call this function to activate or deactivate a tool tip control.

```
void Activate(
   BOOL bActivate
);
```

Parameters

bActivate

Specifies whether the tool tip control is to be activated or deactivated.

Remarks

If *bActivate* is **TRUE**, the control is activated; if **FALSE**, it is deactivated.

When a tool tip control is active, the tool tip information appears when the cursor is on a tool that is registered with the control; when it is inactive, the tool tip information does not appear, even when the cursor is on a tool.

Example

See the example for CPropertySheet::GetTabControl.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::UpdateTipText
CToolTipCtrl::SetDelayTime

Other Resources

CToolTipCtrl Members

CToolTipCtrl::AddTool

Registers a tool with the tool tip control.

```
BOOL AddTool(
   CWnd* pWnd,
   UINT nIDText,
   LPCRECT lpRectTool = NULL,
   UINT_PTR nIDTool = 0
);
BOOL AddTool(
   CWnd* pWnd,
   LPCTSTR lpszText = LPSTR_TEXTCALLBACK,
   LPCRECT lpRectTool = NULL,
   UINT_PTR nIDTool = 0
);
```

Parameters

pWnd

Pointer to the window that contains the tool.

nIDText

ID of the string resource that contains the text for the tool.

lpRectTool

Pointer to a RECT structure containing coordinates of the tool's bounding rectangle. The coordinates are relative to the upper-left corner of the client area of the window identified by *pWnd*.

nIDTool

ID of the tool.

lpszText

Pointer to the text for the tool. If this parameter contains the value **LPSTR_TEXTCALLBACK**, **TTN_NEEDTEXT** notification messages go to the parent of the window that *pWnd* points to.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The **lpRectTool** and **nIDTool** parameters must both be valid, or if **lpRectTool** is NULL, **nIDTool** must be 0.

A tool tip control can be associated with more than one tool. Call this function to register a tool with the tool tip control, so that the information stored in the tool tip is displayed when the cursor is on the tool.

Note
You cannot set a tool tip to a static control using **AddTool**.

Example

See the example for CPropertySheet::GetTabControl.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::DelTool

Other Resources

CToolTipCtrl Members

CToolTipCtrl::AdjustRect

Converts between a tooltip control's text display rectangle and its window rectangle.

```
BOOL AdjustRect(
   LPRECT lprc,
   BOOL bLarger = TRUE
);
```

Parameters

lprc

Pointer to a RECT structure that holds either a tool tip window rectangle or a text display rectangle.

bLarger

If **TRUE**, *lprc* is used to specify a text-display rectangle, and it receives the corresponding window rectangle. If **FALSE**, *lprc* is used to specify a window rectangle, and it receives the corresponding text display rectangle.

Return Value

Nonzero if the rectangle is successfully adjusted; otherwise 0.

Remarks

This member function calculates a tool tip control's text display rectangle from its window rectangle, or the tool tip window rectangle needed to display a specified text display rectangle.

This member function implements the behavior of the Win32 message TTM_ADJUSTRECT, as described in the *Platform SDK*.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::GetBubbleSize

Other Resources

CToolTipCtrl Members

CToolTipCtrl::Create

Creates a tool tip control and attaches it to a **CToolTipCtrl** object.

```
virtual BOOL Create(
   CWnd* pParentWnd,
      DWORD dwStyle = 0
);
```

Parameters

pParentWnd

Specifies the tool tip control's parent window, usually a **CDialog**. It must not be **NULL**.

dwStyle

Specifies the tool tip control's style. See the **Remarks** section for more information.

Return Value

Nonzero if the **CToolTipCtrl** object is successfully created; otherwise 0.

Remarks

You construct a **CToolTipCtrl** in two steps. First, call the constructor to construct the **CToolTipCtrl** object, and then call **Create** to create the tool tip control and attach it to the **CToolTipCtrl** object.

The *dwStyle* parameter can be any combination of Window Styles. In addition, a tool tip control has two class-specific styles: **TTS_ALWAYSTIP** and **TTS_NOPREFIX**.

Style	Meaning
TTS_ALWAYSTIP	Specifies that the tool tip will appear when the cursor is on a tool, regardless of whether the tool tip control's owner window is active or inactive. Without this style, the tool tip control appears when the tool's owner window is active, but not when it is inactive.
TTS_NOPREFIX	This style prevents the system from stripping the ampersand (&) character from a string. If a tool tip control does not have the **TTS_NOPREFIX** style, the system automatically strips ampersand characters, allowing an application to use the same string as both a menu item and as text in a tool tip control.

A tool tip control has the **WS_POPUP** and **WS_EX_TOOLWINDOW** window styles, regardless of whether you specify them when creating the control.

To create a tool tip control with extended windows styles, call CToolTipCtrl::CreateEx instead of **Create**.

Example

See the example for CPropertySheet::GetTabControl.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::CToolTipCtrl

Other Resources

CToolTipCtrl Members

CToolTipCtrl::CreateEx

Creates a control (a child window) and associate it with the **CToolTipCtrl** object.

```
virtual BOOL CreateEx(
   CWnd* pParentWnd,
   DWORD dwStyle = 0,
   DWORD dwStyleEx = 0
);
```

Parameters

pParentWnd

A pointer to the window that is the control's parent.

dwStyle

Specifies the tool tip control's style. See the **Remarks** section of Create for more information.

dwStyleEx

Specifies the extended style of the control being created. For a list of extended Windows styles, see the *dwExStyle* parameter for CreateWindowEx in the *Platform SDK*.

Return Value

Nonzero if successful otherwise 0.

Remarks

Use **CreateEx** instead of **Create** to apply extended Windows styles, specified by the Windows extended style preface **WS_EX_**.

See Also

Reference

CStatusBarCtrl Class
Hierarchy Chart
CToolTipCtrl::CToolTipCtrl

Other Resources

CStatusBarCtrl Members

CToolTipCtrl::CToolTipCtrl

Constructs a **CToolTipCtrl** object.

```
CToolTipCtrl( );
```

Remarks

You must call **Create** after constructing the object.

Example

```
// Declare a CToolTipCtrl object.
CToolTipCtrl  ToolTipCtrl;
```

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::Create
CToolTipCtrl::CreateEx

Other Resources

CToolTipCtrl Members

CToolTipCtrl::DelTool

Removes the tool specified by *pWnd* and *nIDTool* from the collection of tools supported by a tool tip control.

```
void DelTool(
   CWnd* pWnd,
   UINT_PTR nIDTool = 0
);
```

Parameters

pWnd

Pointer to the window that contains the tool.

nIDTool

ID of the tool.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::AddTool

Other Resources

CToolTipCtrl Members

CToolTipCtrl::GetBubbleSize

Retrieves the size of the tool tip.

```
CSize GetBubbleSize(
   LPTOOLINFO lpToolInfo
) const;
```

Parameters

lpToolInfo

A pointer to the tool tip's TOOLINFO structure.

Return Value

The size of the tool tip.

Remarks

This member function implements the behavior of the Win32 message TTM_GETBUBBLESIZE, as described in the *Platform SDK*.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::AdjustRect

Other Resources

CToolTipCtrl Members

CToolTipCtrl::GetDelayTime

Retrieves the initial, pop-up, and reshow durations currently set for a tool tip control.

```
int GetDelayTime(
   DWORD dwDuration
) const;
```

Parameters

dwDuration

Flag that specifies which duration value will be retrieved. This parameter can be one of the following values:

- **TTDT_AUTOPOP** Retrieve the length of time the tool tip window remains visible if the pointer is stationary within a tool's bounding rectangle.
- **TTDT_INITIAL** Retrieve the length of time the pointer must remain stationary within a tool's bounding rectangle before the tool tip window appears.
- **TTDT_RESHOW** Retrieve the length of time it takes for subsequent tool tip windows to appear as the pointer moves from one tool to another.

Return Value

The specified delay time, in milliseconds

Remarks

This member function implements the behavior of the Win32 message TTM_GETDELAYTIME, as described in the *Platform SDK*.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::SetDelayTime

Other Resources

CToolTipCtrl Members

CToolTipCtrl::GetMargin

Retrieves the top, left, bottom, and right margins set for a tool tip window.

```
void GetMargin(
   LPRECT lprc
) const;
```

Parameters

lprc

Address of a **RECT** structure that will receive the margin information. The members of the RECT structure do not define a bounding rectangle. For the purpose of this message, the structure members are interpreted as follows:

Member	Representation
top	Distance between top border and top of tool tip text, in pixels.
left	Distance between left border and left end of tip text, in pixels.
bottom	Distance between bottom border and bottom of tip text, in pixels.
right	Distance between right border and right end of tip text, in pixels.

Remarks

This member function implements the behavior of the Win32 message TTM_GETMARGIN, as described in the *Platform SDK*.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::SetMargin

Other Resources

CToolTipCtrl Members

CToolTipCtrl::GetMaxTipWidth

Retrieves the maximum width for a tool tip window.

```
int GetMaxTipWidth( ) const;
```

Return Value

The maximum width for a tool tip window.

Remarks

This member function implements the behavior of the Win32 message TTM_GETMAXTIPWIDTH, as described in the *Platform SDK*.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::SetMaxTipWidth

Other Resources

CToolTipCtrl Members

CToolTipCtrl::GetText

Retrieves the text that a tool tip control maintains for a tool.

```
void GetText(
   CString& str,
   CWnd* pWnd,
   UINT_PTR nIDTool = 0
) const;
```

Parameters

str

Reference to a **CString** object that receives the tool's text.

pWnd

Pointer to the window that contains the tool.

nIDTool

ID of the tool.

Remarks

The *pWnd* and *nIDTool* parameters identify the tool. If that tool has been previously registered with the tool tip control through a previous call to **CToolTipCtrl::AddTool**, the object referenced by the *str* parameter is assigned the tool's text.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::AddTool
CToolTipCtrl::DelTool

Other Resources

CToolTipCtrl Members

CToolTipCtrl::GetTipBkColor

Retrieves the background color in a tool tip window.

```
COLORREF GetTipBkColor( ) const;
```

Return Value

A COLORREF value that represents the background color.

Remarks

This member function implements the behavior of the Win32 message TTM_GETTIPBKCOLOR, as described in the *Platform SDK*.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::SetTipBkColor

Other Resources

CToolTipCtrl Members

CToolTipCtrl::GetTipTextColor

Retrieves the text color in a tool tip window.

```
COLORREF GetTipTextColor( ) const;
```

Return Value

A COLORREF value that represents the text color.

Remarks

This member function implements the behavior of the Win32 message TTM_GETTIPTEXTCOLOR, as described in the *Platform SDK*.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::SetTipTextColor

Other Resources

CToolTipCtrl Members

CToolTipCtrl::GetToolCount

Retrieves a count of the tools registered with the tool tip control.

```
int GetToolCount( ) const;
```

Return Value

A count of tools registered with the tool tip control.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::AddTool
CToolTipCtrl::DelTool

Other Resources

CToolTipCtrl Members

CToolTipCtrl::GetToolInfo

Retrieves the information that a tool tip control maintains about a tool.

```
BOOL GetToolInfo(
   CToolInfo& ToolInfo,
   CWnd* pWnd,
   UINT_PTR nIDTool = 0
) const;
```

Parameters

ToolInfo

Reference to a **TOOLINFO** object that receives the tool's text.

pWnd

Pointer to the window that contains the tool.

nIDTool

ID of the tool.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The **hwnd** and **uId** members of the TOOLINFO structure referenced by *CToolInfo* identify the tool. If that tool has been registered with the tool tip control through a previous call to **AddTool**, the **TOOLINFO** structure is filled with information about the tool.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::AddTool

Other Resources

CToolTipCtrl Members

CToolTipCtrl::HitTest

Tests a point to determine whether it is within the bounding rectangle of the given tool and, if so, retrieve information about the tool.

```
BOOL HitTest(
   CWnd* pWnd,
   CPoint pt,
   LPTOOLINFO lpToolInfo
) const;
```

Parameters

pWnd

Pointer to the window that contains the tool.

pt

Pointer to a **CPoint** object containing the coordinates of the point to be tested.

lpToolInfo

Pointer to TOOLINFO structure that contains information about the tool.

Return Value

Nonzero if the point specified by the hit-test information is within the tool's bounding rectangle; otherwise 0.

Remarks

If this function returns a nonzero value, the structure pointed to by *lpToolInfo* is filled with information on the tool within whose rectangle the point lies.

The **TTHITTESTINFO** structure is defined as follows:

```
typedef struct _TT_HITTESTINFO { // tthti
    HWND hwnd;   // handle of tool or window with tool
    POINT pt;    // client coordinates of point to test
    TOOLINFO ti; // receives information about the tool
} TTHITTESTINFO, FAR * LPHITTESTINFO;
```

hwnd

Specifies the tool's handle.

pt

Specifies the coordinates of a point if the point is in the tool's bounding rectangle.

ti

Information about the tool. For more information about the **TOOLINFO** structure, see CToolTipCtrl::GetToolInfo.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::GetToolInfo

Other Resources

CToolTipCtrl Members

CToolTipCtrl::Pop

Removes a displayed tool tip window from the view.

```
void Pop( );
```

Remarks

This member function implements the behavior of the Win32 message TTM_POP, as described in the *Platform SDK*.

See Also

Reference

CToolTipCtrl Class

Hierarchy Chart

Other Resources

CToolTipCtrl Members

CToolTipCtrl::RelayEvent

Passes a mouse message to a tool tip control for processing.

```
void RelayEvent(
   LPMSG lpMsg
);
```

Parameters

lpMsg

Pointer to a MSG structure that contains the message to relay.

Remarks

A tool tip control processes only the following messages, which are sent to it by **RelayEvent**:

WM_LBUTTONDOWN	**WM_MOUSEMOVE**
WM_LBUTTONUP	**WM_RBUTTONDOWN**
WM_MBUTTONDOWN	**WM_RBUTTONUP**
WM_MBUTTONUP	

Example

See the example for CPropertySheet::GetTabControl.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CWnd::PreTranslateMessage
CWinApp::PreTranslateMessage

Other Resources

CToolTipCtrl Members

CToolTipCtrl::SetDelayTime

Sets the delay time for a tool tip control.

```
void SetDelayTime(
   UINT nDelay
);
void SetDelayTime(
   DWORD dwDuration,
   int iTime
);
```

Parameters

nDelay

Specifies the new delay time, in milliseconds.

dwDuration

Flag that specifies which duration value will be retrieved. See CToolTipCtrl::GetDelayTime for a description of the valid values.

iTime

The specified delay time, in milliseconds.

Remarks

The delay time is the length of time the cursor must remain on a tool before the tool tip window appears. The default delay time is 500 milliseconds.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::Activate
CToolTipCtrl::HitTest
CToolTipCtrl::GetDelayTime
TTM_SETDELAYTIME

Other Resources

CToolTipCtrl Members

CToolTipCtrl::SetMargin

Sets the top, left, bottom, and right margins for a tool tip window.

```
void SetMargin(
   LPRECT lprc
);
```

Parameters

lprc

Address of a **RECT** structure that contains the margin information to be set. The members of the **RECT** structure do not define a bounding rectangle. See CToolTipCtrl::GetMargin for a description of the margin information.

Remarks

This member function implements the behavior of the Win32 message TTM_SETMARGIN, as described in the *Platform SDK*.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::GetMargin

Other Resources

CToolTipCtrl Members

CToolTipCtrl::SetMaxTipWidth

Sets the maximum width for a tool tip window.

```
int SetMaxTipWidth(
   int iWidth
);
```

Parameters

iWidth

The maximum tool tip window width to be set.

Return Value

The previous maximum tip width.

Remarks

This member function implements the behavior of the Win32 message TTM_SETMAXTIPWIDTH, as described in the *Platform SDK*.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::GetMaxTipWidth

Other Resources

CToolTipCtrl Members

CToolTipCtrl::SetTipBkColor

Sets the background color in a tool tip window.

```
void SetTipBkColor(
   COLORREF clr
);
```

Parameters

clr

The new background color.

Remarks

This member function implements the behavior of the Win32 message TTM_SETTIPBKCOLOR, as described in the *Platform SDK*.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::GetTipBkColor

Other Resources

CToolTipCtrl Members

CToolTipCtrl::SetTipTextColor

Sets the text color in a tool tip window.

```
void SetTipTextColor(
   COLORREF clr
);
```

Parameters

clr

The new text color.

Remarks

This member function implements the behavior of the Win32 message TTM_SETTIPTEXTCOLOR, as described in the *Platform SDK*.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::GetTipTextColor

Other Resources

CToolTipCtrl Members

CToolTipCtrl::SetTitle

Adds a standard icon and title string to a tool tip.

```
BOOL SetTitle(
   UINT uIcon,
   LPCTSTR lpstrTitle
);
```

Parameters

uIcon

See *icon* in TTM_SETTITLE in the *Platform SDK*.

lpstrTitle

Pointer to the title string.

Return Value

Nonzero if successful; otherwise 0.

Remarks

This member function implements the behavior of the Win32 message TTM_SETTITLE, as described in the *Platform SDK*.

See Also

Reference

CToolTipCtrl Class

Hierarchy Chart

Other Resources

CToolTipCtrl Members

CToolTipCtrl::SetToolInfo

Sets the information that a tool tip maintains for a tool.

```
void SetToolInfo(
   LPTOOLINFO lpToolInfo
);
```

Parameters

lpToolInfo

A pointer to a TOOLINFO structure that specifies the information to set.

See Also

Reference

CToolTipCtrl Class

Hierarchy Chart

CToolTipCtrl::GetToolInfo

Other Resources

CToolTipCtrl Members

CToolTipCtrl::SetToolRect

Sets a new bounding rectangle for a tool.

```
void SetToolRect(
   CWnd* pWnd,
   UINT_PTR nIDTool,
   LPCRECT lpRect
);
```

Parameters

pWnd

Pointer to the window that contains the tool.

nIDTool

ID of the tool.

lpRect

Pointer to a RECT structure specifying the new bounding rectangle.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::GetToolInfo

Other Resources

CToolTipCtrl Members

CToolTipCtrl::SetWindowTheme

Sets the visual style of the tool tip window.

```
HRESULT SetWindowTheme(
   LPCWSTR pszSubAppName
);
```

Parameters

pszSubAppName

A pointer to a Unicode string that contains the visual style to set.

Return Value

The return value is not used.

Remarks

This member function emulates the functionality of the TTM_SETWINDOWTHEME message, as described in the *Platform SDK*.

See Also

Reference

CToolTipCtrl Class

Hierarchy Chart

Other Resources

CToolTipCtrl Members

CToolTipCtrl::Update

Forces the current tool to be redrawn.

```
void Update( );
```

See Also

Reference

CToolTipCtrl Class

Hierarchy Chart

CToolTipCtrl::UpdateTipText

Other Resources

CToolTipCtrl Members

CToolTipCtrl::UpdateTipText

Updates the tool tip text for this control's tools.

```
void UpdateTipText(
   LPCTSTR lpszText,
   CWnd* pWnd,
   UINT_PTR nIDTool = 0
);
void UpdateTipText(
   UINT nIDText,
   CWnd* pWnd,
   UINT_PTR nIDTool = 0
);
```

Parameters

lpszText

Pointer to the text for the tool.

pWnd

Pointer to the window that contains the tool.

nIDTool

ID of the tool.

nIDText

ID of the string resource that contains the text for the tool.

See Also

Reference

CToolTipCtrl Class
Hierarchy Chart
CToolTipCtrl::GetToolInfo

Other Resources

CToolTipCtrl Members

CTreeCtrl Class

Provides the functionality of the Windows common tree view control.

```
class CTreeCtrl : public CWnd
```

Remarks

A "tree view control" is a window that displays a hierarchical list of items, such as the headings in a document, the entries in an index, or the files and directories on a disk. Each item consists of a label and an optional bitmapped image, and each item can have a list of subitems associated with it. By clicking an item, the user can expand and collapse the associated list of subitems.

This control (and therefore the **CTreeCtrl** class) is available only to programs running under Windows 98 and Windows NT version 4 and later.

For more information on using **CTreeCtrl**, see:

- Controls
- Using CTreeCtrl
- Tree View Control Reference in the *Platform SDK*.
- Knowledge Base article Q222905 : HOWTO: Display a Context Menu for CTreeCtrl

Requirements

Header: afxcmn.h

See Also

Tasks

CMNCTRL1 Sample: Demonstrates Common Control MFC Classes, Part 1

Reference

CWnd Class

Hierarchy Chart

CImageList Class

Other Resources

CTreeCtrl Members

CTreeCtrl Members

Base Class Members

Construction

Attributes

Operations

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Construction

Create	Creates a tree view control and attaches it to a **CTreeCtrl** object.
CreateEx	Creates a tree control with the specified Windows extended styles and attaches it to a **CTreeCtrl** object.
CTreeCtrl	Constructs a **CTreeCtrl** object.

Attributes

GetBkColor	Retrieves the current background color of the control.
GetCheck	Retrieves the check state of a tree control item.
GetChildItem	Retrieves the child of a specified tree view item.
GetCount	Retrieves the number of tree items associated with a tree view control.
GetDropHilightItem	Retrieves the target of a drag-and-drop operation.
GetEditControl	Retrieves the handle of the edit control used to edit the specified tree view item.
GetFirstVisibleItem	Retrieves the first visible item of the specified tree view item.
GetImageList	Retrieves the handle of the image list associated with a tree view control.
GetIndent	Retrieves the offset (in pixels) of a tree view item from its parent.
GetInsertMarkColor	Retrieves the color used to draw the insertion mark for the tree view.
GetItem	Retrieves the attributes of a specified tree view item.
GetItemData	Returns the 32-bit application-specific value associated with an item.
GetItemHeight	Retrieves the current height of the tree view items.
GetItemImage	Retrieves the images associated with an item.
GetItemRect	Retrieves the bounding rectangle of a tree view item.
GetItemState	Returns the state of an item.

GetItemText	Returns the text of an item.
GetLineColor	Retrieves the current line color for the tree view control.
GetNextItem	Retrieves the next tree view item that matches a specified relationship.
GetNextSiblingItem	Retrieves the next sibling of the specified tree view item.
GetNextVisibleItem	Retrieves the next visible item of the specified tree view item.
GetParentItem	Retrieves the parent of the specified tree view item.
GetPrevSiblingItem	Retrieves the previous sibling of the specified tree view item.
GetPrevVisibleItem	Retrieves the previous visible item of the specified tree view item.
GetRootItem	Retrieves the root of the specified tree view item.
GetScrollTime	Retrieves the maximum scroll time for the tree view control.
GetSelectedItem	Retrieves the currently selected tree view item.
GetTextColor	Retrieves the current text color of the control.
GetToolTips	Retrieves the handle to the child ToolTip control used by a tree view control.
GetVisibleCount	Retrieves the number of visible tree items associated with a tree view control.
ItemHasChildren	Returns nonzero if the specified item has child items.
SetBkColor	Sets the background color of the control.
SetCheck	Sets the check state of a tree control item.
SetImageList	Sets the handle of the image list associated with a tree view control.
SetIndent	Sets the offset (in pixels) of a tree view item from its parent.
SetInsertMark	Sets the insertion mark in a tree view control.
SetInsertMarkColor	Sets the color used to draw the insertion mark for the tree view.
SetItem	Sets the attributes of a specified tree view item.
SetItemData	Sets the 32-bit application-specific value associated with an item.
SetItemHeight	Sets the height of the tree view items.
SetItemImage	Associates images with an item.
SetItemState	Sets the state of an item.
SetItemText	Sets the text of an item.

SetLineColor	Sets the current line color for the tree view control.
SetScrollTime	Sets the maximum scroll time for the tree view control.
SetTextColor	Sets the text color of the control.
SetToolTips	Sets a tree view control's child ToolTip control.

Operations

CreateDragImage	Creates a dragging bitmap for the specified tree view item.
DeleteAllItems	Deletes all items in a tree view control.
DeleteItem	Deletes a new item in a tree view control.
EditLabel	Edits a specified tree view item in-place.
EnsureVisible	Ensures that a tree view item is visible in its tree view control.
Expand	Expands, or collapses, the child items of the specified tree view item.
HitTest	Returns the current position of the cursor related to the **CTreeCtrl** object.
InsertItem	Inserts a new item in a tree view control.
Select	Selects, scrolls into view, or redraws a specified tree view item.
SelectDropTarget	Redraws the tree item as the target of a drag-and-drop operation.
SelectItem	Selects a specified tree view item.
SelectSetFirstVisible	Selects a specified tree view item as the first visible item.
SortChildren	Sorts the children of a given parent item.
SortChildrenCB	Sorts the children of a given parent item using an application-defined sort function.

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

Concepts

CWnd Members

CTreeCtrl Member Functions

For information about the member functions in **CTreeCtrl**, see CTreeCtrl Members.

CTreeCtrl::Create

If you specify the tree control in a dialog box template, or if you are using CTreeView, your tree control is created automatically when the dialog box or view is created.

```
virtual BOOL Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwStyle

Specifies the tree view control's style. Apply window styles, described in CreateWindow, and any combination of tree view control styles as described in the *Platform SDK*.

rect

Specifies the tree view control's size and position. It can be either a CRect object or a RECT structure.

pParentWnd

Specifies the tree view control's parent window, usually a **CDialog**. It must not be **NULL**.

nID

Specifies the tree view control's ID.

Return Value

Nonzero if initialization was successful; otherwise 0.

Remarks

If you want to create the tree control as a child window of some other window, use the **Create** member function. If you create the tree control using **Create**, you must pass it **WS_VISIBLE**, in addition to other tree view styles.

You construct a **CTreeCtrl** in two steps. First call the constructor, then call **Create**, which creates the tree view control and attaches it to the **CTreeCtrl** object.

To create a tree control with extended window styles, call CreateEx instead of **Create**.

Example

```
// Assuming your window has a CTreeCtrl member named m_TreeCtrl,
// you can create the tree control window with a child ID of 0x1005
// using a call like this:

m_TreeCtrl->Create(WS_VISIBLE | WS_TABSTOP | WS_CHILD | WS_BORDER
   | TVS_HASBUTTONS | TVS_LINESATROOT | TVS_HASLINES
   | TVS_DISABLEDRAGDROP,
   CRect(10, 10, 300, 100), this, 0x1005);

// The control will have the appropiate window styles, and the tree
// control styles specified are those most commonly used.
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::CTreeCtrl

Other Resources

CTreeCtrl Members

CTreeCtrl::CreateEx

Call this function to create a control (a child window) and associate it with the **CTreeCtrl** object.

```
virtual BOOL CreateEx(
   DWORD dwExStyle,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID
);
```

Parameters

dwExStyle

Specifies the extended style of the control being created. For a list of extended Windows styles, see the *dwExStyle* parameter for CreateWindowEx in the *Platform SDK*.

dwStyle

Specifies the tree view control's style. Apply window styles, described in CreateWindow, and any combination of tree view control styles as described in the *Platform SDK*.

rect

A reference to a RECT structure describing the size and position of the window to be created, in client coordinates of *pParentWnd*.

pParentWnd

A pointer to the window that is the control's parent.

nID

The control's child-window ID.

Return Value

Nonzero if successful otherwise 0.

Remarks

Use **CreateEx** instead of Create to apply extended Windows styles, specified by the Windows extended style preface **WS_EX_**.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::CTreeCtrl

Other Resources

CTreeCtrl Members

CTreeCtrl::CreateDragImage

Call this function to create a dragging bitmap for the given item in a tree view control, create an image list for the bitmap, and add the bitmap to the image list.

```
CImageList* CreateDragImage(
   HTREEITEM hItem
);
```

Parameters

hItem

Handle of the tree item to be dragged.

Return Value

Pointer to the image list to which the dragging bitmap was added, if successful; otherwise **NULL**.

Remarks

An application uses the image-list functions to display the image when the item is being dragged.

The **CImageList** object is permanent, and you must delete it when finished. For example:

```
CImageList* pImageList = MyTreeCtrl.CreateDragImage(hItem);
   ...
   ...
   delete pImageList;
```

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::SelectDropTarget
CTreeCtrl::GetDropHilightItem
CTreeCtrl::SetImageList

Other Resources

CTreeCtrl Members

CTreeCtrl::CTreeCtrl

Constructs a **CTreeCtrl** object.

```
CTreeCtrl( );
```

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::Create
CTreeCtrl::CreateEx

Other Resources

CTreeCtrl Members

CTreeCtrl::DeleteAllItems

Call this function to delete all items from the tree view control.

```
BOOL DeleteAllItems( );
```

Return Value

Nonzero if successful; otherwise 0.

Example

```
// Gain a pointer to our tree control.
CTreeCtrl* pCtrl = (CTreeCtrl*) GetDlgItem(IDC_TREE1);

// The underlying Windows API always returns TRUE
VERIFY(pCtrl->DeleteAllItems());
```

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::DeleteItem
CTreeCtrl::InsertItem

Other Resources

CTreeCtrl Members

CTreeCtrl::DeleteItem

Call this function to delete an item from the tree view control.

```
BOOL DeleteItem(
   HTREEITEM hItem
);
```

Parameters

hItem

Handle of the tree item to be deleted. If *hitem* has the **TVI_ROOT** value, all items are deleted from the tree view control.

Return Value

Nonzero if successful; otherwise 0.

Example

```
// CTreeCtrl_DeleteItem.cpp
// compile with: /c
#include "afxcmn.h"
int main() {
   // Gain a pointer to our tree control
   CTreeCtrl* pCtrl = (CTreeCtrl*) GetDlgItem(0, 0);
   ASSERT(pCtrl != NULL);

   // Look at all of the root-level items
   HTREEITEM hCurrent = pCtrl->GetRootItem();
   while (hCurrent != NULL) {
      // Get the text for the item. Notice we use TVIF_TEXT because
      // we want to retrieve only the text, but also specify TVIF_HANDLE
      // because we're getting the item by its handle.
      TVITEM item;
      TCHAR szText[1024];
      item.hItem = hCurrent;
      item.mask = TVIF_TEXT | TVIF_HANDLE;
      item.pszText = szText;
      item.cchTextMax = 1024;

      BOOL bWorked = pCtrl->GetItem(&item);

      // Try to get the next item
      hCurrent = pCtrl->GetNextItem(hCurrent, TVGN_NEXT);

      // If we successfuly retrieved an item, and the item's text
      // contains a lowercase letter 'e', delete the item.
      if (bWorked && _tcschr(item.pszText, 'e'))
         pCtrl->DeleteItem(item.hItem);
   }
}
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::DeleteAllItems

CTreeCtrl::InsertItem

Other Resources

CTreeCtrl Members

CTreeCtrl::EditLabel

Call this function to begin in-place editing of the specified item's text.

```
CEdit* EditLabel(
   HTREEITEM hItem
);
```

Parameters

hItem

Handle of the tree item to be edited.

Return Value

If successful, a pointer to the **CEdit** object that is used to edit the item text; otherwise **NULL**.

Remarks

The editing is accomplished by replacing the text of the item with a single-line edit control containing the text.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;

// Make sure the focus is set to the tree control.
pmyTreeCtrl->SetFocus();

// Show the edit control on the label of the root item.
CEdit* pmyEdit = pmyTreeCtrl->EditLabel(pmyTreeCtrl->GetRootItem());
ASSERT(pmyEdit != NULL);
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::GetEditControl

Other Resources

CTreeCtrl Members

CTreeCtrl::EnsureVisible

Call this function to ensure that a tree view item is visible.

```
BOOL EnsureVisible(
   HTREEITEM hItem
);
```

Parameters

hItem

Handle of the tree item being made visible.

Return Value

Returns **TRUE** if the system scrolled the items in the tree-view control to ensure that the specified item is visible. Otherwise, the return value is **FALSE**.

Remarks

If necessary, the function expands the parent item or scrolls the tree view control so that the item is visible.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
// The item that I want to ensure is visible.
extern HTREEITEM hmyItem;

// Expand the parent, if possible.
HTREEITEM hParent = pmyTreeCtrl->GetParentItem(hmyItem);
if (hParent != NULL)
   pmyTreeCtrl->Expand(hParent, TVE_EXPAND);

// Ensure the item is visible.
pmyTreeCtrl->EnsureVisible(hmyItem);
```

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetFirstVisibleItem
CTreeCtrl::GetVisibleCount

Other Resources

CTreeCtrl Members

CTreeCtrl::Expand

Call this function to expand or collapse the list of child items, if any, associated with the given parent item.

```
BOOL Expand(
   HTREEITEM hItem,
   UINT nCode
);
```

Parameters

hItem

Handle of the tree item being expanded.

nCode

A flag indicating the type of action to be taken. This flag can have one of the following values:

- **TVE_COLLAPSE** Collapses the list.
- **TVE_COLLAPSERESET** Collapses the list and removes the child items. The **TVIS_EXPANDEDONCE** state flag is reset. This flag must be used with the **TVE_COLLAPSE** flag.
- **TVE_EXPAND** Expands the list.
- **TVE_TOGGLE** Collapses the list if it is currently expanded or expands it if it is currently collapsed.

Return Value

Nonzero if successful; otherwise 0.

Example

See the example for CTreeCtrl::EnsureVisible.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::EnsureVisible

Other Resources

CTreeCtrl Members

CTreeCtrl::GetBkColor

This member function implements the behavior of the Win32 message TVM_GETBKCOLOR, as described in the *Platform SDK*.

```
COLORREF GetBkColor( ) const;
```

Return Value

A **COLORREF** value that represents the current background color. If this value is -1, the control is using the system color for the background color.

Example

See the example for CTreeCtrl::SetTextColor.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::SetBkColor

Other Resources

CTreeCtrl Members

CTreeCtrl::GetCheck

Call this member function to retrieve an item's check state.

```
BOOL GetCheck(
   HTREEITEM hItem
) const;
```

Parameters

hItem

The **HTREEITEM** about which to receive the state information.

Return Value

Nonzero if the tree control item is checked; otherwise 0.

Example

See the example for CTreeCtrl::SetCheck.

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::SetCheck

Other Resources

CTreeCtrl Members

CTreeCtrl::GetChildItem

Call this function to retrieve the tree view item that is the child of the item specified by *hItem*.

```
HTREEITEM GetChildItem(
   HTREEITEM hItem
) const;
```

Parameters

hItem

Handle of a tree item.

Return Value

The handle of the child item if successful; otherwise **NULL**.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
// The item whose children will be deleted.
extern HTREEITEM hmyItem;

// Delete all of the children of hmyItem.
if (pmyTreeCtrl->ItemHasChildren(hmyItem))
{
   HTREEITEM hNextItem;
   HTREEITEM hChildItem = pmyTreeCtrl->GetChildItem(hmyItem);

   while (hChildItem != NULL)
   {
      hNextItem = pmyTreeCtrl->GetNextItem(hChildItem, TVGN_NEXT);
      pmyTreeCtrl->DeleteItem(hChildItem);
      hChildItem = hNextItem;
   }
}
```

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetItem
CTreeCtrl::GetParentItem
CTreeCtrl::SortChildren

Other Resources

CTreeCtrl Members

CTreeCtrl::GetCount

Call this function to retrieve a count of the items in a tree view control.

```
UINT GetCount( ) const;
```

Return Value

The number of items in the tree view control.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;

// Delete all of the items from the tree control.
pmyTreeCtrl->DeleteAllItems();
ASSERT(pmyTreeCtrl->GetCount() == 0);
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::GetVisibleCount

Other Resources

CTreeCtrl Members

CTreeCtrl::GetDropHilightItem

Call this function to retrieve the item that is the target of a drag-and-drop operation.

```
HTREEITEM GetDropHilightItem( ) const;
```

Return Value

The handle of the item dropped if successful; otherwise **NULL**.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
// The point to test.
extern CPoint myPoint;

// Set the item at the point myPoint as the drop target.
UINT uFlags;
HTREEITEM hItem = pmyTreeCtrl->HitTest(myPoint, &uFlags);

if ((hItem != NULL) && (TVHT_ONITEM & uFlags))
{
   pmyTreeCtrl->SelectDropTarget(hItem);
   ASSERT(pmyTreeCtrl->GetDropHilightItem() == hItem);
}
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::SelectDropTarget

Other Resources

CTreeCtrl Members

CTreeCtrl::GetEditControl

Call this function to retrieve the handle of the edit control being used to edit a tree view item's text.

```
CEdit* GetEditControl( ) const;
```

Return Value

A pointer to the edit control used to edit the item text, if successful; otherwise **NULL**.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
// The string replacing the text in the edit control.
extern LPCTSTR lpszmyString;

// Replace the text in the label edit control, if possible.
CEdit* pEdit = pmyTreeCtrl->GetEditControl();

if (pEdit != NULL)
{
   pEdit->SetWindowText(lpszmyString);
}
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::EditLabel

Other Resources

CTreeCtrl Members

CTreeCtrl::GetFirstVisibleItem

Call this function to retrieve the first visible item of the tree view control.

```
HTREEITEM GetFirstVisibleItem( ) const;
```

Return Value

The handle of the first visible item; otherwise **NULL**.

Example

See the example for CTreeCtrl::SetCheck.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetNextVisibleItem
CTreeCtrl::GetPrevVisibleItem
CTreeCtrl::EnsureVisible
CTreeCtrl::GetVisibleCount

Other Resources

CTreeCtrl Members

CTreeCtrl::GetImageList

Call this function to retrieve the handle of the normal or state image list associated with the tree view control.

```
CImageList* GetImageList(
   UINT nImageList
) const;
```

Parameters

nImageList

Type of image list to retrieve. The image list can be one of the following values:

- **TVSIL_NORMAL** Retrieves the normal image list, which contains the selected and nonselected images for the tree view item.
- **TVSIL_STATE** Retrieves the state image list, which contains the images for tree view items that are in a user-defined state.

Return Value

Pointer to the control's image list if successful; otherwise **NULL**.

Remarks

Each item in a tree view control can have a pair of bitmapped images associated with it. One image is displayed when the item is selected, and the other is displayed when the item is not selected. For example, an item might display an open folder when it is selected and a closed folder when it is not selected.

For more information on image lists, see the CImageList class.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
// The new image list of the tree control.
extern CImageList* pmyImageList;

ASSERT(pmyTreeCtrl->GetImageList(TVSIL_NORMAL) == NULL);

pmyTreeCtrl->SetImageList(pmyImageList, TVSIL_NORMAL);
ASSERT(pmyTreeCtrl->GetImageList(TVSIL_NORMAL) == pmyImageList);
```

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CImageList Class
CTreeCtrl::SetImageList

Other Resources

CTreeCtrl Members

CTreeCtrl::GetIndent

Call this function to retrieve the amount, in pixels, that child items are indented relative to their parent items.

```
UINT GetIndent( ) const;
```

Return Value

The amount of indentation measured in pixels.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;

// Double the indent.
UINT uIndent = pmyTreeCtrl->GetIndent();
pmyTreeCtrl->SetIndent(2*uIndent);
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::SetIndent

Other Resources

CTreeCtrl Members

CTreeCtrl::GetInsertMarkColor

This member function implements the behavior of the Win32 message TVM_GETINSERTMARKCOLOR, as described in the *Platform SDK*.

```
COLORREF GetInsertMarkColor( ) const;
```

Return Value

A **COLORREF** value that contains the current insertion mark color.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;

// Use the highliight color for the insert mark color.
COLORREF crColor = ::GetSysColor(COLOR_HIGHLIGHT);
pmyTreeCtrl->SetInsertMarkColor(crColor);
ASSERT(pmyTreeCtrl->GetInsertMarkColor() == crColor);
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::SetInsertMarkColor

Other Resources

CTreeCtrl Members

CTreeCtrl::GetItem

Call this function to retrieve the attributes of the specified tree view item.

```
BOOL GetItem(
   TVITEM* pItem
) const;
```

Parameters

pItem

A pointer to a TVITEM structure, as described in the *Platform SDK*.

Return Value

Nonzero if successful; otherwise 0.

Example

See the example for CTreeCtrl::DeleteItem.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::SetItem
CTreeCtrl::GetChildItem
CTreeCtrl::GetNextItem
CTreeCtrl::SelectItem

Other Resources

CTreeCtrl Members

CTreeCtrl::GetItemData

Call this function to retrieve the 32-bit application-specific value associated with the specified item.

```
DWORD_PTR GetItemData(
   HTREEITEM hItem
) const;
```

Parameters

hItem

Handle of the item whose data is to be retrieved.

Return Value

A 32-bit application-specific value associated with the item specified by *hItem*.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
// The item whose siblings will be checked.
extern HTREEITEM hmyItem;

// Delete all of the children of hmyItem whose item data is
// not equal to zero.
if (pmyTreeCtrl->ItemHasChildren(hmyItem))
{
   HTREEITEM hNextItem;
   HTREEITEM hChildItem = pmyTreeCtrl->GetChildItem(hmyItem);

   while (hChildItem != NULL)
   {
      hNextItem = pmyTreeCtrl->GetNextItem(hChildItem, TVGN_NEXT);

      if (pmyTreeCtrl->GetItemData(hChildItem) != 0)
      {
         pmyTreeCtrl->DeleteItem(hChildItem);
      }

      hChildItem = hNextItem;
   }
}
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::SetItemData

Other Resources

CTreeCtrl Members

CTreeCtrl::GetItemHeight

This member function implements the behavior of the Win32 message TVM_GETITEMHEIGHT, as described in the *Platform SDK*.

```
SHORT GetItemHeight( ) const;
```

Return Value

The height of the item, in pixels.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;

// Double the height of the items.
SHORT sHeight = pmyTreeCtrl->GetItemHeight();
pmyTreeCtrl->SetItemHeight(2*sHeight);
```

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::SetItemHeight

Other Resources

CTreeCtrl Members

CTreeCtrl::GetItemImage

Each item in a tree view control can have a pair of bitmapped images associated with it.

```
BOOL GetItemImage(
   HTREEITEM hItem,
   int& nImage,
   int& nSelectedImage
) const;
```

Parameters

hItem

The handle of the item whose image is to be retrieved.

nImage

An integer that receives the index of the item's image within the tree view control's image list.

nSelectedImage

An integer that receives the index of the item's selected image within the tree view control's image list.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The images appear on the left side of an item's label. One image is displayed when the item is selected, and the other is displayed when the item is not selected. For example, an item might display an open folder when it is selected and a closed folder when it is not selected.

Call this function to retrieve the index of the item's image and its selected image within the tree view control's image list.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
// The item whose children will be checked.
extern HTREEITEM hmyItem;

// If the selected image is the same as the nonselected image
// then make the selected image index 0.
if (pmyTreeCtrl->ItemHasChildren(hmyItem))
{
   HTREEITEM hItem = pmyTreeCtrl->GetChildItem(hmyItem);
   int nImage, nSelectedImage;

   while (hItem != NULL)
   {
      pmyTreeCtrl->GetItemImage(hItem, nImage, nSelectedImage);

      if (nImage == nSelectedImage)
      {
         pmyTreeCtrl->SetItemImage(hItem, nImage, 0);
      }

      hItem = pmyTreeCtrl->GetNextSiblingItem(hItem);
   }
}
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::SetItemImage
CImageList Class
Other Resources
CTreeCtrl Members

CTreeCtrl::GetItemRect

Call this function to retrieve the bounding rectangle for *hItem* and determine whether it is visible or not.

```
BOOL GetItemRect(
   HTREEITEM hItem,
   LPRECT lpRect,
   BOOL bTextOnly
) const;
```

Parameters

hItem

The handle of a tree view control item.

lpRect

Pointer to a RECT structure that receives the bounding rectangle. The coordinates are relative to the upper-left corner of the tree view control.

bTextOnly

If this parameter is nonzero, the bounding rectangle includes only the text of the item. Otherwise it includes the entire line that the item occupies in the tree view control.

Return Value

Nonzero if the item is visible, with the bounding rectangle contained in *lpRect*. Otherwise, 0 with *lpRect* uninitialized.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
// The item whose bounds we want.
extern HTREEITEM hmyItem;

// Dump the bounds of hmyItem.
#ifdef _DEBUG
if (hmyItem != NULL)
{
   CString str;
   RECT r;

   pmyTreeCtrl->GetItemRect(hmyItem, &r, FALSE);

   str.Format(TEXT("left = %d, top = %d, right = %d, bottom = %d\r\n"),
      r.left,
      r.top,
      r.right,
      r.bottom);
   afxDump << str;
}
```

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetVisibleCount
CTreeCtrl::GetNextVisibleItem
CTreeCtrl::GetPrevVisibleItem
CTreeCtrl::EnsureVisible

Other Resources

CTreeCtrl Members

CTreeCtrl::GetItemState

Returns the state of the item specified by *hItem*.

```
UINT GetItemState(
   HTREEITEM hItem,
   UINT nStateMask
) const;
```

Parameters

hItem

Handle of the item whose state is to be retrieved.

nStateMask

Mask indicating which states are to be retrieved. For more information on possible values for *nStateMask*, see the discussion of the **state** and **stateMask** members of the TVITEM structure in the *Platform SDK*.

Return Value

A **UINT** specifying the item's state. For information on possible values, see CTreeCtrl::GetItem.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;

// Show all of the visible items in bold.
HTREEITEM hItem = pmyTreeCtrl->GetFirstVisibleItem();

while (hItem != NULL)
{
   pmyTreeCtrl->SetItemState(hItem, TVIS_BOLD, TVIS_BOLD);
   hItem = pmyTreeCtrl->GetNextVisibleItem(hItem);
}
```

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetItem

Other Resources

CTreeCtrl Members

CTreeCtrl::GetItemText

Returns the text of the item specified by *hItem*.

```
CString GetItemText(
   HTREEITEM hItem
) const;
```

Parameters

hItem

Handle of the item whose text is to be retrieved.

Return Value

A **CString** object containing the item's text.

Example

See the example for CTreeCtrl::GetNextItem.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::SetItemText

Other Resources

CTreeCtrl Members

CTreeCtrl::GetLineColor

This member function implements the behavior of the win32 message TVM_GETLINECOLOR, as described in the *Platform SDK*.

```
COLORREF GetLineColor( ) const;
```

Return Value

The current line color.

Example

```
// Gain a pointer to our tree control
CTreeCtrl* pCtrl = (CTreeCtrl*) GetDlgItem(IDC_TREE1);
ASSERT(pCtrl != NULL);
COLORREF cr = pCtrl->GetLineColor();
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::SetLineColor

Other Resources

CTreeCtrl Members

CTreeCtrl::GetNextItem

Call this function to retrieve the tree view item that has the specified relationship, indicated by the *nCode* parameter, to *hItem*.

```
HTREEITEM GetNextItem(
   HTREEITEM hItem,
   UINT nCode
) const;
```

Parameters

hItem

Handle of a tree item.

nCode

A flag indicating the type of relation to *hItem*. This flag can be one of the following values:

- **TVGN_CARET** Retrieves the currently selected item.
- **TVGN_CHILD** Retrieves the first child item of the item specified by the *hItem* parameter.
- **TVGN_DROPHILITE** Retrieves the item that is the target of a drag-and-drop operation.
- **TVGN_FIRSTVISIBLE** Retrieves the first visible item.
- **TVGN_LASTVISIBLE** Retrieves the last expanded item in the tree. This does not retrieve the last item visible in the tree-view window.
- **TVGN_NEXT** Retrieves the next sibling item.
- **TVGN_NEXTVISIBLE** Retrieves the next visible item that follows the specified item.
- **TVGN_PARENT** Retrieves the parent of the specified item.
- **TVGN_PREVIOUS** Retrieves the previous sibling item.
- **TVGN_PREVIOUSVISIBLE** Retrieves the first visible item that precedes the specified item.
- **TVGN_ROOT** Retrieves the first child item of the root item of which the specified item is a part.

Return Value

The handle of the next item if successful; otherwise **NULL**.

Remarks

This function will return **NULL** if the item being retrieved is the root node of the tree. For example, if you use this message with the **TVGN_PARENT** flag on a first-level child of the tree view's root node, the message will return **NULL**.

Example

For an example of using **GetNextItem** in a loop, see CTreeCtrl::DeleteItem.

```
// Gain a pointer to our tree control
CTreeCtrl* pCtrl = (CTreeCtrl*) GetDlgItem(IDC_TREE1);
ASSERT(pCtrl != NULL);

// find the currently selected item
HTREEITEM hCurSel = pCtrl->GetNextItem(TVI_ROOT, TVGN_CARET);

// report it to the user
if (hCurSel == NULL)
   MessageBox(_T("There is no selected item"));
else
{
   CString str;
```

```
    str.Format(_T("The currently selected item is \"%s\""),
       (LPCTSTR) pCtrl->GetItemText(hCurSel));
    MessageBox((LPCTSTR) str);
  }
```

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::SetItem
CTreeCtrl::GetChildItem
CTreeCtrl::GetItem
CTreeCtrl::SelectItem
CTreeCtrl::GetPrevSiblingItem

Other Resources

CTreeCtrl Members

CTreeCtrl::GetNextSiblingItem

Call this function to retrieve the next sibling of *hItem*.

```
HTREEITEM GetNextSiblingItem(
   HTREEITEM hItem
) const;
```

Parameters

hItem

Handle of a tree item.

Return Value

The handle of the next sibling item; otherwise **NULL**.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
// The item whose children will be displayed in bold.
extern HTREEITEM hmyItem;

// Show all of the children of hmyItem in bold.
if (pmyTreeCtrl->ItemHasChildren(hmyItem))
{
   HTREEITEM hItem = pmyTreeCtrl->GetChildItem(hmyItem);

   while (hItem != NULL)
   {
      pmyTreeCtrl->SetItemState(hItem, TVIS_BOLD, TVIS_BOLD);
      hItem = pmyTreeCtrl->GetNextSiblingItem(hItem);
   }
}
```

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetPrevSiblingItem
CTreeCtrl::GetChildItem
CTreeCtrl::GetItem
CTreeCtrl::SelectItem
CTreeCtrl::GetParentItem

Other Resources

CTreeCtrl Members

CTreeCtrl::GetNextVisibleItem

Call this function to retrieve the next visible item of *hItem*.

```
HTREEITEM GetNextVisibleItem(
   HTREEITEM hItem
) const;
```

Parameters

hItem

Handle of a tree item.

Return Value

The handle of the next visible item; otherwise **NULL**.

Example

See the example for CTreeCtrl::SetCheck.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetPrevVisibleItem
CTreeCtrl::GetFirstVisibleItem
CTreeCtrl::EnsureVisible
CTreeCtrl::GetParentItem

Other Resources

CTreeCtrl Members

CTreeCtrl::GetParentItem

Call this function to retrieve the parent of *hItem*.

```
HTREEITEM GetParentItem(
   HTREEITEM hItem
) const;
```

Parameters

hItem

Handle of a tree item.

Return Value

The handle of the parent item; otherwise **NULL**.

Remarks

This function will return **NULL** if the parent of the specified item is the root node of the tree.

Example

See the example for CTreeCtrl::EnsureVisible.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetChildItem
CTreeCtrl::GetRootItem
CTreeCtrl::GetItem
CTreeCtrl::GetPrevSiblingItem

Other Resources

CTreeCtrl Members

CTreeCtrl::GetPrevSiblingItem

Call this function to retrieve the previous sibling of *hItem*.

```
HTREEITEM GetPrevSiblingItem(
   HTREEITEM hItem
) const;
```

Parameters

hItem

Handle of a tree item.

Return Value

The handle of the previous sibling; otherwise **NULL**.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
// The item whose previous siblings will be displayed in bold.
extern HTREEITEM hmyItem;

// Show all of the previous siblings of hmyItem in bold.
HTREEITEM hItem = hmyItem;

while (hItem != NULL)
{
   pmyTreeCtrl->SetItemState(hItem, TVIS_BOLD, TVIS_BOLD);
   hItem = pmyTreeCtrl->GetPrevSiblingItem(hItem);
}
```

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetNextSiblingItem
CTreeCtrl::GetParentItem
CTreeCtrl::GetChildItem

Other Resources

CTreeCtrl Members

CTreeCtrl::GetPrevVisibleItem

Call this function to retrieve the previous visible item of *hItem*.

```
HTREEITEM GetPrevVisibleItem(
   HTREEITEM hItem
) const;
```

Parameters

hItem

Handle of a tree item.

Return Value

The handle of the previous visible item; otherwise **NULL**.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
// The item that previous visible items will be displayed in bold.
extern HTREEITEM hmyItem;

// Show all of the previous visible items of hmyItem in bold.
HTREEITEM hItem = hmyItem;

while (hItem != NULL)
{
   pmyTreeCtrl->SetItemState(hItem, TVIS_BOLD, TVIS_BOLD);
   hItem = pmyTreeCtrl->GetPrevVisibleItem(hItem);
}
```

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetNextVisibleItem
CTreeCtrl::GetFirstVisibleItem
CTreeCtrl::EnsureVisible
CTreeCtrl::GetVisibleCount

Other Resources

CTreeCtrl Members

CTreeCtrl::GetRootItem

Call this function to retrieve the root item of the tree view control.

```
HTREEITEM GetRootItem( ) const;
```

Return Value

The handle of the root item; otherwise **NULL**.

Example

See the example for CTreeCtrl::EditLabel.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetItem
CTreeCtrl::GetChildItem
CTreeCtrl::GetParentItem

Other Resources

CTreeCtrl Members

CTreeCtrl::GetSelectedItem

Call this function to retrieve the currently selected item of the tree view control.

```
HTREEITEM GetSelectedItem( ) const;
```

Return Value

The handle of the selected item; otherwise **NULL**.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;

// Expand the selected item and make it visible, if possible.
HTREEITEM hItem = pmyTreeCtrl->GetSelectedItem();

if ((hItem != NULL) && pmyTreeCtrl->ItemHasChildren(hItem))
{
   pmyTreeCtrl->Expand(hItem, TVE_EXPAND);
   pmyTreeCtrl->EnsureVisible(hItem);
}
```

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::Select
CTreeCtrl::SelectDropTarget
CTreeCtrl::GetDropHilightItem

Other Resources

CTreeCtrl Members

CTreeCtrl::GetScrollTime

Call this member function to retrieve the maximum scroll time for the tree view control.

```
UINT GetScrollTime( ) const;
```

Return Value

The maximum scroll time, in milliseconds.

Remarks

This member function implements the behavior of the win32 message TVM_GETSCROLLTIME, as described in the *Platform SDK*.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::SetScrollTime

Other Resources

CTreeCtrl Members

CTreeCtrl::GetTextColor

This member function implements the behavior of the Win32 message TVM_GETTEXTCOLOR, as described in the *Platform SDK*.

```
COLORREF GetTextColor( ) const;
```

Return Value

A **COLORREF** value that represents the current text color. If this value is -1, the control is using the system color for the text color.

Example

See the example for CTreeCtrl::SetTextColor.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::SetTextColor

Other Resources

CTreeCtrl Members

CTreeCtrl::GetToolTips

This member function implements the behavior of the Win32 message TVM_GETTOOLTIPS, as described in the *Platform SDK*.

```
CToolTipCtrl* GetToolTips( ) const;
```

Return Value

A pointer to a CToolTipCtrl object to be used by the tree control. If the Create member function uses the style **TVS_NOTOOLTIPS**, no tooltips are used, and **NULL** is returned.

Remarks

The MFC implementation of **GetToolTips** returns a **CToolTipCtrl** object, which is used by the tree control, rather than a handle to a tooltip control.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
// A pointer to a tooltips control.
extern CToolTipCtrl* pmyToolTip;

// If the tree control does not have a tooltips control,
// then use pmyToolTip as the tooltips for the tree control.
if (pmyTreeCtrl->GetToolTips() == NULL)
{
   pmyTreeCtrl->SetToolTips(pmyToolTip);
}
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::SetToolTips

Other Resources

CTreeCtrl Members

CTreeCtrl::GetVisibleCount

Call this function to retrieve a count of the visible items in a tree view control.

```
UINT GetVisibleCount( ) const;
```

Return Value

The number of visible items in the tree view control; otherwise – 1.

Example

See the example for CTreeCtrl::SetCheck.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetCount
CTreeCtrl::EnsureVisible

Other Resources

CTreeCtrl Members

CTreeCtrl::HitTest

Call this function to determine the location of the specified point relative to the client area of a tree view control.

```
HTREEITEM HitTest(
   CPoint pt,
   UINT* pFlags = NULL
) const;
HTREEITEM HitTest(
   TVHITTESTINFO* pHitTestInfo
) const;
```

Parameters

pt

Client coordinates of the point to test.

pFlags

Pointer to an integer that receives information about the results of the hit test. It can be one or more of the values listed under the **flags** member in the Remarks section.

pHitTestInfo

Address of a TVHITTESTINFO structure that contains the position to hit test and that receives information about the results of the hit test.

Return Value

The handle of the tree view item that occupies the specified point or **NULL** if no item occupies the point.

Remarks

When this function is called, the *pt* parameter specifies the coordinates of the point to test. The function returns the handle of the item at the specified point or **NULL** if no item occupies the point. In addition, the *pFlags* parameter contains a value that indicates the location of the specified point.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
// The point to test.
extern CPoint myPoint;

// Select the item that is at the point myPoint.
UINT uFlags;
HTREEITEM hItem = pmyTreeCtrl->HitTest(myPoint, &uFlags);

if ((hItem != NULL) && (TVHT_ONITEM & uFlags))
{
   pmyTreeCtrl->Select(hItem, TVGN_CARET);
}
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::GetItemRect

Other Resources

CTreeCtrl Members

CTreeCtrl::InsertItem

Call this function to insert a new item in a tree view control.

```
HTREEITEM InsertItem(
   LPTVINSERTSTRUCT lpInsertStruct
);
HTREEITEM InsertItem(
   UINT nMask,
   LPCTSTR lpszItem,
   int nImage,
   int nSelectedImage,
   UINT nState,
   UINT nStateMask,
   LPARAM lParam,
   HTREEITEM hParent,
   HTREEITEM hInsertAfter
);
HTREEITEM InsertItem(
   LPCTSTR lpszItem,
   HTREEITEM hParent = TVI_ROOT,
   HTREEITEM hInsertAfter = TVI_LAST
);
HTREEITEM InsertItem(
   LPCTSTR lpszItem,
   int nImage,
   int nSelectedImage,
   HTREEITEM hParent = TVI_ROOT,
   HTREEITEM hInsertAfter = TVI_LAST
);
```

Parameters

lpInsertStruct

A pointer to a TVINSERTSTRUCT that specifies the attributes of the tree view item to be inserted.

nMask

Integer specifying which attributes to set. See the TVITEM structure in the *Platform SDK*.

lpszItem

Address of a string containing the item's text.

nImage

Index of the item's image in the tree view control's image list.

nSelectedImage

Index of the item's selected image in the tree view control's image list.

nState

Specifies values for the item's states. See Tree View Control Item States in the *Platform SDK* for a list of appropriate states.

nStateMask

Specifies which states are to be set. See the TVITEM structure in the *Platform SDK*.

lParam

A 32-bit application-specific value associated with the item.

hParent

Handle of the inserted item's parent.

hInsertAfter

Handle of the item after which the new item is to be inserted.

Return Value

Handle of the new item if successful; otherwise **NULL**.

Remarks

The example shows situations in which you might want to use each version of the function when inserting a tree control item.

Example

```
// Gain a pointer to our tree control

CTreeCtrl* pCtrl = (CTreeCtrl*) GetDlgItem(IDC_TREE1);
ASSERT(pCtrl != NULL);

// Insert a root item using the structure. We must
// initialize a TVINSERTSTRUCT structure and pass its
// address to the call.

TVINSERTSTRUCT tvInsert;
tvInsert.hParent = NULL;
tvInsert.hInsertAfter = NULL;
tvInsert.item.mask = TVIF_TEXT;
tvInsert.item.pszText = _T("United States");

HTREEITEM hCountry = pCtrl->InsertItem(&tvInsert);

// Insert subitems of that root. Pennsylvania is
// a state in the United States, so its item will be a child
// of the United States item. We won't set any image or states,
// so we supply only the TVIF_TEXT mask flag. This
// override provides nearly complete control over the
// insertion operation without the tedium of initializing
// a structure. If you're going to add lots of items
// to a tree, you might prefer the structure override
// as it affords you a performance win by allowing you
// to initialize some fields of the structure only once,
// outside of your insertion loop.

HTREEITEM hPA = pCtrl->InsertItem(TVIF_TEXT,
   _T("Pennsylvania"), 0, 0, 0, 0, 0, hCountry, NULL);

// Insert the "Washington" item and assure that it is
// inserted after the "Pennsylvania" item. This override is
// more appropriate for conveniently inserting items with
// images.

HTREEITEM hWA = pCtrl->InsertItem(_T("Washington"),
   0, 0, hCountry, hPA);

// We'll add some cities under each of the states.
// The override used here is most appropriate
// for inserting text-only items.

pCtrl->InsertItem(_T("Pittsburgh"), hPA, TVI_SORT);
pCtrl->InsertItem(_T("Harrisburg"), hPA, TVI_SORT);
pCtrl->InsertItem(_T("Altoona"), hPA, TVI_SORT);

pCtrl->InsertItem(_T("Seattle"), hWA, TVI_SORT);
pCtrl->InsertItem(_T("Kalaloch"), hWA, TVI_SORT);
pCtrl->InsertItem(_T("Yakima"), hWA, TVI_SORT);
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart
CTreeCtrl::DeleteItem
CTreeCtrl::HitTest
CTreeCtrl::SelectDropTarget
CTreeCtrl::GetItem
Tree View Control Reference
Other Resources
CTreeCtrl Members

CTreeCtrl::ItemHasChildren

Use this function to determine whether the tree item specified by *hItem* has child items.

```
BOOL ItemHasChildren(
   HTREEITEM hItem
) const;
```

Parameters

hItem

Handle of a tree item.

Return Value

Nonzero if the tree item specified by *hItem* has child items; 0 if it does not.

Remarks

If so, you can then use CTreeCtrl::GetChildItem to retrieve those child items.

Example

See the example for CTreeCtrl::GetSelectedItem.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetChildItem

Other Resources

CTreeCtrl Members

CTreeCtrl::Select

Call this function to select the given tree view item, scroll the item into view, or redraw the item in the style used to indicate the target of a drag-and-drop operation.

```
BOOL Select(
   HTREEITEM hItem,
   UINT nCode
);
```

Parameters

hItem

Handle of a tree item.

nCode

The type of action to take. This parameter can be one of the following values:

- **TVGN_CARET** Sets the selection to the given item.
- **TVGN_DROPHILITE** Redraws the given item in the style used to indicate the target of a drag-and-drop operation.
- **TVGN_FIRSTVISIBLE** Scrolls the tree view vertically so that the given item is the first visible item.

Return Value

Nonzero if successful; otherwise 0.

Remarks

If *nCode* contains the value **TVGN_CARET**, the parent window receives the **TVN_SELCHANGING** and **TVN_SELCHANGED** notification messages. In addition, if the specified item is the child of a collapsed parent item, the parent's list of child items is expanded to reveal the specified item. In this case, the parent window receives the **TVN_ITEMEXPANDING** and **TVN_ITEMEXPANDED** notification messages.

Example

See the example for CTreeCtrl::HitTest.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::SelectItem
CTreeCtrl::GetSelectedItem
CTreeCtrl::SelectDropTarget

Other Resources

CTreeCtrl Members

CTreeCtrl::SelectDropTarget

Call this function to redraw the item in the style used to indicate the target of a drag-and-drop operation.

```
BOOL SelectDropTarget(
   HTREEITEM hItem
);
```

Parameters

hItem

Handle of a tree item.

Return Value

Nonzero if successful; otherwise 0.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
// The point to test.
extern CPoint myPoint;

// Set the item at the point myPoint as the drop target.
UINT uFlags;
HTREEITEM hItem = pmyTreeCtrl->HitTest(myPoint, &uFlags);

if ((hItem != NULL) && (TVHT_ONITEM & uFlags))
{
   pmyTreeCtrl->SelectDropTarget(hItem);
}
```

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::SelectItem
CTreeCtrl::GetDropHilightItem
CTreeCtrl::CreateDragImage

Other Resources

CTreeCtrl Members

CTreeCtrl::SelectItem

Call this function to select the given tree view item.

```
BOOL SelectItem(
   HTREEITEM hItem
);
```

Parameters

hItem

Handle of a tree item.

Return Value

Nonzero if successful; otherwise 0.

Remarks

If *hItem* is **NULL**, then this function selects no item.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
// The point to test.
extern CPoint myPoint;

// Select the item that is at the point myPoint.
UINT uFlags;
HTREEITEM hItem = pmyTreeCtrl->HitTest(myPoint, &uFlags);

if ((hItem != NULL) && (TVHT_ONITEM & uFlags))
{
   pmyTreeCtrl->SelectItem(hItem);
}
```

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::Select
CTreeCtrl::GetSelectedItem
CTreeCtrl::SelectDropTarget

Other Resources

CTreeCtrl Members

CTreeCtrl::SelectSetFirstVisible

Call this function to scroll the tree view vertically so that the given item is the first visible item.

```
BOOL SelectSetFirstVisible(
   HTREEITEM hItem
);
```

Parameters

hItem

Handle of the tree item to be set as the first visible item.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The function sends a message to the window with the **TVM_SELECTITEM** and **TVGN_FIRSTVISIBLE** message parameters.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
// The point to test.
extern CPoint myPoint;

// Select the item at the point myPoint as the first visible item.
UINT uFlags;
HTREEITEM hItem = pmyTreeCtrl->HitTest(myPoint, &uFlags);

if ((hItem != NULL) && (TVHT_ONITEM & uFlags))
{
   pmyTreeCtrl->SelectSetFirstVisible(hItem);
}
```

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::Select
CTreeCtrl::SelectItem
CTreeCtrl::SelectDropTarget

Other Resources

CTreeCtrl Members

CTreeCtrl::SetBkColor

This member function implements the behavior of the Win32 message TVM_SETBKCOLOR, as described in the *Platform SDK*.

```
COLORREF SetBkColor(
   COLORREF clr
);
```

Parameters

clr

A **COLORREF** value that contains the new background color. If this value is -1, the control will revert to using the system color for the background color.

Return Value

A **COLORREF** value that represents the current text color. If this value is -1, the control is using the system color for the text color.

Example

See the example for CTreeCtrl::SetTextColor.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetBkColor

Other Resources

CTreeCtrl Members

CTreeCtrl::SetCheck

Call this member function to set the check state for a tree control item.

```
BOOL SetCheck(
   HTREEITEM hItem,
   BOOL fCheck = TRUE
);
```

Parameters

hItem

The **HTREEITEM** to receive the check state change.

fCheck

Indicates whether the tree control item is to be checked or unchecked. By default, **SetCheck** sets the item to be checked.

Return Value

Nonzero if successful; otherwise 0.

Remarks

When the tree control item is checked (*fCheck* set to **TRUE**), the item appears with an adjacent checkmark.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;

UINT i, uCount = pmyTreeCtrl->GetVisibleCount();
HTREEITEM hItem = pmyTreeCtrl->GetFirstVisibleItem();

// Toggle the check state of all the visible items.
for (i=0;i < uCount;i++)
{
   ASSERT(hItem != NULL);
   pmyTreeCtrl->SetCheck(hItem, !pmyTreeCtrl->GetCheck(hItem));
   hItem = pmyTreeCtrl->GetNextVisibleItem(hItem);
}
```

To use checkboxes, set TVS_CHECKBOXES before populating the tree control.

```
BOOL CMyDialog::OnInitDialog()
{
CDialog::OnInitDialog();

mTree.ModifyStyle( TVS_CHECKBOXES, 0 );
mTree.ModifyStyle( 0, TVS_CHECKBOXES );

HTREEITEM aItem = mTree.InsertItem("AAA");
mTree.SetCheck(aItem);
return TRUE;
}
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::GetCheck

Other Resources

CTreeCtrl Members

CTreeCtrl::SetImageList

Call this function to set the normal or state image list for a tree view control and redraw the control using the new images.

```
CImageList* SetImageList(
   CImageList * pImageList,
   int nImageListType
);
```

Parameters

pImageList

Pointer to the image list to assign. If *pImageList* is **NULL**, all images are removed from the tree view control.

nImageListType

Type of image list to set. The image list can be one of the following values:

- **TVSIL_NORMAL** Sets the normal image list, which contains the selected and nonselected images for the tree view item.
- **TVSIL_STATE** Sets the state image list, which contains the images for tree view items that are in a user-defined state.

Return Value

Pointer to the previous image list, if any; otherwise **NULL**.

Example

See the example for CTreeCtrl::GetImageList.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CImageList Class
CTreeCtrl::GetImageList

Other Resources

CTreeCtrl Members

CTreeCtrl::SetIndent

Call this function to set the width of indentation for a tree view control and redraw the control to reflect the new width.

```
void SetIndent(
   UINT nIndent
);
```

Parameters

nIndent

Width, in pixels, of the indentation. If *nIndent* is less than the system-defined minimum width, the new width is set to the system-defined minimum.

Example

See the example for CTreeCtrl::GetIndent.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetIndent
CTreeCtrl::GetItemRect

Other Resources

CTreeCtrl Members

CTreeCtrl::SetInsertMark

This member function implements the behavior of the Win32 message TVM_SETINSERTMARK, as described in the *Platform SDK*.

```
BOOL SetInsertMark(
   HTREEITEM hItem,
   BOOL fAfter = TRUE
);
```

Parameters

hItem

HTREEITEM that specifies at which item the insertion mark will be placed. If this argument is **NULL**, the insertion mark is removed.

fAfter

BOOL value that specifies if the insertion mark is placed before or after the specified item. If this argument is nonzero, the insertion mark will be placed after the item. If this argument is zero, the insertion mark will be placed before the item.

Return Value

Nonzero if successful; otherwise 0.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
// The point to test.
extern CPoint myPoint;

// Set the insert mark to be before the item at the point myPoint.
UINT uFlags;
HTREEITEM hItem = pmyTreeCtrl->HitTest(myPoint, &uFlags);

if ((hItem != NULL) && (TVHT_ONITEM & uFlags))
{
   pmyTreeCtrl->SetInsertMark(hItem, FALSE);
}
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

Other Resources

CTreeCtrl Members

CTreeCtrl::SetInsertMarkColor

This member function implements the behavior of the Win32 message TVM_SETINSERTMARKCOLOR, as described in the *Platform SDK*.

```
COLORREF SetInsertMarkColor(
   COLORREF clrNew
);
```

Parameters

clrNew

A **COLORREF** value that contains the new insertion mark color.

Return Value

A **COLORREF** value that contains the previous insertion mark color.

Example

See the example for CTreeCtrl::GetInsertMarkColor.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetInsertMarkColor

Other Resources

CTreeCtrl Members

CTreeCtrl::SetItem

Call this function to set the attributes of the specified tree view item.

```
BOOL SetItem(
   TVITEM* pItem
);
BOOL SetItem(
   HTREEITEM hItem,
   UINT nMask,
   LPCTSTR lpszItem,
   int nImage,
   int nSelectedImage,
   UINT nState,
   UINT nStateMask,
   LPARAM lParam
);
```

Parameters

pItem

A pointer to a TVITEM structure that contains the new item attributes, as described in the *Platform SDK*.

hItem

Handle of the item whose attributes are to be set. See the **hItem** member of the **TVITEM** structure in the *Platform SDK*.

nMask

Integer specifying which attributes to set. See the **mask** member of the **TVITEM** structure.

lpszItem

Address of a string containing the item's text.

nImage

Index of the item's image in the tree view control's image list. See the **iImage** member of the **TVITEM** structure.

nSelectedImage

Index of the item's selected image in the tree view control's image list. See the **iSelectedImage** member of the **TVITEM** structure.

nState

Specifies values for the item's states. See the **State** member of the **TVITEM** structure.

nStateMask

Specifies which states are to be set. See the **stateMask** member of the **TVITEM** structure.

lParam

A 32-bit application-specific value associated with the item.

Return Value

Nonzero if successful; otherwise 0.

Remarks

In the **TVITEM** structure, the **hItem** member identifies the item, and the **mask** member specifies which attributes to set.

If the **mask** member or the *nMask* parameter specifies the **TVIF_TEXT** value, the **pszText** member or the *lpszItem* is the address of a null-terminated string and the **cchTextMax** member is ignored. If **mask** (or *nMask*) specifies the **TVIF_STATE** value, the **stateMask** member or the *nStateMask* parameter specifies which item states to change and the **state** member or *nState* parameter contains the values for those states.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
// The point to hit test.
extern CPoint myPoint;

// Show the item at the point myPoint in bold.
UINT uFlags;
HTREEITEM hItem = pmyTreeCtrl->HitTest(myPoint, &uFlags);

if ((hItem != NULL) && (TVHT_ONITEM & uFlags))
{
   pmyTreeCtrl->SetItem(hItem, TVIF_STATE, NULL, 0, 0, TVIS_BOLD,
      TVIS_BOLD, 0);
}
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::GetItem

Other Resources

CTreeCtrl Members

CTreeCtrl::SetItemData

Call this function to set the 32-bit application-specific value associated with the specified item.

```
BOOL SetItemData(
   HTREEITEM hItem,
   DWORD_PTR dwData
);
```

Parameters

hItem

Handle of the item whose data is to be retrieved.

dwData

A 32-bit application-specific value associated with the item specified by *hItem*.

Return Value

Nonzero if successful; otherwise 0.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
CString   str;
HTREEITEM hItem;

// Insert 20 items into the tree control making every item's
// data be the handle of the item.
for (int i=0;i < 20;i++)
{
   str.Format(TEXT("item %d"), i);
   hItem = pmyTreeCtrl->InsertItem(str);

   if (hItem != NULL)
   {
      pmyTreeCtrl->SetItemData(hItem, (DWORD) hItem);
   }
}
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::GetItemData

Other Resources

CTreeCtrl Members

CTreeCtrl::SetItemHeight

This member function implements the behavior of the Win32 message TVM_SETITEMHEIGHT, as described in the *Platform SDK*.

```
SHORT SetItemHeight(
   SHORT cyHeight
);
```

Parameters

cyHeight

Specifies the new height of every item in the tree view, in pixels. If this argument is less than the height of the images, then it will be set to the height of the images. If this argument is not even, it will be rounded down to the nearest even value. If this argument is -1, the control will revert to using its default item height.

Return Value

The previous height of the items, in pixels.

Example

See the example for CTreeCtrl::GetItemHeight.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetItemHeight

Other Resources

CTreeCtrl Members

CTreeCtrl::SetItemImage

Associates images with an item.

```
BOOL SetItemImage(
   HTREEITEM hItem,
   int nImage,
   int nSelectedImage
);
```

Parameters

hItem

Handle of the item whose image is to be set.

nImage

Index of the item's image in the tree view control's image list.

nSelectedImage

Index of the item's selected image in the tree view control's image list.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Each item in a tree view control can have a pair of bitmapped images associated with it. The images appear on the left side of an item's label. One image is displayed when the item is selected, and the other is displayed when the item is not selected. For example, an item might display an open folder when it is selected and a closed folder when it is not selected.

Call this function to set the index of the item's image and its selected image within the tree view control's image list.

For more information on images, see CImageList.

Example

See the example for CTreeCtrl::GetItemImage.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetItemImage
CImageList Class

Other Resources

CTreeCtrl Members

CTreeCtrl::SetItemState

Sets the state of the item specified by *hItem*.

```
BOOL SetItemState(
   HTREEITEM hItem,
   UINT nState,
   UINT nStateMask
);
```

Parameters

hItem

Handle of the item whose state is to be set.

nState

Specifies new states for the item.

nStateMask

Specifies which states are to be changed.

Return Value

Nonzero if successful; otherwise 0.

Remarks

For information on states, see CTreeCtrl::GetItem.

Example

See the example for CTreeCtrl::GetItemState.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetItem
CTreeCtrl::GetItemState

Other Resources

CTreeCtrl Members

CTreeCtrl::SetItemText

Sets the text of the item specified by *hItem*.

```
BOOL SetItemText(
   HTREEITEM hItem,
   LPCTSTR lpszItem
);
```

Parameters

hItem

Handle of the item whose text is to be set.

lpszItem

Address of a string containing the new text for the item

Return Value

Nonzero if successful; otherwise 0.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;
// The point to hit test.
extern CPoint myPoint;

// Clear the text of the item at point myPoint.
UINT uFlags;
HTREEITEM hItem = pmyTreeCtrl->HitTest(myPoint, &uFlags);

if ((hItem != NULL) && (TVHT_ONITEM & uFlags))
{
   pmyTreeCtrl->SetItemText(hItem, NULL);
}
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::GetItemText

Other Resources

CTreeCtrl Members

CTreeCtrl::SetLineColor

Call this member function to set the current line color for the tree view control.

```
COLORREF SetLineColor(
   COLORREF clrNew = CLR_DEFAULT
);
```

Parameters

clrNew

The new line color.

Return Value

The previous line color.

Remarks

This member function implements the behavior of the win32 message TVM_SETLINECOLOR, as described in the *Platform SDK*.

Example

```
// Gain a pointer to our tree control
CTreeCtrl* pCtrl = (CTreeCtrl*) GetDlgItem(IDC_TREE1);
ASSERT(pCtrl != NULL);
COLORREF clrPrev = pCtrl->SetLineColor( RGB( 255, 0, 0 ) );
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::GetLineColor

Other Resources

CTreeCtrl Members

CTreeCtrl::SetScrollTime

Call this member function to set the maximum scroll time for the tree view control.

```
UINT SetScrollTime(
   UINT uScrollTime
);
```

Parameters

uScrollTime

The new maximum scroll time, in milliseconds. If this value is less than 100, it will be rounded up to 100.

Return Value

The previous maximum scroll time, in milliseconds.

Remarks

This member function implements the behavior of the win32 message TVM_SETSCROLLTIME, as described in the *Platform SDK*.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetScrollTime

Other Resources

CTreeCtrl Members

CTreeCtrl::SetTextColor

This member function implements the behavior of the Win32 message TVM_SETTEXTCOLOR, as described in the *Platform SDK*.

```
COLORREF SetTextColor(
   COLORREF clr
);
```

Parameters

clr

A **COLORREF** value that contains the new text color. If this argument is -1, the control will revert to using the system color for the text color.

Return Value

A **COLORREF** value that represents the previous text color. If this value is -1, the control was using the system color for the text color.

Example

```
// gain a pointer to the tree control
CTreeCtrl* pTree = (CTreeCtrl*) GetDlgItem(IDC_TREE1);
ASSERT(pTree != NULL);

// change text color to white and background to dark blue
pTree->SetTextColor(RGB(255, 255, 255));
ASSERT(pTree->GetTextColor() == RGB(255, 255, 255));
pTree->SetBkColor(RGB(0, 0, 128));
ASSERT(pTree->GetBkColor() == RGB(0, 0, 255));

// force repaint immediately
pTree->Invalidate();
```

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetTextColor

Other Resources

CTreeCtrl Members

CTreeCtrl::SetToolTips

This member function implements the behavior of the Win32 message TVM_SETTOOLTIPS, as described in the *Platform SDK*.

```
CToolTipCtrl* SetToolTips(
   CToolTipCtrl* pWndTip
);
```

Parameters

pWndTip

A pointer to a CToolTipCtrl object that the tree control will use.

Return Value

A pointer to a CToolTipCtrl object containing the tooltip previously used by the control, or **NULL** if no tooltips were used previously.

Remarks

To use tooltips, indicate the **TVS_NOTOOLTIPS** style when you create the **CTreeCtrl** object.

Example

See the example for CTreeCtrl::GetToolTips.

See Also

Reference

CTreeCtrl Class
Hierarchy Chart
CTreeCtrl::GetToolTips
CTreeCtrl::Create

Other Resources

CTreeCtrl Members

CTreeCtrl::SortChildren

Call this function to alphabetically sort the child items of the given parent item in a tree view control.

```
BOOL SortChildren(
   HTREEITEM hItem
);
```

Parameters

hItem

Handle of the parent item whose child items are to be sorted. If *hItem* is **NULL**, sorting will proceed from the root of the tree.

Return Value

Nonzero if successful; otherwise 0.

Remarks

SortChildren will not recurse through the tree; only the immediate children of *hItem* will be sorted.

Example

```
// The pointer to my tree control.
extern CTreeCtrl* pmyTreeCtrl;

// Sort all of the items in the tree control.
pmyTreeCtrl->SortChildren(TVI_ROOT);
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::SortChildrenCB

Other Resources

CTreeCtrl Members

CTreeCtrl::SortChildrenCB

Call this function to sort tree view items using an application-defined callback function that compares the items.

```
BOOL SortChildrenCB(
   LPTVSORTCB pSort
);
```

Parameters

pSort

Pointer to a TVSORTCB structure.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The structure's comparison function, **lpfnCompare**, must return a negative value if the first item should precede the second, a positive value if the first item should follow the second, or zero if the two items are equivalent.

The *lParam1* and *lParam2* parameters correspond to the **lParam** member of the TVITEM structure for the two items being compared. The *lParamSort* parameter corresponds to the **lParam** member of the **TV_SORTCB** structure.

Example

```
// Sort the item in reverse alphabetical order.
static int CALLBACK
MyCompareProc(LPARAM lParam1, LPARAM lParam2, LPARAM lParamSort)
{
   // lParamSort contains a pointer to the tree control.
   // The lParam of an item is just its handle,
// as specified with <A HREF="_mfc_ctreectrl.3a3a.setitemdata.htm">SetItemData</A>
   CTreeCtrl* pmyTreeCtrl = (CTreeCtrl*) lParamSort;
   CString    strItem1 = pmyTreeCtrl->GetItemText((HTREEITEM) lParam1);
   CString    strItem2 = pmyTreeCtrl->GetItemText((HTREEITEM) lParam2);

   return strcmp(strItem2, strItem1);
}

   // The pointer to my tree control.
   extern CTreeCtrl* pmyTreeCtrl;
   TVSORTCB tvs;

   // Sort the tree control's items using my
   // callback procedure.
   tvs.hParent = TVI_ROOT;
   tvs.lpfnCompare = MyCompareProc;
   tvs.lParam = (LPARAM) pmyTreeCtrl;

   pmyTreeCtrl->SortChildrenCB(&tvs);
```

See Also

Reference

CTreeCtrl Class

Hierarchy Chart

CTreeCtrl::SortChildren

Other Resources

CTreeCtrl Members

CTreeView Class

Simplifies use of the tree control and of CTreeCtrl, the class that encapsulates tree-control functionality, with MFC's document-view architecture.

```
class CTreeView : public CCtrlView
```

Remarks

For more information on this architecture, see the overview for the CView class and the cross-references cited there.

Requirements

Header: afxcview.h

See Also

Reference

CCtrlView Class
Hierarchy Chart
CView Class
CCtrlView Class
CTreeCtrl Class

Other Resources

CTreeView Members

CTreeView Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

CView Members

CCtrlView Members

Construction

CTreeView	Constructs a **CTreeView** object.

Attributes

GetTreeCtrl	Returns the tree control associated with the view.

See Also

Reference

CTreeView Class

Hierarchy Chart

CTreeView Member Functions

For information about the member functions in **CTreeView**, see CTreeView Members.

CTreeView::CTreeView

Constructs a **CTreeView** object.

```
CTreeView( );
```

See Also

Reference

CTreeView Class

Hierarchy Chart

CTreeCtrl Class

Other Resources

CTreeView Members

CTreeView::GetTreeCtrl

Returns a reference to the tree control associated with the view.

```
CTreeCtrl& GetTreeCtrl( ) const;
```

See Also

Reference

CTreeView Class

Hierarchy Chart

CTreeCtrl Class

Other Resources

CTreeView Members

CTypedPtrArray Class

Provides a type-safe "wrapper" for objects of class **CPtrArray** or **CObArray**.

```
template< class BASE_CLASS, class TYPE >
class CTypedPtrArray : public BASE_CLASS
```

Parameters

BASE_CLASS

Base class of the typed pointer array class; must be an array class (**CObArray** or **CPtrArray**).

TYPE

Type of the elements stored in the base-class array.

Remarks

When you use **CTypedPtrArray** rather than **CPtrArray** or **CObArray**, the C++ type-checking facility helps eliminate errors caused by mismatched pointer types.

In addition, the **CTypedPtrArray** wrapper performs much of the casting that would be required if you used **CObArray** or **CPtrArray**.

Because all **CTypedPtrArray** functions are inline, use of this template does not significantly affect the size or speed of your code.

For more information on using **CTypedPtrArray**, see the articles Collections and Template-Based Classes.

Requirements

Header: afxtempl.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

COLLECT Sample: Illustrates MFC Collection Classes

Reference

Hierarchy Chart

CPtrArray Class

CObArray Class

Other Resources

CTypedPtrArray Members

CTypedPtrArray Members

Element Access

Add	Adds a new element to the end of an array. Grows the array if necessary
Append	Adds the contents of one array to the end of another. Grows the array if necessary
Copy	Copies another array to the array; grows the array if necessary.
ElementAt	Returns a temporary reference to the element pointer within the array.
GetAt	Returns the value at a given index.
InsertAt	Inserts an element (or all the elements in another array) at a specified index.
SetAt	Sets the value for a given index; array not allowed to grow.
SetAtGrow	Sets the value for a given index; grows the array if necessary.

Operators

operator[]	Sets or gets the element at the specified index.

See Also

Reference

CTypedPtrArray Class
Hierarchy Chart

CTypedPtrArray Member Functions

For information about the member functions in **CTypedPtrArray**, see CTypedPtrArray Members.

CTypedPtrArray::Add

This member function calls *BASE_CLASS*::**Add**.

```
INT_PTR Add(
   TYPE newElement
);
```

Parameters

TYPE

Template parameter specifying the type of element to be added to the array.

newElement

The element to be added to this array.

Return Value

The index of the added element.

Remarks

For more detailed remarks, see CObArray::Add.

See Also

Reference

CTypedPtrArray Class

Hierarchy Chart

Other Resources

CTypedPtrArray Members

CTypedPtrArray::Append

This member function calls *BASE_CLASS*::**Append**.

```
INT_PTR Append(
   const CTypedPtrArray< BASE_CLASS, TYPE >& src
);
```

Parameters

BASE_CLASS

Base class of the typed pointer array class; must be an array class (CObArray or CPtrArray).

TYPE

Type of the elements stored in the base-class array.

src

Source of the elements to be appended to an array.

Return Value

The index of the first appended element.

Remarks

For more detailed remarks, see CObArray::Append.

See Also

Reference

CTypedPtrArray Class

Hierarchy Chart

Other Resources

CTypedPtrArray Members

CTypedPtrArray::Copy

This member function calls *BASE_CLASS*::**Copy**.

```
void Copy(
   const CTypedPtrArray<BASE_CLASS,
      TYPE>& src
);
```

Parameters

BASE_CLASS

Base class of the typed pointer array class; must be an array class (CObArray or CPtrArray).

TYPE

Type of the elements stored in the base-class array.

src

Source of the elements to be copied to an array.

Remarks

For more detailed remarks, see CObArray::Copy.

See Also

Reference

CTypedPtrArray Class

Hierarchy Chart

Other Resources

CTypedPtrArray Members

CTypedPtrArray::ElementAt

This inline function calls *BASE_CLASS*::**ElementAt**.

```
TYPE& ElementAt(
   INT_PTR nIndex
);
```

Parameters

TYPE

Template parameter specifying the type of elements stored in this array.

nIndex

An integer index that is greater than or equal to 0 and less than or equal to the value returned by *BASE_CLASS*::**GetUpperBound**.

Return Value

A temporary reference to the element at the location specified by *nIndex*. This element is of the type specified by the template parameter *TYPE*.

Remarks

For more detailed remarks, see CObArray::ElementAt.

See Also

Reference

CTypedPtrArray Class
Hierarchy Chart
CObArray::ElementAt
CObArray::GetUpperBound

Other Resources

CTypedPtrArray Members

CTypedPtrArray::GetAt

This inline function calls *BASE_CLASS*::**GetAt**.

```
TYPE GetAt(
   INT_PTR nIndex
) const;
```

Parameters

TYPE

Template parameter specifying the type of elements stored in the array.

nIndex

An integer index that is greater than or equal to 0 and less than or equal to the value returned by *BASE_CLASS*::**GetUpperBound**.

Return Value

A copy of the element at the location specified by *nIndex*. This element is of the type specified by the template parameter *TYPE*.

Remarks

For more detailed remarks, see CObArray::GetAt

See Also

Reference

CTypedPtrArray Class
Hierarchy Chart
CObArray::GetAt
CObArray::GetUpperBound

Other Resources

CTypedPtrArray Members

CTypedPtrArray::InsertAt

This member function calls *BASE_CLASS*::**InsertAt**.

```
void InsertAt(
   INT_PTR nIndex,
   TYPE newElement,
   INT_PTR nCount = 1
);
void InsertAt(
   INT_PTR nStartIndex,
   CTypedPtrArray< BASE_CLASS, TYPE >* pNewArray
);
```

Parameters

nIndex

An integer index that may be greater than the value returned by CObArray::GetUpperBound.

TYPE

Type of the elements stored in the base-class array.

newElement

The object pointer to be placed in this array. A *newElement* of value **NULL** is allowed.

nCount

The number of times this element should be inserted (defaults to 1).

nStartIndex

An integer index that may be greater than the value returned by **CObArray::GetUpperBound**.

BASE_CLASS

Base class of the typed pointer array class; must be an array class (CObArray or CPtrArray).

pNewArray

Another array that contains elements to be added to this array.

Remarks

For more detailed remarks, see CObArray::InsertAt.

See Also

Reference

CTypedPtrArray Class

Hierarchy Chart

Other Resources

CTypedPtrArray Members

CTypedPtrArray::SetAt

This member function calls *BASE_CLASS*::**SetAt**.

```
void SetAt(
   INT_PTR nIndex,
   TYPE ptr
);
```

Parameters

nIndex

An integer index that is greater than or equal to 0 and less than or equal to the value returned by CObArray::GetUpperBound.

TYPE

Type of the elements stored in the base-class array.

ptr

A pointer to the element to be inserted in the array at the nIndex. A NULL value is allowed.

Remarks

For more detailed remarks, see CObArray::SetAt.

See Also

Reference

CTypedPtrArray Class

Hierarchy Chart

Other Resources

CTypedPtrArray Members

CTypedPtrArray::SetAtGrow

This member function calls *BASE_CLASS*::**SetAtGrow**.

```
void SetAtGrow(
   INT_PTR nIndex,
   TYPE newElement
);
```

Parameters

nIndex

An integer index that is greater than or equal to 0.

TYPE

Type of the elements stored in the base-class array.

newElement

The object pointer to be added to this array. A **NULL** value is allowed.

Remarks

For more detailed remarks, see CObArray::SetAtGrow.

See Also

Reference

CTypedPtrArray Class

Hierarchy Chart

Other Resources

CTypedPtrArray Members

CTypedPtrArray Operators

For information about the operators in **CTypedPtrArray**, see CTypedPtrArray Members.

CTypedPtrArray::operator [ ]

These inline operators call *BASE_CLASS*::**operator [ ]**.

```
TYPE& operator[ ](
   INT_PTR nIndex
);
TYPE operator[ ](
   INT_PTR nIndex
) const;
```

Parameters

TYPE

Template parameter specifying the type of elements stored in the array.

nIndex

An integer index that is greater than or equal to 0 and less than or equal to the value returned by *BASE_CLASS*::**GetUpperBound**.

Remarks

The first operator, called for arrays that are not **const**, can be used on either the right (r-value) or the left (l-value) of an assignment statement. The second, invoked for **const** arrays, can be used only on the right.

The Debug version of the library asserts if the subscript (either on the left or right side of an assignment statement) is out of bounds.

See Also

Reference

CTypedPtrArray Class
Hierarchy Chart
CObArray::operator [ ]

Other Resources

CTypedPtrArray Members

CTypedPtrList Class

Provides a type-safe "wrapper" for objects of class **CPtrList**.

```
template< class BASE_CLASS, class TYPE >
class CTypedPtrList : public BASE_CLASS
```

Parameters

BASE_CLASS

Base class of the typed pointer list class; must be a pointer list class (**CObList** or **CPtrList**).

TYPE

Type of the elements stored in the base-class list.

Remarks

When you use **CTypedPtrList** rather than **CObList** or **CPtrList**, the C++ type-checking facility helps eliminate errors caused by mismatched pointer types.

In addition, the **CTypedPtrList** wrapper performs much of the casting that would be required if you used **CObList** or **CPtrList**.

Because all **CTypedPtrList** functions are inline, use of this template does not significantly affect the size or speed of your code.

Lists derived from **CObList** can be serialized, but those derived from **CPtrList** cannot.

When a **CTypedPtrList** object is deleted, or when its elements are removed, only the pointers are removed, not the entities they reference.

For more information on using **CTypedPtrList**, see the articles Collections and Template-Based Classes.

Example

This example creates an instance of **CTypedPtrList**, adds one object, serializes the list to disk, and then deletes the object:

```
typedef CTypedPtrList<CObList, CMyObject*>  CMyList;
CMyList ml;
CMyObject* pMyObject = new CMyObject();
ml.AddTail(pMyObject);

CFileException e;
CFile myFile;
myFile.Open("MyFile.txt", CFile::modeCreate|CFile::modeWrite, &e);
CArchive ar(&myFile, CArchive::store);
ml.Serialize(ar);

ar.Close();
myFile.Close();

while (!ml.IsEmpty())
{
   delete ml.GetHead();
   ml.RemoveHead();
}

//=====================
//where CMyObject is defined by the following files:

//CMyObject.h
class CMyObject : public CObject
{
public:
     int i;
     void Serialize(CArchive& ar);
     CMyObject() { i = 9876;}
```

```
protected:
     DECLARE_SERIAL(CMyObject)
};

//===================
//CMyObject.cpp
#include "stdafx.h"
#include "CMyObject.h"

IMPLEMENT_SERIAL(CMyObject, CObject, 0)

void CMyObject::Serialize(CArchive& ar)
{
    CObject::Serialize( ar );
    if( ar.IsStoring() )
         ar << i;
    else
         ar >> i;
}
```

Requirements

Header: afxtempl.h

See Also

Tasks

COLLECT Sample: Illustrates MFC Collection Classes

Reference

Hierarchy Chart

CPtrList Class

CObList Class

Other Resources

CTypedPtrList Members

CTypedPtrList Members

Head/Tail Access

GetHead	Returns the head element of the list (cannot be empty).
GetTail	Returns the tail element of the list (cannot be empty).

Operations

AddHead	Adds an element (or all the elements in another list) to the head of the list (makes a new head).
AddTail	Adds an element (or all the elements in another list) to the tail of the list (makes a new tail).
RemoveHead	Removes the element from the head of the list.
RemoveTail	Removes the element from the tail of the list.

Iteration

GetNext	Gets the next element for iterating.
GetPrev	Gets the previous element for iterating.

Retrieval/Modification

GetAt	Gets the element at a given position.
SetAt	Sets the element at a given position.

See Also

Reference

CTypedPtrList Class

Hierarchy Chart

CTypedPtrList Member Functions

For information about the member functions in **CTypedPtrList**, see CTypedPtrList Members.

CTypedPtrList::AddHead

This member function calls *BASE_CLASS*::**AddHead**.

```
POSITION AddHead(
      TYPE newElement
);
void AddHead(
   CTypedPtrList<BASE_CLASS,
      TYPE> *pNewList
);
```

Parameters

TYPE

Type of the elements stored in the base-class list.

newElement

The object pointer to be added to this list. A **NULL** value is allowed.

BASE_CLASS

Base class of the typed pointer list class; must be a pointer list class (CObList or CPtrList).

pNewList

A pointer to another CTypedPtrList object. The elements in *pNewList* will be added to this list.

Return Value

The first version returns the **POSITION** value of the newly inserted element.

Remarks

The first version adds a new element before the head of the list. The second version adds another list of elements before the head.

See Also

Reference

CTypedPtrList Class

Hierarchy Chart

Other Resources

CTypedPtrList Members

CTypedPtrList::AddTail

This member function calls *BASE_CLASS*::**AddTail**.

```
POSITION AddTail(
      TYPE newElement
);
void AddTail(
   CTypedPtrList<BASE_CLASS,
      TYPE> *pNewList
);
```

Parameters

TYPE

Type of the elements stored in the base-class list.

newElement

The object pointer to be added to this list. A **NULL** value is allowed.

BASE_CLASS

Base class of the typed pointer list class; must be a pointer list class (CObList or CPtrList).

pNewList

A pointer to another CTypedPtrList object. The elements in *pNewList* will be added to this list.

Return Value

The first version returns the **POSITION** value of the newly inserted element.

Remarks

The first version adds a new element after the tail of the list. The second version adds another list of elements after the tail of the list.

See Also

Reference

CTypedPtrList Class

Hierarchy Chart

Other Resources

CTypedPtrList Members

CTypedPtrList::GetAt

A variable of type **POSITION** is a key for the list.

```
TYPE& GetAt(
   POSITION position
);
TYPE GetAt(
   POSITION position
) const;
```

Parameters

TYPE

Template parameter specifying the type of elements stored in the list.

position

A **POSITION** value returned by a previous **GetHeadPosition** or **Find** member function call.

Return Value

If the list is accessed through a pointer to a **const CTypedPtrList**, then **GetAt** returns a pointer of the type specified by the template parameter *TYPE*. This allows the function to be used only on the right side of an assignment statement and thus protects the list from modification.

If the list is accessed directly or through a pointer to a **CTypedPtrList**, then **GetAt** returns a reference to a pointer of the type specified by the template parameter *TYPE*. This allows the function to be used on either side of an assignment statement and thus allows the list entries to be modified.

Remarks

It is not the same as an index, and you cannot operate on a **POSITION** value yourself. **GetAt** retrieves the **CObject** pointer associated with a given position.

You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

This inline function calls *BASE_CLASS***::GetAt**.

See Also

Reference

CTypedPtrList Class
Hierarchy Chart
CObList::GetAt

Other Resources

CTypedPtrList Members

CTypedPtrList::GetHead

Gets the pointer that represents the head element of this list.

```
TYPE& GetHead( );
TYPE GetHead( ) const;
```

Parameters

TYPE

Template parameter specifying the type of elements stored in the list.

Return Value

If the list is accessed through a pointer to a **const CTypedPtrList**, then **GetHead** returns a pointer of the type specified by the template parameter *TYPE*. This allows the function to be used only on the right side of an assignment statement and thus protects the list from modification.

If the list is accessed directly or through a pointer to a **CTypedPtrList**, then **GetHead** returns a reference to a pointer of the type specified by the template parameter *TYPE*. This allows the function to be used on either side of an assignment statement and thus allows the list entries to be modified.

Remarks

You must ensure that the list is not empty before calling **GetHead**. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use IsEmpty to verify that the list contains elements.

See Also

Reference

CTypedPtrList Class
Hierarchy Chart
CObList::IsEmpty
CTypedPtrList::GetTail
CTypedPtrList::GetNext
CTypedPtrList::GetPrev

Other Resources

CTypedPtrList Members

CTypedPtrList::GetNext

Gets the list element identified by *rPosition*, then sets *rPosition* to the **POSITION** value of the next entry in the list.

```
TYPE& GetNext(
   POSITION& rPosition
);
TYPE GetNext(
   POSITION& rPosition
) const;
```

Parameters

TYPE

Template parameter specifying the type of elements contained in this list.

rPosition

A reference to a **POSITION** value returned by a previous **GetNext**, **GetHeadPosition**, or other member function call.

Return Value

If the list is accessed through a pointer to a **const CTypedPtrList**, then **GetNext** returns a pointer of the type specified by the template parameter *TYPE*. This allows the function to be used only on the right side of an assignment statement and thus protects the list from modification.

If the list is accessed directly or through a pointer to a **CTypedPtrList**, then **GetNext** returns a reference to a pointer of the type specified by the template parameter *TYPE*. This allows the function to be used on either side of an assignment statement and thus allows the list entries to be modified.

Remarks

You can use **GetNext** in a forward iteration loop if you establish the initial position with a call to **GetHeadPosition** or CPtrList::Find.

You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

If the retrieved element is the last in the list, then the new value of *rPosition* is set to **NULL**.

It is possible to remove an element during an iteration. See the example for CObList::RemoveAt.

See Also

Reference

CTypedPtrList Class
Hierarchy Chart
CObList::Find
CObList::GetHeadPosition
CObList::GetTailPosition
CTypedPtrList::GetPrev
CTypedPtrList::GetHead
CTypedPtrList::GetTail

Other Resources

CTypedPtrList Members

CTypedPtrList::GetPrev

Gets the list element identified by *rPosition*, then sets *rPosition* to the **POSITION** value of the previous entry in the list.

```
TYPE& GetPrev(
   POSITION& rPosition
);
TYPE GetPrev(
   POSITION& rPosition
) const;
```

Parameters

TYPE

Template parameter specifying the type of elements contained in this list.

rPosition

A reference to a **POSITION** value returned by a previous **GetPrev** or other member function call.

Return Value

If the list is accessed through a pointer to a **const CTypedPtrList**, then **GetPrev** returns a pointer of the type specified by the template parameter *TYPE*. This allows the function to be used only on the right side of an assignment statement and thus protects the list from modification.

If the list is accessed directly or through a pointer to a **CTypedPtrList**, then **GetPrev** returns a reference to a pointer of the type specified by the template parameter *TYPE*. This allows the function to be used on either side of an assignment statement and thus allows the list entries to be modified.

Remarks

You can use **GetPrev** in a reverse iteration loop if you establish the initial position with a call to **GetTailPosition** or **Find**.

You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

If the retrieved element is the first in the list, then the new value of *rPosition* is set to **NULL**.

See Also

Reference

CTypedPtrList Class
Hierarchy Chart
CObList::Find
CObList::GetTailPosition
CObList::GetHeadPosition
CTypedPtrList::GetNext
CTypedPtrList::GetHead
CTypedPtrList::GetTail

Other Resources

CTypedPtrList Members

CTypedPtrList::GetTail

Gets the pointer that represents the head element of this list.

```
TYPE& GetTail( );
TYPE GetTail( ) const;
```

Parameters

TYPE

Template parameter specifying the type of elements stored in the list.

Return Value

If the list is accessed through a pointer to a **const CTypedPtrList**, then **GetTail** returns a pointer of the type specified by the template parameter *TYPE*. This allows the function to be used only on the right side of an assignment statement and thus protects the list from modification.

If the list is accessed directly or through a pointer to a **CTypedPtrList**, then **GetTail** returns a reference to a pointer of the type specified by the template parameter *TYPE*. This allows the function to be used on either side of an assignment statement and thus allows the list entries to be modified.

Remarks

You must ensure that the list is not empty before calling **GetTail**. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use IsEmpty to verify that the list contains elements.

See Also

Reference

CTypedPtrList Class
Hierarchy Chart
CObList::IsEmpty
CObList::Find
CObList::GetTailPosition
CObList::GetHeadPosition
CTypedPtrList::GetPrev
CTypedPtrList::GetNext
CTypedPtrList::GetHead

Other Resources

CTypedPtrList Members

CTypedPtrList::RemoveHead

Removes the element from the head of the list and returns it.

```
TYPE RemoveHead( );
```

Parameters

TYPE

Template parameter specifying the type of elements stored in the list.

Return Value

The pointer previously at the head of the list. This pointer is of the type specified by the template parameter *TYPE*.

Remarks

You must ensure that the list is not empty before calling **RemoveHead**. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use IsEmpty to verify that the list contains elements.

See Also

Reference

CTypedPtrList Class
Hierarchy Chart
CTypedPtrList::RemoveTail
CObList::IsEmpty
CObList::GetHead
CObList::AddHead

Other Resources

CTypedPtrList Members

CTypedPtrList::RemoveTail

Removes the element from the tail of the list and returns it.

```
TYPE RemoveTail( );
```

Parameters

TYPE

Template parameter specifying the type of elements stored in the list.

Return Value

The pointer previously at the tail of the list. This pointer is of the type specified by the template parameter *TYPE*.

Remarks

You must ensure that the list is not empty before calling **RemoveTail**. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use IsEmpty to verify that the list contains elements.

See Also

Reference

CTypedPtrList Class
Hierarchy Chart
CTypedPtrList::RemoveHead
CObList::IsEmpty
CObList::GetTail
CObList::AddTail

Other Resources

CTypedPtrList Members

CTypedPtrList::SetAt

This member function calls *BASE_CLASS*::**SetAt**.

```
void SetAt(
   POSITION pos,
      TYPE newElement
);
```

Parameters

pos

The **POSITION** of the element to be set.

TYPE

Type of the elements stored in the base-class list.

newElement

The object pointer to be written to the list.

Remarks

A variable of type **POSITION** is a key for the list. It is not the same as an index, and you cannot operate on a **POSITION** value yourself. **SetAt** writes the object pointer to the specified position in the list.

You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

For more detailed remarks, see CObList::SetAt.

See Also

Reference

CTypedPtrArray Class

Hierarchy Chart

Other Resources

CTypedPtrArray Members

CTypedPtrMap Class

Provides a type-safe "wrapper" for objects of the pointer-map classes **CMapPtrToPtr**, **CMapPtrToWord**, **CMapWordToPtr**, and **CMapStringToPtr**.

```
template< class BASE_CLASS, class KEY, class VALUE >
class CTypedPtrMap : public BASE_CLASS
```

Parameters

BASE_CLASS

Base class of the typed pointer map class; must be a pointer map class (**CMapPtrToPtr**, **CMapPtrToWord**, **CMapWordToPtr**, or **CMapStringToPtr**).

KEY

Class of the object used as the key to the map.

VALUE

Class of the object stored in the map.

Remarks

When you use **CTypedPtrMap**, the C++ type-checking facility helps eliminate errors caused by mismatched pointer types.

Because all **CTypedPtrMap** functions are inline, use of this template does not significantly affect the size or speed of your code.

For more information on using **CTypedPtrMap**, see the articles Collections and Template-Based Classes.

Requirements

Header: afxtempl.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

COLLECT Sample: Illustrates MFC Collection Classes

Reference

Hierarchy Chart
CMapPtrToPtr Class
CMapPtrToWord Class
CMapWordToPtr Class
CMapStringToPtr Class

Other Resources

CTypedPtrMap Members

CTypedPtrMap Members

Element Access

GetNextAssoc	Gets the next element for iterating.
Lookup	Returns a *KEY* based on a *VALUE*.
RemoveKey	Removes an element specified by a key.
SetAt	Inserts an element into the map; replaces an existing element if a matching key is found.

Operators

operator[]	Inserts an element into the map.

See Also

Reference

CTypedPtrMap Class

Hierarchy Chart

CTypedPtrMap Member Functions

For information about the member functions in **CTypedPtrMap**, see CTypedPtrMap Members.

CTypedPtrMap::GetNextAssoc

Retrieves the map element at *rNextPosition*, then updates *rNextPosition* to refer to the next element in the map.

```
void GetNextAssoc(
   POSITION& rPosition,
      KEY& rKey,
      VALUE& rValue
) const;
```

Parameters

rPosition

Specifies a reference to a **POSITION** value returned by a previous **GetNextAssoc** or *BASE_CLASS*::**GetStartPosition** call.

KEY

Template parameter specifying the type of the map's keys.

rKey

Specifies the returned key of the retrieved element.

VALUE

Template parameter specifying the type of the map's values.

rValue

Specifies the returned value of the retrieved element.

Remarks

This function is most useful for iterating through all the elements in the map. Note that the position sequence is not necessarily the same as the key value sequence.

If the retrieved element is the last in the map, then the new value of *rNextPosition* is set to **NULL**.

This inline function calls *BASE_CLASS*::**GetNextAssoc**.

See Also

Reference

CTypedPtrMap Class
Hierarchy Chart
CMapStringToOb::GetNextAssoc
CMapStringToOb::GetStartPosition

Other Resources

CTypedPtrMap Members

CTypedPtrMap::Lookup

Lookup uses a hashing algorithm to quickly find the map element with a key that matches exactly.

```
BOOL Lookup(
      BASE_CLASS::BASE_ARG_KEY key,
      VALUE& rValue
) const;
```

Parameters

BASE_CLASS

Template parameter specifying the base class of this map's class.

key

The key of the element to be looked up.

VALUE

Template parameter specifying the type of values stored in this map.

rValue

Specifies the returned value of the retrieved element.

Return Value

Nonzero if the element was found; otherwise 0.

Remarks

This inline function calls *BASE_CLASS*::**Lookup**.

See Also

Reference

CTypedPtrMap Class
Hierarchy Chart
CMapStringToOb::Lookup

Other Resources

CTypedPtrMap Members

CTypedPtrMap::RemoveKey

This member function calls *BASE_CLASS*::**RemoveKey**.

```
BOOL RemoveKey(
      KEY key
);
```

Parameters

KEY

Template parameter specifying the type of the map's keys.

key

Key for the element to be removed.

Return Value

Nonzero if the entry was found and successfully removed; otherwise 0.

Remarks

For more detailed remarks, see CMapStringToOb::RemoveKey.

See Also

Reference

CTypedPtrMap Class

Hierarchy Chart

Other Resources

CTypedPtrMap Members

CTypedPtrMap::SetAt

This member function calls *BASE_CLASS*::**SetAt**.

```
void SetAt(
      KEY key,
      VALUE newValue
);
```

Parameters

KEY

Template parameter specifying the type of the map's keys.

key

Specifies the key value of the newValue.

newValue

Specifies the object pointer that is the value of the new element.

Remarks

For more detailed remarks, see CMapStringToOb::SetAt.

See Also

Reference

CTypedPtrMap Class

Hierarchy Chart

Other Resources

CTypedPtrMap Members

CTypedPtrMap Operators

For information about the operators in **CTypedPtrMap**, see CTypedPtrMap Members.

CTypedPtrMap::operator [ ]

This operator can be used only on the left side of an assignment statement (an l-value).

```
VALUE& operator [ ](
      BASE_CLASS::BASE_ARG_KEY key
);
```

Parameters

VALUE

Template parameter specifying the type of values stored in this map.

BASE_CLASS

Template parameter specifying the base class of this map's class.

key

The key of the element to be looked up or created in the map.

Remarks

If there is no map element with the specified key, then a new element is created. There is no "right side" (r-value) equivalent to this operator because there is a possibility that a key may not be found in the map. Use the **Lookup** member function for element retrieval.

See Also

Reference

CTypedPtrMap Class
Hierarchy Chart
CTypedPtrMap::Lookup

Other Resources

CTypedPtrMap Members

CUIntArray Class

Supports arrays of unsigned integers.

```
class CUIntArray : public CObject
```

Remarks

An unsigned integer, or **UINT**, differs from words and doublewords in that the physical size of a **UINT** can change depending on the target operating environment. Under Windows version 3.1, a **UINT** is the same size as a **WORD**. Under Windows NT and Windows 95/98, a **UINT** is the same size as a doubleword.

The member functions of **CUIntArray** are similar to the member functions of class CObArray. Because of this similarity, you can use the **CObArray** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a function parameter or return value, substitute a **UINT**.

```
CObject* CObArray::GetAt( int <nIndex> ) const;
```

for example, translates to

```
UINT CUIntArray::GetAt( int <nIndex> ) const;
```

CUIntArray incorporates the IMPLEMENT_DYNAMIC macro to support run-time type access and dumping to a CDumpContext object. If you need a dump of individual unsigned integer elements, you must set the depth of the dump context to 1 or greater. Unsigned integer arrays cannot be serialized.

Note
Before using an array, use **SetSize** to establish its size and allocate memory for it. If you do not use **SetSize**, adding elements to your array causes it to be frequently reallocated and copied. Frequent reallocation and copying are inefficient and can fragment memory.

For more information on using **CUIntArray**, see the article Collections.

Requirements

Header: afxcoll.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CObject Class

Hierarchy Chart

Other Resources

CUIntArray Members

CUIntArray Members

Construction

CUIntArray	Constructs an empty array for unsigned integers.

Bounds

GetCount	Gets the number of elements in this array.
GetSize	Gets the number of elements in this array.
GetUpperBound	Returns the largest valid index.
SetSize	Sets the number of elements to be contained in this array.

Operations

FreeExtra	Frees all unused memory above the current upper bound.
IsEmpty	Determines if the array contains no elements.
RemoveAll	Removes all the elements from this array.

Element Access

ElementAt	Returns a temporary reference to the element pointer within the array.
GetAt	Returns the value at a given index.
GetData	Allows access to elements in the array. Can be **NULL**.
SetAt	Sets the value for a given index; the array is not allowed to grow.

Growing the Array

Add	Adds an element to the end of the array; grows the array if necessary.
Append	Appends another array to the array; grows the array if necessary.
Copy	Copies another array to the array; grows the array if necessary.
SetAtGrow	Sets the value for a given index; grows the array if necessary.

Insertion/Removal

InsertAt	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt	Removes an element at a specific index.

Operators

operator [ ]	Sets or gets the element at the specified index.

See Also

Reference

CUIntArray Class

Hierarchy Chart

Other Resources

CObject Members

CUserException Class

Thrown to stop an end-user operation.

```
class CUserException : public CSimpleException
```

Remarks

Use **CUserException** when you want to use the throw/catch exception mechanism for application-specific exceptions. "User" in the class name can be interpreted as "my user did something exceptional that I need to handle."

A **CUserException** is usually thrown after calling the global function **AfxMessageBox** to notify the user that an operation has failed. When you write an exception handler, handle the exception specially since the user usually has already been notified of the failure. The framework throws this exception in some cases. To throw a **CUserException** yourself, alert the user and then call the global function **AfxThrowUserException**.

In the example below, a function containing operations that may fail alerts the user and throws a **CUserException**. The calling function catches the exception and handles it specially:

```
void DoSomeOperation( )
{
    // Processing
    // If something goes wrong...
    AfxMessageBox( "The x operation failed" );
    AfxThrowUserException( );
}

BOOL TrySomething( )
{
    TRY
    {
        // Could throw a CUserException or other exception.
        DoSomeOperation( );
    }
    CATCH( CUserException, e )
    {
        return FALSE;    // User already notified.
    }
    AND_CATCH( CException, e )
    {
        // For other exception types, notify user here.
        AfxMessageBox( "Some operation failed" );
        return FALSE;
    }
    END_CATCH
    return TRUE;   // No exception thrown.
}
```

For more information on using **CUserException**, see the article Exception Handling (MFC).

Requirements

Header: afxwin.h

See Also

Reference

Hierarchy Chart
CException Class
AfxMessageBox
AfxThrowUserException

Other Resources

CException Members

CView Class

Provides the basic functionality for user-defined view classes.

```
class AFX_NOVTABLE CView : public CWnd
```

Remarks

A view is attached to a document and acts as an intermediary between the document and the user: the view renders an image of the document on the screen or printer and interprets user input as operations upon the document.

A view is a child of a frame window. More than one view can share a frame window, as in the case of a splitter window. The relationship between a view class, a frame window class, and a document class is established by a **CDocTemplate** object. When the user opens a new window or splits an existing one, the framework constructs a new view and attaches it to the document.

A view can be attached to only one document, but a document can have multiple views attached to it at once — for example, if the document is displayed in a splitter window or in multiple child windows in a multiple document interface (MDI) application. Your application can support different types of views for a given document type; for example, a word-processing program might provide both a complete text view of a document and an outline view that shows only the section headings. These different types of views can be placed in separate frame windows or in separate panes of a single frame window if you use a splitter window.

A view may be responsible for handling several different types of input, such as keyboard input, mouse input or input via drag-and-drop, as well as commands from menus, toolbars, or scroll bars. A view receives commands forwarded by its frame window. If the view does not handle a given command, it forwards the command to its associated document. Like all command targets, a view handles messages via a message map.

The view is responsible for displaying and modifying the document's data but not for storing it. The document provides the view with the necessary details about its data. You can let the view access the document's data members directly, or you can provide member functions in the document class for the view class to call.

When a document's data changes, the view responsible for the changes typically calls the CDocument::UpdateAllViews function for the document, which notifies all the other views by calling the **OnUpdate** member function for each. The default implementation of **OnUpdate** invalidates the view's entire client area. You can override it to invalidate only those regions of the client area that map to the modified portions of the document.

To use **CView**, derive a class from it and implement the **OnDraw** member function to perform screen display. You can also use **OnDraw** to perform printing and print preview. The framework handles the print loop for printing and previewing your document.

A view handles scroll-bar messages with the CWnd::OnHScroll and CWnd::OnVScroll member functions. You can implement scroll-bar message handling in these functions, or you can use the **CView** derived class CScrollView to handle scrolling for you.

Besides **CScrollView**, the Microsoft Foundation Class Library provides nine other classes derived from **CView**:

- CCtrlView, a view that allows usage of document - view architecture with tree, list, and rich edit controls.
- CDaoRecordView, a view that displays database records in dialog-box controls.
- CEditView, a view that provides a simple multiline text editor. You can use a **CEditView** object as a control in a dialog box as well as a view on a document.
- CFormView, a scrollable view that contains dialog-box controls and is based on a dialog template resource.
- CListView, a view that allows usage of document - view architecture with list controls.
- CRecordView, a view that displays database records in dialog-box controls.
- CRichEditView, a view that allows usage of document - view architecture with rich edit controls.
- CScrollView, a view that automatically provides scrolling support.
- CTreeView, a view that allows usage of document - view architecture with tree controls.

The **CView** class also has a derived implementation class named **CPreviewView**, which is used by the framework to perform print previewing. This class provides support for the features unique to the print-preview window, such as a toolbar, single- or double-page preview, and zooming, that is, enlarging the previewed image. You don't need to call or override any of **CPreviewView**'s member functions unless you want to implement your own interface for print preview (for example, if you want to support editing in print preview mode). For more information on using **CView**, see Document/View Architecture and Printing. In addition, see Technical Note 30 for more details on customizing print preview.

Requirements

Header: afxwin.h

Smart Device Developer Notes

The Smart Device version of CView Class does not support the following print methods:

- CView::OnPrepareDC
- CView::OnPreparePrinting
- CView::OnBeginPrinting
- CView::OnPrint
- CView::OnEndPrinting

See Also

Tasks

MDIDOCVW Sample: Demonstrates MDI Using Doc/View Architecture

Reference

CWnd Class
Hierarchy Chart
CWnd Class
CFrameWnd Class
CSplitterWnd Class
CDC Class
CDocTemplate Class
CDocument Class

Concepts

CView Members

CView Members

Base Class Members

CObject Members

CCmdTarget Members

CWnd Members

Operations

DoPreparePrinting	Displays Print dialog box and creates printer device context; call when overriding the **OnPreparePrinting** member function.
GetDocument	Returns the document associated with the view.

OLE Overridables

OnDragEnter	Called when an item is first dragged into the drag-and-drop region of a view.
OnDragLeave	Called when a dragged item leaves the drag-and-drop region of a view.
OnDragOver	Called when an item is dragged over the drag-and-drop region of a view.
OnDragScroll	Called to determine whether the cursor is dragged into the scroll region of the window.
OnDrop	Called when an item has been dropped into the drag-and-drop region of a view, default handler.
OnDropEx	Called when an item has been dropped into the drag-and-drop region of a view, primary handler.
OnInitialUpdate	Called after a view is first attached to a document.
OnScroll	Called when OLE items are dragged beyond the borders of the view.
OnScrollBy	Called when a view containing active in-place OLE items is scrolled.

Overridables

IsSelected	Tests whether a document item is selected. Required for OLE support.
OnActivateFrame	Called when the frame window containing the view is activated or deactivated.
OnActivateView	Called when a view is activated.
OnBeginPrinting	Called when a print job begins; override to allocate graphics device interface (GDI) resources.
OnDraw	Called to render an image of the document for screen display, printing, or print preview. Implementation required.
OnEndPrinting	Called when a print job ends; override to deallocate GDI resources.
OnEndPrintPreview	Called when preview mode is exited.
OnPrepareDC	Called before the **OnDraw** member function is called for screen display or the **OnPrint** member function is called for printing or print preview.
OnPreparePrinting	Called before a document is printed or previewed; override to initialize Print dialog box.

OnPrint	Called to print or preview a page of the document.
OnUpdate	Called to notify a view that its document has been modified.

Constructors

CView	Constructs a **CView** object.

Smart Device Developer Notes

The Smart Device version of CView Class does not support the following print methods:

- CView::OnPrepareDC
- CView::OnPreparePrinting
- CView::OnBeginPrinting
- CView::OnPrint
- CView::OnEndPrinting

See Also

Reference

CView Class
Hierarchy Chart

CView Member Functions

For information about the member functions in **CView**, see CView Members.

CView::CView

Constructs a **CView** object.

```
CView( );
```

Remarks

The framework calls the constructor when a new frame window is created or a window is split. Override the OnInitialUpdate member function to initialize the view after the document is attached.

See Also

Reference

CView Class
Hierarchy Chart
CView::OnInitialUpdate

Concepts

CView Members

CView::DoPreparePrinting

Call this function from your override of OnPreparePrinting to invoke the Print dialog box and create a printer device context.

```
BOOL DoPreparePrinting(
   CPrintInfo* pInfo
);
```

Parameters

pInfo

Points to a CPrintInfo structure that describes the current print job.

Return Value

Nonzero if printing or print preview can begin; 0 if the operation has been canceled.

Remarks

This function's behavior depends on whether it is being called for printing or print preview (specified by the **m_bPreview** member of the *pInfo* parameter). If a file is being printed, this function invokes the Print dialog box, using the values in the CPrintInfo structure that *pInfo* points to; after the user has closed the dialog box, the function creates a printer device context based on settings the user specified in the dialog box and returns this device context through the *pInfo* parameter. This device context is used to print the document.

If a file is being previewed, this function creates a printer device context using the current printer settings; this device context is used for simulating the printer during preview.

See Also

Reference

CView Class
Hierarchy Chart
CPrintInfo Structure
CView::OnPreparePrinting

Concepts

CView Members

CView::GetDocument

Call this function to get a pointer to the view's document.

```
CDocument* GetDocument( ) const;
```

Return Value

A pointer to the CDocument object associated with the view. **NULL** if the view is not attached to a document.

Remarks

This allows you to call the document's member functions.

See Also

Reference

CView Class

Hierarchy Chart

CDocument Class

Concepts

CView Members

CView::IsSelected

Called by the framework to check whether the specified document item is selected.

```
virtual BOOL IsSelected(
   const CObject* pDocItem
) const;
```

Parameters

pDocItem

Points to the document item being tested.

Return Value

Nonzero if the specified document item is selected; otherwise 0.

Remarks

The default implementation of this function returns **FALSE**. Override this function if you are implementing selection using CDocItem objects. You must override this function if your view contains OLE items.

See Also

Reference

CView Class
Hierarchy Chart
CDocItem Class
COleClientItem Class

Concepts

CView Members

CView::OnActivateFrame

Called by the framework when the frame window containing the view is activated or deactivated.

```
virtual void OnActivateFrame(
   UINT nState,
   CFrameWnd* pFrameWnd
);
```

Parameters

nState

Specifies whether the frame window is being activated or deactivated. It can be one of the following values:

- **WA_INACTIVE** The frame window is being deactivated.
- **WA_ACTIVE** The frame window is being activated through some method other than a mouse click (for example, by use of the keyboard interface to select the window).
- **WA_CLICKACTIVE** The frame window is being activated by a mouse click

pFrameWnd

Pointer to the frame window that is to be activated.

Remarks

Override this member function if you want to perform special processing when the frame window associated with the view is activated or deactivated. For example, CFormView performs this override when it saves and restores the control that has focus.

See Also

Reference

CView Class
Hierarchy Chart
CWnd::OnActivate
CFormView Class

Concepts

CView Members

CView::OnActivateView

Called by the framework when a view is activated or deactivated.

```
virtual void OnActivateView(
   BOOL bActivate,
   CView* pActivateView,
   CView* pDeactiveView
);
```

Parameters

bActivate

Indicates whether the view is being activated or deactivated.

pActivateView

Points to the view object that is being activated.

pDeactiveView

Points to the view object that is being deactivated.

Remarks

The default implementation of this function sets the focus to the view being activated. Override this function if you want to perform special processing when a view is activated or deactivated. For example, if you want to provide special visual cues that distinguish the active view from the inactive views, you would examine the *bActivate* parameter and update the view's appearance accordingly.

The *pActivateView* and *pDeactiveView* parameters point to the same view if the application's main frame window is activated with no change in the active view — for example, if the focus is being transferred from another application to this one, rather than from one view to another within the application or when switching amongst MDI child windows. This allows a view to re-realize its palette, if needed.

These parameters differ when CFrameWnd::SetActiveView is called with a view that is different from what CFrameWnd::GetActiveView would return. This happens most often with splitter windows.

See Also

Reference

CView Class
Hierarchy Chart
CWnd::OnActivate
CFrameWnd::SetActiveView
CFrameWnd::GetActiveView

Concepts

CView Members

CView::OnBeginPrinting

Called by the framework at the beginning of a print or print preview job, after **OnPreparePrinting** has been called.

```
virtual void OnBeginPrinting(
   CDC* pDC,
   CPrintInfo* pInfo
);
```

Parameters

pDC

Points to the printer device context.

pInfo

Points to a CPrintInfo structure that describes the current print job.

Remarks

The default implementation of this function does nothing. Override this function to allocate any GDI resources, such as pens or fonts, needed specifically for printing. Select the GDI objects into the device context from within the OnPrint member function for each page that uses them. If you are using the same view object to perform both screen display and printing, use separate variables for the GDI resources needed for each display; this allows you to update the screen during printing.

You can also use this function to perform initializations that depend on properties of the printer device context. For example, the number of pages needed to print the document may depend on settings that the user specified from the Print dialog box (such as page length). In such a situation, you cannot specify the document length in the OnPreparePrinting member function, where you would normally do so; you must wait until the printer device context has been created based on the dialog box settings. OnBeginPrinting is the first overridable function that gives you access to the CDC object representing the printer device context, so you can set the document length from this function. Note that if the document length is not specified by this time, a scroll bar is not displayed during print preview.

Smart Device Developer Notes

This method is not supported in smart device projects.

See Also

Reference

CView Class
Hierarchy Chart
CView::OnEndPrinting
CView::OnPreparePrinting
CView::OnPrint

Concepts

CView Members

CView::OnDragEnter

Called by the framework when the mouse first enters the non-scrolling region of the drop target window.

```
virtual DROPEFFECT OnDragEnter(
   COleDataObject* pDataObject,
   DWORD dwKeyState,
   CPoint point
);
```

Parameters

pDataObject

Points to the COleDataObject being dragged into the drop area of the view.

dwKeyState

Contains the state of the modifier keys. This is a combination of any number of the following: **MK_CONTROL**, **MK_SHIFT**, **MK_ALT**, **MK_LBUTTON**, **MK_MBUTTON**, and **MK_RBUTTON**.

point

The current mouse position relative to the client area of the view.

Return Value

A value from the **DROPEFFECT** enumerated type, which indicates the type of drop that would occur if the user dropped the object at this position. The type of drop usually depends on the current key state indicated by *dwKeyState*. A standard mapping of keystates to **DROPEFFECT** values is:

- **DROPEFFECT_NONE** The data object cannot be dropped in this window.
- **DROPEFFECT_LINK** for **MK_CONTROL | MK_SHIFT** Creates a linkage between the object and its server.
- **DROPEFFECT_COPY** for **MK_CONTROL** Creates a copy of the dropped object.
- **DROPEFFECT_MOVE** for **MK_ALT** Creates a copy of the dropped object and delete the original object. This is typically the default drop effect, when the view can accept this data object.

For more information, see the MFC Advanced Concepts sample OCLIENT.

Remarks

Default implementation is to do nothing and return **DROPEFFECT_NONE**.

Override this function to prepare for future calls to the OnDragOver member function. Any data required from the data object should be retrieved at this time for later use in the **OnDragOver** member function. The view should also be updated at this time to give the user visual feedback. For more information, see the article Drag and Drop: Implementing a Drop Target.

See Also

Reference

CView Class
Hierarchy Chart
CView::OnDragOver
CView::OnDrop
CView::OnDropEx
CView::OnDragLeave
COleDropTarget::OnDragEnter

Concepts

CView Members

CView::OnDragLeave

Called by the framework during a drag operation when the mouse is moved out of the valid drop area for that window.

```
virtual void OnDragLeave( );
```

Remarks

Override this function if the current view needs to clean up any actions taken during OnDragEnter or OnDragOver calls, such as removing any visual user feedback while the object was dragged and dropped.

See Also

Reference

CView Class
Hierarchy Chart
CView::OnDragEnter
CView::OnDragOver
CView::OnScroll
COleDropTarget::OnDragLeave

Concepts

CView Members

CView::OnDragOver

Called by the framework during a drag operation when the mouse is moved over the drop target window.

```
virtual DROPEFFECT OnDragOver(
   COleDataObject* pDataObject,
   DWORD dwKeyState,
   CPoint point
);
```

Parameters

pDataObject

Points to the COleDataObject being dragged over the drop target.

dwKeyState

Contains the state of the modifier keys. This is a combination of any number of the following: **MK_CONTROL**, **MK_SHIFT**, **MK_ALT**, **MK_LBUTTON**, **MK_MBUTTON**, and **MK_RBUTTON**.

point

The current mouse position relative to the view client area.

Return Value

A value from the **DROPEFFECT** enumerated type, which indicates the type of drop that would occur if the user dropped the object at this position. The type of drop often depends on the current key state as indicated by *dwKeyState*. A standard mapping of keystates to **DROPEFFECT** values is:

- **DROPEFFECT_NONE** The data object cannot be dropped in this window.
- **DROPEFFECT_LINK** for **MK_CONTROL | MK_SHIFT** Creates a linkage between the object and its server.
- **DROPEFFECT_COPY** for **MK_CONTROL** Creates a copy of the dropped object.
- **DROPEFFECT_MOVE** for **MK_ALT** Creates a copy of the dropped object and delete the original object. This is typically the default drop effect, when the view can accept the data object.

For more information, see the MFC Advanced Concepts sample OCLIENT.

Remarks

The default implementation is to do nothing and return **DROPEFFECT_NONE**.

Override this function to give the user visual feedback during the drag operation. Since this function is called continuously, any code contained within it should be optimized as much as possible. For more information, see the article Drag and Drop: Implementing a Drop Target.

See Also

Reference

CView Class
Hierarchy Chart
CView::OnDragEnter
CView::OnDrop
CView::OnDropEx
CView::OnDragLeave
COleDropTarget::OnDragOver

Concepts

CView Members

CView::OnDragScroll

Called by the framework before calling OnDragEnter or OnDragOver to determine whether the point is in the scrolling region.

```
virtual DROPEFFECT OnDragScroll(
   DWORD dwKeyState,
   CPoint point
);
```

Parameters

dwKeyState

Contains the state of the modifier keys. This is a combination of any number of the following: **MK_CONTROL**, **MK_SHIFT**, **MK_ALT**, **MK_LBUTTON**, **MK_MBUTTON**, and **MK_RBUTTON**.

point

Contains the location of the cursor, in pixels, relative to the screen.

Return Value

A value from the **DROPEFFECT** enumerated type, which indicates the type of drop that would occur if the user dropped the object at this position. The type of drop usually depends on the current key state indicated by *dwKeyState*. A standard mapping of keystates to **DROPEFFECT** values is:

- **DROPEFFECT_NONE** The data object cannot be dropped in this window.
- **DROPEFFECT_LINK** for **MK_CONTROL | MK_SHIFT** Creates a linkage between the object and its server.
- **DROPEFFECT_COPY** for **MK_CONTROL** Creates a copy of the dropped object.
- **DROPEFFECT_MOVE** for **MK_ALT** Creates a copy of the dropped object and delete the original object.
- **DROPEFFECT_SCROLL** Indicates that a drag scroll operation is about to occur or is occurring in the target view.

For more information, see the MFC Advanced Concepts sample OCLIENT.

Remarks

Override this function when you want to provide special behavior for this event. The default implementation automatically scrolls windows when the cursor is dragged into the default scroll region inside the border of each window.For more information, see the article Drag and Drop: Implementing a Drop Target.

See Also

Reference

CView Class
Hierarchy Chart
CView::OnDragEnter
CView::OnDragOver
CView::OnDrop
CView::OnDragLeave
COleDropTarget::OnDragScroll

Concepts

CView Members

CView::OnDraw

Called by the framework to render an image of the document.

```
virtual void OnDraw(
   CDC* pDC
) = 0;
```

Parameters

pDC

Points to the device context to be used for rendering an image of the document.

Remarks

The framework calls this function to perform screen display, printing, and print preview, and it passes a different device context in each case. There is no default implementation.

You must override this function to display your view of the document. You can make graphic device interface (GDI) calls using the CDC object pointed to by the *pDC* parameter. You can select GDI resources, such as pens or fonts, into the device context before drawing and then deselect them afterwards. Often your drawing code can be device-independent; that is, it doesn't require information about what type of device is displaying the image.

To optimize drawing, call the RectVisible member function of the device context to find out whether a given rectangle will be drawn. If you need to distinguish between normal screen display and printing, call the IsPrinting member function of the device context.

See Also

Reference

CView Class
Hierarchy Chart
CDC::IsPrinting
CDC::RectVisible
CView::OnPrint
CWnd::OnCreate
CWnd::OnDestroy
CWnd::PostNcDestroy

Concepts

CView Members

CView::OnDrop

Called by the framework when the user releases a data object over a valid drop target.

```
virtual BOOL OnDrop(
   COleDataObject* pDataObject,
   DROPEFFECT dropEffect,
   CPoint point
);
```

Parameters

pDataObject

Points to the COleDataObject that is dropped into the drop target.

dropEffect

The drop effect that the user has requested.

- **DROPEFFECT_COPY** Creates a copy of the data object being dropped.
- **DROPEFFECT_MOVE** Moves the data object to the current mouse location.
- **DROPEFFECT_LINK** Creates a link between a data object and its server.

point

The current mouse position relative to the view client area.

Return Value

Nonzero if the drop was successful; otherwise 0.

Remarks

The default implementation does nothing and returns **FALSE**.

Override this function to implement the effect of an OLE drop into the client area of the view. The data object can be examined via *pDataObject* for Clipboard data formats and data dropped at the specified point.

Note
The framework does not call this function if there is an override to OnDropEx in this view class.

See Also

Reference

CView Class
Hierarchy Chart
CView::OnDragEnter
CView::OnDragOver
CView::OnDropEx
CView::OnDragLeave
COleDropTarget::OnDrop

Concepts

CView Members

CView::OnDropEx

Called by the framework when the user releases a data object over a valid drop target.

```
virtual DROPEFFECT OnDropEx(
   COleDataObject* pDataObject,
   DROPEFFECT dropDefault,
   DROPEFFECT dropList,
   CPoint point
);
```

Parameters

pDataObject

Points to the COleDataObject that is dropped into the drop target.

dropDefault

The effect that the user chose for the default drop operation based on the current key state. It may be **DROPEFFECT_NONE**. Drop effects are discussed in the Remarks section.

dropList

A list of the drop effects that the drop source supports. Drop effect values can be combined using the bitwise OR (**|**) operation. Drop effects are discussed in the Remarks section.

point

The current mouse position relative to the view client area.

Return Value

The drop effect that resulted from the drop attempt at the location specified by *point*. This must be one of the values indicated by *dropEffectList*. Drop effects are discussed in the Remarks section.

Remarks

The default implementation is to do nothing and return a dummy value (-1) to indicate that the framework should call the OnDrop handler.

Override this function to implement the effect of an right mouse-button drag and drop. Right mouse-button drag and drop typically displays a menu of choices when the right mouse-button is released.

Your override of **OnDropEx** should query for the right mouse-button. You can call GetKeyState or store the right mouse-button state from your OnDragEnter handler.

- If the right mouse-button is down, your override should display a popup menu which offers the drop effects support by the drop source.
    - Examine *dropList* to determine the drop effects supported by the drop source. Enable only these actions on the popup menu.
    - Use SetMenuDefaultItem to set the default action based on *dropDefault*.
    - Finally, take the action indicated by the user selection from the popup menu.
- If the right mouse-button is not down, your override should process this as a standard drop request. Use the drop effect specified in *dropDefault*. Alternately, your override can return the dummy value (-1) to indicate that **OnDrop** will handle this drop operation.

Use *pDataObject* to examine the **COleDataObject** for Clipboard data format and data dropped at the specified point.

Drop effects describe the action associated with a drop operation. See the following list of drop effects:

- **DROPEFFECT_NONE** A drop would not be allowed.
- **DROPEFFECT_COPY** A copy operation would be performed.

- **DROPEFFECT_MOVE** A move operation would be performed.
- **DROPEFFECT_LINK** A link from the dropped data to the original data would be established.
- **DROPEFFECT_SCROLL** Indicates that a drag scroll operation is about to occur or is occurring in the target.

For more information on setting the default menu command, see SetMenuDefaultItem in the *Platform SDK* and CMenu::GetSafeHmenu in this volume.

See Also

Reference

CView Class
Hierarchy Chart
CView::OnDragEnter
CView::OnDragOver
CView::OnDrop
CView::OnDragLeave
COleDropTarget::OnDropEx

Concepts

CView Members

CView::OnEndPrinting

Called by the framework after a document has been printed or previewed.

```
virtual void OnEndPrinting(
   CDC* pDC,
   CPrintInfo* pInfo
);
```

Parameters

pDC

Points to the printer device context.

pInfo

Points to a CPrintInfo structure that describes the current print job.

Remarks

The default implementation of this function does nothing. Override this function to free any GDI resources you allocated in the OnBeginPrinting member function.

Smart Device Developer Notes

This method is not supported in smart device projects.

See Also

Reference

CView Class
Hierarchy Chart
CView::OnBeginPrinting

Concepts

CView Members

CView::OnEndPrintPreview

Called by the framework when the user exits print preview mode.

```
virtual void OnEndPrintPreview(
   CDC* pDC,
   CPrintInfo* pInfo,
   POINT point,
   CPreviewView* pView
);
```

Parameters

pDC

Points to the printer device context.

pInfo

Points to a CPrintInfo structure that describes the current print job.

point

Specifies the point on the page that was last displayed in preview mode.

pView

Points to the view object used for previewing.

Remarks

The default implementation of this function calls the OnEndPrinting member function and restores the main frame window to the state it was in before print preview began. Override this function to perform special processing when preview mode is terminated. For example, if you want to maintain the user's position in the document when switching from preview mode to normal display mode, you can scroll to the position described by the *point* parameter and the **m_nCurPage** member of the **CPrintInfo** structure that the *pInfo* parameter points to.

Always call the base class version of **OnEndPrintPreview** from your override, typically at the end of the function.

See Also

Reference

CView Class
Hierarchy Chart
CPrintInfo Structure
CView::OnEndPrinting

Concepts

CView Members

CView::OnInitialUpdate

Called by the framework after the view is first attached to the document, but before the view is initially displayed.

```
virtual void OnInitialUpdate( );
```

Remarks

The default implementation of this function calls the OnUpdate member function with no hint information (that is, using the default values of 0 for the *lHint* parameter and **NULL** for the *pHint* parameter). Override this function to perform any one-time initialization that requires information about the document. For example, if your application has fixed-sized documents, you can use this function to initialize a view's scrolling limits based on the document size. If your application supports variable-sized documents, use OnUpdate to update the scrolling limits every time the document changes.

See Also

Reference

CView Class

Hierarchy Chart

CView::OnUpdate

Concepts

CView Members

CView::OnPrepareDC

Called by the framework before the OnDraw member function is called for screen display and before the OnPrint member function is called for each page during printing or print preview.

```
virtual void OnPrepareDC(
   CDC* pDC,
   CPrintInfo* pInfo = NULL
);
```

Parameters

pDC

Points to the device context to be used for rendering an image of the document.

pInfo

Points to a CPrintInfo structure that describes the current print job if **OnPrepareDC** is being called for printing or print preview; the **m_nCurPage** member specifies the page about to be printed. This parameter is **NULL** if **OnPrepareDC** is being called for screen display.

Remarks

The default implementation of this function does nothing if the function is called for screen display. However, this function is overridden in derived classes, such as CScrollView, to adjust attributes of the device context; consequently, you should always call the base class implementation at the beginning of your override.

If the function is called for printing, the default implementation examines the page information stored in the *pInfo* parameter. If the length of the document has not been specified, **OnPrepareDC** assumes the document to be one page long and stops the print loop after one page has been printed. The function stops the print loop by setting the **m_bContinuePrinting** member of the structure to **FALSE**.

Override **OnPrepareDC** for any of the following reasons:

- To adjust attributes of the device context as needed for the specified page. For example, if you need to set the mapping mode or other characteristics of the device context, do so in this function.
- To perform print-time pagination. Normally you specify the length of the document when printing begins, using the OnPreparePrinting member function. However, if you don't know in advance how long the document is (for example, when printing an undetermined number of records from a database), override **OnPrepareDC** to test for the end of the document while it is being printed. When there is no more of the document to be printed, set the **m_bContinuePrinting** member of the **CPrintInfo** structure to **FALSE**.
- To send escape codes to the printer on a page-by-page basis. To send escape codes from **OnPrepareDC**, call the **Escape** member function of the *pDC* parameter.

Call the base class version of **OnPrepareDC** at the beginning of your override.

Example

```
void CMyView::OnPrepareDC (CDC* pDC, CPrintInfo* pInfo)
{
   CView::OnPrepareDC(pDC, pInfo);

   // If we are printing, set the mapmode and the window
   // extent properly, then set viewport extent. Use the
   // SetViewportOrg member function in the CDC class to
   // move the viewport origin to the center of the view.

   if(pDC->IsPrinting()) // Is the DC a printer DC.
   {
      CRect rect;
      GetClientRect (&rect);

      int oldMapMode = pDC->SetMapMode(MM_ISOTROPIC);
```

```
        CSize ptOldWinExt = pDC->SetWindowExt(1000, 1000);
        ASSERT( ptOldWinExt.cx != 0 && ptOldWinExt.cy != 0 );
        CSize ptOldViewportExt = pDC->SetViewportExt(rect.Width(),
            -rect.Height());
        ASSERT( ptOldViewportExt.cx != 0 && ptOldViewportExt.cy != 0 );
        CPoint ptOldOrigin = pDC->SetViewportOrg(rect.Width()/2,
            rect.Height()/2);
    }
}
```

Smart Device Developer Notes

This method is not supported in smart device projects.

See Also

Reference

CView Class
Hierarchy Chart
CDC::Escape
CPrintInfo Structure
CView::OnBeginPrinting
CView::OnDraw
CView::OnPreparePrinting
CView::OnPrint

Concepts

CView Members

CView::OnPreparePrinting

Called by the framework before a document is printed or previewed.

```
virtual BOOL OnPreparePrinting(
   CPrintInfo* pInfo
);
```

Parameters

pInfo

Points to a CPrintInfo structure that describes the current print job.

Return Value

Nonzero to begin printing; 0 if the print job has been canceled.

Remarks

The default implementation does nothing.

You must override this function to enable printing and print preview. Call the DoPreparePrinting member function, passing it the *pInfo* parameter, and then return its return value; **DoPreparePrinting** displays the Print dialog box and creates a printer device context. If you want to initialize the Print dialog box with values other than the defaults, assign values to the members of *pInfo*. For example, if you know the length of the document, pass the value to the SetMaxPage member function of *pInfo* before calling **DoPreparePrinting**. This value is displayed in the To: box in the Range portion of the Print dialog box.

DoPreparePrinting does not display the Print dialog box for a preview job. If you want to bypass the Print dialog box for a print job, check that the **m_bPreview** member of *pInfo* is **FALSE** and then set it to **TRUE** before passing it to **DoPreparePrinting**; reset it to **FALSE** afterwards.

If you need to perform initializations that require access to the **CDC** object representing the printer device context (for example, if you need to know the page size before specifying the length of the document), override the **OnBeginPrinting** member function.

If you want to set the value of the **m_nNumPreviewPages** or **m_strPageDesc** members of the *pInfo* parameter, do so after calling **DoPreparePrinting**. The **DoPreparePrinting** member function sets **m_nNumPreviewPages** to the value found in the application's .INI file and sets **m_strPageDesc** to its default value.

Example

Override **OnPreparePrinting** and call **DoPreparePrinting** from the override so that the framework will display a Print dialog box and create a printer DC for you.

```
BOOL CMyView::OnPreparePrinting(CPrintInfo *pInfo)
{
   return CEditView::DoPreparePrinting(pInfo);
}
```

If you know how many pages the document contains, set the maximum page in **OnPreparePrinting** before calling **DoPreparePrinting**. The framework will display the maximum page number in the "to" box of the Print dialog box.

```
BOOL CMyView::OnPreparePrinting(CPrintInfo* pInfo)
{
   //The document has 2 pages.
   pInfo->SetMaxPage(2);
   return CEditView::DoPreparePrinting(pInfo);
}
```

Smart Device Developer Notes

This method is not supported in smart device projects.

See Also

Reference

CView Class
Hierarchy Chart
CPrintInfo Structure
CView::DoPreparePrinting
CView::OnBeginPrinting
CView::OnPrepareDC
CView::OnPrint

Concepts

CView Members

CView::OnPrint

Called by the framework to print or preview a page of the document.

```
virtual void OnPrint(
   CDC* pDC,
   CPrintInfo* pInfo
);
```

Parameters

pDC

Points to the printer device context.

pInfo

Points to a **CPrintInfo** structure that describes the current print job.

Remarks

For each page being printed, the framework calls this function immediately after calling the OnPrepareDC member function. The page being printed is specified by the **m_nCurPage** member of the CPrintInfo structure that *pInfo* points to. The default implementation calls the OnDraw member function and passes it the printer device context.

Override this function for any of the following reasons:

- To allow printing of multipage documents. Render only the portion of the document that corresponds to the page currently being printed. If you're using **OnDraw** to perform the rendering, you can adjust the viewport origin so that only the appropriate portion of the document is printed.
- To make the printed image look different from the screen image (that is, if your application is not WYSIWYG). Instead of passing the printer device context to **OnDraw**, use the device context to render an image using attributes not shown on the screen.

  If you need GDI resources for printing that you don't use for screen display, select them into the device context before drawing and deselect them afterwards. These GDI resources should be allocated in OnBeginPrinting and released in OnEndPrinting.
- To implement headers or footers. You can still use **OnDraw** to do the rendering by restricting the area that it can print on.

Note that the **m_rectDraw** member of the *pInfo* parameter describes the printable area of the page in logical units.

Do not call **OnPrepareDC** in your override of **OnPrint**; the framework calls **OnPrepareDC** automatically before calling **OnPrint**.

Example

The following is a skeleton for an overridden **OnPrint** function:

```
void CMyView::OnPrint( CDC *pDC, CPrintInfo *pInfo )
{
   // Print headers and/or footers, if desired.
   // Find portion of document corresponding to pInfo->m_nCurPage.
   OnDraw( pDC );
}
```

For another example, see CRichEditView::PrintInsideRect.

Smart Device Developer Notes

This method is not supported in smart device projects.

See Also

Reference

CView Class
Hierarchy Chart
CView::OnBeginPrinting
CView::OnEndPrinting
CView::OnPrepareDC
CView::OnDraw

Concepts

CView Members

CView::OnScroll

Called by the framework to determine whether scrolling is possible.

```
virtual BOOL OnScroll(
   UINT nScrollCode,
   UINT nPos,
   BOOL bDoScroll = TRUE
);
```

Parameters

nScrollCode

A scroll-bar code that indicates the user's scrolling request. This parameter is composed of two parts: a low-order byte, which determines the type of scrolling occurring horizontally, and a high-order byte, which determines the type of scrolling occurring vertically:

- **SB_BOTTOM** Scrolls to bottom.
- **SB_LINEDOWN** Scrolls one line down.
- **SB_LINEUP** Scrolls one line up.
- **SB_PAGEDOWN** Scrolls one page down.
- **SB_PAGEUP** Scrolls one page up.
- **SB_THUMBTRACK** Drags scroll box to specified position. The current position is specified in *nPos*.
- **SB_TOP** Scrolls to top.

nPos

Contains the current scroll-box position if the scroll-bar code is **SB_THUMBTRACK**; otherwise it is not used. Depending on the initial scroll range, *nPos* may be negative and should be cast to an **int** if necessary.

bDoScroll

Determines whether you should actually do the specified scrolling action. If **TRUE,** then scrolling should take place; if **FALSE**, then scrolling should not occur.

Return Value

If *bDoScroll* is **TRUE** and the view was actually scrolled, then return nonzero; otherwise 0. If *bDoScroll* is **FALSE**, then return the value that you would have returned if *bDoScroll* were **TRUE**, even though you don't actually do the scrolling.

Remarks

In one case this function is called by the framework with *bDoScroll* set to **TRUE** when the view receives a scrollbar message. In this case, you should actually scroll the view. In the other case this function is called with *bDoScroll* set to **FALSE** when an OLE item is initially dragged into the auto-scrolling region of a drop target before scrolling actually takes place. In this case, you should not actually scroll the view.

See Also

Reference

CView Class
Hierarchy Chart
CView::OnScrollBy
COleClientItem Class

Concepts

CView Members

CView::OnScrollBy

Called by the framework when the user views an area beyond the present view of the document, either by dragging an OLE item against the view's current borders or by manipulating the vertical or horizontal scrollbars.

```
virtual BOOL OnScrollBy(
   CSize sizeScroll,
   BOOL bDoScroll = TRUE
);
```

Parameters

sizeScroll

Number of pixels scrolled horizontally and vertically.

bDoScroll

Determines whether scrolling of the view occurs. If **TRUE,** then scrolling takes place; if **FALSE**, then scrolling does not occur.

Return Value

Nonzero if the view was able to be scrolled; otherwise 0.

Remarks

In derived classes this method checks to see whether the view is scrollable in the direction the user requested and then updates the new region if necessary. This function is automatically called by CWnd::OnHScroll and CWnd::OnVScroll to perform the actual scrolling request.

The default implementation of this method does not change the view, but if it is not called, the view will not scroll in a **CScrollView**-derived class.

If the document width or height exceeds 32767 pixels, scrolling past 32767 will fail because **OnScrollBy** is called with an invalid *sizeScroll* argument.

See Also

Reference

CView Class

Hierarchy Chart

Concepts

CView Members

CView::OnUpdate

Called by the framework after the view's document has been modified; this function is called by CDocument::UpdateAllViews and allows the view to update its display to reflect those modifications.

```
virtual void OnUpdate(
   CView* pSender,
   LPARAM lHint,
   CObject* pHint
);
```

Parameters

pSender

Points to the view that modified the document, or **NULL** if all views are to be updated.

lHint

Contains information about the modifications.

pHint

Points to an object storing information about the modifications.

Remarks

It is also called by the default implementation of OnInitialUpdate. The default implementation invalidates the entire client area, marking it for painting when the next **WM_PAINT** message is received. Override this function if you want to update only those regions that map to the modified portions of the document. To do this you must pass information about the modifications using the hint parameters.

To use *lHint*, define special hint values, typically a bitmask or an enumerated type, and have the document pass one of these values. To use *pHint*, derive a hint class from CObject and have the document pass a pointer to a hint object; when overriding **OnUpdate**, use the CObject::IsKindOf member function to determine the run-time type of the hint object.

Typically you should not perform any drawing directly from **OnUpdate**. Instead, determine the rectangle describing, in device coordinates, the area that requires updating; pass this rectangle to CWnd::InvalidateRect. This causes painting to occur the next time a WM_PAINT message is received.

If *lHint* is 0 and *pHint* is **NULL**, the document has sent a generic update notification. If a view receives a generic update notification, or if it cannot decode the hints, it should invalidate its entire client area.

See Also

Reference

CView Class
Hierarchy Chart
CDocument::UpdateAllViews
CView::OnInitialUpdate
CWnd::Invalidate
CWnd::InvalidateRect

Concepts

CView Members

CWaitCursor Class

Provides a one-line way to show a wait cursor, which is usually displayed as an hourglass, while you're doing a lengthy operation.

```
class CWaitCursor
```

Remarks

CWaitCursor does not have a base class.

Good Windows programming practices require that you display a wait cursor whenever you're performing an operation that takes a noticeable amount of time.

To display a wait cursor, just define a **CWaitCursor** variable before the code that performs the lengthy operation. The object's constructor automatically causes the wait cursor to be displayed.

When the object goes out of scope (at the end of the block in which the **CWaitCursor** object is declared), its destructor sets the cursor to the previous cursor. In other words, the object performs the necessary clean-up automatically.

Note
Because of how their constructors and destructors work, **CWaitCursor** objects are always declared as local variables — they're never declared as global variables nor are they allocated with **new**.

If you perform an operation which might cause the cursor to be changed, such as displaying a message box or dialog box, call the Restore member function to restore the wait cursor. It is okay to call **Restore** even when a wait cursor is currently displayed.

Another way to display a wait cursor is to use the combination of CCmdTarget::BeginWaitCursor, CCmdTarget::EndWaitCursor, and perhaps CCmdTarget::RestoreWaitCursor. However, **CWaitCursor** is easier to use because you don't need to set the cursor to the previous cursor when you're done with the lengthy operation.

Note
MFC sets and restores the cursor using the CWinApp::DoWaitCursor virtual function. You can override this function to provide custom behavior.

Requirements

Header: afxwin.h

Example

```
BOOL CMyWnd::SomeLengthyProcess()
{
   CWaitCursor wait;
   //Do the lengthProcessing.

   MessageBox("Some result");      //This changes the cursor.
   wait.Restore();         //Restore the Wait cursor.
   //Continue Processing.

   //The destructor changes the cursor back to Regular cursor.
   return TRUE;

}
```

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart

CCmdTarget::BeginWaitCursor

CCmdTarget::EndWaitCursor

CCmdTarget::RestoreWaitCursor

CWinApp::DoWaitCursor

Other Resources

CWaitCursor Members

CWaitCursor Members

Construction/Destruction

CWaitCursor	Constructs a **CWaitCursor** object and displays the wait cursor.

Operations

Restore	Restores the wait cursor after it's been changed.

See Also

Reference

CWaitCursor Class

Hierarchy Chart

CWaitCursor Member Functions

For information about the member functions in **CWaitCursor**, see CWaitCursor Members.

CWaitCursor::CWaitCursor

To display a wait cursor, just declare a **CWaitCursor** object before the code that performs the lengthy operation.

```
CWaitCursor( );
```

Remarks

The constructor automatically causes the wait cursor to be displayed.

When the object goes out of scope (at the end of the block in which the **CWaitCursor** object is declared), its destructor sets the cursor to the previous cursor. In other words, the object performs the necessary clean-up automatically.

You can take advantage of the fact that the destructor is called at the end of the block (which might be before the end of the function) to make the wait cursor active in only part of your function. This technique is shown in the second example below.

Note

Because of how their constructors and destructors work, **CWaitCursor** objects are always declared as local variables — they're never declared as global variables, nor are they allocated with **new**.

Example

```
// The following example illustrates the most common case
// of displaying the wait cursor during some lengthy
// processing.

void LengthyFunction( )
{
   // perhaps you display a dialog box before displaying a
   // wait cursor

   CWaitCursor wait;   // display wait cursor

   // do some lengthy processing

}  // destructor automatically removes the wait cursor

// This example shows using a CWaitCursor object inside a block
// so the wait cursor is displayed only while the program is
// performing a lengthy operation.

void ConditionalFunction( )
{
   if ( SomeCondition )
   {
      CWaitCursor wait;   // display wait cursor in this block only

      // do some lengthy processing

   }  // at this point, the destructor removes the wait cursor
   else
   {
      // no wait cursor--only quick processing
   }
}
```

See Also

Reference

CWaitCursor Class
Hierarchy Chart
CWaitCursor::Restore
CCmdTarget::BeginWaitCursor

CCmdTarget::EndWaitCursor

CWaitCursor::Restore

To restore the wait cursor, call this function after performing an operation, such as displaying a message box or dialog box, which might change the wait cursor to another cursor.

```
void Restore( );
```

Remarks

It is OK to call **Restore** even when the wait cursor is currently displayed.

If you need to restore the wait cursor while in a function other than the one in which the **CWaitCursor** object is declared, you can call CCmdTarget::RestoreWaitCursor.

Example

```
// This example illustrates performing an operation
// which changes the wait cursor. You should call
// CWaitCursor::Restore to restore the wait
// cursor after an operation which changes the cursor.

void AnotherLengthyFunction( )
{
   CWaitCursor wait;   // display wait cursor

   // do some lengthy processing

   // The dialog box will normally change the cursor to
   // the standard arrow cursor.
   CSomeDialog dlg;
   dlg.DoModal( );

   // It is necessary to call Restore here in order
   // to change the cursor back to the wait cursor.
   wait.Restore( );

   // do some more lengthy processing

   // destructor automatically removes the wait cursor
}

// If the wait cursor is changed by a function called by
// the function which created the wait cursor, you
// can call CCmdTarget::RestoreWaitCursor to restore
// the wait cursor.
void CalledFunction()
{
   CSomeDialog dlg;
   dlg.DoModal();

   // Since CWinApp is derived from CCmdTarget, we can use a
   // pointer to our application object to make the call to
   // CCmdTarget::RestoreWaitCursor.
   AfxGetApp()->RestoreWaitCursor( );

   // Yet more lengthy processing...
}
```

See Also

Reference

CWaitCursor Class

Hierarchy Chart

CCmdTarget::RestoreWaitCursor

CWinApp Class

The base class from which you derive a Windows application object.

```
class CWinApp : public CWinThread
```

Remarks

An application object provides member functions for initializing your application (and each instance of it) and for running the application.

Each application that uses the Microsoft Foundation classes can only contain one object derived from **CWinApp**. This object is constructed when other C++ global objects are constructed and is already available when Windows calls the **WinMain** function, which is supplied by the Microsoft Foundation Class Library. Declare your derived **CWinApp** object at the global level.

When you derive an application class from **CWinApp**, override the InitInstance member function to create your application's main window object.

In addition to the **CWinApp** member functions, the Microsoft Foundation Class Library provides the following global functions to access your **CWinApp** object and other global information:

- AfxGetApp Obtains a pointer to the **CWinApp** object.
- AfxGetInstanceHandle Obtains a handle to the current application instance.
- AfxGetResourceHandle Obtains a handle to the application's resources.
- AfxGetAppName Obtains a pointer to a string containing the application's name. Alternately, if you have a pointer to the **CWinApp** object, use **m_pszExeName** to get the application's name.

See CWinApp: The Application Class for more on the **CWinApp** class, including an overview of the following:

- **CWinApp**-derived code written by the Application Wizard.
- **CWinApp**'s role in the execution sequence of your application.
- **CWinApp**'s default member function implementations.
- **CWinApp**'s key overridables.

The **m_hPrevInstance** data member no longer exists. For information on detecting a previous instance of **CWinApp**, see the Knowledge Base article "How To Identify a Previous Instance of an Application" (KB106385) at http://support.microsoft.com/default.aspx?scid=kb;en-us;106385.

Requirements

Header: afxwin.h

See Also

Reference

CWinThread Class

Hierarchy Chart

Other Resources

CWinApp Members

CWinApp Members

Base Class Members

Data Members

Construction

Operations

Overridables

Initialization

Command Handlers

Base Class Members

CObject Members

CCmdTarget Members

CWinThread Members

Data Members

m_bHelpMode	Indicates if the user is in Help context mode (typically invoked with SHIFT+F1).
m_bUseHtmlHelp	Indicates whether the application uses HTMLHelp or WinHelp.
m_hInstance	Identifies the current instance of the application.
m_eHelpType	Specifies the type of help used by the application.
m_lpCmdLine	Points to a null-terminated string that specifies the command line for the application.
m_nCmdShow	Specifies how the window is to be shown initially.
m_pActiveWnd	Pointer to the main window of the container application when an OLE server is in-place active.
m_pszAppName	Specifies the name of the application.
m_pszExeName	The module name of the application.
m_pszHelpFilePath	The path to the application's Help file.
m_pszProfileName	The application's .INI filename.
m_pszRegistryKey	Used to determine the full registry key for storing application profile settings.

Construction

CWinApp	Constructs a **CWinApp** object.

Operations

AddDocTemplate	Adds a document template to the application's list of available document templates.
AddToRecentFileList	Adds a filename to the most recently used (MRU) file list.
CreatePrinterDC	Creates a printer device context.

DelRegTree	Deletes a specified key and all its subkeys.
GetFirstDocTemplatePosition	Retrieves the position of the first document template.
GetNextDocTemplate	Retrieves the position of a document template. Can be used recursively.
GetPrinterDeviceDefaults	Retrieves the printer device defaults.
GetProfileBinary	Retrieves binary data from an entry in the application's .INI file.
GetProfileInt	Retrieves an integer from an entry in the application's .INI file.
GetProfileString	Retrieves a string from an entry in the application's .INI file.
LoadCursor	Loads a cursor resource.
LoadIcon	Loads an icon resource.
LoadOEMCursor	Loads a Windows OEM predefined cursor that the **OCR_** constants specify in WINDOWS.H.
LoadOEMIcon	Loads a Windows OEM predefined icon that the **OIC_** constants specify in WINDOWS.H.
LoadStandardCursor	Loads a Windows predefined cursor that the **IDC_** constants specify in WINDOWS.H.
LoadStandardIcon	Loads a Windows predefined icon that the **IDI_** constants specify in WINDOWS.H.
OpenDocumentFile	Called by the framework to open a document from a file.
ParseCommandLine	Parses individual parameters and flags in the command line.
ProcessShellCommand	Handles command-line arguments and flags.
RunAutomated	Tests the application's command line for the **/Automation** option. Obsolete. Instead, use the val ue in CCommandLineInfo::m_bRunAutomated after calling ParseCommandLine.
RunEmbedded	Tests the application's command line for the **/Embedding** option. Obsolete. Instead, use the val ue in CCommandLineInfo::m_bRunEmbedded after calling ParseCommandLine.
SelectPrinter	Selects a printer previously indicated by a user through a print dialog box.
WriteProfileBinary	Writes binary data to an entry in the application's .INI file.
WriteProfileInt	Writes an integer to an entry in the application's .INI file.
WriteProfileString	Writes a string to an entry in the application's .INI file.

Overridables

CloseAllDocuments	Closes all open documents.
DoMessageBox	Implements AfxMessageBox for the application.
DoWaitCursor	Turns the wait cursor on and off.

ExitInstance	Override to clean up when your application terminates.
HideApplication	Hides the application before closing all documents.
HtmlHelp	Calls the **HTMLHelp** Windows function.
InitInstance	Override to perform Windows instance initialization, such as creating your window objects.
OnDDECommand	Called by the framework in response to a dynamic data exchange (DDE) execute command.
OnIdle	Override to perform application-specific idle-time processing.
PreTranslateMessage	Filters messages before they are dispatched to the Windows functions TranslateMessage and DispatchMessage.
ProcessMessageFilter	Intercepts certain messages before they reach the application.
ProcessWndProcException	Intercepts all unhandled exceptions thrown by the application's message and command handlers.
Run	Runs the default message loop. Override to customize the message loop.
SaveAllModified	Prompts the user to save all modified documents.
WinHelp	Calls the **WinHelp** Windows function.

Initialization

EnableHtmlHelp	Implements HTMLHelp for the application, rather than WinHelp.
EnableShellOpen	Allows the user to open data files from the Windows File Manager.
LoadStdProfileSettings	Loads standard .INI file settings and enables the MRU file list feature.
RegisterShellFileTypes	Registers all the application's document types with the Windows File Manager.
Register	Performs customized registration.
SetHelpMode	Sets and initializes the type of help used by the application.
GetHelpMode	Retrieves the type of help used by the application.
Unregister	Unregisters everything known to be registered by the **CWinApp** object.
UnregisterShellFileTypes	Unregisters all the application's document types with the Windows File Manager.
SetRegistryKey	Causes application settings to be stored in the registry instead of .INI files.

Command Handlers

OnContextHelp	Handles SHIFT+F1 Help within the application.
OnFileNew	Implements the **ID_FILE_NEW** command.
OnFileOpen	Implements the **ID_FILE_OPEN** command.

OnFilePrintSetup	Implements the **ID_FILE_PRINT_SETUP** command.
OnHelp	Handles F1 Help within the application (using the current context).
OnHelpFinder	Handles the **ID_HELP_FINDER** and **ID_DEFAULT_HELP** commands.
OnHelpIndex	Handles the **ID_HELP_INDEX** command and provides a default Help topic.
OnHelpUsing	Handles the **ID_HELP_USING** command.

See Also

Reference

CWinApp Class
Hierarchy Chart

CWinApp Member Functions

For information about the member functions in **CWinApp**, see CWinApp Members.

CWinApp::AddDocTemplate

Call this member function to add a document template to the list of available document templates that the application maintains.

```
void AddDocTemplate(
   CDocTemplate* pTemplate
);
```

Parameters

pTemplate

A pointer to the **CDocTemplate** to be added.

Remarks

You should add all document templates to an application before you call RegisterShellFileTypes.

Example

```
BOOL CMyApp::InitInstance()
{
   // ...
   // The following code is produced by the Application Wizard when you
   // choose the MDI (multiple document interface) option.
   CMultiDocTemplate* pDocTemplate;
   pDocTemplate = new CMultiDocTemplate(
      IDR_MYTYPE,
      RUNTIME_CLASS(CMyDoc),
      RUNTIME_CLASS(CMDIChildWnd),        // standard MDI child frame
      RUNTIME_CLASS(CMyView));
   AddDocTemplate(pDocTemplate);
     // ...
}
```

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::RegisterShellFileTypes
CMultiDocTemplate Class
CSingleDocTemplate Class

Other Resources

CWinApp Members

CWinApp::AddToRecentFileList

Call this member function to add *lpszPathName* to the MRU file list.

```
virtual void AddToRecentFileList(
   LPCTSTR lpszPathName
);
```

Parameters

lpszPathName

The path of the file.

Remarks

You should call the LoadStdProfileSettings member function to load the current MRU file list before you use this member function.

The framework calls this member function when it opens a file or executes the Save As command to save a file with a new name.

Example

```
    // This adds the pathname c:\temp\test.doc to the top of
    // the most recently used (MRU) list in the File menu.
     AfxGetApp()->AddToRecentFileList("c:\\temp\\test.doc");
```

See Also

Reference

CWinApp Class

Hierarchy Chart

CWinApp::LoadStdProfileSettings

Other Resources

CWinApp Members

CWinApp::CloseAllDocuments

Call this member function to close all open documents before exiting.

```
void CloseAllDocuments(
   BOOL bEndSession
);
```

Parameters

bEndSession

Specifies whether or not the Windows session is being ended. It is **TRUE** if the session is being ended; otherwise **FALSE**.

Remarks

Call HideApplication before calling **CloseAllDocuments**.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::SaveAllModified
CWinApp::HideApplication

Other Resources

CWinApp Members

CWinApp::CreatePrinterDC

Call this member function to create a printer device context (DC) from the selected printer.

```
BOOL CreatePrinterDC(
   CDC& dc
);
```

Parameters

dc

A reference to a printer device context.

Return Value

Nonzero if the printer device context is created successfully; otherwise 0.

Remarks

CreatePrinterDC initializes the device context that you pass in by reference, so you can use it to print.

If the function is successful, when you have finished printing, you must destroy the device context. You can let the destructor of the CDC object do it, or you can do it explicitly by calling CDC::DeleteDC.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::SelectPrinter

Other Resources

CWinApp Members

CWinApp::CWinApp

Constructs a **CWinApp** object and passes *lpszAppName* to be stored as the application name.

```
CWinApp(
   LPCTSTR lpszAppName = NULL
);
```

Parameters

lpszAppName

A null-terminated string that contains the application name that Windows uses. If this argument is not supplied or is **NULL**, **CWinApp** uses the resource string **AFX_IDS_APP_TITLE** or the filename of the executable file.

Remarks

You should construct one global object of your **CWinApp**-derived class. You can have only one **CWinApp** object in your application. The constructor stores a pointer to the **CWinApp** object so that **WinMain** can call the object's member functions to initialize and run the application.

See Also

Reference

CWinApp Class

Hierarchy Chart

Other Resources

CWinApp Members

CWinApp::DelRegTree

Deletes a specific registry key and all its subkeys.

```
LONG DelRegTree(
   HKEY hParentKey,
   const CString& strKeyName
);
```

Parameters

hParentKey

Handle to a registry key.

strKeyName

The name of the registry key to be deleted.

Remarks

Call this function to delete the specified key and its subkeys.

See Also

Reference

CWinApp Class

Hierarchy Chart

Other Resources

CWinApp Members

CWinApp::DoMessageBox

The framework calls this member function to implement a message box for the global function AfxMessageBox.

```
virtual int DoMessageBox(
   LPCTSTR lpszPrompt,
   UINT nType,
   UINT nIDPrompt
);
```

Parameters

lpszPrompt

Address of text in the message box.

nType

The message box style.

nIDPrompt

An index to a Help context string.

Return Value

Returns the same values as **AfxMessageBox**.

Remarks

Do not call this member function to open a message box; use **AfxMessageBox** instead.

Override this member function to customize your application-wide processing of **AfxMessageBox** calls.

See Also

Reference

CWinApp Class
Hierarchy Chart
AfxMessageBox
MessageBox

Other Resources

CWinApp Members

CWinApp::DoWaitCursor

This member function is called by the framework to implement CWaitCursor, CCmdTarget::BeginWaitCursor, CCmdTarget::EndWaitCursor, and CCmdTarget::RestoreWaitCursor.

```
virtual void DoWaitCursor(
   int nCode
);
```

Parameters

nCode

If this parameter is 1, a wait cursor appears. If 0, the wait cursor is restored without incrementing the reference count. If –1, the wait cursor ends.

Remarks

The default implements an hourglass cursor. **DoWaitCursor** maintains a reference count. When positive, the hourglass cursor is displayed.

While you would not normally call **DoWaitCursor** directly, you could override this member function to change the wait cursor or to do additional processing while the wait cursor is displayed.

For an easier, more streamlined way to implement a wait cursor, use **CWaitCursor**.

Example

```
// The following example shows how to display the
// hourglass cursor during some lengthy processing

void CMyApp::OnLButtonDown()
{
   AfxGetApp()->DoWaitCursor(1); // 1->>display the hourglass cursor

   // do some lengthy processing

   AfxGetApp()->DoWaitCursor(-1); // -1->>remove the hourglass cursor
}

// The next example shows DoWaitCursor with parameter 0. It restores
// the hourglass cursor.

void CMyApp::OnLButtonDown()
{
   AfxGetApp()->DoWaitCursor(1); // display the hourglass cursor

   // do some lengthy processing

   // The message box will normally change the cursor to
   // the standard arrow cursor, and leave the cursor in
   // as the standard arrow cursor when the message box is
   // closed.
   AfxMessageBox ("DoWaitCursor Sample");

   // Call DoWaitCursor with parameter 0 to restore
   // the cursor back to the hourglass cursor.
   AfxGetApp()->DoWaitCursor(0);

   // do some more lengthy processing

   AfxGetApp()->DoWaitCursor(-1); // remove the hourglass cursor
}
```

See Also

Reference

CWinApp Class
Hierarchy Chart
CCmdTarget::BeginWaitCursor
CCmdTarget::EndWaitCursor
CCmdTarget::RestoreWaitCursor
CWaitCursor Class

Other Resources

CWinApp Members

CWinApp::EnableHtmlHelp

Call this member function from within the constructor of your **CWinApp**-derived class to use HTMLHelp for your application's help.

```
void EnableHtmlHelp( );
```

Remarks

EnableHtmlHelp sets m_bUseHtmlHelp to **TRUE**.

See Also

Tasks

Displaying F1 Help for a Dialog Box or Menu Option

Reference

CWinApp Class

Hierarchy Chart

CWinApp::HtmlHelp

Other Resources

CWinApp Members

CWinApp::EnableShellOpen

Call this function, typically from your **InitInstance** override, to enable your application's users to open data files when they double-click the files from within the Windows File Manager.

```
void EnableShellOpen( );
```

Remarks

Call the **RegisterShellFileTypes** member function in conjunction with this function, or provide a .REG file with your application for manual registration of document types.

Example

```
BOOL CMyApp::InitInstance()
{
   // ...

   CMultiDocTemplate* pDocTemplate;
   pDocTemplate = new CMultiDocTemplate(
      IDR_MYTYPE,
      RUNTIME_CLASS(CMyDoc),
      RUNTIME_CLASS(CMDIChildWnd),        // standard MDI child frame
      RUNTIME_CLASS(CMyView));
   AddDocTemplate(pDocTemplate);

   // Create main MDI Frame window.
   CMainFrame* pMainFrame = new CMainFrame;
   if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
      return FALSE;
   // Save the pointer to the main frame window.  This is the
   // only way the framework will have knowledge of what the
   // main frame window is.
   m_pMainWnd = pMainFrame;

   // enable file manager drag/drop and DDE Execute open
   EnableShellOpen();
   RegisterShellFileTypes();
   // ...

   // Show the   main window using the nCmdShow parameter
   // passed to the application when it was first launched.
   pMainFrame->ShowWindow(m_nCmdShow);
   pMainFrame->UpdateWindow();

   // ...
}
```

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::OnDDECommand
CWinApp::RegisterShellFileTypes

Other Resources

CWinApp Members

CWinApp::ExitInstance

Called by the framework from within the **Run** member function to exit this instance of the application.

```
virtual int ExitInstance( );
```

Return Value

The application's exit code; 0 indicates no errors, and values greater than 0 indicate an error. This value is used as the return value from **WinMain**.

Remarks

Do not call this member function from anywhere but within the **Run** member function.

The default implementation of this function writes framework options to the application's .INI file. Override this function to clean up when your application terminates.

Example

```
// MySampleApp.cpp.

int CMySampleApp::ExitInstance()
{
   // TODO: Add your specialized code here and/or
   // call the base class.

   if ( m_pMySampleMem )
      delete m_pMySampleMem;

   DoCleanup();

   return CWinApp::ExitInstance();
}
```

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::Run
CWinApp::InitInstance

Other Resources

CWinApp Members

CWinApp::GetFirstDocTemplatePosition

Gets the position of the first document template in the application.

```
POSITION GetFirstDocTemplatePosition( ) const;
```

Return Value

A **POSITION** value that can be used for iteration or object pointer retrieval; **NULL** if the list is empty.

Remarks

Use the **POSITION** value returned in a call to GetNextDocTemplate to get the first CDocTemplate object.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::AddDocTemplate
CWinApp::GetNextDocTemplate

Other Resources

CWinApp Members

CWinApp::GetHelpMode

Retrieves the type of help used by the application.

```
AFX_HELP_TYPE GetHelpMode( );
```

Return Value

The help type used by the application. See CWinApp::m_eHelpType for more information.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::SetHelpMode
CWinApp::EnableHtmlHelp

Other Resources

CWinApp Members

CWinApp::GetNextDocTemplate

Gets the document template identified by *pos*, then sets *pos* to the **POSITION** value.

```
CDocTemplate* GetNextDocTemplate(
   POSITION& pos
) const;
```

Parameters

pos

A reference to a **POSITION** value returned by a previous call to **GetNextDocTemplate** or GetFirstDocTemplatePosition. The value is updated to the next position by this call.

Return Value

A pointer to a CDocTemplate object.

Remarks

You can use **GetNextDocTemplate** in a forward iteration loop if you establish the initial position with a call to **GetFirstDocTemplatePosition**.

You must ensure that your **POSITION** value is valid. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

If the retrieved document template is the last available, then the new value of *pos* is set to **NULL**.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::AddDocTemplate

Other Resources

CWinApp Members

CWinApp::GetPrinterDeviceDefaults

Call this member function to prepare a printer device context for printing.

```
BOOL GetPrinterDeviceDefaults(
   struct tagPDA* pPrintDlg
);
```

Parameters

pPrintDlg

A pointer to a PRINTDLG structure.

Return Value

Nonzero if successful; otherwise 0.

Remarks

Retrieves the current printer defaults from the Windows .INI file as necessary, or uses the last printer configuration set by the user in Print Setup.

Example

```
void CMyApp::SetLandscapeMode()
{
   PRINTDLG pd;
   pd.lStructSize=(DWORD)sizeof(PRINTDLG);
   BOOL bRet=GetPrinterDeviceDefaults(&pd);
   if(bRet)
   {
      // protect memory handle with ::GlobalLock and ::GlobalUnlock
      DEVMODE FAR *pDevMode=(DEVMODE FAR *)::GlobalLock(m_hDevMode);
      // set orientation to landscape
      pDevMode->dmOrientation=DMORIENT_LANDSCAPE;
      ::GlobalUnlock(m_hDevMode);
   }
}
```

See Also

Reference

CWinApp Class
Hierarchy Chart
CPrintDialog Class

Other Resources

CWinApp Members

CWinApp::GetProfileBinary

Call this member function to retrieve binary data from an entry within a specified section of the application's registry or .INI file.

```
BOOL GetProfileBinary(
   LPCTSTR lpszSection,
   LPCTSTR lpszEntry,
   LPBYTE* ppData,
   UINT* pBytes
);
```

Parameters

lpszSection

Points to a null-terminated string that specifies the section containing the entry.

lpszEntry

Points to a null-terminated string that contains the entry whose value is to be retrieved.

ppData

Points to a pointer that will receive the address of the data.

pBytes

Points to a UINT that will receive the size of the data (in bytes).

Return Value

Nonzero if successful; otherwise 0.

Remarks

The entries are stored as follows:

- In Windows NT, the value is stored to a registry key.
- In Windows 3.x, the value is stored in the WIN.INI file.
- In Windows 95/98, the value is stored in a cached version of WIN.INI.

This member function is not case sensitive, so the strings in the *lpszSection* and *lpszEntry* parameters may differ in case.

Note

GetProfileBinary allocates a buffer and returns its address in **ppData*. The caller is responsible for freeing the buffer using **delete []**.

Security Note

The data returned by this function is not necessarily NULL terminated, and the caller must perform validation. For more information, see Avoiding Buffer Overruns.

Example

```
BOOL CMyApp::InitInstance()
{
    // CMyApp is derived from CWinApp.

    const char *pszKey = "MyApp";
    struct complex {
        double re, im;
    } myData = { 1.4142, -0.5 };
```

```
    // Change the registry key under which our settings are stored.

    SetRegistryKey(_T(""));

    // Write the information to the registry.

    WriteProfileBinary(pszKey, "ComplexData", (LPBYTE)&myData, sizeof(myData) );

    // Read the information from the registry.

    complex* pData;
    UINT n;
    BOOL ret = GetProfileBinary(pszKey, "ComplexData", (LPBYTE*)&pData, &n);

    ASSERT(ret);
    ASSERT(n == sizeof(complex));
    ASSERT(myData.re == pData->re);
    ASSERT(myData.im == pData->im);
    delete [] pData; // free the buffer
    return TRUE;
}
```

For an additional example, see CWinApp::WriteProfileBinary.

Smart Device Developer Notes

Only the registry-based version using CWinApp::SetRegistryKey is supported in Windows CE-based projects.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::GetProfileInt
CWinApp::GetProfileString

Other Resources

CWinApp Members

CWinApp::GetProfileInt

Call this member function to retrieve the value of an integer from an entry within a specified section of the application's registry or .INI file.

```
UINT GetProfileInt(
   LPCTSTR lpszSection,
   LPCTSTR lpszEntry,
   int nDefault
);
```

Parameters

lpszSection

Points to a null-terminated string that specifies the section containing the entry.

lpszEntry

Points to a null-terminated string that contains the entry whose value is to be retrieved.

nDefault

Specifies the default value to return if the framework cannot find the entry. This value can be an unsigned value in the range 0 through 65,535 or a signed value in the range –32,768 through 32,767.

Return Value

The integer value of the string that follows the specified entry if the function is successful. The return value is the value of the *nDefault* parameter if the function does not find the entry. The return value is 0 if the value that corresponds to the specified entry is not an integer.

This member function supports hexadecimal notation for the value in the .INI file. When you retrieve a signed integer, you should cast the value into an **int**.

Remarks

The entries are stored as follows:

- In Windows NT, the value is stored to a registry key.
- In Windows 3.x, the value is stored in the WIN.INI file.
- In Windows 95/98, the value is stored in a cached version of WIN.INI.

This member function is not case sensitive, so the strings in the *lpszSection* and *lpszEntry* parameters may differ in case.

Security Note

The data returned by this function is not necessarily NULL terminated, and the caller must perform validation. For more information, see Avoiding Buffer Overruns.

Example

```
BOOL CMyApp::InitInstance()
{
    // CMyApp is derived from CWinApp.

    const char *pszKey = "MyApp";
    const char *pszName = "Julian";
    int iAge = 26;

    // Change the registry key under which our settings are stored.

    SetRegistryKey(_T(""));

    // Write the information to the registry.
```

```
    WriteProfileString(pszKey, "Name", pszName);
    WriteProfileInt(pszKey, "Age", iAge);

    // Read the information from the registry.

    CString strName = GetProfileString(pszKey, "Name");
    int iAge2 = GetProfileInt(pszKey, "Age", 0);

    ASSERT(strName == pszName);
    ASSERT(iAge2 == iAge);

    return TRUE;
}
```

For an additional example, see CWinApp::WriteProfileInt.

Smart Device Developer Notes

Only the registry-based version using CWinApp::SetRegistryKey is supported in Windows CE-based projects.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::GetProfileString
CWinApp::WriteProfileInt
GetPrivateProfileInt

Other Resources

CWinApp Members

CWinApp::GetProfileString

Call this member function to retrieve the string associated with an entry within the specified section in the application's registry or .INI file.

```
CString GetProfileString(
   LPCTSTR lpszSection,
   LPCTSTR lpszEntry,
   LPCTSTR lpszDefault = NULL
);
```

Parameters

lpszSection

Points to a null-terminated string that specifies the section containing the entry.

lpszEntry

Points to a null-terminated string that contains the entry whose string is to be retrieved. This value must not be **NULL**.

lpszDefault

Points to the default string value for the given entry if the entry cannot be found in the initialization file.

Return Value

The return value is the string from the application's .INI file or *lpszDefault* if the string cannot be found. The maximum string length supported by the framework is **_MAX_PATH**. If *lpszDefault* is **NULL**, the return value is an empty string.

Remarks

The entries are stored as follows:

- In Windows NT, the value is stored to a registry key.
- In Windows 3.x, the value is stored in the WIN.INI file.
- In Windows 95/98, the value is stored in a cached version of WIN.INI.

Security Note

The data returned by this function is not necessarily NULL terminated, and the caller must perform validation. For more information, see Avoiding Buffer Overruns.

Example

```
CString strSection       = "My Section";
CString strStringItem    = "My String Item";
CString strIntItem       = "My Int Item";

CWinApp* pApp = AfxGetApp();

pApp->WriteProfileString(strSection, strStringItem, "test");

CString strValue;
strValue = pApp->GetProfileString(strSection, strStringItem);
ASSERT(strValue == "test");

pApp->WriteProfileInt(strSection, strIntItem, 1234);
int nValue;
nValue = pApp->GetProfileInt(strSection, strIntItem, 0);
ASSERT(nValue == 1234);
```

For another example, see the example for CWinApp::GetProfileInt.

Smart Device Developer Notes

Only the registry-based version using CWinApp::SetRegistryKey is supported in Windows CE-based projects.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::GetProfileInt
CWinApp::WriteProfileString
GetPrivateProfileString

Other Resources

CWinApp Members

CWinApp::HideApplication

Call this member function to hide an application before closing the open documents.

```
void HideApplication( );
```

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::CloseAllDocuments

Other Resources

CWinApp Members

CWinApp::HtmlHelp

Call this member function to invoke the HTMLHelp application.

```
virtual void HtmlHelp(
   DWORD_PTR dwData,
   UINT nCmd = 0x000F
);
```

Parameters

dwData

Specifies additional data. The value used depends on the value of the *nCmd* parameter.

nCmd

Specifies the type of help requested. For a list of possible values and how they affect the *dwData* parameter, see the *uCommand* parameter described in About the HTMLHelp API Function in the *Platform SDK*.

Remarks

The framework also calls this function to invoke the HTMLHelp application.

The framework will automatically close the HTMLHelp application when your application terminates.

See Also

Tasks

Displaying F1 Help for a Dialog Box or Menu Option

Reference

CWinApp Class
Hierarchy Chart
CWinApp::WinHelp
CWinApp::m_bUseHtmlHelp

Other Resources

CWinApp Members

CWinApp::InitInstance

Windows allows several copies of the same program to run at the same time.

```
virtual BOOL InitInstance( );
```

Return Value

Nonzero if initialization is successful; otherwise 0.

Remarks

Application initialization is conceptually divided into two sections: one-time application initialization that is done the first time the program runs, and instance initialization that runs each time a copy of the program runs, including the first time. The framework's implementation of **WinMain** calls this function.

Override **InitInstance** to initialize each new instance of your application running under Windows. Typically, you override **InitInstance** to construct your main window object and set the **CWinThread::m_pMainWnd** data member to point to that window. For more information on overriding this member function, see CWinApp: The Application Class.

Note

MFC applications must be initialized as single threaded apartment (STA). If you call CoInitializeEx in your **InitInstance** override, specify **COINIT_APARTMENTTHREADED** (rather than **COINIT_MULTITHREADED**). For more information, see PRB: MFC Application Stops Responding When You Initialize the Application as a Multithreaded Apartment (828643) at http://support.microsoft.com/default.aspx?scid=kb;en-us;828643.

Example

```
// AppWizard implements the InitInstance overridable function
// according to options you select.  For example, the single document
// interface (SDI) option was chosen for the AppWizard code created
// below. You can add other per-instance initializations to the code
// created by AppWizard.

BOOL CMyApp::InitInstance()
{
   // Standard initialization
   // If you are not using these features and wish to reduce the size
   //  of your final executable, you should remove from the following
   //  the specific initialization routines you do not need.

   LoadStdProfileSettings();  // Load standard INI file options (including MRU)

   // Register the application's document templates.  Document templates
   //  serve as the connection between documents, frame windows and views.

   CSingleDocTemplate* pDocTemplate;
   pDocTemplate = new CSingleDocTemplate(
      IDR_MAINFRAME,
      RUNTIME_CLASS(CMyDoc),
      RUNTIME_CLASS(CMainFrame),     // main SDI frame window
      RUNTIME_CLASS(CMyView));
   AddDocTemplate(pDocTemplate);

   // create a new (empty) document
   OnFileNew();

   if (m_lpCmdLine[0] != '\0')
   {
      // TODO: add command line processing here
   }

   return TRUE;
```

```
}
```

See Also

Reference

CWinApp Class

Hierarchy Chart

Other Resources

CWinApp Members

CWinApp::LoadCursor

Loads the cursor resource named by *lpszResourceName* or specified by *nIDResource* from the current executable file.

```
HCURSOR LoadCursor(
   LPCTSTR lpszResourceName
) const;
HCURSOR LoadCursor(
   UINT nIDResource
) const;
```

Parameters

lpszResourceName

Points to a null-terminated string that contains the name of the cursor resource. You can use a **CString** for this argument.

nIDResource

ID of the cursor resource. For a list of resources, see LoadCursor in the *Platform SDK*.

Return Value

A handle to a cursor if successful; otherwise **NULL**.

Remarks

LoadCursor loads the cursor into memory only if it has not been previously loaded; otherwise, it retrieves a handle of the existing resource.

Use the LoadStandardCursor or LoadOEMCursor member function to access the predefined Windows cursors.

Example

```
HCURSOR hCursor;

   // Load a cursor resource that was originally created using
   // the Graphics Editor and assigned the i.d. IDC_MYCURSOR.
   hCursor = AfxGetApp()->LoadCursor(IDC_MYCURSOR);
```

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::LoadStandardCursor
CWinApp::LoadOEMCursor
LoadCursor

Other Resources

CWinApp Members

CWinApp::LoadIcon

Loads the icon resource named by *lpszResourceName* or specified by *nIDResource* from the executable file.

```
HICON LoadIcon(
   LPCTSTR lpszResourceName
) const;
HICON LoadIcon(
   UINT nIDResource
) const;
```

Parameters

lpszResourceName

Points to a null-terminated string that contains the name of the icon resource. You can also use a **CString** for this argument.

nIDResource

ID number of the icon resource.

Return Value

A handle to an icon if successful; otherwise **NULL**.

Remarks

LoadIcon loads the icon only if it has not been previously loaded; otherwise, it retrieves a handle of the existing resource.

You can use the LoadStandardIcon or LoadOEMIcon member function to access the predefined Windows icons.

Note
This member function calls the Win32 API function LoadIcon, which can only load an icon whose size conforms to the **SM_CXICON** and **SM_CYICON** system metric values.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::LoadStandardIcon
CWinApp::LoadOEMIcon
LoadIcon

Other Resources

CWinApp Members

CWinApp::LoadOEMCursor

Loads the Windows predefined cursor resource specified by *nIDCursor*.

```
HCURSOR LoadOEMCursor(
   UINT nIDCursor
) const;
```

Parameters

nIDCursor

An **OCR_** manifest constant identifier that specifies a predefined Windows cursor. You must have **#define OEMRESOURCE** before **#include <afxwin.h>** to gain access to the **OCR_** constants in WINDOWS.H.

Return Value

A handle to a cursor if successful; otherwise **NULL**.

Remarks

Use the **LoadOEMCursor** or LoadStandardCursor member function to access the predefined Windows cursors.

Example

```
// In the stdafx.h file, add #define OEMRESOURCE to
// include the windows.h definitions of OCR_ values.
#define OEMRESOURCE
#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions (including VB)

   HCURSOR hCursor;
   // Load the predefined WIndows "size all" cursor.
   hCursor = AfxGetApp()->LoadOEMCursor(OCR_SIZEALL);
```

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::LoadCursor
CWinApp::LoadStandardCursor
LoadCursor

Other Resources

CWinApp Members

CWinApp::LoadOEMIcon

Loads the Windows predefined icon resource specified by *nIDIcon*.

```
HICON LoadOEMIcon(
   UINT nIDIcon
) const;
```

Parameters

nIDIcon

An **OIC_** manifest constant identifier that specifies a predefined Windows icon. You must have **#define OEMRESOURCE** before **#include <afxwin.h>** to access the **OIC_** constants in WINDOWS.H.

Return Value

A handle to an icon if successful; otherwise **NULL**.

Remarks

Use the **LoadOEMIcon** or LoadStandardIcon member function to access the predefined Windows icons.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::LoadStandardIcon
CWinApp::LoadIcon
LoadIcon

Other Resources

CWinApp Members

CWinApp::LoadStandardCursor

Loads the Windows predefined cursor resource that *lpszCursorName* specifies.

```
HCURSOR LoadStandardCursor(
   LPCTSTR lpszCursorName
) const;
```

Parameters

lpszCursorName

An **IDC_** manifest constant identifier that specifies a predefined Windows cursor. These identifiers are defined in WINDOWS.H. The following list shows the possible predefined values and meanings for *lpszCursorName*:

- **IDC_ARROW** Standard arrow cursor
- **IDC_IBEAM** Standard text-insertion cursor
- **IDC_WAIT** Hourglass cursor used when Windows performs a time-consuming task
- **IDC_CROSS** Cross-hair cursor for selection
- **IDC_UPARROW** Arrow that points straight up
- **IDC_SIZE** Obsolete and unsupported; use **IDC_SIZEALL**
- **IDC_SIZEALL** A four-pointed arrow. The cursor to use to resize a window.
- **IDC_ICON** Obsolete and unsupported. Use **IDC_ARROW**.
- **IDC_SIZENWSE** Two-headed arrow with ends at upper left and lower right
- **IDC_SIZENESW** Two-headed arrow with ends at upper right and lower left
- **IDC_SIZEWE** Horizontal two-headed arrow
- **IDC_SIZENS** Vertical two-headed arrow

Return Value

A handle to a cursor if successful; otherwise **NULL**.

Remarks

Use the **LoadStandardCursor** or LoadOEMCursor member function to access the predefined Windows cursors.

Example

```
HCURSOR hCursor;

   // Load the predefined Windows "up arrow" cursor.
   hCursor = AfxGetApp()->LoadStandardCursor(IDC_UPARROW);
```

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::LoadOEMCursor
CWinApp::LoadCursor
LoadCursor

Other Resources

CWinApp Members

CWinApp::LoadStandardIcon

Loads the Windows predefined icon resource that *lpszIconName* specifies.

```
HICON LoadStandardIcon(
   LPCTSTR lpszIconName
) const;
```

Parameters

lpszIconName

A manifest constant identifier that specifies a predefined Windows icon. These identifiers are defined in WINDOWS.H. For a list of the possible predefined values and their descriptions, see the *lpIconName* parameter in LoadIcon in the *Platform SDK*.

Return Value

A handle to an icon if successful; otherwise **NULL**.

Remarks

Use the **LoadStandardIcon** or LoadOEMIcon member function to access the predefined Windows icons.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::LoadOEMIcon
CWinApp::LoadIcon
LoadIcon

Other Resources

CWinApp Members

CWinApp::LoadStdProfileSettings

Call this member function from within the InitInstance member function to enable and load the list of most recently used (MRU) files and last preview state.

```
void LoadStdProfileSettings(
   UINT nMaxMRU = _AFX_MRU_COUNT
);
```

Parameters

nMaxMRU

The number of recently used files to track.

Remarks

If *nMaxMRU* is 0, no MRU list will be maintained.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::OnFileOpen
CWinApp::AddToRecentFileList

Other Resources

CWinApp Members

CWinApp::OnContextHelp

Handles SHIFT+F1 Help within the application.

```
afx_msg void OnContextHelp( );
```

Remarks

You must add an `ON_COMMAND( ID_CONTEXT_HELP, OnContextHelp )` statement to your **CWinApp** class message map and also add an accelerator table entry, typically SHIFT+F1, to enable this member function.

OnContextHelp puts the application into Help mode. The cursor changes to an arrow and a question mark, and the user can then move the mouse pointer and press the left mouse button to select a dialog box, window, menu, or command button. This member function retrieves the Help context of the object under the cursor and calls the Windows function WinHelp with that Help context.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::OnHelp
CWinApp::WinHelp

Other Resources

CWinApp Members

CWinApp::OnDDECommand

Called by the framework when the main frame window receives a DDE execute message.

```
virtual BOOL OnDDECommand(
   LPTSTR lpszCommand
);
```

Parameters

lpszCommand

Points to a DDE command string received by the application.

Return Value

Nonzero if the command is handled; otherwise 0.

Remarks

The default implementation checks whether the command is a request to open a document and, if so, opens the specified document. The Windows File Manager usually sends such DDE command strings when the user double-clicks a data file. Override this function to handle other DDE execute commands, such as the command to print.

Example

```
BOOL CMyApp::OnDDECommand(LPTSTR lpszCommand)
{
   if (CWinApp::OnDDECommand(lpszCommand))
      return TRUE;

   // Handle any DDE commands recognized by your application
   // and return TRUE.  See implementation of CWinApp::OnDDEComand
   // for example of parsing the DDE command string.

   // Return FALSE for any DDE commands you do not handle.
   return FALSE;
 }
```

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::EnableShellOpen

Other Resources

CWinApp Members

CWinApp::OnFileNew

Implements the **ID_FILE_NEW** command.

```
afx_msg void OnFileNew( );
```

Remarks

You must add an `ON_COMMAND( ID_FILE_NEW, OnFileNew )` statement to your **CWinApp** class message map to enable this member function. If enabled, this function handles execution of the File New command.

See Technical Note 22 for information on default behavior and guidance on how to override this member function.

Example

```
// The following message map, produced by the Application Wizard, binds
// the File New, Open, and Print Setup menu commands to default
// framework implementations of these commands.
BEGIN_MESSAGE_MAP(CMyApp, CWinApp)
   ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
   // Standard file based document commands
   ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
   ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
   // Standard print setup command
   ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

// The following message map illustrates how to rebind the
// File New, Open and Print Setup menu commands to handlers that
// you implement in your CWinApp-derived class.
// Note, you can name the handler CMyApp::OnFileNew instead of
// CMyApp::OnMyFileNew, and likewise for the other handlers, if desired.
BEGIN_MESSAGE_MAP(CMyApp, CWinApp)
   ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
   ON_COMMAND(ID_FILE_NEW, OnMyFileNew)
   ON_COMMAND(ID_FILE_OPEN, OnMyFileOpen)
   ON_COMMAND(ID_FILE_PRINT_SETUP, OnMyFilePrintSetup)
END_MESSAGE_MAP()
```

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::OnFileOpen

Other Resources

CWinApp Members

CWinApp::OnFileOpen

Implements the **ID_FILE_OPEN** command.

```
afx_msg void OnFileOpen( );
```

Remarks

You must add an `ON_COMMAND( ID_FILE_OPEN, OnFileOpen )` statement to your **CWinApp** class message map to enable this member function. If enabled, this function handles execution of the File Open command.

For information on default behavior and guidance on how to override this member function, see Technical Note 22.

Example

```
// The following message map, produced by the Application Wizard, binds
// the File New, Open, and Print Setup menu commands to default
// framework implementations of these commands.
BEGIN_MESSAGE_MAP(CMyApp, CWinApp)
   ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
   // Standard file based document commands
   ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
   ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
   // Standard print setup command
   ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

// The following message map illustrates how to rebind the
// File New, Open and Print Setup menu commands to handlers that
// you implement in your CWinApp-derived class.
// Note, you can name the handler CMyApp::OnFileNew instead of
// CMyApp::OnMyFileNew, and likewise for the other handlers, if desired.
//
BEGIN_MESSAGE_MAP(CMyApp, CWinApp)
   ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
   ON_COMMAND(ID_FILE_NEW, OnMyFileNew)
   ON_COMMAND(ID_FILE_OPEN, OnMyFileOpen)
   ON_COMMAND(ID_FILE_PRINT_SETUP, OnMyFilePrintSetup)
END_MESSAGE_MAP()
```

See Also

Reference

CWinApp Class

Hierarchy Chart

CWinApp::OnFileNew

Other Resources

CWinApp Members

CWinApp::OnFilePrintSetup

Implements the **ID_FILE_PRINT_SETUP** command.

```
afx_msg void OnFilePrintSetup( );
```

Remarks

You must add an `ON_COMMAND( ID_FILE_PRINT_SETUP, OnFilePrintSetup )` statement to your **CWinApp** class message map to enable this member function. If enabled, this function handles execution of the File Print command.

For information on default behavior and guidance on how to override this member function, see Technical Note 22.

Example

```
// The following message map, produced by the Application Wizard, binds
// the File New, Open, and Print Setup menu commands to default
// framework implementations of these commands.
BEGIN_MESSAGE_MAP(CMyApp, CWinApp)
   ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
   // Standard file based document commands
   ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
   ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
   // Standard print setup command
   ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

// The following message map illustrates how to rebind the
// File New, Open and Print Setup menu commands to handlers that
// you implement in your CWinApp-derived class.
// Note, you can name the handler CMyApp::OnFileNew instead of
// CMyApp::OnMyFileNew, and likewise for the other handlers, if desired.
//
BEGIN_MESSAGE_MAP(CMyApp, CWinApp)
   ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
   ON_COMMAND(ID_FILE_NEW, OnMyFileNew)
   ON_COMMAND(ID_FILE_OPEN, OnMyFileOpen)
   ON_COMMAND(ID_FILE_PRINT_SETUP, OnMyFilePrintSetup)
END_MESSAGE_MAP()
```

See Also

Reference

CWinApp Class

Hierarchy Chart

CWinApp::OnFileNew

Other Resources

CWinApp Members

CWinApp::OnHelp

Handles F1 Help within the application (using the current context).

```
afx_msg void OnHelp( );
```

Remarks

Usually you will also add an accelerator-key entry for the F1 key. Enabling the F1 key is only a convention, not a requirement.

You must add an `ON_COMMAND( ID_HELP, OnHelp )` statement to your **CWinApp** class message map to enable this member function. If enabled, called by the framework when the user presses the F1 key.

The default implementation of this message-handler function determines the Help context that corresponds to the current window, dialog box, or menu item and then calls WINHELP.EXE. If no context is currently available, the function uses the default context.

Override this member function to set the Help context to something other than the window, dialog box, menu item, or toolbar button that currently has the focus. Call **WinHelp** with the desired Help context ID.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::OnContextHelp
CWinApp::OnHelpUsing
CWinApp::OnHelpIndex
CWinApp::WinHelp

Other Resources

CWinApp Members

CWinApp::OnHelpFinder

Handles the **ID_HELP_FINDER** and **ID_DEFAULT_HELP** commands.

```
afx_msg void OnHelpFinder( );
```

Remarks

You must add an `ON_COMMAND( ID_HELP_FINDER, OnHelpFinder )` statement to your **CWinApp** class message map to enable this member function. If enabled, the framework calls this message-handler function when the user of your application selects the Help Finder command to invoke **WinHelp** with the standard **HELP_FINDER** topic.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::OnHelp
CWinApp::OnHelpUsing
CWinApp::WinHelp
CWinApp::OnHelpIndex

Other Resources

CWinApp Members

CWinApp::OnHelpIndex

Handles the **ID_HELP_INDEX** command and provides a default Help topic.

```
afx_msg void OnHelpIndex( );
```

Remarks

You must add an `ON_COMMAND( ID_HELP_INDEX, OnHelpIndex )` statement to your **CWinApp** class message map to enable this member function. If enabled, the framework calls this message-handler function when the user of your application selects the Help Index command to invoke **WinHelp** with the standard **HELP_INDEX** topic.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::OnHelp
CWinApp::OnHelpUsing
CWinApp::WinHelp

Other Resources

CWinApp Members

CWinApp::OnHelpUsing

Handles the **ID_HELP_USING** command.

```
afx_msg void OnHelpUsing( );
```

Remarks

You must add an `ON_COMMAND( ID_HELP_USING, OnHelpUsing )` statement to your **CWinApp** class message map to enable this member function. The framework calls this message-handler function when the user of your application selects the Help Using command to invoke the **WinHelp** application with the standard **HELP_HELPONHELP** topic.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::OnHelp
CWinApp::OnHelpIndex
CWinApp::WinHelp

Other Resources

CWinApp Members

CWinApp::OnIdle

Override this member function to perform idle-time processing.

```
virtual BOOL OnIdle(
   LONG lCount
);
```

Parameters

lCount

A counter incremented each time **OnIdle** is called when the application's message queue is empty. This count is reset to 0 each time a new message is processed. You can use the *lCount* parameter to determine the relative length of time the application has been idle without processing a message.

Return Value

Nonzero to receive more idle processing time; 0 if no more idle time is needed.

Remarks

OnIdle is called in the default message loop when the application's message queue is empty. Use your override to call your own background idle-handler tasks.

OnIdle should return 0 to indicate that no idle processing time is required. The *lCount* parameter is incremented each time **OnIdle** is called when the message queue is empty and resets to 0 each time a new message is processed. You can call your different idle routines based on this count.

The following summarizes idle loop processing:

1. If the message loop in the Microsoft Foundation Class Library checks the message queue and finds no pending messages, it calls `OnIdle` for the application object and supplies 0 as the *lCount* argument.
2. `OnIdle` performs some processing and returns a nonzero value to indicate it should be called again to do further processing.
3. The message loop checks the message queue again. If no messages are pending, it calls `OnIdle` again, incrementing the *lCount* argument.
4. Eventually, `OnIdle` finishes processing all its idle tasks and returns 0. This tells the message loop to stop calling `OnIdle` until the next message is received from the message queue, at which point the idle cycle restarts with the argument set to 0.

Do not perform lengthy tasks during **OnIdle** because your application cannot process user input until **OnIdle** returns.

Note

The default implementation of **OnIdle** updates command user-interface objects such as menu items and toolbar buttons, and it performs internal data structure cleanup. Therefore, if you override **OnIdle**, you must call **CWinApp::OnIdle** with the *lCount* in your overridden version. First call all base-class idle processing (that is, until the base class **OnIdle** returns 0). If you need to perform work before the base-class processing completes, review the base-class implementation to select the proper *lCount* during which to do your work.

If you do not want **OnIdle** to be called whenever a message is retrieved from the message queue, you can override the CWinThreadIsIdleMessage. If an application has set a very short timer, or if the system is sending the **WM_SYSTIMER** message, then **OnIdle** will be called repeatedly, and degrade performance.

Example

The following two examples show how to use **OnIdle**. The first example processes two idle tasks using the *lCount* argument to prioritize the tasks. The first task is high priority, and you should do it whenever possible. The second task is less important and should be done only when there is a long pause in user input. Note the call to the base-class version of **OnIdle**. The second example manages a group of idle tasks with different priorities.

```
BOOL CMyApp::OnIdle(LONG lCount)
{
    BOOL bMore = CWinApp::OnIdle(lCount);

    if (lCount == 0)
    {
    TRACE("App idle for short period of time\n");
    bMore = TRUE;
    }
    else if (lCount == 10)
    {
    TRACE("App idle for longer amount of time\n");
        bMore = TRUE;
    }
    else if (lCount == 100)
    {
        TRACE("App idle for even longer amount of time\n");
        bMore = TRUE;
    }
    else if (lCount == 1000)
    {
        TRACE("App idle for quite a long period of time\n");
     // bMore is not set to TRUE, no longer need idle
     // IMPORTANT: bMore is not set to FALSE since CWinApp::OnIdle may
     // have more idle tasks to complete.
    }

    return bMore;
     // return TRUE as long as there is any more idle tasks
}
```

Second Example

```
// In this example, four idle loop tasks are given various
// opportunities to run:
// Task1 is always given a chance to run during idle time, provided
//   that no message has queued up while the framework was processing
//   its own idle loop tasks (at lCount levels 0 and 1).
// Task2 is given a chance to run only if Task1 has already run,
//   provided that no message has queued up while Task1 was running.
// Task3 and Task4 are given a chance to run only if both Task1 and
//   Task2 have already run, and no message has queued up in the mean
//   time.  If Task3 gets its chance to run, then Task4 always gets
//   a chance to run immediately after Task3.

BOOL CMyApp::OnIdle(LONG lCount)
{
   // In this example, as in most applications, you should let the
   // base class CWinApp::OnIdle complete its processing before you
   // attempt any additional idle loop processing.
   if (CWinApp::OnIdle(lCount))
      return TRUE;

   // The base class CWinApp::OnIdle reserves the lCount values 0
   // and 1 for the framework's own idle processing.   If you wish to
   // share idle processing time at a peer level with the framework,
   // then replace the above if-statement with a straight call to
   // CWinApp::OnIdle; and then add a case statement for lCount value
   // 0 and/or 1. Study the base class implementation first to
   // understand how your idle loop tasks will compete with the
   // framework's idle loop processing.

   switch (lCount)
   {
      case 2:
         Task1();
         return TRUE; // next time give Task2 a chance
```

```
        case 3:
           Task2();
           return TRUE; // next time give Task3 and Task4 a chance
        case 4:
           Task3();
           Task4();
           return FALSE; // cycle through the idle loop tasks again
     }
     return FALSE;
  }
```

See Also

Reference

CWinApp Class

Hierarchy Chart

Other Resources

CWinApp Members

CWinApp::OpenDocumentFile

The framework calls this member function to open the named CDocument file for the application.

```
virtual CDocument* OpenDocumentFile(
   LPCTSTR lpszFileName
);
```

Parameters

lpszFileName

The name of the file to be opened.

Return Value

A pointer to a **CDocument** if successful; otherwise **NULL**.

Remarks

If a document with that name is already open, the first frame window that contains that document will be activated. If an application supports multiple document templates, the framework uses file extension to find the appropriate document template to attempt to load the document. If successful, the document template then creates a frame window and view for the document.

Example

```
BOOL CMyApp::InitInstance()
{
   // ...
   if (m_lpCmdLine[0] == '\0')
   {
      // Create a new (empty) document.
      OnFileNew();
   }
   else
   {
      // Open a file passed as the first command line parameter.
      OpenDocumentFile(m_lpCmdLine);
   }

   // ...
}
```

See Also

Reference

CWinApp Class

Hierarchy Chart

Other Resources

CWinApp Members

CWinApp::ParseCommandLine

Call this member function to parse the command line and send the parameters, one at a time, to CCommandLineInfo::ParseParam.

```
void ParseCommandLine(
   CCommandLineInfo& rCmdInfo
);
```

Parameters

rCmdInfo

A reference to a CCommandLineInfo object.

Remarks

When you start a new MFC project using the Application Wizard, the Application Wizard will create a local instance of **CCommandLineInfo**, and then call **ProcessShellCommand** and **ParseCommandLine** in the InitInstance member function. A command line follows the route described below:

1. After being created in **InitInstance**, the **CCommandLineInfo** object is passed to **ParseCommandLine**.
2. **ParseCommandLine** then calls **CCommandLineInfo::ParseParam** repeatedly, once for each parameter.
3. **ParseParam** fills the **CCommandLineInfo** object, which is then passed to ProcessShellCommand.
4. **ProcessShellCommand** handles the command-line arguments and flags.

Note that you can call **ParseCommandLine** directly as needed.

For a description of the command-line flags, see CCommandLineInfo::m_nShellCommand.

See Also

Reference

CWinApp Class
Hierarchy Chart
CCommandLineInfo Class
CWinApp::InitInstance
CCommandLineInfo::ParseParam
CWinApp::ProcessShellCommand
CCommandLineInfo::m_nShellCommand

Other Resources

CWinApp Members

CWinApp::PreTranslateMessage

Override this function to filter window messages before they are dispatched to the Windows functions TranslateMessage and DispatchMessage The default implementation performs accelerator-key translation, so you must call the **CWinApp::PreTranslateMessage** member function in your overridden version.

```
virtual BOOL PreTranslateMessage(
   MSG* pMsg
);
```

Parameters

pMsg

A pointer to a MSG structure that contains the message to process.

Return Value

Nonzero if the message was fully processed in **PreTranslateMessage** and should not be processed further. Zero if the message should be processed in the normal way.

See Also

Reference

CWinApp Class
Hierarchy Chart
DispatchMessage
TranslateMessage

Other Resources

CWinApp Members

CWinApp::ProcessMessageFilter

The framework's hook function calls this member function to filter and respond to certain Windows messages.

```
virtual BOOL ProcessMessageFilter(
   int code,
   LPMSG lpMsg
);
```

Parameters

code

Specifies a hook code. This member function uses the code to determine how to process *lpMsg*.

lpMsg

A pointer to a Windows MSG structure.

Return Value

Nonzero if the message is processed; otherwise 0.

Remarks

A hook function processes events before they are sent to the application's normal message processing.

If you override this advanced feature, be sure to call the base-class version to maintain the framework's hook processing.

See Also

Reference

CWinApp Class
Hierarchy Chart
MessageProc
About Hooks

Other Resources

CWinApp Members

CWinApp::ProcessShellCommand

This member function is called by InitInstance to accept the parameters passed from the **CCommandLineInfo** object identified by *rCmdInfo*, and perform the indicated action.

```
BOOL ProcessShellCommand(
   CCommandLineInfo& rCmdInfo
);
```

Parameters

rCmdInfo

A reference to a CCommandLineInfo object.

Return Value

Nonzero if the shell command is processed successfully. If 0, return **FALSE** from InitInstance.

Remarks

When you start a new MFC project using the Application Wizard, the Application Wizard will create a local instance of **CCommandLineInfo**, and then call **ProcessShellCommand** and ParseCommandLine in the **InitInstance** member function. A command line follows the route described below:

1. After being created in **InitInstance**, the **CCommandLineInfo** object is passed to **ParseCommandLine**.
2. **ParseCommandLine** then calls CCommandLineInfo::ParseParam repeatedly, once for each parameter.
3. **ParseParam** fills the **CCommandLineInfo** object, which is then passed to **ProcessShellCommand**.
4. **ProcessShellCommand** handles the command-line arguments and flags.

The data members of the **CCommandLineInfo** object, identified by CCommandLineInfo::m_nShellCommand, are of the following enumerated type, which is defined within the **CCommandLineInfo** class.

```
enum{
   FileNew,
   FileOpen,
   FilePrint,
   FilePrintTo,
   FileDDE,
};
```

For a brief description of each of these values, see **CCommandLineInfo::m_nShellCommand**.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::ParseCommandLine
CCommandLineInfo Class
CCommandLineInfo::ParseParam
CCommandLineInfo::m_nShellCommand

Other Resources

CWinApp Members

CWinApp::ProcessWndProcException

The framework calls this member function whenever the handler does not catch an exception thrown in one of your application's message or command handlers.

```
virtual LRESULT ProcessWndProcException(
   CException* e,
   const MSG* pMsg
);
```

Parameters

e

A pointer to an uncaught exception.

pMsg

A MSG structure that contains information about the windows message that caused the framework to throw an exception.

Return Value

The value that should be returned to Windows. Normally this is 0L for windows messages, 1L (**TRUE**) for command messages.

Remarks

Do not call this member function directly.

The default implementation of this member function creates a message box. If the uncaught exception originates with a menu, toolbar, or accelerator command failure, the message box displays a "Command failed" message; otherwise, it displays an "Internal application error" message.

Override this member function to provide global handling of your exceptions. Only call the base functionality if you wish the message box to be displayed.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWnd::WindowProc
CException Class

Other Resources

CWinApp Members

CWinApp::Register

Performs any registration tasks not handled by **RegisterShellFileTypes**.

```
virtual BOOL Register( );
```

Return Value

Nonzero on success; otherwise 0.

Remarks

The default implementation simply returns TRUE. Override this function to provide any customized registration steps.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::RegisterShellFileTypes

Other Resources

CWinApp Members

CWinApp::RegisterShellFileTypes

Call this member function to register all of your application's document types with the Windows File Manager.

```
void RegisterShellFileTypes(
   BOOL bCompat = FALSE
);
```

Parameters

bCompat

TRUE adds registration entries for shell commands Print and Print To, allowing a user to print files directly from the shell, or by dragging the file to a printer object. It also adds a DefaultIcon key. By default, this parameter is **FALSE** for backward compatibility.

Remarks

This allows the user to open a data file created by your application by double-clicking it from within File Manager. Call **RegisterShellFileTypes** after you call AddDocTemplate for each of the document templates in your application. Also call the EnableShellOpen member function when you call **RegisterShellFileTypes**.

RegisterShellFileTypes iterates through the list of CDocTemplate objects that the application maintains and, for each document template, adds entries to the registration database that Windows maintains for file associations. File Manager uses these entries to open a data file when the user double-clicks it. This eliminates the need to ship a .REG file with your application.

If the registration database already associates a given filename extension with another file type, no new association is created. See the **CDocTemplate** class for the format of strings necessary to register this information.

See Also

Reference

CWinApp Class
Hierarchy Chart
CDocTemplate Class
CWinApp::EnableShellOpen
CWinApp::AddDocTemplate

Other Resources

CWinApp Members

CWinApp::Run

Provides a default message loop.

```
virtual int Run( );
```

Return Value

An **int** value that is returned by **WinMain**.

Remarks

Run acquires and dispatches Windows messages until the application receives a **WM_QUIT** message. If the application's message queue currently contains no messages, **Run** calls OnIdle to perform idle-time processing. Incoming messages go to the PreTranslateMessage member function for special processing and then to the Windows function **TranslateMessage** for standard keyboard translation; finally, the **DispatchMessage** Windows function is called.

Run is rarely overridden, but you can override it to provide special behavior.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::PreTranslateMessage
WM_QUIT
DispatchMessage
TranslateMessage

Other Resources

CWinApp Members

CWinApp::RunAutomated

Call this function to determine whether the "**/Automation**" or "**-Automation**" option is present, which indicates whether the server application was launched by a client application.

```
BOOL RunAutomated( );
```

Return Value

Nonzero if the option was found; otherwise 0.

Remarks

If present, the option is removed from the command line. For more information on OLE Automation, see the article Automation Servers.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::RunEmbedded

Other Resources

CWinApp Members

CWinApp::RunEmbedded

Call this function to determine whether the "**/Embedding**" or "**-Embedding**" option is present, which indicates whether the server application was launched by a client application.

```
BOOL RunEmbedded( );
```

Return Value

Nonzero if the option was found; otherwise 0.

Remarks

If present, the option is removed from the command line. For more information on embedding, see the article Servers: Implementing a Server.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::RunAutomated

Other Resources

CWinApp Members

CWinApp::SaveAllModified

Called by the framework to save all documents when the application's main frame window is to be closed, or through a **WM_QUERYENDSESSION** message.

```
virtual BOOL SaveAllModified( );
```

Return Value

Nonzero if safe to terminate the application; 0 if not safe to terminate the application.

Remarks

The default implementation of this member function calls the CDocument::SaveModified member function in turn for all modified documents within the application.

See Also

Reference

CWinApp Class

Hierarchy Chart

Other Resources

CWinApp Members

CWinApp::SelectPrinter

Call this member function to select a specific printer, and release the printer that was previously selected in the Print Dialog box.

```
void SelectPrinter(
   HANDLE hDevNames,
   HANDLE hDevMode,
   BOOL bFreeOld = TRUE
);
```

Parameters

hDevNames

A handle to a DEVNAMES structure that identifies the driver, device, and output port names of a specific printer.

hDevMode

A handle to a DEVMODE structure that specifies information about the device initialization and environment of a printer.

bFreeOld

Frees the previously-selected printer.

Remarks

If both *hDevMode* and *hDevNames* are **NULL**, **SelectPrinter** uses the current default printer.

See Also

Reference

CWinApp Class
Hierarchy Chart
CPrintDialog Class
DEVMODE
DEVNAMES Structure

Other Resources

CWinApp Members

CWinApp::SetHelpMode

Sets the application's help type.

```
void SetHelpMode(
   AFX_HELP_TYPE eHelpType
);
```

Parameters

eHelpType

Specifies the type of help to use. See CWinApp::m_eHelpType for more information.

Remarks

Sets the application's Help type.

To set your application's Help type to HTMLHelp, you can call EnableHTMLHelp. Once you call **EnableHTMLHelp**, your application must use HTMLHelp as its help application. If you want to change to use WinHelp, you can call **SetHelpMode** and set *eHelpType* to **afxWinHelp**.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::GetHelpMode

Other Resources

CWinApp Members

CWinApp::SetRegistryKey

Causes application settings to be stored in the registry instead of INI files.

```
void SetRegistryKey(
   LPCTSTR lpszRegistryKey
);
void SetRegistryKey(
   UINT nIDRegistryKey
);
```

Parameters

lpszRegistryKey

Pointer to a string containing the name of the key.

nIDRegistryKey

ID/index of a key in the registry.

Remarks

This function sets *m_pszRegistryKey*, which is then used by the **GetProfileInt**, **GetProfileString**, **WriteProfileInt**, and **WriteProfileString** member functions of **CWinApp**. If this function has been called, the list of most recently-used (MRU) files is also stored in the registry. The registry key is usually the name of a company. It is stored in a key of the following form: HKEY_CURRENT_USER\Software\<company name>\<application name>\<section name>\<value name>.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::InitInstance
CWinApp::GetProfileInt
CWinApp::GetProfileString
CWinApp::WriteProfileInt
CWinApp::WriteProfileString

Other Resources

CWinApp Members

CWinApp::Unregister

Unregisters all files registered by the application object.

```
virtual BOOL Unregister( );
```

Return Value

Nonzero on success; otherwise 0.

Remarks

The **Unregister** function undoes the registration performed by the application object and the Register function. Normally, both functions are called implicitly by MFC and therefore will not appear in your code.

Override this function to perform custom unregistration steps.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::RegisterShellFileTypes

Other Resources

CWinApp Members

CWinApp::UnregisterShellFileTypes

Call this member function to unregister all of your application's document types with the Windows File Manager.

```
void UnregisterShellFileTypes( );
```

See Also

Reference

CWinApp Class
Hierarchy Chart
CDocTemplate Class
CWinApp::EnableShellOpen
CWinApp::RegisterShellFileTypes

Other Resources

CWinApp Members

CWinApp::WinHelp

Call this member function to invoke the WinHelp application.

```
virtual void WinHelp(
   DWORD_PTR dwData,
   UINT nCmd = HELP_CONTEXT
);
```

Parameters

dwData

Specifies additional data. The value used depends on the value of the *nCmd* parameter.

nCmd

Specifies the type of help requested. For a list of possible values and how they affect the *dwData* parameter, see the WinHelp Windows function.

Remarks

The framework also calls this function to invoke the WinHelp application.

The framework will automatically close the WinHelp application when your application terminates.

Example

```
// Header File: HELPIDS.H
//
// This example header file is #include'd twice:
// (1) It is #include'd by the .CPP file that passes the DWORD
//     context i.d. to CWinApp::WinHelp.
// (2) It is #include'd in the [MAP] section of the .HPJ file,
//     to associate the help context string "HID_MYTOPIC" with
//     the help context numeric i.d., 101.
// The help context string "HID_MYTOPIC" is what identifies the
// help topic in the help .RTF source file, in the "#" footnote:
//     # HID_MYTOPIC
//
// Note, it is not necessary to manage help context id's this way
// for help topics associated with command id's and user interface
// id's defined in your RESOURCE.H file; you should use the MAKEHM
// tool via the custom build rule on your resource.h file to produce
// a help map (.HM) file for these id's.  It is necessary to manage
// help context id's as illustrated here only for help topics not
// associated with command id's or user interface id's.

#define HID_MYTOPIC 101


   // Show the custom help topic that has the context string
   // "HID_MYTOPIC" in the help .RTF file, and which is mapped
   // to the DWORD i.d. HID_MYTOPIC in the above HELPIDS.H file.
   AfxGetApp()->WinHelp(HID_MYTOPIC);


// The following is one line of code in the help map (.HM)
// file produced by the MAKEHM tool, which is called by the custom
// build rule on the resource.h file.  The MAKEHM tool reads the
// following #define in the application's RESOURCE.H file:
//    #define ID_MYCOMMAND 0x08004
// and adds a help id offset value of 0x10000 to create the
// help context DWORD value 0x18004.  See MFC Tech Note 28
// for more information on help id offset values.

HID_MYCOMMAND                           0x18004
```

```
// Rarely will you need to directly call WinHelp yourself
// with the help context i.d. for a command or user interface
// object. The framework will call WinHelp automatically when
// the user, for example, hits F1 when the focus is on a
// My Command menu item. However, if you do want to directly
// call WinHelp for the help topic associated with the command,
// here is how you would do it:

AfxGetApp()->WinHelp(0x10000 + ID_MYCOMMAND);
```

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::OnContextHelp
CWinApp::OnHelpUsing
CWinApp::OnHelp
CWinApp::OnHelpIndex
WinHelp

Other Resources

CWinApp Members

CWinApp::WriteProfileBinary

Call this member function to write binary data into the specified section of the application's registry or .INI file.

```
BOOL WriteProfileBinary(
   LPCTSTR lpszSection,
   LPCTSTR lpszEntry,
   LPBYTE pData,
   UINT nBytes
);
```

Parameters

lpszSection

Points to a null-terminated string that specifies the section containing the entry. If the section does not exist, it is created. The name of the section is case independent; the string may be any combination of uppercase and lowercase letters.

lpszEntry

Points to a null-terminated string that contains the entry into which the value is to be written. If the entry does not exist in the specified section, it is created.

pData

Points to the data to be written.

nBytes

Contains the number of bytes to be written.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The entries are stored as follows:

- In Windows NT, the value is stored to a registry key.
- In Windows 3.x, the value is stored in the WIN.INI file.
- In Windows 95/98, the value is stored in a cached version of WIN.INI.

Example

This example uses `CWinApp* pApp = AfxGetApp();` to get at the CWinApp class illustrating a way that **WriteProfileBinary** and **GetProfileBinary** can be used from any function in an MFC application.

```
CString strSection = "My Section";
CString strItem = "My Binary Item";
double myData = 123.456e12;

CWinApp* pApp = AfxGetApp();

pApp->WriteProfileBinary(strSection, strItem, (LPBYTE)&myData, sizeof(myData));
double *pData;
UINT n;
pApp->GetProfileBinary(strSection, strItem, (LPBYTE*)&pData, &n );
ASSERT( n == sizeof(myData) );
ASSERT( myData = *pData );
delete [] pData;  // free the buffer
```

For another example, see the example for CWinApp::GetProfileBinary.

Smart Device Developer Notes

Only the registry-based version using CWinApp::SetRegistryKey is supported in Windows CE-based projects.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::WriteProfileInt
CWinApp::WriteProfileString
CWinApp::SetRegistryKey

Other Resources

CWinApp Members

CWinApp::WriteProfileInt

Call this member function to write the specified value into the specified section of the application's registry or .INI file.

```
BOOL WriteProfileInt(
   LPCTSTR lpszSection,
   LPCTSTR lpszEntry,
   int nValue
);
```

Parameters

lpszSection

Points to a null-terminated string that specifies the section containing the entry. If the section does not exist, it is created. The name of the section is case independent; the string may be any combination of uppercase and lowercase letters.

lpszEntry

Points to a null-terminated string that contains the entry into which the value is to be written. If the entry does not exist in the specified section, it is created.

nValue

Contains the value to be written.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The entries are stored as follows:

- In Windows NT, the value is stored to a registry key.
- In Windows 3.x, the value is stored in the WIN.INI file.
- In Windows 95/98, the value is stored in a cached version of WIN.INI.

Example

This example uses `CWinApp* pApp = AfxGetApp();` to get at the CWinApp class illustrating a way that **WriteProfileString**, **WriteProfileInt**, **GetProfileString**, and **GetProfileInt** can be used from any function in an MFC application.

```
    CString strSection       = "My Section";
    CString strStringItem    = "My String Item";
    CString strIntItem       = "My Int Item";

    CWinApp* pApp = AfxGetApp();

    pApp->WriteProfileString(strSection, strStringItem, "test");

    CString strValue;
    strValue = pApp->GetProfileString(strSection, strStringItem);
    ASSERT(strValue == "test");

    pApp->WriteProfileInt(strSection, strIntItem, 1234);
    int nValue;
    nValue = pApp->GetProfileInt(strSection, strIntItem, 0);
    ASSERT(nValue == 1234);
```

For another example, see the example for CWinApp::GetProfileInt.

Smart Device Developer Notes

Only the registry-based version using CWinApp::SetRegistryKey is supported in Windows CE-based projects.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::GetProfileInt
CWinApp::WriteProfileString
CWinApp::SetRegistryKey

Other Resources

CWinApp Members

CWinApp::WriteProfileString

Call this member function to write the specified string into the specified section of the application's registry or .INI file.

```
BOOL WriteProfileString(
   LPCTSTR lpszSection,
   LPCTSTR lpszEntry,
   LPCTSTR lpszValue
);
```

Parameters

lpszSection

Points to a null-terminated string that specifies the section containing the entry. If the section does not exist, it is created. The name of the section is case independent; the string may be any combination of uppercase and lowercase letters.

lpszEntry

Points to a null-terminated string that contains the entry into which the value is to be written. If the entry does not exist in the specified section, it is created. If this parameter is **NULL**, the section specified by *lpszSection* is deleted.

lpszValue

Points to the string to be written. If this parameter is **NULL**, the entry specified by the *lpszEntry* parameter is deleted.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The entries are stored as follows:

- In Windows NT, the value is stored to a registry key.
- In Windows 3.x, the value is stored in the WIN.INI file.
- In Windows 95/98, the value is stored in a cached version of WIN.INI.

Example

```
    CString strSection       = "My Section";
    CString strStringItem    = "My String Item";
    CString strIntItem       = "My Int Item";

    CWinApp* pApp = AfxGetApp();

    pApp->WriteProfileString(strSection, strStringItem, "test");

    CString strValue;
    strValue = pApp->GetProfileString(strSection, strStringItem);
    ASSERT(strValue == "test");

    pApp->WriteProfileInt(strSection, strIntItem, 1234);
    int nValue;
    nValue = pApp->GetProfileInt(strSection, strIntItem, 0);
    ASSERT(nValue == 1234);
```

For another example, see the example for CWinApp::GetProfileInt.

Smart Device Developer Notes

Only the registry-based version using CWinApp::SetRegistryKey is supported in Windows CE-based projects.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::GetProfileString
CWinApp::WriteProfileInt
WritePrivateProfileString
CWinApp::SetRegistryKey

Other Resources

CWinApp Members

CWinApp Data Members

For information about the data members in **CWinApp**, see CWinApp Members.

CWinApp::m_bHelpMode

TRUE if the application is in Help context mode (conventionally invoked with SHIFT + F1); otherwise **FALSE**.

```
BOOL m_bHelpMode;
```

Remarks

In Help context mode, the cursor becomes a question mark and the user can move it about the screen. Examine this flag if you want to implement special handling when in the Help mode. **m_bHelpMode** is a public variable of type **BOOL**.

See Also

Reference

CWinApp Class

Hierarchy Chart

Other Resources

CWinApp Members

CWinApp::m_bUseHtmlHelp

TRUE if the application uses HTMLHelp, **FALSE** if the application uses WinHelp.

```
BOOL m_bUseHtmlHelp;
```

Remarks

To set to **TRUE**, call CWinApp::EnableHtmlHelp in the constructor of your **CWinApp**-derived class.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::HtmlHelp

Other Resources

CWinApp Members

CWinApp::m_eHelpType

The type of this data member is the enumerated type **AFX_HELP_TYPE**, which is defined within the **CWinApp** class.

```
AFX_HELP_TYPE m_eHelpType;
```

Remarks

The **AFX_HELP_TYPE** enumeration is defined as follows:

```
enum AFX_HELP_TYPE
{
   afxWinHelp = 0,
   afxHTMLHelp = 1
};
```

- To set the application's help to HTML Help, call SetHelpMode and specify **afxHTMLHelp**.
- To set the application's help to WinHelp, call **SetHelpMode** and specify **afxWinHelp**.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::GetHelpMode
CWinApp::EnableHtmlHelp

Other Resources

CWinApp Members

CWinApp::m_hInstance

Corresponds to the *hInstance* parameter passed by Windows to **WinMain**.

```
HINSTANCE m_hInstance;
```

Remarks

The **m_hInstance** data member is a handle to the current instance of the application running under Windows. This is returned by the global function AfxGetInstanceHandle. **m_hInstance** is a public variable of type **HINSTANCE**.

Example

```
// Typically you do not need to pass the application's hInstance
// to Windows APIs directly because there are equivalent MFC
// member functions that pass the hInstance for you.  The following
// example is not typical:

HCURSOR hCursor;
hCursor = ::LoadCursor(AfxGetApp()->m_hInstance,
   MAKEINTRESOURCE(IDC_MYCURSOR));

// A more direct way to get the application's hInstance is to
// call AfxGetInstanceHandle:
hCursor = ::LoadCursor(AfxGetInstanceHandle(),
   MAKEINTRESOURCE(IDC_MYCURSOR));

// If you need the hInstance to load a resource, it is better
// to call AfxGetResourceHandle instead of AfxGetInstanceHandle:
hCursor = ::LoadCursor(AfxGetResourceHandle(),
   MAKEINTRESOURCE(IDC_MYCURSOR));

// A better way to load the cursor resource is to call
// CWinApp::LoadCursor
hCursor = AfxGetApp()->LoadCursor(IDC_MYCURSOR);
```

See Also

Reference

CWinApp Class

Hierarchy Chart

Other Resources

CWinApp Members

CWinApp::m_lpCmdLine

Corresponds to the *lpCmdLine* parameter passed by Windows to **WinMain**.

```
LPTSTR m_lpCmdLine;
```

Remarks

Points to a null-terminated string that specifies the command line for the application. Use **m_lpCmdLine** to access any command-line arguments the user entered when the application was started. **m_lpCmdLine** is a public variable of type **LPTSTR**.

Example

```
BOOL CMyApp::InitInstance()
{
   // ...

   if (m_lpCmdLine[0] == _T('\0'))
   {
      // Create a new (empty) document.
      OnFileNew();
   }
   else
   {
      // Open a file passed as the first command line parameter.
      OpenDocumentFile(m_lpCmdLine);
   }

   // ...
}
```

See Also

Reference

CWinApp Class

Hierarchy Chart

Other Resources

CWinApp Members

CWinApp::m_nCmdShow

Corresponds to the *nCmdShow* parameter passed by Windows to **WinMain**.

```
int m_nCmdShow;
```

Remarks

You should pass **m_nCmdShow** as an argument when you call CWnd::ShowWindow for your application's main window. **m_nCmdShow** is a public variable of type **int**.

Example

```
BOOL CMyApp::InitInstance()
{
   // ...

   // Create main MDI Frame window.
   CMainFrame* pMainFrame = new CMainFrame;
   if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
      return FALSE;
   // Save the pointer to the main frame window.  This is the
   // only way the framework will have knowledge of what the
   // main frame window is.
   m_pMainWnd = pMainFrame;

   // Show the   main window using the nCmdShow parameter
   // passed to the application when it was first launched.
   pMainFrame->ShowWindow(m_nCmdShow);
   pMainFrame->UpdateWindow();

   // ...
}
```

See Also

Reference

CWinApp Class

Hierarchy Chart

Other Resources

CWinApp Members

CWinApp::m_pActiveWnd

Use this data member to store a pointer to the main window of the OLE container application that has your OLE server application in-place activated.

Remarks

If this data member is **NULL**, the application is not in-place active.

The framework sets this member variable when the frame window is in-place activated by an OLE container application.

See Also

Reference

CWinApp Class
Hierarchy Chart
AfxGetMainWnd
CWinThread::m_pMainWnd

Other Resources

CWinApp Members

CWinApp::m_pszAppName

Specifies the name of the application.

```
LPCTSTR m_pszAppName;
```

Remarks

The application name can come from the parameter passed to the CWinApp constructor, or, if not specified, to the resource string with the ID of **AFX_IDS_APP_TITLE**. If the application name is not found in the resource, it comes from the program's .EXE filename.

Returned by the global function AfxGetAppName. **m_pszAppName** is a public variable of type **const char***.

Note

If you assign a value to **m_pszAppName**, it must be dynamically allocated on the heap. The **CWinApp** destructor calls **free**( ) with this pointer. You many want to use the **_tcsdup**( ) run-time library function to do the allocating. Also, free the memory associated with the current pointer before assigning a new value. For example:

```
//First free the string allocated by MFC at CWinApp startup.
//The string is allocated before InitInstance is called.
free((void*)m_pszAppName);
//Change the name of the application file.
//The CWinApp destructor will free the memory.
m_pszAppName=_tcsdup(_T("d:\\somedir\\myapp.exe"));
```

Example

```
CWnd* pWnd;
   // Set pWnd to some CWnd object whose window has already
   // been created.

   // The following call to CWnd::MessageBox uses the application
   // title as the message box caption.
   pWnd->MessageBox("Some message", AfxGetApp()->m_pszAppName);

   // A more direct way to get the application title is to
   // call AfxGetAppName:
   pWnd->MessageBox("Some message", AfxGetAppName());

   // An easier way to display a message box using the application
   // title as the message box caption is to call AfxMessageBox:
   AfxMessageBox("Some message");
```

See Also

Reference

CWinApp Class

Hierarchy Chart

Other Resources

CWinApp Members

CWinApp::m_pszExeName

Contains the name of the application's executable file without an extension.

```
LPCTSTR m_pszExeName;
```

Remarks

Unlike m_pszAppName, this name cannot contain blanks. **m_pszExeName** is a public variable of type **const char***.

Note

If you assign a value to **m_pszExeName**, it must be dynamically allocated on the heap. The **CWinApp** destructor calls **free**( ) with this pointer. You many want to use the **_tcsdup**( ) run-time library function to do the allocating. Also, free the memory a ssociated with the current pointer before assigning a new value. For example:

```
//First free the string allocated by MFC at CWinApp startup.
//The string is allocated before InitInstance is called.
free((void*)m_pszExeName);
//Change the name of the .EXE file.
//The CWinApp destructor will free the memory.
m_pszExeName=_tcsdup(_T("d:\\somedir\\myapp"));
```

See Also

Reference

CWinApp Class

Hierarchy Chart

Other Resources

CWinApp Members

CWinApp::m_pszHelpFilePath

Contains the path to the application's Help file.

```
LPCTSTR m_pszHelpFilePath;
```

Remarks

By default, the framework initializes **m_pszHelpFilePath** to the name of the application with ".HLP" appended. To change the name of the help file, set **m_pszHelpFilePath** to point to a string that contains the complete name of the desired help file. A convenient place to do this is in the application's InitInstance function. **m_pszHelpFilePath** is a public variable of type **const char***.

Note

If you assign a value to **m_pszHelpFilePath**, it must be dynamically allocated on the heap. The **CWinApp** destructor calls **free**( ) with this pointer. You many want to use the **_tcsdup**( ) run-time library function to do the allocating. Also, free the memory associated with the current pointer before assigning a new value. For example:

```
//First free the string allocated by MFC at CWinApp startup.
//The string is allocated before InitInstance is called.
free((void*)m_pszHelpFilePath);
//Change the name of the .HLP file.
//The CWinApp destructor will free the memory.
m_pszHelpFilePath=_tcsdup(_T("d:\\somedir\\myhelp.hlp"));
```

See Also

Reference

CWinApp Class

Hierarchy Chart

Other Resources

CWinApp Members

CWinApp::m_pszProfileName

Contains the name of the application's .INI file.

```
LPCTSTR m_pszProfileName;
```

Remarks

m_pszProfileName is a public variable of type **const char***.

Note

If you assign a value to **m_pszProfileName**, it must be dynamically allocated on the heap. The **CWinApp** destructor calls **free**( ) with this pointer. You many want to use the **_tcsdup**( ) run-time library function to do the allocating. Also, free the memory associated with the current pointer before assigning a new value. For example:

```
//First free the string allocated by MFC at CWinApp startup.
//The string is allocated before InitInstance is called.
free((void*)m_pszProfileName);
//Change the name of the .INI file.
//The CWinApp destructor will free the memory.
m_pszProfileName=_tcsdup(_T("d:\\somedir\\myini.ini"));
```

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::GetProfileString
CWinApp::GetProfileInt
CWinApp::WriteProfileInt
CWinApp::WriteProfileString

Other Resources

CWinApp Members

CWinApp::m_pszRegistryKey

Used to determine where, in the registry or INI file, application profile settings are stored.

```
LPCTSTR m_pszRegistryKey;
```

Remarks

Normally, this data member is treated as read-only.

Registry entries are stored as follows:

- In Windows NT, Windows 2000, and Windows XP, the value is stored to a registry key. The name for the application profile setting is appended to the following registry key: HKEY_CURRENT_USER/Software/LocalAppWizard-Generated/.
- In Windows 3.x, the value is stored in the WIN.INI file.
- In Windows 95 or Windows 98, the value is stored in a cached version of WIN.INI.

If you assign a value to **m_pszRegistryKey**, it must be dynamically allocated on the heap. The **CWinApp** destructor calls **free**( ) with this pointer. You many want to use the **_tcsdup**( ) run-time library function to do the allocating. Also, free the memory associated with the current pointer before assigning a new value. For example:

```
//First free the string allocated by MFC at CWinApp startup.
//The string is allocated before InitInstance is called.
free((void*)m_pszRegistryKey);
//Change the name of the registry key.
//The CWinApp destructor will free the memory.
m_pszRegistryKey=_tcsdup(_T("HKEY_CURRENT_USER\\Software\\mycompany\\myapp\\thissection\\th
isvalue"));
```

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::SetRegistryKey

Other Resources

CWinApp Members

CWindowDC Class

Derived from **CDC**.

```
class CWindowDC : public CDC
```

Remarks

Calls the Windows functions GetWindowDC at construction time and ReleaseDC at destruction time. This means that a **CWindowDC** object accesses the entire screen area of a CWnd (both client and nonclient areas).

For more information on using **CWindowDC**, see Device Contexts.

Requirements

Header: afxwin.h

See Also

Reference

CDC Class

Hierarchy Chart

CDC Class

Other Resources

CWindowDC Members

CWindowDC Members

Base Class Members

CObject Members

CDC Members

Construction

CWindowDC	Constructs a **CWindowDC** object.

Data Members

m_hWnd	The **HWND** to which this **CWindowDC** is attached.

See Also

Reference

CWindowDC Class

Hierarchy Chart

CWindowDC Member Functions

For information about the member functions in **CWindowDC**, see CWindowDC Members.

CWindowDC::CWindowDC

Constructs a **CWindowDC** object that accesses the entire screen area (both client and nonclient) of the **CWnd** object pointed to by *pWnd*.

```
explicit CWindowDC(
   CWnd* pWnd
);
```

Parameters

pWnd

The window whose client area the device-context object will access.

Remarks

The constructor calls the Windows function GetWindowDC.

An exception (of type **CResourceException**) is thrown if the Windows **GetWindowDC** call fails. A device context may not be available if Windows has already allocated all of its available device contexts. Your application competes for the five common display contexts available at any given time under Windows.

Example

```
// Get a dc for a CWnd object pointer.
CWindowDC dc(pWnd);

// Send my private massage.
::SendMessage(pWnd->m_hWnd, WM_MYMESSAGE, 0, 0);
```

See Also

Reference

CWindowDC Class
Hierarchy Chart
CDC Class
CClientDC Class
CWnd Class

Other Resources

CWindowDC Members

CWindowDC Data Members

For information about the data members in **CWindowDC**, see CWindowDC Members.

CWindowDC::m_hWnd

The **HWND** of the **CWnd** pointer is used to construct the **CWindowDC** object.

```
HWND m_hWnd;
```

Remarks

m_hWnd is a protected variable of type **HWND**.

Example

See the example for CWindowDC::CWindowDC.

See Also

Reference

CWindowDC Class

Hierarchy Chart

Other Resources

CWindowDC Members

CWinFormsControl Class

Provides the basic functionality for hosting of a Windows Forms control.

```
template<class TManagedControl>
class CWinFormsControl : public CWnd
```

Parameters

TManagedControl

A .NET Framework Windows Forms control to be displayed in the MFC application.

Remarks

The **CWinFormsControl** class provides the basic functionality for hosting of a Windows Forms control.

For more information on using Windows Forms, see Using a Windows Form User Control in MFC.

Your MFC code should not cache Window handles (usually stored in **m_hWnd**). Some Windows Forms control properties require that the underlying Win32 **Window** be destroyed and recreated using **DestroyWindow** and **CreateWindow**. The MFC Windows Forms implementation handles the **Destroy** and **Create** events of the controls to update the **m_hWnd** member.

Note
MFC Windows Forms integration works only in projects which link dynamically with MFC (in which AFXDLL is defined).

Requirements

Header: afxwinforms.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CWinFormsDialog Class

CWinFormsView Class

Other Resources

CWinFormsControl Members

CWinFormsControl Members

The following are members of the **CWinFormsControl** Class.

Construction

CWinFormsControl::CWinFormsControl	Constructs an MFC Windows Forms control wrapper object.

Operations

CWinFormsControl::CreateManagedControl	Creates a Windows Forms control in an MFC container.
CWinFormsControl::GetControl	Retrieves a pointer to the Windows Forms control.
CWinFormsControl::GetControlHandle	Retrieves a handle to the Windows Forms control.

Operators

CWinFormsControl::operator ->	Replaces CWinFormsControl::GetControl in expressions.
CWinFormsControl::operator TManagedControl^	Casts a type as a pointer to a Windows Forms control.

See Also

Reference

CWinFormsControl Class

CWinFormsControl Member Functions

For information about the member functions in **CWinFormsControl**, see CWinFormsControl Members.

CWinFormsControl::CreateManagedControl

Creates a Windows Forms control in an MFC container.

```
inline BOOL CreateManagedControl(
   System::Type^ pType,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   int nID
)
inline BOOL CreateManagedControl(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   int nID
);
inline BOOL CreateManagedControl(
   DWORD dwStyle,
   int nPlaceHolderID,
   CWnd* pParentWnd
);
inline BOOL CreateManagedControl(
   typename TManagedControl^ pControl,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   int nID
);
```

Parameters

pType

The data type of the control to be created. Must be a System.Type data type.

dwStyle

The window style to apply to the control. Specify a combination of Window Styles. Currently, only the following styles are supported: WS_TABSTOP, WS_VISIBLE, WS_DISABLED and WS_GROUP.

rect

A RECT Structure that defines the coordinates of the upper-left and lower-right corners of the control (first overload only).

nPlaceHolderID

The handle of the static place holder control placed in the Resource Editor. The newly created Windows Forms control replaces the static control, assuming its position, z-order, and styles (second overload only).

pParentWnd

A pointer to the parent window.

nID

The resource ID number to be assigned to the newly created control.

pControl

An instance of a Windows Forms control to be associated with the CWinFormsControl Class object (fourth overload only).

Return Value

If successful, returns a nonzero value. If unsuccessful, returns zero.

Remarks

This method instantiates a .NET Framework Windows Forms control in an MFC container.

The first overload of the method accepts a .NET Frameworks data type *pType* so that MFC can instantiate a new object of this type. *pType* must be a **System.Type** data type.

The second overload of the method creates a Windows Forms control based on the *TManagedControl* template parameter of the **CWinFormsControl** class. The size and position of the control is based on the **RECT** structure passed to the method. Only *dwStyle* matters for the styles.

The third overload of the method creates a Windows Forms control that replaces a static control, destroying it and assuming its position, z-order, and styles. The static control serves only as a placeholder for the Windows Forms control. When creating the control, this overload combines the styles from *dwStyle* with the static control's resource styles.

The fourth overload of the method allows you to pass in an already instantiated Windows Forms control *pControl* that MFC will wrap. It must be of the same type as the *TManagedControl* template parameter of the **CWinFormsControl** class.

See Using a Windows Form User Control in MFC for samples on using Windows Form controls.

See Also

Reference

CWinFormsControl Class

CWinFormsControl::GetControl

Other Resources

CWinFormsControl Members

CWinFormsControl::CWinFormsControl

Constructs an MFC Windows Forms control wrapper object.

```
CWinFormsControl( );
```

Remarks

The Windows Forms control is instantiated when you call CWinFormsControl::CreateManagedControl.

See Also

Reference

CWinFormsControl Class

Other Resources

CWinFormsControl Members

CWinFormsControl::GetControl

Retrieves a pointer to the Windows Forms control.

```
inline TManagedControl^ GetControl( ) const;
```

Return Value

Returns a pointer to the Windows Forms control.

Example

See CWinFormsControl::CreateManagedControl.

See Also

Reference

CWinFormsControl Class
CWinFormsControl::GetControlHandle
CWinFormsControl::CreateManagedControl

Other Resources

CWinFormsControl Members

CWinFormsControl::GetControlHandle

Retrieves a handle to the Windows Forms control.

```
inline HWND GetControlHandle( ) const;
```

Return Value

Returns a handle to the Windows Forms control.

Remarks

GetControlHandle is a helper method that returns the window handle stored in the .NET Framework control properties. The window handle value is copied to CWnd::m_hWnd during the call to CWnd::Attach.

See Also

Reference

CWinFormsControl Class
CWinFormsControl::GetControl
CWinFormsControl::CreateManagedControl

Other Resources

CWinFormsControl Members

CWinFormsControl Operators

For information about the operators in **CWinFormsControl**, see CWinFormsControl Members.

CWinFormsControl::operator ->

Replaces CWinFormsControl::GetControl in expressions.

```
inline TManagedControl^ operator->( ) const;
```

Remarks

This operator provides a convenient syntax that replaces **GetControl** in expressions.

For more information on Windows Forms, see Using a Windows Form User Control in MFC.

See Also

Reference

CWinFormsControl Class

CWinFormsControl::GetControl

Other Resources

CWinFormsControl Members

CWinFormsControl::operator TManagedControl^

Casts a type as a pointer to a Windows Forms control.

```
inline operator TManagedControl^( ) const;
```

Remarks

This operator passes **CWinFormsControl<***TManagedControl***>** to functions that accept a pointer to a Windows Forms control.

See Also

Reference

CWinFormsControl Class

Other Resources

CWinFormsControl Members

CWinFormsDialog Class

A wrapper for an MFC dialog class that hosts a Windows Forms user control.

```
template <typename TManagedControl>
class CWinFormsDialog :
   public CDialog
```

Parameters

TManagedControl

The .NET Framework user control to be displayed in the MFC application.

Remarks

CWinFormsDialog is a wrapper for an MFC dialog class (CDialog) that hosts a Windows Forms (UserControl) user control. This allows the display of .NET Framework controls on a modal or modeless MFC dialog box.

For more information on using Windows Forms, see Using a Windows Form User Control in MFC and Hosting a Windows Form User Control as an MFC Dialog Box.

Requirements

Header: afxwinforms.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

CWnd Class
CWinFormsView Class
CDialog Class
System.Windows.Forms.UserControl

Other Resources

CWinFormsDialog Members

CWinFormsDialog Members

The following are members of the **CWinFormsDialog** Class. There is no base class.

Construction

CWinFormsDialog	Constructs a **CWinFormsDialog** object.

Operations

GetControl	Retrieves a reference to the Windows Forms user control.
GetControlHandle	Retrieves a window handle to the Windows Forms user control.
OnInitDialog	Initializes the MFC dialog box by creating and hosting a Windows Forms user control on it.

Operators

CWinFormsDialog::operator ->	Replaces CWinFormsDialog::GetControl in expressions.
CWinFormsDialog::operator TManagedControl^	Casts a type as a reference to a Windows Forms user control.

See Also

Reference

CWinFormsDialog Class

CWinFormsDialog Member Functions

For information about the member functions in **CWinFormsDialog**, see CWinFormsDialog Members.

See Also

Reference

CWinFormsDialog Class

Other Resources

CWinFormsDialog Members

CWinFormsDialog::CWinFormsDialog

Constructs a **CWinFormsDialog** object.

```
CWinFormsDialog(
   UINT nIDTemplate = IDD
);
```

Parameters

nIDTemplate

Contains the ID of a dialog box template resource. Use the dialog editor to create the dialog template and store it in the application's resource script file. For more information on dialog templates, see CDialog Class.

See Also

Reference

CWinFormsDialog Class

CDialog Class

Other Resources

CWinFormsDialog Members

CWinFormsDialog::GetControl

Retrieves a reference to the Windows Forms user control.

```
inline TManagedControl^ GetControl( ) const;
```

Return Value

Returns a reference to the UserControl control in the MFC dialog box.

See Also

Reference

CWinFormsDialog Class
CWinFormsDialog::GetControlHandle
CDialog Class
UserControl
System.Windows.Forms.UserControl

Other Resources

CWinFormsDialog Members

CWinFormsDialog::GetControlHandle

Retrieves a window handle to the Windows Forms user control.

```
inline HWND GetControlHandle( ) const throw( );
```

Description

This method is provided as a convenient way to obtain a window handle (**HWND**) for a UserControl control, rather than performing a series of casts on a managed object, such as:

```
reinterpret_cast<HWND>(static_cast<INT_PTR>(GetControl()->Handle));
```

Return Value

Returns a window handle to the Windows Forms user control.

See Also

Reference

CWinFormsDialog Class
CWinFormsDialog::GetControl
System.Windows.Forms.UserControl

Other Resources

CWinFormsDialog Members

CWinFormsDialog::OnInitDialog

Initializes the MFC dialog box by creating and hosting a Windows Forms user control on it.

```
virtual BOOL OnInitDialog();
```

Return Value

A boolean value that specifies whether the application has set the input focus to one of the controls in the dialog box. If **OnInitDialog** returns nonzero, Windows sets the input focus to the first control in the dialog box. This method can return 0 only if the application has explicitly set the input focus to one of the controls in the dialog box.

Remarks

When the MFC dialog box is created (using the Create, CreateIndirect, or DoModal method inherited from CDialog), a **WM_INITDIALOG** message is sent and this method is called. It creates an instance of a UserControl control on the dialog box and adjusts the size of the dialog box to accommodate for the size of the user control. Then it hosts the new control in the MFC dialog box.

Override this member function if you need to perform special processing when the dialog box is initialized. For more information on using this method, see CDialog::OnInitDialog.

See Also

Reference

CWinFormsDialog Class
CDialog::OnInitDialog
System.Windows.Forms.UserControl

Other Resources

CWinFormsDialog Members

CWinFormsDialog Operators

For information about the operators in **CWinFormsDialog**, see CWinFormsDialog Members.

See Also

Reference

CWinFormsDialog Class

Other Resources

CWinFormsDialog Members

CWinFormsDialog::operator ->

Replaces CWinFormsDialog::GetControl in expressions.

```
inline TManagedControl^ operator->( ) const throw( );
```

Remarks

This operator provides a convenient syntax that replaces **GetControl** in expressions.

For information on using Windows Forms, see Using a Windows Form User Control in MFC.

See Also

Reference

CWinFormsDialog Class

CWinFormsDialog::GetControl

Other Resources

CWinFormsDialog Members

CWinFormsDialog::operator TManagedControl^

Casts a type as a reference to a Windows Forms user control.

```
inline operator TManagedControl^( ) const throw( );
```

Remarks

This operator casts a type as a reference to a UserControl control. It is used to pass a **CWinFormsDialog**<*TManagedControl*> dialog box to functions that accept a pointer to a Windows Forms user control object.

See Also

Reference

CWinFormsDialog Class

UserControl

Form

Other Resources

CWinFormsDialog Members

CWinFormsView Class

Provides generic functionality for hosting of a Windows Forms control as an MFC view.

```
class CWinFormsView : public CView;
```

Remarks

MFC uses the **CWinFormsView** class to host a .NET Framework Windows Forms control within an MFC view. The control is a child of the native view and occupies the entire client area of the MFC view. The result is similar to a **CFormView** view, allowing you to take advantage of the Windows Forms designer and run time to create rich form-based views.

For more information on using Windows Forms, see Using a Windows Form User Control in MFC.

Note
MFC Windows Forms integration works only in projects which link dynamically with MFC (projects in which AFXDLL is defined).

Note
CWinFormsView does not support the MFC splitter window (CSplitterWnd Class). Currently only the Windows Forms Splitter control (Splitter) is supported.

Requirements

Header: afxwinforms.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Reference

Hierarchy Chart
CWinFormsControl Class
CWinFormsDialog Class
CFormView Class

Concepts

CWinFormsView Members

CWinFormsView Members

The following are members of the **CWinFormsView** class.

Construction

CWinFormsView::CWinFormsView	Constructs a **CWinFormsView** object.

Operations

CWinFormsView::GetControl	Retrieves a pointer to the Windows Forms control.

Operators

CWinFormsView::operator Control^	Casts a type as a pointer to a Windows Forms control.

See Also

Reference

CWinFormsView Class

CWinFormsView Member Functions

For information about the member functions in **CWinFormsView**, see CWinFormsView Members.

CWinFormsView::CWinFormsView

Constructs a **CWinFormsView** object.

```
CWinFormsView(
   System::Type^ pManagedViewType
);
```

Parameters

pManagedViewType

A pointer to the data type of the Windows Forms user control. For more information, see System.Type.

Example

In the following example, the `CUserView` class inherits from **CWinFormsView** and passes the type of `UserControl1` to the **CWinFormsView** constructor. `UserControl1` is a custom-built control in WindowsControlLibrary1.dll.

```
#using <WindowsControlLibrary1.dll>
class CUserView : public CWinFormsView

{
   ...
};

IMPLEMENT_DYNCREATE(CUserView, CWinFormsView)
BEGIN_MESSAGE_MAP(CUserView, CWinFormsView)
CUserView::CUserView( )
: CWinFormsView(__typeof(WindowsControlLibrary1::UserControl1))
{
}
```

See Also

Reference

CWinFormsView Class

Concepts

CWinFormsView Members

CWinFormsView::GetControl

Retrieves a pointer to the Windows Forms control.

```
System::Windows::Forms::Control^ GetControl( ) const;
```

Return Value

A pointer to a System.Windows.Forms.Control object.

Remarks

For an example of how to use Windows Forms, see Using a Windows Form User Control in MFC.

See Also

Reference

CWinFormsView Class

Concepts

CWinFormsView Members

CWinFormsView Operators

For information about the operators in **CWinFormsView**, see CWinFormsView Members.

CWinFormsView::operator Control^

Casts a type as a pointer to a Windows Forms control.

```
operator System::Windows::Forms::Control^( ) const;
```

Remarks

This operator allows you to pass a **CWinFormsView** view to functions that accept a pointer to a Windows Forms control of type System.Windows.Forms.Control.

Example

See CWinFormsView::GetControl.

See Also

Reference

CWinFormsView Class

Concepts

CWinFormsView Members

CWinThread Class

Represents a thread of execution within an application.

```
class CWinThread : public CCmdTarget
```

Remarks

The main thread of execution is usually provided by an object derived from **CWinApp**; **CWinApp** is derived from **CWinThread**. Additional **CWinThread** objects allow multiple threads within a given application.

There are two general types of threads that **CWinThread** supports: worker threads and user-interface threads. Worker threads have no message pump: for example, a thread that performs background calculations in a spreadsheet application. User-interface threads have a message pump and process messages received from the system. CWinApp and classes derived from it are examples of user-interface threads. Other user-interface threads can also be derived directly from **CWinThread**.

Objects of class **CWinThread** typically exist for the duration of the thread. If you wish to modify this behavior, set m_bAutoDelete to **FALSE**.

The **CWinThread** class is necessary to make your code and MFC fully thread-safe. Thread-local data used by the framework to maintain thread-specific information is managed by **CWinThread** objects. Because of this dependence on **CWinThread** to handle thread-local data, any thread that uses MFC must be created by MFC. For example, a thread created by the run-time function _beginthread, _beginthreadex cannot use any MFC APIs.

To create a thread, call AfxBeginThread. There are two forms, depending on whether you want a worker or user-interface thread. If you want a user-interface thread, pass to **AfxBeginThread** a pointer to the **CRuntimeClass** of your **CWinThread**-derived class. If you want to create a worker thread, pass to **AfxBeginThread** a pointer to the controlling function and the parameter to the controlling function. For both worker threads and user-interface threads, you can specify optional parameters that modify priority, stack size, creation flags, and security attributes. **AfxBeginThread** will return a pointer to your new **CWinThread** object.

Instead of calling **AfxBeginThread**, you can construct a **CWinThread**-derived object and then call **CreateThread**. This two-stage construction method is useful if you want to reuse the **CWinThread** object between successive creation and terminations of thread executions.

For more information on **CWinThread**, see the articles Multithreading with C++ and MFC, Multithreading: Creating User-Interface Threads, Multithreading: Creating Worker Threads, and Multithreading: How to Use the Synchronization Classes.

Requirements

Header: afxwin.h

See Also

Reference

CCmdTarget Class
Hierarchy Chart
CWinApp Class
CCmdTarget Class

Other Resources

CWinThread Members

CWinThread Members

Base Class Members

CObject Members

CCmdTarget Members

Data Members

m_bAutoDelete	Specifies whether to destroy the object at thread termination.
m_hThread	Handle to the current thread.
m_nThreadID	ID of the current thread.
m_pActiveWnd	Pointer to the main window of the container application when an OLE server is in-place active.
m_pMainWnd	Holds a pointer to the application's main window.

Construction

CreateThread	Starts execution of a **CWinThread** object.
CWinThread	Constructs a **CWinThread** object.

Operations

GetMainWnd	Retrieves a pointer to the main window for the thread.
GetThreadPriority	Gets the priority of the current thread.
PostThreadMessage	Posts a message to another **CWinThread** object.
ResumeThread	Decrements a thread's suspend count.
SetThreadPriority	Sets the priority of the current thread.
SuspendThread	Increments a thread's suspend count.

Overridables

ExitInstance	Override to clean up when your thread terminates.
InitInstance	Override to perform thread instance initialization.
IsIdleMessage	Checks for special messages.
OnIdle	Override to perform thread-specific idle-time processing.
PreTranslateMessage	Filters messages before they are dispatched to the Windows functions TranslateMessage and DispatchMessage.
ProcessMessageFilter	Intercepts certain messages before they reach the application.
ProcessWndProcException	Intercepts all unhandled exceptions thrown by the thread's message and command handlers.
PumpMessage	Contains the thread's message loop.

Run	Controlling function for threads with a message pump. Override to customize the default message loop.

Operators

operator HANDLE	Retrieves the handle of the **CWinThread** object.

See Also

Reference

CWinThread Class

Hierarchy Chart

CWinThread Member Functions

For information about the member functions in **CWinThread**, see CWinThread Members.

CWinThread::CreateThread

Creates a thread to execute within the address space of the calling process.

```
BOOL CreateThread(
   DWORD dwCreateFlags = 0,
   UINT nStackSize = 0,
   LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL
);
```

Parameters

dwCreateFlags

Specifies an additional flag that controls the creation of the thread. This flag can contain one of two values:

- **CREATE_SUSPENDED** Start the thread with a suspend count of one. Use **CREATE_SUSPENDED** if you want to initialize any member data of the **CWinThread** object, such as m_bAutoDelete or any members of your derived class, before the thread starts running. Once your initialization is complete, use the CWinThread::ResumeThread to start the thread running. The thread will not execute until **CWinThread::ResumeThread** is called.
- **0** Start the thread immediately after creation.

nStackSize

Specifies the size in bytes of the stack for the new thread. If **0**, the stack size defaults to the same size as that of the process's primary thread.

lpSecurityAttrs

Points to a SECURITY_ATTRIBUTES structure that specifies the security attributes for the thread.

Return Value

Nonzero if the thread is created successfully; otherwise 0.

Remarks

Use **AfxBeginThread** to create a thread object and execute it in one step. Use **CreateThread** if you want to reuse the thread object between successive creation and termination of thread executions.

See Also

Reference

CWinThread Class
Hierarchy Chart
AfxBeginThread
CWinThread::CWinThread
CreateThread

Other Resources

CWinThread Members

CWinThread::CWinThread

Constructs a **CWinThread** object.

```
CWinThread( );
```

Remarks

To begin the thread's execution, call the CreateThread member function. You will usually create threads by calling AfxBeginThread, which will call this constructor and **CreateThread**.

See Also

Reference

CWinThread Class
Hierarchy Chart
CWinThread::CreateThread

Other Resources

CWinThread Members

CWinThread::ExitInstance

Called by the framework from within a rarely overridden Run member function to exit this instance of the thread, or if a call to InitInstance fails.

```
virtual int ExitInstance( );
```

Return Value

The thread's exit code; 0 indicates no errors, and values greater than 0 indicate an error. This value can be retrieved by calling GetExitCodeThread.

Remarks

Do not call this member function from anywhere but within the **Run** member function. This member function is used only in user-interface threads.

The default implementation of this function deletes the **CWinThread** object if m_bAutoDelete is **TRUE**. Override this function if you wish to perform additional clean-up when your thread terminates. Your implementation of **ExitInstance** should call the base class's version after your code is executed.

See Also

Reference

CWinThread Class
Hierarchy Chart
CWinApp::ExitInstance

Other Resources

CWinThread Members

CWinThread::GetMainWnd

If your application is an OLE server, call this function to retrieve a pointer to the active main window of the application instead of directly referring to the **m_pMainWnd** member of the application object.

```
virtual CWnd * GetMainWnd( );
```

Return Value

This function returns a pointer to one of two types of windows. If your thread is part of an OLE server and has an object that is in-place active inside an active container, this function returns the CWinApp::m_pActiveWnd data member of the **CWinThread** object.

If there is no object that is in-place active within a container or your application is not an OLE server, this function returns the m_pMainWnd data member of your thread object.

Remarks

For user-interface threads, this is equivalent to directly referring to the **m_pActiveWnd** member of your application object.

If your application is not an OLE server, then calling this function is equivalent to directly referring to the **m_pMainWnd** member of your application object.

Override this function to modify the default behavior.

See Also

Reference

CWinThread Class
Hierarchy Chart
AfxGetMainWnd

Other Resources

CWinThread Members

CWinThread::GetThreadPriority

Gets the current thread priority level of this thread.

```
int GetThreadPriority( );
```

Return Value

The current thread priority level within its priority class. The value returned will be one of the following, listed from highest priority to lowest:

- **THREAD_PRIORITY_TIME_CRITICAL**
- **THREAD_PRIORITY_HIGHEST**
- **THREAD_PRIORITY_ABOVE_NORMAL**
- **THREAD_PRIORITY_NORMAL**
- **THREAD_PRIORITY_BELOW_NORMAL**
- **THREAD_PRIORITY_LOWEST**
- **THREAD_PRIORITY_IDLE**

For more information on these priorities, see SetThreadPriority in the *Platform SDK*.

See Also

Reference

CWinThread Class
Hierarchy Chart
CWinThread::SetThreadPriority
GetThreadPriority

Other Resources

CWinThread Members

CWinThread::InitInstance

InitInstance must be overridden to initialize each new instance of a user-interface thread.

```
virtual BOOL InitInstance( );
```

Return Value

Nonzero if initialization is successful; otherwise 0.

Remarks

Typically, you override **InitInstance** to perform tasks that must be completed when a thread is first created.

This member function is used only in user-interface threads. Perform initialization of worker threads in the controlling function passed to AfxBeginThread.

See Also

Reference

CWinThread Class
Hierarchy Chart
CWinApp::InitInstance

Other Resources

CWinThread Members

CWinThread::IsIdleMessage

Override this function to keep **OnIdle** from being called after specific messages are generated.

```
virtual BOOL IsIdleMessage(
   MSG* pMsg
);
```

Parameters

pMsg

Points to the current message being processed.

Return Value

Nonzero if **OnIdle** should be called after processing message; otherwise 0.

Remarks

The default implementation does not call **OnIdle** after redundant mouse messages and messages generated by blinking carets.

If an application has created a short timer, **OnIdle** will be called frequently, causing performance problems. To improve such an application's performance, override **IsIdleMessage** in the application's **CWinApp**-derived class to check for **WM_TIMER** messages as follows:

```
BOOL CMyApp::IsIdleMessage( MSG* pMsg )
{
   if (!CWinApp::IsIdleMessage( pMsg ) ||
       pMsg->message == WM_TIMER)
      return FALSE;
   else
      return TRUE;
}
```

Handling **WM_TIMER** in this fashion will improve performance of applications that use short timers.

See Also

Reference

CWinThread Class

Hierarchy Chart

Other Resources

CWinThread Members

CWinThread::OnIdle

Override this member function to perform idle-time processing.

```
virtual BOOL OnIdle(
   LONG lCount
);
```

Parameters

lCount

A counter incremented each time **OnIdle** is called when the thread's message queue is empty. This count is reset to 0 each time a new message is processed. You can use the *lCount* parameter to determine the relative length of time the thread has been idle without processing a message.

Return Value

Nonzero to receive more idle processing time; 0 if no more idle processing time is needed.

Remarks

OnIdle is called in the default message loop when the thread's message queue is empty. Use your override to call your own background idle-handler tasks.

OnIdle should return 0 to indicate that no additional idle processing time is required. The *lCount* parameter is incremented each time **OnIdle** is called when the message queue is empty and is reset to 0 each time a new message is processed. You can call your different idle routines based on this count.

The default implementation of this member function frees temporary objects and unused dynamic link libraries from memory.

This member function is used only in user-interface threads.

Because the application cannot process messages until **OnIdle** returns, do not perform lengthy tasks in this function.

See Also

Reference

CWinThread Class
Hierarchy Chart
CWinApp::OnIdle

Other Resources

CWinThread Members

CWinThread::PostThreadMessage

Called to post a user-defined message to another **CWinThread** object.

```
BOOL PostThreadMessage(
   UINT message ,
   WPARAM wParam,
   LPARAM lParam
);
```

Parameters

message

ID of the user-defined message.

wParam

First message parameter.

lParam

Second message parameter.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The posted message is mapped to the proper message handler by the message map macro **ON_THREAD_MESSAGE**.

Note

When calling the Windows PostThreadMessage function within an MFC application, the MFC message handlers are not called. For more information, see the Knowledge Base article, "PRB: MFC Message Handler Not Called with PostThreadMessage()" (Q142415).

See Also

Reference

CWinThread Class
Hierarchy Chart
ON_THREAD_MESSAGE

Other Resources

CWinThread Members

CWinThread::PreTranslateMessage

Override this function to filter window messages before they are dispatched to the Windows functions TranslateMessage and DispatchMessage.

```
virtual BOOL PreTranslateMessage(
   MSG *pMsg
);
```

Parameters

pMsg

Points to a MSG structure containing the message to process.

Return Value

Nonzero if the message was fully processed in **PreTranslateMessage** and should not be processed further. Zero if the message should be processed in the normal way.

Remarks

This member function is used only in user-interface threads.

See Also

Reference

CWinThread Class
Hierarchy Chart
CWinApp::PreTranslateMessage

Other Resources

CWinThread Members

CWinThread::ProcessMessageFilter

The framework's hook function calls this member function to filter and respond to certain Windows messages.

```
virtual BOOL ProcessMessageFilter(
   int code,
   LPMSG lpMsg
);
```

Parameters

code

Specifies a hook code. This member function uses the code to determine how to process *lpMsg*.

lpMsg

A pointer to a Windows MSG structure.

Return Value

Nonzero if the message is processed; otherwise 0.

Remarks

A hook function processes events before they are sent to the application's normal message processing.

If you override this advanced feature, be sure to call the base-class version to maintain the framework's hook processing.

See Also

Reference

CWinApp Class
Hierarchy Chart
MessageProc
About Hooks

Other Resources

CWinThread Members

CWinThread::ProcessWndProcException

The framework calls this member function whenever the handler does not catch an exception thrown in one of your thread's message or command handlers.

```
virtual LRESULT ProcessWndProcException(
   CException *e,
   const MSG *pMsg
);
```

Parameters

e

Points to an unhandled exception.

pMsg

Points to a MSG structure containing information about the windows message that caused the framework to throw an exception.

Return Value

–1 if a **WM_CREATE** exception is generated; otherwise 0.

Remarks

Do not call this member function directly.

The default implementation of this member function handles only exceptions generated from the following messages:

Command	Action
WM_CREATE	Fail.
WM_PAINT	Validate the affected window, thus preventing another **WM_PAINT** message from being generated.

Override this member function to provide global handling of your exceptions. Call the base functionality only if you wish to display the default behavior.

This member function is used only in threads that have a message pump.

See Also

Reference

CWinThread Class
Hierarchy Chart
CWinApp::ProcessWndProcException

Other Resources

CWinThread Members

CWinThread::PumpMessage

Contains the thread's message loop.

```
virtual BOOL PumpMessage( );
```

Remarks

PumpMessage contains the thread's message loop. **PumpMessage** is called by **CWinThread** to pump the thread's messages. You can call **PumpMessage** directly to force messages to be processed, or you can override **PumpMessage** to change its default behavior.

Calling **PumpMessage** directly and overriding its default behavior is recommended for advanced users only.

See Also

Reference

CWinThread Class

Hierarchy Chart

Other Resources

CWinThread Members

CWinThread::ResumeThread

Called to resume execution of a thread that was suspended by the SuspendThread member function, or a thread created with the **CREATE_SUSPENDED** flag.

```
DWORD ResumeThread( );
```

Return Value

The thread's previous suspend count if successful; `0xFFFFFFFF` otherwise. If the return value is zero, the current thread was not suspended. If the return value is one, the thread was suspended, but is now restarted. Any return value greater than one means the thread remains suspended.

Remarks

The suspend count of the current thread is reduced by one. If the suspend count is reduced to zero, the thread resumes execution; otherwise the thread remains suspended.

See Also

Reference

CWinThread Class
Hierarchy Chart
ResumeThread
CWinThread::m_bAutoDelete
AfxBeginThread

Other Resources

CWinThread Members

CWinThread::Run

Provides a default message loop for user-interface threads.

```
virtual int Run( );
```

Return Value

An **int** value that is returned by the thread. This value can be retrieved by calling GetExitCodeThread.

Remarks

Run acquires and dispatches Windows messages until the application receives a WM_QUIT message. If the thread's message queue currently contains no messages, **Run** calls **OnIdle** to perform idle-time processing. Incoming messages go to the PreTranslateMessage member function for special processing and then to the Windows function TranslateMessage for standard keyboard translation. Finally, the DispatchMessage Windows function is called.

Run is rarely overridden, but you can override it to implement special behavior.

This member function is used only in user-interface threads.

See Also

Reference

CWinThread Class
Hierarchy Chart
CWinApp::Run

Other Resources

CWinThread Members

CWinThread::SetThreadPriority

This function sets the priority level of the current thread within its priority class.

```
BOOL SetThreadPriority(
   int nPriority
);
```

Parameters

nPriority

Specifies the new thread priority level within its priority class. This parameter must be one of the following values, listed from highest priority to lowest:

- **THREAD_PRIORITY_TIME_CRITICAL**
- **THREAD_PRIORITY_HIGHEST**
- **THREAD_PRIORITY_ABOVE_NORMAL**
- **THREAD_PRIORITY_NORMAL**
- **THREAD_PRIORITY_BELOW_NORMAL**
- **THREAD_PRIORITY_LOWEST**
- **THREAD_PRIORITY_IDLE**

For more information on these priorities, see SetThreadPriority in the *Platform SDK*.

Return Value

Nonzero if function was successful; otherwise 0.

Remarks

It can only be called after CreateThread successfully returns.

See Also

Reference

CWinThread Class
Hierarchy Chart
CWinThread::GetThreadPriority
SetThreadPriority

Other Resources

CWinThread Members

CWinThread::SuspendThread

Increments the current thread's suspend count.

```
DWORD SuspendThread( );
```

Return Value

The thread's previous suspend count if successful; `0xFFFFFFFF` otherwise.

Remarks

If any thread has a suspend count above zero, that thread does not execute. The thread can be resumed by calling the ResumeThread member function.

See Also

Reference

CWinThread Class
Hierarchy Chart
CWinThread::ResumeThread
SuspendThread

Other Resources

CWinThread Members

CWinThread Operators

For information about the operators in **CWinThread**, see CWinThread.

CWinThread::operator HANDLE

Retrieves the handle of the **CWinThread** object.

```
operator HANDLE( ) const;
```

Return Value

If successful, the handle of the thread object; otherwise, **NULL**.

Remarks

Use the handle to directly call Windows APIs.

See Also

Reference

CWinThread Class

Hierarchy Chart

Other Resources

CWinThread Members

CWinThread Data Members

For information about the data members in **CWinThread**, see CWinThread Members.

CWinThread::m_bAutoDelete

Specifies whether the **CWinThread** object should be automatically deleted at thread termination.

```
BOOL m_bAutoDelete;
```

Remarks

The **m_bAutoDelete** data member is a public variable of type **BOOL**.

See Also

Reference

CWinThread Class
Hierarchy Chart
AfxBeginThread
CWinThread::ResumeThread

Other Resources

CWinThread Members

CWinThread::m_hThread

Handle to the thread attached to this **CWinThread**.

```
HANDLE m_hThread;
```

Remarks

The **m_hThread** data member is a public variable of type **HANDLE**. It is only valid if underlying thread currently exists.

See Also

Reference

CWinThread Class

Hierarchy Chart

Other Resources

CWinThread Members

CWinThread::m_nThreadID

ID of the thread attached to this **CWinThread**.

Remarks

The **m_nThreadID** data member is a public variable of type **DWORD**. It is only valid if underlying thread currently exists.

```
DWORD m_nThreadID;
```

Example

See the example for AfxGetThread.

See Also

Reference

CWinThread Class

Hierarchy Chart

Other Resources

CWinThread Members

CWinThread::m_pActiveWnd

Use this data member to store a pointer to your thread's active window object.

```
CWnd* m_pActiveWnd;
```

Remarks

The Microsoft Foundation Class Library will automatically terminate your thread when the window referred to by **m_pActiveWnd** is closed. If this thread is the primary thread for an application, the application will also be terminated. If this data member is **NULL**, the active window for the application's **CWinApp** object will be inherited. **m_pActiveWnd** is a public variable of type **CWnd***.

Typically, you set this member variable when you override **InitInstance**. In a worker thread, the value of this data member is inherited from its parent thread.

See Also

Reference

CWinThread Class
Hierarchy Chart
CWinThread::InitInstance
CWinThread::m_pMainWnd

Other Resources

CWinThread Members

CWinThread::m_pMainWnd

Use this data member to store a pointer to your thread's main window object.

```
CWnd* m_pMainWnd;
```

Remarks

The Microsoft Foundation Class Library will automatically terminate your thread when the window referred to by **m_pMainWnd** is closed. If this thread is the primary thread for an application, the application will also be terminated. If this data member is **NULL**, the main window for the application's **CWinApp** object will be used to determine when to terminate the thread. **m_pMainWnd** is a public variable of type **CWnd***.

Typically, you set this member variable when you override **InitInstance**. In a worker thread, the value of this data member is inherited from its parent thread.

See Also

Reference

CWinThread Class

Hierarchy Chart

CWinThread::InitInstance

Other Resources

CWinThread Members

CWnd Class

Provides the base functionality of all window classes in the Microsoft Foundation Class Library.

```
class CWnd : public CCmdTarget
```

Remarks

A **CWnd** object is distinct from a Windows window, but the two are tightly linked. A **CWnd** object is created or destroyed by the **CWnd** constructor and destructor. The Windows window, on the other hand, is a data structure internal to Windows that is created by a **Create** member function and destroyed by the **CWnd** virtual destructor. The DestroyWindow function destroys the Windows window without destroying the object.

The **CWnd** class and the message-map mechanism hide the **WndProc** function. Incoming Windows notification messages are automatically routed through the message map to the proper **On***Message* **CWnd** member functions. You override an **On***Message* member function to handle a member's particular message in your derived classes.

The **CWnd** class also lets you create a Windows child window for your application. Derive a class from **CWnd**, then add member variables to the derived class to store data specific to your application. Implement message-handler member functions and a message map in the derived class to specify what happens when messages are directed to the window.

You create a child window in two steps. First, call the constructor **CWnd** to construct the **CWnd** object, then call the Create member function to create the child window and attach it to the **CWnd** object.

When the user terminates your child window, destroy the **CWnd** object, or call the **DestroyWindow** member function to remove the window and destroy its data structures.

Within the Microsoft Foundation Class Library, further classes are derived from **CWnd** to provide specific window types. Many of these classes, including CFrameWnd, CMDIFrameWnd, CMDIChildWnd, CView, and CDialog, are designed for further derivation. The control classes derived from **CWnd**, such as CButton, can be used directly or can be used for further derivation of classes.

For more information on using **CWnd**, see Frame Windows and Window Objects.

Requirements

Header: afxwin.h

Smart Device Developer Notes

The Smart Device version of CWnd Class does not support the following methods:

- CWnd::OnMouseWheel
- CWnd::OnNcCreate
- CWnd::OnToolHitTest
- CWnd::WindowProc
- CWnd::OnNcPaint
- CWnd::OnNcCalcSize

See Also

Reference

CCmdTarget Class
Hierarchy Chart
CFrameWnd Class
CView Class

Concepts

CWnd Members

CWnd Members

Base Class Members

CObject Members

CCmdTarget Members

Caret Functions	Help command Handlers and Functions	Operators
Coordinate Mapping Functions	Window Message Functions	Nonclient-Area Message Handlers
Dialog-Box Item Functions	Active Accessibility Functions	
Drag-Drop Functions	Overridables	Menu Loop Notification
Initialization	Data-Binding Functions	Initialization Message Handlers
Scrolling Functions	ActiveX Controls	Clipboard Message Handlers
Update/Painting Functions	Alert Functions	Input Message Handlers
Window Access Functions	Timer Functions	Control Message Handlers
Window Size and Position	Tool Tip Functions	General Message Handlers
Window State Functions	Menu Functions	System Message Handlers
Window Text Functions	Clipboard Functions	MDI Message Handlers

Data Members

m_hWnd	Indicates the **HWND** attached to this **CWnd**.

Construction/Destruction

CWnd	Constructs a **CWnd** object.
DestroyWindow	Destroys the attached Windows window.

Initialization

Attach	Attaches a Windows handle to a **CWnd** object.
CalcWindowRect	Called to calculate the window rectangle from the client rectangle.
Create	Creates and initializes the child window associated with the **CWnd** object.
CreateControl	Create an ActiveX control that will be represented in an MFC program by a **CWnd** object.
CreateEx	Creates a Windows overlapped, pop-up, or child window and attaches it to a **CWnd** object.
DeleteTempMap	Called automatically by the **CWinApp** idle-time handler and deletes any temporary **CWnd** objects cre ated by **FromHandle**.
Detach	Detaches a Windows handle from a **CWnd** object and returns the handle.

FromHandle	Returns a pointer to a **CWnd** object when given a handle to a window. If a **CWnd** object is not attached to the handle, a temporary **CWnd** object is created and attached.
FromHandlePermanent	Returns a pointer to a **CWnd** object when given a handle to a window. If a **CWnd** object is not attached to the handle, **NULL** is returned.
GetExStyle	Returns the window's extended style.
GetSafeHwnd	Returns **m_hWnd**, or **NULL** if the **this** pointer is **NULL**.
GetStyle	Returns the current window style.
PreCreateWindow	Called before the creation of the Windows window attached to this **CWnd** object.
PreSubclassWindow	Allows other necessary subclassing to occur before SubclassWindow is called.
SubclassWindow	Attaches a window to a **CWnd** object and makes it route messages through the **CWnd**'s message map.
UnsubclassWindow	Detaches a window from a **CWnd** object

Window State Functions

BeginModalState	Call this member function to make a frame window modal.
EnableWindow	Enables or disables mouse and keyboard input.
EndModalState	Call this member function to change a frame window from modal to modeless.
GetActiveWindow	Retrieves the active window.
GetCapture	Retrieves the **CWnd** that has the mouse capture.
GetDesktopWindow	Retrieves the Windows desktop window.
GetFocus	Retrieves the **CWnd** that currently has the input focus.
GetForegroundWindow	Returns a pointer to the foreground window (the top-level window with which the user is currently working).
GetIcon	Retrieves the handle to an icon.
GetWindowContextHelpId	Retrieves the help context identifier.
IsWindowEnabled	Determines whether the window is enabled for mouse and keyboard input.
ModifyStyle	Modifies the current window style.
ModifyStyleEx	Modifies the window's extended style.
SetActiveWindow	Activates the window.
SetCapture	Causes all subsequent mouse input to be sent to the **CWnd**.

SetFocus	Claims the input focus.
SetForegroundWindow	Puts the thread that created the window into the foreground and activates the window.
SetIcon	Sets the handle to a specific icon.
SetWindowContextHelpId	Sets the help context identifier.

Window Size and Position

ArrangeIconicWindows	Arranges all the minimized (iconic) child windows.
BringWindowToTop	Brings **CWnd** to the top of a stack of overlapping windows.
CloseWindow	Minimizes the window.
GetClientRect	Gets the dimensions of the **CWnd** client area.
GetWindowPlacement	Retrieves the show state and the normal (restored), minimized, and maximized positions of a window.
GetWindowRect	Gets the screen coordinates of **CWnd**.
GetWindowRgn	Retrieves a copy of the window region of a window.
IsIconic	Determines whether **CWnd** is minimized (iconic).
IsZoomed	Determines whether **CWnd** is maximized.
MoveWindow	Changes the position and dimensions of **CWnd**.
SetWindowPlacement	Sets the show state and the normal (restored), minimized, and maximized positions for a window.
SetWindowPos	Changes the size, position, and ordering of child, pop-up, and top-level windows.
SetWindowRgn	Sets the region of a window.

Window Access Functions

CenterWindow	Centers a window relative to its parent.
ChildWindowFromPoint	Determines which, if any, of the child windows contains the specified point.
FindWindow	Returns the handle of the window, which is identified by its window name and window class.
FindWindowEx	Returns the handle of the window, which is identified by its window name and window class.
GetAncestor	Retrieves the ancestor window object of the specified window.
GetDescendantWindow	Searches all descendant windows and returns the window with the specified ID.
GetDlgCtrlID	If the **CWnd** is a child window, calling this function returns its ID value.
GetDlgItem	Retrieves the control with the specified ID from the specified dialog box.
GetLayeredWindowAttributes	Retrieves the opacity and transparency color key of a layered window.

GetLastActivePopup	Determines which pop-up window owned by **CWnd** was most recently active.
GetNextWindow	Returns the next (or previous) window in the window manager's list.
GetOwner	Retrieves a pointer to the owner of a **CWnd**.
GetParent	Retrieves the parent window of **CWnd** (if any).
GetParentFrame	Retrieves the **CWnd** object's parent frame window.
GetParentOwner	Returns a pointer to a child window's parent window.
GetSafeOwner	Retrieves the safe owner for the given window.
GetTopLevelFrame	Retrieves the window's top-level frame window.
GetTopLevelOwner	Retrieves the top-level window.
GetTopLevelParent	Retrieves the window's top-level parent.
GetTopWindow	Returns the first child window that belongs to the **CWnd**.
GetWindow	Returns the window with the specified relationship to this window.
GetWindowInfo	Returns information about the window.
GetWindowedChildCount	Returns the number of associated child windows.
GetWindowlessChildCount	Returns the number of associated windowless child windows.
IsChild	Indicates whether **CWnd** is a child window or other direct descendant of the specified window.
SendMessageToDescendants	Sends a message to all descendant windows of the window.
SetDlgCtrlID	Sets the window or control ID for the window (which can be any child window, not only a contro l in a dialog box).
SetOwner	Changes the owner of a **CWnd**.
SetParent	Changes the parent window.
UpdateData	Initializes or retrieves data from a dialog box.
UpdateDialogControls	Call to update the state of dialog buttons and other controls.
WindowFromPoint	Identifies the window that contains the given point.

Update/Painting Functions

AnimateWindow	Animates the associated window object.
BeginPaint	Prepares **CWnd** for painting.

DrawAnimatedRects	Draws a wire-frame rectangle and animates it to indicate the opening of an icon or the minimizing or maximizing of a window.
DrawCaption	Draws a caption.
EnableScrollBar	Enables or disables one or both arrows of a scroll bar.
EndPaint	Marks the end of painting.
GetDC	Retrieves a display context for the client area.
GetDCEx	Retrieves a display context for the client area, and enables clipping while drawing.
GetUpdateRect	Retrieves the coordinates of the smallest rectangle that completely encloses the **CWnd** update region.
GetUpdateRgn	Retrieves the **CWnd** update region.
GetWindowDC	Retrieves the display context for the whole window, including the caption bar, menus, and scroll bars.
Invalidate	Invalidates the entire client area.
InvalidateRect	Invalidates the client area within the given rectangle by adding that rectangle to the current update region.
InvalidateRgn	Invalidates the client area within the given region by adding that region to the current update region.
IsWindowVisible	Determines whether the window is visible.
LockWindowUpdate	Disables or reenables drawing in the given window.
Print	Draws the current window in the specified device context.
PrintWindow	Copies a visual window into the specified device context, typically a printer DC.
PrintClient	Draws any window in the specified device context (usually a printer device context).
RedrawWindow	Updates the specified rectangle or region in the client area.
ReleaseDC	Releases client and window device contexts, freeing them for use by other applications.
SetLayeredWindowAttributes	Sets the opacity and transparency color key of a layered window.
SetRedraw	Allows changes in **CWnd** to be redrawn or prevents changes from being redrawn.
ShowOwnedPopups	Shows or hides all pop-up windows owned by the window.
ShowWindow	Shows or hides the window.
UnlockWindowUpdate	Unlocks a window that was locked with **CWnd::LockWindowUpdate**.
UpdateLayeredWindow	Updates the position, size, shape, content, and translucency of a layered window.

UpdateWindow	Updates the client area.
ValidateRect	Validates the client area within the given rectangle by removing the rectangle from the current u pdate region.
ValidateRgn	Validates the client area within the given region by removing the region from the current update region.

Coordinate Mapping Functions

ClientToScreen	Converts the client coordinates of a given point or rectangle on the display to screen coordinates.
MapWindowPoints	Converts (maps) a set of points from the coordinate space of the **CWnd** to the coordinate space of another window.
ScreenToClient	Converts the screen coordinates of a given point or rectangle on the display to client coordinates.

Window Text Functions

GetFont	Retrieves the current font.
GetWindowText	Returns the window text or caption title (if it has one).
GetWindowTextLength	Returns the length of the window's text or caption title.
SetFont	Sets the current font.
SetWindowText	Sets the window text or caption title (if it has one) to the specified text.

Scrolling Functions

EnableScrollBarCtrl	Enables or disables a sibling scroll-bar control.
GetScrollBarCtrl	Returns a sibling scroll-bar control.
GetScrollInfo	Retrieves the information that the **SCROLLINFO** structure maintains about a scroll bar.
GetScrollLimit	Retrieves the limit of the scroll bar.
GetScrollPos	Retrieves the current position of a scroll box.
GetScrollRange	Copies the current minimum and maximum scroll-bar positions for the given scroll bar.
RepositionBars	Repositions control bars in the client area.
ScrollWindow	Scrolls the contents of the client area.
ScrollWindowEx	Scrolls the contents of the client area. Similar to **ScrollWindow**, with additional features.
SetScrollInfo	Sets information about the scroll bar.
SetScrollPos	Sets the current position of a scroll box and, if specified, redraws the scroll bar to reflect the new position.
SetScrollRange	Sets minimum and maximum position values for the given scroll bar.
ShowScrollBar	Displays or hides a scroll bar.

Drag-Drop Functions

DragAcceptFiles	Indicates the window will accept dragged files.

Caret Functions

CreateCaret	Creates a new shape for the system caret and gets ownership of the caret.
CreateGrayCaret	Creates a gray block for the system caret and gets ownership of the caret.
CreateSolidCaret	Creates a solid block for the system caret and gets ownership of the caret.
GetCaretPos	Retrieves the client coordinates of the caret's current position.
HideCaret	Hides the caret by removing it from the display screen.
SetCaretPos	Moves the caret to a specified position.
ShowCaret	Shows the caret on the display at the caret's current position. Once shown, the caret begins flashing automatically.

Dialog-Box Item Functions

CheckDlgButton	Places a check mark next to or removes a check mark from a button control.
CheckRadioButton	Checks the specified radio button and removes the check mark from all other radio buttons in the specified group of buttons.
ContinueModal	Continues a window's modal status.
DlgDirList	Fills a list box with a file or directory listing.
DlgDirListComboBox	Fills the list box of a combo box with a file or directory listing.
DlgDirSelect	Retrieves the current selection from a list box.
DlgDirSelectComboBox	Retrieves the current selection from the list box of a combo box.
EndModalLoop	Ends a window's modal status.
ExecuteDlgInit	Initiates a dialog resource.
GetCheckedRadioButton	Returns the ID of the currently checked radio button in a group of buttons.
GetScrollBarInfo	Retrieves information about the specified scroll bar.
GetTitleBarInfo	Retrieves information about the specified title bar.
GetMenuBarInfo	Retrieves information about the specified menu bar.
GetDlgItemInt	Translates the text of a control in the given dialog box to an integer value.
GetDlgItemText	Retrieves the caption or text associated with a control.
GetNextDlgGroupItem	Searches for the next (or previous) control within a group of controls.
GetNextDlgTabItem	Retrieves the first control with the WS_TABSTOP style that follows (or precedes) the specified control.

IsDialogMessage	Determines whether the given message is intended for the modeless dialog box and, if so, processes it.
IsDlgButtonChecked	Determines whether a button control is checked.
RunModalLoop	Retrieves, translates, or dispatches messages for a window that is in modal status.
SendDlgItemMessage	Sends a message to the specified control.
SetDlgItemInt	Sets the text of a control to the string that represents an integer value.
SetDlgItemText	Sets the caption or text of a control in the specified dialog box.
SubclassDlgItem	Attaches a Windows control to a **CWnd** object and makes it route messages through the **CWnd**'s message map.

Data-Binding Functions

BindDefaultProperty	Binds the calling object's default simple bound property, as marked in the type library, to a cursor associated with a data-source control.
BindProperty	Binds a cursor-bound property on a data-bound control to a data-source control and registers that relationship with the MFC binding manager.
GetDSCCursor	Retrieves a pointer to the underlying cursor that is defined by the DataSource, UserName, Password, and SQL properties of a data-source control.

Menu Functions

DrawMenuBar	Redraws the menu bar.
GetMenu	Retrieves a pointer to the specified menu.
GetSystemMenu	Allows the application to access the Control menu for copying and modification.
HiliteMenuItem	Highlights or removes the highlighting from a top-level (menu-bar) menu item.
SetMenu	Sets the menu to the specified menu.

ToolTip Functions

CancelToolTips	Disables the tooltip control.
EnableToolTips	Enables the tooltip control.
EnableTrackingToolTips	Enables the tooltip control in tracking mode.
FilterToolTipMessage	Retrieves the title or text associated with a control in a dialog box.
OnToolHitTest	Determines whether a point is in the bounding rectangle of the specified tool and retrieves information about the tool.

Timer Functions

KillTimer	Kills a system timer.
SetTimer	Installs a system timer that sends a WM_TIMER message when triggered.

Alert Functions

FlashWindow	Flashes the window once.
FlashWindowEx	Flashes the window with additional functionality.
MessageBox	Creates and displays a window that contains an application-supplied message and caption.

Window Message Functions

Default	Calls the default window procedure, which provides default processing for any window messages that a n application does not process.
GetCurrentMessage	Returns a pointer to the message this window is currently processing. Should only be called when in an **On***Message* message-handler member function.
PostMessage	Places a message in the application queue, then returns without waiting for the window to process the m essage.
PreTranslateMessage	Used by **CWinApp** to filter window messages before they are dispatched to the **TranslateMessage** and **DispatchMessage** Windows functions.
SendMessage	Sends a message to the **CWnd** object and does not return until it has processed the message.
SendNotifyMessage	Sends the specified message to the window and returns as soon as possible, depending on whether the calling thread created the window.

Clipboard Functions

ChangeClipboardChain	Removes **CWnd** from the chain of Clipboard viewers.
GetClipboardOwner	Retrieves a pointer to the current owner of the Clipboard.
GetClipboardViewer	Retrieves a pointer to the first window in the chain of Clipboard viewers.
GetOpenClipboardWindow	Retrieves a pointer to the window that currently has the Clipboard open.
OpenClipboard	Opens the Clipboard. Other applications will not be able to modify the Clipboard until the Window s CloseClipboard function is called.
SetClipboardViewer	Adds **CWnd** to the chain of windows that are notified whenever the contents of the Clipboard are changed.

ActiveX Controls

GetControlUnknown	Retrieves a pointer to an unknown ActiveX control.
GetOleControlSite	Retrieves the custom site for the specified ActiveX control.
GetProperty	Retrieves an ActiveX control property.
InvokeHelper	Invokes an ActiveX control method or property.
OnAmbientProperty	Implement ambient property values.
SetProperty	Sets an ActiveX control property.

Overridables

DefWindowProc	Calls the default window procedure, which provides default processing for any window messages that an ap plication does not process.

DoDataExchange	For dialog data exchange and validation. Called by **UpdateData**.
OnChildNotify	Called by a parent window to give a notifying control a chance to respond to a control notification.
OnNotify	Called by the framework to inform a parent window an event has occurred in one of its controls or that the control needs information.
PostNcDestroy	This virtual function is called by the default OnNcDestroy function after the window has been destroyed.
WindowProc	Provides a window procedure for a **CWnd**. The default dispatches messages through the message map.

Help Command Handlers and Functions

HtmlHelp	Called to initiate the HTMLHelp application.
OnHelpFinder	Handles the **ID_HELP_FINDER** and **ID_DEFAULT_HELP** commands.
OnHelpIndex	Handles the **ID_HELP_INDEX** command and provides a default Help topic.
OnHelpUsing	Handles the **ID_HELP_USING** command.
OnHelp	Handles F1 Help within the application (using the current context).
WinHelp	Called to initiate the WinHelp application.

Initialization Message Handlers

OnInitMenu	Called when a menu is about to become active.
OnInitMenuPopup	Called when a pop-up menu is about to become active.

System Message Handlers

OnChangeUIState	Called when the user interface (UI) state should be changed.
OnQueryUIState	Called to retrieve the user interface (UI) state for a window.
OnUpdateUIState	Called to change the user interface (UI) state for the specified window and all its child windows.
OnCompacting	Called when Windows detects that system memory is low.
OnDevModeChange	Called for all top-level windows when the user changes device-mode settings.
OnDropFiles	Called when the user releases the left mouse button over a window that has registered itself as the recipient of dropped files.
OnFontChange	Called when the pool of font resources changes.
OnPaletteChanged	Called to allow windows that use a color palette to realize their logical palettes and update their client areas.
OnPaletteIsChanging	Informs other applications when an application is going to realize its logical palette.
OnSpoolerStatus	Called from Print Manager whenever a job is added to or removed from the Print Manager queue.
OnSysChar	Called when a keystroke translates to a system character.

OnSysColorChange	Called for all top-level windows when a change is made in the system color setting.
OnSysCommand	Called when the user selects a command from the Control menu, or when the user selects the Maximize or Minimize button.
OnSysDeadChar	Called when a keystroke translates to a system dead character (such as accent characters).
OnSysKeyDown	Called when the user holds down the ALT key and then presses another key.
OnSysKeyUp	Called when the user releases a key that was pressed while the ALT key was held down.
OnTimeChange	Called for all top-level windows after the system time changes.
OnWindowPosChanged	Called when the size, position, or Z-order has changed as a result of a call to SetWindowPos or another window-management function.
OnWindowPosChanging	Called when the size, position, or Z-order is about to change as a result of a call to SetWindowPos or another window-management function.
OnWinIniChange	Called for all top-level windows after the Windows initialization file, WIN.INI, is changed.

General Message Handlers

OnActivate	Called when **CWnd** is being activated or deactivated.
OnActivateApp	Called when the application is about to be activated or deactivated.
OnCancelMode	Called to allow **CWnd** to cancel any internal modes, such as mouse capture.
OnChildActivate	Called for multiple document interface (MDI) child windows whenever the size or position of **CWnd** changes or **CWnd** is activated.
OnClose	Called as a signal that **CWnd** should be closed.
OnCommand	Called when the user selects a command.
OnCopyData	Copies data from one application to another.
OnCreate	Called as a part of window creation.
OnCtlColor	Called if **CWnd** is the parent of a control when the control is about to be drawn.
OnDestroy	Called when **CWnd** is being destroyed.
OnDeviceChange	Notifies an application or device driver of a change to the hardware configuration of a device or the computer.
OnEnable	Called when **CWnd** is enabled or disabled.
OnEndSession	Called when the session is ending.
OnEnterIdle	Called to inform an application's main window procedure that a modal dialog box or a menu is entering an idle state.
OnEraseBkgnd	Called when the window background needs erasing.

OnGetMinMaxInfo	Called whenever Windows needs to know the maximized position or dimensions, or the minimum or maximum tracking size.
OnIconEraseBkgnd	Called when **CWnd** is minimized (iconic) and the background of the icon must be filled before painting the icon.
OnKillFocus	Called immediately before **CWnd** loses the input focus.
OnMenuChar	Called when the user presses a menu mnemonic character that doesn't match any of the predefined mnemonics in the current menu.
OnMenuSelect	Called when the user selects a menu item.
OnMove	Called after the position of the **CWnd** has been changed.
OnMoving	Indicates that a user is moving a **CWnd** object.
OnPaint	Called to repaint a portion of the window.
OnParentNotify	Called when a child window is created or destroyed, or when the user clicks a mouse button while the cursor is over the child window.
OnQueryDragIcon	Called when a minimized (iconic) **CWnd** is about to be dragged by the user.
OnQueryEndSession	Called when the user chooses to end the Windows session.
OnQueryNewPalette	Informs **CWnd** that it is about to receive the input focus.
OnQueryOpen	Called when **CWnd** is an icon and the user requests that the icon be opened.
OnSetFocus	Called after **CWnd** gains the input focus.
OnShowWindow	Called when **CWnd** is to be hidden or shown.
OnSize	Called after the size of **CWnd** has changed.
OnSizing	Indicates that the user is resizing the rectangle.
OnStyleChanged	Indicates that the SetWindowLong Windows function has changed one or more of the window's styles.
OnStyleChanged	Indicates that one or more of the window's styles has changed.
OnStyleChanging	Indicates that the SetWindowLong Windows function is about to change one or more of the window's styles.
OnStyleChanging	Indicates that one or more of the window's styles is about to change.

Control Message Handlers

OnCharToItem	Called by a child list box with the LBS_WANTKEYBOARDINPUT style in response to a WM_CHAR message.
OnCompareItem	Called to determine the relative position of a new item in a child sorted owner-draw combo box or list box.

OnDeleteItem	Called when an owner-draw child list box or combo box is destroyed or when items are removed fro m the control.
OnDrawItem	Called when a visual aspect of an owner-draw child button control, combo-box control, list-box contr ol, or menu needs to be drawn.
OnGetDlgCode	Called for a control so the control can process arrow-key and TAB-key input itself.
OnMeasureItem	Called for an owner-draw child combo box, list box, or menu item when the control is created. **CWnd** informs Windows of the dimensions of the control.
OnVKeyToItem	Called by a list box owned by **CWnd** in response to a WM_KEYDOWN message.
OnWndMsg	Indicates if a windows message was handled.
ReflectChildNotify	Helper function which reflects a message to its source.
ReflectLastMsg	Reflects the last message to the child window.
SendChildNotifyLastMsg	Provides a notification message to a child window, from the parent window, so the child window can handle a task.

Input Message Handlers

OnCaptureChanged	Sends a message to the window that is losing the mouse capture.
OnChar	Called when a keystroke translates to a nonsystem character.
OnDeadChar	Called when a keystroke translates to a nonsystem dead character (such as accent characters).
OnHScroll	Called when the user clicks the horizontal scroll bar of **CWnd**.
OnKeyDown	Called when a nonsystem key is pressed.
OnKeyUp	Called when a nonsystem key is released.
OnLButtonDblClk	Called when the user double-clicks the left mouse button.
OnLButtonDown	Called when the user presses the left mouse button.
OnLButtonUp	Called when the user releases the left mouse button.
OnMButtonDblClk	Called when the user double-clicks the middle mouse button.
OnMButtonDown	Called when the user presses the middle mouse button.
OnMButtonUp	Called when the user releases the middle mouse button.
OnMouseActivate	Called when the cursor is in an inactive window and the user presses a mouse button.
OnMouseMove	Called when the mouse cursor moves.
OnMouseWheel	Called when a user rotates the mouse wheel. Uses Windows NT 4.0 message handling.

OnRButtonDblClk	Called when the user double-clicks the right mouse button.
OnRButtonDown	Called when the user presses the right mouse button.
OnRButtonUp	Called when the user releases the right mouse button.
OnRegisteredMouseWheel	Called when a user rotates the mouse wheel. Uses Windows 95/98 and Windows NT 3.51 message-handling.
OnSetCursor	Called if mouse input is not captured and the mouse causes cursor movement within a window.
OnTimer	Called after each interval specified in SetTimer.
OnVScroll	Called when the user clicks the window's vertical scroll bar.

Nonclient-Area Message Handlers

OnNcActivate	Called when the nonclient area needs to be changed to indicate an active or inactive state.
OnNcCalcSize	Called when the size and position of the client area need to be calculated.
OnNcCreate	Called prior to OnCreate when the nonclient area is being created.
OnNcDestroy	Called when the nonclient area is being destroyed.
OnNcHitTest	Called by Windows every time the mouse is moved if **CWnd** contains the cursor or has captured mouse input with **SetCapture**.
OnNcLButtonDblClk	Called when the user double-clicks the left mouse button while the cursor is within a nonclient area of **CWnd**.
OnNcLButtonDown	Called when the user presses the left mouse button while the cursor is within a nonclient area of **CWnd**.
OnNcLButtonUp	Called when the user releases the left mouse button while the cursor is within a nonclient area of **CWnd**.
OnNcMButtonDblClk	Called when the user double-clicks the middle mouse button while the cursor is within a nonclient area of **CWnd**.
OnNcMButtonDown	Called when the user presses the middle mouse button while the cursor is within a nonclient area of **CWnd**.
OnNcMButtonUp	Called when the user releases the middle mouse button while the cursor is within a nonclient area of **CWnd**.
OnNcMouseMove	Called when the cursor is moved within a nonclient area of **CWnd**.
OnNcPaint	Called when the nonclient area needs painting.
OnNcRButtonDblClk	Called when the user double-clicks the right mouse button while the cursor is within a nonclient area of **CWnd**.
OnNcRButtonDown	Called when the user presses the right mouse button while the cursor is within a nonclient area of **CWnd**.

OnNcRButtonUp	Called when the user releases the right mouse button while the cursor is within a nonclient area of **CWnd**.

MDI Message Handlers

OnMDIActivate	Called when an MDI child window is activated or deactivated.

Clipboard Message Handlers

OnAskCbFormatName	Called by a Clipboard viewer application when a Clipboard owner will display the Clipboard contents.
OnChangeCbChain	Notifies that a specified window is being removed from the chain.
OnDestroyClipboard	Called when the Clipboard is emptied through a call to the Windows EmptyClipboard function.
OnDrawClipboard	Called when the contents of the change.
OnHScrollClipboard	Called when a Clipboard owner should scroll the Clipboard image, invalidate the appropriate section, and update the scroll-bar values.
OnPaintClipboard	Called when the client area of the Clipboard viewer needs repainting.
OnRenderAllFormats	Called when the owner application is being destroyed and needs to render all its formats.
OnRenderFormat	Called for the Clipboard owner when a particular format with delayed rendering needs to be rendered.
OnSizeClipboard	Called when the size of the client area of the Clipboard-viewer window has changed.
OnVScrollClipboard	Called when the owner should scroll the Clipboard image, invalidate the appropriate section, and update the scroll-bar values.

Menu Loop Notification

OnEnterMenuLoop	Called when a menu modal loop has been entered.
OnExitMenuLoop	Called when a menu modal loop has been exited.

Operators

operator !=	Determines if a window is not the same as the window whose handle is m_hWnd.
operator ==	Determines if a window is the same as the window whose handle is m_hWnd.
operator HWND	Call to get a handle to a window.

Active Accessibility

accDoDefaultAction	Called by the framework to perform the object's default action.
accHitTest	Called by the framework to retrieve the child element or child object at a given point on the screen.
accLocation	Called by the framework to retrieve the specified object's current screen location.
accNavigate	Called by the framework to traverse to another user interface element within a container and if possible, retrieve the object.
accSelect	Called by the framework to modify the selection or move the keyboard focus of the specified object.
CreateAccessibleProxy	Creates an Active Accessibility proxy for the specified object.

EnableActiveAccessibility	Enables user-defined Active Accessibility functions.
get_accChild	Called by the framework to retrieve the address of an **IDispatch** interface for the specified child.
get_accChildCount	Called by the framework to retrieve the number of children belonging to this object.
get_accDefaultAction	Called by the framework to retrieve a string that describes the object's default action.
get_accDescription	Called by framework to retrieve a string that describes the visual appearance of the specified object.
get_accFocus	Called by the framework to retrieve the object that has the keyboard focus.
get_accHelp	Called by the framework to retrieve an object's **Help** property string.
get_accHelpTopic	Called by the framework to retrieve the full path of the **WinHelp** file associated with the specified o bject and the identifier of the appropriate topic within that file.
get_accKeyboardShortcut	Called by the framework to retrieve the specified object's shortcut key or access key.
get_accName	Called by the framework to retrieve the name of the specified object.
get_accParent	Called by the framework to retrieve the **IDispatch** interface of the object's parent.
get_accRole	Called by the framework to retrieve information that describes the role of the specified object.
get_accSelection	Called by the framework to retrieve the selected children of this object.
get_accState	Called by the framework to retrieve the current state of the specified object.
NotifyWinEvent	Signals the system that a predefined event occurred.
get_accValue	Called by the framework to retrieve the value of the specified object.
PaintWindowlessControls	Draws windowless controls on the control container.

Smart Device Developer Notes

The Smart Device version of CWnd Class does not support the following methods:

- CWnd::OnMouseWheel
- CWnd::OnNcCreate
- CWnd::OnToolHitTest
- CWnd::WindowProc
- CWnd::OnNcPaint
- CWnd::OnNcCalcSize

See Also

Reference

CWnd Class
Hierarchy Chart

CWnd Member Functions

For information about the member functions in **CWnd**, see CWnd Members.

CWnd::ArrangeIconicWindows

Arranges all the minimized (iconic) child windows.

```
UINT ArrangeIconicWindows( );
```

Return Value

The height of one row of icons if the function is successful; otherwise 0.

Remarks

This member function also arranges icons on the desktop window, which covers the entire screen. The GetDesktopWindow member function retrieves a pointer to the desktop window object.

To arrange iconic MDI child windows in an MDI client window, call CMDIFrameWnd::MDIIconArrange.

Example

```
// arrange minimized MDI child windows
// called from menu item; CChildFrame is derived from CMDIChildWnd
void CChildFrame::OnActionArrangeiconicwindows()
{
   UINT height = GetParent()->ArrangeIconicWindows();
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetDesktopWindow
CMDIFrameWnd::MDIIconArrange
ArrangeIconicWindows

Concepts

CWnd Members

CWnd::accDoDefaultAction

Called by the framework to perform the object's default action.

```
virtual HRESULT accDoDefaultAction(
   VARIANT varChild
);
```

Parameters

varChild

Specifies whether the default action to be invoked is that of the object or one of the object's child elements. This parameter can be either CHILDID_SELF (to perform the object's default action) or a child ID (to perform the default action of one of the object's child elements).

Return Value

Returns S_OK on success, a COM error code on failure. See **Return Values** in IAccessible::accDoDefaultAction in the *Platform SDK*.

Remarks

This function is part of MFC's Active Accessibility support.

Override this function in your **CWnd**-derived class to perform your object's default action. For more information, see IAccessible::accDoDefaultAction in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::EnableActiveAccessibility

Concepts

CWnd Members

CWnd::accHitTest

Called by the framework to retrieve the child element or child object at a given point on the screen.

```
virtual HRESULT accHitTest(
   long xLeft,
   long yTop,
   VARIANT *pvarChild
);
```

Parameters

xLeft

X-coordinate of the point to be hit tested (in screen units).

yTop

Y-coordinate of the point to be hit tested (in screen units).

pvarChild

Receives information identifying the object at the point specified by *xLeft* and *yTop*. See *pvarID* in IAccessible::accHitTest in the *Platform SDK*.

Return Value

Returns S_OK on success, a COM error code on failure. See **Return Values** in **IAccessible::accHitTest** in the *Platform SDK*.

Remarks

This function is part of MFC's Active Accessibility support.

Override this function in your **CWnd**-derived class if you have nonwindowed user interface elements (other than windowless ActiveX controls, which MFC handles).

For more information, seeIAccessible::accHitTest in the *Platform SDK*.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::accLocation

Concepts

CWnd Members

CWnd::accLocation

Called by the framework to retrieve the specified object's current screen location.

```
virtual HRESULT accLocation(
   long *pxLeft,
   long *pyTop,
   long *pcxWidth,
   long *pcyHeight,
   VARIANT varChild
);
```

Parameters

pxLeft

Receives x-coordinate of the object's upper-left corner (in screen units).

pyTop

Receives y-coordinate of the object's upper-left corner (in screen units).

pcxWidth

Receives width of the object (in screen units).

pcyHeight

Receives height of the object (in screen units).

varChild

Specifies whether the location to be retrieved is that of the object or one of the object's child elements. This parameter can be either CHILDID_SELF (to obtain information about the object) or a child ID (to obtain information about the object's child element).

Return Value

Returns S_OK on success, a COM error code on failure. See **Return Values** in **IAccessible::accLocation** in the *Platform SDK*.

Remarks

Override this function in your **CWnd**-derived class if you have nonwindowed user interface elements (other than windowless ActiveX controls, which MFC handles).

For more information, see **IAccessible::accLocation** in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::accHitTest

Concepts

CWnd Members

CWnd::accNavigate

Called by the framework to traverse to another user interface element within a container and if possible, retrieve the object.

```
virtual HRESULT accNavigate(
   long navDir,
   VARIANT varStart,
   VARIANT *pvarEndUpAt
);
```

Parameters

navDir

Specifies the direction to navigate. See *navDir* in IAccessible::accNavigate in the *Platform SDK*.

varStart

Specifies the starting object. See *varStart* in **IAccessible::accNavigate** in the *Platform SDK*.

pvarEndUpAt

Receives information about the destination user interface object. See *pvarEnd* in **IAccessible::accNavigate** in the *Platform SDK*.

Return Value

Returns S_OK on success, a COM error code on failure. See **Return Values** in **IAccessible::accNavigate** in the *Platform SDK*.

Remarks

This function is part of MFC's Active Accessibility support.

Override this function in your **CWnd**-derived class if you have nonwindowed user interface elements (other than windowless ActiveX controls, which MFC handles).

For more information, see IAccessible::accNavigate in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::accSelect

Concepts

CWnd Members

CWnd::accSelect

Called by the framework to modify the selection or move the keyboard focus of the specified object.

```
virtual HRESULT accSelect(
   long flagsSelect,
   VARIANT varChild
);
```

Parameters

flagsSelect

Specifies how to change the current selection or focus. See *flagsSelect* in IAccessible::accSelect in the *Platform SDK*.

varChild

Specifies the object to be selected. This parameter can be either CHILDID_SELF (to select the object itself) or a child ID (to select one of the object's children).

Return Value

Returns S_OK on success, a COM error code on failure. See **Return Values** in **IAccessible::accSelect** in the *Platform SDK*.

Remarks

This function is part of MFC's Active Accessibility support.

Override this function in your **CWnd**-derived class if you have nonwindowed user interface elements (other than windowless ActiveX controls, which MFC handles).

For more information, see IAccessible::accSelect in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::get_accFocus
CWnd::get_accSelection

Concepts

CWnd Members

CWnd::AnimateWindow

Produces special effects when showing or hiding windows.

```
BOOL AnimateWindow(
   DWORD dwTime,
   DWORD dwFlags
);
```

Parameters

dwTime

Specifies how long it takes to play the animation, in milliseconds. Typically, an animation takes 200 milliseconds to play.

dwFlags

Specifies the type of animation. For a full list of possible values, see AnimateWindow.

Return Value

Nonzero if the function succeeds; otherwise 0.

Remarks

This member function emulates the functionality of the function AnimateWindow, as described in the *Platform SDK*.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::Attach

Attaches a Windows window to a **CWnd** object.

```
BOOL Attach(
   HWND hWndNew
);
```

Parameters

hWndNew

Specifies a handle to a Windows window.

Return Value

Nonzero if successful; otherwise 0.

Example

```
// Using Attach and Detach to map to the MDI client window
class CMainFrame : public CMDIFrameWnd
{
...
public:
   CWnd  m_wndMDIClient;
}

CMainFrame::~CMainFrame()
{
   // detach MDI client window
   m_wndMDIClient.Detach();
}

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CMDIFrameWnd::OnCreate(lpCreateStruct) == -1)
      return -1;

   // attach MDI client window
   if (m_wndMDIClient.Attach(m_hWndMDIClient) == 0)
   {
      TRACE0("Failed to attach MDIClient.\n");
      return -1;      // fail to create
   }
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::Detach
CWnd::m_hWnd
CWnd::SubclassWindow

Concepts

CWnd Members

CWnd::BeginModalState

Call this member function to make a frame window modal.

```
virtual void BeginModalState( );
```

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::EndModalState

CFrameWnd::BeginModalState

Concepts

CWnd Members

CWnd::BeginPaint

Prepares **CWnd** for painting and fills a **PAINTSTRUCT** data structure with information about the painting.

```
CDC* BeginPaint(
   LPPAINTSTRUCT lpPaint
);
```

Parameters

lpPaint

Points to the PAINTSTRUCT structure that is to receive painting information.

Return Value

Identifies the device context for **CWnd**. The pointer may be temporary and should not be stored beyond the scope of EndPaint.

Remarks

The paint structure contains a RECT data structure that has the smallest rectangle that completely encloses the update region and a flag that specifies whether the background has been erased.

The update region is set by the Invalidate, InvalidateRect, or InvalidateRgn member functions and by the system after it sizes, moves, creates, scrolls, or performs any other operation that affects the client area. If the update region is marked for erasing, **BeginPaint** sends an WM_ONERASEBKGND message.

Do not call the **BeginPaint** member function except in response to a WM_PAINT message. Each call to the **BeginPaint** member function must have a matching call to the EndPaint member function. If the caret is in the area to be painted, the **BeginPaint** member function automatically hides the caret to prevent it from being erased.

Example

```
// Use BeginPaint and EndPaint when responding to WM_PAINT message
// An alternative method is to use CPaintDC in place of
// BeginPaint and EndPaint
void CMyView::OnPaint()
{
   PAINTSTRUCT ps;
   CDC* pDC = BeginPaint(&ps);

   pDC->Rectangle(CRect(0, 0, 100, 100));

   EndPaint(&ps);

   // Do not call CView::OnPaint() for painting messages
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::EndPaint
CWnd::Invalidate
CWnd::InvalidateRgn
BeginPaint
CPaintDC Class

Concepts

CWnd Members

CWnd::BindDefaultProperty

Binds the calling object's default simple bound property (such as an edit control), as marked in the type library, to the underlying cursor that is defined by the DataSource, UserName, Password, and SQL properties of the data-source control.

```
void BindDefaultProperty(
   DISPID dwDispID,
   VARTYPE vtProp,
   LPCTSTR szFieldName,
   CWnd * pDSCWnd
);
```

Parameters

dwDispID

Specifies the DISPID of a property on a data-bound control that is to be bound to a data-source control.

vtProp

Specifies the type of the property to be bound — for example, **VT_BSTR**, **VT_VARIANT**, and so on.

szFieldName

Specifies the name of the column, in the cursor provided by the data-source control, to which the property will be bound.

pDSCWnd

Points to the window that hosts the data-source control to which the property will be bound. Call **GetDlgItem** with the resource ID of the DCS's host window to retrieve this pointer.

Remarks

The **CWnd** object on which you call this function must be a data-bound control.

Example

BindDefaultProperty might be used in the following context:

```
BOOL CMyDlg::OnInitDialog()
{
   ...
   CWnd* pDSC = GetDlgItem(IDC_REMOTEDATACONTROL);
   CWnd* pList = GetDlgItem(IDC_DBLISTBOX);
   pList->BindDefaultProperty(0x2,
      VT_BSTR, _T("CourseID"), pDSC);
   CWnd* pEdit = GetDlgItem(IDC_MASKEDBOX);
   pEdit->BindDefaultProperty(0x16,
      VT_BSTR, _T("InstuctorID"), pDSC);
   ...
   return TRUE;
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetDSCCursor
CWnd::BindProperty

Concepts

CWnd Members

CWnd::BindProperty

Binds a cursor-bound property on a data-bound control (such as a grid control) to a data-source control and registers that relationship with the MFC binding manager.

```
void BindProperty(
   DISPID dwDispId,
   CWnd * pWndDSC
);
```

Parameters

dwDispId

Specifies the DISPID of a property on a data-bound control that is to be bound to a data-source control.

pWndDSC

Points to the window that hosts the data-source control to which the property will be bound. Call **GetDlgItem** with the resource ID of the DCS's host window to retrieve this pointer.

Remarks

The **CWnd** object on which you call this function must be a data-bound control.

Example

BindProperty might be used in the following context:

```
BOOL CMyDlg::OnInitDialog()
{
   ...
   CWnd* pDSC = GetDlgItem(IDC_REMOTEDATACONTROL);
   CWnd* pList= GetDlgItem(IDC_DBLISTBOX);
   pList.BindProperty(0x9, pDSC);
   ...
   return TRUE;
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetDSCCursor
CWnd::BindDefaultProperty

Concepts

CWnd Members

CWnd::BringWindowToTop

Brings **CWnd** to the top of a stack of overlapping windows.

```
void BringWindowToTop( );
```

Remarks

In addition, **BringWindowToTop** activates pop-up, top-level, and MDI child windows. The **BringWindowToTop** member function should be used to uncover any window that is partially or completely obscured by any overlapping windows.

Calling this function is similar to calling the SetWindowPos function to change a window's position in the Z-order. The **BringWindowToTop** function does not change the window style to make it a top-level window of the desktop.

Example

```
// Moves MDI child windows to the top when a mouse passes
// over it. CMyView is derived from CView.
void CMyView::OnMouseMove(UINT nFlags, CPoint point)
{
   GetParentFrame()->BringWindowToTop();
}
```

See Also

Reference

CWnd Class

Hierarchy Chart

BringWindowToTop

Concepts

CWnd Members

CWnd::CalcWindowRect

Call this member function to compute the required size of the window rectangle based on the desired client-rectangle size.

```
virtual void CalcWindowRect(
   LPRECT lpClientRect,
      UINT nAdjustType = adjustBorder
);
```

Parameters

lpClientRect

Points to a RECT structure or **CRect** object that contains the resultant value of the window rectangle.

nAdjustType

An enumerated type used for in-place editing. It can have the following values: **CWnd::adjustBorder** = 0, which means that scroll-bar sizes are ignored in calculation; and **CWnd::adjustOutside** = 1, which means that they are added into the final measurements of the rectangle.

Remarks

A client rectangle is the smallest rectangle that completely encloses a client area. A window rectangle is the smallest rectangle that completely encloses the window.

Example

```
// Uses CalcWindowRect to determine size for new CFrameWnd
// based on the size of the current view. The end result is a
// top level frame window of the same size as CMyView's frame.
void CMyView::OnMyCreateframe()
{
   CFrameWnd* pFrameWnd = new CFrameWnd;
   CRect myRect;
   GetClientRect(myRect);
   pFrameWnd->Create(NULL, "My Frame");
   pFrameWnd->CalcWindowRect(&myRect, CWnd::adjustBorder);
   pFrameWnd->MoveWindow(0, 0, myRect.Width(), myRect.Height());
   pFrameWnd->ShowWindow(SW_SHOW);
}
```

See Also

Reference

CWnd Class

Hierarchy Chart

AdjustWindowRectEx

Concepts

CWnd Members

CWnd::CancelToolTips

Call this member function to remove a tool tip from the screen if a tool tip is currently displayed.

```
static void PASCAL CancelToolTips(
   BOOL bKeys = FALSE
);
```

Parameters

bKeys

TRUE to cancel tool tips when a key is pressed and set the status bar text to the default; otherwise **FALSE**.

Remarks

Note

Using this member function has no effect on tool tips managed by your code. It only affects the tool tip control managed by CWnd::EnableToolTips.

Example

```
// In this example, tool tips were set up to
// pop up when the user moves the mouse
// over this edit control.
// If the mouse is moved to the upper left-hand
// corner, the tool tip would disappear because of
// calling CancelToolTips.
void CMyEdit::OnMouseMove(UINT nFlags, CPoint point)
{
   CRect corner(0, 0, 10, 10);
   if (corner.PtInRect(point))
      CancelToolTips();
   CEdit::OnMouseMove(nFlags, point);
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::EnableToolTips
TTM_ACTIVATE

Concepts

CWnd Members

CWnd::CenterWindow

Centers a window relative to its parent.

```
void CenterWindow(
   CWnd* pAlternateOwner = NULL
);
```

Parameters

pAlternateOwner

Pointer to an alternate window relative to which it will be centered (other than the parent window).

Remarks

Usually called from CDialog::OnInitDialog to center dialog boxes relative to the main window of the application. By default, the function centers child windows relative to their parent window, and pop-up windows relative to their owner. If the pop-up window is not owned, it is centered relative to the screen. To center a window relative to a specific window which is not the owner or parent, the *pAlternateOwner* parameter may be set to a valid window. To force centering relative to the screen, pass the value returned by CWnd::GetDesktopWindow as *pAlternateOwner*.

Example

```
// The following example will center an About dialog in response to a
// button click. It is assumed that the About dialog contains an
// additional button named IDC_CENTERWINDOW, which is also registered
// in the CAboutDlg message map, as follows:
//   BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
//      ON_BN_CLICKED(IDC_CENTERWINDOW, OnCenterWindow)
//   END_MESSAGE_MAP()

void CAboutDlg::OnCenterWindow()
{
   CenterWindow();
}
```

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::GetDesktopWindow

CDialog::OnInitDialog

Concepts

CWnd Members

CWnd::ChangeClipboardChain

Removes **CWnd** from the chain of Clipboard viewers and makes the window specified by *hWndNext* the descendant of the **CWnd** ancestor in the chain.

```
BOOL ChangeClipboardChain(
   HWND hWndNext
);
```

Parameters

hWndNext

Identifies the window that follows **CWnd** in the Clipboard-viewer chain.

Return Value

Nonzero if successful; otherwise 0.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::SetClipboardViewer
ChangeClipboardChain

Concepts

CWnd Members

CWnd::CheckDlgButton

Selects (places a check mark next to) or clears (removes a check mark from) a button, or it changes the state of a three-state button.

```
void CheckDlgButton(
   int nIDButton,
   UINT nCheck
);
```

Parameters

nIDButton

Specifies the button to be modified.

nCheck

Specifies the action to take. If *nCheck* is nonzero, the **CheckDlgButton** member function places a check mark next to the button; if 0, the check mark is removed. For three-state buttons, if *nCheck* is 2, the button state is indeterminate.

Remarks

The **CheckDlgButton** function sends a BM_SETCHECK message to the specified button.

Example

```
// Sets 3 check buttons in various ways.  Note BST_INDETERMINATE
// requires BS_3STATE or BS_AUTO3STATE in the button's style.
void CMyDlg::OnMarkButtons()
{
   CheckDlgButton(IDC_CHECK1, BST_UNCHECKED);   // 0
   CheckDlgButton(IDC_CHECK2, BST_CHECKED);   // 1
   CheckDlgButton(IDC_CHECK3, BST_INDETERMINATE);   // 2
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::IsDlgButtonChecked
CButton::SetCheck
CheckDlgButton

Concepts

CWnd Members

CWnd::CheckRadioButton

Selects (adds a check mark to) a given radio button in a group and clears (removes a check mark from) all other radio buttons in the group.

```
void CheckRadioButton(
   int nIDFirstButton,
   int nIDLastButton,
   int nIDCheckButton
);
```

Parameters

nIDFirstButton

Specifies the integer identifier of the first radio button in the group.

nIDLastButton

Specifies the integer identifier of the last radio button in the group.

nIDCheckButton

Specifies the integer identifier of the radio button to be checked.

Remarks

The **CheckRadioButton** function sends a BM_SETCHECK message to the specified radio button.

Example

```
// Of the 4 radio buttons, selects radio button 3.
void CMyDlg::OnMarkRadio()
{
   CheckRadioButton(IDC_RADIO1, IDC_RADIO4, IDC_RADIO3);
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetCheckedRadioButton
CButton::SetCheck
CheckRadioButton

Concepts

CWnd Members

CWnd::ChildWindowFromPoint

Determines which, if any, of the child windows belonging to **CWnd** contains the specified point.

```
CWnd* ChildWindowFromPoint(
   POINT point
) const;
CWnd* ChildWindowFromPoint(
   POINT point,
   UINT nFlags
) const;
```

Parameters

point

Specifies the client coordinates of the point to be tested.

nflags

Specifies which child windows to skip. This parameter can be a combination of the following values:

Value	Meaning
CWP_ALL	Do not skip any child windows
CWP_SKIPINVISIBLE	Skip invisible child windows
CWP_SKIPDISABLED	Skip disabled child windows
CWP_SKIPTRANSPARENT	Skip transparent child windows

Return Value

Identifies the child window that contains the point. It is **NULL** if the given point lies outside of the client area. If the point is within the client area but is not contained within any child window, **CWnd** is returned.

This member function will return a hidden or disabled child window that contains the specified point.

More than one window may contain the given point. However, this function returns only the **CWnd*** of the first window encountered that contains the point.

The **CWnd*** that is returned may be temporary and should not be stored for later use.

Example

```
void CMyDlg::OnFindCenterChild()
{
   CRect rect;
   GetClientRect(&rect);
   CWnd* pWnd = ChildWindowFromPoint(
      CPoint(rect.Width()/2, rect.Height()/2),
      // Top left is always 0, 0.
      CWP_SKIPINVISIBLE);
   TRACE("Center window is 0x%08x\n", pWnd->m_hWnd);
}
```

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::WindowFromPoint

ChildWindowFromPoint

Concepts

CWnd Members

CWnd::ClientToScreen

Converts the client coordinates of a given point or rectangle on the display to screen coordinates.

```
void ClientToScreen(
   LPPOINT lpPoint
) const;
void ClientToScreen(
   LPRECT lpRect
) const;
```

Parameters

lpPoint

Points to a POINT structure or **CPoint** object that contains the client coordinates to be converted.

lpRect

Points to a RECT structure or **CRect** object that contains the client coordinates to be converted.

Remarks

The **ClientToScreen** member function uses the client coordinates in the **POINT** or **RECT** structure or the **CPoint** or **CRect** object pointed to by *lpPoint* or *lpRect* to compute new screen coordinates; it then replaces the coordinates in the structure with the new coordinates. The new screen coordinates are relative to the upper-left corner of the system display.

The **ClientToScreen** member function assumes that the given point or rectangle is in client coordinates.

Example

```
// resize dialog to client's size
void CAboutDlg::OnButton1()
{
   CRect myRect;
   GetClientRect(&myRect);

   ClientToScreen(myRect);
   MoveWindow(myRect.left, myRect.top,
      myRect.Width(), myRect.Height());
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::ScreenToClient
ClientToScreen

Concepts

CWnd Members

CWnd::CloseWindow

Minimizes the window.

```
void CloseWindow( );
```

Remarks

This member function emulates the functionality of the function CloseWindow, as described in the *Platform SDK*.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::ContinueModal

This member function is called by RunModalLoop to determine when the modal state should be exited.

```
virtual BOOL ContinueModal( );
```

Return Value

Nonzero if modal loop is to be continued; 0 when EndModalLoop is called.

Remarks

By default, it returns non-zero until **EndModalLoop** is called.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::RunModalLoop
CWnd::EndModalLoop

Concepts

CWnd Members

CWnd::Create

Creates a Windows child window and attaches it to the **CWnd** object.

```
virtual BOOL Create(
   LPCTSTR lpszClassName,
   LPCTSTR lpszWindowName,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID,
   CCreateContext* pContext = NULL
);
```

Parameters

lpszClassName

Points to a null-terminated character string that names the Windows class (a **WNDCLASS** structure). The class name can be any name registered with the global **AfxRegisterWndClass** function or any of the predefined control-class names. If **NULL**, uses the default **CWnd** attributes.

lpszWindowName

Points to a null-terminated character string that contains the window name.

dwStyle

Specifies the window style attributes. **WS_POPUP** cannot be used. If you wish to create a pop-up window, use CWnd::CreateEx instead.

rect

The size and position of the window, in client coordinates of *pParentWnd*.

pParentWnd

The parent window.

nID

The ID of the child window.

pContext

Pointer to an optional CCreateContext structure used to override portions of the creation process.

Return Value

Nonzero if successful; otherwise 0.

Remarks

You construct a child window in two steps. First, call the constructor, which constructs the **CWnd** object. Then call **Create**, which creates the Windows child window and attaches it to **CWnd**. **Create** initializes the window's class name and window name and registers values for its style, parent, and ID.

For a list of the predefined control-class names available for the *lpszClassName* parameter, refer to About Window Classes.

Example

```
// Dynamically create static control using CWnd::Create,
// instead of with CStatic::Create, which doesn't
// need the "STATIC" class name.

void CMyDlg::OnCreateStatic()
{
   CWnd* pWnd = new CWnd;
   pWnd->Create(_T("STATIC"), "Hi", WS_CHILD | WS_VISIBLE,
```

```
        CRect(0, 0, 20, 20), this, 1234);
}
```

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::CWnd

CWnd::CreateEx

Concepts

CWnd Members

CWnd::CreateAccessibleProxy

Creates an Active Accessibility proxy for the specified object.

```
virtual HRESULT CreateAccessibleProxy(
   WPARAM wParam,
   LPARAM lParam,
   LRESULT *pResult
);
```

Parameters

wParam

Identifies the object accessed by the Active Accessibility proxy. Can be one of the following values

Value	Meaning
OBJID_CLIENT	Refers to the window's client area.

lParam

Provides additional message-dependent information.

pResult

A pointer to an **LRESULT** that stores the result code.

Remarks

Creates an Active Accessibility proxy for the specified object.

See Also

Reference

CWnd Class
Hierarchy Chart
WM_GETOBJECT
LresultFromObject

Concepts

CWnd Members

CWnd::CreateCaret

Creates a new shape for the system caret and claims ownership of the caret.

```
void CreateCaret(
   CBitmap* pBitmap
);
```

Parameters

pBitmap

Identifies the bitmap that defines the caret shape.

Remarks

The bitmap must have previously been created by the CBitmap::CreateBitmap member function, the CreateDIBitmap Windows function, or the CBitmap::LoadBitmap member function.

CreateCaret automatically destroys the previous caret shape, if any, regardless of which window owns the caret. Once created, the caret is initially hidden. To show the caret, the ShowCaret member function must be called.

The system caret is a shared resource. **CWnd** should create a caret only when it has the input focus or is active. It should destroy the caret before it loses the input focus or becomes inactive.

Example

```
// Changes the caret of the edit control in this dialog box
void CMyDlg::OnChangeCaret()
{
   CBitmap* pBitmap = new CBitmap;
   pBitmap->LoadBitmap(IDB_HAPPY_BITMAP);
   m_editCtrl.CreateCaret(pBitmap);
   m_editCtrl.ShowCaret();
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CBitmap::CreateBitmap
CreateDIBitmap
DestroyCaret
CBitmap::LoadBitmap
CWnd::ShowCaret
CreateCaret

Concepts

CWnd Members

CWnd::CreateControl

Use this member function to create an ActiveX control that will be represented in the MFC program by a **CWnd** object.

```
BOOL CreateControl(
   LPCTSTR pszClass,
   LPCTSTR pszWindowName,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID,
   CFile* pPersist = NULL,
   BOOL bStorage = FALSE,
   BSTR bstrLicKey = NULL
);
BOOL CreateControl(
   REFCLSID clsid,
   LPCTSTR pszWindowName,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID,
   CFile* pPersist = NULL,
   BOOL bStorage = FALSE,
   BSTR bstrLicKey = NULL
);
BOOL CreateControl(
   REFCLSID clsid,
   LPCTSTR pszWindowName,
   DWORD dwStyle,
   const POINT* ppt,
   const SIZE* psize,
   CWnd* pParentWnd,
   UINT nID,
   CFile* pPersist = NULL,
   BOOL bStorage = FALSE,
    BSTR bstrLicKey = NULL
);
```

Parameters

pszClass

This string may contain the OLE "short name" (ProgID) for the class, e.g., "CIRC3.Circ3Ctrl.1". The name needs to match the same name registered by the control. Alternatively, the string may contain the string form of a **CLSID**, contained in braces, e.g., "{9DBAFCCF-592F-101B-85CE-00608CEC297B}". In either case, **CreateControl** converts the string to the corresponding class ID.

pszWindowName

A pointer to the text to be displayed in the control. Sets the value of the control's Caption or Text property (if any). If **NULL**, the control's Caption or Text property is not changed.

dwStyle

Windows styles. The available styles are listed under Remarks.

rect

Specifies the control's size and position. It can be either a CRect object or a RECT structure.

ppt

Points to a POINT structure or **CPoint** object that contains the upper left corner of the control.

pSize

Points to a SIZE structure or **CSize** object that contains the control's size

pParentWnd

Specifies the control's parent window. It must not be **NULL**.

nID

Specifies the control's ID.

pPersist

A pointer to a CFile containing the persistent state for the control. The default value is **NULL**, indicating that the control initializes itself without restoring its state from any persistent storage. If not **NULL**, it should be a pointer to a **CFile**-derived object which contains the control's persistent data, in the form of either a stream or a storage. This data could have been saved in a previous activation of the client. The **CFile** can contain other data, but must have its read-write pointer set to the first byte of persistent data at the time of the call to **CreateControl**.

bStorage

Indicates whether the data in *pPersist* should be interpreted as IStorage or IStream data. If the data in *pPersist* is a storage, *bStorage* should be **TRUE**. If the data in *pPersist* is a stream, *bStorage* should be **FALSE**. The default value is **FALSE**.

bstrLicKey

Optional license key data. This data is needed only for creating controls that require a run-time license key. If the control supports licensing, you must provide a license key for the creation of the control to succeed. The default value is **NULL**.

clsid

The unique class ID of the control.

Return Value

Nonzero if successful; otherwise 0.

Remarks

CreateControl is a direct analog of the CWnd::Create function, which creates the window for a **CWnd**. **CreateControl** creates an ActiveX control instead of an ordinary window.

Only a subset of the Windows *dwStyle* flags are supported for **CreateControl**:

- **WS_VISIBLE** Creates a window that is initially visible. Required if you want the control to be visible immediately, like ordinary windows.
- **WS_DISABLED** Creates a window that is initially disabled. A disabled window cannot receive input from the user. Can be set if the control has an Enabled property.
- **WS_BORDER** Creates a window with a thin-line border. Can be set if control has a BorderStyle property.
- **WS_GROUP** Specifies the first control of a group of controls. The user can change the keyboard focus from one control in the group to the next by using the direction keys. All controls defined with the **WS_GROUP** style after the first control belong to the same group. The next control with the **WS_GROUP** style ends the group and starts the next group.
- **WS_TABSTOP** Specifies a control that can receive the keyboard focus when the user presses the TAB key. Pressing the TAB key changes the keyboard focus to the next control of the **WS_TABSTOP** style.

Example

```
// This code is generated by the Control Wizard.
// It wraps the call to CreateControl in the call to Create.
class CGenocx : public CWnd
{
protected:
   DECLARE_DYNCREATE(CGenocx)
public:
   CLSID const& GetClsid()
   {
      static CLSID const clsid
         = { 0x8a6bbfeb, 0xf9bb, 0x11d1, { 0xb9, 0xc9, 0x0,
         0x60, 0x8, 0x93, 0xb0, 0xfe } };
```

```
        return clsid;
    }
    virtual BOOL Create(LPCTSTR lpszClassName,
        LPCTSTR lpszWindowName, DWORD dwStyle,
        const RECT& rect,
        CWnd* pParentWnd, UINT nID,
        CCreateContext* pContext = NULL)
    {
        return CreateControl(GetClsid(), lpszWindowName,
        dwStyle, rect, pParentWnd, nID);
    }
...
};
```

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

MFC ActiveX Controls

CWnd::CreateEx

Creates an overlapped, pop-up, or child window with the extended style specified in *dwExStyle*.

```
virtual BOOL CreateEx(
   DWORD dwExStyle,
   LPCTSTR lpszClassName,
   LPCTSTR lpszWindowName,
   DWORD dwStyle,
   int x,
   int y,
   int nWidth,
   int nHeight,
   HWND hWndParent,
   HMENU nIDorHMenu,
   LPVOID lpParam = NULL
);
virtual BOOL CreateEx(
   DWORD dwExStyle,
   LPCTSTR lpszClassName,
   LPCTSTR lpszWindowName,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID,
   LPVOID lpParam = NULL
);
```

Parameters

dwExStyle

Specifies the extended style of the **CWnd** being created. Apply any of the extended window styles to the window.

lpszClassName

Points to a null-terminated character string that names the Windows class (a WNDCLASS structure). The class name can be any name registered with the global AfxRegisterWndClass function or any of the predefined control-class names.

lpszWindowName

Points to a null-terminated character string that contains the window name.

dwStyle

Specifies the window style attributes. See Window Styles and CWnd::Create for a description of the possible values.

x

Specifies the initial x-position of the **CWnd** window.

y

Specifies the initial top position of the **CWnd** window.

nWidth

Specifies the width (in device units) of the **CWnd** window.

nHeight

Specifies the height (in device units) of the **CWnd** window.

hwndParent

Identifies the parent or owner window of the **CWnd** window being created. Use **NULL** for top-level windows.

nIDorHMenu

Identifies a menu or a child-window identifier. The meaning depends on the style of the window.

lpParam

Points to the data referenced by the **lpCreateParams** field of the CREATESTRUCT structure.

rect

The size and position of the window, in client coordinates of *pParentWnd*.

pParentWnd

The parent window.

nID

The ID of the child window.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The **CreateEx** parameters specify the **WNDCLASS**, window title, window style, and (optionally) initial position and size of the window. **CreateEx** also specifies the window's parent (if any) and ID.

When **CreateEx** executes, Windows sends the WM_GETMINMAXINFO, WM_NCCREATE, WM_NCCALCSIZE, and WM_CREATE messages to the window.

To extend the default message handling, derive a class from **CWnd**, add a message map to the new class, and provide member functions for the above messages. Override **OnCreate**, for example, to perform needed initialization for a new class.

Override further **On***Message* message handlers to add further functionality to your derived class.

If the **WS_VISIBLE** style is given, Windows sends the window all the messages required to activate and show the window. If the window style specifies a title bar, the window title pointed to by the *lpszWindowName* parameter is displayed in the title bar.

The *dwStyle* parameter can be any combination of window styles.

Example

```
void CMyDlg::OnCreateExtendedCtrl()
{
   CWnd* pWnd = new CStatic;
   pWnd->CreateEx(WS_EX_CLIENTEDGE, // Make a client edge label.
      _T("STATIC"), "Hi",
      WS_CHILD | WS_TABSTOP | WS_VISIBLE,
      5, 5, 30, 30, m_hWnd, (HMENU)1234);
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::Create
CreateWindowEx

Concepts

CWnd Members

CWnd::CreateGrayCaret

Creates a gray rectangle for the system caret and claims ownership of the caret.

```
void CreateGrayCaret(
   int nWidth,
   int nHeight
);
```

Parameters

nWidth

Specifies the width of the caret (in logical units). If this parameter is 0, the width is set to the system-defined window-border width.

nHeight

Specifies the height of the caret (in logical units). If this parameter is 0, the height is set to the system-defined window-border height.

Remarks

The caret shape can be a line or a block.

The parameters *nWidth* and *nHeight* specify the caret's width and height (in logical units); the exact width and height (in pixels) depend on the mapping mode.

The system's window-border width or height can be retrieved by the GetSystemMetrics Windows function with the **SM_CXBORDER** and **SM_CYBORDER** indexes. Using the window-border width or height ensures that the caret will be visible on a high-resolution display.

The **CreateGrayCaret** member function automatically destroys the previous caret shape, if any, regardless of which window owns the caret. Once created, the caret is initially hidden. To show the caret, the ShowCaret member function must be called.

The system caret is a shared resource. **CWnd** should create a caret only when it has the input focus or is active. It should destroy the caret before it loses the input focus or becomes inactive.

Example

```
// Create a 5x10 gray caret in the edit control.
void CMyDlg::OnChangeCaret()
{
   m_editCtrl.CreateGrayCaret(5, 10);
   m_editCtrl.ShowCaret();
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
DestroyCaret
GetSystemMetrics
CWnd::ShowCaret
CreateCaret

Concepts

CWnd Members

CWnd::CreateSolidCaret

Creates a solid rectangle for the system caret and claims ownership of the caret.

```
void CreateSolidCaret(
   int nWidth,
   int nHeight
);
```

Parameters

nWidth

Specifies the width of the caret (in logical units). If this parameter is 0, the width is set to the system-defined window-border width.

nHeight

Specifies the height of the caret (in logical units). If this parameter is 0, the height is set to the system-defined window-border height.

Remarks

The caret shape can be a line or block.

The parameters *nWidth* and *nHeight* specify the caret's width and height (in logical units); the exact width and height (in pixels) depend on the mapping mode.

The system's window-border width or height can be retrieved by the GetSystemMetrics Windows function with the **SM_CXBORDER** and **SM_CYBORDER** indexes. Using the window-border width or height ensures that the caret will be visible on a high-resolution display.

The **CreateSolidCaret** member function automatically destroys the previous caret shape, if any, regardless of which window owns the caret. Once created, the caret is initially hidden. To show the caret, the ShowCaret member function must be called.

The system caret is a shared resource. **CWnd** should create a caret only when it has the input focus or is active. It should destroy the caret before it loses the input focus or becomes inactive.

Example

```
// Create a 5x10 solid caret in the edit control.
void CMyDlg::OnChangeCaret()
{
   m_editCtrl.CreateSolidCaret(5, 10);
   m_editCtrl.ShowCaret();
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
DestroyCaret
GetSystemMetrics
CWnd::ShowCaret
CreateCaret

Concepts

CWnd Members

CWnd::CWnd

Constructs a **CWnd** object.

```
CWnd( );
```

Remarks

The Windows window is not created and attached until the CreateEx or Create member function is called.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::CreateEx
CWnd::Create

Concepts

CWnd Members

CWnd::Default

Calls the default window procedure.

```
LRESULT Default( );
```

Return Value

Depends on the message sent.

Remarks

The default window procedure provides default processing for any window message that an application does not process. This member function ensures that every message is processed.

Example

```
// This sample shows how to avoid any button handling in base class,
// if any, and call the default window procedure directly.
void CTestwndView::OnLButtonDown(UINT nFlags, CPoint point)
{
   CWnd::Default();
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::DefWindowProc
DefWindowProc

Concepts

CWnd Members

CWnd::DefWindowProc

Calls the default window procedure, which provides default processing for any window message that an application does not process.

```
virtual LRESULT DefWindowProc(
   UINT message,
   WPARAM wParam,
   LPARAM lParam
);
```

Parameters

message

Specifies the Windows message to be processed.

wParam

Specifies additional message-dependent information.

lParam

Specifies additional message-dependent information.

Return Value

Depends on the message sent.

Remarks

This member function ensures that every message is processed. It should be called with the same parameters as those received by the window procedure.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::Default
DefWindowProc

Concepts

CWnd Members

CWnd::DeleteTempMap

Called automatically by the idle time handler of the **CWinApp** object.

```
static void PASCAL DeleteTempMap( );
```

Remarks

Deletes any temporary **CWnd** objects created by the **FromHandle** member function.

Example

```
// The following example is correct:
CWnd::DeleteTempMap();

// The following example is incorrect.
// DeleteTempMap() is a static member and cannot be called
// within the scope of an instantiated CWnd object.
pWnd->DeleteTempMap();  // Causes compiler error
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::FromHandle

Concepts

CWnd Members

CWnd::DestroyWindow

Destroys the Windows window attached to the **CWnd** object.

```
virtual BOOL DestroyWindow( );
```

Return Value

Nonzero if the window is destroyed; otherwise 0.

Remarks

The **DestroyWindow** member function sends appropriate messages to the window to deactivate it and remove the input focus. It also destroys the window's menu, flushes the application queue, destroys outstanding timers, removes Clipboard ownership, and breaks the Clipboard-viewer chain if **CWnd** is at the top of the viewer chain. It sends WM_DESTROY and WM_NCDESTROY messages to the window. It does not destroy the **CWnd** object.

DestroyWindow is a place holder for performing cleanup. Because **DestroyWindow** is a virtual function, it is shown in any **CWnd**-derived class in Class View. But even if you override this function in your **CWnd**-derived class, **DestroyWindow** is not necessarily called. If **DestroyWindow** is not called in the MFC code, then you have to explicitly call it in your own code if you want it to be called.

Assume, for example, you have overridden **DestroyWindow** in a **CView**-derived class. Since MFC source code does not call **DestroyWindow** in any of its **CFrameWnd**-derived classes, your overridden **DestroyWindow** will not be called unless you call it explicitly.

If the window is the parent of any windows, these child windows are automatically destroyed when the parent window is destroyed. The **DestroyWindow** member function destroys child windows first and then the window itself.

The **DestroyWindow** member function also destroys modeless dialog boxes created by CDialog::Create.

If the **CWnd** being destroyed is a child window and does not have the WS_EX_NOPARENTNOTIFY style set, then the WM_PARENTNOTIFY message is sent to the parent.

Example

```
// CModeless is CDialog class representing a modeless dialog
// Destruction of the modeless dialog involves calling DestroyWindow in
// OnOK() & OnCancel() handlers
void CModeless::OnOK()
{
   if (!UpdateData(TRUE))
   {
      TRACE0("UpdateData failed during dialog termination\n");
      // The UpdateData routine will set focus to correct item
      return;
   }
   DestroyWindow();
}

void CModeless::OnCancel()
{
   DestroyWindow();
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnDestroy
CWnd::Detach
DestroyWindow

Concepts

CWnd Members

CWnd::Detach

Detaches a Windows handle from a **CWnd** object and returns the handle.

```
HWND Detach( );
```

Return Value

A **HWND** to the Windows object.

Example

See the example for CWnd::Attach.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::Attach

Concepts

CWnd Members

CWnd::DlgDirList

Fills a list box with a file or directory listing.

```
int DlgDirList(
   LPTSTR lpPathSpec,
   int nIDListBox,
   int nIDStaticPath,
   UINT nFileType
);
```

Parameters

lpPathSpec

Points to a null-terminated string that contains the path or filename. **DlgDirList** modifies this string, which should be long enough to contain the modifications. For more information, see the following "Remarks" section.

nIDListBox

Specifies the identifier of a list box. If *nIDListBox* is 0, **DlgDirList** assumes that no list box exists and does not attempt to fill one.

nIDStaticPath

Specifies the identifier of the static-text control used to display the current drive and directory. If *nIDStaticPath* is 0, **DlgDirList** assumes that no such text control is present.

nFileType

Specifies the attributes of the files to be displayed. It can be any combination of the following values:

- **DDL_READWRITE** Read-write data files with no additional attributes.
- **DDL_READONLY** Read-only files.
- **DDL_HIDDEN** Hidden files.
- **DDL_SYSTEM** System files.
- **DDL_DIRECTORY** Directories.
- **DDL_ARCHIVE** Archives.
- **DDL_POSTMSGS** **LB_DIR** flag. If the **LB_DIR** flag is set, Windows places the messages generated by **DlgDirList** in the application's queue; otherwise, they are sent directly to the dialog-box procedure.
- **DDL_DRIVES** Drives. If the **DDL_DRIVES** flag is set, the **DDL_EXCLUSIVE** flag is set automatically. Therefore, to create a directory listing that includes drives and files, you must call **DlgDirList** twice: once with the **DDL_DRIVES** flag set and once with the flags for the rest of the list.
- **DDL_EXCLUSIVE** Exclusive bit. If the exclusive bit is set, only files of the specified type are listed; otherwise normal files and files of the specified type are listed.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

DlgDirList sends LB_RESETCONTENT and LB_DIR messages to the list box. It fills the list box specified by *nIDListBox* with the names of all files that match the path given by *lpPathSpec*.

The *lpPathSpec* parameter has the following form:

```
[drive:] [ [\u]directory[\idirectory]...\u] [filename]
```

In this example, *drive* is a drive letter, *directory* is a valid directory name, and *filename* is a valid filename that must contain at least one wildcard. The wildcards are a question mark (**?**), which means match any character, and an asterisk (*****), meaning match any number of characters.

If you specify a 0-length string for *lpPathSpec*, or if you specify only a directory name but do not include any file specification, the string will be changed to "*.*".

If *lpPathSpec* includes a drive and/or directory name, the current drive and directory are changed to the designated drive and directory before the list box is filled. The text control identified by *nIDStaticPath* is also updated with the new drive and/or directory name.

After the list box is filled, *lpPathSpec* is updated by removing the drive and/or directory portion of the path.

Example

```
// If pDialog points to a CDialog object with a list box
// with the identifier IDC_DIRLIST, this call will populate
// the box with only the non-hidden subdirectories in the root
// directory of the C:\ drive.

enum {sizeOfPath = MAX_PATH};
TCHAR path[sizeOfPath = {0};
_tcscpy_s(path, sizeOfPath, _T("C:\\"));

pDialog->DlgDirList(path, IDC_DIRLIST, 0, DDL_EXCLUSIVE |
                                          DDL_DIRECTORY);
```

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::DlgDirListComboBox

DlgDirList

Concepts

CWnd Members

CWnd::DlgDirListComboBox

Fills the list box of a combo box with a file or directory listing.

```
int DlgDirListComboBox(
   LPTSTR lpPathSpec,
   int nIDComboBox,
   int nIDStaticPath,
   UINT nFileType
);
```

Parameters

lpPathSpec

Points to a null-terminated string that contains the path or filename. **DlgDirListComboBox** modifies this string, so this data should not be in the form of a string literal. See the following "Remarks" section.

nIDComboBox

Specifies the identifier of a combo box in a dialog box. If *nIDComboBox* is 0, **DlgDirListComboBox** assumes that no combo box exists and does not attempt to fill one.

nIDStaticPath

Specifies the identifier of the static-text control used to display the current drive and directory. If *nIDStaticPath* is 0, **DlgDirListComboBox** assumes that no such text control is present.

nFileType

Specifies DOS file attributes of the files to be displayed. It can be any combination of the following values:

- **DDL_READWRITE** Read-write data files with no additional attributes.
- **DDL_READONLY** Read-only files.
- **DDL_HIDDEN** Hidden files.
- **DDL_SYSTEM** System files.
- **DDL_DIRECTORY** Directories.
- **DDL_ARCHIVE** Archives.
- **DDL_POSTMSGS** **CB_DIR** flag. If the **CB_DIR** flag is set, Windows places the messages generated by **DlgDirListComboBox** in the application's queue; otherwise, they are sent directly to the dialog-box procedure.
- **DDL_DRIVES** Drives. If the **DDL_DRIVES** flag is set, the **DDL_EXCLUSIVE** flag is set automatically. Therefore, to create a directory listing that includes drives and files, you must call **DlgDirListComboBox** twice: once with the **DDL_DRIVES** flag set and once with the flags for the rest of the list.
- **DDL_EXCLUSIVE** Exclusive bit. If the exclusive bit is set, only files of the specified type are listed; otherwise normal files and files of the specified type are listed.

Return Value

Specifies the outcome of the function. It is nonzero if a listing was made, even an empty listing. A 0 return value implies that the input string did not contain a valid search path.

Remarks

DlgDirListComboBox sends CB_RESETCONTENT and CB_DIR messages to the combo box. It fills the list box of the combo box specified by *nIDComboBox* with the names of all files that match the path given by *lpPathSpec*.

The *lpPathSpec* parameter has the following form:

```
[drive:] [ [\u]directory[\idirectory]...\u] [filename]
```

In this example, *drive* is a drive letter, *directory* is a valid directory name, and *filename* is a valid filename that must contain at least one wildcard. The wildcards are a question mark (**?**), which means match any character, and an asterisk (*****), which means match any number of characters.

If you specify a zero-length string for *lpPathSpec*, the current directory will be used and *lpPathSpec* will not be modified. If you specify only a directory name but do not include any file specification, the string will be changed to "*".

If *lpPathSpec* includes a drive and/or directory name, the current drive and directory are changed to the designated drive and directory before the list box is filled. The text control identified by *nIDStaticPath* is also updated with the new drive and/or directory name.

After the combo-box list box is filled, *lpPathSpec* is updated by removing the drive and/or directory portion of the path.

Example

```
// If pDialog points to a CDialog object with a combo box
// with the identifier IDC_COMBO1, this call will populate
// the box with only the non-hidden subdirectories in the root
// directory of the C:\ drive.

TCHAR szPath[MAX_PATH];
_tcsncpy_s(szPath, MAX_PATH, _T("C:\\"));
pDialog->DlgDirListComboBox(szPath, IDC_COMBO1, 0, DDL_EXCLUSIVE | DDL_DIRECTORY);

// Note that the first argument is a string and not a string
// literal. This is necessary because DlgDirListComboBox
// modifies the supplied string. Passing a string literal
// will result in an access violation.
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::DlgDirList
CWnd::DlgDirSelect
DlgDirListComboBox

Concepts

CWnd Members

CWnd::DlgDirSelect

Retrieves the current selection from a list box.

```
BOOL DlgDirSelect(
   LPTSTR lpString,
   int nIDListBox
);
```

Parameters

lpString

Points to a buffer that is to receive the current selection in the list box.

nIDListBox

Specifies the integer ID of a list box in the dialog box.

Return Value

Nonzero if successful; otherwise 0.

Remarks

It assumes that the list box has been filled by the DlgDirList member function and that the selection is a drive letter, a file, or a directory name.

The **DlgDirSelect** member function copies the selection to the buffer given by *lpString*. If there is no selection, *lpString* does not change.

DlgDirSelect sends LB_GETCURSEL and LB_GETTEXT messages to the list box.

It does not allow more than one filename to be returned from a list box. The list box must not be a multiple-selection list box.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::DlgDirList
CWnd::DlgDirListComboBox
CWnd::DlgDirSelectComboBox
DlgDirSelectEx

Concepts

CWnd Members

CWnd::DlgDirSelectComboBox

Retrieves the current selection from the list box of a combo box.

```
BOOL DlgDirSelectComboBox(
   LPTSTR lpString,
   int nIDComboBox
);
```

Parameters

lpString

Points to a buffer that is to receive the selected path.

nIDComboBox

Specifies the integer ID of the combo box in the dialog box.

Return Value

Nonzero if successful; otherwise 0.

Remarks

It assumes that the list box has been filled by the DlgDirListComboBox member function and that the selection is a drive letter, a file, or a directory name.

The **DlgDirSelectComboBox** member function copies the selection to the specified buffer. If there is no selection, the contents of the buffer are not changed.

DlgDirSelectComboBox sends CB_GETCURSEL and CB_GETLBTEXT messages to the combo box.

It does not allow more than one filename to be returned from a combo box.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::DlgDirListComboBox
DlgDirSelectComboBoxEx

Concepts

CWnd Members

CWnd::DoDataExchange

Called by the framework to exchange and validate dialog data.

```
virtual void DoDataExchange(
   CDataExchange* pDX
);
```

Parameters

pDX

A pointer to a **CDataExchange** object.

Remarks

Never call this function directly. It is called by the UpdateData member function. Call **UpdateData** to initialize a dialog box's controls or retrieve data from a dialog box.

When you derive an application-specific dialog class from CDialog, you need to override this member function if you wish to utilize the framework's automatic data exchange and validation. The Add Variable wizard will write an overridden version of this member function for you containing the desired "data map" of dialog data exchange (DDX) and validation (DDV) global function calls.

To automatically generate an overridden version of this member function, first create a dialog resource with the dialog editor, then derive an application-specific dialog class. Then use the Add Variable wizard to associate variables, data, and validation ranges with various controls in the new dialog box. The wizard then writes the overridden **DoDataExchange**, which contains a data map. The following is an example DDX/DDV code block generated by the Add Variable wizard:

```
void CPenWidthsDlg::DoDataExchange(CDataExchange* pDX)
{
   CDialog::DoDataExchange(pDX);

      DDX_Text(pDX, IDC_THIN_PEN_WIDTH, m_nThinWidth);

      DDV_MinMaxInt(pDX, m_nThinWidth, 1, 20);

      DDX_Text(pDX, IDC_THICK_PEN_WIDTH, m_nThickWidth);

      DDV_MinMaxInt(pDX, m_nThickWidth, 1, 20);
}
```

The **DoDataExchange** overridden member function must precede the macro statements in your source file.

For more information on dialog data exchange and validation, see Displaying and Manipulating Data in a Form and Dialog Data Exchange and Validation. For a description of the DDX_ and DDV_ macros generated by the Add Variable wizard, see Technical Note 26.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::UpdateData

Concepts

CWnd Members

CWnd::DragAcceptFiles

Call this member function from within a window, using a **CWnd** pointer, in your application's CWinApp::InitInstance function to indicate that the window accepts dropped files from the Windows File Manager or Windows Explorer.

```
void DragAcceptFiles(
   BOOL bAccept = TRUE
);
```

Parameters

BAccept

Flag that indicates whether dragged files are accepted.

Remarks

Only the window that calls **DragAcceptFiles** with the *bAccept* parameter set to **TRUE** has identified itself as able to process the Windows message **WM_DROPFILES**. For example, in an MDI application, if the **CMDIFrameWnd** window pointer is used in the **DragAcceptFiles** function call, only the **CMDIFrameWnd** window gets the **WM_DROPFILES** message. This message is not sent to all open **CMDIChildWnd** windows. For a **CMDIChildWnd** window to receive this message, you must call **DragAcceptFiles** with the **CMDIChildWnd** window pointer.

To discontinue receiving dragged files, call the member function with *bAccept* set to **FALSE**.

See Also

Reference

CWnd Class
Hierarchy Chart
DragAcceptFiles
WM_DROPFILES

Concepts

CWnd Members

CWnd::DragDetect

Captures the mouse and tracks its movement until the user releases the left button, presses the ESC key, or moves the mouse outside the drag rectangle around the specified point.

```
BOOL DragDetect(
   POINT pt
) const;
```

Parameters

pt

Initial position of the mouse, in screen coordinates. The function determines the coordinates of the drag rectangle by using this point.

Return Value

If the user moved the mouse outside of the drag rectangle while holding down the left button , the return value is nonzero.

If the user did not move the mouse outside of the drag rectangle while holding down the left button , the return value is zero.

Remarks

This member function emulates the functionality of the function DragDetect, as described in the *Platform SDK*.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::DrawAnimatedRects

Draws a wire-frame rectangle and animates it to indicate the opening of an icon or the minimizing or maximizing of a window.

```
BOOL DrawAnimatedRects(
   int idAni,
   CONST RECT *lprcFrom,
   CONST RECT *lprcTo
);
```

Parameters

idAni

Specifies the type of animation. If you specify **IDANI_CAPTION**, the window caption will animate from the position specified by *lprcFrom* to the position specified by *lprcTo*. The effect is similar to minimizing or maximizing a window.

lprcFrom

Pointer to a RECT structure specifying the location and size of the icon or minimized window.

lprcTo

Pointer to a RECT structure specifying the location and size of the restored window

Return Value

Nonzero if the function succeeds; otherwise 0.

Remarks

This member function emulates the functionality of the function DrawAnimatedRects, as described in the *Platform SDK*.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::DrawCaption

Draws a window caption.

```
BOOL DrawCaption(
   CDC* pDC,
   LPCRECT lprc,
   UINT uFlags
);
```

Parameters

pDC

A pointer to a device context. The function draws the window caption into this device context.

lprc

A pointer to a RECT structure that specifies the bounding rectangle for the window caption.

uFlags

Specifies drawing options. For a complete list of values, see DrawCaption.

Return Value

Nonzero if the function succeeds; otherwise 0.

Remarks

This member function emulates the functionality of the function DrawCaption, as described in the *Platform SDK*.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::DrawMenuBar

Redraws the menu bar.

```
void DrawMenuBar( );
```

Remarks

If a menu bar is changed after Windows has created the window, call this function to draw the changed menu bar.

Example

See the example for CWnd::GetMenu.

See Also

Reference

CWnd Class

Hierarchy Chart

DrawMenuBar

Concepts

CWnd Members

CWnd::EnableActiveAccessibility

Enables user-defined Active Accessibility functions.

```
void EnableActiveAccessibility( );
```

Remarks

MFC's default Active Accessibility support is sufficient for standard windows and controls, including ActiveX controls; however, if your **CWnd**-derived class contains nonwindowed user interface elements, MFC has no way of knowing about them. In that case, you must override the appropriate Active Accessibility member functions in your class, and you must call **EnableActiveAccessibility** in the class's constructor.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::EnableScrollBar

Enables or disables one or both arrows of a scroll bar.

```
BOOL EnableScrollBar(
   int nSBFlags,
   UINT nArrowFlags = ESB_ENABLE_BOTH
);
```

Parameters

nSBFlags

Specifies the scroll-bar type. Can have one of the following values:

- **SB_BOTH** Enables or disables the arrows of the horizontal and vertical scroll bars associated with the window.
- **SB_HORZ** Enables or disables the arrows of the horizontal scroll bar associated with the window.
- **SB_VERT** Enables or disables the arrows of the vertical scroll bar associated with the window.

nArrowFlags

Specifies whether the scroll-bar arrows are enabled or disabled and which arrows are enabled or disabled. Can have one of the following values:

- **ESB_ENABLE_BOTH** Enables both arrows of a scroll bar (default).
- **ESB_DISABLE_LTUP** Disables the left arrow of a horizontal scroll bar or the up arrow of a vertical scroll bar.
- **ESB_DISABLE_RTDN** Disables the right arrow of a horizontal scroll bar or the down arrow of a vertical scroll bar.
- **ESB_DISABLE_BOTH** Disables both arrows of a scroll bar.

Return Value

Nonzero if the arrows are enabled or disabled as specified. Otherwise it is 0, which indicates that the arrows are already in the requested state or that an error occurred.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::ShowScrollBar
CScrollBar::EnableScrollBar

Concepts

CWnd Members

CWnd::EnableScrollBarCtrl

Enables or disables the scroll bar for this window.

```
void EnableScrollBarCtrl(
   int nBar,
   BOOL bEnable = TRUE
);
```

Parameters

nBar

The scroll-bar identifier.

bEnable

Specifies whether the scroll bar is to be enabled or disabled.

Remarks

If the window has a sibling scroll-bar control, that scroll bar is used; otherwise the window's own scroll bar is used.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetScrollBarCtrl

Concepts

CWnd Members

CWnd::EnableToolTips

Enables tool tips for the given window.

```
BOOL EnableToolTips(
   BOOL bEnable = TRUE
);
```

Parameters

bEnable

Specifies whether the tool tip control is enabled or disabled. **TRUE** enables the control; **FALSE** disables the control.

Return Value

TRUE if tool tips are enabled; otherwise **FALSE**.

Remarks

Override OnToolHitTest to provide the TOOLINFO struct or structs for the window.

Note
Some windows, such as CToolBar, provide a built-in implementation of OnToolHitTest.

See TOOLINFO in the *Platform SDK* for more information on this structure.

Simply calling **EnableToolTips** is not enough to display tool tips for your child controls unless the parent window is derived from **CFrameWnd**. This is because **CFrameWnd** provides a default handler for the **TTN_NEEDTEXT** notification. If your parent window is not derived from **CFrameWnd**, that is, if it is a dialog box or a form view, tool tips for your child controls will not display correctly unless you provide a handler for the **TTN_NEEDTEXT** tool tip notification. See Tool Tips.

The default tool tips provided for your windows by **EnableToolTips** do not have text associated with them. To retrieve text for the tool tip to display, the **TTN_NEEDTEXT** notification is sent to the tool tip control's parent window just before the tool tip window is displayed. If there is no handler for this message to assign some value to the *pszText* member of the **TOOLTIPTEXT** structure, there will be no text displayed for the tool tip.

Example

```
BEGIN_MESSAGE_MAP(CMyView, CView)
   ...
   ON_NOTIFY_EX_RANGE(TTN_NEEDTEXTW, 0, 0xFFFF, OnToolTipNotify)
   ON_NOTIFY_EX_RANGE(TTN_NEEDTEXTA, 0, 0xFFFF, OnToolTipNotify)
END_MESSAGE_MAP()

void CMyView::OnInitialUpdate()
{
   CView::OnInitialUpdate();

   CEdit* pEdit = new CEdit;
   pEdit->Create(ES_MULTILINE | WS_CHILD | WS_VISIBLE | WS_TABSTOP | WS_BORDER,
      CRect(10, 10, 100, 100), this, 111);
   EnableToolTips(TRUE);   // enable tool tips for view
}

//Notification handler
BOOL CMyView::OnToolTipNotify(UINT id, NMHDR *pNMHDR,
   LRESULT *pResult)
{
   // need to handle both ANSI and UNICODE versions of the message
   TOOLTIPTEXTA* pTTTA = (TOOLTIPTEXTA*)pNMHDR;
   TOOLTIPTEXTW* pTTTW = (TOOLTIPTEXTW*)pNMHDR;
   CString strTipText;
   UINT nID = pNMHDR->idFrom;
```

```
    if (pNMHDR->code == TTN_NEEDTEXTA && (pTTTA->uFlags & TTF_IDISHWND) ||
       pNMHDR->code == TTN_NEEDTEXTW && (pTTTW->uFlags & TTF_IDISHWND))
    {
       // idFrom is actually the HWND of the tool
       nID = ::GetDlgCtrlID((HWND)nID);
    }

    if (nID != 0) // will be zero on a separator
       strTipText.Format("Control ID = %d", nID);

    if (pNMHDR->code == TTN_NEEDTEXTA)
       lstrcpyn(pTTTA->szText, strTipText, sizeof(pTTTA->szText));
    else
       ::MultiByteToWideChar( CP_ACP , 0, strTipText, -1, pTTTW->szText, sizeof(pTTTW->szTex
t)/(sizeof pTTTW->szText[0]) );
    *pResult = 0;

    return TRUE;    // message was handled
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::CancelToolTips
CWnd::OnToolHitTest
CToolBar Class
TOOLINFO

Concepts

CWnd Members

CWnd::EnableTrackingToolTips

Enables or disables tracking tooltips.

```
BOOL EnableTrackingToolTips(
   BOOL bEnable = TRUE
);
```

Parameters

bEnable

Specifies whether tracking tool tips are enabled or disabled. If this parameter is **TRUE**, the tracking tool tips will be enabled. If this parameter is **FALSE**, the tracking tool tips will be disabled.

Return Value

Indicates the state before the **EnableWindow** member function was called. The return value is nonzero if the window was previously disabled. The return value is 0 if the window was previously enabled or an error occurred.

Remarks

Tracking tool tips are tool tip windows that you can dynamically position on the screen. By rapidly updating the position, the tool tip window appears to move smoothly, or "track." This functionality can be useful if you need tool tip text to follow the position of the pointer as it moves.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::EnableToolTips
Tooltip Controls

Concepts

CWnd Members

CWnd::EnableWindow

Enables or disables mouse and keyboard input.

```
BOOL EnableWindow(
   BOOL bEnable = TRUE
);
```

Parameters

bEnable

Specifies whether the given window is to be enabled or disabled. If this parameter is **TRUE**, the window will be enabled. If this parameter is **FALSE**, the window will be disabled.

Return Value

Indicates the state before the **EnableWindow** member function was called. The return value is nonzero if the window was previously disabled. The return value is 0 if the window was previously enabled or an error occurred.

Remarks

When input is disabled, input such as mouse clicks and keystrokes is ignored. When input is enabled, the window processes all input.

If the enabled state is changing, the WM_ENABLE message is sent before this function returns.

If disabled, all child windows are implicitly disabled, although they are not sent **WM_ENABLE** messages.

A window must be enabled before it can be activated. For example, if an application is displaying a modeless dialog box and has disabled its main window, the main window must be enabled before the dialog box is destroyed. Otherwise, another window will get the input focus and be activated. If a child window is disabled, it is ignored when Windows tries to determine which window should get mouse messages.

By default, a window is enabled when it is created. An application can specify the **WS_DISABLED** style in the Create or CreateEx member function to create a window that is initially disabled. After a window has been created, an application can also use the **EnableWindow** member function to enable or disable the window.

An application can use this function to enable or disable a control in a dialog box. A disabled control cannot receive the input focus, nor can a user access it.

Example

```
//CMyFileDialog is a CFileDialog-derived class
//OnInitDialog is the handler for WM_INITDIALOG
BOOL CMyFileDialog::OnInitDialog()
{
   CFileDialog::OnInitDialog();

   CWnd* pWndParent = GetParent();

   //make sure you add #include <dlgs.h> for IDs 'edt1' & 'stc3'
   //disables the 'file name' edit and static control
   //of the standard file open dialog

   //get handle of 'file name' edit control & disable it
   CWnd* pWnd = pWndParent->GetDlgItem(edt1);
   pWnd->EnableWindow(FALSE);

   //get handle of 'file name' static control & disable it
   pWnd = pWndParent->GetDlgItem(stc3);
   pWnd->EnableWindow(FALSE);

   return TRUE;
}
```

See Also

Reference

CWnd Class

Hierarchy Chart

EnableWindow

CWnd::OnEnable

Concepts

CWnd Members

CWnd::EndModalLoop

Terminates a call to **RunModalLoop**.

```
virtual void EndModalLoop(
   int nResult
);
```

Parameters

nResult

Contains the value to be returned to the caller of RunModalLoop.

Remarks

The *nResult* parameter is propagated to the return value from **RunModalLoop**.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::RunModalLoop
CWnd::ContinueModal

Concepts

CWnd Members

CWnd::EndModalState

Call this member function to change a frame window from modal to modeless.

```
virtual void EndModalState( );
```

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::BeginModalState

CFrameWnd::EndModalState

Concepts

CWnd Members

CWnd::EndPaint

Marks the end of painting in the given window.

```
void EndPaint(
   LPPAINTSTRUCT lpPaint
);
```

Parameters

lpPaint

Points to a PAINTSTRUCT structure that contains the painting information retrieved by the BeginPaint member function.

Remarks

The **EndPaint** member function is required for each call to the **BeginPaint** member function, but only after painting is complete.

If the caret was hidden by the **BeginPaint** member function, **EndPaint** restores the caret to the screen.

Example

See the example for CWnd::BeginPaint.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::BeginPaint
EndPaint
CPaintDC Class

Concepts

CWnd Members

CWnd::ExecuteDlgInit

Initiates a dialog resource.

```
BOOL ExecuteDlgInit(
   LPCTSTR lpszResourceName
);
BOOL ExecuteDlgInit(
   LPVOID lpResource
);
```

Parameters

lpszResourceName

A pointer to a null-terminated string specifying the name of the resource.

lpResource

A pointer to a resource.

Return Value

TRUE if a dialog resource is executed; otherwise **FALSE**.

Remarks

ExecuteDlgInit will use resources bound to the executing module, or resources from other sources. To accomplish this, **ExecuteDlgInit** finds a resource handle by calling **AfxFindResourceHandle**. If your MFC application does not use the shared DLL (MFCx0[U][D].DLL), **AfxFindResourceHandle** calls AfxGetResourceHandle, which returns the current resource handle for the executable. If your MFC application that uses MFCx0[U][D].DLL, **AfxFindResourceHandle** traverses the **CDynLinkLibrary** object list of shared and extension DLLs looking for the correct resource handle.

See Also

Reference

CWnd Class
Hierarchy Chart
CDialog::OnInitDialog
WM_INITDIALOG

Concepts

CWnd Members

CWnd::FilterToolTipMessage

Called by the framework to display the tool tip message associated with a button on the toolbar.

```
void FilterToolTipMessage(
   MSG* pMsg
);
```

Parameters

pMsg

A pointer to the tool tip message.

Remarks

It is normally called from PreTranslateMessage.

Call it when the framework does not call it for you.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::OnToolHitTest

Concepts

CWnd Members

CWnd::FindWindow

Returns the top-level **CWnd** whose window class is given by *lpszClassName* and whose window name, or title, is given by *lpszWindowName*.

```
static CWnd* PASCAL FindWindow(
   LPCTSTR lpszClassName,
   LPCTSTR lpszWindowName
);
```

Parameters

lpszClassName

Points to a null-terminated string that specifies the window's class name (a **WNDCLASS** structure). If *lpClassName* is **NULL**, all class names match.

lpszWindowName

Points to a null-terminated string that specifies the window name (the window's title). If *lpWindowName* is **NULL**, all window names match.

Return Value

Identifies the window that has the specified class name and window name. It is **NULL** if no such window is found.

The **CWnd*** may be temporary and should not be stored for later use.

Remarks

This function does not search child windows.

Example

```
// activate an application with a window with a specific class name
      BOOL COneT32App::FirstInstance()
      {
         CWnd *pWndPrev, *pWndChild;

         // Determine if a window with the class name exists...
         if (pWndPrev = CWnd::FindWindow(_T("MyNewClass"),NULL))
         {
            // If so, does it have any popups?
            pWndChild = pWndPrev->GetLastActivePopup();

            // If iconic, restore the main window
            if (pWndPrev->IsIconic())
               pWndPrev->ShowWindow(SW_RESTORE);

            // Bring the main window or its popup to the foreground
            pWndChild->SetForegroundWindow();

            // and you are done activating the other application
            return FALSE;
         }
      }
```

See Also

Reference

CWnd Class

Hierarchy Chart

FindWindow

Concepts

CWnd Members

CWnd::FindWindowEx

Retrieves the window object whose class name and window name match the specified strings.

```
static CWnd* FindWindowEx(
   HWND hwndParent,
   HWND hwndChildAfter,
   LPCTSTR lpszClass,
   LPCTSTR lpszWindow
);
```

Parameters

hwndParent

Handle to the parent window whose child windows are to be searched.

hwndChildAfter

Handle to a child window. The search begins with the next child window in the Z order. The child window must be a direct child window of *hwndParent*, not just a descendant window.

lpszClass

Pointer to a null-terminated string that specifies the class name or a class atom created by a previous call to the RegisterClass or RegisterClassEx.

lpszWindow

Pointer to a null-terminated string that specifies the window name (the window's title). If this parameter is **NULL**, all window names match.

Return Value

If the function succeeds, the return value is a pointer to the window object having the specified class and window names. If the function fails, the return value is **NULL**.

Remarks

This member function emulates the functionality of the function FindWindowEx, as described in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::FindWindow

Concepts

CWnd Members

CWnd::FlashWindow

Flashes the given window once.

```
BOOL FlashWindow(
   BOOL bInvert
);
```

Parameters

bInvert

Specifies whether the **CWnd** is to be flashed or returned to its original state. The **CWnd** is flashed from one state to the other if *bInvert* is **TRUE**. If *bInvert* is **FALSE**, the window is returned to its original state (either active or inactive).

Return Value

Nonzero if the window was active before the call to the **FlashWindow** member function; otherwise 0.

Remarks

For successive flashing, create a system timer and repeatedly call **FlashWindow**. Flashing the **CWnd** means changing the appearance of its title bar as if the **CWnd** were changing from inactive to active status, or vice versa. (An inactive title bar changes to an active title bar; an active title bar changes to an inactive title bar.)

Typically, a window is flashed to inform the user that it requires attention but that it does not currently have the input focus.

The *bInvert* parameter should be **FALSE** only when the window is getting the input focus and will no longer be flashing; it should be **TRUE** on successive calls while waiting to get the input focus.

This function always returns nonzero for minimized windows. If the window is minimized, **FlashWindow** will simply flash the window's icon; *bInvert* is ignored for minimized windows.

Example

```
BOOL CAboutDlg::OnInitDialog()
{
   CDialog::OnInitDialog();

   // set timer to cause dialog to flash
   SetTimer(1, 500, NULL);
   return TRUE;  // return TRUE unless you set the focus to a control
}

void CAboutDlg::OnTimer(UINT nIDEvent)
{
   // cause the dialog to flash
   FlashWindow(TRUE);
   CDialog::OnTimer(nIDEvent);
}
```

See Also

Reference

CWnd Class

Hierarchy Chart

FlashWindow

Concepts

CWnd Members

CWnd::FlashWindowEx

Flashes the given window.

```
BOOL FlashWindowEx(
   DWORD dwFlags,
   UINT uCount,
   DWORD dwTimeout
);
```

Parameters

dwFlags

Specifies the flash status. For a complete list of values, see the FLASHWINFO structure.

uCount

Specifies the number of times to flash the window.

dwTimeout

Specifies the rate, in milliseconds, at which the window will be flashed. If *dwTimeout* is zero, the function uses the default cursor blink rate.

Return Value

Nonzero if the function succeeds; otherwise 0.

Remarks

This member function emulates the functionality of the function FlashWindowEx, as described in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::FlashWindow

Concepts

CWnd Members

CWnd::FromHandle

Returns a pointer to a **CWnd** object when given a handle to a window. If a **CWnd** object is not attached to the handle, a temporary **CWnd** object is created and attached.

```
static CWnd* PASCAL FromHandle(
   HWND hWnd
);
```

Parameters

hWnd

An **HWND** of a Windows window.

Return Value

Returns a pointer to a **CWnd** object when given a handle to a window. If a **CWnd** object is not attached to the handle, a temporary **CWnd** object is created and attached.

The pointer may be temporary and should not be stored for later use.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::DeleteTempMap

Concepts

CWnd Members

CWnd::FromHandlePermanent

Returns a pointer to a **CWnd** object when given a handle to a window.

```
static CWnd* PASCAL FromHandlePermanent(
   HWND hWnd
);
```

Parameters

hWnd

An **HWND** of a Windows window.

Return Value

A pointer to a **CWnd** object.

Remarks

If a **CWnd** object is not attached to the handle, **NULL** is returned.

This function, unlike FromHandle, does not create temporary objects.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::FromHandle

Concepts

CWnd Members

CWnd::get_accChild

Called by the framework to retrieve the address of an **IDispatch** interface for the specified child.

```
virtual HRESULT get_accChild(
   VARIANT varChild,
   IDispatch **ppdispChild
);
```

Parameters

varChild

Identifies the child whose **IDispatch** interface is to be retrieved.

ppdispChild

Receives the address of the child object's **IDispatch** interface.

Return Value

Returns S_OK on success, a COM error code on failure. See **Return Values** in IAccessible::get_accChild in the *Platform SDK*.

Remarks

This function is part of MFC's Active Accessibility support.

Override this function in your **CWnd**-derived class if you have nonwindowed user interface elements (other than windowless ActiveX controls, which MFC handles).

For more information, seeIAccessible::get_accChild in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::get_accChildCount
CWnd::get_accParent

Concepts

CWnd Members

CWnd::get_accChildCount

Called by the framework to retrieve the number of children belonging to this object.

```
virtual HRESULT get_accChildCount(
   long *pcountChildren
);
```

Parameters

pcountChildren

Receives the number of children.

Return Value

Returns S_OK on success, a COM error code on failure. See **Return Values** in IAccessible::get_accChildCount in the *Platform SDK*.

Remarks

This function is part of MFC's Active Accessibility support.

Override this function in your **CWnd**-derived class if you have nonwindowed user interface elements (other than windowless ActiveX controls, which MFC handles). Call the base class version and then add the nonwindowed child elements.

For more information, see IAccessible::get_accChildCount in the *Platform SDK*.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::get_accChild

Concepts

CWnd Members

CWnd::get_accDefaultAction

Called by the framework to retrieve a string that describes the object's default action.

```
virtual HRESULT get_accDefaultAction(
   VARIANT varChild,
   BSTR *pszDefaultAction
);
```

Parameters

varChild

Specifies whether the default action to be retrieved is that of the object or one of the object's child elements. This parameter can be either CHILDID_SELF (to obtain information about the object) or a child ID (to obtain information about the object's child element).

pszDefaultAction

Address of a **BSTR** that receives a localized string describing the default action for the specified object, or NULL if this object has no default action.

Return Value

Returns S_OK on success, a COM error code on failure. See **Return Values** in IAccessible::get_accDefaultAction in the *Platform SDK*.

Remarks

This function is part of MFC's Active Accessibility support.

Override this function in your **CWnd**-derived class to describe your object's default action.

For more information, see IAccessible::get_accDefaultAction in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::accDoDefaultAction

Concepts

CWnd Members

CWnd::get_accDescription

Called by framework to retrieve a string that describes the visual appearance of the specified object.

```
virtual HRESULT get_accDescription(
   VARIANT varChild,
   BSTR *pszDescription
);
```

Parameters

varChild

Specifies whether the description to be retrieved is that of the object or one of the object's child elements. This parameter can be either CHILDID_SELF (to obtain information about the object) or a child ID (to obtain information about the object's child element).

pszDescription

Address of a **BSTR** that receives a localized string describing the specified object, or NULL if no description is available for this object.

Return Value

Returns S_OK on success, a COM error code on failure. See **Return Values** in IAccessible::get_accDescription in the *Platform SDK*.

Remarks

This function is part of MFC's Active Accessibility support.

Override this function in your **CWnd**-derived class to describe your object. Call the base class version and add your description.

For more information, see IAccessible::get_accDescription in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::get_accHelp
CWnd::get_accName
CWnd::get_accRole
CWnd::get_accState
CWnd::get_accValue

Concepts

CWnd Members

CWnd::get_accFocus

Called by the framework to retrieve the object that has the keyboard focus.

```
virtual HRESULT get_accFocus(
   VARIANT *pvarChild
);
```

Parameters

pvarChild

Receives information about the object that has the focus. See *pvarID* inIAccessible::get_accFocus in the *Platform SDK*.

Return Value

Returns S_OK on success, a COM error code on failure. See **Return Values** in **IAccessible::get_accFocus** in the *Platform SDK*.

Remarks

This function is part of MFC's Active Accessibility support.

Override this function in your **CWnd**-derived class if you have nonwindowed user interface elements (other than windowless ActiveX controls, which MFC handles).

For more information, see IAccessible::get_accFocus in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::get_accSelection
CWnd::accSelect

Concepts

CWnd Members

CWnd::get_accHelp

Called by the framework to retrieve an object's **Help** property string.

```
virtual HRESULT get_accHelp(
   VARIANT varChild,
   BSTR *pszHelp
);
```

Parameters

varChild

Specifies whether the help information to be retrieved is that of the object or one of the object's child elements. This parameter can be either CHILDID_SELF (to obtain information about the object) or a child ID (to obtain information about the object's child element).

pszHelp

Address of a **BSTR** that receives the localized string containing the help information for the specified object, or NULL if no help information is available.

Return Value

Returns S_OK on success, a COM error code on failure. See **Return Values** in IAccessible::get_accHelp in the *Platform SDK*.

Remarks

This function is part of MFC's Active Accessibility support.

Override this function in your **CWnd**-derived class to provide help text for your object.

For more information, see IAccessible::get_accHelp in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::get_accDescription
CWnd::get_accHelpTopic

Concepts

CWnd Members

CWnd::get_accHelpTopic

Called by the framework to retrieve the full path of the **WinHelp** file associated with the specified object and the identifier of the appropriate topic within that file.

```
virtual HRESULT get_accHelpTopic(
   BSTR *pszHelpFile,
   VARIANT varChild,
   long *pidTopic
);
```

Parameters

pszHelpFile

Address of a **BSTR** that receives the full path of the **WinHelp** file associated with the specified object, if any.

varChild

Specifies whether the Help topic to be retrieved is that of the object or one of the object's child elements. This parameter can be either CHILDID_SELF (to obtain a Help topic for the object) or a child ID (to obtain a Help topic for one of the object's child elements).

pidTopic

Identifies the Help file topic associated with the specified object. See *pidTopic* in IAccessible::get_accHelpTopic in the *Platform SDK*.

Return Value

Returns S_OK on success, a COM error code on failure. See **Return Values** in **IAccessible::get_accHelpTopic** in the *Platform SDK*.

Remarks

This function is part of MFC's Active Accessibility support.

Override this function in your **CWnd**-derived class to provide help information about your object.

For more information, seeIAccessible::get_accHelpTopic in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::get_accDescription
CWnd::get_accHelp

Concepts

CWnd Members

CWnd::get_accKeyboardShortcut

Called by the framework to retrieve the specified object's shortcut key or access key.

```
virtual HRESULT get_accKeyboardShortcut(
   VARIANT varChild,
   BSTR *pszKeyboardShortcut
);
```

Parameters

varChild

Specifies whether the keyboard shortcut to be retrieved is that of the object or one of the object's child elements. This parameter can be either CHILDID_SELF (to obtain information about the object) or a child ID (to obtain information about the object's child element).

pszKeyboardShortcut

Address of a **BSTR** that receives a localized string identifying the keyboard shortcut, or NULL if no keyboard shortcut is associated with the specified object.

Return Value

Returns S_OK on success, a COM error code on failure. See **Return Values** in IAccessible::get_accKeyboardShortcut in the *Platform SDK*.

Remarks

This function is part of MFC's Active Accessibility support.

Override this function in your **CWnd**-derived class to identify the keyboard shortcut for your object.

For more information, see IAccessible::get_accKeyboardShortcut in the *Platform SDK*.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::get_accName

Called by the framework to retrieve the name of the specified object.

```
virtual HRESULT get_accName(
   VARIANT varChild,
   BSTR *pszName
);
```

Parameters

varChild

Specifies whether the name to be retrieved is that of the object or one of the object's child elements. This parameter can be either CHILDID_SELF (to obtain information about the object) or a child ID (to obtain information about the object's child element).

pszName

Address of a **BSTR** that receives a string containing the specified object's name.

Return Value

Returns S_OK on success, a COM error code on failure. See **Return Values** in IAccessible::get_accName in the *Platform SDK*.

Remarks

This function is part of MFC's Active Accessibility support.

Override this function in your **CWnd**-derived class to return the name of your object.

For more information, see IAccessible::get_accName in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::get_accKeyboardShortcut
CWnd::get_accDescription

Concepts

CWnd Members

CWnd::get_accParent

Called by the framework to retrieve the **IDispatch** interface of the object's parent.

```
virtual HRESULT get_accParent(
   IDispatch **ppdispParent
);
```

Parameters

ppdispParent

Receives the address of the parent object's **IDispatch** interface. The variable is set to NULL if no parent exists, or if the child cannot access its parent.

Return Value

Returns S_OK on success, a COM error code on failure. See **Return Values** in IAccessible::get_accParent in the *Platform SDK*.

Remarks

This function is part of MFC's Active Accessibility support.

In most cases you don't have to override this function.

For more information, see IAccessible::get_accParent in the *Platform SDK*.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::get_accChild

Concepts

CWnd Members

CWnd::get_accRole

Called by the framework to retrieve information that describes the role of the specified object.

```
virtual HRESULT get_accRole(
   VARIANT varChild,
   VARIANT *pvarRole
);
```

Parameters

varChild

Specifies whether the role information to be retrieved is that of the object or one of the object's child elements. This parameter can be either CHILDID_SELF (to obtain information about the object) or a child ID (to obtain information about the object's child element).

pvarRole

Receives the role information. See *pvarRole* in IAccessible::get_accRole in the *Platform SDK*.

Return Value

Returns S_OK on success, a COM error code on failure. See **Return Values** in **IAccessible::get_accRole** in the *Platform SDK*.

Remarks

This function is part of MFC's Active Accessibility support.

Override this function in your **CWnd**-derived class if you have nonwindowed user interface elements (other than windowless ActiveX controls, which MFC handles).

For more information, see IAccessible::get_accRole in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
GetRoleText
CWnd::get_accDescription

Concepts

CWnd Members

CWnd::get_accSelection

Called by the framework to retrieve the selected children of this object.

```
virtual HRESULT get_accSelection(
   VARIANT *pvarChildren
);
```

Parameters

pvarChildren

Receives information about which children are selected. See *pvarChildren* in IAccessible::get_accSelection in the *Platform SDK*.

Return Value

Returns S_OK on success, a COM error code on failure. See **Return Values** in **IAccessible::get_accSelection** in the *Platform SDK*.

Remarks

This function is part of MFC's Active Accessibility support.

Override this function in your **CWnd**-derived class if you have nonwindowed user interface elements (other than windowless ActiveX controls, which MFC handles).

For more information, see IAccessible::get_accSelection in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::accSelect
CWnd::get_accFocus

Concepts

CWnd Members

CWnd::get_accState

Called by the framework to retrieve the current state of the specified object.

```
virtual HRESULT get_accState(
   VARIANT varChild,
   VARIANT *pvarState
);
```

Parameters

varChild

Specifies whether the state information to be retrieved is that of the object or one of the object's child elements. This parameter can be either CHILDID_SELF (to obtain information about the object) or a child ID (to obtain information about the object's child element).

pvarState

Receives information about the object's state. See *pvarState* in IAccessible::get_accState in the *Platform SDK*.

Return Value

Returns S_OK on success, a COM error code on failure. See **Return Values** in **IAccessible::get_accState** in the *Platform SDK*.

Remarks

This function is part of MFC's Active Accessibility support.

Override this function in your **CWnd**-derived class if you have nonwindowed user interface elements (other than windowless ActiveX controls, which MFC handles).

For more information, see IAccessible::get_accState in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::get_accDescription

Concepts

CWnd Members

CWnd::get_accValue

Called by the framework to retrieve the value of the specified object.

```
virtual HRESULT get_accValue(
   VARIANT varChild,
   BSTR *pszValue
);
```

Parameters

varChild

Specifies whether the value information to be retrieved is that of the object or one of the object's child elements. This parameter can be either CHILDID_SELF (to obtain information about the object) or a child ID (to obtain information about the object's child element).

pszValue

Address of the **BSTR** that receives a localized string containing the object's current value.

Return Value

Returns S_OK on success, a COM error code on failure. See **Return Values** in IAccessible::get_accValue in the *Platform SDK*.

Remarks

This function is part of MFC's Active Accessibility support.

Override this function in your **CWnd**-derived class if you have nonwindowed user interface elements (other than windowless ActiveX controls, which MFC handles).

For more information, see IAccessible::get_accValue in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::get_accDescription

Concepts

CWnd Members

CWnd::GetActiveWindow

Retrieves a pointer to the active window.

```
static CWnd* PASCAL GetActiveWindow( );
```

Return Value

The active window or **NULL** if no window was active at the time of the call. The pointer may be temporary and should not be stored for later use.

Remarks

The active window is either the window that has the current input focus or the window explicitly made active by the SetActiveWindow member function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::SetActiveWindow
GetActiveWindow

Concepts

CWnd Members

CWnd::GetAncestor

Retrieves the ancestor window object of the specified window.

```
CWnd* GetAncestor(
      UINT gaFlags
) const;
```

Parameters

gaFlags

Specifies the ancestor to be retrieved. For a complete list of possible values, see GetAncestor.

Return Value

If the function succeeds, the return value is a pointer to the ancestor window object. If the function fails, the return value is **NULL**.

Remarks

This member function emulates the functionality of the function GetAncestor, as described in the *Platform SDK*.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::GetCapture

Retrieves the window that has the mouse capture.

```
static CWnd* PASCAL GetCapture( );
```

Return Value

Identifies the window that has the mouse capture. It is **NULL** if no window has the mouse capture.

The return value may be temporary and should not be stored for later use.

Remarks

Only one window has the mouse capture at any given time. A window receives the mouse capture when the SetCapture member function is called. This window receives mouse input whether or not the cursor is within its borders.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::SetCapture
GetCapture

Concepts

CWnd Members

CWnd::GetCaretPos

Retrieves the client coordinates of the caret's current position and returns them as a **CPoint**.

```
static CPoint PASCAL GetCaretPos( );
```

Return Value

CPoint object containing the coordinates of the caret's position.

Remarks

The caret position is given in the client coordinates of the **CWnd** window.

See Also

Reference

CWnd Class

Hierarchy Chart

GetCaretPos

Concepts

CWnd Members

CWnd::GetCheckedRadioButton

Retrieves the ID of the currently checked radio button in the specified group.

```
int GetCheckedRadioButton(
   int nIDFirstButton,
   int nIDLastButton
);
```

Parameters

nIDFirstButton

Specifies the integer identifier of the first radio button in the group.

nIDLastButton

Specifies the integer identifier of the last radio button in the group.

Return Value

ID of the checked radio button, or 0 if none is selected.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::CheckRadioButton

Concepts

CWnd Members

CWnd::GetClientRect

Copies the client coordinates of the **CWnd** client area into the structure pointed to by *lpRect*.

```
void GetClientRect(
   LPRECT lpRect
) const;
```

Parameters

lpRect

Points to a RECT structure or a **CRect** object to receive the client coordinates. The **left** and **top** members will be 0. The **right** and **bottom** members will contain the width and height of the window.

Remarks

The client coordinates specify the upper-left and lower-right corners of the client area. Since client coordinates are relative to the upper-left corners of the **CWnd** client area, the coordinates of the upper-left corner are (0,0).

Example

See the example for CWnd::IsIconic.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetWindowRect
GetClientRect

Concepts

CWnd Members

CWnd::GetClipboardOwner

Retrieves the current owner of the Clipboard.

```
static CWnd* PASCAL GetClipboardOwner( );
```

Return Value

Identifies the window that owns the Clipboard if the function is successful. Otherwise, it is **NULL**.

The returned pointer may be temporary and should not be stored for later use.

Remarks

The Clipboard can still contain data even if it is not currently owned.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetClipboardViewer
GetClipboardOwner

Concepts

CWnd Members

CWnd::GetClipboardViewer

Retrieves the first window in the Clipboard-viewer chain.

```
static CWnd* PASCAL GetClipboardViewer( );
```

Return Value

Identifies the window currently responsible for displaying the Clipboard if successful; otherwise **NULL** (for example, if there is no viewer).

The returned pointer may be temporary and should not be stored for later use.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetClipboardOwner
GetClipboardViewer

Concepts

CWnd Members

CWnd::GetControlUnknown

Call this member function to retrieve a pointer to an unknown OLE control.

```
LPUNKNOWN GetControlUnknown( );
```

Return Value

A pointer to the IUnknown interface of the OLE control represented by this **CWnd** object. If this object does not represent an OLE control, the return value is **NULL**.

Remarks

You should not release this **IUnknown** pointer. Typically, you would use to obtain a specific interface of the control.

The interface pointer returned by **GetControlUnknown** is not reference-counted. Do not call IUnknown::Release on the pointer unless you have previously called IUnknown::AddRef on it.

Example

```
BOOL CMytDlg::OnInitDialog()
{
   CDialog::OnInitDialog();

   // IDC_MSACALCTRL1 is the ID of the Calendar control OCX embedded
   // on this dialog
   CWnd *pWndCal = GetDlgItem(IDC_MSACALCTRL1);

   // Use the IUnknown of the control
   LPUNKNOWN pUnk = pWndCal->GetControlUnknown();

   // From there get the IDispatch interface of control
   LPDISPATCH pDisp = NULL;
   pUnk->QueryInterface(IID_IDispatch, (LPVOID*)&pDisp);

   // use IDispatch method to invoke the control's functionality

   return TRUE;  // return TRUE unless you set the focus to a control
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
IUnknown::Release
IUnknown::QueryInterface

Concepts

CWnd Members

CWnd::GetCurrentMessage

Returns a pointer to the message this window is currently processing. Should only be called when in an **On***Message* message-handler member function.

```
static const MSG* PASCAL GetCurrentMessage( );
```

Return Value

Returns a pointer to the MSG structure that contains the message the window is currently processing. Should only be called when in an **On***Message* handler.

Example

See the example for CMDIFrameWnd::MDICascade.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::GetDC

Retrieves a pointer to a common, class, or private device context for the client area depending on the class style specified for the **CWnd**.

```
CDC* GetDC( );
```

Return Value

Identifies the device context for the **CWnd** client area if successful; otherwise, the return value is **NULL**. The pointer may be temporary and should not be stored for later use.

Remarks

For common device contexts, **GetDC** assigns default attributes to the context each time it is retrieved. For class and private contexts, **GetDC** leaves the previously assigned attributes unchanged. The device context can be used in subsequent graphics device interface (GDI) functions to draw in the client area.

Unless the device context belongs to a window class, the ReleaseDC member function must be called to release the context after painting.

A device context belonging to the **CWnd** class is returned by the **GetDC** member function if **CS_CLASSDC**, **CS_OWNDC**, or **CS_PARENTDC** was specified as a style in the **WNDCLASS** structure when the class was registered.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetDCEx
CWnd::ReleaseDC
CWnd::GetWindowDC
GetDC
CClientDC Class

Concepts

CWnd Members

CWnd::GetDCEx

Retrieves the handle of a device context for the **CWnd** window.

```
CDC* GetDCEx(
   CRgn* prgnClip,
   DWORD flags
);
```

Parameters

prgnClip

Identifies a clipping region that may be combined with the visible region of the client window.

flags

Can have one of the following preset values:

- **DCX_CACHE** Returns a device context from the cache rather than the **OWNDC** or **CLASSDC** window. Overrides **CS_OWNDC** and **CS_CLASSDC**.
- **DCX_CLIPCHILDREN** Excludes the visible regions of all child windows below the **CWnd** window.
- **DCX_CLIPSIBLINGS** Excludes the visible regions of all sibling windows above the **CWnd** window.
- **DCX_EXCLUDERGN** Excludes the clipping region identified by *prgnClip* from the visible region of the returned device context.
- **DCX_INTERSECTRGN** Intersects the clipping region identified by *prgnClip* within the visible region of the returned device context.
- **DCX_LOCKWINDOWUPDATE** Allows drawing even if there is a **LockWindowUpdate** call in effect that would otherwise exclude this window. This value is used for drawing during tracking.
- **DCX_PARENTCLIP** Uses the visible region of the parent window and ignores the parent window's **WS_CLIPCHILDREN** and **WS_PARENTDC** style bits. This value sets the device context's origin to the upper-left corner of the **CWnd** window.
- **DCX_WINDOW** Returns a device context that corresponds to the window rectangle rather than the client rectangle.

Return Value

The device context for the specified window if the function is successful; otherwise **NULL**.

Remarks

The device context can be used in subsequent GDI functions to draw in the client area.

This function, which is an extension to the GetDC function, gives an application more control over how and whether a device context for a window is clipped.

Unless the device context belongs to a window class, the ReleaseDC function must be called to release the context after drawing. Since only five common device contexts are available at any given time, failure to release a device context can prevent other applications from gaining access to a device context.

To obtain a cached device context, an application must specify DCX_CACHE. If **DCX_CACHE** is not specified and the window is neither **CS_OWNDC** nor CS_CLASSDC, this function returns **NULL**.

A device context with special characteristics is returned by the GetDCEx function if the **CS_CLASSDC**, CS_OWNDC, or CS_PARENTDC style was specified in the WNDCLASS structure when the class was registered.

For more information about these characteristics, see the description of the **WNDCLASS** structure in the *Platform SDK*.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::BeginPaint
CWnd::GetDC
CWnd::GetWindowDC
CWnd::ReleaseDC
GetDCEx

Concepts

CWnd Members

CWnd::GetDescendantWindow

Call this member function to find the descendant window specified by the given ID.

```
CWnd* GetDescendantWindow(
   int nID,
   BOOL bOnlyPerm = FALSE
) const;
```

Parameters

nID

Specifies the identifier of the control or child window to be retrieved.

bOnlyPerm

Specifies whether the window to be returned can be temporary. If **TRUE**, only a permanent window can be returned; if **FALSE,** the function can return a temporary window. For more information on temporary windows see Technical Note 3.

Return Value

A pointer to a **CWnd** object, or **NULL** if no child window is found.

Remarks

This member function searches the entire tree of child windows, not only the windows that are immediate children.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetParentFrame
CWnd::IsChild
CWnd::GetDlgItem

Concepts

CWnd Members

CWnd::GetDesktopWindow

Returns the Windows desktop window.

```
static CWnd* PASCAL GetDesktopWindow( );
```

Return Value

Identifies the Windows desktop window. This pointer may be temporary and should not be stored for later use.

Remarks

The desktop window covers the entire screen and is the area on top of which all icons and other windows are painted.

See Also

Reference

CWnd Class

Hierarchy Chart

GetDesktopWindow

Concepts

CWnd Members

CWnd::GetDlgCtrlID

Returns the window or control ID value for any child window, not only that of a control in a dialog box.

```
int GetDlgCtrlID( ) const;
```

Return Value

The numeric identifier of the **CWnd** child window if the function is successful; otherwise 0.

Remarks

Since top-level windows do not have an ID value, the return value of this function is invalid if the **CWnd** is a top-level window.

Example

See the example for CWnd::OnCtlColor.

See Also

Reference

CWnd Class
Hierarchy Chart
GetDlgCtrlID

Concepts

CWnd Members

CWnd::GetDlgItem

Retrieves a pointer to the specified control or child window in a dialog box or other window.

```
CWnd* GetDlgItem(
   int nID
) const;
void GetDlgItem(
   int nID,
   HWND* phWnd
) const;
```

Parameters

nID

Specifies the identifier of the control or child window to be retrieved.

phWnd

A pointer to a child window.

Return Value

A pointer to the given control or child window. If no control with the integer ID given by the *nID* parameter exists, the value is **NULL**.

The returned pointer may be temporary and should not be stored for later use.

Remarks

The pointer returned is usually cast to the type of control identified by *nID*.

Example

```
// uses GetDlgItem to return a pointer to a user interface control
CEdit* pBoxOne;
pBoxOne = (CEdit*) GetDlgItem(IDC_EDIT1);
GotoDlgCtrl(pBoxOne);
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetWindow
CWnd::GetDescendantWindow
CWnd::GetWindow
GetDlgItem

Concepts

CWnd Members

CWnd::GetDlgItemInt

Retrieves the text of the control identified by *nID*.

```
UINT GetDlgItemInt(
   int nID,
   BOOL* lpTrans = NULL,
   BOOL bSigned = TRUE
) const;
```

Parameters

nID

Specifies the integer identifier of the dialog-box control to be translated.

lpTrans

Points to the Boolean variable that is to receive the translated flag.

bSigned

Specifies whether the value to be retrieved is signed.

Return Value

Specifies the translated value of the dialog-box item text. Since 0 is a valid return value, *lpTrans* must be used to detect errors. If a signed return value is desired, cast it as an **int** type.

The function returns 0 if the translated number is greater than INT_MAX (for signed numbers) or UINT_MAX (for unsigned).

When errors occur, such as encountering nonnumeric characters and exceeding the above maximum, **GetDlgItemInt** copies 0 to the location pointed to by *lpTrans*. If there are no errors, *lpTrans* receives a nonzero value. If *lpTrans* is **NULL**, **GetDlgItemInt** does not warn about errors.

Remarks

It translates the text of the specified control in the given dialog box into an integer value by stripping any extra spaces at the beginning of the text and converting decimal digits. It stops the translation when it reaches the end of the text or encounters any nonnumeric character.

If *bSigned* is **TRUE**, **GetDlgItemInt** checks for a minus sign (–) at the beginning of the text and translates the text into a signed number. Otherwise, it creates an unsigned value.

It sends a WM_GETTEXT message to the control.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetDlgItemText
GetDlgItemInt

Concepts

CWnd Members

CWnd::GetDlgItemText

Call this member function to retrieve the title or text associated with a control in a dialog box.

```
int GetDlgItemText(
   int nID,
   LPTSTR lpStr,
   int nMaxCount
) const;
int GetDlgItemText(
   int nID,
   CString& rString
) const;
```

Parameters

nID

Specifies the integer identifier of the control whose title is to be retrieved.

lpStr

Points to the buffer to receive the control's title or text.

nMaxCount

Specifies the maximum length (in characters) of the string to be copied to *lpStr*. If the string is longer than *nMaxCount*, it is truncated.

rString

A reference to a CString.

Return Value

Specifies the actual number of characters copied to the buffer, not including the terminating null character. The value is 0 if no text is copied.

Remarks

The **GetDlgItemText** member function copies the text to the location pointed to by *lpStr* and returns a count of the number of bytes it copies.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetDlgItem
CWnd::GetDlgItemInt
GetDlgItemText
WM_GETTEXT

Concepts

CWnd Members

CWnd::GetDSCCursor

Call this member function to retrieve a pointer to the underlying cursor that is defined by the DataSource, UserName, Password, and SQL properties of the data-source control.

```
IUnknown * GetDSCCursor( );
```

Return Value

A pointer to a cursor that is defined by a data-source control. MFC takes care of calling **AddRef** for the pointer.

Remarks

Use the returned pointer to set the ICursor property of a complex data-bound control, such as the data-bound grid control. A data-source control will not become active until the first bound control requests its cursor. This can happen either explicitly by a call to **GetDSCCursor** or implicitly by the MFC binding manager. In either case, you can force a data-source control to become active by calling **GetDSCCursor** and then calling **Release** on the returned pointer to **IUnknown**. Activation will cause the data-source control to attempt to connect to the underlying data source. The returned pointer might be used in the following context:

Example

```
BOOL CMyDlg::OnInitDialog()
{
   // Find the child controls on the dialog
   CWnd* pDSC = GetDlgItem(IDC_REMOTEDATACONTROL);
   CDBListBox* pList = (CDBListBox*)
   GetDlgItem(IDC_DBLISTBOX);

   // Tell the MFC binding manager that we are
   // binding DISPID 3 to the data-source control.
   pList->BindProperty(0x3, pDSC);

   // Tell the listbox which field to expose as its
   // bound column
   pList->SetBoundColumn(_T("CourseID"));

   // Tell the listbox which cursor and column
   // to populate its list from
   pList->SetListField(_T("CourseID"));
   IPUNKNOWN *pcursor = pDSC->GetDSCCursor();
   ...
   if (!pcursor)
   {
   // The pointer was not successfully assigned.
      return FALSE;
   }
   // The pointer was successfully assigned,
   pList->SetRowSource(pcursor);
   ...
   pcursor->Release();
   return TRUE;
}
```

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::BindDefaultProperty

CWnd::BindProperty

Concepts

CWnd Members

CWnd::GetExStyle

Returns the window's extended style.

```
DWORD GetExStyle( ) const;
```

Return Value

The window's extended style. For more information about the extended window styles used in MFC, see Extended Window Styles.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetStyle
GetWindowLong

Concepts

CWnd Members

CWnd::GetFocus

Retrieves a pointer to the **CWnd** that currently has the input focus.

```
static CWnd* PASCAL GetFocus( );
```

Return Value

A pointer to the window that has the current focus, or **NULL** if there is no focus window.

The pointer may be temporary and should not be stored for later use.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetActiveWindow
CWnd::GetCapture
CWnd::SetFocus
GetFocus

Concepts

CWnd Members

CWnd::GetFont

Gets the current font for this window.

```
CFont* GetFont( ) const;
```

Return Value

A pointer to a CFont that contains the current font.

The pointer may be temporary and should not be stored for later use.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::SetFont
WM_GETFONT
CFont Class

Concepts

CWnd Members

CWnd::GetForegroundWindow

Returns a pointer to the foreground window (the window with which the user is currently working).

```
static CWnd* PASCAL GetForegroundWindow( );
```

Return Value

A pointer to the foreground window. This may be a temporary **CWnd** object.

Remarks

The foreground window applies only to top-level windows (frame windows or dialog boxes).

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::SetForegroundWindow

Concepts

CWnd Members

CWnd::GetIcon

Call this member function to get the handle to either a big (32x32) or the handle to a small (16x16) icon, as indicated by *bBigIcon*.

```
HICON GetIcon(
   BOOL bBigIcon
) const;
```

Parameters

bBigIcon

Specifies a 32 pixel by 32 pixel icon if **TRUE**; specifies a 16 pixel by 16 pixel icon if **FALSE**.

Return Value

A handle to an icon. If unsuccessful, returns **NULL**.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::SetIcon

Concepts

CWnd Members

CWnd::GetLastActivePopup

Determines which pop-up window owned by **CWnd** was most recently active.

```
CWnd* GetLastActivePopup( ) const;
```

Return Value

Identifies the most recently active pop-up window. The return value will be the window itself if any of the following conditions are met:

- The window itself was most recently active.
- The window does not own any pop-up windows.
- The window is not a top-level window or is owned by another window.

The pointer may be temporary and should not be stored for later use.

Example

See the example for CWnd::FindWindow.

See Also

Reference

CWnd Class
Hierarchy Chart
GetLastActivePopup

Concepts

CWnd Members

CWnd::GetLayeredWindowAttributes

Retrieves the opacity and transparency color key of a layered window.

```
BOOL GetLayeredWindowAttributes(
   COLORREF *pcrKey,
   BYTE *pbAlpha,
   DWORD *pdwFlags
) const;
```

Parameters

pcrKey

Pointer to a **COLORREF** value that receives the transparency color key to be used when composing the layered window. All pixels painted by the window in this color will be transparent. This can be **NULL** if the argument is not needed.

pbAlpha

Pointer to a **BYTE** that receives the Alpha value used to describe the opacity of the layered window. When the variable referred to by *pbAlpha* is 0, the window is completely transparent. When the variable referred to by *pbAlpha* is 255, the window is opaque. This can be **NULL** if the argument is not needed.

pdwFlags

Pointer to a **DWORD** that receives a layering flag. This can be **NULL** if the argument is not needed. For a complete list of possible values, see GetLayeredWindowAttributes.

Return Value

Nonzero if the function succeeds; otherwise 0.

Remarks

This member function emulates the functionality of the function GetLayeredWindowAttributes, as described in the *Platform SDK*.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::GetMenu

Retrieves a pointer to the menu for this window.

```
CMenu* GetMenu( ) const;
```

Return Value

Identifies the menu. The value is **NULL** if **CWnd** has no menu. The return value is undefined if **CWnd** is a child window.

The returned pointer may be temporary and should not be stored for later use.

Remarks

This function should not be used for child windows because they do not have a menu.

Example

```
// This example deletes the leftmost popup menu or leftmost
// popup menu item from the application's main window.
CWnd* pMain = AfxGetMainWnd();

// The main window _can_ be NULL, so this code
// doesn't ASSERT and actually tests.
if (pMain != NULL)
{
   // Get the main window's menu
   CMenu* pMenu = pMain->GetMenu();

   // If there is a menu and it has items, we'll
   // delete the first one.
   if (pMenu != NULL && pMenu->GetMenuItemCount() > 0)
   {
      pMenu->DeleteMenu(0, MF_BYPOSITION);
      // force a redraw of the menu bar
      pMain->DrawMenuBar();
   }

   // No need to delete pMenu because it is an MFC
   // temporary object.
}
```

See Also

Reference

CWnd Class

Hierarchy Chart

GetMenu

Concepts

CWnd Members

CWnd::GetMenuBarInfo

Retrieves information about the specified menu bar.

```
BOOL GetMenuBarInfo(
   LONG idObject,
   LONG idItem,
   PMENUBARINFO pmbi
) const;
```

Parameters

idObject

Specifies the menu object. For a list of possible values, see GetMenuBarInfo.

idItem

Specifies the item for which to retrieve information. If this parameter is zero, the function retrieves information about the menu itself. If this parameter is 1, the function retrieves information about the first item on the menu, and so on.

pmbi

Pointer to a MENUBARINFO structure that receives the information.

Return Value

Nonzero if the function succeeds; otherwise 0.

Remarks

This member function emulates the functionality of the function GetMenuBarInfo, as described in the *Platform SDK*.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::GetNextDlgGroupItem

Searches for the previous or next control within a group of controls in a dialog box.

```
CWnd* GetNextDlgGroupItem(
   CWnd* pWndCtl,
   BOOL bPrevious = FALSE
) const;
COleControlSiteOrWnd* GetNextDlgGroupItem(
   COleControlSiteOrWnd *pCurSiteOrWnd = NULL
) const;
```

Parameters

pWndCtl

Identifies the control to be used as the starting point for the search.

bPrevious

Specifies how the function is to search the group of controls in the dialog box. If **TRUE**, the function searches for the previous control in the group; if **FALSE**, it searches for the next control in the group.

pCurSiteOrWnd

Identifies the **COleControlSiteOrWnd** control. For more information about **COleControlSiteOrWnd**, see **Remarks**.

Return Value

Pointer to the previous or next control in the group if the member function is successful.

The returned pointer may be temporary and should not be stored for later use.

Remarks

A group of controls begins with a control that was created with the WS_GROUP style and ends with the last control that was not created with the **WS_GROUP** style.

By default, the **GetNextDlgGroupItem** member function returns a pointer to the next control in the group. If *pWndCtl* identifies the first control in the group and *bPrevious* is **TRUE**, **GetNextDlgGroupItem** returns a pointer to the last control in the group.

Note
Because MFC supports windowless ActiveX controls, standard ActiveX controls, and windows, referring to a control by only a n HWND no longer suffices. The **COleControlSiteOrWnd** object includes information that identifies the object as a windowe d ActiveX control, a windowless ActiveX control, or a window, as follows:

Control or w indow type	**Identifying information**
Windowed A ctiveX control	Contains an HWND and associates a COleControlSite object with it. The **m_hWnd** member of **COleControlSite OrWnd** is set to the HWND of the control, and the **m_pSite** member points to the control's **COleControlSite**.
Windowless ActiveX contr ol	Contains no **HWND**. The **m_pSite** member of **COleControlSiteOrWnd** points to the control's **COleControlSit e**, and the **m_hWnd** member is **NULL**.
Standard win dow	Contains just an **HWND**. The **m_hWnd** member of **COleControlSiteOrWnd** is set to the **HWND** of the windo w, and the **m_pSite** member is **NULL**.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::GetNextDlgTabItem
GetNextDlgGroupItem
Concepts
CWnd Members

CWnd::GetNextDlgTabItem

Retrieves a pointer to the first control that was created with the WS_TABSTOP style and that precedes or follows the specified control.

```
CWnd* GetNextDlgTabItem(
   CWnd* pWndCtl,
   BOOL bPrevious = FALSE
) const;
COleControlSiteOrWnd* GetNextDlgTabItem(
   COleControlSiteOrWnd *pCurSiteOrWnd,
   BOOL bPrevious
) const;
```

Parameters

pWndCtl

Identifies the control to be used as the starting point for the search.

pCurSiteOrWnd

Identifies the **COleControlSiteOrWnd** control. For more information about **COleControlSiteOrWnd**, see CWnd::GetNextDlgGroupItem.

bPrevious

Specifies how the function is to search the dialog box. If **TRUE**, the function searches for the previous control in the dialog box; if **FALSE**, it searches for the next control.

Return Value

Pointer to the previous or next control that has the **WS_TABSTOP** style, if the member function is successful.

The returned pointer may be temporary and should not be stored for later use.

For more information about **COleControlSiteOrWnd**, see CWnd::GetNextDlgGroupItem.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetNextDlgGroupItem
GetNextDlgTabItem

Concepts

CWnd Members

CWnd::GetNextWindow

Searches for the next (or previous) window in the window manager's list.

```
CWnd* GetNextWindow(
   UINT nFlag = GW_HWNDNEXT
) const;
```

Parameters

nFlag

Specifies whether the function returns a pointer to the next window or the previous window. It can be either **GW_HWNDNEXT**, which returns the window that follows the **CWnd** object on the window manager's list, or **GW_HWNDPREV**, which returns the previous window on the window manager's list.

Return Value

Identifies the next (or the previous) window in the window manager's list if the member function is successful.

The returned pointer may be temporary and should not be stored for later use.

Remarks

The window manager's list contains entries for all top-level windows, their associated child windows, and the child windows of any child windows.

If **CWnd** is a top-level window, the function searches for the next (or previous) top-level window; if **CWnd** is a child window, the function searches for the next (or previous) child window.

See Also

Reference

CWnd Class
Hierarchy Chart
GetNextWindow

Concepts

CWnd Members

CWnd::GetOleControlSite

Retrieves the custom site for the specified ActiveX control.

```
COleControlSite* GetOleControlSite(
   UINT idControl
) const;
```

Parameters

idControl

The ID of the ActiveX control.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::GetOpenClipboardWindow

Retrieves the handle of the window that currently has the Clipboard open.

```
static CWnd* PASCAL GetOpenClipboardWindow( );
```

Return Value

The handle of the window that currently has the Clipboard open if the function is successful; otherwise **NULL**.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetClipboardOwner
CWnd::GetClipboardViewer
CWnd::OpenClipboard
GetOpenClipboardWindow

Concepts

CWnd Members

CWnd::GetOwner

Retrieves a pointer to the owner of the window.

```
CWnd* GetOwner( ) const;
```

Return Value

A pointer to a **CWnd** object.

Remarks

If the window has no owner, then a pointer to the parent window object is returned by default. Note that the relationship between the owner and the owned differs from the parent-child aspect in several important aspects. For example, a window with a parent is confined to its parent window's client area. Owned windows can be drawn at any location on the desktop.

The ownership concept of this function is different from the ownership concept of GetWindow.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetParent
CWnd::SetOwner

Concepts

CWnd Members

CWnd::GetParent

Call this function to get a pointer to a child window's parent window (if any).

```
CWnd* GetParent( ) const;
```

Return Value

See the Return Values section in GetParent in the *Platform SDK*.

Remarks

The **GetParent** function returns a pointer to the immediate parent (if it exists). In contrast, the GetParentOwner function returns a pointer to the most immediate parent or owner window that is not a child window (does not have the **WS_CHILD** style). If you have a child window within a child window **GetParent** and **GetParentOwner** return different results.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetAncestor
CWnd::GetParentOwner
CWnd::GetOwner
CWnd::SetOwner
CWnd::SetParent
GetParent

Concepts

CWnd Members

CWnd::GetParentFrame

Call this member function to retrieve the parent frame window.

```
CFrameWnd* GetParentFrame( ) const;
```

Return Value

A pointer to a frame window if successful; otherwise **NULL**.

Remarks

The member function searches up the parent chain until a CFrameWnd (or derived class) object is found.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetDescendantWindow
CWnd::GetParent
CFrameWnd::GetActiveView

Concepts

CWnd Members

CWnd::GetParentOwner

Call this member function to get a pointer to a child window's parent window or owner window.

```
CWnd* GetParentOwner( ) const;
```

Return Value

A pointer to a **CWnd** object. If a **CWnd** object is not attached to the handle, a temporary **CWnd** object is created and attached. The pointer may be temporary and should not be stored for later use.

Remarks

GetParentOwner returns a pointer to the most immediate parent or owner window that is not a child window (does not have the **WS_CHILD** style). The current owner window can be set with SetOwner. By default, the parent of a window is its owner.

In contrast, the GetParent function returns a pointer to the immediate parent, whether it is a child window or not. If you have a child window within a child window **GetParent** and **GetParentOwner** return different results.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetParent
CWnd::GetOwner
CWnd::SetOwner
CWnd::SetParent
GetParent

Concepts

CWnd Members

CWnd::GetProperty

Call this member function to get the ActiveX control property specified by *dwDispID*.

```
void GetProperty(
   DISPID dwDispID,
   VARTYPE vtProp,
   void* pvProp
)const;
```

Parameters

dwDispID

Identifies the property to be retrieved.

vtProp

Specifies the type of the property to be retrieved. For possible values, see the Remarks section for COleDispatchDriver::InvokeHelper.

pvProp

Address of the variable that will that will receive the property value. It must match the type specified by *vtProp*.

Remarks

GetProperty returns the value through *pvProp*.

Note
This function should be called only on a **CWnd** object that represents an ActiveX control.

For more information about using this member function with ActiveX Control Containers, see the article ActiveX Control Containers: Programming ActiveX Controls in an ActiveX Control Container.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::InvokeHelper
COleDispatchDriver Class
CWnd::CreateControl

Concepts

CWnd Members

CWnd::GetSafeHwnd

Returns **m_hWnd**, or **NULL** if the **this** pointer is **NULL**.

```
HWND GetSafeHwnd( ) const;
```

Return Value

Returns the window handle for a window. Returns **NULL** if the **CWnd** is not attached to a window or if it is used with a **NULL CWnd** pointer.

Example

See the example for CWnd::SubclassWindow.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::GetSafeOwner

Call this member function to retrieve the owner window that should be used for dialog boxes or other modal windows.

```
static CWnd* GetSafeOwner(
   CWnd* pParent = NULL,
   HWND* pWndTop = NULL
);
```

Parameters

pParent

A pointer to a parent **CWnd** window. May be **NULL**.

pWndTop

A pointer to the window that is currently on top. May be **NULL**.

Return Value

A pointer to the safe owner for the given window.

Remarks

The safe owner is the first non-child parent window of *pParent*. If *pParent* is **NULL**, the thread's main window (retrieved via AfxGetMainWnd) is used to find an owner.

Note
The framework itself uses this function to determine the correct owner window for dialog boxes and property sheets where the owner is not specified.

See Also

Reference

CWnd Class

Hierarchy Chart

AfxGetMainWnd

Concepts

CWnd Members

CWnd::GetScrollBarCtrl

Call this member function to obtain a pointer to the specified sibling scroll bar or splitter window.

```
virtual CScrollBar* GetScrollBarCtrl(
   int nBar
) const;
```

Parameters

nBar

Specifies the type of scroll bar. The parameter can take one of the following values:

- **SB_HORZ** Retrieves the position of the horizontal scroll bar.
- **SB_VERT** Retrieves the position of the vertical scroll bar.

Return Value

A sibling scroll-bar control, or **NULL** if none.

Remarks

This member function does not operate on scroll bars created when the **WS_HSCROLL** or **WS_VSCROLL** bits are set during the creation of a window. The **CWnd** implementation of this function simply returns **NULL**. Derived classes, such as **CView**, implement the described functionality.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::EnableScrollBarCtrl

Concepts

CWnd Members

CWnd::GetScrollBarInfo

Retrieves information about the specified scroll bar.

```
BOOL GetScrollBarInfo(
   LONG idObject,
   PSCROLLBARINFO psbi
) const;
```

Parameters

idObject

Specifies the menu object. For a list of possible values, see GetScrollBarInfo.

psbi

Pointer to a SCROLLBARINFO structure that receives the information.

Return Value

Nonzero if the function succeeds; otherwise 0.

Remarks

This member function emulates the functionality of the function GetScrollBarInfo, as described in the *Platform SDK*.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::GetScrollInfo

Call this member function to retrieve the information that the **SCROLLINFO** structure maintains about a scroll bar.

```
BOOL GetScrollInfo(
   int nBar,
   LPSCROLLINFO lpScrollInfo,
   UINT nMask = SIF_ALL
);
```

Parameters

nBar

Specifies whether the scroll bar is a control or part of a window's nonclient area. If it is part of the nonclient area, *nBar* also indicates whether the scroll bar is positioned horizontally, vertically, or both. It must be one of the following:

- **SB_CTL** Contains the parameters for a scroll bar control. The **m_hWnd** data member must be the handle of the scroll bar control.
- **SB_HORZ** Specifies that the window is a horizontal scroll bar.
- **SB_VERT** Specifies that the window is a vertical scroll bar.

lpScrollInfo

A pointer to a SCROLLINFO structure. See the *Platform SDK* for more information about this structure.

nMask

Specifies the scroll bar parameters to retrieve. The default specifies a combination of **SIF_PAGE**, **SIF_POS**, **SIF_TRACKPOS**, and **SIF_RANGE**. See **SCROLLINFO** for more information on the *nMask* values.

Return Value

If the message retrieved any values, the return is **TRUE**. Otherwise, it is **FALSE**.

Remarks

GetScrollInfo enables applications to use 32-bit scroll positions.

The SCROLLINFO structure contains information about a scroll bar, including the minimum and maximum scrolling positions, the page size, and the position of the scroll box (the thumb). See the **SCROLLINFO** structure topic in the *Platform SDK* for more information about changing the structure defaults.

The MFC Windows message handlers that indicate scroll-bar position, CWnd::OnHScroll and CWnd::OnVScroll, provide only 16 bits of position data. **GetScrollInfo** and **SetScrollInfo** provide 32 bits of scroll-bar position data. Thus, an application can call **GetScrollInfo** while processing either **CWnd::OnHScroll** or **CWnd::OnVScroll** to obtain 32-bit scroll-bar position data.

See Also

Reference

CWnd Class
Hierarchy Chart
CScrollBar::GetScrollInfo
CWnd::SetScrollInfo
CWnd::SetScrollPos
CWnd::OnVScroll
CWnd::OnHScroll
SCROLLINFO

Concepts

CWnd Members

CWnd::GetScrollLimit

Call this member function to retrieve the maximum scrolling position of the scroll bar.

```
int GetScrollLimit(
   int nBar
);
```

Parameters

nBar

Specifies the type of scroll bar. The parameter can take one of the following values:

- **SB_HORZ** Retrieves the scroll limit of the horizontal scroll bar.
- **SB_VERT** Retrieves the scroll limit of the vertical scroll bar.

Return Value

Specifies the maximum position of a scroll bar if successful; otherwise 0.

See Also

Reference

CWnd Class
Hierarchy Chart
CScrollBar::GetScrollLimit

Concepts

CWnd Members

CWnd::GetScrollPos

Retrieves the current position of the scroll box of a scroll bar.

```
int GetScrollPos(
   int nBar
) const;
```

Parameters

nBar

Specifies the scroll bar to examine. The parameter can take one of the following values:

- **SB_HORZ** Retrieves the position of the horizontal scroll bar.
- **SB_VERT** Retrieves the position of the vertical scroll bar.

Return Value

Specifies the current position of the scroll box in the scroll bar if successful; otherwise 0.

Remarks

The current position is a relative value that depends on the current scrolling range. For example, if the scrolling range is 50 to 100 and the scroll box is in the middle of the bar, the current position is 75.

See Also

Reference

CWnd Class
Hierarchy Chart
GetScrollPos
CScrollBar::GetScrollPos

Concepts

CWnd Members

CWnd::GetScrollRange

Copies the current minimum and maximum scroll-bar positions for the given scroll bar to the locations specified by *lpMinPos* and *lpMaxPos*.

```
void GetScrollRange(
   int nBar,
   LPINT lpMinPos,
   LPINT lpMaxPos
) const;
```

Parameters

nBar

Specifies the scroll bar to examine. The parameter can take one of the following values:

- **SB_HORZ** Retrieves the position of the horizontal scroll bar.
- **SB_VERT** Retrieves the position of the vertical scroll bar.

lpMinPos

Points to the integer variable that is to receive the minimum position.

lpMaxPos

Points to the integer variable that is to receive the maximum position.

Remarks

If **CWnd** does not have a scroll bar, then the **GetScrollRange** member function copies 0 to *lpMinPos* and *lpMaxPos*.

The default range for a standard scroll bar is 0 to 100. The default range for a scroll-bar control is empty (both values are 0).

See Also

Reference

CWnd Class
Hierarchy Chart
GetScrollRange

Concepts

CWnd Members

CWnd::GetStyle

Returns the current window style.

```
DWORD GetStyle( ) const;
```

Return Value

The window's style. For more information about the window styles used in MFC, see Window Styles.

See Also

Reference

CWnd Class
Hierarchy Chart
GetWindowLong

Concepts

CWnd Members

CWnd::GetSystemMenu

Allows the application to access the Control menu for copying and modification.

```
CMenu* GetSystemMenu(
   BOOL bRevert
) const;
```

Parameters

bRevert

Specifies the action to be taken. If *bRevert* is **FALSE**, **GetSystemMenu** returns a handle to a copy of the Control menu currently in use. This copy is initially identical to the Control menu but can be modified. If *bRevert* is **TRUE**, **GetSystemMenu** resets the Control menu back to the default state. The previous, possibly modified, Control menu, if any, is destroyed. The return value is undefined in this case.

Return Value

Identifies a copy of the Control menu if *bRevert* is **FALSE**. If *bRevert* is **TRUE**, the return value is undefined.

The returned pointer may be temporary and should not be stored for later use.

Remarks

Any window that does not use **GetSystemMenu** to make its own copy of the Control menu receives the standard Control menu.

The pointer returned by the **GetSystemMenu** member function can be used with the CMenu::AppendMenu, CMenu::InsertMenu, or CMenu::ModifyMenu functions to change the Control menu.

The Control menu initially contains items identified with various ID values such as **SC_CLOSE**, **SC_MOVE**, and **SC_SIZE**. Items on the Control menu generate WM_SYSCOMMAND messages. All predefined Control-menu items have ID numbers greater than 0xF000. If an application adds items to the Control menu, it should use ID numbers less than F000.

Windows may automatically make items unavailable on the standard Control menu. **CWnd** can carry out its own selection or unavailability by responding to the WM_INITMENU messages, which are sent before any menu is displayed.

Example

```
BOOL CMyDlg::OnInitDialog()
{
   CDialog::OnInitDialog();

   // Add "About..." menu item to system menu.

   // IDM_ABOUTBOX must be in the system command range.
   ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
   ASSERT(IDM_ABOUTBOX < 0xF000);

   CMenu* pSysMenu = GetSystemMenu(FALSE);
   if (pSysMenu != NULL)
   {
      CString strAboutMenu;
      strAboutMenu.LoadString(IDS_ABOUTBOX);
      if (!strAboutMenu.IsEmpty())
      {
         pSysMenu->AppendMenu(MF_SEPARATOR);
         pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
      }
   }

   // Set the icon for this dialog. The framework does this automatically
   // when the application's main window is not a dialog
   SetIcon(m_hIcon, TRUE);   // Set big icon
   SetIcon(m_hIcon, FALSE);  // Set small icon
```

```
    return TRUE;  // Return TRUE unless you set the focus to a control
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CMenu::AppendMenu
CMenu::InsertMenu
CMenu::ModifyMenu
GetSystemMenu

Concepts

CWnd Members

CWnd::GetTitleBarInfo

Retrieves information about the specified title bar.

```
BOOL GetTitleBarInfo(
PTITLEBARINFO pti
) const;
```

Parameters

pti

Pointer to a TITLEBARINFO structure that receives the information.

Remarks

This member function emulates the functionality of the function GetTitleBarInfo, as described in the *Platform SDK*.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::GetTopLevelFrame

Call this member function to retrieve the window's top level frame window, if any.

```
CFrameWnd* GetTopLevelFrame( ) const;
```

Return Value

Identifies the top-level frame window of the window.

The returned pointer may be temporary and should not be stored for later use.

Remarks

If **CWnd** has no attached window, or its top-level parent is not a CFrameWnd-derived object, this function returns **NULL**.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetTopLevelOwner
CWnd::GetTopLevelParent

Concepts

CWnd Members

CWnd::GetTopLevelOwner

Call this member function to retrieve the top-level window.

```
CWnd* GetTopLevelOwner( ) const;
```

Return Value

Identifies the top-level window. The returned pointer may be temporary and should not be stored for later use.

Remarks

The top-level window is the window that is a child of the desktop. If **CWnd** has no attached window, this function returns **NULL**.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetTopLevelFrame
CWnd::GetTopLevelParent

Concepts

CWnd Members

CWnd::GetTopLevelParent

Call this member function to retrieve the window's top-level parent.

```
CWnd* GetTopLevelParent( ) const;
```

Return Value

Identifies the top-level parent window of the window.

The returned pointer may be temporary and should not be stored for later use.

Remarks

GetTopLevelParent is similar to GetTopLevelFrame and GetTopLevelOwner; however, it ignores the value set as the current owner window.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetTopLevelOwner
CWnd::GetTopLevelFrame
CWnd::GetOwner
CWnd::SetOwner

Concepts

CWnd Members

CWnd::GetTopWindow

Searches for the top-level child window that belongs to **CWnd**.

```
CWnd* GetTopWindow( ) const;
```

Return Value

Identifies the top-level child window in a **CWnd** linked list of child windows. If no child windows exist, the value is **NULL**.

The returned pointer may be temporary and should not be stored for later use.

Remarks

If **CWnd** has no children, this function returns **NULL**.

See Also

Reference

CWnd Class

Hierarchy Chart

GetTopWindow

Concepts

CWnd Members

CWnd::GetUpdateRect

Retrieves the coordinates of the smallest rectangle that completely encloses the update region.

```
BOOL GetUpdateRect(
   LPRECT lpRect,
   BOOL bErase = FALSE
);
```

Parameters

lpRect

Points to a **CRect** object or RECT structure that is to receive the client coordinates of the update that encloses the update region.

Set this parameter to **NULL** to determine whether an update region exists within the **CWnd**. If *lpRect* is **NULL**, the **GetUpdateRect** member function returns nonzero if an update region exists and 0 if one does not. This provides a way to determine whether a **WM_PAINT** message resulted from an invalid area. Do not set this parameter to **NULL** in Windows version 3.0 and earlier.

bErase

Specifies whether the background in the update region is to be erased.

Return Value

Specifies the status of the update region. The value is nonzero if the update region is not empty; otherwise 0.

If the *lpRect* parameter is set to **NULL**, the return value is nonzero if an update region exists; otherwise 0.

Remarks

If **CWnd** was created with the **CS_OWNDC** style and the mapping mode is not **MM_TEXT**, the **GetUpdateRect** member function gives the rectangle in logical coordinates. Otherwise, **GetUpdateRect** gives the rectangle in client coordinates. If there is no update region, **GetUpdateRect** sets the rectangle to be empty (sets all coordinates to 0).

The *bErase* parameter specifies whether **GetUpdateRect** should erase the background of the update region. If *bErase* is **TRUE** and the update region is not empty, the background is erased. To erase the background, **GetUpdateRect** sends the WM_ERASEBKGND message.

The update rectangle retrieved by the BeginPaint member function is identical to that retrieved by the **GetUpdateRect** member function.

The **BeginPaint** member function automatically validates the update region, so any call to **GetUpdateRect** made immediately after a call to **BeginPaint** retrieves an empty update region.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::BeginPaint
GetUpdateRect
CWnd::OnPaint
CWnd::RedrawWindow

Concepts

CWnd Members

CWnd::GetUpdateRgn

Retrieves the update region into a region identified by *pRgn*.

```
int GetUpdateRgn(
   CRgn* pRgn,
   BOOL bErase = FALSE
);
```

Parameters

pRgn

Identifies the update region.

bErase

Specifies whether the background will be erased and nonclient areas of child windows will be drawn. If the value is **FALSE**, no drawing is done.

Return Value

Specifies a short-integer flag that indicates the type of resulting region. The value can take any one of the following:

- **SIMPLEREGION** The region has no overlapping borders.
- **COMPLEXREGION** The region has overlapping borders.
- **NULLREGION** The region is empty.
- **ERROR** No region was created.

Remarks

The coordinates of this region are relative to the upper-left corner (client coordinates).

The BeginPaint member function automatically validates the update region, so any call to **GetUpdateRgn** made immediately after a call to **BeginPaint** retrieves an empty update region.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::BeginPaint
GetUpdateRgn

Concepts

CWnd Members

CWnd::GetWindow

Returns a pointer to the window requested, or **NULL** if none.

```
CWnd* GetWindow(
   UINT nCmd
) const;
```

Parameters

nCmd

Specifies the relationship between **CWnd** and the returned window. It can take one of the following values:

- **GW_CHILD** Identifies the **CWnd** first child window.
- **GW_HWNDFIRST** If **CWnd** is a child window, returns the first sibling window. Otherwise, it returns the first top-level window in the list.
- **GW_HWNDLAST** If **CWnd** is a child window, returns the last sibling window. Otherwise, it returns the last top-level window in the list.
- **GW_HWNDNEXT** Returns the next window on the window manager's list.
- **GW_HWNDPREV** Returns the previous window on the window manager's list.
- **GW_OWNER** Identifies the **CWnd** owner.

Return Value

The returned pointer may be temporary and should not be stored for later use.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetParent
CWnd::GetNextWindow
GetWindow

Concepts

CWnd Members

CWnd::GetWindowContextHelpId

Call this member function to retrieve the help context identifier, if any, associated with the window.

```
DWORD GetWindowContextHelpId( ) const;
```

Return Value

The help context identifier. Returns 0 if the window has none.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::GetWindowedChildCount

Call this member function to retrieve the number of associated child windows.

```
long GetWindowedChildCount( );
```

Return Value

The number of child windows associated with the **CWnd** object.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::GetWindowlessChildCount

Concepts

CWnd Members

CWnd::GetWindowDC

Retrieves the display context for the entire window, including caption bar, menus, and scroll bars.

```
CDC* GetWindowDC( );
```

Return Value

Identifies the display context for the given window if the function is successful; otherwise **NULL**.

The returned pointer may be temporary and should not be stored for later use. ReleaseDC should be called once for each successful call to **GetWindowDC**.

Remarks

A window display context permits painting anywhere in **CWnd**, since the origin of the context is the upper-left corner of **CWnd** instead of the client area.

Default attributes are assigned to the display context each time it retrieves the context. Previous attributes are lost.

GetWindowDC is intended to be used for special painting effects within the **CWnd** nonclient area. Painting in nonclient areas of any window is not recommended.

The GetSystemMetrics Windows function can be used to retrieve the dimensions of various parts of the nonclient area, such as the caption bar, menu, and scroll bars.

After painting is complete, the ReleaseDC member function must be called to release the display context. Failure to release the display context will seriously affect painting requested by applications due to limitations on the number of device contexts that can be open at the same time.

See Also

Reference

CWnd Class
Hierarchy Chart
GetSystemMetrics
CWnd::ReleaseDC
GetWindowDC
CWnd::GetDC
CWindowDC Class

Concepts

CWnd Members

CWnd::GetWindowInfo

Retrieves information about the window.

```
BOOL GetWindowInfo(
   PWINDOWINFO pwi
) const;
```

Parameters

pwi

A pointer to a WINDOWINFO structure.

Remarks

This member function emulates the functionality of the function GetWindowInfo, as described in the *Platform SDK*.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::GetWindowlessChildCount

Retrieves the number of associated windowless child windows.

```
long GetWindowlessChildCount( );
```

Return Value

The number of windowless child windows associated with the **CWnd** object.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::GetWindowedChildCount

Concepts

CWnd Members

CWnd::GetWindowPlacement

Retrieves the show state and the normal (restored), minimized, and maximized positions of a window.

```
BOOL GetWindowPlacement(
   WINDOWPLACEMENT* lpwndpl
) const;
```

Parameters

lpwndpl

Points to the **WINDOWPLACEMENT** structure that receives the show state and position information.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

The **flags** member of the WINDOWPLACEMENT structure retrieved by this function is always 0. If **CWnd** is maximized, the **showCmd** member of **WINDOWPLACEMENT** is **SW_SHOWMAXIMIZED**. If the window is minimized, it is **SW_SHOWMINIMIZED.** It is **SW_SHOWNORMAL** otherwise.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::SetWindowPlacement
GetWindowPlacement

Concepts

CWnd Members

CWnd::GetWindowRect

Copies the dimensions of the bounding rectangle of the **CWnd** object to the structure pointed to by *lpRect*.

```
void GetWindowRect(
   LPRECT lpRect
) const;
```

Parameters

lpRect

Points to a **CRect** object or a RECT structure that will receive the screen coordinates of the upper-left and lower-right corners.

Remarks

The dimensions are given in screen coordinates relative to the upper-left corner of the display screen. The dimensions of the caption, border, and scroll bars, if present, are included.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetClientRect
CWnd::MoveWindow
CWnd::SetWindowPos
GetWindowRect

Concepts

CWnd Members

CWnd::GetWindowRgn

Call this member function to get the window region of a window.

```
int GetWindowRgn(
   HRGN hRgn
)const;
```

Parameters

hRgn

A handle to a window region.

Return Value

The return value specifies the type of the region that the function obtains. It can be one of the following values:

- **NULLREGION** The region is empty.
- **SIMPLEREGION** The region is a single rectangle.
- **COMPLEXREGION** The region is more than one rectangle.
- **ERROR** An error occurred; the region is unaffected.

Remarks

The window region determines the area within the window where the operating system permits drawing. The operating system does not display any portion of a window that lies outside of the window region.

The coordinates of a window's window region are relative to the upper-left corner of the window, not the client area of the window.

To set the window region of a window, call CWnd::SetWindowRgn.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::SetWindowRgn

Concepts

CWnd Members

CWnd::GetWindowText

Copies the **CWnd** caption title (if it has one) into the buffer pointed to by *lpszStringBuf* or into the destination string *rString*.

```
int GetWindowText(
   LPTSTR lpszStringBuf,
   int nMaxCount
) const;
void GetWindowText(
   CString& rString
) const;
```

Parameters

lpszStringBuf

Points to the buffer that is to receive the copied string of the window's title.

nMaxCount

Specifies the maximum number of characters to be copied to the buffer, including the terminating null character. If the string is longer than the number of characters specified in *nMaxCount*, it is truncated.

rString

A CString object that is to receive the copied string of the window's title.

Return Value

Specifies the length, in characters, of the copied string, not including the terminating null character. It is 0 if **CWnd** has no caption or if the caption is empty.

Remarks

If the **CWnd** object is a control, the **GetWindowText** member function copies the text within the control instead of copying the caption.

This member function causes the WM_GETTEXT message to be sent to the **CWnd** object.

Example

See the example for CWnd::SetWindowText.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::SetWindowText
WM_GETTEXT
CWnd::GetWindowTextLength

Concepts

CWnd Members

CWnd::GetWindowTextLength

Returns the length of the **CWnd** object caption title.

```
int GetWindowTextLength( ) const;
```

Return Value

Specifies the text length in characters, not including any null-termination character. The value is 0 if no such text exists.

Remarks

If **CWnd** is a control, the **GetWindowTextLength** member function returns the length of the text within the control instead of the caption.

This member function causes the WM_GETTEXTLENGTH message to be sent to the **CWnd** object.

Example

See the example for CWnd::SetWindowText.

See Also

Reference

CWnd Class
Hierarchy Chart
GetWindowTextLength
WM_GETTEXTLENGTH
CWnd::GetWindowText

Concepts

CWnd Members

CWnd::HideCaret

Hides the caret by removing it from the display screen.

```
void HideCaret( );
```

Remarks

Although the caret is no longer visible, it can be displayed again by using the ShowCaret member function. Hiding the caret does not destroy its current shape.

Hiding is cumulative. If **HideCaret** has been called five times in a row, the **ShowCaret** member function must be called five times before the caret will be shown.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::ShowCaret
HideCaret

Concepts

CWnd Members

CWnd::HiliteMenuItem

Highlights or removes the highlight from a top-level (menu-bar) menu item.

```
BOOL HiliteMenuItem(
   CMenu* pMenu,
   UINT nIDHiliteItem,
   UINT nHilite
);
```

Parameters

pMenu

Identifies the top-level menu that contains the item to be highlighted.

nIDHiliteItem

Specifies the menu item to be highlighted, depending on the value of the *nHilite* parameter.

nHilite

Specifies whether the menu item is highlighted or the highlight is removed. It can be a combination of **MF_HILITE** or **MF_UNHILITE** with **MF_BYCOMMAND** or **MF_BYPOSITION**. The values can be combined using the bitwise OR operator. These values have the following meanings:

- **MF_BYCOMMAND** Interprets *nIDHiliteItem* as the menu-item ID (the default interpretation).
- **MF_BYPOSITION** Interprets *nIDHiliteItem* as the zero-based offset of the menu item.
- **MF_HILITE** Highlights the item. If this value is not given, the highlight is removed from the item.
- **MF_UNHILITE** Removes the highlight from the item.

Return Value

Specifies whether the menu item was highlighted. Nonzero if the item was highlighted; otherwise 0.

Remarks

The **MF_HILITE** and **MF_UNHILITE** flags can be used only with this member function; they cannot be used with the CMenu::ModifyMenu member function.

See Also

Reference

CWnd Class
Hierarchy Chart
CMenu::ModifyMenu
HiliteMenuItem

Concepts

CWnd Members

CWnd::HtmlHelp

Call this member function to invoke the HTMLHelp application.

```
virtual void HtmlHelp(
   DWORD_PTR dwData,
   UINT nCmd = 0x000F
);
```

Parameters

dwData

Specifies additional data. The value used depends on the value of the *nCmd* parameter.

nCmd

Specifies the type of help requested. For a list of possible values and how they affect the *dwData* parameter, see the *uCommand* parameter described in the HTMLHelp API Reference in the *Platform SDK*.

Remarks

See CWinApp::HtmlHelp for more information.

See Also

Tasks

Displaying F1 Help for a Dialog Box or Menu Option

Reference

CWnd Class

Hierarchy Chart

CWnd::WinHelp

Concepts

CWnd Members

CWnd::Invalidate

Invalidates the entire client area of **CWnd**.

```
void Invalidate(
   BOOL bErase = TRUE
);
```

Parameters

bErase

Specifies whether the background within the update region is to be erased.

Remarks

The client area is marked for painting when the next WM_PAINT message occurs. The region can also be validated before a **WM_PAINT** message occurs by the ValidateRect or ValidateRgn member function.

The *bErase* parameter specifies whether the background within the update area is to be erased when the update region is processed. If *bErase* is **TRUE**, the background is erased when the BeginPaint member function is called; if *bErase* is **FALSE**, the background remains unchanged. If *bErase* is **TRUE** for any part of the update region, the background in the entire region, not just in the given part, is erased.

Windows sends a WM_PAINT message whenever the **CWnd** update region is not empty and there are no other messages in the application queue for that window.

Example

See the example for CWnd::UpdateWindow.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::BeginPaint
CWnd::ValidateRect
CWnd::ValidateRgn
InvalidateRect

Concepts

CWnd Members

CWnd::InvalidateRect

Invalidates the client area within the given rectangle by adding that rectangle to the **CWnd** update region.

```
void InvalidateRect(
   LPCRECT lpRect,
   BOOL bErase = TRUE
);
```

Parameters

lpRect

Points to a **CRect** object or a RECT structure that contains the rectangle (in client coordinates) to be added to the update region. If *lpRect* is **NULL**, the entire client area is added to the region.

bErase

Specifies whether the background within the update region is to be erased.

Remarks

The invalidated rectangle, along with all other areas in the update region, is marked for painting when the next WM_PAINT message is sent. The invalidated areas accumulate in the update region until the region is processed when the next **WM_PAINT** call occurs, or until the region is validated by the ValidateRect or ValidateRgn member function.

The *bErase* parameter specifies whether the background within the update area is to be erased when the update region is processed. If *bErase* is **TRUE**, the background is erased when the BeginPaint member function is called; if *bErase* is **FALSE**, the background remains unchanged. If *bErase* is **TRUE** for any part of the update region, the background in the entire region is erased, not just in the given part.

Windows sends a WM_PAINT message whenever the **CWnd** update region is not empty and there are no other messages in the application queue for that window.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::BeginPaint
CWnd::ValidateRect
CWnd::ValidateRgn
InvalidateRect

Concepts

CWnd Members

CWnd::InvalidateRgn

Invalidates the client area within the given region by adding it to the current update region of **CWnd**.

```
void InvalidateRgn(
   CRgn* pRgn,
   BOOL bErase = TRUE
);
```

Parameters

pRgn

A pointer to a CRgn object that identifies the region to be added to the update region. The region is assumed to have client coordinates. If this parameter is **NULL**, the entire client area is added to the update region.

bErase

Specifies whether the background within the update region is to be erased.

Remarks

The invalidated region, along with all other areas in the update region, is marked for painting when the WM_PAINT message is next sent. The invalidated areas accumulate in the update region until the region is processed when a **WM_PAINT** message is next sent, or until the region is validated by the ValidateRect or ValidateRgn member function.

The *bErase* parameter specifies whether the background within the update area is to be erased when the update region is processed. If *bErase* is **TRUE**, the background is erased when the BeginPaint member function is called; if *bErase* is **FALSE**, the background remains unchanged. If *bErase* is **TRUE** for any part of the update region, the background in the entire region, not just in the given part, is erased.

Windows sends a WM_PAINT message whenever the **CWnd** update region is not empty and there are no other messages in the application queue for that window.

The given region must have been previously created by one of the region functions.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::BeginPaint
CWnd::ValidateRect
CWnd::ValidateRgn
InvalidateRgn

Concepts

CWnd Members

CWnd::InvokeHelper

Call this member function to invoke the ActiveX Control method or property specified by *dwDispID*, in the context specified by *wFlags*.

```
void AFX_CDECL InvokeHelper(
   DISPID dwDispID,
   WORD wFlags,
   VARTYPE vtRet,
   void* pvRet,
   const BYTE* pbParamInfo,
   ...
);
```

Parameters

dwDispID

Identifies the method or property to be invoked.

wFlags

Flags describing the context of the call to **IDispatch::Invoke**. For possible *wFlags* values, see IDispatch::Invoke in the *Platform SDK*.

vtRet

Specifies the type of the return value. For possible values, see the Remarks section for COleDispatchDriver::InvokeHelper.

pvRet

Address of the variable that will that will receive the property value or return value. It must match the type specified by *vtRet*.

pbParamInfo

Pointer to a null-terminated string of bytes specifying the types of the parameters following *pbParamInfo*. For possible values, see the Remarks section for **COleDispatchDriver::InvokeHelper**.

...

Variable List of parameters, of types specified in *pbParamInfo*.

Remarks

The *pbParamInfo* parameter specifies the types of the parameters passed to the method or property. The variable list of arguments is represented by ... in the syntax declaration.

This function converts the parameters to **VARIANTARG** values, then invokes the **IDispatch::Invoke** method on the ActiveX control. If the call to **IDispatch::Invoke** fails, this function will throw an exception. If the **SCODE** (status code) returned by **IDispatch::Invoke** is **DISP_E_EXCEPTION**, this function throws a COleException object, otherwise it throws a COleDispatchException.

Note
This function should be called only on a **CWnd** object that represents an ActiveX control.

For more information about using this member function with ActiveX Control Containers, see the article ActiveX Control Containers: Programming ActiveX Controls in an ActiveX Control Container.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetProperty
CWnd::SetProperty
COleDispatchDriver Class

CWnd::CreateControl

Concepts

CWnd Members

CWnd::IsChild

Indicates whether the window specified by *pWnd* is a child window or other direct descendant of **CWnd**.

```
BOOL IsChild(
   const CWnd* pWnd
) const;
```

Parameters

pWnd

Identifies the window to be tested.

Return Value

Specifies the outcome of the function. The value is nonzero if the window identified by *pWnd* is a child window of **CWnd**; otherwise 0.

Remarks

A child window is the direct descendant of **CWnd** if the **CWnd** object is in the chain of parent windows that leads from the original pop-up window to the child window.

See Also

Reference

CWnd Class
Hierarchy Chart
IsChild

Concepts

CWnd Members

CWnd::IsDialogMessage

Call this member function to determine whether the given message is intended for a modeless dialog box; if it is, this function processes the message.

```
BOOL IsDialogMessage(
   LPMSG lpMsg
);
```

Parameters

lpMsg

Points to an MSG structure that contains the message to be checked.

Return Value

Specifies whether the member function has processed the given message. It is nonzero if the message has been processed; otherwise 0. If the return is 0, call the CWnd::PreTranslateMessage member function of the base class to process the message. In an override of the **CWnd::PreTranslateMessage** member function the code looks like this :

```
BOOL CMyDlg::PreTranslateMessage( msg )
{
    if( IsDialogMessage( msg ) )
        return TRUE;
    else
        return CWnd::PreTranslateMessage( msg );
}
```

Remarks

When the **IsDialogMessage** function processes a message, it checks for keyboard messages and converts them to selection commands for the corresponding dialog box. For example, the TAB key selects the next control or group of controls, and the DOWN ARROW key selects the next control in a group.

You must not pass a message processed by **IsDialogMessage** to the TranslateMessage or DispatchMessage Windows functions, because it has already been processed.

See Also

Reference

CWnd Class
Hierarchy Chart
DispatchMessage
TranslateMessage
GetMessage
CWnd::PreTranslateMessage
IsDialogMessage

Concepts

CWnd Members

CWnd::IsDlgButtonChecked

Determines whether a button control has a check mark next to it.

```
UINT IsDlgButtonChecked(
   int nIDButton
) const;
```

Parameters

nIDButton

Specifies the integer identifier of the button control.

Return Value

Nonzero if the given control is checked, and 0 if it is not checked. Only radio buttons and check boxes can be checked. For three-state buttons, the return value can be 2 if the button is indeterminate. This member function returns 0 for a pushbutton.

Remarks

If the button is a three-state control, the member function determines whether it is dimmed, checked, or neither.

See Also

Reference

CWnd Class
Hierarchy Chart
IsDlgButtonChecked
CButton::GetCheck

Concepts

CWnd Members

CWnd::IsIconic

Specifies whether **CWnd** is minimized (iconic).

```
BOOL IsIconic( ) const;
```

Return Value

Nonzero if **CWnd** is minimized; otherwise 0.

Example

```
// This code, normally emitted by the Application Wizard for a dialog-
// based project for the dialog's WM_PAINT handler, runs only if the
// window is iconic. The window erase the icon's area, then
// paints the icon referenced by m_hIcon.

if (IsIconic())
{
   CPaintDC dc(this); // device context for painting

   IconEraseBkgnd(dc);

   // Center icon in client rectangle
   int cxIcon = GetSystemMetrics(SM_CXICON);
   int cyIcon = GetSystemMetrics(SM_CYICON);
   CRect rect;
   GetClientRect(&rect);
   int x = (rect.Width() - cxIcon + 1) / 2;
   int y = (rect.Height() - cyIcon + 1) / 2;

   // Draw the icon
   dc.DrawIcon(x, y, m_hIcon);
}
```

See Also

Reference

CWnd Class

Hierarchy Chart

IsIconic

Concepts

CWnd Members

CWnd::IsWindowEnabled

Specifies whether **CWnd** is enabled for mouse and keyboard input.

```
BOOL IsWindowEnabled( ) const;
```

Return Value

Nonzero if **CWnd** is enabled; otherwise 0.

Example

```
//change the background color of an edit control on the dialog
HBRUSH CMyDlg::OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor)
{
   HBRUSH hbr = CDialog::OnCtlColor(pDC, pWnd, nCtlColor);

   if ( pWnd->GetDlgCtrlID() == IDC_EDIT1 )
      if ( pWnd->IsWindowEnabled())
      {
         // Red brush for the background...
         pDC->SetBkColor(RGB(255,0,0));
         // m_pRedBrush is the CBrush object initialized with a red brush
         // using CreateSolidBrush
         return(HBRUSH)m_pRedBrush->GetSafeHandle();
      }
      else
      {
         // Blue brush for the background...
         pDC->SetBkColor(RGB(0, 0, 255));
         // m_pBlueBrush is the CBrush object initialized with a blue
         // brush using CreateSolidBrush
         return (HBRUSH)m_pBlueBrush->GetSafeHandle();
      }
   return hbr;
}
```

See Also

Reference

CWnd Class

Hierarchy Chart

IsWindowEnabled

Concepts

CWnd Members

CWnd::IsWindowVisible

Determines the visibility state of the given window.

```
BOOL IsWindowVisible( ) const;
```

Return Value

Nonzero if **CWnd** is visible (has the WS_VISIBLE style bit set, and parent window is visible). Because the return value reflects the state of the **WS_VISIBLE** style bit, the return value may be nonzero even though **CWnd** is totally obscured by other windows.

Remarks

A window possesses a visibility state indicated by the **WS_VISIBLE** style bit. When this style bit is set with a call to the ShowWindow member function, the window is displayed and subsequent drawing to the window is displayed as long as the window has the style bit set.

Any drawing to a window that has the **WS_VISIBLE** style will not be displayed if the window is covered by other windows or is clipped by its parent window.

Example

```
// This example uses the CWnd::IsWindowVisible() function to
// determine if a dialog box is visible. If it is not, it calls
// CWnd::ShowWindow with the SW_SHOWNORMAL command.

void CSomeClass::DisplayDlgWindow()
{
   if(!m_myDlg.IsWindowVisible())
   {
      m_myDlg.ShowWindow(SW_SHOWNORMAL);
   }
}

// This example uses the CWnd::IsWindowVisible() function to
// determine if a dialog box is visible. If it is, it calls
// CWnd::ShowWindow with the SW_HIDE command.

void CSomeClass::HideDlgWindow()
{
   if(m_myDlg.IsWindowVisible())
   {
      m_myDlg.ShowWindow(SW_HIDE);
   }
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::ShowWindow
IsWindowVisible

Concepts

CWnd Members

CWnd::IsZoomed

Determines whether **CWnd** has been maximized.

```
BOOL IsZoomed( ) const;
```

Return Value

Nonzero if **CWnd** is maximized; otherwise 0.

See Also

Reference

CWnd Class

Hierarchy Chart

IsZoomed

Concepts

CWnd Members

CWnd::KillTimer

Kills the timer event identified by *nIDEvent* from the earlier call to **SetTimer**.

```
BOOL KillTimer(
   UINT_PTR nIDEvent
);
```

Parameters

nIDEvent

The value of the timer event passed to SetTimer.

Return Value

Specifies the outcome of the function. The value is nonzero if the event was killed. It is 0 if the **KillTimer** member function could not find the specified timer event.

Remarks

Pending WM_TIMER messages associated with the timer are not removed from the message queue.

Example

See the example for CWnd::SetTimer.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::SetTimer
KillTimer

Concepts

CWnd Members

CWnd::LockWindowUpdate

Disables drawing in the given window.

```
BOOL LockWindowUpdate( );
```

Return Value

Nonzero if the function is successful. It is 0 if a failure occurs or if the **LockWindowUpdate** function has been used to lock another window.

Remarks

A locked window cannot be moved. Only one window can be locked at a time. To unlock a window locked with **LockWindowUpdate**, call UnlockWindowUpdate.

If an application with a locked window (or any locked child windows) calls the GetDC, GetDCEx, or BeginPaint Windows function, the called function returns a device context whose visible region is empty. This will occur until the application unlocks the window by calling the **UnlockWindowUpdate** member function.

While window updates are locked, the system keeps track of the bounding rectangle of any drawing operations to device contexts associated with a locked window. When drawing is reenabled, this bounding rectangle is invalidated in the locked window and its child windows to force an eventual WM_PAINT message to update the screen. If no drawing has occurred while the window updates were locked, no area is invalidated.

The **LockWindowUpdate** member function does not make the given window invisible and does not clear the WS_VISIBLE style bit.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetDCEx
LockWindowUpdate

Concepts

CWnd Members

CWnd::MapWindowPoints

Converts (maps) a set of points from the coordinate space of the **CWnd** to the coordinate space of another window.

```
void MapWindowPoints(
   CWnd* pwndTo,
   LPRECT lpRect
) const;
void MapWindowPoints(
   CWnd* pwndTo,
   LPPOINT lpPoint,
   UINT nCount
) const;
```

Parameters

pwndTo

Identifies the window to which points are converted. If this parameter is **NULL**, the points are converted to screen coordinates.

lpRect

Specifies the rectangle whose points are to be converted. The first version of this function is available only for Windows 3.1 and later.

lpPoint

A pointer to an array of POINT structures that contain the set of points to be converted.

nCount

Specifies the number of **POINT** structures in the array pointed to by *lpPoint*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::ClientToScreen
CWnd::ScreenToClient
MapWindowPoints

Concepts

CWnd Members

CWnd::MessageBox

Creates and displays a window that contains an application-supplied message and caption, plus a combination of the predefined icons and pushbuttons described in the Message-Box Styles list.

```
int MessageBox(
   LPCTSTR lpszText,
   LPCTSTR lpszCaption = NULL,
   UINT nType = MB_OK
);
```

Parameters

lpszText

Points to a **CString** object or null-terminated string containing the message to be displayed.

lpszCaption

Points to a **CString** object or null-terminated string to be used for the message-box caption. If *lpszCaption* is **NULL**, the default caption "Error" is used.

nType

Specifies the contents and behavior of the message box.

Return Value

This method utilizes the MessageBox function as defined in the *Platform SDK*. This method returns the result of calling this function.

Remarks

Use the global function AfxMessageBox instead of this member function to implement a message box in your application.

The following shows the various system icons that can be used in a message box:

	MB_ICONHAND, **MB_ICONSTOP**, and **MB_ICONERROR**
	MB_ICONQUESTION
	MB_ICONEXCLAMATION and **MB_ICONWARNING**
	MB_ICONASTERISK and **MB_ICONINFORMATION**

Example

```
void CMyWnd::OnDisplayErrorMessage()
{
   // This displays a message box with the title "Error"
   // and the message "Help, Something went wrong."
   // The error icon is displayed in the message box, along with
   // an OK button.
   MessageBox("Help, Something went wrong.", "Error",
      MB_ICONERROR | MB_OK);
}
```

See Also

Reference

CWnd Class

Hierarchy Chart

MessageBox

AfxMessageBox

Concepts

CWnd Members

CWnd::ModifyStyle

Call this member function to modify a window's style.

```
BOOL ModifyStyle(
   DWORD dwRemove,
   DWORD dwAdd,
   UINT nFlags = 0
);
```

Parameters

dwRemove

Specifies window styles to be removed during style modification.

dwAdd

Specifies window styles to be added during style modification.

nFlags

Flags to be passed to SetWindowPos, or zero if **SetWindowPos** should not be called. The default is zero. See the Remarks section for a list of preset flags.

Return Value

Nonzero if style was successfully modified; otherwise, 0.

Remarks

Styles to be added or removed can be combined by using the bitwise OR (|) operator. See the topics Window Styles and CreateWindow in the *Platform SDK* for information about the available window styles.

If *nFlags* is nonzero, **ModifyStyle** calls the Windows API function SetWindowPos and redraws the window by combining *nFlags* with the following four preset flags:

- **SWP_NOSIZE** Retains the current size.
- **SWP_NOMOVE** Retains the current position.
- **SWP_NOZORDER** Retains the current Z order.
- **SWP_NOACTIVATE** Does not activate the window.

To modify a window's extended styles, see ModifyStyleEx.

Note

For some styles in certain controls (the **ES_READONLY** style in the edit control, for example), **ModifyStyle** may not properly change the style because the control may need to perform special internal processing. In these cases, a corresponding message to change the style will be available (**EM_SETREADONLY** in the example mentioned).

Example

```
// This example adds the WS_CLIPCHILDREN style to the window.
// No Styles are removed from the window.

void CMyView::OnInitialUpdate()
{
   CView::OnInitialUpdate();
   ModifyStyle(0, WS_CLIPCHILDREN);
}
```

See Also

Reference

CWnd Class

Hierarchy Chart
CWnd::SetWindowPos
CWnd::ModifyStyleEx
Window Styles
SetWindowPos

Concepts

CWnd Members

CWnd::ModifyStyleEx

Call this member function to modify a window's extended style.

```
BOOL ModifyStyleEx(
   DWORD dwRemove,
   DWORD dwAdd,
   UINT nFlags = 0
);
```

Parameters

dwRemove

Specifies extended styles to be removed during style modification.

dwAdd

Specifies extended styles to be added during style modification.

nFlags

Flags to be passed to SetWindowPos, or zero if **SetWindowPos** should not be called. The default is zero. See the Remarks section for a list of preset flags.

Return Value

Nonzero if style was successfully modified; otherwise, 0.

Remarks

Styles to be added or removed can be combined by using the bitwise OR (|) operator. See the topics Extended Window Styles in this book and CreateWindowEx in the *Platform SDK* for information about the available extended styles

If *nFlags* is nonzero, **ModifyStyleEx** calls the Windows API function SetWindowPos and redraws the window by combining *nFlags* with the following four preset flags:

- **SWP_NOSIZE** Retains the current size.
- **SWP_NOMOVE** Retains the current position.
- **SWP_NOZORDER** Retains the current Z order.
- **SWP_NOACTIVATE** Does not activate the window.

To modify windows using regular window styles, see ModifyStyle.

Example

```
// This example would make the dialog box transparent by
// changing the dialog window's extended styles.

int CMyDialog::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CDialog::OnCreate(lpCreateStruct) == -1)
      return -1;

   ModifyStyleEx(0, WS_EX_TRANSPARENT);

   return 0;
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::ModifyStyle

CreateWindowEx

Concepts

CWnd Members

CWnd::MoveWindow

Changes the position and dimensions.

```
void MoveWindow(
   int x,
   int y,
   int nWidth,
   int nHeight,
   BOOL bRepaint = TRUE
);
void MoveWindow(
   LPCRECT lpRect,
      BOOL bRepaint = TRUE
);
```

Parameters

x

Specifies the new position of the left side of the **CWnd**.

y

Specifies the new position of the top of the **CWnd**.

nWidth

Specifies the new width of the **CWnd**.

nHeight

Specifies the new height of the **CWnd**.

bRepaint

Specifies whether **CWnd** is to be repainted. If **TRUE**, **CWnd** receives a WM_PAINT message in its OnPaint message handler as usual. If this parameter is **FALSE**, no repainting of any kind occurs. This applies to the client area, to the nonclient area (including the title and scroll bars), and to any part of the parent window uncovered as a result of **CWnd**'s move. When this parameter is **FALSE**, the application must explicitly invalidate or redraw any parts of **CWnd** and parent window that must be redrawn.

lpRect

The CRect object or RECT structure that specifies the new size and position.

Remarks

For a top-level **CWnd** object, the *x* and *y* parameters are relative to the upper-left corner of the screen. For a child **CWnd** object, they are relative to the upper-left corner of the parent window's client area.

The **MoveWindow** function sends the WM_GETMINMAXINFO message. Handling this message gives **CWnd** the opportunity to modify the default values for the largest and smallest possible windows. If the parameters to the **MoveWindow** member function exceed these values, the values can be replaced by the minimum or maximum values in the **WM_GETMINMAXINFO** handler.

Example

See the example for CWnd::ClientToScreen.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::SetWindowPos
CWnd::OnGetMinMaxInfo
MoveWindow

Concepts

CWnd Members

CWnd::NotifyWinEvent

Signals the system that a predefined event occurred. If any client applications have registered a hook function for the event, the system calls the client's hook function.

```
void NotifyWinEvent(
   DWORD event,
   LONG idObjectType,
   LONG idObject
);
```

Parameters

event

Specifies the event that occurred. This value must be one of the event constants.

idObjectType

Identifies the kind of object that generated the event. This value is one of the predefined object identifiers or a custom object ID value.

idObject

Identifies whether the event was generated by an object or a child element of the object. If this value is **CHILDID_SELF**, the event was generated by the object itself. If not, this value is the child ID of the element that generated the event.

Remarks

This member function emulates the functionality of the function NotifyWinEvent, as described in the *Platform SDK*.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::OnActivate

The framework calls this member function when a **CWnd** object is being activated or deactivated.

```
afx_msg void OnActivate(
   UINT nState,
   CWnd* pWndOther,
   BOOL bMinimized
);
```

Parameters

nState

Specifies whether the **CWnd** is being activated or deactivated. It can be one of the following values:

- **WA_INACTIVE** The window is being deactivated.
- **WA_ACTIVE** The window is being activated through some method other than a mouse click (for example, by use of the keyboard interface to select the window).
- **WA_CLICKACTIVE** The window is being activated by a mouse click.

pWndOther

Pointer to the **CWnd** being activated or deactivated. The pointer can be **NULL**, and it may be temporary.

bMinimized

Specifies the minimized state of the **CWnd** being activated or deactivated. A value of **TRUE** indicates the window is minimized.

If **TRUE**, the **CWnd** is being activated; otherwise deactivated.

Remarks

If the **CWnd** object is activated with a mouse click, it will also receive an OnMouseActivate member function call.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnMouseActivate
CWnd::OnNcActivate
WM_ACTIVATE

Concepts

CWnd Members

CWnd::OnActivateApp

The framework calls this member function to all top-level windows of the task being activated and for all top-level windows of the task being deactivated.

```
afx_msg void OnActivateApp(
   BOOL bActive,
   DWORD dwThreadID
);
```

Parameters

bActive

Specifies whether the **CWnd** is being activated or deactivated. **TRUE** means the **CWnd** is being activated. **FALSE** means the **CWnd** is being deactivated.

dwThreadID

Specifies the value of the thread ID. If *bActive* is **TRUE**, *dwThreadID* identifies the thread that owns the **CWnd** being deactivated. If *bActive* is **FALSE**, *dwThreadID* identifies the thread that owns the **CWnd** being activated.

Remarks

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class

Hierarchy Chart

WM_ACTIVATEAPP

Concepts

CWnd Members

CWnd::OnAmbientProperty

The framework calls this member function to obtain ambient property values from a window that contains OLE controls.

```
virtual BOOL OnAmbientProperty(
   COleControlSite* pSite,
   DISPID dispid,
   VARIANT* pvar
);
```

Parameters

pSite

Pointer to the site of the control that requested the ambient property.

dispid

The dispatch ID of the requested ambient property.

pvar

Pointer to a caller-allocated VARIANT structure, through which the ambient property's value will be returned.

Return Value

TRUE if the ambient property is supported; **FALSE** if not.

Remarks

Override this function to alter the default ambient property values returned by an OLE control container to its controls. Any ambient property requests not handled by an overriding function should be forwarded to the base class implementation.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::OnAskCbFormatName

The framework calls this member function when the Clipboard contains a data handle for the **CF_OWNERDISPLAY** format (that is, when the Clipboard owner will display the Clipboard contents).

```
afx_msg void OnAskCbFormatName(
   UINT nMaxCount,
   LPTSTR lpszString
);
```

Parameters

nMaxCount

Specifies the maximum number of bytes to copy.

lpszString

Points to the buffer where the copy of the format name is to be stored.

Remarks

The Clipboard owner should provide a name for its format.

Override this member function and copy the name of the **CF_OWNERDISPLAY** format into the specified buffer, not exceeding the maximum number of bytes specified.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class

Hierarchy Chart

WM_ASKCBFORMATNAME

Concepts

CWnd Members

CWnd::OnCancelMode

The framework calls this member function to inform **CWnd** to cancel any internal mode.

```
afx_msg void OnCancelMode( );
```

Remarks

If the **CWnd** object has the focus, its **OnCancelMode** member function is called when a dialog box or message box is displayed. This gives the **CWnd** the opportunity to cancel modes such as mouse capture.

The default implementation responds by calling the ReleaseCapture Windows function. Override this member function in your derived class to handle other modes.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::Default
ReleaseCapture
WM_CANCELMODE

Concepts

CWnd Members

CWnd::OnCaptureChanged

The framework calls this member function to notify the window that is losing the mouse capture.

```
afx_msg void OnCaptureChanged(
   CWnd* pWnd
);
```

Parameters

pWnd

A pointer to the window to gain mouse capture

Remarks

A window receives this message even if it calls ReleaseCapture itself. An application should not attempt to set the mouse capture in response to this message. When it receives this message, a window should redraw itself, if necessary, to reflect the new mouse-capture state.

See the *Platform SDK* for information on the **ReleaseCapture** Windows function.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class

Hierarchy Chart

WM_CAPTURECHANGED

Concepts

CWnd Members

CWnd::OnChangeCbChain

The framework calls this member function for each window in the Clipboard-viewer chain to notify it that a window is being removed from the chain.

```
afx_msg void OnChangeCbChain(
   HWND hWndRemove,
   HWND hWndAfter
);
```

Parameters

hWndRemove

Specifies the window handle that is being removed from the Clipboard-viewer chain.

hWndAfter

Specifies the window handle that follows the window being removed from the Clipboard-viewer chain.

Remarks

Each **CWnd** object that receives an **OnChangeCbChain** call should use the SendMessage Windows function to send the WM_CHANGECBCHAIN message to the next window in the Clipboard-viewer chain (the handle returned by **SetClipboardViewer**). If *hWndRemove* is the next window in the chain, the window specified by *hWndAfter* becomes the next window, and Clipboard messages are passed on to it.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::ChangeClipboardChain
SendMessage

Concepts

CWnd Members

CWnd::OnChangeUIState

Called when the user interface (UI) state should be changed.

```
afx_msg void OnChangeUIState(
   UINT nAction,
   UINT nUIElement
);
```

Parameters

nAction

Specifies the action to be taken. Can be one of the following values:

- **UIS_CLEAR** The UI state element (specified by *nUIElement*) should be hidden.
- **UIS_INITIALIZE** The UI state element (specified by *nUIElement*) should be changed based on the last input event. For more information, see the **Remarks** section of WM_CHANGEUISTATE.
- **UIS_SET** The UI state element (specified by *nUIElement*) should be visible.

nUIElement

Specifies which UI state elements are affected or the style of the control. Can be one of the following values:

- **UISF_HIDEACCEL** Keyboard accelerators.
- **UISF_HIDEFOCUS** Focus indicators.
- **UISF_ACTIVE** **Windows XP:** A control should be drawn in the style used for active controls.

Remarks

This member function emulates the functionality of the WM_CHANGEUISTATE message, as described in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnQueryUIState
CWnd::OnUpdateUIState

Concepts

CWnd Members

CWnd::OnChar

The framework calls this member function when a keystroke translates to a nonsystem character.

```
afx_msg void OnChar(
   UINT nChar,
   UINT nRepCnt,
   UINT nFlags
);
```

Parameters

nChar

Contains the character code value of the key.

nRepCnt

Contains the repeat count, the number of times the keystroke is repeated when user holds down the key.

nFlags

Contains the scan code, key-transition code, previous key state, and context code, as shown in the following list:

Value	Meaning
0-15	Specifies the repeat count. The value is the number of times the keystroke is repeated as a result of the user holding down the key.
16-23	Specifies the scan code. The value depends on the original equipment manufacturer (OEM)
24	Specifies whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101- or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0.
25-28	Used internally by Windows.
29	Specifies the context code. The value is 1 if the ALT key is held down while the key is pressed; otherwise, the value is 0.
30	Specifies the previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up.
31	Specifies the transition state. The value is 1 if the key is being released, or it is 0 if the key is being pressed.

Remarks

This function is called before the OnKeyUp member function and after the OnKeyDown member function are called. **OnChar** contains the value of the keyboard key being pressed or released.

Because there is not necessarily a one-to-one correspondence between keys pressed and **OnChar** calls generated, the information in *nFlags* is generally not useful to applications. The information in *nFlags* applies only to the most recent call to the **OnKeyUp** member function or the **OnKeyDown** member function that precedes the call to **OnChar**.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the slash (/) and ENTER keys in the numeric keypad. Some other keyboards may support the extended-key bit in *nFlags*.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
WM_CHAR
CWnd::OnKeyDown
CWnd::OnKeyUp

Concepts

CWnd Members

CWnd::OnCharToItem

Called when a list box with the LBS_WANTKEYBOARDINPUT style sends its owner a WM_CHARTOITEM message in response to a WM_CHAR message.

```
afx_msg int OnCharToItem(
   UINT nChar,
   CListBox* pListBox,
   UINT nIndex
);
```

Parameters

nChar

Specifies the value of the key pressed by the user.

pListBox

Specifies a pointer to the list box. It may be temporary.

nIndex

Specifies the current caret position.

Return Value

The framework calls this member function to specify the action that the application performed in response to the call. A return value of –2 indicates that the application handled all aspects of selecting the item and wants no further action by the list box. A return value of –1 indicates that the list box should perform the default action in response to the keystroke. A return value of 0 or greater specifies the zero-based index of an item in the list box and indicates that the list box should perform the default action for the keystroke on the given item.

Remarks

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnChar
WM_CHARTOITEM

Concepts

CWnd Members

CWnd::OnChildActivate

If the **CWnd** object is a multiple document interface (MDI) child window, **OnChildActivate** is called by the framework when the user clicks the window's title bar or when the window is activated, moved, or sized.

```
afx_msg void OnChildActivate( );
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::SetWindowPos
WM_CHILDACTIVATE

Concepts

CWnd Members

CWnd::OnChildNotify

This member function is called by this window's parent window when it receives a notification message that applies to this window.

```
virtual BOOL OnChildNotify(
   UINT message,
   WPARAM wParam,
   LPARAM lParam,
   LRESULT* pResult
);
```

Parameters

message

A Windows message number sent to a parent window.

wParam

The **wparam** associated with the message.

lParam

The **lparam** associated with the message.

pLResult

A pointer to a value to be returned from the parent's window procedure. This pointer will be **NULL** if no return value is expected.

Return Value

Nonzero if this window is responsible for handling the message sent to its parent; otherwise 0.

Remarks

Never call this member function directly.

The default implementation of this member function returns 0, which means that the parent should handle the message.

Override this member function to extend the manner in which a control responds to notification messages.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::OnClose

The framework calls this member function as a signal that the **CWnd** or an application is to terminate.

```
afx_msg void OnClose( );
```

Remarks

The default implementation calls **DestroyWindow**.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::DestroyWindow
WM_CLOSE

Concepts

CWnd Members

CWnd::OnCommand

The framework calls this member function when the user selects an item from a menu, when a child control sends a notification message, or when an accelerator keystroke is translated.

```
virtual BOOL OnCommand(
   WPARAM wParam,
   LPARAM lParam
);
```

Parameters

wParam

The low-order word of *wParam* identifies the command ID of the menu item, control, or accelerator. The high-order word of *wParam* specifies the notification message if the message is from a control. If the message is from an accelerator, the high-order word is 1. If the message is from a menu, the high-order word is 0.

lParam

Identifies the control that sends the message if the message is from a control. Otherwise, *lParam* is 0.

Return Value

An application returns nonzero if it processes this message; otherwise 0.

Remarks

OnCommand processes the message map for control notification and **ON_COMMAND** entries, and calls the appropriate member function.

Override this member function in your derived class to handle the WM_COMMAND message. An override will not process the message map unless the base class **OnCommand** is called.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
WM_COMMAND
CCmdTarget::OnCmdMsg

Concepts

CWnd Members

CWnd::OnCompacting

The framework calls this member function for all top-level windows when Windows detects that more than 12.5 percent of system time over a 30- to 60-second interval is being spent compacting memory.

```
afx_msg void OnCompacting(
   UINT nCpuTime
);
```

Parameters

nCpuTime

Specifies the ratio of CPU time currently spent by Windows compacting memory to CPU time spent performing other operations. For example, 8000h represents 50 percent of CPU time spent compacting memory.

Remarks

This indicates that system memory is low.

When a **CWnd** object receives this call, it should free as much memory as possible, taking into account the current level of activity of the application and the total number of applications running in Windows. The application can call the Windows function to determine how many applications are running.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class

Hierarchy Chart

WM_COMPACTING

Concepts

CWnd Members

CWnd::OnCompareItem

The framework calls this member function to specify the relative position of a new item in a child sorted owner-draw combo or list box.

```
afx_msg int OnCompareItem(
   int nIDCtl,
   LPCOMPAREITEMSTRUCT lpCompareItemStruct
);
```

Parameters

nIDCtl

The identifier of the control that sent the **WM_COMPAREITEM** message.

lpCompareItemStruct

Contains a long pointer to a COMPAREITEMSTRUCT data structure that contains the identifiers and application-supplied data for two items in the combo or list box.

Return Value

Indicates the relative position of the two items. It may be any of the following values:

Value	Meaning
–1	Item 1 sorts before item 2.
0	Item 1 and item 2 sort the same.
1	Item 1 sorts after item 2.

Remarks

If a combo or list box is created with the CBS_SORT or LBS_SORT style, Windows sends the combo-box or list-box owner a **WM_COMPAREITEM** message whenever the application adds a new item.

Two items in the combo or list box are reformed in a **COMPAREITEMSTRUCT** structure pointed to by *lpCompareItemStruct*. **OnCompareItem** should return a value that indicates which of the items should appear before the other. Typically, Windows makes this call several times until it determines the exact position for the new item.

If the **hwndItem** member of the **COMPAREITEMSTRUCT** structure belongs to a CListBox or CComboBox object, then the **CompareItem** virtual function of the appropriate class is called. Override **CComboBox::CompareItem** or **CListBox::CompareItem** in your derived **CListBox** or **CComboBox** class to do the item comparison.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
COMPAREITEMSTRUCT Structure
WM_COMPAREITEM
CListBox::CompareItem
CComboBox::CompareItem

Concepts

CWnd Members

CWnd::OnContextMenu

Called by the framework when the user has clicked the right mouse button (right clicked) in the window.

```
afx_msg void OnContextMenu(
   CWnd* pWnd,
   CPoint pos
);
```

Parameters

pWnd

Handle to the window in which the user right clicked the mouse. This can be a child window of the window receiving the message. For more information about processing this message, see the Remarks section.

pos

Position of the cursor, in screen coordinates, at the time of the mouse click.

Remarks

You can process this message by displaying a context menu using the TrackPopupMenu.

If you do not display a context menu you should pass this message onto the DefWindowProc function. If your window is a child window, **DefWindowProc** sends the message to the parent. Otherwise, **DefWindowProc** displays a default context menu if the specified position is in the window's caption.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::OnCopyData

This member function is called by the framework to copy data from one application to another.

```
afx_msg BOOL OnCopyData(
   CWnd* pWnd,
   COPYDATASTRUCT* pCopyDataStruct
);
```

Parameters

pWnd

A pointer to a CWnd object that is sending the data.

pCopyDataStruct

A pointer to a COPYDATASTRUCT structure that contains the data being sent.

Return Value

Returns **TRUE** if the receiving application successfully accepts the data. Otherwise, returns **FALSE**.

Remarks

The data being passed must not contain pointers or other references to objects not accessible to the application receiving the data.

While the data is being copied, it must not be changed by another thread of the sending process.

The receiving application should consider the data read-only. The structure pointed to by the parameter *pCopyDataStruct* is valid only during the transfer of data; however, the receiving application should not free the memory associated with the structure.

If the receiving application needs access to the data after this function returns, it must copy the data received to a local buffer.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
WM_COPYDATA

Concepts

CWnd Members

CWnd::OnCreate

The framework calls this member function when an application requests that the Windows window be created by calling the Create or CreateEx member function.

```
afx_msg int OnCreate(
   LPCREATESTRUCT lpCreateStruct
);
```

Parameters

lpCreateStruct

Points to a CREATESTRUCT structure that contains information about the **CWnd** object being created.

Return Value

OnCreate must return 0 to continue the creation of the **CWnd** object. If the application returns –1, the window will be destroyed.

Remarks

The **CWnd** object receives this call after the window is created but before it becomes visible. **OnCreate** is called before the **Create** or **CreateEx** member function returns.

Override this member function to perform any needed initialization of a derived class.

The **CREATESTRUCT** structure contains copies of the parameters used to create the window.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::CreateEx
CWnd::OnNcCreate
WM_CREATE
CWnd::Default
CWnd::FromHandle

Concepts

CWnd Members

CWnd::OnCtlColor

The framework calls this member function when a child control is about to be drawn.

```
afx_msg HBRUSH OnCtlColor(
   CDC* pDC,
   CWnd* pWnd,
   UINT nCtlColor
);
```

Parameters

pDC

Contains a pointer to the display context for the child window. May be temporary.

pWnd

Contains a pointer to the control asking for the color. May be temporary.

nCtlColor

Contains one of the following values, specifying the type of control:

- **CTLCOLOR_BTN** Button control
- **CTLCOLOR_DLG** Dialog box
- **CTLCOLOR_EDIT** Edit control
- **CTLCOLOR_LISTBOX** List-box control
- **CTLCOLOR_MSGBOX** Message box
- **CTLCOLOR_SCROLLBAR** Scroll-bar control
- **CTLCOLOR_STATIC** Static control

Return Value

OnCtlColor must return a handle to the brush that is to be used for painting the control background.

Remarks

Most controls send this message to their parent (usually a dialog box) to prepare the *pDC* for drawing the control using the correct colors.

To change the text color, call the **SetTextColor** member function with the desired red, green, and blue (RGB) values.

To change the background color of a single-line edit control, set the brush handle in both the **CTLCOLOR_EDIT** and **CTLCOLOR_MSGBOX** message codes, and call the CDC::SetBkColor function in response to the **CTLCOLOR_EDIT** code.

OnCtlColor will not be called for the list box of a drop-down combo box because the drop-down list box is actually a child of the combo box and not a child of the window. To change the color of the drop-down list box, create a **CComboBox** with an override of **OnCtlColor** that checks for **CTLCOLOR_LISTBOX** in the *nCtlColor* parameter. In this handler, the **SetBkColor** member function must be used to set the background color for the text.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

Example

```
// This OnCtlColor handler will change the color of a static control
// with the ID of IDC_MYSTATIC. The code assumes that the CMyDialog
```

```
// class has an initialized and created CBrush member named m_brush.
// The control will be painted with red text and a background
// color of m_brush.

HBRUSH CZilchDlg::OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor)
{
   // Call the base class implementation first! Otherwise, it may
   // undo what we're trying to accomplish here.
   HBRUSH hbr = CDialog::OnCtlColor(pDC, pWnd, nCtlColor);

   // Are we painting the IDC_MYSTATIC control? We can use
   // CWnd::GetDlgCtrlID() to perform the most efficient test.
   if (pWnd->GetDlgCtrlID() == IDC_MYSTATIC)
   {
      // Set the text color to red
      pDC->SetTextColor(RGB(255, 0, 0));

      // Set the background mode for text to transparent
      // so background will show thru.
      pDC->SetBkMode(TRANSPARENT);

      // Return handle to our CBrush object
      hbr = m_brush;
   }

   return hbr;
}
```

See Also

Reference

CWnd Class

Hierarchy Chart

CDC::SetBkColor

Concepts

CWnd Members

CWnd::OnDeadChar

The framework calls this member function when the OnKeyUp member function and the OnKeyDown member functions are called.

```
afx_msg void OnDeadChar(
   UINT nChar,
   UINT nRepCnt,
   UINT nFlags
);
```

Parameters

nChar

Specifies the dead-key character value.

nRepCnt

Specifies the repeat count.

nFlags

Specifies the scan code, key-transition code, previous key state, and context code, as shown in the following list:

Value	Description
0–7	Scan code (OEM-dependent value). Low byte of high-order word.
8	Extended key, such as a function key or a key on the numeric keypad (1 if it is an extended key; otherwise 0).
9–10	Not used.
11–12	Used internally by Windows.
13	Context code (1 if the ALT key is held down while the key is pressed; otherwise 0).
14	Previous key state (1 if the key is down before the call, 0 if the key is up).
15	Transition state (1 if the key is being released, 0 if the key is being pressed).

Remarks

This member function can be used to specify the character value of a dead key. A dead key is a key, such as the umlaut (double-dot) character, that is combined with other characters to form a composite character. For example, the umlaut-O character consists of the dead key, umlaut, and the O key.

An application typically uses **OnDeadChar** to give the user feedback about each key pressed. For example, an application can display the accent in the current character position without moving the caret.

Since there is not necessarily a one-to-one correspondence between keys pressed and **OnDeadChar** calls, the information in *nFlags* is generally not useful to applications. The information in *nFlags* applies only to the most recent call to the OnKeyUp member function or the OnKeyDown member function that precedes the **OnDeadChar** call.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the slash (/) and ENTER keys in the numeric keypad. Some other keyboards may support the extended-key bit in *nFlags*.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class

Hierarchy Chart

WM_DEADCHAR

Concepts

CWnd Members

CWnd::OnDeleteItem

The framework calls this member function to inform the owner of an owner-draw list box or combo box that the list box or combo box is destroyed or that items have been removed by CComboBox::DeleteString, CListBox::DeleteString, CComboBox::ResetContent, or CListBox::ResetContent.

```
afx_msg void OnDeleteItem(
   int nIDCtl,
   LPDELETEITEMSTRUCT lpDeleteItemStruct
);
```

Parameters

nIDCtl

The identifier of the control that sent the **WM_DELETEITEM** message.

lpDeleteItemStruct

Specifies a long pointer to a DELETEITEMSTRUCT data structure that contains information about the deleted list box item.

Remarks

If the **hwndItem** member of the **DELETEITEMSTRUCT** structure belongs to a combo box or list box, then the **DeleteItem** virtual function of the appropriate class is called. Override the **DeleteItem** member function of the appropriate control's class to delete item-specific data.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CComboBox::DeleteString
CListBox::DeleteString
CComboBox::ResetContent
CListBox::ResetContent
WM_DELETEITEM
CListBox::DeleteItem
CComboBox::DeleteItem

Concepts

CWnd Members

CWnd::OnDestroy

The framework calls this member function to inform the **CWnd** object that it is being destroyed.

```
afx_msg void OnDestroy( );
```

Remarks

OnDestroy is called after the **CWnd** object is removed from the screen.

OnDestroy is called first for the **CWnd** being destroyed, then for the child windows of **CWnd** as they are destroyed. It can be assumed that all child windows still exist while **OnDestroy** runs.

If the **CWnd** object being destroyed is part of the Clipboard-viewer chain (set by calling the SetClipboardViewer member function), the **CWnd** must remove itself from the Clipboard-viewer chain by calling the ChangeClipboardChain member function before returning from the **OnDestroy** function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::ChangeClipboardChain
CWnd::DestroyWindow
CWnd::SetClipboardViewer

Concepts

CWnd Members

CWnd::OnDestroyClipboard

The framework calls this member function for the Clipboard owner when the Clipboard is emptied through a call to the EmptyClipboard Windows function.

```
afx_msg void OnDestroyClipboard( );
```

See Also

Reference

CWnd Class
Hierarchy Chart
EmptyClipboard
WM_DESTROYCLIPBOARD

Concepts

CWnd Members

CWnd::OnDeviceChange

The framework calls this member function to notify an application or device driver of a change to the hardware configuration of a device or the computer.

```
afx_msg BOOL OnDeviceChange(
   UINT nEventType,
   DWORD_PTR dwData
);
```

Parameters

nEventType

An event type. See the Remarks section for a description of the available values

dwData

The address of a structure that contains event-specific data. Its meaning depends on the given event.

Remarks

For devices that offer software-controllable features, such as ejection and locking, the operating system typically sends a **DBT_DEVICEREMOVEPENDING** message to let applications and device drivers end their use of the device gracefully.

If the operating system forcefully removes of a device, it may not send a **DBT_DEVICEQUERYREMOVE** message before doing so.

The *nEvent* parameter can be one of these values:

- DBT_DEVICEARRIVAL A device has been inserted and is now available.
- DBT_DEVICEQUERYREMOVE Permission to remove a device is requested. Any application can deny this request and cancel the removal.
- DBT_DEVICEQUERYREMOVEFAILED Request to remove a device has been canceled.
- DBT_DEVICEREMOVEPENDING Device is about to be removed. Cannot be denied.
- DBT_DEVICEREMOVECOMPLETE Device has been removed.
- DBT_DEVICETYPESPECIFIC Device-specific event.
- DBT_CONFIGCHANGED Current configuration has changed.
- **DBT_DEVNODES_CHANGED** Device node has changed.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
WM_DEVICECHANGE

Concepts

CWnd Members

CWnd::OnDevModeChange

The framework calls this member function for all top-level **CWnd** objects when the user changes device-mode settings.

```
afx_msg void OnDevModeChange(
   LPTSTR lpDeviceName
);
```

Parameters

lpDeviceName

Points to the device name specified in the Windows initialization file, WIN.INI.

Remarks

Applications that handle the **WM_DEVMODECHANGE** message may reinitialize their device-mode settings. Applications that use the Windows **ExtDeviceMode** function to save and restore device settings typically do not process this function.

This function is not called when the user changes the default printer from Control Panel. In this case, the **OnWinIniChange** function is called.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class

Hierarchy Chart

WM_DEVMODECHANGE

Concepts

CWnd Members

CWnd::OnDrawClipboard

The framework calls this member function for each window in the Clipboard-viewer chain when the contents of the Clipboard change.

```
afx_msg void OnDrawClipboard( );
```

Remarks

Only applications that have joined the Clipboard-viewer chain by calling the SetClipboardViewer member function need to respond to this call.

Each window that receives an **OnDrawClipboard** call should call the SendMessage Windows function to pass a WM_DRAWCLIPBOARD message on to the next window in the Clipboard-viewer chain. The handle of the next window is returned by the SetClipboardViewer member function; it may be modified in response to an OnChangeCbChain member function call.

See Also

Reference

CWnd Class
Hierarchy Chart
SendMessage
CWnd::SetClipboardViewer
CWnd::OnChangeCbChain
WM_DRAWCLIPBOARD

Concepts

CWnd Members

CWnd::OnDrawItem

The framework calls this member function for the owner of an owner-draw button control, combo-box control, list-box control, or menu when a visual aspect of the control or menu has changed.

```
afx_msg void OnDrawItem(
   int nIDCtl,
   LPDRAWITEMSTRUCT lpDrawItemStruct
);
```

Parameters

nIDCtl

Contains the identifier of the control that sent the **WM_DRAWITEM** message. If a menu sent the message, *nIDCtl* contains 0.

lpDrawItemStruct

Specifies a long pointer to a **DRAWITEMSTRUCT** data structure that contains information about the item to be drawn and the type of drawing required.

Remarks

The **itemAction** member of the DRAWITEMSTRUCT structure defines the drawing operation that is to be performed. The data in this member allows the owner of the control to determine what drawing action is required.

Before returning from processing this message, an application should ensure that the device context identified by the **hDC** member of the **DRAWITEMSTRUCT** structure is restored to the default state.

If the **hwndItem** member belongs to a CButton, CMenu, CListBox, or CComboBox object, then the **DrawItem** virtual function of the appropriate class is called. Override the **DrawItem** member function of the appropriate control's class to draw the item.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
DRAWITEMSTRUCT Structure
WM_DRAWITEM
CButton::DrawItem
CMenu::DrawItem
CListBox::DrawItem
CComboBox::DrawItem

Concepts

CWnd Members

CWnd::OnDropFiles

The framework calls this member function when the user releases the left mouse button over a window that has registered itself as the recipient of dropped files.

```
afx_msg void OnDropFiles(
   HDROP hDropInfo
);
```

Parameters

hDropInfo

A pointer to an internal data structure that describes the dropped files. This handle is used by the **DragFinish**, **DragQueryFile**, and **DragQueryPoint** Windows functions to retrieve information about the dropped files.

Remarks

Typically, a derived class will be designed to support dropped files and it will register itself during window construction.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::DragAcceptFiles
WM_DROPFILES
DragAcceptFiles
DragFinish
DragQueryFile
DragQueryPoint

Concepts

CWnd Members

CWnd::OnEnable

The framework calls this member function when an application changes the enabled state of the **CWnd** object.

```
afx_msg void OnEnable(
   BOOL bEnable
);
```

Parameters

bEnable

Specifies whether the **CWnd** object has been enabled or disabled. This parameter is **TRUE** if the **CWnd** has been enabled; it is **FALSE** if the **CWnd** has been disabled.

Remarks

OnEnable is called before the EnableWindow member function returns, but after the window enabled state (WS_DISABLED style bit) has changed.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::EnableWindow
WM_ENABLE

Concepts

CWnd Members

CWnd::OnEndSession

The framework calls this member function after the **CWnd** object has returned a nonzero value from a OnQueryEndSession member function call.

```
afx_msg void OnEndSession(
   BOOL bEnding
);
```

Parameters

bEnding

Specifies whether or not the session is being ended. It is **TRUE** if the session is being ended; otherwise **FALSE**.

Remarks

The **OnEndSession** call informs the **CWnd** object whether the session is actually ending.

If *bEnding* is **TRUE**, Windows can terminate any time after all applications have returned from processing this call. Consequently, have an application perform all tasks required for termination within **OnEndSession**.

You do not need to call the DestroyWindow member function or PostQuitMessage Windows function when the session is ending.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::DestroyWindow
CWnd::OnQueryEndSession
ExitWindows
PostQuitMessage
CWnd::OnQueryEndSession
CWnd::Default
WM_ENDSESSION

Concepts

CWnd Members

CWnd::OnEnterIdle

The framework calls this member function to inform an application's main window procedure that a modal dialog box or a menu is entering an idle state.

```
afx_msg void OnEnterIdle(
   UINT nWhy,
   CWnd* pWho
);
```

Parameters

nWhy

Specifies whether the message is the result of a dialog box or a menu being displayed. This parameter can be one of the following values:

- **MSGF_DIALOGBOX** The system is idle because a dialog box is being displayed.
- **MSGF_MENU** The system is idle because a menu is being displayed.

pWho

Specifies a pointer to the dialog box (if *nWhy* is **MSGF_DIALOGBOX**), or the window that contains the displayed menu (if *nWhy* is **MSGF_MENU**). This pointer may be temporary and should not be stored for later use.

Remarks

A modal dialog box or menu enters an idle state when no messages are waiting in its queue after it has processed one or more previous messages.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
WM_ENTERIDLE

Concepts

CWnd Members

CWnd::OnEnterMenuLoop

The framework calls this member function when a menu modal loop has been entered.

```
afx_msg void OnEnterMenuLoop(
   BOOL bIsTrackPopupMenu
);
```

Parameters

bIsTrackPopupMenu

Specifies whether the menu involved is a popup menu. Has a nonzero value if the function is successful; otherwise 0.

Remarks

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnExitMenuLoop
WM_ENTERMENULOOP

Concepts

CWnd Members

CWnd::OnEraseBkgnd

The framework calls this member function when the **CWnd** object background needs erasing (for example, when resized).

```
afx_msg BOOL OnEraseBkgnd(
   CDC* pDC
);
```

Parameters

pDC

Specifies the device-context object.

Return Value

Nonzero if it erases the background; otherwise 0.

Remarks

It is called to prepare an invalidated region for painting.

The default implementation erases the background using the window class background brush specified by the **hbrBackground** member of the window class structure.

If the **hbrBackground** member is **NULL**, your overridden version of **OnEraseBkgnd** should erase the background color. Your version should also align the origin of the intended brush with the **CWnd** coordinates by first calling UnrealizeObject for the brush, and then selecting the brush.

An overridden **OnEraseBkgnd** should return nonzero in response to **WM_ERASEBKGND** if it processes the message and erases the background; this indicates that no further erasing is required. If it returns 0, the window will remain marked as needing to be erased. (Typically, this means that the **fErase** member of the **PAINTSTRUCT** structure will be **TRUE**.)

Windows assumes the background is computed with the **MM_TEXT** mapping mode. If the device context is using any other mapping mode, the area erased may not be within the visible part of the client area.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnIconEraseBkgnd
CGdiObject::UnrealizeObject
WM_ERASEBKGND

Concepts

CWnd Members

CWnd::OnExitMenuLoop

The framework calls this member function when a menu modal loop has been exited.

```
afx_msg void OnExitMenuLoop(
   BOOL bIsTrackPopupMenu
);
```

Parameters

bIsTrackPopupMenu

Specifies whether the menu involved is a pop-up menu. Has a nonzero value if the function is successful; otherwise 0.

Remarks

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnEnterMenuLoop
WM_EXITMENULOOP

Concepts

CWnd Members

CWnd::OnFontChange

All top-level windows in the system receive an **OnFontChange** call from the framework after the application changes the pool of font resources.

```
afx_msg void OnFontChange( );
```

Remarks

An application that adds or removes fonts from the system (for example, through the AddFontResource or RemoveFontResource Windows function) should send the WM_FONTCHANGE message to all top-level windows.

To send this message, use the SendMessage Windows function with the *hWnd* parameter set to **HWND_BROADCAST**.

See Also

Reference

CWnd Class
Hierarchy Chart
AddFontResource
RemoveFontResource
SendMessage
WM_FONTCHANGE

Concepts

CWnd Members

CWnd::OnGetDlgCode

Called for a control so the control can process arrow-key and TAB-key input itself.

```
afx_msg UINT OnGetDlgCode( );
```

Return Value

One or more of the following values, indicating which type of input the application processes:

- **DLGC_BUTTON** Button (generic).
- **DLGC_DEFPUSHBUTTON** Default pushbutton.
- **DLGC_HASSETSEL** **EM_SETSEL** messages.
- **DLGC_UNDEFPUSHBUTTON** No default pushbutton processing. (An application can use this flag with **DLGC_BUTTON** to indicate that it processes button input but relies on the system for default pushbutton processing.)
- **DLGC_RADIOBUTTON** Radio button.
- **DLGC_STATIC** Static control.
- **DLGC_WANTALLKEYS** All keyboard input.
- **DLGC_WANTARROWS** Arrow keys.
- **DLGC_WANTCHARS** **WM_CHAR** messages.
- **DLGC_WANTMESSAGE** All keyboard input. The application passes this message on to the control.
- **DLGC_WANTTAB** TAB key.

Remarks

Normally, Windows handles all arrow-key and TAB-key input to a **CWnd** control. By overriding **OnGetDlgCode**, a **CWnd** control can choose a particular type of input to process itself.

The default **OnGetDlgCode** functions for the predefined control classes return a code appropriate for each class.

See Also

Reference

CWnd Class
Hierarchy Chart
WM_GETDLGCODE

Concepts

CWnd Members

CWnd::OnGetMinMaxInfo

The framework calls this member function whenever Windows needs to know the maximized position or dimensions, or the minimum or maximum tracking size.

```
afx_msg void OnGetMinMaxInfo(
   MINMAXINFO* lpMMI
);
```

Parameters

lpMMI

Points to a **MINMAXINFO** structure that contains information about a window's maximized size and position and its minimum and maximum tracking size. For more about this structure, see the MINMAXINFO structure.

Remarks

The maximized size is the size of the window when its borders are fully extended. The maximum tracking size of the window is the largest window size that can be achieved by using the borders to size the window. The minimum tracking size of the window is the smallest window size that can be achieved by using the borders to size the window.

Windows fills in an array of points specifying default values for the various positions and dimensions. The application may change these values in **OnGetMinMaxInfo**.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class

Hierarchy Chart

WM_GETMINMAXINFO

Concepts

CWnd Members

CWnd::OnHelp

Handles F1 Help within the application (using the current context).

```
afx_msg void OnHelp( );
```

Remarks

See CWinApp::OnHelp for more information.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::HtmlHelp
CWnd::WinHelp

Concepts

CWnd Members

CWnd::OnHelpFinder

Handles the **ID_HELP_FINDER** and **ID_DEFAULT_HELP** commands.

```
afx_msg void OnHelpFinder( );
```

Remarks

See CWinApp::OnHelpFinder for more information.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnHelp
CWnd::HtmlHelp
CWnd::WinHelp
CWnd::OnHelpUsing

Concepts

CWnd Members

CWnd::OnHelpIndex

Handles the **ID_HELP_INDEX** command and provides a default Help topic.

```
afx_msg void OnHelpIndex( );
```

Remarks

See CWinApp::OnHelpIndex for more information.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnHelpFinder
CWnd::OnHelp
CWnd::OnHelpUsing

Concepts

CWnd Members

CWnd::OnHelpInfo

Called by the framework when the user presses the F1 key.

```
afx_msg BOOL OnHelpInfo(
   HELPINFO* lpHelpInfo
);
```

Parameters

lpHelpInfo

Pointer to a HELPINFO structure that contains information about the menu item, control, dialog box, or window for which help is requested.

Return Value

Returns TRUE if a window has the keyboard focus or if a menu is active within a window. If no window has the keyboard focus, returns FALSE.

Remarks

If a menu is active when F1 is pressed, **WM_HELP** is sent to the window associated with the menu; otherwise, **WM_HELP** is sent to the window that has the keyboard focus. If no window has the keyboard focus, **WM_HELP** is sent to the currently active window.

See Also

Reference

CWnd Class
Hierarchy Chart
CWinApp::OnHelp
CWinApp::WinHelp

Concepts

CWnd Members

CWnd::OnHelpUsing

Handles the **ID_HELP_USING** command.

```
afx_msg void OnHelpUsing( );
```

Remarks

See CWinApp::OnHelpUsing for more information.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnHelpIndex
CWnd::OnHelp
CWnd::OnHelpFinder

Concepts

CWnd Members

CWnd::OnHScroll

The framework calls this member function when the user clicks a window's horizontal scroll bar.

```
afx_msg void OnHScroll(
   UINT nSBCode,
   UINT nPos,
   CScrollBar* pScrollBar
);
```

Parameters

nSBCode

Specifies a scroll-bar code that indicates the user's scrolling request. This parameter can be one of the following:

- **SB_LEFT** Scroll to far left.
- **SB_ENDSCROLL** End scroll.
- **SB_LINELEFT** Scroll left.
- **SB_LINERIGHT** Scroll right.
- **SB_PAGELEFT** Scroll one page left.
- **SB_PAGERIGHT** Scroll one page right.
- **SB_RIGHT** Scroll to far right.
- **SB_THUMBPOSITION** Scroll to absolute position. The current position is specified by the *nPos* parameter.
- **SB_THUMBTRACK** Drag scroll box to specified position. The current position is specified by the *nPos* parameter.

nPos

Specifies the scroll-box position if the scroll-bar code is **SB_THUMBPOSITION** or **SB_THUMBTRACK**; otherwise, not used. Depending on the initial scroll range, *nPos* may be negative and should be cast to an **int** if necessary.

pScrollBar

If the scroll message came from a scroll-bar control, contains a pointer to the control. If the user clicked a window's scroll bar, this parameter is **NULL**. The pointer may be temporary and should not be stored for later use.

Remarks

The **SB_THUMBTRACK** scroll-bar code typically is used by applications that give some feedback while the scroll box is being dragged.

If an application scrolls the contents controlled by the scroll bar, it must also reset the position of the scroll box with the SetScrollPos member function.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

Example

```
void CMyView::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
   // Get the minimum and maximum scroll-bar positions.
   int minpos;
   int maxpos;
   pScrollBar->GetScrollRange(&minpos, &maxpos);
   maxpos = pScrollBar->GetScrollLimit();
```

```
    // Get the current position of scroll box.
    int curpos = pScrollBar->GetScrollPos();

    // Determine the new position of scroll box.
    switch (nSBCode)
    {
    case SB_LEFT:      // Scroll to far left.
       curpos = minpos;
       break;

    case SB_RIGHT:      // Scroll to far right.
       curpos = maxpos;
       break;

    case SB_ENDSCROLL:   // End scroll.
       break;

    case SB_LINELEFT:      // Scroll left.
       if (curpos > minpos)
          curpos--;
       break;

    case SB_LINERIGHT:   // Scroll right.
       if (curpos < maxpos)
          curpos++;
       break;

    case SB_PAGELEFT:    // Scroll one page left.
    {
       // Get the page size.
       SCROLLINFO   info;
       pScrollBar->GetScrollInfo(&info, SIF_ALL);

       if (curpos > minpos)
       curpos = max(minpos, curpos - (int) info.nPage);
    }
       break;

    case SB_PAGERIGHT:      // Scroll one page right.
    {
       // Get the page size.
       SCROLLINFO   info;
       pScrollBar->GetScrollInfo(&info, SIF_ALL);

       if (curpos < maxpos)
          curpos = min(maxpos, curpos + (int) info.nPage);
    }
       break;

    case SB_THUMBPOSITION: // Scroll to absolute position. nPos is the position
       curpos = nPos;      // of the scroll box at the end of the drag operation.
       break;

    case SB_THUMBTRACK:   // Drag scroll box to specified position. nPos is the
       curpos = nPos;     // position that the scroll box has been dragged to.
       break;
    }

    // Set the new position of the thumb (scroll box).
    pScrollBar->SetScrollPos(curpos);

    CView::OnHScroll(nSBCode, nPos, pScrollBar);
  }
```

See Also

Reference

CWnd Class

Hierarchy Chart
CWnd::SetScrollPos
CWnd::OnVScroll
WM_HSCROLL
Concepts
CWnd Members

CWnd::OnHScrollClipboard

The Clipboard owner's **OnHScrollClipboard** member function is called by the Clipboard viewer when the Clipboard data has the **CF_OWNERDISPLAY** format and there is an event in the Clipboard viewer's horizontal scroll bar.

```
afx_msg void OnHScrollClipboard(
   CWnd* pClipAppWnd,
   UINT nSBCode,
   UINT nPos
);
```

Parameters

pClipAppWnd

Specifies a pointer to a Clipboard-viewer window. The pointer may be temporary and should not be stored for later use.

nSBCode

Specifies one of the following scroll-bar codes in the low-order word:

- **SB_BOTTOM** Scroll to lower right.
- **SB_ENDSCROLL** End scroll.
- **SB_LINEDOWN** Scroll one line down.
- **SB_LINEUP** Scroll one line up.
- **SB_PAGEDOWN** Scroll one page down.
- **SB_PAGEUP** Scroll one page up.
- **SB_THUMBPOSITION** Scroll to the absolute position. The current position is provided in *nPos*.
- **SB_TOP** Scroll to upper left.

nPos

Contains the scroll-box position if the scroll-bar code is **SB_THUMBPOSITION**; otherwise not used.

Remarks

The owner should scroll the Clipboard image, invalidate the appropriate section, and update the scroll-bar values.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnVScrollClipboard
WM_HSCROLLCLIPBOARD

Concepts

CWnd Members

CWnd::OnIconEraseBkgnd

The framework calls this member function for a minimized (iconic) **CWnd** object when the background of the icon must be filled before painting the icon.

```
afx_msg void OnIconEraseBkgnd(
   CDC* pDC
);
```

Parameters

pDC

Specifies the device-context object of the icon. May be temporary and should not be stored for later use.

Remarks

CWnd receives this call only if a class icon is defined for the window default implementation; otherwise OnEraseBkgnd is called.

The DefWindowProc member function fills the icon background with the background brush of the parent window.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnEraseBkgnd
WM_ICONERASEBKGND

Concepts

CWnd Members

CWnd::OnInitMenu

The framework calls this member function when a menu is about to become active.

```
afx_msg void OnInitMenu(
   CMenu* pMenu
);
```

Parameters

pMenu

Specifies the menu to be initialized. May be temporary and should not be stored for later use.

Remarks

OnInitMenu is called when the user clicks an item on the menu bar or presses a menu key. Override this member function to modify the menu before it is displayed.

OnInitMenu is only called once, when a menu is first accessed (for example, when a user clicks an item on the menu bar). This method does not provide information about menu items. As the user moves to items within the menu (for example, by moving the mouse across several menu items) the function is not called again. Once the user exits from the menu (for example, by clicking on the application client area) and later clicks an item on the menu bar, the function will be called again.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnInitMenuPopup
WM_INITMENU

Concepts

CWnd Members

CWnd::OnInitMenuPopup

The framework calls this member function when a pop-up menu is about to become active.

```
afx_msg void OnInitMenuPopup(
   CMenu* pPopupMenu,
   UINT nIndex,
   BOOL bSysMenu
);
```

Parameters

pPopupMenu

Specifies the menu object of the pop-up menu. May be temporary and should not be stored for later use.

nIndex

Specifies the index of the pop-up menu in the main menu.

bSysMenu

TRUE if the pop-up menu is the Control menu; otherwise **FALSE**.

Remarks

This allows an application to modify the pop-up menu before it is displayed without changing the entire menu.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnInitMenu
WM_INITMENUPOPUP

Concepts

CWnd Members

CWnd::OnKeyDown

The framework calls this member function when a nonsystem key is pressed.

```
afx_msg void OnKeyDown(
   UINT nChar,
   UINT nRepCnt,
   UINT nFlags
);
```

Parameters

nChar

Specifies the virtual key code of the given key. For a list of of standard virtual key codes, see Winuser.h

nRepCnt

Repeat count (the number of times the keystroke is repeated as a result of the user holding down the key).

nFlags

Specifies the scan code, key-transition code, previous key state, and context code, as shown in the following list:

Value	Description
0–7	Scan code (OEM-dependent value).
8	Extended key, such as a function key or a key on the numeric keypad (1 if it is an extended key).
9–10	Not used.
11–12	Used internally by Windows.
13	Context code (1 if the ALT key is held down while the key is pressed; otherwise 0).
14	Previous key state (1 if the key is down before the call, 0 if the key is up).
15	Transition state (1 if the key is being released, 0 if the key is being pressed).

For a **WM_KEYDOWN** message, the key-transition bit (bit 15) is 0 and the context-code bit (bit 13) is 0.

Remarks

A nonsystem key is a keyboard key that is pressed when the ALT key is not pressed or a keyboard key that is pressed when **CWnd** has the input focus.

Because of auto-repeat, more than one **OnKeyDown** call may occur before an OnKeyUp member function call is made. The bit that indicates the previous key state can be used to determine whether the **OnKeyDown** call is the first down transition or a repeated down transition.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the slash (/) and ENTER keys in the numeric keypad. Some other keyboards may support the extended-key bit in *nFlags*.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnChar
CWnd::OnKeyUp
WM_KEYDOWN

Concepts

CWnd Members

CWnd::OnKeyUp

The framework calls this member function when a nonsystem key is released.

```
afx_msg void OnKeyUp(
   UINT nChar,
   UINT nRepCnt,
   UINT nFlags
);
```

Parameters

nChar

Specifies the virtual key code of the given key. For a list of of standard virtual key codes, see Winuser.h

nRepCnt

Repeat count (the number of times the keystroke is repeated as a result of the user holding down the key).

nFlags

Specifies the scan code, key-transition code, previous key state, and context code, as shown in the following list:

Value	Description
0–7	Scan code (OEM-dependent value). Low byte of high-order word.
8	Extended key, such as a function key or a key on the numeric keypad (1 if it is an extended key; otherwise 0).
9–10	Not used.
11–12	Used internally by Windows.
13	Context code (1 if the ALT key is held down while the key is pressed; otherwise 0).
14	Previous key state (1 if the key is down before the call, 0 if the key is up).
15	Transition state (1 if the key is being released, 0 if the key is being pressed).

For a **WM_KEYUP** message, the key-transition bit (bit 15) is 1 and the context-code bit (bit 13) is 0.

Remarks

A nonsystem key is a keyboard key that is pressed when the ALT key is not pressed or a keyboard key that is pressed when the **CWnd** has the input focus.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the slash (/) and ENTER keys in the numeric keypad. Some other keyboards may support the extended-key bit in *nFlags*.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class

Hierarchy Chart
CWnd::OnChar
WM_KEYUP
CWnd::Default
WM_KEYDOWN
Concepts
CWnd Members

CWnd::OnKillFocus

The framework calls this member function immediately before losing the input focus.

```
afx_msg void OnKillFocus(
   CWnd* pNewWnd
);
```

Parameters

pNewWnd

Specifies a pointer to the window that receives the input focus (may be **NULL** or may be temporary).

Remarks

If the **CWnd** object is displaying a caret, the caret should be destroyed at this point.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::SetFocus
WM_KILLFOCUS

Concepts

CWnd Members

CWnd::OnLButtonDblClk

The framework calls this member function when the user double-clicks the left mouse button.

```
afx_msg void OnLButtonDblClk(
   UINT nFlags,
   CPoint point
);
```

Parameters

nFlags

Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if the CTRL key is down.
- **MK_LBUTTON** Set if the left mouse button is down.
- **MK_MBUTTON** Set if the middle mouse button is down.
- **MK_RBUTTON** Set if the right mouse button is down.
- **MK_SHIFT** Set if the SHIFT key is down.

point

Specifies the x- and y-coordinate of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks

Only windows that have the **CS_DBLCLKS** WNDCLASS style will receive **OnLButtonDblClk** calls. This is the default for Microsoft Foundation Class windows. Windows calls **OnLButtonDblClk** when the user presses, releases, and then presses the left mouse button again within the system's double-click time limit. Double-clicking the left mouse button actually generates four events: WM_LBUTTONDOWN, WM_LBUTTONUP messages, the **WM_LBUTTONDBLCLK** call, and another **WM_LBUTTONUP** message when the button is released.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnLButtonDown
CWnd::OnLButtonUp
WM_LBUTTONDBLCLK

Concepts

CWnd Members

CWnd::OnLButtonDown

The framework calls this member function when the user presses the left mouse button.

```
afx_msg void OnLButtonDown(
   UINT nFlags,
   CPoint point
);
```

Parameters

nFlags

Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if the CTRL key is down.
- **MK_LBUTTON** Set if the left mouse button is down.
- **MK_MBUTTON** Set if the middle mouse button is down.
- **MK_RBUTTON** Set if the right mouse button is down.
- **MK_SHIFT** Set if the SHIFT key is down.

point

Specifies the x- and y-coordinate of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnLButtonDblClk
CWnd::OnLButtonUp
CWnd::OnLButtonDown

Concepts

CWnd Members

CWnd::OnLButtonUp

The framework calls this member function when the user releases the left mouse button.

```
afx_msg void OnLButtonUp(
   UINT nFlags,
   CPoint point
);
```

Parameters

nFlags

Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if the CTRL key is down.
- **MK_MBUTTON** Set if the middle mouse button is down.
- **MK_RBUTTON** Set if the right mouse button is down.
- **MK_SHIFT** Set if the SHIFT key is down.

point

Specifies the x- and y-coordinate of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnLButtonDblClk
CWnd::OnLButtonDown
CWnd::OnLButtonUp

Concepts

CWnd Members

CWnd::OnMButtonDblClk

The framework calls this member function when the user double-clicks the middle mouse button.

```
afx_msg void OnMButtonDblClk(
   UINT nFlags,
   CPoint point
);
```

Parameters

nFlags

Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if the CTRL key is down.
- **MK_LBUTTON** Set if the left mouse button is down.
- **MK_MBUTTON** Set if the middle mouse button is down.
- **MK_RBUTTON** Set if the right mouse button is down.
- **MK_SHIFT** Set if the SHIFT key is down.

point

Specifies the x- and y-coordinate of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks

Only windows that have the **CS_DBLCLKS** WNDCLASS style will receive **OnMButtonDblClk** calls. This is the default for all Microsoft Foundation Class windows. Windows generates an **OnMButtonDblClk** call when the user presses, releases, and then presses the middle mouse button again within the system's double-click time limit. Double-clicking the middle mouse button actually generates four events: WM_MBUTTONDOWN and WM_MBUTTONUP messages, the **WM_MBUTTONDBLCLK** call, and another **WM_MBUTTONUP** message.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnMButtonDown
CWnd::OnMButtonUp
CWnd::OnMButtonDblClk

Concepts

CWnd Members

CWnd::OnMButtonDown

The framework calls this member function when the user presses the middle mouse button.

```
afx_msg void OnMButtonDown(
   UINT nFlags,
   CPoint point
);
```

Parameters

nFlags

Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if the CTRL key is down.
- **MK_LBUTTON** Set if the left mouse button is down.
- **MK_MBUTTON** Set if the middle mouse button is down.
- **MK_RBUTTON** Set if the right mouse button is down.
- **MK_SHIFT** Set if the SHIFT key is down.

point

Specifies the x- and y-coordinate of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnMButtonDblClk
CWnd::OnMButtonUp
CWnd::OnMButtonDown

Concepts

CWnd Members

CWnd::OnMButtonUp

The framework calls this member function when the user releases the middle mouse button.

```
afx_msg void OnMButtonUp(
   UINT nFlags,
   CPoint point
);
```

Parameters

nFlags

Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if the CTRL key is down.
- **MK_LBUTTON** Set if the left mouse button is down.
- **MK_RBUTTON** Set if the right mouse button is down.
- **MK_SHIFT** Set if the SHIFT key is down.

point

Specifies the x- and y-coordinate of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnMButtonDblClk
CWnd::OnMButtonDown
CWnd::OnMButtonUp

Concepts

CWnd Members

CWnd::OnMDIActivate

The framework calls this member function for the child window being deactivated and the child window being activated.

```
afx_msg void OnMDIActivate(
   BOOL bActivate,
   CWnd* pActivateWnd,
   CWnd* pDeactivateWnd
);
```

Parameters

bActivate

TRUE if the child is being activated and **FALSE** if it is being deactivated.

pActivateWnd

Contains a pointer to the MDI child window to be activated. When received by an MDI child window, *pActivateWnd* contains a pointer to the child window being activated. This pointer may be temporary and should not be stored for later use.

pDeactivateWnd

Contains a pointer to the MDI child window being deactivated. This pointer may be temporary and should not be stored for later use.

Remarks

An MDI child window is activated independently of the MDI frame window. When the frame becomes active, the child window that was last activated with a **OnMDIActivate** call receives an WM_NCACTIVATE message to draw an active window frame and caption bar, but it does not receive another **OnMDIActivate** call.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CMDIFrameWnd::MDIActivate
CMDIFrameWnd::MDIActivate

Concepts

CWnd Members

CWnd::OnMeasureItem

The framework calls this member function by the framework for the owner of an owner-draw button, combo box, list box, or menu item when the control is created.

```
afx_msg void OnMeasureItem(
   int nIDCtl,
      LPMEASUREITEMSTRUCT lpMeasureItemStruct
);
```

Parameters

nIDCtl

The ID of the control.

lpMeasureItemStruct

Points to a MEASUREITEMSTRUCT data structure that contains the dimensions of the owner-draw control.

Remarks

Override this member function and fill in the **MEASUREITEMSTRUCT** data structure pointed to by *lpMeasureItemStruct* and return; this informs Windows of the dimensions of the control and allows Windows to process user interaction with the control correctly.

If a list box or combo box is created with the LBS_OWNERDRAWVARIABLE or CBS_OWNERDRAWVARIABLE style, the framework calls this function for the owner for each item in the control; otherwise this function is called once.

Windows initiates the call to **OnMeasureItem** for the owner of combo boxes and list boxes created with the **OWNERDRAWFIXED** style before sending the WM_INITDIALOG message. As a result, when the owner receives this call, Windows has not yet determined the height and width of the font used in the control; function calls and calculations that require these values should occur in the main function of the application or library.

If the item being measured is a **CMenu**, **CListBox** or **CComboBox** object, then the **MeasureItem** virtual function of the appropriate class is called. Override the **MeasureItem** member function of the appropriate control's class to calculate and set the size of each item.

OnMeasureItem will be called only if the control's class is created at run time, or it is created with the **LBS_OWNERDRAWVARIABLE** or **CBS_OWNERDRAWVARIABLE** style. If the control is created by the dialog editor, **OnMeasureItem** will not be called. This is because the WM_MEASUREITEM message is sent early in the creation process of the control. If you subclass by using **DDX_Control**, **SubclassDlgItem**, or **SubclassWindow**, the subclassing usually occurs after the creation process. Therefore, there is no way to handle the WM_MEASUREITEM message in the control's **OnChildNotify** function, which is the mechanism MFC uses to implement **ON_WM_MEASUREITEM_REFLECT**.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CMenu::MeasureItem
CListBox::MeasureItem
CComboBox::MeasureItem
CComboBox::MeasureItem

Concepts

CWnd Members

CWnd::OnMenuChar

The framework calls this member function when the user presses a menu mnemonic character that doesn't match any of the predefined mnemonics in the current menu.

```
afx_msg LRESULT OnMenuChar(
   UINT nChar,
   UINT nFlags,
   CMenu* pMenu
);
```

Parameters

nChar

Depending on the build settings, specifies the ANSI or Unicode character that the user pressed.

nFlags

Contains the **MF_POPUP** flag if the menu is a pop-up menu. It contains the **MF_SYSMENU** flag if the menu is a Control menu.

pMenu

Contains a pointer to the selected **CMenu**. The pointer may be temporary and should not be stored.

Return Value

The high-order word of the return value should contain one of the following command codes:

Value	Description
0	Tells Windows to discard the character that the user pressed and creates a short beep on the system speaker.
1	Tells Windows to close the current menu.
2	Informs Windows that the low-order word of the return value contains the item number for a specific item. This item is selected by Windows.

The low-order word is ignored if the high-order word contains 0 or 1. Applications should process this message when accelerator (shortcut) keys are used to select bitmaps placed in a menu.

Remarks

It is sent to the **CWnd** that owns the menu. **OnMenuChar** is also called when the user presses ALT and any other key, even if the key does not correspond to a mnemonic character. In this case, *pMenu* points to the menu owned by the **CWnd**, and *nFlags* is 0.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::OnMenuChar

Concepts

CWnd Members

CWnd::OnMenuSelect

If the **CWnd** object is associated with a menu, **OnMenuSelect** is called by the framework when the user selects a menu item.

```
afx_msg void OnMenuSelect(
   UINT nItemID,
   UINT nFlags,
   HMENU hSysMenu
);
```

Parameters

nItemID

Identifies the item selected. If the selected item is a menu item, *nItemID* contains the menu-item ID. If the selected item contains a pop-up menu, *nItemID* contains the pop-up menu index, and *hSysMenu* contains the handle of the main (clicked-on) menu.

nFlags

Contains a combination of the following menu flags:

- **MF_BITMAP** Item is a bitmap.
- **MF_CHECKED** Item is checked.
- **MF_DISABLED** Item is disabled.
- **MF_GRAYED** Item is dimmed.
- **MF_MOUSESELECT** Item was selected with a mouse.
- **MF_OWNERDRAW** Item is an owner-draw item.
- **MF_POPUP** Item contains a pop-up menu.
- **MF_SEPARATOR** Item is a menu-item separator.
- **MF_SYSMENU** Item is contained in the Control menu.

hSysMenu

If *nFlags* contains **MF_SYSMENU**, identifies the menu associated with the message. If *nFlags* contains **MF_POPUP**, identifies the handle of the main menu. If *nFlags* contains neither **MF_SYSMENU** nor **MF_POPUP**, it is unused.

Remarks

If *nFlags* contains 0xFFFF and *hSysMenu* contains 0, Windows has closed the menu because the user pressed the ESC key or clicked outside the menu.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnMenuSelect

Concepts

CWnd Members

CWnd::OnMouseActivate

The framework calls this member function when the cursor is in an inactive window and the user presses a mouse button.

```
afx_msg int OnMouseActivate(
   CWnd* pDesktopWnd,
   UINT nHitTest,
   UINT message
);
```

Parameters

pDesktopWnd

Specifies a pointer to the top-level parent window of the window being activated. The pointer may be temporary and should not be stored.

nHitTest

Specifies the hit-test area code. A hit test is a test that determines the location of the cursor.

message

Specifies the mouse message number.

Return Value

Specifies whether to activate the **CWnd** and whether to discard the mouse event. It must be one of the following values:

- **MA_ACTIVATE** Activate **CWnd** object.
- **MA_NOACTIVATE** Do not activate **CWnd** object.
- **MA_ACTIVATEANDEAT** Activate **CWnd** object and discard the mouse event.
- **MA_NOACTIVATEANDEAT** Do not activate **CWnd** object and discard the mouse event.

Remarks

The default implementation passes this message to the parent window before any processing occurs. If the parent window returns **TRUE**, processing is halted.

For a description of the individual hit-test area codes, see the OnNcHitTest member function

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

Example

```
// The code fragment below shows how to UI activate an ActiveX control.
// CMyOLEControl is a COleControl-derived class.
int CMyOLEControl::OnMouseActivate(CWnd* pDesktopWnd, UINT nHitTest, UINT message)
{
   OnActivateInPlace(TRUE, NULL);  // OnActivateInPlace() is an undocumented function
   return COleControl::OnMouseActivate(pDesktopWnd, nHitTest, message);
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnNcHitTest

CWnd::OnMouseActivate

Concepts

CWnd Members

CWnd::OnMouseMove

The framework calls this member function when the mouse cursor moves.

```
afx_msg void OnMouseMove(
   UINT nFlags,
   CPoint point
);
```

Parameters

nFlags

Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if the CTRL key is down.
- **MK_LBUTTON** Set if the left mouse button is down.
- **MK_MBUTTON** Set if the middle mouse button is down.
- **MK_RBUTTON** Set if the right mouse button is down.
- **MK_SHIFT** Set if the SHIFT key is down.

point

Specifies the x- and y-coordinate of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks

If the mouse is not captured, the **WM_MOUSEMOVE** message is received by the **CWnd** object beneath the mouse cursor; otherwise, the message goes to the window that has captured the mouse.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::SetCapture
CWnd::OnNcHitTest
CWnd::OnMouseMove

Concepts

CWnd Members

CWnd::OnMouseWheel

The framework calls this member function as a user rotates the mouse wheel and encounters the wheel's next notch.

```
afx_msg BOOL OnMouseWheel(
   UINT nFlags,
   short zDelta,
   CPoint pt
);
```

Parameters

nFlags

Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if the CTRL key is down.
- **MK_LBUTTON** Set if the left mouse button is down.
- **MK_MBUTTON** Set if the middle mouse button is down.
- **MK_RBUTTON** Set if the right mouse button is down.
- **MK_SHIFT** Set if the SHIFT key is down.

zDelta

Indicates distance rotated. The *zDelta* value is expressed in multiples or divisions of **WHEEL_DELTA**, which is 120. A value less than zero indicates rotating back (toward the user) while a value greater than zero indicates rotating forward (away from the user). The user can reverse this response by changing the Wheel setting in the mouse software. See the Remarks for more information about this parameter.

pt

Specifies the x- and y-coordinate of the cursor. These coordinates are always relative to the upper-left corner of the screen.

Return Value

Nonzero if mouse wheel scrolling is enabled; otherwise 0.

Remarks

Unless overridden, **OnMouseWheel** calls the default of WM_MOUSEWHEEL. Windows automatically routes the message to the control or child window that has the focus. The Win32 function DefWindowProc propagates the message up the parent chain to the window that processes it.

The *zDelta* parameter is a multiple of **WHEEL_DELTA**, which is set at 120. This value is the threshold for an action to be taken, and one such action (for example, scrolling forward one notch) should occur for each delta.

The delta was set to 120 to allow for future finer-resolution wheels, such as a freely-rotating wheel with no notches. Such a device might send more messages per rotation, but with a smaller value in each message. To support this possibility, either aggregate the incoming delta values until **WHEEL_DELTA** is reached (so you get the same response for a given delta-rotation), or scroll partial lines in response to the more frequent messages. You could also choose your scroll granularity and accumulate deltas until **WHEEL_DELTA** is reached.

Override this member function to provide your own mouse-wheel scrolling behavior.

Note
OnMouseWheel handles messages for Windows NT 4.0. For Windows 95/98 or Windows NT 3.51 message handling, use OnRegisteredMouseWheel.

Smart Device Developer Notes

This method is not supported in smart device projects.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::OnMouseMove

Concepts

CWnd Members

CWnd::OnMove

The framework calls this member function after the **CWnd** object has been moved.

```
afx_msg void OnMove(
   int x,
   int y
);
```

Parameters

x

Specifies the new x-coordinate location of the upper-left corner of the client area. This new location is given in screen coordinates for overlapped and pop-up windows, and parent-client coordinates for child windows.

y

Specifies the new y-coordinate location of the upper-left corner of the client area. This new location is given in screen coordinates for overlapped and pop-up windows, and parent-client coordinates for child windows.

Remarks

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnMoving
CWnd::SetWindowPos

Concepts

CWnd Members

CWnd::OnMoving

The framework calls this member function while a user is moving a **CWnd** object.

```
afx_msg void OnMoving(
   UINT nSide,
   LPRECT lpRect
);
```

Parameters

nSide

The edge of window to be moved.

lpRect

Address of the CRect or RECT structure that will contain the item's coordinates.

Remarks

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::OnMove

Concepts

CWnd Members

CWnd::OnNcActivate

The framework calls this member function when the nonclient area needs to be changed to indicate an active or inactive state.

```
afx_msg BOOL OnNcActivate(
   BOOL bActive
);
```

Parameters

bActive

Specifies when a caption bar or icon needs to be changed to indicate an active or inactive state. The *bActive* parameter is **TRUE** if an active caption or icon is to be drawn. It is **FALSE** for an inactive caption or icon.

Return Value

Nonzero if Windows should proceed with default processing; 0 to prevent the caption bar or icon from being deactivated.

Remarks

The default implementation draws the title bar and title-bar text in their active colors if *bActive* is **TRUE** and in their inactive colors if *bActive* is **FALSE**.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::Default
CWnd::OnNcActivate

Concepts

CWnd Members

CWnd::OnNcCalcSize

The framework calls this member function when the size and position of the client area needs to be calculated.

```
afx_msg void OnNcCalcSize(
   BOOL bCalcValidRects,
   NCCALCSIZE_PARAMS* lpncsp
);
```

Parameters

bCalcValidRects

Specifies whether the application should specify which part of the client area contains valid information. Windows will copy the valid information to the specified area within the new client area. If this parameter is **TRUE**, the application should specify which part of the client area is valid.

lpncsp

Points to a NCCALCSIZE_PARAMS data structure that contains information an application can use to calculate the new size and position of the **CWnd** rectangle (including client area, borders, caption, scroll bars, and so on).

Remarks

By processing this message, an application can control the contents of the window's client area when the size or position of the window changes.

Regardless of the value of *bCalcValidRects*, the first rectangle in the array specified by the **rgrc** structure member of the **NCCALCSIZE_PARAMS** structure contains the coordinates of the window. For a child window, the coordinates are relative to the parent window's client area. For top-level windows, the coordinates are screen coordinates. An application should modify the **rgrc[0]** rectangle to reflect the size and position of the client area.

The **rgrc[1]** and **rgrc[2]** rectangles are valid only if *bCalcValidRects* is **TRUE**. In this case, the **rgrc[1]** rectangle contains the coordinates of the window before it was moved or resized. The **rgrc[2]** rectangle contains the coordinates of the window's client area before the window was moved. All coordinates are relative to the parent window or screen.

The default implementation calculates the size of the client area based on the window characteristics (presence of scroll bars, menu, and so on), and places the result in *lpncsp*.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

Smart Device Developer Notes

This method is not supported in smart device projects.

See Also

Reference

CWnd Class
Hierarchy Chart
WM_NCCALCSIZE
CWnd::MoveWindow
CWnd::SetWindowPos

Concepts

CWnd Members

CWnd::OnNcCreate

The framework calls this member function prior to the WM_CREATE message when the **CWnd** object is first created.

```
afx_msg BOOL OnNcCreate(
   LPCREATESTRUCT lpCreateStruct
);
```

Parameters

lpCreateStruct

Points to the CREATESTRUCT data structure for **CWnd**.

Return Value

Nonzero if the nonclient area is created. It is 0 if an error occurs; the **Create** function will return **failure** in this case.

Remarks

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

Smart Device Developer Notes

This method is not supported in smart device projects.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::Create
CWnd::CreateEx
CWnd::OnNcCreate

Concepts

CWnd Members

CWnd::OnNcDestroy

Called by the framework when the nonclient area is being destroyed, and is the last member function called when the Windows window is destroyed.

```
afx_msg void OnNcDestroy( );
```

Remarks

The default implementation performs some cleanup, then calls the virtual member function PostNcDestroy.

Override **PostNcDestroy** if you want to perform your own cleanup, such as a **delete this** operation. If you override **OnNcDestroy**, you must call **OnNcDestroy** in your base class to ensure that any memory internally allocated for the window is freed.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::DestroyWindow
CWnd::OnNcCreate
WM_NCDESTROY
CWnd::Default
CWnd::PostNcDestroy

Concepts

CWnd Members

CWnd::OnNcHitTest

The framework calls this member function for the **CWnd** object that contains the cursor (or the **CWnd** object that used the SetCapture member function to capture the mouse input) every time the mouse is moved.

```
afx_msg LRESULT OnNcHitTest(
   CPoint point
);
```

Parameters

point

Contains the x- and y-coordinates of the cursor. These coordinates are always screen coordinates.

Return Value

One of the mouse hit-test enumerated values listed below.

Remarks

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

Mouse Enumerated Values

- **HTBORDER** In the border of a window that does not have a sizing border.
- **HTBOTTOM** In the lower horizontal border of the window.
- **HTBOTTOMLEFT** In the lower-left corner of the window border.
- **HTBOTTOMRIGHT** In the lower-right corner of the window border.
- **HTCAPTION** In a title-bar area.
- **HTCLIENT** In a client area.
- **HTERROR** On the screen background or on a dividing line between windows (same as **HTNOWHERE** except that the **DefWndProc** Windows function produces a system beep to indicate an error).
- **HTGROWBOX** In a size box.
- **HTHSCROLL** In the horizontal scroll bar.
- **HTLEFT** In the left border of the window.
- **HTMAXBUTTON** In a Maximize button.
- **HTMENU** In a menu area.
- **HTMINBUTTON** In a Minimize button.
- **HTNOWHERE** On the screen background or on a dividing line between windows.
- **HTREDUCE** In a Minimize button.
- **HTRIGHT** In the right border of the window.
- **HTSIZE** In a size box (same as **HTGROWBOX**).
- **HTSYSMENU** In a Control menu or in a Close button in a child window.
- **HTTOP** In the upper horizontal border of the window.
- **HTTOPLEFT** In the upper-left corner of the window border.

- **HTTOPRIGHT** In the upper-right corner of the window border.
- **HTTRANSPARENT** In a window currently covered by another window.
- **HTVSCROLL** In the vertical scroll bar.
- **HTZOOM** In a Maximize button.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetCapture
WM_NCHITTEST

Concepts

CWnd Members

CWnd::OnNcLButtonDblClk

The framework calls this member function when the user double-clicks the left mouse button while the cursor is within a nonclient area of **CWnd**.

```
afx_msg void OnNcLButtonDblClk(
   UINT nHitTest,
   CPoint point
);
```

Parameters

nHitTest

Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point

Specifies a **CPoint** object that contains the x and y screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks

If appropriate, the WM_SYSCOMMAND message is sent.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
WM_NCLBUTTONDBLCLK
CWnd::OnNcHitTest

Concepts

CWnd Members

CWnd::OnNcLButtonDown

The framework calls this member function when the user presses the left mouse button while the cursor is within a nonclient area of the **CWnd** object.

```
afx_msg void OnNcLButtonDown(
   UINT nHitTest,
   CPoint point
);
```

Parameters

nHitTest

Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point

Specifies a **CPoint** object that contains the x and y screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks

If appropriate, the WM_SYSCOMMAND is sent.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received.If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnNcHitTest
CWnd::OnNcLButtonDblClk
CWnd::OnNcLButtonUp
CWnd::OnSysCommand
WM_NCLBUTTONDOWN
CWnd::Default

Concepts

CWnd Members

CWnd::OnNcLButtonUp

The framework calls this member function when the user releases the left mouse button while the cursor is within a nonclient area.

```
afx_msg void OnNcLButtonUp(
   UINT nHitTest,
   CPoint point
);
```

Parameters

nHitTest

Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point

Specifies a **CPoint** object that contains the x and y screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks

If appropriate, WM_SYSCOMMAND is sent.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnNcHitTest
CWnd::OnNcLButtonDown
CWnd::OnSysCommand
CWnd::OnNcLButtonUp

Concepts

CWnd Members

CWnd::OnNcMButtonDblClk

The framework calls this member function when the user double-clicks the middle mouse button while the cursor is within a nonclient area.

```
afx_msg void OnNcMButtonDblClk(
   UINT nHitTest,
   CPoint point
);
```

Parameters

nHitTest

Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point

Specifies a **CPoint** object that contains the x and y screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnNcHitTest
CWnd::OnNcMButtonDown
CWnd::OnNcMButtonUp
CWnd::OnNcMButtonDblClk

Concepts

CWnd Members

CWnd::OnNcMButtonDown

The framework calls this member function when the user presses the middle mouse button while the cursor is within a nonclient area.

```
afx_msg void OnNcMButtonDown(
   UINT nHitTest,
   CPoint point
);
```

Parameters

nHitTest

Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point

Specifies a **CPoint** object that contains the x and y screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnNcHitTest
CWnd::OnNcMButtonDblClk
CWnd::OnNcMButtonUp
CWnd::OnNcMButtonDown

Concepts

CWnd Members

CWnd::OnNcMButtonUp

The framework calls this member function when the user releases the middle mouse button while the cursor is within a nonclient area.

```
afx_msg void OnNcMButtonUp(
   UINT nHitTest,
   CPoint point
);
```

Parameters

nHitTest

Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point

Specifies a **CPoint** object that contains the x and y screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnNcHitTest
CWnd::OnNcMButtonDblClk
CWnd::OnNcMButtonDown
CWnd::OnNcMButtonUp

Concepts

CWnd Members

CWnd::OnNcMouseMove

The framework calls this member function when the cursor is moved within a nonclient area.

```
afx_msg void OnNcMouseMove(
   UINT nHitTest,
   CPoint point
);
```

Parameters

nHitTest

Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point

Specifies a **CPoint** object that contains the x and y screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks

If appropriate, the WM_SYSCOMMAND message is sent.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnNcHitTest
CWnd::OnSysCommand
CWnd::OnNcMouseMove

Concepts

CWnd Members

CWnd::OnNcPaint

The framework calls this member function when the nonclient area needs to be painted.

```
afx_msg void OnNcPaint( );
```

Remarks

The default implementation paints the window frame.

An application can override this call and paint its own custom window frame. The clipping region is always rectangular, even if the shape of the frame is altered.

Smart Device Developer Notes

This method is not supported in smart device projects.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::OnNcPaint

Concepts

CWnd Members

CWnd::OnNcRButtonDblClk

The framework calls this member function when the user double-clicks the right mouse button while the cursor is within a nonclient area of **CWnd**.

```
afx_msg void OnNcRButtonDblClk(
   UINT nHitTest,
   CPoint point
);
```

Parameters

nHitTest

Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point

Specifies a **CPoint** object that contains the x and y screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnNcHitTest
CWnd::OnNcRButtonDown
CWnd::OnNcRButtonUp
CWnd::OnNcRButtonDblClk

Concepts

CWnd Members

CWnd::OnNcRButtonDown

The framework calls this member function when the user presses the right mouse button while the cursor is within a nonclient area.

```
afx_msg void OnNcRButtonDown(
   UINT nHitTest,
   CPoint point
);
```

Parameters

nHitTest

Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point

Specifies a **CPoint** object that contains the x and y screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnNcHitTest
CWnd::OnNcRButtonDblClk
CWnd::OnNcRButtonUp

Concepts

CWnd Members

CWnd::OnNcRButtonUp

The framework calls this member function when the user releases the right mouse button while the cursor is within a nonclient area.

```
afx_msg void OnNcRButtonUp(
   UINT nHitTest,
   CPoint point
);
```

Parameters

nHitTest

Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point

Specifies a **CPoint** object that contains the x and y screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnNcHitTest
CWnd::OnNcRButtonDblClk
CWnd::OnNcRButtonDown
CWnd::OnNcRButtonUp

Concepts

CWnd Members

CWnd::OnNotify

The framework calls this member function to inform the parent window of a control that an event has occurred in the control or that the control requires some kind of information.

```
virtual BOOL OnNotify(
   WPARAM wParam,
   LPARAM lParam,
   LRESULT* pResult
);
```

Parameters

wParam

Identifies the control that sends the message if the message is from a control. Otherwise, *wParam* is 0.

lParam

Pointer to a notification message (**NMHDR**) structure that contains the notification code and additional information. For some notification messages, this parameter points to a larger structure that has the **NMHDR** structure as its first member.

pResult

Pointer to an **LRESULT** variable in which to store the result code if the message is handled.

Return Value

An application returns nonzero if it processes this message; otherwise 0.

Remarks

OnNotify processes the message map for control notification.

Override this member function in your derived class to handle the **WM_NOTIFY** message. An override will not process the message map unless the base class **OnNotify** is called.

For more information on the WM_NOTIFY message, see Technical Note 61 (TN061), ON_NOTIFY and WM_NOTIFY messages. You may also be interested the related topics described in Control Topics, and TN062, Message Reflection for Windows Controls.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::OnPaint

The framework calls this member function when Windows or an application makes a request to repaint a portion of an application's window.

```
afx_msg void OnPaint( );
```

Remarks

The WM_PAINT message is sent when the UpdateWindow or RedrawWindow member function is called.

A window may receive internal paint messages as a result of calling the **RedrawWindow** member function with the **RDW_INTERNALPAINT** flag set. In this case, the window may not have an update region. An application should call the GetUpdateRect member function to determine whether the window has an update region. If **GetUpdateRect** returns 0, the application should not call the BeginPaint and EndPaint member functions.

It is an application's responsibility to check for any necessary internal repainting or updating by looking at its internal data structures for each **WM_PAINT** message because a **WM_PAINT** message may have been caused by both an invalid area and a call to the **RedrawWindow** member function with the **RDW_INTERNALPAINT** flag set.

An internal **WM_PAINT** message is sent only once by Windows. After an internal **WM_PAINT** message is sent to a window by the **UpdateWindow** member function, no further **WM_PAINT** messages will be sent or posted until the window is invalidated or until the **RedrawWindow** member function is called again with the **RDW_INTERNALPAINT** flag set.

For information on rendering an image in document/view applications, see CView::OnDraw.

For more information about using **WM_Paint**, see the following topics in the *Platform SDK*:

- The WM_PAINT Message
- Using the WM_PAINT Message

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::BeginPaint
CWnd::EndPaint
CWnd::RedrawWindow
CPaintDC Class
CView::OnDraw

Concepts

CWnd Members

CWnd::OnPaintClipboard

A Clipboard owner's **OnPaintClipboard** member function is called by a Clipboard viewer when the Clipboard owner has placed data on the Clipboard in the **CF_OWNERDISPLAY** format and the Clipboard viewer's client area needs repainting.

```
afx_msg void OnPaintClipboard(
   CWnd* pClipAppWnd,
   HGLOBAL hPaintStruct
);
```

Parameters

pClipAppWnd

Specifies a pointer to the Clipboard-application window. The pointer may be temporary and should not be stored for later use.

hPaintStruct

Identifies a PAINTSTRUCT data structure that defines what part of the client area to paint.

Remarks

To determine whether the entire client area or just a portion of it needs repainting, the Clipboard owner must compare the dimensions of the drawing area given in the **rcpaint** member of the **PAINTSTRUCT** structure to the dimensions given in the most recent OnSizeClipboard member function call.

OnPaintClipboard should use the GlobalLock Windows function to lock the memory that contains the **PAINTSTRUCT** data structure and unlock that memory with the GlobalUnlock Windows function before it exits.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
GlobalLock
GlobalUnlock
CWnd::OnSizeClipboard
CWnd::OnPaintClipboard

Concepts

CWnd Members

CWnd::OnPaletteChanged

The framework calls this member function for all top-level windows after the window with input focus has realized its logical palette, thereby changing the system palette.

```
afx_msg void OnPaletteChanged(
   CWnd* pFocusWnd
);
```

Parameters

pFocusWnd

Specifies a pointer to the window that caused the system palette to change. The pointer may be temporary and should not be stored.

Remarks

This call allows a window without the input focus that uses a color palette to realize its logical palettes and update its client area.

The **OnPaletteChanged** member function is called for all top-level and overlapped windows, including the one that changed the system palette and caused the **WM_PALETTECHANGED** message to be sent. If any child window uses a color palette, this message must be passed on to it.

To avoid an infinite loop, the window shouldn't realize its palette unless it determines that *pFocusWnd* does not contain a pointer to itself.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
RealizePalette
WM_PALETTECHANGED
CWnd::OnPaletteIsChanging
CWnd::OnQueryNewPalette

Concepts

CWnd Members

CWnd::OnPaletteIsChanging

The framework calls this member function to inform applications that an application is going to realize its logical palette.

```
afx_msg void OnPaletteIsChanging(
   CWnd* pRealizeWnd
);
```

Parameters

pRealizeWnd

Specifies the window that is about to realize its logical palette.

Remarks

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnPaletteChanged
CWnd::OnQueryNewPalette
WM_PALETTEISCHANGING

Concepts

CWnd Members

CWnd::OnParentNotify

A parent's **OnParentNotify** member function is called by the framework when its child window is created or destroyed, or when the user clicks a mouse button while the cursor is over the child window.

```
afx_msg void OnParentNotify(
   UINT message,
   LPARAM lParam
);
```

Parameters

message

Specifies the event for which the parent is being notified and the identifier of the child window. The event is the low-order word of *message*. If the event is **WM_CREATE** or **WM_DESTROY**, the high-order word of *message* is the identifier of the child window; otherwise, the high-order word is undefined. The event (low-order word of *message*) can be any of these values:

- **WM_CREATE** The child window is being created.
- **WM_DESTROY** The child window is being destroyed.
- **WM_LBUTTONDOWN** The user has placed the mouse cursor over the child window and clicked the left mouse button.
- **WM_MBUTTONDOWN** The user has placed the mouse cursor over the child window and clicked the middle mouse button.
- **WM_RBUTTONDOWN** The user has placed the mouse cursor over the child window and clicked the right mouse button.

lParam

If the event (low-order word) of *message* is **WM_CREATE** or **WM_DESTROY**, *lParam* specifies the window handle of the child window; otherwise *lParam* contains the x and y coordinates of the cursor. The x coordinate is in the low-order word and the y coordinate is in the high-order word.

Remarks

When the child window is being created, the system calls **OnParentNotify** just before the Create member function that creates the window returns. When the child window is being destroyed, the system calls **OnParentNotify** before any processing takes place to destroy the window.

OnParentNotify is called for all ancestor windows of the child window, including the top-level window.

All child windows except those that have the WS_EX_NOPARENTNOTIFY style send this message to their parent windows. By default, child windows in a dialog box have the **WS_EX_NOPARENTNOTIFY** style unless the child window was created without this style by calling the CreateEx member function.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnCreate
CWnd::OnDestroy
CWnd::OnLButtonDown

CWnd::OnMButtonDown
CWnd::OnRButtonDown
WM_PARENTNOTIFY

Concepts

CWnd Members

CWnd::OnQueryDragIcon

The framework calls this member function by a minimized (iconic) window that does not have an icon defined for its class.

```
afx_msg HCURSOR OnQueryDragIcon( );
```

Return Value

A doubleword value that contains a cursor or icon handle in the low-order word. The cursor or icon must be compatible with the display driver's resolution. If the application returns **NULL**, the system displays the default cursor. The default return value is **NULL**.

Remarks

The system makes this call to obtain the cursor to display while the user drags the minimized window. If an application returns the handle of an icon or cursor, the system converts it to black-and-white. If an application returns a handle, the handle must identify a monochrome cursor or icon compatible with the display driver's resolution. The application can call the CWinApp::LoadCursor or CWinApp::LoadIcon member functions to load a cursor or icon from the resources in its executable file and to obtain this handle.

See Also

Reference

CWnd Class
Hierarchy Chart
CWinApp::LoadCursor
CWinApp::LoadIcon
WM_QUERYDRAGICON

Concepts

CWnd Members

CWnd::OnQueryEndSession

The framework calls this member function when the user chooses to end the Windows session or when an application calls the ExitWindows Windows function.

```
afx_msg BOOL OnQueryEndSession( );
```

Return Value

Nonzero if an application can be conveniently shut down; otherwise 0.

Remarks

If any application returns 0, the Windows session is not ended. Windows stops calling **OnQueryEndSession** as soon as one application returns 0 and sends the WM_ENDSESSION message with a parameter value of **FALSE** for any application that has already returned nonzero.

See Also

Reference

CWnd Class
Hierarchy Chart
ExitWindows
CWnd::OnEndSession
WM_QUERYENDSESSION

Concepts

CWnd Members

CWnd::OnQueryNewPalette

The framework calls this member function when the **CWnd** object is about to receive the input focus, giving the **CWnd** an opportunity to realize its logical palette when it receives the focus.

```
afx_msg BOOL OnQueryNewPalette( );
```

Return Value

Nonzero if the **CWnd** realizes its logical palette; otherwise 0.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::Default
CWnd::OnPaletteChanged
WM_QUERYNEWPALETTE

Concepts

CWnd Members

CWnd::OnQueryOpen

The framework calls this member function when the **CWnd** object is minimized and the user requests that the **CWnd** be restored to its preminimized size and position.

```
afx_msg BOOL OnQueryOpen( );
```

Return Value

Nonzero if the icon can be opened, or 0 to prevent the icon from being opened.

Remarks

While in **OnQueryOpen**, **CWnd** should not perform any action that would cause an activation or focus change (for example, creating a dialog box).

See Also

Reference

CWnd Class
Hierarchy Chart
WM_QUERYOPEN

Concepts

CWnd Members

CWnd::OnQueryUIState

Called to retrieve the user interface (UI) state for a window.

```
afx_msg UINT OnQueryUIState( );
```

Return Value

The return value is **NULL** if the focus indicators and the keyboard accelerators are visible. Otherwise, the return value can be one or more of the following values:

- **UISF_HIDEFOCUS** Focus indicators are hidden.
- **UISF_HIDEACCEL** Keyboard accelerators are hidden.
- **UISF_ACTIVE** **Windows XP:** A control should be drawn in the style used for active controls.

Remarks

This member function emulates the functionality of the WM_QUERYUISTATE message, as described in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnChangeUIState
CWnd::OnUpdateUIState

Concepts

CWnd Members

CWnd::OnRButtonDblClk

The framework calls this member function when the user double-clicks the right mouse button.

```
afx_msg void OnRButtonDblClk(
   UINT nFlags,
   CPoint point
);
```

Parameters

nFlags

Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if CTRL key is down.
- **MK_LBUTTON** Set if left mouse button is down.
- **MK_MBUTTON** Set if middle mouse button is down.
- **MK_RBUTTON** Set if right mouse button is down.
- **MK_SHIFT** Set if SHIFT key is down.

point

Specifies the x and y coordinates of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks

Only windows that have the **CS_DBLCLKS** WNDCLASS style can receive **OnRButtonDblClk** calls. This is the default for windows within the Microsoft Foundation Class Library. Windows calls **OnRButtonDblClk** when the user presses, releases, and then again presses the right mouse button within the system's double-click time limit. Double-clicking the right mouse button actually generates four events: WM_RBUTTONDOWN and WM_RBUTTONUP messages, the **OnRButtonDblClk** call, and another **WM_RBUTTONUP** message when the button is released.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnRButtonDown
CWnd::OnRButtonUp
WM_RBUTTONDBLCLK

Concepts

CWnd Members

CWnd::OnRButtonDown

The framework calls this member function when the user presses the right mouse button.

```
afx_msg void OnRButtonDown(
   UINT nFlags,
   CPoint point
);
```

Parameters

nFlags

Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if CTRL key is down.
- **MK_LBUTTON** Set if left mouse button is down.
- **MK_MBUTTON** Set if middle mouse button is down.
- **MK_RBUTTON** Set if right mouse button is down.
- **MK_SHIFT** Set if SHIFT key is down.

point

Specifies the x and y coordinates of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnRButtonDblClk
CWnd::OnRButtonUp
WM_RBUTTONDOWN

Concepts

CWnd Members

CWnd::OnRButtonUp

The framework calls this member function when the user releases the right mouse button.

```
afx_msg void OnRButtonUp(
   UINT nFlags,
   CPoint point
);
```

Parameters

nFlags

Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if CTRL key is down.
- **MK_LBUTTON** Set if left mouse button is down.
- **MK_MBUTTON** Set if middle mouse button is down.
- **MK_SHIFT** Set if SHIFT key is down.

point

Specifies the x and y coordinates of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnRButtonDblClk
CWnd::OnRButtonDown
WM_RBUTTONUP

Concepts

CWnd Members

CWnd::OnRegisteredMouseWheel

The framework calls this member function as a user rotates the mouse wheel and encounters the wheel's next notch.

```
afx_msg LRESULT OnRegisteredMouseWheel(
   WPARAM wParam,
   LPARAM lParam
);
```

Parameters

wParam

Horizontal position of the pointer.

lParam

Vertical position of the pointer.

Return Value

Insignificant at this time. Always zero.

Remarks

Unless overridden, **OnRegisteredMouseWheel** routes the message to the appropriate window (the parent window with focus), and calls the WM_MOUSEWHEEL handler for that window.

Override this member function to provide your own message routing or to alter the mouse-wheel scrolling behavior.

Note
OnRegisteredMouseWheel handles messages for Windows 95/98 and Windows NT 3.51. For Windows NT 4.0 message h andling, use OnMouseWheel.

See Also

Reference

CWnd Class

Hierarchy Chart

RegisterWindowMessage

Concepts

CWnd Members

CWnd::OnRenderAllFormats

The Clipboard owner's **OnRenderAllFormats** member function is called by the framework when the owner application is being destroyed.

```
afx_msg void OnRenderAllFormats( );
```

Remarks

The Clipboard owner should render the data in all the formats it is capable of generating and pass a data handle for each format to the Clipboard by calling the SetClipboardData Windows function. This ensures that the Clipboard contains valid data even though the application that rendered the data is destroyed. The application should call the OpenClipboard member function before calling the SetClipboardData Windows function and call the CloseClipboard Windows function afterward.

See Also

Reference

CWnd Class
Hierarchy Chart
CloseClipboard
CWnd::OpenClipboard
SetClipboardData
CWnd::OnRenderFormat
WM_RENDERALLFORMATS

Concepts

CWnd Members

CWnd::OnRenderFormat

The Clipboard owner's **OnRenderFormat** member function is called by the framework when a particular format with delayed rendering needs to be rendered.

```
afx_msg void OnRenderFormat(
   UINT nFormat
);
```

Parameters

nFormat

Specifies the Clipboard format.

Remarks

The receiver should render the data in that format and pass it to the Clipboard by calling the SetClipboardData Windows function.

Do not call the **OpenClipboard** member function or the **CloseClipboard** Windows function from within **OnRenderFormat**.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CloseClipboard
CWnd::OpenClipboard
SetClipboardData
WM_RENDERFORMAT

Concepts

CWnd Members

CWnd::OnSetCursor

The framework calls this member function if mouse input is not captured and the mouse causes cursor movement within the **CWnd** object.

```
afx_msg BOOL OnSetCursor(
   CWnd* pWnd,
   UINT nHitTest,
   UINT message
);
```

Parameters

pWnd

Specifies a pointer to the window that contains the cursor. The pointer may be temporary and should not be stored for later use.

nHitTest

Specifies the hit-test area code. The hit test determines the cursor's location.

message

Specifies the mouse message number.

Return Value

Nonzero to halt further processing, or 0 to continue.

Remarks

The default implementation calls the parent window's **OnSetCursor** before processing. If the parent window returns **TRUE**, further processing is halted. Calling the parent window gives the parent window control over the cursor's setting in a child window.

The default implementation sets the cursor to an arrow if it is not in the client area or to the registered-class cursor if it is.

If *nHitTest* is **HTERROR** and *message* is a mouse button-down message, the **MessageBeep** member function is called.

The *message* parameter is 0 when **CWnd** enters menu mode.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnNcHitTest
WM_SETCURSOR

Concepts

CWnd Members

CWnd::OnSetFocus

The framework calls this member function after gaining the input focus.

```
afx_msg void OnSetFocus(
   CWnd* pOldWnd
);
```

Parameters

pOldWnd

Contains the **CWnd** object that loses the input focus (may be **NULL**). The pointer may be temporary and should not be stored for later use.

Remarks

To display a caret, **CWnd** should call the appropriate caret functions at this point.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
WM_SETFOCUS

Concepts

CWnd Members

CWnd::OnSettingChange

The framework calls **OnSettingChange** for all top-level windows when the Win32 SystemParametersInfo function changes a system-wide setting.

```
afx_msg void OnSettingChange(
   UINT uFlags,
   LPCTSTR lpszSection
);
```

Parameters

uFlags

When the system sends the message as a result of a **SystemParametersInfo** call, this parameter is a flag that indicates the system parameter that was changed. For a list of values, see SystemParametersInfo in the *Platform SDK*. When an application sends the message, this parameter must be 0.

lpszSection

Points to a string that specifies the name of the section that has changed. (The string does not include the square brackets that enclose the section name.)

Remarks

An application should send the message to all top-level windows when it makes changes to system parameters, and Windows will send the message if the user changes settings via the Control Panel.

The **ON_WM_SETTINGCHANGE** message is similar to the **ON_WM_WININICHANGE** message, with the following difference:

- Use **ON_WM_SETTINGCHANGE** when running Windows NT 4.0 or newer, or under Windows 95/98.
- Use **ON_WININICHANGE** when running Windows NT 3.51 or older. This message is now obsolete.

You should have only one of these macros in your message map. To write a program that works for both Windows 95/98 and Windows NT 4.0, write a handler for **ON_WM_SETTINGCHANGE**. Under Windows NT 3.51, your handler will be called by **OnSettingChange** and *uFlags* and will always be zero.

See Also

Reference

WM_SETTINGCHANGE
CWnd::OnWinIniChange

CWnd::OnShowWindow

The framework calls this member function when the **CWnd** object is about to be hidden or shown.

```
afx_msg void OnShowWindow(
   BOOL bShow,
   UINT nStatus
);
```

Parameters

bShow

Specifies whether a window is being shown. It is **TRUE** if the window is being shown; it is **FALSE** if the window is being hidden.

nStatus

Specifies the status of the window being shown. It is 0 if the message is sent because of a **ShowWindow** member function call; otherwise *nStatus* is one of the following:

- **SW_PARENTCLOSING** Parent window is closing (being made iconic) or a pop-up window is being hidden.
- **SW_PARENTOPENING** Parent window is opening (being displayed) or a pop-up window is being shown.

Remarks

A window is hidden or shown when the **ShowWindow** member function is called, when an overlapped window is maximized or restored, or when an overlapped or pop-up window is closed (made iconic) or opened (displayed on the screen). When an overlapped window is closed, all pop-up windows associated with that window are hidden.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class

Hierarchy Chart

WM_SHOWWINDOW

Concepts

CWnd Members

CWnd::OnSize

The framework calls this member function after the window's size has changed.

```
afx_msg void OnSize(
   UINT nType,
   int cx,
   int cy
);
```

Parameters

nType

Specifies the type of resizing requested. This parameter can be one of the following values:

- **SIZE_MAXIMIZED** Window has been maximized.
- **SIZE_MINIMIZED** Window has been minimized.
- **SIZE_RESTORED** Window has been resized, but neither **SIZE_MINIMIZED** nor **SIZE_MAXIMIZED** applies.
- **SIZE_MAXHIDE** Message is sent to all pop-up windows when some other window is maximized.
- **SIZE_MAXSHOW** Message is sent to all pop-up windows when some other window has been restored to its former size.

cx

Specifies the new width of the client area.

cy

Specifies the new height of the client area.

Remarks

If the SetScrollPos or MoveWindow member function is called for a child window from **OnSize**, the *bRedraw* parameter of **SetScrollPos** or **MoveWindow** should be nonzero to cause the **CWnd** to be repainted.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

Example

```
// Resize the list control contained in the view to
// fill the entire view when the view's window is
// resized. CMyView is a CView derived class.
void CMyView::OnSize(UINT nType, int cx, int cy)
{
   CView::OnSize(nType, cx, cy);
   // Resize list to fill the whole view.
   m_List.MoveWindow (0, 0, cx, cy);
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::MoveWindow
CWnd::SetScrollPos

WM_SIZE

Concepts

CWnd Members

CWnd::OnSizeClipboard

The Clipboard owner's **OnSizeClipboard** member function is called by the Clipboard viewer when the Clipboard contains data with the **CF_OWNERDISPLAY** attribute and the size of the client area of the Clipboard-viewer window has changed.

```
afx_msg void OnSizeClipboard(
   CWnd* pClipAppWnd,
   HGLOBAL hRect
);
```

Parameters

pClipAppWnd

Identifies the Clipboard-application window. The pointer may be temporary and should not be stored.

hRect

Identifies a global memory object. The memory object contains a RECT data structure that specifies the area for the Clipboard owner to paint.

Remarks

The **OnSizeClipboard** member function is called with a null rectangle (0,0,0,0) as the new size when the Clipboard application is about to be destroyed or minimized. This permits the Clipboard owner to free its display resources.

Within **OnSizeClipboard**, an application must use the GlobalLock Windows function to lock the memory that contains the RECT data structure. Have the application unlock that memory with the GlobalUnlock Windows function before it yields or returns control.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
GlobalLock
GlobalUnlock
SetClipboardData
CWnd::SetClipboardViewer
WM_SIZECLIPBOARD

Concepts

CWnd Members

CWnd::OnSizing

The framework calls this member function to indicate that the user is resizing the rectangle.

```
afx_msg void OnSizing(
   UINT nSide,
   LPRECT lpRect
);
```

Parameters

nSide

The edge of window to be moved.

lpRect

Address of the CRect or RECT structure that will contain the item's coordinates.

Remarks

By processing this message, an application can monitor the size and position of the drag rectangle and, if needed, change its size or position.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

Example

```
void CMainFrame::OnSizing(UINT fwSide, LPRECT pRect)
{
   CFrameWnd::OnSizing(fwSide, pRect);

   // Resize the splitter window in the frame. m_SplitWnd is of
   // type CSplitterWnd
   int nWidth=(pRect->right)-(pRect->left);
   m_SplitWnd.SetColumnInfo(0,nWidth/2,10);
   m_SplitWnd.SetColumnInfo(1,nWidth/2,10);
   m_SplitWnd.RecalcLayout();
}
```

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::OnSpoolerStatus

The framework calls this member function from Print Manager whenever a job is added to or removed from the Print Manager queue.

```
afx_msg void OnSpoolerStatus(
   UINT nStatus,
   UINT nJobs
);
```

Parameters

nStatus

Specifies the **SP_JOBSTATUS** flag.

nJobs

Specifies the number of jobs remaining in the Print Manager queue.

Remarks

This call is for informational purposes only.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
WM_SPOOLERSTATUS

Concepts

CWnd Members

CWnd::OnStyleChanged

The framework calls this member function after the SetWindowLong function has changed one or more of the window's styles.

```
afx_msg void OnStyleChanged(
   int nStyleType,
   LPSTYLESTRUCT lpStyleStruct
);
```

Parameters

nStyleType

Specifies whether the window's extended or nonextended styles have changed. This parameter can be a combination of the following values:

- **GWL_EXSTYLE** The window's extended styles have changed.
- **GWL_STYLE** The window's nonextended styles have changed.

lpStyleStruct

Points to a STYLESTRUCT structure that contains the new styles for the window. An application can examine the styles, but it can not change them.

Remarks

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
WM_STYLECHANGED

Concepts

CWnd Members

CWnd::OnStyleChanging

The framework calls this member function when the SetWindowLong function is about to change one or more of the window's styles.

```
afx_msg void OnStyleChanging(
   int nStyleType,
   LPSTYLESTRUCT lpStyleStruct
);
```

Parameters

nStyleType

Specifies whether the window's extended or nonextended styles have changed. This parameter can be a combination of the following values:

- **GWL_EXSTYLE** The window's extended styles have changed.
- **GWL_STYLE** The window's nonextended styles have changed.

lpStyleStruct

Points to a STYLESTRUCT structure that contains the new styles for the window. An application can examine the styles and change them.

Remarks

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::OnSysChar

The framework calls this member function if **CWnd** has the input focus and the WM_SYSKEYUP and WM_SYSKEYDOWN messages are translated.

```
afx_msg void OnSysChar(
   UINT nChar,
   UINT nRepCnt,
   UINT nFlags
);
```

Parameters

nChar

Specifies the ASCII-character key code of a Control-menu key.

nRepCnt

Specifies the repeat count (the number of times the keystroke is repeated as a result of the user holding down the key).

nFlags

The *nFlags* parameter can have these values:

Value	Meaning
0-15	Specifies the repeat count. The value is the number of times the keystroke is repeated as a result of the user holding down the key..
16-23	Specifies the scan code. The value depends on the original equipment manufacturer (OEM)
24	Specifies whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101- or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0.
25-28	Used internally by Windows.
29	Specifies the context code. The value is 1 if the ALT key is held down while the key is pressed; otherwise, the value is 0.
30	Specifies the previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up.
31	Specifies the transition state. The value is 1 if the key is being released, or it is 0 if the key is being pressed.

Remarks

It specifies the virtual key code of the Control-menu key. (For a list of of standard virtual key codes, see Winuser.h)

When the context code is 0, **WM_SYSCHAR** can pass the WM_SYSCHAR message to the TranslateAccelerator Windows function, which will handle it as though it were a normal key message instead of a system character-key. This allows accelerator keys to be used with the active window even if the active window does not have the input focus.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the slash (/) and ENTER keys in the numeric keypad. Some other keyboards may support the extended-key bit in *nFlags*.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
TranslateAccelerator
CWnd::OnSysKeyDown
CWnd::OnSysKeyUp
WM_SYSCHAR

Concepts

CWnd Members

CWnd::OnSysColorChange

The framework calls this member function for all top-level windows when a change is made in the system color setting.

```
afx_msg void OnSysColorChange( );
```

Remarks

Windows calls **OnSysColorChange** for any window that is affected by a system color change.

Applications that have brushes that use the existing system colors should delete those brushes and re-create them with the new system colors.

See Also

Reference

CWnd Class
Hierarchy Chart
SetSysColors
WM_SYSCOLORCHANGE

Concepts

CWnd Members

CWnd::OnSysCommand

The framework calls this member function when the user selects a command from the Control menu, or when the user selects the Maximize or the Minimize button.

```
afx_msg void OnSysCommand(
   UINT nID,
   LPARAM lParam
);
```

Parameters

nID

Specifies the type of system command requested. This parameter can be any one of the following values:

- **SC_CLOSE** Close the **CWnd** object.
- **SC_HOTKEY** Activate the **CWnd** object associated with the application-specified hot key. The low-order word of *lParam* identifies the **HWND** of the window to activate.
- **SC_HSCROLL** Scroll horizontally.
- **SC_KEYMENU** Retrieve a menu through a keystroke.
- **SC_MAXIMIZE** (or **SC_ZOOM**) Maximize the **CWnd** object.
- **SC_MINIMIZE** (or **SC_ICON**) Minimize the **CWnd** object.
- **SC_MOUSEMENU** Retrieve a menu through a mouse click.
- **SC_MOVE** Move the **CWnd** object.
- **SC_NEXTWINDOW** Move to the next window.
- **SC_PREVWINDOW** Move to the previous window.
- **SC_RESTORE** Restore window to normal position and size.
- **SC_SCREENSAVE** Executes the screen-saver application specified in the [boot] section of the SYSTEM.INI file.
- **SC_SIZE** Size the **CWnd** object.
- **SC_TASKLIST** Execute or activate the Windows Task Manager application.
- **SC_VSCROLL** Scroll vertically.

lParam

If a Control-menu command is chosen with the mouse, *lParam* contains the cursor coordinates. The low-order word contains the x coordinate, and the high-order word contains the y coordinate. Otherwise this parameter is not used.

- **SC_HOTKEY** Activate the window associated with the application-specified hot key. The low-order word of *lParam* identifies the window to activate.
- **SC_SCREENSAVE** Execute the screen-save application specified in the Desktop section of Control Panel.

Remarks

By default, **OnSysCommand** carries out the Control-menu request for the predefined actions specified in the preceding table.

In **WM_SYSCOMMAND** messages, the four low-order bits of the *nID* parameter are used internally by Windows. When an application tests the value of *nID*, it must combine the value 0xFFF0 with the *nID* value by using the bitwise-AND operator to obtain the correct result.

The menu items in a Control menu can be modified with the **GetSystemMenu**, **AppendMenu**, **InsertMenu**, and **ModifyMenu** member functions. Applications that modify the Control menu must process **WM_SYSCOMMAND** messages, and any **WM_SYSCOMMAND** messages not handled by the application must be passed on to **OnSysCommand**. Any

command values added by an application must be processed by the application and cannot be passed to **OnSysCommand**.

An application can carry out any system command at any time by passing a **WM_SYSCOMMAND** message to **OnSysCommand**.

Accelerator (shortcut) keystrokes that are defined to select items from the Control menu are translated into **OnSysCommand** calls; all other accelerator keystrokes are translated into WM_COMMAND messages.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class

Hierarchy Chart

WM_SYSCOMMAND

Concepts

CWnd Members

CWnd::OnSysDeadChar

The framework calls this member function if the **CWnd** object has the input focus when the OnSysKeyUp or OnSysKeyDown member function is called.

```
afx_msg void OnSysDeadChar(
   UINT nChar,
   UINT nRepCnt,
   UINT nFlags
);
```

Parameters

nChar

Specifies the dead-key character value.

nRepCnt

Specifies the repeat count.

nFlags

Specifies the scan code, key-transition code, previous key state, and context code, as shown in the following list:

Value	Meaning
0–7	Scan code (OEM-dependent value). Low byte of high-order word.
8	Extended key, such as a function key or a key on the numeric keypad (1 if it is an extended key; otherwise 0).
9–10	Not used.
11–12	Used internally by Windows.
13	Context code (1 if the ALT key is held down while the key is pressed; otherwise 0).
14	Previous key state (1 if the key is down before the call, 0 if the key is up).
15	Transition state (1 if the key is being released, 0 if the key is being pressed).

Remarks

It specifies the character value of a dead key.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnSysKeyDown
CWnd::OnSysKeyUp
WM_SYSDEADCHAR
CWnd::OnDeadChar

Concepts

CWnd Members

CWnd::OnSysKeyDown

If the **CWnd** object has the input focus, the **OnSysKeyDown** member function is called by the framework when the user holds down the ALT key and then presses another key.

```
afx_msg void OnSysKeyDown(
   UINT nChar,
   UINT nRepCnt,
   UINT nFlags
);
```

Parameters

nChar

Specifies the virtual key code of the key being pressed. For a list of of standard virtual key codes, see Winuser.h

nRepCnt

Specifies the repeat count.

nFlags

Specifies the scan code, key-transition code, previous key state, and context code, as shown in the following list:

Value	Meaning
0–7	Scan code (OEM-dependent value). Low byte of high-order word.
8	Extended key, such as a function key or a key on the numeric keypad (1 if it is an extended key; otherwise 0).
9–10	Not used.
11–12	Used internally by Windows.
13	Context code (1 if the ALT key is held down while the key is pressed, 0 otherwise).
14	Previous key state (1 if the key is down before the message is sent, 0 if the key is up).
15	Transition state (1 if the key is being released, 0 if the key is being pressed).

For **OnSysKeyDown** calls, the key-transition bit (bit 15) is 0. The context-code bit (bit 13) is 1 if the ALT key is down while the key is pressed; it is 0 if the message is sent to the active window because no window has the input focus.

Remarks

If no window currently has the input focus, the active window's **OnSysKeyDown** member function is called. The **CWnd** object that receives the message can distinguish between these two contexts by checking the context code in *nFlags*.

When the context code is 0, the **WM_SYSKEYDOWN** message received by **OnSysKeyDown** can be passed to the TranslateAccelerator Windows function, which will handle it as though it were a normal key message instead of a system-key message. This allows accelerator keys to be used with the active window even if the active window does not have the input focus.

Because of auto-repeat, more than one **OnSysKeyDown** call may occur before the WM_SYSKEYUP message is received. The previous key state (bit 14) can be used to determine whether the **OnSysKeyDown** call indicates the first down transition or a repeated down transition.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the slash (/) and ENTER keys in the numeric keypad. Some other keyboards may support the extended-key bit in *nFlags*.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
TranslateAccelerator
CWnd::OnSysKeyUp
WM_SYSKEYDOWN

Concepts

CWnd Members

CWnd::OnSysKeyUp

If the **CWnd** object has the focus, the **OnSysKeyUp** member function is called by the framework when the user releases a key that was pressed while the ALT key was held down.

```
afx_msg void OnSysKeyUp(
   UINT nChar,
   UINT nRepCnt,
   UINT nFlags
);
```

Parameters

nChar

Specifies the virtual key code of the key being pressed. For a list of of standard virtual key codes, see Winuser.h

nRepCnt

Specifies the repeat count.

nFlags

Specifies the scan code, key-transition code, previous key state, and context code, as shown in the following list:

Value	Meaning
0–7	Scan code (OEM-dependent value). Low byte of high-order word.
8	Extended key, such as a function key or a key on the numeric keypad (1 if it is an extended key; otherwise 0).
9–10	Not used.
11–12	Used internally by Windows.
13	Context code (1 if the ALT key is held down while the key is pressed, 0 otherwise).
14	Previous key state (1 if the key is down before the message is sent, 0 if the key is up).
15	Transition state (1 if the key is being released, 0 if the key is being pressed).

For **OnSysKeyUp** calls, the key-transition bit (bit 15) is 1. The context-code bit (bit 13) is 1 if the ALT key is down while the key is pressed; it is 0 if the message is sent to the active window because no window has the input focus.

Remarks

If no window currently has the input focus, the active window's **OnSysKeyUp** member function is called. The **CWnd** object that receives the call can distinguish between these two contexts by checking the context code in *nFlags*.

When the context code is 0, the **WM_SYSKEYUP** message received by **OnSysKeyUp** can be passed to the TranslateAccelerator Windows function, which will handle it as though it were a normal key message instead of a system-key message. This allows accelerator (shortcut) keys to be used with the active window even if the active window does not have the input focus.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the slash (/) and ENTER keys in the numeric keypad. Some other keyboards may support the extended-key bit in *nFlags*.

For non-U.S. Enhanced 102-key keyboards, the right ALT key is handled as the CTRL+ALT key combination. The following shows the sequence of messages and calls that result when the user presses and releases this key:

Sequence	Function Accessed	Message Passed

1.	WM_KEYDOWN	**VK_CONTROL**
2.	WM_KEYDOWN	**VK_MENU**
3.	WM_KEYUP	**VK_CONTROL**
4.	WM_SYSKEYUP	**VK_MENU**

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
TranslateAccelerator
CWnd::OnSysKeyDown
WM_SYSKEYUP

Concepts

CWnd Members

CWnd::OnTCard

The framework calls this member function when the user clicks an authorable button.

```
afx_msg void OnTCard(
   UINT idAction,
   DWORD dwActionData
);
```

Parameters

idAction

Indicates the action the user has taken. This parameter can be one of these values:

- **IDABORT** The user clicked an authorable Abort button.
- **IDCANCEL** The user clicked an authorable Cancel button.
- **IDCLOSE** The user closed the training card.
- **IDHELP** The user clicked an authorable Windows Help button.
- **IDIGNORE** The user clicked an authorable Ignore button.
- **IDOK** The user clicked an authorable OK button.
- **IDNO** The user clicked an authorable No button.
- **IDRETRY** The user clicked an authorable Retry button.
- **HELP_TCARD_DATA** The user clicked an authorable button. The *dwActionData* parameter contains a long integer specified by the help author.
- **HELP_TCARD_NEXT** The user clicked an authorable Next button.
- **HELP_TCARD_OTHER_CALLER** Another application has requested training cards.
- **IDYES** The user clicked an authorable Yes button.

dwActionData

If *idAction* specifies **HELP_TCARD_DATA**, this parameter is a long integer specified by the help author. Otherwise, this parameter is zero.

Remarks

This function is called only when an application has initiated a training card with Windows Help. An application initiates a training card by specifying the **HELP_TCARD** command in a call to the WinHelp function.

See Also

Reference

CWnd Class
Hierarchy Chart
WinHelp
CWinApp::WinHelp

Concepts

CWnd Members

CWnd::OnTimeChange

The framework calls this member function after the system time is changed.

```
afx_msg void OnTimeChange( );
```

Remarks

Have any application that changes the system time send this message to all top-level windows. To send the **WM_TIMECHANGE** message to all top-level windows, an application can use the SendMessage Windows function with its *hwnd* parameter set to **HWND_BROADCAST**.

See Also

Reference

CWnd Class
Hierarchy Chart
SendMessage
WM_TIMECHANGE

Concepts

CWnd Members

CWnd::OnTimer

The framework calls this member function after each interval specified in the SetTimer member function used to install a timer.

```
afx_msg void OnTimer(
   UINT_PTR nIDEvent
);
```

Parameters

nIDEvent

Specifies the identifier of the timer.

Remarks

The DispatchMessage Windows function sends a WM_TIMER message when no other messages are in the application's message queue.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

Example

See the example in CWnd::SetTimer.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::SetTimer
WM_TIMER

Concepts

CWnd Members

CWnd::OnToolHitTest

The framework calls this member function to detemine whether a point is in the bounding rectangle of the specified tool.

```
virtual INT_PTR OnToolHitTest(
   CPoint point,
   TOOLINFO* pTI
) const;
```

Parameters

point

Specifies the x- and y-coordinate of the cursor. These coordinates are always relative to the upper-left corner of the window

pTI

A pointer to a TOOLINFO structure. The following structure values are set by default:

- *hwnd* = **m_hWnd** Handle to a window
- *uId* = **(UINT)hWndChild** Handle to a child window
- *uFlags* |= **TTF_IDISHWND** Handle of the tool
- *lpszText* = **LPSTR_TEXTCALLBACK** Pointer to the string that is to be displayed in the specified window

Return Value

If the tooltip control was found, the window control ID. If the tooltip control was not found, -1.

Remarks

If the point is in the rectangle, it retrieves information about the tool.

If the area with which the tooltip is associated is not a button, **OnToolHitTest** sets the structure flags to **TTF_NOTBUTTON** and **TTF_CENTERTIP**.

Override **OnToolHitTest** to provide different information than the default provides.

See TOOLINFO, in the *Platform SDK*, for more information about the structure.

Smart Device Developer Notes

This method is not supported in smart device projects.

See Also

Reference

CWnd Class
Hierarchy Chart
TOOLINFO
CWnd::FilterToolTipMessage

Concepts

CWnd Members

CWnd::OnUpdateUIState

Called to to change the user interface (UI) state for the specified window and all its child windows.

```
afx_msg void OnUpdateUIState(
   UINT nAction,
   UINT nUIElement
);
```

Parameters

nAction

Specifies the action to be performed. Can be one of the following values:

- **UIS_CLEAR** The UI state element (specified by *nUIElement*) should be hidden.
- **UIS_INITIALIZE** The UI state element (specified by *nUIElement*) should be changed based on the last input event. For more information, see the **Remarks** section of WM_UPDATEISTATE.
- **UIS_SET** The UI state element (specified by *nUIElement*) should be visible.

nUIElement

Specifies which UI state elements are affected or the style of the control. Can be one of the following values:

- **UISF_HIDEACCEL** Keyboard accelerators.
- **UISF_HIDEFOCUS** Focus indicators.
- **UISF_ACTIVE** **Windows XP:** A control should be drawn in the style used for active controls.

Remarks

This member function emulates the functionality of the WM_UPDATEUISTATE message, as described in the *Platform SDK*.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnChangeUIState
CWnd::OnQueryUIState

Concepts

CWnd Members

CWnd::OnVKeyToItem

If the **CWnd** object owns a list box with the LBS_WANTKEYBOARDINPUT style, the list box will send the **WM_VKEYTOITEM** message in response to a **WM_KEYDOWN** message.

```
afx_msg int OnVKeyToItem(
   UINT nKey,
   CListBox* pListBox,
   UINT nIndex
);
```

Parameters

nKey

Specifies the virtual key code of the key that the user pressed. For a list of of standard virtual key codes, see Winuser.h

pListBox

Specifies a pointer to the list box. The pointer may be temporary and should not be stored for later use.

nIndex

Specifies the current caret position.

Return Value

Specifies the action that the application performed in response to the message. A return value of –2 indicates that the application handled all aspects of selecting the item and requires no further action by the list box. A return value of –1 indicates that the list box should perform the default action in response to the keystroke. A return value of 0 or greater specifies the zero-based index of an item in the list box and indicates that the list box should perform the default action for the keystroke on the given item.

Remarks

This member function is called by the framework only for list boxes that have the LBS_HASSTRINGS style.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnKeyDown
WM_VKEYTOITEM

Concepts

CWnd Members

CWnd::OnVScroll

The framework calls this member function when the user clicks the window's vertical scroll bar.

```
afx_msg void OnVScroll(
   UINT nSBCode,
   UINT nPos,
   CScrollBar* pScrollBar
);
```

Parameters

nSBCode

Specifies a scroll-bar code that indicates the user's scrolling request. This parameter can be one of the following:

- **SB_BOTTOM** Scroll to bottom.
- **SB_ENDSCROLL** End scroll.
- **SB_LINEDOWN** Scroll one line down.
- **SB_LINEUP** Scroll one line up.
- **SB_PAGEDOWN** Scroll one page down.
- **SB_PAGEUP** Scroll one page up.
- **SB_THUMBPOSITION** Scroll to the absolute position. The current position is provided in *nPos*.
- **SB_THUMBTRACK** Drag scroll box to specified position. The current position is provided in *nPos*.
- **SB_TOP** Scroll to top.

nPos

Contains the current scroll-box position if the scroll-bar code is **SB_THUMBPOSITION** or **SB_THUMBTRACK**; otherwise not used. Depending on the initial scroll range, *nPos* may be negative and should be cast to an **int** if necessary.

pScrollBar

If the scroll message came from a scroll-bar control, contains a pointer to the control. If the user clicked a window's scroll bar, this parameter is **NULL**. The pointer may be temporary and should not be stored for later use.

Remarks

OnVScroll typically is used by applications that give some feedback while the scroll box is being dragged.

If **OnVScroll** scrolls the contents of the **CWnd** object, it must also reset the position of the scroll box with the SetScrollPos member function.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::SetScrollPos
CWnd::OnHScroll
WM_VSCROLL

Concepts

CWnd Members

CWnd::OnVScrollClipboard

The Clipboard owner's **OnVScrollClipboard** member function is called by the Clipboard viewer when the Clipboard data has the **CF_OWNERDISPLAY** format and there is an event in the Clipboard viewer's vertical scroll bar.

```
afx_msg void OnVScrollClipboard(
   CWnd* pClipAppWnd,
   UINT nSBCode,
   UINT nPos
);
```

Parameters

pClipAppWnd

Specifies a pointer to a Clipboard-viewer window. The pointer may be temporary and should not be stored for later use.

nSBCode

Specifies one of the following scroll-bar values:

- **SB_BOTTOM** Scroll to bottom.
- **SB_ENDSCROLL** End scroll.
- **SB_LINEDOWN** Scroll one line down.
- **SB_LINEUP** Scroll one line up.
- **SB_PAGEDOWN** Scroll one page down.
- **SB_PAGEUP** Scroll one page up.
- **SB_THUMBPOSITION** Scroll to the absolute position. The current position is provided in *nPos*.
- **SB_TOP** Scroll to top.

nPos

Contains the scroll-box position if the scroll-bar code is **SB_THUMBPOSITION**; otherwise *nPos* is not used.

Remarks

The owner should scroll the Clipboard image, invalidate the appropriate section, and update the scroll-bar values.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::Invalidate
CWnd::OnHScrollClipboard
CWnd::InvalidateRect
WM_VSCROLLCLIPBOARD
CWnd::Default

Concepts

CWnd Members

CWnd::OnWindowPosChanged

The framework calls this member function when the size, position, or Z-order has changed as a result of a call to the SetWindowPos member function or another window-management function.

```
afx_msg void OnWindowPosChanged(
   WINDOWPOS* lpwndpos
);
```

Parameters

lpwndpos

Points to a WINDOWPOS data structure that contains information about the window's new size and position.

Remarks

The default implementation sends the WM_SIZE and WM_MOVE messages to the window. These messages are not sent if an application handles the **OnWindowPosChanged** call without calling its base class. It is more efficient to perform any move or size change processing during the call to **OnWindowPosChanged** without calling its base class.

Note

This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class

Hierarchy Chart

WM_WINDOWPOSCHANGED

Concepts

CWnd Members

CWnd::OnWindowPosChanging

The framework calls this member function when the size, position, or Z-order is about to change as a result of a call to the SetWindowPos member function or another window-management function.

```
afx_msg void OnWindowPosChanging(
   WINDOWPOS* lpwndpos
);
```

Parameters

lpwndpos

Points to a **WINDOWPOS** data structure that contains information about the window's new size and position.

Remarks

An application can prevent changes to the window by setting or clearing the appropriate bits in the **flags** member of the WINDOWPOS structure.

For a window with the WS_OVERLAPPED or WS_THICKFRAME style, the default implementation sends a WM_GETMINMAXINFO message to the window. This is done to validate the new size and position of the window and to enforce the **CS_BYTEALIGNCLIENT** and **CS_BYTEALIGN** client styles. An application can override this functionality by not calling its base class.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnWindowPosChanged
WM_WINDOWPOSCHANGING

Concepts

CWnd Members

CWnd::OnWinIniChange

The framework calls this member function after a change has been made to the Windows initialization file, WIN.INI.

```
afx_msg void OnWinIniChange(
   LPCTSTR lpszSection
);
```

Parameters

lpszSection

Points to a string that specifies the name of the section that has changed. (The string does not include the square brackets that enclose the section name.)

Remarks

The SystemParametersInfo Windows function calls **OnWinIniChange** after an application uses the function to change a setting in the WIN.INI file.

To send the **WM_WININICHANGE** message to all top-level windows, an application can use the SendMessage Windows function with its *hwnd* parameter set to **HWND_BROADCAST**.

If an application changes many different sections in WIN.INI at the same time, the application should send one **WM_WININICHANGE** message with *lpszSection* set to **NULL**. Otherwise, an application should send **WM_WININICHANGE** each time it makes a change to WIN.INI.

If an application receives an **OnWinIniChange** call with *lpszSection* set to **NULL**, the application should check all sections in WIN.INI that affect the application.

Note
This member function is called by the framework to allow your application to handle a Windows message. The parameters passed to your function reflect the parameters received by the framework when the message was received. If you call the base-class implementation of this function, that implementation will use the parameters originally passed with the message and not the parameters you supply to the function.

See Also

Reference

CWnd Class
Hierarchy Chart
SendMessage
SystemParametersInfo
WM_WININICHANGE

Concepts

CWnd Members

CWnd::OnWndMsg

This member function is called by **WindowProc**, or is called during message reflection.

```
virtual BOOL OnWndMsg(
   UINT message,
   WPARAM wParam,
   LPARAM lParam,
   LRESULT* pResult
);
```

Parameters

message

Specifies the message to be sent.

wParam

Specifies additional message-dependent information.

lParam

Specifies additional message-dependent information.

pResult

The return value of WindowProc. Depends on the message; may be **NULL**.

Return Value

TRUE if message was handled; otherwise **FALSE**.

Remarks

OnWndMsg determines the message type and either calls the appropriate framework function (for example, OnCommand for **WM_COMMAND**) or finds the appropriate message in the message map.

For more information about message reflection, see Handling Reflected Messages.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnChildNotify
CWnd::SendChildNotifyLastMsg
CWnd::ReflectChildNotify
CCmdTarget::OnCmdMsg
CWnd::ReflectLastMsg

Concepts

CWnd Members

CWnd::OpenClipboard

Opens the Clipboard.

```
BOOL OpenClipboard( );
```

Return Value

Nonzero if the Clipboard is opened via **CWnd**, or 0 if another application or window has the Clipboard open.

Remarks

Other applications will not be able to modify the Clipboard until the CloseClipboard Windows function is called.

The current **CWnd** object will not become the owner of the Clipboard until the EmptyClipboard Windows function is called.

Example

```
//handler for Edit | Copy menu
void CTestDlgView::OnEditCopy()
{
   if ( !OpenClipboard() )
   {
      AfxMessageBox( "Cannot open the Clipboard" );
      return;
   }
   // Remove the current Clipboard contents
   if( !EmptyClipboard() )
   {
      AfxMessageBox( "Cannot empty the Clipboard" );
      return;
   }
   // ...
   // Get the currently selected data, hData handle to
   // global memory of data
   // ...
   // For the appropriate data formats...
   if ( ::SetClipboardData( CF_??, hData ) == NULL )
   {
      AfxMessageBox( "Unable to set Clipboard data" );
      CloseClipboard();
      return;
   }
   // ...
   CloseClipboard();
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CloseClipboard
EmptyClipboard
OpenClipboard

Concepts

CWnd Members

CWnd::PaintWindowlessControls

Draws windowless controls on the control container.

```
BOOL PaintWindowlessControls(
   CDC *pDC
);
```

Parameters

pDC

The device context on which to draw the windowless controls.

Return Value

Returns TRUE if there is a control container and the windowless controls are drawn successfully, otherwise FALSE.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::PostMessage

Places a message in the window's message queue and then returns without waiting for the corresponding window to process the message.

```
BOOL PostMessage(
   UINT message,
   WPARAM wParam = 0,
   LPARAM lParam = 0
);
```

Parameters

message

Specifies the message to be posted.

wParam

Specifies additional message information. The content of this parameter depends on the message being posted.

lParam

Specifies additional message information. The content of this parameter depends on the message being posted.

Return Value

Nonzero if the message is posted; otherwise 0.

Remarks

Messages in a message queue are retrieved by calls to the GetMessage or PeekMessage Windows function.

The Windows PostMessage function can be used to access another application.

Example

See the example for AfxGetMainWnd.

See Also

Reference

CWnd Class
Hierarchy Chart
GetMessage
PeekMessage
PostMessage
PostThreadMessage
CWnd::SendMessage

Concepts

CWnd Members

CWnd::PostNcDestroy

Called by the default OnNcDestroy member function after the window has been destroyed.

```
virtual void PostNcDestroy( );
```

Remarks

Derived classes can use this function for custom cleanup such as the deletion of the **this** pointer.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::OnNcDestroy

Concepts

CWnd Members

CWnd::PreCreateWindow

Called by the framework before the creation of the Windows window attached to this **CWnd** object.

```
virtual BOOL PreCreateWindow(
   CREATESTRUCT& cs
);
```

Parameters

cs

A CREATESTRUCT structure.

Return Value

Nonzero if the window creation should continue; 0 to indicate creation failure.

Remarks

Never call this function directly.

The default implementation of this function checks for a **NULL** window class name and substitutes an appropriate default. Override this member function to modify the **CREATESTRUCT** structure before the window is created.

Each class derived from **CWnd** adds its own functionality to its override of **PreCreateWindow**. By design, these derivations of **PreCreateWindow** are not documented. To determine the styles appropriate to each class and the interdependencies between the styles, you can examine the MFC source code for your application's base class. If you choose to override **PreCreateWindow,** you can determine whether the styles used in your application's base class provide the functionality you need by using information gathered from the MFC source code.

For more information on changing window styles, see the Changing the Styles of a Window Created by MFC.

Example

```
// alter the styles of the main frame window
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
   // Create a window without min/max buttons or sizable border
   cs.style = WS_OVERLAPPED | WS_SYSMENU | WS_BORDER;

   // Size the window to 1/3 screen size and center it
   cs.cy = ::GetSystemMetrics(SM_CYSCREEN) / 3;
   cs.cx = ::GetSystemMetrics(SM_CXSCREEN) / 3;
   cs.y = ((cs.cy * 3) - cs.cy) / 2;
   cs.x = ((cs.cx * 3) - cs.cx) / 2;

    return CFrameWnd::PreCreateWindow(cs);
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::Create
CWnd::CreateEx
CREATESTRUCT Structure

Concepts

CWnd Members

CWnd::PreSubclassWindow

This member function is called by the framework to allow other necessary subclassing to occur before the window is subclassed.

```
virtual void PreSubclassWindow( );
```

Remarks

Overriding this member function allows for dynamic subclassing of controls. It is an advanced overridable.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::SubclassWindow
CWnd::UnsubclassWindow
CWnd::GetSuperWndProcAddr
CWnd::DefWindowProc
CWnd::SubclassDlgItem
CWnd::Attach
CWnd::PreCreateWindow

Concepts

CWnd Members

CWnd::PreTranslateMessage

Used by class CWinApp to translate window messages before they are dispatched to the TranslateMessage and DispatchMessage Windows functions.

```
virtual BOOL PreTranslateMessage(
   MSG* pMsg
);
```

Parameters

pMsg

Points to a MSG structure that contains the message to process.

Return Value

Nonzero if the message was translated and should not be dispatched; 0 if the message was not translated and should be dispatched.

See Also

Reference

CWnd Class
Hierarchy Chart
TranslateMessage
IsDialogMessage
CWinApp::PreTranslateMessage

Concepts

CWnd Members

CWnd::Print

Call this member function to draw the current window in the specified device context, which is most commonly in a printer device context.

```
void Print(
   CDC* pDC,
   DWORD dwFlags
) const;
```

Parameters

pDC

A pointer to a device context.

dwFlags

Specifies the drawing options. This parameter can be one or more of these flags:

- **PRF_CHECKVISIBLE** Draw the window only if it is visible.
- **PRF_CHILDREN** Draw all visible children windows.
- **PRF_CLIENT** Draw the client area of the window.
- **PRF_ERASEBKGND** Erase the background before drawing the window.
- **PRF_NONCLIENT** Draw the nonclient area of the window.
- **PRF_OWNED** Draw all owned windows.

Remarks

CWnd::DefWindowProc function processes this message based on which drawing option is specified:

- If **PRF_CHECKVISIBLE** is specified and the window is not visible, do nothing.
- If **PRF_NONCLIENT** is specified, draw the nonclient area in the given device context.
- If **PRF_ERASEBKGND** is specified, send the window a WM_ERASEBKGND message.
- If **PRF_CLIENT** is specified, send the window a WM_PRINTCLIENT message.
- If **PRF_CHILDREN** is set, send each visible child window a WM_PRINT message.
- If **PRF_OWNED** is set, send each visible owned window a **WM_PRINT** message.

See Also

Reference

CWnd Class
Hierarchy Chart
WM_PRINT
WM_PRINTCLIENT

Concepts

CWnd Members

CWnd::PrintClient

Call this member function to draw any window in the specified device context (usually a printer device context).

```
void PrintClient(
   CDC* pDC,
   DWORD dwFlags
) const;
```

Parameters

pDC

A pointer to a device context.

dwFlags

Specifies drawing options. This parameter can be one or more of these flags:

- **PRF_CHECKVISIBLE** Draw the window only if it is visible.
- **PRF_CHILDREN** Draw all visible children windows.
- **PRF_CLIENT** Draw the client area of the window.
- **PRF_ERASEBKGND** Erase the background before drawing the window.
- **PRF_NONCLIENT** Draw the nonclient area of the window.
- **PRF_OWNED** Draw all owned windows.

See Also

Reference

CWnd Class
Hierarchy Chart
WM_PRINTCLIENT

Concepts

CWnd Members

CWnd::PrintWindow

Copies a visual window into the specified device context, typically a printer DC.

```
BOOL PrintWindow(
   CDC* pDC,
   UINT nFlags
) const;
```

Parameters

pDC

A pointer to the device context to be printed to.

nFlags

Specifies the drawing options. For a list of possible values, see PrintWindow.

Return Value

Nonzero if the function succeeds; otherwise 0.

Remarks

This member function emulates the functionality of the function PrintWindow, as described in the Platform SDK.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::RedrawWindow

Updates the specified rectangle or region in the given window's client area.

```
BOOL RedrawWindow(
   LPCRECT lpRectUpdate = NULL,
   CRgn* prgnUpdate = NULL,
   UINT flags = RDW_INVALIDATE | RDW_UPDATENOW | RDW_ERASE
);
```

Parameters

lpRectUpdate

Points to a RECT structure containing the coordinates of the update rectangle. This parameter is ignored if *prgnUpdate* contains a valid region handle.

prgnUpdate

Identifies the update region. If both *prgnUpdate* and *lpRectUpdate* are **NULL**, the entire client area is added to the update region.

flags

The following flags are used to invalidate the window:

- **RDW_ERASE** Causes the window to receive a WM_ERASEBKGND message when the window is repainted. The **RDW_INVALIDATE** flag must also be specified; otherwise **RDW_ERASE** has no effect.
- **RDW_FRAME** Causes any part of the nonclient area of the window that intersects the update region to receive a WM_NCPAINT message. The **RDW_INVALIDATE** flag must also be specified; otherwise **RDW_FRAME** has no effect.
- **RDW_INTERNALPAINT** Causes a WM_PAINT message to be posted to the window regardless of whether the window contains an invalid region.
- **RDW_INVALIDATE** Invalidate *lpRectUpdate* or *prgnUpdate* (only one may be not **NULL**). If both are **NULL**, the entire window is invalidated.

The following flags are used to validate the window:

- **RDW_NOERASE** Suppresses any pending **WM_ERASEBKGND** messages.
- **RDW_NOFRAME** Suppresses any pending **WM_NCPAINT** messages. This flag must be used with **RDW_VALIDATE** and is typically used with **RDW_NOCHILDREN**. This option should be used with care, as it could prevent parts of a window from painting properly.
- **RDW_NOINTERNALPAINT** Suppresses any pending internal **WM_PAINT** messages. This flag does not affect **WM_PAINT** messages resulting from invalid areas.
- **RDW_VALIDATE** Validates *lpRectUpdate* or *prgnUpdate* (only one may be not **NULL**). If both are **NULL**, the entire window is validated. This flag does not affect internal **WM_PAINT** messages.

The following flags control when repainting occurs. Painting is not performed by the **RedrawWindow** function unless one of these bits is specified.

- **RDW_ERASENOW** Causes the affected windows (as specified by the **RDW_ALLCHILDREN** and **RDW_NOCHILDREN** flags) to receive **WM_NCPAINT** and **WM_ERASEBKGND** messages, if necessary, before the function returns. **WM_PAINT** messages are deferred.
- **RDW_UPDATENOW** Causes the affected windows (as specified by the **RDW_ALLCHILDREN** and **RDW_NOCHILDREN** flags) to receive **WM_NCPAINT**, **WM_ERASEBKGND**, and **WM_PAINT** messages, if necessary, before the function returns.

By default, the windows affected by the **RedrawWindow** function depend on whether the specified window has the **WS_CLIPCHILDREN** style. The child windows of **WS_CLIPCHILDREN** windows are not affected. However, those windows

that are not **WS_CLIPCHILDREN** windows are recursively validated or invalidated until a **WS_CLIPCHILDREN** window is encountered. The following flags control which windows are affected by the **RedrawWindow** function:

- **RDW_ALLCHILDREN** Includes child windows, if any, in the repainting operation.
- **RDW_NOCHILDREN** Excludes child windows, if any, from the repainting operation.

Return Value

Nonzero if the window was redrawn successfully; otherwise 0.

Remarks

When the **RedrawWindow** member function is used to invalidate part of the desktop window, that window does not receive a WM_PAINT message. To repaint the desktop, an application should use CWnd::ValidateRgn, CWnd::InvalidateRgn, CWnd::UpdateWindow, or RedrawWindow

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::ReflectChildNotify

This message function is called by the framework from OnChildNotify.

```
BOOL ReflectChildNotify(
   UINT message,
   WPARAM wParam,
   LPARAM lParam,
   LRESULT* pResult
);
```

Parameters

message

Specifies the message to be reflected.

wParam

Specifies additional message-dependent information.

lParam

Specifies additional message-dependent information.

pResult

The result generated by the child window to be returned by the parent window. Can be **NULL**.

Return Value

TRUE if message was reflected; otherwise **FALSE**.

Remarks

It is a helper function which reflects *message* to its source.

Reflected messages are sent directly to CWnd::OnWndMsg or CCmdTarget::OnCmdMsg.

For more information about message reflection, see Handling Reflected Messages.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnChildNotify
CWnd::SendChildNotifyLastMsg
CWnd::OnWndMsg
CCmdTarget::OnCmdMsg
CWnd::ReflectLastMsg

Concepts

CWnd Members

CWnd::ReflectLastMsg

This member function is called by the framework to reflect the last message to the child window.

```
static BOOL PASCAL ReflectLastMsg(
   HWND hWndChild,
   LRESULT* pResult = NULL
);
```

Parameters

hWndChild

A handle to a child window.

pResult

The result generated by the child window to be returned by the parent window. Can be **NULL**.

Return Value

Nonzero if the message was handled; otherwise 0.

Remarks

This member function calls SendChildNotifyLastMsg if the window identified by *hWndChild* is an OLE control or a window in the permanent map.

For more information about message reflection, see Handling Reflected Messages.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::OnChildNotify
CWnd::SendChildNotifyLastMsg
CWnd::ReflectChildNotify
CCmdTarget::OnCmdMsg

Concepts

CWnd Members

CWnd::ReleaseDC

Releases a device context, freeing it for use by other applications.

```
int ReleaseDC(
   CDC* pDC
);
```

Parameters

pDC

Identifies the device context to be released.

Return Value

Nonzero if successful; otherwise 0.

Remarks

The effect of the **ReleaseDC** member function depends on the device-context type.

The application must call the **ReleaseDC** member function for each call to the GetWindowDC member function and for each call to the GetDC member function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetDC
CWnd::GetWindowDC
ReleaseDC

Concepts

CWnd Members

CWnd::RepositionBars

Called to reposition and resize control bars in the client area of a window.

```
void RepositionBars(
   UINT nIDFirst,
   UINT nIDLast,
   UINT nIDLeftOver,
   UINT nFlag = reposDefault,
   LPRECT lpRectParam = NULL,
   LPCRECT lpRectClient = NULL,
   BOOL bStretch = TRUE
);
```

Parameters

nIDFirst

The ID of the first in a range of control bars to reposition and resize.

nIDLast

The ID of the last in a range of control bars to reposition and resize.

nIDLeftOver

Specifies ID of pane that fills the rest of the client area.

nFlag

Can have one of the following values:

- **CWnd::reposDefault** Performs the layout of the control bars. *lpRectParam* is not used and can be **NULL**.
- **CWnd::reposQuery** The layout of the control bars is not done; instead *lpRectParam* is initialized with the size of the client area, as if the layout had actually been done.
- **CWnd::reposExtra** Adds the values of *lpRectParam* to the client area of *nIDLast* and also performs the layout.

lpRectParam

Points to a RECT structure; the usage of which depends on the value of *nFlag*.

lpRectClient

Points to a RECT structure containing the available client area. If **NULL**, the window's client area will be used.

bStretch

Indicates whether the bar should be stretched to the size of the frame.

Remarks

The *nIDFirst* and *nIDLast* parameters define a range of control-bar IDs to be repositioned in the client area. The *nIDLeftOver* parameter specifies the ID of the child window (normally the view) which is repositioned and resized to fill the rest of the client area not filled by control bars.

See Also

Reference

CWnd Class
Hierarchy Chart
CFrameWnd::RecalcLayout

Concepts

CWnd Members

CWnd::RunModalLoop

Call this member function to retrieve, translate, or dispatch messages until ContinueModal returns **FALSE**.

```
int RunModalLoop(
   DWORD dwFlags = 0
);
```

Parameters

dwFlags

Specifies the Windows message to be sent. Can be one of the following values:

- **MLF_NOIDLEMSG** Don't send WM_ENTERIDLE messages to the parent.
- **MLF_NOKICKIDLE** Don't send **WM_KICKIDLE** messages to the window.
- **MLF_SHOWONIDLE** Show the window when message queue goes idle.

Return Value

Specifies the value of the *nResult* parameter passed to the EndModalLoop member function, which is then used to end the modal loop.

Remarks

By default, **ContinueModal** returns **FALSE** after **EndModalLoop** is called. Returns the value provided as *nResult* to **EndModalLoop**.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::EndModalLoop
WM_ENTERIDLE

Concepts

CWnd Members

CWnd::ScreenToClient

Converts the screen coordinates of a given point or rectangle on the display to client coordinates.

```
void ScreenToClient(
   LPPOINT lpPoint
) const;
void ScreenToClient(
   LPRECT lpRect
) const;
```

Parameters

lpPoint

Points to a CPoint object or POINT structure that contains the screen coordinates to be converted.

lpRect

Points to a CRect object or RECT structure that contains the screen coordinates to be converted.

Remarks

The **ScreenToClient** member function replaces the screen coordinates given in *lpPoint* or *lpRect* with client coordinates. The new coordinates are relative to the upper-left corner of the **CWnd** client area.

Example

See the example for CListCtrl::GetItemRect.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::ClientToScreen
ScreenToClient

Concepts

CWnd Members

CWnd::ScrollWindow

Scrolls the contents of the client area of the current **CWnd** object.

```
void ScrollWindow(
   int xAmount,
   int yAmount,
   LPCRECT lpRect = NULL,
   LPCRECT lpClipRect = NULL
);
```

Parameters

xAmount

Specifies the amount, in device units, of horizontal scrolling. This parameter must be a negative value to scroll to the left.

yAmount

Specifies the amount, in device units, of vertical scrolling. This parameter must be a negative value to scroll up.

lpRect

Points to a CRect object or RECT structure that specifies the portion of the client area to be scrolled. If *lpRect* is **NULL**, the entire client area is scrolled. The caret is repositioned if the cursor rectangle intersects the scroll rectangle.

lpClipRect

Points to a **CRect** object or **RECT** structure that specifies the clipping rectangle to scroll. Only bits inside this rectangle are scrolled. Bits outside this rectangle are not affected even if they are in the *lpRect* rectangle. If *lpClipRect* is **NULL**, no clipping is performed on the scroll rectangle.

Remarks

If the caret is in the **CWnd** being scrolled, **ScrollWindow** automatically hides the caret to prevent it from being erased and then restores the caret after the scroll is finished. The caret position is adjusted accordingly.

The area uncovered by the **ScrollWindow** member function is not repainted but is combined into the current **CWnd** object's update region. The application will eventually receive a WM_PAINT message notifying it that the region needs repainting. To repaint the uncovered area at the same time the scrolling is done, call the UpdateWindow member function immediately after calling **ScrollWindow**.

If *lpRect* is **NULL**, the positions of any child windows in the window are offset by the amount specified by *xAmount* and *yAmount*, and any invalid (unpainted) areas in the **CWnd** are also offset. **ScrollWindow** is faster when *lpRect* is **NULL**.

If *lpRect* is not **NULL**, the positions of child windows are not changed, and invalid areas in **CWnd** are not offset. To prevent updating problems when *lpRect* is not **NULL**, call the **UpdateWindow** member function to repaint **CWnd** before calling **ScrollWindow**.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::UpdateWindow
ScrollWindow

Concepts

CWnd Members

CWnd::ScrollWindowEx

Scrolls the contents of a window's client area.

```
int ScrollWindowEx(
   int dx,
   int dy,
   LPCRECT lpRectScroll,
   LPCRECT lpRectClip,
   CRgn* prgnUpdate,
   LPRECT lpRectUpdate,
   UINT flags
);
```

Parameters

dx

Specifies the amount, in device units, of horizontal scrolling. This parameter must have a negative value to scroll to the left.

dy

Specifies the amount, in device units, of vertical scrolling. This parameter must have a negative value to scroll up.

lpRectScroll

Points to a RECT structure that specifies the portion of the client area to be scrolled. If this parameter is **NULL**, the entire client area is scrolled.

lpRectClip

Points to a **RECT** structure that specifies the clipping rectangle to scroll. This structure takes precedence over the rectangle pointed to by *lpRectScroll*. Only bits inside this rectangle are scrolled. Bits outside this rectangle are not affected even if they are in the *lpRectScroll* rectangle. If this parameter is **NULL**, no clipping is performed on the scroll rectangle.

prgnUpdate

Identifies the region that is modified to hold the region invalidated by scrolling. This parameter may be **NULL**.

lpRectUpdate

Points to a **RECT** structure that will receive the boundaries of the rectangle invalidated by scrolling. This parameter may be **NULL**.

flags

Can have one of the following values:

- **SW_ERASE** When specified with **SW_INVALIDATE**, erases the newly invalidated region by sending a WM_ERASEBKGND message to the window.
- **SW_INVALIDATE** Invalidates the region identified by *prgnUpdate* after scrolling.
- **SW_SCROLLCHILDREN** Scrolls all child windows that intersect the rectangle pointed to by *lpRectScroll* by the number of pixels specified in *dx* and *dy*. Windows sends a WM_MOVE message to all child windows that intersect *lpRectScroll*, even if they do not move. The caret is repositioned when a child window is scrolled and the cursor rectangle intersects the scroll rectangle.

Return Value

The return value is **SIMPLEREGION** (rectangular invalidated region), **COMPLEXREGION** (nonrectangular invalidated region; overlapping rectangles), or **NULLREGION** (no invalidated region), if the function is successful; otherwise the return value is **ERROR**.

Remarks

This function is similar to the ScrollWindow function, with some additional features.

If SW_INVALIDATE and SW_ERASE are not specified, the **ScrollWindowEx** member function does not invalidate the area that is scrolled away from. If either of these flags is set, **ScrollWindowEx** invalidates this area. The area is not updated until the application calls the UpdateWindow member function, calls the RedrawWindow member function (specifying RDW_UPDATENOW or RDW_ERASENOW), or retrieves the WM_PAINT message from the application queue.

If the window has the WS_CLIPCHILDREN style, the returned areas specified by *prgnUpdate* and *lpRectUpdate* represent the total area of the scrolled window that must be updated, including any areas in child windows that need updating.

If the SW_SCROLLCHILDREN flag is specified, Windows will not properly update the screen if part of a child window is scrolled. The part of the scrolled child window that lies outside the source rectangle will not be erased and will not be redrawn properly in its new destination. Use the DeferWindowPos Windows function to move child windows that do not lie completely within the *lpRectScroll* rectangle. The cursor is repositioned if the **SW_SCROLLCHILDREN** flag is set and the caret rectangle intersects the scroll rectangle.

All input and output coordinates (for *lpRectScroll*, *lpRectClip*, *lpRectUpdate*, and *prgnUpdate*) are assumed to be in client coordinates, regardless of whether the window has the **CS_OWNDC** or **CS_CLASSDC** class style. Use the LPtoDP and DPtoLP Windows functions to convert to and from logical coordinates, if necessary.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::RedrawWindow
CDC::ScrollDC
CWnd::ScrollWindow
CWnd::UpdateWindow
DeferWindowPos
ScrollWindowEx

Concepts

CWnd Members

CWnd::SendChildNotifyLastMsg

This member function is called by the framework to provide a notification message to a child window, from the parent window, so the child window can handle a task.

```
BOOL SendChildNotifyLastMsg(
   LRESULT* pResult = NULL
);
```

Parameters

pResult

The result generated by the child window to be returned by the parent window.

Return Value

Nonzero if the child window has handled the message sent to its parent; otherwise 0.

Remarks

SendChildNotifyLastMsg send the current message to the source if it is a message that is reflected.

For more information about message reflection, see Handling Reflected Messages.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::OnChildNotify

Concepts

CWnd Members

CWnd::SendDlgItemMessage

Sends a message to a control.

```
LRESULT SendDlgItemMessage(
   int nID,
   UINT message,
   WPARAM wParam = 0,
   LPARAM lParam = 0
);
```

Parameters

nID

Specifies the identifier of the dialog control that will receive the message.

message

Specifies the message to be sent.

wParam

Specifies additional message-dependent information.

lParam

Specifies additional message-dependent information.

Return Value

Specifies the value returned by the control's window procedure, or 0 if the control was not found.

Remarks

The **SendDlgItemMessage** member function does not return until the message has been processed.

Using **SendDlgItemMessage** is identical to obtaining a **CWnd*** to the given control and calling the SendMessage member function.

Example

```
BOOL CMyDlg::OnInitDialog()
{
   CDialog::OnInitDialog();

   //set the min and max range of the up/down or spin control
   SendDlgItemMessage(IDC_SPIN1, UDM_SETRANGE, 0, (LPARAM) MAKELONG (8, 1));

   return TRUE;  // return TRUE unless you set the focus to a control
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::SendMessage
SendDlgItemMessage

Concepts

CWnd Members

CWnd::SendMessage

Sends the specified message to this window.

```
LRESULT SendMessage(
   UINT message,
   WPARAM wParam = 0,
   LPARAM lParam = 0
);
```

Parameters

message

Specifies the message to be sent.

wParam

Specifies additional message-dependent information.

lParam

Specifies additional message-dependent information.

Return Value

The result of the message processing; its value depends on the message sent.

Remarks

The **SendMessage** member function calls the window procedure directly and does not return until that window procedure has processed the message. This is in contrast to the PostMessage member function, which places the message into the window's message queue and returns immediately.

Example

```
// In a dialog-based app, if you add a minimize button to your
// dialog, you will need the code below to draw the icon.
void CMyDlg::OnPaint()
{
   if (IsIconic())
   {
      CPaintDC dc(this); // device context for painting

      SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

      // Center icon in client rectangle
      int cxIcon = GetSystemMetrics(SM_CXICON);
      int cyIcon = GetSystemMetrics(SM_CYICON);
      CRect rect;
      GetClientRect(&rect);
      int x = (rect.Width() - cxIcon + 1) / 2;
      int y = (rect.Height() - cyIcon + 1) / 2;

      // Draw the icon represented by handle m_hIcon
      dc.DrawIcon(x, y, m_hIcon);
   }
   else
   {
      CDialog::OnPaint();
   }
}
```

See Also

Reference

CWnd Class

Hierarchy Chart
InSendMessage
CWnd::PostMessage
CWnd::SendDlgItemMessage
SendMessage

Concepts

CWnd Members

CWnd::SendMessageToDescendants

Call this member function to send the specified Windows message to all descendant windows.

```
void SendMessageToDescendants(
   UINT message,
   WPARAM wParam = 0,
   LPARAM lParam = 0,
   BOOL bDeep = TRUE,
   BOOL bOnlyPerm = FALSE
);
```

Parameters

message

Specifies the message to be sent.

wParam

Specifies additional message-dependent information.

lParam

Specifies additional message-dependent information.

bDeep

Specifies the level to which to search. If **TRUE**, recursively search all children; if **FALSE**, search only immediate children.

bOnlyPerm

Specifies whether the message will be received by temporary windows. If **TRUE**, temporary windows can receive the message; if **FALSE**, only permanent windows receive the message. For more information on temporary windows see Technical Note 3.

Remarks

If *bDeep* is **FALSE**, the message is sent just to the immediate children of the window; otherwise the message is sent to all descendant windows.

If *bDeep* and *bOnlyPerm* are **TRUE**, the search continues below temporary windows. In this case, only permanent windows encountered during the search receive the message. If *bDeep* is **FALSE**, the message is sent only to the immediate children of the window.

Example

```
// change font of child controls of a dialog
BOOL CCtrldilg::OnInitDialog()
{
    CDialog::OnInitDialog();

    LOGFONT lf;
    // redraw of child controls not needed in OnInitDialog
    // since controls aren't drawn yet.
    short int fRedraw= FALSE;

    memset(&lf, 0, sizeof(LOGFONT));  // Clear out structure.
    lf.lfHeight = 15;                 // Request a 15-pixel-high font

    // with face name "Arial".
    strcpy_s(lf.lfFaceName, LF_FACESIZE, "Arial");

    m_font.CreateFontIndirect(&lf);   // Create the font.

    SendMessageToDescendants(WM_SETFONT,
      (WPARAM)m_font.m_hObject,  //handle to font
      MAKELONG ((WORD) fRedraw, 0),
```

```
        FALSE);  // send to all descendants(TRUE) or
                 // just children of *this (FALSE)

    return TRUE;
}
```

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::SendMessage

CWnd::FromHandlePermanent

CWnd::FromHandle

Concepts

CWnd Members

CWnd::SendNotifyMessage

Sends the specified message to the window.

```
BOOL SendNotifyMessage(
   UINT message,
   WPARAM wParam,
   LPARAM lParam
);
```

Parameters

message

Specifies the message to be sent.

wParam

Specifies additional message-dependent information.

lParam

Specifies additional message-dependent information.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

If the window was created by the calling thread, **SendNotifyMessage** calls the window procedure for the window and does not return until the window procedure has processed the message. If the window was created by a different thread, **SendNotifyMessage** passes the message to the window procedure and returns immediately; it does not wait for the window procedure to finish processing the message.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::SendMessage
SendNotifyMessage

Concepts

CWnd Members

CWnd::SetActiveWindow

Makes **CWnd** the active window.

```
CWnd* SetActiveWindow( );
```

Return Value

The window that was previously active.

The returned pointer may be temporary and should not be stored for later use.

Remarks

The **SetActiveWindow** member function should be used with care since it allows an application to arbitrarily take over the active window and input focus. Normally, Windows takes care of all activation.

See Also

Reference

CWnd Class
Hierarchy Chart
SetActiveWindow
CWnd::GetActiveWindow

Concepts

CWnd Members

CWnd::SetCapture

Causes all subsequent mouse input to be sent to the current **CWnd** object regardless of the position of the cursor.

```
CWnd* SetCapture( );
```

Return Value

A pointer to the window object that previously received all mouse input. It is **NULL** if there is no such window. The returned pointer may be temporary and should not be stored for later use.

Remarks

When **CWnd** no longer requires all mouse input, the application should call the ReleaseCapture function so that other windows can receive mouse input.

See Also

Reference

CWnd Class
Hierarchy Chart
ReleaseCapture
SetCapture
CWnd::GetCapture

Concepts

CWnd Members

CWnd::SetCaretPos

Sets the position of the caret.

```
static void PASCAL SetCaretPos(
   POINT point
);
```

Parameters

point

Specifies the new x and y coordinates (in client coordinates) of the caret.

Remarks

The **SetCaretPos** member function moves the caret only if it is owned by a window in the current task. **SetCaretPos** moves the caret whether or not the caret is hidden.

The caret is a shared resource. A window should not move the caret if it does not own the caret.

Example

```
// The following code snippet shows a caret when the left
// mouse button is pressed, and sets the caret's positon to
// the cursor's position.

void CMyView::OnLButtonDown(UINT nFlags, CPoint point)
{
   //create a solid caret, the width is 2, the length is 20.
   CreateSolidCaret (2, 20);

   SetCaretPos (point);
   ShowCaret ();

   CView::OnLButtonDown(nFlags, point);
}
```

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::GetCaretPos

SetCaretPos

Concepts

CWnd Members

CWnd::SetClipboardViewer

Adds this window to the chain of windows that are notified (by means of the **WM_DRAWCLIPBOARD** message) whenever the content of the Clipboard is changed.

```
HWND SetClipboardViewer( );
```

Return Value

A handle to the next window in the Clipboard-viewer chain if successful. Applications should save this handle (it can be stored as a member variable) and use it when responding to Clipboard-viewer chain messages.

Remarks

A window that is part of the Clipboard-viewer chain must respond to WM_DRAWCLIPBOARD, WM_CHANGECBCHAIN, and WM_DESTROY messages and pass the message to the next window in the chain.

This member function sends a **WM_DRAWCLIPBOARD** message to the window. Since the handle to the next window in the Clipboard-viewer chain has not yet been returned, the application should not pass on the **WM_DRAWCLIPBOARD** message that it receives during the call to **SetClipboardViewer**.

To remove itself from the Clipboard-viewer chain, an application must call the ChangeClipboardChain member function.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::ChangeClipboardChain
SetClipboardViewer

Concepts

CWnd Members

CWnd::SetDlgCtrlID

Sets the window ID or control ID for the window to a new value.

```
int SetDlgCtrlID(
   int nID
);
```

Parameters

nID

The new value to set for the control's identifier.

Return Value

The previous identifier of the window, if successful; otherwise 0.

Remarks

The window can be any child window, not only a control in a dialog box. The window cannot be a top-level window.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetDlgCtrlID
CWnd::Create
CWnd::CreateEx
CWnd::GetDlgItem

Concepts

CWnd Members

CWnd::SetDlgItemInt

Sets the text of a given control in a dialog box to the string representation of a specified integer value.

```
void SetDlgItemInt(
   int nID,
   UINT nValue,
   BOOL bSigned = TRUE
);
```

Parameters

nID

Specifies the integer ID of the control to be changed.

nValue

Specifies the integer value used to generate the item text.

bSigned

Specifies whether the integer value is signed or unsigned. If this parameter is **TRUE**, *nValue* is signed. If this parameter is **TRUE** and *nValue* is less than 0, a minus sign is placed before the first digit in the string. If this parameter is **FALSE**, *nValue* is unsigned.

Remarks

SetDlgItemInt sends a WM_SETTEXT message to the given control.

Example

See the example for CWnd::SetDlgItemText.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetDlgItemInt
SetDlgItemInt
WM_SETTEXT

Concepts

CWnd Members

CWnd::SetDlgItemText

Sets the caption or text of a control owned by a window or dialog box.

```
void SetDlgItemText(
   int nID,
   LPCTSTR lpszString
);
```

Parameters

nID

Identifies the control whose text is to be set.

lpszString

Points to a CString object or null-terminated string that contains the text to be copied to the control.

Remarks

SetDlgItemText sends a WM_SETTEXT message to the given control.

Example

```
BOOL CDataDlg::OnInitDialog()
{
   CDialog::OnInitDialog();

   // Initialize dialog controls
   SetDlgItemText(IDC_EDIT_NAME,"Type in text");
   SetDlgItemInt(IDC_EDIT_NUM, 100);
   return TRUE;  // return TRUE unless you set the focus to a control
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
SetDlgItemText
WM_SETTEXT
CWnd::GetDlgItemText

Concepts

CWnd Members

CWnd::SetForegroundWindow

Puts the thread that created the window into the foreground and activates the window.

```
BOOL SetForegroundWindow(  );
```

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

Keyboard input is directed to the window, and various visual cues are changed for the user. The foreground window is the window with which the user is currently working. The foreground window applies only to top-level windows (frame windows or dialogs boxes).

Example

See the example for CWnd::FindWindow.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetForegroundWindow

Concepts

CWnd Members

CWnd::SetFocus

Claims the input focus.

```
CWnd* SetFocus( );
```

Return Value

A pointer to the window object that previously had the input focus. It is **NULL** if there is no such window. The returned pointer may be temporary and should not be stored.

Remarks

The input focus directs all subsequent keyboard input to this window. Any window that previously had the input focus loses it.

The **SetFocus** member function sends a WM_KILLFOCUS message to the window that loses the input focus and a WM_SETFOCUS message to the window that receives the input focus. It also activates either the window or its parent.

If the current window is active but does not have the focus (that is, no window has the focus), any key pressed will produce the messages WM_SYSCHAR, WM_SYSKEYDOWN, or WM_SYSKEYUP.

See Also

Reference

CWnd Class
Hierarchy Chart
SetFocus
CWnd::GetFocus

Concepts

CWnd Members

CWnd::SetFont

Sets the window's current font to the specified font.

```
void SetFont(
   CFont* pFont,
   BOOL bRedraw = TRUE
);
```

Parameters

pFont

Specifies the new font.

bRedraw

If **TRUE**, redraw the **CWnd** object.

Remarks

If *bRedraw* is **TRUE**, the window will also be redrawn.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetFont
WM_SETFONT

Concepts

CWnd Members

CWnd::SetIcon

Call this member function to set the handle to a specific icon, as identified by *hIcon*.

```
HICON SetIcon(
   HICON hIcon,
   BOOL bBigIcon
);
```

Parameters

hIcon

A handle to a previous icon.

bBigIcon

Specifies a 32 pixel by 32 pixel icon if **TRUE**; specifies a 16 pixel by 16 pixel icon if **FALSE**.

Return Value

A handle to an icon.

Remarks

When the window class is registered, it selects an icon.

Example

See the example for CWnd::GetSystemMenu.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetIcon

Concepts

CWnd Members

CWnd::SetLayeredWindowAttributes

Sets the opacity and transparency color key of a layered window.

```
BOOL SetLayeredWindowAttributes(
   COLORREF crKey,
   BYTE bAlpha,
   DWORD dwFlags
);
```

Parameters

crKey

Pointer to a **COLORREF** value that specifies the transparency color key to be used when composing the layered window. All pixels painted by the window in this color will be transparent. To generate a **COLORREF**, use the RGB macro.

bAlpha

Alpha value used to describe the opacity of the layered window. For more information, see the **SourceConstantAlpha** member of the BLENDFUNCTION structure. When *bAlpha* is 0, the window is completely transparent. When *bAlpha* is 255, the window is opaque.

dwFlags

Specifies an action to take. This parameter can be one or more of the following values. For a list of possible values, seeSetLayeredWindowAttributes.

Return Value

Nonzero if the function succeeds; otherwise 0.

Remarks

This member function emulates the functionality of the function SetLayeredWindowAttributes, as described in the Platform SDK.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::UpdateLayeredWindow

Concepts

CWnd Members

CWnd::SetMenu

Sets the current menu to the specified menu.

```
BOOL SetMenu(
   CMenu* pMenu
);
```

Parameters

pMenu

Identifies the new menu. If this parameter is **NULL**, the current menu is removed.

Return Value

Nonzero if the menu is changed; otherwise 0.

Remarks

Causes the window to be redrawn to reflect the menu change.

SetMenu will not destroy a previous menu. An application should call the CMenu::DestroyMenu member function to accomplish this task.

Example

See the example for CMenu::LoadMenu.

See Also

Reference

CWnd Class
Hierarchy Chart
CMenu::DestroyMenu
CMenu::LoadMenu
SetMenu
CWnd::GetMenu

Concepts

CWnd Members

CWnd::SetOwner

Sets the current window's owner to the specified window object.

```
void SetOwner(
   CWnd* pOwnerWnd
);
```

Parameters

pOwnerWnd

Identifies the new owner of the window object. If this parameter is **NULL**, the window object has no owner.

Remarks

This owner can then receive command messages from the current window object. By default, the parent of the current window is its owner.

It is often useful to establish connections between window objects that are unrelated to the window hierarchy. For example, CToolBar sends notifications to its owner instead of to its parent. This allows the toolbar to become the child of one window (such as an OLE container application window) while sending notifications to another window (such as the in-place frame window). Furthermore, when a server window is deactivated or activated during in-place editing, any window owned by the frame window is hidden or shown. This ownership is explicitly set with a call to **SetOwner**.

The ownership concept of this function is different from the ownership concept of GetWindow.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetOwner
CToolBar Class

Concepts

CWnd Members

CWnd::SetParent

Changes the parent window of a child window.

```
CWnd* SetParent(
   CWnd* pWndNewParent
);
```

Parameters

pWndNewParent

Identifies the new parent window.

Return Value

A pointer to the previous parent window object if successful. The returned pointer may be temporary and should not be stored for later use.

Remarks

If the child window is visible, Windows performs the appropriate redrawing and repainting.

See Also

Reference

CWnd Class
Hierarchy Chart
SetParent
CWnd::GetParent

Concepts

CWnd Members

CWnd::SetProperty

Call this member function to set the OLE control property specified by *dwDispID*.

```
void AFX_CDECL SetProperty(
   DISPID dwDispID,
   VARTYPE vtProp,
      ...
);
```

Parameters

dwDispID

Identifies the property to be set.

vtProp

Specifies the type of the property to be set. For possible values, see the Remarks section for COleDispatchDriver::InvokeHelper.

...

A single parameter of the type specified by *vtProp*.

Remarks

Note
This function should be called only on a **CWnd** object that represents an OLE control.

For more information about using this member function with OLE Control Containers, see the article ActiveX Control Containers: Programming ActiveX Controls in an ActiveX Control Container.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::InvokeHelper
COleDispatchDriver Class
CWnd::CreateControl

Concepts

CWnd Members

CWnd::SetRedraw

An application calls **SetRedraw** to allow changes to be redrawn or to prevent changes from being redrawn.

```
void SetRedraw(
   BOOL bRedraw = TRUE
);
```

Parameters

bRedraw

Specifies the state of the redraw flag. If this parameter is **TRUE**, the redraw flag is set; if **FALSE**, the flag is cleared.

Remarks

This member function sets or clears the redraw flag. While the redraw flag is cleared, the contents will not be updated after each change and will not be repainted until the redraw flag is set. For example, an application that needs to add several items to a list box can clear the redraw flag, add the items, and then set the redraw flag. Finally, the application can call the Invalidate or InvalidateRect member function to cause the list box to be repainted.

Example

```
// Updating a control or window with large amounts of data may cause
// flicker. In such cases it may be better to turn off drawing

//...

   //m_list is a member of type CListCtrl
   m_List.SetRedraw(FALSE);  // turn drawing off regardless of list mode

//...
// Update control
//...

   m_List.SetRedraw(TRUE);  // turn drawing back on and update the window

   // invalidate the entire control, force painting
   m_List.Invalidate();
   m_List.UpdateWindow();
```

See Also

Reference

CWnd Class
Hierarchy Chart
WM_SETREDRAW

Concepts

CWnd Members

CWnd::SetScrollInfo

Call this member function to set the information that the **SCROLLINFO** structure maintains about a scroll bar.

```
BOOL SetScrollInfo(
   int nBar,
   LPSCROLLINFO lpScrollInfo,
   BOOL bRedraw = TRUE
);
```

Parameters

nBar

Specifies whether the scroll bar is a control or part of a window's nonclient area. If it is part of the nonclient area, nBar also indicates whether the scroll bar is positioned horizontally, vertically, or both. It must be one of the following:

- **SB_CTL** Contains the parameters for a scroll bar control. The **m_hWnd** data member must be the handle of the scroll bar control.
- **SB_HORZ** Specifies that the window is a horizontal scroll bar.
- **SB_VERT** Specifies that the window is a vertical scroll bar.

lpScrollInfo

A pointer to a SCROLLINFO structure. See the *Platform SDK* for more information about this structure.

bRedraw

Specifies whether the scroll bar should be redrawn to reflect the new position. If *bRedraw* is **TRUE**, the scroll bar is redrawn. If it is **FALSE**, it is not redrawn. The scroll bar is redrawn by default.

Return Value

If successful, the return is **TRUE**. Otherwise, it is **FALSE**.

Remarks

The SCROLLINFO structure contains information about a scroll bar, including the minimum and maximum scrolling positions, the page size, and the position of the scroll box (the thumb). See the **SCROLLINFO** structure topic in the *Platform SDK* for more information about changing the structure defaults.

The MFC Windows message handlers that indicate scroll-bar position, CWnd::OnHScroll and CWnd::OnVScroll, provide only 16 bits of position data. GetScrollInfo and **SetScrollInfo** provide 32 bits of scroll-bar position data. Thus, an application can call **GetScrollInfo** while processing either **CWnd::OnHScroll** or **CWnd::OnVScroll** to obtain 32-bit scroll-bar position data.

Note
CWnd::GetScrollInfo enables applications to use 32-bit scroll-bar positions.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetScrollInfo
CWnd::SetScrollPos
CWnd::OnVScroll
CWnd::OnHScroll
SCROLLINFO

Concepts

CWnd Members

CWnd::SetScrollPos

Sets the current position of a scroll box and, if requested, redraws the scroll bar to reflect the new position of the scroll box.

```
int SetScrollPos(
   int nBar,
   int nPos,
   BOOL bRedraw = TRUE
);
```

Parameters

nBar

Specifies the scroll bar to be set. This parameter can be either of the following:

- **SB_HORZ** Sets the position of the scroll box in the horizontal scroll bar of the window.
- **SB_VERT** Sets the position of the scroll box in the vertical scroll bar of the window.

nPos

Specifies the new position of the scroll box. It must be within the scrolling range.

bRedraw

Specifies whether the scroll bar should be repainted to reflect the new scroll-box position. If this parameter is **TRUE**, the scroll bar is repainted; if **FALSE**, the scroll bar is not repainted.

Return Value

The previous position of the scroll box.

Remarks

Setting *bRedraw* to **FALSE** is useful whenever the scroll bar will be redrawn by a subsequent call to another function.

See Also

Reference

CWnd Class
Hierarchy Chart
SetScrollPos
CWnd::GetScrollPos
CScrollBar::SetScrollPos

Concepts

CWnd Members

CWnd::SetScrollRange

Sets minimum and maximum position values for the given scroll bar.

```
void SetScrollRange(
   int nBar,
   int nMinPos,
   int nMaxPos,
   BOOL bRedraw = TRUE
);
```

Parameters

nBar

Specifies the scroll bar to be set. This parameter can be either of the following values:

- **SB_HORZ** Sets the range of the horizontal scroll bar of the window.
- **SB_VERT** Sets the range of the vertical scroll bar of the window.

nMinPos

Specifies the minimum scrolling position.

nMaxPos

Specifies the maximum scrolling position.

bRedraw

Specifies whether the scroll bar should be redrawn to reflect the change. If *bRedraw* is **TRUE**, the scroll bar is redrawn; if **FALSE**, the scroll bar is not redrawn.

Remarks

It can also be used to hide or show standard scroll bars.

An application should not call this function to hide a scroll bar while processing a scroll-bar notification message.

If the call to **SetScrollRange** immediately follows a call to the SetScrollPos member function, the *bRedraw* parameter in the **SetScrollPos** member function should be 0 to prevent the scroll bar from being drawn twice.

The default range for a standard scroll bar is 0 through 100. The default range for a scroll bar control is empty (both the *nMinPos* and *nMaxPos* values are 0). The difference between the values specified by *nMinPos* and *nMaxPos* must not be greater than **INT_MAX**.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::SetScrollPos
SetScrollRange
CWnd::GetScrollRange

Concepts

CWnd Members

CWnd::SetTimer

Installs a system timer.

```
UINT_PTR SetTimer(
   UINT_PTR nIDEvent,
   UINT nElapse,
   void (CALLBACK* lpfnTimer
)(HWND,
   UINT,
   UINT_PTR,
   DWORD
)
);
```

Parameters

nIDEvent

Specifies a nonzero timer identifier.

nElapse

Specifies the time-out value, in milliseconds.

lpfnTimer

Specifies the address of the application-supplied `TimerProc` callback function that processes the **WM_TIMER** messages. If this parameter is **NULL**, the **WM_TIMER** messages are placed in the application's message queue and handled by the **CWnd** object.

Return Value

The timer identifier of the new timer if the function is successful. An application passes this value to the KillTimer member function to kill the timer. Nonzero if successful; otherwise 0.

Remarks

A time-out value is specified, and every time a time-out occurs, the system posts a WM_TIMER message to the installing application's message queue or passes the message to an application-defined **TimerProc** callback function.

The *lpfnTimer* callback function need not be named **TimerProc**, but it must be defined as follows:

```
void CALLBACK EXPORT TimerProc(
   HWND hWnd,      // handle of CWnd that called SetTimer
   UINT nMsg,      // WM_TIMER
   UINT nIDEvent   // timer identification
   DWORD dwTime    // system time
);
```

Example

This example uses **CWnd::SetTimer, CWnd::OnTimer**, and **CWnd::KillTimer** to handle **WM_TIMER** messages. A timer is set up to send a **WM_TIMER** message to the main frame window every 2 seconds in `OnStartTimer`. `OnStopTimer` will stop the timer by calling `CWnd::KillTimer`. `OnTimer` was set up to handle **WM_TIMER** messages for the main frame window. In this example, the PC speaker will beep every 2 seconds.

```
void CMainFrame::OnStartTimer()
{
   m_nTimer = SetTimer(1, 2000, 0);
}

void CMainFrame::OnStopTimer()
{
   KillTimer(m_nTimer);
```

```
}

void CMainFrame::OnTimer(UINT nIDEvent)
{
   MessageBeep(0xFFFFFFFF);   // Beep

   // Call base class handler.
   CMDIFrameWnd::OnTimer(nIDEvent);
}
```

See Also

Reference

CWnd Class

Hierarchy Chart

WM_TIMER

CWnd::KillTimer

SetTimer

Concepts

CWnd Members

CWnd::SetWindowContextHelpId

Call this member function to associate a help context identifier with the specified window.

```
BOOL SetWindowContextHelpId(
   DWORD dwContextHelpId
);
```

Parameters

dwContextHelpId

The help context identifier.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

If a child window does not have a help context identifier, it inherits the identifier of its parent window. Likewise, if an owned window does not have a help context identifier, it inherits the identifier of its owner window. This inheritance of help context identifiers allows an application to set just one identifier for a dialog box and all of its controls.

Example

```
BOOL CMyDialog::OnInitDialog()
   {
      CDialog::OnInitDialog();

      //...
      // Associate a help context id with the control.
      // IDC_TESTHELP_CONTROL is the id of the control
      // and HIDC_TESTHELP_CONTROL is its help context
      // id associated with the control.
      CWnd* pWnd = GetDlgItem(IDC_TESTHELP_CONTROL);
      pWndl->SetWindowContextHelpId(HIDC_TESTHELP_CONTROL);

      return TRUE;
      // Return TRUE unless you set the focus to a control.
      // EXCEPTION: OCX Property Pages should return FALSE.
   }
```

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::GetWindowContextHelpId

Concepts

CWnd Members

CWnd::SetWindowPlacement

Sets the show state and the normal (restored), minimized, and maximized positions for a window.

```
BOOL SetWindowPlacement(
   const WINDOWPLACEMENT*lpwndpl
);
```

Parameters

lpwndpl

Points to a WINDOWPLACEMENT structure that specifies the new show state and positions.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetWindowPlacement
SetWindowPlacement

Concepts

CWnd Members

CWnd::SetWindowPos

Call this member function to change the size, position, and Z-order of child, pop-up, and top-level windows.

```
BOOL SetWindowPos(
   const CWnd* pWndInsertAfter,
   int x,
   int y,
   int cx,
   int cy,
   UINT nFlags
);
```

Parameters

pWndInsertAfter

Identifies the **CWnd** object that will precede this **CWnd** object in the Z-order. This parameter can be a pointer to a **CWnd** or a **Pointer** to one of the following values:

- **wndBottom** Places the window at the bottom of the Z-order. If this **CWnd** is a topmost window, the window loses its topmost status; the system places the window at the bottom of all other windows.
- **wndTop** Places the window at the top of the Z-order.
- **wndTopMost** Places the window above all nontopmost windows. The window maintains its topmost position even when it is deactivated.
- **wndNoTopMost** Repositions the window to the top of all nontopmost windows (that is, behind all topmost windows). This flag has no effect if the window is already a nontopmost window.

See the "Remarks" section for this function for rules about how this parameter is used.

x

Specifies the new position of the left side of the window.

y

Specifies the new position of the top of the window.

cx

Specifies the new width of the window.

cy

Specifies the new height of the window.

nFlags

Specifies sizing and positioning options. This parameter can be a combination of the following:

- **SWP_DRAWFRAME** Draws a frame (defined when the window was created) around the window.
- **SWP_FRAMECHANGED** Sends a **WM_NCCALCSIZE** message to the window, even if the window's size is not being changed. If this flag is not specified, **WM_NCCALCSIZE** is sent only when the window's size is being changed.
- **SWP_HIDEWINDOW** Hides the window.
- **SWP_NOACTIVATE** Does not activate the window. If this flag is not set, the window is activated and moved to the top of either the topmost or the nontopmost group (depending on the setting of the *pWndInsertAfter* parameter).
- **SWP_NOCOPYBITS** Discards the entire contents of the client area. If this flag is not specified, the valid contents of the client area are saved and copied back into the client area after the window is sized or repositioned.
- **SWP_NOMOVE** Retains current position (ignores the *x* and *y* parameters).
- **SWP_NOOWNERZORDER** Does not change the owner window's position in the Z-order.

- **SWP_NOREDRAW** Does not redraw changes. If this flag is set, no repainting of any kind occurs. This applies to the client area, the nonclient area (including the title and scroll bars), and any part of the parent window uncovered as a result of the moved window. When this flag is set, the application must explicitly invalidate or redraw any parts of the window and parent window that must be redrawn.
- **SWP_NOREPOSITION** Same as **SWP_NOOWNERZORDER**.
- **SWP_NOSENDCHANGING** Prevents the window from receiving the **WM_WINDOWPOSCHANGING** message.
- **SWP_NOSIZE** Retains current size (ignores the *cx* and *cy* parameters).
- **SWP_NOZORDER** Retains current ordering (ignores *pWndInsertAfter*).
- **SWP_SHOWWINDOW** Displays the window.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

Windows are ordered on the screen according to their Z-order; the window at the top of the Z-order appears on top of all other windows in the order.

All coordinates for child windows are client coordinates (relative to the upper-left corner of the parent window's client area).

A window can be moved to the top of the Z-order either by setting the *pWndInsertAfter* parameter to **&wndTopMost** and ensuring that the **SWP_NOZORDER** flag is not set or by setting a window's Z-order so that it is above any existing topmost windows. When a nontopmost window is made topmost, its owned windows are also made topmost. Its owners are not changed.

A topmost window is no longer topmost if it is repositioned to the bottom (**&wndBottom**) of the Z-order or after any nontopmost window. When a topmost window is made nontopmost, all of its owners and its owned windows are also made nontopmost windows.

If neither **SWP_NOACTIVATE** nor **SWP_NOZORDER** is specified (that is, when the application requests that a window be simultaneously activated and placed in the specified Z-order), the value specified in *pWndInsertAfter* is used only in the following circumstances:

- Neither **&wndTopMost** nor **&wndNoTopMost** is specified in the *pWndInsertAfter* parameter.
- This window is not the active window.

An application cannot activate an inactive window without also bringing it to the top of the Z-order. Applications can change the Z-order of an activated window without restrictions.

A nontopmost window may own a topmost window, but not vice versa. Any window (for example, a dialog box) owned by a topmost window is itself made a topmost window to ensure that all owned windows stay above their owner.

With Windows versions 3.1 and later, windows can be moved to the top of the Z-order and locked there by setting their **WS_EX_TOPMOST** styles. Such a topmost window maintains its topmost position even when deactivated. For example, selecting the WinHelp Always On Top command makes the Help window topmost, and it then remains visible when you return to your application.

To create a topmost window, call **SetWindowPos** with the *pWndInsertAfter* parameter equal to **&wndTopMost**, or set the **WS_EX_TOPMOST** style when you create the window.

If the Z-order contains any windows with the **WS_EX_TOPMOST** style, a window moved with the **&wndTopMost** value is placed at the top of all nontopmost windows, but below any topmost windows. When an application activates an inactive window without the **WS_EX_TOPMOST** bit, the window is moved above all nontopmost windows but below any topmost windows.

If **SetWindowPos** is called when the *pWndInsertAfter* parameter is **&wndBottom** and **CWnd** is a topmost window, the window loses its topmost status (**WS_EX_TOPMOST** is cleared), and the system places the window at the bottom of the Z-order.

Example

```
void CWinApp::HideApplication()
{
   //m_pMainWnd is the main application window, a member of CWinApp
```

```
    ASSERT_VALID(m_pMainWnd);

    // hide the application's windows before closing all the documents
    m_pMainWnd->ShowWindow(SW_HIDE);
    m_pMainWnd->ShowOwnedPopups(FALSE);

    // put the window at the bottom of z-order, so it isn't activated
    m_pMainWnd->SetWindowPos(&CWnd::wndBottom, 0, 0, 0, 0,
       SWP_NOMOVE|SWP_NOSIZE|SWP_NOACTIVATE);
}
```

See Also

Reference

CWnd Class

Hierarchy Chart

DeferWindowPos

SetWindowPos

Concepts

CWnd Members

CWnd::SetWindowRgn

Call this member function to set a window's region.

```
int SetWindowRgn(
   HRGN hRgn,
   BOOL bRedraw
);
```

Parameters

hRgn

A handle to a region.

bRedraw

If **TRUE**, the operating system redraws the window after setting the region; otherwise, it does not. Typically, set *bRedraw* to **TRUE** if the window is visible. If set to **TRUE**, the system sends the **WM_WINDOWPOSCHANGING** and **WM_WINDOWPOSCHANGED** messages to the window.

Return Value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

Remarks

The coordinates of a window's window region are relative to the upper-left corner of the window, not the client area of the window.

After a successful call to **SetWindowRgn**, the operating system owns the region specified by the region handle *hRgn*. The operating system does not make a copy of the region, so do not make any further function calls with this region handle, and do not close this region handle.

See Also

Reference

CWnd Class
Hierarchy Chart
SetWindowRgn
CWnd::GetWindowRgn

Concepts

CWnd Members

CWnd::SetWindowText

Sets the window's title to the specified text.

```
void SetWindowText(
   LPCTSTR lpszString
);
```

Parameters

lpszString

Points to a CString object or null-terminated string to be used as the new title or control text.

Remarks

If the window is a control, the text within the control is set.

This function causes a WM_SETTEXT message to be sent to this window.

Example

```
// set the text in IDC_MYEDIT
CWnd* pWnd = GetDlgItem(IDC_MYEDIT);
pWnd->SetWindowText(_T("Hockey is best!"));

// Get the text back. CString is convenient, because MFC
// will automatically allocate enough memory to hold the
// text--no matter how large it is.

CString str;
pWnd->GetWindowText(str);
ASSERT(str == _T("Hockey is best!"));

// The LPTSTR override works, too, but it might be too short.
// If we supply a buffer that's too small, we'll only get those
// characters that fit.

TCHAR sz[10];
int nRet = pWnd->GetWindowText(sz, 10);

// Nine characters, plus terminating null
ASSERT(lstrcmp(sz, _T("Hockey is")) == 0);
ASSERT(nRet == 9);

// You can query the length of the text without the length of
// the string using CWnd::GetWindowTextLength()
nRet = pWnd->GetWindowTextLength();
ASSERT(nRet == 15);
```

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::GetWindowText

SetWindowText

Concepts

CWnd Members

CWnd::ShowCaret

Shows the caret on the screen at the caret's current position.

```
void ShowCaret( );
```

Remarks

Once shown, the caret begins flashing automatically.

The **ShowCaret** member function shows the caret only if it has a current shape and has not been hidden two or more times consecutively. If the caret is not owned by this window, the caret is not shown.

Hiding the caret is cumulative. If the HideCaret member function has been called five times consecutively, **ShowCaret** must be called five times to show the caret.

The caret is a shared resource. The window should show the caret only when it has the input focus or is active.

Example

See the example for CWnd::CreateCaret.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::HideCaret
ShowCaret

Concepts

CWnd Members

CWnd::ShowOwnedPopups

Shows or hides all pop-up windows owned by this window.

```
void ShowOwnedPopups(
   BOOL bShow = TRUE
);
```

Parameters

bShow

Specifies whether pop-up windows are to be shown or hidden. If this parameter is **TRUE**, all hidden pop-up windows are shown. If this parameter is **FALSE**, all visible pop-up windows are hidden.

Example

See the example for CWnd::SetWindowPos.

See Also

Reference

CWnd Class
Hierarchy Chart
ShowOwnedPopups

Concepts

CWnd Members

CWnd::ShowScrollBar

Shows or hides a scroll bar.

```
void ShowScrollBar(
   UINT nBar,
   BOOL bShow = TRUE
);
```

Parameters

nBar

Specifies whether the scroll bar is a control or part of a window's nonclient area. If it is part of the nonclient area, *nBar* also indicates whether the scroll bar is positioned horizontally, vertically, or both. It must be one of the following:

- **SB_BOTH** Specifies the horizontal and vertical scroll bars of the window.
- **SB_HORZ** Specifies that the window is a horizontal scroll bar.
- **SB_VERT** Specifies that the window is a vertical scroll bar.

bShow

Specifies whether Windows shows or hides the scroll bar. If this parameter is **TRUE**, the scroll bar is shown; otherwise the scroll bar is hidden.

Remarks

An application should not call **ShowScrollBar** to hide a scroll bar while processing a scroll-bar notification message.

See Also

Reference

CWnd Class
Hierarchy Chart
ShowScrollBar
CScrollBar::ShowScrollBar

Concepts

CWnd Members

CWnd::ShowWindow

Sets the visibility state of the window.

```
BOOL ShowWindow(
   int nCmdShow
);
```

Parameters

nCmdShow

Specifies how the **CWnd** is to be shown. It must be one of the following values:

- **SW_HIDE** Hides this window and passes activation to another window.
- **SW_MINIMIZE** Minimizes the window and activates the top-level window in the system's list.
- **SW_RESTORE** Activates and displays the window. If the window is minimized or maximized, Windows restores it to its original size and position.
- **SW_SHOW** Activates the window and displays it in its current size and position.
- **SW_SHOWMAXIMIZED** Activates the window and displays it as a maximized window.
- **SW_SHOWMINIMIZED** Activates the window and displays it as an icon.
- **SW_SHOWMINNOACTIVE** Displays the window as an icon. The window that is currently active remains active.
- **SW_SHOWNA** Displays the window in its current state. The window that is currently active remains active.
- **SW_SHOWNOACTIVATE** Displays the window in its most recent size and position. The window that is currently active remains active.
- **SW_SHOWNORMAL** Activates and displays the window. If the window is minimized or maximized, Windows restores it to its original size and position.

Return Value

Nonzero if the window was previously visible; 0 if the **CWnd** was previously hidden.

Remarks

ShowWindow must be called only once per application for the main window with CWinApp::m_nCmdShow. Subsequent calls to **ShowWindow** must use one of the values listed above instead of the one specified by **CWinApp::m_nCmdShow**.

Example

See the example for CWnd::CalcWindowRect.

See Also

Reference

CWnd Class
Hierarchy Chart
ShowWindow
CWnd::OnShowWindow
CWnd::ShowOwnedPopups

Concepts

CWnd Members

CWnd::SubclassDlgItem

Call this member function to "dynamically subclass" a control created from a dialog template and attach it to this **CWnd** object.

```
BOOL SubclassDlgItem(
   UINT nID,
   CWnd* pParent
);
```

Parameters

nID

The control's ID.

pParent

The control's parent (usually a dialog box).

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

When a control is dynamically subclassed, windows messages will route through the **CWnd**'s message map and call message handlers in the **CWnd**'s class first. Messages that are passed to the base class will be passed to the default message handler in the control.

This member function attaches the Windows control to a **CWnd** object and replaces the control's **WndProc** and **AfxWndProc** functions. The function stores the old **WndProc** in the location returned by the GetSuperWndProcAddr member function.

Example

```
class CMyButton : public CButton {...};
// m_myButton is a CMyButton object member of CAboutDlg

BOOL CAboutDlg::OnInitDialog()
{
   CDialog::OnInitDialog();
   // IDC_BUTTON1 is the ID for a button on the
   // dialog template used for CAboutDlg.
   m_myButton.SubclassDlgItem(IDC_BUTTON1, this);

   return TRUE;   // Return TRUE unless you set the focus to a control
                  // EXCEPTION: OCX Property Pages should return FALSE
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetSuperWndProcAddr
CWnd::DefWindowProc
CWnd::SubclassWindow
CWnd::Attach

Concepts

CWnd Members

CWnd::SubclassWindow

Call this member function to "dynamically subclass" a window and attach it to this **CWnd** object.

```
BOOL SubclassWindow(
   HWND hWnd
);
```

Parameters

hWnd

A handle to the window.

Return Value

Nonzero if the function is successful; otherwise 0.

Remarks

When a window is dynamically subclassed, windows messages will route through the **CWnd**'s message map and call message handlers in the **CWnd**'s class first. Messages that are passed to the base class will be passed to the default message handler in the window.

This member function attaches the Windows control to a **CWnd** object and replaces the window's **WndProc** and **AfxWndProc** functions. The function stores a pointer to the old **WndProc** in the **CWnd** object.

Note
The window must not already be attached to an MFC object when this function is called.

Example

```
// The following code shows how to subclass the edit control and list box
// controls inside a combo box. It uses WM_CTLCOLOR for subclassing.
// CSuperComboBox represents the combo box
HBRUSH CSuperComboBox::OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor)
{
   if (nCtlColor == CTLCOLOR_EDIT)
   {
      //Edit control
      if (m_edit.GetSafeHwnd() == NULL)
         m_edit.SubclassWindow(pWnd->GetSafeHwnd());
   }
   else if (nCtlColor == CTLCOLOR_LISTBOX)
   {
      //ListBox control
      if (m_listbox.GetSafeHwnd() == NULL)
         m_listbox.SubclassWindow(pWnd->GetSafeHwnd());
   }
   HBRUSH hbr = CComboBox::OnCtlColor(pDC, pWnd, nCtlColor);
   return hbr;
}

void CSuperComboBox::OnDestroy()
{
   //unsubclass edit and list box before destruction
   if (m_edit.GetSafeHwnd() != NULL)
      m_edit.UnsubclassWindow();
   if (m_listbox.GetSafeHwnd() != NULL)
      m_listbox.UnsubclassWindow();
   CComboBox::OnDestroy();
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::GetSuperWndProcAddr
CWnd::DefWindowProc
CWnd::SubclassDlgItem
CWnd::Attach
CWnd::PreSubclassWindow
CWnd::UnsubclassWindow

Concepts

CWnd Members

CWnd::UnlockWindowUpdate

Call this member function to unlock a window that was locked with **CWnd::LockWindowUpdate**.

```
void UnlockWindowUpdate();
```

Remarks

Only one window at a time can be locked using **LockWindowUpdate**. See CWnd::LockWindowUpdate or the Win32 function LockWindowUpdate for more information on locking windows.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::UnsubclassWindow

Call this member function to set **WndProc** back to its original value and detach the window identified by **HWND** from the **CWnd** object.

```
HWND UnsubclassWindow( );
```

Return Value

A handle to the unsubclassed window.

Example

See the example for CWnd::SubclassWindow.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::SubclassWindow
CWnd::PreSubclassWindow
CWnd::GetSuperWndProcAddr
CWnd::DefWindowProc
CWnd::SubclassDlgItem
CWnd::Attach

Concepts

CWnd Members

CWnd::UpdateData

Call this member function to initialize data in a dialog box, or to retrieve and validate dialog data.

```
BOOL UpdateData(
   BOOL bSaveAndValidate = TRUE
);
```

Parameters

bSaveAndValidate

Flag that indicates whether dialog box is being initialized (**FALSE**) or data is being retrieved (**TRUE**).

Return Value

Nonzero if the operation is successful; otherwise 0. If *bSaveAndValidate* is **TRUE**, then a return value of nonzero means that the data is successfully validated.

Remarks

The framework automatically calls **UpdateData** with *bSaveAndValidate* set to **FALSE** when a modal dialog box is created in the default implementation of CDialog::OnInitDialog. The call occurs before the dialog box is visible. The default implementation of CDialog::OnOK calls this member function with *bSaveAndValidate* set to **TRUE** to retrieve the data, and if successful, will close the dialog box. (If the Cancel button is clicked in the dialog box, the dialog box is closed without the data being retrieved.)

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::DoDataExchange

Concepts

CWnd Members

CWnd::UpdateDialogControls

Call this member function to update the state of dialog buttons and other controls in a dialog box or window that uses the ON_UPDATE_COMMAND_UI callback mechanism.

```
void UpdateDialogControls(
   CCmdTarget* pTarget,
   BOOL bDisableIfNoHndler
);
```

Parameters

pTarget

Points to the main frame window of the application, and is used for routing update messages.

bDisableIfNoHndler

Flag that indicates whether a control that has no update handler should be automatically displayed as disabled.

Remarks

If a child control does not have a handler and *bDisableIfNoHndler* is **TRUE**, then the child control will be disabled.

The framework calls this member function for controls in dialog bars or toolbars as part of the application's idle processing.

See Also

Reference

CWnd Class
Hierarchy Chart
CFrameWnd::m_bAutoMenuEnable

Concepts

CWnd Members

CWnd::UpdateLayeredWindow

Updates the position, size, shape, content, and translucency of a layered window.

```
BOOL UpdateLayeredWindow(
   CDC* pDCDst,
   POINT *pptDst,
   SIZE *psize,
   CDC *pDCSrc,
   POINT *pptSrc,
   COLORREF crKey,
   BLENDFUNCTION *pblend,
   DWORD dwFlags
);
```

Parameters

pDCDst

A pointer to a device context for the screen. It is used for palette color matching when the window contents are updated. If *pDCDst* is **NULL**, the default palette will be used.

If *pDCSrc* is **NULL**, *pDCDst* must be **NULL**.

pptDst

A pointer to a **POINT** structure specifying the new screen position of the layered window. If the current position is not changing, *pptDst* can be **NULL**.

psize

Pointer to a **SIZE** structure that specifies the new size of the layered window. If the size of the window is not changing, *psize* can be **NULL**.

If *pDCSrc* is **NULL**, *psize* must be **NULL**.

pDCSrc

A pointer to a DC for the surface that defines the layered window. If the shape and visual context of the window are not changing, *pDCSrc* can be **NULL**.

pptSrc

Pointer to a **POINT** structure that specifies the location of the layer in the device context.

If *pDCSrc* is **NULL**, *pptSrc* should be **NULL**.

crKey

Pointer to a **COLORREF** value that specifies the transparency color key to be used when composing the layered window. All pixels painted by the window in this color will be transparent. To generate a **COLORREF**, use the RGB macro.

pblend

Pointer to a BLENDFUNCTION structure that specifies the transparency value to be used when composing the layered window.

dwFlags

Specifies an action to take. This parameter can be one or more of the following values. For a list of possible values, seeUpdateLayeredWindow.

Return Value

Nonzero if the function succeeds; otherwise 0.

Remarks

This member function emulates the functionality of the function UpdateLayeredWindow, as described in the Platform SDK.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::SetLayeredWindowAttributes

Concepts

CWnd Members

CWnd::UpdateWindow

Updates the client area by sending a WM_PAINT message if the update region is not empty.

```
void UpdateWindow( );
```

Remarks

The **UpdateWindow** member function sends a **WM_PAINT** message directly, bypassing the application queue. If the update region is empty, **WM_PAINT** is not sent.

Example

```
// In this example a rectangle is drawn in a view.
// The OnChangeRect() function changes the dimensions
// of the rectangle and then calls CWnd::Invalidate() so the
// client area of the view will be redrawn next time the
// window is updated.  It then calls CWnd::UpdateWindow
// to force the new rectangle to be painted.

void CTestView::OnChangeRect()
{
   // Change Rectangle size.
   m_rcBox = CRect(20, 20, 210, 210);

   // Invalidate window so entire client area
   // is redrawn when UpdateWindow is called.
   Invalidate();

   // Update Window to cause View to redraw.
   UpdateWindow();
}

// On Draw function draws the rectangle.
void CTestView::OnDraw(CDC* pDC)
{
//  ..  Other draw code here.

   pDC->Draw3dRect(m_rcBox, 0x00FF0000, 0x0000FF00);
}
```

See Also

Reference

CWnd Class
Hierarchy Chart
UpdateWindow
CWnd::RedrawWindow

Concepts

CWnd Members

CWnd::ValidateRect

Validates the client area within the given rectangle by removing the rectangle from the update region of the window.

```
void ValidateRect(
   LPCRECT lpRect
);
```

Parameters

lpRect

Points to a CRect object or RECT structure that contains client coordinates of the rectangle to be removed from the update region. If *lpRect* is **NULL**, the entire window is validated.

Remarks

The BeginPaint member function automatically validates the entire client area. Neither the **ValidateRect** nor the ValidateRgn member function should be called if a portion of the update region needs to be validated before WM_PAINT is next generated.

Windows continues to generate **WM_PAINT** messages until the current update region is validated.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::BeginPaint
ValidateRect
CWnd::ValidateRgn

Concepts

CWnd Members

CWnd::ValidateRgn

Validates the client area within the given region by removing the region from the current update region of the window.

```
void ValidateRgn(
   CRgn* pRgn
);
```

Parameters

pRgn

A pointer to a CRgn object that identifies a region that defines the area to be removed from the update region. If this parameter is **NULL**, the entire client area is removed.

Remarks

The given region must have been created previously by a region function. The region coordinates are assumed to be client coordinates.

The BeginPaint member function automatically validates the entire client area. Neither the ValidateRect nor the **ValidateRgn** member function should be called if a portion of the update region must be validated before the next WM_PAINT message is generated.

See Also

Reference

CWnd Class
Hierarchy Chart
ValidateRgn
CWnd::ValidateRect

Concepts

CWnd Members

CWnd::WindowFromPoint

Retrieves the window that contains the specified point; *point* must specify the screen coordinates of a point on the screen.

```
static CWnd* PASCAL WindowFromPoint(
   POINT point
);
```

Parameters

point

Specifies a CPoint object or POINT data structure that defines the point to be checked.

Return Value

A pointer to the window object in which the point lies. It is **NULL** if no window exists at the given point. The returned pointer may be temporary and should not be stored for later use.

Remarks

WindowFromPoint does not retrieve a hidden or disabled window, even if the point is within the window. An application should use the ChildWindowFromPoint member function for a nonrestrictive search.

See Also

Reference

CWnd Class
Hierarchy Chart
WindowFromPoint
CWnd::ChildWindowFromPoint

Concepts

CWnd Members

CWnd::WindowProc

Provides a Windows procedure (**WindowProc**) for a **CWnd** object.

```
virtual LRESULT WindowProc(
   UINT message,
   WPARAM wParam,
   LPARAM lParam
);
```

Parameters

message

Specifies the Windows message to be processed.

wParam

Provides additional information used in processing the message. The parameter value depends on the message.

lParam

Provides additional information used in processing the message. The parameter value depends on the message.

Return Value

The return value depends on the message.

Remarks

It dispatches messages through the window's message map.

Smart Device Developer Notes

This method is not supported in smart device projects.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

CWnd::WinHelp

Called to initiate the WinHelp application.

```
virtual void WinHelp(
   DWORD_PTR dwData,
   UINT nCmd = HELP_CONTEXT
);
```

Parameters

dwData

Specifies additional data. The value used depends on the value of the *nCmd* parameter.

nCmd

Specifies the type of help requested. For a list of possible values and how they affect the *dwData* parameter, see the WinHelp Windows function in the *Platform SDK*.

Remarks

See CWinApp::WinHelp for more information.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::HtmlHelp
CWnd::OnHelpUsing
CWnd::OnHelpIndex

Concepts

CWnd Members

CWnd Data Members

For information about the data member in **CWnd**, see CWnd Members.

CWnd::m_hWnd

The handle of the Windows window attached to this **CWnd**.

```
HWND m_hWnd;
```

Remarks

The **m_hWnd** data member is a public variable of type **HWND**.

See Also

Reference

CWnd Class
Hierarchy Chart
CWnd::Attach
CWnd::Detach
CWnd::FromHandle

Concepts

CWnd Members

CWnd Operators

For information about the operators in **CWnd**, see CWnd Members.

CWnd::operator HWND

Use this operator to get the handle to the **CWnd** object.

```
operator HWND( ) const;
```

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::m_hWnd

Concepts

CWnd Members

CWnd::operator !=

Compares two **CWnd** objects to determine if they do not have the same m_hWnd.

```
BOOL operator!=(
   const CWnd& wnd
) const;
```

Parameters

wnd

A reference to a **CWnd** object.

Return Value

Nonzero if equal; otherwise 0.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::operator ==

Concepts

CWnd Members

CWnd::operator ==

Compares two **CWnd** objects to determine if they have the same m_hWnd.

```
BOOL operator==(
   const CWnd& wnd
) const;
```

Parameters

wnd

A reference to a **CWnd** object.

Return Value

Nonzero if equal; otherwise 0.

See Also

Reference

CWnd Class

Hierarchy Chart

CWnd::operator !=

Concepts

CWnd Members

CWordArray Class

Supports arrays of 16-bit words.

```
class CWordArray : public CObject
```

Remarks

The member functions of **CWordArray** are similar to the member functions of class CObArray. Because of this similarity, you can use the **CObArray** reference documentation for member function specifics. Wherever you see a CObject pointer as a function parameter or return value, substitute a **WORD**.

```
CObject* CObArray::GetAt( int <nIndex> ) const;
```

for example, translates to

```
WORD CWordArray::GetAt( int <nIndex> ) const;
```

CWordArray incorporates the IMPLEMENT_SERIAL macro to support serialization and dumping of its elements. If an array of words is stored to an archive, either with an overloaded insertion operator or with the CObject::Serialize member function, each element is, in turn, serialized.

Note
Before using an array, use **SetSize** to establish its size and allocate memory for it. If you do not use **SetSize**, adding elements to your array causes it to be frequently reallocated and copied. Frequent reallocation and copying are inefficient and can fragment memory.

If you need a dump of individual elements in the array, you must set the depth of the dump context to 1 or greater.

For more information on using **CWordArray**, see the article Collections.

Requirements

Header: afxcoll.h

Smart Device Developer Notes

This class is not supported in Smart Device projects.

See Also

Tasks

COLLECT Sample: Illustrates MFC Collection Classes

Reference

CObject Class

Hierarchy Chart

Other Resources

CWordArray Members

CWordArray Members

Base Class Members

CObject Members

Construction

CWordArray	Constructs an empty array for words.

Bounds

GetCount	Gets number of elements in this array.
GetSize	Gets number of elements in this array.
GetUpperBound	Returns the largest valid index.
SetSize	Sets the number of elements to be contained in this array.

Operations

FreeExtra	Frees all unused memory above the current upper bound.
RemoveAll	Removes all the elements from this array.

Element Access

ElementAt	Returns a temporary reference to the element pointer within the array.
GetAt	Returns the value at a given index.
GetData	Allows access to elements in the array. Can be **NULL**.
IsEmpty	Determines if the array is empty.
SetAt	Sets the value for a given index; array is not allowed to grow.

Growing the Array

Add	Adds an element to the end of the array; grows the array if necessary.
Append	Appends another array to the array; grows the array if necessary.
Copy	Copies another array to the array; grows the array if necessary.
SetAtGrow	Sets the value for a given index; grows the array if necessary.

Insertion/Removal

InsertAt	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt	Removes an element at a specific index.

Operators

operator []	Sets or gets the element at the specified index.

See Also

Reference

CWordArray Class

Hierarchy Chart

ICommandSource Interface

Manages commands sent from a command source object to a user control.

```
interface class ICommandSource
```

Remarks

When you host a user control in an MFC View, CWinFormsView Class routes commands and update command UI messages to the user control to allow it to handle MFC commands (for example, frame menu items and toolbar buttons). By implementing ICommandTarget Interface, you give the user control a reference to the **ICommandSource** object.

See How to: Add Command Routing to the Windows Forms Control for an example of how to use **ICommandTarget**.

For more information on using Windows Forms, see Using a Windows Form User Control in MFC.

Requirements

Header: afxwinforms.h (defined in assembly atlmfc\lib\mfcmifc80.dll)

See Also

Tasks

How to: Add Command Routing to the Windows Forms Control

Reference

ICommandTarget Interface

Other Resources

ICommandSource Members

ICommandSource Members

The following are members of the **ICommandSource** class.

ICommandSource::AddCommandHandler	Adds a command handler to a command source object.
ICommandSource::RemoveCommandHandler	Removes a command handler from a command source object.
ICommandSource::AddCommandUIHandler	Adds a user interface command message handler to a command sourc e object.
ICommandSource::RemoveCommandUIHandler	Removes a user interface command message handler from a command source object.
ICommandSource::AddCommandRangeHandler	Adds a group of command handlers to a command source object.
ICommandSource::RemoveCommandRangeHandler	Removes a group of command handlers from a command source objec t.
ICommandSource::AddCommandRangeUIHandler	Adds a group of user interface command message handlers to a comm and source object.
ICommandSource::RemoveCommandRangeUIHandler	Removes a group of user interface command message handlers from a command source object.
ICommandSource::PostCommand	Posts a message without waiting for it to be processed.
ICommandSource::SendCommand	Sends a message and waits for it to be processed before returning.

See Also

Reference

ICommandSource Interface

ICommandSource Member Functions

For information about the member functions in **ICommandSource**, see ICommandSource Members.

ICommandSource::AddCommandHandler

Adds a command handler to a command source object.

```
void AddCommandHandler(
   unsigned int cmdID,
   CommandHandler^ cmdHandler
);
```

Parameters

cmdID

The command ID.

cmdHandler

A handle to the command handler method.

Remarks

This method adds the command handler *cmdHandler* to the command source object and maps the handler to *cmdID*.

See How to: Add Command Routing to the Windows Forms Control for an example of how to use **AddCommandHandler**.

Example

The following code adds a new command handler method called `singleMenuHandler` to the command source object `m_CmdSrc`.

```
private ICommandSource m_CmdSrc;
public void Initialize (ICommandSource cmdSrc)
{
   m_CmdSrc = cmdSrc;
   newHandler = new CommandHandler (singleMenuHandler);
   m_CmdSrc.AddCommandHandler (<resourceID>, newHandler);
}
```

See Also

Reference

ICommandSource Interface

ICommandSource::RemoveCommandHandler

ICommandSource::AddCommandRangeHandler

Adds a group of command handlers to a command source object.

```
void AddCommandRangeHandler(
   unsigned int cmdIDMin,
   unsigned int cmdIDMax,
   CommandHandler^ cmdHandler
);
```

Parameters

cmdIDMin

The beginning index of the command ID range.

cmdIDMax

The ending index of the command ID range.

cmdHandler

A handle to the message handler method to which the commands are mapped.

Remarks

This method maps a contiguous range of command IDs to a single message handler and adds it to the command source object. This is used for handling a group of related buttons with one method.

See Also

Reference

ICommandSource Interface

ICommandSource::RemoveCommandRangeHandler

ICommandSource::AddCommandRangeUIHandler

Adds a group of user interface command message handlers to a command source object.

```
void AddCommandRangeUIHandler(
   unsigned int cmdIDMin,
   unsigned int cmdIDMax,
   CommandUIHandler^ cmdUIHandler
);
```

Parameters

cmdIDMin

The beginning index of the command ID range.

cmdIDMax

The ending index of the command ID range.

cmdHandler

A handle to the message handler method to which the commands are mapped.

Remarks

This method maps a contiguous range of command IDs to a single user interface command message handler and adds it to the command source object. This is used for handling a group of related buttons with one method.

See Also

Reference

ICommandSource Interface

ICommandSource::RemoveCommandRangeUIHandler

ICommandSource::AddCommandUIHandler

Adds a user interface command message handler to a command source object.

```
void AddCommandUIHandler(
   unsigned int cmdID,
   CommandUIHandler^ cmdUIHandler
);
```

Parameters

cmdID

The command ID.

cmdUIHandler

A handle to the user interface command message handler method.

Remarks

This method adds the user interface command message handler *cmdHandler* to the command source object and maps the handler to *cmdID*.

See Also

Reference

ICommandSource Interface

ICommandSource::RemoveCommandUIHandler

ICommandSource::PostCommand

Posts a message without waiting for it to be processed.

```
void PostCommand(
   unsigned int command
);
```

Parameters

command

The command ID of the message to be posted.

Remarks

This method asynchronously posts the message mapped to the ID specified by *command*. It calls CWnd::PostMessage to place the message in the window's message queue and then returns without waiting for the corresponding window to process the message.

See Also

Reference

ICommandSource Interface

ICommandSource::SendCommand

ICommandSource::RemoveCommandHandler

Removes a command handler from a command source object.

```
void RemoveCommandHandler(
   unsigned int cmdID
);
```

Parameters

cmdID

The command ID.

Remarks

This method removes the command handler mapped to *cmdID* from the command source object.

See Also

Reference

ICommandSource Interface

ICommandSource::AddCommandHandler

ICommandSource::RemoveCommandRangeHandler

Removes a group of command handlers from a command source object.

```
void RemoveCommandRangeUIHandler(
   unsigned int cmdIDMin,
   unsigned int cmdIDMax
);
```

Parameters

cmdIDMin

The beginning index of the command ID range.

cmdIDMax

The ending index of the command ID range.

Remarks

This method removes a group of message handlers, mapped to the command IDs specifed by *cmdIDMin* and *cmdIDMax*, from the command source object.

See Also

Reference

ICommandSource Interface

ICommandSource::AddCommandRangeHandler

ICommandSource::RemoveCommandRangeUIHandler

Removes a group of user interface command message handlers from a command source object.

```
void RemoveCommandRangeUIHandler(
   unsigned int cmdIDMin,
   unsigned int cmdIDMax
);
```

Parameters

cmdIDMin

The beginning index of the command ID range.

cmdIDMax

The ending index of the command ID range.

Remarks

This method removes a group of user interface command message handlers, mapped to the command IDs specifed by *cmdIDMin* and *cmdIDMax*, from the command source object.

See Also

Reference

ICommandSource Interface

ICommandSource::AddCommandRangeUIHandler

ICommandSource::RemoveCommandUIHandler

Removes a user interface command message handler from a command source object.

```
void RemoveCommandUIHandler(
   unsigned int cmdID
);
```

Parameters

cmdID

The command ID.

Remarks

This method removes the user interface command message handler mapped to *cmdID* from the command source object.

See Also

Reference

ICommandSource Interface

ICommandSource::AddCommandUIHandler

ICommandSource::SendCommand

Sends a message and waits for it to be processed before returning.

```
void SendCommand(
   unsigned int command
);
```

Parameters

command

The command ID of the message to be sent.

Remarks

This method synchronously sends the message mapped to the ID specified by *command*. It calls CWnd::SendMessage to place the message in the window's message queue and waits until that window procedure has processed the message before returning.

See Also

Reference

ICommandSource Interface

ICommandSource::PostCommand

ICommandTarget Interface

Provides a user control with an interface to receive commands from a command source object.

```
interface class ICommandTarget
```

Remarks

When you host a user control in an MFC View, CWinFormsView routes commands and update command UI messages to the user control to allow it to handle MFC commands (for example, frame menu items and toolbar buttons). By implementing **ICommandTarget**, you give the user control a reference to the ICommandSource object.

See How to: Add Command Routing to the Windows Forms Control for an example of how to use **ICommandTarget**.

For more information on using Windows Forms, see Using a Windows Form User Control in MFC.

Requirements

Header: afxwinforms.h (defined in assembly atlmfc\lib\mfcmifc80.dll)

See Also

Tasks

How to: Add Command Routing to the Windows Forms Control

Reference

ICommandSource Interface

Other Resources

ICommandTarget Members

ICommandTarget Members

ICommandTarget::Initialize	Initializes the command target object.

See Also

Reference

ICommandTarget Interface

ICommandTarget Member Functions

For information about the member functions in **ICommandTarget**, see ICommandTarget Members.

ICommandTarget::Initialize

Initializes the command target object.

```
void Initialize(
   ICommandSource^ cmdSource
);
```

Parameters

cmdSource

A handle to the command source object.

Remarks

When you host a user control in an MFC View, CWinFormsView routes commands and update command UI messages to the user control to allow it to handle MFC commands.

This method initializes the command target object and associates it with the specified command source object *cmdSource*. It should be called in the user control class implementation. At initialization, you should register command handlers with the command source object by calling ICommandSource::AddCommandHandler in the **Initialize** implementation. See How to: Add Command Routing to the Windows Forms Control for an example of how to use **Initialize** to do this.

See Also

Reference

ICommandTarget Interface

ICommandSource Interface

ICommandUI Interface

Manages user interface commands.

```
interface class ICommandUI
```

Remarks

This interface provides methods and properties that manage user interface commands. **ICommandUI** is similar to CCmdUI Class, except that **ICommandUI** is used for MFC applications that interoperate with .NET components.

ICommandUI is used within an **ON_UPDATE_COMMAND_UI** handler in an ICommandTarget-derived class. When a user of an application activates (selects or clicks) a menu, each menu item is displayed as enabled or disabled. The target of each menu command provides this information by implementing an **ON_UPDATE_COMMAND_UI** handler. For each of the command user interface objects in your application, use the Properties window to create a message-map entry and function prototype for each handler.

For more information on how the **ICommandUI** interface is used in command routing, see How to: Add Command Routing to the Windows Forms Control.

For more information on using Windows Forms, see Using a Windows Form User Control in MFC.

For more information on how user interface commands are managed in MFC, see CCmdUI Class.

Requirements

Header: afxwinforms.h (defined in assembly atlmfc\lib\mfcmifc80.dll)

See Also

Reference

CCmdUI Class

Other Resources

ICommandUI Members

ICommandUI Members

The following are members of the **ICommandUI** class.

Methods

ContinueRouting	Tells the command-routing mechanism to continue routing the current message down the chain of handlers.

Properties

Check	Sets the user interface item for this command to the appropriate check state.
Enabled	Enables or disables the user interface item for this command.
ID	Gets the ID of the user interface object represented by the **ICommandUI** object.
Index	Gets the index of the user interface object represented by the **ICommandUI** object.
Radio	Sets the user interface item for this command to the appropriate check state.
Text	Sets the text of the user interface item for this command.

See Also

Reference

ICommandUI Interface

ICommandUI Member Functions

For information about the member functions in **ICommandUI**, see ICommandUI Members.

ICommandUI::Check

Sets the user interface item for this command to the appropriate check state.

```
property UICheckState Check;
```

Remarks

This property sets the user interface item for this command to the appropriate check state. Set **Check** to the following values:

Term	Definition
0	Uncheck
1	Check
2	Set indeterminate

See Also

Reference

ICommandUI Interface

Other Resources

ICommandUI Members

ICommandUI::ContinueRouting

Tells the command routing mechanism to continue routing the current message down the chain of handlers.

```
void ContinueRouting( );
```

Remarks

This is an advanced member function that should be used in conjunction with an ON_COMMAND_EX handler that returns **FALSE**. For more information, see Technical Note TN006: Message Maps.

See Also

Reference

ICommandUI Interface

ON_COMMAND_EX

Other Resources

ICommandUI Members

ICommandUI::Enabled

Enables or disables the user interface item for this command.

```
property bool Enabled;
```

Remarks

This property enables or disables the user interface item for this command. Set **Enabled** to **TRUE** to enable the item, **FALSE** to disable it.

See Also

Reference

ICommandUI Interface

Other Resources

ICommandUI Members

ICommandUI::ID

Gets the ID of the user interface object represented by the **ICommandUI** object.

```
property unsigned int ID;
```

Remarks

This property gets the ID (a handle) of the menu item, toolbar button, or other user interface object represented by the **ICommandUI** object.

See Also

Reference

ICommandUI Interface

Other Resources

ICommandUI Members

ICommandUI::Index

Gets the index of the user interface object represented by the **ICommandUI** object.

```
property unsigned int Index;
```

Remarks

This property gets the index (a handle) of the menu item, toolbar button, or other user interface object represented by the **ICommandUI** object.

See Also

Reference

ICommandUI Interface

Other Resources

ICommandUI Members

ICommandUI::Radio

Sets the user interface item for this command to the appropriate check state.

```
property bool Radio;
```

Remarks

This property sets the user interface item for this command to the appropriate check state. Set **Radio** to **TRUE** to enable the item; otherwise **FALSE**.

See Also

Reference

ICommandUI Interface

Other Resources

ICommandUI Members

ICommandUI::Text

Sets the text of the user interface item for this command.

```
property String^ Text;
```

Remarks

This property sets the text of the user interface item for this command. Set **Text** to a text string handle.

See Also

Reference

ICommandUI Interface

Other Resources

ICommandUI Members

IView Interface

Implements several methods that CWinFormsView uses to send view notifications to a managed control.

```
interface class IView
```

Remarks

IView implements several methods that **CWinFormsView** uses to forward common view notifications to a hosted managed control. These are OnInitialUpdate, OnUpdate and OnActivateView.

IView is similar to CView, but is used only with managed views and controls.

For more information on using Windows Forms, see Using a Windows Form User Control in MFC.

Requirements

Header: afxwinforms.h (defined in assembly atlmfc\lib\mfcmifc80.dll)

See Also

Reference

CWinFormsView Class

CView Class

Other Resources

IView Members

IView Members

The following are members of the **IView** Class.

IView::OnActivateView	Called by MFC when a view is activated or deactivated.
IView::OnInitialUpdate	Called by the framework after the view is first attached to the document, but before the view is initially displayed.
IView::OnUpdate	Called by MFC after the view's document has been modified; this function allows the view to update its display to reflect modifications.

See Also

Reference

IView Interface

IView Member Functions

For information about the member functions in **IView**, see IView Members.

IView::OnActivateView

Called by MFC when a view is activated or deactivated.

```
void OnActivateView(
   bool activate
);
```

Parameters

activate

Indicates whether the view is being activated or deactivated.

See Also

Tasks

How to: Add Command Routing to the Windows Forms Control

Reference

IView Interface

CView::OnActivateView

IView::OnInitialUpdate

Called by the framework after the view is first attached to the document, but before the view is initially displayed.

```
void OnInitialUpdate();
```

See Also

Tasks

How to: Add Command Routing to the Windows Forms Control

Reference

IView Interface

CView::OnInitialUpdate

IView::OnUpdate

Called by MFC after the view's document has been modified.

```
void OnUpdate();
```

Remarks

This function allows the view to update its display to reflect modifications.

See Also

Tasks

How to: Add Command Routing to the Windows Forms Control

Reference

IView Interface

CView::OnUpdate

MFC Macros and Globals

The Microsoft Foundation Class Library can be divided into two major sections: (1) the MFC classes and (2) macros and globals. If a function or variable is not a member of a class, it is a global function or variable.

The MFC library and the Active Template Library (ATL) share string conversion macros. See String Conversion Macros in the ATL documentation for a discussion of these macros.

The MFC macros and globals offer functionality in the following categories:

General MFC

- Data types
- Type casting of MFC class objects
- Run-time object model services
- Diagnostic services
- Exception processing
- CString formatting and message-box display
- Message maps
- Application information and management
- Standard command and window IDs
- Collection class helpers
- Gray and dithered bitmap functions
- Standard dialog data exchange (DDX) routines
- Standard dialog data validation (DDV) routines

Database

- Record Field Exchange (RFX) functions and Bulk Record Field Exchange (bulk RFX) functions for the MFC ODBC classes
- Record field exchange (DFX) functions for the MFC DAO classes
- Dialog data exchange (DDX) functions for CRecordView and CDaoRecordView (MFC ODBC and DAO classes)
- Dialog data exchange (DDX) functions for OLE controls
- Macros and globals to aid in calling Open Database Connectivity (ODBC) API functions directly
- DAO database engine initialization and termination

Internet

- Internet Server API (ISAPI) parse maps
- Internet URL parsing globals
- Internet Server API (ISAPI) diagnostic macros

DHTML / DHTML Event Maps

- DHTML dialog data exchange (DDX) helper macros
- DHTML event maps
- Multipage DHTML and URL event maps

OLE

- OLE initialization
- Application control
- Dispatch maps

In addition, MFC provides a function called AfxEnableControlContainer that enables any OLE container developed with MFC 4.0 to fully support embedded OLE controls.

OLE Controls

- Variant parameter type constants
- Type library access
- Property pages
- Event maps
- Event sink maps
- Connection maps
- Registering OLE controls
- Class factories and licensing
- Persistence of OLE controls

The first part of this section briefly discusses each of the previous categories and lists each global and macro in the category, along with a brief description of what it does. Following this alphabetically are complete descriptions of the global functions, global variables, and macros in the MFC library.

Note
Many global functions start with the prefix "Afx", but some, such as the dialog data exchange (DDX) functions and many of the database functions, deviate from this convention. All global variables start with "afx" as a prefix. Macros do not start with any particular prefix, but they are written in uppercase letters.

See Also

Concepts

Class Library Overview

Data Types (MFC)

This topic lists the data types most commonly used in the Microsoft Foundation Class Library. Most of the data types are the same as those in the Platform Software Development Kit (SDK), while others are unique to MFC.

For information about the data types used in both the Platform SDK and MFC, see Windows Data Types.

Data types unique to the Microsoft Foundation Class Library include the following:

- **POSITION** A value used to denote the position of an element in a collection; used by MFC collection classes.
- **LPCRECT** A 32-bit pointer to a constant (nonmodifiable) **RECT** structure.

See Also

Concepts

Class Library Overview
MFC Macros and Globals

Type Casting of MFC Class Objects

Type casting macros provide a way to cast a given pointer to a pointer that points to an object of specific class, with or without checking that the cast is legal.

The following table lists the MFC type casting macros.

Macros That Cast Pointers to MFC Class Objects

DYNAMIC_DOWNCAST	Casts a pointer to a pointer to a class object while checking to see if the cast is legal.
STATIC_DOWNCAST	Casts a pointer to an object from one class to a pointer of a related type. In a debug build, causes an **ASSERT** if the object is not a "kind of" the target type.

See Also

Concepts

MFC Macros and Globals

Run-Time Object Model Services

The classes CObject and CRuntimeClass encapsulate several object services, including access to run-time class information, serialization, and dynamic object creation. All classes derived from **CObject** inherit this functionality.

Access to run-time class information enables you to determine information about an object's class at run time. The ability to determine the class of an object at run time is useful when you need extra type-checking of function arguments and when you must write special-purpose code based on the class of an object. Run-time class information is not supported directly by the C++ language.

Serialization is the process of writing or reading an object's contents to or from a file. You can use serialization to store an object's contents even after the application exits. The object can then be read from the file when the application is restarted. Such data objects are said to be "persistent."

Dynamic object creation enables you to create an object of a specified class at run time. For example, document, view, and frame objects must support dynamic creation because the framework needs to create them dynamically.

The following table lists the MFC macros that support run-time class information, serialization, and dynamic creation.

For more information on these run-time object services and serialization, see the article CObject Class: Accessing Run-Time Class Information.

Run-Time Object Model Services Macros

DECLARE_DYNAMIC	Enables access to run-time class information (must be used in the class declaration).
DECLARE_DYNCREATE	Enables dynamic creation and access to run-time class information (must be used in the class declaration).
DECLARE_SERIAL	Enables serialization and access to run-time class information (must be used in the class declaration).
IMPLEMENT_DYNAMIC	Enables access to run-time class information (must be used in the class implementation).
IMPLEMENT_DYNCREATE	Enables dynamic creation and access to run-time information (must be used in the class implementation).
IMPLEMENT_SERIAL	Permits serialization and access to run-time class information (must be used in the class implementation).
RUNTIME_CLASS	Returns the **CRuntimeClass** structure that corresponds to the named class.

OLE frequently requires the dynamic creation of objects at run time. For example, an OLE server application must be able to create OLE items dynamically in response to a request from a client. Similarly, an automation server must be able to create items in response to requests from automation clients.

The Microsoft Foundation Class Library provides two macros specific to OLE.

Dynamic Creation of OLE Objects

DECLARE_OLECREATE	Enables objects to be created through OLE automation.
IMPLEMENT_OLECREATE	Enables objects to be created by the OLE system.

See Also

Concepts

MFC Macros and Globals

Diagnostic Services

The Microsoft Foundation Class Library supplies many diagnostic services that make debugging your programs easier. These diagnostic services include macros and global functions that allow you to track your program's memory allocations, dump the contents of objects during run time, and print debugging messages during run time. The macros and global functions for diagnostic services are grouped into the following categories:

- General diagnostic macros
- General diagnostic functions and variables
- Object diagnostic functions

These macros and functions are available for all classes derived from **CObject** in the Debug and Release versions of MFC. However, all except **DEBUG_NEW** and **VERIFY** do nothing in the Release version.

In the Debug library, all allocated memory blocks are bracketed with a series of "guard bytes." If these bytes are disturbed by an errant memory write, then the diagnostic routines can report a problem. If you include the line:

```
#define new DEBUG_NEW
```

in your implementation file, all calls to **new** will store the filename and line number where the memory allocation took place. The function CMemoryState::DumpAllObjectsSince will display this extra information, allowing you to identify memory leaks. Refer also to the class CDumpContext for additional information on diagnostic output.

In addition, the C run-time library also supports a set of diagnostic functions you can use to debug your applications. For more information, see Debug Routines in the Run-Time Library Reference.

MFC General Diagnostic Macros

ASSERT	Prints a message and then aborts the program if the specified expression evaluates to **FALSE** in the Debug version of the library.
ASSERT_KINDOF	Tests that an object is an object of the specified class or of a class derived from the specified class.
ASSERT_VALID	Tests the internal validity of an object by calling its **AssertValid** member function; typically overridden from **CObject**.
DEBUG_NEW	Supplies a filename and line number for all object allocations in Debug mode to help find memory leaks.
DEBUG_ONLY	Similar to **ASSERT** but does not test the value of the expression; useful for code that should execute only in Debug mode.
TRACE	Provides **printf**-like capability in the Debug version of the library.
VERIFY	Similar to **ASSERT** but evaluates the expression in the Release version of the library as well as in the Debug version.

MFC General Diagnostic Variables and Functions

afxDump	Global variable that sends CDumpContext information to the debugger output window or to the debug terminal.
afxMemDF	Global variable that controls the behavior of the debugging memory allocator.
AfxCheckError	Global variable used to test the passed **SCODE** to see if it is an error and, if so, throws the appropriate error.
AfxCheckMemory	Checks the integrity of all currently allocated memory.

AfxDump	If called while in the debugger, dumps the state of an object while debugging.
AfxDumpStack	Generate an image of the current stack. This function is always linked statically.
AfxEnableMemoryTracking	Turns memory tracking on and off.
AfxIsMemoryBlock	Verifies that a memory block has been properly allocated.
AfxIsValidAddress	Verifies that a memory address range is within the program's bounds.
AfxIsValidString	Determines whether a pointer to a string is valid.
AfxSetAllocHook	Enables the calling of a function on each memory allocation.

MFC Object Diagnostic Functions

AfxDoForAllClasses	Performs a specified function on all **CObject**-derived classes that support run-time type checking.
AfxDoForAllObjects	Performs a specified function on all **CObject**-derived objects that were allocated with **new**.

See Also

Concepts

MFC Macros and Globals

Exception Processing

When a program executes, a number of abnormal conditions and errors called "exceptions" can occur. These may include running out of memory, resource allocation errors, and failure to find files.

The Microsoft Foundation Class Library uses an exception-handling scheme that is modeled closely after the one proposed by the ANSI standards committee for C++. An exception handler must be set up before calling a function that may encounter an abnormal situation. If the function encounters an abnormal condition, it throws an exception and control is passed to the exception handler.

Several macros included with the Microsoft Foundation Class Library will set up exception handlers. A number of other global functions help to throw specialized exceptions and terminate programs, if necessary. These macros and global functions fall into the following categories:

- Exception macros, which structure your exception handler.
- Exception-throwing functions, which generate exceptions of specific types.
- Termination functions, which cause program termination.

For examples and more details, see the article Exceptions.

Exception Macros

TRY	Designates a block of code for exception processing.
CATCH	Designates a block of code for catching an exception from the preceding **TRY** block.
CATCH_ALL	Designates a block of code for catching all exceptions from the preceding **TRY** block.
AND_CATCH	Designates a block of code for catching additional exception types from the preceding **TRY** block.
AND_CATCH_ALL	Designates a block of code for catching all other additional exception types thrown in a preceding **TRY** block .
END_CATCH	Ends the last **CATCH** or **AND_CATCH** code block.
END_CATCH_ALL	Ends the last **CATCH_ALL** code block.
THROW	Throws a specified exception.
THROW_LAST	Throws the currently handled exception to the next outer handler.

Exception-Throwing Functions

AfxThrowArchiveException	Throws an archive exception.
AfxThrowFileException	Throws a file exception.
AfxThrowMemoryException	Throws a memory exception.
AfxThrowNotSupportedException	Throws a not-supported exception.
AfxThrowResourceException	Throws a Windows resource-not-found exception.
AfxThrowUserException	Throws an exception in a user-initiated program action.

MFC provides two exception-throwing functions specifically for OLE exceptions:

OLE Exception Functions

AfxThrowOleDispatchException	Throws an exception within an OLE automation function.
AfxThrowOleException	Throws an OLE exception.

To support database exceptions, the database classes provide two exception classes, **CDBException** and **CDaoException**, and global functions to support the exception types:

DAO Exception Functions

AfxThrowDAOException	Throws a CDaoException from your own code.
AfxThrowDBException	Throws a CDBException from your own code.

MFC provides the following termination function:

Termination Functions

AfxAbort	Called to terminate an application when a fatal error occurs.

See Also

Reference

CException Class

Concepts

MFC Macros and Globals

CString Formatting and Message-Box Display

A number of functions are provided to format and parse **CString** objects. You can use these functions whenever you have to manipulate **CString** objects, but they are particularly useful for formatting strings that will appear in message-box text.

This group of functions also includes a global routine for displaying a message box.

CString Functions

AfxExtractSubString	Extracts substrings separated by a single character from a given source string.
AfxFormatString1	Substitutes a given string for the format characters "%1" in a string contained in the string table.
AfxFormatString2	Substitutes two strings for the format characters "%1" and "%2" in a string contained in the string table.
AfxMessageBox	Displays a message box.

See Also

Reference

CStringT Class

Concepts

MFC Macros and Globals

Application Information and Management

When you write an application, you create a single CWinApp-derived object. At times, you may want to get information about this object from outside the **CWinApp**-derived object.

The Microsoft Foundation Class Library provides the following global functions to help you accomplish these tasks:

Application Information and Management Functions

AfxFreeLibrary	Decrements the reference count of the loaded dynamic-link library (DLL) module; when the reference count reaches zero, the module is unmapped.
AfxGetApp	Returns a pointer to the application's single **CWinApp** object.
AfxGetAppName	Returns a string containing the application's name.
AfxGetInstanceHandle	Returns an **HINSTANCE** representing this instance of the application.
AfxGetMainWnd	Returns a pointer to the current "main" window of a non-OLE application, or the in-place frame window of a server application.
AfxGetResourceHandle	Returns an **HINSTANCE** to the source of the application's default resources. Use this to access the application's resources directly.
AfxInitRichEdit	Initializes the version 1.0 rich edit control for the application.
AfxInitRichEdit2	Initializes the version 2.0 and later rich edit control for the application.
AfxLoadLibrary	Maps a DLL module and returns a handle that can be used to get the address of a DLL function.
AfxRegisterWndClass	Registers a Windows window class to supplement those registered automatically by MFC.
AfxSocketInit	Called in a **CWinApp::InitInstance** override to initialize Windows Sockets.
AfxSetResourceHandle	Sets the **HINSTANCE** handle where the default resources of the application are loaded.
AfxRegisterClass	Registers a window class in a DLL that uses MFC.
AfxBeginThread	Creates a new thread.
AfxEndThread	Terminates the current thread.
AfxGetThread	Retrieves a pointer to the current CWinThread object.
AfxWinInit	Called by the MFC-supplied **WinMain** function, as part of the CWinApp initialization of a GUI-based application, to initialize MFC. Must be called directly for console applications using MFC.

See Also

Reference

CWinApp Class

Concepts

MFC Macros and Globals

Standard Command and Window IDs

The Microsoft Foundation Class Library defines a number of standard command and window IDs in Afxres.h. These IDs are most commonly used within the resource editors and the Properties window to map messages to your handler functions. All standard commands have an **ID_** prefix. For example, when you use the menu editor, you normally bind the File Open menu item to the standard **ID_FILE_OPEN** command ID.

For most standard commands, application code does not need to refer to the command ID, because the framework itself handles the commands through message maps in its primary framework classes (**CWinThread**, **CWinApp**, **CView**, **CDocument**, and so on).

In addition to standard command IDs, a number of other standard IDs are defined which have a prefix of **AFX_ID**. These IDs include standard window IDs (prefix **AFX_IDW**_), string IDs (prefix **AFX_IDS**_), and several other types.

IDs that begin with the **AFX_ID** prefix are rarely used by programmers, but you might need to refer to these IDs when overriding framework functions that also refer to the **AFX_ID**s.

IDs are not individually documented in this reference. You can find more information on them in Technical Notes 20, 21, and 22.

Note
The header file Afxres.h is indirectly included in Afxwin.h. You must explicitly include the following statement in your applicati on's resource script (.rc) file:

```
#include afxres.h
```

See Also

Concepts

MFC Macros and Globals

Collection Class Helpers

The collection classes **CMap**, **CList**, and **CArray** use templated global helper functions for such purposes as comparing, copying, and serializing elements. As part of your implementation of classes based on **CMap**, **CList**, and **CArray**, you must override these functions as necessary with versions tailored to the type of data stored in your map, list, or array. For information on overriding helper functions such as **SerializeElements**, see the article Collections: How to Make a Type-Safe Collection. Note that **ConstructElements** and **DestructElements** have been deprecated.

The Microsoft Foundation Class Library provides the following global functions to help you customize your collection classes:

Collection Class Helpers

CompareElements	Indicates whether elements are the same.
CopyElements	Copies elements from one array to another.
DumpElements	Provides stream-oriented diagnostic output.
HashKey	Calculates a hash key.
SerializeElements	Stores or retrieves elements to or from an archive.

See Also

Reference

CMap Class
CList Class
CArray Class

Concepts

MFC Macros and Globals

Gray and Dithered Bitmap Functions

Gray Bitmap Functions

MFC provides two functions for giving a bitmap the appearance of a disabled control.

AfxDrawGrayBitmap	Draws a gray version of a bitmap.
AfxGetGrayBitmap	Copies a gray version of a bitmap.

Dithered Bitmap Functions

MFC also provides two functions for replacing a bitmap's background with a dithered pattern.

AfxDrawDitheredBitmap	Draws a bitmap with a dithered background.
AfxGetDitheredBitmap	Copies a bitmap with a dithered background.

See Also

Concepts

MFC Macros and Globals

Record Field Exchange Functions

This topic lists the Record Field Exchange (RFX, Bulk RFX, and DFX) functions used to automate the transfer of data between a recordset object and its data source and to perform other operations on the data.

If you are using the ODBC-based classes and you have implemented bulk row fetching, you must manually override the **DoBulkFieldExchange** member function of **CRecordset** by calling the Bulk RFX functions for each data member corresponding to a data source column.

If you have not implemented bulk row fetching in the ODBC-based classes, or if you are using the DAO-based classes, then ClassWizard will override the **DoFieldExchange** member function of **CRecordset** or **CDaoRecordset** by calling the RFX functions (for ODBC classes) or the DFX functions (for DAO classes) for each field data member in your recordset.

The record field exchange functions transfer data each time the framework calls **DoFieldExchange** or **DoBulkFieldExchange**. Each function transfers a specific data type.

For more information about how these functions are used, see the articles Record Field Exchange: How RFX Works (ODBC). For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC).

For columns of data that you bind dynamically, you can also call the RFX or DFX functions yourself, as explained in the articles Recordset: Dynamically Binding Data Columns (ODBC). Additionally, you can write your own custom RFX or DFX routines, as explained in Technical Note 43 (for ODBC) and Technical Note 53 (for DAO).

For an example of RFX and Bulk RFX functions as they appear in the **DoFieldExchange** and **DoBulkFieldExchange** functions, see RFX_Text and RFX_Text_Bulk. DFX functions are very similar to the RFX functions.

RFX Functions (ODBC)

RFX_Binary	Transfers arrays of bytes of type CByteArray.
RFX_Bool	Transfers Boolean data.
RFX_Byte	Transfers a single byte of data.
RFX_Date	Transfers time and date data using CTime or **TIMESTAMP_STRUCT**.
RFX_Double	Transfers double-precision float data.
RFX_Int	Transfers integer data.
RFX_Long	Transfers long integer data.
RFX_LongBinary	Transfers binary large object (BLOB) data with an object of the CLongBinary class.
RFX_Single	Transfers float data.
RFX_Text	Transfers string data.

Bulk RFX Functions (ODBC)

RFX_Binary_Bulk	Transfers arrays of byte data.
RFX_Bool_Bulk	Transfers arrays of Boolean data.
RFX_Byte_Bulk	Transfers arrays of single bytes.
RFX_Date_Bulk	Transfers arrays of data of type **TIMESTAMP_STRUCT**.
RFX_Double_Bulk	Transfers arrays of double-precision, floating-point data.

RFX_Int_Bulk	Transfers arrays of integer data.
RFX_Long_Bulk	Transfers arrays of long integer data.
RFX_Single_Bulk	Transfers arrays of floating-point data.
RFX_Text_Bulk	Transfers arrays of data of type **LPSTR**.

DFX Functions (DAO)

DFX_Binary	Transfers arrays of bytes of type CByteArray.
DFX_Bool	Transfers Boolean data.
DFX_Byte	Transfers a single byte of data.
DFX_Currency	Transfers currency data, of type COleCurrency.
DFX_DateTime	Transfers time and date data, of type COleDateTime.
DFX_Double	Transfers double-precision float data.
DFX_Long	Transfers long integer data.
DFX_LongBinary	Transfers binary large object (BLOB) data with an object of the **CLongBinary** class. For DAO, it is recommended that you use DFX_Binary instead.
DFX_Short	Transfers short integer data.
DFX_Single	Transfers float data.
DFX_Text	Transfers string data.

See Also

Reference

CRecordset::DoFieldExchange
CRecordset::DoBulkFieldExchange
CDaoRecordset::DoFieldExchange

Concepts

MFC Macros and Globals

Dialog Data Exchange Functions for CRecordView and CDaoRecordView

This topic lists the DDX_Field functions used to exchange data between a CRecordset and a CRecordView form or a CDaoRecordset and a CDaoRecordView form.

Note
DDX_Field functions are like DDX functions in that they exchange data with controls in a form. But unlike DDX, they exchange data with the fields of the view's associated recordset object rather than with fields of the record view itself. For more information, see classes **CRecordView** and **CDaoRecordView**.

DDX_Field Functions

DDX_FieldCBIndex	Transfers integer data between a recordset field data member and the index of the current selection in a combo box in a CRecordView or CDaoRecordView.
DDX_FieldCBString	Transfers **CString** data between a recordset field data member and the edit control of a combo box in a **CRecordView** or **CDaoRecordView**. When moving data from the recordset to the control, this function selects the item in the combo box that begins with the characters in the specified string.
DDX_FieldCBStringExact	Transfers **CString** data between a recordset field data member and the edit control of a combo box in a **CRecordView** or **CDaoRecordView**. When moving data from the recordset to the control, this function selects the item in the combo box that exactly matches the specified string.
DDX_FieldCheck	Transfers Boolean data between a recordset field data member and a check box in a **CRecordView** or **CDaoRecordView**.
DDX_FieldLBIndex	Transfers integer data between a recordset field data member and the index of the current selection in a list box in a **CRecordView** or **CDaoRecordView**.
DDX_FieldLBString	Manages the transfer of CString data between a list-box control and the field data members of a recordset. When moving data from the recordset to the control, this function selects the item in the list box that begins with the characters in the specified string.
DDX_FieldLBStringExact	Manages the transfer of **CString** data between a list-box control and the field data members of a recordset. When moving data from the recordset to the control, this function selects the first item that exactly matches the specified string.
DDX_FieldRadio	Transfers integer data between a recordset field data member and a group of radio buttons in a **CRecordView** or **CDaoRecordView**.
DDX_FieldScroll	Sets or gets the scroll position of a scroll bar control in a **CRecordView** or **CDaoRecordView**. Call from your DoFieldExchange function.
DDX_FieldText	Overloaded versions are available for transferring **int**, **UINT**, **long**, **DWORD**, CString, **float**, **double**, **short**, COleDateTime, and COleCurrency data between a recordset field data member and an edit box in a **CRecordView** or **CDaoRecordView**.

See Also

Concepts

MFC Macros and Globals

Dialog Data Exchange Functions for OLE Controls

This topic lists the DDX_OC functions used to exchange data between a property of an OLE control in a dialog box, form view, or control view object and a data member of the dialog box, form view, or control view object.

DDX_OC Functions

DDX_OCBool	Manages the transfer of **BOOL** data between a property of an OLE control and a **BOOL** data member.
DDX_OCBoolRO	Manages the transfer of **BOOL** data between a read-only property of an OLE control and a **BOOL** data member.
DDX_OCColor	Manages the transfer of **OLE_COLOR** data between a property of an OLE control and an **OLE_COLOR** data member.
DDX_OCColorRO	Manages the transfer of **OLE_COLOR** data between a read-only property of an OLE control and an **OLE_COLOR** data member.
DDX_OCFloat	Manages the transfer of **float** (or **double**) data between a property of an OLE control and a **float** (or **double**) data member.
DDX_OCFloatRO	Manages the transfer of **float** (or **double**) data between a read-only property of an OLE control and a **float** (or **double**) data member.
DDX_OCInt	Manages the transfer of **int** (or **long**) data between a property of an OLE control and an **int** (or **long**) data member.
DDX_OCIntRO	Manages the transfer of **int** (or **long**) data between a read-only property of an OLE control and an **int** (or **long**) data member.
DDX_OCShort	Manages the transfer of **short** data between a property of an OLE control and a **short** data member.
DDX_OCShortRO	Manages the transfer of **short** data between a read-only property of an OLE control and a **short** data member.
DDX_OCText	Manages the transfer of **CString** data between a property of an OLE control and a **CString** data member.
DDX_OCTextRO	Manages the transfer of **CString** data between a read-only property of an OLE control and a **CString** data member.

See Also

Concepts

MFC Macros and Globals

Database Macros and Globals

The macros and globals listed below apply to ODBC-based database applications. They are not used with DAO-based applications.

Before MFC 4.2, the macros **AFX_SQL_ASYNC** and **AFX_SQL_SYNC** gave asynchronous operations an opportunity to yield time to other processes. Beginning with MFC 4.2, the implementation of these macros changed because the MFC ODBC classes used only synchronous operations. The macro **AFX_ODBC_CALL** was new to MFC 4.2.

Database Macros

AFX_ODBC_CALL	Calls an ODBC API function that returns **SQL_STILL_EXECUTING**. **AFX_ODBC_CALL** will repeatedly call the function until it no longer returns **SQL_STILL_EXECUTING**.
AFX_SQL_ASYNC	Calls **AFX_ODBC_CALL**.
AFX_SQL_SYNC	Calls an ODBC API function that does not return **SQL_STILL_EXECUTING**.

Database Globals

AfxGetHENV	Retrieves a handle to the ODBC environment currently in use by MFC. You can use this handle in direct ODBC calls.

See Also

Concepts

MFC Macros and Globals

DAO Database Engine Initialization and Termination

When using MFC DAO objects, the DAO database engine must first be initialized and then terminated before your application or DLL quits. Two functions, **AfxDaoInit** and **AfxDaoTerm**, perform these tasks.

DAO Database Engine Initialization and Termination

AfxDaoInit	Initializes the DAO database engine.
AfxDaoTerm	Terminates the DAO database engine.

See Also

Concepts

MFC Macros and Globals

OLE Initialization

Before an application can use OLE system services, it must initialize the OLE system DLLs and verify that the DLLs are the correct version. The **AfxOleInit** function initializes the OLE system DLLs.

OLE Initialization

AfxOleInit	Initializes the OLE libraries.

See Also

Concepts

MFC Macros and Globals

Application Control

OLE requires substantial control over applications and their objects. The OLE system DLLs must be able to launch and release applications automatically, coordinate their production and modification of objects, and so on. The functions in this topic meet those requirements. In addition to being called by the OLE system DLLs, these functions must sometimes be called by applications as well.

Application Control

AfxOleCanExitApp	Indicates whether the application can terminate.
AfxOleGetMessageFilter	Retrieves the application's current message filter.
AfxOleGetUserCtrl	Retrieves the current user-control flag.
AfxOleSetUserCtrl	Sets or clears the user-control flag.
AfxOleLockApp	Increments the framework's global count of the number of active objects in an application.
AfxOleUnlockApp	Decrements the framework's count of the number of active objects in an application.
AfxOleRegisterServerClass	Registers a server in the OLE system registry.
AfxOleSetEditMenu	Implements the user interface for the *typename* Object command.

See Also

Concepts

MFC Macros and Globals

Dispatch Maps

OLE Automation provides ways to call methods and to access properties across applications. The mechanism supplied by the Microsoft Foundation Class Library for dispatching these requests is the "dispatch map," which designates the internal and external names of object functions and properties, as well as the data types of the properties themselves and of function arguments.

Dispatch Maps

DECLARE_DISPATCH_MAP	Declares that a dispatch map will be used to expose a class's methods and properties (must be used in the class declaration).
BEGIN_DISPATCH_MAP	Starts the definition of a dispatch map.
END_DISPATCH_MAP	Ends the definition of a dispatch map.
DISP_FUNCTION	Used in a dispatch map to define an OLE automation function.
DISP_PROPERTY	Defines an OLE automation property.
DISP_PROPERTY_EX	Defines an OLE automation property and names the Get and Set functions.
DISP_PROPERTY_NOTIFY	Defines an OLE automation property with notification.
DISP_PROPERTY_PARAM	Defines an OLE automation property that takes parameters and names the Get and Set functions.
DISP_DEFVALUE	Makes an existing property the default value of an object.

See Also

Concepts

MFC Macros and Globals

Variant Parameter Type Constants

This topic lists new constants that indicate variant parameter types designed for use with the OLE control classes of the Microsoft Foundation Class Library.

The following is a list of class constants:

Variant Data Constants

- **VTS_COLOR** A 32-bit integer used to represent a RGB color value.
- **VTS_FONT** A pointer to the **IFontDisp** interface of an OLE font object.
- **VTS_HANDLE** A Windows handle value.
- **VTS_PICTURE** A pointer to the **IPictureDisp** interface of an OLE picture object.
- **VTS_OPTEXCLUSIVE** A 16-bit value used for a control that is intended to be used in a group of controls, such as radio buttons. This type tells the container that if one control in a group has a **TRUE** value, all others must be **FALSE**.
- **VTS_TRISTATE** A 16-bit signed integer used for properties that can have one of three possible values (selected, cleared, unavailable), for example, a check box.
- **VTS_XPOS_HIMETRIC** A 32-bit unsigned integer used to represent a position along the x-axis in **HIMETRIC** units.
- **VTS_YPOS_HIMETRIC** A 32-bit unsigned integer used to represent a position along the y-axis in **HIMETRIC** units.
- **VTS_XPOS_PIXELS** A 32-bit unsigned integer used to represent a position along the x-axis in pixels.
- **VTS_YPOS_PIXELS** A 32-bit unsigned integer used to represent a position along the y-axis in pixels.
- **VTS_XSIZE_PIXELS** A 32-bit unsigned integer used to represent the width of a screen object in pixels.
- **VTS_YSIZE_PIXELS** A 32-bit unsigned integer used to represent the height of a screen object in pixels.
- **VTS_XSIZE_HIMETRIC** A 32-bit unsigned integer used to represent the width of a screen object in **HIMETRIC** units.
- **VTS_YSIZE_HIMETRIC** A 32-bit unsigned integer used to represent the height of a screen object in **HIMETRIC** units.

Note
Additional variant constants have been defined for all variant types, with the exception of **VTS_FONT** and **VTS_PICTURE**, that provide a pointer to the variant data constant. These constants are named using the **VTS_P***constantname* convention. For example, **VTS_PCOLOR** is a pointer to a **VTS_COLOR** constant.

See Also

Concepts

MFC Macros and Globals

Type Library Access

Type libraries expose the interfaces of an OLE control to other OLE-aware applications. Each OLE control must have a type library if one or more interfaces are to be exposed.

The following macros allow an OLE control to provide access to its own type library:

Type Library Access

DECLARE_OLETYPELIB	Declares a **GetTypeLib** member function of an OLE control (must be used in the class declaration).
IMPLEMENT_OLETYPELIB	Implements a **GetTypeLib** member function of an OLE control (must be used in the class implementation).

See Also

Concepts

MFC Macros and Globals

Property Pages (MFC)

Property pages display the current values of specific OLE control properties in a customizable, graphical interface for viewing and editing by supporting a data-mapping mechanism based on dialog data exchange (DDX).

This data-mapping mechanism maps property page controls to the individual properties of the OLE control. The value of the control property reflects the status or content of the property page control. The mapping between property page controls and properties is specified by **DDP_** function calls in the property page's **DoDataExchange** member function. The following is a list of **DDP_** functions that exchange data entered using the property page of your control:

Property Page Data Transfer

DDP_CBIndex	Links the selected string's index in a combo box with a control's property.
DDP_CBString	Links the selected string in a combo box with a control's property. The selected string can begin with the same letters as the property's value but does not need to match it fully.
DDP_CBStringExact	Links the selected string in a combo box with a control's property. The selected string and the property's string value must match exactly.
DDP_Check	Links a check box in the control's property page with a control's property.
DDP_LBIndex	Links the selected string's index in a list box with a control's property.
DDP_LBString	Links the selected string in a list box with a control's property. The selected string can begin with the sam e letters as the property's value but need not match it fully.
DDP_LBStringExact	Links the selected string in a list box with a control's property. The selected string and the property's strin g value must match exactly.
DDP_PostProcessing	Finishes the transfer of property values from your control.
DDP_Radio	Links a radio button group in the control's property page with a control's property.
DDP_Text	Links a control in the control's property page with a control's property. This function handles several diffe rent types of properties, such as **double**, **short**, **BSTR**, and **long**.

For more information about the **DoDataExchange** function and property pages, see the article ActiveX Controls: Property Pages.

The following is a list of macros used to create and manage property pages for an OLE control:

Property Pages

BEGIN_PROPPAGEIDS	Begins the list of property page IDs.
END_PROPPAGEIDS	Ends the list of property page IDs.
PROPPAGEID	Declares a property page of the control class.

See Also

Concepts

MFC Macros and Globals

Event Maps

Whenever a control wishes to notify its container that some action (determined by the control developer) has happened (such as a keystroke, mouse click, or a change to the control's state) it calls an event-firing function. This function notifies the control container that some important action has occurred by firing the related event.

The Microsoft Foundation Class Library offers a programming model optimized for firing events. In this model, "event maps" are used to designate which functions fire which events for a particular control. Event maps contain one macro for each event. For example, an event map that fires a stock Click event might look like this:

```
BEGIN_EVENT_MAP(CSampleCtrl, COleControl)
   //{{AFX_EVENT_MAP(CSampleCtrl)
   EVENT_STOCK_CLICK( )
   //}}AFX_EVENT_MAP
END_EVENT_MAP()
```

The **EVENT_STOCK_CLICK** macro indicates that the control will fire a stock Click event every time it detects a mouse click. For a more detailed listing of other stock events, see the article ActiveX Controls: Events. Macros are also available to indicate custom events.

Although event-map macros are important, you generally do not insert them directly. This is because the Properties window automatically creates event-map entries in your source files when you use it to associate event-firing functions with events. Any time you want to edit or add an event-map entry, you can use the Properties window.

To support event maps, MFC provides the following macros:

Event Map Declaration and Demarcation

DECLARE_EVENT_MAP	Declares that an event map will be used in a class to map events to event-firing functions (must be used in the class declaration).
BEGIN_EVENT_MAP	Begins the definition of an event map (must be used in the class implementation).
END_EVENT_MAP	Ends the definition of an event map (must be used in the class implementation).

Event Mapping Macros

EVENT_CUSTOM	Indicates which event-firing function will fire the specified event.
EVENT_CUSTOM_ID	Indicates which event-firing function will fire the specified event, with a designated dispatch ID.

Message Mapping Macros

ON_OLEVERB	Indicates a custom verb handled by the OLE control.
ON_STDOLEVERB	Overrides a standard verb mapping of the OLE control.

See Also

Concepts

MFC Macros and Globals

Event Sink Maps

When an embedded OLE control fires an event, the control's container receives the event using a mechanism, called an "event sink map," supplied by MFC. This event sink map designates handler functions for each specific event, as well as parameters of those events. For more information on event sink maps, see the article ActiveX Control Containers.

Event Sink Maps

BEGIN_EVENTSINK_MAP	Starts the definition of an event sink map.
DECLARE_EVENTSINK_MAP	Declares an event sink map.
END_EVENTSINK_MAP	Ends the definition of an event sink map.
ON_EVENT	Defines an event handler for a specific event.
ON_EVENT_RANGE	Defines an event handler for a specific event fired from a set of OLE controls.
ON_EVENT_REFLECT	Receives events fired by the control before they are handled by the control's container.
ON_PROPNOTIFY	Defines a handler for handling property notifications from an OLE control.
ON_PROPNOTIFY_RANGE	Defines a handler for handling property notifications from a set of OLE controls.
ON_PROPNOTIFY_REFLECT	Receives property notifications sent by the control before they are handled by the control's container.

See Also

Concepts

MFC Macros and Globals

Connection Maps

OLE controls are able to expose interfaces to other applications. These interfaces only allow access from a container into that control. If an OLE control wants to access external interfaces of other OLE objects, a connection point must be established. This connection point allows a control outgoing access to external dispatch maps, such as event maps or notification functions.

The Microsoft Foundation Class Library offers a programming model that supports connection points. In this model, "connection maps" are used to designate interfaces or connection points for the OLE control. Connection maps contain one macro for each connection point. For more information on connection maps, see the CConnectionPoint class.

Typically, a control will support just two connection points: one for events and one for property notifications. These are implemented by the **COleControl** base class and require no additional work by the control writer. Any additional connection points you want to implement in your class must be added manually. To support connection maps and points, MFC provides the following macros:

Connection Map Declaration and Demarcation

BEGIN_CONNECTION_PART	Declares an embedded class that implements an additional connection point (must be used in t he class declaration).
END_CONNECTION_PART	Ends the declaration of a connection point (must be used in the class declaration).
CONNECTION_IID	Specifies the interface ID of the control's connection point.
DECLARE_CONNECTION_MAP	Declares that a connection map will be used in a class (must be used in the class declaration).
BEGIN_CONNECTION_MAP	Begins the definition of a connection map (must be used in the class implementation).
END_CONNECTION_MAP	Ends the definition of a connection map (must be used in the class implementation).
CONNECTION_PART	Specifies a connection point in the control's connection map.

The following functions assist a sink in establishing and disconnecting a connection using connection points:

Initialization/Termination of Connection Points

AfxConnectionAdvise	Establishes a connection between a source and a sink.
AfxConnectionUnadvise	Breaks a connection between a source and a sink.

See Also

Concepts

MFC Macros and Globals

Registering OLE Controls

OLE controls, like other OLE server objects, can be accessed by other OLE-aware applications. This is achieved by registering the control's type library and class.

The following functions allow you to add and remove the control's class, property pages, and type library in the Windows registration database:

Registering OLE Controls

AfxOleRegisterControlClass	Adds the control's class to the registration database.
AfxOleRegisterPropertyPageClass	Adds a control property page to the registration database.
AfxOleRegisterTypeLib	Adds the control's type library to the registration database.
AfxOleUnregisterClass	Removes a control class or a property page class from the registration database.
AfxOleUnregisterTypeLib	Removes the control's type library from the registration database.

AfxOleRegisterTypeLib is typically called in a control DLL's implementation of **DllRegisterServer**. Similarly, **AfxOleUnregisterTypeLib** is called by **DllUnregisterServer**. **AfxOleRegisterControlClass**, **AfxOleRegisterPropertyPageClass**, and **AfxOleUnregisterClass** are typically called by the **UpdateRegistry** member function of a control's class factory or property page.

See Also

Concepts

MFC Macros and Globals

Class Factories and Licensing

To create an instance of your OLE control, a container application calls a member function of the control's class factory. Because your control is an actual OLE object, the class factory is responsible for creating instances of your control. Every OLE control class must have a class factory.

Another important feature of OLE controls is their ability to enforce a license. ControlWizard allows you to incorporate licensing during the creation of your control project. For more information on control licensing, see the article ActiveX Controls: Licensing An ActiveX Control.

The following table lists several macros and functions used to declare and implement your control's class factory and to license of your control.

Class Factories and Licensing

DECLARE_OLECREATE_EX	Declares the class factory for an OLE control or property page.
IMPLEMENT_OLECREATE_EX	Implements the control's **GetClassID** function and declares an instance of the class factory.
BEGIN_OLEFACTORY	Begins the declaration of any licensing functions.
END_OLEFACTORY	Ends the declaration of any licensing functions.
AfxVerifyLicFile	Verifies whether a control is licensed for use on a particular computer.

See Also

Concepts

MFC Macros and Globals

Persistence of OLE Controls

One capability of OLE controls is property persistence (or serialization), which allows the OLE control to read or write property values to and from a file or stream. A container application can use serialization to store a control's property values even after the application has destroyed the control. The property values of the OLE control can then be read from the file or stream when a new instance of the control is created at a later time.

Persistence of OLE Controls

PX_Blob	Exchanges a control property that stores binary large object (BLOB) data.
PX_Bool	Exchanges a control property of type **BOOL**.
PX_Color	Exchanges a color property of a control.
PX_Currency	Exchanges a control property of type **CY**.
PX_DataPath	Exchanges a control property of type **CDataPathProperty**.
PX_Double	Exchanges a control property of type **double**.
PX_Font	Exchanges a font property of a control.
PX_Float	Exchanges a control property of type **float**.
PX_IUnknown	Exchanges a control property of undefined type.
PX_Long	Exchanges a control property of type **long**.
PX_Picture	Exchanges a picture property of a control.
PX_Short	Exchanges a control property of type **short**.
PX_ULong	Exchanges a control property of type **ULONG**.
PX_UShort	Exchanges a control property of type **USHORT**.
PX_String	Exchanges a character string control property.
PX_VBXFontConvert	Exchanges a VBX control's font-related properties into an OLE control font property.

In addition, the AfxOleTypeMatchGuid global function is provided to test for a match between a TYPEDESC and a given GUID.

See Also

Concepts

MFC Macros and Globals

Internet Server API (ISAPI) Parse Maps

The Internet Server API, an extended open API set, provides you with the ability to create add-ons for, and run Internet server applications on, your Microsoft Internet Information Services. When a client sends a query to the Internet server, the server processes the query by sending it through a series of parsing macros in the parse map. The parse map maps the client queries to a CHttpServer-derived class's functions and parameters.

ISAPI Parse Maps

BEGIN_PARSE_MAP	Starts the definition of a parse map.
ON_PARSE_COMMAND	Parses the client's command
ON_PARSE_COMMAND_PARAMS	Defines a command to a CHttpServer object from a client.
DEFAULT_PARSE_COMMAND	Calls the default page that is identified by the *FnName* parameter.
END_PARSE_MAP	Ends the definition of a parse map.

See Also

Concepts

MFC Macros and Globals

Internet URL Parsing Globals

The Internet Server API, an extended open API set, provides you with the ability to create add-ons for, and run Internet server applications on, your Microsoft Internet Information Services. When a client sends a query to the Internet server, you can use one of the URL parsing globals to extract information about the client.

Internet URL Parsing Globals

AfxParseURL	Parses a URL string and returns the type of service and its components.
AfxParseURLEx	Parses a URL string and returns the type of service and its components, as well as providing the user name and password.

See Also

Concepts

MFC Macros and Globals

Internet Server API (ISAPI) Diagnostic Macros

The Microsoft Internet Information Services requires the same diagnostic services that MFC programs need; however, the programs written for the Internet Server do not require MFC. The ISAPI macros described below provide the same level of debugging functionality for both MFC programs and programs not written with MFC.

ISAPI Diagnostic Macros

ISAPIASSERT	Provides ASSERT functionality.
ISAPITRACE	Provides TRACE functionality.
ISAPITRACE0	Provides TRACE0 functionality.
ISAPITRACE1	Provides TRACE1 functionality.
ISAPITRACE2	Provides TRACE2 functionality.
ISAPITRACE3	Provides TRACE3 functionality.
ISAPIVERIFY	Provides VERIFY functionality.

See Also

Concepts

MFC Macros and Globals

DHTML Event Maps

The following macros can be used to handle DHTML events in CDHtmlDialog-derived classes.

DHTML Event Map Macros

BEGIN_DHTML_EVENT_MAP	Marks the start of the DHTML event map.
BEGIN_DHTML_EVENT_MAP_INLINE	Marks the start of the DHTML event map.
DECLARE_DHTML_EVENT_MAP	Declares the DHTML event map.
DHTML_EVENT	Used to handle an event at the document level for a single HTML element.
DHTML_EVENT_AXCONTROL	Used to handle an event fired by an ActiveX control.
DHTML_EVENT_CLASS	Used to handle an event at the document level for all HTML elements with a particular CSS class.
DHTML_EVENT_ELEMENT	Used to handle an event at the element level.
DHTML_EVENT_ONAFTERUPDATE	Used to handle the **onafterupdate** event from an HTML element.
DHTML_EVENT_ONBEFOREUPDATE	Used to handle the **onbeforeupdate** event from an HTML element.
DHTML_EVENT_ONBLUR	Used to handle the **onblur** event from an HTML element.
DHTML_EVENT_ONCHANGE	Used to handle the **onchange** event from an HTML element.
DHTML_EVENT_ONCLICK	Used to handle the **onclick** event from an HTML element.
DHTML_EVENT_ONDATAAVAILABLE	Used to handle the **ondataavailable** event from an HTML element.
DHTML_EVENT_ONDATASETCHANGED	Used to handle the **ondatasetchanged** event from an HTML element.
DHTML_EVENT_ONDATASETCOMPLETE	Used to handle the **ondatasetcomplete** event from an HTML element.
DHTML_EVENT_ONDBLCLICK	Used to handle the **ondblclick** event from an HTML element.
DHTML_EVENT_ONDRAGSTART	Used to handle the **ondragstart** event from an HTML element.
DHTML_EVENT_ONERRORUPDATE	Used to handle the **onerrorupdate** event from an HTML element.
DHTML_EVENT_ONFILTERCHANGE	Used to handle the **onfilterchange** event from an HTML element.
DHTML_EVENT_ONFOCUS	Used to handle the **onfocus** event from an HTML element.
DHTML_EVENT_ONHELP	Used to handle the **onhelp** event from an HTML element.
DHTML_EVENT_ONKEYDOWN	Used to handle the **onkeydown** event from an HTML element.
DHTML_EVENT_ONKEYPRESS	Used to handle the **onkeypress** event from an HTML element.

DHTML_EVENT_ONKEYUP	Used to handle the **onkeyup** event from an HTML element.
DHTML_EVENT_ONMOUSEDOWN	Used to handle the **onmousedown** event from an HTML element.
DHTML_EVENT_ONMOUSEMOVE	Used to handle the **onmousemove** event from an HTML element.
DHTML_EVENT_ONMOUSEOUT	Used to handle the **onmouseout** event from an HTML element.
DHTML_EVENT_ONMOUSEOVER	Used to handle the **onmouseover** event from an HTML element.
DHTML_EVENT_ONMOUSEUP	Used to handle the **onmouseup** event from an HTML element.
DHTML_EVENT_ONRESIZE	Used to handle the **onresize** event from an HTML element.
DHTML_EVENT_ONROWENTER	Used to handle the **onrowenter** event from an HTML element.
DHTML_EVENT_ONROWEXIT	Used to handle the **onrowexit** event from an HTML element.
DHTML_EVENT_ONSELECTSTART	Used to handle the **onselectstart** event from an HTML element.
DHTML_EVENT_TAG	Used to handle an event at the document level for all elements with a particular HTML tag.
END_DHTML_EVENT_MAP	Marks the end of the DHTML event map.

See Also

Concepts

MFC Macros and Globals

DHTML Editing Command Maps

The following macros can be used to map DHTML editing commands in CHtmlEditView-derived classes. For an example of their use, see HTMLEdit Sample.

DHTML Editing Command Map Macros

DECLARE_DHTMLEDITING_CMDMAP	Declares a DHTML editing command map in a class.
BEGIN_DHTMLEDITING_CMDMAP	Starts the definition of a DHTML editing command map within a class.
END_DHTMLEDITING_CMDMAP	Marks the end of a DHTML editing command map.
DHTMLEDITING_CMD_ENTRY	Maps a command ID to an HTML editing command.
DHTMLEDITING_CMD_ENTRY_FUNC	Maps a command ID to an HTML editing command and message handler.
DHTMLEDITING_CMD_ENTRY_TYPE	Maps a command ID to an HTML editing command and user interface element.
DHTMLEDITING_CMD_ENTRY_FUNC_TYPE	Maps a command ID to an HTML editing command, message handler, and user interface element.

See Also

Concepts

MFC Macros and Globals

Multipage DHTML and URL Event Maps

The following macros can be used to handle DHTML events in CMultiPageDHtmlDialog-derived classes.

URL Event Map Macros

BEGIN_DHTML_URL_EVENT_MAP	Marks the start of the multipage DHTML and URL event map.
BEGIN_EMBED_DHTML_EVENT_MAP	Marks the start of an embedded DHTML event map.
BEGIN_URL_ENTRIES	Marks the start of a URL event entry map.
DECLARE_DHTML_URL_EVENT_MAP	Declares the multipage DHTML and URL event map.
END_DHTML_URL_EVENT_MAP	Marks the end of the multipage DHTML and URL event map.
END_EMBED_DHTML_EVENT_MAP	Marks the end of an embedded DHTML event map.
END_URL_ENTRIES	Marks the end of a URL event entry map.
URL_EVENT_ENTRY	Maps a URL or HTML resource to a page in a multipage dialog.

See Also

Concepts

MFC Macros and Globals

Macros, Global Functions, and Global Variables

The topics in this section provide descriptions of the global functions, global variables, and macros in the MFC library.

Note
Many global functions start with the prefix "Afx" — but some, such as the dialog data exchange (DDX) functions and many of the database functions, deviate from this convention. All global variables start with the prefix "afx". Macros do not start with a ny particular prefix, but they are written all in uppercase.

The MFC library and the Active Template Library (ATL) share string conversion macros. See String Conversion Macros in the ATL documentation for a discussion of these macros.

For information on the debug version of the C run-time library and diagnostic functions, see Debug Routines, in the *Run-Time Library Reference*.

See Also

Concepts

MFC Macros and Globals

AfxAbort

The default termination function supplied by MFC.

```
void AfxAbort( );
```

Remarks

AfxAbort is called internally by MFC member functions when there is a fatal error, such as an uncaught exception that cannot be handled. You can call **AfxAbort** in the rare case when you encounter a catastrophic error from which you cannot recover.

Example

See the example for CATCH.

See Also

Concepts

MFC Macros and Globals

AfxBeginThread

Call this function to create a new thread.

```
CWinThread* AfxBeginThread(
   AFX_THREADPROC pfnThreadProc,
   LPVOID pParam,
   int nPriority = THREAD_PRIORITY_NORMAL,
   UINT nStackSize = 0,
   DWORD dwCreateFlags = 0,
   LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL
);
CWinThread* AfxBeginThread(
   CRuntimeClass* pThreadClass,
   int nPriority = THREAD_PRIORITY_NORMAL,
   UINT nStackSize = 0,
   DWORD dwCreateFlags = 0,
   LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL
);
```

Parameters

pfnThreadProc

Points to the controlling function for the worker thread. Cannot be **NULL**. This function must be declared as follows:

```
UINT __cdecl MyControllingFunction( LPVOID pParam );
```

pThreadClass

The RUNTIME_CLASS of an object derived from CWinThread.

pParam

Parameter to be passed to the controlling function as shown in the parameter to the function declaration in *pfnThreadProc*.

nPriority

The desired priority of the thread. If 0, the same priority as the creating thread will be used. For a full list and description of the available priorities, see SetThreadPriority in the *Platform SDK*.

nStackSize

Specifies the size in bytes of the stack for the new thread. If 0, the stack size defaults to the same size stack as the creating thread.

dwCreateFlags

Specifies an additional flag that controls the creation of the thread. This flag can contain one of two values:

- **CREATE_SUSPENDED** Start the thread with a suspend count of one. Use **CREATE_SUSPENDED** if you want to initialize any member data of the **CWinThread** object, such as m_bAutoDelete or any members of your derived class, before the thread starts running. Once your initialization is complete, use CWinThread::ResumeThread to start the thread running. The thread will not execute until **CWinThread::ResumeThread** is called.
- **0** Start the thread immediately after creation.

lpSecurityAttrs

Points to a SECURITY_ATTRIBUTES structure that specifies the security attributes for the thread. If **NULL**, the same security attributes as the creating thread will be used. For more information on this structure, see the *Platform SDK*.

Return Value

Pointer to the newly created thread object, or **NULL** if a failure occurs.

Remarks

The first form of **AfxBeginThread** creates a worker thread. The second form creates a user-interface thread.

AfxBeginThread creates a new **CWinThread** object, calls its CreateThread function to start executing the thread, and returns a pointer to the thread. Checks are made throughout the procedure to make sure all objects are deallocated properly should any part of the creation fail. To end the thread, call AfxEndThread from within the thread, or return from the controlling function of the worker thread.

Multithreading must be enabled by the application; otherwise, this function will fail. For more information on enabling multithreading, refer to /MD, /MT, /LD (Use Run-Time Library) under *Visual C++ Compiler Options*.

For more information on **AfxBeginThread**, see the articles Multithreading: Creating Worker Threads and Multithreading: Creating User-Interface Threads.

Example

See the example for CSocket::Attach.

See Also
Reference
AfxGetThread
Concepts
MFC Macros and Globals

AfxCheckError

This function tests the passed **SCODE** to see if it is an error.

```
void AFXAPI AfxCheckError(
   SCODE sc
);
throw CMemoryException*
throw COleException*
```

Remarks

If it is an error, the function throws an exception. If the passed **SCODE** is **E_OUTOFMEMORY**, the function throws a CMemoryException by calling AfxThrowMemoryException. Otherwise, the function throws a COleException by calling AfxThrowOleException.

This function can be used to check the return values of calls to OLE functions in your application. By testing the return value with this function in your application, you can properly react to error conditions with a minimal amount of code.

Note
This function has the same effect in debug and non-debug builds.

Example

```
LPDISPATCH pDisp = NULL;
AfxCheckError(CoCreateInstance(CLSID,
 NULL, CLSCTX_LOCAL_SERVER, IID_IDispatch,
 (LPVOID) &pDisp));
// if there was an error, an exception has already been thrown
// we can start using the returned pointer
COleDispatchDriver disp(pDisp);
// and so on...
```

See Also

Concepts

MFC Macros and Globals

AfxCheckMemory

This function validates the free memory pool and prints error messages as required.

```
BOOL AfxCheckMemory( );
```

Return Value

Nonzero if no memory errors; otherwise 0.

Remarks

If the function detects no memory corruption, it prints nothing.

All memory blocks currently allocated on the heap are checked, including those allocated by **new** but not those allocated by direct calls to underlying memory allocators, such as the **malloc** function or the **GlobalAlloc** Windows function. If any block is found to be corrupted, a message is printed to the debugger output.

If you include the line

```
#define new DEBUG_NEW
```

in a program module, then subsequent calls to **AfxCheckMemory** show the filename and line number where the memory was allocated.

Note

If your module contains one or more implementations of serializable classes, then you must put the `#define` line after the last **IMPLEMENT_SERIAL** macro call.

This function works only in the Debug version of MFC.

Example

```
// example for AfxCheckMemory
CAge* pcage = new CAge( 21 );  // CAge is derived from CObject.
Age* page = new Age( 22 );     // Age is NOT derived from CObject.
*(((char*) pcage) - 1) = 99;   // Corrupt preceding guard byte
*(((char*) page) - 1) = 99;    // Corrupt preceding guard byte
AfxCheckMemory();
```

The results from the program are as follows:

```
memory check error at $0067495F = $63, should be $FD
DAMAGE: before Non-Object block at $00674960
Non-Object allocated at file test02.cxx(48)
Non-Object located at $00674960 is 2 bytes long
memory check error at $00674905 = $63, should be $FD
DAMAGE: before Object block at $00674906
Object allocated at file test02.cxx(47)
Object located at $00674906 is 6 bytes long
```

See Also

Concepts

MFC Macros and Globals

AFX_COMCTL32_IF_EXISTS

Determines whether the Common Controls library implements the specified API.

```
AFX_COMCTL32_IF_EXISTS(
   proc
);
```

Parameters

proc

Pointer to a null-terminated string containing the function name, or specifies the function's ordinal value. If this parameter is an ordinal value, it must be in the low-order word; the high-order word must be zero. This parameter must be in Unicode.

Remarks

Use this macro to determine whether the Common Controls library the function specified by *proc* (instead of calling GetProcAddress).

Requirements

afxcomctl32.h, afxcomctl32.inl

See Also

Reference

AFX_COMCTL32_IF_EXISTS2

Concepts

Isolation of the MFC Common Controls Library

AFX_COMCTL32_IF_EXISTS2

Determines whether the Common Controls library implements the specified API (this is the Unicode version of AFX_COMCTL32_IF_EXISTS).

```
AFX_COMCTL32_IF_EXISTS2(
      proc
);
```

Parameters

proc

Pointer to a null-terminated string containing the function name, or specifies the function's ordinal value. If this parameter is an ordinal value, it must be in the low-order word; the high-order word must be zero. This parameter must be in Unicode.

Remarks

Use this macro to determine whether the Common Controls library the function specified by *proc* (instead of calling GetProcAddress). This macro is the Unicode version of **AFX_COMCTL32_IF_EXISTS**.

Requirements

afxcomctl32.h, afxcomctl32.inl

See Also

Reference

AFX_COMCTL32_IF_EXISTS

Concepts

Isolation of the MFC Common Controls Library

AfxConnectionAdvise

Call this function to establish a connection between a source, specified by *pUnkSrc*, and a sink, specified by *pUnkSink*.

```
BOOL AFXAPI AfxConnectionAdvise(
   LPUNKNOWN pUnkSrc,
   REFIID iid,
   LPUNKNOWN pUnkSink,
   BOOL bRefCount,
   DWORD FAR* pdwCookie
);
```

Parameters

pUnkSrc

A pointer to the object that calls the interface.

pUnkSink

A pointer to the object that implements the interface.

iid

The interface ID of the connection.

bRefCount

TRUE indicates that creating the connection should cause the reference count of *pUnkSink* to be incremented. **FALSE** indicates that the reference count should not be incremented.

pdwCookie

A pointer to a **DWORD** where a connection identifier is returned. This value should be passed as the *dwCookie* parameter to **AfxConnectionUnadvise** when disconnecting the connection.

Return Value

Nonzero if a connection was established; otherwise 0.

Example

```
//CMySink is a CCmdTarget-derived class supporting automation.
//Instantiate the sink class and hold a pointer to it.
m_pSink = new CMySink();

//Get a pointer to sinks IUnknown, no AddRef done. CMySink implements only
//dispinterface and the IUnknown and IDispatch pointers will be same.
LPUNKNOWN pUnkSink = m_pSink->GetIDispatch(FALSE);

//Establish a connection between source and sink.
//m_pUnkSrc is IUnknown of server obtained by CoCreateInstance().
//m_dwCookie is a cookie identifying the connection, and is needed
//to terminate this connection.
AfxConnectionAdvise(m_pUnkSrc, IID_MYEVENT, pUnkSink, FALSE,
   &m_dwCookie);
```

See Also

Reference

AfxConnectionUnadvise

Concepts

MFC Macros and Globals

AfxConnectionUnadvise

Call this function to disconnect a connection between a source, specified by *pUnkSrc*, and a sink, specified by *pUnkSink*.

```
BOOL AFXAPI AfxConnectionUnadvise(
   LPUNKNOWN pUnkSrc,
   REFIID iid,
   LPUNKNOWN pUnkSink,
   BOOL bRefCount,
   DWORD dwCookie
);
```

Parameters

pUnkSrc

A pointer to the object that calls the interface.

pUnkSink

A pointer to the object that implements the interface.

iid

The interface ID of the connection point interface.

bRefCount

TRUE indicates that disconnecting the connection should cause the reference count of *pUnkSink* to be decremented. **FALSE** indicates that the reference count should not be decremented.

dwCookie

The connection identifier returned by **AfxConnectionAdvise**.

Return Value

Nonzero if a connection was disconnected; otherwise 0.

Example

```
//m_pSink is a ptr to CCmdTarget-derived class supporting automation.
//Get a pointer to sinks IUnknown, no AddRef done.
LPUNKNOWN pUnkSink = m_pSink->GetIDispatch(FALSE);

//Terminate a connection between source and sink.
//m_pUnkSrc is IUnknown of server obtained by CoCreateInstance().
//m_dwCookie is a value obtained through AfxConnectionAdvise().
AfxConnectionUnadvise(m_pUnkSrc, IID_MYEVENT, pUnkSink, FALSE,
   m_dwCookie);
```

See Also

Reference

AfxConnectionAdvise

Concepts

MFC Macros and Globals

AfxDaoInit

This function initializes the DAO database engine.

```
void AfxDaoInit( );
throw(
   CDaoException*
);
```

Remarks

In most cases, you don't need to call **AfxDaoInit** because the application automatically calls it when it is needed.

For related information, and for an example of calling **AfxDaoInit**, see Technical Note 54.

See Also

Reference

AfxDaoTerm

Concepts

MFC Macros and Globals

AfxDaoTerm

This function terminates the DAO database engine.

```
void AfxDaoTerm( );
```

Remarks

Typically, you only need to call this function in a regular DLL; an application will automatically call **AfxDaoTerm** when it is needed.

In regular DLLs, call **AfxDaoTerm** before the **ExitInstance** function, but after all MFC DAO objects have been destroyed.

For related information, see Technical Note 54.

See Also

Reference

AfxDaoInit

Concepts

MFC Macros and Globals

AfxDebugBreak (MFC)

Call this function to cause a break (at the location of the call to **AfxDebugBreak**) in the execution of the debug version of your MFC application.

```
void AfxDebugBreak( );
```

Remarks

AfxDebugBreak has no effect in release versions of an MFC application and should be removed. This function should only be used in MFC applications. Use the Win32 API version, **DebugBreak**, to cause a break in non-MFC applications.

See Also

Concepts

MFC Macros and Globals

AfxDbInitModule

For MFC database (or DAO) support from a regular DLL that is dynamically linked to MFC, add a call to this function in your regular DLL's **CWinApp::InitInstance** function to initialize the MFC database DLL.

```
void AFXAPI AfxDbInitModule( );
```

Remarks

Make sure this call occurs before any base-class call or any added code which accesses the MFC database DLL. The MFC database DLL is an extension DLL; in order for an extension DLL to get wired into a **CDynLinkLibrary** chain, it must create a **CDynLinkLibrary** object in the context of every module that will be using it. **AfxDbInitModule** creates the **CDynLinkLibrary** object in your regular DLL's context so that it gets wired into the **CDynLinkLibrary** object chain of the regular DLL.

Requirements

Header: <afxdll_.h>

See Also

Concepts

MFC Macros and Globals

AfxDoForAllClasses

Calls the specified iteration function for all serializable **CObject**-derived classes in the application's memory space.

```
void AFXAPI AfxDoForAllClasses(
   void (*pfn
)(const CRuntimeClass* pClass,
   void* pContext
),
   void* pContext
);
```

Parameters

pfn

Points to an iteration function to be called for each class. The function arguments are a pointer to a **CRuntimeClass** object and a void pointer to extra data that the caller supplies to the function.

pContext

Points to optional data that the caller can supply to the iteration function. This pointer can be **NULL**.

Remarks

Serializable **CObject**-derived classes are classes derived using the **DECLARE_SERIAL** macro. The pointer that is passed to **AfxDoForAllClasses** in *pContext* is passed to the specified iteration function each time it is called.

Note
This function works only in the Debug version of MFC.

Example

```
void DoForAllClasses(const CRuntimeClass* pClass, void* pContext)
{
   ASSERT(pContext != NULL);
   CString *pStr = (CString *)pContext;

   *pStr += pClass->m_lpszClassName;
   *pStr += _T("\n");
}

   CString cStr;
   AfxDoForAllClasses(DoForAllClasses, &cStr);
   AfxMessageBox(cStr);
```

See Also

Reference

DECLARE_SERIAL

Concepts

MFC Macros and Globals

AfxDoForAllObjects

Executes the specified iteration function for all objects derived from **CObject** that have been allocated with **new**.

```
void AfxDoForAllObjects(
   void (*pfn
)(CObject* pObject,
   void* pContext
),
   void* pContext
);
```

Parameters

pfn

Points to an iteration function to execute for each object. The function arguments are a pointer to a **CObject** and a void pointer to extra data that the caller supplies to the function.

pContext

Points to optional data that the caller can supply to the iteration function. This pointer can be **NULL**.

Remarks

Stack, global, or embedded objects are not enumerated. The pointer passed to **AfxDoForAllObjects** in *pContext* is passed to the specified iteration function each time it is called.

Note
This function works only in the Debug version of MFC.

Example

```
#ifdef _DEBUG
void DoForAllObjects(CObject* pObject, void* pContext)
{
   int *pnCount = (int *) pContext;

   pObject->AssertValid();
   if (pnCount != NULL)
         (*pnCount)++;
}
#endif // _DEBUG

   //AfxDoForAllObjects will call the function DoForAllObjects
   //For each CObject-derived object that allocated on the heap
   int nCount = 0;
   AfxDoForAllObjects( DoForAllObjects, &nCount );
   TRACE("%d Objects Checked\n", nCount);
```

See Also

Concepts

MFC Macros and Globals

AfxDrawDitheredBitmap

Draws a bitmap, replacing its background with a dithered (checker) pattern.

```
void AFXAPI AfxDrawDitheredBitmap(
   CDC *pDC,
   int x,
   int y,
   const CBitmap &rSrc,
   COLORREF cr1,
   COLORREF cr2
);
```

Parameters

pDC

Points to the destination DC.

x

The destination x-coordinate.

y

The destination y-coordinate.

rSrc

The source bitmap.

cr1

One of the two dither colors, typically white.

cr2

The other dither color, typically light gray (COLOR_MENU).

Remarks

The source bitmap is drawn on the destination DC with a two-color (*cr1* and *cr2*) checkered pattern replacing the bitmap's background. The background of the source bitmap is defined as its white pixels and all pixels matching the color of the pixel in the upper-left corner of the bitmap.

Example

```
void CProject1View::OnDraw(CDC* pDC)
{
   CBitmap bm;
   bm.LoadBitmap(IDB_BITMAP1);
   AfxDrawDitheredBitmap(pDC, 10, 50, bm, RGB(255,255,255),
      GetSysColor(COLOR_MENU));
}
```

See Also

Reference

AfxGetDitheredBitmap

AfxDrawGrayBitmap

Concepts

MFC Macros and Globals

Other Resources

Gray and Dithered Bitmap Functions

AfxDrawGrayBitmap

Draws a gray version of a bitmap.

```
void AFXAPI AfxDrawGrayBitmap(
   CDC *pDC,
   int x,
   int y,
   const CBitmap &rSrc,
   COLORREF crBackground
);
```

Parameters

pDC

Points to the destination DC.

x

The destination x-coordinate.

y

The destination y-coordinate.

rSrc

The source bitmap.

crBackground

The new background color (typically gray, such as COLOR_MENU).

Remarks

A bitmap drawn with **AfxDrawGrayBitmap** will have the appearance of a disabled control.

Example

```
void CProject1View::OnDraw(CDC* pDC)
{
   CBitmap bm;
   bm.LoadBitmap(IDB_BITMAP1);
   AfxDrawGrayBitmap(pDC, 10, 50, bm, GetSysColor(COLOR_MENU));
}
```

See Also

Reference

AfxGetGrayBitmap

AfxDrawDitheredBitmap

Concepts

MFC Macros and Globals

Other Resources

Gray and Dithered Bitmap Functions

afxDump (CDumpContext in MFC)

Provides basic object-dumping capability in your application.

```
CDumpContext afxDump;
```

Remarks

afxDump is a predefined CDumpContext object that allows you to send **CDumpContext** information to the debugger output window or to a debug terminal. Typically, you supply **afxDump** as a parameter to **CObject::Dump**.

Under Windows NT and all versions of Windows, **afxDump** output is sent to the Output-Debug window of Visual C++ when you debug your application.

This variable is defined only in the Debug version of MFC. For more information on **afxDump**, see Debugging MFC Applications. Technical Note 7 and Technical Note 12 contain additional information.

Note
This function works only in the Debug version of MFC.

Example

```
// example for afxDump
CPerson myPerson = new CPerson;
// set some fields of the CPerson object...
//..
// now dump the contents
#ifdef _DEBUG
afxDump << "Dumping myPerson:\n";
myPerson->Dump( afxDump );
afxDump << "\n";
#endif
```

See Also

Reference

CObject::Dump

AfxDump (MFC)

Concepts

MFC Macros and Globals

AfxDump (MFC)

Call this function while in the debugger to dump the state of an object while debugging.

```
void AfxDump(
   const CObject* pOb
);
```

Parameters

pOb

A pointer to an object of a class derived from **CObject**.

Remarks

AfxDump calls an object's **Dump** member function and sends the information to the location specified by the **afxDump** variable. **AfxDump** is available only in the Debug version of MFC.

Your program code should not call **AfxDump**, but should instead call the **Dump** member function of the appropriate object.

See Also

Reference

CObject::Dump

afxDump (CDumpContext in MFC)

AfxDumpStack

This global function can be used to generate an image of the current stack.

```
void AFXAPI AfxDumpStack(
   DWORD dwTarget = AFX_STACK_DUMP_TARGET_DEFAULT
);
```

Parameters

dwTarget

Indicates the target of the dump output. Possible values, which can be combined using the bitwise-OR (**|**) operator, are as follows:

- **AFX_STACK_DUMP_TARGET_TRACE** Sends output by means of the TRACE macro. The **TRACE** macro generates output in debug builds only; it generates no output in release builds. Also, **TRACE** can be redirected to other targets besides the debugger.
- **AFX_STACK_DUMP_TARGET_DEFAULT** Sends dump output to the default target. For a debug build, output goes to the **TRACE** macro. In a release build, output goes to the Clipboard.
- **AFX_STACK_DUMP_TARGET_CLIPBOARD** Sends output to the Clipboard only. The data is placed on the Clipboard as plain text using the **CF_TEXT** Clipboard format.
- **AFX_STACK_DUMP_TARGET_BOTH** Sends output to the Clipboard and to the **TRACE** macro, simultaneously.
- **AFX_STACK_DUMP_TARGET_ODS** Sends output directly to the debugger by means of the Win32 function **OutputDebugString()**. This option will generate debugger output in both debug and release builds when a debugger is attached to the process. **AFX_STACK_DUMP_TARGET_ODS** always reaches the debugger (if it is attached) and cannot be redirected.

Remarks

The example below reflects a single line of the output generated from calling **AfxDumpStack** from a button handler in an MFC dialog application:

```
=== begin AfxDumpStack output ===

00427D55: DUMP2\DEBUG\DUMP2.EXE! void AfxDumpStack(unsigned long) + 181 bytes
0040160B: DUMP2\DEBUG\DUMP2.EXE! void CDump2Dlg::OnClipboard(void) + 14 bytes
0044F884: DUMP2\DEBUG\DUMP2.EXE! int _AfxDispatchCmdMsg(class CCmdTarget *,
unsigned int,int,void ( CCmdTarget::*)(void),void *,unsigned int,struct AFX_CMDHANDLE
0044FF7B: DUMP2\DEBUG\DUMP2.EXE! virtual int CCmdTarget::OnCmdMsg(unsigned
int,int,void *,struct AFX_CMDHANDLERINFO *) + 626 bytes
00450C71: DUMP2\DEBUG\DUMP2.EXE! virtual int CDialog::OnCmdMsg(unsigned
int,int,void *,struct AFX_CMDHANDLERINFO *) + 36 bytes
00455B27: DUMP2\DEBUG\DUMP2.EXE! virtual int CWnd::OnCommand(unsigned
int,long) + 312 bytes
00454D3D: DUMP2\DEBUG\DUMP2.EXE! virtual int CWnd::OnWndMsg(unsigned
int,unsigned int,long,long *) + 83 bytes
00454CC0: DUMP2\DEBUG\DUMP2.EXE! virtual long CWnd::WindowProc(unsigned
int,unsigned int,long) + 46 bytes
004528D9: DUMP2\DEBUG\DUMP2.EXE! long AfxCallWndProc(class CWnd *,struct
HWND__ *,unsigned int,unsigned int,long) + 237 bytes
00452D34: DUMP2\DEBUG\DUMP2.EXE! long AfxWndProc(struct HWND__ *,unsigned
int,unsigned int,long) + 129 bytes
BFF73663: WINDOWS\SYSTEM\KERNEL32.DLL! ThunkConnect32 + 2148 bytes
BFF928E0: WINDOWS\SYSTEM\KERNEL32.DLL! UTUnRegister + 2492 bytes

=== end AfxDumpStack() output ===
```

Each line in the output above indicates the address of the last function call, the full path name of the module that contains the function call, and the function prototype called. If the function call on the stack does not happen at the exact address of the

function, an offset of bytes is shown.

For example, the following table describes the first line of the above output:

Output	Description
`00427D55:`	The return address of the last function call.
`DUMP2\DEBUG\DUMP2.EXE!`	The full path name of the module that contains the function call.
`void AfxDumpStack(unsigned long)`	The function prototype called.
`+ 181 bytes`	The offset in bytes from the address of the function prototype (in this case, `void AfxDumpStack(unsigned long)`) to the return address (in this case, `00427D55`).

AfxDumpStack is available in debug and nondebug versions of the MFC libraries; however, the function is always linked statically, even when your executable file uses MFC in a shared DLL. In shared-library implementations, the function is found in the MFCS42.LIB library (and its variants).

To use this function successfully:

- The file IMAGEHLP.DLL must be on your path. If you do not have this DLL, the function will display an error message. See Image Help Library for information on the function set provided by IMAGEHLP.
- The modules that have frames on the stack must include debugging information. If they do not contain debugging information, the function will still generate a stack trace, but the trace will be less detailed.

See Also

Reference

afxDump (CDumpContext in MFC)

Concepts

MFC Macros and Globals

AfxEnableControlContainer

Call this function in your application object's **InitInstance** function to enable support for containment of OLE controls.

```
void AfxEnableControlContainer( );
```

Remarks

For more information about OLE controls (now called ActiveX controls), see ActiveX Control Topics.

See Also

Concepts

MFC Macros and Globals

AfxEnableMemoryTracking

Diagnostic memory tracking is normally enabled in the Debug version of MFC.

```
BOOL AfxEnableMemoryTracking(
   BOOL bTrack
);
```

Parameters

bTrack

Setting this value to **TRUE** turns on memory tracking; **FALSE** turns it off.

Return Value

The previous setting of the tracking-enable flag.

Remarks

Use this function to disable tracking on sections of your code that you know are allocating blocks correctly.

For more information on **AfxEnableMemoryTracking**, see Debugging MFC Applications.

Note
This function works only in the Debug version of MFC.

Example

```
BOOL CWinMyApp::InitInstance()
{
  #ifdef _DEBUG
    // Disable tracking of memory for the scope of the InitInstance()
    AfxEnableMemoryTracking(FALSE);
  #endif  // _DEBUG

    ...
    ...

  #ifdef _DEBUG
    // Re-enable tracking of memory
    AfxEnableMemoryTracking(TRUE);
  #endif  // _DEBUG
}
```

See Also

Concepts

MFC Macros and Globals

AfxEndThread

Call this function to terminate the currently executing thread.

```
void AFXAPI AfxEndThread(
   UINT nExitCode,
   BOOL bDelete = TRUE
);
```

Parameters

nExitCode

Specifies the exit code of the thread.

bDelete

Deletes the thread object from memory.

Remarks

Must be called from within the thread to be terminated.

For more information on **AfxEndThread**, see the article Multithreading: Terminating Threads.

See Also

Reference

AfxBeginThread

Concepts

MFC Macros and Globals

AFX_EXT_CLASS

Extension DLLs use the macro **AFX_EXT_CLASS** to export classes; the executables that link to the extension DLL use the macro to import classes.

Remarks

With the **AFX_EXT_CLASS** macro, the same header file(s) used to build the extension DLL can be used with the executables that link to the DLL.

In the header file for your DLL, add the **AFX_EXT_CLASS** keyword to the declaration of your class as follows:

```
class AFX_EXT_CLASS CMyClass : public CDocument
{
// <body of class>
};
```

For more information, see Export and Import Using AFX_EXT_CLASS.

See Also

Concepts

MFC Macros and Globals

AfxExtractSubString

This global function can be used to extract a substring from a given source string.

```
BOOL AFXAPI AfxExtractSubString (
   CString& rString,
   LPCTSTR lpszFullString,
   int iSubString,
   TCHAR chSep = '\n'
);
```

Parameters

rString

- Reference to a CString object that will receive an individual substring.

lpszFullString

- String containing the full text of the string to extract from.

iSubString

- Zero-based index of the substring to extract from *lpszFullString*.

chSep

- Separator character used to delimit substrings.

Return Value

TRUE if the function successfully extracted the substring at the provided index; otherwise, **FALSE**.

Remarks

This function is useful for extracting multiple substrings from a source string when a known single character separates each substring. This function searches from the beginning of the *lpszFullString* parameter each time it is called.

This function will return FALSE if either *lpszFullString* is set to **NULL** or the function reaches the end of *lpszFullString* without finding *iSubString*+1 occurrences of the specified separator character. The *rString* parameter will not be modified from its original value if *lpszFullString* was set to **NULL**; otherwise, the *rString* parameter is set to the empty string if the substring could not be extracted for the specified index.

Example

```
// The following example extracts a series of name, value pairs from a
// given source string:

// Input string consisting of a number of name, value pairs
LPCTSTR lpszSource = "\"Name\"=\"John Smith\"\n\"Company\"=\"Contoso, Ltd\"\n\"Salary\"=\"2
5,000\"";

CString strNameValue; // an individual name, value pair

int i = 0; // substring index to extract
while (AfxExtractSubString(strNameValue, lpszSource, i))
{
   // Prepare to move to the next substring
   i++;

   CString strName, strValue; // individual name and value elements

   // Attempt to extract the name element from the pair
   if (!AfxExtractSubString(strName, strNameValue, 0, '='))
   {
      // Pass an error message to the debugger for display
```

```
        OutputDebugString("Error extracting name\r\n");
        continue;
    }

    // Attempt to extract the value element from the pair
    if (!AfxExtractSubString(strValue, strNameValue, 1, '='))
    {
        // Pass an error message to the debugger for display
        OutputDebugString("Error extracting value element\r\n");
        continue;
    }

    // Pass the name, value pair to the debugger for display
    CString strOutput = strName + " equals " + strValue + "\r\n";
    OutputDebugString(strOutput);
}
```

Requirements

Header: <afxwin.h>

See Also

Concepts

MFC Macros and Globals

Other Resources

Using CString

AfxFindResourceHandle

Use **AfxFindResourceHandle** to walk the resource chain and locate a specific resource by resource ID and resource type.

```
HINSTANCE AFXAPI AfxFindResourceHandle(
   LPCTSTR lpszName,
   LPCTSTR lpszType
);
```

Parameters

lpszName

A pointer to a string containing the resource ID.

lpszType

A pointer to the type of resource. For a list of resource types, see FindResource in the *Platform SDK*.

Return Value

A handle to the module that contains the resource.

Remarks

AfxFindResourceHandle finds the specific resource and returns a handle to the module that contains the resource. The resource might be in any extension DLL you have loaded. **AfxFindResourceHandle** tells you which one has the resource.

See Also

Reference

AfxGetResourceHandle

Concepts

MFC Macros and Globals

AfxFormatString1

Loads the specified string resource and substitutes the characters "%1" for the string pointed to by *lpsz1*.

```
void AfxFormatString1(
   CString& rString,
   UINT nIDS,
   LPCTSTR lpsz1
);
```

Parameters

rString

A reference to a **CString** object that will contain the resultant string after the substitution is performed.

nIDS

The resource ID of the template string on which the substitution will be performed.

lpsz1

A string that will replace the format characters "%1" in the template string.

Remarks

The newly formed string is stored in *rString*. For example, if the string in the string table is "File %1 not found", and *lpsz1* is equal to "C:\MYFILE.TXT", then *rString* will contain the string "File C:\MYFILE.TXT not found". This function is useful for formatting strings sent to message boxes and other windows.

If the format characters "%1" appear in the string more than once, multiple substitutions will be made.

Example

```
void DisplayFileNotFoundMessage( LPCTSTR pszFileName )
{
   CString strMessage;

   // The IDS_FILENOTFOUND string resource contains "Error: File %1 not found"
   AfxFormatString1( strMessage, IDS_FILENOTFOUND, pszFileName );
   // In the previous call, substitute the actual file name for the
   // %1 placeholder
   AfxMessageBox( strMessage );  // Display the error message
}
```

See Also

Reference

AfxFormatString2

Concepts

MFC Macros and Globals

AfxFormatString2

Loads the specified string resource and substitutes the characters "%1" and "%2" for the strings pointed to by *lpsz1* and *lpsz2*.

```
void AfxFormatString2(
   CString& rString,
   UINT nIDS,
   LPCTSTR lpsz1,
   LPCTSTR lpsz2
);
```

Parameters

rString

A reference to the **CString** that will contain the resultant string after the substitution is performed.

nIDS

The string table ID of the template string on which the substitution will be performed.

lpsz1

A string that will replace the format characters "%1" in the template string.

lpsz2

A string that will replace the format characters "%2" in the template string.

Remarks

The newly formed string is stored in *rString*. For example, if the string in the string table is "File %1 not found in directory %2", *lpsz1* points to "MYFILE.TXT", and *lpsz2* points to "C:\MYDIR", then *rString* will contain the string "File MYFILE.TXT not found in directory C:\MYDIR"

If the format characters "%1" or "%2" appear in the string more than once, multiple substitutions will be made. They do not have to be in numerical order.

Example

```
void DisplayFileNotFoundMessage( LPCTSTR pszFileName, LPCTSTR pszDirectory )
{
   CString strMessage;

   // The IDS_FILENOTFOUND string resource contains "Error: File %1 not
   // found in directory %2"
   AfxFormatString2( strMessage, IDS_FIlENOTFOUND, pszFileName, pszDirectory );
   // In the previous call, substitute the actual file and directory
   // names into the message string
   AfxMessageBox( strMessage );  // Display the error message
}
```

See Also

Reference

AfxFormatString1

Concepts

MFC Macros and Globals

AfxFreeLibrary

Both **AfxFreeLibrary** and **AfxLoadLibrary** maintain a reference count for each loaded library module.

```
BOOL AFXAPI AfxFreeLibrary(
   HINSTANCE hInstLib
);
```

Parameters

hInstLib

A handle of the loaded library module. AfxLoadLibrary returns this handle.

Return Value

TRUE if the function succeeds; otherwise, **FALSE**.

Remarks

AfxFreeLibrary decrements the reference count of the loaded dynamic-link library (DLL) module. When the reference count reaches zero, the module is unmapped from the address space of the calling process and the handle is no longer valid. This reference count is incremented each time **AfxLoadLibrary** is called.

Before unmapping a library module, the system enables the DLL to detach from the processes using it. Doing so gives the DLL an opportunity to clean up resources allocated on behalf of the current process. After the entry-point function returns, the library module is removed from the address space of the current process.

Use **AfxLoadLibrary** to map a DLL module.

Be sure to use **AfxFreeLibrary** and **AfxLoadLibrary** (instead of the Win32 functions **FreeLibrary** and **LoadLibrary**) if your application uses multiple threads. Using **AfxLoadLibrary** and **AfxFreeLibrary** ensures that the startup and shutdown code that executes when the extension DLL is loaded and unloaded does not corrupt the global MFC state.

Example

See the example for AfxLoadLibrary.

See Also

Reference

AfxLoadLibrary

Concepts

MFC Macros and Globals

AfxGetAmbientActCtx

Use this function to get the current state of the per-module state flag, which affects the WinSxS behavior of MFC.

```
BOOL AFXAPI AfxGetAmbientActCtx();
```

Return Value

Module state flag current value.

Remarks

When the flag is set (which is the default) and a thread enters an MFC module (see AFX_MANAGE_STATE), the context of the module is activated.

If the flag is not set, the context of the module is not activated on entry.

The context of a module is determined from its manifest, usually embedded in module resources.

See Also

Reference

AFX_MANAGE_STATE
AfxSetAmbientActCtx

Concepts

MFC Macros and Globals
Managing the State Data of MFC Modules

AfxGetApp

The pointer returned by this function can be used to access application information such as the main message-dispatch code or the topmost window.

```
CWinApp* AFXAPI AfxGetApp( );
```

Return Value

A pointer to the single **CWinApp** object for the application.

Example

```
// Print the application's executable filename.
TRACE("Executable filename = %s\n", AfxGetApp()->m_pszExeName);
```

See Also

Concepts

MFC Macros and Globals

AfxGetAppName

The string returned by this function can be used for diagnostic messages or as a root for temporary string names.

```
LPCTSTR AFXAPI AfxGetAppName( );
```

Return Value

A null-terminated string containing the application's name.

Example

```
// Print the application name to the debugger output window.
TRACE("Application name is %s\n", AfxGetAppName());
```

See Also

Concepts

MFC Macros and Globals

AfxGetDitheredBitmap

Copies a bitmap, replacing its background with a dithered (checker) pattern.

```
void AFXAPI AfxGetDitheredBitmap(
   const CBitmap &rSrc,
   CBitmap *pDest,
   COLORREF cr1,
   COLORREF cr2
);
```

Parameters

rSrc

The source bitmap.

pDest

The destination bitmap.

cr1

One of the two dither colors, typically white.

cr2

The other dither color, typically light gray (COLOR_MENU).

Remarks

The source bitmap is copied to the destination bitmap with a two-color (*cr1* and *cr2*) checkered pattern replacing the source bitmap's background. The background of the source bitmap is defined as its white pixels and all pixels matching the color of the pixel in the upper-left corner of the bitmap.

Example

```
CBitmap bm;
bm.LoadBitmap(IDB_BITMAP1);
CBitmap bmGray;
AfxGetDitheredBitmap(bm, &bmGray, RGB(255,255,255),
   GetSysColor(COLOR_MENU));
```

See Also

Reference

AfxDrawDitheredBitmap

AfxGetGrayBitmap

Concepts

MFC Macros and Globals

Other Resources

Gray and Dithered Bitmap Functions

AfxGetGrayBitmap

Copies a gray version of a bitmap.

```
void AFXAPI AfxGetGrayBitmap(
   const CBitmap &rSrc,
   CBitmap *pDest,
   COLORREF crBackground
);
```

Parameters

rSrc

The source bitmap.

pDest

The destination bitmap.

crBackground

The new background color (typically gray, such as COLOR_MENU).

Remarks

A bitmap copied with **AfxGetGrayBitmap** will have the appearance of a disabled control.

Example

```
CBitmap bm;
bm.LoadBitmap(IDB_BITMAP1);
CBitmap bmGray;
AfxGetGrayBitmap(bm, &bmGray, GetSysColor(COLOR_MENU));
```

See Also

Reference

AfxDrawGrayBitmap

AfxGetDitheredBitmap

Concepts

MFC Macros and Globals

Other Resources

Gray and Dithered Bitmap Functions

AfxGetHENV

You can use the returned handle in direct ODBC calls, but you must not close the handle or assume that the handle is still valid and available after any existing **CDatabase**- or **CRecordset**-derived objects have been destroyed.

```
HENV AFXAPI AfxGetHENV( );
```

Return Value

The handle to the ODBC environment currently in use by MFC. Can be **SQL_HENV_NULL** if there are no CDatabase objects and no CRecordset objects in use.

See Also

Concepts

MFC Macros and Globals

AfxGetInstanceHandle

This function allows you to retrieve the instance handle of the current application.

```
HINSTANCE AFXAPI AfxGetInstanceHandle( );
```

Return Value

An **HINSTANCE** to the current instance of the application. If called from within a DLL linked with the USRDLL version of MFC, an **HINSTANCE** to the DLL is returned.

Remarks

AfxGetInstanceHandle always returns the **HINSTANCE** of your executable file (.EXE) unless it is called from within a DLL linked with the USRDLL version of MFC. In this case, it returns an **HINSTANCE** to the DLL.

Example

```
// Print the application instance handle to the debugger output window.
TRACE("Application instance handle is 0x%0X\n", AfxGetInstanceHandle());
```

See Also

Reference

AfxGetResourceHandle
AfxSetResourceHandle

Concepts

MFC Macros and Globals

AfxGetInternetHandleType

Use this global function to determine the type of an Internet handle.

```
DWORD AFXAPI AfxGetInternetHandleType(
 HINTERNET hQuery
);
```

Parameters

hQuery

A handle to an Internet query.

Return Value

Any of the Internet service types defined by WININET.H. See the Remarks section for a list of these Internet services. If the handle is NULL or not recognized, the function returns AFX_INET_SERVICE_UNK.

Remarks

The following list includes possible Internet types returned by **AfxGetInternetHandleType**.

- INTERNET_HANDLE_TYPE_INTERNET
- INTERNET_HANDLE_TYPE_CONNECT_FTP
- INTERNET_HANDLE_TYPE_CONNECT_GOPHER
- INTERNET_HANDLE_TYPE_CONNECT_HTTP
- INTERNET_HANDLE_TYPE_FTP_FIND
- INTERNET_HANDLE_TYPE_FTP_FIND_HTML
- INTERNET_HANDLE_TYPE_FTP_FILE
- INTERNET_HANDLE_TYPE_FTP_FILE_HTML
- INTERNET_HANDLE_TYPE_GOPHER_FIND
- INTERNET_HANDLE_TYPE_GOPHER_FIND_HTML
- INTERNET_HANDLE_TYPE_GOPHER_FILE
- INTERNET_HANDLE_TYPE_GOPHER_FILE_HTML
- INTERNET_HANDLE_TYPE_HTTP_REQUEST

Note

In order to call this function, your project must include AFXINET.H.

See Also

Reference

AfxParseURL

Concepts

MFC Macros and Globals

AfxGetMainWnd

If your application is an OLE server, call this function to retrieve a pointer to the active main window of the application instead of directly referring to the m_pMainWnd member of the application object.

```
CWnd* AFXAPI AfxGetMainWnd( );
```

Return Value

If the server has an object that is in-place active inside a container, and this container is active, this function returns a pointer to the frame window object that contains the in-place active document.

If there is no object that is in-place active within a container, or your application is not an OLE server, this function simply returns the **m_pMainWnd** of your application object.

If **AfxGetMainWnd** is called from the application's primary thread, it returns the application's main window according to the above rules. If the function is called from a secondary thread in the application, the function returns the main window associated with the thread that made the call.

Remarks

If your application is not an OLE server, then calling this function is equivalent to directly referring to the **m_pMainWnd** member of your application object.

Example

```
//The following line send a WM_CLOSE message
//   to the Application's main window. This will cause the
//   Application to exit.
AfxGetMainWnd()->PostMessage(WM_CLOSE, 0, 0);
```

See Also

Reference

CWinThread::m_pMainWnd

Concepts

MFC Macros and Globals

AfxGetResourceHandle

Use the **HINSTANCE** handle returned by this function to access the application's resources directly, for example, in calls to the Windows function **FindResource**.

```
extern HINSTANCE AfxGetResourceHandle( );
```

Return Value

An **HINSTANCE** handle where the default resources of the application are loaded.

Example

```
//Load the menu specifying the module handle where resource is to be
//found & resource ID
HMENU hMenu = ::LoadMenu(AfxGetResourceHandle(), MAKEINTRESOURCE(IDR_PANEL));
```

See Also

Reference

AfxGetInstanceHandle
AfxSetResourceHandle
AfxFindResourceHandle

Concepts

MFC Macros and Globals

AfxGetStaticModuleState

Call this function to set the module state before initialization and/or to restore the previous module state after cleanup.

```
AFX_MODULE_STATE* AFXAPI AfxGetStaticModuleState( );
```

Return Value

A pointer to an **AFX_MODULE_STATE** structure.

Remarks

The **AFX_MODULE_STATE** structure contains global data for the module, that is, the portion of the module state that is pushed or popped.

By default, MFC uses the resource handle of the main application to load the resource template. If you have an exported function in a DLL, such as one that launches a dialog box in the DLL, this template is actually stored in the DLL module. You need to switch the module state for the correct handle to be used. You can do this by adding the following code to the beginning of the function:

```
AFX_MANAGE_STATE(AfxGetStaticModuleState( ));
```

This swaps the current module state with the state returned from **AfxGetStaticModuleState** until the end of the current scope.

For more information on module states and MFC, see "Managing the State Data of MFC Modules" in Creating New Documents, Windows, and Views and Technical Note 58.

See Also

Reference

AFX_MANAGE_STATE

Concepts

MFC Macros and Globals

AfxGetThread

Call this function to get a pointer to the CWinThread object representing the currently executing thread.

```
CWinThread* AfxGetThread( );
```

Return Value

Pointer to the currently executing thread; otherwise **NULL**.

Remarks

Must be called from within the desired thread.

Note

If you are porting an MFC project calling **AfxGetThread** from Visual C++ versions 4.2, 5.0, or 6.0, **AfxGetThread** calls AfxGetApp if no thread is found. In Visual C+ .NET and later, **AfxGetThread** returns **NULL** if no thread was found. If you wan t the application thread, you must call **AfxGetApp**.

Example

```
//Print the current thread ID in the Debug Window
TRACE("Current Thread ID = 0x%X\n", AfxGetThread()->m_nThreadID);
```

See Also

Reference

AfxBeginThread

Concepts

MFC Macros and Globals

AfxInitExtensionModule

Call this function in an extension DLL's **DllMain** to initialize the DLL.

```
BOOL AFXAPI AfxInitExtensionModule(
   AFX_EXTENSION_MODULE& state,
   HMODULE hModule
);
```

Parameters

state

A reference to the AFX_EXTENSION_MODULE structure that will contain the state of the extension DLL module after the initialization. The state includes a copy of the runtime class objects that have been initialized by the extension DLL as part of normal static object construction executed before **DllMain** is entered.

hModule

A handle of the extension DLL module.

Return Value

TRUE if the extension DLL is successfully initialized; otherwise, **FALSE**.

Remarks

For example:

```
static AFX_EXTENSION_MODULE extensionDLL;
extern "C" int APIENTRY
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID)
{
   if (dwReason == DLL_PROCESS_ATTACH)
   {
      // Extension DLL one-time initialization
      if (!AfxInitExtensionModule(extensionDLL, hInstance))
         return 0;
...
```

AfxInitExtensionModule makes a copy of the DLL's **HMODULE** and captures the DLL's runtime-classes (**CRuntimeClass** structures) as well as its object factories (**COleObjectFactory** objects) for use later when the **CDynLinkLibrary** object is created.

MFC extension DLLs need to do two things in their **DllMain** function:

- Call AfxInitExtensionModule and check the return value.
- Create a **CDynLinkLibrary** object if the DLL will be exporting CRuntimeClass objects or has its own custom resources.

You can call **AfxTermExtensionModule** to clean up the extension DLL when each process detaches from the extension DLL (which happens when the process exits, or when the DLL is unloaded as a result of an **AfxFreeLibrary** call).

See Also

Reference

AfxTermExtensionModule

Concepts

MFC Macros and Globals

AfxInitRichEdit

Call this function to initialize the rich edit control (version 1.0) for the application.

```
BOOL AFXAPI AfxInitRichEdit( );
```

Remarks

This function is provided for backward compatibility. Applications created with Visual C++ .NET and later use AfxInitRichEdit2.

AfxInitRichEdit loads RICHED32.DLL to initialize version 1.0 of the rich edit control. To use version 2.0 and 3.0 of the rich edit control, RICHED20.DLL needs to be loaded. This is accomplished with a call to AfxInitRichEdit2. If you have dialog resources with the rich edit control created prior to Visual C++ .NET, the rich edit controls are automatically version 1.0. Rich edit controls inserted using the Visual C++ .NET Resource Editor are version 2.0.

To update rich edit controls in existing Visual C++ applications to version 2.0, open the .RC file as text, change the class name of each rich edit control from "RICHEDIT" to "RichEdit20a". Then replace the call to **AfxInitRichEdit** with **AfxInitRichEdit2**.

This function also initializes the common controls library, if the library hasn't already been initialized for the process. If you use the rich edit control directly from your MFC application, you should call this function to assure that MFC has properly initialized the rich edit control runtime. If you call the Create method of CRichEditCtrl, CRichEditView, or CRichEditDoc, you typically don't need to call this function, but in some cases it might be necessary.

See Also

Reference

AfxInitRichEdit2

Concepts

MFC Macros and Globals

AfxInitRichEdit2

Call this function to initialize the rich edit control (version 2.0 and later) for the application.

```
BOOL AFXAPI AfxInitRichEdit2( );
```

Remarks

Call this function to load the RICHED20.DLL and initialize version 2.0 of the rich edit control. If you call the Create method of CRichEditCtrl, CRichEditView, or CRichEditDoc, you typically don't need to call this function, but in some cases it might be necessary.

See Also

Reference

AfxInitRichEdit

Concepts

MFC Macros and Globals

AfxIsMemoryBlock

Tests a memory address to make sure it represents a currently active memory block that was allocated by the diagnostic version of **new**.

```
BOOL AfxIsMemoryBlock(
   const void* p,
   UINT nBytes,
   LONG* plRequestNumber = NULL
);
```

Parameters

p

Points to the block of memory to be tested.

nBytes

Contains the length of the memory block in bytes.

plRequestNumber

Points to a **long** integer that will be filled in with the memory block's allocation sequence number, or zero if it does not represent a currently active memory block.

Return Value

Nonzero if the memory block is currently allocated and the length is correct; otherwise 0.

Remarks

It also checks the specified size against the original allocated size. If the function returns nonzero, the allocation sequence number is returned in *plRequestNumber*. This number represents the order in which the block was allocated relative to all other **new** allocations.

Example

```
// example for AfxIsMemoryBlock
CAge* pcage = new CAge( 21 ); // CAge is derived from CObject.
ASSERT( AfxIsMemoryBlock( pcage, sizeof( CAge ) ) )
```

See Also

Reference

AfxIsValidAddress

Concepts

MFC Macros and Globals

AfxIsValidAddress

Tests any memory address to ensure that it is contained entirely within the program's memory space.

```
BOOL AfxIsValidAddress(
   const void* lp,
   UINT nBytes,
   BOOL bReadWrite = TRUE
);
```

Parameters

lp

Points to the memory address to be tested.

nBytes

Contains the number of bytes of memory to be tested.

bReadWrite

Specifies whether the memory is both for reading and writing (**TRUE**) or just reading (**FALSE**).

Return Value

In debug builds, nonzero if the specified memory block is contained entirely within the program's memory space; otherwise 0.

In non-debug builds, nonzero if *lp* is not NULL; otherwise 0.

Remarks

The address is not restricted to blocks allocated by **new**.

Example

```
// Allocate a 5 character array, which should have a valid memory address.
char *array = new char[5];

// Create a null pointer, which should be an invalid memory address.
char *null = (char *)0x0;

ASSERT(AfxIsValidAddress(array, 5));
ASSERT(!AfxIsValidAddress(null, 5));
```

See Also

Reference

AfxIsMemoryBlock

AfxIsValidString

Concepts

MFC Macros and Globals

AfxIsValidString

Use this function to determine whether a pointer to a string is valid.

```
BOOL AfxIsValidString(
   LPCSTR lpsz,
   int nLength = -1
);
```

Parameters

lpsz

The pointer to test.

nLength

Specifies the length of the string to be tested, in bytes. A value of –1 indicates that the string will be null-terminated.

Return Value

In debug builds, nonzero if the specified pointer points to a string of the specified size; otherwise 0.

In non-debug builds, nonzero if *lpsz* is not NULL; otherwise 0.

Example

```
// Create a character string which should be valid.
char str[12] = "hello world";

// Create a null pointer, which should be an invalid string.
char *null = (char *)0x0;

ASSERT(AfxIsValidString(str, 12));
ASSERT(!AfxIsValidString(null, 5));
```

See Also

Reference

AfxIsMemoryBlock

AfxIsValidAddress

Concepts

MFC Macros and Globals

AfxLoadLibrary

Use **AfxLoadLibrary** to map a DLL module.

```
HINSTANCE AFXAPI AfxLoadLibrary(
   LPCTSTR lpszModuleName
);
```

Parameters

lpszModuleName

Points to a null-terminated string that contains the name of the module (either a .DLL or .EXE file). The name specified is the filename of the module.

If the string specifies a path but the file does not exist in the specified directory, the function fails.

If a path is not specified and the filename extension is omitted, the default extension .DLL is appended. However, the filename string can include a trailing point character (.) to indicate that the module name has no extension. When no path is specified, the function searches for the file in the following sequence:

- The directory from which the application loaded.
- The current directory.
- **Windows 95/98:** The Windows system directory. **Windows NT:** The 32-bit Windows system directory. The name of this directory is SYSTEM32.
- **Windows NT only:** The 16-bit Windows system directory. There is no Win32 function that obtains the path of this directory, but it is searched. The name of this directory is SYSTEM.
- The Windows directory.
- The directories that are listed in the PATH environment variable.

Return Value

If the function succeeds, the return value is a handle to the module. If the function fails, the return value is NULL.

Remarks

It returns a handle that can be used in GetProcAddress to get the address of a DLL function. **AfxLoadLibrary** can also be used to map other executable modules.

Each process maintains a reference count for each loaded library module. This reference count is incremented each time **AfxLoadLibrary** is called and is decremented each time **AfxFreeLibrary** is called. When the reference count reaches zero, the module is unmapped from the address space of the calling process and the handle is no longer valid.

Be sure to use **AfxLoadLibrary** and **AfxFreeLibrary** (instead of the Win32 functions **LoadLibrary** and **FreeLibrary**) if your application uses multiple threads and if it dynamically loads an extension DLL. Using **AfxLoadLibrary** and **AfxFreeLibrary** insures that the startup and shutdown code that executes when the extension DLL is loaded and unloaded does not corrupt the global MFC state.

Using **AfxLoadLibrary** in an application requires you to dynamically link to the DLL version of MFC; the header file for **AfxLoadLibrary**, Afxdll_.h, is only included if MFC is linked to the application as a DLL. This is by design because you have to link to the DLL version of MFC to use or create extension DLLs.

Example

```
// The following shows how to create a MDI based application
// using a generic CView derived class that is implemented in
// a dynamically loaded MFC Extension DLL.

typedef CRuntimeClass * (*GETDLLVIEW)();

BOOL CMyApp::InitInstance()
{
```

```
    // ....
    // Standard Application Wizard generated initalization excluded.
    // ...

    // Register the application's document templates.  Document
    // templates serve as the connection between documents,
    // frame windows, and views.


    //Load MFC Extension DLL based view class.
    m_hViewDll = AfxLoadLibrary("myview.dll");
    if (!m_hViewDll)
    {
       AfxMessageBox("Error: Cannot find component \"MyView.Dll\"");
       return FALSE;
    }

    GETDLLVIEW GetView = (GETDLLVIEW) GetProcAddress(m_hViewDll,
        "GetView");
    ASSERT (GetView != NULL);

    CMultiDocTemplate* pDocTemplate;
    pDocTemplate = new CMultiDocTemplate(
       IDR_MYTYPE,
       RUNTIME_CLASS(CMyDoc),
       // Custom MDI child frame.
       RUNTIME_CLASS(CChildFrame),
       //Call our DLL function to return view class.
       GetView());
    AddDocTemplate(pDocTemplate);

    // ....
    // Standard Application Wizard generated initalization excluded.
    // ...

    return TRUE;
}


int CMyApp::ExitInstance()
{

    if (m_hViewDll)
    {
       AfxFreeLibrary(m_hViewDll);
       m_hViewDll = NULL;
    }

    return CWinApp::ExitInstance();
}
```

See Also

Reference

AfxFreeLibrary

Concepts

MFC Macros and Globals

AFX_MANAGE_STATE

Call this macro to protect an exported function in a DLL.

```
AFX_MANAGE_STATE(AFX_MODULE_STATE* pModuleState )
```

Parameters

pModuleState

A pointer to an **AFX_MODULE_STATE** structure.

Remarks

When this macro is invoked, *pModuleState* is the effective module state for the remainder of the immediate containing scope. Upon leaving the scope, the previous effective module state will be automatically restored.

The **AFX_MODULE_STATE** structure contains global data for the module, that is, the portion of the module state that is pushed or popped.

By default, MFC uses the resource handle of the main application to load the resource template. If you have an exported function in a DLL, such as one that launches a dialog box in the DLL, this template is actually stored in the DLL module. You need to switch the module state for the correct handle to be used. You can do this by adding the following code to the beginning of the function:

```
AFX_MANAGE_STATE(AfxGetStaticModuleState( ));
```

This swaps the current module state with the state returned from AfxGetStaticModuleState until the end of the current scope.

For more information on module states and MFC, see "Managing the State Data of MFC Modules" in Creating New Documents, Windows, and Views and Technical Note 58.

Note

When MFC creates an activation context for an assembly, it uses AfxWinInit to create the context and **AFX_MANAGE_STATE** to activate and deactivate it. Note also that as of Visual C++ 2005, the behavior of **AFX_MANAGE_STATE** has changed in that it is enabled for static MFC libraries, as well as MFC DLLs, in order to allow MFC code to execute in the proper activation context selected by the User DLL. For more information, see Support for Activation Contexts in the MFC Module State.

See Also

Reference

AfxGetStaticModuleState

Concepts

MFC Macros and Globals

afxMemDF

This variable is accessible from a debugger or your program and allows you to tune allocation diagnostics.

```
int afxMemDF;
```

Remarks

afxMemDF can have the following values as specified by the enumeration **afxMemDF**:

- **allocMemDF** Turns on debugging allocator (default setting in Debug library).
- **delayFreeMemDF** Delays freeing memory. While your program frees a memory block, the allocator does not return that memory to the underlying operating system. This will place maximum memory stress on your program.
- **checkAlwaysMemDF** Calls **AfxCheckMemory** every time memory is allocated or freed. This will significantly slow memory allocations and deallocations.

Example

```
// example for afxMemDF
afxMemDF = allocMemDF | checkAlwaysMemDF;
```

See Also

Concepts

MFC Macros and Globals

AfxMessageBox

Displays a message box on the screen.

```
int AfxMessageBox(
   LPCTSTR lpszText,
   UINT nType = MB_OK,
   UINT nIDHelp = 0
);
int AFXAPI AfxMessageBox(
   UINT nIDPrompt,
   UINT nType = MB_OK,
   UINT nIDHelp = (UINT
) -1
);
```

Parameters

lpszText

Points to a **CString** object or null-terminated string containing the message to be displayed in the message box.

nType

The style of the message box. Apply any of the message-box styles to the box.

nIDHelp

The Help context ID for the message; 0 indicates the application's default Help context will be used.

nIDPrompt

A unique ID used to reference a string in the string table.

Return Value

Zero if there is not enough memory to display the message box; otherwise, one of the following values is returned:

- **IDABORT** The Abort button was selected.
- **IDCANCEL** The Cancel button was selected.
- **IDIGNORE** The Ignore button was selected.
- **IDNO** The No button was selected.
- **IDOK** The OK button was selected.
- **IDRETRY** The Retry button was selected.
- **IDYES** The Yes button was selected.

If a message box has a Cancel button, the **IDCANCEL** value will be returned if either the ESC key is pressed or the Cancel button is selected. If the message box has no Cancel button, pressing the ESC key has no effect.

The functions AfxFormatString1 and AfxFormatString2 can be useful in formatting text that appears in a message box.

Remarks

The first form of this overloaded function displays a text string pointed to by *lpszText* in the message box and uses *nIDHelp* to describe a Help context. The Help context is used to jump to an associated Help topic when the user presses the Help key (typically F1).

The second form of the function uses the string resource with the ID *nIDPrompt* to display a message in the message box. The associated Help page is found through the value of *nIDHelp*. If the default value of *nIDHelp* is used (– 1), the string resource ID, *nIDPrompt*, is used for the Help context. For more information about defining Help contexts, see the article Help Topics and Technical Note 28.

Example

```
// A simple message box, with only the OK button.
AfxMessageBox("Simple message box.");

// A message box that uses a string from a string table
// with yes and no buttons and the stop icon.
// NOTE: nStringID is an integer that contains a valid id of
// a string in the current resource.
AfxMessageBox(nStringID, MB_YESNO|MB_ICONSTOP);
```

See Also

Reference

CWnd::MessageBox

Concepts

MFC Macros and Globals

AfxNetInitModule

For MFC Sockets support from a regular DLL that is dynamically linked to MFC, add a call to this function in your regular DLL's **CWinApp::InitInstance** function to initialize the MFC Sockets DLL.

```
void AFXAPI AfxNetInitModule( );
```

Remarks

The MFC Sockets DLL is an extension DLL; in order for an extension DLL to get wired into a **CDynLinkLibrary** chain, it must create a **CDynLinkLibrary** object in the context of every module that will be using it. **AfxNetInitModule** creates the **CDynLinkLibrary** object in your regular DLL's context so that it gets wired into the **CDynLinkLibrary** object chain of the regular DLL.

Requirements

Header: <afxdll_.h>

See Also

Reference

AfxMessageBox

Concepts

MFC Macros and Globals

AFX_ODBC_CALL

Use this macro to call any ODBC API function that may return **SQL_STILL_EXECUTING**.

```
AFX_ODBC_CALL(SQLFunc )
```

Parameters

SQLFunc

An ODBC API function. For more information about ODBC API functions, see the *Platform SDK*.

Remarks

AFX_ODBC_CALL repeatedly calls the function until it no longer returns **SQL_STILL_EXECUTING**.

Before invoking **AFX_ODBC_CALL**, you must declare a variable, `nRetCode`, of type **RETCODE**.

Note that the MFC ODBC classes now use only synchronous processing. In order to perform an asynchronous operation, you must call the ODBC API function **SQLSetConnectOption**. For more information, see the topic "Executing Functions Asynchronously" in the *Platform SDK*.

Example

This example uses **AFX_ODBC_CALL** to call the **SQLColumns** ODBC API function, which returns a list of the columns in the table named by `strTableName`. Note the declaration of `nRetCode` and the use of recordset data members to pass parameters to the function. The example also illustrates checking the results of the call with **Check**, a member function of class **CRecordset**. The variable `prs` is a pointer to a **CRecordset** object, declared elsewhere.

```
// example for AFX_ODBC_CALL

RETCODE nRetCode;

AFX_ODBC_CALL( ::SQLColumns( prs->m_hstmt,
         (UCHAR *)NULL, SQL_NTS, (UCHAR *)NULL,
         SQL_NTS, (UCHAR *)(constchar*)strTableName,
         SQL_NTS, (UCHAR *)NULL, SQL_NTS ) );

if ( !prs->Check( nRetCode ) )
{
   AfxThrowDBException( nRetCode, prs->m_pdb,
                        prs->m_hstmt );
   TRACE( "SQLColumns failed\n" );
}
```

See Also

Reference

AFX_SQL_ASYNC

AFX_SQL_SYNC

Concepts

MFC Macros and Globals

AfxOleCanExitApp

Indicates whether the application can terminate.

```
BOOL AFXAPI AfxOleCanExitApp( );
```

Return Value

Nonzero if the application can exit; otherwise 0.

Remarks

An application should not terminate if there are outstanding references to its objects. The global functions **AfxOleLockApp** and **AfxOleUnlockApp** increment and decrement, respectively, a counter of references to the application's objects. The application should not terminate when this counter is nonzero. If the counter is nonzero, the application's main window is hidden (not destroyed) when the user chooses Close from the system menu or Exit from the File menu. The framework calls this function in **CFrameWnd::OnClose**.

Example

```
// Helper exit function for automation server
BOOL CDlgautoDlg::CanExit()
{
   if (AfxOleCanExitApp())
   {
      // No outstanding object counts - go ahead and exit
      return TRUE;
   } else
   {
      // There are outstanding OLE object counts...
      // hide app to give user impression that application has exited.
      ShowWindow(SW_HIDE);
      // take user out of control of the app
      AfxOleSetUserCtrl(FALSE);
      return FALSE;
   }
}
```

Requirements

Header: <afxdisp.h>

See Also

Reference

AfxOleLockApp

AfxOleUnlockApp

Concepts

MFC Macros and Globals

AfxOleGetMessageFilter

Retrieves the application's current message filter.

```
COleMessageFilter* AFXAPI AfxOleGetMessageFilter( );
```

Return Value

A pointer to the current message filter.

Remarks

Call this function to access the current **COleMessageFilter**-derived object, just as you would call **AfxGetApp** to access the current application object.

Example

```
// example of AfxOleGetMessageFilter
COleMessageFilter* pFilter = AfxOleGetMessageFilter();
ASSERT_VALID(pFilter);
pFilter->BeginBusyState();
// do things requiring a busy state
pFilter->EndBusyState();

// another example
//CWinApp-derived class
BOOL CSAProjApp::InitInstance()
{
   // Initialize OLE libraries and registers default message filter.
   if (!AfxOleInit())
   {
      AfxMessageBox("Ole initialization failed\n");
      return FALSE;
   }

   CWinThread* pThread = AfxGetThread();
   if (pThread != NULL)
   {
      // Destroy message filter, thereby unregistering it.
      delete pThread->m_pMessageFilter;
      pThread->m_pMessageFilter = NULL;

      // Create the new message filter object.
      //CMyMessageFilter is derived from COleMessageFilter
      pThread->m_pMessageFilter = new CMyMessageFilter;
      ASSERT(AfxOleGetMessageFilter() != NULL);

      // Register the new message filter object.
      AfxOleGetMessageFilter()->Register();
   }
   //...
   //...
   //...
}
```

Requirements

Header: <afxwin.h>

See Also

Reference

COleMessageFilter Class

AfxGetApp

Concepts

AfxOleGetUserCtrl

Retrieves the current user-control flag.

```
BOOL AFXAPI AfxOleGetUserCtrl( );
```

Return Value

Nonzero if the user is in control of the application; otherwise 0.

Remarks

The user is in control of the application when the user has explicitly opened or created a new document. The user is also in control if the application was not launched by the OLE system DLLs — in other words, if the user launched the application with the system shell.

Requirements

Header: <afxdisp.h>

See Also

Reference

AfxOleSetUserCtrl

Concepts

MFC Macros and Globals

AfxOleInit

Initializes OLE support for the application.

```
BOOL AFXAPI AfxOleInit( );
```

Return Value

Nonzero if successful; 0 if initialization fails, possibly because incorrect versions of the OLE system DLLs are installed.

Remarks

Call this function to initialize the OLE support for an MFC application. When this function is called, the following actions occur:

- Initializes the COM library on the current apartment of the calling application. For more information, see OleInitialize.
- Creates a message filter object, implementing the IMessageFilter interface. This message filter can be accessed with a call to AfxOleGetMessageFilter.

Note

If **AfxOleInit** is called from an MFC DLL, the call will fail. The failure occurs because the function assumes that, if it is called from a DLL, the OLE system was previously initialized by the calling application.

Note

MFC applications must be initialized as single threaded apartment (STA). If you call CoInitializeEx in your **InitInstance** override, specify **COINIT_APARTMENTTHREADED** (rather than **COINIT_MULTITHREADED**). For more information, see PRB: MFC Application Stops Responding When You Initialize the Application as a Multithreaded Apartment (828643) at http://support.microsoft.com/default.aspx?scid=kb;en-us;828643.

See Also

Reference

AfxMessageBox

Concepts

MFC Macros and Globals

AfxOleInitModule

For OLE support from a regular DLL that is dynamically linked to MFC, call this function in your regular DLL's **CWinApp::InitInstance** function to initialize the MFC OLE DLL.

```
void AFXAPI AfxOleInitModule( );
```

Remarks

The MFC OLE DLL is an extension DLL; in order for an extension DLL to get wired into a **CDynLinkLibrary** chain, it must create a **CDynLinkLibrary** object in the context of every module that will be using it. **AfxOleInitModule** creates the **CDynLinkLibrary** object in your regular DLL's context so that it gets wired into the **CDynLinkLibrary** object chain of the regular DLL.

If you are building an OLE control and are using **COleControlModule**, you should not call **AfxOleInitModule** because the **InitInstance** member function for **COleControlModule** calls **AfxOleInitModule**.

Requirements

Header: <afxdll_.h>

See Also
Reference
AfxMessageBox
Concepts
MFC Macros and Globals

AfxOleLockApp

Increments the framework's global count of the number of active objects in the application.

```
void AFXAPI AfxOleLockApp( );
```

Remarks

The framework keeps a count of the number of objects active in an application. The **AfxOleLockApp** and **AfxOleUnlockApp** functions, respectively, increment and decrement this count.

When the user attempts to close an application that has active objects — an application for which the count of active objects is nonzero — the framework hides the application from the user's view instead of completely shutting it down. The **AfxOleCanExitApp** function indicates whether the application can terminate.

Call **AfxOleLockApp** from any object that exposes OLE interfaces, if it would be undesirable for that object to be destroyed while still being used by a client application. Also call **AfxOleUnlockApp** in the destructor of any object that calls **AfxOleLockApp** in the constructor. By default, **COleDocument** (and derived classes) automatically lock and unlock the application.

Example

```
// Below is a code sample from an  Application Wizard-generated SDI
// Application with Automation support. The Application Wizard adds a
// dispatch interface to the document class. AfxOleLockApp() and
// AfxOleUnlockApp() respectively increment and decrement the
// application's object count. When the object count is equal to
// zero and if the user has not taken control of the application,
// the server is terminated.

CTestDoc::CTestDoc()
{
   EnableAutomation();
   AfxOleLockApp();
}

CTestDoc::~CTestDoc()
{
   AfxOleUnlockApp();
}
```

Requirements

Header: <afxdisp.h>

See Also

Reference

AfxOleUnlockApp
AfxOleCanExitApp
COleDocument Class

Concepts

MFC Macros and Globals

AfxOleLockControl

Locks the class factory of the specified control so that dynamically created data associated with the control remains in memory.

```
BOOL AFXAPI AfxOleLockControl(
   REFCLSID clsid
);
BOOL AFXAPI AfxOleLockControl(
   LPCTSTR lpszProgID
);
```

Parameters

clsid

The unique class ID of the control.

lpszProgID

The unique program ID of the control.

Return Value

Nonzero if the class factory of the control was successfully locked; otherwise 0.

Remarks

This can significantly speed up display of the controls. For example, once you create a control in a dialog box and lock the control with **AfxOleLockControl**, you do not need to create and kill it again every time the dialog is shown or destroyed. If the user opens and closes a dialog box repeatedly, locking your controls can significantly enhance performance. When you are ready to destroy the control, call **AfxOleUnlockControl**.

Example

```
BOOL CDlgApp::InitInstance()
{
...
   // Starts and locks controls (Microsoft FlexGrid Control) class factory.
   // Control will remain in memory for lifetime of
   // application or until AfxOleUnlockControl() is called.

   AfxOleLockControl("MSFlexGridLib.MSFlexGrid.1");
...
   return TRUE;
}
```

Requirements

Header: <afxwin.h>

See Also

Reference

AfxOleUnlockControl

Concepts

MFC Macros and Globals

AfxOleRegisterControlClass

Registers the control class with the Windows registration database.

```
BOOL AFXAPI AfxOleRegisterControlClass(
   HINSTANCE hInstance,
   REFCLSID clsid,
   LPCTSTR pszProgID,
   UINT idTypeName,
   UINT idBitmap,
   int nRegFlags,
   DWORD dwMiscStatus,
   REFGUID tlid,
   WORD wVerMajor,
   WORD wVerMinor
);
```

Parameters

hInstance

The instance handle of the module associated with the control class.

clsid

The unique class ID of the control.

pszProgID

The unique program ID of the control.

idTypeName

The resource ID of the string that contains a user-readable type name for the control.

idBitmap

The resource ID of the bitmap used to represent the OLE control in a toolbar or palette.

nRegFlags

Contains one or more of the following flags:

- **afxRegInsertable** Allows the control to appear in the Insert Object dialog box for OLE objects.
- **afxRegApartmentThreading** Sets the threading model in the registry to ThreadingModel=Apartment.
- **afxRegFreeThreading** Sets the threading model in the registry to ThreadingModel=Free.

  You can combine the two flags **afxRegApartmentThreading** and **afxRegFreeThreading** to set ThreadingModel=Both. See InprocServer32 in the *Platform SDK* for more information on threading model registration.

Note
In MFC versions before MFC 4.2, the **int** *nRegFlags* parameter was a **BOOL** parameter, *bInsertable*, that allowed or disallowed the control to be inserted from the Insert Object dialog box.

dwMiscStatus

Contains one or more of the following status flags (for a description of the flags, see **OLEMISC** enumeration in the *Platform SDK*):

- OLEMISC_RECOMPOSEONRESIZE
- OLEMISC_ONLYICONIC
- OLEMISC_INSERTNOTREPLACE

- OLEMISC_STATIC
- OLEMISC_CANTLINKINSIDE
- OLEMISC_CANLINKBYOLE1
- OLEMISC_ISLINKOBJECT
- OLEMISC_INSIDEOUT
- OLEMISC_ACTIVATEWHENVISIBLE
- OLEMISC_RENDERINGISDEVICEINDEPENDENT
- OLEMISC_INVISIBLEATRUNTIME
- OLEMISC_ALWAYSRUN
- OLEMISC_ACTSLIKEBUTTON
- OLEMISC_ACTSLIKELABEL
- OLEMISC_NOUIACTIVATE
- OLEMISC_ALIGNABLE
- OLEMISC_IMEMODE
- OLEMISC_SIMPLEFRAME
- OLEMISC_SETCLIENTSITEFIRST

tlid

The unique ID of the control class.

wVerMajor

The major version number of the control class.

wVerMinor

The minor version number of the control class.

Return Value

Nonzero if the control class was registered; otherwise 0.

Remarks

This allows the control to be used by containers that are OLE-control aware. **AfxOleRegisterControlClass** updates the registry with the control's name and location on the system and also sets the threading model that the control supports in the registry. For more information, see Technical Note 64, "Apartment-Model Threading in OLE Controls," and About Processes and Threads in the *Platform SDK*.

Example

```
// Member function implementation of class COleObjectFactory::UpdateRegistry
//
BOOL CMyApartmentAwareCtrl::CApartmentCtrlFactory::UpdateRegistry(BOOL bRegister)
{
// TODO: Verify that your control follows
// apartment-model threading rules.
// Refer to MFC TechNote 64 for more information.
// If your control does not conform to the
// apartment-model rules, then you must modify the
// code below, changing the 6th parameter from
// afxRegInsertable | afxRegApartmentThreading to
// afxRegInsertable.

   if (bRegister)
      return AfxOleRegisterControlClass()
         AfxGetInstanceHandle(),
         m_clsid,
```

```
            m_lpszProgID,
            IDS_APARTMENT,
            IDB_APARTMENT,
            afxRegInsertable | afxRegApartmentThreading,
            _dwApartmentOleMisc,
            _tlid,
            _wVerMajor,
            _wVerMinor);
    else
       return AfxOleUnregisterClass(m_clsid, m_lpszProgID);
```

The above example demonstrates how **AfxOleRegisterControlClass** is called with the flag for insertable and the flag for apartment model ORed together to create the sixth parameter:

```
    afxRegInsertable | afxRegApartmentThreading,
```

The control will show up in the Insert Object dialog box for enabled containers, and it will be apartment model-aware. Apartment model-aware controls must ensure that static class data is protected by locks, so that while a control in one apartment is accessing the static data, it isn't disabled by the scheduler before it is finished, and another instance of the same class starts using the same static data. Any accesses to the static data will be surrounded by critical section code.

Requirements

Header: <afxctl.h>

See Also

Reference

AfxOleRegisterPropertyPageClass
AfxOleRegisterTypeLib
AfxOleUnregisterClass
AfxOleUnregisterTypeLib

Concepts

MFC Macros and Globals

AfxOleRegisterPropertyPageClass

Registers the property page class with the Windows registration database.

```
BOOL AFXAPI AfxOleRegisterPropertyPageClass(
   HINSTANCE hInstance,
      REFCLSID clsid,
   UINT idTypeName,
   int nRegFlags
);
```

Parameters

hInstance

The instance handle of the module associated with the property page class.

clsid

The unique class ID of the property page.

idTypeName

The resource ID of the string that contains a user-readable name for the property page.

nRegFlags

May contain the flag:

- **afxRegApartmentThreading** Sets the threading model in the registry to ThreadingModel = Apartment.

Note
In MFC versions prior to MFC 4.2, the **int** *nRegFlags* parameter was not available. Note also that the **afxRegInsertable** flag is not a valid option for property pages and will cause an ASSERT in MFC if it is set

Return Value

Nonzero if the control class was registered; otherwise 0.

Remarks

This allows the property page to be used by containers that are OLE-control aware. **AfxOleRegisterPropertyPageClass** updates the registry with the property page name and its location on the system and also sets the threading model that the control supports in the registry. For more information, see Technical Note 64, "Apartment-Model Threading in OLE Controls," and About Processes and Threads in the *Platform SDK*.

Requirements

Header: afxctl.h

See Also

Reference

AfxOleRegisterControlClass
AfxOleRegisterTypeLib

AfxOleRegisterServerClass

This function allows you to register your server in the OLE system registry.

```
BOOL AFXAPI AfxOleRegisterServerClass(
   REFCLSID clsid,
   LPCTSTR lpszClassName,
   LPCTSTR lpszShortTypeName,
   LPCTSTR lpszLongTypeName,
   OLE_APPTYPE nAppType = OAT_SERVER,
   LPCTSTR* rglpszRegister = NULL,
   LPCTSTR* rglpszOverwrite = NULL
);
```

Parameters

clsid

Reference to the server's OLE class ID.

lpszClassName

Pointer to a string containing the class name of the server's objects.

lpszShortTypeName

Pointer to a string containing the short name of the server's object type, such as "Chart."

lpszLongTypeName

Pointer to a string containing the long name of the server's object type, such as "Microsoft Excel 5.0 Chart."

nAppType

A value, taken from the **OLE_APPTYPE** enumeration, specifying the type of OLE application. Possible values are the following:

- **OAT_INPLACE_SERVER** Server has full server user-interface.
- **OAT_SERVER** Server supports only embedding.
- **OAT_CONTAINER** Container supports links to embeddings.
- **OAT_DISPATCH_OBJECT** **IDispatch**-capable object.

rglpszRegister

Array of pointers to strings representing the keys and values to be added to the OLE system registry if no existing values for the keys are found.

rglpszOverwrite

Array of pointers to strings representing the keys and values to be added to the OLE system registry if the registry contains existing values for the given keys.

Return Value

Nonzero if the server class is successfully registered; otherwise 0.

Remarks

Most applications can use **COleTemplateServer::Register** to register the application's document types. If your application's system-registry format does not fit the typical pattern, you can use **AfxOleRegisterServerClass** for more control.

The registry consists of a set of keys and values. The *rglpszRegister* and *rglpszOverwrite* arguments are arrays of pointers to strings, each consisting of a key and a value separated by a **NULL** character (`'\0'`). Each of these strings can have replaceable parameters whose places are marked by the character sequences `%1` through `%5`.

The symbols are filled in as follows:

Symbol	Value
%1	Class ID, formatted as a string
%2	Class name
%3	Path to executable file
%4	Short type name
%5	Long type name

Requirements

Header: <afxdisp.h>

See Also

Reference

COleTemplateServer::UpdateRegistry

Concepts

MFC Macros and Globals

AfxOleRegisterTypeLib

Registers the type library with the Windows registration database and allows the type library to be used by other containers that are OLE-control aware.

```
BOOL AfxOleRegisterTypeLib(
   HINSTANCE hInstance,
   REFGUID tlid,
   LPCTSTR pszFileName = NULL,
   LPCTSTR pszHelpDir = NULL
);
```

Parameters

hInstance

The instance handle of the application associated with the type library.

tlid

The unique ID of the type library.

pszFileName

Points to the optional filename of a localized type library (.TLB) file for the control.

pszHelpDir

The name of the directory where the help file for the type library can be found. If **NULL**, the help file is assumed to be in the same directory as the type library itself.

Return Value

Nonzero if the type library was registered; otherwise 0.

Remarks

This function updates the registry with the type library name and its location on the system.

Example

```
const GUID CDECL BASED_CODE _tlid = { 0x43bd9e45, 0x328f, 0x11d0,
      { 0xa6, 0xb9, 0x0, 0xaa, 0x0, 0xa7, 0xf, 0xc2 } };

// Registers type library and the interfaces
// in it, afxctl.h needs to be included
if (!AfxOleRegisterTypeLib(AfxGetInstanceHandle(), _tlid))
  return ResultFromScode(SELFREG_E_TYPELIB);


// proj.tlb should be in the same directory as exe module.
// last param can be null if help file associated w/ tlb is in same dir as .tlb
if (!AfxOleRegisterTypeLib(AfxGetInstanceHandle(), _tlid, _T("proj.tlb"), NULL))
   return ResultFromScode(SELFREG_E_TYPELIB);
```

See Also

Reference

AfxOleUnregisterTypeLib
AfxOleRegisterControlClass
AfxOleUnregisterClass

Concepts

MFC Macros and Globals

AfxOleSetEditMenu

Implements the user interface for the *typename* Object command.

```
void AFXAPI AfxOleSetEditMenu(
   COleClientItem* pClient,
   CMenu* pMenu,
   UINT iMenuItem,
   UINT nIDVerbMin,
   UINT nIDVerbMax = 0,
   UINT nIDConvert = 0
);
```

Parameters

pClient

A pointer to the client OLE item.

pMenu

A pointer to the menu object to be updated.

iMenuItem

The index of the menu item to be updated.

nIDVerbMin

The command ID that corresponds to the primary verb.

nIDVerbMax

The command ID that corresponds to the last verb.

nIDConvert

ID for the Convert menu item.

Remarks

If the server recognizes only a primary verb, the menu item becomes "verb *typename* Object" and the *nIDVerbMin* command is sent when the user chooses the command. If the server recognizes several verbs, then the menu item becomes "*typename* Object" and a submenu listing all the verbs appears when the user chooses the command. When the user chooses a verb from the submenu, *nIDVerbMin* is sent if the first verb is chosen, *nIDVerbMin* + 1 is sent if the second verb is chosen, and so forth. The default **COleDocument** implementation automatically handles this feature.

You must have the following statement in your client's application resource script (.RC) file:

#include <afxolecl.rc>

Requirements

Header: afxole.h

See Also

Reference

COleDocument Class

Concepts

MFC Macros and Globals

AfxOleSetUserCtrl

Sets or clears the user-control flag, which is explained in the reference for **AfxOleGetUserCtrl**.

```
void AFXAPI AfxOleSetUserCtrl(
   BOOL bUserCtrl
);
```

Parameters

bUserCtrl

Specifies whether the user-control flag is to be set or cleared.

Remarks

The framework calls this function when the user creates or loads a document, but not when a document is loaded or created through an indirect action such as loading an embedded object from a container application.

Call this function if other actions in your application should put the user in control of the application.

Requirements

Header: <afxdisp.h>

See Also

Reference

AfxOleGetUserCtrl

Concepts

MFC Macros and Globals

AfxOleUnlockApp

Decrements the framework's count of active objects in the application.

```
void AFXAPI AfxOleUnlockApp( );
```

Remarks

See **AfxOleLockApp** for further information.

When the number of active objects reaches zero, **AfxOleOnReleaseAllObjects** is called.

Example

See the example for AfxOleLockApp.

Requirements

Header: <afxdisp.h>

See Also

Reference

AfxOleLockApp

CCmdTarget::OnFinalRelease

Concepts

MFC Macros and Globals

AfxOleUnlockControl

Unlocks the class factory of the specified control.

```
BOOL AFXAPI AfxOleUnlockControl(
   REFCLSID clsid
);
BOOL AFXAPI AfxOleUnlockControl(
   LPCTSTR lpszProgID
);
```

Parameters

clsid

The unique class ID of the control.

lpszProgID

The unique program ID of the control.

Return Value

Nonzero if the class factory of the control was successfully unlocked; otherwise 0.

Remarks

A control is locked with **AfxOleLockControl**, so that dynamically created data associated with the control remains in memory. This can significantly speed up display of the control because the control need not be created and destroyed every time it is displayed. When you are ready to destroy the control, call **AfxOleUnlockControl**.

Example

```
// Unlock controls (Microsoft FlexGrid Control) class factory.
// Structure MSFlexGridLib::MSFlexGrid below obtained from #import of
// msflxgrd.ocx.

AfxOleUnlockControl(__uuidof(MSFlexGridLib::MSFlexGrid));
```

Requirements

Header: <afxwin.h>

See Also

Reference

AfxOleLockControl

Concepts

MFC Macros and Globals

AfxOleUnregisterClass

Removes the control or property page class entry from the Windows registration database.

```
BOOL AFXAPI AfxOleUnregisterClass(
   REFCLSID clsID,
   LPCSTR pszProgID
);
```

Parameters

clsID

The unique class ID of the control or property page.

pszProgID

The unique program ID of the control or property page.

Return Value

Nonzero if the control or property page class was successfully unregistered; otherwise 0.

See Also

Reference

AfxOleRegisterPropertyPageClass
AfxOleRegisterControlClass
AfxOleRegisterTypeLib

Concepts

MFC Macros and Globals

AfxOleUnregisterTypeLib

Call this function to remove the type library entry from the Windows registration database.

```
BOOL AFXAPI AfxOleUnregisterTypeLib(
   REFGUID tlID
);
```

Parameters

tlID

The unique ID of the type library.

Return Value

Nonzero if the type library was successfully unregistered; otherwise 0.

Example

```
// Type library GUID, corresponds to the uuid attribute of the library
// section in the .odl file.
const GUID CDECL BASED_CODE _tlid =   { 0x9dbafcd2, 0x592f, 0x101b,
 { 0x85, 0xce, 0x0, 0x60, 0x8c, 0xec, 0x29, 0x7b } };

// Type library major version number, number on the left of decimal
// point, in version attribute of the library section in .odl file.
const WORD _wVerMajor = 1;

// Type library minor version number, number on the right of decimal
// point, in version attribute of the library section in .odl file.
const WORD _wVerMinor = 0;

STDAPI DllUnregisterServer(void)
{
   AFX_MANAGE_STATE(AfxGetStaticModuleState());

   // register typelib with locale neutral
   if (!AfxOleUnregisterTypeLib(_tlid, _wVerMajor, _wVerMinor, LOCALE_NEUTRAL))
      return ResultFromScode(SELFREG_E_TYPELIB);

   // register classes
   if (!COleObjectFactoryEx::UpdateRegistryAll(FALSE))
      return ResultFromScode(SELFREG_E_CLASS);

   return S_OK;
}
```

See Also

Reference

AfxOleUnregisterClass

AfxOleRegisterTypeLib

Concepts

MFC Macros and Globals

AfxParseURL

This global is used in CInternetSession::OpenURL.

```
BOOL AFXAPI AfxParseURL(
   LPCTSTR pstrURL,
   DWORD& dwServiceType,
   CString& strServer,
   CString& strObject,
   INTERNET_PORT& nPort
);
```

Parameters

pstrURL

A pointer to a string containing the URL to be parsed.

dwServiceType

Indicates the type of Internet service. Possible values are as follows:

- AFX_INET_SERVICE_FTP
- AFX_INET_SERVICE_HTTP
- AFX_INET_SERVICE_HTTPS
- AFX_INET_SERVICE_GOPHER
- AFX_INET_SERVICE_FILE
- AFX_INET_SERVICE_MAILTO
- AFX_INET_SERVICE_NEWS
- AFX_INET_SERVICE_NNTP
- AFX_INET_SERVICE_TELNET
- AFX_INET_SERVICE_WAIS
- AFX_INET_SERVICE_MID
- AFX_INET_SERVICE_CID
- AFX_INET_SERVICE_PROSPERO
- AFX_INET_SERVICE_AFS
- AFX_INET_SERVICE_UNK

strServer

The first segment of the URL following the service type.

strObject

An object that the URL refers to (may be empty).

nPort

Determined from either the Server or Object portions of the URL, if either exists.

Return Value

Nonzero if the URL was successfully parsed; otherwise, 0 if it is empty or does not contain a known Internet service type.

Remarks

It parses a URL string and returns the type of service and its components.

For example, **AfxParseURL** parses URLs of the form **service://server/dir/dir/object.ext:port** and returns its components stored as follows:

strServer == "server"

strObject == "/dir/dir/object/object.ext"

nPort == #port

dwServiceType == #service

Note
To call this function, your project must include AFXINET.H.

See Also

Reference

AfxGetInternetHandleType

AfxParseURLEx

Concepts

MFC Macros and Globals

AfxParseURLEx

This global function is the extended version of AfxParseURL and is used in CInternetSession::OpenURL.

```
BOOL AFXAPI AfxParseURLEx(
   LPCTSTR pstrURL,
   DWORD& dwServiceType,
   CString& strServer,
   CString& strObject,
   INTERNET_PORT& nPort,
   CString& strUsername,
   CString& strPassword,
   DWORD dwFlags = 0
);
```

Parameters

pstrURL

A pointer to a string containing the URL to be parsed.

dwServiceType

Indicates the type of Internet service. Possible values are as follows:

- AFX_INET_SERVICE_FTP
- AFX_INET_SERVICE_HTTP
- AFX_INET_SERVICE_HTTPS
- AFX_INET_SERVICE_GOPHER
- AFX_INET_SERVICE_FILE
- AFX_INET_SERVICE_MAILTO
- AFX_INET_SERVICE_NEWS
- AFX_INET_SERVICE_NNTP
- AFX_INET_SERVICE_TELNET
- AFX_INET_SERVICE_WAIS
- AFX_INET_SERVICE_MID
- AFX_INET_SERVICE_CID
- AFX_INET_SERVICE_PROSPERO
- AFX_INET_SERVICE_AFS
- AFX_INET_SERVICE_UNK

strServer

The first segment of the URL following the service type.

strObject

An object that the URL refers to (may be empty).

nPort

Determined from either the Server or Object portions of the URL, if either exists.

strUsername

A reference to a **CString** object containing the name of the user.

strPassword

A reference to a **CString** object containing the password of the user.

dwFlags

The flags controlling how to parse the URL. Can be a combination of the following values:

Value	Meaning
ICU_DECODE	Convert %XX escape sequences to characters.
ICU_NO_ENCODE	Do not convert unsafe characters to escape sequence.
ICU_NO_META	Do not remove meta sequences (such as "\ ." and "\ ..") from the URL.
ICU_ENCODE_SPACES_ONLY	Encode spaces only.
ICU_BROWSER_MODE	Do not encode or decode characters after '#' or '?', and do not remove trailing white space after '?'. If this value is not specified, the entire URL is encoded and trailing white space is removed.

If you use the MFC default, which is no flags, the function converts all unsafe characters and meta sequences (such as \.,\ .., and \...) to escape sequences.

Return Value

Nonzero if the URL was successfully parsed; otherwise, 0 if it is empty or does not contain a known Internet service type.

Remarks

It parses a URL string and returns the type of service and its components, as well as providing the user's name and password. The flags indicate how unsafe characters are handled.

Note
To call this function, your project must include AFXINET.H.

See Also

Reference

AfxGetInternetHandleType

Concepts

MFC Macros and Globals

AfxRegisterClass

Use this function to register window classes in a DLL that uses MFC.

```
BOOL AFXAPI AfxRegisterClass(
   WNDCLASS* lpWndClass
);
```

Parameters

lpWndClass

Pointer to a WNDCLASS structure containing information about the window class to be registered. For more information on this structure, see the *Platform SDK*.

Return Value

TRUE if the class is successfully registered; otherwise **FALSE**.

Remarks

If you use this function, the class is automatically unregistered when the DLL is unloaded.

In non-DLL builds, the **AfxRegisterClass** identifier is defined as a macro that maps to the Windows function **RegisterClass**, since classes registered in an application are automatically unregistered. If you use **AfxRegisterClass** instead of **RegisterClass**, your code can be used without change both in an application and in a DLL.

Example

```
BOOL COneT32App::InitInstance()
{

    // Register your unique class name that you wish to use
    WNDCLASS wndcls;

    memset(&wndcls, 0, sizeof(WNDCLASS));   // start with NULL
                                            // defaults

    wndcls.style = CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;

    //you can specify your own window procedure
    wndcls.lpfnWndProc = ::DefWindowProc;
    wndcls.hInstance = AfxGetInstanceHandle();
    wndcls.hIcon = LoadIcon(IDR_MAINFRAME); // or load a different icon
    wndcls.hCursor = LoadCursor( IDC_ARROW );
    wndcls.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
    wndcls.lpszMenuName = NULL;

    // Specify your own class name for using FindWindow later
    wndcls.lpszClassName = _T("MyNewClass");

    // Register the new class and exit if it fails
    if(!AfxRegisterClass(&wndcls))
    {
       TRACE("Class Registration Failed\n");
       return FALSE;
    }

    // Rest of InitInstance goes here
    ...
}
```

See Also

Concepts

MFC Macros and Globals

AfxRegisterWndClass

Allows you to register your own window classes.

```
LPCTSTR AFXAPI AfxRegisterWndClass(
   UINT nClassStyle,
   HCURSOR hCursor = 0,
   HBRUSH hbrBackground = 0,
   HICON hIcon = 0
);
```

Parameters

nClassStyle

Specifies the Windows class style or combination of styles, created by using the bitwise-OR (**|**) operator, for the window class. For a list of class styles, see the WNDCLASS structure in the *Platform SDK*. If **NULL**, the defaults will be set as follows:

- Sets the mouse style to **CS_DBLCLKS**, which sends double-click messages to the window procedure when the user double-clicks the mouse.
- Sets the arrow cursor style to the Windows standard **IDC_ARROW**.
- Sets the background brush to **NULL**, so the window will not erase its background.
- Sets the icon to the standard, waving-flag Windows logo icon.

hCursor

Specifies a handle to the cursor resource to be installed in each window created from the window class. If you use the default of **0**, you will get the standard **IDC_ARROW** cursor.

hbrBackground

Specifies a handle to the brush resource to be installed in each window created from the window class. If you use the default of **0**, you will have a **NULL** background brush, and your window will, by default, not erase its background while processing WM_ERASEBKGND.

hIcon

Specifies a handle to the icon resource to be installed in each window created from the window class. If you use the default of **0**, you will get the standard, waving-flag Windows logo icon.

Return Value

A null-terminated string containing the class name. You can pass this class name to the **Create** member function in **CWnd** or other **CWnd-**derived classes to create a window. The name is generated by the Microsoft Foundation Class Library.

Note
The return value is a pointer to a static buffer. To save this string, assign it to a **CString** variable.

Remarks

The Microsoft Foundation Class Library automatically registers several standard window classes for you. Call this function if you want to register your own window classes.

The name registered for a class by **AfxRegisterWndClass** depends solely on the parameters. If you call **AfxRegisterWndClass** multiple times with identical parameters, it only registers a class on the first call. Subsequent calls to **AfxRegisterWndClass** with identical parameters simply return the already-registered classname.

If you call **AfxRegisterWndClass** for multiple CWnd-derived classes with identical parameters, instead of getting a separate window class for each class, each class shares the same window class. This can cause problems if the **CS_CLASSDC** class style is used. Instead of multiple **CS_CLASSDC** window classes, you end up with one **CS_CLASSDC** window class, and all C++ windows that use that class share the same DC. To avoid this problem, call AfxRegisterClass to register the class.

Refer to Technical Note TN001: Window Class Registration for more information on window class registration and the

AfxRegisterWndClass function.

Example

```
CString strMyClass;

// load stock cursor, brush, and icon for
// my own window class

try
{
   strMyClass = AfxRegisterWndClass(
      CS_VREDRAW | CS_HREDRAW,
      ::LoadCursor(NULL, IDC_ARROW),
      (HBRUSH) ::GetStockObject(WHITE_BRUSH),
      ::LoadIcon(NULL, IDI_APPLICATION));
}
catch (CResourceException* pEx)
{
      AfxMessageBox(
         _T("Couldn't register class! (Already registered?)"));
      pEx->Delete();
}
```

See Also

Reference

CWnd::Create

CWnd::PreCreateWindow

WNDCLASS

AfxRegisterClass

Concepts

MFC Macros and Globals

_AFX_SECURE_NO_WARNINGS

Suppresses compiler warnings for the use of deprecated MFC functions.

```
_AFX_SECURE_NO_WARNINGS
```

Example

This code sample would cause a compiler warning if _AFX_SECURE_NO_WARNINGS were not defined.

```
// afx_secure_no_warnings.cpp
// compile with: /EHsc
#define _AFX_SECURE_NO_WARNINGS
#include "afxcmn.h"

int main() {
   CRichEditCtrl * m_richedit = new CRichEditCtrl;
   m_richedit->GetSelText("aa");
}
```

See Also

Concepts

MFC Macros and Globals

AfxSetAllocHook

Sets a hook that enables calling of the specified function before each memory block is allocated.

```
AFX_ALLOC_HOOK AfxSetAllocHook(
   AFX_ALLOC_HOOK pfnAllocHook
);
```

Parameters

pfnAllocHook

Specifies the name of the function to call. See the Remarks for the prototype of an allocation function.

Return Value

Nonzero if you want to permit the allocation; otherwise 0.

Remarks

The Microsoft Foundation Class Library debug-memory allocator can call a user-defined hook function to allow the user to monitor a memory allocation and to control whether the allocation is permitted. Allocation hook functions are prototyped as follows:

BOOL AFXAPI AllocHook(size_t *nSize*, **BOOL** *bObject*, **LONG** *lRequestNumber* **);**

nSize

The size of the proposed memory allocation.

bObject

TRUE if the allocation is for a **CObject**-derived object; otherwise **FALSE**.

lRequestNumber

The memory allocation's sequence number.

Note that the **AFXAPI** calling convention implies that the callee must remove the parameters from the stack.

See Also

Reference

AfxMessageBox

Concepts

MFC Macros and Globals

AfxSetAmbientActCtx

Use this function to set the per-module state flag, which affects the WinSxS behavior of MFC.

```
void AFXAPI AfxSetAmbientActCtx(
      BOOL bSet
);
```

Parameters

bSet

New value of the module state flag.

Remarks

When the flag is set (which is the default) and a thread enters an MFC module (see AFX_MANAGE_STATE), the context of the module is activated.

If the flag is not set, the context of the module is not activated on entry.

The context of a module is determined from its manifest, usually embedded in module resources.

Example

```
BOOL CMyApp::InitInstance()
{
   AfxSetAmbientActCtx(FALSE);
   ...
}
```

See Also

Reference

AfxGetAmbientActCtx

AFX_MANAGE_STATE

Concepts

MFC Macros and Globals

Managing the State Data of MFC Modules

AfxSetResourceHandle

Use this function to set the **HINSTANCE** handle that determines where the default resources of the application are loaded.

```
void AFXAPI AfxSetResourceHandle(
   HINSTANCE hInstResource
);
```

Parameters

hInstResource

The instance or module handle to an .EXE or DLL file from which the application's resources are loaded.

Example

```
BOOL CMyApp::InitInstance()
{
   //Default Application Wizard code.
   HINSTANCE hRes = NULL;
   hRes= LoadLibrary("ResourceD.dll");
   if(hRes)
      AfxSetResourceHandle(hRes);
   //Rest of wizard code
   return CWinApp::InitInstance();
}
```

See Also

Reference

AfxGetInstanceHandle

AfxGetResourceHandle

Concepts

MFC Macros and Globals

AfxSocketInit

Call this function in your **CWinApp::InitInstance** override to initialize Windows Sockets.

```
BOOL AfxSocketInit(
   WSADATA* lpwsaData = NULL
);
```

Parameters

lpwsaData

A pointer to a WSADATA structure. If *lpwsaData* is not equal to **NULL**, then the address of the **WSADATA** structure is filled by the call to **WSAStartup**. This function also ensures that **WSACleanup** is called for you before the application terminates.

Return Value

Nonzero if the function is successful; otherwise 0.

Example

```
BOOL CMyApp::InitInstance()
{
   //Default Application Wizard code.
   if(!AfxSocketInit())
      AfxMessageBox("Failed to Initialize Sockets",MB_OK| MB_ICONSTOP);
   return CWinApp::InitInstance();
}
```

See Also

Reference

CWinApp::InitInstance

Concepts

MFC Macros and Globals

AFX_SQL_ASYNC

The implementation of this macro changed in MFC 4.2.

```
AFX_SQL_ASYNC(prs, SQLFunc )
```

Parameters

prs

A pointer to a **CRecordset** object or a **CDatabase** object. Beginning with MFC 4.2, this parameter value is ignored.

SQLFunc

An ODBC API function. For more information about ODBC API functions, see the *Platform SDK*.

Remarks

AFX_SQL_ASYNC simply calls the macro AFX_ODBC_CALL and ignores the *prs* parameter. In versions of MFC prior to 4.2, **AFX_SQL_ASYNC** was used to call ODBC API functions that might return **SQL_STILL_EXECUTING**. If an ODBC API function did return **SQL_STILL_EXECUTING**, then **AFX_SQL_ASYNC** would call `prs->OnWaitForDataSource`.

Note
The MFC ODBC classes now use only synchronous processing. In order to perform an asynchronous operation, you must call the ODBC API function **SQLSetConnectOption**. For more information, see the topic "Executing Functions Asynchronously" in the *Platform SDK*.

See Also

Reference

AFX_ODBC_CALL

AFX_SQL_SYNC

Concepts

MFC Macros and Globals

AFX_SQL_SYNC

The **AFX_SQL_SYNC** macro simply calls the function *SQLFunc*.

```
AFX_SQL_SYNC(SQLFunc )
```

Parameters

SQLFunc

An ODBC API function. For more information about these functions, see the *Platform SDK*.

Remarks

Use this macro to call ODBC API functions that will not return **SQL_STILL_EXECUTING**.

Before calling **AFX_SQL_SYNC**, you must declare a variable, `nRetCode`, of type **RETCODE**. You can check the value of `nRetCode` after the macro call.

Note that the implementation of **AFX_SQL_SYNC** changed in MFC 4.2. Because checking the server status was no longer required, **AFX_SQL_SYNC** simply assigns a value to `nRetCode`. For example, instead of making the call

```
    AFX_SQL_SYNC( ::SQLGetInfo( .. ) )
```

you can simply make the assignment

```
    nRetCode = ::SQLGetInfo( .. );
```

See Also

Reference

AFX_SQL_ASYNC

AFX_ODBC_CALL

Concepts

MFC Macros and Globals

AfxTermExtensionModule

Call this function to allow MFC to cleanup the extension DLL when each process detaches from the DLL (which happens when the process exits, or when the DLL is unloaded as a result of a **AfxFreeLibrary** call).

```
void AFXAPI AfxTermExtensionModule(
   AFX_EXTENSION_MODULE& state,
   BOOL bAll  = FALSE
);
```

Parameters

state

A reference to the AFX_EXTENSION_MODULE structure that contains the state of extension DLL module.

bAll

If **TRUE**, cleanup all extension DLL modules. Otherwise, cleanup only the current DLL module.

Remarks

AfxTermExtensionModule will delete any local storage attached to the module and remove any entries from the message map cache. For example:

```
static AFX_EXTENSION_MODULE extensionDLL;
extern "C" int APIENTRY
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID)
{
   if (dwReason == DLL_PROCESS_ATTACH)
   {
      // Extension DLL one-time initialization
      if (!AfxInitExtensionModule( extensionDLL, hInstance))
         return 0;
      // TODO: perform other initialization tasks here
   }
   else if (dwReason == DLL_PROCESS_DETACH)
   {
      // Extension DLL per-process termination
      AfxTermExtensionModule(extensionDLL);
     // TODO: perform other cleanup tasks here
   }
   return 1;   // ok
}
```

If your application loads and frees extension DLLs dynamically, be sure to call **AfxTermExtensionModule**. Since most extension DLLs are not dynamically loaded (usually, they are linked via their import libraries), the call to **AfxTermExtensionModule** is usually not necessary.

MFC extension DLLs need to call AfxInitExtensionModule in their **DllMain**. If the DLL will be exporting CRuntimeClass objects or has its own custom resources, you also need to create a **CDynLinkLibrary** object in **DllMain**.

See Also

Reference

AfxInitExtensionModule

Concepts

MFC Macros and Globals

AfxThrowArchiveException

Throws an archive exception.

```
void AfxThrowArchiveException(
   int cause,
   LPCTSTR lpszArchiveName
);
```

Parameters

cause

Specifies an integer that indicates the reason for the exception. For a list of the possible values, see CArchiveException::m_cause.

lpszArchiveName

Points to a string containing the name of the **CArchive** object that caused the exception (if available).

See Also

Reference

CArchiveException Class

THROW (MFC)

Concepts

MFC Macros and Globals

AfxThrowDaoException

Call this function to throw an exception of type CDaoException from your own code.

```
void AFXAPI AfxThrowDaoException(
   int nAfxDaoError = NO_AFX_DAO_ERROR,
   SCODE scode = S_OK
);
```

Parameters

nAfxDaoError

An integer value representing a DAO extended error code, which can be one of the values listed under CDaoException::m_nAfxDaoError.

scode

An OLE error code from DAO, of type **SCODE**. For information, see CDaoException::m_scode.

Remarks

The framework also calls **AfxThrowDaoException**. In your call, you can pass one of the parameters or both. For example, if you want to raise one of the errors defined in **CDaoException::nAfxDaoError** but you do not care about the *scode* parameter, pass a valid code in the *nAfxDaoError* parameter and accept the default value for *scode*.

For information about exceptions related to the MFC DAO classes, see class **CDaoException** in this book and the article Exceptions: Database Exceptions.

See Also

Reference

CException Class

Concepts

MFC Macros and Globals

AfxThrowDBException

Call this function to throw an exception of type **CDBException** from your own code.

```
void AfxThrowDBException(
   RETCODE nRetCode,
   CDatabase* pdb,
   HSTMT hstmt
);
```

Parameters

nRetCode

A value of type **RETCODE**, defining the type of error that caused the exception to be thrown.

pdb

A pointer to the **CDatabase** object that represents the data source connection with which the exception is associated.

hstmt

An ODBC **HSTMT** handle that specifies the statement handle with which the exception is associated.

Remarks

The framework calls **AfxThrowDBException** when it receives an ODBC **RETCODE** from a call to an ODBC API function and interprets the **RETCODE** as an exceptional condition rather than an expectable error. For example, a data access operation might fail because of a disk read error.

For information about the **RETCODE** values defined by ODBC, see Chapter 8, "Retrieving Status and Error Information," in the Platform SDK. For information about MFC extensions to these codes, see class CDBException.

See Also

Reference

CDBException::m_nRetCode

Concepts

MFC Macros and Globals

AfxThrowFileException

Throws a file exception.

```
void AfxThrowFileException(
   int cause,
   LONG lOsError = -1,
   LPCTSTR lpszFileName = NULL
);
```

Parameters

cause

Specifies an integer that indicates the reason for the exception. For a list of the possible values, see CFileException::m_cause.

lOsError

Contains the operating-system error number (if available) that states the reason for the exception. See your operating-system manual for a listing of error codes.

lpszFileName

Points to a string containing the name of the file that caused the exception (if available).

Remarks

You are responsible for determining the cause based on the operating-system error code.

See Also

Reference

CFileException::ThrowOsError

THROW (MFC)

Concepts

MFC Macros and Globals

AfxThrowInternetException

Throws an Internet exception.

```
void AFXAPI AfxThrowInternetException(
   DWORD dwContext,
   DWORD dwError = 0
);
```

Parameters

dwContext

The context identifier for the operation that caused the error. The default value of *dwContext* is specified originally in CInternetSession and is passed to CInternetConnection- and CInternetFile-derived classes. For specific operations performed on a connection or a file, you usually override the default with a *dwContext* of your own. This value then is returned to CInternetSession::OnStatusCallback to identify the specific operation's status. For more information on context identifiers, see the article Internet First Steps: WinInet.

dwError

The error that caused the exception.

Remarks

You are responsible for determining the cause based on the operating-system error code.

Note
To call this function, your project must include AFXINET.H.

See Also

Reference

CInternetException Class

THROW (MFC)

Concepts

MFC Macros and Globals

AfxThrowInvalidArgException

Throws an invalid argument exception.

```
void AfxThrowInvalidArgException( );
```

Remarks

This function is called when invalid arguments are used.

See Also

Reference

CInvalidArgException Class

THROW (MFC)

Concepts

MFC Macros and Globals

AfxThrowMemoryException

Throws a memory exception.

```
void AfxThrowMemoryException( );
```

Remarks

Call this function if calls to underlying system memory allocators (such as **malloc** and the GlobalAlloc Windows function) fail. You do not need to call it for **new** because **new** will throw a memory exception automatically if the memory allocation fails.

See Also

Reference

CMemoryException Class

THROW (MFC)

Concepts

MFC Macros and Globals

AfxThrowNotSupportedException

Throws an exception that is the result of a request for an unsupported feature.

```
void AfxThrowNotSupportedException( );
```

See Also

Reference

CNotSupportedException Class

THROW (MFC)

Concepts

MFC Macros and Globals

AfxThrowOleDispatchException

Use this function to throw an exception within an OLE automation function.

```
void AFXAPI AfxThrowOleDispatchException(
   WORD wCode,
   LPCSTR lpszDescription,
   UINT nHelpID = 0
);
void AFXAPI AfxThrowOleDispatchException(
   WORD wCode,
   UINT nDescriptionID,
   UINT nHelpID =  -1
);
```

Parameters

wCode

An error code specific to your application.

lpszDescription

Verbal description of the error.

nDescriptionID

Resource ID for the verbal error description.

nHelpID

A help context for your application's help (.HLP) file.

Remarks

The information provided to this function can be displayed by the driving application (Microsoft Visual Basic or another OLE automation client application).

Example

```
// Sort is method of automation class CStrArrayDoc
long CStrArrayDoc::Sort(VARIANT FAR* vArray)
{
   ...
   ...
   USES_CONVERSION;

   // Type check VARIANT parameter. It should contain a BSTR array
   // passed by reference. The array must be passed by reference; it is
   // an in-out-parameter.

   // throwing COleDispatchException allows the EXCEPINFO structure of
   // IDispatch::Invoke() to set
   if (V_VT(vArray) != (VT_ARRAY | VT_BSTR))
      AfxThrowOleDispatchException(1001, "Type Mismatch in Parameter.
         Pass a string array by reference");
   ...
   ...
}
```

Requirements

Header: <afxdisp.h>

See Also

Reference

COleException Class

Concepts

MFC Macros and Globals

AfxThrowOleException

Creates an object of type **COleException** and throws an exception.

```
void AFXAPI AfxThrowOleException(
   SCODE sc
);
void AFXAPI AfxThrowOleException(
   HRESULT hr
);
```

Parameters

sc

An OLE status code that indicates the reason for the exception.

hr

Handle to a result code that indicates the reason for the exception.

Remarks

The version that takes an **HRESULT** as an argument converts that result code into the corresponding **SCODE**. For more information on **HRESULT** and **SCODE**, see Structure of COM Error Codes in the *Platform SDK*.

Requirements

Header: <afxdisp.h>

See Also

Reference

COleException Class

THROW (MFC)

Concepts

MFC Macros and Globals

AfxThrowResourceException

Throws a resource exception.

```
void AfxThrowResourceException( );
```

Remarks

This function is normally called when a Windows resource cannot be loaded.

See Also

Reference

CResourceException Class

THROW (MFC)

Concepts

MFC Macros and Globals

AfxThrowUserException

Throws an exception to stop an end-user operation.

```
void AfxThrowUserException( );
```

Remarks

This function is normally called immediately after **AfxMessageBox** has reported an error to the user.

See Also

Reference

CUserException Class
THROW (MFC)
AfxMessageBox

Concepts

MFC Macros and Globals

AfxVerifyLicFile

Call this function to verify that the license file named by *pszLicFileName* is valid for the OLE control.

```
BOOL AFXAPI AfxVerifyLicFile(
   HINSTANCE hInstance,
   LPCTSTR pszLicFileName,
   LPOLESTR pszLicFileContents,
   UINT cch = -1
);
```

Parameters

hInstance

The instance handle of the DLL associated with the licensed control.

pszLicFileName

Points to a null-terminated character string containing the license filename.

pszLicFileContents

Points to a byte sequence that must match the sequence found at the beginning of the license file.

cch

Number of characters in *pszLicFileContents*.

Return Value

Nonzero if the license file exists and begins with the character sequence in *pszLicFileContents*; otherwise 0.

Remarks

If *cch* is – 1, this function uses:

```
_tcslen(pszLicFileContents)
```

See Also

Reference

COleObjectFactory::VerifyUserLicense

Concepts

MFC Macros and Globals

AfxWinInit

This function is called by the MFC-supplied **WinMain** function, as part of the CWinApp initialization of a GUI-based application, to initialize MFC.

```
BOOL AFXAPI AfxWinInit(
   HINSTANCE hInstance,
   HINSTANCE hPrevInstance,
   LPTSTR lpCmdLine,
   int nCmdShow
);
```

Parameters

hInstance

The handle of the currently running module.

hPrevInstance

A handle to a previous instance of the application. For a Win32-based application, this parameter is always **NULL**.

lpCmdLine

Points to a null-terminated string specifying the command line for the application.

nCmdShow

Specifies how the main window of a GUI application would be shown.

Remarks

For a console application, which does not use the MFC-supplied **WinMain** function, you must call **AfxWinInit** directly to initialize MFC.

If you call **AfxWinInit** yourself, you should declare an instance of a **CWinApp** class. For a console application, you might choose not to derive your own class from **CWinApp** and instead use an instance of **CWinApp** directly. This technique is appropriate if you decide to leave all functionality for your application in your implementation of **main**.

Note

When it creates an activation context for an assembly, MFC uses a manifest resource provided by the user module. The activation context is created in **AfxWinInit**. For more information, see Support for Activation Contexts in the MFC Module State.

Example

```
// this file must be compiled with the /GX and /MT options:
//      cl /GX /MT thisfile.cpp

#include <afx.h>
#include <afxdb.h>
#include <iostream.h>

int main()
{
   // try to initialize MFC

   if (!AfxWinInit(::GetModuleHandle(NULL), NULL, ::GetCommandLine(), 0))
   {
      cerr << "MFC failed to initialize!" << endl;
      return 1;
   }

   // try to connect to an ODBC database that doesn't exist
   // (this wouldn't work at all without initializing MFC)

   CDatabase db;
```

```
    try
    {
       db.Open("This Databsae Doesn't Exist");

       // we shouldn't realistically get here

       cout << "Successful!" << endl;
       cout << "Closing ... ";
       db.Close();
       cout << "Closed!" << endl;
    }
    catch (CDBException* pEx)
    {
       // we got an exception! print an error message
       // (this wouldn't work without initializing MFC)

       char sz[1024];

       cout << "Error: ";
       if (pEx->GetErrorMessage(sz, 1024))
          cout << sz;
       else
          cout << "No error message was available";
       cout << endl;

       pEx->Delete();
       return 1;
    }

    return 0;
 }
```

See Also

Reference

CWinApp Class

main: Program Startup

WinMain

Concepts

MFC Macros and Globals

CWinApp: The Application Class

AND_CATCH

Defines a block of code for catching additional exception types thrown in a preceding **TRY** block.

```
AND_CATCH(exception_class, exception_object_pointer_name )
```

Parameters

exception_class

Specifies the exception type to test for. For a list of standard exception classes, see class CException.

exception_object_pointer_name

A name for an exception-object pointer that will be created by the macro. You can use the pointer name to access the exception object within the **AND_CATCH** block. This variable is declared for you.

Remarks

Use the **CATCH** macro to catch one exception type, then the **AND_CATCH** macro to catch each subsequent type. End the **TRY** block with an **END_CATCH** macro.

The exception-processing code can interrogate the exception object, if appropriate, to get more information about the specific cause of the exception. Call the **THROW_LAST** macro within the **AND_CATCH** block to shift processing to the next outer exception frame. **AND_CATCH** marks the end of the preceding **CATCH** or **AND_CATCH** block.

Note
The **AND_CATCH** block is defined as a C++ scope (delineated by curly braces). If you declare variables in this scope, remem ber that they are accessible only within that scope. This also applies to the *exception_object_pointer_name* variable.

Example

See the example for CATCH.

See Also

Reference

TRY
CATCH
END_CATCH
THROW (MFC)
THROW_LAST
AND_CATCH_ALL
CException Class

Concepts

MFC Macros and Globals

AND_CATCH_ALL

Defines a block of code for catching additional exception types thrown in a preceding **TRY** block.

```
AND_CATCH_ALL(exception_object_pointer_name )
```

Parameters

exception_object_pointer_name

A name for an exception-object pointer that will be created by the macro. You can use the pointer name to access the exception object within the **AND_CATCH_ALL** block. This variable is declared for you.

Remarks

Use the **CATCH** macro to catch one exception type, then the **AND_CATCH_ALL** macro to catch all other subsequent types. If you use **AND_CATCH_ALL**, end the **TRY** block with an **END_CATCH_ALL** macro.

The exception-processing code can interrogate the exception object, if appropriate, to get more information about the specific cause of the exception. Call the **THROW_LAST** macro within the **AND_CATCH_ALL** block to shift processing to the next outer exception frame. **AND_CATCH_ALL** marks the end of the preceding **CATCH** or **AND_CATCH_ALL** block.

Note

The **AND_CATCH_ALL** block is defined as a C++ scope (delineated by braces). If you declare variables in this scope, remember that they are accessible only within that scope.

See Also

Reference

TRY
CATCH_ALL
END_CATCH_ALL
THROW (MFC)
THROW_LAST
AND_CATCH
CException Class

Concepts

MFC Macros and Globals

ASSERT (MFC)

Evaluates its argument.

```
ASSERT(
      booleanExpression
)
```

Parameters

booleanExpression

Specifies an expression (including pointer values) that evaluates to nonzero or 0.

Remarks

If the result is 0, the macro prints a diagnostic message and aborts the program. If the condition is nonzero, it does nothing.

The diagnostic message has the form

```
assertion failed in file <name> in line <num>
```

where *name* is the name of the source file, and *num* is the line number of the assertion that failed in the source file.

In the Release version of MFC, **ASSERT** does not evaluate the expression and thus will not interrupt the program. If the expression must be evaluated regardless of environment, use the **VERIFY** macro in place of **ASSERT**.

Note
This function is available only in the Debug version of MFC.

In an MFC ISAPI application, an assertion in debug mode will bring up a modal dialog box (ASSERT dialog boxes are now modal by default); this will interrupt or hang the execution. To suppress modal assertion dialogs, add the following lines to your project source file (*projectname*.cpp):

```
// For custom assert and trace handling with WebDbg
#ifdef _DEBUG
CDebugReportHook g_ReportHook;
#endif
```

Once you have done this, you can use the WebDbg tool (WebDbg.exe) to see the assertions. For information on using the WebDbg tool, see Viewing Trace Messages And Handling Asserts.

Example

```
// example for ASSERT
CAge* pcage = new CAge( 21 ); // CAge is derived from CObject.
ASSERT( pcage!= NULL )
ASSERT( pcage->IsKindOf( RUNTIME_CLASS( CAge ) ) )
// Terminates program only if pcage is NOT a CAge*.
```

See Also

Reference

VERIFY

Concepts

MFC Macros and Globals

ASSERT_KINDOF

This macro asserts that the object pointed to is an object of the specified class, or is an object of a class derived from the specified class.

```
ASSERT_KINDOF(classname, pobject )
```

Parameters

classname

The name of a **CObject**-derived class.

pobject

A pointer to a class object.

Remarks

The *pobject* parameter should be a pointer to an object and can be **const**. The object pointed to and the class must support **CObject** run-time class information. As an example, to ensure that `pDocument` is a pointer to an object of the `CMyDocument` class, or any of its derivatives, you could code:

```
ASSERT_KINDOF(CMyDocument, pDocument)
```

Using the **ASSERT_KINDOF** macro is exactly the same as coding:

```
ASSERT(pobject->IsKindOf(RUNTIME_CLASS(classname)));
```

This function works only for classes declared with the DECLARE_DYNAMIC or DECLARE_SERIAL macro.

Note
This function is available only in the Debug version of MFC.

See Also

Reference

ASSERT (MFC)

Concepts

MFC Macros and Globals

ASSERT_VALID

Use to test your assumptions about the validity of an object's internal state.

```
ASSERT_VALID(pObject )
```

Parameters

pObject

Specifies an object of a class derived from **CObject** that has an overriding version of the **AssertValid** member function.

Remarks

ASSERT_VALID calls the **AssertValid** member function of the object passed as its argument.

In the Release version of MFC, **ASSERT_VALID** does nothing. In the Debug version, it validates the pointer, checks against **NULL**, and calls the object's own **AssertValid** member functions. If any of these tests fails, an alert message is displayed in the same manner as ASSERT.

Note
This function is available only in the Debug version of MFC.

For more information and examples, see Debugging MFC Applications.

Example

```
// Assure that pMyObject is a valid pointer to an
// object derived from CObject.
ASSERT_VALID(pMyObject);
```

See Also

Reference

ASSERT (MFC)
VERIFY
CObject Class
CObject::AssertValid

Concepts

MFC Macros and Globals

BASED_CODE

Under Win32, this macro expands to nothing and is provided for backward compatibility.

Under 16-bit MFC, the macro ensures that data will be placed in the code segment rather than in the data segment. The result is less impact on your data segment.

See Also

Reference

AfxMessageBox

Concepts

MFC Macros and Globals

BEGIN_CONNECTION_MAP

Each **COleControl**-derived class in your program can provide a connection map to specify connection points that your control will support.

```
BEGIN_CONNECTION_MAP(theClass, theBase )
```

Parameters

theClass

Specifies the name of the control class whose connection map this is.

theBase

Specifies the name of the base class of *theClass*.

Remarks

In the implementation (.CPP) file that defines the member functions for your class, start the connection map with the **BEGIN_CONNECTION_MAP** macro, then add macro entries for each of your connection points using the CONNECTION_PART macro. Finally, complete the connection map with the END_CONNECTION_MAP macro.

See Also

Reference

BEGIN_CONNECTION_PART
DECLARE_CONNECTION_MAP

Concepts

MFC Macros and Globals

BEGIN_CONNECTION_PART

Use the **BEGIN_CONNECTION_PART** macro to begin the definition of additional connection points beyond the event and property notification connection points.

```
BEGIN_CONNECTION_PART(theClass, localClass )
```

Parameters

theClass

Specifies the name of the control class whose connection point this is.

localClass

Specifies the name of the local class that implements the connection point.

Remarks

In the declaration (.h) file that defines the member functions for your class, start the connection point with the **BEGIN_CONNECTION_PART** macro, then add the **CONNECTION_IID** macro and any other member functions you wish to implement, and complete the connection point map with the **END_CONNECTION_PART** macro.

See Also

Reference

BEGIN_CONNECTION_MAP
END_CONNECTION_PART
DECLARE_CONNECTION_MAP

Concepts

MFC Macros and Globals

BEGIN_DELEGATE_MAP

Begins a delegate map.

```
BEGIN_DELEGATE_MAP(
      CLASS
);
```

Parameters

CLASS

The class in which the managed control is hosted.

Remarks

This macro marks the beginning of a list of delegate entries, which compose a delegate map. For an example of how this macro is used, see EVENT_DELEGATE_ENTRY.

Requirements

Header: msclr\event.h

See Also

Tasks

How to: Sink Windows Forms Events from Native C++ Classes

Reference

MAKE_DELEGATE
EVENT_DELEGATE_ENTRY
END_DELEGATE_MAP

BEGIN_DHTMLEDITING_CMDMAP

Starts the definition of a DHTML editing command map within a class.

```
BEGIN_DHTMLEDITING_CMDMAP(className)
```

Parameters

className

The name of the class containing the DHTML editing command map. This class should derive directly or indirectly from CHtmlEditView and include the DECLARE_DHTMLEDITING_CMDMAP macro within its class definition.

Remarks

Add a DHTML editing command map to your class to map user interface commands to HTML editing commands.

Place the **BEGIN_DHTMLEDITING_CMDMAP** macro in the class's implementation (.cpp) file followed by DHTMLEDITING_CMD_ENTRY macros for the commands the class is to map (for example, from **ID_EDIT_CUT** to **IDM_CUT**). Use the END_DHTMLEDITING_CMDMAP macro to mark the end of the event map.

See Also

Concepts

MFC Macros and Globals

Other Resources

DHTML Editing Command Maps

BEGIN_DHTML_EVENT_MAP

Marks the beginning of the DHTML event map when placed in the source file for the class identified by *className*.

```
BEGIN_DHTML_EVENT_MAP(className )
```

Parameters

className

The name of the class containing the DHTML event map. This class should derive directly or indirectly from CDHtmlDialog and include the DECLARE_DHTML_EVENT_MAP macro within its class definition.

Remarks

Add a DHTML event map to your class to provide information to **CDHtmlDialog** that can be used to route events fired by HTML elements or ActiveX controls in a web page to handler functions in your class.

Place the **BEGIN_DHTML_EVENT_MAP** macro in the class's implementation (.cpp) file followed by **DHTML_EVENT** macros for the events the class is to handle (for example, **DHTML_EVENT_ONMOUSEOVER** for mouseover events). Use the END_DHTML_EVENT_MAP macro to mark the end of the event map. These macros implement the following function:

```
virtual const DHtmlEventMapEntry* GetDHtmlEventMap();
```

See Also

Reference

BEGIN_DHTML_EVENT_MAP_INLINE

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

BEGIN_DHTML_EVENT_MAP_INLINE

Marks the beginning of the DHTML event map within the class definition for *className*.

```
BEGIN_DHTML_EVENT_MAP_INLINE(className )
```

Parameters

className

The name of the class containing the DHTML event map. This class should derive directly or indirectly from CDHtmlDialog and include the DECLARE_DHTML_EVENT_MAP macro within its class definition.

Remarks

Add a DHTML event map to your class to provide information to **CDHtmlDialog** that can be used to route events fired by HTML elements or ActiveX controls in a web page to handler functions in your class.

Place the **BEGIN_DHTML_EVENT_MAP** macro in the class's definition (.h) file followed by **DHTML_EVENT** macros for the events the class is to handle (for example, **DHTML_EVENT_ONMOUSEOVER** for mouseover events). Use the END_DHTML_EVENT_MAP_INLINE macro to mark the end of the event map. These macros implement the following function:

```
virtual const DHtmlEventMapEntry* GetDHtmlEventMap();
```

See Also

Reference

BEGIN_DHTML_EVENT_MAP

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

BEGIN_DHTML_URL_EVENT_MAP

Starts the definition of a DHTML and URL event map in a multipage dialog.

```
BEGIN_DHTML_URL_EVENT_MAP( )
```

Remarks

Put **BEGIN_DHTML_URL_EVENT_MAP** in the implementation file of your CMultiPageDHtmlDialog-derived class. Follow it with embedded DHTML event maps and URL entries, and then close it with END_DHTML_URL_EVENT_MAP. Include the DECLARE_DHTML_URL_EVENT_MAP macro within the class definition.

Example

```
// Somewhere in CMyMultiPageDlg.cpp:
BEGIN_DHTML_URL_EVENT_MAP( CMyMultiPageDlg )
   // Define event map for "Page1" ( see URL_EVENT_ENTRY below )
   BEGIN_EMBED_DHTML_EVENT_MAP( CMyMultiPageDlg, Page1 )
      DHTML_EVENT_ONCLICK( _T( "Next" ), OnPage1Next )
   END_EMBED_DHTML_EVENT_MAP( )
   // Define event map for events from "Page2"
   BEGIN_EMBED_DHTML_EVENT_MAP( CMyMultiPageDlg, Page2 )
      DHTML_EVENT_ONCLICK( _T( "Back" ), OnPage2Back )
      DHTML_EVENT_ONCLICK( _T( "Next" ), OnPage2Next )
   END_EMBED_DHTML_EVENT_MAP( )
   BEGIN_URL_ENTRIES( CMyMultiPageDlg )
      // Map "Page1" to HTML resource 131
      URL_EVENT_ENTRY( CMyMultiPageDlg, _T( "131" ), Page1 )
      // Map "Page2" to HTML resource 132
      URL_EVENT_ENTRY( CMyMultiPageDlg, _T( "132" ), Page2 )
   END_URL_ENTRIES( )
END_DHTML_URL_EVENT_MAP( CMyMultiPageDlg )
```

See Also

Concepts

MFC Macros and Globals

Other Resources

Multipage DHTML and URL Event Maps

BEGIN_DISPATCH_MAP

Declares the definition of your dispatch map.

```
BEGIN_DISPATCH_MAP(theClass, baseClass )
```

Parameters

theClass

Specifies the name of the class that owns this dispatch map.

baseClass

Specifies the base class name of *theClass*.

Remarks

In the implementation (.cpp) file that defines the member functions for your class, start the dispatch map with the **BEGIN_DISPATCH_MAP** macro, add macro entries for each of your dispatch functions and properties, and complete the dispatch map with the **END_DISPATCH_MAP** macro.

See Also

Reference

DECLARE_DISPATCH_MAP
END_DISPATCH_MAP
DISP_FUNCTION
DISP_PROPERTY
DISP_PROPERTY_EX
DISP_DEFVALUE

Concepts

MFC Macros and Globals

Other Resources

Dispatch Maps

BEGIN_EMBED_DHTML_EVENT_MAP

Starts the definition of an embedded DHTML event map in a multipage dialog.

```
BEGIN_EMBED_DHTML_EVENT_MAP(className, mapName )
```

Parameters

className

The name of the class containing the event map. This class should derive directly or indirectly from CMultiPageDHtmlDialog. The embedded DHTML event map must be inside a DHTML and URL event map).

mapName

Specifies the page whose event map this is. This matches *mapName* in the URL_EVENT_ENTRY macro actually defining the URL or HTML resource.

Remarks

Because a multipage DHTML dialog consists of multiple HTML pages, each of which can raise DHTML events, embedded event maps are used to map events to handlers on a per-page basis.

Embedded event maps within a DHTML and URL event map consist of a **BEGIN_EMBED_DHTML_EVENT_MAP** macro followed by DHTML_EVENT macros and an END_EMBED_DHTML_EVENT_MAP macro.

Each embedded event map requires a corresponding URL event entry to map *mapName* (specified in **BEGIN_EMBED_DHTML_EVENT_MAP**) to a URL or HTML resource.

Example

See the example in BEGIN_DHTML_URL_EVENT_MAP.

See Also

Concepts

MFC Macros and Globals

Other Resources

Multipage DHTML and URL Event Maps

DHTML Event Maps

BEGIN_EVENT_MAP

Begins the definition of your event map.

```
BEGIN_EVENT_MAP(theClass, baseClass )
```

Parameters

theClass

Specifies the name of the control class whose event map this is.

baseClass

Specifies the name of the base class of *theClass*.

Remarks

In the implementation (.cpp) file that defines the member functions for your class, start the event map with the **BEGIN_EVENT_MAP** macro, then add macro entries for each of your events, and complete the event map with the **END_EVENT_MAP** macro.

For more information on event maps and the **BEGIN_EVENT_MAP** macro, see the article ActiveX Controls: Events.

See Also

Reference

DECLARE_EVENT_MAP

END_EVENT_MAP

Concepts

MFC Macros and Globals

BEGIN_EVENTSINK_MAP

Begins the definition of your event sink map.

```
BEGIN_EVENTSINK_MAP(theClass, baseClass )
```

Parameters

theClass

Specifies the name of the control class whose event sink map this is.

baseClass

Specifies the name of the base class of *theClass*.

Remarks

In the implementation (.cpp) file that defines the member functions for your class, start the event sink map with the **BEGIN_EVENTSINK_MAP** macro, then add macro entries for each event to be notified of, and complete the event sink map with the **END_EVENTSINK_MAP** macro.

For more information on event sink maps and OLE control containers, see the article ActiveX Control Containers.

See Also

Reference

DECLARE_EVENTSINK_MAP

END_EVENTSINK_MAP

Concepts

MFC Macros and Globals

BEGIN_INTERFACE_MAP

Begins the definition of the interfaced map when used in the implementation file.

```
BEGIN_INTERFACE_MAP(theClass, baseClass )
```

Parameters

theClass

The class in which the interface map is to be defined

baseClass

The class from which *theClass* derives from.

Remarks

For each interface that is implemented, there is one or more **INTERFACE_PART** macro invocations. For each aggregate that the class uses, there is one **INTERFACE_AGGREGATE** macro invocation.

For more information on interface maps, see Technical Note 38.

See Also

Reference

END_INTERFACE_MAP

INTERFACE_PART

Concepts

MFC Macros and Globals

BEGIN_MESSAGE_MAP

Begins the definition of your message map.

```
BEGIN_MESSAGE_MAP(theClass, baseClass )
```

Parameters

theClass

Specifies the name of the class whose message map this is.

baseClass

Specifies the name of the base class of *theClass*.

Remarks

In the implementation (.cpp) file that defines the member functions for your class, start the message map with the **BEGIN_MESSAGE_MAP** macro, then add macro entries for each of your message-handler functions, and complete the message map with the **END_MESSAGE_MAP** macro.

Example

```
// Example for BEGIN_MESSAGE_MAP
BEGIN_MESSAGE_MAP( CMyWindow, CFrameWnd )
    ON_WM_PAINT()
    ON_COMMAND( IDM_ABOUT, OnAbout )
END_MESSAGE_MAP( )
```

See Also

Reference

DECLARE_MESSAGE_MAP
END_MESSAGE_MAP
BEGIN_TEMPLATE_MESSAGE_MAP

Concepts

MFC Macros and Globals

BEGIN_TEMPLATE_MESSAGE_MAP

Begins the definition of a message map on a class type containing a single template argument.

```
BEGIN_TEMPLATE_MESSAGE_MAP(theClass, type_name, baseClass )
```

Parameters

theClass

Specifies the name of the class whose message map this is.

type_name

The name of the template parameter specified for the class.

baseClass

Specifies the name of the base class of *theClass*.

Remarks

This macro is similar to the BEGIN_MESSAGE_MAP macro; however, this macro is intended for classes containing a single template argument.

In the method implementation section of your class, start the message map with the **BEGIN_TEMPLATE_MESSAGE_MAP** macro; then add macro entries for each of your message-handler methods as you would for a standard message map. As with the **BEGIN_MESSAGE_MAP** macro, complete the template message map with the END_MESSAGE_MAP macro.

For more information on implementing message maps for template classes, refer to How to: Create a Message Map for a Template Class.

See Also

Reference

DECLARE_MESSAGE_MAP
BEGIN_MESSAGE_MAP
END_MESSAGE_MAP

Concepts

MFC Macros and Globals

BEGIN_OLEFACTORY

Begins the declaration of your class factory in the header file of your control class.

```
BEGIN_OLEFACTORY(class_name )
```

Parameters

class_name

Specifies the name of the control class whose class factory this is.

Remarks

Declarations of class factory licensing functions should begin immediately after **BEGIN_OLEFACTORY**.

See Also

Reference

END_OLEFACTORY

DECLARE_OLECREATE_EX

Concepts

MFC Macros and Globals

BEGIN_PARSE_MAP

Begins the definition of your parse map.

```
BEGIN_PARSE_MAP(theClass, baseClass )
```

Parameters

theClass

Specifies the name of the class that owns this parse map.

baseClass

Specifies the base class name of *theClass*. Must be a class derived from CHttpServer.

Remarks

When a client command is received by a **CHttpServer** object, the parse maps associate the command to its class member function and parameters. Only one parse map is created per **CHttpServer** object.

In the implementation (.cpp) file that defines the member functions for your class, start the parse map with the **BEGIN_PARSE_MAP** macro, add macro entries for each of your parse functions and properties, and complete the parse map with the END_PARSE_MAP macro.

Example

See ON_PARSE_COMMAND for a parse map example.

See Also

Reference

ON_PARSE_COMMAND
ON_PARSE_COMMAND_PARAMS
DEFAULT_PARSE_COMMAND
END_PARSE_MAP
CHttpServer Class

Concepts

MFC Macros and Globals

BEGIN_PROPPAGEIDS

Begins the definition of your control's list of property page IDs.

```
BEGIN_PROPPAGEIDS(class_name, count )
```

Parameters

class_name

The name of the control class for which property pages are being specified.

count

The number of property pages used by the control class.

Remarks

In the implementation (.cpp) file that defines the member functions for your class, start the property page list with the **BEGIN_PROPPAGEIDS** macro, then add macro entries for each of your property pages, and complete the property page list with the **END_PROPPAGEIDS** macro.

For more information on property pages, see the article ActiveX Controls: Property Pages.

See Also

Reference

END_PROPPAGEIDS
DECLARE_PROPPAGEIDS
PROPPAGEID

Concepts

MFC Macros and Globals

BEGIN_URL_ENTRIES

Starts the definition of a URL event entry map in a multipage dialog.

```
BEGIN_URL_ENTRIES(className )
```

Parameters

className

The name of the class containing the URL event entry map. This class should derive directly or indirectly from CMultiPageDHtmlDialog. The URL event entry map must be inside a DHTML and URL event map).

Remarks

Because a multipage DHTML dialog consists of multiple HTML pages, URL event entries are used to map URLs or HTML resources to corresponding embedded DHTML event maps. Put **URL_EVENT_ENTRY** macros between **BEGIN_URL_ENTRIES** and END_URL_ENTRIES macros.

Example

See the example in BEGIN_DHTML_URL_EVENT_MAP.

See Also

Concepts

MFC Macros and Globals

Other Resources

Multipage DHTML and URL Event Maps

CATCH

Defines a block of code that catches the first exception type thrown in the preceding **TRY** block.

```
CATCH(exception_class, exception_object_pointer_name )
```

Parameters

exception_class

Specifies the exception type to test for. For a list of standard exception classes, see class CException.

exception_object_pointer_name

Specifies a name for an exception-object pointer that will be created by the macro. You can use the pointer name to access the exception object within the **CATCH** block. This variable is declared for you.

Remarks

The exception-processing code can interrogate the exception object, if appropriate, to get more information about the specific cause of the exception. Invoke the **THROW_LAST** macro to shift processing to the next outer exception frame. End the **TRY** block with an **END_CATCH** macro.

If *exception_class* is the class **CException**, then all exception types will be caught. You can use the CObject::IsKindOf member function to determine which specific exception was thrown. A better way to catch several kinds of exceptions is to use sequential **AND_CATCH** statements, each with a different exception type.

The exception object pointer is created by the macro. You do not need to declare it yourself.

Note

The **CATCH** block is defined as a C++ scope delineated by braces. If you declare variables in this scope, they are accessible o nly within that scope. This also applies to *exception_object_pointer_name*.

For more information on exceptions and the **CATCH** macro, see the article Exceptions.

Example

```
CFile* pFile = NULL;
   // Constructing a CFile object with this override may throw
   // a CFile exception and won't throw any other exceptions.
   // Calling CString::Format( ) may throw a CMemoryException,
   // so we have a catch block for such exceptions, too. Any
   // other exception types this function throws will be
   // routed to the calling function.
   TRY
   {
      pFile = new CFile( _T( "C:\\WINDOWS\\SYSTEM.INI" ),
         CFile::modeRead | CFile::shareDenyNone );
      ULONGLONG dwLength = pFile->GetLength( );
      CString str;
      str.Format( _T( "Your SYSTEM.INI file is %I64u bytes long.") , dwLength );
      AfxMessageBox( str );
   }
   CATCH( CFileException, pEx )
   {
      // Simply show an error message to the user.
      pEx->ReportError();
   }
   AND_CATCH(CMemoryException, pEx)
   {
      // We can't recover from this memory exception, so we'll
      // just terminate the app without any cleanup. Normally,
      // an application should do everything it possibly can to
      // clean up properly and not call AfxAbort( ).
```

```
        AfxAbort( );
    }
    END_CATCH
    // If an exception occurs in the CFile constructor,
    // the language will free the memory allocated by new
    // and will not complete the assignment to pFile.
    // Thus, our cleanup code needs to test for NULL.
    if ( pFile != NULL )   {
        pFile->Close( );
        delete pFile;
    }
```

See Also

Reference

TRY
AND_CATCH
END_CATCH
THROW (MFC)
THROW_LAST
CATCH_ALL
CException Class

Concepts

MFC Macros and Globals

CATCH_ALL

Defines a block of code that catches all exception types thrown in the preceding **TRY** block.

```
CATCH_ALL(exception_object_pointer_name )
```

Parameters

exception_object_pointer_name

Specifies a name for an exception-object pointer that will be created by the macro. You can use the pointer name to access the exception object within the **CATCH_ALL** block. This variable is declared for you.

Remarks

The exception-processing code can interrogate the exception object, if appropriate, to get more information about the specific cause of the exception. Invoke the **THROW_LAST** macro to shift processing to the next outer exception frame. If you use **CATCH_ALL**, end the **TRY** block with an **END_CATCH_ALL** macro.

Note
The **CATCH_ALL** block is defined as a C++ scope delineated by braces. If you declare variables in this scope, they are accessible only within that scope.

For more information on exceptions, see the article Exceptions.

Example

See the example for CFile::Abort.

See Also

Reference

TRY
AND_CATCH_ALL
END_CATCH
THROW (MFC)
THROW_LAST
CATCH
CException Class

Concepts

MFC Macros and Globals

CommandHandler Delegate

Registers callback methods with a command source.

```
delegate void CommandHandler(
   UINT^ cmdID
);
```

Parameters

cmdID

The command ID.

Remarks

This delegate registers callback methods with a command source. When you add a delegate to the command source object, the callback method becomes a handler for commands coming from the specified source.

For more information, see How to: Add Command Routing to the Windows Forms Control.

For more information on using Windows Forms, see Using a Windows Form User Control in MFC.

Example

The following is an example of how to use **CommandHandler**.

```
private ICommandSource m_CmdSrc;

public void Initialize (ICommandSource cmdSrc)
{
      m_CmdSrc = cmdSrc;
      newHandler = new CommandHandler (singleMenuHandler);
      m_CmdSrc.AddCommandHandler (<resourceID>, newHandler);
}
```

Requirements

Header: afxwinforms.h (defined in assembly atlmfc\lib\mfcmifc80.dll)

See Also

Tasks

How to: Add Command Routing to the Windows Forms Control

CommandUIHandler Delegate

Registers callback methods with a user interface update command message.

```
delegate void CommandUIHandler(
   unsigned int cmdID,
   ICommandUI^ cmdUI
);
```

Parameters

cmdID

The command ID.

cmdUI

The command message ID.

Remarks

This delegate registers callback methods with a user interface update command message. **CommandUIHandler** is similar to CommandHandler except that this delegate is used with user interface object update commands. User interface update commands should be mapped one-to-one with message handler methods.

For more information on using Windows Forms, see Using a Windows Form User Control in MFC.

Requirements

Header: afxwinforms.h (defined in assembly atlmfc\lib\mfcmifc80.dll)

See Also

Tasks

How to: Add Command Routing to the Windows Forms Control

Reference

CommandHandler Delegate

CompareElements

Called directly by CList::Find and indirectly by CMap::Lookup and CMap::operator [].

```
template<class TYPE, class ARG_TYPE>
BOOL AFXAPI CompareElements(
   const TYPE* pElement1,
   const ARG_TYPE* pElement2
);
```

Parameters

TYPE

The type of the first element to be compared.

pElement1

Pointer to the first element to be compared.

ARG_TYPE

The type of the second element to be compared.

pElement2

Pointer to the second element to be compared.

Return Value

Nonzero if the object pointed to by *pElement1* is equal to the object pointed to by *pElement2*; otherwise 0.

Remarks

The **CMap** calls use the **CMap** template parameters *KEY* and *ARG_KEY*.

The default implementation returns the result of the comparison of **pElement1* and **pElement2*. Override this function so that it compares the elements in a way that is appropriate for your application.

The C++ language defines the comparison operator (**==**) for simple types (**char**, **int**, **float**, and so on) but does not define a comparison operator for classes and structures. If you want to use **CompareElements** or to instantiate one of the collection classes that uses it, you must either define the comparison operator or overload **CompareElements** with a version that returns appropriate values.

See Also

Reference

CList Class

CMap Class

Concepts

MFC Macros and Globals

CopyElements

This function is called directly by CArray::Append and CArray::Copy.

```
template<class TYPE>
void AFXAPI CopyElements(
      TYPE* pDest,
   const TYPE* pSrc,
   INT_PTR nCount
);
```

Parameters

TYPE

Template parameter specifying the type of elements to be copied.

pDest

Pointer to the destination where the elements will be copied.

pSrc

Pointer to the source of the elements to be copied.

nCount

Number of elements to be copied.

Remarks

The default implementation uses the simple assignment operator (**=**) to perform the copy operation. If the type being copied does not have an overloaded operator=, then the default implementation performs a bitwise copy.

For information on implementing this and other helper functions, see the article Collections: How to Make a Type-Safe Collection.

See Also

Reference

CArray Class

Concepts

MFC Macros and Globals

CONNECTION_IID

Use between the **BEGIN_CONNECTION_PART** and **END_CONNECTION_PART** macros to define an interface ID for a connection point supported by your OLE control.

```
CONNECTION_IID(iid )
```

Parameters

iid

The interface ID of the interface called by the connection point.

Remarks

The *iid* argument is an interface ID used to identify the interface that the connection point will call on its connected sinks. For example:

```
CONNECTION_IID( IID_IsinkInterface )
```

specifies a connection point that calls the `ISinkInterface` interface.

See Also

Reference

BEGIN_CONNECTION_PART
DECLARE_CONNECTION_MAP
END_CONNECTION_PART

Concepts

MFC Macros and Globals

CONNECTION_PART

Maps a connection point for your OLE control to a specific interface ID.

```
CONNECTION_PART(theClass, iid, localClass )
```

Parameters

theClass

Specifies the name of the control class whose connection point this is.

iid

The interface ID of the interface called by the connection point.

localClass

Specifies the name of the local class that implements the connection point.

Remarks

For example:

```
BEGIN_CONNECTION_MAP(CSampleCtrl, COleControl)
    CONNECTION_PART(CSampleCtrl, IID_ISinkInterface, MyConnPt)
END_CONNECTION_MAP()
```

implements a connection map, with a connection point, that calls the `IID_ISinkInterface` interface .

See Also

Reference

BEGIN_CONNECTION_PART
DECLARE_CONNECTION_MAP
BEGIN_CONNECTION_MAP
CONNECTION_IID

Concepts

MFC Macros and Globals

DDP_CBIndex

Call this function in your property page's `DoDataExchange` function to synchronize the value of an integer property with the index of the current selection in a combo box on the property page.

```
void AFXAPI DDP_CBIndex(
   CDataExchange* pDX,
   int id,
   int& member,
   LPCTSTR pszPropName
);
```

Parameters

pDX

Pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

id

The resource ID of the combo box control associated with the control property specified by *pszPropName*.

member

The member variable associated with the property page control specified by *id* and the property specified by *pszPropName*.

pszPropName

The property name of the control property to be exchanged with the combo box control specified by *id*.

Remarks

This function should be called before the corresponding **DDX_CBIndex** function call.

See Also

Reference

DDP_CBString
DDP_Text
COleControl::DoPropExchange
DDX_CBIndex

Concepts

MFC Macros and Globals

DDP_CBString

Call this function in your property page's `DoDataExchange` function to synchronize the value of a string property with the current selection in a combo box on the property page.

```
void AFXAPI DDP_CBString(
   CDataExchange* pDX,
   int id,
   CString& member,
   LPCTSTR pszPropName
);
```

Parameters

pDX

Pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

id

The resource ID of the combo box control associated with the control property specified by *pszPropName*.

member

The member variable associated with the property page control specified by *id* and the property specified by *pszPropName*.

pszPropName

The property name of the control property to be exchanged with the combo box string specified by *id*.

Remarks

This function should be called before the corresponding **DDX_CBString** function call.

See Also

Reference

DDP_CBStringExact
DDP_CBIndex
COleControl::DoPropExchange
DDX_CBString

Concepts

MFC Macros and Globals

DDP_CBStringExact

Call this function in your property page's `DoDataExchange` function to synchronize the value of a string property that exactly matches the current selection in a combo box on the property page.

```
void AFXAPI DDP_CBStringExact(
   CDataExchange* pDX,
   int id,
   CString& member,
   LPCTSTR pszPropName
);
```

Parameters

pDX

Pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

id

The resource ID of the combo box control associated with the control property specified by *pszPropName*.

member

The member variable associated with the property page control specified by *id* and the property specified by *pszPropName*.

pszPropName

The property name of the control property to be exchanged with the combo box string specified by *id*.

Remarks

This function should be called before the corresponding **DDX_CBStringExact** function call.

See Also

Reference

DDP_CBString
DDP_CBIndex
COleControl::DoPropExchange
DDX_CBStringExact

Concepts

MFC Macros and Globals

DDP_Check

Call this function in your property page's `DoDataExchange` function to synchronize the value of the property with the associated property page check box control.

```
void AFXAPI DDP_Check(
   CDataExchange*pDX,
   int id,
   int &member,
   LPCSTR pszPropName
);
```

Parameters

pDX

Pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

id

The resource ID of the check box control associated with the control property specified by *pszPropName*.

member

The member variable associated with the property page control specified by *id* and the property specified by *pszPropName*.

pszPropName

The property name of the control property to be exchanged with the check box control specified by *id*.

Remarks

This function should be called before the corresponding **DDX_Check** function call.

See Also

Reference

DDP_Radio
DDP_Text
COleControl::DoPropExchange
DDX_Check

Concepts

MFC Macros and Globals

DDP_LBIndex

Call this function in your property page's `DoDataExchange` function to synchronize the value of an integer property with the index of the current selection in a list box on the property page.

```
void AFXAPI DDP_LBIndex(
   CDataExchange* pDX,
   int id,
   int& member,
   LPCTSTR pszPropName
);
```

Parameters

pDX

Pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

id

The resource ID of the list box control associated with the control property specified by *pszPropName*.

member

The member variable associated with the property page control specified by *id* and the property specified by *pszPropName*.

pszPropName

The property name of the control property to be exchanged with the list box string specified by *id*.

Remarks

This function should be called before the corresponding **DDX_LBIndex** function call.

See Also

Reference

DDP_LBString
DDP_CBIndex
COleControl::DoPropExchange
DDX_LBIndex

Concepts

MFC Macros and Globals

DDP_LBString

Call this function in your property page's `DoDataExchange` function to synchronize the value of a string property with the current selection in a list box on the property page.

```
void AFXAPI DDP_LBString(
   CDataExchange* pDX,
   int id,
   CString& member,
   LPCTSTR pszPropName
);
```

Parameters

pDX

Pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

id

The resource ID of the list box control associated with the control property specified by *pszPropName*.

member

The member variable associated with the property page control specified by *id* and the property specified by *pszPropName*.

pszPropName

The property name of the control property to be exchanged with the list box string specified by *id*.

Remarks

This function should be called before the corresponding **DDX_LBString** function call.

See Also

Reference

DDP_LBStringExact
DDP_LBIndex
COleControl::DoPropExchange
DDX_LBString

Concepts

MFC Macros and Globals

DDP_LBStringExact

Call this function in your property page's `DoDataExchange` function to synchronize the value of a string property that exactly matches the current selection in a list box on the property page.

```
void AFXAPI DDP_LBStringExact(
   CDataExchange* pDX,
   int id,
   CString& member,
   LPCTSTR pszPropName
);
```

Parameters

pDX

Pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

id

The resource ID of the list box control associated with the control property specified by *pszPropName*.

member

The member variable associated with the property page control specified by *id* and the property specified by *pszPropName*.

pszPropName

The property name of the control property to be exchanged with the list box string specified by *id*.

Remarks

This function should be called before the corresponding **DDX_LBStringExact** function call.

See Also

Reference

DDP_LBString
DDP_LBIndex
COleControl::DoPropExchange
DDX_LBStringExact

Concepts

MFC Macros and Globals

DDP_PostProcessing

Call this function in your property page's **DoDataExchange** function, to finish the transfer of property values from the property page to your control when property values are being saved.

```
void AFXAPI DDP_PostProcessing(
   CDataExchange *pDX
);
```

Parameters

pDX

Pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

Remarks

This function should be called after all data exchange functions are completed. For example:

```
void CSamplePage::DoDataExchange(CDataExchange* pDX)
{
   DDP_Text(pDX, IDC_POSITIONEDIT, m_NeedlePosition,
          _T("NeedlePosition"));
   DDX_Text(pDX, IDC_POSITIONEDIT, m_NeedlePosition);
   DDV_MinMaxInt(pDX, m_NeedlePosition, 0, 3);
   DDP_PostProcessing(pDX);
}
```

See Also

Reference

COleControl::DoPropExchange

Concepts

MFC Macros and Globals

DDP_Radio

Call this function in your control's `DoPropExchange` function to synchronize the value of the property with the associated property page radio button control.

```
void AFXAPI DDP_Radio(
   CDataExchange*pDX,
   int id,
   int &member,
   LPCTSTR pszPropName
);
```

Parameters

pDX

Pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

id

The resource ID of the radio button control associated with the control property specified by *pszPropName*.

member

The member variable associated with the property page control specified by *id* and the property specified by *pszPropName*.

pszPropName

The property name of the control property to be exchanged with the radio button control specified by *id*.

Remarks

This function should be called before the corresponding **DDX_Radio** function call.

See Also

Reference

DDP_Check
DDP_Text
COleControl::DoPropExchange
DDX_Radio

Concepts

MFC Macros and Globals

DDP_Text

Call this function in your control's `DoDataExchange` function to synchronize the value of the property with the associated property page control.

```
void AFXAPI DDP_Text(
   CDataExchange*pDX,
   int id,
   BYTE &member,
   LPCTSTR pszPropName
);
void AFXAPI DDP_Text(
   CDataExchange*pDX,
   int id,
   int &member,
   LPCTSTR pszPropName
);
void AFXAPI DDP_Text(
   CDataExchange*pDX,
   int id,
   UINT &member,
   LPCTSTR pszPropName
);
void AFXAPI DDP_Text(
   CDataExchange*pDX,
   int id,
   long &member,
   LPCTSTR pszPropName
);
void AFXAPI DDP_Text(
   CDataExchange*pDX,
   int id,
   DWORD &member,
   LPCTSTR pszPropName
);
void AFXAPI DDP_Text(
   CDataExchange*pDX,
   int id,
   float &member,
   LPCTSTR pszPropName
);
void AFXAPI DDP_Text(
   CDataExchange*pDX,
   int id,
   double &member,
   LPCTSTR pszPropName
);
void AFXAPI DDP_Text(
   CDataExchange*pDX,
   int id,
   CString &member,
   LPCTSTR pszPropName
);
```

Parameters

pDX

Pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

id

The resource ID of the control associated with the control property specified by *pszPropName*.

member

The member variable associated with the property page control specified by *id* and the property specified by *pszPropName*.

pszPropName

The property name of the control property to be exchanged with the control specified by *id*.

Remarks

This function should be called before the corresponding **DDX_Text** function call.

See Also

Reference

DDP_Check
DDP_Radio
COleControl::DoPropExchange
DDX_Text

Concepts

MFC Macros and Globals

DDV_MaxChars

Call **DDV_MaxChars** to verify that the amount of characters in the control associated with *value* does not exceed *nChars*.

```
void AFXAPI DDV_MaxChars(
   CDataExchange* pDX,
   CString const& value,
   int nChars
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

value

A reference to a member variable of the dialog box, form view, or control view object with which data is validated.

nChars

Maximum number of characters allowed.

Remarks

For more information about DDV, see Dialog Data Exchange and Validation.

See Also

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Validation Routines

DDV_MinMaxByte

Call **DDV_MinMaxByte** to verify that the value in the control associated with *value* falls between *minVal* and *maxVal*.

```
void AFXAPI DDV_MinMaxByte(
   CDataExchange* pDX,
   BYTE value,
   BYTE minVal,
   BYTE maxVal
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

value

A reference to a member variable of the dialog box, form view, or control view object with which data is validated.

minVal

Minimum value (of type **BYTE**) allowed.

maxVal

Maximum value (of type **BYTE**) allowed.

Remarks

For more information about DDV, see Dialog Data Exchange and Validation.

See Also

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Validation Routines

DDV_MinMaxDateTime

Call **DDV_MinMaxDateTime** to verify that the time/date value in the date and time picker control (CDateTimeCtrl) associated with *refValue* falls between *refMinRange* and *refMaxRange*.

```
void AFXAPI DDV_MinMaxDateTime(
   CDataExchange* pDX,
   CTime& refValue,
   const CTime* refMinRange,
   const CTime* refMaxRange
);
void AFXAPI DDV_MinMaxDateTime(
   CDataExchange* pDX,
   COleDateTime& refValue,
   const COleDateTime* refMinRange,
   const COleDateTime* refMaxRange
);
```

Parameters

pDX

A pointer to a CDataExchange object. The framework supplies this object to establish the context of the data exchange, including its direction. You don't need to delete this object.

refValue

A reference to a CTime or COleDateTime object associated with a member variable of the dialog box, form view, or control view object. This object contains the data to be validated.

refMinRange

Minimum date/time value allowed.

refMaxRange

Maximum date/time value allowed.

Remarks

For more information about DDV, see Dialog Data Exchange and Validation.

See Also

Reference

DDX_DateTimeCtrl

DDV_MinMaxMonth

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Validation Routines

DDV_MinMaxDouble

Call **DDV_MinMaxDouble** to verify that the value in the control associated with *value* falls between *minVal* and *maxVal*.

```
void AFXAPI DDV_MinMaxDouble(
   CDataExchange* pDX,
   double const& value,
   double minVal,
   double maxVal
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

value

A reference to a member variable of the dialog box, form view, or control view object with which data is validated.

minVal

Minimum value (of type **double**) allowed.

maxVal

Maximum value (of type **double**) allowed.

Remarks

For more information about DDV, see Dialog Data Exchange and Validation.

See Also

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Validation Routines

DDV_MinMaxDWord

Call **DDV_MinMaxDWord** to verify that the value in the control associated with *value* falls between *minVal* and *maxVal*.

```
void AFXAPI DDV_MinMaxDWord(
   CDataExchange* pDX,
   DWORD const& value,
   DWORD minVal,
   DWORD maxVal
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

value

A reference to a member variable of the dialog box, form view, or control view object with which data is validated.

minVal

Minimum value (of type **DWORD**) allowed.

maxVal

Maximum value (of type **DWORD**) allowed.

Remarks

For more information about DDV, see Dialog Data Exchange and Validation.

See Also

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Validation Routines

DDV_MinMaxFloat

Call **DDV_MinMaxFloat** to verify that the value in the control associated with *value* falls between *minVal* and *maxVal*.

```
void AFXAPI DDV_MinMaxFloat(
   CDataExchange* pDX,
   float value,
   float minVal,
   float maxVal
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

value

A reference to a member variable of the dialog box, form view, or control view object with which data is validated.

minVal

Minimum value (of type **float**) allowed.

maxVal

Maximum value (of type **float**) allowed.

Remarks

For more information about DDV, see Dialog Data Exchange and Validation.

See Also

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Validation Routines

DDV_MinMaxInt

Call **DDV_MinMaxInt** to verify that the value in the control associated with *value* falls between *minVal* and *maxVal*.

```
void AFXAPI DDV_MinMaxInt(
   CDataExchange* pDX,
   int value,
   int minVal,
   int maxVal
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

value

A reference to a member variable of the dialog box, form view, or control view object with which data is validated.

minVal

Minimum value (of type **int**) allowed.

maxVal

Maximum value (of type **int**) allowed.

Remarks

For more information about DDV, see Dialog Data Exchange and Validation.

See Also

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Validation Routines

DDV_MinMaxLong

Call **DDV_MinMaxLong** to verify that the value in the control associated with *value* falls between *minVal* and *maxVal*.

```
void AFXAPI DDV_MinMaxLong(
   CDataExchange* pDX,
   long value,
   long minVal,
   long maxVal
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

value

A reference to a member variable of the dialog box, form view, or control view object with which data is validated.

minVal

Minimum value (of type **long**) allowed.

maxVal

Maximum value (of type **long**) allowed.

Remarks

For more information about DDV, see Dialog Data Exchange and Validation.

See Also

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Validation Routines

DDV_MinMaxLongLong

Call **DDV_MinMaxLongLong** to verify that the value in the control associated with *value* falls between *minVal* and *maxVal*.

```
void AFXAPI DDV_MinMaxLongLong(
   CDataExchange* pDX,
   LONGLONG value,
   LONGLONG minVal,
   LONGLONG maxVal
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

value

A reference to a member variable of the dialog box, form view, or control view object with which data is validated.

minVal

Minimum value (of type **LONGLONG**) allowed.

maxVal

Maximum value (of type **LONGLONG**) allowed.

Remarks

For more information about DDV, see Dialog Data Exchange and Validation.

See Also

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Validation Routines

DDV_MinMaxMonth

Call **DDV_MinMaxMonth** to verify that the time/date value in the month calendar control (CMonthCalCtrl) associated with *refValue* falls between *refMinRange* and *refMaxRange*.

```
void AFXAPI DDV_MinMaxMonth(
   CDataExchange* pDX,
   CTime& refValue,
   const CTime* refMinRange,
   const CTime* refMaxRange
);
void AFXAPI DDV_MinMaxMonth(
   CDataExchange* pDX,
   COleDateTime& refValue,
   const COleDateTime* refMinRange,
   const COleDateTime* refMaxRange
);
```

Parameters

pDX

A pointer to a CDataExchange object. The framework supplies this object to establish the context of the data exchange, including its direction.

refValue

A reference to an object of type **CTime** or **COleDateTime** associated with a member variable of the dialog box, form view, or control view object. This object contains the data to be validated. MFC passes this reference when **DDV_MinMaxMonth** is called.

refMinRange

Minimum date/time value allowed.

refMaxRange

Maximum date/time value allowed.

Remarks

For more information about DDV, see Dialog Data Exchange and Validation.

See Also

Reference

DDX_DateTimeCtrl

DDV_MinMaxMonth

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Validation Routines

DDV_MinMaxShort

Call **DDV_MinMaxShort** to verify that the value in the control associated with *value* falls between *minVal* and *maxVal*.

```
void AFXAPI DDV_MinMaxShort(
   CDataExchange* pDX,
   short value,
   short minVal,
   short maxVal
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

value

A reference to a member variable of the dialog box, form view, or control view object with which data is validated.

minVal

Minimum value (of type **short**) allowed.

maxVal

Maximum value (of type **short**) allowed.

Remarks

For more information about DDV, see Dialog Data Exchange and Validation.

See Also

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Validation Routines

DDV_MinMaxSlider

Call **DDV_MinMaxSlider** to verify that the value in the control associated with *value* falls between *minVal* and *maxVal*.

```
void AFXAPI DDV_MinMaxSlider(
   CDataExchange* pDX,
   DWORD value,
   DWORD minVal,
   DWORD maxVal
);
```

Parameters

pDX

A pointer to a CDataExchange object. The framework supplies this object to establish the context of the data exchange, including its direction.

value

A reference to the value to be validated. This parameter holds or sets the slider control's current thumb position.

minVal

Minimum value allowed.

maxVal

Maximum value allowed.

Remarks

For more information about DDV, see Dialog Data Exchange and Validation. For information about slider controls, see Using CSliderCtrl.

See Also

Reference

DDX_Slider

DDX_FieldSlider

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Validation Routines

DDV_MinMaxUInt

Call **DDV_MinMaxUInt** to verify that the value in the control associated with *value* falls between *minVal* and *maxVal*.

```
void AFXAPI DDV_MinMaxUInt(
   CDataExchange* pDX,
   UINT value,
   UINT minVal,
   UINT maxVal
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

value

A reference to a member variable of the dialog box, form view, or control view object with which data is validated.

minVal

Minimum value (of type **UINT**) allowed.

maxVal

Maximum value (of type **UINT**) allowed.

Remarks

For more information about DDV, see Dialog Data Exchange and Validation.

See Also

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Validation Routines

DDV_MinMaxULongLong

Call **DDV_MinMaxULongLong** to verify that the value in the control associated with *value* falls between *minVal* and *maxVal*.

```
void AFXAPI DDV_MinMaxULongLong(
   CDataExchange* pDX,
   ULONGLONG value,
   ULONGLONG minVal,
   ULONGLONG maxVal
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

value

A reference to a member variable of the dialog box, form view, or control view object with which data is validated.

minVal

Minimum value (of type **ULONGLONG**) allowed.

maxVal

Maximum value (of type **ULONGLONG**) allowed.

Remarks

For more information about DDV, see Dialog Data Exchange and Validation.

See Also

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Validation Routines

DDV_MinMaxUnsigned

Call **DDV_MinMaxUnsigned** to verify that the value in the control associated with *value* falls between *minVal* and *maxVal*.

```
void AFXAPI DDV_MinMaxUnsigned(
   CDataExchange* pDX,
   unsigned value,
   unsigned minVal,
   unsigned maxVal
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

value

A reference to a member variable of the dialog box, form view, or control view object with which data is validated.

minVal

Minimum value (of type **unsigned**) allowed.

maxVal

Maximum value (of type **unsigned**) allowed.

Remarks

For more information about DDV, see Dialog Data Exchange and Validation.

See Also

Reference

DDX_Slider

DDX_FieldSlider

Concepts

MFC Macros and Globals

DDX_CBIndex

The **DDX_CBIndex** function manages the transfer of **int** data between a combo box control in a dialog box, form view, or control view object and a **int** data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_CBIndex(
   CDataExchange* pDX,
   int nIDC,
   int& index
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The resource ID of the combo box control associated with the control property.

index

A reference to a member variable of the dialog box, form view, or control view object with which data is exchanged.

Remarks

When **DDX_CBIndex** is called, *index* is set to the index of the current combo box selection. If no item is selected, *index* is set to 0.

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDP_CBIndex

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Exchange Routines

DDX_CBString

The **DDX_CBString** function manages the transfer of **CString** data between the edit control of a combo box control in a dialog box, form view, or control view object and a **CString** data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_CBString(
   CDataExchange* pDX,
   int nIDC,
   CString& value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The resource ID of the combo box control associated with the control property.

value

A reference to a member variable of the dialog box, form view, or control view object with which data is exchanged.

Remarks

When **DDX_CBString** is called, *value* is set to the current combo box selection. If no item is selected, *value* is set to a string of zero length.

Note
If the combo box is a drop-down list box, the value exchanged is limited to 255 characters.

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDP_CBString

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Exchange Routines

DDX_CBStringExact

The **DDX_CBStringExact** function manages the transfer of **CString** data between the edit control of a combo box control in a dialog box, form view, or control view object and a **CString** data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_CBStringExact(
   CDataExchange* pDX,
   int nIDC,
   CString& value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The resource ID of the combo box control associated with the control property.

value

A reference to a member variable of the dialog box, form view, or control view object with which data is exchanged.

Remarks

When **DDX_CBStringExact** is called, *value* is set to the current combo box selection. If no item is selected, *value* is set to a string of zero length.

Note
If the combo box is a drop-down list box, the value exchanged is limited to 255 characters.

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDP_CBStringExact

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Exchange Routines

DDX_Check

The **DDX_Check** function manages the transfer of **int** data between a check box control in a dialog box, form view, or control view object and a **int** data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_Check(
   CDataExchange* pDX,
   int nIDC,
   int& value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The resource ID of the check box control associated with the control property.

value

A reference to a member variable of the dialog box, form view, or control view object with which data is exchanged.

Remarks

When **DDX_Check** is called, *value* is set to the current state of the check box control. For a list of the possible state values, see BM_GETCHECK in the *Platform SDK*.

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDP_Check

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Exchange Routines

DDX_Control

The **DDX_Control** function subclasses the control, specified by *nIDC*, of the dialog box, form view, or control view object.

```
void AFXAPI DDX_Control(
   CDataExchange* pDX,
   int nIDC,
   CWnd& rControl
);
```

Parameters

pDX

A pointer to a CDataExchange object.

nIDC

The resource ID of the control to be subclassed.

rControl

A reference to a member variable of the dialog box, form view, or control view object related to the specified control.

Remarks

The *pDX* object is supplied by the framework when the **DoDataExchange** function is called. Therefore, **DDX_Control** should only be called within your override of **DoDataExchange**.

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Exchange Routines

DDX_DateTimeCtrl

The **DDX_DateTimeCtrl** function manages the transfer of date and/or time data between a date and time picker control (CDateTimeCtrl) in a dialog box or form view object and either a CTime or a COleDateTime data member of the dialog box or form view object.

```
void AFXAPI DDX_DateTimeCtrl(
   CDataExchange* pDX,
   int nIDC,
   CTime& value
);
void AFXAPI DDX_DateTimeCtrl(
   CDataExchange* pDX,
   int nIDC,
   COleDateTime& value
);
void AFXAPI DDX_DateTimeCtrl(
   CDataExchange* pDX,
   int nIDC,
   CString& value
);
```

Parameters

pDX

A pointer to a CDataExchange object. The framework supplies this object to establish the context of the data exchange, including its direction. You don't need to delete this object.

nIDC

The resource ID of the date and time picker control associated with the member variable.

value

In the first two versions, a reference to a **CTime** or **COleDateTime** member variable, dialog box, form view, or control view object with which data is exchanged. In the third version, a reference to a **CString** data member control view object.

Remarks

When **DDX_DateTimeCtrl** is called, *value* is set to the current state of the date and time picker control, or the control is set to *value*, depending on the direction of the exchange.

In the third version above, **DDX_DateTimeCtrl** manages the transfer of **CString** data between a date time control and a CString data member of the control view object. The string is formatted using the current locale's rules for formatting dates and times.

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

CDateTimeCtrl Class
CDateTimeCtrl::SetRange
CDateTimeCtrl::GetRange
DDV_MinMaxDateTime

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Exchange Routines

DDX_FieldCBIndex

The **DDX_FieldCBIndex** function synchronizes the index of the selected item in the list box control of a combo box control in a record view and an **int** field data member of a recordset associated with the record view.

```
void AFXAPI DDX_FieldCBIndex(
   CDataExchange* pDX,
   int nIDC,
   int& index,
   CRecordset* pRecordset
);
void AFXAPI DDX_FieldCBIndex(
   CDataExchange* pDX,
   int nIDC,
   int& index,
   CDaoRecordset* pRecordset
);
```

Parameters

pDX

A pointer to a CDataExchange object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of a control in the CRecordView or CDaoRecordView object.

index

A reference to a field data member in the associated **CRecordset** or **CDaoRecordset** object.

pRecordset

A pointer to the CRecordset or CDaoRecordset object with which data is exchanged.

Remarks

When moving data from the recordset to the control, this function sets the selection in the control based on the value specified in *index*. On a transfer from the recordset to the control, if the recordset field is Null, MFC sets the value of the index to 0. On a transfer from control to recordset, if the control is empty or if no item is selected, the recordset field is set to 0.

Use the first version if you are working with the ODBC-based classes. Use the second version if you are working with the DAO-based classes.

For more information about DDX, see Dialog Data Exchange and Validation. For examples and more information about DDX for CRecordView and CDaoRecordView fields, see the article Record Views.

Example

See DDX_FieldText for a general DDX_Field example. The example would be similar for **DDX_FieldCBIndex**.

See Also

Reference

DDX_FieldText
DDX_FieldRadio
DDX_FieldLBString
DDX_FieldLBStringExact
DDX_FieldCBStringExact
DDX_FieldLBIndex
DDX_FieldScroll
DDX_CBIndex

Concepts

MFC Macros and Globals

DDX_FieldCBString

The **DDX_FieldCBString** function manages the transfer of CString data between the edit control of a combo box control in a record view and a **CString** field data member of a recordset associated with the record view.

```
void AFXAPI DDX_FieldCBString(
   CDataExchange* pDX,
   int nIDC,
   CString& value,
   CRecordset* pRecordset
);
void AFXAPI DDX_FieldCBString(
   CDataExchange* pDX,
   int nIDC,
   CString& value,
   CDaoRecordset* pRecordset
);
```

Parameters

pDX

A pointer to a CDataExchange object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of a control in the CRecordView or CDaoRecordView object.

value

A reference to a field data member in the associated **CRecordset** or **CDaoRecordset** object.

pRecordset

A pointer to the CRecordset or CDaoRecordset object with which data is exchanged.

Remarks

When moving data from the recordset to the control, this function sets the current selection in the combo box to the first row that begins with the characters in the string specified in *value*. On a transfer from the recordset to the control, if the recordset field is Null, any selection is removed from the combo box and the edit control of the combo box is set to empty. On a transfer from control to recordset, if the control is empty, the recordset field is set to Null if the field permits.

Use the first version if you are working with the ODBC-based classes. Use the second version if you are working with the DAO-based classes.

For more information about DDX, see Dialog Data Exchange and Validation. For examples and more information about DDX for CRecordView and CDaoRecordView fields, see the article Record Views.

Example

See DDX_FieldText for a general DDX_Field example. The example includes a call to **DDX_FieldCBString**.

See Also

Reference

DDX_FieldText
DDX_FieldRadio
DDX_FieldLBString
DDX_FieldLBStringExact
DDX_FieldCBStringExact

Concepts

MFC Macros and Globals

DDX_FieldCBStringExact

The **DDX_FieldCBStringExact** function manages the transfer of CString data between the edit control of a combo box control in a record view and a **CString** field data member of a recordset associated with the record view.

```
void AFXAPI DDX_FieldCBStringExact(
   CDataExchange* pDX,
   int nIDC,
   CString& value,
   CRecordset* pRecordset
);
void AFXAPI DDX_FieldCBStringExact(
   CDataExchange* pDX,
   int nIDC,
   CString& value,
   CDaoRecordset* pRecordset
);
```

Parameters

pDX

A pointer to a CDataExchange object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of a control in the CRecordView or CDaoRecordView object.

value

A reference to a field data member in the associated **CRecordset** or **CDaoRecordset** object.

pRecordset

A pointer to the CRecordset or CDaoRecordset object with which data is exchanged.

Remarks

When moving data from the recordset to the control, this function sets the current selection in the combo box to the first row that exactly matches the string specified in *value*. On a transfer from the recordset to the control, if the recordset field is NULL, any selection is removed from the combo box and the edit box of the combo box is set to empty. On a transfer from control to recordset, if the control is empty, the recordset field is set to NULL.

Use the first version if you are working with the ODBC-based classes. Use the second version if you are working with the DAO-based classes.

For more information about DDX, see Dialog Data Exchange and Validation. For examples and more information about DDX for CRecordView and CDaoRecordView fields, see the article Record Views.

Example

See DDX_FieldText for a general DDX_Field example. Calls to **DDX_FieldCBStringExact** would be similar.

See Also

Reference

DDX_FieldText
DDX_FieldRadio
DDX_FieldLBString
DDX_FieldLBStringExact
DDX_FieldCBString

Concepts

MFC Macros and Globals

DDX_FieldCheck

The **DDX_FieldCheck** function manages the transfer of **int** data between a check box control in a dialog box, form view, or control view object and an **int** data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_FieldCheck(
   CDataExchange* pDX,
   int nIDC,
   int& value,
   CRecordset* pRecordset
);
void AFXAPI DDX_FieldCheck(
   CDataExchange* pDX,
   int nIDC,
   int& value,
   CDaoRecordset* pRecordset
);
```

Parameters

pDX

A pointer to a CDataExchange object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The resource ID of the check box control associated with the control property.

value

A reference to a member variable of the dialog box, form view, or control view object with which data is exchanged.

pRecordset

A pointer to the CRecordset or CDaoRecordset object with which data is exchanged.

Remarks

When **DDX_FieldCheck** is called, *value* is set to the current state of the check box control, or the control's state is set to *value*, depending on the direction of transfer.

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDX_FieldText
DDX_FieldRadio
DDX_FieldLBString
DDX_FieldLBStringExact
DDX_FieldCBString

Concepts

MFC Macros and Globals

DDX_FieldLBIndex

The **DDX_FieldLBIndex** function synchronizes the index of the selected item in a list box control in a record view and an **int** field data member of a recordset associated with the record view.

```
void AFXAPI DDX_FieldLBIndex(
   CDataExchange* pDX,
   int nIDC,
   int& index,
   CRecordset* pRecordset
);
void AFXAPI DDX_FieldLBIndex(
   CDataExchange* pDX,
   int nIDC,
   int& index,
   CDaoRecordset* pRecordset
);
```

Parameters

pDX

A pointer to a CDataExchange object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of a control in the CRecordView or CDaoRecordView object.

index

A reference to a field data member in the associated **CRecordset** or **CDaoRecordset** object.

pRecordset

A pointer to the CRecordset or CDaoRecordset object with which data is exchanged.

Remarks

When moving data from the recordset to the control, this function sets the selection in the control based on the value specified in *index*. On a transfer from the recordset to the control, if the recordset field is Null, MFC sets the value of the index to 0. On a transfer from control to recordset, if the control is empty, the recordset field is set to 0.

Use the first version if you are working with the ODBC-based classes. Use the second version if you are working with the DAO-based classes.

For more information about DDX, see Dialog Data Exchange and Validation. For examples and more information about DDX for CRecordView and CDaoRecordView fields, see the article Record Views.

Example

See DDX_FieldText for a general DDX_Field example.

See Also

Reference

DDX_FieldText
DDX_FieldRadio
DDX_FieldLBString
DDX_FieldLBStringExact
DDX_FieldCBStringExact
DDX_FieldCBIndex
DDX_FieldScroll
DDX_LBIndex

Concepts

MFC Macros and Globals

DDX_FieldLBString

The **DDX_FieldLBString** copies the current selection of a list box control in a record view to a CString field data member of a recordset associated with the record view.

```
void AFXAPI DDX_FieldLBString(
   CDataExchange* pDX,
   int nIDC,
   CString& value,
   CRecordset* pRecordset
);
void AFXAPI DDX_FieldLBString(
   CDataExchange* pDX,
   int nIDC,
   CString& value,
   CDaoRecordset* pRecordset
);
```

Parameters

pDX

A pointer to a CDataExchange object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of a control in the CRecordView or CDaoRecordView object.

value

A reference to a field data member in the associated **CRecordset** or **CDaoRecordset** object.

pRecordset

A pointer to the CRecordset or CDaoRecordset object with which data is exchanged.

Remarks

In the reverse direction, this function sets the current selection in the list box to the first row that begins with the characters in the string specified by *value*. On a transfer from the recordset to the control, if the recordset field is Null, any selection is removed from the list box. On a transfer from control to recordset, if the control is empty, the recordset field is set to Null.

Use the first version if you are working with the ODBC-based classes. Use the second version if you are working with the DAO-based classes.

For more information about DDX, see Dialog Data Exchange and Validation. For examples and more information about DDX for CRecordView and CDaoRecordView fields, see the article Record Views.

Example

See DDX_FieldText for a general DDX_Field example. Calls to **DDX_FieldLBString** would be similar.

See Also

Reference

DDX_FieldText
DDX_FieldRadio
DDX_FieldLBStringExact
DDX_FieldCBString
DDX_FieldCBStringExact
DDX_FieldCBIndex
DDX_FieldLBIndex
DDX_FieldScroll

Concepts

MFC Macros and Globals

DDX_FieldLBStringExact

The **DDX_FieldLBStringExact** function copies the current selection of a list box control in a record view to a CString field data member of a recordset associated with the record view.

```
void AFXAPI DDX_FieldLBStringExact(
   CDataExchange* pDX,
   int nIDC,
   CString& value,
   CRecordset* pRecordset
);
void AFXAPI DDX_FieldLBStringExact(
   CDataExchange* pDX,
   int nIDC,
   CString& value,
   CDaoRecordset* pRecordset
);
```

Parameters

pDX

A pointer to a CDataExchange object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of a control in the CRecordView or CDaoRecordView object.

value

A reference to a field data member in the associated **CRecordset** or **CDaoRecordset** object.

pRecordset

A pointer to the CRecordset or CDaoRecordset object with which data is exchanged.

Remarks

In the reverse direction, this function sets the current selection in the list box to the first row that exactly matches the string specified in *value*. On a transfer from the recordset to the control, if the recordset field is Null, any selection is removed from the list box. On a transfer from control to recordset, if the control is empty, the recordset field is set to Null.

Use the first version if you are working with the ODBC-based classes. Use the second version if you are working with the DAO-based classes.

For more information about DDX, see Dialog Data Exchange and Validation. For examples and more information about DDX for CRecordView and CDaoRecordView fields, see the article Record Views.

Example

See DDX_FieldText for a general DDX_Field example. Calls to **DDX_FieldLBStringExact** would be similar.

See Also

Reference

DDX_FieldText
DDX_FieldRadio
DDX_FieldLBString
DDX_FieldCBString
DDX_FieldCBStringExact
DDX_FieldCBIndex
DDX_FieldLBIndex
DDX_FieldScroll

Concepts

MFC Macros and Globals

DDX_FieldSlider

The **DDX_FieldSlider** function synchronizes the thumb position of a slider control in a record view and an **int** field data member of a recordset associated with the record view (or with whatever integer variable you choose to map it to).

```
void AFXAPI DDX_FieldSlider(
   CDataExchange* pDX,
      int nIDC,
      int& value,
      CRecordset* pRecordset
);
void AFXAPI DDX_FieldSlider(
   CDataExchange* pDX,
      int nIDC,
      int& value,
      CDaoRecordset* pRecordset
);
```

Parameters

pDX

A pointer to a CDataExchange object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The resource ID of the slider control.

value

A reference to the value to be exchanged. This parameter holds or will be used to set the slider control's current thumb position.

pRecordset

A pointer to the associated **CRecordset** or **CDaoRecordset** object with which data is exchanged.

Remarks

When moving data from the recordset to the slider, this function sets the position of the slider to the value specified in *value*. On a transfer from the recordset to the control, if the recordset field is Null, the slider control's position is set to 0. On a transfer from the control to the recordset, if the control is empty, the value of the recordset field is 0.

DDX_FieldSlider does not exchange range information with slider controls capable of setting a range rather than simply a position.

Use the first override of the function if you are working with the ODBC-based classes. Use the second override with the DAO-based classes.

For more information about DDX, see Dialog Data Exchange and Validation. For examples and more information about DDX for **CRecordView** and **CDaoRecordView** fields, see Record Views. For information about slider controls, see Using CSliderCtrl.

Example

See DDX_FieldText for a general DDX_Field example. Calls to **DDX_FieldSlider** would be similar.

See Also

Reference

DDX_Slider
DDV_MinMaxSlider
DDX_FieldLBString
DDX_FieldLBStringExact
DDX_FieldCBString
DDX_FieldCBStringExact
DDX_FieldCBIndex
DDX_FieldLBIndex

Concepts

MFC Macros and Globals

DDX_FieldRadio

The **DDX_FieldRadio** function associates a zero-based **int** member variable of a record view's recordset with the currently selected radio button in a group of radio buttons in the record view.

```
void AFXAPI DDX_FieldRadio(
   CDataExchange* pDX,
   int nIDC,
   int& value,
   CRecordset* pRecordset
);
void AFXAPI DDX_FieldRadio(
   CDataExchange* pDX,
   int nIDC,
   int& value,
   CDaoRecordset* pRecordset
);
```

Parameters

pDX

A pointer to a CDataExchange object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of the first in a group (with style **WS_GROUP**) of adjacent radio button controls in the CRecordView or CDaoRecordView object.

value

A reference to a field data member in the associated **CRecordset** or **CDaoRecordset** object.

pRecordset

A pointer to the CRecordset or CDaoRecordset object with which data is exchanged.

Remarks

When transferring from the recordset field to the view, this function turns on the *nth* radio button (zero-based) and turns off the other buttons. In the reverse direction, this function sets the recordset field to the ordinal number of the radio button that is currently on (checked). On a transfer from the recordset to the control, if the recordset field is Null, no button is selected. On a transfer from control to recordset, if no control is selected, the recordset field is set to Null if the field permits that.

Use the first version if you are working with the ODBC-based classes. Use the second version if you are working with the DAO-based classes.

For more information about DDX, see Dialog Data Exchange and Validation. For examples and more information about DDX for CRecordView and CDaoRecordView fields, see the article Record Views.

Example

See DDX_FieldText for a general DDX_Field example. Calls to **DDX_FieldRadio** would be similar.

See Also

Reference

DDX_FieldText
DDX_FieldLBString
DDX_FieldLBStringExact
DDX_FieldCBString
DDX_FieldCBStringExact
DDX_FieldCBIndex
DDX_FieldLBIndex
DDX_FieldScroll

Concepts

MFC Macros and Globals

DDX_FieldScroll

The **DDX_FieldScroll** function synchronizes the scroll position of a scroll bar control in a record view and an **int** field data member of a recordset associated with the record view (or with whatever integer variable you choose to map it to).

```
void AFXAPI DDX_FieldScroll(
   CDataExchange* pDX,
   int nIDC,
   int& value,
   CRecordset* pRecordset
);
void AFXAPI DDX_FieldScroll(
   CDataExchange* pDX,
   int nIDC,
   int& value,
   CDaoRecordset* pRecordset
);
```

Parameters

pDX

A pointer to a CDataExchange object. The framework supplies this object to establish the context of the data exchange, including its direction.

*nIDC**

The ID of the first in a group (with style **WS_GROUP**) of adjacent radio button controls in the CRecordView or CDaoRecordView object.

value

A reference to a field data member in the associated **CRecordset** or **CDaoRecordset** object.

pRecordset

A pointer to the CRecordset or CDaoRecordset object with which data is exchanged.

Remarks

When moving data from the recordset to the control, this function sets the scroll position of the scroll bar control to the value specified in *value*. On a transfer from the recordset to the control, if the recordset field is Null, the scroll bar control is set to 0. On a transfer from control to recordset, if the control is empty, the value of the recordset field is 0.

Use the first version if you are working with the ODBC-based classes. Use the second version if you are working with the DAO-based classes.

For more information about DDX, see Dialog Data Exchange and Validation. For examples and more information about DDX for CRecordView and CDaoRecordView fields, see the article Record Views.

Example

See DDX_FieldText for a general DDX_Field example. Calls to **DDX_FieldScroll** would be similar.

See Also

Reference

DDX_FieldText
DDX_FieldLBString
DDX_FieldLBStringExact
DDX_FieldCBString
DDX_FieldCBStringExact
DDX_FieldCBIndex
DDX_FieldLBIndex
DDX_Scroll

Concepts

MFC Macros and Globals

DDX_FieldText

The **DDX_FieldText** function manages the transfer of **int**, **short**, **long**, **DWORD**, CString, **float**, **double**, **BOOL**, or **BYTE** data between an edit box control and the field data members of a recordset.

```
void AFXAPI DDX_FieldText(
   CDataExchange* pDX,
   int nIDC,
   BYTE& value,
   CRecordset* pRecordset
);
void AFXAPI DDX_FieldText(
   CDataExchange* pDX,
   int nIDC,
   int& value,
   CRecordset* pRecordset
);
void AFXAPI DDX_FieldText(
   CDataExchange* pDX,
   int nIDC,
   UINT& value,
   CRecordset* pRecordset
);
void AFXAPI DDX_FieldText(
   CDataExchange* pDX,
   int nIDC,
   long& value,
   CRecordset* pRecordset
);
void AFXAPI DDX_FieldText(
   CDataExchange* pDX,
   int nIDC,
   DWORD& value,
   CRecordset* pRecordset
);
void AFXAPI DDX_FieldText(
   CDataExchange* pDX,
   int nIDC,
   CString& value,
   CRecordset* pRecordset
);
void AFXAPI DDX_FieldText(
   CDataExchange* pDX,
   int nIDC,
   float& value,
   CRecordset* pRecordset
);
void AFXAPI DDX_FieldText(
   CDataExchange* pDX,
   int nIDC,
   double& value,
   CRecordset* pRecordset
);
void AFXAPI DDX_FieldText(
   CDataExchange* pDX,
   int nIDC,
   short& value,
   CDaoRecordset* pRecordset
);
void AFXAPI DDX_FieldText(
   CDataExchange* pDX,
   int nIDC,
   BOOL& value,
   CDaoRecordset* pRecordset
```

```
);
void AFXAPI DDX_FieldText(
   CDataExchange* pDX,
   int nIDC,
   BYTE& value,
   CDaoRecordset* pRecordset
);
void AFXAPI DDX_FieldText(
   CDataExchange* pDX,
   int nIDC,
   long& value,
   CDaoRecordset* pRecordset
);
void AFXAPI DDX_FieldText(
   CDataExchange* pDX,
   int nIDC,
   DWORD& value,
   CDaoRecordset* pRecordset
);
void AFXAPI DDX_FieldText(
   CDataExchange* pDX,
   int nIDC,
   CString& value,
   CDaoRecordset* pRecordset
);
void AFXAPI DDX_FieldText(
   CDataExchange* pDX,
   int nIDC,
   float& value,
   CDaoRecordset* pRecordset
);
void AFXAPI DDX_FieldText(
   CDataExchange* pDX,
   int nIDC,
   double& value,
   CDaoRecordset* pRecordset
);
void AFXAPI DDX_FieldText(
   CDataExchange* pDX,
   int nIDC,
   COleDateTime& value,
   CDaoRecordset* pRecordset
);
void AFXAPI DDX_FieldText(
   CDataExchange* pDX,
   int nIDC,
   COleCurrency& value,
   CDaoRecordset* pRecordset
);
```

Parameters

pDX

A pointer to a CDataExchange object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of a control in the CRecordView or CDaoRecordView object.

value

A reference to a field data member in the associated **CRecordset** or **CDaoRecordset** object. The data type of value depends on which of the overloaded versions of **DDX_FieldText** you use.

pRecordset

A pointer to the CRecordset or CDaoRecordset object with which data is exchanged. This pointer enables **DDX_FieldText** to detect and set Null values.

Remarks

For CDaoRecordset objects, **DDX_FieldText** also manages transferring COleDateTime, and COleCurrency values. An empty edit box control indicates a Null value. On a transfer from the recordset to the control, if the recordset field is Null, the edit box is set to empty. On a transfer from control to recordset, if the control is empty, the recordset field is set to Null.

Use the versions with CRecordset parameters if you are working with the ODBC-based classes. Use the versions with CDaoRecordset parameters if you are working with the DAO-based classes.

For more information about DDX, see Dialog Data Exchange and Validation. For examples and more information about DDX for CRecordView and CDaoRecordView fields, see the article Record Views.

Example

The following **DoDataExchange** function for a CRecordView contains **DDX_FieldText** function calls for three data types: `IDC_COURSELIST` is a combo box; the other two controls are edit boxes. For DAO programming, the *m_pSet* parameter is a pointer to a CRecordset or CDaoRecordset.

```
//Example for DDX_FieldText
void CSectionForm::DoDataExchange( CDataExchange* pDX )
{
    CRecordView::DoDataExchange( pDX );
    DDX_FieldCBString( pDX, IDC_COURSELIST,
                    m_pSet->m_strCourseID, m_pSet);
    DDX_FieldText( pDX, IDC_ROOM, m_pSet->m_nRoomNo,
                    m_pSet );
    DDX_FieldText( pDX, IDC_TUITION,
                    m_pSet->m_dwTuition, m_pSet );
}
```

See Also

Reference

DDX_FieldRadio
DDX_FieldLBString
DDX_FieldLBStringExact
DDX_FieldCBString
DDX_FieldCBStringExact
DDX_FieldCBIndex
DDX_FieldLBIndex
DDX_FieldScroll

Concepts

MFC Macros and Globals

DDX_IPAddress

The **DDX_IPAddress** function manages the transfer of data between an IP Address control and a data member of the control view object.

```
void AFXAPI DDX_IPAddress(
   CDataExchange* pDX,
   int nIDC,
   DWORD& value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The resource ID of the IP Address control associated with the control property.

value

A reference to the **DWORD** containing the four-field value of the IP Address control. The fields are filled or read as follows.

Field	Bits containing the field value
3	0 through 7
2	8 through 15
1	16 through 23
0	24 through 31

Use the Win32 IPM_GETADDRESS to read the value, or use IPM_SETADDRESS to fill the value. These messages are described in the *Platform SDK*.

Remarks

When **DDX_IPAddress** is called, *value* is either read from the IP Address control, or *value* is written to the control, depending on the direction of the exchange.

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

CIPAddressCtrl Class

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Exchange Routines

DDX_LBIndex

The **DDX_LBIndex** function manages the transfer of **int** data between a list box control in a dialog box, form view, or control view object and an **int** data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_LBIndex(
   CDataExchange* pDX,
   int nIDC,
   int& index
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The resource ID of the list box control associated with the control property.

index

A reference to a member variable of the dialog box, form view, or control view object with which data is exchanged.

Remarks

When **DDX_LBIndex** is called, *index* is set to the index of the current list box selection. If no item is selected, *index* is set to -1.

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDP_LBIndex

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Exchange Routines

DDX_LBString

The **DDX_LBString** function manages the transfer of **CString** data between a list box control in a dialog box, form view, or control view object and a **CString** data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_LBString(
   CDataExchange* pDX,
   int nIDC,
   CString& value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The resource ID of the list box control associated with the control property.

value

A reference to a member variable of the dialog box, form view, or control view object with which data is exchanged.

Remarks

When **DDX_LBString** is called to transfer data to a list box control, the first item in the control whose beginning matches *value* is selected. (To match the entire item rather than just a prefix, use DDX_LBStringExact.) If there are no matches, no items are selected. The matching is case-insensitive.

When **DDX_LBString** is called to transfer data from a list box control, *value* is set to the current list box selection. If no item is selected, *value* is set to a string of zero length.

Note
If the list box is a drop-down list box, the value exchanged is limited to 255 characters.

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDP_LBString

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Exchange Routines

DDX_LBStringExact

The **DDX_CBStringExact** function manages the transfer of **CString** data between the edit control of a list box control in a dialog box, form view, or control view object and a **CString** data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_LBStringExact(
   CDataExchange* pDX,
   int nIDC,
   CString& value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The resource ID of the list box control associated with the control property.

value

A reference to a member variable of the dialog box, form view, or control view object with which data is exchanged.

Remarks

When **DDX_LBStringExact** is called to transfer data to a list box control, the first item in the control that matches *value* is selected. (To match just a prefix rather than the entire item, use DDX_LBString.) If there are no matches, no items are selected. The matching is case-insensitive.

When **DDX_CBStringExact** is called to transfer data from a list box control, *value* is set to the current list box selection. If no item is selected, *value* is set to a string of zero length.

Note
If the list box is a drop-down list box, the value exchanged is limited to 255 characters.

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDP_LBString

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Exchange Routines

DDX_ManagedControl

Creates a .NET control matching the control's resource ID.

```
template <typename T>
void DDX_ManagedControl(
   CDataExchange* pDX,
   int nIDC,
   CWinFormsControl<T>& control
);
```

Parameters

pDX

A pointer to a CDataExchange Class object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The resource ID of the control associated with the control property.

control

A reference to a CWinFormsControl Class object.

Remarks

DDX_ManagedControl calls CWinFormsControl::CreateManagedControl to create a control matching the resource control ID. Use **DDX_ManagedControl** to create controls from resource IDs in CDialog::OnInitDialog. For data exchange, you do not need to use the DDX/DDV functions with Windows Forms controls.

For more information, see How to: Do DDX/DDV Data Binding with Windows Forms.

Requirements

Header: afxwinforms.h

See Also

Reference

CWinFormsControl::CreateManagedControl
CDialog::OnInitDialog

DDX_MonthCalCtrl

The **DDX_MonthCalCtrl** function manages the transfer of date data between a month calendar control (CMonthCalCtrl) in a dialog box, form view, or control view object and either a CTime or a COleDateTime data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_MonthCalCtrl(
   CDataExchange* pDX,
   int nIDC,
   CTime& value
);
void AFXAPI DDX_MonthCalCtrl(
   CDataExchange* pDX,
   int nIDC,
   COleDateTime& value
);
```

Parameters

pDX

A pointer to a CDataExchange object. The framework supplies this object to establish the context of the data exchange, including its direction. You don't need to delete this object.

nIDC

The resource ID of the month calendar control associated with the member variable.

value

A reference to a **CTime** or **COleDateTime** member variable of the dialog box, form view, or control view object with which data is exchanged.

Remarks

Note
The control manages a date value only. The time fields in the time object are set to reflect the creation time of the control window, or whatever time was set in the control with a call to **CMonthCalCtrl::SetCurSel**.

When **DDX_MonthCalCtrl** is called, *value* is set to the current state of the month calendar control.

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDX_DateTimeCtrl
CMonthCalCtrl Class
CMonthCalCtrl::GetCurSel
CMonthCalCtrl::SetCurSel

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Exchange Routines

DDX_OCBool

The **DDX_OCBool** function manages the transfer of **BOOL** data between a property of an OLE control in a dialog box, form view, or control view object and a **BOOL** data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_OCBool(
   CDataExchange* pDX,
   int nIDC,
   DISPID dispid,
   BOOL& value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of an OLE control in the dialog box, form view, or control view object.

dispid

The dispatch ID of a property of the control.

value

A reference to a member variable of the dialog box, form view or control view object with which data is exchanged.

Remarks

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDX_OCBoolRO

Concepts

MFC Macros and Globals

DDX_OCBoolRO

The **DDX_OCBoolRO** function manages the transfer of **BOOL** data between a read-only property of an OLE control in a dialog box, form view, or control view object and a **BOOL** data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_OCBoolRO(
   CDataExchange* pDX,
   int nIDC,
   DISPID dispid,
   BOOL& value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of an OLE control in the dialog box, form view, or control view object.

dispid

The dispatch ID of a property of the control.

value

A reference to a member variable of the dialog box, form view or control view object with which data is exchanged.

Remarks

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDX_OCBool

Concepts

MFC Macros and Globals

DDX_OCColor

The **DDX_OCColor** function manages the transfer of **OLE_COLOR** data between a property of an OLE control in a dialog box, form view, or control view object and a **OLE_COLOR** data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_OCColor(
   CDataExchange* pDX,
   int nIDC,
   DISPID dispid,
   OLE_COLOR& value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of an OLE control in the dialog box, form view, or control view object.

dispid

The dispatch ID of a property of the control.

value

A reference to a member variable of the dialog box, form view or control view object with which data is exchanged.

Remarks

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDX_OCColorRO

Concepts

MFC Macros and Globals

DDX_OCColorRO

The **DDX_OCColorRO** function manages the transfer of **OLE_COLOR** data between a read-only property of an OLE control in a dialog box, form view, or control view object and a **OLE_COLOR** data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_OCColorRO(
   CDataExchange* pDX,
   int nIDC,
   DISPID dispid,
   OLE_COLOR & value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of an OLE control in the dialog box, form view, or control view object.

dispid

The dispatch ID of a property of the control.

value

A reference to a member variable of the dialog box, form view or control view object with which data is exchanged.

Remarks

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDX_OCColor

Concepts

MFC Macros and Globals

DDX_OCFloat

The **DDX_OCFloat** function manages the transfer of **float** (or **double**) data between a property of an OLE control in a dialog box, form view, or control view object and a **float** (or **double**) data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_OCFloat(
   CDataExchange* pDX,
   int nIDC,
   DISPID dispid,
   float& value
);
void AFXAPI DDX_OCFloat(
   CDataExchange* pDX,
   int nIDC,
   DISPID dispid,
   double& value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of an OLE control in the dialog box, form view, or control view object.

dispid

The dispatch ID of a property of the control.

value

A reference to a member variable of the dialog box, form view or control view object with which data is exchanged.

Remarks

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDX_OCFloatRO

Concepts

MFC Macros and Globals

DDX_OCFloatRO

The **DDX_OCFloatRO** function manages the transfer of **float** (or **double**) data between a read-only property of an OLE control in a dialog box, form view, or control view object and a **float** (or **double**) data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_OCFloatRO(
   CDataExchange* pDX,
   int nIDC,
   DISPID dispid,
   float& value
);
void AFXAPI DDX_OCFloatRO(
   CDataExchange* pDX,
   int nIDC,
   DISPID dispid,
   double& value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of an OLE control in the dialog box, form view, or control view object.

dispid

The dispatch ID of a property of the control.

value

A reference to a member variable of the dialog box, form view or control view object with which data is exchanged.

Remarks

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDX_OCFloat

Concepts

MFC Macros and Globals

DDX_OCInt

The **DDX_OCInt** function manages the transfer of **int** (or **long**) data between a property of an OLE control in a dialog box, form view, or control view object and a **int** (or **long**) data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_OCInt(
   CDataExchange* pDX,
   int nIDC,
   DISPID dispid,
   int& value
);
void AFXAPI DDX_OCInt(
   CDataExchange* pDX,
   int nIDC,
   DISPID dispid,
   long& value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of an OLE control in the dialog box, form view, or control view object.

dispid

The dispatch ID of a property of the control.

value

A reference to a member variable of the dialog box, form view or control view object with which data is exchanged.

Remarks

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDX_OCIntRO

Concepts

MFC Macros and Globals

DDX_OCIntRO

The **DDX_OCIntRO** function manages the transfer of **int** (or **long**) data between a read-only property of an OLE control in a dialog box, form view, or control view object and a **int** (or **long**) data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_OCIntRO(
   CDataExchange* pDX,
   int nIDC,
   DISPID dispid,
   int& value
);
void AFXAPI DDX_OCIntRO(
   CDataExchange* pDX,
   int nIDC,
   DISPID dispid,
   long& value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of an OLE control in the dialog box, form view, or control view object.

dispid

The dispatch ID of a property of the control.

value

A reference to a member variable of the dialog box, form view or control view object with which data is exchanged.

Remarks

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDX_OCInt

Concepts

MFC Macros and Globals

DDX_OCShort

The **DDX_OCShort** function manages the transfer of short data between a property of an OLE control in a dialog box, form view, or control view object and a short data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_OCShort(
   CDataExchange* pDX,
   int nIDC,
   DISPID dispid,
   short& value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of an OLE control in the dialog box, form view, or control view object.

dispid

The dispatch ID of a property of the control.

value

A reference to a member variable of the dialog box, form view or control view object with which data is exchanged.

Remarks

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDX_OCShortRO

Concepts

MFC Macros and Globals

DDX_OCShortRO

The **DDX_OCShortRO** function manages the transfer of short data between a read-only property of an OLE control in a dialog box, form view, or control view object and a short data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_OCShortRO(
   CDataExchange* pDX,
   int nIDC,
   DISPID dispid,
   short& value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of an OLE control in the dialog box, form view, or control view object.

dispid

The dispatch ID of a property of the control.

value

A reference to a member variable of the dialog box, form view or control view object with which data is exchanged.

Remarks

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDX_OCShort

Concepts

MFC Macros and Globals

DDX_OCText

The **DDX_OCText** function manages the transfer of **CString** data between a property of an OLE control in a dialog box, form view, or control view object and a **CString** data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_OCText(
   CDataExchange* pDX,
   int nIDC,
   DISPID dispid,
   CString& value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of an OLE control in the dialog box, form view, or control view object.

dispid

The dispatch ID of a property of the control.

value

A reference to a member variable of the dialog box, form view or control view object with which data is exchanged.

Remarks

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDX_OCTextRO

Concepts

MFC Macros and Globals

DDX_OCTextRO

The **DDX_OCTextRO** function manages the transfer of **CString** data between a read-only property of an OLE control in a dialog box, form view, or control view object and a **CString** data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_OCTextRO(
   CDataExchange* pDX,
   int nIDC,
   DISPID dispid,
   CString& value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of an OLE control in the dialog box, form view, or control view object.

dispid

The dispatch ID of a property of the control.

value

A reference to a member variable of the dialog box, form view or control view object with which data is exchanged.

Remarks

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDX_OCTextRO

Concepts

MFC Macros and Globals

DDX_Radio

The **DDX_Radio** function manages the transfer of **int** data between a radio control group in a dialog box, form view, or control view object and a **int** data member of the dialog box, form view, or control view object. The value of the **int** data member is determined according to which radio button within the group is selected.

```
void AFXAPI DDX_Radio(
   CDataExchange* pDX,
   int nIDC,
   int& value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The resource ID of the first radio control in the group.

value

A reference to a member variable of the dialog box, form view, or control view object with which data is exchanged.

Remarks

When **DDX_Radio** is called, *value* is set to the current state of the radio control group. The value is set as a 0-based index of the radio control that is currently checked, or -1 if no radio controls are checked.

For example, in case that the first radio button in the group is checked (the button with WS_GROUP style) the value of the **int** member is 0 and so on.

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDX_OCText

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Exchange Routines

DDX_Scroll

The **DDX_Scroll** function manages the transfer of **int** data between a scroll-bar control in a dialog box, form view, or control view object and an **int** data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_Scroll(
   CDataExchange* pDX,
   int nIDC,
   int& value
);
```

Parameters

pDX

A pointer to a **CDataExchange** object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The resource ID of the scroll-bar control associated with the control property.

value

A reference to a member variable of the dialog box, form view or control view object with which data is exchanged.

Remarks

When **DDX_Scroll** is called, *value* is set to the current position of the control's thumb. For more information on the values associated with the current position of the control's thumb, see GetScrollPos in the *Platform SDK*.

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Reference

DDX_OCText

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Exchange Routines

DDX_Slider

The **DDX_Slider** function manages the transfer of **int** data between a slider control in a dialog box or form view and an **int** data member of the dialog box or form view object.

```
void AFXAPI DDX_Slider(
   CDataExchange* pDX,
   int nIDC,
   int& value
);
```

Parameters

pDX

A pointer to a CDataExchange object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The resource ID of the slider control.

value

A reference to the value to be exchanged. This parameter holds or sets the slider control's current position.

Remarks

When **DDX_Slider** is called, *value* is set to the current position of the control's thumb, or the value receives the position, depending on the direction of the exchange.

For more information about DDX, see Dialog Data Exchange and Validation. For information about slider controls, see Using CSliderCtrl.

See Also

Reference

DDX_FieldSlider

DDV_MinMaxSlider

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Exchange Routines

DDX_Text

The **DDX_Text** function manages the transfer of **int**, **UINT**, **long**, **DWORD**, **CString**, **float**, or **double** data between an edit control in a dialog box, form view, or control view and a CString data member of the dialog box, form view, or control view object.

```
void AFXAPI DDX_Text(
   CDataExchange* pDX,
   int nIDC,
   BYTE& value
);
void AFXAPI DDX_Text(
   CDataExchange* pDX,
   int nIDC,
   short& value
);
void AFXAPI DDX_Text(
   CDataExchange* pDX,
   int nIDC,
   int& value
);
void AFXAPI DDX_Text(
   CDataExchange* pDX,
   int nIDC,
   UINT& value
);
void AFXAPI DDX_Text(
   CDataExchange* pDX,
   int nIDC,
   long& value
);
void AFXAPI DDX_Text(
   CDataExchange* pDX,
   int nIDC,
   DWORD& value
);
void AFXAPI DDX_Text(
   CDataExchange* pDX,
   int nIDC,
   CString& value
);
void AFXAPI DDX_Text(
   CDataExchange* pDX,
   int nIDC,
   float& value
);
void AFXAPI DDX_Text(
   CDataExchange* pDX,
   int nIDC,
   double& value
);
void AFXAPI DDX_Text(
   CDataExchange* pDX,
   int nIDC,
   COleCurrency& value
);
void AFXAPI DDX_Text(
   CDataExchange* pDX,
   int nIDC,
   COleDateTime& value
);
```

Parameters

pDX

A pointer to a CDataExchange object. The framework supplies this object to establish the context of the data exchange, including its direction.

nIDC

The ID of an edit control in the dialog box, form view, or control view object.

value

A reference to a data member in the dialog box, form view, or control view object. The data type of *value* depends on which of the overloaded versions of **DDX_Text** you use.

Remarks

For more information about DDX, see Dialog Data Exchange and Validation.

See Also

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Exchange Routines

DEBUG_NEW

Assists in finding memory leaks.

```
#define new DEBUG_NEW
```

Remarks

You can use **DEBUG_NEW** everywhere in your program that you would ordinarily use the **new** operator to allocate heap storage.

In debug mode (when the **_DEBUG** symbol is defined), **DEBUG_NEW** keeps track of the filename and line number for each object that it allocates. Then, when you use the CMemoryState::DumpAllObjectsSince member function, each object allocated with **DEBUG_NEW** is shown with the filename and line number where it was allocated.

To use **DEBUG_NEW**, insert the following directive into your source files:

```
#define new DEBUG_NEW
```

Once you insert this directive, the preprocessor will insert **DEBUG_NEW** wherever you use **new**, and MFC does the rest. When you compile a release version of your program, **DEBUG_NEW** resolves to a simple **new** operation, and the filename and line number information are not generated.

Note
In previous versions of MFC (4.1 and earlier) you needed to put the **#define** statement after all statements that called the **IMPLEMENT_DYNCREATE** or **IMPLEMENT_SERIAL** macros. This is no longer necessary.

See Also

Concepts

MFC Macros and Globals

Other Resources

MFC Debugging Techniques

DEBUG_ONLY

In debug mode (when the **_DEBUG** symbol is defined), **DEBUG_ONLY** evaluates its argument.

```
DEBUG_ONLY(expression )
```

Remarks

In a release build, **DEBUG_ONLY** does not evaluate its argument. This is useful when you have code that should be executed only in debug builds.

The **DEBUG_ONLY** macro is equivalent to surrounding *expression* with **#ifdef _DEBUG** and **#endif**.

Example

```
void example( char* p, int size, char fill )
{
char* q;                 // working copy of pointer
VERIFY( q = p );         // copy buffer pointer and validate
ASSERT( size >= 100 );   // make sure buffer is at least 100 bytes
ASSERT( isalpha(fill) ); // make sure fill character is alphabetic
// if fill character is invalid, substitute 'X' so we can continue
// debugging after the preceding ASSERT fails.
DEBUG_ONLY( if(!isalpha(fill)) fill = 'X' );
...
}
```

See Also

Reference

ASSERT (MFC)

VERIFY

Concepts

MFC Macros and Globals

DECLARE_CONNECTION_MAP

Each **COleControl**-derived class in your program can provide a connection map to specify additional connection points that your control supports.

```
DECLARE_CONNECTION_MAP( )
```

Remarks

If your control supports additional points, use the **DECLARE_CONNECTION_MAP** macro at the end of your class declaration. Then, in the .cpp file that defines the member functions for the class, use the **BEGIN_CONNECTION_MAP** macro, **CONNECTION_PART** macros for each of the control's connection points, and the **END_CONNECTION_MAP** macro to declare the end of the connection map.

See Also

Reference

BEGIN_CONNECTION_PART
BEGIN_CONNECTION_MAP
CONNECTION_IID

Concepts

MFC Macros and Globals

DECLARE_DHTMLEDITING_CMDMAP

Declares a DHTML editing command map in a class.

```
DECLARE_DHTMLEDITING_CMDMAP(className)
```

Parameters

className

The name of the class.

Remarks

This macro is to be used in the definition of CHtmlEditView-derived classes.

Use BEGIN_DHTMLEDITING_CMDMAP to implement the map.

Example

See HTMLEdit Sample.

See Also

Concepts

MFC Macros and Globals

Other Resources

DHTML Editing Command Maps

DECLARE_DHTML_EVENT_MAP

Declares a DHTML event map in a class definition.

```
DECLARE_DHTML_EVENT_MAP( )
```

Remarks

This macro is to be used in the definition of CDHtmlDialog-derived classes.

Use BEGIN_DHTML_EVENT_MAP or BEGIN_DHTML_EVENT_MAP_INLINE to implement the map.

DECLARE_DHTML_EVENT_MAP declares the following function:

```
virtual const DHtmlEventMapEntry* GetDHtmlEventMap( );
```

See Also

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DECLARE_DHTML_URL_EVENT_MAP

Declares a DHTML and URL event map in a class definition.

```
DECLARE_DHTML_URL_EVENT_MAP( )
```

Remarks

This macro is to be used in the definition of CMultiPageDHtmlDialog-derived classes.

A DHTML and URL event map contains embedded DHTML event maps and URL event entries to map DHTML events to handlers on a per-page basis. Use BEGIN_DHTML_URL_EVENT_MAP to implement the map.

See Also

Concepts

MFC Macros and Globals

Other Resources

Multipage DHTML and URL Event Maps

DECLARE_DISPATCH_MAP

If a **CCmdTarget**-derived class in your program supports OLE Automation, that class must provide a dispatch map to expose its methods and properties.

```
DECLARE_DISPATCH_MAP( )
```

Remarks

Use the **DECLARE_DISPATCH_MAP** macro at the end of your class declaration. Then, in the .CPP file that defines the member functions for the class, use the **BEGIN_DISPATCH_MAP** macro. Then include macro entries for each of your class's exposed methods and properties (**DISP_FUNCTION**, **DISP_PROPERTY**, and so on). Finally, use the **END_DISPATCH_MAP** macro.

Note

If you declare any members after **DECLARE_DISPATCH_MAP**, you must specify a new access type (**public**, **private**, or **protected**) for them.

The Application Wizard and code wizards assist in creating Automation classes and in maintaining dispatch maps. For more information on dispatch maps, see Automation Servers.

Example

```
// Example for DECLARE_DISPATCH_MAP
class CMyDoc : public CDocument
{
    // Member declarations

    DECLARE_DISPATCH_MAP()
};
```

See Also

Reference

BEGIN_DISPATCH_MAP
END_DISPATCH_MAP
DISP_FUNCTION
DISP_PROPERTY
DISP_PROPERTY_EX
DISP_DEFVALUE

Concepts

MFC Macros and Globals

Other Resources

Dispatch Maps

DECLARE_DYNAMIC

Adds the ability to access run-time information about an object's class when deriving a class from **CObject**.

```
DECLARE_DYNAMIC(class_name )
```

Parameters

class_name

The actual name of the class.

Remarks

Add the **DECLARE_DYNAMIC** macro to the header (.h) module for the class, then include that module in all .cpp modules that need access to objects of this class.

If you use the **DECLARE_DYNAMIC** and **IMPLEMENT_DYNAMIC** macros as described, you can then use the **RUNTIME_CLASS** macro and the **CObject::IsKindOf** function to determine the class of your objects at run time.

If **DECLARE_DYNAMIC** is included in the class declaration, then **IMPLEMENT_DYNAMIC** must be included in the class implementation.

For more information on the **DECLARE_DYNAMIC** macro, see CObject Class Topics.

Example

See the example for IMPLEMENT_DYNAMIC.

See Also

Reference

IMPLEMENT_DYNAMIC
DECLARE_DYNCREATE
DECLARE_SERIAL
RUNTIME_CLASS
CObject::IsKindOf

Concepts

MFC Macros and Globals

DECLARE_DYNCREATE

Enables objects of **CObject**-derived classes to be created dynamically at run time.

```
DECLARE_DYNCREATE(class_name )
```

Parameters

class_name

The actual name of the class.

Remarks

The framework uses this ability to create new objects dynamically. For example, the new view created when you open a new document. Document, view, and frame classes should support dynamic creation because the framework needs to create them dynamically.

Add the **DECLARE_DYNCREATE** macro in the .h module for the class, then include that module in all .cpp modules that need access to objects of this class.

If **DECLARE_DYNCREATE** is included in the class declaration, then **IMPLEMENT_DYNCREATE** must be included in the class implementation.

For more information on the **DECLARE_DYNCREATE** macro, see CObject Class Topics.

Note
The **DECLARE_DYNCREATE** macro includes all the functionality of **DECLARE_DYNAMIC**.

Example

See the example for IMPLEMENT_DYNCREATE.

See Also

Reference

DECLARE_DYNAMIC
IMPLEMENT_DYNAMIC
IMPLEMENT_DYNCREATE
RUNTIME_CLASS
CObject::IsKindOf

Concepts

MFC Macros and Globals

DECLARE_EVENT_MAP

Each **COleControl**-derived class in your program can provide an event map to specify the events your control will fire.

```
DECLARE_EVENT_MAP( )
```

Remarks

Use the **DECLARE_EVENT_MAP** macro at the end of your class declaration. Then, in the .cpp file that defines the member functions for the class, use the **BEGIN_EVENT_MAP** macro, macro entries for each of the control's events, and the **END_EVENT_MAP** macro to declare the end of the event list.

For more information on event maps, see the article ActiveX Controls: Events.

See Also

Reference

BEGIN_EVENT_MAP
END_EVENT_MAP
EVENT_CUSTOM
EVENT_CUSTOM_ID

Concepts

MFC Macros and Globals

DECLARE_EVENTSINK_MAP

An OLE container can provide an event sink map to specify the events your container will be notified of.

```
DECLARE_EVENTSINK_MAP( )
```

Remarks

Use the **DECLARE_EVENTSINK_MAP** macro at the end of your class declaration. Then, in the .CPP file that defines the member functions for the class, use the **BEGIN_EVENTSINK_MAP** macro, macro entries for each of the events to be notified of, and the **END_EVENTSINK_MAP** macro to declare the end of the event sink list.

For more information on event sink maps, see the article ActiveX Control Containers.

See Also

Reference

BEGIN_EVENTSINK_MAP
END_EVENTSINK_MAP
ON_EVENT
ON_PROPNOTIFY

Concepts

MFC Macros and Globals

DECLARE_MESSAGE_MAP

Declares that the class defines a message map. Each **CCmdTarget**-derived class in your program must provide a message map to handle messages.

```
DECLARE_MESSAGE_MAP( )
```

Remarks

Use the **DECLARE_MESSAGE_MAP** macro at the end of your class declaration. Then, in the .cpp file that defines the member functions for the class, use the **BEGIN_MESSAGE_MAP** macro, macro entries for each of your message-handler functions, and the **END_MESSAGE_MAP** macro.

Note

If you declare any member after **DECLARE_MESSAGE_MAP**, you must specify a new access type (**public**, **private**, or **protected**) for them.

For more information on message maps and the **DECLARE_MESSAGE_MAP** macro, see Message Handling and Mapping Topics.

Example

```
// Example for DECLARE_MESSAGE_MAP
class CMyWnd : public CFrameWnd
{
    // Member declarations

    DECLARE_MESSAGE_MAP( )
};
```

See Also

Reference

BEGIN_MESSAGE_MAP

END_MESSAGE_MAP

Concepts

MFC Macros and Globals

DECLARE_OLECREATE

Enables objects of **CCmdTarget**-derived classes to be created through OLE automation.

```
DECLARE_OLECREATE(class_name )
```

Parameters

class_name

The actual name of the class.

Remarks

This macro enables other OLE-enabled applications to create objects of this type.

Add the **DECLARE_OLECREATE** macro in the .h module for the class, and then include that module in all .cpp modules that need access to objects of this class.

If **DECLARE_OLECREATE** is included in the class declaration, then **IMPLEMENT_OLECREATE** must be included in the class implementation. A class declaration using **DECLARE_OLECREATE** must also use **DECLARE_DYNCREATE** or **DECLARE_SERIAL**.

Requirements

Header: afxdisp.h

See Also

Reference

IMPLEMENT_OLECREATE
DECLARE_DYNCREATE
DECLARE_SERIAL

Concepts

MFC Macros and Globals

DECLARE_OLECREATE_EX

Declares a class factory and the **GetClassID** member function of your control class.

```
DECLARE_OLECREATE_EX(class_name )
```

Parameters

class_name

The name of the control class.

Remarks

Use this macro in the control class header file for a control that does not support licensing.

Note that this macro serves the same purpose as the following code sample:

```
    BEGIN_OLEFACTORY(CSampleCtrl)
    END_OLEFACTORY(CSampleCtrl)
```

See Also

Reference

BEGIN_OLEFACTORY

END_OLEFACTORY

Concepts

MFC Macros and Globals

DECLARE_OLECTLTYPE

Declares the **GetUserTypeNameID** and **GetMiscStatus** member functions of your control class.

```
DECLARE_OLECTLTYPE(class_name )
```

Parameters

class_name

The name of the control class.

Remarks

GetUserTypeNameID and **GetMiscStatus** are pure virtual functions, declared in **COleControl**. Because these functions are pure virtual, they must be overridden in your control class. In addition to **DECLARE_OLECTLTYPE**, you must add the **IMPLEMENT_OLECTLTYPE** macro to your control class declaration.

See Also

Reference

IMPLEMENT_OLECTLTYPE

Concepts

MFC Macros and Globals

DECLARE_OLETYPELIB

Declares the **GetTypeLib** member function of your control class.

```
DECLARE_OLETYPELIB(class_name )
```

Parameters

class_name

The name of the control class related to the type library.

Remarks

Use this macro in the control class header file.

See Also

Reference

IMPLEMENT_OLETYPELIB

Concepts

MFC Macros and Globals

DECLARE_PROPPAGEIDS

Declares that the OLE control provides a list of property pages to display its properties.

```
DECLARE_PROPPAGEIDS(class_name )
```

Parameters

class_name

The name of the control class that owns the property pages.

Remarks

Use the **DECLARE_PROPPAGEIDS** macro at the end of your class declaration. Then, in the .cpp file that defines the member functions for the class, use the **BEGIN_PROPPAGEIDS** macro, macro entries for each of your control's property pages, and the **END_PROPPAGEIDS** macro to declare the end of the property page list.

For more information on property pages, see the article ActiveX Controls: Property Pages.

See Also

Reference

BEGIN_PROPPAGEIDS

END_PROPPAGEIDS

Concepts

MFC Macros and Globals

DECLARE_SERIAL

Generates the C++ header code necessary for a **CObject**-derived class that can be serialized.

```
DECLARE_SERIAL(class_name )
```

Parameters

class_name

The actual name of the class.

Remarks

Serialization is the process of writing or reading the contents of an object to and from a file.

Use the **DECLARE_SERIAL** macro in an .h module, and then include that module in all .cpp modules that need access to objects of this class.

If **DECLARE_SERIAL** is included in the class declaration, then **IMPLEMENT_SERIAL** must be included in the class implementation.

The **DECLARE_SERIAL** macro includes all the functionality of **DECLARE_DYNAMIC** and **DECLARE_DYNCREATE**.

You can use the **AFX_API** macro to automatically export the **CArchive** extraction operator for classes that use the **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** macros. Bracket the class declarations (located in the .h file) with the following code:

```
#undef AFX_API
#define AFX_API AFX_EXT_CLASS

<your class declarations here>

#undef AFX_API
#define AFX_API
```

For more information on the **DECLARE_SERIAL** macro, see CObject Class Topics.

Example

```
class CMyClass: public CObject
{
public:
   CMyClass( );
   void Serialize( CArchive& archive );

   DECLARE_SERIAL( CmyClass )
};
```

See Also

Reference

DECLARE_DYNAMIC
IMPLEMENT_SERIAL
RUNTIME_CLASS
CObject::IsKindOf

Concepts

MFC Macros and Globals

DEFAULT_PARSE_COMMAND

Directs the framework to call the default page that is identified by the *FnName* parameter if the request from a client to the CHttpServer object does not contain a command.

```
DEFAULT_PARSE_COMMAND(FnName, mapClass )
```

Parameters

FnName

The name of the member function. Also the name of the command.

mapClass

The class name to which the function is mapped.

Remarks

The **DEFAULT_PARSE_COMMAND** macro can appear anywhere in the parse map.

See ON_PARSE_COMMAND for a parse map example.

See Also

Reference

BEGIN_PARSE_MAP
ON_PARSE_COMMAND
ON_PARSE_COMMAND_PARAMS
END_PARSE_MAP
CHttpServer Class

Concepts

MFC Macros and Globals

DFX_Binary

Transfers arrays of bytes between the field data members of a CDaoRecordset object and the columns of a record on the data source.

```
void AFXAPI DFX_Binary(
   CDaoFieldExchange* pFX,
   LPCTSTR szName,
   CByteArray& value,
   int nPreAllocSize = AFX_DAO_BINARY_DEFAULT_SIZE,
   DWORD dwBindOptions = 0
);
```

Parameters

pFX

A pointer to an object of class CDaoFieldExchange. This object contains information to define the context for each call of the function.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. For a transfer from recordset to data source, the value, of type CByteArray, is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

nPreAllocSize

The framework preallocates this amount of memory. If your data is larger, the framework will allocated more space as needed. For better performance, set this size to a value large enough to prevent reallocations. The default size is defined in the AFXDAO.H file as **AFX_DAO_BINARY_DEFAULT_SIZE**.

dwBindOptions

An option that lets you take advantage of MFC's double buffering mechanism for detecting recordset fields that have changed. The default, **AFX_DAO_DISABLE_FIELD_CACHE**, does not use double buffering, and you must call SetFieldDirty and SetFieldNull yourself. The other possible value, **AFX_DAO_ENABLE_FIELD_CACHE**, uses double buffering, and you do not have to do extra work to mark fields dirty or Null. For performance and memory reasons, avoid this value unless your binary data is relatively small.

Note
You can control whether data is double buffered for all fields by default by setting CDaoRecordset::m_bCheckCacheForDirtyFields.

Remarks

Data is mapped between type **DAO_BYTES** in DAO and type CByteArray in the recordset.

Example

See DFX_Text.

See Also

Reference

DFX_Text
DFX_Bool
DFX_Currency
DFX_Long
DFX_Short
DFX_Single

DFX_Double
DFX_DateTime
DFX_Byte
DFX_LongBinary
CDaoFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

DFX_Bool

Transfers Boolean data between the field data members of a CDaoRecordset object and the columns of a record on the data source.

```
void AFXAPI DFX_Bool(
   CDaoFieldExchange* pFX,
   LPCTSTR szName,
   BOOL& value,
   DWORD dwBindOptions = AFX_DAO_ENABLE_FIELD_CACHE
);
```

Parameters

pFX

A pointer to an object of class CDaoFieldExchange. This object contains information to define the context for each call of the function.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. For a transfer from recordset to data source, the value, of type **BOOL**, is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

dwBindOptions

An option that lets you take advantage of MFC's double buffering mechanism for detecting recordset fields that have changed. The default, **AFX_DAO_ENABLE_FIELD_CACHE**, uses double buffering. The other possible value is **AFX_DAO_DISABLE_FIELD_CACHE**. If you specify this value, MFC does no checking on this field. You must call **SetFieldDirty** and **SetFieldNull** yourself.

Note
You can control whether data is double buffered by default by setting CDaoRecordset::m_bCheckCacheForDirtyFields.

Remarks

Data is mapped between type **DAO_BOOL** in DAO and type **BOOL** in the recordset.

Example

See DFX_Text.

See Also

Reference

DFX_Text
DFX_Long
DFX_Currency
DFX_Short
DFX_Single
DFX_Double
DFX_DateTime
DFX_Byte
DFX_Binary
DFX_LongBinary
CDaoFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

DFX_Byte

Transfers single bytes between the field data members of a CDaoRecordset object and the columns of a record on the data source.

```
void AFXAPI DFX_Byte(
   CDaoFieldExchange* pFX,
   LPCTSTR szName,
   BYTE& value,
   DWORD dwBindOptions = AFX_DAO_ENABLE_FIELD_CACHE
);
```

Parameters

pFX

A pointer to an object of class CDaoFieldExchange. This object contains information to define the context for each call of the function.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. For a transfer from recordset to data source, the value, of type **BYTE**, is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

dwBindOptions

An option that lets you take advantage of MFC's double buffering mechanism for detecting recordset fields that have changed. The default, **AFX_DAO_ENABLE_FIELD_CACHE**, uses double buffering. The other possible value is **AFX_DAO_DISABLE_FIELD_CACHE**. If you specify this value, MFC does no checking on this field. You must call **SetFieldDirty** and **SetFieldNull** yourself.

Note
You can control whether data is double buffered by default by setting CDaoRecordset::m_bCheckCacheForDirtyFields.

Remarks

Data is mapped between type **DAO_BYTES** in DAO and type **BYTE** in the recordset.

Example

See DFX_Text.

See Also

Reference

DFX_Text
DFX_Bool
DFX_Currency
DFX_Long
DFX_Short
DFX_Single
DFX_Double
DFX_DateTime
DFX_Binary
DFX_LongBinary
CDaoFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

DFX_Currency

Transfers currency data between the field data members of a CDaoRecordset object and the columns of a record on the data source.

```
void AFXAPI DFX_Currency(
   CDaoFieldExchange* pFX,
   LPCTSTR szName,
   COleCurrency& value,
   DWORD dwBindOptions = AFX_DAO_ENABLE_FIELD_CACHE
);
```

Parameters

pFX

A pointer to an object of class CDaoFieldExchange. This object contains information to define the context for each call of the function.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. For a transfer from recordset to data source, this value is taken from the specified data member, of type COleCurrency. For a transfer from data source to recordset, the value is stored in the specified data member.

dwBindOptions

An option that lets you take advantage of MFC's double buffering mechanism for detecting recordset fields that have changed. The default, **AFX_DAO_ENABLE_FIELD_CACHE**, uses double buffering. The other possible value is **AFX_DAO_DISABLE_FIELD_CACHE**. If you specify this value, MFC does no checking on this field. You must call **SetFieldDirty** and **SetFieldNull** yourself.

Note
You can control whether data is double buffered by default by setting CDaoRecordset::m_bCheckCacheForDirtyFields.

Remarks

Data is mapped between type **DAO_CURRENCY** in DAO and type COleCurrency in the recordset.

Example

See DFX_Text.

See Also

Reference

DFX_Text
DFX_Bool
DFX_DateTime
DFX_Long
DFX_Short
DFX_Single
DFX_Double
DFX_Byte
DFX_Binary
DFX_LongBinary
CDaoFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

DFX_DateTime

Transfers time and date data between the field data members of a CDaoRecordset object and the columns of a record on the data source.

```
void AFXAPI DFX_DateTime(
   CDaoFieldExchange* pFX,
   LPCTSTR szName,
   COleDateTime& value,
   DWORD dwBindOptions = AFX_DAO_ENABLE_FIELD_CACHE
);
```

Parameters

pFX

A pointer to an object of class CDaoFieldExchange. This object contains information to define the context for each call of the function.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. The function takes a reference to a COleDateTime object. For a transfer from recordset to data source, this value is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

dwBindOptions

An option that lets you take advantage of MFC's double buffering mechanism for detecting recordset fields that have changed. The default, **AFX_DAO_ENABLE_FIELD_CACHE**, uses double buffering. The other possible value is **AFX_DAO_DISABLE_FIELD_CACHE**. If you specify this value, MFC does no checking on this field. You must call **SetFieldDirty** and **SetFieldNull** yourself.

Note
You can control whether data is double buffered by default by setting CDaoRecordset::m_bCheckCacheForDirtyFields.

Remarks

Data is mapped between type **DAO_DATE** in DAO and type COleDateTime in the recordset.

Note
COleDateTime replaces CTime and **TIMESTAMP_STRUCT** for this purpose in the DAO classes. **CTime** and **TIMESTAMP_STRUCT** are still used for the ODBC-based data access classes.

Example

See DFX_Text.

See Also

Reference

DFX_Text
DFX_Bool
DFX_Currency
DFX_Long
DFX_Short
DFX_Single
DFX_Double
DFX_Byte
DFX_Binary

DFX_LongBinary
CDaoFieldExchange::SetFieldType
Concepts
MFC Macros and Globals

DFX_Double

Transfers **double float** data between the field data members of a CDaoRecordset object and the columns of a record on the data source.

```
void AFXAPI DFX_Double(
   CDaoFieldExchange* pFX,
   LPCTSTR szName,
   double& value,
   DWORD dwBindOptions = AFX_DAO_ENABLE_FIELD_CACHE
);
```

Parameters

pFX

A pointer to an object of class CDaoFieldExchange. This object contains information to define the context for each call of the function.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. For a transfer from recordset to data source, the value, of type **double**, is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

dwBindOptions

An option that lets you take advantage of MFC's double buffering mechanism for detecting recordset fields that have changed. The default, **AFX_DAO_ENABLE_FIELD_CACHE**, uses double buffering. The other possible value is **AFX_DAO_DISABLE_FIELD_CACHE**. If you specify this value, MFC does no checking on this field. You must call **SetFieldDirty** and **SetFieldNull** yourself.

Note
You can control whether data is double buffered by default by setting CDaoRecordset::m_bCheckCacheForDirtyFields.

Remarks

Data is mapped between type **DAO_R8** in DAO and type **double float** in the recordset.

Example

See DFX_Text.

See Also

Reference

DFX_Text
DFX_Bool
DFX_Currency
DFX_Long
DFX_Short
DFX_Single
DFX_DateTime
DFX_Byte
DFX_Binary
DFX_LongBinary
CDaoFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

DFX_Long

Transfers long integer data between the field data members of a CDaoRecordset object and the columns of a record on the data source.

```
void AFXAPI DFX_Long(
   CDaoFieldExchange* pFX,
   LPCTSTR szName,
   long& value,
   DWORD dwBindOptions = AFX_DAO_ENABLE_FIELD_CACHE
);
```

Parameters

pFX

A pointer to an object of class CDaoFieldExchange. This object contains information to define the context for each call of the function.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. For a transfer from recordset to data source, the value, of type **long**, is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

dwBindOptions

An option that lets you take advantage of MFC's double buffering mechanism for detecting recordset fields that have changed. The default, **AFX_DAO_ENABLE_FIELD_CACHE**, uses double buffering. The other possible value is **AFX_DAO_DISABLE_FIELD_CACHE**. If you specify this value, MFC does no checking on this field. You must call **SetFieldDirty** and **SetFieldNull** yourself.

Note
You can control whether data is double buffered by default by setting CDaoRecordset::m_bCheckCacheForDirtyFields.

Remarks

Data is mapped between type **DAO_I4** in DAO and type **long** in the recordset.

Example

See DFX_Text.

See Also

Reference

DFX_Text
DFX_Bool
DFX_Currency
DFX_Short
DFX_Single
DFX_Double
DFX_DateTime
DFX_Byte
DFX_Binary
DFX_LongBinary
CDaoFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

DFX_LongBinary

Important It is recommended that you use DFX_Binary instead of this function.

```
void AFXAPI DFX_LongBinary(
   CDaoFieldExchange* pFX,
   LPCTSTR szName,
   CLongBinary& value,
   DWORD dwPreAllocSize = AFX_DAO_LONGBINARY_DEFAULT_SIZE,
   DWORD dwBindOptions = 0
);
```

Parameters

pFX

A pointer to an object of class CDaoFieldExchange. This object contains information to define the context for each call of the function.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. For a transfer from recordset to data source, the value, of type CLongBinary, is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

dwPreAllocSize

The framework preallocates this amount of memory. If your data is larger, the framework will allocated more space as needed. For better performance, set this size to a value large enough to prevent reallocations.

dwBindOptions

An option that lets you take advantage of MFC's double buffering mechanism for detecting recordset fields that have changed. The default, **AFX_DISABLE_FIELD_CACHE**, does not use double buffering. The other possible value is **AFX_DAO_ENABLE_FIELD_CACHE**. Uses double buffering, and you do not have to do extra work to mark fields dirty or Null. For performance and memory reasons, avoid this value unless your binary data is relatively small.

Note
You can control whether data is double buffered by default by setting CDaoRecordset::m_bCheckCacheForDirtyFields.

Remarks

DFX_LongBinary is provided for compatibility with the MFC ODBC classes. The **DFX_LongBinary** function transfers binary large-object (BLOB) data using class **CLongBinary** between the field data members of a CDaoRecordset object and the columns of a record on the data source. Data is mapped between type **DAO_BYTES** in DAO and type CLongBinary in the recordset.

Example

See DFX_Text.

See Also

Reference

DFX_Text
DFX_Bool
DFX_Currency
DFX_Long
DFX_Short
DFX_Single
DFX_Double

DFX_DateTime
DFX_Byte
CDaoFieldExchange::SetFieldType
CLongBinary Class

Concepts

MFC Macros and Globals

DFX_Short

Transfers short integer data between the field data members of a CDaoRecordset object and the columns of a record on the data source.

```
void AFXAPI DFX_Short(
   CDaoFieldExchange* pFX,
   LPCTSTR szName,
   short& value,
   DWORD dwBindOptions = AFX_DAO_ENABLE_FIELD_CACHE
);
```

Parameters

pFX

A pointer to an object of class CDaoFieldExchange. This object contains information to define the context for each call of the function.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. For a transfer from recordset to data source, the value, of type **short**, is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

dwBindOptions

An option that lets you take advantage of MFC's double buffering mechanism for detecting recordset fields that have changed. The default, **AFX_DAO_ENABLE_FIELD_CACHE**, uses double buffering. The other possible value is **AFX_DAO_DISABLE_FIELD_CACHE**. If you specify this value, MFC does no checking on this field. You must call **SetFieldDirty** and **SetFieldNull** yourself.

Note
You can control whether data is double buffered by default by setting CDaoRecordset::m_bCheckCacheForDirtyFields.

Remarks

Data is mapped between type **DAO_I2** in DAO and type **short** in the recordset.

Note
DFX_Short is equivalent to RFX_Int for the ODBC-based classes.

Example

See DFX_Text.

See Also

Reference

DFX_Text
DFX_Bool
DFX_Currency
DFX_Long
DFX_Single
DFX_Double
DFX_DateTime
DFX_Byte
DFX_Binary
DFX_LongBinary

CDaoFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

DFX_Single

Transfers floating-point data between the field data members of a CDaoRecordset object and the columns of a record on the data source.

```
void AFXAPI DFX_Single(
   CDaoFieldExchange* pFX,
   LPCTSTR szName,
   float& value,
   DWORD dwBindOptions = AFX_DAO_ENABLE_FIELD_CACHE
);
```

Parameters

pFX

A pointer to an object of class CDaoFieldExchange. This object contains information to define the context for each call of the function.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. For a transfer from recordset to data source, the value, of type **float**, is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

dwBindOptions

An option that lets you take advantage of MFC's double buffering mechanism for detecting recordset fields that have changed. The default, **AFX_DAO_ENABLE_FIELD_CACHE**, uses double buffering. The other possible value is **AFX_DAO_DISABLE_FIELD_CACHE**. If you specify this value, MFC does no checking on this field. You must call **SetFieldDirty** and **SetFieldNull** yourself.

Note
You can control whether data is double buffered by default by setting CDaoRecordset::m_bCheckCacheForDirtyFields.

Remarks

Data is mapped between type **DAO_R4** in DAO and type **float** in the recordset.

Example

See DFX_Text.

See Also

Reference

DFX_Text
DFX_Bool
DFX_Currency
DFX_Long
DFX_Short
DFX_Double
DFX_DateTime
DFX_Byte
DFX_Binary
DFX_LongBinary
CDaoFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

DFX_Text

Transfers **CString** data between the field data members of a CDaoRecordset object and columns of a record on the data source.

```
void AFXAPI DFX_Text(
   CDaoFieldExchange* pFX,
   LPCTSTR szName,
   CString& value,
   int nPreAllocSize = AFX_DAO_TEXT_DEFAULT_SIZE,
      DWORD dwBindOptions = AFX_DAO_ENABLE_FIELD_CACHE
);
```

Parameters

pFX

A pointer to an object of class CDaoFieldExchange. This object contains information to define the context for each call of the function.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. For a transfer from recordset to data source, the value, of type CString, is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

nPreAllocSize

The framework preallocates this amount of memory. If your data is larger, the framework will allocated more space as needed. For better performance, set this size to a value large enough to prevent reallocations.

dwBindOptions

An option that lets you take advantage of MFC's double buffering mechanism for detecting recordset fields that have changed. The default, **AFX_DAO_ENABLE_FIELD_CACHE**, uses double buffering. The other possible value is **AFX_DAO_DISABLE_FIELD_CACHE**. If you specify this value, MFC does no checking on this field. You must call SetFieldDirty and SetFieldNull yourself.

Note
You can control whether data is double buffered by default by setting CDaoRecordset::m_bCheckCacheForDirtyFields.

Remarks

Data is mapped between type **DAO_CHAR** in DAO (or, if the symbol **_UNICODE** is defined, **DAO_WCHAR**) and type CString in the recordset.

Example

This example shows several calls to **DFX_Text**. Notice also the two calls to CDaoFieldExchange::SetFieldType. You must write the first call to **SetFieldType** and its **DFX** call. The second call and its associated **DFX** calls are normally written by the code wizard that generated the class.

```
//Example for DFX_Text
void CSections::DoFieldExchange(CDaoFieldExchange* pFX)
{
    pFX->SetFieldType(CDaoFieldExchange::param);
    DFX_Text(pFX, "Name", m_strNameParam);
    pFX->SetFieldType(CDaoFieldExchange::outputColumn);
    DFX_Text(pFX, "CourseID", m_strCourseID);
    DFX_Text(pFX, "InstructorID", m_strInstructorID);
```

```
    DFX_Short(pFX, "LabFee", m_nRoomNo);
    DFX_Text(pFX, "LabFee", m_strSchedule);
    DFX_Short(pFX, "SectionNo", m_nSectionNo);
    DFX_Currency(pFX, "LabFee", m_currLabFee);
}
```

See Also

Reference

DFX_Bool
DFX_Long
DFX_Currency
DFX_Short
DFX_Single
DFX_Double
DFX_DateTime
DFX_Byte
DFX_Binary
DFX_LongBinary
CDaoFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

DHTMLEDITING_CMD_ENTRY

Maps a command ID to an HTML editing command.

```
DHTMLEDITING_CMD_ENTRY(cmdID, dhtmlcmdID)
```

Parameters

cmdID

The command ID (such as **ID_EDIT_COPY**).

dhtmlcmdID

The HTML editing command to which *cmdID* maps (such as **IDM_COPY**).

Example

See HTMLEdit Sample.

See Also

Reference

DHTMLEDITING_CMD_ENTRY_FUNC
DHTMLEDITING_CMD_ENTRY_FUNC_TYPE
DHTMLEDITING_CMD_ENTRY_TYPE

Concepts

MFC Macros and Globals

Other Resources

DHTML Editing Command Maps

DHTMLEDITING_CMD_ENTRY_FUNC

Maps a command ID to an HTML editing command and message handler.

```
DHTMLEDITING_CMD_ENTRY_FUNC(cmdID, dhtmlcmdID, member_func_name)
```

Parameters

cmdID

The command ID (such as **ID_EDIT_COPY**).

dhtmlcmdID

The HTML editing command to which *cmdID* maps (such as **IDM_COPY**).

member_func_name

The name of the message-handler function to which the command is mapped.

Example

See HTMLEdit Sample.

See Also

Reference

DHTMLEDITING_CMD_ENTRY
DHTMLEDITING_CMD_ENTRY_FUNC_TYPE
DHTMLEDITING_CMD_ENTRY_TYPE

Concepts

MFC Macros and Globals

Other Resources

DHTML Editing Command Maps

DHTMLEDITING_CMD_ENTRY_FUNC_TYPE

Maps a command ID to an HTML editing command, message handler, and user interface element.

```
DHTMLEDITING_CMD_ENTRY_FUNC_TYPE(cmdID, dhtmlcmdID, member_func_name, elemType)
```

Parameters

cmdID

The command ID (such as **ID_EDIT_COPY**).

dhtmlcmdID

The HTML editing command to which *cmdID* maps (such as **IDM_COPY**).

member_func_name

The name of the message-handler function to which the command is mapped.

elemType

The user interface element type; one of **AFX_UI_ELEMTYPE_NORMAL**, **AFX_UI_ELEMTYPE_CHECKBOX**, or **AFX_UI_ELEMTYPE_RADIO**.

Example

See HTMLEdit Sample.

See Also

Reference

DHTMLEDITING_CMD_ENTRY
DHTMLEDITING_CMD_ENTRY_FUNC
DHTMLEDITING_CMD_ENTRY_TYPE

Concepts

MFC Macros and Globals

Other Resources

DHTML Editing Command Maps

DHTMLEDITING_CMD_ENTRY_TYPE

Maps a command ID to an HTML editing command and user interface element.

```
DHTMLEDITING_CMD_ENTRY_TYPE(cmdID, dhtmlcmdID, elemType)
```

Parameters

cmdID

The command ID (such as **ID_EDIT_COPY**).

dhtmlcmdID

The HTML editing command to which *cmdID* maps (such as **IDM_COPY**).

elemType

The user interface element type; one of **AFX_UI_ELEMTYPE_NORMAL**, **AFX_UI_ELEMTYPE_CHECKBOX**, or **AFX_UI_ELEMTYPE_RADIO**.

Example

See HTMLEdit Sample.

See Also

Reference

DHTMLEDITING_CMD_ENTRY
DHTMLEDITING_CMD_ENTRY_FUNC
DHTMLEDITING_CMD_ENTRY_FUNC_TYPE

Concepts

MFC Macros and Globals

Other Resources

DHTML Editing Command Maps

DHTML_EVENT

Handles (at the document level) an event identified by *dispid* originated by the HTML element identified by *elemName*.

```
DHTML_EVENT(dispid, elemName, memberFxn )
```

Parameters

dispid

The DISPID of the event to be handled.

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event, or **NULL** to handle document events.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_AXCONTROL

Handles the event identified by *dispid* fired by the ActiveX control identified by *controlName*.

```
DHTML_EVENT_AXCONTROL(dispid, controlName, memberFxn )
```

Parameters

dispid

The dispatch ID of the event to be handled.

controlName

An **LPCWSTR** holding the HTML ID of the control firing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_CLASS

Handles (at the document level) an event identified by *dispid* originated by any HTML element with the CSS class identified by *elemName*.

```
DHTML_EVENT_CLASS(dispid, elemName, memberFxn )
```

Parameters

dispid

The dispatch ID of the event to be handled.

elemName

An **LPCWSTR** holding the CSS class of the HTML elements sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ELEMENT

Handles (at the element identified by *elemName*) an event identified by *dispid*.

```
DHTML_EVENT_ELEMENT(dispid, elemName, memberFxn )
```

Parameters

dispid

The dispatch ID of the event to be handled.

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

If this macro is used to handle nonbubbling events, the source of the event will be the element identified by *elemName*.

If this macro is used to handle bubbling events, the element identified by *elemName* may not be the source of the event (the source could be any element contained by *elemName*).

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONAFTERUPDATE

Handles (at the document level) the **onafterupdate** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONAFTERUPDATE(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONBEFOREUPDATE

Handles (at the document level) the **onbeforeupdate** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONBEFOREUPDATE(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONBLUR

Handles (at the element level) the **onblur** event. This is a nonbubbling event.

```
DHTML_EVENT_ONBLUR(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONCHANGE

Handles (at the element level) the **onchange** event. This is a nonbubbling event.

```
DHTML_EVENT_ONCHANGE(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONCLICK

Handles (at the document level) the **onclick** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONCLICK(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONDATAAVAILABLE

Handles (at the document level) the **ondataavailable** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONDATAAVAILABLE(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONDATASETCHANGED

Handles (at the document level) the **ondatasetchanged** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONDATASETCHANGED(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONDATASETCOMPLETE

Handles (at the document level) the **ondatasetcomplete** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONDATASETCOMPLETE(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONDBLCLICK

Handles (at the document level) the **ondblclick** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONDBLCLICK(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONDRAGSTART

Handles (at the document level) the **ondragstart** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONDRAGSTART(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONERRORUPDATE

Handles (at the document level) the **onerrorupdate** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONERRORUPDATE(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONFILTERCHANGE

Handles (at the document level) the **onfilterchange** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONFILTERCHANGE(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONFOCUS

Handles (at the element level) the **onfocus** event. This is a nonbubbling event.

```
DHTML_EVENT_ONFOCUS(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONHELP

Handles (at the document level) the **onhelp** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONHELP(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONKEYDOWN

Handles (at the document level) the **onkeydown** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONKEYDOWN(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONKEYPRESS

Handles (at the document level) the **onkeypress** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONKEYPRESS(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONKEYUP

Handles (at the document level) the **onkeyup** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONKEYUP(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONMOUSEDOWN

Handles (at the document level) the **onmousedown** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONMOUSEDOWN(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONMOUSEMOVE

Handles (at the document level) the **onmousemove** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONMOUSEMOVE(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONMOUSEOUT

Handles (at the document level) the **onmouseout** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONMOUSEOUT(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONMOUSEOVER

Handles (at the document level) the **onmouseover** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONMOUSEOVER(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONMOUSEUP

Handles (at the document level) the **onmouseup** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONMOUSEUP(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONRESIZE

Handles (at the element level) the **onresize** event. This is a nonbubbling event.

```
DHTML_EVENT_ONRESIZE(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONROWENTER

Handles (at the document level) the **onrowenter** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONROWENTER(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONROWEXIT

Handles (at the document level) the **onrowexit** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONROWEXIT(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_ONSELECTSTART

Handles (at the document level) the **onselectstart** event originated by the HTML element identified by *elemName*.

```
DHTML_EVENT_ONSELECTSTART(elemName, memberFxn )
```

Parameters

elemName

An **LPCWSTR** holding the ID of the HTML element sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DHTML_EVENT_TAG

Handles (at the document level) an event identified by *dispid* originated by any HTML element with the HTML tag identified by *elemName*.

```
DHTML_EVENT_TAG(dispid, elemName, memberFxn )
```

Parameters

dispid

The dispatch ID of the event to be handled.

elemName

The HTML tag of the HTML elements sourcing the event.

memberFxn

The handler function for the event.

Remarks

Use this macro to add an entry to the DHTML event map in your class.

See Also

Reference

CDHtmlDialog Class

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

DISP_DEFVALUE

Makes an existing property the default value of an object.

```
DISP_DEFVALUE(theClass, pszName )
```

Parameters

theClass

Name of the class.

pszName

External name of the property that represents the "value" of the object.

Remarks

Using a default value can make programming your automation object simpler for Visual Basic applications.

The "default value" of your object is the property that is retrieved or set when a reference to an object does not specify a property or member function.

Requirements

Header: afxdisp.h

See Also

Reference

DECLARE_DISPATCH_MAP
DISP_PROPERTY_EX
DISP_FUNCTION
BEGIN_DISPATCH_MAP
END_DISPATCH_MAP

Concepts

MFC Macros and Globals

Other Resources

Dispatch Maps

DISP_FUNCTION

Defines an OLE automation function in a dispatch map.

```
DISP_FUNCTION(theClass, pszName, pfnMember, vtRetVal, vtsParams )
```

Parameters

theClass

Name of the class.

pszName

External name of the function.

pfnMember

Name of the member function.

vtRetVal

A value specifying the function's return type.

vtsParams

A space-separated list of one or more constants specifying the function's parameter list.

Remarks

The *vtRetVal* argument is of type **VARTYPE**. The following possible values for this argument are taken from the **VARENUM** enumeration:

Symbol	Return type
VT_EMPTY	**void**
VT_I2	**short**
VT_I4	**long**
VT_R4	**float**
VT_R8	**double**
VT_CY	**CY**
VT_DATE	**DATE**
VT_BSTR	**BSTR**
VT_DISPATCH	**LPDISPATCH**
VT_ERROR	**SCODE**
VT_BOOL	**BOOL**
VT_VARIANT	**VARIANT**
VT_UNKNOWN	**LPUNKNOWN**

The *vtsParams* argument is a space-separated list of values from the **VTS_** constants. One or more of these values separated by spaces (not commas) specifies the function's parameter list. For example,

```
VTS_I2 VTS_PI2
```

specifies a list containing a short integer followed by a pointer to a short integer.

The **VTS_** constants and their meanings are as follows:

Symbol	Parameter type
VTS_I2	**Short**
VTS_I4	**Long**
VTS_R4	**Float**
VTS_R8	**Double**
VTS_CY	**const CY** or **CY***
VTS_DATE	**DATE**
VTS_BSTR	**LPCSTR**
VTS_DISPATCH	**LPDISPATCH**
VTS_SCODE	**SCODE**
VTS_BOOL	**BOOL**
VTS_VARIANT	**const VARIANT*** or **VARIANT&**
VTS_UNKNOWN	**LPUNKNOWN**
VTS_PI2	**short***
VTS_PI4	**long***
VTS_PR4	**float***
VTS_PR8	**double***
VTS_PCY	**CY***
VTS_PDATE	**DATE***
VTS_PBSTR	**BSTR***
VTS_PDISPATCH	**LPDISPATCH***
VTS_PSCODE	**SCODE***
VTS_PBOOL	**BOOL***

VTS_PVARIANT	**VARIANT***
VTS_PUNKNOWN	**LPUNKNOWN***
VTS_NONE	No parameters

See Also

Reference

DECLARE_DISPATCH_MAP
DISP_PROPERTY
DISP_PROPERTY_EX
BEGIN_DISPATCH_MAP
END_DISPATCH_MAP

Concepts

MFC Macros and Globals

Other Resources

Dispatch Maps

DISP_PROPERTY

Defines an OLE automation property in a dispatch map.

```
DISP_PROPERTY(theClass, pszName, memberName, vtPropType )
```

Parameters

theClass

Name of the class.

pszName

External name of the property.

memberName

Name of the member variable in which the property is stored.

vtPropType

A value specifying the property's type.

Remarks

The *vtPropType* argument is of type **VARTYPE**. Possible values for this argument are taken from the **VARENUM** enumeration:

Symbol	Property type
VT_I2	**short**
VT_I4	**long**
VT_R4	**float**
VT_R8	**double**
VT_CY	**CY**
VT_DATE	**DATE**
VT_BSTR	**CString**
VT_DISPATCH	**LPDISPATCH**
VT_ERROR	**SCODE**
VT_BOOL	**BOOL**
VT_VARIANT	**VARIANT**
VT_UNKNOWN	**LPUNKNOWN**

When an external client changes the property, the value of the member variable specified by *memberName* changes; there is no notification of the change.

Requirements

Header: afxdisp.h

See Also

Reference

DECLARE_DISPATCH_MAP
DISP_PROPERTY_EX
DISP_FUNCTION
BEGIN_DISPATCH_MAP
END_DISPATCH_MAP

Concepts

MFC Macros and Globals

Other Resources

Dispatch Maps

DISP_PROPERTY_EX

Defines an OLE automation property and name the functions used to get and set the property's value in a dispatch map.

```
DISP_PROPERTY_EX(theClass, pszName, memberGet, memberSet, vtPropType )
```

Parameters

theClass

Name of the class.

pszName

External name of the property.

memberGet

Name of the member function used to get the property.

memberSet

Name of the member function used to set the property.

vtPropType

A value specifying the property's type.

Remarks

The *memberGet* and *memberSet* functions have signatures determined by the *vtPropType* argument. The *memberGet* function takes no arguments and returns a value of the type specified by *vtPropType*. The *memberSet* function takes an argument of the type specified by *vtPropType* and returns nothing.

The *vtPropType* argument is of type **VARTYPE**. Possible values for this argument are taken from the **VARENUM** enumeration. For a list of these values, see the Remarks for the *vtRetVal* parameter in DISP_FUNCTION. Note that **VT_EMPTY**, listed in the **DISP_FUNCTION** remarks, is not permitted as a property data type.

Requirements

Header: afxdisp.h

See Also

Reference

DECLARE_DISPATCH_MAP
DISP_PROPERTY
DISP_FUNCTION
BEGIN_DISPATCH_MAP
END_DISPATCH_MAP

Concepts

MFC Macros and Globals

Other Resources

Dispatch Maps

DISP_PROPERTY_NOTIFY

Defines an OLE automation property with notification in a dispatch map.

```
DISP_PROPERTY_NOTIFY(theClass, szExternalName, memberName, pfnAfterSet, vtPropType )
```

Parameters

theClass

Name of the class.

szExternalName

External name of the property.

memberName

Name of the member variable in which the property is stored.

pfnAfterSet

Name of the notification function for *szExternalName*.

vtPropType

A value specifying the property's type.

Remarks

Unlike properties defined with **DISP_PROPERTY**, a property defined with **DISP_PROPERTY_NOTIFY** will automatically call the function specified by *pfnAfterSet* when the property is changed.

The *vtPropType* argument is of type **VARTYPE**. Possible values for this argument are taken from the **VARENUM** enumeration:

Symbol	Property type
VT_I2	**short**
VT_I4	**long**
VT_R4	**float**
VT_R8	**double**
VT_CY	**CY**
VT_DATE	**DATE**
VT_BSTR	**CString**
VT_DISPATCH	**LPDISPATCH**
VT_ERROR	**SCODE**
VT_BOOL	**BOOL**
VT_VARIANT	**VARIANT**
VT_UNKNOWN	**LPUNKNOWN**

Requirements

Header: afxdisp.h

See Also

Reference

DISP_PROPERTY

DISP_FUNCTION

Concepts

MFC Macros and Globals

Other Resources

Dispatch Maps

DISP_PROPERTY_PARAM

Defines a property accessed with separate **Get** and **Set** member functions.

```
DISP_PROPERTY_PARAM(theClass, pszExternalName, pfnGet, pfnSet, vtPropType, vtsParams )
```

Parameters

theClass

Name of the class.

pszExternalName

External name of the property.

pfnGet

Name of the member function used to get the property.

pfnSet

Name of the member function used to set the property.

vtPropType

A value specifying the property's type.

vtsParams

A string of space-separated **VTS_** variant parameter types, one for each parameter.

Remarks

Unlike the **DISP_PROPERTY_EX** macro, this macro allows you to specify a parameter list for the property. This is useful for implementing properties that are indexed or parameterized.

Example

Consider the following declaration of get and set member functions that allow the user to request a specific row and column when accessing the property:

```
afx_msg short GetArray( short row, short column );
afx_msg short SetArray( short row, short column, short nNewValue );
```

These correspond to the following **DISP_PROPERTY_PARAM** macro in the control dispatch map:

```
DISP_PROPERTY_PARAM( CMyCtrl, "Array", GetArray, SetArray, VT-I2, VTS_I2 VTS_I2 )
```

As another example, consider the following get and set member functions:

```
LPDISPATCH CMyObject::GetItem( short index1, short index2, short index3 );
void CMyObject::SetItem( short index1, short index2, short index3, LPDISPATCH newValue );
```

These correspond to the following **DISP_PROPERTY_PARAM** macro in the control dispatch map:

```
DISP_PROPERTY_PARAM( CMyObject, "item", GetItem, SetItem, VT_DISPATCH,
   VTS_I2 VTS_I2 VTS_I2 )
```

Requirements

Header: afxdisp.h

See Also

Reference

DISP_PROPERTY_EX

Concepts

MFC Macros and Globals

Other Resources

Dispatch Maps

DumpElements

Provides stream-oriented diagnostic output in text form for the elements of your collection when overridden.

```
template<class TYPE>
void AFXAPI DumpElements(
   CDumpContext& dc,
   const TYPE* pElements,
   INT_PTR nCount
);
```

Parameters

dc

Dump context for dumping elements.

TYPE

Template parameter specifying the type of the elements.

pElements

Pointer to the elements to be dumped.

nCount

Number of elements to be dumped.

Remarks

The **CArray::Dump**, **CList::Dump**, and **CMap::Dump** functions call this if the depth of the dump is greater than 0.

The default implementation does nothing. If the elements of your collection are derived from **CObject**, your override will typically iterate through the collection's elements, calling **Dump** for each element in turn.

For information on diagnostics and on the **Dump** function, see Debugging MFC Applications.

See Also

Reference

CDumpContext::SetDepth
CObject::Dump
CArray Class
CList Class
CMap Class

Concepts

MFC Macros and Globals

DYNAMIC_DOWNCAST

Provides a handy way to cast a pointer to a pointer to a class object while checking to see if the cast is legal.

```
DYNAMIC_DOWNCAST(class, pointer )
```

Parameters

class

The name of a class.

pointer

A pointer to be cast to a pointer to an object of type *class*.

Remarks

The macro will cast the *pointer* parameter to a pointer to an object of the *class* parameter's type.

If the object referenced by the pointer is a "kind of" the identified class, the macro returns the appropriate pointer. If it is not a legal cast, the macro returns **NULL**.

See Also

Reference

STATIC_DOWNCAST

dynamic_cast Operator

Concepts

MFC Macros and Globals

END_CATCH

Marks the end of the last **CATCH** or **AND_CATCH** block.

```
END_CATCH
```

Remarks

For more information on the **END_CATCH** macro, see the article Exceptions.

See Also

Reference

TRY
CATCH
AND_CATCH
THROW (MFC)
THROW_LAST

Concepts

MFC Macros and Globals

END_CATCH_ALL

Marks the end of the last **CATCH_ALL** or **AND_CATCH_ALL** block.

```
END_CATCH_ALL
```

See Also

Reference

TRY
CATCH_ALL
AND_CATCH_ALL
THROW (MFC)
THROW_LAST

Concepts

MFC Macros and Globals

END_CONNECTION_MAP

Ends the definition of your connection map.

```
END_CONNECTION_MAP( )
```

See Also

Reference

BEGIN_CONNECTION_MAP

DECLARE_CONNECTION_MAP

Concepts

MFC Macros and Globals

END_CONNECTION_PART

Ends the declaration of your connection point.

```
END_CONNECTION_PART(localClass )
```

Parameters

localClass

Specifies the name of the local class that implements the connection point.

See Also

Reference

BEGIN_CONNECTION_PART

DECLARE_CONNECTION_MAP

Concepts

MFC Macros and Globals

END_DELEGATE_MAP

Ends a delegate map.

```
END_DELEGATE_MAP();
```

Remarks

This macro marks the end of a list of delegate entries, which compose a delegate map. For an example of how this macro is used, see EVENT_DELEGATE_ENTRY.

Requirements

Header: msclr\event.h

See Also

Tasks

How to: Sink Windows Forms Events from Native C++ Classes

Reference

MAKE_DELEGATE

EVENT_DELEGATE_ENTRY

BEGIN_DELEGATE_MAP

END_DHTMLEDITING_CMDMAP

Marks the end of a DHTML editing command map.

```
END_DHTMLEDITING_CMDMAP( )
```

Remarks

Use in conjunction with BEGIN_DHTMLEDITING_CMDMAP.

Example

See HTMLEdit Sample.

See Also

Concepts

MFC Macros and Globals

Other Resources

DHTML Editing Command Maps

END_DHTML_EVENT_MAP

Marks the end of the DHTML event map.

```
END_DHTML_EVENT_MAP( )
```

Remarks

Must be used in conjunction with BEGIN_DHTML_EVENT_MAP.

See Also

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

END_DHTML_EVENT_MAP_INLINE

Marks the end of the DHTML event map.

```
END_DHTML_EVENT_MAP_INLINE( )
```

Remarks

Must be used in conjunction with BEGIN_DHTML_EVENT_MAP_INLINE.

See Also

Concepts

MFC Macros and Globals

Other Resources

DHTML Event Maps

END_DHTML_URL_EVENT_MAP

Marks the end of a DHTML and URL event map.

```
END_DHTML_URL_EVENT_MAP(className)
```

Parameters

className

The name of the class containing the event map. This class should derive directly or indirectly from CMultiPageDHtmlDialog. This should match *className* in the corresponding BEGIN_DHTML_URL_EVENT_MAP macro.

Example

See the example in BEGIN_DHTML_URL_EVENT_MAP.

See Also

Concepts

MFC Macros and Globals

Other Resources

Multipage DHTML and URL Event Maps

END_DISPATCH_MAP

Ends the definition of your dispatch map.

```
END_DISPATCH_MAP( )
```

Remarks

It must be used in conjunction with **BEGIN_DISPATCH_MAP**.

Requirements

Header: afxdisp.h

See Also

Reference

DECLARE_DISPATCH_MAP
BEGIN_DISPATCH_MAP
DISP_FUNCTION
DISP_PROPERTY
DISP_PROPERTY_EX
DISP_DEFVALUE

Concepts

MFC Macros and Globals

Other Resources

Dispatch Maps

END_EMBED_DHTML_EVENT_MAP

Marks the end of an embedded DHTML event map.

```
END_EMBED_DHTML_EVENT_MAP( )
```

Example

See the example in BEGIN_DHTML_URL_EVENT_MAP.

See Also

Reference

BEGIN_EMBED_DHTML_EVENT_MAP

Concepts

MFC Macros and Globals

Other Resources

Multipage DHTML and URL Event Maps

END_EVENT_MAP

Use the **END_EVENT_MAP** macro to end the definition of your event map.

```
END_EVENT_MAP( )
```

See Also

Reference

DECLARE_EVENT_MAP

BEGIN_EVENT_MAP

Concepts

MFC Macros and Globals

END_INTERFACE_MAP

Ends the interface map in the implementation file.

```
END_INTERFACE_MAP( )
```

Remarks

For more information about interface maps, see Technical Note 38.

See Also

Reference

BEGIN_INTERFACE_MAP

Concepts

MFC Macros and Globals

END_EVENTSINK_MAP

Ends the definition of your event sink map.

```
END_EVENTSINK_MAP( )
```

See Also

Reference

DECLARE_EVENTSINK_MAP

BEGIN_EVENTSINK_MAP

Concepts

MFC Macros and Globals

END_MESSAGE_MAP

Ends the definition of your message map.

```
END_MESSAGE_MAP( )
```

Remarks

For more information on message maps and the **END_MESSAGE_MAP** macro, see Message Handling and Mapping Topics.

See Also

Reference

DECLARE_MESSAGE_MAP
BEGIN_MESSAGE_MAP
Message Maps (MFC)

Concepts

MFC Macros and Globals

END_OLEFACTORY

Ends the declaration of your control's class factory.

```
END_OLEFACTORY(class_name )
```

Parameters

class_name

The name of the control class whose class factory this is.

See Also

Reference

BEGIN_OLEFACTORY

DECLARE_OLECREATE_EX

Concepts

MFC Macros and Globals

END_PARSE_MAP

Ends the definition of your parse map.

```
END_PARSE_MAP(theClass )
```

Parameters

theClass

Specifies the name of the class that owns this parse map.

Remarks

This macro must be used in conjunction with BEGIN_PARSE_MAP.

See ON_PARSE_COMMAND for a parse map example.

See Also

Reference

BEGIN_PARSE_MAP
ON_PARSE_COMMAND
ON_PARSE_COMMAND_PARAMS
DEFAULT_PARSE_COMMAND
CHttpServer Class

Concepts

MFC Macros and Globals

END_PROPPAGEIDS

Ends the definition of your property page ID list.

```
END_PROPPAGEIDS(class_name )
```

Parameters

class_name

The name of the control class that owns the property page.

See Also

Reference

DECLARE_PROPPAGEIDS

BEGIN_PROPPAGEIDS

Concepts

MFC Macros and Globals

END_URL_ENTRIES

Marks the end of a URL event entry map.

```
END_URL_ENTRIES( )
```

Example

See the example in BEGIN_DHTML_URL_EVENT_MAP.

See Also

Reference

BEGIN_URL_ENTRIES

Concepts

MFC Macros and Globals

Other Resources

Multipage DHTML and URL Event Maps

ENSURE (MFC)

Use to validate data correctness.

```
ENSURE(
      booleanExpression
)
ENSURE_VALID(
      booleanExpression
)
```

Parameters

booleanExpression

Specifies a boolean expression to be tested.

Remarks

The purpose of these macros is to improve the validation of parameters. The macros prevent further processing of incorrect parameters in your code. Unlike the **ASSERT** macros, the **ENSURE** macros throw an exception in addition to generating an assertion.

The macros behave in two ways, according to the project configuration. The macros call **ASSERT** and then throw an exception if the assertion fails. Thus, in Debug configurations (that is, where **_DEBUG** is defined) the macros produce an assertion and exception while in Release configurations, the macros produce only the exception (**ASSERT** does not evaluate the expression in Release configurations).

The macro **ENSURE_ARG** acts like the **ENSURE** macro.

ENSURE_VALID calls the **ASSERT_VALID** macro (which has an effect only in Debug builds). In addition, **ENSURE_VALID** throws an exception if the pointer is NULL. The NULL test is performed in both Debug and Release configurations.

If any of these tests fails, an alert message is displayed in the same manner as **ASSERT**. The macro throws an invalid argument exception if needed.

Example

```
// example for ENSURE and ENSURE_VALID
void CMyClass::Open(CMyObject* pObj)
{
   ENSURE_VALID(pMyObject);
   ENSURE(m_CubeLen > 0);
   pObj->DoSomething();
   ...
}
```

See Also

Reference

VERIFY

ATLENSURE

Concepts

MFC Macros and Globals

EVENT_CUSTOM

Defines an event-map entry for a custom event.

```
EVENT_CUSTOM(pszName, pfnFire, vtsParams )
```

Parameters

pszName

The name of the event.

pfnFire

The name of the event firing function.

vtsParams

A space-separated list of one or more constants specifying the function's parameter list.

Remarks

The *vtsParams* parameter is a space-separated list of values from the **VTS_** constants. One or more of these values separated by spaces (not commas) specifies the function's parameter list. For example:

```
VTS_COLOR VTS_FONT
```

specifies a list containing a 32-bit integer representing an RGB color value, followed by a pointer to the **IFontDisp** interface of an OLE font object.

The **VTS_** constants and their meanings are as follows:

Symbol	Parameter type
VTS_I2	**short**
VTS_I4	**long**
VTS_R4	**float**
VTS_R8	**double**
VTS_COLOR	**OLE_COLOR**
VTS_CY	**CURRENCY**
VTS_DATE	**DATE**
VTS_BSTR	**const char***
VTS_DISPATCH	**LPDISPATCH**
VTS_FONT	**IFontDispatch***
VTS_HANDLE	**HANDLE**
VTS_SCODE	**SCODE**
VTS_BOOL	**BOOL**

VTS_VARIANT	**const VARIANT***
VTS_PVARIANT	**VARIANT***
VTS_UNKNOWN	**LPUNKNOWN**
VTS_OPTEXCLUSIVE	**OLE_OPTEXCLUSIVE**
VTS_PICTURE	**IPictureDisp***
VTS_TRISTATE	**OLE_TRISTATE**
VTS_XPOS_PIXELS	**OLE_XPOS_PIXELS**
VTS_YPOS_PIXELS	**OLE_YPOS_PIXELS**
VTS_XSIZE_PIXELS	**OLE_XSIZE_PIXELS**
VTS_YSIZE_PIXELS	**OLE_YSIZE_PIXELS**
VTS_XPOS_HIMETRIC	**OLE_XPOS_HIMETRIC**
VTS_YPOS_HIMETRIC	**OLE_YPOS_HIMETRIC**
VTS_XSIZE_HIMETRIC	**OLE_XSIZE_HIMETRIC**
VTS_YSIZE_HIMETRIC	**OLE_YSIZE_HIMETRIC**

Note

Additional variant constants have been defined for all variant types, with the exception of **VTS_FONT** and **VTS_PICTURE**, that provide a pointer to the variant data constant. These constants are named using the **VTS_P***constantname* convention. For example, **VTS_PCOLOR** is a pointer to a **VTS_COLOR** constant.

See Also

Reference

EVENT_CUSTOM_ID

DECLARE_EVENT_MAP

Variant Parameter Type Constants

Concepts

MFC Macros and Globals

EVENT_CUSTOM_ID

Defines an event firing function for a custom event belonging to the dispatch ID specified by *dispid*.

```
EVENT_CUSTOM_ID(pszName, dispid, pfnFire, vtsParams )
```

Parameters

pszName

The name of the event.

dispid

The dispatch ID used by the control when firing the event.

pfnFire

The name of the event firing function.

vtsParams

A variable list of parameters passed to the control container when the event is fired.

Remarks

The *vtsParams* argument is a space-separated list of values from the **VTS_** constants. One or more of these values separated by spaces, not commas, specifies the function's parameter list. For example:

```
VTS_COLOR VTS_FONT
```

specifies a list containing a 32-bit integer representing an RGB color value, followed by a pointer to the **IFontDisp** interface of an OLE font object.

For a list of the **VTS_** constants, see EVENT_CUSTOM.

See Also

Reference

EVENT_CUSTOM

Variant Parameter Type Constants

Concepts

MFC Macros and Globals

EVENT_DELEGATE_ENTRY

Creates an entry in the delegate map.

```
EVENT_DELEGATE_ENTRY(
      MEMBER,
      ARG0,
      ARG1
);
```

Parameters

MEMBER

The event handler method to be attached to the control.

ARG0

The first argument of the managed event handler method, such as **Object^**.

ARG1

The second argument of the managed event handler method, such as **EventArgs^**.

Remarks

Each entry in the delegate map corresponds to a managed event handler delegate created by MAKE_DELEGATE.

Example

The following code example shows how to use **EVENT_DELEGATE_ENTRY** to create an entry in the delegate map for the **OnClick** event handler; also see the code example in **MAKE_DELEGATE**. For more information, see How to: Sink Windows Forms Events from Native C++ Classes.

```
BEGIN_DELEGATE_MAP( CMyDialog )
   EVENT_DELEGATE_ENTRY( OnClick, Object^, EventArgs^ )
END_DELEGATE_MAP()
```

Requirements

Header: msclr\event.h

See Also

Reference

MAKE_DELEGATE
BEGIN_DELEGATE_MAP
END_DELEGATE_MAP

HashKey

Calculates a hash value for the given key.

```
template<class ARG_KEY>
AFX_INLINE UINT AFXAPI HashKey(
      ARG_KEY key
);
```

Parameters

ARG_KEY

Template parameter specifying the data type used to access map keys.

key

The key whose hash value is to be calculated.

Return Value

The key's hash value.

Remarks

This function is called directly by CMap::RemoveKey and indirectly by CMap::Lookup and CMap::Operator [].

The default implementation creates a hash value by shifting *key* right by four positions. Override this function so that it returns hash values appropriate for your application.

Example

```
template <> UINT AFXAPI HashKey( unsigned __int64 key )
{
   // Generate the hash value by XORing the lower 32 bits of the number
   // with the upper 32 bits
   return( UINT( key )^UINT( key>>32 ) );
}
```

See Also

Reference

CMap Class

Concepts

MFC Macros and Globals

IMPLEMENT_DYNAMIC

Generates the C++ code necessary for a dynamic **CObject**-derived class with run-time access to the class name and position within the hierarchy.

```
IMPLEMENT_DYNAMIC(class_name, base_class_name )
```

Parameters

class_name

The actual name of the class.

base_class_name

The name of the base class.

Remarks

Use the **IMPLEMENT_DYNAMIC** macro in a .cpp module, and then link the resulting object code only once.

For more information, see CObject Class Topics.

Example

```
/* CAge.h */
class CAge : public CObject
{
     int num;
public:
     DECLARE_DYNAMIC(CAge)
};

//==============
/* CAge.cpp */
#include "stdafx.h"
#include "CAge.h"

IMPLEMENT_DYNAMIC(CAge, CObject)
```

See Also

Reference

DECLARE_DYNAMIC
RUNTIME_CLASS
CObject::IsKindOf

Concepts

MFC Macros and Globals

IMPLEMENT_DYNCREATE

Enables objects of **CObject**-derived classes to be created dynamically at run time when used with the **DECLARE_DYNCREATE** macro.

```
IMPLEMENT_DYNCREATE(class_name, base_class_name )
```

Parameters

class_name

The actual name of the class.

base_class_name

The actual name of the base class.

Remarks

The framework uses this ability to create new objects dynamically, for example, when it reads an object from disk during serialization. Add the **IMPLEMENT_DYNCREATE** macro in the class implementation file. For more information, see CObject Class Topics.

If you use the **DECLARE_DYNCREATE** and **IMPLEMENT_DYNCREATE** macros, you can then use the **RUNTIME_CLASS** macro and the **CObject::IsKindOf** member function to determine the class of your objects at run time.

If **DECLARE_DYNCREATE** is included in the class declaration, then **IMPLEMENT_DYNCREATE** must be included in the class implementation.

Note that this macro definition will invoke the default constructor for your class. If a non-trivial constructor is explicitly implemented by the class, it must also explicitly implement the default constructor as well. The default constructor can be added to the class's **private** or **protected** member sections to prevent it from being called from outside the class implementation.

Example

```
/* CAge.h */
class CAge : public CObject
{
     int m_Age;
public:
     DECLARE_DYNCREATE(CAge)
     CAge( int Age ) { m_Age = Age; }
private:
     CAge() { m_Age = 0; } // provide default constructor only for
                           // dynamic creation
};

//==============
/* CAge.cpp */
#include "stdafx.h"
#include "CAge.h"

IMPLEMENT_DYNCREATE(CAge, CObject)
```

See Also

Reference

DECLARE_DYNCREATE
RUNTIME_CLASS
CObject::IsKindOf

Concepts

MFC Macros and Globals

IMPLEMENT_OLECREATE

Either this macro or IMPLEMENT_OLECREATE_FLAGS must appear in the implementation file for any class that uses **DECLARE_OLECREATE**.

```
IMPLEMENT_OLECREATE(class_name, external_name, l, w1, w2, b1, b2, b3, b4, b5, b6, b7, b8 )
```

Parameters

class_name

The actual name of the class.

external_name

The object name exposed to other applications (enclosed in quotation marks).

l, *w1*, *w2*, *b1*, *b2*, *b3*, *b4*, *b5*, *b6*, *b7*, *b8*

Components of the class's **CLSID**.

Remarks

Note

If you use **IMPLEMENT_OLECREATE**, by default, you support only the single threading model. If you use **IMPLEMENT_OLECREATE_FLAGS**, you can specify which threading model your object supports by using the *nFlags* parameter.

The external name is the identifier exposed to other applications. Client applications use the external name to request an object of this class from an automation server.

The OLE class ID is a unique 128-bit identifier for the object. It consists of one **long**, two **WORD**s, and eight **BYTE**s, as represented by *l*, *w1*, *w2*, and *b1* through *b8* in the syntax description. The Application Wizard and code wizards create unique OLE class IDs for you as required.

Requirements

Header: afxdisp.h

See Also

Reference

DECLARE_OLECREATE

CLSID Key

Concepts

MFC Macros and Globals

IMPLEMENT_OLECREATE_EX

Implements your control's class factory and the GetClassID member function of your control class.

```
IMPLEMENT_OLECREATE_EX(class_name, external_name, l, w1, w2, b1, b2, b3, b4, b5, b6, b7, b8
)
```

Parameters

class_name

The name of the control property page class.

external_name

The object name exposed to applications.

l, w1, w2, b1, b2, b3, b4, b5, b6, b7, b8

Components of the class's **CLSID**. For more information on these parameters, see the Remarks for IMPLEMENT_OLECREATE.

Remarks

This macro must appear in the implementation file for any control class that uses the **DECLARE_OLECREATE_EX** macro or the **BEGIN_OLEFACTORY** and **END_OLEFACTORY** macros. The external name is the identifier of the OLE control that is exposed to other applications. Containers use this name to request an object of this control class.

Requirements

Header: afxdisp.h

See Also

Reference

DECLARE_OLECREATE_EX
BEGIN_OLEFACTORY
END_OLEFACTORY
IMPLEMENT_OLECREATE

Concepts

MFC Macros and Globals

IMPLEMENT_OLECREATE_FLAGS

Either this macro or IMPLEMENT_OLECREATE must appear in the implementation file for any class that uses **DECLARE_OLECREATE**.

```
IMPLEMENT_OLECREATE_FLAGS(class_name, external_name, nFlags, l, w1, w2, b1, b2, b3, b4, b5,
b6, b7, b8 )
```

Parameters

class_name

The actual name of the class.

external_name

The object name exposed to other applications (enclosed in quotation marks).

nFlags

Contains one or more of the following flags:

- **afxRegInsertable** Allows the control to appear in the Insert Object dialog box for OLE objects.
- **afxRegApartmentThreading** Sets the threading model in the registry to ThreadingModel=Apartment.
- **afxRegFreeThreading** Sets the threading model in the registry to ThreadingModel=Free.

  You can combine the two flags **afxRegApartmentThreading** and **afxRegFreeThreading** to set ThreadingModel=Both. See InprocServer32 in the *Platform SDK* for more information on threading model registration.

l, *w1*, *w2*, *b1*, *b2*, *b3*, *b4*, *b5*, *b6*, *b7*, *b8*

Components of the class's **CLSID**.

Remarks

Note
If you use **IMPLEMENT_OLECREATE_FLAGS**, you can specify which threading model your object supports by using the *nFlags* parameter. If you want to support only the single-treading model, use **IMPLEMENT_OLECREATE**.

The external name is the identifier exposed to other applications. Client applications use the external name to request an object of this class from an automation server.

The OLE class ID is a unique 128-bit identifier for the object. It consists of one **long**, two **WORD**s, and eight **BYTE**s, as represented by *l*, *w1*, *w2*, and *b1* through *b8* in the syntax description. The Application Wizard and code wizards create unique OLE class IDs for you as required.

Requirements

Header: afxdisp.h

See Also

Reference

DECLARE_OLECREATE

CLSID Key

Concepts

MFC Macros and Globals

IMPLEMENT_OLECTLTYPE

Implements the **GetUserTypeNameID** and **GetMiscStatus** member functions of your control class.

```
DECLARE_OLECTLTYPE(class_name, idsUserTypeName, dwOleMisc )
```

Parameters

class_name

The name of the control class.

idsUserTypeName

The resource ID of a string containing the external name of the control.

dwOleMisc

An enumeration containing one or more flags. For more information on this enumeration, see OLEMISC in the *Platform SDK*.

Remarks

In addition to **IMPLEMENT_OLECTLTYPE**, you must add the **DECLARE_OLECTLTYPE** macro to your control class declaration.

The **GetUserTypeNameID** member function returns the resource string that identifies your control class. **GetMiscStatus** returns the **OLEMISC** bits for your control. This enumeration specifies a collection of settings describing miscellaneous characteristics of your control. For a full description of the **OLEMISC** settings, see OLEMISC in the *Platform SDK*.

Note
The default settings used by the ActiveX ControlWizard are: **OLEMISC_ACTIVATEWHENVISIBLE**, **OLEMISC_SETCLIENTSITEFIRST**, **OLEMISC_INSIDEOUT**, **OLEMISC_CANTLINKINSIDE**, and **OLEMISC_RECOMPOSEONRESIZE**.

See Also

Reference

DECLARE_OLECTLTYPE

Concepts

MFC Macros and Globals

IMPLEMENT_OLETYPELIB

Implements the control's **GetTypeLib** member function.

```
IMPLEMENT_OLETYPELIB(class_name, tlid, wVerMajor, wVerMinor )
```

Parameters

class_name

The name of the control class related to the type library.

tlid

The ID number of the type library.

wVerMajor

The type library major version number.

wVerMinor

The type library minor version number.

Remarks

This macro must appear in the implementation file for any control class that uses the **DECLARE_OLETYPELIB** macro.

See Also

Reference

DECLARE_OLETYPELIB

Concepts

MFC Macros and Globals

IMPLEMENT_SERIAL

Generates the C++ code necessary for a dynamic **CObject**-derived class with run-time access to the class name and position within the hierarchy.

```
IMPLEMENT_SERIAL(class_name, base_class_name, wSchema )
```

Parameters

class_name

The actual name of the class.

base_class_name

The name of the base class.

wSchema

A **UINT** "version number" that will be encoded in the archive to enable a deserializing program to identify and handle data created by earlier program versions. The class schema number must not be –1.

Remarks

Use the **IMPLEMENT_SERIAL** macro in a .cpp module; then link the resulting object code only once.

You can use the **AFX_API** macro to automatically export the **CArchive** extraction operator for classes that use the **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** macros. Bracket the class declarations (located in the .h file) with the following code:

```
#undef AFX_API
#define AFX_API AFX_EXT_CLASS

<your class declarations here>

#undef AFX_API
#define AFX_API
```

For more information, see the CObject Class Topics.

Example

```
/* MyClass.cpp */
#include "stdafx.h"
#include "MyClass.h"

IMPLEMENT_SERIAL( CMyClass, CObject, VERSIONABLE_SCHEMA | 2 )

...
```

See Also

Reference

DECLARE_SERIAL
RUNTIME_CLASS
CObject::IsKindOf

Concepts

MFC Macros and Globals

INTERFACE_PART

Used between the **BEGIN_INTERFACE_MAP** macro and the **END_INTERFACE_MAP** macro for each interface your object will support.

```
INTERFACE_PART(theClass, iid, localClass)
```

Parameters

theClass

The name of the class that contains the interface map.

iid

The IID that is to be mapped to the embedded class.

localClass

The name of the local class.

Remarks

It allows you to map an IID to a member of the class indicated by *theClass* and *localClass*.

For more information on interface maps, see Technical Note 38.

See Also

Reference

BEGIN_INTERFACE_MAP

END_MESSAGE_MAP

Concepts

MFC Macros and Globals

ISAPIASSERT

Works exactly like the MFC macro ASSERT.

```
ISAPIASSERT(
      booleanExpression
)
```

Parameters

booleanExpression

Specifies an expression (including pointer values) that evaluates to nonzero or 0.

Remarks

Evaluates its argument. If the result is 0, the macro prints a diagnostic message and aborts the program. If the condition is nonzero, it does nothing.

The diagnostic message has the form

```
assertion failed in file <name> in line <num>
```

where *name* is the name of the source file, and *num* is the line number of the assertion that failed in the source file.

In the release version of your application, **ISAPIASSERT** does not evaluate the expression and thus will not interrupt the program. If the expression must be evaluated regardless of environment, use the **ISAPIVERIFY** macro in place of **ISAPIASSERT**. **ISAPIASSERT** is available only in the debug version of your application.

ISAPI applications do not have to use MFC. If MFC is not linked to your application, **ISAPIASSERT** provides the same **ASSERT** functionality. If your application is linked to the MFC, **ISAPIASSERT** simply calls MFC's **ASSERT**.

In an MFC ISAPI application, an assertion in debug mode will bring up a modal dialog box (ASSERT dialog boxes are now modal by default); this will interrupt or hang the execution. To suppress modal assertion dialogs, add the following lines to your project source file (*projectname*.cpp):

```
// For custom assert and trace handling with WebDbg
#ifdef _DEBUG
CDebugReportHook g_ReportHook;
#endif
```

Once you have done this, you can use the WebDbg tool (WebDbg.exe) to see the assertions. For information on using the WebDbg tool, see Viewing Trace Messages And Handling Asserts.

See Also

Reference

ISAPITRACE
ISAPITRACE0
ISAPITRACE1
ISAPITRACE2
ISAPITRACE3
ISAPIVERIFY

Concepts

MFC Macros and Globals

ISAPITRACE

Works exactly like the MFC macro TRACE, which itself provides functionality similar to the **printf** function by sending a formatted string to a dump device such as debug monitor.

```
ISAPITRACE(exp )
```

Parameters

exp

Specifies a variable number of arguments that are used in exactly the same way that a variable number of arguments are used in the run-time function **printf.**

Remarks

Like **printf** for C programs under MS-DOS, the **ISAPITRACE** macro is a convenient way to track the value of variables as your program executes. In the Debug environment, the **ISAPITRACE** macro output goes to the Debug window of Visual C++. In the Release environment, it does nothing.

ISAPI applications do not have to use MFC. If MFC is not linked to your application, **ISAPITRACE** provides the same **TRACE** functionality. If your application is linked to the MFC, **ISAPITRACE** simply calls MFC's **TRACE**.

See Also

Reference

ISAPIASSERT
ISAPITRACE0
ISAPITRACE1
ISAPITRACE2
ISAPITRACE3
ISAPIVERIFY

Concepts

MFC Macros and Globals

ISAPITRACE0

Variant of a group of trace macros that you can use for debug output.

```
ISAPITRACE0(exp )
```

Parameters

exp

A format string as used in the run-time function **printf**.

Remarks

This group includes **ISAPITRACE0**, **ISAPITRACE1**, **ISAPITRACE2**, and **ISAPITRACE3**. The difference between these macros is the number of parameters taken. **ISAPITRACE0** only takes a format string and can be used for simple text messages. **ISAPITRACE1** takes a format string plus one argument — a variable to be dumped. Likewise, **ISAPITRACE2** and **ISAPITRACE3** take two and three parameters after the format string, respectively.

ISAPITRACE0 does nothing if you have compiled a release version of your application. As with **ISAPITRACE**, it only dumps data to the debug output device if you have compiled a debug version of your application.

ISAPITRACE0 works exactly like the MFC macro TRACE0. ISAPI applications do not have to use MFC. If MFC is not linked to your application, **ISAPITRACE0** provides the same **TRACE0** functionality. If your application is linked to the MFC, **ISAPITRACE0** simply calls MFC's **TRACE0**.

See Also

Reference

ISAPIASSERT
ISAPITRACE
ISAPITRACE1
ISAPITRACE2
ISAPITRACE3
ISAPIVERIFY

Concepts

MFC Macros and Globals

ISAPITRACE1

Works exactly like the MFC macro TRACE1.

```
ISAPITRACE1(exp, param1 )
```

Parameters

exp

A format string as used in the run-time function **printf**.

param1

The name of the variable whose value should be dumped.

Remarks

See ISAPITRACE0 for a description of **ISAPITRACE1**.

ISAPI applications do not have to use MFC. If MFC is not linked to your application, **ISAPITRACE1** provides the same **TRACE1** functionality. If your application is linked to the MFC, **ISAPITRACE1** simply calls MFC's **TRACE1**.

See Also

Reference

ISAPIASSERT
ISAPITRACE
ISAPITRACE0
ISAPITRACE2
ISAPITRACE3
ISAPIVERIFY

Concepts

MFC Macros and Globals

ISAPITRACE2

Works exactly like the MFC macro TRACE2.

```
ISAPITRACE2(exp, param1, param2 )
```

Parameters

exp

A format string as used in the run-time function **printf**.

param1, *param2*

The name of the variable whose value should be dumped.

Remarks

See ISAPITRACE0 for a description of **ISAPITRACE2**.

ISAPI applications do not have to use MFC. If MFC is not linked to your application, **ISAPITRACE2** provides the same **TRACE2** functionality. If your application is linked to the MFC, **ISAPITRACE2** simply calls MFC's **TRACE2**.

See Also

Reference

ISAPIASSERT
ISAPITRACE
ISAPITRACE0
ISAPITRACE1
ISAPITRACE3
ISAPIVERIFY

Concepts

MFC Macros and Globals

ISAPITRACE3

Works exactly like the MFC macro TRACE3.

```
ISAPITRACE3(exp, param1, param2, param3 )
```

Parameters

exp

A format string as used in the run-time function **printf**.

param1, *param2*, *param3*

The name of the variable whose value should be dumped.

Remarks

See ISAPITRACE0 for a description of **ISAPITRACE3**.

ISAPI applications do not have to use MFC. If MFC is not linked to your application, **ISAPITRACE3** provides the same **TRACE3** functionality. If your application is linked to the MFC, **ISAPITRACE3** simply calls MFC's **TRACE3**.

See Also

Reference

ISAPIASSERT
ISAPITRACE
ISAPITRACE0
ISAPITRACE1
ISAPITRACE2
ISAPIVERIFY

Concepts

MFC Macros and Globals

ISAPIVERIFY

Works exactly like the MFC macro VERIFY.

```
ISAPIVERIFY(booleanExpression )
```

Parameters

booleanExpression

Specifies an expression (including pointer values) that evaluates to nonzero or 0.

Remarks

In the debug version of your application, the **ISAPIVERIFY** macro evaluates its argument. If the result is 0, the macro prints a diagnostic message and halts the program. If the condition is nonzero, it does nothing.

The diagnostic message has the form

```
assertion failed in file <name> in line <num>
```

where *name* is the name of the source file and *num* is the line number of the assertion that failed in the source file.

In the release version of your application, **ISAPIVERIFY** evaluates the expression but does not print or interrupt the program. For example, if the expression is a function call, the call will be made.

ISAPI applications do not have to use MFC. If MFC is not linked to your application, **ISAPIVERIFY** provides the same **VERIFY** functionality. If your application is linked to the MFC, **ISAPIVERIFY** simply calls MFC's **VERIFY**.

See Also

Reference

ISAPIASSERT
ISAPITRACE
ISAPITRACE0
ISAPITRACE1
ISAPITRACE2
ISAPITRACE3

Concepts

MFC Macros and Globals

MAKE_DELEGATE

Attaches an event handler to a managed control.

```
MAKE_DELEGATE(
      DELEGATE,
      MEMBER
);
```

Parameters

DELEGATE

The type of the managed event handler delegate, such as EventHandler.

MEMBER

The name of the event handler method to be attached to the control.

Remarks

This macro creates a managed event handler delegate of type *DELEGATE* and of the name *MEMBER*. The managed event handler delegate allows a native class to handle managed events.

Example

The following code example shows how to call **MAKE_DELEGATE** to attach an **OnClick** event handler to an MFC control `MyControl`. For a broader explanation of how this macro works in an MFC application, see How to: Sink Windows Forms Events from Native C++ Classes.

```
CWinFormsControl<MyControl> m_MyControl;
// ...
m_MyControl.GetControl()->Click
   += MAKE_DELEGATE( System::EventHandler, OnClick );
```

Requirements

Header: msclr\event.h

See Also

Reference

BEGIN_DELEGATE_MAP
END_DELEGATE_MAP
EVENT_DELEGATE_ENTRY

METHOD_PROLOGUE

Maintains the proper global state when calling methods of an exported interface.

```
METHOD_PROLOGUE(theClass, localClass )
```

Parameters

theClass

Specifies the name of the class whose interface map is being implemented.

localClass

Specifies the name of the local class that implements the interface map.

Remarks

Typically, member functions of interfaces implemented by **CCmdTarget**-derived objects already use this macro to provide automatic initialization of the *pThis* pointer. For example:

```
class CInnerUnknown : public IUnknown
   ...
   CInnerUnknown InnerUnknown;
   ...
// Inner IUnknown implementation

   STDMETHODIMP_(ULONG) CInnerUnknown::AddRef()
   {
   METHOD_PROLOGUE(CCmdTarget, InnerUnknown)
   return pThis->InternalAddRef();
   }
```

See Also

Concepts

MFC Macros and Globals
TN038: MFC/OLE IUnknown Implementation
Creating New Documents, Windows, and Views

ON_COMMAND

This macro maps a command message to a member function.

```
ON_COMMAND(id, memberFxn )
```

Parameters

id

The command ID.

memberFxn

The name of the message-handler function to which the command is mapped.

Remarks

It indicates which function will handle a command message from a command user-interface object such as a menu item or toolbar button.

When a command-target object receives a Windows **WM_COMMAND** message with the specified ID, **ON_COMMAND** will call the member function *memberFxn* to handle the message.

Use **ON_COMMAND** to map a single command to a member function. Use ON_COMMAND_RANGE to map a range of command ids to one member function. Only one message-map entry can match a given command id. That is, you can't map a command to more than one handler. For more information and examples, see Message Handling and Mapping Topics.

Example

```
// example for ON_COMMAND
BEGIN_MESSAGE_MAP( CMyDoc, CDocument )
    ON_COMMAND( ID_MYCMD, OnMyCommand )
    // ... More entries to handle additional commands
END_MESSAGE_MAP( )
```

See Also

Reference

ON_UPDATE_COMMAND_UI

Concepts

MFC Macros and Globals

ON_COMMAND_EX

This macro maps a command message to an extended command-handler member function.

```
ON_COMMAND_EX(
      id,
      memberFxn
);
```

Parameters

id

The command ID.

memberFxn

The name of the message-handler function to which the command is mapped.

Remarks

An extended form of command message handlers is available for advanced uses. The **ON_COMMAND_EX** macro is used for such message handlers, and it provides a superset of the ON_COMMAND functionality. Extended command-handler member functions take a single parameter, a **UINT** containing the command ID, and return a **BOOL**. The return value should be TRUE to indicate that the command has been handled; otherwise routing will continue to other command target objects.

For more information, see Technical Note TN006: Message Maps.

Requirements

Header file: afxmsg_.h

See Also

Reference

ON_COMMAND

TN006: Message Maps

ON_COMMAND_RANGE

Use this macro to map a contiguous range of command IDs to a single message handler function.

```
ON_COMMAND_RANGE(id1, id2, memberFxn )
```

Parameters

id1

Command ID at the beginning of a contiguous range of command IDs.

id2

Command ID at the end of a contiguous range of command IDs.

memberFxn

The name of the message-handler function to which the commands are mapped.

Remarks

The range of IDs starts with *id1* and ends with *id2*.

Use **ON_COMMAND_RANGE** to map a range of command IDs to one member function. Use ON_COMMAND to map a single command to a member function. Only one message-map entry can match a given command ID. That is, you can't map a command to more than one handler. For more information on mapping message ranges, see Handlers for Message-Map Ranges.

There is no automatic support for message map ranges, so you must place the macro yourself.

Example

```
// The code fragment below shows how to use ON_COMMAND_RANGE macro
// to map a contiguous range of command IDs to a single message
// handler function (i.e. OnFileMenuItems() is the sample below). In
// addition, it also shows how to use CheckMenuRadioItem() to check a
// selected menu item and makes it a radio item.
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
   // ...
   ON_COMMAND_RANGE(ID_FILE_MENUITEM1, ID_FILE_MENUITEM3, OnFileMenuItems)
END_MESSAGE_MAP()

void CMainFrame::OnFileMenuItems(UINT nID)
{
   CMenu* mmenu = GetMenu();
   CMenu* submenu = mmenu->GetSubMenu(0);
   submenu->CheckMenuRadioItem(ID_FILE_MENUITEM1, ID_FILE_MENUITEM3,
      nID, MF_BYCOMMAND);
}
```

See Also

Reference

ON_UPDATE_COMMAND_UI_RANGE
ON_CONTROL_RANGE
ON_COMMAND

Concepts

MFC Macros and Globals

ON_CONTROL

Indicates which function will handle a custom-control notification message.

```
ON_CONTROL(wNotifyCode, id, memberFxn )
```

Parameters

wNotifyCode

The notification code of the control.

id

The command ID.

memberFxn

The name of the message-handler function to which the command is mapped.

Remarks

Control notification messages are those sent from a control to its parent window.

There should be exactly one **ON_CONTROL** macro statement in your message map for every control notification message that must be mapped to a message-handler function.

For more information and examples, see Message Handling and Mapping Topics.

See Also

Reference

ON_MESSAGE

ON_REGISTERED_MESSAGE

Concepts

MFC Macros and Globals

ON_CONTROL_RANGE

Use this macro to map a contiguous range of control IDs to a single message handler function for a specified Windows notification message, such as **BN_CLICKED**.

```
ON_CONTROL_RANGE(wNotifyCode, id1, id2, memberFxn )
```

Parameters

wNotifyCode

The notification code to which your handler is responding.

id1

Command ID at the beginning of a contiguous range of control IDs.

id2

Command ID at the end of a contiguous range of control IDs.

memberFxn

The name of the message-handler function to which the controls are mapped.

Remarks

The range of IDs starts with *id1* and ends with *id2*. The handler is called for the specified notification coming from any of the mapped controls.

There is no automatic support for message map ranges, so you must place the macro yourself.

For more information on implementing handler functions for a range of control IDs, refer to Handlers for Message-Map Ranges.

See Also

Reference

ON_UPDATE_COMMAND_UI_RANGE
ON_COMMAND_RANGE

Concepts

MFC Macros and Globals

ON_EVENT

Use the **ON_EVENT** macro to define an event handler function for an event fired by an OLE control.

```
ON_EVENT(theClass, id, dispid, pfnHandler, vtsParams )
```

Parameters

theClass

The class to which this event sink map belongs.

id

The control ID of the OLE control.

dispid

The dispatch ID of the event fired by the control.

pfnHandler

Pointer to a member function that handles the event. This function should have a **BOOL** return type, and parameter types that match the event's parameters (see *vtsParams*). The function should return **TRUE** to indicate the event was handled; otherwise **FALSE**.

vtsParams

A sequence of **VTS_** constants that specifies the types of the parameters for the event. These are the same constants that are used in dispatch map entries such as **DISP_FUNCTION**.

Remarks

The *vtsParams* argument is a space-separated list of values from the **VTS_** constants. One or more of these values separated by spaces (not commas) specifies the function's parameter list. For example:

```
VTS_I2 VTS_BOOL
```

specifies a list containing a short integer followed by a **BOOL**.

For a list of the **VTS_** constants, see EVENT_CUSTOM.

See Also

Reference

ON_EVENT_RANGE
ON_PROPNOTIFY
ON_PROPNOTIFY_RANGE

Concepts

MFC Macros and Globals

ON_EVENT_RANGE

Use the **ON_EVENT_RANGE** macro to define an event handler function for an event fired by any OLE control having a control ID within a contiguous range of IDs.

```
ON_EVENT_RANGE(theClass, idFirst, idLast, dispid, pfnHandler, vtsParams )
```

Parameters

theClass

The class to which this event sink map belongs.

idFirst

The control ID of the first OLE control in the range.

idLast

The control ID of the last OLE control in the range.

dispid

The dispatch ID of the event fired by the control.

pfnHandler

Pointer to a member function that handles the event. This function should have a **BOOL** return type, a first parameter of type **UINT** (for the control ID), and additional parameter types that match the event's parameters (see *vtsParams*). The function should return **TRUE** to indicate the event was handled; otherwise **FALSE**.

vtsParams

A sequence of **VTS_** constants that specifies the types of the parameters for the event. The first constant should be of type **VTS_I4**, for the control ID. These are the same constants that are used in dispatch map entries such as **DISP_FUNCTION**.

Remarks

The *vtsParams* argument is a space-separated list of values from the **VTS_** constants. One or more of these values separated by spaces (not commas) specifies the function's parameter list. For example:

```
VTS_I2 VTS_BOOL
```

specifies a list containing a short integer followed by a **BOOL**.

For a list of the **VTS_** constants, see EVENT_CUSTOM.

Example

The following example demonstrates an event handler, for the MouseDown event, implemented for three controls (`IDC_MYCTRL1` through `IDC_MYCTRL3`). The event handler function, `OnRangeMouseDown`, is declared in the header file of the dialog class (`CMyDlg`) as:

```
BOOL OnRangeMouseDown(UINT CtlID, short MouseButton, short Shift,
    long x, long y);
```

The code below is defined in the implementation file of the dialog class.

```
// example for ON_EVENT_RANGE
BEGIN_EVENTSINK_MAP(CMyDlg, CDialog)
ON_EVENT_RANGE(CMyDlg, IDC_MYCTRL1, IDC_MYCTRL3, -605, OnRangeMouseDown,
                VTS_I4 VTS_I2 VTS_I2 VTS_I4 VTS_I4)
END_EVENTSINK_MAP()
```

See Also

Reference

ON_EVENT

ON_PROPNOTIFY

ON_PROPNOTIFY_RANGE

Concepts

MFC Macros and Globals

ON_EVENT_REFLECT

The **ON_EVENT_REFLECT** macro, when used in the event sink map of an OLE control's wrapper class, receives events fired by the control before they are handled by the control's container.

```
ON_EVENT_REFLECT(theClass, dispid, pfnHandler, vtsParams )
```

Parameters

theClass

The class to which this event sink map belongs.

dispid

The dispatch ID of the event fired by the control.

pfnHandler

Pointer to a member function that handles the event. This function should have a **BOOL** return type and parameter types that match the event's parameters (see *vtsParams*). The function should return **TRUE** to indicate the event was handled; otherwise **FALSE**.

vtsParams

A sequence of **VTS_** constants that specifies the types of the parameters for the event. These are the same constants that are used in dispatch map entries such as **DISP_FUNCTION**.

Remarks

The *vtsParams* argument is a space-separated list of values from the **VTS_** constants.

One or more of these values separated by spaces (not commas) specifies the function's parameter list. For example:

```
VTS_I2 VTS_BOOL
```

specifies a list containing a short integer followed by a **BOOL**.

For a list of the **VTS_** constants, see EVENT_CUSTOM.

See Also

Reference

ON_EVENT
ON_PROPNOTIFY
ON_PROPNOTIFY_REFLECT

Concepts

MFC Macros and Globals

ON_MESSAGE

Indicates which function will handle a user-defined message.

```
ON_MESSAGE(message, memberFxn )
```

Parameters

message

The message ID.

memberFxn

The name of the message-handler function to which the message is mapped.

The type of the function must be `afx_msg LRESULT (CWnd::*)(WPARAM, LPARAM)`.

Remarks

User-defined messages are usually defined in the range **WM_USER** to 0x7FFF. User-defined messages are any messages that are not standard Windows **WM_MESSAGE** messages. There should be exactly one **ON_MESSAGE** macro statement in your message map for every user-defined message that must be mapped to a message-handler function.

Note

In addition to user-defined messages, **ON_MESSAGE** handles less common Windows messages. For more information, see the Knowledge Base article Q99848.

For more information and examples, see Message Handling and Mapping Topics and User-Defined Handlers

Example

```
// example for ON_MESSAGE
#define WM_MYMESSAGE (WM_USER + 1)
BEGIN_MESSAGE_MAP( CMyWnd, CMyParentWndClass )
    ON_MESSAGE( WM_MYMESSAGE, OnMyMessage )
    // ... Possibly more entries to handle additional messages
END_MESSAGE_MAP( )
```

```
// implementation for WM_MYMESSAGE handler
LRESULT CMyWnd::OnMyMessage( WPARAM wParam, LPARAM lParam )
{
   // ... Handle message here
   return 0L;
}
```

See Also

Reference

ON_UPDATE_COMMAND_UI
ON_CONTROL
ON_REGISTERED_MESSAGE
ON_COMMAND

Concepts

MFC Macros and Globals

ON_OLECMD

Routes commands through the command dispatch interface **IOleCommandTarget**.

```
ON_OLECMD(pguid, olecmdid, id )
```

Parameters

pguid

Identifier of the command group to which the command belongs. Use **NULL** for the standard group.

olecmdid

The identifier of the OLE command.

id

The menu ID, toolbar ID, button ID, or other ID of the resource or object issuing the command.

Remarks

IOleCommandTarget allows a container to receive commands that originate in a DocObject's user interface, and allows the container to send the same commands (such as New, Open, SaveAs, and Print on the File menu; and Copy, Paste, Undo, and so forth on the Edit menu) to a DocObject.

IOleCommandTarget is simpler than OLE Automation's **IDispatch**. **IOleCommandTarget** relies entirely on a standard set of commands that rarely have arguments, and no type information is involved (type safety is diminished for command arguments as well). If you do need to dispatch commands with arguments, use COleServerDoc::OnExecOleCmd.

The **IOleCommandTarget** standard menu commands have been implemented by MFC in the following macros:

ON_OLECMD_CLEARSELECTION( )

Dispatches the Edit Clear command. Implemented as:

```
ON_OLECMD(NULL, OLECMDID_CLEARSELECTION, ID_EDIT_CLEAR)
```

ON_OLECMD_COPY( )

Dispatches the Edit Copy command. Implemented as:

```
ON_OLECMD(NULL, OLECMDID_COPY, ID_EDIT_COPY)
```

ON_OLECMD_CUT( )

Dispatches the Edit Cut command. Implemented as:

```
ON_OLECMD(NULL, OLECMDID_CUT, ID_EDIT_CUT)
```

ON_OLECMD_NEW( )

Dispatches the File New command. Implemented as:

```
ON_OLECMD(NULL, OLECMDID_NEW, ID_FILE_NEW)
```

ON_OLECMD_OPEN( )

Dispatches the File Open command. Implemented as:

```
ON_OLECMD(NULL, OLECMDID_OPEN, ID_FILE_OPEN)
```

ON_OLECMD_PAGESETUP( )

Dispatches the File Page Setup command. Implemented as:

```
ON_OLECMD(NULL, OLECMDID_PAGESETUP, ID_FILE_PAGE_SETUP)
```

ON_OLECMD_PASTE( )

Dispatches the Edit Paste command. Implemented as:

```
ON_OLECMD(NULL, OLECMDID_PASTE, ID_EDIT_PASTE)
```

ON_OLECMD_PASTESPECIAL( )

Dispatches the Edit Paste Special command. Implemented as:

```
ON_OLECMD(NULL, OLECMDID_PASTESPECIAL, ID_EDIT_PASTE_SPECIAL)
```

ON_OLECMD_PRINT( )

Dispatches the File Print command. Implemented as:

```
ON_OLECMD(NULL, OLECMDID_PRINT, ID_FILE_PRINT)
```

ON_OLECMD_PRINTPREVIEW( )

Dispatches the File Print Preview command. Implemented as:

```
ON_OLECMD(NULL, OLECMDID_PRINTPREVIEW, ID_FILE_PRINT_PREVIEW)
```

ON_OLECMD_REDO( )

Dispatches the Edit Redo command. Implemented as:

```
ON_OLECMD(NULL, OLECMDID_REDO, ID_EDIT_REDO)
```

ON_OLECMD_SAVE( )

Dispatches the File Save command. Implemented as:

```
ON_OLECMD(NULL, OLECMDID_SAVE, ID_FILE_SAVE)
```

ON_OLECMD_SAVE_AS( )

Dispatches the File Save As command. Implemented as:

```
ON_OLECMD(NULL, OLECMDID_SAVEAS, ID_FILE_SAVE_AS)
```

ON_OLECMD_SAVE_COPY_AS( )

Dispatches the File Save Copy As command. Implemented as:

```
ON_OLECMD(NULL, OLECMDID_SAVECOPYAS, ID_FILE_SAVE_COPY_AS)
```

ON_OLECMD_SELECTALL( )

Dispatches the Edit Select All command. Implemented as:

```
ON_OLECMD(NULL, OLECMDID_SELECTALL, ID_EDIT_SELECT_ALL)
```

ON_OLECMD_UNDO( )

Dispatches the Edit Undo command. Implemented as:

```
ON_OLECMD(NULL, OLECMDID_UNDO, ID_EDIT_UNDO)
```

See Also

Reference

COleCmdUI Class

COleServerDoc::OnExecOleCmd

Concepts

MFC Macros and Globals

ON_OLEVERB

This macro defines a message map entry that maps a custom verb to a specific member function of your control.

```
ON_OLEVERB(idsVerbName, memberFxn )
```

Parameters

idsVerbName

The string resource ID of the verb's name.

memberFxn

The function called by the framework when the verb is invoked.

Remarks

The resource editor can be used to create custom verb names that are added to your string table.

The function prototype for *memberFxn* is:

```
        BOOL memberFxn(
   LPMSG lpMsg,
   HWND hWndParent,
   LPCRECT lpRect
);
```

The values of the *lpMsg*, *hWndParent*, and *lpRect* parameters are taken from the corresponding parameters of the **IOleObject::DoVerb** member function.

See Also

Reference

ON_STDOLEVERB

Concepts

MFC Macros and Globals

ON_PARSE_COMMAND

The **ON_PARSE_COMMAND** macro is used in a parse map to define a command to a CHttpServer object from a client.

```
ON_PARSE_COMMAND(FnName, mapClass, Args )
```

Parameters

FnName

The name of the member function. Also the name of the command.

mapClass

The class name to map the function to.

Args

The arguments that map to the parameter's *FnName*. See Remarks for a list of symbols.

Remarks

The member function identified by *FnName* must take a pointer to the CHttpServerContext as its first parameter. *FnName* is of the type **LPSTR**, and is identified by the symbol **ITS_LPSTR** in the parse map; that is, *FnName* points to a string containing the member function in class *mapClass*.

The parameter *Args* can take one of the following values:

Symbol	Type or Comment
ITS_EMPTY	*Args* cannot be blank. Use **ITS_EMPTY** if you have no arguments.
ITS_PSTR	A pointer to a string.
ITS_RAW	The exact, raw data sent to the ISAPI extension. Do not use **ITS_RAW** with any other parameter type; to do so will ca use an **ASSERT**. See Remarks for an example.
ITS_I2	a short
ITS_I4	a long
ITS_R4	a float
ITS_R8	a double
ITS_I8	a 64-bit integer
ITS_ARGLIST	A pointer to an CHttpArgList. Do not use **ITS_ARGLIST** with any other parameter type; to do so will cause an **ASSERT**.

Examples

```
// The following  example Illustrates extracting
//  a string and a short sent to the server:

BEGIN_PARSE_MAP(CDerivedClass, CHttpServer)
   DEFAULT_PARSE_COMMAND(Myfunc, CDerivedClass)
   ON_PARSE_COMMAND(Myfunc, CDerivedClass, ITS_PSTR
    ITS_I2)
   ON_PARSE_COMMAND_PARAMS("string integer=42")
   ON_PARSE_COMMAND(Myfunc2, CDerivedClass, ITS_PSTR
```

```
     ITS_I2 ITS_PSTR)
    ON_PARSE_COMMAND_PARAMS("string integer
     string2='Default value'")
END_PARSE_MAP(CDerivedClass)
```

Note

Use single quotes if you incorporate spaces into the default values for optional **ITS_PSTR**s.

```
void Myfunc(CHttpServerContext* pCtxt, LPTSTR pszName, int nNumber);
void Myfunc2(CHttpServerContext* pCtxt, LPTSTR pszName, int nNumber, LPTSTR pszTitle);


// The following  example Illustrates extracting
// raw data sent to the server:

BEGIN_PARSE_MAP(CDerivedClass, CHttpServer)
   DEFAULT_PARSE_COMMAND(Myfunc, CDerivedClass)
   ON_PARSE_COMMAND(Myfunc, CDerivedClass, ITS_RAW)
END_PARSE_MAP(CDerivedClass)
```

with the function prototype as follows:

```
void CDerivedClass::Myfunc(CHttpServerContext* pCtxt, void* pVoid, DWORD dwBytes);
```

In the second example above, the `pVoid` pointer points to the data sent to your extension. The `dwBytes` parameter has a count of bytes at `pVoid`. If `dwBytes` is zero, `pVoid` may not point to anything.

Note

The handlers for a parse map command must take a pointer to a **CHttpServerContext** as the first parameter, and the parameters must be declared in the same order in which they're defined in **ON_PARSE_COMMAND**.

See Also

Reference

BEGIN_PARSE_MAP
END_PARSE_MAP
ON_PARSE_COMMAND_PARAMS
DEFAULT_PARSE_COMMAND
CHttpServer Class

Concepts

MFC Macros and Globals

ON_PARSE_COMMAND_PARAMS

The macro **ON_PARSE_COMMAND_PARAMS** identifies and specifies defaults for the parameters associated with the function that is mapped to a command to a **CHttpServer** object by a client.

```
ON_PARSE_COMMAND_PARAMS(Params )
```

Parameters

Params

The parameters, mapped to the *Args* parameter and associated with the function identified by *FnName*, in the macro ON_PARSE_COMMAND immediately preceding **ON_PARSE_COMMAND_PARAMS**.

Remarks

The macro **ON_PARSE_COMMAND_PARAMS** must immediately follow the **ON_PARSE_COMMAND** macro with which it is associated.

If a parameter is named, the client must supply the parameter name in the query. For example, if your parameters are as follows:

```
ON_PARSE_COMMAND_PARAMS("string int=42")
```

then the parameter *string* must be supplied by the client, or the query will fail.

If the parameter is optional, the client need not supply it, and the parse map will supply the default value. For example, if your parameters are as follows:

```
ON_PARSE_COMMAND_PARAMS("string=default int=42")
```

then neither parameter must be defined in the client's query, and the parameter *string* is by default an empty string.

See ON_PARSE_COMMAND for a parse map example.

See Also

Reference

BEGIN_PARSE_MAP
END_PARSE_MAP
ON_PARSE_COMMAND
DEFAULT_PARSE_COMMAND
CHttpServer Class

Concepts

MFC Macros and Globals

ON_PROPNOTIFY

Use the **ON_PROPNOTIFY** macro to define an event sink map entry for handling property notifications from an OLE control.

```
ON_PROPNOTIFY(theClass, id, dispid, pfnRequest, pfnChanged )
```

Parameters

theClass

The class to which this event sink map belongs.

id

The control ID of the OLE control.

dispid

The dispatch ID of the property involved in the notification.

pfnRequest

Pointer to a member function that handles the **OnRequestEdit** notification for this property. This function should have a **BOOL** return type and a **BOOL*** parameter. This function should set the parameter to **TRUE** to allow the property to change and **FALSE** to disallow. The function should return **TRUE** to indicate the notification was handled; otherwise **FALSE**.

pfnChanged

Pointer to a member function that handles the **OnChanged** notification for this property. The function should have a **BOOL** return type and a **UINT** parameter. The function should return **TRUE** to indicate that notification was handled; otherwise **FALSE**.

Remarks

The *vtsParams* argument is a space-separated list of values from the **VTS_** constants. One or more of these values separated by spaces (not commas) specifies the function's parameter list. For example:

```
VTS_I2 VTS_BOOL
```

specifies a list containing a short integer followed by a **BOOL**.

For a list of the **VTS_** constants, see EVENT_CUSTOM.

See Also

Reference

ON_EVENT_RANGE

ON_PROPNOTIFY_RANGE

Concepts

MFC Macros and Globals

ON_PROPNOTIFY_RANGE

Use the **ON_PROPNOTIFY_RANGE** macro to define an event sink map entry for handling property notifications from any OLE control having a control ID within a contiguous range of IDs.

```
ON_PROPNOTIFY_RANGE(theClass, idFirst, idLast, dispid, pfnRequest, pfnChanged )
```

Parameters

theClass

The class to which this event sink map belongs.

idFirst

The control ID of the first OLE control in the range.

idLast

The control ID of the last OLE control in the range.

dispid

The dispatch ID of the property involved in the notification.

pfnRequest

Pointer to a member function that handles the **OnRequestEdit** notification for this property. This function should have a **BOOL** return type and **UINT** and **BOOL*** parameters. The function should set the parameter to **TRUE** to allow the property to change and **FALSE** to disallow. The function should return **TRUE** to indicate that notification was handled; otherwise **FALSE**.

pfnChanged

Pointer to a member function that handles the **OnChanged** notification for this property. The function should have a **BOOL** return type and a **UINT** parameter. The function should return **TRUE** to indicate that notification was handled; otherwise **FALSE**.

See Also

Reference

ON_EVENT_RANGE
ON_PROPNOTIFY
ON_EVENT

Concepts

MFC Macros and Globals

ON_PROPNOTIFY_REFLECT

The **ON_PROPNOTIFY_REFLECT** macro, when used in the event sink map of an OLE control's wrapper class, receives property notifications sent by the control before they are handled by the control's container.

```
ON_PROPNOTIFY_REFLECT(theClass, dispid, pfnRequest, pfnChanged )
```

Parameters

theClass

The class to which this event sink map belongs.

dispid

The dispatch ID of the property involved in the notification.

pfnRequest

Pointer to a member function that handles the **OnRequestEdit** notification for this property. This function should have a **BOOL** return type and a **BOOL*** parameter. This function should set the parameter to **TRUE** to allow the property to change and **FALSE** to disallow. The function should return **TRUE** to indicate the notification was handled; otherwise **FALSE**.

pfnChanged

Pointer to a member function that handles the **OnChanged** notification for this property. The function should have a **BOOL** return type and no parameters. The function should return **TRUE** to indicate the notification was handled; otherwise **FALSE**.

See Also

Reference

ON_EVENT_REFLECT
ON_PROPNOTIFY

Concepts

MFC Macros and Globals

ON_REGISTERED_MESSAGE

The Windows **RegisterWindowMessage** function is used to define a new window message that is guaranteed to be unique throughout the system.

```
ON_REGISTERED_MESSAGE(nMessageVariable, memberFxn )
```

Parameters

nMessageVariable

The registered window-message ID variable.

memberFxn

The name of the message-handler function to which the message is mapped.

Remarks

This macro indicates which function will handle the registered message.

For more information and examples, see Message Handling and Mapping Topics.

Example

```
// example for ON_REGISTERED_MESSAGE
const UINT    wm_Find = RegisterWindowMessage( FINDMSGSTRING );
BEGIN_MESSAGE_MAP( CMyWnd, CMyParentWndClass )
    ON_REGISTERED_MESSAGE( wm_Find, OnFind )
    // ... Possibly more entries to handle additional messages
END_MESSAGE_MAP( )
```

See Also

Reference

ON_MESSAGE
ON_UPDATE_COMMAND_UI
ON_CONTROL
ON_COMMAND
RegisterWindowMessage
User-Defined Handlers

Concepts

MFC Macros and Globals

ON_REGISTERED_THREAD_MESSAGE

Indicates which function will handle the message registered by the Windows **RegisterWindowMessage** function.

```
ON_REGISTERED_THREAD_MESSAGE(nMessageVariable, memberFxn )
```

Parameters

nMessageVariable

The registered window-message ID variable.

memberFxn

The name of the **CWinThread**-message-handler function to which the message is mapped.

Remarks

RegisterWindowMessage is used to define a new window message that is guaranteed to be unique throughout the system. **ON_REGISTERED_THREAD_MESSAGE** must be used instead of **ON_REGISTERED_MESSAGE** when you have a **CWinThread** class.

See Also

Reference

ON_REGISTERED_MESSAGE
ON_THREAD_MESSAGE
RegisterWindowMessage
CWinThread Class

Concepts

MFC Macros and Globals

ON_STDOLEVERB

Use this macro to override the default behavior of a standard verb.

```
ON_STDOLEVERB(iVerb, memberFxn )
```

Parameters

iVerb

The standard verb index for the verb being overridden.

memberFxn

The function called by the framework when the verb is invoked.

Remarks

The standard verb index is of the form **OLEIVERB_**, followed by an action. **OLEIVERB_SHOW**, **OLEIVERB_HIDE**, and **OLEIVERB_UIACTIVATE** are some examples of standard verbs.

See ON_OLEVERB for a description of the function prototype to be used as the *memberFxn* parameter.

See Also

Reference

ON_OLEVERB

Concepts

MFC Macros and Globals

ON_THREAD_MESSAGE

Indicates which function will handle a user-defined message.

```
ON_THREAD_MESSAGE(message, memberFxn )
```

Parameters

message

The message ID.

memberFxn

The name of the **CWinThread**-message-handler function to which the message is mapped.

Remarks

ON_THREAD_MESSAGE must be used instead of **ON_MESSAGE** when you have a **CWinThread** class. User-defined messages are any messages that are not standard Windows **WM_MESSAGE** messages. There should be exactly one **ON_THREAD_MESSAGE** macro statement in your message map for every user-defined message that must be mapped to a message-handler function.

See Also

Reference

ON_MESSAGE
ON_REGISTERED_THREAD_MESSAGE
CWinThread Class

Concepts

MFC Macros and Globals

ON_UPDATE_COMMAND_UI

This macro indicates which function will handle a user-interface update command message.

```
ON_UPDATE_COMMAND_UI(id, memberFxn )
```

Parameters

id

The message ID.

memberFxn

The name of the message-handler function to which the message is mapped.

Remarks

There should be exactly one **ON_UPDATE_COMMAND_UI** macro statement in your message map for every user-interface update command that must be mapped to a message-handler function.

For more information and examples, see Message Handling and Mapping Topics.

See Also

Reference

ON_MESSAGE
ON_REGISTERED_MESSAGE
ON_CONTROL
ON_COMMAND
CCmdUI Class

Concepts

MFC Macros and Globals

ON_UPDATE_COMMAND_UI_RANGE

Maps a contiguous range of command IDs to a single update message handler function.

```
ON_UPDATE_COMMAND_UI_RANGE(id1, id2, memberFxn )
```

Parameters

id1

Command ID at the beginning of a contiguous range of command IDs.

id2

Command ID at the end of a contiguous range of command IDs.

memberFxn

The name of the update message-handler function to which the commands are mapped.

Remarks

Update message handlers update the state of menu items and toolbar buttons associated with the command. The range of IDs starts with *id1* and ends with *id2*.

There is no automatic support for message map ranges, so you must place the macro yourself.

See Also

Reference

ON_COMMAND_RANGE
ON_CONTROL_RANGE

Concepts

MFC Macros and Globals

PROPPAGEID

Adds a property page for use by your OLE control.

```
PROPPAGEID(clsid )
```

Parameters

clsid

The unique class ID of a property page.

Remarks

All **PROPPAGEID** macros must be placed between the **BEGIN_PROPPAGEIDS** and **END_PROPPAGEIDS** macros in your control's implementation file.

See Also

Reference

BEGIN_PROPPAGEIDS

END_PROPPAGEIDS

Concepts

MFC Macros and Globals

PX_Blob

Call this function within your control's **DoPropExchange** member function to serialize or initialize a property that stores binary large object (BLOB) data.

```
BOOL PX_Blob(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   HGLOBAL& hBlob,
   HGLOBAL hBlobDefault = NULL
);
```

Parameters

pPX

Pointer to the CPropExchange object (typically passed as a parameter to **DoPropExchange**).

pszPropName

The name of the property being exchanged.

hBlob

Reference to the variable where the property is stored (typically a member variable of your class).

hBlobDefault

Default value for the property.

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

The property's value will be read from or written to the variable referenced by *hBlob*, as appropriate. This variable should be initialized to **NULL** before initially calling **PX_Blob** for the first time (typically, this can be done in the control's constructor). If *hBlobDefault* is specified, it will be used as the property's default value. This value is used if, for any reason, the control's initialization or serialization process fails.

The handles *hBlob* and *hBlobDefault* refer to a block of memory which contains the following:

- A **DWORD** which contains the length, in bytes, of the binary data that follows, followed immediately by
- A block of memory containing the actual binary data.

Note that **PX_Blob** will allocate memory, using the Windows GlobalAlloc API, when loading BLOB-type properties. You are responsible for freeing this memory. Therefore, the destructor of your control should call GlobalFree on any BLOB-type property handles to free up any memory allocated to your control.

See Also

Reference

COleControl::DoPropExchange

Concepts

MFC Macros and Globals

PX_Bool

Call this function within your control's **DoPropExchange** member function to serialize or initialize a property of type **BOOL**.

```
BOOL PX_Bool(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   BOOL& bValue
);
BOOL PX_Bool(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   BOOL& bValue,
   BOOL bDefault
);
```

Parameters

pPX

Pointer to the CPropExchange object (typically passed as a parameter to **DoPropExchange**).

pszPropName

The name of the property being exchanged.

bValue

Reference to the variable where the property is stored (typically a member variable of your class).

bDefault

Default value for the property.

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

The property's value will be read from or written to the variable referenced by *bValue*, as appropriate. If *bDefault* is specified, it will be used as the property's default value. This value is used if, for any reason, the control's serialization process fails.

See Also

Reference

COleControl::DoPropExchange

Concepts

MFC Macros and Globals

PX_Color

Call this function within your control's **DoPropExchange** member function to serialize or initialize a property of type **OLE_COLOR**.

```
BOOL PX_Color(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   OLE_COLOR& clrValue
);
BOOL PX_Color(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   OLE_COLOR& clrValue,
   OLE_COLOR clrDefault
);
```

Parameters

pPX

Pointer to the CPropExchange object (typically passed as a parameter to **DoPropExchange**).

pszPropName

The name of the property being exchanged.

clrValue

Reference to the variable where the property is stored (typically a member variable of your class).

clrDefault

Default value for the property, as defined by the control developer.

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

The property's value will be read from or written to the variable referenced by *clrValue*, as appropriate. If *clrDefault* is specified, it will be used as the property's default value. This value is used if, for any reason, the control's serialization process fails.

See Also

Reference

COleControl::DoPropExchange

Concepts

MFC Macros and Globals

PX_Currency

Call this function within your control's **DoPropExchange** member function to serialize or initialize a property of type **currency**.

```
BOOL PX_Currency(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   CY& cyValue
);
BOOL PX_Currency(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   CY& cyValue,
   CY cyDefault
);
```

Parameters

pPX

Pointer to the CPropExchange object (typically passed as a parameter to **DoPropExchange**).

pszPropName

The name of the property being exchanged.

cyValue

Reference to the variable where the property is stored (typically a member variable of your class).

cyDefault

Default value for the property.

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

The property's value will be read from or written to the variable referenced by *cyValue*, as appropriate. If *cyDefault* is specified, it will be used as the property's default value. This value is used if, for any reason, the control's serialization process fails.

See Also

Reference

COleControl::DoPropExchange

Concepts

MFC Macros and Globals

PX_DataPath

Call this function within your control's **DoPropExchange** member function to serialize or initialize a data path property of type CDataPathProperty.

```
BOOL PX_DataPath(
   CPropExchange* pPX,
      LPCTSTR pszPropName,
      CDataPathProperty& dataPathProperty
);
BOOL PX_DataPath(
   CPropExchange* pPX,
      CDataPathProperty& dataPathProperty
);
```

Parameters

pPX

Pointer to the CPropExchange object (typically passed as a parameter to **DoPropExchange**).

pszPropName

The name of the property being exchanged.

dataPathProperty

Reference to the variable where the property is stored (typically a member variable of your class).

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

Data path properties implement asynchronous control properties. The property's value will be read from or written to the variable referenced by *dataPathProperty*, as appropriate.

See Also

Reference

COleControl::DoPropExchange

CDataPathProperty Class

Concepts

MFC Macros and Globals

PX_Double

Call this function within your control's **DoPropExchange** member function to serialize or initialize a property of type **double**.

```
BOOL PX_Double(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   double& doubleValue
);
BOOL PX_Double(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   double& doubleValue,
   double doubleDefault
);
```

Parameters

pPX

Pointer to the CPropExchange object (typically passed as a parameter to **DoPropExchange**).

pszPropName

The name of the property being exchanged.

doubleValue

Reference to the variable where the property is stored (typically a member variable of your class).

doubleDefault

Default value for the property.

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

The property's value is read from or written to the variable referenced by *doubleValue*, as appropriate. If *doubleDefault* is specified, it will be used as the property's default value. This value is used if, for any reason, the control's serialization process fails.

See Also

Reference

COleControl::DoPropExchange
PX_Float
PX_Short

Concepts

MFC Macros and Globals

PX_Float

Call this function within your control's **DoPropExchange** member function to serialize or initialize a property of type **float**.

```
BOOL PX_Float(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   float& floatValue
);
BOOL PX_Float(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   float& floatValue,
   float floatDefault
);
```

Parameters

pPX

Pointer to the CPropExchange object (typically passed as a parameter to **DoPropExchange**).

pszPropName

The name of the property being exchanged.

floatValue

Reference to the variable where the property is stored (typically a member variable of your class).

floatDefault

Default value for the property.

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

The property's value is read from or written to the variable referenced by *floatValue*, as appropriate. If *floatDefault* is specified, it will be used as the property's default value. This value is used if, for any reason, the control's serialization process fails.

See Also

Reference

COleControl::DoPropExchange
PX_Double
PX_String

Concepts

MFC Macros and Globals

PX_Font

Call this function within your control's **DoPropExchange** member function to serialize or initialize a property of type font.

```
BOOL PX_Font(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   CFontHolder& font,
   const FONTDESC FAR* pFontDesc = NULL,
   LPFONTDISP pFontDispAmbient = NULL
);
```

Parameters

pPX

Pointer to the CPropExchange object (typically passed as a parameter to **DoPropExchange**).

pszPropName

The name of the property being exchanged.

font

A reference to a **CFontHolder** object that contains the font property.

pFontDesc

A pointer to a **FONTDESC** structure containing the values to use in initializing the default state of the font property, in the case where *pFontDispAmbient* is **NULL**.

pFontDispAmbient

A pointer to the **IFontDisp** interface of a font to use in initializing the default state of the font property.

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

The property's value is read from or written to *font*, a **CFontHolder** reference, when appropriate. If *pFontDesc* and *pFontDispAmbient* are specified, they are used for initializing the property's default value, when needed. These values are used if, for any reason, the control's serialization process fails. Typically, you pass **NULL** for *pFontDesc* and the ambient value returned by **COleControl::AmbientFont** for *pFontDispAmbient*. Note that the font object returned by **COleControl::AmbientFont** must be released by a call to the **IFontDisp::Release** member function.

See Also

Reference

COleControl::DoPropExchange
COleControl::AmbientFont

Concepts

MFC Macros and Globals

PX_IUnknown

Call this function within your control's **DoPropExchange** member function to serialize or initialize a property represented by an object having an **IUnknown**-derived interface.

```
BOOL PX_IUnknown(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   LPUNKNOWN& pUnk,
   REFIID iid,
   LPUNKNOWN pUnkDefault = NULL
);
```

Parameters

pPX

Pointer to the CPropExchange object (typically passed as a parameter to **DoPropExchange**).

pszPropName

The name of the property being exchanged.

pUnk

Reference to a variable containing the interface of the object that represents the value of the property.

iid

An interface ID indicating which interface of the property object is used by the control.

pUnkDefault

Default value for the property.

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

The property's value is read from or written to the variable referenced by *pUnk*, as appropriate. If *pUnkDefault* is specified, it will be used as the property's default value. This value is used if, for any reason, the control's serialization process fails.

See Also

Reference

COleControl::DoPropExchange

Concepts

MFC Macros and Globals

PX_Long

Call this function within your control's **DoPropExchange** member function to serialize or initialize a property of type **long**.

```
BOOL PX_Long(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   long& lValue
);
BOOL PX_Long(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   long& lValue,
   long lDefault
);
```

Parameters

pPX

Pointer to the CPropExchange object (typically passed as a parameter to **DoPropExchange**).

pszPropName

The name of the property being exchanged.

lValue

Reference to the variable where the property is stored (typically a member variable of your class).

lDefault

Default value for the property.

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

The property's value is read from or written to the variable referenced by *lValue*, as appropriate. If *lDefault* is specified, it will be used as the property's default value. This value is used if, for any reason, the control's serialization process fails.

See Also

Reference

COleControl::DoPropExchange

Concepts

MFC Macros and Globals

PX_Picture

Call this function within your control's **DoPropExchange** member function to serialize or initialize a picture property of your control.

```
BOOL PX_Picture(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   CPictureHolder& pict
);
BOOL PX_Picture(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   CPictureHolder& pict,
   CPictureHolder& pictDefault
);
```

Parameters

pPX

Pointer to the CPropExchange object (typically passed as a parameter to **DoPropExchange**).

pszPropName

The name of the property being exchanged.

pict

Reference to a CPictureHolder object where the property is stored (typically a member variable of your class).

pictDefault

Default value for the property.

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

The property's value is read from or written to the variable referenced by *pict*, as appropriate. If *pictDefault* is specified, it will be used as the property's default value. This value is used if, for any reason, the control's serialization process fails.

See Also

Reference

COleControl::DoPropExchange

Concepts

MFC Macros and Globals

PX_Short

Call this function within your control's **DoPropExchange** member function to serialize or initialize a property of type **short**.

```
BOOL PX_Short(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   short& sValue
);
BOOL PX_Short(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   short& sValue,
   short sDefault
);
```

Parameters

pPX

Pointer to the CPropExchange object (typically passed as a parameter to **DoPropExchange**).

pszPropName

The name of the property being exchanged.

sValue

Reference to the variable where the property is stored (typically a member variable of your class).

sDefault

Default value for the property.

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

The property's value is read from or written to the variable referenced by *sValue*, as appropriate. If *sDefault* is specified, it will be used as the property's default value. This value is used if, for any reason, the control's serialization process fails.

See Also

Reference

COleControl::DoPropExchange

Concepts

MFC Macros and Globals

PX_String

Call this function within your control's **DoPropExchange** member function to serialize or initialize a character string property.

```
BOOL PX_String(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   CString& strValue
);
BOOL PX_String(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   CString& strValue,
   CString strDefault
);
```

Parameters

pPX

Pointer to the CPropExchange object (typically passed as a parameter to **DoPropExchange**).

pszPropName

The name of the property being exchanged.

strValue

Reference to the variable where the property is stored (typically a member variable of your class).

strDefault

Default value for the property.

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

The property's value is read from or written to the variable referenced by *strValue*, as appropriate. If *strDefault* is specified, it will be used as the property's default value. This value is used if, for any reason, the control's serialization process fails.

See Also

Reference

COleControl::DoPropExchange

CStringT Class

Concepts

MFC Macros and Globals

PX_ULong

Call this function within your control's **DoPropExchange** member function to serialize or initialize a property of type **ULONG**.

```
BOOL PX_ULong(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   ULONG& ulValue
);
BOOL PX_ULong(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   ULONG& ulValue,
   long ulDefault
);
```

Parameters

pPX

Pointer to the CPropExchange object (typically passed as a parameter to **DoPropExchange**).

pszPropName

Name of the property being exchanged.

ulValue

Reference to the variable where the property is stored (typically a member variable of your class).

ulDefault

Default value for the property.

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

The property's value is read from or written to the variable referenced by *ulValue*, as appropriate. If *ulDefault* is specified, it will be used as the property's default value. This value is used if, for any reason, the control's serialization process fails.

See Also

Reference

COleControl::DoPropExchange

Concepts

MFC Macros and Globals

PX_UShort

Call this function within your control's **DoPropExchange** member function to serialize or initialize a property of type **unsigned short**.

```
BOOL PX_UShort(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   USHORT& usValue
);
BOOL PX_UShort(
   CPropExchange* pPX,
   LPCTSTR pszPropName,
   USHORT& usValue,
   USHORT usDefault
);
```

Parameters

pPX

Pointer to the CPropExchange object (typically passed as a parameter to **DoPropExchange**).

pszPropName

Name of the property being exchanged.

usValue

Reference to the variable where the property is stored (typically a member variable of your class).

usDefault

Default value for the property.

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

The property's value is read from or written to the variable referenced by *usValue*, as appropriate. If *usDefault* is specified, it will be used as the property's default value. This value is used if, for any reason, the control's serialization process fails.

See Also

Reference

COleControl::DoPropExchange

Concepts

MFC Macros and Globals

PX_VBXFontConvert

Call this function within your control's **DoPropExchange** member function to initialize a font property by converting a VBX control's font-related properties.

```
BOOL PX_VBXFontConvert(
   CPropExchange* pPX,
   CFontHolder& font
);
```

Parameters

pPX

Pointer to the CPropExchange object (typically passed as a parameter to **DoPropExchange**).

font

The font property of the OLE control that will contain the converted VBX font-related properties.

Return Value

Nonzero if the exchange was successful; 0 if unsuccessful.

Remarks

This function should be used only by an OLE control that is designed as a direct replacement for a VBX control. When the Visual Basic development environment converts a form containing a VBX control to use the corresponding replacement OLE control, it will call the control's **IDataObject::SetData** function, passing in a property set that contains the VBX control's property data. This operation, in turn, causes the control's **DoPropExchange** function to be invoked. **DoPropExchange** can call **PX_VBXFontConvert** to convert the VBX control's font-related properties (for example, "FontName," "FontSize," and so on) into the corresponding components of the OLE control's font property.

PX_VBXFontConvert should only be called when the control is actually being converted from a VBX form application. For example:

```
void CSampleCtrl::DoPropExchange(CPropExchange* pPX)
{
    ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
    COleControl::DoPropExchange(pPX);

    if (IsConvertingVBX())
        PX_VBXFontConvert(pPX, InternalGetFont());
}
```

See Also

Reference

COleControl::DoPropExchange
COleControl::AmbientFont
PX_Font

Concepts

MFC Macros and Globals

RFX_Binary

Transfers arrays of bytes between the field data members of a **CRecordset** object and the columns of a record on the data source of ODBC type **SQL_BINARY**, **SQL_VARBINARY**, or **SQL_LONGVARBINARY**.

```
void RFX_Binary(
   CFieldExchange* pFX,
   const char* szName,
   CByteArray& value,
   int nMaxLength = 255
);
```

Parameters

pFX

A pointer to an object of class CFieldExchange. This object contains information to define the context for each call of the function. For more information about the operations a **CFieldExchange** object can specify, see the article Record Field Exchange: How RFX Works.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. For a transfer from recordset to data source, the value, of type CByteArray, is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

nMaxLength

The maximum allowed length of the string or array being transferred. The default value of *nMaxLength* is 255. Legal values are 1 to INT_MAX. The framework allocates this amount of space for the data. For best performance, pass a value large enough to accommodate the largest data item you expect.

Remarks

Data in the data source of these types is mapped to and from type **CByteArray** in the recordset.

Example

See RFX_Text.

See Also

Reference

RFX_Text
RFX_Bool
RFX_Long
RFX_Int
RFX_Single
RFX_Double
RFX_Date
RFX_Byte
RFX_LongBinary
CFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

RFX_Binary_Bulk

Transfers multiple rows of byte data from a column of an ODBC data source to a corresponding array in a **CRecordset**-derived object.

```
void RFX_Binary_Bulk(
   CFieldExchange* pFX,
   LPCTSTR szName,
   BYTE** prgByteVals,
   long** prgLengths,
   int nMaxLength
);
```

Parameters

pFX

A pointer to a CFieldExchange object. This object contains information to define the context for each call of the function. For more information, see the article Record Field Exchange: How RFX Works.

szName

The name of a data column.

prgByteVals

A pointer to an array of **BYTE** values. This array will store the data to be transferred from the data source to the recordset.

prgLengths

A pointer to an array of long integers. This array will store the length in bytes of each value in the array pointed to by *prgByteVals*. Note that the value **SQL_NULL_DATA** will be stored if the corresponding data item contains a Null value. For more details, see the ODBC API function **SQLBindCol** in the *ODBC SDK Programmer's Reference*.

nMaxLength

The maximum allowed length of the values stored in the array pointed to by *prgByteVals*. To ensure that data will not be truncated, pass a value large enough to accommodate the largest data item you expect.

Remarks

The data source column can have an ODBC type of **SQL_BINARY**, **SQL_VARBINARY**, or **SQL_LONGVARBINARY**. The recordset must define a field data member of type pointer to **BYTE**.

If you initialize *prgByteVals* and *prgLengths* to **NULL**, then the arrays they point to will be allocated automatically, with sizes equal to the rowset size.

Note
Bulk record field exchange only transfers data from the data source to the recordset object. In order to make your recordset u pdateable, you must use the ODBC API function **SQLSetPos**.

For more information, see the articles Recordset: Fetching Records in Bulk (ODBC) and Record Field Exchange (RFX).

Example

See RFX_Text_Bulk.

See Also

Reference

RFX_Bool_Bulk
RFX_Byte_Bulk
RFX_Date_Bulk
RFX_Double_Bulk
RFX_Int_Bulk
RFX_Long_Bulk

RFX_Single_Bulk
RFX_Text_Bulk
CFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

RFX_Bool

Transfers Boolean data between the field data members of a **CRecordset** object and the columns of a record on the data source of ODBC type **SQL_BIT**.

```
void RFX_Bool(
   CFieldExchange* pFX,
   const char* szName,
   BOOL& value
);
```

Parameters

pFX

A pointer to an object of class CFieldExchange. This object contains information to define the context for each call of the function. For more information about the operations a **CFieldExchange** object can specify, see the article Record Field Exchange: How RFX Works.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. For a transfer from recordset to data source, the value, of type **BOOL**, is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

Example

See RFX_Text.

See Also

Reference

RFX_Text
RFX_Long
RFX_Int
RFX_Single
RFX_Double
RFX_Date
RFX_Byte
RFX_Binary
RFX_LongBinary
CFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

RFX_Bool_Bulk

Transfers multiple rows of Boolean data from a column of an ODBC data source to a corresponding array in a **CRecordset**-derived object.

```
void RFX_Bool_Bulk(
   CFieldExchange* pFX,
   LPCTSTR szName,
   BOOL** prgBoolVals,
   long** prgLengths
);
```

Parameters

pFX

A pointer to a CFieldExchange object. This object contains information to define the context for each call of the function. For more information, see the article Record Field Exchange: How RFX Works.

szName

The name of a data column.

prgBoolVals

A pointer to an array of **BOOL** values. This array will store the data to be transferred from the data source to the recordset.

prgLengths

A pointer to an array of long integers. This array will store the length in bytes of each value in the array pointed to by *prgBoolVals*. Note that the value **SQL_NULL_DATA** will be stored if the corresponding data item contains a Null value. For more details, see the ODBC API function **SQLBindCol** in the *ODBC SDK Programmer's Reference*.

Remarks

The data source column must have an ODBC type of **SQL_BIT**. The recordset must define a field data member of type pointer to **BOOL**.

If you initialize *prgBoolVals* and *prgLengths* to **NULL**, then the arrays they point to will be allocated automatically, with sizes equal to the rowset size.

Note
Bulk record field exchange only transfers data from the data source to the recordset object. To make your recordset updateable, you must use the ODBC API function **SQLSetPos**.

For more information, see the articles Recordset: Fetching Records in Bulk (ODBC) and Record Field Exchange (RFX).

Example

See RFX_Text_Bulk.

See Also

Reference

RFX_Binary_Bulk
RFX_Byte_Bulk
RFX_Date_Bulk
RFX_Double_Bulk
RFX_Int_Bulk
RFX_Long_Bulk
RFX_Single_Bulk
RFX_Text_Bulk
CFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

RFX_Byte

Transfers single bytes between the field data members of a **CRecordset** object and the columns of a record on the data source of ODBC type **SQL_TINYINT**.

```
void RFX_Byte(
   CFieldExchange* pFX,
   const char* szName,
   BYTE& value
);
```

Parameters

pFX

A pointer to an object of class CFieldExchange. This object contains information to define the context for each call of the function. For more information about the operations a **CFieldExchange** object can specify, see the article Record Field Exchange: How RFX Works.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. For a transfer from recordset to data source, the value, of type **BYTE**, is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

Example

See RFX_Text.

See Also

Reference

RFX_Text
RFX_Bool
RFX_Long
RFX_Int
RFX_Single
RFX_Double
RFX_Date
RFX_Binary
RFX_LongBinary
CFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

RFX_Byte_Bulk

Transfers multiple rows of single bytes from a column of an ODBC data source to a corresponding array in a **CRecordset**-derived object.

```
void RFX_Byte_Bulk(
   CFieldExchange* pFX,
   LPCTSTR szName,
   BYTE** prgByteVals,
   long** prgLengths
);
```

Parameters

pFX

A pointer to a CFieldExchange object. This object contains information to define the context for each call of the function. For more information, see the article Record Field Exchange: How RFX Works.

szName

The name of a data column.

prgByteVals

A pointer to an array of **BYTE** values. This array will store the data to be transferred from the data source to the recordset.

prgLengths

A pointer to an array of long integers. This array will store the length in bytes of each value in the array pointed to by *prgByteVals*. Note that the value **SQL_NULL_DATA** will be stored if the corresponding data item contains a Null value. For more details, see the ODBC API function **SQLBindCol** in the *ODBC SDK Programmer's Reference*.

Remarks

The data source column must have an ODBC type of **SQL_TINYINT**. The recordset must define a field data member of type pointer to **BYTE**.

If you initialize *prgByteVals* and *prgLengths* to **NULL**, then the arrays they point to will be allocated automatically, with sizes equal to the rowset size.

Note
Bulk record field exchange only transfers data from the data source to the recordset object. To make your recordset updateable, you must use the ODBC API function **SQLSetPos**.

For more information, see the articles Recordset: Fetching Records in Bulk (ODBC) and Record Field Exchange (RFX).

Example

See RFX_Text_Bulk.

See Also

Reference

RFX_Binary_Bulk
RFX_Bool_Bulk
RFX_Date_Bulk
RFX_Double_Bulk
RFX_Int_Bulk
RFX_Long_Bulk
RFX_Single_Bulk
RFX_Text_Bulk
CFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

RFX_Date

Transfers **CTime** or **TIMESTAMP_STRUCT** data between the field data members of a **CRecordset** object and the columns of a record on the data source of ODBC type **SQL_DATE**, **SQL_TIME**, or **SQL_TIMESTAMP**.

```
void RFX_Date(
   CFieldExchange* pFX,
   const char* szName,
   CTime& value
);
void RFX_Date(
   CFieldExchange* pFX,
   const char* szName,
   TIMESTAMP_STRUCT& value
);
void RFX_Date(
   CFieldExchange* pFX,
   const char* szName,
   COleDateTime& value
);
```

Parameters

pFX

A pointer to an object of class CFieldExchange. This object contains information to define the context for each call of the function. For more information about the operations a **CFieldExchange** object can specify, see the article Record Field Exchange: How RFX Works.

szName

The name of a data column.

value

The value stored in the indicated data member; the value to be transferred. The various versions of the function take different data types for value:

The first version of the function takes a reference to a CTime object. For a transfer from recordset to data source, this value is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

The second version of the function takes a reference to a **TIMESTAMP_STRUCT** structure. You must set up this structure yourself before the call. Neither dialog data exchange (DDX) support nor code wizard support is available for this version.
The third version of the function works similarly to the first version except that it takes a reference to a COleDateTime object.

Remarks

The **CTime** version of the function imposes the overhead of some intermediate processing and has a somewhat limited range. If you find either of these factors too limiting, use the second version of the function. But note its lack of code wizard and DDX support and the requirement that you set up the structure yourself.

Example

See RFX_Text.

See Also

Reference

RFX_Text
RFX_Bool
RFX_Long
RFX_Int
RFX_Single
RFX_Double
RFX_Byte
RFX_Binary

RFX_LongBinary
CFieldExchange::SetFieldType
Concepts
MFC Macros and Globals

RFX_Date_Bulk

Transfers multiple rows of **TIMESTAMP_STRUCT** data from a column of an ODBC data source to a corresponding array in a **CRecordset**-derived object.

```
void RFX_Date_Bulk(
   CFieldExchange* pFX,
   LPCTSTR szName,
   TIMESTAMP_STRUCT** prgTSVals,
   long** prgLengths
);
```

Parameters

pFX

A pointer to a CFieldExchange object. This object contains information to define the context for each call of the function. For more information, see the article Record Field Exchange: How RFX Works.

szName

The name of a data column.

prgTSVals

A pointer to an array of **TIMESTAMP_STRUCT** values. This array will store the data to be transferred from the data source to the recordset. For more information about the **TIMESTAMP_STRUCT** data type, see the topic "C Data Types" in Appendix D of the *ODBC SDK Programmer's Reference*.

prgLengths

A pointer to an array of long integers. This array will store the length in bytes of each value in the array pointed to by *prgTSVals*. Note that the value **SQL_NULL_DATA** will be stored if the corresponding data item contains a Null value. For more details, see the ODBC API function **SQLBindCol** in the *ODBC SDK Programmer's Reference*.

Remarks

The data source column can have an ODBC type of **SQL_DATE**, **SQL_TIME**, or **SQL_TIMESTAMP**. The recordset must define a field data member of type pointer to **TIMESTAMP_STRUCT**.

If you initialize *prgTSVals* and *prgLengths* to **NULL**, then the arrays they point to will be allocated automatically, with sizes equal to the rowset size.

Note
Bulk record field exchange only transfers data from the data source to the recordset object. To make your recordset updateable, you must use the ODBC API function **SQLSetPos**.

For more information, see the articles Recordset: Fetching Records in Bulk (ODBC) and Record Field Exchange (RFX).

Example

See RFX_Text_Bulk.

See Also

Reference

RFX_Binary_Bulk
RFX_Bool_Bulk
RFX_Byte_Bulk
RFX_Double_Bulk
RFX_Int_Bulk
RFX_Long_Bulk
RFX_Single_Bulk
RFX_Text_Bulk
CFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

RFX_Double

Transfers **double float** data between the field data members of a **CRecordset** object and the columns of a record on the data source of ODBC type **SQL_DOUBLE**.

```
void RFX_Double(
   CFieldExchange* pFX,
   const char* szName,
   double& value
);
```

Parameters

pFX

A pointer to an object of class CFieldExchange. This object contains information to define the context for each call of the function. For more information about the operations a **CFieldExchange** object can specify, see the article Record Field Exchange: How RFX Works.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. For a transfer from recordset to data source, the value, of type **double**, is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

Example

See RFX_Text.

See Also

Reference

RFX_Text
RFX_Bool
RFX_Long
RFX_Int
RFX_Single
RFX_Date
RFX_Byte
RFX_Binary
RFX_LongBinary
CFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

RFX_Double_Bulk

Transfers multiple rows of double-precision, floating-point data from a column of an ODBC data source to a corresponding array in a **CRecordset**-derived object.

```
void RFX_Double_Bulk(
   CFieldExchange* pFX,
   LPCTSTR szName,
   double** prgDblVals,
   long** prgLengths
);
```

Parameters

pFX

A pointer to a CFieldExchange object. This object contains information to define the context for each call of the function. For more information, see the article Record Field Exchange: How RFX Works.

szName

The name of a data column.

prgDblVals

A pointer to an array of **double** values. This array will store the data to be transferred from the data source to the recordset.

prgLengths

A pointer to an array of long integers. This array will store the length in bytes of each value in the array pointed to by *prgDblVals*. Note that the value **SQL_NULL_DATA** will be stored if the corresponding data item contains a Null value. For more details, see the ODBC API function **SQLBindCol** in the *ODBC SDK Programmer's Reference*.

Remarks

The data source column must have an ODBC type of **SQL_DOUBLE**. The recordset must define a field data member of type pointer to **double**.

If you initialize *prgDblVals* and *prgLengths* to **NULL**, then the arrays they point to will be allocated automatically, with sizes equal to the rowset size.

Note
Bulk record field exchange only transfers data from the data source to the recordset object. To make your recordset updateable, you must use the ODBC API function **SQLSetPos**.

For more information, see the articles Recordset: Fetching Records in Bulk (ODBC) and Record Field Exchange (RFX).

Example

See RFX_Text_Bulk.

See Also

Reference

RFX_Binary_Bulk
RFX_Bool_Bulk
RFX_Byte_Bulk
RFX_Date_Bulk
RFX_Int_Bulk
RFX_Long_Bulk
RFX_Single_Bulk
RFX_Text_Bulk
CFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

RFX_Int

Transfers integer data between the field data members of a **CRecordset** object and the columns of a record on the data source of ODBC type **SQL_SMALLINT**.

```
void RFX_Int(
   CFieldExchange* pFX,
   const char* szName,
   int& value
);
```

Parameters

pFX

A pointer to an object of class CFieldExchange. This object contains information to define the context for each call of the function. For more information about the operations a **CFieldExchange** object can specify, see the article Record Field Exchange: How RFX Works.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. For a transfer from recordset to data source, the value, of type **int**, is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

Example

See RFX_Text.

See Also

Reference

RFX_Text
RFX_Bool
RFX_Long
RFX_Single
RFX_Double
RFX_Date
RFX_Byte
RFX_Binary
RFX_LongBinary
CFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

RFX_Int_Bulk

Transfers multiple rows of integer data from a column of an ODBC data source to a corresponding array in a **CRecordset**-derived object.

```
void RFX_Int_Bulk(
   CFieldExchange* pFX,
   LPCTSTR szName,
   int** prgIntVals,
   long** prgLengths
);
```

Parameters

pFX

A pointer to a CFieldExchange object. This object contains information to define the context for each call of the function. For more information, see the article Record Field Exchange: How RFX Works.

szName

The name of a data column.

prgIntVals

A pointer to an array of integers. This array will store the data to be transferred from the data source to the recordset.

prgLengths

A pointer to an array of long integers. This array will store the length in bytes of each value in the array pointed to by *prgIntVals*. Note that the value **SQL_NULL_DATA** will be stored if the corresponding data item contains a Null value. For more details, see the ODBC API function **SQLBindCol** in the *ODBC SDK Programmer's Reference*.

Remarks

The data source column must have an ODBC type of **SQL_SMALLINT**. The recordset must define a field data member of type pointer to **int**.

If you initialize *prgIntVals* and *prgLengths* to **NULL**, then the arrays they point to will be allocated automatically, with sizes equal to the rowset size.

Note
Bulk record field exchange only transfers data from the data source to the recordset object. To make your recordset updateable, you must use the ODBC API function **SQLSetPos**.

For more information, see the articles Recordset: Fetching Records in Bulk (ODBC) and Record Field Exchange (RFX).

Example

See RFX_Text_Bulk.

See Also

Reference

RFX_Binary_Bulk
RFX_Bool_Bulk
RFX_Byte_Bulk
RFX_Date_Bulk
RFX_Double_Bulk
RFX_Long_Bulk
RFX_Single_Bulk
RFX_Text_Bulk
CFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

RFX_Long

Transfers long integer data between the field data members of a **CRecordset** object and the columns of a record on the data source of ODBC type **SQL_INTEGER**.

```
void RFX_Long(
   CFieldExchange* pFX,
   const char* szName,
   LONG&
value );
```

Parameters

pFX

A pointer to an object of class CFieldExchange. This object contains information to define the context for each call of the function. For more information about the operations a **CFieldExchange** object can specify, see the article Record Field Exchange: How RFX Works.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. For a transfer from recordset to data source, the value, of type **long**, is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

Example

See RFX_Text.

See Also

Reference

RFX_Text
RFX_Bool
RFX_Int
RFX_Single
RFX_Double
RFX_Date
RFX_Byte
RFX_Binary
RFX_LongBinary
CFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

RFX_Long_Bulk

Transfers multiple rows of long integer data from a column of an ODBC data source to a corresponding array in a **CRecordset**-derived object.

```
void RFX_Long_Bulk(
   CFieldExchange* pFX,
   LPCTSTR szName,
   long** prgLongVals,
   long** prgLengths
);
```

Parameters

pFX

A pointer to a CFieldExchange object. This object contains information to define the context for each call of the function. For more information, see the article Record Field Exchange: How RFX Works.

szName

The name of a data column.

prgLongVals

A pointer to an array of long integers. This array will store the data to be transferred from the data source to the recordset.

prgLengths

A pointer to an array of long integers. This array will store the length in bytes of each value in the array pointed to by *prgLongVals*. Note that the value **SQL_NULL_DATA** will be stored if the corresponding data item contains a Null value. For more details, see the ODBC API function **SQLBindCol** in the *ODBC SDK Programmer's Reference*.

Remarks

The data source column must have an ODBC type of **SQL_INTEGER**. The recordset must define a field data member of type pointer to **long**.

If you initialize *prgLongVals* and *prgLengths* to **NULL**, then the arrays they point to will be allocated automatically, with sizes equal to the rowset size.

Note
Bulk record field exchange only transfers data from the data source to the recordset object. To make your recordset updateable, you must use the ODBC API function **SQLSetPos**.

For more information, see the articles Recordset: Fetching Records in Bulk (ODBC) and Record Field Exchange (RFX).

Example

See RFX_Text_Bulk.

See Also

Reference

RFX_Binary_Bulk
RFX_Bool_Bulk
RFX_Byte_Bulk
RFX_Date_Bulk
RFX_Double_Bulk
RFX_Int_Bulk
RFX_Single_Bulk
RFX_Text_Bulk
CFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

RFX_LongBinary

Transfers binary large object (BLOB) data using class CLongBinary between the field data members of a **CRecordset** object and the columns of a record on the data source of ODBC type **SQL_LONGVARBINARY** or **SQL_LONGVARCHAR**.

```
void RFX_LongBinary(
   CFieldExchange* pFX,
   const char* szName,
   CLongBinary& value
);
```

Parameters

pFX

A pointer to an object of class CFieldExchange. This object contains information to define the context for each call of the function. For more information about the operations a **CFieldExchange** object can specify, see the article Record Field Exchange: How RFX Works.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. For a transfer from recordset to data source, the value, of type **CLongBinary**, is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

Example

See RFX_Text.

See Also

Reference

RFX_Text
RFX_Bool
RFX_Long
RFX_Int
RFX_Single
RFX_Double
RFX_Date
RFX_Byte
RFX_Binary
CFieldExchange::SetFieldType
CLongBinary Class

Concepts

MFC Macros and Globals

RFX_Single

Transfers floating-point data between the field data members of a **CRecordset** object and the columns of a record on the data source of ODBC type **SQL_REAL**.

```
void RFX_Single(
   CFieldExchange* pFX,
   const char* szName,
   float& value
);
```

Parameters

pFX

A pointer to an object of class CFieldExchange. This object contains information to define the context for each call of the function. For more information about the operations a **CFieldExchange** object can specify, see the article Record Field Exchange: How RFX Works.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. For a transfer from recordset to data source, the value, of type **float**, is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

Example

See RFX_Text.

See Also

Reference

RFX_Text
RFX_Bool
RFX_Long
RFX_Int
RFX_Double
RFX_Date
RFX_Byte
RFX_Binary
RFX_LongBinary
CFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

RFX_Single_Bulk

Transfers multiple rows of floating-point data from a column of an ODBC data source to a corresponding array in a **CRecordset**-derived object.

```
void RFX_Single_Bulk(
   CFieldExchange* pFX,
   LPCTSTR szName,
   float** prgFltVals,
   long** prgLengths
);
```

Parameters

pFX

A pointer to a CFieldExchange object. This object contains information to define the context for each call of the function. For more information, see the article Record Field Exchange: How RFX Works.

szName

The name of a data column.

prgFltVals

A pointer to an array of **float** values. This array will store the data to be transferred from the data source to the recordset.

prgLengths

A pointer to an array of long integers. This array will store the length in bytes of each value in the array pointed to by *prgFltVals*. Note that the value **SQL_NULL_DATA** will be stored if the corresponding data item contains a Null value. For more details, see the ODBC API function **SQLBindCol** in the *ODBC SDK Programmer's Reference*.

Remarks

The data source column must have an ODBC type of **SQL_REAL**. The recordset must define a field data member of type pointer to **float**.

If you initialize *prgFltVals* and *prgLengths* to **NULL**, then the arrays they point to will be allocated automatically, with sizes equal to the rowset size.

Note
Bulk record field exchange only transfers data from the data source to the recordset object. To make your recordset updateable, you must use the ODBC API function **SQLSetPos**.

For more information, see the articles Recordset: Fetching Records in Bulk (ODBC) and Record Field Exchange (RFX).

Example

See RFX_Text_Bulk.

See Also

Reference

RFX_Binary_Bulk
RFX_Bool_Bulk
RFX_Byte_Bulk
RFX_Date_Bulk
RFX_Double_Bulk
RFX_Int_Bulk
RFX_Long_Bulk
RFX_Text_Bulk
CFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

RFX_Text

Transfers **CString** data between the field data members of a **CRecordset** object and columns of a record on the data source of ODBC type **SQL_LONGVARCHAR**, **SQL_CHAR**, **SQL_VARCHAR**, **SQL_DECIMAL**, or **SQL_NUMERIC**.

```
void RFX_Text(
   CFieldExchange* pFX,
   const char* szName,
   CString& value,
   int nMaxLength = 255,
   int nColumnType = SQL_VARCHAR,
   short nScale = 0
);
```

Parameters

pFX

A pointer to an object of class **CFieldExchange**. This object contains information to define the context for each call of the function. For more information about the operations a **CFieldExchange** object can specify, see the article Record Field Exchange: How RFX Works.

szName

The name of a data column.

value

The value stored in the indicated data member — the value to be transferred. For a transfer from recordset to data source, the value, of type **CString**, is taken from the specified data member. For a transfer from data source to recordset, the value is stored in the specified data member.

nMaxLength

The maximum allowed length of the string or array being transferred. The default value of *nMaxLength* is 255. Legal values are 1 to INT_MAX. The framework allocates this amount of space for the data. For best performance, pass a value large enough to accommodate the largest data item you expect.

nColumnType

Used mainly for parameters. An integer indicating the data type of the parameter. The type is an ODBC data type of the form **SQL_XXX**.

nScale

Specifies the scale for values of ODBC type **SQL_DECIMAL** or **SQL_NUMERIC**. *nScale* is only useful when setting parameter values. For more information, see the topic "Precision, Scale, Length, and Display Size" in Appendix D of the *ODBC SDK Programmer's Reference*.

Remarks

Data in the data source of all of these types is mapped to and from **CString** in the recordset.

Example

This example shows several calls to **RFX_Text**. Notice also the two calls to **CFieldExchange::SetFieldType**. You must write the first call to **SetFieldType** and its RFX call. The second call and its associated RFX calls are normally written by a code wizard.

```
//Example for RFX_Text
void CSections::DoFieldExchange(CFieldExchange* pFX)
{
   pFX->SetFieldType(CFieldExchange::inputParam);
   RFX_Text(pFX, "Name", m_strNameParam);
   pFX->SetFieldType(CFieldExchange::outputColumn);
   RFX_Text(pFX, "CourseID", m_strCourseID);
   RFX_Text(pFX, "InstructorID", m_strInstructorID);
   RFX_Int(pFX, "RoomNo", m_nRoomNo);
```

```
    RFX_Text(pFX, "Schedule", m_strSchedule);
    RFX_Int(pFX, "SectionNo", m_nSectionNo);
    RFX_Single(pFX, "LabFee", m_flLabFee);
}
```

See Also

Reference

RFX_Bool
RFX_Long
RFX_Int
RFX_Single
RFX_Double
RFX_Date
RFX_Byte
RFX_Binary
RFX_LongBinary
CFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

RFX_Text_Bulk

Transfers multiple rows of character data from a column of an ODBC data source to a corresponding array in a **CRecordset**-derived object.

```
void RFX_Text_Bulk(
   CFieldExchange* pFX,
   LPCTSTR szName,
   LPSTR* prgStrVals,
   long** prgLengths,
   int nMaxLength
);
```

Parameters

pFX

A pointer to a CFieldExchange object. This object contains information to define the context for each call of the function. For more information, see the article Record Field Exchange: How RFX Works.

szName

The name of a data column.

prgStrVals

A pointer to an array of **LPSTR** values. This array will store the data to be transferred from the data source to the recordset. Note that with the current version of ODBC, these values cannot be Unicode.

prgLengths

A pointer to an array of long integers. This array will store the length in bytes of each value in the array pointed to by *prgStrVals*. This length excludes the null termination character. Note that the value **SQL_NULL_DATA** will be stored if the corresponding data item contains a Null value. For more details, see the ODBC API function **SQLBindCol** in the *ODBC SDK Programmer's Reference*.

nMaxLength

The maximum allowed length of the values stored in the array pointed to by *prgStrVals*, including the null termination character. To ensure that data will not be truncated, pass a value large enough to accommodate the largest data item you expect.

Remarks

The data source column can have an ODBC type of **SQL_LONGVARCHAR**, **SQL_CHAR**, **SQL_VARCHAR**, **SQL_DECIMAL**, or **SQL_NUMERIC**. The recordset must define a field data member of type **LPSTR**.

If you initialize *prgStrVals* and *prgLengths* to **NULL**, then the arrays they point to will be allocated automatically, with sizes equal to the rowset size.

Note
Bulk record field exchange only transfers data from the data source to the recordset object. To make your recordset updateable, you must use the ODBC API function **SQLSetPos**.

For more information, see the articles Recordset: Fetching Records in Bulk (ODBC) and Record Field Exchange (RFX).

Example

You must manually write calls in your **DoBulkFieldExchange** override. This example shows a call to **RFX_Text_Bulk**, as well as a call to **RFX_Long_Bulk**, for data transfer. These calls are preceded by a call to CFieldExchange::SetFieldType. Note that for parameters, you must call the RFX functions instead of the Bulk RFX functions.

```
void MultiRowSet::DoBulkFieldExchange( CFieldExchange* pFX )
{
   pFX->SetFieldType( CFieldExchange::outputColumn );
```

```
    RFX_Long_Bulk( pFX, _T( "[colRecID]" ),
                   &m_rgID, &m_rgIDLenghts );
    RFX_Text_Bulk( pFX, _T( "[colName]" ),
                   &m_rgName, &m_rgNameLengths, 30 );

    pFX->SetFieldType( CFieldExchange::inputParam );
    RFX_Text( pFX, "NameParam", m_strNameParam );
  }
```

See Also

Reference

RFX_Binary_Bulk
RFX_Bool_Bulk
RFX_Byte_Bulk
RFX_Date_Bulk
RFX_Double_Bulk
RFX_Int_Bulk
RFX_Long_Bulk
RFX_Single_Bulk
CFieldExchange::SetFieldType

Concepts

MFC Macros and Globals

RUNTIME_CLASS

Gets the run-time class structure from the name of a C++ class.

```
RUNTIME_CLASS(class_name )
```

Parameters

class_name

The actual name of the class (not enclosed in quotation marks).

Remarks

RUNTIME_CLASS returns a pointer to a CRuntimeClass structure for the class specified by *class_name*. Only **CObject**-derived classes declared with **DECLARE_DYNAMIC**, **DECLARE_DYNCREATE**, or **DECLARE_SERIAL** will return pointers to a **CRuntimeClass** structure.

For more information, see CObject Class Topics.

Example

```
// Example for RUNTIME_CLASS
CRuntimeClass* prt = RUNTIME_CLASS( CAge );
ASSERT( lstrcmp( prt->m_lpszClassName, "CAge" )  == 0 );
```

See Also

Reference

DECLARE_DYNAMIC
DECLARE_DYNCREATE
DECLARE_SERIAL
CObject::GetRuntimeClass
CRuntimeClass Structure

Concepts

MFC Macros and Globals

SerializeElements

CArray, CList, and CMap call this function to serialize elements.

```
template< class TYPE >
void AFXAPI SerializeElements(
   CArchive& ar,
      TYPE* pElements,
   INT_PTR nCount
);
```

Parameters

TYPE

Template parameter specifying the type of the elements.

ar

An archive object to archive to or from.

pElements

Pointer to the elements being archived.

nCount

Number of elements being archived

Remarks

The default implementation does a bitwise read or write.

For information on implementing this and other helper functions, see the article Collections: How to Make a Type-Safe Collection.

Example

See the example in the article Collections: How to Make a Type-Safe Collection.

See Also

Reference

CArchive Class

Concepts

MFC Macros and Globals

STATIC_DOWNCAST

Casts *pobject* to a pointer to a *class_name* object.

```
STATIC_DOWNCAST(class_name, pobject )
```

Parameters

class_name

The name of the class being cast to.

pobject

The pointer to be cast to a pointer to a *class_name* object.

Remarks

pobject must either be **NULL**, or point to an object of a class which is derived directly, or indirectly, from *class_name*. In builds of your application with the **_DEBUG** preprocessor symbol defined, the macro will **ASSERT** if *pobject* is not **NULL**, or if it points to an object that is not a "kind of" the class specified in the *class_name* parameter (see CObject::IsKindOf). In non-**_DEBUG** builds, the macro performs the cast without any type checking.

The class specified in the *class_name* parameter must be derived from **CObject** and must use the **DECLARE_DYNAMIC** and **IMPLEMENT_DYNAMIC**, the **DECLARE_DYNCREATE** and **IMPLEMENT_DYNCREATE**, or the **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** macros as explained in the article CObject Class: Deriving a Class from CObject.

For example, you might cast a pointer to `CYourDocument`, called `pYourDoc`, to a pointer to **CDocument** using this expression:

```
CDocument* pDoc = STATIC_DOWNCAST(CDocument, pYourDoc);
```

If `pYourDoc` does not point to an object derived directly or indirectly from **CDocument**, the macro will **ASSERT**.

See Also

Reference

DYNAMIC_DOWNCAST

static_cast Operator

Concepts

MFC Macros and Globals

THIS_FILE

Expands to the name of the file that is being compiled.

```
THIS_FILE
```

Remarks

The information is used by the **ASSERT** and **VERIFY** macros. The Application Wizard and code wizards place the macro in source code files they create.

Example

```
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

// __FILE__ is one of the six predefined ANSI C macros that the
// compiler recognizes.
```

See Also

Reference

ASSERT (MFC)

VERIFY

Concepts

MFC Macros and Globals

THROW (MFC)

Throws the specified exception.

```
THROW(exception_object_pointer )
```

Parameters

exception_object_pointer

Points to an exception object derived from **CException**.

Remarks

THROW interrupts program execution, passing control to the associated **CATCH** block in your program. If you have not provided the **CATCH** block, then control is passed to a Microsoft Foundation Class Library module that prints an error message and exits.

For more information, see the article Exceptions.

See Also

Reference

THROW_LAST
TRY
CATCH
AND_CATCH
END_CATCH
CATCH_ALL
AND_CATCH_ALL
END_CATCH_ALL
AfxThrowArchiveException
AfxThrowFileException
AfxThrowMemoryException
AfxThrowNotSupportedException
AfxThrowResourceException
AfxThrowUserException

Concepts

MFC Macros and Globals

THROW_LAST

Throws the exception back to the next outer **CATCH** block.

```
THROW_LAST( )
```

Remarks

This macro allows you to throw a locally created exception. If you try to throw an exception that you have just caught, it will normally go out of scope and be deleted. With **THROW_LAST**, the exception is passed correctly to the next **CATCH** handler.

For more information, see the article Exceptions.

Example

See the example for CFile::Abort.

See Also

Reference

THROW (MFC)
TRY
CATCH
AND_CATCH
END_CATCH
CATCH_ALL
AND_CATCH_ALL
END_CATCH_ALL

Concepts

MFC Macros and Globals

TRACE

Sends the specified string to the debugger of the current application.

```
TRACE(exp )
TRACE(DWORD category, UINT level, LPCSTR lpszFormat, ... )
```

Remarks

See ATLTRACE2 for a description of **TRACE**. **TRACE** and **ATLTRACE2** have the same behavior.

In the debug version of MFC, this macro sends the specified string to the debugger of the current application. In a release build, this macro compiles to nothing (no code is generated at all).

For more information, see Debugging MFC Applications.

See Also

Reference

AfxDump (MFC)

afxTraceEnabled

Concepts

MFC Macros and Globals

TRY

Sets up a **TRY** block.

```
TRY
```

Remarks

A **TRY** block identifies a block of code that might throw exceptions. Those exceptions are handled in the following **CATCH** and **AND_CATCH** blocks. Recursion is allowed: exceptions may be passed to an outer **TRY** block, either by ignoring them or by using the **THROW_LAST** macro. End the **TRY** block with an **END_CATCH** or **END_CATCH_ALL** macro.

For more information, see the article Exceptions.

Example

See the example for CATCH.

See Also

Reference

CATCH
AND_CATCH
END_CATCH
CATCH_ALL
AND_CATCH_ALL
END_CATCH_ALL
THROW (MFC)
THROW_LAST

Concepts

MFC Macros and Globals

VERIFY

In the Debug version of MFC, evaluates its argument.

```
VERIFY(booleanExpression )
```

Parameters

booleanExpression

Specifies an expression (including pointer values) that evaluates to nonzero or 0.

Remarks

If the result is 0, the macro prints a diagnostic message and halts the program. If the condition is nonzero, it does nothing.

The diagnostic message has the form

```
assertion failed in file <name> in line <num>
```

where *name* is the name of the source file and *num* is the line number of the assertion that failed in the source file.

In the Release version of MFC, **VERIFY** evaluates the expression but does not print or interrupt the program. For example, if the expression is a function call, the call will be made.

Example

```
// VERIFY can be used for things that should never fail, though
// you may want to make sure you can provide better error recovery
// if the error can actually cause a crash in a production system.

// It _is_ possible that GetDC() may fail, but the out-of-memory
// condition that causes it isn't likely. For a test application,
// this use of VERIFY() is fine. For any production code, this
// usage is dubious.

// get the display device context
HDC hdc;
VERIFY( (hdc = ::GetDC(NULL)) != NULL);

// give the display context back
::ReleaseDC(hdc);
```

See Also

Reference

ASSERT (MFC)

Concepts

MFC Macros and Globals

UICheckState Enumeration

Describes the check state of a user interface item for the command.

```
public enum class UICheckState
{
   [DefaultValue(typeid<Microsoft::VisualC::MFC::UICheckState>, "Checked")]
   Unchecked,
   Checked,
   Indeterminate
};
```

Remarks

ICommandUI::Check uses these values to describe the state of a user interface item.

For more information on using Windows Forms, see Using a Windows Form User Control in MFC.

Requirements

Header: afxwinforms.h (defined in assembly atlmfc\lib\mfcmifc80.dll)

See Also

Reference

ICommandUI Interface

URL_EVENT_ENTRY

Maps a URL or HTML resource to a page in a multipage dialog.

```
URL_EVENT_ENTRY(className, url, mapName)
```

Parameters

className

The name of the class containing the URL event entry map. This class should derive directly or indirectly from CMultiPageDHtmlDialog. The URL event entry map must be inside a DHTML and URL event map).

url

The URL or HTML resource for the page.

mapName

Specifies the page whose URL is *url*. This matches *mapName* in the BEGIN_EMBED_DHTML_EVENT_MAP macro that maps events from this page.

Remarks

If the page is an HTML resource, *url* must be the string representation of the resource's ID number (that is, "123", not 123 or ID_HTMLRES1).

The page identifier, *mapName*, is an arbitrary symbol used to link embedded DHTML event maps to URL event entry maps. It is limited in scope to the DHTML and URL event map.

Example

See the example in BEGIN_DHTML_URL_EVENT_MAP.

See Also

Concepts

MFC Macros and Globals

Other Resources

Multipage DHTML and URL Event Maps

Standard Dialog Data Exchange Routines

This topic lists the standard dialog data exchange (DDX) routines used for common MFC dialog controls.

Note
The standard dialog data exchange routines are defined in the header file afxdd_.h. However, applications should include afx win.h.

DDX Functions

DDX_CBIndex	Initializes or retrieves the index of the current selection of a combo box control.
DDX_CBString	Initializes or retrieves the current contents of the edit field of a combo box control.
DDX_CBStringExact	Initializes or retrieves the current contents of the edit field of a combo box control.
DDX_Check	Initializes or retrieves the current state of a check box control.
DDX_Control	Subclasses a given control within a dialog box.
DDX_DateTimeCtrl	Initializes or retrieves date and/or time data of a date and time picker control.
DDX_IPAddress	Initializes or retrieves the current value of an IP address control.
DDX_LBIndex	Initializes or retrieves the index of the current selection of a list box control.
DDX_LBString	Initializes or retrieves the current selection within a list box control.
DDX_LBStringExact	Initializes or retrieves the current selection within a list box control.
DDX_MonthCalCtrl	Initializes or retrieves the current value of a month calendar control.
DDX_Radio	Initializes or retrieves the 0-based index of the radio control that is currently checked within a radio control group.
DDX_Scroll	Initializes or retrieves the current position of a scroll control's thumb.
DDX_Slider	Initializes or retrieves the current position of a slider control's thumb.
DDX_Text	Initializes or retrieves the current value of an edit control.

See Also

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Validation Routines

Standard Dialog Data Validation Routines

This topic lists the standard dialog data validation (DDV) routines used for common MFC dialog controls.

Note
The standard dialog data exchange routines are defined in the header file afxdd_.h. However, applications should include afx win.h.

DDV Functions

DDV_MaxChars	Verifies the number of characters in a given control value does not exceed a given maximum.
DDV_MinMaxByte	Verifies a given control value does not exceed a given **BYTE** range.
DDV_MinMaxDateTime	Verifies a given control value does not exceed a given time range.
DDV_MinMaxDouble	Verifies a given control value does not exceed a given **double** range.
DDV_MinMaxDWord	Verifies a given control value does not exceed a given **DWORD** range.
DDV_MinMaxFloat	Verifies a given control value does not exceed a given **float** range.
DDV_MinMaxInt	Verifies a given control value does not exceed a given **int** range.
DDV_MinMaxLong	Verifies a given control value does not exceed a given **long** range.
DDV_MinMaxLongLong	Verifies a given control value does not exceed a given **LONGLONG** range.
DDV_MinMaxMonth	Verifies a given control value does not exceed a given date range.
DDV_MinMaxShort	Verifies a given control value does not exceed a given **short** range.
DDV_MinMaxSlider	Verifies a given slider control value falls within the given range.
DDV_MinMaxUInt	Verifies a given control value does not exceed a given **UINT** range.
DDV_MinMaxULongLong	Verifies a given control value does not exceed a given **ULONGLONG** range.

See Also

Concepts

MFC Macros and Globals

Other Resources

Standard Dialog Data Exchange Routines

Structures, Styles, Callbacks, and Message Maps

This section documents the structures, styles, and callback functions used by the Microsoft Foundation Class Library and the MFC message maps.

In This Section

Structures Used by MFC

Provides links to the structures called from various member functions.

Styles Used by MFC

Provides links to the styles used when creating MFC objects.

Callback Functions Used by MFC

Provides links to the callback functions appearing in the MFC Library.

Message Maps

Describes the message mapping macros and **CWnd** message-map entries.

Related Sections

Class Library Overview

Lists the classes in the MFC Library according to category.

MFC Samples

Provides links to samples that demonstrate using the MFC Library.

Structures Used by MFC

The following table lists structures that are called from various member functions. For further information on individual structure usage, refer to the classes and member functions noted in the See Also list for each structure.

ABC Structure	**HSE_VERSION_INFO Structure**
ABCFLOAT Structure	HTTP_FILTER_AUTHENT Structure
AFX_EXTENSION_MODULE Structure	HTTP_FILTER_CONTEXT Structure
BITMAP Structure	HTTP_FILTER_LOG Structure
BITMAPINFO Structure	HTTP_FILTER_PREPROC_HEADERS Structure
CDaoDatabaseInfo Structure	HTTP_FILTER_RAW_DATA Structure
CDaoErrorInfo Structure	HTTP_FILTER_URL_MAP Structure
CDaoFieldInfo Structure	HTTP_FILTER_VERSION Structure
CDaoIndexFieldInfo Structure	LINGER Structure
CDaoIndexInfo Structure	LOGBRUSH Structure
CDaoParameterInfo Structure	LOGPEN Structure
CDaoQueryDefInfo Structure	MEASUREITEMSTRUCT Structure
CDaoRelationFieldInfo Structure	MINMAXINFO Structure
CDaoRelationInfo Structure	MSG Structure
CDaoTableDefInfo Structure	NCCALCSIZE_PARAMS Structure
CDaoWorkspaceInfo Structure	PAINTSTRUCT Structure
CODBCFieldInfo Structure	POINT Structure
COLORADJUSTMENT Structure	RECT Structure
COMPAREITEMSTRUCT Structure	RGNDATA Structure
CREATESTRUCT Structure	SOCKADDR Structure
DELETEITEMSTRUCT Structure	SOCKADDR_IN Structure
DEVNAMES Structure	SYSTEMTIME Structure
DHtmlUrlEventMapEntry Structure	WINDOWPLACEMENT Structure
DRAWITEMSTRUCT Structure	WINDOWPOS Structure

EXTENSION_CONTROL_BLOCK Structure	WSADATA Structure
FILETIME Structure	XFORM Structure

See Also

Other Resources

Structures, Styles, Callbacks, and Message Maps

ABC Structure

The **ABC** structure contains the width of a character in a TrueType font.

```
typedef struct _ABC { /* abc */
   int abcA;
   UINT abcB;
   int abcC;
} ABC;
```

Parameters

abcA

Specifies the A spacing of the character. The A spacing is the distance to add to the current position before drawing the character glyph.

abcB

Specifies the B spacing of the character. The B spacing is the width of the drawn portion of the character glyph.

abcC

Specifies the C spacing of the character. The C spacing is the distance to add to the current position to provide white space to the right of the character glyph.

Remarks

The total width of a character is the summation of the A, B, and C spaces. Either the A or the C space can be negative to indicate underhangs or overhangs.

See Also

Reference

CDC::GetCharABCWidths

Other Resources

Structures, Styles, Callbacks, and Message Maps

ABCFLOAT Structure

The **ABCFLOAT** structure contains the A, B, and C widths of a font character.

```
typedef struct _ABCFLOAT { /* abcf */
   FLOAT abcfA;
   FLOAT abcfB;
   FLOAT abcfC;
} ABCFLOAT;
```

Parameters

abcfA

Specifies the A spacing of the character. The A spacing is the distance to add to the current position before drawing the character glyph.

abcfB

Specifies the B spacing of the character. The B spacing is the width of the drawn portion of the character glyph.

abcfC

Specifies the C spacing of the character. The C spacing is the distance to add to the current position to provide white space to the right of the character glyph.

Remarks

The A, B, and C widths are measured along the base line of the font. The character increment (total width) of a character is the sum of the A, B, and C spaces. Either the A or the C space can be negative to indicate underhangs or overhangs.

See Also

Reference

CDC::GetCharABCWidths

Other Resources

Structures, Styles, Callbacks, and Message Maps

AFX_EXTENSION_MODULE Structure

The **AFX_EXTENSION_MODULE** is used during initialization of MFC extension DLLs to hold the state of extension DLL module.

```
struct AFX_EXTENSION_MODULE
{
   BOOL bInitialized;
   HMODULE hModule;
   HMODULE hResource;
   CRuntimeClass* pFirstSharedClass;
   COleObjectFactory* pFirstSharedFactory;
};
```

Parameters

bInitialized

TRUE if the DLL module has been initialized with **AfxInitExtensionModule**.

hModule

Specifies the handle of the DLL module.

hResource

Specifies the handle of the DLL custom resource module.

pFirstSharedClass

A pointer to information (the **CRuntimeClass** structure) about the DLL module's first runtime class. Used to provide the start of the runtime class list.

pFirstSharedFactory

A pointer to the DLL module's first object factory (a **COleObjectFactory** object). Used to provide the start of the class factory list.

Remarks

MFC extension DLLs need to do two things in their **DllMain** function:

- Call AfxInitExtensionModule and check the return value.
- Create a **CDynLinkLibrary** object if the DLL will be exporting CRuntimeClass objects or has its own custom resources.

The **AFX_EXTENSION_MODULE** structure is used to hold a copy of the extension DLL module state, including a copy of the runtime class objects that have been initialized by the extension DLL as part of normal static object construction executed before **DllMain** is entered. For example:

```
static AFX_EXTENSION_MODULE extensionDLL;
extern "C" int APIENTRY
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID)
{
   // initialize this DLL's extension module
   VERIFY(AfxInitExtensionModule(extensionDLL, hInstance));
```

The module information stored in the **AFX_EXTENSION_MODULE** structure can be copied into the **CDynLinkLibrary** object. For example:

```
// CDynLinkLibrary class
IMPLEMENT_DYNAMIC(CDynLinkLibrary, CCmdTarget)
// Constructor
CDynLinkLibrary::CDynLinkLibrary(AFX_EXTENSION_MODULE& state, BOOL bSystem)
{
#ifndef _AFX_NO_OLE_SUPPORT
   m_factoryList.Construct(offsetof(COleObjectFactory, m_pNextFactory));
```

```
#endif
   m_classList.Construct(offsetof(CRuntimeClass, m_pNextClass));

   // copy info from AFX_EXTENSION_MODULE struct
   ASSERT(state.hModule != NULL);
   m_hModule = state.hModule;
   m_hResource = state.hResource;
   m_classList.m_pHead = state.pFirstSharedClass;
#ifndef _AFX_NO_OLE_SUPPORT
   m_factoryList.m_pHead = state.pFirstSharedFactory;
#endif
   m_bSystem = bSystem;
```

See Also

Reference

AfxInitExtensionModule

AfxTermExtensionModule

Other Resources

Structures, Styles, Callbacks, and Message Maps

BITMAP Structure

The **BITMAP** structure defines the height, width, color format, and bit values of a logical bitmap**.**

```
typedef struct tagBITMAP {  /* bm */
   int bmType;
   int bmWidth;
   int bmHeight;
   int bmWidthBytes;
   BYTE bmPlanes;
   BYTE bmBitsPixel;
   LPVOID bmBits;
} BITMAP;
```

Parameters

bmType

Specifies the bitmap type. For logical bitmaps, this member must be 0.

bmWidth

Specifies the width of the bitmap in pixels. The width must be greater than 0.

bmHeight

Specifies the height of the bitmap in raster lines. The height must be greater than 0.

bmWidthBytes

Specifies the number of bytes in each raster line. This value must be an even number since the graphics device interface (GDI) assumes that the bit values of a bitmap form an array of integer (2-byte) values. In other words, **bmWidthBytes** * 8 must be the next multiple of 16 greater than or equal to the value obtained when the **bmWidth** member is multiplied by the **bmBitsPixel** member.

bmPlanes

Specifies the number of color planes in the bitmap.

bmBitsPixel

Specifies the number of adjacent color bits on each plane needed to define a pixel.

bmBits

Points to the location of the bit values for the bitmap. The **bmBits** member must be a long pointer to an array of 1-byte values.

Remarks

The currently used bitmap formats are monochrome and color. The monochrome bitmap uses a 1-bit, 1-plane format. Each scan is a multiple of 16 bits.

Scans are organized as follows for a monochrome bitmap of height *n*:

```
Scan 0
Scan 1
.
.
.
Scan n-2
Scan n-1
```

The pixels on a monochrome device are either black or white. If the corresponding bit in the bitmap is 1, the pixel is turned on (white). If the corresponding bit in the bitmap is 0, the pixel is turned off (black).

All devices support bitmaps that have the **RC_BITBLT** bit set in the **RASTERCAPS** index of the CDC::GetDeviceCaps member

function.

Each device has its own unique color format. In order to transfer a bitmap from one device to another, use the GetDIBits and SetDIBits Windows functions.

See Also

Reference

CBitmap::CreateBitmapIndirect

Other Resources

Structures, Styles, Callbacks, and Message Maps

BITMAPINFO Structure

The **BITMAPINFO** structure defines the dimensions and color information for a Windows device-independent bitmap (DIB).

```
typedef struct tagBITMAPINFO {
   BITMAPINFOHEADER bmiHeader;
   RGBQUAD bmiColors[1];
} BITMAPINFO;
```

Parameters

bmiHeader

Specifies a **BITMAPINFOHEADER** structure that contains information about the dimensions and color format of a device-independent bitmap.

bmiColors

Specifies an array of **RGBQUAD** or **DWORD** data types that define the colors in the bitmap.

Remarks

A device-independent bitmap consists of two distinct parts: a **BITMAPINFO** structure describing the dimensions and colors of the bitmap, and an array of bytes defining the pixels of the bitmap. The bits in the array are packed together, but each scan line must be padded with zeroes to end on a **LONG** boundary. If the height is positive, the origin of the bitmap is the lower-left corner. If the height is negative, the origin is the upper-left corner.

The **biBitCount** member of the **BITMAPINFOHEADER** structure determines the number of bits that define each pixel and the maximum number of colors in the bitmap. This member can be one of the following values:

- The bitmap is monochrome, and the **bmiColors** member contains two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the **bmiColors** table; if the bit is set, the pixel has the color of the second entry in the table.
- The bitmap has a maximum of 16 colors, and the **bmiColors** member contains up to 16 entries. Each pixel in the bitmap is represented by a 4-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, the byte represents two pixels. The first pixel contains the color in the second table entry, and the second pixel contains the color in the sixteenth table entry.
- The bitmap has a maximum of 256 colors, and the **bmiColors** member contains up to 256 entries. In this case, each byte in the array represents a single pixel.
- The bitmap has a maximum of 216 colors. The **biCompression** member of the **BITMAPINFOHEADER** must be **BI_BITFIELDS**. The **bmiColors** member contains 3 **DWORD** color masks which specify the red, green, and blue components, respectively, of each pixel. Bits set in the **DWORD** mask must be contiguous and should not overlap the bits of another mask. All the bits in the pixel do not have to be used. Each **WORD** in the array represents a single pixel.
- The bitmap has a maximum of 224 colors, and the **bmiColors** member is **NULL**. Each 3-byte triplet in the bitmap array represents the relative intensities of blue, green, and red, respectively, of a pixel.
- The bitmap has a maximum of 232 colors. The **biCompression** member of the **BITMAPINFOHEADER** must be **BI_BITFIELDS**. The **bmiColors** member contains three **DWORD** color masks which specify the red, green, and blue components, respectively, of each pixel. Bits set in the **DWORD** mask must be contiguous and should not overlap the bits of another mask. All the bits in the pixel do not have to be used. Each **DWORD** in the array represents a single pixel.

The **biClrUsed** member of the **BITMAPINFOHEADER** structure specifies the number of color indices in the color table that are actually used by the bitmap. If the **biClrUsed** member is set to zero, the bitmap uses the maximum number of colors corresponding to the value of the **biBitCount** member.

The colors in the **bmiColors** table should appear in order of importance. Alternately, for functions that use DIBs, the **bmiColors** member can be an array of 16-bit unsigned integers that specify indices into the currently realized logical palette, instead of explicit RGB values. In this case, an application using the bitmap must call the Windows DIB functions (CreateDIBitmap, CreateDIBPatternBrush, and CreateDIBSection) with the **iUsage** parameter set to **DIB_PAL_COLORS**.

If the bitmap is a packed bitmap (that is, a bitmap in which the bitmap array immediately follows the **BITMAPINFO** header and

which is referenced by a single pointer), the **biClrUsed** member must be set to an even number when using the **DIB_PAL_COLORS** mode so the DIB bitmap array starts on a **DWORD** boundary.

Note
The **bmiColors** member should not contain palette indices if the bitmap is to be stored in a file or transferred to another app lication. Unless the application has exclusive use and control of the bitmap, the bitmap color table should contain explicit RGB values.

See Also

Reference

CBrush::CreateDIBPatternBrush

Other Resources

Structures, Styles, Callbacks, and Message Maps

CDaoDatabaseInfo Structure

The **CDaoDatabaseInfo** structure contains information about a database object defined for data access objects (DAO).

```
struct CDaoDatabaseInfo
{
   CString m_strName;       // Primary
   BOOL m_bUpdatable;       // Primary
   BOOL m_bTransactions;    // Primary
   CString m_strVersion;    // Secondary
   long m_lCollatingOrder;  // Secondary
   short m_nQueryTimeout;   // Secondary
   CString m_strConnect;    // All
};
```

Parameters

m_strName

Uniquely names the database object. To directly retrieve this property, call CDaoDatabase::GetName. For details, see the topic "Name Property" in DAO Help.

m_bUpdatable

Indicates whether changes can be made to the database. To directly retrieve this property, call CDaoDatabase::CanUpdate. For details, see the topic "Updatable Property" in DAO Help.

m_bTransactions

Indicates whether a data source supports transactions — the recording of a series of changes that can later be rolled back (canceled) or committed (saved). If a database is based on the Microsoft Jet database engine, the Transactions property is nonzero and you can use transactions. Other database engines may not support transactions. To directly retrieve this property, call CDaoDatabase::CanTransact. For details, see the topic "Transactions Property" in DAO Help.

m_strVersion

Indicates the version of the Microsoft Jet database engine. To retrieve the value of this property directly, call the database object's GetVersion member function. For details, see the topic "Version Property" in DAO Help.

m_lCollatingOrder

Specifies the sequence of the sort order in text for string comparison or sorting. Possible values include:

- **dbSortGeneral** Use the General (English, French, German, Portuguese, Italian, and Modern Spanish) sort order.
- **dbSortArabic** Use the Arabic sort order.
- **dbSortCyrillic** Use the Russian sort order.
- **dbSortCzech** Use the Czech sort order.
- **dbSortDutch** Use the Dutch sort order.
- **dbSortGreek** Use the Greek sort order.
- **dbSortHebrew** Use the Hebrew sort order.
- **dbSortHungarian** Use the Hungarian sort order.
- **dbSortIcelandic** Use the Icelandic sort order.
- **dbSortNorwdan** Use the Norwegian or Danish sort order.
- **dbSortPDXIntl** Use the Paradox International sort order.
- **dbSortPDXNor** Use the Paradox Norwegian or Danish sort order.
- **dbSortPDXSwe** Use the Paradox Swedish or Finnish sort order.

- **dbSortPolish** Use the Polish sort order.
- **dbSortSpanish** Use the Spanish sort order.
- **dbSortSwedFin** Use the Swedish or Finnish sort order.
- **dbSortTurkish** Use the Turkish sort order.
- **dbSortUndefined** The sort order is undefined or unknown.

For more information, see the topic "Customizing Windows Registry Settings for Data Access" in DAO Help.

m_nQueryTimeout

The number of seconds the Microsoft Jet database engine waits before a timeout error occurs when a query is run on an ODBC database. The default timeout value is 60 seconds. When QueryTimeout is set to 0, no timeout occurs; this can cause the program to hang. To retrieve the value of this property directly, call the database object's GetQueryTimeout member function. For details, see the topic "QueryTimeout Property" in DAO Help.

m_strConnect

Provides information about the source of an open database. For information about connect strings, and for information about retrieving the value of this property directly, see the CDaoDatabase::GetConnect member function. For more information, see the topic "Connect Property" in DAO Help.

Remarks

The database is a DAO object underlying an MFC object of class CDaoDatabase. The references to Primary, Secondary, and All above indicate how the information is returned by the CDaoWorkspace::GetDatabaseInfo member function.

Information retrieved by the CDaoWorkspace::GetDatabaseInfo member function is stored in a **CDaoDatabaseInfo** structure. Call **GetDatabaseInfo** for the **CDaoWorkspace** object in whose Databases collection the database object is stored. **CDaoDatabaseInfo** also defines a **Dump** member function in debug builds. You can use **Dump** to dump the contents of a **CDaoDatabaseInfo** object.

See Also

Reference

CDaoWorkspace Class
CDaoDatabase Class
CDaoWorkspace::GetDatabaseCount

Other Resources

Structures, Styles, Callbacks, and Message Maps

CDaoErrorInfo Structure

The **CDaoErrorInfo** structure contains information about an error object defined for data access objects (DAO).

```
struct CDaoErrorInfo
{
   long m_lErrorCode;
   CString m_strSource;
   CString m_strDescription;
   CString m_strHelpFile;
   long m_lHelpContext;
};
```

Parameters

m_lErrorCode

A numeric DAO error code. See the topic "Trappable Data Access Errors" in DAO Help.

m_strSource

The name of the object or application that originally generated the error. The Source property specifies a string expression representing the object that originally generated the error; the expression is usually the object's class name. For details, see the topic "Source Property" in DAO Help.

m_strDescription

A descriptive string associated with an error. For details, see the topic "Description Property" in DAO Help.

m_strHelpFile

A fully qualified path to a Microsoft Windows Help file. For details, see the topic "HelpContext, HelpFile Properties" in DAO Help.

m_lHelpContext

A context ID for a topic in a Microsoft Windows Help file. For details, see the topic "HelpContext, HelpFile Properties" in DAO Help.

Remarks

MFC does not encapsulate DAO error objects in a class. Instead, the CDaoException class supplies an interface for accessing the Errors collection contained in the DAO **DBEngine** object, the object that also contains all workspaces. When an MFC DAO operation throws a **CDaoException** object that you catch, MFC fills a **CDaoErrorInfo** structure and stores it in the exception object's m_pErrorInfo member. (If you choose to call DAO directly, you must call the exception object's GetErrorInfo member function yourself to fill **m_pErrorInfo**.)

For more information about handling DAO errors, see the article Exceptions: Database Exceptions. For related information, see the topic "Error Object" in DAO Help.

Information retrieved by the CDaoException::GetErrorInfo member function is stored in a **CDaoErrorInfo** structure. Examine the m_pErrorInfo data member from a **CDaoException** object that you catch in an exception handler, or call **GetErrorInfo** from a **CDaoException** object that you create explicitly in order to check errors that might have occurred during a direct call to the DAO interfaces. **CDaoErrorInfo** also defines a **Dump** member function in debug builds. You can use **Dump** to dump the contents of a **CDaoErrorInfo** object.

See Also

Reference

CDaoException Class

Other Resources

Structures, Styles, Callbacks, and Message Maps

CDaoFieldInfo Structure

The **CDaoFieldInfo** structure contains information about a field object defined for data access objects (DAO).

```
struct CDaoFieldInfo
{
   CString m_strName;           // Primary
   short m_nType;               // Primary
   long m_lSize;                // Primary
   long m_lAttributes;          // Primary
   short m_nOrdinalPosition;    // Secondary
   BOOL m_bRequired;            // Secondary
   BOOL m_bAllowZeroLength;     // Secondary
   long m_lCollatingOrder;      // Secondary
   CString m_strForeignName;    // Secondary
   CString m_strSourceField;    // Secondary
   CString m_strSourceTable;    // Secondary
   CString m_strValidationRule; // All
   CString m_strValidationText; // All
   CString m_strDefaultValue;   // All
};
```

Parameters

m_strName

Uniquely names the field object. For details, see the topic "Name Property" in DAO Help.

m_nType

A value that indicates the data type of the field. For details, see the topic "Type Property" in DAO Help. The value of this property can be one of the following:

- **dbBoolean** Yes/No, same as **TRUE**/**FALSE**
- **dbByte** Byte
- **dbInteger** Short
- **dbLong** Long
- **dbCurrency** Currency; see MFC class COleCurrency
- **dbSingle** Single
- **dbDouble** Double
- **dbDate** Date/Time; see MFC class COleDateTime
- **dbText** Text; see MFC class CString
- **dbLongBinary** Long Binary (OLE Object); you might want to use MFC class CByteArray instead of class **CLongBinary** as **CByteArray** is richer and easier to use.
- **dbMemo** Memo; see MFC class **CString**
- **dbGUID** A Globally Unique Identifier/Universally Unique Identifier used with remote procedure calls. For more information, see the topic "Type Property" in DAO Help.

Note
Do not use string data types for binary data. This causes your data to pass through the Unicode/ANSI translation layer, res ulting in increased overhead and possibly unexpected translation.

m_lSize

A value that indicates the maximum size, in bytes, of a DAO field object that contains text or the fixed size of a field object

that contains text or numeric values. For details, see the topic "Size Property" in DAO Help. Sizes can be one of the following values:

Type	Size (Bytes)	Description
dbBoolean	1 byte	Yes/No (same as True/False)
dbByte	1	Byte
dbInteger	2	Integer
dbLong	4	Long
dbCurrency	8	Currency (COleCurrency)
dbSingle	4	Single
dbDouble	8	Double
dbDate	8	Date/Time (COleDateTime)
dbText	1 - 255	Text (CString)
dbLongBinary	0	Long Binary (OLE Object; CByteArray; use instead of **CLongBinary**)
dbMemo	0	Memo (CString)
dbGUID	16	A Globally Unique Identifier/Universally Unique Identifier used with remote procedure calls.

m_lAttributes

Specifies characteristics of a field object contained by a tabledef, recordset, querydef, or index object. The value returned can be a sum of these constants, created with the C++ bitwise-OR (**|**) operator:

- **dbFixedField** The field size is fixed (default for Numeric fields).
- **dbVariableField** The field size is variable (Text fields only).
- **dbAutoIncrField** The field value for new records is automatically incremented to a unique long integer that cannot be changed. Only supported for Microsoft Jet database tables.
- **dbUpdatableField** The field value can be changed.
- **dbDescending** The field is sorted in descending (Z - A or 100 - 0) order (applies only to a field object in a Fields collection of an index object; in MFC, index objects are themselves contained in tabledef objects). If you omit this constant, the field is sorted in ascending (A - Z or 0 - 100) order (default).

When checking the setting of this property, you can use the C++ bitwise-AND operator (**&**) to test for a specific attribute. When setting multiple attributes, you can combine them by combining the appropriate constants with the bitwise-OR (**|**) operator. For details, see the topic "Attributes Property" in DAO Help.

m_nOrdinalPosition

A value that specifies the numeric order in which you want a field represented by a DAO field object to be displayed relative to other fields. You can set this property with CDaoTableDef::CreateField. For details, see the topic "OrdinalPosition Property" in DAO Help.

m_bRequired

Indicates whether a DAO field object requires a non-Null value. If this property is **TRUE**, the field does not allow a Null value. If Required is set to **FALSE**, the field can contain Null values as well as values that meet the conditions specified by the AllowZeroLength and ValidationRule property settings. For details, see the topic "Required Property" in DAO Help. You can set this property for a tabledef with CDaoTableDef::CreateField.

m_bAllowZeroLength

Indicates whether an empty string ("") is a valid value of a DAO field object with a Text or Memo data type. If this property is **TRUE**, an empty string is a valid value. You can set this property to **FALSE** to ensure that you cannot use an empty string to set the value of a field. For details, see the topic "AllowZeroLength Property" in DAO Help. You can set this property for a tabledef with CDaoTableDef::CreateField.

m_lCollatingOrder

Specifies the sequence of the sort order in text for string comparison or sorting. For details, see the topic "Customizing Windows Registry Settings for Data Access" in DAO Help. For a list of the possible values returned, see the **m_lCollatingOrder** member of the CDaoDatabaseInfo structure. You can set this property for a tabledef with CDaoTableDef::CreateField.

m_strForeignName

A value that, in a relation, specifies the name of the DAO field object in a foreign table that corresponds to a field in a primary table. For details, see the topic "ForeignName Property" in DAO Help.

m_strSourceField

Indicates the name of the field that is the original source of the data for a DAO field object contained by a tabledef, recordset, or querydef object. This property indicates the original field name associated with a field object. For example, you could use this property to determine the original source of the data in a query field whose name is unrelated to the name of the field in the underlying table. For details, see the topic "SourceField, SourceTable Properties" in DAO Help. You can set this property for a tabledef with CDaoTableDef::CreateField.

m_strSourceTable

Indicates the name of the table that is the original source of the data for a DAO field object contained by a tabledef, recordset, or querydef object. This property indicates the original table name associated with a field object. For example, you could use this property to determine the original source of the data in a query field whose name is unrelated to the name of the field in the underlying table. For details, see the topic "SourceField, SourceTable Properties" in DAO Help. You can set this property for a tabledef with CDaoTableDef::CreateField.

m_strValidationRule

A value that validates the data in a field as it is changed or added to a table. For details, see the topic "ValidationRule Property" in DAO Help. You can set this property for a tabledef with CDaoTableDef::CreateField.

For related information about tabledefs, see the **m_strValidationRule** member of the CDaoTableDefInfo structure.

m_strValidationText

A value that specifies the text of the message that your application displays if the value of a DAO field object does not satisfy the validation rule specified by the ValidationRule property setting. For details, see the topic "ValidationText Property" in DAO Help. You can set this property for a tabledef with CDaoTableDef::CreateField.

m_strDefaultValue

The default value of a DAO field object. When a new record is created, the DefaultValue property setting is automatically entered as the value for the field. For details, see the topic "DefaultValue Property" in DAO Help. You can set this property for a tabledef with CDaoTableDef::CreateField.

Remarks

The references to Primary, Secondary, and All above indicate how the information is returned by the **GetFieldInfo** member function in classes CDaoTableDef, CDaoQueryDef, and CDaoRecordset.

Field objects are not represented by an MFC class. Instead, the DAO objects underlying MFC objects of the following classes contain collections of field objects: CDaoTableDef, CDaoRecordset, and CDaoQueryDef. These classes supply member functions to access some individual items of field information, or you can access them all at once with a **CDaoFieldInfo** object by calling the **GetFieldInfo** member function of the containing object.

Besides its use for examining object properties, you can also use **CDaoFieldInfo** to construct an input parameter for creating new fields in a tabledef. Simpler options are available for this task, but if you want finer control, you can use the version of CDaoTableDef::CreateField that takes a **CDaoFieldInfo** parameter.

Information retrieved by the **GetFieldInfo** member function (of the class that contains the field) is stored in a **CDaoFieldInfo** structure. Call the **GetFieldInfo** member function of the containing object in whose Fields collection the field object is stored. **CDaoFieldInfo** also defines a **Dump** member function in debug builds. You can use **Dump** to dump the contents of a

CDaoFieldInfo object.

See Also

Reference

CDaoTableDef::GetFieldInfo
CDaoRecordset::GetFieldInfo
CDaoQueryDef::GetFieldInfo

Other Resources

Structures, Styles, Callbacks, and Message Maps

CDaoIndexInfo Structure

The **CDaoIndexInfo** structure contains information about an index object defined for data access objects (DAO).

```
struct CDaoIndexInfo {
   CDaoIndexInfo( );                  // Constructor
   CString m_strName;                 // Primary
   CDaoIndexFieldInfo* m_pFieldInfos; // Primary
   short m_nFields;                   // Primary
   BOOL m_bPrimary;                   // Secondary
   BOOL m_bUnique;                    // Secondary
   BOOL m_bClustered;                 // Secondary
   BOOL m_bIgnoreNulls;               // Secondary
   BOOL m_bRequired;                  // Secondary
   BOOL m_bForeign;                   // Secondary
   long m_lDistinctCount;             // All

   // Below the // Implementation comment:
   // Destructor, not otherwise documented
};
```

Parameters

m_strName

Uniquely names the field object. For details, see the topic "Name Property" in DAO Help.

m_pFieldInfos

A pointer to an array of CDaoIndexFieldInfo objects indicating which tabledef or recordset fields are key fields in an index. Each object identifies one field in the index. The default index ordering is ascending. An index object can have one or more fields representing index keys for each record. These can be ascending, descending, or a combination.

m_nFields

The number of fields stored in **m_pFieldInfos**.

m_bPrimary

If the Primary property is **TRUE**, the index object represents a primary index. A primary index consists of one or more fields that uniquely identify all records in a table in a predefined order. Because the index field must be unique, the Unique property of the Index object is also set to **TRUE** in DAO. If the primary index consists of more than one field, each field can contain duplicate values, but each combination of values from all the indexed fields must be unique. A primary index consists of a key for the table and usually contains the same fields as the primary key.

When you set a primary key for a table, the primary key is automatically defined as the primary index for the table. For more information, see the topics "Primary Property" and "Unique Property" in DAO Help.

Note
There can be, at most, one primary index on a table.

m_bUnique

Indicates whether an index object represents a unique index for a table. If this property is **TRUE**, the index object represents an index that is unique. A unique index consists of one or more fields that logically arrange all records in a table in a unique, predefined order. If the index consists of one field, values in that field must be unique for the entire table. If the index consists of more than one field, each field can contain duplicate values, but each combination of values from all the indexed fields must be unique.

If both the Unique and Primary properties of an index object are set to **TRUE**, the index is unique and primary: It uniquely identifies all records in the table in a predefined, logical order. If the Primary property is set to **FALSE**, the index is a secondary index. Secondary indexes (both key and nonkey) logically arrange records in a predefined order without serving as an identifier for records in the table.

For more information, see the topics "Primary Property" and "Unique Property" in DAO Help.

m_bClustered

Indicates whether an index object represents a clustered index for a table. If this property is **TRUE**, the index object represents a clustered index; otherwise, it does not. A clustered index consists of one or more nonkey fields that, taken together, arrange all records in a table in a predefined order. With a clustered index, the data in the table is literally stored in the order specified by the clustered index. A clustered index provides efficient access to records in a table. For more information, see the topic "Clustered Property" in DAO Help.

Note
The Clustered property is ignored for databases that use the Microsoft Jet database engine because the Jet database engine does not support clustered indexes.

m_bIgnoreNulls

Indicates whether there are index entries for records that have Null values in their index fields. If this property is **TRUE**, fields with Null values do not have an index entry. To make searching for records using a field faster, you can define an index for the field. If you allow Null entries in an indexed field and expect many of the entries to be Null, you can set the IgnoreNulls property for the index object to **TRUE** to reduce the amount of storage space that the index uses. The IgnoreNulls property setting and the Required property setting together determine whether a record with a Null index value has an index entry, as the following table shows.

IgnoreNulls	**Required**	**Null in index field**
True	False	Null value allowed; no index entry added.
False	False	Null value allowed; index entry added.
True or False	True	Null value not allowed; no index entry added.

For more information, see the topic "IgnoreNulls Property" in DAO Help.

m_bRequired

Indicates whether a DAO index object requires a non-Null value. If this property is **TRUE**, the index object does not allow a Null value. For more information, see the topic "Required Property" in DAO Help.

Tip
When you can set this property for either a DAO index object or a field object (contained by a tabledef, recordset, or querydef object), set it for the field object. The validity of the property setting for a field object is checked before that of an index object.

m_bForeign

Indicates whether an index object represents a foreign key in a table. If this property is **TRUE**, the index represents a foreign key in a table. A foreign key consists of one or more fields in a foreign table that uniquely identify a row in a primary table. The Microsoft Jet database engine creates an index object for the foreign table and sets the Foreign property when you create a relationship that enforces referential integrity. For more information, see the topic "Foreign Property" in DAO Help.

m_lDistinctCount

Indicates the number of unique values for the index object that are included in the associated table. Check the DistinctCount property to determine the number of unique values, or keys, in an index. Any key is counted only once, even though there may be multiple occurrences of that value if the index permits duplicate values. This information is useful in applications that attempt to optimize data access by evaluating index information. The number of unique values is also known as the cardinality of an index object. The DistinctCount property will not always reflect the actual number of keys at a particular time. For example, a change caused by a transaction rollback will not be reflected immediately in the DistinctCount property. For more information, see the topic "DistinctCount Property" in DAO Help.

Remarks

The references to Primary, Secondary, and All above indicate how the information is returned by the **GetIndexInfo** member function in classes CDaoTableDef and CDaoRecordset.

Index objects are not represented by an MFC class. Instead, DAO objects underlying MFC objects of class CDaoTableDef or CDaoRecordset contain a collection of index objects, called the Indexes collection. These classes supply member functions to access individual items of index information, or you can access them all at once with a **CDaoIndexInfo** object by calling the **GetIndexInfo** member function of the containing object.

CDaoIndexInfo has a constructor and a destructor in order to properly allocate and deallocate the index field information in **m_pFieldInfos**.

Information retrieved by the **GetIndexInfo** member function of a tabledef object is stored in a **CDaoIndexInfo** structure. Call the **GetIndexInfo** member function of the containing tabledef object in whose Indexes collection the index object is stored. **CDaoIndexInfo** also defines a **Dump** member function in debug builds. You can use **Dump** to dump the contents of a **CDaoIndexInfo** object.

See Also

Reference

CDaoTableDef::GetIndexInfo

Other Resources

Structures, Styles, Callbacks, and Message Maps

CDaoIndexFieldInfo Structure

The **CDaoIndexFieldInfo** structure contains information about an index field object defined for data access objects (DAO).

```
struct CDaoIndexFieldInfo
{
   CString m_strName;          // Primary
   BOOL m_bDescending;         // Primary
};
```

Parameters

m_strName

Uniquely names the index field object. For details, see the topic "Name Property" in DAO Help.

m_bDescending

Indicates the index ordering defined by the index object. **TRUE** if the order is descending.

Remarks

An index object can have a number of fields, indicating which fields a tabledef (or a recordset based on a table) is indexed on. The references to Primary above indicate how the information is returned in the **m_pFieldInfos** member of a CDaoIndexInfo object obtained by calling the **GetIndexInfo** member function of class CDaoTableDef or CDaoRecordset.

Index objects and index field objects are not represented by an MFC class. Instead, the DAO objects underlying MFC objects of class CDaoTableDef or CDaoRecordset contain a collection of index objects, called the Indexes collection. Each index object, in turn, contains a collection of field objects. These classes supply member functions to access individual items of index information, or you can access them all at once with a **CDaoIndexInfo** object by calling the **GetIndexInfo** member function of the containing object. The **CDaoIndexInfo** object, then, has a data member, **m_pFieldInfos**, that points to an array of **CDaoIndexFieldInfo** objects.

Call the **GetIndexInfo** member function of the containing tabledef or recordset object in whose Indexes collection is stored the index object you are interested in. Then access the **m_pFieldInfos** member of the CDaoIndexInfo object. The length of the **m_pFieldInfos** array is stored in **m_nFields**. **CDaoIndexFieldInfo** also defines a **Dump** member function in debug builds. You can use **Dump** to dump the contents of a **CDaoIndexFieldInfo** object.

See Also

Reference

CDaoTableDef::GetIndexInfo

CDaoRecordset::GetIndexInfo

Other Resources

Structures, Styles, Callbacks, and Message Maps

CDaoParameterInfo Structure

The **CDaoParameterInfo** structure contains information about a parameter object defined for data access objects (DAO).

```
struct CDaoParameterInfo
{
   CString m_strName;       // Primary
   short m_nType;           // Primary
   ColeVariant m_varValue;  // Secondary
};
```

Parameters

m_strName

Uniquely names the parameter object. For more information, see the topic "Name Property" in DAO Help.

m_nType

A value that indicates the data type of a parameter object. For a list of the possible values, see the **m_nType** member of the CDaoFieldInfo structure. For more information, see the topic "Type Property" in DAO Help.

m_varValue

The value of the parameter, stored in a COleVariant object.

Remarks

The references to Primary and Secondary above indicate how the information is returned by the GetParameterInfo member function in class **CDaoQueryDef**.

MFC does not encapsulate DAO parameter objects in a class. DAO querydef objects underlying MFC **CDaoQueryDef** objects store parameters in their Parameters collections. To access the parameter objects in a CDaoQueryDef object, call the querydef object's **GetParameterInfo** member function for a particular parameter name or an index into the Parameters collection. You can use the CDaoQueryDef::GetParameterCount member function in conjunction with **GetParameterInfo** to loop through the Parameters collection.

Information retrieved by the CDaoQueryDef::GetParameterInfo member function is stored in a **CDaoParameterInfo** structure. Call **GetParameterInfo** for the querydef object in whose Parameters collection the parameter object is stored.

Note
If you want to get or set only the value of a parameter, use the GetParamValue and SetParamValue member functions of class **CDaoRecordset**.

CDaoParameterInfo also defines a **Dump** member function in debug builds. You can use **Dump** to dump the contents of a **CDaoParameterInfo** object.

See Also

Reference

CDaoQueryDef Class

Other Resources

Structures, Styles, Callbacks, and Message Maps

CDaoQueryDefInfo Structure

The **CDaoQueryDefInfo** structure contains information about a querydef object defined for data access objects (DAO).

```
struct CDaoQueryDefInfo
{
   CString m_strName;               // Primary
   short m_nType;                   // Primary
   COleDateTime m_dateCreated;      // Secondary
   COleDateTime m_dateLastUpdated;  // Secondary
   BOOL m_bUpdatable;               // Secondary
   BOOL m_bReturnsRecords;          // Secondary
   CString m_strSQL;                // All
   CString m_strConnect;            // All
   short m_nODBCTimeout;            // All
};
```

Parameters

m_strName

Uniquely names the querydef object. For more information, see the topic "Name Property" in DAO Help. Call CDaoQueryDef::GetName to retrieve this property directly.

m_nType

A value that indicates the operational type of a querydef object. The value can be one of the following:

- **dbQSelect** Select — the query selects records.
- **dbQAction** Action — the query moves or changes data but does not return records.
- **dbQCrosstab** Crosstab — the query returns data in a spreadsheet-like format.
- **dbQDelete** Delete — the query deletes a set of specified rows.
- **dbQUpdate** Update — the query changes a set of records.
- **dbQAppend** Append — the query adds new records to the end of a table or query.
- **dbQMakeTable** Make-table — the query creates a new table from a recordset.
- **dbQDDL** Data-definition — the query affects the structure of tables or their parts.
- **dbQSQLPassThrough** Pass-through — the SQL statement is passed directly to the database backend, without intermediate processing.
- **dbQSetOperation** Union — the query creates a snapshot-type recordset object containing data from all specified records in two or more tables with any duplicate records removed. To include the duplicates, add the keyword **ALL** in the querydef's SQL statement.
- **dbQSPTBulk** Used with **dbQSQLPassThrough** to specify a query that does not return records.

Note
To create a SQL pass-through query, you do not set the **dbQSQLPassThrough** constant. This is set automatically by the Microsoft Jet database engine when you create a querydef object and set the Connect property.

For more information, see the topic "Type Property" in DAO Help.

m_dateCreated

The date and time the querydef was created. To directly retrieve the date the querydef was created, call the GetDateCreated member function of the **CDaoTableDef** object associated with the table. See Comments below for more information. Also see the topic "DateCreated, LastUpdated Properties" in DAO Help.

m_dateLastUpdated

The date and time of the most recent change made to the querydef. To directly retrieve the date the table was last updated, call the GetDateLastUpdated member function of the querydef. See Comments below for more information. And see the topic "DateCreated, LastUpdated Properties" in DAO Help.

m_bUpdatable

Indicates whether changes can be made to a querydef object. If this property is **TRUE**, the querydef is updatable; otherwise, it is not. Updatable means the querydef object's query definition can be changed. The Updatable property of a querydef object is set to **TRUE** if the query definition can be updated, even if the resulting recordset is not updatable. To retrieve this property directly, call the querydef's CanUpdate member function. For more information, see the topic "Updatable Property" in DAO Help.

m_bReturnsRecords

Indicates whether a SQL pass-through query to an external database returns records. If this property is **TRUE**, the query returns records. To directly retrieve this property, call CDaoQueryDef::GetReturnsRecords. Not all SQL pass-through queries to external databases return records. For example, a SQL **UPDATE** statement updates records without returning records, while a SQL **SELECT** statement does return records. For more information, see the topic "ReturnsRecords Property" in DAO Help.

m_strSQL

The SQL statement that defines the query executed by a querydef object. The SQL property contains the SQL statement that determines how records are selected, grouped, and ordered when you execute the query. You can use the query to select records to include in a dynaset- or snapshot-type recordset object. You can also define bulk queries to modify data without returning records. You can retrieve the value of this property directly by calling the querydef's GetSQL member function.

m_strConnect

Provides information about the source of a database used in a pass-through query. This information takes the form of a connect string. For more information about connect strings, and for information about retrieving the value of this property directly, see the CDaoDatabase::GetConnect member function.

m_nODBCTimeout

The number of seconds the Microsoft Jet database engine waits before a timeout error occurs when a query is run on an ODBC database. When you're using an ODBC database, such as Microsoft SQL Server, there may be delays because of network traffic or heavy use of the ODBC server. Rather than waiting indefinitely, you can specify how long the Microsoft Jet engine waits before it produces an error. The default timeout value is 60 seconds. You can retrieve the value of this property directly by calling the querydef's GetODBCTimeout member function. For more information, see the topic "ODBCTimeout Property" in DAO Help.

Remarks

The querydef is an object of class CDaoQueryDef. The references to Primary, Secondary, and All above indicate how the information is returned by the GetQueryDefInfo member function in class **CDaoDatabase**.

Information retrieved by the CDaoDatabase::GetQueryDefInfo member function is stored in a **CDaoQueryDefInfo** structure. Call **GetQueryDefInfo** for the database object in whose QueryDefs collection the querydef object is stored. **CDaoQueryDefInfo** also defines a **Dump** member function in debug builds. You can use **Dump** to dump the contents of a **CDaoQueryDefInfo** object. Class **CDaoDatabase** also supplies member functions for directly accessing all of the properties returned in a **CDaoQueryDefInfo** object, so you will probably seldom need to call **GetQueryDefInfo**.

When you append a new field or parameter object to the Fields or Parameters collection of a querydef object, an exception is thrown if the underlying database does not support the data type specified for the new object.

The date and time settings are derived from the computer on which the querydef was created or last updated. In a multiuser environment, users should get these settings directly from the file server using the **net time** command to avoid discrepancies in the DateCreated and LastUpdated property settings.

See Also

Reference

CDaoQueryDef Class

CDaoDatabase Class

Other Resources

Structures, Styles, Callbacks, and Message Maps

CDaoRelationInfo Structure

The **CDaoRelationInfo** structure contains information about a relation defined between fields of two tables in a CDaoDatabase object.

```
struct CDaoRelationInfo
{
   CDaoRelationInfo( );                    // Constructor
   CString m_strName;                      // Primary
   CString m_strTable;                     // Primary
   CString m_strForeignTable;              // Primary
   long m_lAttributes;                     // Secondary
   CDaoRelationFieldInfo* m_pFieldInfos;   // Secondary
   short m_nFields;                        // Secondary
   // Below the // Implementation comment:
   // Destructor, not otherwise documented
};
```

Parameters

m_strName

Uniquely names the relation object. For more information, see the topic "Name Property" in DAO Help.

m_strTable

Names the primary table in the relation.

m_strForeignTable

Names the foreign table in the relation. A foreign table is a table used to contain foreign keys. Generally, you use a foreign table to establish or enforce referential integrity. The foreign table is usually on the many side of a one-to-many relationship. Examples of foreign tables include tables containing codes for the American states or Canadian provinces or customer orders.

m_lAttributes

Contains information about the relation type. The value of this member can be any of the following:

- **dbRelationUnique** Relationship is one-to-one.
- **dbRelationDontEnforce** Relationship is not enforced (no referential integrity).
- **dbRelationInherited** Relationship exists in a noncurrent database that contains the two attached tables.
- **dbRelationLeft** The relationship is a left join. A left outer join includes all of the records from the first (left-hand) of two tables, even if there are no matching values for records in the second (right-hand) table.
- **dbRelationRight** The relationship is a right join. A right outer join includes all of the records from the second (right-hand) of two tables, even if there are no matching values for records in the first (left-hand) table.
- **dbRelationUpdateCascade** Updates will cascade.
- **dbRelationDeleteCascade** Deletions will cascade.

m_pFieldInfos

A pointer to an array of CDaoRelationFieldInfo structures. The array contains one object for each field in the relation. The **m_nFields** data member gives a count of the array elements.

m_nFields

The number of **CDaoRelationFieldInfo** objects in the **m_pFieldInfos** data member.

Remarks

The references to Primary and Secondary above indicate how the information is returned by the GetRelationInfo member function in class **CDaoDatabase**.

Relation objects are not represented by an MFC class. Instead, the DAO object underlying an MFC object of the **CDaoDatabase** class maintains a collection of relation objects: **CDaoDatabase** supplies member functions to access some individual items of relation information, or you can access them all at once with a **CDaoRelationInfo** object by calling the **GetRelationInfo** member function of the containing database object.

Information retrieved by the CDaoDatabase::GetRelationInfo member function is stored in a **CDaoRelationInfo** structure. **CDaoRelationInfo** also defines a **Dump** member function in debug builds. You can use **Dump** to dump the contents of a **CDaoRelationInfo** object.

See Also

Reference

CDaoRelationFieldInfo Structure

CDaoRelationFieldInfo Structure

The **CDaoRelationFieldInfo** structure contains information about a field in a relation defined for data access objects (DAO).

```
struct CDaoRelationFieldInfo
{
   CString m_strName;           // Primary
   CString m_strForeignName;    // Primary
};
```

Parameters

m_strName

The name of the field in the primary table of the relation.

m_strForeignName

The name of the field in the foreign table of the relation.

Remarks

A DAO relation object specifies the fields in a primary table and the fields in a foreign table that define the relation. The references to Primary in the structure definition above indicate how the information is returned in the **m_pFieldInfos** member of a CDaoRelationInfo object obtained by calling the GetRelationInfo member function of class **CDaoDatabase**.

Relation objects and relation field objects are not represented by an MFC class. Instead, the DAO objects underlying MFC objects of class CDaoDatabase contain a collection of relation objects, called the Relations collection. Each relation object, in turn, contains a collection of relation field objects. Each relation field object correlates a field in the primary table with a field in the foreign table. Taken together, the relation field objects define a group of fields in each table, which together define the relation. **CDaoDatabase** lets you access relation objects with a **CDaoRelationInfo** object by calling the **GetRelationInfo** member function. The **CDaoRelationInfo** object, then, has a data member, **m_pFieldInfos**, that points to an array of **CDaoRelationFieldInfo** objects.

Call the GetRelationInfo member function of the containing **CDaoDatabase** object in whose Relations collection is stored the relation object you are interested in. Then access the **m_pFieldInfos** member of the CDaoRelationInfo object. **CDaoRelationFieldInfo** also defines a **Dump** member function in debug builds. You can use **Dump** to dump the contents of a **CDaoRelationFieldInfo** object.

See Also

Reference

CDaoRelationInfo Structure

Other Resources

Structures, Styles, Callbacks, and Message Maps

CDaoTableDefInfo Structure

The **CDaoTableDefInfo** structure contains information about a tabledef object defined for data access objects (DAO).

```
struct CDaoTableDefInfo
{
   CString m_strName;               // Primary
   BOOL m_bUpdatable;               // Primary
   long m_lAttributes;              // Primary
   COleDateTime m_dateCreated;      // Secondary
   COleDateTime m_dateLastUpdated;  // Secondary
   CString m_strSrcTableName;       // Secondary
   CString m_strConnect;            // Secondary
   CString m_strValidationRule;     // All
   CString m_strValidationText;     // All
   long m_lRecordCount;             // All
};
```

Parameters

m_strName

Uniquely names the tabledef object. To retrieve the value of this property directly, call the tabledef object's GetName member function. For more information, see the topic "Name Property" in DAO Help.

m_bUpdatable

Indicates whether changes can be made to the table. The quick way to determine whether a table is updatable is to open a **CDaoTableDef** object for the table and call the object's CanUpdate member function. **CanUpdate** always returns nonzero (**TRUE**) for a newly created tabledef object and 0 (**FALSE**) for an attached tabledef object. A new tabledef object can be appended only to a database for which the current user has write permission. If the table contains only nonupdatable fields, **CanUpdate** returns 0. When one or more fields are updatable, **CanUpdate** returns nonzero. You can edit only the updatable fields. For more information, see the topic "Updatable Property" in DAO Help.

m_lAttributes

Specifies characteristics of the table represented by the tabledef object. To retrieve the current attributes of a tabledef, call its GetAttributes member function. The value returned can be a combination of these long constants (using the bitwise-OR (**|**) operator):

- **dbAttachExclusive** For databases that use the Microsoft Jet database engine, indicates the table is an attached table opened for exclusive use.
- **dbAttachSavePWD** For databases that use the Microsoft Jet database engine, indicates that the user ID and password for the attached table are saved with the connection information.
- **dbSystemObject** Indicates the table is a system table provided by the Microsoft Jet database engine. (Read-only.)
- **dbHiddenObject** Indicates the table is a hidden table provided by the Microsoft Jet database engine (for temporary use). (Read-only.)
- **dbAttachedTable** Indicates the table is an attached table from a non-ODBC database, such as a Paradox database.
- **dbAttachedODBC** Indicates the table is an attached table from an ODBC database, such as Microsoft SQL Server.

m_dateCreated

The date and time the table was created. To directly retrieve the date the table was created, call the GetDateCreated member function of the **CDaoTableDef** object associated with the table. See Comments below for more information. For related information, see the topic "DateCreated, LastUpdated Properties" in DAO Help.

m_dateLastUpdated

The date and time of the most recent change made to the design of the table. To directly retrieve the date the table was last updated, call the GetDateLastUpdated member function of the **CDaoTableDef** object associated with the table. See Comments below for more information. For related information, see the topic "DateCreated, LastUpdated Properties" in DAO

Help.

m_strSrcTableName

Specifies the name of an attached table if any. To directly retrieve the source table name, call the GetSourceTableName member function of the **CDaoTableDef** object associated with the table.

m_strConnect

Provides information about the source of an open database. You can check this property by calling the GetConnect member function of your **CDaoTableDef** object. For more information about connect strings, see **GetConnect**.

m_strValidationRule

A value that validates the data in tabledef fields as they are changed or added to a table. Validation is supported only for databases that use the Microsoft Jet database engine. To directly retrieve the validation rule, call the GetValidationRule member function of the **CDaoTableDef** object associated with the table. For related information, see the topic "ValidationRule Property" in DAO Help.

m_strValidationText

A value that specifies the text of the message that your application should display if the validation rule specified by the ValidationRule property is not satisfied. For related information, see the topic "ValidationText Property" in DAO Help.

m_lRecordCount

The number of records accessed in a tabledef object. This property setting is read-only. To directly retrieve the record count, call the GetRecordCount member function of the **CDaoTableDef** object. The documentation for **GetRecordCount** describes the record count further. Note that retrieving this count can be a time-consuming operation if the table contains many records.

Remarks

The tabledef is an object of class CDaoTableDef. The references to Primary, Secondary, and All above indicate how the information is returned by the GetTableDefInfo member function in class **CDaoDatabase**.

Information retrieved by the CDaoDatabase::GetTableDefInfo member function is stored in a **CDaoTableDefInfo** structure. Call the **GetTableDefInfo** member function of the **CDaoDatabase** object in whose TableDefs collection the tabledef object is stored. **CDaoTableDefInfo** also defines a **Dump** member function in debug builds. You can use **Dump** to dump the contents of a **CDaoTableDefInfo** object.

The date and time settings are derived from the computer on which the base table was created or last updated. In a multiuser environment, users should get these settings directly from the file server to avoid discrepancies in the DateCreated and LastUpdated property settings.

See Also

Reference

CDaoTableDef Class
CDaoDatabase Class
CDaoTableDef::CanUpdate
CDaoTableDef::GetAttributes
CDaoTableDef::GetDateCreated
CDaoTableDef::GetDateLastUpdated
CDaoTableDef::GetRecordCount
CDaoTableDef::GetSourceTableName
CDaoTableDef::GetValidationRule
CDaoTableDef::GetValidationText

Other Resources

Structures, Styles, Callbacks, and Message Maps

CDaoWorkspaceInfo Structure

The **CDaoWorkspaceInfo** structure contains information about a workspace defined for data access objects (DAO) database access.

```
struct CDaoWorkspaceInfo
{
   CString m_strName;           // Primary
   CString m_strUserName;       // Secondary
   BOOL m_bIsolateODBCTrans;    // All
};
```

Parameters

m_strName

Uniquely names the workspace object. To retrieve the value of this property directly, call the querydef object's GetName member function. For more information, see the topic "Name Property" in DAO Help.

m_strUserName

A value that represents the owner of a workspace object. For related information, see the topic "UserName Property" in DAO Help.

m_bIsolateODBCTrans

A value that indicates whether multiple transactions that involve the same ODBC database are isolated. For more information, see CDaoWorkspace::SetIsolateODBCTrans. For related information, see the topic "IsolateODBCTrans Property" in DAO Help.

Remarks

The workspace is an object of class CDaoWorkspace. The references to Primary, Secondary, and All above indicate how the information is returned by the GetWorkspaceInfo member function in class **CDaoWorkspace**.

Information retrieved by the CDaoWorkspace::GetWorkspaceInfo member function is stored in a **CDaoWorkspaceInfo** structure. **CDaoWorkspaceInfo** also defines a **Dump** member function in debug builds. You can use **Dump** to dump the contents of a **CDaoWorkspaceInfo** object.

See Also

Reference

CDaoWorkspace Class

Other Resources

Structures, Styles, Callbacks, and Message Maps

CODBCFieldInfo Structure

The **CODBCFieldInfo** structure contains information about the fields in an ODBC data source.

```
struct CODBCFieldInfo
{
   CString m_strName;
   SWORD m_nSQLType;
   UDWORD m_nPrecision;
   SWORD m_nScale;
   SWORD m_nNullability;
};
```

Parameters

m_strName

The name of the field.

m_nSQLType

The SQL data type of the field. This can be an ODBC SQL data type or a driver-specific SQL data type. For a list of valid ODBC SQL data types, see "SQL Data Types" in the *Platform SDK*. For information about driver-specific SQL data types, see the driver's documentation.

m_nPrecision

The maximum precision of the field. For details, see "Precision, Scale, Length, and Display Size" in the *Platform SDK*.

m_nScale

The scale of the field. For details, see "Precision, Scale, Length, and Display Size" in the *Platform SDK*.

m_nNullability

Whether the field accepts a Null value. This can be one of two values: **SQL_NULLABLE** if the field accepts Null values, or **SQL_NO_NULLS** if the field does not accept Null values.

Remarks

To retrieve this information, call CRecordset::GetODBCFieldInfo.

See Also

Reference

CRecordset::GetODBCFieldInfo

CRecordset::GetFieldValue

Other Resources

Structures, Styles, Callbacks, and Message Maps

COLORADJUSTMENT Structure

The **COLORADJUSTMENT** structure defines the color adjustment values used by the Windows **StretchBlt** and **StretchDIBits** functions when the **StretchBlt** mode is **HALFTONE**.

```
typedef struct  tagCOLORADJUSTMENT {    /* ca */
    WORD  caSize;
    WORD  caFlags;
    WORD  caIlluminantIndex;
    WORD  caRedGamma;
    WORD  caGreenGamma;
    WORD  caBlueGamma;
    WORD  caReferenceBlack;
    WORD  caReferenceWhite;
    SHORT caContrast;
    SHORT caBrightness;
    SHORT caColorfulness;
    SHORT caRedGreenTint;
} COLORADJUSTMENT;
```

Parameters

caSize

Specifies the size of the structure in bytes.

caFlags

Specifies how the output image should be prepared. This member can be set to **NULL** or any combination of the following values:

- **CA_NEGATIVE** Specifies that the negative of the original image should be displayed.
- **CA_LOG_FILTER** Specifies that a logarithmic function should be applied to the final density of the output colors. This will increase the color contrast when the luminance is low.

caIlluminantIndex

Specifies the luminance of the light source under which the image object is viewed. This member can be set to one of the following values:

- **ILLUMINANT_EQUAL_ENERGY**
- **ILLUMINANT_A**
- **ILLUMINANT_B**
- **ILLUMINANT_C**
- **ILLUMINANT_D50**
- **ILLUMINANT_D55**
- **ILLUMINANT_D65**
- **ILLUMINANT_D75**
- **ILLUMINANT_F2**
- **ILLUMINANT_TURNGSTEN**
- **ILLUMINANT_DAYLIGHT**
- **ILLUMINANT_FLUORESCENT**
- **ILLUMINANT_NTSC**

caRedGamma

Specifies the nth power gamma-correction value for the red primary of the source colors. The value must be in the range from 2,500 to 65,000. A value of 10,000 means no gamma-correction.

caGreenGamma

Specifies the nth power gamma-correction value for the green primary of the source colors. The value must be in the range from 2,500 to 65,000. A value of 10,000 means no gamma-correction.

caBlueGamma

Specifies the nth power gamma-correction value for the blue primary of the source colors. The value must be in the range from 2,500 to 65,000. A value of 10,000 means no gamma-correction.

caReferenceBlack

Specifies the black reference for the source colors. Any colors that are darker than this are treated as black. The value must be in the range from 0 to 4,000.

caReferenceWhite

Specifies the white reference for the source colors. Any colors that are lighter than this are treated as white. The value must be in the range from 6,000 to 10,000.

caContrast

Specifies the amount of contrast to be applied to the source object. The value must be in the range from -100 to 100. A value of 0 means no contrast adjustment.

caBrightness

Specifies the amount of brightness to be applied to the source object. The value must be in the range from -100 to 100. A value of 0 means no brightness adjustment.

caColorfulness

Specifies the amount of colorfulness to be applied to the source object. The value must be in the range from -100 to 100. A value of 0 means no colorfulness adjustment.

caRedGreenTint

Specifies the amount of red or green tint adjustment to be applied to the source object. The value must be in the range from -100 to 100. Positive numbers would adjust towards red and negative numbers adjust towards green. A 0 means no tint adjustment.

See Also

Reference

CDC::GetColorAdjustment

Other Resources

Structures, Styles, Callbacks, and Message Maps

COMPAREITEMSTRUCT Structure

The **COMPAREITEMSTRUCT** structure supplies the identifiers and application-supplied data for two items in a sorted, owner-drawn list box or combo box.

```
typedef struct tagCOMPAREITEMSTRUCT {
    UINT   CtlType;
    UINT   CtlID;
    HWND   hwndItem;
    UINT   itemID1;
    DWORD  itemData1;
    UINT   itemID2;
    DWORD  itemData2;
} COMPAREITEMSTRUCT;
```

Parameters

CtlType

ODT_LISTBOX (which specifies an owner-draw list box) or **ODT_COMBOBOX** (which specifies an owner-draw combo box).

CtlID

The control ID for the list box or combo box.

hwndItem

The window handle of the control.

itemID1

The index of the first item in the list box or combo box being compared.

itemData1

Application-supplied data for the first item being compared. This value was passed in the call that added the item to the combo or list box.

itemID2

Index of the second item in the list box or combo box being compared.

itemData2

Application-supplied data for the second item being compared. This value was passed in the call that added the item to the combo or list box.

Remarks

Whenever an application adds a new item to an owner-drawn list box or combo box created with the **CBS_SORT** or **LBS_SORT** style, Windows sends the owner a **WM_COMPAREITEM** message. The *lParam* parameter of the message contains a long pointer to a **COMPAREITEMSTRUCT** structure. Upon receiving the message, the owner compares the two items and returns a value indicating which item sorts before the other.

See Also

Reference

CWnd::OnCompareItem

Other Resources

Structures, Styles, Callbacks, and Message Maps

CREATESTRUCT Structure

The **CREATESTRUCT** structure defines the initialization parameters passed to the window procedure of an application.

```
typedef struct tagCREATESTRUCT {
   LPVOID lpCreateParams;
   HANDLE hInstance;
   HMENU hMenu;
   HWND hwndParent;
   int cy;
   int cx;
   int y;
   int x;
   LONG style;
   LPCSTR lpszName;
   LPCSTR lpszClass;
   DWORD dwExStyle;
} CREATESTRUCT;
```

Parameters

lpCreateParams

Points to data to be used to create the window.

hInstance

Identifies the module-instance handle of the module that owns the new window.

hMenu

Identifies the menu to be used by the new window. If a child window, contains the integer ID.

hwndParent

Identifies the window that owns the new window. This member is **NULL** if the new window is a top-level window.

cy

Specifies the height of the new window.

cx

Specifies the width of the new window.

y

Specifies the y-coordinate of the upper left corner of the new window. Coordinates are relative to the parent window if the new window is a child window; otherwise coordinates are relative to the screen origin.

x

Specifies the x-coordinate of the upper left corner of the new window. Coordinates are relative to the parent window if the new window is a child window; otherwise coordinates are relative to the screen origin.

style

Specifies the new window's style.

lpszName

Points to a null-terminated string that specifies the new window's name.

lpszClass

Points to a null-terminated string that specifies the new window's Windows class name (a WNDCLASS structure; for more information, see the *Platform SDK*).

dwExStyle

Specifies the extended style for the new window.

See Also

Reference

CWnd::OnCreate

Other Resources

Structures, Styles, Callbacks, and Message Maps

DELETEITEMSTRUCT Structure

The **DELETEITEMSTRUCT** structure describes a deleted owner-drawn list-box or combo-box item.

```
typedef struct tagDELETEITEMSTRUCT { /* ditms */
    UINT CtlType;
    UINT CtlID;
    UINT itemID;
    HWND hwndItem;
    UINT itemData;
} DELETEITEMSTRUCT;
```

Parameters

CtlType

Specifies **ODT_LISTBOX** (an owner-drawn list box) or **ODT_COMBOBOX** (an owner-drawn combo box).

CtlID

Specifies the identifier of the list box or combo box.

itemID

Specifies index of the item in the list box or combo box being removed.

hwndItem

Identifies the control.

itemData

Specifies application-defined data for the item. This value is passed to the control in the **lParam** parameter of the message that adds the item to the list box or combo box.

Remarks

When an item is removed from the list box or combo box or when the list box or combo box is destroyed, Windows sends the **WM_DELETEITEM** message to the owner for each deleted item. The **lParam** parameter of the message contains a pointer to this structure.

See Also

Reference

CWnd::OnDeleteItem

Other Resources

Structures, Styles, Callbacks, and Message Maps

DEVNAMES Structure

The **DEVNAMES** structure contains strings that identify the driver, device, and output-port names for a printer.

```
typedef struct tagDEVNAMES { /* dvnm */
    WORD wDriverOffset;
    WORD wDeviceOffset;
    WORD wOutputOffset;
    WORD wDefault;
    /* driver, device, and port-name strings follow wDefault */
} DEVNAMES;
```

Parameters

wDriverOffset

(Input/Output) Specifies the offset in characters to a null-terminated string that contains the filename (without the extension) of the device driver. On input, this string is used to determine the printer to display initially in the dialog box.

wDeviceOffset

(Input/Output) Specifies the offset in characters to the null-terminated string (maximum of 32 bytes including the null) that contains the name of the device. This string must be identical to the **dmDeviceName** member of the DEVMODE structure.

wOutputOffset

(Input/Output) Specifies the offset in characters to the null-terminated string that contains the DOS device name for the physical output medium (output port).

wDefault

Specifies whether the strings contained in the **DEVNAMES** structure identify the default printer. This string is used to verify that the default printer has not changed since the last print operation. On input, if the **DN_DEFAULTPRN** flag is set, the other values in the **DEVNAMES** structure are checked against the current default printer. If any of the strings do not match, a warning message is displayed informing the user that the document may need to be reformatted. On output, the **wDefault** member is changed only if the Print Setup dialog box was displayed and the user chose the OK button. The **DN_DEFAULTPRN** flag is set if the default printer was selected. If a specific printer is selected, the flag is not set. All other bits in this member are reserved for internal use by the Print Dialog box procedure.

Remarks

The **PrintDlg** function uses these strings to initialize members in the system-defined Print dialog box. When the user closes the dialog box, information about the selected printer is returned in this structure.

See Also

Reference

CPrintDialog::CreatePrinterDC

Other Resources

Structures, Styles, Callbacks, and Message Maps

DHtmlUrlEventMapEntry Structure

The **DHtmlUrlEventMapEntry** structure provides multi-URL event map support.

```
struct DHtmlUrlEventMapEntry
{
   LPCTSTR szUrl;
   const DHtmlEventMapEntry *pEventMap;
};
```

Parameters

szUrl

The URL.

pEventMap

The event map associated with the URL.

See Also

Other Resources

Structures, Styles, Callbacks, and Message Maps

DRAWITEMSTRUCT Structure

The **DRAWITEMSTRUCT** structure provides information the owner window must have to determine how to paint an owner-drawn control or menu item.

```
typedef struct tagDRAWITEMSTRUCT {
   UINT CtlType;
   UINT CtlID;
   UINT itemID;
   UINT itemAction;
   UINT itemState;
   HWND hwndItem;
   HDC hDC;
   RECT rcItem;
   DWORD itemData;
} DRAWITEMSTRUCT;
```

Parameters

CtlType

The control type. The values for control types are as follows:

- **ODT_BUTTON** Owner-drawn button
- **ODT_COMBOBOX** Owner-drawn combo box
- **ODT_LISTBOX** Owner-drawn list box
- **ODT_MENU** Owner-drawn menu
- **ODT_LISTVIEW** List view control
- **ODT_STATIC** Owner-drawn static control
- **ODT_TAB** Tab control

CtlID

The control ID for a combo box, list box, or button. This member is not used for a menu.

itemID

The menu-item ID for a menu or the index of the item in a list box or combo box. For an empty list box or combo box, this member is a negative value, which allows the application to draw only the focus rectangle at the coordinates specified by the **rcItem** member even though there are no items in the control. The user can thus be shown whether the list box or combo box has the input focus. The setting of the bits in the **itemAction** member determines whether the rectangle is to be drawn as though the list box or combo box has input focus.

itemAction

Defines the drawing action required. This will be one or more of the following bits:

- **ODA_DRAWENTIRE** This bit is set when the entire control needs to be drawn.
- **ODA_FOCUS** This bit is set when the control gains or loses input focus. The **itemState** member should be checked to determine whether the control has focus.
- **ODA_SELECT** This bit is set when only the selection status has changed. The **itemState** member should be checked to determine the new selection state.

itemState

Specifies the visual state of the item after the current drawing action takes place. That is, if a menu item is to be dimmed, the state flag **ODS_GRAYED** will be set. The state flags are as follows:

- **ODS_CHECKED** This bit is set if the menu item is to be checked. This bit is used only in a menu.

- **ODS_DISABLED** This bit is set if the item is to be drawn as disabled.
- **ODS_FOCUS** This bit is set if the item has input focus.
- **ODS_GRAYED** This bit is set if the item is to be dimmed. This bit is used only in a menu.
- **ODS_SELECTED** This bit is set if the item's status is selected.
- **ODS_COMBOBOXEDIT** The drawing takes place in the selection field (edit control) of an ownerdrawn combo box.
- **ODS_DEFAULT** The item is the default item.

hwndItem

Specifies the window handle of the control for combo boxes, list boxes, and buttons. Specifies the handle of the menu (**HMENU**) that contains the item for menus.

hDC

Identifies a device context. This device context must be used when performing drawing operations on the control.

rcItem

A rectangle in the device context specified by the **hDC** member that defines the boundaries of the control to be drawn. Windows automatically clips anything the owner draws in the device context for combo boxes, list boxes, and buttons, but it does not clip menu items. When drawing menu items, the owner must not draw outside the boundaries of the rectangle defined by the **rcItem** member.

itemData

For a combo box or list box, this member contains the value that was passed to the list box by one of the following:

- CComboBox::AddString
- CComboBox::InsertString
- CListBox::AddString
- CListBox::InsertString

For a menu, this member contains the value that was passed to the menu by one of the following:

- CMenu::AppendMenu
- CMenu::InsertMenu
- CMenu::ModifyMenu

Remarks

The owner window of the owner-drawn control or menu item receives a pointer to this structure as the *lParam* parameter of the **WM_DRAWITEM** message.

See Also

Reference

CWnd::OnDrawItem

Other Resources

Structures, Styles, Callbacks, and Message Maps

FILETIME Structure

The **FILETIME** structure is a 64-bit value representing the number of 100-nanosecond intervals since January 1, 1601.

```
typedef struct _FILETIME {
   DWORD dwLowDateTime;   /* low 32 bits  */
   DWORD dwHighDateTime;  /* high 32 bits */
} FILETIME, *PFILETIME, *LPFILETIME;
```

Parameters

dwLowDateTime

Specifies the low 32 bits of the file time.

dwHighDateTime

Specifies the high 32 bits of the file time.

See Also

Reference

CTime::CTime

Other Resources

Structures, Styles, Callbacks, and Message Maps

HSE_VERSION_INFO Structure

This structure is pointed to by the *pVer* parameter in the CHttpServer::GetExtensionVersion member function. It provides the ISA version number and a text description of the ISA.

```
typedef struct _HSE_VERSION_INFO {
   DWORD dwExtensionVersion;
   CHAR lpszExtensionDesc[HSE_MAX_EXT_DLL_NAME_LEN];
} HSE_VERSION_INFO, *LPHSE_VERSION_INFO;
```

Parameters

dwExtensionVersion

The version number of the ISA.

lpszExtensionDesc

The text description of the ISA. The default implementation provides placeholder text; override **CHttpServer::GetExtensionVersion** to provide your own description.

See Also

Reference

CHttpServer::GetExtensionVersion

Other Resources

Structures, Styles, Callbacks, and Message Maps

HTTP_FILTER_AUTHENT Structure

This structure is pointed to by the *pvNotification* in the CHttpFilter::HttpFilterProc when *NotificationType* is **SF_NOTIFY_AUTHENTICATION**, which indicates when the server is about to authenticate the client.

```
typedef struct _HTTP_FILTER_AUTHENT{
   CHAR* pszUser;                          //IN/OUT
   DWORD cbUserBuff;                       //IN
   CHAR* pszPassword;                      //IN/OUT
   DWORD cbPasswordBuff;                   //IN
} HTTP_FILTER_AUTHENT, *PHTTP_FILTER_AUTHENT;
```

Parameters

pszUser

Pointer to a string containing the username for this request. An empty string indicates an anonymous user.

cbUserBuff

Size of the buffer pointed to by **pszUser**. This is guaranteed to be at least **SF_MAX_USERNAME**.

pszPassword

Pointer to a string containing the password for this request.

cbPasswordBuff

Size of the buffer pointed to by **pszPassword**. This is guaranteed to be at least **SF_MAX_PASSWORD**.

Remarks

This structure can be used to implement a different authentication scheme by overriding CHttpFilter::OnAuthentication.

The references to IN or IN/OUT above indicate whether the member applies to messages to the filter (IN) or both to and from the filter (IN/OUT).

See Also

Reference

CHttpFilter::HttpFilterProc
CHttpFilter::OnAuthentication

Other Resources

Structures, Styles, Callbacks, and Message Maps

HTTP_FILTER_AUTH_COMPLETE_INFO Structure

This structure is pointed to by the *pvNotification* in the CHttpFilter::HttpFilterProc when *NotificationType* is **SF_NOTIFY_AUTH_COMPLETE**, which indicates when authorization is complete.

```
typedef struct _HTTP_FILTER_AUTH_COMPLETE_INFO
{
   BOOL (WINAPI * GetHeader) (
      struct _HTTP_FILTER_CONTEXT * pfc,
      LPSTR lpszName,
      LPVOID lpvBuffer,
      LPDWORD lpdwSize
      );
   BOOL (WINAPI * SetHeader) (
      struct _HTTP_FILTER_CONTEXT * pfc,
      LPSTR lpszName,
      LPSTR lpszValue
      );
   BOOL (WINAPI * AddHeader) (
      struct _HTTP_FILTER_CONTEXT * pfc,
      LPSTR lpszName,
      LPSTR lpszValue
      );
   BOOL (WINAPI * GetUserToken) (
      struct _HTTP_FILTER_CONTEXT * pfc,
      HANDLE * phToken
      );
   DWORD HttpStatus;
   BOOL  fResetAuth;
   DWORD dwReserved;
} HTTP_FILTER_AUTH_COMPLETE_INFO, *PHTTP_FILTER_AUTH_COMPLETE_INFO;
```

Parameters

GetHeader

Pointer to a function that retrieves the specified header information. **GetHeader** takes the following parameters:

- *pfc* Pointer to a filter context passed to CHttpFilter::HttpFilterProc.
- *lpszName* The name of the header to retrieve. Header names should include the trailing ':'. The special values "method", "url", and "version" can be used to retrieve the individual portions of the request line.
- *lpvBuffer* Buffer to store header information.
- *lpdwSize* Size of buffer *lpvBuffer*.

SetHeader

Pointer to a function that sets a specified header to a specified value. **SetHeader** takes the following parameters:

- *pfc* Pointer to a filter context passed to **CHttpFilter::HttpFilterProc**.
- *lpszName* The name of the header to set.
- *lpszValue* The value to set.

AddHeader

Pointer to a function that adds a specified header and value. **AddHeader** takes the following parameters:

- *pfc* Pointer to a filter context passed to **CHttpFilter::HttpFilterProc**.
- *lpszName* The name of the header to add.
- *lpszValue* The value of the header.

GetUserToken

Pointer to a function that gets the authenticated user impersonation token. **GetUserToken** takes the following parameters:

- *pfc* Pointer to a filter context passed to **CHttpFilter::HttpFilterProc**.
- *phToken* Handle to the token.

HttpStatus

Status code to user when sending a response.

fResetAuth

Determines whether to reset authorization if URL changes.

dwReserved

Reserved.

See Also

Reference

CHttpFilter::HttpFilterProc

CHttpFilter::OnAuthComplete

Other Resources

Structures, Styles, Callbacks, and Message Maps

HTTP_FILTER_CONTEXT Structure

The **HTTP_FILTER_CONTEXT** structure has the following form:

```
typedef struct _HTTP_FILTER_CONTEXT
{
   DWORD cbSize;                                //IN
   DWORD Revision;                              //IN
   PVOID ServerContext;                         //IN
   DWORD ulReserved;                            //IN
   BOOL fIsSecurePort;                          //IN
   PVOID pFilterContext;                        //IN/OUT
BOOL (WINAPI * GetServerVariable) (
   struct _HTTP_FILTER_CONTEXT * pfc,
   LPSTR lpszVariableName,
   LPVOID lpvBuffer,
   LPDWORD lpdwSize
   );
BOOL (WINAPI * AddResponseHeaders) (
   struct _HTTP_FILTER_CONTEXT * pfc,
   LPSTR lpszHeaders,
   DWORD dwReserved
   );
BOOL (WINAPI * WriteClient) (
   struct _HTTP_FILTER_CONTEXT * pfc,
   LPVOID Buffer,
   LPDWORD lpdwBytes,
   DWORD dwReserved
   );
VOID * (WINAPI * AllocMem) (
   struct _HTTP_FILTER_CONTEXT * pfc,
   DWORD cbSize,
   DWORD dwReserved
   );
BOOL (WINAPI * ServerSupportFunction) (
   struct _HTTP_FILTER_CONTEXT * pfc,
   enum SF_REQ_TYPE sfReq,
   PVOID pData,
   DWORD ul1,
   DWORD ul2
   );
} HTTP_FILTER_CONTEXT, *PHTTP_FILTER_CONTEXT;
```

Parameters

cbSize

Size of this structure, in bytes.

Revision

Revision level of this structure. Less than or equal to the version of the **HTTP_FILTER_REVISION**.

ServerContext

Reserved for server use.

ulReserved

Reserved for server use.

fIsSecurePort

TRUE indicates that this event is occurring over a secure port.

pFilterContext

A pointer to be used by the filter for any context information that the filter wants to associate with this request. Any memory

associated with this request can be safely freed during the **SF_NOTIFY_END_OF_NET_SESSION** notification.

GetServerVariable

Pointer to a function to retrieve information about the server and this connection. See CHttpServerContext::GetServerVariable for details. **GetServerVariable** takes the following parameters:

- *pfc* Pointer to a filter context passed to CHttpFilter::HttpFilterProc.
- *lpszVariableName* Server variable to retrieve.
- *lpvBuffer* Buffer to store value of variable.
- *lpdwSize* Size of buffer *lpvBuffer*.

AddResponseHeaders

Pointer to a function that adds a header to the HTTP response. See the description of **HSE_REQ_SEND_RESPONSE_HEADER** at CHttpServerContext::ServerSupportFunction for details. **AddResponseHeaders** takes the following parameters:

- *pfc* Pointer to a filter context passed to **CHttpFilter::HttpFilterProc**.
- *lpszHeaders* Pointer string containing headers to add.
- *dwReserved* Reserved for future use. Must be 0.

WriteClient

Pointer to a function that sends raw data back to the client. See CHttpFilterContext::WriteClient for details. **WriteClient** takes the following parameters:

- *pfc* Pointer passed to **CHttpFilter::HttpFilterProc**.
- *Buffer* Buffer containing data to send to the client.
- *lpdwBytes* Size of the buffer pointed to by *Buffer*.
- *dwReserved* Reserved for future use.

AllocMem

Pointer to a function used to allocate memory. Any memory allocated with this function will automatically be freed when the request is completed. **AllocMem** takes the following parameters:

- *pfc* Pointer passed to **CHttpFilter::HttpFilterProc**.
- *cbSize* Size of the buffer to allocate.
- *dwReserved* Reserved for future use.

ServerSupportFunction

Pointer to a function used to extend the ISAPI filter APIs. Parameters, listed below, are specific to the ISA used.

- *pfc* Pointer to a function used to extend the ISAPI filter APIs.
- *sfReq* Server function notification. Possible values:

  SF_REQ_SEND_RESPONSE_HEADER Sends a complete HTTP server response header including the status, server version, message time and MIME version. Server extensions should append other information at the end, such as Content-type, Content-length, and so forth, followed by an extra '\r\n'.

  SF_REQ_ADD_HEADERS_ON_DENIAL If the server denies the HTTP request, add the specified headers to the server error response. This allows an authentication filter to advertise its services without filtering every request. Generally the headers will be WWW-Authenticate headers with custom authentication schemes, but no restriction is placed on what headers may be specified.

  SF_REQ_SET_NEXT_READ_SIZE Only used by raw data filters that return **SF_STATUS_READ_NEXT**.

- *pData* Pointer to a string. Specific to the ISA. See the table under the **Comments** section for the appropriate values for each **sfReq** value.

- *ul1*, *ul2* Specific to the ISA. See the table under the **Comments** section for the appropriate values for each **sfReq** value.

Remarks

The references to IN or IN/OUT above indicate whether the member applies to messages to the filter (IN) or both to and from the filter (IN/OUT).

Below are the corresponding possible values for the **ServerSupportFunction** parameters:

sfReq	***pData***	***ul1, ul2***
SF_REQ_SEND_RESPONSE_HEADER	Zero-terminated string pointing to optional status string (i.e., "401 Access Denied") or NULL for the default response of "200 OK".	Zero-terminated string pointing to optional data to be appended and set with the header. If NULL, the header will be terminated with an empty line.
SF_REQ_ADD_HEADERS_ON_DENIAL	Zero-terminated string pointing to one or more header lines with terminating '\r\n'.	
SF_REQ_SET_NEXT_READ_SIZE		Size in bytes for the next read.

See Also

Reference

CHttpFilter::HttpFilterProc
CHttpFilter::OnLog
CHttpServerContext Class
CHttpServerContext::GetServerVariable
CHttpServerContext::ServerSupportFunction
CHttpServerContext::WriteClient

Other Resources

Structures, Styles, Callbacks, and Message Maps

HTTP_FILTER_LOG Structure

This structure is pointed to by the *pvNotification* in the CHttpFilter::HttpFilterProc when *NotificationType* is **SF_NOTIFY_LOG**, which indicates that the server is about to log information to the server log file.

```
typedef struct _HTTP_FILTER_LOG
{
   const CHAR * pszClientHostName;            //IN/OUT
   const CHAR * pszClientUserName;            //IN/OUT
   const CHAR * pszServerName;                //IN/OUT
   const CHAR * pszOperation;                 //IN/OUT
   const CHAR * pszTarget;                    //IN/OUT
   const CHAR * pszParameters;                //IN/OUT
   DWORD dwHttpStatus;                        //IN/OUT
   DWORD dwWin32Status;                       //IN/OUT
} HTTP_FILTER_LOG, *PHTTP_FILTER_LOG;
```

Parameters

pszClientHostName

Client's host name.

pszClientUserName

Client's user name.

pszServerName

Name of the server the client connected to.

pszOperation

HTTP command.

pszTarget

Target of the HTTP command.

pszParameters

Parameters passed to the HTTP command.

dwHttpStatus

HTTP return status.

dwWin32Status

Win32 error code.

Remarks

The strings cannot be changed but pointers can be replaced. If the string pointers are changed, the memory they point to must remain valid until the next notification.

The references to IN/OUT above indicate that the member applies to messages to (IN) and from (OUT) the filter.

See Also

Reference

CHttpFilter::HttpFilterProc
CHttpFilter::OnLog

Other Resources

Structures, Styles, Callbacks, and Message Maps

HTTP_FILTER_PREPROC_HEADERS Structure

This structure is pointed to by the *pvNotification* in CHttpFilter::HttpFilterProc when *NotificationType* is **SF_NOTIFY_PREPROC_HEADERS**, which indicates when the server is about to process the client headers.

```
typedef struct _HTTP_FILTER_PREPROC_HEADERS
{
BOOL (WINAPI * GetHeader) (
   struct _HTTP_FILTER_CONTEXT * pfc,
   LPSTR lpszName,
   LPVOID lpvBuffer,
   LPDWORD lpdwSize
   );
BOOL (WINAPI * SetHeader) (
   struct _HTTP_FILTER_CONTEXT * pfc,
   LPSTR lpszName,
   LPSTR lpszValue
   );
BOOL (WINAPI * AddHeader) (
   struct _HTTP_FILTER_CONTEXT * pfc,
   LPSTR lpszName,
   LPSTR lpszValue
   );
DWORD    dwReserved;
} HTTP_FILTER_PREPROC_HEADERS, *PHTTP_FILTER_PREPROC_HEADERS;
```

Parameters

GetHeader

Pointer to a function that retrieves the specified header value. Header names should include the trailing colon (":"). The special values "method", "url" and "version" can be used to retrieve the individual portions of the request line. **GetHeader** takes the following parameters:

- *pfc* Filter context for this request from the pointer to the filter context passed to the CHttpFilter::HttpFilterProc.
- *lpszName* The name of the header to retrieve.
- *lpvBuffer* Pointer to a buffer of size *lpdwSize* where the value of the header will be stored.
- *lpdwSize* Size of the buffer pointed to by *lpvBuffer*.

SetHeader

Pointer to a function used to change or delete the value of a header. **SetHeader** takes the following parameters:

- *pfc* Filter context for this request from the pointer to the filter context passed to the CHttpFilter::HttpFilterProc.
- *lpszName* Pointer to the name of the header to change or delete.
- *lpszValue* Pointer to the string to change the header to, or a pointer to "\0" to delete the header.

AddHeader

Pointer to a function to add a header. **AddHeader** takes the following parameters:

- *pfc* Filter context for this request from the pointer to the filter context passed to the CHttpFilter::HttpFilterProc.
- *lpszName* Pointer to the name of the header to change or delete.
- *lpszValue* Pointer to the string to change the header to, or a pointer to "\0" to delete the header.

See Also

Reference

CHttpFilter::HttpFilterProc

CHttpFilter::OnPreprocHeaders

Other Resources

HTTP_FILTER_RAW_DATA Structure

This structure is passed to the **SF_NOTIFY_READ_RAW_DATA** and **SF_NOTIFY_SEND_RAW_DATA** notification types for CHttpFilter::HttpFilterProc.

```
typedef struct _HTTP_FILTER_RAW_DATA
{
   PVOID pvInData;                              //IN
   DWORD cbInData;                              //IN
   DWORD cbInBuffer;                            //IN
   DWORD dwReserved;                            //IN
} HTTP_FILTER_RAW_DATA, *PHTTP_FILTER_RAW_DATA;
```

Parameters

pvInData

Pointer to the data buffer (input or output).

cbInData

Amount of data in the buffer pointed to by **pvInData**.

cbInBuffer

Size of the buffer pointed to by **pvInData**.

dwReserved

Reserved for future use.

Remarks

The references to IN above indicate that the message being processed is going to the filter.

See Also

Reference

CHttpFilter::HttpFilterProc
CHttpFilter::OnReadRawData
CHttpFilter::OnSendRawData

Other Resources

Structures, Styles, Callbacks, and Message Maps

HTTP_FILTER_URL_MAP Structure

This structure is pointed to by the *pvNotification* in the CHttpFilter::HttpFilterProc when the *NotificationType* is **SF_NOTIFY_URL_MAP**, which indicates when the server is about to map the specified URL to a physical path. Filters can modify the physical path in place.

```
typedef struct _HTTP_FILTER_URL_MAP
{
   const CHAR * pszURL;                //IN
   CHAR * pszPhysicalPath;             //IN/OUT
   DWORD cbPathBuff;                   //IN
} HTTP_FILTER_URL_MAP, *PHTTP_FILTER_URL_MAP;
```

Parameters

pszURL

Pointer to the URL that is being mapped to a physical path.

pszPhysicalPath

Pointer to the buffer where the physical path is stored.

cbPathBuff

Size of the buffer pointed to by **pszPhysicalPath**.

Remarks

The references to IN or IN/OUT above indicate whether the member applies to messages to the filter (IN) or to and from the filter (IN/OUT).

See Also

Reference

CHttpFilter::HttpFilterProc
CHttpFilter::OnUrlMap

Other Resources

Structures, Styles, Callbacks, and Message Maps

HTTP_FILTER_VERSION Structure

This structure is passed to the application's CHttpFilter::HttpFilterProc entrypoint by the server to associate any context information with the HTTP request.

```
typedef struct _HTTP_FILTER_VERSION
{
   DWORD dwServerFilterVersion;                      //IN
   DWORD dwFilterVersion;                           //OUT
   CHAR lpszFilterDesc[SF_MAX_FILTER_DESC_LEN+1];   //OUT
   DWORD dwFlags;                                   //OUT
} HTTP_FILTER_VERSION, *PHTTP_FILTER_VERSION;
```

Parameters

dwServerFilterVersion

Version of the header used by the filter. The version of the current header file is **HTTP_FILTER_REVISION**.

dwFilterVersion

Version of **HTTP_FILTER_REVISION**.

lpszFilterDesc

Location to store a short string description of the ISAPI filter application.

dwFlags

Combination of **SF_NOTIFY**_* flags to specify what events this application needs, and at what priority the filter is loaded. See CHttpFilter::GetFilterVersion and CHttpFilter::HttpFilterProc for lists of valid flags.

Remarks

The references to IN or OUT above indicate whether the member applies to messages to the filter (IN) or from the filter (OUT).

See Also

Reference

CHttpFilter::HttpFilterProc
CHttpFilter::GetFilterVersion

Other Resources

Structures, Styles, Callbacks, and Message Maps

LINGER Structure

The **LINGER** structure is used for manipulating the **SO_LINGER** and **SO_DONTLINGER** options of **CAsyncSocket::GetSockOpt**.

```
struct linger {
   u_short l_onoff;            // option on/off
   u_short l_linger;           // linger time
};
```

Remarks

Setting the **SO_DONTLINGER** option prevents blocking on member function **Close** while waiting for unsent data to be sent. Setting this option is equivalent to setting **SO_LINGER** with **l_onoff** set to 0.

See Also

Reference

CAsyncSocket::GetSockOpt

CAsyncSocket::SetSockOpt

Other Resources

Structures, Styles, Callbacks, and Message Maps

LOGBRUSH Structure

The **LOGBRUSH** structure defines the style, color, and pattern of a physical brush. It is used by the Windows CreateBrushIndirect and ExtCreatePen functions.

```
typedef struct tag LOGBRUSH { /* lb */
   UINT lbStyle;
   COLORREF lbColor;
   LONG lbHatch;
} LOGBRUSH;
```

Parameters

lbStyle

Specifies the brush style. The **lbStyle** member must be one of the following styles:

- **BS_DIBPATTERN** A pattern brush defined by a device-independent bitmap (DIB) specification. If **lbStyle** is **BS_DIBPATTERN**, the **lbHatch** member contains a handle to a packed DIB.
- **BS_DIBPATTERNPT** A pattern brush defined by a device-independent bitmap (DIB) specification. If **lbStyle** is **BS_DIBPATTERNPT**, the **lbHatch** member contains a pointer to a packed DIB.
- **BS_HATCHED** Hatched brush.
- **BS_HOLLOW** Hollow brush.
- **BS_NULL** Same as **BS_HOLLOW**.
- **BS_PATTERN** Pattern brush defined by a memory bitmap.
- **BS_SOLID** Solid brush.

lbColor

Specifies the color in which the brush is to be drawn. If **lbStyle** is the **BS_HOLLOW** or **BS_PATTERN** style, **lbColor** is ignored. If **lbStyle** is **BS_DIBPATTERN** or **BS_DIBPATTERNBT**, the low-order word of **lbColor** specifies whether the **bmiColors** members of the BITMAPINFO structure contain explicit red, green, blue (RGB) values or indices into the currently realized logical palette. The **lbColor** member must be one of the following values:

- **DIB_PAL_COLORS** The color table consists of an array of 16-bit indices into the currently realized logical palette.
- **DIB_RGB_COLORS** The color table contains literal RGB values.

lbHatch

Specifies a hatch style. The meaning depends on the brush style defined by **lbStyle**. If **lbStyle** is **BS_DIBPATTERN**, the **lbHatch** member contains a handle to a packed DIB. If **lbStyle** is **BS_DIBPATTERNPT**, the **lbHatch** member contains a pointer to a packed DIB. If **lbStyle** is **BS_HATCHED**, the **lbHatch** member specifies the orientation of the lines used to create the hatch. It can be one of the following values:

- **HS_BDIAGONAL** A 45-degree upward, left-to-right hatch
- **HS_CROSS** Horizontal and vertical crosshatch
- **HS_DIAGCROSS** 45-degree crosshatch
- **HS_FDIAGONAL** A 45-degree downward, left-to-right hatch
- **HS_HORIZONTAL** Horizontal hatch
- **HS_VERTICAL** Vertical hatch

If **lbStyle** is **BS_PATTERN**, **lbHatch** is a handle to the bitmap that defines the pattern. If **lbStyle** is **BS_SOLID** or **BS_HOLLOW**, **lbHatch** is ignored.

Remarks

Although **lbColor** controls the foreground color of a hatch brush, the CDC::SetBkMode and CDC::SetBkColor functions control the background color.

See Also

Reference

CDC::GetCharABCWidths

Other Resources

Structures, Styles, Callbacks, and Message Maps

LOGPEN Structure

The **LOGPEN** structure defines the style, width, and color of a pen, a drawing object used to draw lines and borders. The CPen::CreatePenIndirect function uses the **LOGPEN** structure.

```
typedef struct tagLOGPEN {  /* lgpn */
   UINT lopnStyle;
   POINT lopnWidth;
   COLORREF lopnColor;
} LOGPEN;
```

Parameters

lopnStyle

Specifies the pen type. This member can be one of the following values:

- **PS_SOLID** Creates a solid pen.
- **PS_DASH** Creates a dashed pen. (Valid only when the pen width is 1.)
- **PS_DOT** Creates a dotted pen. (Valid only when the pen width is 1.)
- **PS_DASHDOT** Creates a pen with alternating dashes and dots. (Valid only when the pen width is 1.)
- **PS_DASHDOTDOT** Creates a pen with alternating dashes and double dots. (Valid only when the pen width is 1.)
- **PS_NULL** Creates a null pen.
- **PS_INSIDEFRAME** Creates a pen that draws a line inside the frame of closed shapes produced by GDI output functions that specify a bounding rectangle (for example, the **Ellipse**, **Rectangle**, **RoundRect**, **Pie**, and **Chord** member functions). When this style is used with GDI output functions that do not specify a bounding rectangle (for example, the **LineTo** member function), the drawing area of the pen is not limited by a frame.

  If a pen has the **PS_INSIDEFRAME** style and a color that does not match a color in the logical color table, the pen is drawn with a dithered color. The **PS_SOLID** pen style cannot be used to create a pen with a dithered color. The **PS_INSIDEFRAME** style is identical to **PS_SOLID** if the pen width is less than or equal to 1.

  When the **PS_INSIDEFRAME** style is used with GDI objects produced by functions other than **Ellipse**, **Rectangle**, and **RoundRect**, the line may not be completely inside the specified frame.

lopnWidth

Specifies the pen width, in logical units. If the **lopnWidth** member is 0, the pen is 1 pixel wide on raster devices regardless of the current mapping mode.

lopnColor

Specifies the pen color.

Remarks

The **y** value in the POINT structure for the **lopnWidth** member is not used.

See Also

Reference

CPen::CreatePenIndirect

Other Resources

Structures, Styles, Callbacks, and Message Maps

MEASUREITEMSTRUCT Structure

The **MEASUREITEMSTRUCT** structure informs Windows of the dimensions of an owner-drawn control or menu item.

```
typedef struct tagMEASUREITEMSTRUCT {
   UINT CtlType;
   UINT CtlID;
   UINT itemID;
   UINT itemWidth;
   UINT itemHeight;
   DWORD itemData
} MEASUREITEMSTRUCT;
```

Parameters

CtlType

Contains the control type. The values for control types are as follows:

- **ODT_COMBOBOX** Owner-draw combo box
- **ODT_LISTBOX** Owner-draw list box
- **ODT_MENU** Owner-draw menu

CtlID

Contains the control ID for a combo box, list box, or button. This member is not used for a menu.

itemID

Contains the menu-item ID for a menu or the list-box-item ID for a variable-height combo box or list box. This member is not used for a fixed-height combo box or list box, or for a button.

itemWidth

Specifies the width of a menu item. The owner of the owner-draw menu item must fill this member before it returns from the message.

itemHeight

Specifies the height of an individual item in a list box or a menu. Before it returns from the message, the owner of the owner-draw combo box, list box, or menu item must fill out this member. The maximum height of a list box item is 255.

itemData

For a combo box or list box, this member contains the value that was passed to the list box by one of the following:

- CComboBox::AddString
- CComboBox::InsertString
- CListBox::AddString
- CListBox::InsertString

For a menu, this member contains the value that was passed to the menu by one of the following:

- CMenu::AppendMenu
- CMenu::InsertMenu
- CMenu::ModifyMenu

This allows Windows to process user interaction with the control correctly. Failure to fill out the proper members in the **MEASUREITEMSTRUCT** structure will cause improper operation of the control.

See Also

Reference

CWnd::OnMeasureItem

Other Resources

Structures, Styles, Callbacks, and Message Maps

MINMAXINFO Structure

The **MINMAXINFO** structure contains information about a window's maximized size and position and its minimum and maximum tracking size.

```
typedef struct tagMINMAXINFO {
   POINT ptReserved;
   POINT ptMaxSize;
   POINT ptMaxPosition;
   POINT ptMinTrackSize;
   POINT ptMaxTrackSize;
} MINMAXINFO;
```

Parameters

ptReserved

Reserved for internal use.

ptMaxSize

Specifies the maximized width (point.x) and the maximized height (point.y) of the window.

ptMaxPosition

Specifies the position of the left side of the maximized window (point.x) and the position of the top of the maximized window (point.y).

ptMinTrackSize

Specifies the minimum tracking width (point.x) and the minimum tracking height (point.y) of the window.

ptMaxTrackSize

Specifies the maximum tracking width (point.x) and the maximum tracking height (point.y) of the window.

See Also

Reference

CWnd::OnGetMinMaxInfo

Other Resources

Structures, Styles, Callbacks, and Message Maps

MSG Structure

The **MSG** structure contains message information from a thread's message queue.

```
typedef struct tagMSG {     // msg
   HWND hwnd;
   UINT message;
   WPARAM wParam;
   LPARAM lParam;
   DWORD time;
   POINT pt;
} MSG;
```

Parameters

hwnd

Identifies the window whose window procedure receives the message.

message

Specifies the message number.

wParam

Specifies additional information about the message. The exact meaning depends on the value of the **message** member.

lParam

Specifies additional information about the message. The exact meaning depends on the value of the **message** member.

time

Specifies the time at which the message was posted.

pt

Specifies the cursor position, in screen coordinates, when the message was posted.

See Also

Other Resources

Structures, Styles, Callbacks, and Message Maps

NCCALCSIZE_PARAMS Structure

The **NCCALCSIZE_PARAMS** structure contains information that an application can use while processing the **WM_NCCALCSIZE** message to calculate the size, position, and valid contents of the client area of a window.

```
typedef struct tagNCCALCSIZE_PARAMS {
   RECT rgrc[3];
   PWINDOWPOS lppos;
} NCCALCSIZE_PARAMS;
```

Parameters

rgrc

Specifies an array of rectangles. The first contains the new coordinates of a window that has been moved or resized. The second contains the coordinates of the window before it was moved or resized. The third contains the coordinates of the client area of a window before it was moved or resized. If the window is a child window, the coordinates are relative to the client area of the parent window. If the window is a top-level window, the coordinates are relative to the screen.

lppos

Points to a WINDOWPOS structure that contains the size and position values specified in the operation that caused the window to be moved or resized.

See Also

Reference

CWnd::OnNcCalcSize

Other Resources

Structures, Styles, Callbacks, and Message Maps

PAINTSTRUCT Structure

The **PAINTSTRUCT** structure contains information that can be used to paint the client area of a window.

```
typedef struct tagPAINTSTRUCT {
   HDC hdc;
   BOOL fErase;
   RECT rcPaint;
   BOOL fRestore;
   BOOL fIncUpdate;
   BYTE rgbReserved[16];
} PAINTSTRUCT;
```

Parameters

hdc

Identifies the display context to be used for painting.

fErase

Specifies whether the background needs to be redrawn. It is not 0 if the application should redraw the background. The application is responsible for drawing the background if a Windows window-class is created without a background brush (see the description of the **hbrBackground** member of the WNDCLASS structure in the *Platform SDK*).

rcPaint

Specifies the upper left and lower right corners of the rectangle in which the painting is requested.

fRestore

Reserved member. It is used internally by Windows.

fIncUpdate

Reserved member. It is used internally by Windows.

rgbReserved[16]

Reserved member. A reserved block of memory used internally by Windows.

See Also

Reference

CPaintDC::m_ps

Other Resources

Structures, Styles, Callbacks, and Message Maps

POINT Structure

The **POINT** structure defines the x- and y-coordinates of a point.

```
typedef struct tagPOINT {
   LONG x;
   LONG y;
} POINT;
```

Parameters

x

Specifies the x-coordinate of a point.

y

Specifies the y-coordinate of a point.

Example

```
//Alternate ways to initialize a POINT structure:

POINT ptA;
ptA.x = 370;
ptA.y = 550;

POINT ptB = {370,550};
```

See Also

Reference

CPoint Class

Other Resources

Structures, Styles, Callbacks, and Message Maps

RECT Structure

The **RECT** structure defines the coordinates of the upper-left and lower-right corners of a rectangle.

```
typedef struct tagRECT {
   LONG left;
   LONG top;
   LONG right;
   LONG bottom;
} RECT;
```

Members

left

Specifies the x-coordinate of the upper-left corner of a rectangle.

top

Specifies the y-coordinate of the upper-left corner of a rectangle.

right

Specifies the x-coordinate of the lower-right corner of a rectangle.

bottom

Specifies the y-coordinate of the lower-right corner of a rectangle.

Example

```
//Alternate ways to intialize a RECT structure:

RECT rctA;
rctA.left = 20;
rctA.top = 30;
rctA.right  = 180;
rctA.bottom = 230;

RECT rctB = {20,30,180,230};
```

See Also

Reference

CRect Class

Other Resources

Structures, Styles, Callbacks, and Message Maps

RGNDATA Structure

The **RGNDATA** structure contains a header and an array of rectangles that compose a region. These rectangles, sorted top to bottom left to right, do not overlap.

```
typedef struct _RGNDATA { /* rgnd */
   RGNDATAHEADER rdh;
   char Buffer[1];
} RGNDATA;
```

Parameters

rdh

Specifies a RGNDATAHEADER structure. (For more information on this structure, see the *Platform SDK*.) The members of this structure specify the type of region (whether it is rectangular or trapezoidal), the number of rectangles that make up the region, the size of the buffer that contains the rectangle structures, and so on.

Buffer

Specifies an arbitrary-size buffer that contains the RECT structures that make up the region.

See Also

Reference

CRgn::CreateFromData

CRgn::GetRegionData

Other Resources

Structures, Styles, Callbacks, and Message Maps

SOCKADDR Structure

The **SOCKADDR** structure is used to store an Internet Protocol (IP) address for a machine participating in a Windows Sockets communication.

```
struct sockaddr {
   unsigned short sa_family;
   char sa_data[14];
};
```

Parameters

sa_family

Socket address family.

sa_data

Maximum size of all the different socket address structures.

Remarks

The Microsoft TCP/IP Sockets Developer's Kit only supports the Internet address domains. To actually fill in values for each part of an address, you use the **SOCKADDR_IN** data structure, which is specifically for this address format. The **SOCKADDR** and the **SOCKADDR_IN** data structures are the same size. You simply cast to switch between the two structure types.

See Also

Reference

SOCKADDR_IN Structure
CAsyncSocket::Create
CSocket::Create

Other Resources

Structures, Styles, Callbacks, and Message Maps

SOCKADDR_IN Structure

In the Internet address family, the **SOCKADDR_IN** structure is used by Windows Sockets to specify a local or remote endpoint address to which to connect a socket.

```
struct sockaddr_in{
   short sin_family;
   unsigned short sin_port;
   struct in_addr sin_addr;
   char sin_zero[8];
};
```

Parameters

sin_family

Address family (must be **AF_INET**).

sin_port

IP port.

sin_addr

IP address.

sin_zero

Padding to make structure the same size as **SOCKADDR**.

Remarks

This is the form of the **SOCKADDR** structure specific to the Internet address family and can be cast to **SOCKADDR**.

The IP address component of this structure is of type **IN_ADDR**. The **IN_ADDR** structure is defined in Windows Sockets header file WINSOCK.H as follows:

```
struct   in_addr {
    union   {
         struct{
             unsigned  char   s_b1,
                              s_b2,
                              s_b3,
                              s_b4;
        }  S_un_b;
             struct  {
             unsigned  short  s_w1,
                              s_w2;
              }  S_un_w;
               unsigned long  S_addr;
     } S_un;
};
```

See Also

Reference

SOCKADDR Structure

Other Resources

Structures, Styles, Callbacks, and Message Maps

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME {
   WORD wYear;
   WORD wMonth;
   WORD wDayOfWeek;
   WORD wDay;
   WORD wHour;
   WORD wMinute;
   WORD wSecond;
   WORD wMilliseconds;
} SYSTEMTIME;
```

Parameters

wYear

The current year.

wMonth

The current month; January is 1.

wDayOfWeek

The current day of the week; Sunday is 0, Monday is 1, and so on.

wDay

The current day of the month.

wHour

The current hour.

wMinute

The current minute.

wSecond

The current second.

wMilliseconds

The current millisecond.

Example

```
// systemtime_structure.cpp
// Retrieves the current system date and time.  The system
// time is expressed in Coordinated Universal Time (UTC).
// Compile with: /EHsc
#include <afx.h>
#include <afxwin.h>

CWinApp theApp;

int main()
{
    SYSTEMTIME systime;
    GetSystemTime(&systime);

    // Determine day of the week.
    CString day;
    switch (systime.wDayOfWeek)
```

```
    {
        case 0:
            day = "Sunday";
        break;

        case 1:
            day = "Monday";
        break;

        case 2:
            day = "Tuesday";
        break;

        case 3:
            day = "Wednesday";
        break;

        case 4:
            day = "Thursday";
        break;

        case 5:
            day = "Friday";
        break;

        case 6:
            day = "Saturday";
        break;
    }

    // Show the time in a message box.
    enum { sizeOfStr = 50 };
    char str[sizeOfStr];

    sprintf_s(str, sizeOfStr, "%s %u/%u/%u  %u:%u:%u:%u",
              day,
              systime.wYear, systime.wMonth, systime.wDay,
              systime.wHour, systime.wMinute, systime.wSecond,
              systime.wMilliseconds);

    AfxMessageBox(str);

   return 0;
}
```

See Also

Reference

CTime::CTime

Other Resources

Structures, Styles, Callbacks, and Message Maps

WINDOWPLACEMENT Structure

The **WINDOWPLACEMENT** structure contains information about the placement of a window on the screen**.**

```
typedef struct tagWINDOWPLACEMENT {     /* wndpl */
   UINT length;
   UINT flags;
   UINT showCmd;
   POINT ptMinPosition;
   POINT ptMaxPosition;
   RECT rcNormalPosition;
} WINDOWPLACEMENT;
```

Parameters

length

Specifies the length, in bytes, of the structure**.**

flags

Specifies flags that control the position of the minimized window and the method by which the window is restored. This member can be one or both of the following flags:

- **WPF_SETMINPOSITION** Specifies that the x- and y-positions of the minimized window can be specified**.** This flag must be specified if the coordinates are set in the **ptMinPosition** member.
- **WPF_RESTORETOMAXIMIZED** Specifies that the restored window will be maximized, regardless of whether it was maximized before it was minimized. This setting is valid only the next time the window is restored. It does not change the default restoration behavior. This flag is valid only when the **SW_SHOWMINIMIZED** value is specified for the **showCmd** member.

showCmd

Specifies the current show state of the window. This member can be one of the following values:

- **SW_HIDE** Hides the window and passes activation to another window.
- **SW_MINIMIZE** Minimizes the specified window and activates the top-level window in the system's list.
- **SW_RESTORE** Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position (same as **SW_SHOWNORMAL**).
- **SW_SHOW** Activates a window and displays it in its current size and position.
- **SW_SHOWMAXIMIZED** Activates a window and displays it as a maximized window.
- **SW_SHOWMINIMIZED** Activates a window and displays it as an icon.
- **SW_SHOWMINNOACTIVE** Displays a window as an icon. The window that is currently active remains active.
- **SW_SHOWNA** Displays a window in its current state. The window that is currently active remains active.
- **SW_SHOWNOACTIVATE** Displays a window in its most recent size and position. The window that is currently active remains active.
- **SW_SHOWNORMAL** Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position (same as **SW_RESTORE**).

ptMinPosition

Specifies the position of the window's top-left corner when the window is minimized.

ptMaxPosition

Specifies the position of the window's top-left corner when the window is maximized.

rcNormalPosition

Specifies the window's coordinates when the window is in the normal (restored) position.

See Also

Reference

CWnd::SetWindowPlacement

Other Resources

Structures, Styles, Callbacks, and Message Maps

WINDOWPOS Structure

The **WINDOWPOS** structure contains information about the size and position of a window.

```
typedef struct tagWINDOWPOS { /* wp */
   HWND hwnd;
   HWND hwndInsertAfter;
   int x;
   int y;
   int cx;
   int cy;
   UINT flags;
} WINDOWPOS;
```

Parameters

hwnd

Identifies the window.

hwndInsertAfter

Identifies the window behind which this window is placed.

x

Specifies the position of the left edge of the window.

y

Specifies the position of the right edge of the window.

cx

Specifies the window width, in pixels.

cy

Specifies the window height, in pixels.

flags

Specifies window-positioning options. This member can be one of the following values:

- **SWP_DRAWFRAME** Draws a frame (defined in the class description for the window) around the window. The window receives a **WM_NCCALCSIZE** message.
- **SWP_FRAMECHANGED** Sends a **WM_NCCALCSIZE** message to the window, even if the window's size is not being changed. If this flag is not specified, **WM_NCCALCSIZE** is sent only when the window's size is being changed.
- **SWP_HIDEWINDOW** Hides the window.
- **SWP_NOACTIVATE** Does not activate the window.
- **SWP_NOCOPYBITS** Discards the entire contents of the client area. If this flag is not specified, the valid contents of the client area are saved and copied back into the client area after the window is sized or repositioned.
- **SWP_NOMOVE** Retains current position (ignores the **x** and **y** members).
- **SWP_NOOWNERZORDER** Does not change the owner window's position in the Z-order.
- **SWP_NOSIZE** Retains current size (ignores the **cx** and **cy** members).
- **SWP_NOREDRAW** Does not redraw changes.
- **SWP_NOREPOSITION** Same as **SWP_NOOWNERZORDER**.
- **SWP_NOSENDCHANGING** Prevents the window from receiving the **WM_WINDOWPOSCHANGING** message.
- **SWP_NOZORDER** Retains current ordering (ignores the **hwndInsertAfter** member).

- **SWP_SHOWWINDOW** Displays the window.

See Also

Reference

CWnd::OnWindowPosChanging

Other Resources

Structures, Styles, Callbacks, and Message Maps

WSADATA Structure

The **WSADATA** structure is used to store Windows Sockets initialization information returned by a call to the **AfxSocketInit** global function.

```
struct WSAData {
   WORD wVersion;
   WORD wHighVersion;
   char szDescription[WSADESCRIPTION_LEN+1];
   char szSystemStatus[WSASYSSTATUS_LEN+1];
   unsigned short iMaxSockets;
   unsigned short iMaxUdpDg;
   char FAR * lpVendorInfo;
};
```

Parameters

wVersion

The version of the Windows Sockets specification that the Windows Sockets DLL expects the caller to use.

wHighVersion

The highest version of the Windows Sockets specification that this DLL can support (also encoded as above). Normally, this is the same as **wVersion**.

szDescription

A null-terminated ASCII string into which the Windows Sockets DLL copies a description of the Windows Sockets implementation, including vendor identification. The text (up to 256 characters in length) can contain any characters, but vendors are cautioned against including control and formatting characters: the most likely use that an application will put this to is to display it (possibly truncated) in a status message.

szSystemStatus

A null-terminated ASCII string into which the Windows Sockets DLL copies relevant status or configuration information. The Windows Sockets DLL should use this field only if the information might be useful to the user or support staff; it should not be considered as an extension of the **szDescription** field.

iMaxSockets

The maximum number of sockets that a single process can potentially open. A Windows Sockets implementation can provide a global pool of sockets for allocation to any process; alternately, it can allocate per-process resources for sockets. The number can well reflect the way in which the Windows Sockets DLL or the networking software was configured. Application writers can use this number as a crude indication of whether the Windows Sockets implementation is usable by the application. For example, an X Windows server might check **iMaxSockets** when first started: if it is less than 8, the application would display an error message instructing the user to reconfigure the networking software. (This is a situation in which the **szSystemStatus** text might be used.) Obviously there is no guarantee that a particular application can actually allocate **iMaxSockets** sockets, since there can be other Windows Sockets applications in use.

iMaxUdpDg

The size in bytes of the largest User Datagram Protocol (UDP) datagram that can be sent or received by a Windows Sockets application. If the implementation imposes no limit, **iMaxUdpDg** is zero. In many implementations of Berkeley sockets, there is an implicit limit of 8192 bytes on UDP datagrams (which are fragmented if necessary). A Windows Sockets implementation can impose a limit based, for instance, on the allocation of fragment reassembly buffers. The minimum value of **iMaxUdpDg** for a compliant Windows Sockets implementation is 512. Note that regardless of the value of **iMaxUdpDg**, it is inadvisable to attempt to send a broadcast datagram that is larger than the Maximum Transmission Unit (MTU) for the network. (The Windows Sockets API does not provide a mechanism to discover the MTU, but it must be no less than 512 bytes.)

lpVendorInfo

A far pointer to a vendor-specific data structure. The definition of this structure (if supplied) is beyond the scope of the Windows Sockets specification.

Note
In MFC, the **WSADATA** structure is returned by the **AfxSocketInit** function, which you call in your `InitInstance` function. You can retrieve the structure and store it in your program if you need to use information from it later.

See Also

Reference

AfxSocketInit

Other Resources

Structures, Styles, Callbacks, and Message Maps

XFORM Structure

The **XFORM** structure has the following form:

```
typedef struct  tagXFORM {  /* xfrm */
    FLOAT eM11;
    FLOAT eM12;
    FLOAT eM21;
    FLOAT eM22;
    FLOAT eDx;
    FLOAT eDy;
} XFORM;
```

Remarks

The **XFORM** structure specifies a world-space to page-space transformation. The **eDx** and **eDy** members specify the horizontal and vertical translation components, respectively. The following table shows how the other members are used, depending on the operation:

Operation	**eM11**	**eM12**	**eM21**	**eM22**
Rotation	Cosine of rotation angle	Sine of rotation angle	Negative sine of rotation angle	Cosine of rotation angle
Scaling	Horizontal scaling component	Nothing	Nothing	Vertical scaling component
Shear	Nothing	Horizontal proportionality constant	Vertical proportionality constant	Nothing
Reflection	Horizontal reflection component	Nothing	Nothing	Vertical reflection component

See Also

Reference

CRgn::CreateFromData

Other Resources

Structures, Styles, Callbacks, and Message Maps

Styles Used by MFC

Use the following styles when you create the corresponding MFC object. In most cases, these styles are set in the *dwStyle* parameter of the class **Create** function.

MFC Styles

Style	Description
Button styles	Applies to CButton Class objects, such as radio buttons, check boxes and pushbuttons. Specify a combination of styles in the *dwStyle* parameter of CButton::Create.
Combo-box styles	Applies to CComboBox Class objects. Specify a combination of styles in the *dwStyle* parameter of CComboBox::Create.
Edit styles	Applies to CEdit Class objects. Specify a combination of styles in the *dwStyle* parameter of CEdit::Create.
Frame-window styles	Applies to CFrameWnd Class objects. Specify a combination of styles in the *dwStyle* parameter of CFrameWnd::Create.
List-box styles	Applies to CListBox Class objects. Specify a combination of styles in the *dwStyle* parameter of CListBox::Create.
Message-box styles	Applies to AfxMessageBox items. Specify a combination of styles in the *nType* parameter of **AfxMessageBox**.
Scroll-bar styles	Applies to CScrollBar Class objects. Specify a combination of styles in the *dwStyle* parameter of CScrollBar::Create.
Static styles	Applies to CStatic Class objects. Specify a combination of styles in the *dwStyle* parameter of CStatic::Create.
Window styles	Applies to CWnd Class objects. Specify a combination of styles in the *dwStyle* parameter of CWnd::Create or CWnd::CreateEx.
Extended window styles	Applies to CWnd Class objects. Specify a combination of styles in the *dwExStyle* parameter of CWnd::CreateEx.

See Also

Concepts

Class Library Overview

Button Styles

- **BS_3STATE** Same as a check box, except that the box can be dimmed as well as checked. The dimmed state typically is used to show that a check box has been disabled.
- **BS_AUTO3STATE** Same as a three-state check box, except that the box changes its state when the user selects it.
- **BS_AUTOCHECKBOX** Same as a check box, except that a check mark appears in the check box when the user selects the box; the check mark disappears the next time the user selects the box.
- **BS_AUTORADIOBUTTON** Same as a radio button, except that when the user selects it, the button automatically highlights itself and removes the selection from any other radio buttons with the same style in the same group.
- **BS_BITMAP** Specifies that the button displays a bitmap.
- **BS_BOTTOM** Places text at the bottom of the button rectangle.
- **BS_CENTER** Centers text horizontally in the button rectangle.
- **BS_CHECKBOX** Creates a small square that has text displayed to its right (unless this style is combined with the **BS_LEFTTEXT** style).
- **BS_DEFPUSHBUTTON** Creates a button that has a heavy black border. The user can select this button by pressing the ENTER key. This style enables the user to quickly select the most likely option (the default option).
- **BS_FLAT** Specifies that the button is two-dimensional; it does not use the default shading to create a 3-D image.
- **BS_GROUPBOX** Creates a rectangle in which other buttons can be grouped. Any text associated with this style is displayed in the rectangle's upper-left corner.
- **BS_ICON** Specifies that the button displays an icon.
- **BS_LEFT** Left aligns the text in the button rectangle. However, if the button is a check box or radio button that does not have the **BS_RIGHTBUTTON** style, the text is left aligned on the right side of the check box or radio button.
- **BS_LEFTTEXT** When combined with a radio-button or check-box style, the text appears on the left side of the radio button or check box.
- **BS_MULTILINE** Wraps the button text to multiple lines if the text string is too long to fit on a single line in the button rectangle.
- **BS_NOTIFY** Enables a button to send **BN_DBLCLK**, **BN_KILLFOCUS**, and **BN_SETFOCUS** notification messages to its parent window. Note that buttons send the **BN_CLICKED** notification message regardless of whether it has this style.
- **BS_OWNERDRAW** Creates an owner-drawn button. The framework calls the **DrawItem** member function when a visual aspect of the button has changed. This style must be set when using the **CBitmapButton** class.
- **BS_PUSHBUTTON** Creates a pushbutton that posts a **WM_COMMAND** message to the owner window when the user selects the button.
- **BS_PUSHLIKE** Makes a button (such as a check box, three-state check box, or radio button) look and act like a push button. The button looks raised when it isn't pushed or checked, and sunken when it is pushed or checked.
- **BS_RADIOBUTTON** Creates a small circle that has text displayed to its right (unless this style is combined with the **BS_LEFTTEXT** style). Radio buttons are usually used in groups of related, but mutually exclusive, choices.
- **BS_RIGHT** Right aligns the text in the button rectangle. However, if the button is a check box or radio button that does not have the **BS_RIGHTBUTTON** style, the text is right aligned on the right side of the check box or radio button.
- **BS_RIGHTBUTTON** Positions a radio button's circle or a check box's square on the right side of the button rectangle. Same as the **BS_LEFTTEXT** style.
- **BS_TEXT** Specifies that the button displays text.
- **BS_TOP** Places text at the top of the button rectangle.
- **BS_USERBUTTON** Obsolete, but provided for compatibility with 16-bit versions of Windows. Win32-based applications should use **BS_OWNERDRAW** instead.

- **BS_VCENTER** Places text in the middle (vertically) of the button rectangle.

See Also

Reference

CButton::Create

Button Styles

Other Resources

Styles Used by MFC

Combo-Box Styles

The following combo-box styles are available in MFC.

- **CBS_AUTOHSCROLL** Automatically scrolls the text in the edit control to the right when the user types a character at the end of the line. If this style is not set, only text that fits within the rectangular boundary is allowed.
- **CBS_DISABLENOSCROLL** The list box shows a disabled vertical scroll bar when the list box does not contain enough items to scroll. Without this style, the scroll bar is hidden when the list box does not contain enough items.
- **CBS_DROPDOWN** Similar to **CBS_SIMPLE**, except that the list box is not displayed unless the user selects an icon next to the edit control.
- **CBS_DROPDOWNLIST** Similar to **CBS_DROPDOWN**, except that the edit control is replaced by a static-text item that displays the current selection in the list box.
- **CBS_HASSTRINGS** An owner-draw combo box contains items consisting of strings. The combo box maintains the memory and pointers for the strings so the application can use the **GetText** member function to retrieve the text for a particular item.
- **CBS_LOWERCASE** Converts to lowercase all text in both the selection field and the list.
- **CBS_NOINTEGRALHEIGHT** Specifies that the size of the combo box is exactly the size specified by the application when it created the combo box. Normally, Windows sizes a combo box so that the combo box does not display partial items.
- **CBS_OEMCONVERT** Text entered in the combo-box edit control is converted from the ANSI character set to the OEM character set and then back to ANSI. This ensures proper character conversion when the application calls the **AnsiToOem** Windows function to convert an ANSI string in the combo box to OEM characters. This style is most useful for combo boxes that contain filenames and applies only to combo boxes created with the **CBS_SIMPLE** or **CBS_DROPDOWN** styles.
- **CBS_OWNERDRAWFIXED** The owner of the list box is responsible for drawing its contents; the items in the list box are all the same height.
- **CBS_OWNERDRAWVARIABLE** The owner of the list box is responsible for drawing its contents; the items in the list box are variable in height.
- **CBS_SIMPLE** The list box is displayed at all times. The current selection in the list box is displayed in the edit control.
- **CBS_SORT** Automatically sorts strings entered into the list box.
- **CBS_UPPERCASE** Converts to uppercase all text in both the selection field and the list.

See Also

Reference

CComboBox::Create

Combo Box Styles

Other Resources

Styles Used by MFC

Edit Styles

- **ES_AUTOHSCROLL** Automatically scrolls text to the right by 10 characters when the user types a character at the end of the line. When the user presses the ENTER key, the control scrolls all text back to position 0.
- **ES_AUTOVSCROLL** Automatically scrolls text up one page when the user presses ENTER on the last line.
- **ES_CENTER** Centers text in a single-line or multiline edit control.
- **ES_LEFT** Left-aligns text in a single-line or multiline edit control.
- **ES_LOWERCASE** Converts all characters to lowercase as they are typed into the edit control.
- **ES_MULTILINE** Designates a multiple-line edit control. (The default is single line.) If the **ES_AUTOVSCROLL** style is specified, the edit control shows as many lines as possible and scrolls vertically when the user presses the ENTER key. If **ES_AUTOVSCROLL** is not given, the edit control shows as many lines as possible and beeps if ENTER is pressed when no more lines can be displayed. If the **ES_AUTOHSCROLL** style is specified, the multiple-line edit control automatically scrolls horizontally when the caret goes past the right edge of the control. To start a new line, the user must press ENTER. If **ES_AUTOHSCROLL** is not given, the control automatically wraps words to the beginning of the next line when necessary; a new line is also started if ENTER is pressed. The position of the wordwrap is determined by the window size. If the window size changes, the wordwrap position changes and the text is redisplayed. Multiple-line edit controls can have scroll bars. An edit control with scroll bars processes its own scroll-bar messages. Edit controls without scroll bars scroll as described above and process any scroll messages sent by the parent window.
- **ES_NOHIDESEL** Normally, an edit control hides the selection when the control loses the input focus and inverts the selection when the control receives the input focus. Specifying **ES_NOHIDESEL** deletes this default action.
- **ES_NUMBER** Allows only digits to be entered into the edit control.
- **ES_OEMCONVERT** Text entered in the edit control is converted from the ANSI character set to the OEM character set and then back to ANSI. This ensures proper character conversion when the application calls the **AnsiToOem** Windows function to convert an ANSI string in the edit control to OEM characters. This style is most useful for edit controls that contain filenames.
- **ES_PASSWORD** Displays all characters as an asterisk (*****) as they are typed into the edit control. An application can use the **SetPasswordChar** member function to change the character that is displayed.
- **ES_READONLY** Prevents the user from entering or editing text in the edit control.
- **ES_RIGHT** Right-aligns text in a single-line or multiline edit control.
- **ES_UPPERCASE** Converts all characters to uppercase as they are typed into the edit control.
- **ES_WANTRETURN** Specifies that a carriage return be inserted when the user presses the ENTER key while entering text into a multiple-line edit control in a dialog box. Without this style, pressing the ENTER key has the same effect as pressing the dialog boxs default pushbutton. This style has no effect on a single-line edit control.

See Also

Reference

CEdit::Create

Edit Control Styles

Other Resources

Styles Used by MFC

Frame-Window Styles (MFC)

- **FWS_ADDTOTITLE** Specifies information to append to the end of a frame window title. For example, "Microsoft Draw - Drawing in Document1". You can specify the strings displayed in the Document Template Strings tab in the Application Wizard. If you need to turn this option off, override the **CWnd::PreCreateWindow** member function.
- **FWS_PREFIXTITLE** Shows the document name before the application name in a frame window title. For example, "Document - WordPad". You can specify the strings displayed in the Document Template Strings tab in the Application Wizard. If you need to turn this option off, override the **CWnd::PreCreateWindow** member function.
- **FWS_SNAPTOBARS** Controls sizing of the frame window that encloses a control bar when it is in a floating window rather than docked to a frame window. This style sizes the window to fit the control bar.

See Also

Other Resources

Styles Used by MFC

List-Box Styles

- **LBS_DISABLENOSCROLL** The list box shows a disabled vertical scroll bar when the list box does not contain enough items to scroll. Without this style, the scroll bar is hidden when the list box does not contain enough items.
- **LBS_EXTENDEDSEL** The user can select multiple items using the SHIFT key and the mouse or special key combinations.
- **LBS_HASSTRINGS** Specifies an owner-draw list box that contains items consisting of strings. The list box maintains the memory and pointers for the strings so the application can use the **GetText** member function to retrieve the text for a particular item.
- **LBS_MULTICOLUMN** Specifies a multicolumn list box that is scrolled horizontally. The **SetColumnWidth** member function sets the width of the columns.
- **LBS_MULTIPLESEL** String selection is toggled each time the user clicks or double-clicks the string. Any number of strings can be selected.
- **LBS_NODATA** Specifies a no-data list box. Specify this style when the count of items in the list box will exceed one thousand. A no-data list box must also have the **LBS_OWNERDRAWFIXED** style, but must not have the **LBS_SORT** or **LBS_HASSTRINGS** style.

  A no-data list box resembles an owner-drawn list box except that it contains no string or bitmap data for an item. Commands to add, insert, or delete an item always ignore any given item data; requests to find a string within the list box always fail. The system sends the **WM_DRAWITEM** message to the owner window when an item must be drawn. The itemID member of the **DRAWITEMSTRUCT** structure passed with the **WM_DRAWITEM** message specifies the line number of the item to be drawn. A no-data list box does not send a **WM_DELETEITEM** message.
- **LBS_NOINTEGRALHEIGHT** The size of the list box is exactly the size specified by the application when it created the list box. Usually, Windows sizes a list box so that the list box does not display partial items.
- **LBS_NOREDRAW** List-box display is not updated when changes are made. This style can be changed at any time by sending a **WM_SETREDRAW** message.
- **LBS_NOSEL** Specifies that the list box contains items that can be viewed but not selected.
- **LBS_NOTIFY** Parent window receives an input message whenever the user clicks or double-clicks a string.
- **LBS_OWNERDRAWFIXED** The owner of the list box is responsible for drawing its contents; the items in the list box are the same height.
- **LBS_OWNERDRAWVARIABLE** The owner of the list box is responsible for drawing its contents; the items in the list box are variable in height.
- **LBS_SORT** Strings in the list box are sorted alphabetically.
- **LBS_STANDARD** Strings in the list box are sorted alphabetically, and the parent window receives an input message whenever the user clicks or double-clicks a string. The list box contains borders on all sides.
- **LBS_USETABSTOPS** Allows a list box to recognize and expand tab characters when drawing its strings. The default tab positions are 32 dialog units. (A dialog unit is a horizontal or vertical distance. One horizontal dialog unit is equal to one-fourth of the current dialog base width unit. The dialog base units are computed based on the height and width of the current system font. The **GetDialogBaseUnits** Windows function returns the current dialog base units in pixels.) This style should not be used with **LBS_OWNERDRAWFIXED**.
- **LBS_WANTKEYBOARDINPUT** The owner of the list box receives **WM_VKEYTOITEM** or **WM_CHARTOITEM** messages whenever the user presses a key while the list box has input focus. This allows an application to perform special processing on the keyboard input.

See Also

Reference

CListBox::Create

List Box Styles

Other Resources

Styles Used by MFC

Message-Box Styles

The following message-box styles are available.

Message_Box Types

- **MB_ABORTRETRYIGNORE** The message box contains three pushbuttons: Abort, Retry, and Ignore.
- **MB_OK** The message box contains one pushbutton: OK.
- **MB_OKCANCEL** The message box contains two pushbuttons: OK and Cancel.
- **MB_RETRYCANCEL** The message box contains two pushbuttons: Retry and Cancel.
- **MB_YESNO** The message box contains two pushbuttons: Yes and No.
- **MB_YESNOCANCEL** The message box contains three pushbuttons: Yes, No, and Cancel.

Message-Box Modality

- **MB_APPLMODAL** The user must respond to the message box before continuing work in the current window. However, the user can move to the windows of other applications and work in those windows. The default is **MB_APPLMODAL** if neither **MB_SYSTEMMODAL** nor **MB_TASKMODAL** is specified.
- **MB_SYSTEMMODAL** All applications are suspended until the user responds to the message box. System-modal message boxes are used to notify the user of serious, potentially damaging errors that require immediate attention and should be used sparingly.
- **MB_TASKMODAL** Similar to **MB_APPLMODAL**, but not useful within a Microsoft Foundation class application. This flag is reserved for a calling application or library that does not have a window handle available.

Message-Box Icons

- **MB_ICONEXCLAMATION** An exclamation-point icon appears in the message box.
- **MB_ICONINFORMATION** An icon consisting of an "I" in a circle appears in the message box.
- **MB_ICONQUESTION** A question-mark icon appears in the message box.
- **MB_ICONSTOP** A stop-sign icon appears in the message box.

Message-Box Default Buttons

- **MB_DEFBUTTON1** The first button is the default. Note that the first button is always the default unless **MB_DEFBUTTON2** or **MB_DEFBUTTON3** is specified.
- **MB_DEFBUTTON2** The second button is the default.
- **MB_DEFBUTTON3** The third button is the default.

See Also

Reference

AfxMessageBox

Other Resources

Styles Used by MFC

Scroll-Bar Styles

- **SBS_BOTTOMALIGN** Used with the **SBS_HORZ** style. The bottom edge of the scroll bar is aligned with the bottom edge of the rectangle specified in the **Create** member function. The scroll bar has the default height for system scroll bars.
- **SBS_HORZ** Designates a horizontal scroll bar. If neither the **SBS_BOTTOMALIGN** nor **SBS_TOPALIGN** style is specified, the scroll bar has the height, width, and position given in the **Create** member function.
- **SBS_LEFTALIGN** Used with the **SBS_VERT** style. The left edge of the scroll bar is aligned with the left edge of the rectangle specified in the **Create** member function. The scroll bar has the default width for system scroll bars.
- **SBS_RIGHTALIGN** Used with the **SBS_VERT** style. The right edge of the scroll bar is aligned with the right edge of the rectangle specified in the **Create** member function. The scroll bar has the default width for system scroll bars.
- **SBS_SIZEBOX** Designates a size box. If neither the **SBS_SIZEBOXBOTTOMRIGHTALIGN** nor **SBS_SIZEBOXTOPLEFTALIGN** style is specified, the size box has the height, width, and position given in the **Create** member function.
- **SBS_SIZEBOXBOTTOMRIGHTALIGN** Used with the **SBS_SIZEBOX** style. The lower-right corner of the size box is aligned with the lower-right corner of the rectangle specified in the **Create** member function. The size box has the default size for system size boxes.
- **SBS_SIZEBOXTOPLEFTALIGN** Used with the **SBS_SIZEBOX** style. The upper-left corner of the size box is aligned with the upper-left corner of the rectangle specified in the **Create** member function. The size box has the default size for system size boxes.
- **SBS_SIZEGRIP** Same as **SBS_SIZEBOX**, but with a raised edge.
- **SBS_TOPALIGN** Used with the **SBS_HORZ** style. The top edge of the scroll bar is aligned with the top edge of the rectangle specified in the **Create** member function. The scroll bar has the default height for system scroll bars.
- **SBS_VERT** Designates a vertical scroll bar. If neither the **SBS_RIGHTALIGN** nor **SBS_LEFTALIGN** style is specified, the scroll bar has the height, width, and position given in the **Create** member function.

See Also

Reference

CScrollBar::Create

Scroll Bar Control Styles

Other Resources

Styles Used by MFC

Static Styles

- **SS_BITMAP** Specifies a bitmap is to be displayed in the static control. The given text is the name of a bitmap (not a filename) defined elsewhere in the resource file. The style ignores the nWidth and nHeight parameters; the control automatically sizes itself to accommodate the bitmap.
- **SS_BLACKFRAME** Specifies a box with a frame drawn with the same color as window frames. The default is black.
- **SS_BLACKRECT** Specifies a rectangle filled with the color used to draw window frames. The default is black.
- **SS_CENTER** Designates a simple rectangle and displays the given text centered in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next centered line.
- **SS_CENTERIMAGE** Specifies that, if the bitmap or icon is smaller than the client area of the static control, the rest of the client area is filled with the color of the pixel in the top left corner of the bitmap or icon. If the static control contains a single line of text, the text is centered vertically in the client area of the control.
- **SS_ENDELLIPSIS** or **SS_PATHELLIPSIS** Replaces part of the given string with ellipses, if necessary, so that the result fits in the specified rectangle.

  You can specify **SS_END_ELLIPSIS** to replace characters at the end of the string, or **SS_PATHELLIPSIS** to replace characters in the middle of the string. If the string contains backslash (\) characters, **SS_PATHELLIPSIS** preserves as much of the text after the last backslash as possible.
- **SS_ENHMETAFILE** Specifies an enhanced metafile is to be displayed in the static control. The given text is the name of a metafile. An enhanced metafile static control has a fixed size; the metafile is scaled to fit the static control's client area.
- **SS_ETCHEDFRAME** Draws the frame of the static control using the **EDGE_ETCHED** edge style.
- **SS_ETCHEDHORZ** Draws the top and bottom edges of the static control using the **EDGE_ETCHED** edge style.
- **SS_ETCHEDVERT** Draws the left and right edges of the static control using the EDGE_ETCHED edge style.
- **SS_GRAYFRAME** Specifies a box with a frame drawn with the same color as the screen background (desktop). The default is gray.
- **SS_GRAYRECT** Specifies a rectangle filled with the color used to fill the screen background. The default is gray.
- **SS_ICON** Designates an icon displayed in the dialog box. The given text is the name of an icon (not a filename) defined elsewhere in the resource file. The *nWidth* and *nHeight* parameters are ignored; the icon automatically sizes itself.
- **SS_LEFT** Designates a simple rectangle and displays the given text flush-left in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next flush-left line.
- **SS_LEFTNOWORDWRAP** Designates a simple rectangle and displays the given text flush-left in the rectangle. Tabs are expanded, but words are not wrapped. Text that extends past the end of a line is clipped.
- **SS_NOPREFIX** Unless this style is specified, Windows will interpret any ampersand (&) characters in the control's text to be accelerator prefix characters. In this case, the ampersand is removed and the next character in the string is underlined. If a static control is to contain text where this feature is not wanted, **SS_NOPREFIX** may be added. This static-control style may be included with any of the defined static controls. You can combine **SS_NOPREFIX** with other styles by using the bitwise OR operator. This is most often used when filenames or other strings that may contain an ampersand need to be displayed in a static control in a dialog box.
- **SS_NOTIFY** Sends the parent window **STN_CLICKED**, **STN_DBLCLK**, **STN_DISABLE**, and **STN_ENABLE** notification messages when the user clicks or double-clicks the control.
- **SS_OWNERDRAW** Specifies that the owner of the static control is responsible for drawing the control. The owner window receives a **WM_DRAWITEM** message whenever the control needs to be drawn.
- **SS_REALSIZEIMAGE** Prevents a static icon or bitmap control (that is, static controls that have the **SS_ICON** or **SS_BITMAP** style) from being resized as it is loaded or drawn. If the icon or bitmap is larger than the destination area, the image is clipped.

- **SS_RIGHT** Designates a simple rectangle and displays the given text flush-right in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next flush-right line.
- **SS_RIGHTJUST** Specifies that the lower right corner of a static control with the **SS_BITMAP** or **SS_ICON** style is to remain fixed when the control is resized. Only the top and left sides are adjusted to accommodate a new bitmap or icon.
- **SS_SIMPLE** Designates a simple rectangle and displays a single line of text flush-left in the rectangle. The line of text cannot be shortened or altered in any way. (The control's parent window or dialog box must not process the **WM_CTLCOLOR** message.)
- **SS_SUNKEN** Draws a half-sunken border around a static control.
- **SS_USERITEM** Specifies a user-defined item.
- **SS_WHITEFRAME** Specifies a box with a frame drawn with the same color as the window background. The default is white.
- **SS_WHITERECT** Specifies a rectangle filled with the color used to fill the window background. The default is white.
- **SS_WORDELLIPSIS** Truncates text that does not fit and adds ellipses.

See Also

Reference

CStatic::Create

DrawEdge

Static Control Styles

Other Resources

Styles Used by MFC

Window Styles

- **WS_BORDER** Creates a window that has a border.
- **WS_CAPTION** Creates a window that has a title bar (implies the **WS_BORDER** style). Cannot be used with the **WS_DLGFRAME** style.
- **WS_CHILD** Creates a child window. Cannot be used with the **WS_POPUP** style.
- **WS_CHILDWINDOW** Same as the **WS_CHILD** style.
- **WS_CLIPCHILDREN** Excludes the area occupied by child windows when you draw within the parent window. Used when you create the parent window.
- **WS_CLIPSIBLINGS** Clips child windows relative to each other; that is, when a particular child window receives a paint message, the **WS_CLIPSIBLINGS** style clips all other overlapped child windows out of the region of the child window to be updated. (If **WS_CLIPSIBLINGS** is not given and child windows overlap, when you draw within the client area of a child window, it is possible to draw within the client area of a neighboring child window.) For use with the **WS_CHILD** style only.
- **WS_DISABLED** Creates a window that is initially disabled.
- **WS_DLGFRAME** Creates a window with a double border but no title.
- **WS_GROUP** Specifies the first control of a group of controls in which the user can move from one control to the next with the arrow keys. All controls defined with the **WS_GROUP** style **FALSE** after the first control belong to the same group. The next control with the **WS_GROUP** style starts the next group (that is, one group ends where the next begins).
- **WS_HSCROLL** Creates a window that has a horizontal scroll bar.
- **WS_ICONIC** Creates a window that is initially minimized. Same as the **WS_MINIMIZE** style.
- **WS_MAXIMIZE** Creates a window of maximum size.
- **WS_MAXIMIZEBOX** Creates a window that has a Maximize button.
- **WS_MINIMIZE** Creates a window that is initially minimized. For use with the **WS_OVERLAPPED** style only.
- **WS_MINIMIZEBOX** Creates a window that has a Minimize button.
- **WS_OVERLAPPED** Creates an overlapped window. An overlapped window usually has a caption and a border.
- **WS_OVERLAPPEDWINDOW** Creates an overlapped window with the **WS_OVERLAPPED**, **WS_CAPTION**, **WS_SYSMENU**, **WS_THICKFRAME**, **WS_MINIMIZEBOX**, and **WS_MAXIMIZEBOX** styles.
- **WS_POPUP** Creates a pop-up window. Cannot be used with the **WS_CHILD** style.
- **WS_POPUPWINDOW** Creates a pop-up window with the **WS_BORDER**, **WS_POPUP**, and **WS_SYSMENU** styles. The **WS_CAPTION** style must be combined with the **WS_POPUPWINDOW** style to make the Control menu visible.
- **WS_SIZEBOX** Creates a window that has a sizing border. Same as the **WS_THICKFRAME** style.
- **WS_SYSMENU** Creates a window that has a Control-menu box in its title bar. Used only for windows with title bars.
- **WS_TABSTOP** Specifies one of any number of controls through which the user can move by using the TAB key. The TAB key moves the user to the next control specified by the **WS_TABSTOP** style.
- **WS_THICKFRAME** Creates a window with a thick frame that can be used to size the window.
- **WS_TILED** Creates an overlapped window. An overlapped window has a title bar and a border. Same as the **WS_OVERLAPPED** style.
- **WS_TILEDWINDOW** Creates an overlapped window with the **WS_OVERLAPPED**, **WS_CAPTION**, **WS_SYSMENU**, **WS_THICKFRAME**, **WS_MINIMIZEBOX**, and **WS_MAXIMIZEBOX** styles. Same as the **WS_OVERLAPPEDWINDOW** style.
- **WS_VISIBLE** Creates a window that is initially visible.
- **WS_VSCROLL** Creates a window that has a vertical scroll bar.

See Also

Reference

CWnd::Create

CWnd::CreateEx

CreateWindow

Other Resources

Styles Used by MFC

Extended Window Styles

- **WS_EX_ACCEPTFILES** Specifies that a window created with this style accepts drag-and-drop files.
- **WS_EX_APPWINDOW** Forces a top-level window onto the taskbar when the window is visible.
- **WS_EX_CLIENTEDGE** Specifies that a window has a 3D look — that is, a border with a sunken edge.
- **WS_EX_CONTEXTHELP** Includes a question mark in the title bar of the window. When the user clicks the question mark, the cursor changes to a question mark with a pointer. If the user then clicks a child window, the child receives a **WM_HELP** message.
- **WS_EX_CONTROLPARENT** Allows the user to navigate among the child windows of the window by using the TAB key.
- **WS_EX_DLGMODALFRAME** Designates a window with a double border that may (optionally) be created with a title bar when you specify the **WS_CAPTION** style flag in the *dwStyle* parameter.
- **WS_EX_LEFT** Gives window generic left-aligned properties. This is the default.
- **WS_EX_LEFTSCROLLBAR** Places a vertical scroll bar to the left of the client area.
- **WS_EX_LTRREADING** Displays the window text using left-to-right reading order properties. This is the default.
- **WS_EX_MDICHILD** Creates an MDI child window.
- **WS_EX_NOPARENTNOTIFY** Specifies that a child window created with this style will not send the **WM_PARENTNOTIFY** message to its parent window when the child window is created or destroyed.
- **WS_EX_OVERLAPPEDWINDOW** Combines the **WS_EX_CLIENTEDGE** and **WS_EX_WINDOWEDGE** styles
- **WS_EX_PALETTEWINDOW** Combines the **WS_EX_WINDOWEDGE** and **WS_EX_TOPMOST** styles.
- **WS_EX_RIGHT** Gives a window generic right-aligned properties. This depends on the window class.
- **WS_EX_RIGHTSCROLLBAR** Places a vertical scroll bar (if present) to the right of the client area. This is the default.
- **WS_EX_RTLREADING** Displays the window text using right-to-left reading order properties.
- **WS_EX_STATICEDGE** Creates a window with a three-dimensional border style intended to be used for items that do not accept user input.
- **WS_EX_TOOLWINDOW** Creates a tool window, which is a window intended to be used as a floating toolbar. A tool window has a title bar that is shorter than a normal title bar, and the window title is drawn using a smaller font. A tool window does not appear in the task bar or in the window that appears when the user presses ALT+TAB.
- **WS_EX_TOPMOST** Specifies that a window created with this style should be placed above all nontopmost windows and stay above them even when the window is deactivated. An application can use the **SetWindowPos** member function to add or remove this attribute.
- **WS_EX_TRANSPARENT** Specifies that a window created with this style is to be transparent. That is, any windows that are beneath the window are not obscured by the window. A window created with this style receives **WM_PAINT** messages only after all sibling windows beneath it have been updated.
- **WS_EX_WINDOWEDGE** Specifies that a window has a border with a raised edge.

See Also

Reference

CWnd::CreateEx

CreateWindowEx

Other Resources

Styles Used by MFC

Callback Functions Used by MFC

Three callback functions appear in the Microsoft Foundation Class Library. A description of callback functions that are passed to CDC::EnumObjects, CDC::GrayString, and CDC::SetAbortProc follows this topic. For the general usage of the callback functions, see the Remarks section of these member functions. Note that all callback functions must trap MFC exceptions before returning to Windows, since exceptions cannot be thrown across callback boundaries. For more information about exceptions, see the article Exceptions.

See Also

Other Resources

Structures, Styles, Callbacks, and Message Maps

Callback Function for CDC::EnumObjects

The *ObjectFunc* name is a placeholder for the application-supplied function name.

```
int CALLBACK EXPORT ObjectFunc(
   LPSTR lpszLogObject,
   LPSTR* lpData
);
```

Parameters

lpszLogObject

Points to a LOGPEN or LOGBRUSH data structure that contains information about the logical attributes of the object.

lpData

Points to the application-supplied data passed to the **EnumObjects** function.

Return Value

The callback function returns an **int**. The value of this return is user-defined. If the callback function returns 0, **EnumObjects** stops enumeration early.

Remarks

The actual name must be exported.

See Also

Reference

CDC::EnumObjects

Other Resources

Structures, Styles, Callbacks, and Message Maps

Callback Function for CDC::GrayString

OutputFunc is a placeholder for the application-supplied callback function name.

```
BOOL CALLBACK EXPORT OutputFunc(
   HDC hDC,
   LPARAM lpData,
   int nCount
);
```

Parameters

hDC

Identifies a memory device context with a bitmap of at least the width and height specified by *nWidth* and *nHeight* to **GrayString**.

lpData

Points to the character string to be drawn.

nCount

Specifies the number of characters to output.

Return Value

The callback function's return value must be **TRUE** to indicate success; otherwise it is **FALSE**.

Remarks

The callback function (*OutputFunc*) must draw an image relative to the coordinates (0,0) rather than (*x*, *y*).

See Also

Reference

CDC::GrayString

Other Resources

Structures, Styles, Callbacks, and Message Maps

Callback Function for CDC::SetAbortProc

The name *AbortFunc* is a placeholder for the application-supplied function name.

```
BOOL CALLBACK EXPORT AbortFunc(
   HDC hPr,
   int code
);
```

Parameters

hPr

Identifies the device context.

code

Specifies whether an error has occurred. It is 0 if no error has occurred. It is **SP_OUTOFDISK** if the Print Manager is currently out of disk space and more disk space will become available if the application waits. If *code* is **SP_OUTOFDISK**, the application does not have to abort the print job. If it does not, it must yield to the Print Manager by calling the **PeekMessage** or **GetMessage** Windows function.

Return Value

The return value of the abort-handler function is nonzero if the print job is to continue, and 0 if it is canceled.

Remarks

The actual name must be exported as described in the Remarks section of CDC::SetAbortProc.

See Also

Reference

CDC::SetAbortProc

Other Resources

Structures, Styles, Callbacks, and Message Maps

Message Maps (MFC)

This section of the reference lists all message mapping macros and all CWnd message-map entries along with the corresponding member function prototypes:

Category	Description
WM_COMMAND Message Handler	Handles **WM_COMMAND** messages generated by user menu selections or menu access keys.
Child Window Notification Message Handlers	Handle notification messages from child windows.
WM_ Message Handlers	Handle **WM_** messages, such as **WM_PAINT**.
User-Defined Message Handlers	Handle user-defined messages.

(For an explanation of the terminology and conventions used in this reference, see How to Use the Message Map Cross-Reference.)

Since Windows is a message-oriented operating system, a large portion of programming for the Windows environment involves message handling. Each time an event such as a keystroke or mouse click occurs, a message is sent to the application, which must then handle the event.

The Microsoft Foundation Class Library offers a programming model optimized for message-based programming. In this model, "message maps" are used to designate which functions will handle various messages for a particular class. Message maps contain one or more macros that specify which messages will be handled by which functions. For example, a message map containing an **ON_COMMAND** macro might look something like this:

```
BEGIN_MESSAGE_MAP( CMyDoc, CDocument )
    ON_COMMAND( ID_MYCMD, OnMyCommand )
    // ... More entries to handle additional commands
END_MESSAGE_MAP( )
```

The **ON_COMMAND** macro is used to handle command messages generated by menus, buttons, and accelerator keys. Macros are available to map the following:

Windows Messages

- Control notifications
- User-defined messages

Command Messages

- Registered user-defined messages
- User-interface update messages

Ranges of Messages

- Commands
- Update handler messages
- Control notifications

Although message-map macros are important, you generally won't have to use them directly. This is because the Properties window automatically creates message-map entries in your source files when you use it to associate message-handling functions with messages. Any time you want to edit or add a message-map entry, you can use the Properties window.

Note
The Properties window does not support message-map ranges. You must write these message-map entries yourself.

However, message maps are an important part of the Microsoft Foundation Class Library. You should understand what they do, and documentation is provided for them.

See Also

Other Resources

Structures, Styles, Callbacks, and Message Maps

Message Map Macros (MFC)

To support message maps, MFC supplies the following macros:

Message-Map Declaration and Demarcation Macros

DECLARE_MESSAGE_MAP	Declares that a message map will be used in a class to map messages to functions (must be used in the class declaration).
BEGIN_MESSAGE_MAP	Begins the definition of a message map (must be used in the class implementation).
END_MESSAGE_MAP	Ends the definition of a message map (must be used in the class implementation).

Message-Mapping Macros

ON_COMMAND	Indicates which function will handle a specified command message.
ON_CONTROL	Indicates which function will handle a specified control-notification message.
ON_MESSAGE	Indicates which function will handle a user-defined message.
ON_OLECMD	Indicates which function will handle a menu command from a DocObject or its container.
ON_REGISTERED_MESSAGE	Indicates which function will handle a registered user-defined message.
ON_REGISTERED_THREAD_MESSAGE	Indicates which function will handle a registered user-defined message when you have a **CWinThread** class.
ON_THREAD_MESSAGE	Indicates which function will handle a user-defined message when you have a **CWinThread** class.
ON_UPDATE_COMMAND_UI	Indicates which function will handle a specified user-interface update command message.

Message-Map Range Macros

ON_COMMAND_RANGE	Indicates which function will handle the range of command IDs specified in the first two parameters to the macro.
ON_UPDATE_COMMAND_UI_RANGE	Indicates which update handler will handle the range of command IDs specified in the first two parameters to the macro.
ON_CONTROL_RANGE	Indicates which function will handle notifications from the range of control IDs specified in the second and third parameters to the macro. The first parameter is a control-notification message, such as **BN_CLICKED**.

For more information on message maps, the message-map declaration and demarcation macros, and the message-mapping macros, see Message Maps and Message Handling and Mapping Topics. For more information about message-map ranges, see Handlers for Message-Map Ranges.

See Also

Reference

Message Maps (MFC)

How to: Use the Message-Map Cross-Reference

In entries labeled <memberFxn>, write your own member function for a derived CWnd class. Give your function any name you like. Other functions, such as **OnActivate**, are member functions of class **CWnd**. If called, they pass the message to the **DefWindowProc** Windows function. To process Windows notification messages, override the corresponding **CWnd** function in your derived class. Your function should call the overridden function in your base class to let the base class and Windows respond to the message.

In all cases, put the function prototype in the **CWnd**-derived class header, and code the message map entry as shown.

The following terms are used:

Term	Definition
id	Any user-defined menu item ID (**WM_COMMAND** messages) or control ID (child window notification messages).
"message" and "wNotifyCode"	Windows message IDs as defined in WINDOWS.H.
nMessageVariable	Name of a variable that contains the return value from the **RegisterWindowMessage** Windows function. It must be declared **NEAR**.

See Also

Reference

Message Maps (MFC)

WM_COMMAND Message Handler

The following map entry corresponds to the function prototype.

Map entry	Function prototype
ON_COMMAND(<id>, <memberFxn>)	afx_msg void memberFxn( );

See Also

Reference

Message Maps (MFC)

Child Window Notification Message Handlers

There are five categories of child window notification messages:

Category	Description
Generic Control Handler	Handler for generic control notification codes.
User Button Handlers	Handlers for user button notification codes.
Combo Box Handlers	Handlers for combo box notification codes.
Edit Control Handlers	Handlers for edit control notification codes.
List Box Handlers	Handlers for list box notification codes.

See Also

Reference

Message Maps (MFC)

Generic Control Handler

The following map entry corresponds to the function prototype.

Map entry	Function prototype
ON_CONTROL(<wNotifyCode>, <id>, <memberFxn>)	afx_msg void memberFxn( );

See Also

Reference

Message Maps (MFC)

User Button Handlers

The following map entries correspond to the function prototypes.

Map entry	Function prototype
ON_BN_CLICKED(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_BN_DISABLE(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_BN_DOUBLECLICKED(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_BN_HILITE(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_BN_PAINT(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_BN_UNHILITE(<id>, <memberFxn>)	afx_msg void memberFxn( );

See Also

Reference

Message Maps (MFC)

Combo Box Handlers

The following map entries correspond to the function prototypes.

Map entry	Function prototype
ON_CBN_CLOSEUP(<id>, <memberFxn>)	afx_msg void memberFxn( )
ON_CBN_DBLCLK(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_CBN_DROPDOWN(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_CBN_EDITCHANGE(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_CBN_EDITUPDATE(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_CBN_ERRSPACE(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_CBN_KILLFOCUS(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_CBN_SELCHANGE(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_CBN_SELENDCANCEL(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_CBN_SELENDOK(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_CBN_SETFOCUS(<id>, <memberFxn>)	afx_msg void memberFxn( );

See Also

Reference

Message Maps (MFC)

Edit Control Handlers

The following map entries correspond to the function prototype.

Map entry	Function prototype
ON_EN_CHANGE(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_EN_ERRSPACE(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_EN_HSCROLL(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_EN_KILLFOCUS(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_EN_MAXTEXT(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_EN_SETFOCUS(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_EN_UPDATE(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_EN_VSCROLL(<id>, <memberFxn>)	afx_msg void memberFxn( );

See Also

Reference

Message Maps (MFC)

List Box Handlers

The following map entries have the corresponding function prototype.

Map entry	Function prototype
ON_LBN_DBLCLK(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_LBN_ERRSPACE(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_LBN_KILLFOCUS(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_LBN_SELCHANGE(<id>, <memberFxn>)	afx_msg void memberFxn( );
ON_LBN_SETFOCUS(<id>, <memberFxn>)	afx_msg void memberFxn( );

See Also

Reference

Message Maps (MFC)

Handlers for WM_ Messages

The following topics correspond to the map entries.

Topic	Map entries
A - C	**ON_WM_ACTIVATE** through **ON_WM_CTLCOLOR**
D - E	**ON_WM_DEADCHAR** through **ON_WM_ERASEBKGND**
F - K	**ON_WM_FONTCHANGE** through **ON_WM_KILLFOCUS**
L - M	**ON_WM_LBUTTONDBLCLK** through **ON_WM_MOVING**
N - O	**ON_WM_NCACTIVATE** through **ON_WM_NCRBUTTONUP**
P - R	**ON_WM_PAINT** through **ON_WM_RENDERFORMAT**
S	**ON_WM_SETCURSOR** through **ON_WM_SYSKEYUP**
T - Z	**ON_WM_TIMECHANGE** through **ON_WM_WININICHANGE**

See Also

Reference

Message Maps (MFC)

WM_ Message Handlers: A - C

The following map entries correspond to the function prototype on the right.

Map entry	Function prototype
ON_WM_ACTIVATE( )	afx_msg void OnActivate(UINT, CWnd*, BOOL);
ON_WM_ACTIVATEAPP( )	afx_msg void OnActivateApp(BOOL, DWORD);
ON_WM_ASKCBFORMATNAME( )	afx_msg void OnAskCbFormatName(UINT, LPSTR);
ON_WM_CANCELMODE( )	afx_msg void OnCancelMode( );
ON_WM_CAPTURECHANGED( )	afx_msg void OnCaptureChanged(CWnd*);
ON_WM_CHANGECBCHAIN( )	afx_msg void OnChangeCbChain(HWND, HWND);
ON_WM_CHAR( )	afx_msg void OnChar(UINT, UINT, UINT);
ON_WM_CHARTOITEM( )	afx_msg int OnCharToItem(UINT, CWnd*, UINT);
ON_WM_CHILDACTIVATE( )	afx_msg void OnChildActivate( );
ON_WM_CLOSE( )	afx_msg void OnClose( );
ON_WM_COMPACTING( )	afx_msg void OnCompacting(UINT);
ON_WM_COMPAREITEM( )	afx_msg int OnCompareItem(LPCOMPAREITEMSTRUCT);
ON_WM_CONTEXTMENU( )	afx_msg void OnContextMenu(CWnd*, CPoint);
ON_WM_COPYDATA( )	afx_msg BOOL OnCopyData(CWnd* pWnd, COPYDATASTRUCT* pCopyDataStruct);
ON_WM_CREATE( )	afx_msg int OnCreate(LPCREATESTRUCT);
ON_WM_CTLCOLOR( )	afx_msg HBRUSH OnCtlColor(CDC*, CWnd*, UINT);

See Also

Reference

Message Maps (MFC)

Handlers for WM_ Messages

WM_ Message Handlers: D - E

The following map entries correspond to the function prototypes.

Map entry	Function prototype
ON_WM_DEADCHAR( )	afx_msg void OnDeadChar(UINT, UINT, UINT);
ON_WM_DELETEITEM( )	afx_msg void OnDeleteItem(LPDELETEITEMSTRUCT);
ON_WM_DESTROY( )	afx_msg void OnDestroy( );
ON_WM_DESTROYCLIPBOARD( )	afx_msg void OnDestroyClipboard( );
ON_WM_DEVICECHANGE( )	afx_msg void OnDeviceChange(UINT, DWORD);
ON_WM_DEVMODECHANGE( )	afx_msg void OnDevModeChange(LPSTR);
ON_WM_DRAWCLIPBOARD( )	afx_msg void OnDrawClipboard( );
ON_WM_DRAWITEM( )	afx_msg void OnDrawItem(LPDRAWITEMSTRUCT);
ON_WM_DROPFILES( )	afx_msg void OnDropFiles(HDROP);
ON_WM_ENABLE( )	afx_msg void OnEnable(BOOL);
ON_WM_ENDSESSION( )	afx_msg void OnEndSession(BOOL);
ON_WM_ENTERIDLE( )	afx_msg void OnEnterIdle(UINT, CWnd*);
ON_WM_ERASEBKGND( )	afx_msg BOOL OnEraseBkgnd(CDC *);

See Also

Reference

Message Maps (MFC)

Handlers for WM_ Messages

WM_ Message Handlers: F - K

The following map entries correspond to the function prototypes.

Map entry	Function prototype
ON_WM_FONTCHANGE( )	afx_msg void OnFontChange( );
ON_WM_GETDLGCODE( )	afx_msg UINT OnGetDlgCode( );
ON_WM_GETMINMAXINFO( )	afx_msg void OnGetMinMaxInfo(LPPOINT);
ON_WM_HELPINFO( )	afx_msg BOOL OnHelpInfo(HELPINFO*);
ON_WM_HSCROLL( )	afx_msg void OnHScroll(UINT, UINT, CWnd*);
ON_WM_HSCROLLCLIPBOARD( )	afx_msg void OnHScrollClipboard CWnd*, UINT, UINT);
ON_WM_ICONERASEBKGND( )	afx_msg void OnIconEraseBkgnd(CDC *);
ON_WM_INITMENU( )	afx_msg void OnInitMenu(CMenu *);
ON_WM_INITMENUPOPUP( )	afx_msg void OnInitMenuPopup(CMenu *, UINT, BOOL);
ON_WM_KEYDOWN( )	afx_msg void OnKeyDown(UINT, UINT, UINT);
ON_WM_KEYUP( )	afx_msg void OnKeyUp(UINT, UINT, UINT);
ON_WM_KILLFOCUS( )	afx_msg void OnKillFocus(CWnd*);

See Also

Reference

Message Maps (MFC)

Handlers for WM_ Messages

WM_ Message Handlers: L - M

The following map entries correspond to the function prototypes.

Map entry	Function prototype
ON_WM_LBUTTONDBLCLK( )	afx_msg void OnLButtonDblClk(UINT, CPoint);
ON_WM_LBUTTONDOWN( )	afx_msg void OnLButtonDown(UINT, CPoint);
ON_WM_LBUTTONUP( )	afx_msg void OnLButtonUp(UINT, CPoint);
ON_WM_MBUTTONDBLCLK( )	afx_msg void OnMButtonDblClk(UINT, CPoint);
ON_WM_MBUTTONDOWN( )	afx_msg void OnMButtonDown(UINT, CPoint);
ON_WM_MBUTTONUP( )	afx_msg void OnMButtonUp(UINT, CPoint);
ON_WM_MDIACTIVATE( )	afx_msg void OnMDIActivate(BOOL, CWnd*, CWnd*);
ON_WM_MEASUREITEM( )	afx_msg void OnMeasureItem(LPMEASUREITEMSTRUCT);
ON_WM_MENUCHAR( )	afx_msg LONG OnMenuChar(UINT, UINT, CMenu *);
ON_WM_MENUSELECT( )	afx_msg void OnMenuSelect(UINT, UINT, HMENU);
ON_WM_MOUSEACTIVATE( )	afx_msg int OnMouseActivate(CWnd*, UINT, UINT);
ON_WM_MOUSEMOVE( )	afx_msg void OnMouseMove(UINT, CPoint);
ON_WM_MOUSEWHEEL( )	afx_msg BOOL OnMouseWheel(UINT, short, CPoint);
ON_WM_MOVE( )	afx_msg void OnMove(int, int);
ON_WM_MOVING( )	afx_msg void OnMoving(UINT, LPRECT);

See Also

Reference

Message Maps (MFC)
Handlers for WM_ Messages

WM_ Message Handlers: N - O

The following map entries correspond to the function prototypes.

Map entry	Function prototype
ON_WM_NCACTIVATE( )	afx_msg BOOL OnNcActivate(BOOL);
ON_WM_NCCALCSIZE( )	afx_msg void OnNcCalcSize(BOOL, NCCALCSIZE_PARAMS FAR*);
ON_WM_NCCREATE( )	afx_msg BOOL OnNcCreate(LPCREATESTRUCT);
ON_WM_NCDESTROY( )	afx_msg void OnNcDestroy( );
ON_WM_NCHITTEST( )	afx_msg LRESULT OnNcHitTest(CPoint);
ON_WM_NCLBUTTONDBLCLK( )	afx_msg void OnNcLButtonDblClk(UINT, CPoint);
ON_WM_NCLBUTTONDOWN( )	afx_msg void OnNcLButtonDown(UINT, CPoint);
ON_WM_NCLBUTTONUP( )	afx_msg void OnNcLButtonUp(UINT, CPoint);
ON_WM_NCMBUTTONDBLCLK( )	afx_msg void OnNcMButtonDblClk(UINT, CPoint);
ON_WM_NCMBUTTONDOWN( )	afx_msg void OnNcMButtonDown(UINT, CPoint);
ON_WM_NCMBUTTONUP( )	afx_msg void OnNcMButtonUp(UINT, CPoint);
ON_WM_NCMOUSEMOVE( )	afx_msg void OnNcMouseMove(UINT, CPoint);
ON_WM_NCPAINT( )	afx_msg void OnNcPaint( );
ON_WM_NCRBUTTONDBLCLK( )	afx_msg void OnNcRButtonDblClk(UINT, CPoint);
ON_WM_NCRBUTTONDOWN( )	afx_msg void OnNcRButtonDown(UINT, CPoint);
ON_WM_NCRBUTTONUP( )	afx_msg void OnNcRButtonUp(UINT, CPoint);

See Also

Reference

Message Maps (MFC)
Handlers for WM_ Messages

WM_ Messages: P - R

The following map entries correspond to the function prototypes.

Map entry	Function prototype
ON_WM_PAINT( )	afx_msg void OnPaint( );
ON_WM_PAINTCLIPBOARD( )	afx_msg void OnPaintClipboard(CWnd*, HANDLE);
ON_WM_PALETTECHANGED( )	afx_msg void OnPaletteChanged(CWnd*);
ON_WM_PALETTEISCHANGING( )	afx_msg void OnPaletteIsChanging(CWnd*);
ON_WM_PARENTNOTIFY( )	afx_msg void OnParentNotify(UINT, LONG);
ON_WM_QUERYDRAGICON( )	afx_msg HCURSOR OnQueryDragIcon( );
ON_WM_QUERYENDSESSION( )	afx_msg BOOL OnQueryEndSession( );
ON_WM_QUERYNEWPALETTE( )	afx_msg BOOL OnQueryNewPalette( );
ON_WM_QUERYOPEN( )	afx_msg BOOL OnQueryOpen( );
ON_WM_RBUTTONDBLCLK( )	afx_msg void OnRButtonDblClk(UINT, CPoint);
ON_WM_RBUTTONDOWN( )	afx_msg void OnRButtonDown(UINT, CPoint);
ON_WM_RBUTTONUP( )	afx_msg void OnRButtonUp(UINT, CPoint);
ON_WM_RENDERALLFORMATS( )	afx_msg void OnRenderAllFormats( );
ON_WM_RENDERFORMAT( )	afx_msg void OnRenderFormat(UINT);

See Also

Reference

Message Maps (MFC)
Handlers for WM_ Messages

WM_ Messages: S

The following map entries correspond to the function prototypes.

Map entry	Function prototype
ON_WM_SETCURSOR( )	afx_msg BOOL OnSetCursor(CWnd*, UINT, UINT);
ON_WM_SETFOCUS( )	afx_msg void OnSetFocus(CWnd*);
ON_WM_SETTINGCHANGE( )	afx_msg void OnSettingChange(UINT uFlags, LPCTSTR lpszSection);
ON_WM_SHOWWINDOW( )	afx_msg void OnShowWindow(BOOL, UINT);
ON_WM_SIZE( )	afx_msg void OnSize(UINT, int, int);
ON_WM_SIZECLIPBOARD( )	afx_msg void OnSizeClipboard(CWnd*, HANDLE);
ON_WM_SIZING( )	afx_msg void OnSizing(UINT, LPRECT);
ON_WM_SPOOLERSTATUS( )	afx_msg void OnSpoolerStatus(UINT, UINT);
ON_WM_STYLECHANGED( )	afx_msg void OnStyleChanged(int, LPSTYLESTRUCT);
ON_WM_STYLECHANGING( )	afx_msg void OnStyleChanging(int, LPSTYLESTRUCT);
ON_WM_SYSCHAR( )	afx_msg void OnSysChar(UINT, UINT, UINT);
ON_WM_SYSCOLORCHANGE( )	afx_msg void OnSysColorChange( );
ON_WM_SYSCOMMAND( )	afx_msg void OnSysCommand(UINT, LONG);
ON_WM_SYSDEADCHAR( )	afx_msg void OnSysDeadChar(UINT, UINT, UINT);
ON_WM_SYSKEYDOWN( )	afx_msg void OnSysKeyDown(UINT, UINT, UINT);
ON_WM_SYSKEYUP( )	afx_msg void OnSysKeyUp(UINT, UINT, UINT);

See Also

Reference

Message Maps (MFC)
Handlers for WM_ Messages

WM_ Messages: T - Z

The following map entries correspond to the function prototypes.

Map entry	Function prototype
ON_WM_TCARD( )	afx_msg void OnTCard(UINT, DWORD);
ON_WM_TIMECHANGE( )	afx_msg void OnTimeChange( );
ON_WM_TIMER( )	afx_msg void OnTimer(UINT);
ON_WM_VKEYTOITEM( )	afx_msg int OnVKeyToItem(UINT, CWnd*, UINT);
ON_WM_VSCROLL( )	afx_msg void OnVScroll(UINT, UINT, CWnd*);
ON_WM_VSCROLLCLIPBOARD( )	afx_msg void OnVScrollClipboard(CWnd*, UINT, UINT);
ON_WM_WINDOWPOSCHANGED( )	afx_msg void OnWindowPosChanged(WINDOWPOS* lpwndpos);
ON_WM_WINDOWPOSCHANGING( )	afx_msg void OnWindowPosChanging(WINDOWPOS* lpwndpos);
ON_WM_WININICHANGE( )	afx_msg void OnWinIniChange(LPSTR);

See Also

Reference

Message Maps (MFC)
Handlers for WM_ Messages

User-Defined Handlers

The following map entries correspond to the function prototypes.

Map entry	Function prototype
ON_MESSAGE(<message>, <memberFxn>)	afx_msg LRESULT memberFxn(WPARAM, LPARAM);
ON_REGISTERED_MESSAGE(<nMessageVariable>, <memberFxn>)	afx_msg LRESULT memberFxn(WPARAM, LPARAM);
ON_THREAD_MESSAGE(<message>, <memberFxn>)	afx_msg void memberFxn(WPARAM, LPARAM);
ON_REGISTERED_THREAD_MESSAGE(<nMessageVariable>, <memberFxn>)	afx_msg void memberFxn(WPARAM, LPARAM);

See Also

Reference

Message Maps (MFC)

Handlers for WM_ Messages

Obsolete MFC Topics

The following functions, classes, and variables are no longer implemented in MFC. Refer to the topics listed below for details.

- CDatabase::InWaitForDataSource
- CDatabase::OnWaitForDataSource
- CDatabase::SetSynchronousMode
- CFile::ReadHuge
- CFile::WriteHuge
- CPropertyPageEx
- CPropertySheetEx
- CRecordset::OnWaitForDataSource
- CSliderCtrl::VerifyPos
- CTabCtrl::GetBkColor
- CTabCtrl::SetBkColor
- CWinApp::InitApplication
- CWinApp::Enable3dControls
- CWinApp::SetDialogBkColor
- CWnd::GetSuperWndProcAddr
- afxTraceEnabled
- afxTraceFlags
- TRACE0
- TRACE1
- TRACE2
- TRACE3

AfxOleTypeMatchGuid

As of MFC 7.0, AfxOleTypeMatchGuid is obsolete.

Remarks

This function was only used in the implementation of the stock OLE property pages, and in MFC 7.0, MFC no longer implements these property pages (in msstkppg.dll).

See Also

Concepts

MFC Macros and Globals

afxTraceEnabled

As of MFC 7.0, **afxTraceEnabled** is obsolete. Comparable functionality is provided by ATLTRACE2.

afxTraceFlags

As of MFC 7.0, **afxTraceFlags** is obsolete. Comparable functionality is provided by ATLTRACE2.

CArchive::IsByteSwapping

As of MFC 8.0, **CArchive::IsByteSwapping** is obsolete and is no longer supported.

```
BOOL IsByteSwapping( ) const;
```

Return Value

Always returns FALSE.

Remarks

Determines whether the archive is byte swapping.

See Also

Reference

CArchive Class

Hierarchy Chart

Other Resources

CArchive Members

CDatabase::InWaitForDataSource

In MFC 4.2, **CDatabase::InWaitForDataSource** became obsolete.

Remarks

The MFC ODBC classes now use only synchronous processing. To start an asychronous operation, you must directly call the ODBC API function **SQLSetConnectOption**. For more information, see the topic "Executing Functions Asynchronously" in the *Platform SDK*.

See Also

Reference

CDatabase Class

Hierarchy Chart

Other Resources

CDatabase Members

CDatabase::OnWaitForDataSource

In MFC 4.2, **CDatabase::OnWaitForDataSource** became obsolete.

Remarks

The MFC ODBC classes now use only synchronous processing. To start an asychronous operation, you must directly call the ODBC API function **SQLSetConnectOption**. For more information, see the topic "Executing Functions Asynchronously" in the *Platform SDK*.

See Also

Reference

CDatabase Class

Hierarchy Chart

Other Resources

CDatabase Members

CDatabase::SetSynchronousMode

In MFC 4.2, **CDatabase::SetSynchronousMode** became obsolete.

Remarks

The MFC ODBC classes now use only synchronous processing. To start an asychronous operation, you must directly call the ODBC API function **SQLSetConnectOption**. For more information, see the topic "Executing Functions Asynchronously" in the *Platform SDK*.

Note
Although **SetSynchronousMode** is obsolete, your code will still compile if you use this member function. **SetSynchronous Mode** will simply generate a **TRACE** message that it is doing nothing.

See Also

Reference

CDatabase Class

Hierarchy Chart

Other Resources

CDatabase Members

CFile::ReadHuge

ReadHuge is obsolete and is no longer supported.

Remarks

Use Read instead.

See Also

Reference

CFile Class

Hierarchy Chart

Other Resources

CFile Members

CFile::WriteHuge

WriteHuge is obsolete and is no longer supported.

Remarks

Use Write instead.

See Also

Reference

CFile Class

Hierarchy Chart

Other Resources

CFile Members

CInternetConnection::QueryOption

CInternetConnection::QueryOption is obsolete and is no longer supported.

Remarks

CInternetConnection::QueryOption was never a public function, but it is now obsolete and should not be used.

See Also

Reference

CInternetConnection Class
Hierarchy Chart
CInternetSession Class
CGopherConnection Class
CFtpConnection Class
CHttpConnection Class

Other Resources

CInternetConnection Members

CInternetFile::QueryOption

CInternetFile::QueryOption is obsolete and is no longer supported.

Remarks

CInternetFile::QueryOption was never a public function, but it is now obsolete and should not be used.

See Also

Reference

CInternetFile Class

Hierarchy Chart

Concepts

CInternetFile Members

CPropertyPageEx

As of MFC 7.0, **CPropertyPageEx** is obsolete. Its functionality has been included in its former base class, CPropertyPage.

CPropertySheetEx

As of MFC 7.0, **CPropertySheetEx** is obsolete. Its functionality has been included in its former base class, CPropertySheet.

CRecordset::OnWaitForDataSource

In MFC 4.2, **CRecordset::OnWaitForDataSource** became obsolete.

Remarks

The MFC ODBC classes now use only synchronous processing. To start an asychronous operation, you must directly call the ODBC API function **SQLSetConnectOption**. For more information, see the topic "Executing Functions Asynchronously" in the *Platform SDK*.

See Also

Reference

CRecordset Class

Hierarchy Chart

Other Resources

CRecordset Members

CSliderCtrl::VerifyPos

The **VerifyPos** function is obsolete and is no longer supported.

See Also

Reference

CSliderCtrl Class

Hierarchy Chart

Other Resources

CSliderCtrl Members

CTabCtrl::GetBkColor

The **GetBkColor** and **SetBkColor** member functions are not valid for class **CTabCtrl**.

Remarks

These member functions are not implemented in MFC because the underlying Windows 95/98 **TCM_SETCOLOR** message was not implemented.

See Also

Reference

CTabCtrl Class

Hierarchy Chart

COLORREF

Other Resources

CTabCtrl Members

CTabCtrl::SetBkColor

The **SetBkColor** and **GetBkColor** member functions are not valid for class **CTabCtrl**.

Remarks

These member functions are not implemented in MFC because the underlying Windows 95/98 **TCM_SETCOLOR** message was not implemented.

See Also

Reference

CTabCtrl Class
Hierarchy Chart
COLORREF

Other Resources

CTabCtrl Members

CWinApp::Enable3dControls

In MFC 5.0, **Enable3dControls** and **Enable3dControlsStatic** are obsolete because their functionality is incorporated into Microsoft's 32-bit and 64-bit operating systems. Applications targeting Microsoft's 32-bit and 64-bit operating systems may remove calls to these methods or ignore the deprecation warning.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::InitInstance
CWinApp::SetDialogBkColor

Other Resources

CWinApp Members

CWinApp::InitApplication

The **CWinApp::InitApplication** member function is obsolete in MFC.

Remarks

An initialization that you would have done in **InitApplication** should be moved to InitInstance. If you override **CWinApp::InitApplication**, and you do not call the base class function, you will leak the **CDocTemplate** objects that were added through **CWinApp::AddDocTemplate**.

See Also

Reference

CWinApp Class
Hierarchy Chart
CWinApp::InitInstance

Other Resources

CWinApp Members

CWinApp::SetDialogBkColor

This function is obsolete.

Remarks

To set the background color of the dialog box, you must handle **WM_CTLCOLOR**. This message changes the color of the specified dialog box only.

See Also

Reference

CWinApp Class

Hierarchy Chart

Other Resources

CWinApp Members

CWnd::GetSuperWndProcAddr

This function is obsolete.

Remarks

You will not need to override this function because the default implementation in **CWnd** now stores this pointer in all **CWnd** objects.

See Also

Reference

CWnd Class

Hierarchy Chart

Concepts

CWnd Members

TRACE0

The TRACE macros are obsolete, use ATLTRACE2.

TRACE0 is similar to **TRACE** and is one variant of a group of trace macros that you can use for debug output.

```
TRACE0(
      exp
)
```

Parameters

exp

A format string as used in the run-time function **printf**.

Remarks

The group includes:

- **TRACE0** Takes a format string **(Only)** and can be used for simple text messages, which are dumped to **afxDump**.
- **TRACE1** Takes a format string plus one argument (one variable that is dumped to **afxDump**).
- **TRACE2** Takes a format string plus two arguments (two variables that are dumped to **afxDump**).
- **TRACE3** Takes a format string plus three arguments (three variables that are dumped to **afxDump**).

TRACE0 does nothing if you have compiled a release version of your application. As with **TRACE**, it only dumps data to **afxDump** if you have compiled a debug version of your application.

Note
This macro is available only in the debug version of MFC.

Example

```
// Example for TRACE0
TRACE0( "Start Dump of MyClass members:" );

// Another example for TRACE0
// This works, but it is easier to use TRACE() or TRACE1()
DWORD dwLastError = ::GetLastError();
CString str;
str.Format("The last error code for this thread is %d\n", dwLastError);
TRACE0( (LPCTSTR) str);

// These are all normal uses of TRACE0()
TRACE0("This message will be output. ");
TRACE0("This text is on the same line.\n");
TRACE0("This text is on the next line.\n");
```

See Also

Reference

TRACE
TRACE1
TRACE2
TRACE3

Concepts

MFC Macros and Globals

TRACE1

The TRACE macros are obsolete, use ATLTRACE2.

See TRACE0 for a description of the **TRACE1** macro.

```
TRACE1(
      exp,
      param1
)
```

Parameters

exp

A format string as used in the run-time function **printf**.

param1

The name of the variable whose value should be dumped.

Example

```
// Example for TRACE1
int i = 1;
TRACE1( "Integer = %d\n", i );
// Output: 'Integer = 1'

// Another example for TRACE1
DWORD dwLastError = ::GetLastError();
TRACE1("The last error code for this thread is %d\n", dwLastError);
```

See Also

Concepts

MFC Macros and Globals

TRACE2

The TRACE macros are obsolete, use ATLTRACE2.

See TRACE0 for a description of the **TRACE2** macro.

```
TRACE2(
      exp,
      param1,
      param2
)
```

Parameters

exp

A format string as used in the run-time function **printf**.

param1

The name of the variable whose value should be dumped.

param2

The name of the variable whose value should be dumped.

Example

```
// Example for TRACE2
int i = 1;
char sz[] = "one";
TRACE2( "Integer = %d, String = %s\n", i, sz );
// Output: 'Integer = 1, String = one'

// Another example for TRACE2
// Get major and minor version numbers of Windows
DWORD dwVersion = GetVersion();
DWORD dwWindowsMajorVersion = (DWORD)(LOBYTE(LOWORD(dwVersion)));
DWORD dwWindowsMinorVersion = (DWORD)(HIBYTE(LOWORD(dwVersion)));

TRACE2("This is Windows version %d.%2.2d.\n", dwWindowsMajorVersion, dwWindowsMinorVersion)
;
```

See Also

Concepts

MFC Macros and Globals

TRACE3

The TRACE macros are obsolete, use ATLTRACE2.

See TRACE0 for a description of the **TRACE3** macro.

```
TRACE3(
      exp,
      param1,
      param2,
      param3
)
```

Parameters

exp

A format string as used in the run-time function **printf**.

param1

The name of the variable whose value should be dumped.

param2

The name of the variable whose value should be dumped.

param3

The name of the variable whose value should be dumped.

Example

```
// Example for TRACE3
// get the display context
HDC hdc = ::GetDC(NULL);

// Get information about the display
int nVertRes = ::GetDeviceCaps(hdc, VERTRES);
int nHorzRes = ::GetDeviceCaps(hdc, HORZRES);
int nDepth = ::GetDeviceCaps(hdc, BITSPIXEL);

// Done with the DC, give it back
::ReleaseDC(hdc);

// Tell the user
TRACE3("Your screen is at %d by %d resolution, with %d bits per pixel\n",
   nHorzRes, nVertRes, nDepth);
```

See Also

Concepts

MFC Macros and Globals

OLE DB Templates

The OLE DB templates make OLE DB data access easier. Visual C++ provides template classes that implement many of the commonly used OLE DB interfaces. You can use the Visual C++ consumer templates to write an OLE DB client (consumer) application and the provider templates to write a server (provider) application.

In This Section

OLE DB Consumer Templates

Provides links to the OLE DB Consumer Template classes listed by usage.

OLE DB Provider Templates

Provides links to the OLE DB Provider Template classes listed by usage.

Related Sections

OLE DB Programming Overview

Describes what OLE DB is and discusses Universal Data Access in OLE DB.

OLE DB Programmer's Reference

Provides links to the Platform SDK documentation on OLE DB.

OLE DB Templates Samples

Provides links to the OLE DB Templates samples that show how to use OLE DB in your projects.

OLE DB Consumer Templates

Discusses conceptual material about the OLE DB Consumer Templates.

OLE DB Provider Templates

Discusses conceptual material about the OLE DB Provider Templates.

Creating an OLE DB Consumer

Describes creating an OLE DB Templates consumer either with or without the ATL OLE DB Consumer Wizard.

Creating a Simple Read-Only Provider

Describes creating a simple read-only provider that reads a pair of strings.

Creating an Updatable Provider

Describes creating providers that can write to the data store.

OLE DB Consumer Attributes

Discusses a simplified interface to inject code based on the OLE DB Consumer Templates to create working OLE DB consumers.

Active Template Library (ATL) Reference

Provides reference material for the ATL Library, a set of template-based C++ classes that simplify the programming of COM objects.

Visual C++ Libraries

Provides links to the various libraries provided with Visual C++, including ATL, MFC, OLE DB Templates, the C run-time library, and the Standard C++ Library.

Debugging

Provides links to using the Visual Studio debugger to correct logic errors in your application or stored procedures.

OLE DB Consumer Templates Reference

The OLE DB Consumer Templates contain the following classes. The reference material also includes topics on the macros for OLE DB Consumer Templates.

Session Classes

CDataConnection

Manages the connection with the data source. This is a useful class for creating clients because it encapsulates necessary objects (data source and session) and some of the work you need to do when connecting to a data source.

CDataSource

Corresponds to an OLE DB data source object, representing a connection through a provider to a data source. One or more database sessions, each represented by a **CSession** object, can take place on a single connection.

CEnumerator

Corresponds to an OLE DB enumerator object, which retrieves rowset information about available data sources.

CEnumeratorAccessor

Used by **CEnumerator** to access the data from the enumerator rowset. This rowset consists of the data sources and enumerators visible from the current enumerator.

CSession

Represents a single database access session. One or more sessions can be associated with each **CDataSource** object.

Accessor Classes

CAccessor

Used for records that are statically bound to a data source. Use this accessor class when you know the structure of the data source.

CAccessorBase

Base class for all accessor classes.

CDynamicAccessor

An accessor that can be created at run time, based on the column information of the rowset. Use this class to retrieve data if you do not know the structure of the data source.

CDynamicParameterAccessor

An accessor that can be used when command types are unknown. Obtains the parameter information by calling the **ICommandWithParameters** interface, if the provider supports the interface.

CDynamicStringAccessor

Allows you to access a data source when you have no knowledge of the database's underlying structure.

CDynamicStringAccessorA

Similar to **CDynamicStringAccessor** except that this class requests data accessed from the data store as ANSI string data.

CDynamicStringAccessorW

Similar to **CDynamicStringAccessor** except that this class requests data accessed from the data store as UNICODE string data.

CManualAccessor

An accessor with methods to handle both columns and command parameters. With this class, you can use any data types as long as the provider can convert the type.

CNoAccessor

Can be used as a template argument when you do not want the class to support parameters or output columns.

CXMLAccessor

Similar to **CDynamicStringAccessor** except that this class converts all data accessed from the data store as XML-formatted (tagged) data.

Rowset Classes

CAccessorRowset

Encapsulates a rowset and its associated accessors.

CArrayRowset

Used to access elements of a rowset using array syntax.

CBulkRowset

Used to fetch and manipulate rows in bulk by retrieving multiple row handles with a single call.

CNoRowset

Can be used as a template argument if the command does not return a rowset.

CRestrictions

Used to specify restrictions for schema rowsets.

CRowset

Used to manipulate, set, and retrieve rowset data.

CStreamRowset

Returns an **ISequentialStream** object rather than a rowset; you then use the **Read** method to retrieve data in XML format. (SQL Server 2000 does the formatting; note that this feature works with SQL Server 2000 only.)

IRowsetNotifyImpl

Provides a dummy implementation for **IRowsetNotify**, with empty functions for the **IRowsetNotify** methods **OnFieldChange**, **OnRowChange**, and **OnRowsetChange**.

Schema Rowset Classes and Typedef Classes

The OLE DB Templates provide you with a set of classes that correspond to the OLE DB schema rowsets.

Command Classes

CCommand

Used to set and execute a parameter-based OLE DB command. To merely open a simple rowset, use **CTable** instead.

CMultipleResults

Used as a template argument for the **CCommand** template when you want the command to handle multiple result sets.

CNoAccessor

Used as a template argument for template classes, such as **CCommand** and **CTable**, that take an accessor class argument. Use **CNoAccessor** if you do not want the class to support parameters or output columns.

CNoMultipleResults

Used as a template argument for the **CCommand** template when you want the command to handle a single rowset. **CNoMultipleResults** is the default value for the template argument.

CNoRowset

Used as a template argument for **CCommand** or **CTable** if the command or table does not return a rowset.

CTable

Used to access a simple rowset with no parameters.

Property Classes

CDBPropIDSet

Used to pass an array of property IDs for which the consumer wants property information. The properties belong to one

property set.

CDBPropSet

Used to set properties on a provider.

Bookmark Class

CBookmark

Used as an index for accessing data in a rowset.

Error Class

CDBErrorInfo

Used to retrieve OLE DB error information.

See Also

Reference

ConfirmUser

ShowImage

Concepts

OLE DB Provider Templates Reference

Other Resources

OLE DB Templates

CAccessor Class

Represents one of the accessor types.

```
template < class T >
class CAccessor : public CAccessorBase, public T
```

Parameters

T

The user record class.

Remarks

It is used when a record is statically bound to a data source. The record contains the buffer. This class supports multiple accessors on a rowset.

Use this accessor type when you know the structure and the type of the database.

If your accessor contains fields that point to memory (such as a **BSTR** or interface) that must be freed, call the member function CAccessorRowset::FreeRecordMemory before the next record is read.

Requirements

Header: atldbcli.h

See Also

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

CAccessorBase Class

All accessors in the OLE DB Templates derive from this class. **CAccessorBase** allows one rowset to manage multiple accessors. It also provides binding for both parameters and output columns.

```
// Replace with syntax
```

Requirements

Header: atldbcli.h

See Also

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

Other Resources

CAccessorBase Members

CAccessorBase Members

Methods

Close	Closes the accessors.
GetHAccessor	Retrieves the accessor handle.
GetNumAccessors	Retrieves the number of accessors created by the class.
IsAutoAccessor	Tests whether the specified accessor is an autoaccessor.
ReleaseAccessors	Releases the accessors.

See Also

Reference

CAccessorBase Class

CAccessorBase Methods

For information about the methods in **CAccessorBase**, see CAccessorBase Members.

CAccessorBase::Close

Closes the accessors.

```
void Close( );
```

Remarks

You must call ReleaseAccessors first.

See Also

Reference

CAccessorBase Class

Other Resources

CAccessorBase Members

CAccessorBase::GetHAccessor

Retrieves the accessor handle of a specified accessor.

```
HACCESSOR GetHAccessor(
   ULONG nAccessor
) const;
```

Parameters

nAccessor

[in] The zero-offset number for the accessor.

Return Value

The accessor handle.

See Also

Reference

CAccessorBase Class

Other Resources

CAccessorBase Members

CAccessorBase::GetNumAccessors

Retrieves the number of accessors created by the class.

```
ULONG GetNumAccessors( ) const;
```

Return Value

The number of accessors created by the class.

See Also

Reference

CAccessorBase Class

Other Resources

CAccessorBase Members

CAccessorBase::IsAutoAccessor

Returns true if data is automatically retrieved for the accessor during a Move operation.

```
bool IsAutoAccessor(
   ULONG nAccessor
) const;
```

Parameters

nAccessor

[in] The zero-offset number for the accessor.

Return Value

Returns **true** if the accessor is an autoaccessor. Otherwise, it returns **false**.

See Also

Reference

CAccessorBase Class

Other Resources

CAccessorBase Members

CAccessorBase::ReleaseAccessors

Releases the accessors created by the class.

```
HRESULT ReleaseAccessors(
   IUnknown* pUnk
);
```

Parameters

pUnk

[in] A pointer to an **IUnknown** interface for the COM object for which the accessors have been created.

Return Value

A standard **HRESULT**.

Remarks

Called from CAccessorRowset::Close.

See Also

Reference

CAccessorBase Class

Other Resources

CAccessorBase Members

CAccessorRowset Class

Encapsulates a rowset and its associated accessors in a single class.

```
template <
   class TAccessor = CNoAccessor,
   template <typename T> class TRowset = CRowset
>
class CAccessorRowset :
   public TAccessor,
   public TRowset<TAccessor>
```

Parameters

TAccessor

An accessor class.

TRowset

A rowset class.

Remarks

Class *TAccessor* manages the accessor. Class *TRowset* manages the rowset.

Requirements

Header: atldbcli.h

Example

See the MantaWeb Sample and the OnlineAddressBook Sample.

See Also

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

Other Resources

CAccessorRowset Members

CAccessorRowset Members

Methods

Bind	Creates bindings (used when **bBind** is specified as false in CCommand::Open).
CAccessorRowset	Constructor.
Close	Closes the rowset and any accessors.
FreeRecordMemory	Frees any columns in the current record that need to be freed.
GetColumnInfo	Implements IColumnsInfo::GetColumnInfo.

See Also

Reference

CAccessorRowset Class

CAccessorRowset Methods

For information about the methods in **CAccessorRowset**, see CAccessorRowset Members.

CAccessorRowset::Bind

Creates the bindings if you specified **bBind** as false in CCommand::Open.

```
HRESULT Bind( );
```

Return Value

A standard **HRESULT**.

See Also

Reference

CAccessorRowset Class

Other Resources

CAccessorRowset Members

CAccessorRowset::CAccessorRowset

Initializes the **CAccessorRowset** object.

```
CAccessorRowset( );
```

See Also

Reference

CAccessorRowset Class

Other Resources

CAccessorRowset Members

CAccessorRowset::Close

Releases any active accessors and the rowset.

```
void Close( );
```

Remarks

Releases any associated memory.

Example

See the MantaWeb Sample and the OnlineAddressBook Sample.

See Also

Reference

CAccessorRowset Class

Other Resources

CAccessorRowset Members

CAccessorRowset::FreeRecordMemory

Frees any columns in the current record that need to be freed.

```
void FreeRecordMemory( );
```

Remarks

Calls SysFreeString, described in the *Platform SDK*, on any **BSTR**s and **Release** on any interfaces

See Also

Reference

CAccessorRowset Class

Other Resources

CAccessorRowset Members

CAccessorRowset::GetColumnInfo

Gets column information from the opened rowset.

```
HRESULT GetColumnInfo(
   DBORDINAL* pulColumns,
   DBCOLUMNINFO** ppColumnInfo,
   LPOLESTR* ppStrings
) const;
HRESULT GetColumnInfo(
   DBORDINAL* pColumns,
   DBCOLUMNINFO** ppColumnInfo
);
```

Parameters

See IColumnsInfo::GetColumnInfo in the *OLE DB Programmer's Reference*.

Return Value

A standard **HRESULT**.

Remarks

The user must free the returned column information and string buffer. Use the second version of this method when you use CDynamicAccessor and need to override the bindings.

For more information, see IColumnsInfo::GetColumnInfo in the *OLE DB Programmer's Reference*.

See Also
Reference
CAccessorRowset Class
Other Resources
CAccessorRowset Members

CArrayRowset Class

Accesses elements of a rowset using array syntax.

```
template < class TAccessor >
class CArrayRowset :
   public CVirtualBuffer <TAccessor>,
   protected CBulkRowset <TAccessor>
```

Parameters

TAccessor

The type of accessor class that you want the rowset to use.

Requirements

Header: atldbcli.h

See Also

Reference

CRowset Class

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

Other Resources

CArrayRowset Members

CArrayRowset Members

Methods

CArrayRowset	Constructor.
Snapshot	Reads the entire rowset into memory.

Operators

Operator[]	Accesses an element of the rowset.

Data Members

CArrayRowset::m_nRowsRead	The number of rows already read.

See Also

Reference

CArrayRowset Class

Other Resources

CRowset Members

CArrayRowset Methods

For information about the methods in **CArrayRowset**, see CArrayRowset Members.

CArrayRowset::CArrayRowset

Creates a new **CArrayRowset** object.

```
CArrayRowset(
   int nMax = 100000
);
```

Parameters

nMax

[in] Maximum number of rows in the rowset.

See Also

Reference

CArrayRowset Class

Other Resources

CArrayRowset Members

CArrayRowset::Snapshot

Reads the entire rowset into memory, creating an image or snapshot of it.

```
HRESULT Snapshot( ) throw( );
```

See Also

Reference

CArrayRowset Class

Other Resources

CArrayRowset Members

CArrayRowset Operators

For information about the operators in **CArrayRowset**, see CArrayRowset Members.

CArrayRowset::operator[]

Provides array-like syntax for accessing a row in the rowset.

```
TAccessor& operator[](
   int nRow
);
```

Parameters

TAccessor

A templated parameter that specifies the type of accessor stored in the rowset.

nRow

[in] Number of the row (array element) you want to access.

Return Value

The contents of the requested row.

Remarks

If *nRow* exceeds the number of rows in the rowset, an exception is thrown.

See Also

Reference

CArrayRowset Class

Other Resources

CArrayRowset Members

CArrayRowset Data Members

For information about the data members in **CArrayRowset**, see CArrayRowset Members.

CArrayRowset::m_nRowsRead

Contains the number of rows in the rowset that have already been read.

```
ULONG m_nRowsRead;
```

See Also

Reference

CArrayRowset Class

Other Resources

CArrayRowset Members

CBookmark Class

Holds a bookmark value in its buffer.

```
template < DBLENGTH nSize = 0 >
class CBookmark : public CBookmarkBase
template < >
class CBookmark< 0 > : public CBookmarkBase
```

Parameters

nSize

The size of the bookmark buffer in bytes. When *nSize* is zero, the bookmark buffer will be dynamically created at run time.

Remarks

CBookmark<0> is a template specialization of **CBookmark**; its buffer is dynamically created at run time.

Requirements

Header: atldbcli.h

See Also

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

Other Resources

CBookmark Members

CBookmark Members

Methods

CBookmark	The constructor
GetBuffer	Retrieves the pointer to the buffer.
GetSize	Retrieves the size of the buffer in bytes.
SetBookmark	Sets the bookmark value.

Operators

operator =	Assigns one **CBookmark** class to another.

See Also

Reference

CBookmark Class

CBookmark Methods

For information about the methods in **CBookmark**, see CBookmark Members.

CBookmark::CBookmark

The constructor.

```
CBookmark( );
CBookmark(
   DBLENGTH nSize
);
```

Parameters

nSize

[in] Size of the bookmark buffer in bytes.

Remarks

The first function sets the buffer to **NULL** and the buffer size to 0. The second function sets the buffer size to *nSize*, and the buffer to a byte array of *nSize* bytes.

Note
This function is only available in **CBookmark<0>**.

See Also

Reference

CBookmark Class

Other Resources

CBookmark Members

CBookmark::GetBuffer

Retrieves the pointer to the bookmark buffer.

```
virtual BYTE* GetBuffer( ) const throw( );
```

Return Value

A pointer to the bookmark buffer.

See Also

Reference

CBookmark Class

Other Resources

CBookmark Members

CBookmark::GetSize

Retrieves the size of the bookmark buffer.

```
virtual DBLENGTH GetSize( ) const throw( );
```

Return Value

The size of the buffer in bytes.

See Also

Reference

CBookmark Class

Other Resources

CBookmark Members

CBookmark::SetBookmark

Copies the bookmark value referenced by *pBuffer* to the **CBookmark** buffer and sets the buffer size to *nSize*.

```
HRESULT SetBookmark(
   DBLENGTH nSize,
   BYTE* pBuffer
) throw( );
```

Parameters

nSize

[in] The size of the bookmark buffer.

pBuffer

[in] A pointer to the byte array containing the bookmark value.

Return Value

A standard **HRESULT**.

Remarks

This function is only available in **CBookmark<0>**.

See Also

Reference

CBookmark Class

Other Resources

CBookmark Members

CBookmark Operators

For information about the operators in **CBookmark**, see CBookmark Members.

CBookmark::operator =

Assigns a **CBookmark** object to another.

```
CBookmark& operator =(
   const CBookmark& bookmark
) throw( );
```

Remarks

This operator is needed only in **CBookmark<0>**.

See Also

Reference

CBookmark Class

Other Resources

CBookmark Members

CBulkRowset Class

Fetches and manipulates rows to work on data in bulk by retrieving multiple row handles with a single call.

```
template <class TAccessor>
class CBulkRowset : public CRowset<TAccessor>
```

Parameters

TAccessor

An accessor class.

Example

The following example demonstrates use of the **CBulkRowset** class.

```
// sample.cpp
//

#include "stdafx.h"

class CTable1Data
{
   public:
   char m_szField1[50];

   BEGIN_COLUMN_MAP(CTable1Data)
     COLUMN_ENTRY(1, m_szField1)
   END_COLUMN_MAP()
};

int main(int argc, char* argv[])
{
   CoInitialize(NULL);

   CCommand<CAccessor<CTable1Data>, CBulkRowset > cmd;
   CDataSource ds;

   // Open up data link dialogs to create a data source
   ds.Open();

   CSession session;
   session.Open(ds);
   // Could call SetRows() here if you want to fetch
   // more than 10 HROWs at a time.
   cmd.Open(session, "Select * from table1");
   cmd.MoveFirst();
   // Note that the CBulkRowset by default fetched 10 HROWs at a time
   // so that the MoveNext call will not have to make the GetNextRows
   // call to get the second HROW because it has already been fetched
   //by the MoveFirst() call above.
   cmd.MoveNext();

   cmd.Close();
   session.Close();
   ds.Close();

   return 0;
}
```

Requirements

Header: atldbcli.h

See Also

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

Other Resources

CBulkRowset Members

CBulkRowset Members

Methods

AddRefRows	Increments the reference count.
CBulkRowset	Constructor.
MoveFirst	Retrieves the first row of data, performing a new bulk fetch if necessary.
MoveLast	Moves to the last row.
MoveNext	Retrieves the next row of data.
MovePrev	Moves to the previous row.
MoveToBookmark	Fetches the row marked by a bookmark or the row at a specified offset from that bookmark.
MoveToRatio	Fetches rows starting from a fractional position in the rowset.
ReleaseRows	Sets the current row (**m_nCurrentRow**) to zero and releases all rows.
SetRows	Sets the number of row handles to be retrieved by one call.

See Also

Reference

CBulkRowset Class

CBulkRowset Methods

For information about the methods in **CBulkRowset**, see CBulkRowset Members.

CBulkRowset::AddRefRows

Calls IRowset::AddRefRows to increment the reference count for all rows currently retrieved from the bulk rowset.

```
HRESULT AddRefRows( ) throw( );
```

Return Value

A standard **HRESULT**.

See Also

Reference

CBulkRowset Class

CBulkRowset::ReleaseRows

Other Resources

CBulkRowset Members

CBulkRowset::CBulkRowset

Creates a new **CBulkRowset** object and sets the default row count to 10.

```
CBulkRowset( );
```

See Also

Reference

CBulkRowset Class

Other Resources

CBulkRowset Members

CBulkRowset::MoveFirst

Retrieves the first row of data.

```
HRESULT MoveFirst( ) throw( );
```

Return Value

A standard **HRESULT**.

See Also

Reference

CBulkRowset Class

Other Resources

CBulkRowset Members

CBulkRowset::MoveLast

Moves to the last row.

```
HRESULT MoveLast( ) throw( );
```

Return Value

A standard **HRESULT**.

See Also

Reference

CBulkRowset Class

Other Resources

CBulkRowset Members

CBulkRowset::MoveNext

Retrieves the next row of data.

```
HRESULT MoveNext( ) throw( );
```

Return Value

A standard **HRESULT**. When the end of the rowset has been reached, returns **DB_S_ENDOFROWSET**.

See Also

Reference

CBulkRowset Class

Other Resources

CBulkRowset Members

CBulkRowset::MovePrev

Moves to the previous row.

```
HRESULT MovePrev( ) throw( );
```

Return Value

A standard **HRESULT**.

See Also

Reference

CBulkRowset Class

Other Resources

CBulkRowset Members

CBulkRowset::MoveToBookmark

Fetches the row marked by a bookmark or the row at a specified offset (*lSkip*) from that bookmark.

```
HRESULT MoveToBookmark(
   const CBookmarkBase& bookmark,
   DBCOUNTITEM lSkip = 0
) throw( );
```

Parameters

bookmark

[in] A bookmark marking the location from which you want to fetch data.

lSkip

[in] The number count of rows from the bookmark to the target row. If *lSkip* is zero, the first row fetched is the bookmarked row. If *lSkip* is 1, the first row fetched is the row after the bookmarked row. If *lSkip* is –1, the first row fetched is the row before the bookmarked row.

Return Value

See IRowset::GetData in the *OLE DB Programmer's Reference*.

See Also

Reference

CBulkRowset Class

Other Resources

CBulkRowset Members

CBulkRowset::MoveToRatio

Fetches rows starting from a fractional position in the rowset.

```
HRESULT MoveToRatio(
   DBCOUNTITEM nNumerator,
   DBCOUNTITEM nDenominator
) throw( );
```

Parameters

nNumerator

[in] The numerator used to determine the fractional position from which to fetch data.

nDenominator

[in] The denominator used to determine the fractional position from which to fetch data.

Return Value

A standard **HRESULT**.

Remarks

MoveToRatio fetches the rows roughly according to the following formula:

```
( nNumerator *  RowsetSize ) / nDenominator
```

Where `RowsetSize` is the size of the rowset, measured in rows. The accuracy of this formula depends on the specific provider. For details, see IRowsetScroll::GetRowsAtRatio in the *OLE DB Programmer's Reference*.

See Also

Reference

CBulkRowset Class

Other Resources

CBulkRowset Members

CBulkRowset::ReleaseRows

Calls IRowset::ReleaseRows to decrement the reference count for all rows currently retrieved from the bulk rowset.

```
HRESULT ReleaseRows( ) throw( );
```

Return Value

A standard **HRESULT**.

See Also

Reference

CBulkRowset Class

CBulkRowset::AddRefRows

Other Resources

CBulkRowset Members

CBulkRowset::SetRows

Sets the number of row handles retrieved by each call.

```
void SetRows(
   DBROWCOUNT nRows
) throw( );
```

Parameters

nRows

[in] The new size of the rowset (number of rows).

Remarks

If you call this function, it must be before the rowset is opened.

See Also

Reference

CBulkRowset Class

Other Resources

CBulkRowset Members

CColumnAccessor Class

Generates injected consumer code.

```
class CColumnAccessor : public CAccessorBase
```

Remarks

In the injected code, every column is bound as a separate accessor. You should be aware that this class is used in the injected code (for example, you might encounter it when debugging), but you typically never have to use it or its methods directly.

CColumnAccessor implements the following stub methods, each of which correspond in functionality to other accessor class methods:

- **CColumnAccessor** The constructor; instantiates and initializes the **CColumnAccessor** object.
- **CreateAccessor** Allocates memory for the column binding structures and initializes the column data members.
- **BindColumns** Binds columns to accessors.
- **SetParameterBuffer** Allocates buffers for parameters.
- **AddParameter** Adds a parameter entry to the parameter entry structures.
- **HasOutputColumns** Determines whether the accessor has output columns
- **HasParameters** Determines whether the accessor has parameters.
- **BindParameters** Binds the created parameters to columns.

Requirements

Header: atldbcli.h

See Also

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

CCommand Class

Provides methods to set and execute a command.

```
template <
   class TAccessor = CNoAccessor,
   template < typename T > class TRowset = CRowset,
   class TMultiple = CNoMultipleResults
>
class CCommand :
   public CAccessorRowset <
            TAccessor,
            TRowset
   >,
   public CCommandBase,
   public TMultiple
```

Parameters

TAccessor

The type of accessor class (such as **CDynamicParameterAccessor**, **CDynamicStringAccessor**, or **CEnumeratorAccessor**) that you want the command to use. The default is **CNoAccessor**, which specifies that the class not support parameters or output columns.

TRowset

The type of rowset class (such as **CArrayRowset** or **CNoRowset**) that you want the command to use. The default is **CRowset**.

TMultiple

To use an OLE DB command that can return multiple results, specify CMultipleResults. Otherwise, use CNoMultipleResults. For details, see IMultipleResults.

Remarks

Use this class when you need to perform a parameter-based operation or execute a command. If you merely need to open a simple rowset, use CTable instead.

The accessor class you are using determines the method of binding parameters and data.

Note that you cannot use stored procedures with the OLE DB Provider for Jet because that provider does not support stored procedures (only constants are allowed in query strings).

Requirements

Header: atldbcli.h

Example

See the ShowImage Sample.

See Also

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

Other Resources

CCommand Members

CCommand Members

Methods

Close	Closes the current command.
GetNextResult	Fetches the next result when using multiple result sets.
Open	Executes and optionally binds the command.

Inherited Methods

Create	Creates a new command for the specified session, then sets the command text.
CreateCommand	Creates a new command.
GetParameterInfo	Gets a list of the command's parameters, their names, and their types.
Prepare	Validates and optimizes the current command.
ReleaseCommand	Releases the parameter accessor if necessary, then releases the command.
SetParameterInfo	Specifies the native type of each command parameter.
Unprepare	Discards the current command execution plan.

See Also

Reference

CCommand Class

CCommand Methods

For information about the methods in **CCommand**, see CCommand Members.

CCommand::Close

Releases the accessor rowset associated with the command.

```
void Close( );
```

Remarks

A command uses a rowset, result set accessor, and (optionally) a parameter accessor (unlike tables, which do not support parameters and do not need a parameter accessor).

When you execute a command, you should call both **Close** and ReleaseCommand after the command.

When you want to execute the same command repeatedly, you should release each result set accessor by calling **Close** before calling **Execute**. At the end of the series, you should release the parameter accessor by calling **ReleaseCommand**. Another common scenario is calling a stored procedure that has output parameters. On many providers (such as the OLE DB provider for SQL Server) the output parameter values will not be accessible until you close the result set accessor. Call **Close** to close the returned rowset and result set accessor, but not the parameter accessor, thus allowing you to retrieve the output parameter values.

Example

The following example shows how you can call **Close** and **ReleaseCommand** when you execute the same command repeatedly.

```
int main(int argc, char* argv[])
{
    HRESULT hr;

    hr = CoInitialize(NULL);

    CCustOrdersDetail rs;      // Your CCommand-derived class
    rs.m_OrderID = 10248;      // Open order 10248
    hr = rs.OpenAll();         // (Open also executes the command)
    hr = rs.MoveFirst();       // Move to the first row and print it

    printf_s( "Name: %s, Unit Price: %d, Quantity: %d, Discount %d, "
              "Extended Price %d",
              rs.m_ProductName, rs.m_UnitPrice.int64, rs.m_Quantity,
              rs.m_Discount, rs.m_ExtendedPrice.int64 );

    // Close the first command execution
    rs.Close();


    rs.m_OrderID = 10249;      // Open order 10249 (a new order)
    hr = rs.Open();            // (Open also executes the command)
    hr = rs.MoveFirst();       // Move to the first row and print it

    printf_s( "Name: %s, Unit Price: %d, Quantity: %d, Discount %d, "
              "Extended Price %d",
              rs.m_ProductName, rs.m_UnitPrice.int64,
              rs.m_Quantity, rs.m_Discount, rs.m_ExtendedPrice.int64 );

    // Close the second command execution;
    // Instead of the two following lines
    // you could simply call rs.CloseAll()
    // (a wizard-generated method):
    rs.Close();
    rs.ReleaseCommand();

    CoUninitialize();
    return 0;
}
```

See Also

Reference

CCommand Class

CCommand::ReleaseCommand

Other Resources

CCommand Members

CCommand::Create

Calls CCommand::CreateCommand to create a new command for the specified session, then calls ICommandText::SetCommandText to specify the command text.

```
HRESULT CCommandBase::Create(
   const CSession& session,
   LPCWSTR wszCommand,
   REFGUID guidCommand = DBGUID_DEFAULT
) throw ( );
HRESULT CCommandBase::Create(
   const CSession& session,
   LPCSTR szCommand,
   REFGUID guidCommand = DBGUID_DEFAULT
) throw ( );
```

Parameters

session

[in] A session on which to create the command.

wszCommand

[in] A pointer to the Unicode text of the command string.

szCommand

[in] A pointer to the ANSI text of the command string.

guidCommand

[in] A GUID that specifies the syntax and general rules for the provider to use in parsing the command text. For a complete description of dialects, see ICommandText::GetCommandText in the *OLE DB Programmer's Reference*.

Return Value

A standard **HRESULT**.

Remarks

The first form of **Create** takes a Unicode command string. The second form of **Create** takes an ANSI command string (provided for backward compatibility with existing ANSI applications).

See Also

Reference

CCommand Class

Other Resources

CCommand Members

CCommand::CreateCommand

Creates a new command.

```
HRESULT CCommandBase::CreateCommand(
   const CSession& session
) throw ( );
```

Parameters

session

[in] A **CSession** object to be associated with the new command.

Return Value

A standard **HRESULT**.

Remarks

This method creates a command using the specified session object.

See Also

Reference

CCommand Class

Other Resources

CCommand Members

CCommand::GetNextResult

Fetches the next result set if one is available.

```
HRESULT GetNextResult(
   DBROWCOUNT* pulRowsAffected,
   bool bBind = true
) throw( );
```

Parameters

pulRowsAffected

[in/out] A pointer to memory where the count of rows affected by a command is returned.

bBind

[in] Specifies whether to bind the command automatically after being executed. The default is **true**, which causes the command to be bound automatically. Setting *bBind* to **false** prevents the automatic binding of the command so that you can bind manually. (Manual binding is of particular interest to OLAP users.)

Return Value

A standard **HRESULT**.

Remarks

If a result set has been previously fetched, this function releases the previous result set and unbinds the columns. If *bBind* is **true**, it binds the new columns.

You should call this function only if you have specified multiple results by setting the **CCommand** template parameter *TMultiple*=**CMultipleResults**.

See Also

Reference

CCommand Class

Other Resources

CCommand Members

CCommand::GetParameterInfo

Gets a list of the command's parameters, their names, and their types.

```
HRESULT CCommandBase::GetParameterInfo(
   DB_UPARAMS* pParams,
   DBPARAMINFO** ppParamInfo,
   OLECHAR** ppNamesBuffer
) throw ( );
```

Parameters

See ICommandWithParameters::GetParameterInfo in the *OLE DB Programmer's Reference*.

Return Value

A standard **HRESULT**.

See Also

Reference

CCommand Class

Other Resources

CCommand Members

CCommand::Open

Executes and optionally binds the command.

```
HRESULT Open(
   const CSession& session,
   LPCWSTR wszCommand,
   DBPROPSET *pPropSet = NULL,
   DBROWCOUNT* pRowsAffected = NULL,
   REFGUID guidCommand = DBGUID_DEFAULT,
   bool bBind = true,
   ULONG ulPropSets = 0
) throw( );
HRESULT Open(
   const CSession& session,
   LPCSTR szCommand,
   DBPROPSET *pPropSet = NULL,
   DBROWCOUNT* pRowsAffected = NULL,
   REFGUID guidCommand = DBGUID_DEFAULT,
   bool bBind = true,
   ULONG ulPropSets = 0
) throw( );
HRESULT Open(
   const CSession& session,
   INT szCommand = NULL,
   DBPROPSET *pPropSet = NULL,
   DBROWCOUNT* pRowsAffected = NULL,
   REFGUID guidCommand = DBGUID_DEFAULT,
   bool bBind = true,
   ULONG ulPropSets = 0
) throw( );
HRESULT Open(
   DBPROPSET *pPropSet = NULL,
   DBROWCOUNT* pRowsAffected = NULL,
   bool bBind = true,
   ULONG ulPropSets = 0
) throw( );
```

Parameters

session

[in] The session in which to execute the command.

wszCommand

[in] The command to execute, passed as a Unicode string. Can be **NULL** when using **CAccessor**, in which case the command will be retrieved from the value passed to the DEFINE_COMMAND macro. See ICommand::Execute in the *OLE DB Programmer's Reference* for details.

szCommand

[in] Same as *wszCommand* except that this parameter takes an ANSI command string. The fourth form of this method can take a NULL value. See "Remarks" later in this topic for details.

pPropSet

[in] A pointer to an array of DBPROPSET structures containing properties and values to be set. See Property Sets and Property Groups in the *OLE DB Programmer's Reference* in the Platform SDK.

pRowsAffected

[in/out] A pointer to memory where the count of rows affected by a command is returned. If **pRowsAffected* is **NULL**, no row count is returned. Otherwise, **Open** sets **pRowsAffected* according to the following conditions:

If	Then

The **cParamSets** element of *pParams* is greater than 1	**pRowsAffected* represents the total number of rows affected by all of the parameter sets specified in the execution.
The number of affected rows is not available	**pRowsAffected* is set to –1.
The command does not update, delete, or insert rows	**pRowsAffected* is undefined.

guidCommand

[in] A GUID that specifies the syntax and general rules for the provider to use in parsing the command text. See ICommandText::GetCommandText and ICommandText::SetCommandText in the *OLE DB Programmer's Reference* for details.

bBind

[in] Specifies whether to bind the command automatically after being executed. The default is **true**, which causes the command to be bound automatically. Setting *bBind* to **false** prevents the automatic binding of the command so that you can bind manually. (Manual binding is of particular interest to OLAP users.)

ulPropSets

[in] The number of DBPROPSET structures passed in the *pPropSet* argument.

Return Value

A standard **HRESULT**.

Remarks

The first three forms of **Open** take a session, create a command, and execute the command, binding any parameters as necessary.

The first form of **Open** takes a Unicode command string and has no default value.

The second form of **Open** takes an ANSI command string and no default value (provided for backward compatibility with existing ANSI applications).

The third form of **Open** allows the command string to be NULL, because of type **int** with a default value of NULL. It is provided for calling `Open(session, NULL);` or `Open(session);` because NULL is of type **int**. This version requires and asserts that the **int** parameter be NULL.

Use the fourth form of **Open** when you have already created a command and you want to perform a single Prepare and multiple executions.

Note
Open calls **Execute**, which in turn calls **GetNextResult**.

Example

See the ShowImage Sample.

See Also

Reference

CCommand Class

Other Resources

CCommand Members

CCommand::Prepare

Validates and optimizes the current command.

```
HRESULT CCommandBase::Prepare(
   ULONG cExpectedRuns = 0
) throw( );
```

Parameters

cExpectedRuns

[in] The number of times you expect to execute the command.

Return Value

A standard **HRESULT**.

Remarks

This method wraps the OLE DB method ICommandPrepare::Prepare.

See Also

Reference

CCommand Class

Other Resources

CCommand Members

CCommand::ReleaseCommand

Releases the parameter accessor, then releases the command itself.

```
void CCommandBase::ReleaseCommand( ) throw( );
```

Remarks

ReleaseCommand is used in conjunction with **Close**. See Close for usage details.

See Also

Reference

CCommand Class

CCommand::Close

Other Resources

CCommand Members

CCommand::SetParameterInfo

Specifies the native type of each command parameter.

```
HRESULT CCommandBase::SetParameterInfo(
   DB_UPARAMS ulParams,
   const DBORDINAL* pOrdinals,
   const DBPARAMBINDINFO* pParamInfo
) throw( );
```

Parameters

See ICommandWithParameters::SetParameterInfo in the *OLE DB Programmer's Reference*.

Return Value

A standard **HRESULT**.

See Also

Reference

CCommand Class

Other Resources

CCommand Members

CCommand::Unprepare

Discards the current command execution plan.

```
HRESULT CCommandBase::Unprepare( ) throw( );
```

Return Value

A standard **HRESULT**.

Remarks

This method wraps the OLE DB method ICommandPrepare::Unprepare.

See Also

Reference

CCommand Class

Other Resources

CCommand Members

CDataConnection Class

Manages the connection with the data source.

```
class CDataConnection
```

Remarks

CDataConnection is a useful class for creating clients because it encapsulates necessary objects (data source and session) and some of the work you need to do when connecting to a data source. For example, in a typical ATL Server application, an ISAPI object caches **CDataConnection** objects and exposes them to request handlers.

Without **CDataConnection**, you have to create a **CDataSource** object, call its OpenFromInitializationString method, then create an instance of a CSession object, call its Open method, then create a CCommand object and call its **Open*** methods.

With **CDataConnection**, you only need to create a connection object, pass it an initialization string, then use that connection to open commands. If you plan on using your connection to the database repeatedly, it is a good idea to keep the connection open, and **CDataConnection** provides a convenient way to do that. Also, you will need a **CDataConnection** object when you call certain ATL Server methods such as CDataSourceCache::Add and CDataSourceCache::Lookup.

Note
If you are creating a database application that needs to handle multiple sessions, you will need to use OpenNewSession.

Requirements

Header: atldbcli.h

Example

See the MantaWeb Sample.

See Also

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

Other Resources

CDataConnection Members

CDataConnection Members

Methods

CDataConnection	Constructor. Instantiates and initializes a **CDataConnection** object.
Copy	Creates a copy of an existing data connection.
Open	Opens a connection to a data source using an initialization string.
OpenNewSession	Opens a new session on the current connection.

Operators

operator BOOL	Determines whether the current session is open or not.
operator bool	Determines whether the current session is open or not.
operator CDataSource&	Returns a reference to the contained **CDataSource** object.
operator CDataSource*	Returns a pointer to the contained **CDataSource** object.
operator CSession&	Returns a reference to the contained **CSession** object.
operator CSession*	Returns a pointer to the contained **CSession** object.

See Also

Reference

CDataConnection Class

CDataConnection Methods

For information about the methods in **CDataConnection**, see CDataConnection Members.

CDataConnection::CDataConnection

Instantiates and initializes a **CDataConnection** object.

```
CDataConnection();
CDataConnection(
   const CDataConnection &ds
);
```

Parameters

ds

[in] A reference to an existing data connection.

Remarks

The first override creates a new **CDataConnection** object with default settings.

The second override creates a new **CDataConnection** object with settings equivalent to the data connection object you specify.

See Also

Reference

CDataConnection Class

Other Resources

CDataConnection Members

CDataConnection::Copy

Creates a copy of an existing data connection.

```
CDataConnection& Copy(
   const CDataConnection & ds
) throw( );
```

Parameters

ds

[in] A reference to an existing data connection to copy.

See Also

Reference

CDataConnection Class

Other Resources

CDataConnection Members

CDataConnection::Open

Opens a connection to a data source using an initialization string.

```
HRESULT Open(
   LPCOLESTR szInitString
) throw( );
```

Parameters

szInitString

[in] The initialization string for the data source.

Return Value

A standard **HRESULT**.

See Also

Reference

CDataConnection Class

Other Resources

CDataConnection Members

CDataConnection::OpenNewSession

Opens a new session using the current connection object's data source.

```
HRESULT OpenNewSession(
   CSession & session
) throw( );
```

Parameters

session

[in/out] A reference to the new session object.

Remarks

The new session uses the current connection object's contained data source object as its parent, and can access all of the same information as the data source.

Return Value

A standard **HRESULT**.

See Also

Reference

CDataConnection Class

Other Resources

CDataConnection Members

CDataConnection Operators

For information about the operators in **CDataConnection**, see CDataConnection Members.

CDataConnection::operator BOOL

Determines whether the current session is open or not.

```
operator BOOL( ) throw( );
```

Remarks

Returns **BOOL** (MFC typedef) value. **TRUE** means the current session is open; **FALSE** means the current session is closed.

See Also

Reference

CDataConnection Class

CDataConnection::operator bool (OLE DB)

Other Resources

CDataConnection Members

CDataConnection::operator bool (OLE DB)

Determines whether the current session is open or not.

```
operator bool( ) throw( );
```

Remarks

Returns a **bool** (C++ data type) value. **true** means the current session is open; **false** means the current session is closed.

See Also

Reference

CDataConnection Class

CDataConnection::operator BOOL

Other Resources

CDataConnection Members

CDataConnection::operator CDataSource&

Returns a reference to the contained **CDataSource** object.

```
operator const CDataSource&() throw( );
```

Remarks

This operator returns a reference to the contained **CDataSource** object, allowing you to pass a **CDataConnection** object where a **CDataSource** reference is expected.

Example

If you have a function (such as `func` below) that takes a **CDataSource** reference, you can use **CDataSource&** to pass a **CDataConnection** object instead.

```
int func(CDataSource& theSource)
{
...
return 0;
}

int main(void)
{
CDataConnection dc;
dc.Open();
func(dc);
}
```

See Also

Reference

CDataConnection Class

CDataConnection::operator CDataSource*

Other Resources

CDataConnection Members

CDataConnection::operator CDataSource*

Returns a pointer to the contained **CDataSource** object.

```
operator const CDataSource*() throw( );
```

Remarks

This operator returns a pointer to the contained **CDataSource** object, allowing you to pass a **CDataConnection** object where a **CDataSource** pointer is expected.

See operator CDataSource& for a usage example.

See Also

Reference

CDataConnection Class

CDataConnection::operator CDataSource&

Other Resources

CDataConnection Members

CDataConnection::operator CSession&

Returns a reference to the contained **CSession** object.

```
operator const CSession&();
```

Remarks

This operator returns a reference to the contained **CSession** object, allowing you to pass a **CDataConnection** object where a **CSession** reference is expected.

Example

If you have a function (such as `func` below) that takes a **CSession** reference, you can use **CSession&** to pass a **CDataConnection** object instead.

```
int func(CSession& theSession)
{
...
return 0;
}

int main(void)
{
CDataConnection dc;
dc.Open();
func(dc);
}
```

See Also

Reference

CDataConnection Class

CDataConnection::operator CSession*

Other Resources

CDataConnection Members

CDataConnection::operator CSession*

Returns a pointer to the contained **CSession** object.

```
operator const CSession*() throw( );
```

Remarks

This operator returns a pointer to the contained **CSession** object, allowing you to pass a **CDataConnection** object where a **CSession** pointer is expected.

Example

See operator CSession& for a usage example.

See Also

Reference

CDataConnection Class

CDataConnection::operator CSession&

Other Resources

CDataConnection Members

CDataSource Class

Corresponds to an OLE DB data source object, which represents a connection through a provider to a data source.

```
class CDataSource
```

Remarks

One or more database sessions can be created for a single connection. These sessions are represented by **CSession**. You must call CDataSource::Open to open the connection before creating a session with **CSession::Open**.

For an example of how to use **CDataSource**, see the CatDB sample.

Requirements

Header: atldbcli.h

Example

See the ConfirmUser Sample and the ShowImage Sample.

See Also

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

Other Resources

CDataSource Members

CDataSource Members

Methods

Close	Closes the connection.
GetInitializationString	Retrieves the initialization string of the data source that is currently open.
GetProperties	Gets the values of properties currently set for the connected data source.
GetProperty	Gets the value of a single property currently set for the connected data source.
Open	Creates a connection to a provider (data source) using either a **CLSID**, **ProgID**, or a **CEnumerator** moniker provided by the caller.
OpenFromFileName	Opens a data source from a file specified by the user-supplied file name.
OpenFromInitializationString	Opens the data source specified by an initialization string.
OpenWithPromptFileName	Allows the user to select a previously created data link file to open the corresponding data source.
OpenWithServiceComponents	Opens a data source object using the Data Link dialog box.

See Also

Reference

CDataSource Class

CDataSource Methods

For information about the methods in **CDataSource**, see CDataSource Members.

CDataSource::Close

Closes the connection by releasing the **m_spInit** pointer.

```
void Close( ) throw( );
```

See Also

Reference

CDataSource Class

Other Resources

CDataSource Members

CDataSource::GetInitializationString

Retrieves the initialization string of a data source that is currently open.

```
HRESULT GetInitializationString(
   BSTR* pInitializationString,
   bool bIncludePassword = false
) throw( );
```

Parameters

pInitializationString

[out] A pointer to the initialization string.

bIncludePassword

[in] **true** if string includes a password; otherwise **false**.

Return Value

A standard **HRESULT**.

Remarks

The resulting initialization string can be used to later reopen this data source connection.

See Also

Reference

CDataSource Class

Other Resources

CDataSource Members

CDataSource::GetProperties

Returns the property information requested for the connected data source object.

```
HRESULT GetProperties(
   ULONG ulPropIDSets,
   const DBPROPIDSET* pPropIDSet,
   ULONG* pulPropertySets,
   DBPROPSET** ppPropsets
) const throw( );
```

Parameters

See IDBProperties::GetProperties in the *OLE DB Programmer's Reference* in the *Platform SDK*.

Return Value

A standard **HRESULT**.

Remarks

To get a single property, use GetProperty.

See Also

Reference

CDataSource Class

Other Resources

CDataSource Members

CDataSource::GetProperty

Returns the value of a specified property for the connected data source object.

```
HRESULT GetProperty(
   const GUID& guid,
   DBPROPID propid,
   VARIANT* pVariant
) const throw( );
```

Parameters

guid

[in] A GUID identifying the property set for which to return the property.

propid

[in] Property ID for the property to return.

pVariant

[out] A pointer to the variant where **GetProperty** returns the value of the property.

Return Value

A standard **HRESULT**.

Remarks

To get multiple properties, use GetProperties.

See Also

Reference

CDataSource Class

Other Resources

CDataSource Members

CDataSource::Open

Opens a connection to a data source using a **CLSID**, **ProgID**, or **CEnumerator** moniker or prompts the user with a locator dialog box.

```
HRESULT Open(
   const CLSID& clsid,
   DBPROPSET* pPropSet = NULL,
   ULONG nPropertySets = 1
) throw( );
HRESULT Open(
   const CLSID& clsid,
   LPCTSTR pName,
   LPCTSTR pUserName = NULL,
   LPCTSTR pPassword = NULL,
   long nInitMode = 0
) throw( );
HRESULT Open(
   LPCTSTR szProgID,
   DBPROPSET* pPropSet = NULL,
   ULONG nPropertySets = 1
) throw( );
HRESULT Open(
   LPCTSTR szProgID,
   LPCTSTR pName,
   LPCTSTR pUserName = NULL,
   LPCTSTR pPassword = NULL,
   long nInitMode = 0
) throw( );
HRESULT Open(
   const CEnumerator& enumerator,
   DBPROPSET* pPropSet = NULL,
   ULONG nPropertySets = 1
) throw( );
HRESULT Open(
   const CEnumerator& enumerator,
   LPCTSTR pName,
   LPCTSTR pUserName = NULL,
   LPCTSTR pPassword = NULL,
   long nInitMode = 0
) throw( );
HRESULT Open(
   HWND hWnd = GetActiveWindow( ),
   DBPROMPTOPTIONS dwPromptOptions = DBPROMPTOPTIONS_WIZARDSHEET
) throw( );
HRESULT Open(
   LPCWSTR szProgID,
   DBPROPSET* pPropSet = NULL,
   ULONG nPropertySets = 1
) throw( );
HRESULT Open(
   LPCSTR szProgID,
   LPCTSTR pName,
   LPCTSTR pUserName = NULL,
   LPCTSTR pPassword = NULL,
   long nInitMode = 0
) throw( );
```

Parameters

clsid

[in] The **CLSID** of the data provider.

pPropSet

[in] A pointer to an array of DBPROPSET structures containing properties and values to be set. See Property Sets and Property Groups in the *OLE DB Programmer's Reference* in the Platform SDK.

nPropertySets

[in] The number of DBPROPSET structures passed in the *pPropSet* argument.

pName

[in] The name of the database to which to connect.

pUserName

[in] The name of the user.

pPassword

[in] The user's password.

nInitMode

[in] Database initialization mode. See Initialization Propertiesin the *OLE DB Programmer's Reference* in the *Platform SDK* for a list of valid initialization modes. If *nInitMode* is zero, no initialization mode is included in the property set used to open the connection.

szProgID

[in] A program identifier.

enumerator

[in] A CEnumerator object used to obtain a moniker for opening the connection when the caller does not specify a **CLSID**.

hWnd

[in] Handle to the window that is to be the parent of the dialog box. Using the function overload that uses the *hWnd* parameter will automatically invoke Service Components; see Remarks for details.

dwPromptOptions

[in] Determines the style of locator dialog box to display. See Msdasc.h for possible values.

Return Value

A standard **HRESULT**.

Remarks

The method overload that uses the *hWnd* parameter opens a data source object with the service components in oledb32.dll; this DLL contains the implementation of Service Components features such as Resource Pooling, Automatic Transaction Enlistment, and so on. For more information, see "OLE DB Services" in the OLE DB Programmer's Reference at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbole_db_services.asp?frame=true.

The method overloads that do not use the *hWnd* parameter open a data source object without using the service components in oledb32.dll. A CDataSource object opened with these function overloads will be unable to utilize any of the functionality of Service Components.

Example

The following code shows how to open a Jet 4.0 data source with OLE DB Templates. You treat the Jet data source as an OLE DB data source. However, your call to **Open** needs two property sets: one for **DBPROPSET_DBINIT** and the other for **DBPROPSET_JETOLEDB_DBINIT**, so that you can set **DBPROP_JETOLEDB_DATABASEPASSWORD**.

```
CDBPropSet rgDBPropSet[2];  // Declare two property sets;
                            // set properties elsewhere as appropriate.
...
hr = m_DataSource.Open( clsid, rgDBPropSet, 2 );
```

See Also

Reference

CDataSource Class

Other Resources

CDataSource::OpenFromFileName

Opens a data source from a file specified by the user-supplied file name.

```
HRESULT OpenFromFileName(
   LPCOLESTR szFileName
) throw( );
```

Parameters

szFileName

[in] The name of a file, usually a data source connection (.UDL) file.

For more information about data link files (.udl files), see Data Link API Overview in the *Platform SDK*.

Return Value

A standard **HRESULT**.

Remarks

This method opens a data source object using the service components in oledb32.dll; this DLL contains the implementation of Service Components features such as Resource Pooling, Automatic Transaction Enlistment, and so on. For more information, see "OLE DB Services" in the OLE DB Programmer's Reference at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbole_db_services.asp?frame=true.

Example

See the ConfirmUser Sample and the ShowImage Sample.

See Also

Reference

CDataSource Class

Other Resources

CDataSource Members

CDataSource::OpenFromInitializationString

Opens a data source specified by the user-supplied initialization string.

```
HRESULT OpenFromInitializationString(
   LPCOLESTR szInitializationString,
   bool fPromptForInfo = false
) throw( );
```

Parameters

szInitializationString

[in] The initialization string.

fPromptForInfo

[in] If this argument is set to **true**, then **OpenFromInitializationString** will set the **DBPROP_INIT_PROMPT** property to **DBPROMPT_COMPLETEREQUIRED**, which specifies that the user be prompted only if more information is needed. This is useful for situations in which the initialization string specifies a database that requires a password, but the string does not contain the password. The user will be prompted for a password (or any other missing information) when trying to connect to the database.

The default value is **false**, which specifies that the user never be prompted (sets **DBPROP_INIT_PROMPT** to **DBPROMPT_NOPROMPT**).

Return Value

A standard **HRESULT**.

Remarks

This method opens a data source object using the service components in oledb32.dll; this DLL contains the implementation of Service Components features such as Resource Pooling, Automatic Transaction Enlistment, and so on. For more information, see "OLE DB Services" in the OLE DB Programmer's Reference at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbole_db_services.asp?frame=true.

See Also

Reference

CDataSource Class

Other Resources

CDataSource Members

CDataSource::OpenWithPromptFileName

This method prompts the user with a dialog box, then opens a data source using the file specified by the user.

```
HRESULT OpenWithPromptFileName(
   HWND hWnd = GetActiveWindow(
   ),
   DBPROMPTOPTIONS dwPromptOptions = DBPROMPTOPTIONS_NONE,
   LPCOLESTR szInitialDirectory = NULL
) throw( );
```

Parameters

hWnd

[in] Handle to the window that is to be the parent of the dialog box.

dwPromptOptions

[in] Determines the style of locator dialog box to display. See Msdasc.h for possible values.

szInitialDirectory

[in] The initial directory to display in the locator dialog box.

Return Value

A standard **HRESULT**.

Remarks

This method opens a data source object using the service components in oledb32.dll; this DLL contains the implementation of Service Components features such as Resource Pooling, Automatic Transaction Enlistment, and so on. For more information, see "OLE DB Services" in the OLE DB Programmer's Reference at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbole_db_services.asp?frame=true.

See Also

Reference

CDataSource Class

Other Resources

CDataSource Members

CDataSource::OpenWithServiceComponents

Opens a data source object using the service components in oledb32.dll.

```
HRESULT OpenWithServiceComponents (
   const CLSID clsid,
   DBPROPSET* pPropset = NULL,
   ULONG ulPropSets = 1
);
HRESULT OpenWithServiceComponents (
   LPCSTR szProgID,
   DBPROPSET* pPropset = NULL,
   ULONG ulPropSets = 1
);
```

Parameters

clsid

[in] The **CLSID** of a data provider.

szProgID

[in] Program ID of a data provider.

pPropset

[in] A pointer to an array of DBPROPSET structures containing properties and values to be set. See Property Sets and Property Groups in the *OLE DB Programmer's Reference* in the Platform SDK. If the data source object is initialized, the properties must belong to the Data Source property group. If the same property is specified more than once in *pPropset*, then which value is used is provider-specific. If *ulPropSets* is zero, this parameter is ignored.

ulPropSets

[in] The number of DBPROPSET structures passed in the *pPropSet* argument. If this is zero, the provider ignores *pPropset*.

Return Value

A standard **HRESULT**.

Remarks

This method opens a data source object using the service components in oledb32.dll; this DLL contains the implementation of Service Components features such as Resource Pooling, Automatic Transaction Enlistment, and so on. For more information, see "OLE DB Services" in the OLE DB Programmer's Reference at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbole_db_services.asp?frame=true.

See Also

Reference

CDataSource Class

Other Resources

CDataSource Members

CDBErrorInfo Class

Provides support for OLE DB error processing using the OLE DB IErrorRecords interface.

```
class CDBErrorInfo
```

Remarks

This interface returns one or more error records to the user. Call CDBErrorInfo::GetErrorRecords first, to get a count of error records. Then call one of the access functions, such as CDBErrorInfo::GetAllErrorInfo, to retrieve error information for each record.

Requirements

Header: atldbcli.h

See Also

Tasks

DBViewer Sample: Database Browser

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

Other Resources

CDBErrorInfo Members

CDBErrorInfo Members

Methods

GetAllErrorInfo	Returns all error information contained in an error record.
GetBasicErrorInfo	Calls IErrorRecords::GetBasicErrorInfo to return basic information about the specified error.
GetCustomErrorObject	Calls IErrorRecords::GetCustomErrorObject to return a pointer to an interface on a custom error object.
GetErrorInfo	Calls IErrorRecords::GetErrorInfo to return an **IErrorInfo** interface pointer to the specified record.
GetErrorParameters	Calls IErrorRecords::GetErrorParameters to return the error parameters.
GetErrorRecords	Gets error records for the specified object.

See Also

Reference

CDBErrorInfo Class

CDBErrorInfo Methods

For information about the methods in **CDBErrorInfo**, see CDBErrorInfo Members.

CDBErrorInfo::GetAllErrorInfo

Returns all the types of error information contained in an error record.

```
HRESULT GetAllErrorInfo(
   ULONG ulRecordNum,
   LCID lcid,
   BSTR* pbstrDescription,
   BSTR* pbstrSource = NULL,
   GUID* pguid = NULL,
   DWORD* pdwHelpContext = NULL,
   BSTR* pbstrHelpFile = NULL
) const throw( );
```

Parameters

ulRecordNum

[in] The zero-based number of the record for which to return error information.

lcid

[in] The locale ID for the error information to be returned.

pbstrDescription

[out] A pointer to a text description of the error.

pbstrSource

[out] A pointer to a string containing the name of the component that generated the error.

pguid

[out] A pointer to the GUID of the interface that defined the error.

pdwHelpContext

[out] A pointer to the help context ID for the error.

pbstrHelpFile

[out] A pointer to a string containing the path to the help file that describes the error.

Return Value

S_OK if successful. See IErrorRecords::GetErrorInfo in the *OLE DB Programmer's Reference* for other return values.

See Also

Reference

CDBErrorInfo Class

Other Resources

CDBErrorInfo Members

CDBErrorInfo::GetBasicErrorInfo

Calls IErrorRecords::GetBasicErrorInfo to return basic information about the error, such as the return code and provider-specific error number.

```
HRESULT GetBasicErrorInfo(
   ULONG ulRecordNum,
   ERRORINFO* pErrorInfo
) const throw( );
```

Parameters

See IErrorRecords::GetBasicErrorInfo in the *OLE DB Programmer's Reference*.

Return Value

A standard **HRESULT**.

See Also

Reference

CDBErrorInfo Class

Other Resources

CDBErrorInfo Members

CDBErrorInfo::GetCustomErrorObject

Calls IErrorRecords::GetCustomErrorObject to return a pointer to an interface on a custom error object.

```
HRESULT GetCustomErrorObject(
   ULONG ulRecordNum,
   REFIID riid,
   IUnknown** ppObject
) const throw( );
```

Parameters

See IErrorRecords::GetCustomErrorObject in the *OLE DB Programmer's Reference*.

Return Value

A standard **HRESULT**.

See Also

Reference

CDBErrorInfo Class

Other Resources

CDBErrorInfo Members

CDBErrorInfo::GetErrorInfo

Calls IErrorRecords::GetErrorInfo to return an IErrorInfo interface pointer to the specified record.

```
HRESULT GetErrorInfo(
   ULONG ulRecordNum,
   LCID lcid,
   IErrorInfo** ppErrorInfo
) const throw( );
```

Parameters

See IErrorRecords::GetErrorInfo in the *OLE DB Programmer's Reference*.

Return Value

A standard **HRESULT**.

See Also

Reference

CDBErrorInfo Class

Other Resources

CDBErrorInfo Members

CDBErrorInfo::GetErrorParameters

Calls IErrorRecords::GetErrorParameters to return the error parameters.

```
HRESULT GetErrorParameters(
   ULONG ulRecordNum,
   DISPPARAMS* pdispparams
) const throw( );
```

Parameters

See IErrorRecords::GetErrorParameters in the *OLE DB Programmer's Reference*.

Return Value

A standard **HRESULT**.

See Also

Reference

CDBErrorInfo Class

Other Resources

CDBErrorInfo Members

CDBErrorInfo::GetErrorRecords

Gets error records for the specified object.

```
HRESULT GetErrorRecords(
   IUnknown* pUnk,
   const IID& iid,
   ULONG* pcRecords
) throw( );
HRESULT GetErrorRecords(
   ULONG* pcRecords
) throw( );
```

Parameters

pUnk

[in] Interface to the object for which to get error records.

iid

[in] The IID of the interface associated with the error.

pcRecords

[out] A pointer to the (one-based) count of error records.

Return Value

A standard **HRESULT**.

Remarks

Use the first form of the function if you want to check which interface to get the error information from. Otherwise, use the second form.

See Also

Reference

CDBErrorInfo Class

Other Resources

CDBErrorInfo Members

CDBPropIDSet Class

Inherits from the **DBPROPIDSET** structure and adds a constructor that initializes key fields as well as the AddPropertyID access method.

```
class CDBPropIDSet : public tagDBPROPIDSET
```

Remarks

OLE DB consumers use **DBPROPIDSET** structures to pass an array of property IDs for which the consumer wants to get property information. The properties identified in a single DBPROPIDSET structure belong to one property set.

Requirements

Header: atldbcli.h

See Also

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

Other Resources

CDBPropSet Members

CDBPropIDSet Members

Methods

AddPropertyID	Adds a property to the property ID set.
CDBPropIDSet	Constructor.
SetGUID	Sets the GUID of the property ID set.

Operators

operator =	Assigns the contents of one property ID set to another.

See Also

Reference

CDBPropIDSet Class

CDBPropIDSet Methods

For information about the methods in **CDBPropIDSet**, see CDBPropIDSet Members.

CDBPropIDSet::AddPropertyID

Adds a property ID to the property ID set.

```
bool AddPropertyID(
   DBPROPID propid
) throw( );
```

Parameters

propid

[in] The property ID to be added to the property ID set.

See Also

Reference

CDBPropIDSet Class

Other Resources

CDBPropIDSet Members

CDBPropIDSet::CDBPropIDSet

The constructor. Initializes the **rgProperties**, **cProperties**, and (optionally) **guidPropertySet** fields of the DBPROPIDSET structure.

```
CDBPropIDSet(
   const GUID& guid
);
CDBPropIDSet(
   const CDBPropIDSet& propidset
);
CDBPropIDSet( );
```

Parameters

guid

[in] A GUID used to initialize the **guidPropertySet** field.

propidset

[in] Another **CDBPropIDSet** object for copy construction.

See Also

Reference

CDBPropIDSet Class

CDBPropIDSet::SetGUID

Other Resources

CDBPropIDSet Members

CDBPropIDSet::SetGUID

Sets the GUID field in the **DBPROPIDSET** structure.

```
void SetGUID(
   const GUID& guid
) throw( );
```

Parameters

guid

[in] A GUID used to set the **guidPropertySet** field of the DBPROPIDSET structure.

Remarks

This field can be set by the constructor as well. Call this function if you use the default constructor for this class.

See Also

Reference

CDBPropIDSet Class

Other Resources

CDBPropIDSet Members

CDBPropIDSet Operators

For information about the operators in **CDBPropIDSet**, see CDBPropIDSet Members.

CDBPropIDSet::operator =

Assigns the contents of one property ID set to another ID property set.

```
CDBPropIDSet& operator =(
   CDBPropIDSet& propset
) throw( );
```

See Also

Reference

CDBPropIDSet Class

Other Resources

CDBPropIDSet Members

CDBPropSet Class

Inherits from the **DBPROPSET** structure and adds a constructor that initializes key fields as well as the **AddProperty** access method.

```
class CDBPropSet : public tagDBPROPSET
```

Remarks

OLE DB providers and consumers use **DBPROPSET** structures to pass arrays of **DBPROP** structures. Each **DBPROP** structure represents a single property that can be set.

Requirements

Header: atldbcli.h

See Also

Reference

CDBPropIDSet Class
DBPROPSET Structure
DBPROP Structure

Concepts

OLE DB Consumer Templates (C++)
OLE DB Consumer Templates Reference

Other Resources

CDBPropSet Members

CDBPropSet Members

Methods

AddProperty	Adds a property to the property set.
CDBPropSet	Constructor.
SetGUID	Sets the **guidPropertySet** field of the **DBPROPSET** structure.

Operators

operator =	Assigns the contents of one property set to another.

See Also

Reference

CDBPropSet Class

CDBPropSet Methods

For information about the methods in **CDBPropSet**, see CDBPropSet Members.

CDBPropSet::AddProperty

Adds a property to the property set.

```
bool AddProperty(
   DWORD dwPropertyID,
   const VARIANT& var,
   DBPROPOPTIONS propoptions = DBPROPOPTIONS_REQUIRED
) throw( );
bool AddProperty(
   DWORD dwPropertyID,
   LPCSTR szValue,
   DBPROPOPTIONS propoptions = DBPROPOPTIONS_REQUIRED
) throw( );
bool AddProperty(
   DWORD dwPropertyID,
   LPCWSTR szValue,
   DBPROPOPTIONS propoptions = DBPROPOPTIONS_REQUIRED
) throw( );
bool AddProperty(
   DWORD dwPropertyID,
   bool bValue,
   DBPROPOPTIONS propoptions = DBPROPOPTIONS_REQUIRED
) throw( );
bool AddProperty(
   DWORD dwPropertyID,
   BYTE bValue,
   DBPROPOPTIONS propoptions = DBPROPOPTIONS_REQUIRED
);
bool AddProperty(
   DWORD dwPropertyID,
   short nValue,
   DBPROPOPTIONS propoptions = DBPROPOPTIONS_REQUIRED
);
bool AddProperty(
   DWORD dwPropertyID,
   long nValue,
   DBPROPOPTIONS propoptions = DBPROPOPTIONS_REQUIRED
);
bool AddProperty(
   DWORD dwPropertyID,
   float fltValue,
   DBPROPOPTIONS propoptions = DBPROPOPTIONS_REQUIRED
);
bool AddProperty(
   DWORD dwPropertyID,
   double dblValue,
   DBPROPOPTIONS propoptions = DBPROPOPTIONS_REQUIRED
) throw( );
bool AddProperty(
   DWORD dwPropertyID,
   CY cyValue,
   DBPROPOPTIONS propoptions = DBPROPOPTIONS_REQUIRED
) throw( );
```

Parameters

dwPropertyID

[in] The ID of the property to be added. Used to initialize the **dwPropertyID** of the **DBPROP** structure added to the property set.

var

[in] A variant used to initialize the property value for the **DBPROP** structure added to the property set.

szValue

[in] A string used to initialize the property value for the **DBPROP** structure added to the property set.

bValue

[in] A **BYTE** or boolean value used to initialize the property value for the **DBPROP** structure added to the property set.

nValue

[in] An integer value used to initialize the property value for the **DBPROP** structure added to the property set.

fltValue

[in] A floating-point value used to initialize the property value for the **DBPROP** structure added to the property set.

dblValue

[in] A double-precision floating-point value used to initialize the property value for the **DBPROP** structure added to the property set.

cyValue

[in] A CY currency value used to initialize the property value for the **DBPROP** structure added to the property set.

Return Value

true if the property was successfully added. Otherwise, **false**.

See Also

Reference

CDBPropSet Class

DBPROP Structure

Other Resources

CDBPropSet Members

CDBPropSet::CDBPropSet

The constructor. Initializes the **rgProperties**, **cProperties**, and **guidPropertySet** fields of the DBPROPSET structure.

```
CDBPropSet(
   const GUID& guid
);
CDBPropSet(
   const CDBPropSet& propset
);
CDBPropSet( );
```

Parameters

guid

[in] A GUID used to initialize the **guidPropertySet** field.

propset

[in] Another **CDBPropSet** object for copy construction.

See Also

Reference

CDBPropSet Class
CDBPropSet::SetGUID
DBPROP Structure

Other Resources

CDBPropSet Members

CDBPropSet::SetGUID

Sets the **guidPropertySet** field in the **DBPROPSET** structure.

```
void SetGUID(
   const GUID& guid
) throw( );
```

Parameters

guid

[in] A GUID used to set the **guidPropertySet** field of the DBPROPSET structure.

Remarks

This field can be set by the constructor as well.

See Also

Reference

CDBPropSet Class

Other Resources

CDBPropSet Members

CDBPropSet Operators

For information about the operators in **CDBPropSet**, see CDBPropSet Members.

CDBPropSet::operator =

Assigns the contents of one property set to another property set.

```
CDBPropSet& operator =(
   CDBPropSet& propset
) throw( );
```

See Also

Reference

CDBPropSet Class

Other Resources

CDBPropSet Members

CDynamicAccessor Class

Allows you to access a data source when you have no knowledge of the database schema (the database's underlying structure).

```
class CDynamicAccessor : public CAccessorBase
```

Remarks

Use **CDynamicAccessor** methods to obtain column information such as column names, column count, data type, and so on. You then use this column information to create an accessor dynamically at run time.

The column information is stored in a buffer that is created and managed by this class. Obtain data from the buffer using GetValue.

For a discussion and examples of using the dynamic accessor classes, see Using Dynamic Accessors.

Requirements

Header: atldbcli.h

See Also

Reference

CAccessor Class
CDynamicParameterAccessor Class
CManualAccessor Class

Concepts

OLE DB Consumer Templates (C++)
OLE DB Consumer Templates Reference

Other Resources

CDynamicAccessor Members

CDynamicAccessor Members

Methods

AddBindEntry	Adds a bind entry to the output columns when overriding the default accessor.
CDynamicAccessor	Instantiates and initializes the **CDynamicAccessor** object.
Close	Unbinds all the columns, releases the allocated memory, and releases the IAccessor interface pointer in the class.
GetBookmark	Retrieves the bookmark for the current row.
GetBlobHandling	Retrieves the BLOB handling value for the current row.
GetBlobSizeLimit	Retrieves the maximum BLOB size in bytes.
GetColumnCount	Retrieves the number of columns in the rowset.
GetColumnFlags	Retrieves the column characteristics.
GetColumnInfo	Retrieves the column metadata.
GetColumnName	Retrieves the name of a specified column.
GetColumnType	Retrieves the data type of a specified column.
GetLength	Retrieves the maximum possible length of a column in bytes.
GetOrdinal	Retrieves the column index given a column name.
GetStatus	Retrieves the status of a specified column.
GetValue	Retrieves the data from the buffer.
SetBlobHandling	Sets the BLOB handling value for the current row.
SetBlobSizeLimit	Sets the maximum BLOB size in bytes.
SetLength	Sets the length of the column in bytes.
SetStatus	Sets the status of a specified column.
SetValue	Stores the data to the buffer.

See Also

Reference

CDynamicAccessor Class

CDynamicAccessor Methods

For information about the methods in **CDynamicAccessor**, see CDynamicAccessor Members.

CDynamicAccessor::AddBindEntry

Adds a bind entry to the output columns.

```
HRESULT AddBindEntry(
   const DBCOLUMNINFO& info
) throw( );
```

Parameters

info

[in] A **DBCOLUMNINFO** structure containing column information. See "DBCOLUMNINFO Structures" in IColumnsInfo::GetColumnInfo in the *OLE DB Programmer's Reference*.

Return Value

One of the standard **HRESULT** values.

Remarks

Use this method when overriding the default accessor created with **CDynamicAccessor** (see How Do I Fetch Data?).

See Also

Reference

CDynamicAccessor Class

Other Resources

CDynamicAccessor Members

CDynamicAccessor::CDynamicAccessor

Instantiates and initializes the **CDynamicAccessor** object.

```
CDynamicAccessor(
   DBBLOBHANDLINGENUM eBlobHandling = DBBLOBHANDLING_DEFAULT,
   DBLENGTH nBlobSize = 8000
);
```

Parameters

eBlobHandling

Specifies how the binary large object (BLOB) data is to be handled. The default value is **DBBLOBHANDLING_DEFAULT**. See SetBlobHandling for a description of the **DBBLOBHANDLINGENUM** values.

nBlobSize

The maximum BLOB size in bytes; column data over this value is treated as a BLOB. The default value is 8,000. See SetBlobSizeLimit for details.

Remarks

If you use the constructor to initialize the **CDynamicAccessor** object, you can specify how it will bind BLOBs. BLOBs can contain binary data such as graphics, sound, or compiled code. The default behavior is to treat columns more than 8,000 bytes as BLOBs and try to bind them to an **ISequentialStream** object. However, you can specify a different value to be the BLOB size.

You can also specify how **CDynamicAccessor** handles column data that qualifies as BLOB data: it can handle BLOB data in the default manner; it can skip (does not bind) BLOB data; or it can bind BLOB data in provider-allocated memory.

See Also

Reference

CDynamicAccessor Class

Other Resources

CDynamicAccessor Members

CDynamicAccessor::Close

Unbinds all the columns, releases the allocated memory, and releases the IAccessor interface pointer in the class.

```
void Close( ) throw( );
```

See Also

Reference

CDynamicAccessor Class

Other Resources

CDynamicAccessor Members

CDynamicAccessor::GetBlobHandling

Retrieves the BLOB handling value for the current row.

```
const DBBLOBHANDLINGENUM GetBlobHandling( ) const;
```

Remarks

Returns the BLOB handling value *eBlobHandling* as set by SetBlobHandling.

See Also

Reference

CDynamicAccessor Class

Other Resources

CDynamicAccessor Members

CDynamicAccessor::GetBlobSizeLimit

Retrieves the maximum BLOB size in bytes.

```
const DBLENGTH GetBlobSizeLimit( ) const;
```

Remarks

Returns the BLOB handling value *nBlobSize* as set by SetBlobSizeLimit.

See Also

Reference

CDynamicAccessor Class

Other Resources

CDynamicAccessor Members

CDynamicAccessor::GetBookmark

Retrieves the bookmark for the current row.

```
HRESULT GetBookmark(
   CBookmark< >* pBookmark
) const throw( );
```

Parameters

pBookmark

[out] A pointer to the CBookmark object.

Return Value

One of the standard **HRESULT** values.

Remarks

You need to set **DBPROP_IRowsetLocate** to **VARIANT_TRUE** to retrieve a bookmark.

See Also

Reference

CDynamicAccessor Class

Other Resources

CDynamicAccessor Members

CDynamicAccessor::GetColumnCount

Retrieves the number of columns.

```
DBORDINAL GetColumnCount( ) const throw( );
```

Return Value

The number of columns retrieved.

See Also

Reference

CDynamicAccessor Class

Other Resources

CDynamicAccessor Members

CDynamicAccessor::GetColumnFlags

Retrieves the column characteristics.

```
bool GetColumnFlags(
   DBORDINAL nColumn,
   DBCOLUMNFLAGS* pFlags
) const throw( );
```

Parameters

nColumn

[in] The column number. Column numbers start with 1. A value of 0 refers to the bookmark column, if any.

pFlags

[out] A pointer to a bitmask that describes column characteristics. See "DBCOLUMNFLAGS Enumerated Type" in IColumnsInfo::GetColumnInfo in the *OLE DB Programmer's Reference*.

Return Value

Returns **true** if the column characteristics are successfully retrieved. Otherwise, it returns **false**.

Remarks

The column number is offset from one. Column zero is a special case; it is the bookmark if available.

See Also

Reference

CDynamicAccessor Class

Other Resources

CDynamicAccessor Members

CDynamicAccessor::GetColumnInfo

Returns the column metadata needed by most consumers.

```
HRESULT GetColumnInfo(
   IRowset* pRowset,
   DBORDINAL* pColumns,
   DBCOLUMNINFO** ppColumnInfo,
   OLECHAR** ppStringsBuffer
) throw( );
```

Parameters

pRowset

[in] A pointer to the IRowset interface.

pColumns

[out] A pointer to memory in which to return the number of columns in the rowset; this number includes the bookmark column, if there is one.

ppColumnInfo

[out] A pointer to memory in which to return an array of **DBCOLUMNINFO** structures. See "DBCOLUMNINFO Structures" in IColumnsInfo::GetColumnInfo in the *OLE DB Programmer's Reference*.

ppStringsBuffer

[out] A pointer to memory in which to return a pointer to storage for all string values (names used either within *columnid* or for *pwszName*) within a single allocation block.

Return Value

One of the standard **HRESULT** values.

Remarks

See IColumnsInfo::GetColumnInfo in the *OLE DB Programmer's Reference* for information on the data types **DBORDINAL**, **DBCOLUMNINFO**, and **OLECHAR**.

See Also

Reference

CDynamicAccessor Class

Other Resources

CDynamicAccessor Members

CDynamicAccessor::GetColumnName

Retrieves the name of the specified column.

```
LPOLESTR GetColumnName(
   DBORDINAL nColumn
) const throw( );
```

Parameters

nColumn

[in] The column number. Column numbers start with 1. A value of 0 refers to the bookmark column, if any.

Return Value

The name of the specified column.

See Also

Reference

CDynamicAccessor Class

Other Resources

CDynamicAccessor Members

CDynamicAccessor::GetColumnType

Retrieves the data type of a specified column.

```
bool GetColumnType(
   DBORDINAL nColumn,
   DBTYPE* pType
) const throw( );
```

Parameters

nColumn

[in] The column number. Column numbers start with 1. A value of 0 refers to the bookmark column, if any.

pType

[out] A pointer to the data type of the specified column.

Return Value

Returns **true** on success or **false** on failure.

See Also

Reference

CDynamicAccessor Class

DBTYPE

Other Resources

CDynamicAccessor Members

CDynamicAccessor::GetLength

Retrieves the length of the specified column.

```
bool GetLength(
   DBORDINAL nColumn,
   DBLENGTH* pLength
) const throw( );
bool GetLength(
   const CHAR* pColumnName,
   DBLENGTH* pLength
) const throw( );
bool GetLength(
   const WCHAR* pColumnName,
   DBLENGTH* pLength
) const throw( );
```

Parameters

nColumn

[in] The column number. Column numbers start with 1. A value of 0 refers to the bookmark column, if any.

pColumnName

[in] A pointer to a character string containing the column name.

pLength

[out] A pointer to the integer containing the length of the column in bytes.

Return Value

Returns **true** if the specified column is found. Otherwise, this function returns **false**.

Remarks

The first override takes the column number, and the second and third overrides take the column name in ANSI or Unicode format, respectively.

See Also

Reference

CDynamicAccessor Class

CDynamicAccessor::SetLength

Other Resources

CDynamicAccessor Members

CDynamicAccessor::GetOrdinal

Retrieves the column number given a column name.

```
bool GetOrdinal(
   const CHAR* pColumnName,
   DBORDINAL* pOrdinal
) const throw( );
bool GetOrdinal(
   const WCHAR* pColumnName,
   DBORDINAL* pOrdinal
) const throw( );
```

Parameters

pColumnName

[in] A pointer to a character string containing the column name.

pOrdinal

[out] A pointer to the column number.

Return Value

Returns **true** if a column with the specified name is found. Otherwise, this function returns **false**.

See Also

Reference

CDynamicAccessor Class

Other Resources

CDynamicAccessor Members

CDynamicAccessor::GetStatus

Retrieves the status of the specified column.

```
bool GetStatus(
   DBORDINAL nColumn,
   DBSTATUS* pStatus
) const throw( );
bool GetStatus(
   const CHAR* pColumnName,
   DBSTATUS* pStatus
) const throw( );
bool GetStatus(
   const WCHAR* pColumnName,
   DBSTATUS* pStatus
) const throw( );
```

Parameters

nColumn

[in] The column number. Column numbers start with 1. A value of 0 refers to the bookmark column, if any.

pColumnName

[in] A pointer to a character string containing the column name.

pStatus

[out] A pointer to the variable containing the column status. See DBSTATUS in the *OLE DB Programmer's Reference* for more information.

Return Value

Returns **true** if the specified column is found. Otherwise, this function returns **false**.

See Also

Reference

CDynamicAccessor Class

CDynamicAccessor::SetStatus

Other Resources

CDynamicAccessor Members

CDynamicAccessor::GetValue

Retrieves the data for a specified column.

```
void* GetValue(
   DBORDINAL nColumn
) const throw( );
void* GetValue(
   const CHAR* pColumnName
) const throw( );
void* GetValue(
   const WCHAR* pColumnName
) const throw( );
template < class ctype >
bool GetValue(
   DBORDINAL nColumn,
   ctype* pData
) const throw( );
template < class ctype >
bool GetValue(
   const CHAR* pColumnName,
   ctype* pData
) const throw( );
template < class ctype >
bool GetValue(
   const WCHAR* pColumnName,
   ctype* pData
) const throw( );
```

Parameters

ctype

[in] A templated parameter that handles any data type except string types (**CHAR***, **WCHAR***), which require special handling. **GetValue** uses the appropriate data type based on what you specify here.

nColumn

[in] The column number. Column numbers start with 1. A value of 0 refers to the bookmark column, if any.

pColumnName

[in] The column name.

pData

[out] The pointer to the contents of the specified column.

Return Value

If you want to pass string data, use the nontemplated versions of **GetValue**. The nontemplated versions of this method return **void***, which points to the part of the buffer that contains the specified column data. Returns **NULL** if the column is not found.

For all other data types, it is simpler to use the templated versions of **GetValue**. The templated versions return **true** on success or **false** on failure.

Remarks

Use the nontemplated versions to return columns that contain strings and the templated versions for columns that contain other data types.

In debug mode, you will get an assertion if the size of *pData* is unequal to the size of the column to which it points.

See Also

Reference

CDynamicAccessor Class

Other Resources

CDynamicAccessor Members

CDynamicAccessor::SetBlobHandling

Sets the BLOB handling value for the current row.

```
bool SetBlobHandling(
   DBBLOBHANDLINGENUM eBlobHandling
);
```

Parameters
eBlobHandling

Specifies how the BLOB data is to be handled. It can take the following values:

- **DBBLOBHANDLING_DEFAULT**: Handle column data larger than *nBlobSize* (as set by **SetBlobSizeLimit**) as BLOB data and retrieve it through an **ISequentialStream** or **IStream** object. This option will attempt to bind every column containing data larger than *nBlobSize* or listed as **DBTYPE_IUNKNOWN** as BLOB data.
- **DBBLOBHANDLING_NOSTREAMS**: Handle column data larger than *nBlobSize* (as set by **SetBlobSizeLimit**) as BLOB data and retrieve it through reference in provider-allocated, consumer-owned memory. This option is useful for tables that have more than one BLOB column, and the provider supports only one **ISequentialStream** object per accessor.
- **DBBLOBHANDLING_SKIP**: Skip (do not bind) columns qualifying as containing BLOBs (the accessor will not bind or retrieve the column value but it will still retrieve the column status and length).

Remarks

You should call **SetBlobHandling** before calling **Open**.

The constructor method CDynamicAccessor sets the BLOB handling value to **DBBLOBHANDLING_DEFAULT**.

See Also
Reference
CDynamicAccessor Class
Other Resources
CDynamicAccessor Members

CDynamicAccessor::SetBlobSizeLimit

Sets the maximum BLOB size in bytes.

```
void SetBlobSizeLimit(
   DBLENGTH nBlobSize
);
```

Parameters

nBlobSize

Specifies the BLOB size limit.

Remarks

Sets the maximum BLOB size in bytes; column data larger than this value is treated as a BLOB. Some providers give extremely large sizes for columns (such as 2 GB). Rather than attempting to allocate memory for a column this size, you would typically try to bind these columns as BLOBs. In that way you don't have to allocate all the memory, but you can still read all the data without fear of truncation. However, there are some cases in which you might want to force **CDynamicAccessor** to bind large columns in their native data types. To do this, call **SetBlobSizeLimit** before calling **Open**.

The constructor method CDynamicAccessor sets the maximum BLOB size to a default value of 8,000 bytes.

See Also

Reference

CDynamicAccessor Class

Other Resources

CDynamicAccessor Members

CDynamicAccessor::SetLength

Sets the length of the specified column.

```
bool SetLength(
   DBORDINAL nColumn,
   DBLENGTH nLength
) throw( );
bool SetLength(
   const CHAR* pColumnName,
   DBLENGTH nLength
) throw( );
bool SetLength(
   const WCHAR* pColumnName,
   DBLENGTH nLength
) throw( );
```

Parameters

nColumn

[in] The column number. Column numbers start with 1. A value of 0 refers to the bookmark column, if any.

nLength

[in] The length of the column in bytes.

pColumnName

[in] A pointer to a character string containing the column name.

Return Value

Returns **true** if the specified column length is set successfully. Otherwise, this function returns **false**.

See Also

Reference

CDynamicAccessor Class

CDynamicAccessor::GetLength

Other Resources

CDynamicAccessor Members

CDynamicAccessor::SetStatus

Sets the status of the specified column.

```
bool SetStatus(
   DBORDINAL nColumn,
   DBSTATUS status
) throw( );
bool SetStatus(
   const CHAR* pColumnName,
   DBSTATUS status
) throw( );
bool SetStatus(
   const WCHAR* pColumnName,
   DBSTATUS status
) throw( );
```

Parameters

nColumn

[in] The column number. Column numbers start with 1. A value of 0 refers to the bookmark column, if any.

status

[in] The column status. See DBSTATUS in the *OLE DB Programmer's Reference* for more information.

pColumnName

[in] A pointer to a character string containing the column name.

Return Value

Returns **true** if the specified column status is set successfully. Otherwise, this function returns **false**.

See Also

Reference

CDynamicAccessor Class

CDynamicAccessor::GetStatus

Other Resources

CDynamicAccessor Members

CDynamicAccessor::SetValue

Stores data to a specified column.

```
template < class ctype >
bool SetValue(
   DBORDINAL nColumn,
   const ctype& data
) throw( );
template < class ctype >
bool SetValue(
   const CHAR * pColumnName,
   const ctype& data
) throw( );
template <class ctype>
bool SetValue(
   const WCHAR *pColumnName,
   const ctype& data
) throw( );
```

Parameters

ctype

[in] A templated parameter that handles any data type except string types (**CHAR***, **WCHAR***), which require special handling. **GetValue** uses the appropriate data type based on what you specify here.

pColumnName

[in] A pointer to a character string containing the column name.

data

[in] The pointer to the memory containing the data.

nColumn

[in] The column number. Column numbers start with 1. A value of 0 refers to the bookmark column, if any.

Return Value

If you want to set string data, use the nontemplated versions of **GetValue**. The nontemplated versions of this method return **void***, which points to the part of the buffer that contains the specified column data. Returns **NULL** if the column is not found.

For all other data types, it is simpler to use the templated versions of **GetValue**. The templated versions return **true** on success or **false** on failure.

See Also

Reference

CDynamicAccessor Class

Other Resources

CDynamicAccessor Members

CDynamicParameterAccessor Class

Similar to CDynamicAccessor but obtains parameter information to be set by calling the ICommandWithParameters interface.

```
class CDynamicParameterAccessor : public CDynamicAccessor
```

Remarks

The provider must support **ICommandWithParameters** for the consumer to use this class.

The parameter information is stored in a buffer created and managed by this class. Obtain parameter data from the buffer by using GetParam and GetParamType.

For an example demonstrating how to use this class to execute a SQL Server stored procedure and get the output parameter values, see Knowledge Base article Q058860, "HOWTO: Execute Stored Procedure using CDynamicParameterAccessor." Knowledge Base articles are available in the MSDN Library Visual Studio documentation or at http://support.microsoft.com/support/.

Requirements

Header: atldbcli.h

See Also

Reference

CAccessor Class
CDynamicAccessor Class
CManualAccessor Class

Concepts

OLE DB Consumer Templates (C++)
OLE DB Consumer Templates Reference

Other Resources

CDynamicParameterAccessor Members

CDynamicParameterAccessor Members

Methods

CDynamicParameterAccessor	The constructor.
GetParam	Retrieves the parameter data from the buffer.
GetParamCount	Retrieves the number of parameters in the accessor.
GetParamIO	Determines whether the specified parameter is an input or output parameter.
GetParamLength	Retrieves the length of the specified parameter stored in the buffer.
GetParamName	Retrieves the name of a specified parameter.
GetParamStatus	Retrieves the status of the specified parameter stored in the buffer.
GetParamString	Retrieves the string data of the specified parameter stored in the buffer.
GetParamType	Retrieves the data type of a specified parameter.
SetParam	Sets the buffer using the parameter data.
SetParamLength	Sets the length of the specified parameter stored in the buffer.
SetParamStatus	Sets the status of the specified parameter stored in the buffer.
SetParamString	Sets the string data of the specified parameter stored in the buffer.

See Also

Reference

CDynamicParameterAccessor Class

CDynamicParameterAccessor Methods

For information about the methods in **CDynamicParameterAccessor**, see CDynamicParameterAccessor Members.

CDynamicParameterAccessor::CDynamicParameterAccessor

The constructor.

```
typedef CDynamicParameterAccessor _ParamClass;
CDynamicParameterAccessor(
   DBBLOBHANDLINGENUM eBlobHandling = DBBLOBHANDLING_DEFAULT,
   DBLENGTH nBlobSize = 8000 )
   : CDynamicAccessor( eBlobHandling, nBlobSize )
```

Parameters

eBlobHandling

Specifies how the BLOB data is to be handled. The default value is **DBBLOBHANDLING_DEFAULT**. See CDynamicAccessor::SetBlobHandling for a description of the **DBBLOBHANDLINGENUM** values.

nBlobSize

The maximum BLOB size in bytes; column data over this value is treated as a BLOB. The default value is 8,000. See CDynamicAccessor::SetBlobSizeLimit for details.

Remarks

See the CDynamicAccessor::CDynamicAccessor constructor for more information on BLOB handling.

See Also

Reference

CDynamicParameterAccessor Class

Other Resources

CDynamicParameterAccessor Members

CDynamicParameterAccessor::GetParam

Retrieves the nonstring data for a specified parameter from the parameter buffer.

```
template < class ctype > bool GetParam(
   DBORDINAL nParam,
   ctype* pData
) const throw( );
template < class ctype > bool GetParam(
   TCHAR* pParamName,
   ctype* pData
) const throw( );
void* GetParam(
   DBORDINAL nParam
) const throw( );
void* GetParam(
   TCHAR* pParamName
) const throw( );
```

Parameters

ctype

A templated parameter that is the data type.

nParam

[in] The parameter number (offset from 1). Parameter 0 is reserved for return values. The parameter number is the index of the parameter based on its order in the SQL or stored procedure call. See SetParam for an example.

pParamName

[in] The parameter name.

pData

[out] The pointer to the memory containing the data retrieved from the buffer.

Return Value

For nontemplated versions, points to the memory containing the data retrieved from the buffer. For templated versions, returns **true** on success or **false** on failure.

Use **GetParam** to retrieve nonstring parameter data from the buffer. Use GetParamString to retrieve string parameter data from the buffer.

See Also

Reference

CDynamicParameterAccessor Class

Other Resources

CDynamicParameterAccessor Members

CDynamicParameterAccessor::GetParamCount

Retrieves the number of parameters stored in the buffer.

```
DB_UPARAMS GetParamCount( ) const throw( );
```

Return Value

The number of parameters.

See Also

Reference

CDynamicParameterAccessor Class

Other Resources

CDynamicParameterAccessor Members

CDynamicParameterAccessor::GetParamIO

Determines whether the specified parameter is an input or output parameter.

```
bool GetParamIO(
   DBORDINAL nParam,
   DBPARAMIO * pParamIO
) const throw( );
```

Parameters

nParam

[in] The parameter number (offset from 1). Parameter 0 is reserved for return values. The parameter number is the index of the parameter based on its order in the SQL or stored procedure call. See SetParam for an example.

pParamIO

A pointer to the variable containing the **DBPARAMIO** type (input or output) of the specified parameter. It is defined as follows:

```
typedef DWORD DBPARAMIO;
enum DBPARAMIOENUM
   {   DBPARAMIO_NOTPARAM   = 0,
       DBPARAMIO_INPUT      = 0x1,
       DBPARAMIO_OUTPUT     = 0x2
   };
```

Return Value

Returns **true** on success or **false** on failure.

See Also

Reference

CDynamicParameterAccessor Class

Other Resources

CDynamicParameterAccessor Members

CDynamicParameterAccessor::GetParamLength

Retrieves the length of the specified parameter stored in the buffer.

```
bool GetParamLength(
   DBORDINAL nParam,
   DBLENGTH* pLength
);
DBLENGTH* GetParamLength(
   DBORDINAL nParam
) const throw( );
```

Parameters

nParam

[in] The parameter number (offset from 1). Parameter 0 is reserved for return values. The parameter number is the index of the parameter based on its order in the SQL or stored procedure call. See SetParam for an example.

pLength

[out] A pointer to the variable containing the length in bytes of the specified parameter.

Remarks

The first override returns **true** on success or **false** on failure. The second override points to the memory containing the length of the parameter.

See Also

Reference

CDynamicParameterAccessor Class

Other Resources

CDynamicParameterAccessor Members

CDynamicParameterAccessor::GetParamName

Retrieves the name of the specified parameter.

```
LPOLESTR GetParamName(
   DBORDINAL nParam
) const throw( );
```

Parameters

nParam

[in] The parameter number (offset from 1). Parameter 0 is reserved for return values. The parameter number is the index of the parameter based on its order in the SQL or stored procedure call. See SetParam for an example.

Return Value

The name of the specified parameter.

See Also

Reference

CDynamicParameterAccessor Class

Other Resources

CDynamicParameterAccessor Members

CDynamicParameterAccessor::GetParamStatus

Retrieves the status of the specified parameter stored in the buffer.

```
bool GetParamStatus(
   DBORDINAL nParam,
   DBSTATUS* pStatus
);
DBSTATUS* GetParamStatus(
   DBORDINAL nParam
) const throw( );
```

Parameters

nParam

[in] The parameter number (offset from 1). Parameter 0 is reserved for return values. The parameter number is the index of the parameter based on its order in the SQL or stored procedure call. See SetParam for an example.

pStatus

[out] A pointer to the variable containing the **DBSTATUS** status of the specified parameter. For information on **DBSTATUS** values, see Status in the *OLE DB Programmer's Reference*, or search for **DBSTATUS** in oledb.h.

Remarks

The first override returns **true** on success or **false** on failure. The second override points to the memory containing the status of the specified parameter.

See Also

Reference

CDynamicParameterAccessor Class

Other Resources

CDynamicParameterAccessor Members

CDynamicParameterAccessor::GetParamString

Retrieves the string data of the specified parameter stored in the buffer.

```
bool GetParamString(
   DBORDINAL nParam,
   CSimpleStringA& strOutput
) throw( );
bool GetParamString(
   DBORDINAL nParam,
   CSimpleStringW& strOutput
) throw( );
bool GetParamString(
   DBORDINAL nParam,
   CHAR* pBuffer,
   size_t* pMaxLen
) throw( );
bool GetParamString(
   DBORDINAL nParam,
   WCHAR* pBuffer,
   size_t* pMaxLen
) throw( );
```

Parameters

nParam

[in] The parameter number (offset from 1). Parameter 0 is reserved for return values. The parameter number is the index of the parameter based on its order in the SQL or stored procedure call. See SetParam for an example.

strOutput

[out] The ANSI (**CSimpleStringA**) or Unicode (**CSimpleStringW**) string data of the specified parameter. You should pass a parameter of type **CString**, for example:

```
CString MyString;
GetParamString(1, MyString);
```

pBuffer

[out] A pointer to the ANSI (**CHAR**) or Unicode (**WCHAR**) string data of the specified parameter.

pMaxLen

[out] A pointer to the size of the buffer pointed to by *pBuffer* (in characters, including the terminating NULL).

Remarks

Returns **true** on success or **false** on failure.

If *pBuffer* is NULL, this method will set the required buffer size in the memory pointed to by *pMaxLen* and return **true** without copying the data.

This method will fail if the buffer *pBuffer* is not large enough to contain the whole string.

Use **GetParamString** to retrieve string parameter data from the buffer. Use GetParam to retrieve nonstring parameter data from the buffer.

See Also

Reference

CDynamicParameterAccessor Class

Other Resources

CDynamicParameterAccessor Members

CDynamicParameterAccessor::GetParamType

Retrieves the data type of a specified parameter.

```
bool GetParamType(
   DBORDINAL nParam,
   DBTYPE* pType
) const throw( );
```

Parameters

nParam

[in] The parameter number (offset from 1). Parameter 0 is reserved for return values. The parameter number is the index of the parameter based on its order in the SQL or stored procedure call. See SetParam for an example.

pType

[out] A pointer to the variable containing the data type of the specified parameter.

Return Value

Returns **true** on success or **false** on failure.

See Also

Reference

CDynamicParameterAccessor Class

Other Resources

CDynamicParameterAccessor Members

CDynamicParameterAccessor::SetParam

Sets the parameter buffer using the specified (non-string) data.

```
template < class ctype >
bool SetParam(
   DBORDINAL nParam,
   const ctype* pData,
   DBSTATUS status = DBSTATUS_S_OK
) throw( );
template < class ctype >
bool SetParam(
   TCHAR* pParamName,
   const ctype* pData,
   DBSTATUS status = DBSTATUS_S_OK
) throw( );
```

Parameters

ctype

A templated parameter that is the data type.

nParam

[in] The parameter number (offset from 1). Parameter 0 is reserved for return values. The parameter number is the index of the parameter based on its order in the SQL or stored procedure call. For example:

```
SELECT * FROM Authors WHERE State='?' AND LName='?'

CTable<CDynamicAccessor> rs;
...
rs.SetParam(1, m_state);
rs.SetParam(2, m_lastname);
```

pParamName

[in] The parameter name.

pData

[in] The pointer to the memory containing the data to be written to the buffer.

status

[in] The **DBSTATUS** column status. For information on **DBSTATUS** values, see Status in the *OLE DB Programmer's Reference*, or search for **DBSTATUS** in oledb.h.

Return Value

Returns **true** on success or **false** on failure.

Use **SetParam** to set nonstring parameter data in the buffer. Use SetParamString to set string parameter data in the buffer.

See Also

Reference

CDynamicParameterAccessor Class

Other Resources

CDynamicParameterAccessor Members

CDynamicParameterAccessor::SetParamLength

Sets the length of the specified parameter stored in the buffer.

```
bool SetParamLength(
   DBORDINAL nParam,
   DBLENGTH length
);
```

Parameters

nParam

[in] The parameter number (offset from 1). Parameter 0 is reserved for return values. The parameter number is the index of the parameter based on its order in the SQL or stored procedure call. See SetParam for an example.

length

[in] The length in bytes of the specified parameter.

Remarks

Returns **true** on success or **false** on failure.

See Also

Reference

CDynamicParameterAccessor Class

Other Resources

CDynamicParameterAccessor Members

CDynamicParameterAccessor::SetParamStatus

Sets the status of the specified parameter stored in the buffer.

```
bool SetParamStatus(
   DBORDINAL nParam,
   DBSTATUS status
);
```

Parameters

nParam

[in] The parameter number (offset from 1). Parameter 0 is reserved for return values. The parameter number is the index of the parameter based on its order in the SQL or stored procedure call. See SetParam for an example.

status

[in] The **DBSTATUS** status of the specified parameter. For information on **DBSTATUS** values, see Status in the *OLE DB Programmer's Reference*, or search for **DBSTATUS** in oledb.h.

Remarks

Returns **true** on success or **false** on failure.

See Also

Reference

CDynamicParameterAccessor Class

Other Resources

CDynamicParameterAccessor Members

CDynamicParameterAccessor::SetParamString

Sets the string data of the specified parameter stored in the buffer.

```
bool SetParamString(
   DBORDINAL nParam,
   const CHAR* pString,
   DBSTATUS status = DBSTATUS_S_OK
) throw( );
bool SetParamString(
   DBORDINAL nParam,
   const WCHAR* pString,
   DBSTATUS status = DBSTATUS_S_OK
) throw( );
```

Parameters

nParam

[in] The parameter number (offset from 1). Parameter 0 is reserved for return values. The parameter number is the index of the parameter based on its order in the SQL or stored procedure call. See SetParam for an example.

pString

[in] A pointer to the ANSI (**CHAR**) or Unicode (**WCHAR**) string data of the specified parameter. See **DBSTATUS** in oledb.h.

status

[in] The **DBSTATUS** status of the specified parameter. For information on **DBSTATUS** values, see Status in the *OLE DB Programmer's Reference*, or search for **DBSTATUS** in oledb.h.

Remarks

Returns **true** on success or **false** on failure.

SetParamString will fail if you try to set a string that is larger than the maximum size specified for *pString*.

Use **SetParamString** to set string parameter data in the buffer. Use SetParam to set nonstring parameter data in the buffer.

See Also

Reference

CDynamicParameterAccessor Class

Other Resources

CDynamicParameterAccessor Members

CDynamicStringAccessor Class

Allows you to access a data source when you have no knowledge of the database schema (the database's underlying structure).

```
template< typename BaseType, DBTYPEENUM OleDbType >
class CDynamicStringAccessorT : public CDynamicAccessor
```

Remarks

While CDynamicAccessor requests data in the native format reported by the provider, **CDynamicStringAccessor** requests that the provider fetch all data accessed from the data store as string data. This is especially useful for simple tasks that do not require calculation of values in the data store, such as displaying or printing the data store's contents.

The native type of column data in the data store does not matter; as long as the provider can support the data conversion, it will supply the data in string format. If the provider does not support the conversion from the native data type to a string (which is not common), the requesting call will return the success value **DB_S_ERRORSOCCURED**, and the status for the corresponding column will indicate a conversion problem with **DBSTATUS_E_CANTCONVERTVALUE**.

Use **CDynamicStringAccessor** methods to obtain column information. You use this column information to create an accessor dynamically at run time.

The column information is stored in a buffer created and managed by this class. Obtain data from the buffer using GetString, or store it to the buffer using SetString.

For a discussion and examples of using the dynamic accessor classes, see Using Dynamic Accessors.

Requirements

Header: atldbcli.h

See Also

Reference

CAccessor Class
CDynamicParameterAccessor Class
CManualAccessor Class
CDynamicAccessor Class
CDynamicStringAccessorA Class
CDynamicStringAccessorW Class
CXMLAccessor Class

Concepts

OLE DB Consumer Templates (C++)
OLE DB Consumer Templates Reference

Other Resources

CDynamicAccessor Members

CDynamicStringAccessor Members

Methods

GetString	Retrieves the specified column data as a string.
SetString	Sets the specified column data as a string.

See Also

Reference

CDynamicStringAccessor Class

CDynamicStringAccessor Methods

For information about the methods in **CDynamicStringAccessor**, see CDynamicStringAccessor Members.

CDynamicStringAccessor::GetString

Retrieves the specified column data as a string.

```
BaseType* GetString(
    DBORDINAL nColumn
) const throw( );
BaseType* GetString(
    const CHAR* pColumnName
) const throw( );
BaseType* GetString(
    const WCHAR* pColumnName
) const throw( );
```

Parameters

nColumn

[in] The column number. Column numbers start with 1. A value of 0 refers to the bookmark column, if any.

pColumnName

[in] A pointer to a character string containing the column name.

Return Value

A pointer to the string value retrieved from the specified column. The value is of type ***BaseType***, which will be **CHAR** or **WCHAR** depending on whether _UNICODE is defined or not.

Remarks

The second override form takes the column name as an ANSI string and the third override form takes the column name as a Unicode string.

See Also

Reference

CDynamicStringAccessor Class

Other Resources

CDynamicStringAccessor Members

CDynamicStringAccessor::SetString

Sets the specified column data as a string.

```
HRESULT SetString(
    DBORDINAL nColumn,
    BaseType* data
) throw( );
HRESULT SetString(
    const CHAR* pColumnName,
    BaseType* data
) throw( );
HRESULT SetString(
    const WCHAR* pColumnName,
    BaseType* data
) throw( );
```

Parameters

nColumn

[in] The column number. Column numbers start with 1. A value of 0 refers to the bookmark column, if any.

pColumnName

[in] A pointer to a character string containing the column name.

data

[in] A pointer to the string data to be written to the specified column.

Return Value

A pointer to the string value to which to set the specified column. The value is of type ***BaseType***, which will be **CHAR** or **WCHAR** depending on whether _UNICODE is defined or not.

Remarks

The second override form takes the column name as an ANSI string and the third override form takes the column name as a Unicode string.

See Also

Reference

CDynamicStringAccessor Class

Other Resources

CDynamicStringAccessor Members

CDynamicStringAccessorA Class

Allows you to access a data source when you have no knowledge of the database schema (underlying structure).

```
typedef CDynamicStringAccessorT<CHAR, DBTYPE_STR> CDynamicStringAccessorA;
```

Remarks

They both request that the provider fetch all data accessed from the data store as string data, but **CDynamicStringAccessor** requests ANSI string data.

CDynamicStringAccessorA inherits **GetString** and **SetString** from **CDynamicStringAccessor**. When you use these methods in a **CDynamicStringAccessorA** object, ***BaseType*** is **CHAR**.

Requirements

Header: atldbcli.h

See Also

Reference

CAccessor Class
CDynamicParameterAccessor Class
CManualAccessor Class
CDynamicAccessor Class
CDynamicStringAccessor Class

Concepts

OLE DB Consumer Templates (C++)
OLE DB Consumer Templates Reference

Other Resources

CDynamicAccessor Members

CDynamicStringAccessorW Class

Allows you to access a data source when you have no knowledge of the database schema (underlying structure).

```
typedef CDynamicStringAccessorT<WCHAR, DBTYPE_WSTR> CDynamicStringAccessorW;
```

Remarks

They both request that the provider fetch all data accessed from the data store as string data, but **CDynamicStringAccessor** requests Unicode string data.

CDynamicStringAccessorW inherits **GetString** and **SetString** from **CDynamicStringAccessor**. When you use these methods in a **CDynamicStringAccessorW** object, ***BaseType*** is **WCHAR**.

Requirements

Header: atldbcli.h

See Also

Reference

CAccessor Class
CDynamicParameterAccessor Class
CManualAccessor Class
CDynamicAccessor Class
CDynamicStringAccessor Class

Concepts

OLE DB Consumer Templates (C++)
OLE DB Consumer Templates Reference

Other Resources

CDynamicAccessor Members

CEnumerator Class

Uses an OLE DB enumerator object, which exposes the ISourcesRowset interface to return a rowset describing all data sources and enumerators.

```
class CEnumerator :
   public CAccessorRowset< CAccessor <CEnumeratorAccessor >>
```

Remarks

You can retrieve the **ISourcesRowset** data indirectly from this class.

Requirements

Header:atldbcli.h

See Also

Tasks

CatDB Sample: Data Source Schema Browser

DBViewer Sample: Database Browser

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

Other Resources

CEnumerator Members

CEnumerator Members

Methods

Find	Searches through available providers (data sources) looking for one with the specified name.
GetMoniker	Retrieves the **IMoniker** interface for the current record.
Open	Opens the enumerator.

See Also

Reference

CEnumerator Class

CEnumerator Methods

For information about the methods in **CEnumerator**, see CEnumerator Members.

CEnumerator::Find

Looks for a specified name among available providers.

```
bool Find(
   TCHAR* szSearchName
) throw( );
```

Parameters

szSearchName

[in] The name to search for.

Return Value

true if the name was found. Otherwise, **false**.

Remarks

This name maps to the **SOURCES_NAME** member of the ISourcesRowset interface.

See Also

Reference

CEnumerator Class

Other Resources

CEnumerator Members

CEnumerator::GetMoniker

Parses the display name to extract the component of the string that can be converted into a moniker.

```
HRESULT GetMoniker(
   LPMONIKER* ppMoniker
) const throw( );
HRESULT GetMoniker(
   LPMONIKER* ppMoniker,
   LPCTSTR lpszDisplayName
) const throw( );
```

Parameters

ppMoniker

[out] The moniker parsed from the display name (CEnumeratorAccessor::m_szParseName) of the current row.

lpszDisplayName

[in] The display name to parse.

Return Value

A standard **HRESULT**.

See Also

Reference

CEnumerator Class

Other Resources

CEnumerator Members

CEnumerator::Open

Binds the moniker for the enumerator, if one is specified, then retrieves the rowset for the enumerator by calling ISourcesRowset::GetSourcesRowset.

```
HRESULT Open(
   LPMONIKER pMoniker
) throw( );
HRESULT Open(
   const CLSID* pClsid = & CLSID_OLEDB_ENUMERATOR
) throw( );
HRESULT Open(
   const CEnumerator& enumerator
) throw( );
```

Parameters

pMoniker

[in] A pointer to a moniker for an enumerator.

pClsid

[in] A pointer to the **CLSID** of an enumerator.

enumerator

[in] A reference to an enumerator.

Return Value

A standard **HRESULT**.

See Also

Reference

CEnumerator Class

Other Resources

CEnumerator Members

CEnumeratorAccessor Class

Used by CEnumerator to access the data from the enumerator rowset.

```
class CEnumeratorAccessor
```

Remarks

This rowset consists of the data sources and enumerators visible from the current enumerator.

Requirements

Header: atldbcli.h

See Also

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

Other Resources

CEnumeratorAccessor Members

CEnumeratorAccessor Members

Data Members

m_bIsParent	A variable indicating whether the enumerator is a parent enumerator, if the row is an enumerator.
m_nType	A variable indicating whether the row describes a data source or an enumerator.
m_szDescription	The description of the data source or enumerator.
m_szName	The name of the data source or enumerator.
m_szParseName	String to pass to IParseDisplayName to obtain a moniker for the data source or enumerator.

See Also

Reference

CEnumeratorAccessor Class

CEnumeratorAccessor Data Members

For information about the data members in **CEnumeratorAccessor**, see CEnumeratorAccessor Members.

CEnumeratorAccessor::m_bIsParent

A variable indicating whether the enumerator is a parent enumerator, if the row is an enumerator.

```
VARIANT_BOOL m_bIsParent;
```

Remarks

See ISourcesRowset::GetSourcesRowset in the *OLE DB Programmer's Reference* for more information.

See Also

Reference

CEnumeratorAccessor Class

Other Resources

CEnumeratorAccessor Members

CEnumeratorAccessor::m_nType

A variable indicating whether the row describes a data source or an enumerator.

```
USHORT m_nType;
```

Remarks

See ISourcesRowset::GetSourcesRowset in the *OLE DB Programmer's Reference* for more information.

See Also

Reference

CEnumeratorAccessor Class

Other Resources

CEnumeratorAccessor Members

CEnumeratorAccessor::m_szDescription

The description of the data source or enumerator.

```
WCHAR m_szDescription[129];
```

Remarks

See ISourcesRowset::GetSourcesRowset in the *OLE DB Programmer's Reference* for more information.

See Also

Reference

CEnumeratorAccessor Class

Other Resources

CEnumeratorAccessor Members

CEnumeratorAccessor::m_szName

The name of the data source or enumerator.

```
WCHAR m_szName[129];
```

Remarks

See ISourcesRowset::GetSourcesRowset in the *OLE DB Programmer's Reference* for more information.

See Also

Reference

CEnumeratorAccessor Class

Other Resources

CEnumeratorAccessor Members

CEnumeratorAccessor::m_szParseName

String to pass to IParseDisplayName to obtain a moniker for the data source or enumerator.

```
WCHAR m_szParseName[129];
```

Remarks

See ISourcesRowset::GetSourcesRowset in the *OLE DB Programmer's Reference* for more information.

See Also

Reference

CEnumeratorAccessor Class

Other Resources

CEnumeratorAccessor Members

CManualAccessor Class

Represents an accessor type designed for advanced use.

```
class CManualAccessor : public CAccessorBase
```

Remarks

Using **CManualAccessor**, you can specify the parameter and output column binding by run-time function calls.

Requirements

Header: atldbcli.h

See Also

Tasks

DBViewer Sample: Database Browser

Reference

CAccessor Class

CDynamicAccessor Class

CDynamicParameterAccessor Class

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

Other Resources

CManualAccessor Members

CManualAccessor Members

Methods

AddBindEntry	Adds a bind entry to the output columns.
AddParameterEntry	Adds a parameter entry to the parameter accessor.
CreateAccessor	Allocates memory for the column bind structures and initializes the column data members.
CreateParameterAccessor	Allocates memory for the parameter bind structures and initializes the parameter data members.

See Also

Reference

CManualAccessor Class

CManualAccessor Methods

For information about the methods in **CManualAccessor**, see CManualAccessor Members.

CManualAccessor::AddBindEntry

Adds a bind entry to the output columns.

```
void AddBindEntry(
   DBORDINAL nOrdinal,
   DBTYPE wType,
   DBLENGTH nColumnSize,
   void* pData,
   void* pLength = NULL,
   void* pStatus = NULL
) throw ( );
```

Parameters

See DBBINDING in the *OLE DB Programmer's Reference*.

nOrdinal

[in] Column number.

wType

[in] Data type.

nColumnSize

[in] Column size in bytes.

pData

[in] A pointer to the column data stored in the buffer.

pLength

[in] A pointer to the field length, if required.

pStatus

[in] A pointer to the variable to be bound to the column status, if required.

Remarks

To use this function, you must first call CreateAccessor. You cannot add more entries than the number of columns specified in **CreateAccessor**.

See Also

Tasks

DBViewer Sample: Database Browser

Reference

CManualAccessor Class

Other Resources

CManualAccessor Members

CManualAccessor::AddParameterEntry

Adds a parameter entry to the parameter entry structures.

```
void AddParameterEntry(
   DBORDINAL nOrdinal,
   DBTYPE wType,
   DBLENGTH nColumnSize,
   void* pData,
   void* pLength = NULL,
   void* pStatus = NULL,
   DBPARAMIO eParamIO = DBPARAMIO_INPUT
) throw ( );
```

Parameters

See DBBINDING in the *OLE DB Programmer's Reference*.

nOrdinal

[in] Parameter number.

wType

[in] Data type.

nColumnSize

[in] Column size in bytes.

pData

[in] A pointer to the column data stored in the buffer.

pLength

[in] A pointer to the field length, if required.

pStatus

[in] A pointer to the variable to be bound to the column status, if required.

eParamIO

[in] Specifies whether the parameter with which the binding is associated is an input, input/output, or output parameter.

Remarks

To use this function, you must first call CreateParameterAccessor.

See Also

Tasks

DBViewer Sample: Database Browser

Reference

CManualAccessor Class

CManualAccessor::AddBindEntry

Other Resources

CManualAccessor Members

CManualAccessor::CreateAccessor

Allocates memory for the column bind structures and initializes the column data members.

```
HRESULT CreateAccessor(
   int nBindEntries,
   void* pBuffer,
   DBLENGTH nBufferSize
) throw( );
```

Parameters

nBindEntries

[in] Number of columns. This number should match the number of calls to the CManualAccessor::AddBindEntry function.

pBuffer

[in] A pointer to the buffer where the output columns are stored.

nBufferSize

[in] The size of the buffer in bytes.

Return Value

One of the standard **HRESULT** values.

Remarks

Call this function before you call the **CManualAccessor::AddBindEntry** function.

See Also

Tasks

DBViewer Sample: Database Browser

Reference

CManualAccessor Class

Other Resources

CManualAccessor Members

CManualAccessor::CreateParameterAccessor

Allocates memory for the parameter bind structures and initializes the parameter data members.

```
HRESULT CreateParameterAccessor(
   int nBindEntries,
   void* pBuffer,
   DBLENGTH nBufferSize
) throw( );
```

Parameters

nBindEntries

[in] Number of columns.

pBuffer

[in] A pointer to the buffer where the input columns are stored.

nBufferSize

[in] The size of the buffer in bytes.

Return Value

One of the standard **HRESULT** values.

Remarks

You must call this function before calling AddParameterEntry.

See Also

Reference

CManualAccessor Class
CManualAccessor::CreateAccessor
CManualAccessor::AddParameterEntry

Other Resources

CManualAccessor Members

CMultipleResults Class

If you want a command to handle multiple result sets, use **CMultipleResults** for the **CCommand** template argument *TMultiple*.

```
class CMultipleResults
```

Remarks

To handle multiple result sets, CCommand must inherit from this class.

Requirements

Header: atldbcli

See Also

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

CNoAccessor Class

Can be used as a template argument (*TAccessor*) for template classes, such as **CCommand** and **CTable**, that require an accessor class argument.

```
class CNoAccessor
```

Remarks

Use **CNoAccessor** as a template argument when you do not want the class to support parameters or output columns.

CNoAccessor implements the following stub methods, each of which correspond to other accessor class methods:

- **BindColumns** - Binds columns to accessors.
- **BindParameters** - Binds the created parameters to columns.
- **Bind** - Creates bindings.
- **Close** - Closes the accessor.
- **ReleaseAccessors** - Releases the accessors created by the class.
- **FreeRecordMemory** - Frees any columns in the current record that need to be freed.
- **GetColumnInfo** - Gets column information from the opened rowset.
- **GetNumAccessors** - Retrieves the number of accessors created by the class.
- **IsAutoAccessor** - Returns true if data is automatically retrieved for the accessor during a Move operation.
- **GetHAccessor** - Retrieves the accessor handle of a specified accessor.
- **GetBuffer** - Retrieves the pointer to the bookmark buffer.
- **NoBindOnNullRowset** - Prevents data binding on empty rowsets.

Requirements

Header: atldbcli.h

See Also

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

CNoMultipleResults Class

Used as a template argument (*TMultiple*) to CCommand to create an optimized command that handles a single result set.

```
class CNoMultipleResults
```

Remarks

If you want a command to handle multiple result sets, use CMultipleResults instead.

Requirements

Header: atldbcli.h

See Also

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

CNoRowset Class

Can be used as a template argument (*TRowset*) for CCommand or CTable.

```
template <class TAccessor = CAccessorBase>
class CNoRowset
```

Parameters

TAccessor

An accessor class. The default is **CAccessorBase**.

Remarks

Use **CNoRowset** as a template argument if the command does not return a rowset.

CNoRowset implements the following stub methods, each of which correspond to other accessor class methods:

- **BindFinished** - Indicates when binding is complete (returns **S_OK**).
- **Close** - Releases rows and the current IRowset interface.
- **GetIID** - Retrieves the interface ID of a connection point.
- **GetInterface** - Retrieves an interface.
- **GetInterfacePtr** - Retrieves an encapsulated interface pointer.
- **SetAccessor** - Sets a pointer to the accessor.
- **SetupOptionalRowsetInterfaces** - Sets up optional interfaces for the rowset.

Requirements

Header: atldbcli.h

See Also

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

CRestrictions Class

A generic class that allows you to specify restrictions for schema rowsets.

```
template <
   class T,
   short nRestrictions,
   const GUID* pguid
>
class CRestrictions : public CSchemaRowset <
      T,
      nRestrictions
>
```

Parameters

T

The class used for the accessor.

nRestrictions

The number of restriction columns for the schema rowset.

pguid

A pointer to the GUID for the schema.

Requirements

Header: atldbsch.h

See Also

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

Other Resources

CRestrictions Members

CRestrictions Members

Operations

Open	Returns a result set according to the user-supplied restrictions.

See Also

Reference

CRestrictions Class

Schema Rowset Classes and Typedef Classes

CRestrictions Methods

For information about the methods in **CRestrictions**, see CRestrictions Members.

CRestrictions::Open

Returns a result set according to the user-supplied restrictions.

```
HRESULT Open(
   const CSession& session,
   LPCTSTR lpszParam 1 = NULL,
   LPCTSTR lpszParam 2 = NULL,
   LPCTSTR lpszParam 3 = NULL,
   LPCTSTR lpszParam 4 = NULL,
   LPCTSTR lpszParam 5 = NULL,
   LPCTSTR lpszParam 6 = NULL,
   LPCTSTR lpszParam 7 = NULL,
   bool bBind = true
);
```

Parameters

session

[in] Specifies an existing session object used to connect to the data source.

lpszParam

[in] Specifies the restrictions on the schema rowset.

bBind

[in] Specifies whether to bind the column map automatically. The default is **true**, which causes the column map to be bound automatically. Setting *bBind* to **false** prevents the automatic binding of the column map so that you can bind manually. (Manual binding is of particular interest to OLAP users.)

Return Value

One of the standard **HRESULT** values.

Remarks

You can specify a maximum of seven restrictions on a schema rowset.

See IDBSchemaRowset for information about the defined restrictions on each schema rowset.

See Also

Reference

CRestrictions Class

Schema Rowset Classes and Typedef Classes

Other Resources

CRestrictions Members

CRowset Class

Encapsulates an OLE DB rowset object and several related interfaces and provides manipulation methods for rowset data.

```
template <class TAccessor = CAccessorBase>
class CRowset
```

Parameters

TAccessor

An accessor class. The default is **CAccessorBase**.

Remarks

In OLE DB, a rowset is the object through which a program sets and retrieves data.

This class is not meant to be instantiated but rather passed as a template parameter to **CTable** or **CCommand** (**CRowset** is the default).

Example

See the ShowImage Sample.

Requirements

Header: atldbcli.h

See Also

Tasks

DBViewer Sample: Database Browser
MultiRead Sample: Reads Database Table Using Multiple Threads
MultiRead Attributes Sample: Reads Database Table Using Multiple Threads

Concepts

OLE DB Consumer Templates (C++)
OLE DB Consumer Templates Reference

Other Resources

CRowset Members

CRowset Members

Methods

AddRefRows	Increments the reference count associated with the current row.
Close	Releases rows and the current **IRowset** interface.
Compare	Compares two bookmarks using IRowsetLocate::Compare.
CRowset	Creates a new **CRowset** object and (optionally) associates it with an **IRowset** interface supplied as a parameter.
Delete	Deletes rows from the rowset using IRowsetChange:DeleteRows.
FindNextRow	Finds the next matching row after the specified bookmark.
GetApproximatePosition	Returns the approximate position of a row corresponding to a bookmark.
GetData	Retrieves data from the rowset's copy of the row.
GetDataHere	Retrieves data from the specified buffer.
GetOriginalData	Retrieves the data most recently fetched from or transmitted to the data source, ignoring pending ch anges.
GetRowStatus	Returns the status of all rows.
Insert	Creates and inserts a new row using IRowsetChange:InsertRow.
IsSameRow	Compares the specified row with the current row.
MoveFirst	Repositions the next-fetch location to the initial position.
MoveLast	Moves to the last record.
MoveNext	Fetches data from the next sequential row or a specified number of positions beyond the next row.
MovePrev	Moves to the previous record.
MoveToBookmark	Fetches the row marked by a bookmark or the row at a specified offset from that bookmark.
MoveToRatio	Fetches rows starting from a fractional position in the rowset.
ReleaseRows	Calls IRowset::ReleaseRows to release the current row handle.
SetData	Sets data values in one or more columns of a row using IRowsetChange:SetData.
Undo	Undoes any changes made to a row since the last fetch or Update.
Update	Transmits any pending changes made to the current row since the last fetch or update.

UpdateAll	Transmits any pending changes made to all rows since the last fetch or update.

See Also

Reference

CRowset Class

CRowset Methods

For information about the methods in **CRowset**, see CRowset Members.

CRowset::AddRefRows

Calls IRowset::AddRefRows to increment (by one) the reference count associated with the current row handle.

```
HRESULT AddRefRows( ) throw( );
```

Return Value

A standard **HRESULT**.

Remarks

This method increments the reference count for the current row handle. Call ReleaseRows to decrement the count. Rows returned by the move methods have a reference count of one.

See Also

Reference

CRowset Class

CRowset::ReleaseRows

Other Resources

CRowset Members

CRowset::Close

Releases rows and the current IRowset interface.

```
void Close( ) throw( );
```

Remarks

This method releases all rows currently in the rowset.

See Also

Reference

CRowset Class

Other Resources

CRowset Members

CRowset::Compare

Compares two bookmarks using IRowsetLocate::Compare.

```
HRESULT Compare(
   const CBookmarkBase& bookmark1,
   const CBookmarkBase& bookmark2,
   DBCOMPARE* pComparison
) const throw( );
```

Parameters

Bookmark1

[in] The first bookmark to compare.

Bookmark2

[in] The second bookmark to compare.

pComparison

[out] A pointer to the result of the comparison.

Return Value

A standard **HRESULT**.

Remarks

This method requires the optional interface **IRowsetLocate**, which might not be supported on all providers; if this is the case, the method returns **E_NOINTERFACE**. You must also set **DBPROP_IRowsetLocate** to **VARIANT_TRUE** before calling **Open** on the table or command containing the rowset.

For information about using bookmarks in consumers, see Using Bookmarks.

See Also

Reference

CRowset Class

Other Resources

CRowset Members

CRowset::CRowset

Creates a new **CRowset** object and (optionally) associates it with an IRowset interface supplied as a parameter.

```
CRowset( );
CRowset(
   IRowset* pRowset
);
```

Parameters

pRowset

[in] A pointer to an **IRowset** interface to be associated with this class.

See Also

Reference

CRowset Class

Other Resources

CRowset Members

CRowset::Delete

Calls IRowsetChange::DeleteRows to delete the current row from the rowset.

```
HRESULT Delete( ) const throw( );
```

Return Value

A standard **HRESULT**.

See Also

Reference

CRowset Class

Other Resources

CRowset Members

CRowset::FindNextRow

Finds the next matching row after the specified bookmark.

```
HRESULT FindNextRow(
   DBCOMPAREOP op,
   BYTE* pData,
   DBTYPE wType,
   DBLENGTH nLength,
   BYTE bPrecision,
   BYTE bScale,
   BOOL bSkipCurrent = TRUE,
   CBookmarkBase* pBookmark = NULL
) throw( );
```

Parameters

op

[in] The operation to use in comparing row values. For values, see IRowsetFind::FindNextRow.

pData

[in] A pointer to the value to be matched.

wType

[in] Indicates the data type of the value part of the buffer. For information about type indicators, see Data Types in the *OLE DB Programmer's Reference* in the *Platform SDK*.

nLength

[in] The length, in bytes, of the consumer data structure allocated for the data value. For details, see the description of **cbMaxLen** in DBBINDING Structures in the *OLE DB Programmer's Reference*.

bPrecision

[in] The maximum precision used when getting data. Used only if *wType* is **DBTYPE_NUMERIC**. For more information, see Conversions involving DBTYPE_NUMERIC or DBTYPE_DECIMAL in the *OLE DB Programmer's Reference*.

bScale

[in] The scale used when getting data. Used only if *wType* is **DBTYPE_NUMERIC** or **DBTYPE_DECIMAL**. For more information, see Conversions involving DBTYPE_NUMERIC or DBTYPE_DECIMAL in the *OLE DB Programmer's Reference*.

bSkipCurrent

[in] The number of rows from the bookmark at which to start a search.

pBookmark

[in] The bookmark for position at which to start a search.

Return Value

A standard **HRESULT**.

Remarks

This method requires the optional interface **IRowsetFind**, which might not be supported on all providers; if this is the case, the method returns **E_NOINTERFACE**. You must also set **DBPROP_IRowsetFind** to **VARIANT_TRUE** before calling **Open** on the table or command containing the rowset.

For information about using bookmarks in consumers, see Using Bookmarks.

See Also

Reference

CRowset Class

DBBINDING Structures

Other Resources

CRowset Members

CRowset::GetApproximatePosition

Returns the approximate position of a row corresponding to a bookmark.

```
HRESULT GetApproximatePosition(
   const CBookmarkBase* pBookmark,
   DBCOUNTITEM* pPosition,
   DBCOUNTITEM* pcRows
) throw( );
```

Parameters

pBookmark

[in] A pointer to a bookmark that identifies the row whose position is to be found. **NULL** if only the row count is required.

pPosition

[out] A pointer to the location where **GetApproximatePosition** returns the position of the row. **NULL** if the position is not required.

pcRows

[out] A pointer to the location where **GetApproximatePosition** returns the total number of rows. **NULL** if the row count is not required.

Return Value

A standard **HRESULT**.

Remarks

This method requires the optional interface **IRowsetScroll**, which might not be supported on all providers; if this is the case, the method returns **E_NOINTERFACE**. You must also set **DBPROP_IRowsetScroll** to **VARIANT_TRUE** before calling **Open** on the table or command containing the rowset.

For information about using bookmarks in consumers, see Using Bookmarks.

See Also

Reference

CRowset Class

IRowsetScroll::GetApproximatePosition

Other Resources

CRowset Members

CRowset::GetData

Retrieves data from the rowset's copy of the row.

```
HRESULT GetData( ) throw( );
HRESULT GetData(
   int nAccessor
) throw( );
```

Parameters

nAccessor

[in] The (zero-offset) index number of the accessor to use for accessing the data.

Return Value

A standard **HRESULT**.

Remarks

If you specify an accessor that is not an autoaccessor in BEGIN_ACCESSOR, use this method to explicitly get the data by passing the accessor number.

See Also

Reference

CRowset Class

Other Resources

CRowset Members

CRowset::GetDataHere

Retrieves data from the current row and places it into the specified buffer.

```
HRESULT GetDataHere(
   int nAccessor,
   void* pBuffer
) throw( );
```

Parameters

nAccessor

[in] The index number of the accessor to use for accessing the data.

pBuffer

[out] A buffer into which to place the data for the current record.

Return Value

A standard **HRESULT**.

Remarks

For an example of how to use this function, see the MultiRead sample.

See Also

Reference

CRowset Class

CRowset::GetData

Other Resources

CRowset Members

CRowset::GetOriginalData

Calls **IRowsetUpdate::GetOriginalData** to retrieve the data most recently fetched from or transmitted to the data source.

```
HRESULT GetOriginalData( ) throw( );
```

Return Value

A standard **HRESULT**.

Remarks

This method retrieves the data most recently fetched from or transmitted to the data source; it does not retrieve values based on pending changes.

This method requires the optional interface **IRowsetUpdate**, which might not be supported on all providers; if this is the case, the method returns **E_NOINTERFACE**. You must also set **DBPROP_IRowsetUpdate** to **VARIANT_TRUE** before calling **Open** on the table or command containing the rowset.

See Also

Reference

CRowset Class

IRowsetUpdate::GetOriginalData

Other Resources

CRowset Members

CRowset::GetRowStatus

Returns the status of all rows.

```
HRESULT GetRowStatus(
   DBPENDINGSTATUS* pStatus
) const throw( );
```

Parameters

pStatus

[out] A pointer to a location where **GetRowStatus** returns the status value. See DBPENDINGSTATUS in the OLE DB Programmer's Reference.

Return Value

A standard **HRESULT**.

Remarks

This method requires the optional interface **IRowsetUpdate**, which might not be supported on all providers; if this is the case, the method returns **E_NOINTERFACE**. You must also set **DBPROP_IRowsetUpdate** to **VARIANT_TRUE** before calling **Open** on the table or command containing the rowset.

See Also

Reference

CRowset Class

IRowsetUpdate::GetRowStatus

Other Resources

CRowset Members

CRowset::Insert

Creates and initializes a new row using data from the accessor.

```
HRESULT Insert(
   int nAccessor = 0,
   bool bGetHRow = false
) throw( );
```

Parameters

nAccessor

[in] The number of the accessor to use for inserting the data.

bGetHRow

[in] Indicates whether the handle for the inserted row is retrieved.

Return Value

A standard **HRESULT**.

Remarks

This method requires the optional interface **IRowsetChange**, which might not be supported on all providers; if this is the case, the method returns **E_NOINTERFACE**. You must also set **DBPROP_IRowsetChange** to **VARIANT_TRUE** before calling **Open** on the table or command containing the rowset.

Insert might fail if one or more columns is not writable. Modify your cursor map to correct this.

Example

The following example shows how to access a data source through a rowset and then insert a string using a table in that rowset.

First, create a table class by inserting a New ATL Object into your project. For example, right-click the project in the Workspace pane and select **New ATL Object**. From the **Data Access** category, select **Consumer**. Create a consumer object of type **Table**. (Selecting **Table** creates a rowset directly from the table; selecting **Command** creates a rowset through a SQL command.) Select a data source, specifying a table through which to access that data source. If you call your consumer object **CProductTable**, you would then implement your insertion code as follows:

```
// Access the rowset using your wizard-generated class, CProductTable
CProductTable product;

// Insert a product
_tcscpy( product.m_szName, _T( "My Inserted Product" ) );
product.m_dwNameLength = 19;              // bind length
product.m_dwNameStatus = DBSTATUS_S_OK;   // bind status

product.m_nUnitsInStock = 9999;
product.m_dwUnitsInStockStatus = DBSTATUS_S_OK;
// product.m_dwUnitsInStockLength is fixed-length type so it's ignored

HRESULT hr = product.Insert( );

if ( hr !=S_OK )
{
   // Trace out the table
   ATLTRACE( _T( "Insert failed: 0x%X\n" ), hr );
}
```

See Also

Reference

CRowset Class

Other Resources

CRowset Members

CRowset::IsSameRow

Compares the specified row with the current row.

```
HRESULT IsSameRow(
   HROW hRow
) const throw( );
```

Parameters

hRow

[in] A handle to the row to compare to the current row.

Return Value

A standard **HRESULT**. **S_OK** indicates the rows are the same. For other values, see IRowsetIndentity::IsSameRow in the *OLE DB Programmer's Reference* in the *Platform SDK*.

See Also

Reference

CRowset Class

Other Resources

CRowset Members

CRowset::MoveFirst

Moves the cursor to the initial position and retrieves the initial row.

```
HRESULT MoveFirst( ) throw( );
```

Return Value

A standard **HRESULT**.

Remarks

Calls IRowset::RestartPosition to reposition the next-fetch location to the initial position (the position that was the next-fetch location when the rowset was created) and retrieves the initial row.

Example

See the ShowImage Sample.

See Also

Reference

CRowset Class
CRowset::MoveNext
CRowset::MoveToBookmark
CRowset::MovePrev
CRowset::MoveLast
IRowset::RestartPosition

Other Resources

CRowset Members

CRowset::MoveLast

Moves the cursor to the last row.

```
HRESULT MoveLast( ) throw( );
```

Return Value

A standard **HRESULT**.

Remarks

Calls IRowset::RestartPosition to reposition the next-fetch location to the last position and retrieves the last row.

This method requires that you set **DBPROP_CANSCROLLBACKWARDS** to **VARIANT_TRUE** before calling **Open** on the table or command containing the rowset. (For better performance, you might also set **DBPROP_QUICKRESTART** to **VARIANT_TRUE**.)

See Also

Reference

CRowset Class
CRowset::MoveNext
IRowset::RestartPosition
CRowset::MovePrev

Other Resources

CRowset Members

CRowset::MoveNext

Moves the cursor to the next record.

```
HRESULT MoveNext( ) throw( );
HRESULT MoveNext(
   LONG lSkip,
   bool bForward = true
) throw( );
```

Parameters

lSkip

[in] The number of rows to skip before fetching.

bForward

[in] Pass **true** to move forward to the next record, **false** to move backward.

Return Value

A standard **HRESULT**. When the end of the rowset has been reached, returns **DB_S_ENDOFROWSET**.

Remarks

Fetches the next sequential row from the **CRowset** object, remembering the previous position. Optionally, you can choose to skip ahead *lSkip* rows or move backward.

This method requires that you set the following properties before calling **Open** on the table or command containing the rowset:

- **DBPROP_CANSCROLLBACKWARDS** must be **VARIANT_TRUE** if *lSkip* < 0
- **DBPROP_CANFETCHBACKWARDS** must be **VARIANT_TRUE** if *bForward* = false

Otherwise (if *lSkip* >= 0 and *bForward* = true), you do not need to set any additional properties.

See Also

Reference

CRowset Class
CRowset::MoveFirst
CRowset::MoveToBookmark
CRowset::MovePrev
CRowset::MoveLast

Other Resources

CRowset Members

CRowset::MovePrev

Moves the cursor to the previous record.

```
HRESULT MovePrev( ) throw( );
```

Return Value

A standard **HRESULT**.

Remarks

This method requires that you set either **DBPROP_CANFETCHBACKWARDS** or **DBPROP_CANSCROLLBACKWARDS** to **VARIANT_TRUE** before calling **Open** on the table or command containing the rowset.

See Also

Reference

CRowset Class
CRowset::MoveNext
CRowset::MoveToBookmark
CRowset::MoveLast

Other Resources

CRowset Members

CRowset::MoveToBookmark

Fetches the row marked by a bookmark or the row at a specified offset (*lSkip*) from that bookmark.

```
HRESULT MoveToBookmark(
   const CBookmarkBase& bookmark,
   LONG lSkip = 0
) throw( );
```

Parameters

bookmark

[in] A bookmark marking the location from which you want to fetch data.

lSkip

[in] The number count of rows from the bookmark to the target row. If *lSkip* is zero, the first row fetched is the bookmarked row. If *lSkip* is 1, the first row fetched is the row after the bookmarked row. If *lSkip* is –1, the first row fetched is the row before the bookmarked row.

Return Value

A standard **HRESULT**.

Remarks

This method requires the optional interface **IRowsetLocate**, which might not be supported on all providers; if this is the case, the method returns **E_NOINTERFACE**. You must also set **DBPROP_IRowsetLocate** to **VARIANT_TRUE** and set **DBPROP_CANFETCHBACKWARDS** to **VARIANT_TRUE** before calling **Open** on the table or command containing the rowset.

For information about using bookmarks in consumers, see Using Bookmarks.

See Also

Reference

CRowset Class
CRowset::MoveNext
CRowset::MoveFirst
IRowsetLocate::GetRowsAt
CRowset::MovePrev
CRowset::MoveLast

Other Resources

CRowset Members

CRowset::MoveToRatio

Fetches rows starting from a fractional position in the rowset.

```
HRESULT MoveToRatio(
   DBCOUNTITEM nNumerator,
   DBCOUNTITEM nDenominator,
   bool bForward = true
) throw( );
```

Parameters

nNumerator

[in] The numerator used to determine the fractional positional from which to fetch data.

nDenominator

[in] The denominator used to determine the fractional positional from which to fetch data.

bForward

[in] Indicates whether to move forward or backward. The default is forward.

Return Value

A standard **HRESULT**.

Remarks

MoveToRatio fetches rows according roughly to the following formula:

```
( nNumerator *  RowsetSize ) / nDenominator
```

where `RowsetSize` is the size of the rowset, measured in rows. The accuracy of this formula depends on the specific provider. For details, see IRowsetScroll::GetRowsAtRatio.

This method requires the optional interface **IRowsetScroll**, which might not be supported on all providers; if this is the case, the method returns **E_NOINTERFACE**. You must also set **DBPROP_IRowsetScroll** to **VARIANT_TRUE** before calling **Open** on the table or command containing the rowset.

See Also

Reference

CRowset Class

Other Resources

CRowset Members

CRowset::ReleaseRows

Calls IRowset::ReleaseRows to release the current row handle.

```
HRESULT ReleaseRows( ) throw( );
```

Return Value

A standard **HRESULT**.

See Also

Reference

CRowset Class

Other Resources

CRowset Members

CRowset::SetData

Sets data values in one or more columns of a row.

```
HRESULT SetData( ) const throw( );
HRESULT SetData(
   int nAccessor
) const throw( );
```

Parameters

nAccessor

[in] The number of the accessor to use for accessing the data.

Return Value

A standard **HRESULT**.

Remarks

For the **SetData** form that accepts no arguments, all accessors are used for updating. You typically call **SetData** to set data values in columns in a row, then call Update to transmit those changes.

This method requires the optional interface **IRowsetChange**, which might not be supported on all providers; if this is the case, the method returns **E_NOINTERFACE**. You must also set **DBPROP_IRowsetChange** to **VARIANT_TRUE** before calling **Open** on the table or command containing the rowset.

The setting operation might fail if one or more columns is not writable. Modify your cursor map to correct this.

See Also

Reference

CRowset Class

CRowset::Update

Other Resources

CRowset Members

CRowset::Undo

Undoes any changes made to a row since the last fetch or Update.

```
HRESULT Undo(
   DBCOUNTITEM* pcRows = NULL,
   HROW* phRow = NULL,
   DBROWSTATUS* pStatus = NULL
) throw( );
```

Parameters

pcRows

[out] A pointer to the location where **Undo** returns the number of rows it attempted to undo if required.

phRow

[out] A pointer to the location where **Undo** returns an array of handles to all rows it attempted to undo if required.

pStatus

[out] A pointer to the location where **Undo** returns the row status value. No status is returned if *pStatus* is null.

Return Value

A standard **HRESULT**.

Remarks

This method requires the optional interface **IRowsetUpdate**, which might not be supported on all providers; if this is the case, the method returns **E_NOINTERFACE**. You must also set **DBPROP_IRowsetUpdate** to **VARIANT_TRUE** before calling **Open** on the table or command containing the rowset.

See Also

Reference

CRowset Class

IRowsetUpdate::Undo

Other Resources

CRowset Members

CRowset::Update

Transmits any pending changes made to the current row since the last fetch or **Update** call on it.

```
HRESULT Update(
   DBCOUNTITEM* pcRows = NULL,
   HROW* phRow = NULL,
   DBROWSTATUS* pStatus = NULL
) throw( );
```

Parameters

pcRows

[out] A pointer to the location where **Update** returns the number of rows it attempted to update, if required.

phRow

[out] A pointer to the location where **Update** returns the handle of the row it attempted to update. No handle is returned if *phRow* is null.

pStatus

[out] A pointer to the location where **Update** returns the row status value. No status is returned if *pStatus* is null.

Return Value

A standard **HRESULT**.

Remarks

Transmits any pending changes made to the current row since that row was last fetched or updated (using **Update** or UpdateAll). You typically call SetData to set data values in columns in a row, and then call **Update** to transmit those changes.

This method requires the optional interface **IRowsetUpdate**, which might not be supported on all providers; if this is the case, the method returns **E_NOINTERFACE**. You must also set **DBPROP_IRowsetUpdate** to **VARIANT_TRUE** before calling **Open** on the table or command containing the rowset.

See Also

Reference

CRowset Class
IRowsetUpdate::Update
CRowset::UpdateAll
CRowset::SetData

Other Resources

CRowset Members

CRowset::UpdateAll

Transmits any pending changes made to all rows since the last fetch or **Update** call on it.

```
HRESULT UpdateAll(
   DBCOUNTITEM* pcRows = NULL,
   HROW** pphRow = NULL,
   DBROWSTATUS** ppStatus = NULL
) throw( );
```

Parameters

pcRows

[out] A pointer to the location where **UpdateAll** returns the number of rows it attempted to update, if required.

pphRow

[out] A pointer to memory in which **UpdateAll** returns the handle of the row it attempted to update. No handle is returned if *pphRow* is null.

ppStatus

[out] A pointer to the location where **Update** returns the row status value. No status is returned if *ppStatus* is null.

Remarks

Transmits any pending changes made to all rows since those rows were last fetched or updated using Update or **UpdateAll**. **UpdateAll** will update every row that has been modified, regardless of whether you still have the handle for them (see *pphRow*) or not.

For example, if you used **Insert** to insert five rows in a rowset, you could either call **Update** five times or call **UpdateAll** once to update them all.

This method requires the optional interface **IRowsetUpdate**, which might not be supported on all providers; if this is the case, the method returns **E_NOINTERFACE**. You must also set **DBPROP_IRowsetUpdate** to **VARIANT_TRUE** before calling **Open** on the table or command containing the rowset.

Return Value

A standard **HRESULT**.

See Also

Reference

CRowset Class
IRowsetUpdate::Update
CRowset::SetData
CRowset::Update

Other Resources

CRowset Members

CSession Class

Represents a single database access session.

```
class CSession
```

Remarks

One or more sessions can be associated with each provider connection (data source), which is represented by a CDataSource object. To create a new **CSession** for a **CDataSource**, call CSession::Open. To begin a database transaction, **CSession** provides the **StartTransaction** method. Once a transaction is started, you can commit to it using the **Commit** method, or cancel it using the **Abort** method.

Requirements

Header: atldbcli.h

Example

See the ConfirmUser Sample and the ShowImage Sample.

See Also

Tasks

CatDB Sample: Data Source Schema Browser

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

Other Resources

CSession Members

CSession Members

Methods

Abort	Cancels (terminates) the transaction.
Close	Closes the session.
Commit	Commits the transaction.
GetTransactionInfo	Returns information regarding a transaction.
Open	Opens a new session for the data source object.
StartTransaction	Begins a new transaction for this session.

See Also

Reference

CSession Class

CSession Methods

For information about the methods in **CSession**, see CSession Members.

CSession::Abort

Terminates the transaction.

```
HRESULT Abort(
   BOID* pboidReason = NULL,
   BOOL bRetaining = FALSE,
   BOOL bAsync = FALSE
) const throw( );
```

Parameters

See ITransaction::Abort in the *OLE DB Programmer's Reference*.

Return Value

A standard **HRESULT**.

See Also

Reference

CSession Class

Other Resources

CSession Members

CSession::Close

Closes the session, which was opened by CSession::Open.

```
void Close( ) throw( );
```

Remarks

Releases the **m_spOpenRowset** pointer.

See Also

Reference

CSession Class

Other Resources

CSession Members

CSession::Commit

Commits the transaction.

```
HRESULT Commit(
   BOOL bRetaining = FALSE,
   DWORD grfTC = XACTTC_SYNC,
   DWORD grfRM = 0
) const throw( );
```

Parameters

See ITransaction::Commit in the *OLE DB Programmer's Reference*.

Return Value

A standard **HRESULT**.

Remarks

For more information, see ITransaction::Commit.

See Also

Reference

CSession Class

Other Resources

CSession Members

CSession::GetTransactionInfo

Returns information regarding a transaction.

```
HRESULT GetTransactionInfo(
   XACTTRANSINFO* pInfo
) const throw( );
```

Parameters

See ITransaction::GetTransactionInfo in the *OLE DB Programmer's Reference*.

Return Value

A standard **HRESULT**.

Remarks

For more information, see ITransaction::GetTransactionInfo in the *OLE DB Programmer's Reference*.

See Also

Reference

CSession Class

Other Resources

CSession Members

CSession::Open

Opens a new session for the data source object.

```
HRESULT Open(
   const CDataSource& ds,
   DBPROPSET *pPropSet = NULL,
   ULONG ulPropSets = 0
) throw( );
```

Parameters

ds

[in] The data source for which the session is to be opened.

pPropSet

[in] A pointer to an array of DBPROPSET structures containing properties and values to be set. See Property Sets and Property Groups in the *OLE DB Programmer's Reference* in the Platform SDK.

ulPropSets

[in] The number of DBPROPSET structures passed in the *pPropSet* argument.

Return Value

A standard **HRESULT**.

Remarks

You must open the data source object using CDataSource::Open before passing it to **CSession::Open**.

Example

See the ConfirmUser Sample and the ShowImage Sample.

See Also

Reference

CSession Class

Other Resources

CSession Members

CSession::StartTransaction

Begins a new transaction for this session.

```
HRESULT StartTransaction(
   ISOLEVEL isoLevel = ISOLATIONLEVEL_READCOMMITTED,
   ULONG isoFlags = 0,
   ITransactionOptions* pOtherOptions = NULL,
   ULONG* pulTransactionLevel = NULL
) const throw( );
```

Parameters

See ITransactionLocal::StartTransaction in the *OLE DB Programmer's Reference*.

Return Value

A standard **HRESULT**.

Remarks

For more information, see ITransactionLocal::StartTransaction in the *OLE DB Programmer's Reference*.

See Also

Reference

CSession Class

Other Resources

CSession Members

CStreamRowset Class

Used in a **CCommand** or **CTable** declaration.

```
template <class TAccessor = CAccessorBase>
class CStreamRowset
```

Parameters

TAccessor

An accessor class.

Remarks

Use **CStreamRowset** in your **CCommand** or **CTable** declaration, for example:

```
CCommand< CAccessor<CMyAccessor>, CStreamRowset > myCmd;
```

or

```
CCommand< CNoAccessor, CStreamRowset > myCmd;
```

ICommand::Execute returns an **ISequentialStream** pointer, which is stored in **m_spStream**. You then use the **Read** method to retrieve the (Unicode string) data in XML format. For example:

```
myCmd.m_spStream->Read()
```

SQL Server 2000 performs the XML formatting, and will return all columns and all rows of the rowset as one XML string.

Note
This feature works with SQL Server 2000 only.

Requirements

Header: atldbcli.h

See Also

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

Other Resources

CDataConnection Members

CStreamRowset Members

Methods

CStreamRowset	Constructor. Instantiates and initializes the **CStreamRowset** object.
Close	Releases the ISequentialStream interface pointer in the class.

See Also

Reference

CStreamRowset Class

CStreamRowset Methods

For information about the methods in **CStreamRowset**, see CStreamRowset Members.

CStreamRowset::Close

Releases the ISequentialStream interface pointer in the class.

```
void Close( );
```

See Also

Reference

CStreamRowset Class

Other Resources

CStreamRowset Members

CStreamRowset::CStreamRowset

Instantiates and initializes the **CStreamRowset** object.

```
CStreamRowset( );
```

See Also

Reference

CStreamRowset Class

Other Resources

CStreamRowset Members

CTable Class

Provides a means to directly access a simple rowset (one with no parameters).

```
template <
   class TAccessor = CNoAccessor,
   template <typename T> class TRowset = CRowset
>
class CTable :
   public CAccessorRowset <
            TAccessor,
            TRowset
   >
```

Parameters

TAccessor

An accessor class.

TRowset

A rowset class.

Remarks

See CCommand for information on how to execute a command to access a rowset.

Requirements

Header: atldbcli.h

See Also

Reference

IOpenRowset::OpenRowset

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates Reference

Other Resources

CTable Members

CTable Members

Methods

Open	Opens the table.

See Also

Reference

CTable Class

CTable Methods

For information about the methods in **CTable**, see CTable Members.

CTable::Open

Opens the table.

```
HRESULT Open(
   const CSession& session,
   LPCWSTR wszTableName,
   DBPROPSET* pPropSet = NULL,
   ULONG ulPropSets = 0
) throw ( );
HRESULT Open(
   const CSession& session,
   LPCSTR szTableName,
   DBPROPSET* pPropSet = NULL,
   ULONG ulPropSets = 0
) throw ( );
HRESULT Open(
   const CSession& session,
   DBID& dbid,
   DBPROPSET* pPropSet = NULL,
   ULONG ulPropSets = 0
) throw ( );
```

Parameters

session

[in] The session for which the table is opened.

wszTableName

[in] The name of the table to open, passed as a Unicode string.

szTableName

[in] The name of the table to open, passed as an ANSI string.

dbid

[in] The **DBID** of the table to open.

pPropSet

[in] A pointer to an array of DBPROPSET structures containing properties and values to be set. See Property Sets and Property Groups in the *OLE DB Programmer's Reference* in the Platform SDK. The default value of NULL specifies no properties.

ulPropSets

[in] The number of DBPROPSET structures passed in the *pPropSet* argument.

Return Value

A standard **HRESULT**.

Remarks

For more details, see IOpenRowset::OpenRowset in the *OLE DB Programmer's Reference*.

See Also

Reference

CTable Class

Other Resources

CTable Members

CXMLAccessor Class

Allows you to access data sources as string data when you have no knowledge of the data store's schema (underlying structure).

```
class CXMLAccessor : public CDynamicStringAccessorW
```

Remarks

However, **CXMLAccessor** differs from **CDynamicStringAccessorW** in that it converts all data accessed from the data store as XML-formatted (tagged) data. This is especially useful for output to XML-aware Web pages. The XML tag names will match the data store's column names as closely as possible.

Use **CDynamicAccessor** methods to obtain column information. You use this column information to create an accessor dynamically at run time.

The column information is stored in a buffer created and managed by this class. Obtain column information using GetXMLColumnData or obtain column data by rows using GetXMLRowData.

Example

```
#include "stdafx.h"

int main( int argc, char* argv[] )
{
    HRESULT hr = CoInitialize( NULL );

    CDataSource ds;
    CSession ss;

    CTable<CXMLAccessor> rs;

    // The following is an example initialization string:
    hr = ds.OpenFromInitializationString(L"Provider=SQLOLEDB.1;"
     L"Integrated Security=SSPI;Persist Security Info=False;"
     L"Initial Catalog=your_database_name;"
     L"Use Procedure for Prepare=1;Auto Translate=True;"
     L"Packet Size=4096;Workstation ID=LOGINNAME01;"
     L"Use Encryption for Data=False;Data Source=your_data_source;"
     L"Tag with column collation when possible=False");

    hr = ss.Open( ds );
    hr = rs.Open( ss, "your_table_name" );

    CStringW strColumnInfo;
    rs.GetXMLColumnData(strColumnInfo);
    printf_s( "%S\n", strColumnInfo );

    hr = rs.MoveFirst( );
    while( SUCCEEDED( hr ) && hr != DB_S_ENDOFROWSET )
    {
        CStringW strRowData;
        rs.GetXMLRowData( strRowData );
        printf_s( "%S\n", strRowData );
        hr = rs.MoveNext( );
    }

    ss.Close();
    ds.Close();
    CoUninitialize();
    return 0;
}
```

Requirements

Header: atldbcli.h

See Also

Reference

CAccessor Class
CDynamicAccessor Class
CDynamicParameterAccessor Class
CDynamicStringAccessor Class
CDynamicStringAccessorA Class
CDynamicStringAccessorW Class
CManualAccessor Class

Concepts

OLE DB Consumer Templates (C++)
OLE DB Consumer Templates Reference

Other Resources

CDynamicAccessor Members

CXMLAccessor Members

Methods

GetXMLColumnData	Retrieves the column information.
GetXMLRowData	Retrieves the entire contents of a table by rows.

See Also

Reference

CXMLAccessor Class

CXMLAccessor Methods

For information about the methods in **CXMLAccessor**, see CXMLAccessor Members.

CXMLAccessor::GetXMLColumnData

Retrieves the column type information of a table as XML-formatted string data, by column.

```
HRESULT GetXMLColumnData(
   CSimpleStringW& strOutput
) throw( );
```

Parameters

strOutput

[out] A reference to a string buffer containing the column type information to be retrieved. The string is formatted with XML tag names that match the data store's column names.

Return Value

One of the standard **HRESULT** values.

Remarks

The following shows how the column type information is formatted in XML. `type` specifies the column's data type. Note that the data types are based on OLE DB data types, not those of the database being accessed.

```
<columninfo>
   <column type = I2/> ColumnName
</columninfo>
```

See Also

Reference

CXMLAccessor Class

Other Resources

CXMLAccessor Members

CXMLAccessor::GetXMLRowData

Retrieves the entire contents of a table as XML-formatted string data, by row.

```
HRESULT GetXMLRowData(
   CSimpleStringW& strOutput,
   bool bAppend = false
) throw( );
```

Parameters

strOutput

[out] A reference to a buffer containing the table data to be retrieved. The data is formatted as string data with XML tag names that match the data store's column names.

bAppend

[in] A Boolean value specifying whether to append a string to the end of the output data.

Return Value

One of the standard **HRESULT** values.

Remarks

The following shows how the row data is formatted in XML. `DATA` below represents the row data. Use move methods to move to the desired row.

```
<row>
   <column name>DATA</column name>
</row>
```

See Also

Reference

CXMLAccessor Class

Other Resources

CXMLAccessor Members

IRowsetNotifyImpl Class

Implements and registers IRowsetNotify on the consumer (also known as the "sink") so that it can handle notifications.

```
class ATL_NO_VTABLE IRowsetNotifyImpl : public IRowsetNotify
```

Remarks

See Receiving Notifications about implementing the connection point interface on the consumer.

IRowsetNotifyImpl provides a dummy implementation for **IRowsetNotify**, with empty functions for the **IRowsetNotify** methods OnFieldChange, OnRowChange, and OnRowsetChange. If you inherit from this class when you implement an **IRowsetNotify** interface, you can implement only the methods you need. You also need to provide empty implementations for the other methods yourself.

Requirements

Header: atldbcli.h

See Also

Reference

IRowsetNotify

IRowsetNotifyCP Class

Concepts

OLE DB Consumer Templates (C++)

OLE DB Consumer Templates (C++)

IRowsetNotifyImpl Members

Methods

OnFieldChange	Notifies the consumer of any change to the value of a column.
OnRowChange	Notifies the consumer of the first change to a row or of any change that affects the entire row.
OnRowsetChange	Notifies the consumer of any change affecting the entire rowset.

See Also

Reference

IRowsetNotifyImpl Class

IRowsetNotifyImpl Methods

For information about the methods in **IRowsetNotifyImpl**, see IRowsetNotifyImpl Members.

IRowsetNotifyImpl::OnFieldChange

Notifies the consumer of any change to the value of a column.

```
STDMETHOD(OnFieldChange)(
      /* [in] */ IRowset* /* pRowset */,
      /* [in] */ HROW /* hRow */,
      /* [in] */ DBORDINAL /* cColumns */,
      /* [size_is][in] */ DBORDINAL /* rgColumns */ [] ,
      /* [in] */ DBREASON /* eReason */,
      /* [in] */ DBEVENTPHASE /* ePhase */,
      /* [in] */ BOOL /* fCantDeny */)
```

Parameters

See IRowsetNotify::OnFieldChange for parameter descriptions.

Return Value

See IRowsetNotify::OnFieldChange for return value descriptions.

Remarks

This method wraps the IRowsetNotify::OnFieldChange method. See that method's description in the OLE DB Programmer's Reference for details.

See Also

Reference

IRowsetNotifyImpl Class

IRowsetNotify::OnFieldChange

IRowsetNotifyImpl::OnRowChange

Notifies the consumer of the first change to a row or of any change that affects the entire row.

```
STDMETHOD(OnRowChange)(
      /* [in] */ IRowset* /* pRowset */,
      /* [in] */ DBCOUNTITEM /* cRows */,
      /* [size_is][in] */ const HROW /* rghRows*/ [] ,
      /* [in] */ DBREASON /* eReason */,
      /* [in] */ DBEVENTPHASE /* ePhase */,
      /* [in] */ BOOL /* fCantDeny */)
```

Parameters

See IRowsetNotify::OnRowChange for parameter descriptions.

Return Value

See IRowsetNotify::OnRowChange for return value descriptions.

Remarks

This method wraps the IRowsetNotify::OnRowChange method. See that method's description in the OLE DB Programmer's Reference for details.

See Also

Reference

IRowsetNotifyImpl Class

IRowsetNotify::OnRowChange

IRowsetNotifyImpl::OnRowsetChange

Notifies the consumer of any change affecting the entire rowset.

```
STDMETHOD(OnRowsetChange)(
   /* [in] */ IRowset* /* pRowset */,
   /* [in] */ DBREASON /* eReason */,
   /* [in] */ DBEVENTPHASE /* ePhase */,
   /* [in] */ BOOL /* fCantDeny */)
```

Parameters

See IRowsetNotify::OnRowsetChange for parameter descriptions.

Return Value

See IRowsetNotify::OnRowsetChange for return value descriptions.

Remarks

This method wraps the IRowsetNotify::OnRowsetChange method. See that method's description in the OLE DB Programmer's Reference for details.

See Also

Reference

IRowsetNotifyImpl Class
IRowsetNotify::OnRowsetChange

Schema Rowset Classes and Typedef Classes

A schema is a collection of database objects that are owned, or have been created by, a particular user. A catalog can contain one or more schemas, but must always contain a schema called INFORMATION_SCHEMA, which contains the views and domains of the information schema. Schema information in OLE DB is retrieved using predefined schema rowsets, and includes types, tables, columns, indexes, views, assertions and constraints, statistics, character sets, collations, and domains.

Schema rowsets are predefined rowsets representing metadata. Schema rowsets are generally used in dynamic programming, where the database structure is not known at compile time. You can use these schema rowsets to obtain information about a database at run time.

Use the typedef classes to instantiate the schema rowsets. The corresponding typedef and schema rowset classes are listed below. You must call CRestrictions::Open after you have created an instance of the schema rowset. This method returns a result set based on the restrictions you specify. See IDBSchemaRowset for information on restriction columns associated with each schema rowset.

The following table displays each OLE DB Schema Rowset and its corresponding OLE DB Templates typedef class and info class.

OLE DB Schema Rowset	Typedef class	Info class
ASSERTIONS	CAssertions	CAssertionInfo
CATALOGS	CCatalogs	CCatalogInfo
CHARACTER_SETS	CCharacterSets	CCharacterSetInfo
COLLATIONS	CCollations	CCollationInfo
COLUMN_PRIVILEGES	CColumnPrivileges	CColumnPrivilegeInfo
COLUMNS	CColumns	CColumnsInfo
CONSTRAINT_COLUMN_USAGE	CConstraintColumnUsage	CConstraintColumnUsageInfo
CONSTRAINT_TABLE_USAGE	CConstraintTableUsage	CConstraintTableUsageInfo
CHECK_CONSTRAINTS	CCheckConstraints	CCheckConstraintInfo
COLUMN_DOMAIN_USAGE	CColumnDomainUsage	CColumnDomainUsageInfo
FOREIGN_KEYS	CForeignKeys	CForeignKeysInfo
INDEXES	CIndexes	CIndexInfo
KEY_COLUMN_USAGE	CKeyColumnUsage	CKeyColumnUsageInfo
PRIMARY_KEYS	CPrimaryKeys	CPrimaryKeyInfo
PROCEDURES	CProcedures	CProcedureInfo
PROCEDURE_COLUMNS	CProcedureColumns	CProcedureColumnInfo
PROCEDURE_PARAMETERS	CProcedureParameters	CProcedureParameterInfo
PROVIDER_TYPES	CProviderTypes	CProviderInfo

REFERENTIAL_CONSTRAINTS	CReferentialConstraints	CReferentialConstraintInfo
SCHEMATA	CSchemata	CSchemataInfo
SQL_LANGUAGES	CSQLLanguages	CSQLLanguageInfo
STATISTICS	CStatistics	CStatisticInfo
TABLE_CONSTRAINTS	CTableConstraints	CTableConstraintInfo
TABLES	CTables	CTableInfo
TABLE_PRIVILEGES	CTablePrivileges	CTablePrivilegeInfo
TRANSLATIONS	CTranslations	CTranslationInfo
USAGE_PRIVILEGES	CUsagePrivileges	CUsagePrivilegeInfo
VIEW_COLUMN_USAGE	CViewColumnUsage	CViewColumnInfo
VIEWS	CViews	CViewInfo
VIEW_TABLE_USAGE	CViewTableUsage	CViewTableInfo

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

Schema Rowset Classes

For information about the schema rowset classes, see Schema Rowset Classes and Typedef Classes.

CAssertions, CAssertionInfo

Call the typedef class **CAssertions** to implement its parameter class **CAssertionInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the assertions defined in the catalog that are owned by a given user.

The following table lists the class data members for **CAssertionInfo** and their corresponding OLE DB Columns. See ASSERTIONS Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szCatalog	CONSTRAINT_CATALOG
m_szSchema	CONSTRAINT_SCHEMA
m_szName	CONSTRAINT_NAME
m_bIsDeferrable	IS_DEFERRABLE
m_bInitiallyDeferred	INITIALLY_DEFERRED
m_szDescription	DESCRIPTION

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CCatalogs, CCatalogInfo

Call the typedef class **CCatalogs** to implement its parameter class **CCatalogInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the physical attributes associated with catalogs accessible from the DBMS.

The following table lists the class data members and their corresponding OLE DB Columns. See CATALOGS Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szName	CATALOG_NAME
m_szDescription	DESCRIPTION

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CCharacterSets, CCharacterSetInfo

Call the typedef class **CCharacterSets** to implement its parameter class **CCharacterSetInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the character sets defined in the catalog that are accessible to a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See CHARACTER_SETS Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szCatalog	CHARACTER_SET_CATALOG
m_szSchema	CHARACTER_SET_SCHEMA
m_szName	CHARACTER_SET_NAME
m_szFormOfUse	FORM_OF_USE
m_nNumCharacters	NUMBER_OF_CHARACTERS
m_szCollateCatalog	DEFAULT_COLLATE_CATALOG
m_szCollateSchema	DEFAULT_COLLATE_SCHEMA
m_szCollateName	DEFAULT_COLLATE_NAME

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CCheckConstraints, CCheckConstraintInfo

Call the typedef class **CCheckConstraints** to implement its parameter class **CCheckConstraintInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the check constraints, defined in the catalog, that are owned by a given user. A check constraint specifies the data values or formats that are acceptable in one or more columns in a table.

The following table lists the class data members and their corresponding OLE DB Columns. See CHECK_CONSTRAINTS Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szCatalog	CONSTRAINT_CATALOG
m_szSchema	CONSTRAINT_SCHEMA
m_szName	CONSTRAINT_NAME
m_szCheckClause	CHECK_CLAUSE
m_szDescription	DESCRIPTION

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CCollations, CCollationInfo

Call the typedef class **CCollations** to implement its parameter class **CCollationInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the character collations, defined in the catalog, that are accessible to a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See COLLATIONS Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szCatalog	COLLATION_CATALOG
m_szSchema	COLLATION_SCHEMA
m_szName	COLLATION_NAME
m_szCharSetCatalog	CHARACTER_SET_CATALOG
m_szCharSetSchema	CHARACTER_SET_SCHEMA
m_szCharSetName	CHARACTER_SET_NAME
m_szPadAttribute	PAD_ATTRIBUTE

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CColumnDomainUsage, CColumnDomainUsageInfo

Call the typedef class **CColumnDomainUsage** to implement its parameter class **CColumnDomainUsageInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the columns, defined in the catalog, that are dependent on a domain defined in the catalog and owned by a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See COLUMN_DOMAIN_USAGE Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szCatalog	DOMAIN_CATALOG
m_szSchema	DOMAIN_SCHEMA
m_szName	DOMAIN_NAME
m_szTableCatalog	TABLE_CATALOG
m_szTableSchema	TABLE_SCHEMA
m_szTableName	TABLE_NAME
m_szColumnName	COLUMN_NAME
m_guidColumn	COLUMN_GUID
m_nColumnPropID	COLUMN_PROPID

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CColumnPrivileges, CColumnPrivilegeInfo

Call the typedef class **CColumnPrivileges** to implement its parameter class **CColumnPrivilegeInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the privileges on columns of tables, defined in the catalog, that are available to or granted by a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See COLUMN_PRIVILEGES Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szGrantor	GRANTOR
m_szGrantee	GRANTEE
m_szTableCatalog	TABLE_CATALOG
m_szTableSchema	TABLE_SCHEMA
m_szTableName	TABLE_NAME
m_szColumnName	COLUMN_NAME
m_guidColumn	COLUMN_GUID
m_nColumnPropID	COLUMN_PROPID
m_szPrivilegeType	PRIVILEGE_TYPE
m_bIsGrantable	IS_GRANTABLE

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CColumns, CColumnsInfo

Call the typedef class **CColumns** to implement its parameter class **CColumnsInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the columns of tables defined in the catalog that are accessible to a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See COLUMNS Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szTableCatalog	TABLE_CATALOG
m_szTableSchema	TABLE_SCHEMA
m_szTableName	TABLE_NAME
m_szColumnName	COLUMN_NAME
m_guidColumn	COLUMN_GUID
m_nColumnPropID	COLUMN_PROPID
m_nOrdinalPosition	ORDINAL_POSITION
m_bColumnHasDefault	COLUMN_HASDEFAULT
m_szColumnDefault	COLUMN_DEFAULT
m_nColumnFlags	COLUMN_FLAGS
m_bIsNullable	IS_NULLABLE
m_nDataType	DATA_TYPE
m_guidType	TYPE_GUID
m_nMaxLength	CHARACTER_MAXIMUM_LENGTH
m_nOctetLength	CHARACTER_OCTET_LENGTH
m_nNumericPrecision	NUMERIC_PRECISION
m_nNumericScale	NUMERIC_SCALE
m_nDateTimePrecision	DATETIME_PRECISION
m_szCharSetCatalog	CHARACTER_SET_CATALOG
m_szCharSetSchema	CHARACTER_SET_SCHEMA
m_szCharSetName	CHARACTER_SET_NAME

m_szCollationCatalog	COLLATION_CATALOG
m_szCollationSchema	COLLATION_SCHEMA
m_szCollationName	COLLATION_NAME
m_szDomainCatalog	DOMAIN_CATALOG
m_szDomainSchema	DOMAIN_SCHEMA
m_szDomainName	DOMAIN_NAME
m_szDescription	DESCRIPTION

Requirements

Header: atldbsch.h

See Also

Tasks

CatDB Sample: Data Source Schema Browser

Reference

CRestrictions Class

CConstraintColumnUsage, CConstraintColumnUsageInfo

Call the typedef class **CConstraintColumnUsage** to implement its parameter class **CConstraintColumnUsageInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the columns used by referential constraints, unique constraints, check constraints, and assertions, defined in the catalog and owned by a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See CONSTRAINT_COLUMN_USAGE Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szTableCatalog	TABLE_CATALOG
m_szTableSchema	TABLE_SCHEMA
m_szTableName	TABLE_NAME
m_szColumnName	COLUMN_NAME
m_guidColumn	COLUMN_GUID
m_nColumnPropID	COLUMN_PROPID
m_szConstraintCatalog	CONSTRAINT_CATALOG
m_szConstraintSchema	CONSTRAINT_SCHEMA
m_szConstraintName	CONSTRAINT_NAME

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CConstraintTableUsage, CConstraintTableUsageInfo

Call the typedef class **CConstraintTableUsage** to implement its parameter class **CConstraintTableUsageInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the tables used by referential constraints, unique constraints, check constraints, and assertions, defined in the catalog and owned by a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See CONSTRAINT_TABLE_USAGE Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szTableCatalog	TABLE_CATALOG
m_szTableSchema	TABLE_SCHEMA
m_szTableName	TABLE_NAME
m_szConstraintCatalog	CONSTRAINT_CATALOG
m_szConstraintSchema	CONSTRAINT_SCHEMA
m_szConstraintName	CONSTRAINT_NAME

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CForeignKeys, CForeignKeysInfo

Call the typedef class **CForeignKeys** to implement its parameter class **CForeignKeysInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the foreign key columns defined in the catalog by a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See FOREIGN_KEYS Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szPKTableCatalog	PK_TABLE_CATALOG
m_szPKTableSchema	PK_TABLE_SCHEMA
m_szPKTableName	PK_TABLE_NAME
m_szPKColumnName	PK_COLUMN_NAME
m_guidPKColumn	PK_COLUMN_GUID
m_nPKColumnPropID	PK_COLUMN_PROPID
m_szFKTableCatalog	FK_TABLE_CATALOG
m_szFKTableSchema	FK_TABLE_SCHEMA
m_szFKTableName	FK_TABLE_NAME
m_szFKColumnName	FK_COLUMN_NAME
m_guidFKColumn	FK_COLUMN_GUID
m_nFKColumnPropID	FK_COLUMN_PROPID
m_nOrdinal	ORDINAL
m_szUpdateRule	UPDATE_RULE
m_szDeleteRule	DELETE_RULE

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CIndexes, CIndexInfo

Call the typedef class **CIndexes** to implement its parameter class **CIndexInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the indexes, defined in the catalog, that are owned by a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See INDEXES Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szTableCatalog	TABLE_CATALOG
m_szTableSchema	TABLE_SCHEMA
m_szTableName	TABLE_NAME
m_szIndexCatalog	INDEX_CATALOG
m_szIndexSchema	INDEX_SCHEMA
m_szIndexName	INDEX_NAME
m_bPrimaryKey	PRIMARY_KEY
m_bUnique	UNIQUE
m_bClustered	CLUSTERED
m_nType	TYPE
m_nFillFactor	FILL_FACTOR
m_nInitialSize	INITIAL_SIZE
m_nNulls	NULLS
m_bSortBookmarks	SORT_BOOKMARKS
m_bAutoUpdate	AUTO_UPDATE
m_nNullCollation	NULL_COLLATION
m_nOrdinalPosition	ORDINAL_POSITION
m_szColumnName	COLUMN_NAME
m_guidColumn	COLUMN_GUID
m_nColumnPropID	COLUMN_PROPID
m_nCollation	COLLATION

m_nCardinality	CARDINALITY
m_nPages	PAGES
m_szFilterCondition	FILTER_CONDITION

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CKeyColumns, CKeyColumnInfo

Call the typedef class **CKeyColumns** to implement its parameter class **CKeyColumnInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the columns, defined in the catalog, that are constrained as keys by a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See KEY_COLUMN_USAGE Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szConstraintCatalog	CONSTRAINT_CATALOG
m_szConstraintSchema	CONSTRAINT_SCHEMA
m_szConstraintName	CONSTRAINT_NAME
m_szTableCatalog	TABLE_CATALOG
m_szTableSchema	TABLE_SCHEMA
m_szTableName	TABLE_NAME
m_szColumnName	COLUMN_NAME
m_guidColumn	COLUMN_GUID
m_nColumnPropID	COLUMN_PROPID
m_nOrdinalPosition	ORDINAL_POSITION

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CPrimaryKeys, CPrimaryKeyInfo

Call the typedef class **CPrimaryKeys** to implement its parameter class **CPrimaryKeyInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the primary key columns defined in the catalog by a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See PRIMARY_KEYS Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szTableCatalog	TABLE_CATALOG
m_szTableSchema	TABLE_SCHEMA
m_szTableName	TABLE_NAME
m_szColumnName	COLUMN_NAME
m_guidColumn	COLUMN_GUID
m_nColumnPropID	COLUMN_PROPID
m_nOrdinal	ORDINAL

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CProcedureColumns, CProcedureColumnInfo

Call the typedef class **CProcedureColumns** to implement its parameter class **CProcedureColumnInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class returns information about the columns of rowsets returned by procedures.

The following table lists the class data members and their corresponding OLE DB Columns. See PROCEDURE_COLUMNS Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	OLE DB columns
m_szCatalog	PROCEDURE_CATALOG
m_szSchema	PROCEDURE_SCHEMA
m_szName	PROCEDURE_NAME
m_szColumnName	COLUMN_NAME
m_guidColumn	COLUMN_GUID
m_nColumnPropID	COLUMN_PROPID
m_nRowsetNumber	ROWSET_NUMBER
m_nOrdinalPosition	ORDINAL_POSITION
m_bIsNullable	IS_NULLABLE
m_nDataType	DATA_TYPE
m_guidType	TYPE_GUID
m_nMaxLength	CHARACTER_MAXIMUM_LENGTH
m_nOctetLength	CHARACTER_OCTET_LENGTH
m_nPrecision	NUMERIC_PRECISION
m_nScale	NUMERIC_SCALE
m_szDescription	DESCRIPTION

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CProcedureParameters CProcedureParamInfo

Call the typedef class **CProcedureParameters** to implement its parameter class **CProcedureParamInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class returns information about the parameters and return codes of procedures.

The following table lists the class data members and their corresponding OLE DB Columns. See PROCEDURE_PARAMETERS Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	OLE DB columns
m_szCatalog	PROCEDURE_CATALOG
m_szSchema	PROCEDURE_SCHEMA
m_szName	PROCEDURE_NAME
m_szParameterName	PARAMETER_NAME
m_nOrdinalPosition	ORDINAL_POSITION
m_nType	PARAMETER_TYPE
m_bHasDefault	PARAMETER_HASDEFAULT
m_szDefault	PARAMETER_DEFAULT
m_bIsNullable	IS_NULLABLE
m_nDataType	DATA_TYPE
m_nMaxLength	CHARACTER_MAXIMUM_LENGTH
m_nOctetLength	CHARACTER_OCTET_LENGTH
m_nPrecision	NUMERIC_PRECISION
m_nScale	NUMERIC_SCALE
m_szDescription	DESCRIPTION

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CProcedures, CProcedureInfo

Call the typedef class **CProcedures** to implement its parameter class **CProcedureInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the procedures, defined in the catalog, that are owned by a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See PROCEDURES Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szCatalog	PROCEDURE_CATALOG
m_szSchema	PROCEDURE_SCHEMA
m_szName	PROCEDURE_NAME
m_nType	PROCEDURE_TYPE
m_szDefinition	PROCEDURE_DEFINITION
m_szDescription	DESCRIPTION

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CProviderTypes, CProviderInfo

Call the typedef class **CProviderTypes** to implement its parameter class **CProviderInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the (base) data types supported by the data provider.

The following table lists the class data members and their corresponding OLE DB Columns. See PROVIDER_TYPES Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szTypeName	TYPE_NAME
m_nDataType	DATA_TYPE
m_nColumnSize	COLUMN_SIZE
m_szLiteralPrefix	LITERAL_PREFIX
m_szLiteralSuffix	LITERAL_SUFFIX
m_szCreateParams	CREATE_PARAMS
m_bIsNullable	IS_NULLABLE
m_bCaseSensitive	CASE_SENSITIVE
m_nSearchable	SEARCHABLE
m_bUnsignedAttribute	UNSIGNED_ATTRIBUTE
m_bFixedPrecScale	FIXED_PREC_SCALE
m_bAutoUniqueValue	AUTO_UNIQUE_VALUE
m_szLocalTypeName	LOCAL_TYPE_NAME
m_nMinScale	MINIMUM_SCALE
m_nMaxScale	MAXIMUM_SCALE
m_guidType	GUID
m_szTypeLib	TYPELIB
m_szVersion	VERSION
m_bIsLong	IS_LONG
m_bBestMatch	BEST_MATCH

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CReferentialConstraints, CReferentialConstraintInfo

Call the typedef class **CReferentialConstraints** to implement its parameter class **CReferentialConstraintInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the referential constraints, defined in the catalog, that are owned by a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See REFERENTIAL_CONSTRAINTS Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szCatalog	CONSTRAINT_CATALOG
m_szSchema	CONSTRAINT_SCHEMA
m_szName	CONSTRAINT_NAME
m_szUniqueCatalog	UNIQUE_CONSTRAINT_CATALOG
m_szUniqueSchema	UNIQUE_CONSTRAINT_SCHEMA
m_szUniqueName	UNIQUE_CONSTRAINT_NAME
m_szMatchOption	MATCH_OPTION
m_szUpdateRule	UPDATE_RULE
m_szDeleteRule	DELETE_RULE
m_szDescription	DESCRIPTION

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CSchemata, CSchemataInfo

Call the typedef class **CSchemata** to implement its parameter class **CSchemataInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the schemas that are owned by a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See SCHEMATA Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szCatalog	CATALOG_NAME
m_szName	SCHEMA_NAME
m_szOwner	SCHEMA_OWNER
m_szCharCatalog	DEFAULT_CHARACTER_SET_CATALOG
m_szCharSchema	DEFAULT_CHARACTER_SET_SCHEMA
m_szCharName	DEFAULT_CHARACTER_SET_NAME

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CSQLLanguages, CSQLLanguageInfo

Call the typedef class **CSQLLanguages** to implement its parameter class **CSQLLanguageInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the conformance levels, options, and dialects supported by the SQL-implementation processing data defined in the catalog.

The following table lists the class data members and their corresponding OLE DB Columns. See SQL_LANGUAGES Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szSource	SQL_LANGUAGE_SOURCE
m_szYear	SQL_LANGUAGE_YEAR
m_szConformance	SQL_LANGUAGE_CONFORMANCE
m_szIntegrity	SQL_LANGUAGE_INTEGRITY
m_szImplementation	SQL_LANGUAGE_IMPLEMENTATION
m_szBindingStyle	SQL_LANGUAGE_BINDING_STYLE
m_szProgrammingLanguage	SQL_LANGUAGE_PROGRAMMING_LANGUAGE

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CStatistics, CStatisticInfo

Call the typedef class **CStatistics** to implement its parameter class **CStatisticInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the statistics, defined in the catalog, that are owned by a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See STATISTICS Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szTableCatalog	TABLE_CATALOG
m_szTableSchema	TABLE_SCHEMA
m_szTableName	TABLE_NAME
m_nCardinality	CARDINALITY

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CTableConstraints, CTableConstraintInfo

Call the typedef class **CTableConstraints** to implement its parameter class **CTableConstraintInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the table constraints, defined in the catalog, that are owned by a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See TABLE_CONSTRAINTS Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szCatalog	CONSTRAINT_CATALOG
m_szSchema	CONSTRAINT_SCHEMA
m_szName	CONSTRAINT_NAME
m_szTableCatalog	TABLE_CATALOG
m_szTableSchema	TABLE_SCHEMA
m_szTableName	TABLE_NAME
m_szType	CONSTRAINT_TYPE
m_bIsDeferrable	IS_DEFERRABLE
m_bInitiallyDeferred	INITIALLY_DEFERRED
m_szDescription	DESCRIPTION

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CTablePrivileges, CTablePrivilegeInfo

Call the typedef class **CTablePrivileges** to implement its parameter class **CTablePrivilegeInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the tables defined in the catalog that are accessible to a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See TABLE_PRIVILEGES Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szGrantor	GRANTOR
m_szGrantee	GRANTEE
m_szCatalog	TABLE_CATALOG
m_szSchema	TABLE_SCHEMA
m_szName	TABLE_NAME
m_szType	PRIVILEGE_TYPE
m_bIsGrantable	IS_GRANTABLE

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CTables, CTableInfo

Call the typedef class **CTables** to implement its parameter class **CTableInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the privileges on tables, defined in the catalog, that are available to or granted by a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See TABLES Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szCatalog	TABLE_CATALOG
m_szSchema	TABLE_SCHEMA
m_szName	TABLE_NAME
m_szType	TABLE_TYPE
m_guidTable	TABLE_GUID
m_szDescription	DESCRIPTION

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CTranslations, CTranslationInfo

Call the typedef class **CTranslations** to implement its parameter class **CTranslationInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the character translations defined in the catalog that are accessible to a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See TRANSLATIONS Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szCatalog	TRANSLATION_CATALOG
m_szSchema	TRANSLATION_SCHEMA
m_szName	TRANSLATION_NAME
m_szSourceCatalog	SOURCE_CHARACTER_SET_CATALOG
m_szSourceSchema	SOURCE_CHARACTER_SET_SCHEMA
m_szSourceName	SOURCE_CHARACTER_SET_NAME
m_szTargetCatalog	TARGET_CHARACTER_SET_CATALOG
m_szTargetSchema	TARGET_CHARACTER_SET_SCHEMA
m_szTargetName	TARGET_CHARACTER_SET_NAME

Requirements

Header: atldbsch.h

See Also

Tasks

CatDB Sample: Data Source Schema Browser

Reference

CRestrictions Class

CUsagePrivileges, CUsagePrivilegeInfo

Call the typedef class **CUsagePrivileges** to implement its parameter class **CUsagePrivilegeInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the USAGE privileges on objects defined in the catalog that are available to or granted by a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See USAGE_PRIVILEGES Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szGrantor	GRANTOR
m_szGrantee	GRANTEE
m_szObjectCatalog	OBJECT_CATALOG
m_szObjectSchema	OBJECT_SCHEMA
m_szObjectName	OBJECT_NAME
m_szObjectType	OBJECT_TYPE
m_szPrivilegeType	PRIVILEGE_TYPE
m_bIsGrantable	IS_GRANTABLE

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CViewColumnUsage, CViewColumnInfo

Call the typedef class **CViewColumnUsage** to implement its parameter class **CViewColumnInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the columns on which viewed tables, defined in the catalog and owned by a given user, are dependent.

The following table lists the class data members and their corresponding OLE DB Columns. See VIEW_COLUMN_USAGE Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szCatalog	VIEW_CATALOG
m_szSchema	VIEW_SCHEMA
m_szName	VIEW_NAME
m_szTableCatalog	TABLE_CATALOG
m_szTableSchema	TABLE_SCHEMA
m_szTableName	TABLE_NAME
m_szColumnName	COLUMN_NAME
m_guidColumn	COLUMN_GUID
m_nColumnPropID	COLUMN_PROPID

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CViews, CViewInfo

Call the typedef class **CViews** to implement its parameter class **CViewInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the tables on which viewed tables, defined in the catalog and owned by a given user, are dependent.

The following table lists the class data members and their corresponding OLE DB Columns. See VIEWS Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szTableCatalog	TABLE_CATALOG
m_szTableSchema	TABLE_SCHEMA
m_szTableName	TABLE_NAME
m_szDefinition	VIEW_DEFINITION
m_bCheckOption	CHECK_OPTION
m_bIsUpdatable	IS_UPDATABLE
m_szDescription	DESCRIPTION

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

CViewTableUsage, CViewTableInfo

Call the typedef class **CViewTableUsage** to implement its parameter class **CViewTableInfo**.

Remarks

See Schema Rowset Classes and Typedef Classes for more information on using typedef classes.

This class identifies the viewed tables, defined in the catalog, that are accessible to a given user.

The following table lists the class data members and their corresponding OLE DB Columns. See VIEW_TABLE_USAGE Rowset in the *OLE DB Programmer's Reference* for more information about the schema and columns.

Data members	**OLE DB columns**
m_szCatalog	VIEW_CATALOG
m_szSchema	VIEW_SCHEMA
m_szName	VIEW_NAME
m_szTableCatalog	TABLE_CATALOG
m_szTableSchema	TABLE_SCHEMA
m_szTableName	TABLE_NAME

Requirements

Header: atldbsch.h

See Also

Reference

CRestrictions Class

Macros and Global Functions for OLE DB Consumer Templates

The OLE DB Consumer Templates include the following macros and global functions:

Global Functions

AtlTraceErrorRecords	Dumps OLE DB Error Record information to the dump device if an error is returned.

Accessor Map Macros

BEGIN_ACCESSOR	Marks the beginning of an accessor entry.
BEGIN_ACCESSOR_MAP	Marks the beginning of the accessor map entries.
END_ACCESSOR	Marks the end of an accessor entry.
END_ACCESSOR_MAP	Marks the end of the accessor map entries.

Column Map Macros

BEGIN_COLUMN_MAP	Marks the beginning of the column map entries in the user record class.
BLOB_ENTRY	Used to bind a binary large object (BLOB).
BLOB_ENTRY_LENGTH	Reports the length of the BLOB data column.
BLOB_ENTRY_LENGTH_STATUS	Reports the length and status of the BLOB data column.
BLOB_ENTRY_STATUS	Reports the status of the BLOB data column.
BLOB_NAME	Used to bind a binary large object by column name.
BLOB_NAME_LENGTH	Reports the length of the BLOB data column.
BLOB_NAME_LENGTH_STATUS	Reports the length and status of the BLOB data column.
BLOB_NAME_STATUS	Reports the status of the BLOB data column.
BOOKMARK_ENTRY	Represents a bookmark entry on the rowset. A bookmark entry is a special kind of column entry.
COLUMN_ENTRY	Represents a binding to a specific column in the database.
COLUMN_ENTRY_EX	Represents a binding to the specific column in the database. Supports *type*, *length*, *precision*, *scale*, and *status* parameters.
COLUMN_ENTRY_LENGTH	Represents a binding to the specific column in the database. Supports the *length* variable.
COLUMN_ENTRY_LENGTH_STATUS	Represents a binding to the specific column in the database. Supports *status* and *length* parameters.
COLUMN_ENTRY_PS	Represents a binding to the specific column in the database. Supports *precision* and *scale* parameters.

COLUMN_ENTRY_PS_LENGTH	Represents a binding to the specific column in the database. Supports the *length* variable, *precision* and *scale* parameters.
COLUMN_ENTRY_PS_LENGTH_STATUS	Represents a binding to the specific column in the database. Supports *status* and *length* variables, *precision* and *scale* parameters.
COLUMN_ENTRY_PS_STATUS	Represents a binding to the specific column in the database. Supports the *status* variable, *precision* and *scale* parameters.
COLUMN_ENTRY_STATUS	Represents a binding to the specific column in the database. Supports the *status* variable.
COLUMN_ENTRY_TYPE	Represents a binding to a specific column in the database. Supports *type* parameter.
COLUMN_ENTRY_TYPE_SIZE	Represents a binding to the specific column in the database. Supports *type* and *size* parameters.
COLUMN_NAME	Represents a binding to a specific column in the database by name.
COLUMN_NAME_EX	Represents a binding to a specific column in the database by name. Supports specification of data type, size, precision, scale, column length, and column status.
COLUMN_NAME_LENGTH	Represents a binding to a specific column in the database by name. Supports specification of column length.
COLUMN_NAME_LENGTH_STATUS	Represents a binding to a specific column in the database by name. Supports specification of column length and status.
COLUMN_NAME_PS	Represents a binding to a specific column in the database by name. Supports specification of precision and scale.
COLUMN_NAME_PS_LENGTH	Represents a binding to a specific column in the database by name. Supports specification of precision, scale, and column length.
COLUMN_NAME_PS_LENGTH_STATUS	Represents a binding to a specific column in the database by name. Supports specification of precision, scale, column length, and column status.
COLUMN_NAME_PS_STATUS	Represents a binding to a specific column in the database by name. Supports specification of precision, scale, and column status.
COLUMN_NAME_STATUS	Represents a binding to a specific column in the database by name. Supports specification of column status.
COLUMN_NAME_TYPE	Represents a binding to a specific column in the database by name. Supports specification of data type.
COLUMN_NAME_TYPE_PS	Represents a binding to a specific column in the database by name. Supports specification of data type, precision, and scale.
COLUMN_NAME_TYPE_SIZE	Represents a binding to a specific column in the database by name. Supports specification of data type and size.
COLUMN_NAME_TYPE_STATUS	Represents a binding to a specific column in the database by name. Supports specification of data type and column status.
END_COLUMN_MAP	Marks the end of the column map entries.

Command Macros

DEFINE_COMMAND	Specifies the command that will be used to create the rowset when using the CCommand class. Accepts only string types matching the specified application type (ANSI or Unicode). It is recommended that you use DEFINE_COMMAND_EX instead of **DEFINE_COMMAND**.
DEFINE_COMMAND_EX	Specifies the command that will be used to create the rowset when using the CCommand class. Supports ANSI and Unicode applications.

Parameter Map Macros

BEGIN_PARAM_MAP	Marks the beginning of the parameter map entries in the user record class.
END_PARAM_MAP	Marks the end of the parameter map entries.
SET_PARAM_TYPE	Specifies **COLUMN_ENTRY** macros that follow the **SET_PARAM_TYPE** macro as input, output, or input/output.

See Also

Concepts

OLE DB Consumer Templates (C++)
OLE DB Consumer Templates (C++)
OLE DB Consumer Templates Reference

AtlTraceErrorRecords

Dumps OLE DB Error Record information to the dump device if an error is returned.

```
inline void AtlTraceErrorRecords(
   HRESULT hrErr = S_OK
);
```

Parameters

hErr

[in] An **HRESULT** returned by an OLE DB Consumer Template member function.

Remarks

If *hErr* is not **S_OK**, **AtlTraceErrorRecords** dumps OLE DB Error Record information to the dump device (the **Debug** tab of the Output window or a file). The Error Record information, which is obtained from the provider, includes row number, source, description, help file, context, and GUID for each error record entry. **AtlTraceErrorRecords** dumps this information only in debug builds. In release builds, it is an empty stub that is optimized out.

See Also

Reference

CDBErrorInfo Class

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

BEGIN_ACCESSOR

Marks the beginning of an accessor entry.

```
BEGIN_ACCESSOR(num, bAuto )
```

Parameters

num

[in] The zero-offset number for the accessor in this accessor map.

bAuto

[in] Specifies if this accessor is an auto accessor or a manual accessor. If **true**, the accessor is auto; if **false**, the accessor is manual. An auto accessor means data is fetched for you on move operations.

Remarks

In the case of multiple accessors on a rowset, you need to specify **BEGIN_ACCESSOR_MAP** and use the **BEGIN_ACCESSOR** macro for each individual accessor. The **BEGIN_ACCESSOR** macro is completed with the **END_ACCESSOR** macro. The **BEGIN_ACCESSOR_MAP** macro is completed with the **END_ACCESSOR_MAP** macro.

Example

See BEGIN_ACCESSOR_MAP.

See Also

Reference

BEGIN_ACCESSOR_MAP
END_ACCESSOR
END_ACCESSOR_MAP

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

BEGIN_ACCESSOR_MAP

Marks the beginning of the accessor map entries.

```
BEGIN_ACCESSOR_MAP(x, num )
```

Parameters

x

[in] The name of the user record class.

num

[in] The number of accessors in this accessor map.

Remarks

In the case of multiple accessors on a rowset, you need to specify **BEGIN_ACCESSOR_MAP** at the beginning and use the **BEGIN_ACCESSOR** macro for each individual accessor. The **BEGIN_ACCESSOR** macro is completed with the **END_ACCESSOR** macro. The accessor map is completed with the **END_ACCESSOR_MAP** macro.

If you have only one accessor in the user record, use the macro BEGIN_COLUMN_MAP.

Example

```
class CArtists : public CAccessor<CArtists>
{
public:
// Data Elements
   TCHAR m_szFirstName[20];
   TCHAR m_szLastName[30];
   short m_nAge;

// Output binding map
BEGIN_ACCESSOR_MAP( CArtists, 2 )
   BEGIN_ACCESSOR( 0, true )
        COLUMN_ENTRY( 1, m_szFirstName )
        COLUMN_ENTRY( 2, m_szLastName )
   END_ACCESSOR( )
   BEGIN_ACCESSOR( 1, false ) // Not an auto accessor
        COLUMN_ENTRY( 3, m_nAge )
   END_ACCESSOR( )
END_ACCESSOR_MAP( )
};
```

See Also

Reference

BEGIN_ACCESSOR
END_ACCESSOR
END_ACCESSOR_MAP

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

BEGIN_COLUMN_MAP

Marks the beginning of a column map entry.

```
BEGIN_COLUMN_MAP(x )
```

Parameters

x

[in] The name of the user record class derived from **CAccessor**.

Remarks

This macro is used in the case of a single accessor on a rowset. If you have multiple accessors on a rowset, use BEGIN_ACCESSOR_MAP.

The **BEGIN_COLUMN_MAP** macro is completed with the **END_COLUMN_MAP** macro. This macro is used when there is only one accessor required in the user record.

Columns correspond to fields in the rowset you want to bind.

Example

Here is a sample column and parameter map:

```
class CArtists
{
public:
// Data Elements
   TCHAR m_szFirstName[20];
   TCHAR m_szLastName[30];
   short m_nAge;

// Output binding map
BEGIN_COLUMN_MAP( CArtists)
   COLUMN_ENTRY( 1, m_szFirstName )
   COLUMN_ENTRY( 2, m_szLastName )
   COLUMN_ENTRY( 3, m_nAge )
END_COLUMN_MAP( )

// Parameter binding map
BEGIN_PARAM_MAP( CArtists )
   COLUMN_ENTRY( 1, m_nAge )
END_PARAM_MAP( )
};
```

See Also

Reference

END_COLUMN_MAP
COLUMN_ENTRY
COLUMN_ENTRY_EX

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

BEGIN_PARAM_MAP

Marks the beginning of the parameter map entries.

```
BEGIN_PARAM_MAP(x )
```

Parameters

x

[in] The name of the user record class.

Remarks

Parameters are used by commands.

Example

See BEGIN_COLUMN_MAP and see the ConfirmUser Sample.

See Also

Reference

END_PARAM_MAP

SET_PARAM_TYPE

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

BLOB_ENTRY

Used with **BEGIN_COLUMN_MAP** and **END_COLUMN_MAP** to bind a binary large object (BLOB).

```
BLOB_ENTRY(nOrdinal, IID, flags, data )
```

Parameters

nOrdinal

[in] The column number.

IID

[in] Interface GUID, such as **IDD_ISequentialStream**, used to retrieve the BLOB.

flags

[in] Storage-mode flags as defined by the OLE Structured Storage model (for example, **STGM_READ**).

data

[in] The corresponding data member in the user record.

Example

See How Can I Retrieve a BLOB?.

See Also

Reference

BEGIN_COLUMN_MAP
END_COLUMN_MAP
COLUMN_ENTRY
BLOB_ENTRY_LENGTH
BLOB_ENTRY_LENGTH_STATUS
BLOB_ENTRY_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

BLOB_ENTRY_LENGTH

Used with **BEGIN_COLUMN_MAP** and **END_COLUMN_MAP** to bind a binary large object (BLOB). Similar to BLOB_ENTRY, except that this macro also gets the length in bytes of the BLOB column.

```
BLOB_ENTRY_LENGTH(nOrdinal, IID, flags, data, length )
```

Parameters

nOrdinal

[in] The column number.

IID

[in] Interface GUID, such as **IDD_ISequentialStream**, used to retrieve the BLOB.

flags

[in] Storage-mode flags as defined by the OLE Structured Storage model (for example, **STGM_READ**).

data

[in] The corresponding data member in the user record.

length

[out] The (actual) length in bytes of the BLOB column.

Example

See How Can I Retrieve a BLOB?.

See Also

Reference

BEGIN_COLUMN_MAP
END_COLUMN_MAP
COLUMN_ENTRY
BLOB_ENTRY
BLOB_ENTRY_LENGTH_STATUS
BLOB_ENTRY_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

BLOB_ENTRY_LENGTH_STATUS

Used with **BEGIN_COLUMN_MAP** and **END_COLUMN_MAP** to bind a binary large object (BLOB). Similar to BLOB_ENTRY, except that this macro also gets the length and status of the BLOB column.

```
BLOB_ENTRY_LENGTH_STATUS(nOrdinal, IID, flags, data, length, status )
```

Parameters

nOrdinal

[in] The column number.

IID

[in] Interface GUID, such as **IDD_ISequentialStream**, used to retrieve the BLOB.

flags

[in] Storage-mode flags as defined by the OLE Structured Storage model (for example, **STGM_READ**).

data

[in] The corresponding data member in the user record.

length

[out] The (actual) length in bytes of the BLOB column.

status

[out] The status of the BLOB data column.

Example

See How Can I Retrieve a BLOB?.

See Also

Reference

BEGIN_COLUMN_MAP
END_COLUMN_MAP
COLUMN_ENTRY
BLOB_ENTRY
BLOB_ENTRY_LENGTH
BLOB_ENTRY_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

BLOB_ENTRY_STATUS

Used with **BEGIN_COLUMN_MAP** or **BEGIN_ACCESSOR_MAP** to bind a binary large object (BLOB). Similar to BLOB_ENTRY, except that this macro also gets the status of the BLOB column.

```
BLOB_ENTRY_STATUS(nOrdinal, IID, flags, data, status )
```

Parameters

nOrdinal

[in] The column number.

IID

[in] Interface GUID, such as **IDD_ISequentialStream**, used to retrieve the BLOB.

flags

[in] Storage-mode flags as defined by the OLE Structured Storage model (for example, **STGM_READ**).

data

[in] The corresponding data member in the user record.

status

[out] The status of the BLOB field.

Example

See How Can I Retrieve a BLOB?.

See Also

Reference

BEGIN_COLUMN_MAP
END_COLUMN_MAP
COLUMN_ENTRY
BLOB_ENTRY
BLOB_ENTRY_LENGTH
BLOB_ENTRY_LENGTH_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

BLOB_NAME

Used with **BEGIN_COLUMN_MAP** and **END_COLUMN_MAP** to bind a binary large object (BLOB). Similar to BLOB_ENTRY, except that this macro takes a column name instead of a column number.

```
BLOB_NAME(pszName, IID, flags, data )
```

Parameters

pszName

[in] A pointer to the column name. The name must be a Unicode string. You can accomplish this by putting an 'L' in front of the name, for example: `L"MyColumn"`.

IID

[in] Interface GUID, such as **IDD_ISequentialStream**, used to retrieve the BLOB.

flags

[in] Storage-mode flags as defined by the OLE Structured Storage model (for example, **STGM_READ**).

data

[in] The corresponding data member in the user record.

Example

See How Can I Retrieve a BLOB?.

See Also

Reference

BEGIN_COLUMN_MAP
END_COLUMN_MAP
COLUMN_ENTRY
BLOB_NAME_LENGTH
BLOB_NAME_LENGTH_STATUS
BLOB_NAME_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

BLOB_NAME_LENGTH

Used with **BEGIN_COLUMN_MAP** and **END_COLUMN_MAP** to bind a binary large object (BLOB). Similar to BLOB_NAME, except that this macro also gets the length in bytes of the BLOB data column.

```
BLOB_NAME_LENGTH(pszName, IID, flags, data, length )
```

Parameters

pszName

[in] A pointer to the column name. The name must be a Unicode string. You can accomplish this by putting an 'L' in front of the name, for example: `L"MyColumn"`.

IID

[in] Interface GUID, such as **IDD_ISequentialStream**, used to retrieve the BLOB.

flags

[in] Storage-mode flags as defined by the OLE Structured Storage model (for example, **STGM_READ**).

data

[in] The corresponding data member in the user record.

length

[out] The (actual) length in bytes of the BLOB column.

See Also

Reference

BEGIN_COLUMN_MAP
END_COLUMN_MAP
COLUMN_ENTRY
BLOB_NAME
BLOB_NAME_LENGTH_STATUS
BLOB_NAME_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

BLOB_NAME_LENGTH_STATUS

Used with **BEGIN_COLUMN_MAP** and **END_COLUMN_MAP** to bind a binary large object (BLOB). Similar to BLOB_NAME, except that this macro also gets the length and status of the BLOB data column.

```
BLOB_NAME_LENGTH_STATUS(pszName, IID, flags, data, length, status )
```

Parameters

pszName

[in] A pointer to the column name. The name must be a Unicode string. You can accomplish this by putting an 'L' in front of the name, for example: `L"MyColumn"`.

IID

[in] Interface GUID, such as **IDD_ISequentialStream**, used to retrieve the BLOB.

flags

[in] Storage-mode flags as defined by the OLE Structured Storage model (for example, **STGM_READ**).

data

[in] The corresponding data member in the user record.

length

[out] The (actual) length in bytes of the BLOB column.

status

[out] The status of the BLOB field.

See Also

Reference

BEGIN_COLUMN_MAP
END_COLUMN_MAP
COLUMN_ENTRY
BLOB_NAME
BLOB_NAME_LENGTH
BLOB_NAME_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

BLOB_NAME_STATUS

Used with **BEGIN_COLUMN_MAP** and **END_COLUMN_MAP** to bind a binary large object (BLOB). Similar to BLOB_NAME, except that this macro also gets the status of the BLOB data column.

```
BLOB_NAME_STATUS(pszName, IID, flags, data, status )
```

Parameters

pszName

[in] A pointer to the column name. The name must be a Unicode string. You can accomplish this by putting an 'L' in front of the name, for example: `L"MyColumn"`.

IID

[in] Interface GUID, such as **IDD_ISequentialStream**, used to retrieve the BLOB.

flags

[in] Storage-mode flags as defined by the OLE Structured Storage model (for example, **STGM_READ**).

data

[in] The corresponding data member in the user record.

status

[out] The status of the BLOB field.

See Also

Reference

BEGIN_COLUMN_MAP
END_COLUMN_MAP
COLUMN_ENTRY
BLOB_NAME
BLOB_NAME_LENGTH
BLOB_NAME_LENGTH_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

BOOKMARK_ENTRY

Binds the bookmark column.

```
BOOKMARK_ENTRY(variable )
```

Parameters

variable

[in] The variable to be bound to the bookmark column.

Example

```
class CArtists
{
public:
// Data Elements
   CBookmark<4> m_bookmark;
   TCHAR m_szFirstName[20];
   TCHAR m_szLastName[30];
   short m_nAge;

// Output binding map
BEGIN_COLUMN_MAP( CArtists )
   BOOKMARK_ENTRY( m_bookmark )
   COLUMN_ENTRY( 1, m_szFirstName )
   COLUMN_ENTRY( 2, m_szLastName )
   COLUMN_ENTRY( 3, m_nAge )
END_COLUMN_MAP( )
};
```

See Also

Reference

CBookmark Class

DBPROP_BOOKMARKS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_ENTRY

Represents a binding on the rowset to the specific column in the rowset.

```
COLUMN_ENTRY(nOrdinal, data )
```

Parameters

See DBBINDING in the *OLE DB Programmer's Reference*.

nOrdinal

[in] The column number.

data

[in] The corresponding data member in the user record.

Remarks

The **COLUMN_ENTRY** macro is used in the following places:

- Between the BEGIN_COLUMN_MAP and END_COLUMN_MAP macros.
- Between the BEGIN_ACCESSOR and END_ACCESSOR macros.
- Between the BEGIN_PARAM_MAP and END_PARAM_MAP macros.

Example

See BEGIN_COLUMN_MAP, BEGIN_ACCESSOR_MAP, and the ConfirmUser Sample.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_ENTRY_EX
COLUMN_ENTRY_PS
COLUMN_ENTRY_PS_LENGTH
COLUMN_ENTRY_LENGTH
COLUMN_ENTRY_LENGTH_STATUS
COLUMN_ENTRY_PS_LENGTH_STATUS
COLUMN_ENTRY_STATUS
COLUMN_ENTRY_PS_STATUS
COLUMN_ENTRY_TYPE
COLUMN_ENTRY_TYPE_SIZE
END_ACCESSOR
END_ACCESSOR_MAP
END_COLUMN_MAP

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_ENTRY_EX

Represents a binding on the rowset to the specific column in the database.

```
COLUMN_ENTRY_EX(nOrdinal, wType, nLength, nPrecision, nScale, data, length, status )
```

Parameters

See DBBINDING in the *OLE DB Programmer's Reference*.

nOrdinal

[in] The column number.

wType

[in] The data type.

nLength

[in] The data size in bytes.

nPrecision

[in] The maximum precision to use when getting data and *wType* is **DBTYPE_NUMERIC**. Otherwise, this parameter is ignored.

nScale

[in] The scale to use when getting data and *wType* is **DBTYPE_NUMERIC** or **DBTYPE_DECIMAL**.

data

[in] The corresponding data member in the user record.

length

[in] The variable to be bound to the column length.

status

[in] The variable to be bound to the column status.

Remarks

The **COLUMN_ENTRY_EX** macro is used in the following places:

- Between the BEGIN_COLUMN_MAP and END_COLUMN_MAP macros.
- Between the BEGIN_ACCESSOR and END_ACCESSOR macros.
- Between the BEGIN_PARAM_MAP and END_PARAM_MAP macros.

Example

See BOOKMARK_ENTRY.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_ENTRY
COLUMN_ENTRY_PS
COLUMN_ENTRY_PS_LENGTH
COLUMN_ENTRY_LENGTH
COLUMN_ENTRY_LENGTH_STATUS
COLUMN_ENTRY_PS_LENGTH_STATUS
COLUMN_ENTRY_STATUS

COLUMN_ENTRY_PS_STATUS
END_ACCESSOR
END_ACCESSOR_MAP
END_COLUMN_MAP

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_ENTRY_LENGTH

Represents a binding on the rowset to the specific column in the database.

```
COLUMN_ENTRY_LENGTH(nOrdinal, data, length )
```

Parameters

See DBBINDING in the *OLE DB Programmer's Reference*.

nOrdinal

[in] The column number, starting with one. Bookmark corresponds to column zero.

data

[in] The corresponding data member in the user record.

length

[in] The variable to be bound to the column length.

Remarks

This macro supports the *length* variable. It is used in the following places:

- Between the BEGIN_COLUMN_MAP and END_COLUMN_MAP macros.
- Between the BEGIN_ACCESSOR and END_ACCESSOR macros.
- Between the BEGIN_PARAM_MAP and END_PARAM_MAP macros.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_ENTRY
COLUMN_ENTRY_EX
COLUMN_ENTRY_PS
COLUMN_ENTRY_PS_LENGTH
COLUMN_ENTRY_LENGTH_STATUS
COLUMN_ENTRY_PS_LENGTH_STATUS
COLUMN_ENTRY_STATUS
COLUMN_ENTRY_PS_STATUS
END_ACCESSOR
END_ACCESSOR_MAP
END_COLUMN_MAP

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_ENTRY_LENGTH_STATUS

Represents a binding on the rowset to the specific column in the database.

```
COLUMN_ENTRY_LENGTH_STATUS(nOrdinal, data, length, status )
```

Parameters

See DBBINDING in the *OLE DB Programmer's Reference*.

nOrdinal

[in] The column number.

data

[in] The corresponding data member in the user record.

length

[in] The variable to be bound to the column length.

status

[in] The variable to be bound to the column status.

Remarks

Use this macro when you want to support length and status variables. It is used in the following places:

- Between the BEGIN_COLUMN_MAP and END_COLUMN_MAP macros.
- Between the BEGIN_ACCESSOR and END_ACCESSOR macros.
- Between the BEGIN_PARAM_MAP and END_PARAM_MAP macros.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_ENTRY
COLUMN_ENTRY_EX
COLUMN_ENTRY_LENGTH
COLUMN_ENTRY_PS
COLUMN_ENTRY_PS_LENGTH
COLUMN_ENTRY_PS_LENGTH_STATUS
COLUMN_ENTRY_STATUS
COLUMN_ENTRY_PS_STATUS
END_ACCESSOR
END_ACCESSOR_MAP
END_COLUMN_MAP

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_ENTRY_PS

Represents a binding on the rowset to the specific column in the rowset.

```
COLUMN_ENTRY_PS(nOrdinal, nPrecision, nScale, data )
```

Parameters

See DBBINDING in the *OLE DB Programmer's Reference*.

nOrdinal

[in] The column number.

nPrecision

[in] The maximum precision of the column you want to bind.

nScale

[in] The scale of the column you want to bind.

data

[in] The corresponding data member in the user record.

Remarks

Allows you to specify the precision and scale of the column you want to bind. It is used in the following places:

- Between the BEGIN_COLUMN_MAP and END_COLUMN_MAP macros.
- Between the BEGIN_ACCESSOR and END_ACCESSOR macros.
- Between the BEGIN_PARAM_MAP and END_PARAM_MAP macros.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_ENTRY
COLUMN_ENTRY_EX
COLUMN_ENTRY_LENGTH
COLUMN_ENTRY_PS_LENGTH
COLUMN_ENTRY_LENGTH_STATUS
COLUMN_ENTRY_PS_LENGTH_STATUS
COLUMN_ENTRY_STATUS
COLUMN_ENTRY_PS_STATUS
END_ACCESSOR
END_ACCESSOR_MAP
END_COLUMN_MAP

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_ENTRY_PS_LENGTH

Represents a binding on the rowset to the specific column in the database.

```
COLUMN_ENTRY_PS_LENGTH(nOrdinal, nPrecision, nScale, data, length )
```

Parameters

See DBBINDING in the *OLE DB Programmer's Reference*.

nOrdinal

[in] The column number, starting with one. Bookmark corresponds to column zero.

nPrecision

[in] The maximum precision of the column you want to bind.

nScale

[in] The scale of the column you want to bind.

data

[in] The corresponding data member in the user record.

length

[in] The variable to be bound to the column length.

Remarks

Allows you to specify the precision and scale of the column you want to bind. This macro supports the *length* variable. It is used in the following places:

- Between the BEGIN_COLUMN_MAP and END_COLUMN_MAP macros.
- Between the BEGIN_ACCESSOR and END_ACCESSOR macros.
- Between the BEGIN_PARAM_MAP and END_PARAM_MAP macros.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_ENTRY
COLUMN_ENTRY_EX
COLUMN_ENTRY_LENGTH
COLUMN_ENTRY_PS
COLUMN_ENTRY_LENGTH_STATUS
COLUMN_ENTRY_PS_LENGTH_STATUS
COLUMN_ENTRY_STATUS
COLUMN_ENTRY_PS_STATUS
END_ACCESSOR
END_ACCESSOR_MAP
END_COLUMN_MAP

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_ENTRY_PS_LENGTH_STATUS

Represents a binding on the rowset to the specific column in the database.

```
COLUMN_ENTRY_PS_LENGTH_STATUS(nOrdinal, nPrecision, nScale, data, length, status )
```

Parameters

See DBBINDING in the *OLE DB Programmer's Reference*.

nOrdinal

[in] The column number.

nPrecision

[in] The maximum precision of the column you want to bind.

nScale

[in] The scale of the column you want to bind.

data

[in] The corresponding data member in the user record.

length

[in] The variable to be bound to the column length.

status

[in] The variable to be bound to the column status.

Remarks

Allows you to specify the precision and scale of the column you want to bind. Use this macro when you want to support length and status variables. It is used in the following places:

- Between the BEGIN_COLUMN_MAP and END_COLUMN_MAP macros.
- Between the BEGIN_ACCESSOR and END_ACCESSOR macros.
- Between the BEGIN_PARAM_MAP and END_PARAM_MAP macros.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_ENTRY
COLUMN_ENTRY_EX
COLUMN_ENTRY_LENGTH
COLUMN_ENTRY_PS
COLUMN_ENTRY_PS_LENGTH
COLUMN_ENTRY_LENGTH_STATUS
COLUMN_ENTRY_STATUS
COLUMN_ENTRY_PS_STATUS
END_ACCESSOR
END_ACCESSOR_MAP
END_COLUMN_MAP

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_ENTRY_PS_STATUS

Represents a binding on the rowset to the specific column in the database.

```
COLUMN_ENTRY_PS_STATUS(nOrdinal, nPrecision, nScale, data, status )
```

Parameters

See DBBINDING in the *OLE DB Programmer's Reference*.

nOrdinal

[in] The column number.

nPrecision

[in] The maximum precision of the column you want to bind.

nScale

[in] The scale of the column you want to bind.

data

[in] The corresponding data member in the user record.

status

[in] The variable to be bound to the column status.

Remarks

Allows you to specify the precision and scale of the column you want to bind. This macro supports the *status* variable. It is used in the following places:

- Between the BEGIN_COLUMN_MAP and END_COLUMN_MAP macros.
- Between the BEGIN_ACCESSOR and END_ACCESSOR macros.
- Between the BEGIN_PARAM_MAP and END_PARAM_MAP macros.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_ENTRY
COLUMN_ENTRY_EX
COLUMN_ENTRY_LENGTH
COLUMN_ENTRY_PS
COLUMN_ENTRY_PS_LENGTH
COLUMN_ENTRY_LENGTH_STATUS
COLUMN_ENTRY_PS_LENGTH_STATUS
COLUMN_ENTRY_STATUS
END_ACCESSOR
END_ACCESSOR_MAP
END_COLUMN_MAP

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_ENTRY_STATUS

Represents a binding on the rowset to the specific column in the database.

```
COLUMN_ENTRY_STATUS(nOrdinal, data, status )
```

Parameters

See DBBINDING in the *OLE DB Programmer's Reference*.

nOrdinal

[in] The column number.

data

[in] The corresponding data member in the user record.

status

[in] The variable to be bound to the column status.

Remarks

This macro supports the *status* variable. It is used in the following places:

- Between the BEGIN_COLUMN_MAP and END_COLUMN_MAP macros.
- Between the BEGIN_ACCESSOR and END_ACCESSOR macros.
- Between the BEGIN_PARAM_MAP and END_PARAM_MAP macros.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_ENTRY
COLUMN_ENTRY_EX
COLUMN_ENTRY_LENGTH
COLUMN_ENTRY_PS
COLUMN_ENTRY_PS_LENGTH
COLUMN_ENTRY_LENGTH_STATUS
COLUMN_ENTRY_PS_LENGTH_STATUS
COLUMN_ENTRY_PS_STATUS
END_ACCESSOR
END_ACCESSOR_MAP
END_COLUMN_MAP

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_ENTRY_TYPE

Represents a binding to the specific column in the database. Supports *type* parameter.

```
COLUMN_ENTRY_TYPE (nOrdinal, wType, data )
```

Parameters

nOrdinal

[in] The column number.

wType

[in] Data type of column entry.

data

[in] The corresponding data member in the user record.

Remarks

This macro is a specialized variant of the COLUMN_ENTRY macro that provides a means of specifying data type.

See Also

Reference

BEGIN_COLUMN_MAP

END_COLUMN_MAP

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_ENTRY_TYPE_SIZE

Represents a binding to the specific column in the database. Supports *type* and *size* parameters.

```
COLUMN_ENTRY_TYPE_SIZE(nOrdinal, wType, nLength, data )
```

Parameters

nOrdinal

[in] The column number.

wType

[in] Data type of column entry.

nLength

[in] Size of column entry in bytes.

data

[in] The corresponding data member in the user record.

Remarks

This macro is a specialized variant of the COLUMN_ENTRY macro that provides a means of specifying data size and type.

See Also

Reference

BEGIN_COLUMN_MAP
END_COLUMN_MAP

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_NAME

Represents a binding on the rowset to the specific column in the rowset. Similar to COLUMN_ENTRY, except that this macro takes the column name instead of the column number.

```
COLUMN_NAME(pszName, data )
```

Parameters

pszName

[in] A pointer to the column name. The name must be a Unicode string. You can accomplish this by putting an 'L' in front of the name, for example: `L"MyColumn"`.

data

[in] The corresponding data member in the user record.

Remarks

The **COLUMN_NAME_*** macros are used in the same places as COLUMN_ENTRY:

- Between the BEGIN_COLUMN_MAP and END_COLUMN_MAP macros.
- Between the BEGIN_ACCESSOR and END_ACCESSOR macros.
- Between the BEGIN_PARAM_MAP and END_PARAM_MAP macros.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_NAME_EX
COLUMN_NAME_LENGTH
COLUMN_NAME_LENGTH_STATUS
COLUMN_NAME_STATUS
COLUMN_NAME_PS
COLUMN_NAME_PS_LENGTH
COLUMN_NAME_PS_STATUS
COLUMN_NAME_PS_LENGTH_STATUS
COLUMN_NAME_TYPE
COLUMN_NAME_TYPE_PS
COLUMN_NAME_TYPE_SIZE
COLUMN_NAME_TYPE_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_NAME_EX

Represents a binding on the rowset to the specific column in the rowset. Similar to COLUMN_NAME, except that this macro also takes data type, size, precision, scale, column length, and column status.

```
COLUMN_NAME_EX(pszName, wType, nLength, nPrecision, nScale, data, length, status )
```

Parameters

pszName

[in] A pointer to the column name. The name must be a Unicode string. You can accomplish this by putting an 'L' in front of the name, for example: `L"MyColumn"`.

wType

[in] The data type.

nLength

[in] The data size in bytes.

nPrecision

[in] The maximum precision to use when getting data and *wType* is **DBTYPE_NUMERIC**. Otherwise, this parameter is ignored.

nScale

[in] The scale to use when getting data and *wType* is **DBTYPE_NUMERIC** or **DBTYPE_DECIMAL**.

data

[in] The corresponding data member in the user record.

length

[in] The variable to be bound to the column length.

status

[in] The variable to be bound to the column status.

Remarks

See COLUMN_NAME for information on where the **COLUMN_NAME_*** macros are used.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_NAME
COLUMN_NAME_LENGTH
COLUMN_NAME_LENGTH_STATUS
COLUMN_NAME_STATUS
COLUMN_NAME_PS
COLUMN_NAME_PS_LENGTH
COLUMN_NAME_PS_STATUS
COLUMN_NAME_PS_LENGTH_STATUS
COLUMN_NAME_TYPE
COLUMN_NAME_TYPE_PS
COLUMN_NAME_TYPE_SIZE
COLUMN_NAME_TYPE_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_NAME_LENGTH

Represents a binding on the rowset to the specific column in the rowset. Similar to COLUMN_NAME, except that this macro also takes column length.

```
COLUMN_NAME_LENGTH(pszName, data, length )
```

Parameters

pszName

[in] A pointer to the column name. The name must be a Unicode string. You can accomplish this by putting an 'L' in front of the name, for example: `L"MyColumn"`.

data

[in] The corresponding data member in the user record.

length

[in] The variable to be bound to the column length.

Remarks

See COLUMN_NAME for information on where the **COLUMN_NAME_*** macros are used.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_NAME
COLUMN_NAME_EX
COLUMN_NAME_LENGTH_STATUS
COLUMN_NAME_STATUS
COLUMN_NAME_PS
COLUMN_NAME_PS_LENGTH
COLUMN_NAME_PS_STATUS
COLUMN_NAME_PS_LENGTH_STATUS
COLUMN_NAME_TYPE
COLUMN_NAME_TYPE_PS
COLUMN_NAME_TYPE_SIZE
COLUMN_NAME_TYPE_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_NAME_LENGTH_STATUS

Represents a binding on the rowset to the specific column in the rowset. Similar to COLUMN_NAME, except that this macro also takes column length and column status.

```
COLUMN_NAME_LENGTH_STATUS(pszName, data, length, status )
```

Parameters

pszName

[in] A pointer to the column name. The name must be a Unicode string. You can accomplish this by putting an 'L' in front of the name, for example: `L"MyColumn"`.

data

[in] The corresponding data member in the user record.

length

[in] The variable to be bound to the column length.

status

[in] The variable to be bound to the column status.

Remarks

See COLUMN_NAME for information on where the **COLUMN_NAME_*** macros are used.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_NAME
COLUMN_NAME_EX
COLUMN_NAME_LENGTH
COLUMN_NAME_STATUS
COLUMN_NAME_PS
COLUMN_NAME_PS_LENGTH
COLUMN_NAME_PS_STATUS
COLUMN_NAME_PS_LENGTH_STATUS
COLUMN_NAME_TYPE
COLUMN_NAME_TYPE_PS
COLUMN_NAME_TYPE_SIZE
COLUMN_NAME_TYPE_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_NAME_PS

Represents a binding on the rowset to the specific column in the rowset. Similar to COLUMN_NAME, except that this macro also takes precision and scale.

```
COLUMN_NAME_PS(pszName, nPrecision, nScale, data )
```

Parameters

pszName

[in] A pointer to the column name. The name must be a Unicode string. You can accomplish this by putting an 'L' in front of the name, for example: `L"MyColumn"`.

nPrecision

[in] The maximum precision of the column you want to bind.

nScale

[in] The scale of the column you want to bind.

data

[in] The corresponding data member in the user record.

Remarks

See COLUMN_NAME for information on where the **COLUMN_NAME_*** macros are used.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_NAME
COLUMN_NAME_EX
COLUMN_NAME_LENGTH
COLUMN_NAME_LENGTH_STATUS
COLUMN_NAME_STATUS
COLUMN_NAME_PS_LENGTH
COLUMN_NAME_PS_STATUS
COLUMN_NAME_PS_LENGTH_STATUS
COLUMN_NAME_TYPE
COLUMN_NAME_TYPE_PS
COLUMN_NAME_TYPE_SIZE
COLUMN_NAME_TYPE_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_NAME_PS_LENGTH

Represents a binding on the rowset to the specific column in the rowset. Similar to COLUMN_NAME, except that this macro also takes precision, scale, and column length.

```
COLUMN_NAME_PS_LENGTH(pszName, nPrecision, nScale, data, length )
```

Parameters

pszName

[in] A pointer to the column name. The name must be a Unicode string. You can accomplish this by putting an 'L' in front of the name, for example: `L"MyColumn"`.

nPrecision

[in] The maximum precision of the column you want to bind.

nScale

[in] The scale of the column you want to bind.

data

[in] The corresponding data member in the user record.

length

[in] The variable to be bound to the column length.

Remarks

See COLUMN_NAME for information on where the **COLUMN_NAME_*** macros are used.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_NAME
COLUMN_NAME_EX
COLUMN_NAME_LENGTH
COLUMN_NAME_LENGTH_STATUS
COLUMN_NAME_STATUS
COLUMN_NAME_PS
COLUMN_NAME_PS_STATUS
COLUMN_NAME_PS_LENGTH_STATUS
COLUMN_NAME_TYPE
COLUMN_NAME_TYPE_PS
COLUMN_NAME_TYPE_SIZE
COLUMN_NAME_TYPE_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_NAME_PS_LENGTH_STATUS

Represents a binding on the rowset to the specific column in the rowset. Similar to COLUMN_NAME, except that this macro also takes precision, scale, column length, and column status.

```
COLUMN_NAME_PS_LENGTH_STATUS(pszName, nPrecision, nScale, data, length, status )
```

Parameters

pszName

[in] A pointer to the column name. The name must be a Unicode string. You can accomplish this by putting an 'L' in front of the name, for example: `L"MyColumn"`.

nPrecision

[in] The maximum precision of the column you want to bind.

nScale

[in] The scale of the column you want to bind.

data

[in] The corresponding data member in the user record.

length

[in] The variable to be bound to the column length.

status

[in] The variable to be bound to the column status.

Remarks

See COLUMN_NAME for information on where the **COLUMN_NAME_*** macros are used.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_NAME
COLUMN_NAME_EX
COLUMN_NAME_LENGTH
COLUMN_NAME_LENGTH_STATUS
COLUMN_NAME_STATUS
COLUMN_NAME_PS
COLUMN_NAME_PS_LENGTH
COLUMN_NAME_PS_STATUS
COLUMN_NAME_TYPE
COLUMN_NAME_TYPE_PS
COLUMN_NAME_TYPE_SIZE
COLUMN_NAME_TYPE_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_NAME_PS_STATUS

Represents a binding on the rowset to the specific column in the rowset. Similar to COLUMN_NAME, except that this macro also takes precision, scale, and column status.

```
COLUMN_NAME_PS_STATUS(pszName, nPrecision, nScale, data, status )
```

Parameters

pszName

[in] A pointer to the column name. The name must be a Unicode string. You can accomplish this by putting an 'L' in front of the name, for example: `L"MyColumn"`.

nPrecision

[in] The maximum precision of the column you want to bind.

nScale

[in] The scale of the column you want to bind.

data

[in] The corresponding data member in the user record.

status

[in] The variable to be bound to the column status.

Remarks

See COLUMN_NAME for information on where the **COLUMN_NAME_*** macros are used.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_NAME
COLUMN_NAME_EX
COLUMN_NAME_LENGTH
COLUMN_NAME_LENGTH_STATUS
COLUMN_NAME_STATUS
COLUMN_NAME_PS
COLUMN_NAME_PS_LENGTH
COLUMN_NAME_PS_LENGTH_STATUS
COLUMN_NAME_TYPE
COLUMN_NAME_TYPE_PS
COLUMN_NAME_TYPE_SIZE
COLUMN_NAME_TYPE_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_NAME_STATUS

Represents a binding on the rowset to the specific column in the rowset. Similar to COLUMN_NAME, except that this macro also takes column status.

```
COLUMN_NAME_STATUS(pszName, data, status )
```

Parameters

pszName

[in] A pointer to the column name. The name must be a Unicode string. You can accomplish this by putting an 'L' in front of the name, for example: `L"MyColumn"`.

data

[in] The corresponding data member in the user record.

status

[in] The variable to be bound to the column status.

Remarks

See COLUMN_NAME for information on where the **COLUMN_NAME_*** macros are used.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_NAME
COLUMN_NAME_EX
COLUMN_NAME_LENGTH
COLUMN_NAME_LENGTH_STATUS
COLUMN_NAME_PS
COLUMN_NAME_PS_LENGTH
COLUMN_NAME_PS_STATUS
COLUMN_NAME_PS_LENGTH_STATUS
COLUMN_NAME_TYPE
COLUMN_NAME_TYPE_PS
COLUMN_NAME_TYPE_SIZE
COLUMN_NAME_TYPE_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_NAME_TYPE

Represents a binding on the rowset to the specific column in the rowset. Similar to COLUMN_NAME, except that this macro also takes data type.

```
COLUMN_NAME_TYPE(pszName, wType, data )
```

Parameters

pszName

[in] A pointer to the column name. The name must be a Unicode string. You can accomplish this by putting an 'L' in front of the name, for example: `L"MyColumn"`.

wType

[in] The data type.

data

[in] The corresponding data member in the user record.

Remarks

See COLUMN_NAME for information on where the **COLUMN_NAME_*** macros are used.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_NAME
COLUMN_NAME_EX
COLUMN_NAME_LENGTH
COLUMN_NAME_LENGTH_STATUS
COLUMN_NAME_STATUS
COLUMN_NAME_PS
COLUMN_NAME_PS_LENGTH
COLUMN_NAME_PS_STATUS
COLUMN_NAME_PS_LENGTH_STATUS
COLUMN_NAME_TYPE_PS
COLUMN_NAME_TYPE_SIZE
COLUMN_NAME_TYPE_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_NAME_TYPE_PS

Represents a binding on the rowset to the specific column in the rowset. Similar to COLUMN_NAME, except that this macro also takes data type, precision, and scale.

```
COLUMN_NAME_TYPE_PS(pszName, wType, nPrecision, nScale, data )
```

Parameters

pszName

[in] A pointer to the column name. The name must be a Unicode string. You can accomplish this by putting an 'L' in front of the name, for example: `L"MyColumn"`.

wType

[in] The data type.

nPrecision

[in] The maximum precision to use when getting data and *wType* is **DBTYPE_NUMERIC**. Otherwise, this parameter is ignored.

nScale

[in] The scale to use when getting data and *wType* is **DBTYPE_NUMERIC** or **DBTYPE_DECIMAL**.

data

[in] The corresponding data member in the user record.

Remarks

See COLUMN_NAME for information on where the **COLUMN_NAME_*** macros are used.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_NAME
COLUMN_NAME_EX
COLUMN_NAME_LENGTH
COLUMN_NAME_LENGTH_STATUS
COLUMN_NAME_STATUS
COLUMN_NAME_PS
COLUMN_NAME_PS_LENGTH
COLUMN_NAME_PS_STATUS
COLUMN_NAME_PS_LENGTH_STATUS
COLUMN_NAME_TYPE
COLUMN_NAME_TYPE_SIZE
COLUMN_NAME_TYPE_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_NAME_TYPE_SIZE

Represents a binding on the rowset to the specific column in the rowset. Similar to COLUMN_NAME, except that this macro also takes data type and size.

```
COLUMN_NAME_TYPE_SIZE(pszName, wType, nLength, data )
```

Parameters

pszName

[in] A pointer to the column name. The name must be a Unicode string. You can accomplish this by putting an 'L' in front of the name, for example: `L"MyColumn"`.

wType

[in] The data type.

nLength

[in] The data size in bytes.

data

[in] The corresponding data member in the user record.

Remarks

See COLUMN_NAME for information on where the **COLUMN_NAME_*** macros are used.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_NAME
COLUMN_NAME_EX
COLUMN_NAME_LENGTH
COLUMN_NAME_LENGTH_STATUS
COLUMN_NAME_STATUS
COLUMN_NAME_PS
COLUMN_NAME_PS_LENGTH
COLUMN_NAME_PS_STATUS
COLUMN_NAME_PS_LENGTH_STATUS
COLUMN_NAME_TYPE
COLUMN_NAME_TYPE_PS
COLUMN_NAME_TYPE_STATUS

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

COLUMN_NAME_TYPE_STATUS

Represents a binding on the rowset to the specific column in the rowset. Similar to COLUMN_NAME, except that this macro also takes data type and column status.

```
COLUMN_NAME_TYPE_STATUS(pszName, wType, status, data )
```

Parameters

pszName

[in] A pointer to the column name. The name must be a Unicode string. You can accomplish this by putting an 'L' in front of the name, for example: `L"MyColumn"`.

wType

[in] The data type.

status

[in] The variable to be bound to the column status.

data

[in] The corresponding data member in the user record.

Remarks

See COLUMN_NAME for information on where the **COLUMN_NAME_*** macros are used.

See Also

Reference

BEGIN_ACCESSOR
BEGIN_ACCESSOR_MAP
BEGIN_COLUMN_MAP
COLUMN_NAME
COLUMN_NAME_EX
COLUMN_NAME_LENGTH
COLUMN_NAME_LENGTH_STATUS
COLUMN_NAME_STATUS
COLUMN_NAME_PS
COLUMN_NAME_PS_LENGTH
COLUMN_NAME_PS_STATUS
COLUMN_NAME_PS_LENGTH_STATUS
COLUMN_NAME_TYPE
COLUMN_NAME_TYPE_PS
COLUMN_NAME_TYPE_SIZE

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

DEFINE_COMMAND

Specifies the command that will be used to create the rowset when using the CCommand class. Accepts only string types matching the specified application type (ANSI or Unicode).

Note
It is recommended that you use DEFINE_COMMAND_EX instead of **DEFINE_COMMAND**.

```
DEFINE_COMMAND(x, szCommand )
```

Parameters

x

[in] The name of the user record (command) class.

szCommand

[in] The command string that will be used to create the rowset when using CCommand.

Remarks

The command string that you specify will be used as the default if you do not specify command text in the CCommand::Open method.

This macro accepts ANSI strings if you build your application as ANSI, or Unicode strings if you build your application as Unicode. It is recommended that you use DEFINE_COMMAND_EX instead of **DEFINE_COMMAND**, because the former accepts Unicode strings, regardless of the ANSI or Unicode application type.

Example

See BOOKMARK_ENTRY.

See Also

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

DEFINE_COMMAND_EX

Specifies the command that will be used to create the rowset when using the CCommand class. Supports Unicode and ANSI applications.

```
DEFINE_COMMAND_EX(x, wszCommand )
```

Parameters

x

[in] The name of the user record (command) class.

wszCommand

[in] The command string that will be used to create the rowset when using CCommand.

Remarks

The command string that you specify will be used as the default if you do not specify command text in the CCommand::Open method.

This macro accepts Unicode strings, regardless of the application type. This macro is preferred over DEFINE_COMMAND because it supports Unicode as well as ANSI applications.

Example

See BOOKMARK_ENTRY.

See Also

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

END_ACCESSOR

Marks the end of an accessor entry.

```
END_ACCESSOR( )
```

Remarks

For multiple accessors on a rowset, you need to specify **BEGIN_ACCESSOR_MAP** and use the **BEGIN_ACCESSOR** macro for each individual accessor. The **BEGIN_ACCESSOR** macro is completed with the **END_ACCESSOR** macro. The **BEGIN_ACCESSOR_MAP** macro is completed with the **END_ACCESSOR_MAP** macro.

Example

See BEGIN_ACCESSOR_MAP.

See Also

Reference

BEGIN_ACCESSOR_MAP
BEGIN_ACCESSOR
END_ACCESSOR_MAP

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

END_ACCESSOR_MAP

Marks the end of the accessor map entries.

```
END_ACCESSOR_MAP( )
```

Remarks

For multiple accessors on a rowset, you need to specify **BEGIN_ACCESSOR_MAP** and use the **BEGIN_ACCESSOR** macro for each individual accessor. The **BEGIN_ACCESSOR** macro is completed with the **END_ACCESSOR** macro. The **BEGIN_ACCESSOR_MAP** macro is completed with the **END_ACCESSOR_MAP** macro.

Example

See BEGIN_ACCESSOR_MAP.

See Also

Reference

BEGIN_ACCESSOR_MAP
BEGIN_ACCESSOR

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

END_COLUMN_MAP

Marks the end of the column map entries.

```
END_COLUMN_MAP( )
```

Remarks

It is used with a single accessor on a rowset. The **BEGIN_COLUMN_MAP** macro is completed with the **END_COLUMN_MAP** macro.

Example

See BEGIN_COLUMN_MAP.

See Also

Reference

BEGIN_COLUMN_MAP
COLUMN_ENTRY
COLUMN_ENTRY_EX

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

END_PARAM_MAP

Marks the end of the parameter map entries.

```
END_PARAM_MAP( )
```

Example

See BEGIN_PARAM_MAP and see the ConfirmUser Sample.

See Also

Reference

BEGIN_PARAM_MAP

SET_PARAM_TYPE

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

SET_PARAM_TYPE

Specifies **COLUMN_ENTRY** macros that follow the **SET_PARAM_TYPE** macro input, output, or input/output.

```
SET_PARAM_TYPE(type )
```

Parameters

type

[in] The type to set for the parameter.

Remarks

Providers support only parameter input/output types that are supported by the underlying data source. The type is a combination of one or more **DBPARAMIO** values (see DBBINDING Structures in the *OLE DB Programmer's Reference*):

- **DBPARAMIO_NOTPARAM** The accessor has no parameters. Typically, you set **eParamIO** to this value in row accessors to remind the user that parameters are ignored.
- **DBPARAMIO_INPUT** An input parameter.
- **DBPARAMIO_OUTPUT** An output parameter.
- **DBPARAMIO_INPUT | DBPARAMIO_OUTPUT** The parameter is both an input and an output parameter.

Example

```
class CGetProperty
{
public:
   LONG    m_nReturn
   LONG   m_nAge;
   TCHAR m_name[31];
   TCHAR m_property[65];

BEGIN_PARAM_MAP( CGetProperty )
   SET_PARAM_TYPE( DBPARAMIO_OUTPUT )
   COLUMN_ENTRY( 1, m_nReturn )
   SET_PARAM_TYPE( DBPARAMIO_INPUT )
   COLUMN_ENTRY( 2, m_nAge )
END_PARAM_MAP( )

BEGIN_COLUMN_MAP( CGetProperty )
   COLUMN_ENTRY( 1, m_name )
   COLUMN_ENTRY( 2, m_property )
END_COLUMN_MAP( )

};
```

For another example, see the ConfirmUser Sample.

See Also

Other Resources

Macros and Global Functions for OLE DB Consumer Templates

OLE DB Provider Templates Reference

The classes and interfaces for the OLE DB Provider Templates can be grouped into the following categories. The reference material also includes information about the macros for OLE DB Provider Templates.

The classes use the following naming convention: a class named with the pattern **IWidgetImpl** would provide an implementation of the interface **IWidget**.

Session Classes

IDBCreateSessionImpl

Creates a new session from the data source object and returns the requested interface on the newly created session. Mandatory interface on data source objects.

ISessionPropertiesImpl

Implements session properties by calling a static function defined by the property set map. The property set map should be specified in your session class. Mandatory interface on sessions.

Rowset Classes

CRowsetImpl

Provides a standard OLE DB rowset implementation without requiring multiple inheritance of many implementation interfaces. The only method for which you must provide implementation is **Execute**.

CSimpleRow

Provides a default implementation for the row handle, which is used in the **IRowsetImpl** class. A row handle is logically a unique tag for a result row. **IRowsetImpl** creates a new **CSimpleRow** for every row requested in **IRowsetImpl::GetNextRows**.

IAccessorImpl

OLE DB requires providers to implement an **HACCESSOR**, which is a tag to an array of **DBBINDING** structures. Provides **HACCESSOR**s that are addresses of the **BindType** structures. Mandatory on rowsets and commands.

IColumnsInfoImpl

Delegates to a static function defined by the provider column map. Mandatory interface on rowsets and commands.

IConvertTypeImpl

Gives information on the availability of type conversions on a command or on a rowset. Mandatory on commands, rowsets, and index rowsets. Implements the **IConvertType** interface by delegating to the conversion object supplied by OLE DB.

IDBSchemaRowsetImpl

Implements the **IDBSchemaRowset** interface and the templatized creator function **CreateSchemaRowset**.

IOpenRowsetImpl

Opens and returns a rowset that includes all rows from a single base table or index. Mandatory interface for a session object.

IRowsetChangeImpl

Implements the OLE DB IRowsetChange interface, which enables updating of the values of columns in existing rows, deleting rows, and inserting new rows.

IRowsetCreatorImpl

This class inherits from IObjectWithSite and overrides IObjectWithSite::SetSite. **IRowsetCreatorImpl** performs the same functions as **IObjectWithSite** but also enables the OLE DB properties **DBPROPCANSCROLLBACKWARDS** and **DBPROPCANFETCHBACKWARDS**.

IRowsetIdentityImpl

Implements the **IRowsetIdentity** interface, which allows you to compare whether two rows of data are identical or not.

IRowsetImpl

Provides an implementation of the **IRowset** interface, which is the base rowset interface.

IRowsetInfoImpl

Implements the rowset properties by using the property set map defined in your command class. Mandatory interface on rowsets.

IRowsetLocateImpl

Implements the OLE DB IRowsetLocate interface, which fetches arbitrary rows from a rowset. To support OLE DB bookmarks in a rowset, make the rowset inherit from this class.

IRowsetNotifyCP

Implements broadcast functions to advise listeners on the connection point **IID_IRowsetNotify** of changes to the contents of the rowset. Consumers that handle notifications implement IRowsetNotify and register it on that connection point.

IRowsetUpdateImpl

Implements the OLE DB IRowsetUpdate interface, which enables consumers to delay the transmission of changes made with IRowsetChange to the data source and undo changes before transmission.

Command Classes

ICommandImpl

Provides an implementation of the **ICommand** interface. This interface is not visible, but is handled by **ICommandTextImpl**. A mandatory interface on the command object.

ICommandPropertiesImpl

This implementation of the **ICommandProperties** interface is provided by a static function defined by the **BEGIN_PROPSET_MAP** macro. Mandatory on commands.

ICommandTextImpl

Sets, stores, and returns the command text. Mandatory on commands.

IDBCreateCommandImpl

Creates a new command from the session object and returns the requested interface on the newly created command. Optional interface on session objects.

Other command classes are **IColumnsInfoImpl** and **IAccessorImpl**, described in the Rowset Classes section above.

Data Source Classes

IDBInitializeImpl

Creates and deletes the connection with the consumer. Mandatory interface on data source objects and optional interface on enumerators.

IDBPropertiesImpl

IDBProperties is a mandatory interface for data source objects and an optional interface for enumerators. However, if an enumerator exposes **IDBInitialize**, it must expose **IDBProperties** (properties on the data source).

IGetDataSourceImpl

Obtains an interface pointer to the data source object. Mandatory interface on the session.

Other Classes

CUtlProps

Implements properties for a variety of OLE DB property interfaces (for example, **IDBProperties**, **ISessionProperties**, and **IRowsetInfo**).

IErrorRecordsImpl

Implements the OLE DB IErrorRecords interface, adding records to and retrieving records from a data member.

See Also

Concepts

OLE DB Consumer Templates Reference

Other Resources

CRowsetImpl Class

Provides a standard OLE DB rowset implementation without requiring multiple inheritance of many implementation interfaces.

```
template <
   class T,
   class Storage,
   class CreatorClass,
   class ArrayType = CAtlArray<Storage>,
   class RowClass = CSimpleRow,
   class RowsetInterface = IRowsetImpl < T, IRowset >
>
class CRowsetImpl :
   public CComObjectRootEx<CreatorClass::_ThreadModel>,
   public CRowsetBaseImpl<T, Storage, ArrayType, RowsetInterface>,
   public IRowsetInfoImpl<T, CreatorClass::_PropClass>
```

Parameters

T

The user's class that derives from **CRowsetImpl**.

Storage

The user record class.

CreatorClass

The class that contains properties for the rowset; typically the command.

ArrayType

The class that will act as storage for the rowset's data. This parameter defaults to **CAtlArray**, but it can be any class that supports the required functionality.

Remarks

CRowsetImpl provides overrides in the form of static upcasts. The methods control the manner in which a given rowset will validate command text. You can create your own **CRowsetImpl**-style class by making your implementation interfaces multiple-inherited. The only method for which you must provide implementation is **Execute**. Depending on what type of rowset you are creating, the creator methods will expect different signatures for **Execute**. For example, if you are using a **CRowsetImpl**-derived class to implement a schema rowset, the **Execute** method will have the following signature:

```
HRESULT Execute(LONG* pcRows, ULONG cRestrictions, const VARIANT* rgRestrictions)
```

If you are creating a **CRowsetImpl**-derived class to implement a command or session's rowset, the **Execute** method will have the following signature:

```
HRESULT Execute(LONG* pcRows, DBPARAMS* pParams)
```

To implement any of the **CRowsetImpl**-derived **Execute** methods, you must populate your internal data buffers (m_rgRowData).

Requirements

Header: atldb.h

See Also

Other Resources

CRowsetImpl Members

CRowsetImpl Members

Methods

NameFromDBID	Extracts a string from a **DBID** and copies it to the *bstr* passed in.
SetCommandText	Validates and stores the **DBID**s in the two strings (m_strCommandText and m_strIndexText).

Overridable Methods

GetColumnInfo	Retrieves column information for a particular client request.
GetCommandFromID	Checks to see if either or both parameters contain string values, and if so, copies the string values to the data members m_strCommandText and m_strIndexText.
ValidateCommandID	Checks to see if either or both **DBID**s contain string values, and if so, copies them to its data members m_strCommandText and m_strIndexText.

Data Members

m_rgRowData	By default, a **CAtlArray** that templatizes on the user record template argument to **CRowsetImpl**. Another array type class can be used by changing the *ArrayType* template argument to **CRowsetImpl**.
m_strCommandText	Contains the rowset's initial command.
m_strIndexText	Contains the rowset's initial index.

See Also

Reference

CRowsetImpl Class

CRowsetImpl Methods

For information about the methods in **CRowsetImpl**, see CRowsetImpl Members.

CRowsetImpl::GetColumnInfo

Retrieves column information for a particular client request.

```
static ATLCOLUMNINFO* CRowsetBaseImpl::GetColumnInfo(
   T* pv,
   ULONG* pcCols
);
```

Parameters

pv

[in] A pointer to the user's **CRowsetImpl** derived class.

pcCols

[in] A pointer (output) to the number of columns returned.

Return Value

A pointer to a static **ATLCOLUMNINFO** structure.

Remarks

This method is an advanced override.

This method is called by several base implementation classes to retrieve column information for a particular client request. Usually, this method would be called by **IColumnsInfoImpl**. If you override this method, you must place a version of the method in your **CRowsetImpl**-derived class. Because the method may be placed in a non-templatized class, you must change *pv* to the appropriate **CRowsetImpl**-derived class.

The following example demonstrates **GetColumnInfo's** usage. In this example, **CMyRowset** is a **CRowsetImpl**-derived class. In order to override **GetColumnInfo** for all instances of this class, place the following method in the **CMyRowset** class definition:

```
static ATLCOLUMNINFO* GetColumnInfo(CMyRowset* pRowset, ULONG* pcCols)
{
   ...
}
```

See Also

Reference

CRowsetImpl Class

Other Resources

CRowsetImpl Members

CRowsetImpl::GetCommandFromID

Checks to see if either or both parameters contain string values, and if so, copies the string values to the data members m_strCommandText and m_strIndexText.

```
HRESULT CRowsetBaseImpl::GetCommandFromID(
   DBID* pTableID,
   DBID* pIndexID
);
```

Parameters

pTableID

[in] A pointer to the **DBID** representing the Table ID.

pIndexID

[in] A pointer to the **DBID** representing the Index ID.

Return Value

A standard **HRESULT**.

Remarks

This method is called through a static upcast by **CRowsetImpl** to populate the data members m_strCommandText and m_strIndexText. By default, this method checks to see if either or both parameters contain string values. If they contain string values, this method copies the string values to the data members. By placing a method with this signature in your **CRowsetImpl**-derived class, your method will be called instead of the base implementation.

See Also

Reference

CRowsetImpl Class

CRowsetImpl::SetCommandText

Other Resources

CRowsetImpl Members

CRowsetImpl::NameFromDBID

Extracts a string from a **DBID** and copies it to the *bstr* passed in.

```
HRESULT CRowsetBaseImpl::NameFromDBID(
   DBID* pDBID,
   CComBSTR& bstr,
   bool bIndex
);
```

Parameters

pDBID

[in] A pointer to the **DBID** from which to extract a string.

bstr

[in] A CComBSTR reference to place a copy of the **DBID** string.

bIndex

[in] **true** if an index **DBID**; **false** if a table **DBID**.

Return Value

A standard **HRESULT**. Depending on whether the **DBID** is a table or an index (denoted by *bIndex*), the method will either return **DB_E_NOINDEX** or **DB_E_NOTABLE**.

Remarks

This method is called by the **CRowsetImpl** implementations of ValidateCommandID and GetCommandFromID.

See Also

Reference

CRowsetImpl Class

Other Resources

CRowsetImpl Members

CRowsetImpl::SetCommandText

Validates and stores the **DBID**s in the two strings (m_strCommandText and m_strIndexText).

```
HRESULT CRowsetBaseImpl::SetCommandText(
   DBID* pTableID,
   DBID* pIndexID
);
```

Parameters

pTableID

[in] A pointer to the **DBID** representing the table ID.

pIndexID

[in] A pointer to the **DBID** representing the index ID.

Return Value

A standard **HRESULT**.

Remarks

The **SetCommentText** method is called by **CreateRowset**, a static templatized method of **IOpenRowsetImpl**.

This method delegates its work by calling ValidateCommandID and GetCommandFromID through an upcasted pointer.

See Also

Reference

CRowsetImpl Class

Other Resources

CRowsetImpl Members

CRowsetImpl::ValidateCommandID

Checks to see if either or both **DBID**s contain string values, and if so, copies them to its data members m_strCommandText and m_strIndexText.

```
HRESULT CRowsetBaseImpl::ValidateCommandID(
   DBID* pTableID,
   DBID* pIndexID
);
```

Parameters

pTableID

[in] A pointer to the **DBID** representing the table ID.

pIndexID

[in] A pointer to the **DBID** representing the index ID.

Return Value

A standard **HRESULT**.

Remarks

This method is called through a static upcast by **CRowsetImpl** to populate its data members m_strCommandText and m_strIndexText. By default, this method checks to see if either or both **DBID**s contain string values, and if so, copies them to its data members. By placing a method with this signature in your **CRowsetImpl**-derived class, your method will be called instead of the base implementation.

See Also

Reference

CRowsetImpl Class

CRowsetImpl::SetCommandText

Other Resources

CRowsetImpl Members

CRowsetImpl Data Members

For information about the data members in **CRowsetImpl**, see CRowsetImpl Members.

CRowsetImpl::m_rgRowData

By default, a **CAtlArray** that templatizes on the user record template argument to **CRowsetImpl**.

```
ArrayType CRowsetBaseImpl::m_rgRowData;
```

Remarks

ArrayType is a template parameter to **CRowsetImpl**.

See Also

Reference

CRowsetImpl Class
CRowsetImpl::m_strCommandText
CRowsetImpl::m_strIndexText

Other Resources

CRowsetImpl Members

CRowsetImpl::m_strCommandText

Contains the rowset's initial command.

```
CComBSTR CRowsetBaseImpl::m_strCommandText;
```

See Also

Reference

CRowsetImpl Class
CRowsetImpl::m_strIndexText
CRowsetImpl::m_rgRowData
CRowsetImpl::GetCommandFromID
CRowsetImpl::SetCommandText
CRowsetImpl::ValidateCommandID

Other Resources

CRowsetImpl Members

CRowsetImpl::m_strIndexText

Contains the rowset's initial index.

```
CComBSTR CRowsetBaseImpl::m_strIndexText;
```

See Also

Reference

CRowsetImpl Class
CRowsetImpl::m_strCommandText
CRowsetImpl::m_rgRowData
CRowsetImpl::GetCommandFromID
CRowsetImpl::SetCommandText
CRowsetImpl::ValidateCommandID

Other Resources

CRowsetImpl Members

CSimpleRow Class

Provides a default implementation for the row handle, which is used in the IRowsetImpl class.

```
class CSimpleRow
```

Remarks

A row handle is logically a unique tag for a result row. **IRowsetImpl** creates a new **CSimpleRow** for every row requested in IRowsetImpl::GetNextRows. **CSimpleRow** can also be replaced with your own implementation of the row handle, as it is a default template argument to **IRowsetImpl**. The only requirement to replacing this class is to have the replacement class provide a constructor that accepts a single parameter of type **LONG**.

Requirements

Header: atldb.h

See Also

Reference

IRowsetImpl Class

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

CSimpleRow Members

CSimpleRow Members

Methods

AddRefRow	Adds a reference count to an existing row handle.
Compare	Compares two rows to see if they refer to the same row instance.
CSimpleRow	The constructor.
ReleaseRow	Releases rows.

Data Members

m_dwRef	Reference count to an existing row handle.
m_iRowset	An index to the rowset representing the cursor.

See Also

Reference

CSimpleRow Class

CSimpleRow Methods

For information about the methods in **CSimpleRow**, see CSimpleRow Members.

CSimpleRow::CSimpleRow

The constructor.

```
CSimpleRow(
   DBCOUNTITEM iRowsetCur
);
```

Parameters

iRowsetCur

[in] Index to the current rowset.

Remarks

Sets m_iRowset to *iRowsetCur*.

See Also

Reference

CSimpleRow Class

Other Resources

CSimpleRow Members

CSimpleRow::AddRefRow

Adds a reference count to an existing row handle in a thread-safe manner.

```
DWORD AddRefRow( );
```

See Also

Reference

CSimpleRow Class
CSimpleRow::ReleaseRow
IRowsetImpl::AddRefRows

Other Resources

CSimpleRow Members

CSimpleRow::Compare

Compares two rows to see if they refer to the same row instance.

```
HRESULT Compare(
   CSimpleRow* pRow
);
```

Parameters

pRow

A pointer to a **CSimpleRow** object.

Return Value

An **HRESULT** value, usually **S_OK**, indicating the two rows are the same row instance, or **S_FALSE**, indicating the two rows are different. See IRowsetIdentity::IsSameRow in the *OLE DB Programmer's Reference* for other possible return values.

See Also

Reference

CSimpleRow Class
CSimpleRow::ReleaseRow
IRowsetImpl::RefRows

Other Resources

CSimpleRow Members

CSimpleRow::ReleaseRow

Releases rows in a thread-safe manner.

```
DWORD ReleaseRow( );
```

See Also

Reference

CSimpleRow Class
CSimpleRow::AddRefRow
IRowsetImpl::RefRows

Other Resources

CSimpleRow Members

CSimpleRow Data Members

For information about the data members in **CSimpleRow**, see CSimpleRow Members.

CSimpleRow::m_dwRef

Reference count to an existing row handle.

```
DWORD m_dwRef;
```

See Also

Reference

CSimpleRow Class
CSimpleRow::AddRefRow
CSimpleRow::ReleaseRow

Other Resources

CSimpleRow Members

CSimpleRow::m_iRowset

Index to the rowset representing the cursor.

```
KeyType m_iRowset;
```

See Also

Reference

CSimpleRow Class

CSimpleRow::CSimpleRow

Other Resources

CSimpleRow Members

CUtlProps Class

Implements properties for a variety of OLE DB property interfaces (for example, **IDBProperties**, **IDBProperties**, and **IRowsetInfo**).

```
template < class T >
class ATL_NO_VTABLE CUtlProps : public CUtlPropsBase
```

Parameters

T

The class that contains the **BEGIN_PROPSET_MAP**.

Remarks

Most of this class is an implementation detail.

CUtlProps contains two members for setting properties internally: GetPropValue and SetPropValue.

For more information on the macros used in a property set map, see BEGIN_PROPSET_MAP and END_PROPSET_MAP.

Requirements

Header: atldb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

CUtlProps Members

CUtlProps Members

Methods

GetPropValue	Gets a property from a property set.
IsValidValue	Used to validate a value before setting a property.
OnInterfaceRequested	Handles requests for an optional interface when a consumer calls a method on an object creation interface.
OnPropertyChanged	Called after setting a property to handle chained properties.
SetPropValue	Sets a property in a property set.

See Also

Reference

CUtlProps Class

CUtlProps Methods

For information about the methods in **CUtlProps**, see CUtlProps Members.

CUtlProps::GetPropValue

Gets a property from a property set.

```
OUT_OF_LINE HRESULT GetPropValue(
   const GUID* pguidPropSet,
   DBPROPID dwPropId,
   VARIANT* pvValue
);
```

Parameters

pguidPropSet

[in] The GUID for the PropSet.

dwPropId

[in] The property index.

pvValue

[out] A pointer to a variant that contains the new property value.

Return Value

Failure on failure and **S_OK** if successful.

See Also

Reference

CUtlProps Class

Other Resources

CUtlProps Members

CUtlProps::IsValidValue

Used to validate a value before setting a property.

```
virtual HRESULT CUtlPropsBase::IsValidValue(
   ULONG /* iCurSet */,
   DBPROP* pDBProp
);
```

Parameters

iCurSet

The index into the property-set array; zero if there is only one property set.

pDBProp

The property ID and new value in a DBPROP structure.

Return Value

A standard **HRESULT**. The default return value is **S_OK**.

Remarks

If you have any validation routines you want to run on a value that you are about to use to set a property, you should override this function. For example, you could validate **DBPROP_AUTH_PASSWORD** against a password table to determine a valid value.

See Also

Reference

CUtlProps Class

Other Resources

CUtlProps Members

CUtlProps::OnInterfaceRequested

Handles requests for an optional interface when a consumer calls a method on one of the object creation interfaces.

```
virtual HRESULT CUtlPropsBase::OnInterfaceRequested(
   REFIID riid
);
```

Parameters

riid

[in] The IID for the requested interface. For more details, see the description of the *riid* parameter of **ICommand::Execute** in the *OLE DB Programmer's Reference* (in the *MDAC SDK*).

Remarks

OnInterfaceRequested handles consumer requests for an optional interface when a consumer calls a method on one of the object creation interfaces (such as **IDBCreateSession**, **IDBCreateCommand**, **IOpenRowset**, or **ICommand**). It sets the corresponding OLE DB property for the requested interface. For example, if the consumer requests **IID_IRowsetLocate**, **OnInterfaceRequested** sets the **DBPROP_IRowsetLocate** interface. Doing so maintains the correct state during rowset creation.

This method is called when the consumer calls **IOpenRowset::OpenRowset** or **ICommand::Execute**.

If a consumer opens an object and requests an optional interface, the provider should set the property associated with that interface to **VARIANT_TRUE**. To allow property-specific processing, **OnInterfaceRequested** is called before the provider's **Execute** method is called. By default, **OnInterfaceRequested** handles the following interfaces:

- **IRowsetLocate**
- **IRowsetChange**
- **IRowsetUpdate**
- **IConnectionPointContainer**
- **IRowsetScroll**

If you wish to handle other interfaces, override this function in your data source, session, command, or rowset class to process functions. Your override should go through the normal set/get properties interfaces to ensure that setting properties also sets any chained properties (see OnPropertyChanged).

See Also

Reference

CUtlProps Class

Other Resources

CUtlProps Members

CUtlProps::OnPropertyChanged

Called after setting a property to handle chained properties.

```
virtual HRESULT OnPropertyChanged(
   ULONG /* iCurSet */,
   DBPROP* pDBProp
);
```

Parameters

iCurSet

The index into the property-set array; zero if there is only one property set.

pDBProp

The property ID and new value in a DBPROP structure.

Return Value

A standard **HRESULT**. The default return value is **S_OK**.

Remarks

If you want to handle chained properties, such as bookmarks or updates whose values are dependent on another property's value, you should override this function.

Example

In this function, the user gets the property ID from the `DBPROP*` parameter. Now, it is possible to compare the ID against a property to chain. When the property is found, `SetProperties` is called with the property that will now be set in conjunction with the other property. In this case, if one gets the `DBPROP_IRowsetLocate`, `DBPROP_LITERALBOOKMARKS`, or `DBPROP_ORDEREDBOOKMARKS` property, one can set the `DBPROP_BOOKMARKS` property.

```
HRESULT OnPropertyChanged(ULONG iCurSet, DBPROP* pDBProp)
{
   ATLASSERT(pDBProp != NULL);

   DWORD dwPropertyID = pDBProp->dwPropertyID;

   if (dwPropertyID == DBPROP_IRowsetLocate ||
      dwPropertyID == DBPROP_LITERALBOOKMARKS ||
      dwPropertyID == DBPROP_ORDEREDBOOKMARKS)
   {
      CComVariant var = pDBProp->vValue;

      if (var.boolVal == VARIANT_TRUE)
      {
         // Set the bookmarks property as these are chained
         CComVariant bookVar(true);
         CDBPropSet set(DBPROPSET_ROWSET);
         set.AddProperty(DBPROP_BOOKMARKS, bookVar);

         return SetProperties(1, &set);
      }
   }

   return S_OK;
}
```

See Also

Reference

CUtlProps Class

Other Resources

CUtlProps::SetPropValue

Sets a property in a property set.

```
HRESULT SetPropValue(
   const GUID* pguidPropSet,
   DBPROPID dwPropId,
   VARIANT* pvValue
);
```

Parameters

pguidPropSet

[in] The GUID for the PropSet.

dwPropId

[in] The property index.

pvValue

[in] A pointer to a variant that contains the new property value.

Return Value

Failure on failure and **S_OK** if successful.

See Also

Reference

CUtlProps Class

Other Resources

CUtlProps Members

IAccessorImpl Class

Provides an implementation of the IAccessor interface.

```
template <
   class T,
   class BindType = ATLBINDINGS,
   class BindingVector = CAtlMap <
      HACCESSOR hAccessor,
            BindType* pBindingsStructure
   >
>
class ATL_NO_VTABLE IAccessorImpl : public IAccessorImplBase<BindType>
```

Parameters

T

Your rowset or command object class.

BindType

Storage unit for binding information. The default is the **ATLBINDINGS** structure (see atldb.h).

BindingVector

Storage unit for column information. The default is CAtlMap where the key element is an **HACCESSOR** value and the value element is a pointer to a *BindType* structure.

Remarks

This is mandatory on rowsets and commands. OLE DB requires providers to implement an **HACCESSOR**, which is a tag to an array of DBBINDING structures. **HACCESSOR**s provided by **IAccessorImpl** are addresses of the *BindType* structures. By default, *BindType* is defined as an **ATLBINDINGS** in **IAccessorImpl**'s template definition. *BindType* provides a mechanism used by **IAccessorImpl** to track the number of elements in its **DBBINDING** array as well as a reference count and accessor flags.

Requirements

Header: atldb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

IAccessorImpl Members

IAccessorImpl Members

Methods

IAccessorImpl	The constructor.

Interface Methods

AddRefAccessor	Adds a reference count to an existing accessor.
CreateAccessor	Creates an accessor from a set of bindings.
GetBindings	Returns the bindings in an accessor.
ReleaseAccessor	Releases an accessor.

See Also

Reference

IAccessorImpl Class

IAccessorImpl Methods

For information about the methods in **IAccessorImpl**, see IAccessorImpl Members.

IAccessorImpl::AddRefAccessor

Adds a reference count to an existing accessor.

```
STDMETHOD(AddRefAccessor)(
   HACCESSOR hAccessor,
   DBREFCOUNT* pcRefCount
);
```

Parameters

See IAccessor::AddRefAccessor in the *OLE DB Programmer's Reference*.

See Also

Reference

IAccessorImpl Class
IAccessorImpl::CreateAccessor
IAccessorImpl::ReleaseAccessor

Other Resources

IAccessorImpl Members

IAccessorImpl::CreateAccessor

Creates an accessor from a set of bindings.

```
STDMETHOD(CreateAccessor)(
   DBACCESSORFLAGS dwAccessorFlags,
   DBCOUNTITEM cBindings,
   const DBBINDING rgBindings[],
   DBLENGTH cbRowSize,
   HACCESSOR* phAccessor,
   DBBINDSTATUS rgStatus[]
);
```

Parameters

See IAccessor::CreateAccessor in the *OLE DB Programmer's Reference*.

See Also

Reference

IAccessorImpl Class
IAccessorImpl::AddRefAccessor
IAccessorImpl::ReleaseAccessor

Other Resources

IAccessorImpl Members

IAccessorImpl::GetBindings

Returns the basic columns bindings from the consumer in an accessor.

```
STDMETHOD(GetBindings)(
   HACCESSOR hAccessor,
   DBACCESSORFLAGS* pdwAccessorFlags,
   DBCOUNTITEM* pcBindings,
   DBBINDING** prgBindings
);
```

Parameters

See IAccessor::GetBindings in the *OLE DB Programmer's Reference*.

See Also

Reference

IAccessorImpl Class

Other Resources

IAccessorImpl Members

IAccessorImpl::IAccessorImpl

The constructor.

```
IAccessorImpl( );
```

See Also

Reference

IAccessorImpl Class

Other Resources

IAccessorImpl Members

IAccessorImpl::ReleaseAccessor

Releases an accessor.

```
STDMETHOD(ReleaseAccessor)(
   HACCESSOR hAccessor,
   DBREFCOUNT* pcRefCount
);
```

Parameters

See IAccessor::ReleaseAccessor in the *OLE DB Programmer's Reference*.

See Also

Reference

IAccessorImpl Class
IAccessorImpl::CreateAccessor
IAccessorImpl::AddRefAccessor

Other Resources

IAccessorImpl Members

IColumnsInfoImpl Class

Provides an implementation of the IColumnsInfo interface.

```
template <class T>
class ATL_NO_VTABLE IColumnsInfoImpl :
   public IColumnsInfo,
   public CDBIDOps
```

Parameters

T

Your class, derived from **IColumnsInfoImpl**.

Remarks

A mandatory interface on rowsets and commands. To modify the behavior of your provider's **IColumnsInfo** implementation, you need to modify the provider column map.

Requirements

Header: atldb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

IColumnsInfoImpl Members

IColumnsInfoImpl Members

Methods

GetColumnInfo	Returns the column metadata needed by most consumers.
MapColumnIDs	Returns an array of ordinals of the columns in a rowset that are identified by the specified column IDs.

See Also

Reference

IColumnsInfoImpl Class

IColumnsInfoImpl Methods

For information about the methods in **IColumnsInfoImpl**, see IColumnsInfoImpl Members.

IColumnsInfoImpl::GetColumnInfo

Returns the column metadata needed by most consumers.

```
STDMETHOD (GetColumnInfo)(
   DBORDINAL* pcColumns,
   DBCOLUMNINFO** prgInfo,
   OLECHAR** ppStringsBuffer
);
```

Parameters

See IColumnsInfo::GetColumnInfo in the *OLE DB Programmer's Reference*.

See Also

Reference

IColumnsInfoImpl Class

IColumnsInfoImpl::MapColumnIDs

Other Resources

IColumnsInfoImpl Members

IColumnsInfoImpl::MapColumnIDs

Returns an array of ordinals of the columns in a rowset that are identified by the specified column IDs.

```
STDMETHOD (MapColumnIDs)(
   DBORDINAL cColumnIDs,
   const DBID rgColumnIDs[],
   DBORDINAL rgColumns[]
);
```

Parameters

See IColumnsInfo::MapColumnIDs in the *OLE DB Programmer's Reference*.

See Also

Reference

IColumnsInfoImpl Class

IColumnsInfoImpl::GetColumnInfo

Other Resources

IColumnsInfoImpl Members

ICommandImpl Class

Provides implementation for the ICommand interface.

```
template <class T, class CommandBase = ICommand>
class ATL_NO_VTABLE ICommandImpl : public CommandBase
```

Parameters

T

Your class, derived from **ICommandImpl**.

CommandBase

A command interface. The default is **ICommand**.

Remarks

A mandatory interface on the command object.

Requirements

Header: atldb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

ICommandImpl Members

ICommandImpl Members

Methods

CancelExecution	Cancels the current command execution.
Cancel	Cancels the current command execution.
CreateRowset	Creates a rowset object.
Execute	Executes the command.
GetDBSession	Returns an interface pointer to the session that created the command.
ICommandImpl	The constructor.

Data Members

m_bCancel	Indicates whether the command is to be canceled.
m_bCancelWhenExecuting	Indicates whether the command is to be canceled when executing.
m_bIsExecuting	Indicates whether the command is currently executing.

See Also

Reference

ICommandImpl Class

ICommandImpl Methods

For information about the methods in **ICommandImpl**, see ICommandImpl Members.

ICommandImpl::Cancel

Cancels the current command execution.

```
STDMETHOD(Cancel)( );
```

Remarks

See ICommand::Cancel in the *OLE DB Programmer's Reference*.

See Also

Reference

ICommandImpl Class

ICommandImpl::CancelExecution

Other Resources

ICommandImpl Members

ICommandImpl::CancelExecution

Cancels the current command execution.

```
HRESULT CancelExecution( );
```

See Also

Reference

ICommandImpl Class
ICommandImpl::Execute
ICommandImpl::Cancel

Other Resources

ICommandImpl Members

ICommandImpl::CreateRowset

Called by Execute to create a single rowset.

```
template <class RowsetClass>
HRESULT CreateRowset(
   IUnknown* pUnkOuter,
   REFIID riid,
   DBPARAMS* pParams,
   DBROWCOUNT* pcRowsAffected,
   IUnknown** ppRowset,
   RowsetClass*& pRowsetObj
);
```

Parameters

RowsetClass

A template class member representing the user's rowset class. Usually generated by the wizard.

pUnkOuter

[in] A pointer to the controlling **IUnknown** interface if the rowset is being created as part of an aggregate; otherwise, it is null.

riid

[in] Corresponds to *riid* in **ICommand::Execute**.

pParams

[in/out] Corresponds to *pParams* in **ICommand::Execute**.

pcRowsAffected

Corresponds to *pcRowsAffected* in **ICommand::Execute**.

ppRowset

[in/out] Corresponds to *ppRowset* in **ICommand::Execute**.

pRowsetObj

[out] A pointer to a rowset object. Typically this parameter is not used, but it can be used if you must perform more work on the rowset before passing it to a COM object. The lifetime of *pRowsetObj* is bound by *ppRowset*.

Return Value

A standard **HRESULT** value. See **ICommand::Execute** for a list of typical values.

Remarks

To create more than one rowset, or to provide your own conditions for creating different rowsets, place different calls to **CreateRowset** from within **Execute**.

See ICommand::Execute in the *OLE DB Programmer's Reference*.

See Also

Reference

ICommandImpl Class

Other Resources

ICommandImpl Members

ICommandImpl::Execute

Executes the command.

```
HRESULT Execute(
   IUnknown* pUnkOuter,
   REFIID riid,
   DBPARAMS* pParams,
   DBROWCOUNT* pcRowsAffected,
   IUnknown** ppRowset
);
```

Parameters

See ICommand::Execute in the *OLE DB Programmer's Reference*.

Remarks

The outgoing interface requested will be an interface acquired from the rowset object that this function creates.

Execute calls CreateRowset. Override the default implementation to create more than one rowset or to provide your own conditions for creating different rowsets.

See Also

Reference

ICommandImpl Class

ICommandImpl::CancelExecution

Other Resources

ICommandImpl Members

ICommandImpl::GetDBSession

Returns an interface pointer to the session that created the command.

```
STDMETHOD (GetDBSession) (
   REFIID riid,
   IUnknown** ppSession
);
```

Parameters

See ICommand::GetDBSession in the *OLE DB Programmer's Reference*.

Remarks

Useful for retrieving properties from the session.

See Also

Reference

ICommandImpl Class

Other Resources

ICommandImpl Members

ICommandImpl::ICommandImpl

The constructor.

```
ICommandImpl( );
```

See Also

Reference

ICommandImpl Class

Other Resources

ICommandImpl Members

ICommandImpl Data Members

For information about the data members in **ICommandImpl**, see ICommandImpl Members.

ICommandImpl::m_bCancel

Indicates whether the command is canceled.

```
unsigned m_bCancel:1;
```

Remarks

You can retrieve this variable in the **Execute** method of your command class and cancel as appropriate.

See Also

Reference

ICommandImpl Class

ICommandImpl::m_bCancelWhenExecuting

Other Resources

ICommandImpl Members

ICommandImpl::m_bCancelWhenExecuting

Indicates whether the command can be canceled when executing.

```
unsigned m_bCancelWhenExecuting:1;
```

Remarks

Defaults to **true** (can be canceled).

See Also

Reference

ICommandImpl Class

ICommandImpl::m_bCancel

Other Resources

ICommandImpl Members

ICommandImpl::m_bIsExecuting

Indicates whether the command is currently executing.

```
unsigned m_bIsExecuting:1;
```

Remarks

The **Execute** method of your command class can set this variable to **true**.

See Also

Reference

ICommandImpl Class

ICommandImpl::m_bCancelWhenExecuting

Other Resources

ICommandImpl Members

ICommandPropertiesImpl Class

Provides an implementation of the ICommandProperties interface.

```
template <class T, class PropClass = T>
class ATL_NO_VTABLE ICommandPropertiesImpl
   : public ICommandProperties, public CUtlProps<PropClass>
```

Parameters

T

Your class, derived from

PropClass

Your properties class.

Remarks

This is mandatory on commands. The implementation is provided by a static function defined by the BEGIN_PROPSET_MAP macro.

Requirements

Header: atldb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

ICommandPropertiesImpl Members

ICommandPropertiesImpl Members

Interface Methods

GetProperties	Returns the list of properties in the Rowset property group that are currently requested for the rowset.
SetProperties	Sets properties in the Rowset property group.

See Also

Reference

ICommandPropertiesImpl Class

ICommandPropertiesImpl Methods

For information about the methods in **ICommandPropertiesImpl**, see ICommandPropertiesImpl Members.

ICommandPropertiesImpl::GetProperties

Returns all the requested property sets using the command's property map.

```
STDMETHOD(GetProperties)(
   const ULONG cPropertyIDSets,
   const DBPROPIDSET rgPropertyIDSets[],
   ULONG * pcPropertySets,
   DBPROPSET ** prgPropertySets
);
```

Parameters

See ICommandProperties::GetProperties in the *OLE DB Programmer's Reference*.

Remarks

See BEGIN_PROPSET_MAP.

See Also

Reference

ICommandPropertiesImpl Class

ICommandPropertiesImpl::SetProperties

Other Resources

ICommandPropertiesImpl Members

ICommandPropertiesImpl::SetProperties

Sets properties for the command object.

```
STDMETHOD(SetProperties)(
   ULONG cPropertySets,
   DBPROPSET rgPropertySets[]
);
```

Parameters

See ICommandProperties::SetProperties in the *OLE DB Programmer's Reference*.

See Also

Reference

ICommandPropertiesImpl Class

ICommandPropertiesImpl::GetProperties

Other Resources

ICommandPropertiesImpl Members

ICommandTextImpl Class

Provides an implementation for the ICommandText interface.

```
template <class T >
class ATL_NO_VTABLE ICommandTextImpl
   : public ICommandImpl<T, ICommandText>
```

Parameters

T

The command class derived from **ICommandTextImpl**.

Remarks

A mandatory interface on commands.

Requirements

Header: altdb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

ICommandTextImpl Members

ICommandTextImpl Members

Interface Methods

GetCommandText	Returns the text command set by the last call to SetCommandText.
SetCommandText	Sets the command text, replacing the existing command text.

Data Members

m_strCommandText	Stores the command text.

See Also

Reference

ICommandTextImpl Class

ICommandTextImpl Methods

For information about the methods in **ICommandTextImpl**, see ICommandTextImpl Members.

ICommandTextImpl::GetCommandText

Returns the text command set by the last call to SetCommandText.

```
STDMETHOD(GetCommandText)(
   GUID * pguidDialect,
   LPOLESTR * ppwszCommand
);
```

Parameters

See ICommandText::GetCommandText in the *OLE DB Programmer's Reference*. The *pguidDialect* parameter is ignored by default.

See Also

Reference

ICommandTextImpl Class

Other Resources

ICommandTextImpl Members

ICommandTextImpl::SetCommandText

Sets the command text, replacing the existing command text.

```
STDMETHOD(SetCommandText)(
   REFGUID rguidDialect,
   LPCOLESTR pwszCommand
);
```

Parameters

See ICommandText::SetCommandText in the *OLE DB Programmer's Reference*.

See Also

Reference

ICommandTextImpl Class

ICommandTextImpl::GetCommandText

Other Resources

ICommandTextImpl Members

ICommandTextImpl Data Members

For information about the data members in **ICommandTextImpl**, see ICommandTextImpl Members.

ICommandTextImpl::m_strCommandText

Stores the command text string.

```
CComBSTR m_strCommandText;
```

See Also

Reference

ICommandTextImpl Class
ICommandTextImpl::GetCommandText
ICommandTextImpl::SetCommandText

Other Resources

ICommandTextImpl Members

IConvertTypeImpl Class

Provides an implementation of the IConvertType interface.

```
template <class T>
class ATL_NO_VTABLE IConvertTypeImpl
   : public IConvertType, public CConvertHelper
```

Parameters

T

Your class, derived from **IConvertTypeImpl**.

Remarks

This interface is mandatory on commands, rowsets, and index rowsets. **IConvertTypeImpl** implements the interface by delegating to the conversion object supplied by OLE DB.

Requirements

Header: atldb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

IConvertTypeImpl Members

IConvertTypeImpl Members

Interface Methods

CanConvert	Gives information on the availability of type conversions on a command or on a rowset.

See Also

Reference

IConvertTypeImpl Class

IConvertTypeImpl Methods

For information about the methods in **IConvertTypeImpl**, see IConvertTypeImpl Members.

IConvertTypeImpl::CanConvert

Gives information on the availability of type conversions on a command or on a rowset.

```
STDMETHOD(CanConvert)(
   DBTYPE wFromType,
   DBTYPE wToType,
   DBCONVERTFLAGS dwConvertFlags
);
```

Parameters

See IConvertType::CanConvert in the *OLE DB Programmer's Reference*.

Remarks

Uses OLE DB data conversion in `MSADC.DLL`.

See Also

Reference

IConvertTypeImpl Class

Other Resources

IConvertTypeImpl Members

IDBCreateCommandImpl Class

Provides an implementation of the IDBCreateCommand interface.

```
template <class T, class CommandClass >
class ATL_NO_VTABLE IDBCreateCommandImpl
   : public IDBCreateCommand
```

Parameters

T

The session object derived from **IDBCreateCommandImpl**.

CommandClass

Your command class.

Remarks

An optional interface on the session object to obtain a new command.

Requirements

Header: atldb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

IDBCreateCommandImpl Members

IDBCreateCommandImpl Members

Interface Methods

CreateCommand	Creates a new command.

See Also

Reference

IDBCreateCommandImpl Class

IDBCreateCommandImpl Methods

For information about the methods in **IDBCreateCommandImpl**, see IDBCreateCommandImpl Members.

IDBCreateCommandImpl::CreateCommand

Creates a new command and returns the requested interface.

```
STDMETHOD(CreateCommand)(
   IUnknown * pUnkOuter,
   REFIID riid,
   IUnknown ** ppvCommand
);
```

Parameters

See IDBCreateCommand::CreateCommand in the *OLE DB Programmer's Reference*.

Some parameters correspond to *OLE DB Programmer's Reference* parameters of different names, which are described in **IDBCreateCommand::CreateCommand**:

OLE DB Template parameters	***OLE DB Programmer's Reference* parameters**
ppvCommand	*ppCommand*

See Also

Reference

IDBCreateCommandImpl Class

Other Resources

IDBCreateCommandImpl Members

IDBCreateSessionImpl Class

Provides an implementation for the IDBCreateSession interface.

```
template <class T, class SessionClass>
class ATL_NO_VTABLE IDBCreateSessionImpl
   : public IDBCreateSession
```

Parameters

T

YOUR CLASS, DERIVED FROM

SessionClass

The session object.

Remarks

A mandatory interface on data source objects.

Requirements

Header: atldb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

IDBCreateSessionImpl Members

IDBCreateSessionImpl Members

Interface Methods

CreateSession	Creates a new session from the data source object and returns the requested interface on the newly created session.

See Also

Reference

IDBCreateSessionImpl Class

IDBCreateSessionImpl Methods

For information about the methods in **IDBCreateSessionImpl**, see IDBCreateSessionImpl Members.

IDBCreateSessionImpl::CreateSession

Creates a new session from the data source object and returns the requested interface on the newly created session.

```
STDMETHOD(CreateSession)(
   IUnknown * pUnkOuter,
   REFIID riid,
   IUnknown ** ppDBSession
);
```

Parameters

See IDBCreateSession::CreateSession in the *OLE DB Programmer's Reference*.

See Also

Reference

IDBCreateSessionImpl Class

Other Resources

IDBCreateSessionImpl Members

IDBInitializeImpl Class

Provides an implementation for the IDBInitialize interface.

```
template <class T>
class ATL_NO_VTABLE IDBInitializeImpl : public IDBInitialize
```

Parameters

T

Your class, derived from **IDBInitializeImpl**.

Remarks

A mandatory interface on data source objects and optional interface on enumerators.

Requirements

Header: atldb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

IDBInitializeImpl Members

IDBInitializeImpl Members

Methods

IDBInitializeImpl	The constructor.

Interface Methods

Initialize	Starts the provider.
Uninitialize	Stops the provider.

Data Members

m_dwStatus	Data source flags.
m_pCUtlPropInfo	A pointer to implementation of DB Properties information.

See Also

Reference

IDBInitializeImpl Class

IDBInitializeImpl Methods

For information about the methods in **IDBInitializeImpl**, see IDBInitializeImpl Members.

IDBInitializeImpl::IDBInitializeImpl

The constructor.

```
IDBInitializeImpl( );
```

Remarks

Initializes all data members.

See Also

Reference

IDBInitializeImpl Class

Other Resources

IDBInitializeImpl Members

IDBInitializeImpl::Initialize

Initializes the data source object by preparing its property support.

```
STDMETHOD(Initialize)(
   void
);
```

Remarks

See IDBInitialize::Initialize in the *OLE DB Programmer's Reference*.

See Also

Reference

IDBInitializeImpl Class

IDBInitializeImpl::Uninitialize

Other Resources

IDBInitializeImpl Members

IDBInitializeImpl::Uninitialize

Places the data source object in an uninitialized state by freeing internal resources such as the property support.

```
STDMETHOD(Uninitialize)(
   void
);
```

Remarks

See IDBInitialize::Uninitialize in the *OLE DB Programmer's Reference*.

See Also

Reference

IDBInitializeImpl Class

IDBInitializeImpl::Initialize

Other Resources

IDBInitializeImpl Members

IDBInitializeImpl Data Members

For information about the data members in **IDBInitializeImpl**, see IDBInitializeImpl Members.

IDBInitializeImpl::m_dwStatus

Data source flags.

```
DWORD m_dwStatus;
```

Remarks

These flags specify or indicate the status of various attributes for the data source object. Contains one or more of the following **enum** values:

```
enum DATASOURCE_FLAGS {
   DSF_MASK_INIT     = 0xFFFFF00F,
   DSF_PERSIST_DIRTY = 0x00000001,
   DSF_INITIALIZED   = 0x00000010,
};
```

DSF_MASK_INIT	A mask to enable restoration of the uninitialized state.
DSF_PERSIST_DIRTY	Set if data source object requires persistence (that is, if there have been changes).
DSF_INITIALIZED	Set if data source has been initialized.

See Also

Reference

IDBInitializeImpl Class
IDBInitializeImpl::IDBInitializeImpl
IDBInitializeImpl::Uninitialize

Other Resources

IDBInitializeImpl Members

IDBInitializeImpl::m_pCUtlPropInfo

A pointer to implementation object for DB Properties information.

```
CUtlPropInfo<T>* m_pCUtlPropInfo;
```

See Also

Reference

IDBInitializeImpl Class

Other Resources

IDBInitializeImpl Members

IDBPropertiesImpl Class

Provides an implementation for the **IDBProperties** interface.

```
template <class T>
class ATL_NO_VTABLE IDBPropertiesImpl
   : public IDBProperties, public CUtlProps<T>
```

Parameters

T

Your class, derived from **IDBPropertiesImpl**.

Remarks

IDBProperties is a mandatory interface for data source objects and an optional interface for enumerators. However, if an enumerator exposes IDBInitialize, it must expose **IDBProperties**. **IDBPropertiesImpl** implements **IDBProperties** by using a static function defined by BEGIN_PROPSET_MAP.

Requirements

Header: atldb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

IDBPropertiesImpl Members

IDBPropertiesImpl Members

Interface Methods

GetProperties	Returns the values of properties in the Data Source, Data Source Information, and Initialization property groups that are currently set on the data source object or the values of properties in the Initialization property group that are currently set on the enumerator.
GetPropertyInfo	Returns information about all properties supported by the provider.
SetProperties	Sets properties in the Data Source and Initialization property groups, for data source objects, or the Initialization property group, for enumerators.

See Also

Reference

IDBPropertiesImpl Class

IDBPropertiesImpl Methods

For information about the methods in **IDBPropertiesImpl**, see IDBPropertiesImpl Members.

IDBPropertiesImpl::GetProperties

Returns the values of properties in the Data Source, Data Source Information, and Initialization property groups that are currently set on the data source object or the values of properties in the Initialization property group that are currently set on the enumerator.

```
STDMETHOD(GetProperties)(
   ULONG cPropertySets,
   const DBPROPIDSET rgPropertySets[],
   ULONG * pcProperties,
   DBPROPSET ** prgProperties
);
```

Parameters

See IDBProperties::GetProperties in the *OLE DB Programmer's Reference*.

Some parameters correspond to *OLE DB Programmer's Reference* parameters of different names, which are described in **IDBProperties::GetProperties**:

OLE DB Template parameters	***OLE DB Programmer's Reference* parameters**
cPropertySets	*cPropertyIDSets*
rgPropertySets	*rgPropertyIDSets*
pcProperties	*pcPropertySets*
prgProperties	*prgPropertySets*

Remarks

If the provider is initialized, this method returns the values of properties in the **DBPROPSET_DATASOURCE**, **DBPROPSET_DATASOURCEINFO**, **DBPROPSET_DBINIT** property groups that are currently set on the data source object. If the provider is not initialized, it returns **DBPROPSET_DBINIT** group properties only.

See Also

Reference

IDBPropertiesImpl Class
IDBPropertiesImpl::GetPropertyInfo
IDBPropertiesImpl::SetProperties

Other Resources

IDBPropertiesImpl Members

IDBPropertiesImpl::GetPropertyInfo

Returns property information supported by the data source.

```
STDMETHOD(GetPropertyInfo)(
   ULONG cPropertySets,
   const DBPROPIDSET rgPropertySets[],
   ULONG * pcPropertyInfoSets,
   DBPROPINFOSET ** prgPropertyInfoSets,
   OLECHAR ** ppDescBuffer
);
```

Parameters

See IDBProperties::GetPropertyInfo in the *OLE DB Programmer's Reference*.

Some parameters correspond to *OLE DB Programmer's Reference* parameters of different names, which are described in **IDBProperties::GetPropertyInfo**:

OLE DB Template parameters	***OLE DB Programmer's Reference* parameters**
cPropertySets	*cPropertyIDSets*
rgPropertySets	*rgPropertyIDSets*

Remarks

Uses IDBInitializeImpl::m_pCUtlPropInfo to implement this functionality.

See Also

Reference

IDBPropertiesImpl Class
IDBPropertiesImpl::GetProperties
IDBPropertiesImpl::SetProperties

Other Resources

IDBPropertiesImpl Members

IDBPropertiesImpl::SetProperties

Sets properties in the Data Source and Initialization property groups, for data source objects, or the Initialization property group, for enumerators.

```
STDMETHOD(SetProperties)(
   ULONG cPropertySets,
   DBPROPSET rgPropertySets[]
);
```

Parameters

See IDBProperties::SetProperties in the *OLE DB Programmer's Reference*.

Remarks

If the provider is initialized, this method sets the values of properties in the **DBPROPSET_DATASOURCE**, **DBPROPSET_DATASOURCEINFO**, **DBPROPSET_DBINIT** property groups for the data source object. If the provider is not initialized, it sets **DBPROPSET_DBINIT** group properties only.

See Also

Reference

IDBPropertiesImpl Class
IDBPropertiesImpl::GetProperties
IDBPropertiesImpl::GetPropertyInfo

Other Resources

IDBPropertiesImpl Members

IDBSchemaRowsetImpl Class

Provides implementation for schema rowsets.

```
template <class SessionClass>
class ATL_NO_VTABLE IDBSchemaRowsetImpl : public IDBSchemaRowset
```

Parameters

SessionClass

The class by which **IDBSchemaRowsetImpl** is inherited. Typically, this class will be the user's session class.

Remarks

This class implements the IDBSchemaRowset interface and the templatized creator function CreateSchemaRowset.

OLE DB uses schema rowsets to return data about the data in a provider. Such data is often called "metadata." By default, a provider must always support **DBSCHEMA_TABLES**, **DBSCHEMA_COLUMNS**, and **DBSCHEMA_PROVIDER_TYPES**, as described in IDBSchemaRowset in the *OLE DB Programmer's Reference*. Schema rowsets are designated in a schema map. For information about the schema map entries, see SCHEMA_ENTRY.

The OLE DB Provider Wizard, in the ATL Object Wizard, automatically generates code for the schema rowsets in your project. (By default, the wizard supports the mandatory schema rowsets previously mentioned.) When you create a consumer using the ATL Object Wizard, the wizard uses schema rowsets to bind the correct data to a provider. If you do not implement your schema rowsets to provide the correct metadata, the wizard will not bind the correct data.

For information on how to support schema rowsets in your provider, see Supporting Schema Rowsets.

For more information about schema rowsets, see Schema Rowsets in the *OLE DB Programmer's Reference*.

Requirements

Header: atldb.h

See Also

Reference

Schema Rowset Classes and Typedef Classes

Concepts

Supporting Schema Rowsets

Other Resources

IDBSchemaRowsetImpl Members

IDBSchemaRowsetImpl Members

Implementation Methods

CheckRestrictions	Checks the validity of restrictions against a schema rowset.
CreateSchemaRowset	Implements a COM object creator function for the object specified by the template parameter.
SetRestrictions	Specifies which restrictions you support on a particular schema rowset.

Interface Methods

GetRowset	Returns a schema rowset.
GetSchemas	Returns a list of schema rowsets accessible by IDBSchemaRowsetImpl::GetRowset.

See Also

Reference

IDBSchemaRowsetImpl Class

Schema Rowset Classes and Typedef Classes

IDBSchemaRowsetImpl Methods

For information about the methods in **IDBSchemaRowsetImpl**, see IDBSchemaRowsetImpl Members.

IDBSchemaRowsetImpl::CheckRestrictions

Checks the validity of restrictions against a schema rowset.

```
HRESULT CheckRestrictions(
   REFGUID rguidSchema,
   ULONG cRestrictions,
   const VARIANT rgRestrictions[]
);
```

Parameters

rguidSchema

[in] A reference to the requested schema rowset GUID (for example, **DBSCHEMA_TABLES**).

cRestrictions

[in] The number of restrictions that the consumer passed in for the schema rowset.

rgRestrictions

[in] An array of length *cRestrictions* of restriction values to be set. For more information, see the description of the *rgRestrictions* parameter in SetRestrictions.

Remarks

Use **CheckRestrictions** to check the validity of restrictions against a schema rowset. It checks restrictions for **DBSCHEMA_TABLES**, **DBSCHEMA_COLUMNS**, and **DBSCHEMA_PROVIDER_TYPES** schema rowsets. Call it to determine if a consumer's call to **IDBSchemaRowset::GetRowset** is correct. If you want to support schema rowsets other than those listed above, you should create your own function to carry out this task.

CheckRestrictions determines if the consumer is calling GetRowset with the correct restriction and the correct restriction type (for example, a **VT_BSTR** for a string) that the provider supports. It also determines if the correct number of restrictions are supported. By default, **CheckRestrictions** will ask the provider, through the SetRestrictions call, which restrictions it supports on a given rowset. It then compares the restrictions from the consumer against those supported by the provider and either succeeds or fails.

For more information on schema rowsets, see IDBSchemaRowset in the *OLE DB Programmer's Reference* in the *Platform SDK*.

See Also

Reference

IDBSchemaRowsetImpl Class

Schema Rowset Classes and Typedef Classes

Other Resources

IDBSchemaRowsetImpl Members

IDBSchemaRowsetImpl::CreateSchemaRowset

Implements a COM object creator function for the object specified by the template parameter.

```
template < class SchemaRowsetClass >
HRESULT CreateSchemaRowset(
   IUnknown *pUnkOuter,
   ULONG cRestrictions,
   const VARIANT rgRestrictions[],
   REFIID riid,
   ULONG cPropertySets,
   DBPROPSET rgPropertySets[],
   IUnknown** ppRowset,
   SchemaRowsetClass*& pSchemaRowset
);
```

Parameters

pUnkOuter

[in] An outer IUnknown when aggregating, otherwise **NULL**.

cRestrictions

[in] The count of restrictions applied to the schema rowset.

rgRestrictions

[in] An array of *cRestrictions* **VARIANT**s to be applied to the rowset.

riid

[in] The interface to QueryInterface for on the output **IUnknown**.

cPropertySets

[in] The number of property sets to set.

rgPropertySets

[in] An array of DBPROPSET structures that specify the properties being set.

ppRowset

[out] The outgoing **IUnknown** requested by *riid*. This **IUnknown** is an interface on the schema rowset object.

pSchemaRowset

[out] A pointer to an instance of the schema rowset class. Typically, this parameter is not used, but it can be used if you must perform more work on the schema rowset before handing it out to a COM object. The lifetime of *pSchemaRowset* is bound by *ppRowset*.

Return Value

A standard **HRESULT** value.

Remarks

This function implements a generic creator for all types of schema rowsets. Typically, the user does not call this function. It is called by the implementation of the schema map.

See Also

Reference

IDBSchemaRowsetImpl Class
SCHEMA_ENTRY
Schema Rowset Classes and Typedef Classes

Other Resources

IDBSchemaRowsetImpl Members

IDBSchemaRowsetImpl::GetRowset

Returns a schema rowset.

```
STDMETHOD (GetRowset)(
   IUnknown *pUnkOuter,
   REFGUID rguidSchema,
   ULONG cRestrictions,
   const VARIANT rgRestrictions[],
   REFIID riid,
   ULONG cPropertySets,
   DBPROPSET rgPropertySets[],
   IUnknown **ppRowset
);
```

Parameters

pUnkOuter

[in] An outer **IUnknown** when aggregating; otherwise **NULL**.

rguidSchema

[in] A reference to the requested schema rowset GUID (for example, **DBSCHEMA_TABLES**).

cRestrictions

[in] A count of restrictions to be applied to the rowset.

rgRestrictions

[in] An array of *cRestrictions* **VARIANT**s that represent the restrictions.

riid

[in] The IID to request of the newly created schema rowset.

cPropertySets

[in] The number of property sets to set.

rgPropertySets

[in/out] An array of DBPROPSET structures to set on the newly created schema rowset.

ppRowset

[out] A pointer to the requested interface on the newly created schema rowset.

Remarks

This method requires the user to have a schema map in the session class. Using the schema map information, **GetRowset** creates a given rowset object if the *rguidSchema* parameter is equal to one of the map entries GUIDs. See SCHEMA_ENTRY for a description of the map entry.

See IDBSchemaRowset::GetRowset in the *Platform SDK*.

See Also

Reference

IDBSchemaRowsetImpl Class
IDBSchemaRowsetImpl::GetSchemas
Schema Rowset Classes and Typedef Classes

Other Resources

IDBSchemaRowsetImpl Members

IDBSchemaRowsetImpl::GetSchemas

Returns a list of schema rowsets accessible by IDBSchemaRowsetImpl::GetRowset.

```
STDMETHOD ( GetSchema s )(
   ULONG * pcSchemas,
   GUID ** prgSchemas,
   ULONG** prgRest
);
```

Parameters

pcSchemas

[out] A pointer to a **ULONG** that is filled with the number of schemas.

prgSchemas

[out] A pointer to an array of GUIDs that is filled with a pointer to an array of schema rowset GUIDs.

prgRest

[out] A pointer to an array of **ULONG**s that is to be filled with the restriction array.

Remarks

This method returns an array of all schema rowsets supported by the provider. See IDBSchemaRowset::GetSchemas in the *Platform SDK*.

The implementation of this function requires the user to have a schema map in the session class. Using the schema map information, it then responds with the array of GUIDs for the schemas in the map. This represents the schemas supported by the provider.

See Also

Reference

IDBSchemaRowsetImpl Class
IDBSchemaRowsetImpl::GetRowset
IDBSchemaRowset::GetRowset
Schema Rowset Classes and Typedef Classes

Other Resources

IDBSchemaRowsetImpl Members

IDBSchemaRowsetImpl::SetRestrictions

Specifies which restrictions you support on a particular schema rowset.

```
void SetRestrictions(
   ULONG cRestrictions,
   GUID* /* rguidSchema */,
   ULONG* rgRestrictions
);
```

Parameters

cRestrictions

[in] The number of restrictions in the *rgRestrictions* array and the number of GUIDs in the *rguidSchema* array.

rguidSchema

[in] An array of the GUIDs of the schema rowsets for which to fetch restrictions. Each array element contains the GUID of one schema rowset (for example, **DBSCHEMA_TABLES**).

rgRestrictions

[in] An array of length *cRestrictions* of restriction values to be set. Each element corresponds to the restrictions on the schema rowset identified by the GUID. If a schema rowset is not supported by the provider, the element is set to zero. Otherwise, the **ULONG** value contains a bit mask that represents the restrictions supported on that schema rowset. For more information on which restrictions correspond to a particular schema rowset, consult the table of schema rowset GUIDs in IDBSchemaRowset in the *OLE DB Programmer's Reference* in the *Platform SDK*.

Remarks

The **IDBSchemaRowset** object calls **SetRestrictions** to determine which restrictions you support on a particular schema rowset (it is called by GetSchemas through an upcasted pointer). Restrictions allow consumers to fetch only matching rows (for example, find all the columns in the table "MyTable"). Restrictions are optional, and in the case in which none are supported (the default), all data is always returned.

The default implementation of this method sets the *rgRestrictions* array elements to 0. Override the default in your session class to set restrictions other than the default.

For information on implementing schema rowset support, see Supporting Schema Rowsets.

For an example of an provider that supports schema rowsets, see the UpdatePV sample.

For more information on schema rowsets, see IDBSchemaRowset in the *OLE DB Programmer's Reference* in the *Platform SDK*.

See Also

Tasks

UpdatePV Sample: Implements an Updatable OLE DB Provider

Reference

IDBSchemaRowsetImpl Class

Schema Rowset Classes and Typedef Classes

Concepts

Supporting Schema Rowsets

Other Resources

IDBSchemaRowsetImpl Members

IErrorRecordsImpl Class

Implements the OLE DB IErrorRecords interface, adding records to and retrieving records from a data member (m_rgErrors) of type **CAtlArray<***RecordClass***>**.

```
template <
   class T,
   class RecordClass = ATLERRORINFO
>
class IErrorRecordsImpl : public IErrorRecords
```

Parameters

T

A class derived from **IErrorRecordsImpl**.

RecordClass

A class that represents an OLE DB error object.

Requirements

Header: atldb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

IErrorRecordsImpl Members

IErrorRecordsImpl Members

Methods

GetErrorDescriptionString	Gets the error description string from an error record.
GetErrorGUID	Gets the error GUID from an error record.
GetErrorHelpContext	Gets the help context ID from an error record.
GetErrorHelpFile	Gets the full pathname of the help file from an error record.
GetErrorSource	Gets the error source code from an error record.

Interface Methods

AddErrorRecord	Adds a record to the OLE DB error object.
GetBasicErrorInfo	Returns basic information about the error, such as the return code and provider-specific error number.
GetCustomErrorObject	Returns a pointer to an interface on a custom error object.
GetErrorInfo	Returns an IErrorInfo interface pointer on the specified record.
GetErrorParameters	Returns the error parameters.
GetRecordCount	Returns the number of records in the OLE DB record object.

Data Members

m_rgErrors	An array of error records.

See Also

Reference

IRowsetIdentityImpl Class

IErrorRecordsImpl Methods

For information about the methods in **IErrorRecordsImpl**, see IErrorRecordsImpl Members.

IErrorRecordsImpl::AddErrorRecord

Adds a record to the OLE DB error object.

```
STDMETHOD( AddErrorRecord )(
   ERRORINFO *pErrorInfo,
   DWORD dwLookupID,
   DISPPARAMS *pdispparams,
   IUnknown *punkCustomError,
   DWORD dwDynamicErrorID
);
```

Parameters

See IErrorRecords::AddErrorRecord in the *OLE DB Programmer's Reference*.

See Also

Reference

IErrorRecordsImpl Class

Other Resources

IErrorRecordsImpl Members

IErrorRecordsImpl::GetBasicErrorInfo

Returns basic information about the error, such as the return code and provider-specific error number.

```
STDMETHOD( GetBasicErrorInfo )(
   ULONG ulRecordNum,
   ERRORINFO *pErrorInfo
);
```

Parameters

See IErrorRecords::GetBasicErrorInfo in the *OLE DB Programmer's Reference*.

See Also

Reference

IErrorRecordsImpl Class

Other Resources

IErrorRecordsImpl Members

IErrorRecordsImpl::GetCustomErrorObject

Returns a pointer to an interface on a custom error object.

```
STDMETHOD( GetCustomErrorObject )(
   ULONG ulRecordNum,
   REFIID riid,
   IUnknown **ppObject
);
```

Parameters

See IErrorRecords::GetCustomErrorObject in the *OLE DB Programmer's Reference*.

See Also

Reference

IErrorRecordsImpl Class

Other Resources

IErrorRecordsImpl Members

IErrorRecordsImpl::GetErrorDescriptionString

Gets the error description string from an error record.

```
LPOLESTR GetErrorDescriptionString(
   ERRORINFO& rCurError
);
```

Parameters

rCurError

An **ERRORINFO** record in an **IErrorInfo** interface.

Return Value

A pointer to a string describing the error.

See Also

Reference

IErrorRecordsImpl Class

Other Resources

IErrorRecordsImpl Members

IErrorRecordsImpl::GetErrorGUID

Gets the error GUID from an error record.

```
REFGUID GetErrorGUID(
   ERRORINFO& rCurError
);
```

Parameters

rCurError

An **ERRORINFO** record in an **IErrorInfo** interface.

Return Value

A reference to a GUID for the error.

See Also

Reference

IErrorRecordsImpl Class

Other Resources

IErrorRecordsImpl Members

IErrorRecordsImpl::GetErrorHelpContext

Gets the help context ID from an error record.

```
DWORD GetErrorHelpContext(
   ERRORINFO& rCurError
);
```

Parameters

rCurError

An **ERRORINFO** record in an **IErrorInfo** interface.

Return Value

The help context ID for the error.

See Also

Reference

IErrorRecordsImpl Class

Other Resources

IErrorRecordsImpl Members

IErrorRecordsImpl::GetErrorHelpFile

Gets the path name of the help file from an error record.

```
LPOLESTR GetErrorHelpFile(
   ERRORINFO& rCurError
);
```

Parameters

rCurError

An **ERRORINFO** record in an **IErrorInfo** interface.

Return Value

Pointer to a string that contains the path name of the help file for the error.

See Also

Reference

IErrorRecordsImpl Class

Other Resources

IErrorRecordsImpl Members

IErrorRecordsImpl::GetErrorInfo

Returns an IErrorInfo interface pointer on the specified record.

```
STDMETHOD( GetErrorInfo )(
   ULONG ulRecordNum,
   LCID lcid,
   IErrorInfo **ppErrorInfo
);
```

Parameters

See IErrorRecords::GetErrorInfo in the *OLE DB Programmer's Reference*.

See Also

Reference

IErrorRecordsImpl Class

Other Resources

IErrorRecordsImpl Members

IErrorRecordsImpl::GetErrorParameters

Returns the error parameters.

```
STDMETHOD( GetErrorParameters )(
   ULONG ulRecordNum,
   DISPPARAMS *pdispparams
);
```

Parameters

See IErrorRecords::GetErrorParameters in the *OLE DB Programmer's Reference*.

See Also

Reference

IErrorRecordsImpl Class

Other Resources

IErrorRecordsImpl Members

IErrorRecordsImpl::GetRecordCount

Returns the number of records in the OLE DB record object.

```
STDMETHOD( GetRecordCount )(
   ULONG *pcRecords
);
```

Parameters

See IErrorRecords::GetRecordCount in the *OLE DB Programmer's Reference*.

See Also

Reference

IErrorRecordsImpl Class

Other Resources

IErrorRecordsImpl Members

IErrorRecordsImpl::GetErrorSource

Gets the source code that caused the error from an error record.

```
LPOLESTR GetErrorSource(
   ERRORINFO& rCurError
);
```

Parameters

rCurError

An **ERRORINFO** record in an **IErrorInfo** interface.

Return Value

Pointer to a string containing the source code for the error.

See Also

Reference

IErrorRecordsImpl Class

Other Resources

IErrorRecordsImpl Members

IErrorRecordsImpl Data Members

For information about the data members in **IErrorRecordsImpl**, see IErrorRecordsImpl Members.

IErrorRecordsImpl::m_rgErrors

An array of error records.

```
CAtlArray<RecordClass> m_rgErrors;
```

See Also

Reference

IErrorRecordsImpl Class

Other Resources

IErrorRecordsImpl Members

IGetDataSourceImpl Class

Provides an implementation of the IGetDataSource object.

```
template <class T>
class ATL_NO_VTABLE IGetDataSourceImpl : public IGetDataSource
```

Parameters

T

Your class, derived from **IGetDataSourceImpl**.

Remarks

This is a mandatory interface on the session for obtaining an interface pointer to the data source object.

Requirements

Header: atldb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

IGetDataSourceImpl Members

IGetDataSourceImpl Members

Interface Methods

GetDataSource	Returns an interface pointer on the data source object that created the session.

See Also

Reference

IGetDataSourceImpl Class

IGetDataSourceImpl Methods

For information about the methods in **IGetDataSourceImpl**, see IGetDataSourceImpl Members.

IGetDataSourceImpl::GetDataSource

Returns an interface pointer on the data source object that created the session.

```
STDMETHOD(GetDataSource)(
   REFIID riid,
   IUnknown ** ppDataSource
);
```

Parameters

See IGetDataSource::GetDataSource in the *OLE DB Programmer's Reference*.

Remarks

Useful if you need to access properties in the data source object.

See Also

Reference

IGetDataSourceImpl Class

Other Resources

IGetDataSourceImpl Members

IOpenRowsetImpl Class

Provides implementation for the **IOpenRowset** interface.

```
template <class SessionClass>
class IOpenRowsetImpl : public IOpenRowset
```

Parameters

SessionClass

Your class, derived from **IOpenRowsetImpl**.

Remarks

The IOpenRowset interface is mandatory for a session object. It opens and returns a rowset that includes all rows from a single base table or index.

Requirements

Header: atldb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

IOpenRowsetImpl Members

IOpenRowsetImpl Members

Methods

CreateRowset	Creates a rowset object. Not called directly by user.
OpenRowset	Opens and returns a rowset that includes all rows from a single base table or index. (Not in ATLDB.H)

See Also

Reference

IOpenRowsetImpl Class

IOpenRowsetImpl Methods

For information about the methods in **IOpenRowsetImpl**, see IOpenRowsetImpl Members.

IOpenRowsetImpl::CreateRowset

Creates a rowset object. Not called directly by user. See IOpenRowset::OpenRowset in the *OLE DB Programmer's Reference*.

```
template <class RowsetClass>
HRESULT CreateRowset(
   IUnknown* pUnkOuter,
   DBID* pTableID,
   DBID* pIndexID,
   REFIID riid,
   ULONG cPropertySets,
   DBPROPSET rgPropertySets[],
   IUnknown** ppRowset,
   RowsetClass*& pRowsetObj
);
```

Parameters

RowsetClass

A template class member representing the user's rowset class. Usually generated by the wizard.

pRowsetObj

[out] A pointer to a rowset object. Typically this parameter is not used, but it can be used if you must perform more work on the rowset before passing it to a COM object. The lifetime of *pRowsetObj* is bound by *ppRowset*.

For other parameters, see IOpenRowset::OpenRowset in the *OLE DB Programmer's Reference*.

See Also

Reference

IOpenRowsetImpl Class

Other Resources

IOpenRowsetImpl Members

IOpenRowsetImpl::OpenRowset

Opens and returns a rowset that includes all rows from a single base table or index.

```
HRESULT OpenRowset(
   IUnknown* pUnkOuter,
   DBID* pTableID,
   DBID* pIndexID,
   REFIID riid,
   ULONG cPropertySets,
   DBPROPSET rgPropertySets[],
   IUnknown** ppRowset
);
```

Parameters

See IOpenRowset::OpenRowset in the *OLE DB Programmer's Reference*.

Remarks

This method is not found in ATLDB.H. It is created by the ATL Object Wizard when you create a provider.

See Also

Reference

IOpenRowsetImpl Class

Other Resources

IOpenRowsetImpl Members

IRowsetChangeImpl Class

The OLE DB Templates implementation of the IRowsetChange interface in the OLE DB specification.

```
template <
   class T,
   class Storage,
   class BaseInterface = IRowsetChange,
   class RowClass = CSimpleRow,
   class MapClass = CAtlMap < RowClass::KeyType, RowClass* >
>
class ATL_NO_VTABLE IRowsetChangeImpl : public BaseInterface
```

Parameters

T

A class derived from **IRowsetChangeImpl**.

Storage

The user record.

BaseInterface

The base class for the interface, such as **IRowsetChange**.

RowClass

The storage unit for the row handle.

MapClass

The storage unit for all row handles held by the provider.

Remarks

This interface is responsible for immediate write operations to a data store. "Immediate" means that when the end user (the person using the consumer) makes any changes, those changes are immediately transmitted to the data store (and cannot be undone).

IRowsetChangeImpl implements the OLE DB **IRowsetChange** interface, which enables updating of values of columns in existing rows, deleting rows, and inserting new rows.

The OLE DB Templates implementation supports all the base methods (**SetData**, **InsertRow**, and **DeleteRows**).

Important
It is strongly recommended that you read the following documentation BEFORE attempting to implement your provider:

- Creating an Updatable Provider
- Chapter 6 of the *OLE DB Programmer's Reference*
- Also see how the `RUpdateRowset` class is used in the UpdatePV sample

Requirements

Header: atldb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

IRowsetUpdateImpl Members

IRowsetChangeImpl Members

Interface Methods (Used with IRowsetChange)

DeleteRows	Deletes rows from the rowset.
InsertRow	Inserts a row into the rowset.
SetData	Sets data values in one or more columns.

Implementation Method (Callback)

FlushData	Overidden by provider to commit data to its store.

See Also

Reference

IRowsetChangeImpl Class

IRowsetChangeImpl Methods

For information about the methods in **IRowsetChangeImpl**, see IRowsetChangeImpl Members.

IRowsetChangeImpl::DeleteRows

Deletes rows from the rowset.

```
STDMETHOD ( DeleteRows )(
   HCHAPTER /* hReserved */,
   DBCOUNTITEM cRows,
   const HROW rghRows[],
   DBROWSTATUS rgRowStatus[]
);
```

Parameters

See IRowsetChange::DeleteRows in the *OLE DB Programmer's Reference*.

See Also

Reference

IRowsetChangeImpl Class

Other Resources

IRowsetChangeImpl Members

IRowsetChangeImpl::FlushData

Overidden by provider to commit data to its store.

```
HRESULT FlushData(
   HROW hRowToFlush,
   HACCESSOR hAccessorToFlush
);
```

Parameters

hRowToFlush

[in] Handle to the rows for the data. The type of this row is determined from the *RowClass* template argument of the **IRowsetImpl** class (**CSimpleRow** by default).

hAccessorToFlush

[in] Handle to the accessor, which contains binding information and type information in its **PROVIDER_MAP** (see IAccessorImpl).

Return Value

A standard **HRESULT**.

See Also

Reference

IRowsetChangeImpl Class

Other Resources

IRowsetChangeImpl Members

IRowsetChangeImpl::InsertRow

Creates and initializes a new row in the rowset.

```
STDMETHOD ( InsertRow )(
   HCHAPTER /* hReserved */,
   HACCESSOR hAccessor,
   void* pData,
   HROW* phRow
);
```

Parameters

See IRowsetChange::InsertRow in the *OLE DB Programmer's Reference*.

See Also

Reference

IRowsetChangeImpl Class

Other Resources

IRowsetChangeImpl Members

IRowsetChangeImpl::SetData

Sets data values in one or more columns.

```
STDMETHOD ( SetData )(
   HROW hRow,
   HACCESSOR hAccessor,
   void* pSrcData
);
```

Parameters

See IRowsetChange::SetData in the *OLE DB Programmer's Reference*.

See Also

Reference

IRowsetChangeImpl Class

Other Resources

IRowsetChangeImpl Members

IRowsetCreatorImpl Class

Performs the same functions as **IObjectWithSite** but also enables the OLE DB properties **DBPROPCANSCROLLBACKWARDS DBPROPCANFETCHBACKWARDS**.

```
template < class T >
class ATL_NO_VTABLE IRowsetCreatorImpl
   : public IObjectWithSiteImpl< T >
```

Parameters

T

A class derived from **IRowsetCreator**.

Remarks

This class inherits from IObjectWithSite and overrides IObjectWithSite::SetSite. When a provider command or session object creates a rowset, it calls **QueryInterface** on the rowset object looking for **IObjectWithSite** and calls *SetSite* passing the rowset object's **IUnkown** interface as the site interface.

Requirements

Header: atldb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

IRowsetCreatorImpl Members

IRowsetCreatorImpl Members

Methods

SetSite	Sets the site that contains the rowset object.

See Also

Reference

IRowsetCreatorImpl Class

IRowsetCreatorImpl Methods

For information about the methods in **IRowsetCreatorImpl**, see IRowsetCreatorImpl Members.

IRowsetCreatorImpl::SetSite

Sets the site that contains the rowset object. For more information, see IObjectWithSite::SetSite.

```
STDMETHOD( SetSite )(
   IUnknown* pCreator
);
```

Parameters

pCreator

[in] Pointer to the **IUnknown** interface pointer of the site managing the rowset object.

Return Value

A standard **HRESULT**.

Remarks

In addition, **IRowsetCreatorImpl::SetSite** enables the OLE DB **DBPROPCANSCROLLBACKWARDS DBPROPCANFETCHBACKWARDS** properties.

See Also

Reference

IRowsetCreatorImpl Class

Other Resources

IRowsetCreatorImpl Members

IRowsetIdentityImpl Class

Implements the OLE DB IRowsetIdentity interface, which enables testing for row identity.

```
template <class T, class RowClass = CSimpleRow>
class ATL_NO_VTABLE IRowsetIdentityImpl
   : public IRowsetIdentity
```

Parameters

T

A class derived from **IRowsetIdentityImpl**.

RowClass

The storage unit for the **HROW**.

Requirements

Header: atldb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

IRowsetIdentityImpl Members

IRowsetIdentityImpl Members

Methods

IsSameRow	Compares two row handles to see if they refer to the same row.

See Also

Reference

IRowsetIdentityImpl Class

IRowsetIdentityImpl Methods

For information about the methods in **IRowsetIdentityImpl**, see IRowsetIdentityImpl Members.

IRowsetIdentityImpl::IsSameRow

Compares two row handles to see if they refer to the same row.

```
STDMETHOD( IsSameRow )(
   HROW hThisRow,
   HROW hThatRow
);
```

Parameters

See IRowsetIdentity::IsSameRow in the *OLE DB Programmer's Reference*.

Remarks

To compare row handles, this method casts the **HROW** handles to **RowClass** members and calls **memcmp** on the pointers.

See Also

Reference

IRowsetIdentityImpl Class

Other Resources

IRowsetIdentityImpl Members

IRowsetImpl Class

Provides an implementation of the **IRowset** interface.

```
template <
   class T,
   class RowsetInterface,
   class RowClass = CSimpleRow,
   class MapClass = CAtlMap <
            RowClass::KeyType,
            RowClass*
   >
>
class ATL_NO_VTABLE IRowsetImpl : public RowsetInterface
```

Parameters

T

Your class, derived from **IRowsetImpl**.

RowsetInterface

A class derived from **IRowsetImpl**.

RowClass

Storage unit for the **HROW**.

MapClass

Storage unit for all row handles held by the provider.

Remarks

IRowset is the base rowset interface.

Requirements

Header: atldb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

IRowsetImpl Members

IRowsetImpl Members

Methods

AddRefRows	Adds a reference count to an existing row handle.
CreateRow	Called by GetNextRows to allocate a new **HROW**. Not called directly by user.
GetData	Retrieves data from the rowset's copy of the row.
GetDBStatus	Returns the status for the specified field.
GetNextRows	Fetches rows sequentially, remembering the previous position.
IRowsetImpl	The constructor. Not called directly by user.
RefRows	Called by AddRefRows and ReleaseRows. Not called directly by user.
ReleaseRows	Releases rows.
RestartPosition	Repositions the next fetch position to its initial position; that is, its position when the rowset was first created.
SetDBStatus	Sets the status flags for the specified field.

Data Members

m_bCanFetchBack	Indicates whether a provider supports backward fetching.
m_bCanScrollBack	Indicates whether a provider can have its cursor scroll backwards.
m_bReset	Indicates whether a provider has reset its cursor position. This has special meaning when scrolling backwards or fetching backwards in GetNextRows.
m_iRowset	An index to the rowset, representing the cursor.
m_rgRowHandles	A list of row handles.

See Also

Reference

IRowsetImpl Class

IRowsetImpl Methods

For information about the methods in **IRowsetImpl**, see IRowsetImpl Members.

IRowsetImpl::AddRefRows

Adds a reference count to an existing row handle.

```
STDMETHOD( AddRefRows )(
   DBCOUNTITEM cRows,
   const HROW rghRows[],
   DBREFCOUNT rgRefCounts[],
   DBROWSTATUS rgRowStatus[]
);
```

Parameters

See IRowset::AddRefRows in the *OLE DB Programmer's Reference*.

See Also

Reference

IRowsetImpl Class
IRowsetImpl::RefRows
IRowsetImpl::GetNextRows
IRowsetImpl::ReleaseRows

Other Resources

IRowsetImpl Members

IRowsetImpl::CreateRow

A helper method called by GetNextRows to allocate a new **HROW**.

```
HRESULT CreateRow(
   DBROWOFFSET lRowsOffset,
   DBCOUNTITEM& cRowsObtained,
   HROW* rgRows
);
```

Parameters

lRowsOffset

Cursor position of the row being created.

cRowsObtained

A reference passed back to the user indicating the number of rows created.

rgRows

An array of **HROW**s returned to the caller with the newly created row handles.

Remarks

If the row exists, this method calls AddRefRows and returns. Otherwise, it allocates a new instance of the RowClass template variable and adds it to m_rgRowHandles.

See Also

Reference

IRowsetImpl Class

Other Resources

IRowsetImpl Members

IRowsetImpl::GetData

Retrieves data from the rowset's copy of the row.

```
STDMETHOD( GetData )(
   HROW hRow,
   HACCESSOR hAccessor,
   void* pDstData
);
```

Parameters

See IRowset::GetData in the *OLE DB Programmer's Reference*.

Some parameters correspond to *OLE DB Programmer's Reference* parameters of different names, which are described in **IRowset::GetData**:

OLE DB Template parameters	***OLE DB Programmer's Reference* parameters**
pDstData	*pData*

Remarks

Also handles data conversion using the OLE DB data conversion DLL.

See Also

Reference

IRowsetImpl Class

Other Resources

IRowsetImpl Members

IRowsetImpl::GetDBStatus

Returns the **DBSTATUS** status flags for the specified field.

```
virtual DBSTATUS GetDBStatus(
   RowClass* currentRow,
   ATLCOLUMNINFO* columnNames
);
```

Parameters

[in] *currentRow*

The current row.

[in] *columnNames*

The column for which status is being requested.

Return Value

The DBSTATUS flags for the column.

See Also

Reference

IRowsetImpl Class

Other Resources

IRowsetImpl Members

IRowsetImpl::GetNextRows

Fetches rows sequentially, remembering the previous position.

```
STDMETHOD( GetNextRows )(
   HCHAPTER hReserved,
   DBROWOFFSET lRowsOffset,
   DBROWCOUNT cRows,
   DBCOUNTITEM* pcRowsObtained,
   HROW** prghRows
);
```

Parameters

See IRowset::GetNextRows in the *OLE DB Programmer's Reference*.

See Also

Reference

IRowsetImpl Class
IRowsetImpl::AddRefRows
IRowsetImpl::ReleaseRows

Other Resources

IRowsetImpl Members

IRowsetImpl::IRowsetImpl

The constructor.

```
IRowsetImpl( );
```

Remarks

You usually do not need to call this method directly.

See Also

Reference

IRowsetImpl Class

Other Resources

IRowsetImpl Members

IRowsetImpl::RefRows

Called by AddRefRows and ReleaseRows to either increment or release a reference count to an existing row handle.

```
HRESULT RefRows(
   DBCOUNTITEM cRows,
   const HROW rghRows[],
   DBREFCOUNT rgRefCounts[],
   DBROWSTATUS rgRowStatus[],
   BOOL bAdd
);
```

Parameters

See IRowset::AddRefRows in the *OLE DB Programmer's Reference*.

Return Value

A standard **HRESULT** value.

See Also

Reference

IRowsetImpl Class

CSimpleRow Class

Other Resources

IRowsetImpl Members

IRowsetImpl::ReleaseRows

Releases rows.

```
STDMETHOD( ReleaseRows )(
   DBCOUNTITEM cRows,
   const HROW rghRows[],
   DBROWOPTIONS rgRowOptions[],
   DBREFCOUNT rgRefCounts[],
   DBROWSTATUS rgRowStatus[]
);
```

Parameters

See IRowset::ReleaseRows in the *OLE DB Programmer's Reference*.

See Also

Reference

IRowsetImpl Class
IRowsetImpl::AddRefRows
IRowsetImpl::RefRows

Other Resources

IRowsetImpl Members

IRowsetImpl::RestartPosition

Repositions the next fetch position to its initial position; that is, its position when the rowset was first created.

```
STDMETHOD( RestartPosition )(
   HCHAPTER /* hReserved */
);
```

Parameters

See IRowset::RestartPosition in the *OLE DB Programmer's Reference*.

Remarks

The rowset position is undefined until **GetNextRow** is called. You can move backwards in a rowet by calling RestartPosition and then fetching or scrolling backwards.

See Also
Reference
IRowsetImpl Class
Other Resources
IRowsetImpl Members

IRowsetImpl::SetDBStatus

Sets the **DBSTATUS** status flags for the specified field.

```
virtual HRESULT SetDBStatus(
   DBSTATUS* statusFlags,
   RowClass* currentRow,
   ATLCOLUMNINFO* columnInfo
);
```

Parameters

statusFlags

The DBSTATUS flags to set for the column.

currentRow

The current row.

columnInfo

The column for which status is being set.

Return Value

A standard **HRESULT** value.

Remarks

The provider overrides this function to provide special processing for **DBSTATUS_S_ISNULL** and **DBSTATUS_S_DEFAULT**.

See Also

Reference

IRowsetImpl Class

Other Resources

IRowsetImpl Members

IRowsetImpl Data Members

For information about the data members in **IRowsetImpl**, see IRowsetImpl Members.

IRowsetImpl::m_bCanFetchBack

Indicates whether a provider supports backward fetching.

```
unsigned m_bCanFetchBack:1;
```

Remarks

Linked to the **DBPROP_CANFETCHBACKWARDS** property in the **DBPROPSET_ROWSET** group. The provider must support **DBPROP_CANFETCHBACKWARDS** for **m_bCanFetchBackwards** to be true.

See Also

Reference

IRowsetImpl Class

IRowsetImpl::m_bCanScrollBack

Other Resources

IRowsetImpl Members

IRowsetImpl::m_bCanScrollBack

Indicates whether a provider can have its cursor scroll backwards.

```
unsigned  m_bCanScrollBack:1;
```

Remarks

Linked to the **DBPROP_CANSCROLLBACKWARDS** property in the **DBPROPSET_ROWSET** group. The provider must support **DBPROP_CANSCROLLBACKWARDS** for **m_bCanFetchBackwards** to be true.

See Also

Reference

IRowsetImpl Class

IRowsetImpl::m_bCanFetchBack

Other Resources

IRowsetImpl Members

IRowsetImpl::m_bReset

A bit flag used to determine if the cursor position is defined on the rowset.

```
unsigned m_bReset:1;
```

Remarks

If the consumer calls GetNextRows with a negative *lOffset* or *cRows* and **m_bReset** is true, **GetNextRows** moves to the end of the rowset. If **m_bReset** is false, the consumer receives an error code, in conformance with the OLE DB specification. The **m_bReset** flag gets set to **true** when the rowset is first created and when the consumer calls IRowsetImpl::RestartPosition. It gets set to **false** when you call **GetNextRows**.

See Also

Reference

IRowsetImpl Class

Other Resources

IRowsetImpl Members

IRowsetImpl::m_iRowset

An index to the rowset, representing the cursor.

```
DBROWOFFSET m_iRowset;
```

See Also

Reference

IRowsetImpl Class

Other Resources

IRowsetImpl Members

IRowsetImpl::m_rgRowHandles

A map of row handles currently contained by the provider in response to **GetNextRows**.

```
MapClass m_rgRowHandles;
```

Remarks

Row handles are removed by calling **ReleaseRows**. See the IRowsetImpl overview for the definition of *MapClass*.

See Also

Reference

IRowsetImpl Class

Other Resources

IRowsetImpl Members

IRowsetInfoImpl Class

Provides an implementation for the IRowsetInfo interface.

```
template <class T, class PropClass = T>
class ATL_NO_VTABLE IRowsetInfoImpl :
   public IRowsetInfo,
   public CUtlProps<PropClass>
```

Parameters

T

Your class, derived from **IRowsetInfoImpl**.

PropClass

A user-definable property class that defaults to *T*.

Remarks

A mandatory interface on rowsets. This class implements the rowset properties by using the property set map defined in your command class. Although the rowset class appears to be using the command class' property sets, the rowset is supplied with its own copy of the run-time properties, when it is created by a command or session object.

Requirements

Header: altdb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

IRowsetInfoImpl Members

IRowsetInfoImpl Members

Interface Methods

GetProperties	Returns the current settings of all properties supported by the rowset.
GetReferencedRowset	Returns an interface pointer to the rowset to which a bookmark applies.
GetSpecification	Returns an interface pointer on the object (command or session) that created this rowset.

See Also

Reference

IRowsetInfoImpl Class

IRowsetInfoImpl Methods

For information about the methods in **IRowsetInfoImpl**, see IRowsetInfoImpl Members.

IRowsetInfoImpl::GetProperties

Returns the current settings for properties in the **DBPROPSET_ROWSET** group.

```
STDMETHOD ( GetProperties )(
   const ULONG cPropertyIDSets,
   const DBPROPIDSET rgPropertyIDSets[],
   ULONG* pcPropertySets,
   DBPROPSET** prgPropertySets
);
```

Parameters

See IRowsetInfo::GetProperties in the *OLE DB Programmer's Reference*.

See Also

Reference

IRowsetInfoImpl Class
IRowsetInfoImpl::GetReferencedRowset
IRowsetInfoImpl::GetSpecification

Other Resources

IRowsetInfoImpl Members

IRowsetInfoImpl::GetReferencedRowset

Returns an interface pointer to the rowset to which a bookmark applies.

```
STDMETHOD ( GetReferencedRowset )(
   DBORDINAL iOrdinal,
   REFIID riid,
   IUnknown** ppReferencedRowset
);
```

Parameters

See IRowsetInfo::GetReferencedRowset in the *OLE DB Programmer's Reference*. The *iOrdinal* parameter must be a bookmark column.

See Also

Reference

IRowsetInfoImpl Class
IRowsetInfoImpl::GetSpecification
IRowsetInfoImpl::GetProperties

Other Resources

IRowsetInfoImpl Members

IRowsetInfoImpl::GetSpecification

Returns an interface pointer on the object (command or session) that created this rowset.

```
STDMETHOD ( GetSpecification )(
   REFIID riid,
   IUnknown** ppSpecification
);
```

Parameters

See IRowsetInfo::GetSpecification in the *OLE DB Programmer's Reference*.

Remarks

Use this method with IGetDataSourceImpl to retrieve properties from the data source object.

See Also

Reference

IRowsetInfoImpl Class
IRowsetInfoImpl::GetReferencedRowset
IRowsetInfoImpl::GetProperties

Other Resources

IRowsetInfoImpl Members

IRowsetLocateImpl Class

Implements the OLE DB IRowsetLocate interface, which fetches arbitrary rows from a rowset.

```
template <
   class T,
   class RowsetInterface,
   class RowClass = CSimpleRow,
   class MapClass = CAtlMap < RowClass::KeyType, RowClass* >,
   class BookmarkKeyType = LONG,
   class BookmarkType = LONG,
   class BookmarkMapClass = CAtlMap < RowClass::KeyType, RowClass* >
>
class ATL_NO_VTABLE IRowsetLocateImpl : public IRowsetImpl<
      T,
      RowsetInterface,
      RowClass,
      MapClass
>
```

Parameters

T

A class derived from **IRowsetLocateImpl**.

RowsetInterface

A class derived from **IRowsetImpl**.

RowClass

The storage unit for the **HROW**.

MapClass

The storage unit for all row handles held by the provider.

BookmarkKeyType

The type of the bookmark, such as a LONG or a string. Ordinary bookmarks must have a length of at least two bytes. (Single-byte length is reserved for the OLE DB standard bookmarks **DBBMK_FIRST**, **DBBMK_LAST**, and **DBBMK_INVALID**.)

BookmarkType

The mapping mechanism for maintaining bookmark-to-data relationships.

BookmarkMapClass

The storage unit for all row handles held by the bookmark.

Remarks

IRowsetLocateImpl is the OLE DB Templates implementation of the IRowsetLocate interface. **IRowsetLocate** is used to fetch arbitrary rows from a rowset. A rowset that does not implement this interface is a *sequential* rowset. When **IRowsetLocate** is present on a rowset, column 0 is the bookmark for the rows; reading this column will obtain a bookmark value that can be used to reposition to the same row.

IRowsetLocateImpl is used to implement bookmark support in providers. Bookmarks are placeholders (indices on a rowset) that enable the consumer to return quickly to a row, allowing high-speed access to data. The provider determines what bookmarks can uniquely identify a row. Using **IRowsetLocateImpl** methods, you can compare bookmarks, fetch rows by offset, fetch rows by bookmark, and return hash values for bookmarks.

To support OLE DB bookmarks in a rowset, make the rowset inherit from this class.

For information on implementing bookmark support, see Provider Support for Bookmarks in the *Visual C++ Programmer's Guide* and Bookmarks in the *OLE DB Programmer's Reference* in the *Platform SDK*.

Requirements

Header: atldb.h

See Also

Reference

IRowsetLocate:IRowset

Provider Support for Bookmarks

Bookmarks

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

IRowsetLocateImpl Members

IRowsetLocateImpl Members

Interface Methods

Compare	Compares two bookmarks.
GetRowsAt	Fetches rows starting with the row specified by an offset from a bookmark.
GetRowsByBookmark	Fetches the rows that match the specified bookmarks.
Hash	Returns hash values for the specified bookmarks.

Data Members

m_rgBookmarks	An array of bookmarks.

See Also

Reference

IRowsetLocateImpl Class

IRowsetLocateImpl Methods

For information about the methods in **IRowsetLocateImpl**, see IRowsetLocateImpl Members.

IRowsetLocateImpl::Compare

Compares two bookmarks.

```
STDMETHOD ( Compare )(
   HCHAPTER /* hReserved */,
   DBBKMARK cbBookmark1,
   const BYTE* pBookmark1,
   DBBKMARK cbBookmark2,
   const BYTE* pBookmark2,
   DBCOMPARE* pComparison
);
```

Parameters

See IRowsetLocate::Compare in the *OLE DB Programmer's Reference*.

Remarks

Either of the bookmarks can be a standard OLE DB-defined standard bookmark (**DBBMK_FIRST**, **DBBMK_LAST**, or **DBBMK_INVALID**). The value returned in *pComparison* indicates the relationship between the two bookmarks:

- **DBCOMPARE_LT** (*cbBookmark1* is before *cbBookmark2*.)
- **DBCOMPARE_EQ** (*cbBookmark1* is equal to *cbBookmark2*.)
- **DBCOMPARE_GT** (*cbBookmark1* is after *cbBookmark2*.)
- **DBCOMPARE_NE** (The bookmarks are equal and not ordered.)
- **DBCOMPARE_NOTCOMPARABLE** (The bookmarks cannot be compared.)

See Also

Reference

IRowsetLocateImpl Class

Other Resources

IRowsetLocateImpl Members

IRowsetLocateImpl::GetRowsAt

Fetches rows starting with the row specified by an offset from a bookmark.

```
STDMETHOD ( GetRowsAt )(
   HWATCHREGION /* hReserved1 */,
   HCHAPTER hReserved2,
   DBBKMARK cbBookmark,
   const BYTE* pBookmark,
   DBROWOFFSET lRowsOffset,
   DBROWCOUNT cRows,
   DBCOUNTITEM* pcRowsObtained,
   HROW** prghRows
);
```

Parameters

See IRowsetLocate::GetRowsAt in the *OLE DB Programmer's Reference*.

Remarks

To fetch from the cursor position instead, use IRowset::GetRowsAt.

IRowsetLocateImpl::GetRowsAt does not change the cursor position.

See Also

Reference

IRowsetLocateImpl Class

IRowsetLocateImpl::GetRowsByBookmark

Other Resources

IRowsetLocateImpl Members

IRowsetLocateImpl::GetRowsByBookmark

Fetches one or more rows that match the specified bookmarks.

```
STDMETHOD ( GetRowsByBookmark )(
   HCHAPTER /* hReserved */,
   DBCOUNTITEM cRows,
   const DBBKMARK rgcbBookmarks[],
   const BYTE* rgpBookmarks,
   HROW rghRows[],
   DBROWSTATUS* rgRowStatus[]
);
```

Parameters

hReserved

[in] Corresponds to *hChapter* parameter to IRowsetLocate::GetRowsByBookmark.

For other parameters, see IRowsetLocate::GetRowsByBookmark in the *OLE DB Programmer's Reference*.

Remarks

The bookmark can be a value you define or an OLE DB standard bookmarks (**DBBMK_FIRST** or **DBBMK_LAST**). Does not change the cursor position.

See Also

Reference

IRowsetLocateImpl Class

IRowsetLocateImpl::GetRowsAt

Other Resources

IRowsetLocateImpl Members

IRowsetLocateImpl::Hash

Returns hash values for the specified bookmarks.

```
STDMETHOD ( Hash )(
   HCHAPTER /* hReserved */,
   DBBKMARK cbBookmarks,
   const DBBKMARK* rgcbBookmarks[],
   const BYTE* rgpBookmarks[],
   DBHASHVALUE rgHashValues[],
   DBROWSTATUS rgBookmarkStatus[]
);
```

Parameters

hReserved

[in] Corresponds to *hChapter* parameter to IRowsetLocate::Hash.

For other parameters, see IRowsetLocate::Hash in the *OLE DB Programmer's Reference*.

See Also

Reference

IRowsetLocateImpl Class

Other Resources

IRowsetLocateImpl Members

IRowsetLocateImpl Data Members

For information about the data members in **IRowsetLocateImpl**, see IRowsetLocateImpl Members.

IRowsetLocateImpl::m_rgBookmarks

An array of bookmarks.

```
CAtlArray<DBROWCOUNT> m_rgBookmarks;
```

See Also

Reference

IRowsetLocateImpl Class

Other Resources

IRowsetLocateImpl Members

IRowsetNotifyCP Class

Implements the provider site for the connection point interface IRowsetNotify.

```
template <
   class T,
   class ReentrantEventSync = CComSharedMutex
>
class IRowsetNotifyCP :
   public IConnectionPointImpl<
            T,
            piid = &__uuidof(IRowsetNotify),
      CComDynamicUnkArray DynamicUnkArray
   >,
   public ReentrantEventSync
```

Parameters

T

A class derived from **IRowsetNotifyCP**.

ReentrantEventSync

A mutex class that supports reentrancy (the default is **CComSharedMutex**). A mutex is a synchronization object that allows one thread mutually exclusive access to a resource.

piid

A interface ID pointer (**IID***) for an **IRowsetNotify** connection point interface. The default value is **&__uuidof(IRowsetNotify)**.

DynamicUnkArray

An array of type CComDynamicUnkArray, which is a dynamically allocated array of **IUnknown** pointers to the client sink interfaces.

Remarks

IRowsetNotifyCP implements broadcast functions to advise listeners on the connection point **IID_IRowsetNotify** of changes to the contents of the rowset.

Note that you must also implement and register **IRowsetNotify** on the consumer (also known as the "sink") using IRowsetNotifyImpl so that the consumer can handle notifications. See Receiving Notifications about implementing the connection point interface on the consumer.

For detailed information on implementing notifications, see "Supporting Notifications" in Creating an Updatable Provider.

Requirements

Header: atldb.h

See Also

Reference

Notifications (COM)
Overview of Notifications (OLE DB)
BEGIN_CONNECTION_POINT_MAP
END_CONNECTION_POINT_MAP
CONNECTION_POINT_ENTRY

Concepts

OLE DB Provider Templates (C++)
The OLE DB Provider Template Architecture
Creating an Updatable Provider

Other Resources

IRowsetNotifyCP Members

IRowsetNotifyCP Members

Implementation Methods

Fire_OnFieldChange	Notifies the consumer of a change to the value of a column.
Fire_OnRowChange	Notifies the consumer of a change affecting the rows.
Fire_OnRowsetChange	Notifies the consumer of a change affecting the entire rowset.

See Also

Reference

IRowsetNotifyCP Class

IRowsetNotifyCP Methods

For information about the methods in **IRowsetNotifyCP**, see IRowsetNotifyCP Members.

IRowsetNotifyCP::Fire_OnFieldChange

Broadcasts an OnFieldChange event to notify consumers of a change to the value of a column.

```
HRESULT Fire_OnFieldChange(
   IRowset* pRowset,
   HROW hRow,
   DBORDINAL cColumns,
   DBORDINAL* rgColumns,
   DBREASON eReason,
   DBEVENTPHASE ePhase,
   BOOL fCantDeny
);
```

Parameters

See IRowsetNotify::OnFieldChange in the *OLE DB Programmer's Reference*.

See Also

Reference

IRowsetNotifyCP Class

Other Resources

IRowsetNotifyCP Members

IRowsetNotifyCP::Fire_OnRowChange

Broadcasts an OnRowChange event to all listeners on the connection point **IID_IRowsetNotify** to notify consumers of a change affecting the rows.

```
HRESULT Fire_OnRowChange(
   IRowset* pRowset,
   DBCOUNTITEM cRows,
   const HROW rghRows[],
   DBREASON eReason,
   DBEVENTPHASE ePhase,
   BOOL fCantDeny
);
```

Parameters

See IRowsetNotify::OnRowChange in the *OLE DB Programmer's Reference*.

See Also

Reference

IRowsetNotifyCP Class

Other Resources

IRowsetNotifyCP Members

IRowsetNotifyCP::Fire_OnRowsetChange

Broadcasts an OnRowsetChange event to all listeners on the connection point **IID_IRowsetNotify** to notify consumers of a change affecting the entire rowset.

```
HRESULT Fire_OnRowsetChange(
   IRowset* pRowset,
   DBREASON eReason,
   DBEVENTPHASE ePhase,
   BOOL fCantDeny
);
```

Parameters

See IRowsetNotify::OnRowsetChange in the *OLE DB Programmer's Reference*.

See Also

Reference

IRowsetNotifyCP Class

Other Resources

IRowsetNotifyCP Members

IRowsetUpdateImpl Class

The OLE DB Templates implementation of the IRowsetUpdate interface.

```
template <
   class T,
   class Storage,
   class UpdateArray = CAtlArray<Storage>,
   class RowClass = CSimpleRow,
   class MapClass = CAtlMap <RowClass::KeyType, RowClass*>
>
class IRowsetUpdateImpl : public IRowsetChangeImpl<
      T,
      Storage,
   IRowsetUpdate,
      RowClass,
      MapClass
>
```

Parameters

T

A class derived from **IRowsetUpdateImpl**.

Storage

The user record.

UpdateArray

An array containing cached data for updating the rowset.

RowClass

The storage unit for the **HROW**.

MapClass

The storage unit for all row handles held by the provider.

Remarks

You should first read and understand the documentation for IRowsetChange, because everything described there also applies here. You should also read chapter 6 of the *OLE DB Programmer's Reference* on setting data.

IRowsetUpdateImpl implements the OLE DB **IRowsetUpdate** interface, which enables consumers to delay the transmission of changes made with **IRowsetChange** to the data source and undo changes before transmission.

Important
It is strongly recommended that you read the following documentation BEFORE attempting to implement your provider:

- Creating an Updatable Provider
- Chapter 6 of the *OLE DB Programmer's Reference*
- Also see how the `RUpdateRowset` class is used in the UpdatePV sample

Requirements

Header: atldb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Creating an Updatable Provider

Other Resources

IRowsetUpdateImpl Members

IRowsetUpdateImpl Members

Interface Methods (Used with IRowsetChange)

SetData	Sets data values in one or more columns.

Interface Methods (Used with IRowsetUpdate)

GetOriginalData	Gets the data most recently transmitted to or obtained from the data source, ignoring pending changes.
GetPendingRows	Returns a list of rows with pending changes.
GetRowStatus	Returns the status of specified rows.
Undo	Undoes any changes to the row since the last fetch or update.
Update	Transmits any changes made to the row since the last fetch or update.

Implementation Methods (Callback)

IsUpdateAllowed	Used to check for security, integrity, and so on before allowing updates.

Data Members

m_mapCachedData	Contains the original data for the deferred operation.

See Also

Reference

IRowsetUpdateImpl Class

IRowsetUpdateImpl Methods

For information about the methods in **IRowsetUpdateImpl**, see IRowsetUpdateImpl Members.

IRowsetUpdateImpl::GetPendingRows

Returns a list of rows with pending changes.

```
STDMETHOD ( GetPendingRows )(
   HCHAPTER /* hReserved */,
   DBPENDINGSTATUS dwRowStatus,
   DBCOUNTITEM* pcPendingRows,
   HROW** prgPendingRows,
   DBPENDINGSTATUS** prgPendingStatus
);
```

Parameters

hReserved

[in] Corresponds to the *hChapter* parameter in IRowsetUpdate::GetPendingRows.

For other parameters, see IRowsetUpdate::GetPendingRows in the *OLE DB Programmer's Reference*.

Remarks

For more information, see IRowsetUpdate::GetPendingRows in the *OLE DB Programmer's Reference*.

See Also

Reference

IRowsetUpdateImpl Class

Other Resources

IRowsetUpdateImpl Members

IRowsetUpdateImpl::GetOriginalData

Gets the data most recently transmitted to or obtained from the data source, ignoring pending changes.

```
STDMETHOD ( GetOriginalData )(
   HROW hRow,
   HACCESSOR hAccessor,
   void* pData
);
```

Parameters

See IRowsetUpdate::GetOriginalData in the *OLE DB Programmer's Reference*.

See Also

Reference

IRowsetUpdateImpl Class

Other Resources

IRowsetUpdateImpl Members

IRowsetUpdateImpl::GetRowStatus

Returns the status of specified rows.

```
STDMETHOD ( GetRowStatus )(
   HCHAPTER /* hReserved */,
   DBCOUNTITEM cRows,
   const HROW rghRows[],
   DBPENDINGSTATUS rgPendingStatus[]
);
```

Parameters

hReserved

[in] Corresponds to the *hChapter* parameter in IRowsetUpdate::GetRowStatus.

For other parameters, see IRowsetUpdate::GetRowStatus in the *OLE DB Programmer's Reference*.

See Also

Reference

IRowsetUpdateImpl Class

Other Resources

IRowsetUpdateImpl Members

IRowsetUpdateImpl::IsUpdateAllowed

Override this method to check for security, integrity, and so on before updates.

```
HRESULT IsUpdateAllowed(
   DBPENDINGSTATUS /* [in] *//* status */,
   HROW /* [in] *//* hRowUpdate */,
   DBROWSTATUS* /* [out] *//* pRowStatus */
);
```

Parameters

status

[in] The status of pending operations on the rows.

hRowUpdate

[in] Handle for the rows the user wants to update.

pRowStatus

[out] The status returned to the user.

Remarks

If you determine that an update should be allowed, returns **S_OK**; otherwise returns **E_FAIL**. If you allow an update, you also need to set the **DBROWSTATUS** in IRowsetUpdateImpl::Update to an appropriate row state.

See Also

Reference

IRowsetUpdateImpl Class

Other Resources

IRowsetUpdateImpl Members

IRowsetUpdateImpl::SetData

Sets data values in one or more columns.

```
STDMETHOD ( SetData )(
   HROW hRow,
   HACCESSOR hAccessor,
   void* pSrcData
);
```

Parameters

See IRowsetChange::SetData in the *OLE DB Programmer's Reference*.

Remarks

This method overrides the IRowsetChangeImpl::SetData method but includes caching of original data to permit either immediate or deferred processing of the operation.

See Also

Reference

IRowsetUpdateImpl Class

Other Resources

IRowsetUpdateImpl Members

IRowsetUpdateImpl::Undo

Undoes any changes to the row since the last fetch or update.

```
STDMETHOD ( Undo )(
   HCHAPTER /* hReserved */,
   DBCOUNTITEM cRows,
   const HROW rghRows[ ],
   DBCOUNTITEM* pcRowsUndone,
   HROW** prgRowsUndone,
   DBROWSTATUS** prgRowStatus
);
```

Parameters

hReserved

[in] Corresponds to the *hChapter* parameter in IRowsetUpdate::Undo.

pcRowsUndone

[out] Corresponds to the *pcRows* parameter in IRowsetUpdate::Undo.

prgRowsUndone

[in] Corresponds to the *prgRows* parameter in IRowsetUpdate::Undo.

For other parameters, see IRowsetUpdate::Undo in the *OLE DB Programmer's Reference*.

See Also

Reference

IRowsetUpdateImpl Class

Other Resources

IRowsetUpdateImpl Members

IRowsetUpdateImpl::Update

Transmits any changes made to the row since the last fetch or update.

```
STDMETHOD ( Update )(
   HCHAPTER /* hReserved */,
   DBCOUNTITEM cRows,
   const HROW rghRows[],
   DBCOUNTITEM* pcRows,
   HROW** prgRows,
   DBROWSTATUS** prgRowStatus
);
```

Parameters

hReserved

[in] Corresponds to the *hChapter* parameter in IRowsetUpdate::Update.

For other parameters, see IRowsetUpdate::Update in the *OLE DB Programmer's Reference*.

Remarks

Changes are transmitted by calling IRowsetChangeImpl::FlushData. The consumer must call CRowset::Update for the changes to take effect. Set *prgRowstatus* to an appropriate value as described in Row States in the *OLE DB Programmer's Reference*.

See Also

Reference

IRowsetUpdateImpl Class

Other Resources

IRowsetUpdateImpl Members

IRowsetUpdateImpl Data Members

For information about the data members in **IRowsetUpdateImpl**, see IRowsetUpdateImpl Members.

IRowsetUpdateImpl::m_mapCachedData

A map containing the original data for the deferred operation.

```
CAtlMap<
   HROW hRow,
      Storage* pData
>
m_mapCachedData;
```

Parameters

hRow

Handle to the rows for the data.

pData

A pointer to the data to be cached. The data is of type *Storage* (the user record class). See the *Storage* template argument in IRowsetUpdateImpl Class.

See Also

Reference

IRowsetUpdateImpl Class

Other Resources

IRowsetUpdateImpl Members

ISessionPropertiesImpl Class

Provides an implementation of the ISessionProperties interface.

```
template <class T, class PropClass = T>
class ATL_NO_VTABLE ISessionPropertiesImpl :
   public ISessionProperties,
   public CUtlProps<PropClass>
```

Parameters

T

Your class, derived from **ISessionPropertiesImpl**.

PropClass

A user-definable property class that defaults to *T*.

Remarks

A mandatory interface on sessions. This class implements session properties by calling a static function defined by the property set map. The property set map should be specified in your session class.

Requirements

Header: atldb.h

See Also

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

ISessionPropertiesImpl Members

ISessionPropertiesImpl Members

Interface Methods

GetProperties	Returns the list of properties in the Session property group that are currently set on the session.
SetProperties	Sets properties in the Session property group.

See Also

Reference

ISessionPropertiesImpl Class

ISessionPropertiesImpl Methods

For information about the methods in **ISessionPropertiesImpl**, see ISessionPropertiesImpl Members.

ISessionPropertiesImpl::GetProperties

Returns the list of properties in the **DBPROPSET_SESSION** property group that are currently set on the session.

```
STDMETHOD(GetProperties)(
   ULONG cPropertyIDSets,
   const DBPROPIDSET rgPropertyIDSets[],
   ULONG * pcPropertySets,
   DBPROPSET ** prgPropertySets
);
```

Parameters

See ISessionProperties::GetProperties in the *OLE DB Programmer's Reference*.

See Also

Reference

ISessionPropertiesImpl Class

ISessionPropertiesImpl::SetProperties

Other Resources

ISessionPropertiesImpl Members

ISessionPropertiesImpl::SetProperties

Sets properties in the **DBPROPSET_SESSION** property group.

```
STDMETHOD(SetProperties)(
   ULONG cPropertySets,
   DBPROPSET rgPropertySets[]
);
```

Parameters

See ISessionProperties::SetProperties in the *OLE DB Programmer's Reference*.

See Also

Reference

ISessionPropertiesImpl Class

ISessionPropertiesImpl::GetProperties

Other Resources

ISessionPropertiesImpl Members

Macros for OLE DB Provider Templates

The OLE DB Templates Provider macros offer functionality in the following categories:

Property Set Map Macros

BEGIN_PROPERTY_SET	Marks the beginning of a property set.
BEGIN_PROPERTY_SET_EX	Marks the beginning of a property set.
BEGIN_PROPSET_MAP	Marks the beginning of a property set that can be hidden or defined outside the scope of the provider.
CHAIN_PROPERTY_SET	Chains property groups together.
END_PROPERTY_SET	Marks the end of a property set.
END_PROPSET_MAP	Marks the end of a property set map.
PROPERTY_INFO_ENTRY	Sets a specific property in a property set to a default value.
PROPERTY_INFO_ENTRY_EX	Sets a specific property in a property set to a value supplied by you. Also enables you to set flags and options.
PROPERTY_INFO_ENTRY_VALUE	Sets a specific property in a property set to a value supplied by you.

Column Map Macros

BEGIN_PROVIDER_COLUMN_MAP	Marks the beginning of the provider column map entries.
END_PROVIDER_COLUMN_MAP	Marks the end of the provider column map entries.
PROVIDER_COLUMN_ENTRY	Represents a specific column supported by the provider.
PROVIDER_COLUMN_ENTRY_GN	Represents a specific column supported by the provider. You can specify the column's size, data type, precision, scale, and schema rowset GUID.
PROVIDER_COLUMN_ENTRY_FIXED	Represents a specific column supported by the provider. You can specify the column data type.
PROVIDER_COLUMN_ENTRY_LENGTH	Represents a specific column supported by the provider. You can specify the column size.
PROVIDER_COLUMN_ENTRY_STR	Represents a specific column supported by the provider. It assumes the column type is a string.
PROVIDER_COLUMN_ENTRY_TYPE_LENGTH	Represents a specific column supported by the provider. Like PROVIDER_COLUMN_ENTRY_LENGTH, but also allows you to specify the column's data type as well as size.
PROVIDER_COLUMN_ENTRY_WSTR	Represents a specific column supported by the provider. It assumes the column type is a Unicode character string.

Schema Rowset Macros

BEGIN_SCHEMA_MAP	Marks the beginning of a schema map.

SCHEMA_ENTRY	Associates a GUID with a class.
END_SCHEMA_MAP	Marks the end of a schema map.

See Also

Reference

Creating an OLE DB Provider

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

OLE DB Provider Templates Reference

BEGIN_PROPERTY_SET

Marks the beginning of a property set in a property set map.

```
BEGIN_PROPERTY_SET(guid )
```

Parameters

guid

[in] The property GUID.

Example

See BEGIN_PROPSET_MAP.

See Also

Reference

Creating an OLE DB Provider

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

Macros for OLE DB Provider Templates

BEGIN_PROPERTY_SET_EX

Marks the beginning of a property set in a property set map.

```
BEGIN_PROPERTY_SET_EX(guid, flags )
```

Parameters

guid

[in] The property GUID.

flags

[in] **UPROPSET_HIDDEN** for any property sets you do not wish to expose, or **UPROPSET_PASSTHROUGH** for a provider exposing properties defined outside the scope of the provider.

Example

See BEGIN_PROPSET_MAP.

See Also

Reference

Creating an OLE DB Provider

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

Macros for OLE DB Provider Templates

BEGIN_PROPSET_MAP

Marks the beginning of the property set map entries.

```
BEGIN_PROPSET_MAP(Class )
```

Parameters

Class

[in] The class in which this property set is specified. A property set can be specified in the following OLE DB objects:

- Data Source Objects
- Session Objects
- Commands

Example

Here is a sample property set map:

```
BEGIN_PROPSET_MAP(CDataSource)
   BEGIN_PROPERTY_SET(0, &DBPROPSET_DATASOURCEINFO, 0, InitSrcInfoSupported)
      PROPERTY_INFO_ENTRY(ACTIVESESSIONS, VT_I4, DBPROPFLAGS_DATASOURCEINFO | DBPROPFLAGS_R
EAD, 0)
      PROPERTY_INFO_ENTRY(ASYNCTXNABORT,   VT_BOOL, DBPROPFLAGS_DATASOURCEINFO | DBPROPFLAG
S_READ, VARIANT_FALSE)
   END_PROPERTY_SET(0)
   BEGIN_PROPERTY_SET(1, &DBPROPSET_DBINIT, 0, InitDBInitSupported)
      PROPERTY_INFO_ENTRY(AUTH_PASSWORD,VT_BSTR,   DBPROPFLAGS_DBINIT | DBPROPFLAGS_READ |
DBPROPFLAGS_WRITE, myPasswordString)
      PROPERTY_INFO_ENTRY(AUTH_PERSIST_SENSITIVE_AUTHINFO, VT_BOOL, DBPROPFLAGS_DBINIT | DB
PROPFLAGS_READ | DBPROPFLAGS_WRITE, VARIANT_FALSE)
   END_PROPERTY_SET(1)
END_PROPSET_MAP()
```

See Also

Reference

Creating an OLE DB Provider

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

Macros for OLE DB Provider Templates

BEGIN_PROVIDER_COLUMN_MAP

Marks the beginning of the provider column map entries.

```
BEGIN_PROVIDER_COLUMN_MAP(theClass )
```

Parameters

theClass

[in] The name of the class this map belongs to.

Example

Here is a sample provider column map:

```
BEGIN_PROVIDER_COLUMN_MAP(CWindowsFile)
   PROVIDER_COLUMN_ENTRY("FileAttributes", 1, dwFileAttributes)
   PROVIDER_COLUMN_ENTRY("FileSizeHigh", 2, nFileSizeHigh)
   PROVIDER_COLUMN_ENTRY("FileSizeLow", 3, nFileSizeLow)
   PROVIDER_COLUMN_ENTRY("FileName", 4, cFileName)
   PROVIDER_COLUMN_ENTRY("AltFileName", 5, cAlternateFileName)
END_PROVIDER_COLUMN_MAP()
```

See Also

Reference

Creating an OLE DB Provider

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

Macros for OLE DB Provider Templates

BEGIN_SCHEMA_MAP

Denotes the beginning of a schema map.

```
BEGIN_SCHEMA_MAP(
      SchemaClass
);
```

Parameters

SchemaClass

The class that contains the MAP. Typically this will be the session class.

Remarks

See IDBSchemaRowset in the *Platform SDK* for more information about schema rowsets.

See Also

Reference

SCHEMA_ENTRY
END_SCHEMA_MAP
IDBSchemaRowsetImpl Class

Other Resources

Macros for OLE DB Provider Templates

CHAIN_PROPERTY_SET

This macro chains property groups together.

```
CHAIN_PROPERTY_SET(ChainClass )
```

Parameters

ChainClass

[in] The name of the class to chain properties for. This is a class generated by the ATL Project Wizard that already contains a map (such as a session, command, or data source object class).

Remarks

You can chain a property set from another class to your own class, then access the properties directly from your class.

Caution
Use this macro sparingly. Improper use can cause a consumer to fail the OLE DB conformance tests.

See Also

Other Resources

Macros for OLE DB Provider Templates

END_PROPERTY_SET

Marks the end of a property set.

```
END_PROPERTY_SET(guid )
```

Parameters

guid

[in] The property GUID.

Example

See BEGIN_PROPSET_MAP.

See Also

Reference

Creating an OLE DB Provider

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

Macros for OLE DB Provider Templates

END_PROPSET_MAP

Marks the end of property set map entries.

```
END_PROPSET_MAP( )
```

Example

See BEGIN_PROPSET_MAP.

See Also

Reference

Creating an OLE DB Provider

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

Macros for OLE DB Provider Templates

END_PROVIDER_COLUMN_MAP

Marks the end of the provider column map entries.

```
END_PROVIDER_COLUMN_MAP( )
```

Example

See BEGIN_PROVIDER_COLUMN_MAP.

See Also

Reference

Creating an OLE DB Provider

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

Macros for OLE DB Provider Templates

END_SCHEMA_MAP

Denotes the end of the schema map.

```
END_SCHEMA_MAP( )
```

See Also

Reference

BEGIN_SCHEMA_MAP
SCHEMA_ENTRY
IDBSchemaRowsetImpl Class

Other Resources

Macros for OLE DB Provider Templates

PROVIDER_COLUMN_ENTRY

Represents a specific column supported by the provider.

```
PROVIDER_COLUMN_ENTRY (name, ordinal, member )
```

Parameters

name

[in] The column name.

ordinal

[in] The column number. Unless the column is a Bookmark column, the column number must not be 0.

member

[in] The member variable in *dataClass* corresponding to the column.

See Also

Reference

Creating an OLE DB Provider

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

Macros for OLE DB Provider Templates

PROVIDER_COLUMN_ENTRY_GN

Represents a specific column supported by the provider.

```
PROVIDER_COLUMN_ENTRY_GN (name, ordinal, flags, colSize, dbtype, precision, scale, guid )
```

Parameters

name

[in] The column name.

ordinal

[in] The column number. Unless the column is a Bookmark column, the column number must not be 0.

flags

[in] Specifies how data is returned. See the **dwFlags** description in DBBINDING Structures.

colSize

[in] The column size.

dbtype

[in] Indicates the data type of the value. See the **wType** description in DBBINDING Structures.

precision

[in] Indicates the precision to use when getting data if *dbType* is **DBTYPE_NUMERIC** or **DBTYPE_DECIMAL**. See the **bPrecision** description in DBBINDING Structures.

scale

[in] Indicates the scale to use when getting data if dbType is **DBTYPE_NUMERIC** or **DBTYPE_DECIMAL**. See the **bScale** description in DBBINDING Structures.

guid

A schema rowset GUID. See IDBSchemaRowset in the *OLE DB Programmer's Reference* for a list of schema rowsets and their GUIDs.

Remarks

Allows you to specify the column's size, data type, precision, scale, and schema rowset GUID.

See Also

Reference

Creating an OLE DB Provider

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

Macros for OLE DB Provider Templates

PROVIDER_COLUMN_ENTRY_FIXED

Represents a specific column supported by the provider.

```
PROVIDER_COLUMN_ENTRY_FIXED(name, ordinal, dbtype, member )
```

Parameters

name

[in] The column name.

ordinal

[in] The column number. Unless the column is a Bookmark column, the column number must not be 0.

dbtype

[in] The data type in DBTYPE.

member

[in] The member variable in *dataClass* that stores the data.

Remarks

Allows you to specify the column data type.

Example

See BEGIN_PROVIDER_COLUMN_MAP.

See Also

Reference

Creating an OLE DB Provider

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

Macros for OLE DB Provider Templates

PROVIDER_COLUMN_ENTRY_LENGTH

Represents a specific column supported by the provider.

```
PROVIDER_COLUMN_ENTRY_LENGTH(name, ordinal, size, member )
```

Parameters

name

[in] The column name.

ordinal

[in] The column number. Unless the column is a Bookmark column, the column number must not be 0.

size

[in] The column size in bytes.

member

[in] The member variable in *dataClass* that stores the column data.

Remarks

Allows you to specify the column size.

Example

See BEGIN_PROVIDER_COLUMN_MAP.

See Also

Reference

Creating an OLE DB Provider

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

Macros for OLE DB Provider Templates

PROVIDER_COLUMN_ENTRY_STR

Represents a specific column supported by the provider.

```
PROVIDER_COLUMN_ENTRY_STR(name, ordinal, member )
```

Parameters

name

[in] The column name.

ordinal

[in] The column number. Unless the column is a Bookmark column, the column number must not be 0.

member

[in] The member variable in the data class that stores the data.

Remarks

Use this macro when the column data is assumed to be DBTYPE_STR.

Example

See BEGIN_PROVIDER_COLUMN_MAP.

See Also

Reference

Creating an OLE DB Provider

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

Macros for OLE DB Provider Templates

PROVIDER_COLUMN_ENTRY_TYPE_LENGTH

Represents a specific column supported by the provider.

```
PROVIDER_COLUMN_ENTRY_TYPE_LENGTH(name, ordinal, dbtype, size, member )
```

Parameters

name

[in] The column name.

ordinal

[in] The column number. Unless the column is a Bookmark column, the column number must not be 0.

dbtype

[in] The data type in DBTYPE.

size

[in] The column size in bytes.

member

[in] The member variable in the data class that stores the data.

Remarks

Similar to PROVIDER_COLUMN_ENTRY_LENGTH but also allows you to specify the column's data type as well as size.

See Also

Reference

Creating an OLE DB Provider

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

Macros for OLE DB Provider Templates

PROVIDER_COLUMN_ENTRY_WSTR

Represents a specific column supported by the provider.

```
PROVIDER_COLUMN_ENTRY_WSTR(name, ordinal, member )
```

Parameters

name

[in] The column name.

ordinal

[in] The column number. Unless the column is a Bookmark column, the column number must not be 0.

member

[in] The member variable in the data class that stores the data.

Remarks

Use this macro when the column data is a null terminated Unicode character string, DBTYPE_WSTR.

See Also

Reference

Creating an OLE DB Provider

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

Macros for OLE DB Provider Templates

PROPERTY_INFO_ENTRY

Represents a specific property in a property set.

```
PROPERTY_INFO_ENTRY(dwPropID )
```

Parameters

dwPropID

[in] A DBPROPID value that can be used in conjunction with the property set GUID to identify a property.

Remarks

This macro sets the property value of type **DWORD** to a default value defined in ATLDB.H. To set the property to a value of your choosing, use PROPERTY_INFO_ENTRY_VALUE. To set the VARTYPE and DBPROPFLAGS for the property at the same time, use PROPERTY_INFO_ENTRY_EX.

Example

See BEGIN_PROPSET_MAP.

See Also

Reference

Creating an OLE DB Provider

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

Macros for OLE DB Provider Templates

PROPERTY_INFO_ENTRY_EX

Represents a specific property in a property set.

```
PROPERTY_INFO_ENTRY_EX(dwPropID, vt, dwFlags, value, options )
```

Parameters

dwPropID

[in] A DBPROPID value that can be used in conjunction with the property set GUID to identify a property.

vt

[in] The VARTYPE of this property entry.

dwFlags

[in] A DBPROPFLAGS value describing this property entry.

value

[in] The property value of type **DWORD**.

options

Either **DBPROPOPTIONS_REQUIRED** or **DBPROPOPTIONS_SETIFCHEAP**. Normally, a provider does not need to set *options* since it is set by the consumer.

Remarks

With this macro, you can directly specify the property value of type **DWORD** as well as options and flags. To merely set a property to a default value defined in ATLDB.H, use PROPERTY_INFO_ENTRY. To set a property to a value of your choice, without setting options or flags on it, use PROPERTY_INFO_ENTRY_VALUE.

Example

See BEGIN_PROPSET_MAP.

See Also

Reference

Creating an OLE DB Provider

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

Macros for OLE DB Provider Templates

PROPERTY_INFO_ENTRY_VALUE

Represents a specific property in a property set.

```
PROPERTY_INFO_ENTRY_VALUE(dwPropID, value )
```

Parameters

dwPropID

[in] A DBPROPID value that can be used in conjunction with the property set GUID to identify a property.

value

[in] The property value of type **DWORD**.

Remarks

With this macro, you can directly specify the property value of type **DWORD**. To set the property to default value defined in ATLDB.H, use PROPERTY_INFO_ENTRY. To set the value, flags, and options for the property, use PROPERTY_INFO_ENTRY_EX.

Example

See BEGIN_PROPSET_MAP.

See Also

Reference

Creating an OLE DB Provider

Concepts

OLE DB Provider Templates (C++)

The OLE DB Provider Template Architecture

Other Resources

Macros for OLE DB Provider Templates

SCHEMA_ENTRY

Associates a GUID with a class.

```
SCHEMA_ENTRY(
      guid,
      rowsetClass
);
```

Parameters

guid

A schema rowset GUID. See IDBSchemaRowset in the *OLE DB Programmer's Reference* for a list of schema rowsets and their GUIDs.

rowsetClass

The class that will be created to represent the schema rowset.

Remarks

IDBSchemaRowsetImpl can then query the map for a list of GUIDs, or it can create a rowset if it is given a GUID. The schema rowset **IDBSchemaRowsetImpl** creates is similar to a standard **CRowsetImpl**-derived class, except it must provide an **Execute** method that has the following signature:

```
HRESULT Execute (LONG* pcRowsAffected, ULONG cRestrictions,
                 const VARIANT* rgRestrictions)
```

This **Execute** function populates the rowset's data. The ATL Project Wizard creates, as described in IDBSchemaRowset in the *OLE DB Programmer's Reference*, three initial schema rowsets in the project for each of the three mandatory OLE DB schemas:

- **DBSCHEMA_TABLES**
- **DBSCHEMA_COLUMNS**
- **DBSCHEMA_PROVIDER_TYPES**

The wizard also adds three corresponding entries in the schema map. See Creating an OLE DB Template Provider for more information about using the wizard to create a provider.

See Also

Reference

BEGIN_SCHEMA_MAP

END_SCHEMA_MAP

Other Resources

Macros for OLE DB Provider Templates

OpenMP in Visual C++

The OpenMP C and C++ application program interface lets you write applications that effectively use multiple processors. Visual C++ supports the OpenMP 2.0 standard.

In This Section

OpenMP Directives

Provides links to directives used in the OpenMP API.

OpenMP Clauses

Provides links to clauses used in the OpenMP API.

OpenMP Library Reference

Provides links to constructs used in the OpenMP API.

OpenMP C and C++ Application Program Interface

Discusses the OpenMP C and C++ API, as documented in the version 2.0 specification from the OpenMP Architecture Review Board.

Related Sections

/openmp (Enable OpenMP 2.0 Support)

Causes the compiler to process **#pragma omp**.

Predefined Macros

Names the predefined ANSI C and Microsoft C++ implementation macros. See the _OPENMP macro.

OpenMP Directives

Provides links to directives used in the OpenMP API.

Visual C++ supports the following OpenMP directives:

Directive	Description
atomic	Specifies that a memory location that will be updated atomically.
barrier	Synchronizes all threads in a team; all threads pause at the barrier, until all threads execute the barrier.
critical	Specifies that code is only executed on one thread at a time.
flush (OpenMP)	Specifies that all threads have the same view of memory for all shared objects.
for (OpenMP)	Causes the work done in a for loop inside a parallel region to be divided among threads.
master	Specifies that only the master threadshould execute a section of the program.
ordered (OpenMP Directives)	Specifies that code under a parallelized for loop should be executed like a sequential loop.
parallel	Defines a parallel region, which is code that will be executed by multiple threads in parallel.
sections (OpenMP)	Identifies code sections to be divided among all threads.
single	Lets you specify that a section of code should be executed on a single thread, not necessarily the master thread.
threadprivate	Specifies that a variable is private to a thread.

See Also

Reference

OpenMP Clauses

Other Resources

OpenMP in Visual C++

atomic

Specifies that a memory location that will be updated atomically.

```
#pragma omp atomic
      expression
```

Parameters

expression

The statement containing the lvalue whose memory location you want to protect against multiple writes. For more information about legal expression forms, see the OpenMP specification.

Remarks

The **atomic** directive supports no OpenMP clauses.

For more information, see 2.6.4 atomic Construct.

Example

```
// omp_atomic.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

#define MAX 10

int main() {
   int count = 0;
   #pragma omp parallel num_threads(MAX)
   {
      #pragma omp atomic
      count++;
   }
   printf_s("Number of threads: %d\n", count);
}
```

Output

```
Number of threads: 10
```

See Also

Other Resources

OpenMP in Visual C++

barrier

Synchronizes all threads in a team; all threads pause at the barrier, until all threads execute the barrier.

```
#pragma omp barrier
```

Remarks

The **barrier** directive supports no OpenMP clauses.

For more information, see 2.6.3 barrier Directive.

Example

For a sample of how to use **barrier**, see master.

See Also

Reference

OpenMP Directives

critical

Specifies that code is only be executed on one thread at a time.

```
#pragma omp critical [(name)]
{
      code_block
}
```

Remarks

where,

(*name*) (optional)

A name to identify the critical code. Note that name must be enclosed in parentheses.

Remarks

The **critical** directive supports no OpenMP clauses.

For more information, see 2.6.2 critical Construct.

Example

```
// omp_critical.cpp
// compile with: /openmp
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

#define SIZE 10

int main()
{
    int i;
    int max;
    int a[SIZE];

    for (i = 0; i < SIZE; i++)
    {
        a[i] = rand();
        printf_s("%d\n", a[i]);
    }

    max = a[0];
    #pragma omp parallel for num_threads(4)
        for (i = 1; i < SIZE; i++)
        {
            if (a[i] > max)
            {
                #pragma omp critical
                {
                    // compare a[i] and max again because max
                    // could have been changed by another thread after
                    // the comparison outside the critical section
                    if (a[i] > max)
                        max = a[i];
                }
            }
        }

    printf_s("max = %d\n", max);
}
```

Output

```
41
18467
6334
26500
19169
15724
11478
29358
26962
24464
max = 29358
```

See Also

Reference

OpenMP Directives

flush (OpenMP)

Specifies that all threads have the same view of memory for all shared objects.

```
#pragma omp flush [(var)]
```

Remarks

where,

var (optional)

A comma-separated list of variables that represent objects you want to synchronize. If *var* is not specified, all memory is flushed.

Remarks

The **flush** directive supports no OpenMP clauses.

For more information, see 2.6.5 flush Directive.

Example

```
// omp_flush.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

void read(int *data) {
   printf_s("read data\n");
   *data = 1;
}

void process(int *data) {
   printf_s("process data\n");
   (*data)++;
}

int main() {
   int data;
   int flag;

   flag = 0;

   #pragma omp parallel sections num_threads(2)
   {
      #pragma omp section
      {
         printf_s("Thread %d: ", omp_get_thread_num( ));
         read(&data);
         #pragma omp flush(data)
         flag = 1;
         #pragma omp flush(flag)
         // Do more work.
      }

      #pragma omp section
      {
         while (!flag) {
            #pragma omp flush(flag)
         }
         #pragma omp flush(data)

         printf_s("Thread %d: ", omp_get_thread_num( ));
         process(&data);
         printf_s("data = %d\n", data);
```

```
        }
    }
}
```

Sample Output

```
Thread 0: read data
Thread 1: process data
data = 2
```

See Also

Reference

OpenMP Directives

for (OpenMP)

Causes the work done in a for loop inside a parallel region to be divided among threads.

```
#pragma omp [parallel] for [clauses]
      for_statement
```

Remarks

where,

clause (optional)

Zero or more clauses. See the Remarks section for a list of the clauses supported by **for**.

for_statement

A for loop. Undefined behavior will result if user code in the for loop changes the index variable.

Remarks

The **for** directive supports the following OpenMP clauses:

- firstprivate
- lastprivate
- nowait
- ordered
- private
- reduction
- schedule

If **parallel** is also specified, *clause* can be any clause accepted by the **parallel** or **for** directives, except **nowait**.

For more information, see 2.4.1 for Construct.

Example

```
// omp_for.cpp
// compile with: /openmp
#include <stdio.h>
#include <math.h>
#include <omp.h>

#define NUM_THREADS 4
#define NUM_START 1
#define NUM_END 10

int main() {
   int i, nRet = 0, nSum = 0, nStart = NUM_START, nEnd = NUM_END;
   int nThreads = 0, nTmp = nStart + nEnd;
   unsigned uTmp = (unsigned((abs(nStart - nEnd) + 1)) *
                            unsigned(abs(nTmp))) / 2;
   int nSumCalc = uTmp;

   if (nTmp < 0)
      nSumCalc = -nSumCalc;

   omp_set_num_threads(NUM_THREADS);

   #pragma omp parallel default(none) private(i) shared(nSum, nThreads, nStart, nEnd)
   {
      #pragma omp master
```

```
        nThreads = omp_get_num_threads();

        #pragma omp for
        for (i=nStart; i<=nEnd; ++i) {
              #pragma omp atomic
              nSum += i;
        }
     }

     if  (nThreads == NUM_THREADS) {
        printf_s("%d OpenMP threads were used.\n", NUM_THREADS);
        nRet = 0;
     }
     else {
        printf_s("Expected %d OpenMP threads, but %d were used.\n",
                 NUM_THREADS, nThreads);
        nRet = 1;
     }

     if (nSum != nSumCalc) {
        printf_s("The sum of %d through %d should be %d, "
                 "but %d was reported!\n",
                 NUM_START, NUM_END, nSumCalc, nSum);
        nRet = 1;
     }
     else
        printf_s("The sum of %d through %d is %d\n",
                 NUM_START, NUM_END, nSum);
  }
```

Output

```
4 OpenMP threads were used.
The sum of 1 through 10 is 55
```

See Also

Reference

OpenMP Directives

master

Specifies that only the master threadshould execute a section of the program.

```
#pragma omp master
{
      code_block
}
```

Remarks

The **master** directive supports no OpenMP clauses.

The single directive lets you specify that a section of code should be executed on a single thread, not necessarily the master thread.

For more information, see 2.6.1 master Construct.

Example

```
// omp_master.cpp
// compile with: /openmp
#include <omp.h>
#include <stdio.h>

int main( )
{
    int a[5], i;

    #pragma omp parallel
    {
        // Perform some computation.
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] = i * i;

        // Print intermediate results.
        #pragma omp master
            for (i = 0; i < 5; i++)
                printf_s("a[%d] = %d\n", i, a[i]);

        // Wait.
        #pragma omp barrier

        // Continue with the computation.
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] += i;
    }
}
```

Output

```
a[0] = 0
a[1] = 1
a[2] = 4
a[3] = 9
a[4] = 16
```

See Also

Reference

OpenMP Directives

ordered (OpenMP Directives)

Specifies that code under a parallelized for loop should be executed like a sequential loop.

```
#pragma omp ordered
      structured-block
```

Remarks

The **ordered** directive must be within the dynamic extent of a for (OpenMP) or **parallel for** construct with an **ordered** clause.

The **ordered** directive supports no OpenMP clauses.

For more information, see 2.6.6 ordered Construct.

Example

```
// omp_ordered.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

static float a[1000], b[1000], c[1000];

void test(int first, int last)
{
    #pragma omp for schedule(static) ordered
    for (int i = first; i <= last; ++i) {
        // Do something here.
        if (i % 2)
        {
            #pragma omp ordered
            printf_s("test() iteration %d\n", i);
        }
    }
}

void test2(int iter)
{
    #pragma omp ordered
    printf_s("test2() iteration %d\n", iter);
}

int main( )
{
    int i;
    #pragma omp parallel
    {
        test(1, 8);
        #pragma omp for ordered
        for (i = 0 ; i < 5 ; i++)
            test2(i);
    }
}
```

Output

```
test() iteration 1
test() iteration 3
test() iteration 5
test() iteration 7
test2() iteration 0
test2() iteration 1
test2() iteration 2
test2() iteration 3
```

```
test2() iteration 4
```

See Also

Reference

OpenMP Directives

parallel

Defines a parallel region, which is code that will be executed by multiple threads in parallel.

```
#pragma omp parallel [clauses]
{
      code_block
}
```

Remarks

where,

clause (optional)

Zero or more clauses. See the Remarks section for a list of the clauses supported by **parallel**.

Remarks

The **parallel** directive supports the following OpenMP clauses:

- copyin
- default (OpenMP)
- firstprivate
- if (OpenMP)
- num_threads
- private (OpenMP)
- reduction
- shared (OpenMP)

parallel can also be used with the sections (OpenMP) and for (OpenMP) directives.

For more information, see 2.3 parallel Construct.

Example

The following sample shows how to set the number of threads and define a parallel region. By default, the number of threads is equal to the number of logical processors on the machine. For example, if you have a machine with one physical processor that has hyperthreading enabled, it will have two logical processors and, therefore, two threads.

```
// omp_parallel.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int main() {
   #pragma omp parallel num_threads(4)
   {
      int i = omp_get_thread_num();
      printf_s("Hello from thread %d\n", i);
   }
}
```

Sample Output

```
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
```

Comment

Note that the order of output can vary on different machines.

See Also

Reference

OpenMP Directives

sections (OpenMP)

Identifies code sections to be divided among all threads.

```
#pragma omp [parallel] sections [clauses]
{
   #pragma omp section
   {
            code_block
   }
}
```

Remarks

where,

clause (optional)

Zero or more clauses. See the Remarks section for a list of the clauses supported by **sections**.

Remarks

The **sections** directive can contain zero or more **section** directives.

The **sections** directive supports the following OpenMP clauses:

- firstprivate
- lastprivate
- nowait
- private (OpenMP)
- reduction

If **parallel** is also specified, *clause* can be any clause accepted by the **parallel** or **sections** directives, except **nowait**.

For more information, see 2.4.2 sections Construct.

Example

```
// omp_sections.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel sections num_threads(4)
    {
        printf_s("Hello from thread %d\n", omp_get_thread_num());
        #pragma omp section
        printf_s("Hello from thread %d\n", omp_get_thread_num());
    }
}
```

Sample Output

```
Hello from thread 0
Hello from thread 0
```

See Also

Reference

OpenMP Directives

single

Lets you specify that a section of code should be executed on a single thread, not necessarily the master thread.

```
#pragma omp single [clauses]
{
      code_block
}
```

Parameters

clause (optional)

Zero or more clauses. See the Remarks section for a list of the clauses supported by **single**.

Remarks

The **single** directive supports the following OpenMP clauses:

- copyprivate
- firstprivate
- nowait
- private (OpenMP)

The master directive lets you specify that a section of code should be executed only on the master thread.

For more information, see 2.4.3 single Construct.

Example

```
// omp_single.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int main() {
   #pragma omp parallel num_threads(2)
   {
      #pragma omp single
      // Only a single thread can read the input.
      printf_s("read input\n");

      // Multiple threads in the team compute the results.
      printf_s("compute results\n");

      #pragma omp single
      // Only a single thread can write the output.
      printf_s("write output\n");
    }
}
```

Sample Output

```
read input
compute results
compute results
write output
```

See Also

Reference

OpenMP Directives

threadprivate

Specifies that a variable is private to a thread.

```
#pragma omp threadprivate(var)
```

Remarks

where,

var

A comma-separated list of variables that you want to make private to a thread. *var* must be either a global- or namespace-scoped variable or a local static variable.

Remarks

The **threadprivate** directive supports no OpenMP clauses.

For more information, see 2.7.1 threadprivate Directive.

The **threadprivate** directive is based on the thread **__declspec** attribute; limits on **__declspec(thread)** apply to **threadprivate**.

You cannot use **threadprivate** in any DLL that will be loaded via LoadLibrary. This includes DLLs that are loaded with /DELAYLOAD (Delay Load Import), which also uses **LoadLibrary**.

You can use **threadprivate** in a DLL that is statically loaded at process startup.

Because **threadprivate** is based on **__declspec(thread)**, a **threadprivate** variable will exist in any thread started in the process, not just those threads that are part of a thread team spawned by a parallel region. This is an implementation detail that you may want to be aware of, since you may notice, for example, constructors for a **threadprivate** user-defined type called more often then expected.

A **threadprivate** variable of a destructable type is not guaranteed to have its destructor called. For example:

```
struct MyType
{
    ~MyType();
};

MyType threaded_var;
#pragma omp threadprivate(threaded_var)
int main()
{
    #pragma omp parallel
    {}
}
```

Users have no control as to when the threads constituting the parallel region will terminate. If those threads exist when the process exits, the threads will not be notified about the process exit, and the destructor will not be called for `threaded_var` on any thread except the one that exits (here, the primary thread). So code should not count on proper destruction of **threadprivate** variables.

Example

For a sample of using **threadprivate**, see private.

See Also

Reference

OpenMP Directives

OpenMP Clauses

Provides links to clauses used in the OpenMP API.

Visual C++ supports the following OpenMP clauses:

Clause	Description
copyin	Allows threads to access the master thread's value, for a threadprivate variable.
copyprivate	Specifies that one or more variables should be shared among all threads.
default (OpenMP)	Specifies the behavior of unscoped variables in a parallel region.
firstprivate	Specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable, because it exists before the parallel construct.
if (OpenMP)	Specifies whether a loop should be executed in parallel or in serial.
lastprivate	Specifies that the enclosing context's version of the variable is set equal to the private version of whichever thread executes the final iteration (for-loop construct) or last section (#pragma sections).
nowait	Overrides the barrier implicit in a directive.
num_threads	Sets the number of threads in a thread team.
ordered (OpenMP Clauses)	Required on a parallel for (OpenMP) statement if an ordered (OpenMP Directives) directive is to be used in the loop.
private (OpenMP)	Specifies that each thread should have its own instance of a variable.
reduction	Specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region.
schedule	Applies to the for (OpenMP) directive.
shared (OpenMP)	Specifies that one or more variables should be shared among all threads.

See Also

Reference

OpenMP Directives

Other Resources

OpenMP in Visual C++

copyin

Allows threads to access the master thread's value, for a threadprivate variable.

```
copyin(var)
```

Remarks

where,

var

The **threadprivate** variable that will be initialized with the value of the variable in the master thread, as it exists before the parallel construct.

Remarks

copyin applies to the following directives:

- parallel
- for (OpenMP)
- sections (OpenMP)

For more information, see 2.7.2.7 copyin.

Example

See threadprivate for an example of using **copyin**.

See Also

Reference

OpenMP Clauses

copyprivate

Specifies that one or more variables should be shared among all threads.

```
copyprivate(var)
```

Remarks

where,

var

One or more variables to share. If more than one variable is specified, separate variable names with a comma.

Remarks

copyprivate applies to the single directive.

For more information, see 2.7.2.8 copyprivate.

Example

```
// omp_copyprivate.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

float x, y, fGlobal = 1.0;
#pragma omp threadprivate(x, y)

float get_float() {
   fGlobal += 0.001;
   return fGlobal;
}

void use_float(float f, int t) {
   printf_s("Value = %f, thread = %d\n", f, t);
}

void CopyPrivate(float a, float b) {
   #pragma omp single copyprivate(a, b, x, y)
   {
      a = get_float();
      b = get_float();
      x = get_float();
      y = get_float();
    }

   use_float(a, omp_get_thread_num());
   use_float(b, omp_get_thread_num());
   use_float(x, omp_get_thread_num());
   use_float(y, omp_get_thread_num());
}

int main() {
   float a = 9.99, b = 123.456;

   printf_s("call CopyPrivate from a single thread\n");
   CopyPrivate(9.99, 123.456);

   printf_s("call CopyPrivate from a parallel region\n");
   #pragma omp parallel
   {
      CopyPrivate(a, b);
   }
}
```

Sample Output

```
call CopyPrivate from a single thread
Value = 1.001000, thread = 0
Value = 1.002000, thread = 0
Value = 1.003000, thread = 0
Value = 1.004000, thread = 0
call CopyPrivate from a parallel region
Value = 1.005000, thread = 0
Value = 1.005000, thread = 1
Value = 1.006000, thread = 0
Value = 1.006000, thread = 1
Value = 1.007000, thread = 0
Value = 1.007000, thread = 1
Value = 1.008000, thread = 0
Value = 1.008000, thread = 1
```

See Also

Reference

OpenMP Clauses

default (OpenMP)

Specifies the behavior of unscoped variables in a parallel region.

```
default(shared | none)
```

Remarks

shared, which is in effect if the **default** clause is not specified, means that any variable in a parallel region will be treated as if it were specified with the shared (OpenMP) clause. **none** means that any variables used in a parallel region that are not scoped with the private (OpenMP), shared (OpenMP), reduction, firstprivate, or lastprivate clause will cause a compiler error.

default applies to the following directives:

- parallel
- for (OpenMP)
- sections (OpenMP)

For more information, see 2.7.2.5 default.

Example

See private (OpenMP) for an example of using **default**.

See Also

Reference

OpenMP Clauses

firstprivate

Specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable, because it exists before the parallel construct.

```
firstprivate(var)
```

Remarks

where,

var

The variable to have instances in each thread and that will be initialized with the value of the variable, because it exists before the parallel construct. If more than one variable is specified, separate variable names with a comma.

Remarks

firstprivate applies to the following directives:

- for (OpenMP)
- parallel
- sections (OpenMP)
- single

For more information, see 2.7.2.2 firstprivate.

Example

See sections (OpenMP) for an example of using **firstprivate**.

See Also
Reference
OpenMP Clauses

if (OpenMP)

Specifies whether a loop should be executed in parallel or in serial.

```
if(expression)
```

Remarks

where,

expression

An integral expression that, if it evaluates to true (nonzero), causes the code in the parallel region to execute in parallel. If the expression evaluates to false (zero), the parallel region is executed in serial (by a single thread).

Remarks

if applies to the following directives:

- parallel
- for (OpenMP)
- sections (OpenMP)

For more information, see 2.3 parallel Construct.

Example

```
// omp_if.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

void test(int val)
{
    #pragma omp parallel if (val) num_threads(val)
    if (omp_in_parallel())
        #pragma omp single
        printf_s("val = %d, parallelized with %d threads\n",
                 val, omp_get_num_threads());
    else
        printf_s("val = %d, serialized\n", val);
}

int main( )
{
    test(0);
    test(2);
}
```

Output

```
val = 0, serialized
val = 2, parallelized with 2 threads
```

See Also

Reference

OpenMP Clauses

lastprivate

Specifies that the enclosing context's version of the variable is set equal to the private version of whichever thread executes the final iteration (for-loop construct) or last section (#pragma sections).

```
lastprivate(var)
```

Remarks

where,

var

The variable that is set equal to the private version of whichever thread executes the final iteration (for-loop construct) or last section (#pragma sections).

Remarks

lastprivate applies to the following directives:

- for (OpenMP)
- sections (OpenMP)

For more information, see 2.7.2.3 lastprivate.

Example

See schedule for an example of using **lastprivate** clause.

See Also

Reference

OpenMP Clauses

nowait

Overrides the barrier implicit in a directive.

```
nowait
```

Remarks

nowait applies to the following directives:

- for (OpenMP)
- sections (OpenMP)
- single

For more information, see 2.4.1 for Construct, 2.4.2 sections Construct, and 2.4.3 single Construct.

Example

```
// omp_nowait.cpp
// compile with: /openmp /c
#include <stdio.h>

#define SIZE 5

void test(int *a, int *b, int *c, int size)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (i = 0; i < size; i++)
            b[i] = a[i] * a[i];

        #pragma omp for nowait
        for (i = 0; i < size; i++)
            c[i] = a[i]/2;
    }
}

int main( )
{
    int a[SIZE], b[SIZE], c[SIZE];
    int i;

    for (i=0; i<SIZE; i++)
        a[i] = i;

    test(a,b,c, SIZE);

    for (i=0; i<SIZE; i++)
        printf_s("%d, %d, %d\n", a[i], b[i], c[i]);
}
```

Output

```
0, 0, 0
1, 1, 0
2, 4, 1
3, 9, 1
4, 16, 2
```

See Also

Reference

num_threads

Sets the number of threads in a thread team.

```
num_threads(num)
```

Remarks

where,

num

The number of threads

Remarks

The **num_threads** clause has the same functionality as the omp_set_num_threads function.

num_threads applies to the following directives:

- parallel
- for (OpenMP)
- sections (OpenMP)

For more information, see 2.3 parallel Construct.

Example

See parallel for an example of using **num_threads** clause.

See Also

Reference

OpenMP Clauses

ordered (OpenMP Clauses)

Required on a parallel for (OpenMP) statement if an ordered (OpenMP Directives) directive is to be used in the loop.

```
ordered
```

Remarks

ordered applies to the for (OpenMP) directive.

For more information, see 2.4.1 for Construct.

Example

See ordered (OpenMP Directives) for an example of using **ordered** clause.

See Also

Reference

OpenMP Clauses

private (OpenMP)

Specifies that each thread should have its own instance of a variable.

```
private(var)
```

Remarks

where,

var

The variable to have instances in each thread.

Remarks

private applies to the following directives:

- for (OpenMP)
- parallel
- sections (OpenMP)
- single

For more information, see 2.7.2.1 private.

Example

```
// openmp_private.c
// compile with: /openmp
#include <windows.h>
#include <assert.h>
#include <stdio.h>
#include <omp.h>

#define NUM_THREADS 4
#define SLEEP_THREAD 1
#define NUM_LOOPS 2

enum Types {
   ThreadPrivate,
   Private,
   FirstPrivate,
   LastPrivate,
   Shared,
   MAX_TYPES
};

int nSave[NUM_THREADS][MAX_TYPES][NUM_LOOPS] = {{0}};
int nThreadPrivate;

#pragma omp threadprivate(nThreadPrivate)
#pragma warning(disable:4700)

int main() {
   int nPrivate = NUM_THREADS;
   int nFirstPrivate = NUM_THREADS;
   int nLastPrivate = NUM_THREADS;
   int nShared = NUM_THREADS;
   int nRet = 0;
   int i;
   int j;
   int nLoop = 0;
```

```
   nThreadPrivate = NUM_THREADS;
   printf_s("These are the variables before entry "
           "into the parallel region.\n");
   printf_s("nThreadPrivate = %d\n", nThreadPrivate);
   printf_s("      nPrivate = %d\n", nPrivate);
   printf_s(" nFirstPrivate = %d\n", nFirstPrivate);
   printf_s("  nLastPrivate = %d\n", nLastPrivate);
   printf_s("       nShared = %d\n\n", nShared);
   omp_set_num_threads(NUM_THREADS);

   #pragma omp parallel copyin(nThreadPrivate) private(nPrivate) shared(nShared) firstpriva
te(nFirstPrivate)
   {
      #pragma omp for schedule(static) lastprivate(nLastPrivate)
      for (i = 0 ; i < NUM_THREADS ; ++i) {
         for (j = 0 ; j < NUM_LOOPS ; ++j) {
            int nThread = omp_get_thread_num();
            assert(nThread < NUM_THREADS);

            if (nThread == SLEEP_THREAD)
               Sleep(100);
            nSave[nThread][ThreadPrivate][j] = nThreadPrivate;
            nSave[nThread][Private][j] = nPrivate;
            nSave[nThread][Shared][j] = nShared;
            nSave[nThread][FirstPrivate][j] = nFirstPrivate;
            nSave[nThread][LastPrivate][j] = nLastPrivate;
            nThreadPrivate = nThread;
            nPrivate = nThread;
            nShared = nThread;
            nLastPrivate = nThread;
            --nFirstPrivate;
         }
      }
   }

   for (i = 0 ; i < NUM_LOOPS ; ++i) {
      for (j = 0 ; j < NUM_THREADS ; ++j) {
         printf_s("These are the variables at entry of "
                  "loop %d of thread %d.\n", i + 1, j);
         printf_s("nThreadPrivate = %d\n",
                  nSave[j][ThreadPrivate][i]);
         printf_s("      nPrivate = %d\n",
                  nSave[j][Private][i]);
         printf_s(" nFirstPrivate = %d\n",
                  nSave[j][FirstPrivate][i]);
         printf_s("  nLastPrivate = %d\n",
                  nSave[j][LastPrivate][i]);
         printf_s("       nShared = %d\n\n",
                  nSave[j][Shared][i]);
      }
   }

   printf_s("These are the variables after exit from "
            "the parallel region.\n");
   printf_s("nThreadPrivate = %d (The last value in the "
            "master thread)\n", nThreadPrivate);
   printf_s("      nPrivate = %d (The value prior to "
            "entering parallel region)\n", nPrivate);
   printf_s(" nFirstPrivate = %d (The value prior to "
            "entering parallel region)\n", nFirstPrivate);
   printf_s("  nLastPrivate = %d (The value from the "
            "last iteration of the loop)\n", nLastPrivate);
   printf_s("       nShared = %d (The value assigned, "
            "from the delayed thread, %d)\n\n",
            nShared, SLEEP_THREAD);
}
```

Sample Output

```
These are the variables before entry into the parallel region.
nThreadPrivate = 4
      nPrivate = 4
 nFirstPrivate = 4
  nLastPrivate = 4
       nShared = 4

These are the variables at entry of loop 1 of thread 0.
nThreadPrivate = 4
      nPrivate = 1310720
 nFirstPrivate = 4
  nLastPrivate = 1245104
       nShared = 3

These are the variables at entry of loop 1 of thread 1.
nThreadPrivate = 4
      nPrivate = 4488
 nFirstPrivate = 4
  nLastPrivate = 19748
       nShared = 0

These are the variables at entry of loop 1 of thread 2.
nThreadPrivate = 4
      nPrivate = -132514848
 nFirstPrivate = 4
  nLastPrivate = -513199792
       nShared = 4

These are the variables at entry of loop 1 of thread 3.
nThreadPrivate = 4
      nPrivate = 1206
 nFirstPrivate = 4
  nLastPrivate = 1204
       nShared = 2

These are the variables at entry of loop 2 of thread 0.
nThreadPrivate = 0
      nPrivate = 0
 nFirstPrivate = 3
  nLastPrivate = 0
       nShared = 0

These are the variables at entry of loop 2 of thread 1.
nThreadPrivate = 1
      nPrivate = 1
 nFirstPrivate = 3
  nLastPrivate = 1
       nShared = 1

These are the variables at entry of loop 2 of thread 2.
nThreadPrivate = 2
      nPrivate = 2
 nFirstPrivate = 3
  nLastPrivate = 2
       nShared = 2

These are the variables at entry of loop 2 of thread 3.
nThreadPrivate = 3
      nPrivate = 3
 nFirstPrivate = 3
  nLastPrivate = 3
       nShared = 3

These are the variables after exit from the parallel region.
nThreadPrivate = 0 (The last value in the master thread)
      nPrivate = 4 (The value prior to entering parallel region)
 nFirstPrivate = 4 (The value prior to entering parallel region)
  nLastPrivate = 3 (The value from the last iteration of the loop)
       nShared = 1 (The value assigned, from the delayed thread, 1)
```

See Also

Reference

reduction

Specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region.

```
reduction(operation:var)
```

Remarks

where,

operation

The operator for the operation to perform on the variables (*var*) at the end of the parallel region.

var

One more more variables on which to perform scalar reduction. If more than one variable is specified, separate variable names with a comma.

Remarks

reduction applies to the following directives:

- for (OpenMP)
- parallel
- sections (OpenMP)

For more information, see 2.7.2.6 reduction.

Example

```
// omp_reduction.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

#define NUM_THREADS 4
#define SUM_START   1
#define SUM_END     10
#define FUNC_RETS   {1, 1, 1, 1, 1}

int bRets[5] = FUNC_RETS;
int nSumCalc = ((SUM_START + SUM_END) * (SUM_END - SUM_START + 1)) / 2;

int func1( ) {return bRets[0];}
int func2( ) {return bRets[1];}
int func3( ) {return bRets[2];}
int func4( ) {return bRets[3];}
int func5( ) {return bRets[4];}

int main( )
{
    int nRet = 0,
        nCount = 0,
        nSum = 0,
        i,
        bSucceed = 1;

    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel reduction(+ : nCount)
    {
        nCount += 1;
```

```
        #pragma omp for reduction(+ : nSum)
        for (i = SUM_START ; i <= SUM_END ; ++i)
            nSum += i;

        #pragma omp sections reduction(&& : bSucceed)
        {
            #pragma omp section
            {
                bSucceed = bSucceed && func1( );
            }

            #pragma omp section
            {
                bSucceed = bSucceed && func2( );
            }

            #pragma omp section
            {
                bSucceed = bSucceed && func3( );
            }

            #pragma omp section
            {
                bSucceed = bSucceed && func4( );
            }

            #pragma omp section
            {
                bSucceed = bSucceed && func5( );
            }
        }
    }

    printf_s("The parallel section was executed %d times "
             "in parallel.\n", nCount);
    printf_s("The sum of the consecutive integers from "
             "%d to %d, is %d\n", 1, 10, nSum);

    if (bSucceed)
        printf_s("All of the the functions, func1 through "
                 "func5 succeeded!\n");
    else
        printf_s("One or more of the the functions, func1 "
                 "through func5 failed!\n");

    if (nCount != NUM_THREADS)
    {
        printf_s("ERROR: For %d threads, %d were counted!\n",
                 NUM_THREADS, nCount);
        nRet |= 0x1;
    }

    if (nSum != nSumCalc)
    {
        printf_s("ERROR: The sum of %d through %d should be %d, "
                 "but %d was reported!\n",
                 SUM_START, SUM_END, nSumCalc, nSum);
        nRet |= 0x10;
    }

    if (bSucceed != (bRets[0] && bRets[1] &&
                     bRets[2] && bRets[3] && bRets[4]))
    {
        printf_s("ERROR: The sum of %d through %d should be %d, "
                 "but %d was reported!\n",
                 SUM_START, SUM_END, nSumCalc, nSum);
        nRet |= 0x100;
```

```
    }
  }
```

Output

```
The parallel section was executed 4 times in parallel.
The sum of the consecutive integers from 1 to 10, is 55
All of the the functions, func1 through func5 succeeded!
```

See Also

Reference

OpenMP Clauses

schedule

Applies to the for (OpenMP) directive.

```
schedule(type[,size])
```

Parameters

type

The kind of scheduling:

- **dynamic**
- **guided**
- **runtime**
- **static**

size (optional)

Specifies the size of iterations. *size* must be an integer. Not valid when *type* is **runtime**.

Remarks

For more information, see 2.4.1 for Construct.

Example

```
// omp_schedule.cpp
// compile with: /openmp /c
#include <windows.h>
#include <stdio.h>
#include <omp.h>

#define NUM_THREADS 4
#define STATIC_CHUNK 5
#define DYNAMIC_CHUNK 5
#define NUM_LOOPS 20
#define SLEEP_EVERY_N 3

int main( )
{
    int nStatic1[NUM_LOOPS],
        nStaticN[NUM_LOOPS];
    int nDynamic1[NUM_LOOPS],
        nDynamicN[NUM_LOOPS];
    int nGuided[NUM_LOOPS];

    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel
    {
        #pragma omp for schedule(static, 1)
        for (int i = 0 ; i < NUM_LOOPS ; ++i)
        {
            if ((i % SLEEP_EVERY_N) == SLEEP_EVERY_N)
                Sleep(0);
            nStatic1[i] = omp_get_thread_num( );
        }

        #pragma omp for schedule(static, STATIC_CHUNK)
        for (int i = 0 ; i < NUM_LOOPS ; ++i)
        {
            if ((i % SLEEP_EVERY_N) == SLEEP_EVERY_N)
                Sleep(0);
```

```
            nStaticN[i] = omp_get_thread_num( );
        }

        #pragma omp for schedule(dynamic, 1)
        for (int i = 0 ; i < NUM_LOOPS ; ++i)
        {
            if ((i % SLEEP_EVERY_N) == SLEEP_EVERY_N)
                Sleep(0);
            nDynamic1[i] = omp_get_thread_num( );
        }

        #pragma omp for schedule(dynamic, DYNAMIC_CHUNK)
        for (int i = 0 ; i < NUM_LOOPS ; ++i)
        {
            if ((i % SLEEP_EVERY_N) == SLEEP_EVERY_N)
                Sleep(0);
            nDynamicN[i] = omp_get_thread_num( );
        }

        #pragma omp for schedule(guided)
        for (int i = 0 ; i < NUM_LOOPS ; ++i)
        {
            if ((i % SLEEP_EVERY_N) == SLEEP_EVERY_N)
                Sleep(0);
            nGuided[i] = omp_get_thread_num( );
        }
    }

    printf_s("-----------------------------------------------\n");
    printf_s("| static | static | dynamic | dynamic | guided |\n");
    printf_s("|    1   |    %d   |    1    |    %d    |        |\n",
             STATIC_CHUNK, DYNAMIC_CHUNK);
    printf_s("-----------------------------------------------\n");

    for (int i=0; i<NUM_LOOPS; ++i)
    {
        printf_s("|    %d   |    %d   |    %d    |    %d    |"
                 "    %d   |\n",
                 nStatic1[i], nStaticN[i],
                 nDynamic1[i], nDynamicN[i], nGuided[i]);
    }

    printf_s("-----------------------------------------------\n");
}
```

Output

```
-----------------------------------------------
| static | static | dynamic | dynamic | guided |
|    1   |    5   |    1    |    5    |        |
-----------------------------------------------
|    0   |    0   |    0    |    0    |    0   |
|    1   |    0   |    0    |    0    |    0   |
|    2   |    0   |    0    |    0    |    0   |
|    3   |    0   |    0    |    0    |    0   |
|    0   |    0   |    0    |    0    |    0   |
|    1   |    1   |    0    |    0    |    0   |
|    2   |    1   |    0    |    0    |    0   |
|    3   |    1   |    0    |    0    |    0   |
|    0   |    1   |    0    |    0    |    0   |
|    1   |    1   |    0    |    0    |    0   |
|    2   |    2   |    0    |    0    |    0   |
|    3   |    2   |    0    |    0    |    0   |
|    0   |    2   |    0    |    0    |    0   |
|    1   |    2   |    0    |    0    |    0   |
|    2   |    2   |    0    |    0    |    0   |
|    3   |    3   |    0    |    0    |    0   |
|    0   |    3   |    0    |    0    |    0   |
|    1   |    3   |    0    |    0    |    0   |
```

2	3	0	0	0
3	3	0	0	0

See Also

Reference

OpenMP Clauses

shared (OpenMP)

Specifies that one or more variables should be shared among all threads.

```
shared(var)
```

Remarks

where,

var

One more more variables to share. If more than one variable is specified, separate variable names with a comma.

Remarks

Another way to share variables among threads is with the copyprivate clause.

shared applies to the following directives:

- for (OpenMP)
- parallel
- sections (OpenMP)

For more information, see 2.7.2.4 shared.

Example

See private (OpenMP) for an example of using **shared**.

See Also

Reference

OpenMP Clauses

OpenMP Library Reference

Provides links to constructs used in the OpenMP API.

The Visual C++ implementation of the OpenMP standard includes the following constructs.

Construct	Description
OpenMP Libraries	Discusses the .lib files that make up the OpenMP run-time libraries in Visual C++.
OpenMP Data Types	Provides links to data types used in the OpenMP API.
OpenMP Environment Variables	Provides links to environment variables used in the OpenMP API.
OpenMP Functions	Provides links to functions used in the OpenMP API.

See Also

Other Resources

OpenMP in Visual C++

OpenMP Libraries

Discusses the .lib files that make up the OpenMP run-time libraries in Visual C++.

The following libraries contain the Visual C++ OpenMP run-time library functions.

OpenMP run-time library	Characteristics
VCOMP.LIB	Multithreaded, dynamic link (import library for VCOMP.LIB).
VCOMPD.LIB	Multithreaded, dynamic link (import library for VCOMPD.LID) (debug)

If _DEBUG is defined in a compilation and if `#include omp.h` is in source code, VCOMPD.LIB will be the default lib. Otherwise, VCOMP.LIB will be used.

You can use /NODEFAULTLIB (Ignore Libraries) to remove the default lib and explicitly link with the lib of your choice.

The OpenMP DLLs are in the Visual C++ redistributable directory and need to be distributed with applications that use OpenMP.

See Also

Reference

OpenMP Library Reference

OpenMP Data Types

Provides links to data types used in the OpenMP API.

The Visual C++ implementation of the OpenMP standard includes the following data types.

Data type	Description
omp_lock_t	A type that holds the status of a lock, whether the lock is available or if a thread owns a lock.
omp_nest_lock_t	A type that holds one of the following pieces of information about a lock: whether the lock is available, and the identity of the thread that owns the lock and a nesting count.

See Also

Reference

OpenMP Library Reference

omp_lock_t

A type that holds the status of a lock, whether the lock is available or if a thread owns a lock.

The following functions use **omp_lock_t**:

- omp_init_lock
- omp_destroy_lock
- omp_set_lock
- omp_unset_lock
- omp_test_lock

For more information, see 3.2 Lock Functions.

Example

See omp_init_lock for an example of using **omp_lock_t**.

See Also

Concepts

OpenMP Data Types

omp_nest_lock_t

A type that holds the following pieces of information about a lock: whether the lock is available, and the identity of the thread that owns the lock and a nesting count.

The following functions use **omp_nest_lock_t**:

- omp_init_nest_lock
- omp_destroy_nest_lock
- omp_set_nest_lock
- omp_unset_nest_lock
- omp_test_nest_lock

For more information, see 3.2 Lock Functions.

Example

See omp_init_nest_lock for an example of using **omp_nest_lock_t**.

See Also

Concepts

OpenMP Data Types

OpenMP Environment Variables

Provides links to environment variables used in the OpenMP API.

The Visual C++ implementation of the OpenMP standard includes the following environment variables. These environment variables are read at program startup and modifications to their values are ignored at runtime (for example, using _putenv, _wputenv).

Environment variable	Description
OMP_DYNAMIC	Specifies whether the OpenMP run time can adjust the number of threads in a parallel region.
OMP_NESTED	Specifies whether nested parallelism is enabled, unless nested parallelism is enabled or disabled with **omp_set_nested**.
OMP_NUM_THREADS	Sets the maximum number of threads in the parallel region, unless overridden by omp_set_num_threads or num_threads.
OMP_SCHEDULE	Modifies the behavior of the schedule clause when **schedule(runtime)** is specified in a **for** or **parallel for** directive.

See Also

Reference

OpenMP Library Reference

OMP_DYNAMIC

Specifies whether the OpenMP run time can adjust the number of threads in a parallel region.

```
set OMP_DYNAMIC[=TRUE | =FALSE]
```

Remarks

The **OMP_DYNAMIC** environment variable can be overridden by the omp_set_dynamic function.

The default value in the Visual C++ implementation of the OpenMP standard is `OMP_DYNAMIC=FALSE`.

For more information, see 4.3 OMP_DYNAMIC.

Example

The following command sets the **OMP_DYNAMIC** environment variable to TRUE:

```
set OMP_DYNAMIC=TRUE
```

The following command displays the current setting of the **OMP_DYNAMIC** environment variable:

```
set OMP_DYNAMIC
```

See Also

Reference

OpenMP Environment Variables

OMP_NESTED

Specifies whether nested parallelism is enabled, unless nested parallelism is enabled or disabled with **omp_set_nested**.

```
set OMP_NESTED[=TRUE | =FALSE]
```

Remarks

The **OMP_NESTED** environment variable can be overridden by the omp_set_nested function.

The default value in the Visual C++ implementation of the OpenMP standard is `OMP_DYNAMIC=FALSE`.

For more information, see 4.4 OMP_NESTED.

Example

The following command sets the **OMP_NESTED** environment variable to TRUE:

```
set OMP_NESTED=TRUE
```

The following command displays the current setting of the **OMP_NESTED** environment variable:

```
set OMP_NESTED
```

See Also

Reference

OpenMP Environment Variables

OMP_NUM_THREADS

Sets the maximum number of threads in the parallel region, unless overridden by omp_set_num_threads or num_threads.

```
set OMP_NUM_THREADS[=num]
```

Remarks

where,

num

The maximum number of threads you want in the parallel region, up to 64 in the Visual C++ implementation.

Remarks

The **OMP_NUM_THREADS** environment variable can be overridden by the omp_set_num_threads function or by num_threads.

The default value of *num* in the Visual C++ implementation of the OpenMP standard is the number of virtual processors, including hyperthreading CPUs.

For more information, see 4.2 OMP_NUM_THREADS.

Example

The following command sets the **OMP_NUM_THREADS** environment variable to 16:

```
set OMP_NUM_THREADS=16
```

The following command displays the current setting of the **OMP_NUM_THREADS** environment variable:

```
set OMP_NUM_THREADS
```

See Also

Reference

OpenMP Environment Variables

OMP_SCHEDULE

Modifies the behavior of the schedule clause when **schedule(runtime)** is specified in a **for** or **parallel for** directive.

```
set OMP_SCHEDULE[=type[,size]]
```

Remarks

where,

size (optional)

Specifies the size of iterations. *size* must be a positive integer. The default is 1, except when *type* is static. Not valid when *type* is **runtime**.

type

The kind of scheduling:

- **dynamic**
- **guided**
- **runtime**
- **static**

Remarks

The default value in the Visual C++ implementation of the OpenMP standard is `OMP_SCHEDULE=static,0`.

For more information, see 4.1 OMP_SCHEDULE.

Example

The following command sets the **OMP_SCHEDULE** environment variable:

```
set OMP_SCHEDULE="guided,2"
```

The following command displays the current setting of the **OMP_SCHEDULE** environment variable:

```
set OMP_SCHEDULE
```

See Also

Reference

OpenMP Environment Variables

OpenMP Functions

Provides links to functions used in the OpenMP API.

The Visual C++ implementation of the OpenMP standard includes the following functions.

Function	Description
omp_destroy_lock	Uninitializes a lock.
omp_destroy_nest_lock	Uninitializes a nestable lock.
omp_get_dynamic	Returns a value that indicates if the number of threads available in subsequent parallel region can be adjusted by the run time.
omp_get_max_threads	Returns an integer that is equal to or greater than the number of threads that would be available if a parallel region without num_threads were defined at that point in the code.
omp_get_nested	Returns a value that indicates if nested parallelism is enabled.
omp_get_num_procs	Returns the number of processors that are available when the function is called.
omp_get_num_threads	Returns the number of threads in the parallel region.
omp_get_thread_num	Returns the thread number of the thread executing within its thread team.
omp_get_wtick	Returns the number of seconds between processor clock ticks.
omp_get_wtime	Returns a value in seconds of the time elapsed from some point.
omp_in_parallel	Returns nonzero if called from within a parallel region.
omp_init_lock	Initializes a simple lock.
omp_init_nest_lock	Initializes a lock.
omp_set_dynamic	Indicates that the number of threads available in subsequent parallel region can be adjusted by the run time.
omp_set_lock	Blocks thread execution until a lock is available.
omp_set_nest_lock	Blocks thread execution until a lock is available.
omp_set_nested	Enables nested parallelism.
omp_set_num_threads	Sets the number of threads in subsequent parallel regions, unless overridden by a num_threads clause.
omp_test_lock	Attempts to set a lock but does not block thread execution.
omp_test_nest_lock	Attempts to set a nestable lock but does not block thread execution.
omp_unset_lock	Releases a lock.

omp_unset_nest_lock	Releases a nestable lock.

See Also

Reference

OpenMP Library Reference

omp_destroy_lock

Uninitializes a lock.

```
void omp_destroy_lock(
   omp_lock_t *lock
);
```

Remarks

where,

lock

A variable of type omp_lock_t that was initialized with omp_init_lock.

Remarks

For more information, see 3.2.2 omp_destroy_lock and omp_destroy_nest_lock Functions.

Example

See omp_init_lock for an example of using **omp_destroy_lock**.

See Also

Concepts

OpenMP Functions

omp_destroy_nest_lock

Uninitializes a nestable lock.

```
void omp_destroy_nest_lock(
   omp_nest_lock_t *lock
);
```

Remarks

where,

lock

A variable of type omp_nest_lock_t that was initialized with omp_init_nest_lock.

Remarks

For more information, see 3.2.2 omp_destroy_lock and omp_destroy_nest_lock Functions.

Example

See omp_init_nest_lock for an example of using **omp_destroy_nest_lock**.

See Also

Concepts

OpenMP Functions

omp_get_dynamic

Returns a value that indicates if the number of threads available in subsequent parallel region can be adjusted by the run time.

```
int omp_get_dynamic();
```

Return Value

If nonzero, dynamic adjustment of threads is enabled.

Remarks

Dynamic adjustment of threads is specified with omp_set_dynamic and OMP_DYNAMIC.

For more information, see 3.1.7 omp_set_dynamic Function.

Example

See omp_set_dynamic for an example of using **omp_get_dynamic**.

See Also

Concepts

OpenMP Functions

omp_get_max_threads

Returns an integer that is equal to or greater than the number of threads that would be available if a parallel region without num_threads were defined at that point in the code.

```
int omp_get_max_threads( )
```

Remarks

For more information, see 3.1.3 omp_get_max_threads Function.

Example

```
// omp_get_max_threads.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int main( )
{
    omp_set_num_threads(8);
    printf_s("%d\n", omp_get_max_threads( ));
    #pragma omp parallel
        #pragma omp master
        {
            printf_s("%d\n", omp_get_max_threads( ));
        }

    printf_s("%d\n", omp_get_max_threads( ));

    #pragma omp parallel num_threads(3)
        #pragma omp master
        {
            printf_s("%d\n", omp_get_max_threads( ));
        }

    printf_s("%d\n", omp_get_max_threads( ));
}
```

Output

```
8
8
8
8
8
```

See Also

Concepts

OpenMP Functions

omp_get_nested

Returns a value that indicates if nested parallelism is enabled.

```
int omp_get_nested( );
```

Return Value

If nonzero, nested parallelism is enabled.

Remarks

Nested parallelism is specified with omp_set_nested and OMP_NESTED.

For more information, see 3.1.10 omp_get_nested Function.

Example

See omp_set_nested for an example of using **omp_get_nested**.

See Also

Concepts

OpenMP Functions

omp_get_num_procs

Returns the number of processors that are available when the function is called.

```
int omp_get_num_procs();
```

Remarks

For more information, see 3.1.5 omp_get_num_procs Function.

Example

```
// omp_get_num_procs.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int main( )
{
    printf_s("%d\n", omp_get_num_procs( ));
    #pragma omp parallel
        #pragma omp master
        {
            printf_s("%d\n", omp_get_num_procs( ));
        }
}
```

Sample Output

```
// Expect the following output when the example is run on a two-processor machine:
2
2
```

See Also

Concepts

OpenMP Functions

omp_get_num_threads

Returns the number of threads in the parallel region.

```
int omp_get_num_threads( );
```

Remarks

For more information, see 3.1.2 omp_get_num_threads Function.

Example

```
// omp_get_num_threads.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int main()
{
    omp_set_num_threads(4);
    printf_s("%d\n", omp_get_num_threads( ));
    #pragma omp parallel
        #pragma omp master
        {
            printf_s("%d\n", omp_get_num_threads( ));
        }

    printf_s("%d\n", omp_get_num_threads( ));

    #pragma omp parallel num_threads(3)
        #pragma omp master
        {
            printf_s("%d\n", omp_get_num_threads( ));
        }

    printf_s("%d\n", omp_get_num_threads( ));
}
```

Output

```
1
4
1
3
1
```

See Also

Concepts

OpenMP Functions

omp_get_thread_num

Returns the thread number of the thread executing within its thread team.

```
int omp_get_thread_num( );
```

Remarks

For more information, see 3.1.4 omp_get_thread_num Function.

Example

See parallel for an example of using **omp_get_thread_num**.

See Also

Concepts

OpenMP Functions

omp_get_wtick

Returns the number of seconds between processor clock ticks.

```
double omp_get_wtick( );
```

Remarks

For more information, see 3.3.2 omp_get_wtick Function.

Example

See omp_get_wtime for an example of using **omp_get_wtick**.

See Also

Concepts

OpenMP Functions

omp_get_wtime

Returns a value in seconds of the time elapsed from some point.

```
double omp_get_wtime( );
```

Return Value

Returns a value in seconds of the time elapsed from some arbitrary, but consistent point.

Remarks

That point will remain consistent during program execution, making subsequent comparisons possible.

For more information, see 3.3.1 omp_get_wtime Function.

Example

```
// omp_get_wtime.cpp
// compile with: /openmp
#include "omp.h"
#include <stdio.h>
#include <windows.h>

int main() {
    double start = omp_get_wtime( );
    Sleep(1000);
    double end = omp_get_wtime( );
    double wtick = omp_get_wtick( );

    printf_s("start = %.16g\nend = %.16g\ndiff = %.16g\n",
             start, end, end - start);

    printf_s("wtick = %.16g\n1/wtick = %.16g\n",
             wtick, 1.0 / wtick);
}
```

Sample Output

```
start = 594255.3671159324
end = 594256.3664474116
diff = 0.9993314791936427
wtick = 2.793651148400146e-007
1/wtick = 3579545
```

See Also

Concepts

OpenMP Functions

omp_init_lock

Initializes a simple lock.

```
void omp_init_lock(
   omp_lock_t *lock
);
```

Parameters

lock

A variable of type omp_lock_t.

Remarks

For more information, see 3.2.1 omp_init_lock and omp_init_nest_lock Functions.

Example

```
// omp_init_lock.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

omp_lock_t my_lock;

int main() {
   omp_init_lock(&my_lock);

   #pragma omp parallel num_threads(4)
   {
      int tid = omp_get_thread_num( );
      int i, j;

      for (i = 0; i < 5; ++i) {
         omp_set_lock(&my_lock);
         printf_s("Thread %d - starting locked region\n", tid);
         printf_s("Thread %d - ending locked region\n", tid);
         omp_unset_lock(&my_lock);
      }
   }

   omp_destroy_lock(&my_lock);
}
```

Sample Output

```
Thread 0 - starting locked region
Thread 0 - ending locked region
Thread 0 - starting locked region
Thread 0 - ending locked region
Thread 0 - starting locked region
Thread 0 - ending locked region
Thread 0 - starting locked region
Thread 0 - ending locked region
Thread 0 - starting locked region
Thread 0 - ending locked region
Thread 1 - starting locked region
Thread 1 - ending locked region
Thread 1 - starting locked region
Thread 1 - ending locked region
Thread 1 - starting locked region
Thread 1 - ending locked region
Thread 1 - starting locked region
Thread 1 - ending locked region
Thread 1 - starting locked region
```

```
Thread 1 - ending locked region
Thread 2 - starting locked region
Thread 2 - ending locked region
Thread 2 - starting locked region
Thread 2 - ending locked region
Thread 2 - starting locked region
Thread 2 - ending locked region
Thread 2 - starting locked region
Thread 2 - ending locked region
Thread 2 - starting locked region
Thread 2 - ending locked region
Thread 3 - starting locked region
Thread 3 - ending locked region
Thread 3 - starting locked region
Thread 3 - ending locked region
Thread 3 - starting locked region
Thread 3 - ending locked region
Thread 3 - starting locked region
Thread 3 - ending locked region
Thread 3 - starting locked region
Thread 3 - ending locked region
```

See Also

Concepts

OpenMP Functions

omp_init_nest_lock

Initializes a lock.

```
void omp_init_nest_lock(
   omp_nest_lock_t *lock
);
```

Remarks

where,

lock

A variable of type omp_nest_lock_t.

Remarks

The initial nesting count is zero.

For more information, see 3.2.1 omp_init_lock and omp_init_nest_lock Functions.

Example

```
// omp_init_nest_lock.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

omp_nest_lock_t my_lock;

void Test() {
   int tid = omp_get_thread_num( );
   omp_set_nest_lock(&my_lock);
   printf_s("Thread %d - starting nested locked region\n", tid);
   printf_s("Thread %d - ending nested locked region\n", tid);
   omp_unset_nest_lock(&my_lock);
}

int main() {
   omp_init_nest_lock(&my_lock);

   #pragma omp parallel num_threads(4)
   {
      int i, j;

      for (i = 0; i < 5; ++i) {
         omp_set_nest_lock(&my_lock);
            if (i % 3)
               Test();
            omp_unset_nest_lock(&my_lock);
         }
     }

     omp_destroy_nest_lock(&my_lock);
}
```

Sample Output

```
Thread 0 - starting nested locked region
Thread 0 - ending nested locked region
Thread 0 - starting nested locked region
Thread 0 - ending nested locked region
Thread 3 - starting nested locked region
Thread 3 - ending nested locked region
Thread 3 - starting nested locked region
```

```
Thread 3 - ending nested locked region
Thread 3 - starting nested locked region
Thread 3 - ending nested locked region
Thread 2 - starting nested locked region
Thread 2 - ending nested locked region
Thread 2 - starting nested locked region
Thread 2 - ending nested locked region
Thread 2 - starting nested locked region
Thread 2 - ending nested locked region
Thread 1 - starting nested locked region
Thread 1 - ending nested locked region
Thread 1 - starting nested locked region
Thread 1 - ending nested locked region
Thread 1 - starting nested locked region
Thread 1 - ending nested locked region
Thread 0 - starting nested locked region
Thread 0 - ending nested locked region
```

See Also

Concepts

OpenMP Functions

omp_in_parallel

Returns nonzero if called from within a parallel region.

```
int omp_in_parallel( );
```

Remarks

For more information, see 3.1.6 omp_in_parallel Function.

Example

```
// omp_in_parallel.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int main( )
{
    omp_set_num_threads(4);
    printf_s("%d\n", omp_in_parallel( ));

    #pragma omp parallel
        #pragma omp master
        {
            printf_s("%d\n", omp_in_parallel( ));
        }
}
```

Output

```
0
1
```

See Also

Concepts

OpenMP Functions

omp_set_dynamic

Indicates that the number of threads available in subsequent parallel region can be adjusted by the run time.

```
void omp_set_dynamic(
   int val
);
```

Remarks

where,

val

A value that indicates if the number of threads available in subsequent parallel region can be adjusted by the runtime. If nonzero, the runtime can adjust the number of threads, if zero, the runtime will not dynamically adjust the number of threads.

Remarks

The number of threads will never exceed the value set by omp_set_num_threads or by OMP_NUM_THREADS.

Use omp_get_dynamic to display the current setting of **omp_set_dynamic**.

The setting for **omp_set_dynamic** will override the setting of the OMP_DYNAMIC environment variable.

For more information, see 3.1.7 omp_set_dynamic Function.

Example

```
// omp_set_dynamic.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int main()
{
    omp_set_dynamic(9);
    omp_set_num_threads(4);
    printf_s("%d\n", omp_get_dynamic( ));
    #pragma omp parallel
        #pragma omp master
        {
            printf_s("%d\n", omp_get_dynamic( ));
        }
}
```

Output

```
1
1
```

See Also

Concepts

OpenMP Functions

omp_set_lock

Blocks thread execution until a lock is available.

```
void omp_set_lock(
   omp_lock_t *lock
);
```

Remarks

where,

lock

A variable of type omp_lock_t that was initialized with omp_init_lock.

Remarks

For more information, see 3.2.3 omp_set_lock and omp_set_nest_lock Functions.

Examples

See omp_init_lock for an example of using **omp_set_lock**.

See Also

Concepts

OpenMP Functions

omp_set_nest_lock

Blocks thread execution until a lock is available.

```
void omp_set_nest_lock(
   omp_nest_lock_t *lock
);
```

Remarks

where,

lock

A variable of type omp_nest_lock_t that was initialized with omp_init_nest_lock.

Remarks

For more information, see 3.2.3 omp_set_lock and omp_set_nest_lock Functions.

Examples

See omp_init_nest_lock for an example of using **omp_set_nest_lock**.

See Also

Concepts

OpenMP Functions

omp_set_nested

Enables nested parallelism.

```
void omp_set_nested(
   int val
);
```

Remarks

where,

val

If nonzero, enables nested parallelism. If zero, disables nested parallelism.

Remarks

OMP nested parallelism can be turned on with **omp_set_nested**, or by setting the OMP_NESTED environment variable.

The setting for **omp_set_nested** will override the setting of the **OMP_NESTED** environment variable.

When enabled, the environment variable can break an otherwise operational program because the number of threads increases exponentially when nesting parallel regions. For example a function that recurses 6 times with the number of OMP threads set to 4 requires 4,096 (4 to the power of 6) threads In general, the performance of your application will degrade if the number of thread exceeds the number of processors. One exception to this would be I/O bound applications.

Use omp_get_nested to display the current setting of **omp_set_nested**.

For more information, see 3.1.9 omp_set_nested Function.

Example

```
// omp_set_nested.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int main( )
{
    omp_set_nested(1);
    omp_set_num_threads(4);
    printf_s("%d\n", omp_get_nested( ));
    #pragma omp parallel
        #pragma omp master
        {
            printf_s("%d\n", omp_get_nested( ));
        }
}
```

Output

```
1
1
```

See Also

Concepts

OpenMP Functions

omp_set_num_threads

Sets the number of threads in subsequent parallel regions, unless overridden by a num_threads clause.

```
void omp_set_num_threads(
   int num_threads
);
```

Remarks

where,

num_threads

The number of threads in the parallel region.

Remarks

For more information, see 3.1.1 omp_set_num_threads Function.

Example

See omp_get_num_threads for an example of using **omp_set_num_threads**.

See Also

Concepts

OpenMP Functions

omp_test_lock

Attempts to set a lock but does not block thread execution.

```
int omp_test_lock(
   omp_lock_t *lock
);
```

Remarks

where,

lock

A variable of type omp_lock_t that was initialized with omp_init_lock.

Remarks

For more information, see 3.2.5 omp_test_lock and omp_test_nest_lock Functions.

Example

```
// omp_test_lock.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

omp_lock_t simple_lock;

int main() {
    omp_init_lock(&simple_lock);

    #pragma omp parallel num_threads(4)
    {
        int tid = omp_get_thread_num();

        while (!omp_test_lock(&simple_lock))
            printf_s("Thread %d - failed to acquire simple_lock\n",
                     tid);

        printf_s("Thread %d - acquired simple_lock\n", tid);

        printf_s("Thread %d - released simple_lock\n", tid);
        omp_unset_lock(&simple_lock);
    }

    omp_destroy_lock(&simple_lock);
}
```

Sample Output

```
Thread 1 - acquired simple_lock
Thread 1 - released simple_lock
Thread 0 - failed to acquire simple_lock
Thread 3 - failed to acquire simple_lock
Thread 0 - failed to acquire simple_lock
Thread 3 - failed to acquire simple_lock
Thread 2 - acquired simple_lock
Thread 0 - failed to acquire simple_lock
Thread 3 - failed to acquire simple_lock
Thread 0 - failed to acquire simple_lock
Thread 3 - failed to acquire simple_lock
Thread 2 - released simple_lock
Thread 0 - failed to acquire simple_lock
Thread 3 - failed to acquire simple_lock
Thread 0 - acquired simple_lock
```

```
Thread 3 - failed to acquire simple_lock
Thread 0 - released simple_lock
Thread 3 - failed to acquire simple_lock
Thread 3 - acquired simple_lock
Thread 3 - released simple_lock
```

See Also

Concepts

OpenMP Functions

omp_test_nest_lock

Attempts to set a nestable lock but does not block thread execution.

```
int omp_test_nest_lock(
   omp_nest_lock_t *lock
);
```

Remarks

where,

lock

A variable of type omp_nest_lock_t that was initialized with omp_init_nest_lock.

Remarks

For more information, see 3.2.5 omp_test_lock and omp_test_nest_lock Functions.

Example

```
// omp_test_nest_lock.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

omp_nest_lock_t nestable_lock;

int main() {
   omp_init_nest_lock(&nestable_lock);

   #pragma omp parallel num_threads(4)
   {
      int tid = omp_get_thread_num();
      while (!omp_test_nest_lock(&nestable_lock))
         printf_s("Thread %d - failed to acquire nestable_lock\n",
                tid);

      printf_s("Thread %d - acquired nestable_lock\n", tid);

      if (omp_test_nest_lock(&nestable_lock)) {
         printf_s("Thread %d - acquired nestable_lock again\n",
                tid);
         printf_s("Thread %d - released nestable_lock\n",
                tid);
         omp_unset_nest_lock(&nestable_lock);
      }

      printf_s("Thread %d - released nestable_lock\n", tid);
      omp_unset_nest_lock(&nestable_lock);
   }

   omp_destroy_nest_lock(&nestable_lock);
}
```

Sample Output

```
Thread 1 - acquired nestable_lock
Thread 0 - failed to acquire nestable_lock
Thread 1 - acquired nestable_lock again
Thread 0 - failed to acquire nestable_lock
Thread 1 - released nestable_lock
Thread 0 - failed to acquire nestable_lock
Thread 1 - released nestable_lock
Thread 0 - failed to acquire nestable_lock
```

```
Thread 3 - acquired nestable_lock
Thread 0 - failed to acquire nestable_lock
Thread 3 - acquired nestable_lock again
Thread 0 - failed to acquire nestable_lock
Thread 2 - failed to acquire nestable_lock
Thread 3 - released nestable_lock
Thread 2 - failed to acquire nestable_lock
Thread 3 - released nestable_lock
Thread 2 - failed to acquire nestable_lock
Thread 0 - acquired nestable_lock
Thread 2 - failed to acquire nestable_lock
Thread 2 - failed to acquire nestable_lock
Thread 2 - failed to acquire nestable_lock
Thread 0 - acquired nestable_lock again
Thread 2 - failed to acquire nestable_lock
Thread 0 - released nestable_lock
Thread 2 - failed to acquire nestable_lock
Thread 0 - released nestable_lock
Thread 2 - failed to acquire nestable_lock
Thread 2 - acquired nestable_lock
Thread 2 - acquired nestable_lock again
Thread 2 - released nestable_lock
Thread 2 - released nestable_lock
```

See Also

Concepts

OpenMP Functions

omp_unset_lock

Releases a lock.

```
void omp_unset_lock(
   omp_lock_t *lock
);
```

Remarks

where,

lock

A variable of type omp_lock_t that was initialized with omp_init_lock, owned by the thread and executing in the function.

Remarks

For more information, see 3.2.4 omp_unset_lock and omp_unset_nest_lock Functions.

Example

See omp_init_lock for an example of using **omp_unset_lock**.

See Also

Concepts

OpenMP Functions

omp_unset_nest_lock

Releases a nestable lock.

```
void omp_unset_nest_lock(
   omp_nest_lock_t *lock
);
```

Remarks

where,

lock

A variable of type omp_nest_lock_t that was initialized with omp_init_nest_lock, owned by the thread and executing in the function.

Remarks

For more information, see 3.2.4 omp_unset_lock and omp_unset_nest_lock Functions.

Example

See omp_init_nest_lock for an example of using **omp_unset_nest_lock**.

See Also

Concepts

OpenMP Functions

OpenMP C and C++ Application Program Interface

Discusses the OpenMP C and C++ API, as documented in the version 2.0 specification from the OpenMP Architecture Review Board.

Version 2.0 March 2002

Copyright 1997-2002 OpenMP Architecture Review Board.

Permission to copy without fee all or part of this material is granted, provided the OpenMP Architecture Review Board copyright notice and the title of this document appear. Notice is given that copying is by permission of OpenMP Architecture Review Board.

See Also

Concepts

Contents

Other Resources

OpenMP in Visual C++

Contents

1. Introduction

1.1 Scope

1.2 Definition of Terms

1.3 Execution Model

1.4 Compliance

1.5 Normative References

1.6 Organization

2. Directives

2.1 Directive Format

2.2 Conditional Compilation

2.3 parallel Construct

2.4 Work-sharing Constructs

2.4.1 for Construct

2.4.2 sections Construct

2.4.3 single Construct

2.5 Combined Parallel Work-sharing Constructs

2.5.1 parallel for Construct

2.5.2 parallel sections Construct

2.6 Master and Synchronization Directives

2.6.1 master Construct

2.6.2 critical Construct

2.6.3 barrier Directive

2.6.4 atomic Construct

2.6.5 flush Directive

2.6.6 ordered Construct

2.7 Data Environment

2.7.1 threadprivate Directive

2.7.2 Data-Sharing Attribute Clauses

2.7.2.1 private

2.7.2.2 firstprivate

2.7.2.3 lastprivate

2.7.2.4 shared

2.7.2.5 default

2.7.2.6 reduction

2.7.2.7 copyin

2.7.2.8 copyprivate

2.8 Directive Binding

2.9 Directive Nesting
3. Run-time Library Functions
3.1 Execution Environment Functions
3.1.1 omp_set_num_threads Function
3.1.2 omp_get_num_threads Function
3.1.3 omp_get_max_threads Function
3.1.4 omp_get_thread_num Function
3.1.5 omp_get_num_procs Function
3.1.6 omp_in_parallel Function
3.1.7 omp_set_dynamic Function
3.1.8 omp_get_dynamic Function
3.1.9 omp_set_nested Function
3.1.10 omp_get_nested Function
3.2 Lock Functions
3.2.1 omp_init_lock and omp_init_nest_lock Functions
3.2.2 omp_destroy_lock and omp_destroy_nest_lock Functions
3.2.3 omp_set_lock and omp_set_nest_lock Functions
3.2.4 omp_unset_lock and omp_unset_nest_lock Functions
3.2.5 omp_test_lock and omp_test_nest_lock Functions
3.3 Timing Routines
3.3.1 omp_get_wtime Function
3.3.2 omp_get_wtick Function
4. Environment Variables
4.1 OMP_SCHEDULE
4.2 OMP_NUM_THREADS
4.3 OMP_DYNAMIC
4.4 OMP_NESTED
A. Examples
A.1 Executing a Simple Loop in Parallel
A.2 Specifying Conditional Compilation
A.3 Using Parallel Regions
A.4 Using the nowait Clause
A.5 Using the critical Directive
A.6 Using the lastprivate Clause
A.7 Using the reduction Clause
A.8 Specifying Parallel Sections
A.9 Using single Directives
A.10 Specifying Sequential Ordering
A.11 Specifying a Fixed Number of Threads
A.12 Using the atomic Directive

A.13 Using the flush Directive with a List

A.14 Using the flush Directive without a List

A.15 Determining the Number of Threads Used

A.16 Using Locks

A.17 Using Nestable Locks

A.18 Nested for Directives

A.19 Examples Showing Incorrect Nesting of Work-sharing Directives

A.20 Binding of barrier Directives

A.21 Scoping Variables with the private Clause

A.22 Using the default(none) Clause

A.23 Examples of the ordered Directive

A.24 Example of the private Clause

A.25 Examples of the copyprivate Data Attribute Clause

A.26 Using the threadprivate Directive

A.27 Use of C99 Variable Length Arrays

A.28 Use of num_threads Clause

A.29 Use of Work-Sharing Constructs Inside a critical Construct

A.30 Use of Reprivatization

A.31 Thread-Safe Lock Functions

B. Stubs for Run-time Library Functions

C. OpenMP C and C++ Grammar

C.1 Notation

C.2 Rules

D. Using the schedule Clause

E. Implementation-Defined Behaviors in OpenMP C/C++

F. New Features and Clarifications in Version 2.0

See Also

Concepts

OpenMP C and C++ Application Program Interface

1. Introduction

This document specifies a collection of compiler directives, library functions, and environment variables that can be used to specify shared-memory parallelism in C and C++ programs. The functionality described in this document is collectively known as the *OpenMP C/C++ Application Program Interface (API)*. The goal of this specification is to provide a model for parallel programming that allows a program to be portable across shared-memory architectures from different vendors. The OpenMP C/C++ API will be supported by compilers from numerous vendors. More information about OpenMP, including the *OpenMP Fortran Application Program Interface*, can be found at the following web site:

http://www.openmp.org

The directives, library functions, and environment variables defined in this document will allow users to create and manage parallel programs while permitting portability. The directives extend the C and C++ sequential programming model with single program multiple data (SPMD) constructs, work-sharing constructs, and synchronization constructs, and they provide support for the sharing and privatization of data. Compilers that support the OpenMP C and C++ API will include a command-line option to the compiler that activates and allows interpretation of all OpenMP compiler directives.

1.1 Scope

This specification covers only user-directed parallelization, wherein the user explicitly specifies the actions to be taken by the compiler and run-time system in order to execute the program in parallel. OpenMP C and C++ implementations are not required to check for dependencies, conflicts, deadlocks, race conditions, or other problems that result in incorrect program execution. The user is responsible for ensuring that the application using the OpenMP C and C++ API constructs executes correctly. Compiler-generated automatic parallelization and directives to the compiler to assist such parallelization are not covered in this document.

1.2 Definition of Terms

The following terms are used in this document:

barrier

A synchronization point that must be reached by all threads in a team. Each thread waits until all threads in the team arrive at this point. There are explicit barriers identified by directives and implicit barriers created by the implementation.

construct

A construct is a statement. It consists of a directive and the subsequent structured block. Note that some directives are not part of a construct. (See *openmp-directive* in Appendix C).

directive

A C or C++ **#pragma** followed by the **omp** identifier, other text, and a new line. The directive specifies program behavior.

dynamic extent

All statements in the *lexical extent*, plus any statement inside a function that is executed as a result of the execution of statements within the lexical extent. A dynamic extent is also referred to as a *region*.

lexical extent

Statements lexically contained within a *structured block*.

master thread

The thread that creates a team when a *parallel region* is entered.

parallel region

Statements that bind to an OpenMP parallel construct and may be executed by multiple threads.

private

A private variable names a block of storage that is unique to the thread making the reference. Note that there are several ways to specify that a variable is private: a definition within a parallel region, a **threadprivate** directive, a **private**, **firstprivate**, **lastprivate**, or **reduction** clause, or use of the variable as a **for**loop control variable in a **for** loop immediately following a **for** or **parallel for** directive.

region

A dynamic extent.

serial region

Statements executed only by the *master thread* outside of the dynamic extent of any *parallel region*.

serialize

To execute a parallel construct with a team of threads consisting of only a single thread (which is the master thread for that parallel construct), with serial order of execution for the statements within the structured block (the same order as if the block were not part of a parallel construct), and with no effect on the value returned by **omp_in_parallel()** (apart from the effects of any nested parallel constructs).

shared

A shared variable names a single block of storage. All threads in a team that access this variable will access this single block of storage.

structured block

A structured block is a statement (single or compound) that has a single entry and a single exit. No statement is a structured block if there is a jump into or out of that statement (including a call to **longjmp**(3C) or the use of **throw**, but a call to **exit** is permitted). A compound statement is a structured block if its execution always begins at the opening **{** and always ends at the closing **}**. An expression statement, selection statement, iteration statement, or **try** block is a structured block if the corresponding compound statement obtained by enclosing it in **{** and **}**would be a structured block. A jump statement, labeled statement, or declaration statement is not a structured block.

team

One or more threads cooperating in the execution of a construct.

thread

An execution entity having a serial flow of control, a set of private variables, and access to shared variables.

variable

An identifier, optionally qualified by namespace names, that names an object.

1.3 Execution Model

OpenMP uses the fork-join model of parallel execution. Although this fork-join model can be useful for solving a variety of problems, it is somewhat tailored for large array-based applications. OpenMP is intended to support programs that will execute correctly both as parallel programs (multiple threads of execution and a full OpenMP support library) and as sequential programs (directives ignored and a simple OpenMP stubs library). However, it is possible and permitted to develop a program that does not behave correctly when executed sequentially. Furthermore, different degrees of parallelism may result in different numeric results because of changes in the association of numeric operations. For example, a serial addition reduction may have a different pattern of addition associations than a parallel reduction. These different associations may change the results of floating-point addition.

A program written with the OpenMP C/C++ API begins execution as a single thread of execution called the *master thread*. The master thread executes in a serial region until the first parallel construct is encountered. In the OpenMP C/C++ API, the **parallel** directive constitutes a parallel construct. When a parallel construct is encountered, the master thread creates a team of threads, and the master becomes master of the team. Each thread in the team executes the statements in the dynamic extent of a parallel region, except for the work-sharing constructs. Work-sharing constructs must be encountered by all threads in the team in the same order, and the statements within the associated structured block are executed by one or more of the threads. The barrier implied at the end of a work-sharing construct without a **nowait** clause is executed by all threads in the team.

If a thread modifies a shared object, it affects not only its own execution environment, but also those of the other threads in the program. The modification is guaranteed to be complete, from the point of view of one of the other threads, at the next sequence point (as defined in the base language) only if the object is declared to be volatile. Otherwise, the modification is guaranteed to be complete after first the modifying thread, and then (or concurrently) the other threads, encounter a **flush** directive that specifies the object (either implicitly or explicitly). Note that when the **flush** directives that are implied by other OpenMP directives are not sufficient to ensure the desired ordering of side effects, it is the programmer's responsibility to supply additional, explicit **flush** directives.

Upon completion of the parallel construct, the threads in the team synchronize at an implicit barrier, and only the master thread continues execution. Any number of parallel constructs can be specified in a single program. As a result, a program may fork and join many times during execution.

The OpenMP C/C++ API allows programmers to use directives in functions called from within parallel constructs. Directives that do not appear in the lexical extent of a parallel construct but may lie in the dynamic extent are called *orphaned* directives. Orphaned directives give programmers the ability to execute major portions of their program in parallel with only minimal changes to the sequential program. With this functionality, users can code parallel constructs at the top levels of the program call tree and use directives to control execution in any of the called functions.

Unsynchronized calls to C and C++ output functions that write to the same file may result in output in which data written by different threads appears in nondeterministic order. Similarly, unsynchronized calls to input functions that read from the same file may read data in nondeterministic order. Unsynchronized use of I/O, such that each thread accesses a different file, produces the same results as serial execution of the I/O functions.

1.4 Compliance

An implementation of the OpenMP C/C++ API is *OpenMP-compliant* if it recognizes and preserves the semantics of all the elements of this specification, as laid out in Chapters 1, 2, 3, 4, and Appendix C. Appendices A, B, D, E, and F are for information purposes only and are not part of the specification. Implementations that include only a subset of the API are not OpenMP-compliant.

The OpenMP C and C++ API is an extension to the base language that is supported by an implementation. If the base language does not support a language construct or extension that appears in this document, the OpenMP implementation is not required to support it.

All standard C and C++ library functions and built-in functions (that is, functions of which the compiler has specific knowledge) must be thread-safe. Unsynchronized use of thread–safe functions by different threads inside a parallel region does not produce undefined behavior. However, the behavior might not be the same as in a serial region. (A random number generation function is an example.)

The OpenMP C/C++ API specifies that certain behavior is *implementation-defined*. A conforming OpenMP implementation is required to define and document its behavior in these cases. See Appendix E, page 97, for a list of implementation-defined behaviors.

1.5 Normative References

- ISO/IEC 9899:1999, *Information Technology - Programming Languages - C*. This OpenMP API specification refers to ISO/IEC 9899:1999 as C99.
- ISO/IEC 9899:1990, *Information Technology - Programming Languages - C*. This OpenMP API specification refers to ISO/IEC 9899:1990 as C90.
- ISO/IEC 14882:1998, *Information Technology - Programming Languages - C++*. This OpenMP API specification refers to ISO/IEC 14882:1998 as C++.

Where this OpenMP API specification refers to C, reference is made to the base language supported by the implementation.

1.6 Organization

- Directives (see Chapter 2).
- Run-time library functions (see Chapter 3).
- Environment variables (see Chapter 4).
- Examples (see Appendix A).
- Stubs for the run-time library (see Appendix B).
- OpenMP Grammar for C and C++ (see Appendix C).
- Using the **schedule** clause (see Appendix D).
- Implementation-defined behaviors in OpenMP C/C++ (see Appendix E).
- New features in OpenMP C/C++ Version 2.0 (see Appendix F).

2. Directives

Directives are based on **#pragma** directives defined in the C and C++ standards. Compilers that support the OpenMP C and C++ API will include a command-line option that activates and allows interpretation of all OpenMP compiler directives.

2.1 Directive Format

The syntax of an OpenMP directive is formally specified by the grammar in Appendix C, and informally as follows:

```
#pragma omp directive-name  [clause[ [,] clause]...] new-line
```

Each directive starts with **#pragma omp**, to reduce the potential for conflict with other (non-OpenMP or vendor extensions to OpenMP) pragma directives with the same names. The remainder of the directive follows the conventions of the C and C++ standards for compiler directives. In particular, white space can be used before and after the **#**, and sometimes white space must be used to separate the words in a directive. Preprocessing tokens following the **#pragma omp** are subject to macro replacement.

Directives are case-sensitive. The order in which clauses appear in directives is not significant. Clauses on directives may be repeated as needed, subject to the restrictions listed in the description of each clause. If *variable-list* appears in a clause, it must specify only variables. Only one *directive-name* can be specified per directive. For example, the following directive is not allowed:

```
/* ERROR - multiple directive names not allowed */
#pragma omp parallel barrier
```

An OpenMP directive applies to at most one succeeding statement, which must be a structured block.

2.2 Conditional Compilation

The **_OPENMP** macro name is defined by OpenMP-compliant implementations as the decimal constant *yyyymm*, which will be the year and month of the approved specification. This macro must not be the subject of a **#define** or a **#undef** preprocessing directive.

```
#ifdef _OPENMP
iam = omp_get_thread_num() + index;
#endif
```

If vendors define extensions to OpenMP, they may specify additional predefined macros.

2.3 parallel Construct

The following directive defines a parallel region, which is a region of the program that is to be executed by multiple threads in parallel. This is the fundamental construct that starts parallel execution.

```
#pragma omp parallel [clause[ [, ]clause] ...] new-line
   structured-block
```

The *clause* is one of the following:

if(*scalar-expression* **)**

private(*variable-list* **)**

firstprivate(*variable-list* **)**

default(shared | none)

shared(*variable-list* **)**

copyin(*variable-list* **)**

reduction(*operator* **:** *variable-list* **)**

num_threads(*integer-expression* **)**

When a thread encounters a parallel construct, a team of threads is created if one of the following cases is true:

- No **if** clause is present.
- The **if** expression evaluates to a nonzero value.

This thread becomes the master thread of the team, with a thread number of 0, and all threads in the team, including the master thread, execute the region in parallel. If the value of the **if** expression is zero, the region is serialized.

To determine the number of threads that are requested, the following rules will be considered in order. The first rule whose condition is met will be applied:

1. If the **num_threads** clause is present, then the value of the integer expression is the number of threads requested.
2. If the **omp_set_num_threads** library function has been called, then the value of the argument in the most recently executed call is the number of threads requested.
3. If the environment variable **OMP_NUM_THREADS** is defined, then the value of this environment variable is the number of threads requested.
4. If none of the methods above were used, then the number of threads requested is implementation-defined.

If the **num_threads** clause is present then it supersedes the number of threads requested by the **omp_set_num_threads** library function or the **OMP_NUM_THREADS** environment variable only for the parallel region it is applied to. Subsequent parallel regions are not affected by it.

The number of threads that execute the parallel region also depends upon whether or not dynamic adjustment of the number of threads is enabled. If dynamic adjustment is disabled, then the requested number of threads will execute the parallel region. If dynamic adjustment is enabled then the requested number of threads is the maximum number of threads that may execute the parallel region.

If a parallel region is encountered while dynamic adjustment of the number of threads is disabled, and the number of threads requested for the parallel region exceeds the number that the run-time system can supply, the behavior of the program is implementation-defined. An implementation may, for example, interrupt the execution of the program, or it may serialize the parallel region.

The **omp_set_dynamic** library function and the **OMP_DYNAMIC** environment variable can be used to enable and disable dynamic adjustment of the number of threads.

The number of physical processors actually hosting the threads at any given time is implementation-defined. Once created, the number of threads in the team remains constant for the duration of that parallel region. It can be changed either explicitly by the user or automatically by the run-time system from one parallel region to another.

The statements contained within the dynamic extent of the parallel region are executed by each thread, and each thread can execute a path of statements that is different from the other threads. Directives encountered outside the lexical extent of a parallel region are referred to as orphaned directives.

There is an implied barrier at the end of a parallel region. Only the master thread of the team continues execution at the end of a parallel region.

If a thread in a team executing a parallel region encounters another parallel construct, it creates a new team, and it becomes the master of that new team. Nested parallel regions are serialized by default. As a result, by default, a nested parallel region is executed by a team composed of one thread. The default behavior may be changed by using either the runtime library function **omp_set_nested** or the environment variable **OMP_NESTED**. However, the number of threads in a team that execute a nested parallel region is implementation-defined.

Restrictions to the **parallel** directive are as follows:

- At most one **if** clause can appear on the directive.
- It is unspecified whether any side effects inside the if expression or **num_threads** expression occur.
- A **throw** executed inside a parallel region must cause execution to resume within the dynamic extent of the same structured block, and it must be caught by the same thread that threw the exception.
- Only a single **num_threads** clause can appear on the directive. The **num_threads** expression is evaluated outside the context of the parallel region, and must evaluate to a positive integer value.
- The order of evaluation of the **if** and **num_threads** clauses is unspecified.

Cross References:

- **private**, **firstprivate**, **default**, **shared**, **copyin**, and **reduction** clauses, see Section 2.7.2 on page 25.
- **OMP_NUM_THREADS** environment variable, Section 4.2 on page 48.
- **omp_set_dynamic** library function, see Section 3.1.7 on page 39.
- **OMP_DYNAMIC** environment variable, see Section 4.3 on page 49.
- **omp_set_nested** function, see Section 3.1.9 on page 40.
- **OMP_NESTED** environment variable, see Section 4.4 on page 49.
- **omp_set_num_threads** library function, see Section 3.1.1 on page 36.

2.4 Work-sharing Constructs

A work-sharing construct distributes the execution of the associated statement among the members of the team that encounter it. The work-sharing directives do not launch new threads, and there is no implied barrier on entry to a work-sharing construct.

The sequence of work-sharing constructs and **barrier** directives encountered must be the same for every thread in a team.

OpenMP defines the following work-sharing constructs, and these are described in the sections that follow:

- **for** directive
- **sections** directive
- **single** directive

2.4.1 for Construct

The **for** directive identifies an iterative work-sharing construct that specifies that the iterations of the associated loop will be executed in parallel. The iterations of the **for** loop are distributed across threads that already exist in the team executing the parallel construct to which it binds. The syntax of the **for** construct is as follows:

```
#pragma omp for [clause[[,] clause] ... ] new-line
   for-loop
```

The clause is one of the following:

private(*variable-list* **)**

firstprivate(*variable-list* **)**

lastprivate(*variable-list* **)**

reduction(*operator* **:** *variable-list***)**

ordered

schedule(*kind*[, *chunk_size*]**)**

nowait

The **for** directive places restrictions on the structure of the corresponding **for** loop. Specifically, the corresponding **for** loop must have canonical shape:

for (*init-expr* **;** *var logical-op b*; *incr-expr***)**

init-expr

One of the following:

var = lb

integer-type var = lb

incr-expr

One of the following:

++var

var ++

-- var

var --

var += incr

var -= incr

var = var + incr

var = incr + var

var = var - incr

var

A signed integer variable. If this variable would otherwise be shared, it is implicitly made private for the duration of the **for**. This variable must not be modified within the body of the **for** statement. Unless the variable is specified **lastprivate**, its value after the loop is indeterminate.

logical-op

One of the following:

<

<=

>

>=

lb, *b*, and *incr*

Loop invariant integer expressions. There is no synchronization during the evaluation of these expressions. Thus, any evaluated side effects produce indeterminate results.

Note that the canonical form allows the number of loop iterations to be computed on entry to the loop. This computation is performed with values in the type of *var*, after integral promotions. In particular, if value of *b* - *lb* + *incr* cannot be represented in that type, the result is indeterminate. Further, if *logical-op* is < or <= then *incr-expr* must cause *var* to increase on each iteration of the loop. If *logical-op* is > or >= then *incr-expr* must cause *var* to decrease on each iteration of the loop.

The **schedule** clause specifies how iterations of the **for** loop are divided among threads of the team. The correctness of a program must not depend on which thread executes a particular iteration. The value of *chunk_size*, if specified, must be a loop invariant integer expression with a positive value. There is no synchronization during the evaluation of this expression. Thus, any evaluated side effects produce indeterminate results. The schedule *kind* can be one of the following:

TABLE 2-1 **schedule** clause *kind* values

static	When **schedule(static,** *chunk_size***)** is specified, iterations are divided into chunks of a size specified by *chunk_size*. The chunks are statically assigned to threads in the team in a round-robin fashion in the order of the thread number. When no *chunk_size* is specified, the iteration space is divided into chunks that are approximately equal in size, with one chunk assigned to each thread.
dynamic	When **schedule(dynamic,** *chunk_size***)** is specified, the iterations are divided into a series of chunks, each containing *chunk_size* iterations. Each chunk is assigned to a thread that is waiting for an assignment. The thread executes the chunk of iterations and then waits for its next assignment, until no chunks remain to be assigned. Note that the last chunk to be assigned may have a smaller number of iterations. When no *chunk_size* is specified, it defaults to 1.
guided	When **schedule(guided,** *chunk_size***)** is specified, the iterations are assigned to threads in chunks with decreasing sizes. When a thread finishes its assigned chunk of iterations, it is dynamically assigned another chunk, until none remain. For a *chunk_size* of 1, the size of each chunk is approximately the number of unassigned iterations divided by the number of threads. These sizes decrease approximately exponentially to 1. For a *chunk_size* with value *k* greater than 1, the sizes decrease approximately exponentially to *k*, except that the last chunk may have fewer than *k* iterations. When no *chunk_size* is specified, it defaults to 1.
runtime	When **schedule(runtime)** is specified, the decision regarding scheduling is deferred until runtime. The schedule *kind* and size of the chunks can be chosen at run time by setting the environment variable **OMP_SCHEDULE**. If this environment variable is not set, the resulting schedule is implementation-defined. When **schedule(runtime)** is specified, *chunk_size* must not be specified.

In the absence of an explicitly defined **schedule** clause, the default **schedule** is implementation-defined.

An OpenMP-compliant program should not rely on a particular schedule for correct execution. A program should not rely on a schedule *kind* conforming precisely to the description given above, because it is possible to have variations in the implementations of the same schedule *kind* across different compilers. The descriptions can be used to select the schedule that is appropriate for a particular situation.

The **ordered** clause must be present when **ordered** directives bind to the **for** construct.

There is an implicit barrier at the end of a **for** construct unless a **nowait** clause is specified.

Restrictions to the **for** directive are as follows:

- The **for** loop must be a structured block, and, in addition, its execution must not be terminated by a **break** statement.
- The values of the loop control expressions of the **for** loop associated with a **for** directive must be the same for all the threads in the team.

- The **for** loop iteration variable must have a signed integer type.
- Only a single **schedule** clause can appear on a **for** directive.
- Only a single **ordered** clause can appear on a **for** directive.
- Only a single **nowait** clause can appear on a **for** directive.
- It is unspecified if or how often any side effects within the *chunk_size*, *lb*, *b*, or *incr* expressions occur.
- The value of the *chunk_size* expression must be the same for all threads in the team.

Cross References:

- **private**, **firstprivate**, **lastprivate**, and **reduction** clauses, see Section 2.7.2 on page 25.
- **OMP_SCHEDULE** environment variable, see Section 4.1 on page 48.
- **ordered** construct, see Section 2.6.6 on page 22.
- Appendix D, page 93, gives more information on using the schedule clause.

2.4.2 sections Construct

The **sections** directive identifies a noniterative work-sharing construct that specifies a set of constructs that are to be divided among threads in a team. Each section is executed once by a thread in the team. The syntax of the **sections** directive is as follows:

```
#pragma omp sections [clause[[,] clause] ...] new-line
   {
   [#pragma omp section new-line]
      structured-block
   [#pragma omp section new-line
      structured-block ]
...
}
```

The clause is one of the following:

private(*variable-list* **)**

firstprivate(*variable-list* **)**

lastprivate(*variable-list* **)**

reduction(*operator* **:** *variable-list* **)**

nowait

Each section is preceded by a **section** directive, although the **section** directive is optional for the first section. The **section** directives must appear within the lexical extent of the **sections** directive. There is an implicit barrier at the end of a **sections** construct, unless a **nowait** is specified.

Restrictions to the **sections** directive are as follows:

- A **section** directive must not appear outside the lexical extent of the **sections** directive.
- Only a single **nowait** clause can appear on a **sections** directive.

Cross References:

- **private**, **firstprivate**, **lastprivate**, and **reduction** clauses, see Section 2.7.2 on page 25.

2.4.3 single Construct

The **single** directive identifies a construct that specifies that the associated structured block is executed by only one thread in the team (not necessarily the master thread). The syntax of the **single** directive is as follows:

```
#pragma omp single [clause[[,] clause] ...] new-line
   structured-block
```

The clause is one of the following:

private(*variable-list* **)**

firstprivate(*variable-list* **)**

copyprivate(*variable-list* **)**

nowait

There is an implicit barrier after the **single** construct unless a **nowait** clause is specified.

Restrictions to the **single** directive are as follows:

- Only a single **nowait** clause can appear on a **single** directive.
- The **copyprivate** clause must not be used with the **nowait** clause.

Cross References:

- **private**, **firstprivate**, and **copyprivate** clauses, see Section 2.7.2 on page 25.

2.5 Combined Parallel Work-sharing Constructs

Combined parallel work–sharing constructs are shortcuts for specifying a parallel region that contains only one work-sharing construct. The semantics of these directives are identical to that of explicitly specifying a **parallel** directive followed by a single work-sharing construct.

The following sections describe the combined parallel work-sharing constructs:

- the **parallel for** directive.
- the **parallel sections** directive.

2.5.1 parallel for Construct

The **parallel for** directive is a shortcut for a **parallel** region that contains only a single **for** directive. The syntax of the **parallel for** directive is as follows:

```
#pragma omp parallel for [clause[[,] clause] ...] new-line
   for-loop
```

This directive allows all the clauses of the **parallel** directive and the **for** directive, except the **nowait** clause, with identical meanings and restrictions. The semantics are identical to explicitly specifying a **parallel** directive immediately followed by a **for** directive.

Cross References:

- **parallel** directive, see Section 2.3 on page 8.
- **for** directive, see Section 2.4.1 on page 11.
- Data attribute clauses, see 2.7.2 Data-Sharing Attribute Clauses on page 25.

2.5.2 parallel sections Construct

The **parallel sections** directive provides a shortcut form for specifying a **parallel** region containing only a single **sections** directive. The semantics are identical to explicitly specifying a **parallel** directive immediately followed by a **sections** directive. The syntax of the **parallel sections** directive is as follows:

```
#pragma omp parallel sections  [clause[[,] clause] ...] new-line
   {
   [#pragma omp section new-line]
      structured-block
   [#pragma omp section new-line
      structured-block  ]
   ...
}
```

The *clause* can be one of the clauses accepted by the **parallel** and **sections** directives, except the **nowait** clause.

Cross References:

- **parallel** directive, see Section 2.3 on page 8.
- **sections** directive, see Section 2.4.2 on page 14.

2.6 Master and Synchronization Directives

The following sections describe :

- the **master** construct.
- the **critical** construct.
- the **barrier** directive.
- the **atomic** construct.
- the **flush** directive.
- the **ordered** construct.

2.6.1 master Construct

The **master** directive identifies a construct that specifies a structured block that is executed by the master thread of the team. The syntax of the **master** directive is as follows:

```
#pragma omp master new-line
   structured-block
```

Other threads in the team do not execute the associated structured block. There is no implied barrier either on entry to or exit from the master construct.

2.6.2 critical Construct

The **critical** directive identifies a construct that restricts execution of the associated structured block to a single thread at a time. The syntax of the **critical** directive is as follows:

```
#pragma omp critical [(name)]  new-line
   structured-block
```

An optional *name* may be used to identify the critical region. Identifiers used to identify a critical region have external linkage and are in a name space which is separate from the name spaces used by labels, tags, members, and ordinary identifiers.

A thread waits at the beginning of a critical region until no other thread is executing a critical region (anywhere in the program) with the same name. All unnamed **critical** directives map to the same unspecified name.

2.6.3 barrier Directive

The **barrier** directive synchronizes all the threads in a team. When encountered, each thread in the team waits until all of the others have reached this point. The syntax of the **barrier** directive is as follows:

```
#pragma omp barrier new-line
```

After all threads in the team have encountered the barrier, each thread in the team begins executing the statements after the barrier directive in parallel. Note that because the **barrier** directive does not have a C language statement as part of its syntax, there are some restrictions on its placement within a program. See Appendix C for the formal grammar. The example below illustrates these restrictions.

```
/* ERROR - The barrier directive cannot be the immediate
*          substatement of an if statement
*/
if (x!=0)
   #pragma omp barrier
...

/* OK - The barrier directive is enclosed in a
*      compound statement.
*/
if (x!=0) {
   #pragma omp barrier
}
```

2.6.4 atomic Construct

The **atomic** directive ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads. The syntax of the **atomic** directive is as follows:

```
#pragma omp atomic new-line
   expression-stmt
```

The expression statement must have one of the following forms:

x binop= expr

x++

++x

x--

--x

In the preceding expressions:

- *x* is an lvalue expression with scalar type.
- *expr* is an expression with scalar type, and it does not reference the object designated by *x*.
- *binop* is not an overloaded operator and is one of +, *, -, /, &, ^, |, <<, or >>.

Although it is implementation-defined whether an implementation replaces all **atomic** directives with **critical** directives that have the same unique *name*, the **atomic** directive permits better optimization. Often hardware instructions are available that can perform the atomic update with the least overhead.

Only the load and store of the object designated by *x* are atomic; the evaluation of *expr* is not atomic. To avoid race conditions, all updates of the location in parallel should be protected with the **atomic** directive, except those that are known to be free of race conditions.

Restrictions to the **atomic** directive are as follows:

- All atomic references to the storage location x throughout the program are required to have a compatible type.

Examples:

```
extern float a[], *p = a, b;
/* Protect against races among multiple updates. */
#pragma omp atomic
a[index[i]] += b;
/* Protect against races with updates through a. */
#pragma omp atomic
p[i] -= 1.0f;

extern union {int n; float x;} u;
/* ERROR - References through incompatible types. */
#pragma omp atomic
u.n++;
#pragma omp atomic
u.x -= 1.0f;
```

2.6.5 flush Directive

The **flush** directive, whether explicit or implied, specifies a "cross-thread" sequence point at which the implementation is required to ensure that all threads in a team have a consistent view of certain objects (specified below) in memory. This means that previous evaluations of expressions that reference those objects are complete and subsequent evaluations have not yet begun. For example, compilers must restore the values of the objects from registers to memory, and hardware may need to flush write buffers to memory and reload the values of the objects from memory.

The syntax of the **flush** directive is as follows:

```
#pragma omp flush [(variable-list)]  new-line
```

If the objects that require synchronization can all be designated by variables, then those variables can be specified in the optional *variable-list*. If a pointer is present in the *variable-list*, the pointer itself is flushed, not the object the pointer refers to.

A **flush** directive without a *variable-list* synchronizes all shared objects except inaccessible objects with automatic storage duration. (This is likely to have more overhead than a **flush** with a *variable-list*.) A **flush** directive without a *variable-list* is implied for the following directives:

- **barrier**
- At entry to and exit from **critical**
- At entry to and exit from **ordered**
- At entry to and exit from **parallel**
- At exit from **for**
- At exit from **sections**
- At exit from **single**
- At entry to and exit from **parallel for**
- At entry to and exit from **parallel sections**

The directive is not implied if a **nowait** clause is present. It should be noted that the **flush** directive is not implied for any of the following:

- At entry to **for**
- At entry to or exit from **master**
- At entry to **sections**
- At entry to **single**

A reference that accesses the value of an object with a volatile-qualified type behaves as if there were a **flush** directive specifying that object at the previous sequence point. A reference that modifies the value of an object with a volatile-qualified type behaves as if there were a **flush** directive specifying that object at the subsequent sequence point.

Note that because the **flush** directive does not have a C language statement as part of its syntax, there are some restrictions on its placement within a program. See Appendix C for the formal grammar. The example below illustrates these restrictions.

```
/* ERROR - The flush directive cannot be the immediate
*          substatement of an if statement.
*/
if (x!=0)
   #pragma omp flush (x)
...

/* OK - The flush directive is enclosed in a
*      compound statement
*/
if (x!=0) {
```

```
    #pragma omp flush (x)
}
```

Restrictions to the **flush** directive are as follows:

- A variable specified in a **flush** directive must not have a reference type.

2.6.6 ordered Construct

The structured block following an **ordered** directive is executed in the order in which iterations would be executed in a sequential loop. The syntax of the **ordered** directive is as follows:

```
#pragma omp ordered new-line
   structured-block
```

An **ordered** directive must be within the dynamic extent of a **for** or **parallel for** construct. The **for** or **parallel for** directive to which the **ordered** construct binds must have an **ordered** clause specified as described in Section 2.4.1 on page 11. In the execution of a **for** or **parallel for** construct with an **ordered** clause, **ordered** constructs are executed strictly in the order in which they would be executed in a sequential execution of the loop.

Restrictions to the **ordered** directive are as follows:

- An iteration of a loop with a **for** construct must not execute the same ordered directive more than once, and it must not execute more than one **ordered** directive.

2.7 Data Environment

This section presents a directive and several clauses for controlling the data environment during the execution of parallel regions, as follows:

- A **threadprivate** directive (see the following section) is provided to make file- scope, namespace-scope, or static block-scope variables local to a thread.
- Clauses that may be specified on the directives to control the sharing attributes of variables for the duration of the parallel or work-sharing constructs are described in Section 2.7.2 on page 25.

2.7.1 threadprivate Directive

The **threadprivate** directive makes the named file-scope, namespace-scope, or static block-scope variables specified in the *variable-list* private to a thread. *variable-list* is a comma-separated list of variables that do not have an incomplete type. The syntax of the **threadprivate** directive is as follows:

```
#pragma omp threadprivate(variable-list) new-line
```

Each copy of a **threadprivate** variable is initialized once, at an unspecified point in the program prior to the first reference to that copy, and in the usual manner (i.e., as the master copy would be initialized in a serial execution of the program). Note that if an object is referenced in an explicit initializer of a **threadprivate** variable, and the value of the object is modified prior to the first reference to a copy of the variable, then the behavior is unspecified.

As with any private variable, a thread must not reference another thread's copy of a **threadprivate** object. During serial regions and master regions of the program, references will be to the master thread's copy of the object.

After the first parallel region executes, the data in the **threadprivate** objects is guaranteed to persist only if the dynamic threads mechanism has been disabled and if the number of threads remains unchanged for all parallel regions.

The restrictions to the **threadprivate** directive are as follows:

- A **threadprivate** directive for file-scope or namespace-scope variables must appear outside any definition or declaration, and must lexically precede all references to any of the variables in its list.
- Each variable in the *variable-list* of a **threadprivate** directive at file or namespace scope must refer to a variable declaration at file or namespace scope that lexically precedes the directive.
- A **threadprivate** directive for static block-scope variables must appear in the scope of the variable and not in a nested scope. The directive must lexically precede all references to any of the variables in its list.
- Each variable in the *variable-list* of a **threadprivate** directive in block scope must refer to a variable declaration in the same scope that lexically precedes the directive. The variable declaration must use the static storage-class specifier.
- If a variable is specified in a **threadprivate** directive in one translation unit, it must be specified in a **threadprivate** directive in every translation unit in which it is declared.
- A **threadprivate** variable must not appear in any clause except the **copyin**, **copyprivate**, **schedule**, **num_threads**, or the **if** clause.
- The address of a **threadprivate** variable is not an address constant.
- A **threadprivate** variable must not have an incomplete type or a reference type.
- A **threadprivate** variable with non-POD class type must have an accessible, unambiguous copy constructor if it is declared with an explicit initializer.

The following example illustrates how modifying a variable that appears in an initializer can cause unspecified behavior, and also how to avoid this problem by using an auxiliary object and a copy-constructor.

```
int x = 1;
T a(x);
const T b_aux(x); /* Capture value of x = 1 */
T b(b_aux);
#pragma omp threadprivate(a, b)

void f(int n) {
   x++;
   #pragma omp parallel for
   /* In each thread:
   * Object a is constructed from x (with value 1 or 2?)
   * Object b is copy-constructed from b_aux
   */
   for (int i=0; i<n; i++) {
      g(a, b); /* Value of a is unspecified. */
   }
```

```
}
```

Cross References:

- Dynamic threads, see Section 3.1.7 on page 39.
- **OMP_DYNAMIC** environment variable, see Section 4.3 on page 49.

2.7.2 Data-Sharing Attribute Clauses

Several directives accept clauses that allow a user to control the sharing attributes of variables for the duration of the region. Sharing attribute clauses apply only to variables in the lexical extent of the directive on which the clause appears. Not all of the following clauses are allowed on all directives. The list of clauses that are valid on a particular directive are described with the directive.

If a variable is visible when a parallel or work-sharing construct is encountered, and the variable is not specified in a sharing attribute clause or **threadprivate** directive, then the variable is shared. Static variables declared within the dynamic extent of a parallel region are shared. Heap allocated memory (for example, using **malloc()** in C or C++ or the **new** operator in C++) is shared. (The pointer to this memory, however, can be either private or shared.) Variables with automatic storage duration declared within the dynamic extent of a parallel region are private.

Most of the clauses accept a *variable-list* argument, which is a comma-separated list of variables that are visible. If a variable referenced in a data-sharing attribute clause has a type derived from a template, and there are no other references to that variable in the program, the behavior is undefined.

All variables that appear within directive clauses must be visible. Clauses may be repeated as needed, but no variable may be specified in more than one clause, except that a variable can be specified in both a **firstprivate** and a **lastprivate** clause.

The following sections describe the data-sharing attribute clauses:

- **private**, Section 2.7.2.1 on page 25.
- **firstprivate**, Section 2.7.2.2 on page 26.
- **lastprivate**, Section 2.7.2.3 on page 27.
- **shared**, Section 2.7.2.4 on page 27.
- **default**, Section 2.7.2.5 on page 28.
- **reduction**, Section 2.7.2.6 on page 28.
- **copyin**, Section 2.7.2.7 on page 31.
- **copyprivate**, Section 2.7.2.8 on page 32.

2.7.2.1 private

The **private** clause declares the variables in variable-list to be private to each thread in a team. The syntax of the **private** clause is as follows:

```
private(variable-list)
```

The behavior of a variable specified in a **private** clause is as follows. A new object with automatic storage duration is allocated for the construct. The size and alignment of the new object are determined by the type of the variable. This allocation occurs once for each thread in the team, and a default constructor is invoked for a class object if necessary; otherwise the initial value is indeterminate. The original object referenced by the variable has an indeterminate value upon entry to the construct, must not be modified within the dynamic extent of the construct, and has an indeterminate value upon exit from the construct.

In the lexical extent of the directive construct, the variable references the new private object allocated by the thread.

The restrictions to the **private** clause are as follows:

- A variable with a class type that is specified in a **private** clause must have an accessible, unambiguous default constructor.
- A variable specified in a **private** clause must not have a **const**-qualified type unless it has a class type with a **mutable** member.
- A variable specified in a **private** clause must not have an incomplete type or a reference type.
- Variables that appear in the **reduction** clause of a **parallel** directive cannot be specified in a **private** clause on a work-sharing directive that binds to the parallel construct.

2.7.2.2 firstprivate

The **firstprivate** clause provides a superset of the functionality provided by the **private** clause. The syntax of the **firstprivate** clause is as follows:

```
firstprivate(variable-list)
```

Variables specified in *variable-list* have **private** clause semantics, as described in Section 2.7.2.1 on page 25. The initialization or construction happens as if it were done once per thread, prior to the thread's execution of the construct. For a **firstprivate** clause on a parallel construct, the initial value of the new private object is the value of the original object that exists immediately prior to the parallel construct for the thread that encounters it. For a **firstprivate** clause on a work-sharing construct, the initial value of the new private object for each thread that executes the work-sharing construct is the value of the original object that exists prior to the point in time that the same thread encounters the work-sharing construct. In addition, for C++ objects, the new private object for each thread is copy constructed from the original object.

The restrictions to the **firstprivate** clause are as follows:

- A variable specified in a **firstprivate** clause must not have an incomplete type or a reference type.
- A variable with a class type that is specified as **firstprivate** must have an accessible, unambiguous copy constructor.
- Variables that are private within a parallel region or that appear in the **reduction** clause of a **parallel** directive cannot be specified in a **firstprivate** clause on a work-sharing directive that binds to the parallel construct.

2.7.2.3 lastprivate

The **lastprivate** clause provides a superset of the functionality provided by the **private** clause. The syntax of the **lastprivate** clause is as follows:

```
lastprivate(variable-list)
```

Variables specified in the *variable-list* have **private** clause semantics. When a **lastprivate** clause appears on the directive that identifies a work-sharing construct, the value of each **lastprivate** variable from the sequentially last iteration of the associated loop, or the lexically last section directive, is assigned to the variable's original object. Variables that are not assigned a value by the last iteration of the **for** or **parallel for**, or by the lexically last section of the **sections** or **parallel sections** directive, have indeterminate values after the construct. Unassigned subobjects also have an indeterminate value after the construct.

The restrictions to the **lastprivate** clause are as follows:

- All restrictions for **private** apply.
- A variable with a class type that is specified as **lastprivate** must have an accessible, unambiguous copy assignment operator.
- Variables that are private within a parallel region or that appear in the **reduction** clause of a **parallel** directive cannot be specified in a **lastprivate** clause on a work-sharing directive that binds to the parallel construct.

2.7.2.4 shared

This clause shares variables that appear in the *variable-list* among all the threads in a team. All threads within a team access the same storage area for **shared** variables.

The syntax of the **shared** clause is as follows:

```
shared(variable-list)
```

2.7.2.5 default

The **default** clause allows the user to affect the data-sharing attributes of variables. The syntax of the **default** clause is as follows:

```
default(shared | none)
```

Specifying **default(shared)** is equivalent to explicitly listing each currently visible variable in a **shared** clause, unless it is **threadprivate** or **const**-qualified. In the absence of an explicit **default** clause, the default behavior is the same as if **default(shared)** were specified.

Specifying **default(none)** requires that at least one of the following must be true for every reference to a variable in the lexical extent of the parallel construct:

- The variable is explicitly listed in a data-sharing attribute clause of a construct that contains the reference.
- The variable is declared within the parallel construct.
- The variable is **threadprivate**.
- The variable has a **const**-qualified type.
- The variable is the loop control variable for a **for** loop that immediately follows a **for** or **parallel for** directive, and the variable reference appears inside the loop.

Specifying a variable on a **firstprivate**, **lastprivate**, or **reduction** clause of an enclosed directive causes an implicit reference to the variable in the enclosing context. Such implicit references are also subject to the requirements listed above.

Only a single **default** clause may be specified on a **parallel** directive.

A variable's default data-sharing attribute can be overridden by using the **private**, **firstprivate**, **lastprivate**, **reduction**, and **shared** clauses, as demonstrated by the following example:

```
#pragma  omp  parallel  for  default(shared)  firstprivate(i)\
   private(x)  private(r)  lastprivate(i)
```

2.7.2.6 reduction

This clause performs a reduction on the scalar variables that appear in *variable-list*, with the operator op. The syntax of the **reduction** clause is as follows:

```
reduction(
op
:
variable-list
)
```

A reduction is typically specified for a statement with one of the following forms:

```
        x     =  x     op     expr
x     binop=  expr
x     =  expr     op     x            (except for subtraction)
x++
++x
x--
--x
```

where:

x

One of the reduction variables specified in the *list*.

variable-list

A comma-separated list of scalar reduction variables.

expr

An expression with scalar type that does not reference *x*.

op

Not an overloaded operator but one of +, *, -, &, ^, |, &&, or ||.

binop

Not an overloaded operator but one of +, *, -, &, ^, or |.

The following is an example of the **reduction** clause:

```
#pragma omp parallel for reduction(+: a, y) reduction(||: am)
for (i=0; i<n; i++) {
   a += b[i];
   y = sum(y, c[i]);
   am = am || b[i] == c[i];
}
```

As shown in the example, an operator may be hidden inside a function call. The user should be careful that the operator specified in the **reduction** clause matches the reduction operation.

Although the right operand of the || operator has no side effects in this example, they are permitted, but should be used with care. In this context, a side effect that is guaranteed not to occur during sequential execution of the loop may occur during parallel execution. This difference can occur because the order of execution of the iterations is indeterminate.

The operator is used to determine the initial value of any private variables used by the compiler for the reduction and to determine the finalization operator. Specifying the operator explicitly allows the reduction statement to be outside the lexical extent of the construct. Any number of **reduction** clauses may be specified on the directive, but a variable may appear in at

most one **reduction** clause for that directive.

A private copy of each variable in *variable-list* is created, one for each thread, as if the **private** clause had been used. The private copy is initialized according to the operator (see the following table).

At the end of the region for which the **reduction** clause was specified, the original object is updated to reflect the result of combining its original value with the final value of each of the private copies using the operator specified. The reduction operators are all associative (except for subtraction), and the compiler may freely reassociate the computation of the final value. (The partial results of a subtraction reduction are added to form the final value.)

The value of the original object becomes indeterminate when the first thread reaches the containing clause and remains so until the reduction computation is complete. Normally, the computation will be complete at the end of the construct; however, if the **reduction** clause is used on a construct to which **nowait** is also applied, the value of the original object remains indeterminate until a barrier synchronization has been performed to ensure that all threads have completed the **reduction** clause.

The following table lists the operators that are valid and their canonical initialization values. The actual initialization value will be consistent with the data type of the reduction variable.

Operator	**Initialization**
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

The restrictions to the **reduction** clause are as follows:

- The type of the variables in the **reduction** clause must be valid for the reduction operator except that pointer types and reference types are never permitted.
- A variable that is specified in the **reduction** clause must not be **const**-qualified.
- Variables that are private within a parallel region or that appear in the **reduction** clause of a **parallel** directive cannot be specified in a **reduction** clause on a work-sharing directive that binds to the parallel construct.

```
#pragma omp parallel private(y)
{ /* ERROR - private variable y cannot be specified
             in a reduction clause */
   #pragma omp for reduction(+: y)
   for (i=0; i<n; i++)
      y += b[i];
}

/* ERROR - variable x cannot be specified in both
           a shared and a reduction clause */
#pragma omp parallel for shared(x) reduction(+: x)
```

2.7.2.7 copyin

The **copyin** clause provides a mechanism to assign the same value to **threadprivate** variables for each thread in the team executing the parallel region. For each variable specified in a **copyin** clause, the value of the variable in the master thread of the team is copied, as if by assignment, to the thread-private copies at the beginning of the parallel region. The syntax of the **copyin** clause is as follows:

```
copyin(
variable-list
)
```

The restrictions to the **copyin** clause are as follows:

- A variable that is specified in the **copyin** clause must have an accessible, unambiguous copy assignment operator.
- A variable that is specified in the **copyin** clause must be a **threadprivate** variable.

2.7.2.8 copyprivate

The **copyprivate** clause provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members. It is an alternative to using a shared variable for the value when providing such a shared variable would be difficult (for example, in a recursion requiring a different variable at each level). The **copyprivate** clause can only appear on the **single** directive.

The syntax of the **copyprivate** clause is as follows:

```
copyprivate(
variable-list
)
```

The effect of the **copyprivate** clause on the variables in its variable-list occurs after the execution of the structured block associated with the **single** construct, and before any of the threads in the team have left the barrier at the end of the construct. Then, in all other threads in the team, for each variable in the *variable-list*, that variable becomes defined (as if by assignment) with the value of the corresponding variable in the thread that executed the construct's structured block.

Restrictions to the **copyprivate** clause are as follows:

- A variable that is specified in the **copyprivate** clause must not appear in a **private** or **firstprivate** clause for the same **single** directive.
- If a **single** directive with a **copyprivate** clause is encountered in the dynamic extent of a parallel region, all variables specified in the **copyprivate** clause must be private in the enclosing context.
- A variable that is specified in the **copyprivate** clause must have an accessible unambiguous copy assignment operator.

2.8 Directive Binding

Dynamic binding of directives must adhere to the following rules:

- The **for**, **sections**, **single**, **master**, and **barrier** directives bind to the dynamically enclosing **parallel**, if one exists, regardless of the value of any **if** clause that may be present on that directive. If no parallel region is currently being executed, the directives are executed by a team composed of only the master thread.
- The **ordered** directive binds to the dynamically enclosing **for**.
- The **atomic** directive enforces exclusive access with respect to **atomic** directives in all threads, not just the current team.
- The **critical** directive enforces exclusive access with respect to **critical** directives in all threads, not just the current team.
- A directive can never bind to any directive outside the closest dynamically enclosing **parallel**.

2.9 Directive Nesting

Dynamic nesting of directives must adhere to the following rules:

- A **parallel** directive dynamically inside another **parallel** logically establishes a new team, which is composed of only the current thread, unless nested parallelism is enabled.
- **for**, **sections**, and **single** directives that bind to the same **parallel** are not allowed to be nested inside each other.
- **critical** directives with the same name are not allowed to be nested inside each other. Note this restriction is not sufficient to prevent deadlock.
- **for**, **sections**, and **single** directives are not permitted in the dynamic extent of **critical**, **ordered**, and **master** regions if the directives bind to the same **parallel** as the regions.
- **barrier** directives are not permitted in the dynamic extent of **for**, **ordered**, **sections**, **single**, **master**, and **critical** regions if the directives bind to the same **parallel** as the regions.
- **master** directives are not permitted in the dynamic extent of **for**, **sections**, and **single** directives if the **master** directives bind to the same **parallel** as the work-sharing directives.
- **ordered** directives are not allowed in the dynamic extent of **critical** regions if the directives bind to the same **parallel** as the regions.
- Any directive that is permitted when executed dynamically inside a parallel region is also permitted when executed outside a parallel region. When executed dynamically outside a user-specified parallel region, the directive is executed by a team composed of only the master thread.

3. Run-time Library Functions

This section describes the OpenMP C and C++ run-time library functions. The header **<omp.h>** declares two types, several functions that can be used to control and query the parallel execution environment, and lock functions that can be used to synchronize access to data.

The type **omp_lock_t** is an object type capable of representing that a lock is available, or that a thread owns a lock. These locks are referred to as *simple locks*.

The type **omp_nest_lock_t** is an object type capable of representing either that a lock is available, or both the identity of the thread that owns the lock and a *nesting count* (described below). These locks are referred to as *nestable locks*.

The library functions are external functions with "C" linkage.

The descriptions in this chapter are divided into the following topics:

- Execution environment functions (see Section 3.1 on page 35).
- Lock functions (see Section 3.2 on page 41).

3.1 Execution Environment Functions

The functions described in this section affect and monitor threads, processors, and the parallel environment:

- the **omp_set_num_threads** function.
- the **omp_get_num_threads** function.
- the **omp_get_max_threads** function.
- the **omp_get_thread_num** function.
- the **omp_get_num_procs** function.
- the **omp_in_parallel** function.
- the **omp_set_dynamic** function.
- the **omp_get_dynamic** function.
- the **omp_set_nested** function.
- the **omp_get_nested** function.

3.1.1 omp_set_num_threads Function

The **omp_set_num_threads** function sets the default number of threads to use for subsequent parallel regions that do not specify a **num_threads** clause. The format is as follows:

```
#include <omp.h>
void omp_set_num_threads(int num_threads);
```

The value of the parameter *num_threads* must be a positive integer. Its effect depends upon whether dynamic adjustment of the number of threads is enabled. For a comprehensive set of rules about the interaction between the **omp_set_num_threads** function and dynamic adjustment of threads, see Section 2.3 on page 8.

This function has the effects described above when called from a portion of the program where the **omp_in_parallel** function returns zero. If it is called from a portion of the program where the **omp_in_parallel** function returns a nonzero value, the behavior of this function is undefined.

This call has precedence over the **OMP_NUM_THREADS** environment variable. The default value for the number of threads, which may be established by calling **omp_set_num_threads** or by setting the **OMP_NUM_THREADS** environment variable, can be explicitly overridden on a single **parallel** directive by specifying the **num_threads** clause.

Cross References:

- **omp_set_dynamic** function, see Section 3.1.7 on page 39.
- **omp_get_dynamic** function, see Section 3.1.8 on page 40.
- **OMP_NUM_THREADS** environment variable, see Section 4.2 on page 48, and Section 2.3 on page 8.
- **num_threads** clause, see Section 2.3 on page 8

3.1.2 omp_get_num_threads Function

The **omp_get_num_threads** function returns the number of threads currently in the team executing the parallel region from which it is called. The format is as follows:

```
#include <omp.h>
int omp_get_num_threads(void);
```

The **num_threads** clause, the **omp_set_num_threads** function, and the **OMP_NUM_THREADS** environment variable control the number of threads in a team.

If the number of threads has not been explicitly set by the user, the default is implementation-defined. This function binds to the closest enclosing **parallel** directive. If called from a serial portion of a program, or from a nested parallel region that is serialized, this function returns 1.

Cross References:

- **OMP_NUM_THREADS** environment variable, see Section 4.2 on page 48.
- **num_threads** clause, see Section 2.3 on page 8.
- **parallel** construct, see Section 2.3 on page 8.

3.1.3 omp_get_max_threads Function

The **omp_get_max_threads** function returns an integer that is guaranteed to be at least as large as the number of threads that would be used to form a team if a parallel region without a **num_threads** clause were to be encountered at that point in the code. The format is as follows:

```
#include <omp.h>
int omp_get_max_threads(void);
```

The following expresses a lower bound on the value of **omp_get_max_threads**:

```
threads-used-for-next-team
 <= omp_get_max_threads
```

Note that if a subsequent parallel region uses the **num_threads** clause to request a specific number of threads, the guarantee on the lower bound of the result of **omp_get_max_threads** no long holds.

The **omp_get_max_threads** function's return value can be used to dynamically allocate sufficient storage for all threads in the team formed at the subsequent parallel region.

Cross References:

- **omp_get_num_threads** function, see Section 3.1.2 on page 37.
- **omp_set_num_threads** function, see Section 3.1.1 on page 36.
- **omp_set_dynamic** function, see Section 3.1.7 on page 39.
- **num_threads** clause, see Section 2.3 on page 8.

3.1.4 omp_get_thread_num Function

The **omp_get_thread_num** function returns the thread number, within its team, of the thread executing the function. The thread number lies between 0 and **omp_get_num_threads()**–1, inclusive. The master thread of the team is thread 0.

The format is as follows:

```
#include <omp.h>
int omp_get_thread_num(void);
```

If called from a serial region, **omp_get_thread_num** returns 0. If called from within a nested parallel region that is serialized, this function returns 0.

Cross References:

- **omp_get_num_threads** function, see Section 3.1.2 on page 37.

3.1.5 omp_get_num_procs Function

The **omp_get_num_procs** function returns the number of processors that are available to the program at the time the function is called. The format is as follows:

```
#include <omp.h>
int omp_get_num_procs(void);
```

3.1.6 omp_in_parallel Function

The **omp_in_parallel** function returns a nonzero value if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, it returns 0. The format is as follows:

```
#include <omp.h>
int omp_in_parallel(void);
```

This function returns a nonzero value when called from within a region executing in parallel, including nested regions that are serialized.

3.1.7 omp_set_dynamic Function

The **omp_set_dynamic** function enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. The format is as follows:

```
#include <omp.h>
void omp_set_dynamic(int dynamic_threads);
```

If *dynamic_threads* evaluates to a nonzero value, the number of threads that are used for executing subsequent parallel regions may be adjusted automatically by the run-time environment to best utilize system resources. As a consequence, the number of threads specified by the user is the maximum thread count. The number of threads in the team executing a parallel region remains fixed for the duration of that parallel region and is reported by the **omp_get_num_threads** function.

If *dynamic_threads* evaluates to 0, dynamic adjustment is disabled.

This function has the effects described above when called from a portion of the program where the **omp_in_parallel** function returns zero. If it is called from a portion of the program where the **omp_in_parallel** function returns a nonzero value, the behavior of this function is undefined.

A call to **omp_set_dynamic** has precedence over the **OMP_DYNAMIC** environment variable.

The default for the dynamic adjustment of threads is implementation-defined. As a result, user codes that depend on a specific number of threads for correct execution should explicitly disable dynamic threads. Implementations are not required to provide the ability to dynamically adjust the number of threads, but they are required to provide the interface in order to support portability across all platforms.

Cross References:

- **omp_get_num_threads** function, see Section 3.1.2 on page 37.
- **OMP_DYNAMIC** environment variable, see Section 4.3 on page 49.
- **omp_in_parallel** function, see Section 3.1.6 on page 38.

3.1.8 omp_get_dynamic Function

The **omp_get_dynamic** function returns a nonzero value if dynamic adjustment of threads is enabled, and returns 0 otherwise. The format is as follows:

```
#include <omp.h>
int omp_get_dynamic(void);
```

If the implementation does not implement dynamic adjustment of the number of threads, this function always returns 0.

Cross References:

- For a description of dynamic thread adjustment, see Section 3.1.7 on page 39.

3.1.9 omp_set_nested Function

The **omp_set_nested** function enables or disables nested parallelism. The format is as follows:

```
#include <omp.h>
void omp_set_nested(int nested);
```

If *nested* evaluates to 0, nested parallelism is disabled, which is the default, and nested parallel regions are serialized and executed by the current thread. If *nested* evaluates to a nonzero value, nested parallelism is enabled, and parallel regions that are nested may deploy additional threads to form nested teams.

This function has the effects described above when called from a portion of the program where the **omp_in_parallel** function returns zero. If it is called from a portion of the program where the **omp_in_parallel** function returns a nonzero value, the behavior of this function is undefined.

This call has precedence over the **OMP_NESTED** environment variable.

When nested parallelism is enabled, the number of threads used to execute nested parallel regions is implementation-defined. As a result, OpenMP-compliant implementations are allowed to serialize nested parallel regions even when nested parallelism is enabled.

Cross References:

- **OMP_NESTED** environment variable, see Section 4.4 on page 49.
- **omp_in_parallel** function, see Section 3.1.6 on page 38.

3.1.10 omp_get_nested Function

The **omp_get_nested** function returns a nonzero value if nested parallelism is enabled and 0 if it is disabled. For more information on nested parallelism, see Section 3.1.9 on page 40. The format is as follows:

```
#include <omp.h>
int omp_get_nested(void);
```

If an implementation does not implement nested parallelism, this function always returns 0.

3.2 Lock Functions

The functions described in this section manipulate locks used for synchronization.

For the following functions, the lock variable must have type **omp_lock_t**. This variable must only be accessed through these functions. All lock functions require an argument that has a pointer to **omp_lock_t** type.

- The **omp_init_lock** function initializes a simple lock.
- The **omp_destroy_lock** function removes a simple lock.
- The **omp_set_lock** function waits until a simple lock is available.
- The **omp_unset_lock** function releases a simple lock.
- The **omp_test_lock** function tests a simple lock.

For the following functions, the lock variable must have type **omp_nest_lock_t**. This variable must only be accessed through these functions. All nestable lock functions require an argument that has a pointer to **omp_nest_lock_t** type.

- The **omp_init_nest_lock** function initializes a nestable lock.
- The **omp_destroy_nest_lock** function removes a nestable lock.
- The **omp_set_nest_lock** function waits until a nestable lock is available.
- The **omp_unset_nest_lock** function releases a nestable lock.
- The **omp_test_nest_lock** function tests a nestable lock.

The OpenMP lock functions access the lock variable in such a way that they always read and update the most current value of the lock variable. Therefore, it is not necessary for an OpenMP program to include explicit **flush** directives to ensure that the lock variable's value is consistent among different threads. (There may be a need for **flush** directives to make the values of other variables consistent.)

3.2.1 omp_init_lock and omp_init_nest_lock Functions

These functions provide the only means of initializing a lock. Each function initializes the lock associated with the parameter *lock* for use in subsequent calls. The format is as follows:

```
#include <omp.h>
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

The initial state is unlocked (that is, no thread owns the lock). For a nestable lock, the initial nesting count is zero. It is noncompliant to call either of these routines with a lock variable that has already been initialized.

3.2.2 omp_destroy_lock and omp_destroy_nest_lock Functions

These functions ensure that the pointed to lock variable *lock* is uninitialized. The format is as follows:

```
#include <omp.h>
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

It is noncompliant to call either of these routines with a lock variable that is uninitialized or unlocked.

3.2.3 omp_set_lock and omp_set_nest_lock Functions

Each of these functions blocks the thread executing the function until the specified lock is available and then sets the lock. A simple lock is available if it is unlocked. A nestable lock is available if it is unlocked or if it is already owned by the thread executing the function. The format is as follows:

```
#include <omp.h>
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

For a simple lock, the argument to the **omp_set_lock** function must point to an initialized lock variable. Ownership of the lock is granted to the thread executing the function.

For a nestable lock, the argument to the **omp_set_nest_lock** function must point to an initialized lock variable. The nesting count is incremented, and the thread is granted, or retains, ownership of the lock.

3.2.4 omp_unset_lock and omp_unset_nest_lock Functions

These functions provide the means of releasing ownership of a lock. The format is as follows:

```
#include <omp.h>
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

The argument to each of these functions must point to an initialized lock variable owned by the thread executing the function. The behavior is undefined if the thread does not own that lock.

For a simple lock, the **omp_unset_lock** function releases the thread executing the function from ownership of the lock.

For a nestable lock, the **omp_unset_nest_lock** function decrements the nesting count, and releases the thread executing the function from ownership of the lock if the resulting count is zero.

3.2.5 omp_test_lock and omp_test_nest_lock Functions

These functions attempt to set a lock but do not block execution of the thread. The format is as follows:

```
#include <omp.h>
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

The argument must point to an initialized lock variable. These functions attempt to set a lock in the same manner as **omp_set_lock** and **omp_set_nest_lock**, except that they do not block execution of the thread.

For a simple lock, the **omp_test_lock** function returns a nonzero value if the lock is successfully set; otherwise, it returns zero.

For a nestable lock, the **omp_test_nest_lock** function returns the new nesting count if the lock is successfully set; otherwise, it returns zero.

3.3 Timing Routines

The functions described in this section support a portable wall-clock timer:

- The **omp_get_wtime** function returns elapsed wall-clock time.
- The **omp_get_wtick** function returns seconds between successive clock ticks.

3.3.1 omp_get_wtime Function

The **omp_get_wtime** function returns a double-precision floating point value equal to the elapsed wall clock time in seconds since some "time in the past". The actual "time in the past" is arbitrary, but it is guaranteed not to change during the execution of the application program. The format is as follows:

```
#include <omp.h>
double omp_get_wtime(void);
```

It is anticipated that the function will be used to measure elapsed times as shown in the following example:

```
double start;
double end;
start = omp_get_wtime();
... work to be timed ...
end = omp_get_wtime();
printf_s("Work took %f sec. time.\n", end-start);
```

The times returned are "per-thread times" by which is meant they are not required to be globally consistent across all the threads participating in an application.

3.3.2 omp_get_wtick Function

The **omp_get_wtick** function returns a double-precision floating point value equal to the number of seconds between successive clock ticks. The format is as follows:

```
#include <omp.h>
double omp_get_wtick(void);
```

4. Environment Variables

This chapter describes the OpenMP C and C++ API environment variables (or equivalent platform-specific mechanisms) that control the execution of parallel code. The names of environment variables must be uppercase. The values assigned to them are case insensitive and may have leading and trailing white space. Modifications to the values after the program has started are ignored.

The environment variables are as follows:

- **OMP_SCHEDULE** sets the run-time schedule type and chunk size.
- **OMP_NUM_THREADS** sets the number of threads to use during execution.
- **OMP_DYNAMIC** enables or disables dynamic adjustment of the number of threads.
- **OMP_NESTED** enables or disables nested parallelism.

The examples in this chapter only demonstrate how these variables might be set in Unix C shell (csh) environments. In Korn shell and DOS environments the actions are similar, as follows:

csh:

```
setenv OMP_SCHEDULE "dynamic"
```

ksh:

```
export OMP_SCHEDULE="dynamic"
```

DOS:

```
set OMP_SCHEDULE="dynamic"
```

4.1 OMP_SCHEDULE

OMP_SCHEDULE applies only to **for** and **parallel for** directives that have the schedule type **runtime**. The schedule type and chunk size for all such loops can be set at run time by setting this environment variable to any of the recognized schedule types and to an optional *chunk_size*.

For **for** and **parallel for** directives that have a schedule type other than **runtime**, **OMP_SCHEDULE** is ignored. The default value for this environment variable is implementation-defined. If the optional *chunk_size* is set, the value must be positive. If *chunk_size* is not set, a value of 1 is assumed, except in the case of a **static** schedule. For a **static** schedule, the default chunk size is set to the loop iteration space divided by the number of threads applied to the loop.

Example:

```
setenv OMP_SCHEDULE "guided,4"
setenv OMP_SCHEDULE "dynamic"
```

Cross References:

- **for** directive, see Section 2.4.1 on page 11.
- **parallel for** directive, see Section 2.5.1 on page 16.

4.2 OMP_NUM_THREADS

The **OMP_NUM_THREADS** environment variable sets the default number of threads to use during execution, unless that number is explicitly changed by calling the **omp_set_num_threads** library routine or by an explicit **num_threads** clause on a **parallel** directive.

The value of the **OMP_NUM_THREADS** environment variable must be a positive integer. Its effect depends upon whether dynamic adjustment of the number of threads is enabled. For a comprehensive set of rules about the interaction between the **OMP_NUM_THREADS** environment variable and dynamic adjustment of threads, see Section 2.3 on page 8.

If no value is specified for the **OMP_NUM_THREADS** environment variable, or if the value specified is not a positive integer, or if the value is greater than the maximum number of threads the system can support, the number of threads to use is implementation-defined.

Example:

```
setenv OMP_NUM_THREADS 16
```

Cross References:

- **num_threads** clause, see Section 2.3 on page 8.
- **omp_set_num_threads** function, see Section 3.1.1 on page 36.
- **omp_set_dynamic** function, see Section 3.1.7 on page 39.

4.3 OMP_DYNAMIC

The **OMP_DYNAMIC** environment variable enables or disables dynamic adjustment of the number of threads available for execution of parallel regions unless dynamic adjustment is explicitly enabled or disabled by calling the **omp_set_dynamic** library routine. Its value must be **TRUE** or **FALSE**.

If set to **TRUE**, the number of threads that are used for executing parallel regions may be adjusted by the runtime environment to best utilize system resources. If set to **FALSE**, dynamic adjustment is disabled. The default condition is implementation-defined.

Example:

```
setenv OMP_DYNAMIC TRUE
```

Cross References:

- For more information on parallel regions, see Section 2.3 on page 8.
- **omp_set_dynamic** function, see Section 3.1.7 on page 39.

4.4 OMP_NESTED

The **OMP_NESTED** environment variable enables or disables nested parallelism unless nested parallelism is enabled or disabled by calling the **omp_set_nested** library routine. If set to **TRUE**, nested parallelism is enabled; if it is set to **FALSE**, nested parallelism is disabled. The default value is **FALSE**.

Example:

```
setenv OMP_NESTED TRUE
```

Cross Reference:

- **omp_set_nested** function, see Section 3.1.9 on page 40.

A. Examples

The following are examples of the constructs defined in this document. Note that a statement following a directive is compound only when necessary, and a non-compound statement is indented with respect to a directive preceding it.

A.1 Executing a Simple Loop in Parallel

The following example demonstrates how to parallelize a simple loop using the **parallel for** directive (Section 2.5.1 on page 16). The loop iteration variable is private by default, so it is not necessary to specify it explicitly in a private clause.

```
#pragma omp parallel for
    for (i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
```

A.2 Specifying Conditional Compilation

The following examples illustrate the use of conditional compilation using the OpenMP macro **_OPENMP** (Section 2.2 on page 8). With OpenMP compilation, the **_OPENMP** macro becomes defined.

```
# ifdef _OPENMP
    printf_s("Compiled by an OpenMP-compliant implementation.\n");
# endif
```

The defined preprocessor operator allows more than one macro to be tested in a single directive.

```
# if defined(_OPENMP) && defined(VERBOSE)
    printf_s("Compiled by an OpenMP-compliant implementation.\n");
# endif
```

A.3 Using Parallel Regions

The **parallel** directive (Section 2.3 on page 8) can be used in coarse-grain parallel programs. In the following example, each thread in the parallel region decides what part of the global array *x* to work on, based on the thread number:

```
#pragma omp parallel shared(x, npoints) private(iam, np, ipoints)
{
    iam = omp_get_thread_num();
    np =  omp_get_num_threads();
    ipoints = npoints / np;
    subdomain(x, iam, ipoints);
}
```

A.4 Using the nowait Clause

If there are multiple independent loops within a parallel region, you can use the **nowait** clause (Section 2.4.1 on page 11) to avoid the implied barrier at the end of the **for** directive, as follows:

```
#pragma omp parallel
{
    #pragma omp for nowait
        for (i=1; i<n; i++)
             b[i] = (a[i] + a[i-1]) / 2.0;
    #pragma omp for nowait
        for (i=0; i<m; i++)
            y[i] = sqrt(z[i]);
}
```

A.5 Using the critical Directive

The following example includes several **critical** directives (Section 2.6.2 on page 18). The example illustrates a queuing model in which a task is dequeued and worked on. To guard against multiple threads dequeuing the same task, the dequeuing operation must be in a **critical** section. Because the two queues in this example are independent, they are protected by **critical** directives with different names, *xaxis* and *yaxis*.

```
#pragma omp parallel shared(x, y) private(x_next, y_next)
{
    #pragma omp critical ( xaxis )
        x_next = dequeue(x);
    work(x_next);
    #pragma omp critical ( yaxis )
        y_next = dequeue(y);
    work(y_next);
}
```

A.6 Using the lastprivate Clause

Correct execution sometimes depends on the value that the last iteration of a loop assigns to a variable. Such programs must list all such variables as arguments to a **lastprivate** clause (Section 2.7.2.3 on page 27) so that the values of the variables are the same as when the loop is executed sequentially.

```
#pragma omp parallel
{
   #pragma omp for lastprivate(i)
      for (i=0; i<n-1; i++)
         a[i] = b[i] + b[i+1];
}
a[i]=b[i];
```

In the preceding example, the value of *i* at the end of the parallel region will equal *n–1*, as in the sequential case.

A.7 Using the reduction Clause

The following example demonstrates the reduction clause (Section 2.7.2.6 on page 28):

```
#pragma omp parallel for private(i) shared(x, y, n) \
                         reduction(+: a, b)
    for (i=0; i<n; i++) {
        a = a + x[i];
        b = b + y[i];
    }
```

A.8 Specifying Parallel Sections

In the following example, (for Section 2.4.2 on page 14) functions *xaxis*, *yaxis*, and *zaxis* can be executed concurrently. The first **section** directive is optional. Note that all **section** directives need to appear in the lexical extent of the **parallel sections** construct.

```
#pragma omp parallel sections
{
    #pragma omp section
        xaxis();
    #pragma omp section
        yaxis();
    #pragma omp section
        zaxis();
}
```

A.9 Using single Directives

The following example demonstrates the **single** directive (Section 2.4.3 on page 15). In the example, only one thread (usually the first thread that encounters the **single** directive) prints the progress message. The user must not make any assumptions as to which thread will execute the **single** section. All other threads will skip the **single** section and stop at the barrier at the end of the **single** construct. If other threads can proceed without waiting for the thread executing the **single** section, a **nowait** clause can be specified on the **single** directive.

```
#pragma omp parallel
{
    #pragma omp single
        printf_s("Beginning work1.\n");
    work1();
    #pragma omp single
        printf_s("Finishing work1.\n");
    #pragma omp single nowait
        printf_s("Finished work1 and beginning work2.\n");
    work2();
}
```

A.10 Specifying Sequential Ordering

Ordered sections (Section 2.6.6 on page 22) are useful for sequentially ordering the output from work that is done in parallel. The following program prints out the indexes in sequential order:

```
#pragma omp for ordered schedule(dynamic)
    for (i=lb; i<ub; i+=st)
        work(i);
void work(int k)
{
    #pragma omp ordered
        printf_s(" %d", k);
}
```

A.11 Specifying a Fixed Number of Threads

Some programs rely on a fixed, prespecified number of threads to execute correctly. Because the default setting for the dynamic adjustment of the number of threads is implementation-defined, such programs can choose to turn off the dynamic threads capability and set the number of threads explicitly to ensure portability. The following example shows how to do this using **omp_set_dynamic** (Section 3.1.7 on page 39), and **omp_set_num_threads** (Section 3.1.1 on page 36):

```
omp_set_dynamic(0);
omp_set_num_threads(16);
#pragma omp parallel shared(x, npoints) private(iam, ipoints)
{
    if (omp_get_num_threads() != 16)
      abort();
    iam = omp_get_thread_num();
    ipoints = npoints/16;
    do_by_16(x, iam, ipoints);
}
```

In this example, the program executes correctly only if it is executed by 16 threads. If the implementation is not capable of supporting 16 threads, the behavior of this example is implementation-defined.

Note that the number of threads executing a parallel region remains constant during a parallel region, regardless of the dynamic threads setting. The dynamic threads mechanism determines the number of threads to use at the start of the parallel region and keeps it constant for the duration of the region.

A.12 Using the atomic Directive

The following example avoids race conditions (simultaneous updates of an element of *x* by multiple threads) by using the **atomic** directive (Section 2.6.4 on page 19):

```
#pragma omp parallel for shared(x, y, index, n)
    for (i=0; i<n; i++)
    {
        #pragma omp atomic
            x[index[i]] += work1(i);
        y[i] += work2(i);
    }
```

The advantage of using the **atomic** directive in this example is that it allows updates of two different elements of x to occur in parallel. If a **critical** directive (Section 2.6.2 on page 18) were used instead, then all updates to elements of *x* would be executed serially (though not in any guaranteed order).

Note that the **atomic** directive applies only to the C or C++ statement immediately following it. As a result, elements of *y* are not updated atomically in this example.

A.13 Using the flush Directive with a List

The following example uses the **flush** directive for point-to-point synchronization of specific objects between pairs of threads:

```
int   sync[NUMBER_OF_THREADS];
float work[NUMBER_OF_THREADS];
#pragma omp parallel private(iam,neighbor) shared(work,sync)
{
    iam = omp_get_thread_num();
    sync[iam] = 0;
    #pragma omp barrier

    // Do computation into my portion of work array
    work[iam] = ...;

    //  Announce that I am done with my work
    // The first flush ensures that my work is
    // made visible before sync.
    // The second flush ensures that sync is made visible.
    #pragma omp flush(work)
    sync[iam] = 1;
    #pragma omp flush(sync)

    // Wait for neighbor
    neighbor = (iam>0 ? iam : omp_get_num_threads()) - 1;
    while (sync[neighbor]==0)
    {
        #pragma omp flush(sync)
    }

    // Read neighbor's values of work array
    ... = work[neighbor];
}
```

A.14 Using the flush Directive without a List

The following example (for Section 2.6.5 on page 20) distinguishes the shared objects affected by a **flush** directive with no list from the shared objects that are not affected:

Example

```
// omp_flush_without_list.c
#include <omp.h>

int x, *p = &x;

void f1(int *q)
{
    *q = 1;
    #pragma omp flush
    // x, p, and *q are flushed
    //   because they are shared and accessible
    // q is not flushed because it is not shared.
}

void f2(int *q)
{
    #pragma omp barrier
    *q = 2;

    #pragma omp barrier
    // a barrier implies a flush
    // x, p, and *q are flushed
    //   because they are shared and accessible
    // q is not flushed because it is not shared.
}

int g(int n)
{
    int i = 1, j, sum = 0;
    *p = 1;

    #pragma omp parallel reduction(+: sum) num_threads(10)
    {
        f1(&j);
        // i, n and sum were not flushed
        //   because they were not accessible in f1
        // j was flushed because it was accessible
        sum += j;
        f2(&j);
        // i, n, and sum were not flushed
        //   because they were not accessible in f2
        // j was flushed because it was accessible
        sum += i + j + *p + n;
    }
    return sum;
}

int main()
{
}
```

A.15 Determining the Number of Threads Used

Consider the following incorrect example (for Section 3.1.2 on page 37):

```
np = omp_get_num_threads(); // misplaced
#pragma omp parallel for schedule(static)
    for (i=0; i<np; i++)
        work(i);
```

The **omp_get_num_threads()** call returns 1 in the serial section of the code, so *np* will always be equal to 1 in the preceding example. To determine the number of threads that will be deployed for the parallel region, the call should be inside the parallel region.

The following example shows how to rewrite this program without including a query for the number of threads:

```
#pragma omp parallel private(i)
{
    i = omp_get_thread_num();
    work(i);
}
```

A.16 Using Locks

In the following example, (for Section 3.2 on page 41) note that the argument to the lock functions should have type **omp_lock_t**, and that there is no need to flush it. The lock functions cause the threads to be idle while waiting for entry to the first critical section, but to do other work while waiting for entry to the second. The **omp_set_lock** function blocks, but the **omp_test_lock** function does not, allowing the work in skip() to be done.

Example

```
// omp_using_locks.c
// compile with: /openmp /c
#include <stdio.h>
#include <omp.h>

void work(int);
void skip(int);

int main() {
   omp_lock_t lck;
   int id;

   omp_init_lock(&lck);
   #pragma omp parallel shared(lck) private(id)
   {
      id = omp_get_thread_num();

      omp_set_lock(&lck);
      printf_s("My thread id is %d.\n", id);

      // only one thread at a time can execute this printf
      omp_unset_lock(&lck);

      while (! omp_test_lock(&lck)) {
         skip(id);   // we do not yet have the lock,
                     // so we must do something else
      }
      work(id);     // we now have the lock
                    // and can do the work
      omp_unset_lock(&lck);
   }
   omp_destroy_lock(&lck);
}
```

A.17 Using Nestable Locks

The following example (for Section 3.2 on page 41) demonstrates how a nestable lock can be used to synchronize updates both to a whole structure and to one of its members.

```
#include <omp.h>
typedef struct {int a,b; omp_nest_lock_t lck;} pair;

void incr_a(pair *p, int a)
{
    // Called only from incr_pair, no need to lock.
    p->a += a;
}

void incr_b(pair *p, int b)
{
    // Called both from incr_pair and elsewhere,
    // so need a nestable lock.

    omp_set_nest_lock(&p->lck);
    p->b += b;
    omp_unset_nest_lock(&p->lck);
}

void incr_pair(pair *p, int a, int b)
{
    omp_set_nest_lock(&p->lck);
    incr_a(p, a);
    incr_b(p, b);
    omp_unset_nest_lock(&p->lck);
}

void f(pair *p)
{
    extern int work1(), work2(), work3();
    #pragma omp parallel sections
    {
        #pragma omp section
            incr_pair(p, work1(), work2());
        #pragma omp section
            incr_b(p, work3());
    }
}
```

A.18 Nested for Directives

The following example of **for** directive nesting (Section 2.9 on page 33) is compliant because the inner and outer **for** directives bind to different parallel regions:

```
#pragma omp parallel default(shared)
{
    #pragma omp for
        for (i=0; i<n; i++)
        {
            #pragma omp parallel shared(i, n)
            {
                #pragma omp for
                    for (j=0; j<n; j++)
                        work(i, j);
            }
        }
}
```

A following variation of the preceding example is also compliant:

```
#pragma omp parallel default(shared)
{
    #pragma omp for
        for (i=0; i<n; i++)
            work1(i, n);
}

void work1(int i, int n)
{
    int j;
    #pragma omp parallel default(shared)
    {
        #pragma omp for
            for (j=0; j<n; j++)
                work2(i, j);
    }
    return;
}
```

A.19 Examples Showing Incorrect Nesting of Work-sharing Directives

The examples in this section illustrate the directive nesting rules. For more information on directive nesting, see Section 2.9 on page 33.

The following example is noncompliant because the inner and outer **for** directives are nested and bind to the same **parallel** directive:

```
void wrong1(int n)
{
  #pragma omp parallel default(shared)
  {
      int i, j;
      #pragma omp for
      for (i=0; i<n; i++) {
          #pragma omp for
              for (j=0; j<n; j++)
                 work(i, j);
     }
   }
}
```

The following dynamically nested version of the preceding example is also noncompliant:

```
void wrong2(int n)
{
  #pragma omp parallel default(shared)
  {
    int i;
    #pragma omp for
      for (i=0; i<n; i++)
        work1(i, n);
  }
}

void work1(int i, int n)
{
  int j;
  #pragma omp for
    for (j=0; j<n; j++)
      work2(i, j);
}
```

The following example is noncompliant because the **for** and **single** directives are nested, and they bind to the same parallel region:

```
void wrong3(int n)
{
  #pragma omp parallel default(shared)
  {
    int i;
    #pragma omp for
      for (i=0; i<n; i++) {
        #pragma omp single
          work(i);
      }
  }
}
```

The following example is noncompliant because a **barrier** directive inside a **for** can result in deadlock:

```
void wrong4(int n)
{
  #pragma omp parallel default(shared)
  {
    int i;
    #pragma omp for
      for (i=0; i<n; i++) {
        work1(i);
        #pragma omp barrier
        work2(i);
      }
  }
}
```

The following example is noncompliant because the **barrier** results in deadlock due to the fact that only one thread at a time can enter the critical section:

```
void wrong5()
{
  #pragma omp parallel
  {
    #pragma omp critical
    {
       work1();
       #pragma omp barrier
       work2();
    }
  }
}
```

The following example is noncompliant because the **barrier** results in deadlock due to the fact that only one thread executes the **single** section:

```
void wrong6()
{
  #pragma omp parallel
  {
    setup();
    #pragma omp single
    {
      work1();
      #pragma omp barrier
      work2();
    }
    finish();
  }
}
```

A.20 Binding of barrier Directives

The directive binding rules call for a **barrier** directive to bind to the closest enclosing **parallel** directive. For more information on directive binding, see Section 2.8 on page 32.

In the following example, the call from *main* to *sub2* is compliant because the **barrier** (in *sub3*) binds to the parallel region in *sub2*. The call from *main* to *sub1* is compliant because the **barrier** binds to the parallel region in subroutine *sub2*. The call from *main* to *sub3* is compliant because the **barrier** does not bind to any parallel region and is ignored. Also note that the **barrier** only synchronizes the team of threads in the enclosing parallel region and not all the threads created in *sub1*.

```
int main()
{
    sub1(2);
    sub2(2);
    sub3(2);
}

void sub1(int n)
{
    int i;
    #pragma omp parallel private(i) shared(n)
    {
        #pragma omp for
        for (i=0; i<n; i++)
            sub2(i);
    }
}

void sub2(int k)
{
     #pragma omp parallel shared(k)
     sub3(k);
}

void sub3(int n)
{
    work(n);
    #pragma omp barrier
    work(n);
}
```

A.21 Scoping Variables with the private Clause

The values of *i* and *j* in the following example are undefined on exit from the parallel region:

```
int i, j;
i = 1;
j = 2;
#pragma omp parallel private(i) firstprivate(j)
{
  i = 3;
  j = j + 2;
}
printf_s("%d %d\n", i, j);
```

For more information on the **private** clause, see Section 2.7.2.1 on page 25.

A.22 Using the default(none) Clause

The following example distinguishes the variables that are affected by the **default(none)** clause from those that are not:

```
// openmp_using_clausedefault.c
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int x, y, z[1000];
#pragma omp threadprivate(x)

void fun(int a) {
   const int c = 1;
   int i = 0;

   #pragma omp parallel default(none) private(a) shared(z)
   {
      int j = omp_get_num_thread();
             //O.K.  - j is declared within parallel region
      a = z[j];       // O.K.  - a is listed in private clause
                      //      - z is listed in shared clause
      x = c;          // O.K.  - x is threadprivate
                      //      - c has const-qualified type
      z[i] = y;       // C3052 error - cannot reference i or y here

      #pragma omp for firstprivate(y)
         for (i=0; i<10 ; i++) {
            z[i] = y;  // O.K. - i is the loop control variable
                       // - y is listed in firstprivate clause
          }
       z[i] = y;   // Error - cannot reference i or y here
   }
}
```

For more information on the **default** clause, see Section 2.7.2.5 on page 28.

A.23 Examples of the ordered Directive

It is possible to have multiple ordered sections with a **for** specified with the **ordered** clause. The first example is noncompliant because the API specifies the following:

"An iteration of a loop with a **for** construct must not execute the same **ordered** directive more than once, and it must not execute more than one **ordered** directive." (See Section 2.6.6 on page 22)

In this noncompliant example, all iterations execute 2 ordered sections:

```
#pragma omp for ordered
for (i=0; i<n; i++)
{
    ...
    #pragma omp ordered
    { ... }
    ...
    #pragma omp ordered
    { ... }
    ...
}
```

The following compliant example shows a **for** with more than one ordered section:

```
#pragma omp for ordered
for (i=0; i<n; i++)
{
    ...
    if (i <= 10)
    {
        ...
        #pragma omp ordered
        { ... }
    }
    ...
    (i > 10)
    {
        ...
        #pragma omp ordered
        { ... }
    }
    ...
}
```

A.24 Example of the private Clause

The **private** clause (Section 2.7.2.1 on page 25) of a parallel region is only in effect for the lexical extent of the region, not for the dynamic extent of the region. Therefore, in the example that follows, any uses of the variable *a* within the **for** loop in the routine *f* refers to a private copy of *a*, while a usage in routine *g* refers to the global *a*.

```
int a;

void f(int n)
{
    a = 0;

    #pragma omp parallel for private(a)
    for (int i=1; i<n; i++)
    {
        a = i;
        g(i, n);
        d(a);     // Private copy of "a"
        ...
    }
    ...

void g(int k, int n)
{
    h(k,a); // The global "a", not the private "a" in f
}
```

A.25 Examples of the copyprivate Data Attribute Clause

Example 1: The **copyprivate** clause (Section 2.7.2.8 on page 32) can be used to broadcast values acquired by a single thread directly to all instances of the private variables in the other threads.

```
float x, y;
#pragma omp threadprivate(x, y)

void init( )
{
    float a;
    float b;

    #pragma omp single copyprivate(a,b,x,y)
    {
        get_values(a,b,x,y);
    }

    use_values(a, b, x, y);
}
```

If routine *init* is called from a serial region, its behavior is not affected by the presence of the directives. After the call to the *get_values* routine has been executed by one thread, no thread leaves the construct until the private objects designated by *a*, *b*, *x*, and *y* in all threads have become defined with the values read.

Example 2: In contrast to the previous example, suppose the read must be performed by a particular thread, say the master thread. In this case, the **copyprivate** clause cannot be used to do the broadcast directly, but it can be used to provide access to a temporary shared object.

```
float read_next( )
{
    float * tmp;
    float return_val;

    #pragma omp single copyprivate(tmp)
    {
        tmp = (float *) malloc(sizeof(float));
    }

    #pragma omp master
    {
        get_float( tmp );
    }

    #pragma omp barrier
    return_val = *tmp;
    #pragma omp barrier

    #pragma omp single
    {
       free(tmp);
    }

    return return_val;
}
```

Example 3: Suppose that the number of lock objects required within a parallel region cannot easily be determined prior to entering it. The **copyprivate** clause can be used to provide access to shared lock objects that are allocated within that parallel region.

```
#include <omp.h>

omp_lock_t *new_lock()
```

```
{
    omp_lock_t *lock_ptr;

    #pragma omp single copyprivate(lock_ptr)
    {
        lock_ptr = (omp_lock_t *) malloc(sizeof(omp_lock_t));
        omp_init_lock( lock_ptr );
    }

    return lock_ptr;
}
```

A.26 Using the threadprivate Directive

The following examples demonstrate how to use the **threadprivate** directive (Section 2.7.1 on page 23) to give each thread a separate counter.

Example 1:

```
int counter = 0;
#pragma omp threadprivate(counter)

int sub()
{
    counter++;
    return(counter);
}
```

Example 2:

```
int sub()
{
    static int counter = 0;
    #pragma omp threadprivate(counter)
    counter++;
    return(counter);
}
```

A.27 Use of C99 Variable Length Arrays

The following example demonstrates how to use C99 Variable Length Arrays (VLAs) in a **firstprivate** directive (Section 2.7.2.2 on page 26).

Note

Variable length arrays are not currently supported in Visual C++.

```
void f(int m, int C[m][m])
{
    double v1[m];
    ...
    #pragma omp parallel firstprivate(C, v1)
    ...
}
```

A.28 Use of num_threads Clause

The following example demonstrates the **num_threads** clause (Section 2.3 on page 8). The parallel region is executed with a maximum of 10 threads.

```
#include <omp.h>
main()
{
    omp_set_dynamic(1);
    ...
    #pragma omp parallel num_threads(10)
    {
        ... parallel region ...
    }
}
```

A.29 Use of Work-Sharing Constructs Inside a critical Construct

The following example demonstrates using a work-sharing construct inside a **critical** construct. This example is compliant because the work-sharing construct and the **critical** construct do not bind to the same parallel region.

```
void f()
{
  int i = 1;
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      #pragma omp critical (name)
      {
        #pragma omp parallel
        {
          #pragma omp single
          {
            i++;
          }
        }
      }
    }
  }
}
```

A.30 Use of Reprivatization

The following example demonstrates the reprivatization of variables. Private variables can be marked **private** again in a nested directive. They do not have to be shared in the enclosing parallel region.

```
int i, a;
...
#pragma omp parallel private(a)
{
  ...
  #pragma omp parallel for private(a)
  for (i=0; i<10; i++)
     {
        ...
     }
}
```

A.31 Thread-Safe Lock Functions

The following C++ example demonstrates how to initialize an array of locks in a parallel region by using **omp_init_lock** (Section 3.2.1 on page 42).

Example

```
// A_13_omp_init_lock.cpp
// compile with: /openmp
#include <omp.h>

omp_lock_t *new_locks() {
   int i;
   omp_lock_t *lock = new omp_lock_t[1000];
   #pragma omp parallel for private(i)
   for (i = 0 ; i < 1000 ; i++)
      omp_init_lock(&lock[i]);

   return lock;
}

int main () {}
```

B. Stubs for Run-time Library Functions

This section provides stubs for the run-time library functions defined in the OpenMP C and C++ API. The stubs are provided to enable portability to platforms that do not support the OpenMP C and C++ API. On these platforms, OpenMP programs must be linked with a library containing these stub functions. The stub functions assume that the directives in the OpenMP program are ignored. As such, they emulate serial semantics.

Note

The lock variable that appears in the lock functions must be accessed exclusively through these functions. It should not be initialized or otherwise modified in the user program. Users should not make assumptions about mechanisms used by OpenMP C and C++ implementations to implement locks based on the scheme used by the stub functions.

```
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"
#ifdef __cplusplus
extern "C" {
#endif


void omp_set_num_threads(int num_threads)
{
}
int omp_get_num_threads(void)
{
    return 1;
}
int omp_get_max_threads(void)
{
    return 1;
}
int omp_get_thread_num(void)
{
    return 0;
}
int omp_get_num_procs(void)
{
    return 1;
}
void omp_set_dynamic(int dynamic_threads)
{
}
int omp_get_dynamic(void)
{
    return 0;
}
int omp_in_parallel(void)
{
    return 0;
}
void omp_set_nested(int nested)
{
}
int omp_get_nested(void)
{
    return 0;
}
enum {UNLOCKED = -1, INIT, LOCKED};
void omp_init_lock(omp_lock_t *lock)
{
    *lock = UNLOCKED;
}
void omp_destroy_lock(omp_lock_t *lock)
```

```
{
    *lock = INIT;
}
void omp_set_lock(omp_lock_t *lock)
{
    if (*lock == UNLOCKED)
    {
        *lock = LOCKED;
    }
    else
        if (*lock == LOCKED)
        {
         fprintf_s(stderr, "error: deadlock in using lock variable\n");
         exit(1);
        } else {
         fprintf_s(stderr, "error: lock not initialized\n");
         exit(1);
        }
}


void omp_unset_lock(omp_lock_t *lock)
{
    if (*lock == LOCKED)
    {
        *lock = UNLOCKED;
    }
    else
        if (*lock == UNLOCKED)
        {
            fprintf_s(stderr, "error: lock not set\n");
            exit(1);
        } else {
            fprintf_s(stderr, "error: lock not initialized\n");
            exit(1);
        }
}

int omp_test_lock(omp_lock_t *lock)
{
    if (*lock == UNLOCKED)
    {
        *lock = LOCKED;
        return 1;
    } else if (*lock == LOCKED) {
        return 0;
    } else {
        fprintf_s(stderr, "error: lock not initialized\n");
        exit(1);
    }
}

#ifndef OMP_NEST_LOCK_T
typedef struct {  // This really belongs in omp.h
    int owner;
    int count;
} omp_nest_lock_t;
#endif
enum {MASTER = 0};
void omp_init_nest_lock(omp_nest_lock_t *lock)
{
    lock->owner = UNLOCKED;
    lock->count = 0;
}
void omp_destroy_nest_lock(omp_nest_lock_t *lock)
{
    lock->owner = UNLOCKED;
    lock->count = UNLOCKED;
```

```
}


void omp_set_nest_lock(omp_nest_lock_t *lock)
{
    if (lock->owner == MASTER && lock->count >= 1)
    {
        lock->count++;
    } else
        if (lock->owner == UNLOCKED && lock->count == 0)
        {
            lock->owner = MASTER;
            lock->count = 1;
        } else
        {
       fprintf_s(stderr, "error: lock corrupted or not initialized\n");
         exit(1);
    }
}

void omp_unset_nest_lock(omp_nest_lock_t *lock)
{
    if (lock->owner == MASTER && lock->count >= 1)
    {
        lock->count--;
        if (lock->count == 0)
        {
            lock->owner = UNLOCKED;
        }
    } else
        if (lock->owner == UNLOCKED && lock->count == 0)
        {
            fprintf_s(stderr, "error: lock not set\n");
            exit(1);
        } else
        {
       fprintf_s(stderr, "error: lock corrupted or not initialized\n");
       exit(1);
    }
}


int omp_test_nest_lock(omp_nest_lock_t *lock)
{
    omp_set_nest_lock(lock);
    return lock->count;
}


double omp_get_wtime(void)
{
    // This function does not provide a working
    // wallclock timer. Replace it with a version
    // customized for the target machine.
    return 0.0;
}

double omp_get_wtick(void)
{
    // This function does not provide a working
    // clock tick function. Replace it with
    // a version customized for the target machine.
    return 365. * 86400.;
}

#ifdef __cplusplus
}
#endif
```

C. OpenMP C and C++ Grammar

C.1 Notation

The grammar rules consist of the name for a non-terminal, followed by a colon, followed by replacement alternatives on separate lines.

The syntactic expression term_{opt} indicates that the term is optional within the replacement.

The syntactic expression $\textit{term}_{\text{optseq}}$ is equivalent to $\textit{term-seq}_{\text{opt}}$ with the following additional rules:

term-seq :

term

term-seq term

term-seq , *term*

C.2 Rules

The notation is described in section 6.1 of the C standard. This grammar appendix shows the extensions to the base language grammar for the OpenMP C and C++ directives.

/* in C++ (ISO/IEC 14882:1998) */

statement-seq:

statement

openmp-directive

statement-seq statement

statement-seq openmp-directive

/* in C90 (ISO/IEC 9899:1990) */

statement-list:

statement

openmp-directive

statement-list statement

statement-list openmp-directive

/* in C99 (ISO/IEC 9899:1999) */

block-item:

declaration

statement

openmp-directive

statement:

/* standard statements */

openmp-construct

openmp-construct:

parallel-construct

for-construct

sections-construct

single-construct

parallel-for-construct

parallel-sections-construct

master-construc

critical-construct

atomic-construct

ordered-construct

openmp-directive:

barrier-directive

flush-directive

structured-block:

statement

parallel-construct:

parallel-directive structured-block

parallel-directive:

 # pragma omp parallel *parallel-clause*$_{\text{optseq}}$ *new-line*

parallel-clause:

unique-parallel-clause

data-clause

unique-parallel-clause:

if (*expression* **)**

num_threads (*expression* **)**

for-construct:

for-directive iteration-statement

for-directive:

 # pragma omp for *for-clause*$_{\text{optseq}}$ *new-line*

for-clause:

unique-for-clause

data-clause

nowait

unique-for-clause:

ordered

schedule (*schedule-kind* **)**

schedule (*schedule-kind* **,** *expression* **)**

schedule-kind:

static

dynamic

guided

runtime

sections-construct:

sections-directive section-scope

sections-directive:

 # pragma omp sections *sections-clause*$_{\text{optseq}}$ *new-line*

sections-clause:

data-clause

nowait

section-scope:

{ section-sequence }

section-sequence:

*section-directive*opt *structured-block*

section-sequence section-directive structured-block

section-directive:

 # pragma omp section *new-line*

single-construct:

single-directive structured-block

single-directive:

 # pragma omp single *single-clause*$_{\text{optseq}}$ *new-line*

single-clause:

data-clause

nowait

parallel-for-construct:

parallel-for-directive iteration-statement

parallel-for-directive:

 # pragma omp parallel for *parallel-for-clause*$_{\text{optseq}}$ *new-line*

parallel-for-clause:

unique-parallel-clause

unique-for-clause

data-clause

parallel-sections-construct:

parallel-sections-directive section-scope

parallel-sections-directive:

 # pragma omp parallel sections *parallel-sections-clause*$_{\text{optseq}}$ *new-line*

parallel-sections-clause:

unique-parallel-clause

data-clause

master-construct:

master-directive structured-block

master-directive:

 # pragma omp master *new-line*

critical-construct:

critical-directive structured-block

critical-directive:

 # pragma omp critical *region-phrase*$_{\text{opt}}$ *new-line*

region-phrase:

(identifier)

barrier-directive:

 # pragma omp barrier *new-line*

atomic-construct:

atomic-directive expression-statement

atomic-directive:

 # pragma omp atomic *new-line*

flush-directive:

 # pragma omp flush *flush-vars*$_{opt}$ *new-line*

flush-vars:

(variable-list)

ordered-construct:

ordered-directive structured-block

ordered-directive:

 # pragma omp ordered *new-line*

declaration:

/* standard declarations */

threadprivate-directive

threadprivate-directive:

 # pragma omp threadprivate (*variable-list* **)** *new-line*

data-clause:

private (*variable-list* **)**

copyprivate (*variable-list* **)**

firstprivate (*variable-list* **)**

lastprivate (*variable-list* **)**

shared (*variable-list* **)**

default (shared)

default (none)

reduction (*reduction-operator* **:** *variable-list* **)**

copyin (*variable-list* **)**

reduction-operator:

One of: **+ * - & ^ | && ||**

/* in C */

variable-list:

identifier

variable-list **,** *identifier*

/* in C++ */

variable-list:

id-expression

variable-list **,** *id-expression*

D. Using the schedule Clause

A parallel region has at least one barrier, at its end, and may have additional barriers within it. At each barrier, the other members of the team must wait for the last thread to arrive. To minimize this wait time, shared work should be distributed so that all threads arrive at the barrier at about the same time. If some of that shared work is contained in **for** constructs, the **schedule** clause can be used for this purpose.

When there are repeated references to the same objects, the choice of schedule for a **for** construct may be determined primarily by characteristics of the memory system, such as the presence and size of caches and whether memory access times are uniform or nonuniform. Such considerations may make it preferable to have each thread consistently refer to the same set of elements of an array in a series of loops, even if some threads are assigned relatively less work in some of the loops. This can be done by using the **static** schedule with the same bounds for all the loops. In the following example, note that zero is used as the lower bound in the second loop, even though **k** would be more natural if the schedule were not important.

```
#pragma omp parallel
{
#pragma omp for schedule(static)
  for(i=0; i<n; i++)
    a[i] = work1(i);
#pragma omp for schedule(static)
  for(i=0; i<n; i++)
    if(i>=k) a[i] += work2(i);
}
```

In the remaining examples, it is assumed that memory access is not the dominant consideration, and, unless otherwise stated, that all threads receive comparable computational resources. In these cases, the choice of schedule for a **for** construct depends on all the shared work that is to be performed between the nearest preceding barrier and either the implied closing barrier or the nearest subsequent barrier, if there is a **nowait** clause. For each kind of schedule, a short example shows how that schedule kind is likely to be the best choice. A brief discussion follows each example.

The **static** schedule is also appropriate for the simplest case, a parallel region containing a single **for** construct, with each iteration requiring the same amount of work.

```
#pragma omp parallel for schedule(static)
for(i=0; i<n; i++) {
  invariant_amount_of_work(i);
}
```

The **static** schedule is characterized by the properties that each thread gets approximately the same number of iterations as any other thread, and each thread can independently determine the iterations assigned to it. Thus no synchronization is required to distribute the work, and, under the assumption that each iteration requires the same amount of work, all threads should finish at about the same time.

For a team of p threads, let *ceiling(n/p)* be the integer q, which satisfies $n = p*q - r$ with $0 <= r < p$. One implementation of the **static** schedule for this example would assign q iterations to the first $p-1$ threads, and $q-r$ iterations to the last thread. Another acceptable implementation would assign q iterations to the first $p-r$ threads, and $q-1$ iterations to the remaining r threads. This illustrates why a program should not rely on the details of a particular implementation.

The **dynamic** schedule is appropriate for the case of a **for** construct with the iterations requiring varying, or even unpredictable, amounts of work.

```
#pragma omp parallel for schedule(dynamic)
  for(i=0; i<n; i++) {
    unpredictable_amount_of_work(i);
}
```

The **dynamic** schedule is characterized by the property that no thread waits at the barrier for longer than it takes another thread to execute its final iteration. This requires that iterations be assigned one at a time to threads as they become available, with synchronization for each assignment. The synchronization overhead can be reduced by specifying a minimum chunk size k greater than 1, so that threads are assigned k at a time until fewer than k remain. This guarantees that no thread waits at the barrier longer than it takes another thread to execute its final chunk of (at most) k iterations.

The **dynamic** schedule can be useful if the threads receive varying computational resources, which has much the same effect as varying amounts of work for each iteration. Similarly, the dynamic schedule can also be useful if the threads arrive at the **for** construct at varying times, though in some of these cases the **guided** schedule may be preferable.

The **guided** schedule is appropriate for the case in which the threads may arrive at varying times at a **for** construct with each iteration requiring about the same amount of work. This can happen if, for example, the **for** construct is preceded by one or more sections or **for** constructs with **nowait** clauses.

```
#pragma omp parallel
{
  #pragma omp sections nowait
  {
    // ...
  }
  #pragma omp for schedule(guided)
  for(i=0; i<n; i++) {
    invariant_amount_of_work(i);
  }
}
```

Like **dynamic**, the **guided** schedule guarantees that no thread waits at the barrier longer than it takes another thread to execute its final iteration, or final *k* iterations if a chunk size of *k* is specified. Among such schedules, the **guided** schedule is characterized by the property that it requires the fewest synchronizations. For chunk size *k*, a typical implementation will assign $q = ceiling(n/p)$ iterations to the first available thread, set n to the larger of $n-q$ and $p*k$, and repeat until all iterations are assigned.

When the choice of the optimum schedule is not as clear as it is for these examples, the **runtime** schedule is convenient for experimenting with different schedules and chunk sizes without having to modify and recompile the program. It can also be useful when the optimum schedule depends (in some predictable way) on the input data to which the program is applied.

To see an example of the trade-offs between different schedules, consider sharing 1000 iterations among 8 threads. Suppose there is an invariant amount of work in each iteration, and use that as the unit of time.

If all threads start at the same time, the **static** schedule will cause the construct to execute in 125 units, with no synchronization. But suppose that one thread is 100 units late in arriving. Then the remaining seven threads wait for 100 units at the barrier, and the execution time for the whole construct increases to 225.

Because both the **dynamic** and **guided** schedules ensure that no thread waits for more than one unit at the barrier, the delayed thread causes their execution times for the construct to increase only to 138 units, possibly increased by delays from synchronization. If such delays are not negligible, it becomes important that the number of synchronizations is 1000 for **dynamic** but only 41 for **guided**, assuming the default chunk size of one. With a chunk size of 25, **dynamic** and **guided** both finish in 150 units, plus any delays from the required synchronizations, which now number only 40 and 20, respectively.

E. Implementation-Defined Behaviors in OpenMP C/C++

This appendix summarizes the behaviors that are described as "implementation-defined" in this API. Each behavior is cross-referenced back to its description in the main specification.

Remarks

An implementation is required to define and document its behavior in these cases, but this list may be incomplete.

- **Number of threads:** If a parallel region is encountered while dynamic adjustment of the number of threads is disabled, and the number of threads requested for the parallel region exceeds the number that the run-time system can supply, the behavior of the program is implementation-defined (see page 9).

  In Visual C++, for a non-nested parallel region, 64 threads (the maximum) will be provided.

- **Number of processors:** The number of physical processors actually hosting the threads at any given time is implementation-defined (see page 10).

  In Visual C++, this number is not constant, and is controlled by the operating system.

- **Creating teams of threads:** The number of threads in a team that execute a nested parallel region is implementation-defined.(see page 10).

  In Visual C++, this is determined by the operating system.

- **schedule(runtime):** The decision regarding scheduling is deferred until run time. The schedule type and chunk size can be chosen at run time by setting the **OMP_SCHEDULE** environment variable. If this environment variable is not set, the resulting schedule is implementation-defined (see page 13).

  In Visual C++, schedule type is **static** with no chunk size.

- **Default scheduling:** In the absence of the schedule clause, the default schedule is implementation-defined (see page 13).

  In Visual C++, the default schedule type is **static** with no chunk size.

- **ATOMIC:** It is implementation-defined whether an implementation replaces all **atomic** directives with **critical** directives that have the same unique name (see page 20).

  In Visual C++, if data modified by atomic is not on a natural alignment or if it is 1 or 2 bytes long all atomic operations that satisfy that property will use one critical section. Otherwise, critical sections will not be used.

- **omp_get_num_threads:** If the number of threads has not been explicitly set by the user, the default is implementation-defined (see page 9, and Section 3.1.2 on page 37).

  In Visual C++, the default number of threads is equal to the number of processors.

- **omp_set_dynamic:** The default for dynamic thread adjustment is implementation-defined (see Section 3.1.7 on page 39).

  In Visual C++, the default is **FALSE**.

- **omp_set_nested:** When nested parallelism is enabled, the number of threads used to execute nested parallel regions is implementation-defined (see Section 3.1.9 on page 40).

  In Visual C++, the number of threads is determined by the operating system.

- **OMP_SCHEDULE** environment variable: The default value for this environment variable is implementation-defined (see Section 4.1 on page 48).

  In Visual C++, schedule type is **static** with no chunk size.

- **OMP_NUM_THREADS** environment variable: If no value is specified for the **OMP_NUM_THREADS** environment variable, or if the value specified is not a positive integer, or if the value is greater than the maximum number of threads the system can support, the number of threads to use is implementation-defined (see Section 4.2 on page 48).

  In Visual C++, if value specified is zero or less, the number of threads is equal to the number of processors. If value is greater than 64, the number of threads is 64.

- **OMP_DYNAMIC** environment variable: The default value is implementation-defined (see Section 4.3 on page 49).

  In Visual C++, the default is **FALSE**.

F. New Features and Clarifications in Version 2.0

This appendix summarizes the key changes made to the OpenMP C/C++ specification in moving from version 1.0 to version 2.0. The following items are new features added to the specification:

- Commas are permitted in OpenMP directives (Section 2.1 on page 7).
- Addition of the **num_threads** clause. This clause allows a user to request a specific number of threads for a parallel construct (Section 2.3 on page 8).
- The **threadprivate** directive has been extended to accept static block-scope variables (Section 2.7.1 on page 23).
- C99 Variable Length Arrays are complete types, and thus can be specified anywhere complete types are allowed, for instance in the lists of **private**, **firstprivate**, and **lastprivate** clauses (Section 2.7.2 on page 25).
- A private variable in a parallel region can be marked private again in a nested directive (Section 2.7.2.1 on page 25).
- The **copyprivate** clause has been added. It provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members. It is an alternative to using a shared variable for the value when providing such a shared variable would be difficult (for example, in a recursion requiring a different variable at each level). The **copyprivate** clause can only appear on the **single** directive (Section 2.7.2.8 on page 32).
- Addition of timing routines **omp_get_wtick** and **omp_get_wtime** similar to the MPI routines. These functions are necessary for performing wall clock timings (Section 3.3.1 on page 44 and Section 3.3.2 on page 45).
- An appendix with a list of implementation-defined behaviors in OpenMP C/C++ has been added. An implementation is required to define and document its behavior in these cases (Appendix E on page 97).
- The following changes serve to clarify or correct features in the previous OpenMP API specification for C/C++:
    - Clarified that the behavior of **omp_set_nested** and **omp_set_dynamic** when **omp_in_parallel** returns nonzero is undefined (Section 3.1.7 on page 39, and Section 3.1.9 on page 40).
    - Clarified directive nesting when nested parallel is used (Section 2.9 on page 33).
    - The lock initialization and lock destruction functions can be called in a parallel region (Section 3.2.1 on page 42 and Section 3.2.2 on page 42).
    - New examples have been added (Appendix A on page 51).

Run-Time Library Reference

The Microsoft run-time library provides routines for programming for the Microsoft Windows operating system. These routines automate many common programming tasks that are not provided by the C and C++ languages.

Sample programs are included in the individual reference topics for most routines in the library.

In This Section

C Run-Time Libraries

Discusses the .lib files that comprise the C run-time libraries.

Run-Time Routines by Category

Provides links to the run-time library by category.

Global Variables and Standard Types

Provides links to the global variables and standard types provided by the run-time library.

Global Constants

Provides links to the global constants defined by the run-time library.

Alphabetical Function Reference

Provides a table of contents entry point into an alphabetical listing of all C run-time library functions.

Generic-Text Mappings

Provides links to the generic-text mappings defined in Tchar.h.

Language and Country/Region Strings

Describes how to use the **setlocale** function to set the language and Country/Region strings.

Related Sections

Debug Routines

Provides links to the debug versions of the run-time library routines.

Run-Time Error Checking

Provides links to functions that support run-time error checks.

Run-Time Library Behavior

Discusses the entry point into the CRT DLL.

Visual C++ Libraries

Provides links to the various libraries provided with Visual C++, including ATL, MFC, OLE DB Templates, the C run-time library, and the Standard C++ Library.

Debugging

Provides links to using the Visual Studio debugger to correct logic errors in your application or stored procedures.

C Run-Time Libraries

This topic discusses the various .lib files that comprise the C run-time libraries as well as their associated compiler options and preprocessor directives.

C Run-Time Libraries (CRT)

The following libraries contain the C run-time library functions.

C run-time library (without iostream or standard C++ library)	Associated DLL	Characteristics	Option	Preprocessor directives
libcmt.lib	None, static link.	Multithreaded, static link	/MT	_MT
msvcrt.lib	msvcr80.dll	Multithreaded, dynamic link (import library for MSVCR80.DLL). Be aware that if you use the Standard C++ Library, your program will need MSVCP80.DLL to run.	/MD	_MT, _DLL
libcmtd.lib	None, static link	Multithreaded, static link (debug)	**/MTd**	_DEBUG, _MT
msvcrtd.lib	msvcr80d.dll	Multithreaded, dynamic link (import library for MSVCR80D.DLL) (debug).	**/MDd**	_DEBUG, _MT, _DLL
msvcmrt.lib	msvcm80.dll	C Runtime import library. Used for mixed managed/native code.	**/clr**	
msvcurt.lib	msvcm80.dll	C Runtime import library compiled as 100% pure MSIL code. All code complies with the ECMA URT spec for MSIL.	**/clr:pure**	

Note

The single-threaded CRT (libc.lib, libcd.lib) (formerly the **/ML** or **/MLd** options) is no longer available. Instead, use the multithreaded CRT. See Multithreaded Libraries Performance.

If you link your program from the command line without a compiler option that specifies a C run-time library, the linker will use LIBCMT.LIB. This is different from previous versions of Visual C++ which used LIBC.LIB, the single-threaded library, instead.

Using the statically linked CRT implies that any state information saved by the C runtime library will be local to that instance of the CRT. For example, if you use strtok, _strtok_l, wcstok, _wcstok_l, _mbstok, _mbstok_l when using a statically linked CRT, the position of the **strtok** parser is unrelated to the **strtok** state used in code in the same process (but in a different DLL or EXE) that is linked to another instance of the static CRT. In contrast, the dynamically linked CRT shares state for all code within a process that is dynamically linked to the CRT. This concern does not apply if you use the new more secure versions of these functions; for example, **strtok_s** does not have this problem.

Because a DLL built by linking to a static CRT will have its own CRT state, it is not recommended to link statically to the CRT in a DLL unless the consequences of this are specifically desired and understood. For example, if you call _set_se_translator in an executable that loads the DLL linked to its own static CRT, any hardware exceptions generated by the code in the DLL will not be caught by the translator, but hardware exceptions generated by code in the main executable will be caught.

If you are using the **/clr** compiler switch, your code will be linked with an import library, msvcmrt.lib. The import library references a new library, msvcm80.dll, which provides a proxy between your managed code and the native CRT. You cannot use the statically linked CRT (**/MT** or **/MTd** options) with **/clr**. Use the dynamically-linked libraries (**/MD** or **/MDd**) instead.

If you are using the **/clr:pure** compiler switch, your code will be linked with the import library msvcurt.lib, which also

references msvcm80.dll. As with **/clr**, you cannot link with the statically linked library.

For more information on using the CRT with **/clr**, see Mixed (Native and Managed) Assemblies; for **/clr:pure**, see Pure and Verifiable Code.

To build a debug version of your application, the _DEBUG flag must be defined and the application must be linked with a debug version of one of these libraries. For more information about using the debug versions of the library files, see CRT Debugging Techniques.

This version of Visual C++ is not conformant with the C99 standard.

Standard C++ Library

Standard C++ Library	Characteristics	Option	Preprocessor directives
LIBCPMT.LIB	Multithreaded, static link	**/MT**	_MT
MSVCPRT.LIB	Multithreaded, dynamic link (import library for MSVCP80.dll)	**/MD**	_MT, _DLL
LIBCPMTD.LIB	Multithreaded, static link	**/MTd**	_DEBUG, _MT
MSVCPRTD.LIB	Multithreaded, dynamic link (import library for MSVCP80D.DLL)	**/MDd**	_DEBUG, _MT, _DLL

Note Starting in Visual C++ 2005, LIBCP.LIB and LIBCPD.LIB (via the old **/ML** and **/MLd** options) have been removed. Use LIBCPMT.LIB and LIBCPMTD.LIB instead via the **/MT** and **/MTd** options.

When you build a release version of your project, one of the basic C run-time libraries (LIBCMT.LIB, MSVCMRT.LIB, MSVCRT.LIB) is linked by default, depending on the compiler option you choose (multithreaded, DLL, /clr). If you include one of the Standard C++ Library Header Files in your code, a Standard C++ Library will be linked in automatically by Visual C++ at compile time. For example:

```
#include <ios>
```

What is the difference between msvcrt.dll and msvcr80.dll?

The msvcrt.dll is now a "known DLL," meaning that it is a system component owned and built by Windows. It is intended for future use only by system-level components.

What problems exist if an application uses both msvcrt.dll and msvcr80.dll?

If you have a .lib or .obj file that needs to link to msvcrt.lib, then you should not have to recompile it to work with the new msvcrt.lib in Visual C++ 2005. The .lib or .obj file may rely on the sizes, field offsets, or member function names of various CRT classes or variables, and those should all still exist in a compatible way. When you relink against msvcrt.lib, your final EXE and DLL image will now have a dependency on msvcr80.dll instead of msvcrt.dll.

If you have more than one DLL or EXE, then you may have more than one CRT, whether or not you are using different versions of Visual C++. For example, statically linking the CRT into multiple DLLs can present the same problem. Developers encountering this problem with static CRTs have been instructed to compile with **/MD** to use the CRT DLL. Now that the CRT DLL has been renamed to msvcr80.dll, applications may have some components linked to msvcrt.dll and others to msvcr80.dll. If your DLLs pass CRT resources across the msvcrt.dll and msvcr80.dll boundary, you will encounter issues with mismatched CRTs and need to recompile your project with Visual C++ 2005.

If your program is using more than one version of the CRT, some care is needed when passing certain CRT objects (such as file handles, locales and environment variables) across DLL boundaries. For more information on the issues involved and how to resolve them, see Potential Errors Passing CRT Objects Across DLL Boundaries.

Standard Libraries in WinSxS Global Assembly Cache

In Visual C++ 2005, the C and C++ runtime libraries (e.g. msvcm80.dll, msvcm80d.dll) are stored in the global assembly cache.

See Also

Other Resources

Run-Time Library Reference

Breaking Changes (CRT)

Breaking changes in the CRT.

Breaking changes

- Many functions have been deprecated. See Deprecated CRT Functions.
- Many functions now validate their parameters, halting execution if given invalid parameters. This may break code that passes invalid parameters and relies on the function ignoring them or just returning an error code. See Parameter Validation.
- The file descriptor value -2 is now used to indicate that **stdout** and **stderr** are not available for output, as for example in a Windows application that has no console window. The previous value used was -1. For more information, see _fileno.
- The single-threaded CRT libraries, **libc.lib** and **libcd.lib**, have been removed. Use the multi-threaded CRT libraries. The /ML compiler flag is no longer supported. Non-locking versions of some functions have been added in cases where the performance difference between the multithreaded code and the single-threaded code is potentially significant.
- The overload of pow, **double pow(int, int)**, was removed to better conform with the standard.
- The **%n** format specifier is no longer supported by default in any of the printf family of functions because it is inherently insecure. The default behavior if **%n** is encountered is to invoke the invalid parameter handler. To enable **%n** support, use _set_printf_count_output (also see _get_printf_count_output).
- sprintf now prints the negative sign of a signed zero.
- swprintf has been changed to conform with the Standard; it now requires a size parameter. The form of **swprintf** without a size parameter has been deprecated.
- **_set_security_error_handler** has been removed. Remove any calls to that function; the default handler is a much safer way of dealing with security errors.
- **time_t** is now a 64-bit value (unless _USE_32BIT_TIME_T is defined).
- The _spawn, _wspawn Functions now leave errno untouched on success, as specified by the C Standard.
- RTC now uses wide characters by default.
- Floating-point control word support functions have been deprecated for applications compiled with /CLR or /CLR:PURE. The affected functions are _clear87, _clearfp, _control87, _controlfp, _fpreset, _status87, _statusfp. You can disable the deprecation warning by defining _CRT_MANAGED_FP_NO_DEPRECATE, but the use of these functions in managed code is unpredictable and unsupported.
- Some functions now return **const** pointers. The old, non-const behavior can be reinstated by defining _CONST_RETURN. The affected functions are
  - memchr, wmemchr
  - strchr, wcschr, _mbschr, _mbschr_l
  - strpbrk, wcspbrk, _mbspbrk, _mbspbrk_l
  - strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
  - strstr, wcsstr, _mbsstr, _mbsstr_l
- When linking with Setargv.obj or Wsetargv.obj, it is no longer possible to suppress the expansion of a wildcard character on the command line by enclosing it in double quotes. For more information, see Expanding Wildcard Arguments.

See Also

Reference

C Run-Time Libraries

Concepts

What's New in the Visual C++ Libraries

Building the Run-Time Libraries

The following procedure shows you how to build your own copies of the C run-time library files.

To build your own copies of the C run-time library files

1. If you have not already done so, perform a custom install of Visual C++ and install the CRT source directory.
2. Set the VCTOOLS environment variable to be set to the Vc directory. For example, on the command line type:

```
set VCTOOLS=C:\Program Files\Microsoft Visual Studio .NET 2003\Vc
```

3. Set environment variable LLP64=1 if you want to build for IA64 or LLP64=2 if you want to build for X64.
4. Change directories to vc\crt\src and depending on your operating, run:
   - BLDWIN9x.BAT, if you are Windows 98 or Windows Me.
   - BLDNT.CMD, if you are on Windows NT, Windows 2000, or Windows XP.

The CRT rebuild produces customized versions of CRT DLLs and corresponding import .lib files. To avoid conflicts with the existing, registered DLLs, the build process creates files with different names, using prefix sample instead of msvc. The following table shows the correspondence between the files produced by the CRT rebuild process and the CRT files installed by Visual C++:

New file	CRT file equivalent
sample.dll	msvcr80.dll
sample_p.dll	msvcp80.dll
sampld.dll	msvcr80d.dll
sampld_p.dll	msvcp80d.dll
sample.lib	msvcr80.lib
sample_p.lib	msvcp80.lib
sampld.lib	msvcr80d.lib
sampld_p.lib	msvcp80d.lib

Note

You cannot build the CRT using a 64-bit native toolset.

See Also

Reference

C Run-Time Libraries

Compatibility

The Microsoft run-time library supports American National Standards Institute (ANSI) C and UNIX C. In this documentation, references to UNIX include XENIX, other UNIX-like systems, and the POSIX subsystem in Windows 98, Windows ME, Windows NT, Windows 2000, Windows XP, and Windows Server 2003. The description of each run-time library routine includes a compatibility section for the following targets.

- ANSI
- Windows 98
- Windows Me
- Windows NT
- Windows 2000
- Windows XP
- Windows Server 2003

All run-time library routines included with this product are compatible with the Win32 API.

See Also
Reference
C Run-Time Libraries

ANSI C Compliance

The naming convention for all Microsoft-specific identifiers in the run-time system (such as functions, macros, constants, variables, and type definitions) is ANSI-compliant. In this documentation, any run-time function that follows the ANSI/ISO C standards is noted as being ANSI compatible. ANSI-compliant applications should only use these ANSI compatible functions.

The names of Microsoft-specific functions and global variables begin with a single underscore. These names can be overridden only locally, within the scope of your code. For example, when you include Microsoft run-time header files, you can still locally override the Microsoft-specific function named **_open** by declaring a local variable of the same name. However, you cannot use this name for your own global function or global variable.

The names of Microsoft-specific macros and manifest constants begin with two underscores, or with a single leading underscore immediately followed by an uppercase letter. The scope of these identifiers is absolute. For example, you cannot use the Microsoft-specific identifier **_UPPER** for this reason.

See Also
Reference
Compatibility

UNIX

If you plan to port your programs to UNIX, follow these guidelines:

- Do not remove header files from the SYS subdirectory. You can place the SYS header files elsewhere only if you do not plan to transport your programs to UNIX.
- Use the UNIX-compatible path delimiter in routines that take strings representing paths and filenames as arguments. UNIX supports only the forward slash (/) for this purpose, whereas Win32 operating systems support both the backslash (\) and the forward slash (/). Thus this documentation uses UNIX-compatible forward slashes as path delimiters in **#include** statements, for example. (However, the Windows operating system command shell, CMD.EXE, does not support the forward slash in commands entered at the command prompt.)
- Use paths and filenames that work correctly in UNIX, which is case sensitive. The file allocation table (FAT) file system in Win32 operating systems is not case sensitive; the installable Windows NT file system (NTFS) of Windows NT preserves case for directory listings but ignores case in file searches and other system operations.

Note
In this version of Visual C++, UNIX compatibility information has been removed from the function descriptions.

See Also
Reference
Compatibility

Win32 Platforms

The C run-time libraries support Windows 98, Windows Me, Windows NT, Windows 2000, Windows XP, and Windows Server 2003, but not Win32s. Windows 98, Windows Me, Windows NT, Windows 2000, and Windows XP support the Win32 Application Programming Interface (API), but only Windows NT, Windows 2000, Windows XP, and Windows Server 2003 provide full Unicode support. In addition, any Win32 application can use a multibyte character set (MBCS).

See Also

Reference

Compatibility

Backward Compatibility

For compatibility between product versions, the library OLDNAMES.LIB maps old names to new names. For instance, **open** maps to **_open**. You must explicitly link with OLDNAMES.LIB only when you compile with the following combinations of command-line options:

- **/Zl** (omit default library name from object file) and **/Ze** (the default — use Microsoft extensions)
- **/link** (linker-control), **/NOD** (no default-library search), and **/Ze**

For more information about compiler command-line options, see Compiler Reference.

See Also
Reference
Compatibility

Required and Optional Header Files

The description of each run-time routine includes a list of the required and optional include, or header (.H), files for that routine. Required header files need to be included to obtain the function declaration for the routine or a definition used by another routine called internally. Optional header files are usually included to take advantage of predefined constants, type definitions, or inline macros. The following table lists some examples of optional header file contents:

Definition	**Example**
Macro definition	If a library routine is implemented as a macro, the macro definition may be in a header file other than the header file for the original routine. For instance, the **_toupper** macro is defined in the header file CTYPE.H, while the function **toupper** is declared in STDLIB.H.
Predefined Constant	Many library routines refer to constants that are defined in header files. For instance, the **_open** routine uses constants such as **_O_CREAT**, which is defined in the header file FCNTL.H.
Type definition	Some library routines return a structure or take a structure as an argument. For example, stream input/output routines use a structure of type **FILE**, which is defined in STDIO.H.

The run-time library header files provide function declarations in the ANSI/ISO C standard recommended style. The compiler performs type checking on any routine reference that occurs after its associated function declaration. Function declarations are especially important for routines that return a value of some type other than **int**, which is the default. Routines that do not specify their appropriate return value in their declaration will be considered by the compiler to return an **int**, which can cause unexpected results. See Type Checking for more information.

See Also
Reference
C Run-Time Libraries

Files and Streams

A program communicates with the target environment by reading and writing files. A file can be:

- A data set that you can read and write repeatedly.
- A stream of bytes generated by a program (such as a pipeline).
- A stream of bytes received from or sent to a peripheral device.

The last two items are interactive files. Files are typically the principal means by which to interact with a program. You manipulate all these kinds of files in much the same way — by calling library functions. You include the standard header STDIO.H to declare most of these functions.

Before you can perform many of the operations on a file, the file must be opened. Opening a file associates it with a stream, a data structure within the Standard C Library that glosses over many differences among files of various kinds. The library maintains the state of each stream in an object of type FILE.

The target environment opens three files before program startup. You can open a file by calling the library function fopen, _wfopen with two arguments. (The **fopen** function has been deprecated, use fopen_s, _wfopen_s instead.) The first argument is a filename. The second argument is a C string that specifies:

- Whether you intend to read data from the file or write data to it or both.
- Whether you intend to generate new contents for the file (or create a file if it did not previously exist) or leave the existing contents in place.
- Whether writes to a file can alter existing contents or should only append bytes at the end of the file.
- Whether you want to manipulate a text stream or a binary stream.

Once the file is successfully opened, you can then determine whether the stream is byte oriented (a byte stream) or wide oriented (a wide stream). A stream is initially unbound. Calling certain functions to operate on the stream makes it byte oriented, while certain other functions make it wide oriented. Once established, a stream maintains its orientation until it is closed by a call to fclose or freopen.

© 1989-2001 by P.J. Plauger and Jim Brodie. All rights reserved.

See Also
Reference
Text and Binary Streams
Byte and Wide Streams
Controlling Streams
Stream States

Text and Binary Streams

A text stream consists of one or more lines of text that can be written to a text-oriented display so that they can be read. When reading from a text stream, the program reads an `NL` (newline) at the end of each line. When writing to a text stream, the program writes an `NL` to signal the end of a line. To match differing conventions among target environments for representing text in files, the library functions can alter the number and representations of characters transmitted between the program and a text stream.

Thus, positioning within a text stream is limited. You can obtain the current file-position indicator by calling fgetpos or ftell. You can position a text stream at a position obtained this way, or at the beginning or end of the stream, by calling fsetpos or fseek. Any other change of position might well be not supported.

For maximum portability, the program should not write:

- Empty files.
- Space characters at the end of a line.
- Partial lines (by omitting the `NL` at the end of a file).
- characters other than the printable characters, NL, and `HT` (horizontal tab).

If you follow these rules, the sequence of characters you read from a text stream (either as byte or multibyte characters) will match the sequence of characters you wrote to the text stream when you created the file. Otherwise, the library functions can remove a file you create if the file is empty when you close it. Or they can alter or delete characters you write to the file.

A binary stream consists of one or more bytes of arbitrary information. You can write the value stored in an arbitrary object to a (byte-oriented) binary stream and read exactly what was stored in the object when you wrote it. The library functions do not alter the bytes you transmit between the program and a binary stream. They can, however, append an arbitrary number of null bytes to the file that you write with a binary stream. The program must deal with these additional null bytes at the end of any binary stream.

Thus, positioning within a binary stream is well defined, except for positioning relative to the end of the stream. You can obtain and alter the current file-position indicator the same as for a text stream. Moreover, the offsets used by ftell and fseek count bytes from the beginning of the stream (which is byte zero), so integer arithmetic on these offsets yields predictable results.

A byte stream treats a file as a sequence of bytes. Within the program, the stream looks like the same sequence of bytes, except for the possible alterations described above.

See Also

Reference

Files and Streams

Byte and Wide Streams

A byte stream treats a file as a sequence of bytes. Within the program, the stream is the identical sequence of bytes.

By contrast, a wide stream treats a file as a sequence of generalized multibyte characters, which can have a broad range of encoding rules. (Text and binary files are still read and written as previously described.) Within the program, the stream looks like the corresponding sequence of wide characters. Conversions between the two representations occur within the Standard C Library. The conversion rules can, in principle, be altered by a call to setlocale that alters the category **LC_CTYPE**. Each wide stream determines its conversion rules at the time it becomes wide oriented, and retains these rules even if the category **LC_CTYPE** subsequently changes.

Positioning within a wide stream suffers the same limitations as for text steams. Moreover, the file-position indicator may well have to deal with a state-dependent encoding. Typically, it includes both a byte offset within the stream and an object of type **mbstate_t**. Thus, the only reliable way to obtain a file position within a wide stream is by calling fgetpos, and the only reliable way to restore a position obtained this way is by calling fsetpos.

See Also

Reference

Files and Streams

setlocale, _wsetlocale

Controlling Streams

fopen returns the address of an object of type **FILE**. You use this address as the `stream` argument to several library functions to perform various operations on an open file. For a byte stream, all input takes place as if each character is read by calling fgetc, and all output takes place as if each character is written by calling fputc. For a wide stream, all input takes place as if each character is read by calling fgetwc, and all output takes place as if each character is written by calling fputwc.

You can close a file by calling fclose, after which the address of the **FILE** object is invalid.

A **FILE** object stores the state of a stream, including:

- An error indicator set nonzero by a function that encounters a read or write error.
- An end-of-file indicator set nonzero by a function that encounters the end of the file while reading.
- A file-position indicator specifies the next byte in the stream to read or write, if the file can support positioning requests.
- A stream state specifies whether the stream will accept reads and/or writes and whether the stream is unbound, byte oriented, or wide oriented.
- A conversion state remembers the state of any partly assembled or generated generalized multibyte character, as well as any shift state for the sequence of bytes in the file).
- A file buffer specifies the address and size of an array object that library functions can use to improve the performance of read and write operations to the stream.

Do not alter any value stored in a **FILE** object or in a file buffer that you specify for use with that object. You cannot copy a **FILE** object and portably use the address of the copy as a `stream` argument to a library function.

See Also
Reference
Files and Streams

Stream States

The valid states, and state transitions, for a stream are shown in the following figure.

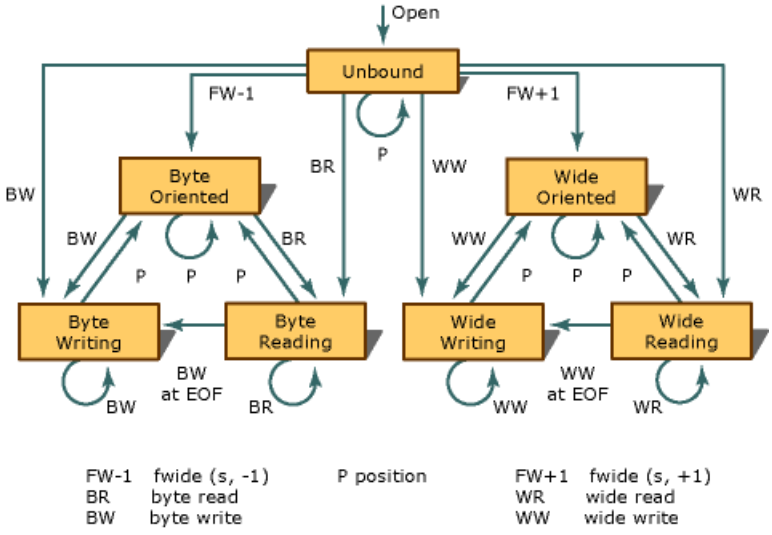

Each of the circles denotes a stable state. Each of the lines denotes a transition that can occur as the result of a function call that operates on the stream. Five groups of functions can cause state transitions.

Functions in the first three groups are declared in <stdio.h>:

- The byte read functions — fgetc, fgets, fread, fscanf, getc, getchar, gets, scanf, and ungetc
- The byte write functions — fprintf, fputc, fputs, fwrite, printf, putc, putchar, puts, vfprintf, and vprintf
- The position functions — fflush, fseek, fsetpos, and rewind

Functions in the remaining two groups are declared in <wchar.h>:

- The wide read functions — fgetwc, fgetws, fwscanf, getwc, getwchar, ungetwc, and wscanf,
- The wide write functions — fwprintf, fputwc, fputws, putwc, putwchar, vfwprintf, vwprintf, and wprintf,

The state diagram shows that you must call one of the position functions between most write and read operations:

- You cannot call a read function if the last operation on the stream was a write.
- You cannot call a write function if the last operation on the stream was a read, unless that read operation set the end-of-file indicator.

Finally, the state diagram shows that a position operation never decreases the number of valid function calls that can follow.

See Also
Reference
Files and Streams

Recommendations for Choosing Between Functions and Macros

Most Microsoft run-time library routines are compiled or assembled functions, but some routines are implemented as macros. When a header file declares both a function and a macro version of a routine, the macro definition takes precedence, because it always appears after the function declaration. When you invoke a routine that is implemented as both a function and a macro, you can force the compiler to use the function version in two ways:

- Enclose the routine name in parentheses.

```
#include <ctype.h>
a = _toupper(a);    // Use macro version of toupper.
a = (_toupper)(a);  // Force compiler to use
                    // function version of toupper.
```

- "Undefine" the macro definition with the **#undef** directive:

```
#include <ctype.h>
#undef _toupper
```

If you need to choose between a function and a macro implementation of a library routine, consider the following trade-offs:

- **Speed versus size** The main benefit of using macros is faster execution time. During preprocessing, a macro is expanded (replaced by its definition) inline each time it is used. A function definition occurs only once regardless of how many times it is called. Macros may increase code size but do not have the overhead associated with function calls.
- **Function evaluation** A function evaluates to an address; a macro does not. Thus you cannot use a macro name in contexts requiring a pointer. For instance, you can declare a pointer to a function, but not a pointer to a macro.
- **Type-checking** When you declare a function, the compiler can check the argument types. Because you cannot declare a macro, the compiler cannot check macro argument types; although it can check the number of arguments you pass to a macro.

See Also
Reference
C Run-Time Libraries

Type Checking (CRT)

The compiler performs limited type checking on functions that can take a variable number of arguments, as follows:

Function call	Type-checked arguments
_cprintf, **_cscanf**, **printf**, **scanf**	First argument (format string)
fprintf, **fscanf**, **sprintf**, **sscanf**	First two arguments (file or buffer and format string)
_snprintf	First three arguments (file or buffer, count, and format string)
_open	First two arguments (path and **_open** flag)
_sopen	First three arguments (path, **_open** flag, and sharing mode)
_execl, **_execle**, **_execlp**, **_execlpe**	First two arguments (path and first argument pointer)
_spawnl, **_spawnle**, **_spawnlp**, **_spawnlpe**	First three arguments (mode flag, path, and first argument pointer)

The compiler performs the same limited type checking on the wide-character counterparts of these functions.

See Also

Reference

C Run-Time Libraries

Direction Flag

The direction flag is a CPU flag specific to Intel 80x86 processors. It applies to all assembly instructions that use the REP (repeat) prefix, such as MOVS, MOVSD, MOVSW, and others. Addresses provided to applicable instructions are increased if the direction flag is cleared.

The C run-time routines assume that the direction flag is cleared. If you are using other functions with the C run-time functions, you must ensure that the other functions leave the direction flag alone or restore it to its original condition. Expecting the direction flag to be clear upon entry makes the run-time code faster and more efficient.

The C Run-Time library functions, such as the string-manipulation and buffer-manipulation routines, expect the direction flag to be clear.

See Also
Reference
C Run-Time Libraries

Security Enhancements in the CRT

Significant enhancements have been made to make the CRT more secure. Many CRT functions now have more secure versions. If a new secure function exists, the older, less secure version is marked as deprecated and the new version has the **_s** ("secure") suffix.

It should be noted that in this context, "deprecated" just means that a function's use is not recommended; it does not indicate that the function is scheduled to be removed from the CRT.

It should also be noted that the secure functions do not prevent or correct security errors; rather, they catch errors when they occur. They perform additional checks for error conditions, and in the case of an error, they invoke an error handler (see Parameter Validation).

For example, the **strcpy** function has no way of telling if the string that it is copying is too big for its destination buffer. However, its secure counterpart, **strcpy_s**, takes the size of the buffer as a parameter, so it can determine if a buffer overrun will occur. If you use **strcpy_s** to copy eleven characters into a ten-character buffer, that is an error on your part; **strcpy_s** cannot correct your mistake, but it can detect your error and inform you by invoking the invalid parameter handler.

Eliminating deprecation warnings

There are several ways to eliminate deprecation warnings for the older, less secure functions. The simplest is simply to define **_CRT_SECURE_NO_WARNINGS** or use the warning pragma. Either will disable deprecation warnings, but of course the security issues that caused the warnings still exist. It is far better to leave deprecation warnings enabled and take advantage of the new CRT security features.

In C++, the easiest way to do that is to use Secure Template Overloads, which in many cases will eliminate deprecation warnings by replacing calls to deprecated functions with calls to the new secure versions of those functions. For example, consider this deprecated call to **strcpy**:

```
char szBuf[10];
strcpy(szBuf, "test"); // warning: deprecated
```

Defining **_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES** as 1 eliminates the warning by changing the **strcpy** call to **strcpy_s**, which prevents buffer overruns. For more information, see Secure Template Overloads.

For those deprecated functions without secure template overloads, you should definitely consider manually updating your code to use the secure versions.

Another source of deprecation warnings, unrelated to security, is the POSIX functions. Replace POSIX function names with their standard equivalents (for example, change access to _access), or disable POSIX-related deprecation warnings by defining **_CRT_NONSTDC_NO_WARNINGS**. For more information, see Deprecated CRT Functions.

Security Enhancements

Some of the security enhancements include the following:

- **Parameter Validation**. Parameters passed to CRT functions are validated, in both secure functions and in many preexisting versions of functions. These validations include:
  - Checking for **NULL** values passed to the functions.
  - Checking enumerated values for validity.
  - Checking that integral values are in valid ranges.
- For more information, see Parameter Validation.
- A handler for invalid parameters is also accessible to the developer. When an encountering an invalid parameter, instead of asserting and exiting the application, the CRT provides a way to check these problems with the _set_invalid_parameter_handler function.
- **Sized Buffers**. The secure functions require that the buffer size be passed to any function that writes to a buffer. The secure versions validate that the buffer is large enough before writing to it, helping to avoid dangerous buffer overrun errors that could allow malicious code to execute. These functions usually return an **errno** type of error code and invoke the invalid parameter handler if the size of the buffer is too small. Functions that read from input buffers, such as **gets**,

have secure versions that require you to specify a maximum size.

- **Null termination**. Some functions that left potentially non-terminated strings have secure versions which ensure that strings are properly null terminated.
- **Enhanced error reporting**. The secure functions return error codes with more error information than was available with the preexisting functions. The secure functions and many of the preexisting functions now set **errno** and often return an **errno** code type as well, to provide better error reporting.
- **Filesystem security**. Secure file I/O APIs support secure file access in the default case.
- **Windows security**. Secure process APIs enforce security policies and allow ACLs to be specified.
- **Format string syntax checking**. Invalid strings are now detected, for example, using incorrect type field characters in **printf** format strings.

The documentation for each function describes additional security enhancements.

For more information, see the following articles on MSDN Online:

- Safe! Repel Attacks on Your Code with the Visual Studio 2005 Safe C and C++ Libraries by Martyn Lovell.
- Saying Goodbye to an Old Friend by Michael Howard.

See Also

Reference

Security-Enhanced Versions of CRT Functions
Secure Template Overloads
C Run-Time Libraries

Concepts

Parameter Validation
Deprecated CRT Functions

Security-Enhanced Versions of CRT Functions

More secure versions of run-time library routines are available in Visual C++ 2005. For further information concerning Security Enhancements in the CRT, see Security Enhancements in the CRT.

Secure Functions

Function	Use	.NET Framework equivalent
_access_s, _waccess_s	Determine file-access permission	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_malloca	Allocates memory on the stack	Not applicable

asctime_s, _wasctime_s	Convert time from type **struct tm** to character string	System::DateTime::ToLongDateString, System::DateTime::ToLongTimeString, System::DateTime::ToShortDateString, System::DateTime::ToShortTimeString, System::DateTime::ToString
bsearch_s	Performs a binary search of a sorted array	BinarySearch

_cgets_s, _cgetws_s	Get a character string from the console	Not applicable
_chsize_s	Changes the size of a file	• System::IO::Stream::SetLength, System::IO::FileStream::SetLength
clearerr_s	Resets the error indicator for a stream	• Not applicable

_controlfp_s	Get and set the floating-point control word	• Not applicable
_cprintf_s, _cprintf_s_l, _cwprintf_s, _cwprintf_s_l	Formats and prints to the console	• Not applicable
_cscanf_s, _cscanf_s_l, _cwscanf_s, _cwscanf_s_l	Reads formatted data from the console	• Not applicable

_ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s	Convert time from type **time_t**, **__time32_t** or **__time64_t** to character string	System::DateTime::GetDateTimeFormats, System::DateTime::ToString, System::DateTime::ToLongTimeString, System::DateTime::ToShortTimeString
_ecvt_s	Converts a **double** number to a string	System::Convert::ToString

_fcvt_s	Converts a floating-point number to a string	ToString
fopen_s, _wfopen_s	Open a file	• System::IO::File::Open, System::IO::FileStream::FileStream
fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l	Printformatted data to a stream	System::IO::StreamWriter::Write
freopen_s, _wfreopen_s	Reassign a file pointer	• System::IO::File::Open, System::IO::FileStream::FileStream

fscanf_s, _fscanf_s_l, fwscanf_s, _fwscanf_s_l	Read formatted data from a stream	• System::IO::StreamReader::ReadLine. See also **Parse** methods, such as System::Double::Parse
_ftime_s, _ftime32_s, _ftime64_s	Get the current time	System::DateTime::Now
_gcvt_s	Converts a floating-point value to a string, which it stores in a buffer	**ToString**

getenv_s, _wgetenv_s	Get a value from the current environment.	System::Environment::GetEnvironmentVariable
gets_s, _getws_s	Get a line from the **stdin** stream	System::Console::Read

_gmtime32_s, _gmtime64_s	Convert time from type **time_t** to **struct tm** or from type **__time64_t** to **struct tm**	System::DateTime::UtcNow, System::DateTime::ToUniversalTime
_itoa_s, _i64toa_s, _ui64toa_s, _itow_s, _i64tow_s, _ui64tow_s	Convert an integer to a string	System::Convert::ToString

_lfind_s	Performs a linear search for the specified key	System::Collections::ArrayList::Contains

localtime_s, _localtime32_s, _localtime64_s	Convert time from type **time_t** to **struct tm** or from type **__time64_t** to **struct tm** with local correction	System::DateTime::ToLocalTime

_lsearch_s	Performs a linear search for a value; adds to end of list if not found	Not applicable
_ltoa_s, _ltow_s	Convert a long integer to a string	System::Convert::ToString

_makepath_s, _wmakepath_s	Create a path name from components	System::IO::File::Create
_mbccpy_s, _mbccpy_s_l	Copies a multibyte character from one string to another string	Not applicable

_mbsnbcat_s, _mbsnbcat_s_l	Appends, at most, the first *n* bytes of one multibyte character string to another	Not applicable
_mbsnbcpy_s, _mbsnbcpy_s_l	Copies *n* bytes of a string to a destination string	Not applicable

mbsrtowcs_s	Converts a multibyte character string to a corresponding wide character string	Not applicable

mbstowcs_s, _mbstowcs_s_l	Converts a sequence of multibyte characters to a corresponding sequence of wide characters	Not applicable
memcpy_s, wmemcpy_s	Copies characters between buffers	Not applicable

memmove_s, wmemmove_s	Moves one buffer to another	Not applicable
_mktemp_s, _wmktemp_s	Create a unique filename	Not applicable
printf_s, _printf_s_l, wprintf_s, _wprintf_s_l	Printformatted output to the standard output stream	• System::Console::Write, System::Console::WriteLine

_putenv_s, _wputenv_s	Create, modify, or remove environment variables	• Not applicable
qsort_s	Performs a quick sort	System::Collections::ArrayList::Sort
rand_s	Generates a pseudorandom number	System::Random Class •

scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l	Read formatted data from the standard input stream	System::Console::Read, System::Console::ReadLine, see also **Parse** methods, such as System::Double::Parse
_searchenv_s, _wsearchenv_s	Search for a file using environment paths	Not applicable
_snprintf_s, _snprintf_s_l, _snwprintf_s, _snwprintf_s_l	Write formatted data to a string	Not applicable

_snscanf_s, _snscanf_s_l, _snwscanf_s, _snwscanf_s_l	Read formatted data of a specified length from a string.	Not applicable
_sopen_s, _wsopen_s	Open a file for sharing	Not applicable
_splitpath_s, _wsplitpath_s	Break a pathname into components	Not applicable

sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l	Write formatted data to a string	System::String::Format
sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l	Read formatted data from a string	See **Parse** methods, such as System::Double::Parse
strcat_s, wcscat_s, _mbscat_s	Append a string	System::String::Concat
strcpy_s, wcscpy_s, _mbscpy_s	Copy a string	System::String::Copy

_strdate_s, _wstrdate_s	Return current system date as string	System::DateTime::Parse

strerror_s, _strerror_s, _wcserror_s, __wcserror_s	Get a system error message (**strerror**, _**wcserror**) or prints a user-supplied error message (_**strerror**, __**wcserror**)	System::Exception::Message

_strlwr_s, _strlwr_s_l, _mbslwr_s, _mbslwr_s_l, _wcslwr_s, _wcslwr_s_l	Convert a string to lowercase	System::String::ToLower
strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l	Append characters to a string	System::String::Concat
strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l	Copy characters of one string to another	System::String::Copy

_strtime_s, _wstrtime_s	Return current system time as string	System::DateTime::ToLongDateString, System::DateTime::ToLongTimeString, System::DateTime::ToShortDateString, System::DateTime::ToShortTimeString, System::DateTime::ToString
strtok_s, _strtok_s_l, wcstok_s, _wcstok_s_l, _mbstok_s, _mbstok_s_l	Find the next token in a string, using the current locale or a locale passed in	Not applicable

_strupr_s, _strupr_s_l, _mbsupr_s, _mbsupr_s_l, _wcsupr_s, _wcsupr_s_l	Convert a string to uppercase	System::String::ToUpper
tmpfile_s	Creates a temporary file	Not applicable
tmpnam_s, _wtmpnam_s	Generate names you can use to create temporary files	Not applicable

_ultoa_s, _ultow_s	Convert an unsigned long integer to a string	System::Convert::ToString
_umask_s	Sets the default file-permission mask	System::IO::File::SetAttributes

_vcprintf_s, _vcprintf_s_l, _vcwprintf_s, _vcwprintf_s_l	Write formatted output to the console using a pointer to a list of arguments	System::Console::Write

vfprintf_s, _vfprintf_s_l, vfwprintf_s, _vfwprintf_s_l	Write formatted output using a pointer to a list of arguments	Not applicable
vprintf_s, _vprintf_s_l, vwprintf_s, _vwprintf_s_l	Write formatted output using a pointer to a list of arguments	System::Console::Write

vsnprintf_s, _vsnprintf_s, _vsnprintf_s_l, _vsnwprintf_s, _vsnwprintf_s_l	Write formatted output using a pointer to a list of arguments	Not applicable
vsprintf_s, _vsprintf_s_l, vswprintf_s, _vswprintf_s_l	Write formatted output using a pointer to a list of arguments	System::String::Format

wcrtomb_s	Convert a wide character into its multibyte character representation	Not applicable

wcsrtombs_s	Convert a wide character string to its multibyte character string representation	Not applicable

wcstombs_s, _wcstombs_s_l	Converts a sequence of wide characters to a corresponding sequence of multibyte characters	Not applicable

wctomb_s, _wctomb_s_l	Converts a wide character to the corresponding multibyte character	Not applicable

See Also

Reference

C Run-Time Libraries

Parameter Validation

Most of the security-enhanced CRT functions and many of the preexisting functions validate their parameters. This could include checking pointers for NULL, checking that integers fall into a valid range, or checking that enumeration values are valid. When an invalid parameter is found, the invalid parameter handler is executed.

Invalid Parameter Handler Routine

The behavior of the C Runtime when an invalid parameter is found is to call the currently assigned invalid parameter handler. The default invalid parameter invokes Watson crash reporting, which causes the application to crash and asks the user if they want to load the crash dump to Microsoft for analysis. In Debug mode, an invalid parameter also results in a failed assertion.

This behavior can be changed by using the function _set_invalid_parameter_handler to set the invalid parameter handler to your own function. If the function you specify does not terminate the application, control is returned to the function that received the invalid parameters, and these functions will normally cease execution, return an error code, and set **errno** to an error code. In many cases, the **errno** value and the return value are both **EINVAL**, indicating an invalid parameter. In some cases, a more specific error code is returned, such as **EBADF** for a bad file pointer passed in as a parameter. For more information on errno, see errno, _doserrno, _sys_errlist, and _sys_nerr.

See Also

Reference

C Run-Time Libraries

Concepts

Security Enhancements in the CRT

Secure Template Overloads

Many CRT functions have been deprecated in favor of newer, security-enhanced versions (for example, strcpy_s is the more secure replacement for strcpy). The CRT provides template overloads to help ease the transition to the more secure variants.

For example, this code generates a warning because strcpy is deprecated:

```
char szBuf[10];

strcpy(szBuf, "test"); // warning: deprecated
```

You can ignore the warning, define the symbol _CRT_SECURE_NO_WARNINGS to suppress the warning, or update the code to use strcpy_s:

```
char szBuf[10];

strcpy_s(szBuf, 10, "test"); // security-enhanced _s function
```

The template overloads provide additional choices. Defining _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES to be 1 enables template overloads of standard CRT functions that call the more secure variants automatically. If _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES is 1, then no changes to the code are necessary. Behind the scenes, the call to strcpy will be changed to a call to strcpy_s with the size argument supplied automatically.

```
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1

...

char szBuf[10];

strcpy(szBuf, "test"); // ==> strcpy_s(szBuf, 10, "test")
```

_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES does not affect the functions that take a count, such as strncpy. To enable template overloads for the count functions, define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES_COUNT to be 1. Before doing so, however, make sure that your code passes the count of characters, not the size of the buffer (a common mistake). Also, code that explicitly writes a null terminator at the end of the buffer after the function call is unnecessary if the secure variant is called. If you need truncation behavior, see _TRUNCATE.

Defining _CRT_SECURE_CPP_OVERLOAD_SECURE_NAMES to be 1 enables template overloads of the secure variants (names ending in "_s"). In this case, if _CRT_SECURE_CPP_OVERLOAD_SECURE_NAMES is 1, then one small change must be made to the original code:

```
#define _CRT_SECURE_CPP_OVERLOAD_SECURE_NAMES 1

...

char szBuf[10];

strcpy_s(szBuf, "test"); // ==> strcpy_s(szBuf, 10, "test")
```

Only the name of the function needs to be changed (by adding "_s"); the template overload will take care of providing the size argument.

By default, _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES and _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES_COUNT are defined as 0 (disabled) and _CRT_SECURE_CPP_OVERLOAD_SECURE_NAMES is defined as 1 (enabled).

Note that these template overloads only work for static arrays. Dynamically allocated buffers require additional source code changes. Revisiting the above examples:

```
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1

...

char *szBuf = (char*)malloc(10);

strcpy(szBuf, "test"); // still deprecated; have to change to

// strcpy_s(szBuf, 10, "test");
```

And this:

```
#define _CRT_SECURE_CPP_OVERLOAD_SECURE_NAMES 1

...

char *szBuf = (char*)malloc(10);

strcpy_s(szBuf, "test"); // doesn't compile; have to change to
```

```
// strcpy_s(szBuf, 10, "test");
```

See Also

Reference

C Run-Time Libraries

Concepts

Security Enhancements in the CRT

SAL Annotations

If you examine the library header files, you will notice some unusual annotations such as **__in_z** and **__out_ecount_part**. These are examples of Microsoft's standard source code annotation language (SAL), which provides a set of annotations to describe how a function uses its parameters—the assumptions it makes about them, and the guarantees it makes upon finishing. The header file <sal.h> defines the annotations.

Annotations may be placed before either a function parameter's type or its return type, and describe the function's behavior regarding the parameter or return value. There are two classes of annotations: buffer annotations and advanced annotations. Buffer annotations describe how functions use their pointer parameters, and advanced annotations either describe complex/unusual buffer behavior, or provide additional information about a parameter that is not otherwise expressible.

Buffer Annotations

The most important annotations provide a consistent way to annotate buffer parameters or return values for a function. Each of these annotations describes a single buffer (which could be a string, a fixed-length or variable-length array, or just a pointer) that the function interacts with—where it is, how large it is, how much is initialized, and what the function does with it.

The appropriate macro for a given buffer can be constructed using the table below. Pick the appropriate values from each category, and combine them together with a leading underscore. Some combinations of values do not make sense as buffer annotations. Only meaningful annotations can be added to your code; for a list of these, see the buffer annotation definitions section in <sal.h>.

Only a single buffer annotation should be used for each parameter.

Category	**Possible choices**
Level	(none) **_deref** **_deref_opt**
Usage	(none) **_in** **_out** **_inout**
Size	(none) **_ecount** **_bcount**
Output	(none) **_full** **_part**
Null-termination	**_z** **_nz**
Optional	(none) **_opt**

Parameters	(none) **(** *size* **)** **(** *size* **,** *length* **)**

The various categories and their respective choices are described below.

Level

Describes the buffer pointer's level of indirection from the parameter or return value *p*.

(none)	*p* is the buffer pointer.
_deref	**p* is the buffer pointer. *p* must not be NULL.
_deref_opt	**p* may be the buffer pointer. *p* may be NULL, in which case the rest of the annotation is ignored.

Usage

Describes how the function uses the buffer.

(none)	The buffer is not accessed. If used on the return value or with **_deref**, the function will provide the buffer, and it will be uninitialized at exit. Otherwise, the caller must provide the buffer. This should only be used for alloc and free functions.
_in	The function will only read from the buffer. The caller must provide the buffer and initialize it. Cannot be used with **_deref**.
_out	The function will only write to the buffer. If used on the return value or with **_deref**, the function will provide the buffer and initialize it. Otherwise, the caller must provide the buffer, and the function will initialize it.
_inout	The function may freely read from and write to the buffer. The caller must provide the buffer and initialize it. If used with **_deref**, the buffer may be reallocated by the function.

Size

Describes the total size of the buffer. This may be less than the space actually allocated for the buffer, in which case it describes the accessible amount.

(none)	No buffer size is given. If the type specifies the buffer size (such as with LPSTR and LPWSTR), that amount is used. Otherwise, the buffer is one element long. Must be used with **_in**, **_out**, or **_inout**.
_ecount	The buffer size is an explicit element count.
_bcount	The buffer size is an explicit byte count.

Output

Describes how much of the buffer will be initialized by the function. For **_inout** buffers, this also describes how much is initialized at entry. Omit this category for **_in** buffers; they must be fully initialized by the caller.

(none)	The type specifies how much is initialized. For instance, a function initializing an LPWSTR must null-terminate the string.
_full	The function initializes the entire buffer.
_part	The function initializes part of the buffer, and explicitly indicates how much.

Null-Termination

States if the presence of a '\0' marks the end of valid elements in the buffer.

_z	A '\0' indicates the end of the buffer.
_nz	The buffer may not be null-terminated and a '\0' does not indicate the end of the buffer.

Optional

Describes if the buffer itself is optional.

(none)	The pointer to the buffer must not be NULL.
_opt	The pointer to the buffer might be NULL. It will be checked before being dereferenced.

Parameters

Gives explicit counts for the size and length of the buffer.

(none)	There is no explicit count. Use when neither **_ecount** nor **_bcount** is used.
(*size* **)**	Only the buffer's total size is given. Use with **_ecount** or **_bcount** but not **_part**.
(*size* **,** *length* **)**	The buffer's total size and initialized length are given. Use with **_ecount_part** and **_bcount_part**.

Example

```
// Buffer Annotation Examples

LWSTDAPI_(BOOL) StrToIntExA(
    LPCSTR pszString,    // No annotation required, const implies __in.
    DWORD dwFlags,
    __out int *piRet     // A pointer whose dereference will be filled
);

void MyPaintingFunction(
    __in HWND hwndControl,      // An initialized read-only parameter.
    __in_opt HDC hdcOptional,   // An initialized read-only parameter
                                //  that might be NULL.
    __inout IPropertyStore *ppsStore  // An initialized parameter that
                                // may be freeely used and modified.
);

LWSTDAPI_(BOOL) PathCompactPathExA(
    __out_ecount(cchMax) LPSTR pszOut,   // A string buffer with cchMax
                       // elements that will be null-terminated on exit.
    LPCSTR pszSrc,                       // No annotation required,
                                         //  const implies __in.
    UINT cchMax,
    DWORD dwFlags
);

HRESULT SHLocalAllocBytes(
    size_t cb,
    __deref_bcount(cb) T **ppv  // A pointer whose dereference will be
                          // set to uninitialized buffer with cb bytes.
);

__inout_bcount_full(cb) // A buffer with cb elements that is fully
                        //  initialized at entry and exit, and may be
                        //  written to by this function.

__out_ecount_part(count, *countOut) // A buffer with count elements
                       // that will be partially initialized by this
                       //  function. The function indicates how much it
                       //  initialized by setting *countOut.
```

Advanced Annotations

Advanced annotations describe behavior that is not expressible with the regular buffer macros. These may be used either to annotate buffer parameters that involve complex or conditional behavior, or to enrich existing annotations with additional information.

__success(*expr* **)** *f*	*expr* indicates whether function *f* succeeded or not. If *expr* is true at exit, all the function's guarantees (as given by other annotations) must hold. If *expr* is false at exit, the caller should not expect any of the function's guarantees to hold. If not used, the function must always satisfy its guarantees. Added automatically to functions that indicate success in standard ways, such as by returning an HRESULT.
__nullterminated *p*	Pointer *p* is a buffer that may be read or written up to and including the first null character or pointer. May be used on typedefs that mark valid (properly initialized) instances of that type as being null-terminated.
__nullnullterminated *p*	Pointer *p* is a buffer that may be read or written up to and including the first sequence of two null characters or pointers. May be used on typedefs that mark valid instances of that type as being double-null-terminated.
__reserved *v*	Value *v* must be 0 or NULL. Reserved for future use.
__checkReturn *v*	Return value *v* must not be ignored by callers of this function.
__typefix(*ctype* **)** *v*	Value *v* should be treated as an instance of *ctype*, rather than its declared type.
__override *f*	Specify C#-style 'override' behavior for overriding virtual methods.
__callback *f*	Function *f* can be used as a function pointer.
__format_string *p*	Pointer *p* is a string that contains % markers in the style of printf.
__blocksOn(*resource* **)** *f*	Function *f* blocks on the resource *resource*.
__fallthrough	Annotates **switch** statement labels where fall-through is desired, to distinguish from forgotten **break** statements.

Example

```
// Advanced Annotation Examples

__success(return == TRUE) LWSTDAPI_(BOOL)
PathCanonicalizeA(__out_ecount(MAX_PATH) LPSTR pszBuf, LPCSTR pszPath)
// pszBuf is only guaranteed to be null-terminated when TRUE is returned.
```

```
typedef __nullterminated WCHAR* LPWSTR
// Initialized LPWSTRs are null-terminated strings.

__out_ecount(cch) __typefix(LPWSTR) void *psz
// psz is a buffer parameter which will be a null-terminated WCHAR
//  string at exit, and which initially contains cch WCHARs.
```

See Also

Reference

C Run-Time Libraries

Multithreaded Libraries Performance

The single-threaded CRT is no longer available. This topic discusses how to get the maximum performance from the multithreaded libraries.

Maximizing performance

The performance of the multithreaded libraries has been improved and is close to the performance of the now-eliminated single-threaded libraries. For those situations when even higher performance is required, there are several new features.

- Independent stream locking allows you to lock a stream and then use _nolock Functions that access the stream directly. This allows lock usage to be hoisted outside critical loops.
- Per-thread locale reduces the cost of locale access for multithreaded scenarios (see _configthreadlocale).
- Locale-dependent functions (with names ending in _l) take the locale as a parameter, removing substantial cost (for example, printf, _printf_l, wprintf, _wprintf_l).
- Optimizations for common codepages reduce the cost of many short operations.
- Defining _CRT_DISABLE_PERFCRIT_LOCKS forces all I/O operations to assume a single-threaded I/O model and use the _nolock forms of the functions. This allows highly I/O-based single-threaded applications to get better performance.
- Exposure of the CRT heap handle allows you to enable the Windows Low Fragmentation Heap (LFH) for the CRT heap, which can substantially improve performance in highly scaled scenarios.

See Also

Reference

C Run-Time Libraries

Link Options

The CRT lib directory includes a number of small object files that enable specific CRT features without any code change. These are called "link options" since you just have to add them to the linker command line to use them.

Beginning in Visual C++ 2005, pure mode versions have been added. Use the regular versions for native and /clr code, use the pure versions (prefixed with a p) for /clr:pure mode.

Native and /clr	Pure mode	Description
binmode.obj	pbinmode.obj	Sets the default file-translation mode to binary. See _fmode.
chkstk.obj	n/a	Provides stack-checking and alloca support when not using the CRT.
commode.obj	pcommode.obj	Sets the global commit flag to "commit". See fopen, _wfopen and fopen_s, _wfopen_s.
fp10.obj	n/a	Changes the default precision control to 64 bits. See Floating-Point Support.
invalidcontinue.obj	pinvalidcontinue.obj	Sets a default invalid parameter handler that does nothing, meaning that invalid parameters passed to CRT functions will just set errno and return an error result.
loosefpmath.obj	n/a	Ensures that floating point code tolerates denormal values.
newmode.obj	pnewmode.obj	Causes malloc to call the new handler on failure. See _set_new_mode, _set_new_handler, calloc, and realloc.
noarg.obj	pnoarg.obj	Disables all processing of argc and argv.
nochkclr.obj	n/a	Does nothing. Remove from your project.
noenv.obj	pnoenv.obj	Disables the creation of a cached environment for the CRT.
nothrownew.obj	pnothrownew.obj	Enables the non-throwing version of new in the CRT. See The new and delete Operators.
setargv.obj	psetargv.obj	Enables command-line argument wildcard expansion. See Expanding Wildcard Arguments.
smalheap.obj	n/a	Installs a very simple small heap manager.
threadlocale.obj	pthreadlocale.obj	Enables per-thread locale for all new threads by default.
wsetargv.obj	pwsetargv.obj	Enables command-line argument wildcard expansion. See Expanding Wildcard Arguments.

See Also

Reference

C Run-Time Libraries

Potential Errors Passing CRT Objects Across DLL Boundaries

When you pass C Run-time (CRT) objects such as file handles, locales, and environment variables into or out of a DLL (function calls across the DLL boundary), unexpected behavior can occur if the DLL, as well as the files calling into the DLL, use different copies of the CRT libraries.

A related problem can occur when you allocate memory (either explicitly with **new** or **malloc**, or implicitly with **strdup**, **strstreambuf::str**, and so on) and then pass a pointer across a DLL boundary to be freed. This can cause a memory access violation or heap corruption if the DLL and its users use different copies of the CRT libraries.

Another symptom of this problem can be an error in the output window during debugging such as:

HEAP[]: Invalid Address specified to RtlValidateHeap(#,#)

Causes

Each copy of the CRT library has a separate and distinct state. As such, CRT objects such as file handles, environment variables, and locales are only valid for the copy of the CRT where these objects are allocated or set. When a DLL and its users use different copies of the CRT library, you cannot pass these CRT objects across the DLL boundary and expect them to be picked up correctly on the other side.

Also, because each copy of the CRT library has its own heap manager, allocating memory in one CRT library and passing the pointer across a DLL boundary to be freed by a different copy of the CRT library is a potential cause for heap corruption.

If you design your DLL so that it passes CRT objects across the boundary or allocates memory and expects it to be freed outside the DLL, you restrict the DLL users to use the same copy of the CRT library as the DLL. The DLL and its users use the same copy of the CRT library only if both are linked with the same version of the CRT DLL. This could be a problem if you mix applications built with Visual C++ 5.0 with DLLs that are built by Visual C++ 4.1 or earlier. Because the DLL version of the CRT library used by Visual C++ 4.1 is msvcrt40.dll and the one used by Visual 5.0 is msvcrt.dll, you cannot build your application to use the same copy of the CRT library as these DLLs.

However, there is an exception. In US English version and some other localized versions of Windows NT 4.0 and Windows 2000, such as German, French, and Czech, a forwarder version of the msvcrt40.dll (version 4.20)is shipped. As a result, even though the DLL is linked with msvcrt40.dll and its user is linked with msvcrt.dll, you are still using the same copy of the CRT library because all calls made to msvcrt40.dll are forwarded to msvcrt.dll.

However this forwarder version of msvcrt40.dll is not available in Windows 95, Windows 98, Windows Millennium Edition (Me), and some localized versions of Windows NT 4.0 and Windows 2000, such as Japanese, Korean, and Chinese. So, if your application targets these operating systems, you need to either obtain an upgraded version of the DLL that doesn't rely on msvcrt40.dll or alter your application to not rely on using the same copy of the CRT libraries. If you have developed the DLL, this means rebuilding it with Visual C++ 4.2 or later. If it is a third- party DLL, you need to contact your vendor for an upgrade.

Please note that this forwarder DLL version of msvcrt40.dll (version 4.20) cannot be redistributed.

Example

This example passes a file handle across a DLL boundary.

The DLL and .exe file are built with /MD, so they share a single copy of the CRT.

If you rebuild with /MT so that they use separate copies of the CRT, running the resulting test1Main.exe results in an access violation.

```
// test1Dll.cpp
// compile with: /MD /LD
#include <stdio.h>
__declspec(dllexport) void writeFile(FILE *stream)
{
   char   s[] = "this is a string\n";
   fprintf( stream, "%s", s );
   fclose( stream );
}
```

```
// test1Main.cpp
// compile with: /MD test1dll.lib
```

```
#include <stdio.h>
#include <process.h>
void writeFile(FILE *stream);

int main(void)
{
   FILE  * stream;
   errno_t err = fopen_s( &stream, "fprintf.out", "w" );
   writeFile(stream);
   system( "type fprintf.out" );
}
```

Output

```
this is a string
```

Example

This example passes environment variables across a DLL boundary.

```
// test2Dll.cpp
// compile with: /MT /LD
#include <stdio.h>
#include <stdlib.h>

__declspec(dllexport) void readEnv()
{
   char *libvar;
   size_t libvarsize;

   /* Get the value of the MYLIB environment variable. */
   _dupenv_s( &libvar, &libvarsize, "MYLIB" );

   if( libvar != NULL )
      printf( "New MYLIB variable is: %s\n", libvar);
   else
      printf( "MYLIB has not been set.\n");
   free( libvar );
}
```

```
// test2Main.cpp
// compile with: /MT /link test2dll.lib
#include <stdlib.h>
#include <stdio.h>

void readEnv();

int main( void )
{
   _putenv( "MYLIB=c:\\mylib;c:\\yourlib" );
   readEnv();
}
```

Output

```
MYLIB has not been set.
```

If both the DLL and .exe file are built with /MD so that only one copy of the CRT is used, the program runs successfully and produces the following output:

```
New MYLIB variable is: c:\mylib;c:\yourlib
```

See Also

Reference

C Run-Time Libraries

Run-Time Routines by Category

This section lists and describes Microsoft run-time library routines by category. For reference convenience, some routines are listed in more than one category. Multibyte-character routines and wide-character routines are grouped with single-byte – character counterparts, where they exist.

This section also provides .NET Framework equivalent methods or classes where available.

The main categories of Microsoft run-time library routines are:

Argument access	Floating-point support
Buffer manipulation	Input and output
Byte classification	Internationalization
Character classification	Memory allocation
Data alignment	Process and environment control
Data conversion	Robustness
Debug	Run-time error checking
Directory control	Searching and sorting
Error handling	String manipulation
Exception handling	System calls
File handling	Time management

See Also

Other Resources

Run-Time Library Reference

Argument Access

The **va_arg**, **va_end**, and **va_start** macros provide access to function arguments when the number of arguments is variable. These macros are defined in STDARG.H for ANSI C compatibility and in VARARGS.H for compatibility with UNIX System V.

Argument-Access Macros

Macro	Use	.NET Framework equivalent
va_arg	Retrieve argument from list	System::ParamArrayAttribute Class
va_end	Reset pointer	System::ParamArrayAttribute Class
va_start	Set pointer to beginning of argument list	System::ParamArrayAttribute Class

See Also

Reference

Run-Time Routines by Category

Buffer Manipulation

Use these routines to work with areas of memory on a byte-by-byte basis.

Buffer-Manipulation Routines

Routine	Use	.NET Framework equivalent
_memccpy	Copy characters from one buffer to another until given character or given number of characters has been copied	System::Buffer::BlockCopy, System::String::Copy
memchr, wmemchr	Return pointer to first occurrence, within specified number of characters, of given character in buffer	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
memcmp, wmemcmp	Compare specified number of characters from two buffers	System::String::Compare, System::String::Equals
memcpy, wmemcpy, memcpy_s, wmemcpy_s	Copy specified number of characters from one buffer to another	System::Buffer::BlockCopy, System::String::Copy
_memicmp, _memicmp_l	Compare specified number of characters from two buffers without regard to case	System::String::Compare, System::String::Equals
memmove, wmemmove, memmove_s, wmemmove_s	Copy specified number of characters from one buffer to another	System::Buffer::BlockCopy
memset, wmemset	Use given character to initialize specified number of bytes in the buffer	System::Buffer::SetByte
_swab	Swap bytes of data and store them at specified location	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

When the source and target areas overlap, only **memmove** is guaranteed to copy the full source properly.

See Also

Reference

Run-Time Routines by Category

Byte Classification

Each of these routines tests a specified byte of a multibyte character for satisfaction of a condition. Except where specified otherwise, the output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Note
By definition, the ASCII characters between 0 and 127 are a subset of all multibyte-character sets. For example, the Japanese katakana character set includes ASCII as well as non-ASCII characters.

The predefined constants in the following table are defined in CTYPE.H.

Multibyte-Character Byte-Classification Routines

Routine	Byte Test Condition	.NET Framework equivalent
isleadbyte, _isleadbyte_l	Lead byte; test result depends on **LC_CTYPE** category setting of current locale	Not applicable, but see System::Globalization::CultureInfo
_ismbbalnum, _ismbbalnum_l	**isalnum || _ismbbkalnum**	Not applicable, but see System::Globalization::CultureInfo
_ismbbalpha, _ismbbalpha_l	**isalpha || _ismbbkalnum**	Not applicable, but see System::Globalization::CultureInfo
_ismbbgraph, _ismbbgraph_l	Same as **_ismbbprint**, but **_ismbbgraph** does not include the space character (0x20)	Not applicable, but see System::Globalization::CultureInfo
_ismbbkalnum, _ismbbkalnum_l	Non-ASCII text symbol other than punctuation. For example, in code page 932 only, **_ismbbkalnum** tests for katakana alphanumeric	Not applicable, but see System::Globalization::CultureInfo
_ismbbkana, _ismbbkana_l	Katakana (0xA1 – 0xDF), code page 932 only	Not applicable, but see System::Globalization::CultureInfo
_ismbbkprint, _ismbbkprint_l	Non-ASCII text or non-ASCII punctuation symbol. For example, in code page 932 only, **_ismbbkprint** tests for katakana alphanumeric or katakana punctuation (range: 0xA1 – 0xDF).	Not applicable, but see System::Globalization::CultureInfo
_ismbbkpunct, _ismbbkpunct_l	Non-ASCII punctuation. For example, in code page 932 only, **_ismbbkpunct** tests for katakana punctuation.	Not applicable, but see System::Globalization::CultureInfo
_ismbblead, _ismbblead_l	First byte of multibyte character. For example, in code page 932 only, valid ranges are 0x81 – 0x9F, 0xE0 – 0xFC.	Not applicable, but see System::Globalization::CultureInfo
_ismbbprint, _ismbbprint_l	**isprint || _ismbbkprint. ismbbprint** includes the space character (0x20)	Not applicable, but see System::Globalization::CultureInfo
_ismbbpunct, _ismbbpunct_l	**ispunct || _ismbbkpunct**	Not applicable, but see System::Globalization::CultureInfo
_ismbbtrail, _ismbbtrail_l	Second byte of multibyte character. For example, in code page 932 only, valid ranges are 0x40 – 0x7E, 0x80 – 0xEC.	Not applicable, but see System::Globalization::CultureInfo

_ismbslead, _ismbslead_l	Lead byte (in string context)	Not applicable, but see System::Globalization::CultureInfo
ismbstrail, _ismbstrail_l	Trail byte (in string context)	Not applicable, but see System::Globalization::CultureInfo
_mbbtype, _mbbtype_l	Return byte type based on previous byte	Not applicable, but see System::Globalization::CultureInfo
_mbsbtype, _mbsbtype_l	Return type of byte within string	Not applicable, but see System::Globalization::CultureInfo
mbsinit	Tracks the state of a multibyte character conversion.	Not applicable, but see System::Globalization::CultureInfo

The **MB_LEN_MAX** macro, defined in LIMITS.H, expands to the maximum length in bytes that any multibyte character can have. **MB_CUR_MAX**, defined in STDLIB.H, expands to the maximum length in bytes of any multibyte character in the current locale.

See Also
Reference
Run-Time Routines by Category

Character Classification

Each of these routines tests a specified single-byte character, wide character, or multibyte character for satisfaction of a condition. (By definition, the ASCII character set between 0 and 127 are a subset of all multibyte-character sets. For example, Japanese katakana includes ASCII as well as non-ASCII characters.)

The test conditions are affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Generally these routines execute faster than tests you might write and should be favored over. For example, the following code executes slower than a call to **isalpha(c)**:

```
if ((c >= 'A') && (c <= 'Z')) || ((c >= 'a') && (c <= 'z'))
    return TRUE;
```

Character-Classification Routines

Routine	Character test condition	.NET Framework equivalent
isalnum, iswalnum, _isalnum_l, _iswalnum_l, _ismbcalnum, _ismbcalnum_l, _ismbcalpha, _ismbcalpha_, _ismbcdigit, _ismbcdigit_l	Alphanumeric	System::Char::IsLetterOrDigit.
_ismbcalnum, _ismbcalnum_l, _ismbcalpha, _ismbcalpha_, _ismbcdigit, _ismbcdigit_l	Alphanumeric	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
isalpha, iswalpha, _isalpha_l, _iswalpha_l, _ismbcalnum, _ismbcalnum_l, _ismbcalpha, _ismbcalpha_, _ismbcdigit, _ismbcdigit_l	Alphabetic	System::Char::IsLetter
__isascii, iswascii	ASCII	System::Char::IsLetter

iscntrl, iswcntrl, _iscntrl_l, _iswcntrl_l	Control	System::Char::IsControl
__iscsym, __iswcsym, __iscsymf, __iswcsymf, _iscsym_l, _iswcsym_l, _iscsymf_l, _iswcsymf_l	Letter, underscore, or digit	System::Char::IsControl
__iscsym, __iswcsym, __iscsymf, __iswcsymf, _iscsym_l, _iswcsym_l, _iscsymf_l, _iswcsymf_l	Letter or underscore	System::Char::IsControl
isdigit, iswdigit, _isdigit_l, _iswdigit_l, _ismbcalnum, _ismbcalnum_l, _ismbcalpha, _ismbcalpha_, _ismbcdigit, _ismbcdigit_l	Decimal digit	System::Char::IsDigit
isgraph, iswgraph, _isgraph_l, _iswgraph_l, _ismbcgraph, _ismbcgraph_l, _ismbcprint, _ismbcprint_l, _ismbcpunct, _ismbcpunct_l, _ismbcspace, _ismbcspace_l	Printable other than space	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
islower, iswlower, _islower_l, _iswlower_l, _ismbclower, _ismbclower_l, _ismbcupper, _ismbcupper_l	Lowercase	System::Char::IsLower

_ismbchira, _ismbchira_l, _ismbckata, _ismbckata_l	Hiragana	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_ismbchira, _ismbchira_l, _ismbckata, _ismbckata_l	Katakana	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_ismbclegal, _ismbclegal_l, _ismbcsymbol, _ismbcsymbol_l	Legal multibyte character	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_ismbcl0, _ismbcl0_l, _ismbcl1, _ismbcl1_l, _ismbcl2, _ismbcl2_l	Japan-level 0 multibyte character	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

_ismbcl0, _ismbcl0_l, _ismbcl1, _ismbcl1_l, _ismbcl2, _ismbcl2_l	Japan-level1 multibyte character	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_ismbcl0, _ismbcl0_l, _ismbcl1, _ismbcl1_l, _ismbcl2, _ismbcl2_l	Japan-level2 multibyte character	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_ismbclegal, _ismbclegal_l, _ismbcsymbol, _ismbcsymbol_l	Non-alphanumeric multibyte character	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

isprint, iswprint, _isprint_l, _iswprint_l, _ismbcgraph, _ismbcgraph_l, _ismbcprint, _ismbcprint_l, _ismbcpunct, _ismbcpunct_l, _ismbcspace, _ismbcspace_l	Printable	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
ispunct, iswpunct, _ispunct_l, _iswpunct_l, _ismbcgraph, _ismbcgraph_l, _ismbcprint, _ismbcprint_l, _ismbcpunct, _ismbcpunct_l, _ismbcspace, _ismbcspace_l	Punctuation	System::Char::IsPunctuation
isspace, iswspace, _isspace_l, _iswspace_l, _ismbcgraph, _ismbcgraph_l, _ismbcprint, _ismbcprint_l, _ismbcpunct, _ismbcpunct_l, _ismbcspace, _ismbcspace_l	White-space	System::Char::IsWhiteSpace
Isupper, iswupper, _ismbclower, _ismbclower_l, _ismbcupper, _ismbcupper_l	Uppercase	System::Char::IsUpper
_isctype, iswctype, _isctype_l, _iswctype_l	Property specified by **desc**argument	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
isxdigit, iswxdigit, _isxdigit_l, _iswxdigit_l	Hexadecimaldigit	System::Char::IsNumber

_mbclen, mblen, _mblen_l	Return length of valid multibyte character; result depends on **LC_CTYPE** category setting of current locale	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Run-Time Routines by Category

Data Alignment

The following C run-time functions support data alignment.

Data-Alignment Routines

Routine	Use	.NET Framework equivalent
_aligned_free	Frees a block of memory that was allocated with _aligned_malloc or _aligned_offset_malloc.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_aligned_malloc	Allocates memory on a specified alignment boundary.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_aligned_offset_malloc	Allocates memory on a specified alignment boundary.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_aligned_offset_realloc	Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_aligned_realloc	Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Run-Time Routines by Category

Data Conversion

These routines convert data from one form to another. Generally these routines execute faster than conversions you might write. Each routine that begins with a **to** prefix is implemented as a function and as a macro. See Choosing Between Functions and Macros for information about choosing an implementation.

Data-Conversion Routines

Routine	Use	.NET Framework equivalent
abs	Find absolute value of integer	System::Math::Abs
atof, _atof_l, _wtof, _wtof_l	Convert string to **float**	System::Convert::ToDouble
atoi, _atoi_l, _wtoi, _wtoi_l	Convert string to **int**	System::Convert::ToInt32, System::Convert::ToUInt32
_atoi64, _atoi64_l, _wtoi64, _wtoi64_l	Convert string to **__int64**	System::Convert::ToInt64, System::Convert::ToUInt64
atol, _atol_l, _wtol, _wtol_l	Convert string to **long**	System::Convert::ToInt64, System::Convert::ToUInt64
_ecvt, _ecvt_s	Convert **double** to string of specified length	System::Convert::ToString
_fcvt, _fcvt_s	Convert **double** to string with specified number of digits following decimal point	System::Convert::ToString
_gcvt, _gcvt_s	Convert **double** number to string; store string in buffer	System::Convert::ToString
_itoa, _i64toa, _ui64toa, _itow, _i64tow, _ui64tow, _itoa_s, _i64toa_s, _ui64toa_s, _itow_s, _i64tow_s, _ui64tow_s	Convert **int** or **__int64** to string	System::Convert::ToString
labs	Find absolute value of **long** integer	System::Math::Abs
_ltoa, _ltow, _ltoa_s, _ltow_s	Convert **long** to string	System::Convert::ToString
_mbbtombc, _mbbtombc_l	Convert 1-byte multibyte character to corresponding 2-byte multibyte character	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_mbcjistojms, _mbcjistojms_l, _mbcjmstojis, _mbcjmstojis_l	Convert Japan Industry Standard (JIS) character to Japan Microsoft (JMS) character	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_mbcjistojms, _mbcjistojms_l, _mbcjmstojis, _mbcjmstojis_l	Convert JMS character to JIS character	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

_mbctohira, _mbctohira_l, _mbctokata, _mbctokata_l	Convert multibyte character to 1-byte hiragana code	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_mbctohira, _mbctohira_l, _mbctokata, _mbctokata_l	Convert multibyte character to 1-byte katakana code	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_mbctombb, _mbctombb_l	Convert 2-byte multibyte character to corresponding 1-byte multibyte character	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
mbstowcs, _mbstowcs_l, mbstowcs_s, _mbstowcs_s_l	Convert sequence of multibyte characters to corresponding sequence of wide characters	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
mbtowc, _mbtowc_l	Convert multibyte character to corresponding wide character	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
strtod, _strtod_l, wcstod, _wcstod_l	Convert string to **double**	System::Convert::ToDouble
strtol, wcstol, _strtol_l, _wcstol_l	Convert string to **long** integer	System::Convert::ToInt64
strtoul, _strtoul_l, wcstoul, _wcstoul_l	Convert string to **unsigned long** integer	System::Convert::ToUInt64
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l	Transform string into collated form based on locale-specific information	System::IFormattable::ToString
__toascii	Convert character to ASCII code	
tolower, _tolower, towlower, _tolower_l, _towlower_l, _mbctolower, _mbctolower_l, _mbctoupper, _mbctoupper_l	Test character and convert to lowercase if currently uppercase	System::Char::ToLower
tolower, _tolower, towlower, _tolower_l, _towlower_l	Convert character to lowercase unconditionally	System::String::ToLower
toupper, _toupper, towupper, _toupper_l, _towupper_l, _mbctolower, _mbctolower_l, _mbctoupper, _mbctoupper_l	Test character and convert to uppercase if currently lowercase	System::Char::ToUpper
toupper, _toupper, towupper, _toupper_l, _towupper_l	Convert character to uppercase unconditionally	System::String::ToUpper
_ultoa, _ultow, _ultoa_s, _ultow_s	Convert **unsigned long** to string	System::Convert::ToString

wcstombs, _wcstombs_l, wcstombs_s, _wcstombs_s_l	Convert sequence of wide characters to corresponding sequence of multibyte characters	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
wctomb, _wctomb_l, wctomb_s, _wctomb_s_l	Convert wide character to corresponding multibyte character	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
atof, _atof_l, _wtof, _wtof_l	Convert wide-character string to a **double**	System::Convert::ToInt64, System::Convert::ToUInt64, System::Convert::ToSingle, System::Convert::ToDouble
atoi, _atoi_l, _wtoi, _wtoi_l	Convert wide-character string to **int**	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_atoi64, _atoi64_l, _wtoi64, _wtoi64_l	Convert wide-character string to **__int64**	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
atol, _atol_l, _wtol, _wtol_l	Convert wide-character string to **long**	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Run-Time Routines by Category

Debug Routines

The debug version of the C run-time library supplies many diagnostic services that make debugging programs easier and allow developers to:

- Step directly into run-time functions during debugging
- Resolve assertions, errors, and exceptions
- Trace heap allocations and prevent memory leaks
- Report debug messages to the user

To use these routines, the _DEBUG flag must be defined. All of these routines do nothing in a retail build of an application. For more information on how to use the new debug routines, see CRT Debugging Techniques.

Debug Versions of the C Run-Time Library Routines

Routine	Use	.NET Framework equivalent
_ASSERT	Evaluate an expression and generates a debug report when the result is FALSE	System::Diagnostics::Debug::Assert
_ASSERTE	Similar to **_ASSERT**, but includes the failed expression in the generated report	System::Diagnostics::Debug::Assert
_CrtCheckMemory	Confirm the integrity of the memory blocks allocated on the debug heap	System::Diagnostics::PerformanceCounter
_CrtDbgBreak	Sets a break point.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_CrtDbgReport, _CrtDbgReportW	Generate a debug report with a user message and send the report to three possible destinations	System::Diagnostics::Debug::Write, System::Diagnostics::Debug::Writeline, System::Diagnostics::Debug::WriteIf, System::Diagnostics::Debug::WriteLineIf
_CrtDoForAllClientObjects	Call an application-supplied function for all **_CLIENT_BLOCK** types on the heap	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_CrtDumpMemoryLeaks	Dump all of the memory blocks on the debug heap when a significant memory leak has occurred	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_CrtIsMemoryBlock	Verify that a specified memory block is located within the local heap and that it has a valid debug heap block type identifier	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_CrtIsValidHeapPointer	Verifies that a specified pointer is in the local heap	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_CrtIsValidPointer	Verify that a specified memory range is valid for reading and writing	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

_CrtMemCheckpoint	Obtain the current state of the debug heap and store it in an application-supplied **_CrtMemState** structure	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_CrtMemDifference	Compare two memory states for significant differences and return the results	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_CrtMemDumpAllObjectsSince	Dump information about objects on the heap since a specified checkpoint was taken or from the start of program execution	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_CrtMemDumpStatistics	Dump the debug header information for a specified memory state in a user-readable form	System::Diagnostics::PerformanceCounter
_CrtReportBlockType	Returns the block type/subtype associated with a given debug heap block pointer.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_CrtSetAllocHook	Install a client-defined allocation function by hooking it into the C run-time debug memory allocation process	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_CrtSetBreakAlloc	Set a breakpoint on a specified object allocation order number	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_CrtSetDbgFlag	Retrieve or modify the state of the **_crtDbgFlag** flag to control the allocation behavior of the debug heap manager	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_CrtSetDumpClient	Install an application-defined function that is called every time a debug dump function is called to dump **_CLIENT_BLOCK** type memory blocks	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_CrtSetReportFile	Identify the file or stream to be used as a destination for a specific report type by **_CrtDbgReport**	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_CrtSetReportHook	Install a client-defined reporting function by hooking it into the C run-time debug reporting process	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_CrtSetReportHook2, _CrtSetReportHookW2	Installs or uninstalls a client-defined reporting function by hooking it into the C run-time debug reporting process.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_CrtSetReportMode	Specify the general destination(s) for a specific report type generated by **_CrtDbgReport**	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

_RPT[0,1,2,3,4]	Track the application's progress by generating a debug report by calling **_CrtDbgReport** with a format string and a variable number of arguments. Provides no source file and line number information.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_RPTF[0,1,2,3,4]	Similar to the **_RPTn** macros, but provides the source file name and line number where the report request originated	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_calloc_dbg	Allocate a specified number of memory blocks on the heap with additional space for a debugging header and overwrite buffers	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_expand_dbg	Resize a specified block of memory on the heap by expanding or contracting the block	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_free_dbg	Free a block of memory on the heap	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_fullpath_dbg, _wfullpath_dbg	Create an absolute or full path name for the specified relative path name, using _malloc_dbg to allocate memory.	System::IO::File::Create
_getcwd_dbg, _wgetcwd_dbg	Get the current working directory, using _malloc_dbg to allocate memory.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_malloc_dbg	Allocate a block of memory on the heap with additional space for a debugging header and overwrite buffers	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_msize_dbg	Calculate the size of a block of memory on the heap	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_realloc_dbg	Reallocate a specified block of memory on the heap by moving and/or resizing the block	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_strdup_dbg, _wcsdup_dbg	Duplicates a string, using _malloc_dbg to allocate memory.	System::String::Clone
_tempnam_dbg, _wtempnam_dbg	Generate names you can use to create temporary files, using _malloc_dbg to allocate memory.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

The debug routines can be used to step through the source code for most of the other C run-time routines during the debugging process. However, Microsoft considers some technology to be proprietary and, therefore, does not provide the source code for these routines. Most of these routines belong to either the exception handling or floating-point processing groups, but a few others are included as well. The following table lists these routines.

C Run-Time Routines That Are Not Available in Source Code Form

acos, acosf	_fpclass	_nextafter
asin	_fpieee_flt	pow
atan, atan2	_fpreset	printf, _printf_l, wprintf, _wprintf_l, printf_s, _printf_s_l, wprintf_s, _wprintf_s_l*
_cabs	frexp	_scalb
ceil	_hypot	scanf, _scanf_l, wscanf, _wscanf_l, scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l*
_chgsign, _chgsignl	_isnan	setjmp
_clear87, _clearfp	_j0	sin
_control87, _controlfp, __control87_2	_j1	sinh
_copysign, _copysignl	_jn	sqrt
cos	ldexp	_status87, _statusfp
cosh	log	tan
Exp	log10	tanh
fabs	_logb	_y0
_finite	longjmp	_y1
floor	_matherr	_yn
fmod	modf	

* Although source code is available for most of this routine, it makes an internal call to another routine for which source code is not provided.

Some C run-time functions and C++ operators behave differently when called from a debug build of an application. (Note that a debug build of an application can be done by either defining the **_DEBUG** flag or by linking with a debug version of the C run-time library.) The behavioral differences usually consist of extra features or information provided by the routine to support the debugging process. The following table lists these routines.

Routines that Behave Differently in a Debug Build of an Application

C abort routine	C++ delete operator
C assert routine	C++ new operator

For more information about using the debug versions of the C++ operators in the preceding table, see Using the Debug Heap from C++.

See Also

Reference

Run-Time Routines by Category

Run-Time Error Checking

Deprecated CRT Functions

Deprecated CRT functions and their recommended replacements.

Deprecated POSIX functions

The following POSIX names for functions are deprecated. In most cases, prepending an underscore character gives the standard equivalent name. Note that some functions have more secure versions ("_s" suffix).

You can also eliminate POSIX deprecation warnings by defining **_CRT_NONSTDC_NO_DEPRECATE**.

Deprecated function	Replacement function
access	_access, _access_s
cabs	_cabs
cgets	_cgets, _cgets_s
chdir	_chdir
chmod	_chmod
chsize	_chsize, _chsize_s
close	_close
cprintf	_cprintf, _cprintf_s
cputs	_cputs
creat	_creat
cscanf	_cscanf, _cscanf_s
cwait	_cwait
dup	_dup
dup2	_dup2
ecvt	_ecvt, _ecvt_s
eof	_eof
execl	_execl
execle	_execle
execlp	_execlp
execlpe	_execlpe
execv	_execv

execve	_execve
execvp	_execvp
execvpe	_execvpe
fcloseall	_fcloseall
fcvt	_fcvt, _fcvt_s
fdopen	_fdopen
fgetchar	_fgetchar
filelength	_filelength
fileno	_fileno
flushall	_flushall
fputchar	_fputchar, _fputwchar
gcvt	_gcvt, _gcvt_s
getch	_getch
getche	_getche
getcwd	_getcwd, _wgetcwd
getpid	_getpid
getw	_getw
hypot	_hypot
inp	_inp
inpw	_inpw
isascii	__isascii
isatty	_isatty
iscsym	__iscsym
iscsymf	__iscsymf
itoa	_itoa, _itoa_s
j0	_j0
j1	_j1

jn	_jn
kbhit	_kbhit
lfind	_lfind, _lfind_s
locking	_locking
lsearch	_lsearch, _lsearch_s
lseek	_lseek
ltoa	_ltoa, _ltoa_s
memccpy	_memccpy
memicmp	_memicmp
mkdir	_mkdir
mktemp	_mktemp, _mktemp_s
open	_open
outp	_outp
outpw	_outpw
putch	_putch
putenv	_putenv, _putenv_s
putw	_putw
read	_read
rmdir	_rmdir
rmtmp	_rmtmp
setmode	_setmode
sopen	_sopen, _sopen_s
spawnl	_spawnl
spawnle	_spawnle
spawnlp	_spawnlp
spawnlpe	_spawnlpe
spawnv	_spawnv

spawnve	_spawnve
spawnvp	_spawnvp
spawnvpe	_spawnvpe
strcmpi	_stricmp
strdup	_strdup
stricmp	_stricmp
strlwr	_strlwr, _strlwr_s
strnicmp	_strnicmp
strnset	_strnset, _strnset_s
strrev	_strrev
strset	_strset, _strset_s
strupr	_strupr, _strupr_s
swab	_swab
tell	_tell
tempnam	_tempnam
toascii	__toascii
tzset	_tzset
ultoa	_ultoa, _ultoa_s
umask	_umask, _umask_s
ungetch	_ungetch
unlink	_unlink
wcsdup	_wcsdup
wcsicmp	_wcsicmp
wcsicoll	_wcsicoll
wcslwr	_wcslwr, _wcslwr_s
wcsnicmp	_wcsnicmp
wcsnset	_wcsnset, _wcsnset_s

wcsrev	_wcsrev
wcsset	_wcsset, _wcsset_s
wcsupr	_wcsupr, _wcsupr_s
write	_write
y0	_y0
y1	_y1
yn	_yn

Security-enhanced functions

The following functions should be avoided because a more secure version of the function should be used instead. For more information, see Security Enhancements in the CRT.

Deprecated function	Security-enhanced replacement
_alloca	_malloca
asctime	asctime_s
_cgets, _cgetws	_cgets_s, _cgetws_s
_chsize	_chsize_s
_controlfp	_controlfp_s
_creat	_sopen_s
_cscanf	_cscanf_s
_cscanf_l	_cscanf_s_l
ctime	ctime_s
_ctime32	_ctime32_s
_ctime64	_ctime64_s
_cwscanf	_cwscanf_s
_cwscanf_l	_cwscanf_s_l
_ecvt	_ecvt_s
_fcvt	_fcvt_s
fopen	fopen_s
freopen	freopen_s

fscanf	fscanf_s
_fscanf_l	_fscanf_s_l
fwscanf	fwscanf_s
_fwscanf_l	_fwscanf_s_l
_gcvt	_gcvt_s
getenv	getenv_s
gets, _getws	gets_s, _getws_s
gmtime	_gmtime_s
_gmtime32	_gmtime32_s
_gmtime64	_gmtime64_s
_i64toa	_i64toa_s
_i64tow	_i64tow_s
_itoa	_itoa_s
_itow	_itow_s
localtime	_localtime_s
_localtime32	_localtime32_s
_localtime64	_localtime64_s
_ltoa, _ltow	_ltoa_s, _ltow_s
_mbccpy	_mbccpy_s
_mbccpy_l	_mbccpy_s_l
_mbscat	_mbscat_s
_mbscpy	_mbscpy_s
_mbslwr, _mbslwr_l	_mbslwr_s, _mbslwr_s_l
_mbsnbcat, _mbsnbcat_l	_mbsnbcat_s, _mbsnbcat_s_l
_mbsnbcpy, _mbsnbcpy_l	_mbsnbcpy_s, _mbsnbcpy_s_l
_mbsnbset	_mbsnbset_s
_mbsnbset_l	_mbsnbset_s_l

_mbsncat	_mbsncat_s
_mbsncat_l	_mbsncat_s_l
_mbsncpy	_mbsncpy_s
_mbsncpy_l	_mbsncpy_s_l
_mbsnset	_mbsnset_s
_mbsnset_l	_mbsnset_s_l
mbsrtowcs	mbsrtowcs_s
_mbsset	_mbsset_s
_mbsset_l	_mbsset_s_l
_mbstok	_mbstok_s
_mbstok_l	_mbstok_s_l
mbstowcs, _mbstowcs_l	mbstowcs_s, _mbstowcs_s_l
_mbsupr, _mbsupr_l	_mbsupr_s, _mbsupr_s_l
memcpy	memcpy_s
memmove	memmove_s
_mktemp	_mktemp_s
_open	_sopen_s
scanf	scanf_s
_scanf_l	_scanf_s_l
_searchenv	_searchenv_s
setbuf	setvbuf
_snprintf	_snprintf_s
_snprintf_l	_snprintf_s_l
_snscanf	_snscanf_s
_snscanf_l	_snscanf_s_l
_snwprintf	_snwprintf_s
_snwprintf_l	_snwprintf_s_l

_snwscanf	_snwscanf_s
_snwscanf_l	_snwscanf_s_l
_sopen	_sopen_s
_splitpath	_splitpath_s
sprintf	sprintf_s
_sprintf_l	_sprintf_s_l
sscanf	sscanf_s
_sscanf_l	_sscanf_s_l
strcat	strcat_s
strcpy	strcpy_s
_strdate	_strdate_s
strerror, _strerror	strerror_s, _strerror_s
_strlwr, _strlwr_l	_strlwr_s, _strlwr_s_l
strncat, _strncat_l	strncat_s, _strncat_s_l
strncpy, _strncpy_l	strncpy_s, _strncpy_s_l
_strnset	_strnset_s
_strnset_l	_strnset_s_l
_strset	_strset_s
_strset_l	_strset_s_l
_strtime	_strtime_s
strtok	strtok_s
_strtok_l	_strtok_s_l
_strupr, _strupr_l	_strupr_s, _strupr_s_l
swprintf	swprintf_s
_swprintf_l	_swprintf_s_l
swscanf	swscanf_s
_swscanf_l	_swscanf_s_l

tmpfile	tmpfile_s
_ui64toa, _ui64tow	_ui64toa_s, _ui64tow_s
_ultoa, _ultow	_ultoa_s, _ultow_s
_umask	_umask_s
vsnprintf, _vsnprintf, _vsnprintf_l, _vsnwprintf, _vsnwprintf_l	vsnprintf_s, _vsnprintf_s, _vsnprintf_s_l, _vsnwprintf_s, _vsnwprintf_s_l
vsprintf, _vsprintf_l, vswprintf, _vswprintf_l, __vswprintf_l	vsprintf_s, _vsprintf_s_l, vswprintf_s, _vswprintf_s_l
_wasctime	_wasctime_s
_wcreat	_wsopen_s
wcrtomb	wcrtomb_s
wcscat	wcscat_s
wcscpy	wcscpy_s
_wcserror, __wcserror	_wcserror_s, __wcserror_s
_wcslwr, _wcslwr_l	_wcslwr_s, _wcslwr_s_l
wcsncat, wcsncat_l	wcsncat_s, _wcsncat_s_l
_wcsncpy, _wcsncpy_l	wcsncpy_s, _wcsncpy_s_l
_wcsnset, _wcsnset_l	_wcsnset_s, _wcsnset_s_l
wcsrtombs	wcsrtombs_s
_wcsset, _wcsset_l	_wcsset_s_wcsset_s_l
wcstok, _wcstok_l	wcstok_s, _wcstok_s_l
wcstombs, _wcstombs_l	wcstombs_s, _wcstombs_s_l
_wcsupr_l, _wcsupr	_wcsupr_s, _wcsupr_s_l
_wctime, _wctime32, _wctime64	_wctime_s, _wctime32_s, _wctime64_s
wctomb, _wctomb_l	wctomb_s, _wctomb_s_l
_wfopen	_wfopen_s
_wfreopen	_wfreopen_s
_wgetenv	_wgetenv_s
wmemcpy	wmemcpy_s

wmemmove	wmemmove_s
_wmktemp	_wmktemp_s
_wopen	_wsopen_s
_wscanf, _wscanf_l	_wscanf_s, _wscanf_s_l
_wsearchenv	_wsearchenv_s
_wsopen	_wsopen_s
_wsplitpath	_wsplitpath_s
_wstrdate	_wstrdate_s
_wstrtime	_wstrtime_s

Functions with Secure Template Overloads

Some of the functions in the preceding section have secure template overloads which can eliminate deprecation warnings by replacing calls to deprecated functions with calls to the new secure versions of those functions. For more information, see Secure Template Overloads.

Deprecated functions with secure template overloads
_cgets, _cgetws
gets, _getws
_itoa, _i64toa, _ui64toa, _itow, _i64tow, _ui64tow
_ltoa, _ltow
_mbsnbcat, _mbsnbcat_l
_mbsnbcpy, _mbsnbcpy_l
mbsrtowcs
mbstowcs, _mbstowcs_l
_mktemp, _wmktemp
_searchenv, _wsearchenv
_snprintf, _snprintf_l, _snwprintf, _snwprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
strcat, wcscat, _mbscat
strcpy, wcscpy, _mbscpy
_strdate, _wstrdate

_strlwr, _wcslwr, _mbslwr, _strlwr_l, _wcslwr_l, _mbslwr_l
strncat, _strncat_l, wcsncat, wcsncat_l, _mbsncat _mbsncat_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
_strtime, _wstrtime
_strupr, _strupr_l, _mbsupr, _mbsupr_l, _wcsupr_l, _wcsupr
_ultoa, _ultow
vsnprintf, _vsnprintf, _vsnprintf_l, _vsnwprintf, _vsnwprintf_l
vsprintf, _vsprintf_l, vswprintf, _vswprintf_l, __vswprintf_l
wcrtomb
wcsrtombs
wcstombs, _wcstombs_l

See Also

Reference

Run-Time Routines by Category

Directory Control

These routines access, modify, and obtain information about the directory structure.

Directory-Control Routines

Routine	Use	.NET Framework equivalent
_chdir, _wchdir	Change current working directory	System::Environment::CurrentDirectory
_chdrive	Change current drive	System::Environment::CurrentDirectory
_getcwd, _wgetcwd	Get current working directory for default drive	System::Environment::CurrentDirectory
_getdcwd, _wgetdcwd	Get current working directory for specified drive	System::Environment::CurrentDirectory
_getdiskfree	Populates a **_diskfree_t** structure with information about a disk drive.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_getdrive	Get current (default) drive	System::Environment::CurrentDirectory
_getdrives	Returns a bitmask representing the currently available disk drives.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_mkdir, _wmkdir	Make new directory	System::IO::Directory::CreateDirectory, System::IO::DirectoryInfo::CreateSubdirectory
_rmdir, _wrmdir	Remove directory	System::IO::Directory::Delete
_searchenv, _wsearchenv, _searchenv_s, _wsearchenv_s	Search for given file on specified paths	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Run-Time Routines by Category

Error Handling (CRT)

Use these routines to handle program errors.

Error-Handling Routines

Routine	Use	.NET Framework equivalent
assert macro	Test for programming logic errors; available in both the release and debug versions of the run-time library	System::Diagnostics::Debug::Assert
_ASSERT, _ASSERTE macros	Similar to **assert**, but only available in the debug versions of the run-time library	System::Diagnostics::Debug::Assert
clearerr	Reset error indicator. Calling **rewind** or closing a stream also resets the error indicator.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_eof	Check for end of file in low-level I/O	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
feof	Test for end of file. End of file is also indicated when **_read** returns 0.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
ferror	Test for stream I/O errors	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_RPT, _RPTF macros	Generate a report similar to **printf**, but only available in the debug versions of the run-time library	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_set_error_mode	Modifies **__error_mode** to determine a non-default location where the C run time writes an error message for an error that will possibly end the program.	
_set_purecall_handler	Sets the handler for a pure virtual function call.	

See Also

Reference

Run-Time Routines by Category

Exception Handling Routines

Use the C++ exception-handling functions to recover from unexpected events during program execution.

Exception-Handling Functions

Function	Use	.NET Framework equivalent
_set_se_translator	Handle Win32 exceptions (C structured exceptions) as C++ typed exceptions	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
set_terminate	Install your own termination routine to be called by **terminate**	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
set_unexpected	Install your own termination function to be called by **unexpected**	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
terminate	Called automatically under certain circumstances after exception is thrown. The **terminate** function calls **abort** or a function you specify using **set_terminate**	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
unexpected	Calls **terminate** or a function you specify using **set_unexpected**. The **unexpected** function is not used in current Microsoft C++ exception-handling implementation	System::Exception Class

See Also

Reference

Run-Time Routines by Category

File Handling

Use these routines to create, delete, and manipulate files and to set and check file-access permissions.

The C run-time libraries have a 512 limit for the number of files that can be open at any one time. Attempting to open more than the maximum number of file descriptors or file streams causes program failure. Use _setmaxstdio to change this number.

The following routines operate on files designated by a file descriptor.

File-Handling Routines (File Descriptor)

Routine	Use	.NET Framework equivalent
_chsize,_chsize_s	Change file size	System::IO::Stream::SetLength, System::IO::FileStream::SetLength
_filelength, _filelengthi64	Get file length	System::IO::Stream::SetLength, System::IO::FileStream::SetLength
_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32	Get file-status information on descriptor	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_get_osfhandle	Return operating-system file handle associated with existing C run-time file descriptor	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_isatty	Check for character device	System::IO::Stream::CanWrite, System::IO::FileStream::CanWrite
_locking	Lock areas of file	System::IO::FileStream::Lock
_open_osfhandle	Associate C run-time file descriptor with existing operating-system file handle	System::IO::FileStream::Handle
_setmode	Set file-translation mode	System::IO::BinaryReader Class, System::IO::TextReader Class

The following routines operate on files specified by a path or filename.

File-Handling Routines (Path or Filename)

Routine	Use	.NET Framework equivalent
_access, _waccess, _access_s, _waccess_s	Check file-permission setting	System::IO::FileAccess Enumeration
_chmod, _wchmod	Change file-permission setting	System::IO::File::SetAttributes, System::Security::Permissions::FileIOPermission
_fullpath, _wfullpath	Expand a relative path to its absolute path name	System::IO::File::Create
_makepath, _wmakepath, _makepath_s, _wmakepath_s	Merge path components into single, full path	System::IO::File::Create

_mktemp, _wmktemp, _mktemp_s, _wmktemp_s	Create unique filename	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
remove, _wremove	Delete file	System::IO::File::Delete
rename, _wrename	Rename file	System::IO::File::Move
_splitpath, _wsplitpath, _splitpath_s, _wsplitpath_s	Parse path into components	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_stat, _stat64, _stati64, _wstat, _wstat64, _wstati64	Get file-status information on named file	System::IO::File::GetAttributes, System::IO::File::GetCreationTime, System::IO::File::GetLastAccessTime, System::IO::File::GetLastWriteTime
_umask, _umask_s	Set default permission mask for new files created by program	System::IO::File::SetAttributes
_unlink, _wunlink	Delete file	System::IO::File::Delete

The following routines open files.

File-Handling Routines (Open File)

Routine	Use	.NET Framework equivalent
fopen, _wfopen, fopen_s, _wfopen_s	Opens a file and returns a pointer to the open file.	System::IO::File::Open, System::IO::FileStream::FileStream
_fsopen, _wfsopen	Open a stream with file sharing and returns a pointer to the open file.	System::IO::File::Open, System::IO::FileStream::FileStream
_open, _wopen	Opens a file and returns a file descriptor to the opened file.	System::IO::File::Open, System::IO::FileStream::FileStream
_sopen, _wsopen, _sopen_s, _wsopen_s	Open a file with file sharing and returns a file descriptor to the open file.	
_pipe	Creates a pipe for reading and writing.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
freopen, _wfreopen, freopen_s, _wfreopen_s	Reassign a file pointer.	System::IO::File::Open, System::IO::FileStream::FileStream

The following functions provide a way to change the representation of the file between a **FILE** structure, a file descriptor, and a Win32 file handle.

_fdopen, _wfdopen	Associates a stream with a file that was previously opened for low-level I/O and returns a pointer to the open stream.	System::IO::FileStream::FileStream
_fileno	Gets the file descriptor associated with a stream.	System::IO::FileStream::Handle

_get_osfhandle	Return operating-system file handle associated with existing C run-time file descriptor	Not applicable. To call the standard C function, use **P/Invoke**. For more information, see Platform Invoke Examples.
_open_osfhandle	Associates C run-time file descriptor with an existing operating-system file handle.	System::IO::FileStream::Handle

The following Win32 functions also open files and pipes:

- CreateFile
- CreatePipe
- CreateNamedPipe

See Also
Reference
Run-Time Routines by Category

Floating-Point Support

Many Microsoft run-time library functions require floating-point support from a math coprocessor or from the floating-point libraries that accompany the compiler. Floating-point support functions are loaded only if required.

When you use a floating-point type specifier in the format string of a call to a function in the **printf** or **scanf** family, you must specify a floating-point value or a pointer to a floating-point value in the argument list to tell the compiler that floating-point support is required.

To handle floating-point exceptions, refer to the sample code in the _fpieee_flt topic.

Floating-point precision of intermediate values are controlled by the functions _control87, _controlfp, __control87_2. By default, **_controlfp's** precision control is set to 53 bits (_PC_53). Linking with FP10.OBJ changes the default precision control to 64 bits (_PC_64). On the linker command line, FP10.OBJ must appear before LIBC.LIB, LIBCMT.LIB, or MSVCRT.LIB.

Floating-Point Functions

Routine	Use	.NET Framework equivalent
abs	Return absolute value of **int**	System::Math::Abs
acos, acosf	Calculate arccosine	System::Math::Acos
asin, asinf	Calculate arcsine	System::Math::Asin
atan, atanf, atan2, atan2f	Calculate arctangent	System::Math::Atan, System::Math::Atan2
atof, _atof_l, _wtof, _wtof_l	Convert character string to double-precision floating-point value	System::Convert::ToSingle, System::Convert::ToDouble
Bessel functions	Calculate Bessel functions **_j0**, **_j1**, **_jn**, **_y0**, **_y1**, **_yn**	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_cabs	Find absolute value of complex number	Not applicable.
ceil, ceilf	Find integer ceiling	System::Math::Ceiling
_chgsign, _chgsignl	Reverse sign of double-precision floating-point or long double-precision floating-point argument	Not applicable.
_clear87, _clearfp	Get and clear floating-point status word	Not applicable.
_control87, _controlfp, __control87_2, _controlfp_s	Get old floating-point control word and set new control-word value	Not applicable.
_copysign, _copysignl	Return one value with sign of another	Not applicable.
cos, cosf, cosh, coshf	Calculate cosine	System::Math::Cos, System::Math::Cosh
difftime	Compute difference between two specified time values	System::DateTime::Subtract
div	Divide one integer by another, returning quotient and remainder	Not applicable.

_ecvt, _ecvt_s	Convert **double** to character string of specified length	System::Convert::ToString
exp, expf	Calculate exponential function	System::Math::Exp
fabs, fabsf	Find absolute value	System::Math::Abs
_fcvt, _fcvt_s	Convert **double** to string with specified number of digits following decimal point	System::Convert::ToString
_finite	Determine whether given double-precision floating-point value is finite	System::Double::IsInfinity
floor, floorf	Find largest integer less than or equal to argument	System::Math::Floor
fmod, fmodf	Find floating-point remainder	System::Math::IEEERemainder
_fpclass	Return status word containing information on floating-point class	System::Double::IsInfinity, System::Double::IsNegativeInfinity, System::Double::IsPositiveInfinity, System::Double::IsNan
_fpieee_flt	Invoke user-defined trap handler for IEEE floating-point exceptions	Not applicable.
_fpreset	Reinitialize floating-point math package	
frexp	Calculate exponential value	Not applicable.
_gcvt, _gcvt_s	Convert floating-point value to character string	System::Convert::ToString
_hypot, hypotf	Calculate hypotenuse of right triangle	Not applicable.
_isnan	Check given double-precision floating-point value for not a number (NaN)	System::Double::IsNan
labs	Return absolute value of **long**	System::Math::Abs
ldexp	Calculate product of argument and 2 to specified power	System::Math::Pow
ldiv	Divide one **long** integer by another, returning quotient and remainder	Not applicable.
log, logf, log10, log10f	Calculate natural or base-10 logarithm.	System::Math::Log, System::Math::Log10
_logb	Extract exponential value of double-precision floating-point argument	Not applicable.

_lrotl, _lrotr	Shift **unsigned long int** left (**_lrotl**) or right (**_lrotr**)	Not applicable.
_matherr	Handle math errors	Not applicable.
__max	Return larger of two values	System::Math::Max
__min	Return smaller of two values	System::Math::Min
modf, modff	Split argument into integer and fractional parts	Not applicable.
_nextafter	Return next representable neighbor	Not applicable.
pow, powf	Calculate value raised to a power	System::Math::Pow
printf, _printf_l, wprintf, _wprintf_l, printf_s, _printf_s_l, wprintf_s, _wprintf_s_l	Write data to **stdout** according to specified format	System::Console::Write, System::Console::WriteLine
rand, rand_s	Get pseudorandom number	System::Random Class
_rotl, _rotr	Shift **unsigned int** left (**_rotl**) or right (**_rotr**)	Not applicable.
_scalb	Scale argument by power of 2	Not applicable.
scanf, wscanf, scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l	Read data from **stdin** according to specified format and write data to specified location	System::Console::Read, System::Console::ReadLine
_set_controlfp	set new control-word value	Not applicable.
sin, sinf, sinh, sinhf	Calculate sine or hyperbolic sine	System::Math::Sin, System::Math::Sinh
sqrt	Find square root	System::Math::Sqrt
srand	Initialize pseudorandom series	System::Random Class
_status87, _statusfp, _statusfp2	Get floating-point status word	Not applicable.
strtod, _strtod_l, wcstod, _wcstod_l	Convert character string to double-precision value	System::Convert::ToDouble
tan, tanf, tanh, tanhf	Calculate tangent or hyperbolic tangent	System::Math::Tan, System::Math::Tanh

See Also

Reference

Run-Time Routines by Category

Long Double

Previous 16-bit versions of Microsoft C/C++ and Microsoft Visual C++ supported the **long double**, 80-bit precision data type. In Win32 programming, however, the **long double** data type maps to the **double**, 64-bit precision data type. The Microsoft run-time library provides **long double** versions of the math functions only for backward compatibility. The **long double** function prototypes are identical to the prototypes for their **double** counterparts, except that the **long double** data type replaces the **double** data type. The **long double** versions of these functions should not be used in new code.

Double Functions and Their Long Double Counterparts

Function	Long double counterpart	Function	Long double counterpart
acos	**acosl**	frexp	**frexpl**
asin	**asinl**	_hypot	**_hypotl**
atan	**atanl**	ldexp	**ldexpl**
atan2	**atan2l**	log	**logl**
atof	**_atold**	log10	**log10l**
Bessel functions j0, j1, jn	**j0l, j1l, jnl**	_matherr	**_matherrl**
Bessel functions y0, y1, yn	**y0l, y1l, ynl**	modf	**modfl**
_cabs	**_cabsl**	pow	**powl**
ceil	**ceill**	sin	**sinl**
cos	**cosl**	sinh	**sinhl**
cosh	**coshl**	sqrt	**sqrtl**
exp	**expl**	strtod	**_strtold**
fabs	**fabsl**	tan	**tanl**
floor	**floorl**	tanh	**tanhl**
fmod	**fmodl**		

See Also

Reference

Run-Time Routines by Category

Input and Output

The I/O functions read and write data to and from files and devices. File I/O operations take place in text mode or binary mode. The Microsoft run-time library has three types of I/O functions:

- Stream I/O functions treat data as a stream of individual characters.
- Low-level I/O functions invoke the operating system directly for lower-level operation than that provided by stream I/O.
- Console and port I/O functions read or write directly to a console (keyboard and screen) or an I/O port (such as a printer port).

Note
Because stream functions are buffered and low-level functions are not, these two types of functions are generally incompatible. For processing a particular file, use either stream or low-level functions exclusively.

See Also

Reference

Run-Time Routines by Category

Text and Binary Mode File I/O

File I/O operations take place in one of two translation modes, text or binary, depending on the mode in which the file is opened. Data files are usually processed in text mode. To control the file translation mode, one can:

- Retain the current default setting and specify the alternative mode only when you open selected files.
- Use the function _set_fmode to change the default mode for newly opened files. Use _get_fmode to find the current default mode. The initial default setting is text mode (**_O_TEXT**).
- Change the default translation mode directly by setting the global variable _fmode in your program. The function **_set_fmode** sets the value of this variable, but it can also be set directly.

When you call a file-open function such as _open, fopen, fopen_s, freopen, freopen_s, _fsopen or _sopen_s, you can override the current default setting of **_fmode** by specifying the appropriate argument to the function _set_fmode. The **stdin**, **stdout**, and **stderr** streams always open in text mode by default; you can also override this default when opening any of these files. Use _setmode to change the translation mode using the file descriptor after the file is open.

See Also
Reference
Input and Output
Run-Time Routines by Category

Unicode Stream I/O in Text and Binary Modes

When a Unicode stream I/O routine (such as **fwprintf**, **fwscanf**, **fgetwc**, **fputwc**, **fgetws**, or **fputws**) operates on a file that is open in text mode (the default), two kinds of character conversions take place:

- Unicode-to-MBCS or MBCS-to-Unicode conversion. When a Unicode stream-I/O function operates in text mode, the source or destination stream is assumed to be a sequence of multibyte characters. Therefore, the Unicode stream-input functions convert multibyte characters to wide characters (as if by a call to the **mbtowc** function). For the same reason, the Unicode stream-output functions convert wide characters to multibyte characters (as if by a call to the **wctomb** function).
- Carriage return – linefeed (CR-LF) translation. This translation occurs before the MBCS – Unicode conversion (for Unicode stream input functions) and after the Unicode – MBCS conversion (for Unicode stream output functions). During input, each carriage return – linefeed combination is translated into a single linefeed character. During output, each linefeed character is translated into a carriage return – linefeed combination.

However, when a Unicode stream-I/O function operates in binary mode, the file is assumed to be Unicode, and no CR-LF translation or character conversion occurs during input or output. Use the _setmode(_fileno(stdin), _O_BINARY); instruction in order to correctly use wcin on a UNICODE text file.

See Also
Reference
Run-Time Routines by Category
Input and Output

Stream I/O

These functions process data in different sizes and formats, from single characters to large data structures. They also provide buffering, which can improve performance. The default size of a stream buffer is 4K. These routines affect only buffers created by the run-time library routines, and have no effect on buffers created by the operating system.

Stream I/O Routines

Routine	Use	.NET Framework equivalent
clearerr, clearerr_s	Clear error indicator for stream	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
fclose	Close stream	System::IO::Stream::Close, System::IO::BinaryReader::Close, System::IO::BinaryWriter::Close, System::IO::TextReader::Close, System::IO::TextWriter::Close, System::IO::StringReader::Close, System::IO::StringWriter::Close, System::IO::StreamReader::Close, System::IO::StreamWriter::Close
_fcloseall	Close all open streams except **stdin**, **stdout**, and **stderr**	System::IO::Stream::Close, System::IO::BinaryReader::Close, System::IO::BinaryWriter::Close, System::IO::TextReader::Close, System::IO::TextWriter::Close, System::IO::StringReader::Close, System::IO::StringWriter::Close, System::IO::StreamReader::Close, System::IO::StreamWriter::Close
_fdopen, wfdopen	Associate stream with file descriptor of open file	System::IO::FileStream::FileStream
feof	Test for end of file on stream	System::IO::FileStream::Read
ferror	Test for error on stream	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
fflush	Flush stream to buffer or storage device	System::IO::FileStream::Flush
fgetc, fgetwc	Read character from stream (function versions of **getc** and **getwc**)	System::IO::StreamReader::Read

_fgetchar, _fgetwchar	Read character from **stdin** (function versions of **getchar** and **getwchar**)	System::Console::Read
fgetpos	Get position indicator of stream	System::IO::FileStream::Position
fgets, fgetws	Read string from stream	System::IO::StreamReader::ReadLine, System::IO::TextReader::ReadBlock
_fileno	Get file descriptor associated with stream	System::IO::FileStream::Handle
_flushall	Flush all streams to buffer or storage device	System::IO::FileStream::Flush, System::IO::StreamWriter::Flush, System::IO::TextWriter::Flush, System::IO::BinaryWriter::Flush
fopen, _wfopen, fopen_s, _wfopen_s	Open stream	System::IO::File::Open
fprintf, _fprintf_l, fwprintf, _fwprintf_l, fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l	Write formatted data to stream	System::IO::StreamWriter::Write
fputc, fputwc	Write a character to a stream (function versions of **putc** and **putwc**)	System::IO::StreamWriter::Write
_fputchar, _fputwchar	Write character to **stdout** (function versions of **putchar** and **putwchar**)	System::Console::Write
fputs, fputws	Write string to stream	System::IO::StreamWriter::Write
fread	Read unformatted data from stream	System::IO::FileStream::Read

freopen, _wfreopen, freopen_s, _wfreopen_s	Reassign **FILE** stream pointer to new file or device	System::IO::File::Open
fscanf, fwscanf, fscanf_s, _fscanf_s_l, fwscanf_s, _fwscanf_s_l	Read formatted data from stream	System::IO::StreamReader::ReadLine; see also **Parse** methods, such as System::Double::Parse.
fseek, _fseeki64	Move file position to given location	System::IO::FileStream::Position, System::IO::FileStream::Seek
fsetpos	Set position indicator of stream	System::IO::FileStream::Position
_fsopen, _wfsopen	Open stream with file sharing	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
ftell, _ftelli64	Get current file position	System::IO::FileStream::Position
fwrite	Write unformatted data items to stream	System::IO::FileStream::Write
getc, getwc	Read character from stream (macro versions of **fgetc** and **fgetwc**)	System::IO::StreamReader::Read
getchar, getwchar	Read character from **stdin** (macro versions of **fgetchar** and **fgetwchar**)	System::Console::Read
_getmaxstdio	Returns the number of simultaneously open files permitted at the stream I/O level.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
gets, getws, gets_s, _getws_s	Read line from **stdin**	System::Console::Read

_getw	Read binary **int** from stream	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
printf, _printf_l, wprintf, _wprintf_l,printf_s, _printf_s_l, wprintf_s, _wprintf_s_l	Write formatted data to **stdout**	System::Console::Write
putc, putwc	Write character to a stream (macro versions of **fputc** and **fputwc**)	System::IO::StreamWriter::Write
putchar, putwchar	Write character to **stdout** (macro versions of **fputchar** and **fputwchar**)	System::Console::Write
puts, _putws	Write line to stream	System::Console::Write
_putw	Write binary **int** to stream	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
rewind	Move file position to beginning of stream	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_rmtmp	Remove temporary files created by **tmpfile**	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
scanf, _scanf_l, wscanf, _wscanf_l,scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l	Read formatted data from **stdin**	System::Console::ReadLine; see also **Parse** methods, such as System::Double::Parse.
setbuf	Control stream buffering	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_setmaxstdio	Set a maximum for the number of simultaneously open files at the stream I/O level.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

setvbuf	Control stream buffering and buffer size	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_snprintf, _snwprintf, _snprintf_s, _snprintf_s_l, _snwprintf_s, _snwprintf_s_l	Write formatted data of specified length to string	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_snscanf, _snwscanf, _snscanf_s, _snscanf_s_l, _snwscanf_s, _snwscanf_s_l	Read formatted data of a specified length from the standard input stream.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
sprintf, swprintf, sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l	Write formatted data to string	System::String::Format
sscanf, swscanf, sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l	Read formatted data from string	See **Parse** methods, such as System::Double::Parse
_tempnam, _wtempnam	Generate temporary file name in given directory	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
tmpfile, tmpfile_s	Create temporary file	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
tmpnam, _wtmpnam, tmpnam_s, _wtmpnam_s	Generate temporary file name	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
ungetc, ungetwc	Push character back onto stream	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_vcprintf, _vcwprintf, _vcprintf_s, _vcprintf_s_l, _vcwprintf_s, _vcwprintf_s_l	Write formatted data to the console.	System::Console::Write
vfprintf, vfwprintf, vfprintf_s, _vfprintf_s_l, vfwprintf_s, _vfwprintf_s_l	Write formatted data to stream	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
vprintf, vwprintf, vprintf_s, _vprintf_s_l, vwprintf_s, _vwprintf_s_l	Write formatted data to **stdout**	System::Console::Write

_vsnprintf, _vsnwprintf, vsnprintf_s, _vsnprintf_s, _vsnprintf_s_l, _vsnwprintf_s, _vsnwprintf_s_l	Write formatted data of specified length to buffer	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
vsprintf, vswprintf, vsprintf_s, _vsprintf_s_l, vswprintf_s, _vswprintf_s_l	Write formatted data to buffer	System::String::Format

When a program begins execution, the startup code automatically opens several streams: standard input (pointed to by **stdin**), standard output (pointed to by **stdout**), and standard error (pointed to by **stderr**). These streams are directed to the console (keyboard and screen) by default. Use **freopen** to redirect **stdin**, **stdout**, or **stderr** to a disk file or a device.

Files opened using the stream routines are buffered by default. The **stdout** and **stderr** functions are flushed whenever they are full or, if you are writing to a character device, after each library call. If a program terminates abnormally, output buffers may not be flushed, resulting in loss of data. Use **fflush** or **_flushall** to ensure that the buffer associated with a specified file or all open buffers are flushed to the operating system, which can cache data before writing it to disk. The commit-to-disk feature ensures that the flushed buffer contents are not lost in the event of a system failure.

There are two ways to commit buffer contents to disk:

- Link with the file COMMODE.OBJ to set a global commit flag. The default setting of the global flag is **n**, for "no-commit."
- Set the mode flag to **c** with **fopen** or **_fdopen**.

Any file specifically opened with either the **c** or the **n** flag behaves according to the flag, regardless of the state of the global commit/no-commit flag.

If your program does not explicitly close a stream, the stream is automatically closed when the program terminates. However, you should close a stream when your program finishes with it, as the number of streams that can be open at one time is limited. See _setmaxstdio for information on this limit.

Input can follow output directly only with an intervening call to **fflush** or to a file-positioning function (**fseek**, **fsetpos**, or **rewind**). Output can follow input without an intervening call to a file-positioning function if the input operation encounters the end of the file.

See Also

Reference

Input and Output

Run-Time Routines by Category

Low-Level I/O

These functions invoke the operating system directly for lower-level operation than that provided by stream I/O. Low-level input and output calls do not buffer or format data.

Low-level routines can access the standard streams opened at program startup using the following predefined file descriptors.

Stream	File Descriptor
stdin	0
stdout	1
stderr	2

Low-level I/O routines set the errno global variable when an error occurs. You must include STDIO.H when you use low-level functions only if your program requires a constant that is defined in STDIO.H, such as the end-of-file indicator (**EOF**).

Low-Level I/O Functions

Function	Use
_close	Close file
_commit	Flush file to disk
_creat, _wcreat	Create file
_dup	Return next available file descriptor for given file
_dup2	Create second descriptor for given file
_eof	Test for end of file
_lseek, _lseeki64	Reposition file pointer to given location
_open, _wopen	Open file
_read	Read data from file
_sopen, _wsopen, _sopen_s, _wsopen_s	Open file for file sharing
_tell, _telli64	Get current file-pointer position
_umask, _umask_s	Set file-permission mask
_write	Write data to file

_dup and **_dup2** are typically used to associate the predefined file descriptors with different files.

See Also

Reference

Input and Output

Run-Time Routines by Category

Console and Port I/O

These routines read and write on your console or on the specified port. The console I/O routines are not compatible with stream I/O or low-level I/O library routines. The console or port does not have to be opened or closed before I/O is performed, so there are no open or close routines in this category. In the Windows operating systems, the output from these functions is always directed to the console and cannot be redirected.

Console and Port I/O Routines

Routine	Use
_cgets, _cgetws, _cgets_s, _cgetws_s	Read string from console
_cprintf, _cwprintf, _cprintf_s, _cprintf_s_l, _cwprintf_s, _cwprintf_s_l	Write formatted data to console
_cputs	Write string to console
_cscanf, _cwscanf, _cscanf_s, _cscanf_s_l, _cwscanf_s, _cwscanf_s_l	Read formatted data from console
_getch, _getwch	Read character from console
_getche, _getwche	Read character from console and echo it
_inp	Read one byte from specified I/O port
_inpd	Read double word from specified I/O port
_inpw	Read 2-byte word from specified I/O port
_kbhit	Check for keystroke at console; use before attempting to read fro m console
_outp	Write one byte to specified I/O port
_outpd	Write double word to specified I/O port
_outpw	Write word to specified I/O port
_putch, _putwch	Write character to console
_ungetch, _ungetwch	"Unget" last character read from console so it becomes next charac ter read

See Also

Reference

Input and Output

Run-Time Routines by Category

_nolock Functions

These are functions that do not perform any locking. They are provided for users requiring maximum performance. For more information, see Multithreaded Libraries Performance.

Use _nolock functions only if your program is truly single-threaded or if it does its own locking.

_fclose_nolock

_fflush_nolock

_fgetc_nolock, _fgetwc_nolock

_fread_nolock

_fseek_nolock, _fseeki64_nolock

_ftell_nolock, _ftelli64_nolock

_fwrite_nolock

_getc_nolock, _getwc_nolock

_getch_nolock, _getwch_nolock

_getchar_nolock, _getwchar_nolock

_getche_nolock, _getwche_nolock

_getdcwd_nolock, _wgetdcwd_nolock

_putc_nolock, _putwc_nolock

_putch_nolock, _putwch_nolock

_putchar_nolock, _putwchar_nolock

_ungetc_nolock, _ungetwc_nolock

_ungetch_nolock, _ungetwch_nolock

See Also

Reference

Input and Output

Run-Time Routines by Category

Internationalization

The Microsoft run-time library provides many routines that are useful for creating different versions of a program for international markets. This includes locale-related routines, wide-character routines, multibyte-character routines, and generic-text routines. For convenience, most locale-related routines are also categorized in this reference according to the operations they perform. In this section and in the alphabetic reference, multibyte-character routines and wide-character routines are described with single-byte-character counterparts, where they exist.

For information about .NET Framework equivalents, see System::Globalization::CultureInfo Class.

See Also

Reference

Run-Time Routines by Category

Locale

The "Locale" refers to the locality (the Country/Region, and language) for which you can customize certain aspects of your program. Some locale-dependent categories include the formatting of dates and the display format for monetary values. For more information, see Locale Categories.

Use the setlocale function to change or query some or all of the current program or thread locale information while using functions without the **_l** suffix. The functions with the **_l** suffix will use the locale parameter passed in for their locale information during the execution of that specific function only. Use _configthreadlocale to control whether each thread has its own locale, or all threads in a program share the same locale. For more information, see Locales and Code Pages.

More secure versions of the functions in the following table are available, indicated by the **_s** ("secure") suffix. For more information, see Security Enhancements in the CRT.

Locale-Dependent Routines

Routine	Use	setlocale category setting dependence
atof, _atof_l, _wtof, _wtof_l	Convert character to floating-point value	**LC_NUMERIC**
atoi, _atoi_l, _wtoi, _wtoi_l	Convert character to integer value	**LC_NUMERIC**
_atoi64, _atoi64_l, _wtoi64, _wtoi64_l	Convert character to 64-bit integer value	**LC_NUMERIC**
atol, _atol_l, _wtol, _wtol_l	Convert character to long value	**LC_NUMERIC**
_atodbl, _atodbl_l, _atoldbl, _atoldbl_l, _atoflt _atoflt_l	Convert character to double-long value	**LC_NUMERIC**
is Routines	Test given integer for particular condition.	**LC_CTYPE**
isleadbyte, _isleadbyte_l	Test for lead byte	**LC_CTYPE**
localeconv	Read appropriate values for formatting numeric quantities	**LC_MONETARY, LC_NUMERIC**
MB_CUR_MAX	Maximum length in bytes of any multibyte character in current locale (macro defined in STDLIB.H)	**LC_CTYPE**
_mbccpy, _mbccpy_l,_mbccpy_s, _mbccpy_s_l	Copy one multibyte character	**LC_CTYPE**
_mbclen, mblen, _mblen_l	Validate and return number of bytes in multibyte character	**LC_CTYPE**
strlen, strlen_l, wcslen, wcslen_l, _mbslen, _mbslen_l, _mbstrlen, _mbstrlen_l	For multibyte-character strings: validate each character in string; return string length	**LC_CTYPE**
mbstowcs, _mbstowcs_l,mbstowcs_s, _mbstowcs_s_l	Convert sequence of multibyte characters to corresponding sequence of wide characters	**LC_CTYPE**

mbtowc, _mbtowc_l	Convert multibyte character to corresponding wide character	**LC_CTYPE**
printf functions	Write formatted output	**LC_NUMERIC** (determines radix character output)
scanf functions	Read formatted input	**LC_NUMERIC** (determines radix character recognition)
setlocale, _wsetlocale	Select locale for program	Not applicable
strcoll, wcscoll, _mbscoll, _strcoll_l, _wcscoll_l, _mbscoll_l	Compare characters of two strings	**LC_COLLATE**
_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l	Compare two strings without regard to case	**LC_CTYPE**
_stricoll, _wcsicoll, _mbsicoll, _stricoll_l, _wcsicoll_l, _mbsicoll_l	Compare characters of two strings (case insensitive)	**LC_COLLATE**
_strncoll, _wcsncoll, _mbsncoll, _strncoll_l, _wcsncoll_l, _mbsncoll_l	Compare first *n* characters of two strings	**LC_COLLATE**
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l	Compare characters of two strings without regard to case.	**LC_CTYPE**
_strnicoll, _wcsnicoll, _mbsnicoll, _strnicoll_l, _wcsnicoll_l, _mbsnicoll_l	Compare first *n* characters of two strings (case insensitive)	**LC_COLLATE**
strftime, wcsftime, _strftime_l, _wcsftime_l	Format date and time value according to supplied *format* argument	**LC_TIME**
_strlwr, _wcslwr, _mbslwr, _strlwr_l, _wcslwr_l, _mbslwr_l, _strlwr_s, _strlwr_s_l, _wcslwr_s, _wcslwr_s_l, _mbslwr_s, _mbslwr_s_l	Convert, in place, each uppercase letter in given string to lowercase	**LC_CTYPE**
strtod, _strtod_l, wcstod, _wcstod_l	Convert character string to **double** value	**LC_NUMERIC** (determines radix character recognition)
strtol, wcstol, _strtol_l, _wcstol_l	Convert character string to **long** value	**LC_NUMERIC** (determines radix character recognition)
strtoul, _strtoul_l, wcstoul, _wcstoul_l	Convert character string to unsigned long value	**LC_NUMERIC** (determines radix character recognition)
_strupr, _strupr_l, _mbsupr, _mbsupr_l, _wcsupr_l, _wcsupr, _strupr_s, _strupr_s_l, _mbsupr_s, _mbsupr_s_l, _wcsupr_s, _wcsupr_s_l	Convert, in place, each lowercase letter in string to uppercase	**LC_CTYPE**
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l	Transform string into collated form according to locale	**LC_COLLATE**

tolower, _tolower, towlower, _tolower_l, _towlower_l,_mbctolower, _mbctolower_l, _mbctoupper, _mbctoupper_l	Convert given character to corresponding lowercase character	**LC_CTYPE**
toupper, _toupper, towupper, _toupper_l, _towupper_l,_mbctolower, _mbctolower_l, _mbctoupper, _mbctoupper_l	Convert given character to corresponding uppercase letter	**LC_CTYPE**
wcstombs, _wcstombs_l,wcstombs_s, _wcstombs_s_l	Convert sequence of wide characters to corresponding sequence of multibyte characters	**LC_CTYPE**
wctomb, _wctomb_l,wctomb_s, _wctomb_s_l	Convert wide character to corresponding multibyte character	**LC_CTYPE**

Note

For multibyte routines, the multibyte code page must be equivalent to the locale set with setlocale. _setmbcp, with an argument of **_MB_CP_LOCALE** makes the multibyte code page the same as the **setlocale** code page.

See Also

Reference

Internationalization

Run-Time Routines by Category

Code Pages

A **code page** is a character set, which can include numbers, punctuation marks, and other glyphs. Different languages and locales may use different code pages. For example, ANSI code page 1252 is used for English and most European languages; OEM code page 932 is used for Japanese Kanji.

A code page can be represented in a table as a mapping of characters to single-byte values or multibyte values. Many code pages share the ASCII character set for characters in the range 0x00 – 0x7F.

The Microsoft run-time library uses the following types of code pages:

- System-default ANSI code page. By default, at startup the run-time system automatically sets the multibyte code page to the system-default ANSI code page, which is obtained from the operating system. The call:

  ```
  setlocale ( LC_ALL, "" );
  ```

  also sets the locale to the system-default ANSI code page.

- Locale code page. The behavior of a number of run-time routines is dependent on the current locale setting, which includes the locale code page. (For more information, see Locale-Dependent Routines.) By default, all locale-dependent routines in the Microsoft run-time library use the code page that corresponds to the "C" locale. At run-time you can change or query the locale code page in use with a call to setlocale.

- Multibyte code page. The behavior of most of the multibyte-character routines in the run-time library depends on the current multibyte code page setting. By default, these routines use the system-default ANSI code page. At run-time you can query and change the multibyte code page with _getmbcp and _setmbcp, respectively.

- The "C" locale is defined by ANSI to correspond to the locale in which C programs have traditionally executed. The code page for the "C" locale ("C" code page) corresponds to the ASCII character set. For example, in the "C" locale, **islower** returns true for the values 0x61 – 0x7A only. In another locale, **islower** may return true for these as well as other values, as defined by that locale.

See Also
Reference
Internationalization
Run-Time Routines by Category

Interpretation of Multibyte-Character Sequences

Most multibyte-character routines in the Microsoft run-time library recognize multibyte-character sequences relating to a multibyte code page. The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Locale-Dependent Multibyte Routines

Routine	Use
_mbclen, mblen, _mblen_l	Validate and return number of bytes in multibyte character
strlen, strlen_l, wcslen, wcslen_l, _mbslen, _mbslen_l, _mbstrlen, _mbstrlen_l	For multibyte character strings: validate each character in string; return string length. For wide character strings: return string length.
mbstowcs, _mbstowcs_l, mbstowcs_s, _mbstowcs_s_l	Convert sequence of multibyte characters to corresponding sequence of wide characters
mbtowc, _mbtowc_l	Convert multibyte character to corresponding wide character
wcstombs, _wcstombs_l, wcstombs_s, _wcstombs_s_l	Convert sequence of wide characters to corresponding sequence of multibyte characters
wctomb, _wctomb_l, wctomb_s, _wctomb_s_l	Convert wide character to corresponding multibyte character

See Also

Reference

Internationalization

Run-Time Routines by Category

Single-Byte and Multibyte Character Sets

The ASCII character set defines characters in the range 0x00 – 0x7F. There are a number of other character sets, primarily European, that define the characters within the range 0x00 – 0x7F identically to the ASCII character set and also define an extended character set from 0x80 – 0xFF. Thus an 8-bit, single-byte-character set (**SBCS**) is sufficient to represent the ASCII character set as well as the character sets for many European languages. However, some non-European character sets, such as Japanese Kanji, include many more characters than can be represented in a single-byte coding scheme, and therefore require multibyte-character set (**MBCS**) encoding.

Note

Many **SBCS** routines in the Microsoft run-time library handle multibyte bytes, characters, and strings as appropriate. Many m ultibyte-character sets define the ASCII character set as a subset. In many multibyte character sets, each character in the rang e 0x00 – 0x7F is identical to the character that has the same value in the ASCII character set. For example, in both **ASCII** and **MBCS** character strings, the one-byte **NULL** character ('\0') has value 0x00 and indicates the terminating null character.

A multibyte character set may consist of both one-byte and two-byte characters. Thus a multibyte-character string may contain a mixture of single-byte and double-byte characters. A two-byte multibyte character has a lead byte and a trail byte. In a particular multibyte-character set, the lead bytes fall within a certain range, as do the trail bytes. When these ranges overlap, it may be necessary to evaluate the particular context to determine whether a given byte is functioning as a lead byte or a trail byte.

See Also

Reference

Internationalization

Run-Time Routines by Category

SBCS and MBCS Data Types

Any Microsoft **MBCS** run-time library routine that handles only one multibyte character or one byte of a multibyte character expects an **unsigned int** argument (where 0x00 <= character value <= 0xFFFF and 0x00 <= byte value <= 0xFF). An **MBCS** routine that handles multibyte bytes or characters in a string context expects a multibyte-character string to be represented as an **unsigned char** pointer.

Caution
Each byte of a multibyte character can be represented in an 8-bit **char**. However, an **SBCS** or **MBCS** single-byte character of type **char** with a value greater than 0x7F is negative. When such a character is converted directly to an **int** or a **long**, the result is sign-extended by the compiler and can therefore yield unexpected results.

Therefore it is best to represent a byte of a multibyte character as an 8-bit **unsigned char**. Or, to avoid a negative result, simply convert a single-byte character of type **char** to an **unsigned char** before converting it to an **int** or a **long**.

Because some **SBCS** string-handling functions take (signed) **char*** parameters, a type mismatch compiler warning will result when **_MBCS** is defined. There are three ways to avoid this warning, listed in order of efficiency:

1. Use the "type-safe" inline function thunks in TCHAR.H. This is the default behavior.
2. Use the "direct" macros in TCHAR.H by defining **_MB_MAP_DIRECT** on the command line. If you do this, you must manually match types. This is the fastest method but is not type-safe.
3. Use the "type-safe" statically linked library function thunks in TCHAR.H. To do so, define the constant **_NO_INLINING** on the command line. This is the slowest method, but the most type-safe.

See Also
Reference
Internationalization
Run-Time Routines by Category

Unicode: The Wide-Character Set

A wide character is a 2-byte multilingual character code. Any character in use in modern computing worldwide, including technical symbols and special publishing characters, can be represented according to the Unicode specification as a wide character. Developed and maintained by a large consortium that includes Microsoft, the Unicode standard is now widely accepted.

A wide character is of type **wchar_t**. A wide-character string is represented as a **wchar_t[]** array and is pointed to by a **wchar_t*** pointer. You can represent any ASCII character as a wide character by prefixing the letter **L** to the character. For example, L'\0' is the terminating wide (16-bit) **NULL** character. Similarly, you can represent any ASCII string literal as a wide-character string literal simply by prefixing the letter L to the ASCII literal (L"Hello").

Generally, wide characters take up more space in memory than multibyte characters but are faster to process. In addition, only one locale can be represented at a time in multibyte encoding, whereas all character sets in the world are represented simultaneously by the Unicode representation.

See Also

Reference

Internationalization

Run-Time Routines by Category

Using Generic-Text Mappings

Microsoft Specific

To simplify code development for various international markets, the Microsoft run-time library provides Microsoft-specific "generic-text" mappings for many data types, routines, and other objects. These mappings are defined in TCHAR.H. You can use these name mappings to write generic code that can be compiled for any of the three kinds of character sets: ASCII (SBCS), MBCS, or Unicode, depending on a manifest constant you define using a **#define** statement. Generic-text mappings are Microsoft extensions that are not ANSI compatible.

Preprocessor Directives for Generic-Text Mappings

#define	Compiled version	Example
_UNICODE	Unicode (wide-character)	**_tcsrev** maps to **_wcsrev**
_MBCS	Multibyte-character	**_tcsrev** maps to **_mbsrev**
None (the default: neither **_UNICODE** nor **_MBCS** defined)	SBCS (ASCII)	**_tcsrev** maps to **strrev**

For example, the generic-text function **_tcsrev**, defined in TCHAR.H, maps to **mbsrev** if **MBCS** has been defined in your program, or to **_wcsrev** if **_UNICODE** has been defined. Otherwise **_tcsrev** maps to **strrev**.

The generic-text data type **_TCHAR**, also defined in TCHAR.H, maps to type **char** if **_MBCS** is defined, to type **wchar_t** if **_UNICODE** is defined, and to type **char** if neither constant is defined. Other data type mappings are provided in TCHAR.H for programming convenience, but **_TCHAR** is the type that is most useful.

Generic-Text Data Type Mappings

Generic-text data type name	SBCS (_UNICODE, _MBCS not defined)	_MBCS defined	_UNICODE defined
_TCHAR	**char**	**char**	**wchar_t**
_TINT	**int**	**int**	**wint_t**
_TSCHAR	**signed char**	**signed char**	**wchar_t**
_TUCHAR	**unsigned char**	**unsigned char**	**wchar_t**
_TXCHAR	**char**	**unsigned char**	**wchar_t**
_T or **_TEXT**	No effect (removed by preprocessor)	No effect (removed by preprocessor)	**L** (converts following character or string to its Unicode counterpart)

For a complete list of generic-text mappings of routines, variables, and other objects, see Generic-Text Mappings.

The following code fragments illustrate the use of **_TCHAR** and **_tcsrev** for mapping to the MBCS, Unicode, and SBCS models.

```
_TCHAR *RetVal, *szString;
RetVal = _tcsrev(szString);
```

If **MBCS** has been defined, the preprocessor maps the preceding fragment to the following code:

```
char *RetVal, *szString;
RetVal = _mbsrev(szString);
```

If **_UNICODE** has been defined, the preprocessor maps the same fragment to the following code:

```
wchar_t *RetVal, *szString;
RetVal = _wcsrev(szString);
```

If neither **_MBCS** nor **_UNICODE** has been defined, the preprocessor maps the fragment to single-byte ASCII code, as follows:

```
char *RetVal, *szString;
RetVal = strrev(szString);
```

Thus you can write, maintain, and compile a single source code file to run with routines that are specific to any of the three kinds of character sets.

END Microsoft Specific

See Also
Reference
Generic-Text Mappings
Data Type Mappings
Constant and Global Variable Mappings
Routine Mappings
A Sample Generic-Text Program

A Sample Generic-Text Program

Microsoft Specific

The following program, GENTEXT.C, provides a more detailed illustration of the use of generic-text mappings defined in TCHAR.H:

```
// GENTEXT.C
// use of generic-text mappings defined in TCHAR.H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <direct.h>
#include <errno.h>
#include <tchar.h>

int __cdecl _tmain(int argc, _TCHAR **argv, _TCHAR **envp)
{
   _TCHAR buff[_MAX_PATH];
   _TCHAR *str = _T("Astring");
   char *amsg = "Reversed";
   wchar_t *wmsg = L"Is";

#ifdef _UNICODE
   printf("Unicode version\n");
#else /* _UNICODE */
#ifdef _MBCS
   printf("MBCS version\n");
#else
   printf("SBCS version\n");
#endif
#endif /* _UNICODE */

   if (_tgetcwd(buff, _MAX_PATH) == NULL)
       printf("Can't Get Current Directory - errno=%d\n", errno);
   else
       _tprintf(_T("Current Directory is '%s'\n"), buff);
   _tprintf(_T("'%s' %hs %ls:\n"), str, amsg, wmsg);
   _tprintf(_T("'%s'\n"), _tcsrev(str));
   return 0;
}
```

If **_MBCS** has been defined, GENTEXT.C maps to the following MBCS program:

```
// crt_mbcsgtxt.c

/*
 * Use of generic-text mappings defined in TCHAR.H
 * Generic-Text-Mapping example program
 * MBCS version of GENTEXT.C
 */

#include <stdlib.h>
#include <direct.h>

int __cdecl main(int argc, char **argv, char **envp)
{
   char buff[_MAX_PATH];
   char *str = "Astring";
   char *amsg = "Reversed";
   wchar_t *wmsg = L"Is";
```

```
    printf("MBCS version\n");

    if (_getcwd(buff, _MAX_PATH) == NULL) {
        printf("Can't Get Current Directory - errno=%d\n", errno);
    }
    else {
        printf("Current Directory is '%s'\n", buff);
    }

    printf("'%s' %hs %ls:\n", str, amsg, wmsg);
    printf("'%s'\n", _mbsrev(str));
    return 0;
}
```

If **_UNICODE** has been defined, GENTEXT.C maps to the following Unicode version of the program. For more information about using **wmain** in Unicode programs as a replacement for **main**, see Using wmain in *C Language Reference*.

```
// crt_unicgtxt.c

/*
 * Use of generic-text mappings defined in TCHAR.H
 * Generic-Text-Mapping example program
 * Unicode version of GENTEXT.C
 */

#include <stdlib.h>
#include <direct.h>

int __cdecl wmain(int argc, wchar_t **argv, wchar_t **envp)
{
   wchar_t buff[_MAX_PATH];
   wchar_t *str = L"Astring";
   char *amsg = "Reversed";
   wchar_t *wmsg = L"Is";

   printf("Unicode version\n");

   if (_wgetcwd(buff, _MAX_PATH) == NULL) {
      printf("Can't Get Current Directory - errno=%d\n", errno);
   }
   else {
       wprintf(L"Current Directory is '%s'\n", buff);
   }

   wprintf(L"'%s' %hs %ls:\n", str, amsg, wmsg);
   wprintf(L"'%s'\n", wcsrev(str));
   return 0;
}
```

If neither **_MBCS** nor **_UNICODE** has been defined, GENTEXT.C maps to single-byte ASCII code, as follows:

```
// crt_sbcsgtxt.c
/*
 * Use of generic-text mappings defined in TCHAR.H
 * Generic-Text-Mapping example program
 * Single-byte (SBCS) Ascii version of GENTEXT.C
 */

#include <stdlib.h>
#include <direct.h>

int __cdecl main(int argc, char **argv, char **envp)
{
   char buff[_MAX_PATH];
   char *str = "Astring";
   char *amsg = "Reversed";
```

```
    wchar_t *wmsg = L"Is";

    printf("SBCS version\n");

    if (_getcwd(buff, _MAX_PATH) == NULL) {
        printf("Can't Get Current Directory - errno=%d\n", errno);
    }
    else {
        printf("Current Directory is '%s'\n", buff);
    }

    printf("'%s' %hs %ls:\n", str, amsg, wmsg);
    printf("'%s'\n", strrev(str));
    return 0;
}
```

END Microsoft Specific

See Also

Reference

Generic-Text Mappings
Data Type Mappings
Constant and Global Variable Mappings
Routine Mappings
Using Generic-Text Mappings

Using TCHAR.H Data Types with _MBCS

Microsoft Specific

As the table of generic-text routine mappings indicates (see Generic-Text Mappings), when the manifest constant **_MBCS** is defined, a given generic-text routine maps to one of the following kinds of routines:

- An SBCS routine that handles multibyte bytes, characters, and strings appropriately. In this case, the string arguments are expected to be of type **char***. For example, **_tprintf** maps to **printf**; the string arguments to **printf** are of type **char***. If you use the **_TCHAR** generic-text data type for your string types, the formal and actual parameter types for **printf** match because **_TCHAR*** maps to **char***.
- An MBCS-specific routine. In this case, the string arguments are expected to be of type **unsigned char***. For example, **_tcsrev** maps to **_mbsrev**, which expects and returns a string of type **unsigned char***. Again, if you use the **_TCHAR** generic-text data type for your string types, there is a potential type conflict because **_TCHAR** maps to type **char**.

Following are three solutions for preventing this type conflict (and the C compiler warnings or C++ compiler errors that would result):

- Use the default behavior. TCHAR.H provides generic-text routine prototypes for routines in the run-time libraries, as in the following example.

  ```
  char *_tcsrev(char *);
  ```

  In the default case, the prototype for **_tcsrev** maps to **_mbsrev** through a thunk in LIBC.LIB. This changes the types of the **_mbsrev** incoming parameters and outgoing return value from **_TCHAR *** (such as **char ***) to **unsigned char ***. This method ensures type matching when you are using **_TCHAR**, but it is relatively slow because of the function call overhead.

- Use function inlining by incorporating the following preprocessor statement in your code.

  ```
  #define _USE_INLINING
  ```

  This method causes an inline function thunk, provided in TCHAR.H, to map the generic-text routine directly to the appropriate MBCS routine. The following code excerpt from TCHAR.H provides an example of how this is done.

  ```
  __inline char *_tcsrev(char *_s1)
  {return (char *)_mbsrev((unsigned char *)_s1);}
  ```

  If you can use inlining, this is the best solution, because it guarantees type matching and has no additional time cost.

- Use "direct mapping" by incorporating the following preprocessor statement in your code.

  ```
  #define _MB_MAP_DIRECT
  ```

  This approach provides a fast alternative if you do not want to use the default behavior or cannot use inlining. It causes the generic-text routine to be mapped by a macro directly to the MBCS version of the routine, as in the following example from TCHAR.H.

  ```
  #define _tcschr _mbschr
  ```

When you take this approach, you must be careful to ensure that appropriate data types are used for string arguments and string return values. You can use type casting to ensure proper type matching or you can use the **_TXCHAR** generic-text data type. **_TXCHAR** maps to type **char** in SBCS code but maps to type **unsigned char** in MBCS code. For more information about generic-text macros, see Generic-Text Mappings.

END Microsoft Specific

See Also

Reference

Internationalization

Run-Time Routines by Category

Memory Allocation

Use these routines to allocate, free, and reallocate memory.

Memory-Allocation Routines

Routine	Use	.NET Framework equivalent
_alloca, _malloca	Allocate memory from stack	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
calloc	Allocate storage for array, initializing every byte in allocated block to 0	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_calloc_dbg	Debug version of **calloc**; only available in the debug versions of the run-time libraries	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
operator delete	Free allocated block	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
operator delete[]	Free allocated block	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_expand	Expand or shrink block of memory without moving it	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_expand_dbg	Debug version of **_expand**; only available in the debug versions of the run-time libraries	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
free	Free allocated block	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_free_dbg	Debug version of **free**; only available in the debug versions of the run-time libraries	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_freea	Free allocated block from stack	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_get_heap_handle	Get Win32 HANDLE of the CRT heap.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_get_sbh_threshold	Return the upper limit for the size of a memory allocation that will be supported by the small-block heap	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

_heapadd	Add memory to heap	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_heapchk	Check heap for consistency	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_heapmin	Release unused memory in heap	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_heapset	Fill free heap entries with specified value	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_heapwalk	Return information about each entry in heap	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
malloc	Allocate block of memory from heap	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_malloc_dbg	Debug version of **malloc**; only available in the debug versions of the run-time libraries	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_msize	Return size of allocated block	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_msize_dbg	Debug version of **_msize**; only available in the debug versions of the run-time libraries	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
new	Allocate block of memory from heap	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
new[]	Allocate block of memory from heap	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_query_new_handler	Return address of current new handler routine as set by **_set_new_handler**	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_query_new_mode	Return integer indicating new handler mode set by **_set_new_mode** for **malloc**	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
realloc	Reallocate block to new size	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

_realloc_dbg	Debug version of **realloc**; only available in the debug versions of the run-time libraries	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_set_new_handler	Enable error-handling mechanism when **new** operator fails (to allocate memory) and enable compilation of Standard Template Libraries (STL)	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_set_new_mode	Set new handler mode for **malloc**	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_set_sbh_threshold	Set the upper limit for the size of a memory allocation that will be supported by the small-block heap	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Run-Time Routines by Category

Process and Environment Control

Use the process-control routines to start, stop, and manage processes from within a program. Use the environment-control routines to get and change information about the operating-system environment.

Process and Environment Control Functions

Routine	Use	.NET Framework equivalent
abort	Abort process without flushing buffers or calling functions registered by **atexit** and **_onexit**	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
assert	Test for logic error	System::Diagnostics::Debug::Assert
_ASSERT, _ASSERTE macros	Similar to **assert**, but only available in the debug versions of the run-time libraries	System::Diagnostics::Debug::Assert
atexit	Schedule routines for execution at program termination	System::Diagnostics::Process::Exited
_beginthread, _beginthreadex	Create a new thread on a Windows operating system process	System::Threading::Thread::Start
_cexit	Perform **exit** termination procedures (such as flushing buffers), then return control to calling program without terminating process	System::Diagnostics::Process::CloseMainWindow
_c_exit	Perform **_exit** termination procedures, then return control to calling program without terminating process	System::Diagnostics::Process::CloseMainWindow
_cwait	Wait until another process terminates	System::Diagnostics::Process::WaitForExit
_endthread, _endthreadex	Terminate a Windows operating system thread	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_execl, _wexecl	Execute new process with argument list	System::Diagnostics::Process Class, System::Diagnostics::ProcessStartInfo Class
_execle, _wexecle	Execute new process with argument list and given environment	System::Diagnostics::Process Class, System::Diagnostics::ProcessStartInfo Class
_execlp, _wexeclp	Execute new process using **PATH** variable and argument list	System::Diagnostics::Process Class, System::Diagnostics::ProcessStartInfo Class
_execlpe, _wexeclpe	Execute new process using **PATH** variable, given environment, and argument list	System::Diagnostics::Process Class, System::Diagnostics::ProcessStartInfo Class
_execv, _wexecv	Execute new process with argument array	System::Diagnostics::Process Class, System::Diagnostics::ProcessStartInfo Class
_execve, _wexecve	Execute new process with argument array and given environment	System::Diagnostics::Process Class, System::Diagnostics::ProcessStartInfo Class

_execvp, _wexecvp	Execute new process using **PATH** variable and argument array	System::Diagnostics::Process Class, System::Diagnostics::ProcessStartInfo Class
_execvpe, _wexecvpe	Execute new process using **PATH** variable, given environment, and argument array	System::Diagnostics::Process Class, System::Diagnostics::ProcessStartInfo Class
exit	Call functions registered by **atexit** and **_onexit**, flush all buffers, close all open files, and terminate process	System::Diagnostics::Process::Kill
_exit	Terminate process immediately without calling **atexit** or **_onexit** or flushing buffers	System::Diagnostics::Process::Kill
getenv, _wgetenv, getenv_s, _wgetenv_s	Get value of environment variable	System::Environment::GetEnvironmentVariable
_getpid	Get process ID number	System::Diagnostics::Process::Id
longjmp	Restore saved stack environment; use it to execute a nonlocal **goto**	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_onexit	Schedule routines for execution at program termination; use for compatibility with Microsoft C/C++ version 7.0 and earlier	System::Diagnostics::Process::Exited
_pclose	Wait for new command processor and close stream on associated pipe	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
perror, _wperror	Print error message	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_pipe	Create pipe for reading and writing	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_popen, _wpopen	Create pipe and execute command	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_putenv, _wputenv, _putenv_s, _wputenv_s	Add or change value of environment variable	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
raise	Send signal to calling process	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
setjmp	Save stack environment; use to execute nonlocal **goto**	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

signal	Handle interrupt signal	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_spawnl, _wspawnl	Create and execute new process with specified argument list	System::Diagnostics::Process Class, System::Diagnostics::ProcessStartInfo Class
_spawnle, _wspawnle	Create and execute new process with specified argument list and environment	System::Diagnostics::Process Class, System::Diagnostics::ProcessStartInfo Class
_spawnlp, _wspawnlp	Create and execute new process using **PATH** variable and specified argument list	System::Diagnostics::Process Class, System::Diagnostics::ProcessStartInfo Class
_spawnlpe, _wspawnlpe	Create and execute new process using **PATH** variable, specified environment, and argument list	System::Diagnostics::Process Class, System::Diagnostics::ProcessStartInfo Class
_spawnv, _wspawnv	Create and execute new process with specified argument array	System::Diagnostics::Process Class, System::Diagnostics::ProcessStartInfo Class
_spawnve, _wspawnve	Create and execute new process with specified environment and argument array	System::Diagnostics::Process Class, System::Diagnostics::ProcessStartInfo Class
_spawnvp, _wspawnvp	Create and execute new process using **PATH** variable and specified argument array	System::Diagnostics::Process Class, System::Diagnostics::ProcessStartInfo Class
_spawnvpe, _wspawnvpe	Create and execute new process using **PATH** variable, specified environment, and argument array	System::Diagnostics::Process Class, System::Diagnostics::ProcessStartInfo Class
system, _wsystem	Execute operating-system command	System::Diagnostics::ProcessStartInfo Class, System::Diagnostics::Process Class

In the Windows operating system, the spawned process is equivalent to the spawning process. Any process can use **_cwait** to wait for any other process for which the process ID is known.

The difference between the **_exec** and **_spawn** families is that a **_spawn** function can return control from the new process to the calling process. In a **_spawn** function, both the calling process and the new process are present in memory unless **_P_OVERLAY** is specified. In an **_exec** function, the new process overlays the calling process, so control cannot return to the calling process unless an error occurs in the attempt to start execution of the new process.

The differences among the functions in the **_exec** family, as well as among those in the **_spawn** family, involve the method of locating the file to be executed as the new process, the form in which arguments are passed to the new process, and the method of setting the environment, as shown in the following table. Use a function that passes an argument list when the number of arguments is constant or is known at compile time. Use a function that passes a pointer to an array containing the arguments when the number of arguments is to be determined at run time. The information in the following table also applies to the wide-character counterparts of the **_spawn** and **_exec** functions.

_spawn and _exec Function Families

Functions	Use PATH variable to locate file	Argument-passing convention	Environment settings
_execl, _spawnl	No	List	Inherited from calling process
_execle, _spawnle	No	List	Pointer to environment table for new process passed as last argument
_execlp, _spawnlp	Yes	List	Inherited from calling process

_execlpe, _spawnlpe	Yes	List	Pointer to environment table for new process passed as last argument
_execv, _spawnv	No	Array	Inherited from calling process
_execve, _spawnve	No	Array	Pointer to environment table for new process passed as last argument
_execvp, _spawnvp	Yes	Array	Inherited from calling process
_execvpe, _spawnvpe	Yes	Array	Pointer to environment table for new process passed as last argument

See Also

Reference

Run-Time Routines by Category

Robustness

Use the following C run-time library functions to improve the robustness of your program.

Run-Time Robustness Functions

Function	Use	.NET Framework equivalent
_set_new_handler	Transfers control to your error-handling mechanism if the **new** operator fails to allocate memory.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_set_se_translator	Handles Win32 exceptions (C structured exceptions) as C++ typed exceptions.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
set_terminate	Installs your own termination function to be called by terminate.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
set_unexpected	Installs your own termination function to be called by unexpected.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Run-Time Routines by Category

SetUnhandledExceptionFilter

Run-Time Error Checking

The C run-time library contains the functions that support run-time error checks (RTC). Run-time error checking allows you to build your program such that certain kinds of run-time errors are reported. You specify how the errors are reported and which kinds of errors are reported. For more information, see Run-Time Error Checks.

Use the following functions to customize the way your program does run-time error checking.

Run-Time Error Checking Functions

Function	Use	.NET Framework equivalent
_RTC_GetErrDesc	Returns a brief description of a run-time error check type.	
_RTC_NumErrors	Returns the total number of errors that can be detected by run-time error checks.	
_RTC_SetErrorFunc	Designates a function as the handler for reporting run-time error checks.	
_RTC_SetErrorType	Associates an error that is detected by run-time error checks with a type.	

See Also

Tasks

RTC Sample: Run-Time Error Checks

Reference

Run-Time Routines by Category
/RTC (Run-Time Error Checks)
runtime_checks
Debug Routines

Searching and Sorting

Use the following functions for searching and sorting.

Searching and Sorting Functions

Function	Search or Sort	.NET Framework equivalent
bsearch	Binary search	System::Collections::ArrayList::BinarySearch
bsearch_s	A more secure version of **bsearch**.	System::Collections::ArrayList::BinarySearch
_lfind	Linear search for given value	System::Collections::ArrayList::Contains
_lfind_s	A more secure version of **_lfind**	System::Collections::ArrayList::Contains
_lsearch	Linear search for given value, which is ad ded to array if not found	Not applicable. To call the standard C function, use **PInvoke**. For more in formation, see Platform Invoke Examples.
_lsearch_s	A more secure version of **_lsearch**	Not applicable. To call the standard C function, use **PInvoke**. For more in formation, see Platform Invoke Examples.
qsort	Quick sort	System::Collections::ArrayList::Sort
qsort_s	A more secure version of **qsort**	System::Collections::ArrayList::Sort

See Also

Reference

Run-Time Routines by Category

String Manipulation (CRT)

These routines operate on null-terminated single-byte character, wide-character, and multibyte-character strings. Use the buffer-manipulation routines, described in Buffer Manipulation, to work with character arrays that do not end with a null character.

String-Manipulation Routines

Routine	Use	.NET Framework equivalent
strcoll, wcscoll, _mbscoll, _strcoll_l, _wcscoll_l, _mbscoll_l, _stricoll, _wcsicoll, _mbsicoll, _stricoll_l, _wcsicoll_l, _mbsicoll_l, _strncoll, _wcsncoll, _mbsncoll, _strncoll_l, _wcsncoll_l, _mbsncoll_l, _strnicoll, _wcsnicoll, _mbsnicoll, _strnicoll_l, _wcsnicoll_l, _mbsnicoll_l	Compare two character strings using code page information (**_mbsicoll** and **_mbsnicoll** are case-insensitive)	System::String::Compare
_mbsdec, _mbsdec_l, _strdec, _wcsdec	Move string pointer back one character	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_mbsinc, _mbsinc_l, _strinc, _wcsinc	Advance string pointer by one character	Not applicable.
_mbsnbcat, _mbsnbcat_l, _mbsnbcat_s, _mbsnbcat_s_l	Append, at most, first *n* bytes of one character string to another	Not applicable.
_mbsnbcmp, _mbsnbcmp_l	Compare first *n* bytes of two character strings	Not applicable.

_mbsnbcnt, _mbsnbcnt_l, _mbsnccnt, _mbsnccnt_l, _strncnt, _wcsncnt	Return number of character bytes within supplied character count	Not applicable.
_mbsnbcpy, _mbsnbcpy_l, _mbsnbcpy_s, _mbsnbcpy_s_l	Copy *n* bytes of string	Not applicable.
_mbsnbicmp, _mbsnbicmp_l	Compare *n* bytes of two character strings, ignoring case	Not applicable.
_mbsnbset, _mbsnbset_l	Set first *n* bytes of character string to specified character	Not applicable.
_mbsnbcnt, _mbsnbcnt_l, _mbsnccnt, _mbsnccnt_l, _strncnt, _wcsncnt	Return number of characters within supplied byte count	Not applicable.
_mbsnextc, _mbsnextc_l, _strnextc, _wcsnextc	Find next character in string	Not applicable.
_mbsninc, _mbsninc_l, _strninc, _wcsninc	Advance string pointer by *n* characters	Not applicable.

_mbsspnp, _mbsspnp_l, _strspnp, _wcsspnp	Return pointer to first character in given string that is not in another given string	Not applicable.
_scprintf, _scprintf_l, _scwprintf, _scwprintf_l	Return the number of characters in a formatted string	Not applicable.
_snscanf, _snscanf_l, _snwscanf, _snwscanf_l, _snscanf_s, _snscanf_s_l, _snwscanf_s, _snwscanf_s_l	Read formatted data of a specified length from the standard input stream.	Not applicable.
sscanf, _sscanf_l, swscanf, _swscanf_l, sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l	Read formatted data of a specified length from the standard input stream.	Not applicable.
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l, sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l, _sprintf_p, _sprintf_p_l, _swprintf_p, _swprintf_p_l	Write formatted data to a string	System::String::Format
strcat, wcscat, _mbscat, strcat_s, wcscat_s, _mbscat_s	Append one string to another	System::String::Concat

strchr, wcschr, _mbschr, _mbschr_l	Find first occurrence of specified character in string	System::String::IndexOf
strcmp, wcscmp, _mbscmp	Compare two strings	System::String::CompareOrdinal
strcoll, wcscoll, _mbscoll, _strcoll_l, _wcscoll_l, _mbscoll_l, _stricoll, _wcsicoll, _mbsicoll, _stricoll_l, _wcsicoll_l, _mbsicoll_l, _strncoll, _wcsncoll, _mbsncoll, _strncoll_l, _wcsncoll_l, _mbsncoll_l, _strnicoll, _wcsnicoll, _mbsnicoll, _strnicoll_l, _wcsnicoll_l, _mbsnicoll_l	Compare two strings using current locale code page information (**_stricoll**, **_wcsicoll**, **_strnicoll**, and **_wcsnicoll** are case-insensitive)	System::String::Compare
strcpy, wcscpy, _mbscpy, strcpy_s, wcscpy_s, _mbscpy_s	Copy one string to another	System::String::Copy
strcspn, wcscspn, _mbscspn, _mbscspn_l	Find first occurrence of character from specified character set in string	System::String::Substring
_strdup, _wcsdup, _mbsdup, _strdup_dbg, _wcsdup_dbg	Duplicate string	System::String::Clone
strerror, _strerror, _wcserror, __wcserror, strerror_s, _strerror_s, _wcserror_s, __wcserror_s	Map error number to message string	System::Exception::Message

strftime, wcsftime, _strftime_l, _wcsftime_l	Format date-and-time string	System::Convert::ToString
_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l	Compare two strings without regard to case	System::String::Compare
strlen, strlen_l, wcslen, wcslen_l, _mbslen, _mbslen_l, _mbstrlen, _mbstrlen_l, strnlen, strnlen_l, wcsnlen, wcsnlen_l, _mbsnlen, _mbsnlen_l, _mbstrnlen, _mbstrnlen_l	Find length of string	System::String::Length
_strlwr, _wcslwr, _mbslwr, _strlwr_l, _wcslwr_l, _mbslwr_l, _strlwr_s, _strlwr_s_l, _mbslwr_s, _mbslwr_s_l, _wcslwr_s, _wcslwr_s_l	Convert string to lowercase	System::String::ToLower
strncat, _strncat_l, wcsncat, wcsncat_l, _mbsncat _mbsncat_l, strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l	Append characters of string	System::String::Concat
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l	Compare characters of two strings	System::String::Compare
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l, strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l	Copy characters of one string to another	System::String::Copy
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l	Compare characters of two strings without regard to case	System::String::Compare
_strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l	Set first *n* characters of string to specified character	System::String::Replace

strpbrk, wcspbrk, _mbspbrk, _mbspbrk_l	Find first occurrence of character from one string in another string	System::String::IndexOfAny
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l	Find last occurrence of given character in string	System::String::LastIndexOf
_strrev, _wcsrev, _mbsrev, _mbsrev_l	Reverse string	Not applicable.
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l	Set all characters of string to specified character	Not applicable.
strspn, wcsspn, _mbsspn, _mbsspn_l	Find first substring from one string in another string	System::String::Substring
strstr, wcsstr, _mbsstr, _mbsstr_l	Find first occurrence of specified string in another string	System::String::IndexOf
strtok, _strtok_l, wcstok, _wcstok_l, _mbstok, _mbstok_l, strtok_s, _strtok_s_l, wcstok_s, _wcstok_s_l, _mbstok_s, _mbstok_s_l	Find next token in string	Not applicable.
_strupr, _strupr_l, _mbsupr, _mbsupr_l, _wcsupr_l, _wcsupr, _strupr_s, _strupr_s_l, _mbsupr_s, _mbsupr_s_l, _wcsupr_s, _wcsupr_s_l	Convert string to uppercase	System::String::ToUpper

strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l	Transform string into collated form based on locale-specific information	Not applicable.
vsprintf, _vsprintf_l, vswprintf, _vswprintf_l, __vswprintf_l, vsprintf_s, _vsprintf_s_l, vswprintf_s, _vswprintf_s_l, _vsprintf_p, _vsprintf_p_l, _vswprintf_p, _vswprintf_p_l	Write formatted output using a pointer to a list of arguments	System::String::Format
vsnprintf, _vsnprintf, _vsnprintf_l, _vsnwprintf, _vsnwprintf_l, vsnprintf_s, _vsnprintf_s, _vsnprintf_s_l, _vsnwprintf_s, _vsnwprintf_s_l	Write formatted output using a pointer to a list of arguments	System::String::Format

See Also

Reference

Run-Time Routines by Category

System Calls

The following functions are Windows operating system calls.

System Call Functions

Function	Use	.NET Framework equivalent
_findclose	Release resources from previous find operations	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_findfirst, _findfirst32, _findfirst64, _findfirsti64, _findfirst32i64, _findfirst64i32, _wfindfirst, _wfindfirst32, _wfindfirst64, _wfindfirsti64, _wfindfirst32i64, _wfindfirst64i32	Find file with specified attributes	System::IO::DirectoryInfo::GetFiles
_findnext, _findnext32, _findnext64, _findnexti64, _findnext32i64, _findnext64i32, _wfindnext, _wfindnext32, _wfindnexti64, _wfindnext64, _wfindnexti64	Find next file with specified attributes	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Run-Time Routines by Category

Time Management

Use these functions to get the current time and convert, adjust, and store it as necessary. The current time is the system time.

The **_ftime** and **localtime** routines use the **TZ** environment variable. If **TZ** is not set, the run-time library attempts to use the time-zone information specified by the operating system. If this information is unavailable, these functions use the default value of PST8PDT. For more information on **TZ**, see _tzset; also see _daylight, timezone, and _tzname.

Time Routines

Function	Use	.NET Framework equivalent
asctime, _wasctime, asctime_s, _wasctime_s	Convert time from type **struct tm** to character string. The versions of these functions with the **_s** suffix are more secure.	System::DateTime::ToLongDateString, System::DateTime::ToLongTimeString, System::DateTime::ToShortDateString, System::DateTime::ToShortTimeString, System::DateTime::ToString
clock	Return elapsed CPU time for process.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64, _ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s	Convert time from type **time_t**, **__time32_t** or **__time64_t** to character string. The versions of these functions with the **_s** suffix are more secure.	System::DateTime::GetDateTimeFormats, System::DateTime::ToString, System::DateTime::ToLongTimeString, System::DateTime::ToShortTimeString
difftime, _difftime32, _difftime64	Compute difference between two times.	System::DateTime::Subtract
_ftime, _ftime32, _ftime64,_ftime_s, _ftime32_s, _ftime64_s	Store current system time in variable of type **struct _timeb** or type **struct __timeb64** The versions of these functions with the **_s** suffix are more secure.	System::DateTime::Now
_futime, _futime32, _futime64	Set modification time on open file	System::IO::File::SetLastAccessTime, System::IO::File::SetLastWriteTime, System::IO::File::SetCreationTime

gmtime, _gmtime32, _gmtime64, _gmtime_s, _gmtime32_s, _gmtime64_s	Convert time from type **time_t** to **struct tm** or from type **__time64_t** to **struct tm**. The versions of these functions with the **_s** suffix are more secure.	System::DateTime::UtcNow, System::DateTime::ToUniversalTime
localtime, _localtime32, _localtime64, localtime_s, _localtime32_s, _localtime64_s	Convert time from type **time_t** to **struct tm** or from type **__time64_t** to **struct tm** with local correction. The versions of these functions with the **_s** suffix are more secure.	System::DateTime::ToLocalTime
_mkgmtime, _mkgmtime32, _mkgmtime64	Convert time to calendar value in Greenwich Mean Time.	System::DateTime::ToUniversalTime
mktime, _mktime32, _mktime64	Convert time to calendar value.	System::DateTime::DateTime
_strdate, _wstrdate, _strdate_s, _wstrdate_s	Return current system date as string. The versions of these functions with the **_s** suffix are more secure.	System::DateTime::Parse
strftime, wcsftime, _strftime_l, _wcsftime_l	Format date-and-time string for international use.	System::DateTime::ToLongDateString, System::DateTime::ToLongTimeString, System::DateTime::ToShortDateString, System::DateTime::ToShortTimeString, System::DateTime::ToString

_strtime, _wstrtime, _strtime_s, _wstrtime_s	Return current system time as string. The versions of these functions with the _**s** suffix are more secure.	System::DateTime::ToLongDateString, System::DateTime::ToLongTimeString, System::DateTime::ToShortDateString, System::DateTime::ToShortTimeString, System::DateTime::ToString
time, _time32, _time64	Get current system time as type **time_t**, **__time32_t** or as type **__time64_t**.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_tzset	Set external time variables from environment time variable **TZ**.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.
_utime, _utime32, _utime64, _wutime, _wutime32, _wutime64	Set modification time for specified file using either current time or time value stored in structure.	Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

Note

In all versions of Microsoft C/C++ except Microsoft C/C++ version 7.0, and in all versions of Microsoft Visual C++ 2005, the **time** function returns the current time as the number of seconds elapsed since midnight on January 1, 1970. In Microsoft C/C++ version 7.0, **time** returned the current time as the number of seconds elapsed since midnight on December 31, 1899.

Note

In versions of Visual C++ and Microsoft C/C++ before Visual C++ 2005, **time_t** was a **long int** (32 bits) and hence could not be used for dates past 3:14:07 January 19, 2038, UTC. In Visual C++ 2005, **time_t** is equivalent to **__time64_t** by default, but defining **_USE_32BIT_TIME_T** changes **time_t** to **__time32_t** and forces many time functions to call versions that take the 32-bit **time_t**. For more information, see Standard Types and comments in the documentation for the individual time functions.

See Also

Reference

Run-Time Routines by Category

Global Variables and Standard Types

The Microsoft run-time library contains definitions for global variables, control flags, and standard types used by library routines. Access these variables, flags, and types by declaring them in your program or by including the appropriate header files.

See Also

Reference

Global Constants

Other Resources

Run-Time Library Reference

Global Variables

The Microsoft run-time library provides the following global variables. These global variables have been deprecated in Visual C++ 2005 for the more secure functional versions, which should be used in place of the global variable (except **_pgmptr** and **_wpgmptr**).

Variable	Description
_amblksiz	Controls memory heap granularity
daylight, _timezone, _tzname	Adjust for local time; used in some date and time functions
_doserrno, errno, _sys_errlist, _sys_nerr	Store error codes and related information
_environ, _wenviron	Pointers to arrays of pointers to strings that constitute process environment
_fileinfo	Specifies whether information regarding open files of a process is passed to new processes
_fmode	Sets default file-translation mode
_osver, _winmajor, _winminor, _winver	Store build and version numbers of operating system
_pgmptr, _wpgmptr	Initialized at program startup to value such as program name, filename, relative path, or full path

See Also

Reference

Global Constants
_get_amblksiz
_set_amblksiz
_get_daylight
_get_dstbias
_get_timezone
_get_tzname
perror
strerror
_get_doserrno
_set_doserrno
_get_errno
_set_errno
getenv, _wgetenv
getenv_s, _wgetenv_s
_putenv, _wputenv
_putenv_s, _wputenv_s
_get_fmode
_set_fmode
_get_osver
_get_winmajor
_get_winminor
_get_winver

Other Resources

Run-Time Library Reference

_amblksiz

Controls memory heap granularity. This global variable has been deprecated in Visual C++ 2005 for the more secure functional versions _get_amblksiz and _set_amblksiz, which should be used in place of the global variable. It is declared in Malloc.h as:

```
extern unsigned int _amblksiz;
```

Remarks

The value of **_amblksiz** specifies the size of blocks allocated by the operating system for the heap. The initial requested size for a segment of heap memory is just enough to satisfy the current allocation request (for example, a call to **malloc**) plus memory required for heap manager overhead. The value of **_amblksiz** should represent a trade-off between the number of times the operating system is to be called to increase the heap to the required size and the amount of memory potentially wasted (available but not used) at the end of the heap.

The default value of **_amblksiz** is 8K. You can change this value by using the set function in your program. For example:

```
_set_amblksiz(2045);
```

If you assign a value to **_amblksiz**, the actual value used internally by the heap manager is the assigned value rounded up to the nearest whole power of 2. Thus, in the previous example, the heap manager would reset the value of **_amblksize** to 2048.

See Also

Reference

Global Variables
_get_amblksiz
_set_amblksiz

_daylight, _dstbias, _timezone, and _tzname

_daylight, **_dstbias**, **_timezone**, and **_tzname** are used in some time and date routines to make local-time adjustments. These global variables have been deprecated in Visual C++ 2005 for the more secure functional versions, which should be used in place of the global variables.

Global variable	Functional equivalent
_daylight	_get_daylight
_dstbias	_get_dstbias
_timezone	_get_timezone
_tzname	_get_tzname

They are declared in Time.h as follows.

```
extern int _daylight;
extern int _dstbias;
extern long _timezone;
extern char *_tzname[2];
```

Remarks

On a call to **_ftime**, **localtime**, or **_tzset**, the values of **_daylight**, **_dstbias**, **_timezone**, and **_tzname** are determined from the value of the **TZ** environment variable. If you do not explicitly set the value of **TZ**, **_tzname[0]** and **_tzname[1]** contain the default settings of "PST" and "PDT" respectively. The time-manipulation functions (_tzset, _ftime, and localtime) attempt to set the values of **_daylight**, **_dstbias** and **_timezone** by querying the operating system for the default value of each variable. The time-zone global variable values are shown in the following table.

Variable	Value
_daylight	Nonzero if daylight saving time (DST) zone is specified in **TZ** or determined from the operating system; otherwise, 0. The default value is 1.
_dstbias	Offset for daylight saving time.
_timezone	Difference in seconds between coordinated universal time and local time. The default value is 28,800.
_tzname[0]	Time-zone name derived from the **TZ** environment variable. The default value is "PST".
_tzname[1]	DST zone name derived from the **TZ** environment variable. The default value is "PDT" (Pacific daylight time).

See Also

Reference

Global Variables
_get_daylight
_get_dstbias
_get_timezone
_get_tzname

errno, _doserrno, _sys_errlist, and _sys_nerr

These global variables hold error codes used by the perror and strerror functions for printing error messages. The more secure functional versions listed in this table should be used in place of their global variable.

Global Variable	Functional Equivalents
_doserrno	_get_doserrno, _set_doserrno
_errno	_get_errno, _set_errno

These variables are declared in STDLIB.H as

```
extern int _doserrno;
extern int errno;
extern char *_sys_errlist[ ];
extern int _sys_nerr;
```

Remarks

errno is set on an error in a system-level call. Because **errno** holds the value for the last call that set it, this value may be changed by succeeding calls. Always check **errno** immediately before and after a call that may set it.

On an error, **errno** is not necessarily set to the same value as the error code returned by a system call. For I/O operations, use **_doserrno** to access the operating-system error-code equivalents of **errno** codes. For most non-I/O operations the value of **_doserrno** is undefined.

Each **errno** value is associated with an error message that can be printed using **perror** or stored in a string using **strerror**. **perror** and **strerror** use the **_sys_errlist** array and **_sys_nerr**, the number of elements in **_sys_errlist**, to process error information.

Library math routines set **errno** by calling _matherr. To handle math errors differently, write your own routine according to the **_matherr** reference description and name it **_matherr**.

All **errno** values, defined as predefined constants in ERRNO.H, are UNIX-compatible and are listed below. Only **ERANGE**, **EILSEQ** and **EDOM** are specified in the ANSI standard.

Constant	System error message	Value
EPERM	Operation not permitted	1
ENOENT	No such file or directory	2
ESRCH	No such process	3
EINTR	Interrupted function	4
EIO	I/O error	5
ENXIO	No such device or address	6
E2BIG	Argument list too long	7
ENOEXEC	Exec format error	8
EBADF	Bad file number	9
ECHILD	No spawned processes	10

EAGAIN	No more processes or not enough memory or maximum nesting level reached	11
ENOMEM	Not enough memory	12
EACCES	Permission denied	13
EFAULT	Bad address	14
EBUSY	Device or resource busy	16
EEXIST	File exists	17
EXDEV	Cross-device link	18
ENODEV	No such device	19
ENOTDIR	Not a directory	20
EISDIR	Is a directory	21
EINVAL	Invalid argument	22
ENFILE	Too many files open in system	23
EMFILE	Too many open files	24
ENOTTY	Inappropriate I/O control operation	25
EFBIG	File too large	27
ENOSPC	No space left on device	28
ESPIPE	Invalid seek	29
EROFS	Read-only file system	30
EMLINK	Too many links	31
EPIPE	Broken pipe	32
EDOM	Math argument	33
ERANGE	Result too large	34
EDEADLK	Resource deadlock would occur	36
EDEADLOCK	Same as EDEADLK for compatibility with older Microsoft C versions	36
ENAMETOOLONG	Filename too long	38
ENOLCK	No locks available	39
ENOSYS	Function not supported	40

ENOTEMPTY	Directory not empty	41
EILSEQ	Illegal byte sequence	42
STRUNCATE	String was truncated	80

See Also

Reference

Global Variables
errno Constants
perror
strerror
_get_doserrno
_set_doserrno
_get_errno
_set_errno

_environ, _wenviron

The **_environ** variable is a pointer to an array of pointers to the multibyte-character strings that constitute the process environment. This global variable has been deprecated in Visual C++ 2005 for the more secure functional versions getenv_s, _wgetenv_s and _putenv_s, _wputenv_s, which should be used in place of the global variable. **_environ** is declared in Stdlib.h.

```
extern char **_environ;
```

Remarks

In a program that uses the **main** function, **_environ** is initialized at program startup according to settings taken from the operating-system environment. The environment consists of one or more entries of the form

ENVVARNAME =*string*

getenv_s and **putenv_s** use the **_environ** variable to access and modify the environment table. When **_putenv** is called to add or delete environment settings, the environment table changes size. Its location in memory may also change, depending on the program's memory requirements. The value of **_environ** is automatically adjusted accordingly.

The **_wenviron** variable, declared in Stdlib.h as:

```
extern wchar_t **_wenviron;
```

is a wide-character version of **_environ**. In a program that uses the **wmain** function, **_wenviron** is initialized at program startup according to settings taken from the operating-system environment.

In a program that uses **main**, **_wenviron** is initially **NULL** because the environment is composed of multibyte-character strings. On the first call to **_wgetenv** or **_wputenv**, a corresponding wide-character string environment is created and is pointed to by **_wenviron**.

Similarly, in a program that uses **wmain**, **_environ** is initially **NULL** because the environment is composed of wide-character strings. On the first call to **_getenv** or **_putenv**, a corresponding multibyte-character string environment is created and is pointed to by **_environ**.

When two copies of the environment (MBCS and Unicode) exist simultaneously in a program, the run-time system must maintain both copies, resulting in slower execution time. For example, whenever you call **_putenv**, a call to **_wputenv** is also executed automatically, so that the two environment strings correspond.

Caution
In rare instances, when the run-time system is maintaining both a Unicode version and a multibyte version of the environment, these two environment versions might not correspond exactly. This is because, although any unique multibyte-character string maps to a unique Unicode string, the mapping from a unique Unicode string to a multibyte-character string is not necessarily unique. Therefore, two distinct Unicode strings might map to the same multibyte string.

Polling **_environ** in a Unicode context is meaningless when /MD or **/MDd** linkage is used. For the CRT DLL, the type (wide or multibyte) of the program is unknown. Only the multibyte type is created because that is the most likely scenario.

The following pseudo-code illustrates how this can happen.

```
int i, j;
i = _wputenv( "env_var_x=string1" );  // results in the implicit call:
                                      // putenv ("env_var_z=string1")
j = _wputenv( "env_var_y=string2" );  // also results in implicit call:
                                      // putenv("env_var_z=string2")
```

In the notation used for this example, the character strings are not C string literals; rather, they are placeholders that represent Unicode environment string literals in the **_wputenv** call and multibyte environment strings in the **putenv** call. The character placeholders '*x*' and '*y*' in the two distinct Unicode environment strings do not map uniquely to characters in the current MBCS. Instead, both map to some MBCS character '*z*' that is the default result of the attempt to convert the strings.

Thus, in the multibyte environment, the value of "*env_var_z*" after the first implicit call to **putenv** would be "*string1*", but this value would be overwritten on the second implicit call to **putenv**, when the value of "*env_var_z*" is set to "*string2*". The Unicode environment (in **_wenviron**) and the multibyte environment (in **_environ**) would therefore differ following this series of calls.

See Also

Reference

Global Variables
getenv, _wgetenv
getenv_s, _wgetenv_s
_putenv, _wputenv
_putenv_s, _wputenv_s

_fileinfo

Starting with Visual C++ 2005, the global variable **_fileinfo** is no longer supported in any of the Microsoft developers' tools or libraries.

The **_fileinfo** variable determines whether information (such as the file handles and translation modes of open files used in a process) is passed to new processes by functions such as **_spawn**. For more information about the effect of changing the value of this variable, see _spawn. This global variable has been deprecated in Visual C++ 2005 for the more secure functional versions _get_fileinfo and _set_fileinfo, which should be used in place of the global variable. **_fileinfo** is declared in Stdlib.h as follows.

```
extern int _fileinfo;
```

Remarks

If **_fileinfo** is 0 (the default), information about open files is not passed to new processes; otherwise the information is passed. You can modify the default value of **_fileinfo** by setting the **_fileinfo** variable to a nonzero value in your program.

See Also

Reference

Global Variables
_set_fileinfo
_get_fileinfo

_fmode

The **_fmode** variable sets the default file-translation mode for text or binary translation. This global variable has been deprecated in Visual C++ 2005 for the more secure functional versions _get_fmode and _set_fmode, which should be used in place of the global variable. It is declared in Stdlib.h as follows.

```
extern int _fmode;
```

Remarks

The default setting of **_fmode** is **_O_TEXT** for text-mode translation. **_O_BINARY** is the setting for binary mode.

You can change the value of **_fmode** in three ways:

- Link with Binmode.obj. This changes the initial setting of **_fmode** to **_O_BINARY**, causing all files except **stdin**, **stdout**, and **stderr** to be opened in binary mode.
- Make a call to **_get_fmode** or **_set_fmode** to get or set the **_fmode** global variable, respectively.
- Change the value of **_fmode** directly by setting it in your program.

See Also

Reference

Global Variables
_get_fmode
_set_fmode

_osplatform, _osver, _winmajor, _winminor, _winver

These variables store build and version numbers of the 32-bit Windows operating systems. These variables have been deprecated because more secure functional equivalents are available.

Global variable	Functional equivalent
_osplatform	_get_osplatform
_osver	_get_osver
_winmajor	_get_winmajor
_winminor	_get_winminor
_winver	_get_winver

Declarations for these variables in Stdlib.h are as follows.

```
extern unsigned int _osplatform;
extern unsigned int _osver;
extern unsigned int _winmajor;
extern unsigned int _winminor;
extern unsigned int _winver;
```

Remarks

These variables are useful in programs that run in different versions of the Windows operating systems.

Variable	Description
_osplatform	Current platform.
_osver	Current build number.
_winmajor	Major version number.
_winminor	Minor version number.
_winver	Holds value of **_winmajor** in high byte and value of **_winminor** in low byte.

These global variables have been deprecated because the CRT cannot guarantee that they are initialized before they are used. Unpredictability in Windows DLL startup means that the CRT's DLL main might not run before the user's DLL main, meaning data exports might be uninitialized. This could lead to unsafe behavior, so global variable accesses should be replaced by calls to the appropriate functional equivalents which can return an error if the CRT is uninitialized.

See Also

Reference

Global Variables
_get_osplatform
_get_osver
_get_winmajor
_get_winminor
_get_winver

_pctype, _pwctype, _wctype, _mbctype, _mbcasemap

These global variables contain information used by the character classification functions. They are for internal use only.

```
extern const unsigned short *_pctype;
extern const wctype_t *_pwctype;
extern const unsigned short _wctype[];
extern unsigned char _mbctype[];
extern unsigned char _mbcasemap[];
```

Remarks

The information in **_pctype**, **_pwctype**, and **_wctype** is used internally by functions isupper, _isupper_l, iswupper, _iswupper_l, islower, iswlower, _islower_l, _iswlower_l, isdigit, iswdigit, _isdigit_l, _iswdigit_l, isxdigit, iswxdigit, _isxdigit_l, _iswxdigit_l, isspace, iswspace, _isspace_l, _iswspace_l, isalnum, iswalnum, _isalnum_l, _iswalnum_l, ispunct, iswpunct, _ispunct_l, _iswpunct_l, isgraph, iswgraph, _isgraph_l, _iswgraph_l, and iscntrl, iswcntrl, _iscntrl_l, _iswcntrl_l, as well as the toupper, _toupper, towupper, _toupper_l, _towupper_l and tolower, _tolower, towlower, _tolower_l, _towlower_l functions. These functions should be used instead of accessing these global variables.

The information in **_mbctype** and **_mbcasemap** is used internally by _ismbbkalnum, _ismbbkalnum_l, _ismbbkana, _ismbbkana_l, _ismbbkpunct, _ismbbkpunct_l, _ismbbkprint, _ismbbkprint_l, _ismbbalpha, _ismbbalpha_l, _ismbbpunct, _ismbbpunct_l, _ismbbalnum, _ismbbalnum_l, _ismbbprint, _ismbbprint_l, _ismbbgraph, _ismbbgraph_l, _ismbblead, _ismbblead_l, _ismbbtrail, _ismbbtrail_l, _ismbslead, _ismbstrail, _ismbslead_l, _ismbstrail_l, and _ismbslead, _ismbstrail, _ismbslead_l, _ismbstrail_l. Use these functions instead of accessing the global variables.

Requirements

Not for public use.

See Also

Reference

is, isw Routines

_pgmptr, _wpgmptr

The path of the executable file. Deprecated; use _get_pgmptr and _get_wpgmptr.

```
extern char *_pgmptr;
extern wchar_t *_wpgmptr;
```

Remarks

When a program is run from the command interpreter (Cmd.exe), **_pgmptr** is automatically initialized to the full path of the executable file. For example, if Hello.exe is in C:\BIN and C:\BIN is in the path, **_pgmptr** is set to C:\BIN\Hello.exe when you execute:

```
C> hello
```

When a program is not run from the command line, **_pgmptr** might be initialized to the program name (the file's base name without the file name extension) or to a file name, relative path, or full path.

_wpgmptr is the wide-character counterpart of **_pgmptr** for use with programs that use **wmain**.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tpgmptr	**_pgmptr**	**_pgmptr**	**_wpgmptr**

Requirements

Variable	Required header	Compatibility
_pgmptr, **_wpgmptr**	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003, Windows Server 2003

Example

The following program demonstrates the use of **_pgmptr**.

```
// crt_pgmptr.c
// compile with: /W3
// The following program demonstrates the use of _pgmptr.
//
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
   printf("The full path of the executing program is : %Fs\n",
     _pgmptr); // C4996
   // Note: _pgmptr is deprecated; use _get_pgmptr instead
}
```

You could use **_wpgmptr** by changing `%Fs` to `%S` and **main** to **wmain**.

See Also

Reference

Global Variables

Control Flags

The debug version of the Microsoft C run-time library uses the following flags to control the heap allocation and reporting process. For more information, see CRT Debugging Techniques.

Flag	Description
_CRTDBG_MAP_ALLOC	Maps the base heap functions to their debug version counterparts
_DEBUG	Enables the use of the debugging versions of the run-time functions
_crtDbgFlag	Controls how the debug heap manager tracks allocations

These flags can be defined with a /D command-line option or with a **#define** directive. When the flag is defined with **#define**, the directive must appear before the header file include statement for the routine declarations.

See Also

Reference

Global Variables and Standard Types

_CRTDBG_MAP_ALLOC

When the **_CRTDBG_MAP_ALLOC** flag is defined in the debug version of an application, the base version of the heap functions are directly mapped to their debug versions. This flag is declared in Crtdbg.h. This flag is only available when the _DEBUG flag has been defined in the application.

For more information about using the debug version versus the base version of a heap function, see Using the Debug Version Versus the Base Version.

See Also

Reference

Control Flags

_DEBUG

The compiler defines **_DEBUG** when you specify the /MTd or /MDd option. These options specify debug versions of the C run-time library.

For more information, see CRT Debugging Techniques.

See Also
Reference
Control Flags

_crtDbgFlag

The **_crtDbgFlag** flag consists of five bit fields that control how memory allocations on the debug version of the heap are tracked, verified, reported, and dumped. The bit fields of the flag are set using the _CrtSetDbgFlag function. This flag and its bit fields are declared in Crtdbg.h. This flag is only available when the _DEBUG flag has been defined in the application.

For more information about using this flag in conjunction with other debug functions, see Heap State Reporting Functions.

See Also
Reference
Control Flags

Standard Types

The Microsoft run-time library defines the following standard types.

Type	Description	Declared in
clock_t (long)	Stores time values; used by clock.	TIME.H
_complex structure	Stores real and imaginary parts of complex numbers; used by _cabs.	MATH.H
_CRT_ALLOC_HOOK	A type define for the user-defined hook function. Used in _CrtSetAllocHook.	CRTDBG.H
_CRT_DUMP_CLIENT, **_CRT_DUMP_CLIENT_M**	A type define for a call-back function that will get called in _CrtMemDumpAllObjectsSince.	CRTDBG.H
_CrtMemState structure	Provides information about the current state of the C run-time debug heap.	CRTDBG.H
_CRT_REPORT_HOOK, **_CRT_REPORT_HOOKW**, **_CRT_REPORT_HOOKW_M**	A type define for a call-back function that will get called in _CrtDbgReport. The parameters for this function are: report type, output message and the return value from the call-back function.	CRTDBG.H
dev_t, **_dev_t** short or unsigned integer	Represents device handles.	SYS\TYPES.H
_diskfree_t structure	Contains information about a disk drive. Used by _getdiskfree**.**	DOS.H and DIRECT.H
div_t, **ldiv_t** structures	Store values returned by div and ldiv, respectively.	STDLIB.H
errno_t integer	Used for a function return type or parameter that deals with the error codes of **errno**.	STDDEF.H, CRTDEFS.H
_exception structure	Stores error information for _matherr.	MATH.H
_EXCEPTION_POINTERS	Contains an exception record. See EXCEPTION_POINTERS for more information.	FPIEEE.H
FILE structure	Stores information about current state of stream; used in all stream I/O operations.	STDIO.H

_finddata_t, **_wfinddata_t**, **_finddata32_t**, **_wfinddata32_t**, **_finddatai64_t**, **_wfinddatai64_t**, **__finddata64_t**, **__wfinddata64_t**, **__finddata32i64_t**, **__wfinddata32i64_t**, **__finddata64i32_t**, **__wfinddata64i32_t** structures	Store file-attribute information returned by _findfirst, _wfindfirst and _findnext, _wfindnext. See _find, _wfind Functions for information on structure members.	IO.H, WCHAR.H
_FPIEEE_RECORD structure	Contains information pertaining to IEEE floating-point exception; passed to user-defined trap handler by _fpieee_flt.	FPIEEE.H
fpos_t (long integer, **__int64**, or structure, depending on the target platform)	Used by fgetpos and fsetpos to record information for uniquely specifying every position within a file.	STDIO.H
_fsize_t (unsigned long integer)	Used to represent a 64-bit size or value.	IO.H, WCHAR.H
_HEAPINFO structure	Contains information about next heap entry for _heapwalk.	MALLOC.H
_HFILE (void *)	An operating system file handle.	CRTDBG.H
lconv structure	Contains formatting rules for numeric values in different countries/regions.	LOCALE.H
ino_t, **_ino_t** (unsigned short)	For returning status information.	WCHAR.H
intptr_t (long integer or **__int64**, depending on the target platform)	Stores a pointer (or HANDLE) on both Win32 and Win64 platforms.	STDDEF.H and other include files
jmp_buf array	Used by setjmp and longjmp to save and restore program environment.	SETJMP.H
_LDOUBLE, **_LONGDOUBLE**, **_LDBL12** (long double or an unsigned char array)	Use to represent a long double value.	STDLIB.H
_locale_t structure	Stores current locale values; used in all locale specific C run-time libraries.	CRTDEF.H
mbstate_t	Tracks the state of a multibyte character conversion.	WCHAR.H
off_t, **_off_t** long integer	Represents file-offset value.	WCHAR.H, SYS\TYPES.H

_onexit_t, **_onexit_m_t** pointer	Returned by _onexit, _onexit_m.	STDLIB.H
_PNH pointer to function	Type of argument to _set_new_handler.	NEW.H
ptrdiff_t (long integer or **__int64**, depending on the target platform)	Result of subtraction of two pointers.	CRTDEFS.H
_purecall_handler, **_purecall_handler_m**	A type define for a call-back function that is called when a pure virtual function is called. Used by _set_purecall_handler, _set_purecall_handler_m. A **_purecall_handler** function should have a void return type.	STDLIB.H
_RTC_error_fn type define	A type define for a function that will handle run-time error checks. Used in _RTC_SetErrorFunc.	RTCAPI.H
_RTC_error_fnW type define	A type define for a function that will handle run-time error checks. Used in _RTC_SetErrorFuncW.	RTCAPI.H
_RTC_ErrorNumber enumeration	Defines error conditions for _RTC_GetErrDesc and _RTC_SetErrorType.	RTCAPI.H
_se_translator_function	A type define for a call-back function that translates an exception. The first parameter is the exception code and the second parameter is the exception record. Used by _set_se_translator.	EH.H
sig_atomic_t integer	Type of object that can be modified as atomic entity, even in presence of asynchronous interrupts; used with signal.	SIGNAL.H
size_t unsigned integer	Result of **sizeof** operator.	STDDEF.H and other include files
_stat structure	Contains file-status information returned by _stat and _fstat.	SYS\STAT.H
__stat64 structure	Contains file-status information returned by _fstat64 and _stat64, and _wstat64.	SYS\STAT.H
_stati64 structure	Contains file-status information returned by _fstati64, _stati64, and _wstati64.	SYS\STAT.H
_STATIC_ASSERT	For enforcing Boolean/integral conditions at compile time. Available in the non debug version.	CRTDBG.H
terminate_function type define	A type define for a call-back function that is called when terminate is called. Used by set_terminate.	EH.H

time_t (__int64 or long integer)	Represents time values in mktime, time, ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64, ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s, ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64 and gmtime, _gmtime32, _gmtime64. If _USE_32BIT_TIME_T is defined, **time_t** is a long integer. If not defined, it is a 64-bit integer.	TIME.H, SYS\STAT.H, SYS\TIMEB.H
__time32_t (long integer)	Represents time values in mktime, _mktime32, _mktime64, ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64, ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s, gmtime, _gmtime32, _gmtime64 and localtime, _localtime32, _localtime64.	CRTDEFS.H, SYS\STAT.H, SYS\TIMEB.H
__time64_t (**__int64**)	Represents time values in mktime, _mktime32, _mktime64, _ctime64, _wctime64, ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s, _gmtime64, _localtime64 and _time64.	TIME.H, SYS\STAT.H, SYS\TIMEB.H
_timeb structure	Used by _ftime and _ftime_s, _ftime32_s, _ftime64_s to store current system time.	SYS\TIMEB.H
__timeb32 structure	Used by _ftime, _ftime32, _ftime64 and _ftime_s, _ftime32_s, _ftime64_s to store current system time.	SYS\TIMEB.H
__timeb64 structure	Used by _ftime64 and _ftime_s, _ftime32_s, _ftime64_s to store current system time.	SYS\TIMEB.H
tm structure	Used by asctime, _wasctime, asctime_s, _wasctime_s, gmtime, _gmtime32, _gmtime64, _gmtime_s, _gmtime32_s, _gmtime64_s, localtime, _localtime32, _localtime64, localtime_s, _localtime32_s, _localtime64_s, mktime, _mktime32, _mktime64 and strftime, wcsftime, _strftime_l, _wcsftime_l to store and retrieve time information.	TIME.H
uintptr_t (long integer or **__int64**, depending on the target platform)	An unsigned integer or unsigned __int64 version of **intptr_t**.	STDDEF.H and other include files
unexpected_function	A type define for a call-back function that is called when unexpected is called. Used by set_unexpected.	EH.H

_utimbuf structure	Stores file access and modification times used by _utime, _wutime and _futime, _futime32, _futime64 to change file-modification dates.	SYS\UTIME.H
_utimbuf32 structure	Stores file access and modification times used by _utime, _utime32 _utime64, _wutime, _wutime32, _wutime64 and _futime, _futime32, _futime64 to change file-modification dates.	SYS\UTIME.H
__utimbuf64 structure	Used by _utime64, _wutime64 and _futime64 to store the current time.	SYS\UTIME.H
va_list structure	Used to hold information needed by va_arg and va_end macros. Called function declares variable of type **va_list** that can be passed as argument to another function.	STDARG.H, CRTDEFS.H
wchar_t wide character	Useful for writing portable programs for international markets.	STDDEF.H, STDLIB.H, CRTDEFS.H, SYS\STAT.H
wctrans_t integer	Represents locale-specific character mappings.	WCTYPE.H
wctype_t integer	Can represent all characters of any national character set.	WCHAR.H, CRTDEFS.H
wint_t integer	Type of data object that can hold any wide character or wide end-of-file value.	WCHAR.H, CRTDEFS.H

See Also

Other Resources

Run-Time Library Reference

Global Constants

The Microsoft run-time library contains definitions for global constants used by library routines. To use these constants, include the appropriate header files as indicated in the description for each constant. The global constants are listed in the following table.

32-bit Windows Time/Date Formats	Locale Categories
BUFSIZ	_locking Constants
CLOCKS_PER_SEC, CLK_TCK	Math Constants
Commit-To-Disk Constants	Math Error Constants
Data Type Constants	_MAX_ENV
Environmental Constants	MB_CUR_MAX
EOF, WEOF	NULL
errno	Path Field Limits
Exception-Handling Constants	RAND_MAX
EXIT_SUCCESS, EXIT_FAILURE	setvbuf Constants
File Attribute Constants	Sharing Constants
File Constants	signal Constants
File Permission Constants	signal Action Constants
File Read/Write Access Constants	_spawn Constants
File Translation Constants	_stat Structure st_mode Field Constants
FILENAME_MAX	stdin, stdout, stderr
FOPEN_MAX, _SYS_OPEN	TMP_MAX, L_tmpnam
_FREEENTRY, _USEDENTRY	Translation Mode Constants
fseek, _lseek	TZNAME_MAX
Heap Constants	_WAIT_CHILD, _WAIT_GRANDCHILD
_HEAP_MAXREQ	WCHAR_MAX
HUGE_VAL	WCHAR_MIN

See Also

Reference

Global Variables

Considerations for Writing Prolog/Epilog Code

Other Resources

Run-Time Library Reference

32-Bit Windows Time/Date Formats

The file time and the date are stored individually, using unsigned integers as bit fields. File time and date are packed as follows:

Time

Bit position:	**0 1 2 3 4**	**5 6 7 8 9 A**	**B C D E F**
Length:	5	6	5
Contents:	hours	minutes	2-second increments
Value Range:	0–23	0–59	0–29 in 2-second intervals

Date

Bit position:	**0 1 2 3 4 5 6**	**7 8 9 A**	**B C D E F**
Length:	7	4	5
Contents:	year	month	day
Value Range:	0–119	1–12	1–31
	(relative to 1980)		

See Also

Reference

Global Constants

BUFSIZ

```
#include <stdio.h>
```

Remarks

BUFSIZ is the required user-allocated buffer for the setvbuf routine.

See Also

Reference

Stream I/O

Global Constants

CLOCKS_PER_SEC, CLK_TCK

```
#include <time.h>
```

Remarks

The time in seconds is the value returned by the **clock** function, divided by **CLOCKS_PER_SEC**. **CLK_TCK** is equivalent, but considered obsolete.

See Also

Reference

clock

Global Constants

Commit-To-Disk Constants

Microsoft Specific

```
#include <stdio.h>
```

Remarks

These Microsoft-specific constants specify whether the buffer associated with the open file is flushed to operating system buffers or to disk. The mode is included in the string specifying the type of read/write access (**"r"**, **"w"**, **"a"**, **"r+"**, **"w+"**, **"a+"**).

The commit-to-disk modes are as follows:

c

Writes the unwritten contents of the specified buffer to disk. This commit-to-disk functionality only occurs at explicit calls to either the fflush or the _flushall function. This mode is useful when dealing with sensitive data. For example, if your program terminates after a call to **fflush** or **_flushall**, you can be sure that your data reached the operating system's buffers. However, unless a file is opened with the **c** option, the data might never make it to disk if the operating system also terminates.

n

Writes the unwritten contents of the specified buffer to the operating system's buffers. The operating system can cache data and then determine an optimal time to write to disk. Under many conditions, this behavior makes for efficient program behavior. However, if the retention of data is critical (such as bank transactions or airline ticket information) consider using the **c** option. The **n** mode is the default.

Note
The **c** and **n** options are not part of the ANSI standard for **fopen**, but are Microsoft extensions and should not be used where ANSI portability is desired.

Using the Commit-to-Disk Feature with Existing Code

By default, calls to the fflush or _flushall library functions write data to buffers maintained by the operating system. The operating system determines the optimal time to actually write the data to disk. The commit-to-disk feature of the run-time library lets you ensure that critical data is written directly to disk rather than to the operating system's buffers. You can give this capability to an existing program without rewriting it by linking its object files with COMMODE.OBJ.

In the resulting executable file, calls to **fflush** write the contents of the buffer directly to disk, and calls to **_flushall** write the contents of all buffers to disk. These two functions are the only ones affected by COMMODE.OBJ.

END Microsoft Specific

See Also

Reference

Stream I/O
_fdopen, _wfdopen
fopen, _wfopen
Global Constants

_CRT_DISABLE_PERFCRIT_LOCKS

Disables performance-critical locking in I/O operations.

```
#define _CRT_DISABLE_PERFCRIT_LOCKS
```

Remarks

Defining this symbol can improve performance in single-threaded I/O-bound programs by forcing all I/O operations to assume a single-threaded I/O model. For more information, see Multithreaded Libraries Performance.

See Also

Reference

Global Constants

Data Type Constants

Data type constants are implementation-dependent ranges of values allowed for integral data types. The constants listed below give the ranges for the integral data types and are defined in LIMITS.H.

Note
The /J compiler option changes the default **char** type to **unsigned**.

Constant	Value	Meaning
SCHAR_MAX	127	Maximum signed **char** value
SCHAR_MIN	–128	Minimum signed **char** value
UCHAR_MAX	255 (0xff)	Maximum **unsigned char** value
CHAR_BIT	8	Number of bits in a **char**
USHRT_MAX	65535 (0xffff)	Maximum **unsigned short** value
SHRT_MAX	32767	Maximum (signed) **short** value
SHRT_MIN	–32768	Minimum (signed) **short** value
UINT_MAX	4294967295 (0xffffffff)	Maximum **unsigned int** value
ULONG_MAX	4294967295 (0xffffffff)	Maximum **unsigned long** value
INT_MAX	2147483647	Maximum (signed) **int** value
INT_MIN	–2147483647–1	Minimum (signed) **int** value
LONG_MAX	2147483647	Maximum (signed) **long** value
LONG_MIN	–2147483647–1	Minimum (signed) **long** value
CHAR_MAX	127 (255 if /J option used)	Maximum **char** value
CHAR_MIN	–128 (0 if /J option used)	Minimum **char** value
MB_LEN_MAX	2	Maximum number of bytes in multibyte **char**
_I64_MAX	9223372036854775807	Maximum (signed) **__int64** value
_I64_MIN	-9223372036854775807-1	Minimum (signed) **__int64** value
_UI64_MAX	0xffffffffffffffff	Maximum (unsigned) **__int64** value

The following constants give the range and other characteristics of the **double** and **float** data types, and are defined in FLOAT.H:

Constant	Value	Description
DBL_DIG	15	# of decimal digits of precision

DBL_EPSILON	2.2204460492503131e-016	Smallest such that 1.0+**DBL_EPSILON** !=1.0
DBL_MANT_DIG	53	# of bits in mantissa
DBL_MAX	1.7976931348623158e+308	Maximum value
DBL_MAX_10_EXP	308	Maximum decimal exponent
DBL_MAX_EXP	1024	Maximum binary exponent
DBL_MIN	2.2250738585072014e-308	Minimum positive value
DBL_MIN_10_EXP	(-307)	Minimum decimal exponent
DBL_MIN_EXP	(–1021)	Minimum binary exponent
_DBL_RADIX	2	Exponent radix
_DBL_ROUNDS	1	Addition rounding: near
FLT_DIG	6	Number of decimal digits of precision
FLT_EPSILON	1.192092896e-07F	Smallest such that 1.0+**FLT_EPSILON** !=1.0
FLT_MANT_DIG	24	Number of bits in mantissa
FLT_MAX	3.402823466e+38F	Maximum value
FLT_MAX_10_EXP	38	Maximum decimal exponent
FLT_MAX_EXP	128	Maximum binary exponent
FLT_MIN	1.175494351e-38F	Minimum positive value
FLT_MIN_10_EXP	(–37)	Minimum decimal exponent
FLT_MIN_EXP	(–125)	Minimum binary exponent
FLT_RADIX	2	Exponent radix
FLT_ROUNDS	1	Addition rounding: near

See Also

Reference

Global Constants

Environmental Constants

```
#include <stdlib.h>
```

Remarks

This constant defines the environmental length for strings.

Constant	Meaning
_MAX_ENV	Maximum string size of an environmental string.

See Also

Reference

Global Constants

EOF, WEOF

Remarks

EOF is returned by an I/O routine when the end-of-file (or in some cases, an error) is encountered.

WEOF yields the return value, of type **wint_t**, used to signal the end of a wide stream, or to report an error condition.

See Also

Reference

putc, putwc
ungetc, ungetwc
scanf, _scanf_l, wscanf, _wscanf_l
fflush
fclose, _fcloseall
_ungetch, _ungetwch, _ungetch_nolock, _ungetwch_nolock
_putch, _putwch
__isascii, iswascii
Global Constants

errno Constants

```
#include <errno.h>
```

Remarks

The **errno** values are constants assigned to errno in the event of various error conditions.

ERRNO.H contains the definitions of the **errno** values. However, not all the definitions given in ERRNO.H are used in 32-bit Windows operating systems. Some of the values in ERRNO.H are present to maintain compatibility with the UNIX family of operating systems.

The **errno** values in a 32-bit Windows operating system are a subset of the values for **errno** in XENIX systems. Thus, the **errno** value is not necessarily the same as the actual error code returned by a system call from the Windows operating systems. To access the actual operating system error code, use the _doserrno variable, which contains this value.

The following **errno** values are supported:

ECHILD

No spawned processes.

EAGAIN

No more processes. An attempt to create a new process failed because there are no more process slots, or there is not enough memory, or the maximum nesting level has been reached.

E2BIG

Argument list too long.

EACCES

Permission denied. The file's permission setting does not allow the specified access. This error signifies that an attempt was made to access a file (or, in some cases, a directory) in a way that is incompatible with the file's attributes.

For example, the error can occur when an attempt is made to read from a file that is not open, to open an existing read-only file for writing, or to open a directory instead of a file. Under MS-DOS operating system versions 3.0 and later, **EACCES** may also indicate a locking or sharing violation.

The error can also occur in an attempt to rename a file or directory or to remove an existing directory.

EBADF

Bad file number. There are two possible causes: 1) The specified file descriptor is not a valid value or does not refer to an open file. 2) An attempt was made to write to a file or device opened for read-only access.

EDEADLOCK

Resource deadlock would occur. The argument to a math function is not in the domain of the function.

EDOM

Math argument.

EEXIST

Files exist. An attempt has been made to create a file that already exists. For example, the **_O_CREAT** and **_O_EXCL** flags are specified in an **_open** call, but the named file already exists.

EILSEQ

Illegal sequence of bytes (for example, in an MBCS string).

EINVAL

Invalid argument. An invalid value was given for one of the arguments to a function. For example, the value given for the

origin when positioning a file pointer (by means of a call to **fseek**) is before the beginning of the file.

EMFILE

Too many open files. No more file descriptors are available, so no more files can be opened.

ENOENT

No such file or directory. The specified file or directory does not exist or cannot be found. This message can occur whenever a specified file does not exist or a component of a path does not specify an existing directory.

ENOEXEC

Exec format error. An attempt was made to execute a file that is not executable or that has an invalid executable-file format.

ENOMEM

Not enough core. Not enough memory is available for the attempted operator. For example, this message can occur when insufficient memory is available to execute a child process, or when the allocation request in a **_getcwd** call cannot be satisfied.

ENOSPC

No space left on device. No more space for writing is available on the device (for example, when the disk is full).

ERANGE

Result too large. An argument to a math function is too large, resulting in partial or total loss of significance in the result. This error can also occur in other functions when an argument is larger than expected (for example, when the *buffer* argument to **_getcwd** is longer than expected).

EXDEV

Cross-device link. An attempt was made to move a file to a different device (using the **rename** function).

STRUNCATE

A string copy or concatenation resulted in a truncated string. See _TRUNCATE.

See Also

Reference

Global Constants

Exception-Handling Constants

The constant **EXCEPTION_CONTINUE_SEARCH**, **EXCEPTION_CONTINUE_EXECUTION**, or **EXCEPTION_EXECUTE_HANDLER** is returned when an exception occurs during execution of the guarded section of a **try-except** statement. The return value determines how the exception is handled. For more information, see try-except Statement in the *C++ Language Reference*.

See Also

Reference

Global Constants

EXIT_SUCCESS, EXIT_FAILURE

```
#include <stdlib.h>
```

Remarks

These are arguments for the **exit** and **_exit** functions and the return values for the **atexit** and **_onexit** functions.

See Also

Reference

atexit
exit, _exit
_onexit, _onexit_m
Global Constants

File Attribute Constants

```
#include <io.h>
```

Remarks

These constants specify the current attributes of the file or directory specified by the function.

The attributes are represented by the following manifest constants:

_A_ARCH

Archive. Set whenever the file is changed, and cleared by the BACKUP command. Value: 0x20

_A_HIDDEN

Hidden file. Not normally seen with the DIR command, unless the /AH option is used. Returns information about normal files as well as files with this attribute. Value: 0x02

_A_NORMAL

Normal. File can be read or written to without restriction. Value: 0x00

_A_RDONLY

Read-only. File cannot be opened for writing, and a file with the same name cannot be created. Value: 0x01

_A_SUBDIR

Subdirectory. Value: 0x10

_A_SYSTEM

System file. Not normally seen with the DIR command, unless the /AS option is used. Value: 0x04

Multiple constants can be combined with the OR operator (|).

See Also

Reference

_find, _wfind Functions
Global Constants

File Constants

```
#include <fcntl.h>
```

Remarks

The integer expression formed from one or more of these constants determines the type of reading or writing operations permitted. It is formed by combining one or more constants with a translation-mode constant.

The file constants are as follows:

_O_APPEND

Repositions the file pointer to the end of the file before every write operation.

_O_CREAT

Creates and opens a new file for writing; this has no effect if the file specified by *filename* exists.

_O_EXCL

Returns an error value if the file specified by *filename* exists. Only applies when used with **_O_CREAT**.

_O_RDONLY

Opens file for reading only; if this flag is given, neither **_O_RDWR** nor **_O_WRONLY** can be given.

_O_RDWR

Opens file for both reading and writing; if this flag is given, neither **_O_RDONLY** nor **_O_WRONLY** can be given.

_O_TRUNC

Opens and truncates an existing file to zero length; the file must have write permission. The contents of the file are destroyed. If this flag is given, you cannot specify **_O_RDONLY**.

_O_WRONLY

Opens file for writing only; if this flag is given, neither **_O_RDONLY** nor **_O_RDWR** can be given.

See Also

Reference

_open, _wopen
_sopen, _wsopen
Global Constants

File Permission Constants

```
#include <sys/stat.h>
```

Remarks

One of these constants is required when **_O_CREAT** (**_open**, **_sopen**) is specified.

The *pmode* argument specifies the file's permission settings as follows.

Constant	Meaning
_S_IREAD	Reading permitted
_S_IWRITE	Writing permitted
_S_IREAD | **_S_IWRITE**	Reading and writing permitted

When used as the *pmode* argument for **_umask**, the manifest constant sets the permission setting, as follows.

Constant	Meaning
_S_IREAD	Writing not permitted (file is read-only)
_S_IWRITE	Reading not permitted (file is write-only)
_S_IREAD | **_S_IWRITE**	Neither reading nor writing permitted

See Also

Reference

_open, _wopen
_sopen, _wsopen
_umask
Standard Types
Global Constants

File Read/Write Access Constants

```
#include <stdio.h>
```

Remarks

These constants specify the access type ("a", "r", or "w") requested for the file. Both the translation mode ("b" or "t") and the commit-to-disk mode ("c" or "n") can be specified with the type of access.

The access types are described below.

"a"

Opens for writing at the end of the file (appending); creates the file first if it does not exist. All write operations occur at the end of the file. Although the file pointer can be repositioned using **fseek** or **rewind**, it is always moved back to the end of the file before any write operation is carried out.

"a+"

Same as above, but also allows reading.

"r"

Opens for reading. If the file does not exist or cannot be found, the call to open the file will fail.

"r+"

Opens for both reading and writing. If the file does not exist or cannot be found, the call to open the file will fail.

"w"

Opens an empty file for writing. If the given file exists, its contents are destroyed.

"w+"

Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.

When the "r+", "w+", or "a+" type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when you switch between reading and writing, there must be an intervening **fflush**, **fsetpos**, **fseek**, or **rewind** operation. The current position can be specified for the **fsetpos** or **fseek** operation.

See Also

Reference

_fdopen, _wfdopen
fopen, _wfopen
freopen, _wfreopen
_fsopen, _wfsopen
_popen, _wpopen
Global Constants

File Translation Constants

```
#include <stdio.h>
```

Remarks

These constants specify the mode of translation (**"b"** or **"t"**). The mode is included in the string specifying the type of access (**"r"**, **"w"**, **"a"**, **"r+"**, **"w+"**, **"a+"**).

The translation modes are as follows:

t

Opens in text (translated) mode. In this mode, carriage-return/linefeed (CR-LF) combinations are translated into single linefeeds (LF) on input, and LF characters are translated into CR-LF combinations on output. Also, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading or reading/writing, **fopen** checks for CTRL+Z at the end of the file and removes it, if possible. This is done because using the **fseek** and **ftell** functions to move within a file ending with CTRL+Z may cause **fseek** to behave improperly near the end of the file.

Note
The **t** option is not part of the ANSI standard for **fopen** and **freopen**. It is a Microsoft extension and should not be used where ANSI portability is desired.

b

Opens in binary (untranslated) mode. The above translations are suppressed.

If **t** or **b** is not given in *mode*, the translation mode is defined by the default-mode variable _fmode. For more information about using text and binary modes, see Text and Binary Mode File I/O.

See Also

Reference

_fdopen, _wfdopen
fopen, _wfopen
freopen, _wfreopen
_fsopen, _wfsopen
Global Constants

FILENAME_MAX

The maximum permissible length for a *filename* string buffer size.

```
#include <stdio.h>
```

See Also

Reference

Path Field Limits

Global Constants

FOPEN_MAX, _SYS_OPEN

```
#include <stdio.h>
```

Remarks

This is the maximum number of files that can be opened simultaneously. **FOPEN_MAX** is the ANSI-compatible name. **_SYS_OPEN** is provided for compatibility with existing code.

See Also

Reference

Global Constants

_FREEENTRY, _USEDENTRY

```
#include <malloc.h>
```

Remarks

These constants represent values assigned by the **_heapwalk** routines to the **_useflag** element of the **_HEAPINFO** structure. They indicate the status of the heap entry.

See Also

Reference

_heapwalk

Global Constants

fseek, _lseek Constants

```
#include <stdio.h>
```

Remarks

The *origin* argument specifies the initial position and can be one of the following manifest constants:

Constant	Meaning
SEEK_END	End of file
SEEK_CUR	Current position of file pointer
SEEK_SET	Beginning of file

See Also

Reference

fseek, _fseeki64

_lseek, _lseeki64

Global Constants

Heap Constants

```
#include <malloc.h>
```

Remarks

These constants give the return value indicating status of the heap.

Constant	Meaning
_HEAPBADBEGIN	Initial header information was not found or was invalid.
_HEAPBADNODE	Bad node was found, or heap is damaged.
_HEAPBADPTR	**_pentry** field of **_HEAPINFO** structure does not contain valid pointer into heap (**_heapwalk** routine only).
_HEAPEMPTY	Heap has not been initialized.
_HEAPEND	End of heap was reached successfully (**_heapwalk** routine only).
_HEAPOK	Heap is consistent (**_heapset** and **_heapchk** routines only). No errors so far; **_HEAPINFO** structure contains information about next entry (**_heapwalk** routine only).

See Also

Reference

_heapchk
_heapset
_heapwalk
Global Constants

_HEAP_MAXREQ

```
#include <malloc.h>
```

Remarks

The maximum size of a user request for memory that can be granted.

See Also

Reference

malloc
calloc
Global Constants

HUGE_VAL

```
#include <math.h>
```

Remarks

HUGE_VAL is the largest representable double value. This value is returned by many run-time math functions when an error occurs. For some functions, –**HUGE_VAL** is returned.

See Also

Reference

Global Constants

Locale Categories

```
#include <locale.h>
```

Remarks

Locale categories are manifest constants used by the localization routines to specify which portion of a program's locale information will be used. The locale refers to the locality (or Country/Region) for which certain aspects of your program can be customized. Locale-dependent areas include, for example, the formatting of dates or the display format for monetary values.

Locale category	Parts of program affected
LC_ALL	All locale-specific behavior (all categories)
LC_COLLATE	Behavior of **strcoll** and **strxfrm** functions
LC_CTYPE	Behavior of character-handling functions (except **isdigit**, **isxdigit**, **mbstowcs**, and **mbtowc**, which are unaffected)
LC_MAX	Same as **LC_TIME**
LC_MIN	Same as **LC_ALL**
LC_MONETARY	Monetary formatting information returned by the **localeconv** function
LC_NUMERIC	Decimal-point character for formatted output routines (for example, **printf**), data conversion routines, and nonmonetary formatting information returned by **localeconv** function
LC_TIME	Behavior of **strftime** function

See Also

Reference

localeconv
setlocale, _wsetlocale
strcoll Functions
strftime, wcsftime, _strftime_l, _wcsftime_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l
Global Constants

_locking Constants

```
#include <sys/locking.h>
```

Remarks

The *mode* argument in the call to the **_locking** function specifies the locking action to be performed.

The *mode* argument must be one of the following manifest constants.

_LK_LOCK

Locks the specified bytes. If the bytes cannot be locked, the function tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, the function returns an error.

_LK_RLCK

Same as **_LK_LOCK**.

_LK_NBLCK

Locks the specified bytes. If bytes cannot be locked, the function returns an error.

_LK_NBRLCK

Same as **_LK_NBLCK**.

_LK_UNLCK

Unlocks the specified bytes. (The bytes must have been previously locked.)

See Also

Reference

_locking

Global Constants

Math Constants

```
#include <math.h>
```

Remarks

The following symbols are defined for the values of their indicated expressions:

Symbol	Expression	Value
M_E	e	2.71828182845904523536
M_LOG2E	log2(e)	1.44269504088896340736
M_LOG10E	log10(e)	0.434294481903251827651
M_LN2	ln(2)	0.693147180559945309417
M_LN10	ln(10)	2.30258509299404568402
M_PI	pi	3.14159265358979323846
M_PI_2	pi/2	1.57079632679489661923
M_PI_4	pi/4	0.785398163397448309616
M_1_PI	1/pi	0.318309886183790671538
M_2_PI	2/pi	0.636619772367581343076
M_2_SQRTPI	2/sqrt(pi)	1.12837916709551257390
M_SQRT2	sqrt(2)	1.41421356237309504880
M_SQRT1_2	1/sqrt(2)	0.707106781186547524401

The "Math Constants" are not defined in Standard C/C++. Therefore, in order to use them, you have to first define _USE_MATH_DEFINES and then include cmath or math.h as below.

If compiling as C++:

```
#define _USE_MATH_DEFINES
#include <cmath>
```

If compiling as C:

```
#define _USE_MATH_DEFINES
#include <math.h>
```

See Also

Reference

Global Constants

Math Error Constants

```
#include <math.h>
```

Remarks

The math routines of the run-time library can generate math error constants.

These errors, described as follows, correspond to the exception types defined in MATH.H and are returned by the **_matherr** function when a math error occurs.

Constant	Meaning
_DOMAIN	Argument to function is outside domain of function.
_OVERFLOW	Result is too large to be represented in function's return type.
_PLOSS	Partial loss of significance occurred.
_SING	Argument singularity: argument to function has illegal value. (For example, value 0 is passed to function that requires nonzero value.)
_TLOSS	Total loss of significance occurred.
_UNDERFLOW	Result is too small to be represented.

See Also

Reference

_matherr

Global Constants

_MAX_ENV

The maximum permissible string length of an environmental variable.

```
#include <stdio.h>
```

See Also

Reference

Environmental Constants

Global Constants

MB_CUR_MAX

```
#include <stdlib.h>
```

Remarks

Context: ANSI multibyte- and wide-character conversion functions

The value of **MB_CUR_MAX** is the maximum number of bytes in a multibyte character for the current locale.

See Also

Reference

_mbclen, mblen, _mblen_l
mbstowcs, _mbstowcs_l
mbtowc, _mbtowc_l
Standard Types
wcstombs, _wcstombs_l
wctomb, _wctomb_l
Data Type Constants
Global Constants

NULL (CRT)

NULL is the null-pointer value used with many pointer operations and functions. It is equivalent to 0. **NULL** is defined in the following header files: CRTDBG.H, LOCALE.H, STDDEF.H, STDIO.H, STDLIB.H, STRING.H, TCHAR.H, TIME.H and WCHAR.H.

See Also

Reference

Global Constants

Path Field Limits

```
#include <stdlib.h>
```

Remarks

These constants define the maximum length for the path and for the individual fields within the path.

Constant	Meaning
_MAX_DIR	Maximum length of directory component
_MAX_DRIVE	Maximum length of drive component
_MAX_EXT	Maximum length of extension component
_MAX_FNAME	Maximum length of filename component
_MAX_PATH	Maximum length of full path

Note The C Runtime supports path lengths up to 32768 characters in length, but it is up to the operating system, specifically the file system, to support these longer paths. The sum of the fields should not exceed **_MAX_PATH** for full backwards compatibility with Windows 98 FAT32 file systems. Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003, and Windows Server 2003 NTFS file system supports paths up to 32768 characters in length, but only when using the Unicode APIs. When using long path names, prefix the path with the characters \\?\ and use the Unicode versions of the C Runtime functions.

See Also

Reference

Global Constants

RAND_MAX

```
#include <stdlib.h>
```

Remarks

The constant **RAND_MAX** is the maximum value that can be returned by the **rand** function. **RAND_MAX** is defined as the value 0x7fff.

See Also

Reference

rand

Global Constants

setvbuf Constants

```
#include <stdio.h>
```

Remarks

These constants represent the type of buffer for **setvbuf**.

The possible values are given by the following manifest constants:

Constant	Meaning
_IOFBF	Full buffering: Buffer specified in call to **setvbuf** is used and its size is as specified in **setvbuf** call. If buffer pointer is **NULL**, automatically allocated buffer of specified size is used.
_IOLBF	Same as **_IOFBF**.
_IONBF	No buffer is used, regardless of arguments in call to **setvbuf**.

See Also

Reference

setbuf

Global Constants

Sharing Constants

```
#include <share.h>
```

Remarks

The *shflag* argument determines the sharing mode, which consists of one or more manifest constants. These can be combined with the *oflag* arguments (see File Constants).

The constants and their meanings are listed below:

Constant	Meaning
_SH_COMPAT	Sets compatibility mode
_SH_DENYRW	Denies read and write access to file
_SH_DENYWR	Denies write access to file
_SH_DENYRD	Denies read access to file
_SH_DENYNO	Permits read and write access
_SH_SECURE	Sets secure mode (shared read, exclusive write access).

See Also

Reference

_sopen, _wsopen
_fsopen, _wfsopen
Global Constants

signal Constants

```
#include <signal.h>
```

Remarks

The *sig* argument must be one of the manifest constants listed below (defined in SIGNAL.H).

SIGABRT

Abnormal termination. The default action terminates the calling program with exit code 3.

SIGABRT_COMPAT

Same as SIGABRT. For compatibility with other platforms.

SIGFPE

Floating-point error, such as overflow, division by zero, or invalid operation. The default action terminates the calling program.

SIGILL

Illegal instruction. The default action terminates the calling program.

SIGINT

CTRL+C interrupt. The default action terminates the calling program with exit code 3.

SIGSEGV

Illegal storage access. The default action terminates the calling program.

SIGTERM

Termination request sent to the program. The default action terminates the calling program with exit code 3.

SIG_ERR

A return type from a signal indicating an error has occurred.

See Also

Reference

signal
raise
Global Constants

signal Action Constants

The action taken when the interrupt signal is received depends on the value of *func*.

```
#include <signal.h>
```

Remarks

The *func* argument must be either a function address or one of the manifest constants listed below and defined in SIGNAL.H.

SIG_DFL

Uses system-default response. If the calling program uses stream I/O, buffers created by the run-time library are not flushed.

SIG_IGN

Ignores interrupt signal. This value should never be given for **SIGFPE**, since the floating-point state of the process is left undefined.

SIG_SGE

Indicates an error occurred in the signal.

SIG_ACK

Indicates an acknowledgement was received.

SIG_ERR

A return type from a signal indicating an error has occurred.

See Also

Reference

signal

Global Constants

spawn Constants

```
#include <process.h>
```

Remarks

The *mode* argument determines the action taken by the calling process before and during a spawn operation. The following values for *mode* are possible:

Constant	Meaning
_P_OVERLAY	Overlays calling process with new process, destroying calling process (same effect as **_exec** calls).
_P_WAIT	Suspends calling thread until execution of new process is complete (synchronous **_spawn**).
_P_NOWAIT, **_P_NOWAITO**	Continues to execute calling process concurrently with new process (asynchronous **_spawn**).
_P_DETACH	Continues to execute calling process; new process is run in background with no access to console or keyboard. Calls to **_cwait** against new process will fail. This is an asynchronous **_spawn**.

See Also

Reference

_spawn, _wspawn Functions

Global Constants

_stat Structure st_mode Field Constants

```
#include <sys/stat.h>
```

Remarks

These constants are used to indicate file type in the **st_mode** field of the _stat structure.

The bit mask constants are described below:

Constant	Meaning
_S_IFMT	File type mask
_S_IFDIR	Directory
_S_IFCHR	Character special (indicates a device if set)
_S_IFREG	Regular
_S_IREAD	Read permission, owner
_S_IWRITE	Write permission, owner
_S_IEXEC	Execute/search permission, owner

See Also

Reference

_stat, _wstat Functions
_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32
Standard Types
Global Constants

stdin, stdout, stderr

```
FILE *stdin;
FILE *stdout;
FILE *stderr;
#include <stdio.h>
```

Remarks

These are standard streams for input, output, and error output.

By default, standard input is read from the keyboard, while standard output and standard error are printed to the screen.

The following stream pointers are available to access the standard streams:

Pointer	Stream
stdin	Standard input
stdout	Standard output
stderr	Standard error

These pointers can be used as arguments to functions. Some functions, such as **getchar** and **putchar**, use **stdin** and **stdout** automatically.

These pointers are constants, and cannot be assigned new values. The **freopen** function can be used to redirect the streams to disk files or to other devices. The operating system allows you to redirect a program's standard input and output at the command level.

See Also

Reference

Stream I/O

Global Constants

TMP_MAX, L_tmpnam

```
#include <stdio.h>
```

Remarks

TMP_MAX is the maximum number of unique filenames that the **tmpnam** function can generate. **L_tmpnam** is the length of temporary filenames generated by **tmpnam**.

See Also

Reference

Global Constants

Translation Mode Constants

```
#include <fcntl.h>
```

Remarks

The **_O_BINARY** and **_O_TEXT** manifest constants determine the translation mode for files (**_open** and **_sopen**) or the translation mode for streams (**_setmode**).

The allowed values are:

_O_TEXT

Opens file in text (translated) mode. Carriage return – linefeed (CR-LF) combinations are translated into a single linefeed (LF) on input. Linefeed characters are translated into CR-LF combinations on output. Also, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading and reading/writing, **fopen** checks for CTRL+Z at the end of the file and removes it, if possible. This is done because using the **fseek** and **ftell** functions to move within a file ending with CTRL+Z may cause **fseek** to behave improperly near the end of the file.

_O_BINARY

Opens file in binary (untranslated) mode. The above translations are suppressed.

_O_RAW

Same as **_O_BINARY**. Supported for C 2.0 compatibility.

For more information, see Text and Binary Mode File I/O and File Translation.

See Also

Reference

_open, _wopen
_pipe
_sopen, _wsopen
_setmode
Global Constants

_TRUNCATE

Specifies string truncation behavior.

```
#include <stdlib.h>
```

Remarks

_TRUNCATE enables truncation behavior when passed as the *count* parameter to these functions:

strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l

strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l

mbstowcs_s, _mbstowcs_s_l

mbsrtowcs_s

wcstombs_s, _wcstombs_s_l

wcsrtombs_s

_snprintf_s, _snprintf_s_l, _snwprintf_s, _snwprintf_s_l

vsnprintf_s, _vsnprintf_s, _vsnprintf_s_l, _vsnwprintf_s, _vsnwprintf_s_l

If the destination buffer is too small to hold the entire string, the normal behavior of these functions is to treat it as an error situation (see Parameter Validation). However, if string truncation is enabled by passing **_TRUNCATE**, these functions will copy only as much of the string as will fit, leaving the destination buffer null-terminated, and return successfully.

String truncation changes the return values of the affected functions. The following functions return 0 if no truncation occurs, or **STRUNCATE** if truncation does occur:

strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l

strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l

wcstombs_s, _wcstombs_s_l

mbstowcs_s, _mbstowcs_s_l

The following functions return the number of characters copied if no truncation occurs, or -1 if truncation does occur (matching the behavior of the original snprintf functions):

_snprintf_s, _snprintf_s_l, _snwprintf_s, _snwprintf_s_l

vsnprintf_s, _vsnprintf_s, _vsnprintf_s_l, _vsnwprintf_s, _vsnwprintf_s_l

Example

```
// crt_truncate.c
#include <stdlib.h>
#include <errno.h>

int main()
{
   char src[] = "1234567890";
   char dst[5];
   errno_t err = strncpy_s(dst, _countof(dst), src, _TRUNCATE);
   if ( err == STRUNCATE )
      printf( "truncation occurred!\n" );
   printf( "'%s'\n", dst );
}
```

Output

```
truncation occurred!
'1234'
```

See Also

Reference

Global Constants

TZNAME_MAX

The maximum permissible string length for a time zone name variable.

```
#include <limits.h>
```

See Also

Reference

Environmental Constants

Global Constants

_get_tzname

_WAIT_CHILD, _WAIT_GRANDCHILD

```
#include <process.h>
```

Remarks

The **_cwait** function can be used by any process to wait for any other process (if the process ID is known). The action argument can be one of the following values:

Constant	Meaning
_WAIT_CHILD	Calling process waits until specified new process terminates.
_WAIT_GRANDCHILD	Calling process waits until specified new process, and all processes created by that new process, terminate.

See Also

Reference

_cwait

Global Constants

WCHAR_MAX

Maximum value for type **wchar_t**.

```
#include <wchar.h>
```

See Also

Reference

Global Constants

WCHAR_MIN

Minimum value for type **wchar_t**.

```
#include <wchar.h>
```

See Also

Reference

Global Constants

Alphabetical Function Reference

This section is an alphabetical reference to the routines in the C Run-Time (CRT) Library. To find a CRT routine based on functionality, see Run-Time Routines by Category.

__isascii, iswascii

__max

__min

__toascii

_access, _waccess

_access_s, _waccess_s

_aligned_free

_aligned_malloc

_aligned_offset_malloc

_aligned_offset_realloc

_aligned_realloc

_alloca

_ASSERT, _ASSERTE Macros

_atodbl, _atodbl_l, _atoldbl, _atoldbl_l, _atoflt _atoflt_l

_atoi64, _atoi64_l, _wtoi64, _wtoi64_l

_beginthread, _beginthreadex

_byteswap_uint64, _byteswap_ulong, _byteswap_ushort

_cabs

_calloc_dbg

_cexit, _c_exit

_cgets, _cgetws

_cgets_s, _cgetws_s

_chdir, _wchdir

_chdrive

_chgsign, _chgsignl

_chmod, _wchmod

_chsize

_chsize_s

_clear87, _clearfp

_close

_commit

_configthreadlocale

_control87, _controlfp, __control87_2

_controlfp_s

_copysign, _copysignl

_countof Macro

_cprintf, _cprintf_l, _cwprintf, _cwprintf_l

_cprintf_p, _cprintf_p_l, _cwprintf_p, _cwprintf_p_l

_cprintf_s, _cprintf_s_l, _cwprintf_s, _cwprintf_s_l

_cputs, _cputws

_creat, _wcreat

_create_locale

_CrtCheckMemory

_CrtDbgBreak

_CrtDbgReport, _CrtDbgReportW

_CrtDoForAllClientObjects

_CrtDumpMemoryLeaks

_CrtGetAllocHook

_CrtGetDumpClient

_CrtGetReportHook

_CrtIsMemoryBlock

_CrtIsValidHeapPointer

_CrtIsValidPointer

_CrtMemCheckpoint

_CrtMemDifference

_CrtMemDumpAllObjectsSince

_CrtMemDumpStatistics

_CrtReportBlockType

_CrtSetAllocHook

_CrtSetBreakAlloc

_CrtSetDbgFlag

_CrtSetDebugFillThreshold

_CrtSetDumpClient

_CrtSetReportFile

_CrtSetReportHook

_CrtSetReportHook2, _CrtSetReportHookW2

_CrtSetReportMode

_cscanf, _cscanf_l, _cwscanf, _cwscanf_l

_cscanf_s, _cscanf_s_l, _cwscanf_s, _cwscanf_s_l

_cwait

_dup, _dup2

_dupenv_s, _wdupenv_s

_dupenv_s_dbg, _wdupenv_s_dbg

_ecvt

_ecvt_s

_endthread, _endthreadex

_eof

_exec, _wexec Functions

_execl, _wexecl

_execle, _wexecle

_execlp, _wexeclp

_execlpe, _wexeclpe

_execv, _wexecv

_execve, _wexecve

_execvp, _wexecvp

_execvpe, _wexecvpe

_expand

_expand_dbg

_fclose_nolock

_fcvt

_fcvt_s

_fdopen, _wfdopen

_fflush_nolock

_fgetc_nolock, _fgetwc_nolock

_fgetchar, _fgetwchar

_filelength, _filelengthi64

_fileno

_find, _wfind Functions

_findclose

_findfirst, _wfindfirst

_findnext, _wfindnext

_finite

_flushall

_fpclass

_fpieee_flt

_fpreset

_fprintf_p, _fprintf_p_l, _fwprintf_p, _fwprintf_p_l

_fputc_nolock, _fputwc_nolock

_fputchar, _fputwchar

_fread_nolock

_free_dbg

_free_locale

_freea

_fseek_nolock, _fseeki64_nolock

_fsopen, _wfsopen

_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32

_ftell_nolock, _ftelli64_nolock

_ftime, _ftime32, _ftime64

_ftime_s, _ftime32_s, _ftime64_s

_fullpath, _wfullpath

_fullpath_dbg, _wfullpath_dbg

_futime, _futime32, _futime64

_fwrite_nolock

_gcvt

_gcvt_s

_get_amblksiz

_get_current_locale

_get_daylight

_get_doserrno

_get_dstbias

_get_errno

_get_fileinfo

_get_fmode

_get_heap_handle

_get_osfhandle

_get_osplatform

_get_osver

_get_output_format

_get_pgmptr

_get_printf_count_output

_get_sbh_threshold

_get_terminate

_get_timezone

_get_tzname

_get_unexpected

_get_winmajor

_get_winminor

_get_winver

_get_wpgmptr

_getc_nolock, _getwc_nolock

_getch, _getwch

_getch_nolock, _getwch_nolock

_getchar_nolock, _getwchar_nolock

_getche, _getwche

_getche_nolock, _getwche_nolock

_getcwd, _wgetcwd

_getcwd_dbg, _wgetcwd_dbg

_getdcwd, _wgetdcwd

_getdcwd_dbg, _wgetdcwd_dbg

_getdcwd_nolock, _wgetdcwd_nolock

_getdiskfree

_getdrive

_getdrives

_getmaxstdio

_getmbcp

_getpid

_getw

_gmtime_s, _gmtime32_s, _gmtime64_s

_heapadd

_heapchk

_heapmin

_heapset

_heapwalk

_hypot, hypotf

_inp, _inpw, _inpd

_isatty

_isctype, iswctype, _isctype_l, _iswctype_l

_ismbb Routines

_ismbbalnum, _ismbbalnum_l

_ismbbalpha, _ismbbalpha_l

_ismbbgraph, _ismbbgraph_l

_ismbbkalnum, _ismbbkalnum_l

_ismbbkana, _ismbbkana_l

_ismbbkprint, _ismbbkprint_l

_ismbbkpunct, _ismbbkpunct_l

_ismbblead, _ismbblead_l

_ismbbprint, _ismbbprint_l

_ismbbpunct, _ismbbpunct_l

_ismbbtrail, _ismbbtrail_l

_ismbc Routines

_ismbcalnum, _ismbcalnum_l, _ismbcalpha, _ismbcalpha_, _ismbcdigit, _ismbcdigit_l

_ismbcgraph, _ismbcgraph_l, _ismbcprint, _ismbcprint_l, _ismbcpunct, _ismbcpunct_l, _ismbcspace, _ismbcspace_l

_ismbchira, _ismbchira_l, _ismbckata, _ismbckata_l

_ismbcl0, _ismbcl0_l, _ismbcl1, _ismbcl1_l, _ismbcl2, _ismbcl2_l

_ismbclegal, _ismbclegal_l, _ismbcsymbol, _ismbcsymbol_l

_ismbclower, _ismbclower_l, _ismbcupper, _ismbcupper_l

_ismbslead, _ismbstrail, _ismbslead_l, _ismbstrail_l

_isnan

_itoa, _i64toa, _ui64toa, _itow, _i64tow, _ui64tow

_itoa_s, _i64toa_s, _ui64toa_s, _itow_s, _i64tow_s, _ui64tow_s

_kbhit

_lfind

_lfind_s

localtime_s, _localtime32_s, _localtime64_s

_lock_file

_locking

_logb

_lrotl, _lrotr

_lsearch

_lsearch_s

_lseek, _lseeki64

_ltoa, _ltow

_ltoa_s, _ltow_s

_makepath, _wmakepath

_makepath_s, _wmakepath_s

_malloc_dbg

_malloca

_matherr

_mbbtombc, _mbbtombc_l

_mbbtype, _mbbtype_l

_mbccpy, _mbccpy_l

_mbccpy_s, _mbccpy_s_l

_mbcjistojms, _mbcjistojms_l, _mbcjmstojis, _mbcjmstojis_l

_mbclen, mblen, _mblen_l

_mbctohira, _mbctohira_l, _mbctokata, _mbctokata_l

_mbctolower, _mbctolower_l, _mbctoupper, _mbctoupper_l

_mbctombb, _mbctombb_l

_mbsbtype, _mbsbtype_l

_mbsdec, _mbsdec_l, _strdec, _wcsdec

_mbsinc, _mbsinc_l, _strinc, _wcsinc

_mbsnbcat, _mbsnbcat_l

_mbsnbcat_s, _mbsnbcat_s_l

_mbsnbcmp, _mbsnbcmp_l

_mbsnbcnt, _mbsnbcnt_l, _mbsnccnt, _mbsnccnt_l, _strncnt, _wcsncnt

_mbsnbcoll, _mbsnbcoll_l, _mbsnbicoll, _mbsnbicoll_l

_mbsnbcpy, _mbsnbcpy_l

_mbsnbcpy_s, _mbsnbcpy_s_l

_mbsnbicmp, _mbsnbicmp_l

_mbsnbset, _mbsnbset_l

_mbsnbset_s, _mbsnbset_s_l

_mbsnextc, _mbsnextc_l, _strnextc, _wcsnextc

_mbsninc, _mbsninc_l, _strninc, _wcsninc

_mbsspnp, _mbsspnp_l, _strspnp, _wcsspnp

_memccpy

_memicmp, _memicmp_l

_mkdir, _wmkdir

_mkgmtime, _mkgmtime32, _mkgmtime64

_mktemp, _wmktemp

_mktemp_s, _wmktemp_s

_msize

_msize_dbg

_nextafter

_onexit, _onexit_m

_open, _wopen

_open_osfhandle

_outp, _outpw, _outpd

_pclose

_pipe

_popen, _wpopen

_printf_p, _printf_p_l, _wprintf_p, _wprintf_p_l

_putc_nolock, _putwc_nolock

_putch, _putwch

_putch_nolock, _putwch_nolock

_putchar_nolock, _putwchar_nolock

_putenv, _wputenv

_putenv_s, _wputenv_s

_putw

_query_new_handler

_query_new_mode

_read

_realloc_dbg

_resetstkoflw

_rmdir, _wrmdir

_rmtmp

_rotl, _rotl64, _rotr, _rotr64

_RPT, _RPTF, _RPTW, _RPTFW Macros

_RTC_GetErrDesc

_RTC_NumErrors

_RTC_SetErrorFunc

_RTC_SetErrorFuncW

_RTC_SetErrorType

_scalb

_scprintf, _scprintf_l, _scwprintf, _scwprintf_l

_scprintf_p, _scprintf_p_l, _scwprintf_p, _scwprintf_p_l

_searchenv, _wsearchenv

_searchenv_s, _wsearchenv_s

_set_abort_behavior

_set_amblksiz

_set_controlfp

_set_doserrno

_set_errno

_set_error_mode

_set_fileinfo

_set_fmode

_set_invalid_parameter_handler

_set_new_handler

_set_new_mode

_set_output_format

_set_printf_count_output

_set_purecall_handler, _set_purecall_handler_m

_set_sbh_threshold

_set_se_translator

_set_SSE2_enable

_setmaxstdio

_setmbcp

_setmode

_snprintf, _snprintf_l, _snwprintf, _snwprintf_l

_snprintf_s, _snprintf_s_l, _snwprintf_s, _snwprintf_s_l

_snscanf, _snscanf_l, _snwscanf, _snwscanf_l

_snscanf_s, _snscanf_s_l, _snwscanf_s, _snwscanf_s_l

_sopen, _wsopen

_sopen_s, _wsopen_s

_spawn, _wspawn Functions

_spawnl, _wspawnl

_spawnle, _wspawnle

_spawnlp, _wspawnlp

_spawnlpe, _wspawnlpe

_spawnv, _wspawnv

_spawnve, _wspawnve

_spawnvp, _wspawnvp

_spawnvpe, _wspawnvpe

_splitpath, _wsplitpath

_splitpath_s, _wsplitpath_s

_sprintf_p, _sprintf_p_l, _swprintf_p, _swprintf_p_l

_stat, _wstat Functions

_status87, _statusfp, _statusfp2

_strdate, _wstrdate

_strdate_s, _wstrdate_s

_strdup, _wcsdup, _mbsdup

_strdup_dbg, _wcsdup_dbg

_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l

_stricoll, _wcsicoll, _mbsicoll, _stricoll_l, _wcsicoll_l, _mbsicoll_l

_strlwr, _wcslwr, _mbslwr, _strlwr_l, _wcslwr_l, _mbslwr_l

_strlwr_s, _strlwr_s_l, _mbslwr_s, _mbslwr_s_l, _wcslwr_s, _wcslwr_s_l

_strncoll, _wcsncoll, _mbsncoll, _strncoll_l, _wcsncoll_l, _mbsncoll_l

_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l

_strnicoll, _wcsnicoll, _mbsnicoll, _strnicoll_l, _wcsnicoll_l, _mbsnicoll_l

_strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l

_strnset_s, _strnset_s_l, _wcsnset_s, _wcsnset_s_l, _mbsnset_s, _mbsnset_s_l

_strrev, _wcsrev, _mbsrev, _mbsrev_l

_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l

_strset_s, _strset_s_l, _wcsset_s, _wcsset_s_l, _mbsset_s, _mbsset_s_l

_strtime, _wstrtime

_strtime_s, _wstrtime_s

_strtoi64, _wcstoi64, _strtoi64_l, _wcstoi64_l

_strtoui64, _wcstoui64, _strtoui64_l, _wcstoui64_l

_strupr, _strupr_l, _mbsupr, _mbsupr_l, _wcsupr_l, _wcsupr

_strupr_s, _strupr_s_l, _mbsupr_s, _mbsupr_s_l, _wcsupr_s, _wcsupr_s_l

_swab

_tell, _telli64

_tempnam, _wtempnam, tmpnam, _wtmpnam

_tempnam_dbg, _wtempnam_dbg

_tzset

_ultoa, _ultow

_ultoa_s, _ultow_s

_umask

_umask_s

_ungetc_nolock, _ungetwc_nolock

_ungetch, _ungetwch, _ungetch_nolock, _ungetwch_nolock

_unlink, _wunlink

_unlock_file

_utime, _utime32 _utime64, _wutime, _wutime32, _wutime64

_vcprintf, _vcprintf_l, _vcwprintf, _vcwprintf_l

_vcprintf_p, _vcprintf_p_l, _vcwprintf_p, _vcwprintf_p_l

_vcprintf_s, _vcprintf_s_l, _vcwprintf_s, _vcwprintf_s_l

_vfprintf_p, _vfprintf_p_l, _vfwprintf_p, _vfwprintf_p_l

_vprintf_p, _vprintf_p_l, _vwprintf_p, _vwprintf_p_l

_vscprintf, _vscprintf_l, _vscwprintf, _vscwprintf_l

_vscprintf_p, _vscprintf_p_l, _vscwprintf_p, _vscwprintf_p_l

_vsprintf_p, _vsprintf_p_l, _vswprintf_p, _vswprintf_p_l

_write

abort

abs, _abs64

acos, acosf

Alphabetical Function Reference

asctime, _wasctime

asctime_s, _wasctime_s

asin, asinf

assert (CRT)

atan, atanf, atan2, atan2f

atexit

atof, _atof_l, _wtof, _wtof_l

atoi, _atoi_l, _wtoi, _wtoi_l

atol, _atol_l, _wtol, _wtol_l

Bessel Functions

Bessel Functions: _j0, _j1, _jn

Bessel Functions: _y0, _y1, _yn

bsearch

bsearch_s

btowc

calloc

ceil, ceilf

clearerr

clearerr_s

clock

cos, cosf, cosh, coshf

ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64

ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s

difftime, _difftime32, _difftime64

div

exit, _exit

exp, expf

fabs, fabsf

fclose, _fcloseall

feof

ferror

fflush

fgetc, fgetwc

fgetpos

fgets, fgetws

Flag Directives

floor, floorf

fmod, fmodf

fopen, _wfopen

fopen_s, _wfopen_s

Format Specification Fields: printf and wprintf Functions

Format Specification Fields: scanf and wscanf Functions

fprintf, _fprintf_l, fwprintf, _fwprintf_l

fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l

fputc, fputwc

fputs, fputws

fread

free

freopen, _wfreopen

freopen_s, _wfreopen_s

frexp

fscanf, _fscanf_l, fwscanf, _fwscanf_l

fscanf_s, _fscanf_s_l, fwscanf_s, _fwscanf_s_l

fseek, _fseeki64

fsetpos

ftell, _ftelli64

fwrite

getc, getwc

getchar, getwchar

getenv, _wgetenv

getenv_s, _wgetenv_s

gets, _getws

gets_s, _getws_s

gmtime, _gmtime32, _gmtime64

is, isw Routines

isalnum, iswalnum, _isalnum_l, _iswalnum_l

isalpha, iswalpha, _isalpha_l, _iswalpha_l

iscntrl, iswcntrl, _iscntrl_l, _iswcntrl_l

__iscsym, __iswcsym, __iscsymf, __iswcsymf, _iscsym_l, _iswcsym_l, _iscsymf_l, _iswcsymf_l

isdigit, iswdigit, _isdigit_l, _iswdigit_l

isgraph, iswgraph, _isgraph_l, _iswgraph_l

isleadbyte, _isleadbyte_l

islower, iswlower, _islower_l, _iswlower_l

isprint, iswprint, _isprint_l, _iswprint_l

ispunct, iswpunct, _ispunct_l, _iswpunct_l

isspace, iswspace, _isspace_l, _iswspace_l

isupper, _isupper_l, iswupper, _iswupper_l

isxdigit, iswxdigit, _isxdigit_l, _iswxdigit_l

labs

ldexp

ldiv

localeconv

localtime, _localtime32, _localtime64

log, logf, log10, log10f

longjmp

malloc

mbrlen

mbrtowc

mbsinit

mbsrtowcs

mbsrtowcs_s

mbstowcs, _mbstowcs_l

mbstowcs_s, _mbstowcs_s_l

mbtowc, _mbtowc_l

memchr, wmemchr

memcmp, wmemcmp

memcpy, wmemcpy

memcpy_s, wmemcpy_s

memmove, wmemmove

memmove_s, wmemmove_s

memset, wmemset

mktime, _mktime32, _mktime64

modf, modff

offsetof

operator delete (CRT)

operator delete[] (CRT)

operator new (CRT)

operator new[] (CRT)

perror, _wperror

pow, powf

Precision Specification

printf Type Field Characters

printf Width Specification

printf, _printf_l, wprintf, _wprintf_l

printf_p Positional Parameters

printf_s, _printf_s_l, wprintf_s, _wprintf_s_l

putc, putwc

putchar, putwchar

puts, _putws

qsort

qsort_s

raise

rand

rand_s

realloc

remove, _wremove

rename, _wrename

rewind

scanf Type Field Characters

scanf Width Specification

scanf, _scanf_l, wscanf, _wscanf_l

scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l

set_terminate (CRT)

set_unexpected (CRT)

setbuf

setjmp

setlocale, _wsetlocale

setvbuf

signal

sin, sinf, sinh, sinhf

Size and Distance Specification

sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l

sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l

sqrt, sqrtf

srand

sscanf, _sscanf_l, swscanf, _swscanf_l

sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l

strcat, wcscat, _mbscat

strcat_s, wcscat_s, _mbscat_s

strchr, wcschr, _mbschr, _mbschr_l

strcmp, wcscmp, _mbscmp

strcoll Functions

strcoll, wcscoll, _mbscoll, _strcoll_l, _wcscoll_l, _mbscoll_l

strcpy, wcscpy, _mbscpy

strcpy_s, wcscpy_s, _mbscpy_s

strcspn, wcscspn, _mbscspn, _mbscspn_l

strerror, _strerror, _wcserror, __wcserror

strerror_s, _strerror_s, _wcserror_s, __wcserror_s

strftime, wcsftime, _strftime_l, _wcsftime_l

String to Numeric Value Functions

strlen, strlen_l, wcslen, wcslen_l, _mbslen, _mbslen_l, _mbstrlen, _mbstrlen_l

strncat, _strncat_l, wcsncat, wcsncat_l, _mbsncat _mbsncat_l

strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l

strncmp, wcsncmp, _mbsncmp, _mbsncmp_l

strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l

strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l

strnlen, strnlen_l, wcsnlen, wcsnlen_l, _mbsnlen, _mbsnlen_l, _mbstrnlen, _mbstrnlen_l

strpbrk, wcspbrk, _mbspbrk, _mbspbrk_l

strrchr, wcsrchr, _mbsrchr, _mbsrchr_l

strspn, wcsspn, _mbsspn, _mbsspn_l

strstr, wcsstr, _mbsstr, _mbsstr_l

strtod, _strtod_l, wcstod, _wcstod_l

strtok, _strtok_l, wcstok, _wcstok_l, _mbstok, _mbstok_l

strtok_s, _strtok_s_l, wcstok_s, _wcstok_s_l, _mbstok_s, _mbstok_s_l

strtol, wcstol, _strtol_l, _wcstol_l

strtoul, _strtoul_l, wcstoul, _wcstoul_l

strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

system, _wsystem

tan, tanf, tanh, tanhf

terminate (CRT)

time, _time32, _time64

tmpfile

tmpfile_s

tmpnam_s, _wtmpnam_s

to Functions

tolower, _tolower, towlower, _tolower_l, _towlower_l

toupper, _toupper, towupper, _toupper_l, _towupper_l

towctrans

unexpected (CRT)

ungetc, ungetwc

va_arg, va_end, va_start

vfprintf, _vfprintf_l, vfwprintf, _vfwprintf_l

vfprintf_s, _vfprintf_s_l, vfwprintf_s, _vfwprintf_s_l

vprintf Functions

vprintf, _vprintf_l, vwprintf, _vwprintf_l

vprintf_s, _vprintf_s_l, vwprintf_s, _vwprintf_s_l

vsnprintf, _vsnprintf, _vsnprintf_l, _vsnwprintf, _vsnwprintf_l

vsnprintf_s, _vsnprintf_s, _vsnprintf_s_l, _vsnwprintf_s, _vsnwprintf_s_l

vsprintf, _vsprintf_l, vswprintf, _vswprintf_l, __vswprintf_l

vsprintf_s, _vsprintf_s_l, vswprintf_s, _vswprintf_s_l

wcrtomb

wcrtomb_s

wcsrtombs

wcsrtombs_s

wcstombs, _wcstombs_l

wcstombs_s, _wcstombs_s_l

wctob

wctomb, _wctomb_l

wctomb_s, _wctomb_s_l

wctrans

wctype

See Also

Other Resources

Run-Time Library Reference

__isascii, iswascii

Determines whether a particular character is an ASCII character.

```
int __isascii(
   int c
);
int iswascii(
   wint_t c
);
```

Parameters

c

Integer to test.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of an ASCII character. **__isascii** returns a nonzero value if *c* is an ASCII character (in the range 0x00 – 0x7F). **iswascii** returns a nonzero value if *c* is a wide-character representation of an ASCII character. Each of these routines returns 0 if *c* does not satisfy the test condition.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_istascii	**isascii**	**isascii**	**iswascii**

Requirements

Routine	Required header	Compatibility
__isascii	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
iswascii	<ctype.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

.NET Framework Equivalent

System::Char::IsLetter

See Also

Reference

Character Classification
Locale
is, isw Routines

__max

Returns the larger of two values.

```
type __max(
      type a,
      type b
);
```

Parameters

type

Any numeric data type.

a, b

Values of any numeric type to be compared.

Return Value

__max returns the larger of its arguments.

Remarks

The **__max** macro compares two values and returns the value of the larger one. The arguments can be of any numeric data type, signed or unsigned. Both arguments and the return value must be of the same data type.

Requirements

Routine	Required header	Compatibility
__max	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

For more information, see the example for __min.

.NET Framework Equivalent

System::Math::Max

See Also

Reference

Floating-Point Support

__min

__min

Returns the smaller of two values.

```
type __min(
      type a,
      type b
);
```

Parameters

type

Any numeric data type.

a, b

Values of any numeric type to be compared.

Return Value

The smaller of the two arguments.

Remarks

The **__min** macro compares two values and returns the value of the smaller one. The arguments can be of any numeric data type, signed or unsigned. Both arguments and the return value must be of the same data type.

Requirements

Routine	Required header	Compatibility
__min	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

```
// crt_minmax.c

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   int a = 10;
   int b = 21;

   printf( "The larger of %d and %d is %d\n",  a, b, __max( a, b ) );
   printf( "The smaller of %d and %d is %d\n", a, b, __min( a, b ) );
}
```

Output

```
The larger of 10 and 21 is 21
The smaller of 10 and 21 is 10
```

.NET Framework Equivalent

System::Math::Min

See Also

Reference

Floating-Point Support

__max

__toascii

Converts characters.

```
int __toascii(
   int c
);
```

Parameters

c

Character to convert.

Return Value

__toascii converts a copy of *c* if possible and returns the result. There is no return value reserved to indicate an error.

Remarks

The **__toascii** routine converts the given character to an ASCII character.

Requirements

Routine	Required header	Compatibility
__toascii	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

.NET Framework Equivalent

System::IFormattable::ToString

See Also

Reference

Data Conversion
is, isw Routines
to Functions

abort

Aborts the current process and returns an error code.

```
void abort( void );
```

Return Value

abort does not return control to the calling process. By default, it terminates the current process and returns an exit code of 3.

Remarks

By default, the **abort** routine prints the message:

"This application has requested the Runtime to terminate it in an unusual way. Please contact the application's support team for more information."

It then calls **raise(SIGABRT)**. The action taken in response to the **SIGABRT** signal depends on what action has been defined for that signal in a prior call to the **signal** function. The default **SIGABRT** action is for the calling process to terminate with exit code 3, returning control to the calling process or operating system. **abort** does not flush stream buffers or do **atexit**/**_onexit** processing.

abort determines the destination of the message based on the type of application that called the routine. Console applications always receive the message through **stderr**. In a single or multithreaded Windows-based application, **abort** calls the Windows MessageBox function to create a message box to display the message with an **OK** button. When the user clicks **OK**, the program aborts immediately. The message can be suppressed by calling _set_abort_behavior with the appropriate arguments.

When the application is linked with a debug version of the run-time libraries, **abort** creates a message box with three buttons: **Abort**, **Retry**, and **Ignore**. If the user clicks **Abort**, the program aborts immediately. If the user clicks **Retry**, the debugger is called and the user can debug the program if just-in-time (JIT) debugging is enabled. If the user clicks **Ignore**, **abort** continues with its normal execution: creating the message box with the **OK** button.

In Visual C++ 2005, **abort** also invokes an error reporting mechanism to report failures to Microsoft. This behavior can be disabled by calling **_set_abort_behavior**.

For more information on CRT debugging, see CRT Debugging Techniques.

Requirements

Routine	Required header	Compatibility
abort	<process.h> or <stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

The following program tries to open a file and aborts if the attempt fails.

```
// crt_abort.c
// compile with: /c
// This program demonstrates the use of
// the abort function by attempting to open a file
// and aborts if the attempt fails.

#include  <stdio.h>
#include  <stdlib.h>

int main( void )
{
    FILE    *stream = NULL;
    errno_t err = 0;

    err = fopen_s(&stream, "NOSUCHF.ILE", "r" );
```

```
    if ((err != 0) || (stream == NULL))
    {
        perror( "File could not be opened" );
        abort();
    }
    else
    {
        fclose( stream );
    }
}
```

Output

```
File could not be opened: No such file or directory

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Using abort
The abort Function
Process and Environment Control
_exec, _wexec Functions
exit, _exit
raise
signal
_spawn, _wspawn Functions
_DEBUG
_set_abort_behavior

abs, _abs64

Calculate the absolute value.

```
int abs(
   int n
);
long abs(
   long n
);   // C++ only
double abs(
   double n
);   // C++ only
long double abs(
   long double n
);   // C++ only
float abs(
   float n
);   // C++ only
__int64 _abs64(
   __int64 n
);
```

Parameters

n

Integer value.

Return Value

The **abs** function returns the absolute value of its parameter. There is no error return.

Remarks

C++ allows overloading, so you can call overloads of **abs**. In a C program, **abs** always takes and returns an int.

Requirements

Routine	Required header	Compatibility
abs	<stdlib.h> or <math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_abs64	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

This program computes and displays the absolute values of several numbers.

```
// crt_abs.c
// This program demonstrates the user of the abs function
// by computing and displaying the absolute values of
// several numbers.

#include  <stdio.h>
#include  <math.h>
#include  <stdlib.h>

int main( void )
{
    int     ix = -4,
            iy;
    long    lx = -41567L,
```

```
            ly;
    double  dx = -3.141593,
            dy;
    __int64 wx = -1, wy;

    // absolute 64 bit integer value
    wy = _abs64( wx );
    printf_s( "The absolute value of %I64x is %I64x\n", wx, wy);

    // absolute 32 bit integer value
    iy = abs( ix );
    printf_s( "The absolute value of %d is %d\n", ix, iy);

    // absolute long integer value
    ly = labs( lx );
    printf_s( "The absolute value of %ld is %ld\n", lx, ly);

    // absolute double value
    dy = fabs( dx );
    printf_s( "The absolute value of %f is %f\n", dx, dy );
}
```

Output

```
The absolute value of ffffffffffffffff is 1
The absolute value of -4 is 4
The absolute value of -41567 is 41567
The absolute value of -3.141593 is 3.141593
```

.NET Framework Equivalent

System::Math::Abs

See Also

Reference

Data Conversion
Floating-Point Support
_cabs
fabs, fabsf
labs

access (CRT)

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _access or security-enhanced _access_s instead.

_access, _waccess

Determines if a file is read-only or not. More secure versions are available; see _access_s, _waccess_s.

```
int _access(
   const char *path,
   int mode
);
int _waccess(
   const wchar_t *path,
   int mode
);
```

Parameters

path

File or directory path.

mode

Read/write attribute.

Return Value

Each function returns 0 if the file has the given mode. The function returns –1 if the named file does not exist or does not have the given mode; in this case, **errno** is set as shown in the following table.

EACCES

Access denied: the file's permission setting does not allow specified access.

ENOENT

File name or path not found.

EINVAL

Invalid parameter.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

When used with files, the **_access** function determines whether the specified file or directory exists and has the attributes specified by the value of *mode*. When used with directories, **_access** determines only whether the specified directory exists; in Windows NT 4.0 and Windows 2000, all directories have read and write access.

***mode* value**	**Checks file for**
00	Existence only
02	Write-only
04	Read-only
06	Read and write

This function only checks whether the file and directory are read-only or not, it does not check the filesystem security settings. For that you need an access token. For more information on filesystem security, see Access Tokens. An ATL class exists to provide this functionality; see CAccessToken Class.

_waccess is a wide-character version of **_access**; the *path* argument to **_waccess** is a wide-character string. **_waccess** and **_access** behave identically otherwise.

This function validates its parameters. If *path* is **NULL** or *mode* does not specify a valid mode, the invalid parameter handler is

invoked, as described in Parameter Validation. If execution is allowed to continue, the function sets **errno** to **EINVAL** and returns -1.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_taccess	**_access**	**_access**	**_waccess**

Requirements

Routine	Required header	Optional headers	Compatibility
_access	<io.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_waccess	<wchar.h> or <io.h>	<errno.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

The following example uses **_access** to check the file named crt_ACCESS.C to see whether it exists and whether writing is allowed.

```
// crt_access.c
// compile with: /W1
// This example uses _access to check the file named
// crt_ACCESS.C to see if it exists and if writing is allowed.

#include  <io.h>
#include  <stdio.h>
#include  <stdlib.h>

int main( void )
{
    // Check for existence.
    if( (_access( "crt_ACCESS.C", 0 )) != -1 )
    {
        printf_s( "File crt_ACCESS.C exists.\n" );

        // Check for write permission.
        // Assume file is read-only.
        if( (_access( "crt_ACCESS.C", 2 )) == -1 )
            printf_s( "File crt_ACCESS.C does not have write permission.\n" );
    }
}
```

Output

```
File crt_ACCESS.C exists.
File crt_ACCESS.C does not have write permission.
```

.NET Framework Equivalent

System.IO.FileAccess

See Also

Reference

File Handling
_chmod, _wchmod
_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32
_open, _wopen
_stat, _wstat Functions

_access_s, _waccess_s

Determines file read/write permissions. This is a version of _access, _waccess with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _access_s(
   const char *path,
   int mode
);
errno_t _waccess_s(
   const wchar_t *path,
   int mode
);
```

Parameters

path

File or directory path.

mode

Permission setting.

Return Value

Each function returns 0 if the file has the given mode. The function returns an error code if the named file does not exist or is not accessible in the given mode. In this case, the function returns an error code from the set as follows and also sets **errno** to the same value.

EACCES

Access denied. The file's permission setting does not allow specified access.

ENOENT

File name or path not found.

EINVAL

Invalid parameter.

For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

When used with files, the **_access_s** function determines whether the specified file exists and can be accessed as specified by the value of *mode*. When used with directories, **_access_s** determines only whether the specified directory exists. In Windows NT 4.0 and Windows 2000, all directories have read and write access.

mode value	Checks file for
00	Existence only.
02	Write permission.
04	Read permission.
06	Read and write permission.

Permission to read or write the file is not enough to ensure the ability to open a file. For example, if a file is locked by another process, it might not be accessible even though **_access_s** returns 0.

_waccess_s is a wide-character version of **_access_s**, where the *path* argument to **_waccess_s** is a wide-character string. Otherwise, **_waccess_s** and **_access_s** behave identically.

These functions validate their parameters. If *path* is **NULL** or *mode* does not specify a valid mode, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **EINVAL**.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_taccess_s	**_access_s**	**_access_s**	**_waccess_s**

Requirements

Routine	Required header	Optional header	Compatibility
_access_s	<io.h>	<errno.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_waccess_s	<wchar.h> or <io.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

This example uses **_access_s** to check the file named crt_access_s.c to see whether it exists and whether writing is allowed.

```
// crt_access_s.c

#include <io.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    errno_t err = 0;

    // Check for existence.
    if ((err = _access_s( "crt_access_s.c", 0 )) == 0 )
    {
        printf_s( "File crt_access_s.c exists.\n" );

        // Check for write permission.
        if ((err = _access_s( "crt_access_s.c", 2 )) == 0 )
        {
            printf_s( "File crt_access_s.c does have "
                      "write permission.\n" );
        }
        else
        {
            printf_s( "File crt_access_s.c does not have "
                      "write permission.\n" );
        }
    }
    else
    {
        printf_s( "File crt_access_s.c does not exist.\n" );
    }
}
```

Output

```
File crt_access_s.c exists.
File crt_access_s.c does not have write permission.
```

.NET Framework Equivalent

System.IO.FileAccess

See Also

Reference

File Handling

_access, _waccess

_chmod, _wchmod

_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32

_open, _wopen

_stat, _wstat Functions

acos, acosf

Calculate the arccosine.

```
double acos(
   double x
);
float acos(
   float x
);   // C++ only
long double acos(
   long double x
);   // C++ only
float acosf(
   float x
);
```

Parameters

x

Value between –1 and 1 whose arccosine is to be calculated.

Return Value

The **acos** function returns the arccosine of *x* in the range 0 to π radians.

If *x* is less than –1 or greater than 1, **acos** returns an indefinite by default.

Input	SEH Exception	Matherr Exception
± ∞	**INVALID**	**_DOMAIN**
± QNAN,IND	none	**_DOMAIN**
|x|>1	**INVALID**	**_DOMAIN**

Remarks

C++ allows overloading, so you can call overloads of **acos**. In a C program, **acos** always takes and returns a double.

Requirements

Routine	Required header	Optional headers	Compatibility
acos, **acosf**	<math.h>	<errno.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

This program prompts for a value in the range -1 to 1. Input values outside this range will produce **_DOMAIN** error messages. If a valid value is entered, the program prints the arcsine and the arccosine of that value.

```
// crt_asincos.c
// arguments: 0

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main( int ac, char* av[] )
{
```

```
    double  x,
            y;
    errno_t err;

    // argument checking
    if (ac != 2)
    {
        fprintf_s( stderr, "Usage: %s <number between -1 and 1>\n",
                   av[0]);
        return 1;
    }

    // Convert argument into a double value
    if ((err = sscanf_s( av[1], "%lf", &x )) != 1)
    {
        fprintf_s( stderr, "Error converting argument into ",
                   "double value.\n");
        return 1;
    }

    // Arcsine of X
    y = asin( x );
    printf_s( "Arcsine of %f = %f\n", x, y );

    // Arccosine of X
    y = acos( x );
    printf_s( "Arccosine of %f = %f\n", x, y );
}
```

Output

```
Arcsine of 0.000000 = 0.000000
Arccosine of 0.000000 = 1.570796
```

.NET Framework Equivalent

System::Math::Acos

See Also

Reference

Floating-Point Support
asin, asinf
atan, atanf, atan2, atan2f
cos, cosf, cosh, coshf
_matherr
sin, sinf, sinh, sinhf
tan, tanf, tanh, tanhf

_aligned_free

Frees a block of memory that was allocated with _aligned_malloc or _aligned_offset_malloc.

```
void _aligned_free (
   void *memblock
);
```

Parameters

memblock

A pointer to the memory block that was returned to the **_aligned_malloc** or **_aligned_offset_malloc** function.

Remarks

_aligned_free is marked **__declspec(noalias)**, meaning that the function is guaranteed not to modify global variables. For more information, see noalias.

This function does not validate its parameter, unlike the other _aligned CRT functions. If *memblock* is a **NULL** pointer, this function simply performs no actions. It does not change **errno** and it does not invoke the invalid parameter handler. If an error occurs in the function due to not using _aligned functions previously to allocate the block of memory or a misalignment of memory occurs due to some unforeseen calamity, the function generates a debug report from the _RPT, _RPTF, _RPTW, _RPTFW Macros.

Requirements

Routine	Required header	Compatibility
_aligned_free	<malloc.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

For more information, see _aligned_malloc.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Data Alignment

_aligned_malloc

Allocates memory on a specified alignment boundary.

```
void * _aligned_malloc(
    size_t size,
    size_t alignment
);
```

Parameters

size

Size of the requested memory allocation.

alignment

The alignment value, which must be an integer power of 2.

Return Value

A pointer to the memory block that was allocated or **NULL** if the operation failed.

Remarks

_aligned_malloc is based on **malloc**. For more information about using **_aligned_malloc**, see malloc.

_aligned_malloc is marked **__declspec(noalias)** and **__declspec(restrict)**, meaning that the function is guaranteed not to modify global variables and that the pointer returned is not aliased. For more information, see noalias and restrict.

In Visual C++ 2005, this function sets **errno** to **ENOMEM** if the memory allocation failed or if the requested size was greater than **_HEAP_MAXREQ**. For more information about **errno**, see errno, _doserrno, _sys_errlist, and _sys_nerr. Also, **_aligned_malloc** validates its parameters. If *alignment* is not a power of 2 or *size* is zero, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns **NULL** and sets **errno** to **EINVAL**.

Requirements

Routine	Required header	Compatibility
_aligned_malloc	<malloc.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

```
// crt_aligned_malloc.c

#include <malloc.h>
#include <stdio.h>

int main() {
    void    *ptr;
    size_t  alignment,
            off_set;

    // Note alignment should be 2^N where N is any positive int.
    alignment = 16;
    off_set = 5;

    // Using _aligned_malloc
    ptr = _aligned_malloc(100, alignment);
    if (ptr == NULL)
    {
        printf_s( "Error allocation aligned memory.");
        return -1;
    }
```

```
    if (((int)ptr % alignment ) == 0)
        printf_s( "This pointer, %d, is aligned on %d\n",
                  ptr, alignment);
    else
        printf_s( "This pointer, %d, is not aligned on %d\n",
                  ptr, alignment);

    // Using _aligned_realloc
    ptr = _aligned_realloc(ptr, 200, alignment);
    if ( ((int)ptr % alignment ) == 0)
        printf_s( "This pointer, %d, is aligned on %d\n",
                  ptr, alignment);
    else
        printf_s( "This pointer, %d, is not aligned on %d\n",
                  ptr, alignment);
    _aligned_free(ptr);

    // Using _aligned_offset_malloc
    ptr = _aligned_offset_malloc(200, alignment, off_set);
    if (ptr == NULL)
    {
        printf_s( "Error allocation aligned offset memory.");
        return -1;
    }
    if ( ( (((int)ptr) + off_set) % alignment ) == 0)
        printf_s( "This pointer, %d, is offset by %d on alignment of %d\n",
                  ptr, off_set, alignment);
    else
        printf_s( "This pointer, %d, does not satisfy offset %d "
                  "and alignment %d\n",ptr, off_set, alignment);

    // Using _aligned_offset_realloc
    ptr = _aligned_offset_realloc(ptr, 200, alignment, off_set);
    if (ptr == NULL)
    {
        printf_s( "Error reallocation aligned offset memory.");
        return -1;
    }
    if ( ( (((int)ptr) + off_set) % alignment ) == 0)
        printf_s( "This pointer, %d, is offset by %d on alignment of %d\n",
                  ptr, off_set, alignment);
    else
        printf_s( "This pointer, %d, does not satisfy offset %d and "
                  "alignment %d\n", ptr, off_set, alignment);

    // Note that _aligned_free works for both _aligned_malloc
    // and _aligned_offset_malloc. Using free is illegal.
    _aligned_free(ptr);
}
```

Sample Output

```
This pointer, 3280880, is aligned on 16
This pointer, 3280880, is aligned on 16
This pointer, 3280891, is offset by 5 on alignment of 16
This pointer, 3280891, is offset by 5 on alignment of 16
```

See Also

Reference

Data Alignment

_aligned_offset_malloc

Allocates memory on a specified alignment boundary.

```
void * _aligned_offset_malloc(
   size_t size,
   size_t alignment,
   size_t offset
);
```

Parameters

[in] *size*

The size of the requested memory allocation.

[in] *alignment*

The alignment value, which must be an integer power of 2.

[in] *offset*

The offset into the memory allocation to force the alignment.

Return Value

A pointer to the memory block that was allocated or **NULL** if the operation failed.

Remarks

_aligned_offset_malloc is useful in situations where alignment is needed on a nested element; for example, if alignment was needed on a nested class.

_aligned_offset_malloc is based on **malloc**; for more information, see malloc.

_aligned_offset_malloc is marked **__declspec(noalias)** and **__declspec(restrict)**, meaning that the function is guaranteed not to modify global variables and that the pointer returned is not aliased. For more information, see noalias and restrict.

In Visual C++ 2005, this function sets **errno** to **ENOMEM** if the memory allocation failed or if the requested size was greater than **_HEAP_MAXREQ**. For more information about **errno**, see errno, _doserrno, _sys_errlist, and _sys_nerr. Also, **_aligned_offset_malloc** validates its parameters. If *alignment* is not a power of 2 or if *offset* is greater than or equal to *size* and nonzero, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns **NULL** and sets **errno** to **EINVAL**.

Requirements

Routine	Required header	Compatibility
_aligned_offset_malloc	<malloc.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

For more information, see _aligned_malloc.

See Also

Reference

Data Alignment

_aligned_offset_realloc

Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc.

```
void * _aligned_offset_realloc(
   void *memblock,
   size_t size,
   size_t alignment,
   size_t offset
);
```

Parameters

memblock

The current memory block pointer.

size

The size of the memory allocation.

alignment

The alignment value, which must be an integer power of 2.

offset

The offset into the memory allocation to force the alignment.

Return Value

_aligned_offset_realloc returns a void pointer to the reallocated (and possibly moved) memory block. The return value is **NULL** if the size is zero and the buffer argument is not **NULL**, or if there is not enough available memory to expand the block to the given size. In the first case, the original block is freed. In the second case, the original block is unchanged. The return value points to a storage space that is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void, use a type cast on the return value.

_aligned_offset_realloc is marked **__declspec(noalias)** and **__declspec(restrict)**, meaning that the function is guaranteed not to modify global variables and that the pointer returned is not aliased. For more information, see noalias and restrict.

Remarks

Like _aligned_offset_malloc, **_aligned_offset_realloc** allows a structure to be aligned at an offset within the structure.

_aligned_offset_realloc is based on **malloc**. For more information about using **_aligned_offset_malloc**, see malloc. If *memblock* is **NULL**, the function calls **_aligned_offset_malloc** internally.

In Visual C++ 2005, this function sets **errno** to **ENOMEM** if the memory allocation failed or if the requested size was greater than **_HEAP_MAXREQ**. For more information about **errno**, see errno, _doserrno, _sys_errlist, and _sys_nerr. Also, **_aligned_offset_realloc** validates its parameters. If *alignment* is not a power of 2 or if *offset* is greater than or equal to *size* and nonzero, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns **NULL** and sets **errno** to **EINVAL**.

Requirements

Routine	Required header	Compatibility
_aligned_offset_realloc	<malloc.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

For more information, see _aligned_malloc.

See Also

Reference

Data Alignment

_aligned_offset_recalloc

Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc and initializes the memory to 0.

```
void * _aligned_offset_recalloc(
   void *memblock,
   size_t num,
   size_t size,
   size_t alignment,
   size_t offset
);
```

Parameters

memblock

The current memory block pointer.

num

Number of elements.

size

Length in bytes of each element.

alignment

The alignment value, which must be an integer power of 2.

offset

The offset into the memory allocation to force the alignment.

Return Value

_aligned_offset_recalloc returns a void pointer to the reallocated (and possibly moved) memory block. The return value is **NULL** if the size is zero and the buffer argument is not **NULL**, or if there is not enough available memory to expand the block to the given size. In the first case, the original block is freed. In the second case, the original block is unchanged. The return value points to a storage space that is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void, use a type cast on the return value.

_aligned_offset_recalloc is marked **__declspec(noalias)** and **__declspec(restrict)**, meaning that the function is guaranteed not to modify global variables and that the pointer returned is not aliased. For more information, see noalias and restrict.

Remarks

Like _aligned_offset_malloc, **_aligned_offset_recalloc** allows a structure to be aligned at an offset within the structure.

_aligned_offset_recalloc is based on **malloc**. For more information about using **_aligned_offset_malloc**, see malloc. If *memblock* is **NULL**, the function calls **_aligned_offset_malloc** internally.

In Visual C++ 2005, this function sets **errno** to **ENOMEM** if the memory allocation failed or if the requested size (*num* * *size*) was greater than **_HEAP_MAXREQ**. For more information about **errno**, see errno, _doserrno, _sys_errlist, and _sys_nerr. Also, **_aligned_offset_recalloc** validates its parameters. If *alignment* is not a power of 2 or if *offset* is greater than or equal to the requested size and nonzero, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns **NULL** and sets **errno** to **EINVAL**.

Requirements

Routine	Required header	Compatibility
_aligned_offset_recalloc	<malloc.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Data Alignment

_recalloc

_aligned_recalloc

_aligned_realloc

Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc.

```
void * _aligned_realloc(
   void *memblock,
   size_t size,
   size_t alignment
);
```

Parameters

[in] *memblock*

The current memory block pointer.

[in] *size*

The size of the requested memory allocation.

[in] *alignment*

The alignment value, which must be an integer power of 2.

Return Value

_aligned_realloc returns a void pointer to the reallocated (and possibly moved) memory block. The return value is **NULL** if the size is zero and the buffer argument is not **NULL**, or if there is not enough available memory to expand the block to the given size. In the first case, the original block is freed. In the second, the original block is unchanged. The return value points to a storage space that is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void, use a type cast on the return value.

It is an error to reallocate memory and change the alignment of a block.

Remarks

_aligned_realloc is based on **malloc**. For more information about using **_aligned_offset_malloc**, see malloc.

In Visual C++ 2005, this function sets **errno** to **ENOMEM** if the memory allocation failed or if the requested size was greater than **_HEAP_MAXREQ**. For more information about **errno**, see errno, _doserrno, _sys_errlist, and _sys_nerr. Also, **_aligned_realloc** validates its parameters. If *alignment* is not a power of 2, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns **NULL** and sets **errno** to **EINVAL**.

Requirements

Routine	Required header	Compatibility
_aligned_realloc	<malloc.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

For more information, see _aligned_malloc.

See Also

Reference

Data Alignment

_aligned_recalloc

Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc and initializes the memory to 0.

```
void * _aligned_recalloc(
   void *memblock,
   size_t num,
   size_t size,
   size_t alignment
);
```

Parameters

[in] *memblock*

The current memory block pointer.

[in] *num*

The number of elements.

[in] *size*

The size in bytes of each element.

[in] *alignment*

The alignment value, which must be an integer power of 2.

Return Value

_aligned_recalloc returns a void pointer to the reallocated (and possibly moved) memory block. The return value is **NULL** if the size is zero and the buffer argument is not **NULL**, or if there is not enough available memory to expand the block to the given size. In the first case, the original block is freed. In the second case, the original block is unchanged. The return value points to a storage space that is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void, use a type cast on the return value.

It is an error to reallocate memory and change the alignment of a block.

Remarks

_aligned_recalloc is based on **malloc**. For more information about using **_aligned_offset_malloc**, see malloc.

In Visual C++ 2005, this function sets **errno** to **ENOMEM** if the memory allocation failed or if the requested size was greater than **_HEAP_MAXREQ**. For more information about **errno**, see errno, _doserrno, _sys_errlist, and _sys_nerr. Also, **_aligned_recalloc** validates its parameters. If *alignment* is not a power of 2, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns **NULL** and sets **errno** to **EINVAL**.

Requirements

Routine	Required header	Compatibility
_aligned_recalloc	<malloc.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Data Alignment
_recalloc
_aligned_offset_recalloc

_alloca

Allocates memory on the stack. This function is deprecated because a more secure version is available; see _malloca.

```
void *_alloca(
   size_t size
);
```

Parameters

[in] *size*

Bytes to be allocated from the stack.

Return Value

The **_alloca** routine returns a **void** pointer to the allocated space, which is guaranteed to be suitably aligned for storage of any type of object. If *size* is 0, **_alloca** allocates a zero-length item and returns a valid pointer to that item.

A stack overflow exception is generated if the space cannot be allocated. The stack overflow exception is not a C++ exception; it is a structured exception. Instead of using C++ exception handling, you must use Structured Exception Handling (SEH).

Remarks

_alloca allocates *size* bytes from the program stack. The allocated space is automatically freed when the calling function exits (not when the allocation merely passes out of scope). Therefore, do not pass the pointer value returned by **_alloca** as an argument to free.

There are restrictions to explicitly calling **_alloca** in an exception handler (EH). EH routines that run on x86-class processors operate in their own memory frame: They perform their tasks in memory space that is not based on the current location of the stack pointer of the enclosing function. The most common implementations include Windows NT structured exception handling (SEH) and C++ catch clause expressions. Therefore, explicitly calling **_alloca** in any of the following scenarios results in program failure during the return to the calling EH routine:

- Windows NT SEH exception filter expression: **__except** (**_alloca ()**)
- Windows NT SEH final exception handler: **__finally** {**_alloca ()** }
- C++ EH catch clause expression

However, **_alloca** can be called directly from within an EH routine or from an application-supplied callback that gets invoked by one of the EH scenarios previously listed.

Security Note
In Windows XP, if **_alloca** is called inside a try/catch block, you must call _resetstkoflw in the catch block.

In addition to the above restrictions, when using the/clr (Common Language Runtime Compilation) option, **_alloca** cannot be used in **__except** blocks. For more information, see /clr Restrictions.

Requirements

Routine	**Required header**	**Compatibility**
_alloca	<malloc.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

```
// crt_alloca.c
// This program demonstrates the use of
// _alloca and trapping any exceptions
// that may occur.
```

```
#include <windows.h>
#include <stdio.h>
#include <malloc.h>

int main()
{
    int     size = 1000;
    int     errcode = 0;
    void    *pData = NULL;

    // Note: Do not use try/catch for _alloca,
    // use __try/__except, since _alloca throws
    // Structured Exceptions, not C++ exceptions.

    __try {
        // An unbounded _alloca can easily result in a
        // stack overflow.
        // Checking for a size < 1024 bytes is recommended.
        if (size > 0 && size < 1024)
        {
            pData = _alloca( size );
            printf_s( "Allocated %d bytes of stack at 0x%p",
                      size, pData);
        }
        else
        {
            printf_s("Tried to allocate too many bytes.\n");
        }
    }

    // If an exception occured with the _alloca function
    __except( GetExceptionCode() == STATUS_STACK_OVERFLOW )
    {
        printf_s("_alloca failed!\n");

        // If the stack overflows, use this function to restore.
        errcode = _resetstkoflw();
        if (errcode)
        {
            printf_s("Could not reset the stack!\n");
            _exit(1);
        }
    };
}
```

Sample Output

```
Allocated 1000 bytes of stack at 0x0012FB50
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation
calloc
malloc
realloc
_resetstkoflw
_malloca

asctime, _wasctime

Convert a **tm** time structure to a character string. These functions are deprecated because more secure versions are available; see asctime_s, _wasctime_s.

```
char *asctime(
   const struct tm *timeptr
);
wchar_t *_wasctime(
   const struct tm *timeptr
);
```

Parameters

timeptr

Time/date structure.

Return Value

asctime returns a pointer to the character string result; **_wasctime** returns a pointer to the wide-character string result. There is no error return value.

Remarks

More secure versions of these functions are available; see asctime_s, _wasctime_s.

The **asctime** function converts a time stored as a structure to a character string. The *timeptr* value is usually obtained from a call to **gmtime** or **localtime**, which both return a pointer to a **tm** structure, defined in TIME.H.

timeptr member	**Value**
tm_hour	Hours since midnight (0–23)
tm_isdst	Positive if daylight saving time is in effect; 0 if daylight saving time is not in effect; negative if status of daylight saving time is unknown. The C run-time library assumes the United States' rules for implementing the calculation of Daylight Saving Time (DST).
tm_mday	Day of month (1–31)
tm_min	Minutes after hour (0–59)
tm_mon	Month (0–11; January = 0)
tm_sec	Seconds after minute (0–59)
tm_wday	Day of week (0–6; Sunday = 0)
tm_yday	Day of year (0–365; January 1 = 0)
tm_year	Year (current year minus 1900)

The converted character string is also adjusted according to the local time zone settings. For information about configuring the local time, see the time, _ftime, and localtime functions and the _tzset function for information about defining the time zone environment and global variables.

The string result produced by **asctime** contains exactly 26 characters and has the form `Wed Jan 02 02:03:55 1980\n\0`. A 24-hour clock is used. All fields have a constant width. The newline character and the null character occupy the last two positions of the string. **asctime** uses a single, statically allocated buffer to hold the return string. Each call to this function destroys the result of the previous call.

_wasctime is a wide-character version of **asctime**. **_wasctime** and **asctime** behave identically otherwise.

These functions validate their parameters. If *timeptr* is a null pointer, or if it contains out-of-range values, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns **NULL** and sets **errno** to **EINVAL**.

Generic-Text Routine Mapping

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tasctime	**asctime**	**asctime**	**_wasctime**

Requirements

Routine	Required header	Compatibility
asctime	<time.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wasctime	<time.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

This program places the system time in the long integer **aclock**, translates it into the structure **newtime** and then converts it to string form for output, using the **asctime** function.

```
// crt_asctime.c
// compile with: /W3

#include <time.h>
#include <stdio.h>

int main( void )
{
    struct tm   *newTime;
    time_t      szClock;

    // Get time in seconds
    time( &szClock );

    // Convert time to struct tm form
    newTime = localtime( &szClock );

    // Print local time as a string.
    printf_s( "Current date and time: %s", asctime( newTime ) ); // C4996
    // Note: asctime is deprecated; consider using asctime_s instead
}
```

Sample Output

```
Current date and time: Sun Feb 03 11:38:58 2002
```

.NET Framework Equivalent

- System::DateTime::ToLongDateString

- System::DateTime::ToLongTimeString
- System::DateTime::ToShortDateString
- System::DateTime::ToShortTimeString
- System::DateTime::ToString

See Also

Reference

Time Management
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
_ftime, _ftime32, _ftime64
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
time, _time32, _time64
_tzset
asctime_s, _wasctime_s

asctime_s, _wasctime_s

Convert a **tm** time structure to a character string. These functions are versions of asctime, _wasctime with security enhancements as described in Security Enhancements in the CRT.

```
errno_t asctime_s(
   char* buffer,
   size_t numberOfElements,
   const struct tm *_tm
);
errno_t _wasctime_s(
   wchar_t* buffer,
   size_t numberOfElements
   const struct tm *_tm
);
template <size_t size>
errno_t asctime_s(
   char (&buffer)[size],
   const struct tm *_tm
); // C++ only
template <size_t size>
errno_t _wasctime_s(
   wchar_t (&buffer)[size],
   const struct tm *_tm
); // C++ only
```

Parameters

buffer

[out] A pointer to a buffer to store the character string result. This function assumes a pointer to a valid memory location with a size specified by *numberOfElements*.

numberOfElements

[in] The size of the buffer used to store the result.

_tm

[in] Time/date structure. This function assumes a pointer to a valid **struct tm** object.

Return Value

Zero if successful. If there is a failure, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the return value is an error code. Error codes are defined in ERRNO.H. For more information, see errno Constants. The actual error codes returned for each error condition are shown in the following table.

Error Conditions

buffer	***numberOfElements***	***tm***	**Return**	**Value in *buffer***
NULL	Any	Any	**EINVAL**	Not modified
Not **NULL** (points to valid memory)	0	Any	**EINVAL**	Not modified
Not **NULL**	0< size < 26	Any	**EINVAL**	Empty string
Not **NULL**	>= 26	**NULL**	**EINVAL**	Empty string

Not **NULL**	> = 26	Invalid time structure or out of range values for components of the time	**EINVAL**	Empty string

Note
Error conditions for **wasctime_s** are similar to **asctime_s** with the exception that the size limit is measured in words.

Remarks

The **asctime** function converts a time stored as a structure to a character string. The *_tm* value is usually obtained from a call to **gmtime** or **localtime**. Both functions can be used to fill in a **tm** structure, as defined in TIME.H.

timeptr member	**Value**
tm_hour	Hours since midnight (0–23)
tm_isdst	Positive if daylight saving time is in effect; 0 if daylight saving time is not in effect; negative if status of daylight saving time is unknown. The C run-time library assumes the United States' rules for implementing the calculation of Daylight Saving Time (DST).
tm_mday	Day of month (1–31)
tm_min	Minutes after hour (0–59)
tm_mon	Month (0–11; January = 0)
tm_sec	Seconds after minute (0–59)
tm_wday	Day of week (0–6; Sunday = 0)
tm_yday	Day of year (0–365; January 1 = 0)
tm_year	Year (current year minus 1900)

The converted character string is also adjusted according to the local time zone settings. See the time, _time32, _time64, _ftime, _ftime32, _ftime64, and localtime_s, _localtime32_s, _localtime64_s functions for information about configuring the local time and the _tzset function for information about defining the time zone environment and global variables.

The string result produced by **asctime_s** contains exactly 26 characters and has the form `Wed Jan 02 02:03:55 1980\n\0`. A 24-hour clock is used. All fields have a constant width. The new line character and the null character occupy the last two positions of the string. The value passed in as the second parameter should be at least this big. If it is less, an error code, **EINVAL**, will be returned.

_wasctime_s is a wide-character version of **asctime_s**. **_wasctime_s** and **asctime_s** behave identically otherwise.

Generic-Text Routine Mapping

TCHAR.H routine	**_UNICODE & _MBCS not defined**	**_MBCS defined**	**_UNICODE defined**
_tasctime_s	**asctime_s**	**asctime_s**	**_wasctime_s**

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically,

eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

Requirements

Routine	Required header	Compatibility
asctime_s	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wasctime_s	<time.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Security

If the buffer pointer is not **NULL** and the pointer does not point to a valid buffer, the function will overwrite whatever is at the location. This can also result in an access violation.

A buffer overrun can occur if the size argument passed in is greater than the actual size of the buffer.

Example

This program places the system time in the long integer **aclock**, translates it into the structure **newtime** and then converts it to string form for output, using the **asctime_s** function.

```
// crt_asctime_s.c
#include <time.h>
#include <stdio.h>

struct tm newtime;
__time32_t aclock;

int main( void )
{
   char buffer[32];
   errno_t errNum;
   _time32( &aclock );   // Get time in seconds.
   _localtime32_s( &newtime, &aclock );   // Convert time to struct tm form.

   // Print local time as a string.

   errNum = asctime_s(buffer, 32, &newtime);
   if (errNum)
   {
       printf("Error code: %d", (int)errNum);
       return 1;
   }
   printf( "Current date and time: %s", buffer );
   return 0;
}
```

Sample Output

```
Current date and time: Wed May 14 15:30:17 2003
```

.NET Framework Equivalent

- System.DateTime.ToLongDateString
- System.DateTime.ToLongTimeString
- System.DateTime.ToShortDateString
- System.DateTime.ToShortTimeString
- System.DateTime.ToString

See Also

Reference

Time Management
ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s
_ftime, _ftime32, _ftime64
_gmtime_s, _gmtime32_s, _gmtime64_s
localtime_s, _localtime32_s, _localtime64_s
time, _time32, _time64
_tzset

asin, asinf

Calculate the arcsine.

```
double asin(
   double x
);
float asin(
   float x
);  // C++ only
long double asin(
   long double x
);  // C++ only
float asinf (
   float x
);
```

Parameters

x

Value whose arcsine is to be calculated.

Return Value

The **asin** function returns the arcsine of *x* in the range –π/2 to π/2 radians.

If *x* is less than –1 or greater than 1, **asin** returns an indefinite by default.

Input	SEH Exception	Matherr Exception
± ∞	**INVALID**	**_DOMAIN**
± **QNAN**,**IND**	none	**_DOMAIN**
|x|>1	**INVALID**	**_DOMAIN**

Remarks

C++ allows overloading, so you can call overloads of **asin**. In a C program, **asin** always takes and returns a double.

Requirements

Routine	Required header	Compatibility
asin, **asinf**	<math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

For more information, see acos, acosf.

.NET Framework Equivalent

System::Math::Asin

See Also

Reference

Floating-Point Support
acos, acosf
atan, atanf, atan2, atan2f
cos, cosf, cosh, coshf
_matherr
sin, sinf, sinh, sinhf
tan, tanf, tanh, tanhf

assert (CRT)

Evaluates an expression and, when the result is **false**, prints a diagnostic message and aborts the program.

```
void assert(
   int expression
);
```

Parameters

expression

Expression (including pointers) that evaluates to nonzero or 0.

Remarks

The **assert** macro is typically used to identify logic errors during program development by implementing the *expression* argument to evaluate to **false** only when the program is operating incorrectly. After debugging is complete, assertion checking can be turned off without modifying the source file by defining the identifier **NDEBUG**. **NDEBUG** can be defined with a **/D** command-line option or with a **#define** directive. If **NDEBUG** is defined with **#define**, the directive must appear before ASSERT.H is included.

assert prints a diagnostic message when *expression* evaluates to **false** (0) and calls abort to terminate program execution. No action is taken if *expression* is **true** (nonzero). The diagnostic message includes the failed expression, the name of the source file and line number where the assertion failed.

In Visual C++ 2005, the diagnostic message is printed in wide characters. Thus, it will work as expected even if there are Unicode characters in the expression.

The destination of the diagnostic message depends on the type of application that called the routine. Console applications always receive the message through **stderr**. In a Windows-based application, **assert** calls the Windows MessageBox function to create a message box to display the message along with an **OK** button. When the user clicks **OK**, the program aborts immediately.

When the application is linked with a debug version of the run-time libraries, **assert** creates a message box with three buttons: **Abort**, **Retry**, and **Ignore**. If the user clicks **Abort**, the program aborts immediately. If the user clicks **Retry**, the debugger is called and the user can debug the program if just-in-time (JIT) debugging is enabled. If the user clicks **Ignore**, **assert** continues with its normal execution: creating the message box with the **OK** button. Note that clicking **Ignore** when an error condition exists can result in undefined behavior.

For more information about CRT debugging, see CRT Debugging Techniques.

The **assert** routine is available in both the release and debug versions of the C run-time libraries. Two other assertion macros, _ASSERT and _ASSERTE, are also available, but they only evaluate the expressions passed to them when the _DEBUG flag has been defined.

Requirements

Routine	Required header	Compatibility
assert	<assert.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

In this program, the **analyze_string** function uses the **assert** function to test several conditions related to string and length. If any of the conditions fails, the program prints a message indicating what caused the failure.

```
// crt_assert.c
// compile with: /c
#include <stdio.h>
#include <assert.h>
#include <string.h>

void analyze_string( char *string );   // Prototype
```

```
int main( void )
{
   char  test1[] = "abc", *test2 = NULL, test3[] = "";

   printf ( "Analyzing string '%s'\n", test1 ); fflush( stdout );
   analyze_string( test1 );
   printf ( "Analyzing string '%s'\n", test2 ); fflush( stdout );
   analyze_string( test2 );
   printf ( "Analyzing string '%s'\n", test3 ); fflush( stdout );
   analyze_string( test3 );
}

// Tests a string to see if it is NULL,
// empty, or longer than 0 characters.
void analyze_string( char * string )
{
   assert( string != NULL );        // Cannot be NULL
   assert( *string != '\0' );       // Cannot be empty
   assert( strlen( string ) > 2 );  // Length must exceed 2
}
```

Sample Output

```
Analyzing string 'abc'
Analyzing string '(null)'
Assertion failed: string != NULL, file crt_assert.c, line 24

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

.NET Framework Equivalent

System::Diagnostics::Debug::Assert

See Also

Reference

Error Handling (CRT)
Process and Environment Control
abort
raise
signal
_ASSERT, _ASSERTE Macros
_DEBUG

_ASSERT, _ASSERTE Macros

Evaluate an expression and generate a debug report when the result is **False** (debug version only).

```
_ASSERT(
      booleanExpression
);
_ASSERTE(
      booleanExpression
);
```

Parameters

booleanExpression

Expression (including pointers) that evaluates to nonzero or 0.

Remarks

The **_ASSERT** and **_ASSERTE** macros provide an application with a clean and simple mechanism for checking assumptions during the debugging process. They are very flexible because they do not need to be enclosed in **#ifdef** statements to prevent them from being called in a retail build of an application. This flexibility is achieved by using the _DEBUG macro. **_ASSERT** and **_ASSERTE** are only available when **_DEBUG** is defined. When **_DEBUG** is not defined, calls to these macros are removed during preprocessing.

_ASSERT and **_ASSERTE** evaluate their *booleanExpression* argument and when the result is **false** (0), they print a diagnostic message and call _CrtDbgReportW to generate a debug report. The **_ASSERT** macro prints a simple diagnostic message, while **_ASSERTE** includes a string representation of the failed expression in the message. These macros do nothing when *booleanExpression* evaluates to nonzero.

In Visual C++ 2005, **_ASSERT** and **_ASSERTE** invoke **_CrtDbgReportW**, which causes all output to be in wide characters, and **_ASSERTE** properly prints Unicode characters in *booleanExpression*.

Because the **_ASSERTE** macro specifies the failed expression in the generated report, it enables users to identify the problem without referring to the application source code. However, a disadvantage exists in that every expression evaluated by **_ASSERTE** is included in the output (debug version) file of your application as a string constant. Therefore, if a large number of calls are made to **_ASSERTE**, these expressions can greatly increase the size of your output file.

Unless you specify otherwise with the _CrtSetReportMode and _CrtSetReportFile functions, messages appear in a pop-up dialog box equivalent to setting:

```
_CrtSetReportMode(CRT_ASSERT, _CRTDBG_MODE_WNDW);
```

_CrtDbgReport or **_CrtDbgReportW** generates the debug report and determines its destination or destinations, based on the current report mode or modes and file defined for the **_CRT_ASSERT** report type. By default, assertion failures and errors are directed to a debug message window. The _CrtSetReportMode and _CrtSetReportFile functions are used to define the destinations for each report type.

When the destination is a debug message window and the user clicks the **Retry** button, **_CrtDbgReport** or **_CrtDbgReportW** returns 1, causing the **_ASSERT** and **_ASSERTE** macros to start the debugger provided that just-in-time (JIT) debugging is enabled.

For more information about the reporting process, see the _CrtDbgReport, _CrtDbgReportW function. For more information about resolving assertion failures and using these macros as a debugging error handling mechanism, see Using Macros for Verification and Reporting.

The _RPT, _RPTF debug macros are also available for generating a debug report, but they do not evaluate an expression. The **_RPT** macros generate a simple report. The **_RPTF** macros include the source file and line number where the report macro was called in the generated report. In addition to the **_ASSERTE** macros, the ANSI assert routine can be used to verify program logic. This routine is available in both the debug and release versions of the libraries. In Visual C++ 2005, wide character versions of these macros are available (**_RPTWn**, **_RPTFWn**). The wide character versions are identical to the narrow character versions except that wide character strings are used for all string parameters and output.

Although **_ASSERT** and **_ASSERTE** are macros and are obtained by including Crtdbg.h, the application must link with one of

the following libraries because these macros call other run-time functions.

Libcmtd.lib

Multithread static library, debug version.

Msvcrtd.lib

Import library for Msvcr80d.dll, debug version.

Requirements

Macro	Required header	Compatibility
_ASSERT	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ASSERTE	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

In this program, calls are made to the **_ASSERT** and **_ASSERTE** macros to test the condition `string1 == string2`. If the condition fails, these macros print a diagnostic message. The **_RPTn** and **_RPTFn** group of macros is also exercised in this program, as an alternative to the **printf** function.

```
// crt_ASSERT_macro.c
// compile with: /D_DEBUG /MTd /Od /Zi /link /verbose:lib /debug
//
// This program uses the _ASSERT and _ASSERTE debugging macros.
//

#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <crtdbg.h>

int main()
{
   char *p1, *p2;

   // The Reporting Mode and File must be specified
   // before generating a debug report via an assert
   // or report macro.
   // This program sends all report types to STDOUT.
   _CrtSetReportMode(_CRT_WARN, _CRTDBG_MODE_FILE);
   _CrtSetReportFile(_CRT_WARN, _CRTDBG_FILE_STDOUT);
   _CrtSetReportMode(_CRT_ERROR, _CRTDBG_MODE_FILE);
   _CrtSetReportFile(_CRT_ERROR, _CRTDBG_FILE_STDOUT);
   _CrtSetReportMode(_CRT_ASSERT, _CRTDBG_MODE_FILE);
   _CrtSetReportFile(_CRT_ASSERT, _CRTDBG_FILE_STDOUT);

   // Allocate and assign the pointer variables.
   p1 = (char *)malloc(10);
   strcpy_s(p1, 10, "I am p1");
   p2 = (char *)malloc(10);
   strcpy_s(p2, 10, "I am p2");

   // Use the report macros as a debugging
   // warning mechanism, similar to printf.
   // Use the assert macros to check if the
   // p1 and p2 variables are equivalent.
   // If the expression fails, _ASSERTE will
   // include a string representation of the
   // failed expression in the report.
   // _ASSERT does not include the
   // expression in the generated report.
   _RPT0(_CRT_WARN,
```

```
        "Use the assert macros to evaluate the expression p1 == p2.\n");
    _RPTF2(_CRT_WARN, "\n Will _ASSERT find '%s' == '%s' ?\n", p1, p2);
    _ASSERT(p1 == p2);

    _RPTF2(_CRT_WARN, "\n\n Will _ASSERTE find '%s' == '%s' ?\n",
           p1, p2);
    _ASSERTE(p1 == p2);

    _RPT2(_CRT_ERROR, "'%s' != '%s'\n", p1, p2);

    free(p2);
    free(p1);

    return 0;
  }
```

Output

```
Use the assert macros to evaluate the expression p1 == p2.
crt_ASSERT_macro.c(54) :
 Will _ASSERT find 'I am p1' == 'I am p2' ?
crt_ASSERT_macro.c(55) : Assertion failed!
crt_ASSERT_macro.c(58) :

 Will _ASSERTE find 'I am p1' == 'I am p2' ?
crt_ASSERT_macro.c(59) : Assertion failed: p1 == p2
'I am p1' != 'I am p2'
```

.NET Framework Equivalent

System::Diagnostics::Debug::Assert

See Also

Reference

Debug Routines

_RPT, _RPTF, _RPTW, _RPTFW Macros

atan, atanf, atan2, atan2f

Calculates the arctangent of *x* (**atan** or **atanf**) or the arctangent of *y/x* (**atan2** or **atan2f**).

```
double atan(
   double x
);
float atan(
   float x
);  // C++ only
long double atan(
   long double x
);  // C++ only
double atan2(
   double y,
   double x
);
float atan2(
   float y,
   float x
);  // C++ only
long double atan2(
   long double y,
   long double x
);  // C++ only
float atanf(
   float x
);
float atan2f(
   float y,
   float x
);
```

Parameters

x, *y*

Any numbers.

Return Value

atan returns the arctangent of *x* in the range of –π/2 to π/2 radians. **atan2** returns the arctangent of *y/x* in the range –π to π radians. If *x* is 0, **atan** returns 0. If both parameters of **atan2** are 0, the function returns 0. All results are in radians.

atan2 uses the signs of both parameters to determine the quadrant of the return value.

Input	SEH Exception	Matherr Exception
± **QNAN**,**IND**	none	**_DOMAIN**

Remarks

The **atan** function calculates the arctangent of *x*. **atan2** calculates the arctangent of *y/x* (if *x* equals 0, **atan2** returns π/2 if *y* is positive, -π/2 if *y* is negative, or 0 if *y* is 0.)

atan has an implementation that uses Streaming SIMD Extensions 2 (SSE2). See _set_SSE2_enable for information and restrictions on using the SSE2 implementation.

C++ allows overloading, so you can call overloads of **atan** and **atan2**. In a C program, **atan** and **atan2** always take and return doubles.

Requirements

Routine	Required header	Compatibility

atan, **atan2**, **atanf**, **atan2f**	<math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

```
// crt_atan.c
// arguments: 5 0.5
#include <math.h>
#include <stdio.h>
#include <errno.h>

int main( int ac, char* av[] )
{
   double x, y, theta;
   if( ac != 3 ){
      fprintf( stderr, "Usage: %s <x> <y>\n", av[0] );
      return 1;
   }
   x = atof( av[1] );
   theta = atan( x );
   printf( "Arctangent of %f: %f\n", x, theta );
   y = atof( av[2] );
   theta = atan2( y, x );
   printf( "Arctangent of %f / %f: %f\n", y, x, theta );
   return 0;
}
```

Output

```
Arctangent of 5.000000: 1.373401
Arctangent of 0.500000 / 5.000000: 0.099669
```

.NET Framework Equivalent

- System::Math::Atan
- System::Math::Atan2

See Also

Reference

Floating-Point Support
acos, acosf
asin, asinf
cos, cosf, cosh, coshf
_matherr
sin, sinf, sinh, sinhf
tan, tanf, tanh, tanhf

atexit

Processes the specified function at exit.

```
int atexit(
   void (__cdecl *func )( void )
);
```

Parameters

func

Function to be called.

Return Value

atexit returns 0 if successful, or a nonzero value if an error occurs.

Remarks

The **atexit** function is passed the address of a function (*func*) to be called when the program terminates normally. Successive calls to **atexit** create a register of functions that are executed in last-in, first-out (LIFO) order. The functions passed to **atexit** cannot take parameters. **atexit** and **_onexit** use the heap to hold the register of functions. Thus, the number of functions that can be registered is limited only by heap memory.

The code in the **atexit** function should not contain any dependency on any DLL which could have already been unloaded when the **atexit** function is called.

To generate an ANSI-compliant application, use the ANSI-standard **atexit** function (rather than the similar **_onexit** function).

Requirements

Routine	Required header	Compatibility
atexit	<stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

This program pushes four functions onto the stack of functions to be executed when **atexit** is called. When the program exits, these programs are executed on a last in, first out basis.

```
// crt_atexit.c
#include <stdlib.h>
#include <stdio.h>

void fn1( void ), fn2( void ), fn3( void ), fn4( void );

int main( void )
{
   atexit( fn1 );
   atexit( fn2 );
   atexit( fn3 );
   atexit( fn4 );
   printf( "This is executed first.\n" );
}

void fn1()
{
   printf( "next.\n" );
}

void fn2()
{
   printf( "executed " );
}
```

```
void fn3()
{
   printf( "is " );
}

void fn4()
{
   printf( "This " );
}
```

Output

```
This is executed first.
This is executed next.
```

.NET Framework Equivalent

System::Diagnostics::Process::Exited

See Also

Reference

Process and Environment Control
abort
exit, _exit
_onexit, _onexit_m

_atodbl, _atodbl_l, _atoldbl, _atoldbl_l, _atoflt _atoflt_l

Converts a string to a double (**_atodbl**), long double (**_atoldbl**), or float (**_atoflt**).

```
int _atodbl(
   _CRT_DOUBLE * value,
   char * str
);
int _atodbl_l (
   _CRT_DOUBLE * value,
   char * str,
   locale_t locale
);
int _atoldbl(
   _LDOUBLE * value,
   char * str
);
int _atoldbl_l (
   _LDOUBLE * value,
   char * str,
   locale_t locale
);
int _atoflt(
   _CRT_FLOAT * value,
   char * str
);
int _atoflt_l(
   _CRT_FLOAT * value,
   char * str,
   locale_t locale
);
```

Parameters

value

The double, long double, or float value produced by converting the string to a floating-point value. These values are wrapped in a structure.

str

The string to be parsed to convert into a floating-point value.

locale

The locale to use.

Return Value

Returns 0 if successful; possible error codes are **_UNDERFLOW** or **_OVERFLOW**, defined in the header file Math.h.

Remarks

These functions convert a string to a floating-point value. The difference between these functions and the **atof** family of functions is that these functions do not generate floating-point code and hence do not result in hardware exceptions. Instead, error conditions are reported as an error code.

If a string does not have a valid interpretation as a floating-point value, *value* is set to zero and the return value is zero.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Requirements

Routines	Required header	Compatibility

_atodbl, **_atoldbl**, **_atoflt** **_atodbl_l**, **_atoldbl_l**, **_atoflt_l**	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

```
// crt_atodbl.c
// Uses _atodbl to convert a string to a double precision
// floating point value.

#include <stdlib.h>
#include <stdio.h>

int main()
{
   char str1[256] = "3.141592654";
   char abc[256] = "abc";
   char oflow[256] = "1.0E+5000";
   _CRT_DOUBLE dblval;
   _CRT_FLOAT fltval;
   int retval;

   retval = _atodbl(&dblval, str1);

   printf("Double value: %lf\n", dblval.x);
   printf("Return value: %d\n\n", retval);

   retval = _atoflt(&fltval, str1);
   printf("Float value: %f\n", fltval.f);
   printf("Return value: %d\n\n", retval);

   // A non-floating point value: returns 0.
   retval = _atoflt(&fltval, abc);
   printf("Float value: %f\n", fltval.f);
   printf("Return value: %d\n\n", retval);

   // Overflow.
   retval = _atoflt(&fltval, oflow);
   printf("Float value: %f\n", fltval.f);
   printf("Return value: %d\n\n", retval);

   return 0;
}
```

Output

```
Double value: 3.141593
Return value: 0

Float value: 3.141593
Return value: 0

Float value: 0.000000
Return value: 0

Float value: 1.#INF00
Return value: 3
```

See Also

Reference

Data Conversion

Floating-Point Support

Locale

atof, _atof_l, _wtof, _wtof_l

atof, _atof_l, _wtof, _wtof_l

Convert a string to double.

```
double atof(
   const char *str
);
double _wtof(
   const wchar_t *str
);
```

Parameters

str

String to be converted.

locale

Locale to use.

Return Value

Each function returns the **double** value produced by interpreting the input characters as a number. The return value is 0.0 if the input cannot be converted to a value of that type.

In Visual C++ 2005, in all out-of-range cases, errno is set to **ERANGE**. If the parameter passed in is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return 0.

Remarks

These functions convert a character string to a double-precision, floating-point value.

The input string is a sequence of characters that can be interpreted as a numerical value of the specified type. The function stops reading the input string at the first character that it cannot recognize as part of a number. This character may be the null character ('\0' or L'\0') terminating the string.

The *str* argument to **atof** and **_wtof** has the following form:

[*whitespace*] [*sign*] [*digits*] [.*digits*] [{**d** | **D** | **e** | **E** }[*sign*]*digits*]

A *whitespace* consists of space or tab characters, which are ignored; *sign* is either plus (+) or minus (–); and *digits* are one or more decimal digits. If no digits appear before the decimal point, at least one must appear after the decimal point. The decimal digits may be followed by an exponent, which consists of an introductory letter (**d**, **D**, **e**, or **E**) and an optionally signed decimal integer.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tstof	**atof**	**atof**	**_wtof**
_ttof	**atof**	**atof**	**_wtof**

Requirements

Routine(s)	Required header	Compatibility
atof	<math.h> and <stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_atof_l	<math.h> and <stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wtof, _wtof_l	<stdlib.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

This program shows how numbers stored as strings can be converted to numeric values using the **atof** function.

```
// crt_atof.c
//
// This program shows how numbers stored as
// strings can be converted to numeric
// values using the atof function.

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    char    *str = NULL;
    double  value = 0;

    // An example of the atof function
    // using leading and training spaces.
    str = "  3336402735171707160320 ";
    value = atof( str );
    printf( "Function: atof( \"%s\" ) = %e\n", str, value );

    // Another example of the atof function
    // using the 'd' exponential formatting keyword.
    str = "3.1412764583d210";
    value = atof( str );
    printf( "Function: atof( \"%s\" ) = %e\n", str, value );

    // An example of the atof function
    // using the 'e' exponential formatting keyword.
    str = "  -2309.12E-15";
    value = atof( str );
    printf( "Function: atof( \"%s\" ) = %e\n", str, value );

}
```

Output

```
Function: atof( "  3336402735171707160320 " ) = 3.336403e+021
Function: atof( "3.1412764583d210" ) = 3.141276e+210
Function: atof( "  -2309.12E-15" ) = -2.309120e-012
```

.NET Framework Equivalent

- System::Convert::ToSingle
- System::Convert::ToDouble

See Also

Reference

Data Conversion
Floating-Point Support
Locale
_ecvt
_fcvt
_gcvt

setlocale, _wsetlocale
_atodbl, _atodbl_l, _atoldbl, _atoldbl_l, _atoflt _atoflt_l

atoi, _atoi_l, _wtoi, _wtoi_l

Convert a string to integer.

```
int atoi(
   const char *str
);
int _wtoi(
   const wchar_t *str
);
int _atoi_l(
   const char *str,
   _locale_t locale
);
int _wtoi_l(
   const wchar_t *str,
   _locale_t locale
);
```

Parameters

str

String to be converted.

locale

Locale to use.

Return Value

Each function returns the **int** value produced by interpreting the input characters as a number. The return value is 0 for **atoi** and **_wtoi**, if the input cannot be converted to a value of that type.

In Visual C++ 2005, in the case of overflow with large negative integral values, **LONG_MIN** is returned. **atoi** and **_wtoi** return **INT_MAX** and **INT_MIN** on these conditions. In all out-of-range cases, **errno** is set to **ERANGE**. If the parameter passed in is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return 0.

Remarks

These functions convert a character string to an integer value (**atoi** and **_wtoi**). The input string is a sequence of characters that can be interpreted as a numerical value of the specified type. The function stops reading the input string at the first character that it cannot recognize as part of a number. This character may be the null character ('\0' or L'\0') terminating the string.

The *str* argument to **atoi** and **_wtoi** has the following form:

[*whitespace*] [*sign*] [*digits*]]

A *whitespace* consists of space or tab characters, which are ignored; *sign* is either plus (+) or minus (–); and *digits* are one or more digits.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tstoi	**atoi**	**atoi**	**_wtoi**
_ttoi	**atoi**	**atoi**	**_wtoi**

Requirements

Routines	Required header	Compatibility

atoi	<stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_atoi_l, **_wtoi**, **_wtoi_l**	<stdlib.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

This program shows how numbers stored as strings can be converted to numeric values using the **atoi** functions.

```
// crt_atoi.c
// This program shows how numbers
// stored as strings can be converted to
// numeric values using the atoi functions.

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int main( void )
{
    char    *str = NULL;
    int     value = 0;

    // An example of the atoi function.
    str = "  -2309 ";
    value = atoi( str );
    printf( "Function: atoi( \"%s\" ) = %d\n", str, value );

    // Another example of the atoi function.
    str = "31412764";
    value = atoi( str );
    printf( "Function: atoi( \"%s\" ) = %d\n", str, value );

    // Another example of the atoi function
    // with an overflow condition occuring.
    str = "3336402735171707160320";
    value = atoi( str );
    printf( "Function: atoi( \"%s\" ) = %d\n", str, value );
    if (errno == ERANGE)
    {
       printf("Overflow condition occurred.\n");
    }
}
```

Output

```
Function: atoi( "  -2309 " ) = -2309
Function: atoi( "31412764" ) = 31412764
Function: atoi( "3336402735171707160320" ) = 2147483647
Overflow condition occurred.
```

.NET Framework Equivalent

- System::Convert::ToInt32
- System::Convert::ToUInt32

See Also

Reference

Data Conversion
Floating-Point Support
Locale
_ecvt
_fcvt

_gcvt
setlocale, _wsetlocale
_atodbl, _atodbl_l, _atoldbl, _atoldbl_l, _atoflt _atoflt_l

_atoi64, _atoi64_l, _wtoi64, _wtoi64_l

Converts a string to a 64-bit integer.

```
__int64 _atoi64(
   const char *str
);
__int64 _wtoi64(
   const wchar_t *str
);
__int64 _atoi64_l(
   const char *str,
   _locale_t locale
);
__int64 _wtoi64_l(
   const wchar_t *str,
   _locale_t locale
);
```

Parameters

str

String to be converted.

locale

Locale to use.

Return Value

Each function returns the **__int64** value produced by interpreting the input characters as a number. The return value is 0 for **_atoi64** if the input cannot be converted to a value of that type.

In Visual C++ 2005, in the case of overflow with large positive integral values, **_atoi64** returns **I64_MAX** and **I64_MIN** in the case of overflow with large negative integral values.

In all out-of-range cases, **errno** is set to **ERANGE**. If the parameter passed in is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return 0.

Remarks

These functions convert a character string to a 64-bit integer value.

The input string is a sequence of characters that can be interpreted as a numerical value of the specified type. The function stops reading the input string at the first character that it cannot recognize as part of a number. This character might be the null character ('\0' or L'\0') terminating the string.

The *str* argument to **_atoi64** has the following form:

```
[whitespace] [sign] [digits]]
```

A *whitespace* consists of space or tab characters, which are ignored; *sign* is either plus (+) or minus (–); and *digits* are one or more digits.

_wtoi64 is identical to **_atoi64** except that it takes a wide character string as a parameter.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tstoi64	_atoi64	_atoi64	_wtoi64

_ttoi64	_atoi64	_atoi64	_wtoi64

Requirements

Routines	Required header	Compatibility
_atoi64, **_atoi64_l**	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wtoi64, **_wtoi64_l**	<stdlib.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

This program shows how numbers stored as strings can be converted to numeric values using the **_atoi64** functions.

```
// crt_atoi64.c
// This program shows how numbers stored as
// strings can be converted to numeric values
// using the _atoi64 functions.
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int main( void )
{
    char    *str = NULL;
    __int64 value = 0;

    // An example of the _atoi64 function
    // with leading and trailing white spaces.
    str = "  -2309 ";
    value = _atoi64( str );
    printf( "Function: _atoi64( \"%s\" ) = %d\n", str, value );

    // Another example of the _atoi64 function
    // with an arbitrary decimal point.
    str = "314127.64";
    value = _atoi64( str );
    printf( "Function: _atoi64( \"%s\" ) = %d\n", str, value );

    // Another example of the _atoi64 function
    // with an overflow condition occurring.
    str = "3336402735171707160320";
    value = _atoi64( str );
    printf( "Function: _atoi64( \"%s\" ) = %d\n", str, value );
    if (errno == ERANGE)
    {
       printf("Overflow condition occurred.\n");
    }
}
```

Output

```
Function: _atoi64( "  -2309 " ) = -2309
Function: _atoi64( "314127.64" ) = 314127
Function: _atoi64( "3336402735171707160320" ) = -1
Overflow condition occurred.
```

.NET Framework Equivalent

- System::Convert::ToInt64
- System::Convert::ToUInt64

See Also

Reference

Data Conversion
Floating-Point Support
Locale
_ecvt
_fcvt
_gcvt
setlocale, _wsetlocale
_atodbl, _atodbl_l, _atoldbl, _atoldbl_l, _atoflt _atoflt_l

atol, _atol_l, _wtol, _wtol_l

Convert a string to a long integer.

```
long atol(
   const char *str
);
long _atol_l(
   const char *str,
   _locale_t locale
);
long _wtol(
   const wchar_t *str
);
long _wtol_l(
   const wchar_t *str,
   _locale_t locale
);
```

Parameters

str

String to be converted.

locale

Locale to use.

Return Value

Each function returns the **long** value produced by interpreting the input characters as a number. The return value is 0L for **atol** if the input cannot be converted to a value of that type.

In Visual C++ 2005, in the case of overflow with large positive integral values, **atol** returns **LONG_MAX**; in the case of overflow with large negative integral values, **LONG_MIN** is returned. In all out-of-range cases, **errno** is set to **ERANGE**. If the parameter passed in is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return 0.

Remarks

These functions convert a character string to a long integer value (**atol**).

The input string is a sequence of characters that can be interpreted as a numerical value of the specified type. The function stops reading the input string at the first character that it cannot recognize as part of a number. This character may be the **NULL** character ('\0' or L'\0') terminating the string.

The *str* argument to **atol** has the following form:

[*whitespace*] [*sign*] [*digits*]]

A *whitespace* consists of space or tab characters, which are ignored; *sign* is either plus (+) or minus (–); and *digits* are one or more digits.

_wtol is identical to **atol** except that it takes a wide character string.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tstol	**atol**	**atol**	**_wtol**
_ttol	**atol**	**atol**	**_wtol**

Requirements

Routines	Required header	Compatibility
atol	<stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_atol_l, **_wtol**, **_wtol_l**	<stdlib.h> and <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

This program shows how numbers stored as strings can be converted to numeric values using the **atol** function.

```
// crt_atol.c
// This program shows how numbers stored as
// strings can be converted to numeric values
// using the atol functions.
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int main( void )
{
    char    *str = NULL;
    long    value = 0;

    // An example of the atol function
    // with leading and trailing white spaces.
    str = "  -2309 ";
    value = atol( str );
    printf( "Function: atol( \"%s\" ) = %d\n", str, value );

    // Another example of the atol function
    // with an arbitrary decimal point.
    str = "314127.64";
    value = atol( str );
    printf( "Function: atol( \"%s\" ) = %d\n", str, value );

    // Another example of the atol function
    // with an overflow condition occurring.
    str = "3336402735171707160320";
    value = atol( str );
    printf( "Function: atol( \"%s\" ) = %d\n", str, value );
    if (errno == ERANGE)
    {
       printf("Overflow condition occurred.\n");
    }
}
```

Output

```
Function: atol( "  -2309 " ) = -2309
Function: atol( "314127.64" ) = 314127
Function: atol( "3336402735171707160320" ) = 2147483647
Overflow condition occurred.
```

.NET Framework Equivalent

- System::Convert::ToInt32
- System::Convert::ToUInt32

See Also

Reference

Data Conversion

Floating-Point Support
Locale
_ecvt
_fcvt
_gcvt
setlocale, _wsetlocale
_atodbl, _atodbl_l, _atoldbl, _atoldbl_l, _atoflt _atoflt_l

_beginthread, _beginthreadex

Creates a thread.

```
uintptr_t _beginthread(
   void( *start_address )( void * ),
   unsigned stack_size,
   void *arglist
);
uintptr_t _beginthreadex(
   void *security,
   unsigned stack_size,
   unsigned ( *start_address )( void * ),
   void *arglist,
   unsigned initflag,
   unsigned *thrdaddr
);
```

Parameters

start_address

Start address of a routine that begins execution of a new thread. For **_beginthread**, the calling convention is either __cdecl or __clrcall; for **_beginthreadex**, it is either __stdcall or __clrcall.

stack_size

Stack size for a new thread or 0.

arglist

Argument list to be passed to a new thread or NULL.

security

Pointer to a SECURITY_ATTRIBUTES structure that determines whether the returned handle can be inherited by child processes. If NULL, the handle cannot be inherited. Must be NULL for Windows 95 applications.

initflag

Initial state of a new thread (**0** for running or **CREATE_SUSPENDED** for suspended); use ResumeThread to execute the thread.

thrdaddr

Points to a 32-bit variable that receives the thread identifier. Might be **NULL**, in which case it is not used.

Return Value

If successful, each of these functions returns a handle to the newly created thread; however, if the newly created thread exits too quickly, **_beginthread** might not return a valid handle (see the discussion in the **Remarks** section). **_beginthread** returns -1L on an error, in which case **errno** is set to **EAGAIN** if there are too many threads, to **EINVAL** if the argument is invalid or the stack size is incorrect, or to **EACCES** in the case of insufficient resources (such as memory). **_beginthreadex** returns 0 on an error, in which case **errno** and **_doserrno** are set.

If *startaddress* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

For more information about **uintptr_t**, see Standard Types.

Remarks

The **_beginthread** function creates a thread that begins execution of a routine at *start_address*. The routine at *start_address* must use the **__cdecl** calling convention and should have no return value. When the thread returns from that routine, it is terminated automatically. For more information about threads, see Multithreading.

_beginthreadex resembles the Win32 CreateThread API more closely than **_beginthread** does. **_beginthreadex** differs from

_beginthread in the following ways:

- **_beginthreadex** has three additional parameters: *initflag*, *security*, and *threadaddr*. The new thread can be created in a suspended state, with a specified security (Windows NT only), and can be accessed using *thrdaddr*, which is the thread identifier.
- The routine at *start_address* passed to **_beginthreadex** must use the **__stdcall** calling convention and must return a thread exit code.
- **_beginthreadex** returns 0 on failure, rather than -1L.
- A thread created with **_beginthreadex** is terminated by a call to **_endthreadex**.

The **_beginthreadex** function gives you more control over how the thread is created than **_beginthread** does. The **_endthreadex** function is also more flexible. For example, with **_beginthreadex**, you can use security information, set the initial state of the thread (running or suspended), and get the thread identifier of the newly created thread. You are also able to use the thread handle returned by **_beginthreadex** with the synchronization APIs, which you cannot do with **_beginthread**.

It is safer to use **_beginthreadex** than **_beginthread**. If the thread generated by **_beginthread** exits quickly, the handle returned to the caller of **_beginthread** might be invalid or, worse, point to another thread. However, the handle returned by **_beginthreadex** has to be closed by the caller of **_beginthreadex**, so it is guaranteed to be a valid handle if **_beginthreadex** did not return an error.

You can call _endthread or _endthreadex explicitly to terminate a thread; however, **_endthread** or **_endthreadex** is called automatically when the thread returns from the routine passed as a parameter. Terminating a thread with a call to **endthread** or **_endthreadex** helps to ensure proper recovery of resources allocated for the thread.

_endthread automatically closes the thread handle (whereas **_endthreadex** does not). Therefore, when using **_beginthread** and **_endthread**, do not explicitly close the thread handle by calling the Win32 CloseHandle API. This behavior differs from the Win32 ExitThread API.

Note
For an executable file linked with Libcmt.lib, do not call the Win32 **ExitThread** API; this prevents the run-time system from reclaiming allocated resources. **_endthread** and **_endthreadex** reclaim allocated thread resources and then call **ExitThread**.

The operating system handles the allocation of the stack when either **_beginthread** or **_beginthreadex** is called; you do not need to pass the address of the thread stack to either of these functions. In addition, the *stack_size* argument can be 0, in which case the operating system uses the same value as the stack specified for the main thread.

arglist is a parameter to be passed to the newly created thread. Typically it is the address of a data item, such as a character string. *arglist* can be **NULL** if it is not needed, but **_beginthread** and **_beginthreadex** must be provided with some value to pass to the new thread. All threads are terminated if any thread calls **abort**, **exit**, **_exit**, or **ExitProcess**.

The locale of the new thread is inherited from its parent thread. If per thread locale is enabled by a call to _configthreadlocale (either globally or for new threads only), the thread can change its locale independently from its parent by calling **setlocale** or **_wsetlocale**. For more information, see Locale.

For mixed and pure code, **_beginthread** and **_beginthreadex** both have two overloads, one taking a native calling-convention function pointer, the other taking a **__clrcall** function pointer. The first overload is not application domain-safe and never will be. If you are writing mixed or pure code you must ensure that the new thread enters the correct application domain before it accesses managed resources. You can do this, for example, by using call_in_appdomain Function. The second overload is application domain-safe; the newly created thread will always end up in the application domain of the caller of **_beginthread** or **_beginthreadex**.

Requirements

Routine	**Required header**	**Compatibility**
_beginthread	<process.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_beginthreadex	<process.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Multithreaded versions of the C run-time libraries only.

To use **_beginthread** or **_beginthreadex**, the application must link with one of the multithreaded C run-time libraries.

Example

The following example uses **_beginthread** and **_endthread**.

```
// crt_BEGTHRD.C
// compile with: /MT /D "_X86_" /c
// processor: x86
#include <windows.h>
#include <process.h>    /* _beginthread, _endthread */
#include <stddef.h>
#include <stdlib.h>
#include <conio.h>

void Bounce( void *ch );
void CheckKey( void *dummy );

/* GetRandom returns a random integer between min and max. */
#define GetRandom( min, max ) ((rand() % (int)(((max) + 1) - (min))) + (min))

BOOL repeat = TRUE;     /* Global repeat flag and video variable */
HANDLE hStdOut;         /* Handle for console window */
CONSOLE_SCREEN_BUFFER_INFO csbi;    /* Console information structure */

int main()
{
    CHAR    ch = 'A';

    hStdOut = GetStdHandle( STD_OUTPUT_HANDLE );

    /* Get display screen's text row and column information. */
   GetConsoleScreenBufferInfo( hStdOut, &csbi );

    /* Launch CheckKey thread to check for terminating keystroke. */
    _beginthread( CheckKey, 0, NULL );

    /* Loop until CheckKey terminates program. */
    while( repeat )
    {
        /* On first loops, launch character threads. */
        _beginthread( Bounce, 0, (void *) (ch++)  );

        /* Wait one second between loops. */
        Sleep( 1000L );
    }
}

/* CheckKey - Thread to wait for a keystroke, then clear repeat flag. */
void CheckKey( void *dummy )
{
    _getch();
    repeat = 0;    /* _endthread implied */

}

/* Bounce - Thread to create and and control a colored letter that moves
```

```
 * around on the screen.
 *
 * Params: ch - the letter to be moved
 */
void Bounce( void *ch )
{
    /* Generate letter and color attribute from thread argument. */
    char    blankcell = 0x20;
    char    blockcell = (char) ch;
    BOOL    first = TRUE;
   COORD   oldcoord, newcoord;
   DWORD   result;


    /* Seed random number generator and get initial location. */
    srand( _threadid );
    newcoord.X = GetRandom( 0, csbi.dwSize.X - 1 );
    newcoord.Y = GetRandom( 0, csbi.dwSize.Y - 1 );
    while( repeat )
    {
        /* Pause between loops. */
        Sleep( 100L );

        /* Blank out our old position on the screen, and draw new letter. */
        if( first )
            first = FALSE;
        else
         WriteConsoleOutputCharacter( hStdOut, &blankcell, 1, oldcoord, &result );
         WriteConsoleOutputCharacter( hStdOut, &blockcell, 1, newcoord, &result );

        /* Increment the coordinate for next placement of the block. */
        oldcoord.X = newcoord.X;
        oldcoord.Y = newcoord.Y;
        newcoord.X += GetRandom( -1, 1 );
        newcoord.Y += GetRandom( -1, 1 );

        /* Correct placement (and beep) if about to go off the screen. */
        if( newcoord.X < 0 )
            newcoord.X = 1;
        else if( newcoord.X == csbi.dwSize.X )
            newcoord.X = csbi.dwSize.X - 2;
        else if( newcoord.Y < 0 )
            newcoord.Y = 1;
        else if( newcoord.Y == csbi.dwSize.Y )
            newcoord.Y = csbi.dwSize.Y - 2;

        /* If not at a screen border, continue, otherwise beep. */
        else
            continue;
        Beep( ((char) ch - 'A') * 100, 175 );
    }
    /* _endthread given to terminate */
    _endthread();
}
```

Input

```
press any key to end
```

The following sample code demonstrates how you can use the thread handle returned by **_beginthreadex** with the synchronization API WaitForSingleObject. The main thread waits for the second thread to terminate before it continues. When the second thread calls **_endthreadex**, it causes its thread object to go to the signaled state. This allows the primary thread to continue running. This cannot be done with **_beginthread** and **_endthread**, because **_endthread** calls **CloseHandle**, destroying the thread object before it can be set to the signaled state.

```
// crt_begthrdex.cpp
// compile with: /MT
```

```
#include <windows.h>
#include <stdio.h>
#include <process.h>

unsigned Counter;
unsigned __stdcall SecondThreadFunc( void* pArguments )
{
    printf( "In second thread...\n" );

    while ( Counter < 1000000 )
        Counter++;

    _endthreadex( 0 );
    return 0;
}

int main()
{
    HANDLE hThread;
    unsigned threadID;

    printf( "Creating second thread...\n" );

    // Create the second thread.
    hThread = (HANDLE)_beginthreadex( NULL, 0, &SecondThreadFunc, NULL, 0, &threadID );

    // Wait until second thread terminates. If you comment out the line
    // below, Counter will not be correct because the thread has not
    // terminated, and Counter most likely has not been incremented to
    // 1000000 yet.
    WaitForSingleObject( hThread, INFINITE );
    printf( "Counter should be 1000000; it is-> %d\n", Counter );
    // Destroy the thread object.
    CloseHandle( hThread );
}
```

Output

```
Creating second thread...
In second thread...
Counter should be 1000000; it is-> 1000000
```

.NET Framework Equivalent

System::Threading::Thread::Start

See Also

Reference

Process and Environment Control
_endthread, _endthreadex
abort
exit, _exit
GetExitCodeThread

Bessel Functions

The Bessel functions are commonly used in the mathematics of electromagnetic wave theory.

_j0, _j1, _jn

These routines return Bessel functions of the first kind: orders 0, 1, and *n*, respectively.

_y0, _y1, _yn

These routines return Bessel functions of the second kind: orders 0, 1, and *n*, respectively.

Example

```
// crt_bessel1.c
#include <math.h>
#include <stdio.h>

int main( void )
{
   double x = 2.387;
   int n = 3, c;

   printf( "Bessel functions for x = %f:\n", x );
   printf( " Kind   Order  Function     Result\n\n" );
   printf( " First  0      _j0( x )     %f\n", _j0( x ) );
   printf( " First  1      _j1( x )     %f\n", _j1( x ) );
   for( c = 2; c < 5; c++ )
      printf( " First  %d      _jn( %d, x )  %f\n", c, c, _jn( c, x ) );
   printf( " Second 0      _y0( x )     %f\n", _y0( x ) );
   printf( " Second 1      _y1( x )     %f\n", _y1( x ) );
   for( c = 2; c < 5; c++ )
      printf( " Second %d      _yn( %d, x )  %f\n", c, c, _yn( c, x ) );
}
```

Output

```
Bessel functions for x = 2.387000:
 Kind   Order  Function     Result

 First  0      _j0( x )     0.009288
 First  1      _j1( x )     0.522941
 First  2      _jn( 2, x )  0.428870
 First  3      _jn( 3, x )  0.195734
 First  4      _jn( 4, x )  0.063131
 Second 0      _y0( x )     0.511681
 Second 1      _y1( x )     0.094374
 Second 2      _yn( 2, x )  -0.432608
 Second 3      _yn( 3, x )  -0.819314
 Second 4      _yn( 4, x )  -1.626833
```

See Also

Reference

Floating-Point Support

_matherr

Bessel Functions: _y0, _y1, _yn

Compute the Bessel function.

```
double _y0(
   double x
);
double _y1(
   double x
);
double _yn(
   int n,
   double x
);
```

Parameters

x

Floating-point value.

n

Integer order of Bessel function.

Return Value

Each routine returns a Bessel function of *x*. If *x* is negative, the routine sets **errno** to **EDOM**, prints a **_DOMAIN** error message to **stderr**, and returns **_HUGE_VAL**. You can modify error handling by using **_matherr**.

Remarks

The **_y0**, **_y1**, and **_yn** routines return Bessel functions of the second kind: orders 0, 1, and n, respectively.

Input	SEH Exception	Matherr Exception
± **QNAN**,**IND**	**INVALID**	**_DOMAIN**
± 0	**ZERODIVIDE**	**_SING**
|x|<0.0	**INVALID**	**_DOMAIN**

Requirements

Routine	Required header	Compatibility
_y0	<math.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_y1	<math.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_yn	<math.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See Bessel Functions for an example.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support

Bessel Functions

_matherr

Bessel Functions: _j0, _j1, _jn

Compute the Bessel function.

```
double _j0(
   double x
);
double _j1(
   double x
);
double _jn(
   int n,
   double x
);
```

Parameters

x

Floating-point value.

n

Integer order of Bessel function.

Return Value

Each of these routines returns a Bessel function of *x*. You can modify error handling by using **_matherr**.

Remarks

The **_j0**, **_j1**, and **_jn** routines return Bessel functions of the first kind: orders 0, 1, and n, respectively.

Input	SEH Exception	Matherr Exception
± **QNAN**,**IND**	**INVALID**	**_DOMAIN**

Requirements

Routine	Required header	Compatibility
_j0	<math.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_j1	<math.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_jn	<math.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See Bessel Functions for an example.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support
Bessel Functions
_matherr

bsearch

Performs a binary search of a sorted array. A more secure version is available; see bsearch_s.

```
void *bsearch(
   const void *key,
   const void *base,
   size_t num,
   size_t width,
   int ( __cdecl *compare ) ( const void *, const void *)
);
```

Parameters

key

Object to search for.

base

Pointer to base of search data.

num

Number of elements.

width

Width of elements.

compare

Callback function that compares two elements. The first is a pointer to the key for the search and the second is a pointer to the array element to be compared with the key.

Return Value

bsearch returns a pointer to an occurrence of *key* in the array pointed to by *base*. If *key* is not found, the function returns **NULL**. If the array is not in ascending sort order or contains duplicate records with identical keys, the result is unpredictable.

Remarks

The **bsearch** function performs a binary search of a sorted array of *num* elements, each of *width* bytes in size. The *base* value is a pointer to the base of the array to be searched, and *key* is the value being sought. The *compare* parameter is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. **bsearch** calls the *compare* routine one or more times during the search, passing pointers to two array elements on each call. The *compare* routine compares the elements, then returns one of the following values:

Value returned by *compare* routine	**Description**
< 0	*elem1* less than *elem2*
0	*elem1* equal to *elem2*
> 0	*elem1* greater than *elem2*

This function validates its parameters. If *compare*, *key* or *num* is **NULL**, or if *base* is **NULL** and **num* is nonzero, or if *width* is zero, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **NULL**.

Requirements

Routine	**Required header**	**Compatibility**

bsearch	<stdlib.h> and <search.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

This program sorts a string array with qsort, and then uses bsearch to find the word "cat".

```
// crt_bsearch.c
#include <search.h>
#include <string.h>
#include <stdio.h>

int compare( char **arg1, char **arg2 )
{
   /* Compare all of both strings: */
   return _strcmpi( *arg1, *arg2 );
}

int main( void )
{
   char *arr[] = {"dog", "pig", "horse", "cat", "human", "rat", "cow", "goat"};
   char **result;
   char *key = "cat";
   int i;

   /* Sort using Quicksort algorithm: */
   qsort( (void *)arr, sizeof(arr)/sizeof(arr[0]), sizeof( char * ), (int (*)(const
   void*, const void*))compare );

   for( i = 0; i < sizeof(arr)/sizeof(arr[0]); ++i )    /* Output sorted list */
      printf( "%s ", arr[i] );

   /* Find the word "cat" using a binary search algorithm: */
   result = (char **)bsearch( (char *) &key, (char *)arr, sizeof(arr)/sizeof(arr[0]),
                              sizeof( char * ), (int (*)(const void*, const void*))compare
);
   if( result )
      printf( "\n%s found at %Fp\n", *result, result );
   else
      printf( "\nCat not found!\n" );
}
```

Sample Output

```
cat cow dog goat horse human pig rat
cat found at 002F0F04
```

.NET Framework Equivalent

System::Collections::ArrayList::BinarySearch

See Also

Reference

Searching and Sorting
_lfind
_lsearch
qsort

bsearch_s

Performs a binary search of a sorted array. This is version of bsearch with security enhancements as described in Security Enhancements in the CRT.

```
void *bsearch_s(
   const void *key,
   const void *base,
   size_t num,
   size_t width,
   int ( __cdecl *compare ) ( void *, const void *, const void *),
   void * context
);
```

Parameters

key

Object to search for.

base

Pointer to base of search data.

num

Number of elements.

width

Width of elements.

compare

Callback function that compares two elements. The first argument is the *context* pointer. The second argument is a pointer to the *key* for the search. The third argument is a pointer to the array element to be compared with *key*.

context

A pointer to an object that can be accessed in the comparison function.

Return Value

bsearch_s returns a pointer to an occurrence of *key* in the array pointed to by *base*. If *key* is not found, the function returns **NULL**. If the array is not in ascending sort order or contains duplicate records with identical keys, the result is unpredictable.

If invalid parameters are passed to the function, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **NULL**. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Error Conditions

key	*base*	*compare*	*num*	*width*	*errno*
NULL	any	any	any	any	**EINVAL**
any	**NULL**	any	!= 0	any	**EINVAL**
any	any	any	any	= 0	**EINVAL**
any	any	**NULL**	an	any	**EINVAL**

Remarks

The **bsearch_s** function performs a binary search of a sorted array of *num* elements, each of *width* bytes in size. The *base* value is a pointer to the base of the array to be searched, and *key* is the value being sought. The *compare* parameter is a pointer to a

user-supplied routine that compares two array elements and returns a value specifying their relationship. **bsearch_s** calls the *compare* routine one or more times during the search, passing the *context* pointer and pointers to two array elements on each call. The *compare* routine compares the elements, and then returns one of the following values:

Value returned by *compare* routine	**Description**
< 0	*elem1* less than *elem2*
0	*elem1* equal to *elem2*
> 0	*elem1* greater than *elem2*

The *context* pointer may be useful if the searched data structure is part of an object, and the compare function needs to access members of the object. The *compare* function may cast the void pointer into the appropriate object type and access members of that object. The addition of the *context* parameter makes **bsearch_s** more secure since additional context may be used to avoid reentrancy bugs associated with using static variables to make data available to the *compare* function.

Requirements

Routine	**Required header**	**Compatibility**
bsearch_s	<stdlib.h> and <search.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

This program sorts a string array with qsort_s, and then uses bsearch_s to find the word "cat".

```
// crt_bsearch_s.cpp
// This program uses bsearch_s to search a string array,
// passing a locale as the context.
// compile with: /EHsc
#include <stdlib.h>
#include <stdio.h>
#include <search.h>
#include <process.h>
#include <locale.h>
#include <locale>
#include <windows.h>
using namespace std;

// The sort order is dependent on the code page.  Use 'chcp' at the
// command line to change the codepage.  When executing this application,
// the command prompt codepage must match the codepage used here:

#define CODEPAGE_850

#ifdef CODEPAGE_850
#define ENGLISH_LOCALE "English_US.850"
#endif

#ifdef CODEPAGE_1252
#define ENGLISH_LOCALE "English_US.1252"
#endif

// The context parameter lets you create a more generic compare.
// Without this parameter, you would have stored the locale in a
// static variable, thus making it vulnerable to thread conflicts
// (if this were a multithreaded program).

int compare( void *pvlocale, char **str1, char **str2)
{
    char *s1 = *str1;
```

```
     char *s2 = *str2;

     locale& loc = *( reinterpret_cast< locale * > ( pvlocale));

     return use_facet< collate<char> >(loc).compare(
        s1, s1+strlen(s1),
        s2, s2+strlen(s2) );
}

int main( void )
{
   char *arr[] = {"dog", "pig", "horse", "cat", "human", "rat", "cow", "goat"};

   char *key = "cat";
   char **result;
   int i;

   /* Sort using Quicksort algorithm: */
   qsort_s( arr,
            sizeof(arr)/sizeof(arr[0]),
            sizeof( char * ),
            (int (*)(void*, const void*, const void*))compare,
            &locale(ENGLISH_LOCALE) );

   for( i = 0; i < sizeof(arr)/sizeof(arr[0]); ++i )    /* Output sorted list */
      printf( "%s ", arr[i] );

   /* Find the word "cat" using a binary search algorithm: */
   result = (char **)bsearch_s( &key,
                                arr,
                                sizeof(arr)/sizeof(arr[0]),
                                sizeof( char * ),
                                (int (*)(void*, const void*, const void*))compare,
                                &locale(ENGLISH_LOCALE) );
   if( result )
      printf( "\n%s found at %Fp\n", *result, result );
   else
      printf( "\nCat not found!\n" );
}
```

Sample Output

```
cat cow dog goat horse human pig rat
cat found at 002F0F04
```

.NET Framework Equivalent

BinarySearch

See Also

Reference

Searching and Sorting
_lfind
_lsearch
qsort

btowc

Determine whether an integer represents a valid single-byte character in the initial shift state.

```
wint_t btowc(
   int c
);
```

Parameters

c

Integer to test.

Return Value

Returns the wide-character representation of the character if the integer represents a valid single-byte character in the initial shift state. Returns WEOF if the integer is EOF or is not a valid single-byte character in the initial shift state. The output of this function is affected by the current **LC_TYPE** locale.

Requirements

Routine	Required header	Compatibility
btowc	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

mbtowc, _mbtowc_l

_byteswap_uint64, _byteswap_ulong, _byteswap_ushort

Reverses the order of bytes in an integer.

```
unsigned short _byteswap_ushort (
   unsigned short val
);
unsigned long _byteswap_ulong (
   unsigned long val
);
unsigned __int64 _byteswap_uint64 (
   unsigned __int64 val
);
```

Parameters

val

The integer to reverse byte order.

Requirements

Routine	Required header	Compatibility
_byteswap_ushort	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_byteswap_ulong	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_byteswap_uint64	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_byteswap.c
#include <stdlib.h>

int main()
{
   unsigned __int64 u64 = 0x0102030405060708;
   unsigned long ul = 0x01020304;

   printf("byteswap of %I64x = %I64x\n", u64, _byteswap_uint64(u64));
   printf("byteswap of %Ix = %Ix", ul, _byteswap_ulong(ul));
}
```

Output

```
byteswap of 102030405060708 = 807060504030201
byteswap of 1020304 = 4030201
```

See Also

Reference

Run-Time Routines by Category

cabs

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _cabs instead.

_cabs

Calculates the absolute value of a complex number.

```
double _cabs(
   struct _complex z
);
```

Parameters

z

Complex number.

Return Value

_cabs returns the absolute value of its argument if successful. On overflow, **_cabs** returns **HUGE_VAL** and sets **errno** to **ERANGE**. You can change error handling with _matherr.

Remarks

The **_cabs** function calculates the absolute value of a complex number, which must be a structure of type _complex. The structure *z* is composed of a real component *x* and an imaginary component *y*. A call to **_cabs** produces a value equivalent to that of the expression **sqrt**(*z.x**z.x* **+** *z.y**z.y*).

Requirements

Routine	Required header	Compatibility
_cabs	<math.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_cabs.c
/* Using _cabs, this program calculates
 * the absolute value of a complex number.
 */
#include <math.h>
#include <stdio.h>

int main( void )
{
   struct _complex number = { 3.0, 4.0 };
   double d;

   d = _cabs( number );
   printf( "The absolute value of %f + %fi is %f\n",
           number.x, number.y, d );
}
```

Output

```
The absolute value of 3.000000 + 4.000000i is 5.000000
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support
abs, _abs64
fabs, fabsf

labs

calloc

Allocates an array in memory with elements initialized to 0.

```
void *calloc(
   size_t num,
   size_t size
);
```

Parameters

num

Number of elements.

size

Length in bytes of each element.

Return Value

calloc returns a pointer to the allocated space. The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than **void**, use a type cast on the return value.

Remarks

The **calloc** function allocates storage space for an array of *num* elements, each of length *size* bytes. Each element is initialized to 0.

In Visual C++ 2005, **calloc** sets **errno** to **ENOMEM** if a memory allocation fails or if the amount of memory requested exceeds **_HEAP_MAXREQ**. For information on this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

calloc calls **malloc** to use the C++ _set_new_mode function to set the new handler mode. The new handler mode indicates whether, on failure, **malloc** is to call the new handler routine as set by _set_new_handler. By default, **malloc** does not call the new handler routine on failure to allocate memory. You can override this default behavior so that, when **calloc** fails to allocate memory, **malloc** calls the new handler routine in the same way that the **new** operator does when it fails for the same reason. To override the default, call

```
_set_new_mode(1)
```

early in your program, or link with NEWMODE.OBJ (see Link Options).

When the application is linked with a debug version of the C run-time libraries, **calloc** resolves to _calloc_dbg. For more information about how the heap is managed during the debugging process, see The CRT Debug Heap.

calloc is marked **__declspec(noalias)** and **__declspec(restrict)**, meaning that the function is guaranteed not to modify global variables, and that the pointer returned is not aliased. For more information, see noalias and restrict.

Requirements

Routine	Required header	Compatibility
calloc	<stdlib.h> and <malloc.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_calloc.c
// This program uses calloc to allocate space for
// 40 long integers. It initializes each element to zero.

#include <stdio.h>
#include <malloc.h>
```

```
int main( void )
{
   long *buffer;

   buffer = (long *)calloc( 40, sizeof( long ) );
   if( buffer != NULL )
      printf( "Allocated 40 long integers\n" );
   else
      printf( "Can't allocate memory\n" );
   free( buffer );
}
```

Output

```
Allocated 40 long integers
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation
free
malloc
realloc

_calloc_dbg

Allocates a number of memory blocks in the heap with additional space for a debugging header and overwrite buffers (debug version only).

```
void *_calloc_dbg(
   size_t num,
   size_t size,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

num

Requested number of memory blocks.

size

Requested size of each memory block (bytes).

blockType

Requested type of memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

For information about the allocation block types and how they are used, see Types of Blocks on the Debug Heap.

filename

Pointer to name of the source file that requested allocation operation or **NULL**.

linenumber

Line number in the source file where allocation operation was requested or **NULL**.

The *filename* and *linenumber* parameters are only available when **_calloc_dbg** has been called explicitly or the _CRTDBG_MAP_ALLOC preprocessor constant has been defined.

Return Value

On successful completion, this function returns a pointer to the user portion of the last allocated memory block, calls the new handler function, or returns **NULL**. For a complete description of the return behavior, see the Remarks section. For more information about how the new handler function is used, see the calloc function.

Remarks

_calloc_dbg is a debug version of the calloc function. When _DEBUG is not defined, each call to **_calloc_dbg** is reduced to a call to **calloc**. Both **calloc** and **_calloc_dbg** allocate *num* memory blocks in the base heap, but **_calloc_dbg** offers several debugging features:

- Buffers on either side of the user portion of the block to test for leaks.
- A block type parameter to track specific allocation types.
- *filename/linenumber* information to determine the origin of allocation requests.

_calloc_dbg allocates each memory block with slightly more space than the requested *size*. The additional space is used by the debug heap manager to link the debug memory blocks and to provide the application with debug header information and overwrite buffers. When the block is allocated, the user portion of the block is filled with the value 0xCD and each of the overwrite buffers are filled with 0xFD.

In Visual C++ 2005, **_calloc_dbg** sets **errno** to **ENOMEM** if a memory allocation fails; **EINVAL** is returned if the amount of memory needed (including the overhead mentioned previously) exceeds **_HEAP_MAXREQ**. For information about this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see

Memory Management and the Debug Heap. For information about the differences between calling a standard heap function versus its debug version in a debug build of an application, see Using the Debug Version Versus the Base Version.

Requirements

Routine	Required header	Compatibility
_calloc_dbg	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_callocd.c
/*
 * This program uses _calloc_dbg to allocate space for
 * 40 long integers. It initializes each element to zero.
 */
#include <stdio.h>
#include <malloc.h>
#include <crtdbg.h>

int main( void )
{
        long *bufferN, *bufferC;

        /*
         * Call _calloc_dbg to include the filename and line number
         * of our allocation request in the header and also so we can
         * allocate CLIENT type blocks specifically
         */
        bufferN = (long *)_calloc_dbg( 40, sizeof(long), _NORMAL_BLOCK, __FILE__, __LINE__
);
        bufferC = (long *)_calloc_dbg( 40, sizeof(long), _CLIENT_BLOCK, __FILE__, __LINE__
);
        if( bufferN != NULL && bufferC != NULL )
              printf( "Allocated memory successfully\n" );
        else
              printf( "Problem allocating memory\n" );

        /*
         * _free_dbg must be called to free CLIENT type blocks
         */
        free( bufferN );
        _free_dbg( bufferC, _CLIENT_BLOCK );
}
```

Output

```
Allocated memory successfully
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines
calloc
_malloc_dbg
_DEBUG

ceil, ceilf

Calculates the ceiling of a value.

```
double ceil(
   double x
);
float ceil(
   float x
);  // C++ only
long double ceil(
   long double x
);  // C++ only
float ceilf(
   float x
);
```

Parameters

x

Floating-point value.

Return Value

The **ceil** function returns a **double** value representing the smallest integer that is greater than or equal to *x*. There is no error return.

Input	SEH Exception	Matherr Exception
± **QNAN**,**IND**	none	**_DOMAIN**

ceil has an implementation that uses Streaming SIMD Extensions 2 (SSE2). See _set_SSE2_enable for information and restrictions on using the SSE2 implementation.

Remarks

C++ allows overloading, so you can call overloads of **ceil**. In a C program, **ceil** always takes and returns a double.

Requirements

Routine	Required header	Compatibility
ceil, **ceilf**	<math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for floor.

.NET Framework Equivalent

System::Math::Ceiling

See Also

Reference

Floating-Point Support
floor, floorf
fmod, fmodf

_cexit, _c_exit

Performs cleanup operations and returns without terminating the process.

```
void _cexit( void );
void _c_exit( void );
```

Remarks

The **_cexit** function calls, in last-in, first-out (LIFO) order, the functions registered by **atexit** and **_onexit**. Then **_cexit** flushes all I/O buffers and closes all open streams before returning. **_c_exit** is the same as **_exit** but returns to the calling process without processing **atexit** or **_onexit** or flushing stream buffers. The behavior of **exit**, **_exit**, **_cexit**, and **_c_exit** is shown in the following table.

Function	Behavior
exit	Performs complete C library termination procedures, terminates process, and exits with supplied status code.
_exit	Performs quick C library termination procedures, terminates process, and exits with supplied status code.
_cexit	Performs complete C library termination procedures and returns to caller, but does not terminate process.
_c_exit	Performs quick C library termination procedures and returns to caller, but does not terminate process.

When you call the **_cexit** or **_c_exit** functions, the destructors for any temporary or automatic objects that exist at the time of the call are not called. An automatic object is an object that is defined in a function where the object is not declared to be static. A temporary object is an object created by the compiler. To destroy an automatic object before calling **_cexit** or **_c_exit**, explicitly call the destructor for the object, as follows:

```
myObject.myClass::~myClass( );
```

Requirements

Routine	Required header	Compatibility
_cexit	<process.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_c_exit	<process.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::Diagnostics::Process::CloseMainWindow

See Also

Reference

Process and Environment Control
abort
atexit
_exec, _wexec Functions
exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

cgets

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _cgets or security-enhanced _cgets_s instead.

_cgets, _cgetws

Gets a character string from the console. These functions are deprecated because more secure versions are available; see _cgets_s, _cgetws_s.

```
char *_cgets(
   char *buffer
);
wchar_t *_cgetws(
   wchar_t *buffer
);
template <size_t size>
char *_cgets(
   char (&buffer)[size]
); // C++ only
template <size_t size>
wchar_t *_cgetws(
   wchar_t (&buffer)[size]
); // C++ only
```

Parameters

buffer

Storage location for data.

Return Value

_cgets and **_cgetws** return a pointer to the start of the string, at *buffer[2]*. If *buffer* is **NULL**, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, they return **NULL** and set **errno** to **EINVAL**.

Remarks

These functions read a string of characters from the console and store the string and its length in the location pointed to by *buffer*. The *buffer* parameter must be a pointer to a character array. The first element of the array, *buffer[0]*, must contain the maximum length (in characters) of the string to be read. The array must contain enough elements to hold the string, a terminating null character ('\0'), and 2 additional bytes. The function reads characters until a carriage return–line feed (CR-LF) combination or the specified number of characters is read. The string is stored starting at *buffer[2]*. If the function reads a CR-LF, it stores the null character ('\0'). The function then stores the actual length of the string in the second array element, *buffer[1]*.

Because all editing keys are active when **_cgets** or **_cgetws** is called while in a console window, pressing the F3 key repeats the last entered entry.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_cgetts	**_cgets**	**_cgets**	**_cgetws**

Requirements

Routine	Required header	Compatibility
_cgets	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_cgetws	<conio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_cgets.c
// compile with: /c /W3
// This program creates a buffer and initializes
// the first byte to the size of the buffer. Next, the
// program accepts an input string using _cgets and displays
// the size and text of that string.

#include <conio.h>
#include <stdio.h>
#include <errno.h>

int main( void )
{
   char buffer[83] = { 80 };  // Maximum characters in 1st byte
   char *result;

   printf( "Input line of text, followed by carriage return:\n");

   // Input a line of text:
   result = _cgets( buffer ); // C4996
   // Note: _cgets is deprecated; consider using _cgets_s
   if (!result)
   {
      printf( "An error occurred reading from the console:"
              " error code %d\n", errno);
   }
   else
   {
      printf( "\nLine length = %d\nText = %s\n",
              buffer[1], result );
   }
}
```

Input

```
A line of input.
```

Output

```
Input line of text, followed by carriage return:
Line Length = 16
Text = A line of input.
```

See Also

Reference

Console and Port I/O

_getch, _getwch

_cgets_s, _cgetws_s

Gets a character string from the console. This is a version of _cgets and _cgetws with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _cgets_s(
   char *buffer,
   size_t numberOfElements,
   size_t *pSizeRead
);
errno_t _cgetws_s(
   wchar_t *buffer
   size_t numberOfElements,
   size_t *pSizeRead
);
template <size_t size>
errno_t _cgets_s(
   char (&buffer)[size],
   size_t *pSizeRead
); // C++ only
template <size_t size>
errno_t _cgetws_s(
   wchar_t (&buffer)[size],
   size_t *pSizeRead
); // C++ only
```

Parameters

[out] *buffer*

Storage location for data.

[in] *numberOfElements*

The size of the buffer, which is also the maximum number of characters to be read.

[in] *pSizeRead*

The number of characters actually read.

Return Value

The return value is zero if successful; an error code if a failure occurs.

Error Conditions

buffer	***numberOfElements***	***pSizeRead***	**Return**	**Contents of *buffer***
NULL	any	any	**EINVAL**	n/a
not **NULL**	zero	any	**EINVAL**	not modified
not **NULL**	any	**NULL**	**EINVAL**	zero-length string

Remarks

_cgets_s and **_cgetws_s** read a string from the console and copy the string (with a null terminator) into *buffer*. **_cgetws_s** is the wide character version of the function; other than the character size, the behavior of these two functions is identical. The maximum size of the string to be read is passed in as the *numberOfElements* parameter. This size should include an extra character for the terminating null. The actual number of characters read is placed in *pSizeRead*.

If an error occurs during the operation or in validating the parameters, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, **errno** is set to **EINVAL** and **EINVAL** is returned.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_cgetts_s	**_cgets_s**	**_cgets_s**	**_cgetws_s**

Requirements

Routine	Required header	Compatibility
_cgets_s	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_cgetws_s	<conio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Console and Port I/O

_getch, _getwch

chdir

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _chdir instead.

_chdir, _wchdir

Changes the current working directory.

```
int _chdir(
   const char *dirname
);
int _wchdir(
   const wchar_t *dirname
);
```

Parameters

dirname

Path of new working directory.

Return Value

These functions return a value of 0 if successful. A return value of –1 indicates failure. If the specified path could not be found, **errno** is set to **ENOENT**. If *dirname* is NULL, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns -1.

Remarks

The **_chdir** function changes the current working directory to the directory specified by *dirname*. The *dirname* parameter must refer to an existing directory. This function can change the current working directory on any drive. If a new drive letter is specified in *dirname*, the default drive letter is changed as well. For example, if A is the default drive letter and \BIN is the current working directory, the following call changes the current working directory for drive C and establishes C as the new default drive:

```
_chdir("c:\\temp");
```

When you use the optional backslash character (****) in paths, you must place two backslashes (****) in a C string literal to represent a single backslash (****).

_wchdir is a wide-character version of **_chdir**; the *dirname* argument to **_wchdir** is a wide-character string**. _wchdir** and **_chdir** behave identically otherwise.

Generic-Text Routine Mapping:

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tchdir	**_chdir**	**_chdir**	**_wchdir**

Requirements

Routine	Required header	Optional header	Compatibility
_chdir	<direct.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wchdir	<direct.h> or <wchar.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_chdir.c
// arguments: C:\WINDOWS

/* This program uses the _chdir function to verify
```

```
   that a given directory exists. */

#include <direct.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main( int argc, char *argv[] )
{

   if(_chdir( argv[1] ) )
   {
      switch (errno)
      {
      case ENOENT:
         printf( "Unable to locate the directory: %s\n", argv[1] );
         break;
      case EINVAL:
         printf( "Invalid buffer.\n");
         break;
      default:
         printf( "Unknown error.\n");
      }
   }
   else
      system( "dir *.exe");
}
```

Sample Output

```
Volume in drive C has no label.
 Volume Serial Number is 2018-08A1

 Directory of c:\windows

08/29/2002  04:00 AM         1,004,032 explorer.exe
12/17/2002  04:43 PM            10,752 hh.exe
03/03/2003  09:24 AM            33,792 ieuninst.exe
10/29/1998  04:45 PM           306,688 IsUninst.exe
08/29/2002  04:00 AM            66,048 NOTEPAD.EXE
03/03/2003  09:24 AM            33,792 Q330994.exe
08/29/2002  04:00 AM           134,144 regedit.exe
02/28/2003  06:26 PM            46,352 setdebug.exe
08/29/2002  04:00 AM            15,360 TASKMAN.EXE
08/29/2002  04:00 AM            49,680 twunk_16.exe
08/29/2002  04:00 AM            25,600 twunk_32.exe
08/29/2002  04:00 AM           256,192 winhelp.exe
08/29/2002  04:00 AM           266,752 winhlp32.exe
              13 File(s)      2,249,184 bytes
               0 Dir(s)  67,326,029,824 bytes free
```

.NET Framework Equivalent

System::Environment::CurrentDirectory

See Also

Reference

Directory Control
_mkdir, _wmkdir
_rmdir, _wrmdir
system, _wsystem

_chdrive

Changes the current working drive.

```
int _chdrive(
   int drive
);
```

Parameters

drive

Number of the new working drive.

Return Value

_chdrive returns a value of 0 if the working drive is successfully changed. A return value of –1 indicates an error.

Remarks

The **_chdrive** function changes the current working drive to the drive specified by *drive*. The *drive* parameter uses an integer to specify the new working drive (1=A, 2=B, and so forth). This function changes only the working drive; **_chdir** changes the working directory.

If *drive* does not specify a valid drive, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns -1 and **errno** is set to **EACCES**, and **_doserrno** is set to **ERROR_INVALID_DRIVE**.

Requirements

Routine	Required header	Compatibility
_chdrive	<direct.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example for _getdrive.

.NET Framework Equivalent

System::Environment::CurrentDirectory

See Also

Reference

Directory Control
_chdir, _wchdir
_fullpath, _wfullpath
_getcwd, _wgetcwd
_getdrive
_mkdir, _wmkdir
_rmdir, _wrmdir
system, _wsystem

_chgsign, _chgsignl

Reverses the sign of a double-precision, floating-point or a long double-precision, floating-point argument.

```
double _chgsign(
   double x
);
long double _chgsignl(
   long double x
);
```

Parameters

x

Double-precision or long double-precision, floating-point value to be changed.

Return Value

_chgsign returns a value equal to its double-precision, floating-point argument *x*, but with its sign reversed. **_chgsignl** returns a value equal to its long double-precision, floating-point argument *x*, but with its sign reversed. There is no error return.

Requirements

Routine	Required header	Compatibility
_chgsign	<float.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_chgsignl	<math.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support
fabs, fabsf
_copysign, _copysignl

chmod

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _chmod instead.

_chmod, _wchmod

Changes the file-permission settings.

```
int _chmod(
   const char *filename,
   int pmode
);
int _wchmod(
   const wchar_t *filename,
   int pmode
);
```

Parameters

filename

Name of the existing file.

pmode

Permission setting for the file.

Return Value

These functions return 0 if the permission setting is successfully changed. A return value of –1 indicates failure. If the specified file could not be found, **errno** is set to **ENOENT**; if a parameter is invalid, **errno** is set to **EINVAL**.

Remarks

The **_chmod** function changes the permission setting of the file specified by *filename*. The permission setting controls the read and write access to the file. The integer expression *pmode* contains one or both of the following manifest constants, defined in SYS\Stat.h.

_S_IWRITE

Writing permitted.

_S_IREAD

Reading permitted.

_S_IREAD | _S_IWRITE

Reading and writing permitted.

When both constants are given, they are joined with the bitwise **OR** operator (**|**). If write permission is not given, the file is read-only. Note that all files are always readable; it is not possible to give write-only permission. Thus, the modes **_S_IWRITE** and **_S_IREAD | _S_IWRITE** are equivalent.

_wchmod is a wide-character version of **_chmod**; the **filename** argument to **_wchmod** is a wide-character string. **_wchmod** and **_chmod** behave identically otherwise.

This function validates its parameters. If *pmode* is not a combination of one of the manifest constants or incorporates an alternate set of constants, the function simply ignores those. If **filename** is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns -1.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tchmod	_chmod	_chmod	_wchmod

Requirements

Routine	Required header	Optional header	Compatibility

_chmod	<io.h>	<sys/types.h>, <sys/stat.h>, <errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wchmod	<io.h> or <wchar.h>	<sys/types.h>, <sys/stat.h>, <errno.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_chmod.c
// This program uses _chmod to
// change the mode of a file to read-only.
// It then attempts to modify the file.
//

#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

// Change the mode and report error or success
void set_mode_and_report(char * filename, int mask)
{
   // Check for failure
   if( _chmod( filename, mask ) == -1 )
   {
      // Determine cause of failure and report.
      switch (errno)
      {
         case EINVAL:
            fprintf( stderr, "Invalid parameter to chmod.\n");
            break;
         case ENOENT:
            fprintf( stderr, "File %s not found\n", filename );
            break;
         default:
            // Should never be reached
            fprintf( stderr, "Unexpected error in chmod.\n" );
       }
   }
   else
   {
      if (mask == _S_IREAD)
        printf( "Mode set to read-only\n" );
      else if (mask & _S_IWRITE)
        printf( "Mode set to read/write\n" );
   }
   fflush(stderr);
}

int main( void )
{

   // Create or append to a file.
   system( "echo /* End of file */ >> crt_chmod.c_input" );

   // Set file mode to read-only:
   set_mode_and_report("crt_chmod.c_input ", _S_IREAD );

   system( "echo /* End of file */ >> crt_chmod.c_input " );

   // Change back to read/write:
```

```
    set_mode_and_report("crt_chmod.c_input ", _S_IWRITE );

    system( "echo /* End of file */ >> crt_chmod.c_input " );
}
```

Input

```
A line of text.
```

Sample Output

```
Mode set to read-only
Access is denied.
Mode set to read/write
```

.NET Framework Equivalent

- System::IO::File::SetAttributes
- System::Security::Permissions::FileIOPermission

See Also

Reference

File Handling
_access, _waccess
_creat, _wcreat
_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32
_open, _wopen
_stat, _wstat Functions

chsize

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _chsize or security-enhanced _chsize_s instead.

_chsize

Changes the size of a file. A more secure version is available; see _chsize_s.

```
int _chsize(
   int fd,
   long size
);
```

Parameters

fd

File descriptor referring to an open file.

size

New length of the file in bytes.

Return Value

_chsize returns the value 0 if the file size is successfully changed. A return value of –1 indicates an error: **errno** is set to **EACCES** if the specified file is locked against access, to **EBADF** if the specified file is read-only or the descriptor is invalid, **ENOSPC** if no space is left on the device, or **EINVAL** if *size* is less than zero.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

Remarks

The **_chsize** function extends or truncates the file associated with *fd* to the length specified by *size*. The file must be open in a mode that permits writing. Null characters ('\0') are appended if the file is extended. If the file is truncated, all data from the end of the shortened file to the original length of the file is lost.

This function validates its parameters. If *size* is less than zero or *fd* is a bad file descriptor, the invalid parameter handler is invoked, as described in Parameter Validation.

Requirements

Routine	Required header	Optional header	Compatibility
_chsize	<io.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_chsize.c
// This program uses _filelength to report the size
// of a file before and after modifying it with _chsize.

#include <io.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <share.h>

int main( void )
{
   int fh, result;
   unsigned int nbytes = BUFSIZ;

   // Open a file
   if( _sopen_s( &fh, "data", _O_RDWR | _O_CREAT, _SH_DENYNO,
```

```
                    _S_IREAD | _S_IWRITE ) == 0 )
    {
       printf( "File length before: %ld\n", _filelength( fh ) );
       if( ( result = _chsize( fh, 329678 ) ) == 0 )
          printf( "Size successfully changed\n" );
       else
          printf( "Problem in changing the size\n" );
       printf( "File length after:  %ld\n", _filelength( fh ) );
       _close( fh );
    }
 }
```

Output

```
File length before: 0
Size successfully changed
File length after:  329678
```

.NET Framework Equivalent

- System::IO::Stream::SetLength
- System::IO::FileStream::SetLength

See Also

Reference

File Handling
_close
_sopen, _wsopen
_open, _wopen

_chsize_s

Changes the size of a file. This is a version of _chsize with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _chsize_s(
   int fd,
   __int64 size
);
```

Parameters

fd

File descriptor referring to an open file.

size

New length of the file in bytes.

Return Value

_chsize_s returns the value 0 if the file size is successfully changed. A nonzero return value indicates an error: the return value is **EACCES** if the specified file is locked against access, **EBADF** if the specified file is read-only or the descriptor is invalid, **ENOSPC** if no space is left on the device, or **EINVAL** if size is less than zero. **errno** is set to the same value.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_chsize_s** function extends or truncates the file associated with *fd* to the length specified by *size*. The file must be open in a mode that permits writing. Null characters ('\0') are appended if the file is extended. If the file is truncated, all data from the end of the shortened file to the original length of the file is lost.

_chsize_s takes a 64-bit integer as the file size, and therefore can handle file sizes greater than 4 GB. **_chsize** is limited to 32-bit file sizes.

This function validates its parameters. If *fd* is not a valid file descriptor or size is less than zero, the invalid parameter handler is invoked, as described in Parameter Validation.

Requirements

Routine	Required header	Optional header	Compatibility
_chsize_s	<io.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

- System::IO::Stream::SetLength
- System::IO::FileStream::SetLength

See Also

Reference

File Handling
_chsize
_close
_creat, _wcreat
_open, _wopen

_clear87, _clearfp

Gets and clears the floating-point status word.

```
unsigned int _clear87( void );
unsigned int _clearfp( void );
```

Return Value

The bits in the value returned indicate the floating-point status before the call to **_clear87** or **_clearfp**. For a complete definition of the bits returned by **_clear87**, see Float.h. Many of the math library functions modify the 8087/80287 status word, with unpredictable results. Return values from **_clear87** and **_status87** become more reliable as fewer floating-point operations are performed between known states of the floating-point status word.

Remarks

The **_clear87** function clears the exception flags in the floating-point status word, sets the busy bit to 0, and returns the status word. The floating-point status word is a combination of the 8087/80287 status word and other conditions detected by the 8087/80287 exception handler, such as floating-point stack overflow and underflow.

_clearfp is a platform-independent, portable version of the **_clear87** routine. It is identical to **_clear87** on Intel (x86) platforms and is also supported by the MIPS and ALPHA platforms. To ensure that your floating-point code is portable to MIPS or ALPHA, use **_clearfp**. If you are only targeting x86 platforms, you can use either **_clear87** or **_clearfp**.

These functions are deprecated when compiling with /clr (Common Language Runtime Compilation) or **/clr:pure** because the common language runtime only supports the default floating-point precision.

Requirements

Routine	Required header	Compatibility
_clear87	<float.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_clearfp	<float.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_clear87.c
// compile with: /Od

// This program creates various floating-point
// problems, then uses _clear87 to report on these problems.
// Compile this program with Optimizations disabled (/Od).
// Otherwise the optimizer will remove the code associated with
// the unused floating-point values.
//

#include <stdio.h>
#include <float.h>

int main( void )
{
   double a = 1e-40, b;
   float x, y;

   printf( "Status: %.4x - clear\n", _clear87()  );

   // Store into y is inexact and underflows:
   y = a;
```

```
    printf( "Status: %.4x - inexact, underflow\n", _clear87() );

    // y is denormal:
    b = y;
    printf( "Status: %.4x - denormal\n", _clear87() );
}
```

Output

```
Status: 0000 - clear
Status: 0003 - inexact, underflow
Status: 80000 - denormal
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support
_control87, _controlfp, __control87_2
_status87, _statusfp, _statusfp2

clock

Calculates the wall-clock time used by the calling process.

```
clock_t clock( void );
```

Return Value

The elapsed wall-clock time since the start of the process (elapsed time in seconds times **CLOCKS_PER_SEC**). If the amount of elapsed time is unavailable, the function returns –1, cast as a **clock_t**.

Remarks

The **clock** function tells how much time the calling process has used. A timer tick is approximately equal to 1/**CLOCKS_PER_SEC** second. In versions of Microsoft C before 6.0, the **CLOCKS_PER_SEC** constant was called **CLK_TCK**.

Requirements

Routine	Required header	Compatibility
clock	<time.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_clock.c
// This example prompts for how long
// the program is to run and then continuously
// displays the elapsed time for that period.
//

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void sleep( clock_t wait );

int main( void )
{
   long    i = 6000000L;
   clock_t start, finish;
   double  duration;

   // Delay for a specified time.
   printf( "Delay for three seconds\n" );
   sleep( (clock_t)3 * CLOCKS_PER_SEC );
   printf( "Done!\n" );

   // Measure the duration of an event.
   printf( "Time to do %ld empty loops is ", i );
   start = clock();
   while( i-- )
      ;
   finish = clock();
   duration = (double)(finish - start) / CLOCKS_PER_SEC;
   printf( "%2.1f seconds\n", duration );
}

// Pauses for a specified number of milliseconds.
void sleep( clock_t wait )
{
   clock_t goal;
```

```
    goal = wait + clock();
    while( goal > clock() )
       ;
}
```

Sample Output

```
Delay for three seconds
Done!
Time to do 6000000 empty loops is 0.1 seconds
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Time Management
difftime, _difftime32, _difftime64
time, _time32, _time64

clearerr

Resets the error indicator for a stream. A more secure version is available; see clearerr_s.

```
void clearerr(
   FILE *stream
);
```

Parameters

stream

Pointer to **FILE** structure.

Remarks

The **clearerr** function resets the error indicator and end-of-file indicator for *stream*. Error indicators are not automatically cleared; once the error indicator for a specified stream is set, operations on that stream continue to return an error value until **clearerr**, **fseek**, **fsetpos**, or **rewind** is called.

If *stream* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns. For more information on **errno** and error codes, see errno Constants.

A more secure version of this function is available; see clearerr_s.

Requirements

Routine	Required header	Compatibility
clearerr	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_clearerr.c
// This program creates an error
// on the standard input stream, then clears
// it so that future reads won't fail.

#include <stdio.h>

int main( void )
{
   int c;
   // Create an error by writing to standard input.
   putc( 'c', stdin );
   if( ferror( stdin ) )
   {
      perror( "Write error" );
      clearerr( stdin );
   }

   // See if read causes an error.
   printf( "Will input cause an error? " );
   c = getc( stdin );
   if( ferror( stdin ) )
   {
      perror( "Read error" );
      clearerr( stdin );
   }
   else
      printf( "No read error\n" );
}
```

Input

```
n
```

Sample Output

```
Write error: No error
Will input cause an error? n
No read error
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Error Handling (CRT)
Stream I/O
_eof
feof
ferror
perror, _wperror

clearerr_s

Resets the error indicator for a stream. This is a version of clearerr with security enhancements as described in Security Enhancements in the CRT.

```
errno_t clearerr_s(
   FILE *stream
);
```

Parameters

stream

Pointer to **FILE** structure

Return Value

Zero if successful; **EINVAL** if *stream* is NULL.

Remarks

The **clearerr_s** function resets the error indicator and end-of-file indicator for *stream*. Error indicators are not automatically cleared; once the error indicator for a specified stream is set, operations on that stream continue to return an error value until **clearerr_s**, **clearerr**, **fseek**, **fsetpos**, or **rewind** is called.

If *stream* is NULL, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Requirements

Routine	Required header	Compatibility
clearerr_s	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_clearerr_s.c
// This program creates an error
// on the standard input stream, then clears
// it so that future reads won't fail.


#include <stdio.h>

int main( void )
{
   int c;
   errno_t err;

   // Create an error by writing to standard input.
   putc( 'c', stdin );
   if( ferror( stdin ) )
   {
      perror( "Write error" );
      err = clearerr_s( stdin );
      if (err != 0)
      {
         abort();
      }
   }

   // See if read causes an error.
   printf( "Will input cause an error? " );
```

```
    c = getc( stdin );
    if( ferror( stdin ) )
    {
       perror( "Read error" );
       err = clearerr_s( stdin );
       if (err != 0)
       {
          abort();
       }
    }
 }
```

Input

```
n
```

Sample Output

```
Write error: Bad file descriptor
Will input cause an error? n
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Error Handling (CRT)
Stream I/O
clearerr
_eof
feof
ferror
perror, _wperror

close

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _close instead.

_close

Closes a file.

```
int _close(
   int fd
);
```

Parameters

fd

File descriptor referring to the open file.

Return Value

_close returns 0 if the file was successfully closed. A return value of –1 indicates an error.

Remarks

The **_close** function closes the file associated with *fd*.

The file descriptor and the underlying OS file handle are closed. Thus, it is not necessary to call **CloseHandle** if the file was originally opened using the Win32 function **CreateFile** and converted to a file descriptor using **_open_osfhandle**.

This function validates its parameters. If *fd* is a bad file descriptor, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions returns -1 and **errno** is set to **EBADF**.

Requirements

Routine	Required header	Optional header	Compatibility
_close	<io.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example for _open.

See Also

Reference

Low-Level I/O
_chsize
_creat, _wcreat
_dup, _dup2
_open, _wopen
_unlink, _wunlink

_commit

Flushes a file directly to disk.

```
int _commit(
   int fd
);
```

Parameters

fd

File descriptor referring to the open file.

Return Value

_commit returns 0 if the file was successfully flushed to disk. A return value of –1 indicates an error.

Remarks

The **_commit** function forces the operating system to write the file associated with *fd* to disk. This call ensures that the specified file is flushed immediately, not at the operating system's discretion.

If *fd* is an invalid file descriptor, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns -1 and **errno** is set to **EBADF**.

Requirements

Routine	Required header	Optional headers	Compatibility
_commit	<io.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

See Also

Reference

Low-Level I/O
_creat, _wcreat
_open, _wopen
_read
_write

_configthreadlocale

Configures per-thread locale options.

```
int _configthreadlocale(
   int type
);
```

Parameters

type

The option or options to set. One of the options listed in the following table.

Return Value

The previous per-thread locale status (**_DISABLE_PER_THREAD_LOCALE** or **_ENABLE_PER_THREAD_LOCALE**), or -1 on failure.

Remarks

The **_configurethreadlocale** function is used to control the use of thread-specific locales.

_ENABLE_PER_THREAD_LOCALE

Make the current thread use a thread-specific locale. Subsequent calls to **setlocale** in this thread affect only the thread's own locale.

_DISABLE_PER_THREAD_LOCALE

Make the current thread use the global locale. Subsequent calls to **setlocale** in this thread affect other threads using the global locale.

zero

Retrieves the current setting for this particular thread.

These functions affect the behavior of **setlocale**, **_tsetlocale**, **_wsetlocale**, **_beginthread**, and **_beginthreadex**. If another method is used to create threads, the locale settings have no affect on those threads.

When per-thread locale is disabled, any subsequent call to **setlocale** or **_wsetlocale** changes the locale of all threads. When per-thread locale is enabled, **setlocale** or **_wsetlocale** only affects the current thread's locale. Per-thread locale can be enabled or disabled globally for all threads. When per-thread locale is enabled globally, **setlocale** or **_wsetlocale** on any thread (not just the current thread) only affect that thread.

If *type* is not one of the values listed in the table, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns -1.

Requirements

Routine	Required header	Compatibility
_configthreadlocale	<locale.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

```
// crt_configthreadlocale.cpp
//
// This program demonstrates the use of _configthreadlocale when
// using is two independent threads.
//

#include <locale.h>
#include <process.h>
#include <windows.h>
```

```c
#include <stdio.h>
#include <time.h>

#define BUFF_SIZE 100

// Retrieve the date and time in the current
// locale's format.
int get_time(unsigned char* str)
{
    __time64_t  ltime;
    struct tm   thetime;

    // Retieve the time
    _time64(&ltime);
    _gmtime64_s(&thetime, &ltime);

    // Format the current time structure into a string
    // using %#x is the long date representation,
    // appropriate to the current locale
    if (!strftime((char *)str, BUFF_SIZE, "%#x",
                  (const struct tm*)&thetime))
    {
        printf("strftime failed!\n");
        return -1;
    }
    return 0;
}

// This thread sets its locale to German
// and prints the time.
unsigned __stdcall SecondThreadFunc( void* pArguments )
{
    unsigned char str[BUFF_SIZE];

    // Set the thread local
    printf("The thread local is now set to %s.\n",
           setlocale(LC_ALL, "German"));

    // Retrieve the time string from the helper function
    if (get_time(str) == 0)
    {
        printf("The time in German locale is: '%s'\n", str);
    }

    _endthreadex( 0 );
    return 0;
}

// The main thread spawns a second thread (above) and then
// sets the locale to English and prints the time.
int main()
{
    HANDLE          hThread;
    unsigned        threadID;
    unsigned char   str[BUFF_SIZE];

    // Configure per-thread locale to cause all subsequently created
    // threads to have their own locale.
    _configthreadlocale(_ENABLE_PER_THREAD_LOCALE);

    // Retrieve the time string from the helper function
    printf("The thread local is now set to %s.\n",
           setlocale(LC_ALL, "English"));

    // Create the second thread.
    hThread = (HANDLE)_beginthreadex( NULL, 0, &SecondThreadFunc,
                                      NULL, 0, &threadID );
```

```
    if (get_time(str) == 0)
    {
        // Retrieve the time string from the helper function
        printf("The time in English locale is: '%s'\n\n", str);
    }

    // Wait for the created thread to finish.
    WaitForSingleObject( hThread, INFINITE );

    // Destroy the thread object.
    CloseHandle( hThread );
}
```

Sample Output

```
The thread local is now set to English_United States.1252.
The time in English locale is: 'Wednesday, May 12, 2004'

The thread local is now set to German_Germany.1252.
The time in German locale is: 'Mittwoch, 12. Mai 2004'
```

.NET Framework Equivalent

Not applicable. However, see Using the CurrentCulture Property.

See Also

Reference

setlocale, _wsetlocale
_beginthread, _beginthreadex
Locale

_control87, _controlfp, __control87_2

Gets and sets the floating-point control word. **_controlfp** is deprecated because a more secure version is available; see _controlfp_s.

```
unsigned int _control87(
   unsigned int new,
   unsigned int mask
);
unsigned int _controlfp(
   unsigned int new,
   unsigned int mask
);
int __control87_2(
   unsigned int new,
   unsigned int mask,
   unsigned int* x86_cw,
   unsigned int* sse2_cw
);
```

Parameters

new

New control-word bit values.

mask

Mask for new control-word bits to set.

x86_cw

Filled in with the control word for the x87 floating point unit. Pass in 0 (**NULL**) to set only the SSE2 control word.

sse2_cw

Control word for the SSE floating-point unit. Pass in 0 (**NULL**) to set only the x87 control word.

Return Value

For **_control87** and **_controlfp**, the bits in the value returned indicate the floating-point control state. For a complete definition of the bits returned by **_control87**, see FLOAT.H.

For **__control87_2**, the return value is 1, indicating success.

Remarks

The **_control87** function gets and sets the floating-point control word. The floating-point control word allows the program to change the precision, rounding, and infinity modes in the floating-point math package. You can also mask or unmask floating-point exceptions using **_control87**. If the value for *mask* is equal to 0, **_control87** gets the floating-point control word. If *mask* is nonzero, a new value for the control word is set: For any bit that is on (equal to 1) in *mask*, the corresponding bit in *new* is used to update the control word. In other words, *fpcntrl* **=** ((*fpcntrl* **& ~mask**) | (**new & mask**)) where *fpcntrl* is the floating-point control word.

Note
The run-time libraries mask all floating-point exceptions by default.

_controlfp is a platform-independent, portable version of **_control87**. It is nearly identical to the **_control87** function on Intel (x86) platforms and is supported by the MIPS and ALPHA platforms. To ensure that your floating-point code is portable to MIPS or ALPHA, use **_controlfp**. If you are targeting x86 platforms, use **_control87** or **_controlfp**.

The difference between **_control87** and **_controlfp** is the way these two functions treat DENORMAL values. For Intel (x86) platforms, **_control87** can set and clear the DENORMAL OPERAND exception mask. ALPHA platforms do not support this exception, and **_controlfp** does not modify the DENORMAL OPERAND exception mask. The following example demonstrates the difference:

```
_control87( _EM_INVALID, _MCW_EM );
// DENORMAL is unmasked by this call
_controlfp( _EM_INVALID, _MCW_EM );
// DENORMAL exception mask remains unchanged
```

The possible values for the mask constant (*mask*) and new control values (*new*) are shown in the following Hexadecimal Values table. Use the portable constants listed below (**_MCW_EM**, **_EM_INVALID**, and so forth) as arguments to these functions, rather than supplying the hexadecimal values explicitly.

ALPHA platforms support the DENORMAL input and output values in software. The default behavior of Windows NT on these platforms is to flush the DENORMAL input and output values to zero. **_controlfp** provides a new mask to preserve and flush the input and output DENORMAL values.

Intel (x86) platforms support the DENORMAL input and output values in hardware. The behavior is to preserve DENORMAL values. **_control87** does not provide a mask to change this behavior. The following example demonstrates this difference:

```
controlfp( _DN_SAVE, _MCW_DN);
// Denormal values preserved by software on ALPHA. NOP on x86
controlfp( _DN_FLUSH, _MCW_DN);
// Denormal values flushed to zero by hardware on ALPHA. Ignored on x86
```

Both **_control87** and **_controlfp** affect the control words for both the x87 and the SSE2, if present. The function **__control87_2** allows both the x87 and SSE2 floating point units to be controlled together or separately. If you want to affect both units, pass in the addresses of two integers to *x86_cw* and *sse2_cw*. If you only want to affect one unit, pass in an address for that parameter but pass in 0 (NULL) for the other. If 0 is passed for one of these parameters, the function has no effect on that floating point unit. This functionality could be useful in situations where part of the code uses the x87 floating point unit and another part of the code uses the SSE2 floating point unit. If you use **__control87_2** in one part of a program and set different values for the floating point control words, and then use **_control87** or **_controlfp** to further manipulate the control word, then **_control87** and **_controlfp** might be unable to return a single control word to represent the state of both floating point units. In such a case, these functions set the **EM_AMBIGUOUS** flag in the returned integer value to indicate that there is an inconsistency between the two control words. This is a warning that the returned control word might not represent the state of both floating point control words accurately.

On the x64 architecture, changing the floating point precision is not supported. If the precision control mask is used on that platform, an assertion and the invalid parameter handler is invoked, as described in Parameter Validation.

In Visual C++ 2005, **_controlfp** has been optimized for better performance on all machine architectures.

These functions are deprecated when compiling with /clr (Common Language Runtime Compilation) or **/clr:pure** because the common language runtime only supports the default floating-point precision.

Hexadecimal Values

For the **_MCW_EM** mask, clearing the mask sets the exception, which allows the hardware exception; setting the mask hides the exception. Note that if a **_EM_UNDERFLOW** or **_EM_OVERFLOW** occurs, no hardware exception is thrown until the next floating-point instruction is executed. To generate a hardware exception immediately after **_EM_UNDERFLOW** or **_EM_OVERFLOW**, call the FWAIT MASM instruction.

Mask	Hex value	Constant	Hex value
_MCW_DN (Denormal control)	0x03000000	**_DN_SAVE**	0x00000000
		_DN_FLUSH	0x01000000
_MCW_EM (Interrupt exception mask)	0x0008001F	**_EM_INVALID**	0x00000010
		_EM_DENORMAL	0x00080000
		_EM_ZERODIVIDE	0x00000008
		_EM_OVERFLOW	0x00000004
		_EM_UNDERFLOW	0x00000002
		_EM_INEXACT	0x00000001

_MCW_IC (Infinity control)	0x00040000	**_IC_AFFINE** **_IC_PROJECTIVE**	0x00040000 0x00000000
_MCW_RC (Rounding control)	0x00000300	**_RC_CHOP** **_RC_UP** **_RC_DOWN** **_RC_NEAR**	0x00000300 0x00000200 0x00000100 0x00000000
_MCW_PC (Precision control)	0x00030000	**_PC_24** (24 bits) **_PC_53** (53 bits) **_PC_64** (64 bits)	0x00020000 0x00010000 0x00000000

Requirements

Routine	Required header	Compatibility
_control87, **_controlfp**, **_control87_2**	<float.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003; x 86 processor only

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_cntrl87.c
// processor: x86
// This program uses __control87_2 to output the x87 control
// word, set the precision to 24 bits, and reset the status to
// the default.
//

#include <stdio.h>
#include <float.h>
#pragma fenv_access (on)

int main( void )
{
    double a = 0.1;
    unsigned int control_word_x87;

    // Show original x87 control word and do calculation.
    control_word_x87 = __control87_2(0, 0,
                                     &control_word_x87, 0);
    printf( "Original: 0x%.4x\n", control_word_x87 );
    printf( "%1.1f * %1.1f = %.15e\n", a, a, a * a );

    // Set precision to 24 bits and recalculate.
    control_word_x87 = __control87_2(_PC_24, MCW_PC,
                                     &control_word_x87, 0);
    printf( "24-bit:   0x%.4x\n", control_word_x87 );
    printf( "%1.1f * %1.1f = %.15e\n", a, a, a * a );

    // Restore default precision-control bits and recalculate.
    control_word_x87 = __control87_2( _CW_DEFAULT, MCW_PC,
                                     &control_word_x87, 0 );
    printf( "Default:  0x%.4x\n", control_word_x87 );
    printf( "%1.1f * %1.1f = %.15e\n", a, a, a * a );
}
```

Output

```
Original: 0x0001
0.1 * 0.1 = 1.000000000000000e-002
24-bit:   0x0001
0.1 * 0.1 = 9.999999776482582e-003
Default:  0x0001
0.1 * 0.1 = 1.000000000000000e-002
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support
_clear87, _clearfp
_status87, _statusfp, _statusfp2

_controlfp_s

Gets and sets the floating-point control word. This is a version of _control87, _controlfp, __control87_2 with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _controlfp_s(
    unsigned int *currentControl,
    unsigned int newControl,
    unsigned int mask
);
```

Parameters

currentControl

The current control-word bit value.

newControl

New control-word bit values.

mask

Mask for new control-word bits to set.

Return Value

Zero if successful or an **errno** value error code.

Remarks

The **_controlfp_s** is a platform-independent and more secure version of **_control87**, which gets and sets the floating-point control word into the address of *currentControl* and *newControl*, respectively. The bits in the values indicate the floating-point control state. The floating-point control state allows the program to change the precision, rounding, and infinity modes in the floating-point math package. One can also mask or unmask floating-point exceptions using **_controlfp_s**.

If the value for *mask* is equal to 0, **_controlfp_s** gets the floating-point control word. If *mask* is nonzero, a new value for the control word is set: For any bit that is on (equal to 1) in *mask*, the corresponding bit in *new* is used to update the control word. In other words, *fpcntrl* **=** ((*fpcntrl* **& ~mask**) | (**new & mask**)) where *fpcntrl* is the floating-point control word.

Note
The run-time libraries mask all floating-point exceptions by default.

_controlfp_s is nearly identical to the **_control87** function on Intel (x86) platforms and is supported by the MIPS and ALPHA platforms. To ensure that the floating-point code is portable to MIPS or ALPHA, use **_controlfp_s**. If you are targeting x86 platforms, use **_control87** or **_controlfp_s**.

The difference between **_control87** and **_controlfp_s** is the way these two functions treat **DENORMAL** values. For Intel (x86) platforms, **_control87** can set and clear the DENORMAL OPERAND exception mask. ALPHA platforms do not support this exception, and **_controlfp_s** does not modify the DENORMAL OPERAND exception mask. The following example demonstrates the difference:

```
_control87( _EM_INVALID, _MCW_EM );
// DENORMAL is unmasked by this call.
_controlfp( _EM_INVALID, _MCW_EM );
// DENORMAL exception mask remains unchanged.
```

The possible values for the mask constant (*mask*) and new control values (*newControl*) are shown in the following Hexadecimal Values table. Use the portable constants listed below (**_MCW_EM**, **_EM_INVALID**, and so on) as arguments to these functions, rather than supplying the hexadecimal values explicitly.

ALPHA platforms support the DENORMAL input and output values in software. The default behavior of Windows NT on these platforms is to flush the DENORMAL input and output values to zero. **_controlfp_s** provides a new mask to preserve and flush

the input and output DENORMAL values.

Intel (x86) platforms support the DENORMAL input and output values in hardware. The behavior is to preserve DENORMAL values. **_control87** does not provide a mask to change this behavior. The following example demonstrates this difference:

```
controlfp( _DN_SAVE, _MCW_DN);
// Denormal values preserved by software on ALPHA. NOP on x86
controlfp( _DN_FLUSH, _MCW_DN);
// Denormal values flushed to zero by hardware on ALPHA. Ignored on x86
```

_controlfp_s affects the control words for both the x87 and the SSE2, if present. It is possible for the two control words to be inconsistent with each other (because of a previous call to __control87_2, for example); if there is an inconsistency between the two control words, **_controlfp_s** sets the **EM_AMBIGUOUS** flag in *currentControl*. This is a warning that the returned control word might not represent the state of both floating-point control words accurately.

On the x64 architecture, changing the floating-point precision is not supported. If the precision control mask is used on that platform, the invalid parameter handler is invoked, as described in Parameter Validation.

In Visual C++ 2005, **_controlfp_s** has been optimized for better performance on all machine architectures.

If the mask is not set correctly, this function generates an invalid parameter exception, as described in Parameter Validation. If execution is allowed to continue, this function returns **EINVAL** and sets **errno** to **EINVAL**.

This function is deprecated when compiling with /clr (Common Language Runtime Compilation) or **/clr:pure** because the common language runtime only supports the default floating-point precision.

Hexadecimal Values

For the **_MCW_EM** mask, clearing the mask sets the exception, which allows the hardware exception; setting the mask hides the exception. Note that if a **_EM_UNDERFLOW** or **_EM_OVERFLOW** occurs, no hardware exception is thrown until the next floating-point instruction is executed. To generate a hardware exception immediately after **_EM_UNDERFLOW** or **_EM_OVERFLOW**, call the FWAIT MASM instruction.

Mask	Hex value	Constant	Hex value
_MCW_DN (Denormal control)	0x03000000	**_DN_SAVE** **_DN_FLUSH**	0x00000000 0x01000000
_MCW_EM (Interrupt exception mask)	0x0008001F	**_EM_INVALID** **_EM_DENORMAL** **_EM_ZERODIVIDE** **_EM_OVERFLOW** **_EM_UNDERFLOW** **_EM_INEXACT**	0x00000010 0x00080000 0x00000008 0x00000004 0x00000002 0x00000001
_MCW_IC (Infinity control)	0x00040000	**_IC_AFFINE** **_IC_PROJECTIVE**	0x00040000 0x00000000
_MCW_RC (Rounding control)	0x00000300	**_RC_CHOP** **_RC_UP** **_RC_DOWN** **_RC_NEAR**	0x00000300 0x00000200 0x00000100 0x00000000
_MCW_PC (Precision control)	0x00030000	**_PC_24** (24 bits) **_PC_53** (53 bits) **_PC_64** (64 bits)	0x00020000 0x00010000 0x00000000

Requirements

Routine	Required header	Compatibility
_controlfp_s	<float.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003; x86 processor only

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_contrlfp_s.c
// processor: x86
// This program uses _controlfp_s to output the FP control
// word, set the precision to 24 bits, and reset the status to
// the default.
//

#include <stdio.h>
#include <float.h>
#pragma fenv_access (on)

int main( void )
{
    double a = 0.1;
    unsigned int control_word;
    int err;

    // Show original FP control word and do calculation.
    err = _controlfp_s(&control_word, 0, 0);
    if ( err ) /* handle error here */;

    printf( "Original: 0x%.4x\n", control_word );
    printf( "%1.1f * %1.1f = %.15e\n", a, a, a * a );

    // Set precision to 24 bits and recalculate.
    err = _controlfp_s(&control_word, _PC_24, MCW_PC);
    if ( err ) /* handle error here */;

    printf( "24-bit:   0x%.4x\n", control_word );
    printf( "%1.1f * %1.1f = %.15e\n", a, a, a * a );

    // Restore default precision-control bits and recalculate.
    err = _controlfp_s(&control_word, _CW_DEFAULT, MCW_PC);
    if ( err ) /* handle error here */;

    printf( "Default:  0x%.4x\n", control_word );
    printf( "%1.1f * %1.1f = %.15e\n", a, a, a * a );
}
```

Output

```
Original: 0x9001f
0.1 * 0.1 = 1.000000000000000e-002
24-bit:   0xa001f
0.1 * 0.1 = 9.999999776482582e-003
Default:  0x9001f
0.1 * 0.1 = 1.000000000000000e-002
```

NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support

_clear87, _clearfp
_status87, _statusfp, _statusfp2

_copysign, _copysignl

Returns one value with the sign of another.

```
double _copysign(
   double x,
   double y
);
long double _copysignl(
   long double x,
   long double y
);
```

Parameters

x

Double-precision or long double-precision, floating-point value to be changed.

y

Double-precision or long double-precision, floating-point value.

Floating-Point Support Routines

Return Value

_copysign returns its double-precision, floating-point argument *x* with the same sign as its double-precision, floating-point argument *y*. **_copysignl** returns its long double-precision, floating-point argument *x* with the same sign as its long double-precision, floating-point argument *y*. There is no error return.

Requirements

Routine	Required header	Compatibility
_copysign	<float.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_copysignl	<math.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

fabs, fabsf

_chgsign, _chgsignl

cos, cosf, cosh, coshf

Calculate the cosine (**cos** or **cosf**) or hyperbolic cosine (**cosh** or **coshf**).

```
double cos(
   double x
);
float cos(
   float x
);  // C++ only
long double cos(
   long double x
);  // C++ only
float cosf(
   float x
);
double cosh(
   double x
);
float cosh(
   float x
);  // C++ only
long double cosh(
   long double x
);  // C++ only
float coshf(
   float x
);
```

Parameters

x

Angle in radians.

Return Value

The cosine or hyperbolic cosine of *x*. If *x* is greater than or equal to 263, or less than or equal to –263, a loss of significance in the result of a call to **cos** occurs.

If the result is too large in a **cosh** or **coshf** call, the function returns **HUGE_VAL** and sets **errno** to **ERANGE**, by default.

Input	SEH Exception	Matherr Exception
± **QNAN**,**IND**	none	**_DOMAIN**
± ∞ (**cosf**, **cos**)	**INVALID**	**_DOMAIN**
x ≥ 7.104760e+002 (**cosh**, **coshf**)	**INEXACT**+**OVERFLOW**	**OVERFLOW**

Remarks

C++ allows overloading, so you can call overloads of **cos** and **cosh**. In a C program, **cos** and **cosh** always take and return a double.

Requirements

Routine	Required header	Compatibility
cos, **cosh**, **cosf**, **coshf**	<math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example in sin, sinf, sinh, sinhf.

.NET Framework Equivalent

- System::Math::Cos
- System::Math::Cosh

See Also

Reference

Floating-Point Support
acos, acosf
asin, asinf
atan, atanf, atan2, atan2f
_matherr
sin, sinf, sinh, sinhf
tan, tanf, tanh, tanhf

_countof Macro

Compute the number of elements in a statically-allocated array.

```
_countof(
      array
);
```

Parameters

array

The name of an array.

Return Value

The number of elements in the array.

Remarks

Ensure that *array* is actually an array, not a pointer. In C, **_countof** will produce erroneous results if *array* is a pointer. In C++, **_countof** will fail to compile if *array* is a pointer.

Requirements

Macro	Required header	Compatibility
_countof	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

```
// crt_countof.cpp
#define _UNICODE
#include <stdio.h>
#include <stdlib.h>
#include <tchar.h>

int main( void )
{
   _TCHAR arr[20], *p;
   printf( "sizeof(arr) = %d bytes\n", sizeof(arr) );
   printf( "_countof(arr) = %d elements\n", _countof(arr) );
   // In C++, the following line would generate a compile-time error:
   // printf( "%d\n", _countof(p) ); // error C2784 (because p is a pointer)

   _tcscpy_s( arr, _countof(arr), _T("a string") );
   // unlike sizeof, _countof works here for both narrow- and wide-character strings
}
```

Output

```
sizeof(arr) = 40 bytes
_countof(arr) = 20 elements
```

See Also

Reference

sizeof Operator

cprintf

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _cprintf or security-enhanced _cprintf_s instead.

_cprintf, _cprintf_l, _cwprintf, _cwprintf_l

Formats and prints to the console. More secure versions are available; see _cprintf_s, _cprintf_s_l, _cwprintf_s, _cwprintf_s_l.

```
int _cprintf(
   const char * format [,
      argument] ...
);
int _cprintf_l(
   const char * format,
   locale_t locale [,
      argument] …
);
int _cwprintf(
   const wchar * format [,
      argument] …
);
int _cwprintf_l(
   const wchar * format,
   locale_t locale [,
      argument] …
);
```

Parameters

format

Format-control string.

argument

Optional parameters.

locale

The locale to use.

Return Value

The number of characters printed.

Remarks

These functions format and print a series of characters and values directly to the console, using the **_putch** function (**_putwch** for **_cwprintf**) to output characters. Each *argument* (if any) is converted and output according to the corresponding format specification in *format*. The format has the same form and function as the *format* parameter for the printf function. Unlike the **fprintf**, **printf**, and **sprintf** functions, neither **_cprintf** nor **_cwprintf** translates line-feed characters into carriage return–line feed (CR-LF) combinations when output.

An important distinction is that **_cwprintf** displays Unicode characters when used in Windows NT. Unlike **_cprintf**, **_cwprintf** uses the current console locale settings.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

_cprintf validates the *format* parameter. If *format* is a null pointer, the function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the function returns -1 and sets **errno** to **EINVAL**.

Security Note
Ensure that *format* is not a user-defined string.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tcprintf	**_cprintf**	**_cprintf**	**_cwprintf**

_tcprintf_l	_cprintf_l	_cprintf_l	_cwprintf_l

Requirements

Routine	Required header	Compatibility
_cprintf, **_cprintf_l**	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_cwprintf, **_cwprintf_l**	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_cprintf.c
// compile with: /c
// This program displays some variables to the console.

#include <conio.h>

int main( void )
{
    int         i = -16,
                h = 29;
    unsigned    u = 62511;
    char        c = 'A';
    char        s[] = "Test";

    // Note that console output does not translate \n as
    // standard output does. Use \r\n instead.
    //
    _cprintf( "%d  %.4x  %u  %c %s\r\n", i, h, u, c, s );
}
```

Sample Output

```
-16  001d  62511  A Test
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Console and Port I/O
_cscanf, _cscanf_l, _cwscanf, _cwscanf_l
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
vfprintf, _vfprintf_l, vfwprintf, _vfwprintf_l
_cprintf_s, _cprintf_s_l, _cwprintf_s, _cwprintf_s_l
_cprintf_p, _cprintf_p_l, _cwprintf_p, _cwprintf_p_l

_cprintf_p, _cprintf_p_l, _cwprintf_p, _cwprintf_p_l

Formats and prints to the console, with support for positional parameters in the format string.

```
int _cprintf_p(
   const char * format [,
      argument] ...
);
int _cprintf_p_l(
   const char * format,
   locale_t locale [,
      argument] ...
);
int _cwprintf_p(
   const wchar * format [,
      argument] ...
);
int _cwprintf_p_l(
   const wchar * format,
   locale_t locale [,
      argument] ...
);
```

Parameters

format

Format-control string.

argument

Optional parameters.

locale

The locale to use.

Return Value

The number of characters printed or a negative value if an error occurs.

Remarks

These functions format and print a series of characters and values directly to the console, using the **_putch** and **_putwch** functions to output characters. Each *argument* (if any) is converted and output according to the corresponding format specification in *format*. The format has the same form and function as the *format* parameter for the printf_p function. The difference between **_cprintf_p** and **cprintf_s** is that **_cprintf_p** supports positional parameters, which allows specifying the order in which the arguments are used in the format string. For more information, see printf_p Positional Parameters.

Unlike the **fprintf_p**, **printf_p**, and **sprintf_p** functions, neither **_cprintf_p** nor **_cwprintf_p** translates line-feed characters into carriage return–line feed (CR-LF) combinations when output. An important distinction is that **_cwprintf_p** displays Unicode characters when used in Windows NT. Unlike **_cprintf_p**, **_cwprintf_p** uses the current console locale settings.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

Security Note
Ensure that *format* is not a user-defined string.

Also, like **_cprintf_s** and **_cwprintf_s**, they validate the input pointer and the format string. If *format* or *argument* are **NULL**, or of the format string contains invalid formatting characters, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined

_tcprintf_p	**_cprintf_p**	**_cprintf_p**	**_cwprintf_p**
_tcprintf_p_l	**_cprintf_p_l**	**_cprintf_p_l**	**_cwprintf_p_l**

Requirements

Routine	Required header	Compatibility
_cprintf_p, **_cprintf_p_l**	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_cwprintf_p, **_cwprintf_p_l**	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_cprintf_p.c
// This program displays some variables to the console
// using the _cprintf_p function.

#include <conio.h>

int main( void )
{
    int         i = -16,
                h = 29;
    unsigned    u = 62511;
    char        c = 'A';
    char        s[] = "Test";

    // Note that console output does not translate
    // \n as standard output does. Use \r\n instead.
    _cprintf_p( "%2$d  %1$.4x  %3$u  %4$c %5$s\r\n",
                h, i, u, c, s );
}
```

Sample Output

```
-16  001d  62511  A Test
```

See Also

Reference

Console and Port I/O
_cscanf, _cscanf_l, _cwscanf, _cwscanf_l
_cscanf_s, _cscanf_s_l, _cwscanf_s, _cwscanf_s_l
_fprintf_p, _fprintf_p_l, _fwprintf_p, _fwprintf_p_l
fprintf, _fprintf_l, fwprintf, _fwprintf_l
_printf_p, _printf_p_l, _wprintf_p, _wprintf_p_l
printf, _printf_l, wprintf, _wprintf_l
_sprintf_p, _sprintf_p_l, _swprintf_p, _swprintf_p_l
_vfprintf_p, _vfprintf_p_l, _vfwprintf_p, _vfwprintf_p_l
_cprintf_s, _cprintf_s_l, _cwprintf_s, _cwprintf_s_l
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l
printf_p Positional Parameters
printf_p Positional Parameters

_cprintf_s, _cprintf_s_l, _cwprintf_s, _cwprintf_s_l

Formats and prints to the console. These are versions of _cprintf, _cprintf_l, _cwprintf, _cwprintf_l with security enhancements as described in Security Enhancements in the CRT.

```
int _cprintf_s(
   const char * format [,
   argument] ...
);
int _cprintf_s_l(
   const char * format,
   locale_t locale [,
   argument] ...
);
int _cwprintf_s(
   const wchar * format [,
   argument] ...
);
int _cwprintf_s_l(
   const wchar * format,
   locale_t locale [,
   argument] ...
);
```

Parameters

format

Format-control string.

argument

Optional parameters.

locale

The locale to use.

Return Value

The number of characters printed.

Remarks

These functions format and print a series of characters and values directly to the console, using the **_putch** function (**_putwch** for **_cwprintf_s**) to output characters. Each *argument* (if any) is converted and output according to the corresponding format specification in *format*. The format has the same form and function as the *format* parameter for the printf_s function. Unlike the **fprintf_s**, **printf_s**, and **sprintf_s** functions, neither **_cprintf_s** nor **_cwprintf_s** translates line-feed characters into carriage return–line feed (CR-LF) combinations when output.

An important distinction is that **_cwprintf_s** displays Unicode characters when used in Windows NT. Unlike **_cprintf_s**, **_cwprintf_s** uses the current console locale

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

Security Note
Ensure that *format* is not a user-defined string.

Like the non-secure versions (see _cprintf, _cprintf_l, _cwprintf, _cwprintf_l), these functions validate their parameters and invoke the invalid parameter handler, as described in Parameter Validation, if *format* is a null pointer. These functions differ from the non-secure versions in that the format string itself is also validated. If there are any unknown or badly formed formatting specifiers, these functions invoke the invalid parameter handler. In all cases, If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tcprintf_s	**_cprintf_s**	**_cprintf_s**	**_cwprintf_s**
_tcprintf_s_l	**_cprintf_s_l**	**_cprintf_s_l**	**_cwprintf_s_l**

Requirements

Routine	Required header	Compatibility
_cprintf_s, **_cprintf_s_l**	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_cwprintf_s, **_cwprintf_s_l**	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_cprintf_s.c
// compile with: /c
// This program displays some variables to the console.


#include <conio.h>

int main( void )
{
   int      i = -16, h = 29;
   unsigned u = 62511;
   char     c = 'A';
   char     s[] = "Test";

   /* Note that console output does not translate \n as
    * standard output does. Use \r\n instead.
    */
   _cprintf_s( "%d  %.4x  %u  %c %s\r\n", i, h, u, c, s );
}
```

Output

```
-16  001d  62511  A Test
```

See Also

Reference

Console and Port I/O
_cscanf, _cscanf_l, _cwscanf, _cwscanf_l
fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l
printf_s, _printf_s_l, wprintf_s, _wprintf_s_l
sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l
vfprintf_s, _vfprintf_s_l, vfwprintf_s, _vfwprintf_s_l

cputs

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _cputs instead.

_cputs, _cputws

Puts a string to the console.

```
int _cputs(
   const char *str
);
int _cputws(
   const wchar_t *str
);
```

Parameters

str

Output string.

Return Value

If successful, **_cputs** returns 0. If the function fails, it returns a nonzero value.

Remarks

The **_cputs** function writes the null-terminated string pointed to by *str* directly to the console. A carriage return–line feed (CR-LF) combination is not automatically appended to the string.

This function validates its parameter. If *str* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and -1 is returned.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_cputts	**_cputs**	**_cputs**	**_cputws**

Requirements

Routine	Required header	Optional header	Compatibility
_cputs	<conio.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_cputws	<conio.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_cputs.c
// compile with: /c
// This program first displays a string to the console.


#include <conio.h>
#include <errno.h>

void print_to_console(char* buffer)
{
   int retval;
```

```
    retval = _cputs( buffer );
    if (retval)
    {
        if (errno == EINVAL)
        {
          _cputs( "Invalid buffer in print_to_console.\r\n");
        }
        else
          _cputs( "Unexpected error in print_to_console.\r\n");
    }
}


void wprint_to_console(wchar_t* wbuffer)
{
    int retval;
    retval = _cputws( wbuffer );
    if (retval)
    {
        if (errno == EINVAL)
        {
          _cputws( L"Invalid buffer in wprint_to_console.\r\n");
        }
        else
          _cputws( L"Unexpected error in wprint_to_console.\r\n");
    }
}

int main()
{

    // String to print at console.
    // Note the \r (return) character.
    char* buffer = "Hello world (courtesy of _cputs)!\r\n";
    wchar_t *wbuffer = L"Hello world (courtesy of _cputws)!\r\n";
    print_to_console(buffer);
    wprint_to_console( wbuffer );
}
```

Output

```
Hello world (courtesy of _cputs)!
Hello world (courtesy of _cputws)!
```

See Also

Reference

Console and Port I/O

_putch, _putwch

creat

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _creat instead.

_creat, _wcreat

Creates a new file. **_creat** and **_wcreat** have been deprecated in Visual C++ 2005; use _sopen_s, _wsopen_s instead.

```
int _creat(
   const char *filename,
   int pmode
);
int _wcreat(
   const wchar_t *filename,
   int pmode
);
```

Parameters

filename

Name of new file.

pmode

Permission setting.

Return Value

These functions, if successful, return a file descriptor to the created file. Otherwise, the functions return –1 and set **errno** as shown in the following table.

errno setting	Description
EACCES	*filename* specifies an existing read-only file or specifies a directory instead of a file.
EMFILE	No more file descriptors are available.
ENOENT	Specified file could not be found.

If *filename* is NULL, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_creat** function creates a new file or opens and truncates an existing one. **_wcreat** is a wide-character version of **_creat**; the *filename* argument to **_wcreat** is a wide-character string. **_wcreat** and **_creat** behave identically otherwise.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tcreat	**_creat**	**_creat**	**_wcreat**

If the file specified by *filename* does not exist, a new file is created with the given permission setting and is opened for writing. If the file already exists and its permission setting allows writing, **_creat** truncates the file to length 0, destroying the previous contents, and opens it for writing. The permission setting, *pmode*, applies to newly created files only. The new file receives the specified permission setting after it is closed for the first time. The integer expression *pmode* contains one or both of the manifest constants **_S_IWRITE** and **_S_IREAD**, defined in SYS\Stat.h. When both constants are given, they are joined with the bitwise **OR** operator (**|**). The *pmode* parameter is set to one of the following values.

Value	Definition
_S_IWRITE	Writing permitted.
_S_IREAD	Reading permitted.

_S_IREAD | _S_IWRITE	Reading and writing permitted.

If write permission is not given, the file is read-only. All files are always readable; it is impossible to give write-only permission. The modes **_S_IWRITE** and **_S_IREAD | _S_IWRITE** are then equivalent. Files opened using **_creat** are always opened in compatibility mode (see _sopen) with **_SH_DENYNO**.

_creat applies the current file-permission mask to *pmode* before setting the permissions (see _umask). **_creat** is provided primarily for compatibility with previous libraries. A call to **_open** with **_O_CREAT** and **_O_TRUNC** in the *oflag* parameter is equivalent to **_creat** and is preferable for new code.

Requirements

Routine	Required header	Optional header	Compatibility
_creat	<io.h>	<sys/types.h>, <sys/stat.h>, <errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcreat	<io.h> or <wchar.h>	<sys/types.h>, <sys/stat.h>, <errno.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_creat.c
// compile with: /W3
// This program uses _creat to create
// the file (or truncate the existing file)
// named data and open it for writing.

#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   int fh;

   fh = _creat( "data", _S_IREAD | _S_IWRITE ); // C4996
   // Note: _creat is deprecated; use _sopen_s instead
   if( fh == -1 )
      perror( "Couldn't create data file" );
   else
   {
      printf( "Created data file.\n" );
      _close( fh );
   }
}
```

Output

```
Created data file.
```

See Also

Reference

Low-Level I/O
_chmod, _wchmod
_chsize
_close

_dup, _dup2
_open, _wopen
_sopen, _wsopen
_umask

_create_locale

Creates a locale object.

```
_locale_t _create_locale(
   int category,
   const char *locale
);
```

Parameters

category

Category.

locale

Locale name.

Return Value

If a valid locale and category are given, returns the specified locale settings as a **_locale_t** object. The current locale settings of the program are not changed.

Remarks

The **_create_locale** function allows you to create an object that represents certain region-specific settings, for use in locale-specific versions of many CRT functions (functions with the **_l** suffix). The behavior is similar to **setlocale**, except that instead of applying the specified locale settings to the current environment, the settings are saved in a **_locale_t** structure that is returned.

The *category* argument specifies the parts of the locale-specific behavior that are affected. The flags used for *category* and the parts of the program they affect are as shown in the following table.

LC_ALL

All categories, as listed below.

LC_COLLATE

The **strcoll**, **_stricoll**, **wcscoll**, **_wcsicoll**, **strxfrm**, **_strncoll**, **_strnicoll**, **_wcsncoll**, **_wcsnicoll**, and **wcsxfrm** functions.

LC_CTYPE

The character-handling functions (except **isdigit**, **isxdigit**, **mbstowcs**, and **mbtowc**, which are unaffected).

LC_MONETARY

Monetary-formatting information returned by the **localeconv** function.

LC_NUMERIC

Decimal-point character for the formatted output routines (such as **printf**), for the data-conversion routines, and for the non-monetary formatting information returned by **localeconv**. In addition to the decimal-point character, **LC_NUMERIC** sets the thousands separator and the grouping control string returned by localeconv.

LC_TIME

The **strftime** and **wcsftime** functions.

This function validates the *category* and *locale* parameters. If the category parameter is not one of the values given in the previous table or if *locale* is **NULL**, the function returns **NULL**.

The *locale* argument is a pointer to a string that specifies the name of the locale. If *locale* points to an empty string, the locale is the implementation-defined native environment. A value of *C* specifies the minimal ANSI conforming environment for C translation. The *C* locale assumes that all **char** data types are 1 byte and that their value is always less than 256. The *locale* argument takes the following form:

```
locale :: "lang[_country_region[.code_page]]"
```

```
                | ".code_page"
                | ""
                | NULL
```

The set of available languages, country/region codes, and code pages includes all those supported by the Win32 NLS API. The set of language and country/region codes supported by **_create_locale** is listed in Language and Country/Region Strings. For more information about locale settings, see setlocale, _wsetlocale.

The previous name of this function, **__create_locale** (with two leading underscores), has been deprecated.

Requirements

Routine	Required header	Compatibility
_create_locale	<locale.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_create_locale.c
// Sets the current locale to "Germany" using the
// setlocale function and demonstrates its effect on the strftime
// function.

#include <stdio.h>
#include <locale.h>
#include <time.h>

int main(void)
{
        time_t ltime;
        struct tm thetime;
        unsigned char str[100];
        _locale_t locale;

        // Create a locale object representing the German locale
        locale = _create_locale(LC_ALL, "German");
        time (&ltime);
        _gmtime64_s(&thetime, &ltime);

        // %#x is the long date representation, appropriate to
        // the current locale
        //
        if (!_strftime_l((char *)str, 100, "%#x",
                      (const struct tm *)&thetime, locale))
                printf("_strftime_l failed!\n");
        else
                printf("In German locale, _strftime_l returns '%s'\n",
                       str);

        // Create a locale object representing the default C locale
        locale = _create_locale(LC_ALL, "C");
        time (&ltime);
        _gmtime64_s(&thetime, &ltime);

        if (!_strftime_l((char *)str, 100, "%#x",
                      (const struct tm *)&thetime, locale))
                printf("_strftime_l failed!\n");
        else
                printf("In 'C' locale, _strftime_l returns '%s'\n",
                       str);
}
```

Sample Output

```
In German locale, _strftime_l returns 'Samstag, 9. Februar 2002'
In 'C' locale, _strftime_l returns 'Saturday, February 09, 2002'
```

.NET Framework Equivalent

System::Globalization::CultureInfo Class

See Also

Reference

_configthreadlocale
setlocale
Locale
localeconv
_mbclen, mblen, _mblen_l
strlen, strlen_l, wcslen, wcslen_l, _mbslen, _mbslen_l, _mbstrlen, _mbstrlen_l
mbstowcs, _mbstowcs_l
mbtowc, _mbtowc_l
_setmbcp
strcoll Functions
strftime, wcsftime, _strftime_l, _wcsftime_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l
wcstombs, _wcstombs_l
wctomb, _wctomb_l

_CrtCheckMemory

Confirms the integrity of the memory blocks allocated in the debug heap (debug version only).

```
int _CrtCheckMemory( void );
```

Return Value

If successful, **_CrtCheckMemory** returns TRUE; otherwise, the function returns FALSE.

Remarks

The **_CrtCheckMemory** function validates memory allocated by the debug heap manager by verifying the underlying base heap and inspecting every memory block. If an error or memory inconsistency is encountered in the underlying base heap, the debug header information, or the overwrite buffers, **_CrtCheckMemory** generates a debug report with information describing the error condition. When _DEBUG is not defined, calls to **_CrtCheckMemory** are removed during preprocessing.

The behavior of **_CrtCheckMemory** can be controlled by setting the bit fields of the _crtDbgFlag flag using the _CrtSetDbgFlag function. Turning the **_CRTDBG_CHECK_ALWAYS_DF** bit field ON results in **_CrtCheckMemory** being called every time a memory allocation operation is requested. Although this method slows down execution, it is useful for catching errors quickly. Turning the **_CRTDBG_ALLOC_MEM_DF** bit field OFF causes **_CrtCheckMemory** to not verify the heap and immediately return **TRUE**.

Because this function returns **TRUE** or **FALSE**, it can be passed to one of the _ASSERT macros to create a simple debugging error handling mechanism. The following example causes an assertion failure if corruption is detected in the heap:

```
_ASSERTE( _CrtCheckMemory( ) );
```

For more information about how **_CrtCheckMemory** can be used with other debug functions, see Heap State Reporting Functions. For an overview of memory management and the debug heap, see Memory Management and the Debug Heap.

Requirements

Routine	Required header	Compatibility
_CrtCheckMemory	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

For a sample of how to use **_CrtCheckMemory**, see crt_dbg1.

.NET Framework Equivalent

System::Diagnostics::PerformanceCounter

See Also

Reference

Debug Routines
_crtDbgFlag
_CrtSetDbgFlag

_CrtDbgBreak

Sets a break point on a particular line of code. (Used in debug mode only.)

```
void _CrtDbgBreak( void );
```

Return Value

There is no return value.

Remarks

The **_CrtDbgBreak** function sets a debug break point on the particular line of code where the function resides. The functions is only used in debug mode only and is dependant on **_DEBUG** begin previous define.

For more information about using other hook-capable run-time functions and writing your own client-defined hook functions, see Writing Your Own Debug Hook Functions.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

Requirements

Routine	Required header	Compatibility
_CrtDbgBreak	<CRTDBG.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003. For more information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

See Also

Reference

Debug Routines

_CrtDbgReport, _CrtDbgReportW

Generates a report with a debugging message and sends the report to three possible destinations (debug version only).

```
int _CrtDbgReport(
   int reportType,
   const char *filename,
   int linenumber,
   const char *moduleName,
   const char *format [,
   argument] ...
);
int _CrtDbgReportW(
   int reportType,
   const wchar_t *filename,
   int linenumber,
   const wchar_t *moduleName,
   const wchar_t *format [,
   argument] ...
);
```

Parameters

reportType

Report type: **_CRT_WARN**, **_CRT_ERROR**, and **_CRT_ASSERT**.

filename

Pointer to name of source file where assert/report occurred or **NULL**.

linenumber

Line number in source file where assert/report occurred or **NULL**.

moduleName

Pointer to name of module (.exe or .dll) where assert/report occurred.

format

Pointer to format-control string used to create the user message.

argument

Optional substitution arguments used by *format*.

Return Value

For all report destinations, **_CrtDbgReport** and **_CrtDbgReportW** return –1 if an error occurs and 0 if no errors are encountered. However, when the report destination is a debug message window and the user clicks the **Retry** button, these functions return 1. If the user clicks the **Abort** button in the Debug Message window, these functions immediately abort and do not return a value.

The _RPT, _RPTF debug macros call **_CrtDbgReport** to generate their debug reports. The wide-character versions of these macros as well as _ASSERT[E], **_RPTWn** and **_RPTFWn**, use **_CrtDbgReportW** to generate their debug reports. When **_CrtDbgReport** or **_CrtDbgReportW** return 1, these macros start the debugger, provided that just-in-time (JIT) debugging is enabled.

Remarks

_CrtDbgReport and **_CrtDbgReportW** can send the debug report to three different destinations: a debug report file, a debug monitor (the Visual Studio debugger), or a debug message window. Two configuration functions, _CrtSetReportMode and _CrtSetReportFile, are used to specify the destination or destinations for each report type. These functions allow the reporting destination or destinations for each report type to be separately controlled. For example, it is possible to specify that a *reportType* of **_CRT_WARN** only be sent to the debug monitor, while a *reportType* of **_CRT_ASSERT** be sent to a debug message window and a user-defined report file.

In Visual C++ 2005, **_CrtDbgReportW** is the wide-character version of **_CrtDbgReport**. All its output and string parameters are in wide-character strings; otherwise it is identical to the single-byte character version.

_CrtDbgReport and **_CrtDbgReportW** create the user message for the debug report by substituting the *argument*[*n*] arguments into the *format* string, using the same rules defined by the **printf** or **wprintf** functions. These functions then generate the debug report and determine the destination or destinations, based on the current report modes and file defined for *reportType*. When the report is sent to a debug message window, the *filename*, *lineNumber*, and *moduleName* are included in the information displayed in the window.

The following table lists the available choices for the report mode or modes and file and the resulting behavior of **_CrtDbgReport** and **_CrtDbgReportW**. These options are defined as bit flags in Crtdbg.h.

Report mode	Report file	_CrtDbgReport, _CrtDbgReportW behavior
_CRTDBG_MODE_DEBUG	Not applicable	Writes message to Windows OutputDebugString API.
_CRTDBG_MODE_WNDW	Not applicable	Calls Windows MessageBox API to create message box to display the message along with **Abort**, **Retry**, and **Ignore** buttons. If a user clicks **Abort**, **_CrtDbgReport** or **_CrtDbgReport** immediately aborts. If a user clicks **Retry**, it returns 1. If a user clicks **Ignore**, execution continues and **_CrtDbgReport** and **_CrtDbgReportW** return 0. Note that clicking **Ignore** when an error condition exists often results in "undefined behavior."
_CRTDBG_MODE_FILE	**__HFILE**	Writes message to user-supplied **HANDLE**, using the Windows WriteFile API and does not verify validity of file handle; the application is responsible for opening the report file and passing a valid file handle.
_CRTDBG_MODE_FILE	**_CRTDBG_FILE_STDERR**	Writes message to **stderr**.
_CRTDBG_MODE_FILE	**_CRTDBG_FILE_STDOUT**	Writes message to **stdout**.

The report can be sent to one, two, or three destinations or to no destination at all. For more information about specifying the report mode or modes and report file, see the _CrtSetReportMode and _CrtSetReportFile functions. For more information about using the debug macros and reporting functions, see Using Macros for Verification and Reporting.

If your application needs more flexibility than that provided by **_CrtDbgReport** and **_CrtDbgReportW**, you can write your own reporting function and hook it into the C run-time library reporting mechanism by using the _CrtSetReportHook function.

Requirements

Routine	Required header	Compatibility
_CrtDbgReport	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_CrtDbgReportW	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

```
// crt_crtdbgreport.c
#include <crtdbg.h>

int main() {
#ifdef _DEBUG
   CrtDbgReport(_CRT_ASSERT, NULL, NULL, "some module", NULL);
#endif
}
```

See crt_dbg2 for an example of how to change the report function.

.NET Framework Equivalent

- System::Diagnostics::Debug::Write
- System::Diagnostics::Debug::Writeline
- System::Diagnostics::Debug::WriteIf
- System::Diagnostics::Debug::WriteLineIf

See Also

Reference

Debug Routines
_CrtSetReportMode
_CrtSetReportFile
printf, _printf_l, wprintf, _wprintf_l
_DEBUG

_CrtDoForAllClientObjects

Calls an application-supplied function for all **_CLIENT_BLOCK** types in the heap (debug version only).

```
void _CrtDoForAllClientObjects(
   void ( * pfn )( void *, void * ),
   void *context
);
```

Parameters

pfn

Pointer to the application-supplied function callback function. The first parameter to this function points to the data. The second parameter is the context pointer that is passed to the call to **_CrtDoForAllClientObjects**.

context

Pointer to the application-supplied context to pass to the application-supplied function.

Remarks

The **_CrtDoForAllClientObjects** function searches the heap's linked list for memory blocks with the **_CLIENT_BLOCK** type and calls the application-supplied function when a block of this type is found. The found block and the *context* parameter are passed as arguments to the application-supplied function. During debugging, an application can track a specific group of allocations by explicitly calling the debug heap functions to allocate the memory and specifying that the blocks be assigned the **_CLIENT_BLOCK** block type. These blocks can then be tracked separately and reported on differently during leak detection and memory state reporting.

If the **_CRTDBG_ALLOC_MEM_DF** bit field of the _crtDbgFlag flag is not turned on, **_CrtDoForAllClientObjects** immediately returns. When _DEBUG is not defined, calls to **_CrtDoForAllClientObjects** are removed during preprocessing.

For more information about the **_CLIENT_BLOCK** type and how it can be used by other debug functions, see Types of Blocks on the Debug Heap. For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see Memory Management and the Debug Heap.

If *pfn* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, errno, _doserrno, _sys_errlist, and _sys_nerr is set to **EINVAL** and the function returns.

Requirements

Routine	Required header	Compatibility
_CrtDoForAllClientObjects	<crtdbg.h>, <errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries: Debug versions of C Run-Time Libraries only.

Example

See dfacobjs Sample: C Run-Time _CrtDoForAllClientObjects Function.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines
_CrtSetDbgFlag
_CrtReportBlockType

Concepts

Heap State Reporting Functions

_CrtDumpMemoryLeaks

Dumps all the memory blocks in the debug heap when a memory leak has occurred (debug version only).

```
int _CrtDumpMemoryLeaks( void );
```

Return Value

_CrtDumpMemoryLeaks returns TRUE if a memory leak is found. Otherwise, the function returns FALSE.

Remarks

The **_CrtDumpMemoryLeaks** function determines whether a memory leak has occurred since the start of program execution. When a leak is found, the debug header information for all the objects in the heap is dumped in a user-readable form. When _DEBUG is not defined, calls to **_CrtDumpMemoryLeaks** are removed during preprocessing.

_CrtDumpMemoryLeaks is frequently called at the end of program execution to verify that all memory allocated by the application has been freed. The function can be called automatically at program termination by turning on the **_CRTDBG_LEAK_CHECK_DF** bit field of the _crtDbgFlag flag using the _CrtSetDbgFlag function.

_CrtDumpMemoryLeaks calls _CrtMemCheckpoint to obtain the current state of the heap and then scans the state for blocks that have not been freed. When an unfreed block is encountered, **_CrtDumpMemoryLeaks** calls _CrtMemDumpAllObjectsSince to dump information for all the objects allocated in the heap from the start of program execution.

By default, internal C run-time blocks (**_CRT_BLOCK**) are not included in memory dump operations. The _CrtSetDbgFlag function can be used to turn on the **_CRTDBG_CHECK_CRT_DF** bit of **_crtDbgFlag** to include these blocks in the leak detection process.

For more information about heap state functions and the **_CrtMemState** structure, see Heap State Reporting Functions. For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see Memory Management and the Debug Heap.

Requirements

Routine	Required header	Compatibility
_CrtDumpMemoryLeaks	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

For a sample of how to use **_CrtDumpMemoryLeaks**, see crt_dbg1.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_CrtGetAllocHook

Retrieves the current client-defined allocation function for hooking into the C run-time debug memory allocation process (debug version only).

```
_CRT_ALLOC_HOOK _CrtGetAllocHook( void );
```

Return Value

Returns the currently defined allocation hook function.

Remarks

_CrtGetAllocHook retrieves the current client-defined application hook function for the C run-time debug library memory allocation process.

For more information about using other hook-capable run-time functions and writing your own client-defined hook functions, see Writing Your Own Debug Hook Functions.

Requirements

Routine	Required header	Compatibility
_CrtGetAllocHook	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_CrtSetAllocHook

_CrtGetDumpClient

Retrieves the current application-defined function for dumping the **_CLIENT_BLOCK** type memory blocks (debug version only).

```
_CRT_DUMP_CLIENT _CrtGetDumpClient( void );
```

Return Value

Returns the current dump routine.

Remarks

The **_CrtGetDumpClient** function retrieves the current hook function for dumping objects stored in the **_CLIENT_BLOCK** memory blocks for the C run-time debug memory dump process.

For more information about using other hook-capable run-time functions and writing your own client-defined hook functions, see Writing Your Own Debug Hook Functions.

Requirements

Routine	Required header	Compatibility
_CrtGetDumpClient	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines
_CrtReportBlockType
_CrtSetDumpClient

_CrtGetReportHook

Retrieves the client-defined reporting function for hooking it into the C run time for the debug reporting process (debug version only).

```
_CRT_REPORT_HOOK _CrtGetReportHook( void );
```

Return Value

Returns the current client-defined reporting function.

Remarks

_CrtGetReportHook allows an application to retrieve the current reporting function for the C run-time debug library reporting process.

For more information about using other hook-capable run-time functions and writing your own client-defined hook functions, see Writing Your Own Debug Hook Functions.

Requirements

Routine	Required header	Compatibility
_CrtGetReportHook	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

For a sample of how to use **_CrtSetReportHook**, see report.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_CrtSetReportHook

_CrtIsMemoryBlock

Verifies that a specified memory block is in the local heap and that it has a valid debug heap block type identifier (debug version only).

```
int _CrtIsMemoryBlock(
   const void *userData,
   unsigned int size,
   long *requestNumber,
   char **filename,
   int *linenumber
);
```

Parameters

userData

Pointer to the beginning of the memory block to verify.

size

Size of the specified block (in bytes).

requestNumber

Pointer to the allocation number of the block or **NULL**.

filename

Pointer to name of the source file that requested the block or **NULL**.

linenumber

Pointer to the line number in the source file or **NULL**.

Return Value

_CrtIsMemoryBlock returns TRUE if the specified memory block is located within the local heap and has a valid debug heap block type identifier; otherwise, the function returns FALSE.

Remarks

The **_CrtIsMemoryBlock** function verifies that a specified memory block is located within the application's local heap and that it has a valid block type identifier. This function can also be used to obtain the object allocation order number and the source file name/line number where the memory block allocation was originally requested. Passing non-NULL values for the *requestNumber*, *filename*, or *linenumber* parameters causes **_CrtIsMemoryBlock** to set these parameters to the values in the memory block's debug header, if it finds the block in the local heap. When _DEBUG is not defined, calls to **_CrtIsMemoryBlock** are removed during preprocessing.

Because this function returns TRUE or FALSE, it can be passed to one of the _ASSERT macros to create a simple debugging error handling mechanism. The following example causes an assertion failure if the specified address is not located within the local heap:

```
_ASSERTE( _CrtIsMemoryBlock( userData, size, &requestNumber,
&filename, &linenumber ) );
```

For more information about how **_CrtIsMemoryBlock** can be used with other debug functions and macros, see Using Macros for Verification and Reporting. For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see Memory Management and the Debug Heap.

Requirements

Routine	Required header	Compatibility

_CrtIsMemoryBlock	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

See the example for the _CrtIsValidHeapPointer topic.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_CrtIsValidHeapPointer

Verifies that a specified pointer is in the local heap (debug version only).

```
int _CrtIsValidHeapPointer(
   const void *userData
);
```

Parameters

userData

Pointer to the beginning of an allocated memory block.

Return Value

_CrtIsValidHeapPointer returns TRUE if the specified pointer is in the local heap. Otherwise, the function returns FALSE.

Remarks

The **_CrtIsValidHeapPointer** function is used to ensure that a specific memory address is within the local heap. The local heap refers to the heap created and managed by a particular instance of the C run-time library. If a dynamic-link library (DLL) contains a static link to the run-time library, it has its own instance of the run-time heap, and therefore its own heap, independent of the application's local heap. When _DEBUG is not defined, calls to **_CrtIsValidHeapPointer** are removed during preprocessing.

Because this function returns TRUE or FALSE, it can be passed to one of the _ASSERT macros to create a simple debugging error handling mechanism. The following example causes an assertion failure if the specified address is not located within the local heap:

```
_ASSERTE( _CrtIsValidHeapPointer( userData ) );
```

For more information about how **_CrtIsValidHeapPointer** can be used with other debug functions and macros, see Using Macros for Verification and Reporting. For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see Memory Management and the Debug Heap.

Requirements

Routine	Required header	Compatibility
_CrtIsValidHeapPointer	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

```
// crt_isvalid.c
/*
 * This program allocates a block of memory using _malloc_dbg
 * and then tests the validity of this memory by calling
 * _CrtIsMemoryBlock,_CrtIsValidPointer, and _CrtIsValidHeapPointer.
 */

#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <crtdbg.h>
```

```
#define  TRUE   1
#define  FALSE  0

int main( void )
{
        char *my_pointer;

        /*
         * Call _malloc_dbg to include the filename and line number
         * of our allocation request in the header information
         */
        my_pointer = (char *)_malloc_dbg( sizeof(char) * 10,
        _NORMAL_BLOCK, __FILE__, __LINE__ );

        // Ensure that the memory got allocated correctly
        _CrtIsMemoryBlock((const void *)my_pointer, sizeof(char) * 10,
        NULL, NULL, NULL );

         // Test for read/write accessibility
        if (_CrtIsValidPointer((const void *)my_pointer,
        sizeof(char) * 10, TRUE))
                printf("my_pointer has read and write accessibility.\n");
        else
                printf("my_pointer only has read access.\n");

        // Make sure my_pointer is within the local heap
        if (_CrtIsValidHeapPointer((const void *)my_pointer))
                printf("my_pointer is within the local heap.\n");
        else
                printf("my_pointer is not located within the local"
                       " heap.\n");

        free(my_pointer);
}
```

Output

```
my_pointer has read and write accessibility.
my_pointer is within the local heap.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_CrtIsValidPointer

Verifies that a specified memory range is valid for reading and writing (debug version only).

```
int _CrtIsValidPointer(
   const void *address,
   unsigned int size,
   int access
);
```

Parameters

address

Points to the beginning of the memory range to test for validity.

size

Size of the specified memory range (in bytes).

access

Read/write accessibility to determine for the memory range.

Return Value

_CrtIsValidPointer returns TRUE if the specified memory range is valid for the specified operation or operations. Otherwise, the function returns FALSE.

Remarks

The **_CrtIsValidPointer** function verifies that the memory range beginning at *address* and extending for *size* bytes is valid for the specified accessibility operation or operations. When *access* is set to TRUE, the memory range is verified for both reading and writing. When *address* is FALSE, the memory range is only validated for reading. When _DEBUG is not defined, calls to **_CrtIsValidPointer** are removed during preprocessing.

Because this function returns TRUE or FALSE, it can be passed to one of the _ASSERT macros to create a simple debugging error handling mechanism. The following example causes an assertion failure if the memory range is not valid for both reading and writing operations:

```
_ASSERTE( _CrtIsValidPointer( address, size, TRUE ) );
```

For more information about how **_CrtIsValidPointer** can be used with other debug functions and macros, see Using Macros for Verification and Reporting. For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see Memory Management and the Debug Heap.

Requirements

Routine	Required header	Compatibility
_CrtIsValidPointer	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

See the example for the _CrtIsValidHeapPointer topic.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_CrtMemCheckpoint

Obtains the current state of the debug heap and stores in an application-supplied **_CrtMemState** structure (debug version only).

```
void _CrtMemCheckpoint(
   _CrtMemState *state
);
```

Parameters

state

Pointer to **_CrtMemState** structure to fill with the memory checkpoint.

Remarks

The **_CrtMemCheckpoint** function creates a snapshot of the current state of the debug heap at any given moment. This snapshot can be used by other heap state functions to help detect memory leaks and other problems. When _DEBUG is not defined, calls to **_CrtMemState** are removed during preprocessing.

The application must pass a pointer to a previously allocated instance of the **_CrtMemState** structure, defined in Crtdbg.h, in the *state* parameter. If **_CrtMemCheckpoint** encounters an error during the checkpoint creation, the function generates a **_CRT_WARN** debug report describing the problem.

For more information about heap state functions and the **_CrtMemState** structure, see Heap State Reporting Functions. For more information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see Memory Management and the Debug Heap.

If *state* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, errno, _doserrno, _sys_errlist, and _sys_nerr is set to **EINVAL** and the function returns.

Requirements

Routine	Required header	Compatibility
_CrtMemCheckpoint	<crtdbg.h>, <errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries: Debug versions of C Run-Time Libraries only.

Example

See crt_dbg1 Sample: C Run-Time Basic Debugging Features.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_CrtMemDifference

Compares two memory states and returns their differences (debug version only).

```
int _CrtMemDifference(
   _CrtMemState *stateDiff,
   const _CrtMemState *oldState,
   const _CrtMemState *newState
);
```

Parameters

stateDiff

Pointer to a **_CrtMemState** structure that is used to store the differences between the two memory states (returned).

oldState

Pointer to an earlier memory state (**_CrtMemState** structure).

newState

Pointer to a later memory state (**_CrtMemState** structure).

Return Value

If the memory states are significantly different, **_CrtMemDifference** returns TRUE. Otherwise, the function returns FALSE.

Remarks

The **_CrtMemDifference** function compares *oldState* and *newState* and stores their differences in *stateDiff*, which can then be used by the application to detect memory leaks and other memory problems. When _DEBUG is not defined, calls to **_CrtMemDifference** are removed during preprocessing.

newState and *oldState* must each be a valid pointer to a **_CrtMemState** structure, defined in Crtdbg.h, that has been filled in by _CrtMemCheckpoint before calling **_CrtMemDifference**. *stateDiff* must be a pointer to a previously allocated instance of the **_CrtMemState** structure. If *stateDiff*, *newState*, or *oldState* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, errno, _doserrno, _sys_errlist, and _sys_nerr is set to **EINVAL** and the function returns FALSE.

_CrtMemDifference compares the **_CrtMemState** field values of the blocks in *oldState* to those in *newState* and stores the result in *stateDiff*. When the number of allocated block types or total number of allocated blocks for each type differs between the two memory states, the states are said to be significantly different. The difference between the largest amount ever allocated at once for the two states and the difference between total allocations for the two states are also stored in *stateDiff*.

By default, internal C run-time blocks (**_CRT_BLOCK**) are not included in memory state operations. The _CrtSetDbgFlag function can be used to turn on the **_CRTDBG_CHECK_CRT_DF** bit of **_crtDbgFlag** to include these blocks in leak detection and other memory state operations. Freed memory blocks (**_FREE_BLOCK**) do not cause **_CrtMemDifference** to return TRUE.

For more information about heap state functions and the **_CrtMemState** structure, see Heap State Reporting Functions. For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see Memory Management and the Debug Heap.

Requirements

Routine	Required header	Optional header	Compatibility
_CrtMemDifference	<crtdbg.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries: Debug versions of C Run-Time Libraries only.

Example

See crt_dbg1 Sample: C Run-Time Basic Debugging Features.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_crtDbgFlag

_CrtMemDumpAllObjectsSince

Dumps information about objects in the heap from the start of program execution or from a specified heap state (debug version only).

```
void _CrtMemDumpAllObjectsSince(
   const _CrtMemState *state
);
```

Parameters

state

Pointer to the heap state to begin dumping from or **NULL**.

Remarks

The **_CrtMemDumpAllObjectsSince** function dumps the debug header information of objects allocated in the heap in a user-readable form. The dump information can be used by the application to track allocations and detect memory problems. When _DEBUG is not defined, calls to **_CrtMemDumpAllObjectsSince** are removed during preprocessing.

_CrtMemDumpAllObjectsSince uses the value of the *state* parameter to determine where to initiate the dump operation. To begin dumping from a specified heap state, the *state* parameter must be a pointer to a **_CrtMemState** structure that has been filled in by _CrtMemCheckpoint before **_CrtMemDumpAllObjectsSince** was called. When *state* is **NULL**, the function begins the dump from the start of program execution.

If the application has installed a dump hook function by calling _CrtSetDumpClient, then every time **_CrtMemDumpAllObjectsSince** dumps information about a **_CLIENT_BLOCK** type of block, it calls the application-supplied dump function as well. By default, internal C run-time blocks (**_CRT_BLOCK**) are not included in memory dump operations. The _CrtSetDbgFlag function can be used to turn on the **_CRTDBG_CHECK_CRT_DF** bit of **_crtDbgFlag** to include these blocks. In addition, blocks marked as freed or ignored (**_FREE_BLOCK**, **_IGNORE_BLOCK**) are not included in the memory dump.

For more information about heap state functions and the **_CrtMemState** structure, see Heap State Reporting Functions. For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see Memory Management and the Debug Heap.

Requirements

Routine	Required header	Compatibility
_CrtMemDumpAll-ObjectsSince	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

For a sample of how to use **_CrtMemDumpAllObjectsSince**, see crt_dbg2.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_crtDbgFlag

_CrtMemDumpStatistics

Dumps the debug header information for a specified heap state in a user-readable form (debug version only).

```
void _CrtMemDumpStatistics(
   const _CrtMemState *state
);
```

Parameters
state

Pointer to the heap state to dump.

Remarks

The **_CrtMemDumpStatistics** function dumps the debug header information for a specified state of the heap in a user-readable form. The dump statistics can be used by the application to track allocations and detect memory problems. The memory state can contain a specific heap state or the difference between two states. When _DEBUG is not defined, calls to **_CrtMemDumpStatistics** are removed during preprocessing.

The *state* parameter must be a pointer to a **_CrtMemState** structure that has been filled in by _CrtMemCheckpoint or returned by _CrtMemDifference before **_CrtMemDumpStatistics** is called. If *state* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and no action is taken. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

For more information about heap state functions and the **_CrtMemState** structure, see Heap State Reporting Functions. For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see Memory Management and the Debug Heap.

Requirements

Routine	Required header	Optional headers	Compatibility
_CrtMemDumpStatistics	<crtdbg.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries: Debug versions of C Run-Time Libraries only.

Example

See crt_dbg1 Sample: C Run-Time Basic Debugging Features.

.NET Framework Equivalent

System.Diagnostics.PerformanceCounter

See Also
Reference
Debug Routines

_CrtReportBlockType

Returns the block type/subtype associated with a given debug heap block pointer.

```
int _CrtReportBlockType(
      const void * pBlock
};
```

Parameters

pBlock

Pointer to a valid debug heap block.

Return Value

When passed a valid debug heap pointer, the **_CrtReportBlockType** function returns the block type and subtype in the form of an **int**. When passed an invalid pointer, the function returns -1.

Remarks

To extract the type and subtype returned by **_CrtReportBlockType**, use the macros **_BLOCK_TYPE** and **_BLOCK_SUBTYPE** (both defined in Crtdbg.h) on the return value.

For information about the allocation block types and how they are used, see Types of Blocks on the Debug Heap.

Requirements

Routine	Required header	Compatibility
_CrtReportBlockType	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

```
// crt_crtreportblocktype.cpp
// compile with: /MDd

#include <malloc.h>
#include <stdio.h>
#include <crtdbg.h>

void __cdecl Dumper(void *ptr, void *)
{
    int block = _CrtReportBlockType(ptr);
    _RPT3(_CRT_WARN, "Dumper found block at %p: type %d, subtype %d\n", ptr,
          _BLOCK_TYPE(block), _BLOCK_SUBTYPE(block));
}

void __cdecl LeakDumper(void *ptr, size_t sz)
{
    int block = _CrtReportBlockType(ptr);
    _RPT4(_CRT_WARN, "LeakDumper found block at %p:"
                     " type %d, subtype %d, size %d\n", ptr,
          _BLOCK_TYPE(block), _BLOCK_SUBTYPE(block), sz);
}

int main(void)
```

```
{
    _CrtSetDbgFlag(_CrtSetDbgFlag(_CRTDBG_REPORT_FLAG) |
    _CRTDBG_LEAK_CHECK_DF);
    _CrtSetReportMode( _CRT_WARN, _CRTDBG_MODE_FILE );
    _CrtSetReportFile( _CRT_WARN, _CRTDBG_FILE_STDOUT );
    _malloc_dbg(10, _NORMAL_BLOCK , __FILE__, __LINE__);
    _malloc_dbg(10, _CLIENT_BLOCK | (1 << 16), __FILE__, __LINE__);
    _malloc_dbg(20, _CLIENT_BLOCK | (2 << 16), __FILE__, __LINE__);
    _malloc_dbg(30, _CLIENT_BLOCK | (3 << 16), __FILE__, __LINE__);
    _CrtDoForAllClientObjects(Dumper, NULL);
    _CrtSetDumpClient(LeakDumper);
}
```

Sample Output

```
Dumper found block at 00314F78: type 4, subtype 3
Dumper found block at 00314F38: type 4, subtype 2
Dumper found block at 00314F00: type 4, subtype 1
Detected memory leaks!
Dumping objects ->
crt_crtreportblocktype.cpp(30) : {55} client block at 0x00314F78, subtype 3, 30 bytes long.
 Data: <                > CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD
crt_crtreportblocktype.cpp(29) : {54} client block at 0x00314F38, subtype 2, 20 bytes long.
 Data: <                > CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD
crt_crtreportblocktype.cpp(28) : {53} client block at 0x00314F00, subtype 1, 10 bytes long.
 Data: <          > CD CD CD CD CD CD CD CD CD CD
crt_crtreportblocktype.cpp(27) : {52} normal block at 0x00314EC8, 10 bytes long.
 Data: <          > CD CD CD CD CD CD CD CD CD CD
Object dump complete.
```

See Also

Reference

_CrtDoForAllClientObjects
_CrtSetDumpClient
_CrtMemDumpAllObjectsSince
_CrtDumpMemoryLeaks

_CrtSetAllocHook

Installs a client-defined allocation function by hooking it into the C run-time debug memory allocation process (debug version only).

```
_CRT_ALLOC_HOOK _CrtSetAllocHook(
   _CRT_ALLOC_HOOK allocHook
);
```

Parameters

allocHook

New client-defined allocation function to hook into the C run-time debug memory allocation process.

Return Value

Returns the previously defined allocation hook function, or **NULL** if *allocHook* is **NULL**.

Remarks

_CrtSetAllocHook allows an application to hook its own allocation function into the C run-time debug library memory allocation process. As a result, every call to a debug allocation function to allocate, reallocate, or free a memory block triggers a call to the application's hook function. **_CrtSetAllocHook** provides an application with an easy method for testing how the application handles insufficient memory situations, the ability to examine allocation patterns, and the opportunity to log allocation information for later analysis. When _DEBUG is not defined, calls to **_CrtSetAllocHook** are removed during preprocessing.

The **_CrtSetAllocHook** function installs the new client-defined allocation function specified in *allocHook* and returns the previously defined hook function. The following example demonstrates how a client-defined allocation hook should be prototyped:

```
int YourAllocHook( int allocType, void *userData, size_t size, int
blockType, long requestNumber, const unsigned char *filename, int
lineNumber);
```

The `allocType` argument specifies the type of allocation operation (**_HOOK_ALLOC**, **_HOOK_REALLOC**, and **_HOOK_FREE**) that triggered the call to the allocation's hook function. When the triggering allocation type is **_HOOK_FREE**, `userData` is a pointer to the user data section of the memory block about to be freed. However, when the triggering allocation type is **_HOOK_ALLOC** or **_HOOK_REALLOC**, `userData` is **NULL** because the memory block has not been allocated yet.

`size` specifies the size of the memory block in bytes, `blockType` indicates the type of the memory block, `requestNumber` is the object allocation order number of the memory block, and, if available, `filename` and `lineNumber` specify the source file name and line number where the triggering allocation operation was initiated.

After the hook function has finished processing, it must return a Boolean value, which tells the main C run-time allocation process how to continue. When the hook function wants the main allocation process to continue as if the hook function had never been called, then the hook function should return **TRUE**. This causes the original triggering allocation operation to be executed. Using this implementation, the hook function can gather and save allocation information for later analysis, without interfering with the current allocation operation or state of the debug heap.

When the hook function wants the main allocation process to continue as if the triggering allocation operation was called and it failed, then the hook function should return **FALSE**. Using this implementation, the hook function can simulate a wide range of memory conditions and debug heap states to test how the application handles each situation.

To clear the hook function, pass **NULL** to **_CrtSetAllocHook**.

For more information about how **_CrtSetAllocHook** can be used with other memory management functions or how to write your own client-defined hook functions, see Writing Your Own Debugging Hook Functions.

Note
_CrtSetAllocHook is not supported under **/clr:pure**.

Requirements

Routine	Required header	Compatibility
_CrtSetAllocHook	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

For a sample of how to use **_CrtSetAllocHook**, see crt_dbg2.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_CrtGetAllocHook

_CrtSetBreakAlloc

Sets a breakpoint on a specified object allocation order number (debug version only).

```
long _CrtSetBreakAlloc(
   long lBreakAlloc
);
```

Parameters

lBreakAlloc

Allocation order number, for which to set the breakpoint.

Return Value

Returns the previous object allocation order number that had a breakpoint set.

Remarks

_CrtSetBreakAlloc allows an application to perform memory leak detection by breaking at a specific point of memory allocation and tracing back to the origin of the request. The function uses the sequential object allocation order number assigned to the memory block when it was allocated in the heap. When _DEBUG is not defined, calls to **_CrtSetBreakAlloc** are removed during preprocessing.

The object allocation order number is stored in the *lRequest* field of the **_CrtMemBlockHeader** structure, defined in Crtdbg.h. When information about a memory block is reported by one of the debug dump functions, this number is enclosed in braces, such as {36}.

For more information about how **_CrtSetBreakAlloc** can be used with other memory management functions, see Tracking Heap Allocation Requests.

Requirements

Routine	Required header	Compatibility
_CrtSetBreakAlloc	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

```
// crt_setbrkal.c
// compile with: -D_DEBUG /MTd -Od -Zi -W3 /c /link -verbose:lib -debug

/*
 * In this program, a call is made to the _CrtSetBreakAlloc routine
 * to verify that the debugger halts program execution when it reaches
 * a specified allocation number.
 */

#include <malloc.h>
#include <crtdbg.h>

int main( )
{
        long allocReqNum;
        char *my_pointer;
```

```
        /*
         * Allocate "my_pointer" for the first
         * time and ensure that it gets allocated correctly
         */
        my_pointer = malloc(10);
        _CrtIsMemoryBlock(my_pointer, 10, &allocReqNum, NULL, NULL);

        /*
         * Set a breakpoint on the allocation request
         * number for "my_pointer"
         */
        _CrtSetBreakAlloc(allocReqNum+2);
        _crtBreakAlloc = allocReqNum+2;

        /*
         * Alternate freeing and reallocating "my_pointer"
         * to verify that the debugger halts program execution
         * when it reaches the allocation request
         */
        free(my_pointer);
        my_pointer = malloc(10);
        free(my_pointer);
        my_pointer = malloc(10);
        free(my_pointer);
}
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_CrtSetDbgFlag

Retrieves or modifies the state of the **_crtDbgFlag** flag to control the allocation behavior of the debug heap manager (debug version only).

```
int _CrtSetDbgFlag(
   int newFlag
);
```

Parameters

newFlag

New state for **_crtDbgFlag**.

Return Value

Returns the previous state of **_crtDbgFlag**.

Remarks

The **_CrtSetDbgFlag** function allows the application to control how the debug heap manager tracks memory allocations by modifying the bit fields of the **_crtDbgFlag** flag. By setting the bits (turning on), the application can instruct the debug heap manager to perform special debugging operations, including checking for memory leaks when the application exits and reporting if any are found, simulating low-memory conditions by specifying that freed memory blocks should remain in the heap's linked list, and verifying the integrity of the heap by inspecting each memory block at every allocation request. When _DEBUG is not defined, calls to **_CrtSetDbgFlag** are removed during preprocessing.

The following table lists the bit fields for **_crtDbgFlag** and describes their behavior. Because setting the bits results in increased diagnostic output and reduced program execution speed, these bits are not set (turned off) by default. For more information about these bit fields, see Using the Debug Heap.

Bit field	Default	Description
_CRTDBG_ALLOC_MEM_DF	OFF	ON: Enable debug heap allocations and use of memory block type identifiers, such as **_CLIENT_BLOCK**. OFF: Add new allocations to heap's linked list, but set block type to **_IGNORE_BLOCK**. Can also be combined with any of the heap-frequency check macros.
_CRTDBG_CHECK_ALWAYS_DF	OFF	ON: Call _CrtCheckMemory at every allocation and deallocation request. OFF: **_CrtCheckMemory** must be called explicitly. Heap-frequency check macros have no effect when this flag is set.
_CRTDBG_CHECK_CRT_DF	OFF	ON: Include **_CRT_BLOCK** types in leak detection and memory state difference operations. OFF: Memory used internally by the run-time library is ignored by these operations. Can also be combined with any of the heap-frequency check macros.
_CRTDBG_DELAY_FREE_MEM_DF	OFF	ON: Keep freed memory blocks in the heap's linked list, assign them the **_FREE_BLOCK** type, and fill them with the byte value 0xDD. OFF: Do not keep freed blocks in the heap's linked list. Can also be combined with any of the heap-frequency check macros.
_CRTDBG_LEAK_CHECK_DF	OFF	ON: Perform automatic leak checking at program exit through a call to _CrtDumpMemoryLeaks and generate an error report if the application failed to free all the memory it allocated. OFF: Do not automatically perform leak checking at program exit. Can also be combined with any of the heap-frequency check macros.

Heap-Check Frequency Macros

You can specify how often the C run-time library performs validation of the debug heap (**_CrtCheckMemory**) based on the number of calls to **malloc**, **realloc**, **free**, and **_msize**.

_CrtSetDbgFlag then inspect the upper 16 bits of the *newFlag* parameter for a value. The value specified are the number of **malloc**, **realloc**, **free**, and **_msize** calls between **_CrtCheckMemory** calls. Four predefined macros are provided for this purpose.

Macro	**Number of malloc, realloc, free, and _msize calls between calls to _CrtCheckMemory**
_CRTDBG_CHECK_EVERY_16_DF	16
_CRTDBG_CHECK_EVERY_128_DF	128
_CRTDBG_CHECK_EVERY_1024_DF	1024
_CRTDBG_CHECK_DEFAULT_DF	0 (by default, no heap checks)

By default, **_CrtCheckMemory** is called once every 1,024 times you call **malloc**, **realloc**, **free**, and **_msize**.

For example, you could specify a heap check every 16 **malloc**, **realloc**, **free**, and **_msize** operations with the following code:

```
#include <crtdbg.h>
int main( )
{
int tmp;

// Get the current bits
tmp = _CrtSetDbgFlag(_CRTDBG_REPORT_FLAG);

// Clear the upper 16 bits and OR in the desired freqency
tmp = (tmp & 0x0000FFFF) | _CRTDBG_CHECK_EVERY_16_DF;

// Set the new bits
_CrtSetDbgFlag(tmp);
}
```

The upper 16 bits of the *newFlag* parameter are ignored when _CRTDBG_CHECK_ALWAYS_DF is specified. In this case, **_CrtCheckMemory** is called each time you call **malloc**, **realloc**, **free**, and **_msize**.

newFlag is the new state to apply to the **_crtDbgFlag** and is a combination of the values for each of the bit fields.

To change one or more of these bit fields and create a new state for the flag

1. Call **_CrtSetDbgFlag** with *newFlag* equal to **_CRTDBG_REPORT_FLAG** to obtain the current **_crtDbgFlag** state and store the returned value in a temporary variable.
2. Turn on any bits by **OR**-ing the temporary variable with the corresponding bitmasks (represented in the application code by manifest constants).
3. Turn off the other bits by **AND**-ing the variable with a bitwise **NOT** of the appropriate bitmasks.
4. Call **_CrtSetDbgFlag** with *newFlag* equal to the value stored in the temporary variable to set the new state for **_crtDbgFlag**.

The following code demonstrates how to simulate low-memory conditions by keeping freed memory blocks in the heap's linked list and prevent **_CrtCheckMemory** from being called at every allocation request:

```
// Get the current state of the flag
// and store it in a temporary variable
int tmpFlag = _CrtSetDbgFlag( _CRTDBG_REPORT_FLAG );

// Turn On (OR) - Keep freed memory blocks in the
// heap's linked list and mark them as freed
```

```
tmpFlag |= _CRTDBG_DELAY_FREE_MEM_DF;

// Turn Off (AND) - prevent _CrtCheckMemory from
// being called at every allocation request
tmpFlag &= ~_CRTDBG_CHECK_ALWAYS_DF;

// Set the new state for the flag
_CrtSetDbgFlag( tmpFlag );
```

For an overview of memory management and the debug heap, see Memory Management and the Debug Heap.

To disable a flag with the **_CrtSetDbgFlag** function, you should **AND** the variable with the bitwise **NOT** of the bitmask.

If *newFlag* is not a valid value, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns the previous state of **_crtDbgFlag**.

Requirements

Routine	Required header	Compatibility
_CrtSetDbgFlag	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

```
// crt_crtsetdflag.c
// compile with: /c -D_DEBUG /MTd -Od -Zi -W3 /link -verbose:lib /debug
/*
 * This program concentrates on allocating and freeing memory
 * blocks to test the functionality of the _crtDbgFlag flag..
 */

#include <string.h>
#include <malloc.h>
#include <crtdbg.h>

int main( )
{
        char *p1, *p2;
        int tmpDbgFlag;

        _CrtSetReportMode( _CRT_ERROR, _CRTDBG_MODE_FILE );
        _CrtSetReportFile( _CRT_ERROR, _CRTDBG_FILE_STDERR );
        /*
         * Set the debug-heap flag to keep freed blocks in the
         * heap's linked list - This will allow us to catch any
         * inadvertent use of freed memory
         */
        tmpDbgFlag = _CrtSetDbgFlag(_CRTDBG_REPORT_FLAG);
        tmpDbgFlag |= _CRTDBG_DELAY_FREE_MEM_DF;
        tmpDbgFlag |= _CRTDBG_LEAK_CHECK_DF;
        _CrtSetDbgFlag(tmpDbgFlag);

        /*
         * Allocate 2 memory blocks and store a string in each
         */
        p1 = malloc( 34 );
        p2 = malloc( 38 );
        strcpy_s( p1, 34, "p1 points to a Normal allocation block" );
        strcpy_s( p2, 38, "p2 points to a Client allocation block" );
```

```
        /*
         * Free both memory blocks
         */
        free( p2 );
        free( p1 );

        /*
         * Set the debug-heap flag to no longer keep freed blocks in the
         * heap's linked list and turn on Debug type allocations (CLIENT)
         */
        tmpDbgFlag = _CrtSetDbgFlag(_CRTDBG_REPORT_FLAG);
        tmpDbgFlag |= _CRTDBG_ALLOC_MEM_DF;
        tmpDbgFlag &= ~_CRTDBG_DELAY_FREE_MEM_DF;
        _CrtSetDbgFlag(tmpDbgFlag);

        /*
         * Explicitly call _malloc_dbg to obtain the filename and
         * line number of our allocation request and also so we can
         * allocate CLIENT type blocks specifically for tracking
         */
        p1 = _malloc_dbg( 40, _NORMAL_BLOCK, __FILE__, __LINE__ );
        p2 = _malloc_dbg( 40, _CLIENT_BLOCK, __FILE__, __LINE__ );
        strcpy_s( p1, 40, "p1 points to a Normal allocation block" );
        strcpy_s( p2, 40, "p2 points to a Client allocation block" );

        /*
         * _free_dbg must be called to free the CLIENT block
         */
        _free_dbg( p2, _CLIENT_BLOCK );
        free( p1 );

        /*
         * Allocate p1 again and then exit - this will leave unfreed
         * memory on the heap
         */
        p1 = malloc( 10 );
}
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines
_crtDbgFlag
_CrtCheckMemory

Concepts

Heap State Reporting Functions

_CrtSetDebugFillThreshold

Retrieves or modifies the threshold controlling buffer-filling behavior in debug functions.

```
size_t _CrtSetDebugFillThreshold(
   size_t _NewThreshold
);
```

Parameters

newThreshold

New threshold.

Return Value

The previous threshold.

Remarks

The debug versions of some security-enhanced CRT functions fill the buffer passed to them with a special character (0xFD). This helps to find cases where the incorrect size was passed to the function. Unfortunately, it also reduces performance. To improve performance, use **_CrtSetDebugFillThreshold** to disable buffer-filling for buffers larger than the threshold. A threshold of 0 will disable it for all buffers.

The default threshold is **SIZE_T_MAX**.

Here is a list of the affected functions.

- strcpy_s, wcscpy_s, _mbscpy_s
- strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l
- _mbsnbcpy_s, _mbsnbcpy_s_l
- strcat_s, wcscat_s, _mbscat_s
- strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l
- _mbsnbcat_s, _mbsnbcat_s_l
- _strset_s, _strset_s_l, _wcsset_s, _wcsset_s_l, _mbsset_s, _mbsset_s_l
- _strnset_s, _strnset_s_l, _wcsnset_s, _wcsnset_s_l, _mbsnset_s, _mbsnset_s_l
- _mbsnbset_s, _mbsnbset_s_l
- _makepath_s, _wmakepath_s
- _splitpath_s, _wsplitpath_s
- _strlwr_s, _strlwr_s_l, _mbslwr_s, _mbslwr_s_l, _wcslwr_s, _wcslwr_s_l
- _strupr_s, _strupr_s_l, _mbsupr_s, _mbsupr_s_l, _wcsupr_s, _wcsupr_s_l
- strerror_s, _strerror_s, _wcserror_s, __wcserror_s
- _itoa_s, _i64toa_s, _ui64toa_s, _itow_s, _i64tow_s, _ui64tow_s
- _ecvt_s
- _fcvt_s
- _gcvt_s

Requirements

Routine	Required header	Compatibility

_CrtSetDebugFillThreshold	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

```
// crt_crtsetdebugfillthreshold.cpp
// compile with: /MTd
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <crtdbg.h>

void Clear( char buff[], size_t size )
{
   for( int i=0; i<size; ++i )
      buff[i] = 0;
}

void Print( char buff[], size_t size )
{
   for( int i=0; i<size; ++i )
      printf( "%02x  %c\n", (unsigned char)buff[i], buff[i] );
}

int main( void )
{
   char buff[10];

   printf( "With buffer-filling on:\n" );
   strcpy_s( buff, _countof(buff), "howdy" );
   Print( buff, _countof(buff) );

   _CrtSetDebugFillThreshold( 0 );

   printf( "With buffer-filling off:\n" );
   Clear( buff, _countof(buff) );
   strcpy_s( buff, _countof(buff), "howdy" );
   Print( buff, _countof(buff) );
}
```

```
With buffer-filling on:
68  h
6f  o
77  w
64  d
79  y
00
fd  ²
fd  ²
fd  ²
fd  ²
With buffer-filling off:
68  h
6f  o
77  w
64  d
79  y
00
00
```

```
00
00
00
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_CrtSetDumpClient

Installs an application-defined function to dump **_CLIENT_BLOCK** type memory blocks (debug version only).

```
_CRT_DUMP_CLIENT _CrtSetDumpClient(
   _CRT_DUMP_CLIENT dumpClient
);
```

Parameters

dumpClient

New client-defined memory dump function to hook into the C run-time debug memory dump process.

Return Value

Returns the previously defined client block dump function.

Remarks

The **_CrtSetDumpClient** function allows the application to hook its own function to dump objects stored in **_CLIENT_BLOCK** memory blocks into the C run-time debug memory dump process. As a result, every time a debug dump function such as _CrtMemDumpAllObjectsSince or _CrtDumpMemoryLeaks dumps a **_CLIENT_BLOCK** memory block, the application's dump function is called as well. **_CrtSetDumpClient** provides an application with an easy method for detecting memory leaks and validating or reporting the contents of data stored in **_CLIENT_BLOCK** blocks. When _DEBUG is not defined, calls to **_CrtSetDumpClient** are removed during preprocessing.

The **_CrtSetDumpClient** function installs the new application-defined dump function specified in *dumpClient* and returns the previously defined dump function. An example of a client block dump function is as follows:

```
void DumpClientFunction( void *userPortion, size_t blockSize );
```

The `userPortion` argument is a pointer to the beginning of the user data portion of the memory block and `blockSize` specifies the size of the allocated memory block in bytes. The client block dump function must return **void**. The pointer to the client dump function that is passed to **_CrtSetDumpClient** is of type **_CRT_DUMP_CLIENT**, as defined in Crtdbg.h:

```
typedef void (__cdecl *_CRT_DUMP_CLIENT)( void *, size_t );
```

For an example of how to implement an application-defined dump function, see crt_dbg2 Sample: C Run-Time Debugging Hook Functions. For more information about functions that operate on **_CLIENT_BLOCK** type memory blocks, see Client Block Hook Functions. The _CrtReportBlockType function can be used to return information about block types and subtypes.

Requirements

Routine	Required header	Compatibility
_CrtSetDumpClient	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

For a sample of how to use **_CrtSetDumpClient**, see crt_dbg2.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_CrtReportBlockType

_CrtGetDumpClient

_CrtSetReportFile

After specifying **_CRTDBG_MODE_FILE** with _CrtSetReportMode, you can specify the file handle to receive the message text. **_CrtSetReportFile** is also used by _CrtDbgReport, _CrtDbgReportW to specify the destination of text (debug version only).

```
_HFILE _CrtSetReportFile(
   int reportType,
   _HFILE reportFile
);
```

Parameters

reportType

Report type: **_CRT_WARN**, **_CRT_ERROR**, and **_CRT_ASSERT**.

reportFile

New report file for *reportType*.

Return Value

On successful completion, **_CrtSetReportFile** returns the previous report file defined for the report type specified in *reportType*. If an invalid value is passed in for *reportType*, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **_CRTDBG_HFILE_ERROR**. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

_CrtSetReportFile is used with the _CrtSetReportMode function to define the destination or destinations for a specific report type generated by **_CrtDbgReport**. When **_CrtSetReportMode** has been called to assign the **_CRTDBG_MODE_FILE** reporting mode for a specific report type, **_CrtSetReportFile** should then be called to define the specific file or stream to use as the destination. When _DEBUG is not defined, calls to **_CrtSetReportFile** are removed during preprocessing.

The following table shows a list of the available choices for *reportFile* and the resulting behavior of **_CrtDbgReport**. These options are defined as bit flags in Crtdbg.h.

file handle

A handle to the file that will be the destination for messages. No attempt is made to verify the validity of the handle. You must open and close the handle to the file. For example:

```
HANDLE hLogFile;
hLogFile = CreateFile("c:\\log.txt", GENERIC_WRITE,
   FILE_SHARE_WRITE, NULL, CREATE_ALWAYS,
   FILE_ATTRIBUTE_NORMAL, NULL);
_CrtSetReportMode(_CRT_WARN, _CRTDBG_MODE_FILE);
_CrtSetReportFile(_CRT_WARN, hLogFile);

_RPT0(_CRT_WARN,"file message\n");
CloseHandle(hLogFile);
```

_CRTDBG_FILE_STDERR

Writes message to **stderr**, which can be redirected as follows:

```
freopen( "c:\\log2.txt", "w", stderr);
_CrtSetReportMode(_CRT_ERROR, _CRTDBG_MODE_FILE);
_CrtSetReportFile(_CRT_ERROR, _CRTDBG_FILE_STDERR);

_RPT0(_CRT_ERROR,"1st message\n");
```

_CRTDBG_FILE_STDOUT

Writes message to **stdout**, which you can redirect.

_CRTDBG_REPORT_FILE

Returns the current report mode.

The report file used by each report type can be separately controlled. For example, it is possible to specify that a *reportType* of **_CRT_ERROR** be reported to **stderr**, while a *reportType* of **_CRT_ASSERT** be reported to a user-defined file handle or stream.

Requirements

Routine	**Required header**	**Optional header**	**Compatibility**
_CrtSetReportFile	<crtdbg.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries: Debug versions of C Run-Time Libraries only.

Example

See report Sample: C Run-Time Debugging Report Functions.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_CrtSetReportHook

Installs a client-defined reporting function by hooking it into the C run-time debug reporting process (debug version only).

```
_CRT_REPORT_HOOK _CrtSetReportHook(
   _CRT_REPORT_HOOK reportHook
);
```

Parameters

reportHook

New client-defined reporting function to hook into the C run-time debug reporting process.

Return Value

Returns the previous client-defined reporting function.

Remarks

_CrtSetReportHook allows an application to use its own reporting function into the C run-time debug library reporting process. As a result, whenever _CrtDbgReport is called to generate a debug report, the application's reporting function is called first. This functionality enables an application to perform operations such as filtering debug reports so it can focus on specific allocation types or send a report to destinations not available by using **_CrtDbgReport**. When _DEBUG is not defined, calls to **_CrtSetReportHook** are removed during preprocessing.

For a more robust version of **_CrtSetReportHook**, see _CrtSetReportHook2.

The **_CrtSetReportHook** function installs the new client-defined reporting function specified in *reportHook* and returns the previous client-defined hook. The following example demonstrates how a client-defined report hook should be prototyped:

```
int YourReportHook( int reportType, char *message, int *returnValue );
```

where `reportType` is the debug report type (**_CRT_WARN**, **_CRT_ERROR**, or **_CRT_ASSERT**), `message` is the fully assembled debug user message to be contained in the report, and `returnValue` is the value specified by the client-defined reporting function that should be returned by **_CrtDbgReport**. For a complete description of the available report types, see the _CrtSetReportMode function.

If the client-defined reporting function completely handles the debug message such that no further reporting is required, then the function should return **TRUE**. When the function returns **FALSE**, **_CrtDbgReport** is called to generate the debug report using the current settings for the report type, mode, and file. In addition, by specifying the **_CrtDbgReport** return value in `returnValue`, the application can also control whether a debug break occurs. For a complete description of how the debug report is configured and generated, see **_CrtSetReportMode**, _CrtSetReportFile, and **_CrtDbgReport**.

For more information about using other hook-capable run-time functions and writing your own client-defined hook functions, see Writing Your Own Debug Hook Functions.

Note

If your application is compiled with **/clr** and the reporting function is called after the application has exited main, the CLR will throw an exception if the reporting function calls any CRT functions.

Requirements

Routine	Required header	Compatibility
_CrtSetReportHook	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

For a sample of how to use **_CrtSetReportHook**, see report.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_CrtGetReportHook

_CrtSetReportHook2, _CrtSetReportHookW2

Installs or uninstalls a client-defined reporting function by hooking it into the C run-time debug reporting process (debug version only).

```
int _CrtSetReportHook2(
   int mode,
   _CRT_REPORT_HOOK pfnNewHook
);
int _CrtSetReportHookW2(
   int mode,
   _CRT_REPORT_HOOKW pfnNewHook
);
```

Parameters

mode

The action to take: **_CRT_RPTHOOK_INSTALL** or **_CRT_RPTHOOK_REMOVE**.

pfnNewHook

Report hook to install or remove in the narrow-character version of this function.

pfnNewHook

Report hook to install or remove in the wide-character version of this function.

Return Value

-1 if an error was encountered, with **EINVAL** or **ENOMEM** set; otherwise returns the reference count of *pfnNewHook* after the call.

Remarks

_CrtSetReportHook2 and **_CrtSetReportHookW2** let you hook or unhook a function, whereas _CrtSetReportHook only lets you hook a function.

_CrtSetReportHook2 or **_CrtSetReportHookW2** should be used instead of **_CrtSetReportHook** when the hook call is made in a DLL and when multiple DLLs might be loaded and setting their own hook functions. In such a situation, DLLs can be unloaded in a different order than they were loaded and the hook function can be left pointing at an unloaded DLL. Any debug output crashes the process if the hook functions were added with **_CrtSetReportHook**.

Any hook functions added with **_CrtSetReportHook** are called if there are no hook functions added with **_CrtSetReportHook2** or **_CrtSetReportHookW2** or if all hook functions added with **_CrtSetReportHook2** and **_CrtSetReportHookW2** return **FALSE**.

The wide-character version of this function is available in Visual C++ 2005. The report hook functions take a string whose type (wide or narrow characters) must match the version of this function used. Use the following function prototype for the report hooks used with the wide-character version of this function:

```
int YourReportHook( int reportType, wchar_t *message, int *returnValue );
```

Use the following prototype for the narrow-character report hooks:

```
int YourReportHook( int reportType, char *message, int *returnValue );
```

These functions validate their parameters. If *mode* or *pfnNewNook* is invalid, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1.

Note

If your application is compiled with **/clr** and the reporting function is called after the application has exited main, the CLR will throw an exception if the reporting function calls any CRT functions.

Requirements

Routine	Required header	Optional header	Compatibility
_CrtSetReportHook2	<crtdbg.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_CrtSetReportHookW2	<crtdbg.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

```
// crt_setreporthook2.c
#include <windows.h>
#include <stdio.h>
#include <crtdbg.h>
#include <assert.h>

int __cdecl TestHook1(int nReportType, char* szMsg, int* pnRet)
{
   int nRet = FALSE;

   printf("CRT report hook 1.\n");
   printf("CRT report type is \"");
   switch (nReportType)
   {
      case _CRT_ASSERT:
      {
         printf("_CRT_ASSERT");
         // nRet = TRUE;   // Always stop for this type of report
         break;
      }

      case _CRT_WARN:
      {
         printf("_CRT_WARN");
         break;
      }

      case _CRT_ERROR:
      {
         printf("_CRT_ERROR");
         break;
      }

      default:
      {
         printf("???Unknown???");
         break;
      }
   }

   printf("\".\nCRT report message is:\n\t");
   printf(szMsg);
```

```
   if   (pnRet)
      *pnRet = 0;

   return   nRet;
}

int __cdecl   TestHook2(int nReportType, char* szMsg, int* pnRet)
{
   int   nRet = FALSE;

   printf("CRT report hook 2.\n");
   printf("CRT report type is \"");
   switch   (nReportType)
   {
      case _CRT_WARN:
      {
         printf("_CRT_WARN");
         break;
      }

      case _CRT_ERROR:
      {
         printf("_CRT_ERROR");
         break;
      }

      case _CRT_ASSERT:
      {
         printf("_CRT_ASSERT");
         nRet = TRUE;   // Always stop for this type of report
         break;
      }

      default:
      {
         printf("???Unknown???");
         break;
      }
   }

   printf("\".\nCRT report message is: \t");
   printf(szMsg);

   if   (pnRet)
      *pnRet = 0;
   // printf("CRT report code is %d.\n", *pnRet);
   return   nRet;
}

int   main(int argc, char* argv[])
{
   int   nRet = 0;

   nRet = _CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook1);
   printf("_CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook1)"
          " returned %d\n", nRet);
   nRet = _CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook2);
   printf("_CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook2)"
          " returned %d\n", nRet);
   nRet = _CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook2);
   printf("_CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook2)"
          " returned %d\n", nRet);
   nRet = _CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook1);
   printf("_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook1)"
          " returned %d\n", nRet);
   nRet = _CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook1);
   printf("_CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook1)"
```

```
            " returned %d\n", nRet);

    _ASSERT(0);

    nRet = _CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook2);
    printf("_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook2)"
            " returned %d\n", nRet);
    nRet = _CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook2);
    printf("_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook2)"
            " returned %d\n", nRet);
    nRet = _CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook2);
    printf("_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook2)"
            " returned %d\n", nRet);
    nRet = _CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook1);
    printf("_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook1)"
            " returned %d\n", nRet);

    return   nRet;
}
```

Output

```
_CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook1) returned 0
_CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook2) returned 0
_CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook2) returned 0
_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook1) returned 0
_CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook1) returned 0
_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook2) returned 0
_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook2) returned 0
_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook2) returned 0
_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook1) returned 0
```

See Also

Reference

Debug Routines

_CrtSetReportMode

Specifies the destination or destinations for a specific report type generated by **_CrtDbgReport** and any macros that call _CrtDbgReport, _CrtDbgReportW, such as _ASSERT, _ASSERTE Macros, _ASSERT, _ASSERTE Macros, _RPT, _RPTF, _RPTW, _RPTFW Macros, and _RPT, _RPTF, _RPTW, _RPTFW Macros (debug version only).

```
int _CrtSetReportMode(
   int reportType,
   int reportMode
);
```

Parameters

reportType

Report type: **_CRT_WARN**, **_CRT_ERROR**, and **_CRT_ASSERT**.

reportMode

New report mode or modes for *reportType*.

Return Value

On successful completion, **_CrtSetReportMode** returns the previous report mode or modes for the report type specified in *reportType*. If an invalid value is passed in as *reportType* or an invalid mode is specified for *reportMode*, **_CrtSetReportMode** invokes the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns -1. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

_CrtSetReportMode specifies the output destination for **_CrtDbgReport**. Because the macros **_ASSERT**, **_ASSERTE**, **_RPT**, and **_RPTF** call **_CrtDbgReport**, **_CrtSetReportMode** specifies the output destination of text specified with those macros.

When _DEBUG is not defined, calls to **_CrtSetReportMode** are removed during preprocessing.

If you do not call **_CrtSetReportMode** to define the output destination of messages, then the following defaults are in effect:

- Assertion failures and errors are directed to a debug message window.
- Warnings from Windows applications are sent to the debugger's output window.
- Warnings from console applications are not displayed.

The following table lists the report types defined in Crtdbg.h.

Report type	Description
_CRT_WARN	Warnings, messages, and information that does not need immediate attention.
_CRT_ERROR	Errors, unrecoverable problems, and issues that require immediate attention.
_CRT_ASSERT	Assertion failures (asserted expressions that evaluate to **FALSE**).

The **_CrtSetReportMode** function assigns the new report mode specified in *reportMode* to the report type specified in *reportType* and returns the previously defined report mode for *reportType*. The following table lists the available choices for *reportMode* and the resulting behavior of **_CrtDbgReport**. These options are defined as bit flags in Crtdbg.h.

Report mode	_CrtDbgReport behavior
_CRTDBG_MODE_DEBUG	Writes the message to the debugger's output window.
_CRTDBG_MODE_FILE	Writes the message to a user-supplied file handle. _CrtSetReportFile should be called to define the specific file or stream to use as the destination.

_CRTDBG_MODE_WNDW	Creates a message box to display the message along with the **Abort**, **Retry**, and **Ignore** buttons.
_CRTDBG_REPORT_MODE	Returns *reportMode* for the specified *reportType*: 1 **_CRTDBG_MODE_FILE** 2 **_CRTDBG_MODE_DEBUG** 4 **_CRTDBG_MODE_WNDW**

Each report type can be reported using one, two, or three modes or no mode at all. Therefore, it is possible to have more than one destination defined for a single report type. For example, the following code fragment causes assertion failures to be sent to both a debug message window and to **stderr**:

```
_CrtSetReportMode( _CRT_ASSERT, _CRTDBG_MODE_FILE | _CRTDBG_MODE_WNDW );
_CrtSetReportFile( _CRT_ASSERT, _CRTDBG_FILE_STDERR );
```

In addition, the reporting mode or modes for each report type can be separately controlled. For example, it is possible to specify that a *reportType* of **_CRT_WARN** be sent to an output debug string, while **_CRT_ASSERT** be displayed using a debug message window and sent to **stderr**, as previously illustrated.

Requirements

Routine	Required header	Optional header	Compatibility
_CrtSetReportMode	<crtdbg.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries: Debug versions of C Run-Time Libraries only.

Example

See report Sample: C Run-Time Debugging Report Functions.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

cscanf

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _cscanf instead.

_cscanf, _cscanf_l, _cwscanf, _cwscanf_l

Reads formatted data from the console. These functions are deprecated because more secure versions are available; see _cscanf_s, _cscanf_s_l, _cwscanf_s, _cwscanf_s_l.

```
int _cscanf(
   const char *format [,
      argument] ...
);
int _cscanf_l(
   const char *format,
   locale_t locale [,
      argument] ...
);
int _cwscanf(
   const wchar_t *format [,
      argument] ...
);
int _cwscanf_l(
   const wchar_t *format,
   locale_t locale [,
      argument] ...
);
```

Parameters

format

Format-control string.

argument

Optional parameters.

locale

The locale to use.

Return Value

The number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is **EOF** for an attempt to read at end of file. This can occur when keyboard input is redirected at the operating-system command-line level. A return value of 0 means that no fields were assigned.

Remarks

The **_cscanf** function reads data directly from the console into the locations given by *argument*. The _getche function is used to read characters. Each optional parameter must be a pointer to a variable with a type that corresponds to a type specifier in *format*. The format controls the interpretation of the input fields and has the same form and function as the *format* parameter for the scanf function. While **_cscanf** normally echoes the input character, it does not do so if the last call was to **_ungetch**.

This function validates its parameters. If format is NULL, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **EOF**.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tcscanf	**_cscanf**	**_cscanf**	**_cwscanf**
_tcscanf_l	**_cscanf_l**	**_cscanf_l**	**_cwscanf_l**

Requirements

Routine	Required header	Compatibility
_cscanf, **_cscanf_l**	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_cwscanf, **_cwscanf_l**	<conio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_cscanf.c
// compile with: /c /W3
/* This program prompts for a string
 * and uses _cscanf to read in the response.
 * Then _cscanf returns the number of items
 * matched, and the program displays that number.
 */

#include <stdio.h>
#include <conio.h>

int main( void )
{
   int   result, i[3];

   _cprintf_s( "Enter three integers: ");
   result = _cscanf( "%i %i %i", &i[0], &i[1], &i[2] ); // C4996
   // Note: _cscanf is deprecated; consider using _cscanf_s instead
   _cprintf_s( "\r\nYou entered " );
   while( result-- )
      _cprintf_s( "%i ", i[result] );
   _cprintf_s( "\r\n" );
}
```

Input

```
1 2 3
```

Output

```
Enter three integers: 1 2 3
You entered 3 2 1
```

See Also

Reference

Console and Port I/O
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l
fscanf, _fscanf_l, fwscanf, _fwscanf_l
scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l
sscanf, _sscanf_l, swscanf, _swscanf_l

_cscanf_s, _cscanf_s_l, _cwscanf_s, _cwscanf_s_l

Read formatted data from the console. These are more secure versions of _cscanf, _cscanf_l, _cwscanf, _cwscanf_l with security enhancements as described in Security Enhancements in the CRT.

```
int _cscanf_s(
   const char *format [,
      argument] ...
);
int _cscanf_s_l(
   const char *format,
   locale_t locale [,
      argument] ...
);
int _cwscanf_s(
   const wchar_t *format [,
      argument] ...
);
int _cwscanf_s_l(
   const wchar_t *format,
   locale_t locale [,
      argument] ...
);
```

Parameters

format

Format-control string.

argument

Optional parameters.

locale

The locale to use.

Return Value

The number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is **EOF** for an attempt to read at end of file. This can occur when keyboard input is redirected at the operating-system command-line level. A return value of 0 means that no fields were assigned.

These functions validate their parameters. If *format* is a null pointer, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return **EOF** and **errno** is set to **EINVAL**.

Remarks

The **_cscanf_s** function reads data directly from the console into the locations given by *argument*. The _getche function is used to read characters. Each optional parameter must be a pointer to a variable with a type that corresponds to a type specifier in *format*. The format controls the interpretation of the input fields and has the same form and function as the *format* parameter for the scanf_s function. While **_cscanf_s** normally echoes the input character, it does not do so if the last call was to **_ungetch**.

Like other secure versions of functions in the **scanf** family, **_cscanf_s** and **_cswscanf_s** require size arguments for the type field characters **c**, **C**, **s**, **S**, and **[**. For more information, see scanf Width Specification.

Note
The size parameter is of type **unsigned**, not **size_t**.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined

_tcscanf_s	**_cscanf_s**	**_cscanf_s**	**_cwscanf_s**
_tcscanf_s_l	**_cscanf_s_l**	**_cscanf_s_l**	**_cwscanf_s_l**

Requirements

Routine	Required header	Compatibility
_cscanf_s, **_cscanf_s_l**	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_cwscanf_s, **_cwscanf_s_l**	<conio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_cscanf_s.c
// compile with: /c
/* This program prompts for a string
 * and uses _cscanf_s to read in the response.
 * Then _cscanf_s returns the number of items
 * matched, and the program displays that number.
 */

#include <stdio.h>
#include <conio.h>

int main( void )
{
   int result, n[3];
   int i;

   result = _cscanf_s( "%i %i %i", &n[0], &n[1], &n[2] );
   _cprintf_s( "\r\nYou entered " );
   for( i=0; i<result; i++ )
      _cprintf_s( "%i ", n[i] );
   _cprintf_s( "\r\n" );
}
```

Input

```
1 2 3
```

Output

```
You entered 1 2 3
```

See Also

Reference

Console and Port I/O
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l
fscanf_s, _fscanf_s_l, fwscanf_s, _fwscanf_s_l
scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l
sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l

ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64

Convert a time value to a string and adjust for local time zone settings. These functions are deprecated because more secure versions are available; see ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s.

```
char *ctime(
   const time_t *timer
);
char *_ctime32(
   const __time32_t *timer )
;
char *_ctime64(
   const __time64_t *timer )
;
wchar_t *_wctime(
   const time_t *timer
);
wchar_t *_wctime32(
   const __time32_t *timer
);
wchar_t *_wctime64(
   const __time64_t *timer
);
```

Parameters

timer

Pointer to stored time.

Return Value

A pointer to the character string result. **NULL** will be returned if:

- *time* represents a date before midnight, January 1, 1970, UTC.
- If you use **_ctime32** or **_wctime32** and *time* represents a date after 03:14:07 January 19, 2038.
- If you use **_ctime64** or **_wctime64** *and time* represents a date after 23:59:59, December 31, 3000, UTC.

In Visual C++ 2005, **ctime** is an inline function which evaluates to **_ctime64** and **time_t** is equivalent to **__time64_t**. If you need to force the compiler to interpret **time_t** as the old 32-bit **time_t**, you can define **_USE_32BIT_TIME_T**. Doing this will cause **ctime** to evaluate to **_ctime32**. This is not recommended because your application may fail after January 18, 2038, and it is not allowed on 64-bit platforms.

Remarks

The **ctime** function converts a time value stored as a time_t value into a character string. The *timer* value is usually obtained from a call to time, which returns the number of seconds elapsed since midnight (00:00:00), January 1, 1970, coordinated universal time (UTC). The return value string contains exactly 26 characters and has the form:

```
Wed Jan 02 02:03:55 1980\n\0
```

A 24-hour clock is used. All fields have a constant width. The newline character ('\n') and the null character ('\0') occupy the last two positions of the string.

The converted character string is also adjusted according to the local time zone settings. See the **time**, _ftime, and localtime functions for information on configuring the local time and the _tzset function for details about defining the time zone environment and global variables.

A call to **ctime** modifies the single statically allocated buffer used by the **gmtime** and **localtime** functions. Each call to one of these routines destroys the result of the previous call. **ctime** shares a static buffer with the **asctime** function. Thus, a call to **ctime** destroys the results of any previous call to **asctime**, **localtime**, or **gmtime**.

_wctime and **_wctime64** are the wide-character version of **ctime** and **_ctime64**; returning a pointer to wide-character string.

Otherwise, **_ctime64**, **_wctime**, and **_wctime64** behave identically to **ctime**.

These functions validate their parameters. If *timer* is a null pointer, or if the timer value is negative, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the functions return **NULL** and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tctime	**ctime**	**ctime**	**_wctime**
_tctime32	**_ctime32**	**_ctime32**	**_wctime32**
_tctime64	**_ctime64**	**_ctime64**	**_wctime64**

Requirements

Routine	Required header	Compatibility
ctime	<time.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ctime32	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ctime64	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wctime	<time.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wctime32	<time.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wctime64	<time.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_ctime64.c
// compile with: /W3
/* This program gets the current
 * time in _time64_t form, then uses ctime to
 * display the time in string form.
 */

#include <time.h>
#include <stdio.h>

int main( void )
{
   __time64_t ltime;

   _time64( &ltime );
   printf( "The time is %s\n", _ctime64( &ltime ) ); // C4996
   // Note: _ctime64 is deprecated; consider using _ctime64_s
}
```

Sample Output

```
The time is Wed Feb 13 16:04:43 2002
```

.NET Framework Equivalent

- System::DateTime::GetDateTimeFormats
- System::DateTime::ToString
- System::DateTime::ToLongTimeString
- System::DateTime::ToShortTimeString

See Also

Reference

Time Management
asctime, _wasctime
ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s
_ftime, _ftime32, _ftime64
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
time, _time32, _time64

ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s

Convert a time value to a string and adjust for local time zone settings. These are versions of ctime, _ctime64, _wctime, _wctime64 with security enhancements as described in Security Enhancements in the CRT.

```
errno_t ctime_s(
   char* buffer,
   size_t numberOfElements,
   const time_t *time
);
errno_t _ctime32_s(
   char* buffer,
   size_t numberOfElements,
   const __time32_t *time
);
errno_t _ctime64_s(
   char* buffer,
   size_t numberOfElements,
   const __time64_t *time )
;
errno_t _wctime_s(
   wchar_t* buffer,
   size_t numberOfElements,
   const time_t *time
);
errno_t _wctime32_s(
   wchar_t* buffer,
   size_t numberOfElements,
   const __time32_t *time
);
errno_t _wctime64_s(
   wchar_t* buffer,
   size_t numberOfElements,
   const __time64_t *time
);
template <size_t size>
errno_t _ctime32_s(
   char (&buffer)[size],
   const __time32_t *time
); // C++ only
template <size_t size>
errno_t _ctime64_s(
   char (&buffer)[size],
      const __time64_t *time
); // C++ only
template <size_t size>
errno_t _wctime32_s(
   wchar_t (&buffer)[size],
   const __time32_t *time
); // C++ only
template <size_t size>
errno_t _wctime64_s(
   wchar_t (&buffer)[size],
   const __time64_t *time
); // C++ only
```

Parameters

[out] *buffer*

Must be large enough to hold 26 characters. A pointer to the character string result, or **NULL** if:

- *time* represents a date before midnight, January 1, 1970, UTC.

- If you use **_ctime32_s** or **_wctime32_s** and *time* represents a date after 03:14:07 January 19, 2038.
- If you use **_ctime64_s** or **_wctime64_s** and *time* represents a date after 23:59:59, December 31, 3000, UTC.
- If you use **_ctime_s** or **_wctime_s**, these functions are wrappers to the previous functions. See the Remarks section.

[in] *numberOfElements*

The size of the buffer.

[in] *time*

Pointer to stored time.

Return Value

Zero if successful. If there is a failure due to an invalid parameter, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, an error code is returned. Error codes are defined in ERRNO.H; for a listing of these errors, see errno. The actual error codes thrown for each error condition are shown in the following table.

Error Conditions

buffer	***numberOfElements***	***time***	**Return**	**Value in *buffer***
NULL	any	any	**EINVAL**	Not modified
Not **NULL** (points to valid memory)	0	any	**EINVAL**	Not modified
Not **NULL**	0< size < 26	any	**EINVAL**	Empty string
Not **NULL**	>= 26	NULL	**EINVAL**	Empty string
Not **NULL**	>= 26	< 0	**EINVAL**	Empty string

Remarks

The **ctime_s** function converts a time value stored as a time_t structure into a character string. The *time* value is usually obtained from a call to time, which returns the number of seconds elapsed since midnight (00:00:00), January 1, 1970, coordinated universal time (UTC). The return value string contains exactly 26 characters and has the form:

```
Wed Jan 02 02:03:55 1980\n\0
```

A 24-hour clock is used. All fields have a constant width. The new line character ('\n') and the null character ('\0') occupy the last two positions of the string.

The converted character string is also adjusted according to the local time zone settings. See the **time**, _ftime, and localtime32_s functions for information about configuring the local time and the _tzset function for information about defining the time zone environment and global variables.

_wctime32_s and **_wctime64_s** are the wide-character version of **_ctime32_s** and **_ctime64_s**; returning a pointer to wide-character string. Otherwise, **_ctime64_s**, **_wctime32_s**, and **_wctime64_s** behave identically to **_ctime32_s**.

In Visual C++ 2005, **ctime_s** is an inline function that evaluates to **_ctime64_s** and **time_t** is equivalent to **__time64_t**. If you need to force the compiler to interpret **time_t** as the old 32-bit **time_t**, you can define **_USE_32BIT_TIME_T**. Doing this will cause **ctime_s** to evaluate to **_ctime32_s**. This is not recommended because your application may fail after January 18, 2038, and it is not allowed on 64-bit platforms.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically, eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	**_UNICODE & _MBCS not defined**	**_MBCS defined**	**_UNICODE defined**
_tctime_s	**ctime_s**	**ctime_s**	**_wctime_s**
_tctime32_s	**_ctime32_s**	**_ctime32_s**	**_wctime32_s**

_tctime64_s	_ctime64_s	_ctime64_s	_wctime64_s

Requirements

Routine	Required header	Compatibility
ctime_s, _ctime32_s, _ctime64_s	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wctime_s, _wctime32_s, _wctime64_s	<time.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_wctime_s.c
/* This program gets the current
 * time in time_t form and then uses _wctime_s to
 * display the time in string form.
 */

#include <time.h>
#include <stdio.h>

#define SIZE 26

int main( void )
{
   time_t ltime;
   wchar_t buf[SIZE];
   errno_t err;

   time( &ltime );


   err = _wctime_s( buf, SIZE, &ltime );
   if (err != 0)
   {
      printf("Invalid Arguments for _wctime_s. Error Code: %d\n", err);
   }
   wprintf( L"The time is %s\n", buf );
}
```

Sample Output

```
The time is Fri Apr 25 13:03:39 2003
```

.NET Framework Equivalent

- System::DateTime::GetDateTimeFormats
- System::DateTime::ToString
- System::DateTime::ToLongTimeString
- System::DateTime::ToShortTimeString

See Also

Reference

Time Management
asctime_s, _wasctime_s
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
_ftime, _ftime32, _ftime64
_gmtime_s, _gmtime32_s, _gmtime64_s
localtime_s, _localtime32_s, _localtime64_s
time, _time32, _time64

cwait

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _cwait instead.

_cwait

Waits until another process terminates.

```
intptr_t _cwait(
   int *termstat,
   intptr_t procHandle,
   int action
);
```

Parameters

termstat

Pointer to a buffer where the result code of the specified process will be stored, or NULL.

procHandle

The handle to the process to wait on (the process that has to terminate before **_cwait** can return).

action

NULL: Ignored by Windows operating system applications; for other applications: action code to perform on *procHandle*.

Return Value

When the specified process has successfully completed, returns the handle of the specified process and sets *termstat* to the result code returned by the specified process. Otherwise, returns –1 and sets **errno** as follows.

Value	Description
ECHILD	No specified process exists, *procHandle* is invalid, or the call to the GetExitCodeProcess or WaitForSingleObject API failed.
EINVAL	*action* is invalid.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_cwait** function waits for the termination of the process ID of the specified process that is provided by *procHandle*. The value of *procHandle* passed to **_cwait** should be the value returned by the call to the _spawn function that created the specified process. If the process ID terminates before **_cwait** is called, **_cwait** returns immediately. **_cwait** can be used by any process to wait for any other known process for which a valid handle (*procHandle*) exists.

termstat points to a buffer where the return code of the specified process will be stored. The value of *termstat* indicates whether the specified process terminated normally by calling the Windows NT ExitProcess API. **ExitProcess** is called internally if the specified process calls **exit** or **_exit**, returns from **main**, or reaches the end of **main**. For more information about the value passed back through *termstat*, see **GetExitCodeProcess**. If **_cwait** is called with a NULL value for *termstat*, the return code of the specified process is not stored.

The *action* parameter is ignored by the Windows operating system because parent-child relationships are not implemented in these environments.

Note that unless *procHandle* is -1 or -2 (handles to the current process or thread), the handle will be closed. Therefore, in this situation, do not use the returned handle.

Requirements

Routine	Required header	Optional header	Compatibility
_cwait	<process.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_cwait.c
// compile with: /c
// This program launches several processes and waits
// for a specified process to finish.
//

#include <windows.h>
#include <process.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

// Macro to get a random integer within a specified range
#define getrandom( min, max ) (( rand() % (int)((( max ) + 1 ) - ( min ))) + ( min ))

struct PROCESS
{
   int     nPid;
   char    name[40];
} process[4] = { { 0, "Ann" }, { 0, "Beth" }, { 0, "Carl" }, { 0, "Dave" } };

int main( int argc, char *argv[] )
{
   int termstat, c;

   srand( (unsigned)time( NULL ) );    // Seed randomizer

   // If no arguments, this is the calling process
   if( argc == 1 )
   {
      // Spawn processes in numeric order
      for( c = 0; c < 4; c++ ){
         _flushall();
         process[c].nPid = _spawnl( _P_NOWAIT, argv[0], argv[0],
                                    process[c].name, NULL );
      }

      // Wait for randomly specified process, and respond when done
      c = getrandom( 0, 3 );
      printf( "Come here, %s.\n", process[c].name );
      _cwait( &termstat, process[c].nPid, _WAIT_CHILD );
      printf( "Thank you, %s.\n", process[c].name );

   }
   // If there are arguments, this must be a spawned process
   else
   {
      // Delay for a period determined by process number
      Sleep( (argv[1][0] - 'A' + 1) * 1000L );
      printf( "Hi, Dad. It's %s.\n", argv[1] );
   }
}
```

Output

```
Hi, Dad. It's Ann.
Come here, Ann.
Thank you, Ann.
Hi, Dad. It's Beth.
Hi, Dad. It's Carl.
Hi, Dad. It's Dave.
```

.NET Framework Equivalent

System::Diagnostics::Process::WaitForExit

See Also

Reference

Process and Environment Control

_spawn, _wspawn Functions

difftime, _difftime32, _difftime64

Finds the difference between two times.

```
double difftime(
   time_t timer1,
   time_t timer0
);
double _difftime32(
      __time32_t timer1,
   __time32_t timer0
);
double _difftime64(
   __time64_t timer1,
   __time64_t timer0
);
```

Parameters

timer1

Ending time.

timer0

Beginning time.

Return Value

difftime returns the elapsed time in seconds, from *timer0* to *timer1*. The value returned is a double precision floating-point number. The return value may be 0, indicating an error.

Remarks

The **difftime** function computes the difference between the two supplied time values *timer0* and *timer1*.

The time value supplied must fit within the range of **time_t**. In Visual C++ 2005, **time_t** is a 64-bit value. Thus, the end of the range was extended from 03:14:07 January 19, 2038 to 23:59:59, December 31, 3000. The lower range of **time_t** is still midnight, January 1, 1970.

In Visual C++ 2005, **difftime** is an inline function that evaluates to either **_difftime32** or **_difftime64** depending on whether **_USE_32BIT_TIME_T** is defined. _difftime32 and _difftime64 can be used directly to force the use of a particular size of the time type.

These functions validate their parameters. If either of the parameters is zero or negative, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return 0 and set **errno** to **EINVAL**.

Requirements

Routine	Required header	Compatibility
difftime	<time.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_difftime32	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_difftime64	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_difftime.c
// This program calculates the amount of time
// needed to do a floating-point multiply 500 million times.
//

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main( void )
{
   time_t   start, finish;
   long loop;
   double   result, elapsed_time;

   printf( "Multiplying 2 floating point numbers 500 million times...\n" );

   time( &start );
   for( loop = 0; loop < 500000000; loop++ )
      result = 3.63 * 5.27;
   time( &finish );

   elapsed_time = difftime( finish, start );
   printf( "\nProgram takes %6.0f seconds.\n", elapsed_time );
}
```

Sample Output

```
Multiplying 2 floating point numbers 500 million times...

Program takes      5 seconds.
```

.NET Framework Equivalent

System::DateTime::Subtract

See Also

Reference

Floating-Point Support
Time Management
time, _time32, _time64

div

Computes the quotient and the remainder of two integer values.

```
div_t div(
   int numer,
      int denom
);
ldiv_t div(
      long numer,
   long denom
);
```

Parameters

numer

The numerator.

denom

The denominator.

Return Value

div called with arguments of type **int** returns a structure of type **div_t**, comprising the quotient and the remainder. The return value of the overload with arguments of type **long** is **ldiv_t**. Both **div_t** and **ldiv_t** are defined in STDLIB.H.

Remarks

The **div** function divides *numer* by *denom*, computing the quotient and the remainder. The div_t structure contains **int quot**, the quotient, and **int rem**, the remainder. The sign of the quotient is the same as that of the mathematical quotient. Its absolute value is the largest integer that is less than the absolute value of the mathematical quotient. If the denominator is 0, the program terminates with an error message.

The overload taking arguments of type **long** is only available to C++ code. The return type ldiv_t contains the members **long quot** and **long rem**, which have the same meanings as the members of **div_t**.

Requirements

Routine	Required header	Compatibility
div	<stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_div.c
// arguments: 876 13

// This example takes two integers as command-line
// arguments and displays the results of the integer
// division. This program accepts two arguments on the
// command line following the program name, then calls
// div to divide the first argument by the second.
// Finally, it prints the structure members quot and rem.
//

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main( int argc, char *argv[] )
{
```

```
    int x,y;
    div_t div_result;

    x = atoi( argv[1] );
    y = atoi( argv[2] );

    printf( "x is %d, y is %d\n", x, y );
    div_result = div( x, y );
    printf( "The quotient is %d, and the remainder is %d\n",
            div_result.quot, div_result.rem );
}
```

Output

```
x is 876, y is 13
The quotient is 67, and the remainder is 5
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support

ldiv

dup, dup2

These POSIX functions are deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _dup, _dup2 instead.

_dup, _dup2

Create a second file descriptor for an open file (**_dup**), or reassign a file descriptor (**_dup2**).

```
int _dup(
   int fd
);
int _dup2(
   int fd1,
   int fd2
);
```

Parameters

fd, *fd1*

File descriptors referring to open file.

fd2

Any file descriptor.

Return Value

_dup returns a new file descriptor. **_dup2** returns 0 to indicate success. If an error occurs, each function returns –1 and sets **errno** to **EBADF** if the file descriptor is invalid or to **EMFILE** if no more file descriptors are available. In the case of an invalid file descriptor, the function also invokes the invalid parameter handler, as described in Parameter Validation.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_dup** and **_dup2** functions associate a second file descriptor with a currently open file. These functions can be used to associate a predefined file descriptor, such as that for **stdout**, with a different file. Operations on the file can be carried out using either file descriptor. The type of access allowed for the file is unaffected by the creation of a new descriptor. **_dup** returns the next available file descriptor for the given file. **_dup2** forces *fd2* to refer to the same file as *fd1*. If *fd2* is associated with an open file at the time of the call, that file is closed.

Both **_dup** and **_dup2** accept file descriptors as parameters. To pass a stream **(FILE *)** to either of these functions, use _fileno. The **fileno** routine returns the file descriptor currently associated with the given stream. The following example shows how to associate **stderr** (defined as **FILE *** in Stdio.h) with a file descriptor:

```
cstderr = _dup( _fileno( stderr ));
```

Requirements

Routine	Required header	Compatibility
_dup	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_dup2	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_dup.c
// This program uses the variable old to save
// the original stdout. It then opens a new file named
// DataFile and forces stdout to refer to it. Finally, it
// restores stdout to its original state.
//
```

```
#include <io.h>
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   int old;
   FILE *DataFile;

   old = _dup( 1 );   // "old" now refers to "stdout"
                      // Note:  file descriptor 1 == "stdout"
   if( old == -1 )
   {
      perror( "_dup( 1 ) failure" );
      exit( 1 );
   }
   _write( old, "This goes to stdout first\n", 26 );
   if( fopen_s( &DataFile, "data", "w" ) != 0 )
   {
      puts( "Can't open file 'data'\n" );
      exit( 1 );
   }

   // stdout now refers to file "data"
   if( -1 == _dup2( _fileno( DataFile ), 1 ) )
   {
      perror( "Can't _dup2 stdout" );
      exit( 1 );
   }
   puts( "This goes to file 'data'\n" );

   // Flush stdout stream buffer so it goes to correct file
   fflush( stdout );
   fclose( DataFile );

   // Restore original stdout
   _dup2( old, 1 );
   puts( "This goes to stdout\n" );
   puts( "The file 'data' contains:" );
   _flushall();
   system( "type data" );
}
```

Output

```
This goes to stdout first
This goes to stdout

The file 'data' contains:
This goes to file 'data'
```

See Also

Reference

Low-Level I/O

_close

_creat, _wcreat

_open, _wopen

_dupenv_s, _wdupenv_s

Get a value from the current environment.

```
errno_t _dupenv_s(
   char **buffer,
   size_t *numberOfElements,
   const char *varname
);
errno_t _wdupenv_s(
   wchar_t **buffer,
   size_t * numberOfElements,
   const wchar_t *varname
);
```

Parameters

buffer

Buffer to store the variable's value.

numberOfElements

Size of *buffer*.

varname

Environment variable name.

Return Value

Zero on success, an error code on failure.

These functions validate their parameters; if *buffer* or *varname* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, the functions set **errno** to **EINVAL** and return **EINVAL**.

If these functions cannot allocate enough memory, they set *buffer* to **NULL** and *numberOfElements* to 0, and return **ENOMEM**.

Remarks

The **_dupenv_s** function searches the list of environment variables for *varname*. If the variable is found, a buffer is allocated, the variable's value is copied into the buffer, and the buffer's address and length are returned in *buffer* and *numberOfElements*. By allocating the buffer itself, **_dupenv_s** provides a more convenient alternative to getenv_s, _wgetenv_s.

Note
It is the calling program's responsibility to free the memory by calling free.

If the variable is not found, then *buffer* is set to **NULL**, *numberOfElements* is set to 0, and the return value is 0 because this situation is not considered to be an error condition.

if you are not interested in the size of the buffer you can pass **NULL** for *numberOfElements*.

_dupenv_s is not case sensitive in the Windows operating system. **_dupenv_s** uses the copy of the environment pointed to by the global variable **_environ** to access the environment. See the Remarks in getenv_s, _wgetenv_s for a discussion of **_environ**.

The value in *buffer* is a copy of the environment variable's value; modifying it has no effect on the environment. Use the _putenv_s, _wputenv_s function to modify the value of an environment variable.

_wdupenv_s is a wide-character version of **_dupenv_s**; the arguments of **_wdupenv_s** are wide-character strings. The **_wenviron** global variable is a wide-character version of **_environ**. See the Remarks in getenv_s, _wgetenv_s for more on **_wenviron**.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined

_tdupenv_s	_dupenv_s	_dupenv_s	_wdupenv_s

Requirements

Routine	Required header	Compatibility
_dupenv_s	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wdupenv_s	<stdlib.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_dupenv_s.c
#include  <stdlib.h>

int main( void )
{
   char *pValue;
   size_t len;
   errno_t err = _dupenv_s( &pValue, &len, "pathext" );
   if ( err ) return -1;
   printf( "pathext = %s\n", pValue );
   free( pValue );
   err = _dupenv_s( &pValue, &len, "nonexistentvariable" );
   if ( err ) return -1;
   printf( "nonexistentvariable = %s\n", pValue );
   free( pValue ); // It's OK to call free with NULL
}
```

Sample Output

```
pathext = .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.pl
nonexistentvariable = (null)
```

.NET Framework Equivalent

System::Environment::GetEnvironmentVariable

See Also

Reference

Process and Environment Control
Environmental Constants
_dupenv_s_dbg, _wdupenv_s_dbg
getenv_s, _wgetenv_s
_putenv_s, _wputenv_s

_dupenv_s_dbg, _wdupenv_s_dbg

Get a value from the current environment. Versions of _dupenv_s, _wdupenv_s that allocate memory with _malloc_dbg to provide additional debugging information.

```
errno_t _dupenv_s_dbg(
   char **buffer,
   size_t *numberOfElements,
   const char *varname,
   int blockType,
   const char *filename,
   int linenumber
);
errno_t _wdupenv_s_dbg(
   wchar_t **buffer,
   size_t * numberOfElements,
   const wchar_t *varname,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

buffer

Buffer to store the variable's value.

numberOfElements

Size of *buffer*.

varname

Environment variable name.

blockType

Requested type of the memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

filename

Pointer to the name of the source file or **NULL**.

linenumber

Line number in source file or **NULL**.

Return Value

Zero on success, an error code on failure.

These functions validate their parameters; if *buffer* or *varname* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, the functions set **errno** to **EINVAL** and return **EINVAL**.

If these functions cannot allocate enough memory, they set *buffer* to **NULL** and *numberOfElements* to 0, and return **ENOMEM**.

Remarks

The **_dupenv_s_dbg** and **_wdupenv_s_dbg** functions are identical to **_dupenv_s** and **_wdupenv_s** except that, when **_DEBUG** is defined, these functions use the debug version of malloc, _malloc_dbg, to allocate memory for the value of the environment variable. For information on the debugging features of **_malloc_dbg**, see _malloc_dbg.

You do not need to call these functions explicitly in most cases. Instead, you can define the flag **_CRTDBG_MAP_ALLOC**. When **_CRTDBG_MAP_ALLOC** is defined, calls to **_dupenv_s** and **_wdupenv_s** are remapped to **_dupenv_s_dbg** and **_wdupenv_s_dbg**, respectively, with the *blockType* set to **_NORMAL_BLOCK**. Thus, you do not need to call these functions explicitly unless you want to mark the heap blocks as **_CLIENT_BLOCK**. For more information on block types, see Types of Blocks on the Debug Heap.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tdupenv_s_dbg	**_dupenv_s_dbg**	**_dupenv_s_dbg**	**_wdupenv_s_dbg**

Requirements

Routine	Required header	Compatibility
_dupenv_s_dbg	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wdupenv_s_dbg	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_dupenv_s_dbg.c
#include  <stdlib.h>
#include <crtdbg.h>

int main( void )
{
   char *pValue;
   size_t len;
   errno_t err = _dupenv_s_dbg( &pValue, &len, "pathext",
      _NORMAL_BLOCK, __FILE__, __LINE__ );
   if ( err ) return -1;
   printf( "pathext = %s\n", pValue );
   free( pValue );
   err = _dupenv_s_dbg( &pValue, &len, "nonexistentvariable",
      _NORMAL_BLOCK, __FILE__, __LINE__ );
   if ( err ) return -1;
   printf( "nonexistentvariable = %s\n", pValue );
   free( pValue ); // It's OK to call free with NULL
}
```

Sample Output

```
pathext = .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.pl
nonexistentvariable = (null)
```

.NET Framework Equivalent

System::Environment::GetEnvironmentVariable

See Also

Reference

Process and Environment Control
Environmental Constants
getenv_s, _wgetenv_s
_putenv_s, _wputenv_s

ecvt

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _ecvt or security-enhanced _ecvt_s instead.

_ecvt

Converts a **double** number to a string. This function is deprecated because a more secure version is available; see _ecvt_s.

```
char *_ecvt(
   double value,
   int count,
   int *dec,
   int *sign
);
```

Parameters

value

Number to be converted.

count

Number of digits stored.

dec

Stored decimal-point position.

sign

Sign of the converted number.

Return Value

_ecvt returns a pointer to the string of digits; NULL if an error occurred.

Remarks

The **_ecvt** function converts a floating-point number to a character string. The *value* parameter is the floating-point number to be converted. This function stores up to *count* digits of *value* as a string and appends a null character ('\0'). If the number of digits in *value* exceeds *count*, the low-order digit is rounded. If there are fewer than *count* digits, the string is padded with zeros.

The total number of digits returned by **_ecvt** will not exceed **_CVTBUFSIZE**.

Only digits are stored in the string. The position of the decimal point and the sign of *value* can be obtained from *dec* and *sign* after the call. The *dec* parameter points to an integer value giving the position of the decimal point with respect to the beginning of the string. A 0 or negative integer value indicates that the decimal point lies to the left of the first digit. The *sign* parameter points to an integer that indicates the sign of the converted number. If the integer value is 0, the number is positive. Otherwise, the number is negative.

The difference between **_ecvt** and **_fcvt** is in the interpretation of the *count* parameter. **_ecvt** interprets *count* as the total number of digits in the output string, whereas **_fcvt** interprets *count* as the number of digits after the decimal point.

_ecvt and **_fcvt** use a single statically allocated buffer for the conversion. Each call to one of these routines destroys the result of the previous call.

This function validates its parameters. If *dec* or *sign* is NULL, or *count* is 0, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and NULL is returned.

Requirements

Function	Required header	Compatibility
_ecvt	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_ecvt.c
// compile with: /W3
// This program uses _ecvt to convert a
// floating-point number to a character string.

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   int     decimal,   sign;
   char    *buffer;
   int     precision = 10;
   double  source = 3.1415926535;

   buffer = _ecvt( source, precision, &decimal, &sign ); // C4996
   // Note: _ecvt is deprecated; consider using _ecvt_s instead
   printf( "source: %2.10f   buffer: '%s'  decimal: %d  sign: %d\n",
           source, buffer, decimal, sign );
}
```

Output

```
source: 3.1415926535   buffer: '3141592654'  decimal: 1  sign: 0
```

.NET Framework Equivalent

System::Convert::ToString

See Also

Reference

Data Conversion
Floating-Point Support
atof, _atof_l, _wtof, _wtof_l
_fcvt
_gcvt

_ecvt_s

Converts a **double** number to a string. This is a version of _ecvt with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _ecvt_s(
   char * _Buffer,
   size_t _SizeInBytes,
   double _Value,
   int _Count,
   int *_Dec,
   int *_Sign
);
template <size_t size>
errno_t _ecvt_s(
   char (&_Buffer)[size],
   double _Value,
   int _Count,
   int *_Dec,
   int *_Sign
); // C++ only
```

Parameters

[out] *_Buffer*

Filled with the pointer to the string of digits, the result of the conversion.

[in] *_SizeInBytes*

Size of the buffer in bytes.

[in] *_Value*

Number to be converted.

[in] *_Count*

Number of digits stored.

[out] *_Dec*

Stored decimal-point position.

[out] *_Sign*

Sign of the converted number.

Return Value

Zero if successful. The return value is an error code if there is a failure. Error codes are defined in Errno.h. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

In the case of an invalid parameter, as listed in the following table, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Error Conditions

_Buffer	***_SizeInBytes***	**_Value**	**_Count**	**_Dec**	**_Sign**	**Return value**	**Value in *buffer***
NULL	any	any	any	any	any	**EINVAL**	Not modified.
Not **NULL** (points to valid memory)	<=0	any	any	any	any	**EINVAL**	Not modified.
any	any	any	any	**NULL**	any	**EINVAL**	Not modified.
any	any	any	any	any	**NULL**	**EINVAL**	Not modified.

Security Issues

_ecvt_s might generate an access violation if *buffer* does not point to valid memory and is not **NULL**.

Remarks

The **_ecvt_s** function converts a floating-point number to a character string. The *_Value* parameter is the floating-point number to be converted. This function stores up to *count* digits of *_Value* as a string and appends a null character ('\0'). If the number of digits in *_Value* exceeds *_Count*, the low-order digit is rounded. If there are fewer than *count* digits, the string is padded with zeros.

Only digits are stored in the string. The position of the decimal point and the sign of *_Value* can be obtained from *_Dec* and *_Sign* after the call. The *_Dec* parameter points to an integer value giving the position of the decimal point with respect to the beginning of the string. A 0 or negative integer value indicates that the decimal point lies to the left of the first digit. The *_Sign* parameter points to an integer that indicates the sign of the converted number. If the integer value is 0, the number is positive. Otherwise, the number is negative.

A buffer of length **_CVTBUFSIZE** is sufficient for any floating-point value.

The difference between **_ecvt_s** and **_fcvt_s** is in the interpretation of the *_Count* parameter. **_ecvt_s** interprets *_Count* as the total number of digits in the output string, whereas **_fcvt_s** interprets *_Count* as the number of digits after the decimal point.

In C++, using this function is simplified by a template overload; the overload can infer buffer length automatically, eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

The debug version of this function first fills the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Requirements

Function	Required header	Optional header	Compatibility
_ecvt_s	<stdlib.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// ecvt_s.c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main( )
{
  char * buf = 0;
  int decimal;
  int sign;
  int err;

  buf = (char*) malloc(_CVTBUFSIZE);
  err = _ecvt_s(buf, _CVTBUFSIZE, 1.2, 5, &decimal, &sign);

  if (err != 0)
  {
     printf("_ecvt_s failed with error code %d\n", err);
     exit(1);
  }

  printf("Converted value: %s\n", buf);

}
```

Output

```
Converted value: 12000
```

.NET Framework Equivalent

ToString

See Also

Reference

Data Conversion

Floating-Point Support

atof, _atof_l, _wtof, _wtof_l

_ecvt

_fcvt_s

_gcvt_s

_endthread, _endthreadex

Terminates a thread created by **_beginthread** or **_beginthreadex**, respectively.

```
void _endthread( void );
void _endthreadex(
   unsigned retval
);
```

Parameters

retval

Thread exit code.

Remarks

You can call **_endthread** or **_endthreadex** explicitly to terminate a thread; however, **_endthread** or **_endthreadex** is called automatically when the thread returns from the routine passed as a parameter to **_beginthread** or **_beginthreadex**. Terminating a thread with a call to **endthread** or **_endthreadex** helps ensure proper recovery of resources allocated for the thread.

Note
For an executable file linked with Libcmt.lib, do not call the Win32 ExitThread API; this prevents the run-time system from recl aiming allocated resources. **_endthread** and **_endthreadex** reclaim allocated thread resources and then call **ExitThread**.

_endthread automatically closes the thread handle. (This behavior differs from the Win32 **ExitThread** API.) Therefore, when you use **_beginthread** and **_endthread**, do not explicitly close the thread handle by calling the Win32 CloseHandle API.

Like the Win32 **ExitThread** API, **_endthreadex** does not close the thread handle. Therefore, when you use **_beginthreadex** and **_endthreadex**, you must close the thread handle by calling the Win32 **CloseHandle** API.

Note
_endthread and **_endthreadex** cause C++ destructors pending in the thread not to be called.

Requirements

Function	**Required header**	**Compatibility**
_endthread	<process.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_endthreadex	<process.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Multithreaded versions of the C run-time libraries only.

Example

See the example for _beginthread.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Process and Environment Control

_beginthread, _beginthreadex

eof

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _eof instead.

_eof

Tests for end of file (EOF).

```
int _eof(
   int fd
);
```

Parameters

fd

File descriptor referring to the open file.

Return Value

_eof returns 1 if the current position is end of file, or 0 if it is not. A return value of –1 indicates an error; in this case, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EBADF**, which indicates an invalid file descriptor.

Remarks

The **_eof** function determines whether the end of the file associated with *fd* has been reached.

Requirements

Function	Required header	Optional header	Compatibility
_eof	<io.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_eof.c
// This program reads data from a file
// ten bytes at a time until the end of the
// file is reached or an error is encountered.
//
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <share.h>

int main( void )
{
   int  fh, count, total = 0;
   char buf[10];
   if( _sopen_s( &fh, "crt_eof.txt", _O_RDONLY, _SH_DENYNO, 0 ) )
   {
        perror( "Open failed");
        exit( 1 );
   }
   // Cycle until end of file reached:
   while( !_eof( fh ) )
   {
      // Attempt to read in 10 bytes:
      if( (count = _read( fh, buf, 10 )) == -1 )
      {
         perror( "Read error" );
         break;
      }
      // Total actual bytes read
```

```
        total += count;
    }
    printf( "Number of bytes read = %d\n", total );
    _close( fh );
}
```

Input: crt_eof.txt

```
This file contains some text.
```

Output

```
Number of bytes read = 29
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Error Handling (CRT)
Low-Level I/O
clearerr
feof
ferror
perror, _wperror

_exec, _wexec Functions

Each function in this family loads and executes a new process:

_execl, _wexecl	_execv, _wexecv
_execle, _wexecle	_execve, _wexecve
_execlp, _wexeclp	_execvp, _wexecvp
_execlpe, _wexeclpe	_execvpe, _wexecvpe

The letter at the end of the function name determines the variation.

_exec function suffix	**Description**
e	*envp*, array of pointers to environment settings, is passed to the new process.
l	Command-line arguments are passed individually to **_exec** function. Typically used when the number of parameters to the new process is known in advance.
p	**PATH** environment variable is used to find the file to execute.
v	*argv*, array of pointers to command-line arguments, is passed to **_exec**. Typically used when the number of parameters to the new process is variable.

Remarks

Each **_exec** function loads and executes a new process. All **_exec** functions use the same operating-system function. The **_exec** functions automatically handle multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the multibyte code page currently in use. The **_wexec** functions are wide-character versions of the **_exec** functions. The **_wexec** functions behave identically to their **_exec** family counterparts except that they do not handle multibyte-character strings.

Generic-Text Routine Mappings

Tchar.h routine	**_UNICODE and _MBCS not defined**	**_MBCS defined**	**_UNICODE defined**
_texecl	**_execl**	**_execl**	**_wexecl**
_texecle	**_execle**	**_execle**	**_wexecle**
_texeclp	**_execlp**	**_execlp**	**_wexeclp**
_texeclpe	**_execlpe**	**_execlpe**	**_wexeclpe**
_texecv	**_execv**	**_execv**	**_wexecv**
_texecve	**_execve**	**_execve**	**_wexecve**
_texecvp	**_execvp**	**_execvp**	**_wexecvp**
_texecvpe	**_execvpe**	**_execvpe**	**_wexecvpe**

When a call to an **_exec** function is successful, the new process is placed in the memory previously occupied by the calling process. Sufficient memory must be available for loading and executing the new process.

The *cmdname* parameter specifies the file to be executed as the new process. It can specify a full path (from the root), a partial

path (from the current working directory), or a file name. If *cmdname* does not have a file name extension or does not end with a period (.), the **_exec** function searches for the named file. If the search is unsuccessful, it tries the same base name with the .com file name extension and then with the .exe, .bat, and .cmd file name extensions. If *cmdname* has a file name extension, only that extension is used in the search. If *cmdname* ends with a period, the **_exec** function searches for *cmdname* with no file name extension. **_execlp**, **_execlpe**, **_execvp**, and **_execvpe** search for *cmdname* (using the same procedures) in the directories specified by the **PATH** environment variable. If *cmdname* contains a drive specifier or any slashes (that is, if it is a relative path), the **_exec** call searches only for the specified file; the path is not searched.

Parameters are passed to the new process by giving one or more pointers to character strings as parameters in the **_exec** call. These character strings form the parameter list for the new process. The combined length of the inherited environment settings and the strings forming the parameter list for the new process must not exceed 32 kilobytes. The terminating null character ('\0') for each string is not included in the count, but space characters (inserted automatically to separate the parameters) are counted.

Note

Spaces embedded in strings may cause unexpected behavior; for example, passing **_exec** the string `"hi there"` will result in the new process getting two arguments, `"hi"` and `"there"`. If the intent was to have the new process open a file named "hi th ere", the process would fail. You can avoid this by quoting the string: `"\"hi there\""`.

Security Note

Do not pass user input to **_exec** without explicitly checking its content. **_exec** will result in a call to CreateProcess so keep in mind that unqualified path names could lead to potential security vulnerabilities.

In Visual C++ 2005, the **_exec** functions validate their parameters. If expected parameters are null pointers, empty strings, or omitted, the **_exec** functions invoke the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1. No new process is executed.

The argument pointers can be passed as separate parameters (in **_execl**, **_execle**, **_execlp**, and **_execlpe**) or as an array of pointers (in **_execv**, **_execve**, **_execvp**, and **_execvpe**). At least one parameter, *arg0*, must be passed to the new process; this parameter is *argv*[0] of the new process. Usually, this parameter is a copy of *cmdname*. (A different value does not produce an error.)

The **_execl**, **_execle**, **_execlp**, and **_execlpe** calls are typically used when the number of parameters is known in advance. The parameter *arg0* is usually a pointer to *cmdname*. The parameters *arg1* through *argn* point to the character strings forming the new parameter list. A null pointer must follow *argn* to mark the end of the parameter list.

The **_execv**, **_execve**, **_execvp**, and **_execvpe** calls are useful when the number of parameters to the new process is variable. Pointers to the parameters are passed as an array, *argv*. The parameter *argv*[0] is usually a pointer to *cmdname*. The parameters *argv*[1] through *argv*[*n*] point to the character strings forming the new parameter list. The parameter *argv*[*n*+1] must be a **NULL** pointer to mark the end of the parameter list.

Files that are open when an **_exec** call is made remain open in the new process. In **_execl**, **_execlp**, **_execv**, and **_execvp** calls, the new process inherits the environment of the calling process. **_execle**, **_execlpe**, **_execve**, and **_execvpe** calls alter the environment for the new process by passing a list of environment settings through the *envp* parameter. *envp* is an array of character pointers, each element of which (except for the final element) points to a null-terminated string defining an environment variable. Such a string usually has the form *NAME=value* where *NAME* is the name of an environment variable and *value* is the string value to which that variable is set. (Note that *value* is not enclosed in double quotation marks.) The final element of the *envp* array should be **NULL**. When *envp* itself is **NULL**, the new process inherits the environment settings of the calling process.

A program executed with one of the **_exec** functions is always loaded into memory as if the maximum allocation field in the program's .exe file header were set to the default value of 0xFFFFH.

The **_exec** calls do not preserve the translation modes of open files. If the new process must use files inherited from the calling process, use the _setmode routine to set the translation mode of these files to the desired mode. You must explicitly flush (using **fflush** or **_flushall**) or close any stream before the **_exec** function call. Signal settings are not preserved in new processes that are created by calls to **_exec** routines. The signal settings are reset to the default in the new process.

Example

```
// crt_args.c
// Illustrates the following variables used for accessing
// command-line arguments and environment variables:
// argc  argv  envp
```

```
// This program will be executed by crt_exec which follows.


#include <stdio.h>

int main( int argc,  // Number of strings in array argv
 char *argv[],       // Array of command-line argument strings
 char **envp )       // Array of environment variable strings
{
    int count;

    // Display each command-line argument.
    printf( "\nCommand-line arguments:\n" );
    for( count = 0; count < argc; count++ )
        printf( "  argv[%d]   %s\n", count, argv[count] );

    // Display each environment variable.
    printf( "\nEnvironment variables:\n" );
    while( *envp != NULL )
        printf( "  %s\n", *(envp++) );

    return;
}
```

Run the following program to execute Crt_args.exe:

```
// crt_exec.c
// Illustrates the different versions of exec, including
//      _execl          _execle          _execlp          _execlpe
//      _execv          _execve          _execvp          _execvpe
//
// Although CRT_EXEC.C can exec any program, you can verify how
// different versions handle arguments and environment by
// compiling and specifying the sample program CRT_ARGS.C. See
// "_spawn, _wspawn Functions" for examples of the similar spawn
// functions.


#include <stdio.h>
#include <conio.h>
#include <process.h>

char *my_env[] =     // Environment for exec?e
{
   "THIS=environment will be",
   "PASSED=to new process by",
   "the EXEC=functions",
   NULL
};

int main( int ac, char* av[] )
{
   char *args[4];
   int ch;
   if( ac != 3 ){
      fprintf( stderr, "Usage: %s <program> <number (1-8)>\n", av[0] );
      return;
   }

   // Arguments for _execv?
   args[0] = av[1];
   args[1] = "exec??";
   args[2] = "two";
   args[3] = NULL;

   switch( atoi( av[2] ) )
   {
```

```
    case 1:
       _execl( av[1], av[1], "_execl", "two", NULL );
       break;
    case 2:
       _execle( av[1], av[1], "_execle", "two", NULL, my_env );
       break;
    case 3:
       _execlp( av[1], av[1], "_execlp", "two", NULL );
       break;
    case 4:
       _execlpe( av[1], av[1], "_execlpe", "two", NULL, my_env );
       break;
    case 5:
       _execv( av[1], args );
       break;
    case 6:
       _execve( av[1], args, my_env );
       break;
    case 7:
       _execvp( av[1], args );
       break;
    case 8:
       _execvpe( av[1], args, my_env );
       break;
    default:
       break;
    }

    // This point is reached only if exec fails.
    printf( "\nProcess was not execed." );
    exit( 0 );
 }
```

.NET Framework Equivalent

- System::Diagnostics::Process Class
- System::Diagnostics::ProcessStartInfo Class

See Also

Reference

Process and Environment Control
abort
atexit
exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

execl

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _execl instead.

_execl, _wexecl

Loads and executes new child processes.

```
intptr_t _execl(
   const char *cmdname,
   const char *arg0,
   ... const char *argn,
   NULL
);
intptr_t _wexecl(
   const wchar_t *cmdname,
   const wchar_t *arg0,
   ... const wchar_t *argn,
   NULL
);
```

Parameters

cmdname

Path of the file to be executed.

arg0, ... *argn*

List of pointers to the parameters.

Return Value

If successful, these functions do not return to the calling process. A return value of –1 indicates an error, in which case the **errno** global variable is set.

errno value	Description
E2BIG	The space required for the arguments and environment settings exceeds 32 KB.
EACCES	The specified file has a locking or sharing violation.
EINVAL	Invalid parameter (one or more of the parameters was a null pointer or empty string).
EMFILE	Too many files open (the specified file must be opened to determine whether it is executable).
ENOENT	The file or path is not found.
ENOEXEC	The specified file is not executable or has an invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process; the available memory has been corrupted; or an invalid b lock exists, indicating that the calling process was not allocated properly.

Remarks

Each of these functions loads and executes a new process, passing each command-line argument as a separate parameter. The first argument is the command or executable file name, and the second argument should be the same as the first. It becomes `argv[0]` in the executed process. The third argument is the first argument, `argv[1]`, of the process being executed.

In Visual C++ 2005, the **_execl** functions validate their parameters. If either *cmdname* or *arg0* is a null pointer or empty string, these functions invoke the invalid parameter handler as described in Parameter Validation If execution is allowed to continue,

these functions set **errno** to **EINVAL** and return -1. No new process is executed.

Requirements

Function	Required header	Optional header	Compatibility
_execl	<process.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wexecl	<process.h> or <wchar.h>	<errno.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example in _exec, _wexec Functions.

.NET Framework Equivalent

- System::Diagnostics::Process Class
- System::Diagnostics::ProcessStartInfo Class

See Also

Reference

Process and Environment Control
_exec, _wexec Functions
abort
atexit
exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

execle

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _execle instead.

_execle, _wexecle

Loads and executes new child processes.

```
intptr_t _execle(
   const char *cmdname,
   const char *arg0,
   ... const char *argn,
   NULL,
   const char *const *envp
);
intptr_t _wexecle(
   const wchar_t *cmdname,
   const wchar_t *arg0,
   ... const wchar_t *argn,
   NULL,
   const char *const *envp
);
```

Parameters

cmdname

Path of the file to execute.

arg0, ... *argn*

List of pointers to parameters.

envp

Array of pointers to environment settings.

Return Value

If successful, these functions do not return to the calling process. A return value of –1 indicates an error, in which case the **errno** global variable is set.

errno value	Description
E2BIG	The space required for the arguments and environment settings exceeds 32 KB.
EACCES	The specified file has a locking or sharing violation.
EINVAL	Invalid parameter.
EMFILE	Too many files open (the specified file must be opened to determine whether it is executable).
ENOENT	The file or path is not found.
ENOEXEC	The specified file is not executable or has an invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process; the available memory has been corrupted; or an invalid block exists, indicating that the calling process was not allocated properly.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions loads and executes a new process, passing each command-line argument as a separate parameter and passing an array of pointers to environment settings.

In Visual C++ 2005, the **_execle** functions validate their parameters. If *cmdname* or *arg0* is a null pointer or empty string, these functions invoke the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1. No new process is launched.

Requirements

Function	**Required header**	**Optional header**	**Compatibility**
_execle	<process.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wexecle	<process.h> or <wchar.h>	<errno.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example in _exec, _wexec Functions.

.NET Framework Equivalent

- System::Diagnostics::Process Class
- System::Diagnostics::ProcessStartInfo Class

See Also

Reference

Process and Environment Control
_exec, _wexec Functions
abort
atexit
exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

execlp

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _execlp instead.

_execlp, _wexeclp

Loads and executes new child processes.

```
intptr_t _execlp(
   const char *cmdname,
   const char *arg0,
   ... const char *argn,
   NULL
);
intptr_t _wexeclp(
   const wchar_t *cmdname,
   const wchar_t *arg0,
   ... const wchar_t *argn,
   NULL
);
```

Parameters

cmdname

Path of the file to execute.

arg0, ... argn

List of pointers to parameters.

Return Value

If successful, these functions do not return to the calling process. A return value of –1 indicates an error, in which case the **errno** global variable is set.

errno value	Description
E2BIG	The space required for the arguments and environment settings exceeds 32 KB.
EACCES	The specified file has a locking or sharing violation.
EINVAL	Invalid parameter.
EMFILE	Too many files open (the specified file must be opened to determine whether it is executable).
ENOENT	The file or path not found.
ENOEXEC	The specified file is not executable or has an invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process; the available memory has been corrupted; or an invalid block exists, indicating that the calling process was not allocated properly.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions loads and executes a new process, passing each command-line argument as a separate parameter and using the **PATH** environment variable to find the file to execute.

In Visual C++ 2005, the **_execlp** functions validate their parameters. If *cmdname* or *arg0* is a null pointer or empty string, these functions invoke the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1. No new process is launched.

Requirements

Function	Required header	Optional header	Compatibility
_execlp	<process.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wexclp	<process.h> or <wchar.h>	<errno.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example in _exec, _wexec Functions.

.NET Framework Equivalent

- System::Diagnostics::Process Class
- System::Diagnostics::ProcessStartInfo Class

See Also

Reference

Process and Environment Control
_exec, _wexec Functions
abort
atexit
exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

execlpe

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _execlpe instead.

_execlpe, _wexeclpe

Loads and executes new child processes.

```
intptr_t _execlpe(
   const char *cmdname,
   const char *arg0,
   ... const char *argn,
   NULL,
   const char *const *envp
);
intptr_t _wexeclpe(
   const wchar_t *cmdname,
   const wchar_t *arg0,
   ... const wchar_t *argn,
   NULL,
   const wchar_t *const *envp
);
```

Parameters

cmdname

Path of the file to execute.

arg0, ... *argn*

List of pointers to parameters.

envp

Array of pointers to environment settings.

Return Value

If successful, these functions do not return to the calling process. A return value of –1 indicates an error, in which case the **errno** global variable is set.

errno value	Description
E2BIG	The space required for the arguments and environment settings exceeds 32 KB.
EACCES	The specified file has a locking or sharing violation.
EINVAL	Invalid parameter.
EMFILE	Too many files open (the specified file must be opened to determine whether it is executable).
ENOENT	The file or path not found.
ENOEXEC	The specified file is not executable or has an invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process; the available memory has been corrupted; or an invalid block exists, indicating that the calling process was not allocated properly.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions loads and executes a new process, passing each command-line argument as a separate parameter and also passing an array of pointers to environment settings. These functions use the **PATH** environment variable to find the file to execute.

Beginning in Visual C++ 2005, the **_execlpe** functions validate their parameters. If either *cmdname* or *arg0* is a null pointers or empty string, these functions invoke the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1. No new process is launched.

Requirements

Function	**Required header**	**Optional header**	**Compatibility**
_execlpe	<process.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wexclpe	<process.h> or <wchar.h>	<errno.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example in _exec, _wexec Functions.

.NET Framework Equivalent

- System::Diagnostics::Process Class
- System::Diagnostics::ProcessStartInfo Class

See Also

Reference

Process and Environment Control
_exec, _wexec Functions
abort
atexit
exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

execv

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _execv instead.

_execv, _wexecv

Loads and executes new child processes.

```
intptr_t _execv(
   const char *cmdname,
   const char *const *argv
);
intptr_t _wexecv(
   const wchar_t *cmdname,
   const wchar_t *const *argv
);
```

Parameters

cmdname

Path of the file to execute.

argv

Array of pointers to parameters.

Return Value

If successful, these functions do not return to the calling process. A return value of –1 indicates an error, in which case the **errno** global variable is set.

errno value	Description
E2BIG	The space required for the arguments and environment settings exceeds 32 KB.
EACCES	The specified file has a locking or sharing violation.
EINVAL	Invalid parameter.
EMFILE	Too many files open (the specified file must be opened to determine whether it is executable).
ENOENT	The file or path not found.
ENOEXEC	The specified file is not executable or has an invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process; the available memory has been corrupted; or an invalid block exists, indicating that the calling process was not allocated properly.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions loads and executes a new process, passing an array of pointers to command-line arguments.

In Visual C++ 2005, the **_execv** functions validate their parameters. If *cmdname* is a null pointer, or if *argv* is a null pointer, pointer to an empty array, or if the array contains an empty string as the first argument, the **_execv** functions invoke the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1. No process is launched.

Requirements

Function	Required header	Optional header	Compatibility
_execv	<process.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wexecv	<process.h> or <wchar.h>	<errno.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example in _exec, _wexec Functions.

.NET Framework Equivalent

- System::Diagnostics::Process Class
- System::Diagnostics::ProcessStartInfo Class

See Also

Reference

Process and Environment Control
_exec, _wexec Functions
abort
atexit
exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

execve

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _execve instead.

_execve, _wexecve

Loads and executes new child processes.

```
intptr_t _execve(
   const char *cmdname,
   const char *const *argv,
   const char *const *envp
);
intptr_t _wexecve(
   const wchar_t *cmdname,
   const wchar_t *const *argv,
   const wchar_t *const *envp
);
```

Parameters

cmdname

Path of the file to execute.

argv

Array of pointers to parameters.

envp

Array of pointers to environment settings.

Return Value

If successful, these functions do not return to the calling process. A return value of –1 indicates an error, in which case the **errno** global variable is set.

errno value	Description
E2BIG	The space required for the arguments and environment settings exceeds 32 KB.
EACCES	The specified file has a locking or sharing violation.
EINVAL	Invalid parameter.
EMFILE	Too many files open (the specified file must be opened to determine whether it is executable).
ENOENT	The file or path not found.
ENOEXEC	The specified file is not executable or has an invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process; the available memory has been corrupted; or an invalid b lock exists, indicating that the calling process was not allocated properly.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions loads and executes a new process, passing an array of pointers to command-line arguments and an array of pointers to environment settings.

In Visual C++ 2005, **_execve** and **_wexecve** validate their parameters. If *cmdname* is a null pointer, or if *argv* is a null pointer, pointer to an empty array, or if the array contains an empty string as the first argument, these functions invoke the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1. No process is launched.

Requirements

Function	**Required header**	**Optional header**	**Compatibility**
_execve	<process.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wexecve	<process.h> or <wchar.h>	<errno.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example in _exec, _wexec Functions.

.NET Framework Equivalent

- System::Diagnostics::Process Class
- System::Diagnostics::ProcessStartInfo Class

See Also

Reference

Process and Environment Control
_exec, _wexec Functions
abort
atexit
exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

execvp

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _execvp instead.

_execvp, _wexecvp

Loads and executes new child processes.

```
intptr_t _execvp(
   const char *cmdname,
   const char *const *argv
);
intptr_t _wexecvp(
   const wchar_t *cmdname,
   const wchar_t *const *argv
);
```

Parameters

cmdname

Path of the file to execute.

argv

Array of pointers to parameters.

Return Value

If successful, these functions do not return to the calling process. A return value of –1 indicates an error, in which case the **errno** global variable is set.

errno value	Description
E2BIG	The space required for the arguments and environment settings exceeds 32 KB.
EACCES	The specified file has a locking or sharing violation.
EINVAL	Invalid parameter.
EMFILE	Too many files open (the specified file must be opened to determine whether it is executable).
ENOENT	The file or path not found.
ENOEXEC	The specified file is not executable or has an invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process; the available memory has been corrupted; or an invalid block exists, indicating that the calling process was not allocated properly.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions loads and executes a new process, passing an array of pointers to command-line arguments and using the **PATH** environment variable to find the file to execute.

In Visual C++ 2005, the **_execvp** functions validate their parameters. If the *cmdname* is a null pointer, or *argv* is a null pointer, pointer to an empty array, or if the array contains an empty string as the first argument, these functions invoke the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1. No process is launched.

Requirements

Function	Required header	Optional header	Compatibility
_execvp	<process.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wexecvp	<process.h> or <wchar.h>	<errno.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example in _exec, _wexec Functions.

.NET Framework Equivalent

- System::Diagnostics::Process Class
- System::Diagnostics::ProcessStartInfo Class

See Also

Reference

Process and Environment Control
_exec, _wexec Functions
abort
atexit
exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

execvpe

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _execvpe instead.

_execvpe, _wexecvpe

Loads and executes new child processes.

```
intptr_t _execvpe(
   const char *cmdname,
   const char *const *argv,
   const char *const *envp
);
intptr_t _wexecvpe(
   const wchar_t *cmdname,
   const wchar_t *const *argv,
   const wchar_t *const *envp
);
```

Parameters

cmdname

Path of the file to execute.

argv

Array of pointers to parameters.

envp

Array of pointers to environment settings.

Return Value

If successful, these functions do not return to the calling process. A return value of –1 indicates an error, in which case the **errno** global variable is set.

errno value	Description
E2BIG	The space required for the arguments and environment settings exceeds 32 KB.
EACCES	The specified file has a locking or sharing violation.
EMFILE	Too many files open (the specified file must be opened to determine whether it is executable).
ENOENT	The file or path not found.
ENOEXEC	The specified file is not executable or has an invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process; the available memory has been corrupted; or an invalid block exists, indicating that the calling process was not allocated properly.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions loads and executes a new process, passing an array of pointers to command-line arguments and an array of pointers to environment settings. These functions use the **PATH** environment variable to find the file to execute.

In Visual C++ 2005, the **_execvpe** functions validate their parameters. If the *cmdname* is a null pointer, or if *argv* is a null pointer, pointer to an empty array, or if the array contains an empty string as the first argument, these functions invoke the

invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1. No process is launched.

Requirements

Function	Required header	Optional header	Compatibility
_execvpe	<process.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wexecvpe	<process.h> or <wchar.h>	<errno.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example in _exec, _wexec Functions.

.NET Framework Equivalent

- System::Diagnostics::Process Class
- System::Diagnostics::ProcessStartInfo Class

See Also

Reference

Process and Environment Control
_exec, _wexec Functions
abort
atexit
exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

exit, _exit

Terminate the calling process after cleanup (**exit**) or immediately (**_exit**).

```
void exit(
   int status
);
void _exit(
   int status
);
```

Parameters

status

Exit status.

Remarks

The **exit** and **_exit** functions terminate the calling process. **exit** calls, in last-in-first-out (LIFO) order, the functions registered by **atexit** and **_onexit**, then flushes all file buffers before terminating the process. **_exit** terminates the process without processing **atexit** or **_onexit** or flushing stream buffers. The *status* value is typically set to 0 to indicate a normal exit and set to some other value to indicate an error.

Although the **exit** and **_exit** calls do not return a value, the low-order byte of *status* is made available to the waiting calling process, if one exists, after the calling process exits. The *status* value is available to the operating-system batch command **ERRORLEVEL** and is represented by one of two constants: **EXIT_SUCCESS**, which represents a value of 0, or **EXIT_FAILURE**, which represents a value of 1. The behavior of **exit**, **_exit**, **_cexit**, and **_c_exit** is as follows.

Function	Description
exit	Performs complete C library termination procedures, terminates the process, and exits with the supplied status code.
_exit	Performs quick C library termination procedures, terminates the process, and exits with the supplied status code.
_cexit	Performs complete C library termination procedures and returns to the caller, but does not terminate the process.
_c_exit	Performs quick C library termination procedures and returns to the caller, but does not terminate the process.

When you call the **exit** or **_exit** functions, the destructors for any temporary or automatic objects that exist at the time of the call are not called. An automatic object is an object that is defined in a function where the object is not declared to be static. A temporary object is an object created by the compiler. To destroy an automatic object before calling **exit** or **_exit**, explicitly call the destructor for the object, as follows:

```
myObject.myClass::~myClass();
```

You should not call **exit** from **DllMain** with **DLL_PROCESS_ATTACH**. If you want to exit the **DLLMain** function, return **FALSE** from **DLL_PROCESS_ATTACH**.

Requirements

Function	Required header	Compatibility
exit	<process.h> or <stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_exit	<process.h> or <stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_exit.c
// This program returns an exit code of 1. The
// error code could be tested in a batch file.


#include <stdlib.h>

int main( void )
{
   exit( 1 );
}
```

.NET Framework Equivalent

System::Diagnostics::Process::Kill

See Also

Reference

Process and Environment Control
abort
atexit
_cexit, _c_exit
_exec, _wexec Functions
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

exp, expf

Calculates the exponential.

```
double exp(
   double x
);
float exp(
   float x
);  // C++ only
long double exp(
   long double x
);  // C++ only
float expf(
   float x
);
```

Parameters

x

Floating-point value.

Return Value

The **exp** function returns the exponential value of the floating-point parameter, *x*, if successful. That is, the result is e to the power *x*, where e is the base of the natural logarithm. On overflow, the function returns INF (infinite) and on underflow, **exp** returns 0.

Input	SEH Exception	Matherr Exception
± QNAN,IND	None	_DOMAIN
± ∞	INVALID	_DOMAIN
x ≥ 7.097827e+002	INEXACT+OVERFLOW	OVERFLOW
X ≤ -7.083964e+002	INEXACT+UNDERFLOW	UNDERFLOW

exp has an implementation that uses Streaming SIMD Extensions 2 (SSE2). See _set_SSE2_enable for information and restrictions on using the SSE2 implementation.

Remarks

C++ allows overloading, so you can call overloads of **exp**. In a C program, **exp** always takes and returns a double.

Requirements

Function	Required header	Compatibility
exp, **expf**	<math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_exp.c

#include <math.h>
#include <stdio.h>

int main( void )
{
```

```
    double x = 2.302585093, y;

    y = exp( x );
    printf( "exp( %f ) = %f\n", x, y );
}
```

Output

```
exp( 2.302585 ) = 10.000000
```

.NET Framework Equivalent

System::Math::Exp

See Also

Reference

Floating-Point Support

log, logf, log10, log10f

_expand

Changes the size of a memory block.

```
void *_expand(
   void *memblock,
   size_t size
);
```

Parameters

memblock

Pointer to previously allocated memory block.

size

New size in bytes.

Return Value

_expand returns a void pointer to the reallocated memory block. **_expand**, unlike **realloc**, cannot move a block to change its size. Thus, if there is sufficient memory available to expand the block without moving it, the *memblock* parameter to **_expand** is the same as the return value.

_expand returns **NULL** when an error is detected during its operation. For example, if **_expand** is used to shrink a memory block, it might detect corruption in the small block heap or an invalid block pointer and return **NULL**.

If there is insufficient memory available to expand the block to the given size without moving it, the function returns **NULL**. **_expand** never returns a block expanded to a size less than requested. If a failure occurs, **errno** indicates the nature of the failure. For more information about **errno**, see errno, _doserrno, _sys_errlist, and _sys_nerr.

The return value points to a storage space that is guaranteed to be suitably aligned for storage of any type of object. To check the new size of the item, use **_msize**. To get a pointer to a type other than **void**, use a type cast on the return value.

Remarks

The **_expand** function changes the size of a previously allocated memory block by trying to expand or contract the block without moving its location in the heap. The *memblock* parameter points to the beginning of the block. The *size* parameter gives the new size of the block, in bytes. The contents of the block are unchanged up to the shorter of the new and old sizes. *memblock* should not be a block that has been freed.

Note
On 64-bit platforms, **_expand** might not contract the block if the new size is less than the current size; in particular, if the block was less than 16K in size and therefore allocated in the Low Fragmentation Heap, **_expand** leaves the block unchanged and returns *memblock*.

When the application is linked with a debug version of the C run-time libraries, **_expand** resolves to _expand_dbg. For more information about how the heap is managed during the debugging process, see The CRT Debug Heap.

This function validates its parameters. If *memblock* is a null pointer, this function invokes an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **NULL**. If *size* is greater than **_HEAP_MAXREQ**, **errno** is set to **ENOMEM** and the function returns **NULL**.

Requirements

Function	Required header	Compatibility
_expand	<malloc.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_expand.c

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

int main( void )
{
   char *bufchar;
   printf( "Allocate a 512 element buffer\n" );
   if( (bufchar = (char *)calloc( 512, sizeof( char ) )) == NULL )
      exit( 1 );
   printf( "Allocated %d bytes at %Fp\n",
         _msize( bufchar ), (void *)bufchar );
   if( (bufchar = (char *)_expand( bufchar, 1024 )) == NULL )
      printf( "Can't expand" );
   else
      printf( "Expanded block to %d bytes at %Fp\n",
            _msize( bufchar ), (void *)bufchar );
   // Free memory
   free( bufchar );
   exit( 0 );
}
```

Sample Output

```
Allocate a 512 element buffer
Allocated 512 bytes at 002C12BC
Expanded block to 1024 bytes at 002C12BC
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation
calloc
free
malloc
_msize
realloc

_expand_dbg

Resizes a specified block of memory in the heap by expanding or contracting the block (debug version only).

```
void *_expand_dbg(
   void *userData,
   size_t newSize,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

userData

Pointer to the previously allocated memory block.

newSize

Requested new size for the block (in bytes).

blockType

Requested type for resized block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

filename

Pointer to the name of the source file that requested expand operation or **NULL**.

linenumber

Line number in the source file where the expand operation was requested or **NULL**.

The *filename* and *linenumber* parameters are only available when **_expand_dbg** has been called explicitly or the _CRTDBG_MAP_ALLOC preprocessor constant has been defined.

Return Value

On successful completion, **_expand_dbg** returns a pointer to the resized memory block. Because the memory is not moved, the address is the same as the userData. If an error occurred or the block could not be expanded to the requested size, it returns **NULL**. If a failure occurs, **errno** is with information from the operating system about the nature of the failure. For more information about **errno**, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **_expand_dbg** function is a debug version of the _expand function. When _DEBUG is not defined, each call to **_expand_dbg** is reduced to a call to **_expand**. Both **_expand** and **_expand_dbg** resize a memory block in the base heap, but **_expand_dbg** accommodates several debugging features: buffers on either side of the user portion of the block to test for leaks, a block type parameter to track specific allocation types, and *filename/linenumber* information to determine the origin of allocation requests.

_expand_dbg resizes the specified memory block with slightly more space than the requested *newSize*. *newSize* might be greater or less than the size of the originally allocated memory block. The additional space is used by the debug heap manager to link the debug memory blocks and to provide the application with debug header information and overwrite buffers. The resize is accomplished by either expanding or contracting the original memory block. **_expand_dbg** does not move the memory block, as does the _realloc_dbg function.

When *newSize* is greater than the original block size, the memory block is expanded. During an expansion, if the memory block cannot be expanded to accommodate the requested size, **NULL** is returned. When *newSize* is less than the original block size, the memory block is contracted until the new size is obtained.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see Memory Management and the Debug Heap. For information about the allocation block types and how they are used, see Types of Blocks on the Debug Heap. For information about the differences between calling a standard heap function and its debug version in a debug build of an application, see Using the Debug Version Versus the Base Version.

This function validates its parameters. If *memblock* is a null pointer, or if size is greater than **_HEAP_MAXREQ**, this function

invokes an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **NULL**.

Requirements

Routine	Required header	Compatibility
_expand_dbg	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

```
// crt_expand_dbg.c
//
// This program allocates a block of memory using _malloc_dbg
// and then calls _msize_dbg to display the size of that block.
// Next, it uses _expand_dbg to expand the amount of
// memory used by the buffer and then calls _msize_dbg again to
// display the new amount of memory allocated to the buffer.
//

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <crtdbg.h>

int main( void )
{
   long *buffer;
   size_t size;

   // Call _malloc_dbg to include the filename and line number
   // of our allocation request in the header
   buffer = (long *)_malloc_dbg( 40 * sizeof(long),
                                 _NORMAL_BLOCK, __FILE__, __LINE__ );
   if( buffer == NULL )
      exit( 1 );

   // Get the size of the buffer by calling _msize_dbg
   size = _msize_dbg( buffer, _NORMAL_BLOCK );
   printf( "Size of block after _malloc_dbg of 40 longs: %u\n", size );

   // Expand the buffer using _expand_dbg and show the new size
   buffer = (long *)_expand_dbg( buffer, size + sizeof(long),
                                 _NORMAL_BLOCK, __FILE__, __LINE__ );

   if( buffer == NULL )
      exit( 1 );
   size = _msize_dbg( buffer, _NORMAL_BLOCK );
   printf( "Size of block after _expand_dbg of 1 more long: %u\n",
           size );

   free( buffer );
   exit( 0 );
}
```

Sample Output

```
Size of block after _malloc_dbg of 40 longs: 160
Size of block after _expand_dbg of 1 more long: 164
```

Comment

The output of this program depends on your computer's ability to expand all the sections. If all sections are expanded, the output is reflected in the Output section.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_malloc_dbg

fabs, fabsf

Calculates the absolute value of the floating-point argument.

```
double fabs(
   double x
);
float fabs(
   float x
); // C++ only
long double fabs(
   long double x
); // C++ only
float fabsf(
   float x
);
```

Parameters

x

Floating-point value.

Return Value

fabs returns the absolute value of its argument. There is no error return.

Input	SEH Exception	Matherr Exception
± QNAN,IND	none	_DOMAIN

Remarks

C++ allows overloading, so you can call overloads of **fabs**. In a C program, **fabs** always takes and returns a double.

Requirements

Function	Required header	Compatibility
fabs, **fabsf**	<math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for abs.

.NET Framework Equivalent

System::Math::Abs

See Also

Reference

Floating-Point Support
abs, _abs64
_cabs
labs

fclose, _fcloseall

Closes a stream (**fclose**) or closes all open streams (**_fcloseall**).

```
int fclose(
   FILE *stream
);
int _fcloseall( void );
```

Parameters

stream

Pointer to **FILE** structure.

Return Value

fclose returns 0 if the stream is successfully closed. **_fcloseall** returns the total number of streams closed. Both functions return **EOF** to indicate an error.

Remarks

The **fclose** function closes *stream*. If *stream* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **fclose** sets **errno** to **EINVAL** and returns **EOF**. It is recommended that the *stream* pointer always be checked prior to calling this function.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

The **_fcloseall** function closes all open streams except **stdin**, **stdout**, **stderr** (and, in MS-DOS, **_stdaux** and **_stdprn**). It also closes and deletes any temporary files created by **tmpfile**. In both functions, all buffers associated with the stream are flushed prior to closing. System-allocated buffers are released when the stream is closed. Buffers assigned by the user with **setbuf** and **setvbuf** are not automatically released.

Note: When these functions are used to close a stream, the underlying file descriptor and OS file handle (or socket) are closed, as well as the stream. Thus, if the file was originally opened as a file handle or file descriptor and is closed with **fclose**, do not also call **_close** to close the file descriptor; do not call the Win32 function **CloseHandle** to close the file handle; and do not call **closesocket** to also close the socket.

fclose and **_fcloseall** include code to protect against interference from other threads. For non-locking version of a **fclose**, see **_fclose_nolock**.

Requirements

Function	Required header	Compatibility
fclose	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_fcloseall	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for fopen.

.NET Framework Equivalent

- System::IO::BinaryReader::Close
- System::IO::BinaryWriter::Close
- System::IO::StringReader::Close
- System::IO::StringWriter::Close

- System::IO::Stream::Close
- System::IO::StreamReader::Close
- System::IO::StreamWriter::Close
- System::IO::TextReader::Close
- System::IO::TextWriter::Close

See Also

Reference

Stream I/O
_close
_fdopen, _wfdopen
fflush
fopen, _wfopen
freopen, _wfreopen

_fclose_nolock

Closes a stream without thread-locking.

```
int _fclose_nolock(
   FILE *stream
);
```

Parameters

stream

Pointer to the **FILE** structure.

Return Value

fclose returns 0 if the stream is successfully closed. Returns **EOF** to indicate an error.

Remarks

This functions is a non-locking version of **fclose**. It is identical except that it is not protected from interference by other threads. It might be faster because it does not incur the overhead of locking out other threads. Use this function only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Requirements

Function	Required header	Compatibility
_fclose_nolock	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

- System::IO::BinaryReader::Close
- System::IO::BinaryWriter::Close
- System::IO::StringReader::Close
- System::IO::StringWriter::Close
- System::IO::Stream::Close
- System::IO::StreamReader::Close
- System::IO::StreamWriter::Close
- System::IO::TextReader::Close
- System::IO::TextWriter::Close

See Also

Reference

Stream I/O
_close
_fdopen, _wfdopen
fflush
fopen, _wfopen
freopen, _wfreopen

fcloseall

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _fcloseall instead.

fcvt

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _fcvt or security-enhanced _fcvt_s instead.

_fcvt

Converts a floating-point number to a string. This function is deprecated because a more secure version is available; see _fcvt_s.

```
char *_fcvt(
   double value,
   int count,
   int *dec,
   int *sign
);
```

Parameters

value

Number to be converted.

count

Number of digits after the decimal point.

dec

Pointer to the stored decimal-point position.

sign

Pointer to the stored sign indicator.

Return Value

_fcvt returns a pointer to the string of digits, NULL on error.

Remarks

The **_fcvt** function converts a floating-point number to a null-terminated character string. The *value* parameter is the floating-point number to be converted. **_fcvt** stores the digits of *value* as a string and appends a null character ('\0'). The *count* parameter specifies the number of digits to be stored after the decimal point. Excess digits are rounded off to *count* places. If there are fewer than *count* digits of precision, the string is padded with zeros.

The total number of digits returned by **_fcvt** will not exceed **_CVTBUFSIZE**.

Only digits are stored in the string. The position of the decimal point and the sign of *value* can be obtained from *dec* and sign after the call. The *dec* parameter points to an integer value; this integer value gives the position of the decimal point with respect to the beginning of the string. A zero or negative integer value indicates that the decimal point lies to the left of the first digit. The parameter *sign* points to an integer indicating the sign of *value*. The integer is set to 0 if *value* is positive and is set to a nonzero number if *value* is negative.

The difference between **_ecvt** and **_fcvt** is in the interpretation of the *count* parameter. **_ecvt** interprets *count* as the total number of digits in the output string, whereas **_fcvt** interprets *count* as the number of digits after the decimal point.

_ecvt and **_fcvt** use a single statically allocated buffer for the conversion. Each call to one of these routines destroys the results of the previous call.

This function validates its parameters. If *dec* or *sign* is NULL, or *count* is 0, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and NULL is returned.

Requirements

Function	Required header	Compatibility
_fcvt	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fcvt.c
// compile with: /W3
// This program converts the constant
// 3.1415926535 to a string and sets the pointer
// buffer to point to that string.

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   int  decimal, sign;
   char *buffer;
   double source = 3.1415926535;

   buffer = _fcvt( source, 7, &decimal, &sign ); // C4996
   // Note: _fcvt is deprecated; consider using _fcvt_s instead
   printf( "source: %2.10f   buffer: '%s'   decimal: %d   sign: %d\n",
            source, buffer, decimal, sign );
}
```

Output

```
source: 3.1415926535   buffer: '31415927'   decimal: 1   sign: 0
```

.NET Framework Equivalent

System::Convert::ToString

See Also

Reference

Data Conversion
Floating-Point Support
atof, _atof_l, _wtof, _wtof_l
_ecvt
_gcvt

_fcvt_s

Converts a floating-point number to a string. This is a version of _fcvt with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _fcvt_s(
   char* buffer,
   size_t sizeInBytes,
   double value,
   int count,
   int *dec,
   int *sign
);
template <size_t size>
errno_t _fcvt_s(
   char (&buffer)[size],
   double value,
   int count,
   int *dec,
   int *sign
); // C++ only
```

Parameters

[out] *buffer*

The supplied buffer that will hold the result of the conversion.

[in] *sizeInBytes*

The size of the buffer in bytes.

[in] *value*

Number to be converted.

[in] *count*

Number of digits after the decimal point.

[out] *dec*

Pointer to the stored decimal-point position.

[out] *sign*

Pointer to the stored sign indicator.

Return Value

Zero if successful. The return value is an error code if there is a failure. Error codes are defined in Errno.h. For a listing of these errors, see errno, _doserrno, _sys_errlist, and _sys_nerr.

In the case of an invalid parameter, as listed in the following table, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Error Conditions

buffer	***sizeInBytes***	**value**	**count**	**dec**	**sign**	**Return**	**Value in *buffer***
NULL	any	any	any	any	any	**EINVAL**	Not modified.
Not **NULL** (points to valid memory)	<=0	any	any	any	any	**EINVAL**	Not modified.
any	any	any	any	**NULL**	any	**EINVAL**	Not modified.
any	any	any	any	any	**NULL**	**EINVAL**	Not modified.

Security Issues

_fcvt_s might generate an access violation if *buffer* does not point to valid memory and is not **NULL**.

Remarks

The **_fcvt_s** function converts a floating-point number to a null-terminated character string. The *value* parameter is the floating-point number to be converted. **_fcvt_s** stores the digits of **value** as a string and appends a null character ('\0'). The *count* parameter specifies the number of digits to be stored after the decimal point. Excess digits are rounded off to *count* places. If there are fewer than *count* digits of precision, the string is padded with zeros.

Only digits are stored in the string. The position of the decimal point and the sign of *value* can be obtained from *dec* and *sign* after the call. The *dec* parameter points to an integer value; this integer value gives the position of the decimal point with respect to the beginning of the string. A zero or negative integer value indicates that the decimal point lies to the left of the first digit. The parameter *sign* points to an integer indicating the sign of *value*. The integer is set to 0 if *value* is positive and is set to a nonzero number if *value* is negative.

A buffer of length **_CVTBUFSIZE** is sufficient for any floating point value.

The difference between **_ecvt_s** and **_fcvt_s** is in the interpretation of the *count* parameter. **_ecvt_s** interprets *count* as the total number of digits in the output string, and **_fcvt_s** interprets *count* as the number of digits after the decimal point.

In C++, using this function is simplified by a template overload; the overload can infer buffer length automatically, eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

The debug version of this function first fills the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Requirements

Function	Required header	Optional header	Compatibility
_fcvt_s	<stdlib.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries: All versions of the C Run-Time Libraries.

Example

```
// fcvt_s.c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main()
{
  char * buf = 0;
  int decimal;
  int sign;
  int err;

  buf = (char*) malloc(_CVTBUFSIZE);
  err = _fcvt_s(buf, _CVTBUFSIZE, 1.2, 5, &decimal, &sign);

  if (err != 0)
  {
     printf("_fcvt_s failed with error code %d\n", err);
     exit(1);
  }

  printf("Converted value: %s\n", buf);

}
```

Output

```
Converted value: 120000
```

.NET Framework Equivalent

ToString

See Also

Reference

Data Conversion
Floating-Point Support
atof, _atof_l, _wtof, _wtof_l
_ecvt_s
_gcvt_s
_fcvt

fdopen

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _fdopen instead.

_fdopen, _wfdopen

Associates a stream with a file that was previously opened for low-level I/O.

```
FILE *_fdopen(
   int fd,
   const char *mode
);
FILE *_wfdopen(
   int fd,
   const wchar_t *mode
);
```

Parameters

fd

File descriptor of the open file.

mode

Type of file access.

Return Value

Each of these functions returns a pointer to the open stream. A null pointer value indicates an error. When an error occurs, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set either to **EBADF**, indicating a bad file descriptor, or **EINVAL**, indicating that *mode* was a null pointer.

For more information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_fdopen** function associates an I/O stream with the file identified by *fd*, thus allowing a file opened for low-level I/O to be buffered and formatted. **_wfdopen** is a wide-character version of **_fdopen**; the *mode* argument to **_wfdopen** is a wide-character string. **_wfdopen** and **_fdopen** behave identically otherwise.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tfdopen	_fdopen	_fdopen	_wfdopen

The *mode* character string specifies the type of file and file access.

The character string *mode* specifies the type of access requested for the file, as shown in the following table.

"r"

Opens for reading. If the file does not exist or cannot be found, the **fopen** call fails.

"w"

Opens an empty file for writing. If the given file exists, its contents are destroyed.

"a"

Opens for writing at the end of the file (appending); creates the file first if it does not exist.

"r+"

Opens for both reading and writing. (The file must exist.)

"w+"

Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.

"a+"

Opens for reading and appending; creates the file first if it does not exist.

When a file is opened with the **"a"** or **"a+"** access type, all write operations occur at the end of the file. The file pointer can be repositioned using **fseek** or **rewind**, but is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten. When the **"r+"**, **"w+"**, or **"a+"** access type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when you switch between reading and writing, there must be an intervening **fflush**, **fsetpos**, **fseek**, or **rewind** operation. The current position can be specified for the **fsetpos** or **fseek** operation, if desired.

In addition to the above values, the following characters can be included in *mode* to specify the translation mode for newline characters.

t

Open in text (translated) mode. In this mode, carriage return–line feed (CR-LF) combinations are translated into single line feeds (LF) on input, and LF characters are translated to CR-LF combinations on output. Also, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading/writing, **fopen** checks for a CTRL+Z at the end of the file and removes it, if possible. This is done because using the **fseek** and **ftell** functions to move within a file that ends with a CTRL+Z might cause **fseek** to behave improperly near the end of the file.

b

Open in binary (untranslated) mode; the above translations are suppressed.

c

Enable the commit flag for the associated *filename* so that the contents of the file buffer are written directly to disk if either **fflush** or **_flushall** is called.

n

Reset the commit flag for the associated *filename* to "no-commit." This is the default. It also overrides the global commit flag if you link your program with Commode.obj. The global commit flag default is "no-commit" unless you explicitly link your program with Commode.obj.

S

Specifies that caching is optimized for, but not restricted to, sequential access from disk.

R

Specifies that caching is optimized for, but not restricted to, random access from disk.

T

Specifies a file as temporary. If possible, it is not flushed to disk.

D

Specifies a file as temporary. It is deleted when the last file descriptor is closed.

The **t**, **c**, and **n** *mode* options are Microsoft extensions for **fopen** and **_fdopen** and should not be used where ANSI portability is desired.

If **t** or **b** is not given in *mode*, the default translation mode is defined by the global variable _fmode. If **t** or **b** is prefixed to the argument, the function fails and returns **NULL**. For a discussion of text and binary modes, see Text and Binary Mode File I/O.

Valid characters for the *mode* string used in **fopen** and **_fdopen** correspond to *oflag* arguments used in _open and _sopen, as follows.

Characters in *mode* string	**Equivalent *oflag* value for _open/_sopen**
a	**_O_WRONLY | _O_APPEND** (usually **_O_WRONLY | _O_CREAT | _O_APPEND**)
a+	**_O_RDWR | _O_APPEND** (usually **_O_RDWR | _O_APPEND | _O_CREAT**)
r	**_O_RDONLY**
r+	**_O_RDWR**
w	**_O_WRONLY** (usually **_O_WRONLY | _O_CREAT | _O_TRUNC**)

w+	**_O_RDWR** (usually **_O_RDWR | _O_CREAT | _O_TRUNC**)
b	**_O_BINARY**
t	**_O_TEXT**
c	None
n	None
S	**_O_SEQUENTIAL**
R	**_O_RANDOM**
T	**_O_SHORT_LIVED**
D	**_O_TEMPORARY**

Requirements

Function	Required header	Compatibility
_fdopen	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfdopen	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fdopen.c
// This program opens a file using low-level
// I/O, then uses _fdopen to switch to stream
// access. It counts the lines in the file.


#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
#include <share.h>

int main( void )
{
   FILE *stream;
   int  fd, count = 0;
   char inbuf[128];

   // Open a file.
   if( _sopen_s( &fd, "crt_fdopen.txt", _O_RDONLY, _SH_DENYNO, 0 ) )
      exit( 1 );

   // Get stream from file descriptor.
   if( (stream = _fdopen( fd, "r" )) == NULL )
      exit( 1 );

   while( fgets( inbuf, 128, stream ) != NULL )
      count++;

   // After _fdopen, close with fclose, not _close.
```

```
    fclose( stream );
    printf( "Lines in file: %d\n", count );
}
```

Input: crt_fdopen.txt

```
Line one
Line two
```

Output

```
Lines in file: 2
```

.NET Framework Equivalent

System::IO::FileStream::FileStream

See Also

Reference

Stream I/O
_dup, _dup2
fclose, _fcloseall
fopen, _wfopen
freopen, _wfreopen
_open, _wopen

feof

Tests for end-of-file on a stream.

```
int feof(
   FILE *stream
);
```

Parameters

stream

Pointer to **FILE** structure.

Return Value

The **feof** function returns a nonzero value if a read operation has attempted to read past the end of the file; it returns 0 otherwise. If the stream pointer is **NULL**, the function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the **feof** returns 0.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

The **feof** routine (implemented both as a function and as a macro) determines whether the end of *stream* has been passed. When the end of file is passed, read operations return an end-of-file indicator until the stream is closed or until **rewind**, **fsetpos**, **fseek**, or **clearerr** is called against it.

For example, if a file contains 10 bytes and you read 10 bytes from the file, **feof** will return 0 because, even though the file pointer is at the end of the file, you have not attempted to read beyond the end. Only after you try to read an 11th byte will **feof** return a nonzero value.

Requirements

Function	Required header	Compatibility
feof	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_feof.c
// This program uses feof to indicate when
// it reaches the end of the file CRT_FEOF.TXT. It also
// checks for errors with ferror.
//

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   int  count, total = 0;
   char buffer[100];
   FILE *stream;

   fopen_s( &stream, "crt_feof.txt", "r" );
   if( stream == NULL )
      exit( 1 );

   // Cycle until end of file reached:
   while( !feof( stream ) )
   {
```

```
        // Attempt to read in 100 bytes:
        count = fread( buffer, sizeof( char ), 100, stream );
        if( ferror( stream ) )      {
           perror( "Read error" );
           break;
        }

        // Total up actual bytes read
        total += count;
     }
     printf( "Number of bytes read = %d\n", total );
     fclose( stream );
  }
```

Input: crt_feof.txt

```
  Line one.
  Line two.
```

Output

```
Number of bytes read = 19
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Error Handling (CRT)
Stream I/O
clearerr
_eof
ferror
perror, _wperror

ferror

Tests for an error on a stream.

```
int ferror(
   FILE *stream
);
```

Parameters

stream

Pointer to **FILE** structure.

Return Value

If no error has occurred on *stream*, **ferror** returns 0. Otherwise, it returns a nonzero value. If stream is **NULL**, **ferror** invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns 0.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

The **ferror** routine (implemented both as a function and as a macro) tests for a reading or writing error on the file associated with *stream*. If an error has occurred, the error indicator for the stream remains set until the stream is closed or rewound, or until **clearerr** is called against it.

Requirements

Function	Required header	Compatibility
ferror	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for feof.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Error Handling (CRT)
Stream I/O
clearerr
_eof
feof
fopen, _wfopen
perror, _wperror

fflush

Flushes a stream.

```
int fflush(
   FILE *stream
);
```

Parameters

stream

Pointer to **FILE** structure.

Return Value

fflush returns 0 if the buffer was successfully flushed. The value 0 is also returned in cases in which the specified stream has no buffer or is open for reading only. A return value of **EOF** indicates an error.

Note

If **fflush** returns **EOF**, data may have been lost due to a write failure. When setting up a critical error handler, it is safest to turn buffering off with the **setvbuf** function or to use low-level I/O routines such as **_open**, **_close**, and **_write** instead of the stream I/O functions.

Remarks

The **fflush** function flushes a stream. If the file associated with *stream* is open for output, **fflush** writes to that file the contents of the buffer associated with the stream. If the stream is open for input, **fflush** clears the contents of the buffer. **fflush** negates the effect of any prior call to **ungetc** against *stream*. Also, **fflush(NULL)** flushes all streams opened for output. The stream remains open after the call. **fflush** has no effect on an unbuffered stream.

Buffers are normally maintained by the operating system, which determines the optimal time to write the data automatically to disk: when a buffer is full, when a stream is closed, or when a program terminates normally without closing the stream. The commit-to-disk feature of the run-time library lets you ensure that critical data is written directly to disk rather than to the operating-system buffers. Without rewriting an existing program, you can enable this feature by linking the program's object files with COMMODE.OBJ. In the resulting executable file, calls to **_flushall** write the contents of all buffers to disk. Only **_flushall** and **fflush** are affected by COMMODE.OBJ.

For information about controlling the commit-to-disk feature, see Stream I/O, fopen, and _fdopen.

This function locks the calling thread and is therefore thread-safe. For a non-locking version, see **_fflush_nolock**.

Requirements

Function	Required header	Compatibility
fflush	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fflush.c
#include <stdio.h>
#include <conio.h>

int main( void )
{
   int integer;
   char string[81];

   // Read each word as a string.
```

```
    printf( "Enter a sentence of four words with scanf: " );
    for( integer = 0; integer < 4; integer++ )
    {
       scanf_s( "%s", string, sizeof(string) );
       printf( "%s\n", string );
    }

    // You must flush the input buffer before using gets.
    // fflush on input stream is an extension to the C standard
    fflush( stdin );
    printf( "Enter the same sentence with gets: " );
    gets_s( string, sizeof(string) );
    printf( "%s\n", string );
 }
```

Input

```
This is a test
This is a test
```

Sample Output

```
Enter a sentence of four words with scanf: This is a test
This
is
a
test
Enter the same sentence with gets: This is a test
This is a test
```

.NET Framework Equivalent

System::IO::FileStream::Flush

See Also

Reference

Stream I/O
fclose, _fcloseall
_flushall
setvbuf

_fflush_nolock

Flushes a stream without locking the thread.

```
int _fflush_nolock(
   FILE *stream
);
```

Parameters

stream

Pointer to the **FILE** structure.

Return Value

See fflush.

Remarks

This function is a non-locking version of **fflush**. It is identical to **fflush** except that it is not protected from interference by other threads. It might be faster because it does not incur the overhead of locking out other threads. Use this function only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Requirements

Function	**Required header**	**Compatibility**
_fflush_nolock	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::IO::FileStream::Flush

See Also

Reference

Stream I/O
fclose, _fcloseall
_flushall
setvbuf

fgetc, fgetwc

Read a character from a stream.

```
int fgetc(
   FILE *stream
);
wint_t fgetwc(
   FILE *stream
);
```

Parameters

stream

Pointer to **FILE** structure.

Return Value

fgetc returns the character read as an **int** or returns **EOF** to indicate an error or end of file. **fgetwc** returns, as a wint_t, the wide character that corresponds to the character read or returns **WEOF** to indicate an error or end of file. For both functions, use **feof** or **ferror** to distinguish between an error and an end-of-file condition. If a read error occurs, the error indicator for the stream is set. If *stream* is **NULL**, **fgetc** and **fgetwc** invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **EOF**.

Remarks

Each of these functions reads a single character from the current position of the file associated with *stream*. The function then increments the associated file pointer (if defined) to point to the next character. If the stream is at end of file, the end-of-file indicator for the stream is set.

fgetc is equivalent to **getc**, but is implemented only as a function, rather than as a function and a macro.

fgetwc is the wide-character version of **fgetc**; it reads *c* as a multibyte character or a wide character according to whether *stream* is opened in text mode or binary mode.

The versions with the **_nolock** suffix are identical except that they are not protected from interference by other threads.

For more information about processing wide characters and multibyte characters in text and binary modes, see Unicode Stream I/O in Text and Binary Modes.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_fgettc	**fgetc**	**fgetc**	**fgetwc**

Requirements

Function	Required header	Compatibility
fgetc	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
fgetwc	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fgetc.c
// This program uses getc to read the first
// 80 input characters (or until the end of input)
// and place them into a string named buffer.
```

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   FILE *stream;
   char buffer[81];
   int  i, ch;

   // Open file to read line from:
   fopen_s( &stream, "crt_fgetc.txt", "r" );
   if( stream == NULL )
      exit( 0 );

   // Read in first 80 characters and place them in "buffer":
   ch = fgetc( stream );
   for( i=0; (i < 80 ) && ( feof( stream ) == 0 ); i++ )
   {
      buffer[i] = (char)ch;
      ch = fgetc( stream );
   }

   // Add null to end string
   buffer[i] = '\0';
   printf( "%s\n", buffer );
   fclose( stream );
}
```

Input: crt_fgetc.txt

```
Line one.
Line two.
```

Output

```
Line one.
Line two.
```

.NET Framework Equivalent

- System::IO::StreamReader::Read
- System::Console::Read

See Also

Reference

Stream I/O
fputc, fputwc
getc, getwc

_fgetc_nolock, _fgetwc_nolock

Reads a character from a stream without locking the thread.

```
int _fgetc_nolock(
   FILE *stream
);
wint_t _fgetwc_nolock(
   FILE *stream
);
```

Parameters

stream

Pointer to the **FILE** structure.

Return Value

See fgetc, fgetwc.

Remarks

_fgetc_nolock and **_fgetwc_nolock** are identical to **fgetc** and **fgetwc**, respectively, except that they are not protected from interference by other threads. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_fgettc_nolock	**_fgetc_nolock**	**_fgetc_nolock**	**_fgetwc_nolock**

Requirements

Function	Required header	Compatibility
_fgetc_nolock	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_fgetwc_nolock	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fgetc_nolock.c
// This program uses getc to read the first
// 80 input characters (or until the end of input)
// and place them into a string named buffer.


#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   FILE *stream;
   char buffer[81];
   int  i, ch;

   // Open file to read line from:
   if( fopen_s( &stream, "crt_fgetc_nolock.txt", "r" ) != 0 )
      exit( 0 );
```

```
    // Read in first 80 characters and place them in "buffer":
    ch = fgetc( stream );
    for( i=0; (i < 80 ) && ( feof( stream ) == 0 ); i++ )
    {
       buffer[i] = (char)ch;
       ch = _fgetc_nolock( stream );
    }

    // Add null to end string
    buffer[i] = '\0';
    printf( "%s\n", buffer );
    fclose( stream );
 }
```

Input: crt_fgetc_nolock.txt

```
 Line one.
 Line two.
```

Output

```
Line one.
Line two.
```

.NET Framework Equivalent

- System::IO::StreamReader::Read
- System::Console::Read

See Also

Reference

Stream I/O
fputc, fputwc
getc, getwc

fgetchar

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _fgetchar instead.

_fgetchar, _fgetwchar

Reads a character from **stdin**.

```
int _fgetchar( void );
wint_t _fgetwchar( void );
```

Return Value

_fgetchar returns the character read as an **int** or return **EOF** to indicate an error or end of file. **_fgetwchar** returns, as a wint_t, the wide character that corresponds to the character read or returns **WEOF** to indicate an error or end of file. For both functions, use **feof** or **ferror** to distinguish between an error and an end-of-file condition.

Remarks

These functions read a single character from **stdin**. The function then increments the associated file pointer (if defined) to point to the next character. If the stream is at end of file, the end-of-file indicator for the stream is set.

_fgetchar is equivalent to **fgetc(stdin)**. It is also equivalent to **getchar**, but implemented only as a function, rather than as a function and a macro. **_fgetwchar** is the wide-character version of **_fgetchar**.

These functions are not compatible with the ANSI standard.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_fgettchar	**_fgetchar**	**_fgetchar**	**_fgetwchar**

Requirements

Function	Required header	Compatibility
_fgetchar	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_fgetwchar	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fgetchar.c
// This program uses _fgetchar to read the first
// 80 input characters (or until the end of input)
// and place them into a string named buffer.
//

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   char buffer[81];
   int  i, ch;

   // Read in first 80 characters and place them in "buffer":
   ch = _fgetchar();
   for( i=0; (i < 80 ) && ( feof( stdin ) == 0 ); i++ )
   {
      buffer[i] = (char)ch;
      ch = _fgetchar();
   }
```

```
    // Add null to end string
    buffer[i] = '\0';
    printf( "%s\n", buffer );
}
```

Input

```
Line one.
Line two.
```

Output

```
Line one.
Line two.
```

.NET Framework Equivalent

- System::IO::StreamReader::Read
- System::Console::Read

See Also

Reference

Stream I/O
fputc, fputwc
getc, getwc

fgetpos

Gets a stream's file-position indicator.

```
int fgetpos(
   FILE *stream,
   fpos_t *pos
);
```

Parameters

stream

Target stream.

pos

Position-indicator storage.

Return Value

If successful, **fgetpos** returns 0. On failure, it returns a nonzero value and sets **errno** to one of the following manifest constants (defined in STDIO.H): **EBADF**, which means the specified stream is not a valid file pointer or is not accessible, or **EINVAL**, which means the *stream* value or the value of *pos* is invalid, such as if either is a null pointer. If *stream* or *pos* is a **NULL** pointer, the function invokes the invalid parameter handler, as described in Parameter Validation.

Remarks

The **fgetpos** function gets the current value of the *stream* argument's file-position indicator and stores it in the object pointed to by *pos*. The **fsetpos** function can later use information stored in *pos* to reset the *stream* argument's pointer to its position at the time **fgetpos** was called. The *pos* value is stored in an internal format and is intended for use only by **fgetpos** and **fsetpos**.

Requirements

Function	Required header	Compatibility
fgetpos	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fgetpos.c
// This program uses fgetpos and fsetpos to
// return to a location in a file.

#include <stdio.h>

int main( void )
{
   FILE   *stream;
   fpos_t pos;
   char   buffer[20];

   if( fopen_s( &stream, "crt_fgetpos.txt", "rb" ) ) {
      perror( "Trouble opening file" );
      return -1;
   }

   // Read some data and then save the position.
   fread( buffer, sizeof( char ), 8, stream );
   if( fgetpos( stream, &pos ) != 0 ) {
      perror( "fgetpos error" );
      return -1;
```

```
    }

    fread( buffer, sizeof( char ), 13, stream );
    printf( "after fgetpos: %.13s\n", buffer );

    // Restore to old position and read data
    if( fsetpos( stream, &pos ) != 0 ) {
       perror( "fsetpos error" );
       return -1;
    }

    fread( buffer, sizeof( char ), 13, stream );
    printf( "after fsetpos: %.13s\n", buffer );
    fclose( stream );
 }
```

Input: crt_fgetpos.txt

```
 fgetpos gets a stream's file-position indicator.
```

Output crt_fgetpos.txt

```
 after fgetpos: gets a stream
 after fsetpos: gets a stream
```

.NET Framework Equivalent

System::IO::FileStream::Position

See Also

Reference

Stream I/O

fsetpos

fgets, fgetws

Get a string from a stream.

```
char *fgets(
   char *str,
   int n,
   FILE *stream
);
wchar_t *fgetws(
   wchar_t *str,
   int n,
   FILE *stream
);
```

Parameters

str

Storage location for data.

n

Maximum number of characters to read.

stream

Pointer to **FILE** structure.

Return Value

Each of these functions returns *str*. **NULL** is returned to indicate an error or an end-of-file condition. Use **feof** or **ferror** to determine whether an error occurred. If *str* or *stream* is a null pointer, or *n* is less than or equal to zero, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **NULL**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

The **fgets** function reads a string from the input *stream* argument and stores it in *str*. **fgets** reads characters from the current stream position to and including the first newline character, to the end of the stream, or until the number of characters read is equal to *n* – 1, whichever comes first. The result stored in *str* is appended with a null character. The newline character, if read, is included in the string.

fgetws is a wide-character version of **fgets**.

fgetws reads the wide-character argument *str* as a multibyte-character string or a wide-character string according to whether *stream* is opened in text mode or binary mode, respectively. For more information about using text and binary modes in Unicode and multibyte stream-I/O, see Text and Binary Mode File I/O and Unicode Stream I/O in Text and Binary Modes.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_fgetts	**fgets**	**fgets**	**fgetws**

Requirements

Function	Required header	Compatibility
fgets	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
fgetws	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fgets.c
// This program uses fgets to display
// a line from a file on the screen.
//

#include <stdio.h>

int main( void )
{
   FILE *stream;
   char line[100];

   if( fopen_s( &stream, "crt_fgets.txt", "r" ) == 0 )
   {
      if( fgets( line, 100, stream ) == NULL)
         printf( "fgets error\n" );
      else
         printf( "%s", line);
      fclose( stream );
   }
}
```

Input: crt_fgets.txt

```
Line one.
Line two.
```

Output

```
Line one.
```

.NET Framework Equivalent

- System::IO::StreamReader::ReadLine
- System::IO::TextReader::ReadBlock

See Also

Reference

Stream I/O
fputs, fputws
gets, _getws
puts, _putws

filelength

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _filelength instead.

_filelength, _filelengthi64

Gets the length of a file.

```
long _filelength(
   int fd
);
__int64 _filelengthi64(
   int fd
);
```

Parameters

fd

Target the file descriptor.

Return Value

Both **_filelength** and **_filelengthi64** return the file length, in bytes, of the target file associated with *fd*. If *fd* is an invalid file descriptor, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, both functions return –1L to indicate an error and set **errno** to **EBADF**.

Requirements

Function	Required header	Compatibility
_filelength	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_filelengthi64	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example for _chsize.

.NET Framework Equivalent

- System::IO::Stream::SetLength
- System::IO::FileStream::SetLength

See Also

Reference

File Handling
_chsize
_fileno
_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32
_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32
_stat, _wstat Functions
_stat, _wstat Functions

fileno

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _fileno instead.

_fileno

Gets the file descriptor associated with a stream.

```
int _fileno(
   FILE *stream
);
```

Parameters

stream

Pointer to the **FILE** structure.

Return Value

_fileno returns the file descriptor. There is no error return. The result is undefined if *stream* does not specify an open file. If stream is **NULL**, **_fileno** invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns -1 and sets **errno** to **EINVAL**.

For more information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Note

In Visual C++ 2005, there is a behavior change. If **stdout** or **stderr** is not associated with an output stream (for example, in a Windows application without a console window), the file descriptor returned is -2. In previous versions, the file descriptor ret urned was -1. This change allows applications to distinguish this condition from an error.

Remarks

The **_fileno** routine returns the file descriptor currently associated with *stream*. This routine is implemented both as a function and as a macro. For information about choosing either implementation, see Choosing Between Functions and Macros.

Requirements

Function	Required header	Compatibility
_fileno	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fileno.c
// This program uses _fileno to obtain
// the file descriptor for some standard C streams.
//

#include <stdio.h>

int main( void )
{
   printf( "The file descriptor for stdin is %d\n", _fileno( stdin ) );
   printf( "The file descriptor for stdout is %d\n", _fileno( stdout ) );
   printf( "The file descriptor for stderr is %d\n", _fileno( stderr ) );
}
```

Output

```
The file descriptor for stdin is 0
The file descriptor for stdout is 1
The file descriptor for stderr is 2
```

.NET Framework Equivalent

System::IO::FileStream::Handle

See Also

Reference

Stream I/O
_fdopen, _wfdopen
_filelength, _filelengthi64
fopen, _wfopen
freopen, _wfreopen

_find, _wfind Functions

These functions search for and close searches for specified file names:

- _findclose
- _findnext, _wfindnext
- _findfirst, _wfindfirst

Remarks

The **_findfirst** function provides information about the first instance of a file name that matches the file specified in the *filespec* argument. Any wildcard combination supported by the host operating system can be used in *filespec*. File information is returned in a **_finddata_t** structure, defined in IO.h. There are many variations on the **_finddata_t** structure that are used by various functions in the family. The basic **_finddata_t** structure includes the following elements.

unsigned attrib

File attribute.

time_t time_create

Time of file creation (–1L for FAT file systems).

time_t time_access

Time of the last file access (–1L for FAT file systems).

time_t time_write

Time of the last write to file.

_fsize_t size

Length of the file in bytes.

char name[**_MAX_PATH**]

Null-terminated name of matched file/directory, without the path.

In file systems that do not support the creation and last access times of a file, such as the FAT system, the **time_create** and **time_access** fields are always –1L.

_MAX_PATH is defined in Stdlib.h as 260 bytes.

You cannot specify target attributes (such as **_A_RDONLY**) by which to limit the find operation. This attribute is returned in the **attrib** field of the **_finddata_t** structure and can have the following values (defined in IO.h). Users should not rely on these being the only values possible for the **attrib** field.

_A_ARCH

Archive. Set whenever the file is changed and cleared by the **BACKUP** command. Value: 0x20.

_A_HIDDEN

Hidden file. Not normally seen with the DIR command, unless the **/AH** option is used. Returns information about normal files and files with this attribute. Value: 0x02.

_A_NORMAL

Normal. File has no other attributes set and can be read or written to without restriction. Value: 0x00.

_A_RDONLY

Read-only. File cannot be opened for writing and a file with the same name cannot be created. Value: 0x01.

_A_SUBDIR

Subdirectory. Value: 0x10.

_A_SYSTEM

System file. Not normally seen with the **DIR** command, unless the **/A** or **/A:S** option is used. Value: 0x04.

_findnext finds the next name, if any, that matches the *filespec* argument specified in a prior call to **_findfirst**. The *fileinfo* argument should point to a structure initialized by a previous call to **_findfirst**. If a match is found, the *fileinfo* structure contents are altered as described above. **_findclose** closes the specified search handle and releases all associated resources for both **_findfirst** and **_findnext**. The handle returned by either **_findfirst** or **_findnext** must first be passed to **_findclose**, before modification operations, such as deleting, can be performed on the directories that form the paths passed to them.

The **_find** functions allow nested calls. For example, if the file found by a call to **_findfirst** or **_findnext** is a subdirectory, a new search can be initiated with another call to **_findfirst** or **_findnext**.

_wfindfirst and **_wfindnext** are wide-character versions of **_findfirst** and **_findnext**. The structure argument of the wide-character versions has the **_wfinddata_t** data type, which is defined in IO.h and in Wchar.h. The fields of this data type are the same as those of the **_finddata_t** data type, except that in **_wfinddata_t** the name field is of type **wchar_t** rather than type **char**. Otherwise **_wfindfirst** and **_wfindnext** behave identically to **_findfirst** and **_findnext**.

In Visual C++ 2005, **_findfirst** and **_findnext** use the 64-bit time type. If you need to use the old 32-bit time type, you can define **_USE_32BIT_TIME_T**. The versions of these functions with the **32** suffix in their names use the 32-bit time type, and those with the **64** suffix use the 64-bit time type.

Functions **_findfirst32i64**, **_findnext32i64**, **_wfindfirst32i64**, and **_wfindnext32i64** also behave identically to the 32-bit time type versions of these functions except they use and return 64-bit file lengths. Functions **_findfirst64i32**, **_findnext64i32**, **_wfindfirst64i32**, and **_wfindnext64i32** use the 64-bit time type but use 32-bit file lengths. These functions use appropriate variations of the **_finddata_t** type in which the fields have different types for the time and the file size.

_finddata_t is actually a macro that evaluates to **_finddata64i32_t** (or **_finddata32_t** if **_USE_32BIT_TIME_T** is defined). The following table summarizes the variations on **_finddata_t**.

Structure	Time type	File size type
_finddata_t, **_wfinddata_t**	**__time64_t**	**_fsize_t**
_finddata32_t, **_wfinddata32_t**	**__time32_t**	**_fsize_t**
__finddata64_t, **__wfinddata64_t**	**__time64_t**	**__int64**
_finddata32i64_t, **_wfinddata32i64_t**	**__time32_t**	**__int64**
_finddata64i32_t, **_wfinddata64i32_t**	**__time64_t**	**_fsize_t**

In Visual C++ 2005, **_fsize_t** is a **typedef** for **unsigned long** (32 bits).

Example

```
// crt_find.c
// This program uses the 32-bit _find functions to print
// a list of all files (and their attributes) with a .C extension
// in the current directory.

#include <stdio.h>
#include <stdlib.h>
#include <io.h>
#include <time.h>

int main( void )
{
   struct _finddata_t c_file;
   intptr_t hFile;

   // Find first .c file in current directory
   if( (hFile = _findfirst( "*.c", &c_file )) == -1L )
      printf( "No *.c files in current directory!\n" );
   else
   {
      printf( "Listing of .c files\n\n" );
      printf( "RDO HID SYS ARC  FILE         DATE %25c SIZE\n", ' ' );
      printf( "--- --- --- ---  ----         ---- %25c ----\n", ' ' );
```

```
        do {
           char buffer[30];
           printf( ( c_file.attrib & _A_RDONLY ) ? " Y  " : " N  " );
           printf( ( c_file.attrib & _A_SYSTEM ) ? " Y  " : " N  " );
           printf( ( c_file.attrib & _A_HIDDEN ) ? " Y  " : " N  " );
           printf( ( c_file.attrib & _A_ARCH )   ? " Y  " : " N  " );
           ctime_s( buffer, _countof(buffer), &c_file.time_write );
           printf( " %-12s %.24s  %9ld\n",
              c_file.name, buffer, c_file.size );
        } while( _findnext( hFile, &c_file ) == 0 );
        _findclose( hFile );
     }
  }
```

Sample Output

```
Listing of .c files

RDO HID SYS ARC  FILE         DATE                           SIZE
--- --- --- ---  ----         ----                           ----
 N   N   N   Y   blah.c       Wed Feb 13 09:21:42 2002       1715
 N   N   N   Y   test.c       Wed Feb 06 14:30:44 2002        312
```

See Also

Reference

System Calls

_findclose

Closes the specified search handle and releases associated resources.

```
int _findclose(
   intptr_t handle
);
```

Parameters

handle

Search handle returned by a previous call to **_findfirst**.

Return Value

If successful, **_findclose** returns 0. Otherwise, it returns –1 and sets **errno** to **ENOENT**, indicating that no more matching files could be found.

Requirements

Function	Required header	Compatibility
_findclose	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

System Calls

_find, _wfind Functions

_findfirst, _wfindfirst

Provide information about the first instance of a file name that matches the file specified in the *filespec* argument.

```
intptr_t _findfirst(
   const char *filespec,
   struct _finddata_t *fileinfo
);
intptr_t _findfirst32(
   const char *filespec,
   struct _finddata32_t *fileinfo
);
intptr_t _findfirst64(
   const char *filespec,
   struct __finddata64_t *fileinfo
);
intptr_t _findfirsti64(
   const char *filespec,
   struct _finddatai64_t *fileinfo
);
intptr_t _findfirst32i64(
   const char *filespec,
   struct _finddata32i64_t *fileinfo
);
intptr_t _findfirst64i32(
   const char *filespec,
   struct _finddata64i32_t *fileinfo
);
intptr_t _wfindfirst(
   const wchar_t *filespec,
   struct _wfinddata_t *fileinfo
);
intptr_t _wfindfirst32(
   const wchar_t *filespec,
   struct __wfinddata32_t *fileinfo
);
intptr_t _wfindfirst64(
   const wchar_t *filespec,
   struct __wfinddata64_t *fileinfo
);
intptr_t _wfindfirsti64(
   const wchar_t *filespec,
   struct _wfinddatai64_t *fileinfo
);
intptr_t _wfindfirst32i64(
   const wchar_t *filespec,
   struct _wfinddata32i64_t *fileinfo
);
intptr_t _wfindfirst64i32(
   const wchar_t *filespec,
   struct _wfinddata64i32_t *fileinfo
);
```

Parameters

filespec

Target file specification (can include wildcards).

fileinfo

File information buffer.

Return Value

If successful, **_findfirst** returns a unique search handle identifying the file or group of files matching the *filespec* specification,

which can be used in a subsequent call to _findnext or to **_findclose**. Otherwise, **_findfirst** returns –1 and sets **errno** to one of the following values.

EINVAL

Invalid parameter: *filespec* or *fileinfo* was **NULL**. Or, the operating system returned an unexpected error.

ENOENT

File specification that could not be matched.

ENOMEM

Not enough memory or the file name given was greater than **MAX_PATH**.

EINVAL

Invalid file name specification.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

If an invalid parameter is passed in, these functions invoke the invalid parameter handler, as described in Parameter Validation.

Remarks

You must call _findclose after you are finished using either the **_findfirst** or **_findnext** function (or any variants). This frees up resources used by these functions in your application.

The variations of these functions with the **w** prefix are wide-character versions; otherwise, they are identical to the corresponding single-byte functions.

Variations of these functions support 32-bit or 64-bit time types and 32-bit or 64-bit file sizes. The first numerical suffix (**32** or **64**) indicates the size of the time type used; the second suffix is either **i32** or **i64**, indicating whether the file size is represented as a 32-bit or 64-bit integer. For information about which versions support 32-bit and 64-bit time types and file sizes, see the following table. The **i32** or **i64** suffix is omitted if it is the same as the size of the time type, so **_findfirst64** also supports 64-bit file lengths and **_findfirst32** supports only 32-bit file lengths.

These functions use various forms of the **_finddata_t** structure for the *fileinfo* parameter. For more information about the structure, see _find, _wfind Functions.

The variations that use a 64-bit time type allow file-creation dates to be expressed up through 23:59:59, December 31, 3000, UTC; whereas those using 32-bit time types only represent dates through 19:14:07 January 18, 2038, UTC. Midnight, January 1, 1970, is the lower bound of the date range for all these functions.

Unless you have a specific reason to use the versions that specify the time size explicitly, use **_findfirst** or **_wfindfirst** or, if you need to support file sizes greater than 3 GB, use **_findfirsti64** or **_wfindfirsti64**. All these functions use the 64-bit time type in Visual C++ 2005. In previous versions, these functions used a 32-bit time type. If this is a breaking change for an application, you might define **_USE_32BIT_TIME_T** to get the old behavior. If **_USE_32BIT_TIME_T** is defined, **_findfirst**, **_finfirsti64**, and their corresponding Unicode versions use a 32-bit time.

Time Type and File Length Type Variations of _findfirst

Functions	_USE_32BIT_TIME_T defined?	Time type	File length type
_findfirst, **_wfindfirst**	Not defined	64-bit	32-bit
_findfirst, **_wfindfirst**	Defined	32-bit	32-bit
_findfirst32, **_wfindfirst32**	Not affected by the macro definition	32-bit	32-bit
_findfirst64, **_wfindfirst64**	Not affected by the macro definition	64-bit	64-bit
_findfirsti64, **_wfindfirsti64**	Not defined	64-bit	64-bit
_findfirsti64, **_wfindfirsti64**	Defined	32-bit	64-bit
_findfirst32i64, **_wfindfirst32i64**	Not affected by the macro definition	32-bit	64-bit

_findfirst64i32, **_wfindfirst64i32**	Not affected by the macro definition	64-bit	32-bit

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tfindfirst	**_findfirst**	**_findfirst**	**_wfindfirst**
_tfindfirst32	**_findfirst32**	**_findfirst32**	**_wfindfirst32**
_tfindfirst64	**_findfirst64**	**_findfirst64**	**_wfindfirst64**
_tfindfirsti64	**_findfirsti64**	**_findfirsti64**	**_wfindfirsti64**
_tfindfirst32i64	**_findfirst32i64**	**_findfirst32i64**	**_wfindfirst32i64**
_tfindfirst64i32	**_findfirst64i32**	**_findfirst64i32**	**_wfindfirst64i32**

Requirements

Function	Required header	Compatibility
_findfirst	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_findfirst32	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_findfirst64	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_findfirsti64	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_findfirst32i64	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_findfirst64i32	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfindfirst	<io.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfindfirst32	<io.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfindfirst64	<io.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfindfirsti64	<io.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfindfirst32i64	<io.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfindfirst64i32	<io.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::IO::DirectoryInfo::GetFiles

See Also

Reference

System Calls

_find, _wfind Functions

_findnext, _wfindnext

Find the next name, if any, that matches the *filespec* argument in a previous call to _findfirst, and then alter the *fileinfo* structure contents accordingly.

```
int _findnext(
   intptr_t handle,
   struct _finddata_t *fileinfo
);
int _findnext32(
   intptr_t handle,
   struct _finddata32_t *fileinfo
);
int _findnext64(
   intptr_t handle,
   struct __finddata64_t *fileinfo
);
int _findnexti64(
   intptr_t handle,
   struct __finddatai64_t *fileinfo
);
int _findnext32i64(
   intptr_t handle,
   struct _finddata32i64_t *fileinfo
);
int _findnext64i32(
   intptr_t handle,
   struct _finddata64i32_t *fileinfo
);
int _wfindnext(
   intptr_t handle,
   struct _wfinddata_t *fileinfo
);
int _wfindnext32(
   intptr_t handle,
   struct _wfinddata32_t *fileinfo
);
int _wfindnext64(
   intptr_t handle,
   struct _wfinddata64_t *fileinfo
);
int _wfindnexti64(
   intptr_t handle,
   struct _wfinddatai64_t *fileinfo
);
int _wfindnext32i64(
   intptr_t handle,
   struct _wfinddatai64_t *fileinfo
);
int _wfindnext64i32(
   intptr_t handle,
   struct _wfinddata64i32_t *fileinfo
);
```

Parameters

handle

Search handle returned by a previous call to **_findfirst**.

fileinfo

File information buffer.

Return Value

If successful, returns 0. Otherwise, returns –1 and sets **errno** to a value indicating the nature of the failure. Possible error codes are shown in the following table.

EINVAL

Invalid parameter: *fileinfo* was **NULL**. Or, the operating system returned an unexpected error.

ENOENT

No more matching files could be found.

ENOMEM

Not enough memory or the file name's length exceeded **MAX_PATH**.

If an invalid parameter is passed in, these functions invoke the invalid parameter handler, as described in Parameter Validation.

Remarks

You must call _findclose after you are finished using either the **_findfirst** or **_findnext** function (or any variants). This frees up resources used by these functions in your application.

The variations of these functions with the **w** prefix are wide-character versions; otherwise, they are identical to the corresponding single-byte functions.

Variations of these functions support 32-bit or 64-bit time types and 32-bit or 64-bit file sizes. The first numerical suffix (**32** or **64**) indicates the size of the time type used; the second suffix is either **i32** or **i64**, indicating whether the file size is represented as a 32-bit or 64-bit integer. For information about which versions support 32-bit and 64-bit time types and file sizes, see the following table. The variations that use a 64-bit time type allow file-creation dates to be expressed up through 23:59:59, December 31, 3000, UTC; whereas those using 32-bit time types only represent dates through 19:14:07 January 18, 2038, UTC. Midnight, January 1, 1970, is the lower bound of the date range for all these functions.

Unless you have a specific reason to use the versions that specify the time size explicitly, use **_findnext** or **_wfindnext** or, if you need to support file sizes greater than 3 GB, use **_findnexti64** or **_wfindnexti64**. All these functions use the 64-bit time type in Visual C++ 2005. In previous versions, these functions used a 32-bit time type. If this is a breaking change for an application, you might define **_USE_32BIT_TIME_T** to get the old behavior. If **_USE_32BIT_TIME_T** is defined, **_findnext**, **_finnexti64** and their corresponding Unicode versions use a 32-bit time.

Time Type and File Length Type Variations of _findnext

Functions	_USE_32BIT_TIME_T defined?	Time type	File length type
_findnext, **_wfindnext**	Not defined	64-bit	32-bit
_findnext, **_wfindnext**	Defined	32-bit	32-bit
_findnext32, **_wfindnext32**	Not affected by the macro definition	32-bit	32-bit
_findnext64, **_wfindnext64**	Not affected by the macro definition	64-bit	64-bit
_findnexti64, **_wfindnexti64**	Not defined	64-bit	64-bit
_findnexti64, **_wfindnexti64**	Defined	32-bit	64-bit
_findnext32i64, **_wfindnext32i64**	Not affected by the macro definition	32-bit	64-bit
_findnext64i32, **_wfindnext64i32**	Not affected by the macro definition	64-bit	32-bit

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tfindnext	**_findnext**	**_findnext**	**_wfindnext**
_tfindnext32	**_findnext32**	**_findnext32**	**_wfindnext32**
_tfindnext64	**_findnext64**	**_findnext64**	**_wfindnext64**

_tfindnexti64	**_findnexti64**	**_findnexti64**	**_wfindnexti64**
_tfindnext32i64	**_findnext32i64**	**_findnext32i64**	**_wfindnext32i64**
_tfindnext64i32	**_findnext64i32**	**_findnext64i32**	**_wfindnext64i32**

Requirements

Function	Required header	Compatibility
_findnext	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_findnext32	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_findnext64	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_findnexti64	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_findnext32i64	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_findnext64i32	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfindnext	<io.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfindnext32	<io.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfindnext64	<io.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfindnexti64	<io.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfindnext32i64	<io.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfindnext64i32	<io.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

System Calls

_finite

Determines whether the given double-precision floating-point value is finite.

```
int _finite(
   double x
);
```

Parameters

x

Double-precision floating-point value.

Return Value

_finite returns a nonzero value if its argument *x* is not infinite; that is, if –INF < *x* < +INF. It returns 0 if the argument is infinite or a NAN.

Requirements

Function	Required header	Compatibility
_finite	<float.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::Double::IsInfinity

See Also

Reference

Floating-Point Support
_isnan
_fpclass

Flag Directives

The first optional field of the format specification is *flags*. A flag directive is a character that justifies output and prints signs, blanks, decimal points, and octal and hexadecimal prefixes. More than one flag directive may appear in a format specification.

Flag Characters

Flag	Meaning	Default
–	Left align the result within the given field width.	Right align.
+	Prefix the output value with a sign (+ or –) if the output value is of a signed type.	Sign appears only for negative signed values (–).
0	If *width* is prefixed with **0**, zeros are added until the minimum width is reached. If **0** and **–** appear, the **0** is ignored. If **0** is specified with an integer format (**i**, **u**, **x**, **X**, **o**, **d**) and a precision specification is also present (for example, `%04.d`), the **0** is ignored.	No padding.
blank (' ')	Prefix the output value with a blank if the output value is signed and positive; the blank is ignored if both the blank and + flags appear.	No blank appears.
#	When used with the **o**, **x**, or **X** format, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.	No blank appears.
	When used with the **e**, **E**, **f**, **a** or **A** format, the **#** flag forces the output value to contain a decimal point in all cases.	Decimal point appears only if digits follow it.
	When used with the **g** or **G** format, the **#** flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros. Ignored when used with **c**, **d**, **i**, **u**, or **s**.	Decimal point appears only if digits follow it. Trailing zeros are truncated.

See Also

Reference

Precision Specification

floor, floorf

Calculates the floor of a value.

```
double floor(
   double x
);
float floor(
   float x
); // C++ only
long double floor(
   long double x
); // C++ only
float floorf(
   float x
);
```

Parameters

x

Floating-point value.

Return Value

The **floor** function returns a floating-point value representing the largest integer that is less than or equal to *x*. There is no error return.

Input	SEH Exception	Matherr Exception
± QNAN,IND	none	_DOMAIN

floor has an implementation that uses Streaming SIMD Extensions 2 (SSE2). See _set_SSE2_enable for information and restrictions on using the SSE2 implementation.

Remarks

C++ allows overloading, so you can call overloads of **floor**. In a C program, **floor** always takes and returns a double.

Requirements

Function	Required header	Compatibility
floor, **floorf**	<math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_floor.c
// This example displays the largest integers
// less than or equal to the floating-point values 2.8
// and -2.8. It then shows the smallest integers greater
// than or equal to 2.8 and -2.8.


#include <math.h>
#include <stdio.h>

int main( void )
{
   double y;
```

```
    y = floor( 2.8 );
    printf( "The floor of 2.8 is %f\n", y );
    y = floor( -2.8 );
    printf( "The floor of -2.8 is %f\n", y );

    y = ceil( 2.8 );
    printf( "The ceil of 2.8 is %f\n", y );
    y = ceil( -2.8 );
    printf( "The ceil of -2.8 is %f\n", y );
}
```

Output

```
The floor of 2.8 is 2.000000
The floor of -2.8 is -3.000000
The ceil of 2.8 is 3.000000
The ceil of -2.8 is -2.000000
```

.NET Framework Equivalent

System::Math::Floor

See Also

Reference

Floating-Point Support
ceil, ceilf
fmod, fmodf

flushall

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _flushall instead.

_flushall

Flushes all streams; clears all buffers.

```
int _flushall( void );
```

Return Value

_flushall returns the number of open streams (input and output). There is no error return.

Remarks

By default, the **_flushall** function writes to appropriate files the contents of all buffers associated with open output streams. All buffers associated with open input streams are cleared of their current contents. (These buffers are normally maintained by the operating system, which determines the optimal time to write the data automatically to disk: when a buffer is full, when a stream is closed, or when a program terminates normally without closing streams.)

If a read follows a call to **_flushall**, new data is read from the input files into the buffers. All streams remain open after the call to **_flushall**.

The commit-to-disk feature of the run-time library lets you ensure that critical data is written directly to disk rather than to the operating system buffers. Without rewriting an existing program, you can enable this feature by linking the program's object files with Commode.obj. In the resulting executable file, calls to **_flushall** write the contents of all buffers to disk. Only **_flushall** and **fflush** are affected by Commode.obj.

For information about controlling the commit-to-disk feature, see Stream I/O, fopen, and _fdopen.

Requirements

Function	Required header	Compatibility
_flushall	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_flushall.c
// This program uses _flushall
// to flush all open buffers.


#include <stdio.h>

int main( void )
{
   int numflushed;

   numflushed = _flushall();
   printf( "There were %d streams flushed\n", numflushed );
}
```

Output

```
There were 3 streams flushed
```

.NET Framework Equivalent

- System::IO::FileStream::Flush
- System::IO::StreamWriter::Flush
- System::IO::TextWriter::Flush

- System::IO::BinaryWriter::Flush

See Also

Reference

Stream I/O
_commit
fclose, _fcloseall
fflush
_flushall
setvbuf

fmod, fmodf

Calculates the floating-point remainder.

```
double fmod(
   double x,
   double y
);
float fmod(
   float x,
   float y
);  // C++ only
long double fmod(
   long double x,
   long double y
);  // C++ only
float fmodf(
   float x,
   float y
);
```

Parameters

x, *y*

Floating-point values.

Return Value

fmod returns the floating-point remainder of *x* / *y*. If the value of *y* is 0.0, **fmod** returns a quiet NaN. For information about representation of a quiet NaN by the **printf** family, see printf.

Remarks

The **fmod** function calculates the floating-point remainder *f* of *x* / *y* such that *x* = *i* * *y* + *f*, where *i* is an integer, *f* has the same sign as *x*, and the absolute value of *f* is less than the absolute value of *y*.

C++ allows overloading, so you can call overloads of **fmod**. In a C program, **fmod** always takes two doubles and returns a double.

Requirements

Function	Required header	Compatibility
fmod, **fmodf**	<math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fmod.c
// This program displays a floating-point remainder.

#include <math.h>
#include <stdio.h>

int main( void )
{
   double w = -10.0, x = 3.0, z;

   z = fmod( w, x );
   printf( "The remainder of %.2f / %.2f is %f\n", w, x, z );
}
```

Output

```
The remainder of -10.00 / 3.00 is -1.000000
```

.NET Framework Equivalent

System::Math::IEEERemainder

See Also

Reference

Floating-Point Support
ceil, ceilf
fabs, fabsf
floor, floorf

fopen, _wfopen

Open a file. These functions are deprecated because more secure versions are available; see fopen_s, _wfopen_s.

```
FILE *fopen(
   const char *filename,
   const char *mode
);
FILE *_wfopen(
   const wchar_t *filename,
   const wchar_t *mode
);
```

Parameters

filename

Filename.

mode

Type of access permitted.

Return Value

Each of these functions returns a pointer to the open file. A null pointer value indicates an error. If *filename* or *mode* is **NULL** or an empty string, these functions trigger the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return **NULL** and set **errno** to **EINVAL**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

More secure versions of these functions exist, see fopen_s, _wfopen_s.

The **fopen** function opens the file specified by *filename*. **_wfopen** is a wide-character version of **fopen**; the arguments to **_wfopen** are wide-character strings. **_wfopen** and **fopen** behave identically otherwise. Simply using **_wfopen** has no effect on the coded character set used in the file stream.

fopen will accept paths that are valid on the file system at the point of execution; UNC paths and paths involving mapped network drives are accepted by **fopen** as long as the system executing the code has access to the share or mapped network drive at the time of execution. Special care must be taken when constructing paths for **fopen** to avoid making assumptions about available drives, paths or network shares in the execution environment.

Always check the return value to see if the pointer is NULL before performing any further operations on the file. If an error occurs, the global variable **errno** is set and may be used to get specific error information. For further information, see errno.

In Visual C++ 2005, **fopen** supports Unicode file streams. A flag specifying the desired encoding may be passed to **fopen** when opening a new file or overwriting an existing file, like this:

```
fopen("newfile.txt", "rw, ccs=<encoding>");
```

Allowed values of the *encoding* include **UNICODE**, **UTF-8**, and **UTF16-LE**. If the file is already in existence and is opened for reading or appending, the Byte Order Mark (BOM) is used to determine the correct encoding. It is not necessary to specify the encoding with a flag. In fact, the flag will be ignored if it conflicts with the type of the file as indicated by the BOM. The flag is only used when no BOM is present or if the file is a new file. The following table summarizes the modes used in for various flags given to fopen and Byte Order Marks used in the file.

Encodings Used Based on Flag and BOM

Flag	No BOM (or new file)	BOM: UTF-8	BOM: UTF-16
UNICODE	**ANSI**	**UTF-8**	**UTF-16LE**
UTF-8	**UTF-8**	**UTF-8**	**UTF-16LE**
UTF-16LE	**UTF-16LE**	**UTF-8**	**UTF-16LE**

If *mode* is "`a, ccs=<encoding>`", **fopen** will first try to open the file with both read and write access. If it succeeds, it will read the BOM to determine the encoding for this file; however, if it fails, it will use the default encoding for the file. In either case, **fopen** will then re-open the file with write-only access. (This applies to mode **a** only, not **a+**.)

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tfopen	**fopen**	**fopen**	**_wfopen**

The character string *mode* specifies the type of access requested for the file, as follows:

"r"

Opens for reading. If the file does not exist or cannot be found, the **fopen** call fails.

"w"

Opens an empty file for writing. If the given file exists, its contents are destroyed.

"a"

Opens for writing at the end of the file (appending) without removing the EOF marker before writing new data to the file; creates the file first if it doesn't exist.

"r+"

Opens for both reading and writing. (The file must exist.)

"w+"

Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.

"a+"

Opens for reading and appending; the appending operation includes the removal of the EOF marker before new data is written to the file and the EOF marker is restored after writing is complete; creates the file first if it doesn't exist.

When a file is opened with the **"a"** or **"a+"** access type, all write operations occur at the end of the file. The file pointer can be repositioned using **fseek** or **rewind**, but is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten.

The **"a"** mode does not remove the EOF marker before appending to the file. After appending has occurred, the MS-DOS TYPE command only shows data up to the original EOF marker and not any data appended to the file. The **"a+"** mode does remove the EOF marker before appending to the file. After appending, the MS-DOS TYPE command shows all data in the file. The **"a+"** mode is required for appending to a stream file that is terminated with the CTRL+Z EOF marker.

When the **"r+"**, **"w+"**, **or "a+"** access type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when you switch between reading and writing, there must be an intervening **fflush**, **fsetpos**, **fseek**, or **rewind** operation. The current position can be specified for the **fsetpos** or **fseek** operation, if desired.

In addition to the above values, the following characters can be included in *mode* to specify the translation mode for newline characters:

t

Open in text (translated) mode. In this mode, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading/writing with **"a+"**, **fopen** checks for a CTRL+Z at the end of the file and removes it, if possible. This is done because using **fseek** and **ftell** to move within a file that ends with a CTRL+Z, may cause **fseek** to behave improperly near the end of the file.

Also, in text mode, carriage return–linefeed combinations are translated into single linefeeds on input, and linefeed characters are translated to carriage return–linefeed combinations on output. When a Unicode stream-I/O function operates in text mode (the default), the source or destination stream is assumed to be a sequence of multibyte characters. Therefore, the Unicode stream-input functions convert multibyte characters to wide characters (as if by a call to the **mbtowc** function). For the same reason, the Unicode stream-output functions convert wide characters to multibyte characters (as if by a call to the **wctomb** function).

b

Open in binary (untranslated) mode; translations involving carriage-return and linefeed characters are suppressed.

If **t** or **b** is not given in *mode*, the default translation mode is defined by the global variable _fmode. If **t** or **b** is prefixed to the

argument, the function fails and returns **NULL**.

For more information about using text and binary modes in Unicode and multibyte stream-I/O, see Text and Binary Mode File I/O and Unicode Stream I/O in Text and Binary Modes.

c

Enable the commit flag for the associated *filename* so that the contents of the file buffer are written directly to disk if either **fflush** or **_flushall** is called.

n

Reset the commit flag for the associated *filename* to "no-commit." This is the default. It also overrides the global commit flag if you link your program with COMMODE.OBJ. The global commit flag default is "no-commit" unless you explicitly link your program with COMMODE.OBJ (see Link Options).

N

Specifies that the file is not inherited by child processes.

S

Specifies that caching is optimized for, but not restricted to, sequential access from disk.

R

Specifies that caching is optimized for, but not restricted to, random access from disk.

T

Specifies a file as temporary. If possible, it is not flushed to disk.

D

Specifies a file as temporary. It is deleted when the last file pointer is closed.

ccs=ENCODING

Specifies the coded character set to use (**UTF-8**, **UTF-16LE**, or **UNICODE**) for this file. Leave unspecified if you want ANSI encoding. This option is available in Visual C++ 2005 and later.

Valid characters for the *mode* string used in **fopen** and **_fdopen** correspond to *oflag* arguments used in _open and _sopen, as follows.

Characters in mode string	Equivalent oflag value for _open/_sopen
a	**_O_WRONLY | _O_APPEND** (usually **_O_WRONLY | _O_CREAT | _O_APPEND**)
a+	**_O_RDWR | _O_APPEND** (usually **_O_RDWR | _O_APPEND | _O_CREAT**)
r	**_O_RDONLY**
r+	**_O_RDWR**
w	**_O_WRONLY** (usually **_O_WRONLY | _O_CREAT | _O_TRUNC**)
w+	**_O_RDWR** (usually **_O_RDWR | _O_CREAT | _O_TRUNC**)
b	**_O_BINARY**
t	**_O_TEXT**
c	None
n	None
S	**_O_SEQUENTIAL**

R	**_O_RANDOM**
T	**_O_SHORTLIVED**
D	**_O_TEMPORARY**
ccs=UNICODE	**_O_WTEXT**
ccs=UTF-8	**_O_UTF8**
ccs=UTF-16LE	**_O_UTF16**

If you are using **rb** mode, won't need to port your code, and expect to read a lot of the file and/or don't care about network performance, memory mapped Win32 files might also be an option.

Requirements

Function	Required header	Compatibility
fopen	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfopen	<stdio.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

The **c**, **n**, **t**, **S**, **R**, **T** and **D** *mode* options are Microsoft extensions for **fopen** and **_fdopen** and should not be used where ANSI portability is desired.

Example

This program opens two files. It uses **fclose** to close the first file and **_fcloseall** to close all remaining files.

```
// crt_fopen.c
// compile with: /W3
// This program opens two files. It uses
// fclose to close the first file and
// _fcloseall to close all remaining files.

#include <stdio.h>

FILE *stream, *stream2;

int main( void )
{
   int numclosed;

   // Open for read (will fail if file "crt_fopen.c" does not exist)
   if( (stream  = fopen( "crt_fopen.c", "r" )) == NULL ) // C4996
   // Note: fopen is deprecated; consider using fopen_s instead
      printf( "The file 'crt_fopen.c' was not opened\n" );
   else
      printf( "The file 'crt_fopen.c' was opened\n" );

   // Open for write
   if( (stream2 = fopen( "data2", "w+" )) == NULL ) // C4996
      printf( "The file 'data2' was not opened\n" );
   else
      printf( "The file 'data2' was opened\n" );

   // Close stream if it is not NULL
```

```
    if( stream)
    {
       if ( fclose( stream ) )
       {
          printf( "The file 'crt_fopen.c' was not closed\n" );
       }
    }

    // All other files are closed:
    numclosed = _fcloseall( );
    printf( "Number of files closed by _fcloseall: %u\n", numclosed );
 }
```

Output

```
The file 'crt_fopen.c' was opened
The file 'data2' was opened
Number of files closed by _fcloseall: 1
```

This program creates a file (or overwrites one if it exists), in text mode using Unicode encoding. It then writes two strings into the file and then closes the file. The output is a file named _wfopen_test.xml, which should contain the data from the output section.

```
// crt__wfopen.c
// compile with: /W3
// This program creates a file (or overwrites one if
// it exists), in text mode using Unicode encoding.
// It then writes two strings into the file
// and then closes the file.

#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <wchar.h>

#define BUFFER_SIZE 50

int main(int argc, char** argv)
{
wchar_t str[BUFFER_SIZE];
size_t  strSize;
FILE*   fileHandle;

    // Create an the xml file in text and Unicode encoding mode.
    if ((fileHandle = _wfopen( L"_wfopen_test.xml",L"wt+,ccs=UNICODE")) == NULL) // C4996
    // Note: _wfopen is deprecated; consider using _wfopen_s instead
    {
        wprintf(L"_wfopen failed!\n");
        return(0);
    }

    // Write a string into the file.
    wcscpy_s(str, sizeof(str)/sizeof(wchar_t), L"<xmlTag>\n");
    strSize = wcslen(str);
    if (fwrite(str, sizeof(wchar_t), strSize, fileHandle) != strSize)
    {
        wprintf(L"fwrite failed!\n");
    }

    // Write a string into the file.
    wcscpy_s(str, sizeof(str)/sizeof(wchar_t), L"</xmlTag>");
    strSize = wcslen(str);
    if (fwrite(str, sizeof(wchar_t), strSize, fileHandle) != strSize)
    {
        wprintf(L"fwrite failed!\n");
    }

    // Close the file.
```

```
    if (fclose(fileHandle))
    {
        wprintf(L"fclose failed!\n");
    }
    return 0;
}
```

.NET Framework Equivalent

- System::IO::File::Open
- System::IO::FileStream::FileStream

See Also

Reference

Stream I/O
Interpretation of Multibyte-Character Sequences
fclose, _fcloseall
_fdopen, _wfdopen
ferror
_fileno
freopen, _wfreopen
_open, _wopen
_setmode
_sopen, _wsopen

fopen_s, _wfopen_s

Open a file. These are versions of fopen, _wfopen with security enhancements as described in Security Enhancements in the CRT.

```
errno_t fopen_s(
   FILE** pFile,
   const char *filename,
   const char *mode
);
errno_t _wfopen_s(
   FILE** pFile,
   const wchar_t *filename,
   const wchar_t *mode
);
```

Parameters

[out] *pFile*

A pointer to the file pointer that will receive the pointer to the opened file.

[in] *filename*

Filename.

[in] *mode*

Type of access permitted.

Return Value

Zero if successful; an error code on failure. See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Error Conditions

pFile	***filename***	***mode***	**Return Value**	**Contents of *pFile***
NULL	any	any	**EINVAL**	unchanged
any	NULL	any	**EINVAL**	unchanged
any	any	NULL	**EINVAL**	unchanged

Remarks

Files opened by **fopen_s** and **_wfopen_s** are not sharable. If you require that a file be sharable, use _fsopen, _wfsopen with the appropriate sharing mode constant (for example, **_SH_DENYNO** for read/write sharing).

The **fopen_s** function opens the file specified by *filename*. **_wfopen_s** is a wide-character version of **fopen_s**; the arguments to **_wfopen_s** are wide-character strings. **_wfopen_s** and **fopen_s** behave identically otherwise.

fopen_s will accept paths that are valid on the file system at the point of execution; UNC paths and paths involving mapped network drives are accepted by **fopen_s** as long as the system executing the code has access to the share or mapped network drive at the time of execution. Special care must be taken when constructing paths for **fopen_s** to avoid making assumptions about available drives, paths or network shares in the execution environment.

These functions validate their parameters. If *pFile*, filename, or mode is a null pointer, these functions generate an invalid parameter exception, as described in Parameter Validation.

Always check the return value to see if the function succeeded before performing any further operations on the file. If an error occurs, the error code is returned and the global variable **errno** is set. For further information, see errno.

In Visual C++ 2005, **fopen_s** supports Unicode file streams. A flag specifying the desired encoding may be passed to **fopen_s** when opening a new file or overwriting an existing file, like this:

```
fopen_s(&fp, "newfile.txt", "rw, ccs=<encoding>");
```

Allowed values of the *encoding* include **UNICODE**, **UTF-8**, and **UTF-16LE**. If the file is already in existence and is opened for reading or appending, the Byte Order Mark (BOM) is used to determine the correct encoding. It is not necessary to specify the encoding with a flag. In fact, the flag will be ignored if it conflicts with the type of the file as indicated by the BOM. The flag is only used when no BOM is present or if the file is a new file. The following table summarizes the modes used in for various flags given to fopen and Byte Order Marks used in the file.

Encodings Used Based on Flag and BOM

Flag	No BOM (or new file)	BOM: UTF-8	BOM: UTF-16
UNICODE	**ANSI**	**UTF-8**	**UTF-16LE**
UTF-8	**UTF-8**	**UTF-8**	**UTF-16LE**
UTF-16LE	**UTF-16LE**	**UTF-8**	**UTF-16LE**

If *mode* is "`a, ccs=<encoding>`", **fopen_s** will first try to open the file with both read and write access. If it succeeds, it will read the BOM to determine the encoding for this file; however, if it fails, it will use the default encoding for the file. In either case, **fopen_s** will then re-open the file with write-only access. (This applies to **a** mode only, not **a+**.)

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tfopen_s	**fopen_s**	**fopen_s**	**_wfopen_s**

The character string *mode* specifies the type of access requested for the file, as follows:

"r"

Opens for reading. If the file does not exist or cannot be found, the **fopen_s** call fails.

"w"

Opens an empty file for writing. If the given file exists, its contents are destroyed.

"a"

Opens for writing at the end of the file (appending) without removing the EOF marker before writing new data to the file; creates the file first if it doesn't exist.

"r+"

Opens for both reading and writing. (The file must exist.)

"w+"

Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.

"a+"

Opens for reading and appending; the appending operation includes the removal of the EOF marker before new data is written to the file and the EOF marker is restored after writing is complete; creates the file first if it doesn't exist.

When a file is opened with the **"a"** or **"a+"** access type, all write operations occur at the end of the file. The file pointer can be repositioned using **fseek** or **rewind**, but is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten.

The **"a"** mode does not remove the EOF marker before appending to the file. After appending has occurred, the MS-DOS TYPE command only shows data up to the original EOF marker and not any data appended to the file. The **"a+"** mode does remove the EOF marker before appending to the file. After appending, the MS-DOS TYPE command shows all data in the file. The **"a+"** mode is required for appending to a stream file that is terminated with the CTRL+Z EOF marker.

When the **"r+", "w+",** or **"a+"** access type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when you switch between reading and writing, there must be an intervening **fflush**, **fsetpos**, **fseek**, or **rewind** operation. The current position can be specified for the **fsetpos** or **fseek** operation, if desired.

In addition to the above values, the following characters can be included in *mode* to specify the translation mode for newline characters:

t

Open in text (translated) mode. In this mode, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading/writing with **"a+"**, **fopen_s** checks for a CTRL+Z at the end of the file and removes it, if possible. This is done because using **fseek** and **ftell** to move within a file that ends with a CTRL+Z, may cause **fseek** to behave improperly near the end of the file.

Also, in text mode, carriage return–linefeed combinations are translated into single linefeeds on input, and linefeed characters are translated to carriage return–linefeed combinations on output. When a Unicode stream-I/O function operates in text mode (the default), the source or destination stream is assumed to be a sequence of multibyte characters. Therefore, the Unicode stream-input functions convert multibyte characters to wide characters (as if by a call to the **mbtowc** function). For the same reason, the Unicode stream-output functions convert wide characters to multibyte characters (as if by a call to the **wctomb** function).

b

Open in binary (untranslated) mode; translations involving carriage-return and linefeed characters are suppressed.

If **t** or **b** is not given in *mode*, the default translation mode is defined by the global variable _fmode. If **t** or **b** is prefixed to the argument, the function fails and returns **NULL**.

For more information about using text and binary modes in Unicode and multibyte stream-I/O, see Text and Binary Mode File I/O and Unicode Stream I/O in Text and Binary Modes.

c

Enable the commit flag for the associated *filename* so that the contents of the file buffer are written directly to disk if either **fflush** or **_flushall** is called.

n

Reset the commit flag for the associated *filename* to "no-commit." This is the default. It also overrides the global commit flag if you link your program with COMMODE.OBJ. The global commit flag default is "no-commit" unless you explicitly link your program with COMMODE.OBJ (see Link Options).

N

Specifies that the file is not inherited by child processes.

S

Specifies that caching is optimized for, but not restricted to, sequential access from disk.

R

Specifies that caching is optimized for, but not restricted to, random access from disk.

T

Specifies a file as temporary. If possible, it is not flushed to disk.

D

Specifies a file as temporary. It is deleted when the last file pointer is closed.

ccs=ENCODING

Specify the coded character set to use (ANSI, UTF-8, UTF-16LE, and UNICODE) for this file. This option is available in Visual C++ 2005 and later.

Valid characters for the *mode* string used in **fopen_s** and **_fdopen** correspond to *oflag* arguments used in _open and _sopen, as follows.

Characters in mode string	Equivalent *oflag* value for _open/_sopen
a	**_O_WRONLY | _O_APPEND** (usually **_O_WRONLY | _O_CREAT | _O_APPEND**)
a+	**_O_RDWR | _O_APPEND** (usually **_O_RDWR | _O_APPEND | _O_CREAT**)
r	**_O_RDONLY**
r+	**_O_RDWR**

w	**_O_WRONLY** (usually **_O_WRONLY | _O_CREAT | _O_TRUNC**)
w+	**_O_RDWR** (usually **_O_RDWR | _O_CREAT | _O_TRUNC**)
b	**_O_BINARY**
t	**_O_TEXT**
c	None
n	None
S	**_O_SEQUENTIAL**
R	**_O_RANDOM**
T	**_O_SHORTLIVED**
D	**_O_TEMPORARY**
ccs=UNICODE	**_O_WTEXT**
ccs=UTF-8	**_O_UTF8**
ccs=UTF-16LE	**_O_UTF16**

If you are using **rb** mode, won't need to port your code, and expect to read a lot of the file and/or don't care about network performance, memory mapped Win32 files might also be an option.

Requirements

Function	Required header	Compatibility
fopen_s	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfopen_s	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

The **c**, **n**, and **t** *mode* options are Microsoft extensions for **fopen_s** and **_fdopen** and should not be used where ANSI portability is desired.

Example

```
// crt_fopen_s.c
// This program opens two files. It uses
// fclose to close the first file and
// _fcloseall to close all remaining files.


#include <stdio.h>

FILE *stream, *stream2;
```

```
int main( void )
{
   int numclosed;
   errno_t err;

   // Open for read (will fail if file "crt_fopen_s.c" does not exist)
   if( (err  = fopen_s( &stream, "crt_fopen_s.c", "r" )) !=0 )
      printf( "The file 'crt_fopen_s.c' was not opened\n" );
   else
      printf( "The file 'crt_fopen_s.c' was opened\n" );

   // Open for write
   if( (err = fopen_s( &stream2, "data2", "w+" )) != 0 )
      printf( "The file 'data2' was not opened\n" );
   else
      printf( "The file 'data2' was opened\n" );

   // Close stream if it is not NULL
   if( stream)
   {
      if ( fclose( stream ) )
      {
         printf( "The file 'crt_fopen_s.c' was not closed\n" );
      }
   }

   // All other files are closed:
   numclosed = _fcloseall( );
   printf( "Number of files closed by _fcloseall: %u\n", numclosed );
}
```

Output

```
The file 'crt_fopen_s.c' was opened
The file 'data2' was opened
Number of files closed by _fcloseall: 1
```

.NET Framework Equivalent

- System::IO::File::Open
- System::IO::FileStream::FileStream

See Also

Reference

Stream I/O
fclose, _fcloseall
_fdopen, _wfdopen
ferror
_fileno
freopen, _wfreopen
_open, _wopen
_setmode

Format Specification Fields: printf and wprintf Functions

This topic describes the syntax for format specifications fields, used in **printf**, **wprintf** and related functions. More secured versions of these functions are available, see printf_s, _printf_s_l, wprintf_s, _wprintf_s_l. For details on the individual functions, see the documentation for those specific functions. For a complete listing of these functions, see Stream I/O.

A format specification, which consists of optional and required fields, has the following form:

%[flags] [width] [**.**precision] [{**h** | **l** | **ll** | **I** | **I32** | **I64**}]type

Each field of the format specification is a single character or a number signifying a particular format option. The simplest format specification contains only the percent sign and a *type* character (for example, `%s`). If a percent sign is followed by a character that has no meaning as a format field, the character is copied to **stdout**. For example, to print a percent-sign character, use **%%**.

The optional fields, which appear before the *type* character, control other aspects of the formatting, as follows:

type

Required character that determines whether the associated *argument* is interpreted as a character, a string, or a number (see the "printf Type Field Characters" table in printf Type Field Characters).

flags

Optional character or characters that control justification of output and printing of signs, blanks, decimal points, and octal and hexadecimal prefixes (see the "Flag Characters" table in Flag Directives). More than one flag can appear in a format specification.

width

Optional number that specifies the minimum number of characters output (see printf Width Specification).

precision

Optional number that specifies the maximum number of characters printed for all or part of the output field, or the minimum number of digits printed for integer values (see the "How Precision Values Affect Type" table in Precision Specification).

h| **l**| **ll**| **I**| **I32**| **I64**

Optional prefixes to *type*-that specify the size of *argument* (see the "Size Prefixes" table in Size and Distance Specification).

Security Note

Ensure that format specification strings are not user-defined. For example, consider a program that prompts the user to enter his name and stores the input in a string variable called **name**. To print **name**, do not do this:

```
printf( name ); // Danger!  If name contains "%s", program will crash
```

Note

Instead, do this:

```
printf( "%s", name );
```

See Also

Reference

printf, _printf_l, wprintf, _wprintf_l
printf_s, _printf_s_l, wprintf_s, _wprintf_s_l
printf_p Positional Parameters

Format Specification Fields: scanf and wscanf Functions

The information here applies to the entire **scanf** family of functions, including the secure versions and describes the symbols used to tell the **scanf** functions how to parse the input stream, such as the input stream **stdin** for **scanf**, into values that are inserted into program variables.

A format specification has the following form:

%[*****] [width] [{**h** | **l** | **ll** | **I64** | **L**}]type

The *format* argument specifies the interpretation of the input and can contain one or more of the following:

- White-space characters: blank (' '); tab ('\t'); or newline ('\n'). A white-space character causes **scanf** to read, but not store, all consecutive white-space characters in the input up to the next non–white-space character. One white-space character in the format matches any number (including 0) and combination of white-space characters in the input.
- Non–white-space characters, except for the percent sign (**%**). A non–white-space character causes **scanf** to read, but not store, a matching non–white-space character. If the next character in the input stream does not match, **scanf** terminates.
- Format specifications, introduced by the percent sign (**%**). A format specification causes **scanf** to read and convert characters in the input into values of a specified type. The value is assigned to an argument in the argument list.

The format is read from left to right. Characters outside format specifications are expected to match the sequence of characters in the input stream; the matching characters in the input stream are scanned but not stored. If a character in the input stream conflicts with the format specification, **scanf** terminates, and the character is left in the input stream as if it had not been read.

When the first format specification is encountered, the value of the first input field is converted according to this specification and stored in the location that is specified by the first *argument*. The second format specification causes the second input field to be converted and stored in the second *argument*, and so on through the end of the format string.

An input field is defined as all characters up to the first white-space character (space, tab, or newline), or up to the first character that cannot be converted according to the format specification, or until the field width (if specified) is reached. If there are too many arguments for the given specifications, the extra arguments are evaluated but ignored. The results are unpredictable if there are not enough arguments for the format specification.

Each field of the format specification is a single character or a number signifying a particular format option. The *type* character, which appears after the last optional format field, determines whether the input field is interpreted as a character, a string, or a number.

The simplest format specification contains only the percent sign and a *type* character (for example, `%s`). If a percent sign (**%**) is followed by a character that has no meaning as a format-control character, that character and the following characters (up to the next percent sign) are treated as an ordinary sequence of characters, that is, a sequence of characters that must match the input. For example, to specify that a percent-sign character is to be input, use `%%`.

An asterisk (*****) following the percent sign suppresses assignment of the next input field, which is interpreted as a field of the specified type. The field is scanned but not stored.

The secure versions (those with the **_s** suffix) of the **scanf** family of functions require that a buffer size parameter be passed immediately following each parameter of type **c**, **C**, **s**, **S** or **[**. For more information on the secure versions of the **scanf** family of functions, see scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l.

See Also
Reference
scanf Width Specification
scanf Type Field Characters
scanf, _scanf_l, wscanf, _wscanf_l
scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l

_fpclass

Returns the status word containing information about the floating-point class.

```
int _fpclass(
   double x
);
```

Parameters

x

Double-precision floating-point value.

Return Value

_fpclass returns an integer value that indicates the floating-point class of its argument *x*. The status word might have one of the following values, defined in Float.h.

Value	Description
_FPCLASS_SNAN	Signaling NaN
_FPCLASS_QNAN	Quiet NaN
_FPCLASS_NINF	Negative infinity (–INF)
_FPCLASS_NN	Negative normalized non-zero
_FPCLASS_ND	Negative denormalized
_FPCLASS_NZ	Negative zero (– 0)
_FPCLASS_PZ	Positive 0 (+0)
_FPCLASS_PD	Positive denormalized
_FPCLASS_PN	Positive normalized non-zero
_FPCLASS_PINF	Positive infinity (+INF)

Requirements

Function	Required header	Compatibility
_fpclass	<float.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

- System::Double::IsInfinity
- System::Double::IsNegativeInfinity
- System::Double::IsPositiveInfinity
- System::Double::IsNan

See Also

Reference

Floating-Point Support

_isnan

_fpieee_flt

Invokes a user-defined trap handler for IEEE floating-point exceptions.

```
int _fpieee_flt(
   unsigned long excCode,
   struct _EXCEPTION_POINTERS *excInfo,
   int handler(_FPIEEE_RECORD *)
);
```

Parameters

excCode

Exception code.

excInfo

Pointer to the Windows NT exception information structure.

handler

Pointer to the user's IEEE trap-handler routine.

Return Value

The return value of **_fpieee_flt** is the value returned by *handler*. As such, the IEEE filter routine might be used in the except clause of a structured exception-handling (SEH) mechanism.

Remarks

The **_fpieee_flt** function invokes a user-defined trap handler for IEEE floating-point exceptions and provides it with all relevant information. This routine serves as an exception filter in the SEH mechanism, which invokes your own IEEE exception handler when necessary.

The **_FPIEEE_RECORD** structure, defined in Fpieee.h, contains information pertaining to an IEEE floating-point exception. This structure is passed to the user-defined trap handler by **_fpieee_flt**.

_FPIEEE_RECORD field	Description
unsigned int RoundingMode, **unsigned int Precision**	These fields contain information about the floating-point environment at the time the exception occurred.
unsigned int Operation	Indicates the type of operation that caused the trap. If the type is a comparison (**_FpCodeCompare**), you can supply one of the special **_FPIEEE_COMPARE_RESULT** values (as defined in Fpieee.h) in the **Result.Value** field. The conversion type (**_FpCodeConvert**) indicates that the trap occurred during a floating-point conversion operation. You can look at the **Operand1** and **Result** types to determine the type of conversion being attempted.
_FPIEEE_VALUE Operand1, **_FPIEEE_VALUE Operand2**, **_FPIEEE_VALUE Operand3**, **_FPIEEE_VALUE Result**	These structures indicate the types and values of the proposed result and operands: **OperandValid** Flag indicating whether the responding value is valid. **Format** Data type of the corresponding value. The format type might be returned even if the corresponding value is not valid. **Value** Result or operand data value. Note: **Operand3** is only used with Itanium Processor Family (IPF) functions.

_FPIEEE_EXCEPTION_FLAGS Cause, **_FPIEEE_EXCEPTION_FLAGS Enable**, **_FPIEEE_EXCEPTION_FLAGS Status**	_FPIEEE_EXCEPTION_FLAGS contains one bit field per type of floating point exception. There is a correspondence between these fields and the arguments used to mask the exceptions supplied to _controlfp. The exact meaning of each bit depends on context: **Cause** Each set bit indicates the particular exception that was raised. **Enable** Each set bit indicates that the particular exception is currently unmasked. **Status** Each set bit indicates that the particular exception is currently pending. This includes exceptions that have not been raised because they were masked by **_controlfp**.

Pending exceptions that are disabled are raised when you enable them. This can result in undefined behavior when using **_fpieee_flt** as an exception filter. Always call _clearfp before enabling floating point exceptions.

Requirements

Function	Required header	Compatibility
_fpieee_flt	<fpieee.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fpieee.c
// This program demonstrates the implementation of
// a user-defined floating-point exception handler using the
// _fpieee_flt function.

#include <fpieee.h>
#include <excpt.h>
#include <float.h>
#include <stddef.h>

int fpieee_handler( _FPIEEE_RECORD * );

int fpieee_handler( _FPIEEE_RECORD *pieee )
{
   // user-defined ieee trap handler routine:
   // there is one handler for all
   // IEEE exceptions

   // Assume the user wants all invalid
   // operations to return 0.

   if ((pieee->Cause.InvalidOperation) &&
       (pieee->Result.Format == _FpFormatFp32))
   {
        pieee->Result.Value.Fp32Value = 0.0F;

        return EXCEPTION_CONTINUE_EXECUTION;
   }
   else
      return EXCEPTION_EXECUTE_HANDLER;
}

#define _EXC_MASK    \
    _EM_UNDERFLOW  + \
    _EM_OVERFLOW   + \
    _EM_ZERODIVIDE + \
    _EM_INEXACT

int main( void )
{
```

```
    // ...

    __try {
       // unmask invalid operation exception
       _controlfp_s(NULL, _EXC_MASK, _MCW_EM);

       // code that may generate
       // fp exceptions goes here
    }
    __except ( _fpieee_flt( GetExceptionCode(),
                 GetExceptionInformation(),
                 fpieee_handler ) ){

       // code that gets control

       // if fpieee_handler returns
       // EXCEPTION_EXECUTE_HANDLER goes here

    }

    // ...
 }
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support
_control87, _controlfp, __control87_2
_controlfp_s

_fpreset

Resets the floating-point package.

```
void _fpreset( void );
```

Remarks

The **_fpreset** function reinitializes the floating-point math package. **_fpreset** is usually used with **signal**, **system**, or the **_exec** or **_spawn** functions. If a program traps floating-point error signals (**SIGFPE**) with **signal**, it can safely recover from floating-point errors by invoking **_fpreset** and using **longjmp**.

This function is deprecated when compiling with /clr (Common Language Runtime Compilation) or **/clr:pure** because the common language runtime only supports the default floating-point precision.

Requirements

Function	Required header	Compatibility
_fpreset	<float.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fpreset.c
// This program uses signal to set up a
// routine for handling floating-point errors.

#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
#include <stdlib.h>
#include <float.h>
#include <math.h>
#include <string.h>

jmp_buf mark;              // Address for long jump to jump to
int     fperr;             // Global error number

void __cdecl fphandler( int sig, int num );   // Prototypes
void fpcheck( void );

int main( void )
{
   double n1 = 5.0;
   double n2 = 0.0;
   double r;
   int jmpret;

   // Unmask all floating-point exceptions.
    _control87( 0, _MCW_EM );
   // Set up floating-point error handler. The compiler
   // will generate a warning because it expects
   // signal-handling functions to take only one argument.
   if( signal( SIGFPE, (void (__cdecl *)(int)) fphandler ) == SIG_ERR )
    {
       fprintf( stderr, "Couldn't set SIGFPE\n" );
       abort();
    }

   // Save stack environment for return in case of error. First
   // time through, jmpret is 0, so true conditional is executed.
```

```
   // If an error occurs, jmpret will be set to -1 and false
   // conditional will be executed.
   jmpret = setjmp( mark );
   if( jmpret == 0 )
   {
      printf( "Dividing %4.3g by %4.3g...\n", n1, n2 );
      r = n1 / n2;
      // This won't be reached if error occurs.
      printf( "\n\n%4.3g / %4.3g = %4.3g\n", n1, n2, r );

      r = n1 * n2;
      // This won't be reached if error occurs.
      printf( "\n\n%4.3g * %4.3g = %4.3g\n", n1, n2, r );
   }
   else
      fpcheck();
}
// fphandler handles SIGFPE (floating-point error) interrupt. Note
// that this prototype accepts two arguments and that the
// prototype for signal in the run-time library expects a signal
// handler to have only one argument.
//
// The second argument in this signal handler allows processing of
// _FPE_INVALID, _FPE_OVERFLOW, _FPE_UNDERFLOW, and
// _FPE_ZERODIVIDE, all of which are Microsoft-specific symbols
// that augment the information provided by SIGFPE. The compiler
// will generate a warning, which is harmless and expected.

void fphandler( int sig, int num )
{
   // Set global for outside check since we don't want
   // to do I/O in the handler.
   fperr = num;

   // Initialize floating-point package. */
   _fpreset();

   // Restore calling environment and jump back to setjmp. Return
   // -1 so that setjmp will return false for conditional test.
   longjmp( mark, -1 );
}

void fpcheck( void )
{
   char fpstr[30];
   switch( fperr )
   {
   case _FPE_INVALID:
       strcpy_s( fpstr, sizeof(fpstr), "Invalid number" );
       break;
   case _FPE_OVERFLOW:
       strcpy_s( fpstr, sizeof(fpstr), "Overflow" );

       break;
   case _FPE_UNDERFLOW:
       strcpy_s( fpstr, sizeof(fpstr), "Underflow" );
       break;
   case _FPE_ZERODIVIDE:
       strcpy_s( fpstr, sizeof(fpstr), "Divide by zero" );
       break;
   default:
       strcpy_s( fpstr, sizeof(fpstr), "Other floating point error" );
       break;
   }
   printf( "Error %d: %s\n", fperr, fpstr );
}
```

Output

```
Dividing    5 by    0...
Error 131: Divide by zero
```

See Also

Reference

Floating-Point Support
_exec, _wexec Functions
signal
_spawn, _wspawn Functions
system, _wsystem

fprintf, _fprintf_l, fwprintf, _fwprintf_l

Print formatted data to a stream. These functions are deprecated because more secure versions are available; see fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l.

```
int fprintf(
   FILE *stream,
   const char *format [,
      argument ]...
);
int _fprintf_l(
   FILE *stream,
   const char *format,
   locale_t locale [,
      argument ]...
);
int fwprintf(
   FILE *stream,
   const wchar_t *format [,
      argument ]...
);
int _fwprintf_l(
   FILE *stream,
   const wchar_t *format,
   locale_t locale [,
      argument ]...
);
```

Parameters

stream

Pointer to **FILE** structure.

format

Format-control string.

argument

Optional arguments.

locale

The locale to use.

Return Value

fprintf returns the number of bytes written. **fwprintf** returns the number of wide characters written. Each of these functions returns a negative value instead when an output error occurs. If *stream* or *format* is **NULL**, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**. The format string is not checked for valid formatting characters as it is when using **fprintf_s** or **fwprintf_s**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

fprintf formats and prints a series of characters and values to the output *stream*. Each function *argument* (if any) is converted and output according to the corresponding format specification in *format*. For **fprintf**, the *format* argument has the same syntax and use that it has in **printf**.

fwprintf is a wide-character version of **fprintf**; in **fwprintf**, *format* is a wide-character string. These functions behave identically if the stream is opened in ANSI mode. **fprintf** does not currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Security Note

Ensure that *format* is not a user-defined string.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_ftprintf	**fprintf**	**fprintf**	**fwprintf**
_ftprintf_l	**_fprintf_l**	**_fprintf_l**	**_fwprintf_l**

For more information, see Format Specifications.

Requirements

Function	Required header	Compatibility
fprintf, **_fprintf_l**	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
fwprintf, **_fwprintf_l**	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fprintf.c
/* This program uses fprintf to format various
 * data and print it to the file named FPRINTF.OUT. It
 * then displays FPRINTF.OUT on the screen using the system
 * function to invoke the operating-system TYPE command.
 */

#include <stdio.h>
#include <process.h>

FILE *stream;

int main( void )
{
   int    i = 10;
   double fp = 1.5;
   char   s[] = "this is a string";
   char   c = '\n';

   fopen_s( &stream, "fprintf.out", "w" );
   fprintf( stream, "%s%c", s, c );
   fprintf( stream, "%d\n", i );
   fprintf( stream, "%f\n", fp );
   fclose( stream );
   system( "type fprintf.out" );
}
```

Output

```
this is a string
10
1.500000
```

.NET Framework Equivalent

System::IO::StreamWriter::Write

See Also

Reference

Stream I/O
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l
fscanf, _fscanf_l, fwscanf, _fwscanf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l

_fprintf_p, _fprintf_p_l, _fwprintf_p, _fwprintf_p_l

Prints formatted data to a stream.

```
int _fprintf_p(
   FILE *stream,
   const char *format [,
      argument ]...
);
int _fprintf_p_l(
   FILE *stream,
   const char *format,
   locale_t locale [,
      argument ]...
);
int _fwprintf_p(
   FILE *stream,
   const wchar_t *format [,
      argument ]...
);
int _fwprintf_p_l(
   FILE *stream,
   const wchar_t *format,
   locale_t locale [,
      argument ]...
);
```

Parameters

stream

Pointer to the **FILE** structure.

format

Format-control string.

argument

Optional arguments.

locale

The locale to use.

Return Value

_fprintf_p and **_fwprintf_p** return the number of characters written or return a negative value when an output error occurs.

Remarks

_fprintf_p formats and prints a series of characters and values to the output *stream*. Each function *argument* (if any) is converted and output according to the corresponding format specification in *format*. For **_fprintf_p**, the *format* argument has the same syntax and use that it has in **_printf_p**. These functions support positional parameters, meaning that the order of the parameters used by the format string can be changed. For more information about positional parameters, see printf_p Positional Parameters.

_fwprintf_p is a wide-character version of **_fprintf_p**; in **_fwprintf_p**, *format* is a wide-character string. These functions behave identically if the stream is opened in ANSI mode. **_fprintf_p** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

Security Note
Ensure that *format* is not a user-defined string.

Like the non-secure versions (see fprintf, _fprintf_l, fwprintf, _fwprintf_l), these functions validate their parameters and invoke the invalid parameter handler, as described in Parameter Validation, if either *stream* or *format* is a null pointer or if there are any unknown or badly formed formatting specifiers. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_ftprintf_p	**_fprintf_p**	**_fprintf_p**	**_fwprintf_p**
_ftprintf_p_l	**_fprintf_p_l**	**_fprintf_p_l**	**_fwprintf_p_l**

For more information, see Format Specifications.

Requirements

Function	Required header	Compatibility
_fprintf_p, **_fprintf_p_l**	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_fwprintf_p, **_fwprintf_p_l**	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fprintf_p.c
// This program uses _fprintf_p to format various
// data and print it to the file named FPRINTF_P.OUT. It
// then displays FPRINTF_P.OUT on the screen using the system
// function to invoke the operating-system TYPE command.
//

#include <stdio.h>
#include <process.h>


int main( void )
{
    FILE    *stream = NULL;
    int     i = 10;
    double  fp = 1.5;
    char    s[] = "this is a string";
    char    c = '\n';

    // Open the file
    if ( fopen_s( &stream, "fprintf_p.out", "w" ) == 0)
    {
        // Format and print data
        _fprintf_p( stream, "%2$s%1$c", c, s );
        _fprintf_p( stream, "%d\n", i );
        _fprintf_p( stream, "%f\n", fp );

        // Close the file
        fclose( stream );
    }

    // Verify our data
    system( "type fprintf_p.out" );
}
```

Output

```
this is a string
10
1.500000
```

.NET Framework Equivalent

System::IO::StreamWriter::Write

See Also

Reference

Stream I/O
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l
fscanf, _fscanf_l, fwscanf, _fwscanf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
printf_p Positional Parameters
_cprintf_p, _cprintf_p_l, _cwprintf_p, _cwprintf_p_l
_cprintf_s, _cprintf_s_l, _cwprintf_s, _cwprintf_s_l
printf_p Positional Parameters
fscanf_s, _fscanf_s_l, fwscanf_s, _fwscanf_s_l

fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l

Print formatted data to a stream. These are versions of fprintf, _fprintf_l, fwprintf, _fwprintf_l with security enhancements as described in Security Enhancements in the CRT.

```
int fprintf_s(
   FILE *stream,
   const char *format [,
      argument ]...
);
int _fprintf_s_l(
   FILE *stream,
   const char *format,
   locale_t locale [,
      argument ]...
);
int fwprintf_s(
   FILE *stream,
   const wchar_t *format [,
      argument ]...
);
int _fwprintf_s_l(
   FILE *stream,
   const wchar_t *format,
   locale_t locale [,
      argument ]…
);
```

Parameters

stream

Pointer to **FILE** structure.

format

Format-control string.

argument

Optional arguments.

locale

The locale to use.

Return Value

fprintf_s returns the number of bytes written. **fwprintf_s** returns the number of wide characters written. Each of these functions returns a negative value instead when an output error occurs.

Remarks

fprintf_s formats and prints a series of characters and values to the output *stream*. Each function *argument* (if any) is converted and output according to the corresponding format specification in *format*. For **fprintf_s**, the *format* argument has the same syntax and use that it has in **printf_s**.

fwprintf_s is a wide-character version of **fprintf_s**; in **fwprintf_s**, *format* is a wide-character string. These functions behave identically if the stream is opened in ANSI mode. **fprintf_s** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

Security Note
Ensure that *format* is not a user-defined string.

Like the non-secure versions (see fprintf, _fprintf_l, fwprintf, _fwprintf_l), these functions validate their parameters and invoke the invalid parameter handler, as described in Parameter Validation, if either *stream* or *format* is a null pointer. These functions differ from the non-secure versions in that the format string itself is also validated. If there are any unknown or badly formed formatting specifiers, these functions generate the invalid parameter exception. In all cases, If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**. See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_ftprintf_s	**fprintf_s**	**fprintf_s**	**fwprintf_s**
_ftprintf_s_l	**_fprintf_s_l**	**_fprintf_s_l**	**_fwprintf_s_l**

For more information, see Format Specifications.

Requirements

Function	Required header	Compatibility
fprintf_s, **_fprintf_s_l**	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
fwprintf_s, **_fwprintf_s_l**	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fprintf_s.c
// This program uses fprintf_s to format various
// data and print it to the file named FPRINTF_S.OUT. It
// then displays FPRINTF_S.OUT on the screen using the system
// function to invoke the operating-system TYPE command.

#include <stdio.h>
#include <process.h>

FILE *stream;

int main( void )
{
   int    i = 10;
   double fp = 1.5;
   char   s[] = "this is a string";
   char   c = '\n';

   fopen_s( &stream, "fprintf_s.out", "w" );
   fprintf_s( stream, "%s%c", s, c );
   fprintf_s( stream, "%d\n", i );
   fprintf_s( stream, "%f\n", fp );
   fclose( stream );
   system( "type fprintf_s.out" );
}
```

Output

```
this is a string
10
1.500000
```

.NET Framework Equivalent

System::IO::StreamWriter::Write

See Also

Reference

Stream I/O

_cprintf, _cprintf_l, _cwprintf, _cwprintf_l

fscanf, _fscanf_l, fwscanf, _fwscanf_l

sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l

fputc, fputwc

Writes a character to a stream.

```
int fputc(
   int c,
   FILE *stream
);
wint_t fputwc(
   wchar_t c,
   FILE *stream
);
```

Parameters

c

Character to be written.

stream

Pointer to **FILE** structure.

Return Value

Each of these functions returns the character written. For **fputc**, a return value of **EOF** indicates an error. For **fputwc**, a return value of **WEOF** indicates an error. If *stream* is **NULL**, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, they return **EOF** and set **errno** to **EINVAL**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

Each of these functions writes the single character *c* to a file at the position indicated by the associated file position indicator (if defined) and advances the indicator as appropriate. In the case of **fputc** and **fputwc**, the file is associated with *stream*. If the file cannot support positioning requests or was opened in append mode, the character is appended to the end of the stream.

The two functions behave identically if the stream is opened in ANSI mode. **fputc** does not currently support output into a UNICODE stream.

The versions with the **_nolock** suffix are identical except that they are not protected from interference by other threads. For more information, see _fputc_nolock, _fputwc_nolock.

Routine-specific remarks follow.

Routine	Remarks
fputc	Equivalent to **putc**, but implemented only as a function, rather than as a function and a macro.
fputwc	Wide-character version of **fputc**. Writes *c* as a multibyte character or a wide character according to whether *stream* is opened in text mode or binary mode.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_fputtc	**fputc**	**fputc**	**fputwc**

Requirements

Function	Required header	Compatibility
fputc	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

fputwc	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fputc.c
// This program uses fputc
// to send a character array to stdout.


#include <stdio.h>

int main( void )
{
   char strptr1[] = "This is a test of fputc!!\n";
   char *p;

   // Print line to stream using fputc.
   p = strptr1;
   while( (*p != '\0') && fputc( *(p++), stdout ) != EOF ) ;

}
```

Output

```
This is a test of fputc!!
```

.NET Framework Equivalent

- System::IO::StreamWriter::Write
- System::Console::Write

See Also

Reference

Stream I/O
fgetc, fgetwc
putc, putwc

_fputc_nolock, _fputwc_nolock

Writes a character to a stream without locking the thread.

```
int _fputc_nolock(
   int c,
   FILE *stream
);
wint_t _fputwc_nolock(
   wchar_t c,
   FILE *stream
);
```

Parameters

c

Character to be written.

stream

Pointer to the **FILE** structure.

Return Value

Each of these functions returns the character written. For error information, see fputc, fputwc.

Remarks

_fputc_nolock and **_fputwc_nolock** are identical to **fputc** and **fputwc**, respectively, except that they are not protected from interference by other threads. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

The two functions behave identically if the stream is opened in ANSI mode. **_fputc_nolock** does not currently support output into a UNICODE stream.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_fputtc_nolock	**_fputc_nolock**	**_fputc_nolock**	**_fputwc_nolock**

Requirements

Function	Required header	Compatibility
_fputc_nolock	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_fputwc_nolock	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fputc_nolock.c
// This program uses _fputc_nolock
// to send a character array to stdout.


#include <stdio.h>

int main( void )
{
   char strptr1[] = "This is a test of _fputc_nolock!!\n";
```

```
    char *p;

    // Print line to stream using fputc.
    p = strptr1;
    while( (*p != '\0') && _fputc_nolock( *(p++), stdout ) != EOF ) ;

  }
```

Output

```
This is a test of _fputc_nolock!!
```

.NET Framework Equivalent

- System::IO::StreamWriter::Write
- System::Console::Write

See Also

Reference

Stream I/O
fgetc, fgetwc
putc, putwc

fputchar

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _fputchar instead.

_fputchar, _fputwchar

Writes a character to **stdout**.

```
int _fputchar(
   int c
);
wint_t _fputwchar(
   wchar_t c
);
```

Parameters

c

Character to be written.

Return Value

Each of these functions returns the character written. For **_fputchar**, a return value of **EOF** indicates an error. For **_fputwchar**, a return value of **WEOF** indicates an error. If c is **NULL**, these functions generate an invalid parameter exception, as described in Parameter Validation. If execution is allowed to continue, they return **EOF** (or **WEOF**) and set **errno** to **EINVAL**.

For more information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Both of these functions writes the single character *c* to **stdout** and advances the indicator as appropriate. **_fputchar** is equivalent to **fputc(stdout)**. It is also equivalent to **putchar**, but implemented only as a function, rather than as a function and a macro. Unlike **fputc** and **putchar**, these functions are not compatible with the ANSI standard.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_fputtchar	**_fputchar**	**_fputchar**	**_fputwchar**

Requirements

Function	Required header	Compatibility
_fputchar	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_fputwchar	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fputchar.c
// This program uses _fputchar
// to send a character array to stdout.


#include <stdio.h>

int main( void )
{
    char strptr[] = "This is a test of _fputchar!!\n";
    char *p = NULL;

    // Print line to stream using _fputchar.
    p = strptr;
```

```
    while( (*p != '\0') && _fputchar( *(p++) ) != EOF )
      ;
}
```

Output

```
This is a test of _fputchar!!
```

.NET Framework Equivalent

- System::IO::StreamWriter::Write
- System::Console::Write

See Also

Reference

Stream I/O
fgetc, fgetwc
putc, putwc

fputs, fputws

Write a string to a stream.

```
int fputs(
   const char *str,
   FILE *stream
);
int fputws(
   const wchar_t *str,
   FILE *stream
);
```

Parameters

str

Output string.

stream

Pointer to **FILE** structure.

Return Value

Each of these functions returns a nonnegative value if it is successful. On an error, **fputs** returns **EOF**, and **fputws** returns **WEOF**. If *str* or *stream* is a null pointer, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **EOF** or **WEOF**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

Each of these functions copies *str* to the output *stream* at the current position. **fputws** copies the wide-character argument *str* to *stream* as a multibyte-character string or a wide-character string according to whether *stream* is opened in text mode or binary mode, respectively. Neither function copies the terminating null character.

The two functions behave identically if the stream is opened in ANSI mode. **fputs** does not currently support output into a UNICODE stream.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_fputts	**fputs**	**fputs**	**fputws**

Requirements

Function	Required header	Compatibility
fputs	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
fputws	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fputs.c
// This program uses fputs to write
// a single line to the stdout stream.

#include <stdio.h>
```

```
int main( void )
{
   fputs( "Hello world from fputs.\n", stdout );
}
```

Output

```
Hello world from fputs.
```

.NET Framework Equivalent

System::IO::StreamWriter::Write

See Also

Reference

Stream I/O
fgets, fgetws
gets, _getws
puts, _putws

fread

Reads data from a stream.

```
size_t fread(
   void *buffer,
   size_t size,
   size_t count,
   FILE *stream
);
```

Parameters

buffer

Storage location for data.

size

Item size in bytes.

count

Maximum number of items to be read.

stream

Pointer to **FILE** structure.

Return Value

fread returns the number of full items actually read, which may be less than *count* if an error occurs or if the end of the file is encountered before reaching *count*. Use the **feof** or **ferror** function to distinguish a read error from an end-of-file condition. If *size* or *count* is 0, **fread** returns 0 and the buffer contents are unchanged. If *stream* or *buffer* is a null pointer, **fread** invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns 0.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

The **fread** function reads up to *count* items of *size* bytes from the input *stream* and stores them in *buffer*. The file pointer associated with *stream* (if there is one) is increased by the number of bytes actually read. If the given stream is opened in text mode, carriage return–linefeed pairs are replaced with single linefeed characters. The replacement has no effect on the file pointer or the return value. The file-pointer position is indeterminate if an error occurs. The value of a partially read item cannot be determined.

This function locks out other threads. If you need a non-locking version, use **_fread_nolock**.

Requirements

Function	Required header	Compatibility
fread	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fread.c
// This program opens a file named FREAD.OUT and
// writes 25 characters to the file. It then tries to open
// FREAD.OUT and read in 25 characters. If the attempt succeeds,
// the program displays the number of actual items read.
```

```
#include <stdio.h>

int main( void )
{
   FILE *stream;
   char list[30];
   int  i, numread, numwritten;

   // Open file in text mode:
   if( fopen_s( &stream, "fread.out", "w+t" ) == 0 )
   {
      for ( i = 0; i < 25; i++ )
         list[i] = (char)('z' - i);
      // Write 25 characters to stream
      numwritten = fwrite( list, sizeof( char ), 25, stream );
      printf( "Wrote %d items\n", numwritten );
      fclose( stream );

   }
   else
      printf( "Problem opening the file\n" );

   if( fopen_s( &stream, "fread.out", "r+t" ) == 0 )
   {
      // Attempt to read in 25 characters
      numread = fread( list, sizeof( char ), 25, stream );
      printf( "Number of items read = %d\n", numread );
      printf( "Contents of buffer = %.25s\n", list );
      fclose( stream );
   }
   else
      printf( "File could not be opened\n" );
}
```

Output

```
Wrote 25 items
Number of items read = 25
Contents of buffer = zyxwvutsrqponmlkjihgfedcb
```

.NET Framework Equivalent

System::IO::FileStream::Read

See Also

Reference

Stream I/O

fwrite

_read

_fread_nolock

Reads data from a stream, without locking other threads.

```
size_t _fread_nolock(
   void *buffer,
   size_t size,
   size_t count,
   FILE *stream
);
```

Parameters

buffer

Storage location for data.

size

Item size in bytes.

count

Maximum number of items to be read.

stream

Pointer to the **FILE** structure.

Return Value

See fread.

Remarks

This function is a non-locking version of **fread**. It is identical to **fread** except that it is not protected from interference by other threads. It might be faster because it does not incur the overhead of locking out other threads. Use this function only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Requirements

Function	**Required header**	**Compatibility**
_fread_nolock	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::IO::FileStream::Read

See Also

Reference

Stream I/O
fwrite
_read

free

Deallocates or frees a memory block.

```
void free(
   void *memblock
);
```

Parameters

memblock

Previously allocated memory block to be freed.

Remarks

The **free** function deallocates a memory block (*memblock*) that was previously allocated by a call to **calloc**, **malloc**, or **realloc**. The number of freed bytes is equivalent to the number of bytes requested when the block was allocated (or reallocated, in the case of **realloc**). If *memblock* is **NULL**, the pointer is ignored and **free** immediately returns. Attempting to free an invalid pointer (a pointer to a memory block that was not allocated by **calloc**, **malloc**, or **realloc**) may affect subsequent allocation requests and cause errors.

In Visual C++ 2005, if an error occurs in freeing the memory, **errno** is set with information from the operating system on the nature of the failure. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

After a memory block has been freed, _heapmin minimizes the amount of free memory on the heap by coalescing the unused regions and releasing them back to the operating system. Freed memory that is not released to the operating system is restored to the free pool and is available for allocation again.

When the application is linked with a debug version of the C run-time libraries, **free** resolves to _free_dbg. For more information about how the heap is managed during the debugging process, see The CRT Debug Heap.

free is marked **__declspec(noalias)**, meaning that the function is guaranteed not to modify global variables. For more information, see noalias.

To free memory allocated with _malloca, use _freea.

Requirements

Function	Required header	Compatibility
free	<stdlib.h> and <malloc.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for malloc.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation
_alloca
calloc
malloc
realloc
_free_dbg
_heapmin
_freea

_free_dbg

Frees a block of memory in the heap (debug version only).

```
void _free_dbg(
   void *userData,
   int blockType
);
```

Parameters

userData

Pointer to the allocated memory block to be freed.

blockType

Type of allocated memory block to be freed: **_CLIENT_BLOCK**, **_NORMAL_BLOCK**, or **_IGNORE_BLOCK**.

Remarks

The **_free_dbg** function is a debug version of the free function. When _DEBUG is not defined, each call to **_free_dbg** is reduced to a call to **free**. Both **free** and **_free_dbg** free a memory block in the base heap, but **_free_dbg** accommodates two debugging features: the ability to keep freed blocks in the heap's linked list to simulate low memory conditions and a block type parameter to free specific allocation types.

_free_dbg performs a validity check on all specified files and block locations before performing the free operation. The application is not expected to provide this information. When a memory block is freed, the debug heap manager automatically checks the integrity of the buffers on either side of the user portion and issues an error report if overwriting has occurred. If the **_CRTDBG_DELAY_FREE_MEM_DF** bit field of the _crtDbgFlag flag is set, the freed block is filled with the value 0xDD, assigned the **_FREE_BLOCK** block type, and kept in the heap's linked list of memory blocks.

In Visual C++ 2005, if an error occurs in freeing the memory, **errno** is set with information from the operating system on the nature of the failure. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see Memory Management and the Debug Heap. For information about the allocation block types and how they are used, see Types of Blocks on the Debug Heap. For information about the differences between calling a standard heap function and its debug version in a debug build of an application, see Using the Debug Version Versus the Base Version.

Requirements

Routine	Required header	Compatibility
_free_dbg	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

For a sample of how to use **_free_dbg**, see crt_dbg2.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_malloc_dbg

_free_locale

Frees a locale object.

```
void _free_locale(
   _locale_t locale
);
```

Parameters

locale

Locale object to free.

Remarks

The **_free_locale** function is used to free the locale object obtained from a call to **_get_current_locale** or **_create_locale**.

The previous name of this function, **__free_locale** (with two leading underscores) has been deprecated.

Requirements

Routine	Required header	Compatibility
_free_locale	<locale.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

No equivalent.

See Also

Reference

_get_current_locale

_create_locale

_freea

Deallocates or frees a memory block.

```
void _freea(
   void *memblock
);
```

Parameters

memblock

Previously allocated memory block to be freed.

Return Value

None.

Remarks

The **_freea** function deallocates a memory block (*memblock*) that was previously allocated by a call to _malloca. **_freea** checks to see if the memory was allocated on the heap or the stack. If it was allocated on the stack, **_freea** does nothing. If it was allocated on the heap, the number of freed bytes is equivalent to the number of bytes requested when the block was allocated. If *memblock* is **NULL**, the pointer is ignored and **_freea** immediately returns. Attempting to free an invalid pointer (a pointer to a memory block that was not allocated by **_malloca**) might affect subsequent allocation requests and cause errors.

_freea calls **free** internally if it finds that the memory is allocated on the heap. Whether the memory is on the heap or the stack is determined by a marker placed in memory at the address immediately preceding the allocated memory.

In Visual C++ 2005, if an error occurs in freeing the memory, **errno** is set with information from the operating system on the nature of the failure. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

After a memory block has been freed, _heapmin minimizes the amount of free memory on the heap by coalescing the unused regions and releasing them back to the operating system. Freed memory that is not released to the operating system is restored to the free pool and is available for allocation again.

A call to **_freea** must accompany all calls to **_malloca**. It is also an error to call **_freea** twice on the same memory. When the application is linked with a debug version of the C run-time libraries, particularly with _malloc_dbg features enabled by defining **_CRTDBG_MAP_ALLOC**, it is easier to find missing or duplicated calls to **_freea**. For more information about how the heap is managed during the debugging process, see The CRT Debug Heap.

_freea is marked **__declspec(noalias)**, meaning that the function is guaranteed not to modify global variables. For more information, see noalias.

Requirements

Function	Required header	Compatibility
_freea	<stdlib.h> and <malloc.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example for _malloca.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation

_malloca

calloc
malloc
_malloc_dbg
realloc
_free_dbg
_heapmin

freopen, _wfreopen

Reassign a file pointer. These functions are deprecated because more secure versions are available; see freopen_s, _wfreopen_s.

```
FILE *freopen(
   const char *path,
   const char *mode,
   FILE *stream
);
FILE *_wfreopen(
   const wchar_t *path,
   const wchar_t *mode,
   FILE *stream
);
```

Parameters

path

Path of new file.

mode

Type of access permitted.

stream

Pointer to **FILE** structure.

Return Value

Each of these functions returns a pointer to the newly opened file. If an error occurs, the original file is closed and the function returns a **NULL** pointer value. If *path*, *mode*, or *stream* is a null pointer, or if *filename* is an empty string, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **NULL**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

More secure versions of these functions exist, see freopen_s, _wfreopen_s.

The **freopen** function closes the file currently associated with *stream* and reassigns *stream* to the file specified by *path*. **_wfreopen** is a wide-character version of **_freopen**; the *path* and *mode* arguments to **_wfreopen** are wide-character strings. **_wfreopen** and **_freopen** behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tfreopen	**freopen**	**freopen**	**_wfreopen**

freopen is typically used to redirect the pre-opened files **stdin**, **stdout**, and **stderr** to files specified by the user. The new file associated with *stream* is opened with *mode,* which is a character string specifying the type of access requested for the file, as follows:

"r"

Opens for reading. If the file does not exist or cannot be found, the **freopen** call fails.

"w"

Opens an empty file for writing. If the given file exists, its contents are destroyed.

"a"

Opens for writing at the end of the file (appending) without removing the EOF marker before writing new data to the file; creates the file first if it does not exist.

"r+"

Opens for both reading and writing. (The file must exist.)

"w+"

Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.

"a+"

Opens for reading and appending; the appending operation includes the removal of the EOF marker before new data is written to the file and the EOF marker is restored after writing is complete; creates the file first if it does not exist.

Use the **"w"** and **"w+"** types with care, as they can destroy existing files.

When a file is opened with the **"a"** or **"a+"** access type, all write operations take place at the end of the file. Although the file pointer can be repositioned using **fseek** or **rewind**, the file pointer is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten.

The **"a"** mode does not remove the EOF marker before appending to the file. After appending has occurred, the MS-DOS TYPE command only shows data up to the original EOF marker and not any data appended to the file. The **"a+"** mode does remove the EOF marker before appending to the file. After appending, the MS-DOS TYPE command shows all data in the file. The **"a+"** mode is required for appending to a stream file that is terminated with the CTRL+Z EOF marker.

When the **"r+"**, **"w+"**, or **"a+"** access type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when you switch between reading and writing, there must be an intervening fsetpos, fseek, or rewind operation. The current position can be specified for the **fsetpos** or **fseek** operation, if desired. In addition to the above values, one of the following characters may be included in the *mode* string to specify the translation mode for new lines.

t

Open in text (translated) mode; carriage return–linefeed (CR-LF) combinations are translated into single linefeed (LF) characters on input; LF characters are translated to CR-LF combinations on output. Also, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading or for writing and reading with **"a+"**, the run-time library checks for a CTRL+Z at the end of the file and removes it, if possible. This is done because using **fseek** and **ftell** to move within a file may cause **fseek** to behave improperly near the end of the file. The **t** option is a Microsoft extension that should not be used where ANSI portability is desired.

b

Open in binary (untranslated) mode; the above translations are suppressed.

If **t** or **b** is not given in *mode*, the default translation mode is defined by the global variable _fmode. If **t** or **b** is prefixed to the argument, the function fails and returns **NULL**.

For a discussion of text and binary modes, see Text and Binary Mode File I/O.

Requirements

Function	Required header	Compatibility
freopen	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfreopen	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_freopen.c
// compile with: /W3
// This program reassigns stderr to the file
// named FREOPEN.OUT and writes a line to that file.
#include <stdio.h>
#include <stdlib.h>

FILE *stream;
```

```
int main( void )
{
   // Reassign "stderr" to "freopen.out":
   stream = freopen( "freopen.out", "w", stderr ); // C4996
   // Note: freopen is deprecated; consider using freopen_s instead

   if( stream == NULL )
      fprintf( stdout, "error on freopen\n" );
   else
   {
      fprintf( stdout, "successfully reassigned\n" ); fflush( stdout );
      fprintf( stream, "This will go to the file 'freopen.out'\n" );
      fclose( stream );
   }
   system( "type freopen.out" );
}
```

Output

```
successfully reassigned
This will go to the file 'freopen.out'
```

.NET Framework Equivalent

- System::IO::File::Open
- System::IO::FileStream::FileStream

See Also

Reference

Stream I/O
fclose, _fcloseall
_fdopen, _wfdopen
_fileno
fopen, _wfopen
_open, _wopen
_setmode

freopen_s, _wfreopen_s

Reassign a file pointer. These are versions of freopen, _wfreopen with security enhancements as described in Security Enhancements in the CRT.

```
errno_t freopen(
   FILE** pFile,
   const char *path,
   const char *mode,
   FILE *stream
);
errno_t _wfreopen(
   FILE** pFile,
   const wchar_t *path,
   const wchar_t *mode,
   FILE *stream
);
```

Parameters

[out] *pFile*

A pointer to the file pointer to be provided by the call.

[in] *path*

Path of new file.

[in] *mode*

Type of access permitted.

[in] *stream*

Pointer to **FILE** structure.

Return Value

Each of these functions returns a pointer to the newly opened file. If an error occurs, the original file is closed and the function returns an error code.

Remarks

The **freopen_s** function closes the file currently associated with *stream* and reassigns *stream* to the file specified by *path*. **_wfreopen_s** is a wide-character version of **_freopen_s**; the *path* and *mode* arguments to **_wfreopen_s** are wide-character strings. **_wfreopen_s** and **_freopen_s** behave identically otherwise.

If any of *pFile*, *path*, *mode*, or *stream* are **NULL**, or if *path* is an empty string, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tfreopen_s	**freopen_s**	**freopen_s**	**_wfreopen_s**

freopen_s is typically used to redirect the pre-opened files **stdin**, **stdout**, and **stderr** to files specified by the user. The new file associated with *stream* is opened with *mode*, which is a character string specifying the type of access requested for the file, as follows:

"r"

Opens for reading. If the file does not exist or cannot be found, the **freopen_s** call fails.

"w"

Opens an empty file for writing. If the given file exists, its contents are destroyed.

"a"

Opens for writing at the end of the file (appending) without removing the EOF marker before writing new data to the file; creates the file first if it does not exist.

"r+"

Opens for both reading and writing. (The file must exist.)

"w+"

Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.

"a+"

Opens for reading and appending; the appending operation includes the removal of the EOF marker before new data is written to the file and the EOF marker is restored after writing is complete; creates the file first if it does not exist.

Use the **"w"** and **"w+"** types with care, as they can destroy existing files.

When a file is opened with the **"a"** or **"a+"** access type, all write operations take place at the end of the file. Although the file pointer can be repositioned using **fseek** or **rewind**, the file pointer is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten.

The **"a"** mode does not remove the EOF marker before appending to the file. After appending has occurred, the MS-DOS TYPE command only shows data up to the original EOF marker and not any data appended to the file. The **"a+"** mode does remove the EOF marker before appending to the file. After appending, the MS-DOS TYPE command shows all data in the file. The **"a+"** mode is required for appending to a stream file that is terminated with the CTRL+Z EOF marker.

When the **"r+", "w+",** or **"a+"** access type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when you switch between reading and writing, there must be an intervening fsetpos, fseek, or rewind operation. The current position can be specified for the **fsetpos** or **fseek** operation, if desired. In addition to the above values, one of the following characters may be included in the *mode* string to specify the translation mode for new lines.

t

Open in text (translated) mode; carriage return–linefeed (CR-LF) combinations are translated into single linefeed (LF) characters on input; LF characters are translated to CR-LF combinations on output. Also, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading or for writing and reading with **"a+"**, the run-time library checks for a CTRL+Z at the end of the file and removes it, if possible. This is done because using **fseek** and **ftell** to move within a file may cause **fseek** to behave improperly near the end of the file. The **t** option is a Microsoft extension that should not be used where ANSI portability is desired.

b

Open in binary (untranslated) mode; the above translations are suppressed.

If **t** or **b** is not given in *mode*, the default translation mode is defined by the global variable _fmode. If **t** or **b** is prefixed to the argument, the function fails and returns **NULL**.

For a discussion of text and binary modes, see Text and Binary Mode File I/O.

Requirements

Function	Required header	Compatibility
freopen_s	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfreopen_s	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_freopen_s.c
// This program reassigns stderr to the file
// named FREOPEN.OUT and writes a line to that file.
```

```
#include <stdio.h>
#include <stdlib.h>

FILE *stream;

int main( void )
{
   errno_t err;
   // Reassign "stderr" to "freopen.out":
   err = freopen_s( &stream, "freopen.out", "w", stderr );

   if( err != 0 )
      fprintf( stdout, "error on freopen\n" );
   else
   {
      fprintf( stdout, "successfully reassigned\n" ); fflush( stdout );
      fprintf( stream, "This will go to the file 'freopen.out'\n" );
      fclose( stream );
   }
   system( "type freopen.out" );
}
```

Output

```
successfully reassigned
This will go to the file 'freopen.out'
```

.NET Framework Equivalent

- System::IO::File::Open
- System::IO::FileStream::FileStream

See Also

Reference

Stream I/O
freopen, _wfreopen
fclose, _fcloseall
_fdopen, _wfdopen
_fileno
fopen, _wfopen
_open, _wopen
_setmode

frexp

Gets the mantissa and exponent of a floating-point number.

```
double frexp(
   double x,
   int *expptr
);
float frexp(
   float x,
   int * expptr
);  // C++ only
long double frexp(
   long double x,
   int * expptr
);  // C++ only
```

Parameters

x

Floating-point value.

expptr

Pointer to stored integer exponent.

Return Value

frexp returns the mantissa. If *x* is 0, the function returns 0 for both the mantissa and the exponent. If *expptr* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns 0.

Remarks

The **frexp** function breaks down the floating-point value (*x*) into a mantissa (*m*) and an exponent (*n*), such that the absolute value of *m* is greater than or equal to 0.5 and less than 1.0, and $x = m*2^n$. The integer exponent *n* is stored at the location pointed to by *expptr*.

C++ allows overloading, so you can call overloads of **frexp**. In a C program, **frexp** always takes a double and an integer and returns a double.

Requirements

Function	Required header	Compatibility
frexp	<math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_frexp.c
// This program calculates frexp( 16.4, &n )
// then displays y and n.


#include <math.h>
#include <stdio.h>

int main( void )
{
   double x, y;
   int n;
```

```
    x = 16.4;
    y = frexp( x, &n );
    printf( "frexp( %f, &n ) = %f, n = %d\n", x, y, n );
}
```

Output

```
frexp( 16.400000, &n ) = 0.512500, n = 5
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support
ldexp
modf, modff

fscanf, _fscanf_l, fwscanf, _fwscanf_l

Read formatted data from a stream. These functions are deprecated because more secure versions are available; see fscanf_s, _fscanf_s_l, fwscanf_s, _fwscanf_s_l.

```
int fscanf(
   FILE *stream,
   const char *format [,
      argument ]...
);
int _fscanf_l(
   FILE *stream,
   const char *format,
   locale_t locale [,
      argument ]...
);
int fwscanf(
   FILE *stream,
   const wchar_t *format [,
      argument ]...
);
int _fwscanf_l(
   FILE *stream,
   const wchar_t *format,
   locale_t locale [,
      argument ]...
);
```

Parameters

stream

Pointer to **FILE** structure.

format

Format-control string.

argument

Optional arguments.

locale

The locale to use.

Return Value

Each of these functions returns the number of fields successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned. If an error occurs, or if the end of the file stream is reached before the first conversion, the return value is **EOF** for **fscanf** and **fwscanf**.

These functions validate their parameters. If *stream* or *format* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **EOF** and set **errno** to **EINVAL**.

Remarks

The **fscanf** function reads data from the current position of *stream* into the locations given by *argument* (if any). Each *argument* must be a pointer to a variable of a type that corresponds to a type specifier in *format*. *format* controls the interpretation of the input fields and has the same form and function as the *format* argument for **scanf**; see scanf for a description of *format*.

fwscanf is a wide-character version of **fscanf**; the format argument to **fwscanf** is a wide-character string. These functions behave identically identically if the stream is opened in ANSI mode. **fscanf** doesn't currently support input from a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the

current thread locale.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_ftscanf	**fscanf**	**fscanf**	**fwscanf**
_ftscanf_l	**_fscanf_l**	**_fscanf_l**	**_fwscanf_l**

For more information, see Format Specification Fields – scanf functions and wscanf Functions.

Requirements

Function	Required header	Compatibility
fscanf, **_fscanf_l**	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
fwscanf, **_fwscanf_l**	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fscanf.c
// compile with: /W3
// This program writes formatted
// data to a file. It then uses fscanf to
// read the various data back from the file.

#include <stdio.h>

FILE *stream;

int main( void )
{
   long l;
   float fp;
   char s[81];
   char c;

   if( fopen_s( &stream, "fscanf.out", "w+" ) != 0 )
      printf( "The file fscanf.out was not opened\n" );
   else
   {
      fprintf( stream, "%s %ld %f%c", "a-string",
               65000, 3.14159, 'x' );
      // Security caution!
      // Beware loading data from a file without confirming its size,
      // as it may lead to a buffer overrun situation.

      // Set pointer to beginning of file:
      fseek( stream, 0L, SEEK_SET );

      // Read data back from file:
      fscanf( stream, "%s", s );   // C4996
      fscanf( stream, "%ld", &l ); // C4996

      fscanf( stream, "%f", &fp ); // C4996
      fscanf( stream, "%c", &c );  // C4996
      // Note: fscanf is deprecated; consider using fscanf_s instead

      // Output data read:
      printf( "%s\n", s );
      printf( "%ld\n", l );
```

```
        printf( "%f\n", fp );
        printf( "%c\n", c );

        fclose( stream );
    }
}
```

Output

```
a-string
65000
3.141590
x
```

.NET Framework Equivalent

System::IO::StreamReader::ReadLine. See also **Parse** methods, such as System::Double::Parse.

See Also

Reference

Stream I/O
_cscanf, _cscanf_l, _cwscanf, _cwscanf_l
fprintf, _fprintf_l, fwprintf, _fwprintf_l
scanf, _scanf_l, wscanf, _wscanf_l
sscanf, _sscanf_l, swscanf, _swscanf_l
fscanf_s, _fscanf_s_l, fwscanf_s, _fwscanf_s_l

fscanf_s, _fscanf_s_l, fwscanf_s, _fwscanf_s_l

Read formatted data from a stream. These are versions of fscanf, _fscanf_l, fwscanf, _fwscanf_l with security enhancements as described in Security Enhancements in the CRT.

```
int fscanf_s(
   FILE *stream,
   const char *format [,
      argument ]...
);
int _fscanf_s_l(
   FILE *stream,
   const char *format,
   locale_t locale [,
      argument ]...
);
int fwscanf_s(
   FILE *stream,
   const wchar_t *format [,
      argument ]...
);
int _fwscanf_s_l(
   FILE *stream,
   const wchar_t *format,
   locale_t locale [,
      argument ]...
);
```

Parameters

stream

Pointer to **FILE** structure.

format

Format-control string.

argument

Optional arguments.

locale

The locale to use.

Return Value

Each of these functions returns the number of fields successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned. If an error occurs, or if the end of the file stream is reached before the first conversion, the return value is **EOF** for **fscanf_s** and **fwscanf_s**.

These functions validate their parameters. If stream is ian nvalid file pointer, or *format* is a null pointer, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return **EOF** and set **errno** to **EINVAL**.

Remarks

The **fscanf_s** function reads data from the current position of *stream* into the locations given by *argument* (if any). Each *argument* must be a pointer to a variable of a type that corresponds to a type specifier in *format*. *format* controls the interpretation of the input fields and has the same form and function as the *format* argument for **scanf_s**; see Format Specification Fields – scanf functions and wscanf Functions for a description of *format*. **fwscanf_s** is a wide-character version of **fscanf_s**; the format argument to **fwscanf_s** is a wide-character string. These functions behave identically if the stream is opened in ANSI mode. **fscanf_s** doesn't currently support input from a UNICODE stream.

The main difference between the secure functions (with the **_s** suffix) and the older functions is that the secure functions require the size of each **c**, **C**, **s**, **S** and **[** type field to be passed as an argument immediately following the variable. For more

information, see scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l and scanf Width Specification.

Note
The size parameter is of type **unsigned**, not **size_t**.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_ftscanf_s	**fscanf_s**	**fscanf_s**	**fwscanf_s**
_ftscanf_s_l	**_fscanf_s_l**	**_fscanf_s_l**	**_fwscanf_s_l**

Requirements

Function	Required header	Compatibility
fscanf_s, **_fscanf_s_l**	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
fwscanf_s, **_fwscanf_s_l**	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fscanf_s.c
// This program writes formatted
// data to a file. It then uses fscanf to
// read the various data back from the file.

#include <stdio.h>

FILE *stream;

int main( void )
{
   long l;
   float fp;
   char s[81];
   char c;

   errno_t err = fopen_s( &stream, "fscanf.out", "w+" );
   if( err )
      printf_s( "The file fscanf.out was not opened\n" );
   else
   {
      fprintf_s( stream, "%s %ld %f%c", "a-string",
               65000, 3.14159, 'x' );
      // Set pointer to beginning of file:
      fseek( stream, 0L, SEEK_SET );

      // Read data back from file:
      fscanf_s( stream, "%s", s, 81 );
      fscanf_s( stream, "%ld", &l );

      fscanf_s( stream, "%f", &fp );
      fscanf_s( stream, "%c", &c, 1 );

      // Output data read:
      printf( "%s\n", s );
```

```
        printf( "%ld\n", l );
        printf( "%f\n", fp );
        printf( "%c\n", c );

        fclose( stream );
    }
}
```

Output

```
a-string
65000
3.141590
x
```

.NET Framework Equivalent

System::IO::StreamReader::ReadLine. See also **Parse** methods, such as System::Double::Parse.

See Also

Reference

Stream I/O
_cscanf_s, _cscanf_s_l, _cwscanf_s, _cwscanf_s_l
fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l
scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l
sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l
fscanf, _fscanf_l, fwscanf, _fwscanf_l

fseek, _fseeki64

Moves the file pointer to a specified location.

```
int fseek(
   FILE *stream,
   long offset,
   int origin
);
int _fseeki64(
   FILE *stream,
   __int64 offset,
   int origin
);
```

Parameters

stream

Pointer to **FILE** structure.

offset

Number of bytes from *origin*.

origin

Initial position.

Return Value

If successful, **fseek** and **_fseeki64** returns 0. Otherwise, it returns a nonzero value. On devices incapable of seeking, the return value is undefined. If *stream* is a null pointer, or if *origin* is not one of allowed values described below, **fseek** and **_fseeki64** invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1.

Remarks

The **fseek** and **_fseeki64** functions moves the file pointer (if any) associated with *stream* to a new location that is *offset* bytes from *origin*. The next operation on the stream takes place at the new location. On a stream open for update, the next operation can be either a read or a write. The argument origin must be one of the following constants, defined in STDIO.H:

SEEK_CUR

Current position of file pointer.

SEEK_END

End of file.

SEEK_SET

Beginning of file.

You can use **fseek** and **_fseeki64** to reposition the pointer anywhere in a file. The pointer can also be positioned beyond the end of the file. **fseek** and **_fseeki64** clears the end-of-file indicator and negates the effect of any prior **ungetc** calls against *stream*.

When a file is opened for appending data, the current file position is determined by the last I/O operation, not by where the next write would occur. If no I/O operation has yet occurred on a file opened for appending, the file position is the start of the file.

For streams opened in text mode, **fseek** and **_fseeki64** have limited use, because carriage return–linefeed translations can cause **fseek** and **_fseeki64** to produce unexpected results. The only **fseek** and **_fseeki64** operations guaranteed to work on streams opened in text mode are:

- Seeking with an offset of 0 relative to any of the origin values.
- Seeking from the beginning of the file with an offset value returned from a call to **ftell** when using **fseek** or **_ftelli64**

when using **_fseeki64**.

Also in text mode, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading/writing, **fopen** and all related routines check for a CTRL+Z at the end of the file and remove it if possible. This is done because using the combination of **fseek** and **ftell** or **_fseeki64** and **_ftelli64**, to move within a file that ends with a CTRL+Z may cause **fseek** or **_fseeki64** to behave improperly near the end of the file.

When the CRT opens a file that begins with a Byte Order Mark (BOM), the file pointer is positioned after the BOM (that is, at the start of the file's actual content). If you have to **fseek** to the beginning of the file, use **ftell** to get the initial position and **fseek** to it rather than to position 0.

This function locks out other threads during execution and is therefore thread-safe. For a non-locking version, see _fseek_nolock, _fseeki64_nolock.

Requirements

Function	Required header	Compatibility
fseek	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_fseeki64	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fseek.c
// This program opens the file FSEEK.OUT and
// moves the pointer to the file's beginning.

#include <stdio.h>

int main( void )
{
   FILE *stream;
   char line[81];
   int  result;

   if ( fopen_s( &stream, "fseek.out", "w+" ) != 0 )
   {
      printf( "The file fseek.out was not opened\n" );
      return -1;
   }
   fprintf( stream, "The fseek begins here: "
                    "This is the file 'fseek.out'.\n" );
   result = fseek( stream, 23L, SEEK_SET);
   if( result )
      perror( "Fseek failed" );
   else
   {
      printf( "File pointer is set to middle of first line.\n" );
      fgets( line, 80, stream );
      printf( "%s", line );
    }
   fclose( stream );
}
```

Output

```
File pointer is set to middle of first line.
This is the file 'fseek.out'.
```

.NET Framework Equivalent

- System::IO::FileStream::Position
- System::IO::FileStream::Seek

See Also

Reference

Stream I/O
fopen, _wfopen
ftell, _ftelli64
_lseek, _lseeki64
rewind

_fseek_nolock, _fseeki64_nolock

Moves the file pointer to a specified location.

```
int _fseek_nolock(
   FILE *stream,
   long offset,
   int origin
);
int _fseeki64_nolock(
   FILE *stream,
   __int64 offset,
   int origin
);
```

Parameters

stream

Pointer to the **FILE** structure.

offset

Number of bytes from *origin*.

origin

Initial position.

Return Value

Same as fseek, _fseeki64 respectively.

Remarks

These functions are the non-locking versions of **fseek** and **_fseeki64**, respectively. These are identical to **fseek** and **_fseeki64** except that they are not protected from interference by other threads. These functions might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Requirements

Function	Required header	Compatibility
fseek	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_fseeki64	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

- System::IO::FileStream::Position
- System::IO::FileStream::Seek

See Also

Reference

Stream I/O
ftell, _ftelli64
_lseek, _lseeki64
rewind

fsetpos

Sets the stream-position indicator.

```
int fsetpos(
   FILE *stream,
   const fpos_t *pos
);
```

Parameters

stream

Pointer to **FILE** structure.

pos

Position-indicator storage.

Return Value

If successful, **fsetpos** returns 0. On failure, the function returns a nonzero value and sets **errno** to one of the following manifest constants (defined in ERRNO.H): **EBADF**, which means the file is not accessible or the object that *stream* points to is not a valid file structure; or **EINVAL**, which means an invalid value for *stream* or *pos* was passed. If an invalid parameter is passed in, these functions invoke the invalid parameter handler, as described in Parameter Validation.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

Remarks

The **fsetpos** function sets the file-position indicator for *stream* to the value of *pos,* which is obtained in a prior call to **fgetpos** against *stream*. The function clears the end-of-file indicator and undoes any effects of ungetc on *stream*. After calling **fsetpos**, the next operation on *stream* may be either input or output.

Requirements

Function	Required header	Compatibility
fsetpos	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for fgetpos.

.NET Framework Equivalent

System::IO::FileStream::Position

See Also

Reference

Stream I/O

fgetpos

_fsopen, _wfsopen

Opens a stream with file sharing.

```
FILE *_fsopen(
   const char *filename,
   const char *mode,
   int shflag
);
FILE *_wfsopen(
   const wchar_t *filename,
   const wchar_t *mode,
   int shflag
);
```

Parameters

filename

Name of the file to open.

mode

Type of access permitted.

shflag

Type of sharing allowed.

Return Value

Each of these functions returns a pointer to the stream. A null pointer value indicates an error. If *filename* or *mode* is **NULL** or an empty string, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return **NULL** and set **errno** to **EINVAL**.

For more information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_fsopen** function opens the file specified by *filename* as a stream and prepares the file for subsequent shared reading or writing, as defined by the mode and *shflag* arguments. **_wfsopen** is a wide-character version of **_fsopen**; the *filename* and *mode* arguments to **_wfsopen** are wide-character strings. **_wfsopen** and **_fsopen** behave identically otherwise.

The character string *mode* specifies the type of access requested for the file, as shown in the following table.

Term	Definition
"r"	Opens for reading. If the file does not exist or cannot be found, the **_fsopen** call fails.
"w"	Opens an empty file for writing. If the given file exists, its contents are destroyed.
"a"	Opens for writing at the end of the file (appending); creates the file first if it does not exist.
"r+"	Opens for both reading and writing. (The file must exist.)
"w+"	Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.
"a+"	Opens for reading and appending; creates the file first if it does not exist.

Use the **"w"** and **"w+"** types with care, as they can destroy existing files.

When a file is opened with the **"a"** or **"a+"** access type, all write operations occur at the end of the file. The file pointer can be repositioned using **fseek** or **rewind**, but it is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten. When the **"r+"**, **"w+"**, or **"a+"** access type is specified, both reading and writing are

allowed (the file is said to be open for update). However, when switching between reading and writing, there must be an intervening fsetpos, fseek, or rewind operation. The current position can be specified for the **fsetpos** or **fseek** operation, if desired. In addition to the above values, one of the following characters can be included in *mode* to specify the translation mode for new lines.

Term	Definition
t	Opens a file in text (translated) mode. In this mode, carriage return–line feed (CR-LF) combinations are translated into single line feeds (LF) on input and LF characters are translated to CR-LF combinations on output. Also, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading or reading/writing, **_fsopen** checks for a CTRL+Z at the end of the file and removes it, if possible. This is done because using **fseek** and **ftell** to move within a file that ends with a CTRL+Z might cause **fseek** to behave improperly near the end of the file.
b	Opens a file in binary (untranslated) mode; the above translations are suppressed.

If **t** or **b** is not given in *mode*, the translation mode is defined by the default-mode variable **_fmode**. If **t** or **b** is prefixed to the argument, the function fails and returns **NULL**. For a discussion of text and binary modes, see Text and Binary Mode File I/O.

The argument *shflag* is a constant expression consisting of one of the following manifest constants, defined in Share.h.

Term	Definition
_SH_COMPAT	Sets Compatibility mode for 16-bit applications.
_SH_DENYNO	Permits read and write access.
_SH_DENYRD	Denies read access to the file.
_SH_DENYRW	Denies read and write access to the file.
_SH_DENYWR	Denies write access to the file.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tfsopen	**_fsopen**	**_fsopen**	**_wfsopen**

Requirements

Function	Required header	Optional headers	Compatibility
_fsopen	<stdio.h>	<share.h> For manifest constant for *shflag* parameter.	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfsopen	<stdio.h> or <wchar.h>	<share.h> For manifest constant for *shflag* parameter.	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

```
// crt_fsopen.c

#include <stdio.h>
#include <stdlib.h>
#include <share.h>

int main( void )
```

```
{
   FILE *stream;

   // Open output file for writing. Using _fsopen allows us to
   // ensure that no one else writes to the file while we are
   // writing to it.
    //
   if( (stream = _fsopen( "outfile", "wt", _SH_DENYWR )) != NULL )
   {
      fprintf( stream, "No one else in the network can write "
                       "to this file until we are done.\n" );
      fclose( stream );
   }
   // Now others can write to the file while we read it.
   system( "type outfile" );
}
```

Output

```
No one else in the network can write to this file until we are done.
```

.NET Framework Equivalent

- System::IO::File::Open
- System::IO::FileStream::FileStream

See Also

Reference

Stream I/O
fclose, _fcloseall
_fdopen, _wfdopen
ferror
_fileno
fopen, _wfopen
freopen, _wfreopen
_open, _wopen
_setmode
_sopen, _wsopen

_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32

Gets information about an open file.

```
int _fstat(
   int fd,
   struct _stat *buffer
);
int _fstat32(
   int fd,
   struct __stat32 *buffer
);
int _fstat64(
   int fd,
   struct __stat64 *buffer
);
int _fstati64(
   int fd,
      struct _stati64 *buffer
);
int _fstat32i64(
   int fd,
   struct _stat32i64 *buffer
);
int _fstat64i32(
   int fd,
   struct _stat64i32 *buffer
);
```

Parameters

fd

File descriptor of the open file.

buffer

Pointer to the structure to store results.

Return Value

Returns 0 if the file-status information is obtained. A return value of –1 indicates an error. If the file descriptor is invalid or *buffer* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EBADF**, in the case of an invalid file descriptor, or to **EINVAL**, if *buffer* is **NULL**.

Remarks

The **_fstat** function obtains information about the open file associated with *fd* and stores it in the structure pointed to by *buffer*. The **_stat** structure, defined in SYS\Stat.h, contains the following fields.

st_atime

Time of the last file access.

st_ctime

Time of the creation of the file.

st_dev

If a device, *fd*; otherwise 0.

st_mode

Bit mask for file-mode information. The **_S_IFCHR** bit is set if *fd* refers to a device. The **_S_IFREG** bit is set if *fd* refers to an ordinary file. The read/write bits are set according to the file's permission mode. **_S_IFCHR** and other constants are defined in SYS\Stat.h.

st_mtime

Time of the last modification of the file.

st_nlink

Always 1 on non-NTFS file systems.

st_rdev

If a device, *fd*; otherwise 0.

st_size

Size of the file in bytes.

If *fd* refers to a device, the **st_atime**, **st_ctime**, **st_mtime**, and **st_size** fields are not meaningful.

Because Stat.h uses the _dev_t type, which is defined in Types.h, you must include Types.h before Stat.h in your code.

_fstat64, which uses the **__stat64** structure, allows file-creation dates to be expressed up through 23:59:59, December 31, 3000, UTC; whereas the other functions only represent dates through 03:14:07 January 19, 2038, UTC. Midnight, January 1, 1970, is the lower bound of the date range for all these functions.

Variations of these functions support 32-bit or 64-bit time types and 32-bit or 64-bit file lengths. The first numerical suffix (**32** or **64**) indicates the size of the time type used; the second suffix is either **i32** or **i64**, indicating whether the file size is represented as a 32-bit or 64-bit integer.

In Visual C++ 2005, **_fstat** is equivalent to **_fstat64i32**, and **struct _stat** contains a 64-bit time. This is true unless **_USE_32BIT_TIME_T** is defined, in which case the old behavior is in effect; **_fstat** uses a 32-bit time, and **struct _stat** contains a 32-bit time. The same is true for **_fstati64**.

Time Type and File Length Type Variations of _stat

Functions	_USE_32BIT_TIME_T defined?	Time type	File length type
_fstat	Not defined	64-bit	32-bit
_fstat	Defined	32-bit	32-bit
_fstat32	Not affected by the macro definition	32-bit	32-bit
_fstat64	Not affected by the macro definition	64-bit	64-bit
_fstati64	Not defined	64-bit	64-bit
_fstati64	Defined	32-bit	64-bit
_fstat32i64	Not affected by the macro definition	32-bit	64-bit
_fstat64i32	Not affected by the macro definition	64-bit	32-bit

Requirements

Function	Required header	Compatibility
_fstat	<sys/stat.h> and <sys/types.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_fstat32	<sys/stat.h> and <sys/types.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_fstat64	<sys/stat.h> and <sys/types.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_fstati64	<sys/stat.h> and <sys/types.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_fstat32i64	<sys/stat.h> and <sys/types.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_fstat64i32	<sys/stat.h> and <sys/types.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fstat.c
// This program uses _fstat to report
// the size of a file named F_STAT.OUT.


#include <io.h>
#include <fcntl.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <share.h>

int main( void )
{
   struct _stat buf;
   int fd, result;
   char buffer[] = "A line to output";
   char timebuf[26];
   errno_t err;

   _sopen_s( &fd,
             "f_stat.out",
             _O_CREAT | _O_WRONLY | _O_TRUNC,
             _SH_DENYNO,
             _S_IREAD | _S_IWRITE );
   if( fd != -1 )
      _write( fd, buffer, strlen( buffer ) );

   // Get data associated with "fd":
   result = _fstat( fd, &buf );

   // Check if statistics are valid:
   if( result != 0 )
   {
      if (errno == EBADF)
        printf( "Bad file descriptor.\n" );
      else if (errno == EINVAL)
        printf( "Invalid argument to _fstat.\n" );
   }
   else
   {
      printf( "File size     : %ld\n", buf.st_size );
      err = ctime_s(timebuf, 26, &buf.st_mtime);
      if (err)
      {
         printf("Invalid argument to ctime_s.");
         exit(1);
      }
      printf( "Time modified : %s", timebuf );
   }
```

```
    _close( fd );
}
```

Sample Output

```
File size     : 16
Time modified : Wed May 07 15:25:11 2003
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

File Handling
_access, _waccess
_chmod, _wchmod
_filelength, _filelengthi64
_stat, _wstat Functions

ftell, _ftelli64

Gets the current position of a file pointer.

```
long ftell(
   FILE *stream
);
__int64 _ftelli64(
   FILE *stream
);
```

Parameters

stream

Target **FILE** structure.

Return Value

ftell and **_ftelli64** return the current file position. The value returned by **ftell** and **_ftelli64** may not reflect the physical byte offset for streams opened in text mode, because text mode causes carriage return–linefeed translation. Use **ftell** with **fseek** or **_ftelli64** with **_fseeki64** to return to file locations correctly. On error, **ftell** and **_ftelli64** invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return –1L and set **errno** to one of two constants, defined in ERRNO.H. The **EBADF** constant means the *stream* argument is not a valid file pointer value or does not refer to an open file. **EINVAL** means an invalid *stream* argument was passed to the function. On devices incapable of seeking (such as terminals and printers), or when *stream* does not refer to an open file, the return value is undefined.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

Remarks

The **ftell** and **_ftelli64** functions retrieve the current position of the file pointer (if any) associated with *stream*. The position is expressed as an offset relative to the beginning of the stream.

Note that when a file is opened for appending data, the current file position is determined by the last I/O operation, not by where the next write would occur. For example, if a file is opened for an append and the last operation was a read, the file position is the point where the next read operation would start, not where the next write would start. (When a file is opened for appending, the file position is moved to end of file before any write operation.) If no I/O operation has yet occurred on a file opened for appending, the file position is the beginning of the file.

In text mode, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading/writing, **fopen** and all related routines check for a CTRL+Z at the end of the file and remove it if possible. This is done because using the combination of **ftell** and **fseek** or **_ftelli64** and **_fseeki64**, to move within a file that ends with a CTRL+Z may cause **ftell** or **_ftelli64** to behave improperly near the end of the file.

This function locks the calling thread during execution and is therefore thread-safe. For a non-locking version, see **_ftell_nolock**.

Requirements

Function	Required header	Optional headers	Compatibility
ftell	<stdio.h>	<errno.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ftelli64	<stdio.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_ftell.c
```

```
// This program opens a file named CRT_FTELL.C
// for reading and tries to read 100 characters. It
// then uses ftell to determine the position of the
// file pointer and displays this position.


#include <stdio.h>

FILE *stream;

int main( void )
{
   long position;
   char list[100];
   if( fopen_s( &stream, "crt_ftell.c", "rb" ) == 0 )
   {
      // Move the pointer by reading data:
      fread( list, sizeof( char ), 100, stream );
      // Get position after read:
      position = ftell( stream );
      printf( "Position after trying to read 100 bytes: %ld\n",
              position );
      fclose( stream );
   }
}
```

Output

```
Position after trying to read 100 bytes: 100
```

.NET Framework Equivalent

System::IO::FileStream::Position

See Also

Reference

Stream I/O
fopen, _wfopen
fgetpos
fseek, _fseeki64
_lseek, _lseeki64

_ftell_nolock, _ftelli64_nolock

Gets the current position of a file pointer, without locking the thread.

```
long _ftell_nolock(
   FILE *stream
);
__int64 _ftelli64_nolock(
   FILE *stream
);
```

Parameters

stream

Target the **FILE** structure.

Return Value

Same as **ftell** and **_ftelli64**. For more information, see ftell, _ftelli64**.**

Remarks

These functions are non-locking versions of **ftell** and **_ftelli64**, respectively. They are identical to **ftell** and **_ftelli64** except that they are not protected from interference by other threads. These functions might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Requirements

Function	Required header	Optional header	Compatibility
ftell_nolock	<stdio.h>	<errno.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ftelli64_nolock	<stdio.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::IO::FileStream::Position

See Also

Reference

Stream I/O
fgetpos
fseek, _fseeki64
_lseek, _lseeki64
ftell, _ftelli64

_ftime, _ftime32, _ftime64

Get the current time. These functions are deprecated because more secure versions are available; see _ftime_s, _ftime32_s, _ftime64_s.

```
void _ftime(
   struct _timeb *timeptr
);
void _ftime32(
   struct __timeb32 *timeptr
);
void _ftime64(
   struct __timeb64 *timeptr
);
```

Parameters

timeptr

Pointer to a **_timeb**, **__timeb32** or **__timeb64** structure.

Remarks

The **_ftime** function gets the current local time and stores it in the structure pointed to by *timeptr*. The **_timeb**, **__timeb32**, and **__timeb64** structures are defined in SYS\Timeb.h. They contain four fields, which are listed in the following table.

dstflag

Nonzero if daylight savings time is currently in effect for the local time zone. (See _tzset for an explanation of how daylight savings time is determined.)

millitm

Fraction of a second in milliseconds.

time

Time in seconds since midnight (00:00:00), January 1, 1970, coordinated universal time (UTC).

timezone

Difference in minutes, moving westward, between UTC and local time. The value of *timezone* is set from the value of the global variable **_timezone** (see **_tzset**).

_ftime64, which uses the **__timeb64** structure, allows file-creation dates to be expressed up through 23:59:59, December 31, 3000, UTC; whereas **_ftime32** only represents dates through 03:14:07 January 19, 2038, UTC. Midnight, January 1, 1970, is the lower bound of the date range for all these functions.

In Visual C++ 2005, **_ftime** is equivalent to **_ftime64** and **_timeb** contains a 64-bit time. This is true unless **_USE_32BIT_TIME_T** is defined, in which case the old behavior is in effect; **_ftime** uses a 32-bit time and **_timeb** contains a 32-bit time.

_ftime validates its parameters. If passed a null pointer as *timeptr*, the function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the function sets **errno** to **EINVAL**.

Requirements

Function	Required header	Compatibility
_ftime	<sys/types.h> and <sys/timeb.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ftime32	<sys/types.h> and <sys/timeb.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_ftime64	<sys/types.h> and <sys/timeb.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_ftime.c
// compile with: /W3
// This program uses _ftime to obtain the current
// time and then stores this time in timebuffer.

#include <stdio.h>
#include <sys/timeb.h>
#include <time.h>

int main( void )
{
   struct _timeb timebuffer;
   char timeline[26];
   errno_t err;
   time_t time1;
   unsigned short millitm1;
   short timezone1;
   short dstflag1;

   _ftime( &timebuffer ); // C4996
   // Note: _ftime is deprecated; consider using _ftime_s instead

   time1 = timebuffer.time;
   millitm1 = timebuffer.millitm;
   timezone1 = timebuffer.timezone;
   dstflag1 = timebuffer.dstflag;

   printf( "Seconds since midnight, January 1, 1970 (UTC): %I64d\n",
   time1);
   printf( "Milliseconds: %d\n", millitm1);
   printf( "Minutes between UTC and local time: %d\n", timezone1);
   printf( "Daylight savings time flag (1 means Daylight time is in "
           "effect): %d\n", dstflag1);

   err = ctime_s( timeline, 26, & ( timebuffer.time ) );
   if (err)
   {
       printf("Invalid argument to ctime_s. ");
   }
   printf( "The time is %.19s.%hu %s", timeline, timebuffer.millitm,
           &timeline[20] );
}
```

Sample Output

```
Seconds since midnight, January 1, 1970 (UTC): 1051553334
Milliseconds: 230
Minutes between UTC and local time: 480
Daylight savings time flag (1 means Daylight time is in effect): 1
The time is Mon Apr 28 11:08:54.230 2003
```

.NET Framework Equivalent

System::DateTime::Now

See Also

Reference

Time Management

asctime, _wasctime

ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
time, _time32, _time64

_ftime_s, _ftime32_s, _ftime64_s

Gets the current time. These are versions of _ftime, _ftime32, _ftime64 with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _ftime_s(
   struct _timeb *timeptr
);
errno_t _ftime32_s(
   struct __timeb32 *timeptr
);
errno_t _ftime64_s(
   struct __timeb64 *timeptr
);
```

Parameters

timeptr

Pointer to a **_timeb, __timeb32**, or **__timeb64** structure.

Return Value

Zero if successful, an error code on failure. If *timeptr* is **NULL**, the return value is **EINVAL**.

Remarks

The **_ftime_s** function gets the current local time and stores it in the structure pointed to by *timeptr*. The **_timeb, __timeb32**, and **__timeb64** structures are defined in SYS\Timeb.h. They contain four fields, which are listed in the following table.

dstflag

Nonzero if daylight savings time is currently in effect for the local time zone. (See _tzset for an explanation of how daylight savings time is determined.)

millitm

Fraction of a second in milliseconds.

time

Time in seconds since midnight (00:00:00), January 1, 1970, coordinated universal time (UTC).

timezone

Difference in minutes, moving westward, between UTC and local time. The value of *timezone* is set from the value of the global variable **_timezone** (see **_tzset**).

_ftime64_s, which uses the **__timeb64** structure, allows file-creation dates to be expressed up through 23:59:59, December 31, 3000, UTC; whereas **_ftime32_s** only represents dates through 03:14:07 January 19, 2038, UTC. Midnight, January 1, 1970, is the lower bound of the date range for all these functions.

In Visual C++ 2005, **_ftime_s** is equivalent to **_ftime64_s** and **_timeb** contains a 64-bit time. This is true unless **_USE_32BIT_TIME_T** is defined, in which case the old behavior is in effect; **_ftime_s** uses a 32-bit time and **_timeb** contains a 32-bit time.

_ftime_s validates its parameters. If passed a null pointer as *timeptr*, the function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the function sets **errno** to **EINVAL**.

Requirements

Function	Required header	Compatibility
_ftime_s	<sys/types.h> and <sys/timeb.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_ftime32_s	<sys/types.h> and <sys/timeb.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ftime64_s	<sys/types.h> and <sys/timeb.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_ftime64_s.c
// This program uses _ftime64_s to obtain the current
// time and then stores this time in timebuffer.

#include <stdio.h>
#include <sys/timeb.h>
#include <time.h>

int main( void )
{
   struct _timeb timebuffer;
   char timeline[26];
   errno_t err;
   time_t time1;
   unsigned short millitm1;
   short timezone1;
   short dstflag1;

   _ftime64_s( &timebuffer );

    time1 = timebuffer.time;
    millitm1 = timebuffer.millitm;
    timezone1 = timebuffer.timezone;
    dstflag1 = timebuffer.dstflag;

   printf( "Seconds since midnight, January 1, 1970 (UTC): %I64d\n",
   time1);
   printf( "Milliseconds: %d\n", millitm1);
   printf( "Minutes between UTC and local time: %d\n", timezone1);
   printf( "Daylight savings time flag (1 means Daylight time is in "
           "effect): %d\n", dstflag1);

   err = ctime_s( timeline, 26, & ( timebuffer.time ) );
   if (err)
   {
       printf("Invalid argument to ctime_s. ");
   }
   printf( "The time is %.19s.%hu %s", timeline, timebuffer.millitm,
           &timeline[20] );
}
```

Sample Output

```
Seconds since midnight, January 1, 1970 (UTC): 1051553334
Milliseconds: 230
Minutes between UTC and local time: 480
Daylight savings time flag (1 means Daylight time is in effect): 1
The time is Mon Apr 28 11:08:54.230 2003
```

.NET Framework Equivalent

System::DateTime::Now

See Also

Reference

Time Management
asctime, _wasctime
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
time, _time32, _time64

_fullpath, _wfullpath

Creates an absolute or full path name for the specified relative path name.

```
char *_fullpath(
   char *absPath,
   const char *relPath,
   size_t maxLength
);
wchar_t *_wfullpath(
   wchar_t *absPath,
   const wchar_t *relPath,
   size_t maxLength
);
```

Parameters

absPath

Pointer to a buffer containing the absolute or full path name, or NULL.

relPath

Relative path name.

maxLength

Maximum length of the absolute path name buffer (*absPath*). This length is in bytes for **_fullpath** but in wide characters (**wchar_t**) for **_wfullpath**.

Return Value

Each of these functions returns a pointer to a buffer containing the absolute path name (*absPath*). If there is an error (for example, if the value passed in *relPath* includes a drive letter that is not valid or cannot be found, or if the length of the created absolute path name (*absPath*) is greater than *maxLength*), the function returns **NULL**.

Remarks

The **_fullpath** function expands the relative path name in *relPath* to its fully qualified or absolute path and stores this name in *absPath*. If *absPath* is NULL, **malloc** is used to allocate a buffer of sufficient length to hold the path name. It is the responsibility of the caller to free this buffer. A relative path name specifies a path to another location from the current location (such as the current working directory: "."). An absolute path name is the expansion of a relative path name that states the entire path required to reach the desired location from the root of the file system. Unlike **_makepath**, **_fullpath** can be used to obtain the absolute path name for relative paths (*relPath*) that include "./" or "../" in their names.

For example, to use C run-time routines, the application must include the header files that contain the declarations for the routines. Each header file include statement references the location of the file in a relative manner (from the application's working directory):

```
#include <stdlib.h>
```

when the absolute path (actual file system location) of the file might be:

```
\\machine\shareName\msvcSrc\crt\headerFiles\stdlib.h
```

_fullpath automatically handles multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the multibyte code page currently in use. **_wfullpath** is a wide-character version of **_fullpath**; the string arguments to **_wfullpath** are wide-character strings. **_wfullpath** and **_fullpath** behave identically except that **_wfullpath** does not handle multibyte-character strings.

If **_DEBUG** and **_CRTDBG_MAP_ALLOC** are both defined, calls to **_fullpath** and **_wfullpath** are replaced by calls to **_fullpath_dbg** and **_wfullpath_dbg** to allow for debugging memory allocations. For more information, see _fullpath_dbg, _wfullpath_dbg.

This function invokes the invalid parameter handler, as described in Parameter Validation, if *maxlen* is less than or equal to 0. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **NULL**.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tfullpath	**_fullpath**	**_fullpath**	**_wfullpath**

If the *absPath* buffer is **NULL**, **_fullpath** calls malloc to allocate a buffer and ignores the *maxLength* argument. It is the caller's responsibility to deallocate this buffer (using free) as appropriate. If the *relPath* argument specifies a disk drive, the current directory of this drive is combined with the path.

Requirements

Function	Required header	Compatibility
_fullpath	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfullpath	<stdlib.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_fullpath.c
// This program demonstrates how _fullpath
// creates a full path from a partial path.

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <direct.h>

void PrintFullPath( char * partialPath )
{
   char full[_MAX_PATH];
   if( _fullpath( full, partialPath, _MAX_PATH ) != NULL )
      printf( "Full path is: %s\n", full );
   else
      printf( "Invalid path\n" );
}

int main( void )
{
   PrintFullPath( "test" );
   PrintFullPath( "\\test" );
   PrintFullPath( "..\\test" );
}
```

Sample Output

```
Full path is: C:\Documents and Settings\user\My Documents\test
Full path is: C:\test
Full path is: C:\Documents and Settings\user\test
```

.NET Framework Equivalent

System::IO::File::Create

See Also

Reference

File Handling
_getcwd, _wgetcwd
_getdcwd, _wgetdcwd
_makepath, _wmakepath

_splitpath, _wsplitpath

_fullpath_dbg, _wfullpath_dbg

Versions of _fullpath, _wfullpath that use the debug version of **malloc** to allocate memory.

```
char *_fullpath_dbg(
   char *absPath,
   const char *relPath,
   size_t maxLength,
   int blockType,
   const char *filename,
   int linenumber
);
wchar_t *_wfullpath_dbg(
   wchar_t *absPath,
   const wchar_t *relPath,
   size_t maxLength,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

absPath

Pointer to a buffer containing the absolute or full path name, or **NULL**.

relPath

Relative path name.

maxLength

Maximum length of the absolute path name buffer (*absPath*). This length is in bytes for **_fullpath** but in wide characters (**wchar_t**) for **_wfullpath**.

blockType

Requested type of memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

filename

Pointer to the name of the source file that requested allocation operation or **NULL**.

linenumber

Line number in the source file where the allocation operation was requested or **NULL**.

Return Value

Each function returns a pointer to a buffer containing the absolute path name (*absPath*). If there is an error (for example, if the value passed in *relPath* includes a drive letter that is not valid or cannot be found, or if the length of the created absolute path name (*absPath*) is greater than *maxLength*) the function returns **NULL**.

Remarks

The **_fullpath_dbg** and **_wfullpath_dbg** functions are identical to **_fullpath** and **_wfullpath** except that, when **_DEBUG** is defined, these functions use the debug version of **malloc**, **_malloc_dbg**, to allocate memory if NULL is passed as the first parameter. For information on the debugging features of **_malloc_dbg**, see _malloc_dbg.

You do not need to call these functions explicitly in most cases. Instead, you can define the **_CRTDBG_MAP_ALLOC** flag. When **_CRTDBG_MAP_ALLOC** is defined, calls to **_fullpath** and **_wfullpath** are remapped to **_fullpath_dbg** and **_wfullpath_dbg**, respectively, with the *blockType* set to **_NORMAL_BLOCK**. Thus, you do not need to call these functions explicitly unless you want to mark the heap blocks as **_CLIENT_BLOCK**. For more information, see Types of Blocks on the Debug Heap.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined

_tfullpath_dbg	_fullpath_dbg	_fullpath_dbg	_wfullpath_dbg

Requirements

Function	Required header	Compatibility
_fullpath_dbg	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wfullpath_dbg	<crtdbg.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Create

See Also

Reference

File Handling

_fullpath, _wfullpath

Concepts

Debug Versions of Heap Allocation Functions

_futime, _futime32, _futime64

Sets the modification time on an open file.

```
int _futime(
   int fd,
   struct _utimbuf *filetime
);
int _futime32(
   int fd,
   struct __utimbuf32 *filetime
);
int _futime64(
   int fd,
   struct __utimbuf64 *filetime
);
```

Parameters

fd

File descriptor to the open file.

filetime

Pointer to the structure containing the new modification date.

Return Value

Return 0 if successful. If an error occurs, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns –1 and **errno** is set to **EBADF**, indicating an invalid file descriptor, or **EINVAL**, indicating an invalid parameter.

Remarks

The **_futime** routine sets the modification date and the access time on the open file associated with *fd*. **_futime** is identical to _utime, except that its argument is the file descriptor of an open file, rather than the name of a file or a path to a file. The **_utimbuf** structure contains fields for the new modification date and access time. Both fields must contain valid values. **_utimbuf32** and **_utimbuf64** are identical to **_utimbuf** except for the use of the 32-bit and 64-bit time types, respectively. In Visual C++ 2005, **_futime** and **_utimbuf** use a 64-bit time type and **_futime** is identical in behavior to **_futime64**. If you need to force the old behavior, define **_USE_32BIT_TIME_T**. Doing this causes **_futime** to be identical in behavior to **_futime32** and causes the **_utimbuf** structure to use the 32-bit time type, making it equivalent to **__utimbuf32**.

_futime64, which uses the **__utimbuf64** structure, can read and modify file dates through 23:59:59, December 31, 3000, UTC; whereas a call to **_futime32** fails if the date on the file is later than 19:14:07 January 18, 2038, UTC. Midnight, January 1, 1970, is the lower bound of the date range for these functions.

Requirements

Function	Required header	Optional header	Compatibility
_futime	<sys/utime.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_futime32	<sys/utime.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_futime64	<sys/utime.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_futime.c
// This program uses _futime to set the
// file-modification time to the current time.

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <io.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/utime.h>
#include <share.h>

int main( void )
{
   int hFile;

   // Show file time before and after.
   system( "dir crt_futime.c_input" );

   _sopen_s( &hFile, "crt_futime.c_input", _O_RDWR, _SH_DENYNO, 0 );

   if( _futime( hFile, NULL ) == -1 )
      perror( "_futime failed\n" );
   else
      printf( "File time modified\n" );

   _close (hFile);

   system( "dir crt_futime.c_input" );
}
```

Input: crt_futime.c_input

```
Arbitrary file contents.
```

Sample Output

```
Volume in drive Z has no label.
 Volume Serial Number is 5C68-57C1

 Directory of Z:\crt

03/25/2004  10:40 AM                24 crt_futime.c_input
               1 File(s)             24 bytes
               0 Dir(s)  24,268,476,416 bytes free
 Volume in drive Z has no label.
 Volume Serial Number is 5C68-57C1

 Directory of Z:\crt

03/25/2004  10:41 AM                24 crt_futime.c_input
               1 File(s)             24 bytes
               0 Dir(s)  24,268,476,416 bytes free
File time modified
```

.NET Framework Equivalent

- System::IO::File::SetLastAccessTime
- System::IO::File::SetLastWriteTime
- System::IO::File::SetCreationTime

See Also

Reference

fwide

Unimplemented.

```
int fwide(
   FILE *stream,
   int mode;
);
```

Parameters

stream

Pointer to **FILE** structure (ignored).

mode

The new width of the stream: positive for wide character, negative for byte, zero to leave unchanged. (This value is ignored.)

Return Value

This function currently just returns *mode*.

Remarks

The current version of this function does not comply with the Standard.

Requirements

Function	Required header	Compatibility
fwide	<wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more information, see Compatibility.

fwrite

Writes data to a stream.

```
size_t fwrite(
   const void *buffer,
   size_t size,
   size_t count,
   FILE *stream
);
```

Parameters

buffer

Pointer to data to be written.

size

Item size in bytes.

count

Maximum number of items to be written.

stream

Pointer to **FILE** structure.

Return Value

fwrite returns the number of full items actually written, which may be less than *count* if an error occurs. Also, if an error occurs, the file-position indicator cannot be determined. If either *stream* or *buffer* is a null pointer, the function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns 0.

Remarks

The **fwrite** function writes up to *count* items, of *size* length each, from *buffer* to the output *stream*. The file pointer associated with *stream* (if there is one) is incremented by the number of bytes actually written. If *stream* is opened in text mode, each carriage return is replaced with a carriage-return – linefeed pair. The replacement has no effect on the return value.

This function locks the calling thread and is therefore thread-safe. For a non-locking version, see **_fwrite_nolock**.

Requirements

Function	Required header	Compatibility
fwrite	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for fread.

.NET Framework Equivalent

System::IO::FileStream::Write

See Also

Reference

Stream I/O
fread
_write

_fwrite_nolock

Writes data to a stream, without locking the thread.

```
size_t _fwrite_nolock(
   const void *buffer,
   size_t size,
   size_t count,
   FILE *stream
);
```

Parameters

buffer

Pointer to the data to be written.

size

Item size in bytes.

count

Maximum number of items to be written.

stream

Pointer to the **FILE** structure.

Return Value

Same as fwrite.

Remarks

This function is a non-locking version of **fwrite**. It is identical to **fwrite** except that it is not protected from interference by other threads. It might be faster because it does not incur the overhead of locking out other threads. Use this function only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Requirements

Function	Required header	Compatibility
_fwrite_nolock	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example for fread.

.NET Framework Equivalent

System::IO::FileStream::Write

See Also

Reference

Stream I/O
fread
_write

gcvt

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _gcvt or security-enhanced _gcvt_s instead.

_gcvt

Converts a floating-point value to a string, which it stores in a buffer. This function is deprecated because a more secure version is available; see _gcvt_s.

```
char *_gcvt(
   double value,
   int digits,
   char *buffer
);
```

Parameters

value

Value to be converted.

digits

Number of significant digits stored.

buffer

Storage location for the result.

Return Value

_gcvt returns a pointer to the string of digits.

Remarks

The **_gcvt** function converts a floating-point *value* to a character string (which includes a decimal point and a possible sign byte) and stores the string in *buffer*. The *buffer* should be large enough to accommodate the converted value plus a terminating null character, which is appended automatically. If a buffer size of *digits* + 1 is used, the function overwrites the end of the buffer. This is because the converted string includes a decimal point and can contain sign and exponent information. There is no provision for overflow. **_gcvt** attempts to produce *digits* digits in decimal format. If it cannot, it produces *digits* digits in exponential format. Trailing zeros might be suppressed in the conversion.

A *buffer* of length **_CVTBUFSIZE** is sufficient for any floating point value.

This function validates its parameters. If *buffer* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **NULL**.

Requirements

Routine	Required header	Compatibility
_gcvt	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_gcvt.c
// compile with: /W3
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main( void )
{
   char buffer[_CVTBUFSIZE];
   double value = -1234567890.123;
   printf( "The following numbers were converted by _gcvt(value,12,buffer):\n" );
   _gcvt( value, 12, buffer ); // C4996
   // Note: _gcvt is deprecated; consider using _gcvt_s instead
```

```
    printf( "buffer: '%s' (%d chars)\n", buffer, strlen(buffer) );
    value *= 10;
    _gcvt( value, 12, buffer ); // C4996
    printf( "buffer: '%s' (%d chars)\n", buffer, strlen(buffer) );
    value *= 10;
    _gcvt( value, 12, buffer ); // C4996
    printf( "buffer: '%s' (%d chars)\n", buffer, strlen(buffer) );
    value *= 10;
    _gcvt( value, 12, buffer ); // C4996
    printf( "buffer: '%s' (%d chars)\n", buffer, strlen(buffer) );

    printf( "\n" );
    value = -12.34567890123;
    _gcvt( value, 12, buffer ); // C4996
    printf( "buffer: '%s' (%d chars)\n", buffer, strlen(buffer) );
    value /= 10;
    _gcvt( value, 12, buffer ); // C4996
    printf( "buffer: '%s' (%d chars)\n", buffer, strlen(buffer) );
    value /= 10;
    _gcvt( value, 12, buffer ); // C4996
    printf( "buffer: '%s' (%d chars)\n", buffer, strlen(buffer) );
    value /= 10;
    _gcvt( value, 12, buffer ); // C4996
    printf( "buffer: '%s' (%d chars)\n", buffer, strlen(buffer) );
}
```

Output

```
The following numbers were converted by _gcvt(value,12,buffer):
buffer: '-1234567890.12' (14 chars)
buffer: '-12345678901.2' (14 chars)
buffer: '-123456789012' (13 chars)
buffer: '-1.23456789012e+012' (19 chars)

buffer: '-12.3456789012' (14 chars)
buffer: '-1.23456789012' (14 chars)
buffer: '-0.123456789012' (15 chars)
buffer: '-1.23456789012e-002' (19 chars)
```

.NET Framework Equivalent

System::Convert::ToString

See Also

Reference

Data Conversion
Floating-Point Support
atof, _atof_l, _wtof, _wtof_l
_ecvt
_fcvt

_gcvt_s

Converts a floating-point value to a string. This is a version of _gcvt with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _gcvt_s(
   char *buffer,
   size_t sizeInBytes,
   double value,
   int digits
);
template <size_t cchStr>
errno_t _gcvt_s(
   char (&buffer)[cchStr],
   double value,
   int digits
); // C++ only
```

Parameters

[out] *buffer*

Buffer to store the result of the conversion.

[in] *sizeInBytes*

Size of the buffer.

[in] *value*

Value to be converted.

[in] *digits*

Number of significant digits stored.

Return Value

Zero if successful. If a failure occurs due to an invalid parameter (see the following table for invalid values), the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, an error code is returned. Error codes are defined in Errno.h. For a listing of these errors, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Error Conditions

buffer	***sizeInBytes***	***value***	***digits***	**Return**	**Value in *buffer***
NULL	any	any	any	**EINVAL**	Not modified.
Not **NULL** (points to valid memory)	zero	any	any	**EINVAL**	Not modified.
Not **NULL** (points to valid memory)	any	any	>= *sizeInBytes*	**EINVAL**	Not modified.

Security Issues

_gcvt_s can generate an access violation if *buffer* does not point to valid memory and is not **NULL**.

Remarks

The **_gcvt_s** function converts a floating-point *value* to a character string (which includes a decimal point and a possible sign byte) and stores the string in *buffer*. *buffer* should be large enough to accommodate the converted value plus a terminating null character, which is appended automatically. A buffer of length **_CVTBUFSIZE** is sufficient for any floating point value. If a buffer size of *digits* + 1 is used, the function will not overwrite the end of the buffer, so be sure to supply a sufficient buffer for this operation. **_gcvt_s** attempts to produce *digits* digits in decimal format. If it cannot, it produces *digits* digits in exponential format. Trailing zeros can be suppressed in the conversion.

In C++, using this function is simplified by a template overload; the overload can infer buffer length automatically, eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

The debug version of this function first fills the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Requirements

Routine	Required header	Optional header	Compatibility
_gcvt_s	<stdlib.h>	<error.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_gcvt_s.c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main()
{
  char buf[_CVTBUFSIZE];
  int decimal;
  int sign;
  int err;

  err = _gcvt_s(buf, _CVTBUFSIZE, 1.2, 5);

  if (err != 0)
  {
     printf("_gcvt_s failed with error code %d\n", err);
     exit(1);
  }

  printf("Converted value: %s\n", buf);

}
```

Output

```
Converted value: 1.2
```

.NET Framework Equivalent

ToString

See Also

Reference

Data Conversion
Floating-Point Support
atof, _atof_l, _wtof, _wtof_l
_ecvt_s
_fcvt_s
_gcvt

_get_amblksiz

Gets the current block size used in memory allocations.

```
errno_t _get_amblksiz(
   size_t * blockSize
);
```

Parameters

[out] *blockSize*

The block size will be written to this variable.

Return Value

Zero if successful; an error code on failure.

Remarks

The _amblksiz global variable controls the block size used by the memory allocation functions such as malloc. This function provides a secure way of getting access to the value of this variable.

If *blockSize* is **NULL**, this function invokes an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns -1 and sets **errno** to **EINVAL**.

Requirements

Routine	Required header	Compatibility
_get_amblksiz	<malloc.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

_amblksiz

_set_amblksiz

_get_current_locale

Gets a locale object representing the current locale.

```
_locale_t _get_current_locale(void);
```

Return Value

A locale object representing the current locale.

Remarks

The **_get_current_locale** function gets the currently set locale for the thread and returns a locale object representing that locale.

The previous name of this function, **__get_current_locale** (with two leading underscores) has been deprecated.

Requirements

Routine	Required header	Compatibility
_get_current_locale	<locale.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

No equivalent.

See Also

Reference

setlocale, _wsetlocale
_create_locale
_free_locale

_get_daylight

Retrieves the daylight saving time offset in hours.

```
error_t _get_daylight(
    int* hours
);
```

Parameters

hours

The offset in hours of daylight saving time.

Return Value

Zero if successful or an **errno** value if an error occurs.

Remarks

The **_get_daylight** function retrieves the number of hours in daylight saving time as an integer. If daylight saving time is in effect, the default offset is one hour (although a few regions do observe a two-hour offset).

If *hours* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

Requirements

Routine	Required header	Compatibility
_get_daylight	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more information, see Compatibility.

See Also

Reference

Time Management
errno, _doserrno, _sys_errlist, and _sys_nerr
_get_dstbias
_get_timezone
_get_tzname

_get_doserrno

_doserrno is deprecated. Use **_get_errno** instead.

```
errno_t _get_doserrno(
   int * pValue
);
```

Parameters

[out] *pValue*

A pointer to an integer to be filled with the current value of the **_doserrno** variable.

Return Value

Returns zero if successful; an error code on failure. If *pValue* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Remarks

Possible values of **_doserrno** are defined in Errno.h.

Requirements

Routine	Required header	Optional header	Compatibility
_get_doserrno	<stdlib.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

See Also

Reference

_set_doserrno

errno, _doserrno, _sys_errlist, and _sys_nerr

_get_dstbias

Retrieves the daylight saving time offset in seconds.

```
error_t _get_dstbias(
    int* seconds
);
```

Parameters

seconds

The offset in seconds of daylight saving time.

Return Value

Zero if successful or an **errno** value if an error occurs.

Remarks

The **_get_dstbias** function retrieves the number of seconds in daylight saving time as an integer. If daylight saving time is in effect, the default offset is 3600 seconds, which is the number of seconds in one hour (though a few regions do observe a two-hour offset).

If *seconds* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

Requirements

Routine	Required header	Compatibility
_get_dayseconds	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more information, see Compatibility.

See Also

Reference

Time Management
errno, _doserrno, _sys_errlist, and _sys_nerr
_get_daylight
_get_timezone
_get_tzname

_get_errno

Gets the current value of the errno global variable.

```
errno_t _get_errno(
   int * pValue
);
```

Parameters

[out] *pValue*

A pointer to an integer to be filled with the current value of the **errno** variable.

Return Value

Returns zero if successful; an error code on failure. If *pValue* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Remarks

Possible values of **errno** are defined in Errno.h. Also, see errno Constants.

Example

```
// crt_get_errno.c
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <share.h>
#include <errno.h>

int main()
{
   errno_t err;
   int pfh;
   _sopen_s( &pfh, "nonexistent.file", _O_WRONLY, _SH_DENYNO, _S_IWRITE );
   _get_errno( &err );
   printf( "errno = %d\n", err );
   printf( "fyi, ENOENT = %d\n", ENOENT );
}
```

Output

```
errno = 2
fyi, ENOENT = 2
```

Requirements

Routine	Required header	Optional header	Compatibility
_get_errno	<stdlib.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

_set_errno

errno, _doserrno, _sys_errlist, and _sys_nerr

_get_fileinfo

Gets the value of the **_fileinfo** variable, which determines what information about open files is passed to spawned processes.

Starting with Visual C++ 2005, **_get_fileinfo** is no longer supported in any Microsoft developer tools or libraries.

```
errno_t _get_fileinfo(
   int * pFileInfo
);
```

Parameters

[out] *pFileInfo*

A pointer to an integer to be filled with the current value of **_fileinfo**.

Return Value

Returns zero if successful; an error code on failure. If *pFileInfo* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function returns **EINVAL** and sets **errno** to **EINVAL**.

Remarks

The function gets the value of the _fileinfo global variable. This variable determines what information is passed to processes created with the spawn functions. For more information, see _spawn, _wspawn Functions.

Requirements

Routine	Required header	Compatibility
_get_fileinfo	None	None

See Also

Reference

_fmode
_set_fmode
_setmode
Text and Binary Mode File I/O

_get_fmode

Gets the default file translation mode for file I/O operations.

```
errno_t _get_fmode(
   int * pmode
);
```

Parameters

[out] *pmode*

A pointer to an integer to be filled with the current default mode: **_O_TEXT** or **_O_BINARY**.

Return Value

Returns zero if successful; an error code on failure. If *pmode* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **EINVAL**.

Remarks

The function gets the value of the _fmode global variable. This variable specifies the default file translation mode for both low-level and stream file I/O operations, such as **_open**, **_pipe**, **fopen**, and **freopen**.

Requirements

Routine	Required header	Optional header	Compatibility
_get_fmode	<stdlib.h>	<fcntl.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example in _set_fmode.

NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

_fmode
_set_fmode
_setmode
Text and Binary Mode File I/O

_get_heap_handle

Returns the handle of the heap used by the C run-time system.

```
intptr_t _get_heap_handle( void );
```

Return Value

Returns the handle to the Win32 heap used by the C run-time system.

Remarks

Use this function if you want to call HeapSetInformation and enable the Low Fragmentation Heap on the CRT heap.

Requirements

Routine	Required header	Compatibility
_get_heap_handle	<malloc.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Sample

```
// crt_get_heap_handle.cpp
// compile with: /MT
#include <windows.h>
#include <malloc.h>
#include <stdio.h>

int main(void)
{
    intptr_t hCrtHeap = _get_heap_handle();
    ULONG ulEnableLFH = 2;
    if (HeapSetInformation((PVOID)hCrtHeap,
                           HeapCompatibilityInformation,
                           &ulEnableLFH, sizeof(ulEnableLFH)))
        puts("Enabling Low Fragmentation Heap succeeded");
    else
        puts("Enabling Low Fragmentation Heap failed");
    return 0;
}
```

See Also

Reference

Memory Allocation

_get_osfhandle

Returns operating-system file handle associated with existing low-level file descriptor.

```
long _get_osfhandle(
   int fd
);
```

Parameters

fd

User file descriptor.

Return Value

The **_get_osfhandle** function returns an operating-system file handle if *fd* (the low-level file descriptor) is in range and is internally marked as free. Otherwise, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function returns **INVALID_HANDLE_VALUE** (–1) and sets **errno** to **EBADF**, indicating an invalid file handle.

Remarks

To close a file opened with **_get_osfhandle**, call **_close**. The underlying handle is also closed by a call to **_close**, so it is not necessary to call the Win32 function **CloseHandle** on the original handle.

Requirements

Routine	Required header	Compatibility
_get_osfhandle	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

File Handling
_close
_creat, _wcreat
_dup, _dup2
_open, _wopen

_get_osplatform

Gets the current value of the **_osplatform** global variable. This function is obsolete; consider using GetVersionEx which provides more precise and correct information.

```
errno_t _get_osplatform(
   int * pValue
);
```

Parameters

[out] *pValue*

A pointer to an integer to be filled with the current value of the **_osplatform** variable.

Return Value

Returns zero if successful; an error code on failure. If *pValue* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Remarks

This function gets the value of the **_osplatform** global variable.

Requirements

Routine	Required header	Compatibility
_get_osplatform	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

_osplatform, _osver, _winmajor, _winminor, _winver
_get_osver
_get_winmajor
_get_winminor
_get_winver

_get_osver

Gets the current value of the **_osver** global variable. This function is obsolete; consider using GetVersionEx which provides more precise and correct information.

```
errno_t _get_osver(
   int * pValue
);
```

Parameters

[out] *pValue*

A pointer to an integer to be filled with the current operating system build number.

Return Value

Returns zero if successful; an error code on failure. If *pValue* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Remarks

The **_osver** global variable contains the build number of the operating system.

Requirements

Routine	Required header	Compatibility
_get_osver	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

_osplatform, _osver, _winmajor, _winminor, _winver
_get_osplatform
_get_winmajor
_get_winminor
_get_winver

_get_output_format

Gets the current value of the output format flag.

```
unsigned int _get_output_format();
```

Return Value

The current value of the output format flag.

Remarks

The output format flag controls features of formatted I/O. At present the flag has two possible values: 0 and **_TWO_DIGIT_EXPONENT**. If **_TWO_DIGIT_EXPONENT** is set, the floating point numbers is printed with only two digits in the exponent unless a third digit is required by the size of the exponent. If the flag is zero, the floating point output displays three digits of exponent, using zeroes if necessary to pad the value to three digits.

Requirements

Routine	Required header	Compatibility
_get_output_format	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

printf, _printf_l, wprintf, _wprintf_l
printf Type Field Characters
_set_output_format

_get_pgmptr

Gets the current value of the **_pgmptr** global variable.

```
errno_t _get_pgmptr(
   char **pValue
);
```

Parameters

[out] *pValue*

A pointer to a string to be filled with the current value of the **_pgmptr** variable.

Return Value

Returns zero if successful; an error code on failure. If *pValue* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Remarks

The **_pgmptr** global variable contains the full path to the executable associated with the process. For more information, see _pgmptr, _wpgmptr.

Requirements

Routine	Required header	Compatibility
_get_pgmptr	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

_get_wpgmptr

_get_printf_count_output

Indicates whether printf, _printf_l, wprintf, _wprintf_l-family functions support the **%n** format.

```
int _get_printf_count_output();
```

Return Value

Non-zero if **%n** is supported, 0 if **%n** is not supported.

Remarks

If **%n** is not supported (the default), encountering **%n** in the format string of any of the **printf** functions will invoke the invalid parameter handler as described in Parameter Validation. If **%n** support is enabled (see _set_printf_count_output) then **%n** will behave as described in printf Type Field Characters.

Requirements

Routine	Required header	Compatibility
_get_printf_count_output	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for _set_printf_count_output.

NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

_get_sbh_threshold

Returns the upper limit for the size of a memory allocation that is supported by the small-block heap.

```
size_t _get_sbh_threshold( void );
```

Return Value

Returns the upper limit for the size of a memory allocation that is supported by the small-block heap.

Remarks

_get_sbh_threshold gets the current threshold value for the small-block heap. The default threshold size is 1016 bytes for Windows 95, Windows 98, Windows Millennium Edition, and Windows NT 4.0 and zero for Windows 2000 and later platforms. By default, the small-block heap is not used on Windows 2000 and later platforms, though **_set_sbh_threshold** can be called with a nonzero value to enable the small-block heap in those instances.

Requirements

Routine	Required header	Compatibility
_get_sbh_threshold	<malloc.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

This program shows how to retrieve the upper limit for the size of a memory allocation that is supported by the small-block heap.

```
// crt__get_sbh_threshold.c
#include <stdio.h>
#include <malloc.h>

int main( void )
{
    size_t  sbh_threshold;

    sbh_threshold = _get_sbh_threshold();

    printf( "The upper limit for the size of a memory allocation is %d.\n", sbh_threshold);
}
```

Output

```
The upper limit for the size of a memory allocation is 0.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation

_set_sbh_threshold

_get_terminate

Returns the termination routine to be called by **terminate**.

```
terminate_function _get_terminate( void );
```

Return Value

Returns a pointer to the function registered by set_terminate (CRT). If no function has been set, the return value may be used to restore the default behavior; this value may be NULL.

Requirements

Routine	Required header	Compatibility
_get_terminate	<eh.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Exception Handling Routines
abort
set_unexpected (CRT)
terminate (CRT)
unexpected (CRT)

_get_timezone

Retrieves the difference in seconds between coordinated universal time (UTC) and local time.

```
error_t _get_timezone(
    int* seconds
);
```

Parameters

seconds

The difference in seconds between UTC and local time.

Return Value

Zero if successful or an **errno** value if an error occurs.

Remarks

The **_get_timezone** function retrieves the difference in seconds between UTC and local time as an integer. The default value is 28,800 seconds, for Pacific Standard Time (eight hours behind UTC).

If *seconds* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Requirements

Routine	Required header	Compatibility
_get_timezone	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more information, see Compatibility.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Time Management
errno, _doserrno, _sys_errlist, and _sys_nerr
_get_daylight
_get_dstbias
_get_tzname

_get_tzname

Retrieves the character string representation of the time zone name or the daylight standard time zone name (DST).

```
errno_t _get_tzname(
    size_t* pReturnValue,
    char* timeZoneName,
    size_t sizeInBytes,
    int index
);
```

Parameters

[out] *pReturnValue*

The string length of *timeZoneName* including a NULL terminator.

[out] *timeZoneName*

The address of a character string for the representation of the time zone name or the daylight standard time zone name (DST), depending on *index*.

[in] *sizeInBytes*

The size of the *timeZoneName* character string in bytes.

[in] *index*

The index of one of the two time zone names to retrieve.

Return Value

Zero if successful, otherwise an **errno** type value.

If either *timeZoneName* is **NULL**, or *sizeInBytes* is zero or less than zero (but not both), an invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Error Conditions

pReturnValue	***timeZoneName***	***sizeInBytes***	***index***	**Return value**	**Contents of *timeZoneName***
size of TZ name	**NULL**	0	0 or 1	0	not modified
size of TZ name	any	> 0	0 or 1	0	TZ name
not modified	**NULL**	> 0	any	**EINVAL**	not modified
not modified	any	zero	any	**EINVAL**	not modified
not modified	any	> 0	> 1	**EINVAL**	not modified

Remarks

The **_get_tzname** function retrieves the character string representation of the time zone name or the daylight standard time zone name (DST) into the address of *timeZoneName* depending on the index value, along with the size of the string in *pReturnValue*. If *timeZoneName* is **NULL** and *sizeInBytes* is zero, just the size of the string of either time zone in bytes is returned in *pReturnValue*. The index values must be either 0 for standard time zone or 1 for daylight standard time zone; any other values of index have undetermined results.

Index values

index	**Contents of *timeZoneName***	***timeZoneName* default value**
0	Time zone name	"PST"

1	Daylight standard time zone name	"PDT"
> 1 or < 0	**errno** set to **EINVAL**	not modified

Unless the values are explicitly changed during run time, the default values are "PST" and "PDT" respectively. The sizes of these character arrays are governed by **TZNAME_MAX** value.

Requirements

Routine	Required header	Compatibility
_get_tzname	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more information, see Compatibility.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Time Management
errno, _doserrno, _sys_errlist, and _sys_nerr
_get_daylight
_get_dstbias
_get_timezone
TZNAME_MAX

_get_unexpected

Returns the termination routine to be called by **unexpected**.

```
unexpected_function _get_unexpected( void );
```

Return Value

Returns a pointer to the function registered by set_unexpected (CRT). If no function has been set, the return value may be used to restore the default behavior; this value may be NULL.

Requirements

Routine	Required header	Compatibility
_get_unexpected	<eh.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Exception Handling Routines
abort
set_terminate (CRT)
terminate (CRT)
unexpected (CRT)

_get_winmajor

Gets the current value of the **_winmajor** global variable. This function is obsolete; consider using GetVersionEx which provides more precise and correct information.

```
errno_t _get_winmajor(
   int * pValue
);
```

Parameters

[out] *pValue*

A pointer to an integer to be filled with the current major version number of the Windows operating system.

Return Value

Returns zero if successful; an error code on failure. If *pValue* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Remarks

The **_winmajor** global variable contains the major version of the Windows operating system.

Requirements

Routine	Required header	Compatibility
_get_winmajor	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

_osplatform, _osver, _winmajor, _winminor, _winver
_get_osplatform
_get_osver
_get_winminor
_get_winver

_get_winminor

Gets the current value of the **_winminor** global variable. This function is obsolete; consider using GetVersionEx which provides more precise and correct information.

```
errno_t _get_winminor(
   int * pValue
);
```

Parameters

[out] *pValue*

A pointer to an integer to be filled with the current minor version number of the Windows operating system.

Return Value

Returns zero if successful; an error code on failure. If *pValue* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Remarks

The **_winminor** global variable contains the minor version of the Windows operating system.

Requirements

Routine	Required header	Compatibility
_get_winminor	<stdlpb.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

_osplatform, _osver, _winmajor, _winminor, _winver
_get_osplatform
_get_osver
_get_winmajor
_get_winver

_get_winver

Gets the current value of the **_winver** global variable. This function is obsolete; consider using GetVersionEx which provides more precise and correct information.

```
errno_t _get_winver(
   int * pValue
);
```

Parameters

[out] *pValue*

A pointer to an integer to be filled with the current Windows version.

Return Value

Returns zero if successful; an error code on failure. If *pValue* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Remarks

The **_winver** global variable contains the version of the operating system. The high byte is the major version number. The low byte is the minor version number.

Requirements

Routine	Required header	Compatibility
_get_winver	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

_osplatform, _osver, _winmajor, _winminor, _winver
_get_osplatform
_get_osver
_get_winmajor
_get_winminor

_get_wpgmptr

Gets the current value of the **_wpgmptr** global variable.

```
errno_t _get_wpgmptr(
   wchar_t **pValue
);
```

Parameters

[out] *pValue*

A pointer to a string to be filled with the current value of the **_wpgmptr** variable.

Return Value

Returns zero if successful; an error code on failure. If *pValue* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Remarks

The **_wpgmptr** global variable contains the full path to the executable associated with the process as a wide-character string. For more information, see _pgmptr, _wpgmptr.

Requirements

Routine	Required header	Compatibility
_get_wpgmptr	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

_get_pgmptr

getc, getwc

Read a character from a stream.

```
int getc(
   FILE *stream
);
wint_t getwc(
   FILE *stream
);
```

Parameters

stream

Input stream.

Return Value

Returns the character read. To indicate a read error or end-of-file condition, **getc** returns **EOF**, and **getwc** returns **WEOF**. For **getc**, use **ferror** or **feof** to check for an error or for end of file. If *stream* is **NULL**, **getc** and **getwc** invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return **EOF** (or **WEOF** for **getwc**) and set **errno** to **EINVAL**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

Each routine reads a single character from a file at the current position and increments the associated file pointer (if defined) to point to the next character. The file is associated with *stream*.

These functions lock the calling thread and are therefore thread-safe. For a non-locking version, see _getc_nolock, _getwc_nolock.

Routine-specific remarks follow.

Routine	Remarks
getc	Same as **fgetc**, but implemented as a function and as a macro.
getwc	Wide-character version of **getc**. Reads a multibyte character or a wide character according to whether *stream* is opened in text mode or binary mode.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_gettc	**getc**	**getc**	**getwc**

Requirements

Routine	Required header	Compatibility
getc	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
getwc	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_getc.c
// Use getc to read a line from a file.
```

```
#include <stdio.h>

int main()
{
    char buffer[81];
    int i, ch;
    FILE* fp;

    // Read a single line from the file "crt_getc.txt".

    fopen_s(&fp, "crt_getc.txt", "r");
    if (!fp)
    {
       printf("Failed to open file crt_getc.txt.\n");
       exit(1);
    }

    for (i = 0; (i < 80) && ((ch = getc(fp)) != EOF)
                         && (ch != '\n'); i++)
    {
        buffer[i] = (char) ch;
    }

    // Terminate string with a null character
    buffer[i] = '\0';
    printf( "Input was: %s\n", buffer);

    fclose(fp);
}
```

Input: crt_getc.txt

```
Line one.
Line two.
```

Output

```
Input was: Line one.
```

.NET Framework Equivalent

- System::IO::StreamReader::Read
- System::Console::Read

See Also

Reference

Stream I/O
fgetc, fgetwc
_getch, _getwch
putc, putwc
ungetc, ungetwc

_getc_nolock, _getwc_nolock

Reads a character from a stream.

```
int _getc_nolock(
   FILE *stream
);
wint_t _getwc_nolock(
   FILE *stream
);
```

Parameters

stream

Input stream.

Return Value

See getc, getwc.

Remarks

These functions are identical to **getc** and **getwc** except that they do not lock the calling thread. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_gettc_nolock	**getc_nolock**	**getc_nolock**	**getwc_nolock**

Requirements

Routine	Required header	Compatibility
getc_nolock	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
getwc_nolock	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_getc_nolock.c
// Use getc to read a line from a file.

#include <stdio.h>

int main()
{
    char buffer[81];
    int i, ch;
    FILE* fp;

    // Read a single line from the file "crt_getc_nolock.txt".
    fopen_s(&fp, "crt_getc_nolock.txt", "r");
    if (!fp)
    {
       printf("Failed to open file crt_getc_nolock.txt.\n");
       exit(1);
    }
```

```
    for (i = 0; (i < 80) && ((ch = getc(fp)) != EOF)
                         && (ch != '\n'); i++)
    {
        buffer[i] = (char) ch;
    }

    // Terminate string with a null character
    buffer[i] = '\0';
    printf( "Input was: %s\n", buffer);

    fclose(fp);
}
```

Input: crt_getc_nolock.txt

```
Line the first.
Line the second.
```

Output

```
Input was: Line the first.
```

.NET Framework Equivalent

- System::IO::StreamReader::Read
- System::Console::Read

See Also

Reference

Stream I/O
fgetc, fgetwc
_getch, _getwch
putc, putwc
ungetc, ungetwc

getch

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _getch instead.

_getch, _getwch

Gets a character from the console without echo.

```
int _getch( void );
wint_t _getwch( void );
```

Return Value

Returns the character read. There is no error return.

Remarks

The **_getch** and **_getwch** functions read a single character from the console without echoing the character. None of these functions can be used to read CTRL+C. When reading a function key or an arrow key, each function must be called twice; the first call returns 0 or 0xE0, and the second call returns the actual key code.

These functions lock the calling thread and are therefore thread-safe. For non-locking versions, see _getch_nolock, _getwch_nolock.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_gettch	**_getch**	**_getch**	**_getwch**

Requirements

Routine	Required header	Compatibility
_getch	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_getwch	<conio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_getch.c
// compile with: /c
// This program reads characters from
// the keyboard until it receives a 'Y' or 'y'.


#include <conio.h>
#include <ctype.h>

int main( void )
{
   int ch;

   _cputs( "Type 'Y' when finished typing keys: " );
   do
   {
      ch = _getch();
      ch = toupper( ch );
   } while( ch != 'Y' );

   _putch( ch );
   _putch( '\r' );    // Carriage return
   _putch( '\n' );    // Line feed
}
```

Input

```
abcdey
```

Output

```
Type 'Y' when finished typing keys: Y
```

NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Console and Port I/O
_getche, _getwche
_cgets, _cgetws
getc, getwc
_ungetch, _ungetwch, _ungetch_nolock, _ungetwch_nolock

_getch_nolock, _getwch_nolock

Gets a character from the console without echo and without locking the thread.

```
int _getch_nolock( void );
wint_t _getwch_nolock( void );
```

Return Value

Returns the character read. There is no error return.

Remarks

_getch_nolock and **_getwch_nolock** are identical to **_getch** and **_getchw** except that they not protected from interference by other threads. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_gettch_nolock	**_getch_nolock**	**_getch_nolock**	**_getwch_nolock**

Requirements

Routine	Required header	Compatibility
_getch_nolock	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_getwch_nolock	<conio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_getch_nolock.c
// compile with: /c
// This program reads characters from
// the keyboard until it receives a 'Y' or 'y'.


#include <conio.h>
#include <ctype.h>

int main( void )
{
   int ch;

   _cputs( "Type 'Y' when finished typing keys: " );
   do
   {
      ch = _getch_nolock();
      ch = toupper( ch );
   } while( ch != 'Y' );

   _putch_nolock( ch );
   _putch_nolock( '\r' );    // Carriage return
   _putch_nolock( '\n' );    // Line feed
}
```

Input

```
abcdey
```

Output

```
Type 'Y' when finished typing keys: Y
```

NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Console and Port I/O
_getche, _getwche
_cgets, _cgetws
getc, getwc
_ungetch, _ungetwch, _ungetch_nolock, _ungetwch_nolock

getchar, getwchar

Read a character from standard input.

```
int getchar();
wint_t getwchar();
```

Return Value

Returns the character read. To indicate a read error or end-of-file condition, **getchar returns EOF**, and **getwchar** returns **WEOF**. For **getchar**, use **ferror** or **feof** to check for an error or for end of file.

Remarks

Each routine reads a single character from **stdin** and increments the associated file pointer to point to the next character. **getchar** is the same as _fgetchar, but it is implemented as a function and as a macro.

These functions lock the calling thread and are therefore thread-safe. For a non-locking version, see _getchar_nolock, _getwchar_nolock.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_gettchar	**getchar**	**getchar**	**getwchar**

Requirements

Routine	Required header	Compatibility
getchar	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
getwchar	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_getchar.c
// Use getchar to read a line from stdin.

#include <stdio.h>

int main()
{
    char buffer[81];
    int i, ch;

    for (i = 0; (i < 80) && ((ch = getchar()) != EOF)
                         && (ch != '\n'); i++)
    {
        buffer[i] = (char) ch;
    }

    // Terminate string with a null character
    buffer[i] = '\0';
    printf( "Input was: %s\n", buffer);
}
```

Input

```
This text
```

Output

```
Input was: This text
```

.NET Framework Equivalent

- System::IO::StreamReader::Read
- System::Console::Read

See Also

Reference

Stream I/O
getc, getwc
fgetc, fgetwc
_getch, _getwch
putc, putwc
ungetc, ungetwc

_getchar_nolock, _getwchar_nolock

Reads a character from standard input.

```
int _getchar_nolock( void );
wint_t _getwchar_nolock( void );
```

Return Value

See getchar, getwchar.

Remarks

_getchar_nolock and **_getwchar_nolock** are identical to **getchar** and **getwchar** except that they are not protected from interference by other threads. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_gettchar_nolock	**_getchar_nolock**	**_getchar_nolock**	**_getwchar_nolock**

Requirements

Routine	Required header	Compatibility
_getchar_nolock	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_getwchar_nolock	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_getchar_nolock.c
// Use _getchar_nolock to read a line from stdin.

#include <stdio.h>

int main()
{
    char buffer[81];
    int i, ch;

    for (i = 0; (i < 80) && ((ch = _getchar_nolock()) != EOF)
                         && (ch != '\n'); i++)
    {
        buffer[i] = (char) ch;
    }

    // Terminate string with a null character

    buffer[i] = '\0';
    printf( "Input was: %s\n", buffer);
}
```

Input

```
This text
```

Output

```
Input was: This text
```

.NET Framework Equivalent

- System::IO::StreamReader::Read
- System::Console::Read

See Also

Reference

Stream I/O
getc, getwc
fgetc, fgetwc
_getch, _getwch
putc, putwc
ungetc, ungetwc

getche

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _getche instead.

_getche, _getwche

Gets a character from the console with echo.

```
int _getche( void );
wint_t _getwche( void );
```

Return Value

Returns the character read. There is no error return.

Remarks

The **_getch** and **_getwch** functions read a single character from the console with echo, meaning that the character is displayed at the console. None of these functions can be used to read CTRL+C. When reading a function key or an arrow key, each function must be called twice; the first call returns 0 or 0xE0, and the second call returns the actual key code.

These functions lock the calling thread and are therefore thread-safe. For non-locking versions, see _getche_nolock, _getwche_nolock.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_gettche	**_getche**	**_getch**	**_getwche**

Requirements

Routine	Required header	Compatibility
_getche	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_getwche	<conio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_getche.c
// compile with: /c
// This program reads characters from
// the keyboard until it receives a 'Y' or 'y'.

#include <conio.h>
#include <ctype.h>

int main( void )
{
   int ch;

   _cputs( "Type 'Y' when finished typing keys: " );
   do
   {
      ch = _getche();
      ch = toupper( ch );
   } while( ch != 'Y' );

   _putch( ch );
   _putch( '\r' );    // Carriage return
   _putch( '\n' );    // Line feed
}
```

Input

```
abcdey
```

Output

```
Type 'Y' when finished typing keys: Y
```

NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Console and Port I/O
_cgets, _cgetws
getc, getwc
_ungetch, _ungetwch, _ungetch_nolock, _ungetwch_nolock

_getche_nolock, _getwche_nolock

Gets a character from the console with echo and without locking the thread.

```
int _getche_nolock( void );
wint_t _getwche_nolock( void );
```

Return Value

Returns the character read. There is no error return.

Remarks

_getche_nolock and **_getwche_nolock** are identical to **_getche** and **_getwche** except that they not protected from interference by other threads. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_gettche_nolock	**_getche_nolock**	**_getch_nolock**	**_getwche_nolock**

Requirements

Routine	Required header	Compatibility
_getche_nolock	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_getwche_nolock	<conio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_getche_nolock.c
// compile with: /c
// This program reads characters from
// the keyboard until it receives a 'Y' or 'y'.


#include <conio.h>
#include <ctype.h>

int main( void )
{
   int ch;

   _cputs( "Type 'Y' when finished typing keys: " );
   do
   {
      ch = _getche_nolock();
      ch = toupper( ch );
   } while( ch != 'Y' );

   _putch_nolock( ch );
   _putch_nolock( '\r' );    // Carriage return
   _putch_nolock( '\n' );    // Line feed
}
```

Input

```
abcdey
```

Output

```
Type 'Y' when finished typing keys: Y
```

NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Console and Port I/O
_cgets, _cgetws
getc, getwc
_ungetch, _ungetwch, _ungetch_nolock, _ungetwch_nolock

getcwd

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _getcwd instead.

_getcwd, _wgetcwd

Gets the current working directory.

```
char *_getcwd(
   char *buffer,
   int maxlen
);
wchar_t *_wgetcwd(
   wchar_t *buffer,
   int maxlen
);
```

Parameters

buffer

Storage location for the path.

maxlen

Maximum length of the path in characters: **char** for **_getcwd** and **wchar_t** for **_wgetcwd**.

Return Value

Returns a pointer to *buffer*. A **NULL** return value indicates an error, and **errno** is set either to **ENOMEM**, indicating that there is insufficient memory to allocate *maxlen* bytes (when a **NULL** argument is given as *buffer*), or to **ERANGE**, indicating that the path is longer than *maxlen* characters. If *maxlen* is less than or equal to zero, this function invokes an invalid parameter handler, as described in Parameter Validation.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_getcwd** function gets the full path of the current working directory for the default drive and stores it at *buffer*. The integer argument *maxlen* specifies the maximum length for the path. An error occurs if the length of the path (including the terminating null character) exceeds *maxlen*. The *buffer* argument can be **NULL**; a buffer of at least size *maxlen* (more only if necessary) is automatically allocated, using **malloc**, to store the path. This buffer can later be freed by calling **free** and passing it the **_getcwd** return value (a pointer to the allocated buffer).

_getcwd returns a string that represents the path of the current working directory. If the current working directory is the root, the string ends with a backslash (****). If the current working directory is a directory other than the root, the string ends with the directory name and not with a backslash.

_wgetcwd is a wide-character version of **_getcwd**; the *buffer* argument and return value of **_wgetcwd** are wide-character strings. **_wgetcwd** and **_getcwd** behave identically otherwise.

When **_DEBUG** and **_CRTDBG_MAP_ALLOC** are defined, calls to **_getcwd** and **_wgetcwd** are replaced by calls to **_getcwd_dbg** and **_wgetcwd_dbg** to allow for debugging memory allocations. For more information, see _getcwd_dbg, _wgetcwd_dbg.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tgetcwd	**_getcwd**	**_getcwd**	**_wgetcwd**

Requirements

Routine	Required header	Compatibility
_getcwd	<direct.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wgetcwd	<direct.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_getcwd.c
// This program places the name of the current directory in the
// buffer array, then displays the name of the current directory
// on the screen. Passing NULL as the buffer forces getcwd to allocate
// memory for the path, which allows the code to support file paths
// longer than _MAX_PATH, which are supported by NTFS.

#include <direct.h>
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char* buffer;

   // Get the current working directory:
   if( (buffer = _getcwd( NULL, 0 )) == NULL )
      perror( "_getcwd error" );
   else
   {
      printf( "%s \nLength: %d\n", buffer, strlen(buffer) );
      free(buffer);
   }
}
```

Sample Output

```
C:\Code
```

.NET Framework Equivalent

System::Environment::CurrentDirectory

See Also

Reference

Directory Control
_chdir, _wchdir
_mkdir, _wmkdir
_rmdir, _wrmdir

_getcwd_dbg, _wgetcwd_dbg

Debug versions of the _getcwd, _wgetcwd functions (only available during debug).

```
char *_getcwd_dbg(
   char *buffer,
   int maxlen,
   int blockType,
   const char *filename,
   int linenumber
);
wchar_t *_wgetcwd_dbg(
   wchar_t *buffer,
   int maxlen,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

buffer

Storage location for the path.

maxlen

Maximum length of the path in characters: **char** for **_getcwd_dbg** and **wchar_t** for **_wgetcwd_dbg**.

blockType

Requested type of the memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

filename

Pointer to the name of the source file that requested the allocation operation or **NULL**.

linenumber

Line number in the source file where the allocation operation was requested or **NULL**.

Return Value

Returns a pointer to *buffer*. A **NULL** return value indicates an error, and **errno** is set either to **ENOMEM**, indicating that there is insufficient memory to allocate *maxlen* bytes (when a **NULL** argument is given as *buffer*), or to **ERANGE**, indicating that the path is longer than *maxlen* characters.

For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **_getcwd_dbg** and **_wgetcwd_dbg** functions are identical to **_getcwd** and **_wgetcwd** except that, when **_DEBUG** is defined, these functions use the debug version of **malloc** and **_malloc_dbg** to allocate memory if **NULL** is passed as the first parameter. For more information, see _malloc_dbg.

You do not need to call these functions explicitly in most cases. Instead, you can define the **_CRTDBG_MAP_ALLOC** flag. When **_CRTDBG_MAP_ALLOC** is defined, calls to **_getcwd** and **_wgetcwd** are remapped to **_getcwd_dbg** and **_wgetcwd_dbg**, respectively, with the *blockType* set to **_NORMAL_BLOCK**. Thus, you do not need to call these functions explicitly unless you want to mark the heap blocks as **_CLIENT_BLOCK**. For more information, see Types of Blocks on the Debug Heap.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tgetcwd_dbg	_getcwd_dbg	_getcwd_dbg	_wgetcwd_dbg

Requirements

Routine	Required header	Compatibility

_getcwd_dbg	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wgetcwd_dbg	<crtdbg.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

CurrentDirectory

See Also

Reference

_getcwd, _wgetcwd

Directory Control

Concepts

Debug Versions of Heap Allocation Functions

_getdcwd, _wgetdcwd

Gets the full path name of the current working directory on the specified drive.

```
char *_getdcwd(
   int drive,
   char *buffer,
   int maxlen
);
wchar_t *_wgetdcwd(
   int drive,
   wchar_t *buffer,
   int maxlen
);
```

Parameters

drive

Disk drive.

buffer

Storage location for the path.

maxlen

Maximum length of the path in characters: **char** for **_getdcwd** and **wchar_t** for **_wgetdcwd**.

Return Value

Returns *buffer*. A **NULL** return value indicates an error, and **errno** is set either to **ENOMEM**, indicating that there is insufficient memory to allocate *maxlen* bytes (when a **NULL** argument is given as *buffer*), or to **ERANGE**, indicating that the path is longer than *maxlen* characters. If the *drive* does not specify a valid drive or if *maxlen* is less than or equal to zero, this function invokes an invalid parameter handler, as described in Parameter Validation.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_getdcwd** function gets the full path of the current working directory on the specified drive and stores it at *buffer*. An error occurs if the length of the path (including the terminating null character) exceeds *maxlen*. The *drive* argument specifies the drive (0 = default drive, 1 = A, 2 = B, and so on). The *buffer* argument can be **NULL**; a buffer of at least size *maxlen* (more only if necessary) is automatically allocated, using **malloc**, to store the path. This buffer can later be freed by calling **free** and passing it the **_getdcwd** return value (a pointer to the allocated buffer).

_getdcwd returns a string that represents the path of the current working directory. If the current working directory is set to the root, the string ends with a backslash (\). If the current working directory is set to a directory other than the root, the string ends with the name of the directory and not with a backslash.

_wgetdcwd is a wide-character version of **_getdcwd**; the *buffer* argument and return value of **_wgetdcwd** are wide-character strings. **_wgetdcwd** and **_getdcwd** behave identically otherwise.

The versions with the **_nolock** suffix are identical except that they are not protected from interference by other threads. For more information, see _getdcwd_nolock, _wgetdcwd_nolock.

When **_DEBUG** and **_CRTDBG_MAP_ALLOC** are defined, calls to **_getdcwd** and **_wgetdcwd** are replaced by calls to **_getdcwd_dbg** and **_wgetdcwd_dbg** to allow for debugging memory allocations. For more information, see _getdcwd_dbg, _wgetdcwd_dbg.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tgetdcwd	**_getdcwd**	**_getdcwd**	**_wgetdcwd**

Requirements

Routine	Required header	Compatibility
_getdcwd	<direct.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wgetdcwd	<direct.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example in _getdrive.

.NET Framework Equivalent

System::Environment::CurrentDirectory

See Also

Reference

Directory Control
_chdir, _wchdir
_getcwd, _wgetcwd
_getdrive
_mkdir, _wmkdir
_rmdir, _wrmdir

_getdcwd_dbg, _wgetdcwd_dbg

Debug versions of the _getdcwd, _wgetdcwd functions (only available during debug).

```
char *_getdcwd_dbg(
   int drive,
   char *buffer,
   int maxlen,
   int blockType,
   const char *filename,
   int linenumber
);
wchar_t *_wgetdcwd_dbg(
   int drive,
   wchar_t *buffer,
   int maxlen,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

drive

Name of the disk drive.

buffer

Storage location for the path.

maxlen

Maximum length of the path in characters: **char** for **_getdcwd_dbg** and **wchar_t** for **_wgetdcwd_dbg**.

blockType

Requested type of the memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

filename

Pointer to the name of the source file that requested the allocation operation or **NULL**.

linenumber

Line number in the source file where the allocation operation was requested or **NULL**.

Return Value

Returns a pointer to *buffer*. A **NULL** return value indicates an error, and *errno* is set either to **ENOMEM**, indicating that there is insufficient memory to allocate *maxlen* bytes (when a **NULL** argument is given as *buffer*), or to **ERANGE**, indicating that the path is longer than *maxlen* characters. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **_getdcwd_dbg** and **_wgetdcwd_dbg** functions are identical to **_getdcwd** and **_wgetdcwd** except that, when **_DEBUG** is defined, these functions use the debug version of **malloc** and **_malloc_dbg** to allocate memory if **NULL** is passed as the *buffer* parameter. For more information, see _malloc_dbg.

You do not need to call these functions explicitly in most cases. Instead, you can define the **_CRTDBG_MAP_ALLOC** flag. When **_CRTDBG_MAP_ALLOC** is defined, calls to **_getdcwd** and **_wgetdcwd** are remapped to **_getdcwd_dbg** and **_wgetdcwd_dbg**, respectively, with the *blockType* set to **_NORMAL_BLOCK**. Thus, you do not need to call these functions explicitly unless you want to mark the heap blocks as **_CLIENT_BLOCK**. For more information, see Types of Blocks on the Debug Heap.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined

_tgetdcwd_dbg	_getdcwd_dbg	_getdcwd_dbg	_wgetdcwd_dbg

Requirements

Routine	Required header	Compatibility
_getdcwd_dbg	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wgetdcwd_dbg	<crtdbg.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System.Environment.CurrentDirectory

See Also

Reference

_getdcwd, _wgetdcwd

Directory Control

Concepts

Debug Versions of Heap Allocation Functions

_getdcwd_nolock, _wgetdcwd_nolock

Gets the full path name of the current working directory on the specified drive.

```
char *_getdcwd_nolock(
   int drive,
   char *buffer,
   int maxlen
);
wchar_t *_wgetdcwd_nolock(
   int drive,
   wchar_t *buffer,
   int maxlen
);
```

Parameters

drive

Disk drive.

buffer

Storage location for the path.

maxlen

Maximum length of path in characters: **char** for **_getdcwd** and **wchar_t** for **_wgetdcwd**.

Return Value

See _getdcwd, _wgetdcwd.

Remarks

_getdcwd_nolock and **_wgetdcwd_nolock** are identical to **_getdcwd** and **_wgetdcwd**, respectively, except that they are not protected from interference by other threads. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tgetdcwd_nolock	**_getdcwd_nolock**	**_getdcwd_nolock**	**_wgetdcwd_nolock**

Requirements

Routine	Required header	Compatibility
_getdcwd_nolock	<direct.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wgetdcwd_nolock	<direct.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::Environment::CurrentDirectory

See Also

Reference

Directory Control

_chdir, _wchdir

_getcwd, _wgetcwd
_getdrive
_mkdir, _wmkdir
_rmdir, _wrmdir

_getdiskfree

Populates a **_diskfree_t** structure with information about a disk drive.

```
unsigned _getdiskfree(
   unsigned drive,
   struct _diskfree_t * driveinfo
);
```

Parameters

[in] *drive*

The disk drive for which you want information.

[out] *driveinfo*

A **_diskfree_t** structure that will be populated with information about the drive.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is the error code. The value **errno** is set for any errors returned by the operating system. For more information about error conditions indicated by **errno**, see errno.

Remarks

The **_diskfree_t** structure is defined in Direct.h.

```
struct _diskfree_t {
   unsigned total_clusters;
   unsigned avail_clusters;
   unsigned sectors_per_cluster;
   unsigned bytes_per_sector;
};
```

This function validates its parameters. If the *driveinfo* pointer is **NULL** or *drive* specifies an invalid drive, this function invokes an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the function returns **EINVAL** and sets **errno** to **EINVAL**. Valid drives range from 0 to 26. A *drive* value of 0 specifies the current drive; thereafter, numbers map onto letters of the English alphabet, so 1 indicates drive A, 3 indicates drive C, and so on.

total_clusters

The total number of clusters, used and available, on the disk.

avail_clusters

The number of unused clusters on the disk.

sectors_per_cluster

The number of sectors in each cluster.

bytes_per_sector

The size of each sector in bytes.

Requirements

Routine	Required header	Compatibility
_getdiskfree	<direct.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_getdiskfree.c
// compile with: /c
#include <windows.h>
#include <direct.h>
#include <stdio.h>
#include <tchar.h>

TCHAR   g_szBorder[] = _T("================================================================
======\n");
TCHAR   g_szTitle1[] = _T("|DRIVE|TOTAL CLUSTERS|AVAIL CLUSTERS|SECTORS / CLUSTER|BYTES / S
ECTOR|\n");
TCHAR   g_szTitle2[] = _T("|=====|==============|==============|=================|========
=====|\n");
TCHAR   g_szLine[]   = _T("|  A: |              |              |                 |
|\n");

void utoiRightJustified(TCHAR* szLeft, TCHAR* szRight, unsigned uVal);

int main(int argc, char* argv[]) {
   TCHAR szMsg[4200];
   struct _diskfree_t df = {0};
   ULONG uDriveMask = _getdrives();
   unsigned uErr, uLen, uDrive;

   printf(g_szBorder);
   printf(g_szTitle1);
   printf(g_szTitle2);

   for (uDrive=1; uDrive<=26; ++uDrive) {
      if (uDriveMask & 1) {
         uErr = _getdiskfree(uDrive, &df);
         memcpy(szMsg, g_szLine, sizeof(g_szLine));
         szMsg[3] = uDrive + 'A' - 1;

         if (uErr == 0) {
            utoiRightJustified(szMsg+8,  szMsg+19, df.total_clusters);
            utoiRightJustified(szMsg+23, szMsg+34, df.avail_clusters);
            utoiRightJustified(szMsg+38, szMsg+52, df.sectors_per_cluster);
            utoiRightJustified(szMsg+56, szMsg+67, df.bytes_per_sector);
         }
         else {
            uLen = FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, NULL,
                            uErr, 0, szMsg+8, 4100, NULL);
            szMsg[uLen+6] = ' ';
            szMsg[uLen+7] = ' ';
            szMsg[uLen+8] = ' ';
         }

         printf(szMsg);
      }

      uDriveMask >>= 1;
   }

   printf(g_szBorder);
}

void utoiRightJustified(TCHAR* szLeft, TCHAR* szRight, unsigned uVal) {
   TCHAR* szCur = szRight;
   int nComma = 0;

   if (uVal) {
      while (uVal && (szCur >= szLeft)) {
         if   (nComma == 3) {
            *szCur = ',';
            nComma = 0;
         }
         else {
```

```
            *szCur = (uVal % 10) | 0x30;
            uVal /= 10;
            ++nComma;
         }

         --szCur;
      }
   }
   else {
      *szCur = '0';
      --szCur;
   }

   if (uVal) {
      szCur = szLeft;

      while   (szCur <= szRight) {
         *szCur = '*';
         ++szCur;
      }
   }
}
```

Sample Output

```
=====================================================================
|DRIVE|TOTAL CLUSTERS|AVAIL CLUSTERS|SECTORS / CLUSTER|BYTES / SECTOR|
|=====|==============|==============|=================|==============|
|  A: | The device is not ready.    |                 |              |
|  C: |    4,721,093 |    3,778,303 |               8 |          512 |
|  D: |    1,956,097 |    1,800,761 |               8 |          512 |
|  E: | The device is not ready.    |                 |              |
=====================================================================
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Directory Control

_getdrive

Gets the current disk drive.

```
int _getdrive( void );
```

Return Value

Returns the current (default) drive (1=A, 2=B, and so on). There is no error return.

Requirements

Routine	Required header	Compatibility
_getdrive	<direct.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_getdrive.c
// compile with: /c
// Illustrates drive functions including:
//    _getdrive       _chdrive        _getdcwd
//

#include <stdio.h>
#include <direct.h>
#include <stdlib.h>
#include <ctype.h>

int main( void )
{
   int ch, drive, curdrive;
   static char path[_MAX_PATH];

   // Save current drive.
   curdrive = _getdrive();

   printf( "Available drives are:\n" );

   // If we can switch to the drive, it exists.
   for( drive = 1; drive <= 26; drive++ )
   {
      if( !_chdrive( drive ) )
      {
         printf( "%c:", drive + 'A' - 1 );
         if( _getdcwd( drive, path, _MAX_PATH ) != NULL )
            printf( " (Current directory is %s)", path );
         putchar( '\n' );
      }
   }

   // Restore original drive.
   _chdrive( curdrive );
}
```

Sample Output

```
Available drives are:
A: (Current directory is A:\)
C: (Current directory is C:\)
```

```
E: (Current directory is E:\testdir\bin)
F: (Current directory is F:\)
G: (Current directory is G:\)
```

.NET Framework Equivalent

System::Environment::CurrentDirectory

See Also

Reference

Directory Control
_chdrive
_getcwd, _wgetcwd
_getdcwd, _wgetdcwd

_getdrives

Returns a bitmask representing the currently available disk drives.

```
unsigned long _getdrives( void );
```

Return Value

If the function succeeds, the return value is a bitmask representing the currently available disk drives. Bit position 0 (the least-significant bit) is drive A, bit position 1 is drive B, bit position 2 is drive C, and so on. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Routine	Required header	Compatibility
_getdrives	<direct.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_getdrives.c
// This program retrives and lists out
// all the logical drives that are
// currently mounted on the machine.

#include <windows.h>
#include <direct.h>
#include <stdio.h>
#include <tchar.h>

TCHAR g_szDrvMsg[] = _T("A:\n");

int main(int argc, char* argv[]) {
   ULONG uDriveMask = _getdrives();

   if (uDriveMask == 0)
   {
      printf( "_getdrives() failed with failure code: %d\n",
              GetLastError());
   }
   else
   {
      printf("The following logical drives are being used:\n");

      while (uDriveMask) {
         if (uDriveMask & 1)
            printf(g_szDrvMsg);

         ++g_szDrvMsg[0];
         uDriveMask >>= 1;
      }
   }
}
```

Sample Output

```
The following logical drives are being used:
A:
C:
D:
E:
```

NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Directory Control

getenv, _wgetenv

Get a value from the current environment. These functions are deprecated because more secure versions are available; see getenv_s, _wgetenv_s.

```
char *getenv(
   const char *varname
);
wchar_t *_wgetenv(
   const wchar_t *varname
);
```

Parameters

varname

Environment variable name.

Return Value

Returns a pointer to the environment table entry containing *varname*. It is not safe to modify the value of the environment variable using the returned pointer. Use the **_putenv** function to modify the value of an environment variable. The return value is **NULL** if *varname* is not found in the environment table.

Remarks

The **getenv** function searches the list of environment variables for *varname*. **getenv** is not case sensitive in the Windows operating system. **getenv** and **_putenv** use the copy of the environment pointed to by the global variable **_environ** to access the environment. **getenv** operates only on the data structures accessible to the run-time library and not on the environment "segment" created for the process by the operating system. Therefore, programs that use the *envp* argument to main or wmain may retrieve invalid information.

If *varname* is **NULL**, this function invokes an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **NULL**.

_wgetenv is a wide-character version of **getenv**; the argument and return value of **_wgetenv** are wide-character strings. The **_wenviron** global variable is a wide-character version of **_environ**.

In an MBCS program (for example, in an SBCS ASCII program), **_wenviron** is initially **NULL** because the environment is composed of multibyte-character strings. Then, on the first call to **_wputenv**, or on the first call to **_wgetenv** if an (MBCS) environment already exists, a corresponding wide-character string environment is created and is then pointed to by **_wenviron**.

Similarly in a Unicode (**_wmain**) program, **_environ** is initially **NULL** because the environment is composed of wide-character strings. Then, on the first call to **_putenv**, or on the first call to **getenv** if a (Unicode) environment already exists, a corresponding MBCS environment is created and is then pointed to by **_environ**.

When two copies of the environment (MBCS and Unicode) exist simultaneously in a program, the run-time system must maintain both copies, resulting in slower execution time. For example, whenever you call **_putenv**, a call to **_wputenv** is also executed automatically, so that the two environment strings correspond.

Caution

In rare instances, when the run-time system is maintaining both a Unicode version and a multibyte version of the environment, these two environment versions may not correspond exactly. This is because, although any unique multibyte-character string maps to a unique Unicode string, the mapping from a unique Unicode string to a multibyte-character string is not necessarily unique. For more information, see _environ, _wenviron.

Note

The **_putenv** and **_getenv** families of functions are not thread-safe. **_getenv** could return a string pointer while **_putenv** is modifying the string, causing random failures. Make sure that calls to these functions are synchronized.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tgetenv	**getenv**	**getenv**	**_wgetenv**

To check or change the value of the **TZ** environment variable, use **getenv**, **_putenv** and **_tzset** as necessary. For more information about **TZ**, see _tzset and _daylight, timezone, and _tzname.

Requirements

Routine	Required header	Compatibility
getenv	<stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wgetenv	<stdlib.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_getenv.c
// compile with: /W3
// This program uses getenv to retrieve
// the LIB environment variable and then uses
// _putenv to change it to a new value.

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char *libvar;

   // Get the value of the LIB environment variable.
   libvar = getenv( "LIB" ); // C4996
   // Note: getenv is deprecated; consider using getenv_s instead

   if( libvar != NULL )
      printf( "Original LIB variable is: %s\n", libvar );

   // Attempt to change path. Note that this only affects the environment
   // variable of the current process. The command processor's
   // environment is not changed.
   _putenv( "LIB=c:\\mylib;c:\\yourlib" ); // C4996
   // Note: _putenv is deprecated; consider using putenv_s instead

   // Get new value.
   libvar = getenv( "LIB" ); // C4996

   if( libvar != NULL )
      printf( "New LIB variable is: %s\n", libvar );
}
```

Sample Output

```
Original LIB variable is: C:\progra~1\devstu~1\vc\lib
New LIB variable is: c:\mylib;c:\yourlib
```

.NET Framework Equivalent

System::Environment::GetEnvironmentVariable

See Also

Reference

Process and Environment Control

_putenv, _wputenv
Environmental Constants

getenv_s, _wgetenv_s

Get a value from the current environment. These are versions of getenv, _wgetenv with security enhancements as described in Security Enhancements in the CRT.

```
errno_t getenv_s(
   size_t *pReturnValue,
   char* buffer,
   size_t numberOfElements,
   const char *varname
);
errno_t _wgetenv_s(
   size_t *pReturnValue,
   wchar_t *buffer,
     size_t numberOfElements,
   const wchar_t *varname
);
template <size_t size>
errno_t getenv_s(
   size_t *pReturnValue,
   char (&buffer)[size],
   const char *varname
); // C++ only
template <size_t size>
errno_t _wgetenv_s(
   size_t *pReturnValue,
   wchar_t (&buffer)[size],
   const wchar_t *varname
); // C++ only
```

Parameters

pReturnValue

The buffer size needed, or 0 if the variable is not found.

buffer

Buffer to store the variable and its value.

numberOfElements

Size of *buffer*.

varname

Environment variable name.

Return Value

Zero if successful, an error code on failure.

Error Conditions

pReturnValue	***buffer***	***numberOfElements***	***varname***	**Return Value**
NULL	any	any	any	**EINVAL**
any	**NULL**	>0	any	**EINVAL**
any	any	any	**NULL**	**EINVAL**

Any of the above error conditions will invoke an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the functions set **errno** to **EINVAL** and return **EINVAL**.

Also, if the buffer is too small, these functions return **ERANGE**. They do not invoke an invalid parameter handler. They will write out the required buffer size in *pReturnValue*, enabling programs to call the function again with a larger buffer.

Remarks

The **getenv_s** function searches the list of environment variables for *varname*. **getenv_s** is not case sensitive in the Windows operating system. **getenv_s** and **_putenv_s** use the copy of the environment pointed to by the global variable **_environ** to access the environment. **getenv_s** operates only on the data structures accessible to the run-time library and not on the environment "segment" created for the process by the operating system. Therefore, programs that use the *envp* argument to main or wmain may retrieve invalid information.

It is not safe to modify the value of the environment variable using the pointer returned in *buffer*. Use the **_putenv_s** function to modify the value of an environment variable. The value of *buffer* is set to an empty string if *varname* is not found in the environment table.

_wgetenv_s is a wide-character version of **getenv_s**; the argument and return value of **_wgetenv_s** are wide-character strings. The **_wenviron** global variable is a wide-character version of **_environ**.

In an MBCS program (for example, in an SBCS ASCII program), **_wenviron** is initially **NULL** because the environment is composed of multibyte-character strings. Then, on the first call to **_wputenv**, or on the first call to **_wgetenv_s** if an (MBCS) environment already exists, a corresponding wide-character string environment is created and is then pointed to by **_wenviron**.

Similarly in a Unicode **(_wmain**) program, **_environ** is initially **NULL** because the environment is composed of wide-character strings. Then, on the first call to **_putenv**, or on the first call to **getenv_s** if a (Unicode) environment already exists, a corresponding MBCS environment is created and is then pointed to by **_environ**.

When two copies of the environment (MBCS and Unicode) exist simultaneously in a program, the run-time system must maintain both copies, resulting in slower execution time. For example, whenever you call **_putenv**, a call to **_wputenv** is also executed automatically, so that the two environment strings correspond.

Caution
In rare instances, when the run-time system is maintaining both a Unicode version and a multibyte version of the environment, these two environment versions may not correspond exactly. This is because, although any unique multibyte-character string maps to a unique Unicode string, the mapping from a unique Unicode string to a multibyte-character string is not necessarily unique. For more information, see _environ, _wenviron.

Note
The **_putenv_s** and **_getenv_s** families of functions are not thread-safe. **_getenv_s** could return a string pointer while **_putenv_s** is modifying the string, causing random failures. Make sure that calls to these functions are synchronized.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically, eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tgetenv_s	**getenv_s**	**getenv_s**	**_wgetenv_s**

To check or change the value of the **TZ** environment variable, use **getenv_s**, **_putenv** and **_tzset** as necessary. For more information about **TZ**, see _tzset and see _daylight, timezone, and _tzname.

Requirements

Routine	Required header	Compatibility
getenv_s	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wgetenv_s	<stdlib.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_getenv_s.c
// This program uses getenv_s to retrieve
// the LIB environment variable and then uses
// _putenv to change it to a new value.

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char* libvar;
   size_t requiredSize;

   getenv_s( &requiredSize, NULL, 0, "LIB");

   libvar = (char*) malloc(requiredSize * sizeof(char));
   if (!libvar)
   {
      printf("Failed to allocate memory!\n");
      exit(1);
   }

   // Get the value of the LIB environment variable.
   getenv_s( &requiredSize, libvar, requiredSize, "LIB" );

   if( libvar != NULL )
      printf( "Original LIB variable is: %s\n", libvar );

   // Attempt to change path. Note that this only affects
   // the environment variable of the current process. The command
   // processor's environment is not changed.
   _putenv_s( "LIB", "c:\\mylib;c:\\yourlib" );

   getenv_s( &requiredSize, NULL, 0, "LIB");

   libvar = (char*) malloc(requiredSize * sizeof(char));
   if (!libvar)
   {
      printf("Failed to allocate memory!\n");
      exit(1);
   }

   // Get the new value of the LIB environment variable.
   getenv_s( &requiredSize, libvar, requiredSize, "LIB" );

   if( libvar != NULL )
      printf( "New LIB variable is: %s\n", libvar );
}
```

Sample Output

```
Original LIB variable is: c:\vctools\lib;c:\vctools\atlmfc\lib;c:\vctools\PlatformSDK\l
ib;c:\vctools\Visual Studio SDKs\DIA Sdk\lib;c:\vctools\Visual Studio SDKs\BSC Sdk\lib
New LIB variable is: c:\mylib;c:\yourlib
```

.NET Framework Equivalent

System::Environment::GetEnvironmentVariable

See Also

Reference

Process and Environment Control
Environmental Constants
_putenv, _wputenv
_dupenv_s, _wdupenv_s

_getmaxstdio

Returns the number of simultaneously open files permitted at the stream I/O level.

```
int _getmaxstdio( void );
```

Return Value

Returns a number that represents the number of simultaneously open files currently permitted at the **stdio** level.

Remarks

Use _setmaxstdio to configure the number of simultaneously open files permitted at the **stdio** level.

Requirements

Routine	Required header	Compatibility
_getmaxstdio	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_setmaxstdio.c
// The program retrieves the maximum number
// of open files and prints the results
// to the console.

#include <stdio.h>

int main()
{
   printf( "%d\n", _getmaxstdio());

   _setmaxstdio(2048);

   printf( "%d\n", _getmaxstdio());
}
```

Output

```
512
2048
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O

_getmbcp

Retrieves the current code page.

```
int _getmbcp( void );
```

Return Value

Returns the current multibyte code page. A return value of 0 indicates that a single byte code page is in use.

Requirements

Routine	Required header	Compatibility
_getmbcp	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

_setmbcp

getpid

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _getpid instead.

_getpid

Gets the process identification.

```
int _getpid( void );
```

Return Value

Returns the process ID obtained from the system. There is no error return.

Remarks

The **_getpid** function obtains the process ID from the system. The process ID uniquely identifies the calling process.

Requirements

Routine	Required header	Compatibility
_getpid	<process.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_getpid.c
// This program uses _getpid to obtain
// the process ID and then prints the ID.


#include <stdio.h>
#include <process.h>

int main( void )
{
   // If run from command line, shows different ID for
   // command line than for operating system shell.

   printf( "Process id: %d\n", _getpid() );
}
```

Sample Output

```
Process id: 3584
```

.NET Framework Equivalent

System::Diagnostics::Process::Id

See Also

Reference

Process and Environment Control

_mktemp, _wmktemp

gets, _getws

Get a line from the **stdin** stream. These functions are deprecated because more secure versions are available; see gets_s, _getws_s.

```
char *gets(
   char *buffer
);
wchar_t *_getws(
   wchar_t *buffer
);
template <size_t size>
char *gets(
   char (&buffer)[size]
); // C++ only
template <size_t size>
wchar_t *_getws(
   wchar_t (&buffer)[size]
); // C++ only
```

Parameters

buffer

Storage location for input string.

Return Value

Returns its argument if successful. A **NULL** pointer indicates an error or end-of-file condition. Use ferror or feof to determine which one has occurred. If *buffer* is **NULL**, these functions invoke an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return **NULL** and set errno to **EINVAL**.

Remarks

The **gets** function reads a line from the standard input stream **stdin** and stores it in *buffer*. The line consists of all characters up to and including the first newline character ('\n'). **gets** then replaces the newline character with a null character ('\0') before returning the line. In contrast, the **fgets** function retains the newline character. **_getws** is a wide-character version of **gets**; its argument and return value are wide-character strings.

Security Note
Because there is no way to limit the number of characters read by gets, untrusted input can easily cause buffer overruns. Use **fgets** instead.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_getts	**gets**	**gets**	**_getws**

Requirements

Routine	Required header	Compatibility
gets	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_getws	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_gets.c
// compile with: /W3

#include <stdio.h>

int main( void )
{
   char line[21]; // room for 20 chars + '\0'
   gets( line );  // C4996
   // Danger: No way to limit input to 20 chars.
   // Consider using gets_s instead.
   printf( "The line entered was: %s\n", line );
}
```

Note that input longer than 20 characters will overrun the line buffer and almost certainly cause the program to crash.

Input

```
Hello there!
```

Output

```
The line entered was: Hello there!
```

.NET Framework Equivalent

System::Console::Read

See Also

Reference

Stream I/O
fgets, fgetws
fputs, fputws
puts, _putws

gets_s, _getws_s

Get a line from the **stdin** stream. These are versions of gets, _getws with security enhancements as described in Security Enhancements in the CRT.

```
char *gets_s(
   char *buffer,
   size_t sizeInCharacters
);
wchar_t *_getws_s(
   wchar_t *buffer,
   size_t sizeInCharacters
);
template <size_t size>
char *gets_s(
   char (&buffer)[size]
); // C++ only
template <size_t size>
wchar_t *_getws_s(
   wchar_t (&buffer)[size]
); // C++ only
```

Parameters

[out] *buffer*

Storage location for input string.

[in] *sizeInCharacters*

The size of the buffer.

Return Value

Returns *buffer* if successful. A **NULL** pointer indicates an error or end-of-file condition. Use ferror or feof to determine which one has occurred.

Remarks

The **gets_s** function reads a line from the standard input stream **stdin** and stores it in *buffer*. The line consists of all characters up to and including the first newline character ('\n'). **gets_s** then replaces the newline character with a null character ('\0') before returning the line. In contrast, the **fgets_s** function retains the newline character.

If the first character read is the end-of-file character, a null character is stored at the beginning of *buffer* and **NULL** is returned.

_getws is a wide-character version of **gets_s**; its argument and return value are wide-character strings.

If *buffer* is **NULL** or *sizeInCharacters* is less than or equal to zero, or if the buffer is too small to contain the input line and null terminator, these functions invoke an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return **NULL** and set errno to **EINVAL**.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_getts	**gets_s**	**gets_s**	**_getws**

Requirements

Routine	Required header	Compatibility
gets_s	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_getws	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_gets_s.c
// This program retrieves a string from the stdin and
// prints the same string to the console.

#include <stdio.h>

int main( void )
{
   char line[21]; // room for 20 chars + '\0'
   gets_s( line, 20 );
   printf( "The line entered was: %s\n", line );
}
```

Input

```
Hello there!
```

Output

```
The line entered was: Hello there!
```

.NET Framework Equivalent

System::Console::Read

See Also

Reference

Stream I/O
gets, _getws
fgets, fgetws
fputs, fputws
puts, _putws

getw

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _getw instead.

_getw

Gets an integer from a stream.

```
int _getw(
   FILE *stream
);
```

Parameters

stream

Pointer to the **FILE** structure.

Return Value

_getw returns the integer value read. A return value of **EOF** indicates either an error or end of file. However, because the **EOF** value is also a legitimate integer value, use **feof** or **ferror** to verify an end-of-file or error condition. If *stream* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **EOF**.

Remarks

The **_getw** function reads the next binary value of type **int** from the file associated with *stream* and increments the associated file pointer (if there is one) to point to the next unread character. **_getw** does not assume any special alignment of items in the stream. Problems with porting can occur with **_getw** because the size of the **int** type and the ordering of bytes within the **int** type differ across systems.

Requirements

Routine	Required header	Compatibility
_getw	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_getw.c
// This program uses _getw to read a word
// from a stream, then performs an error check.


#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   FILE *stream;
   int i;

   if( fopen_s( &stream, "crt_getw.txt", "rb" ) )
      printf( "Couldn't open file\n" );
   else
   {
      // Read a word from the stream:
      i = _getw( stream );

      // If there is an error...
      if( ferror( stream ) )
      {
         printf( "_getw failed\n" );
         clearerr_s( stream );
      }
```

```
        else
           printf( "First data word in file: 0x%.4x\n", i );
        fclose( stream );
     }
  }
```

Input: crt_getw.txt

```
  Line one.
  Line two.
```

Output

```
First data word in file: 0x656e694c
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O

_putw

gmtime, _gmtime32, _gmtime64

Convert a time value to a structure. These functions are deprecated because more secure versions are available; see _gmtime_s, _gmtime32_s, _gmtime64_s.

```
struct tm *gmtime(
   const time_t *timer
);
struct tm *_gmtime32(
   const time32_t *timer
);
struct tm *_gmtime64(
   const __time64_t *timer
);
```

Parameters

timer

Pointer to stored time. The time is represented as seconds elapsed since midnight (00:00:00), January 1, 1970, coordinated universal time (UTC).

Return Value

Return a pointer to a structure of type tm. The fields of the returned structure hold the evaluated value of the *timer* argument in UTC rather than in local time. Each of the structure fields is of type **int**, as follows:

tm_sec

Seconds after minute (0 – 59).

tm_min

Minutes after hour (0 – 59).

tm_hour

Hours since midnight (0 – 23).

tm_mday

Day of month (1 – 31).

tm_mon

Month (0 – 11; January = 0).

tm_year

Year (current year minus 1900).

tm_wday

Day of week (0 – 6; Sunday = 0).

tm_yday

Day of year (0 – 365; January 1 = 0).

tm_isdst

Always 0 for **gmtime**.

Both the 32-bit and 64-bit versions of **gmtime**, **mktime**, **mkgmtime**, and **localtime** all use a single **tm** structure per thread for the conversion. Each call to one of these functions destroys the result of any previous call. If *timer* represents a date before midnight, January 1, 1970, **gmtime** returns **NULL**. There is no error return.

_gmtime64, which uses the **__time64_t** structure, allows dates to be expressed up through 23:59:59, December 31, 3000, UTC, whereas **_gmtime32** only represent dates through 03:14:07 January 19, 2038, UTC. Midnight, January 1, 1970, is the lower bound of the date range for both these functions.

In Visual C++ 2005, **gmtime** is an inline function which evaluates to **_gmtime64** and **time_t** is equivalent to **__time64_t**. If you need to force the compiler to interpret **time_t** as the old 32-bit **time_t**, you can define **_USE_32BIT_TIME_T**. Doing this will cause **gmtime** to be in-lined to **_gmtime32**. This is not recommended because your application may fail after January 18, 2038, and it is not allowed on 64-bit platforms.

These functions validate their parameter. If *timer* is a null pointer, or if the timer value is negative, these functions invoke an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the functions return **NULL** and set *errno* to *EINVAL*.

Remarks

The *_gmtime32* function breaks down the *timer* value and stores it in a statically allocated structure of type *tm*, defined in TIME.H. The value of *timer* is usually obtained from a call to the *time* function.

Note
The target environment should try to determine whether daylight savings time is in effect. The C run-time library assumes the United States rules for implementing the calculation of Daylight Saving Time (DST).

Requirements

Routine	**Required header**	**Compatibility**
gmtime	<time.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_gmtime32	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_gmtime64	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_gmtime.c
// compile with: /W3
// This program uses _gmtime64 to convert a long-
// integer representation of coordinated universal time
// to a structure named newtime, then uses asctime to
// convert this structure to an output string.

#include <time.h>
#include <stdio.h>

int main( void )
{
   struct tm *newtime;
   __int64 ltime;
   char buff[80];

   _time64( &ltime );

   // Obtain coordinated universal time:
   newtime = _gmtime64( &ltime ); // C4996
   // Note: _gmtime64 is deprecated; consider using _gmtime64_s
   asctime_s( buff, sizeof(buff), newtime );
   printf( "Coordinated universal time is %s\n", buff );
}
```

Sample Output

```
Coordinated universal time is Tue Feb 12 23:11:31 2002
```

.NET Framework Equivalent

- System::DateTime::UtcNow
- System::DateTime::ToUniversalTime

See Also

Reference

Time Management
asctime, _wasctime
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
_ftime, _ftime32, _ftime64
_gmtime_s, _gmtime32_s, _gmtime64_s
localtime, _localtime32, _localtime64
_mkgmtime, _mkgmtime32, _mkgmtime64
mktime, _mktime32, _mktime64
time, _time32, _time64

_gmtime_s, _gmtime32_s, _gmtime64_s

Converts a time value to a structure. These are versions of _gmtime32, _gmtime64 with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _gmtime_s(
   struct tm* _tm,
   const __time_t* time
);
errno_t _gmtime32_s(
   struct tm* _tm,
   const __time32_t* time
);
errno_t _gmtime64_s(
   struct tm* _tm,
   const __time64_t* time
);
```

Parameters

_tm

Pointer to a **tm** structure. The fields of the returned structure hold the evaluated value of the *timer* argument in UTC rather than in local time.

time

Pointer to stored time. The time is represented as seconds elapsed since midnight (00:00:00), January 1, 1970, coordinated universal time (UTC).

Return Value

Zero if successful. The return value is an error code if there is a failure. Error codes are defined in Errno.h; for a listing of these errors, see errno. If any of the conditions below applies, the function invokes an invalid parameter handler, as described in Parameter Validation.

Error Conditions

_tm	***time***	**Return**	**Value in *_tm***
NULL	any	**EINVAL**	Not modified.
Not **NULL** (points to valid memory)	**NULL**	**EINVAL**	All fields set to -1.
Not **NULL**	< 0	**EINVAL**	All fields set to -1.

Remarks

The **_gmtime32_s** function breaks down the *time* value and stores it in a statically allocated structure of type **tm**, defined in Time.h. The value of *time* is usually obtained from a call to the **time** function.

Note
The target environment should try to determine whether daylight savings time is in effect. The C run-time library assumes the United States rules for implementing the calculation of daylight saving time .

Each of the structure fields is of type **int**, as shown in the following table.

tm_sec

Seconds after minute (0 – 59).

tm_min

Minutes after hour (0 – 59).

tm_hour

Hours since midnight (0 – 23).

tm_mday

Day of month (1 – 31).

tm_mon

Month (0 – 11; January = 0).

tm_year

Year (current year minus 1900).

tm_wday

Day of week (0 – 6; Sunday = 0).

tm_yday

Day of year (0 – 365; January 1 = 0).

tm_isdst

Always 0 for **gmtime**.

_gmtime64_s, which uses the **__time64_t** structure, allows dates to be expressed up through 23:59:59, December 31, 3000, UTC; whereas **gmtime32_s** only represent dates through 03:14:07 January 19, 2038, UTC. Midnight, January 1, 1970, is the lower bound of the date range for both these functions.

In Visual C++ 2005, **gmtime_s** is an inline function which evaluates to **_gmtime64_s** and **time_t** is equivalent to **__time64_t**. If you need to force the compiler to interpret **time_t** as the old 32-bit **time_t**, you can define **_USE_32BIT_TIME_T**. Doing this will cause **gmtime_s** to be in-lined to **_gmtime32_s**. This is not recommended because your application may fail after January 18, 2038, and it is not allowed on 64-bit platforms.

Requirements

Routine	Required header	Compatibility
_gmtime_s	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_gmtime32_s	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_gmtime64_s	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_gmtime64_s.c
// This program uses _gmtime64_s to convert a 64-bit
// integer representation of coordinated universal time
// to a structure named newtime, then uses asctime_s to
// convert this structure to an output string.


#include <time.h>
#include <stdio.h>

int main( void )
{
   struct tm newtime;
   __int64 ltime;
```

```
    char buf[26];
    errno_t err;

    _time64( &ltime );

    // Obtain coordinated universal time:
    err = _gmtime64_s( &newtime, &ltime );
    if (err)
    {
       printf("Invalid Argument to _gmtime64_s.");
    }

    // Convert to an ASCII representation
    err = asctime_s(buf, 26, &newtime);
    if (err)
    {
       printf("Invalid Argument to asctime_s.");
    }

    printf( "Coordinated universal time is %s\n",
            buf );
 }
```

Sample Output

```
Coordinated universal time is Fri Apr 25 20:12:33 2003
```

.NET Framework Equivalent

- System::DateTime::UtcNow
- System::DateTime::ToUniversalTime

See Also

Reference

Time Management
asctime_s, _wasctime_s
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
_ftime, _ftime32, _ftime64
gmtime, _gmtime32, _gmtime64
localtime_s, _localtime32_s, _localtime64_s
_mkgmtime, _mkgmtime32, _mkgmtime64
mktime, _mktime32, _mktime64
time, _time32, _time64

_heapadd

Adds memory to the heap.

```
int _heapadd(
   void *memblock,
   size_t size
);
```

Parameters

memblock

Pointer to the heap memory.

size

Size of memory to add, in bytes.

Return Value

If successful, **_heapadd** returns 0; otherwise, the function returns –1 and sets **errno** to **ENOSYS**.

For more information about this and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Beginning with Visual C++ version 4.0, the underlying heap structure was moved to the C run-time libraries to support the new debugging features. As a result, **_heapadd** is no longer supported on any platform that is based on the Win32 API.

Requirements

Routine	Required header	Optional header	Compatibility
_heapadd	<malloc.h>	<errno.h>	None

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation
free
_heapchk
_heapmin
_heapset
_heapwalk
malloc
realloc

_heapchk

Runs consistency checks on the heap.

```
int _heapchk( void );
```

Return Value

_heapchk returns one of the following integer manifest constants defined in Malloc.h.

_HEAPBADBEGIN

Initial header information is bad or cannot be found.

_HEAPBADNODE

Bad node has been found or heap is damaged.

_HEAPBADPTR

Pointer into heap is not valid.

_HEAPEMPTY

Heap has not been initialized.

_HEAPOK

Heap appears to be consistent.

In addition, if an error occurs, **_heapchk** sets **errno** to **ENOSYS**.

Remarks

The **_heapchk** function helps debug heap-related problems by checking for minimal consistency of the heap. If the operating system does not support **_heapchk** (for example, Windows 98), the function returns **_HEAPOK** and sets **errno** to **ENOSYS**.

Requirements

Routine	Required header	Optional header	Compatibility
_heapchk	<malloc.h>	<errno.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_heapchk.c
// This program checks the heap for
// consistency and prints an appropriate message.

#include <malloc.h>
#include <stdio.h>

int main( void )
{
   int  heapstatus;
   char *buffer;

   // Allocate and deallocate some memory
   if( (buffer = (char *)malloc( 100 )) != NULL )
      free( buffer );

   // Check heap status
   heapstatus = _heapchk();
   switch( heapstatus )
```

```
    {
    case _HEAPOK:
       printf(" OK - heap is fine\n" );
       break;
    case _HEAPEMPTY:
       printf(" OK - heap is empty\n" );
       break;
    case _HEAPBADBEGIN:
       printf( "ERROR - bad start of heap\n" );
       break;
    case _HEAPBADNODE:
       printf( "ERROR - bad node in heap\n" );
       break;
    }
 }
```

Output

```
OK - heap is fine
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation
_heapadd
_heapmin
_heapset
_heapwalk

_heapmin

Releases unused heap memory to the operating system.

```
int _heapmin( void );
```

Return Value

If successful, **_heapmin** returns 0; otherwise, the function returns –1 and sets **errno** to **ENOSYS**.

For more information about this and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_heapmin** function minimizes the heap by releasing unused heap memory to the operating system. If the operating system does not support **_heapmin** (for example, Windows 98), the function returns –1 and sets **errno** to **ENOSYS**.

Requirements

Routine	Required header	Optional header	Compatibility
_heapmin	<malloc.h>	<errno.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation
free
_heapadd
_heapchk
_heapset
_heapwalk
malloc

_heapset

Checks heaps for minimal consistency and sets the free entries to a specified value.

```
int _heapset(
   unsigned int fill
);
```

Parameters

fill

Fill character.

Return Value

_heapset returns one of the following integer manifest constants defined in Malloc.h.

_HEAPBADBEGIN

Initial header information invalid or not found.

_HEAPBADNODE

Heap damaged or bad node found.

_HEAPEMPTY

Heap not initialized.

_HEAPOK

Heap appears to be consistent.

In addition, if an error occurs, **_heapset** sets **errno** to **ENOSYS**.

Remarks

The **_heapset** function shows free memory locations or nodes that have been unintentionally overwritten.

_heapset checks for minimal consistency on the heap and then sets each byte of the heap's free entries to the *fill* value. This known value shows which memory locations of the heap contain free nodes and which contain data that were unintentionally written to freed memory. If the operating system does not support **_heapset** (for example, Windows 98), the function returns **_HEAPOK** and sets **errno** to **ENOSYS**.

Requirements

Routine	Required header	Optional header	Compatibility
_heapset	<malloc.h>	<errno.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_heapset.c
// This program checks the heap and
// fills in free entries with the character 'Z'.


#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   int heapstatus;
```

```
    char *buffer;

    if( (buffer = malloc( 1 )) == NULL ) // Make sure heap is
       exit( 0 );                        //    initialized
    heapstatus = _heapset( 'Z' );        // Fill in free entries
    switch( heapstatus )
    {
    case _HEAPOK:
       printf( "OK - heap is fine\n" );
       break;
    case _HEAPEMPTY:
       printf( "OK - heap is empty\n" );
       break;
    case _HEAPBADBEGIN:
       printf( "ERROR - bad start of heap\n" );
       break;
    case _HEAPBADNODE:
       printf( "ERROR - bad node in heap\n" );
       break;
    }
    free( buffer );
  }
```

Output

```
OK - heap is fine
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation
_heapadd
_heapchk
_heapmin
_heapwalk

_heapwalk

Traverses the heap and returns information about the next entry.

```
int _heapwalk(
   _HEAPINFO *entryinfo
);
```

Parameters

entryinfo

Buffer to contain heap information.

Return Value

_heapwalk returns one of the following integer manifest constants defined in Malloc.h.

_HEAPBADBEGIN

Initial header information invalid or not found.

_HEAPBADNODE

Heap damaged or bad node found.

_HEAPBADPTR

_pentry field of the **_HEAPINFO** structure does not contain a valid pointer into the heap or *entryinfo* is a null pointer.

_HEAPEND

End of the heap reached successfully.

_HEAPEMPTY

Heap not initialized.

_HEAPOK

No errors so far; *entryinfo* is updated with information about the next heap entry.

In addition, if an error occurs, **_heapwalk** sets **errno** to **ENOSYS**.

Remarks

The **_heapwalk** function helps debug heap-related problems in programs. The function walks through the heap, traversing one entry per call, and returns a pointer to a structure of type **_HEAPINFO** that contains information about the next heap entry. The **_HEAPINFO** type, defined in Malloc.h, contains the following elements.

int *_pentry

Heap entry pointer.

size_t _size

Size of the heap entry.

int _useflag

Flag that indicates whether the heap entry is in use.

A call to **_heapwalk** that returns **_HEAPOK** stores the size of the entry in the **_size** field and sets the **_useflag** field to either **_FREEENTRY** or **_USEDENTRY** (both are constants defined in Malloc.h). To obtain this information about the first entry in the heap, pass **_heapwalk** a pointer to a **_HEAPINFO** structure whose **_pentry** member is **NULL**. If the operating system does not support **_heapwalk** (for example, Windows 98), the function returns **_HEAPEND** and sets **errno** to **ENOSYS**.

This function validates its parameter. If *entryinfo* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **_HEAPBADPTR**.

Requirements

Routine	Required header	Optional header	Compatibility
_heapwalk	<malloc.h>	<errno.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_heapwalk.c
// processor: x86
// (This program isn't really x86-specific, but on a
// 64-bit machine it might run for a *long* time.)

// This program "walks" the heap, starting
// at the beginning (_pentry = NULL). It prints out each
// heap entry's use, location, and size. It also prints
// out information about the overall state of the heap as
// soon as _heapwalk returns a value other than _HEAPOK.


#include <stdio.h>
#include <malloc.h>

void heapdump( void );

int main( void )
{
   char *buffer;

   heapdump();
   if( (buffer = (char *)malloc( 59 )) != NULL )
   {
      heapdump();
      free( buffer );
   }
   heapdump();
}

void heapdump( void )
{
   _HEAPINFO hinfo;
   int heapstatus;
   hinfo._pentry = NULL;
   while( ( heapstatus = _heapwalk( &hinfo ) ) == _HEAPOK )
   { printf( "%6s block at %Fp of size %4.4X\n",
        ( hinfo._useflag == _USEDENTRY ? "USED" : "FREE" ),
          hinfo._pentry, hinfo._size );
   }

   switch( heapstatus )
   {
   case _HEAPEMPTY:
      printf( "OK - empty heap\n" );
      break;
   case _HEAPEND:
      printf( "OK - end of heap\n" );
      break;
   case _HEAPBADPTR:
      printf( "ERROR - bad pointer to heap\n" );
      break;
   case _HEAPBADBEGIN:
      printf( "ERROR - bad start of heap\n" );
      break;
   case _HEAPBADNODE:
      printf( "ERROR - bad node in heap\n" );
      break;
```

```
    }
  }
```

Sample Output

```
  USED block at 00310650 of size 0100
  USED block at 00310758 of size 0800
  USED block at 00310F60 of size 0080
  FREE block at 00310FF0 of size 0398
  USED block at 00311390 of size 000D
  USED block at 003113A8 of size 00B4
  USED block at 00311468 of size 0034
  USED block at 003114A8 of size 0039
...
  USED block at 00312228 of size 0010
  USED block at 00312240 of size 1000
  FREE block at 00313250 of size 1DB0
OK - end of heap
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation
_heapadd
_heapchk
_heapmin
_heapset

hypot

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _hypot instead.

_hypot, hypotf

Calculates the hypotenuse.

```
double _hypot(
   double x,
   double y
);
float _hypotf(
   float x,
   float y
);
```

Parameters

x, y

Floating-point values.

Return Value

_hypot returns the length of the hypotenuse if successful or INF (infinity) on overflow. The **errno** variable is set to **ERANGE** on overflow. You can modify error handling with **_matherr**.

For more information about this and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Platforms

_hypotf is only available on Itanium Processor Family (IPF) platforms. **_hypot** is available on all platforms.

Remarks

The **_hypot** function calculates the length of the hypotenuse of a right triangle, given the length of the two sides *x* and *y* (in other words, the square root of $x^2 + y^2$).

Requirements

Routine	Required header	Compatibility
_hypot	<math.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
hypotf	<math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_hypot.c
// This program prints the hypotenuse of a right triangle.

#include <math.h>
#include <stdio.h>

int main( void )
{
   double x = 3.0, y = 4.0;

   printf( "If a right triangle has sides %2.1f and %2.1f, "
           "its hypotenuse is %2.1f\n", x, y, _hypot( x, y ) );
}
```

Output

```
If a right triangle has sides 3.0 and 4.0, its hypotenuse is 5.0
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support

_cabs

_matherr

inp, inpw

These POSIX functions are deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _inp, _inpw, _inpd instead.

_inp, _inpw, _inpd

Inputs a byte (**_inp**), a word (**_inpw**), or a double word (**_inpd**) from a port.

```
int _inp(
   unsigned short port
);
unsigned short _inpw(
   unsigned short port
);
unsigned long _inpd(
   unsigned short port
);
```

Parameters

port

I/O port number.

Return Value

The functions return the byte, word, or double word read from *port*. There is no error return.

Remarks

The **_inp**, **_inpw**, and **_inpd** functions read a byte, a word, and a double word, respectively, from the specified input port. The input value can be any unsigned short integer in the range 0 – 65,535.

Because these functions read directly from an I/O port, they might not be used in user code in Windows NT, Windows 2000, Windows XP, and Windows Server 2003. For information about using I/O ports in these operating systems, use the Win32 Communications API.

Requirements

Routine	Required header	Compatibility
_inp	<conio.h>	Windows 98, Windows Me, and kernel mode on Windows NT, Windows 2000, Windows XP, and Windows Server 2003
_inpw	<conio.h>	Windows 98, Windows Me, and kernel mode on Windows NT, Windows 2000, Windows XP, and Windows Server 2003
_inpd	<conio.h>	Windows 98, Windows Me, and kernel mode on Windows NT, Windows 2000, Windows XP, and Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Console and Port I/O

_outp, _outpw, _outpd

is, isw Routines

isalnum, iswalnum, _isalnum_l, _iswalnum_l	islower, iswlower, _islower_l, _iswlower_l
isalpha, iswalpha, _isalpha_l, _iswalpha_l	isprint, iswprint, _isprint_l, _iswprint_l
__isascii, iswascii	ispunct, iswpunct, _ispunct_l, _iswpunct_l
iscntrl, iswcntrl, _iscntrl_l, _iswcntrl_l	isspace, iswspace, _isspace_l, _iswspace_l
__iscsym, __iswcsym __iscsymf, __iswcsymf	isupper, _isupper_l, iswupper, _iswupper_l
isdigit, iswdigit, _isdigit_l, _iswdigit_l	_isctype, iswctype, _isctype_l, _iswctype_l
isgraph, iswgraph, _isgraph_l, _iswgraph_l	isxdigit, iswxdigit, _isxdigit_l, _iswxdigit_l
isleadbyte, _isleadbyte_l	

Remarks

These routines test characters for specified conditions.

The **is** routines produce meaningful results for any integer argument from –1 (**EOF**) to **UCHAR_MAX** (0xFF), inclusive. The expected argument type is **int**.

Caution
For the **is** routines, passing an argument of type **char** may yield unpredictable results. An SBCS or MBCS single-byte charact er of type **char** with a value greater than 0x7F is negative. If a **char** is passed, the compiler may convert the value to a signed **int** or a signed **long**. This value may be sign-extended by the compiler, with unexpected results.

The **isw** routines produce meaningful results for any integer value from – 1 (**WEOF**) to 0xFFFF, inclusive. The **wint_t** data type is defined in WCHAR.H as an **unsigned short**; it can hold any wide character or the wide-character end-of-file (**WEOF**) value.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

In the "C" locale, the test conditions for the **is** routines are as follows:

isalnum

Alphanumeric (A – Z, a – z, or 0 – 9).

isalpha

Alphabetic (A – Z or a – z).

__isascii

ASCII character (0x00 – 0x7F).

iscntrl

Control character (0x00 – 0x1F or 0x7F).

__iscsym

Letter, underscore, or digit.

__iscsymf

Letter or underscore.

isdigit

Decimal digit (0 – 9).

isgraph

Printable character except space ( ).

islower

Lowercase letter (a – z).

isprint

Printable character including space (0x20 – 0x7E).

ispunct

Punctuation character.

isspace

White-space character (0x09 – 0x0D or 0x20).

isupper

Uppercase letter (A – Z).

isxdigit

Hexadecimal digit (A – F, a – f, or 0 – 9).

For the **isw** routines, the result of the test for the specified condition is independent of locale. The test conditions for the **isw** functions are as follows:

iswalnum

iswalpha or **iswdigit**.

iswalpha

Any wide character that is one of an implementation-defined set for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is nonzero. **iswalpha** returns nonzero only for wide characters for which **iswupper** or **iswlower** is nonzero.

iswascii

Wide-character representation of ASCII character (0x0000 – 0x007F).

iswcntrl

Control wide character.

__iswcsym

Any wide character for which **isalnum** is true, or the '_' character.

__iswcsymf

Any wide character for which **iswalpha** is true, or the '_' character.

iswctype

Character has property specified by the *desc* argument. For each valid value of the *desc* argument of **iswctype**, there is an equivalent wide-character classification routine, as shown in the following table:

Equivalence of iswctype(*c, desc*) to Other isw Testing Routines

Value of *desc* argument	**iswctype(*c, desc*) equivalent**
_ALPHA	**iswalpha(** *c* **)**
_ALPHA | **_DIGIT**	**iswalnum(** *c* **)**

_CONTROL	**iswcntrl(** *c* **)**
_DIGIT	**iswdigit(** *c* **)**
_ALPHA | **_DIGIT** | **_PUNCT**	**iswgraph(** *c* **)**
_LOWER	**iswlower(** *c* **)**
_ALPHA | **_BLANK** | **_DIGIT** | **_PUNCT**	**iswprint(** *c* **)**
_PUNCT	**iswpunct(** *c* **)**
_SPACE	**iswspace(** *c* **)**
_UPPER	**iswupper(** *c* **)**
_HEX	**iswxdigit(** *c* **)**

iswdigit

Wide character corresponding to a decimal-digit character.

iswgraph

Printable wide character except space wide character (L' ').

iswlower

Lowercase letter, or one of implementation-defined set of wide characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is nonzero. **iswlower** returns nonzero only for wide characters that correspond to lowercase letters.

iswprint

Printable wide character, including space wide character (L' ').

iswpunct

Printable wide character that is neither space wide character (L' ') nor wide character for which **iswalnum** is nonzero.

iswspace

Wide character that corresponds to standard white-space character or is one of implementation-defined set of wide characters for which **iswalnum** is false. Standard white-space characters are: space (L' '), formfeed (L'\f'), newline (L'\n'), carriage return (L'\r'), horizontal tab (L'\t'), and vertical tab (L'\v').

iswupper

Wide character that is uppercase or is one of an implementation-defined set of wide characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is nonzero. **iswupper** returns nonzero only for wide characters that correspond to uppercase characters.

iswxdigit

Wide character that corresponds to a hexadecimal-digit character.

Example

```
// crt_isfam.c
/* This program tests all characters between 0x0
 * and 0x7F, then displays each character with abbreviations
 * for the character-type codes that apply.
 */

#include <stdio.h>
#include <ctype.h>
```

```
int main( void )
{
   int ch;
   for( ch = 0; ch <= 0x7F; ch++ )
   {
      printf( "%.2x  ", ch );
      printf( " %c", isprint( ch )  ? ch   : ' ' );
      printf( "%4s", isalnum( ch )  ? "AN" : "" );
      printf( "%3s", isalpha( ch )  ? "A"  : "" );
      printf( "%3s", __isascii( ch )  ? "AS" : "" );
      printf( "%3s", iscntrl( ch )  ? "C"  : "" );
      printf( "%3s", __iscsym( ch )  ? "CS "  : "" );
      printf( "%3s", __iscsymf( ch )  ? "CSF"  : "" );
      printf( "%3s", isdigit( ch )  ? "D"  : "" );
      printf( "%3s", isgraph( ch )  ? "G"  : "" );
      printf( "%3s", islower( ch )  ? "L"  : "" );
      printf( "%3s", ispunct( ch )  ? "PU" : "" );
      printf( "%3s", isspace( ch )  ? "S"  : "" );
      printf( "%3s", isprint( ch )  ? "PR" : "" );
      printf( "%3s", isupper( ch )  ? "U"  : "" );
      printf( "%3s", isxdigit( ch ) ? "X"  : "" );
      printf( ".\n" );
   }
}
```

Output

```
00            AS  C                                .
01            AS  C                                .
02            AS  C                                .
03            AS  C                                .
04            AS  C                                .
05            AS  C                                .
06            AS  C                                .
07            AS  C                                .
08            AS  C                                .
09            AS  C                    S           .
0a            AS  C                    S           .
0b            AS  C                    S           .
0c            AS  C                    S           .
0d            AS  C                    S           .
0e            AS  C                                .
0f            AS  C                                .
10            AS  C                                .
11            AS  C                                .
12            AS  C                                .
13            AS  C                                .
14            AS  C                                .
15            AS  C                                .
16            AS  C                                .
17            AS  C                                .
18            AS  C                                .
19            AS  C                                .
1a            AS  C                                .
1b            AS  C                                .
1c            AS  C                                .
1d            AS  C                                .
1e            AS  C                                .
1f            AS  C                                .
20            AS                           S  PR   .
21   !        AS              G    PU      PR      .
22   "        AS              G    PU      PR      .
23   #        AS              G    PU      PR      .
24   $        AS              G    PU      PR      .
25   %        AS              G    PU      PR      .
26   &        AS              G    PU      PR      .
27   '        AS              G    PU      PR      .
28   (        AS              G    PU      PR      .
29   )        AS              G    PU      PR      .
2a   *        AS              G    PU      PR      .
```

2b	+			AS				G		PU	PR	.
2c	,			AS				G		PU	PR	.
2d	-			AS				G		PU	PR	.
2e	.			AS				G		PU	PR	.
2f	/			AS				G		PU	PR	.
30	0	AN		AS	CS		D	G			PR	X.
31	1	AN		AS	CS		D	G			PR	X.
32	2	AN		AS	CS		D	G			PR	X.
33	3	AN		AS	CS		D	G			PR	X.
34	4	AN		AS	CS		D	G			PR	X.
35	5	AN		AS	CS		D	G			PR	X.
36	6	AN		AS	CS		D	G			PR	X.
37	7	AN		AS	CS		D	G			PR	X.
38	8	AN		AS	CS		D	G			PR	X.
39	9	AN		AS	CS		D	G			PR	X.
3a	:			AS				G		PU	PR	.
3b	;			AS				G		PU	PR	.
3c	<			AS				G		PU	PR	.
3d	=			AS				G		PU	PR	.
3e	>			AS				G		PU	PR	.
3f	?			AS				G		PU	PR	.
40	@			AS				G		PU	PR	.
41	A	AN	A	AS	CS	CSF		G			PR U	X.
42	B	AN	A	AS	CS	CSF		G			PR U	X.
43	C	AN	A	AS	CS	CSF		G			PR U	X.
44	D	AN	A	AS	CS	CSF		G			PR U	X.
45	E	AN	A	AS	CS	CSF		G			PR U	X.
46	F	AN	A	AS	CS	CSF		G			PR U	X.
47	G	AN	A	AS	CS	CSF		G			PR U	.
48	H	AN	A	AS	CS	CSF		G			PR U	.
49	I	AN	A	AS	CS	CSF		G			PR U	.
4a	J	AN	A	AS	CS	CSF		G			PR U	.
4b	K	AN	A	AS	CS	CSF		G			PR U	.
4c	L	AN	A	AS	CS	CSF		G			PR U	.
4d	M	AN	A	AS	CS	CSF		G			PR U	.
4e	N	AN	A	AS	CS	CSF		G			PR U	.
4f	O	AN	A	AS	CS	CSF		G			PR U	.
50	P	AN	A	AS	CS	CSF		G			PR U	.
51	Q	AN	A	AS	CS	CSF		G			PR U	.
52	R	AN	A	AS	CS	CSF		G			PR U	.
53	S	AN	A	AS	CS	CSF		G			PR U	.
54	T	AN	A	AS	CS	CSF		G			PR U	.
55	U	AN	A	AS	CS	CSF		G			PR U	.
56	V	AN	A	AS	CS	CSF		G			PR U	.
57	W	AN	A	AS	CS	CSF		G			PR U	.
58	X	AN	A	AS	CS	CSF		G			PR U	.
59	Y	AN	A	AS	CS	CSF		G			PR U	.
5a	Z	AN	A	AS	CS	CSF		G			PR U	.
5b	[			AS				G		PU	PR	.
5c	\			AS				G		PU	PR	.
5d	]			AS				G		PU	PR	.
5e	^			AS				G		PU	PR	.
5f	_			AS	CS	CSF		G		PU	PR	.
60	`			AS				G		PU	PR	.
61	a	AN	A	AS	CS	CSF		G	L		PR	X.
62	b	AN	A	AS	CS	CSF		G	L		PR	X.
63	c	AN	A	AS	CS	CSF		G	L		PR	X.
64	d	AN	A	AS	CS	CSF		G	L		PR	X.
65	e	AN	A	AS	CS	CSF		G	L		PR	X.
66	f	AN	A	AS	CS	CSF		G	L		PR	X.
67	g	AN	A	AS	CS	CSF		G	L		PR	.
68	h	AN	A	AS	CS	CSF		G	L		PR	.
69	i	AN	A	AS	CS	CSF		G	L		PR	.
6a	j	AN	A	AS	CS	CSF		G	L		PR	.
6b	k	AN	A	AS	CS	CSF		G	L		PR	.
6c	l	AN	A	AS	CS	CSF		G	L		PR	.
6d	m	AN	A	AS	CS	CSF		G	L		PR	.
6e	n	AN	A	AS	CS	CSF		G	L		PR	.
6f	o	AN	A	AS	CS	CSF		G	L		PR	.
70	p	AN	A	AS	CS	CSF		G	L		PR	.
71	q	AN	A	AS	CS	CSF		G	L		PR	.
72	r	AN	A	AS	CS	CSF		G	L		PR	.

73	s	AN	A	AS		CS	CSF		G	L		PR	.
74	t	AN	A	AS		CS	CSF		G	L		PR	.
75	u	AN	A	AS		CS	CSF		G	L		PR	.
76	v	AN	A	AS		CS	CSF		G	L		PR	.
77	w	AN	A	AS		CS	CSF		G	L		PR	.
78	x	AN	A	AS		CS	CSF		G	L		PR	.
79	y	AN	A	AS		CS	CSF		G	L		PR	.
7a	z	AN	A	AS		CS	CSF		G	L		PR	.
7b	{			AS					G		PU	PR	.
7c	|			AS					G		PU	PR	.
7d	}			AS					G		PU	PR	.
7e	~			AS					G		PU	PR	.
7f				AS	C								.

See Also

Reference

Character Classification
Locale
setlocale, _wsetlocale
Interpretation of Multibyte-Character Sequences
to Functions

isalpha, iswalpha, _isalpha_l, _iswalpha_l

Check an integer to see it is represents an alphabetic character in a character set.

```
int isalpha(
   int c
);
int iswalpha(
   wint_t c
);
int _isalpha_l(
   int c,
   _locale_t locale
);
int _iswalpha_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c

Integer to test.

locale

The locale to use instead of the current locale.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of an alphabetic character. **isalpha** returns a nonzero value if *c* is within the ranges A – Z or a – z. **iswalpha** returns a nonzero value only for wide characters for which **iswupper** or **iswlower** is nonzero, that is, for any wide character that is one of an implementation-defined set for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is nonzero. Each of these routines returns 0 if *c* does not satisfy the test condition.

When used with a debug CRT library, **isalpha** will display a CRT assert if passed a parameter that is not EOF or in the range of 0 through 0xFF. When used with a release CRT library, **isalpha** will use the parameter as an index into an array, with undefined results if the parameter is not EOF or in the range of 0 through 0xFF.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_istalpha	**isalpha**	**_ismbcalpha**	**iswalpha**
_istalpha_l	**_isalpha_l**	**_ismbcalpha_l**	**_iswalpha_l**

Requirements

Routine	Required header	Compatibility
isalpha	<ctype.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
iswalpha	<ctype.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_isalpha_l	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_iswalpha_l	<ctype.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::Char::IsLetter

See Also
Reference
Character Classification
Locale
is, isw Routines

isalnum, iswalnum, _isalnum_l, _iswalnum_l

Determines if a particular character is an alphanumeric character.

```
int isalnum(
   int c
);
int iswalnum(
   wint_t c
);
int _isalnum_l(
   int c,
   _locale_t locale
);
int _iswalnum_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c

Integer to test.

locale

The locale to use.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of an alphanumeric character. **isalnum** returns a nonzero value if either **isalpha** or **isdigit** is nonzero for *c*, that is, if *c* is within the ranges A – Z, a – z, or 0 – 9. **iswalnum** returns a nonzero value if either **iswalpha** or **iswdigit** is nonzero for *c*. Each of these routines returns 0 if *c* does not satisfy the test condition.

When used with a debug CRT library, **isalnum** will display a CRT assert if passed a parameter that is not EOF or in the range of 0 through 0xFF. **isalnum** will use the parameter as an index into an array, with undefined results if the parameter is not EOF or in the range of 0 through 0xFF.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_istalnum	**isalnum**	_ismbcalnum	**iswalnum**
_istalnum_l	**_isalnum_l**	**_ismbcalnum_l**	**_iswalnum_l**

Requirements

Routine	Required header	Compatibility
isalnum	<ctype.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
iswalnum	<ctype.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_isalnum_l	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_iswalnum_l	<ctype.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::Char::IsLetterOrDigit

See Also

Reference

Character Classification
Locale
is, isw Routines

isascii

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant __isascii instead.

isatty

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _isatty instead.

_isatty

Determines whether a file descriptor is associated with a character device.

```
int _isatty(
   int fd
);
```

Parameters

fd

File descriptor referring to the device to be tested.

Return Value

_isatty returns a nonzero value if the descriptor is associated with a character device. Otherwise, **_isatty** returns 0.

Remarks

The **_isatty** function determines whether *fd* is associated with a character device (a terminal, console, printer, or serial port).

This function validates the *fd* parameter. If *fd* is a bad file pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns 0 and sets **errno** to **EBADF**.

Requirements

Routine	Required header	Compatibility
_isatty	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_isatty.c
/* This program checks to see whether
 * stdout has been redirected to a file.
 */

#include <stdio.h>
#include <io.h>

int main( void )
{
   if( _isatty( _fileno( stdout ) ) )
      printf( "stdout has not been redirected to a file\n" );
   else
      printf( "stdout has been redirected to a file\n");
}
```

Sample Output

```
stdout has not been redirected to a file
```

.NET Framework Equivalent

- System::IO::Stream::CanWrite

- System::IO::FileStream::CanWrite

See Also

Reference

File Handling

iscntrl, iswcntrl, _iscntrl_l, _iswcntrl_l

Determine if an integer represents a control character.

```
int iscntrl(
   int c
);
int iswcntrl(
   wint_t c
);
int _iscntrl_l(
   int c,
   _locale_t locale
);
int _iswcntrl_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c

Integer to test

locale

The locale to use.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of a control character. **iscntrl** returns a nonzero value if *c* is a control character (0x00 – 0x1F or 0x7F). **iswcntrl** returns a nonzero value if *c* is a control wide character. Each of these routines returns 0 if *c* does not satisfy the test condition.

When used with a debug CRT library, **iscntrl** will display a CRT assert if passed a parameter that is not EOF or in the range of 0 through 0xFF. When used with a debug CRT library, **iscntrl** will use the parameter as an index into an array, with undefined results if the parameter is not EOF or in the range of 0 through 0xFF.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_istcntrl	**iscntrl**	**iscntrl**	**iswcntrl**
_istcntrl_l	**_iscntrl_l**	**_iscntrl_l**	**_iswcntrl_l**

Requirements

Routine	Required header	Compatibility
iscntrl	<ctype.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
iswcntrl	<ctype.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_iscntrl_l	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_iswcntrl_l	<ctype.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::Char::IsControl

See Also
Reference
Character Classification
Locale
is, isw Routines

_isctype, iswctype, _isctype_l, _iswctype_l

Tests *c* for the property specified by the *desc* argument. For each valid value of *desc*, there is an equivalent wide-character classification routine.

```
int _isctype(
   int c,
   _ctype_t desc
);
int _isctype_l(
   int c,
   _ctype_t desc,
   _locale_t locale
);
int iswctype(
   wint_t c,
   wctype_t desc
);
int _iswctype_l(
   wint_t c,
   wctype_t desc,
   _locale_t locale
);
```

Parameters

c

Integer to test.

desc

Property to test for. This is normally retrieved using ctype or wctype.

locale

The locale to use for any locale-dependent tests.

Return Value

_isctype and **iswctype** return a nonzero value if *c* has the property specified by *desc* in the current locale or 0 if it does not. The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
n/a	**_isctype**	**n/a**	**_iswctype**
n/a	**_isctype_l**	**n/a**	**_iswctype_l**

Requirements

Routine	Required header	Compatibility
_isctype	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
iswctype	<ctype.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_isctype_l	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_iswctype_l	<ctype.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Character Classification
Locale
is, isw Routines

iscsym, iscsymf

These POSIX functions are deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant __iscsym, __iswcsym, __iscsymf, __iswcsymf, _iscsym_l, _iswcsym_l, _iscsymf_l, _iswcsymf_l instead.

__iscsym, __iswcsym, __iscsymf, __iswcsymf, _iscsym_l, _iswcsym_l, _iscsymf_l, _iswcsymf_l

Determine if an integer represents a character that may be used in an identifier.

```
int __iscsym(
   int c
);
int __iswcsym(
   wint_t c
);
int __iscsymf(
   int c
);
int __iswcsymf(
   wint_t c
);
int _iscsym_l(
   int c,
   _locale_t locale
);
int _iswcsym_l(
   wint_t c,
   _locale_t locale
);
int _iscsymf_l(
   int c,
   _locale_t locale
);
int _iswcsymf_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c

Integer to test. *c* should be in the range of 0-255 for the narrow character version of the function.

locale

The locale to use.

Return Value

__iscsym returns a nonzero value if *c* is a letter, underscore, or digit. **__iscsymf** returns a nonzero value if *c* is a letter or an underscore. Each of these routines returns 0 if *c* does not satisfy the test condition. Both of these routines are macros, so be careful using expressions with side effects within the argument list; the arguments will be evaluated more than once.

The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior.

In Visual C++ 2005, the wide character versions of these macros are available.

The following table shows the equivalent expressions for each of these macros:

Macro	Equivalent
`__iscsym(c)`	`(isalnum(c) || ((c) == '_'))`
`__iswcsym(c)`	`(iswalnum(c) || ((c) == '_'))`
`__iscsymf(c)`	`(isalpha(c) || ((c) == '_'))`

`__iswcsymf(c)`	`(iswalpha(c) || ((c) == '_'))`

Requirements

Routine	Required header	Compatibility
__iscsym	<ctype.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
__iswcsym	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
__iscsymf	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
__iswcsymf	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_iscsym_l	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_iswcsym_l	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_iscsymf_l	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_iswcsymf_l	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::Char::IsSymbol

See Also

Reference

Character Classification
Locale
is, isw Routines

isdigit, iswdigit, _isdigit_l, _iswdigit_l

Determines if a particular character is a decimal-digit character.

```
int isdigit(
   int c
);
int iswdigit(
   wint_t c
);
int _isdigit_l(
   int c,
   _locale_t locale
);
int _iswdigit_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c

Integer to test.

locale

The locale to use.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of a decimal-digit character. **isdigit** returns a nonzero value if *c* is a decimal digit (0 – 9). **iswdigit** returns a nonzero value if *c* is a wide character corresponding to a decimal-digit character. Each of these routines returns 0 if *c* does not satisfy the test condition.

When used with a debug CRT library, **isdigit** will display a CRT assert if passed a parameter that is not EOF or in the range of 0 through 0xFF. When used with a debug CRT library, **isdigit** will use the parameter as an index into an array, with undefined results if the parameter is not EOF or in the range of 0 through 0xFF.

The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_istdigit	**isdigit**	_ismbcdigit	**iswdigit**
_istdigit_l	**_isdigit_l**	_ismbcdigit_l	**_iswdigit_l**

Requirements

Routine	Required header	Compatibility
isdigit	<ctype.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
iswdigit	<ctype.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_isdigit_l	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_isw digit _l	<ctype.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::Char::IsDigit

See Also

Reference

Character Classification
Locale
is, isw Routines

isgraph, iswgraph, _isgraph_l, _iswgraph_l

Determine if an integer represents a graphical character.

```
int isgraph(
   int c
);
int iswgraph(
   wint_t c
);
int _isgraph_l(
   int c,
   _locale_t locale
);
int _iswgraph_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c

Integer to test.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of a printable character other than a space. **isgraph** returns a nonzero value if *c* is a printable character other than a space. **iswgraph** returns a nonzero value if *c* is a printable wide character other than a wide character space. Each of these routines returns 0 if *c* does not satisfy the test condition.

When used with a debug CRT library, **isgraph** will display a CRT assert if passed a parameter that is not EOF or in the range of 0 through 0xFF. When used with a debug CRT library, **isgraph** will use the parameter as an index into an array, with undefined results if the parameter is not EOF or in the range of 0 through 0xFF.

The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_istgraph	**isgraph**	_ismbcgraph	**iswgraph**
_istgraph_l	**_isgraph_l**	_ismbcgraph_l	**_iswgraph_l**

Requirements

Routine	Required header	Compatibility
isgraph	<ctype.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
iswgraph	<ctype.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_isgraph_l	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_iswgraph_l	<ctype.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Character Classification
Locale
is, isw Routines

isleadbyte, _isleadbyte_l

Determine if a character is the lead byte of a multibyte character.

```
int isleadbyte(
   int c
);
int _isleadbyte_l(
   int c
);
```

Parameters

c

Integer to test.

Return Value

isleadbyte returns a nonzero value if the argument satisfies the test condition or 0 if it does not. In the "C" locale and in single-byte character set (SBCS) locales, **isleadbyte** always returns 0.

Remarks

The **isleadbyte** macro returns a nonzero value if its argument is the first byte of a multibyte character. **isleadbyte** produces a meaningful result for any integer argument from –1 (**EOF**) to **UCHAR_MAX** (0xFF), inclusive.

The expected argument type of **isleadbyte** is **int**; if a signed character is passed, the compiler may convert it to an integer by sign extension, yielding unpredictable results.

The version of this function with the **_l** suffix is identical except that it uses the locale passed in instead of the current locale for its locale-dependent behavior.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_istleadbyte	Always returns false	**_ isleadbyte**	Always returns false

Requirements

Routine	Required header	Compatibility
isleadbyte	<ctype.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_isleadbyte_l	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable, but see System::Globalization::CultureInfo.

See Also

Reference

Byte Classification
Locale
_ismbb Routines

islower, iswlower, _islower_l, _iswlower_l

Determines if a particular character is in lower case.

```
int islower(
   int c
);
int iswlower(
   wint_t c
);
int islower_l(
   int c,
   _locale_t locale
);
int _iswlower_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c

Integer to test.

locale

Locale to use.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of a lowercase character. **islower** returns a nonzero value if *c* is a lowercase character (a – z). **iswlower** returns a nonzero value if *c* is a wide character that corresponds to a lowercase letter, or if *c* is one of an implementation-defined set of wide characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is nonzero. Each of these routines returns 0 if *c* does not satisfy the test condition.

When used with a debug CRT library, **islower** will display a CRT assert if passed a parameter that is not EOF or in the range of 0 through 0xFF. When used with a debug CRT library, **islower** will use the parameter as an index into an array, with undefined results if the parameter is not EOF or in the range of 0 through 0xFF.

The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_istlower	**islower**	_ismbclower	**iswlower**
_istlower_l	**_islower _l**	_ismbclower_l	**_liswlower_l**

Requirements

Routine	Required header	Compatibility
islower	<ctype.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
iswlower	<ctype.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_islower_l	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_swlower_l	<ctype.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::Char::IsLower

See Also

Reference

Character Classification
Locale
is, isw Routines

_ismbb Routines

Each routine in the **_ismbb** family tests the given integer value *c* for a particular condition, using the current locale or a specified LC_CTYPE conversion state category.

_ismbbalnum, _ismbbalnum_l	_ismbbkpunct, _ismbbkpunct_l
_ismbbalpha, _ismbbalpha_l	_ismbblead, _ismbblead_l
_ismbbgraph, _ismbbgraph_l	_ismbbprint, _ismbbprint_l
_ismbbkalnum, _ismbbkalnum_l	_ismbbpunct, _ismbbpunct_l
_ismbbkana, _ismbbkana_l	_ismbbtrail, _ismbbtrail_l
_ismbbkprint, _ismbbkprint_l	

Remarks

Each routine in the **_ismbb** family tests the given integer value *c* for a particular condition. The test result depends on the multibyte code page in effect. By default, the multibyte code page is set to the system-default ANSI code page obtained from the operating system at program startup. You can query or change the multibyte code page in use with _getmbcp or _setmbcp, respectively.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

The routines in the **_ismbb** family test the given integer *c* as follows.

Routine	Byte test condition
_ismbbalnum	**isalnum** || **_ismbbkalnum**.
_ismbbalpha	**isalpha** || **_ismbbkalnum**.
_ismbbgraph	Same as **_ismbbprint**, but **_ismbbgraph** does not include the space character (0x20).
_ismbbkalnum	Non-ASCII text symbol other than punctuation. For example, in code page 932 only, **_ismbbkalnum** tests for k atakana alphanumeric.
_ismbbkana	Katakana (0xA1 – 0xDF). Specific to code page 932.
_ismbbkprint	Non-ASCII text or non-ASCII punctuation symbol. For example, in code page 932 only, **_ismbbkprint** tests for katakana alphanumeric or katakana punctuation (range: 0xA1 – 0xDF).
_ismbbkpunct	Non-ASCII punctuation. For example, in code page 932 only, **_ismbbkpunct** tests for katakana punctuation.
_ismbblead	First byte of multibyte character. For example, in code page 932 only, valid ranges are 0x81 – 0x9F, 0xE0 – 0xFC .
_ismbbprint	**isprint** || **_ismbbkprint**. **ismbbprint** includes the space character (0x20).
_ismbbpunct	**ispunct** || **_ismbbkpunct**.
_ismbbtrail	Second byte of multibyte character. For example, in code page 932 only, valid ranges are 0x40 – 0x7E, 0x80 – 0 xEC.

The following table shows the ORed values that compose the test conditions for these routines. The manifest constants **_BLANK**, **_DIGIT**, **_LOWER**, **_PUNCT**, and **_UPPER** are defined in Ctype.h.

Routine	_BLANK	_DIGIT	LOWER	_PUNCT	UPPER	Non- ASCII text	Non- ASCII punct
_ismbbalnum	—	x	x	—	x	x	—
_ismbbalpha	—	—	x	—	x	x	—
_ismbbgraph	—	x	x	x	x	x	x
_ismbbkalnum	—	—	—	—	—	x	—
_ismbbkprint	—	—	—	—	—	x	x
_ismbbkpunct	—	—	—	—	—	—	x
_ismbbprint	x	x	x	x	x	x	x
_ismbbpunct	—	—	—	x	—	—	x

The **_ismbb** routines are implemented both as functions and as macros. For more information about choosing either implementation, see Choosing Between Functions and Macros.

.NET Framework Equivalent

Not applicable, but see System::Globalization::CultureInfo.

See Also

Reference

Byte Classification
is, isw Routines
_mbbtombc, _mbbtombc_l
_mbctombb, _mbctombb_l

_ismbbalnum, _ismbbalnum_l

Determines whether a particular multibyte character is alpha or numeric.

```
int _ismbbalnum(
   unsigned int c
);
int _ismbbalnum_l(
   unsigned int c
);
```

Parameters

c

Integer to be tested.

locale

Locale to use.

Return Value

_ismbbalnum returns a nonzero value if the expression:

```
isalnum || _ismbbkalnum
```

is nonzero for *c*, or 0 if it is not.

The version of this function with the **_l** suffix is identical except that it uses the locale passed in instead of the current locale for its locale-dependent behavior.

Requirements

Routine	Required header	Compatibility
_ismbbalnum	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbbalnum_l	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Byte Classification

_ismbb Routines

_ismbbalpha, _ismbbalpha_l

Determines whether a particular multibyte character is alpha.

```
int _ismbbalpha(
   unsigned int c
);
int _ismbbalpha_l(
   unsigned int c
);
```

Parameters

c

Integer to be tested.

locale

Locale to use.

Return Value

_ismbbalpha returns a nonzero value if the expression:

```
isalpha || _ismbbkalnum
```

is nonzero for *c*, or 0 if it is not. **_ismbbalpha** uses the current locale for any locale-dependent character settings. **_ismbbalpha_l** is identical except that it uses the locale passed in.

Requirements

Routine	Required header	Compatibility
_ismbbalpha	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbbalpha_l	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Byte Classification

_ismbb Routines

_ismbbgraph, _ismbbgraph_l

Determines whether a particular multibyte character is a graphical character.

```
int _ismbbgraph (
   unsigned int c
);
int _ismbbgraph_l (
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Integer to be tested.

locale

Locale to use.

Return Value

Returns a nonzero value if the expression:

```
( _PUNCT | _UPPER | _LOWER | _DIGIT ) || _ismbbkprint
```

is nonzero for *c*, or 0 if it is not. **_ismbbgraph** uses the current locale for any locale-dependent behavior. **_ismbbgraph_l** is identical except that it uses the locale passed in instead.

Requirements

Routine	Required header	Compatibility
_ismbbgraph	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbbgraph_l	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Byte Classification

_ismbb Routines

_ismbbkalnum, _ismbbkalnum_l

Determines whether a particular multibyte character is a non-ASCII text symbol.

```
int _ismbbkalnum(
   unsigned int c
);
int _ismbbkalnum_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Integer to be tested.

locale

Locale to use.

Return Value

_ismbbkalnum returns a nonzero value if the integer *c* is a non-ASCII text symbol other than punctuation or 0 if it is not. **_ismbbkalnum** uses the current locale for locale-dependent character information. **_ismbbkalnum_l** is identical to **_ismbbkalnum** except that it takes the locale as a parameter.

Requirements

Routine	Required header	Compatibility
_ismbbkalnum	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbbkalnum_l	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Byte Classification

_ismbb Routines

_ismbbkana, _ismbbkana_l

Tests for a katakana symbol and is specific to code page 932.

```
int _ismbbkana(
   unsigned int c
);
int _ismbbkana_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Integer to be tested.

locale

Locale to use.

Return Value

_ismbbkana returns a nonzero value if the integer *c* is a katakana symbol or 0 if it is not. **_ismbbkana** uses the current locale for locale-dependent character information. **_ismbbkana_l** is identical except that it uses the locale object passed in.

Requirements

Routine	Required header	Compatibility
_ismbbkana	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbbkana_l	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Byte Classification

_ismbb Routines

_ismbbkprint, _ismbbkprint_l

Determines whether a particular multibyte character is a punctuation symbol.

```
int _ismbbkprint(
   unsigned int c
);
int _ismbbkprint_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Integer to be tested.

locale

Locale to use.

Return Value

_ismbbkprint returns a nonzero value if the integer *c* is a non-ASCII text or non-ASCII punctuation symbol or 0 if it is not. For example, in code page 932 only, **_ismbbkprint** tests for katakana alphanumeric or katakana punctuation (range: 0xA1 – 0xDF). **_ismbbkprint** uses the current locale for locale-dependent character settings. **_ismbbkprint_l** is identical except that it uses the locale passed in.

Requirements

Routine	Required header	Compatibility
_ismbbkprint	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003, Windows Server 2003
_ismbbkprint_l	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Byte Classification

_ismbb Routines

_ismbbkpunct, _ismbbkpunct_l

```
int _ismbbkpunct(
   unsigned int c
);
int _ismbbkpunct_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Integer to be tested.

locale

Locale to use.

Return Value

_ismbbkpunct returns a nonzero value if the integer *c* is a non-ASCII punctuation symbol, or 0 if it is not. For example, in code page 932 only, **_ismbbkpunct** tests for katakana punctuation. **_ismbbkpunct** uses the current locale for any locale-dependent character settings. **_ismbbkpunct_l** is identical except that it uses the locale passed in.

Requirements

Routine	Required header	Compatibility
_ismbbkpunct	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbbkpunct_l	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

See Also

Reference

Byte Classification

_ismbb Routines

_ismbblead, _ismbblead_l

Tests a character to determine whether it is a lead byte of a multibyte character.

```
int _ismbblead(
   unsigned int c
);
int _ismbblead_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Integer to be tested.

locale

Locale to use.

Return Value

Returns a nonzero value if the integer *c* is the first byte of a multibyte character.

Remarks

Multibyte characters consist of a lead byte followed by a trailing byte. Lead bytes are distinguished by being in a particular range for a given character set. For example, in code page 932 only, lead bytes range from 0x81 – 0x9F and 0xE0 – 0xFC.

_ismbblead uses the current locale for locale-dependent behavior. **_ismbblead_l** is identical except that it uses the locale passed in instead.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_istlead	Always returns false	**_ismbblead**	Always returns false

Requirements

Routine	Required header	Optional header	Compatibility
_ismbblead	<mbctype.h> or <mbstring.h>	<ctype.h>,* <limits.h>, <stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbblead_l	<mbctype.h> or <mbstring.h>	<ctype.h>,* <limits.h>, <stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

* For manifest constants for the test conditions.

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Byte Classification

_ismbb Routines

_ismbbprint, _ismbbprint_l

Determines whether a particular multibyte character is a print character.

```
int _ismbbprint(
   unsigned int c
);
int _ismbbprint_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Integer to be tested.

locale

Locale to use.

Return Value

_ismbbprint returns a nonzero value if the expression:

```
isprint || _ismbbkprint
```

is nonzero for *c*, or 0 if it is not. **_ismbbprint** uses the current locale for any locale-dependent behavior. **_ismbbprint_l** is identical except that it uses the locale passed in instead.

Requirements

Routine	Required header	Compatibility
_ismbbprint	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbbprint_l	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Byte Classification

_ismbb Routines

_ismbbpunct, _ismbbpunct_l

Determines whether a particular character is a punctuation character.

```
int _ismbbpunct(
   unsigned int c
);
int _ismbbpunct_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Integer to be tested.

locale

Locale to use.

Return Value

_ismbbpunct returns a nonzero value if the integer *c* is a non-ASCII punctuation symbol. **_ismbbpunct** uses the current locale for any locale-dependent character settings. **_ismbbpunct_l** is identical except that it uses the locale passed in.

Requirements

Routine	Required header	Compatibility
_ismbbpunct	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbbpunct_l	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Byte Classification

_ismbb Routines

_ismbbtrail, _ismbbtrail_l

Determines whether a byte is a trailing byte of a multibyte character.

```
int _ismbbtrail(
   unsigned int c
);
int _ismbbtrail_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Integer to be tested.

locale

Locale to use.

Return Value

_ismbbtrail returns a nonzero value if the integer *c* is the second byte of a multibyte character. For example, in code page 932 only, valid ranges are 0x40 to 0x7E and 0x80 to 0xFC.

Remarks

_ismbbtrail uses the current locale for locale-dependent behavior. **_ismbbtrail_l** is identical except that it uses the locale passed in instead.

Requirements

Routine	Required header	Optional header	Compatibility
_ismbbtrail	<mbctype.h> or <mbstring.h>	<ctype.h>,* <limits.h>, <stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbbtrail_l	<mbctype.h> or <mbstring.h>	<ctype.h>,* <limits.h>, <stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

* For manifest constants for the test conditions.

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Byte Classification

_ismbb Routines

_ismbc Routines

Each **_ismbc** routine tests a given multibyte character *c* for a particular condition.

_ismbcalnum, _ismbcalnum_l, _ismbcalpha, _ismbcalpha_, _ismbcdigit, _ismbcdigit_l	_ismbcl0, _ismbcl0_l, _ismbcl1, _ismbcl1_l, _ismbcl2, _ismbcl2_l
_ismbcgraph, _ismbcgraph_l, _ismbcprint, _ismbcprint_l, _ismbcpunct, _ismbcpunct_l, _ismbcspace, _ismbcspace_l	_ismbclegal, _ismbclegal_l, _ismbcsymbol, _ismbcsymbol_l
_ismbchira, _ismbchira_l, _ismbckata, _ismbckata_l	_ismbclower, _ismbclower_l, _ismbcupper, _ismbcupper_l

Remarks

The test result of each **_ismbc** routine depends on the multibyte code page in effect. Multibyte code pages have single-byte alphabetic characters. By default, the multibyte code page is set to the system-default ANSI code page obtained from the operating system at program startup. You can query or change the multibyte code page in use with _getmbcp or _setmbcp, respectively.

The output value is affected by the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Routine	**Test condition**	**Code page 932 example**
_ismbcalnum, _ismbcalnum_l, _ismbcalpha, _ismbcalpha_, _ismbcdigit, _ismbcdigit_l	Alphanumeric	Returns nonzero if and only if *c* is a single-byte representation of an ASCII English letter: See examples for **_ismbcdigit** and **_ismbcalpha**.
_ismbcalnum, _ismbcalnum_l, _ismbcalpha, _ismbcalpha_, _ismbcdigit, _ismbcdigit_l	Alphabetic	Returns nonzero if and only if *c* is a single-byte representation of an ASCII English letter: See examples for **_ismbcupper** and **_ismbclower**; or a katakana letter: 0xA6<=*c*<=0xDF.
_ismbcalnum, _ismbcalnum_l, _ismbcalpha, _ismbcalpha_, _ismbcdigit, _ismbcdigit_l	Digit	Returns nonzero if and only if *c* is a single-byte representation of an ASCII digit: 0x30<=*c*<=0x39.
_ismbcgraph, _ismbcgraph_l, _ismbcprint, _ismbcprint_l, _ismbcpunct, _ismbcpunct_l, _ismbcspace, _ismbcspace_l	Graphic	Returns nonzero if and only if *c* is a single-byte representation of any ASCII or katakana printable character except a white space ( ). See examples for **_ismbcdigit**, **_ismbcalpha**, and **_ismbcpunct**.
_ismbclegal, _ismbclegal_l, _ismbcsymbol, _ismbcsymbol_l	Valid multibyte character	Returns nonzero if and only if the first byte of *c* is within ranges 0x81 – 0x9F or 0xE0 – 0xFC, while the second byte is within ranges 0x40 – 0x7E or 0x80 - FC.
_ismbclower, _ismbclower_l, _ismbcupper, _ismbcupper_l	Lowercase alphabetic	Returns nonzero if and only if *c* is a single-byte representation of an ASCII lowercase English letter: 0x61<=*c*<=0x7A.
_ismbcgraph, _ismbcgraph_l, _ismbcprint, _ismbcprint_l, _ismbcpunct, _ismbcpunct_l, _ismbcspace, _ismbcspace_l	Printable	Returns nonzero if and only if *c* is a single-byte representation of any ASCII or katakana printable character including a white space ( ): See examples for **_ismbcspace**, **_ismbcdigit**, **_ismbcalpha**, and **_ismbcpunct**.
_ismbcgraph, _ismbcgraph_l, _ismbcprint, _ismbcprint_l, _ismbcpunct, _ismbcpunct_l, _ismbcspace, _ismbcspace_l	Punctuation	Returns nonzero if and only if *c* is a single-byte representation of any ASCII or katakana punctuation character.
_ismbcgraph, _ismbcgraph_l, _ismbcprint, _ismbcprint_l, _ismbcpunct, _ismbcpunct_l, _ismbcspace, _ismbcspace_l	White space	Returns nonzero if and only if *c* is a white space character: *c*=0x20 or 0x09<=*c*<=0x0D.
_ismbclegal, _ismbclegal_l, _ismbcsymbol, _ismbcsymbol_l	Multibyte symbol	Returns nonzero if and only if 0x8141<=*c*<=0x81AC.
_ismbclower, _ismbclower_l, _ismbcupper, _ismbcupper_l	Uppercase alphabetic	Returns nonzero if and only if *c* is a single-byte representation of an ASCII uppercase English letter: 0x41<=*c*<=0x5A.

Code Page 932 Specific

The following routines are specific to code page 932.

Routine	**Test condition (code page 932 only)**
_ismbchira, _ismbchira_l, _ismbckata, _ismbckata_l	Double-byte Hiragana: 0x829F<=*c*<=0x82F1.

_ismbchira, _ismbchira_l, _ismbckata, _ismbckata_l	Double-byte katakana: 0x8340<=*c*<=0x8396.
_ismbcl0, _ismbcl0_l, _ismbcl1, _ismbcl1_l, _ismbcl2, _ismbcl2_l	JIS non-Kanji: 0x8140<=*c*<=0x889E.
_ismbcl0, _ismbcl0_l, _ismbcl1, _ismbcl1_l, _ismbcl2, _ismbcl2_l	JIS level-1: 0x889F<=*c*<=0x9872.
_ismbcl0, _ismbcl0_l, _ismbcl1, _ismbcl1_l, _ismbcl2, _ismbcl2_l	JIS level-2: 0x989F<=*c*<=0xEA9E.

_ismbcl0, **_ismbcl1**, and **_ismbcl2** check that the specified value *c* matches the test conditions described in the preceding table, but do not check that *c* is a valid multibyte character. If the lower byte is in the ranges 0x00 – 0x3F, 0x7F, or 0xFD – 0xFF, these functions return a nonzero value, indicating that the character satisfies the test condition. Use _ismbbtrail, _ismbbtrail_l to test whether the multibyte character is defined.

END Code Page 932 Specific

See Also
Reference
Character Classification
is, isw Routines
_ismbb Routines

_ismbcalnum, _ismbcalnum_l, _ismbcalpha, _ismbcalpha_, _ismbcdigit, _ismbcdigit_l

Checks whether a multibyte character is an alphanumeric, alpha, or digit character.

```
int _ismbcalnum
(
   unsigned int c
);
int _ismbcalnum_l
(
   unsigned int c,
   _locale_t locale
);
int _ismbcalpha
(
   unsigned int c
);
int _ismbcalpha_l
(
   unsigned int c,
   _locale_t locale
);
int _ismbcdigit
(
   unsigned int c
);
int _ismbcdigit_l
(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Character to be tested.

locale

Locale to use.

Return Value

Each of these routines returns a nonzero value if the character satisfies the test condition or 0 if it does not. If *c* <= 255 and there is a corresponding **_ismbb** routine (for example, **_ismbcalnum** corresponds to **_ismbbalnum**), the result is the return value of the corresponding **_ismbb** routine.

Remarks

Each of these routines tests a given multibyte character for a given condition.

The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior.

Routine	Test condition	Code page 932 example
_ismbcalnum,_ismbcalnum_l	Alphanumeric	Returns nonzero if and only if *c* is a single-byte representation of an ASCII English letter: See examples for **_ismbcdigit** and **_ismbcalpha**.
_ismbcalpha,_ismbcalpha_l	Alphabetic	Returns nonzero if and only if *c* is a single-byte representation of an ASCII English letter: 0x41<=*c*<=0x5A or 0x61<=*c*<=0x7A; or a katakana letter: 0xA6<=*c*<=0xDF.

_ismbcdigit,_ismbcdigit	Digit	Returns nonzero if and only if *c* is a single-byte representation of an ASCII digit: 0x30<=*c*<=0x39.

Requirements

Routine	Required header	Compatibility
_ismbcalnum, _ismbcalnum_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbcalpha,_ismbcalpha_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbcdigit,_ismbcdigit_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

- System::Char::IsLetter
- System::Char::IsDigit
- For **_ismbcalnum**: Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Character Classification
_ismbc Routines
is, isw Routines
_ismbb Routines

_ismbcgraph, _ismbcgraph_l, _ismbcprint, _ismbcprint_l, _ismbcpunct, _ismbcpunct_l, _ismbcspace, _ismbcspace_l

Determines whether character is a graphical character, a display character, a punctuation character, or a space character.

```
int _ismbcgraph(
   unsigned int c
);
int _ismbcgraph_l(
   unsigned int c,
   _locale_t locale
);
int _ismbcprint(
   unsigned int c
);
int _ismbcprint_l(
   unsigned int c,
   _locale_t locale
);
int _ismbcpunct(
   unsigned int c
);
int _ismbcpunct_l(
   unsigned int c,
   _locale_t locale
);
int _ismbcspace(
   unsigned int c
);
int _ismbcspace_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Character to be determined.

locale

Locale to use.

Return Value

Each of these routines returns a nonzero value if the character satisfies the test condition or 0 if it does not. If *c* <= 255 and there is a corresponding **_ismbb** routine (for example, **_ismbcalnum** corresponds to **_ismbbalnum**), the result is the return value of the corresponding **_ismbb** routine.

The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior.

Remarks

Each of these functions tests a given multibyte character for a given condition.

Routine	Test condition	Code page 932 example
_ismbcgraph	Graphic	Returns nonzero if and only if *c* is a single-byte representation of any ASCII or katakana printable character except a white space ( ).

_ismbcprint	Printable	Returns nonzero if and only if *c* is a single-byte representation of any ASCII or katakana printable character including a white space ( ).
_ismbcpunct	Punctuation	Returns nonzero if and only if *c* is a single-byte representation of any ASCII or katakana punctuation character.
_ismbcspace	White space	Returns nonzero if and only if *c* is a white-space character: *c*=0x20 or 0x09<=*c*<=0x0D.

Requirements

Routine	Required header	Compatibility
_ismbcgraph	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbcgraph_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbcprint	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbcprint_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbcpunct	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbcpunct_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbcspace	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbcspace_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

- System::Char::IsPunctuation
- System::Char::IsWhiteSpace
- For **_ismbcgraph** and **_ismbcprint**: Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Character Classification
Locale
Interpretation of Multibyte-Character Sequences
_ismbc Routines
is, isw Routines
_ismbb Routines

_ismbchira, _ismbchira_l, _ismbckata, _ismbckata_l

Code Page 932 Specific functions.

```
int _ismbchira(
   unsigned int c
);
int _ismbchira_l(
   unsigned int c,
   _locale_t locale
);
int _ismbckata(
   unsigned int c
);
int _ismbckata_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Character to be tested.

locale

Locale to use.

Return Value

Each of these routines returns a nonzero value if the character satisfies the test condition or 0 if it does not. If *c* <= 255 and there is a corresponding **_ismbb** routine (for example, **_ismbcalnum** corresponds to **_ismbbalnum**), the result is the return value of the corresponding **_ismbb** routine.

Remarks

Each of these functions tests a given multibyte character for a given condition.

The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior.

Routine	Test condition (code page 932 only)
_ismbchira	Double-byte Hiragana: 0x829F<=*c*<=0x82F1.
_ismbchira_l	Double-byte Hiragana: 0x829F<=*c*<=0x82F1.
_ismbckata	Double-byte katakana: 0x8340<=*c*<=0x8396.
_ismbckata_l	Double-byte katakana: 0x8340<=*c*<=0x8396.

End Code Page 932 Specific

Requirements

Routine	Required header	Compatibility
_ismbchira	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbchira_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_ismbckata	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbckata_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Character Classification
_ismbc Routines
is, isw Routines
Locale
Interpretation of Multibyte-Character Sequences

_ismbcl0, _ismbcl0_l, _ismbcl1, _ismbcl1_l, _ismbcl2, _ismbcl2_l

Code Page 932 Specific functions, using the current locale or a specified LC_CTYPE conversion state category.

```
int _ismbcl0(
   unsigned int c
);
int _ismbcl0_l(
   unsigned int c,
   _locale_t locale
);
int _ismbcl1(
   unsigned int c
);
int _ismbcl1_l(
   unsigned int c ,
   _locale_t locale
);
int _ismbcl2(
   unsigned int c
);
int _ismbcl2_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Character to be tested.

locale

Locale to use.

Return Value

Each of these routines returns a nonzero value if the character satisfies the test condition or 0 if it does not. If *c* <= 255 and there is a corresponding **_ismbb** routine (for example, **_ismbcalnum** corresponds to **_ismbbalnum**), the result is the return value of the corresponding **_ismbb** routine.

Remarks

Each of these functions tests a given multibyte character for a given condition.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Routine	Test condition (code page 932 only)
_ismbcl0	JIS non-Kanji: 0x8140<=*c*<=0x889E.
_ismbcl0_l	JIS non-Kanji: 0x8140<=*c*<=0x889E.
_ismbcl1	JIS level-1: 0x889F<=*c*<=0x9872.
_ismbcl1_l	JIS level-1: 0x889F<=*c*<=0x9872.
_ismbcl2	JIS level-2: 0x989F<=*c*<=0xEAA4.
_ismbcl2_l	JIS level-2: 0x989F<=*c*<=0xEAA4.

The functions check that the specified value *c* matches the test conditions described above, but do not check that *c* is a valid multibyte character. If the lower byte is in the ranges 0x00 – 0x3F, 0x7F, or 0xFD – 0xFF, these functions return a nonzero value, indicating that the character satisfies the test condition. Use _ismbbtrail to test whether the multibyte character is defined.

End Code Page 932 Specific

Requirements

Routine	Required header	Compatibility
_ismbcl0	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbcl0_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbcl1	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbcl1_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbcl2	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbcl2_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Character Classification
_ismbc Routines
is, isw Routines

_ismbclegal, _ismbclegal_l, _ismbcsymbol, _ismbcsymbol_l

Checks whether a multibyte character is an legal or symbol character.

```
int _ismbclegal(
   unsigned int c
);
int _ismbclegal_l(
   unsigned int c,
   _locale_t locale
);
int _ismbcsymbol(
   unsigned int c
);
int _ismbcsymbol_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Character to be tested.

locale

Locale to use.

Return Value

Each of these routines returns a nonzero value if the character satisfies the test condition or 0 if it does not. If *c* <= 255 and there is a corresponding **_ismbb** routine (for example, **_ismbcalnum** corresponds to **_ismbbalnum**), the result is the return value of the corresponding **_ismbb** routine.

Remarks

Each of these functions tests a given multibyte character for a given condition.

The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior.

Routine	Test condition	Code page 932 example
_ismbclegal	Valid multibyte	Returns nonzero if and only if the first byte of *c* is within ranges 0x81 – 0x9F or 0xE0 – 0xFC, while the second byte is within ranges 0x40 - 0x7E or 0x80 - FC.
_ismbcsymbol	Multibyte symbol	Returns nonzero if and only if 0x8141<=*c*<=0x81AC.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_istlegal	Always returns false	**_ismbclegal**	Always returns false.
_istlegal_l	Always returns false	**_ismbclegal_l**	Always returns false.

Requirements

Routine	Required header	Compatibility

_ismbclegal,_ismbclegal_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbcsymbol, _ismbcsymbol_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Character Classification
_ismbc Routines
is, isw Routines
_ismbb Routines

_ismbclower, _ismbclower_l, _ismbcupper, _ismbcupper_l

Checks whether a multibyte character is a lowercase or uppercase character.

```
int _ismbclower(
   unsigned int c
);
int _ismbclower_l(
   unsigned int c,
   _locale_t locale
);
int _ismbcupper(
   unsigned int c
);
int _ismbcupper_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Character to be tested.

locale

Locale to use.

Return Value

Each of these routines returns a nonzero value if the character satisfies the test condition or 0 if it does not. If *c* <= 255 and there is a corresponding **_ismbb** routine (for example, **_ismbcalnum** corresponds to **_ismbbalnum**), the result is the return value of the corresponding **_ismbb** routine.

Remarks

Each of these functions tests a given multibyte character for a given condition.

The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior.

Routine	Test condition	Code page 932 example
_ismbclower	Lowercase alphabetic	Returns nonzero if and only if *c* is a single-byte representation of an ASCII lowercase English letter: 0x61<=*c*<=0x7A.
_ismbclower_l	Lowercase alphabetic	Returns nonzero if and only if *c* is a single-byte representation of an ASCII lowercase English letter: 0x61<=*c*<=0x7A.
_ismbcupper	Uppercase alphabetic	Returns nonzero if and only if *c* is a single-byte representation of an ASCII uppercase English letter: 0x41<=*c*<=0x5A.
_ismbcupper_l	Uppercase alphabetic	Returns nonzero if and only if *c* is a single-byte representation of an ASCII uppercase English letter: 0x41<=*c*<=0x5A.

Requirements

Routine	Required header	Compatibility
_ismbclower	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_ismbclower_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbcupper	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbcupper_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

- System::Char::IsLower
- System::Char::IsUpper

See Also

Reference

Character Classification
_ismbc Routines
Locale
Interpretation of Multibyte-Character Sequences
is, isw Routines
_ismbb Routines

_ismbslead, _ismbstrail, _ismbslead_l, _ismbstrail_l

Perform context-sensitive tests for multibyte-character string lead and trail bytes and determine whether a given substring pointer points to a lead byte or a trail byte.

```
int _ismbslead(
   const unsigned char *str,
   const unsigned char *current
);
int _ismbstrail(
   const unsigned char *str,
   const unsigned char *current
);
int _ismbslead_l(
   const unsigned char *str,
   const unsigned char *current,
   _locale_t locale
);
int _ismbstrail_l(
   const unsigned char *str,
   const unsigned char *current,
   _locale_t locale
);
```

Parameters

str

Pointer to the start of the string or previous known lead byte.

current

Pointer to the position in the string to be tested.

locale

Locale to use.

Return Value

_ismbslead and **_ismbstrail** return –1 if the character is a lead or trail byte, respectively. If the input strings are valid strings but not a lead or trail byte, these functions return zero. If either argument is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **NULL** and set **errno** to **EINVAL**.

Remarks

_ismbslead and **_ismbstrail** are slower than the **_ismbblead** and **_ismbbtrail** versions because they take the string context into account.

The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior.

Requirements

Routine	Required header	Optional header	Compatibility
_ismbslead	<mbctype.h> or <mbstring.h>	<ctype.h>,* <limits.h>, <stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbstrail	<mbctype.h> or <mbstring.h>	<ctype.h>,* <limits.h>, <stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_ismbslead_l	<mbctype.h> or <mbstring.h>	<ctype.h>,* <limits.h>, <stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ismbstrail_l	<mbctype.h> or <mbstring.h>	<ctype.h>,* <limits.h>, <stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

* For manifest constants for the test conditions.

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Character Classification
_ismbc Routines
is, isw Routines
_ismbb Routines

_isnan

Checks a given double-precision floating-point value for not a number (NAN).

```
int _isnan(
   double x
);
```

Parameters

x

Double-precision floating-point value.

Return Value

_isnan returns a nonzero value if the argument *x* is a NAN; otherwise it returns 0.

Remarks

The **_isnan** function tests a given double-precision floating-point value *x*, returning a nonzero value if *x* is a not a number (NAN). A NAN is generated when the result of a floating-point operation cannot be represented in Institute of Electrical and Electronics Engineers (IEEE) format. For information about how a NAN is represented for output, see printf.

Requirements

Routine	Required header	Compatibility
_isnan	<float.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

System::Double::IsNan

See Also

Reference

Floating-Point Support
_finite
_fpclass

isxdigit, iswxdigit, _isxdigit_l, _iswxdigit_l

Determine if an integer represents character that is a hex digit.

```
int isxdigit(
   int c
);
int iswxdigit(
   wint_t c
);
int _isxdigit_l(
   int c,
   _locale_t locale
);
int _iswxdigit_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c

Integer to test.

locale

Locale to use.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of a hexadecimal digit. **isxdigit** returns a nonzero value if *c* is a hexadecimal digit (A – F, a – f, or 0 – 9). **iswxdigit** returns a nonzero value if *c* is a wide character that corresponds to a hexadecimal digit character. Each of these routines returns 0 if *c* does not satisfy the test condition.

For the "C" locale, the **iswxdigit** function does not provide support for Unicode full width hexadecimal characters.

When used with a debug CRT library, **isxdigit** will display a CRT assert if passed a parameter that is not EOF or in the range of 0 through 0xFF. When used with a debug CRT library, **isxdigit** will use the parameter as an index into an array, with undefined results if the parameter is not EOF or in the range of 0 through 0xFF.

The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_istxdigit	**isxdigit**	**isxdigit**	**iswxdigit**

Requirements

Routine	Required header	Compatibility
isxdigit	<ctype.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
iswxdigit	<ctype.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_isxdigit_l	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_iswxdigit_l	<ctype.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::Char::IsNumber

See Also

Reference

Character Classification
Locale
is, isw Routines

isupper, _isupper_l, iswupper, _iswupper_l

Determines if a particular character is in upper case.

```
int isupper(
   int c
);
int _isupper_l (
   int c,
   _locale_t locale
);
int iswupper(
   wint_t c
);
int _iwsupper_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c

Integer to test.

locale

Locale to use.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of an uppercase letter. **isupper** returns a nonzero value if *c* is an uppercase character (A – Z). **iswupper** returns a nonzero value if *c* is a wide character that corresponds to an uppercase letter, or if *c* is one of an implementation-defined set of wide characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is nonzero. Each of these routines returns 0 if *c* does not satisfy the test condition.

When used with a debug CRT library, **isupper** will display a CRT assert if passed a parameter that is not EOF or in the range of 0 through 0xFF. When used with a debug CRT library, **isupper** will use the parameter as an index into an array, with undefined results if the parameter is not EOF or in the range of 0 through 0xFF.

The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_istupper	**isupper**	_ismbcupper	**iswupper**
_istupper_l	**_isupper_l**	_ismbclower, _ismbclower_l, _ismbcupper, _ismbcupper_l	**_iswupper_l**

Requirements

Routine	Required header	Compatibility
isupper	<ctype.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_isupper_l	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
iswupper	<ctype.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_iswupper_l	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::Char::IsUpper

See Also
Reference
Character Classification
Locale
is, isw Routines

isspace, iswspace, _isspace_l, _iswspace_l

Determines if a particular character represents a space character.

```
int isspace(
   int c
);
int iswspace(
   wint_t c
);
int _isspace_l(
   int c,
   _locale_t locale
);
int _iswspace_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c

Integer to test.

locale

Locale to use.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of a space character. **isspace** returns a nonzero value if *c* is a white-space character (0x09 – 0x0D or 0x20). The result of the test condition for the **isspace** function depends on the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for any locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale passed in instead.

iswspace returns a nonzero value if *c* is a wide character that corresponds to a standard white-space character.

When used with a debug CRT library, **isspace** will display a CRT assert if passed a parameter that is not EOF or in the range of 0 through 0xFF. When used with a non-debug CRT library, **isspace** will use the parameter as an index into an array, with undefined results if the parameter is not EOF or in the range of 0 through 0xFF.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_ istspace	**isspace**	_ismbcspace	**iswspace**

Requirements

Routine	Required header	Compatibility
isspace	<ctype.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
iswspace	<ctype.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_isspace_l	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_iswspace_l	<ctype.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::Char::IsWhiteSpace

See Also

Reference

Character Classification
Locale
is, isw Routines

ispunct, iswpunct, _ispunct_l, _iswpunct_l

Determines if a particular character represents a punctuation character.

```
int ispunct(
   int c
);
int iswpunct(
   wint_t c
);
int _ispunct_l(
   int c,
   _locale_t locale
);
int _iswpunct_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c

Integer to test.

locale

The locale to use.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of a punctuation character. **ispunct** returns a nonzero value for any printable character that is not a space character or a character for which **isalnum** is nonzero. **iswpunct** returns a nonzero value for any printable wide character that is neither the space wide character nor a wide character for which **iswalnum** is nonzero. Each of these routines returns 0 if *c* does not satisfy the test condition.

The result of the test condition for the **ispunct** function depends on the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for any locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale passed in instead.

When used with a debug CRT library, **ispunct** will display a CRT assert if passed a parameter that is not EOF or in the range of 0 through 0xFF. When used with a debug CRT library, **ispunct** will use the parameter as an index into an array, with undefined results if the parameter is not EOF or in the range of 0 through 0xFF.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_ istpunct	**ispunct**	_ismbcpunct	**iswpunct**

Requirements

Routine	Required header	Compatibility
ispunct	<ctype.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
iswpunct	<ctype.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ispunct_l	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_iswpunct_l	<ctype.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

See Also

Reference

Character Classification

Locale

is, isw Routines

isprint, iswprint, _isprint_l, _iswprint_l

Determine if an integer represents a printable character.

```
int isprint(
   int c
);
int iswprint(
   wint_t c
);
int _isprint_l(
   int c,
   _locale_t locale
);
int _iswprint_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c

Integer to test.

locale

The locale to use.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of a printable character. **isprint** returns a nonzero value if *c* is a printable character, including the space character (0x20 – 0x7E). **iswprint** returns a nonzero value if *c* is a printable wide character, including the space wide character. Each of these routines returns 0 if *c* does not satisfy the test condition.

The result of the test condition for these functions depends on the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for any locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale passed in instead.

When used with a debug CRT library, **isprint** will display a CRT assert if passed a parameter that is not EOF or in the range of 0 through 0xFF. When used with a debug CRT library, **isprint** will use the parameter as an index into an array, with undefined results if the parameter is not EOF or in the range of 0 through 0xFF.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_unicode defined
_ istprint	**isprint**	_ismbcprint	**iswprint**

Requirements

Routine	Required header	Compatibility
isprint	<ctype.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
iswprint	<ctype.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_isprint_l	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_isw print_l	<ctype.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Character Classification
Locale
is, isw Routines

itoa

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _itoa or security-enhanced _itoa_s instead.

_itoa, _i64toa, _ui64toa, _itow, _i64tow, _ui64tow

Converts an integer to a string. These functions are deprecated because more secure versions are available; see _itoa_s, _i64toa_s, _ui64toa_s, _itow_s, _i64tow_s, _ui64tow_s.

```
char *_itoa(
   int value,
   char *str,
   int radix
);
char *_i64toa(
   __int64 value,
   char *str,
   int radix
);
char * _ui64toa(
   unsigned _int64 value,
   char *str,
   int radix
);
wchar_t * _itow(
   int value,
   wchar_t *str,
   int radix
);
wchar_t * _i64tow(
   __int64 value,
   wchar_t *str,
   int radix
);
wchar_t * _ui64tow(
   unsigned __int64 value,
   wchar_t *str,
   int radix
);
template <size_t size>
char *_itoa(
   int value,
   char (&str)[size],
   int radix
); // C++ only
template <size_t size>
char *_i64toa(
   __int64 value,
   char (&str)[size],
   int radix
); // C++ only
template <size_t size>
char * _ui64toa(
   unsigned _int64 value,
   char (&str)[size],
   int radix
); // C++ only
template <size_t size>
wchar_t * _itow(
   int value,
   wchar_t (&str)[size],
   int radix
); // C++ only
template <size_t size>
wchar_t * _i64tow(
   __int64 value,
   wchar_t (&str)[size],
   int radix
```

```
); // C++ only
template <size_t size>
wchar_t * _ui64tow(
   unsigned __int64 value,
   wchar_t (&str)[size],
   int radix
); // C++ only
```

Parameters

value

Number to be converted.

str

String result.

radix

Base of *value*; which must be in the range 2–36.

Return Value

Each of these functions returns a pointer to *str*. There is no error return.

Remarks

The **_itoa**, **_i64toa**, and **_ui64toa** functions convert the digits of the given *value* argument to a null-terminated character string and stores the result (up to 33 characters for **_itoa** and 65 for **_i64toa** and **_ui64toa**) in *str*. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (**–**). **_itow**, **_i64tow**, and **_ui64tow** are wide-character versions of **_itoa**, **_i64toa**, and **_ui64toa**, respectively.

Security Note
To prevent buffer overruns, ensure that the *str* buffer is large enough to hold the converted digits plus the trailing null-character and a sign character.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

Tchar.h routine	**_UNICODE and _MBCS not defined**	**_MBCS defined**	**_UNICODE defined**
_itot	**_itoa**	**_itoa**	**_itow**
_i64tot	**_i64toa**	**_i64toa**	**_i64tow**
_ui64tot	**_ui64toa**	**_ui64toa**	**_ui64tow**

Requirements

Routine	**Required header**	**Compatibility**
_itoa	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_i64toa	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ui64toa	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_itow	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_i64tow	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ui64tow	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_itoa.c
// compile with: /W3
// This program makes use of the _itoa functions
// in various examples.

#include <stdlib.h>

int main( void )
{
   char buffer[65];
   int r;
   for( r=10; r>=2; --r )
   {
     _itoa( -1, buffer, r ); // C4996
     // Note: _itoa is deprecated; consider using _itoa_s instead
     printf( "base %d: %s (%d chars)\n", r, buffer, strlen(buffer) );
   }
   printf( "\n" );
   for( r=10; r>=2; --r )
   {
     _i64toa( -1L, buffer, r ); // C4996
     // Note: _i64toa is deprecated; consider using _i64toa_s
     printf( "base %d: %s (%d chars)\n", r, buffer, strlen(buffer) );
   }
   printf( "\n" );
   for( r=10; r>=2; --r )
   {
     _ui64toa( 0xffffffffffffffffL, buffer, r ); // C4996
     // Note: _ui64toa is deprecated; consider using _ui64toa
     printf( "base %d: %s (%d chars)\n", r, buffer, strlen(buffer) );
   }
}
```

Output

```
base 10: -1 (2 chars)
base 9: 12068657453 (11 chars)
base 8: 37777777777 (11 chars)
base 7: 211301422353 (12 chars)
base 6: 1550104015503 (13 chars)
base 5: 32244002423140 (14 chars)
base 4: 3333333333333333 (16 chars)
base 3: 102002022201221111210 (21 chars)
base 2: 11111111111111111111111111111111 (32 chars)

base 10: -1 (2 chars)
base 9: 145808576354216723756 (21 chars)
base 8: 1777777777777777777777 (22 chars)
base 7: 4501202152252313413460 1 (23 chars)
base 6: 3520522010102100444244423 (25 chars)
base 5: 2214220303114400424121122430 (28 chars)
base 4: 3333333333333333333333333333333333 (32 chars)
base 3: 111122200221221201012110201202102102112 20 (41 chars)
base 2: 1111111111111111111111111111111111111111111111111111111111111111 (64 chars)

base 10: 18446744073709551615 (20 chars)
```

```
base 9: 145808576354216723756 (21 chars)
base 8: 1777777777777777777777 (22 chars)
base 7: 4501202152252313413460 1 (23 chars)
base 6: 3520522010102100444244423 (25 chars)
base 5: 2214220303114400424121122430 (28 chars)
base 4: 3333333333333333333333333333333 (32 chars)
base 3: 1111222002212212010121102012021021021 1220 (41 chars)
base 2: 1111111111111111111111111111111111111111111111111111111111111111 (64 chars)
```

.NET Framework Equivalent

System::Convert::ToString

See Also

Reference

Data Conversion
_ltoa, _ltow
_ltoa_s, _ltow_s
_ultoa, _ultow
_ultoa_s, _ultow_s

_itoa_s, _i64toa_s, _ui64toa_s, _itow_s, _i64tow_s, _ui64tow_s

Converts an integer to a string. These are versions of _itoa, _i64toa, _ui64toa, _itow, _i64tow, _ui64tow with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _itoa_s(
   int value,
   char *buffer,
   size_t sizeInCharacters,
   int radix
);
errno_t _i64toa_s(
   __int64 value,
   char *buffer,
   size_t sizeInCharacters,
   int radix
);
errno_t _ui64toa_s(
   unsigned _int64 value,
   char *buffer,
   size_t sizeInCharacters,
   int radix
);
errno_t _itow_s(
   int value,
   wchar_t *buffer,
   size_t sizeInCharacters,
   int radix
);
errno_t _i64tow_s(
   __int64 value,
   wchar_t *buffer,
   size_t sizeInCharacters,
   int radix
);
errno_t _ui64tow_s(
   unsigned __int64 value,
   wchar_t *buffer,
   size_t sizeInCharacters,
   int radix
);
template <size_t size>
errno_t _itoa_s(
   int value,
   char (&buffer)[size],
   int radix
); // C++ only
template <size_t size>
errno_t _itow_s(
   int value,
   wchar_t (&buffer)[size],
   int radix
); // C++ only
```

Parameters

[in] *value*

Number to be converted.

[out] *buffer*

Filled with the result of the conversion.

[in] *sizeInCharacters*

Size of the buffer in single-byte characters or wide characters.

[in] *radix*

Base of *value*; which must be in the range 2–36.

Return Value

Zero if successful; an error code on failure. If any of the following conditions applies, the function invokes an invalid parameter handler, as described in Parameter Validation.

Error Conditions

value	buffer	sizeInCharacters	radix	Return
any	**NULL**	any	any	**EINVAL**
any	any	<=0	any	**EINVAL**
any	any	<= length of the result string required	any	**EINVAL**
any	any	any	*radix* < 2 or *radix* > 36	**EINVAL**

Security Issues

These functions can generate an access violation if *buffer* does not point to valid memory and is not **NULL**, or if the length of the buffer is not long enough to hold the result string.

Remarks

Except for the parameters and return value, the **_itoa_s** functions have the same behavior as the corresponding less secure versions.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_itot_s	**_itoa_s**	**_itoa_s**	**_itow_s**
_i64tot_s	**_i64toa_s**	**_i64toa_s**	**_i64tow_s**
_ui64tot_s	**_ui64toa_s**	**_ui64toa_s**	**_ui64tow_s**

Requirements

Routine	Required header	Compatibility
_itoa_s	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_i64toa_s	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ui64toa_s	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_itow_s	<stdlib.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_i64tow_s	<stdlib.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ui64tow_s	<stdlib.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_itoa_s.c
#include <stdlib.h>

int main( void )
{
   char buffer[65];
   int r;
   for( r=10; r>=2; --r )
   {
     _itoa_s( -1, buffer, 65, r );
     printf( "base %d: %s (%d chars)\n", r, buffer, strlen(buffer) );
   }
   printf( "\n" );
   for( r=10; r>=2; --r )
   {
     _i64toa_s( -1L, buffer, 65, r );
     printf( "base %d: %s (%d chars)\n", r, buffer, strlen(buffer) );
   }
   printf( "\n" );
   for( r=10; r>=2; --r )
   {
     _ui64toa_s( 0xffffffffffffffffL, buffer, 65, r );
     printf( "base %d: %s (%d chars)\n", r, buffer, strlen(buffer) );
   }
}
```

Output

```
base 10: -1 (2 chars)
base 9: 12068657453 (11 chars)
base 8: 37777777777 (11 chars)
base 7: 211301422353 (12 chars)
base 6: 1550104015503 (13 chars)
base 5: 32244002423140 (14 chars)
base 4: 3333333333333333 (16 chars)
base 3: 102002022201221111210 (21 chars)
base 2: 11111111111111111111111111111111 (32 chars)

base 10: -1 (2 chars)
base 9: 145808576354216723756 (21 chars)
base 8: 1777777777777777777777 (22 chars)
base 7: 45012021522523134134601 (23 chars)
base 6: 3520522010102100444244423 (25 chars)
base 5: 2214220303114400424121122430 (28 chars)
base 4: 33333333333333333333333333333333 (32 chars)
base 3: 1111222002212212010121102012021021021122 0 (41 chars)
base 2: 1111111111111111111111111111111111111111111111111111111111111111 (64 chars)

base 10: 18446744073709551615 (20 chars)
base 9: 14580857635421672375 6 (21 chars)
base 8: 1777777777777777777777 (22 chars)
base 7: 45012021522523134134601 (23 chars)
base 6: 3520522010102100444244423 (25 chars)
base 5: 2214220303114400424121122430 (28 chars)
base 4: 33333333333333333333333333333333 (32 chars)
base 3: 1111222002212212010121102012021021021122 0 (41 chars)
base 2: 1111111111111111111111111111111111111111111111111111111111111111 (64 chars)
```

.NET Framework Equivalent

System::Convert::ToString

See Also

Reference

Data Conversion

_ltoa, _ltow

_ultoa, _ultow

j0, j1, jn

These POSIX functions are deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant Bessel Functions: _j0, _j1, _jn instead.

kbhit

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _kbhit instead.

_kbhit

Checks the console for keyboard input.

```
int _kbhit( void );
```

Return Value

_kbhit returns a nonzero value if a key has been pressed. Otherwise, it returns 0.

Remarks

The **_kbhit** function checks the console for a recent keystroke. If the function returns a nonzero value, a keystroke is waiting in the buffer. The program can then call **_getch** or **_getche** to get the keystroke.

Requirements

Routine	Required header	Compatibility
_kbhit	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_kbhit.c
// compile with: /c
/* This program loops until the user
 * presses a key. If _kbhit returns nonzero, a
 * keystroke is waiting in the buffer. The program
 * can call _getch or _getche to get the keystroke.
 */

#include <conio.h>
#include <stdio.h>

int main( void )
{
   /* Display message until key is pressed. */
   while( !_kbhit() )
      _cputs( "Hit me!! " );

   /* Use _getch to throw key away. */
   printf( "\nKey struck was '%c'\n", _getch() );
}
```

Sample Output

```
Hit me!! Hit me!! Hit me!! Hit me!! Hit me!! Hit me!! Hit me!!
Key struck was 'q'
```

See Also

Reference

Console and Port I/O

labs

Calculates the absolute value of a long integer.

```
long labs(
      long n
);
```

Parameters

n

Long-integer value.

Return Value

The **labs** function returns the absolute value of its argument. There is no error return.

Requirements

Routine	Required header	Compatibility
labs	<stdlib.h> and <math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

See the example for abs.

.NET Framework Equivalent

System::Math::Abs

See Also

Reference

Data Conversion
Floating-Point Support
abs, _abs64
_cabs
fabs, fabsf

ldexp

Computes a real number from the mantissa and exponent.

```
double ldexp(
   double x,
   int exp
);
float ldexp(
   float x,
      int exp
);  // C++ only
long double ldexp(
   long double x,
   int exp
);  // C++ only
```

Parameters

x

Floating-point value.

exp

Integer exponent.

Return Value

The **ldexp** function returns the value of *x* * 2exp if successful. On overflow (depending on the sign of *x*), **ldexp** returns +/– **HUGE_VAL**; the **errno** variable is set to **ERANGE**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on this, and other, return codes.

Remarks

C++ allows overloading, so you can call overloads of **ldexp**. In a C program, **ldexp** always takes a double and an int and returns a double.

Requirements

Routine	Required header	Compatibility
ldexp	<math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_ldexp.c

#include <math.h>
#include <stdio.h>

int main( void )
{
   double x = 4.0, y;
   int p = 3;

   y = ldexp( x, p );
```

```
    printf( "%2.1f times two to the power of %d is %2.1f\n", x, p, y );
}
```

Output

```
4.0 times two to the power of 3 is 32.0
```

.NET Framework Equivalent

System::Math::Pow

See Also

Reference

Floating-Point Support
frexp
modf, modff

ldiv

Computes the quotient and remainder of a long integer.

```
ldiv_t ldiv(
   long int numer,
   long int denom
);
```

Parameters

numer

Numerator.

denom

Denominator.

Return Value

ldiv returns a structure of type **ldiv_t** that comprises both the quotient and the remainder.

Remarks

The **ldiv** function divides *numer* by *denom*, computing the quotient and remainder. The sign of the quotient is the same as that of the mathematical quotient. The absolute value of the quotient is the largest integer that is less than the absolute value of the mathematical quotient. If the denominator is 0, the program terminates with an error message. **ldiv** is the same as **div**, except that the arguments of **ldiv** and the members of the returned structure are all of type **long int**.

The **ldiv_t** structure, defined in STDLIB.H, contains **long int quot**, the quotient, and **long int rem**, the remainder.

Requirements

Routine	Required header	Compatibility
ldiv	<stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_ldiv.c

#include <stdlib.h>
#include <math.h>
#include <stdio.h>

int main( void )
{
   long x = 5149627, y = 234879;
   ldiv_t div_result;

   div_result = ldiv( x, y );
   printf( "For %ld / %ld, the quotient is ", x, y );
   printf( "%ld, and the remainder is %ld\n",
            div_result.quot, div_result.rem );
}
```

Output

```
For 5149627 / 234879, the quotient is 21, and the remainder is 217168
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support

div

lfind

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _lfind or security-enhanced _lfind_s instead.

_lfind

Performs a linear search for the specified key. A more secure version is available; see _lfind_s.

```
void *_lfind(
   const void *key,
   const void *base,
   unsigned int *num,
   unsigned int width,
   int (__cdecl *compare)(const void *, const void *)
);
```

Parameters

key

Object to search for.

base

Pointer to the base of search data.

num

Number of array elements.

width

Width of array elements.

compare

Pointer to comparison routine. The first parameter is a pointer to key for search. The second parameter is a pointer to array element to be compared with key.

Return Value

If the key is found, **_lfind** returns a pointer to the element of the array at *base* that matches *key*. If the key is not found, **_lfind** returns **NULL**.

Remarks

The **_lfind** function performs a linear search for the value *key* in an array of *num* elements, each of *width* bytes. Unlike **bsearch**, **_lfind** does not require the array to be sorted. The *base* argument is a pointer to the base of the array to be searched. The *compare* argument is a pointer to a user-supplied routine that compares two array elements and then returns a value specifying their relationship. **_lfind** calls the *compare* routine one or more times during the search, passing pointers to two array elements on each call. The *compare* routine must compare the elements and then return nonzero (meaning the elements are different) or 0 (meaning the elements are identical).

This function validates its parameters. If *compare*, *key* or *num* is **NULL**, or if *base* is NULL and **num* is nonzero, or if *width* is less than zero, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **NULL**.

Requirements

Routine	Required header	Compatibility
_lfind	<search.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_lfind.c
// This program uses _lfind to search a string array
// for an occurrence of "hello".
```

```
#include <search.h>
#include <string.h>
#include <stdio.h>

int compare(const void *arg1, const void *arg2 )
{
   return( _stricmp( * (char**)arg1, * (char**)arg2 ) );
}

int main( )
{
   char *arr[] = {"Hi", "Hello", "Bye"};
   int n = sizeof(arr) / sizeof(char*);
   char **result;
   char *key = "hello";

   result = (char **)_lfind( &key, arr,
                     &n, sizeof(char *), compare );

   if( result )
      printf( "%s found\n", *result );
   else
      printf( "hello not found!\n" );
}
```

Output

```
Hello found
```

.NET Framework Equivalent

System::Collections::ArrayList::Contains

See Also

Reference

Searching and Sorting
_lfind_s
bsearch
_lsearch
qsort

_lfind_s

Performs a linear search for the specified key. A version of _lfind with security enhancements as described in Security Enhancements in the CRT.

```
void *_lfind_s(
   const void *key,
   const void *base,
   unsigned int *num,
   size_t size,
   int (__cdecl *compare)(void *, const void *, const void *),
   void * context
);
```

Parameters

key

Object to search for.

base

Pointer to the base of search data.

num

Number of array elements.

size

Size of array elements in bytes.

compare

Pointer to comparison routine. The first parameter is the *context* pointer. The second parameter is a pointer to key for search. The third parameter is a pointer to array element to be compared with key.

context

A pointer to an object that might be accessed in the comparison function.

Return Value

If the key is found, **_lfind_s** returns a pointer to the element of the array at *base* that matches *key*. If the key is not found, **_lfind_s** returns **NULL**.

If invalid parameters are passed to the function, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **NULL**.

Error Conditions

key	base	compare	num	size	errno
NULL	any	any	any	any	**EINVAL**
any	**NULL**	any	!= 0	any	**EINVAL**
any	any	any	any	zero	**EINVAL**
any	any	**NULL**	an	any	**EINVAL**

Remarks

The **_lfind_s** function performs a linear search for the value *key* in an array of *num* elements, each of *width* bytes. Unlike **bsearch_s**, **_lfind_s** does not require the array to be sorted. The *base* argument is a pointer to the base of the array to be searched. The *compare* argument is a pointer to a user-supplied routine that compares two array elements and then returns a value specifying their relationship. **_lfind_s** calls the *compare* routine one or more times during the search, passing the *context* pointer and pointers to two array elements on each call. The *compare* routine must compare the elements then return nonzero

(meaning that the elements are different) or 0 (meaning the elements are identical).

_lfind_s is similar to **_lfind** except for the addition of the *context* pointer to the arguments of the comparison function and the parameter list of the function. The *context* pointer can be useful if the searched data structure is part of an object and the *compare* function needs to access members of the object. The *compare* function can cast the void pointer into the appropriate object type and access members of that object. The addition of the *context* parameter makes **_lfind_s** more secure because additional context can be used to avoid reentrancy bugs associated with using static variables to make data available to the *compare* function.

Requirements

Routine	Required header	Compatibility
_lfind_s	<search.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_lfind_s.cpp
// This program uses _lfind_s to search a string array,
// passing a locale as the context.
// compile with: /EHsc
#include <stdlib.h>
#include <stdio.h>
#include <search.h>
#include <process.h>
#include <locale.h>
#include <locale>
#include <windows.h>
using namespace std;

// The sort order is dependent on the code page.  Use 'chcp' at the
// command line to change the codepage.  When executing this application,
// the command prompt codepage must match the codepage used here:

#define CODEPAGE_850

#ifdef CODEPAGE_850
// Codepage 850 is the OEM codepage used by the command line,
// so \x00e1 is the German Sharp S

char *array1[] = { "wei\x00e1", "weis", "annehmen", "weizen", "Zeit",
                   "weit" };

#define GERMAN_LOCALE "German_Germany.850"

#endif

#ifdef CODEPAGE_1252
   // If using codepage 1252 (ISO 8859-1, Latin-1), use \x00df
   // for the German Sharp S
char *array1[] = { "wei\x00df", "weis", "annehmen", "weizen", "Zeit",
                   "weit" };

#define GERMAN_LOCALE "German_Germany.1252"

#endif

// The context parameter lets you create a more generic compare.
// Without this parameter, you would have stored the locale in a
// static variable, thus making it vulnerable to thread conflicts
// (if this were a multithreaded program).

int compare( void *pvlocale, const void *str1, const void *str2)
{
```

```
    char *s1 = *(char**)str1;
    char *s2 = *(char**)str2;

    locale& loc = *( reinterpret_cast< locale * > ( pvlocale));

    return use_facet< collate<char> >(loc).compare(
       s1, s1+strlen(s1),
       s2, s2+strlen(s2) );
}

void find_it( char *key, char *array[], unsigned int num, locale &loc )
{
   char **result = (char **)_lfind_s( &key, array,
                      &num, sizeof(char *), compare, &loc );
   if( result )
      printf( "%s found\n", *result );
   else
      printf( "%s not found\n", key );
}

int main( )
{
   find_it( "weit", array1, sizeof(array1)/sizeof(char*), locale(GERMAN_LOCALE) );
}
```

Output

```
weit found
```

.NET Framework Equivalent

Contains

See Also

Reference

Searching and Sorting
bsearch_s
_lsearch_s
qsort_s
_lfind

localeconv

Gets detailed information on locale settings.

```
struct lconv *localeconv( void );
```

Return Value

localeconv returns a pointer to a filled-in object of type struct lconv. The values contained in the object can be overwritten by subsequent calls to **localeconv** and do not directly modify the object. Calls to setlocale with *category* values of **LC_ALL**, **LC_MONETARY**, or **LC_NUMERIC** overwrite the contents of the structure.

Remarks

The **localeconv** function gets detailed information about numeric formatting for the current locale. This information is stored in a structure of type **lconv**. The **lconv** structure, defined in LOCALE.H, contains the following members:

char *decimal_point

Decimal-point character for nonmonetary quantities.

char *thousands_sep

Character that separates groups of digits to left of decimal point for nonmonetary quantities.

char *grouping

Size of each group of digits in nonmonetary quantities.

char *int_curr_symbol

International currency symbol for current locale. First three characters specify alphabetic international currency symbol as defined in the *ISO 4217 Codes for the Representation of Currency and Funds* standard. Fourth character (immediately preceding null character) separates international currency symbol from monetary quantity.

char *currency_symbol

Local currency symbol for current locale.

char *mon_decimal_point

Decimal-point character for monetary quantities.

char *mon_thousands_sep

Separator for groups of digits to left of decimal place in monetary quantities.

char *mon_grouping

Size of each group of digits in monetary quantities.

char *positive_sign

String denoting sign for nonnegative monetary quantities.

char *negative_sign

String denoting sign for negative monetary quantities.

char int_frac_digits

Number of digits to right of decimal point in internationally formatted monetary quantities.

char frac_digits

Number of digits to right of decimal point in formatted monetary quantities.

char p_cs_precedes

Set to 1 if currency symbol precedes value for nonnegative formatted monetary quantity. Set to 0 if symbol follows value.

char p_sep_by_space

Set to 1 if currency symbol is separated by space from value for nonnegative formatted monetary quantity. Set to 0 if there is no space separation.

char n_cs_precedes

Set to 1 if currency symbol precedes value for negative formatted monetary quantity. Set to 0 if symbol succeeds value.

char n_sep_by_space

Set to 1 if currency symbol is separated by space from value for negative formatted monetary quantity. Set to 0 if there is no space separation.

char p_sign_posn

Position of positive sign in nonnegative formatted monetary quantities.

char n_sign_posn

Position of positive sign in negative formatted monetary quantities.

The **char *** members of the structure are pointers to strings. Any of these (other than **char *decimal_point**) that equals "" is either of zero length or is not supported in the current locale. The **char** members of the structure are nonnegative numbers. Any of these that equals **CHAR_MAX** is not supported in the current locale.

The elements of **grouping** and **mon_grouping** are interpreted according to the following rules.

CHAR_MAX

Do not perform any further grouping.

0

Use previous element for each of remaining digits.

n

Number of digits that make up current group. Next element is examined to determine size of next group of digits before current group.

The values for **int_curr_symbol** are interpreted according to the following rules:

- The first three characters specify the alphabetic international currency symbol as defined in the *ISO 4217 Codes for the Representation of Currency and Funds* standard.
- The fourth character (immediately preceding the null character) separates the international currency symbol from the monetary quantity.

The values for **p_cs_precedes** and **n_cs_precedes** are interpreted according to the following rules (the **n_cs_precedes** rule is in parentheses):

0

Currency symbol follows value for nonnegative (negative) formatted monetary value.

1

Currency symbol precedes value for nonnegative (negative) formatted monetary value.

The values for **p_sep_by_space** and **n_sep_by_space** are interpreted according to the following rules (the **n_sep_by_space** rule is in parentheses):

0

Currency symbol is separated from value by space for nonnegative (negative) formatted monetary value.

1

There is no space separation between currency symbol and value for nonnegative (negative) formatted monetary value.

The values for **p_sign_posn** and **n_sign_posn** are interpreted according to the following rules:

0

Parentheses surround quantity and currency symbol.

1

Sign string precedes quantity and currency symbol.

2

Sign string follows quantity and currency symbol.

3

Sign string immediately precedes currency symbol.

4

Sign string immediately follows currency symbol.

Requirements

Routine	Required header	Compatibility
localeconv	<locale.h>	ANSI, Windows 98, Windows Me, Windows NT, Windows 2000, Windows XP, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

See Also

Reference

Locale
setlocale
strcoll Functions
strftime, wcsftime, _strftime_l, _wcsftime_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

localtime, _localtime32, _localtime64

Convert a time value and correct for the local time zone. These functions are deprecated because more secure versions are available; see localtime_s, _localtime32_s, _localtime64_s.

```
struct tm *localtime(
   const time_t *timer
);
struct tm *_localtime32(
   const __time32_t *timer
);
struct tm *_localtime64(
   const __time64_t *timer
);
```

Parameters

timer

Pointer to stored time.

Return Value

Return a pointer to the structure result, or **NULL** if the date passed to the function is:

- Before midnight, January 1, 1970.
- After 03:14:07, January 19, 2038, UTC (using **_time32** and **time32_t**).
- After 23:59:59, December 31, 3000, UTC (using **_time64** and **__time64_t**).

_localtime64, which uses the **__time64_t** structure, allows dates to be expressed up through 23:59:59, December 31, 3000, coordinated universal time (UTC), whereas **_localtime32** represents dates through 03:14:07 January 19, 2038, UTC.

In Visual C++ 2005, **localtime** is an inline function which evaluates to **_localtime64**, and **time_t** is equivalent to **__time64_t**. If you need to force the compiler to interpret **time_t** as the old 32-bit **time_t**, you can define **_USE_32BIT_TIME_T**. Doing this will cause **localtime** to evaluate to **_localtime32**. This is not recommended because your application may fail after January 19, 2038, and it is not allowed on 64-bit platforms.

The fields of the structure type tm store the following values, each of which is an **int**:

tm_sec

Seconds after minute (0 – 59).

tm_min

Minutes after hour (0 – 59).

tm_hour

Hours after midnight (0 – 23).

tm_mday

Day of month (1 – 31).

tm_mon

Month (0 – 11; January = 0).

tm_year

Year (current year minus 1900).

tm_wday

Day of week (0 – 6; Sunday = 0).

tm_yday

Day of year (0 – 365; January 1 = 0).

tm_isdst

Positive value if daylight saving time is in effect; 0 if daylight saving time is not in effect; negative value if status of daylight saving time is unknown. If the **TZ** environment variable is set, the C run-time library assumes rules appropriate to the United States for implementing the calculation of daylight-saving time (DST).

Remarks

The **localtime** function converts a time stored as a time_t value and stores the result in a structure of type **tm**. The **long** value *timer* represents the seconds elapsed since midnight (00:00:00), January 1, 1970, UTC. This value is usually obtained from the **time** function.

Both the 32-bit and 64-bit versions of **gmtime**, **mktime**, **mkgmtime**, and **localtime** all use a single **tm** structure per thread for the conversion. Each call to one of these routines destroys the result of the previous call.

localtime corrects for the local time zone if the user first sets the global environment variable **TZ**. When **TZ** is set, three other environment variables (**_timezone**, **_daylight**, and **_tzname**) are automatically set as well. If the **TZ** variable is not set, **localtime** attempts to use the time zone information specified in the Date/Time application in Control Panel. If this information cannot be obtained, PST8PDT, which signifies the Pacific Time Zone, is used by default. See _tzset for a description of these variables. **TZ** is a Microsoft extension and not part of the ANSI standard definition of **localtime**.

Note
The target environment should try to determine whether daylight saving time is in effect.

These functions validate their parameters. If *timer* is a null pointer, or if the timer value is negative, these functions invoke an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the functions return **NULL** and set **errno** to **EINVAL**.

Requirements

Routine	**Required header**	**Compatibility**
localtime	<time.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_localtime32	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_localtime64	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_localtime.cpp
// compile with: /W3
/* This program uses _time64 to get the current time
 * and then uses localtime64() to convert this time to a structure
 * representing the local time. The program converts the result
 * from a 24-hour clock to a 12-hour clock and determines the
 * proper extension (AM or PM).
 */

#include <stdio.h>
#include <string.h>
#include <time.h>

int main( void )
{
        struct tm *newtime;
        char am_pm[] = "AM";
```

```
        __time64_t long_time;

        _time64( &long_time );           // Get time as 64-bit integer.
                                         // Convert to local time.
        newtime = _localtime64( &long_time ); // C4996
        // Note: _localtime64 deprecated; consider _localetime64_s

        if( newtime->tm_hour > 12 )        // Set up extension.
                strcpy_s( am_pm, sizeof(am_pm), "PM" );
        if( newtime->tm_hour > 12 )        // Convert from 24-hour
                newtime->tm_hour -= 12;    //   to 12-hour clock.
        if( newtime->tm_hour == 0 )        // Set hour to 12 if midnight.
                newtime->tm_hour = 12;

        char buff[30];
        asctime_s( buff, sizeof(buff), newtime );
        printf( "%.19s %s\n", buff, am_pm );
}
```

Sample Output

```
Tue Feb 12 10:05:58 AM
```

.NET Framework Equivalent

System::DateTime::ToLocalTime

See Also

Reference

Time Management
asctime, _wasctime
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
_ftime, _ftime32, _ftime64
gmtime, _gmtime32, _gmtime64
localtime_s, _localtime32_s, _localtime64_s
time, _time32, _time64
_tzset

localtime_s, _localtime32_s, _localtime64_s

Converts a time value and corrects for the local time zone. These are versions of localtime, _localtime32, _localtime64 with security enhancements as described in Security Enhancements in the CRT.

```
errno_t localtime_s(
   struct tm* _tm,
   const time_t *time
);
errno_t _localtime32_s(
   struct tm* _tm,
   const time32_t *time
);
errno_t _localtime64_s(
   struct tm* _tm,
   const _time64_t *time
);
```

Parameters

_tm

Pointer to the time structure to be filled in.

time

Pointer to the stored time.

Return Value

Zero if successful. The return value is an error code if there is a failure. Error codes are defined in Errno.h. For a listing of these errors, see errno.

Error Conditions

_tm	*time*	**Return value**	**Value in *_tm***	**Invokes invalid parameter handler**
NULL	any	**EINVAL**	Not modified	Yes
Not **NULL** (points to valid memory)	**NULL**	**EINVAL**	All fields set to -1	Yes
Not **NULL** (points to valid memory)	less than 0 or greater than **_MAX__TIME64_T**	**EINVAL**	All fields set to -1	No

In the case of the first two error conditions, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **EINVAL**.

Remarks

The **_localtime32_s** function converts a time stored as a time_t value and stores the result in a structure of type **tm**. The **long** value *timer* represents the seconds elapsed since midnight (00:00:00), January 1, 1970, UTC. This value is usually obtained from the **time** function.

_localtime32_s corrects for the local time zone if the user first sets the global environment variable **TZ**. When **TZ** is set, three other environment variables (**_timezone**, **_daylight**, and **_tzname**) are automatically set as well. If the **TZ** variable is not set, **localtime32_s** attempts to use the time zone information specified in the Date/Time application in Control Panel. If this information cannot be obtained, PST8PDT, which signifies the Pacific time zone, is used by default. See _tzset for a description of these variables. **TZ** is a Microsoft extension and not part of the ANSI standard definition of **localtime**.

Note
The target environment should try to determine whether daylight saving time is in effect.

_localtime64_s, which uses the **__time64_t** structure, allows dates to be expressed up through 23:59:59, December 31, 3000, coordinated universal time (UTC), whereas **_localtime32_s** represents dates through 03:14:07 January 19, 2038, UTC.

In Visual C++ 2005, **localtime_s** is an inline function which evaluates to **_localtime64_s**, and **time_t** is equivalent to **__time64_t**. If you need to force the compiler to interpret **time_t** as the old 32-bit **time_t**, you can define **_USE_32BIT_TIME_T**. Doing this will cause **localtime_s** to evaluate to **_localtime32_s**. This is not recommended because your application may fail after January 19, 2038, and it is not allowed on 64-bit platforms.

The fields of the structure type tm store the following values, each of which is an **int**.

tm_sec

Seconds after minute (0 – 59).

tm_min

Minutes after hour (0 – 59).

tm_hour

Hours after midnight (0 – 23).

tm_mday

Day of month (1 – 31).

tm_mon

Month (0 – 11; January = 0).

tm_year

Year (current year minus 1900).

tm_wday

Day of week (0 – 6; Sunday = 0).

tm_yday

Day of year (0 – 365; January 1 = 0).

tm_isdst

Positive value if daylight saving time is in effect; 0 if daylight saving time is not in effect; negative value if status of daylight saving time is unknown. If the **TZ** environment variable is set, the C run-time library assumes rules appropriate to the United States for implementing the calculation of daylight saving time (DST).

Requirements

Routine	Required header	Compatibility
localtime_s	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_localtime32_s	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_localtime64_s	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_localtime_s.c
/* This program uses _time64 to get the current time
 * and then uses _localtime64_s() to convert this time to a structure
 * representing the local time. The program converts the result
 * from a 24-hour clock to a 12-hour clock and determines the
 * proper extension (AM or PM).
 */
```

```
#include <stdio.h>
#include <string.h>
#include <time.h>

int main( void )
{
        struct tm newtime;
        char am_pm[] = "AM";
        __time64_t long_time;
        char timebuf[26];
        errno_t err;

        // Get time as 64-bit integer.
        _time64( &long_time );
        // Convert to local time.
        err = _localtime64_s( &newtime, &long_time );
        if (err)
        {
            printf("Invalid argument to _localtime64_s.");
            exit(1);
        }
        if( newtime.tm_hour > 12 )        // Set up extension.
                strcpy_s( am_pm, sizeof(am_pm), "PM" );
        if( newtime.tm_hour > 12 )        // Convert from 24-hour
                newtime.tm_hour -= 12;    // to 12-hour clock.
        if( newtime.tm_hour == 0 )        // Set hour to 12 if midnight.
                newtime.tm_hour = 12;

        // Convert to an ASCII representation.
        err = asctime_s(timebuf, 26, &newtime);
        if (err)
        {
           printf("Invalid argument to asctime_s.");
           exit(1);
        }
        printf( "%.19s %s\n", timebuf, am_pm );
}
```

Sample Output

```
Fri Apr 25 01:19:27 PM
```

.NET Framework Equivalent

System::DateTime::ToLocalTime

See Also

Reference

Time Management
asctime_s, _wasctime_s
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
_ftime, _ftime32, _ftime64
_gmtime_s, _gmtime32_s, _gmtime64_s
localtime, _localtime32, _localtime64
time, _time32, _time64
_tzset

_lock_file

Locks a file, preventing other processes from accessing the file.

```
void _lock_file(
   FILE* file
);
```

Parameters

file

File handle.

Remarks

The **_lock_file** function locks the file specified by *file*. Locking a file prevents access to the file by other processes. Use _unlock_file to release the lock on the file. Calls to **_lock_file** and **_unlock_file** must be matched in a thread. Attempting to lock a file that is already locked or unlock a file that is not locked can result in a deadlock.

Requirements

Routine	Required header	Compatibility
_lock_file	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_lock_file.c
// This program will Lock a file,
// preventing other processes or threads
// from gaining access to the file until
// it's unlocked.

#include <stdio.h>
#include <stdlib.h>

int main()
{
    FILE    *pFile = NULL;
    char    *fileName = "crt_lock_file.txt";
    char    commandLine[256];

    // Open the file in write mode
    fopen_s(&pFile, fileName ,"w+");
    if (!pFile)
    {
        printf("Error opening file %s!\n", fileName);
        exit(1);
    }

    // Lock the file.
    _lock_file(pFile);
    printf("Locking the file %s.\n", fileName);

    // Add some data to the file
    fprintf(pFile, "The sound of bagpipes in the distant glen...\n");

    printf("\nAttempting to access the file from another process:\n");
    sprintf_s(commandLine, sizeof(commandLine), "type %s", fileName);
    system(commandLine);
```

```
    // Unlock the file
    _unlock_file(pFile);

    // Close the file
    fclose(pFile);

    printf("\nAttempting to access the file again:\n");
    system(commandLine);
}
```

Sample Output

```
Locking the file crt_lock_file.txt.

Attempting to access the file from another process:
The process cannot access the file because it is being used by another process.

Attempting to access the file again:
The sound of bagpipes in the distant glen...
```

.NET Framework Equivalent

System::IO::FileStream::Lock

See Also

Reference

File Handling
_creat, _wcreat
_open, _wopen
_unlock_file

locking

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _locking instead.

_locking

Locks or unlocks bytes of a file.

```
int _locking(
   int fd,
   int mode,
   long nbytes
);
```

Parameters

fd

File descriptor.

mode

Locking action to perform.

nbytes

Number of bytes to lock.

Return Value

_locking returns 0 if successful. A return value of –1 indicates failure, in which case errno is set to one of the following values.

EACCES

Locking violation (file already locked or unlocked).

EBADF

Invalid file descriptor.

EDEADLOCK

Locking violation. Returned when the **_LK_LOCK** or **_LK_RLCK** flag is specified and the file cannot be locked after 10 attempts.

EINVAL

An invalid argument was given to **_locking**.

If the failure is due to a bad parameter, such as an invalid file descriptor, the invalid parameter handler is invoked, as described in Parameter Validation.

Remarks

The **_locking** function locks or unlocks *nbytes* bytes of the file specified by *fd*. Locking bytes in a file prevents access to those bytes by other processes. All locking or unlocking begins at the current position of the file pointer and proceeds for the next *nbytes* bytes. It is possible to lock bytes past end of file.

mode must be one of the following manifest constants, which are defined in Locking.h.

_LK_LOCK

Locks the specified bytes. If the bytes cannot be locked, the program immediately tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, the constant returns an error.

_LK_NBLCK

Locks the specified bytes. If the bytes cannot be locked, the constant returns an error.

_LK_NBRLCK

Same as **_LK_NBLCK**.

_LK_RLCK

Same as **_LK_LOCK**.

_LK_UNLCK

Unlocks the specified bytes, which must have been previously locked.

Multiple regions of a file that do not overlap can be locked. A region being unlocked must have been previously locked. **_locking** does not merge adjacent regions; if two locked regions are adjacent, each region must be unlocked separately. Regions should be locked only briefly and should be unlocked before closing a file or exiting the program.

Requirements

Routine	Required header	Optional header	Compatibility
_locking	<io.h> and <sys/locking.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_locking.c
/* This program opens a file with sharing. It locks
 * some bytes before reading them, then unlocks them. Note that the
 * program works correctly only if the file exists.
 */

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/locking.h>
#include <share.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <io.h>

int main( void )
{
   int  fh, numread;
   char buffer[40];

   /* Quit if can't open file or system doesn't
    * support sharing.
    */
   errno_t err = _sopen_s( &fh, "crt_locking.txt", _O_RDONLY, _SH_DENYNO,
                           _S_IREAD | _S_IWRITE );
   printf( "%d %d\n", err, fh );
   if( err != 0 )
      exit( 1 );

   /* Lock some bytes and read them. Then unlock. */
   if( _locking( fh, LK_NBLCK, 30L ) != -1 )
   {
      long lseek_ret;
      printf( "No one can change these bytes while I'm reading them\n" );
      numread = _read( fh, buffer, 30 );
      buffer[30] = '\0';
      printf( "%d bytes read: %.30s\n", numread, buffer );
      lseek_ret = _lseek( fh, 0L, SEEK_SET );
      _locking( fh, LK_UNLCK, 30L );
      printf( "Now I'm done. Do what you will with them\n" );
   }
   else
```

```
        perror( "Locking failed\n" );

    _close( fh );
  }
```

Input: crt_locking.txt

```
  The first thirty bytes of this file will be locked.
```

Sample Output

```
  No one can change these bytes while I'm reading them
  30 bytes read: The first thirty bytes of this
  Now I'm done. Do what you will with them
```

.NET Framework Equivalent

System::IO::FileStream::Lock

See Also

Reference

File Handling

_creat, _wcreat

_open, _wopen

log, logf, log10, log10f

Calculates logarithms.

```
double log(
   double x
);
float log(
   float x
);  // C++ only
long double log(
   long double x
);  // C++ only
float logf(
   float x
);
double log10(
   double x
);
float log10(
   float x
);  // C++ only
long double log10(
   long double x
);  // C++ only
float log10f (
   float x
);
```

Parameters

x

Value whose logarithm is to be found.

Return Value

The **log** functions return the natural logarithm (base e) of *x* if successful. The log10 functions return the base-10 logarithm. If *x* is negative, these functions return an indefinite, by default. If *x* is 0, they return INF (infinite).

Input	SEH Exception	Matherr Exception
± QNAN,IND	none	_DOMAIN
± 0	ZERODIVIDE	_SING
x < 0	INVALID	_DOMAIN

log and **log10** has an implementation that uses Streaming SIMD Extensions 2 (SSE2). See _set_SSE2_enable for information and restrictions on using the SSE2 implementation.

Remarks

C++ allows overloading, so you can call overloads of **log** and **log10**. In a C program, **log** and **log10** always take and return a double.

Requirements

Routine	Required header	Compatibility

log, **logf**, **log10**, **log10f**	<math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_log.c
/* This program uses log and log10
 * to calculate the natural logarithm and
 * the base-10 logarithm of 9,000.
 */

#include <math.h>
#include <stdio.h>

int main( void )
{
   double x = 9000.0;
   double y;

   y = log( x );
   printf( "log( %.2f ) = %f\n", x, y );
   y = log10( x );
   printf( "log10( %.2f ) = %f\n", x, y );
}
```

Output

```
log( 9000.00 ) = 9.104980
log10( 9000.00 ) = 3.954243
```

To generate logarithms for other bases, use the mathematical relation: log base b of a == natural log (a) / natural log (b).

```
// logbase.cpp
#include <math.h>
#include <stdio.h>

double logbase(double a, double base)
{
   return log(a) / log(base);
}

int main()
{
   double x = 65536;
   double result;

   result = logbase(x, 2);
   printf("Log base 2 of %lf is %lf\n", x, result);
}
```

Output

```
Log base 2 of 65536.000000 is 16.000000
```

.NET Framework Equivalent

- System::Math::Log
- System::Math::Log10

See Also

Reference

Floating-Point Support
exp, expf
_matherr
pow, powf

_logb

Extracts an exponential value of a double-precision floating-point argument.

```
double _logb(
   double x
);
```

Parameters

x

Double-precision floating-point value.

Return Value

_logb returns the unbiased exponential value of *x*.

Remarks

The **_logb** function extracts the exponential value of its double-precision floating-point argument *x*, as though *x* were represented with infinite range. If the argument *x* is denormalized, it is treated as if it were normalized.

Input	SEH exception	Matherr exception
± QNAN,IND	None	_DOMAIN
± 0	ZERODIVIDE	_SING

Requirements

Routine	Required header	Compatibility
_logb	<float.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support

frexp

longjmp

Restores stack environment and execution locale.

```
void longjmp(
   jmp_buf env,
   int value
);
```

Parameters

env

Variable in which environment is stored.

value

Value to be returned to **setjmp** call.

Remarks

The **longjmp** function restores a stack environment and execution locale previously saved in *env* by **setjmp**. **setjmp** and **longjmp** provide a way to execute a nonlocal **goto**; they are typically used to pass execution control to error-handling or recovery code in a previously called routine without using the normal call and return conventions.

A call to **setjmp** causes the current stack environment to be saved in *env*. A subsequent call to **longjmp** restores the saved environment and returns control to the point immediately following the corresponding **setjmp** call. Execution resumes as if *value* had just been returned by the **setjmp** call. The values of all variables (except register variables) that are accessible to the routine receiving control contain the values they had when **longjmp** was called. The values of register variables are unpredictable. The value returned by **setjmp** must be nonzero. If *value* is passed as 0, the value 1 is substituted in the actual return.

Call **longjmp** before the function that called **setjmp** returns; otherwise the results are unpredictable.

Observe the following restrictions when using **longjmp**:

- Do not assume that the values of the register variables will remain the same. The values of register variables in the routine calling **setjmp** may not be restored to the proper values after **longjmp** is executed.
- Do not use **longjmp** to transfer control out of an interrupt-handling routine unless the interrupt is caused by a floating-point exception. In this case, a program may return from an interrupt handler via **longjmp** if it first reinitializes the floating-point math package by calling **_fpreset**.

  Note Be careful when using **setjmp** and **longjmp** in C++ programs. Because these functions do not support C++ object semantics, it is safer to use the C++ exception-handling mechanism.

For more information, see Using setjmp and longjmp.

Requirements

Routine	Required header	Compatibility
longjmp	<setjmp.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

See the example for _fpreset.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Process and Environment Control

setjmp

_lrotl, _lrotr

Rotates bits to the left (**_lrotl**) or right (**_lrotr**).

```
unsigned long _lrotl(
   unsigned long value,
   int shift
);
unsigned long _lrotr(
   unsigned long value,
   int shift
);
```

Parameters

value

Value to be rotated.

shift

Number of bits to shift *value*.

Return Value

Both functions return the rotated value. There is no error return.

Remarks

The **_lrotl** and **_lrotr** functions rotate *value* by *shift* bits. **_lrotl** rotates the value left. **_lrotr** rotates the value right. Both functions wrap bits rotated off one end of *value* to the other end.

Requirements

Routine	Required header	Compatibility
_lrotl	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_lrotr	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_lrot.c

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   unsigned long val = 0x0fac35791;

   printf( "0x%8.8lx rotated left eight times is 0x%8.8lx\n",
            val, _lrotl( val, 8 ) );
   printf( "0x%8.8lx rotated right four times is 0x%8.8lx\n",
            val, _lrotr( val, 4 ) );
```

```
}
```

Output

```
0xfac35791 rotated left eight times is 0xc35791fa
0xfac35791 rotated right four times is 0x1fac3579
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support

_rotl, _rotl64, _rotr, _rotr64

lsearch

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _lsearch or security-enhanced _lsearch_s instead.

_lsearch

Performs a linear search for a value; adds to end of list if not found. A more secure version of this function is available; see _lsearch_s.

```
void *_lsearch(
   const void *key,
   void *base,
   unsigned int *num,
   unsigned int width,
   int (__cdecl *compare)(const void *, const void *)
);
```

Parameters

key

Object to search for.

base

Pointer to the base of array to be searched.

num

Number of elements.

width

Width of each array element.

compare

Pointer to the comparison routine. The first parameter is a pointer to the key for search. The second parameter is a pointer to an array element to be compared with the key.

Return Value

If the key is found, **_lsearch** returns a pointer to the element of the array at *base* that matches *key*. If the key is not found, **_lsearch** returns a pointer to the newly added item at the end of the array.

Remarks

The **_lsearch** function performs a linear search for the value *key* in an array of *num* elements, each of *width* bytes. Unlike **bsearch**, **_lsearch** does not require the array to be sorted. If *key* is not found, **_lsearch** adds it to the end of the array and increments *num*.

The *compare* argument is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. **_lsearch** calls the *compare* routine one or more times during the search, passing pointers to two array elements on each call. *compare* must compare the elements and return either nonzero (meaning the elements are different) or 0 (meaning the elements are identical).

This function validates its parameters. If *compare*, *key* or *num* is **NULL**, or if *base* is NULL and **num* is nonzero, or if *width* is less than zero, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **NULL**.

Requirements

Routine	Required header	Compatibility
_lsearch	<search.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_lsearch.c
```

```
#include <search.h>
#include <string.h>
#include <stdio.h>

int compare( const void *arg1, const void *arg2 );

int main(void)
{
   char * wordlist[4] = { "hello", "thanks", "bye" };
                            // leave room to grow...
   int n = 3;
   char **result;
   char *key = "extra";
   int i;

   printf( "wordlist before _lsearch:" );
   for( i=0; i<n; ++i ) printf( " %s", wordlist[i] );
   printf( "\n" );

   result = (char **)_lsearch( &key, wordlist,
                      &n, sizeof(char *), compare );

   printf( "wordlist after _lsearch:" );
   for( i=0; i<n; ++i ) printf( " %s", wordlist[i] );
   printf( "\n" );
}

int compare(const void *arg1, const void *arg2 )
{
   return( _stricmp( * (char**)arg1, * (char**)arg2 ) );
}
```

Output

```
wordlist before _lsearch: hello thanks bye
wordlist after _lsearch: hello thanks bye extra
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Searching and Sorting
bsearch
_lfind
_lsearch_s

_lsearch_s

Performs a linear search for a value. A version of _lsearch with security enhancements as described in Security Enhancements in the CRT.

```
void *_lsearch_s(
   const void *key,
   void *base,
   unsigned int *num,
   size_t size,
   int (__cdecl *compare)(void *, const void *, const void *),
   void * context
);
```

Parameters

key

Object to search for.

base

Pointer to the base of array to be searched.

num

Number of elements.

size

Size of each array element in bytes.

compare

Pointer to the comparison routine. The second parameter is a pointer to the key for search. The third parameter is a pointer to an array element to be compared with the key.

context

A pointer to an object that might be accessed in the comparison function.

Return Value

If *key* is found, **_lsearch_s** returns a pointer to the element of the array at *base* that matches *key*. If *key* is not found, **_lsearch_s** returns a pointer to the newly added item at the end of the array.

If invalid parameters are passed to the function, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, then **errno** is set to **EINVAL** and the function returns **NULL**. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Error Conditions

key	***base***	***compare***	***num***	***size***	**errno**
NULL	any	any	any	any	**EINVAL**
any	**NULL**	any	!= 0	any	**EINVAL**
any	any	any	any	zero	**EINVAL**
any	any	**NULL**	an	any	**EINVAL**

Remarks

The **_lsearch_s** function performs a linear search for the value *key* in an array of *num* elements, each of *width* bytes. Unlike **bsearch_s**, **_lsearch_s** does not require the array to be sorted. If *key* is not found, then **_lsearch_s** adds it to the end of the array and increments *num*.

The *compare* function is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. The *compare* function also takes the pointer to the context as the first argument. **_lsearch_s** calls *compare* one or more times during the search, passing pointers to two array elements on each call. *compare* must compare the elements and then return either nonzero (meaning the elements are different) or 0 (meaning the elements are identical).

The *context* pointer can be useful if the searched data structure is part of an object and the *compare* function needs to access members of the object. For example, code in the *compare* function can cast the void pointer into the appropriate object type and access members of that object. The addition of the *context* pointer makes **_lsearch_s** more secure because additional context can be used to avoid reentrancy bugs associated with using static variables to make data available to the *compare* function.

Requirements

Routine	Required header	Compatibility
_lsearch_s	<search.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Searching and Sorting
bsearch_s
_lfind_s
_lsearch

lseek

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _lseek instead.

_lseek, _lseeki64

Moves a file pointer to the specified location.

```
long _lseek(
   int fd,
   long offset,
   int origin
);
__int64 _lseeki64(
   int fd,
   __int64 offset,
   int origin
);
```

Parameters

fd

File descriptor referring to an open file.

offset

Number of bytes from *origin*.

origin

Initial position.

Return Value

_lseek returns the offset, in bytes, of the new position from the beginning of the file. **_lseeki64** returns the offset in a 64-bit integer. The function returns –1L to indicate an error. If passed an invalid parameter, such as a bad file descriptor, or the value for *origin* is invalid or the position specified by *offset* is before the beginning of the file, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EBADF** and return -1L. On devices incapable of seeking (such as terminals and printers), the return value is undefined.

For more information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_lseek** function moves the file pointer associated with *fd* to a new location that is *offset* bytes from *origin*. The next operation on the file occurs at the new location. The *origin* argument must be one of the following constants, which are defined in Stdio.h.

SEEK_SET

Beginning of the file.

SEEK_CUR

Current position of the file pointer.

SEEK_END

End of file.

You can use **_lseek** to reposition the pointer anywhere in a file or beyond the end of the file.

Requirements

Routine	Required header	Compatibility
_lseek	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_lseeki64	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_lseek.c
/* This program first opens a file named lseek.txt.
 * It then uses _lseek to find the beginning of the file,
 * to find the current position in the file, and to find
 * the end of the file.
 */

#include <io.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <share.h>

int main( void )
{
   int fh;
   long pos;               /* Position of file pointer */
   char buffer[10];

   _sopen_s( &fh, "crt_lseek.c_input", _O_RDONLY, _SH_DENYNO, 0 );

   /* Seek the beginning of the file: */
   pos = _lseek( fh, 0L, SEEK_SET );
   if( pos == -1L )
      perror( "_lseek to beginning failed" );
   else
      printf( "Position for beginning of file seek = %ld\n", pos );

   /* Move file pointer a little */
    _read( fh, buffer, 10 );

   /* Find current position: */
   pos = _lseek( fh, 0L, SEEK_CUR );
   if( pos == -1L )
      perror( "_lseek to current position failed" );
   else
      printf( "Position for current position seek = %ld\n", pos );

   /* Set the end of the file: */
   pos = _lseek( fh, 0L, SEEK_END );
   if( pos == -1L )
      perror( "_lseek to end failed" );
   else
      printf( "Position for end of file seek = %ld\n", pos );

   _close( fh );
}
```

Input: crt_lseek.c_input

```
Line one.
Line two.
Line three.
Line four.
Line five.
```

Output

```
Position for beginning of file seek = 0
Position for current position seek = 10
Position for end of file seek = 57
```

See Also

Reference

Low-Level I/O

fseek, _fseeki64

_tell, _telli64

ltoa

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _ltoa or security-enhanced _ltoa_s instead.

_ltoa, _ltow

Converts a long integer to a string. These functions are deprecated because more secure versions are available; see _ltoa_s, _ltow_s.

```
char *_ltoa(
   long value,
   char *str,
   int radix
);
wchar_t *_ltow(
   long value,
   wchar_t *str,
   int radix
);
template <size_t size>
char *_ltoa(
   long value,
   char (&str)[size],
   int radix
); // C++ only
template <size_t size>
wchar_t *_ltow(
   long value,
   wchar_t (&str)[size],
   int radix
); // C++ only
```

Parameters

value

Number to be converted.

str

String result.

radix

Base of *value*.

Return Value

Each of these functions returns a pointer to *str*. There is no error return.

Remarks

The **_ltoa** function converts the digits of *value* to a null-terminated character string and stores the result (up to 33 bytes) in *str*. The *radix* argument specifies the base of *value*, which must be in the range 2 – 36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (–). **_ltow** is a wide-character version of **_ltoa**; the second argument and return value of **_ltow** are wide-character strings. Each of these functions is Microsoft-specific.

Security Note
To prevent buffer overruns, ensure that the *str* buffer is large enough to hold the converted digits plus the trailing null-character and a sign character.

In C++, these functions have template overloads. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_ltot	_ltoa	_ltoa	_ltow

Requirements

Routine	Required header	Compatibility
_ltoa	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ltow	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example for _itoa.

.NET Framework Equivalent

System::Convert::ToString

See Also

Reference

Data Conversion
_itoa, _i64toa, _ui64toa, _itow, _i64tow, _ui64tow
_ultoa, _ultow

_ltoa_s, _ltow_s

Converts a long integer to a string. These are versions of _ltoa, _ltow with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _ltoa_s(
    long value,
    char *str,
    size_t sizeOfstr,
    int radix
);
errno_t _ltow_s(
    long value,
    wchar_t *str,
    size_t sizeOfstr,
    int radix
);
template <size_t size>
errno_t _ltoa_s(
    long value,
    char (&str)[size],
    int radix
); // C++ only
template <size_t size>
errno_t _ltow_s(
    long value,
    wchar_t (&str)[size],
    int radix
); // C++ only
```

Parameters

value

Number to be converted.

str

Buffer for the resulting string.

sizeOfstr

Size of the *str* in bytes for **_ltoa_s** or words for **_ltow_s**.

radix

Base of *value*.

Return Value

Zero if the function was successful or an error code.

Remarks

The **_ltoa_s** function converts the digits of *value* to a null-terminated character string and stores the result (up to 33 bytes) in *str*. The *radix* argument specifies the base of *value*, which must be in the range 2 – 36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (–). **_ltow_s** is a wide character version of **_ltoa_s**; the second argument of **_ltow_s** is a wide character strings.

If *str* is a **NULL** pointer or *sizeOfstr* is less than or equal to zero, these functions invoke an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL** or if the *value* or *str* out of range of a long integer, these functions return a -1 and set the **errno** to **ERANGE**.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_ltot_s	**_ltoa_s**	**_ltoa_s**	**_ltow_s**

Requirements

Routine	Required header	Compatibility
_ltoa_s	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ltow_s	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::Convert::ToString

See Also

Reference

Data Conversion
_itoa, _i64toa, _ui64toa, _itow, _i64tow, _ui64tow
_ultoa, _ultow
_ultoa_s, _ultow_s

_makepath, _wmakepath

Create a path name from components. These functions are deprecated because more secure versions are available; see _makepath_s, _wmakepath_s.

```
void _makepath(
   char *path,
   const char *drive,
   const char *dir,
   const char *fname,
   const char *ext
);
void _wmakepath(
   wchar_t *path,
   const wchar_t *drive,
   const wchar_t *dir,
   const wchar_t *fname,
   const wchar_t *ext
);
```

Parameters

path

Full path buffer. **_makepath** does not check that *path* does not exceed **_MAX_PATH**.

drive

Drive letter.

dir

Directory path.

fname

File name.

ext

File name extension.

Remarks

The **_makepath** function creates a single path and stores it in *path*. The path might include a drive letter, directory path, file name, and file name extension. **_wmakepath** is a wide-character version of **_makepath**; the arguments to **_wmakepath** are wide-character strings. **_wmakepath** and **_makepath** behave identically otherwise.

Security Note
Use a null-terminated string. The null-terminated string must not exceed the size of the destination buffer. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tmakepath	**_makepath**	**_makepath**	**_wmakepath**

The following arguments point to buffers containing the path elements.

drive

Contains a letter (A, B, and so on) corresponding to the desired drive and an optional trailing colon. **_makepath** inserts the colon automatically in the composite path if it is missing. If *drive* is a null character or an empty string, no drive letter and colon appear in the composite *path* string.

dir

Contains the path of directories, not including the drive designator or the actual file name. The trailing slash is optional, and either a forward slash (/) or a backslash (\) or both might be used in a single *dir* argument. If a trailing slash (/ or \) is not specified, it is inserted automatically. If *dir* is a null character or an empty string, no slash is inserted in the composite *path* string.

fname

Contains the base file name without any file name extensions. If *fname* is **NULL** or points to an empty string, no file name is inserted in the composite *path* string.

ext

Contains the actual file name extension, with or without a leading period (.). **_makepath** inserts the period automatically if it does not appear in *ext*. If *ext* is a null character or an empty string, no period is inserted in the composite *path* string.

The *path* argument must point to an empty buffer large enough to hold the complete path. Although there are no size limits on any of the fields that constitute *path*, the composite *path* must be no larger than the **_MAX_PATH** constant, defined in Stdlib.h. **_MAX_PATH** might be larger than the current operating-system version can handle.

In Visual C++ 2005, these functions validate the *path* parameters. If path is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. In addition, **errno** is set to **EINVAL**. **NULL** values are allowed for the other parameters.

Requirements

Routine	Required header	Compatibility
_makepath	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003, Windows Server 2003
_wmakepath	<stdlib.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_makepath.c
// compile with: /W3
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char path_buffer[_MAX_PATH];
   char drive[_MAX_DRIVE];
   char dir[_MAX_DIR];
   char fname[_MAX_FNAME];
   char ext[_MAX_EXT];

   _makepath( path_buffer, "c", "\\sample\\crt\\", "makepath", "c" ); // C4996
   // Note: _makepath is deprecated; consider using _makepath_s instead
   printf( "Path created with _makepath: %s\n\n", path_buffer );
   _splitpath( path_buffer, drive, dir, fname, ext ); // C4996
   // Note: _splitpath is deprecated; consider using _splitpath_s instead
   printf( "Path extracted with _splitpath:\n" );
   printf( "  Drive: %s\n", drive );
   printf( "  Dir: %s\n", dir );
   printf( "  Filename: %s\n", fname );
   printf( "  Ext: %s\n", ext );
}
```

Output

```
Path created with _makepath: c:\sample\crt\makepath.c

Path extracted with _splitpath:
```

```
Drive: c:
Dir: \sample\crt\
Filename: makepath
Ext: .c
```

.NET Framework Equivalent

System::IO::File::Create

See Also

Reference

File Handling

_fullpath, _wfullpath

_splitpath, _wsplitpath

_makepath_s, _wmakepath_s

Create a path name from components. These are versions of _makepath, _wmakepath with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _makepath_s(
   char *path,
   size_t sizeInCharacters,
   const char *drive,
   const char *dir,
   const char *fname,
   const char *ext
);
errno_t _wmakepath_s(
   wchar_t *path,
   size_t sizeInCharacters,
   const wchar_t *drive,
   const wchar_t *dir,
   const wchar_t *fname,
   const wchar_t *ext
);
template <size_t size>
errno_t _makepath_s(
   char (&path)[size],
   const char *drive,
   const char *dir,
   const char *fname,
   const char *ext
); // C++ only
template <size_t size>
errno_t _wmakepath_s(
   wchar_t (&path)[size],
   const wchar_t *drive,
   const wchar_t *dir,
   const wchar_t *fname,
   const wchar_t *ext
); // C++ only
```

Parameters

[out] *path*

Full path buffer.

[in] *sizeInCharacters*

Size of the buffer in characters.

[in] *drive*

Drive letter.

[in] *dir*

Directory path.

[in] *fname*

File name.

[in] *ext*

File name extension.

Return Value

Zero if successful; an error code on failure.

Error Conditions

path	*sizeInCharacters*	Return	Contents of *path*
NULL	any	**EINVAL**	not modified
any	<= 0	**EINVAL**	not modified

If any of the above error conditions occurs, these functions invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the functions returns **EINVAL. NULL** is allowed for the parameters *drive*, *fname*, and *ext*. For information about the behavior when these parameters are null pointers or empty strings, see the Remarks section.

Remarks

The **_makepath_s** function creates a single path and stores it in *path*. The path might include a drive letter, directory path, file name, and file name extension. **_wmakepath_s** is a wide-character version of **_makepath_s**; the arguments to **_wmakepath_s** are wide-character strings. **_wmakepath_s** and **_makepath_s** behave identically otherwise.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tmakepath_s	**_makepath_s**	**_makepath_s**	**_wmakepath_s**

The following arguments point to buffers containing the path elements.

drive

Contains a letter (A, B, and so on) corresponding to the desired drive and an optional trailing colon. **_makepath_s** inserts the colon automatically in the composite path if it is missing. If *drive* is a null character or an empty string, no drive letter and colon appear in the composite *path* string.

dir

Contains the path of directories, not including the drive designator or the actual file name. The trailing slash is optional, and either a forward slash (/) or a backslash (\) or both might be used in a single *dir* argument. If a trailing slash (/ or \) is not specified, it is inserted automatically. If *dir* is a null character or an empty string, no slash is inserted in the composite *path* string.

fname

Contains the base file name without any extensions. If *fname* is **NULL** or points to an empty string, no file name is inserted in the composite *path* string.

ext

Contains the actual file name extension, with or without a leading period (.). **_makepath_s** inserts the period automatically if it does not appear in *ext*. If *ext* is a null character or an empty string, no period is inserted in the composite *path* string.

The *path* argument must point to an empty buffer large enough to hold the complete path. Although there are no size limits on any of the fields that constitute *path*, the composite *path* must be no larger than the **_MAX_PATH** constant, defined in Stdlib.h. **_MAX_PATH** might be larger than the current operating-system version can handle.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Requirements

Routine	Required header	Compatibility
_makepath_s	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_wmakepath_s	<stdlib.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_makepath_s.c

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char path_buffer[_MAX_PATH];
   char drive[_MAX_DRIVE];
   char dir[_MAX_DIR];
   char fname[_MAX_FNAME];
   char ext[_MAX_EXT];
   errno_t err;

   err = _makepath_s( path_buffer, _MAX_PATH, "c", "\\sample\\crt\\",
                      "crt_makepath_s", "c" );
   if (err != 0)
   {
      printf("Error creating path. Error code %d.\n", err);
      exit(1);
   }
   printf( "Path created with _makepath_s: %s\n\n", path_buffer );
   err = _splitpath_s( path_buffer, drive, _MAX_DRIVE, dir, _MAX_DIR, fname,
                       _MAX_FNAME, ext, _MAX_EXT );
   if (err != 0)
   {
      printf("Error splitting the path. Error code %d.\n", err);
      exit(1);
   }
   printf( "Path extracted with _splitpath_s:\n" );
   printf( "  Drive: %s\n", drive );
   printf( "  Dir: %s\n", dir );
   printf( "  Filename: %s\n", fname );
   printf( "  Ext: %s\n", ext );
}
```

Output

```
Path created with _makepath_s: c:\sample\crt\crt_makepath_s.c

Path extracted with _splitpath_s:
  Drive: c:
  Dir: \sample\crt\
  Filename: crt_makepath_s
  Ext: .c
```

.NET Framework Equivalent

System::IO::File::Create

See Also

Reference

File Handling
_fullpath, _wfullpath
_splitpath_s, _wsplitpath_s
_makepath, _wmakepath

malloc

Allocates memory blocks.

```
void *malloc(
   size_t size
);
```

Parameters

size

Bytes to allocate.

Return Value

malloc returns a void pointer to the allocated space or **NULL** if there is insufficient memory available. To return a pointer to a type other than **void**, use a type cast on the return value. The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. If size is 0, **malloc** allocates a zero-length item in the heap and returns a valid pointer to that item. Always check the return from **malloc**, even if the amount of memory requested is small.

Remarks

The **malloc** function allocates a memory block of at least *size* bytes. The block may be larger than *size* bytes because of space required for alignment and maintenance information.

In Visual C++ 2005, **malloc** sets **errno** to **ENOMEM** if a memory allocation fails or if the amount of memory requested exceeds **_HEAP_MAXREQ**. For information on this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

The startup code uses **malloc** to allocate storage for the **_environ**, **envp**, and **argv** variables. The following functions and their wide-character counterparts also call **malloc**:

calloc	fscanf	_getw	setvbuf
_exec functions	fseek	_popen	_spawn functions
fgetc	fsetpos	printf	_strdup
_fgetchar	_fullpath	putc	system
fgets	fwrite	putchar	_tempnam
fprintf	getc	_putenv	ungetc
fputc	getchar	puts	vfprintf
_fputchar	_getcwd	_putw	vprintf
fputs	_getdcwd	scanf	
fread	gets	_searchenv	

The C++ _set_new_mode function sets the new handler mode for **malloc**. The new handler mode indicates whether, on failure, **malloc** is to call the new handler routine as set by _set_new_handler. By default, **malloc** does not call the new handler routine on failure to allocate memory. You can override this default behavior so that, when **malloc** fails to allocate memory, **malloc** calls the new handler routine in the same way that the **new** operator does when it fails for the same reason. To override the default, call

```
_set_new_mode(1)
```

early in your program, or link with NEWMODE.OBJ (see Link Options).

When the application is linked with a debug version of the C run-time libraries, **malloc** resolves to _malloc_dbg. For more information about how the heap is managed during the debugging process, see The CRT Debug Heap.

malloc is marked **__declspec(noalias)** and **__declspec(restrict)**, meaning that the function is guaranteed not to modify global variables, and that the pointer returned is not aliased. For more information, see noalias and restrict.

Requirements

Routine	Required header	Compatibility
malloc	<stdlib.h> and <malloc.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_malloc.c
// This program allocates memory with
// malloc, then frees the memory with free.


#include <stdlib.h>   // For _MAX_PATH definition
#include <stdio.h>
#include <malloc.h>

int main( void )
{
   char *string;

   // Allocate space for a path name
   string = malloc( _MAX_PATH );

   // In a C++ file, explicitly cast malloc's return.  For example,
   // string = (char *)malloc( _MAX_PATH );

   if( string == NULL )
      printf( "Insufficient memory available\n" );
   else
   {
      printf( "Memory space allocated for path name\n" );
      free( string );
      printf( "Memory freed\n" );
   }
}
```

Output

```
Memory space allocated for path name
Memory freed
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation
calloc
free
realloc

_malloc_dbg

Allocates a block of memory in the heap with additional space for a debugging header and overwrite buffers (debug version only).

```
void *_malloc_dbg(
   size_t size,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

size

Requested size of the memory block (in bytes).

blockType

Requested type of the memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

filename

Pointer to the name of the source file that requested the allocation operation or NULL.

linenumber

Line number in the source file where the allocation operation was requested or NULL.

The *filename* and *linenumber* parameters are only available when **_malloc_dbg** has been called explicitly or the _CRTDBG_MAP_ALLOC preprocessor constant has been defined.

Return Value

On successful completion, this function returns a pointer to the user portion of the allocated memory block, calls the new handler function, or returns NULL. For a complete description of the return behavior, see the following Remarks section. For more information about how the new handler function is used, see the malloc function.

Remarks

_malloc_dbg is a debug version of the malloc function. When _DEBUG is not defined, each call to **_malloc_dbg** is reduced to a call to **malloc**. Both **malloc** and **_malloc_dbg** allocate a block of memory in the base heap, but **_malloc_dbg** offers several debugging features: buffers on either side of the user portion of the block to test for leaks, a block type parameter to track specific allocation types, and *filename/linenumber* information to determine the origin of allocation requests.

_malloc_dbg allocates the memory block with slightly more space than the requested *size*. The additional space is used by the debug heap manager to link the debug memory blocks and to provide the application with debug header information and overwrite buffers. When the block is allocated, the user portion of the block is filled with the value 0xCD and each of the overwrite buffers are filled with 0xFD.

In Visual C++ 2005, **_malloc_dbg** sets **errno** to **ENOMEM** if a memory allocation fails or if the amount of memory needed (including the overhead mentioned previously) exceeds **_HEAP_MAXREQ**. For information about this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see Memory Management and the Debug Heap.

For information about the allocation block types and how they are used, see Types of Blocks on the Debug Heap.

Requirements

Routine	Required header	Compatibility
_malloc_dbg	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

For a sample of how to use **_malloc_dbg**, see crt_dbg1.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines
malloc
_calloc_dbg
_calloc_dbg

_malloca

Allocates memory on the stack. This is a version of _alloca with security enhancements as described in Security Enhancements in the CRT.

```
void *_malloca(
   size_t size
);
```

Parameters

size

Bytes to be allocated from the stack.

Return Value

The **_malloca** routine returns a **void** pointer to the allocated space, which is guaranteed to be suitably aligned for storage of any type of object. If *size* is 0, **_malloca** allocates a zero-length item and returns a valid pointer to that item.

A stack overflow exception is generated if the space cannot be allocated. The stack overflow exception is not a C++ exception; it is a structured exception. Instead of using C++ exception handling, you must use Structured Exception Handling (SEH).

Remarks

_malloca allocates *size* bytes from the program stack or the heap if the request exceeds a certain size in bytes given by **_ALLOCA_S_THRESHOLD**. The difference between **_malloca** and **_alloca** is that **_alloca** always allocates on the stack, regardless of the size. Unlike **_alloca**, which does not require or permit a call to **free** to free the memory so allocated, **_malloca** requires the use of _freea to free memory. In debug mode, **_malloca** always allocates memory from the heap.

There are restrictions to explicitly calling **_malloca** in an exception handler (EH). EH routines that run on x86-class processors operate in their own memory frame: They perform their tasks in memory space that is not based on the current location of the stack pointer of the enclosing function. The most common implementations include Windows NT structured exception handling (SEH) and C++ catch clause expressions. Therefore, explicitly calling **_malloca** in any of the following scenarios results in program failure during the return to the calling EH routine:

- Windows NT SEH exception filter expression: **__except** (**_malloca ()**)
- Windows NT SEH final exception handler: **__finally** {**_malloca ()** }
- C++ EH catch clause expression

However, **_malloca** can be called directly from within an EH routine or from an application-supplied callback that gets invoked by one of the EH scenarios previously listed.

Security Note
In Windows XP, if **_malloca** is called inside a try/catch block, you must call _resetstkoflw in the catch block.

In addition to the above restrictions, when using the /clr (Common Language Runtime Compilation) option, **_malloca** cannot be used in **__except** blocks. For more information, see /clr Restrictions.

Requirements

Routine	**Required header**	**Compatibility**
_malloca	<malloc.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

```
// crt_malloca_simple.c
#include <stdio.h>
#include <malloc.h>
```

```
void Fn()
{
   char * buf = _malloca( 100 );
   // do something with buf
   _freea( buf );
}

int main()
{
   Fn();
}
```

```
// crt_malloca_exception.c
// This program demonstrates the use of
// _malloca and trapping any exceptions
// that may occur.

#include <windows.h>
#include <stdio.h>
#include <malloc.h>

int main()
{
    int     size;
    int     numberRead = 0;
    int     errcode = 0;
    void    *p = NULL;
    void    *pMarker = NULL;

    while (numberRead == 0)
    {
        printf_s("Enter the number of bytes to allocate "
                 "using _malloca: ");
        numberRead = scanf_s("%d", &size);
    }

    // Do not use try/catch for _malloca,
    // use __try/__except, since _malloca throws
    // Structured Exceptions, not C++ exceptions.

    __try
    {
        if (size > 0)
        {
            p =  _malloca( size );
        }
        else
        {
            printf_s("Size must be a positive number.");
        }
        _freea( p );
    }

    // Catch any exceptions that may occur.
    __except( GetExceptionCode() == STATUS_STACK_OVERFLOW )
    {
        printf_s("_malloca failed!\n");

        // If the stack overflows, use this function to restore.
        errcode = _resetstkoflw();
        if (errcode)
        {
            printf("Could not reset the stack!");
            _exit(1);
        }
    };
}
```

Input

```
1000
```

Sample Output

```
Enter the number of bytes to allocate using _malloca: 1000
Allocated 1000 bytes of stack at 0x0012FAC4
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation
calloc
malloc
realloc
_resetstkoflw

_matherr

Handles math errors.

```
int _matherr(
   struct _exception *except
);
```

Parameters

except

Pointer to the structure containing error information.

Return Value

_matherr returns 0 to indicate an error or a nonzero value to indicate success. If **_matherr** returns 0, an error message can be displayed and **errno** is set to an appropriate error value. If **_matherr** returns a nonzero value, no error message is displayed and **errno** remains unchanged.

For more information about return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_matherr** function processes errors generated by the floating-point functions of the math library. These functions call **_matherr** when an error is detected.

For special error handling, you can provide a different definition of **_matherr**. If you use the dynamically linked version of the C run-time library (Msvcr80.dll), you can replace the default **_matherr** routine in a client executable with a user-defined version. However, you cannot replace the default **_matherr** routine in a DLL client of Msvcr80.dll.

When an error occurs in a math routine, **_matherr** is called with a pointer to an **_exception** type structure (defined in Math.h) as an argument. The **_exception** structure contains the following elements.

int type

Exception type.

char *name

Name of the function where error occurred.

double arg1, **arg2**

First and second (if any) arguments to the function.

double retval

Value to be returned by the function.

The **type** specifies the type of math error. It is one of the following values, defined in Math.h.

_DOMAIN

Argument domain error.

_SING

Argument singularity.

_OVERFLOW

Overflow range error.

_PLOSS

Partial loss of significance.

_TLOSS

Total loss of significance.

_UNDERFLOW

The result is too small to be represented. (This condition is not currently supported.)

The structure member **name** is a pointer to a null-terminated string containing the name of the function that caused the error. The structure members **arg1** and **arg2** specify the values that caused the error. (If only one argument is given, it is stored in **arg1**.)

The default return value for the given error is **retval**. If you change the return value, it must specify whether an error actually occurred.

Requirements

Routine	Required header	Compatibility
_matherr	<math.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_matherr.c
/* illustrates writing an error routine for math
 * functions. The error function must be:
 *      _matherr
 */

#include <math.h>
#include <string.h>
#include <stdio.h>

int main()
{
    /* Do several math operations that cause errors. The _matherr
     * routine handles _DOMAIN errors, but lets the system handle
     * other errors normally.
     */
    printf( "log( -2.0 ) = %e\n", log( -2.0 ) );
    printf( "log10( -5.0 ) = %e\n", log10( -5.0 ) );
    printf( "log( 0.0 ) = %e\n", log( 0.0 ) );
}

/* Handle several math errors caused by passing a negative argument
 * to log or log10 (_DOMAIN errors). When this happens, _matherr
 * returns the natural or base-10 logarithm of the absolute value
 * of the argument and suppresses the usual error message.
 */
int _matherr( struct _exception *except )
{
    /* Handle _DOMAIN errors for log or log10. */
    if( except->type == _DOMAIN )
    {
        if( strcmp( except->name, "log" ) == 0 )
        {
            except->retval = log( -(except->arg1) );
            printf( "Special: using absolute value: %s: _DOMAIN "
                    "error\n", except->name );
            return 1;
        }
        else if( strcmp( except->name, "log10" ) == 0 )
        {
            except->retval = log10( -(except->arg1) );
```

```
            printf( "Special: using absolute value: %s: _DOMAIN "
                     "error\n", except->name );
            return 1;
        }
    }
    printf( "Normal: " );
    return 0;    /* Else use the default actions */
}
```

Output

```
Special: using absolute value: log: _DOMAIN error
log( -2.0 ) = 6.931472e-001
Special: using absolute value: log10: _DOMAIN error
log10( -5.0 ) = 6.989700e-001
Normal: log( 0.0 ) = -1.#INF00e+000
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support

Long Double

_mbbtombc, _mbbtombc_l

Converts a single-byte multibyte character to a corresponding double-byte multibyte character.

```
unsigned int _mbbtombc(
   unsigned int c
);
unsigned int _mbbtombc_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Single-byte character to convert.

locale

Locale to use.

Return Value

If **_mbbtombc** successfully converts *c*, it returns a multibyte character; otherwise it returns *c*.

Remarks

The **_mbbtombc** function converts a given single-byte multibyte character to a corresponding double-byte multibyte character. Characters must be within the range 0x20 – 0x7E or 0xA1 – 0xDF to be converted.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The **_mbbtombc** version of this function uses the current locale for this locale-dependent behavior; the **_mbbtombc_l** version is identical except that it use the locale parameter passed in instead.

In earlier versions, **_mbbtombc** was called **hantozen**. For new code, use **_mbbtombc** instead.

Requirements

Routine	Required header	Compatibility
_mbbtombc	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbbtombc_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Data Conversion

_mbctombb, _mbctombb_l

_mbbtype, _mbbtype_l

Returns the byte type based on the previous byte.

```
int _mbbtype(
   unsigned char c,
   int type
);
int _mbbtype_l(
   unsigned char c,
   int type,
   _locale_t locale
);
```

Parameters

c

Character to test.

type

Type of byte to test for.

locale

Locale to use.

Return Value

_mbbtype returns the type of byte within a string. This decision is context-sensitive as specified by the value of *type*, which provides the control test condition. *type* is the type of the previous byte in the string. The manifest constants in the following table are defined in Mbctype.h.

Value of *type*	**_mbbtype tests for**	**Return value**	***c***
Any value except 1	Valid single byte or lead byte	**_MBC_SINGLE** (0)	Single byte (0x20 – 0x7E, 0xA1 – 0xDF)
Any value except 1	Valid single byte or lead byte	**_MBC_LEAD** (1)	Lead byte of multibyte character (0x81 – 0x9F, 0xE0 – 0xFC)
Any value except 1	Valid single-byte or lead byte	**_MBC_ILLEGAL** (–1)	Invalid character (any value except 0x20 – 0x7E, 0xA1 – 0xDF, 0x81 – 0x9F, 0xE0 – 0xFC
1	Valid trail byte	**_MBC_TRAIL** (2)	Trailing byte of multibyte character (0x40 – 0x7E, 0x80 – 0xFC)
1	Valid trail byte	**_MBC_ILLEGAL** (–1)	Invalid character (any value except 0x20 – 0x7E, 0xA1 – 0xDF, 0x81 – 0x9F, 0xE0 – 0xFC

Remarks

The **_mbbtype** function determines the type of a byte in a multibyte character. If the value of *type* is any value except 1, **_mbbtype** tests for a valid single-byte or lead byte of a multibyte character. If the value of *type* is 1, **_mbbtype** tests for a valid trail byte of a multibyte character.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The **_mbbtype** version of this function uses the current locale for this locale-dependent behavior; the **_mbbtype_l** version is identical except that it use the locale parameter passed in instead.

In earlier versions, **_mbbtype** was called **chkctype**. For new code, use **_mbbtype** instead.

Requirements

Routine	Required header	Optional header	Compatibility
_mbbtype	<mbstring.h>	<mbctype.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbbtype_l	<mbstring.h>	<mbctype.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

* For definitions of manifest constants used as return values.

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable, but see System::Globalization::CultureInfo.

See Also

Reference

Byte Classification

_mbccpy, _mbccpy_l

Copy a multibyte character from one string to another string. These functions are deprecated because more secure versions are available; see _mbccpy_s, _mbccpy_s_l.

```
void _mbccpy(
   unsigned char *dest,
   const unsigned char *src
);
void _mbccpy_l(
   unsigned char *dest,
   const unsigned char *src,
   _locale_t locale
);
```

Parameters

dest

Copy destination.

src

Multibyte character to copy.

locale

Locale to use.

Remarks

The **_mbccpy** function copies one multibyte character from *src* to *dest*.

This function validates its parameters. If **_mbccpy** is passed a null pointer for *dest* or *src*, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL**.

_mbccpy uses the current locale for any locale-dependent behavior. **_mbccpy_l** is identical to **_mbccpy** except that **_mbccpy_l** uses the locale passed in for any locale-dependent behavior.

Security Note Use a null-terminated string. The null-terminated string must not exceed the size of the destination buffer. For more information, see Avoiding Buffer Overruns. Buffer overrun problems are a frequent method of system attack, resulting in an unwarranted elevation of privilege.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tccpy	Maps to macro or inline function	**_mbccpy**	Maps to macro or inline function
_tccpy_l	n/a	**_mbccpy_l**	n/a

Requirements

Routine	Required header	Compatibility
_mbccpy	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbccpy_l	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Locale

Interpretation of Multibyte-Character Sequences

_mbclen, mblen, _mblen_l

_mbccpy_s, _mbccpy_s_l

Copies a single multibyte character from one string to another string. These are versions of _mbccpy, _mbccpy_l with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _mbccpy_s(
   unsigned char *dest,
   size_t buffSizeInBytes,
   int * pCopied,
   const unsigned char *src
);
errno_t _mbccpy_s_l(
   unsigned char *dest,
   size_t buffSizeInBytes,
   int * pCopied,
   const unsigned char *src,
   locale_t locale
);
template <size_t size>
errno_t _mbccpy_s(
   unsigned char (&dest)[size],
   int * pCopied,
   const unsigned char *src
); // C++ only
template <size_t size>
errno_t _mbccpy_s_l(
   unsigned char (&dest)[size],
   int * pCopied,
   const unsigned char *src,
   locale_t locale
); // C++ only
```

Parameters

[out] *dest*

Copy destination.

[in] *buffSizeInBytes*

Size of the destination buffer.

[out] *pCopied*

Filled with the number of bytes copied (1 or 2 if successful). Pass **NULL** if you don't care about the number.

[in] *src*

Multibyte character to copy.

[in] *locale*

Locale to use.

Return Value

Zero if successful; an error code on failure. If *src* or *dest* is **NULL**, or if more than *buffSizeinBytes* bytes would be copied to *dest*, then the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return **EINVAL** and **errno** is set to **EINVAL**.

Remarks

The **_mbccpy_s** function copies one multibyte character from *src* to *dest*. If *src* does not point to the lead byte of a multibyte character as determined by an implicit call to _ismbblead, then the single byte that *src* points to is copied. If *src* points to a lead byte but the following byte is 0 and thus invalid, then 0 is copied to *dest*, **errno** is set to **EILSEQ**, and the function returns **EILSEQ**.

_mbccpy_s does not append a null terminator; however, if *src* points to a null character, then that null is copied to *dest* (this is

just a regular single-byte copy).

The value in *pCopied* is filled with the number of bytes copied. Possible values are 1 and 2 if the operation is successful. If **NULL** is passed in, this parameter is ignored.

src	**copied to *dest***	***pCopied***	**Return value**
non-lead-byte	non-lead-byte	1	0
0	0	1	0
lead-byte followed by non-0	lead-byte followed by non-0	2	0
lead-byte followed by 0	0	1	**EILSEQ**

Note that the second row is just a special case of the first. Also note that the table assumes *buffSizeInBytes* >= *pCopied*.

_mbccpy_s uses the current locale for any locale-dependent behavior. **_mbccpy_s_l** is identical to **_mbccpy_s** except that **_mbccpy_s_l** uses the locale passed in for any locale-dependent behavior.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically, eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

Tchar.h routine	**_UNICODE and _MBCS not defined**	**_MBCS defined**	**_UNICODE defined**
_tccpy_s	Maps to macro or inline function.	**_mbccpy_s**	Maps to macro or inline function.

Requirements

Routine	**Required header**	**Compatibility**
_mbccpy_s	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbccpy_s_l	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

See Also

Reference

Locale

Interpretation of Multibyte-Character Sequences

_mbclen, mblen, _mblen_l

_mbcjistojms, _mbcjistojms_l, _mbcjmstojis, _mbcjmstojis_l

Converts between Japan Industry Standard (JIS) and Japan Microsoft (JMS) characters.

```
unsigned int _mbcjistojms(
   unsigned int c
);
unsigned int _mbcjistojms_l(
   unsigned int c,
   _locale_t locale
);
unsigned int _mbcjmstojis(
   unsigned int c
);
unsigned int _mbcjmstojis_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Character to convert.

local

Locale to use.

Return Value

On Japanese locale, these functions return a converted character or return 0 if no conversion is possible. On a non-Japanese locale, these functions return the character passed in.

Remarks

The **_mbcjistojms** function converts a Japan Industry Standard (JIS) character to a Microsoft Kanji (Shift JIS) character. The character is converted only if the lead and trail bytes are in the range 0x21 – 0x7E. If the lead or trial byte is outside this range, **errno** is set to **EILSEQ**. For more information about this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

The **_mbcjmstojis** function converts a Shift JIS character to a JIS character. The character is converted only if the lead byte is in the range 0x81 – 0x9F or 0xE0 – 0xFC and the trail byte is in the range 0x40 – 0x7E or 0x80 – 0xFC. Note that some code points in that range do not have a character assigned and so cannot be converted.

The value *c* should be a 16-bit value whose upper 8 bits represent the lead byte of the character to convert and whose lower 8 bits represent the trail byte.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

In earlier versions, **_mbcjistojms** and **_mbcjmstojis** were called **jistojms** and **jmstojis**, respectively. **_mbcjistojms**, **_mbcjistojms_l**, **_mbcjmstojis** and **_mbcjmstojis_l** should be used instead.

Requirements

Routine	Required header	Compatibility
_mbcjistojms	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbcjistojms_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_mbcjmstojis	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbcjmstojis_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Data Conversion

_ismbb Routines

_mbclen, mblen, _mblen_l

Gets the length and determines the validity of a multibyte character.

```
size_t _mbclen(
   const unsigned char *c
);
int mblen(
   const char *mbstr,
   size_t count
);
int _mblen_l(
   const char *mbstr,
   size_t count,
   _locale_t locale
);
```

Parameters

c

Multibyte character.

mbstr

Address of a multibyte-character byte sequence.

count

Number of bytes to check.

locale

Locale to use.

Return Value

_mbclen returns 1 or 2, according to whether the multibyte character *c* is 1 or 2 bytes long. There is no error return for **_mbclen**. If *mbstr* is not **NULL**, **mblen** returns the length, in bytes, of the multibyte character. If *mbstr* is **NULL** or it points to the wide-character null character, **mblen** returns 0. If the object that *mbstr* points to does not form a valid multibyte character within the first *count* characters, **mblen** returns –1.

Remarks

The **_mbclen** function returns the length, in bytes, of the multibyte character *c*. If *c* does not point to the lead byte of a multibyte character as determined by an implicit call to **_ismbblead**, the result of **_mbclen** is unpredictable.

mblen returns the length in bytes of *mbstr* if it is a valid multibyte character and determines multibyte-character validity associated with the code page. **mblen** examines *count* or fewer bytes contained in *mbstr*, but not more than **MB_CUR_MAX** bytes.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tclen	Maps to macro or inline function	**_mbclen**	Maps to macro or inline function

Requirements

Routine	Required header	Compatibility
_mbclen	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

mblen	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mblen_l	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_mblen.c
/* illustrates the behavior of the mblen function
 */

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    int      i;
    char    *pmbc = (char *)malloc( sizeof( char ) );
    wchar_t  wc   = L'a';

    printf( "Convert wide character to multibyte character:\n" );
    wctomb_s( &i, pmbc, sizeof(char), wc );
    printf( "  Characters converted: %u\n", i );
    printf( "  Multibyte character: %x\n\n", *pmbc );

    i = mblen( pmbc, MB_CUR_MAX );
    printf( "Length in bytes of multibyte character %x: %u\n", *pmbc, i );

    pmbc = NULL;
    i = mblen( pmbc, MB_CUR_MAX );
    printf( "Length in bytes of NULL multibyte character %x: %u\n", pmbc, i );
}
```

Output

```
Convert wide character to multibyte character:
  Characters converted: 1
  Multibyte character: 61

Length in bytes of multibyte character 61: 1
Length in bytes of NULL multibyte character 0: 0
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Character Classification
Locale
Interpretation of Multibyte-Character Sequences
_mbccpy, _mbccpy_l
strlen, strlen_l, wcslen, wcslen_l, _mbslen, _mbslen_l, _mbstrlen, _mbstrlen_l

_mbctohira, _mbctohira_l, _mbctokata, _mbctokata_l

Converts between hiragana and katakana characters.

```
unsigned int _mbctohira(
   unsigned int c
);
unsigned int _mbctohira_l(
   unsigned int c,
   _locale_t locale
);
unsigned int _mbctokata(
   unsigned int c
);
unsigned int _mbctokata_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Multibyte character to convert.

locale

Locale to use.

Return Value

Each of these functions returns the converted character *c*, if possible. Otherwise it returns the character *c* unchanged.

Remarks

The **_mbctohira** and **_mbctokata** functions test a character *c* and, if possible, apply one of the following conversions.

Routines	Converts
_mbctohira,_mbctohira_l	Multibyte katakana to multibyte hiragana.
_mbctokata,_mbctokata_l	Multibyte hiragana to multibyte katakana.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

In previous versions, **_mbctohira** was called **jtohira** and **_mbctokata** was called **jtokata**. For new code, use the new names instead.

Requirements

Routine	Required header	Compatibility
_mbctohira	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbctohira_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbctokata	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_mbctokata_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Data Conversion
_mbcjistojms, _mbcjistojms_l, _mbcjmstojis, _mbcjmstojis_l
_mbctolower, _mbctolower_l, _mbctoupper, _mbctoupper_l
_mbctombb, _mbctombb_l

_mbctolower, _mbctolower_l, _mbctoupper, _mbctoupper_l

Tests and converts the case of a multibyte character.

```
unsigned int _mbctolower(
   unsigned int c
);
unsigned int _mbctolower_l(
   unsigned int c,
   _locale_t locale
);
unsigned int _mbctoupper(
   unsigned int c
);
unsigned int _mbctoupper_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Multibyte character to convert.

locale

Locale to use.

Return Value

Each of these functions returns the converted character *c*, if possible. Otherwise it returns the character *c* unchanged.

Remarks

The functions test a character *c* and, if possible, apply one of the following conversions.

Routines	Converts
_mbctolower,_mbctolower_l	Uppercase character to lowercase character.
_mbctoupper,_mbctoupper_l	Lowercase character to uppercase character.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The version of this function without the **_l** suffix uses the current locale for this locale-dependent behavior; the version with the **_l** suffix is identical except that it uses the locale parameter passed in instead.

In previous versions, **_mbctolower** was called **jtolower**, and **_mbctoupper** was called **jtoupper**. For new code, use the new names instead.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_totlower	**tolower**	**_mbctolower**	**towlower**
_totlower_l	**_tolower_l**	**_mbctolower_l**	**_towlower_t**
_totupper	**toupper**	**_mbctoupper**	**towupper**
_totupper_l	**toupper_l**	**_mbctoupper_l**	**_towupper_l**

Requirements

Routines	Required header	Compatibility
_mbctolower, _mbctolower_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbctoupper, _mbctoupper_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

See Also

Reference

Data Conversion
_mbbtombc, _mbbtombc_l
_mbcjistojms, _mbcjistojms_l, _mbcjmstojis, _mbcjmstojis_l
_mbctohira, _mbctohira_l, _mbctokata, _mbctokata_l
_mbctombb, _mbctombb_l

_mbctombb, _mbctombb_l

Converts a double-byte multibyte character to a corresponding single-byte multibyte character.

```
unsigned int _mbctombb(
   unsigned int c
);
unsigned int _mbctombb_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c

Multibyte character to convert.

locale

Locale to use.

Return Value

If successful, **_mbctombb** and **_mbctombb_l** returns the single-byte character that corresponds to *c*; otherwise it returns *c*.

Remarks

The **_mbctombb** and **_mbctombb_l** functions convert a given multibyte character to a corresponding single-byte multibyte character. Characters must correspond to single-byte characters within the range 0x20 – 0x7E or 0xA1 – 0xDF to be converted.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The version of this function without the **_l** suffix uses the current locale for this locale-dependent behavior; the version with the **_l** suffix is identical except that it use the locale parameter passed in instead.

In previous versions, **_mbctombb** was called **zentohan**. Use **_mbctombb** instead.

Requirements

Routine	Required header	Compatibility
_mbctombb	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbctombb_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Data Conversion
_mbbtombc, _mbbtombc_l
_mbcjistojms, _mbcjistojms_l, _mbcjmstojis, _mbcjmstojis_l
_mbctohira, _mbctohira_l, _mbctokata, _mbctokata_l
_mbctolower, _mbctolower_l, _mbctoupper, _mbctoupper_l

mbrlen

Determine the number of bytes in a string, with the capability of restarting in the middle of a multibyte character if need be, while using the current locale.

```
size_t mbrlen(
   const char *str,
   size_t maxSize,
   mbstate_t mbstate
);
```

Parameters

str

Null-terminated string.

maxSize

The maximum size of the string, excluding the terminating null character.

mbstate

The shift state of conversion.

Return Value

If the string is less than *maxSize* characters in length, each of these functions returns the number of characters in *str*, excluding the terminal **NULL**. If the string is greater than *maxSize* characters in length, then *maxSize* is returned.

0

If the next count or fewer bytes complete the multibyte character that represents the **NULL** wide character.

> 0

If the next count or fewer bytes complete a valid multibyte character, the value returned is the number of bytes that complete the multibyte character.

-1

If the next count bytes contribute to an incomplete multibyte and all count bytes have been processed.

-2

If an encoding error occurs, in which case the next count or fewer bytes do not contribute to the complete and valid multibyte character, the value errno value will be EILSEQ and the conversion state ambiguous.

Remarks

The **mbrlen** function determines the number of bytes constituting the multibyte character sequence of *str*, with the capability of restarting in the middle of a multi-byte character if need be, examining at most *maxSize* bytes. The **mbstate_t** argument *mbstate* is used to keep track of the shift state. If it is NULL, **mbrlen** uses an internal, static **mbstate_t** object. It is equivalent to:

```
mbrtowc(NULL, str, maxSize, mbstate)
```

Except when the case of *mbstate* is **NULL**, **mbrlen** relies upon its own static, internal **mbstate_t** object to keep track of the shift state.

The **mbrlen** function differs from _mbclen, mblen, _mblen_l by its restartability. The conversion state is stored in *mbstate* for subsequent calls to the same or other restartable functions. Results are undefined when mixing the use of restartable and nonrestartable functions. For example, an application would use **wcsrlen** rather than **wcslen**, if a subsequent call to **wcsrtombs** where used instead of **wcstombs.**

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined

n/a	**n/a**	**mbrlen**	**n/a**

Requirements

Routine	Required header	Compatibility
mbrlen	<wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_mbrlen.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <wchar.h>

size_t Example(const char * pStr)
{
    size_t      charLen = 0;
    size_t      charCount = 0;
    mbstate_t   mbState;

    memset(&mbState, 0, sizeof(mbState));

    while ((charLen = mbrlen(pStr, MB_CUR_MAX, &mbState)) != 0 &&
            charLen != (size_t)-1 && charLen != (size_t)-2)
    {
        pStr += charLen;
        charCount++;
    }

    return (charCount);
}

int main( void )
{
    size_t      charCount = 0;
    const char  *pSample = "Every good boy does fine.";

    charCount = Example(pSample);
    printf("%s\nLength: %d\n", pSample, charCount);
}
```

Output

```
Every good boy does fine.
Length: 25
```

.NET Framework Equivalent

System::String::Length

See Also

Reference

String Manipulation (CRT)

Locale

mbrtowc

Convert a multi-byte character into the equivalent wide character.

```
size_t mbrtowc(
   wchar_t *wchar,
   const char *mbchar,
   size_t count,
   mbstate_t mbstate
);
```

Parameters

wchar

Address of a wide character to receive the converted wide character string (type **wchar_t**). This value can be **NULL** if no return wide character is required.

mbchar

Address of a sequence of bytes (a multibyte character).

count

Number of bytes to check.

mbstate

Conversion state. If this value is NULL, an internal conversion state category is used.

Return Value

0

If the next *count* or fewer bytes complete the multibyte character that represents the **NULL** wide character.

> 0

If the next *count* or fewer bytes complete a valid multibyte character, the value returned is the number of bytes that complete the multibyte character.

-1

If an encoding error occurs, in which case the next *count* or fewer bytes do not contribute to the complete and valid multibyte character, the errno value will be EILSEQ and the conversion state ambiguous.

-2

If the next *count* bytes contribute to an incomplete multibyte and all count bytes have been processed.

Remarks

If **wchar** is a NULL value, the function is equivalent to the call:

```
mbrtowc(NULL, NULL, 1, mbstate)
```

In this case, the value of the arguments **wchar** and **count** are ignored.

If **wchar** is not NULL, the function examines **count** bytes from **mbchar** to determine the required number of bytes needed to complete the next multibyte character. If the next character is valid, the corresponding multibyte character is stored in **wchar** if it is not NULL. If the character is the corresponding wide NULL character, the resulting state is the initial conversion state.

The **mbrtowc** function differs from mbtowc, _mbtowc_l by its restartability. The conversion state is stored in *mbstate* for subsequent calls to the same or other restartable functions. Results are undefined when mixing the use of restartable and nonrestartable functions. For example, an application would use **wcsrlen** rather than **wcslen**, if a subsequent call to **wcsrtombs** where used instead of **wcstombs**.

Example

Converts a multibyte character to its wide character equivalent.

```
// crt_mbrtowc.cpp

#include <stdio.h>
#include <mbctype.h>
#include <string.h>
#include <locale.h>
#include <wchar.h>

#define BUF_SIZE 100

int Sample(char* szIn, wchar_t* wcOut, int nMax)
{
    mbstate_t   state = {0}; // Initial state
    size_t      nConvResult,
                nmbLen = 0,
                nwcLen = 0;
    wchar_t*    wcCur = wcOut;
    wchar_t*    wcEnd = wcCur + nMax;
    const char* mbCur = szIn;
    const char* mbEnd = mbCur + strlen(mbCur) + 1;
    char*       szLocal;

    // Sets all locale to French_Canada.1252
    szLocal = setlocale(LC_ALL, "French_Canada.1252");
    if (!szLocal)
    {
        printf("The fuction setlocale(LC_ALL, \"French_Canada.1252\") failed!\n");
        return 1;
    }

    printf("Locale set to: \"%s\"\n", szLocal);

    // Sets the code page associated current locale's code page
    // from a previous call to setlocale.
    if (_setmbcp(_MB_CP_SBCS) == -1)
    {
        printf("The fuction _setmbcp(_MB_CP_SBCS) failed!");
        return 1;
    }

    while ((mbCur < mbEnd) && (wcCur < wcEnd))
    {
        //
        nConvResult = mbrtowc(wcCur, mbCur, 1, &state);
        switch (nConvResult)
        {
            case 0:
            {  // done
                printf("Conversion succeeded!\nMultibyte String: ");
                printf(szIn);
                printf("\nWC String: ");
                wprintf(wcOut);
                printf("\n");
                mbCur = mbEnd;
                break;
            }

            case -1:
            {  // encoding error
                printf("The call to mbrtowc has detected an encoding error.\n");
                mbCur = mbEnd;
                break;
            }

            case -2:
            {  // incomplete character
                if   (!mbsinit(&state))
                {
```

```
                    printf("Currently in middle of mb conversion, state = %x\n", state);
                    // state will contain data regarding lead byte of mb character
                }

                ++nmbLen;
                ++mbCur;
                break;
            }

            default:
            {
                if   (nConvResult > 2) // The multibyte should never be larger than 2
                {
                    printf("Error: The size of the converted multibyte is %d.\n", nConvResu
lt);
                }

                ++nmbLen;
                ++nwcLen;
                ++wcCur;
                ++mbCur;
            break;
            }
        }
    }

   return 0;
}

int main(int argc, char* argv[])
{
    char    mbBuf[BUF_SIZE] = "AaBbCc\x9A\x8B\xE0\xEF\xF0xXyYzZ";
    wchar_t wcBuf[BUF_SIZE] = {L''};

    return Sample(mbBuf, wcBuf, BUF_SIZE);
}
```

Sample Output

```
Locale set to: "French_Canada.1252"
Conversion succeeded!
Multibyte String: AaBbCcÜïαn≡xXyYzZ
WC String: AaBbCcÜïαn≡xXyYzZ
```

Requirements

Routine	Required header	Compatibility
mbrtowc	<wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Data Conversion

Locale

Interpretation of Multibyte-Character Sequences

_mbsbtype, _mbsbtype_l

Returns the type of byte within a string.

```
int _mbsbtype(
   const unsigned char *mbstr,
   size_t count
);
int _mbsbtype_l(
   const unsigned char *mbstr,
   size_t count,
   _locale_t locale
);
```

Parameters

mbstr

Address of a sequence of multibyte characters.

count

Byte offset from head of string.

locale

Locale to use.

Return Value

_mbsbtype and **_mbsbtype_l** returns an integer value indicating the result of the test on the specified byte. The manifest constants in the following table are defined in Mbctype.h.

Return value	Byte type
_MBC_SINGLE (0)	Single-byte character. For example, in code page 932, **_mbsbtype** returns 0 if the specified byte is within the range 0x20 – 0x7E or 0xA1 – 0xDF.
_MBC_LEAD (1)	Lead byte of multibyte character. For example, in code page 932, **_mbsbtype** returns 1 if the specified byte is within the range 0x81 – 0x9F or 0xE0 – 0xFC.
_MBC_TRAIL (2)	Trailing byte of multibyte character. For example, in code page 932, **_mbsbtype** returns 2 if the specified byte is within the range 0x40 – 0x7E or 0x80 – 0xFC.
_MBC_ILLEGAL (–1)	**NULL** string, invalid character, or **NULL** byte found before the byte at offset *count* in *mbstr*.

Remarks

The **_mbsbtype** function determines the type of a byte in a multibyte character string. The function examines only the byte at offset *count* in *mbstr*, ignoring invalid characters before the specified byte.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The version of this function without the **_l** suffix uses the current locale for this locale-dependent behavior; the version with the **_l** suffix is identical except that it use the locale parameter passed in instead.

If the input string is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **_MBC_ILLEGAL**.

Requirements

Routine	Required header	Optional header	Compatibility

_mbsbtype	<mbstring.h>	<mbctype.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsbtype_l	<mbstring.h>	<mbctype.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

* For manifest constants used as return values.

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable, but see System::Globalization::CultureInfo.

See Also
Reference
Byte Classification

_mbsdec, _mbsdec_l, _strdec, _wcsdec

Moves a string pointer back one character.

```
unsigned char *_mbsdec(
   const unsigned char *start,
   const unsigned char *current
);
unsigned char *_mbsdec_l(
   const unsigned char *start,
   const unsigned char *current,
   _locale_t locale
);
```

Parameters

start

Pointer to first byte of any multibyte character in the source string; *start* must precede *current* in the source string.

current

Pointer to first byte of any multibyte character in the source string; *current* must follow *start* in the source string.

locale

Locale to use.

Return Value

_mbsdec, **_mbsdec_l**, **_strdec**, and **_wcsdec** each return a pointer to the character that immediately proceeds *current*; **_mbsdec** returns **NULL** if the value of *start* is greater than or equal to that of *current*. **_tcsdec** maps to one of these functions and its return value depends on the mapping.

Remarks

The **_mbsdec** and **_mbsdec_l** function returns a pointer to the first byte of the multibyte character that immediately precedes *current* in the string that contains *start*.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. **_mbsdec** recognizes multibyte-character sequences according to the local currently in use, while **_mbsdec_l** is identical except that it use the locale parameter passed in instead.

If *start* or *current* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function returns **EINVAL** and sets **errno** to **EINVAL**.

Security Note This API incurs a potential threat brought about by a buffer overrun problem. Buffer overrun problems are a frequent method of system attack, resulting in an unwarranted elevation of privilege. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsdec	**_strdec**	**_mbsdec**	**_wcsdec**

_strdec and **_wcsdec** are single-byte character and wide-character versions of **_mbsdec** and **_mbsdec_l**. **_strdec** and **_wcsdec** are provided only for this mapping and should not be used otherwise.

For more information, see Using Generic-Text Mappings and Generic-Text Mappings.

Requirements

Routine	Required header	Optional header	Compatibility

_mbsdec	<mbstring.h>	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsdec_l	<mbstring.h>	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_strdec	<tchar.h>		Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsdec	<tchar.h>		Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

The following code snippet shows a use of **_tcsdec**.

```
    const TCHAR *str = _T("some text");
    const TCHAR *str2;
    TCHAR *answer;
    str2 = str;
    str2++;
    answer = _tcsdec( str, str2 );
```

The following code snippet shows a use of **_mbsdec**.

```
    char *str = "some text";
    char *str2;
    unsigned char *answer;
    str2 = str;
    str2++;
    answer = _mbsdec( reinterpret_cast<unsigned char *>( str ), reinterpret_cast<unsigned ch
ar *>( str2 ));
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

String Manipulation (CRT)
_mbsinc, _mbsinc_l, _strinc, _wcsinc
_mbsnextc, _mbsnextc_l, _strnextc, _wcsnextc
_mbsninc, _mbsninc_l, _strninc, _wcsninc

_mbsinc, _mbsinc_l, _strinc, _wcsinc

Advances a string pointer by one character.

```
unsigned char *_mbsinc(
   const unsigned char *current
);
unsigned char *_mbsinc(
   const unsigned char *current,
   _locale_t locale
);
char *_strinc(
   const char *current,
   _locale_t locale
);
wchar_t *_wcsinc(
   const wchar_t *current,
   _locale_t locale
);
```

Parameters

current

Character pointer.

locale

Locale to use.

Return Value

Each of these routines returns a pointer to the character that immediately follows *current*.

Remarks

The **_mbsinc** function returns a pointer to the first byte of the multibyte character that immediately follows *current*. **_mbsinc** recognizes multibyte-character sequences according to the multibyte code page currently in use, while **_mbsinc_l** is identical except that it uses the locale parameter passed in instead.

The generic-text function **_tcsinc**, defined in Tchar.h, maps to **_mbsinc** if **_MBCS** has been defined or to **_wcsinc** if **_UNICODE** has been defined. Otherwise **_tcsinc** maps to **_strinc**. **_strinc** and **_wcsinc** are single-byte character and wide-character versions of **_mbsinc**. **_strinc** and **_wcsinc** are provided only for this mapping and should not be used otherwise. For more information, see Using Generic-Text Mappings and Generic-Text Mappings.

If *current* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function returns **EINVAL** and sets **errno** to **EINVAL**.

Security Note This API incurs a potential threat brought about by a buffer overrun problem. Buffer overrun problems are a frequent method of system attack, resulting in an unwarranted elevation of privilege. For more information, see Avoiding Buffer Overruns.

Requirements

Routine	Required header	Compatibility
_mbsinc	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsinc_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_strinc	<tchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_wcsinc	<tchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

String Manipulation (CRT)
_mbsdec, _mbsdec_l, _strdec, _wcsdec
_mbsnextc, _mbsnextc_l, _strnextc, _wcsnextc
_mbsninc, _mbsninc_l, _strninc, _wcsninc

mbsinit

Tracks the state of a multibyte character conversion.

```
int mbsinit(
   const mbstate_t* ps
);
```

Parameters

ps

A pointer to an mbstate_t variable.

Return Value

Nonzero if *ps* is NULL or if not in the middle of a conversion.

Remarks

When using one of the ANSI functions that takes an **mbstate_t** pointer, passing the address of your **mbstate_t** will return information about whether the last byte in the buffer was converted.

The appropriate code page needs to be installed to support your multibyte characters.

Example

```
// crt_mbsinit.cpp
#include <stdio.h>
#include <mbctype.h>
#include <string.h>
#include <locale.h>
#include <cwchar>
#include <xlocinfo.h>

#define   BUF_SIZE   0x40

wchar_t      g_wcBuf[BUF_SIZE] = L"This a wc buffer that will be over written...";
char      g_mbBuf[BUF_SIZE] = "AaBbCc\x9A\x8B\xE0\xEF\xF0xXyYzZ";
int      g_nInit = strlen(g_mbBuf);

int MbsinitSample(char* szIn, wchar_t* wcOut, int nMax)
{
   mbstate_t   state = {0};
   size_t      nConvResult, nmbLen = 0, nwcLen = 0;
   wchar_t*   wcCur = wcOut;
   wchar_t*   wcEnd = wcCur + nMax;
   const char*   mbCur = szIn;
   const char*   mbEnd = mbCur + strlen(mbCur) + 1;
   char*      szLocal = setlocale(LC_ALL, "japanese");

   printf("Locale set to: \"%s\"\n", szLocal);

   if   (_setmbcp(_MB_CP_LOCALE) != -1)
   {
      while   ((mbCur < mbEnd) && (wcCur < wcEnd))
      {
         nConvResult = mbrtowc(wcCur, mbCur, 1, &state);

         switch   (nConvResult)
         {
            case 0:
            {   // done
               printf("Conversion succeeded!\nMB String: ");
               printf(szIn);
               printf("\nWC String: ");
               wprintf(wcOut);
```

```
                printf("\n");
                mbCur = mbEnd;
                break;
            }

            case -1:
            {   // encoding error
                printf("ERROR: The call to mbrtowc has detected an encoding error.\n");
                mbCur = mbEnd;
                break;
            }

            case -2:
            {   // incomplete character
                if   (!mbsinit(&state))
                {
                    printf("Currently in middle of mb conversion, state = %x\n", state);
                    // state will contain data regarding lead byte of mb character
                }

                ++nmbLen;
                ++mbCur;
                break;
            }

            default:
            {
                if   (nConvResult > 2)   // Microsoft mb should never be larger than 2
                    printf("ERROR: nConvResult = %d\n", nConvResult);

                ++nmbLen;
                ++nwcLen;
                ++wcCur;
                ++mbCur;
                break;
            }
        }
    }
}
else
    printf("ERROR: _setmbcp(932) failed!");

return 0;
}

int main(int argc, char* argv[])
{
    return MbsinitSample(g_mbBuf, g_wcBuf, BUF_SIZE);
}
```

Sample Output

```
Locale set to: "Japanese_Japan.932"
Currently in middle of mb conversion, state = 9a
Currently in middle of mb conversion, state = e0
Currently in middle of mb conversion, state = f0
Conversion succeeded!
MB String: AaBbCcxXyYzZ
WC String: AaBbCcxXyYzZ
```

See Also

Reference

Byte Classification

_mbsnbcat, _mbsnbcat_l

Append, at most, the first *n* bytes of one multibyte-character string to another. These functions are deprecated because more secure versions are available; see _mbsnbcat_s, _mbsnbcat_s_l.

```
unsigned char *_mbsnbcat(
   unsigned char *dest,
   const unsigned char *src,
   size_t count
);
unsigned char *_mbsnbcat_l(
   unsigned char *dest,
   const unsigned char *src,
   size_t count,
   _locale_t locale
);
template <size_t size>
unsigned char *_mbsnbcat(
   unsigned char (&dest)[size],
   const unsigned char *src,
   size_t count
); // C++ only
template <size_t size>
unsigned char *_mbsnbcat_l(
   unsigned char (&dest)[size],
   const unsigned char *src,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

dest

Null-terminated multibyte-character destination string.

src

Null-terminated multibyte-character source string.

count

Number of bytes from *src* to append to *dest*.

locale

Locale to use.

Return Value

_mbsnbcat returns a pointer to the destination string. No return value is reserved to indicate an error.

Remarks

The **_mbsnbcat** function appends, at most, the first *count* bytes of *src* to *dest*. If the byte immediately preceding the null character in *dest* is a lead byte, the initial byte of *src* overwrites this lead byte. Otherwise, the initial byte of *src* overwrites the terminating null character of *dest*. If a null byte appears in *src* before *count* bytes are appended, **_mbsnbcat** appends all bytes from *src*, up to the null character. If *count* is greater than the length of *src*, the length of *src* is used in place of *count*. The resulting string is terminated with a null character. If copying takes place between strings that overlap, the behavior is undefined.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The **_mbsnbcat** version of the function uses the current locale for this locale-dependent behavior; the **_mbsnbcat_l** version is identical except that they use the locale parameter passed in instead.

Security Note Use a null-terminated string. The null-terminated string must not exceed the size of the destination buffer. For more information, see Avoiding Buffer Overruns.

If *dest* or *src* is **NULL**, the function will generate an invalid parameter error, as described in Parameter Validation. If the error is handled, the function returns **EINVAL** and sets **errno** to **EINVAL**.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsncat	strncat	**_mbsnbcat**	wcsncat
_tcsncat_l	**_strncat_l**	**_mbsnbcat_l**	**_wcsncat_l**

Requirements

Routine	Required header	Compatibility
_mbsnbcat	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsnbcat_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

String Manipulation (CRT)
_mbsnbcmp, _mbsnbcmp_l
_mbsnbcnt, _mbsnbcnt_l, _mbsnccnt, _mbsnccnt_l, _strncnt, _wcsncnt
_mbsnbcpy, _mbsnbcpy_l
_mbsnbicmp, _mbsnbicmp_l
_mbsnbset, _mbsnbset_l
strncat, _strncat_l, wcsncat, wcsncat_l, _mbsncat _mbsncat_l
_mbsnbcat_s, _mbsnbcat_s_l

_mbsnbcat_s, _mbsnbcat_s_l

Appends, at most, the first *n* bytes of one multibyte-character string to another. These are versions of _mbsnbcat, _mbsnbcat_l with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _mbsnbcat_s(
   unsigned char *dest,
   size_t sizeInBytes,
   const unsigned char *src,
   size_t count
);
errno_t _mbsnbcat_s_l(
   unsigned char *dest,
   size_t sizeInBytes,
   const unsigned char *src,
   size_t count,
   _locale_t locale
);
template <size_t size>
errno_t _mbsnbcat_s(
   unsigned char (&dest)[size],
   const unsigned char *src,
   size_t count
); // C++ only
template <size_t size>
errno_t _mbsnbcat_s_l(
   unsigned char (&dest)[size],
   const unsigned char *src,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

dest

Null-terminated multibyte-character destination string.

sizeInBytes

Size of the *dest* buffer in bytes.

src

Null-terminated multibyte-character source string.

Count

Number of bytes from *src* to append to *dest*.

locale

Locale to use.

Return Value

Zero if successful; an error code otherwise.

Error Conditions

Dest	***sizeInBytes***	***src***	**Return value**
NULL	any	any	**EINVAL**
Any	<= 0	any	**EINVAL**
Any	any	**NULL**	**EINVAL**

If any of the above error conditions occurs, the function generates an invalid parameter error, as described in Parameter Validation. If the error is handled, the function returns **EINVAL** and sets **errno** to **EINVAL**.

Remarks

The **_mbsnbcat_s** function appends, at most, the first *count* bytes of *src* to *dest*. If the byte immediately preceding the null character in *dest* is a lead byte, the initial byte of *src* overwrites this lead byte. Otherwise, the initial byte of *src* overwrites the terminating null character of *dest*. If a null byte appears in *src* before *count* bytes are appended, **_mbsnbcat_s** appends all bytes from *src*, up to the null character. If *count* is greater than the length of *src*, the length of *src* is used in place of *count*. The resulting string is terminated with a null character. If copying takes place between strings that overlap, the behavior is undefined.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsncat	strncat	**_mbsnbcat_s**	wcsncat
_tcsncat_s_l	**_strncat_s_l**	**_mbsnbcat_s_l**	**_wcsncat_s_l**

Requirements

Routine	Required header	Compatibility
_mbsnbcat_s	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsnbcat_s_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

String Manipulation (CRT)
_mbsnbcmp, _mbsnbcmp_l
_mbsnbcnt, _mbsnbcnt_l, _mbsnccnt, _mbsnccnt_l, _strncnt, _wcsncnt
_mbsnbcpy, _mbsnbcpy_l
_mbsnbcpy_s, _mbsnbcpy_s_l
_mbsnbset, _mbsnbset_l
strncat, _strncat_l, wcsncat, wcsncat_l, _mbsncat _mbsncat_l
strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l

_mbsnbcmp, _mbsnbcmp_l

Compares the first *n* bytes of two multibyte-character strings.

```
int _mbsnbcmp(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count
);
int _mbsnbcmp_l(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count,
   _locale_t locale
);
```

Parameters

string1, *string2*

Strings to compare.

count

Number of bytes to compare.

locale

Locale to use.

Return Value

The return value indicates the relation of the substrings of *string1* and *string*.

Return value	Description
< 0	*string1* substring less than *string2* substring.
0	*string1* substring identical to *string2* substring.
> 0	*string1* substring greater than *string2* substring.

On an error, **_mbsnbcmp** returns **_NLSCMPERROR**, which is defined in String.h and Mbstring.h.

Remarks

The **_mbsnbcmp** function lexicographically compares, at most, the first *count* bytes in *string1* and *string2* and returns a value indicating the relationship between the substrings. **_mbsnbcmp** is a case-sensitive version of **_mbsnbicmp**. Unlike **strcoll**, **_mbsnbcmp** is not affected by locale. **_mbsnbcmp** recognizes multibyte-character sequences according to the current multibyte code page.

_mbsnbcmp is similar to **_mbsncmp**, except that **_mbsncmp** compares strings by characters rather than by bytes.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The version of this function without the **_l** suffix uses the current locale for this locale-dependent behavior; the version with the **_l** suffix is identical except that it uses the locale parameter passed in instead.

If either *string1* or *string2* is a null pointer, this function invokes the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, the function returns **_NLSCMPERROR** and **errno** is set to **EINVAL**.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsncmp	strncmp	**_mbsnbcmp**	wcsncmp

_tcsncmp_l	strncmp	**_mbsnbcml**	wcsncmp

Requirements

Routine	Required header	Compatibility
_mbsnbcmp	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsnbcmp_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_mbsnbcmp.c
#include <mbstring.h>
#include <stdio.h>

char string1[] = "The quick brown dog jumps over the lazy fox";
char string2[] = "The QUICK brown fox jumps over the lazy dog";

int main( void )
{
   char tmp[20];
   int result;
   printf( "Compare strings:\n          %s\n", string1 );
   printf( "          %s\n\n", string2 );
   printf( "Function: _mbsnbcmp (first 10 characters only)\n" );
   result = _mbsncmp( string1, string2 , 10 );
   if( result > 0 )
      _mbscpy_s( tmp, sizeof(tmp), "greater than" );
   else if( result < 0 )
      _mbscpy_s( tmp, sizeof(tmp), "less than" );
   else
      _mbscpy_s( tmp, sizeof(tmp), "equal to" );
   printf( "Result:   String 1 is %s string 2\n\n", tmp );
   printf( "Function: _mbsnicmp _mbsnicmp (first 10 characters only)\n" );
   result = _mbsnicmp( string1, string2, 10 );
   if( result > 0 )
      _mbscpy_s( tmp, sizeof(tmp), "greater than" );
   else if( result < 0 )
      _mbscpy_s( tmp, sizeof(tmp), "less than" );
   else
      _mbscpy_s( tmp, sizeof(tmp), "equal to" );
   printf( "Result:   String 1 is %s string 2\n\n", tmp );
}
```

Output

```
Compare strings:
          The quick brown dog jumps over the lazy fox
          The QUICK brown fox jumps over the lazy dog

Function: _mbsnbcmp (first 10 characters only)
Result:   String 1 is greater than string 2

Function: _mbsnicmp _mbsnicmp (first 10 characters only)
Result:   String 1 is equal to string 2
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

String Manipulation (CRT)
_mbsnbcat, _mbsnbcat_l
_mbsnbicmp, _mbsnbicmp_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
Locale
Interpretation of Multibyte-Character Sequences

_mbsnbcnt, _mbsnbcnt_l, _mbsnccnt, _mbsnccnt_l, _strncnt, _wcsncnt

Returns the number of characters or bytes within a supplied count.

```
size_t _mbsnbcnt(
   const unsigned char *str,
   size_t count
);
size_t _mbsnbcnt_l(
   const unsigned char *str,
   size_t count,
   _locale_t locale
);
size_t _mbsnccnt(
   const unsigned char *str,
   size_t count
);
size_t _mbsnccnt_l(
   const unsigned char *str,
   size_t count,
   _locale_t locale
);
size_t _strncnt(
   const char *str,
   size_t count
);
size_t _wcsncnt(
   const wchar_t *str,
   size_t count
);
```

Parameters

str

String to be examined.

count

Number of characters or bytes to be examined in *str*.

locale

Locale to use.

Return Value

_mbsnbcnt and **_mbsnbcnt_l** return the number of bytes found in the first *count* of multibyte characters of *str*. **_mbsnccnt** and **_mbsnccnt_l** return the number of characters found in the first *count* of bytes of *str*. If a NULL character is encountered before the examination of *str* has completed, they return the number of bytes or characters found before the NULL character. If *str* consists of fewer than *count* characters or bytes, they return the number of characters or bytes in the string. If *count* is less than zero, they return 0. In previous versions, these functions had a return value of type **int** rather than **size_t**.

_strncnt returns the number of characters in the first *count* bytes of the single-byte string *str*. **_wcsncnt** returns the number of characters in the first *count* wide characters of the wide-character string *str*.

Remarks

_mbsnbcnt and **_mbsnbcnt_l** count the number of bytes found in the first *count* of multibyte characters of *str*. **_mbsnbcnt** and **_mbsnbcnt_l** replace **mtob** and should be used in place of **mtob**.

_mbsnccnt and **_mbsnccnt_l** count the number of characters found in the first *count* of bytes of *str*. If **_mbsnccnt** and **_mbsnccnt_l** encounter a NULL in the second byte of a double-byte character, the first byte is also considered to be NULL and is not included in the returned count value. **_mbsnccnt** and **_mbsnccnt_l** replace **btom** and should be used in place of **btom**.

If *str* is a null pointer or is *count* is 0, these functions invoke the invalid parameter handler as described in Parameter Validation, **errno** is set to **EINVAL**, and the function returns 0.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Generic-Text Routine Mappings

Routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsnbcnt	**_strncnt**	**_mbsnbcnt**	**_wcsncnt**
_tcsnccnt	**_strncnt**	**_mbsnbcnt**	**n/a**
_wcsncnt	**n/a**	**n/a**	**_mbsnbcnt**
_wcsncnt	**n/a**	**n/a**	**_mbsnccnt**
n/a	**n/a**	**_mbsnbcnt_l**	**_mbsnccnt_l**

Requirements

Routine	Required header	Compatibility
_mbsnbcnt	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsnbcnt_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsnccnt	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsnccnt_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_strncnt	<tchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsncnt	<tchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_mbsnbcnt.c

#include  <mbstring.h>
#include  <stdio.h>

int main( void )
{
   unsigned char str[] = "This is a multibyte-character string.";
   unsigned int char_count, byte_count;
   char_count = _mbsnccnt( str, 10 );
   byte_count = _mbsnbcnt( str, 10 );
   if ( byte_count - char_count )
      printf( "The first 10 characters contain %d multibyte characters\n", char_count );
   else
      printf( "The first 10 characters are single-byte.\n");
}
```

Output

```
The first 10 characters are single-byte.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
_mbsnbcat, _mbsnbcat_l

_mbsnbcoll, _mbsnbcoll_l, _mbsnbicoll, _mbsnbicoll_l

Compares *n* bytes of two multibyte-character strings using multibyte code page information.

```
int _mbsnbcoll(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count
);
int _mbsnbcoll_l(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count,
   _locale_t locale
);
int _mbsnbicoll(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count
);
int _mbsnbicoll_l(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count,
   _locale_t locale
);
```

Parameters

string1, *string2*

Strings to compare.

count

Number of bytes to compare.

locale

Locale to use.

Return Value

The return value indicates the relation of the substrings of *string1* and *string2*.

Return value	Description
< 0	*string1* substring less than *string2* substring.
0	*string1* substring identical to *string2* substring.
> 0	*string1* substring greater than *string2* substring.

If *string1* or *string2* is **NULL** or *count* is greater than **INT_MAX**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **_NLSCMPERROR** and set **errno** to **EINVAL**. To use **_NLSCMPERROR**, include either String.h or Mbstring.h.

Remarks

Each of these functions collates, at most, the first *count* bytes in *string1* and *string2* and returns a value indicating the relationship between the resulting substrings of *string1* and *string2*. If the final byte in the substring of *string1* or *string2* is a lead byte, it is not included in the comparison; these functions compare only complete characters in the substrings. **_mbsnbicoll** is a case-insensitive version of **_mbsnbcoll**. Like **_mbsnbcmp** and **_mbsnbicmp**, **_mbsnbcoll** and **_mbsnbicoll** collate the two multibyte-character strings according to the lexicographic order specified by the multibyte code page currently in use.

For some code pages and corresponding character sets, the order of characters in the character set might differ from the lexicographic character order. In the "C" locale, this is not the case: the order of characters in the ASCII character set is the same as the lexicographic order of the characters. However, in certain European code pages, for example, the character 'a' (value 0x61) precedes the character 'ä' (value 0xE4) in the character set, but the character 'ä' precedes the character 'a' lexicographically. To perform a lexicographic comparison of strings by bytes in such an instance, use **_mbsnbcoll** rather than **_mbsnbcmp**; to check only for string equality, use **_mbsnbcmp**.

Because the **coll** functions collate strings lexicographically for comparison, whereas the **cmp** functions simply test for string equality, the **coll** functions are much slower than the corresponding **cmp** versions. Therefore, the **coll** functions should be used only when there is a difference between the character set order and the lexicographic character order in the current code page and this difference is of interest for the comparison.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsncoll	_strncoll	**_mbsnbcoll**	_wcsncoll
_tcsncoll_l	_strncoll, _wcsncoll, _mbsncoll, _strncoll_l, _wcsncoll_l, _mbsncoll_l	**_mbsnbcoll_l**	_wcsncoll_l
_tcsnicoll	_strnicoll	**_mbsnbicoll**	_wcsnicoll
_tcsnicoll_l	_strnicoll_l	**_mbsnbicoll_l**	_wcsnicoll_l

Requirements

Routine	Required header	Compatibility
_mbsnbcoll	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsnbcoll_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsnbicoll	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsnbicoll_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

See Also

Reference

String Manipulation (CRT)
_mbsnbcat, _mbsnbcat_l
_mbsnbcmp, _mbsnbcmp_l
_mbsnbicmp, _mbsnbicmp_l
strcoll Functions
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l

_mbsnbcpy, _mbsnbcpy_l

Copy *n* bytes of a string to a destination string. These functions are deprecated because more secure versions are available; see _mbsnbcpy_s, _mbsnbcpy_s_l.

```
unsigned char * _mbsnbcpy(
   unsigned char * strDest,
   const unsigned char * strSource,
   size_t count
);
unsigned char * _mbsnbcpy_l(
   unsigned char * strDest,
   const unsigned char * strSource,
   size_t count,
   _locale_t locale
);
template <size_t size>
unsigned char * _mbsnbcpy(
   unsigned char (&strDest)[size],
   const unsigned char * strSource,
   size_t count
); // C++ only
template <size_t size>
unsigned char * _mbsnbcpy_l(
   unsigned char (&strDest)[size],
   const unsigned char * strSource,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

strDest

Destination for character string to be copied.

strSource

Character string to be copied.

count

Number of bytes to be copied.

locale

Locale to use.

Return Value

_mbsnbcpy returns a pointer to the destination character string. No return value is reserved to indicate an error.

Remarks

The **_mbsnbcpy** function copies *count* bytes from *strSource* to *strDest*. If *count* is less than or equal to the size in bytes of *strSource*, a null character is not appended automatically to the copied string. If size in bytes of *strSource* is less than *count*, the destination buffer is padded with null characters. If *count* exceeds the size of *strDest* or the source and destination strings overlap, the behavior of **_mbstrncpy** is undefined.

If *strSource* or *strDest* is a null pointer, this function invokes the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, the function returns **NULL** and sets **errno** to **EINVAL**.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Security Note This API incurs a potential threat brought about by a buffer overrun problem. Buffer overrun problems are a frequent method of system attack, resulting in an unwarranted elevation of privilege. For more information, see

Avoiding Buffer Overruns.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsncpy	strncpy	**_mbsnbcpy**	wcsncpy
_tcsncpy_l	**_strncpy_l**	**_mbsnbcp_l**	**_wcsncpy_l**

Requirements

Routine	Required header	Compatibility
_mbsnbcpy	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsnbcpy_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

String Manipulation (CRT)
_mbsnbcat, _mbsnbcat_l
_mbsnbcmp, _mbsnbcmp_l
_mbsnbcnt, _mbsnbcnt_l, _mbsnccnt, _mbsnccnt_l, _strncnt, _wcsncnt
_mbsnbset, _mbsnbset_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l

_mbsnbcpy_s, _mbsnbcpy_s_l

Copies *n* bytes of a string to a destination string. These are versions of _mbsnbcpy, _mbsnbcpy_l with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _mbsnbcpy_s(
   unsigned char * strDest,
   size_t sizeInBytes,
   const unsigned char * strSource,
   size_t count
);
errno_t _mbsnbcpy_s_l(
   unsigned char * strDest,
   size_t sizeInBytes,
   const unsigned char * strSource,
   size_t count,
   _locale_t locale
);
template <size_t size>
errno_t _mbsnbcpy_s(
   unsigned char (&strDest)[size],
   const unsigned char * strSource,
   size_t count
); // C++ only
template <size_t size>
errno_t _mbsnbcpy_s_l(
   unsigned char (&strDest)[size],
   const unsigned char * strSource,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

strDest

Destination for character string to be copied.

sizeInBytes

Destination buffer size.

strSource

Character string to be copied.

count

Number of bytes to be copied.

locale

Locale to use.

Return Value

Zero if successful; **EINVAL** if a bad parameter was passed in.

Remarks

The **_mbsnbcpy_s** function copies *count* bytes from *strSource* to *strDest*. If *count* exceeds the size of *strDest*, either of the input strings is a null pointer, or *sizeInBytes* or *count* is 0, the function invokes the invalid parameter handler as described in Parameter Validation . If execution is allowed to continue, the function returns **EINVAL**. If the source and destination strings overlap, the behavior of **_mbstrncpy** is undefined.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Note
Unlike the non-secure version of this function, **_mbsnbcpy_s** does not do any null padding and always null terminates the string.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsncpy_s	**_strncpy_s**	**_mbsnbcpy_s**	**_wcsncpy_s**
_tcsncpy_s_l	**_strncpy_s_l**	**_mbsnbcpy_s_l**	**_wcsncpy_s_l**

Requirements

Routine	Required header	Compatibility
_mbsnbcpy_s	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsnbcpy_s_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

String Manipulation (CRT)
_mbsnbcat, _mbsnbcat_l
_mbsnbcmp, _mbsnbcmp_l
_mbsnbcnt, _mbsnbcnt_l, _mbsnccnt, _mbsnccnt_l, _strncnt, _wcsncnt
_mbsnbicmp, _mbsnbicmp_l
_mbsnbset, _mbsnbset_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l

_mbsnbicmp, _mbsnbicmp_l

Compares *n* bytes of two multibyte-character strings, ignoring case.

```
int _mbsnbicmp(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count
);
```

Parameters

string1, *string2*

Null-terminated strings to compare.

count

Number of bytes to compare.

Return Value

The return value indicates the relationship between the substrings.

Return value	Description
< 0	*string1* substring less than *string2* substring.
0	*string1* substring identical to *string2* substring.
> 0	*string1* substring greater than *string2* substring.

On an error, **_mbsnbcmp** returns **_NLSCMPERROR**, which is defined in String.h and Mbstring.h.

Remarks

The **_mbsnbicmp** function lexicographically compares, at most, the first *count* bytes of *string1* and *string2*. The comparison is performed without regard to case; **_mbsnbcmp** is a case-sensitive version of **_mbsnbicmp**. The comparison ends if a terminating null character is reached in either string before *count* characters are compared. If the strings are equal when a terminating null character is reached in either string before *count* characters are compared, the shorter string is lesser.

_mbsnbicmp is similar to **_mbsnicmp**, except that it compares strings by bytes instead of by characters.

Two strings containing characters located between 'Z' and 'a' in the ASCII table ('[', '\', ']', '^', '_', and '`') compare differently, depending on their case. For example, the two strings "`ABCDE`" and "`ABCD^`" compare one way if the comparison is lowercase ("`abcde`" > "`abcd^`") and the other way ("`ABCDE`" < "`ABCD^`") if it is uppercase.

_mbsnbicmp recognizes multibyte-character sequences according to the multibyte code page currently in use. It is not affected by the current locale setting.

If either *string1* or *string2* is a null pointer, **_mbsnbicmp** invokes the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, the function returns **_NLSCMPERROR** and sets **errno** to **EINVAL**.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsnicmp	**_strnicmp**	**_mbsnbicmp**	**_wcsnicmp**
_tcsnicmp_l	**_strnicmp_l**	**_mbsnbicmp_l**	**_wcsnicmp_l**

Requirements

Routine	Required header	Compatibility

_mbsnbicmp	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example for _mbsnbcmp, _mbsnbcmp_l.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

String Manipulation (CRT)
_mbsnbcat, _mbsnbcat_l
_mbsnbcmp, _mbsnbcmp_l
_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l

_mbsnbset, _mbsnbset_l

Set the first *n* bytes of a multibyte-character string to a specified character. These functions have been deprecated because more secure versions are available; see _mbsnbset_s, _mbsnbset_s_l.

```
unsigned char *_mbsnbset(
   unsigned char *str,
   unsigned int c,
   size_t count
);
unsigned char *_mbsnbset_l(
   unsigned char *str,
   unsigned int c,
   size_t count,
   _locale_t locale
);
```

Parameters

str

String to be altered.

c

Single-byte or multibyte-character setting.

count

Number of bytes to be set.

locale

Locale to use.

Return Value

_mbsnbset returns a pointer to the altered string.

Remarks

The **_mbsnbset** and **_mbsnbset_l** functions set, at most, the first *count* bytes of *str* to *c*. If *count* is greater than the length of *str*, the length of *str* is used instead of *count*. If *c* is a multibyte character and cannot be set entirely into the last byte specified by *count*, the last byte is padded with a blank character. **_mbsnbset** and **_mbsnbset_l** does not place a terminating null at the end of *str*.

_mbsnbset and **_mbsnbset_l** is similar to **_mbsnset**, except that it sets *count* bytes rather than *count* characters of *c*.

If *str* is **NULL** or *count* is zero, this function generates an invalid parameter exception as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **NULL**. Also, if *c* is not a valid multibyte character, **errno** is set to **EINVAL** and a space is used instead.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The **_mbsnbset** version of this function uses the current locale for this locale-dependent behavior; the **_mbsnbset_l** version is identical except that it use the locale parameter passed in instead.

Security Note This API incurs a potential threat brought about by a buffer overrun problem. Buffer overrun problems are a frequent method of system attack, resulting in an unwarranted elevation of privilege. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsnset	**_strnset**	**_mbsnbset**	**_wcsnset**
_tcsnset_l	**_strnset_l**	**_mbsnbset_l**	**_wcsnset_l**

Requirements

Routine	Required header	Compatibility
_mbsnbset	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsnbset_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_mbsnbset.c
// compile with: /W3
#include <mbstring.h>
#include <stdio.h>

int main( void )
{
   char string[15] = "This is a test";
   /* Set not more than 4 bytes of string to be *'s */
   printf( "Before: %s\n", string );
   _mbsnbset( string, '*', 4 ); // C4996
   // Note; _mbsnbset is deprecated; consider _mbsnbset_s
   printf( "After:  %s\n", string );
}
```

Output

```
Before: This is a test
After:  **** is a test
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

String Manipulation (CRT)
_mbsnbcat, _mbsnbcat_l
_strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l

_mbsnbset_s, _mbsnbset_s_l

Set the first *n* bytes of a multibyte-character string to a specified character. Versions of _mbsnbset, _mbsnbset_l with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _mbsnbset_s(
   unsigned char *str,
   size_t size,
   unsigned int c,
   size_t count
);
errno_t _mbsnbset_s_l(
   unsigned char *str,
   size_t size,
   unsigned int c,
   size_t count,
   _locale_t locale
);
template <size_t size>
errno_t _mbsnbset_s(
   unsigned char (&str)[size],
   unsigned int c,
   size_t count
); // C++ only
template <size_t size>
errno_t _mbsnbset_s_l(
   unsigned char (&str)[size],
   unsigned int c,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

str

String to be altered.

size

The size of the string buffer.

c

Single-byte or multibyte-character setting.

count

Number of bytes to be set.

locale

Locale to use.

Return Value

Zero if successful; an error code otherwise.

Remarks

The **_mbsnbset_s** and **_mbsnbset_s_l** functions set, at most, the first *count* bytes of *str* to *c*. If *count* is greater than the length of *str*, the length of *str* is used instead of *count*. If *c* is a multibyte character and cannot be set entirely into the last byte specified by *count*, the last byte is padded with a blank character. **_mbsnbset_s** and **_mbsnbset_s_l** do not place a terminating null at the end of *str*.

_mbsnbset_s and **_mbsnbset_s_l** are similar to **_mbsnset**, except that they set *count* bytes rather than *count* characters of *c*.

If *str* is **NULL** or *count* is zero, this function generates an invalid parameter exception as described in Parameter Validation. If

execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **NULL**. Also, if *c* is not a valid multibyte character, **errno** is set to **EINVAL** and a space is used instead.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The **_mbsnbset_s** version of this function uses the current locale for this locale-dependent behavior; the **_mbsnbset_s_l** version is identical except that it use the locale parameter passed in instead.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically, eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsnset_s	**_strnset_s**	**_mbsnbset_s**	**_wcsnset_s**
_tcsnset_s_l	**_strnset_s _l**	**_mbsnbset_s_l**	**_wcsnset_s_l**

Requirements

Routine	Required header	Compatibility
_mbsnbset_s	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsnbset_s_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_mbsnbset_s.c
#include <mbstring.h>
#include <stdio.h>

int main( void )
{
   char string[15] = "This is a test";
   /* Set not more than 4 bytes of string to be *'s */
   printf( "Before: %s\n", string );
   _mbsnbset_s( string, sizeof(string), '*', 4 );
   printf( "After:  %s\n", string );
}
```

Output

```
Before: This is a test
After:  **** is a test
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

String Manipulation (CRT)
_mbsnbcat, _mbsnbcat_l
_strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l

_mbsnextc, _mbsnextc_l, _strnextc, _wcsnextc

Finds the next character in a string.

```
unsigned int _mbsnextc(
   const unsigned char *str
);
unsigned int _mbsnextc_l(
   const unsigned char *str,
   _locale_t locale
);
unsigned int _strnextc(
   const char *str
);
unsigned int _wscnextc(
   const wchar_t *str
);
```

Parameters

str

Source string.

locale

Locale to use.

Return Value

Each of these functions returns the integer value of the next character in *str*.

Remarks

The **_mbsnextc** function returns the integer value of the next multibyte character in *str*, without advancing the string pointer. **_mbsnextc** recognizes multibyte-character sequences according to the multibyte code page currently in use.

If *str* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns 0.

Security Note This API incurs a potential threat brought about by a buffer overrun problem. Buffer overrun problems are a frequent method of system attack, resulting in an unwarranted elevation of privilege. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsnextc	**_strnextc**	**_mbsnextc**	**_wcsnextc**

_strnextc and **_wcsnextc** are single-byte–character string and wide-character string versions of **_mbsnextc**. **_wcsnextc** returns the integer value of the next wide character in *string*; **_strnextc** returns the integer value of the next single-byte character in *string*. **_strnextc** and **_wcsnextc** are provided only for this mapping and should not be used otherwise. **_mbsnextc_l** is identical except that it uses the locale parameter passed in instead. For more information, see Using Generic-Text Mappings and Generic-Text Mappings.

Requirements

Routine	Required header	Compatibility
_mbsnextc	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsnextc_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_strnextc	<tchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsnextc	<tchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
_mbsdec, _mbsdec_l, _strdec, _wcsdec
_mbsinc, _mbsinc_l, _strinc, _wcsinc
_mbsninc, _mbsninc_l, _strninc, _wcsninc

_mbsninc, _mbsninc_l, _strninc, _wcsninc

Advances a string pointer by *n* characters.

```
unsigned char *_mbsninc(
   const unsigned char *str,
   size_t count
);
unsigned char *_mbsninc(
   const unsigned char *str,
   size_t count,
   _locale_t locale
);
char *_strninc(
   const char *str,
   size_t count
);
wchar_t *_wcsninc(
   const wchar_t *str,
   size_t count
);
```

Parameters

str

Source string.

count

Number of characters to increment a string pointer.

locale

Locale to use.

Return Value

Each of these routines returns a pointer to *str* after *str* has been incremented by *count* characters or **NULL** if the supplied pointer is **NULL**. If *count* is greater than or equal to the number of characters in *str*, the result is undefined.

Remarks

The **_mbsninc** function increments *str* by *count* multibyte characters. **_mbsninc** recognizes multibyte-character sequences according to the multibyte code page currently in use.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsninc	**_strninc**	**_mbsninc**	**_wcsninc**

_strninc and **_wcsninc** are single-byte–character string and wide-character string versions of **_mbsninc**. **_wcsninc** and **_strninc** are provided only for this mapping and should not be used otherwise. **_mbsninc_l** is identical except that it uses the locale parameter passed in instead. For more information, see Using Generic-Text Mappings and Generic-Text Mappings.

Requirements

Routine	Required header	Compatibility
_mbsninc	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsninc_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_strninc	<tchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsninc	<tchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
_mbsdec, _mbsdec_l, _strdec, _wcsdec
_mbsinc, _mbsinc_l, _strinc, _wcsinc
_mbsnextc, _mbsnextc_l, _strnextc, _wcsnextc

_mbsspnp, _mbsspnp_l, _strspnp, _wcsspnp

Returns a pointer to the first character in a given string that is not in another given string.

```
unsigned char *_mbsspnp(
   const unsigned char *str,
   const unsigned char *charset
);
unsigned char *_mbsspnp_l(
   const unsigned char *str,
   const unsigned char *charset,
   _locale_t locale
);
char *_strspnp(
   const char *str,
   const char *charset
);
wchar_t *_wcsspnp(
   const unsigned wchar_t *str,
   const unsigned wchar_t *charset
);
```

Parameters

str

Null-terminated string to search.

charset

Null-terminated character set.

locale

Locale to use.

Return Value

_strspnp, **_wcsspnp**, and **_mbsspnp** return a pointer to the first character in *str* that does not belong to the set of characters in *charset*. Each of these functions returns **NULL** if *str* consists entirely of characters from *charset*. For each of these routines, no return value is reserved to indicate an error.

Remarks

The **_mbsspnp** function returns a pointer to the multibyte character that is the first character in *str* that does not belong to the set of characters in *charset*. **_mbsspnp** recognizes multibyte-character sequences according to the multibyte code page currently in use. The search does not include terminating null characters.

If either *str* or *charset* is a null pointer, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the function returns **NULL** and sets **errno** to **EINVAL**.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsspnp	**_strspnp**	**_mbsspnp**	**_wcsspnp**

_strspnp and **_wcsspnp** are single-byte character and wide-character versions of **_mbsspnp**. **_strspnp** and **_wcsspnp** behave identically to **_mbsspnp** otherwise; they are provided only for this mapping and should not be used for any other reason. **_mbsspnp_l** is identical except that it uses the locale parameter passed in instead. For more information, see Using Generic-Text Mappings and Generic-Text Mappings.

Requirements

Routine	Required header	Compatibility

_mbsspnp	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003, Windows Server 2003
_strspnp	<tchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003, Windows Server 2003
_wcsspnp	<tchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_mbsspnp.c
#include <mbstring.h>
#include <stdio.h>

int main( void ) {
   const unsigned char string1[] = "cabbage";
   const unsigned char string2[] = "c";
   unsigned char *ptr = 0;
   ptr = _mbsspnp( string1, string2 );
   printf( "%s\n", ptr);
}
```

Output

```
abbage
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
strspn, wcsspn, _mbsspn, _mbsspn_l
strncat, _strncat_l, wcsncat, wcsncat_l, _mbsncat _mbsncat_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l

mbsrtowcs

Converts a multibyte character string to a corresponding wide characters string. This function is deprecated because a more secure version is available; see mbsrtowcs.

```
size_t mbsrtowcs(
   wchar_t *wcstr,
   const char **mbstr,
   sizeof count,
   mbstate_t *mbstate
);
template <size_t size>
size_t mbsrtowcs(
   wchar_t (&wcstr)[size],
   const char **mbstr,
   sizeof count,
   mbstate_t *mbstate
); // C++ only
```

Parameters

[out] *wcstr*

The resulting converted wide character string's address location.

[in] *mbstr*

Indirectly points to the location of the multibyte character string to be converted.

[in] *count*

The number of character to be converted.

[in] *mbstate*

A pointer to an **mbstate_t** conversion state object.

Return Value

Returns the number of words successfully converted, not including the null terminating null word (if any), otherwise a -1 if an error occurred.

Remarks

The **mbsrtowcs** function converts a string of multibyte characters, beginning in the specified conversion state contained in *mbstate*, from the values indirect pointed to in *mbstr*, into the address of *wcstr*. The conversion will continue for each character until: after a null terminating multibyte character is encountered, when a non corresponding character is encountered or when the next character would exceed the limit contained in *count*. If **mbsrtowcs** encounters the multibyte null character ('\0') either before or when *count* occurs, it converts it to a 16-bit null terminating character and stops.

Thus, the wide character string at *wcstr* is null-terminated only if **mbsrtowcs** encounters a multibyte null character during conversion. If the sequences pointed to by *mbstr* and *wcstr* overlap, the behavior of **mbsrtowcs** is undefined. **mbsrtowcs** is affected by the LC_TYPE category of the current locale.

The **mbsrtowcs** function differs from mbstowcs, _mbstowcs_l by its restartability. The conversion state is stored in *mbstate* for subsequent calls to the same or other restartable functions. Results are undefined when mixing the use of restartable and nonrestartable functions. For example, an application would use **mbsrlen** rather than **mbslen**, if a subsequent call to **mbsrtowcs** where used instead of **mbstowcs.**

If the *wcstr* argument is **NULL**, **mbsrtombs** returns the required size in words of the destination string. If *mbstate* is null, the internal **mbstate_t** conversion state is used. If the character sequence *wchar* does not have a corresponding multibyte character representation, a -1 is returned and the **errno** is set to **EILSEQ**.

If *mbstr* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns -1.

In C++, this function has a template overload that invokes the newer, secure counterpart of this function. For more

information, see Secure Template Overloads.

Exceptions

The **mbsrtowcs** function is multithread safe as long as no function in the current thread calls **setlocale** while this function is executing and the *mbstate* is null.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

Requirements

Routine	Required header	Compatibility
mbsrtowcs	<wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

See Also

Reference

Data Conversion
Locale
Interpretation of Multibyte-Character Sequences
mbrtowc
mbtowc, _mbtowc_l
mbstowcs, _mbstowcs_l
mbsinit

mbsrtowcs_s

Convert a wide character string to its multibyte character string representation. A version of mbsrtowcs with security enhancements as described in Security Enhancements in the CRT.

```
errno_t mbsrtowcs_s(
   size_t *pReturnValue,
   wchar_t *wcstr,
   size_t sizeInWords,
   const char **mbstr,
   size_t count,
   mbstate_t *mbstate
);
template <size_t size>
errno_t mbsrtowcs_s(
   size_t *pReturnValue,
   wchar_t (&wcstr)[size],
   const char **mbstr,
   size_t count,
   mbstate_t *mbstate
); // C++ only
```

Parameters

[out] *pReturnValue*

The number of characters converted.

[out] *wcstr*

Address of buffer for the resulting converted wide character string.

[out] *sizeInWords*

The size of *wcstr* in words (wide characters).

[in] *mbstr*

Points to the location of the multibyte character string to be converted.

[in] *count*

The maximum number of wide characters to store in the *wcstr* buffer, not including the terminating null, or _TRUNCATE.

[in] *mbstate*

A pointer to an **mbstate_t** conversion state object.

Return Value

Zero if successful, an error code on failure.

Error condition	Return value and errno
wcstr is **NULL** and *sizeInWords* > 0	**EINVAL**
mbstr is **NULL**	**EINVAL**
The destination buffer is too small to contain the converted string (unless *count* is **_TRUNCATE**; see Remarks below)	**ERANGE**

If any of these conditions occurs, the invalid parameter exception is invoked as described in Parameter Validation . If execution is allowed to continue, the function returns an error code and sets **errno** as indicated in the table.

Remarks

The **mbsrtowcs_s** function converts a string of multibyte characters pointed to by *mbstr* into wide characters stored in the buffer pointed to by *wcstr*, using the conversion state contained in *mbstate*. The conversion will continue for each character until one of these conditions is met:

- A multibyte null character is encountered
- An invalid multibyte character is encountered
- The number of wide characters stored in the *wcstr* buffer equals *count*.

The destination string is always null-terminated (even in the case of an error).

If *count* is the special value _TRUNCATE, then **mbsrtowcs_s** converts as much of the string as will fit into the destination buffer, while still leaving room for a null terminator.

If **mbsrtowcs_s** successfully converts the source string, it puts the size in wide characters of the converted string, including the null terminator, into **pReturnValue* (provided *pReturnValue* is not **NULL**). This occurs even if the *wcstr* argument is **NULL** and provides a way to determine the required buffer size. Note that if *wcstr* is **NULL**, *count* is ignored.

If *mbstate* is **NULL**, the internal **mbstate_t** conversion state is used.

If **mbsrtowcs_s** encounters an invalid multibyte character, it puts -1 in **pReturnValue*, sets the destination buffer to an empty string, sets **errno** to **EILSEQ**, and returns **EILSEQ**.

If the sequences pointed to by *mbstr* and *wcstr* overlap, the behavior of **mbsrtowcs_s** is undefined. **mbsrtowcs_s** is affected by the LC_TYPE category of the current locale.

Security Note
Ensure that *wcstr* and *mbstr* do not overlap, and that *count* correctly reflects the number of multibyte characters to convert.

The **mbsrtowcs_s** function differs from mbstowcs_s, _mbstowcs_s_l by its restartability. The conversion state is stored in *mbstate* for subsequent calls to the same or other restartable functions. Results are undefined when mixing the use of restartable and nonrestartable functions. For example, an application would use **mbsrlen** rather than **mbslen**, if a subsequent call to **mbsrtowcs_s** were used instead of **mbstowcs_s.**

In C++, using this function is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Exceptions

The **mbsrtowcs** function is multithread safe as long as no function in the current thread calls **setlocale** while this function is executing and the *mbstate* is null.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

Requirements

Routine	**Required header**	**Compatibility**
mbsrtowcs	<wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

See Also

Reference

Data Conversion
Locale
Interpretation of Multibyte-Character Sequences
mbrtowc
mbtowc, _mbtowc_l
mbstowcs_s, _mbstowcs_s_l
mbsinit

mbstowcs, _mbstowcs_l

Converts a sequence of multibyte characters to a corresponding sequence of wide characters. These functions are deprecated because more secure versions are available; see mbstowcs_s, _mbstowcs_s_l.

```
size_t mbstowcs(
   wchar_t *wcstr,
   const char *mbstr,
   size_t count
);
size_t _mbstowcs_l(
   wchar_t *wcstr,
   const char *mbstr,
   size_t count,
   _locale_t locale
);
template <size_t size>
size_t mbstowcs(
   wchar_t (&wcstr)[size],
   const char *mbstr,
   size_t count
); // C++ only
template <size_t size>
size_t _mbstowcs_l(
   wchar_t (&wcstr)[size],
   const char *mbstr,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

[out] *wcstr*

The address of a sequence of wide characters.

[in] *mbstr*

The address of a sequence of null terminated multibyte characters.

[in] *count*

The maximum number of multibyte characters to convert.

[in] *locale*

The locale to use.

Return Value

If **mbstowcs** successfully converts the source string, it returns the number of converted multibyte characters. If the *wcstr* argument is **NULL**, the function returns the required size (in wide characters) of the destination string. If **mbstowcs** encounters an invalid multibyte character, it returns –1. If the return value is *count*, the wide-character string is not null-terminated.

Security Note
Ensure that *wcstr* and *mbstr* do not overlap, and that *count* correctly reflects the number of multibyte characters to convert.

Remarks

The **mbstowcs** function converts up to a maximum number of *count* multibyte characters pointed to by *mbstr* to a string of corresponding wide characters that are determined by the current locale. It stores the resulting wide-character string at the address represented by *wcstr*. The result is similar to a series of calls to **mbtowc**. If **mbstowcs** encounters the single-byte null character ('\0') either before or when *count* occurs, it converts the null character to a wide-character null character (L'\0') and stops. Thus the wide-character string at *wcstr* is null-terminated only if a null character is encountered during conversion. If the sequences pointed to by *wcstr* and *mbstr* overlap, the behavior is undefined.

If the *wcstr* argument is **NULL**, **mbstowcs** returns the number of wide characters that would result from conversion, not including a null terminator. The source string must be null-terminated for the correct value to be returned. If you need the resulting wide character string to be null-terminated, add one to the returned value.

If the *mbstr* argument is **NULL**, or if *count* is > **INT_MAX**, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, errno is set to **EINVAL** and the function returns -1.

mbstowcs uses the current locale for any locale-dependent behavior; **_mbstowcs_l** is identical except that it uses the locale passed in instead.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Requirements

Routine	Required header	Compatibility
mbstowcs	<stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbstowcs_l	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_mbstowcs.c
// compile with: /W3
// illustrates the behavior of the mbstowcs function

#include <stdlib.h>
#include <stdio.h>
#include <locale.h>

int main( void )
{
    size_t size;
    int nChar = 2; // number of characters to convert
    int requiredSize;

    unsigned char    *pmbnull  = NULL;
    unsigned char    *pmbhello = NULL;
    char* localeInfo;

    wchar_t *pwchello = L"\x3042\x3043"; // 2 Hiragana characters
    wchar_t *pwc;

    /* Enable the Japanese locale and codepage */
    localeInfo = setlocale(LC_ALL, "Japanese_Japan.932");
    printf("Locale information set to %s\n", localeInfo);

    printf( "Convert to multibyte string:\n" );

    requiredSize = wcstombs( NULL, pwchello, 0); // C4996
    // Note: wcstombs is deprecated; consider using wcstombs_s
    printf("  Required Size: %d\n", requiredSize);

    /* Add one to leave room for the null terminator. */
    pmbhello = (unsigned char *)malloc( requiredSize + 1);
    if (! pmbhello)
    {
        printf("Memory allocation failure.\n");
        return 1;
    }
    size = wcstombs( pmbhello, pwchello, requiredSize + 1); // C4996
```

```
    // Note: wcstombs is deprecated; consider using wcstombs_s
    if (size == (size_t) (-1))
    {
        printf("Couldn't convert string. Code page 932 may"
                " not be available.\n");
        return 1;
    }
    printf( "  Number of bytes written to multibyte string: %u\n",
            (unsigned int) size );
    printf( "  Hex values of the " );
    printf( " multibyte characters: %#.2x %#.2x %#.2x %#.2x\n",
            pmbhello[0], pmbhello[1], pmbhello[2], pmbhello[3] );
    printf( "  Codepage 932 uses 0x81 to 0x9f as lead bytes.\n\n");

    printf( "Convert back to wide-character string:\n" );

    /* Assume we don't know the length of the multibyte string.
     Get the required size in characters, and allocate enough space. */

    requiredSize = mbstowcs(NULL, pmbhello, 0); // C4996
    /* Add one to leave room for the NULL terminator */
    pwc = (wchar_t *)malloc( (requiredSize + 1) * sizeof( wchar_t ));
    if (! pwc)
    {
        printf("Memory allocation failure.\n");
        return 1;
    }
    size = mbstowcs( pwc, pmbhello, requiredSize + 1); // C4996
    if (size == (size_t) (-1))
    {
       printf("Couldn't convert string--invalid multibyte character.\n");
    }
    printf( "  Characters converted: %u\n", (unsigned int)size );
    printf( "  Hex value of first 2" );
    printf( " wide characters: %#.4x %#.4x\n\n", pwc[0], pwc[1] );
    free(pwc);
    free(pmbhello);
}
```

Sample Output

```
Locale information set to Japanese_Japan.932
Convert to multibyte string:
  Required Size: 4
  Number of bytes written to multibyte string: 4
  Hex values of the  multibyte characters: 0x82 0xa0 0x82 0xa1
  Codepage 932 uses 0x81 to 0x9f as lead bytes.

Convert back to wide-character string:
  Characters converted: 2
  Hex value of first 2 wide characters: 0x3042 0x3043
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Data Conversion
Locale
Interpretation of Multibyte-Character Sequences
_mbclen, mblen, _mblen_l
mbtowc, _mbtowc_l
wcstombs, _wcstombs_l
wctomb, _wctomb_l
MultiByteToWideChar

mbstowcs_s, _mbstowcs_s_l

Converts a sequence of multibyte characters to a corresponding sequence of wide characters. Versions of mbstowcs, _mbstowcs_l with security enhancements as described in Security Enhancements in the CRT.

```
errno_t mbstowcs_s(
   size_t *pReturnValue,
   wchar_t *wcstr,
   size_t sizeInWords,
   const char *mbstr,
   size_t count
);
errno_t _mbstowcs_s_l(
   size_t *pReturnValue,
   wchar_t *wcstr,
   size_t sizeInWords,
   const char *mbstr,
   size_t count,
   _locale_t locale
);
template <size_t size>
errno_t mbstowcs_s(
   size_t *pReturnValue,
   wchar_t (&wcstr)[size],
   const char *mbstr,
   size_t count
); // C++ only
template <size_t size>
errno_t _mbstowcs_s_l(
   size_t *pReturnValue,
   wchar_t (&wcstr)[size],
   const char *mbstr,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

[out] *pReturnValue*

The number of characters converted.

[out] *wcstr*

Address of buffer for the resulting converted wide character string.

[in] *sizeInWords*

The size of the *wcstr* buffer in words.

[in] *mbstr*

The address of a sequence of null terminated multibyte characters.

[in] *count*

The maximum number of wide characters to store in the *wcstr* buffer, not including the terminating null, or _TRUNCATE.

[in] *locale*

The locale to use.

Return Value

Zero if successful, an error code on failure.

Error condition	Return value and errno

wcstr is **NULL** and *sizeInWords* > 0	**EINVAL**
mbstr is **NULL**	**EINVAL**
The destination buffer is too small to contain the converted string (unless *count* is **_TRUNCATE**; see Remarks below)	**ERANGE**

If any of these conditions occurs, the invalid parameter exception is invoked as described in Parameter Validation . If execution is allowed to continue, the function returns an error code and sets **errno** as indicated in the table.

Remarks

The **mbstowcs_s** function converts a string of multibyte characters pointed to by *mbstr* into wide characters stored in the buffer pointed to by *wcstr*. The conversion will continue for each character until one of these conditions is met:

- A multibyte null character is encountered
- An invalid multibyte character is encountered
- The number of wide characters stored in the *wcstr* buffer equals *count*.

The destination string is always null-terminated (even in the case of an error).

If *count* is the special value _TRUNCATE, then **mbstowcs_s** converts as much of the string as will fit into the destination buffer, while still leaving room for a null terminator.

If **mbstowcs_s** successfully converts the source string, it puts the size in wide characters of the converted string, including the null terminator, into **pReturnValue* (provided *pReturnValue* is not **NULL**). This occurs even if the *wcstr* argument is **NULL** and provides a way to determine the required buffer size. Note that if *wcstr* is **NULL**, *count* is ignored.

If **mbstowcs_s** encounters an invalid multibyte character, it puts 0 in **pReturnValue*, sets the destination buffer to an empty string, sets **errno** to **EILSEQ**, and returns **EILSEQ**.

If the sequences pointed to by *mbstr* and *wcstr* overlap, the behavior of **mbstowcs_s** is undefined.

Security Note
Ensure that *wcstr* and *mbstr* do not overlap, and that *count* correctly reflects the number of multibyte characters to convert.

mbstowcs_s uses the current locale for any locale-dependent behavior; **_mbstowcs_s_l** is identical except that it uses the locale passed in instead.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Requirements

Routine	**Required header**	**Compatibility**
mbstowcs_s	<stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbstowcs_s_l	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Data Conversion
Locale
MultiByteToWideChar
Interpretation of Multibyte-Character Sequences
_mbclen, mblen, _mblen_l
mbtowc, _mbtowc_l
wcstombs, _wcstombs_l
wctomb, _wctomb_l

mbtowc, _mbtowc_l

Convert a multibyte character to a corresponding wide character.

```
int mbtowc(
   wchar_t *wchar,
   const char *mbchar,
   size_t count
);
int _mbtowc_l(
   wchar_t *wchar,
   const char *mbchar,
   size_t count,
   _locale_t locale
);
```

Parameters

wchar

Address of a wide character (type **wchar_t**).

mbchar

Address of a sequence of bytes (a multibyte character).

count

Number of bytes to check.

locale

The locale to use.

Return Value

If **mbchar** is not **NULL** and if the object that *mbchar* points to forms a valid multibyte character, **mbtowc** returns the length in bytes of the multibyte character. If *mbchar* is **NULL** or the object that it points to is a wide-character null character (L'\0'), the function returns 0. If the object that *mbchar* points to does not form a valid multibyte character within the first *count* characters, it returns –1.

Remarks

The **mbtowc** function converts *count* or fewer bytes pointed to by *mbchar*, if *mbchar* is not **NULL**, to a corresponding wide character. **mbtowc** stores the resulting wide character at *wchar*, if *wchar* is not **NULL**. **mbtowc** does not examine more than **MB_CUR_MAX** bytes. **mbtowc** uses the current locale for locale-dependent behavior; **_mbtowc_l** is identical except that it uses the locale passed in instead.

Requirements

Routine	Required header	Compatibility
mbtowc	<stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbtowc_l	<stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_mbtowc.c
/* Illustrates the behavior of the mbtowc function
 */

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    int      i;
    char    *pmbc    = (char *)malloc( sizeof( char ) );
    wchar_t  wc      = L'a';
    wchar_t *pwcnull = NULL;
    wchar_t *pwc     = (wchar_t *)malloc( sizeof( wchar_t ) );
    printf( "Convert a wide character to multibyte character:\n" );
    wctomb_s( &i, pmbc, sizeof(char), wc );
    printf( "  Characters converted: %u\n", i );
    printf( "  Multibyte character: %x\n\n", *pmbc );

    printf( "Convert multibyte character back to a wide "
            "character:\n" );
    i = mbtowc( pwc, pmbc, MB_CUR_MAX );
    printf( "  Bytes converted: %u\n", i );
    printf( "  Wide character: %x\n\n", *pwc );
    printf( "Attempt to convert when target is NULL\n" );
    printf( "  returns the length of the multibyte character:\n" );
    i = mbtowc( pwcnull, pmbc, MB_CUR_MAX );
    printf( "  Length of multibyte character: %u\n\n", i );

    printf( "Attempt to convert a NULL pointer to a" );
    printf( " wide character:\n" );
    pmbc = NULL;
    i = mbtowc( pwc, pmbc, MB_CUR_MAX );
    printf( "  Bytes converted: %u\n", i );
}
```

Output

```
Convert a wide character to multibyte character:
  Characters converted: 1
  Multibyte character: 61

Convert multibyte character back to a wide character:
  Bytes converted: 1
  Wide character: 61

Attempt to convert when target is NULL
  returns the length of the multibyte character:
  Length of multibyte character: 1

Attempt to convert a NULL pointer to a wide character:
  Bytes converted: 0
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Data Conversion
MultiByteToWideChar
Locale
Interpretation of Multibyte-Character Sequences
_mbclen, mblen, _mblen_l
wcstombs, _wcstombs_l
wctomb, _wctomb_l

memccpy

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _memccpy instead.

_memccpy

Copies characters from a buffer.

```
void *_memccpy(
   void *dest,
   const void *src,
   int c,
   size_t count
);
```

Parameters

dest

Pointer to the destination.

src

Pointer to the source.

c

Last character to copy.

count

Number of characters.

Return Value

If the character *c* is copied, **_memccpy** returns a pointer to the char in *dest* that immediately follows the character. If *c* is not copied, it returns **NULL**.

Remarks

The **_memccpy** function copies 0 or more characters of *src* to *dest*, halting when the character *c* has been copied or when *count* characters have been copied, whichever comes first.

Security Note Make sure that the destination buffer is the same size or larger than the source buffer. For more information, see Avoiding Buffer Overruns.

Requirements

Routine	Required header	Compatibility
_memccpy	<memory.h> or <string.h>	Windows 98, Windows Me, Windows NT, Windows 2000, Windows XP, and Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_memccpy.c

#include <memory.h>
#include <stdio.h>
#include <string.h>

char string1[60] = "The quick brown dog jumps over the lazy fox";

int main( void )
{
   char buffer[61];
   char *pdest;
```

```
    printf( "Function: _memccpy 60 characters or to character 's'\n" );
    printf( "Source: %s\n", string1 );
    pdest = _memccpy( buffer, string1, 's', 60 );
    *pdest = '\0';
    printf( "Result: %s\n", buffer );
    printf( "Length: %d characters\n", strlen( buffer ) );
}
```

Output

```
Function: _memccpy 60 characters or to character 's'
Source: The quick brown dog jumps over the lazy fox
Result: The quick brown dog jumps
Length: 25 characters
```

.NET Framework Equivalent

- System::Buffer::BlockCopy
- System::String::Copy

See Also

Reference

Buffer Manipulation
memchr, wmemchr
memcmp, wmemcmp
memcpy, wmemcpy
memset, wmemset

memchr, wmemchr

Find characters in a buffer.

```
void *memchr(
   const void *buf,
   int c,
   size_t count
); // C only
void *memchr(
   void *buf,
   int c,
   size_t count
); // C++ only
const void *memchr(
   const void *buf,
   int c,
   size_t count
); // C++ only
wchar_t *wmemchr(
   const wchar_t * buf,
   wchar_t c,
   size_t count
); // C only
wchar_t *wmemchr(
   wchar_t * buf,
   wchar_t c,
   size_t count
); // C++ only
const wchar_t *wmemchr(
   const wchar_t * buf,
   wchar_t c,
   size_t count
); // C++ only
```

Parameters

buf

Pointer to buffer.

c

Character to look for.

count

Number of characters to check.

Return Value

If successful, returns a pointer to the first location of *c* in *buf*. Otherwise it returns **NULL**.

Remarks

memchr and **wmemchr** look for the first occurrence of *c* in the first *count* bytes of *buf*. It stops when it finds *c* or when it has checked the first *count* bytes.

In C, these functions take a **const** pointer for the first argument. In C++, two overloads are available. The overload taking a pointer to **const** returns a pointer to **const**; the version that takes a pointer to non-**const** returns a pointer to non-**const**. The macro _CONST_CORRECT_OVERLOADS is defined if both the **const** and non-**const** versions of these functions are available. If you require the non-**const** behavior for both C++ overloadsin C++, define the symbol _CONST_RETURN.

Requirements

Routine	Required header	Compatibility

memchr	<memory.h> or <string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wmemchr	<wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more information about compatibility, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_memchr.c

#include <memory.h>
#include <stdio.h>

int  ch = 'r';
char str[] =    "lazy";
char string[] = "The quick brown dog jumps over the lazy fox";
char fmt1[] =   "         1         2         3         4         5";
char fmt2[] =   "123456789012345678901234567890123456789012345678901234567890";

int main( void )
{
   char *pdest;
   int result;
   printf( "String to be searched:\n             %s\n", string );
   printf( "             %s\n             %s\n\n", fmt1, fmt2 );

   printf( "Search char: %c\n", ch );
   pdest = memchr( string, ch, sizeof( string ) );
   result = (int)(pdest - string + 1);
   if ( pdest != NULL )
      printf( "Result:      %c found at position %d\n", ch, result );
   else
      printf( "Result:      %c not found\n" );
}
```

Output

```
String to be searched:
             The quick brown dog jumps over the lazy fox
                      1         2         3         4         5
             123456789012345678901234567890123456789012345678901234567890

Search char: r
Result:      r found at position 12
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Buffer Manipulation
_memccpy
memcmp, wmemcmp
memcpy, wmemcpy
memset, wmemset
strchr, wcschr, _mbschr, _mbschr_l

memcmp, wmemcmp

Compare characters in two buffers.

```
int memcmp(
   const void *buf1,
   const void *buf2,
   size_t count
);
int wmemcmp(
   const wchar_t * buf1,
   const wchar_t * buf2,
   size_t count
);
```

Parameters

buf1

First buffer.

buf2

Second buffer.

count

Number of characters (bytes for **memcmp**, wide characters for **wmemcmp**).

Return Value

The return value indicates the relationship between the buffers.

Return value	Relationship of first count bytes of buf1 and buf2
< 0	*buf1* less than *buf2*
0	*buf1* identical to *buf2*
> 0	*buf1* greater than *buf2*

Remarks

Compares the first *count* characters of *buf1* and *buf2* and returns a value indicating their relationship.

Requirements

Routine	Required header	Compatibility
memcmp	<memory.h> or <string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wmemcmp	<wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_memcmp.c
```

```
/* This program uses memcmp to compare
 * the strings named first and second. If the first
 * 19 bytes of the strings are equal, the program
 * considers the strings to be equal.
 */

#include <string.h>
#include <stdio.h>

int main( void )
{
   char first[]  = "12345678901234567890";
   char second[] = "12345678901234567891";
   int int_arr1[] = {1,2,3,4};
   int int_arr2[] = {1,2,3,4};
   int result;

   printf( "Compare '%.19s' to '%.19s':\n", first, second );
   result = memcmp( first, second, 19 );
   if( result < 0 )
      printf( "First is less than second.\n" );
   else if( result == 0 )
      printf( "First is equal to second.\n" );
   else
      printf( "First is greater than second.\n" );

   printf( "Compare '%d,%d' to '%d,%d':\n", int_arr1[0], int_arr1[1], int_arr2[0], int_arr2
[1]);
   result = memcmp( int_arr1, int_arr2, sizeof(int) * 2 );
   if( result < 0 )
      printf( "int_arr1 is less than int_arr2.\n" );
   else if( result == 0 )
      printf( "int_arr1 is equal to int_arr2.\n" );
   else
      printf( "int_arr1 is greater than int_arr2.\n" );
}
```

Output

```
Compare '1234567890123456789' to '1234567890123456789':
First is equal to second.
Compare '1,2' to '1,2':
int_arr1 is equal to int_arr2.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Buffer Manipulation
_memccpy
memchr, wmemchr
memcpy, wmemcpy
memset, wmemset
strcmp, wcscmp, _mbscmp
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l

memcpy, wmemcpy

Copies bytes between buffers. These functions are deprecated because more secure versions are available; see memcpy_s, wmemcpy_s.

```
void *memcpy(
   void *dest,
   const void *src,
   size_t count
);
wchar_t *wmemcpy(
   wchar_t *dest,
   const wchar_t *src,
   size_t count
);
```

Parameters

dest

New buffer.

src

Buffer to copy from.

count

Number of characters to copy.

Return Value

The value of *dest*.

Remarks

memcpy copies *count* bytes from *src* to *dest*; **wmemcpy** copies *count* wide characters (two bytes). If the source and destination overlap, the behavior of **memcpy** is undefined. Use **memmove** to handle overlapping regions.

Security Note Make sure that the destination buffer is the same size or larger than the source buffer. For more information, see Avoiding Buffer Overruns.

The **memcpy** and **wmemcpy** functions will only be deprecated if the constant **_CRT_SECURE_DEPRECATE_MEMORY** is defined prior to the inclusion statement in order for the functions to be deprecated, such as in the example below:

```
#define _CRT_SECURE_DEPRECATE_MEMORY
#include <memory.h>
```

or

```
#define _CRT_SECURE_DEPRECATE_MEMORY
#include <wchar.h>
```

Requirements

Routine	Required header	Compatibility
memcpy	<memory.h> or <string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wmemcpy	<wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See memmove for a sample of how to use **memcpy**.

See Also

Reference

Buffer Manipulation
_memccpy
memchr, wmemchr
memcmp, wmemcmp
memmove, wmemmove
memset, wmemset
strcpy, wcscpy, _mbscpy
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l

memcpy_s, wmemcpy_s

Copies bytes between buffers. These are versions of memcpy, wmemcpy with security enhancements as described in Security Enhancements in the CRT.

```
errno_t memcpy_s(
   void *dest,
   size_t numberOfElements,
   const void *src,
   size_t count
);
errno_t wmemcpy_s(
   wchar_t *dest,
   size_t numberOfElements,
   const wchar_t *src,
   size_t count
);
```

Parameters

dest

New buffer.

numberOfElements

Size of the destination buffer.

src

Buffer to copy from.

count

Number of characters to copy.

Return Value

Zero if successful; an error code on failure.

Error Conditions

dest	***numberOfElements***	***src***	**Return value**	**Contents of *dest***
NULL	any	any	**EINVAL**	not modified
any	any	**NULL**	**EINVAL**	not modified
any	< *count*	any	**ERANGE**	not modified

Remarks

memcpy_s copies *count* bytes from *src* to *dest*; **wmemcpy_s** copies *count* wide characters (two bytes). If the source and destination overlap, the behavior of **memcpy_s** is undefined. Use **memmove_s** to handle overlapping regions.

These functions validate their parameters. If *dest* or *src* is a null pointer, or *numberOfElements* is too small for the buffer, these functions invoke the invalid parameter handler, as described in Parameter Validation . If execution is allowed to continue, these functions return **EINVAL** and set **errno** to **EINVAL**.

Requirements

Routine	**Required header**	**Compatibility**
memcpy_s	<memory.h> or <string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

wmemcpy_s	<wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_memcpy_s.c
// Copy memory in a more secure way.

#include <memory.h>
#include <stdio.h>

int main()
{
   int a1[10], a2[100], i;
   errno_t err;

   // Populate a2 with squares of integers
   for (i = 0; i < 100; i++)
   {
      a2[i] = i*i;
   }

   // Tell memcpy_s to copy 10 ints (40 bytes), giving
   // the size of the a1 array (also 40 bytes).
   err = memcpy_s(a1, sizeof(a1), a2, 10 * sizeof (int) );
   if (err)
   {
      printf("Error executing memcpy_s.\n");
   }
   else
   {
     for (i = 0; i < 10; i++)
       printf("%d ", a1[i]);
   }
   printf("\n");
}
```

Output

```
0 1 4 9 16 25 36 49 64 81
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Buffer Manipulation
_memccpy
memchr, wmemchr
memcmp, wmemcmp
memmove, wmemmove
memset, wmemset
strcpy, wcscpy, _mbscpy
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l

memicmp

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _memicmp instead.

_memicmp, _memicmp_l

Compares characters in two buffers (case-insensitive).

```
int _memicmp(
   const void *buf1,
   const void *buf2,
   size_t count
);
int _memicmp_l(
   const void *buf1,
   const void *buf2,
   size_t count,
   _locale_t locale
);
```

Parameters

buf1

First buffer.

buf2

Second buffer.

count

Number of characters.

locale

Locale to use.

Return Value

The return value indicates the relationship between the buffers.

Return value	Relationship of first count bytes of buf1 and buf2
< 0	*buf1* less than *buf2*.
0	*buf1* identical to *buf2*.
> 0	*buf1* greater than *buf2*.
_NLSCMPERROR	An error occurred.

Remarks

The **_memicmp** function compares the first *count* characters of the two buffers *buf1* and *buf2* byte by byte. The comparison is not case-sensitive.

If either *buf1* or *buf2* is a null pointer, this function invokes an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the function returns **_NLSCMPERROR** and sets **errno** to **EINVAL**.

_memicmp uses the current locale for locale-dependent behavior; **_memicmp_l** is identical except that it uses the locale passed in instead.

Requirements

Routine	Required header	Compatibility
_memicmp	<memory.h> or <string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_memicmp_l	<memory.h> or <string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_memicmp.c
// This program uses _memicmp to compare
// the first 29 letters of the strings named first and
// second without regard to the case of the letters.

#include <memory.h>
#include <stdio.h>
#include <string.h>

int main( void )
{
   int result;
   char first[] = "Those Who Will Not Learn from History";
   char second[] = "THOSE WHO WILL NOT LEARN FROM their mistakes";
   // Note that the 29th character is right here ^

   printf( "Compare '%.29s' to '%.29s'\n", first, second );
   result = _memicmp( first, second, 29 );
   if( result < 0 )
      printf( "First is less than second.\n" );
   else if( result == 0 )
      printf( "First is equal to second.\n" );
   else if( result > 0 )
      printf( "First is greater than second.\n" );
}
```

Output

```
Compare 'Those Who Will Not Learn from' to 'THOSE WHO WILL NOT LEARN FROM'
First is equal to second.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Buffer Manipulation
_memccpy
memchr, wmemchr
memcmp, wmemcmp
memcpy, wmemcpy
memset, wmemset
_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l

memmove, wmemmove

Moves one buffer to another. These functions are deprecated because more secure versions are available; see memmove_s, wmemmove_s.

```
void *memmove(
   void *dest,
   const void *src,
   size_t count
);
wchar_t *wmemmove(
   wchar_t *dest,
   const wchar_t *src,
   size_t count
);
```

Parameters

dest

Destination object.

src

Source object.

count

Number of bytes (**memmove**) or characters (**wmemmove**) to copy.

Return Value

The value of *dest*.

Remarks

Copies *count* bytes (**wmemmove**) or characters (**wmemmove**) from *src* to *dest*. If some regions of the source area and the destination overlap, both functions ensure that the original source bytes in the overlapping region are copied before being overwritten.

Security Note Make sure that the destination buffer is the same size or larger than the source buffer. For more information, see Avoiding Buffer Overruns.

The **memmove** and **wmemmove** functions will only be deprecated if the constant **_CRT_SECURE_DEPRECATE_MEMORY** is defined prior to the inclusion statement in order for the functions to be deprecated, such as in the example below:

```
#define _CRT_SECURE_DEPRECATE_MEMORY
#include <string.h>
or
#define _CRT_SECURE_DEPRECATE_MEMORY
#include <wchar.h>
```

Requirements

Routine	Required header	Compatibility
memmove	<string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wmemmove	<wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_memcpy.c
// Illustrate overlapping copy: memmove
// always handles it correctly; memcpy may handle
// it correctly.
//

#include <memory.h>
#include <string.h>
#include <stdio.h>

char str1[7] = "aabbcc";

int main( void )
{
   printf( "The string: %s\n", str1 );
   memcpy( str1 + 2, str1, 4 );
   printf( "New string: %s\n", str1 );

   strcpy_s( str1, sizeof(str1), "aabbcc" );   // reset string

   printf( "The string: %s\n", str1 );
   memmove( str1 + 2, str1, 4 );
   printf( "New string: %s\n", str1 );
}
```

Output

```
The string: aabbcc
New string: aaaabb
The string: aabbcc
New string: aaaabb
```

.NET Framework Equivalent

System::Buffer::BlockCopy

See Also

Reference

Buffer Manipulation
_memccpy
memcpy, wmemcpy
strcpy, wcscpy, _mbscpy
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l

memmove_s, wmemmove_s

Moves one buffer to another. These are versions of memmove, wmemmove with security enhancements as described in Security Enhancements in the CRT.

```
errno_t memmove_s(
   void *dest,
   size_t numberOfElements,
   const void *src,
   size_t count
);
errno_t wmemmove_s(
   wchar_t *dest,
   size_t numberOfElements,
   const wchar_t *src,
   size_t count
);
```

Parameters

dest

Destination object.

numberOfElements

Size of the destination buffer.

src

Source object.

count

Number of bytes of characters to copy.

Return Value

Zero if successful; an error code on failure

Error Conditions

dest	***numberOfElements***	***src***	**Return value**	**Contents of *dest***
NULL	any	any	**EINVAL**	not modified
any	any	**NULL**	**EINVAL**	not modified
any	< *count*	any	**ERANGE**	not modified

Remarks

Copies *count* bytes of characters from *src* to *dest*. If some regions of the source area and the destination overlap, **memmove_s** ensures that the original source bytes in the overlapping region are copied before being overwritten.

If *dest* or if *src* is a null pointer, or if the destination string is too small, these functions invoke an invalid parameter handler, as described in Parameter Validation . If execution is allowed to continue, these functions return **EINVAL** and set **errno** to **EINVAL**.

Requirements

Routine	**Required header**	**Compatibility**
memmove_s	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

wmemmove_s	<wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_memmove_s.c
//
// The program demonstrates the
// memmove_s function which works as expected
// for moving overlapping regions.

#include <stdio.h>
#include <string.h>

int main()
{
   char *str = "0123456789";

   printf("Before: %s\n", str);

   // Move six bytes from the start of the string
   // to a new position shifted by one byte. To protect against
   // buffer overrun, the secure version of memmove requires the
   // the length of the destination string to be specified.

   memmove_s((str + 1), strnlen(str + 1, 10), str, 6);

   printf_s(" After: %s\n", str);
}
```

Output

```
Before: 0123456789
 After: 0012345789
```

.NET Framework Equivalent

System::Buffer::BlockCopy

See Also

Reference

Buffer Manipulation
_memccpy
memcpy, wmemcpy
strcpy_s, wcscpy_s, _mbscpy_s
strcpy, wcscpy, _mbscpy
strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l

memset, wmemset

Sets buffers to a specified character.

```
void *memset(
   void *dest,
   int c,
   size_t count
);
wchar_t *wmemset(
   wchar_t *dest,
   wchar_t c,
   size_t count
);
```

Parameters

dest

Pointer to destination.

c

Character to set.

count

Number of characters.

Return Value

The value of *dest*.

Remarks

Sets the first *count* characters of *dest* to the character *c*.

Security Note Make sure that the destination buffer has enough room for at least *count* characters. For more information, see Avoiding Buffer Overruns.

Requirements

Routine	Required header	Compatibility
memset	<memory.h> or <string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wmemset	<wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_memset.c
/* This program uses memset to
 * set the first four chars of buffer to "*".
 */

#include <memory.h>
#include <stdio.h>
```

```
int main( void )
{
   char buffer[] = "This is a test of the memset function";

   printf( "Before: %s\n", buffer );
   memset( buffer, '*', 4 );
   printf( "After:  %s\n", buffer );
}
```

Output

```
Before: This is a test of the memset function
After:  **** is a test of the memset function
```

Here's an example of the use of wmemset:

```
// crt_wmemset.c
/* This program uses memset to
 * set the first four chars of buffer to "*".
 */

#include <wchar.h>
#include <stdio.h>

int main( void )
{
   wchar_t buffer[] = L"This is a test of the wmemset function";

   wprintf( L"Before: %s\n", buffer );
   wmemset( buffer, '*', 4 );
   wprintf( L"After:  %s\n", buffer );
}
```

Output

```
Before: This is a test of the wmemset function
After:  **** is a test of the wmemset function
```

.NET Framework Equivalent

System::Buffer::SetByte

See Also

Reference

Buffer Manipulation
_memccpy
memchr, wmemchr
memcmp, wmemcmp
memcpy, wmemcpy
_strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l

mkdir

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _mkdir instead.

_mkdir, _wmkdir

Creates a new directory.

```
int _mkdir(
   const char *dirname
);
int _wmkdir(
   const wchar_t *dirname
);
```

Parameters

dirname

Path for a new directory.

Return Value

Each of these functions returns the value 0 if the new directory was created. On an error, the function returns –1 and sets **errno** as follows.

EEXIST

Directory was not created because *dirname* is the name of an existing file, directory, or device.

ENOENT

Path was not found.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_mkdir** function creates a new directory with the specified *dirname*. **_mkdir** can create only one new directory per call, so only the last component of *dirname* can name a new directory. **_mkdir** does not translate path delimiters. In Windows NT, both the backslash (\) and the forward slash (/) are valid path delimiters in character strings in run-time routines.

_wmkdir is a wide-character version of **_mkdir**; the *dirname* argument to **_wmkdir** is a wide-character string. **_wmkdir** and **_mkdir** behave identically otherwise.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tmkdir	**_mkdir**	**_mkdir**	**_wmkdir**

Requirements

Routine	Required header	Compatibility
_mkdir	<direct.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wmkdir	<direct.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_makedir.c

#include <direct.h>
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   if( _mkdir( "\\testtmp" ) == 0 )
   {
      printf( "Directory '\\testtmp' was successfully created\n" );
      system( "dir \\testtmp" );
      if( _rmdir( "\\testtmp" ) == 0 )
        printf( "Directory '\\testtmp' was successfully removed\n"  );
      else
         printf( "Problem removing directory '\\testtmp'\n" );
   }
   else
      printf( "Problem creating directory '\\testtmp'\n" );
}
```

Sample Output

```
Directory '\testtmp' was successfully created
 Volume in drive C has no label.
 Volume Serial Number is E078-087A

 Directory of C:\testtmp

02/12/2002  09:56a      <DIR>          .
02/12/2002  09:56a      <DIR>          ..
               0 File(s)              0 bytes
               2 Dir(s)  15,498,690,560 bytes free
Directory '\testtmp' was successfully removed
```

.NET Framework Equivalent

- System::IO::Directory::CreateDirectory
- System::IO::DirectoryInfo::CreateSubdirectory

See Also

Reference

Directory Control
_chdir, _wchdir
_rmdir, _wrmdir

_mkgmtime, _mkgmtime32, _mkgmtime64

Converts a local time to a calendar value UTC time.

```
time_t _mkgmtime(
   struct tm* timeptr
);
__time32_t _mkgmtime32(
   struct tm* timeptr
);
__time64_t _mkgmtime64(
   struct tm* timeptr
);
```

Parameters

timeptr

A pointer to the time as a **struct tm** to convert.

Return Value

A quantity of type **__time32_t** or **__time64_t** representing the number of seconds elapsed since midnight, January 1, 1970, in Coordinated Universal Time (UTC). If the date is out of range (see the Remarks section) or the input cannot be interpreted as a valid time, the return value is –1.

Remarks

The **_mkgmtime32** and **_mkgmtime64** functions convert a local time to a **__time32_t** or **__time64_t** type representing the time in UTC. Otherwise, this family of functions is analogous to **mktime**, **_mktime32**, and **_mktime64**.

In Visual C++ 2005, **_mkgmtime** is an inline function that evaluates to **_mkgmtime64**, and **time_t** is equivalent to **__time64_t**. If you need to force the compiler to interpret **time_t** as the old 32-bit **time_t**, you can define **_USE_32BIT_TIME_T**. This is not recommended because your application might fail after January 19, 2038 (the maximum range of a 32-bit **time_t**), and it is not allowed at all on 64-bit platforms.

The time structure passed in will be changed as follows, in the same way as they are changed with the **_mktime** functions: the **tm_wday** and **tm_yday** fields are set to new values based on the values of **tm_mday** and **tm_year**. When specifying a **tm** structure time, set the **tm_isdst** field to:

- Zero (0) to indicate that standard time is in effect.
- A value greater than 0 to indicate that daylight saving time is in effect.
- A value less than zero to have the C run-time library code compute whether standard time or daylight saving time is in effect.

The C run-time library uses the TZ environment variable to determine the correct daylight savings time. If TZ is not set, the operating system is queried to get the correct regional daylight savings time behavior. **tm_isdst** is a required field. If not set, its value is undefined and the return value from **mktime** is unpredictable.

The range of the **_mkgmtime32** function is from midnight, January 1, 1970, UTC to January 19, 3:14:07, 2038, UTC. The range of **_mkgmtime64** is from midnight, January 1, 1970, UTC to 23:59:59, December 31, 3000, UTC. An out-of-range date results in a return value of –1. The range of **_mkgmtime** depends on whether **_USE_32BIT_TIME_T** is defined. If not defined (the default) the range is that of **_mkgmtime64**; otherwise, the range is limited to the 32-bit range of **_mkgmtime32**.

The time passed in is interpreted as a local time and is adjusted to UTC based on the **TZ** environment variable. If the local time is close to the range limits, it might lie outside the range limit when converted to UTC. If this occurs, the return value is 1.

Note that **gmtime** and **localtime** use a single statically allocated buffer for the conversion. If you supply this buffer to **mkgmtime**, the previous contents are destroyed.

Example

```
// crt_mkgmtime.c
#include <stdio.h>
#include <time.h>
```

```
int main()
{
    struct tm t1;
    time_t now, mytime, gmtime;
    char buff[30];

    time( & now );

    _localtime64_s( &t1, &now );

    mytime = mktime(&t1);
    gmtime = _mkgmtime(&t1);

    printf("Seconds since midnight, January 1, 1970\n");
    printf("My time: %I64d\nGM time (UTC): %I64d\n\n", mytime, gmtime);

    /* Use asctime_s to display these times. */

    _localtime64_s( &t1, &mytime );
    asctime_s( buff, sizeof(buff), &t1 );
    printf( "Local Time: %s\n", buff );

    _localtime64_s( &t1, &gmtime );
    asctime_s( buff, sizeof(buff), &t1 );
    printf( "Greenwich Mean Time: %s\n", buff );

}
```

Sample Output

```
Seconds since midnight, January 1, 1970
My time: 1051302521
GM time (UTC): 1051277321

Local Time: Fri Apr 25 13:28:41 2003

Greenwich Mean Time: Fri Apr 25 06:28:41 2003
```

The following example shows how the incomplete structure is filled out with the computed values of the day of the week and the day of the year.

```
// crt_mkgmtime2.c
#include <stdio.h>
#include <time.h>
#include <memory.h>

int main()
{
    struct tm t1, t2;
    time_t gmtime;
    char buff[30];

    memset(&t1, 0, sizeof(struct tm));
    memset(&t2, 0, sizeof(struct tm));

    t1.tm_mon = 1;
    t1.tm_isdst = 0;
    t1.tm_year = 103;
    t1.tm_mday = 12;

    // The day of the week and year will be incorrect in the output here.
    asctime_s( buff, sizeof(buff), &t1);
    printf("Before calling _mkgmtime, t1 = %s t.tm_yday = %d\n",
            buff, t1.tm_yday );

    gmtime = _mkgmtime(&t1);
```

```
    // The correct day of the week and year were determined.
    asctime_s( buff, sizeof(buff), &t1);
    printf("After calling _mkgmtime, t1 = %s t.tm_yday = %d\n",
            buff, t1.tm_yday );

  }
```

Output

```
Before calling _mkgmtime, t1 = Sun Feb 12 00:00:00 2003
 t.tm_yday = 0
After calling _mkgmtime, t1 = Wed Feb 12 00:00:00 2003
 t.tm_yday = 42
```

See Also

Reference

Time Management
asctime, _wasctime
asctime_s, _wasctime_s
gmtime, _gmtime32, _gmtime64
_gmtime_s, _gmtime32_s, _gmtime64_s
localtime_s, _localtime32_s, _localtime64_s
time, _time32, _time64

mktemp

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _mktemp or security-enhanced _mktemp_s instead.

_mktemp, _wmktemp

Creates a unique file name. These functions are deprecated because more secure versions are available; see _mktemp_s, _wmktemp_s.

```
char *_mktemp(
   char *template
);
wchar_t *_wmktemp(
   wchar_t *template
);
template <size_t size>
char *_mktemp(
   char (&template)[size]
); // C++ only
template <size_t size>
wchar_t *_wmktemp(
   wchar_t (&template)[size]
); // C++ only
```

Parameters

template

File name pattern.

Return Value

Each of these functions returns a pointer to the modified template. The function returns **NULL** if *template* is badly formed or no more unique names can be created from the given template.

Remarks

The **_mktemp** function creates a unique file name by modifying the *template* argument. **_mktemp** automatically handles multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the multibyte code page currently in use by the run-time system. **_wmktemp** is a wide-character version of **_mktemp**; the argument and return value of **_wmktemp** are wide-character strings. **_wmktemp** and **_mktemp** behave identically otherwise, except that **_wmktemp** does not handle multibyte-character strings.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tmktemp	**_mktemp**	**_mktemp**	**_wmktemp**

The *template* argument has the form *base*XXXXXX, where *base* is the part of the new file name that you supply and each X is a placeholder for a character supplied by **_mktemp**. Each placeholder character in *template* must be an uppercase X. **_mktemp** preserves *base* and replaces the first trailing X with an alphabetic character. **_mktemp** replaces the following trailing X's with a five-digit value; this value is a unique number identifying the calling process, or in multithreaded programs, the calling thread.

Each successful call to **_mktemp** modifies *template*. In each subsequent call from the same process or thread with the same *template* argument, **_mktemp** checks for file names that match names returned by **_mktemp** in previous calls. If no file exists for a given name, **_mktemp** returns that name. If files exist for all previously returned names, **_mktemp** creates a new name by replacing the alphabetic character it used in the previously returned name with the next available lowercase letter, in order, from 'a' through 'z'. For example, if *base* is:

```
fn
```

and the five-digit value supplied by **_mktemp** is 12345, the first name returned is:

```
fna12345
```

If this name is used to create file FNA12345 and this file still exists, the next name returned on a call from the same process or

thread with the same *base* for *template* is:

```
fnb12345
```

If FNA12345 does not exist, the next name returned is again:

```
fna12345
```

_mktemp can create a maximum of 26 unique file names for any given combination of base and template values. Therefore, FNZ12345 is the last unique file name **_mktemp** can create for the *base* and *template* values used in this example.

On failure, **errno** is set. If *template* has an invalid format (for example, fewer than 6 X's), **errno** is set to **EINVAL**. If **_mktemp** is unable to create a unique name because all 26 possible file names already exist, **_mktemp** sets template to an empty string and returns **EEXIST**.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Requirements

Routine	Required header	Compatibility
_mktemp	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wmktemp	<io.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_mktemp.c
// compile with: /W3
/* The program uses _mktemp to create
 * unique filenames. It opens each filename
 * to ensure that the next name is unique.
 */

#include <io.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>

char *template = "fnXXXXXX";
char *result;
char names[27][9];

int main( void )
{
   int i;
   FILE *fp;

   for( i = 0; i < 27; i++ )
   {
      strcpy_s( names[i], sizeof( names[i] ), template );
      /* Attempt to find a unique filename: */
      result = _mktemp( names[i] );  // C4996
      // Note: _mktemp is deprecated; consider using _mktemp_s instead
      if( result == NULL )
      {
         printf( "Problem creating the template\n" );
         if (errno == EINVAL)
         {
             printf( "Bad parameter\n");
         }
```

```
            else if (errno == EEXIST)
            {
                printf( "Out of unique filenames\n");
            }
        }
        else
        {
            fopen_s( &fp, result, "w" );
            if( fp != NULL )
               printf( "Unique filename is %s\n", result );
            else
               printf( "Cannot open %s\n", result );
            fclose( fp );
        }
    }
}
```

Sample Output

```
Unique filename is fna03912
Unique filename is fnb03912
Unique filename is fnc03912
Unique filename is fnd03912
Unique filename is fne03912
Unique filename is fnf03912
Unique filename is fng03912
Unique filename is fnh03912
Unique filename is fni03912
Unique filename is fnj03912
Unique filename is fnk03912
Unique filename is fnl03912
Unique filename is fnm03912
Unique filename is fnn03912
Unique filename is fno03912
Unique filename is fnp03912
Unique filename is fnq03912
Unique filename is fnr03912
Unique filename is fns03912
Unique filename is fnt03912
Unique filename is fnu03912
Unique filename is fnv03912
Unique filename is fnw03912
Unique filename is fnx03912
Unique filename is fny03912
Unique filename is fnz03912
Problem creating the template.
Out of unique filenames.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

File Handling
fopen, _wfopen
_getmbcp
_getpid
_open, _wopen
_setmbcp
_tempnam, _wtempnam, tmpnam, _wtmpnam
tmpfile

_mktemp_s, _wmktemp_s

Creates a unique file name. These are versions of _mktemp, _wmktemp with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _mktemp_s(
   char *template,
   size_t sizeInChars
);
errno_t _wmktemp_s(
   wchar_t *template,
   size_t sizeInChars
);
template <size_t size>
errno_t _mktemp_s(
   char (&template)[size]
); // C++ only
template <size_t size>
errno_t _wmktemp_s(
   wchar_t (&template)[size]
); // C++ only
```

Parameters

template

File name pattern.

sizeInChars

Size of the buffer in single-byte characters in **_mktemp_s**; wide characters in **_wmktemp_s**, including the null terminator.

Return Value

Both of these functions return zero on success; an error code on failure.

Error Conditions

template	***sizeInChars***	**return value**	**new value in template**
NULL	any	**EINVAL**	**NULL**
Incorrect format (see **Remarks** section for correct format)	any	**EINVAL**	empty string
any	<= number of X's	**EINVAL**	empty string

If any of the above error conditions occurs, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the functions returns **EINVAL**.

Remarks

The **_mktemp_s** function creates a unique file name by modifying the *template* argument, so that after the call, the *template* pointer points to a string containing the new file name. **_mktemp_s** automatically handles multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the multibyte code page currently in use by the run-time system. **_wmktemp_s** is a wide-character version of **_mktemp_s**; the argument of **_wmktemp_s** is a wide-character string. **_wmktemp_s** and **_mktemp_s** behave identically otherwise, except that **_wmktemp_s** does not handle multibyte-character strings.

Generic-Text Routine Mappings

Tchar.h routine	**_UNICODE and _MBCS not defined**	**_MBCS defined**	**_UNICODE defined**
_tmktemp_s	**_mktemp_s**	**_mktemp_s**	**_wmktemp_s**

The *template* argument has the form *baseXXXXXX*, where *base* is the part of the new file name that you supply and each X is a placeholder for a character supplied by **_mktemp_s**. Each placeholder character in *template* must be an uppercase X.

_mktemp_s preserves *base* and replaces the first trailing X with an alphabetic character. **_mktemp_s** replaces the following trailing X's with a five-digit value; this value is a unique number identifying the calling process, or in multithreaded programs, the calling thread.

Each successful call to **_mktemp_s** modifies *template*. In each subsequent call from the same process or thread with the same *template* argument, **_mktemp_s** checks for file names that match names returned by **_mktemp_s** in previous calls. If no file exists for a given name, **_mktemp_s** returns that name. If files exist for all previously returned names, **_mktemp_s** creates a new name by replacing the alphabetic character it used in the previously returned name with the next available lowercase letter, in order, from 'a' through 'z'. For example, if *base* is:

```
fn
```

and the five-digit value supplied by **_mktemp_s** is 12345, the first name returned is:

```
fna12345
```

If this name is used to create file FNA12345 and this file still exists, the next name returned on a call from the same process or thread with the same *base* for *template* is:

```
fnb12345
```

If FNA12345 does not exist, the next name returned is again:

```
fna12345
```

_mktemp_s can create a maximum of 26 unique file names for any given combination of base and template values. Therefore, FNZ12345 is the last unique file name **_mktemp_s** can create for the *base* and *template* values used in this example.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Requirements

Routine	Required header	Compatibility
_mktemp_s	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wmktemp_s	<io.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_mktemp_s.c
/* The program uses _mktemp to create
 * five unique filenames. It opens each filename
 * to ensure that the next name is unique.
 */

#include <io.h>
#include <string.h>
#include <stdio.h>

char *template = "fnXXXXXX";
char names[5][9];

int main( void )
{
   int i, err, sizeInChars;
   FILE *fp;
```

```
    for( i = 0; i < 5; i++ )
    {
       strcpy_s( names[i], sizeof(names[i]), template );
       /* Get the size of the string and add one for the null terminator.*/
       sizeInChars = strlen(names[i]) + 1;
       /* Attempt to find a unique filename: */
       err = _mktemp_s( names[i], sizeInChars );
       if( err != 0 )
          printf( "Problem creating the template" );
       else
       {
          if( fopen_s( &fp, names[i], "w" ) == 0 )
             printf( "Unique filename is %s\n", names[i] );
          else
             printf( "Cannot open %s\n", names[i] );
          fclose( fp );
       }
    }
 }
```

Sample Output

```
Unique filename is fna03836
Unique filename is fnb03836
Unique filename is fnc03836
Unique filename is fnd03836
Unique filename is fne03836
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

File Handling
fopen, _wfopen
_getmbcp
_getpid
_open, _wopen
_setmbcp
_tempnam, _wtempnam, tmpnam, _wtmpnam
tmpfile_s

mktime, _mktime32, _mktime64

Convert the local time to a calendar value.

```
time_t mktime(
   struct tm *timeptr
);
__time32_t _mktime32(
   struct tm *timeptr
);
__time64_t _mktime64(
   struct tm *timeptr
);
```

Parameters

timeptr

Pointer to time structure; see asctime.

Return Value

_mktime32 returns the specified calendar time encoded as a value of type time_t. If *timeptr* references a date before midnight, January 1, 1970, or if the calendar time cannot be represented, **_mktime32** returns –1 cast to type **time_t**. When using **_mktime32** and if *timeptr* references a date after 03:14:07 January 19, 2038, Coordinated Universal Time (UTC), it will return –1 cast to type **time_t**.

_mktime64 will return –1 cast to type **__time64_t** if *timeptr* references a date after 23:59:59, December 31, 3000, UTC.

Remarks

The **mktime**, **_mktime32** and **_mktime64** functions convert the supplied time structure (possibly incomplete) pointed to by *timeptr* into a fully defined structure with normalized values and then converts it to a **time_t** calendar time value. The converted time has the same encoding as the values returned by the time function. The original values of the **tm_wday** and **tm_yday** components of the *timeptr* structure are ignored, and the original values of the other components are not restricted to their normal ranges.

mktime is an inline function that is equivalent to **_mktime64**, unless **_USE_32BIT_TIME_T** is defined, in which case it is equivalent to **_mktime32**.

After an adjustment to UTC, **_mktime32** handles dates from midnight, January 1, 1970, to 03:14:07, January 19, 2038. **_mktime64** handles dates from midnight, January 1, 1970 to 23:59:59, December 31, 3000. This adjustment may cause these functions to return -1 (cast to **time_t**, **__time32_t** or **__time64_t**) even though the date you specify is within range. For example, if you are in Cairo, Egypt, which is two hours ahead of UTC, two hours will first be subtracted from the date you specify in *timeptr*; this may now put your date out of range.

These functions may be used to validate and fill in a **tm** structure. If successful, these functions set the values of **tm_wday** and **tm_yday** as appropriate and set the other components to represent the specified calendar time, but with their values forced to the normal ranges. The final value of **tm_mday** is not set until **tm_mon** and **tm_year** are determined. When specifying a **tm** structure time, set the **tm_isdst** field to:

- Zero (0) to indicate that standard time is in effect.
- A value greater than 0 to indicate that daylight saving time is in effect.
- A value less than zero to have the C run-time library code compute whether standard time or daylight saving time is in effect.

The C run-time library will determine the daylight savings time behavior from the TZ environment variable. If **TZ** is not set, the Win32 API call GetTimeZoneInformation is used to get the daylight savings time information from the operating system. If this fails, the library assumes the United States' rules for implementing the calculation of daylight saving time are used. **tm_isdst** is a required field. If not set, its value is undefined and the return value from these functions is unpredictable. If *timeptr* points to a **tm** structure returned by a previous call to **asctime**, **gmtime**, or **localtime** (or variants of these functions), the **tm_isdst** field contains the correct value.

Note that **gmtime** and **localtime** (and **_gmtime32**, **_gmtime64**, **_localtime32**, and **_localtime64**) use a single buffer per

thread for the conversion. If you supply this buffer to **mktime**, **_mktime32** or **_mktime64**, the previous contents are destroyed.

These functions validate their parameter. If *timeptr* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

Requirements

Routine	Required header	Compatibility
mktime	<time.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mktime32	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mktime64	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_mktime.c
/* The example takes a number of days
 * as input and returns the time, the current
 * date, and the specified number of days.
 */

#include <time.h>
#include <stdio.h>

int main( void )
{
   struct tm  when;
   __time64_t now, result;
   int        days;
   char       buff[80];

   time( &now );
   _localtime64_s( &when, &now );
   asctime_s( buff, sizeof(buff), &when );
   printf( "Current time is %s\n", buff );
   days = 20;
   when.tm_mday = when.tm_mday + days;
   if( (result = mktime( &when )) != (time_t)-1 ) {
      asctime_s( buff, sizeof(buff), &when );
      printf( "In %d days the time will be %s\n", days, buff );
   } else
      perror( "mktime failed" );
}
```

Sample Output

```
Current time is Fri Apr 25 13:34:07 2003

In 20 days the time will be Thu May 15 13:34:07 2003
```

.NET Framework Equivalent

System::DateTime::DateTime

See Also

Reference

Time Management
asctime, _wasctime
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
time, _time32, _time64

modf, modff

Splits a floating-point value into fractional and integer parts.

```
double modf(
   double x,
   double *intptr
);
float modf(
   float x,
   float *intptr
);  // C++ only
long double modf(
   long double x,
   long double * intptr
);  // C++ only
float modff(
   float x,
   float *intptr
);
```

Parameters

x

Floating-point value.

intptr

Pointer to stored integer portion.

Return Value

This function returns the signed fractional portion of *x*. There is no error return.

Remarks

The **modf** function breaks down the floating-point value *x* into fractional and integer parts, each of which has the same sign as *x*. The signed fractional portion of *x* is returned. The integer portion is stored as a floating-point value at *intptr*.

modf has an implementation that uses Streaming SIMD Extensions 2 (SSE2). See _set_SSE2_enable for information and restrictions on using the SSE2 implementation.

C++ allows overloading, so you can call overloads of **modf**. In a C program, **modf** always takes two double values and returns a double value.

Requirements

Routine	Required header	Compatibility
modf, **modff**	<math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_modf.c

#include <math.h>
#include <stdio.h>
```

```
int main( void )
{
   double x, y, n;

   x = -14.87654321;      /* Divide x into its fractional */
   y = modf( x, &n );     /* and integer parts            */

   printf( "For %f, the fraction is %f and the integer is %.f\n",
           x, y, n );
}
```

Output

```
For -14.876543, the fraction is -0.876543 and the integer is -14
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support
Long Double
frexp
ldexp

_msize

Returns the size of a memory block allocated in the heap.

```
size_t _msize(
   void *memblock
);
```

Parameters

memblock

Pointer to the memory block.

Return Value

_msize returns the size (in bytes) as an unsigned integer.

Remarks

The **_msize** function returns the size, in bytes, of the memory block allocated by a call to **calloc**, **malloc**, or **realloc**.

When the application is linked with a debug version of the C run-time libraries, **_msize** resolves to _msize_dbg. For more information about how the heap is managed during the debugging process, see The CRT Debug Heap.

This function validates its parameter. If *memblock* is a null pointer, **_msize** invokes an invalid parameter handler, as described in Parameter Validation. If the error is handled, the function sets **errno** to **EINVAL** and returns -1.

Requirements

Routine	Required header	Compatibility
_msize	<malloc.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

See the example for realloc.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation
calloc
_expand
malloc
realloc

_msize_dbg

Calculates the size of a block of memory in the heap (debug version only).

```
size_t _msize_dbg(
   void *userData,
   int blockType
);
```

Parameters

userData

Pointer to the memory block for which to determine the size.

blockType

Type of the specified memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

Return Value

On successful completion, **_msize_dbg** returns the size (in bytes) of the specified memory block; otherwise it returns NULL.

Remarks

_msize_dbg is a debug version of the _msize function. When _DEBUG is not defined, each call to **_msize_dbg** is reduced to a call to **_msize**. Both **_msize** and **_msize_dbg** calculate the size of a memory block in the base heap, but **_msize_dbg** adds two debugging features: It includes the buffers on either side of the user portion of the memory block in the returned size and it allows size calculations for specific block types.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see Memory Management and the Debug Heap. For information about the allocation block types and how they are used, see Types of Blocks on the Debug Heap. For information about the differences between calling a standard heap function and its debug version in a debug build of an application, see Using the Debug Version Versus the Base Version.

This function validates its parameter. If *memblock* is a null pointer, **_msize** invokes an invalid parameter handler, as described in Parameter Validation. If the error is handled, the function sets **errno** to **EINVAL** and returns -1.

Requirements

Routine	Required header	Compatibility
_msize_dbg	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

```
// crt_msize_dbg.c
/*
 * This program allocates a block of memory using _malloc_dbg
 * and then calls _msize_dbg to display the size of that block.
 * Next, it uses _realloc_dbg to expand the amount of
 * memory used by the buffer and then calls _msize_dbg again to
 * display the new amount of memory allocated to the buffer.
 */

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
```

```
#include <crtdbg.h>

int main( void )
{
        long *buffer, *newbuffer;
        size_t size;

        /*
         * Call _malloc_dbg to include the filename and line number
         * of our allocation request in the header
         */
        buffer = (long *)_malloc_dbg( 40 * sizeof(long), _NORMAL_BLOCK, __FILE__, __LINE__
);
        if( buffer == NULL )
               exit( 1 );

        /*
         * Get the size of the buffer by calling _msize_dbg
         */
        size = _msize_dbg( buffer, _NORMAL_BLOCK );
        printf( "Size of block after _malloc_dbg of 40 longs: %u\n", size );

        /*
         * Reallocate the buffer using _realloc_dbg and show the new size
         */
        newbuffer = _realloc_dbg( buffer, size + (40 * sizeof(long)), _NORMAL_BLOCK, __FILE
__, __LINE__ );
        if( newbuffer == NULL )
               exit( 1 );
        buffer = newbuffer;
        size = _msize_dbg( buffer, _NORMAL_BLOCK );
        printf( "Size of block after _realloc_dbg of 40 more longs: %u\n", size );

        free( buffer );
        exit( 0 );
}
```

Output

```
Size of block after _malloc_dbg of 40 longs: 160
Size of block after _realloc_dbg of 40 more longs: 320
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_malloc_dbg

_nextafter

Returns the next representable neighbor.

```
double _nextafter(
   double x,
   double y
);
```

Parameters

x, *y*

Double-precision floating-point values.

Return Value

If *x*=*y*, **_nextafter** returns *x*, with no exception triggered. If either *x* or *y* is a quiet NaN, then the return value is one or the other of the input NaNs.

Remarks

The **_nextafter** function returns the closest representable neighbor of *x* in the direction toward *y*.

Requirements

Routine	Required header	Compatibility
_nextafter	<float.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support

_isnan

offsetof

Retrieves the offset of a member from the beginning of its parent structure.

```
size_t offsetof(
      structName,
      memberName
);
```

Parameters

structName

Name of the parent data structure.

memberName

Name of the member in the parent data structure for which to determine the offset.

Return Value

offsetof returns the offset in bytes of the specified member from the beginning of its parent data structure. It is undefined for bit fields.

Remarks

The **offsetof** macro returns the offset in bytes of *memberName* from the beginning of the structure specified by *structName*. You can specify types with the **struct** keyword.

Note
offsetof is not a function and cannot be described using a C prototype.

Requirements

Routine	**Required header**	**Compatibility**
offsetof	<stddef.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

See Also

Reference

Memory Allocation

_onexit, _onexit_m

Registers a routine to be called at exit time.

```
_onexit_t _onexit(
   _onexit_t function
);
_onexit_t_m _onexit_m(
   _onexit_t_m function
);
```

Parameters

function

Pointer to a function to be called at exit.

Return Value

_onexit returns a pointer to the function if successful or **NULL** if there is no space to store the function pointer.

Remarks

The **_onexit** function is passed the address of a function (*function*) to be called when the program terminates normally. Successive calls to **_onexit** create a register of functions that are executed in LIFO (last-in-first-out) order. The functions passed to **_onexit** cannot take parameters.

In the case when **_onexit** is called from within a DLL, routines registered with **_onexit** run on a DLL's unloading after **DllMain** is called with DLL_PROCESS_DETACH.

_onexit is a Microsoft extension. For ANSI portability, use atexit. The **_onexit_m** version of the function is for mixed mode use.

Requirements

Routine	Required header	Compatibility
_onexit	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_onexit.c

#include <stdlib.h>
#include <stdio.h>

/* Prototypes */
int fn1(void), fn2(void), fn3(void), fn4 (void);

int main( void )
{
   _onexit( fn1 );
   _onexit( fn2 );
   _onexit( fn3 );
   _onexit( fn4 );
   printf( "This is executed first.\n" );
}

int fn1()
{
   printf( "next.\n" );
   return 0;
```

```
}

int fn2()
{
   printf( "executed " );
   return 0;
}

int fn3()
{
   printf( "is " );
   return 0;
}

int fn4()
{
   printf( "This " );
   return 0;
}
```

Output

```
This is executed first.
This is executed next.
```

.NET Framework Equivalent

System::Diagnostics::Process::Exited

See Also

Reference

Process and Environment Control
atexit
exit, _exit

open

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _open instead.

_open, _wopen

Opens a file. These functions are deprecated because more secure versions are available; see _sopen_s, _wsopen_s.

```
int _open(
   const char *filename,
   int oflag [,
   int pmode]
);
int _wopen(
   const wchar_t *filename,
   int oflag [,
   int pmode]
);
```

Parameters

filename

File name.

oflag

Type of operations allowed.

pmode

Permission mode.

Return Value

Each of these functions returns a file descriptor for the opened file. A return value of -1 indicates an error, in which case **errno** is set to one of the following values.

EACCES

Tried to open read-only file for writing, file's sharing mode does not allow specified operations, or given path is directory.

EEXIST

_O_CREAT and **_O_EXCL** flags specified, but *filename* already exists.

EINVAL

Invalid *oflag* or *pmode* argument.

EMFILE

No more file descriptors available (too many open files).

ENOENT

File or path not found.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_open** function opens the file specified by *filename* and prepares the file for reading or writing, as specified by *oflag*. **_wopen** is a wide-character version of **_open**; the *filename* argument to **_wopen** is a wide-character string. **_wopen** and **_open** behave identically otherwise.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_topen	**_open**	**_open**	**_wopen**

oflag is an integer expression formed from one or more of the following manifest constants or constant combinations defined in Fcntl.h.

_O_APPEND

Moves file pointer to end of file before every write operation.

_O_BINARY

Opens file in binary (untranslated) mode. (See fopen for a description of binary mode.)

_O_CREAT

Creates and opens a new file for writing. Has no effect if the file specified by *filename* exists. *pmode* argument is required when **_O_CREAT** is specified.

_O_CREAT| _O_SHORT_LIVED

Create a file as temporary and if possible do not flush to disk. *pmode* argument is required when **_O_CREAT** is specified.

_O_CREAT| _O_TEMPORARY

Create a file as temporary; the file is deleted when the last file descriptor is closed. *pmode* argument is required when **_O_CREAT** is specified.

_O_CREAT| _O_EXCL

Returns an error value if the file specified by *filename* exists. Applies only when used with **_O_CREAT**.

_O_NOINHERIT

Prevents creation of a shared file descriptor.

_O_RANDOM

Specifies that caching is optimized for, but not restricted to, random access from disk.

_O_RDONLY

Opens a file for reading only; cannot be specified with **_O_RDWR** or **_O_WRONLY**.

_O_RDWR

Opens file for both reading and writing; you cannot specify this flag with **_O_RDONLY** or **_O_WRONLY**.

_O_SEQUENTIAL

Specifies that caching is optimized for, but not restricted to, sequential access from disk.

_O_TEXT

Opens a file in text (translated) mode. (For more information, see Text and Binary Mode File I/O and fopen.)

_O_TRUNC

Opens a file and truncates it to zero length; the file must have write permission. You cannot specify this flag with **_O_RDONLY**. **_O_TRUNC** used with **_O_CREAT** opens an existing file or creates a new file.

Note
The **_O_TRUNC** flag destroys the contents of the specified file.

_O_WRONLY

Opens the file for writing only; cannot be specified with **_O_RDONLY** or **_O_RDWR**.

_O_U16TEXT

Open the file in Unicode UTF-16 mode. This option is available in Visual C++ 2005.

_O_U8TEXT

Open the file in Unicode UTF-8 mode. This option is available in Visual C++ 2005.

_O_WTEXT

Open the file in Unicode mode. This option is available in Visual C++ 2005.

To specify the file access mode, you must specify either **_O_RDONLY**, **_O_RDWR**, or **_O_WRONLY**. There is no default value

for the access mode.

If **_O_WTEXT** is used to open a file for reading, **_open** reads the beginning of the file and check for a byte order mark (BOM). If there is a BOM, the file is treated as UTF-8 or UTF-16LE depending on the BOM. If no BOM is present, the file is treated as ANSI. When a file is opened for writing using **_O_WTEXT**, UTF-16 is used. If **_O_UTF8** is used, the file is always opened as UTF-8 and if **_O_UTF16** is used, the file is always opened as UTF-16 regardless of any previous setting or byte order mark.

If **_open** is called with **_O_WRONLY|_O_APPEND** (append mode) and **_O_WTEXT**, **_O_U16TEXT**, or **_O_U8TEXT**, it will first try to open the file for reading and writing, read the BOM, then reopen it for writing only. If opening the file for reading and writing fails, it will open the file for writing only and use the default value for the Unicode mode setting.

When two or more manifest constants are used to form the *oflag* argument, the constants are combined with the bitwise-OR operator (**|**). For a discussion of binary and text modes, see Text and Binary Mode File I/O.

The *pmode* argument is required only when **_O_CREAT** is specified. If the file already exists, *pmode* is ignored. Otherwise, *pmode* specifies the file permission settings, which are set when the new file is closed the first time. **_open** applies the current file-permission mask to *pmode* before setting the permissions (for more information, see _umask). *pmode* is an integer expression containing one or both of the following manifest constants, defined in SYS\Stat.h.

_S_IREAD

Reading only permitted.

_S_IWRITE

Writing permitted (effectively permits reading and writing).

_S_IREAD | _S_IWRITE

Reading and writing permitted.

When both constants are given, they are joined with the bitwise-OR operator (**|**). In Windows NT, all files are readable, so write-only permission is not available; thus the modes **_S_IWRITE** and **_S_IREAD | _S_IWRITE** are equivalent.

If a value other than the above is specified for *pmode* (even if it would specify a valid *pmode* in another operating system) or any value other than the allowed *oflag* values is specified, the function generates an assertion in Debug mode and invokes the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, the function returns -1 and sets **errno** to **EINVAL**.

Requirements

Routine	Required header	Optional header	Compatibility
_open	<io.h>	<fcntl.h>, <sys/types.h>, <sys/stat.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wopen	<io.h> or <wchar.h>	<fcntl.h>, <sys/types.h>, <sys/stat.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_open.c
// compile with: /W3
/* This program uses _open to open a file
 * named CRT_OPEN.C for input and a file named CRT_OPEN.OUT
 * for output. The files are then closed.
 */
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
```

```
#include <stdio.h>

int main( void )
{
   int fh1, fh2;

   fh1 = _open( "CRT_OPEN.C", _O_RDONLY ); // C4996
   // Note: _open is deprecated; consider using _sopen_s instead
   if( fh1 == -1 )
      perror( "Open failed on input file" );
   else
   {
      printf( "Open succeeded on input file\n" );
      _close( fh1 );
   }

   fh2 = _open( "CRT_OPEN.OUT", _O_WRONLY | _O_CREAT, _S_IREAD |
                           _S_IWRITE ); // C4996
   if( fh2 == -1 )
      perror( "Open failed on output file" );
   else
   {
      printf( "Open succeeded on output file\n" );
      _close( fh2 );
   }
}
```

Output

```
Open succeeded on input file
Open succeeded on output file
```

.NET Framework Equivalent

- System::IO::File::Open
- System::IO::FileStream::FileStream

See Also

Reference

Low-Level I/O
_chmod, _wchmod
_close
_creat, _wcreat
_dup, _dup2
fopen, _wfopen
_sopen, _wsopen

_open_osfhandle

Associates a C run-time file descriptor with an existing operating-system file handle.

```
int _open_osfhandle (
   intptr_t osfhandle,
      int flags
);
```

Parameters

osfhandle

Operating-system file handle.

flags

Types of operations allowed.

Return Value

If successful, **_open_osfhandle** returns a C run-time file descriptor. Otherwise, it returns –1.

Remarks

The **_open_osfhandle** function allocates a C run-time file descriptor and associates it with the operating-system file handle specified by *osfhandle*. The *flags* argument is an integer expression formed from one or more of the manifest constants defined in Fcntl.h. When two or more manifest constants are used to form the *flags* argument, the constants are combined with the bitwise-OR operator (**|**).

Fcntl.h defines the following manifest constants.

_O_APPEND

Positions a file pointer to the end of the file before every write operation.

_O_RDONLY

Opens the file for reading only.

_O_TEXT

Opens the file in text (translated) mode.

To close a file opened with **_open_osfhandle**, call **_close**. The underlying handle is also closed by a call to **_close**, so it is not necessary to call the Win32 function **CloseHandle** on the original handle.

Requirements

Routine	Required header	Compatibility
_open_osfhandle	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

System::IO::FileStream::Handle

See Also

Reference

File Handling

operator new[] (CRT)

Allocate block of memory from heap

```
void *__cdecl operator new[](
   size_t count
);
void *__cdecl operator new[] (
   size_t count,
   void * object
) throw();
void *__cdecl operator new[] (
   size_t count,
   const std::nothrow_t&
) throw();
```

Parameters

count

The size of the allocation.

object

A pointer to a block of memory in which the object will be created.

Return Value

A pointer to the lowest byte address of the newly allocated storage.

Remarks

This form of **operator new** is known as vector new, in contrast to the scalar new form (operator new).

The first form of this operator is known as the nonplacement form. The second form of this operator is known as the placement form and the third form of this operator is the nonthrowing placement form.

The first form of the operator is defined by the compiler and does not require new.h to be included in your program.

operator delete[] frees memory allocated with operator new.

You can configure whether **operator new[]** returns null or throws an exception on failure. See The new and delete Operators for more information.

With the exception of throwing or no-throwing behavior, the CRT **operator new** behaves like operator new[] in the Standard C++ Library.

Requirements

Routine	Required header	Compatibility
new[]	<new.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

The following shows how to use the vector, nonplacement form of **operator new**.

```
// crt_new4.cpp
#include <stdio.h>
```

```
int main() {
   int * k = new int[10];
   k[0] = 21;
   printf("%d\n", k[0]);
   delete [] k;
}
```

The following shows how to use the vector, placement form of **operator new**.

```
// crt_new5.cpp
#include <stdio.h>
#include <new.h>
int main() {
   int * i = new int[10];
   i[0] = 21;
   printf("%d\n", i[0]);
   // initialize existing memory (i) with, in this case, int[[10]
   int * j = new(i) int[10];   // placement vector new
   printf("%d\n", j[0]);
   j[0] = 22;
   printf("%d\n", i[0]);
   delete [] i;   // or, could have deleted [] j
}
```

The following shows how to use the vector, placement, no-throw form of **operator new**.

```
// crt_new6.cpp
#include <stdio.h>
#include <new.h>
int main() {
   int * k = new(std::nothrow) int[10];
   k[0] = 21;
   printf("%d\n", k[0]);
   delete [] k;
}
```

See Also

Reference

Memory Allocation

operator new (CRT)

Allocates block of memory from heap

```
void *__cdecl operator new(
   size_t count
);
void *__cdecl operator new(
   size_t count,
   void * object
) throw();
void *__cdecl operator new(
   size_t count,
   const std::nothrow_t&
) throw();
```

Parameters

count

The size of the allocation.

object

A pointer to a block of memory in which the object will be created.

Return Value

A pointer to the lowest byte address of the newly allocated storage.

Remarks

This form of **operator new** is known as scalar new, in contrast to the vector new form (operator new[]).

The first form of this operator is known as the nonplacement form. The second form of this operator is known as the placement form and the third form of this operator is the nonthrowing, placement form.

The first form of the operator is defined by the compiler and does not require new.h to be included in your program.

operator delete frees memory allocated with **operator new**.

You can configure whether operator new returns null or throws an exception on failure. See The new and delete Operators for more information.

With the exception of throwing or no-throwing behavior, the CRT **operator new** behaves like operator new in the Standard C++ Library.

Requirements

Routine	Required header	Compatibility
new	<new.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

The following shows how to use the scalar, nonplacement form of **operator new**.

```
// crt_new1.cpp
#include <stdio.h>
```

```
int main() {
   int * i = new int(6);
   printf("%d\n", *i);
   delete i;
}
```

The following shows how to use the scalar, placement form of **operator new**.

```
// crt_new2.cpp
#include <stdio.h>
#include <new.h>
int main() {
   int * i = new int(12);
   printf("*i = %d\n", *i);
   // initialize existing memory (i) with, in this case, int(7)
   int * j = new(i) int(7);   // placement new
   printf("*j = %d\n", *j);
   printf("*i = %d\n", *i);
   delete i;   // or, could have deleted j
}
```

The following shows how to use the scalar, placement, no-throw form of **operator new**.

```
// crt_new3.cpp
#include <stdio.h>
#include <new.h>
int main() {
   // allocates memory, initialize (8) and if call fails, new returns null
   int * k = new(std::nothrow) int(8);   // placement new
   printf("%d\n", *k);
   delete k;
}
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation

operator delete[] (CRT)

Frees allocated block.

```
void __cdecl operator delete[](
   void * object
);
void __cdecl operator delete[](
   void * object,
   void * memory
) throw();
void __cdecl operator delete[](
   void * object,
   const std::nothrow_t&
) throw();
```

Parameters

memory

The memory location being freed.

object

A pointer to the object being deleted.

Remarks

This form of **operator delete** is known as vector delete, in contrast to the scalar delete form (operator delete).

operator delete[] frees memory allocated by operator new[].

The first form of this operator is known as the nonplacement form. The second and third forms of this operator will commonly not be called from code but exist to give the compiler a matching delete to call when a placement new fails.

The first form of the operator is defined by the compiler and does not require new.h to be included in your program.

With the exception of throwing or no-throwing behavior, the CRT **operator delete[]** behaves like operator delete[] in the Standard C++ Library.

Requirements

Routine	Required header	Compatibility
delete[]	<new.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

See operator new[] for examples of using operator **delete**.

See Also

Reference

Memory Allocation

operator delete (CRT)

Frees allocated block.

```
void __cdecl operator delete(
   void * object
);
void __cdecl operator delete(
   void * object,
   void * memory
) throw();
void __cdecl operator delete(
   void * object,
   const std::nothrow_t&
) throw();
```

Parameters

memory

The memory location being freed.

object

A pointer to the object being deleted.

Remarks

This form of **operator delete** is known as scalar delete, in contrast to the vector delete form (operator delete[]).

operator delete frees memory allocated by operator new.

The first form of this operator is known as the nonplacement form. The second and third forms of this operator will commonly not be called from code but exist to give the compiler a matching delete to call when a placement new fails.

The first form of the operator is defined by the compiler and does not require new.h to be included in your program.

With the exception of throwing or no-throwing behavior, the CRT **operator delete** behaves like operator delete in the Standard C++ Library.

Requirements

Routine	Required header	Compatibility
delete	<new.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

See operator new for examples of using operator **delete**.

See Also

Reference

Memory Allocation

outp, outpw

These POSIX functions are deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _outp, _outpw, _outpd instead.

_outp, _outpw, _outpd

Outputs a byte (**_outp**), a word (**_outpw**), or a double word (**_outpd**) at a port.

```
int _outp(
   unsigned short port,
   int databyte
);
unsigned short _outpw(
   unsigned short port,
   unsigned short dataword
);
unsigned long _outpd(
   unsigned short port,
   unsigned long dataword
);
```

Parameters

port

Port number.

databyte, dataword

Output values.

Return Value

The functions return the data output. There is no error return.

Remarks

The **_outp**, **_outpw**, and **_outpd** functions write a byte, a word, and a double word, respectively, to the specified output port. The *port* argument can be any unsigned integer in the range 0 – 65,535; *databyte* can be any integer in the range 0 – 255; and *dataword* can be any value in the range of an integer, an unsigned short integer, and an unsigned long integer, respectively.

Because these functions write directly to an I/O port, they cannot be used in user code in Windows NT, Windows 2000, Windows XP, and Windows Server 2003. For information about using I/O ports in these operating systems, use the Win32 Communications API.

Requirements

Routine	Required header	Compatibility
_outp	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition
_outpw	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition
_outpd	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

See Also

Reference

Console and Port I/O

_inp, _inpw, _inpd

_pclose

Waits for a new command processor and closes the stream on the associated pipe.

```
int _pclose(
   FILE *stream
);
```

Parameters

stream

Return value from the previous call to **_popen**.

Return Value

Returns the exit status of the terminating command processor, or –1 if an error occurs. The format of the return value is the same as that for **_cwait**, except the low-order and high-order bytes are swapped. If stream is **NULL**, **_pclose** sets **errno** to **EINVAL** and returns -1.

For information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_pclose** function looks up the process ID of the command processor (Cmd.exe) started by the associated **_popen** call, executes a _cwait call on the new command processor, and closes the stream on the associated pipe.

Requirements

Routine	Required header	Compatibility
_pclose	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Process and Environment Control
_pipe
_popen, _wpopen

perror, _wperror

Print an error message.

```
void perror(
   const char *string
);
void _wperror(
   const wchar_t *string
);
```

Parameters

string

String message to print.

Remarks

The **perror** function prints an error message to **stderr**. **_wperror** is a wide-character version of **_perror**; the *string* argument to **_wperror** is a wide-character string. **_wperror** and **_perror** behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tperror	**perror**	**perror**	**_wperror**

string is printed first, followed by a colon, then by the system error message for the last library call that produced the error, and finally by a newline character. If *string* is a null pointer or a pointer to a null string, **perror** prints only the system error message.

The error number is stored in the variable errno (defined in ERRNO.H). The system error messages are accessed through the variable _sys_errlist, which is an array of messages ordered by error number. **perror** prints the appropriate error message using the **errno** value as an index to **_sys_errlist**. The value of the variable _sys_nerr is defined as the maximum number of elements in the **_sys_errlist** array.

For accurate results, call **perror** immediately after a library routine returns with an error. Otherwise, subsequent calls can overwrite the **errno** value.

In the Windows operating system, some **errno** values listed in ERRNO.H are unused. These values are reserved for use by the UNIX operating system. See _doserrno, errno, _sys_errlist, and _sys_nerr for a listing of **errno** values used by the Windows operating system. **perror** prints an empty string for any **errno** value not used by these platforms.

Note
In the Visual C++ implementation of the C run-time library, **perror** does not write to the same stream as **stderr**.

Requirements

Routine	Required header	Compatibility
perror	<stdio.h> or <stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wperror	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_perror.c
// compile with: /W3
/* This program attempts to open a file named
 * NOSUCHF.ILE. Because this file probably doesn't exist,
 * an error message is displayed. The same message is
 * created using perror, strerror, and _strerror.
 */

#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <share.h>

int main( void )
{
   int  fh;

   if( _sopen_s( &fh, "NOSUCHF.ILE", _O_RDONLY, _SH_DENYNO, 0 ) != 0 )
   {
      /* Three ways to create error message: */
      perror( "perror says open failed" );
      printf( "strerror says open failed: %s\n",
         strerror( errno ) ); // C4996
      printf( _strerror( "_strerror says open failed" ) ); // C4996
      // Note: strerror and _strerror are deprecated; consider
      // using strerror_s and _strerror_s instead.
   }
   else
   {
      printf( "open succeeded on input file\n" );
      _close( fh );
   }
}
```

Output

```
perror says open failed: No such file or directory
strerror says open failed: No such file or directory
_strerror says open failed: No such file or directory
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Process and Environment Control

clearerr

ferror

strerror, _strerror, _wcserror, __wcserror

_pipe

Creates a pipe for reading and writing.

```
int _pipe(
   int *pfds,
   unsigned int psize,
   int textmode
);
```

Parameters

pfds[2]

Array to hold read and write file descriptors.

psize

Amount of memory to reserve.

textmode

File mode.

Return Value

Returns 0 if successful. Returns –1 to indicate an error, in which case **errno** is set to one of three values: **EMFILE**, which indicates no more file descriptors available; **ENFILE**, which indicates a system file table overflow; or **EINVAL**, which indicates that either the array *pfds* was a null pointer or an invalid value for *textmode* was passed in.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_pipe** function creates a pipe. A *pipe* is an artificial I/O channel that a program uses to pass information to other programs. A pipe is similar to a file in that it has a file pointer, a file descriptor, or both and can be read from or written to using the standard library's input and output functions. However, a pipe does not represent a specific file or device. Instead, it represents temporary storage in memory that is independent of the program's own memory and is controlled entirely by the operating system.

_pipe is similar to **_open** but opens the pipe for reading and writing, returning two file descriptors instead of one. The program can use both sides of the pipe or close the one it does not need. For example, the command processor in Windows NT creates a pipe when executing a command such as:

```
PROGRAM1 | PROGRAM2
```

The standard output descriptor of `PROGRAM1` is attached to the pipe's write descriptor. The standard input descriptor of `PROGRAM2` is attached to the pipe's read descriptor. This eliminates the need for creating temporary files to pass information to other programs.

The **_pipe** function returns two file descriptors to the pipe in the *pfds* argument. The element *pfds*[0] contains the read descriptor and the element *pfds*[1] contains the write descriptor. Pipe file descriptors are used in the same way as other file descriptors. (The low-level input and output functions **_read** and **_write** can read from and write to a pipe.) To detect the end-of-pipe condition, check for a **_read** request that returns 0 as the number of bytes read.

The *psize* argument specifies the amount of memory, in bytes, to reserve for the pipe. The *textmode* argument specifies the translation mode for the pipe. The manifest constant **_O_TEXT** specifies a text translation, and the constant **_O_BINARY** specifies binary translation. (See fopen for a description of text and binary modes.) If the *textmode* argument is 0, **_pipe** uses the default translation mode specified by the default-mode variable _fmode.

In multithreaded programs, no locking is performed. The file descriptors returned are newly opened and should not be referenced by any thread until after the **_pipe** call is complete.

To use the **_pipe** function to communicate between a parent and a child process, each process must have only one descriptor open on the pipe. The descriptors must be opposites: if the parent has a read descriptor open, then the child must have a write descriptor open. The easiest way to do this is to **OR** (**|**) the **_O_NOINHERIT** flag with *textmode*. Then, use **_dup** or **_dup2** to

create an inheritable copy of the pipe descriptor you want to pass to the child. Close the original descriptor, and spawn the child process. On returning from the spawn call, close the duplicate descriptor in the parent process. For more information, see example 2 below.

In the Windows operating system, a pipe is destroyed when all its descriptors have been closed. (If all read descriptors on the pipe have been closed, writing to the pipe causes an error.) All read and write operations on the pipe wait until there is enough data or enough buffer space to complete the I/O request.

Requirements

Routine	Required header	Optional header	Compatibility
_pipe	<io.h>	<fcntl.h>, 1 <errno.h>2	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

1 For **_O_BINARY** and **_O_TEXT** definitions.

2 **errno** definitions.

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example 1

```
// crt_pipe.c
/* This program uses the _pipe function to pass streams of
 * text to spawned processes.
 */

#include <stdlib.h>
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <process.h>
#include <math.h>

enum PIPES { READ, WRITE }; /* Constants 0 and 1 for READ and WRITE */
#define NUMPROBLEM 8

int main( int argc, char *argv[] )
{

   int fdpipe[2];
   char hstr[20];
   int pid, problem, c;
   int termstat;

   /* If no arguments, this is the spawning process */
   if( argc == 1 )
   {

      setvbuf( stdout, NULL, _IONBF, 0 );

      /* Open a set of pipes */
      if( _pipe( fdpipe, 256, O_BINARY ) == -1 )
          exit( 1 );


      /* Convert pipe read descriptor to string and pass as argument
       * to spawned program. Program spawns itself (argv[0]).
       */
      _itoa_s( fdpipe[READ], hstr, sizeof(hstr), 10 );
      if( ( pid = _spawnl( P_NOWAIT, argv[0], argv[0],
            hstr, NULL ) ) == -1 )
```

```
            printf( "Spawn failed" );

        /* Put problem in write pipe. Since spawned program is
         * running simultaneously, first solutions may be done
         * before last problem is given.
         */
        for( problem = 1000; problem <= NUMPROBLEM * 1000; problem += 1000)
        {

           printf( "Son, what is the square root of %d?\n", problem );
           _write( fdpipe[WRITE], (char *)&problem, sizeof( int ) );

        }

        /* Wait until spawned program is done processing. */
        _cwait( &termstat, pid, WAIT_CHILD );
        if( termstat & 0x0 )
           printf( "Child failed\n" );


        _close( fdpipe[READ] );
        _close( fdpipe[WRITE] );

     }

     /* If there is an argument, this must be the spawned process. */
     else
     {

        /* Convert passed string descriptor to integer descriptor. */
        fdpipe[READ] = atoi( argv[1] );

        /* Read problem from pipe and calculate solution. */
        for( c = 0; c < NUMPROBLEM; c++ )
        {

          _read( fdpipe[READ], (char *)&problem, sizeof( int ) );
          printf( "Dad, the square root of %d is %3.2f.\n",
                   problem, sqrt( ( double )problem ) );

        }
     }
  }
```

Sample Output

```
Son, what is the square root of 1000?
Son, what is the square root of 2000?
Son, what iDad, the square root of 1000 is 31.62.
Dad, the square root of 2000 is 44.72.
s the square root of 3000?
Dad, the square root of 3000 is 54.77.
Son, what is the square root of 4000?
Dad, the square root of 4000 is 63.25.
Son, what is the square root of 5000?
Dad, the square root of 5000 is 70.71.
Son, what is the square root of 6000?
SonDad, the square root of 6000 is 77.46.
, what is the square root of 7000?
Dad, the square root of 7000 is 83.67.
Son, what is the square root of 8000?
Dad, the square root of 8000 is 89.44.
```

Example 2

This is a simple filter application. It spawns the application crt_pipe_beeper after creating a pipe that directs the spawned application's stdout to the filter. The filter removes ASCII 7 (beep) characters.

```
// crt_pipe_beeper.c

#include <stdio.h>
#include <string.h>

int main()
{
   int   i;
   for(i=0;i<10;++i)
      {
         printf("This is speaker beep number %d...\n\7", i+1);
      }
   return 0;
}
```

The actual filter application:

```
// crt_pipe_BeepFilter.C
// arguments: crt_pipe_beeper.exe

#include <windows.h>
#include <process.h>
#include <memory.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

#define   OUT_BUFF_SIZE 512
#define   READ_FD 0
#define   WRITE_FD 1
#define   BEEP_CHAR 7

char szBuffer[OUT_BUFF_SIZE];

int Filter(char* szBuff, ULONG nSize, int nChar)
{
   char* szPos = szBuff + nSize -1;
   char* szEnd = szPos;
   int nRet = nSize;

   while (szPos > szBuff)
   {
      if (*szPos == nChar)
         {
            memmove(szPos, szPos+1, szEnd - szPos);
            --nRet;
         }
      --szPos;
   }
   return nRet;
}

int main(int argc, char** argv)
{
   int nExitCode = STILL_ACTIVE;
   if (argc >= 2)
   {
      HANDLE hProcess;
      int fdStdOut;
      int fdStdOutPipe[2];

      // Create the pipe
      if(_pipe(fdStdOutPipe, 512, O_NOINHERIT) == -1)
         return   1;

      // Duplicate stdout file descriptor (next line will close original)
```

```
        fdStdOut = _dup(_fileno(stdout));

        // Duplicate write end of pipe to stdout file descriptor
        if(_dup2(fdStdOutPipe[WRITE_FD], _fileno(stdout)) != 0)
           return   2;

        // Close original write end of pipe
        _close(fdStdOutPipe[WRITE_FD]);

        // Spawn process
        hProcess = (HANDLE)_spawnvp(P_NOWAIT, argv[1],
         (const char* const*)&argv[1]);

        // Duplicate copy of original stdout back into stdout
        if(_dup2(fdStdOut, _fileno(stdout)) != 0)
           return   3;

        // Close duplicate copy of original stdout
        _close(fdStdOut);

        if(hProcess)
        {
           int nOutRead;
           while   (nExitCode == STILL_ACTIVE)
           {
              nOutRead = _read(fdStdOutPipe[READ_FD],
               szBuffer, OUT_BUFF_SIZE);
              if(nOutRead)
              {
                 nOutRead = Filter(szBuffer, nOutRead, BEEP_CHAR);
                 fwrite(szBuffer, 1, nOutRead, stdout);
              }

              if(!GetExitCodeProcess(hProcess,(unsigned long*)&nExitCode))
                 return 4;
           }
        }
     }
     return nExitCode;
  }
```

Output

```
This is speaker beep number 1...
This is speaker beep number 2...
This is speaker beep number 3...
This is speaker beep number 4...
This is speaker beep number 5...
This is speaker beep number 6...
This is speaker beep number 7...
This is speaker beep number 8...
This is speaker beep number 9...
This is speaker beep number 10...
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Process and Environment Control

_open, _wopen

_popen, _wpopen

Creates a pipe and executes a command.

```
FILE *_popen(
   const char *command,
   const char *mode
);
FILE *_wpopen(
   const wchar_t *command,
   const wchar_t *mode
);
```

Parameters

command

Command to be executed.

mode

Mode of the returned stream.

Return Value

Returns a stream associated with one end of the created pipe. The other end of the pipe is associated with the spawned command's standard input or standard output. The functions return **NULL** on an error. If the error is an invalid parameter, such as if *command* or *mode* is a null pointer, or *mode* is not a valid mode, **errno** is set to **EINVAL**. See the Remarks section for valid modes.

For information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_popen** function creates a pipe and asynchronously executes a spawned copy of the command processor with the specified string *command*. The character string *mode* specifies the type of access requested, as follows.

"r"

The calling process can read the spawned command's standard output using the returned stream.

"w"

The calling process can write to the spawned command's standard input using the returned stream.

"b"

Open in binary mode.

"t"

Open in text mode.

Note
If used in a Windows program, the **_popen** function returns an invalid file pointer that causes the program to stop respond ing indefinitely. **_popen** works properly in a console application. To create a Windows application that redirects input and output, see Creating a Child Process with Redirected Input and Output in the *Platform SDK*.

_wpopen is a wide-character version of **_popen**; the *path* argument to **_wpopen** is a wide-character string. **_wpopen** and **_popen** behave identically otherwise.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tpopen	_popen	_popen	_wpopen

Requirements

Routine	Required header	Compatibility
_popen	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wpopen	<stdio.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_popen.c
/* This program uses _popen and _pclose to receive a
 * stream of text from a system process.
 */

#include <stdio.h>
#include <stdlib.h>

int main( void )
{

   char   psBuffer[128];
   FILE   *pPipe;

        /* Run DIR so that it writes its output to a pipe. Open this
         * pipe with read text attribute so that we can read it
         * like a text file.
         */

   if( (pPipe = _popen( "dir *.c /on /p", "rt" )) == NULL )
      exit( 1 );

   /* Read pipe until end of file, or an error occurs. */

   while(fgets(psBuffer, 128, pPipe))
   {
      printf(psBuffer);
   }


   /* Close pipe and print return value of pPipe. */
   if (feof( pPipe))
   {
     printf( "\nProcess returned %d\n", _pclose( pPipe ) );
   }
   else
   {
     printf( "Error: Failed to read the pipe to the end.\n");
   }
}
```

Sample Output

This output assumes that there is only one file in the current directory with a .c file name extension.

```
 Volume in drive C is CDRIVE
 Volume Serial Number is 0E17-1702
```

```
 Directory of D:\proj\console\test1

07/17/98  07:26p                  780 popen.c
               1 File(s)            780 bytes
                             86,597,632 bytes free

Process returned 0
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Process and Environment Control

_pclose

_pipe

pow, powf

Calculates *x* raised to the power of *y*.

```
double pow(
   double x,
   double y
);
double pow(
   double x,
   int y
);  // C++ only
float pow(
   float x,
   float y
);  // C++ only
float pow(
   float x,
   int y
);  // C++ only
long double pow(
   long double x,
   long double y
);  // C++ only
long double pow(
   long double x,
   int y
);  // C++ only
float powf(
   float x,
   float y
);
```

Parameters

x

Base.

y

Exponent.

Return Value

Returns the value of *xy*. No error message is printed on overflow or underflow.

Values of x and y	Return value of pow
x < > 0 and *y* = 0.0	1
x = 0.0 and *y* = 0.0	1
x = 0.0 and *y* < 0	INF

Remarks

The **pow** function computes *x* raised to the power of *y*.

pow does not recognize integral floating-point values greater than 2^{64}, such as `1.0E100`.

pow has an implementation that uses Streaming SIMD Extensions 2 (SSE2). See _set_SSE2_enable for information and restrictions on using the SSE2 implementation.

C++ allows overloading, so you can call any of the various overloads of **pow**. In a C program, **pow** always takes two double values and returns a double value.

Visual C++ 2005 introduces a breaking change to better conform with the ISO standard. The **pow(int, int)** overload is no longer available. If you are using this overload, the compiler may emit C2668. To avoid this problem, cast the first parameter to **double**, **float**, or **long double**.

Requirements

Routine	Required header	Compatibility
pow, **powf**	<math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_pow.c

#include <math.h>
#include <stdio.h>

int main( void )
{
   double x = 2.0, y = 3.0, z;

   z = pow( x, y );
   printf( "%.1f to the power of %.1f is %.1f\n", x, y, z );
}
```

Output

```
2.0 to the power of 3.0 is 8.0
```

.NET Framework Equivalent

System::Math::Pow

See Also

Reference

Floating-Point Support
exp, expf
log, logf, log10, log10f
sqrt, sqrtf

Precision Specification

The third optional field of the format specification is the precision specification. It specifies a nonnegative decimal integer, preceded by a period (**.**), which specifies the number of characters to be printed, the number of decimal places, or the number of significant digits (see the How Precision Values Affect Type table). Unlike the width specification, the precision specification can cause either truncation of the output value or rounding of a floating-point value. If *precision* is specified as 0 and the value to be converted is 0, the result is no characters output, as shown below:

```
printf( "%.0d", 0 );      /* No characters output */
```

If the precision specification is an asterisk (*****), an **int** argument from the argument list supplies the value. The *precision* argument must precede the value being formatted in the argument list.

The type determines the interpretation of *precision* and the default when *precision* is omitted, as shown in the following table.

How Precision Values Affect Type

Type	Meaning	Default
a, A	The precision specifies the number of digits after the point.	Default precision is 6. If precision is 0, no point is printed unless the # flag is used.
c, C	The precision has no effect.	Character is printed.
d, i, u, o, x, X	The precision specifies the minimum number of digits to be printed. If the number of digits in the argument is less than *precision*, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds *precision*.	Default precision is 1.
e, E	The precision specifies the number of digits to be printed after the decimal point. The last printed digit is rounded.	Default precision is 6; if *precision* is 0 or the period (.) appears without a number following it, no decimal point is printed.
f	The precision value specifies the number of digits after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.	Default precision is 6; if *precision* is 0, or if the period (.) appears without a number following it, no decimal point is printed.
g, G	The precision specifies the maximum number of significant digits printed.	Six significant digits are printed, with any trailing zeros truncated.
s, S	The precision specifies the maximum number of characters to be printed. Characters in excess of *precision* are not printed.	Characters are printed until a null character is encountered.

If the argument corresponding to a floating-point specifier is infinite, indefinite, or NAN, **printf** gives the following output.

Value	Output
+ infinity	**1.#INF** *random-digits*
– infinity	**–1.#INF** *random-digits*
Indefinite (same as quiet NaN)	*digit.#* **IND** *random-digits*
NAN	*digit.#* **NAN** *random-digits*

See Also

Reference

printf, _printf_l, wprintf, _wprintf_l

printf Type Field Characters

The *type* character is the only required format field; it appears after any optional format fields. The *type* character determines whether the associated argument is interpreted as a character, string, or number. The types **C**, **n**, **p**, and **S**, and the behavior of **c** and **s** with **printf** functions, are Microsoft extensions and are not ANSI compatible.

printf Type Field Characters

Character	Type	Output format
c	**int** or **wint_t**	When used with **printf** functions, specifies a single-byte character; when used with **wprintf** functions, specifies a wide character.
C	**int** or **wint_t**	When used with **printf** functions, specifies a wide character; when used with **wprintf** functions, specifies a single-byte character.
d	**int**	Signed decimal integer.
i	**int**	Signed decimal integer.
o	**int**	Unsigned octal integer.
u	**int**	Unsigned decimal integer.
x	**int**	Unsigned hexadecimal integer, using "abcdef."
X	**int**	Unsigned hexadecimal integer, using "ABCDEF."
e	**double**	Signed value having the form [–]*d***.***dddd* **e** [*sign*]*dd[d]* where *d* is a single decimal digit, *dddd* is one or more decimal digits, *dd[d]* is two or three decimal digits depending on the output format and size of the exponent, and *sign* is + or –.
E	**double**	Identical to the **e** format except that **E** rather than **e** introduces the exponent.
f	**double**	Signed value having the form [–]*dddd***.***dddd*, where *dddd* is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision.
g	**double**	Signed value printed in **f** or **e** format, whichever is more compact for the given value and precision. The **e** format is used only when the exponent of the value is less than –4 or greater than or equal to the precision argument. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.
G	**double**	Identical to the **g** format, except that **E**, rather than **e**, introduces the exponent (where appropriate).
a	**double**	Signed hexadecimal double precision floating point value having the form [–]0x*h.hhhh* **p±***dd*, where *h.hhhh* are the hex digits (using lower case letters) of the mantissa, and *dd* are one or more digits for the exponent. The precision specifies the number of digits after the point.

A	**double**	Signed hexadecimal double precision floating point value having the form [−]0X*h.hhhh* **P±***dd*, where *h.hhhh* are the hex digits (using capital letters) of the mantissa, and *dd* are one or more digits for the exponent. The precision specifies the number of digits after the point.
n	Pointer to integer	Number of characters successfully written so far to the stream or buffer; this value is stored in the integer whose address is given as the argument. See Security Note below.
p	Pointer to **void**	Prints the address of the argument in hexadecimal digits.
s	String	When used with **printf** functions, specifies a single-byte–character string; when used with **wprintf** functions, specifies a wide-character string. Characters are printed up to the first null character or until the *precision* value is reached.
S	String	When used with **printf** functions, specifies a wide-character string; when used with **wprintf** functions, specifies a single-byte–character string. Characters are printed up to the first null character or until the *precision* value is reached.

Note If the argument corresponding to **%s** or **%S** is a null pointer, "(null)" will be printed.

Note In all exponential formats, the default number of digits of exponent to display is three. Using the _set_output_format function, the number of digits displayed may be set to two, expanding to three if demanded by the size of exponent.

Security Note The **%n** format is inherently insecure and is disabled by default; if **%n** is encountered in a format string, the invalid parameter handler is invoked as described in Parameter Validation. To enable **%n** support, see _set_printf_count_output.

See Also
Reference
printf, _printf_l, wprintf, _wprintf_l

printf Width Specification

The second optional field of the format specification is the width specification. The *width* argument is a nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified width, blanks are added to the left or the right of the values — depending on whether the – flag (for left alignment) is specified — until the minimum width is reached. If *width* is prefixed with 0, zeros are added until the minimum width is reached (not useful for left-aligned numbers).

The width specification never causes a value to be truncated. If the number of characters in the output value is greater than the specified width, or if *width* is not given, all characters of the value are printed (subject to the precision specification).

If the width specification is an asterisk (*), an **int** argument from the argument list supplies the value. The *width* argument must precede the value being formatted in the argument list. A nonexistent or small field width does not cause the truncation of a field; if the result of a conversion is wider than the field width, the field expands to contain the conversion result.

See Also
Reference
printf, _printf_l, wprintf, _wprintf_l

printf, _printf_l, wprintf, _wprintf_l

Print formatted output to the standard output stream. More secure versions are available, see printf_s, _printf_s_l, wprintf_s, _wprintf_s_l.

```
int printf(
   const char *format [,
      argument]...
);
int _printf_l(
   const char *format,
   locale_t locale [,
      argument]...
);
int wprintf(
   const wchar_t *format [,
      argument]...
);
int _wprintf_l(
   const wchar_t *format,
   locale_t locale [,
      argument]...
);
```

Parameters

format

Format control.

argument

Optional arguments.

locale

The locale to use.

Return Value

Returns the number of characters printed, or a negative value if an error occurs. If *format* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns -1 and sets **errno** to **EINVAL**.

For information on **errno** and error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **printf** function formats and prints a series of characters and values to the standard output stream, **stdout**. If arguments follow the *format* string, the *format* string must contain specifications that determine the output format for the arguments. **printf** and fprintf behave identically except that **printf** writes output to **stdout** rather than to a destination of type **FILE**.

wprintf is a wide-character version of **printf**; *format* is a wide-character string. **wprintf** and **printf** behave identically if the stream is opened in ANSI mode. **printf** does not currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_unicode defined
_tprintf	printf	printf	wprintf

The *format* argument consists of ordinary characters, escape sequences, and (if arguments follow *format*) format specifications. The ordinary characters and escape sequences are copied to **stdout** in order of their appearance. For example, the line:

```
printf("Line one\n\t\tLine two\n");
```

produces the output:

```
Line one
        Line two
```

Format specifications always begin with a percent sign (**%**) and are read left to right. When **printf** encounters the first format specification (if any), it converts the value of the first argument after *format* and outputs it accordingly. The second format specification causes the second argument to be converted and output, and so on. If there are more arguments than there are format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for all the format specifications.

Security Note
Ensure that *format* is not a user-defined string.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tprintf	**printf**	**printf**	**wprintf**
_tprintf_l	**_printf_l**	**_printf_l**	**_wprintf_l**

Requirements

Routine	Required header	Compatibility
printf, **_printf_l**	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wprintf, **_wprintf_l**	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_printf.c
// This program uses the printf and wprintf functions
// to produce formatted output.

#include <stdio.h>

int main( void )
{
   char     ch = 'h',
            *string = "computer";
   wchar_t  wch = L'w',
            *wstring = L"Unicode";
   int      count = -9234;
   double   fp = 251.7366;

   // Display integers
   printf( "Integer formats:\n"
           "   Decimal: %d  Justified: %.6d  "
           "Unsigned: %u\n",
           count, count, count, count );

   // Display decimals
   printf( "Decimal %d as:\n   Hex: %Xh  "
           "C hex: 0x%x  Octal: %o\n",
```

```
             count, count, count, count );

   // Display in different radixes
   printf( "Digits 10 equal:\n   Hex: %i  "
           "Octal: %i  Decimal: %i\n",
            0x10, 010, 10 );

   // Display characters
   printf("Characters in field (1):\n"
          "%10c%5hc%5C%5lc\n",
          ch, ch, wch, wch);
   wprintf(L"Characters in field (2):\n"
           L"%10C%5hc%5c%5lc\n",
           ch, ch, wch, wch);

   // Display strings
   printf("Strings in field (1):\n%25s\n"
          "%25.4hs\n   %S%25.3ls\n",
          string, string, wstring, wstring);
   wprintf(L"Strings in field (2):\n%25S\n"
           L"%25.4hs\n   %s%25.3ls\n",
           string, string, wstring, wstring);

   // Display real numbers
   printf("Real numbers:\n   %f %.2f %e %E\n",
          fp, fp, fp, fp );

   // Display pointer
   printf( "\nAddress as:   %p\n", &count);
}
```

Sample Output

```
Integer formats:
   Decimal: -9234  Justified: -009234  Unsigned: 4294958062
Decimal -9234 as:
   Hex: FFFFDBEEh  C hex: 0xffffdbee  Octal: 37777755756
Digits 10 equal:
   Hex: 16  Octal: 8  Decimal: 10
Characters in field (1):
         h    h    w    w
Characters in field (2):
         h    h    w    w
Strings in field (1):
                 computer
                     comp
   Unicode                      Uni
Strings in field (2):
                 computer
                     comp
   Unicode                      Uni
Real numbers:
   251.736600 251.74 2.517366e+002 2.517366E+002

Address as:   0012FF3C
```

.NET Framework Equivalent

- System::Console::Write
- System::Console::WriteLine

See Also

Reference

Floating-Point Support
Stream I/O
Locale

fopen, _wfopen
_fprintf_p, _fprintf_p_l, _fwprintf_p, _fwprintf_p_l
scanf, _scanf_l, wscanf, _wscanf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
vprintf Functions

printf_p Positional Parameters

Positional parameters provide the ability to specify by number which of the arguments is to be substituted into a field in a format string. The following positional parameter **printf** functions are available:

printf, _printf_l, wprintf, _wprintf_l

_printf_p, _printf_p_l, _wprintf_p, _wprintf_p_l

sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l

_sprintf_p, _sprintf_p_l, _swprintf_p, _swprintf_p_l

_cprintf, _cprintf_l, _cwprintf, _cwprintf_l

_cprintf_p, _cprintf_p_l, _cwprintf_p, _cwprintf_p_l

fprintf, _fprintf_l, fwprintf, _fwprintf_l

_fprintf_p, _fprintf_p_l, _fwprintf_p, _fwprintf_p_l

vprintf, _vprintf_l, vwprintf, _vwprintf_l

_vprintf_p, _vprintf_p_l, _vwprintf_p, _vwprintf_p_l

vfprintf, _vfprintf_l, vfwprintf, _vfwprintf_l

_vfprintf_p, _vfprintf_p_l, _vfwprintf_p, _vfwprintf_p_l

vsprintf, _vsprintf_l, vswprintf, _vswprintf_l, __vswprintf_l

_vsprintf_p, _vsprintf_p_l, _vswprintf_p, _vswprintf_p_l

Specifying Positional Parameters

Parameter Indexing

By default the positional functions behave identically to the non position ones, if no positional formatting is present. Positional parameters are specified using the format "*%m$x*", where *m* denotes a numeric ordinal number indicating the position of the parameter in the list of parameters, preceding the format string and *x* denotes the type field character type specified in the **printf** function. The parameters in the list are indexed starting at the value 1 for the first element in the list and so forth. For additional information concerning type field characters, see printf Type Field Characters.

For an example of this behavior:

```
_printf_p("%1$s %2$s", "November", "10");
```

will print

```
November 10
```

The order of the numbers used need not match the order of the arguments given. Thus the following is valid:

```
_printf_p("%2$s %1$s", "November", "10");
```

will print

```
10 November
```

Parameter may be used more than once while formatting, unlike in traditional format strings, so that

```
_printf_p("%{1$d times %1$d is %2$d", 10, 100);
```

will print

```
10 times 10 is 100
```

However, all arguments must be used at least once somewhere in the format string.

The maximum number of positional parameters allowed in a format string is given by **_ARGMAX**.

Width and Prevision

When the * symbol is used to specify that the width or precision is to be determined from an argument, then the position of the width or prevision value must appear immediately following the * symbol. For example,

```
_printf_p("%1$*2$s","Hello", 10);
```

or

```
_printf_p("%2$*1$s",10, "Hello");
```

Mixing positional and non positional arguments

Positional parameters may not be mixed with non-positional parameters in the same format string. However, **printf_s** and related functions still support non-positional parameters in format strings containing no positional parameters.

Example

```
// positional_args.c
// Positional arguments allow the specification of the order
// in which arguments are consumed in a formatting string.

#include <stdio.h>

int main(int argc, char *argv[])
{
    int     i = 1,
            j = 2,
            k = 3;
    double  x = 0.1,
            y = 0.2,
            z = 0.3;
    char    *s1 = "abc",
            *s2 = "def",
            *s3 = "ghi";

    // If positional arguments are unspecified,
    // normal input order is used.
    _printf_p("%d %d %d\n", i, j, k);

    // Positional args are numbers indicating the
    // argument enclosed in curly braces.
    _printf_p("%3$d %1$d %2$d\n", i, j, k);

    // The same positional argument may be reused.
    _printf_p("%1$d %2$d %1$d\n", i, j);

    _printf_p("%1$s %2$s %3$s\n", s1, s2, s3);

    _printf_p("%3$s %1$s %2$s\n", s1, s2, s3);
}
```

Output

```
1 2 3
3 1 2
1 2 1
abc def ghi
ghi abc def
```

See Also

Reference

printf Type Field Characters

printf Width Specification

Precision Specification

_printf_p, _printf_p_l, _wprintf_p, _wprintf_p_l

Prints formatted output to the standard output stream, with the ability to specify the order in which parameters are used in the format string.

```
int _printf_p(
   const char *format [,
      argument]...
);
int _printf_p_l(
   const char *format,
   locale_t locale [,
      argument]...
);
int _wprintf_p(
   const wchar_t *format [,
      argument]...
);
int _wprintf_p_l(
   const wchar_t *format,
   locale_t locale [,
      argument]...
);
```

Parameters

format

Format control.

argument

Optional arguments.

locale

The locale to use.

Return Value

Returns the number of characters printed or a negative value if an error occurs.

Remarks

The **_printf_p** function formats and prints a series of characters and values to the standard output stream, **stdout**. If arguments follow the *format* string, the *format* string must contain specifications that determine the output format for the arguments (see printf_p Positional Parameters).

The difference between **_printf_p** and **printf_s** is that **_printf_p** supports positional parameters, which allows specifying the order in which the arguments are used in the format string. For more information, see printf_p Positional Parameters.

_wprintf_p is the wide-character version of **_printf_p**; they behave identically if the stream is opened in ANSI mode. **_printf_p** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Security Note
Ensure that *format* is not a user-defined string.

If *format* or *argument* are **NULL**, or of the format string contains invalid formatting characters, **_printf_p** and **_wprintf_p** functions invoke an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the function returns -1 and sets **errno** to **EINVAL**.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tprintf_p	**_printf_p**	**_printf_p**	**_wprintf_p**
_tprintf_p_l	**_printf_p_l**	**_printf_p_l**	**_wprintf_p_l**

Requirements

Routine	Required header	Compatibility
_printf_p, **_printf_p_l**	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wprintf_p, **_wprintf_p_l**	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_printf_p.c
// This program uses the _printf_p and _wprintf_p
// functions to choose the order in which parameters
// are used.

#include <stdio.h>

int main( void )
{
   // Positional arguments
   _printf_p( "Specifying the order: %2$s %3$s %1$s %4$s %5$s.\n",
              "little", "I'm", "a", "tea", "pot");

   // Resume arguments
   _wprintf_p( L"Reusing arguments: %1$d %1$d %1$d %1$d\n", 10);

   // Width argument
   _printf_p("Width specifiers: %1$*2$s", "Hello\n", 10);
}
```

Output

```
Specifying the order: I'm a little tea pot.
Reusing arguments: 10 10 10 10
Width specifiers:     Hello
```

.NET Framework Equivalent

- System::Console::Write
- System::Console::WriteLine

See Also

Reference

Floating-Point Support
Stream I/O
Locale
fopen, _wfopen
_fprintf_p, _fprintf_p_l, _fwprintf_p, _fwprintf_p_l
fprintf, _fprintf_l, fwprintf, _fwprintf_l
fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l
scanf, _scanf_l, wscanf, _wscanf_l
scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l
_sprintf_p, _sprintf_p_l, _swprintf_p, _swprintf_p_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l

sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l
vprintf Functions

printf_s, _printf_s_l, wprintf_s, _wprintf_s_l

Print formatted output to the standard output stream. These are versions of printf, _printf_l, wprintf, _wprintf_l with security enhancements as described in Security Enhancements in the CRT.

```
int printf_s(
   const char *format [,
      argument]...
);
int _printf_s_l(
   const char *format,
   locale_t locale [,
      argument]...
);
int wprintf_s(
   const wchar_t *format [,
      argument]...
);
int _wprintf_s_l(
   const wchar_t *format,
   locale_t locale [,
      argument]...
);
```

Parameters

format

Format control.

argument

Optional arguments.

locale

The locale to use.

Return Value

Returns the number of characters printed, or a negative value if an error occurs.

Remarks

The **printf_s** function formats and prints a series of characters and values to the standard output stream, **stdout**. If arguments follow the *format* string, the *format* string must contain specifications that determine the output format for the arguments.

The main difference between **printf_s** and **printf** is that **printf_s** checks the format string for valid formatting characters, whereas **printf** only checks if the format string is a null pointer. If either check fails, an invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns -1 and sets **errno** to **EINVAL**.

For information on **errno** and error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

printf_s and **fprintf_s** behave identically except that **printf_s** writes output to **stdout** rather than to a destination of type **FILE**. For more information, see fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l.

wprintf_s is a wide-character version of **printf_s**; *format* is a wide-character string. **wprintf_s** and **printf_s** behave identically if the stream is opened in ANSI mode. **printf_s** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_unicode defined
_tprintf_s	printf_s	printf_s	wprintf_s

_tprintf_s_l	_printf_s_l	_printf_s_l	_wprintf_s_l

The *format* argument consists of ordinary characters, escape sequences, and (if arguments follow *format*) format specifications. The ordinary characters and escape sequences are copied to **stdout** in order of their appearance. For example, the line

```
printf_s("Line one\n\t\tLine two\n");
```

produces the output

```
Line one
        Line two
```

Format specifications always begin with a percent sign (**%**) and are read left to right. When **printf_s** encounters the first format specification (if any), it converts the value of the first argument after *format* and outputs it accordingly. The second format specification causes the second argument to be converted and output, and so on. If there are more arguments than there are format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for all the format specifications.

Security Note
Ensure that *format* is not a user-defined string.

Requirements

Routine	Required header	Compatibility
printf_s, **_printf_s_l**	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wprintf_s, **_wprintf_s_l**	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_printf_s.c
/* This program uses the printf_s and wprintf_s functions
 * to produce formatted output.
 */

#include <stdio.h>

int main( void )
{
   char   ch = 'h', *string = "computer";
   int    count = -9234;
   double fp = 251.7366;
   wchar_t wch = L'w', *wstring = L"Unicode";

   /* Display integers. */
   printf_s( "Integer formats:\n"
           "   Decimal: %d  Justified: %.6d  Unsigned: %u\n",
           count, count, count, count );

   printf_s( "Decimal %d as:\n   Hex: %Xh  C hex: 0x%x  Octal: %o\n",
            count, count, count, count );

   /* Display in different radixes. */
   printf_s( "Digits 10 equal:\n   Hex: %i  Octal: %i  Decimal: %i\n",
            0x10, 010, 10 );
```

```
    /* Display characters. */

    printf_s("Characters in field (1):\n%10c%5hc%5C%5lc\n", ch, ch, wch, wch);
    wprintf_s(L"Characters in field (2):\n%10C%5hc%5c%5lc\n", ch, ch, wch, wch);

    /* Display strings. */

    printf_s("Strings in field (1):\n%25s\n%25.4hs\n   %S%25.3ls\n",
    string, string, wstring, wstring);
    wprintf_s(L"Strings in field (2):\n%25S\n%25.4hs\n   %s%25.3ls\n",
        string, string, wstring, wstring);

    /* Display real numbers. */
    printf_s( "Real numbers:\n   %f %.2f %e %E\n", fp, fp, fp, fp );

    /* Display pointer. */
    printf_s( "\nAddress as:   %p\n", &count);

}
```

Sample Output

```
Integer formats:
   Decimal: -9234  Justified: -009234  Unsigned: 4294958062
Decimal -9234 as:
   Hex: FFFFDBEEh  C hex: 0xffffdbee  Octal: 37777755756
Digits 10 equal:
   Hex: 16  Octal: 8  Decimal: 10
Characters in field (1):
         h    h    w    w
Characters in field (2):
         h    h    w    w
Strings in field (1):
                 computer
                     comp
   Unicode                      Uni
Strings in field (2):
                 computer
                     comp
   Unicode                      Uni
Real numbers:
   251.736600 251.74 2.517366e+002 2.517366E+002

Address as:   0012FF78
```

.NET Framework Equivalent

- System::Console::Write
- System::Console::WriteLine

See Also

Reference

Floating-Point Support
Stream I/O
Locale
fopen, _wfopen
fprintf, _fprintf_l, fwprintf, _fwprintf_l
scanf, _scanf_l, wscanf, _wscanf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
vprintf Functions

putc, putwc

Write a character to a stream.

```
int putc(
   int c,
   FILE *stream
);
wint_t putwc(
   wchar_t c,
   FILE *stream
);
```

Parameters

c

Character to be written.

stream

Pointer to **FILE** structure.

Return Value

Returns the character written. To indicate an error or end-of-file condition, **putc** and **putchar** return **EOF**; **putwc** and **putwchar** return **WEOF**. For all four routines, use ferror or feof to check for an error or end of file. If passed a null pointer for *stream*, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **EOF** or **WEOF** and set **errno** to **EINVAL**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

The **putc** routine writes the single character *c* to the output *stream* at the current position. Any integer can be passed to **putc**, but only the lower 8 bits are written. The **putchar** routine is identical to **putc(** *c***, stdout)**. For each routine, if a read error occurs, the error indicator for the stream is set. **putc** and **putchar** are similar to **fputc** and **_fputchar**, respectively, but are implemented both as functions and as macros (see Choosing Between Functions and Macros). **putwc** and **putwchar** are wide-character versions of **putc** and **putchar**, respectively. **putwc** and **putc** behave identically if the stream is opened in ANSI mode. **putc** doesn't currently support output into a UNICODE stream.

The versions with the **_nolock** suffix are identical except that they are not protected from interference by other threads. For more information, see **_putc_nolock, _putwc_nolock**.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_puttc	**putc**	**putc**	**putwc**

Requirements

Routine	Required header	Compatibility
putc	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
putwc	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_putc.c
/* This program uses putc to write buffer
 * to a stream. If an error occurs, the program
 * stops before writing the entire buffer.
 */

#include <stdio.h>

int main( void )
{
   FILE *stream;
   char *p, buffer[] = "This is the line of output\n";
   int  ch;

   ch = 0;
   /* Make standard out the stream and write to it. */
   stream = stdout;
   for( p = buffer; (ch != EOF) && (*p != '\0'); p++ )
      ch = putc( *p, stream );
}
```

Output

```
This is the line of output
```

.NET Framework Equivalent

- System::IO::StreamWriter::Write
- System::Console::Write

See Also

Reference

Stream I/O

fputc, fputwc

getc, getwc

_putc_nolock, _putwc_nolock

Writes a character to a stream without locking the thread.

```
int _putc_nolock(
   int c,
   FILE *stream
);
wint_t _putwc_nolock(
   wchar_t c,
   FILE *stream
);
```

Parameters

c

Character to be written.

stream

Pointer to the **FILE** structure.

Return Value

See **putc, putwc**.

Remarks

_putc_nolock and **_putwc_nolock** are identical to the versions without the **_nolock** suffix except that they are not protected from interference by other threads. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

_putwc_nolock is the wide-character version of **_putc_nolock**; the two functions behave identically if the stream is opened in ANSI mode. **_putc_nolock** doesn't currently support output into a UNICODE stream.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_puttc_nolock	**_putc_nolock**	**_putc_nolock**	**_putwc_nolock**

Requirements

Routine	Required header	Compatibility
_putc_nolock	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_putwc_nolock	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_putc_nolock.c
/* This program uses putc to write buffer
 * to a stream. If an error occurs, the program
 * stops before writing the entire buffer.
```

```
 */

#include <stdio.h>

int main( void )
{
   FILE *stream;
   char *p, buffer[] = "This is the line of output\n";
   int  ch;

   ch = 0;
   /* Make standard out the stream and write to it. */
   stream = stdout;
   for( p = buffer; (ch != EOF) && (*p != '\0'); p++ )
      ch = _putc_nolock( *p, stream );
}
```

Output

```
This is the line of output
```

.NET Framework Equivalent

- System::IO::StreamWriter::Write
- System::Console::Write

See Also

Reference

Stream I/O
fputc, fputwc
getc, getwc

putch

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _putch instead.

_putch, _putwch

Writes a character to the console.

```
int _putch(
   int c
);
wint_t _putwch(
   wchar_t c
);
```

Parameters

c

Character to be output.

Return Value

Returns *c* if successful. If **_putch** fails, it returns **EOF**; if **_putwch** fails, it returns **WEOF**.

Remarks

These functions write the character *c* directly, without buffering, to the console. In Windows NT, **_putwch** writes Unicode characters using the current console locale setting.

The versions with the **_nolock** suffix are identical except that they are not protected from interference by other threads. For more information, see **_putch_nolock**, **_putwch_nolock**.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_puttch	**_putch**	**_putch**	**_putwch**

Requirements

Routine	Required header	Compatibility
_putch	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_putwch	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

See the example for _getch.

See Also

Reference

Console and Port I/O
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l
_getch, _getwch

_putch_nolock, _putwch_nolock

Writes a character to the console without locking the thread.

```
int _putch_nolock(
   int c
);
wint_t _putwch_nolock(
   wchar_t c
);
```

Parameters

c

Character to be output.

Return Value

Returns *c* if successful. If **_putch_nolock** fails, it returns **EOF**; if **_putwch_nolock** fails, it returns **WEOF**.

Remarks

_putch_nolock and **_putwch_nolock** are identical to **_putch** and **_putwch**, respectively, except that they are not protected from interference by other threads. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_puttch_nolock	**_putch_nolock**	**_putch_nolock**	**_putwch_nolock**

Requirements

Routine	Required header	Compatibility
_putch_nolock	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_putwch_nolock	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See Also

Reference

Console and Port I/O

_cprintf, _cprintf_l, _cwprintf, _cwprintf_l

_getch, _getwch

putchar, putwchar

Write a character to **stdout**.

```
int putchar(
   int c
);
wint_t putwchar(
   wchar_t c
);
```

Parameters

c

Character to be written.

Return Value

Returns the character written. To indicate an error or end-of-file condition, **putc** and **putchar** return **EOF**; **putwc** and **putwchar** return **WEOF**. For all four routines, use ferror or feof to check for an error or end of file. If passed a null pointer for *stream*, these functions generate an invalid parameter exception, as described in Parameter Validation. If execution is allowed to continue, they return **EOF** or **WEOF** and set **errno** to **EINVAL**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

The **putc** routine writes the single character *c* to the output *stream* at the current position. Any integer can be passed to **putc**, but only the lower 8 bits are written. The **putchar** routine is identical to **putc(** *c***, stdout)**. For each routine, if a read error occurs, the error indicator for the stream is set. **putc** and **putchar** are similar to **fputc** and **_fputchar**, respectively, but are implemented both as functions and as macros (see Choosing Between Functions and Macros). **putwc** and **putwchar** are wide-character versions of **putc** and **putchar**, respectively.

The versions with the **_nolock** suffix are identical except that they are not protected from interference by other threads. They may be faster since they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_puttchar	**putchar**	**putchar**	**putwchar**

Requirements

Routine	Required header	Compatibility
putchar	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
putwchar	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_putchar.c
/* This program uses putc to write buffer
```

```
 * to a stream. If an error occurs, the program
 * stops before writing the entire buffer.
 */

#include <stdio.h>

int main( void )
{
   FILE *stream;
   char *p, buffer[] = "This is the line of output\n";
   int  ch;

   ch = 0;

   for( p = buffer; (ch != EOF) && (*p != '\0'); p++ )
      ch = putchar( *p );
}
```

Output

```
This is the line of output
```

.NET Framework Equivalent

- System::IO::StreamWriter::Write
- System::Console::Write

See Also

Reference

Stream I/O
fputc, fputwc
getc, getwc

_putchar_nolock, _putwchar_nolock

Writes a character to **stdout** without locking the thread.

```
int _putchar_nolock(
   int c
);
wint_t _putwchar_nolock(
   wchar_t c
);
```

Parameters

c

Character to be written.

Return Value

See **putchar, putwchar**.

Remarks

putchar_nolock and **_putwchar_nolock** are identical to the versions without the **_nolock** suffix except that they are not protected from interference by other threads. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_puttchar_nolock	**_putchar_nolock**	**_putchar_nolock**	**_putwchar_nolock**

Requirements

Routine	Required header	Compatibility
_putchar_nolock	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_putwchar_nolock	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_putchar_nolock.c
/* This program uses putchar to write buffer
 * to stdout. If an error occurs, the program
 * stops before writing the entire buffer.
 */

#include <stdio.h>

int main( void )
{
   FILE *stream;
```

```
    char *p, buffer[] = "This is the line of output\n";
    int  ch;

    ch = 0;

    for( p = buffer; (ch != EOF) && (*p != '\0'); p++ )
       ch = _putchar_nolock( *p );
 }
```

Output

```
This is the line of output
```

.NET Framework Equivalent

- System::IO::StreamWriter::Write
- System::Console::Write

See Also

Reference

Stream I/O
fputc, fputwc
fgetc, fgetwc

putenv

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _putenv or security-enhanced _putenv_s instead.

_putenv, _wputenv

Creates, modifies, or removes environment variables. These functions are deprecated because more secure versions are available; see _putenv_s, _wputenv_s.

```
int _putenv(
   const char *envstring
);
int _wputenv(
   const wchar_t *envstring
);
```

Parameters

envstring

Environment-string definition.

Return Value

Return 0 if successful or –1 in the case of an error.

Remarks

The **_putenv** function adds new environment variables or modifies the values of existing environment variables. Environment variables define the environment in which a process executes (for example, the default search path for libraries to be linked with a program). **_wputenv** is a wide-character version of **_putenv**; the *envstring* argument to **_wputenv** is a wide-character string.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tputenv	**_putenv**	**_putenv**	**_wputenv**

The *envstring* argument must be a pointer to a string of the form *varname=string*, where *varname* is the name of the environment variable to be added or modified and *string* is the variable's value. If *varname* is already part of the environment, its value is replaced by *string*; otherwise, the new *varname* variable and its *string* value are added to the environment. You can remove a variable from the environment by specifying an empty *string* — in other words, by specifying only *varname=*.

_putenv and **_wputenv** affect only the environment that is local to the current process; you cannot use them to modify the command-level environment. That is, these functions operate only on data structures accessible to the run-time library and not on the environment segment created for a process by the operating system. When the current process terminates, the environment reverts to the level of the calling process (in most cases, the operating-system level). However, the modified environment can be passed to any new processes created by **_spawn**, **_exec**, or **system**, and these new processes get any new items added by **_putenv** and **_wputenv**.

Do not change an environment entry directly: instead, use **_putenv** or **_wputenv** to change it. In particular, direct freeing elements of the **_environ[]** global array might lead to invalid memory being addressed.

getenv and **_putenv** use the global variable **_environ** to access the environment table; **_wgetenv** and **_wputenv** use **_wenviron**. **_putenv** and **_wputenv** might change the value of **_environ** and **_wenviron**, thus invalidating the **_envp** argument to **main** and the *_wenvp* argument to **wmain**. Therefore, it is safer to use **_environ** or **_wenviron** to access the environment information. For more information about the relation of **_putenv** and **_wputenv** to global variables, see _environ, _wenviron.

Note
The **_putenv** and **_getenv** families of functions are not thread-safe. **_getenv** could return a string pointer while **_putenv** is modifying the string, causing random failures. Make sure that calls to these functions are synchronized.

Requirements

Routine	Required header	Compatibility

_putenv	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wputenv	<stdlib.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

For a sample of how to use **_putenv**, see getenv, _wgetenv.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Process and Environment Control
getenv, _wgetenv
_searchenv, _wsearchenv

_putenv_s, _wputenv_s

Create, modify, or remove environment variables. These are versions of _putenv, _wputenv with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _putenv_s(
   const char *name,
   const char *value
);
errno_t _wputenv_s(
   const wchar_t *name,
   const wchar_t *value
);
```

Parameters

name

Environment variable name.

value

Value to set the environment variable to.

Return Value

Return 0 if successful, or an error code.

Error Conditions

name	***value***	**Return value**
NULL	any	**EINVAL**
any	**NULL**	**EINVAL**

If one of the error conditions occurs, these functions will invoke an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return **EINVAL** and set **errno** to **EINVAL**.

Remarks

The **_putenv_s** function adds new environment variables or modifies the values of existing environment variables. Environment variables define the environment in which a process executes (for example, the default search path for libraries to be linked with a program). **_wputenv_s** is a wide-character version of **_putenv_s**; the *envstring* argument to **_wputenv_s** is a wide-character string.

Generic-Text Routine Mappings

TCHAR.H routine	**_UNICODE & _MBCS not defined**	**_MBCS defined**	**_UNICODE defined**
_tputenv_s	**_putenv_s**	**_putenv_s**	**_wputenv_s**

name is the name of the environment variable to be added or modified and *value* is the variable's value. If *name* is already part of the environment, its value is replaced by *value*; otherwise, the new *name* variable and its *value* are added to the environment. You can remove a variable from the environment by specifying an empty string (that is, "") for *value*.

_putenv_s and **_wputenv_s** affect only the environment that is local to the current process; you cannot use them to modify the command-level environment. That is, these functions operate only on data structures accessible to the run-time library and not on the environment "segment" created for a process by the operating system. When the current process terminates, the environment reverts to the level of the calling process (in most cases, the operating-system level). However, the modified environment can be passed to any new processes created by **_spawn**, **_exec**, or **system**, and these new processes get any new items added by **_putenv_s** and **_wputenv_s**.

Do not change an environment entry directly: instead, use **_putenv_s** or **_wputenv_s** to change it. In particular, direct freeing elements of the **_environ[]** global array may lead to invalid memory being addressed.

getenv and **_putenv_s** use the global variable **_environ** to access the environment table; **_wgetenv** and **_wputenv_s** use

_wenviron. **_putenv_s** and **_wputenv_s** may change the value of **_environ** and **_wenviron**, thus invalidating the *envp* argument to **main** and the *_wenvp* argument to **wmain**. Therefore, it is safer to use **_environ** or **_wenviron** to access the environment information. For more information about the relation of **_putenv_s** and **_wputenv_s** to global variables, see _environ, _wenviron.

Note
The **_putenv_s** and **_getenv_s** families of functions are not thread-safe. **_getenv_s** could return a string pointer while **_putenv_s** is modifying the string, causing random failures. Make sure that calls to these functions are synchronized.

Requirements

Routine	**Required header**	**Compatibility**
_putenv_s	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wputenv_s	<stdlib.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

For a sample of how to use **_putenv_s**, see getenv_s, _wgetenv_s.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Process and Environment Control
getenv, _wgetenv
_searchenv, _wsearchenv

puts, _putws

Write a string to **stdout**.

```
int puts(
   const char *str
);
int _putws(
   const wchar_t *str
);
```

Parameters

str

Output string.

Return Value

Returns a nonnegative value if successful. If **puts** fails, it returns **EOF**; if **_putws** fails, it returns **WEOF**. If *str* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions set **errno** to **EINVAL** and return **EOF** or **WEOF**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **puts** function writes *str* to the standard output stream **stdout**, replacing the string's terminating null character ('\0') with a newline character ('\n') in the output stream.

_putws is the wide-character version of **puts**; the two functions behave identically if the stream is opened in ANSI mode. **puts** doesn't currently support output into a UNICODE stream.

Under Windows NT, **_putwch** writes Unicode characters using the current CONSOLE LOCALE setting.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_putts	**puts**	**puts**	**_putws**

Requirements

Routine	Required header	Compatibility
puts	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_putws	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_puts.c
/* This program uses puts to write a string to stdout.
 */

#include <stdio.h>
```

```
int main( void )
{
   puts( "Hello world from puts!" );
}
```

Output

```
Hello world from puts!
```

.NET Framework Equivalent

System::Console::Write

See Also

Reference

Stream I/O
fputs, fputws
fgets, fgetws

putw

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _putw instead.

_putw

Writes an integer to a stream.

```
int _putw(
   int binint,
   FILE *stream
);
```

Parameters

binint

Binary integer to be output.

stream

Pointer to the **FILE** structure.

Return Value

Returns the value written. A return value of **EOF** might indicate an error. Because **EOF** is also a legitimate integer value, use **ferror** to verify an error. If *stream* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EOF**.

For information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_putw** function writes a binary value of type **int** to the current position of *stream*. **_putw** does not affect the alignment of items in the stream nor does it assume any special alignment. **_putw** is primarily for compatibility with previous libraries. Portability problems might occur with **_putw** because the size of an **int** and the ordering of bytes within an **int** differ across systems.

Requirements

Routine	Required header	Compatibility
_putw	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_putw.c
/* This program uses _putw to write a
 * word to a stream, then performs an error check.
 */

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   FILE *stream;
   unsigned u;
   if( fopen_s( &stream, "data.out", "wb" ) )
      exit( 1 );
   for( u = 0; u < 10; u++ )
   {
```

```
        _putw( u + 0x2132, stream );   /* Write word to stream. */
        if( ferror( stream ) )         /* Make error check. */
        {
           printf( "_putw failed" );
           clearerr_s( stream );
           exit( 1 );
        }
     }
     printf( "Wrote ten words\n" );
     fclose( stream );
  }
```

Output

```
Wrote ten words
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O

_getw

_query_new_handler

Returns the address of the current new handler routine.

```
_PNH _query_new_handler(
   void
);
```

Return Value

Returns the address of the current new handler routine as set by **_set_new_handler**.

Remarks

The C++ **_query_new_handler** function returns the address of the current exception-handling function set by the C++ _set_new_handler function. **_set_new_handler** is used to specify an exception-handling function that is to gain control if the **new** operator fails to allocate memory. For more information, see the discussions of the operator new and operator delete functions in *C++ Language Reference*.

Requirements

Routine	Required header	Compatibility
_query_new_handler	<new.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation

free

_query_new_mode

Returns an integer indicating the new handler mode set by **_set_new_mode** for **malloc**.

```
int _query_new_mode(
   void
);
```

Return Value

Returns the current new handler mode, namely 0 or 1, for **malloc**. A return value of 1 indicates that, on failure to allocate memory, **malloc** calls the new handler routine; a return value of 0 indicates that it does not.

Remarks

The C++ **_query_new_mode** function returns an integer that indicates the new handler mode that is set by the C++ _set_new_mode function for malloc. The new handler mode indicates whether, on failure to allocate memory, **malloc** is to call the new handler routine as set by _set_new_handler. By default, **malloc** does not call the new handler routine on failure. You can use **_set_new_mode** to override this behavior so that on failure **malloc** calls the new handler routine in the same way that the **new** operator does when it fails to allocate memory. For more information, see the operator delete and operator new functions in *C++ Language Reference*.

Requirements

Routine	Required header	Compatibility
_query_new_mode	<new.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation
calloc
free
realloc
_query_new_handler

qsort

Performs a quick sort. A more secure version is available, see qsort_s.

```
void qsort(
   void *base,
   size_t num,
   size_t width,
   int (__cdecl *compare )(const void *, const void *)
);
```

Parameters

base

Start of target array.

num

Array size in elements.

width

Element size in bytes.

compare

Comparison function. The first parameter is a pointer to the key for the search and the second parameter is a pointer to the array element to be compared with the key.

Remarks

The **qsort** function implements a quick-sort algorithm to sort an array of *num* elements, each of *width* bytes. The argument *base* is a pointer to the base of the array to be sorted. **qsort** overwrites this array with the sorted elements. The argument *compare* is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. **qsort** calls the *compare* routine one or more times during the sort, passing pointers to two array elements on each call:

```
compare( (void *) & elem1, (void *) & elem2 );
```

The routine must compare the elements and then return one of the following values:

Compare function return value	Description
< 0	*elem1* less than *elem2*
0	*elem1* equivalent to *elem2*
> 0	*elem1* greater than *elem2*

The array is sorted in increasing order, as defined by the comparison function. To sort an array in decreasing order, reverse the sense of "greater than" and "less than" in the comparison function.

This function validates its parameters. If *compare* or *num* is **NULL**, or if *base* is **NULL** and **num* is nonzero, or if *width* is less than zero, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns and **errno** is set to **EINVAL**.

Requirements

Routine	Required header	Compatibility
qsort	<stdlib.h> and <search.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_qsort.c
// arguments: every good boy deserves favor

/* This program reads the command-line
 * parameters and uses qsort to sort them. It
 * then displays the sorted arguments.
 */

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int compare( const void *arg1, const void *arg2 );

int main( int argc, char **argv )
{
   int i;
   /* Eliminate argv[0] from sort: */
   argv++;
   argc--;

   /* Sort remaining args using Quicksort algorithm: */
   qsort( (void *)argv, (size_t)argc, sizeof( char * ), compare );

   /* Output sorted list: */
   for( i = 0; i < argc; ++i )
      printf( " %s", argv[i] );
   printf( "\n" );
}

int compare( const void *arg1, const void *arg2 )
{
   /* Compare all of both strings: */
   return _stricmp( * ( char** ) arg1, * ( char** ) arg2 );
}
```

Output

```
boy deserves every favor good
```

.NET Framework Equivalent

System::Collections::ArrayList::Sort

See Also

Reference

Searching and Sorting

bsearch

_lsearch

qsort_s

Performs a quick sort. A version of qsort with security enhancements as described in Security Enhancements in the CRT.

```
void qsort_s(
   void *base,
   size_t num,
   size_t width,
   int (__cdecl *compare )(void *, const void *, const void *),
   void * context
);
```

Parameters

base

Start of target array.

num

Array size in elements.

width

Element size in bytes.

compare

Comparison function. The first argument is the *context* pointer. The second argument is a pointer to the *key* for the search. The third argument is a pointer to the array element to be compared with *key*.

context

A pointer to a context, which can be any object that the *compare* routine needs to access.

Remarks

The **qsort_s** function implements a quick-sort algorithm to sort an array of *num* elements, each of *width* bytes. The argument *base* is a pointer to the base of the array to be sorted. **qsort_s** overwrites this array with the sorted elements. The argument *compare* is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. **qsort_s** calls the *compare* routine one or more times during the sort, passing pointers to two array elements on each call:

```
compare( context, (void *) & elem1, (void *) & elem2 );
```

The routine must compare the elements and then return one of the following values:

Return value	Description
< 0	*elem1* less than *elem2*
0	*elem1* equivalent to *elem2*
> 0	*elem1* greater than *elem2*

The array is sorted in increasing order, as defined by the comparison function. To sort an array in decreasing order, reverse the sense of "greater than" and "less than" in the comparison function.

If invalid parameters are passed to the function, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, then the function returns and **errno** is set to **EINVAL**. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Error Conditions

key	base	compare	num	width	errno

NULL	any	any	any	any	**EINVAL**
any	**NULL**	any	!= 0	any	**EINVAL**
any	any	any	any	<= 0	**EINVAL**
any	any	**NULL**	any	any	**EINVAL**

qsort_s has the same behavior as **qsort** but has the *context* parameter and sets **errno**. By passing a *context* parameter, comparison functions can use an object pointer to access object functionality or other information not accessible through an element pointer. The addition of the *context* parameter makes **qsort_s** more secure because *context* can be used to avoid reentrancy bugs introduced by using static variables to make shared information available to the *compare* function.

Requirements

Routine	Required header	Compatibility
qsort_s	<stdlib.h> and <search.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries: All versions of the C Run-Time Libraries.

Example

The following example demonstrates how to use the *context* parameter in the **qsort_s** function. The *context* parameter makes it easier to perform thread-safe sorts. Instead of using static variables that must be synchronized to ensure thread safety, pass a different *context* parameter in each sort. In this example, a locale object is used as the *context* parameter.

```
// crt_qsort_s.cpp
// compile with: /EHsc /MT
#include <stdlib.h>
#include <stdio.h>
#include <search.h>
#include <process.h>
#include <locale.h>
#include <locale>
#include <windows.h>
using namespace std;

// The sort order is dependent on the code page.  Use 'chcp' at the
// command line to change the codepage.  When executing this application,
// the command prompt codepage must match the codepage used here:

#define CODEPAGE_850

#ifdef CODEPAGE_850
// Codepage 850 is the OEM codepage used by the command line,
// so \x00e1 is the German Sharp S in that codepage and \x00a4
// is the n tilde.

char *array1[] = { "wei\x00e1", "weis", "annehmen", "weizen", "Zeit",
                   "weit" };
char *array2[] = { "Espa\x00a4ol", "Espa\x00a4" "a", "espantado" };
char *array3[] = { "table", "tableux", "tablet" };

#define GERMAN_LOCALE "German_Germany.850"
#define SPANISH_LOCALE "Spanish_Spain.850"
#define ENGLISH_LOCALE "English_US.850"

#endif

#ifdef CODEPAGE_1252
```

```
   // If using codepage 1252 (ISO 8859-1, Latin-1), use \x00df
   // for the German Sharp S and \x001f for the n tilde.
char *array1[] = { "wei\x00df", "weis", "annehmen", "weizen", "Zeit",
                   "weit" };
char *array2[] = { "Espa\x00f1ol", "Espa\x00f1" "a", "espantado" };
char *array3[] = { "table", "tableux", "tablet" };

#define GERMAN_LOCALE "German_Germany.1252"
#define SPANISH_LOCALE "Spanish_Spain.1252"
#define ENGLISH_LOCALE "English_US.1252"

#endif

// The context parameter lets you create a more generic compare.
// Without this parameter, you would have stored the locale in a
// static variable, thus making sort_array vulnerable to thread
// conflicts.

int compare( void *pvlocale, const void *str1, const void *str2)
{
    char s1[256];
    char s2[256];
    strcpy_s(s1, 256, *(char**)str1);
    strcpy_s(s2, 256, *(char**)str2);
    _strlwr_s( s1, sizeof(s1) );
    _strlwr_s( s2, sizeof(s2) );

    locale& loc = *( reinterpret_cast< locale * > ( pvlocale));

    return use_facet< collate<char> >(loc).compare(s1,
       &s1[strlen(s1)], s2, &s2[strlen(s2)]);

}

void sort_array(char *array[], int num, locale &loc)
{
    qsort_s(array, num, sizeof(char*), compare, &loc);
}

void print_array(char *a[], int c)
{
   for (int i = 0; i < c; i++)
     printf("%s ", a[i]);
   printf("\n");

}

void sort_german(void * Dummy)
{
   sort_array(array1, 6, locale(GERMAN_LOCALE));
}

void sort_spanish(void * Dummy)
{
   sort_array(array2, 3, locale(SPANISH_LOCALE));
}

void sort_english(void * Dummy)
{
   sort_array(array3, 3, locale(ENGLISH_LOCALE));
}

int main( )
{

   int i;
   HANDLE threads[3];
```

```
    printf("Unsorted input:\n");
    print_array(array1, 6);
    print_array(array2, 3);
    print_array(array3, 3);


    // Create several threads that perform sorts in different
    // languages at the same time.

    threads[0] = reinterpret_cast<HANDLE>(
                  _beginthread( sort_german , 0, NULL));
    threads[1] = reinterpret_cast<HANDLE>(
                  _beginthread( sort_spanish, 0, NULL));
    threads[2] = reinterpret_cast<HANDLE>(
                  _beginthread( sort_english, 0, NULL));

    for (i = 0; i < 3; i++)
    {
       if (threads[i] == reinterpret_cast<HANDLE>(-1))
       {
          printf("Error creating threads.\n");
          exit(1);
       }
    }

    // Wait until all threads have terminated.
    WaitForMultipleObjects(3, threads, true, INFINITE);

    printf("Sorted output: \n");

    print_array(array1, 6);
    print_array(array2, 3);
    print_array(array3, 3);



}
```

Sample Output

```
Unsorted input:
weiß weis annehmen weizen Zeit weit
Español España espantado
table tableux tablet
Sorted output:
annehmen weiß weis weit weizen Zeit
España Español espantado
table tablet tableux
```

.NET Framework Equivalent

Sort

See Also

Reference

Searching and Sorting
bsearch_s
_lsearch_s
qsort

raise

Sends a signal to the executing program.

```
int raise(
   int sig
);
```

Parameters

sig

Signal to be raised.

Return Value

If successful, **raise** returns 0. Otherwise, it returns a nonzero value.

Remarks

The **raise** function sends *sig* to the executing program. If a previous call to **signal** has installed a signal-handling function for *sig*, **raise** executes that function. If no handler function has been installed, the default action associated with the signal value *sig* is taken, as follows.

Signal	Meaning	Default
SIGABRT	Abnormal termination	Terminates the calling program with exit code 3
SIGFPE	Floating-point error	Terminates the calling program
SIGILL	Illegal instruction	Terminates the calling program
SIGINT	CTRL+C interrupt	Terminates the calling program
SIGSEGV	Illegal storage access	Terminates the calling program
SIGTERM	Termination request sent to the program	Ignores the signal

If the argument is not a valid signal as specified above, the invalid parameter handler is invoked, as described in Parameter Validation. If not handled, the function sets **errno** to **EINVAL** and returns a nonzero value.

Requirements

Routine	Required header	Compatibility
raise	<signal.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Process and Environment Control

abort
signal

rand

Generates a pseudorandom number. A more secure version is available, see rand_s.

```
int rand( void );
```

Return Value

rand returns a pseudorandom number, as described above. There is no error return.

Remarks

The **rand** function returns a pseudorandom integer in the range 0 to **RAND_MAX** (32767). Use the srand function to seed the pseudorandom-number generator before calling **rand**.

Requirements

Routine	Required header	Compatibility
rand	<stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_rand.c
// This program seeds the random-number generator
// with the time, then exercises the rand function.
//

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void SimpleRandDemo( int n )
{
   // Print n random numbers.
   int i;
   for( i = 0; i < n; i++ )
      printf( "  %6d\n", rand() );
}

void RangedRandDemo( int range_min, int range_max, int n )
{
   // Generate random numbers in the half-closed interval
   // [range_min, range_max). In other words,
   // range_min <= random number < range_max
   int i;
   for ( i = 0; i < n; i++ )
   {
      int u = (double)rand() / (RAND_MAX + 1) * (range_max - range_min)
            + range_min;
      printf( "  %6d\n", u);
   }
}

int main( void )
{
   // Seed the random-number generator with the current time so that
   // the numbers will be different every time we run.
   srand( (unsigned)time( NULL ) );

   SimpleRandDemo( 10 );
```

```
    printf("\n");
    RangedRandDemo( -100, 100, 10 );
}
```

Sample Output

```
22036
18330
11651
27464
18093
 3284
11785
14686
11447
11285

   74
   48
   27
   65
   96
   64
   -5
  -42
  -55
   66
```

.NET Framework Equivalent

System::Random Class

See Also

Reference

Floating-Point Support

srand

rand_s

rand_s

Generates a pseudorandom number. A version of rand with security enhancements as described in Security Enhancements in the CRT.

```
errno_t rand_s(   unsigned int* randomValue);
```

Return Value

Zero if successful, otherwise, an error code. If the input pointer *randomValue* is a null pointer, the function invokes an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the function returns **EINVAL** and sets **errno** to **EINVAL**.

Remarks

The **rand_s** function writes a pseudorandom integer in the range 0 to **UINT_MAX** to the input pointer. The **rand_s** function uses the operating system to generate cryptographically secure random numbers. It does not use the seed generated by the srand function, nor does it affect the random number sequence used by **rand**.

The **rand_s** function requires that constant **_CRT_RAND_S** be defined prior to the inclusion statement for the function to be declared, as in the following example:

```
#define _CRT_RAND_S
#include <stdlib.h>
```

rand_s depends on the RtlGenRandom API, which is only available in Windows XP and later.

Requirements

Routine	Required header	Compatibility
rand_s	<stdlib.h>	Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more information, see Compatibility.

Example

```
// crt_rand_s.c
// This program illustrates how to generate random
// integer or floating point numbers in a specified range.

// Remembering to define _CRT_RAND_S prior
// to inclusion statement.
#define _CRT_RAND_S

#include <stdlib.h>
#include <stdio.h>
#include <limits.h>

int main( void )
{
    int             i;
    unsigned int    number;
    double          max = 100.0;
    errno_t         err;

    // Display 10 random integers in the range [ 1,10 ].
    for( i = 0; i < 10;i++ )
    {
        err = rand_s( &number );
        if (err != 0)
        {
            printf_s("The rand_s function failed!\n");
        }
```

```
        printf_s( "  %u\n", (unsigned int) ((double)number /
                            (double) UINT_MAX * 10.0) + 1);
    }

    printf_s("\n");

    // Display 10 random doubles between 0 and max.
    for (i = 0; i < 10;i++ )
    {
        err = rand_s( &number );
        if (err != 0)
        {
            printf_s("The rand_s function failed!\n");
        }
        printf_s( "  %g\n", (double) number /
                            (double) UINT_MAX * max );
    }
}
```

Sample Output

```
  10
  4
  5
  2
  8
  2
  5
  6
  1
  1

  32.6617
  29.4471
  11.5413
  6.41924
  20.711
  60.2878
  61.0094
  20.1222
  80.9192
  65.0712
```

.NET Framework Equivalent

System::Random Class

See Also

Reference

Floating-Point Support

srand

read

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _read instead.

_read

Reads data from a file.

```
int _read(
   int fd,
   void *buffer,
   unsigned int count
);
```

Parameters

fd

File descriptor referring to the open file.

buffer

Storage location for data.

count

Maximum number of bytes.

Return Value

_read returns the number of bytes read, which might be less than *count* if there are fewer than *count* bytes left in the file or if the file was opened in text mode, in which case each carriage return–line feed (CR-LF) pair is replaced with a single linefeed character. Only the single linefeed character is counted in the return value. The replacement does not affect the file pointer.

If the function tries to read at end of file, it returns 0. If *fd* is invalid, the file is not open for reading, or the file is locked, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns –1 and sets **errno** to **EBADF**.

If *buffer* is **NULL**, the invalid parameter handler is invoked. If execution is allowed to continue, the function returns -1 and **errno** is set to **EINVAL**.

For more information about this and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_read** function reads a maximum of *count* bytes into *buffer* from the file associated with *fd*. The read operation begins at the current position of the file pointer associated with the given file. After the read operation, the file pointer points to the next unread character.

If the file was opened in text mode, the read terminates when **_read** encounters a CTRL+Z character, which is treated as an end-of-file indicator. Use _lseek to clear the end-of-file indicator.

Requirements

Routine	Required header	Compatibility
_read	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_read.c
/* This program opens a file named crt_read.txt
 * and tries to read 60,000 bytes from
```

```
 * that file using _read. It then displays the
 * actual number of bytes read.
 */

#include <fcntl.h>      /* Needed only for _O_RDWR definition */
#include <io.h>
#include <stdlib.h>
#include <stdio.h>
#include <share.h>

char buffer[60000];

int main( void )
{
   int fh;
   unsigned int nbytes = 60000, bytesread;

   /* Open file for input: */
   if( _sopen_s( &fh, "crt_read.txt", _O_RDONLY, _SH_DENYNO, 0 ) )
   {
      perror( "open failed on input file" );
      exit( 1 );
   }

   /* Read in input: */
   if( ( bytesread = _read( fh, buffer, nbytes ) ) <= 0 )
      perror( "Problem reading file" );
   else
      printf( "Read %u bytes from file\n", bytesread );

   _close( fh );
}
```

Input: crt_read.txt

```
Line one.
Line two.
```

Output

```
Read 19 bytes from file
```

See Also

Reference

Low-Level I/O
_creat, _wcreat
fread
_open, _wopen
_write

realloc

Reallocate memory blocks.

```
void *realloc(
   void *memblock,
   size_t size
);
```

Parameters

memblock

Pointer to previously allocated memory block.

size

New size in bytes.

Return Value

realloc returns a **void** pointer to the reallocated (and possibly moved) memory block.

If there is not enough available memory to expand the block to the given size, the original block is left unchanged, and **NULL** is returned.

If *size* is zero, then the block pointed to by *memblock* is freed; the return value is **NULL**, and *memblock* is left pointing at a freed block.

The return value points to a storage space that is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than **void**, use a type cast on the return value.

Remarks

The **realloc** function changes the size of an allocated memory block. The *memblock* argument points to the beginning of the memory block. If *memblock* is **NULL**, **realloc** behaves the same way as **malloc** and allocates a new block of *size* bytes. If *memblock* is not **NULL**, it should be a pointer returned by a previous call to **calloc**, **malloc**, or **realloc**.

The *size* argument gives the new size of the block, in bytes. The contents of the block are unchanged up to the shorter of the new and old sizes, although the new block can be in a different location. Because the new block can be in a new memory location, the pointer returned by **realloc** is not guaranteed to be the pointer passed through the *memblock* argument. **realloc** does not zero newly allocated memory in the case of buffer growth.

In Visual C++ 2005, **realloc** sets **errno** to **ENOMEM** if the memory allocation fails or if the amount of memory requested exceeds **_HEAP_MAXREQ**. For information on this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

realloc calls **malloc** in order to use the C++ _set_new_mode function to set the new handler mode. The new handler mode indicates whether, on failure, **malloc** is to call the new handler routine as set by _set_new_handler. By default, **malloc** does not call the new handler routine on failure to allocate memory. You can override this default behavior so that, when **realloc** fails to allocate memory, **malloc** calls the new handler routine in the same way that the **new** operator does when it fails for the same reason. To override the default, call

```
_set_new_mode(1)
```

early in ones program, or link with NEWMODE.OBJ (see Link Options).

When the application is linked with a debug version of the C run-time libraries, **realloc** resolves to _realloc_dbg. For more information about how the heap is managed during the debugging process, see The CRT Debug Heap.

realloc is marked **__declspec(noalias)** and **__declspec(restrict)**, meaning that the function is guaranteed not to modify global variables, and that the pointer returned is not aliased. For more information, see noalias and restrict.

Requirements

Routine	Required header	Compatibility

realloc	<stdlib.h> and <malloc.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_realloc.c
// This program allocates a block of memory for
// buffer and then uses _msize to display the size of that
// block. Next, it uses realloc to expand the amount of
// memory used by buffer and then calls _msize again to
// display the new amount of memory allocated to buffer.


#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

int main( void )
{
   long *buffer, *oldbuffer;
   size_t size;

   if( (buffer = (long *)malloc( 1000 * sizeof( long ) )) == NULL )
      exit( 1 );

   size = _msize( buffer );
   printf_s( "Size of block after malloc of 1000 longs: %u\n", size );

   // Reallocate and show new size:
   oldbuffer = buffer;     // save pointer in case realloc fails
   if( (buffer = realloc( buffer, size + (1000 * sizeof( long )) ))
        ==  NULL )
   {
      free( oldbuffer );  // free original block
      exit( 1 );
   }
   size = _msize( buffer );
   printf_s( "Size of block after realloc of 1000 more longs: %u\n",
            size );

   free( buffer );
   exit( 0 );
}
```

Output

```
Size of block after malloc of 1000 longs: 4000
Size of block after realloc of 1000 more longs: 8000
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation
calloc
free
malloc

_realloc_dbg

Reallocates a specified block of memory in the heap by moving and/or resizing the block (debug version only).

```
void *_realloc_dbg(
   void *userData,
   size_t newSize,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

userData

Pointer to the previously allocated memory block.

newSize

Requested size for the reallocated block (bytes).

blockType

Requested type for the reallocated block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

filename

Pointer to the name of the source file that requested the **realloc** operation or NULL.

linenumber

Line number in the source file where the **realloc** operation was requested or NULL.

The *filename* and *linenumber* parameters are only available when **_realloc_dbg** has been called explicitly or the _CRTDBG_MAP_ALLOC preprocessor constant has been defined.

Return Value

On successful completion, this function either returns a pointer to the user portion of the reallocated memory block, calls the new handler function, or returns NULL. For a complete description of the return behavior, see the following Remarks section. For more information about how the new handler function is used, see the realloc function.

Remarks

_realloc_dbg is a debug version of the realloc function. When _DEBUG is not defined, each call to **_realloc_dbg** is reduced to a call to **realloc**. Both **realloc** and **_realloc_dbg** reallocate a memory block in the base heap, but **_realloc_dbg** accommodates several debugging features: buffers on either side of the user portion of the block to test for leaks, a block type parameter to track specific allocation types, and *filename*/*linenumber* information to determine the origin of allocation requests.

_realloc_dbg reallocates the specified memory block with slightly more space than the requested *newSize*. *newSize* might be greater or less than the size of the originally allocated memory block. The additional space is used by the debug heap manager to link the debug memory blocks and to provide the application with debug header information and overwrite buffers. The reallocation might result in moving the original memory block to a different location in the heap, as well as changing the size of the memory block. If the memory block is moved, the contents of the original block are overwritten.

In Visual C++ 2005, **_realloc_dbg** sets **errno** to **ENOMEM** if a memory allocation fails or if the amount of memory needed (including the overhead mentioned previously) exceeds **_HEAP_MAXREQ**. For information about this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see Memory Management and the Debug Heap. For information about the allocation block types and how they are used, see Types of Blocks on the Debug Heap. For information about the differences between calling a standard heap function and its debug version in a debug build of an application, see Using the Debug Version Versus the Base Version.

Requirements

Routine	Required header	Compatibility
_realloc_dbg	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Example

See the example in the _msize_dbg topic.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_malloc_dbg

_recalloc

A combination of **realloc** and **calloc**. Reallocates an array in memory and initializes its elements to 0.

```
void *_recalloc(
   void *memblock
   size_t num,
   size_t size
);
```

Parameters

memblock

Pointer to previously allocated memory block.

num

Number of elements.

size

Length in bytes of each element.

Return Value

_recalloc returns a **void** pointer to the reallocated (and possibly moved) memory block.

If there is not enough available memory to expand the block to the given size, the original block is left unchanged, and **NULL** is returned.

If the requested size is zero, then the block pointed to by *memblock* is freed; the return value is **NULL**, and *memblock* is left pointing at a freed block.

The return value points to a storage space that is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than **void**, use a type cast on the return value.

Remarks

The **_recalloc** function changes the size of an allocated memory block. The *memblock* argument points to the beginning of the memory block. If *memblock* is **NULL**, **_recalloc** behaves the same way as calloc and allocates a new block of *num* * *size* bytes. Each element is initialized to 0. If *memblock* is not **NULL**, it should be a pointer returned by a previous call to **calloc**, malloc, or realloc.

Because the new block can be in a new memory location, the pointer returned by **_recalloc** is not guaranteed to be the pointer passed through the *memblock* argument.

In Visual C++ 2005, **_recalloc** sets **errno** to **ENOMEM** if the memory allocation fails or if the amount of memory requested exceeds **_HEAP_MAXREQ**. For information on this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

recalloc calls **realloc** in order to use the C++ _set_new_mode function to set the new handler mode. The new handler mode indicates whether, on failure, **realloc** is to call the new handler routine as set by _set_new_handler. By default, **realloc** does not call the new handler routine on failure to allocate memory. You can override this default behavior so that, when **_recalloc** fails to allocate memory, **realloc** calls the new handler routine in the same way that the **new** operator does when it fails for the same reason. To override the default, call

```
_set_new_mode(1)
```

early in the program, or link with NEWMODE.OBJ.

When the application is linked with a debug version of the C run-time libraries, **_recalloc** resolves to _recalloc_dbg. For more information about how the heap is managed during the debugging process, see The CRT Debug Heap.

_recalloc is marked **__declspec(noalias)** and **__declspec(restrict)**, meaning that the function is guaranteed not to modify global variables, and that the pointer returned is not aliased. For more information, see noalias and restrict.

Requirements

Routine	Required header	Compatibility
_recalloc	<stdlib.h> and <malloc.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation
_recalloc_dbg
_aligned_recalloc
_aligned_offset_recalloc
free

Concepts

Link Options

_recalloc_dbg

Reallocates an array and initializes its elements to 0 (debug version only).

```
void *_recalloc_dbg(
   void *userData,
   size_t num,
   size_t size,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

userData

Pointer to the previously allocated memory block.

num

Requested number of memory blocks.

size

Requested size of each memory block (bytes).

blockType

Requested type of memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

For information about the allocation block types and how they are used, see Types of Blocks on the Debug Heap.

filename

Pointer to name of the source file that requested allocation operation or **NULL**.

linenumber

Line number in the source file where allocation operation was requested or **NULL**.

The *filename* and *linenumber* parameters are only available when **_recalloc_dbg** has been called explicitly or the _CRTDBG_MAP_ALLOC preprocessor constant has been defined.

Return Value

On successful completion, this function either returns a pointer to the user portion of the reallocated memory block, calls the new handler function, or returns NULL. For a complete description of the return behavior, see the following Remarks section. For more information about how the new handler function is used, see the _recalloc function.

Remarks

_recalloc_dbg is a debug version of the _recalloc function. When _DEBUG is not defined, each call to **_recalloc_dbg** is reduced to a call to **_recalloc**. Both **_recalloc** and **_recalloc_dbg** reallocate a memory block in the base heap, but **_recalloc_dbg** accommodates several debugging features: buffers on either side of the user portion of the block to test for leaks, a block type parameter to track specific allocation types, and *filename/linenumber* information to determine the origin of allocation requests.

_recalloc_dbg reallocates the specified memory block with slightly more space than the requested size (*num* * *size*) which might be greater or less than the size of the originally allocated memory block. The additional space is used by the debug heap manager to link the debug memory blocks and to provide the application with debug header information and overwrite buffers. The reallocation might result in moving the original memory block to a different location in the heap, as well as changing the size of the memory block. The user portion of the block is filled with the value 0xCD and each of the overwrite buffers are filled with 0xFD.

In Visual C++ 2005, **_recalloc_dbg** sets **errno** to **ENOMEM** if a memory allocation fails; **EINVAL** is returned if the amount of memory needed (including the overhead mentioned previously) exceeds **_HEAP_MAXREQ**. For information about this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see Memory Management and the Debug Heap. For information about the differences between calling a standard heap function versus its debug version in a debug build of an application, see Using the Debug Version Versus the Base Version.

Requirements

Routine	Required header	Compatibility
_recalloc_dbg	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

remove, _wremove

Delete a file.

```
int remove(
   const char *path
);
int _wremove(
   const wchar_t *path
);
```

Parameters

path

Path of file to be removed.

Return Value

Each of these functions returns 0 if the file is successfully deleted. Otherwise, it returns -1 and sets **errno** either to **EACCES** to indicate that the path specifies a read-only file or the file is open, or to **ENOENT** to indicate that the filename or path was not found or that the path specifies a directory.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these and other return codes.

Remarks

The **remove** function deletes the file specified by *path*. **_wremove** is a wide-character version of **_remove**; the *path* argument to **_wremove** is a wide-character string. **_wremove** and **_remove** behave identically otherwise. All handles to a file must be closed before it can be deleted.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tremove	**remove**	**remove**	**_wremove**

Requirements

Routine	Required header	Compatibility
remove	<stdio.h> or <io.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wremove	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_remove.c
/* This program uses remove to delete crt_remove.txt */

#include <stdio.h>

int main( void )
{
   if( remove( "crt_remove.txt" ) == -1 )
      perror( "Could not delete 'CRT_REMOVE.TXT'" );
   else
```

```
        printf( "Deleted 'CRT_REMOVE.TXT'\n" );
}
```

Input: crt_remove.txt

```
This file will be deleted.
```

Sample Output

```
Deleted 'CRT_REMOVE.TXT'
```

.NET Framework Equivalent

System::IO::File::Delete

See Also

Reference

File Handling

_unlink, _wunlink

rename, _wrename

Rename a file or directory.

```
int rename(
   const char *oldname,
   const char *newname
);
int _wrename(
   const wchar_t *oldname,
   const wchar_t *newname
);
```

Parameters

oldname

Pointer to old name.

newname

Pointer to new name.

Return Value

Each of these functions returns 0 if it is successful. On an error, the function returns a nonzero value and sets **errno** to one of the following values:

EACCES

File or directory specified by *newname* already exists or could not be created (invalid path); or *oldname* is a directory and *newname* specifies a different path.

ENOENT

File or path specified by *oldname* not found.

EINVAL

Name contains invalid characters.

For other possible return values, see _doserrno, _errno, syserrlist, and _sys_nerr.

Remarks

The **rename** function renames the file or directory specified by *oldname* to the name given by *newname*. The old name must be the path of an existing file or directory. The new name must not be the name of an existing file or directory. You can use **rename** to move a file from one directory or device to another by giving a different path in the *newname* argument. However, you cannot use **rename** to move a directory. Directories can be renamed, but not moved.

_wrename is a wide-character version of **_rename**; the arguments to **_wrename** are wide-character strings. **_wrename** and **_rename** behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_trename	**rename**	**rename**	**_wrename**

Requirements

Routine	Required header	Compatibility
rename	<io.h> or <stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_wrename	<stdio.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_renamer.c
/* This program attempts to rename a file named
 * CRT_RENAMER.OBJ to CRT_RENAMER.JBO. For this operation
 * to succeed, a file named CRT_RENAMER.OBJ must exist and
 * a file named CRT_RENAMER.JBO must not exist.
 */

#include <stdio.h>

int main( void )
{
   int  result;
   char old[] = "CRT_RENAMER.OBJ", new[] = "CRT_RENAMER.JBO";

   /* Attempt to rename file: */
   result = rename( old, new );
   if( result != 0 )
      printf( "Could not rename '%s'\n", old );
   else
      printf( "File '%s' renamed to '%s'\n", old, new );
}
```

Output

```
File 'CRT_RENAMER.OBJ' renamed to 'CRT_RENAMER.JBO'
```

.NET Framework Equivalent

System::IO::File::Move

See Also

Reference

File Handling

_resetstkoflw

Recovers from stack overflow.

```
int _resetstkoflw ( void );
```

Return Value

Nonzero if the function succeeds, zero if it fails.

Remarks

The **_resetstkoflw** function recovers from a stack overflow condition, allowing a program to continue instead of failing with a fatal exception error. If the **_resetstkoflw** function is not called, there are no guard page after the previous exception. The next time that there is a stack overflow, there are no exception at all and the process terminates without warning.

If a thread in an application causes an **EXCEPTION_STACK_OVERFLOW** exception, the thread has left its stack in a damaged state. This is in contrast to other exceptions such as **EXCEPTION_ACCESS_VIOLATION** or **EXCEPTION_INT_DIVIDE_BY_ZERO**, where the stack is not damaged. The stack is set to an arbitrarily small value when the program is first loaded. The stack then grows on demand to meet the needs of the thread. This is implemented by placing a page with PAGE_GUARD access at the end of the current stack. For more information, see Creating Guard Pages.

When the code causes the stack pointer to point to an address on this page, an exception occurs and the system does the following three things:

- Removes the PAGE_GUARD protection on the guard page so that the thread can read and write data to the memory.
- Allocates a new guard page that is located one page below the last one.
- Reruns the instruction that raised the exception.

In this way, the system can increase the size of the stack for the thread automatically. Each thread in a process has a maximum stack size. The stack size is set at compile time by the /STACK (Stack Allocations), or by the STACKSIZE statement in the .def file for the project.

When this maximum stack size is exceeded, the system does the following three things:

- Removes the PAGE_GUARD protection on the guard page, as previously described.
- Tries to allocate a new guard page below the last one. However, this fails because the maximum stack size has been exceeded.
- Raises an exception so that the thread can handle it in the exception block.

Note that, at that point, the stack no longer has a guard page. The next time that the program grows the stack all the way to the end, where there should be a guard page, the program writes beyond the end of the stack and causes an access violation.

Call **_resetstkoflw** to restore the guard page whenever recovery is done after a stack overflow exception. This function can be called from inside the main body of an **__except** block or outside an **__except** block. However, there are some restrictions on when it should be used. **_resetstkoflw** should never be called from:

- A filter expression.
- A filter function.
- A function called from a filter function.
- A **catch** block.
- A **__finally** block.

At these points, the stack is not yet sufficiently unwound.

Stack overflow exceptions are generated as structured exceptions, not C++ exceptions, so **_resetstkoflw** is not useful in an ordinary **catch** block because it will not catch a stack overflow exception. However, if _set_se_translator is used to implement a structured exception translator that throws C++ exceptions (as in the second example), a stack overflow exception results in a

C++ exception that can be handled by a C++ catch block.

It is not safe to call **_resetstkoflw** in a C++ catch block that is reached from an exception thrown by the structured exception translator function. In this case, the stack space is not freed and the stack pointer is not reset until outside the catch block, even though destructors have been called for any destructible objects before the catch block. This function should not be called until the stack space is freed and the stack pointer has been reset. Therefore, it should be called only after exiting the catch block. As little stack space as possible should be used in the catch block because a stack overflow that occurs in the catch block that is itself attempting to recover from a previous stack overflow is not recoverable and can cause the program to stop responding as the overflow in the catch block triggers an exception that itself is handled by the same catch block.

There are situations where **_resetstkoflw** can fail even if used in a correct location, such as within an **__except** block. If, even after unwinding the stack, there is still not enough stack space left to execute **_resetstkoflw** without writing into the last page of the stack, **_resetstkoflw** fails to reset the last page of the stack as the guard page and returns 0, indicating failure. Therefore, safe usage of this function should include checking the return value instead of assuming that the stack is safe to use.

Structured exception handling will not catch a **STATUS_STACK_OVERFLOW** exception when the application is compiled with **/clr** or **/clr:pure** (see /clr (Common Language Runtime Compilation)).

Requirements

Routine	Required header	Compatibility
_resetstkoflw	<malloc.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries: All versions of the C Run-Time Libraries.

Example

The following example shows the recommended usage of the **_resetstkoflw** function.

```
// crt_resetstkoflw.c
// Launch program with and without arguments to observe
// the difference made by calling _resetstkoflw.

#include <malloc.h>
#include <stdio.h>
#include <windows.h>

void recursive(int recurse)
{
   _alloca(2000);
   if (recurse)
      recursive(recurse);
}

// Filter for the stack overflow exception.
// This function traps the stack overflow exception, but passes
// all other exceptions through.
int stack_overflow_exception_filter(int exception_code)
{
   if (exception_code == EXCEPTION_STACK_OVERFLOW)
   {
       // Do not call _resetstkoflw here, because
       // at this point, the stack is not yet unwound.
       // Instead, signal that the handler (the __except block)
       // is to be executed.
       return EXCEPTION_EXECUTE_HANDLER;
   }
   else
       return EXCEPTION_CONTINUE_SEARCH;
}

int main(int ac)
```

```
{
   int i = 0;
   int recurse = 1, result = 0;

   for (i = 0 ; i < 10 ; i++)
   {
      printf("loop #%d\n", i + 1);
      __try
      {
         recursive(recurse);

      }

      __except(stack_overflow_exception_filter(GetExceptionCode()))
      {
         // Here, it is safe to reset the stack.

         if (ac >= 2)
         {
            puts("resetting stack overflow");
            result = _resetstkoflw();
         }
      }

      // Terminate if _resetstkoflw failed (returned 0)
      if (!result)
         return 3;
   }

   return 0;
}
```

Sample Output

With no program arguments:

```
loop #1
```

The program stops responding without executing further iterations.

With program arguments:

```
loop #1
resetting stack overflow
loop #2
resetting stack overflow
loop #3
resetting stack overflow
loop #4
resetting stack overflow
loop #5
resetting stack overflow
loop #6
resetting stack overflow
loop #7
resetting stack overflow
loop #8
resetting stack overflow
loop #9
resetting stack overflow
loop #10
resetting stack overflow
```

The following example shows the recommended use of **_resetstkoflw** in a program where structured exceptions are converted to C++ exceptions.

```
// crt_resetstkoflw2.cpp
// compile with: /EHa
// _set_se_translator requires the use of /EHa
#include <malloc.h>
#include <stdio.h>
#include <windows.h>
#include <eh.h>

class Exception { };

class StackOverflowException : Exception { };

// Because the overflow is deliberate, disable the warning that
// this function will cause a stack overflow.
#pragma warning (disable: 4717)
void CauseStackOverflow (int i)
{
        // Overflow the stack by allocating a large stack-based array
        // in a recursive function.
        int a[10000];
        printf("%d ", i);
        CauseStackOverflow (i + 1);
}

void __cdecl SEHTranslator (unsigned int code, _EXCEPTION_POINTERS*)
{
   // For stack overflow exceptions, throw our own C++
   // exception object.
   // For all other exceptions, throw a generic exception object.
   // Use minimal stack space in this function.
   // Do not call _resetstkoflw in this function.

   if (code == EXCEPTION_STACK_OVERFLOW)
      throw StackOverflowException ( );
   else
      throw Exception( );
}

int main ( )
{
        bool stack_reset = false;
        bool result = false;

        // Set up a function to handle all structured exceptions,
        // including stack overflow exceptions.
        _set_se_translator (SEHTranslator);

        try
        {
            CauseStackOverflow (0);
        }
        catch (StackOverflowException except)
        {
                // Use minimal stack space here.
                // Do not call _resetstkoflw here.
                printf("\nStack overflow!\n");
                stack_reset = true;
        }
        catch (Exception except)
        {
                // Do not call _resetstkoflw here.
                printf("\nUnknown Exception!\n");
        }
        if (stack_reset)
        {
          result = _resetstkoflw();
          // If stack reset failed, terminate the application.
          if (result == 0)
```

```
            exit(1);
        }

        void* pv = _alloca(100000);
        printf("Recovered from stack overflow and allocated 100,000 bytes"
               " using _alloca.");

    return 0;
}
```

Sample Output

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
Stack overflow!
Recovered from stack overflow and allocated 100,000 bytes using _alloca.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

_alloca

rewind

Repositions the file pointer to the beginning of a file.

```
void rewind(
   FILE *stream
);
```

Parameters

stream

Pointer to **FILE** structure.

Remarks

The **rewind** function repositions the file pointer associated with *stream* to the beginning of the file. A call to **rewind** is similar to

(void) fseek(*stream*, **0L, SEEK_SET);**

However, unlike **fseek**, **rewind** clears the error indicators for the stream as well as the end-of-file indicator. Also, unlike **fseek**, **rewind** does not return a value to indicate whether the pointer was successfully moved.

To clear the keyboard buffer, use **rewind** with the stream **stdin**, which is associated with the keyboard by default.

If stream is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function returns and **errno** is set to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Requirements

Routine	Required header	Compatibility
rewind	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_rewind.c
/* This program first opens a file named
 * crt_rewind.out for input and output and writes two
 * integers to the file. Next, it uses rewind to
 * reposition the file pointer to the beginning of
 * the file and reads the data back in.
 */
#include <stdio.h>

int main( void )
{
   FILE *stream;
   int data1, data2;

   data1 = 1;
   data2 = -37;

   fopen_s( &stream, "crt_rewind.out", "w+" );
   if( stream != NULL )
```

```
    {
       fprintf( stream, "%d %d", data1, data2 );
       printf( "The values written are: %d and %d\n", data1, data2 );
       rewind( stream );
       fscanf_s( stream, "%d %d", &data1, &data2 );
       printf( "The values read are: %d and %d\n", data1, data2 );
       fclose( stream );
    }
 }
```

Output

```
The values written are: 1 and -37
The values read are: 1 and -37
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O

rmdir

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _rmdir instead.

_rmdir, _wrmdir

Deletes a directory.

```
int _rmdir(
   const char *dirname
);
int _wrmdir(
   const wchar_t *dirname
);
```

Parameters

dirname

Path of the directory to be removed.

Return Value

Each of these functions returns 0 if the directory is successfully deleted. A return value of –1 indicates an error and **errno** is set to one of the following values:

ENOTEMPTY

Given path is not a directory, the directory is not empty, or the directory is either the current working directory or the root directory.

ENOENT

Path is invalid.

EACCES

A program has an open handle to the directory.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_rmdir** function deletes the directory specified by *dirname*. The directory must be empty, and it must not be the current working directory or the root directory.

_wrmdir is a wide-character version of **_rmdir**; the *dirname* argument to **_wrmdir** is a wide-character string. **_wrmdir** and **_rmdir** behave identically otherwise.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_trmdir	**_rmdir**	**_rmdir**	**_wrmdir**

Requirements

Routine	Required header	Compatibility
_rmdir	<direct.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wrmdir	<direct.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

See the example for _mkdir.

.NET Framework Equivalent

System::IO::Directory::Delete

See Also

Reference

Directory Control
_chdir, _wchdir
_mkdir, _wmkdir

rmtmp

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _rmtmp instead.

_rmtmp

Removes temporary files.

```
int _rmtmp( void );
```

Return Value

_rmtmp returns the number of temporary files closed and deleted.

Remarks

The **_rmtmp** function cleans up all temporary files in the current directory. The function removes only those files created by **tmpfile**; use it only in the same directory in which the temporary files were created.

Requirements

Routine	Required header	Compatibility
_rmtmp	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

See the example for tmpfile.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O
_flushall
tmpfile
_tempnam, _wtempnam, tmpnam, _wtmpnam

_rotl, _rotl64, _rotr, _rotr64

Rotates bits to the left (**_rotl**) or right (**_rotr**).

```
unsigned int _rotl(
   unsigned int value,
   int shift
);
unsigned __int64 _rotl64(
   unsigned __int64 value,
   int shift
);
unsigned int _rotr(
   unsigned int value,
   int shift
);
unsigned __int64 _rotl64(
   unsigned __int64 value,
   int shift
);
```

Parameters

value

Value to be rotated.

shift

Number of bits to shift.

Return Value

The rotated value. There is no error return.

Remarks

The **_rotl** and **_rotr** functions rotate the unsigned *value* by *shift* bits. **_rotl** rotates the value left. **_rotr** rotates the value right. Both functions wrap bits rotated off one end of *value* to the other end.

Requirements

Routine	Required header	Compatibility
_rotl, _rotl64	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_rotr, _rotr64	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_rot.c
/* This program shifts values to rotate an integer.
 */

#include <stdlib.h>
```

```
#include <stdio.h>

int main( void )
{
   unsigned val = 0x0fd93;
   __int64 val2 = 0x0101010101010101;

   printf( "0x%4.4x rotated left three times is 0x%4.4x\n",
           val, _rotl( val, 3 ) );
   printf( "0x%4.4x rotated right four times is 0x%4.4x\n",
           val, _rotr( val, 4 ) );

   printf( "%I64x rotated left three times is %I64x\n",
           val2, _rotl64( val2, 3 ) );
   printf( "%I64x rotated right four times is %I64x\n",
           val2, _rotr64( val2, 4 ) );
}
```

Output

```
0xfd93 rotated left three times is 0x7ec98
0xfd93 rotated right four times is 0x30000fd9
101010101010101 rotated left three times is 808080808080808
101010101010101 rotated right four times is 1010101010101010
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support

_lrotl, _lrotr

_RPT, _RPTF, _RPTW, _RPTFW Macros

Tracks an application's progress by generating a debug report (debug version only). Note that *n* specifies the number of arguments in *args* and can be 0, 1, 2, 3, or 4.

```
_RPTn(
      reportType,
      format,
...[args]
);
_RPTFn(
      reportType,
      format,
   [args]
);
_RPTWn(
      reportType,
      format
   [args]
);
_RPTFWn(
      reportType,
      format
   [args]
);
```

Parameters

reportType

Report type: **_CRT_WARN**, **_CRT_ERROR**, or **_CRT_ASSERT**.

format

Format-control string used to create the user message.

args

Substitution arguments used by *format*.

Remarks

All these macros take the *reportType* and *format* parameters. In addition, they might also take up to four additional arguments, signified by the number appended to the macro name. For example, **_RPT0** and **_RPTF0** take no additional arguments, **_RPT1** and **_RPTF1** take *arg1*, **_RPT2** and **_RPTF2** take *arg1* and *arg2*, and so on.

The **_RPT** and **_RPTF** macros are similar to the printf function, because they can be used to track an application's progress during the debugging process. However, these macros are more flexible than **printf** because they do not need to be enclosed in **#ifdef** statements to prevent them from being called in a retail build of an application. This flexibility is achieved by using the _DEBUG macro; the **_RPT** and **_RPTF** macros are only available when the **_DEBUG** flag is defined. When **_DEBUG** is not defined, calls to these macros are removed during preprocessing.

The **_RPTW** and **_RPTFW** macros are new in Visual C++ 2005 and are wide-character versions of these macros. They are like **wprintf** and take wide-character strings as arguments.

The **_RPT** macros call the _CrtDbgReport function to generate a debug report with a user message. The **_RPTW** macros call the **_CrtDbgReportW** function to generate the same report with wide characters. The **_RPTF** and **_RPTFW** macros create a debug report with the source file and line number where the report macro was called, in addition to the user message. The user message is created by substituting the *arg*[*n*] arguments into the *format* string, using the same rules defined by the printf function.

_CrtDbgReport or **_CrtDbgReportW** generates the debug report and determines its destinations based on the current report modes and file defined for *reportType*. The _CrtSetReportMode and _CrtSetReportFile functions are used to define the destinations for each report type.

If an **_RPT** macro is called and neither **_CrtSetReportMode** nor **_CrtSetReportFile** has been called, messages are displayed as

follows.

Report type	Output destination
_CRT_WARN	Warning text is not displayed.
_CRT_ERROR	A pop-up window. Same as if `_CrtSetReportMode(_CRT_ERROR, _CRTDBG_MODE_WNDW);` had been specified.
_CRT_ASSERT	Same as **_CRT_ERROR**.

When the destination is a debug message window and the user chooses the **Retry** button, **_CrtDbgReport** or **_CrtDbgReportW** returns 1, causing these macros to start the debugger, provided that just-in-time (JIT) debugging is enabled. For more information about using these macros as a debugging error handling mechanism, see Using Macros for Verification and Reporting.

Two other macros exist that generate a debug report. The _ASSERT macro generates a report, but only when its expression argument evaluates to FALSE. _ASSERTE is exactly like **_ASSERT**, but includes the failed expression in the generated report.

Requirements

Macro	Required header	Compatibility
_RPT macros	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_RPTF macros	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_RPTW macros	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_RPTFW macros	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Libraries

Debug versions of C run-time libraries only.

Although these are macros and are obtained by including Crtdbg.h, the application must link with one of the debug libraries because these macros call other run-time functions.

Example

See the example in the _ASSERT topic.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Debug Routines

_RTC_GetErrDesc

Returns a brief description of a run-time error check (RTC) type.

```
const char * _RTC_GetErrDesc(
   _RTC_ErrorNumber errnum
);
```

Parameters

errnum

A number between zero and one less than the value returned by **_RTC_NumErrors**.

Return Value

A character string that contains a short description of one of the error types detected by the run-time error check system. If error is less than zero or greater than or equal to the value returned by _RTC_NumErrors, **_RTC_GetErrDesc** returns NULL.

Requirements

Routine	Required header	Compatibility
_RTC_GetErrDesc	<rtcapi.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more information, see Compatibility.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Tasks

RTC Sample: Run-Time Error Checks

Reference

_RTC_NumErrors

Run-Time Error Checking

_RTC_NumErrors

Returns the total number of errors that can be detected by run-time error checks (RTC). You can use this number as the control in a **for** loop, where each value in the loop is passed to _RTC_GetErrDesc.

```
int _RTC_NumErrors( void );
```

Return Value

An integer whose value represents the total number of errors that can be detected by the Visual C++ run-time error checks.

Requirements

Routine	Required header	Compatibility
_RTC_NumErrors	<rtcapi.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more information, see Compatibility.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Tasks

RTC Sample: Run-Time Error Checks

Reference

_RTC_GetErrDesc

Run-Time Error Checking

_RTC_SetErrorFunc

Designates a function as the handler for reporting run-time error checks (RTCs). This function is deprecated; use **_RTC_SetErrorFuncW** instead.

```
_RTC_error_fn _RTC_SetErrorFunc(
   _RTC_error_fn function
);
```

Parameters

function

The address of the function that will handle run-time error checks.

Return Value

The previously defined error function. If there is no previously defined function, returns NULL.

Remarks

Do not use this function; instead, use **_RTC_SetErrorFuncW**. It is retained only for backward compatibility.

Requirements

Routine	Required header	Compatibility
_RTC_SetErrorFunc	<rtcapi.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more information, see Compatibility.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Tasks

RTC Sample: Run-Time Error Checks

Reference

_CrtDbgReport, _CrtDbgReportW

Run-Time Error Checking

_RTC_SetErrorFuncW

Designates a function as the handler for reporting run-time error checks (RTCs).

```
_RTC_error_fnW _RTC_SetErrorFuncW(
   _RTC_error_fnW function
);
```

Parameters

function

The address of the function that will handle run-time error checks.

Return Value

The previously defined error function. If there is no previously defined function, returns **NULL**.

Remarks

Use only **_RTC_SetErrorFuncW** in new code. **_RTC_SetErrorFunc** is only included in the library for backward compatibility.

Make sure that the address you pass to **_RTC_SetErrorFuncW** is that of a valid error handling function.

If an error has been assigned a type of –1 with _RTC_SetErrorType, the error handler function is not called.

Before you can call this function, you must first call one of the run-time error check initialization functions; see Using Run-Time Checks without the C Run-Time Library

_RTC_error_fnW is defined as follows:

typedef int (__cdecl *_RTC_error_fnW)(int *errorType* **, const wchar_t *** *filename* **, int** *linenumber* **, const wchar_t *** *moduleName* **, const wchar_t *** *format* **, ...);**

where:

errorType

The type of error specified by _RTC_SetErrorType.

filename

The source file where the failure occurred or null if no debug information is available.

linenumber

The line in *filename* where the failure occurred or 0 if no debug information is available.

moduleName

The DLL or executable name where the failure occurred.

format

printf style string to display an error message, using the remaining parameters. The first argument of the VA_ARGLIST is the RTC Error number that occurred.

For an example of using **_RTC_error_fnW**, see Customizing CRT Run-Time Error Checking.

Requirements

Routine	Required header	Compatibility
_RTC_SetErrorFuncW	<rtcapi.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more information, see Compatibility.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Tasks

RTC Sample: Run-Time Error Checks

Reference

_CrtDbgReport, _CrtDbgReportW

Run-Time Error Checking

_RTC_SetErrorType

Associates an error that is detected by run-time error checks (RTCs) with a type. Your error handler processes how to output errors of the specified type.

```
int _RTC_SetErrorType(
   _RTC_ErrorNumber errnum,
   int ErrType
);
```

Parameters

errnum

A number between zero and one less than the value returned by _RTC_NumErrors.

ErrType

A value to assign to this *errnum*. For example, you might use **_CRT_ERROR**. If you are using **_CrtDbgReport** as your error handler, *ErrType* can only be one of the symbols defined in _CrtSetReportMode. If you have your own error handler (_RTC_SetErrorFunc), you can have as many *ErrTypes* as there are *errnums*.

An *ErrType* of _RTC_ERRTYPE_IGNORE has special meaning to **_CrtSetReportMode**; the error is ignored.

Return Value

The previous value for the error type *type*.

Remarks

By default, all errors are set to *ErrType* = 1, which corresponds to **_CRT_ERROR**. For more information about the default error types such as **_CRT_ERROR**, see _CrtDbgReport.

Before you can call this function, you must first call one of the run-time error check initialization functions; see Using Run-Time Checks without the C Run-Time Library

Requirements

Routine	Required header	Compatibility
_RTC_SetErrorType	<rtcapi.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more information, see Compatibility.

Libraries

All versions of the C run-time libraries.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Tasks

RTC Sample: Run-Time Error Checks

Reference

_RTC_GetErrDesc

Run-Time Error Checking

_scalb

Scales argument by a power of 2.

```
double _scalb(
   double x,
   long exp
);
```

Parameters

x

Double-precision, floating-point value.

exp

Long integer exponent.

Return Value

Returns an exponential value if successful. On overflow (depending on the sign of *x*), **_scalb** returns +/– **HUGE_VAL**; the **errno** variable is set to **ERANGE**.

For more information about this and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_scalb** function calculates the value of *x* * 2exp.

Requirements

Routine	Required header	Compatibility
_scalb	<float.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support

ldexp

scanf Type Field Characters

The following information applies to any of the **scanf** family of functions, including the secure versions, such as **scanf_s**.

The *type* character is the only required format field; it appears after any optional format fields. The *type* character determines whether the associated argument is interpreted as a character, string, or number.

Type Characters for scanf functions

Character	Type of input expected	Type of argument	Size argument in secure version?
c	Character. When used with **scanf** functions, specifies single-byte character; when used with **wscanf** functions, specifies wide character. White-space characters that are ordinarily skipped are read when **c** is specified. To read next non–white-space single-byte character, use **%1s**; to read next non–white-space wide character, use **%1ws**.	Pointer to **char** when used with **scanf** functions, pointer to **wchar_t** when used with **wscanf** functions.	Required. Size does not include space for a null terminator.
C	Opposite size character. When used with **scanf** functions, specifies wide character; when used with **wscanf** functions, specifies single-byte character. White-space characters that are ordinarily skipped are read when **C** is specified. To read next non–white-space single-byte character, use **%1s**; to read next non–white-space wide character, use **%1ws**.	Pointer to **wchar_t** when used with **scanf** functions, pointer to **char** when used with **wscanf** functions.	Required. Size argument does not include space for a null terminator.
d	Decimal integer.	Pointer to **int**.	No.
i	Decimal, hexadecimal, or octal integer.	Pointer to **int**.	No.
o	Octal integer.	Pointer to **int**.	No.
u	Unsigned decimal integer.	Pointer to **unsigned int**.	No.
x	Hexadecimal integer.	Pointer to **int**.	No.
e, **E**, **f**, **g**, **G**	Floating-point value consisting of optional sign (+ or –), series of one or more decimal digits containing decimal point, and optional exponent ("e" or "E") followed by an optionally signed integer value.	Pointer to **float**.	No.
n	No input read from stream or buffer.	Pointer to **int**, into which is stored number of characters successfully read from stream or buffer up to that point in current call to **scanf** functions or **wscanf** functions.	No.

s	String, up to first white-space character (space, tab or newline). To read strings not delimited by space characters, use set of square brackets (**[ ]**), as discussed in scanf Width Specification.	When used with **scanf** functions, signifies single-byte character array; when used with **wscanf** functions, signifies wide-character array. In either case, character array must be large enough for input field plus terminating null character, which is automatically appended.	Required. Size includes space for a null terminator.
S	Opposite-size character string, up to first white-space character (space, tab or newline). To read strings not delimited by space characters, use set of square brackets (**[ ]**), as discussed in scanf Width Specification.	When used with **scanf** functions, signifies wide-character array; when used with **wscanf** functions, signifies single-byte–character array. In either case, character array must be large enough for input field plus terminating null character, which is automatically appended.	Required. Size includes space for a null terminator.

The **a** and **A** specifiers (see printf Type Field Characters) are not available with **scanf**.

The size arguments, if required, should be passed in the parameter list immediately following the argument they apply to. For example, the following code:

```
char string1[11], string2[9];
scanf("%10s %8s", string1, 11, string2, 9);
```

reads a single character into `c`, a decimal integer into `i`, a string with a maximum length of 10 into `string1`, and a string with a maximum length of 8 into `string2`. The buffer sizes should be at least one more than the width specifications since space must be reserved for the null terminator.

The format string can handle single-byte or wide character input regardless of whether the single-byte character or wide-character version of the function is used. Thus, to read single-byte or wide characters with **scanf** functions and **wscanf** functions, use format specifiers as follows.

To read character as	Use this function	With these format specifiers
single byte	**scanf** functions	**c**, **hc**, or **hC**
single byte	**wscanf** functions	**C**, **hc**, or **hC**
wide	**wscanf** functions	**c**, **lc**, or **lC**
wide	**scanf** functions	**C**, **lc**, or **lC**

To scan strings with **scanf** functions, and **wscanf** functions, use the above table with format type-specifiers **s** and **S** instead of **c** and **C**.

See Also

Reference

scanf, _scanf_l, wscanf, _wscanf_l

scanf Width Specification

This information applies to the interpretation of format strings in the **scanf** family of functions, including the secure versions such as **scanf_s**. These functions normally assume the input stream is divided into a sequence of tokens. Tokens are separated by whitespace (space, tab, or newline), or in the case of numerical types, by the natural end of a numerical data type as indicated by the first character that cannot be converted into numerical text. However, the width specification may be used to cause parsing of the input to stop before the natural end of a token.

The *width* specification consists of characters between the `%` and the type field specifier, which may include a positive integer called the *width* field and one or more characters indicating the size of the field, which may also be considered as modifiers of the type of the field, such as an indication of whether the integer type is **short** or **long**. Such characters are referred to as the size prefix.

The Width Field

The width field is a positive decimal integer controlling the maximum number of characters to be read for that field. No more than *width* characters are converted and stored at the corresponding *argument*. Fewer than *width* characters may be read if a whitespace character (space, tab, or newline) or a character that cannot be converted according to the given format occurs before *width* is reached.

The width specification is separate and distinct from the buffer size argument required by the secure versions of these functions (i.e., **scanf_s**, **wscanf_s**, etc.). In the following example, the width specification is 20, indicating that up to 20 characters are to be read from the input stream. The buffer length is 21, which includes room for the possible 20 characters plus the null terminator:

```
char str[21];
scanf("%20s", str, 21);
```

If the *width* field is not used, **scanf** will attempt to read the entire token into the string. If the size specified is not large enough to hold the entire token, nothing will be written to the destination string. If the *width* field is specified, then the first *width* characters in the token will be written to the destination string along with the null terminator.

The Size Prefix

The optional prefixes **h**, **l**, **ll**, **I64**, and **L** indicate the size of the *argument* (long or short, single-byte character or wide character, depending upon the type character that they modify). These format-specification characters are used with type characters in **scanf** or **wscanf** functions to specify interpretation of arguments as shown in the following table. The type prefix **I64** is a Microsoft extension and is not ANSI compatible. The type characters and their meanings are described in the "Type Characters for scanf functions" table in scanf Type Field Characters.

Note
The **h**, **l**, and **L** prefixes are Microsoft extensions when used with data of type **char**.

Size Prefixes for scanf and wscanf Format-Type Specifiers

To specify	Use prefix	With type specifier
double	**l**	**e**, **E**, **f**, **g**, or **G**
long double (same as double)	**L**	**e**, **E**, **f**, **g**, or **G**
long int	**l**	**d**, **i**, **o**, **x**, or **X**
long unsigned int	**l**	**u**
long long	**ll**	**d**, **i**, **o**, **x**, or **X**
short int	**h**	**d**, **i**, **o**, **x**, or **X**
short unsigned int	**h**	**u**

__int64	**I64**	**d**, **i**, **o**, **u**, **x**, or **X**
Single-byte character with **scanf**	**h**	**c** or **C**
Single-byte character with **wscanf**	**h**	**c** or **C**
Wide character with **scanf**	**l**	**c** or **C**
Wide character with **wscanf**	**l**	**c**, or **C**
Single-byte – character string with **scanf**	**h**	**s** or **S**
Single-byte – character string with **wscanf**	**h**	**s** or **S**
Wide-character string with **scanf**	**l**	**s** or **S**
Wide-character string with **wscanf**	**l**	**s** or **S**

The following examples use **h** and **l** with **scanf_s** functions and **wscanf_s** functions:

```
scanf_s( "%ls", &x, 2 );     // Read a wide-character string
wscanf_s( "%hC",&x, 2 );    // Read a single-byte character
```

If using an unsecure function in the **scanf** family, omit the size parameter indicating the buffer length of the preceding argument.

Reading Undelimited strings

To read strings not delimited by whitespace characters, a set of characters in brackets (**[ ]**) can be substituted for the **s** (string) type character. The set of characters in brackets is referred to as a control string. The corresponding input field is read up to the first character that does not appear in the control string. If the first character in the set is a caret (**^**), the effect is reversed: The input field is read up to the first character that does appear in the rest of the character set.

Note that **%[a-z]** and **%[z-a]** are interpreted as equivalent to **%[abcde...z]**. This is a common **scanf** function extension, but note that the ANSI standard does not require it.

Reading Unterminated strings

To store a string without storing a terminating null character ('\0'), use the specification **%*n*c** where *n* is a decimal integer. In this case, the **c** type character indicates that the argument is a pointer to a character array. The next *n* characters are read from the input stream into the specified location, and no null character ('\0') is appended. If *n* is not specified, its default value is 1.

When scanf stops reading a field

The **scanf** function scans each input field, character by character. It may stop reading a particular input field before it reaches a space character for a variety of reasons:

- The specified width has been reached.
- The next character cannot be converted as specified.
- The next character conflicts with a character in the control string that it is supposed to match.
- The next character fails to appear in a given character set.

For whatever reason, when the **scanf** function stops reading an input field, the next input field is considered to begin at the first unread character. The conflicting character, if there is one, is considered unread and is the first character of the next input field or the first character in subsequent read operations on the input stream.

See Also

Reference

scanf, _scanf_l, wscanf, _wscanf_l

scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l

scanf, _scanf_l, wscanf, _wscanf_l

Read formatted data from the standard input stream. These functions are deprecated because more secure versions are available; see scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l.

```
int scanf(
   const char *format [,
      argument]...
);
int _scanf_l(
   const char *format,
   locale_t locale [,
      argument]...
);
int wscanf(
   const wchar_t *format [,
      argument]...
);
int _wscanf_l(
    const wchar_t *format,
   locale_t locale [,
      argument]...
);
```

Parameters

format

Format control string.

argument

Optional arguments.

locale

The locale to use.

Return Value

Returns the number of fields successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned.

If *format* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **EOF** and set **errno** to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **scanf** function reads data from the standard input stream **stdin** and writes the data into the location given by *argument*. Each *argument* must be a pointer to a variable of a type that corresponds to a type specifier in *format*. If copying takes place between strings that overlap, the behavior is undefined.

Security Note
When reading a string with **scanf**, always specify a width for the **%s** format (for example, `"%32s"` instead of `"%s"`); otherwise, improperly formatted input can easily cause a buffer overrun. Alternately, consider using scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l or fgets.

wscanf is a wide-character version of **scanf**; the *format* argument to **wscanf** is a wide-character string. **wscanf** and **scanf** behave identically if the stream is opened in ANSI mode. **scanf** doesn't currently support input from a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tscanf	**scanf**	**scanf**	**wscanf**
_tscanf_l	**_scanf_l**	**_scanf_l**	**_wscanf_l**

For more information, see Format Specification Fields — scanf functions and wscanf Functions.

Requirements

Routine	Required header	Compatibility
scanf, **_scanf_l**	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wscanf, **_wscanf_l**	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_scanf.c
// compile with: /W3
 /* This program uses the scanf and wscanf functions
  * to read formatted input.
  */

#include <stdio.h>

int main( void )
{
   int   i, result;
   float fp;
   char  c, s[81];
   wchar_t wc, ws[81];
   result = scanf( "%d %f %c %C %80s %80S", &i, &fp, &c, &wc, s, ws ); // C4996
   // Note: scanf and wscanf are deprecated; consider using scanf_s and wscanf_s
   printf( "The number of fields input is %d\n", result );
   printf( "The contents are: %d %f %c %C %s %S\n", i, fp, c, wc, s, ws);
   result = wscanf( L"%d %f %hc %lc %80S %80ls", &i, &fp, &c, &wc, s, ws ); // C4996
   wprintf( L"The number of fields input is %d\n", result );
   wprintf( L"The contents are: %d %f %C %c %hs %s\n", i, fp, c, wc, s, ws);
}
```

Input

```
71 98.6 h z Byte characters
36 92.3 y n Wide characters
```

Output

```
The number of fields input is 6
The contents are: 71 98.599998 h z Byte characters
The number of fields input is 6
The contents are: 36 92.300003 y n Wide characters
```

.NET Framework Equivalent

- System::Console::Read
- System::Console::ReadLine
- See also **Parse** methods, such as System::Double::Parse.

See Also

Reference

Floating-Point Support
Stream I/O
Locale
fscanf, _fscanf_l, fwscanf, _fwscanf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
sscanf, _sscanf_l, swscanf, _swscanf_l

scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l

Read formatted data from the standard input stream. These are versions of scanf, _scanf_l, wscanf, _wscanf_l with security enhancements as described in Security Enhancements in the CRT.

```
int scanf_s(
   const char *format [,
      argument]...
);
int _scanf_s_l(
   const char *format,
   locale_t locale [,
      argument]...
);
int wscanf_s(
   const wchar_t *format [,
      argument]...
);
int _wscanf_s_l(
   const wchar_t *format,
   locale_t locale [,
      argument]...
);
```

Parameters

format

Format control string.

argument

Optional arguments.

locale

The locale to use.

Return Value

Returns the number of fields successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned. The return value is **EOF** for an error or if the end-of-file character or the end-of-string character is encountered in the first attempt to read a character. If *format* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **scanf_s** and **wscanf_s** return **EOF** and set **errno** to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **scanf_s** function reads data from the standard input stream **stdin** and writes the data into the location given by *argument*. Each *argument* must be a pointer to a variable of a type that corresponds to a type specifier in *format*. If copying takes place between strings that overlap, the behavior is undefined.

wscanf_s is a wide-character version of **scanf_s**; the *format* argument to **wscanf_s** is a wide-character string. **wscanf_s** and **scanf_s** behave identically identically if the stream is opened in ANSI mode. **scanf_s** doesn't currently support input from a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Unlike **scanf** and **wscanf**, **scanf_s** and **wscanf_s** require the buffer size to be specified for all input parameters of type **c**, **C**, **s**, **S**, or **[**. The buffer size is passed as an additional parameter immediately following the pointer to the buffer or variable. For example, if reading a string, the buffer size for that string is passed as follows:

```
char s[10];

scanf("%9s", s, 10);
```

The buffer size includes the terminating null. A width specification field may be used to ensure that the token read in will fit into the buffer. If no width specification field is used, and the token read is too big to fit in the buffer, nothing will be written to that buffer.

Note
The size parameter is of type **unsigned**, not **size_t**.

In the case of characters, one may read a single character as follows:

```
char c;
scanf("%c", &c, 1);
```

When reading multiple characters for non-null terminated strings, integers are used as the width specification and the buffer size.

```
char c[4];
scanf("%4c", &c, 4); // not null terminated
```

For more information, see scanf Width Specification.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tscanf_s	**scanf_s**	**scanf_s**	**wscanf_s**
_tscanf_s_l	**_scanf_s_l**	**_scanf_s_l**	**_wscanf_s_l**

For more information, see Format Specification Fields — scanf functions and wscanf Functions.

Requirements

Routine	Required header	Compatibility
scanf_s, **_scanf_s_l**	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wscanf_s, **_wscanf_s_l**	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_scanf_s.c
// This program uses the scanf_s and wscanf_s functions
// to read formatted input.

#include <stdio.h>

int main( void )
{
   int      i,
            result;
   float    fp;
   char     c,
            s[81];
   wchar_t  wc,
            ws[81];

   result = scanf_s( "%d %f %c %C %s %S", &i, &fp, &c, 1,
                     &wc, 1, s, 80, ws, 80 );
   printf( "The number of fields input is %d\n", result );
   printf( "The contents are: %d %f %c %C %s %S\n", i, fp, c,
           wc, s, ws);
```

```
    result = wscanf_s( L"%d %f %hc %lc %S %ls", &i, &fp, &c, 2,
                       &wc, 1, s, 80, ws, 80 );
    wprintf( L"The number of fields input is %d\n", result );
    wprintf( L"The contents are: %d %f %C %c %hs %s\n", i, fp,
             c, wc, s, ws);
}
```

Input

```
71 98.6 h z Byte characters
36 92.3 y n Wide characters
```

Output

```
The number of fields input is 6
The contents are: 71 98.599998 h z Byte characters
The number of fields input is 6
The contents are: 36 92.300003 y n Wide characters
```

.NET Framework Equivalent

- System::Console::Read
- System::Console::ReadLine
- See also **Parse** methods, such as System::Double::Parse.

See Also

Reference

Floating-Point Support
Stream I/O
Locale
fscanf, _fscanf_l, fwscanf, _fwscanf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
sscanf, _sscanf_l, swscanf, _swscanf_l

_scprintf, _scprintf_l, _scwprintf, _scwprintf_l

Returns the number of characters in the formatted string.

```
int _scprintf(
   const char *format [,
      argument] ...
);
int _scprintf_l(
   const char *format,
   locale_t locale [,
      argument] ...
);
int _scwprintf(
   const wchar_t *format [,
      argument] ...
);
int _scwprintf_l(
   const wchar_t *format,
   locale_t locale [,
      argument] ...
);
```

Parameters

format

Format-control string.

argument

Optional arguments.

locale

The locale to use.

For more information, see Format Specifications.

Return Value

Returns the number of characters that would be generated if the string were to be printed or sent to a file or buffer using the specified formatting codes. The value returned does not include the terminating null character. **_scwprintf** performs the same function for wide characters.

If *format* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

For information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each *argument* (if any) is converted according to the corresponding format specification in *format*. The format consists of ordinary characters and has the same form and function as the *format* argument for printf.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Security Note
Ensure that *format* is not a user-defined string.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_sctprintf	**_scprintf**	**_scprintf**	**_scwprintf**

_sctprintf_l	_scprintf_l	_scprintf_l	_scwprintf_l

Requirements

Routine	Required header	Compatibility
_scprintf, **_scprintf_l**	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_scwprintf, **_scwprintf_l**	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt__scprintf.c

#define _USE_MATH_DEFINES

#include <stdio.h>
#include <math.h>
#include <malloc.h>

int main( void )
{
   int count;
   int size;
   char *s = NULL;

   count = _scprintf( "The value of Pi is calculated to be %f.\n",
                      M_PI);

   size = count + 1; // the string will need one more char for the null terminator
   s = malloc(sizeof(char) * size);
   sprintf_s(s, size, "The value of Pi is calculated to be %f.\n",
                      M_PI);
   printf("The length of the following string will be %i.\n", count);
   printf("%s", s);
   free( s );
}
```

Output

```
The length of the following string will be 46.
The value of Pi is calculated to be 3.141593.
```

See Also

Reference

Stream I/O
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
scanf, _scanf_l, wscanf, _wscanf_l
sscanf, _sscanf_l, swscanf, _swscanf_l
vprintf Functions

_scprintf_p, _scprintf_p_l, _scwprintf_p, _scwprintf_p_l

Returns the number of characters in the formatted string, with the ability to specify the order in which parameters are used in the format string.

```
int _scprintf_p(
   const char *format [,
      argument] ...
);
int _scprintf_p_l(
   const char *format,
   locale_t locale [,
      argument] ...
);
int _scwprintf_p (
   const wchar_t *format [,
      argument] ...
);
int _scwprintf_p _l(
   const wchar_t *format,
   locale_t locale [,
      argument] ...
);
```

Parameters

format

Format-control string.

argument

Optional arguments.

locale

The locale to use.

Return Value

Returns the number of characters that would be generated if the string were to be printed or sent to a file or buffer using the specified formatting codes. The value returned does not include the terminating null character. **_scwprintf_p** performs the same function for wide characters.

The difference between **_scprintf_p** and **_scprintf** is that **_scprintf_p** supports positional parameters, which allows specifying the order in which the arguments are used in the format string. For more information, see printf_p Positional Parameters.

If *format* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

For information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each *argument* (if any) is converted according to the corresponding format specification in *format*. The format consists of ordinary characters and has the same form and function as the *format* argument for printf.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Security Note
Ensure that *format* is not a user-defined string.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined

_sctprintf_p	**_scprintf_p**	**_scprintf_p**	**_scwprintf_p**
_sctprintf_p_l	**_scprintf_p_l**	**_scprintf_p_l**	**_scwprintf_p_l**

Requirements

Routine	Required header	Compatibility
_scprintf_p, **_scprintf_p_l**	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_scwprintf_p, **_scwprintf_p_l**	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

See Also

Reference

Stream I/O

_scprintf, _scprintf_l, _scwprintf, _scwprintf_l

_printf_p, _printf_p_l, _wprintf_p, _wprintf_p_l

_searchenv, _wsearchenv

Searches for a file using environment paths. These functions are deprecated because more secure versions are available; see _searchenv_s, _wsearchenv_s.

```
void _searchenv(
   const char *filename,
   const char *varname,
   char *pathname
);
void _wsearchenv(
   const wchar_t *filename,
   const wchar_t *varname,
   wchar_t *pathname
);
template <size_t size>
void _searchenv(
   const char *filename,
   const char *varname,
   char (&pathname)[size]
); // C++ only
template <size_t size>
void _wsearchenv(
   const wchar_t *filename,
   const wchar_t *varname,
   wchar_t (&pathname)[size]
); // C++ only
```

Parameters

filename

Name of the file to search for.

varname

Environment to search.

pathname

Buffer to store the complete path.

Remarks

The **_searchenv** routine searches for the target file in the specified domain. The *varname* variable can be any environment or user-defined variable that specifies a list of directory paths, such as **PATH**, **LIB**, and **INCLUDE**. Because **_searchenv** is case-sensitive, *varname* should match the case of the environment variable.

The routine searches first for the file in the current working directory. If it does not find the file, it looks next through the directories specified by the environment variable. If the target file is in one of those directories, the newly created path is copied into *pathname*. If the *filename* file is not found, *pathname* contains an empty null-terminated string.

The *pathname* buffer should be at least **_MAX_PATH** characters long to accommodate the full length of the constructed path name. Otherwise, **_searchenv** might overrun the *pathname* buffer resulting in unexpected behavior.

_wsearchenv is a wide-character version of **_searchenv**; the arguments to **_wsearchenv** are wide-character strings. **_wsearchenv** and **_searchenv** behave identically otherwise.

If *filename* is an empty string, these functions return **ENOENT**.

If *filename* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

For more information about **errno** and error codes, see errno Constants.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tsearchenv	**_searchenv**	**_searchenv**	**_wsearchenv**

Requirements

Routine	Required header	Compatibility
_searchenv	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wsearchenv	<stdlib.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_searchenv.c
// compile with: /W3
// This program searches for a file in
// a directory specified by an environment variable.

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char pathbuffer[_MAX_PATH];
   char searchfile[] = "CL.EXE";
   char envvar[] = "PATH";

   // Search for file in PATH environment variable:
   _searchenv( searchfile, envvar, pathbuffer ); // C4996
   // Note: _searchenv is deprecated; consider using _searchenv_s
   if( *pathbuffer != '\0' )
      printf( "Path for %s:\n%s\n", searchfile, pathbuffer );
   else
      printf( "%s not found\n", searchfile );
}
```

Sample Output

```
Path for CL.EXE:
C:\Program Files\Microsoft Visual Studio 8\VC\BIN\CL.EXE
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Directory Control
getenv, _wgetenv
_putenv, _wputenv
_searchenv_s, _wsearchenv_s

_searchenv_s, _wsearchenv_s

Searches for a file using environment paths. These are versions of _searchenv, _wsearchenv with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _searchenv_s(
   const char *filename,
   const char *varname,
   char *pathname,
   size_t numberOfElements
);
errno_t _wsearchenv_s(
   const wchar_t *filename,
   const wchar_t *varname,
   wchar_t *pathname,
   size_t numberOfElements
);
template <size_t size>
errno_t _searchenv_s(
   const char *filename,
   const char *varname,
   char (&pathname)[size]
); // C++ only
template <size_t size>
errno_t _wsearchenv_s(
   const wchar_t *filename,
   const wchar_t *varname,
   wchar_t (&pathname)[size]
); // C++ only
```

Parameters

[in] *filename*

Name of the file to search for.

[in] *varname*

Environment to search.

[out] *pathname*

Buffer to store the complete path.

[in] *numberOfElements*

Size of the *pathname* buffer.

Return Value

Zero if successful; an error code on failure.

If *filename* is an empty string, the return value is **ENOENT**.

Error Conditions

filename	***varname***	***pathname***	***numberOfElements***	**Return value**	**Contents of *pathname***
any	any	**NULL**	any	**EINVAL**	n/a
NULL	any	any	any	**EINVAL**	not changed
any	any	any	<= 0	**EINVAL**	not changed

If any of these error conditions occurs, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **EINVAL**.

Remarks

The **_searchenv_s** routine searches for the target file in the specified domain. The *varname* variable can be any environment or user-defined variable that specifies a list of directory paths, such as **PATH**, **LIB**, and **INCLUDE**. Because **_searchenv_s** is case-sensitive, *varname* should match the case of the environment variable. If *varname* does not match the name of an environment variable defined in the process's environment, the function returns zero and the *pathname* variable is unchanged.

The routine searches first for the file in the current working directory. If it does not find the file, it looks next through the directories specified by the environment variable. If the target file is in one of those directories, the newly created path is copied into *pathname*. If the *filename* file is not found, *pathname* contains an empty null-terminated string.

The *pathname* buffer should be at least **_MAX_PATH** characters long to accommodate the full length of the constructed path name. Otherwise, **_searchenv_s** might overrun the *pathname* buffer resulting in unexpected behavior.

_wsearchenv_s is a wide-character version of **_searchenv_s**; the arguments to **_wsearchenv_s** are wide-character strings. **_wsearchenv_s** and **_searchenv_s** behave identically otherwise.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tsearchenv_s	**_searchenv_s**	**_searchenv_s**	**_wsearchenv_s**

Requirements

Routine	Required header	Compatibility
_searchenv_s	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wsearchenv_s	<stdlib.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_searchenv_s.c
/* This program searches for a file in
 * a directory specified by an environment variable.
 */

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char pathbuffer[_MAX_PATH];
   char searchfile[] = "CL.EXE";
   char envvar[] = "PATH";
   errno_t err;

   /* Search for file in PATH environment variable: */
   err = _searchenv_s( searchfile, envvar, pathbuffer, _MAX_PATH );
   if (err != 0)
   {
      printf("Error searching the path. Error code: %d\n", err);
   }
   if( *pathbuffer != '\0' )
      printf( "Path for %s:\n%s\n", searchfile, pathbuffer );
   else
      printf( "%s not found\n", searchfile );
}
```

Sample Output

```
Path for CL.EXE:
C:\Program Files\Microsoft Visual Studio 2005\VC7\BIN\CL.EXE
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Directory Control
_searchenv, _wsearchenv
getenv, _wgetenv
_putenv, _wputenv

__security_init_cookie

Initializes the global security cookie.

```
void __security_init_cookie(void);
```

Remarks

The global security cookie is used for buffer overrun protection in code compiled with /GS (Buffer Security Check) and in code that uses exception handling. Essentially, on entry to an overrun-protected function, the cookie is put on the stack, and on exit, the value on the stack is compared against the global cookie. Any difference between them indicates that a buffer overrun has occurred and results in immediate termination of the program.

Normally, **__security_init_cookie** is called by the CRT when it starts up. If you bypass CRT initialization (for example, by writing a DLL and specifying an entry-point with /ENTRY), then you must call **__security_init_cookie** yourself.

The call to **__security_init_cookie** must be made before any overrun-protected function is entered; otherwise a spurious buffer overrun will be detected. For more information, see C Run-Time Error R6035.

Example

See the examples in C Run-Time Error R6035.

Requirements

Routine	Required header	Compatibility
__security_init_cookie	<process.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. This function should only be called from native code, not managed.

See Also

Reference

Compiler Security Checks In Depth

_set_abort_behavior

Specifies the action to be taken when a program aborts.

```
unsigned int _set_abort_behavior(
   unsigned int flags,
   unsigned int mask
);
```

Parameters

[in] *flags*

New value of the abort flags.

[in] *mask*

Mask for the abort flags bits to set.

Return Value

The old value of the flags.

Remarks

There are two abort flags: **_WRITE_ABORT_MSG** and **_CALL_REPORTFAULT**. **_WRITE_ABORT_MSG** determines whether a helpful text message is printed when a program is aborted. The message states in English that the application has requested the runtime to terminate it in an unusual way and suggests that the user contact the application's support team for more information. The default behavior is to print the message. **_CALL_REPORTFAULT**, if set, specifies that a Watson crash dump is generated and reported when **abort** is called. Crash dump reporting is enabled by default.

Requirements

Routine	Required header	Compatibility
_set_abort_behavior	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_set_abort_behavior.c
// compile with: /c
#include <stdlib.h>

int main()
{
   printf("Suppressing the abort message. If successful, this message"
          " will be the only output.\n");
   // Suppress the abort message
   _set_abort_behavior( 0, _WRITE_ABORT_MSG);
   abort();
}
```

Sample Output

```
Suppressing the abort message. If successful, this message will be the only output.
```

See Also

Reference

abort

_set_amblksiz

Provides a secure way to set the block size for memory allocations.

```
errno_t _set_amblksiz(
   size_t blockSize
);
```

Parameters

[in] *blockSize*

The desired size of memory blocks for memory allocation.

Return Value

Zero if successful; an error code on failure.

Remarks

This function sets the value of the _amblksiz global variable, which is used by malloc and other memory allocation functions to determine the smallest block of memory that might be allocated from the heap at once. If *blockSize* is not a multiple of two, the actual block size is the next multiple of two.

Requirements

Routine	Required header	Compatibility
_set_amblksiz	<malloc.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

_amblksiz

_get_amblksiz

_set_controlfp

Sets the floating-point control word.

```
void __cdecl _set_controlfp(
    unsigned int newControl,
    unsigned int mask
);
```

Parameters

newControl

New control-word bit values.

mask

Mask for new control-word bits to set.

Return Value

None.

Remarks

The **_set_controlfp** is similar to **_control87**, but it only sets the floating-point control word to *newControl*. The bits in the values indicate the floating-point control state. The floating-point control state allows the program to change the precision, rounding, and infinity modes in the floating-point math package. You can also mask or unmask floating-point exceptions using **_set_controlfp**. For more information, see _control87, _controlfp, __control87_2.

This function is deprecated when compiling with /clr (Common Language Runtime Compilation) or **/clr:pure** because the common language runtime only supports the default floating-point precision.

Requirements

Routine	Required header	Compatibility
_set_controlfp	<float.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003; x86 processor only

For more compatibility information, see Compatibility in the Introduction.

See Also

Reference

Floating-Point Support
_clear87, _clearfp
_status87, _statusfp, _statusfp2

_set_doserrno

Sets the value of the _doserrno global variable.

```
errno_t _set_doserrno(
   int value
);
```

Parameters

[in] *value*

The new value of **_doserrno**.

Return Value

Returns zero if successful.

Remarks

Possible values are defined in Errno.h.

Requirements

Routine	Required header	Optional header	Compatibility
_set_doserrno	<stdlib.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

See Also

Reference

_get_doserrno

errno, _doserrno, _sys_errlist, and _sys_nerr

_set_errno

Set the value of the **errno** global variable.

```
errno_t _set_errno(
   int value
);
```

Parameters

[in] *value*

The new value of **errno**.

Return Value

Returns zero if successful.

Remarks

Possible values are defined in Errno.h. Also, see errno Constants.

Example

```
// crt_set_errno.c
#include <stdio.h>
#include <errno.h>

int main()
{
   _set_errno( EILSEQ );
   perror( "Oops" );
}
```

Output

```
Oops: Illegal byte sequence
```

Requirements

Routine	Required header	Optional header	Compatibility
_set_errno	<stdlib.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

See Also

Reference

_get_errno

errno, _doserrno, _sys_errlist, and _sys_nerr

_set_error_mode

Modifies **__error_mode** to determine a non-default location where the C run time writes an error message for an error that will possibly end the program.

```
int _set_error_mode(
   int modeval
);
```

Parameters

modeval

Destination of error messages.

Return Value

Returns the old setting or -1 if an error occurs.

Remarks

Controls the error output sink by setting the value of **__error_mode**. For example, you can direct output to standard error or use the **MessageBox** API.

The *modeval* parameter can be set to one of the following.

Parameter	Description
_OUT_TO_DEFAULT	Error sink is determined by **__app_type**.
_OUT_TO_STDERR	Error sink is a standard error.
_OUT_TO_MSGBOX	Error sink is a message box.
_REPORT_ERRMODE	Report the current **__error_mode** value.

If a value other than those listed above is passed in, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **_set_error_mode** sets **errno** to **EINVAL** and returns -1.

When used with an assert, **_set_error_mode** displays the failed statement in the dialog box and give you the option of clicking **Ignore**, which allows you to continue executing the program.

Requirements

Routine	Required header	Compatibility
_set_error_mode	<stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

```
// crt_set_error_mode.c
// compile with: /c
#include <stdlib.h>
#include <assert.h>

int main()
{
   _set_error_mode(_OUT_TO_STDERR);
   assert(2+2==5);
}
```

Sample Output

```
Assertion failed: 2+2==5, file crt_set_error_mode.c, line 8

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

See Also

Reference

assert (CRT)

_set_fileinfo

Sets the default file translation mode for file I/O operations.

Starting with Visual C++ 2005, **_get_fileinfo** is no longer supported in any Microsoft developer tools or libraries.

```
errno_t _set_fileinfo(
   int value
);
```

Parameters

[in] *value*

The new value of the **_fileinfo** variable.

Return Value

Returns zero.

Remarks

The function sets the _fileinfo global variable. This variable determines what information is passed to processes created with the spawn functions. For more information, see _spawn, _wspawn Functions.

Requirements

Routine	Required header	Compatibility
_set_fileinfo	None	None

See Also

Reference

_fmode
_get_fmode
_setmode
Text and Binary Mode File I/O
_get_fileinfo

_set_fmode

Sets the default file translation mode for file I/O operations.

```
errno_t _set_fmode(
   int mode
);
```

Parameters

[in] *mode*

The file translation mode desired: **_O_TEXT** or **_O_BINARY**.

Return Value

Returns zero if successful, an error code on failure. If *mode* is not **_O_TEXT** or **_O_BINARY** or **_O_WTEXT**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Remarks

The function sets the _fmode global variable. This variable specifies the default file translation mode for the file I/O operations **_open** and **_pipe**.

_O_TEXT and **_O_BINARY** are defined in Fcntl.h. **EINVAL** is defined in Errno.h.

Requirements

Routine	Required header	Optional header	Compatibility
_set_fmode	<stdlib.h>	<fcntl.h>, <errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_set_fmode.c
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>     /* for _O_TEXT and _O_BINARY */
#include <errno.h>     /* for EINVAL */
#include <sys\stat.h>  /* for _S_IWRITE */
#include <share.h>     /* for _SH_DENYNO */

int main()
{
   int mode, fd, ret;
   errno_t err;
   int buf[12] = { 65, 66, 67, 68, 69, 70, 71, 72, 73, 74,
                   75, 76 };
   char * filename = "fmode.out";

   err = _get_fmode(&mode);
   if (err == EINVAL)
   {
      printf( "Invalid parameter: mode\n");
      return 1;
   }
   else
      printf( "Default Mode is %s\n", mode == _O_TEXT ? "text" :
              "binary");
```

```
    err = _set_fmode(_O_BINARY);
    if (err == EINVAL)
    {
       printf( "Invalid mode.\n");
       return 1;
    }

    if ( _sopen_s(&fd, filename, _O_RDWR | _O_CREAT, _SH_DENYNO, _S_IWRITE | _S_IREAD) != 0
 )
    {
       printf( "Error opening the file %s\n", filename);
       return 1;
    }

    if (ret = _write(fd, buf, 12*sizeof(int)) < 12*sizeof(int))
    {
       printf( "Problem writing to the file %s.\n", filename);
       printf( "Number of bytes written: %d\n", ret);
    }

    if (_close(fd) != 0)
    {
       printf("Error closing the file %s. Error code %d.\n",
              filename, errno);
    }

    system("type fmode.out");
 }
```

Sample Output

```
Default Mode is binary
A   B   C   D   E   F   G   H   I   J   K   L
```

See Also

Reference

_fmode
_get_fmode
_setmode
Text and Binary Mode File I/O

_set_invalid_parameter_handler

Sets a function to be called when the CRT detects an invalid argument.

```
_invalid_parameter_handler _set_invalid_parameter_handler(
      _invalid_parameter_handler pNew
);
```

Parameters

[in] *pNew*

The function pointer to the new invalid parameter handler.

Return Value

A pointer to the invalid parameter handler before the call.

Remarks

Many C run-time functions check the validity of arguments passed to them. If an invalid argument is passed, the function can set the **errno** error number or return an error code. In such cases, the invalid parameter handler is also called. This function allows that handler to be set to another function. Only one function can be specified as the invalid argument handler at a time.

The invalid parameter handler function must have the following prototype:

```
void _invalid_parameter(
   const wchar_t * expression,
   const wchar_t * function,
   const wchar_t * file,
   unsigned int line,
   uintptr_t pReserved
);
```

The first argument is the argument expression. The second argument is the name of the CRT function that received the invalid argument. The third argument is the file name in the CRT source. The fourth argument is the line in that file. The last argument is reserved. The parameters all have the value **NULL** unless a debug version of the CRT library is used.

Requirements

Routine	Required header	Compatibility
_set_invalid_parameter_handler	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

In the following example, an invalid parameter error handler is used to print the function that received the invalid parameter and the file and line in CRT sources. When the debug CRT library is used, invalid parameter errors also throw an assertion, which is disabled in this example using _CrtSetReportMode.

```
// crt_set_invalid_parameter_handler.c
// compile with: /Zi /MTd
#include <stdio.h>
#include <stdlib.h>
#include <crtdbg.h>  // For _CrtSetReportMode

void myInvalidParameterHandler(const wchar_t* expression,
   const wchar_t* function,
   const wchar_t* file,
```

```
    unsigned int line,
    uintptr_t pReserved)
{
    wprintf(L"Invalid parameter detected in function %s."
            L" File: %s Line: %d\n", function, file, line);
    wprintf(L"Expression: %s\n", expression);
}


int main( )
{
    char* formatString;

    _invalid_parameter_handler oldHandler, newHandler;
    newHandler = myInvalidParameterHandler;
    oldHandler = _set_invalid_parameter_handler(newHandler);

    // Disable the message box for assertions.
    _CrtSetReportMode(_CRT_ASSERT, 0);

    // Call printf_s with invalid parameters.

    formatString = NULL;
    printf(formatString);
}
```

Output

```
Invalid parameter detected in function printf. File: printf.c Line: 54
Expression: (format != NULL)
```

See Also

Reference

Security-Enhanced Versions of CRT Functions

errno, _doserrno, _sys_errlist, and _sys_nerr

_set_new_handler

Transfers control to your error-handling mechanism if the **new** operator fails to allocate memory.

```
_PNH _set_new_handler(
   _PNH pNewHandler
);
```

Parameters

pNewHandler

Pointer to the application-supplied memory handling function. An argument of 0 causes the new handler to be removed.

Return Value

Returns a pointer to the previous exception handling function registered by **_set_new_handler**, so that the previous function can be restored later. If no previous function has been set, the return value can be used to restore the default behavior; this value can be **NULL**.

Remarks

The C++ **_set_new_handler** function specifies an exception-handling function that gains control if the **new** operator fails to allocate memory. If **new** fails, the run-time system automatically calls the exception-handling function that was passed as an argument to **_set_new_handler**. **_PNH**, defined in New.h, is a pointer to a function that returns type **int** and takes an argument of type **size_t**. Use **size_t** to specify the amount of space to be allocated.

There is no default handler.

_set_new_handler is essentially a garbage-collection scheme. The run-time system retries allocation each time your function returns a nonzero value and fails if your function returns 0.

An occurrence of the **_set_new_handler** function in a program registers the exception-handling function specified in the argument list with the run-time system:

```
#include <new.h>
int handle_program_memory_depletion( size_t )
{
   // Your code
}
int main( void )
{
   _set_new_handler( handle_program_memory_depletion );
   int *pi = new int[BIG_NUMBER];
}
```

You can save the function address that was last passed to the **_set_new_handler** function and reinstate it later:

```
_PNH old_handler = _set_new_handler( my_handler );
   // Code that requires my_handler
   _set_new_handler( old_handler )
   // Code that requires old_handler
```

The C++ _set_new_mode function sets the new handler mode for malloc. The new handler mode indicates whether, on failure, **malloc** is to call the new handler routine as set by **_set_new_handler**. By default, **malloc** does not call the new handler routine on failure to allocate memory. You can override this default behavior so that, when **malloc** fails to allocate memory, **malloc** calls the new handler routine in the same way that the **new** operator does when it fails for the same reason. To override the default, call:

```
_set_new_mode(1)
```

early in your program or link with Newmode.obj.

If a user-defined **operator new** is provided, the new handler functions are not automatically called on failure.

For more information, see new and delete in the *C++ Language Reference*.

There is a single **_set_new_handler** handler for all dynamically linked DLLs or executables; even if you call **_set_new_handler** your handler might be replaced by another or that you are replacing a handler set by another DLL or executable.

Requirements

Routine	Required header	Compatibility
_set_new_handler	<new.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

In this example, when the allocation fails, control is transferred to MyNewHandler. The argument passed to MyNewHandler is the number of bytes requested. The value returned from MyNewHandler is a flag indicating whether allocation should be retried: a nonzero value indicates that allocation should be retried, and a zero value indicates that allocation has failed.

```
// crt_set_new_handler.cpp
// compile with: /c
#include <stdio.h>
#include <new.h>
#define BIG_NUMBER 0x1fffffff

int coalesced = 0;

int CoalesceHeap()
{
   coalesced = 1;  // Flag RecurseAlloc to stop
   // do some work to free memory
   return 0;
}
// Define a function to be called if new fails to allocate memory.
int MyNewHandler( size_t size )
{
   printf("Allocation failed. Coalescing heap.\n");

   // Call a function to recover some heap space.
   return CoalesceHeap();
}

int RecurseAlloc() {
   int *pi = new int[BIG_NUMBER];
   if (!coalesced)
      RecurseAlloc();
   return 0;
}

int main()
{
   // Set the failure handler for new to be MyNewHandler.
   _set_new_handler( MyNewHandler );
   RecurseAlloc();
}
```

Output

```
Allocation failed. Coalescing heap.

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation

calloc

free

realloc

_set_new_mode

Sets a new handler mode for **malloc**.

```
int _set_new_mode(
   int newhandlermode
);
```

Parameters

newhandlermode

New handler mode for **malloc**; valid value is 0 or 1.

Return Value

Returns the previous handler mode set for **malloc**. A return value of 1 indicates that, on failure to allocate memory, **malloc** previously called the new handler routine; a return value of 0 indicates that it did not. If the *newhandlermode* argument does not equal 0 or 1, returns –1.

Remarks

The C++ **_set_new_mode** function sets the new handler mode for malloc. The new handler mode indicates whether, on failure, **malloc** is to call the new handler routine as set by _set_new_handler. By default, **malloc** does not call the new handler routine on failure to allocate memory. You can override this default behavior so that, when **malloc** fails to allocate memory, **malloc** calls the new handler routine in the same way that the **new** operator does when it fails for the same reason. For more information, see the new and delete operators in the *C++ Language Reference*. To override the default, call:

```
_set_new_mode(1)
```

early in your program or link with Newmode.obj (see Link Options).

This function validates its parameter. If *newhandlermode* is anything other than 0 or 1, the function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, **_set_new_mode** returns -1 and sets **errno** to **EINVAL**.

Requirements

Routine	Required header	Compatibility
_set_new_mode	<new.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation
calloc
free
realloc
_query_new_handler
_query_new_mode

_set_output_format

Customizes output formats used by formatted I/O functions.

```
unsigned int _set_output_format(
   unsigned int format
);
```

Parameters

[in] *format*

An value representing the format to use.

Return value

The previous output format.

Remarks

_set_output_format is used to configure the output of formatted I/O functions such as printf. At present, the only formatting convention that can be changed by this function is the number of digits displayed in exponents in the output of floating point numbers.

By default, the output of floating point numbers by functions such as **printf**, **wprintf**, and related functions in the Visual C++ Standard C library prints three digits for the exponent, even if three digits are not required to represent the value of the exponent. Zeroes are used to pad the value to three digits. **_set_output_format** allows you to change this behavior so that only two digits are printed in the exponent unless a third digit is required by the size of the exponent.

To enable two-digit exponents, call this function with the parameter **_TWO_DIGIT_EXPONENT**, as shown in the example. To disable two digit exponents, call this function with an argument of 0.

Requirements

Routine	Required header	Compatibility
_set_output_format	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

Example

```
// crt_set_output_format.c
#include <stdio.h>

void printvalues(double x, double y)
{
   printf("%11.4e %11.4e\n", x, y);
   printf("%11.4E %11.4E\n", x, y);
   printf("%11.4g %11.4g\n", x, y);
   printf("%11.4G %11.4G\n", x, y);
}


int main()
{
   double x = 1.211E-5;
   double y = 2.3056E-112;
   unsigned int old_exponent_format;
```

```
    // Use the default format
    printvalues(x, y);

    // Enable two-digit exponent format
    old_exponent_format = _set_output_format(_TWO_DIGIT_EXPONENT);

    printvalues(x, y);

    // Disable two-digit exponent format
    _set_output_format( old_exponent_format );

    printvalues(x, y);
  }
```

Output

```
1.2110e-005 2.3056e-112
1.2110E-005 2.3056E-112
 1.211e-005  2.306e-112
 1.211E-005  2.306E-112
 1.2110e-05 2.3056e-112
 1.2110E-05 2.3056E-112
  1.211e-05  2.306e-112
  1.211E-05  2.306E-112
1.2110e-005 2.3056e-112
1.2110E-005 2.3056E-112
 1.211e-005  2.306e-112
 1.211E-005  2.306E-112
```

See Also

Reference

printf, _printf_l, wprintf, _wprintf_l

printf Type Field Characters

_get_output_format

_set_printf_count_output

Enable or disable support of the **%n** format in printf, _printf_l, wprintf, _wprintf_l-family functions.

```
int _set_printf_count_output(
   int enable
);
```

Parameters

enable

A non-zero value to enable **%n** support, 0 to disable **%n** support.

Property Value/Return Value

The state of **%n** support before calling this function: non-zero if **%n** support was enabled, 0 if it was disabled.

Remarks

Because of security reasons, support for the **%n** format specifier is disabled by default in **printf** and all its variants. If **%n** is encountered in a **printf** format specification, the default behavior is to invoke the invalid parameter handler as described in Parameter Validation. Calling **_set_printf_count_output** with a non-zero argument will cause **printf**-family functions to interpret **%n** as described in printf Type Field Characters.

Requirements

Routine	Required header	Compatibility
_set_printf_count_output	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_set_printf_count_output.c
#include <stdio.h>

int main()
{
   int e;
   int i;
   e = _set_printf_count_output( 1 );
   printf( "%%n support was %sabled.\n",
        e ? "en" : "dis" );
   printf( "%%n support is now %sabled.\n",
        _get_printf_count_output() ? "en" : "dis" );
   printf( "12345%n6789\n", &i ); // %n format should set i to 5
   printf( "i = %d\n", i );
}
```

Output

```
%n support was disabled.
%n support is now enabled.
123456789
i = 5
```

NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

_get_printf_count_output

_set_purecall_handler, _set_purecall_handler_m

Sets the handler for a pure virtual function call.

```
_purecall_handler _set_purecall_handler(
      _purecall_handler function
);
_purecall_handler _set_purecall_handler_m(
      _purecall_handler_m function
);
```

Parameters

function

The function to be called when a pure virtual function is called. A **_purecall_handler** or **_purecall_handler_m** function should have a void return type.

Return Value

The previous **_purecall_handler**. Returns **NULL** if there was no previous handler.

Remarks

Use **_set_purecall_handler** if you want to catch pure virtual functions and report them to the user in a specific way or catch them for debugging purposes.

Because there is one **_purecall_handler** for the whole process, calling this function immediately impacts all threads. The last caller on any thread sets the handler.

There is a single **_set_purecall_handler** handler for all dynamically linked DLLs or executables; even if you call **_set_purecall_handler** your handler may be replaced by another or that you are replacing a handler set by another DLL or executable.

To restore default behavior, call **_set_purecall_handler** with a **NULL** argument.

The **_set_purecall_handler_m** version of the function is for mixed mode CRT.

Requirements

Routine	Required header	Compatibility
_set_purecall_handler, **_set_purecall_handler_m**	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// _set_purecall_handler.cpp
// compile with: /W1
#include <tchar.h>
#include <stdio.h>
#include <stdlib.h>

class CDerived;
class CBase
{
public:
   CBase(CDerived *derived): m_pDerived(derived) {};
   ~CBase();
   virtual void function(void) = 0;
```

```
   CDerived * m_pDerived;
};

class CDerived : public CBase
{
public:
   CDerived() : CBase(this) {};   // C4355
   virtual void function(void) {};
};

CBase::~CBase()
{
   m_pDerived -> function();
}

void myPurecallHandler(void)
{
   printf("In _purecall_handler.");
   exit(0);
}

int _tmain(int argc, _TCHAR* argv[])
{
   _set_purecall_handler(myPurecallHandler);
   CDerived myDerived;
}
```

Output

```
In _purecall_handler.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Error Handling (CRT)

_set_sbh_threshold

Sets the upper limit for the size of a memory allocation that will be supported by the small-block heap.

```
int _set_sbh_threshold(
   size_t size
);
```

Parameters

size

The new small-block threshold size to be set.

Return Value

Returns 1 if the operation of setting the small-block threshold size is successful. If the input threshold size is too big, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns 0.

Remarks

_set_sbh_threshold sets the current threshold value for the small-block heap. The default threshold size is 1016 bytes for Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition and Windows NT 4.0 and zero for Windows 2000 and later operating systems. By default, the small-block heap is not used on Windows 2000 and later operating systems, though **_set_sbh_threshold** can be called with a nonzero value to enable the small-block heap in those instances.

Requirements

Routine	Required header	Compatibility
_set_sbh_threshold	<malloc.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Memory Allocation

_get_sbh_threshold

_set_SSE2_enable

Enables or disables the use of Streaming SIMD Extensions 2 Instructions (SSE2) instructions in CRT math routines.

```
int _set_SSE2_enable(
   int flag
);
```

Parameters

flag

1 for enabling the SSE2 implementation. 0 for disabling the SSE2 implementation. By default, SSE2 implementation is enabled on processors that support it.

Return Value

Nonzero if SSE2 implementation is enabled; zero if SSE2 implementation is disabled.

Remarks

The following functions have SSE2 implementations that can be enabled with **_set_SSE2_enable**:

- atan
- ceil
- exp
- floor
- log
- log10
- modf
- pow

The SSE2 implementation might give slightly different answers from the default implementations of these functions, because SSE2 intermediate values are 64-bit floating-point quantities but the default implementation intermediate values are 80-bit floating-point quantities.

The SSE2 implementation is only used if all exceptions are masked. Use _control87, _controlfp to mask exceptions.

Requirements

Routine	Required header	Compatibility
_set_SSE2_enable	<math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_set_SSE2_enable.c
// processor: x86
#include <math.h>
#include <stdio.h>

int main()
{
   int i = _set_SSE2_enable(1);
```

```
    if (i)
       printf("SSE2 enabled.\n");
    else
       printf("SSE2 not enabled; processor does not support SSE2.\n");
 }
```

Output

```
SSE2 enabled.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

C Run-Time Libraries

_set_se_translator

Handles Win32 exceptions (C structured exceptions) as C++ typed exceptions.

```
_se_translator_function _set_se_translator(
   _se_translator_function seTransFunction
);
```

Parameters

seTransFunction

Pointer to a C structured exception translator function that you write.

Return Value

Returns a pointer to the previous translator function registered by **_set_se_translator**, so that the previous function can be restored later. If no previous function has been set, the return value can be used to restore the default behavior; this value can be NULL.

Remarks

The **_set_se_translator** function provides a way to handle Win32 exceptions (C structured exceptions) as C++ typed exceptions. To allow each C exception to be handled by a C++ **catch** handler, first define a C exception wrapper class that can be used, or derived from, to attribute a specific class type to a C exception. To use this class, install a custom C exception translator function that is called by the internal exception-handling mechanism each time a C exception is raised. Within your translator function, you can throw any typed exception that can be caught by a matching C++ **catch** handler.

You must use /EHa when using **_set_se_translator**.

To specify a custom translation function, call **_set_se_translator** with the name of your translation function as its argument. The translator function that you write is called once for each function invocation on the stack that has **try** blocks. There is no default translator function.

Your translator function should do no more than throw a C++ typed exception. If it does anything in addition to throwing (such as writing to a log file, for example) your program might not behave as expected, because the number of times the translator function is invoked is platform-dependent.

In a multithreaded environment, translator functions are maintained separately for each thread. Each new thread needs to install its own translator function. Thus, each thread is in charge of its own translation handling. **_set_se_translator** is specific to one thread; another DLL can install a different translation function.

The *seTransFunction* function that you write must be a native-compiled function (not compiled with /clr). It must take an unsigned integer and a pointer to a Win32 **_EXCEPTION_POINTERS** structure as arguments. The arguments are the return values of calls to the Win32 API **GetExceptionCode** and **GetExceptionInformation** functions, respectively.

```
typedef void (*_se_translator_function)(unsigned int, struct _EXCEPTION_POINTERS* );
```

For **_set_se_translator**, there are implications when dynamically linking to the CRT; another DLL in the process might call **_set_se_translator** and replace your handler with its own.

When using **_set_se_translator** from managed code (code compiled with /clr) or mixed native and managed code, be aware that the translator affects exceptions generated in native code only. Any managed exceptions generated in managed code (such as when raising **System::Exception**) are not routed through the translator function. Exceptions raised in managed code using the Win32 function **RaiseException** or caused by a system exception like a divide by zero exception are routed through the translator.

Requirements

Routine	Required header	Compatibility
_set_se_translator	<eh.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

The functionality provided by **_set_se_translator** is not available in code compiled with the /clr:pure compiler option.

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_settrans.cpp
// compile with: /EHa
#include <stdio.h>
#include <windows.h>
#include <eh.h>

void SEFunc();
void trans_func( unsigned int, EXCEPTION_POINTERS* );

class SE_Exception
{
private:
    unsigned int nSE;
public:
    SE_Exception() {}
    SE_Exception( unsigned int n ) : nSE( n ) {}
    ~SE_Exception() {}
    unsigned int getSeNumber() { return nSE; }
};
int main( void )
{
    try
    {
        _set_se_translator( trans_func );
        SEFunc();
    }
    catch( SE_Exception e )
    {
        printf( "Caught a __try exception with SE_Exception.\n" );
    }
}
void SEFunc()
{
    __try
    {
        int x, y=0;
        x = 5 / y;
    }
    __finally
    {
        printf( "In finally\n" );
    }
}
void trans_func( unsigned int u, EXCEPTION_POINTERS* pExp )
{
    printf( "In trans_func.\n" );
    throw SE_Exception();
}
```

Output

```
In trans_func.
In finally
Caught a __try exception with SE_Exception.
```

Although the functionality provided by **_set_se_translator** is not available in managed code, it is possible to use this mapping in native code, even if that native code is in a compilation under the **/clr** switch, as long as the native code is indicated using `#pragma unmanaged`. If a structured exception is being thrown in managed code that is to be mapped, the code that generates and handles the exception must be marked with the **pragma**. The following code shows a possible use. For more information, see Pragma Directives.

```
// crt_set_se_translator_clr.cpp
```

```
// compile with: /clr
#include <windows.h>
#include <eh.h>
#include <assert.h>
#include <stdio.h>

int thrower_func(int i) {
   int j = i/0;
  return 0;
}

class CMyException{
};

#pragma unmanaged
void my_trans_func(unsigned int u, PEXCEPTION_POINTERS pExp )
{
printf("Translating the structured exception to a C++"
              " exception.\n");
throw CMyException();
}

void DoTest()
{
    try
    {
      thrower_func(10);
    }

    catch(CMyException e)
    {
printf("Caught CMyException.\n");
    }
    catch(...)
    {
      printf("Caught unexpected SEH exception.\n");
    }
}
#pragma managed

int main(int argc, char** argv) {
    _set_se_translator(my_trans_func);
    DoTest();
    return 0;
}
```

Sample Output

```
Translating the structured exception to a C++ exception.
Caught CMyException.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Exception Handling Routines
set_terminate (CRT)
set_unexpected (CRT)
terminate (CRT)
unexpected (CRT)

set_terminate (CRT)

Installs your own termination routine to be called by **terminate**.

```
terminate_function set_terminate(
   terminate_function termFunction
);
```

Parameters

termFunction

Pointer to a terminate function that you write.

Return Value

Returns a pointer to the previous function registered by **set_terminate** so that the previous function can be restored later. If no previous function has been set, the return value may be used to restore the default behavior; this value may be NULL.

Remarks

The **set_terminate** function installs *termFunction* as the function called by **terminate**. **set_terminate** is used with C++ exception handling and may be called at any point in your program before the exception is thrown. **terminate** calls **abort** by default. You can change this default by writing your own termination function and calling **set_terminate** with the name of your function as its argument. **terminate** calls the last function given as an argument to **set_terminate**. After performing any desired cleanup tasks, *termFunction* should exit the program. If it does not exit (if it returns to its caller), **abort** is called.

In a multithreaded environment, terminate functions are maintained separately for each thread. Each new thread needs to install its own terminate function. Thus, each thread is in charge of its own termination handling.

The **terminate_function** type is defined in EH.H as a pointer to a user-defined termination function, *termFunction* that returns **void**. Your custom function *termFunction* can take no arguments and should not return to its caller. If it does, **abort** is called. An exception may not be thrown from within *termFunction*.

```
typedef void ( *terminate_function )( );
```

Note
The **set_terminate** function only works outside the debugger.

There is a single **set_terminate** handler for all dynamically linked DLLs or EXEs; even if you call **set_terminate** your handler may be replaced by another, or you may be replacing a handler set by another DLL or EXE.

This function is not supported under **/clr:pure**.

Requirements

Routine	Required header	Compatibility
set_terminate	<eh.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for terminate.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Exception Handling Routines
abort
_get_terminate
set_unexpected (CRT)
terminate (CRT)
unexpected (CRT)

set_unexpected (CRT)

Installs your own termination function to be called by **unexpected**.

```
unexpected_function set_unexpected(
   unexpected_function unexpFunction
);
```

Parameters

unexpFunction

Pointer to a function that you write to replace the **unexpected** function.

Return Value

Returns a pointer to the previous termination function registered by **_set_unexpected** so that the previous function can be restored later. If no previous function has been set, the return value may be used to restore the default behavior; this value may be NULL.

Remarks

The **set_unexpected** function installs *unexpFunction* as the function called by **unexpected**. **unexpected** is not used in the current C++ exception-handling implementation. The **unexpected_function** type is defined in EH.H as a pointer to a user-defined unexpected function, *unexpFunction* that returns **void**. Your custom *unexpFunction* function should not return to its caller.

```
typedef void ( *unexpected_function )( );
```

By default, **unexpected** calls **terminate**. You can change this default behavior by writing your own termination function and calling **set_unexpected** with the name of your function as its argument. **unexpected** calls the last function given as an argument to **set_unexpected**.

Unlike the custom termination function installed by a call to **set_terminate**, an exception can be thrown from within *unexpFunction*.

In a multithreaded environment, unexpected functions are maintained separately for each thread. Each new thread needs to install its own unexpected function. Thus, each thread is in charge of its own unexpected handling.

In the current Microsoft implementation of C++ exception handling, **unexpected** calls **terminate** by default and is never called by the exception-handling run-time library. There is no particular advantage to calling **unexpected** rather than **terminate**.

There is a single **set_unexpected** handler for all dynamically linked DLLs or EXEs; even if you call **set_unexpected** your handler may be replaced by another or that you are replacing a handler set by another DLL or EXE.

Requirements

Routine	Required header	Compatibility
set_unexpected	<eh.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Exception Handling Routines
abort
_get_unexpected

set_terminate (CRT)
terminate (CRT)
unexpected (CRT)

setbuf

Controls stream buffering. This function is deprecated; use setvbuf instead.

```
void setbuf(
   FILE *stream,
   char *buffer
);
```

Parameters

stream

Pointer to **FILE** structure.

buffer

User-allocated buffer.

Remarks

The **setbuf** function controls buffering for *stream*. The *stream* argument must refer to an open file that has not been read or written. If the *buffer* argument is **NULL**, the stream is un-buffered. If not, the buffer must point to a character array of length **BUFSIZ**, where **BUFSIZ** is the buffer size as defined in STDIO.H. The user-specified buffer, instead of the default system-allocated buffer for the given stream, is used for I/O buffering. The **stderr** stream is un-buffered by default, but you can use **setbuf** to assign buffers to **stderr**.

setbuf has been replaced by setvbuf, which is the preferred routine for new code. **setbuf** is retained for compatibility with existing code.

Requirements

Routine	Required header	Compatibility
setbuf	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_setbuf.c
// compile with: /W3
// This program first opens files named DATA1 and
// DATA2. Then it uses setbuf to give DATA1 a user-assigned
// buffer and to change DATA2 so that it has no buffer.

#include <stdio.h>

int main( void )
{
   char buf[BUFSIZ];
   FILE *stream1, *stream2;

   fopen_s( &stream1, "data1", "a" );
   fopen_s( &stream2, "data2", "w" );

   if( (stream1 != NULL) && (stream2 != NULL) )
   {
      // "stream1" uses user-assigned buffer:
      setbuf( stream1, buf ); // C4996
      // Note: setbuf is deprecated; consider using setvbuf instead
      printf( "stream1 set to user-defined buffer at: %Fp\n", buf );

      // "stream2" is unbuffered
      setbuf( stream2, NULL ); // C4996
```

```
        printf( "stream2 buffering disabled\n" );
        _fcloseall();
    }
}
```

Sample Output

```
stream1 set to user-defined buffer at: 0012FCDC
stream2 buffering disabled
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O
fclose, _fcloseall
fflush
fopen, _wfopen
setvbuf

setjmp

Saves the current state of the program.

```
int setjmp(
   jmp_buf env
);
```

Parameters

env

Variable in which environment is stored.

Return Value

Returns 0 after saving the stack environment. If **setjmp** returns as a result of a **longjmp** call, it returns the *value* argument of **longjmp**, or if the *value* argument of **longjmp** is 0, **setjmp** returns 1. There is no error return.

Remarks

The **setjmp** function saves a stack environment, which you can subsequently restore, using **longjmp**. When used together, **setjmp** and **longjmp** provide a way to execute a non-local **goto**. They are typically used to pass execution control to error-handling or recovery code in a previously called routine without using the normal calling or return conventions.

A call to **setjmp** saves the current stack environment in *env*. A subsequent call to **longjmp** restores the saved environment and returns control to the point just after the corresponding **setjmp** call. All variables (except register variables) accessible to the routine receiving control contain the values they had when **longjmp** was called.

It is not possible to use **setjmp** to jump from native to managed code.

Note **setjmp** and **longjmp** do not support C++ object semantics. In C++ programs, use the C++ exception-handling mechanism.

For more information, see Using setjmp and longjmp.

Requirements

Routine	Required header	Compatibility
setjmp	<setjmp.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for _fpreset.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Process and Environment Control

longjmp

setlocale, _wsetlocale

Define the locale.

```
char *setlocale(
   int category,
   const char *locale
);
wchar_t *_wsetlocale(
   int category,
   const wchar_t *locale
);
```

Parameters

category

Category affected by locale.

locale

Locale name.

Return Value

If a valid locale and category are given, returns a pointer to the string associated with the specified locale and category. If the locale or category is invalid, returns a null pointer and the current locale settings of the program are not changed.

For example, the call

```
setlocale( LC_ALL, "English" );
```

sets all categories, returning only the string

```
English_United States.1252
```

If all categories are not explicitly set by a call to **setlocale**, the function returns a string indicating the current setting of each of the categories, separated by semicolons. If the *locale* argument is a null pointer, **setlocale** returns a pointer to the string associated with the *category* of the program's locale; the program's current locale setting is not changed.

The null pointer is a special directive that tells **setlocale** to query rather than set the international environment. For example, the sequence of calls

```
// Set all categories and return " English_United States.1252"
setlocale( LC_ALL, "English" );
// Set only the LC_MONETARY category and return "French_France.1252"
setlocale( LC_MONETARY, "French" );
setlocale( LC_ALL, NULL );
```

returns

```
LC_COLLATE= English_United States.1252;
LC_CTYPE= English_United States.1252;
LC_MONETARY=French_France.1252;
LC_NUMERIC= English_United States.1252;
LC_TIME= English_United States.1252
```

which is the string associated with the **LC_ALL** category.

One can use the string pointer returned by **setlocale** in subsequent calls to restore that part of the program's locale information, assuming that your program does not alter the pointer or the string. Later calls to **setlocale** overwrite the string; you can use _strdup to save a specific locale string.

Remarks

Use the **setlocale** function to set, change, or query some or all of the current program locale information specified by *locale*

and *category*. *locale* refers to the locality (country/region and language) for which you can customize certain aspects of your program. Some locale-dependent categories include the formatting of dates and the display format for monetary values. If you set *locale* to the default string for a language with multiple forms supported on your computer, you should check the **setlocale** return code to see which language is in effect. For example, using "chinese" could result in a return value of **chinese-simplified** or **chinese-traditional**.

_wsetlocale is a wide-character version of **setlocale**; the *locale* argument and return value of **_wsetlocale** are wide-character strings. **_wsetlocale** and **setlocale** behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tsetlocale	**setlocale**	**setlocale**	**_wsetlocale**

The *category* argument specifies the parts of a program's locale information that are affected. The macros used for *category* and the parts of the program they affect are as follows:

LC_ALL

All categories, as listed below.

LC_COLLATE

The **strcoll**, **_stricoll**, **wcscoll**, **_wcsicoll**, **strxfrm**, **_strncoll**, **_strnicoll**, **_wcsncoll**, **_wcsnicoll**, and **wcsxfrm** functions.

LC_CTYPE

The character-handling functions (except **isdigit**, **isxdigit**, **mbstowcs**, and **mbtowc**, which are unaffected).

LC_MONETARY

Monetary-formatting information returned by the **localeconv** function.

LC_NUMERIC

Decimal-point character for the formatted output routines (such as **printf**), for the data-conversion routines, and for the non-monetary formatting information returned by **localeconv**. In addition to the decimal-point character, **LC_NUMERIC** also sets the thousands separator and the grouping control string returned by localeconv.

LC_TIME

The **strftime** and **wcsftime** functions.

This function validates the category parameter. If the category parameter is not one of the values given in the above table, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function sets **errno** to **EINVAL** and returns **NULL**.

The *locale* argument is a pointer to a string that specifies the name of the locale. If *locale* points to an empty string, the locale is the implementation-defined native environment. A value of *C* specifies the minimal ANSI conforming environment for C translation. The *C* locale assumes that all **char** data types are 1 byte and that their value is always less than 256.

At program startup, the equivalent of the following statement is executed:

```
setlocale( LC_ALL, "C" );
```

The *locale* argument takes the following form:

```
locale :: "lang[_country_region[.code_page]]"
| ".code_page"
| ""
| NULL
```

The set of available languages, country/region codes, and code pages includes all those supported by the Win32 NLS API except code pages that require more than two bytes per character, such as UTF-7 and UTF-8. If you provide a code page like UTF-7 or UTF-8, **setlocale** will fail, returning NULL. The set of language and country/region codes supported by **setlocale** is listed in Language and Country/Region Strings.

If *locale* is a null pointer, **setlocale** queries, rather than sets, the international environment and returns a pointer to the string associated with the specified *category*. The program's current locale setting is not changed. For example,

```
setlocale( LC_ALL, NULL );
```

returns the string associated with *category*.

The following examples pertain to the **LC_ALL** category. Either of the strings ".OCP" and ".ACP" can be used in place of a code page number to specify use of the user-default OEM code page and user-default ANSI code page, respectively.

`setlocale( LC_ALL, "" );`

Sets the locale to the default, which is the user-default ANSI code page obtained from the operating system.

`setlocale( LC_ALL, ".OCP" );`

Explicitly sets the locale to the current OEM code page obtained from the operating system.

`setlocale( LC_ALL, ".ACP" );`

Sets the locale to the ANSI code page obtained from the operating system.

`setlocale( LC_ALL, "[lang_ctry]" );`

Sets the locale to the language and country/region indicated, using the default code page obtained from the host operating system.

`setlocale( LC_ALL, "[lang_ctry.cp]" );`

Sets the locale to the language, country/region, and code page indicated in the [*lang_ctry.cp*] string. You can use various combinations of language, country/region, and code page. For example:

```
setlocale( LC_ALL, "French_Canada.1252" );
// Set code page to French Canada ANSI default
setlocale( LC_ALL, "French_Canada.ACP" );
// Set code page to French Canada OEM default
setlocale( LC_ALL, "French_Canada.OCP" );
```

`setlocale( LC_ALL, "[lang]" );`

Sets the locale to the country/region indicated, using the default country/region for the language specified, and the user-default ANSI code page for that country/region as obtained from the host operating system. For example, the following two calls to **setlocale** are functionally equivalent:

```
setlocale( LC_ALL, "English" );
setlocale( LC_ALL, "English_United States.1252" );
```

`setlocale( LC_ALL, "[.code_page]" );`

Sets the code page to the value indicated, using the default country/region and language (as defined by the host operating system) for the specified code page.

The category must be either **LC_ALL** or **LC_CTYPE** to effect a change of code page. For example, if the default country/region and language of the host operating system are "United States" and "English," the following two calls to **setlocale** are functionally equivalent:

`setlocale( LC_ALL, ".1252" );`

`setlocale( LC_ALL, "English_United States.1252");`

For more information see the setlocale pragma in the *Preprocessor Reference*.

The function _configthreadlocale is used to control whether **setlocale** affects the locale of all threads in a program or only the locale of the calling thread.

Requirements

Routine	**Required header**	**Compatibility**
setlocale	<locale.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wsetlocale	<locale.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_setlocale.cpp
//
// This program demonstrates the use of setlocale when
// using two independent threads.
//

#include <locale.h>
#include <process.h>
#include <windows.h>
#include <stdio.h>
#include <time.h>

#define BUFF_SIZE 100

// Retrieve the date and time in the current
// locale's format.
int get_time(unsigned char* str)
{
    __time64_t ltime;
    struct tm  thetime;

    // Retieve the time
    _time64(&ltime);
    _gmtime64_s(&thetime, &ltime);

    // Format the current time structure into a string
    // using %#x is the long date representation,
    // appropriate to the current locale
    if (!strftime((char *)str, BUFF_SIZE, "%#x",
                  (const struct tm *)&thetime))
    {
        printf("strftime failed!\n");
        return -1;
    }
    return 0;
}

// This thread sets its locale to German
// and prints the time.
unsigned __stdcall SecondThreadFunc( void* pArguments )
{
    unsigned char str[BUFF_SIZE];

    // Set the thread local
    printf("The thread locale is now set to %s.\n",
           setlocale(LC_ALL, "German"));

    // Retrieve the time string from the helper function
    if (get_time(str) == 0)
    {
        printf("The time in German locale is: '%s'\n", str);
    }

    _endthreadex( 0 );
    return 0;
}

// The main thread spawns a second thread (above) and then
// sets the locale to English and prints the time.
int main()
{
    HANDLE          hThread;
    unsigned        threadID;
    unsigned char   str[BUFF_SIZE];
```

```
    // Configure per-thread locale to cause all subsequently created
    // threads to have their own locale.
    _configthreadlocale(_ENABLE_PER_THREAD_LOCALE);

    // Retrieve the time string from the helper function
    printf("The thread locale is now set to %s.\n",
           setlocale(LC_ALL, "English"));

    // Create the second thread.
    hThread = (HANDLE)_beginthreadex( NULL, 0, &SecondThreadFunc,
                                      NULL, 0, &threadID );

    if (get_time(str) == 0)
    {
        // Retrieve the time string from the helper function
        printf("The time in English locale is: '%s'\n\n", str);
    }

    // Wait for the created thread to finish.
    WaitForSingleObject( hThread, INFINITE );

    // Destroy the thread object.
    CloseHandle( hThread );
}
```

Sample Output

```
The thread locale is now set to English_United States.1252.
The time in English locale is: 'Wednesday, May 12, 2004'

The thread locale is now set to German_Germany.1252.
The time in German locale is: 'Mittwoch, 12. Mai 2004'
```

.NET Framework Equivalent

System::Globalization::CultureInfo Class

See Also

Reference

_configthreadlocale
Locale
localeconv
_mbclen, mblen, _mblen_l
strlen, strlen_l, wcslen, wcslen_l, _mbslen, _mbslen_l, _mbstrlen, _mbstrlen_l
mbstowcs, _mbstowcs_l
mbtowc, _mbtowc_l
_setmbcp
strcoll Functions
strftime, wcsftime, _strftime_l, _wcsftime_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l
wcstombs, _wcstombs_l
wctomb, _wctomb_l

_setmaxstdio

Sets a maximum for the number of simultaneously open files at the **stdio** level.

```
int _setmaxstdio(
   int newmax
);
```

Parameters

newmax

New maximum for the number of simultaneously open files at the **stdio** level.

Return Value

Returns *newmax* if successful; –1 otherwise.

If *newmax* is less than **_IOB_ENTRIES** or greater then the maximum number of handles available in the operating system, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function returns -1 and sets **errno** to **EINVAL**.

For information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_setmaxstdio** function changes the maximum value for the number of files that might be simultaneously open at the **stdio** level.

C run-time I/O now supports many more open files on Win32 platforms than in previous versions. Up to 2,048 files can be open simultaneously at the lowio level (that is, opened and accessed by means of the **_open**, **_read**, **_write**, and so forth family of I/O functions). Up to 512 files can be open simultaneously at the stdio level (that is, opened and accessed by means of the **fopen**, **fgetc**, **fputc**, and so forth family of functions). The limit of 512 open files at the **stdio** level can be increased to a maximum of 2,048 by means of the **_setmaxstdio** function.

Because **stdio**-level functions, such as **fopen**, are built on top of the **lowio** functions, the maximum of 2,048 is a hard upper limit for the number of simultaneously open files accessed through the C run-time library.

Note
This upper limit might be beyond what is supported by a particular Win32 platform and configuration.

Requirements

Routine	**Required header**	**Compatibility**
_setmaxstdio	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See _getmaxstdio for an example of using **_setmaxstdio**.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O

_setmbcp

Sets a new multibyte code page.

```
int _setmbcp(
   int codepage
);
```

Parameters

codepage

New code page setting for locale-independent multibyte routines.

Return Value

Returns 0 if the code page is set successfully. If an invalid code page value is supplied for *codepage*, returns –1 and the code page setting is unchanged. Sets **errno** to **EINVAL** if a memory allocation failure occurs.

Remarks

The **_setmbcp** function specifies a new multibyte code page. By default, the run-time system automatically sets the multibyte code page to the system-default ANSI code page. The multibyte code page setting affects all multibyte routines that are not locale dependent. However, it is possible to instruct **_setmbcp** to use the code page defined for the current locale (see the following list of manifest constants and associated behavior results). For a list of the multibyte routines that are dependent on the locale code page rather than the multibyte code page, see Interpretation of Multibyte-Character Sequences.

The multibyte code page also affects multibyte-character processing by the following run-time library routines:

_exec functions	_mktemp	_stat
_fullpath	_spawn functions	_tempnam
_makepath	_splitpath	tmpnam

In addition, all run-time library routines that receive multibyte-character *argv* or *envp* program arguments as parameters (such as the **_exec** and **_spawn** families) process these strings according to the multibyte code page. Therefore, these routines are also affected by a call to **_setmbcp** that changes the multibyte code page.

The *codepage* argument can be set to any of the following values:

- **_MB_CP_ANSI** Use ANSI code page obtained from operating system at program startup.
- **_MB_CP_LOCALE** Use the current locale's code page obtained from a previous call to setlocale.
- **_MB_CP_OEM** Use OEM code page obtained from operating system at program startup.
- **_MB_CP_SBCS** Use single-byte code page. When the code page is set to **_MB_CP_SBCS**, a routine such as _ismbblead always returns false.
- Any other valid code page value, regardless of whether the value is an ANSI, OEM, or other operating-system-supported code page (except UTF-7 and UTF-8, which are not supported).

Requirements

Routine	Required header	Compatibility
_setmbcp	<mbctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

See Also

Reference

_getmbcp

setlocale, _wsetlocale

setmode

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _setmode instead.

_setmode

Sets the file translation mode.

```
int _setmode (
   int fd,
      int mode
);
```

Parameters

fd

File descriptor.

mode

New translation mode.

Return Value

If successful, returns the previous translation mode.

If invalid parameters are passed to this function, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function returns –1 and sets **errno** to either **EBADF**, indicating an invalid file descriptor, or **EINVAL**, indicating an invalid *mode* argument (neither **_O_TEXT** nor **_O_BINARY**).

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_setmode** function sets to *mode* the translation mode of the file given by *fd*. The mode must be one of two manifest constants: **_O_TEXT** or **_O_BINARY**. **_O_TEXT** sets text (translated) mode. Carriage return–line feed (CR-LF) combinations are translated into a single line feed character on input. Line feed characters are translated into CR-LF combinations on output. **_O_BINARY** sets binary (untranslated) mode, in which these translations are suppressed.

_setmode is typically used to modify the default translation mode of **stdin** and **stdout**, but you can use it on any file. If you apply **_setmode** to the file descriptor for a stream, call **_setmode** before performing any input or output operations on the stream.

Requirements

Routine	Required header	Optional Headers	Compatibility
_setmode	<io.h>	<fcntl.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_setmode.c
// This program uses _setmode to change
// stdin from text mode to binary mode.


#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int main( void )
{
   int result;
```

```
    // Set "stdin" to have binary mode:
    result = _setmode( _fileno( stdin ), _O_BINARY );
    if( result == -1 )
       perror( "Cannot set mode" );
    else
       printf( "'stdin' successfully changed to binary mode\n" );
 }
```

Output

```
'stdin' successfully changed to binary mode
```

.NET Framework Equivalent

- System::IO::BinaryReader Class
- System::IO::TextReader Class

See Also

Reference

File Handling
_creat, _wcreat
fopen, _wfopen
_open, _wopen
_set_fmode

setvbuf

Controls stream buffering and buffer size.

```
int setvbuf(
   FILE *stream,
   char *buffer,
   int mode,
   size_t size
);
```

Parameters

stream

Pointer to **FILE** structure.

buffer

User-allocated buffer.

mode

Mode of buffering.

size

Buffer size in bytes. Allowable range: 2 <= *size* <= INT_MAX (2147483647). Internally, the value supplied for *size* is rounded down to the nearest multiple of 2.

Return Value

Returns 0 if successful.

If *stream* or *buffer* are a **NULL** pointer, or if *mode* or *size* is not within a valid change, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function returns -1 and sets **errno** to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **setvbuf** function allows the program to control both buffering and buffer size for *stream*. *stream* must refer to an open file that has not undergone an I/O operation since it was opened. The array pointed to by *buffer* is used as the buffer, unless it is **NULL**, in which case **setvbuf** uses an automatically allocated buffer of length *size*/2 * 2 bytes.

The mode must be **_IOFBF**, **_IOLBF**, or **_IONBF**. If *mode* is **_IOFBF** or **_IOLBF**, then *size* is used as the size of the buffer. If *mode* is **_IONBF**, the stream is unbuffered and *size* and *buffer* are ignored. Values for *mode* and their meanings are:

_IOFBF

Full buffering; that is, *buffer* is used as the buffer and *size* is used as the size of the buffer. If *buffer* is **NULL**, an automatically allocated buffer *size* bytes long is used.

_IOLBF

For some systems, this provides line buffering. However, for Win32, the behavior is the same as **_IOFBF** - Full Buffering.

_IONBF

No buffer is used, regardless of *buffer* or *size*.

Requirements

Routine	Required header	Compatibility
setvbuf	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_setvbuf.c
// This program opens two streams: stream1
// and stream2. It then uses setvbuf to give stream1 a
// user-defined buffer of 1024 bytes and stream2 no buffer.
//

#include <stdio.h>

int main( void )
{
   char buf[1024];
   FILE *stream1, *stream2;

   if( fopen_s( &stream1, "data1", "a" ) == 0 &&
       fopen_s( &stream2, "data2", "w" ) == 0 )
   {
      if( setvbuf( stream1, buf, _IOFBF, sizeof( buf ) ) != 0 )
         printf( "Incorrect type or size of buffer for stream1\n" );
      else
         printf( "'stream1' now has a buffer of 1024 bytes\n" );
      if( setvbuf( stream2, NULL, _IONBF, 0 ) != 0 )
         printf( "Incorrect type or size of buffer for stream2\n" );
      else
         printf( "'stream2' now has no buffer\n" );
      _fcloseall();
   }
}
```

Output

```
'stream1' now has a buffer of 1024 bytes
'stream2' now has no buffer
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O
fclose, _fcloseall
fflush
fopen, _wfopen
setbuf

signal

Sets interrupt signal handling.

```
void (__cdecl *signal(
   int sig,
   void (__cdecl *func ) (int [, int ] )))
   (int);
```

Parameters

sig

Signal value.

func

Function to be executed. The first parameter is a signal value and the second parameter is a sub-code that can be used when the first parameter is SIGFPE.

Return Value

signal returns the previous value of *func* associated with the given signal. For example, if the previous value of *func* was **SIG_IGN**, the return value is also **SIG_IGN**. A return value of **SIG_ERR** indicates an error, in which case **errno** is set to **EINVAL**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on this, and other, return codes.

Remarks

The **signal** function allows a process to choose one of several ways to handle an interrupt signal from the operating system. The *sig* argument is the interrupt to which **signal** responds; it must be one of the following manifest constants, defined in SIGNAL.H.

***sig* value**	**Description**
SIGABRT	Abnormal termination
SIGFPE	Floating-point error
SIGILL	Illegal instruction
SIGINT	CTRL+C signal
SIGSEGV	Illegal storage access
SIGTERM	Termination request

If *sig* is not one of the above values, the invalid parameter handler is invoked, as defined in Parameter Validation . If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **SIG_ERR**.

By default, **signal** terminates the calling program with exit code 3, regardless of the value of *sig*.

Note
SIGINT is not supported for any Win32 application. When a CTRL+C interrupt occurs, Win32 operating systems generate a new thread to specifically handle that interrupt. This can cause a single-thread application such as one in UNIX to become multithreaded, resulting in unexpected behavior.

The *func* argument is an address to a signal handler that you write, or one of the predefined constants **SIG_DFL** or **SIG_IGN**, also defined in SIGNAL.H. If *func* is a function, it is installed as the signal handler for the given signal. The signal handler's prototype requires one formal argument, *sig*, of type **int**. The operating system provides the actual argument through *sig* when an interrupt occurs; the argument is the signal that generated the interrupt. Thus you can use the six manifest constants (listed

in the preceding table) inside your signal handler to determine which interrupt occurred and take appropriate action. For example, you can call **signal** twice to assign the same handler to two different signals, then test the *sig* argument inside the handler to take different actions based on the signal received.

If you are testing for floating-point exceptions (**SIGFPE**), *func* points to a function that takes an optional second argument that is one of several manifest constants defined in FLOAT.H of the form **FPE_xxx**. When a **SIGFPE** signal occurs, you can test the value of the second argument to determine the type of floating-point exception and then take appropriate action. This argument and its possible values are Microsoft extensions.

For floating-point exceptions, the value of *func* is not reset upon receiving the signal. To recover from floating-point exceptions, use try/except clauses surrounding the floating point operations. It is also possible to recover using setjmp with longjmp. In either case, the calling process resumes execution with the floating-point state of the process left undefined.

If the signal handler returns, the calling process resumes execution immediately following the point at which it received the interrupt signal. This is true regardless of the type of signal or operating mode.

Before the specified function is executed, the value of *func* is set to **SIG_DFL**. The next interrupt signal is treated as described for **SIG_DFL**, unless an intervening call to **signal** specifies otherwise. This feature lets you reset signals in the called function.

Because signal-handler routines are usually called asynchronously when an interrupt occurs, your signal-handler function may get control when a run-time operation is incomplete and in an unknown state. The list below summarizes restrictions that determine which functions you can use in your signal-handler routine.

- Do not issue low-level or STDIO.H I/O routines (such as **printf** and **fread**).
- Do not call heap routines or any routine that uses the heap routines (such as **malloc**, **_strdup**, and **_putenv**). See malloc for more information.
- Do not use any function that generates a system call (e.g., **_getcwd**, **time**).
- Do not use **longjmp** unless the interrupt is caused by a floating-point exception (i.e., *sig* is **SIGFPE**). In this case, first reinitialize the floating-point package with a call to **_fpreset**.
- Do not use any overlay routines.

A program must contain floating-point code if it is to trap the **SIGFPE** exception with the function. If your program does not have floating-point code and requires the run-time library's signal-handling code, simply declare a volatile double and initialize it to zero:

```
volatile double d = 0.0f;
```

The **SIGILL** and **SIGTERM** signals are not generated under Windows NT. They are included for ANSI compatibility. Thus you can set signal handlers for these signals with **signal**, and you can also explicitly generate these signals by calling raise.

Signal settings are not preserved in spawned processes created by calls to **_exec** or **_spawn** functions. The signal settings are reset to the default in the new process.

Requirements

Routine	Required header	Compatibility
signal	<signal.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

The following example shows the use of **signal** to add some custom behavior to the **SIGABRT** signal. For additional information on abort behavior, see _set_abort_behavior.

```
// crt_signal.c
// compile with: /c
// Use signal to attach a signal handler to the abort routine
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
```

```
#include <tchar.h>

void SignalHandler(int signal)
{
    printf("Application aborting...\n");
}

int main()
{
    typedef void (*SignalHandlerPointer)(int);

    SignalHandlerPointer previousHandler;
    previousHandler = signal(SIGABRT, SignalHandler);

    abort();
}
```

Output

```
This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
Application aborting...
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Process and Environment Control
abort
_exec, _wexec Functions
exit, _exit
_fpreset
_spawn, _wspawn Functions

sin, sinf, sinh, sinhf

Calculate sines and hyperbolic sines.

```
double sin(
   double x
);
float sin(
   float x
);  // C++ only
long double sin(
   long double x
);  // C++ only
float sinf(
   float x
);
double sinh(
   double x
);
float sinh(
   float x
);  // C++ only
long double sinh(
   long double x
);  // C++ only
float sinhf(
   float x
);
```

Parameters

x

Angle in radians.

Return Value

sin returns the sine of *x*. If *x* is greater than or equal to 263, or less than or equal to –263, a loss of significance in the result occurs

sinh returns the hyperbolic sine of *x*. If the result is too large, **sinh** sets **errno** to **ERANGE** and returns ±**HUGE_VAL**, by default.

Input	SEH Exception	Matherr Exception
± QNAN,IND	None	_DOMAIN
± ∞ (sin, sinf)	INVALID	_DOMAIN
|x| ≥ 7.104760e+002 (sinh, sinhf)	OVERFLOW+INEXACT	OVERFLOW

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

Remarks

C++ allows overloading, so users can call overloads of **sin** and **sinh** that take double, float or long double types. In a C program, the **sin** and **sinh** functions always take and return double and float, respectively.

Requirements

Routine	Required header	Compatibility
sin, **sinf**, **sinh**, **sinhf**	<math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_sincos.c
// This program displays the sine, hyperbolic
// sine, cosine, and hyperbolic cosine of pi / 2.
//

#include <math.h>
#include <stdio.h>

int main( void )
{
   double pi = 3.1415926535;
   double x, y;

   x = pi / 2;
   y = sin( x );
   printf( "sin( %f ) = %f\n", x, y );
   y = sinh( x );
   printf( "sinh( %f ) = %f\n",x, y );
   y = cos( x );
   printf( "cos( %f ) = %f\n", x, y );
   y = cosh( x );
   printf( "cosh( %f ) = %f\n",x, y );
}
```

Output

```
sin( 1.570796 ) = 1.000000
sinh( 1.570796 ) = 2.301299
cos( 1.570796 ) = 0.000000
cosh( 1.570796 ) = 2.509178
```

.NET Framework Equivalent

- System::Math::Sin
- System::Math::Sinh

See Also

Reference

Floating-Point Support
acos, acosf
asin, asinf
atan, atanf, atan2, atan2f
cos, cosf, cosh, coshf
tan, tanf, tanh, tanhf

Size and Distance Specification

The optional prefixes to *type*, **h**, **l**, **I**, **I32**, **I64**, and **ll** specify the "size" of *argument* (long or short, 32- or 64-bit, single-byte character or wide character, depending upon the type specifier that they modify). These type-specifier prefixes are used with type characters in **printf** functions or **wprintf** functions to specify interpretation of arguments, as shown in the following table. These prefixes are Microsoft extensions and are not ANSI-compatible.

Note
The **h** and **l** prefixes are Microsoft extensions when used with data of type **char**.

Size Prefixes for printf and wprintf Format-Type Specifiers

To specify	Use prefix	With type specifier
long int	**l (lowercase L)**	**d**, **i**, **o**, **x**, or **X**
long unsigned int	**l**	**o**, **u**, **x**, or **X**
long long	**ll**	**d**, **i**, **o**, **x**, or **X**
short int	**h**	**d**, **i**, **o**, **x**, or **X**
short unsigned int	**h**	**o**, **u**, **x**, or **X**
__int32	**I32**	**d**, **i**, **o**, **x**, or **X**
unsigned __int32	**I32**	**o**, **u**, **x**, or **X**
__int64	**I64**	**d**, **i**, **o**, **x**, or **X**
unsigned __int64	**I64**	**o**, **u**, **x**, or **X**
ptrdiff_t (that is, **__int32** on 32-bit platforms, **__int64** on 64-bit platforms)	**l**	**d**, **i**, **o**, **x**, or **X**
size_t (that is, **unsigned __int32** on 32-bit platforms, **unsigned __int64** on 64-bit platforms)	**l**	**o**, **u**, **x**, or **X**
long double	**l** or **L**	**f**
Single-byte character with **printf** functions	**h**	**c** or **C**
Single-byte character with **wprintf** functions	**h**	**c** or **C**
Wide character with **printf** functions	**l**	**c** or **C**
Wide character with **wprintf** functions	**l**	**c** or **C**
Single-byte – character string with **printf** functions	**h**	**s** or **S**
Single-byte – character string with **wprintf** functions	**h**	**s** or **S**
Wide-character string with **printf** functions	**l**	**s** or **S**

Wide-character string with **wprintf** functions	**l**	**s** or **S**
Wide character	**w**	**c**
Wide-character string	**w**	**s**

Thus to print single-byte or wide-characters with **printf** functions and **wprintf** functions, use format specifiers as follows.

To print character as	Use function	With format specifier
single byte	**printf**	**c**, **hc**, or **hC**
single byte	**wprintf**	**C**, **hc**, or **hC**
wide	**wprintf**	**c**, **lc**, **lC**, or **wc**
wide	**printf**	**C**, **lc**, **lC**, or **wc**

To print strings with **printf** functions and **wprintf** functions, use the prefixes **h** and **l** analogously with format type-specifiers **s** and **S**.

See Also

Reference

printf, _printf_l, wprintf, _wprintf_l

_snprintf, _snprintf_l, _snwprintf, _snwprintf_l

Writes formatted data to a string. These functions are deprecated because more secure versions are available; see _snprintf_s, _snprintf_s_l, _snwprintf_s, _snwprintf_s_l.

```
int _snprintf(
   char *buffer,
   size_t count,
   const char *format [,
      argument] ...
);
int _snprintf_l(
   char *buffer,
   size_t count,
   const char *format,
   locale_t locale [,
      argument] ...
);
int _snwprintf(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format [,
      argument] ...
);
int _snwprintf_l(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format,
   locale_t locale [,
      argument] ...
);
template <size_t size>
int _snprintf(
   char (&buffer)[size],
   size_t count,
   const char *format [,
      argument] ...
); // C++ only
template <size_t size>
int _snprintf_l(
   char (&buffer)[size],
   size_t count,
   const char *format,
   locale_t locale [,
      argument] ...
); // C++ only
template <size_t size>
int _snwprintf(
   wchar_t (&buffer)[size],
   size_t count,
   const wchar_t *format [,
      argument] ...
); // C++ only
template <size_t size>
int _snwprintf_l(
   wchar_t (&buffer)[size],
   size_t count,
   const wchar_t *format,
   locale_t locale [,
      argument] ...
); // C++ only
```

Parameters

buffer

Storage location for the output.

count

Maximum number of characters to store.

format

Format-control string.

argument

Optional arguments.

locale

The locale to use.

Return Value

Let *len* be the length of the formatted data string (not including the terminating null). *len* and *count* are in bytes for **_snprintf**, wide characters for **_snwprintf**.

If *len* < *count*, then *len* characters are stored in *buffer*, a null-terminator is appended, and *len* is returned.

If *len* = *count*, then *len* characters are stored in *buffer*, no null-terminator is appended, and *len* is returned.

If *len* > *count*, then *count* characters are stored in *buffer*, no null-terminator is appended, and a negative value is returned.

If *buffer* is a null pointer and *count* is nonzero, or *format* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

For information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_snprintf** function formats and stores *count* or fewer characters in *buffer*, and appends a terminating null character if the formatted string length is strictly less than *count* characters. Each *argument* (if any) is converted and output according to the corresponding format specification in *format*. The format consists of ordinary characters and has the same form and function as the *format* argument for printf. If copying occurs between strings that overlap, the behavior is undefined.

Security Note
Ensure that *format* is not a user-defined string. Because this function does not guarantee NULL termination (in particular, wh en the return value is *count*), ensure that it is followed by code that adds the null terminator. For more information, see Avoiding Buffer Overruns.

_snwprintf is a wide-character version of **_snprintf**; the pointer arguments to **_snwprintf** are wide-character strings. Detection of encoding errors in **_snwprintf** might differ from that in **_snprintf**. **_snwprintf**, like **swprintf**, writes output to a string rather than to a destination of type **FILE**.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

Tchar.h routine	**_UNICODE and _MBCS not defined**	**_MBCS defined**	**_UNICODE defined**
_sntprintf	**_snprintf**	**_snprintf**	**_snwprintf**
_sntprintf_l	**_snprintf_l**	**_snprintf_l**	**_snwprintf_l**

Requirements

Routine	**Required header**	**Compatibility**
_snprintf, **_snprintf_l**	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_snwprintf, **_snwprintf_l**	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_snprintf.c
// compile with: /W3
#include <stdio.h>
#include <stdlib.h>

#if !defined(__cplusplus)
typedef int bool;
const bool true = 1;
const bool false = 0;
#endif

#define FAIL 0 // change to 1 and see what happens

int main(void)
{
   char buffer[200];
   const static char s[] = "computer"
#if FAIL
"computercomputercomputercomputercomputercomputercomputercomputercomputer"
"computercomputercomputercomputercomputercomputercomputercomputercomputer"
"computercomputercomputercomputercomputercomputercomputercomputercomputer"
"computercomputercomputercomputercomputercomputercomputercomputercomputer"
#endif
   ;
   const char c = 'l';
   const int i = 35;
#if FAIL
   const double fp = 1e300; // doesn't fit in the buffer
#else
   const double fp = 1.7320534;
#endif
   /* !subtract one to prevent "squeezing out" the terminal nul! */
   const int bufferSize = sizeof(buffer)/sizeof(buffer[0]) - 1;
   int bufferUsed = 0;
   int bufferLeft = bufferSize - bufferUsed;
   bool bSuccess = true;
   buffer[0] = 0;

   /* Format and print various data: */

   if (bufferLeft > 0)
   {
      int perElementBufferUsed = _snprintf(&buffer[bufferUsed],
      bufferLeft, "   String: %s\n", s ); // C4996
      // Note: _snprintf is deprecated; consider _snprintf_s instead
      if (bSuccess = (perElementBufferUsed >= 0))
      {
         bufferUsed += perElementBufferUsed;
         bufferLeft -= perElementBufferUsed;
         if (bufferLeft > 0)
         {
            int perElementBufferUsed = _snprintf(&buffer[bufferUsed],
            bufferLeft, "   Character: %c\n", c ); // C4996
            if (bSuccess = (perElementBufferUsed >= 0))
            {
               bufferUsed += perElementBufferUsed;
               bufferLeft -= perElementBufferUsed;
               if (bufferLeft > 0)
               {
```

```
                    int perElementBufferUsed = _snprintf(&buffer
                    [bufferUsed], bufferLeft, "   Integer: %d\n", i ); // C4996
                    if (bSuccess = (perElementBufferUsed >= 0))
                    {
                       bufferUsed += perElementBufferUsed;
                       bufferLeft -= perElementBufferUsed;
                       if (bufferLeft > 0)
                       {
                          int perElementBufferUsed = _snprintf(&buffer
                          [bufferUsed], bufferLeft, "   Real: %f\n", fp ); // C4996
                          if (bSuccess = (perElementBufferUsed >= 0))
                          {
                             bufferUsed += perElementBufferUsed;
                          }
                       }
                    }
                 }
              }
           }
        }
     }

     if (!bSuccess)
     {
        printf("%s\n", "failure");
     }
     else
     {
        /* !store nul because _snprintf doesn't necessarily (if the string
         * fits without the terminal nul, but not with it)!
         * bufferUsed might be as large as bufferSize, which normally is
         * like going one element beyond a buffer, but in this case
         * subtracted one from bufferSize, so we're ok.
         */
        buffer[bufferUsed] = 0;
        printf( "Output:\n%s\ncharacter count = %d\n", buffer, bufferUsed );
     }
     return EXIT_SUCCESS;
  }
```

Output

```
Output:
   String: computer
   Character: l
   Integer: 35
   Real: 1.732053

character count = 69
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
scanf, _scanf_l, wscanf, _wscanf_l
sscanf, _sscanf_l, swscanf, _swscanf_l
vprintf Functions

_snprintf_s, _snprintf_s_l, _snwprintf_s, _snwprintf_s_l

Writes formatted data to a string. These are versions of _snprintf, _snprintf_l, _snwprintf, _snwprintf_l with security enhancements as described in Security Enhancements in the CRT.

```
int _snprintf_s(
   char *buffer,
   size_t sizeOfBuffer,
   size_t count,
   const char *format [,
      argument] ...
);
int _snprintf_s_l(
   char *buffer,
   size_t sizeOfBuffer,
   size_t count,
   const char *format,
   locale_t locale [,
      argument] ...
);
int _snwprintf_s(
   wchar_t *buffer,
   size_t sizeOfBuffer,
   size_t count,
   const wchar_t *format [,
      argument] ...
);
int _snwprintf_s_l(
   wchar_t *buffer,
   size_t sizeOfBuffer,
   size_t count,
   const wchar_t *format,
   locale_t locale [,
      argument] ...
);
template <size_t size>
int _snprintf_s(
   char (&buffer)[size],
   size_t count,
   const char *format [,
      argument] ...
); // C++ only
template <size_t size>
int _snwprintf_s(
   wchar_t (&buffer)[size],
   size_t count,
   const wchar_t *format [,
      argument] ...
); // C++ only
```

Parameters

buffer

Storage location for the output.

sizeOfBuffer

The size of the storage location for output. Size in **bytes** for **_snprintf_s** or size in **words** for **_snwprintf_s**.

Count

Maximum number of characters to store, or _TRUNCATE.

format

Format-control string.

argument

Optional arguments.

locale

The locale to use.

Return Value

_snprintf_s returns the number of characters stored in *buffer*, not counting the terminating null character. **_snwprintf_s** returns the number of wide characters stored in *buffer*, not counting the terminating null wide character.

If the storage required to store the data and a terminating null exceeds *sizeOfBuffer*, the invalid parameter handler is invoked, as described in Parameter Validation. If execution continues after the invalid parameter handler, these functions set *buffer* to an empty string, set **errno** to **ERANGE**, and return -1.

If *buffer* or *format* is a **NULL** pointer, or if *count* is less than or equal to zero, the invalid parameter handler is invoked. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1.

For information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_snprintf_s** function formats and stores *count* or fewer characters in *buffer* and appends a terminating null. Each argument (if any) is converted and output according to the corresponding format specification in *format*. The formatting is consistent with the **printf** family of functions; see Format Specification Fields: printf and wprintf Functions. If copying occurs between strings that overlap, the behavior is undefined.

If *count* is _TRUNCATE, then **_snprintf_s** writes as much of the string as will fit in *buffer* while leaving room for a terminating null. If the entire string (with terminating null) fits in *buffer*, then **_snprintf_s** returns the number of characters written (not including the terminating null); otherwise, **_snprintf_s** returns -1 to indicate that truncation occurred.

Security Note
Ensure that *format* is not a user-defined string.

_snwprintf_s is a wide-character version of **_snprintf_s**; the pointer arguments to **_snwprintf_s** are wide-character strings. Detection of encoding errors in **_snwprintf_s** might differ from that in **_snprintf_s**. **_snwprintf_s**, like **swprintf_s**, writes output to a string rather than to a destination of type **FILE**.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

Tchar.h routine	**_UNICODE and _MBCS not defined**	**_MBCS defined**	**_UNICODE defined**
_sntprintf_s	**_snprintf_s**	**_snprintf_s**	**_snwprintf_s**
_sntprintf_s_l	**_snprintf_s_l**	**_snprintf_s_l**	**_snwprintf_s_l**

Requirements

Routine	**Required header**	**Compatibility**
_snprintf_s, **_snprintf_s_l**	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_snwprintf_s, **_snwprintf_s_l**	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_snprintf_s.cpp
// compile with: /MTd

// These #defines enable secure template overloads
// (see last part of Examples() below)
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES_COUNT 1

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <crtdbg.h>  // For _CrtSetReportMode
#include <errno.h>

// This example uses a 10-byte destination buffer.

int snprintf_s_tester( const char * fmt, int x, int count )
{
   char dest[10];

   printf( "\n" );

   if ( count == _TRUNCATE )
      printf( "%d-byte buffer; truncation semantics\n",
               _countof(dest) );
   else
      printf( "count = %d; %d-byte buffer\n",
               count, _countof(dest) );

   int ret = _snprintf_s( dest, _countof(dest), count, fmt, x );

   printf( "    new contents of dest: '%s'\n", dest );

   return ret;
}


void Examples()
{
   // formatted output string is 9 characters long: "<<<123>>>"
   snprintf_s_tester( "<<<%d>>>", 121, 8 );
   snprintf_s_tester( "<<<%d>>>", 121, 9 );
   snprintf_s_tester( "<<<%d>>>", 121, 10 );

   printf( "\nDestination buffer too small:\n" );

   snprintf_s_tester( "<<<%d>>>", 1221, 10 );

   printf( "\nTruncation examples:\n" );

   int ret = snprintf_s_tester( "<<<%d>>>", 1221, _TRUNCATE );
   printf( "    truncation %s occur\n", ret == -1 ? "did"
                                                   : "did not" );

   ret = snprintf_s_tester( "<<<%d>>>", 121, _TRUNCATE );
   printf( "    truncation %s occur\n", ret == -1 ? "did"
                                                   : "did not" );
   printf( "\nSecure template overload example:\n" );

   char dest[10];
   _snprintf( dest, 10, "<<<%d>>>", 12321 );
   // With secure template overloads enabled (see #defines
   // at top of file), the preceding line is replaced by
   //    _snprintf_s( dest, _countof(dest), 10, "<<<%d>>>", 12345 );
   // Instead of causing a buffer overrun, _snprintf_s invokes
```

```
      // the invalid parameter handler.
      // If secure template overloads were disabled, _snprintf would
      // write 10 characters and overrun the dest buffer.
      printf( "    new contents of dest: '%s'\n", dest );
   }

   void myInvalidParameterHandler(
      const wchar_t* expression,
      const wchar_t* function,
      const wchar_t* file,
      unsigned int line,
      uintptr_t pReserved)
   {
      wprintf(L"Invalid parameter handler invoked: %s\n", expression);
   }

   int main( void )
   {
      _invalid_parameter_handler oldHandler, newHandler;

      newHandler = myInvalidParameterHandler;
      oldHandler = _set_invalid_parameter_handler(newHandler);
      // Disable the message box for assertions.
      _CrtSetReportMode(_CRT_ASSERT, 0);

      Examples();
   }
```

Output

```
count = 8; 10-byte buffer
    new contents of dest: '<<<121>>'

count = 9; 10-byte buffer
    new contents of dest: '<<<121>>>'

count = 10; 10-byte buffer
    new contents of dest: '<<<121>>>'

Destination buffer too small:

count = 10; 10-byte buffer
Invalid parameter handler invoked: ("Buffer too small", 0)
    new contents of dest: ''

Truncation examples:

10-byte buffer; truncation semantics
    new contents of dest: '<<<1221>>'
    truncation did occur

10-byte buffer; truncation semantics
    new contents of dest: '<<<121>>>'
    truncation did not occur

Secure template overload example:
Invalid parameter handler invoked: ("Buffer too small", 0)
    new contents of dest: ''
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O

sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l

fprintf, _fprintf_l, fwprintf, _fwprintf_l

printf, _printf_l, wprintf, _wprintf_l

scanf, _scanf_l, wscanf, _wscanf_l

sscanf, _sscanf_l, swscanf, _swscanf_l
vprintf Functions

_snscanf, _snscanf_l, _snwscanf, _snwscanf_l

Reads formatted data of a specified length from a string. These functions are deprecated because more secure versions are available; see _snscanf_s, _snscanf_s_l, _snwscanf_s, _snwscanf_s_l.

```
int __cdecl _snscanf(
   const char * input,
   size_t length,
   const char * format,
      ...
);
int __cdecl _snscanf_l(
   const char * input,
   size_t length,
   const char * format,
   locale_t locale,
      ...
);
int __cdecl _snwscanf(
   const wchar_t * input,
   size_t length,
   const wchar_t * format,
      ...
);
int __cdecl _snwscanf_l(
   const wchar_t * input,
   size_t length,
   const wchar_t * format,
   locale_t locale,
      ...
);
```

Parameters

input

Input string to examine.

length

Number of characters to examine in *input*.

format

One or more format specifiers.

... (optional)

Variables that will be used to store the values extracted from the input string by the format specifiers in *format*.

locale

The locale to use.

Return Value

Both of these functions returns the number of fields successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned. The return value is **EOF** for an error or if the end of the string is reached before the first conversion. For more information, see sscanf.

If *input* or *format* is a **NULL** pointer, or if *length* is less than or equal to zero, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **EOF** and set **errno** to **EINVAL**.

For information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

This function is like **sscanf** except that it provides the ability to specify a fixed number of characters to examine from the input

string. For more information, see sscanf.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_sntscanf	**_snscanf**	**_snscanf**	**_snwscanf**
_sntscanf_l	**_snscanf_l**	**_snscanf_l**	**_snwscanf_l**

Requirements

Routine	Required header	Compatibility
_snscanf, **_snscanf_l**	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_snwscanf, **_snwscanf_l**	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_snscanf.c
// compile with: /W3

#include <stdio.h>
int main( )
{
   char  str1[] = "15 12 14...";
   wchar_t  str2[] = L"15 12 14...";
   char  s1[3];
   wchar_t  s2[3];
   int   i;
   float fp;

   i = _snscanf( str1, 6,  "%s %f", s1, &fp); // C4996
   // Note: _snscanf is deprecated; consider using _snscanf_s instead
   printf("_snscanf converted %d fields: ", i);
   printf("%s and %f\n", s1, fp);

   i = _snwscanf( str2, 6,  L"%s %f", s2, &fp); // C4996
   // Note: _snwscanf is deprecated; consider using _snwscanf_s instead
   wprintf(L"_snwscanf converted %d fields: ", i);
   wprintf(L"%s and %f\n", s2, fp);
}
```

Output

```
_snscanf converted 2 fields: 15 and 12.000000
_snwscanf converted 2 fields: 15 and 12.000000
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

scanf Width Specification

_snscanf_s, _snscanf_s_l, _snwscanf_s, _snwscanf_s_l

Reads formatted data of a specified length from a string. These are versions of _snscanf, _snscanf_l, _snwscanf, _snwscanf_l with security enhancements as described in Security Enhancements in the CRT.

```
int __cdecl _snscanf_s(
   const char * input,
   size_t length,
   const char * format,
   ...
);
int __cdecl _snscanf_s_l(
   const char * input,
   size_t length,
   const char * format,
   locale_t locale,
   ...
);
int __cdecl _snwscanf_s(
   const wchar_t * input,
   size_t length,
   const wchar_t * format,
   ...
);
int __cdecl _snwscanf_s_l(
   const wchar_t * input,
   size_t length,
   const wchar_t * format,
   locale_t locale,
   …
);
```

Parameters

input

Input string to examine.

length

Number of characters to examine in *input*.

format

One or more format specifiers.

... (optional)

Variables that will be used to store the values extracted from the input string by the format specifiers in *format*.

locale

The locale to use.

Return Value

Both of these functions returns the number of fields successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned. The return value is **EOF** for an error or if the end of the string is reached before the first conversion. For more information, see sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l.

If *input* or *format* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **EOF** and set **errno** to **EINVAL**.

For information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

This function is like **sscanf_s** except that it provides the ability to specify a fixed number of characters to examine from the input string. For more information, see sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l.

The buffer size parameter is required with the type field characters **c**, **C**, **s**, **S**, and **[**. For more information, see scanf Type Field Characters.

Note
The size parameter is of type **unsigned**, not **size_t**.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Generic-Text Routine Mappings

Tchar.h routine	**_UNICODE and _MBCS not defined**	**_MBCS defined**	**_UNICODE defined**
_sntscanf_s	**_snscanf_s**	**_snscanf_s**	**_snwscanf_s**
_sntscanf_s_l	**_snscanf_s_l**	**_snscanf_s_l**	**_snwscanf_s_l**

Requirements

Routine	**Required header**	**Compatibility**
_snscanf_s, **_snscanf_s_l**	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_snwscanf_s, **_snwscanf_s_l**	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_snscanf_s.c
// This example scans a string of
// numbers, using both the character
// and wide character secure versions
// of the snscanf function.

#include <stdio.h>

int main( )
{
    char        str1[] = "15 12 14...";
    wchar_t     str2[] = L"15 12 14...";
    char        s1[3];
    wchar_t     s2[3];
    int         i;
    float       fp;

    i = _snscanf_s( str1, 6,  "%s %f", s1, 3, &fp);
    printf_s("_snscanf_s converted %d fields: ", i);
    printf_s("%s and %f\n", s1, fp);

    i = _snwscanf_s( str2, 6,  L"%s %f", s2, 3, &fp);
    wprintf_s(L"_snwscanf_s converted %d fields: ", i);
    wprintf_s(L"%s and %f\n", s2, fp);
}
```

Output

```
_snscanf_s converted 2 fields: 15 and 12.000000
_snwscanf_s converted 2 fields: 15 and 12.000000
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

scanf Width Specification

sopen

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _sopen or security-enhanced _sopen_s instead.

_sopen, _wsopen

Opens a file for sharing. These functions are deprecated because more secure versions are available; see _sopen_s, _wsopen_s.

```
int _sopen(
   const char *filename,
      int oflag,
   int shflag [,
   int pmode ]
);
int _wsopen(
   const wchar_t *filename,
      int oflag,
   int shflag [,
   int pmode ]
);
```

Parameters

filename

File name.

oflag

Type of operations allowed.

shflag

Type of sharing allowed.

pmode

Permission setting.

Return Value

Each of these functions returns a file descriptor for the opened file.

If *filename* or *oflag* is a **NULL** pointer, or if *oflag* or *shflag* is not within a valid range of values, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to one of the following values.

EACCES

Given path is a directory, or file is read-only, but an open-for-writing operation was attempted.

EEXIST

_O_CREAT and **_O_EXCL** flags were specified, but *filename* already exists.

EINVAL

Invalid *oflag* or *shflag* argument.

EMFILE

No more file descriptors available.

ENOENT

File or path not found.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_sopen** function opens the file specified by *filename* and prepares the file for shared reading or writing, as defined by *oflag* and *shflag*. **_wsopen** is a wide-character version of **_sopen**; the *filename* argument to **_wsopen** is a wide-character string. **_wsopen** and **_sopen** behave identically otherwise.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tsopen	**_sopen**	**_sopen**	**_wsopen**

The integer expression *oflag* is formed by combining one or more of the following manifest constants, defined in the file Fcntl.h. When two or more constants form the argument *oflag*, they are combined with the bitwise-OR operator (**|**).

_O_APPEND

Repositions a file pointer to the end of the file before every write operation.

_O_BINARY

Opens a file in binary (untranslated) mode. (See fopen for a description of binary mode.)

_O_CREAT

Creates and opens new file for writing. Has no effect if the file specified by *filename* exists. The *pmode* argument is required when **_O_CREAT** is specified.

_O_CREAT| _O_SHORT_LIVED

Create a file as temporary and if possible do not flush to disk. The *pmode* argument is required when **_O_CREAT** is specified.

_O_CREAT| _O_TEMPORARY

Create a file as temporary; the file is deleted when the last file descriptor is closed. The *pmode* argument is required when **_O_CREAT** is specified.

_O_CREAT| _O_EXCL

Returns an error value if a file specified by *filename* exists. Applies only when used with **_O_CREAT**.

_O_NOINHERIT

Prevents creation of a shared file descriptor.

_O_RANDOM

Specifies primarily random access from disk.

_O_RDONLY

Opens a file for reading only; cannot be specified with **_O_RDWR** or **_O_WRONLY**.

_O_RDWR

Opens file for both reading and writing; cannot be specified with **_O_RDONLY** or **_O_WRONLY**.

_O_SEQUENTIAL

Specifies primarily sequential access from disk.

_O_TEXT

Opens a file in text (translated) mode. (For more information, see Text and Binary Mode File I/O and fopen.)

_O_TRUNC

Opens a file and truncates it to zero length; the file must have write permission. You cannot specify this flag with **_O_RDONLY**. **_O_TRUNC** used with **_O_CREAT** opens an existing file or creates a new file.

Note
The **_O_TRUNC** flag destroys the contents of the specified file.

_O_WRONLY

Opens a file for writing only; cannot be specified with **_O_RDONLY** or **_O_RDWR**.

_O_U16TEXT

Open the file in Unicode UTF-16 mode. This option is available in Visual C++ 2005.

_O_U8TEXT

Open the file in Unicode UTF-8 mode. This option is available in Visual C++ 2005.

_O_WTEXT

Open the file in Unicode mode. This option is available in Visual C++ 2005.

To specify the file access mode, you must specify either **_O_RDONLY**, **_O_RDWR**, or **_O_WRONLY**. There is no default value for the access mode.

If **_sopen** is called with **_O_WRONLY|_O_APPEND** (append mode) and **_O_WTEXT**, **_O_U16TEXT**, or **_O_U8TEXT**, it will first try to open the file for reading and writing, read the BOM, then reopen it for writing only. If opening the file for reading and writing fails, it will open the file for writing only and use the default value for the Unicode mode setting.

The argument *shflag* is a constant expression consisting of one of the following manifest constants, defined in Share.h.

_SH_DENYRW

Denies read and write access to a file.

_SH_DENYWR

Denies write access to a file.

_SH_DENYRD

Denies read access to a file.

_SH_DENYNO

Permits read and write access.

The *pmode* argument is required only when one specifies **_O_CREAT**. If the file does not exist, *pmode* specifies the file's permission settings, which are set when the new file is closed the first time. Otherwise *pmode* is ignored. *pmode* is an integer expression that contains one or both of the manifest constants **_S_IWRITE** and **_S_IREAD**, defined in SYS\Stat.h. When both constants are given, they are combined with the bitwise-OR operator. The meaning of *pmode* is as follows.

_S_IWRITE

Writing permitted.

_S_IREAD

Reading permitted.

_S_IREAD | _S_IWRITE

Reading and writing permitted.

If write permission is not given, the file is read-only. Under the Windows operating system, all files are readable; it is not possible to give write-only permission. Thus, the modes **_S_IWRITE** and **_S_IREAD | _S_IWRITE** are equivalent.

_sopen applies the current file-permission mask to *pmode* before setting the permissions (see _umask).

Requirements

Routine	Required header	Optional header	Compatibility
_sopen	<io.h>	<fcntl.h>, <sys/types.h>, <sys/stat.h>, <share.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wsopen	<io.h> or <wchar.h>	<fcntl.h>, <sys/types.h>, <sys/stat.h>, <share.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example for _locking.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Low-Level I/O
_close
_creat, _wcreat
fopen, _wfopen
_fsopen, _wfsopen
_open, _wopen

_sopen_s, _wsopen_s

Opens a file for sharing. These are versions of _sopen and _wsopen with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _sopen_s(
   int* pfh,
   const char *filename,
      int oflag,
   int shflag,
   int pmode
);
errno_t _wsopen_s(
   int* pfh,
   const wchar_t *filename,
   int oflag,
   int shflag,
   int pmode,
);
```

Parameters

[out] *pfh*

The file handle, or -1 in the case of an error.

[in] *filename*

File name.

[in] *oflag*

Type of operations allowed.

[in] *shflag*

Type of sharing allowed.

[in] *pmode*

Permission setting.

Return Value

A nonzero return value indicates an error, in which case **errno** is set to one of the following values.

EACCES

Given path is a directory, or file is read-only, but an open-for-writing operation was attempted.

EEXIST

_O_CREAT and **_O_EXCL** flags were specified, but *filename* already exists.

EINVAL

Invalid *oflag*, *shflag*, or *pmode* argument, or *pfh* or *filename* was a null pointer.

EMFILE

No more file descriptors available.

ENOENT

File or path not found.

If an invalid argument is passed to the function, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and **EINVAL** is returned.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

In the case of an error, -1 will be returned through *pfh* (unless *pfh* is a null pointer).

Remarks

The **_sopen_s** function opens the file specified by *filename* and prepares the file for shared reading or writing, as defined by *oflag* and *shflag*. **_wsopen_s** is a wide-character version of **_sopen_s**; the *filename* argument to **_wsopen_s** is a wide-character string. **_wsopen_s** and **_sopen_s** behave identically otherwise.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tsopen_s	_sopen_s	_sopen_s	_wsopen_s

The integer expression *oflag* is formed by combining one or more manifest constants, defined in the file Fcntl.h. When two or more constants form the argument *oflag*, they are combined with the bitwise-OR operator (**|**).

_O_APPEND

Repositions a file pointer to the end of the file before every write operation.

_O_BINARY

Opens a file in binary (untranslated) mode. (See fopen for a description of binary mode.)

_O_CREAT

Creates and opens new file for writing. Has no effect if file specified by *filename* exists.

_O_CREAT| _O_SHORT_LIVED

Create a file as temporary and if possible do not flush to disk.

_O_CREAT| _O_TEMPORARY

Create a file as temporary; the file is deleted when the last file descriptor is closed.

_O_CREAT| _O_EXCL

Returns an error value if the file specified by *filename* exists. Applies only when used with **_O_CREAT**.

_O_NOINHERIT

Prevents creation of a shared file descriptor.

_O_RANDOM

Specifies primarily random access from disk.

_O_RDONLY

Opens a file for reading only; cannot be specified with **_O_RDWR** or **_O_WRONLY**.

_O_RDWR

Opens a file for both reading and writing; cannot be specified with **_O_RDONLY** or **_O_WRONLY**.

_O_SEQUENTIAL

Specifies primarily sequential access from disk.

_O_TEXT

Opens a file in text (translated) mode. (For more information, see Text and Binary Mode File I/O and fopen.)

_O_TRUNC

Opens a file and truncates it to zero length; the file must have write permission. You cannot specify this flag with **_O_RDONLY**. **_O_TRUNC** used with **_O_CREAT** opens an existing file or creates a new file.

Note
The **_O_TRUNC** flag destroys the contents of the specified file.

_O_WRONLY

Opens a file for writing only; cannot be specified with **_O_RDONLY** or **_O_RDWR**.

_O_U16TEXT

Open the file in Unicode UTF-16 mode. This option is available in Visual C++ 2005.

_O_U8TEXT

Open the file in Unicode UTF-8 mode. This option is available in Visual C++ 2005.

_O_WTEXT

Open the file in Unicode mode. This option is available in Visual C++ 2005.

To specify the file access mode, you must specify either **_O_RDONLY**, **_O_RDWR**, or **_O_WRONLY**. There is no default value for the access mode.

If **_sopen_s** is called with **_O_WRONLY|_O_APPEND** (append mode) and **_O_WTEXT**, **_O_U16TEXT**, or **_O_U8TEXT**, it will first try to open the file for reading and writing, read the BOM, then reopen it for writing only. If opening the file for reading and writing fails, it will open the file for writing only and use the default value for the Unicode mode setting.

The argument *shflag* is a constant expression consisting of one of the following manifest constants, defined in Share.h.

_SH_DENYRW

Denies read and write access to a file.

_SH_DENYWR

Denies write access to a file.

_SH_DENYRD

Denies read access to a file.

_SH_DENYNO

Permits read and write access.

The *pmode* argument is always required, unlike in **_sopen**. When you specify **_O_CREAT**, if the file does not exist, *pmode* specifies the file's permission settings, which are set when the new file is closed the first time. Otherwise *pmode* is ignored. *pmode* is an integer expression that contains one or both of the manifest constants **_S_IWRITE** and **_S_IREAD**, defined in SYS\Stat.h. When both constants are given, they are combined with the bitwise-OR operator. The meaning of *pmode* is as follows.

_S_IWRITE

Writing permitted.

_S_IREAD

Reading permitted.

_S_IREAD | _S_IWRITE

Reading and writing permitted.

If write permission is not given, the file is read-only. Under the Windows operating system, all files are readable; it is not possible to give write-only permission. Thus, the modes **_S_IWRITE** and **_S_IREAD | _S_IWRITE** are equivalent.

_sopen_s applies the current file-permission mask to *pmode* before setting the permissions (see _umask).

Requirements

Routine	Required header	Optional header	Compatibility
_sopen_s	<io.h>	<fcntl.h>, <sys/types.h>, <sys/stat.h>, <share.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_wsopen_s	<io.h> or <wchar.h>	<fcntl.h>, <sys/types.h>, <sys/stat.h>, <share.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example for _locking.

See Also

Reference

Low-Level I/O
_close
_creat, _wcreat
fopen, _wfopen
_fsopen, _wfsopen
_open, _wopen

_spawn, _wspawn Functions

Each of the **_spawn** functions creates and executes a new process:

_spawnl, _wspawnl	_spawnv, _wspawnv
_spawnle, _wspawnle	_spawnve, _wspawnve
_spawnlp, _wspawnlp	_spawnvp, _wspawnvp
_spawnlpe, _wspawnlpe	_spawnvpe, _wspawnvpe

The letters at the end of the function name determine the variation.

e

envp, array of pointers to environment settings, is passed to new process.

l

Command-line arguments are passed individually to **_spawn** function. This suffix is typically used when a number of parameters to a new process is known in advance.

p

PATH environment variable is used to find the file to execute.

v

argv, array of pointers to command-line arguments, is passed to **_spawn** function. This suffix is typically used when a number of parameters to a new process is variable.

Remarks

The **_spawn** functions each create and execute a new process. They automatically handle multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the multibyte code page currently in use. The **_wspawn** functions are wide-character versions of the **_spawn** functions; they do not handle multibyte-character strings. Otherwise, the **_wspawn** functions behave identically to their **_spawn** counterparts.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_tspawnl	**_spawnl**	**_spawnl**	**_wspawnl**
_tspawnle	**_spawnle**	**_spawnle**	**_wspawnle**
_tspawnlp	**_spawnlp**	**_spawnlp**	**_wspawnlp**
_tspawnlpe	**_spawnlpe**	**_spawnlpe**	**_wspawnlpe**
_tspawnv	**_spawnv**	**_spawnv**	**_wspawnv**
_tspawnve	**_spawnve**	**_spawnve**	**_wspawnve**
_tspawnvp	**_spawnvp**	**_spawnvp**	**_wspawnvp**
_tspawnvpe	**_spawnvpe**	**_spawnvpe**	**_wspawnvpe**

Enough memory must be available for loading and executing the new process. The *mode* argument determines the action taken by the calling process before and during **_spawn**. The following values for *mode* are defined in Process.h:

_P_OVERLAY

Overlays a calling process with a new process, destroying the calling process (same effect as **_exec** calls).

_P_WAIT

Suspends a calling thread until execution of the new process is complete (synchronous **_spawn**).

_P_NOWAIT or **_P_NOWAITO**

Continues to execute a calling process concurrently with the new process (asynchronous **_spawn**).

_P_DETACH

Continues to execute the calling process; the new process is run in the background with no access to the console or keyboard. Calls to **_cwait** against the new process fail (asynchronous **_spawn**).

The *cmdname* argument specifies the file that is executed as the new process and can specify a full path (from the root), a partial path (from the current working directory), or just a file name. If *cmdname* does not have a file name extension or does not end with a period (.), the **_spawn** function first tries the .com file name extension and then the .exe file name extension, the .bat file name extension, and finally the .cmd file name extension.

If *cmdname* has a file name extension, only that extension is used. If *cmdname* ends with a period, the **_spawn** call searches for *cmdname* with no file name extension. The **_spawnlp**, **_spawnlpe**, **_spawnvp**, and **_spawnvpe** functions search for *cmdname* (using the same procedures) in the directories specified by the **PATH** environment variable.

If *cmdname* contains a drive specifier or any slashes (that is, if it is a relative path), the **_spawn** call searches only for the specified file; no path searching is done.

Visual C++ 2005 introduces a breaking change in the **_spawn** functions. In the past, some of these functions set **errno** to zero on success; the new behavior is to leave **errno** untouched on success, as specified by the C standard. If you need to emulate the old behavior, set **errno** to zero just before calling these functions.

Note
To ensure proper overlay initialization and termination, do not use the **setjmp** or **longjmp** function to enter or leave an over lay routine.

Arguments for the Spawned Process

To pass arguments to the new process, give one or more pointers to character strings as arguments in the **_spawn** call. These character strings form the argument list for the spawned process. The combined length of the strings forming the argument list for the new process must not exceed 1024 bytes. The terminating null character ('\0') for each string is not included in the count, but space characters (automatically inserted to separate arguments) are included.

Note
Spaces embedded in strings may cause unexpected behavior; for example, passing **_spawn** the string `"hi there"` will result i n the new process getting two arguments, `"hi"` and `"there"`. If the intent was to have the new process open a file named "hi there", the process would fail. You can avoid this by quoting the string: `"\"hi there\""`.

Security Note
Do not pass user input to **_spawn** without explicitly checking its content. **_spawn** will result in a call to CreateProcess so kee p in mind that unqualified path names could lead to potential security vulnerabilities.

You can pass argument pointers as separate arguments (in **_spawnl**, **_spawnle**, **_spawnlp**, and **_spawnlpe**) or as an array of pointers (in **_spawnv**, **_spawnve**, **_spawnvp**, and **_spawnvpe**). You must pass at least one argument, *arg0* or *argv*[0], to the spawned process. By convention, this argument is the name of the program as you would type it on the command line. A different value does not produce an error.

The **_spawnl**, **_spawnle**, **_spawnlp**, and **_spawnlpe** calls are typically used in cases where the number of arguments is known in advance. The *arg0* argument is usually a pointer to *cmdname*. The arguments *arg1* through *argn* are pointers to the character strings forming the new argument list. Following *argn*, there must be a **NULL** pointer to mark the end of the argument list.

The **_spawnv**, **_spawnve**, **_spawnvp**, and **_spawnvpe** calls are useful when there is a variable number of arguments to the new process. Pointers to the arguments are passed as an array, *argv*. The argument *argv*[0] is usually a pointer to a path in real

mode or to the program name in protected mode, and *argv*[1] through *argv*[*n*] are pointers to the character strings forming the new argument list. The argument *argv*[*n* +1] must be a **NULL** pointer to mark the end of the argument list.

Environment of the Spawned Process

Files that are open when a **_spawn** call is made remain open in the new process. In the **_spawnl**, **_spawnlp**, **_spawnv**, and **_spawnvp** calls, the new process inherits the environment of the calling process. You can use the **_spawnle**, **_spawnlpe**, **_spawnve**, and **_spawnvpe** calls to alter the environment for the new process by passing a list of environment settings through the *envp* argument. The argument *envp* is an array of character pointers, each element (except the final element) of which points to a null-terminated string defining an environment variable. Such a string usually has the form *NAME=value* where *NAME* is the name of an environment variable and *value* is the string value to which that variable is set. (Note that *value* is not enclosed in double quotation marks.) The final element of the *envp* array should be **NULL**. When *envp* itself is **NULL**, the spawned process inherits the environment settings of the parent process.

The **_spawn** functions can pass all information about open files, including the translation mode, to the new process. This information is passed in real mode through the **C_FILE_INFO** entry in the environment. The startup code normally processes this entry and then deletes it from the environment. However, if a **_spawn** function spawns a non-C process, this entry remains in the environment. Printing the environment shows graphics characters in the definition string for this entry because the environment information is passed in binary form in real mode. It should not have any other effect on normal operations. In protected mode, the environment information is passed in text form and therefore contains no graphics characters.

You must explicitly flush (using **fflush** or **_flushall**) or close any stream before calling a **_spawn** function.

New processes created by calls to **_spawn** routines do not preserve signal settings. Instead, the spawned process resets signal settings to the default.

Redirecting Output

If you are calling **_spawn** from a DLL or a GUI application and want to redirect the output to a pipe, you have two options:

- Use the Win32 API to create a pipe, then call AllocConsole, set the handle values in the startup structure, and call CreateProcess.
- Call _popen, _wpopen which will create a pipe and invoke the app using **cmd.exe /c** (or **command.exe /c**).

Example

```
// crt_spawn.c
// This program accepts a number in the range
// 1-8 from the command line. Based on the number it receives,
// it executes one of the eight different procedures that
// spawn the process named child. For some of these procedures,
// the CHILD.EXE file must be in the same directory; for
// others, it only has to be in the same path.
//

#include <stdio.h>
#include <process.h>

char *my_env[] =
{
   "THIS=environment will be",
   "PASSED=to child.exe by the",
   "_SPAWNLE=and",
   "_SPAWNLPE=and",
   "_SPAWNVE=and",
   "_SPAWNVPE=functions",
   NULL
};

int main( int argc, char *argv[] )
{
   char *args[4];

   // Set up parameters to be sent:
   args[0] = "child";
   args[1] = "spawn??";
   args[2] = "two";
   args[3] = NULL;
```

```
    if (argc <= 2)
    {
       printf( "SYNTAX: SPAWN <1-8> <childprogram>\n" );
       exit( 1 );
    }

    switch (argv[1][0])   // Based on first letter of argument
    {
    case '1':
       _spawnl( _P_WAIT, argv[2], argv[2], "_spawnl", "two", NULL );
       break;
    case '2':
       _spawnle( _P_WAIT, argv[2], argv[2], "_spawnle", "two",
                 NULL, my_env );
       break;
    case '3':
       _spawnlp( _P_WAIT, argv[2], argv[2], "_spawnlp", "two", NULL );
       break;
    case '4':
       _spawnlpe( _P_WAIT, argv[2], argv[2], "_spawnlpe", "two",
                  NULL, my_env );
       break;
    case '5':
       _spawnv( _P_OVERLAY, argv[2], args );
       break;
    case '6':
       _spawnve( _P_OVERLAY, argv[2], args, my_env );
       break;
    case '7':
       _spawnvp( _P_OVERLAY, argv[2], args );
       break;
    case '8':
       _spawnvpe( _P_OVERLAY, argv[2], args, my_env );
       break;
    default:
       printf( "SYNTAX: SPAWN <1-8> <childprogram>\n" );
       exit( 1 );
    }
    printf( "from SPAWN!\n" );
  }
```

Sample Output

```
child process output
from SPAWN!
```

See Also

Reference

Process and Environment Control
abort
atexit
_exec, _wexec Functions
exit, _exit
_flushall
_getmbcp
_onexit, _onexit_m
_setmbcp
system, _wsystem

spawnl

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _spawnl instead.

_spawnl, _wspawnl

Creates and executes a new process.

```
intptr_t _spawnl(
   int mode,
      const char *cmdname,
   const char *arg0,
   const char *arg1,
   ... const char *argn,
   NULL
);
intptr_t _wspawnl(
   int mode,
   const wchar_t *cmdname,
   const wchar_t *arg0,
   const wchar_t *arg1,
   ... const wchar_t *argn,
   NULL
);
```

Parameters

mode

Execution mode for the calling process.

cmdname

Path of the file to be executed.

arg0, arg1, ... argn

List of pointers to arguments.

Return Value

The return value from a synchronous **_spawnl** or **_wspawnl** (**_P_WAIT** specified for *mode*) is the exit status of the new process. The return value from an asynchronous **_spawnl** or **_wspawnl** (**_P_NOWAIT** or **_P_NOWAITO** specified for *mode*) is the process handle. The exit status is 0 if the process terminated normally. You can set the exit status to a nonzero value if the spawned process specifically calls the **exit** routine with a nonzero argument. If the new process did not explicitly set a positive exit status, a positive exit status indicates an abnormal exit with an abort or an interrupt. A return value of –1 indicates an error (the new process is not started). In this case, **errno** is set to one of the following values.

E2BIG

Argument list exceeds 1024 bytes.

EINVAL

mode argument is invalid.

ENOENT

File or path is not found.

ENOEXEC

Specified file is not executable or has invalid executable-file format.

ENOMEM

Not enough memory is available to execute the new process.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

These functions validate their parameters. If either *cmdname* or *arg0* is an empty string or a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL**, and return -1. No new process is spawned.

Remarks

Each of these functions creates and executes a new process, passing each command-line argument as a separate parameter.

Requirements

Routine	Required header	Compatibility
_spawnl	<process.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wspawnl	<stdio.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example in _spawn, _wspawn Functions.

.NET Framework Equivalent

- System::Diagnostics::Process Class
- System::Diagnostics::ProcessStartInfo Class

See Also

Reference

Process and Environment Control
_spawn, _wspawn Functions
abort
atexit
_exec, _wexec Functions
exit, _exit
_flushall
_getmbcp
_onexit, _onexit_m
_setmbcp
system, _wsystem

spawnle

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _spawnle instead.

_spawnle, _wspawnle

Creates and executes a new process.

```
intptr_t _spawnle(
   int mode,
      const char *cmdname,
   const char *arg0,
   const char *arg1,
   ... const char *argn,
   NULL,
   const char *const *envp
);
intptr_t _wspawnle(
   int mode,
   const wchar_t *cmdname,
   const wchar_t *arg0,
   const wchar_t *arg1,
   ... const wchar_t *argn,
   NULL,
   const wchar_t *const *envp
);
```

Parameters

mode

Execution mode for the calling process.

cmdname

Path of the file to be executed.

arg0, arg1, ... argn

List of pointers to arguments.

envp

Array of pointers to environment settings.

Return Value

The return value from a synchronous **_spawnle** or **_wspawnle** (**_P_WAIT** specified for *mode*) is the exit status of the new process. The return value from an asynchronous **_spawnle** or **_wspawnle** (**_P_NOWAIT** or **_P_NOWAITO** specified for *mode*) is the process handle. The exit status is 0 if the process terminated normally. You can set the exit status to a nonzero value if the spawned process specifically calls the **exit** routine with a nonzero argument. If the new process did not explicitly set a positive exit status, a positive exit status indicates an abnormal exit with an abort or an interrupt. A return value of –1 indicates an error (the new process is not started). In this case, **errno** is set to one of the following values.

E2BIG

Argument list exceeds 1024 bytes.

EINVAL

mode argument is invalid.

ENOENT

File or path is not found.

ENOEXEC

Specified file is not executable or has invalid executable-file format.

ENOMEM

Not enough memory is available to execute the new process.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions creates and executes a new process, passing each command-line argument as a separate parameter and also passing an array of pointers to environment settings.

These functions validate their parameters. If either *cmdname* or *arg0* is an empty string or a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL**, and return -1. No new process is spawned.

Requirements

Routine	Required header	Compatibility
_spawnle	<process.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wspawnle	<stdio.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example in _spawn, _wspawn Functions.

.NET Framework Equivalent

- System::Diagnostics::Process Class
- System::Diagnostics::ProcessStartInfo Class

See Also

Reference

Process and Environment Control
_spawn, _wspawn Functions
abort
atexit
_exec, _wexec Functions
exit, _exit
_flushall
_getmbcp
_onexit, _onexit_m
_setmbcp
system, _wsystem

spawnlp

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _spawnlp instead.

_spawnlp, _wspawnlp

Creates and executes a new process.

```
intptr_t _spawnlp(
   int mode,
   const char *cmdname,
   const char *arg0,
   const char *arg1,
   ... const char *argn,
   NULL
);
intptr_t _wspawnlp(
   int mode,
   const wchar_t *cmdname,
   const wchar_t *arg0,
   const wchar_t *arg1,
   ... const wchar_t *argn,
   NULL
);
```

Parameters

mode

Execution mode for the calling process.

cmdname

Path of the file to be executed.

arg0, arg1, ... argn

List of pointers to arguments.

Return Value

The return value from a synchronous **_spawnlp** or **_wspawnlp** (**_P_WAIT** specified for *mode*) is the exit status of the new process. The return value from an asynchronous **_spawnlp** or **_wspawnlp** (**_P_NOWAIT** or **_P_NOWAITO** specified for *mode*) is the process handle. The exit status is 0 if the process terminated normally. You can set the exit status to a nonzero value if the spawned process specifically calls the **exit** routine with a nonzero argument. If the new process did not explicitly set a positive exit status, a positive exit status indicates an abnormal exit with an abort or an interrupt. A return value of –1 indicates an error (the new process is not started). In this case, **errno** is set to one of the following values.

E2BIG

Argument list exceeds 1024 bytes.

EINVAL

mode argument is invalid.

ENOENT

File or path is not found.

ENOEXEC

Specified file is not executable or has invalid executable-file format.

ENOMEM

Not enough memory is available to execute the new process.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions creates and executes a new process, passing each command-line argument as a separate parameter

and using the **PATH** environment variable to find the file to execute.

These functions validate their parameters. If either *cmdname* or *arg0* is an empty string or a null pointer, these functions generate an invalid parameter exception, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL**, and return -1. No new process is spawned.

Requirements

Routine	Required header	Compatibility
_spawnlp	<process.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wspawnlp	<stdio.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example in _spawn, _wspawn Functions.

.NET Framework Equivalent

- System::Diagnostics::Process Class
- System::Diagnostics::ProcessStartInfo Class

See Also

Reference

Process and Environment Control
_spawn, _wspawn Functions
abort
atexit
_exec, _wexec Functions
exit, _exit
_flushall
_getmbcp
_onexit, _onexit_m
_setmbcp
system, _wsystem

spawnlpe

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _spawnlpe instead.

_spawnlpe, _wspawnlpe

Creates and executes a new process.

```
intptr_t _spawnlpe(
   int mode,
      const char *cmdname,
   const char *arg0,
   const char *arg1,
   ... const char *argn,
   NULL,
   const char *const *envp
);
intptr_t _wspawnlpe(
   int mode,
   const wchar_t *cmdname,
   const wchar_t *arg0,
   const wchar_t *arg1,
   ... const wchar_t *argn,
   NULL,
   const wchar_t *const *envp
);
```

Parameters

mode

Execution mode for the calling process.

cmdname

Path of the file to be executed.

arg0, arg1, ... argn

List of pointers to arguments.

envp

Array of pointers to environment settings.

Return Value

The return value from a synchronous **_spawnlpe** or **_wspawnlpe** (**_P_WAIT** specified for *mode*) is the exit status of the new process. The return value from an asynchronous **_spawnlpe** or **_wspawnlpe** (**_P_NOWAIT** or **_P_NOWAITO** specified for *mode*) is the process handle. The exit status is 0 if the process terminated normally. You can set the exit status to a nonzero value if the spawned process specifically calls the **exit** routine with a nonzero argument. If the new process did not explicitly set a positive exit status, a positive exit status indicates an abnormal exit with an abort or an interrupt. A return value of –1 indicates an error (the new process is not started). In this case, **errno** is set to one of the following values.

E2BIG

Argument list exceeds 1024 bytes.

EINVAL

mode argument is invalid.

ENOENT

File or path is not found.

ENOEXEC

Specified file is not executable or has invalid executable-file format.

ENOMEM

Not enough memory is available to execute the new process.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions creates and executes a new process, passing each command-line argument as a separate parameter and passing an array of pointers to environment settings. These functions use the **PATH** environment variable to find the file to execute.

These functions validate their parameters. If either *cmdname* or *arg0* is an empty string or a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL**, and return -1. No new process is spawned.

Requirements

Routine	Required header	Compatibility
_spawnlpe	<process.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wspawnlpe	<stdio.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example in _spawn, _wspawn Functions.

.NET Framework Equivalent

- System::Diagnostics::Process Class
- System::Diagnostics::ProcessStartInfo Class

See Also

Reference

Process and Environment Control
_spawn, _wspawn Functions
abort
atexit
_exec, _wexec Functions
exit, _exit
_flushall
_getmbcp
_onexit, _onexit_m
_setmbcp
system, _wsystem

spawnv

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _spawnv instead.

_spawnv, _wspawnv

Creates and executes a new process.

```
intptr_t _spawnv(
   int mode,
      const char *cmdname,
   const char *const *argv
);
intptr_t _wspawnv(
   int mode,
   const wchar_t *cmdname,
   const wchar_t *const *argv
);
```

Parameters

mode

Execution mode for the calling process.

cmdname

Path of the file to be executed.

argv

Array of pointers to arguments.

Return Value

The return value from a synchronous **_spawnv** or **_wspawnv** (**_P_WAIT** specified for *mode*) is the exit status of the new process. The return value from an asynchronous **_spawnv** or **_wspawnv** (**_P_NOWAIT** or **_P_NOWAITO** specified for *mode*) is the process handle. The exit status is 0 if the process terminated normally. You can set the exit status to a nonzero value if the spawned process specifically calls the **exit** routine with a nonzero argument. If the new process did not explicitly set a positive exit status, a positive exit status indicates an abnormal exit with an abort or an interrupt. A return value of –1 indicates an error (the new process is not started). In this case, **errno** is set to one of the following values.

E2BIG

Argument list exceeds 1024 bytes.

EINVAL

mode argument is invalid.

ENOENT

File or path is not found.

ENOEXEC

Specified file is not executable or has invalid executable-file format.

ENOMEM

Not enough memory is available to execute the new process.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions creates and executes a new process, passing an array of pointers to command-line arguments.

These functions validate their parameters. If either *cmdname* or *argv* is a null pointer, or if *argv* points to null pointer, or *argv[0]* is an empty string, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL**, and return -1. No new process is spawned.

Requirements

Routine	Required header	Compatibility
_spawnv	<stdio.h> or <process.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wspawnv	<stdio.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example in _spawn, _wspawn Functions.

.NET Framework Equivalent

- System::Diagnostics::Process Class
- System::Diagnostics::ProcessStartInfo Class

See Also

Reference

Process and Environment Control
_spawn, _wspawn Functions
abort
atexit
_exec, _wexec Functions
exit, _exit
_flushall
_getmbcp
_onexit, _onexit_m
_setmbcp
system, _wsystem

spawnve

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _spawnve instead.

_spawnve, _wspawnve

Creates and executes a new process.

```
intptr_t _spawnve(
   int mode,
      const char *cmdname,
   const char *const *argv,
   const char *const *envp
);
intptr_t _wspawnve(
   int mode,
   const wchar_t *cmdname,
   const wchar_t *const *argv,
   const wchar_t *const *envp
);
```

Parameters

mode

Execution mode for a calling process.

cmdname

Path of the file to be executed.

argv

Array of pointers to arguments.

envp

Array of pointers to environment settings.

Return Value

The return value from a synchronous **_spawnve** or **_wspawnve** (**_P_WAIT** specified for *mode*) is the exit status of the new process. The return value from an asynchronous **_spawnve** or **_wspawnve** (**_P_NOWAIT** or **_P_NOWAITO** specified for *mode*) is the process handle. The exit status is 0 if the process terminated normally. You can set the exit status to a nonzero value if the spawned process specifically calls the **exit** routine with a nonzero argument. If the new process did not explicitly set a positive exit status, a positive exit status indicates an abnormal exit with an abort or an interrupt. A return value of –1 indicates an error (the new process is not started). In this case, **errno** is set to one of the following values.

E2BIG

Argument list exceeds 1024 bytes.

EINVAL

mode argument is invalid.

ENOENT

File or path is not found.

ENOEXEC

Specified file is not executable or has invalid executable-file format.

ENOMEM

Not enough memory is available to execute the new process.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions creates and executes a new process, passing an array of pointers to command-line arguments and an array of pointers to environment settings.

These functions validate their parameters. If either *cmdname* or *argv* is a null pointer, or if *argv* points to null pointer, or *argv[0]* is an empty string, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set *errno* to *EINVAL*, and return -1. No new process is spawned.

Requirements

Routine	Required header	Compatibility
_spawnve	<stdio.h> or <process.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wspawnve	<stdio.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

See the example in _spawn, _wspawn Functions.

.NET Framework Equivalent

- System::Diagnostics::Process Class
- System::Diagnostics::ProcessStartInfo Class

See Also

Reference

Process and Environment Control
_spawn, _wspawn Functions
abort
atexit
_exec, _wexec Functions
exit, _exit
_flushall
_getmbcp
_onexit, _onexit_m
_setmbcp
system, _wsystem

spawnvp

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _spawnvp instead.

_spawnvp, _wspawnvp

Create and execute a new process.

```
intptr_t _spawnvp(
   int mode,
   const char *cmdname,
   const char *const *argv
);
intptr_t _wspawnvp(
   int mode,
   const wchar_t *cmdname,
   const wchar_t *const *argv
);
```

Parameters

mode

Execution mode for calling process

cmdname

Path of file to be executed

argv

Array of pointers to arguments

Return Value

The return value from a synchronous **_spawnvp** or **_wspawnvp** (**_P_WAIT** specified for *mode*) is the exit status of the new process. The return value from an asynchronous **_spawnvp** or **_wspawnvp** (**_P_NOWAIT** or **_P_NOWAITO** specified for *mode*) is the process handle. The exit status is 0 if the process terminated normally. You can set the exit status to a nonzero value if the spawned process specifically calls the **exit** routine with a nonzero argument. If the new process did not explicitly set a positive exit status, a positive exit status indicates an abnormal exit with an abort or an interrupt. A return value of –1 indicates an error (the new process is not started). In this case, **errno** is set to one of the following values:

E2BIG

Argument list exceeds 1024 bytes

EINVAL

mode argument is invalid

ENOENT

File or path is not found

ENOEXEC

Specified file is not executable or has invalid executable-file format

ENOMEM

Not enough memory is available to execute new process

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

Remarks

Each of these functions creates and executes a new process, passing an array of pointers to command-line arguments and using the **PATH** environment variable to find the file to execute.

These functions validate their parameters. If either *cmdname* or *argv* is a null pointer, or if *argv* points to null pointer, or *argv[0]* is an empty string, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, these functions set **errno** to **EINVAL**, and return -1. No new process is spawned.

Requirements

Routine	Required header	Compatibility
_spawnvp	<stdio.h> or <process.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wspawnvp	<stdio.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example in _spawn, _wspawn Functions.

.NET Framework Equivalent

- System::Diagnostics::Process Class
- System::Diagnostics::ProcessStartInfo Class

See Also

Reference

Process and Environment Control
_spawn, _wspawn Functions
abort
atexit
_exec, _wexec Functions
exit, _exit
_flushall
_getmbcp
_onexit, _onexit_m
_setmbcp
system, _wsystem

spawnvpe

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _spawnvpe instead.

_spawnvpe, _wspawnvpe

Create and execute a new process.

```
intptr_t _spawnvpe(
   int mode,
      const char *cmdname,
   const char *const *argv,
   const char *const *envp
);
intptr_t _wspawnvpe(
   int mode,
   const wchar_t *cmdname,
   const wchar_t *const *argv,
   const wchar_t *const *envp
);
```

Parameters

mode

Execution mode for calling process

cmdname

Path of file to be executed

argv

Array of pointers to arguments

envp

Array of pointers to environment settings

Return Value

The return value from a synchronous **_spawnvpe** or **_wspawnvpe** (**_P_WAIT** specified for *mode*) is the exit status of the new process. The return value from an asynchronous **_spawnvpe** or **_wspawnvpe** (**_P_NOWAIT** or **_P_NOWAITO** specified for *mode*) is the process handle. The exit status is 0 if the process terminated normally. You can set the exit status to a nonzero value if the spawned process specifically calls the **exit** routine with a nonzero argument. If the new process did not explicitly set a positive exit status, a positive exit status indicates an abnormal exit with an abort or an interrupt. A return value of –1 indicates an error (the new process is not started). In this case, **errno** is set to one of the following values:

E2BIG

Argument list exceeds 1024 bytes

EINVAL

mode argument is invalid

ENOENT

File or path is not found

ENOEXEC

Specified file is not executable or has invalid executable-file format

ENOMEM

Not enough memory is available to execute new process

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

Remarks

Each of these functions creates and executes a new process, passing an array of pointers to command-line arguments and an array of pointers to environment settings. These functions use the **PATH** environment variable to find the file to execute.

These functions validate their parameters. If either *cmdname* or *argv* is a null pointer, or if *argv* points to null pointer, or *argv[0]* is an empty string, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, these functions set **errno** to **EINVAL**, and return -1. No new process is spawned.

Requirements

Routine	Required header	Compatibility
_spawnvpe	<stdio.h> or <process.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wspawnvpe	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example in _spawn, _wspawn Functions.

.NET Framework Equivalent

- System::Diagnostics::Process Class
- System::Diagnostics::ProcessStartInfo Class

See Also

Reference

abort
atexit
_exec, _wexec Functions
exit, _exit
_flushall
_getmbcp
_onexit, _onexit_m
_setmbcp
system, _wsystem

_splitpath, _wsplitpath

Break a path name into components. These functions are deprecated because more secure versions are available, see _splitpath_s, _wsplitpath_s.

```
void _splitpath(
   const char *path,
   char *drive,
   char *dir,
   char *fname,
   char *ext
);
void _wsplitpath(
   const wchar_t *path,
   wchar_t *drive,
   wchar_t *dir,
   wchar_t *fname,
   wchar_t *ext
);
```

Parameters

path

Full path

drive

Optional drive letter, followed by a colon (**:**)

dir

Optional directory path, including trailing slash. Forward slashes (**/**), backslashes (****), or both may be used.

fname

Base filename (no extension)

ext

Optional filename extension, including leading period (**.**)

Remarks

The **_splitpath** function breaks a path into its four components. **_splitpath** automatically handles multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the multibyte code page currently in use. **_wsplitpath** is a wide-character version of **_splitpath**; the arguments to **_wsplitpath** are wide-character strings. These functions behave identically otherwise

Security Note These functions incur a potential threat brought about by a buffer overrun problem. Buffer overrun problems are a frequent method of system attack, resulting in an unwarranted elevation of privilege. For more information, see Avoiding Buffer Overruns. More secure versions of these functions are availalble; see _splitpath_s, _wsplitpath_s.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tsplitpath	**_splitpath**	**_splitpath**	**_wsplitpath**

Each argument is stored in a buffer; the manifest constants **_MAX_DRIVE**, **_MAX_DIR**, **_MAX_FNAME**, and **_MAX_EXT** (defined in STDLIB.H) specify the maximum size necessary for each buffer. The other arguments point to buffers used to store the path elements. After a call to **_splitpath** is executed, these arguments contain empty strings for components not found in *path*. You can pass a **NULL** pointer to **_splitpath** for any component you don't need.

If *path* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **EINVAL**.

Requirements

Routine	Required header	Compatibility
_splitpath	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wsplitpath	<stdlib.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for _makepath.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

File Handling
_fullpath, _wfullpath
_getmbcp
_makepath, _wmakepath
_setmbcp

_splitpath_s, _wsplitpath_s

Break a path name into components. These are versions of _splitpath, _wsplitpath with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _splitpath_s(
   const char * path,
   char * drive,
   size_t driveNumberOfElements,
   char * dir,
   size_t dirNumberOfElements,
   char * fname,
   size_t nameNumberOfElements,
   char * ext,
   size_t extNumberOfElements
);
errno_t _wsplitpath_s(
   const wchar_t * path,
   wchar_t * drive,
   size_t driveNumberOfElements,
   wchar_t *dir,
   size_t dirNumberOfElements,
   wchar_t * fname,
   size_t nameNumberOfElements,
   wchar_t * ext,
   size_t extNumberOfElements
);
template <size_t drivesize, size_t dirsize, size_t fnamesize, size_t extsize>
errno_t _splitpath_s(
   const char *path,
   char (&drive)[drivesize],
   char (&dir)[dirsize],
   char (&fname)[fnamesize],
   char (&ext)[extsize]
); // C++ only
template <size_t drivesize, size_t dirsize, size_t fnamesize, size_t extsize>
errno_t _wsplitpath_s(
   const wchar_t *path,
   wchar_t (&drive)[drivesize],
   wchar_t (&dir)[dirsize],
   wchar_t (&fname)[fnamesize],
   wchar_t (&ext)[extsize]
); // C++ only
```

Parameters

[in] *path*

Full path

[out] *drive*

Optional drive letter, followed by a colon (**:**)

[in] *driveNumberOfElements*

The size of the *drive* buffer in single-byte or wide characters.

[out] *dir*

Optional directory path, including trailing slash. Forward slashes (**/**), backslashes (****), or both may be used.

[in] *dirNumberOfElements*

The size of the *dir* buffer in single-byte or wide characters.

[out] *fname*

Base filename (no extension)

[in] *nameNumberOfElements*

The size of the *fname* buffer in single-byte or wide characters.

[out] *ext*

Optional filename extension, including leading period (**.**)

[in] *extNumberOfElements*

The size of *ext* buffer in single-byte or wide characters.

Return Value

Zero if successful; an error code on failure.

Error Conditions

Condition	**Return Value**
path is **NULL**	**EINVAL**
drive is **NULL**, *driveNumberOfElements* is non-zero	**EINVAL**
drive is non-**NULL**, *driveNumberOfElements* is zero	**EINVAL**
dir is **NULL**, *dirNumberOfElements* is non-zero	**EINVAL**
dir is non-**NULL**, *dirNumberOfElements* is zero	**EINVAL**
fname is **NULL**, *nameNumberOfElements* is non-zero	**EINVAL**
fname is non-**NULL**, *nameNumberOfElements* is zero	**EINVAL**
ext is **NULL**, *extNumberOfElements* is non-zero	**EINVAL**
ext is non-**NULL**, *extNumberOfElements* is zero	**EINVAL**

If any of the above conditions occurs, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **EINVAL**.

If any of the buffers is too short to hold the result, these functions clear all the buffers to empty strings, set **errno** to **ERANGE**, and return **ERANGE**.

Remarks

The **_splitpath_s** function breaks a path into its four components. **_splitpath_s** automatically handles multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the multibyte code page currently in use. **_wsplitpath_s** is a wide-character version of **_splitpath_s**; the arguments to **_wsplitpath_s** are wide-character strings. These functions behave identically otherwise

Generic-Text Routine Mappings

TCHAR.H routine	**_UNICODE & _MBCS not defined**	**_MBCS defined**	**_UNICODE defined**
_tsplitpath	**_splitpath**	**_splitpath**	**_wsplitpath**

Each argument is stored in a buffer; the manifest constants **_MAX_DRIVE**, **_MAX_DIR**, **_MAX_FNAME**, and **_MAX_EXT** (defined in STDLIB.H) specify the maximum size necessary for each buffer. The other arguments point to buffers used to store the path elements. After a call to **_splitpath_s** is executed, these arguments contain empty strings for components not found in *path*. You can pass a **NULL** pointer to **_splitpath_s** for any component you don't need.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically, eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Requirements

Routine	Required header	Compatibility
_splitpath_s	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wsplitpath_s	<stdlib.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for _makepath_s, _wmakepath_s.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

File Handling
_splitpath, _wsplitpath
_fullpath, _wfullpath
_getmbcp
_makepath, _wmakepath
_setmbcp

sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l

Write formatted data to a string. These functions are deprecated because more secure versions are available; see sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l.

```
int sprintf(
   char *buffer,
   const char *format [,
      argument] ...
);
int _sprintf_l(
   char *buffer,
   const char *format,
   locale_t locale [,
      argument] ...
);
int swprintf(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format [,
      argument]...
);
int _swprintf_l(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format,
   locale_t locale [,
      argument] ...
);
int __swprintf_l(
   wchar_t *buffer,
   const wchar_t *format,
   locale_t locale [,
      argument] ...
);
template <size_t size>
int sprintf(
   char (&buffer)[size],
   const char *format [,
      argument] ...
); // C++ only
template <size_t size>
int _sprintf_l(
   char (&buffer)[size],
   const char *format,
   locale_t locale [,
      argument] ...
); // C++ only
template <size_t size>
int swprintf(
   wchar_t (&buffer)[size],
   size_t count,
   const wchar_t *format [,
      argument]...
); // C++ only
template <size_t size>
int _swprintf_l(
   wchar_t (&buffer)[size],
   size_t count,
   const wchar_t *format,
   locale_t locale [,
      argument] ...
); // C++ only
```

Parameters

buffer

Storage location for output

count

Maximum number of characters to store in the Unicode version of this function.

format

Format-control string

argument

Optional arguments

locale

The locale to use.

For more information, see Format Specifications.

Return Value

The number of characters written, or –1 if an error occurred. If *buffer* or *format* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

sprintf returns the number of bytes stored in *buffer*, not counting the terminating null character. **swprintf** returns the number of wide characters stored in *buffer*, not counting the terminating null wide character.

Remarks

The **sprintf** function formats and stores a series of characters and values in *buffer*. Each *argument* (if any) is converted and output according to the corresponding format specification in *format*. The format consists of ordinary characters and has the same form and function as the *format* argument for printf. A null character is appended after the last character written. If copying occurs between strings that overlap, the behavior is undefined.

Security Note
Using **sprintf**, there is no way to limit the number of characters written, which means that code using **sprintf** is susceptible to buffer overruns. Consider using the related function _snprintf, which specifies a maximum number of characters to be written to *buffer*, or use _scprintf to determine how large a buffer is required. Also, ensure that *format* is not a user-defined string.

swprintf is a wide-character version of **sprintf**; the pointer arguments to **swprintf** are wide-character strings. Detection of encoding errors in **swprintf** may differ from that in **sprintf**. **swprintf** and **fwprintf** behave identically except that **swprintf** writes output to a string rather than to a destination of type **FILE**, and **swprintf** requires the *count* parameter to specify the maximum number of characters to be written. The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

In Visual C++ 2005, **swprintf** conforms to the ISO C Standard, which requires the second parameter, *count*, of type **size_t**. To force the old nonstandard behavior, define **_CRT_NON_CONFORMING_SWPRINTFS**. In a future version, the old behavior may be removed, so code should be changed to use the new conformant behavior.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	**_UNICODE & _MBCS not defined**	**_MBCS defined**	**_UNICODE defined**
_stprintf	**sprintf**	**sprintf**	**swprintf**
_stprintf_l	**_sprintf_l**	**_sprintf_l**	**__swprintf_l**

Requirements

Routine	**Required header**	**Compatibility**

sprintf, **_sprintf_l**	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
swprintf, **_swprintf_l**	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_sprintf.c
// compile with: /W3
// This program uses sprintf to format various
// data and place them in the string named buffer.

#include <stdio.h>

int main( void )
{
   char  buffer[200], s[] = "computer", c = 'l';
   int   i = 35, j;
   float fp = 1.7320534f;

   // Format and print various data:
   j  = sprintf( buffer,     "   String:    %s\n", s ); // C4996
   j += sprintf( buffer + j, "   Character: %c\n", c ); // C4996
   j += sprintf( buffer + j, "   Integer:   %d\n", i ); // C4996
   j += sprintf( buffer + j, "   Real:      %f\n", fp );// C4996
   // Note: sprintf is deprecated; consider using sprintf_s instead

   printf( "Output:\n%s\ncharacter count = %d\n", buffer, j );
}
```

Output

```
Output:
   String:    computer
   Character: l
   Integer:   35
   Real:      1.732053

character count = 79
```

```
// crt_swprintf.c
// wide character example
// also demonstrates swprintf returning error code
#include <stdio.h>

int main( void )
{
   wchar_t buf[100];
   int len = swprintf( buf, 100, L"%s", L"Hello world" );
   printf( "wrote %d characters\n", len );
   len = swprintf( buf, 100, L"%s", L"Hello\xffff world" );
   // swprintf fails because string contains WEOF (\xffff)
   printf( "wrote %d characters\n", len );
}
```

Output

```
wrote 11 characters
wrote -1 characters
```

.NET Framework Equivalent

System::String::Format

See Also

Reference

Stream I/O

fprintf, _fprintf_l, fwprintf, _fwprintf_l

printf, _printf_l, wprintf, _wprintf_l

scanf, _scanf_l, wscanf, _wscanf_l

sscanf, _sscanf_l, swscanf, _swscanf_l

vprintf Functions

_sprintf_p, _sprintf_p_l, _swprintf_p, _swprintf_p_l

Write formatted data to a string with the ability to specify the order that the parameters are used in the format string.

```
int _sprintf_p(
   char *buffer,
   size_t sizeOfBuffer,
   const char *format [,
      argument] ...
);
int _sprintf_p_l(
   char *buffer,
   size_t sizeOfBuffer,
   const char *format,
   locale_t locale [,
      argument] ...
);
int _swprintf_p(
   wchar_t *buffer,
   size_t sizeOfBuffer,
   const wchar_t *format [,
      argument]...
);
int _swprintf_p_l(
   wchar_t *buffer,
   size_t sizeOfBuffer,
   const wchar_t *format,
   locale_t locale [,
      argument] …
);
```

Parameters

buffer

Storage location for output

sizeOfBuffer

Maximum number of characters to store.

format

Format-control string

argument

Optional arguments

locale

The locale to use.

Return Value

The number of characters written, or –1 if an error occurred.

Remarks

The **_sprintf_p** function formats and stores a series of characters and values in *buffer*. Each *argument* (if any) is converted and output according to the corresponding format specification in *format*. The format consists of ordinary characters and has the same form and function as the *format* argument for **printf_p**. A **NULL** character is appended after the last character written. If copying occurs between strings that overlap, the behavior is undefined. The difference between **_sprintf_p** and **sprintf_s** is that **_sprintf_p** supports positional parameters, which allows specifying the order in which the arguments are used in the format string. For more information, see printf_p Positional Parameters.

_swprintf_p is a wide-character version of **_sprintf_p**; the pointer arguments to **_swprintf_p** are wide-character strings. Detection of encoding errors in **_swprintf_p** may differ from that in **_sprintf_p**. **_swprintf_p** and **fwprintf_p** behave

identically except that **_swprintf_p** writes output to a string rather than to a destination of type **FILE**, and **_swprintf_p** requires the *count* parameter to specify the maximum number of characters to be written. The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

_sprintf_p returns the number of bytes stored in *buffer*, not counting the terminating **NULL** character. **_swprintf_p** returns the number of wide characters stored in *buffer*, not counting the terminating **NULL** wide character. If *buffer* or *format* is a null pointer, or if the format string contains invalid formatting characters, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_stprintf_p	**_sprintf_p**	**_sprintf_p**	**_swprintf_p**
_stprintf_p_l	**_sprintf_p_l**	**_sprintf_p_l**	**_swprintf_p_l**

Requirements

Routine	Required header	Compatibility
_sprintf_p, **_sprintf_p_l**	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_swprintf_p, **_swprintf_p_l**	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_sprintf_p.c
// This program uses _sprintf_p to format various
// data and place them in the string named buffer.
//

#include <stdio.h>

int main( void )
{
    char     buffer[200],
            s[] = "computer", c = 'l';
    int      i = 35,
            j;
    float    fp = 1.7320534f;

    // Format and print various data:
    j  = _sprintf_p( buffer, 200,
                     "   String:    %s\n", s );
    j += _sprintf_p( buffer + j, 200 - j,
                     "   Character: %c\n", c );
    j += _sprintf_p( buffer + j, 200 - j,
                     "   Integer:   %d\n", i );
    j += _sprintf_p( buffer + j, 200 - j,
                     "   Real:      %f\n", fp );

    printf( "Output:\n%s\ncharacter count = %d\n",
            buffer, j );
}
```

Output

```
Output:
   String:    computer
   Character: l
   Integer:   35
   Real:      1.732053
```

```
character count = 79
```

```
// crt_swprintf_p.c
// This is the wide character example which
// also demonstrates _swprintf_p returning
// error code.
#include <stdio.h>

#define BUFFER_SIZE 100

int main( void )
{
    wchar_t buffer[BUFFER_SIZE];
    int     len;

    len = _swprintf_p(buffer, BUFFER_SIZE, L"%2$s %1$d",
                      0, L" marbles in your head.");
    _printf_p( "Wrote %d characters\n", len );

    // _swprintf_p fails because string contains WEOF (\xffff)
    len = _swprintf_p(buffer, BUFFER_SIZE, L"%s",
                      L"Hello\xffff world" );
    _printf_p( "Wrote %d characters\n", len );
}
```

Output

```
Wrote 24 characters
Wrote -1 characters
```

.NET Framework Equivalent

System::String::Format

See Also

Reference

Stream I/O
_fprintf_p, _fprintf_p_l, _fwprintf_p, _fwprintf_p_l
fprintf, _fprintf_l, fwprintf, _fwprintf_l
_printf_p, _printf_p_l, _wprintf_p, _wprintf_p_l
printf, _printf_l, wprintf, _wprintf_l
scanf, _scanf_l, wscanf, _wscanf_l
sscanf, _sscanf_l, swscanf, _swscanf_l
sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l
vprintf Functions
printf_p Positional Parameters

sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l

Write formatted data to a string. These are versions of sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l with security enhancements as described in Security Enhancements in the CRT.

```
int sprintf_s(
   char *buffer,
   size_t sizeOfBuffer,
   const char *format [,
      argument] ...
);
int _sprintf_s_l(
   char *buffer,
   size_t sizeOfBuffer,
   const char *format,
   locale_t locale [,
      argument] ...
);
int swprintf_s(
   wchar_t *buffer,
   size_t sizeOfBuffer,
   const wchar_t *format [,
      argument]...
);
int _swprintf_s_l(
   wchar_t *buffer,
   size_t sizeOfBuffer,
   const wchar_t *format,
   locale_t locale [,
      argument]…
);
template <size_t size>
int sprintf_s(
   char (&buffer)[size],
   const char *format [,
      argument] ...
); // C++ only
template <size_t size>
int swprintf_s(
   wchar_t (&buffer)[size],
   const wchar_t *format [,
      argument]...
); // C++ only
```

Parameters

buffer

Storage location for output

sizeOfBuffer

Maximum number of characters to store.

format

Format-control string

argument

Optional arguments

locale

The locale to use.

For more information, see Format Specifications.

Return Value

The number of characters written, or –1 if an error occurred. If *buffer* or *format* is a null pointer, **sprintf_s** and **swprintf_s** return -1 and set **errno** to **EINVAL**.

sprintf_s returns the number of bytes stored in *buffer*, not counting the terminating null character. **swprintf_s** returns the number of wide characters stored in *buffer*, not counting the terminating null wide character.

Remarks

The **sprintf_s** function formats and stores a series of characters and values in *buffer*. Each *argument* (if any) is converted and output according to the corresponding format specification in *format*. The format consists of ordinary characters and has the same form and function as the *format* argument for printf. A null character is appended after the last character written. If copying occurs between strings that overlap, the behavior is undefined.

One main difference between **sprintf_s** and **sprintf** is that **sprintf_s** checks the format string for valid formatting characters, whereas **sprintf** only checks if the format string or buffer are **NULL** pointers. If either check fails, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns -1 and sets **errno** to **EINVAL**.

The other main difference between **sprintf_s** and **sprintf** is that **sprintf_s** takes a length parameter specifying the size of the output buffer in characters. If the buffer is too small for the text being printed then the buffer is set to an empty string and the invalid parameter handler is invoked. Unlike **snprintf**, **sprintf_s** guarantees that the buffer will be null-terminated (unless the buffer size is zero).

swprintf_s is a wide-character version of **sprintf_s**; the pointer arguments to **swprintf_s** are wide-character strings. Detection of encoding errors in **swprintf_s** may differ from that in **sprintf_s**. The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

There are versions of **sprintf_s** that offer additional control over what happens if the buffer is too small. For more information, see _snprintf_s, _snprintf_s_l, _snwprintf_s, _snwprintf_s_l.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_stprintf_s	**sprintf_s**	**sprintf_s**	**swprintf_s**
_stprintf_s_l	**_sprintf_s_l**	**_sprintf_s_l**	**_swprintf_s_l**

Requirements

Routine	Required header	Compatibility
sprintf_s, **_sprintf_s_l**	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
swprintf_s, **_swprintf_s_l**	<stdio.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_sprintf_s.c
// This program uses sprintf_s to format various
// data and place them in the string named buffer.
//

#include <stdio.h>

int main( void )
{
   char  buffer[200], s[] = "computer", c = 'l';
```

```
    int   i = 35, j;
    float fp = 1.7320534f;

    // Format and print various data:
    j  = sprintf_s( buffer, 200,     "   String:    %s\n", s );
    j += sprintf_s( buffer + j, 200 - j, "   Character: %c\n", c );
    j += sprintf_s( buffer + j, 200 - j, "   Integer:   %d\n", i );
    j += sprintf_s( buffer + j, 200 - j, "   Real:      %f\n", fp );

    printf_s( "Output:\n%s\ncharacter count = %d\n", buffer, j );
  }
```

Output

```
Output:
   String:    computer
   Character: l
   Integer:   35
   Real:      1.732053

character count = 79
```

```
  // crt_swprintf_s.c
  // wide character example
  // also demonstrates swprintf_s returning error code
  #include <stdio.h>

  int main( void )
  {
     wchar_t buf[100];
     int len = swprintf_s( buf, 100, L"%s", L"Hello world" );
     printf( "wrote %d characters\n", len );
     len = swprintf_s( buf, 100, L"%s", L"Hello\xffff world" );
     // swprintf_s fails because string contains WEOF (\xffff)
     printf( "wrote %d characters\n", len );
  }
```

Output

```
wrote 11 characters
wrote -1 characters
```

.NET Framework Equivalent

System::String::Format

See Also

Reference

Stream I/O
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
scanf, _scanf_l, wscanf, _wscanf_l
sscanf, _sscanf_l, swscanf, _swscanf_l
vprintf Functions

sqrt, sqrtf

Calculates the square root.

```
double sqrt(
   double x
);
float sqrt(
   float x
);  // C++ only
long double sqrt(
   long double x
);  // C++ only
float sqrtf(
   float x
);
```

Parameters

x

Nonnegative floating-point value

Remarks

C++ allows overloading, so users can call overloads of **sqrt** that take float or long double types. In a C program, **sqrt** always takes and returns double.

Return Value

The **sqrt** function returns the square-root of *x*. If *x* is negative, **sqrt** returns an indefinite, by default.

Input	SEH Exception	Matherr Exception
± QNAN,IND	none	_DOMAIN
- ∞	INVALID	_DOMAIN
x>0	INVALID	_DOMAIN

Requirements

Routine	Required header	Compatibility
sqrt, **sqrtf**	<math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_sqrt.c
// This program calculates a square root.

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   double question = 45.35, answer;

   answer = sqrt( question );
   if( question < 0 )
      printf( "Error: sqrt returns %.2f\n, answer" );
```

```
    else
       printf( "The square root of %.2f is %.2f\n", question, answer );
 }
```

Output

```
The square root of 45.35 is 6.73
```

.NET Framework Equivalent

System::Math::Sqrt

See Also

Reference

Floating-Point Support
exp, expf
log, logf, log10, log10f
pow, powf

srand

Sets a random starting point.

```
void srand(
   unsigned int seed
);
```

Parameters

seed

Seed for random-number generation

Remarks

The **srand** function sets the starting point for generating a series of pseudorandom integers. To reinitialize the generator, use 1 as the *seed* argument. Any other value for *seed* sets the generator to a random starting point. **rand** retrieves the pseudorandom numbers that are generated. Calling **rand** before any call to **srand** generates the same sequence as calling **srand** with *seed* passed as 1.

Requirements

Routine	Required header	Compatibility
srand	<stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for rand.

.NET Framework Equivalent

System::Random Class

See Also

Reference

Floating-Point Support

rand

sscanf, _sscanf_l, swscanf, _swscanf_l

Read formatted data from a string. These functions are deprecated because more secure versions are available; see sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l.

```
int sscanf(
   const char *buffer,
   const char *format [,
      argument ] ...
);
int _sscanf_l(
   const char *buffer,
   const char *format,
   locale_t locale [,
      argument ] ...
);
int swscanf(
   const wchar_t *buffer,
   const wchar_t *format [,
      argument ] ...
);
int _swscanf_l(
   const wchar_t *buffer,
   const wchar_t *format,
   locale_t locale [,
      argument ] ...
);
```

Parameters

buffer

Stored data

format

Format-control string. For more information, see Format Specifications.

argument

Optional arguments

locale

The locale to use

Return Value

Each of these functions returns the number of fields successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned. The return value is **EOF** for an error or if the end of the string is reached before the first conversion.

If *buffer* or *format* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **sscanf** function reads data from *buffer* into the location given by each *argument*. Every *argument* must be a pointer to a variable with a type that corresponds to a type specifier in *format*. The *format* argument controls the interpretation of the input fields and has the same form and function as the *format* argument for the **scanf** function. If copying takes place between strings that overlap, the behavior is undefined.

Security Note

When reading a string with **sscanf**, always specify a width for the **%s** format (for example, "%32s" instead of "%s"); otherwise, improperly formatted input can easily cause a buffer overrun.

swscanf is a wide-character version of **sscanf**; the arguments to **swscanf** are wide-character strings. **sscanf** does not handle multibyte hexadecimal characters. **swscanf** does not handle Unicode full-width hexadecimal or "compatibility zone" characters. Otherwise, **swscanf** and **sscanf** behave identically.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_stscanf	**sscanf**	**sscanf**	**swscanf**
_stscanf_l	**_sscanf_l**	**_sscanf_l**	**_swscanf_l**

Requirements

Routine	Required header	Compatibility
sscanf, **_sscanf_l**	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
swscanf, **_swscanf_l**	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_sscanf.c
// compile with: /W3
// This program uses sscanf to read data items
// from a string named tokenstring, then displays them.

#include <stdio.h>

int main( void )
{
   char  tokenstring[] = "15 12 14...";
   char  s[81];
   char  c;
   int   i;
   float fp;

   // Input various data from tokenstring:
   // max 80 character string:
   sscanf( tokenstring, "%80s", s ); // C4996
   sscanf( tokenstring, "%c", &c );  // C4996
   sscanf( tokenstring, "%d", &i );  // C4996
   sscanf( tokenstring, "%f", &fp ); // C4996
   // Note: sscanf is deprecated; consider using sscanf_s instead

   // Output the data read
   printf( "String    = %s\n", s );
   printf( "Character = %c\n", c );
   printf( "Integer:  = %d\n", i );
   printf( "Real:     = %f\n", fp );
}
```

Output

```
String    = 15
```

```
Character = 1
Integer:  = 15
Real:     = 15.000000
```

.NET Framework Equivalent

See **Parse** methods, such as System::Double::Parse.

See Also

Reference

Stream I/O

fscanf, _fscanf_l, fwscanf, _fwscanf_l

scanf, _scanf_l, wscanf, _wscanf_l

sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l

_snprintf, _snprintf_l, _snwprintf, _snwprintf_l

sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l

Read formatted data from a string. These are versions of sscanf, _sscanf_l, swscanf, _swscanf_l with security enhancements as described in Security Enhancements in the CRT.

```
int sscanf_s(
   const char *buffer,
   const char *format [,
      argument ] ...
);
int _sscanf_s_l(
   const char *buffer,
   const char *format,
   locale_t locale [,
      argument ] ...
);
int swscanf_s(
   const wchar_t *buffer,
   const wchar_t *format [,
      argument ] ...
);
int _swscanf_s_l(
   const wchar_t *buffer,
   const wchar_t *format,
   locale_t locale [,
      argument ] ...
);
```

Parameters

buffer

Stored data

format

Format-control string. For more information, see Format Specifications.

argument

Optional arguments

locale

The locale to use

Return Value

Each of these functions returns the number of fields successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned. The return value is **EOF** for an error or if the end of the string is reached before the first conversion.

If *buffer* or *format* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **sscanf_s** function reads data from *buffer* into the location given by each *argument*. The arguments after the format string specify pointers to variables with a type that corresponds to a type specifier in *format*. Unlike the less secure version **sscanf**, a buffer size parameter *sizeOfBuffer* is required when using the type field characters **c**, **C**, **s**, **S** and **[**. This parameter must be supplied as an additional parameter after each buffer which requires it. For more information, see scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l and scanf Type Field Characters.

Note

The size parameter is of type **unsigned**, not **size_t**.

The *format* argument controls the interpretation of the input fields and has the same form and function as the *format* argument for the **scanf_s** function. If copying takes place between strings that overlap, the behavior is undefined.

swscanf_s is a wide-character version of **sscanf_s;** the arguments to **swscanf_s** are wide-character strings. **sscanf_s** does not handle multibyte hexadecimal characters. **swscanf_s** does not handle Unicode full width hexadecimal or "compatibility zone" characters. Otherwise, **swscanf_s** and **sscanf_s** behave identically.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_stscanf_s	**sscanf_s**	**sscanf_s**	**swscanf_s**
_stscanf_s_l	**_sscanf_s_l**	**_sscanf_s_l**	**_swscanf_s_l**

Requirements

Routine	Required header	Compatibility
sscanf_s, **_sscanf_s_l**	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
swscanf_s, **_swscanf_s_l**	<stdio.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_sscanf_s.c
// This program uses sscanf_s to read data items
// from a string named tokenstring, then displays them.

#include <stdio.h>

int main( void )
{
   char  tokenstring[] = "15 12 14...";
   char  s[81];
   char  c;
   int   i;
   float fp;

   // Input various data from tokenstring:
   // max 80 character string plus NULL terminator
   sscanf_s( tokenstring, "%s", s, sizeof(s) );
   sscanf_s( tokenstring, "%c", &c, sizeof(char) );
   sscanf_s( tokenstring, "%d", &i );
   sscanf_s( tokenstring, "%f", &fp );

   // Output the data read
   printf_s( "String    = %s\n", s );
   printf_s( "Character = %c\n", c );
   printf_s( "Integer:  = %d\n", i );
   printf_s( "Real:     = %f\n", fp );
}
```

Output

```
String    = 15
```

```
Character = 1
Integer:  = 15
Real:     = 15.000000
```

.NET Framework Equivalent

See **Parse** methods, such as System::Double::Parse.

See Also

Reference

Stream I/O

fscanf, _fscanf_l, fwscanf, _fwscanf_l

scanf, _scanf_l, wscanf, _wscanf_l

sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l

_snprintf, _snprintf_l, _snwprintf, _snwprintf_l

_stat, _wstat Functions

Get status information on a file.

```
int _stat(
   const char *path,
   struct _stat *buffer
);
int _stat32(
   const char *path,
   struct __stat32 *buffer
);
int _stat64(
   const char *path,
   struct __stat64 *buffer
);
int _stati64(
   const char *path,
   struct _stati64 *buffer
);
int _stat32i64(str
   const char *path,
      struct _stat32i64 *buffer
);
int _stat64i32(str
      const char *path,
   struct _stat64i32 *buffer
);
int _wstat(
   const wchar_t *path,
   struct _stat *buffer
);
int _wstat32(
   const wchar_t *path,
   struct __stat32 *buffer
);
int _wstat64(
   const wchar_t *path,
   struct __stat64 *buffer
);
int _wstati64(
   const wchar_t *path,
   struct _stati64 *buffer
);
int _wstat32i64(
   const wchar_t *path,
   struct _stat32i64 *buffer
);
int _wstat64i32(
   const wchar_t *path,
   struct _stat64i32 *buffer
);
```

Parameters

path

Pointer to a string containing the path of existing file.

buffer

Pointer to structure that stores results.

Return Value

Each of these functions returns 0 if the file-status information is obtained. A return value of –1 indicates an error, in which case

errno is set to **ENOENT**, indicating that the filename or path could not be found. A return value of **EINVAL** indicates an invalid parameter; **errno** is also set to **EINVAL** in this case.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on this, and other, return codes.

The date stamp on a file can be represented if it is later than midnight, January 1, 1970, and before 23:59:59, December 31, 3000, UTC, unless you use **_stat32** or **_wstat32**, or have defined **_USE_32BIT_TIME_T**, in which case the date can be represented only until 03:14:07 January 19, 2038, UTC.

Remarks

The **_stat** function obtains information about the file or directory specified by *path* and stores it in the structure pointed to by *buffer*. **_stat** automatically handles multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the multibyte code page currently in use.

_wstat is a wide-character version of **_stat**; the *path* argument to **_wstat** is a wide-character string. **_wstat** and **_stat** behave identically except that **_wstat** does not handle multibyte-character strings.

Variations of these functions support 32- or 64-bit time types, and 32- or 64-bit file lengths. The first numerical suffix (**32** or **64**) indicates the size of the time type used; the second suffix is either **i32** or **i64**, indicating whether the file size is represented as a 32-bit or 64-bit integer.

In Visual C++ 2005, **_stat** is equivalent to **_stat64i32**, and **struct _stat** contains a 64-bit time. This is true unless **_USE_32BIT_TIME_T** is defined, in which case the old behavior is in effect; **_stat** uses a 32-bit time, and **struct _stat** contains a 32-bit time. The same is true for **_stati64**.

This function validates its parameters. If either *path* or *buffer* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation.

Time Type and File Length Type Variations of _stat

Functions	_USE_32BIT_TIME_T defined?	Time type	File length type
_stat, **_wstat**	Not defined	64-bit	32-bit
_stat, **_wstat**	Defined	32-bit	32-bit
_stat32, **_wstat32**	Not affected by the macro definition	32-bit	32-bit
_stat64, **_wstat64**	Not affected by the macro definition	64-bit	64-bit
_stati64, **_wstati64**	Not defined	64-bit	64-bit
_stati64, **_wstati64**	Defined	32-bit	64-bit
_stat32i64, **_wstat32i64**	Not affected by the macro definition	32-bit	64-bit
_stat64i32, **_wstat64i32**	Not affected by the macro definition	64-bit	32-bit

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tstat	**_stat**	**_stat**	**_wstat**
_tstat64	**_stat64**	**_stat64**	**_wstat64**
_tstati64	**_stati64**	**_stati64**	**_wstati64**
_tstat32i64	**_stat32i64**	**_stat32i64**	**_wstat32i64**
_tstat64i32	**_stat64i32**	**_stat64i32**	**_wstat64i32**

The **_stat** structure, defined in SYS\STAT.H, includes the following fields.

st_gid

Numeric identifier of group that owns the file (UNIX-specific) This field will always be zero on Windows systems. A redirected file is classified as a Windows file.

st_atime

Time of last access of file. Valid on NTFS but not on FAT formatted disk drives.

st_ctime

Time of creation of file. Valid on NTFS but not on FAT formatted disk drives.

st_dev

Drive number of the disk containing the file (same as **st_rdev**).

st_ino

Number of the information node (the *inode*) for the file (UNIX-specific). On UNIX file systems, the *inode* describes the file date and time stamps, permissions, and content. When files are hard-linked to one another, they share the same *inode*. The *inode*, and therefore **st_ino**, has no meaning in the FAT, HPFS, or NTFS file systems.

st_mode

Bit mask for file-mode information. The **_S_IFDIR** bit is set if *path* specifies a directory; the **_S_IFREG** bit is set if *path* specifies an ordinary file or a device. User read/write bits are set according to the file's permission mode; user execute bits are set according to the filename extension.

st_mtime

Time of last modification of file.

st_nlink

Always 1 on non-NTFS file systems.

st_rdev

Drive number of the disk containing the file (same as **st_dev**).

st_size

Size of the file in bytes; a 64-bit integer for variations with the **i64** suffix**.**

st_uid

Numeric identifier of user who owns file (UNIX-specific). This field will always be zero on Windows systems. A redirected file is classified as a Windows file.

If *path* refers to a device, the **st_size**, various time fields, **st_dev**, and **st_rdev** fields in the **_stat** structure are meaningless. Because STAT.H uses the _dev_t type that is defined in TYPES.H, you must include TYPES.H before STAT.H in your code.

Requirements

Routine	Required header	Optional headers	Compatibility
_stat, **_stat32**, **_stat64**, **_stati64**, **_stat32i64**, **_stat64i32**	<sys/types.h> followed by <sys/stat.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wstat, **_wstat32**, **_wstat64**, **_wstati64**, **_wstat32i64**, **_wstat64i32**	<sys/types.h> followed by <sys/stat.h> or <wchar.h>	<errno.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_stat.c
// This program uses the _stat function to
```

```
// report information about the file named crt_stat.c.

#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <errno.h>

int main( void )
{
   struct _stat buf;
   int result;
   char timebuf[26];
   char* filename = "crt_stat.c";
   errno_t err;

   // Get data associated with "crt_stat.c":
   result = _stat( filename, &buf );

   // Check if statistics are valid:
   if( result != 0 )
   {
      perror( "Problem getting information" );
      switch (errno)
      {
         case ENOENT:
           printf("File %s not found.\n", filename);
           break;
         case EINVAL:
           printf("Invalid parameter to _stat.\n");
           break;
         default:
           /* Should never be reached. */
           printf("Unexpected error in _stat.\n");
      }
   }
   else
   {
      // Output some of the statistics:
      printf( "File size     : %ld\n", buf.st_size );
      printf( "Drive         : %c:\n", buf.st_dev + 'A' );
      err = ctime_s(timebuf, 26, &buf.st_mtime);
      if (err)
      {
         printf("Invalid arguments to ctime_s.");
         exit(1);
      }
      printf( "Time modified : %s", timebuf );
   }
}
```

Sample Output

```
File size     : 732
Drive         : C:
Time modified : Thu Feb 07 14:39:36 2002
```

.NET Framework Equivalent

- System::IO::File::GetAttributes
- System::IO::File::GetCreationTime
- System::IO::File::GetLastAccessTime
- System::IO::File::GetLastWriteTime

See Also

Reference

File Handling
_access, _waccess
_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32
_getmbcp
_setmbcp

_status87, _statusfp, _statusfp2

Get the floating point status word.

```
unsigned int _status87( void );
unsigned int _statusfp( void );
void _statusfp2(unsigned int *px86, unsigned int *pSSE2)
```

Parameters

px86

This address is filled with the status word for the x87 floating point unit.

pSSE2

This address is filled with the status word for the SSE2 floating point unit.

Return Value

For **_status87** and **_statusfp**, the bits in the value returned indicate the floating-point status. See the FLOAT.H include file for a complete definition of the bits returned by **_status87**. Many math library functions modify the 8087/80287 status word, with unpredictable results. Return values from **_clear87** and **_status87** are more reliable if fewer floating-point operations are performed between known states of the floating-point status word. **_statusfp2** has no return value.

Remarks

The **_status87** function gets the floating-point status word. The status word is a combination of the 8087/80287/80387 status word and other conditions detected by the 8087/80287/80387 exception handler, such as floating-point stack overflow and underflow. Unmasked exceptions are checked for before returning the contents of the status word. This means that the caller is informed of pending exceptions.

_statusfp is a platform-independent, portable version of **_status87**. It is identical to **_status87** on Intel (x86) platforms and is also supported by the MIPS platform. To ensure that your floating-point code is portable to MIPS, use **_statusfp**. If you are only targeting x86 platforms, use either **_status87** or **_statusfp**.

_statusfp2 is recommended for chips (such as the Pentium IV and later) that have both an x87 and an SSE2 floating point processor. For **_statusfp2**, the addresses are filled in with the floating-point status word for both the x87 or the SSE2 floating-point processor. When using a chip that supports x87 and SSE2 floating point processors, EM_AMBIGUOUS is set to 1 if **_statusfp** or **_controlfp** is used and the action was ambiguous because it could refer to the x87 or the SSE2 floating-point status word.

These functions are deprecated when compiling with /clr (Common Language Runtime Compilation) or **/clr:pure** because the common language runtime only supports the default floating-point precision.

Requirements

Routine	Required header	Compatibility
_status87, **_statusfp**, **_statusfp2**	<float.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_status87.c
// This program creates various floating-point errors and
// then uses _status87 to display messages indicating these problems.
// Compile this program with optimizations disabled (/Od). Otherwise,
// the optimizer removes the code related to the unused floating-
// point values.
//
```

```
#include <stdio.h>
#include <float.h>

int main( void )
{
   double a = 1e-40, b;
   float  x, y;

   printf( "Status = %.4x - clear\n",_status87() );

   // Assignment into y is inexact & underflows:
   y = a;
   printf( "Status = %.4x - inexact, underflow\n", _status87() );

   // y is denormal:
   b = y;
   printf( "Status = %.4x - inexact underflow, denormal\n",
           _status87() );

   // Clear user 8087:
   _clear87();
}
```

Output

```
Status = 0000 - clear
Status = 0003 - inexact, underflow
Status = 80003 - inexact underflow, denormal
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Floating-Point Support
_clear87, _clearfp
_control87, _controlfp, __control87_2

strcat, wcscat, _mbscat

Append a string. These functions are deprecated because more secure versions are available; see strcat_s, wcscat_s, _mbscat_s.

```
char *strcat(
   char *strDestination,
   const char *strSource
);
wchar_t *wcscat(
   wchar_t *strDestination,
   const wchar_t *strSource
);
unsigned char *_mbscat(
   unsigned char *strDestination,
   const unsigned char *strSource
);
template <size_t size>
char *strcat(
   char (&strDestination)[size],
   const char *strSource
); // C++ only
template <size_t size>
wchar_t *wcscat(
   wchar_t (&strDestination)[size],
   const wchar_t *strSource
); // C++ only
template <size_t size>
unsigned char *_mbscat(
   unsigned char (&strDestination)[size],
   const unsigned char *strSource
); // C++ only
```

Parameters

strDestination

Null-terminated destination string.

strSource

Null-terminated source string.

Return Value

Each of these functions returns the destination string (*strDestination*). No return value is reserved to indicate an error.

Remarks

The **strcat** function appends *strSource* to *strDestination* and terminates the resulting string with a null character. The initial character of *strSource* overwrites the terminating null character of *strDestination*. The behavior of **strcat** is undefined if the source and destination strings overlap.

Security Note
Because **strcat** does not check for sufficient space in *strDestination* before appending *strSource*, it is a potential cause of buff er overruns. Consider using strncat instead.

wcscat and **_mbscat** are wide-character and multibyte-character versions of **strcat**. The arguments and return value of **wcscat** are wide-character strings; those of **_mbscat** are multibyte-character strings. These three functions behave identically otherwise.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcscat	**strcat**	**_mbscat**	**wcscat**

Requirements

Routine	Required header	Compatibility
strcat	<string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcscat	<string.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbscat	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for strcpy.

.NET Framework Equivalent

System::String::Concat

See Also

Reference

String Manipulation (CRT)
strncat, _strncat_l, wcsncat, wcsncat_l, _mbsncat _mbsncat_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
strspn, wcsspn, _mbsspn, _mbsspn_l

strcat_s, wcscat_s, _mbscat_s

Append a string. These are versions of strcat, wcscat, _mbscat with security enhancements as described in Security Enhancements in the CRT.

```
errno_t strcat_s(
   char *strDestination,
   size_t numberOfElements,
   const char *strSource
);
errno_t wcscat_s(
   wchar_t *strDestination,
   size_t numberOfElements,
   const wchar_t *strSource
);
errno_t _mbscat_s(
   unsigned char *strDestination,
   size_t numberOfElements,
   const unsigned char *strSource
);
template <size_t size>
errno_t strcat_s(
   char (&strDestination)[size],
   const char *strSource
); // C++ only
template <size_t size>
errno_t wcscat_s(
   wchar_t (&strDestination)[size],
   const wchar_t *strSource
); // C++ only
template <size_t size>
errno_t _mbscat_s(
   unsigned char (&strDestination)[size],
   const unsigned char *strSource
); // C++ only
```

Parameters

strDestination

Null-terminated destination string buffer.

numberOfElements

Size of the destination string buffer.

strSource

Null-terminated source string buffer.

Return Value

Zero if successful; an error code on failure.

Error Conditions

strDestination	***numberOfElements***	***strSource***	**Return value**	**Contents of *strDestination***
NULL or unterminated	any	any	**EINVAL**	not modified
any	any	**NULL**	**EINVAL**	not modified
any	0, or too small	any	**ERANGE**	not modified

Remarks

The **strcat_s** function appends *strSource* to *strDestination* and terminates the resulting string with a null character. The initial

character of *strSource* overwrites the terminating null character of *strDestination*. The behavior of **strcat_s** is undefined if the source and destination strings overlap.

Note that the second parameter is the total size of the buffer, not the remaining size:

```
char buf[16];
strcpy_s(buf, 16, "Start");
strcat_s(buf, 16, " End");                // Correct
strcat_s(buf, 16 - strlen(buf), " End"); // Incorrect
```

wcscat_s and **_mbscat_s** are wide-character and multibyte-character versions of **strcat_s**. The arguments and return value of **wcscat_s** are wide-character strings; those of **_mbscat_s** are multibyte-character strings. These three functions behave identically otherwise.

If *strDestination* is a null pointer, or is not null-terminated, or if *strSource* is a **NULL** pointer, or if the destination string is too small, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, these functions return **EINVAL** and set **errno** to **EINVAL**.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcscat_s	**strcat_s**	**_mbscat_s**	**wcscat_s**

Requirements

Routine	Required header	Compatibility
strcat_s	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcscat_s	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbscat_s	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the code example in strcpy_s, wcscpy_s, _mbscpy_s.

.NET Framework Equivalent

System::String::Concat

See Also

Reference

String Manipulation (CRT)
strncat, _strncat_l, wcsncat, wcsncat_l, _mbsncat _mbsncat_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
strspn, wcsspn, _mbsspn, _mbsspn_l

strchr, wcschr, _mbschr, _mbschr_l

Find a character in a string, using the current locale or a specified LC_CTYPE conversion state category.

```
char *strchr(
   const char *str,
   int c
);  // C only
char *strchr(
   char * str,
   int c
); // C++ only
const char *strchr(
   const char * str,
   int c
); // C++ only
wchar_t *wcschr(
   const wchar_t *str,
   wchar_t c
); // C only
wchar_t *wcschr(
   wchar_t *str,
   wchar_t c
);  // C++ only
const wchar_t *wcschr(
   const wchar_t *str,
   wchar_t c
);  // C++ only
unsigned char *_mbschr(
   const unsigned char *str,
   unsigned int c
); // C only
unsigned char *_mbschr(
   unsigned char *str,
   unsigned int c
); // C++ only
const unsigned char *_mbschr(
   const unsigned char *str,
   unsigned int c
); // C++ only
unsigned char *_mbschr_l(
   const unsigned char *str,
   unsigned int c,
   _locale_t locale
); // C only
unsigned char *_mbschr_l(
   unsigned char *str,
   unsigned int c,
   _locale_t locale
); // C++ only
const unsigned char *_mbschr_l(
   const unsigned char *str,
   unsigned int c,
   _locale_t locale
); // C++ only
```

Parameters

str

Null-terminated source string.

c

Character to be located.

locale

Locale to use.

Return Value

Each of these functions returns a pointer to the first occurrence of *c* in *str*, or **NULL** if *c* is not found.

Remarks

The **strchr** function finds the first occurrence of *c* in *str*, or it returns **NULL** if *c* is not found. The null terminating character is included in the search.

wcschr, **_mbschr** and **_mbschr_l** are wide-character and multibyte-character versions of **strchr**. The arguments and return value of **wcschr** are wide-character strings; those of **_mbschr** are multibyte-character strings. **_mbschr** recognizes multibyte-character sequences. Also, if the string is a null pointer, **_mbschr** invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, **_mbschr** returns **NULL** and sets **errno** to **EINVAL**. **strchr** and **wcschr** do not validate their parameters. These three functions behave identically otherwise.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; for more information, see setlocale. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

In C, these functions take a **const** pointer for the first argument. In C++, two overloads are available. The overload taking a pointer to **const** returns a pointer to **const**; the version that takes a pointer to non-**const** returns a pointer to non-**const**. The macro _CONST_CORRECT_OVERLOADS is defined if both the **const** and non-**const** versions of these functions are available. If you require the non-**const** behavior for both C++ overloads, define the symbol _CONST_RETURN.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcschr	**strchr**	**_mbschr**	**wcschr**
_n/a	**n/a**	**_mbschr_l**	**n/a**

Requirements

Routine	Required header	Compatibility
strchr	<string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcschr	<string.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbschr, **_mbschr_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more information about compatibility, see Compatibility.

Example

```
// crt_strchr.c
//
// This program illustrates searching for a character
// with strchr (search forward) or strrchr (search backward).
//

#include <string.h>
#include <stdio.h>

int  ch = 'r';

char string[] = "The quick brown dog jumps over the lazy fox";
char fmt1[] =   "         1         2         3         4         5";
```

```
char fmt2[] =   "12345678901234567890123456789012345678901234567890";

int main( void )
{
   char *pdest;
   int result;

   printf_s( "String to be searched:\n      %s\n", string );
   printf_s( "      %s\n      %s\n\n", fmt1, fmt2 );
   printf_s( "Search char:   %c\n", ch );

   // Search forward.
   pdest = strchr( string, ch );
   result = (int)(pdest - string + 1);
   if ( pdest != NULL )
      printf_s( "Result:   first %c found at position %d\n",
               ch, result );
   else
      printf_s( "Result:   %c not found\n" );

   // Search backward.
   pdest = strrchr( string, ch );
   result = (int)(pdest - string + 1);
   if ( pdest != NULL )
      printf_s( "Result:   last %c found at position %d\n", ch, result );
   else
      printf_s( "Result:\t%c not found\n", ch );
}
```

Output

```
String to be searched:
      The quick brown dog jumps over the lazy fox
                1         2         3         4         5
      12345678901234567890123456789012345678901234567890

Search char:   r
Result:   first r found at position 12
Result:   last r found at position 30
```

.NET Framework Equivalent

System::String::IndexOf

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
strcspn, wcscspn, _mbscspn, _mbscspn_l
strncat, _strncat_l, wcsncat, wcsncat_l, _mbsncat _mbsncat_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strpbrk, wcspbrk, _mbspbrk, _mbspbrk_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
strstr, wcsstr, _mbsstr, _mbsstr_l

strcmp, wcscmp, _mbscmp

Compare strings.

```
int strcmp(
   const char *string1,
   const char *string2
);
int wcscmp(
   const wchar_t *string1,
   const wchar_t *string2
);
int _mbscmp(
   const unsigned char *string1,
   const unsigned char *string2
);
```

Parameters

string1, *string2*

Null-terminated strings to compare.

Return Value

The return value for each of these functions indicates the lexicographic relation of *string1* to *string2*.

Value	Relationship of string1 to string2
< 0	*string1* less than *string2*
0	*string1* identical to *string2*
> 0	*string1* greater than *string2*

On an error, **_mbscmp** returns **_NLSCMPERROR**, which is defined in STRING.H and MBSTRING.H.

Remarks

The **strcmp** function compares *string1* and *string2* lexicographically and returns a value indicating their relationship. **wcscmp** and **_mbscmp** are, respectively, wide-character and multibyte-character versions of **strcmp**. **_mbscmp** recognizes multibyte-character sequences according to the current multibyte code page and returns **_NLSCMPERROR** on an error. (For more information, see Code Pages.) Also, if *string1* or *string2* is a null pointer, **_mbscmp** invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, **_mbscmp** returns **_NLSCMPERROR** and sets **errno** to **EINVAL**. **strcmp** and **wcscmp** do not validate their parameters. These three functions behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcscmp	**strcmp**	**_mbscmp**	**wcscmp**

The **strcmp** functions differ from the **strcoll** functions in that **strcmp** comparisons are not affected by locale, whereas the manner of **strcoll** comparisons is determined by the **LC_COLLATE** category of the current locale. For more information on the **LC_COLLATE** category, see setlocale.

In the "C" locale, the order of characters in the character set (ASCII character set) is the same as the lexicographic character order. However, in other locales, the order of characters in the character set may differ from the lexicographic order. For example, in certain European locales, the character 'a' (value 0x61) precedes the character 'ä' (value 0xE4) in the character set, but the character 'ä' precedes the character 'a' lexicographically.

In locales for which the character set and the lexicographic character order differ, use **strcoll** rather than **strcmp** for lexicographic comparison of strings according to the **LC_COLLATE** category setting of the current locale. Thus, to perform a lexicographic comparison of the locale in the above example, use **strcoll** rather than **strcmp**. Alternatively, you can use **strxfrm** on the original strings, then use **strcmp** on the resulting strings.

_stricmp, **_wcsicmp**, and **_mbsicmp** compare strings by first converting them to their lowercase forms. Two strings containing characters located between 'Z' and 'a' in the ASCII table ('[', '\', ']', '^', '_', and '`') compare differently, depending on their case. For example, the two strings `"ABCDE"` and `"ABCD^"` compare one way if the comparison is lowercase (`"abcde" > "abcd^"`) and the other way (`"ABCDE" < "ABCD^"`) if the comparison is uppercase.

Requirements

Routine	Required header	Compatibility
strcmp	<string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcscmp	<string.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbscmp	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_strcmp.c

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

char string1[] = "The quick brown dog jumps over the lazy fox";
char string2[] = "The QUICK brown dog jumps over the lazy fox";

int main( void )
{
   char tmp[20];
   int result;

   // Case sensitive
   printf( "Compare strings:\n   %s\n   %s\n\n", string1, string2 );
   result = strcmp( string1, string2 );
   if( result > 0 )
      strcpy_s( tmp, _countof(tmp), "greater than" );
   else if( result < 0 )
      strcpy_s( tmp, _countof (tmp), "less than" );
   else
      strcpy_s( tmp, _countof (tmp), "equal to" );
   printf( "   strcmp:   String 1 is %s string 2\n", tmp );

   // Case insensitive (could use equivalent _stricmp)
   result = _stricmp( string1, string2 );
   if( result > 0 )
      strcpy_s( tmp, _countof (tmp), "greater than" );
   else if( result < 0 )
      strcpy_s( tmp, _countof (tmp), "less than" );
   else
      strcpy_s( tmp, _countof (tmp), "equal to" );
   printf( "   _stricmp:  String 1 is %s string 2\n", tmp );
}
```

Output

```
Compare strings:
   The quick brown dog jumps over the lazy fox
```

```
The QUICK brown dog jumps over the lazy fox

strcmp:   String 1 is greater than string 2
_stricmp:  String 1 is equal to string 2
```

.NET Framework Equivalent

System::String::CompareOrdinal

See Also

Reference

String Manipulation (CRT)
memcmp, wmemcmp
_memicmp, _memicmp_l
strcoll Functions
_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
strspn, wcsspn, _mbsspn, _mbsspn_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

strcmpi

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _stricmp instead.

strcoll Functions

Each of the **strcoll** and **wcscoll** functions compares two strings according to the **LC_COLLATE** category setting of the locale code page currently in use. Each of the **_mbscoll** functions compares two strings according to the multibyte code page currently in use. Use the **coll** functions for string comparisons when there is a difference between the character set order and the lexicographic character order in the current code page and this difference is of interest for the comparison. Use the corresponding **cmp** functions to test only for string equality.

strcoll Functions

SBCS	Unicode	MBCS	Description
strcoll	wcscoll	_mbscoll	Collate two strings
_stricoll	_wcsicoll	_mbsicoll	Collate two strings (case insensitive)
_strncoll	_wcsncoll	_mbsncoll	Collate first *count* characters of two strings
_strnicoll	_wcsnicoll	_mbsnicoll	Collate first *count* characters of two strings (case-insensitive)

Remarks

The single-byte character (SBCS) versions of these functions (**strcoll**, **stricoll**, **_strncoll**, and **_strnicoll**) compare *string1* and *string2* according to the **LC_COLLATE** category setting of the current locale. These functions differ from the corresponding **strcmp** functions in that the **strcoll** functions use locale code page information that provides collating sequences. For string comparisons in locales in which the character set order and the lexicographic character order differ, the **strcoll** functions should be used rather than the corresponding **strcmp** functions. For more information on **LC_COLLATE**, see setlocale.

For some code pages and corresponding character sets, the order of characters in the character set may differ from the lexicographic character order. In the "C" locale, this is not the case: the order of characters in the ASCII character set is the same as the lexicographic order of the characters. However, in certain European code pages, for example, the character 'a' (value 0x61) precedes the character 'ä' (value 0xE4) in the character set, but the character 'ä' precedes the character 'a' lexicographically. To perform a lexicographic comparison in such an instance, use **strcoll** rather than **strcmp**. Alternatively, you can use **strxfrm** on the original strings, then use **strcmp** on the resulting strings.

strcoll, **stricoll**, **_strncoll**, and **_strnicoll** automatically handle multibyte-character strings according to the locale code page currently in use, as do their wide-character (Unicode) counterparts. The multibyte-character (MBCS) versions of these functions, however, collate strings on a character basis according to the multibyte code page currently in use.

Because the **coll** functions collate strings lexicographically for comparison, whereas the **cmp** functions simply test for string equality, the **coll** functions are much slower than the corresponding **cmp** versions. Therefore, the **coll** functions should be used only when there is a difference between the character set order and the lexicographic character order in the current code page and this difference is of interest for the string comparison.

See Also

Reference

Locale
String Manipulation (CRT)
localeconv
_mbsnbcoll, _mbsnbcoll_l, _mbsnbicoll, _mbsnbicoll_l
setlocale, _wsetlocale
strcmp, wcscmp, _mbscmp
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

strcoll, wcscoll, _mbscoll, _strcoll_l, _wcscoll_l, _mbscoll_l

Compare strings using the current locale or a specified LC_CTYPE conversion state category.

```
int strcoll(
   const char *string1,
   const char *string2
);
int wcscoll(
   const wchar_t *string1,
   const wchar_t *string2
);
int _mbscoll(
   const unsigned char *string1,
   const unsigned char *string2
);
int _strcoll_l(
   const char *string1,
   const char *string2,
   _locale_t locale
);
int wcscoll_l(
   const wchar_t *string1,
   const wchar_t *string2,
   _locale_t locale
);
int _mbscoll_l(
   const unsigned char *string1,
   const unsigned char *string2,
   _locale_t locale
);
```

Parameters

string1, *string2*

Null-terminated strings to compare.

locale

Locale to use.

Return Value

Each of these functions returns a value indicating the relationship of *string1* to *string2*, as follows.

Return value	Relationship of string1 to string2
< 0	*string1* less than *string2*
0	*string1* identical to *string2*
> 0	*string1* greater than *string2*

Each of these functions returns **_NLSCMPERROR** on an error. To use **_NLSCMPERROR**, include either STRING.H or MBSTRING.H. **wcscoll** can fail if either *string1* or *string2* is NULL or contains wide-character codes outside the domain of the collating sequence. When an error occurs, **wcscoll** may set **errno** to **EINVAL**. To check for an error on a call to **wcscoll**, set **errno** to 0 and then check **errno** after calling **wcscoll**.

Remarks

Each of these functions performs a case-sensitive comparison of *string1* and *string2* according to the code page currently in use. These functions should be used only when there is a difference between the character set order and the lexicographic character order in the current code page and this difference is of interest for the string comparison.

All of these functions validate their parameters. If either *string1* or *string2* is a null pointer, or if *count* is greater than **INT_MAX**, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, these functions return **_NLSCMPERROR** and set **errno** to **EINVAL**.

The comparison of the two strings is a locale-dependent operation since each locale has different rules for ordering characters. The versions of these functions without the **_l** suffix use the current thread's locale for this locale-dependent behavior; the versions with the **_l** suffix are identical to the corresponding function without the suffix except that they use the locale passed in as a parameter instead of the current locale.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcscoll	**strcoll**	**_mbscoll**	**wcscoll**

Requirements

Routine	Required header	Compatibility
strcoll	<string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcscoll	<wchar.h>, <string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbscoll, **_mbscoll_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_strcoll_l	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcscoll_l	<wchar.h>, <string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::String::Compare

See Also

Reference

Locale
String Manipulation (CRT)
strcoll Functions
localeconv
_mbsnbcoll, _mbsnbcoll_l, _mbsnbicoll, _mbsnbicoll_l
setlocale, _wsetlocale
strcmp, wcscmp, _mbscmp
_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

strcpy, wcscpy, _mbscpy

Copy a string. These functions are deprecated because more secure versions are available; see strcpy_s, wcscpy_s, _mbscpy_s.

```
char *strcpy(
   char *strDestination,
   const char *strSource
);
wchar_t *wcscpy(
   wchar_t *strDestination,
   const wchar_t *strSource
);
unsigned char *_mbscpy(
   unsigned char *strDestination,
   const unsigned char *strSource
);
template <size_t size>
char *strcpy(
   char (&strDestination)[size],
   const char *strSource
); // C++ only
template <size_t size>
wchar_t *wcscpy(
   wchar_t (&strDestination)[size],
   const wchar_t *strSource
); // C++ only
template <size_t size>
unsigned char *_mbscpy(
   unsigned char (&strDestination)[size],
   const unsigned char *strSource
); // C++ only
```

Parameters

strDestination

Destination string.

strSource

Null-terminated source string.

Return Value

Each of these functions returns the destination string. No return value is reserved to indicate an error.

Remarks

The **strcpy** function copies *strSource*, including the terminating null character, to the location specified by *strDestination*. The behavior of **strcpy** is undefined if the source and destination strings overlap.

Security Note
Because **strcpy** does not check for sufficient space in *strDestination* before copying *strSource*, it is a potential cause of buffer overruns. Consider using strcpy_s instead.

wcscpy and **_mbscpy** are wide-character and multibyte-character versions of **strcpy**. The arguments and return value of **wcscpy** are wide-character strings; those of **_mbscpy** are multibyte-character strings. These three functions behave identically otherwise.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined

_tcscpy	strcpy	_mbscpy	wcscpy

Requirements

Routine	Required header	Compatibility
strcpy	<string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcscpy	<string.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbscpy	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strcpy.c
// compile with: /W3
// This program uses strcpy
// and strcat to build a phrase.

#include <string.h>
#include <stdio.h>

int main( void )
{
   char string[80];

   // Note that if you change the previous line to
   //   char string[20];
   // strcpy and strcat will happily overrun the string
   // buffer.  See the examples for strncpy and strncat
   // for safer string handling.

   strcpy( string, "Hello world from " ); // C4996
   // Note: strcpy is deprecated; consider using strcpy_s instead
   strcat( string, "strcpy " );           // C4996
   // Note: strcat is deprecated; consider using strcat_s instead
   strcat( string, "and " );              // C4996
   strcat( string, "strcat!" );           // C4996
   printf( "String = %s\n", string );
}
```

Output

```
String = Hello world from strcpy and strcat!
```

.NET Framework Equivalent

System::String::Copy

See Also

Reference

String Manipulation (CRT)
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strncat, _strncat_l, wcsncat, wcsncat_l, _mbsncat _mbsncat_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l

strspn, wcsspn, _mbsspn, _mbsspn_l

strcpy_s, wcscpy_s, _mbscpy_s

Copy a string. These are versions of strcpy, wcscpy, _mbscpy with security enhancements as described in Security Enhancements in the CRT.

```
errno_t strcpy_s(
   char *strDestination,
   size_t numberOfElements,
   const char *strSource
);
errno_t wcscpy_s(
   wchar_t *strDestination,
   size_t numberOfElements,
   const wchar_t *strSource
);
errno_t _mbscpy_s(
   unsigned char *strDestination,
   size_t numberOfElements,
   const unsigned char *strSource
);
template <size_t size>
errno_t strcpy_s(
   char (&strDestination)[size],
   const char *strSource
); // C++ only
template <size_t size>
errno_t wcscpy_s(
   wchar_t (&strDestination)[size],
   const wchar_t *strSource
); // C++ only
template <size_t size>
errno_t _mbscpy_s(
   unsigned char (&strDestination)[size],
   const unsigned char *strSource
); // C++ only
```

Parameters

strDestination

Location of destination string buffer

numberOfElements

Size of the destination string buffer.

strSource

Null-terminated source string buffer.

Return Value

Zero if successful; an error otherwise.

Error Conditions

strDestination	***numberOfElements***	***strSource***	**Return value**	**Contents of *strDestination***
NULL	any	any	**EINVAL**	not modified
any	any	**NULL**	**EINVAL**	not modified
any	0, or too small	any	**ERANGE**	not modified

Remarks

The **strcpy_s** function copies the contents in the address of *strSource*, including the terminating null character, to the location

specified by *strDestination*. The destination string must be large enough to hold the source string, including the terminating null character. The behavior of **strcpy_s** is undefined if the source and destination strings overlap.

wcscpy_s and **_mbscpy_s** are wide-character and multibyte-character versions of **strcpy_s** respectively. The arguments and return value of **wcscpy_s** are wide character strings; those of **_mbscpy_s** are multibyte character strings. These three functions behave identically otherwise.

If *strDestination* or *strSource* is a null pointer, or if the destination string is too small, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, these functions return **EINVAL** and set **errno** to **EINVAL**.

Upon successful execution, the destination string will always be null terminated.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcscpy_s	**strcpy_s**	**_mbscpy_s**	**wcscpy_s**

Requirements

Routine	Required header	Compatibility
strcpy_s	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcscpy_s	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbscpy_s	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strcpy_s.cpp
// This program uses strcpy_s and strcat_s
// and strcat to build a phrase.
//

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int main( void )
{
   char string[80];
   // using template versions of strcpy_s and strcat_s:
   strcpy_s( string, "Hello world from " );
   strcat_s( string, "strcpy_s " );
   strcat_s( string, "and " );
   // of course we can supply the size explicitly if we want to:
   strcat_s( string, _countof(string), "strcat_s!" );

   printf( "String = %s\n", string );
}
```

Output

```
String = Hello world from strcpy_s and strcat_s!
```

.NET Framework Equivalent

System::String::Copy

See Also

Reference

String Manipulation (CRT)
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
strspn, wcsspn, _mbsspn, _mbsspn_l

strcspn, wcscspn, _mbscspn, _mbscspn_l

Returns the index of the first occurrence of a character in a string that belongs to a set of characters.

```
size_t strcspn(
   const char *str,
   const char *strCharSet
);
size_t wcscspn(
   const wchar_t *str,
   const wchar_t *strCharSet
);
size_t _mbscspn(
   const unsigned char *str,
   const unsigned char *strCharSet
);
size_t _mbscspn(
   const unsigned char *str,
   const unsigned char *strCharSet,
   _locale_t locale
);
```

Parameters

str

Null-terminated searched string.

strCharSet

Null-terminated character set.

locale

Locale to use.

Return Value

These functions return the index of the first character in *str* that is in *strCharSet*. If none of the characters in *str* is in *strCharSet*, then the return value is the length of *str*.

No return value is reserved to indicate an error.

Remarks

wcscspn and **_mbscspn** are wide-character and multibyte-character versions of **strcspn**. The arguments of **wcscspn** are wide-character strings; those of **_mbscspn** are multibyte-character strings.

_mbscspn validates its parameters. If either *str* or *strCharSet* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns 0 and sets **errno** to **EINVAL**. **strcspn** and **wcscspn** do not validate their parameters. These three functions behave identically otherwise.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcscspn	**strcspn**	**_mbscspn**	**wcscspn**
n/a	**n/a**	**_mbscspn_l**	**n/a**

Requirements

Routine	Required header	Compatibility

strcspn	<string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcscspn	<string.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbscspn, **_mbscspn_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strcspn.c

#include <string.h>
#include <stdio.h>

void test( const char * str, const char * strCharSet )
{
   int pos = strcspn( str, strCharSet );
   printf( "strcspn( \"%s\", \"%s\" ) = %d\n", str, strCharSet, pos );
}

int main( void )
{
   test( "xyzbxz", "abc" );
   test( "xyzbxz", "xyz" );
   test( "xyzbxz", "no match" );
   test( "xyzbxz", "" );
   test( "", "abc" );
   test( "", "" );
}
```

Output

```
strcspn( "xyzbxz", "abc" ) = 3
strcspn( "xyzbxz", "xyz" ) = 0
strcspn( "xyzbxz", "no match" ) = 6
strcspn( "xyzbxz", "" ) = 6
strcspn( "", "abc" ) = 0
strcspn( "", "" ) = 0
```

.NET Framework Equivalent

System::String::Substring

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
strncat, _strncat_l, wcsncat, wcsncat_l, _mbsncat _mbsncat_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
strspn, wcsspn, _mbsspn, _mbsspn_l

_strdate, _wstrdate

Copy current system date to a buffer. These functions are deprecated because more secure versions are available; see _strdate_s, _wstrdate_s.

```
char *_strdate(
   char *datestr
);
wchar_t *_wstrdate(
   wchar_t *datestr
);
template <size_t size>
char *_strdate(
   char (&datestr)[size]
); // C++ only
template <size_t size>
wchar_t *_wstrdate(
   wchar_t (&datestr)[size]
); // C++ only
```

Parameters

datestr

A pointer to a buffer containing the formatted date string.

Return Value

Each of these functions returns a pointer to the resulting character string *datestr*.

Remarks

More secure versions of these functions are available; see _strdate_s, _wstrdate_s. It is recommended that the more secure functions be used wherever possible.

The **_strdate** function copies the current system date to the buffer pointed to by *datestr*, formatted *mm/dd/yy*, where *mm* is two digits representing the month, *dd* is two digits representing the day, and *yy* is the last two digits of the year. For example, the string `12/05/99` represents December 5, 1999. The buffer must be at least 9 bytes long.

If *datestr* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

_wstrdate is a wide-character version of **_strdate**; the argument and return value of **_wstrdate** are wide-character strings. These functions behave identically otherwise.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tstrdate	**_strdate**	**_strdate**	**_wstrdate**

Requirements

Routine	Required header	Compatibility
_strdate	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wstrdate	<time.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strdate.c
// compile with: /W3
#include <time.h>
#include <stdio.h>
int main()
{
    char tmpbuf[9];

    // Set time zone from TZ environment variable. If TZ is not set,
    // the operating system is queried to obtain the default value
    // for the variable.
    //
    _tzset();

    printf( "OS date: %s\n", _strdate(tmpbuf) ); // C4996
    // Note: _strdate is deprecated; consider using _strdate instead
}
```

Sample Output

```
OS date: 04/25/03
```

.NET Framework Equivalent

System::DateTime::Parse

See Also

Reference

Time Management
asctime, _wasctime
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
mktime, _mktime32, _mktime64
time, _time32, _time64
_tzset

_strdate_s, _wstrdate_s

Copy the current system date to a buffer. These are versions of _strdate, _wstrdate with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _strdate_s(
   char *buffer,
   size_t numberOfElements
);
errno_t _wstrdate_s(
   wchar_t *buffer,
   size_t numberOfElements
);
template <size_t size>
errno_t _strdate_s(
   char (&buffer)[size]
); // C++ only
template <size_t size>
errno_t _wstrdate_s(
   wchar_t (&buffer)[size]
); // C++ only
```

Parameters

[out] *buffer*

A pointer to a buffer which will be filled in with the formatted date string.

[in] *numberOfElements*

Size of the buffer.

Return Value

Zero if successful. The return value is an error code if there is a failure. Error codes are defined in ERRNO.H; see table below for the exact errors generated by this function. For more information on error codes, see errno.

Error Conditions

buffer	***numberOfElements***	**Return**	**Contents of *buffer***
NULL	(any)	**EINVAL**	Not modified
Not **NULL** (pointing to valid buffer)	0	**EINVAL**	Not modified
Not **NULL** (pointing to valid buffer)	0 < size < 9	**EINVAL**	Empty string
Not **NULL** (pointing to valid buffer)	Size > 9	0	Current date formatted as specified in the remarks

Security Issues

Passing in an invalid non **NULL** value for the buffer will result in an access violation if the size parameter is greater than 9.

Passing values for size that is greater than the actual size of the *buffer* will result in buffer overrun.

Remarks

These functions provide more secure versions of **_strdate** and **_wstrdate**. The **_strdate_s** function copies the current system date to the buffer pointed to by *buffer*, formatted *mm/dd/yy*, where *mm* is two digits representing the month, *dd* is two digits representing the day, and *yy* is the last two digits of the year. For example, the string `12/05/99` represents December 5, 1999. The buffer must be at least 9 bytes long.

_wstrdate_s is a wide-character version of **_strdate_s**; the argument and return value of **_wstrdate_s** are wide-character strings. These functions behave identically otherwise.

If *buffer* is a **NULL** pointer, or if *numberOfElements* is less than 9, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL** if the buffer is

NULL or if *numberOfElements* is less than or equal to 0, or set **errno** to **ERANGE** if *numberOfElements* is less than 9.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Generic-Text Routine Mapping:

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tstrdate_s	**_strdate_s**	**_strdate_s**	**_wstrdate_s**

Requirements

Routine	Required header	Compatibility
_strdate	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wstrdate	<time.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

Example

See the example for time.

.NET Framework Equivalent

System::DateTime::Parse

See Also

Reference

Time Management
asctime_s, _wasctime_s
ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s
_gmtime_s, _gmtime32_s, _gmtime64_s
localtime_s, _localtime32_s, _localtime64_s
mktime, _mktime32, _mktime64
time, _time32, _time64
_tzset

strdup, wcsdup

These POSIX functions are deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _strdup, _wcsdup, _mbsdup instead.

_strdup, _wcsdup, _mbsdup

Duplicate strings.

```
char *_strdup(
   const char *strSource
);
wchar_t *_wcsdup(
   const wchar_t *strSource
);
unsigned char *_mbsdup(
   const unsigned char *strSource
);
```

Parameters

strSource

Null-terminated source string.

Return Value

Each of these functions returns a pointer to the storage location for the copied string or **NULL** if storage cannot be allocated.

Remarks

The **_strdup** function calls malloc to allocate storage space for a copy of *strSource* and then copies *strSource* to the allocated space.

_wcsdup and **_mbsdup** are wide-character and multibyte-character versions of **_strdup**. The arguments and return value of **_wcsdup** are wide-character strings; those of **_mbsdup** are multibyte-character strings. These three functions behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsdup	**_strdup**	**_mbsdup**	**_wcsdup**

Because **_strdup** calls **malloc** to allocate storage space for the copy of *strSource*, it is good practice always to release this memory by calling the free routine on the pointer returned by the call to **_strdup**.

If **_DEBUG** and **_CRTDBG_MAP_ALLOC** are defined, **_strdup** and **_wcsdup** are replaced by calls to **_strdup_dbg** and **_wcsdup_dbg** to allow for debugging memory allocations. For more information, see _strdup_dbg, _wcsdup_dbg.

Requirements

Routine	Required header	Compatibility
_strdup	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsdup	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsdup	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strdup.c
```

```
#include <string.h>
#include <stdio.h>

int main( void )
{
   char buffer[] = "This is the buffer text";
   char *newstring;
   printf( "Original: %s\n", buffer );
   newstring = _strdup( buffer );
   printf( "Copy:     %s\n", newstring );
   free( newstring );
}
```

Output

```
Original: This is the buffer text
Copy:     This is the buffer text
```

.NET Framework Equivalent

System::String::Clone

See Also

Reference

String Manipulation (CRT)
memset, wmemset
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strncat, _strncat_l, wcsncat, wcsncat_l, _mbsncat _mbsncat_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
strspn, wcsspn, _mbsspn, _mbsspn_l

_strdup_dbg, _wcsdup_dbg

Versions of _strdup and _wcsdup that use the debug version of **malloc**.

```
char *_strdup_dbg(
   const char *strSource,
   int blockType,
   const char *filename,
   int linenumber
);
wchar_t *_wcsdup_dbg(
   const wchar_t *strSource,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

strSource

Null-terminated source string.

blockType

Requested type of memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

filename

Pointer to name of source file that requested allocation operation or NULL.

linenumber

Line number in source file where allocation operation was requested or NULL.

Return Value

Each of these functions returns a pointer to the storage location for the copied string or **NULL** if storage cannot be allocated.

Remarks

The **_strdup_dbg** and **_wcsdup_dbg** functions are identical to **_strdup** and **_wcsdup** except that, when **_DEBUG** is defined, these functions use the debug version of **malloc**, **_malloc_dbg**, to allocate memory for the duplicated string. For information on the debugging features of **_malloc_dbg**, see _malloc_dbg.

You do not need to call these functions explicitly in most cases. Instead, you can define the flag **_CRTDBG_MAP_ALLOC**. When **_CRTDBG_MAP_ALLOC** is defined, calls to **_strdup** and **_wcsdup** are remapped to **_strdup_dbg** and **_wcsdup_dbg**, respectively, with the *blockType* set to **_NORMAL_BLOCK**. Thus, you do not need to call these functions explicitly unless you want to mark the heap blocks as **_CLIENT_BLOCK**. For more information on block types, see Types of Blocks on the Debug Heap.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsdup_dbg	**_strdup_dbg**	**_mbsdup**	**_wcsdup_dbg**

Requirements

Routine	Required header	Compatibility
_strdup_dbg, **_wcsdup_dbg**	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All debug versions of the C run-time libraries.

.NET Framework Equivalent

System::String::Clone

See Also

Reference

String Manipulation (CRT)

_strdup, _wcsdup, _mbsdup

Concepts

Debug Versions of Heap Allocation Functions

strerror, _strerror, _wcserror, __wcserror

Get a system error message (**strerror**, **_wcserror**) or prints a user-supplied error message (**_strerror**, **__wcserror**). These functions are deprecated because more secure versions are available; see strerror_s, _strerror_s, _wcserror_s, __wcserror_s .

```
char *strerror(
   int errnum
);
char *_strerror(
   const char *strErrMsg
);
wchar_t * _wcserror(
   int errnum
);
wchar_t * __wcserror(
   const wchar_t *strErrMsg
);
```

Parameters

errnum

Error number.

strErrMsg

User-supplied message.

Return Value

All these functions return a pointer to the error-message string. Subsequent calls can overwrite the string.

Remarks

The **strerror** function maps *errnum* to an error-message string, returning a pointer to the string. Neither **strerror** nor **_strerror** actually prints the message: For that, you need to call an output function such as fprintf:

```
if (( _access( "datafile",2 )) == -1 )
   fprintf( stderr, _strerror(NULL) );
```

If *strErrMsg* is passed as **NULL**, **_strerror** returns a pointer to a string containing the system error message for the last library call that produced an error. The error-message string is terminated by the newline character ('\n'). If *strErrMsg* is not equal to **NULL**, then **_strerror** returns a pointer to a string containing (in order) your string message, a colon, a space, the system error message for the last library call producing an error, and a newline character. Your string message can be, at most, 94 characters long.

The actual error number for **_strerror** is stored in the variable errno. The system error messages are accessed through the variable _sys_errlist, which is an array of messages ordered by error number. **_strerror** accesses the appropriate error message by using the **errno** value as an index to the variable **_sys_errlist**. The value of the variable _sys_nerr is defined as the maximum number of elements in the **_sys_errlist** array. To produce accurate results, call **_strerror** immediately after a library routine returns with an error. Otherwise, subsequent calls to **strerror** or **_strerror** can overwrite the **errno** value.

_wcserror and **__wcserror** are wide-character versions of **strerror** and **_strerror**, respectively.

_strerror, **_wcserror**, and **__wcserror** are not part of the ANSI definition but are instead Microsoft extensions to it. Do not use them where portability is desired; for ANSI compatibility, use **strerror** instead.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcserror	**strerror**	**strerror**	**_wcserror**

Requirements

Routine	Required header	Compatibility
strerror	<string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_strerror	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcserror, **__wcserror**	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for perror.

.NET Framework Equivalent

System::Exception::Message

See Also

Reference

String Manipulation (CRT)
clearerr
ferror
perror, _wperror

strerror_s, _strerror_s, _wcserror_s, __wcserror_s

Get a system error message (**strerror_s**, **_wcserror_s**) or print a user-supplied error message (**_strerror_s**, **__wcserror_s**). These are versions of strerror, _strerror, _wcserror, __wcserror with security enhancements as described in Security Enhancements in the CRT.

```
errno_t strerror_s(
   char *buffer,
   size_t numberOfElements,
   int errnum
);
errno_t _strerror_s(
   char *buffer,
   size_t numberOfElements,
   const char *strErrMsg
);
errno_t _wcserror_s(
   wchar_t *buffer,
      size_t numberOfElements,
   int errnum
);
errno_t __wcserror_s(
   wchar_t *buffer,
   size_t numberOfElements,
   const wchar_t *strErrMsg
);
template <size_t size>
errno_t strerror_s(
   char (&buffer)[size],
   int errnum
); // C++ only
template <size_t size>
errno_t _strerror_s(
   char (&buffer)[size],
   const char *strErrMsg
); // C++ only
template <size_t size>
errno_t _wcserror_s(
   wchar_t (&buffer)[size],
   int errnum
); // C++ only
template <size_t size>
errno_t __wcserror_s(
   wchar_t (&buffer)[size],
   const wchar_t *strErrMsg
); // C++ only
```

Parameters

buffer

Buffer to hold error string.

numberOfElements

Size of buffer.

errnum

Error number.

strErrMsg

User-supplied message.

Return Value

Zero if successful, an error code on failure.

Error Condtions

buffer	*numberOfElements*	*strErrMsg*	Contents of *buffer*
NULL	any	any	n/a
any	0	any	not modified

Remarks

The **strerror_s** function maps *errnum* to an error-message string, returning a pointer to the string. **_strerror_s** doesn't take the error number; it uses the current value of **errno** to determine the appropriate message. Neither **strerror_s** nor **_strerror_s** actually prints the message: For that, you need to call an output function such as fprintf:

```
if (( _access( "datafile",2 )) == -1 )
{
   _strerror_s(buffer, 80);
   fprintf( stderr, buffer );
}
```

If *strErrMsg* is **NULL**, **_strerror_s** returns a pointer to a string containing the system error message for the last library call that produced an error. The error-message string is terminated by the newline character ('\n'). If *strErrMsg* is not equal to **NULL**, then **_strerror_s** returns a pointer to a string containing (in order) your string message, a colon, a space, the system error message for the last library call producing an error, and a newline character. Your string message can be, at most, 94 characters long.

These functions truncate the error message if its length exceeds *numberOfElements* -1. The resulting string in *buffer* is always null-terminated.

The actual error number for **_strerror_s** is stored in the variable errno. The system error messages are accessed through the variable _sys_errlist, which is an array of messages ordered by error number. **_strerror_s** accesses the appropriate error message by using the **errno** value as an index to the variable **_sys_errlist**. The value of the variable _sys_nerr is defined as the maximum number of elements in the **_sys_errlist** array. To produce accurate results, call **_strerror_s** immediately after a library routine returns with an error. Otherwise, subsequent calls to **strerror_s** or **_strerror_s** can overwrite the **errno** value.

_wcserror_s and **__wcserror_s** are wide-character versions of **strerror_s** and **_strerror_s**, respectively.

These functions validate their parameters. If buffer is **NULL** or if the size parameter is 0, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, the functions return **EINVAL** and set **errno** to **EINVAL**.

_strerror_s, _wcserror_s, and **__wcserror_s** are not part of the ANSI definition but are instead Microsoft extensions to it. Do not use them where portability is desired; for ANSI compatibility, use **strerror_s** instead.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically, eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcserror_s	**strerror_s**	**strerror_s**	**_wcserror_s**

Requirements

Routine	Required header	Compatibility
strerror_s, **_strerror_s**	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcserror_s, **__wcserror_s**	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for perror.

.NET Framework Equivalent

System::Exception::Message

See Also

Reference

String Manipulation (CRT)
clearerr
ferror
perror, _wperror

strftime, wcsftime, _strftime_l, _wcsftime_l

Format a time string.

```
size_t strftime(
   char *strDest,
   size_t maxsize,
   const char *format,
   const struct tm *timeptr
);
size_t _strftime_l(
   char *strDest,
   size_t maxsize,
   const char *format,
   const struct tm *timeptr,
   _locale_t locale
);
size_t wcsftime(
   wchar_t *strDest,
   size_t maxsize,
   const wchar_t *format,
   const struct tm *timeptr
);
size_t _wcsftime_l(
   wchar_t *strDest,
   size_t maxsize,
   const wchar_t *format,
   const struct tm *timeptr,
   _locale_t locale
);
```

Parameters

strDest

Output string.

maxsize

Size of *strDest* buffer.

format

Format-control string.

timeptr

tm data structure.

locale

The locale to use.

Return Value

strftime returns the number of characters placed in *strDest* and **wcsftime** returns the corresponding number of wide characters.

If the total number of characters, including the terminating null, is more than *maxsize*, both **strftime** and **wcsftime** return 0 and the contents of *strDest* are indeterminate.

The number of characters in *strDest* is equal to the number of literal characters in *format* as well as any characters that may be added to *format* via formatting codes. The terminating null of a string is not counted in the return value.

Remarks

The **strftime** and **wcsftime** functions format the **tm** time value in *timeptr* according to the supplied *format* argument and store the result in the buffer *strDest*. At most, *maxsize* characters are placed in the string. For a description of the fields in the

timeptr structure, see asctime. **wcsftime** is the wide-character equivalent of **strftime**; its string-pointer argument points to a wide-character string. These functions behave identically otherwise.

Note
In versions before Visual C++ 2005, the documentation described the *format* parameter of **wcsftime** as having the data type **const wchar_t ***, but the actual implementation of the *format* data type was **const char ***. In Visual C++ 2005, the implementation of the *format* data type has been updated to reflect the previous and current documentation, that is, **const wchar_t ***.

This function validates its parameters. If *strDest*, *format*, or *timeptr* is a null pointer, or if the **tm** data structure addressed by *timeptr* is invalid (for example, if it contains out of range values for the time or date), the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns 0 and sets **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsftime	strftime	strftime	wcsftime

The *format* argument consists of one or more codes; as in **printf**, the formatting codes are preceded by a percent sign (**%**). Characters that do not begin with **%** are copied unchanged to *strDest*. The **LC_TIME** category of the current locale affects the output formatting of **strftime**.(For more information on **LC_TIME**, see setlocale.) The functions without the **_l** suffix use the currently set locale. The versions of these functions with the **_l** suffix are identical except that they take the locale as a parameter and use that instead of the currently set locale.

The formatting codes for **strftime** are listed below:

%a

Abbreviated weekday name

%A

Full weekday name

%b

Abbreviated month name

%B

Full month name

%c

Date and time representation appropriate for locale

%d

Day of month as decimal number (01 – 31)

%H

Hour in 24-hour format (00 – 23)

%I

Hour in 12-hour format (01 – 12)

%j

Day of year as decimal number (001 – 366)

%m

Month as decimal number (01 – 12)

%M

Minute as decimal number (00 – 59)

%p

Current locale's A.M./P.M. indicator for 12-hour clock

%S

Second as decimal number (00 – 59)

%U

Week of year as decimal number, with Sunday as first day of week (00 – 53)

%w

Weekday as decimal number (0 – 6; Sunday is 0)

%W

Week of year as decimal number, with Monday as first day of week (00 – 53)

%x

Date representation for current locale

%X

Time representation for current locale

%y

Year without century, as decimal number (00 – 99)

%Y

Year with century, as decimal number

%z, %Z

Either the time-zone name or time zone abbreviation, depending on registry settings; no characters if time zone is unknown

%%

Percent sign

As in the **printf** function, the **#** flag may prefix any formatting code. In that case, the meaning of the format code is changed as follows.

Format code	Meaning
%#a, %#A, %#b, %#B, %#p, %#X, %#z, %#Z, %#%	**#** flag is ignored.
%#c	Long date and time representation, appropriate for current locale. For example: "Tuesday, March 14, 1995, 12:41:29".
%#x	Long date representation, appropriate to current locale. For example: "Tuesday, March 14, 1995".
%#d, %#H, %#I, %#j, %#m, %#M, %#S, %#U, %#w, %#W, %#y, %#Y	Remove leading zeros (if any).

Requirements

Routine	Required header	Compatibility
strftime	<time.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcsftime	<time.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_strftime_l	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsftime_l	<time.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for time.

.NET Framework Equivalent

- System::DateTime::ToLongDateString
- System::DateTime::ToLongTimeString
- System::DateTime::ToShortDateString
- System::DateTime::ToShortTimeString
- System::DateTime::ToString

See Also

Reference

Locale
Time Management
String Manipulation (CRT)
localeconv
setlocale, _wsetlocale
strcoll Functions
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

stricmp, wcsicmp

These POSIX functions are deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l instead.

_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l

Perform a lowercase comparison of strings.

```
int _stricmp(
   const char *string1,
   const char *string2
);
int _wcsicmp(
   const wchar_t *string1,
   const wchar_t *string2
);
int _mbsicmp(
   const unsigned char *string1,
   const unsigned char *string2
);
int _stricmp_l(
   const char *string1,
   const char *string2,
   _locale_t locale
);
int _wcsicmp_l(
   const wchar_t *string1,
   const wchar_t *string2,
   _locale_t locale
);
int _mbsicmp_l(
   const unsigned char *string1,
   const unsigned char *string2,
   _locale_t locale
);
```

Parameters

string1, *string2*

Null-terminated strings to compare.

locale

Locale to use.

Return Value

The return value indicates the relation of *string1* to *string2* as follows.

Return value	Description
< 0	*string1* less than *string2*
0	*string1* identical to *string2*
> 0	*string1* greater than *string2*

On an error, **_mbsicmp** returns **_NLSCMPERROR**, which is defined in STRING.H and MBSTRING.H.

Remarks

The **_stricmp** function lexicographically compares lowercase versions of *string1* and *string2* and returns a value indicating their relationship. **_stricmp** differs from **_stricoll** in that the **_stricmp** comparison is affected by **LC_CTYPE**, whereas the **_stricoll** comparison is according to the **LC_CTYPE** and **LC_COLLATE** categories of the locale. For more information on the **LC_COLLATE** category, see setlocale and Locale Categories. The versions of these functions without the **_l** suffix use the current locale for locale-dependent behavior. The versions with the suffix are identical except that they use the locale passed in instead.

The **_strcmpi** function is equivalent to **_stricmp** and is provided for backward compatibility only.

Because **stricmp** does lowercase comparisons, it may result in unexpected behavior.

To illustrate when case conversion by **stricmp** affects the outcome of a comparison, assume that you have the two strings JOHNSTON and JOHN_HENRY. The string JOHN_HENRY will be considered less than JOHNSTON because the "_" has a lower ASCII value than a lowercase S. In fact, any character that has an ASCII value between 91 and 96 will be considered less than any letter.

If the strcmp function is used instead of **stricmp**, JOHN_HENRY will be greater than JOHNSTON.

_wcsicmp and **_mbsicmp** are wide-character and multibyte-character versions of **_stricmp**. The arguments and return value of **_wcsicmp** are wide-character strings; those of **_mbsicmp** are multibyte-character strings. **_mbsicmp** recognizes multibyte-character sequences according to the current multibyte code page and returns **_NLSCMPERROR** on an error. (For more information, see Code Pages.) These three functions behave identically otherwise.

_wcsicmp and **wcscmp** behave identically except that **wcscmp** does not convert its arguments to lowercase before comparing them. **_mbsicmp** and **_mbscmp** behave identically except that **_mbscmp** does not convert its arguments to lowercase before comparing them.

You will need to call setlocale for **_wcsicmp** to work with Latin 1 characters. The C locale is in effect by default, so, for example, ä will not compare equal to Ä. Call **setlocale** with any locale other than the C locale before the call to **_wcsicmp**. The following sample demonstrates how **_wcsicmp** is sensitive to the locale:

```
// crt_stricmp_locale.c
#include <string.h>
#include <stdio.h>
#include <locale.h>

int main() {
   setlocale(LC_ALL,"C");   // in effect by default
   printf("\n%d",_wcsicmp(L"ä", L"Ä"));   // compare fails
   setlocale(LC_ALL,"");
   printf("\n%d",_wcsicmp(L"ä", L"Ä"));   // compare succeeds
}
```

An alternative is to call _create_locale and pass the returned locale object as a parameter to **_wcsicmp_l**.

All of these functions validate their parameters. If either *string1* or *string2* are **NULL** pointers, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, these functions return **_NLSCMPERROR** and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsicmp	**_stricmp**	**_mbsicmp**	**_wcsicmp**

Requirements

Routine	Required header	Compatibility
_stricmp, **_stricmp_l**	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsicmp, **_wcsicmp_l**	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsicmp, **_mbsicmp_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_stricmp.c
```

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

char string1[] = "The quick brown dog jumps over the lazy fox";
char string2[] = "The QUICK brown dog jumps over the lazy fox";

int main( void )
{
   char tmp[20];
   int result;

   // Case sensitive
   printf( "Compare strings:\n   %s\n   %s\n\n", string1, string2 );
   result = strcmp( string1, string2 );
   if( result > 0 )
      strcpy_s( tmp, _countof(tmp), "greater than" );
   else if( result < 0 )
      strcpy_s( tmp, _countof(tmp), "less than" );
   else
      strcpy_s( tmp, _countof(tmp), "equal to" );
   printf( "   strcmp:   String 1 is %s string 2\n", tmp );

   // Case insensitive (could use equivalent _stricmp)
   result = _stricmp( string1, string2 );
   if( result > 0 )
      strcpy_s( tmp, _countof(tmp), "greater than" );
   else if( result < 0 )
      strcpy_s( tmp, _countof(tmp), "less than" );
   else
      strcpy_s( tmp, _countof(tmp), "equal to" );
   printf( "   _stricmp:  String 1 is %s string 2\n", tmp );
}
```

Output

```
Compare strings:
   The quick brown dog jumps over the lazy fox
   The QUICK brown dog jumps over the lazy fox

   strcmp:   String 1 is greater than string 2
   _stricmp:  String 1 is equal to string 2
```

.NET Framework Equivalent

System::String::Compare

See Also

Reference

String Manipulation (CRT)
memcmp, wmemcmp
_memicmp, _memicmp_l
strcmp, wcscmp, _mbscmp
strcoll Functions
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l
strspn, wcsspn, _mbsspn, _mbsspn_l

_stricoll, _wcsicoll, _mbsicoll, _stricoll_l, _wcsicoll_l, _mbsicoll_l

Compare strings using locale-specific information.

```
int _stricoll(
   const char *string1,
   const char *string2
);
int _wcsicoll(
   const wchar_t *string1,
   const wchar_t *string2
);
int _mbsicoll(
   const unsigned char *string1,
   const unsigned char *string2
);
int _stricoll_l(
   const char *string1,
   const char *string2,
   _locale_t locale
);
int _wcsicoll_l(
   const wchar_t *string1,
   const wchar_t *string2,
   _locale_t locale
);
int _mbsicoll_l(
   const unsigned char *string1,
   const unsigned char *string2,
   _locale_t locale
);
```

Parameters

string1, *string2*

Null-terminated strings to compare.

locale

The locale to use.

Return Value

Each of these functions returns a value indicating the relationship of *string1* to *string2*, as follows.

Return value	**Relationship of string1 to string2**
< 0	*string1* less than *string2*
0	*string1* identical to *string2*
> 0	*string1* greater than *string2*
_NLSCMPERROR	An error occurred.

Each of these functions returns **_NLSCMPERROR**. To use **_NLSCMPERROR**, include either **STRING.H** or **MBSTRING.H**. **_wcsicoll** can fail if either *string1* or *string2* contains wide-character codes outside the domain of the collating sequence. When an error occurs, **_wcsicoll** may set **errno** to **EINVAL**. To check for an error on a call to **_wcsicoll**, set **errno** to 0 and then check **errno** after calling **_wcsicoll**.

Remarks

Each of these functions performs a case-insensitive comparison of *string1* and *string2* according to the code page currently in

use. These functions should be used only when there is a difference between the character set order and the lexicographic character order in the current code page and this difference is of interest for the string comparison.

_stricmp differs from **_stricoll** in that the **_stricmp** comparison is affected by **LC_CTYPE**, whereas the **_stricoll** comparison is according to the **LC_CTYPE** and **LC_COLLATE** categories of the locale. For more information on the **LC_COLLATE** category, see setlocale and Locale Categories. The versions of these functions without the **_l** suffix use the current locale; the versions with the **_l** suffix are identical except that they use the locale passed in instead.

All of these functions validate their parameters. If either *string1* or *string2* are **NULL** pointers, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **_NLSCMPERROR** and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsicoll	**_stricoll**	**_mbsicoll**	**_wcsicoll**

Requirements

Routine	Required header	Compatibility
_stricoll, **_stricoll_l**	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsicoll, **_wcsicoll_l**	<wchar.h>, <string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsicoll, **_mbsicoll_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::String::Compare

See Also

Reference

Locale
String Manipulation (CRT)
strcoll Functions
localeconv
_mbsnbcoll, _mbsnbcoll_l, _mbsnbicoll, _mbsnbicoll_l
setlocale, _wsetlocale
strcmp, wcscmp, _mbscmp
_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

String to Numeric Value Functions

- strtod, _strtod_l, wcstod, _wcstod_l
- strtol, wcstol, _strtol_l, _wcstol_l
- strtoul, _strtoul_l, wcstoul, _wcstoul_l
- _strtoi64, _wcstoi64, _strtoi64_l, _wcstoi64_l
- _strtoui64, _wcstoui64, _strtoui64_l, _wcstoui64_l

Remarks

Each function in the **strtod** family converts a null-terminated string to a numeric value. The available functions are listed in the following table.

Function	Description
strtod	Convert string to double-precision floating point value
strtol	Convert string to long integer
strtoul	Convert string to unsigned long integer
_strtoi64	Convert string to 64-bit **__int64** integer
_strtoui64	Convert string to unsigned 64-bit **__int64** integer

wcstod, **wcstol**, **wcstoul**, and **_wcstoi64** are wide-character versions of **strtod**, **strtol**, **strtoul**, and **_strtoi64**, respectively. The string argument to each of these wide-character functions is a wide-character string; each function behaves identically to its single-byte–character counterpart otherwise.

The **strtod** function takes two arguments: the first is the input string, and the second a pointer to the character which ends the conversion process. **strtol**, **strtoul**, **_strtoi64** and **_strtoui64** take a third argument as the number base to use in the conversion process.

The input string is a sequence of characters that can be interpreted as a numerical value of the specified type. Each function stops reading the string at the first character it cannot recognize as part of a number. This may be the terminating null character. For **strtol**, **strtoul**, **_strtoi64**, and **_strtoui64**, this terminating character can also be the first numeric character greater than or equal to the user-supplied number base.

If the user-supplied pointer to an end-of-conversion character is not set to **NULL** at call time, a pointer to the character that stopped the scan will be stored there instead. If no conversion can be performed (no valid digits were found or an invalid base was specified), the value of the string pointer is stored at that address.

strtod expects a string of the following form:

[*whitespace*] [*sign*] [*digits*] [**.***digits*] [{**d** | **D** | **e** | **E**}[*sign*]*digits*]

A *whitespace* may consist of space or tab characters, which are ignored; *sign* is either plus (**+**) or minus (**–**); and *digits* are one or more decimal digits. If no digits appear before the radix character, at least one must appear after the radix character. The decimal digits can be followed by an exponent, which consists of an introductory letter (**d**, **D**, **e**, or **E**) and an optionally signed integer. If neither an exponent part nor a radix character appears, a radix character is assumed to follow the last digit in the string. The first character that does not fit this form stops the scan.

The **strtol**, **strtoul**, **_strtoi64**, and **_strtoui64** functions expect a string of the following form:

[*whitespace*] [{**+** | **–**}] [**0** [{ **x** | **X** }]] [*digits*]

If the base argument is between 2 and 36, then it is used as the base of the number. If it is 0, the initial characters referenced to by the end-of-conversion pointer are used to determine the base. If the first character is 0 and the second character is not 'x' or 'X', the string is interpreted as an octal integer; otherwise, it is interpreted as a decimal number. If the first character is '0' and

the second character is 'x' or 'X', the string is interpreted as a hexadecimal integer. If the first character is '1' through '9', the string is interpreted as a decimal integer. The letters 'a' through 'z' (or 'A' through 'Z') are assigned the values 10 through 35; only letters whose assigned values are less than *base* are permitted. **strtoul** and **_strtoui64** allow a plus (**+**) or minus (**–**) sign prefix; a leading minus sign indicates that the return value is negated.

The output value is affected by the setting of the **LC_NUMERIC** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

When the value returned by these functions would cause an overflow or underflow, or when conversion is not possible, special case values are returned as shown:

Function	Condition	Value returned
strtod	Overflow	+/- **HUGE_VAL**
strtod	Underflow or no conversion	0
strtol	+ Overflow	**LONG_MAX**
strtol	- Overflow	**LONG_MIN**
strtol	Underflow or no conversion	0
_strtoi64	+ Overflow	**_I64_MAX**
_strtoi64	- Overflow	**_I64_MIN**
_strtoi64	No conversion	0
_strtoui64	Overflow	**_UI64_MAX**
_strtoui64	No conversion	0

_I64_MAX, **_I64_MIN**, and **_UI64_MAX** are defined in LIMITS.H.

wcstod, **wcstol**, **wcstoul**, **_wcstoi64**, and **_wcstoui64** are wide-character versions of **strtod**, **strtol**, **strtoul**, **_strtoi64**, and **_strtoui64**, respectively; the pointer to an end-of-conversion argument to each of these wide-character functions is a wide-character string. Otherwise, each of these wide-character functions behaves identically to its single-byte–character counterpart.

See Also

Reference

Data Conversion
Locale
Interpretation of Multibyte-Character Sequences
Floating-Point Support
atof, _atof_l, _wtof, _wtof_l

strlen, strlen_l, wcslen, wcslen_l, _mbslen, _mbslen_l, _mbstrlen, _mbstrlen_l

Get the length of a string, using the current locale or a specified locale. More secure versions of these functions are available; see strnlen, strnlen_l, wcsnlen, wcsnlen_l, _mbsnlen, _mbsnlen_l, _mbstrnlen, _mbstrnlen_l

```
size_t strlen(
   const char *str
);
size_t strlen_l(
   const char *str,
   _locale_t locale
);
size_t wcslen(
   const wchar_t *str
);
size_t wcslen_l(
   const wchar_t *str,
   _locale_t locale
);
size_t _mbslen(
   const unsigned char *str
);
size_t _mbslen_l(
   const unsigned char *str,
   _locale_t locale
);
size_t _mbstrlen(
   const char *str
);
size_t _mbstrlen_l(
   const char *str,
   _locale_t locale
);
```

Parameters

str

Null-terminated string.

locale

Locale to use.

Return Value

Each of these functions returns the number of characters in *str*, excluding the terminal **NULL**. No return value is reserved to indicate an error, except for **_mbstrlen**, which returns -1 if the string contains an invalid multibyte character.

Remarks

However, **strlen** interprets the string as a single-byte character string, so its return value is always equal to the number of bytes, even if the string contains multibyte characters. **wcslen** is a wide-character version of **strlen**; the argument of **wcslen** is a wide-character string and the count of characters is in wide (two-byte) characters. **wcslen** and **strlen** behave identically otherwise.

Security Note These functions incur a potential threat brought about by a buffer overrun problem. Buffer overrun problems are a frequent method of system attack, resulting in an unwarranted elevation of privilege. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined

_tcslen	**strlen**	**strlen**	**wcslen**
_tcsclen	**strlen**	**_mbslen**	**wcslen**
_tcsclen_l	**strlen_l**	**_mbslen_l**	**wcslen_l**

_mbslen, **_mbslen_l**, **_mbstrlen** and **_mbstrlen_l** return the number of multibyte characters in a multibyte-character string but they do not test for multibyte-character validity. **_mbstrlen** and **_mbstrlen_l** tests for multibyte-character validity and recognizes multibyte-character sequencessetlocale, _wsetlocale. If the string passed to **_mbstrlen** or **_mbstrlen_l** contain an invalid multibyte character for the code page, it returns -1 and sets **errno** to **EILSEQ**.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Requirements

Routine	Required header	Compatibility
strlen	<string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
strlen_l	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcslen, **wcslen_l**	<string.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbslen, **_mbslen_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbstrlen, **_mbstrlen_l**	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strlen.c
// Determine the length of a string. For the multi-byte character
// example to work correctly, the Japanese language support for
// non-Unicode programs must be enabled by the operating system.


#include <string.h>
#include <locale.h>

int main()
{
   char* str1 = "Count.";
   wchar_t* wstr1 = L"Count.";
   char * mbstr1;
   char * locale_string;

   // strlen gives the length of single-byte character string
   printf("Length of '%s' : %d\n", str1, strlen(str1) );

   // wstrlen gives the length of a wide character string
   wprintf(L"Length of '%s' : %d\n", wstr1, wcslen(wstr1) );
```

```
    // A multibyte string: [A] [B] [C] [katakana A] [D] [\0]
    // in Code Page 932. For this example to work correctly,
    // the Japanese language support must be enabled by the
    // operating system.
    mbstr1 = "ABC" "\x83\x40" "D";

    locale_string = setlocale(LC_CTYPE, "Japanese_Japan");

    if (locale_string == NULL)
    {
       printf("Japanese locale not enabled. Exiting.\n");
       exit(1);
    }
    else
    {
       printf("Locale set to %s\n", locale_string);
    }

    // _mbslen will recognize the Japanese multibyte character if the
    // current locale used by the operating system is Japanese
    printf("Length of '%s' : %d\n", mbstr1, _mbslen(mbstr1) );

    // _mbstrlen will recognize the Japanese multibyte character
    // since the CRT locale is set to Japanese even if the OS locale
    // isnot.
    printf("Length of '%s' : %d\n", mbstr1, _mbstrlen(mbstr1) );
    printf("Bytes in '%s' : %d\n", mbstr1, strlen(mbstr1) );

  }
```

Sample Output

```
Length of 'Count.' : 6
Length of 'Count.' : 6
Length of 'ABCｱD' : 5
Length of 'ABCｱD' : 5
Bytes in 'ABCｱD' : 6
```

.NET Framework Equivalent

System::String::Length

See Also

Reference

String Manipulation (CRT)
Interpretation of Multibyte-Character Sequences
Locale
setlocale, _wsetlocale
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strcoll Functions
strcpy, wcscpy, _mbscpy
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l
strspn, wcsspn, _mbsspn, _mbsspn_l

strlwr, wcslwr

These POSIX functions are deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _strlwr, _wcslwr, _mbslwr, _strlwr_l, _wcslwr_l, _mbslwr_l or security-enhanced _strlwr_s, _strlwr_s_l, _mbslwr_s, _mbslwr_s_l, _wcslwr_s, _wcslwr_s_l instead.

_strlwr, _wcslwr, _mbslwr, _strlwr_l, _wcslwr_l, _mbslwr_l

Convert a string to lowercase. These functions are deprecated because more secure versions are available; see _strlwr_s, _strlwr_s_l, _mbslwr_s, _mbslwr_s_l, _wcslwr_s, _wcslwr_s_l.

```
char *_strlwr(
   char * str
);
wchar_t *_wcslwr(
   wchar_t * str
);
unsigned char *_mbslwr(
   unsigned char * str
);
char *_strlwr_l(
   char * str,
   _locale_t locale
);
wchar_t *_wcslwr_l(
   wchar_t * str,
   _locale_t locale
);
unsigned char *_mbslwr_l(
   unsigned char * str,
   _locale_t locale
);
template <size_t size>
char *_strlwr(
   char (&str)[size]
); // C++ only
template <size_t size>
wchar_t *_wcslwr(
   wchar_t (&str)[size]
); // C++ only
template <size_t size>
unsigned char *_mbslwr(
   unsigned char (&str)[size]
); // C++ only
template <size_t size>
char *_strlwr_l(
   char (&str)[size],
   _locale_t locale
); // C++ only
template <size_t size>
wchar_t *_wcslwr_l(
   wchar_t (&str)[size],
   _locale_t locale
); // C++ only
template <size_t size>
unsigned char *_mbslwr_l(
   unsigned char (&str)[size],
   _locale_t locale
); // C++ only
```

Parameters

str

Null-terminated string to convert to lowercase.

locale

The locale to use.

Return Value

Each of these functions returns a pointer to the converted string. Because the modification is done in place, the pointer returned is the same as the pointer passed as the input argument. No return value is reserved to indicate an error.

Remarks

The **_strlwr** function converts any uppercase letters in *str* to lowercase as determined by the **LC_CTYPE** category setting of the locale. Other characters are not affected. For more information on **LC_CTYPE**, see setlocale. The versions of these functions without the **_l** suffix use the current locale for their locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale passed in instead.

The **_wcslwr** and **_mbslwr** functions are wide-character and multibyte-character versions of **_strlwr**. The argument and return value of **_wcslwr** are wide-character strings; those of **_mbslwr** are multibyte-character strings. These three functions behave identically otherwise.

If *str* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, these functions return the original string and set **errno** to **EINVAL**.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcslwr	**_strlwr**	**_mbslwr**	**_wcslwr**
_tcslwr_l	**_strlwr_l**	**_mbslwr_l**	**_wcslwr_l**

Requirements

Routine	Required header	Compatibility
_strlwr, **_strlwr_l**	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcslwr, **_wcslwr_l**	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbslwr, **_mbslwr_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strlwr.c
// compile with: /W3
// This program uses _strlwr and _strupr to create
// uppercase and lowercase copies of a mixed-case string.
#include <string.h>
#include <stdio.h>

int main( void )
{
   char string[100] = "The String to End All Strings!";
   char * copy1 = _strdup( string ); // make two copies
   char * copy2 = _strdup( string );

   _strlwr( copy1 ); // C4996
   // Note: _strlwr is deprecated; consider using _strlwr_s instead
   _strupr( copy2 ); // C4996
   // Note: _strupr is deprecated; consider using _strupr_s instead

   printf( "Mixed: %s\n", string );
   printf( "Lower: %s\n", copy1 );
   printf( "Upper: %s\n", copy2 );
```

```
    free( copy1 );
    free( copy2 );
}
```

Output

```
Mixed: The String to End All Strings!
Lower: the string to end all strings!
Upper: THE STRING TO END ALL STRINGS!
```

.NET Framework Equivalent

System::String::ToLower

See Also

Reference

String Manipulation (CRT)

Locale

_strupr, _strupr_l, _mbsupr, _mbsupr_l, _wcsupr_l, _wcsupr

_strlwr_s, _strlwr_s_l, _mbslwr_s, _mbslwr_s_l, _wcslwr_s, _wcslwr_s_l

Convert a string to lowercase, using the current locale or a locale object passed in. These are versions of _strlwr, _wcslwr, _mbslwr, _strlwr_l, _wcslwr_l, _mbslwr_l with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _strlwr_s(
   char *str,
   size_t numberOfElements
);
errno_t _strlwr_s_l(
   char *str,
   size_t numberOfElements,
   _locale_t locale
);
errno_t _mbslwr_s(
   unsigned char *str,
   size_t numberOfElements
);
errno_t _mbslwr_s_l(
   unsigned char *str,
   size_t numberOfElements,
   _locale_t locale
);
errno_t _wcslwr_s(
   wchar_t *str,
   size_t numberOfElements
);
errno_t _wcslwr_s_l(
   wchar_t *str,
   size_t numberOfElements,
   _locale_t locale
);
template <size_t size>
errno_t _strlwr_s(
   char (&str)[size]
); // C++ only
template <size_t size>
errno_t _strlwr_s_l(
   char (&str)[size],
   _locale_t locale
); // C++ only
template <size_t size>
errno_t _mbslwr_s(
   unsigned char (&str)[size]
); // C++ only
template <size_t size>
errno_t _mbslwr_s_l(
   unsigned char (&str)[size],
   _locale_t locale
); // C++ only
template <size_t size>
errno_t _wcslwr_s(
   wchar_t (&str)[size]
); // C++ only
template <size_t size>
errno_t _wcslwr_s_l(
   wchar_t (&str)[size],
   _locale_t locale
); // C++ only
```

Parameters

str

Null-terminated string to convert to lowercase.

numberOfElements

Size of the buffer.

locale

The locale to use.

Return Value

Zero if successful; a non-zero error code on failure.

These functions validate their parameters. If *str* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, the functions return **EINVAL** and set **errno** to **EINVAL**. If *numberOfElements* is less than the length of the string, the functions return **ERANGE** and set **errno** to **ERANGE**.

Remarks

The **_strlwr_s** function converts, in place, any uppercase letters in *str* to lowercase. **_mbslwr_s** is a multi-byte character version of **_strlwr_s**. **_wcslwr_s** is a wide-character version of **_strlwr_s**.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcslwr_s	**_strlwr_s**	**_mbslwr_s**	**_wcslwr_s**
_tcslwr_s_l	**_strlwr_s_l**	**_mbslwr_s_l**	**_wcslwr_s_l**

Requirements

Routine	Required header	Compatibility
_strlwr_s, **_strlwr_s_l**	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbslwr_s, **_mbslwr_s_l**	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcslwr_s, **_wcslwr_s_l**	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strlwr_s.c
// This program uses _strlwr_s and _strupr_s to create
// uppercase and lowercase copies of a mixed-case string.
//

#include <string.h>
#include <stdio.h>
```

```
int main( void )
{
   char string[100] = "The String to End All Strings!";
   char *copy1, *copy2;
   errno_t err;

   err = _strlwr_s( copy1 = _strdup(string), 100);
   err = _strupr_s( copy2 = _strdup(string), 100);

   printf( "Mixed: %s\n", string );
   printf( "Lower: %s\n", copy1 );
   printf( "Upper: %s\n", copy2 );

   free( copy1 );
   free( copy2 );
}
```

Output

```
Mixed: The String to End All Strings!
Lower: the string to end all strings!
Upper: THE STRING TO END ALL STRINGS!
```

.NET Framework Equivalent

System::String::ToLower

Smart Device Developer Notes

These methods are provided for Windows CE; however, the underlying CRT method used in CE is currently not secure. Use of these functions is still recommended so that your code can benefit from the secure versions of the methods when those become available.

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
_strupr_s, _strupr_s_l, _mbsupr_s, _mbsupr_s_l, _wcsupr_s, _wcsupr_s_l

strncat, _strncat_l, wcsncat, wcsncat_l, _mbsncat _mbsncat_l

Append characters of a string. More secure versions are available, see strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l .

```
char *strncat(
   char *strDest,
   const char *strSource,
   size_t count
);
wchar_t *wcsncat(
   wchar_t *strDest,
   const wchar_t *strSource,
   size_t count
);
unsigned char *_mbsncat(
   unsigned char *strDest,
   const unsigned char *strSource,
   size_t count
);
unsigned char *_mbsncat_l(
   unsigned char *strDest,
   const unsigned char *strSource,
   size_t count,
   _locale_t locale
);
template <size_t size>
char *strncat(
   char (&strDest)[size],
   const char *strSource,
   size_t count
); // C++ only
template <size_t size>
wchar_t *wcsncat(
   wchar_t (&strDest)[size],
   const wchar_t *strSource,
   size_t count
); // C++ only
template <size_t size>
unsigned char *_mbsncat(
   unsigned char (&strDest)[size],
   const unsigned char *strSource,
   size_t count
); // C++ only
template <size_t size>
unsigned char *_mbsncat_l(
   unsigned char (&strDest)[size],
   const unsigned char *strSource,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

strDest

Null-terminated destination string.

strSource

Null-terminated source string.

count

Number of characters to append.

locale

Locale to use.

Return Value

Returns a pointer to the destination string. No return value is reserved to indicate an error.

Remarks

The **strncat** function appends, at most, the first *count* characters of *strSource* to *strDest*. The initial character of *strSource* overwrites the terminating null character of *strDest*. If a null character appears in *strSource* before *count* characters are appended, **strncat** appends all characters from *strSource*, up to the null character. If *count* is greater than the length of *strSource*, the length of *strSource* is used in place of *count*. The all cases, the resulting string is terminated with a null character. If copying takes place between strings that overlap, the behavior is undefined.

Security Note
strncat does not check for sufficient space in *strDest*; it is therefore a potential cause of buffer overruns. Keep in mind that *count* limits the number of characters appended; it is not a limit on the size of *strDest*. See the example below. For more information, see Avoiding Buffer Overruns.

wcsncat and **_mbsncat** are wide-character and multibyte-character versions of **strncat**. The string arguments and return value of **wcsncat** are wide-character strings; those of **_mbsncat** are multibyte-character strings. These three functions behave identically otherwise.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

In C++, these functions have template overloads. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsncat	**strncat**	**_mbsnbcat**	**wcsncat**
_tcsncat_l	**_strncat_l**	**_mbsnbcat_l**	**_wcsncat_l**

Note
_strncat_l and **_wcsncat_l** have no locale dependence and are not meant to be called directly. They are provided for internal use by **_tcsncat_l**.

Requirements

Routine	Required header	Compatibility
strncat	<string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcsncat	<string.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsncat	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsncat_l	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strncat.c
// Use strcat and strncat to append to a string.
#include <stdlib.h>

#define MAXSTRINGLEN 39

char string[MAXSTRINGLEN+1];
// or char *string = malloc(MAXSTRINGLEN+1);

void BadAppend( char suffix[], int n )
{
   strncat( string, suffix, n );
}

void GoodAppend( char suffix[], size_t n )
{
   strncat( string, suffix, __min( n, MAXSTRINGLEN-strlen(string)) );
}

int main( void )
{
   string[0] = '\0';
   printf( "string can hold up to %d characters\n", MAXSTRINGLEN );

   strcpy( string, "This is the initial string!" );
   // concatenate up to 20 characters...
   BadAppend( "Extra text to add to the string...", 20 );
   printf( "After BadAppend :  %s (%d chars)\n", string, strlen(string) );

   strcpy( string, "This is the initial string!" );
   // concatenate up to 20 characters...
   GoodAppend( "Extra text to add to the string...", 20 );
   printf( "After GoodAppend:  %s (%d chars)\n", string, strlen(string) );
}
```

Output

```
string can hold up to 39 characters
After BadAppend :  This is the initial string!Extra text to add to (47 chars)
After GoodAppend:  This is the initial string!Extra text t (39 chars)
```

Note that **BadAppend** caused a buffer overrun.

.NET Framework Equivalent

System::String::Concat

See Also

Reference

String Manipulation (CRT)
_mbsnbcat, _mbsnbcat_l
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strcpy, wcscpy, _mbscpy
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l
strspn, wcsspn, _mbsspn, _mbsspn_l
Locale
Interpretation of Multibyte-Character Sequences

strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l

Append characters to a string. These are versions of strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l with security enhancements as described in Security Enhancements in the CRT.

```
errno_t strncat_s(
   char *strDest,
   size_t numberOfElements,
   const char *strSource,
   size_t count
);
errno_t _strncat_s_l(
   char *strDest,
   size_t numberOfElements,
   const char *strSource,
   size_t count,
   _locale_t locale
);
errno_t wcsncat_s(
   wchar_t *strDest,
   size_t numberOfElements,
   const wchar_t *strSource,
   size_t count
);
errno_t _wcsncat_s_l(
   wchar_t *strDest,
   size_t numberOfElements,
   const wchar_t *strSource,
   size_t count,
   _locale_t locale
);
errno_t _mbsncat_s(
   unsigned char *strDest,
   size_t numberOfElements,
   const unsigned char *strSource,
   size_t count
);
errno_t _mbsncat_s_l(
   unsigned char *strDest,
   size_t numberOfElements,
   const unsigned char *strSource,
   size_t count,
   _locale_t locale
);
template <size_t size>
errno_t strncat_s(
   char (&strDest)[size],
   const char *strSource,
   size_t count
); // C++ only
template <size_t size>
errno_t _strncat_s_l(
   char (&strDest)[size],
   const char *strSource,
   size_t count,
   _locale_t locale
); // C++ only
template <size_t size>
errno_t wcsncat_s(
   wchar_t (&strDest)[size],
   const wchar_t *strSource,
   size_t count
```

```
); // C++ only
template <size_t size>
errno_t _wcsncat_s_l(
   wchar_t (&strDest)[size],
   const wchar_t *strSource,
   size_t count,
   _locale_t locale
); // C++ only
template <size_t size>
errno_t _mbsncat_s(
   unsigned char (&strDest)[size],
   const unsigned char *strSource,
   size_t count
); // C++ only
template <size_t size>
errno_t _mbsncat_s_l(
   unsigned char (&strDest)[size],
   const unsigned char *strSource,
   size_t count,
      _locale_t locale
); // C++ only
```

Parameters

[out] *strDest*

Null-terminated destination string.

[in] *numberOfElements*

Size of the destination buffer.

[in] *strSource*

Null-terminated source string.

[in] *count*

Number of characters to append, or _TRUNCATE.

[in] *locale*

Locale to use.

Return Value

Returns 0 if successful, an error code on failure.

Error Conditions

strDestination	***numberOfElements***	***strSource***	**Return value**	**Contents of *strDestination***
NULL or unterminated	any	any	**EINVAL**	not modified
any	any	**NULL**	**EINVAL**	not modified
any	0, or too small	any	**ERANGE**	not modified

Remarks

These functions try to append the first *D* characters of *strSource* to the end of *strDest*, where *D* is the lesser of *count* and the length of *strSource*. If appending those *D* characters will fit within *strDest* (whose size is given as *numberOfElements*) and still leave room for a null terminator, then those characters are appended, starting at the original terminating null of *strDest*, and a new terminating null is appended; otherwise, *strDest*[0] is set to the null character and the invalid parameter handler is invoked, as described in Parameter Validation.

There is an exception to the above paragraph. If *count* is _TRUNCATE then as much of *strSource* as will fit is appended to *strDest* while still leaving room to append a terminating null.

For example,

```
char dst[5];
```

```
strncpy_s(dst, _countof(dst), "12", 2);
strncat_s(dst, _countof(dst), "34567", 3);
```

means that we are asking **strncat_s** to append three characters to two characters in a buffer five characters long; this would leave no space for the null terminator, hence **strncat_s** zeroes out the string and calls the invalid parameter handler.

If truncation behavior is needed, use **_TRUNCATE** or adjust the *size* parameter accordingly:

```
strncat_s(dst, _countof(dst), "34567", _TRUNCATE);
```

or

```
strncat_s(dst, _countof(dst), "34567", _countof(dst)-strlen(dst)-1);
```

In all cases, the resulting string is terminated with a null character. If copying takes place between strings that overlap, the behavior is undefined.

If *strSource* or *strDest* is **NULL**, or if *numberOfElements* is zero, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, the function returns **EINVAL** without modifying its parameters.

wcsncat_s and **_mbsncat_s** are wide-character and multibyte-character versions of **strncat_s**. The string arguments and return value of **wcsncat_s** are wide-character strings; those of **_mbsncat_s** are multibyte-character strings. These three functions behave identically otherwise.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsncat_s	**strncat_s**	**_mbsnbcat_s**	**wcsncat_s**
_tcsncat_s_l	**_strncat_s_l**	**_mbsnbcat_s_l**	**_wcsncat_s_l**

_strncat_s_l and **_wcsncat_s_l** have no locale dependence; they are only provided for **_tcsncat_s_l**.

Requirements

Routine	Required header	Compatibility
strncat_s	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcsncat_s	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsncat_s, **_mbsncat_s_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strncat_s.cpp
// compile with: /MTd

// These #defines enable secure template overloads
// (see last part of Examples() below)
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1
```

```
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES_COUNT 1

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <crtdbg.h>  // For _CrtSetReportMode
#include <errno.h>

// This example uses a 10-byte destination buffer.

errno_t strncat_s_tester( const char * initialDest,
                          const char * src,
                          int count )
{
   char dest[10];
   strcpy_s( dest, _countof(dest), initialDest );

   printf_s( "\n" );

   if ( count == _TRUNCATE )
      printf_s( "Appending '%s' to %d-byte buffer dest with truncation semantics\n",
                src, _countof(dest) );
   else
      printf_s( "Appending %d chars of '%s' to %d-byte buffer dest\n",
               count, src, _countof(dest) );

   printf_s( "    old contents of dest: '%s'\n", dest );

   errno_t err = strncat_s( dest, _countof(dest), src, count );

   printf_s( "    new contents of dest: '%s'\n", dest );

   return err;
}


void Examples()
{
   strncat_s_tester( "hi ", "there", 4 );
   strncat_s_tester( "hi ", "there", 5 );
   strncat_s_tester( "hi ", "there", 6 );

   printf_s( "\nDestination buffer too small:\n" );
   strncat_s_tester( "hello ", "there", 4 );

   printf_s( "\nTruncation examples:\n" );

   errno_t err = strncat_s_tester( "hello ", "there", _TRUNCATE );
   printf_s( "    truncation %s occur\n", err == STRUNCATE ? "did"
                                                        : "did not" );

   err = strncat_s_tester( "hello ", "!", _TRUNCATE );
   printf_s( "    truncation %s occur\n", err == STRUNCATE ? "did"
                                                        : "did not" );

   printf_s( "\nSecure template overload example:\n" );

   char dest[10] = "cats and ";
   strncat( dest, "dachshunds", 15 );
   // With secure template overloads enabled (see #define
   // at top of file), the preceding line is replaced by
   //    strncat_s( dest, _countof(dest), "dachshunds", 15 );
   // Instead of causing a buffer overrun, strncat_s invokes
   // the invalid parameter handler.
   // If secure template overloads were disabled, strncat would
   // append "dachshunds" and overrun the dest buffer.
   printf_s( "    new contents of dest: '%s'\n", dest );
}
```

```
void myInvalidParameterHandler(
   const wchar_t* expression,
   const wchar_t* function,
   const wchar_t* file,
   unsigned int line,
   uintptr_t pReserved)
{
   wprintf_s(L"Invalid parameter handler invoked: %s\n", expression);
}

int main( void )
{
   _invalid_parameter_handler oldHandler, newHandler;

   newHandler = myInvalidParameterHandler;
   oldHandler = _set_invalid_parameter_handler(newHandler);
   // Disable the message box for assertions.
   _CrtSetReportMode(_CRT_ASSERT, 0);

   Examples();
}
```

Output

```
Appending 4 chars of 'there' to 10-byte buffer dest
    old contents of dest: 'hi '
    new contents of dest: 'hi ther'

Appending 5 chars of 'there' to 10-byte buffer dest
    old contents of dest: 'hi '
    new contents of dest: 'hi there'

Appending 6 chars of 'there' to 10-byte buffer dest
    old contents of dest: 'hi '
    new contents of dest: 'hi there'

Destination buffer too small:

Appending 4 chars of 'there' to 10-byte buffer dest
    old contents of dest: 'hello '
Invalid parameter handler invoked: (L"Buffer is too small" && 0)
    new contents of dest: ''

Truncation examples:

Appending 'there' to 10-byte buffer dest with truncation semantics
    old contents of dest: 'hello '
    new contents of dest: 'hello the'
    truncation did occur

Appending '!' to 10-byte buffer dest with truncation semantics
    old contents of dest: 'hello '
    new contents of dest: 'hello !'
    truncation did not occur

Secure template overload example:
Invalid parameter handler invoked: (L"Buffer is too small" && 0)
    new contents of dest: ''
```

.NET Framework Equivalent

System::String::Concat

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
_mbsnbcat, _mbsnbcat_l

strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strcpy, wcscpy, _mbscpy
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l
strspn, wcsspn, _mbsspn, _mbsspn_l

strncmp, wcsncmp, _mbsncmp, _mbsncmp_l

Compare characters of two strings, using the current locale or a specified locale.

```
int strncmp(
   const char *string1,
   const char *string2,
   size_t count
);
int wcsncmp(
   const wchar_t *string1,
   const wchar_t *string2,
   size_t count
);
int _mbsncmp(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count
);
int _mbsncmp_l(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count,
   _locale_t locale
);
```

Parameters

string1, *string2*

Strings to compare.

count

Number of characters to compare.

locale

Locale to use.

Return Value

The return value indicates the relation of the substrings of *string1* and *string2* as follows.

Return value	Description
< 0	*string1* substring less than *string2* substring
0	*string1* substring identical to *string2* substring
> 0	*string1* substring greater than *string2* substring

On an error, **_mbsncmp** returns **_NLSCMPERROR**, which is defined in STRING.H and MBSTRING.H.

Remarks

The **strncmp** function lexicographically compares, at most, the first *count* characters in *string1* and *string2* and returns a value indicating the relationship between the substrings. **strncmp** is a case-sensitive version of **_strnicmp**. **wcsncmp** and **_mbsncmp** are case-sensitive versions of **_wcsnicmp** and **_mbsnicmp**.

wcsncmp and **_mbsncmp** are wide-character and multibyte-character versions of **strncmp**. The arguments and return value of **wcsncmp** are wide-character strings; those of **_mbsncmp** are multibyte-character strings. **_mbsncmp** recognizes multibyte-character sequences according to a multibyte code page and returns **_NLSCMPERROR** on an error.

Also, **_mbsncmp** validates its parameters. If *string1* or *string2* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, **_mbsncmp** returns **_NLSCMPERROR** and sets **errno** to

EINVAL. **strncmp** and **wcsncmp** do not validate their parameters. These three functions behave identically otherwise.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsnccmp	**strncmp**	**_mbsnbcmp**	**wcsncmp**
_tcsncmp	**strncmp**	**_mbsnbcmp**	**wcsncmp**
_tccmp	Maps to macro or inline function	**_mbsncmp**	Maps to macro or inline function
n/a	**n/a**	**_mbsncmp_l**	**n/a**

Requirements

Routine	Required header	Compatibility
strncmp	<string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcsncmp	<string.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsncmp, **_mbsncmp_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strncmp.c
#include <string.h>
#include <stdio.h>

char string1[] = "The quick brown dog jumps over the lazy fox";
char string2[] = "The QUICK brown fox jumps over the lazy dog";

int main( void )
{
   char tmp[20];
   int result;
   printf( "Compare strings:\n      %s\n      %s\n\n",
           string1, string2 );
   printf( "Function:   strncmp (first 10 characters only)\n" );
   result = strncmp( string1, string2 , 10 );
   if( result > 0 )
      strcpy_s( tmp, sizeof(tmp), "greater than" );
   else if( result < 0 )
      strcpy_s( tmp, sizeof(tmp), "less than" );
   else
      strcpy_s( tmp, sizeof(tmp), "equal to" );
   printf( "Result:      String 1 is %s string 2\n\n", tmp );
   printf( "Function:   strnicmp _strnicmp (first 10 characters only)\n" );
   result = _strnicmp( string1, string2, 10 );
   if( result > 0 )
      strcpy_s( tmp, sizeof(tmp), "greater than" );
   else if( result < 0 )
      strcpy_s( tmp, sizeof(tmp), "less than" );
```

```
    else
       strcpy_s( tmp, sizeof(tmp), "equal to" );
    printf( "Result:      String 1 is %s string 2\n", tmp );
  }
```

Output

```
Compare strings:
      The quick brown dog jumps over the lazy fox
      The QUICK brown fox jumps over the lazy dog

Function:   strncmp (first 10 characters only)
Result:      String 1 is greater than string 2

Function:   strnicmp _strnicmp (first 10 characters only)
Result:      String 1 is equal to string 2
```

.NET Framework Equivalent

System::String::Compare

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
_mbsnbcmp, _mbsnbcmp_l
_mbsnbicmp, _mbsnbicmp_l
strcmp, wcscmp, _mbscmp
strcoll Functions
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l
strspn, wcsspn, _mbsspn, _mbsspn_l

_strncoll, _wcsncoll, _mbsncoll, _strncoll_l, _wcsncoll_l, _mbsncoll_l

Compare strings using locale-specific information.

```
int _strncoll(
   const char *string1,
   const char *string2,
   size_t count
);
int _wcsncoll(
   const wchar_t *string1,
   const wchar_t *string2,
   size_t count
);
int _mbsncoll(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count
);
int _strncoll_l(
   const char *string1,
   const char *string2,
   size_t count,
   _locale_t locale
);
int _wcsncoll_l(
   const wchar_t *string1,
   const wchar_t *string2,
   size_t count,
   _locale_t locale
);
int _mbsncoll_l(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count,
   _locale_t locale
);
```

Parameters

string1, *string2*

Null-terminated strings to compare.

count

Number of characters to compare.

locale

The locale to use.

Return Value

Each of these functions returns a value indicating the relationship of the substrings of *string1* and *string2*, as follows.

Return value	**Relationship of string1 to string2**
< 0	*string1* less than *string2*
0	*string1* identical to *string2*
> 0	*string1* greater than *string2*

Each of these functions returns **_NLSCMPERROR**. To use **_NLSCMPERROR**, include either **STRING.H** or **MBSTRING.H**. **_wcsncoll** can fail if either *string1* or *string2* contains wide-character codes outside the domain of the collating sequence. When an error occurs, **_wcsncoll** may set **errno** to **EINVAL**. To check for an error on a call to **_wcsncoll**, set **errno** to 0 and then check **errno** after calling **_wcsncoll.**

Remarks

Each of these functions performs a case-sensitive comparison of the first *count* characters in *string1* and *string2* according to the code page currently in use. These functions should be used only when there is a difference between the character set order and the lexicographic character order in the code page and this difference is of interest for the string comparison. The character set order is locale-dependent. The versions of these functions without the **_l** suffix use the current locale; the versions with the **_l** suffix are identical except that they use the locale passed in instead.

All of these functions validate their parameters. If either *string1* or *string2* is a null pointer, or count is greater than **INT_MAX**, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, these functions return **_NLSCMPERROR** and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsnccoll	**_strncoll**	**_mbsncoll**	**_wcsncoll**
_tcsncoll	**_strncoll**	_mbsnbcoll	**_wcsncoll**

Requirements

Routine	Required header	Compatibility
_strncoll, **_strncoll_l**	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsncoll, **_wcsncoll_l**	<wchar.h> or <string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsncoll, **_mbsncoll_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::String::Compare

See Also

Reference

Locale
String Manipulation (CRT)
strcoll Functions
localeconv
_mbsnbcoll, _mbsnbcoll_l, _mbsnbicoll, _mbsnbicoll_l
setlocale, _wsetlocale
strcmp, wcscmp, _mbscmp
_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l

Copy characters of one string to another. These functions are deprecated because more secure versions are available; see strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l.

```
char *strncpy(
   char *strDest,
   const char *strSource,
   size_t count
);
char *_strncpy_l(
   char *strDest,
   const char *strSource,
   size_t count,
   locale_t locale
);
wchar_t *wcsncpy(
   wchar_t *strDest,
   const wchar_t *strSource,
   size_t count
);
wchar_t *_wcsncpy_l(
   wchar_t *strDest,
   const wchar_t *strSource,
   size_t count,
   locale_t locale
);
unsigned char *_mbsncpy(
   unsigned char *strDest,
   const unsigned char *strSource,
   size_t count
);
unsigned char *_mbsncpy_l(
   unsigned char *strDest,
   const unsigned char *strSource,
   size_t count,
   _locale_t locale
);
template <size_t size>
char *strncpy(
   char (&strDest)[size],
   const char *strSource,
   size_t count
); // C++ only
template <size_t size>
char *_strncpy_l(
   char (&strDest)[size],
   const char *strSource,
   size_t count,
   locale_t locale
); // C++ only
template <size_t size>
wchar_t *wcsncpy(
   wchar_t (&strDest)[size],
   const wchar_t *strSource,
   size_t count
); // C++ only
template <size_t size>
wchar_t *_wcsncpy_l(
   wchar_t (&strDest)[size],
   const wchar_t *strSource,
   size_t count,
   locale_t locale
); // C++ only
```

```
template <size_t size>
unsigned char *_mbsncpy(
   unsigned char (&strDest)[size],
   const unsigned char *strSource,
   size_t count
); // C++ only
template <size_t size>
unsigned char *_mbsncpy_l(
   unsigned char (&strDest)[size],
   const unsigned char *strSource,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

strDest

Destination string.

strSource

Source string.

count

Number of characters to be copied.

locale

Locale to use.

Return Value

Returns *strDest*. No return value is reserved to indicate an error.

Remarks

The **strncpy** function copies the initial *count* characters of *strSource* to *strDest* and returns *strDest*. If *count* is less than or equal to the length of *strSource*, a null character is not appended automatically to the copied string. If *count* is greater than the length of *strSource*, the destination string is padded with null characters up to length *count*. The behavior of **strncpy** is undefined if the source and destination strings overlap.

Security Note
strncpy does not check for sufficient space in *strDest*; it is therefore a potential cause of buffer overruns. Keep in mind that *c ount* limits the number of characters copied; it is not a limit on the size of *strDest*. See the example below. For more informati on, see Avoiding Buffer Overruns.

If *strDest* or *strSource* is a **NULL** pointer, or if *count* is less than or equal to zero, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**

wcsncpy and **_mbsncpy** are wide-character and multibyte-character versions of **strncpy**. The arguments and return value of **wcsncpy** and **_mbsncpy** vary accordingly. These six functions behave identically otherwise.

The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsncpy	strncpy	_mbsnbcpy	wcsncpy
_tcsncpy_l	_strncpy_l	_mbsnbcpy_l	_wcsncpy_l

Note

_strncpy_l and **_wcsncpy_l** have no locale dependence; they are provided just for **_tcsncpy_l** and are not intended to be called directly.

Requirements

Routine	Required header	Compatibility
strncpy	<string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcsncpy	<string.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsncpy, **_mbsncpy_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strncpy_x86.c
// compile with: /W1
// processor: x86
#include <stdio.h>
#include <string.h>

int main() {
   char a[20] = "test";
   char s[20];
   char *p = 0, *q = 0;

   strcpy_s( s, sizeof(s), "AA BB CC" );
   // Note: strncpy is deprecated; consider using strncpy_s instead
   strncpy( s, "aa", 2 );   // "aa BB CC"         C4996
   strncpy( s+3, "bb", 2 ); // "aa bb CC"         C4996
   strncpy( s, "ZZ", 3 );   // "ZZ",              C4996
                            // count greater than strSource, null added
   printf( "%s\n", s );

   strcpy_s( s, sizeof(s), "AA BB CC" );
   p = strstr(s, "BB");
   q = strstr(s, "CC");
   strncpy(s, "aa", p - s - 1);   // "aa BB CC"   C4996
   strncpy(p, "bb", q - p - 1);   // "aa bb CC"   C4996
   strncpy(q, "cc",  q - s);      // "aa bb cc"   C4996
   strncpy(q, "dd", strlen(q));   // "aa bb dd"   C4996
   printf( "%s\n", s );

   // some problems with strncpy
   strcpy_s( s, sizeof(s), "AA BB CC" );
   strncpy( s, "this is a very long string", 20 ); // C4996
   // Danger: at this point, s has no terminating null

   strcpy_s( s, sizeof(s), "dogs like cats" );
   strncpy( s+10, "to chase cars", 14);
   // strncpy has caused a buffer overrun and corrupted string a
   printf( "Buffer overrun: a = '%s' (should be 'test')\n", a );
   // In this case, a is just a char array, but if a were a function
   // pointer, this would be an exploitable buffer overrun.
}
```

Output

```
ZZ
aa bb dd
Buffer overrun: a = 'ars' (should be 'test')
```

```
// crt_strncpy_x64.c
// compile with: /W3
// processor: x64 IPF
#include <stdio.h>
#include <string.h>

int main() {
   char a[20];
   char s[20];
   char b[20] = "test";
   char *p = 0, *q = 0;

   strcpy_s( s, sizeof(s), "AA BB CC" );
   // Note: strncpy is deprecated; consider using strncpy_s instead
   strncpy( s, "aa", 2 );   // "aa BB CC"         C4996
   strncpy( s+3, "bb", 2 ); // "aa bb CC"         C4996
   strncpy( s, "ZZ", 3 );   // "ZZ",              C4996
                            // count greater than strSource, null added
   printf( "%s\n", s );

   strcpy_s( s, sizeof(s), "AA BB CC" );
   p = strstr(s, "BB");
   q = strstr(s, "CC");
   strncpy(s, "aa", p - s - 1);   // "aa BB CC"   C4996
   strncpy(p, "bb", q - p - 1);   // "aa bb CC"   C4996
   strncpy(q, "cc",  q - s);      // "aa bb cc"   C4996
   strncpy(q, "dd", strlen(q));   // "aa bb dd"   C4996
   printf( "%s\n", s );

   // some problems with strncpy
   strcpy_s( s, sizeof(s), "AA BB CC" );
   strncpy( s, "this is a very long string", 20 ); // C4996
   // Danger: at this point, s has no terminating null

   strcpy_s( s, sizeof(s), "dogs like cats" );
   strncpy( s+10, "to chase cars and buses and things", 36);
   // strncpy has caused a buffer overrun and corrupted string b
   printf( "Buffer overrun: b = '%s' (should be 'test')\n", b );
   // In this case, b is just a char array, but if b were a function
   // pointer, this would be an exploitable buffer overrun.
}
```

Output

```
ZZ
aa bb dd
Buffer overrun: b = 's and things' (should be 'test')
```

.NET Framework Equivalent

System::String::Copy

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
_mbsnbcpy, _mbsnbcpy_l
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strcpy, wcscpy, _mbscpy
strncat, _strncat_l, wcsncat, wcsncat_l, _mbsncat _mbsncat_l

strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l
strspn, wcsspn, _mbsspn, _mbsspn_l
strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l
strcpy_s, wcscpy_s, _mbscpy_s

strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l

Copy characters of one string to another. These are versions of strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l with security enhancements as described in Security Enhancements in the CRT.

```
errno_t strncpy_s(
   char *strDest,
   size_t numberOfElements,
   const char *strSource,
   size_t count
);
errno_t _strncpy_s_l(
   char *strDest,
   size_t numberOfElements,
   const char *strSource,
   size_t count,
   _locale_t locale
);
errno_t wcsncpy_s(
   wchar_t *strDest,
   size_t numberOfElements,
   const wchar_t *strSource,
   size_t count
);
errno_t _wcsncpy_s_l(
   wchar_t *strDest,
   size_t numberOfElements,
   const wchar_t *strSource,
   size_t count,
   _locale_t locale
);
errno_t _mbsncpy_s(
   unsigned char *strDest,
   size_t numberOfElements,
   const unsigned char *strSource,
   size_t count
);
errno_t _mbsncpy_s_l(
   unsigned char *strDest,
   size_t numberOfElements,
   const unsigned char *strSource,
   size_t count,
   locale_t locale
);
template <size_t size>
errno_t strncpy_s(
   char (&strDest)[size],
   const char *strSource,
   size_t count
); // C++ only
template <size_t size>
errno_t _strncpy_s_l(
   char (&strDest)[size],
   const char *strSource,
   size_t count,
   _locale_t locale
); // C++ only
template <size_t size>
errno_t wcsncpy_s(
   wchar_t (&strDest)[size],
   const wchar_t *strSource,
```

```
   size_t count
); // C++ only
template <size_t size>
errno_t _wcsncpy_s_l(
   wchar_t (&strDest)[size],
   const wchar_t *strSource,
   size_t count,
   _locale_t locale
); // C++ only
template <size_t size>
errno_t _mbsncpy_s(
   unsigned char (&strDest)[size],
   const unsigned char *strSource,
   size_t count
); // C++ only
template <size_t size>
errno_t _mbsncpy_s_l(
   unsigned char (&strDest)[size],
   const unsigned char *strSource,
   size_t count,
   locale_t locale
); // C++ only
```

Parameters

strDest

Destination string.

numberOfElements

The size of the destination string.

strSource

Source string.

count

Number of characters to be copied, or _TRUNCATE.

locale

The locale to use.

Return Value

Zero if successful, otherwise an error code.

Error Conditions

strDest	***numberOfElements***	***strSource***	**Return value**	**Contents of *strDest***
NULL	any	any	**EINVAL**	not modified
any	any	**NULL**	**EINVAL**	not modified
any	0	any	**EINVAL**	not modified
not **NULL**	too small	any	**EINVAL**	*strDest*[0] set to 0

Remarks

These functions try to copy the first *D* characters of *strSource* to *strDest*, where *D* is the lesser of *count* and the length of *strSource*. If those *D* characters will fit within *strDest* (whose size is given as *numberOfElements*) and still leave room for a null terminator, then those characters are copied and a terminating null is appended; otherwise, *strDest*[0] is set to the null character and the invalid parameter handler is invoked, as described in Parameter Validation.

There is an exception to the above paragraph. If *count* is **_TRUNCATE**, then as much of *strSource* as will fit into *strDest* is copied while still leaving room for the terminating null which is always appended.

For example,

```
char dst[5];
strncpy_s(dst, 5, "a long string", 5);
```

means that we are asking **strncpy_s** to copy five characters into a buffer five bytes long; this would leave no space for the null terminator, hence **strncpy_s** zeroes out the string and calls the invalid parameter handler.

If truncation behavior is needed, use **_TRUNCATE** or (*size* – 1):

```
strncpy_s(dst, 5, "a long string", _TRUNCATE);
strncpy_s(dst, 5, "a long string", 4);
```

Note that unlike **strncpy**, if *count* is greater than the length of *strSource*, the destination string is NOT padded with null characters up to length *count*.

The behavior of **strncpy_s** is undefined if the source and destination strings overlap.

If *strDest* or *strSource* is **NULL**, or *numberOfElements* is 0, the invalid parameter handler is invoked. If execution is allowed to continue, the function returns **EINVAL** and sets **errno** to **EINVAL**.

wcsncpy_s and **_mbsncpy_s** are wide-character and multibyte-character versions of **strncpy_s**. The arguments and return value of **wcsncpy_s** and **mbsncpy_s** do vary accordingly. These six functions behave identically otherwise.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsncpy_s	**strncpy_s**	**_mbsnbcpy_s**	**wcsncpy_s**
_tcsncpy_s_l	**_strncpy_s_l**	**_mbsnbcpy_s_l**	**_wcsncpy_s_l**

Note

_strncpy_s_l, **_wcsncpy_s_l** and **_mbsncpy_s_l** have no locale dependence and are provided just for **_tcsncpy_s_l** and are not intended to be called directly.

Requirements

Routine	Required header	Compatibility
strncpy_s, **_strncpy_s_l**	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcsncpy_s, **_wcsncpy_s_l**	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsncpy_s, **_mbsncpy_s_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strncpy_s_1.cpp
// compile with: /MTd

// these #defines enable secure template overloads
```

```
// (see last part of Examples() below)
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES_COUNT 1

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <crtdbg.h>  // For _CrtSetReportMode
#include <errno.h>

// This example uses a 10-byte destination buffer.

errno_t strncpy_s_tester( const char * src,
                          int count )
{
   char dest[10];

   printf( "\n" );

   if ( count == _TRUNCATE )
      printf( "Copying '%s' to %d-byte buffer dest with truncation semantics\n",
               src, _countof(dest) );
   else
      printf( "Copying %d chars of '%s' to %d-byte buffer dest\n",
              count, src, _countof(dest) );

   errno_t err = strncpy_s( dest, _countof(dest), src, count );

   printf( "    new contents of dest: '%s'\n", dest );

   return err;
}


void Examples()
{
   strncpy_s_tester( "howdy", 4 );
   strncpy_s_tester( "howdy", 5 );
   strncpy_s_tester( "howdy", 6 );

   printf( "\nDestination buffer too small:\n" );
   strncpy_s_tester( "Hi there!!", 10 );

   printf( "\nTruncation examples:\n" );

   errno_t err = strncpy_s_tester( "How do you do?", _TRUNCATE );
   printf( "    truncation %s occur\n", err == STRUNCATE ? "did"
                                                         : "did not" );

   err = strncpy_s_tester( "Howdy.", _TRUNCATE );
   printf( "    truncation %s occur\n", err == STRUNCATE ? "did"
                                                         : "did not" );

   printf( "\nSecure template overload example:\n" );

   char dest[10];
   strncpy( dest, "very very very long", 15 );
   // With secure template overloads enabled (see #defines at
   // top of file), the preceding line is replaced by
   //    strncpy_s( dest, _countof(dest), "very very very long", 15 );
   // Instead of causing a buffer overrun, strncpy_s invokes
   // the invalid parameter handler.
   // If secure template overloads were disabled, strncpy would
   // copy 15 characters and overrun the dest buffer.
   printf( "    new contents of dest: '%s'\n", dest );
}

void myInvalidParameterHandler(
```

```
   const wchar_t* expression,
   const wchar_t* function,
   const wchar_t* file,
   unsigned int line,
   uintptr_t pReserved)
{
   wprintf(L"Invalid parameter handler invoked: %s\n", expression);
}

int main( void )
{
   _invalid_parameter_handler oldHandler, newHandler;

   newHandler = myInvalidParameterHandler;
   oldHandler = _set_invalid_parameter_handler(newHandler);
   // Disable the message box for assertions.
   _CrtSetReportMode(_CRT_ASSERT, 0);

   Examples();
}
```

Output

```
Copying 4 chars of 'howdy' to 10-byte buffer dest
    new contents of dest: 'howd'

Copying 5 chars of 'howdy' to 10-byte buffer dest
    new contents of dest: 'howdy'

Copying 6 chars of 'howdy' to 10-byte buffer dest
    new contents of dest: 'howdy'

Destination buffer too small:

Copying 10 chars of 'Hi there!!' to 10-byte buffer dest
Invalid parameter handler invoked: (L"Buffer is too small" && 0)
    new contents of dest: ''

Truncation examples:

Copying 'How do you do?' to 10-byte buffer dest with truncation semantics
    new contents of dest: 'How do yo'
    truncation did occur

Copying 'Howdy.' to 10-byte buffer dest with truncation semantics
    new contents of dest: 'Howdy.'
    truncation did not occur

Secure template overload example:
Invalid parameter handler invoked: (L"Buffer is too small" && 0)
    new contents of dest: ''
```

```
// crt_strncpy_s_2.c
// contrasts strncpy and strncpy_s

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   char a[20] = "test";
   char s[20];

   // simple strncpy usage:

   strcpy_s( s, 20, "dogs like cats" );
   printf( "Original string:\n   '%s'\n", s );

   // Here we can't use strncpy_s since we don't
```

```
    // want null termination
    strncpy( s, "mice", 4 );
    printf( "After strncpy (no null-termination):\n   '%s'\n", s );
    strncpy( s+5, "love", 4 );
    printf( "After strncpy into middle of string:\n   '%s'\n", s );

    // If we use strncpy_s, the string is terminated
    strncpy_s( s, _countof(s), "mice", 4 );
    printf( "After strncpy_s (with null-termination):\n   '%s'\n", s );

  }
```

Output

```
Original string:
   'dogs like cats'
After strncpy (no null-termination):
   'mice like cats'
After strncpy into middle of string:
   'mice love cats'
After strncpy_s (with null-termination):
   'mice'
```

.NET Framework Equivalent

System::String::Copy

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
_mbsnbcpy, _mbsnbcpy_l
strcat_s, wcscat_s, _mbscat_s
strcmp, wcscmp, _mbscmp
strcpy_s, wcscpy_s, _mbscpy_s
strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l
strspn, wcsspn, _mbsspn, _mbsspn_l

strnicmp, wcsnicmp

These POSIX functions are deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l instead.

_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l

Compare characters of two strings without regard to case.

```
int _strnicmp(
   const char *string1,
   const char *string2,
   size_t count
);
int _wcsnicmp(
   const wchar_t *string1,
   const wchar_t *string2,
   size_t count
);
int _mbsnicmp(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count
);
int _strnicmp_l(
   const char *string1,
   const char *string2,
   size_t count,
   _locale_t locale
);
int _wcsnicmp_l(
   const wchar_t *string1,
   const wchar_t *string2,
   size_t count,
   _locale_t locale
);
int _mbsnicmp_l(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count,
   _locale_t locale
);
```

Parameters

string1, *string2*

Null-terminated strings to compare.

count

Number of characters to compare.

locale

Locale to use.

Return Value

Indicates the relationship between the substrings as follows.

Return value	Description
< 0	*string1* substring less than *string2* substring
0	*string1* substring identical to *string2* substring
> 0	*string1* substring greater than *string2* substring

On an error, **_mbsnicmp** returns **_NLSCMPERROR**, which is defined in STRING.H and MBSTRING.H.

Remarks

The **_strnicmp** function lexicographically compares, at most, the first *count* characters of *string1* and *string2*. The comparison is performed without regard to case; **_strnicmp** is a case-insensitive version of **strncmp**. The comparison ends if a terminating null character is reached in either string before *count* characters are compared. If the strings are equal when a terminating null character is reached in either string before *count* characters are compared, the shorter string is lesser.

The characters from 91 to 96 in the ASCII table ('[', '\', ']', '^', '_', and '`') will evaluate as less than any alphabetic character. This ordering is identical to that of **stricmp**.

_wcsnicmp and **_mbsnicmp** are wide-character and multibyte-character versions of **_strnicmp**. The arguments and return value of **_wcsnicmp** are wide-character strings; those of **_mbsnicmp** are multibyte-character strings. **_mbsnicmp** recognizes multibyte-character sequences according to the current multibyte code page and returns **_NLSCMPERROR** on an error. For more information, see Code Pages. These three functions behave identically otherwise. These functions are affected by the locale setting. The versions without the **_l** suffix use the current locale for their locale-dependent behavior. The versions with the **_l** suffix are identical except that they use the locale passed in instead.

All of these functions validate their parameters. If either *string1* or *string2* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **_NLSCMPERROR** and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsncicmp	**_strnicmp**	**_mbsnicmp**	**_wcsnicmp**
_tcsnicmp	**_strnicmp**	**_mbsnbicmp**	**_wcsnicmp**
_tcsncicmp_l	**_strnicmp_l**	**_mbsnicmp_l**	**_wcsnicmp_l**

Requirements

Routine	Required header	Compatibility
_strnicmp, **_strnicmp_l**	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsnicmp, **_wcsnicmp_l**	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsnicmp, **_mbsnicmp_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for strncmp.

.NET Framework Equivalent

System::String::Compare

See Also

Reference

String Manipulation (CRT)
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strcpy, wcscpy, _mbscpy
strncat, _strncat_l, wcsncat, wcsncat_l, _mbsncat _mbsncat_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l

strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l
strspn, wcsspn, _mbsspn, _mbsspn_l

_strnicoll, _wcsnicoll, _mbsnicoll, _strnicoll_l, _wcsnicoll_l, _mbsnicoll_l

Compare strings using locale-specific information.

```
int _strnicoll(
   const char *string1,
   const char *string2,
   size_t count
);
int _wcsnicoll(
   const wchar_t *string1,
   const wchar_t *string2 ,
   size_t count
);
int _mbsnicoll(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count
);
int _strnicoll_l(
   const char *string1,
   const char *string2,
   size_t count,
   _locale_t locale
);
int _wcsnicoll_l(
   const wchar_t *string1,
   const wchar_t *string2 ,
   size_t count,
   _locale_t locale
);
int _mbsnicoll_l(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count,
   _locale_t locale
);
```

Parameters

string1, *string2*

Null-terminated strings to compare

count

Number of characters to compare

locale

The locale to use.

Return Value

Each of these functions returns a value indicating the relationship of the substrings of *string1* and *string2*, as follows.

Return value	**Relationship of string1 to string2**
< 0	*string1* less than *string2*
0	*string1* identical to *string2*
> 0	*string1* greater than *string2*

Each of these functions returns **_NLSCMPERROR**. To use **_NLSCMPERROR**, include either STRING.H or MBSTRING.H. **_wcsnicoll** can fail if either *string1* or *string2* contains wide-character codes outside the domain of the collating sequence. When an error occurs, **_wcsnicoll** may set **errno** to **EINVAL**. To check for an error on a call to **_wcsnicoll**, set **errno** to 0 and then check **errno** after calling **_wcsnicoll.**

Remarks

Each of these functions performs a case-insensitive comparison of the first *count* characters in *string1* and *string2* according to the code page. These functions should be used only when there is a difference between the character set order and the lexicographic character order in the code page and this difference is of interest for the string comparison. The versions of these functions without the **_l** suffix use the current locale and code page. The versions with the **_l** suffix are identical except that they use the locale passed in instead.

All of these functions validate their parameters. If either *string1* or *string2* is a null pointer, or if count is greater than **INT_MAX**, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, these functions return **_NLSCMPERROR** and set **errno** to **EINVAL.**

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsncicoll	**_strnicoll**	**_mbsnbicoll**	**_wcsnicoll**
_tcsnicoll	**_strnicoll**	_mbsnbicoll	**_wcsnicoll**
_tcsnicoll_l	**_strnicoll_l**	**_mbsnbicoll_l**	**_wcsnicoll_l**

Requirements

Routine	Required header	Compatibility
_strnicoll, **_strnicoll_l**	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsnicoll, **_wcsnicoll_l**	<wchar.h> or <string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsnicoll, **_mbsnicoll_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::String::Compare

See Also

Reference

Locale
String Manipulation (CRT)
strcoll Functions
localeconv
_mbsnbcoll, _mbsnbcoll_l, _mbsnbicoll, _mbsnbicoll_l
setlocale, _wsetlocale
strcmp, wcscmp, _mbscmp
_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

strnlen, strnlen_l, wcsnlen, wcsnlen_l, _mbsnlen, _mbsnlen_l, _mbstrnlen, _mbstrnlen_l

Get the length of a string, using the current locale or one that has been passed in. More secure versions of strlen, strlen_l, wcslen, wcslen_l, _mbslen, _mbslen_l, _mbstrlen, _mbstrlen_l.

```
size_t strnlen(
   const char *str,
   size_t numberOfElements
);
size_t strnlen_l(
   const char *str,
   size_t numberOfElements,
   _locale_t locale
);
size_t wcsnlen(
   const wchar_t *str,
   size_t numberOfElements
);
size_t wcsnlen_l(
   const wchar_t *str,
   size_t numberOfElements,
   _locale_t locale
);
size_t _mbsnlen(
   const unsigned char *str,
   size_t numberOfElements
);
size_t _mbsnlen_l(
   const unsigned char *str,
   size_t numberOfElements,
   _locale_t locale
);
size_t _mbstrnlen(
   const char *str,
   size_t numberOfElements
);
size_t _mbstrnlen_l(
   const char *str,
   size_t numberOfElements,
   _locale_t locale
);
```

Parameters

str

Null-terminated string.

numberOfElements

The size of the string buffer.

locale

Locale to use.

Return Value

These functions return the number of characters in the string (excluding the terminating null). If there is no null terminator within the first *numberOfElements* characters, then *numberOfElements* is returned to indicate the error condition; null-terminated strings have lengths strictly less than *numberOfElements*.

_mbstrnlen and **_mbstrnlen_l** return -1 if the string contains an invalid multibyte character.

Remarks

Note
strnlen is not a replacement for **strlen**; **strnlen** is only intended to be used to calculate the size of incoming untrusted data in a buffer of known size (such as a network packet). **strnlen** will calculate the length but not walk past the end of your buffer if the string is unterminated. For other situations, use **strlen**. (The same applies to **wcsnlen**, **_mbsnlen**, and **_mbstrnlen**.)

Each of these functions returns the number of characters in *str*, not including the terminating null character. However, **strnlen** and **strnlen_l** interpret the string as a single-byte character string, so its return value is always equal to the number of bytes, even if the string contains multibyte characters. **wcsnlen** and **wcsnlen_l** are wide-character versions of **strnlen** and **strnlen_l** respectively; the arguments for **wcsnlen** and **wcsnlen_l** are wide-character strings and the count of characters are in wide character units. **wcsnlen**, **wcsnlen_l**, **strnlen** and **strnlen_l** otherwise behave identically.

strnlen, **wcsnlen,** and **_mbsnlen** do not validate their parameters. If *str* is **NULL**, an access violation will occur.

_mbstrnlen validates its parameters. If *str* is **NULL**, or if *numberOfElements* is greater than **INT_MAX**, **_mbstrnlen** generates an invalid parameter exception, as described in Parameter Validation. If execution is allowed to continue, **_mbstrnlen** sets **errno** to **EINVAL** and returns -1.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsnlen	**strnlen**	**strnlen**	**wcsnlen**
_tcscnlen	**strnlen**	**_mbsnlen**	**wcsnlen**
_tcscnlen_l	**strnlen_l**	**_mbsnlen_l**	**wcsnlen_l**

_mbsnlen and **_mbstrnlen** return the number of multibyte characters in a multibyte-character string. **_mbsnlen** recognizes multibyte-character sequences according to the multibyte code page currently in use or the locale passed in; it does not test for multibyte-character validity. **_mbstrnlen** tests for multibyte-character validity and recognizes multibyte-character sequences. If the string passed to **_mbstrnlen** contains an invalid multibyte character, **errno** is set to **EILSEQ**.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Requirements

Routine	Required header	Compatibility
strnlen	<string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
strnlen_l	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcsnlen, **wcsnlen_l**	<string.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsnlen, **_mbsnlen_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbstrnlen, **_mbstrnlen_l**	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strnlen.c

#include <string.h>

int main()
{
   // str1 is 82 characters long. str2 is 159 characters long

   char* str1 = "The length of a string is the number of characters\n"
                "excluding the terminating null.";
   char* str2 = "strnlen takes a maximum size. If the string is longer\n"
                 "than the maximum size specified, the maximum size is\n"
                 "returned rather than the actual size of the string.";
   size_t len;
   size_t maxsize = 100;

   len = strnlen(str1, maxsize);
   printf("%s\n Length: %d \n\n", str1, len);

   len = strnlen(str2, maxsize);
   printf("%s\n Length: %d \n", str2, len);
}
```

Output

```
The length of a string is the number of characters
excluding the terminating null.
 Length: 82

strnlen takes a maximum size. If the string is longer
than the maximum size specified, the maximum size is
returned rather than the actual size of the string.
 Length: 100
```

.NET Framework Equivalent

System::String::Length

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
setlocale, _wsetlocale
strncat, _strncat_l, wcsncat, wcsncat_l, _mbsncat _mbsncat_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strcoll Functions
strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l
strspn, wcsspn, _mbsspn, _mbsspn_l

strnset, wcsnset

These POSIX functions are deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l or security-enhanced _strnset_s, _strnset_s_l, _wcsnset_s, _wcsnset_s_l, _mbsnset_s, _mbsnset_s_l instead.

_strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l

Initialize characters of a string to a given character. These functions are deprecated because more secure versions exist; see _strnset_s, _strnset_s_l, _wcsnset_s, _wcsnset_s_l, _mbsnset_s, _mbsnset_s_l.

```
char *_strnset(
   char *str,
   int c,
   size_t count
);
char *_strnset_l(
   char *str,
   int c,
   size_t count,
   locale_t locale
);
wchar_t *_wcsnset(
   wchar_t *str,
   wchar_t c,
   size_t count
);
wchar_t *_wcsnset_l(
   wchar_t *str,
   wchar_t c,
   size_t count,
   _locale_t locale
);
unsigned char *_mbsnset(
   unsigned char *str,
   unsigned int c,
   size_t count
);
unsigned char *_mbsnset_l(
   unsigned char *str,
   unsigned int c,
   size_t count,
   _locale_t locale
);
```

Parameters

str

String to be altered.

c

Character setting.

count

Number of characters to be set.

locale

Locale to use.

Return Value

Returns a pointer to the altered string.

Remarks

The **_strnset** function sets, at most, the first *count* characters of *str* to *c* (converted to **char**). If *count* is greater than the length of *str*, the length of *str* is used instead of *count*.

_wcsnset and **_mbsnset** are wide-character and multibyte-character versions of **_strnset**. The string arguments and return value of **_wcsnset** are wide-character strings; those of **_mbsnset** are multibyte-character strings. These three functions behave

identically otherwise.

_mbsnset validates its parameters; if *str* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, **_mbsnset** returns NULL and sets **errno** to **EINVAL**. **_strnset** and **_wcsnset** do not validate their parameters.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsnset	**_strnset**	**_mbsnbset**	**_wcsnset**
_tcsnset_l	**_strnset_l**	**_mbsnbset_l**	**_wcsnset_l**

Requirements

Routine	Required header	Compatibility
_strnset	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_strnset_l	<tchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsnset	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsnset_l	<tchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsnset, **_mbsnset_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strnset.c
// compile with: /W3
#include <string.h>
#include <stdio.h>

int main( void )
{
   char string[15] = "This is a test";
   /* Set not more than 4 characters of string to be *'s */
   printf( "Before: %s\n", string );
   _strnset( string, '*', 4 ); // C4996
   // Note: _strnset is deprecated; consider using _strnset_s
   printf( "After:  %s\n", string );
}
```

Output

```
Before: This is a test
After:  **** is a test
```

.NET Framework Equivalent

System::String::Replace

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strcpy, wcscpy, _mbscpy
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l

_strnset_s, _strnset_s_l, _wcsnset_s, _wcsnset_s_l, _mbsnset_s, _mbsnset_s_l

Initialize characters of a string to a given character. These are versions of _strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _strnset_s(
   char *str,
     size_t numberOfElements,
   int c,
   size_t count
);
errno_t _strnset_s_l(
   char *str,
   size_t numberOfElements,
   int c,
   size_t count,
   locale_t locale
);
errno_t _wcsnset_s(
   wchar_t *str,
   size_t numberOfElements,
   wchar_t c,
   size_t count
);
errno_t _wcsnset_s_l(
   wchar_t *str,
   size_t numberOfElements,
   wchar_t c,
   size_t count,
   _locale_t locale
);
errno_t _mbsnset_s(
   unsigned char *str,
   size_t numberOfElements,
   unsigned int c,
   size_t count
);
errno_t _mbsnset_s_l(
   unsigned char *str,
   size_t numberOfElements,
   unsigned int c,
   size_t count,
   _locale_t locale
);
```

Parameters

str

String to be altered.

numberOfElements

The size of the *str* buffer.

c

Character setting.

count

Number of characters to be set.

locale

Locale to use.

Return Value

Zero if successful, otherwise an error code.

These functions validate their arguments. If *str* is a null pointer or the size argument is less than or equal to 0, then the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **EINVAL** and set **errno** to **EINVAL**.

Remarks

These functions set, at most, the first *count* characters of *str* to *c*. If *count* is greater than the size of *str* (that is, *numberOfElements*), the size is used instead of *count*.

_wcsnset_s and **_mbsnset_s** are wide-character and multibyte-character versions of **_strnset_s**. The string argument of **_wcsnset_s** is a wide-character string; that of **_mbsnset_s** is a multibyte-character string. These three functions behave identically otherwise.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsnset_s	**_strnset_s**	**_mbsnbset_s**	**_wcsnset_s**
_tcsnset_s_l	**_strnset_s_l**	**_mbsnbset_s_l**	**_wcsnset_s_l**

Requirements

Routine	Required header	Compatibility
_strnset_s	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003, Windows Server 2003
_strnset_s_l	<tchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003, Windows Server 2003
_wcsnset_s	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003, Windows Server 2003
_wcsnset_s_l	<tchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003, Windows Server 2003
_mbsnset_s, **_mbsnset_s_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strnset_s.c
#include <string.h>
#include <stdio.h>
```

```
int main( void )
{
   char string[15] = "This is a test";
   /* Set not more than 4 characters of string to be *'s */
   printf( "Before: %s\n", string );
   _strnset_s( string, sizeof(string), '*', 4 );
   printf( "After:  %s\n", string );
}
```

Output

```
Before: This is a test
After:  **** is a test
```

.NET Framework Equivalent

System::String::Replace

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strcpy, wcscpy, _mbscpy
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l

strpbrk, wcspbrk, _mbspbrk, _mbspbrk_l

Scan strings for characters in specified character sets.

```
char *strpbrk(
   const char *str,
   const char *strCharSet
); // C only
char *strpbrk(
   char *str,
   const char *strCharSet
); // C++ only
const char *strpbrk(
   const char *str,
   const char *strCharSet
); // C++ only
wchar_t *wcspbrk(
   const wchar_t *str,
   const wchar_t *strCharSet
); // C only
wchar_t *wcspbrk(
   wchar_t *str,
   const wchar_t *strCharSet
); // C++ only
const wchar_t *wcspbrk(
   const wchar_t *str,
   const wchar_t *strCharSet
); // C++ only
unsigned char *_mbspbrk(
   const unsigned char *str,
   const unsigned char *strCharSet
); // C only
unsigned char *_mbspbrk(
   unsigned char *str,
   const unsigned char *strCharSet
); // C++ only
const unsigned char *_mbspbrk(
   const unsigned char *str,
   const unsigned char *strCharSet
); // C++ only
unsigned char *_mbspbrk_l(
   const unsigned char *str,
   const unsigned char *strCharSet,
   _locale_t locale
); // C only
unsigned char *_mbspbrk_l(
   unsigned char *str,
   const unsigned char *strCharSet,
   _locale_t locale
); // C++ only
const unsigned char *_mbspbrk_l(
   const unsigned char *str,
   const unsigned char* strCharSet,
   _locale_t locale
); // C++ only
```

Parameters

str

Null-terminated, searched string.

strCharSet

Null-terminated character set.

locale

Locale to use.

Return Value

Returns a pointer to the first occurrence of any character from *strCharSet* in *str*, or a **NULL** pointer if the two string arguments have no characters in common.

Remarks

The **strpbrk** function returns a pointer to the first occurrence of a character in *str* that belongs to the set of characters in *strCharSet*. The search does not include the terminating null character.

wcspbrk and **_mbspbrk** are wide-character and multibyte-character versions of **strpbrk**. The arguments and return value of **wcspbrk** are wide-character strings; those of **_mbspbrk** are multibyte-character strings.

_mbspbrk validates its parameters. If *str* or *strCharSet* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **_mbspbrk** returns **NULL** and sets **errno** to **EINVAL**. **strpbrk** and **wcspbrk** do not validate their parameters. These three functions behave identically otherwise.

_mbspbrk is similar to **_mbscspn** except that **_mbspbrk** returns a pointer rather than a value of type size_t.

In C, these functions take a **const** pointer for the first argument. In C++, two overloads are available. The overload taking a pointer to **const** returns a pointer to **const**; the version that takes a pointer to non-**const** returns a pointer to non-**const**. The macro _CONST_CORRECT_OVERLOADS is defined if both the **const** and non-**const** versions of these functions are available. If you require the non-**const** behavior for both C++ overloads, define the symbol _CONST_RETURN.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; for more information, see setlocale. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the version with the **_l** suffix is identical except that it uses the locale parameter passed in instead.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcspbrk	**strpbrk**	**_mbspbrk**	**wcspbrk**
n/a	**n/a**	**_mbspbrk_l**	**n/a**

Requirements

Routine	Required header	Compatibility
strpbrk	<string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcspbrk	<string.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbspbrk, **_mbspbrk_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more information about compatibility, see Compatibility.

Example

```
// crt_strpbrk.c

#include <string.h>
#include <stdio.h>

int main( void )
{
   char string[100] = "The 3 men and 2 boys ate 5 pigs\n";
   char *result = NULL;
```

```
    // Return pointer to first digit in "string".
    printf( "1: %s\n", string );
    result = strpbrk( string, "0123456789" );
    printf( "2: %s\n", result++ );
    result = strpbrk( result, "0123456789" );
    printf( "3: %s\n", result++ );
    result = strpbrk( result, "0123456789" );
    printf( "4: %s\n", result );
}
```

Output

```
1: The 3 men and 2 boys ate 5 pigs

2: 3 men and 2 boys ate 5 pigs

3: 2 boys ate 5 pigs

4: 5 pigs
```

.NET Framework Equivalent

System::String::IndexOfAny

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
strcspn, wcscspn, _mbscspn, _mbscspn_l
strchr, wcschr, _mbschr, _mbschr_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l

strrchr, wcsrchr, _mbsrchr, _mbsrchr_l

Scan a string for the last occurrence of a character.

```
char *strrchr(
   const char *str,
   int c
); // C only
char *strrchr(
   char *str,
   int c
); // C++ only
const char *strrchr(
   const char *str,
   int c
); // C++ only
wchar_t *wcsrchr(
   const wchar_t *str,
   wchar_t c
); // C only
wchar_t *wcsrchr(
   wchar_t *str,
   wchar_t c
); // C++ only
const wchar_t *wcsrchr(
   const wchar_t *str,
   wchar_t c
); // C++ only
unsigned char *_mbsrchr(
   const unsigned char *str,
   unsigned int c
); // C only
unsigned char *_mbsrchr(
   unsigned char *str,
   unsigned int c
); // C++ only
const unsigned char *_mbsrchr(
   const unsigned char *str,
   unsigned int c
); // C++ only
unsigned char *_mbsrchr_l(
   const unsigned char *str,
   unsigned int c,
   _locale_t locale
); // C only
unsigned char *_mbsrchr_l(
   unsigned char *str,
   unsigned int c,
   _locale_t locale
); // C++ only
const unsigned char *_mbsrchr_l(
   const unsigned char *str,
   unsigned int c,
   _locale_t locale
); // C++ only
```

Parameters

str

Null-terminated string to search.

c

Character to be located.

locale

Locale to use.

Return Value

Returns a pointer to the last occurrence of *c* in *str*, or **NULL** if *c* is not found.

Remarks

The **strrchr** function finds the last occurrence of *c* (converted to **char**) in *str*. The search includes the terminating null character.

wcsrchr and **_mbsrchr** are wide-character and multibyte-character versions of **strrchr**. The arguments and return value of **wcsrchr** are wide-character strings; those of **_mbsrchr** are multibyte-character strings.

In C, these functions take a **const** pointer for the first argument. In C++, two overloads are available. The overload taking a pointer to **const** returns a pointer to **const**; the version that takes a pointer to non-**const** returns a pointer to non-**const**. The macro _CONST_CORRECT_OVERLOADS is defined if both the **const** and non-**const** versions of these functions are available. If you require the non-**const** behavior for both C++ overloads, define the symbol _CONST_RETURN.

_mbsrchr validates its parameters. If *str* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and **_mbsrchr** returns 0. **strrchr** and **wcsrchr** do not validate their parameters. These three functions behave identically otherwise.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; for more information, see setlocale. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsrchr	**strrchr**	**_mbsrchr**	**wcsrchr**
n/a	**n/a**	**_mbsrchr_l**	**n/a**

Requirements

Routine	Required header	Compatibility
strrchr	<string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcsrchr	<string.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsrchr, **_mbsrchr_l**	<mbstring.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more information about compatibility, see Compatibility.

Example

For an example of using **strrchr**, see strchr.

.NET Framework Equivalent

System::String::LastIndexOf

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
strchr, wcschr, _mbschr, _mbschr_l
strcspn, wcscspn, _mbscspn, _mbscspn_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l

strpbrk, wcspbrk, _mbspbrk, _mbspbrk_l
strspn, wcsspn, _mbsspn, _mbsspn_l

strrev, wcsrev

These POSIX functions are deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _strrev, _wcsrev, _mbsrev, _mbsrev_l instead.

_strrev, _wcsrev, _mbsrev, _mbsrev_l

Reverse characters of a string.

```
char *_strrev(
   char *str
);
wchar_t *_wcsrev(
   wchar_t *str
);
unsigned char *_mbsrev(
   unsigned char *str
);
unsigned char *_mbsrev_l(
   unsigned char *str,
   _locale_t locale
);
```

Parameters

str

Null-terminated string to reverse.

locale

Locale to use.

Return Value

Returns a pointer to the altered string. No return value is reserved to indicate an error.

Remarks

The **_strrev** function reverses the order of the characters in *string*. The terminating null character remains in place. **_wcsrev** and **_mbsrev** are wide-character and multibyte-character versions of **_strrev**. The arguments and return value of **_wcsrev** are wide-character strings; those of **_mbsrev** are multibyte-character strings. For **_mbsrev**, the order of bytes in each multibyte character in *string* is not changed. These three functions behave identically otherwise.

_mbsrev validates its parameters. If either *string1* or *string2* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **_mbsrev** returns **NULL** and sets **errno** to **EINVAL**. **_strrev** and **_wcsrev** do not validate their parameters.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Security Note These functions incur a potential threat brought about by a buffer overrun problem. Buffer overrun problems are a frequent method of system attack, resulting in an unwarranted elevation of privilege. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsrev	**_strrev**	**_mbsrev**	**_wcsrev**
n/a	**n/a**	**_mbsrev_l**	**n/a**

Requirements

Routine	Required header	Compatibility
_strrev	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_wcsrev	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsrev, **_mbsrev_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strrev.c
// This program checks a string to see
// whether it is a palindrome: that is, whether
// it reads the same forward and backward.
//

#include <string.h>
#include <stdio.h>

int main( void )
{
   char* string = "Able was I ere I saw Elba";
   int result;

   // Reverse string and compare (ignore case):
   result = _stricmp( string, _strrev( _strdup( string ) ) );
   if( result == 0 )
      printf( "The string \"%s\" is a palindrome\n", string );
   else
      printf( "The string \"%s\" is not a palindrome\n", string );
}
```

Output

```
The string "Able was I ere I saw Elba" is a palindrome
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
strcpy, wcscpy, _mbscpy
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l

strset, wcsset

These POSIX functions are deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l or security-enhanced _strset_s, _strset_s_l, _wcsset_s, _wcsset_s_l, _mbsset_s, _mbsset_s_l instead.

_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l

Set characters of a string to a character. These functions are deprecated because more secure versions are available; see _strset_s, _strset_s_l, _wcsset_s, _wcsset_s_l, _mbsset_s, _mbsset_s_l.

```
char *_strset(
   char *str,
   int c
);
char *_strset_l(
   char *str,
   int c,
   locale_t locale
);
wchar_t *_wcsset(
   wchar_t *str,
    wchar_t c
);
wchar_t *_wcsset_l(
   wchar_t *str,
   wchar_t c,
   locale_t locale
);
unsigned char *_mbsset(
   unsigned char *str,
   unsigned int c
);
unsigned char *_mbsset_l(
   unsigned char *str,
   unsigned int c,
   _locale_t locale
);
```

Parameters

str

Null-terminated string to be set.

c

Character setting.

locale

Locale to use.

Return Value

Returns a pointer to the altered string.

Remarks

The **_strset** function sets all the characters of *str* to *c* (converted to **char**), except the terminating null character. **_wcsset** and **_mbsset_l** are wide-character and multibyte-character versions of **_strset**. The data types of the arguments and return values vary accordingly. These functions behave identically otherwise.

_mbsset validates its parameters. If *str* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **_mbsset** returns **NULL** and sets **errno** to **EINVAL**. **_strset** and **_wcsset** do not validate their parameters.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Security Note These functions incur a potential threat brought about by a buffer overrun problem. Buffer overrun problems are a frequent method of system attack, resulting in an unwarranted elevation of privilege. For more information, see

Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsset	**_strset**	**_mbsset**	**_wcsset**
_tcsset_l	**_strset_l**	**_mbsset_l**	**_wcsset_l**

Requirements

Routine	Required header	Compatibility
_strset	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_strset_l	<tchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsset	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsset_l	<tchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsset, **_mbsset_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strset.c
// compile with: /W3

#include <string.h>
#include <stdio.h>

int main( void )
{
   char string[] = "Fill the string with something.";
   printf( "Before: %s\n", string );
   _strset( string, '*' ); // C4996
   // Note: _strset is deprecated; consider using _strset_s instead
   printf( "After:  %s\n", string );
}
```

Output

```
Before: Fill the string with something.
After:  ******************************
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
_mbsnbset, _mbsnbset_l
memset, wmemset
strcat, wcscat, _mbscat

strcmp, wcscmp, _mbscmp
strcpy, wcscpy, _mbscpy
_strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l

_strset_s, _strset_s_l, _wcsset_s, _wcsset_s_l, _mbsset_s, _mbsset_s_l

Set characters of a string to a character. These are versions of _strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _strset_s(
   char *str,
   size_t numberOfElements,
   int c
);
errno_t _strset_s_l(
   char *str,
   size_t numberOfElements,
   int c,
   locale_t locale
);
errno_t _wcsset_s(
   wchar_t *str,
   size_t numberOfElements,
   wchar_t c
);
errno_t *_wcsset_s_l(
   wchar_t *str,
   size_t numberOfElements,
   wchar_t c,
   locale_t locale
);
errno_t _mbsset_s(
      unsigned char *str,
   size_t numberOfElements,
   unsigned int c
);
errno_t _mbsset_s_l(
   unsigned char *str,
   size_t numberOfElements,
   unsigned int c,
   _locale_t locale
);
```

Parameters

str

Null-terminated string to be set.

numberOfElements

The size of the *str* buffer.

c

Character setting.

locale

Locale to use.

Return Value

Zero if successful, otherwise an error code.

These functions validate their arguments. If *str* is a null pointer, or the size argument is less than or equal to 0, or the block passed in is not null-terminated, then the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **EINVAL** and set **errno** to **EINVAL**.

Remarks

The **_strset_s** function sets all the characters of *str* to *c* (converted to **char**), except the terminating null character. **_wcsset_s** and **_mbsset_s** are wide-character and multibyte-character versions of **_strset_s**. The data types of the arguments and return values vary accordingly. These functions behave identically otherwise.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsset_s	**_strset_s**	**_mbsset_s**	**_wcsset_s**
_tcsset_s_l	**_strset_s_l**	**_mbsset_s_l**	**_wcsset_s_l**

Requirements

Routine	Required header	Compatibility
_strset_s	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_strset_s_l	<tchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsset_s	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsset_s_l	<tchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsset_s, **_mbsset_s_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strset_s.c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   char string[] = "Fill the string with something.";
   printf( "Before: %s\n", string );
   _strset_s( string, _countof(string), '*' );
   printf( "After:  %s\n", string );
}
```

Output

```
Before: Fill the string with something.
After:  ******************************
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
_mbsnbset, _mbsnbset_l
memset, wmemset
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strcpy, wcscpy, _mbscpy
_strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l

strspn, wcsspn, _mbsspn, _mbsspn_l

Returns the index of the first character in a string that does not belong to a set of characters.

```
size_t strspn(
   const char *str,
   const char *strCharSet
);
size_t wcsspn(
   const wchar_t *str,
   const wchar_t *strCharSet
);
size_t _mbsspn(
   const unsigned char *str,
   const unsigned char *strCharSet
);
size_t _mbsspn_l(
   const unsigned char *str,
   const unsigned char *strCharSet,
   _locale_t locale
);
```

Parameters

str

Null-terminated string to search.

strCharSet

Null-terminated character set.

locale

Locale to use.

Return Value

Returns an integer value specifying the length of the substring in *str* that consists entirely of characters in *strCharSet*. If *str* begins with a character not in *strCharSet*, the function returns 0.

Remarks

The **strspn** function returns the index of the first character in *str* that does not belong to the set of characters in *strCharSet*. The search does not include terminating null characters.

wcsspn and **_mbsspn** are wide-character and multibyte-character versions of **strspn.** The arguments of **wcsspn** are wide-character strings; those of **_mbsspn** are multibyte-character strings. **_mbsspn** validates its parameters. If *str* or *strCharSet* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, **_mbspn** sets **errno** to **EINVAL** and returns 0. **strspn** and **wcsspn** do not validate their parameters. These three functions behave identically otherwise.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsspn	strspn	_mbsspn	wcsspn
n/a	n/a	_mbsspn_l	n/a

Requirements

Routine	Required header	Compatibility

strspn	<string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcsspn	<string.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsspn, **_mbsspn_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strspn.c
// This program uses strspn to determine
// the length of the segment in the string "cabbage"
// consisting of a's, b's, and c's. In other words,
// it finds the first non-abc letter.
//

#include <string.h>
#include <stdio.h>

int main( void )
{
   char string[] = "cabbage";
   int  result;
   result = strspn( string, "abc" );
   printf( "The portion of '%s' containing only a, b, or c "
           "is %d bytes long\n", string, result );
}
```

Output

```
The portion of 'cabbage' containing only a, b, or c is 5 bytes long
```

.NET Framework Equivalent

System::String::Substring

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
_mbsspnp, _mbsspnp_l, _strspnp, _wcsspnp
strcspn, wcscspn, _mbscspn, _mbscspn_l
strncat, _strncat_l, wcsncat, wcsncat_l, _mbsncat _mbsncat_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l

strstr, wcsstr, _mbsstr, _mbsstr_l

Return a pointer to the first occurrence of a search string in a string.

```
char *strstr(
   const char *str,
   const char *strSearch
); // C only
char *strstr(
   char *str,
   const char *strSearch
); // C++ only
const char *strstr(
   const char *str,
   const char *strSearch
); // C++ only
wchar_t *wcsstr(
   const wchar_t *str,
   const wchar_t *strSearch
); // C only
wchar_t *wcsstr(
   wchar_t *str,
   const wchar_t *strSearch
); // C++ only
const wchar_t *wcsstr(
   const wchar_t *str,
   const wchar_t *strSearch
); // C++ only
unsigned char *_mbsstr(
   const unsigned char *str,
   const unsigned char *strSearch
); // C only
unsigned char *_mbsstr(
   unsigned char *str,
   const unsigned char *strSearch
); // C++ only
const unsigned char *_mbsstr(
   const unsigned char *str,
   const unsigned char *strSearch
); // C++ only
unsigned char *_mbsstr_l(
   const unsigned char *str,
   const unsigned char *strSearch,
   _locale_t locale
); // C only
unsigned char *_mbsstr_l(
   unsigned char *str,
   const unsigned char *strSearch,
   _locale_t locale
); // C++ only
const unsigned char *_mbsstr_l(
   const unsigned char *str,
   const unsigned char *strSearch,
   _locale_t locale
); // C++ only
```

Parameters

str

Null-terminated string to search.

strSearch

Null-terminated string to search for.

locale

Locale to use.

Return Value

Returns a pointer to the first occurrence of *strSearch* in *str*, or **NULL** if *strSearch* does not appear in *str*. If *strSearch* points to a string of zero length, the function returns *str*.

Remarks

The **strstr** function returns a pointer to the first occurrence of *strSearch* in *str*. The search does not include terminating null characters. **wcsstr** and **_mbsstr** are wide-character and multibyte-character versions of **strstr**. The arguments and return value of **wcsstr** are wide-character strings; those of **_mbsstr** are multibyte-character strings. **_mbsstr** validates its parameters. If *str* or *strSearch* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, **_mbsstr** sets **errno** to **EINVAL** and returns 0. **strstr** and **wcsstr** do not validate their parameters. These three functions behave identically otherwise.

Security Note
These functions incur a potential threat brought about by a buffer overrun problem. Buffer overrun problems are a frequent method of system attack, resulting in an unwarranted elevation of privilege. For more information, see Avoiding Buffer Overruns.

In C, these functions take a **const** pointer for the first argument. In C++, two overloads are available. The overload taking a pointer to **const** returns a pointer to **const**; the version that takes a pointer to non-**const** returns a pointer to non-**const**. The macro _CONST_CORRECT_OVERLOADS is defined if both the **const** and non-**const** versions of these functions are available. If you require the non-**const** behavior for both C++ overloads, define the symbol _CONST_RETURN.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; for more information, see setlocale. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsstr	**strstr**	**_mbsstr**	**wcsstr**
n/a	**n/a**	**_mbsstr_l**	**n/a**

Requirements

Routine	Required header	Compatibility
strstr	<string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcsstr	<string.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsstr, **_mbsstr_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more information about compatibility, see Compatibility.

Example

```
// crt_strstr.c

#include <string.h>
#include <stdio.h>

char str[] =    "lazy";
char string[] = "The quick brown dog jumps over the lazy fox";
```

```
char fmt1[] =   "         1         2         3         4         5";
char fmt2[] =   "12345678901234567890123456789012345678901234567890";

int main( void )
{
   char *pdest;
   int  result;
   printf( "String to be searched:\n   %s\n", string );
   printf( "   %s\n   %s\n\n", fmt1, fmt2 );
   pdest = strstr( string, str );
   result = (int)(pdest - string + 1);
   if ( pdest != NULL )
      printf( "%s found at position %d\n", str, result );
   else
      printf( "%s not found\n", str );
}
```

Output

```
String to be searched:
   The quick brown dog jumps over the lazy fox
            1         2         3         4         5
   12345678901234567890123456789012345678901234567890

lazy found at position 36
```

.NET Framework Equivalent

System::String::IndexOf

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
strcspn, wcscspn, _mbscspn, _mbscspn_l
strcmp, wcscmp, _mbscmp
strpbrk, wcspbrk, _mbspbrk, _mbspbrk_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
strspn, wcsspn, _mbsspn, _mbsspn_l

_strtime, _wstrtime

Copy the time to a buffer. These functions are deprecated because more secure versions are available; see _strtime_s, _wstrtime_s.

```
char *_strtime(
   char *timestr
);
wchar_t *_wstrtime(
   wchar_t *timestr
);
template <size_t size>
char *_strtime(
   char (&timestr)[size]
); // C++ only
template <size_t size>
wchar_t *_wstrtime(
   wchar_t (&timestr)[size]
); // C++ only
```

Parameters

timestr

Time string.

Return Value

Returns a pointer to the resulting character string *timestr*.

Remarks

The **_strtime** function copies the current local time into the buffer pointed to by *timestr*. The time is formatted as *hh:mm:ss* where *hh* is two digits representing the hour in 24-hour notation, *mm* is two digits representing the minutes past the hour, and *ss* is two digits representing seconds. For example, the string `18:23:44` represents 23 minutes and 44 seconds past 6 P.M. The buffer must be at least 9 bytes long.

_wstrtime is a wide-character version of **_strtime**; the argument and return value of **_wstrtime** are wide-character strings. These functions behave identically otherwise.If *timestr* is **NULL** pointer or if *timestr* is formatted incorrectly, the invalid parameter handler is invoked, as described in Parameter Validation. If the exception is allowed to continue, these functions return a NULL and set **errno** to **EINVAL** if *timestr* was a NULL or set **errno** to **ERANGE** if *timestr* is formatted incorrectly.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tstrtime	**_strtime**	**_strtime**	**_wstrtime**

Requirements

Routine	Required header	Compatibility
_strtime	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wstrtime	<time.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strtime.c
// compile with: /W3

#include <time.h>
#include <stdio.h>

int main( void )
{
   char tbuffer [9];
   _strtime( tbuffer ); // C4996
   // Note: _strtime is deprecated; consider using _strtime_s instead
   printf( "The current time is %s \n", tbuffer );
}
```

Sample Output

```
The current time is 14:21:44
```

.NET Framework Equivalent

- System::DateTime::ToLongDateString
- System::DateTime::ToLongTimeString
- System::DateTime::ToShortDateString
- System::DateTime::ToShortTimeString
- System::DateTime::ToString

See Also

Reference

Time Management
asctime, _wasctime
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
mktime, _mktime32, _mktime64
time, _time32, _time64
_tzset

_strtime_s, _wstrtime_s

Copy the current time to a buffer. These are versions of _strtime, _wstrtime with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _strtime_s(
   char *buffer,
   size_t numberOfElements
);
errno_t _wstrtime_s(
   wchar_t *buffer,
   size_t numberOfElements
);
template <size_t size>
errno_t _strtime_s(
   char (&buffer)[size]
); // C++ only
template <size_t size>
errno_t _wstrtime_s(
   wchar_t (&buffer)[size]
); // C++ only
```

Parameters

[out] *buffer*

A buffer, at least 10 bytes long, where the time will be written.

[in] *numberOfElements*

The size of the buffer.

Return Value

Zero if successful.

If an error condition occurs, the invalid parameter handler is invoked, as described in Parameter Validation. The return value is an error code if there is a failure. Error codes are defined in ERRNO.H; see the following table for the exact errors generated by this function. For more information on error codes, see errno Constants.

Error Conditions

buffer	***numberOfElements***	**Return**	**Contents of *buffer***
NULL	(any)	**EINVAL**	Not modified
Not **NULL** (pointing to valid buffer)	0	**EINVAL**	Not modified
Not **NULL** (pointing to valid buffer)	0 < *numberOfElements* < 9	**EINVAL**	Empty string
Not **NULL** (pointing to valid buffer)	*numberOfElements* > 9	0	Current time formatted as specified in the remarks

Security Issues

Passing in an invalid non-NULL value for the buffer will result in an access violation if the size parameter is greater than 9.

Passing values for size that is greater than the actual size of the buffer will result in buffer overrun.

Remarks

These functions provide more secure versions of **_strtime** and **_wstrtime**. The **_strtime_s** function copies the current local time into the buffer pointed to by *timestr*. The time is formatted as *hh:mm:ss* where *hh* is two digits representing the hour in 24-hour notation, *mm* is two digits representing the minutes past the hour, and *ss* is two digits representing seconds. For example, the string `18:23:44` represents 23 minutes and 44 seconds past 6 P.M. The buffer must be at least 9 bytes long; the actual size is specified by the second parameter.

_wstrtime is a wide-character version of **_strtime**; the argument and return value of **_wstrtime** are wide-character strings.

These functions behave identically otherwise.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Generic-Text Routine Mapping:

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tstrtime_s	**_strtime_s**	**_strtime_s**	**_wstrtime_s**

Requirements

Routine	Required header	Compatibility
_strtime_s	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wstrtime_s	<time.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// strtime_s.c

#include <time.h>
#include <stdio.h>

int main()
{
    char tmpbuf[9];
    errno_t err;

    // Set time zone from TZ environment variable. If TZ is not set,
    // the operating system is queried to obtain the default value
    // for the variable.
    //
    _tzset();

    // Display operating system-style date and time.
    err = _strtime_s( tmpbuf, 9 );
    if (err)
    {
       printf("_strdate_s failed due to an invalid argument.");
      exit(1);
    }
    printf( "OS time:\t\t\t\t%s\n", tmpbuf );
    err = _strdate_s( tmpbuf, 9 );
    if (err)
    {
       printf("_strdate_s failed due to an invalid argument.");
       exit(1);
    }
    printf( "OS date:\t\t\t\t%s\n", tmpbuf );

}
```

Sample Output

```
OS time:            14:37:49
OS date:            04/25/03
```

.NET Framework Equivalent

- System::DateTime::ToLongDateString

- System::DateTime::ToLongTimeString
- System::DateTime::ToShortDateString
- System::DateTime::ToShortTimeString
- System::DateTime::ToString

See Also

Reference

Time Management
asctime_s, _wasctime_s
ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s
_gmtime_s, _gmtime32_s, _gmtime64_s
localtime_s, _localtime32_s, _localtime64_s
mktime, _mktime32, _mktime64
time, _time32, _time64
_tzset

strtod, _strtod_l, wcstod, _wcstod_l

Convert strings to a double-precision value.

```
double strtod(
   const char *nptr,
   char **endptr
);
double _strtod_l(
   const char *nptr,
   char **endptr,
   _locale_t locale
);
double wcstod(
   const wchar_t *nptr,
   wchar_t **endptr
);
double wcstod_l(
   const wchar_t *nptr,
   wchar_t **endptr,
   _locale_t locale
);
```

Parameters

nptr

Null-terminated string to convert.

endptr

Pointer to character that stops scan.

locale

The locale to use.

Return Value

strtod returns the value of the floating-point number, except when the representation would cause an overflow, in which case the function returns +/–**HUGE_VAL**. The sign of **HUGE_VAL** matches the sign of the value that cannot be represented. **strtod** returns 0 if no conversion can be performed or an underflow occurs.

wcstod returns values analogously to **strtod**. For both functions, **errno** is set to **ERANGE** if overflow or underflow occurs and the invalid parameter handler is invoked, as described in Parameter Validation.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on this and other return codes.

Remarks

Each function converts the input string *nptr* to a **double**. The **strtod** function converts *nptr* to a double-precision value. **strtod** stops reading the string *nptr* at the first character it cannot recognize as part of a number. This may be the terminating null character. **wcstod** is a wide-character version of **strtod**; its *nptr* argument is a wide-character string. These functions behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcstod	**strtod**	**strtod**	**wcstod**
_tcstod_l	**_strtod_l**	**_strtod_l**	**_wcstod_l**

The **LC_NUMERIC** category setting of the current locale determines recognition of the radix character in *nptr;* for more information, see setlocale. The functions without the **_l** suffix use the current locale; **_strtod_l** is identical to **_strtod_l** except that they use the locale passed in instead.

If *endptr* is not **NULL**, a pointer to the character that stopped the scan is stored at the location pointed to by *endptr*. If no conversion can be performed (no valid digits were found or an invalid base was specified), the value of *nptr* is stored at the location pointed to by *endptr*.

strtod expects *nptr* to point to a string of the following form:

[*whitespace*] [*sign*] [*digits*] [.*digits*] [{**d** | **D** | **e** | **E**}[*sign*]*digits*]

A *whitespace* may consist of space and tab characters, which are ignored; *sign* is either plus (+) or minus (–); and *digits* are one or more decimal digits. If no digits appear before the radix character, at least one must appear after the radix character. The decimal digits can be followed by an exponent, which consists of an introductory letter (**d**, **D**, **e**, or **E**) and an optionally signed integer. If neither an exponent part nor a radix character appears, a radix character is assumed to follow the last digit in the string. The first character that does not fit this form stops the scan.

Requirements

Routine	Required header	Compatibility
strtod, **_strtod_l**	<stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcstod, **_wcstod_l**	<stdlib.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strtod.c
// This program uses strtod to convert a
// string to a double-precision value; strtol to
// convert a string to long integer values; and strtoul
// to convert a string to unsigned long-integer values.
//

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char   *string, *stopstring;
   double x;
   long   l;
   int    base;
   unsigned long ul;

   string = "3.1415926This stopped it";
   x = strtod( string, &stopstring );
   printf( "string = %s\n", string );
   printf("   strtod = %f\n", x );
   printf("   Stopped scan at: %s\n\n", stopstring );

   string = "-10110134932This stopped it";
   l = strtol( string, &stopstring, 10 );
   printf( "string = %s\n", string );
   printf("   strtol = %ld\n", l );
   printf("   Stopped scan at: %s\n\n", stopstring );

   string = "10110134932";
   printf( "string = %s\n", string );

   // Convert string using base 2, 4, and 8:
   for( base = 2; base <= 8; base *= 2 )
   {
      // Convert the string:
      ul = strtoul( string, &stopstring, base );
      printf( "   strtol = %ld (base %d)\n", ul, base );
```

```
        printf( "   Stopped scan at: %s\n", stopstring );
    }
}
```

Output

```
string = 3.1415926This stopped it
   strtod = 3.141593
   Stopped scan at: This stopped it

string = -10110134932This stopped it
   strtol = -2147483648
   Stopped scan at: This stopped it

string = 10110134932
   strtol = 45 (base 2)
   Stopped scan at: 34932
   strtol = 4423 (base 4)
   Stopped scan at: 4932
   strtol = 2134108 (base 8)
   Stopped scan at: 932
```

.NET Framework Equivalent

System::Convert::ToDouble

See Also

Reference

Data Conversion
Floating-Point Support
Interpretation of Multibyte-Character Sequences
Locale
String to Numeric Value Functions
strtol, wcstol, _strtol_l, _wcstol_l
strtoul, _strtoul_l, wcstoul, _wcstoul_l
atof, _atof_l, _wtof, _wtof_l
localeconv

_strtoi64, _wcstoi64, _strtoi64_l, _wcstoi64_l

Convert a string to an **__int64** value.

```
__int64 _strtoi64(
   const char *nptr,
   char **endptr,
   int base
);
__int64 _wcstoi64(
   const wchar_t *nptr,
   wchar_t **endptr,
   int base
);
__int64 _strtoi64_l(
   const char *nptr,
   char **endptr,
   int base,
   _locale_t locale
);
__int64 _wcstoi64_l(
   const wchar_t *nptr,
   wchar_t **endptr,
   int base,
   _locale_t locale
);
```

Parameters

nptr

Null-terminated string to convert.

endptr

Pointer to character that stops scan.

base

Number base to use.

locale

The locale to use.

Return Value

_strtoi64 returns the value represented in the string *nptr*, except when the representation would cause an overflow, in which case it returns **_I64_MAX** or **_I64_MIN**. The function will return 0 if no conversion can be performed. **_wcstoi64** returns values analogously to **strtoi64**.

_I64_MAX and **_I64_MIN** are defined in LIMITS.H.

If *nptr* is **NULL** or the *base* is nonzero and either less than 2 or greater than 36, **errno** is set to **EINVAL**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

Remarks

The **_strtoi64** function converts *nptr* to an **__int64**. Both functions stop reading the string *nptr* at the first character they cannot recognize as part of a number. This may be the terminating null character, or it may be the first numeric character greater than or equal to *base*. **_wcstoi64** is a wide-character version of **_strtoi64**; its *nptr* argument is a wide-character string. These functions behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined

_tcstoi64	**_strtoi64**	**_strtoi64**	**_wcstoi64**
_tcstoi64_l	**_strtoi64_l**	**_strtoi64_l**	**_wcstoi64_l**

The locale's **LC_NUMERIC** category setting determines recognition of the radix character in *nptr;* for more information, see setlocale. The functions without the _l suffix use the current locale; **_strtoi64_l** and **_wcstoi64_l** are identical to the corresponding function without the **_l** suffix except that they use the locale passed in instead.

If *endptr* is not **NULL**, a pointer to the character that stopped the scan is stored at the location pointed to by *endptr*. If no conversion can be performed (no valid digits were found or an invalid base was specified), the value of *nptr* is stored at the location pointed to by *endptr*.

_strtoi64 expects *nptr* to point to a string of the following form:

[*whitespace*] [{**+** | **–**}] [**0** [{ **x** | **X** }]] [*digits*]

A *whitespace* may consist of space and tab characters, which are ignored; *digits* are one or more decimal digits. The first character that does not fit this form stops the scan. If *base* is between 2 and 36, then it is used as the base of the number. If *base* is 0, the initial characters of the string pointed to by *nptr* are used to determine the base. If the first character is 0 and the second character is not 'x' or 'X', the string is interpreted as an octal integer; otherwise, it is interpreted as a decimal number. If the first character is '0' and the second character is 'x' or 'X', the string is interpreted as a hexadecimal integer. If the first character is '1' through '9', the string is interpreted as a decimal integer. The letters 'a' through 'z' (or 'A' through 'Z') are assigned the values 10 through 35; only letters whose assigned values are less than *base* are permitted.

Requirements

Routine	**Required header**	**Compatibility**
_strtoi64, **_strtoi64_l**	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcstoi64, **_wcstoi64_l**	<stdlib.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

See Also

Reference

Data Conversion
Locale
localeconv
setlocale, _wsetlocale
String to Numeric Value Functions
strtod, _strtod_l, wcstod, _wcstod_l
strtoul, _strtoul_l, wcstoul, _wcstoul_l
atof, _atof_l, _wtof, _wtof_l

strtok, _strtok_l, wcstok, _wcstok_l, _mbstok, _mbstok_l

Find the next token in a string, using the current locale or a specified locale passed in. These functions are deprecated because more secure versions are available; see strtok_s, _strtok_s_l, wcstok_s, _wcstok_s_l, _mbstok_s, _mbstok_s_l.

```
char *strtok(
   char *strToken,
   const char *strDelimit
);
wchar_t *wcstok(
   wchar_t *strToken,
   const wchar_t *strDelimit
);
unsigned char *_mbstok(
   unsigned char*strToken,
   const unsigned char *strDelimit
);
unsigned char *_mbstok(
   unsigned char*strToken,
   const unsigned char *strDelimit,
   _locale_t locale
);
```

Parameters

strToken

String containing token or tokens.

strDelimit

Set of delimiter characters.

locale

Locale to use.

Return Value

Returns a pointer to the next token found in *strToken*. They return **NULL** when no more tokens are found. Each call modifies *strToken* by substituting a NULL character for each delimiter that is encountered.

Remarks

The **strtok** function finds the next token in *strToken*. The set of characters in *strDelimit* specifies possible delimiters of the token to be found in *strToken* on the current call. **wcstok** and **_mbstok** are wide-character and multibyte-character versions of **strtok**. The arguments and return value of **wcstok** are wide-character strings; those of **_mbstok** are multibyte-character strings. These three functions behave identically otherwise.

Security Note
These functions incur a potential threat brought about by a buffer overrun problem. Buffer overrun problems are a frequent method of system attack, resulting in an unwarranted elevation of privilege. For more information, see Avoiding Buffer Overruns.

On the first call to **strtok**, the function skips leading delimiters and returns a pointer to the first token in *strToken*, terminating the token with a null character. More tokens can be broken out of the remainder of *strToken* by a series of calls to **strtok**. Each call to **strtok** modifies *strToken* by inserting a null character after the **token** returned by that call. To read the next token from *strToken*, call **strtok** with a **NULL** value for the *strToken* argument. The **NULL** *strToken* argument causes **strtok** to search for the next token in the modified *strToken*. The *strDelimit* argument can take any value from one call to the next so that the set of delimiters may vary.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Note
Each function uses a static variable for parsing the string into tokens. If multiple or simultaneous calls are made to the same function, a high potential for data corruption and inaccurate results exists. Therefore, do not attempt to call the same function simultaneously for different strings and be aware of calling one of these functions from within a loop where another routine may be called that uses the same function. However, calling this function simultaneously from multiple threads does not have undesirable effects.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcstok	**strtok**	**_mbstok**	**wcstok**
_tcstok	**_strtok_l**	**_mbstok_l**	**_wcstok_l**

Requirements

Routine	Required header	Compatibility
strtok	<string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcstok	<string.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbstok, **_mbstok_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strtok.c
// compile with: /W3
// In this program, a loop uses strtok
// to print all the tokens (separated by commas
// or blanks) in the string named "string".
//
#include <string.h>
#include <stdio.h>

char string[] = "A string\tof ,,tokens\nand some  more tokens";
char seps[]   = " ,\t\n";
char *token;

int main( void )
{
   printf( "Tokens:\n" );

   // Establish string and get the first token:
   token = strtok( string, seps ); // C4996
   // Note: strtok is deprecated; consider using strtok_s instead
   while( token != NULL )
   {
      // While there are tokens in "string"
      printf( " %s\n", token );

      // Get next token:
      token = strtok( NULL, seps ); // C4996
   }
}
```

Output

```
Tokens:
 A
 string
 of
 tokens
 and
 some
 more
 tokens
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

String Manipulation (CRT)
Locale
Interpretation of Multibyte-Character Sequences
strcspn, wcscspn, _mbscspn, _mbscspn_l
strspn, wcsspn, _mbsspn, _mbsspn_l

strtok_s, _strtok_s_l, wcstok_s, _wcstok_s_l, _mbstok_s, _mbstok_s_l

Find the next token in a string, using the current locale or a locale passed in. These are versions of strtok, _strtok_l, wcstok, _wcstok_l, _mbstok, _mbstok_l with security enhancements as described in Security Enhancements in the CRT.

```
char *strtok_s(
   char *strToken,
   const char *strDelimit,
      char **context
);
char *_strtok_s_l(
   char *strToken,
   const char *strDelimit,
      char **context,
   _locale_t locale
);
wchar_t *wcstok_s(
   wchar_t *strToken,
   const wchar_t *strDelimit,
   wchar_t **context
);
wchar_t *_wcstok_s_l(
   wchar_t *strToken,
   const wchar_t *strDelimit,
   wchar_t **context,
   _locale_t locale
);
unsigned char *_mbstok_s(
   unsigned char*strToken,
   const unsigned char *strDelimit,
      char **context
);
unsigned char *_mbstok_s(
   unsigned char*strToken,
   const unsigned char *strDelimit,
      char **context,
   _locale_t locale
);
```

Parameters

strToken

String containing token or tokens.

strDelimit

Set of delimiter characters.

context

Used to store position information between calls to **strtok_s**

locale

Locale to use.

Return Value

Returns a pointer to the next token found in *strToken*. They return **NULL** when no more tokens are found. Each call modifies *strToken* by substituting a **NULL** character for each delimiter that is encountered.

Error Conditions

strToken	***strDelimit***	***context***	**Return value**	***errno***

NULL	any	pointer to a null pointer	**NULL**	**EINVAL**
any	**NULL**	any	**NULL**	**EINVAL**
any	any	**NULL**	**NULL**	**EINVAL**

If *strToken* is **NULL** but context is a pointer to a valid context pointer, there is no error.

Remarks

The **strtok_s** function finds the next token in *strToken*. The set of characters in *strDelimit* specifies possible delimiters of the token to be found in *strToken* on the current call. **wcstok_s** and **_mbstok_s** are wide-character and multibyte-character versions of **strtok_s**. The arguments and return values of **wcstok_s** and **_wcstok_s_l** are wide-character strings; those of **_mbstok_s** and **_mbstok_s_l** are multibyte-character strings. These three functions behave identically otherwise.

This function validates its parameters. If an error condition occurs, as in the Error Conditions table, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **NULL**.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcstok_s	**strtok_s**	**_mbstok_s**	**wcstok_s**
_tcstok_s_l	**_strtok_s_l**	**_mbstok_s_l**	**_wcstok_s_l**

On the first call to **strtok_s** the function skips leading delimiters and returns a pointer to the first token in *strToken*, terminating the token with a null character. More tokens can be broken out of the remainder of *strToken* by a series of calls to **strtok_s**. Each call to **strtok_s** modifies *strToken* by inserting a null character after the token returned by that call. The *context* pointer keeps track of which string is being read and where in the string the next token is to be read. To read the next token from *strToken*, call **strtok_s** with a **NULL** value for the *strToken* argument, and pass the same *context* parameter. The **NULL** *strToken* argument causes **strtok_s** to search for the next token in the modified *strToken*. The *strDelimit* argument can take any value from one call to the next so that the set of delimiters may vary.

Since the *context* parameter supersedes the static buffers used in **strtok** and **_strtok_l**, it is possible to parse two strings simultaneously in the same thread.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Requirements

Routine	Required header	Compatibility
strtok_s	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_strtok_s_l	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcstok_s, **_wcstok_s_l**	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbstok_s, **_mbstok_s_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strtok_s.c
// In this program, a loop uses strtok_s
// to print all the tokens (separated by commas
// or blanks) in two strings at the same time.
//

#include <string.h>
#include <stdio.h>

char string1[] =
    "A string\tof ,,tokens\nand some  more tokens";
char string2[] =
    "Another string\n\tparsed at the same time.";
char seps[]   = " ,\t\n";
char *token1,
     *token2,
     *next_token1,
     *next_token2;

int main( void )
{
    printf( "Tokens:\n" );

    // Establish string and get the first token:
    token1 = strtok_s( string1, seps, &next_token1);
    token2 = strtok_s ( string2, seps, &next_token2);

    // While there are tokens in "string1" or "string2"
    while ((token1 != NULL) || (token2 != NULL))
    {
        // Get next token:
        if (token1 != NULL)
        {
            printf( " %s\n", token1 );
            token1 = strtok_s( NULL, seps, &next_token1);
        }
        if (token2 != NULL)
        {
            printf("        %s\n", token2 );
            token2 = strtok_s (NULL, seps, &next_token2);
        }
    }
}
```

Output

```
Tokens:
 A
        Another
 string
        string
 of
        parsed
 tokens
        at
 and
        the
 some
        same
 more
        time.
 tokens
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

String Manipulation (CRT)

Locale

Interpretation of Multibyte-Character Sequences

strcspn, wcscspn, _mbscspn, _mbscspn_l

strspn, wcsspn, _mbsspn, _mbsspn_l

strtol, wcstol, _strtol_l, _wcstol_l

Convert strings to a long-integer value.

```
long strtol(
   const char *nptr,
   char **endptr,
   int base
);
long wcstol(
   const wchar_t *nptr,
   wchar_t **endptr,
   int base
);
long _strtol_l(
   const char *nptr,
   char **endptr,
   int base,
   _locale_t locale
);
long _wcstol_l(
   const wchar_t *nptr,
   wchar_t **endptr,
   int base,
   _locale_t locale
);
```

Parameters

nptr

Null-terminated string to convert.

endptr

Pointer to character that stops scan.

base

Number base to use.

locale

Locale to use.

Return Value

strtol returns the value represented in the string *nptr*, except when the representation would cause an overflow, in which case it returns **LONG_MAX** or **LONG_MIN**. **strtol** returns 0 if no conversion can be performed. **wcstol** returns values analogously to **strtol**. For both functions, **errno** is set to **ERANGE** if overflow or underflow occurs.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these and other return codes.

Remarks

The **strtol** function converts *nptr* to a **long**. **strtol** stops reading the string *nptr* at the first character it cannot recognize as part of a number. This may be the terminating null character, or it may be the first numeric character greater than or equal to *base*.

wcstol is a wide-character version of **strtol**; its *nptr* argument is a wide-character string. These functions behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcstol	**strtol**	**strtol**	**wcstol**
_tcstol_l	**_strtol_l**	**_strtol_l**	**_wcstol_l**

The current locale's **LC_NUMERIC** category setting determines recognition of the radix character in *nptr;* for more information, see setlocale. The functions without the **_l** suffix use the current locale; **_strtol_l** and **_wcstol_l** are identical to the corresponding functions without the **_l** suffix except that they use the locale passed in instead.

If *endptr* is not **NULL**, a pointer to the character that stopped the scan is stored at the location pointed to by *endptr*. If no conversion can be performed (no valid digits were found or an invalid base was specified), the value of *nptr* is stored at the location pointed to by *endptr*.

strtol expects *nptr* to point to a string of the following form:

[**whitespace**] [{**+** | **–**}] [**0** [{ **x** | **X** }]] [**digits**]

A **whitespace** may consist of space and tab characters, which are ignored; *digits* are one or more decimal digits. The first character that does not fit this form stops the scan. If *base* is between 2 and 36, then it is used as the base of the number. If *base* is 0, the initial characters of the string pointed to by *nptr* are used to determine the base. If the first character is 0 and the second character is not 'x' or 'X', the string is interpreted as an octal integer; otherwise, it is interpreted as a decimal number. If the first character is '0' and the second character is 'x' or 'X', the string is interpreted as a hexadecimal integer. If the first character is '1' through '9', the string is interpreted as a decimal integer. The letters 'a' through 'z' (or 'A' through 'Z') are assigned the values 10 through 35; only letters whose assigned values are less than *base* are permitted.

Requirements

Routine	**Required header**	**Compatibility**
strtol	<stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcstol	<stdlib.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_strtol_l	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for strtod.

.NET Framework Equivalent

System::Convert::ToInt64

See Also

Reference

Data Conversion
Locale
localeconv
setlocale, _wsetlocale
String to Numeric Value Functions
strtod, _strtod_l, wcstod, _wcstod_l
strtoul, _strtoul_l, wcstoul, _wcstoul_l
atof, _atof_l, _wtof, _wtof_l

strtoul, _strtoul_l, wcstoul, _wcstoul_l

Convert strings to an unsigned long-integer value.

```
unsigned long strtoul(
   const char *nptr,
   char **endptr,
   int base
);
unsigned long _strtoul_l(
   const char *nptr,
   char **endptr,
   int base,
   _locale_t locale
);
unsigned long wcstoul(
   const wchar_t *nptr,
   wchar_t **endptr,
   int base
);
unsigned long _wcstoul_l(
   const wchar_t *nptr,
   wchar_t **endptr,
   int base,
   _locale_t locale
);
```

Parameters

nptr

Null-terminated string to convert.

endptr

Pointer to character that stops scan.

base

Number base to use.

locale

Locale to use.

Return Value

strtoul returns the converted value, if any, or **ULONG_MAX** on overflow. **strtoul** returns 0 if no conversion can be performed. **wcstoul** returns values analogously to **strtoul**. For both functions, **errno** is set to **ERANGE** if overflow or underflow occurs.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on this, and other, return codes.

Remarks

Each of these functions converts the input string *nptr* to an **unsigned long**.

strtoul stops reading the string *nptr* at the first character it cannot recognize as part of a number. This may be the terminating null character, or it may be the first numeric character greater than or equal to *base*. The **LC_NUMERIC** category setting of the locale determines recognition of the radix character in *nptr*; for more information, see setlocale. **strtoul** and **wcstoul** use the current locale; **_strtoul_l** and **_wcstoul_l** are identical except that they use the locale passed in instead.

If *endptr* is not **NULL**, a pointer to the character that stopped the scan is stored at the location pointed to by *endptr*. If no conversion can be performed (no valid digits were found or an invalid base was specified), the value of *nptr* is stored at the location pointed to by *endptr*.

wcstoul is a wide-character version of **strtoul**; its *nptr* argument is a wide-character string. Otherwise these functions behave identically.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcstoul	**strtoul**	**strtoul**	**wcstoul**
_tcstoul_l	**strtoul_l**	**_strtoul_l**	**_wcstoul_l**

strtoul expects *nptr* to point to a string of the following form:

[**whitespace**] [{**+** | **–**}] [**0** [{ **x** | **X** }]] [**digits**]

A **whitespace** may consist of space and tab characters, which are ignored; **digits** are one or more decimal digits. The first character that does not fit this form stops the scan. If *base* is between 2 and 36, then it is used as the base of the number. If *base* is 0, the initial characters of the string pointed to by *nptr* are used to determine the base. If the first character is 0 and the second character is not 'x' or 'X', the string is interpreted as an octal integer; otherwise, it is interpreted as a decimal number. If the first character is '0' and the second character is 'x' or 'X', the string is interpreted as a hexadecimal integer. If the first character is '1' through '9', the string is interpreted as a decimal integer. The letters 'a' through 'z' (or 'A' through 'Z') are assigned the values 10 through 35; only letters whose assigned values are less than *base* are permitted. **strtoul** allows a plus (**+**) or minus (**–**) sign prefix; a leading minus sign indicates that the return value is negated.

Requirements

Routine	Required header	Compatibility
strtoul	<stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcstoul	<stdlib.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_strtoul_l	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcstoul_l	<stdlib.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for strtod.

.NET Framework Equivalent

System::Convert::ToUInt64

See Also

Reference

Data Conversion
Locale
localeconv
setlocale, _wsetlocale
String to Numeric Value Functions
strtod, _strtod_l, wcstod, _wcstod_l
strtol, wcstol, _strtol_l, _wcstol_l
atof, _atof_l, _wtof, _wtof_l

_strtoui64, _wcstoui64, _strtoui64_l, _wcstoui64_l

Convert a string to an unsigned **__int64** value.

```
unsigned __int64 _strtoui64(
   const char *nptr,
   char **endptr,
   int base
);
unsigned __int64 _wcstoui64(
   const wchar_t *nptr,
   wchar_t **endptr,
   int base
);
unsigned __int64 _strtoui64_l(
   const char *nptr,
   char **endptr,
   int base,
   _locale_t locale
);
unsigned __int64 _wcstoui64(
   const wchar_t *nptr,
   wchar_t **endptr,
   int base,
   _locale_t locale
);
```

Parameters

nptr

Null-terminated string to convert.

endptr

Pointer to character that stops scan.

base

Number base to use.

locale

Locale to use.

Return Value

_strtoui64 returns the value represented in the string *nptr*, except when the representation would cause an overflow, in which case it returns **_UI64_MAX**. **_strtoui64** will return 0 if no conversion can be performed.

_UI64_MAX is defined in LIMITS.H.

If *nptr* is **NULL** or the *base* is nonzero and either less than 2 or greater than 36, **errno** is set to **EINVAL**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

Remarks

The **_strtoui64** function converts *nptr* to an **unsigned __int64**. **_wcstoui64** is a wide-character version of **_strtoui64**; its *nptr* argument is a wide-character string. Otherwise these functions behave identically.

Both functions stop reading the string *nptr* at the first character they cannot recognize as part of a number. This may be the terminating null character, or it may be the first numeric character greater than or equal to *base*.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcstoui64	**_strtoui64**	**_strtoui64**	**_wstrtoui64**

_tcstoui64_l	_strtoui64_l	_strtoui64_l	_wstrtoui64_l

The current locale's **LC_NUMERIC** category setting determines recognition of the radix character in *nptr*; for more information, see setlocale. The functions without the _l suffix use the current locale; **_strtoui64_l** and **_wcstoui64_l** are identical to the corresponding functions without the **_l** suffix except that they use the locale passed in instead.

If *endptr* is not **NULL**, a pointer to the character that stopped the scan is stored at the location pointed to by *endptr*. If no conversion can be performed (no valid digits were found or an invalid base was specified), the value of *nptr* is stored at the location pointed to by *endptr*.

_strtoui64 expects *nptr* to point to a string of the following form:

[**whitespace**] [{**+** | **–**}] [**0** [{ **x** | **X** }]] [**digits**]

A **whitespace** may consist of space and tab characters, which are ignored; **digits** are one or more decimal digits. The first character that does not fit this form stops the scan. If *base* is between 2 and 36, then it is used as the base of the number. If *base* is 0, the initial characters of the string pointed to by *nptr* are used to determine the base. If the first character is 0 and the second character is not 'x' or 'X', the string is interpreted as an octal integer; otherwise, it is interpreted as a decimal number. If the first character is '0' and the second character is 'x' or 'X', the string is interpreted as a hexadecimal integer. If the first character is '1' through '9', the string is interpreted as a decimal integer. The letters 'a' through 'z' (or 'A' through 'Z') are assigned the values 10 through 35; only letters whose assigned values are less than *base* are permitted.

Requirements

Routine	Required header	Compatibility
_strtoui64	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcstoui64	<stdlib.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_strtoui64_l	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcstoui64_l	<stdlib.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_strtoui64.c
#include <stdio.h>

unsigned __int64 atoui64(const char *szUnsignedInt) {
   return _strtoui64(szUnsignedInt, NULL, 10);
}

int main() {
   unsigned __int64 u = atoui64("18446744073709551615");
   printf( "u = %I64u\n", u );
}
```

Output

```
u = 18446744073709551615
```

See Also

Reference

Data Conversion

Locale

localeconv

setlocale, _wsetlocale
String to Numeric Value Functions
strtod, _strtod_l, wcstod, _wcstod_l
strtoul, _strtoul_l, wcstoul, _wcstoul_l
atof, _atof_l, _wtof, _wtof_l

strupr, wcsupr

These POSIX functions are deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _strupr, _strupr_l, _mbsupr, _mbsupr_l, _wcsupr_l, _wcsupr or security-enhanced _strupr_s, _strupr_s_l, _mbsupr_s, _mbsupr_s_l, _wcsupr_s, _wcsupr_s_l instead.

_strupr, _strupr_l, _mbsupr, _mbsupr_l, _wcsupr_l, _wcsupr

Convert a string to uppercase. These functions are deprecated because more secure versions are available; see _strupr_s, _strupr_s_l, _mbsupr_s, _mbsupr_s_l, _wcsupr_s, _wcsupr_s_l.

```
char *_strupr(
   char *str
);
wchar_t *_wcsupr(
   wchar_t *str
);
unsigned char *_mbsupr(
   unsigned char *str
);
char *_strupr_l(
   char *str,
   _locale_t locale
);
wchar_t *_wcsupr_l(
   wchar_t *str,
   _locale_t locale
);
unsigned char *_mbsupr_l(
   unsigned char *str,
   _locale_t locale
);
template <size_t size>
char *_strupr(
   char (&str)[size]
); // C++ only
template <size_t size>
wchar_t *_wcsupr(
   wchar_t (&str)[size]
); // C++ only
template <size_t size>
unsigned char *_mbsupr(
   unsigned char (&str)[size]
); // C++ only
template <size_t size>
char *_strupr_l(
   char (&str)[size],
   _locale_t locale
); // C++ only
template <size_t size>
wchar_t *_wcsupr_l(
   wchar_t (&str)[size],
   _locale_t locale
); // C++ only
template <size_t size>
unsigned char *_mbsupr_l(
   unsigned char (&str)[size],
   _locale_t locale
); // C++ only
```

Parameters

str

String to capitalize.

locale

The locale to use.

Return Value

Returns a pointer to the altered string. Because the modification is done in place, the pointer returned is the same as the pointer passed as the input argument. No return value is reserved to indicate an error.

Remarks

The **_strupr** function converts, in place, each lowercase letter in *str* to uppercase. The conversion is determined by the **LC_CTYPE** category setting of the locale. Other characters are not affected. For more information on **LC_CTYPE**, see setlocale. The versions of these functions without the **_l** suffix use the current locale; the versions with the **_l** suffix are identical except that they use the locale passed in instead.

_wcsupr and **_mbsupr** are wide-character and multibyte-character versions of **_strupr**. The argument and return value of **_wcsupr** are wide-character strings; those of **_mbsupr** are multibyte-character strings. These three functions behave identically otherwise.

If *str* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, these functions return the original string and set **errno** to **EINVAL**.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsupr	**_strupr**	**_mbsupr**	**_wcsupr**
_tcsupr_l	**_strupr_l**	**_mbsupr_l**	**_wcsupr_l**

Requirements

Routine	Required header	Compatibility
_strupr, **_strupr_l**	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsupr, **_wcsupr_l**	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_mbsupr, **_mbsupr_l**	<mbstring.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for _strlwr.

.NET Framework Equivalent

System::String::ToUpper

See Also

Reference

Locale

String Manipulation (CRT)

_strlwr, _wcslwr, _mbslwr, _strlwr_l, _wcslwr_l, _mbslwr_l

_strupr_s, _strupr_s_l, _mbsupr_s, _mbsupr_s_l, _wcsupr_s, _wcsupr_s_l

Convert a string to uppercase, using the current locale or a specified locale passed in. These are versions of _strupr, _strupr_l, _mbsupr, _mbsupr_l, _wcsupr_l, _wcsupr with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _strupr_s(
   char *str,
   size_t numberOfElements
);
errno_t _wcsupr_s(
   wchar_t * str,
   size_t numberOfElements
);
errno_t _strupr_s_l(
   char * str,
   size_t numberOfElements,
   _locale_t locale
);
errno_t _wcsupr_s_l(
   wchar_t * str,
   size_t numberOfElements,
   _locale_t locale
);
errno_t _mbsupr_s(
   unsigned char *str,
   size_t numberOfElements
);
errno_t _mbsupr_s_l(
   unsigned char *str,
   size_t numberOfElements,
   _locale_t locale
);
template <size_t size>
errno_t _strupr_s(
   char (&str)[size]
); // C++ only
template <size_t size>
errno_t _wcsupr_s(
   wchar_t (&str)[size]
); // C++ only
template <size_t size>
errno_t _strupr_s_l(
   char (&str)[size],
   _locale_t locale
); // C++ only
template <size_t size>
errno_t _wcsupr_s_l(
   wchar_t (&str)[size],
   _locale_t locale
); // C++ only
template <size_t size>
errno_t _mbsupr_s(
   unsigned char (&str)[size]
); // C++ only
template <size_t size>
errno_t _mbsupr_s_l(
   unsigned char (&str)[size],
   _locale_t locale
); // C++ only
```

Parameters

str

String to capitalize.

numberOfElements

Size of the buffer.

locale

The locale to use.

Return Value

Zero if successful; a non-zero error code on failure.

These functions validate their parameters. If *str* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, the functions return **EINVAL** and set **errno** to **EINVAL**. If *numberOfElements* is less than the length of the string, the functions return **ERANGE** and set **errno** to **ERANGE**.

Remarks

The **_strupr_s** function converts, in place, each lowercase letter in *str* to uppercase. **_wcsupr_s** is the wide-character version of **_strupr_s**. **_mbsupr_s** is the multi-byte character version of **_strupr_s**.

The conversion is determined by the **LC_CTYPE** category setting of the locale. Other characters are not affected. For more information on **LC_CTYPE**, see setlocale. The versions of these functions without the **_l** suffix use the current locale; the visions with the **_l** suffix are identical except that they use the locale passed in instead.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsupr_s	**_strupr_s**	**_mbsupr_s**	**_wcsupr_s**
_tcsupr_s_l	**_strupr_s_l**	**_mbsupr_s_l**	**_wcsupr_s_l**

Requirements

Routine	Required header	Compatibility
_strupr_s, **_strupr_s_l**	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsupr_s, **_wcsupr_s_l**, **_mbsupr_s**, **_mbsupr_s_l**	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for _strlwr_s, _strlwr_s_l, _mbslwr_s, _mbslwr_s_l, _wcslwr_s, _wcslwr_s_l .

.NET Framework Equivalent

System::String::ToUpper

See Also

Reference

Locale

Interpretation of Multibyte-Character Sequences

String Manipulation (CRT)

_strlwr_s, _strlwr_s_l, _mbslwr_s, _mbslwr_s_l, _wcslwr_s, _wcslwr_s_l

strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

Transform a string based on locale-specific information.

```
size_t strxfrm(
   char *strDest,
   const char *strSource,
   size_t count
);
size_t wcsxfrm(
   wchar_t *strDest,
   const wchar_t *strSource,
   size_t count
);
size_t _strxfrm_l(
   char *strDest,
   const char *strSource,
   size_t count,
   _locale_t locale
);
size_t wcsxfrm(
   wchar_t *strDest,
   const wchar_t *strSource,
   size_t count,
   _locale_t locale
);
```

Parameters

strDest

Destination string.

strSource

Source string.

count

Maximum number of characters to place in *strDest*.

locale

The locale to use.

Return Value

Returns the length of the transformed string, not counting the terminating null character. If the return value is greater than or equal to *count*, the content of *strDest* is unpredictable. On an error, each function sets **errno** and returns **INT_MAX**. For an invalid character, **errno** is set to **EILSEQ**.

Remarks

The **strxfrm** function transforms the string pointed to by *strSource* into a new collated form that is stored in *strDest*. No more than *count* characters, including the null character, are transformed and placed into the resulting string. The transformation is made using the locale's **LC_COLLATE** category setting. For more information on **LC_COLLATE**, see setlocale. **strxfrm** uses the current locale for its locale-dependent behavior; **_strxfrm_l** is identical except that it uses the locale passed in instead of the current locale.

After the transformation, a call to **strcmp** with the two transformed strings yields results identical to those of a call to **strcoll** applied to the original two strings. As with **strcoll** and **stricoll**, **strxfrm** automatically handles multibyte-character strings as appropriate.

wcsxfrm is a wide-character version of **strxfrm**; the string arguments of **wcsxfrm** are wide-character pointers. For **wcsxfrm**, after the string transformation, a call to **wcscmp** with the two transformed strings yields results identical to those of a call to **wcscoll** applied to the original two strings. **wcsxfrm** and **strxfrm** behave identically otherwise. **wcsxfrm** uses the current locale for its locale-dependent behavior; **_wcsxfrm_l** uses the locale passed in instead of the current locale.

These functions validate their parameters. If *strSource* is a null pointer, or *strDest* is a NULL pointer (unless count is zero), or if *count* is greater than **INT_MAX**, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **INT_MAX**.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tcsxfrm	**strxfrm**	**strxfrm**	**wcsxfrm**
_tcsxfrm_l	**_strxfrm_l**	**_strxfrm_l**	**_wcsxfrm_l**

In the "C" locale, the order of the characters in the character set (ASCII character set) is the same as the lexicographic order of the characters. However, in other locales, the order of characters in the character set may differ from the lexicographic character order. For example, in certain European locales, the character 'a' (value 0x61) precedes the character 'ä' (value 0xE4) in the character set, but the character 'ä' precedes the character 'a' lexicographically.

In locales for which the character set and the lexicographic character order differ, use **strxfrm** on the original strings and then **strcmp** on the resulting strings to produce a lexicographic string comparison according to the current locale's **LC_COLLATE** category setting. Thus, to compare two strings lexicographically in the above locale, use **strxfrm** on the original strings, then **strcmp** on the resulting strings. Alternately, you can use **strcoll** rather than **strcmp** on the original strings.

The value of the following expression is the size of the array needed to hold the **strxfrm** transformation of the source string:

```
1 + strxfrm( NULL, string, 0 )
```

In the "C" locale only, **strxfrm** is equivalent to the following:

```
strncpy( _string1, _string2, _count );
return( strlen( _string1 ) );
```

Requirements

Routine	Required header	Compatibility
strxfrm	<string.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
wcsxfrm	<string.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_strxfrm_l	<string.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcsxfrm_l	<string.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Data Conversion
localeconv
setlocale, _wsetlocale
Locale
String Manipulation (CRT)
strcoll Functions
strcmp, wcscmp, _mbscmp

strncmp, wcsncmp, _mbsncmp, _mbsncmp_l

swab

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _swab instead.

_swab

Swaps bytes.

```
void _swab(
   char *src,
   char *dest,
   int n
);
```

Parameters

src

Data to be copied and swapped.

dest

Storage location for swapped data.

n

Number of bytes to be copied and swapped.

Remarks

If *n* is even, the **_swab** function copies *n* bytes from *src*, swaps each pair of adjacent bytes, and stores the result at *dest*. If *n* is odd, **_swab** copies and swaps the first *n-1* bytes of *src*. **_swab** is typically used to prepare binary data for transfer to a machine that uses a different byte order.

Requirements

Routine	Required header	Compatibility
_swab	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_swab.c

#include <stdlib.h>
#include <stdio.h>

char from[] = "BADCFEHGJILKNMPORQTSVUXWZY";
char to[] =   ".........................";

int main()
{
    printf( "Before: %s\n        %s\n\n", from, to );
    _swab( from, to, sizeof( from ) );
    printf( "After:  %s\n        %s\n\n", from, to );
}
```

Output

```
Before: BADCFEHGJILKNMPORQTSVUXWZY
        .........................

After:  BADCFEHGJILKNMPORQTSVUXWZY
        ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Buffer Manipulation

system, _wsystem

Execute a command.

```
int system(
   const char *command
);
int _wsystem(
   const wchar_t *command
);
```

Parameters

command

Command to be executed.

Return Value

If *command* is **NULL** and the command interpreter is found, returns a nonzero value. If the command interpreter is not found, returns 0 and sets **errno** to **ENOENT**. If *command* is not **NULL**, **system** returns the value that is returned by the command interpreter. It returns the value 0 only if the command interpreter returns the value 0. A return value of – 1 indicates an error, and **errno** is set to one of the following values:

E2BIG

Argument list (which is system dependent) is too big.

ENOENT

Command interpreter cannot be found.

ENOEXEC

Command-interpreter file has invalid format and is not executable.

ENOMEM

Not enough memory is available to execute command; or available memory has been corrupted; or invalid block exists, indicating that process making call was not allocated properly.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

Remarks

The **system** function passes *command* to the command interpreter, which executes the string as an operating-system command. **system** refers to the **COMSPEC** and **PATH** environment variables that locate the command-interpreter file (the file named CMD.EXE in Windows NT and later). If *command* is NULL, the function simply checks to see whether the command interpreter exists.

You must explicitly flush (using **fflush** or **_flushall**) or close any stream before calling **system**.

_wsystem is a wide-character version of **system**; the *command* argument to **_wsystem** is a wide-character string. These functions behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tsystem	**system**	**system**	**_wsystem**

Requirements

Routine	Required header	Compatibility
system	<process.h> or <stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_wsystem	<process.h> or <stdlib.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

This program uses **system** to TYPE a text file.

```
// crt_system.c

#include <process.h>

int main( void )
{
   system( "type crt_system.txt" );
}
```

Input: crt_system.txt

```
Line one.
Line two.
```

Output

```
Line one.
Line two.
```

.NET Framework Equivalent

- System::Diagnostics::ProcessStartInfo Class
- System::Diagnostics::Process Class

See Also

Reference

Process and Environment Control
_exec, _wexec Functions
exit, _exit
_flushall
_spawn, _wspawn Functions

tan, tanf, tanh, tanhf

Calculate the tangent (**tan** or **tanf**) or hyperbolic tangent (**tanh** or **tanhf**).

```
double tan(
   double x
);
float tan(
   float x
);  // C++ only
long double tan(
   long double x
);  // C++ only
float tanf(
   float x
);
double tanh(
   double x
);
float tanh(
   float x
);  // C++ only
long double tanh(
   long double x
);  // C++ only
float tanhf(
   float x
);
```

Parameters

x

Angle in radians.

Return Value

tan returns the tangent of *x*. If *x* is greater than or equal to 263, or less than or equal to –263, a loss of significance in the result occurs.

Input	SEH Exception	Matherr Exception
± QNAN,IND	none	_DOMAIN
± ∞ (**tan**, **tanf**)	**INVALID**	_DOMAIN

tanh returns the hyperbolic tangent of *x*. There is no error return.

Remarks

C++ allows overloading, so users can call overloads of **tan** and **tanh** that take float or long double types. In a C program, the **tan** and **tanh** functions always take and return double.

Requirements

Routine	Required header	Compatibility
tan, **tanf**, **tanh**, **tanhf**	<math.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_tan.c
// This program displays the tangent of pi / 4
// and the hyperbolic tangent of the result.
//

#include <math.h>
#include <stdio.h>

int main( void )
{
   double pi = 3.1415926535;
   double x, y;

   x = tan( pi / 4 );
   y = tanh( x );
   printf( "tan( %f ) = %f\n", pi/4, x );
   printf( "tanh( %f ) = %f\n", x, y );
}
```

Output

```
tan( 0.785398 ) = 1.000000
tanh( 1.000000 ) = 0.761594
```

.NET Framework Equivalent

- System::Math::Tan
- System::Math::Tanh

See Also

Reference

Floating-Point Support
Long Double
acos, acosf
asin, asinf
atan, atanf, atan2, atan2f
cos, cosf, cosh, coshf
sin, sinf, sinh, sinhf

tell

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _tell instead.

_tell, _telli64

Get the position of the file pointer.

```
long _tell(
   int handle
);
__int64 _telli64(
   int handle
);
```

Parameters

handle

File descriptor referring to open file.

Return Value

The current position of the file pointer. On devices incapable of seeking, the return value is undefined.

A return value of –1L indicates an error. If *handle* is an invalid file descriptor, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EBADF** and return -1L.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on this, and other, return codes.

Remarks

The **_tell** function gets the current position of the file pointer (if any) associated with the *handle* argument. The position is expressed as the number of bytes from the beginning of the file. For the **_telli64** function, this value is expressed as a 64-bit integer.

Requirements

Routine	Required header	Compatibility
_tell, **_telli64**	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_tell.c
// This program uses _tell to tell the
// file pointer position after a file read.
//

#include <io.h>
#include <stdio.h>
#include <fcntl.h>
#include <share.h>
#include <string.h>

int main( void )
{
   int  fh;
   char buffer[500];

   if ( _sopen_s( &fh, "crt_tell.txt", _O_RDONLY, _SH_DENYNO, 0) )
   {
      char buff[50];
      _strerror_s( buff, sizeof(buff), NULL );
      printf( buff );
      exit( -1 );
   }
```

```
    if( _read( fh, buffer, 500 ) > 0 )
       printf( "Current file position is: %d\n", _tell( fh ) );
    _close( fh );
 }
```

Input: crt_tell.txt

```
 Line one.
 Line two.
```

Output

```
Current file position is: 20
```

See Also

Reference

Low-Level I/O

ftell, _ftelli64

_lseek, _lseeki64

tempnam

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _tempnam instead.

_tempnam, _wtempnam, tmpnam, _wtmpnam

Generate names you can use to create temporary files. These functions are deprecated because more secure versions are available; see tmpnam_s, _wtmpnam_s.

```
char *_tempnam(
   const char *dir,
   const char *prefix
);
wchar_t *_wtempnam(
   const wchar_t *dir,
   const wchar_t *prefix
);
char *tmpnam(
   char *str
);
wchar_t *_wtmpnam(
   wchar_t *str
);
```

Parameters

prefix

The string that will be pre-pended to names returned by **_tempnam**.

dir

The path used in the file name if there is no TMP environment variable, or if TMP is not a valid directory.

str

Pointer that will hold the generated name and will be identical to the name returned by the function. This is a convenient way to save the generated name.

Return Value

Each of these functions returns a pointer to the name generated or **NULL** if there is a failure. Failure can occur if you attempt more than **TMP_MAX** (see STDIO.H) calls with **tmpnam** or if you use **_tempnam** and there is an invalid directory name specified in the TMP environment variable and in the *dir* parameter.

Note
The pointers returned by **tmpnam** and **_wtmpnam** point to internal static buffers. free should not be called to deallocate th ose pointers. **free** needs to be called for pointers allocated by **_tempnam** and **_wtempnam**.

Remarks

Each of these functions returns the name of a file that does not currently exist. **tmpnam** returns a name unique in the current working directory and **_tempnam** lets you generate a unique name in a directory other than the current one. Note than when a file name is pre-pended with a backslash and no path information, such as \fname21, this indicates that the name is valid for the current working directory.

For **tmpnam**, you can store this generated file name in *str*. If *str* is **NULL**, then **tmpnam** leaves the result in an internal static buffer. Thus any subsequent calls destroy this value. The name generated by **tmpnam** consists of a program-generated file name and, after the first call to **tmpnam**, a file extension of sequential numbers in base 32 (.1-.vvu, when **TMP_MAX** in STDIO.H is 32,767).

_tempnam will generate a unique file name for a directory chosen by the following rules:

- If the TMP environment variable is defined and set to a valid directory name, unique file names will be generated for the directory specified by TMP.
- If the TMP environment variable is not defined or if it is set to the name of a directory that does not exist, **_tempnam** will use the *dir* parameter as the path for which it will generate unique names.

- If the TMP environment variable is not defined or if it is set to the name of a directory that does not exist, and if *dir* is either **NULL** or set to the name of a directory that does not exist, **_tempnam** will use the current working directory to generate unique names. Currently, if both TMP and *dir* specify names of directories that do not exist, the **_tempnam** function call will fail.

The name returned by **_tempnam** will be a concatenation of *prefix* and a sequential number, which will combine to create a unique file name for the specified directory. **_tempnam** generates file names that have no extension. **_tempnam** uses malloc to allocate space for the filename; the program is responsible for freeing this space when it is no longer needed.

_tempnam and **tmpnam** automatically handle multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the OEM code page obtained from the operating system. **_wtempnam** is a wide-character version of **_tempnam**; the arguments and return value of **_wtempnam** are wide-character strings. **_wtempnam** and **_tempnam** behave identically except that **_wtempnam** does not handle multibyte-character strings. **_wtmpnam** is a wide-character version of **tmpnam**; the argument and return value of **_wtmpnam** are wide-character strings. **_wtmpnam** and **tmpnam** behave identically except that **_wtmpnam** does not handle multibyte-character strings.

If **_DEBUG** and **_CRTDBG_MAP_ALLOC** are defined, **_tempnam** and **_wtempnam** are replaced by calls to _tempnam_dbg and _wtempnam_dbg.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_ttmpnam	**tmpnam**	**tmpnam**	**_wtmpnam**
_ttempnam	**_tempnam**	**_tempnam**	**_wtempnam**

Requirements

Routine	Required header	Compatibility
_tempnam	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wtempnam, **_wtmpnam**	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
tmpnam	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_tempnam.c
// compile with: /W3
// This program uses tmpnam to create a unique filename in the
// current working directory, then uses _tempnam to create
// a unique filename with a prefix of stq.

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   char* name1 = NULL;
   char* name2 = NULL;

   // Create a temporary filename for the current working directory:
   if( ( name1 = tmpnam( NULL ) ) != NULL ) // C4996
   // Note: tmpnam is deprecated; consider using tmpnam_s instead
      printf( "%s is safe to use as a temporary file.\n", name1 );
   else
      printf( "Cannot create a unique filename\n" );
```

```
    // Create a temporary filename in temporary directory with the
    // prefix "stq". The actual destination directory may vary
    // depending on the state of the TMP environment variable and
    // the global variable P_tmpdir.

    if( ( name2 = _tempnam( "c:\\tmp", "stq" ) ) != NULL )
       printf( "%s is safe to use as a temporary file.\n", name2 );
    else
       printf( "Cannot create a unique filename\n" );

    // When name2 is no longer needed :
    if(name2)
      free(name2);

  }
```

Sample Output

```
\s1gk. is safe to use as a temporary file.
C:\DOCUME~1\user\LOCALS~1\Temp\2\stq2 is safe to use as a temporary file.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O
_getmbcp
malloc
_setmbcp
tmpfile
tmpfile_s

_tempnam_dbg, _wtempnam_dbg

Function versions of _tempnam, _wtempnam, tmpnam, _wtmpnam that use the debug version of **malloc, _malloc_dbg**.

```
char *_tempnam_dbg(
   const char *dir,
   const char *prefix,
   int blockType,
   const char *filename,
   int linenumber
);
wchar_t *_wtempnam_dbg(
   const wchar_t *dir,
   const wchar_t *prefix,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

dir

The path used in the file name if there is no TMP environment variable, or if TMP is not a valid directory.

prefix

The string that will be pre-pended to names returned by **_tempnam**.

blockType

Requested type of memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

filename

Pointer to name of source file that requested allocation operation or **NULL**.

linenumber

Line number in source file where allocation operation was requested or **NULL**.

Return Value

Each function returns a pointer to the name generated or **NULL** if there is a failure. Failure can occur if there is an invalid directory name specified in the TMP environment variable and in the *dir* parameter.

Note
free (or **free_dbg**) does need to be called for pointers allocated by **_tempnam_dbg** and **_wtempnam_dbg**.

Remarks

The **_tempnam_dbg** and **_wtempnam_dbg** functions are identical to **_tempnam** and **_wtempnam** except that, when **_DEBUG** is defined, these functions use the debug version of **malloc** and **_malloc_dbg**, to allocate memory if **NULL** is passed as the first parameter. For more information, see _malloc_dbg.

You do not need to call these functions explicitly in most cases. Instead, you can define the flag **_CRTDBG_MAP_ALLOC**. When **_CRTDBG_MAP_ALLOC** is defined, calls to **_tempnam** and **_wtempnam** are remapped to **_tempnam_dbg** and **_wtempnam_dbg**, respectively, with the *blockType* set to **_NORMAL_BLOCK**. Thus, you do not need to call these functions explicitly unless you want to mark the heap blocks as **_CLIENT_BLOCK**. For more information, see Types of Blocks on the Debug Heap.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_ttempnam_dbg	_tempnam_dbg	_tempnam_dbg	_wtempnam_dbg

Requirements

Routine	Required header	Compatibility
_tempnam_dbg, **_wtempnam_dbg**	<crtdbg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

_tempnam, _wtempnam, tmpnam, _wtmpnam

Stream I/O

Concepts

Debug Versions of Heap Allocation Functions

terminate (CRT)

Calls **abort** or a function you specify using **set_terminate**.

```
void terminate( void );
```

Remarks

The **terminate** function is used with C++ exception handling and is called in the following cases:

- A matching catch handler cannot be found for a thrown C++ exception.
- An exception is thrown by a destructor function during stack unwind.
- The stack is corrupted after throwing an exception.

terminate calls **abort** by default. You can change this default by writing your own termination function and calling **set_terminate** with the name of your function as its argument. **terminate** calls the last function given as an argument to **set_terminate**. For more information, see Unhandled C++ Exceptions.

Requirements

Routine	Required header	Compatibility
terminate	<eh.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_terminate.cpp
// compile with: /EHsc
#include <eh.h>
#include <process.h>
#include <iostream>
using namespace std;

void term_func();

int main()
{
    int i = 10, j = 0, result;
    set_terminate( term_func );
    try
    {
        if( j == 0 )
            throw "Divide by zero!";
        else
            result = i/j;
    }
    catch( int )
    {
        cout << "Caught some integer exception.\n";
    }
    cout << "This should never print.\n";
}

void term_func()
{
    cout << "term_func() was called by terminate().\n";

    // ... cleanup tasks performed here
```

```
    // If this function does not exit, abort is called.

    exit(-1);
}
```

Output

```
term_func() was called by terminate().
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Exception Handling Routines
abort
_set_se_translator
set_terminate (CRT)
set_unexpected (CRT)
unexpected (CRT)

time, _time32, _time64

Get the system time.

```
time_t time(
   time_t *timer
);
__time32_t _time32(
   __time32_t *timer
);
__time64_t _time64(
   __time64_t *timer
);
```

Parameters

timer

Pointer to the storage location for time.

Return Value

Return the time as seconds elapsed since midnight, January 1, 1970. There is no error return.

Remarks

The **time** function returns the number of seconds elapsed since midnight (00:00:00), January 1, 1970, Coordinated Universal Time (UTC), according to the system clock. The return value is stored in the location given by *timer*. This parameter may be **NULL**, in which case the return value is not stored.

In Visual C++ 2005, **time** is a wrapper for **_time64** and **time_t** is, by default, equivalent to **__time64_t**. If you need to force the compiler to interpret **time_t** as the old 32-bit **time_t**, you can define **_USE_32BIT_TIME_T**. This is not recommended because your application may fail after January 18, 2038; the use of this macro is not allowed on 64-bit platforms.

Requirements

Routine	Required header	Compatibility
time	<time.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_time32, **_time64**	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_times.c
// compile with: /W3
// This program demonstrates these time and date functions:
//      time         _ftime    ctime_s     asctime_s
//      _localtime64_s    _gmtime64_s    mktime    _tzset
//      _strtime_s     _strdate_s  strftime
//
// Also the global variable:
//      _tzname
//

#include <time.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/timeb.h>
#include <string.h>
```

```
int main()
{
    char tmpbuf[128], timebuf[26], ampm[] = "AM";
    time_t ltime;
    struct _timeb tstruct;
    struct tm today, gmt, xmas = { 0, 0, 12, 25, 11, 93 };
    errno_t err;

    // Set time zone from TZ environment variable. If TZ is not set,
    // the operating system is queried to obtain the default value
    // for the variable.
    //
    _tzset();

    // Display operating system-style date and time.
    _strtime_s( tmpbuf, 128 );
    printf( "OS time:\t\t\t\t%s\n", tmpbuf );
    _strdate_s( tmpbuf, 128 );
    printf( "OS date:\t\t\t\t%s\n", tmpbuf );

    // Get UNIX-style time and display as number and string.
    time( &ltime );
    printf( "Time in seconds since UTC 1/1/70:\t%ld\n", ltime );
    err = ctime_s(timebuf, 26, &ltime);
    if (err)
    {
       printf("ctime_s failed due to an invalid argument.");
       exit(1);
    }
    printf( "UNIX time and date:\t\t\t%s", timebuf );

    // Display UTC.
    err = _gmtime64_s( &gmt, &ltime );
    if (err)
    {
       printf("_gmtime64_s failed due to an invalid argument.");
    }
    err = asctime_s(timebuf, 26, &gmt);
    if (err)
    {
       printf("asctime_s failed due to an invalid argument.");
       exit(1);
    }
    printf( "Coordinated universal time:\t\t%s", timebuf );

    // Convert to time structure and adjust for PM if necessary.
    err = _localtime64_s( &today, &ltime );
    if (err)
    {
       printf("_localtime64_s failed due to an invalid argument.");
       exit(1);
    }
    if( today.tm_hour >= 12 )
    {
   strcpy_s( ampm, sizeof(ampm), "PM" );
   today.tm_hour -= 12;
    }
    if( today.tm_hour == 0 )  // Adjust if midnight hour.
   today.tm_hour = 12;

    // Convert today into an ASCII string
    err = asctime_s(timebuf, 26, &today);
    if (err)
    {
       printf("asctime_s failed due to an invalid argument.");
       exit(1);
    }
```

```
      // Note how pointer addition is used to skip the first 11
      // characters and printf is used to trim off terminating
      // characters.
      //
      printf( "12-hour time:\t\t\t\t%.8s %s\n",
         timebuf + 11, ampm );

      // Print additional time information.
      _ftime( &tstruct ); // C4996
      // Note: _ftime is deprecated; consider using _ftime_s instead
      printf( "Plus milliseconds:\t\t\t%u\n", tstruct.millitm );
      printf( "Zone difference in hours from UTC:\t%u\n",
               tstruct.timezone/60 );
      printf( "Time zone name:\t\t\t\t%s\n", _tzname[0] ); //C4996
      // Note: _tzname is deprecated; consider using _get_tzname
      printf( "Daylight savings:\t\t\t%s\n",
               tstruct.dstflag ? "YES" : "NO" );

      // Make time for noon on Christmas, 1993.
      if( mktime( &xmas ) != (time_t)-1 )
      {
         err = asctime_s(timebuf, 26, &xmas);
         if (err)
         {
            printf("asctime_s failed due to an invalid argument.");
            exit(1);
         }
         printf( "Christmas\t\t\t\t%s\n", timebuf );
      }

      // Use time structure to build a customized time string.
      err = _localtime64_s( &today, &ltime );
      if (err)
      {
          printf(" _localtime64_s failed due to invalid arguments.");
          exit(1);
      }

      // Use strftime to build a customized time string.
      strftime( tmpbuf, 128,
           "Today is %A, day %d of %B in the year %Y.\n", &today );
      printf( tmpbuf );
  }
```

Sample Output

```
OS time:            13:51:23
OS date:            04/25/03
Time in seconds since UTC 1/1/70:   1051303883
UNIX time and date:         Fri Apr 25 13:51:23 2003
Coordinated universal time:      Fri Apr 25 20:51:23 2003
12-hour time:            01:51:23 PM
Plus milliseconds:         552
Zone difference in hours from UTC:   8
Time zone name:            Pacific Standard Time
Daylight savings:         YES
Christmas            Sat Dec 25 12:00:00 1993

Today is Friday, day 25 of April in the year 2003.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Time Management

asctime, _wasctime

asctime_s, _wasctime_s
_ftime, _ftime32, _ftime64
gmtime, _gmtime32, _gmtime64
_gmtime_s, _gmtime32_s, _gmtime64_s
localtime, _localtime32, _localtime64
localtime_s, _localtime32_s, _localtime64_s
_utime, _utime32 _utime64, _wutime, _wutime32, _wutime64

tmpfile

Creates a temporary file. This function is deprecated because a more secure version is available; see tmpfile_s.

```
FILE *tmpfile( void );
```

Return Value

If successful, **tmpfile** returns a stream pointer. Otherwise, it returns a **NULL** pointer.

Remarks

The **tmpfile** function creates a temporary file and returns a pointer to that stream. The temporary file is created in the root directory. To create a temporary file in a directory other than the root, use tmpnam or tempnam in conjunction with fopen.

If the file cannot be opened, **tmpfile** returns a **NULL** pointer. This temporary file is automatically deleted when the file is closed, when the program terminates normally, or when **_rmtmp** is called, assuming that the current working directory does not change. The temporary file is opened in **w+b** (binary read/write) mode.

Failure can occur if you attempt more than TMP_MAX (see STDIO.H) calls with **tmpfile**.

Requirements

Routine	Required header	Compatibility
tmpfile	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_tmpfile.c
// compile with: /W3
// This program uses tmpfile to create a
// temporary file, then deletes this file with _rmtmp.
#include <stdio.h>

int main( void )
{
   FILE *stream;
   char tempstring[] = "String to be written";
   int  i;

   // Create temporary files.
   for( i = 1; i <= 3; i++ )
   {
      if( (stream = tmpfile()) == NULL ) // C4996
      // Note: tmpfile is deprecated; consider using tmpfile_s instead
         perror( "Could not open new temporary file\n" );
      else
         printf( "Temporary file %d was created\n", i );
   }

   // Remove temporary files.
   printf( "%d temporary files deleted\n", _rmtmp() );
}
```

Output

```
Temporary file 1 was created
Temporary file 2 was created
Temporary file 3 was created
3 temporary files deleted
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O

_rmtmp

_tempnam, _wtempnam, tmpnam, _wtmpnam

tmpfile_s

Creates a temporary file. A version of tmpfile with security enhancements as described in Security Enhancements in the CRT.

```
errno_t tmpfile(
   FILE** pFilePtr
);
```

Parameters

[out] *pFilePtr*

The address of a pointer to store the address of the generated pointer to a stream.

Return Value

Returns 0 if successful, an error code on failure.

Error Conditions

pFilePtr	**Return Value**	**Contents of *pFilePtr***
NULL	**EINVAL**	not changed

If the above parameter validation error occurs, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the return value is **EINVAL**.

Remarks

The **tmpfile** function creates a temporary file and puts a pointer to that stream in the *pFilePtr* argument. The temporary file is created in the root directory. To create a temporary file in a directory other than the root, use tmpnam or tempnam in conjunction with fopen.

If the file cannot be opened, **tmpfile_s** writes **NULL** to the *pFilePtr* parameter. This temporary file is automatically deleted when the file is closed, when the program terminates normally, or when **_rmtmp** is called, assuming that the current working directory does not change. The temporary file is opened in **w+b** (binary read/write) mode.

Failure can occur if you attempt more than **TMP_MAX_S** (see STDIO.H) calls with **tmpfile_s.**

Requirements

Routine	**Required header**	**Compatibility**
tmpfile_s	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_tmpfile_s.c
// This program uses tmpfile to create a
// temporary file, then deletes this file with _rmtmp.
//

#include <stdio.h>

int main( void )
{
   FILE *stream;
   char tempstring[] = "String to be written";
   int  i;
   errno_t err;

   // Create temporary files.
   for( i = 1; i <= 3; i++ )
   {
```

```
        err = tmpfile_s(&stream);
        if( err )
           perror( "Could not open new temporary file\n" );
        else
           printf( "Temporary file %d was created\n", i );
     }

     // Remove temporary files.
     printf( "%d temporary files deleted\n", _rmtmp() );
  }
```

Output

```
Temporary file 1 was created
Temporary file 2 was created
Temporary file 3 was created
3 temporary files deleted
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O

_rmtmp

_tempnam, _wtempnam, tmpnam, _wtmpnam

tmpnam_s, _wtmpnam_s

Generate names you can use to create temporary files. These are versions of tmpnam and _wtmpnam with security enhancements as described in Security Enhancements in the CRT.

```
errno_t tmpnam_s(
   char * str,
   size_t sizeInChars
);
errno_t _wtmpnam_s(
   wchar_t *str,
   size_t sizeInChars
);
template <size_t size>
errno_t tmpnam_s(
   char (&str)[size]
); // C++ only
template <size_t size>
errno_t _wtmpnam_s(
   wchar_t (&str)[size]
); // C++ only
```

Parameters

[out] *str*

Pointer that will hold the generated name.

[in] *sizeInChars*

The size of the buffer in characters.

Return Value

Both of these functions return 0 if successful or an error number on failure.

Error Conditions

str	*sizeInChars*	**Return Value**	**Contents of** *str*
NULL	any	**EINVAL**	not modified
not **NULL** (points to valid memory)	too short	**ERANGE**	not modified

If *str* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **EINVAL**.

Remarks

Each of these functions returns the name of a file that does not currently exist. **tmpnam_s** returns a name unique in the current working directory. Note than when a file name is pre-pended with a backslash and no path information, such as \fname21, this indicates that the name is valid for the current working directory.

For **tmpnam_s**, you can store this generated file name in *str*. The maximum length of a string returned by **tmpnam_s** is **L_tmpnam_s**, defined in STDIO.H. If *str* is **NULL**, then **tmpnam_s** leaves the result in an internal static buffer. Thus any subsequent calls destroy this value. The name generated by **tmpnam_s** consists of a program-generated file name and, after the first call to **tmpnam_s**, a file extension of sequential numbers in base 32 (.1-.1vvvvvu, when **TMP_MAX_S** in STDIO.H is INT_MAX).

tmpnam_s automatically handles multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the OEM code page obtained from the operating system. **_wtmpnam_s** is a wide-character version of **tmpnam_s**; the argument and return value of **_wtmpnam_s** are wide-character strings. **_wtmpnam_s** and **tmpnam_s** behave identically except that **_wtmpnam_s** does not handle multibyte-character strings.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically,

eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_ttmpnam_s	**tmpnam_s**	**tmpnam_s**	**_wtmpnam_s**

Requirements

Routine	Required header	Compatibility
tmpnam_s	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wtmpnam_s	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_tmpnam_s.c
// This program uses tmpnam_s to create a unique filename in the
// current working directory.
//

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   char name1[L_tmpnam_s];
   errno_t err;
   int i;

   for (i = 0; i < 15; i++)
   {
      err = tmpnam_s( name1, L_tmpnam_s );
      if (err)
      {
         printf("Error occurred creating unique filename.\n");
         exit(1);
      }
      else
      {
         printf( "%s is safe to use as a temporary file.\n", name1 );
      }
   }
}
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O
_getmbcp
malloc
_setmbcp
tmpfile_s

to Functions

Each of the **to** functions and its associated macro, if any, converts a single character to another character.

__toascii	toupper, _toupper, towupper
tolower, _tolower, towlower	

Remarks

The **to** functions and macro conversions are as follows.

Routine	Macro	Description
__toascii	**__toascii**	Converts *c* to ASCII character
tolower	**tolower**	Converts *c* to lowercase if appropriate
_tolower	**_tolower**	Converts *c* to lowercase
towlower	None	Converts *c* to corresponding wide-character lowercase letter
toupper	**toupper**	Converts *c* to uppercase if appropriate
_toupper	**_toupper**	Converts *c* to uppercase
towupper	None	Converts c to corresponding wide-character uppercase letter

To use the function versions of the **to** routines that are also defined as macros, either remove the macro definitions with **#undef** directives or do not include CTYPE.H. If you use the /Za compiler option, the compiler uses the function version of **toupper** or **tolower**. Declarations of the **toupper** and **tolower** functions are in STDLIB.H.

The **__toascii** routine sets all but the low-order 7 bits of *c* to 0, so that the converted value represents a character in the ASCII character set. If *c* already represents an ASCII character, *c* is unchanged.

The **tolower** and **toupper** routines:

- Are dependent on the **LC_CTYPE** category of the current locale (**tolower** calls **isupper** and **toupper** calls **islower**).
- Convert *c* if *c* represents a convertible letter of the appropriate case in the current locale and the opposite case exists for that locale. Otherwise, *c* is unchanged.

The **_tolower** and **_toupper** routines:

- Are locale-independent, much faster versions of **tolower** and **toupper.**
- Can be used only when **isascii(*c*)** and either **isupper(*c*)** or **islower(*c*)**, respectively, are nonzero.
- Have undefined results if *c* is not an ASCII letter of the appropriate case for converting.

The **towlower** and **towupper** functions return a converted copy of *c* if and only if both of the following conditions are nonzero. Otherwise, *c* is unchanged.

- *c* is a wide character of the appropriate case (that is, for which **iswupper** or **iswlower,** respectively, is nonzero).
- There is a corresponding wide character of the target case (that is, for which **iswlower** or **iswupper,** respectively, is nonzero).

Example

```
// crt_toupper.c
/* This program uses toupper and tolower to
 * analyze all characters between 0x0 and 0x7F. It also
```

```
 * applies _toupper and _tolower to any code in this
 * range for which these functions make sense.
 */

#include <ctype.h>
#include <string.h>

char msg[] = "Some of THESE letters are Capitals.";
char *p;

int main( void )
{
   printf( "%s\n", msg );

   /* Reverse case of message. */
   for( p = msg; p < msg + strlen( msg ); p++ )
   {
      if( islower( *p ) )
         putchar( _toupper( *p ) );
      else if( isupper( *p ) )
         putchar( _tolower( *p ) );
      else
         putchar( *p );
   }
}
```

Output

```
Some of THESE letters are Capitals.
sOME OF these LETTERS ARE cAPITALS.
```

See Also

Reference

Data Conversion

Locale

is, isw Routines

toascii

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant __toascii instead.

tolower, _tolower, towlower, _tolower_l, _towlower_l

Convert character to lowercase.

```
int tolower(
   int c
);
int _tolower(
   int c
);
int towlower(
   wint_t c
);
int _tolower_l(
   int c,
   _locale_t locale
);
int _towlower_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c

Character to convert.

locale

Locale to use for locale-specific translation.

Return Value

Each of these routines converts a copy of *c*, if possible, and returns the result. There is no return value reserved to indicate an error.

Remarks

Each of these routines converts a given uppercase letter to a lowercase letter if possible and appropriate. The case conversion of **towlower** is locale-specific. Only the characters relevant to the current locale are changed in case. The functions without the **_l** suffix use the currently set locale. The versions of these functions with the **_l** suffix take the locale as a parameter and use that instead of the currently set locale.

In order for **tolower** to give the expected results, __isascii and isupper must both return nonzero.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_totlower	**tolower**	**_mbctolower**	**towlower**
_totlower_l	**_tolower_l**	**_mbctolower_l**	**_towlower_l**

Note

_tolower_l and **_tolowper_l** have no locale dependence and are not meant to be called directly. They are provided for internal use by **_totlower_l**.

Requirements

Routine	Required header	Compatibility
tolower	<ctype.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

_tolower	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
towlower	<ctype.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example in to Functions.

.NET Framework Equivalent

System::Char::ToLower

See Also

Reference

Data Conversion
is, isw Routines
to Functions
Locale
Interpretation of Multibyte-Character Sequences

toupper, _toupper, towupper, _toupper_l, _towupper_l

Convert character to uppercase.

```
int toupper(
   int c
);
int _toupper(
   int c
);
int towupper(
   wint_t c
);
int _toupper_l(
   int c ,
   _locale_t locale
);
int _towupper_l(
   wint_t c ,
   _locale_t locale
);
```

Parameters

c

Character to convert.

locale

Locale to use.

Return Value

Each of these routines converts a copy of *c*, if possible, and returns the result.

If *c* is a wide character for which **iswlower** is nonzero and there is a corresponding wide character for which **iswupper** is nonzero, **towupper** returns the corresponding wide character; otherwise, **towupper** returns *c* unchanged.

There is no return value reserved to indicate an error.

In order for **toupper** to give the expected results, __isascii and islower must both return nonzero.

Remarks

Each of these routines converts a given lowercase letter to an uppercase letter if possible and appropriate. The case conversion of **towupper** is locale-specific. Only the characters relevant to the current locale are changed in case. The functions without the **_l** suffix use the currently set locale. The versions of these functions with the **_l** suffix take the locale as a parameter and use that instead of the currently set locale.

In order for **toupper** to give the expected results, __isascii and isupper must both return nonzero.

Data Conversion Routines

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_totupper	**toupper**	**_mbctoupper**	**towupper**
_totupper_l	**_toupper_l**	**_mbctoupper_l**	**_towupper_l**

Note

_toupper_l and **_towupper_l** have no locale dependence and are not meant to be called directly. They are provided for internal use by **_totupper_l**.

Requirements

Routine	Required header	Compatibility
toupper	<ctype.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_toupper	<ctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
towupper	<ctype.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example in to Functions.

.NET Framework Equivalent

System::Char::ToUpper

See Also

Reference

is, isw Routines
to Functions
Locale
Interpretation of Multibyte-Character Sequences

towctrans

Transforms a character.

```
wint_t towctrans(
   wint_t c,
   wctrans_t category
);
```

Parameters

c

The character you want to transform.

category

An identifier that contains the return value of wctrans.

Return Value

The character *c*, after **towctrans** used the transform rule in *category*.

Remarks

The value of *category* must have been returned by an earlier successful call to wctrans.

Requirements

Routine	Required header	Compatibility
towctrans	<wctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See **wctrans** for a sample that uses **towctrans**.

See Also

Reference

Data Conversion

tzset

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _tzset instead.

_tzset

Sets time environment variables.

```
void _tzset( void );
```

Remarks

The **_tzset** function uses the current setting of the environment variable **TZ** to assign values to three global variables: **_daylight**, **_timezone**, and **_tzname**. These variables are used by the _ftime and localtime functions to make corrections from coordinated universal time (UTC) to local time, and by the **time** function to compute UTC from system time. Use the following syntax to set the **TZ** environment variable:

set TZ=*tzn*[+ | –]*hh*[**:***mm*[**:***ss*]][*dzn*]

tzn

Three-letter time-zone name, such as PST. You must specify the correct offset from local time to UTC.

hh

Difference in hours between UTC and local time. Sign (+) optional for positive values.

mm

Minutes. Separated from *hh* by a colon (**:**).

ss

Seconds. Separated from *mm* by a colon (**:**).

dzn

Three-letter daylight-saving-time zone such as PDT. If daylight saving time is never in effect in the locality, set **TZ** without a value for *dzn*. The C run-time library assumes the United States' rules for implementing the calculation of daylight saving time (DST).

Note
Take care in computing the sign of the time difference. Because the time difference is the offset from local time to UTC (rather than the reverse), its sign may be the opposite of what you might intuitively expect. For time zones ahead of UTC, the time difference is negative; for those behind UTC, the difference is positive.

For example, to set the **TZ** environment variable to correspond to the current time zone in Germany, enter the following on the command line:

```
set TZ=GST-1GDT
```

This command uses GST to indicate German standard time, assumes that UTC is one hour behind Germany (or in other words, that Germany is one hour ahead of UTC), and assumes that Germany observes daylight-saving time.

If the **TZ** value is not set, **_tzset** attempts to use the time zone information specified by the operating system. In the Windows operating system, this information is specified in the Date/Time application in Control Panel. If **_tzset** cannot obtain this information, it uses PST8PDT by default, which signifies the Pacific Time zone.

Based on the **TZ** environment variable value, the following values are assigned to the global variables **_daylight**, **_timezone**, and **_tzname** when **_tzset** is called:

Global variable	Description	Default value
_daylight	Nonzero value if a daylight-saving-time zone is specified in **TZ** setting; otherwise, 0.	1

_timezone	Difference in seconds between local time and UTC.	28800 (28800 seconds equals 8 hours)
_tzname[0]	String value of time-zone name from **TZ** environmental variable; empty if **TZ** has not been set.	PST
_tzname[1]	String value of daylight-saving-time zone; empty if daylight-saving-time zone is omitted from **TZ** environmental variable.	PDT

The default values shown in the preceding table for **_daylight** and the **_tzname** array correspond to "PST8PDT." If the DST zone is omitted from the **TZ** environmental variable, the value of **_daylight** is 0 and the **_ftime**, **gmtime**, and **localtime** functions return 0 for their DST flags.

Requirements

Routine	Required header	Compatibility
_tzset	<time.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more information, see Compatibility.

Example

```
// crt_tzset.cpp
// This program first sets up the time zone by
// placing the variable named TZ=EST5 in the environment
// table. It then uses _tzset to set the global variables
// named _daylight, _timezone, and _tzname.

#include <time.h>
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   if( _putenv_s( "TZ", "EST5EDT" ) != 0 )
   {
      printf( "Unable to set TZ\n" );
      exit( 1 );
   }
   else
   {
      _tzset();
      int daylight;
      _get_daylight( &daylight );
      printf( "_daylight = %d\n", daylight );
      long timezone;
      _get_timezone( &timezone );
      printf( "_timezone = %ld\n", timezone );
      size_t s;
      char tzname[100];
      _get_tzname( &s, tzname, sizeof(tzname), 0 );
      printf( "_tzname[0] = %s\n", tzname );
   }
   exit( 0 );
}
```

Output

```
_daylight = 1
_timezone = 18000
_tzname[0] = EST
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Time Management
asctime, _wasctime
_ftime, _ftime32, _ftime64
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
time, _time32, _time64
_utime, _utime32 _utime64, _wutime, _wutime32, _wutime64

ultoa

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _ultoa or security-enhanced _ultoa_s instead.

_ultoa, _ultow

Convert an unsigned long integer to a string. These functions are deprecated because more secure versions are available; see _ultoa_s, _ultow_s.

```
char *_ultoa(
   unsigned long value,
   char *str,
   int radix
);
wchar_t *_ultow(
   unsigned long value,
   wchar_t *str,
   int radix
);
template <size_t size>
char *_ultoa(
   unsigned long value,
   char (&str)[size],
   int radix
); // C++ only
template <size_t size>
wchar_t *_ultow(
   unsigned long value,
   wchar_t (&str)[size],
   int radix
); // C++ only
```

Parameters

value

Number to be converted.

str

String result.

radix

Base of *value*.

Return Value

Each of these functions returns a pointer to *str*. There is no error return.

Remarks

The **_ultoa** function converts *value* to a null-terminated character string and stores the result (up to 33 bytes) in *str*. No overflow checking is performed. *radix* specifies the base of *value*; *radix* must be in the range 2 – 36. **_ultow** is a wide-character version of **_ultoa**.

Security Note
To prevent buffer overruns, ensure that the *str* buffer is large enough to hold the converted digits plus the trailing null-charac ter.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_ultot	**_ultoa**	**_ultoa**	**_ultow**

Requirements

Routine	Required header	Compatibility
_ultoa	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ultow	<stdlib.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for _itoa.

.NET Framework Equivalent

System::Convert::ToString

See Also

Reference

Data Conversion

_itoa, _i64toa, _ui64toa, _itow, _i64tow, _ui64tow

_ultoa_s, _ultow_s

Convert an unsigned long integer to a string. These are versions of _ultoa, _ultow with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _ultoa_s(
    unsigned long value,
    char *str,
    size_t sizeOfstr,
    int radix
);
errno_t _ultow_s(
    unsigned long value,
    wchar_t *str,
    size_t sizeOfstr,
    int radix
);
template <size_t size>
errno_t _ultoa_s(
    unsigned long value,
    char (&str)[size],
    int radix
); // C++ only
template <size_t size>
errno_t _ultow_s(
    unsigned long value,
    wchar_t (&str)[size],
    int radix
); // C++ only
```

Parameters

value

Number to be converted.

str

String result.

sizeOfstr

The size of the *str* in bytes for **_ultoa_s** or words for **_ultow_s**.

radix

Base of *value*.

Return Value

Zero if the function was successful or an error code.

Remarks

The **_ultoa_s** function converts the digits of *value* to a null-terminated character string and stores the result (up to 33 bytes) in *str*. The *radix* argument specifies the base of *value*, which must be in the range 2 – 36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (–). **_ultow_s** is a wide character version of **_ultoa_s**; the second argument of **_ultow_s** is a wide character strings.

If *str* is a **NULL** pointer, or if *sizeOfstr* is less than or equal to zero, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL** or if the *value* or *str* out of range of a long integer, these functions will return a -1 and set the **errno** to **ERANGE**.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_ultot_s	**_ultoa_s**	**_ultoa_s**	**_ultow_s**

Requirements

Routine	Required header	Compatibility
_ultoa_s	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ultow_s	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::Convert::ToString

See Also

Reference

Data Conversion
_ultoa, _ultow
_ltoa, _ltow
_ltoa_s, _ltow_s
_ltoa_s, _ltow_s

umask

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _umask or security-enhanced _umask_s instead.

_umask

Sets the default file-permission mask. This function is deprecated because a more secure version is available; see _umask_s.

```
int _umask(
   int pmode
);
```

Parameters

pmode

Default permission setting.

Return Value

_umask returns the previous value of *pmode*. There is no error return.

Remarks

The **_umask** function sets the file-permission mask of the current process to the mode specified by *pmode*. The file-permission mask modifies the permission setting of new files created by **_creat**, **_open**, or **_sopen**. If a bit in the mask is 1, the corresponding bit in the file's requested permission value is set to 0 (disallowed). If a bit in the mask is 0, the corresponding bit is left unchanged. The permission setting for a new file is not set until the file is closed for the first time.

The integer expression *pmode* contains one or both of the following manifest constants, defined in SYS\STAT.H:

_S_IWRITE

Writing permitted.

_S_IREAD

Reading permitted.

_S_IREAD | _S_IWRITE

Reading and writing permitted.

When both constants are given, they are joined with the bitwise-OR operator (**|**). If the *pmode* argument is **_S_IREAD**, reading is not allowed (the file is write-only). If the *pmode* argument is **_S_IWRITE**, writing is not allowed (the file is read-only). For example, if the write bit is set in the mask, any new files will be read-only. Note that with MS-DOS and the Windows operating systems, all files are readable; it is not possible to give write-only permission. Therefore, setting the read bit with **_umask** has no effect on the file's modes.

If *pmode* is not a combination of one of the manifest constants or incorporates an alternate set of constants, the function will simply ignore those.

Requirements

Routine	Required header	Compatibility
_umask	<io.h>, <sys/stat.h>, <sys/types.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_umask.c
// compile with: /W3
// This program uses _umask to set
```

```
// the file-permission mask so that all future
// files will be created as read-only files.
// It also displays the old mask.
#include <sys/stat.h>
#include <sys/types.h>
#include <io.h>
#include <stdio.h>

int main( void )
{
   int oldmask;

   /* Create read-only files: */
   oldmask = _umask( _S_IWRITE ); // C4996
   // Note: _umask is deprecated; consider using _umask_s instead
   printf( "Oldmask = 0x%.4x\n", oldmask );
}
```

Output

```
Oldmask = 0x0000
```

.NET Framework Equivalent

System::IO::File::SetAttributes

See Also

Reference

File Handling
Low-Level I/O
_chmod, _wchmod
_creat, _wcreat
_mkdir, _wmkdir
_open, _wopen

_umask_s

Sets the default file-permission mask. A version of _umask with security enhancements as described in Security Enhancements in the CRT.

```
errno_t _umask_s(
   int mode,
   int * pOldMode
);
```

Parameters

[in] *mode*

Default permission setting.

[out] *oldMode*

The previous value of the permission setting.

Return Value

Returns an error code if *Mode* does not specify a valid mode or the *pOldMode* pointer is **NULL**.

Error Conditions

mode	***pOldMode***	**Return Value**	**Contents of *oldMode***
any	**NULL**	**EINVAL**	not modified
invalid mode	any	**EINVAL**	not modified

If one of the above conditions occurs, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **_umask_s** returns **EINVAL** and sets **errno** to **EINVAL**.

Remarks

The **_umask_s** function sets the file-permission mask of the current process to the mode specified by *mode*. The file-permission mask modifies the permission setting of new files created by **_creat**, **_open**, or **_sopen**. If a bit in the mask is 1, the corresponding bit in the file's requested permission value is set to 0 (disallowed). If a bit in the mask is 0, the corresponding bit is left unchanged. The permission setting for a new file is not set until the file is closed for the first time.

The integer expression *pmode* contains one or both of the following manifest constants, defined in SYS\STAT.H:

_S_IWRITE

Writing permitted.

_S_IREAD

Reading permitted.

_S_IREAD | _S_IWRITE

Reading and writing permitted.

When both constants are given, they are joined with the bitwise-OR operator (**|**). If the *mode* argument is **_S_IREAD**, reading is not allowed (the file is write-only). If the *mode* argument is **_S_IWRITE**, writing is not allowed (the file is read-only). For example, if the write bit is set in the mask, any new files will be read-only. Note that with MS-DOS and the Windows operating systems, all files are readable; it is not possible to give write-only permission. Therefore, setting the read bit with **_umask_s** has no effect on the file's modes.

If *pmode* is not a combination of one of the manifest constants or incorporates an alternate set of constants, the function will simply ignore those.

Requirements

Routine	**Required header**	**Compatibility**

_umask_s	<io.h> and <sys/stat.h> and <sys/types.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_umask_s.c
/* This program uses _umask_s to set
 * the file-permission mask so that all future
 * files will be created as read-only files.
 * It also displays the old mask.
 */

#include <sys/stat.h>
#include <sys/types.h>
#include <io.h>
#include <stdio.h>

int main( void )
{
   int oldmask, err;

   /* Create read-only files: */
   err = _umask_s( _S_IWRITE, &oldmask );
   if (err)
   {
      printf("Error setting the umask.\n");
      exit(1);
   }
   printf( "Oldmask = 0x%.4x\n", oldmask );
}
```

Output

```
Oldmask = 0x0000
```

.NET Framework Equivalent

System::IO::File::SetAttributes

See Also

Reference

File Handling
Low-Level I/O
_chmod, _wchmod
_creat, _wcreat
_mkdir, _wmkdir
_open, _wopen
_umask

unexpected (CRT)

Calls **terminate** or function you specify using **set_unexpected**.

```
void unexpected( void );
```

Remarks

The **unexpected** routine is not used with the current implementation of C++ exception handling. **unexpected** calls **terminate** by default. You can change this default behavior by writing a custom termination function and calling **set_unexpected** with the name of your function as its argument. **unexpected** calls the last function given as an argument to **set_unexpected**.

Requirements

Routine	Required header	Compatibility
unexpected	<eh.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::Exception Class

See Also

Reference

Exception Handling Routines
abort
_set_se_translator
set_terminate (CRT)
set_unexpected (CRT)
terminate (CRT)

ungetc, ungetwc

Pushes a character back onto the stream.

```
int ungetc(
   int c,
      FILE *stream
);
wint_t ungetwc(
   wint_t c,
   FILE *stream
);
```

Parameters

c

Character to be pushed.

stream

Pointer to **FILE** structure.

Return Value

If successful, each of these functions returns the character argument *c*. If *c* cannot be pushed back or if no character has been read, the input stream is unchanged and **ungetc** returns **EOF**; **ungetwc** returns **WEOF**. If *stream* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **EOF** or **WEOF** is returned and **errno** is set to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **ungetc** function pushes the character *c* back onto *stream* and clears the end-of-file indicator. The stream must be open for reading. A subsequent read operation on *stream* starts with *c*. An attempt to push **EOF** onto the stream using **ungetc** is ignored.

Characters placed on the stream by **ungetc** may be erased if **fflush**, **fseek**, **fsetpos**, or **rewind** is called before the character is read from the stream. The file-position indicator will have the value it had before the characters were pushed back. The external storage corresponding to the stream is unchanged. On a successful **ungetc** call against a text stream, the file-position indicator is unspecified until all the pushed-back characters are read or discarded. On each successful **ungetc** call against a binary stream, the file-position indicator is decremented; if its value was 0 before a call, the value is undefined after the call.

Results are unpredictable if **ungetc** is called twice without a read or file-positioning operation between the two calls. After a call to **fscanf**, a call to **ungetc** may fail unless another read operation (such as **getc**) has been performed. This is because **fscanf** itself calls **ungetc**.

ungetwc is a wide-character version of **ungetc**. However, on each successful **ungetwc** call against a text or binary stream, the value of the file-position indicator is unspecified until all pushed-back characters are read or discarded.

These functions are thread-safe and lock sensitive data during execution. For a non-locking version, see _ungetc_nolock, _ungetwc_nolock.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_ungettc	**ungetc**	**ungetc**	**ungetwc**

Requirements

Routine	Required header	Compatibility
ungetc	<stdio.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

ungetwc	<stdio.h> or <wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_ungetc.c
// This program first converts a character
// representation of an unsigned integer to an integer. If
// the program encounters a character that is not a digit,
// the program uses ungetc to replace it in the  stream.
//

#include <stdio.h>
#include <ctype.h>

int main( void )
{
   int ch;
   int result = 0;

   // Read in and convert number:
   while( ((ch = getchar()) != EOF) && isdigit( ch ) )
      result = result * 10 + ch - '0';    // Use digit.
   if( ch != EOF )
      ungetc( ch, stdin );                // Put nondigit back.
   printf( "Number = %d\nNext character in stream = '%c'",
            result, getchar() );
}
```

Input

```
521a
```

Output

```
Number = 521
Next character in stream = 'a'
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O

getc, getwc

putc, putwc

_ungetc_nolock, _ungetwc_nolock

Pushes a character back onto the stream.

```
int _ungetc_nolock(
   int c,
      FILE *stream
);
wint_t _ungetwc_nolock(
   wint_t c,
   FILE *stream
);
```

Parameters

c

Character to be pushed.

stream

Pointer to **FILE** structure.

Return Value

If successful, each of these functions returns the character argument *c*. If *c* cannot be pushed back or if no character has been read, the input stream is unchanged and **_ungetc_nolock** returns **EOF**; **_ungetwc_nolock** returns **WEOF**. If *stream* is **NULL**, **EOF** or **WEOF** is returned and **errno** is set to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

These functions are non-locking versions of **ungetc** and **ungetwc**. The versions with the **_nolock** suffix are identical except that they are not protected from interference by other threads. They may be faster since they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_ungettc_nolock	**_ungetc_nolock**	**_ungetc_nolock**	**_ungetwc_nolock**

Requirements

Routine	Required header	Compatibility
_ungetc_nolock	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ungetwc_nolock	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O
getc, getwc
putc, putwc

ungetch

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _ungetch instead.

_ungetch, _ungetwch, _ungetch_nolock, _ungetwch_nolock

Pushes back the last character read from the console.

```
int _ungetch(
   int c
);
wint_t _ungetwch(
   wint_t c
);
int _ungetch_nolock(
   int c
);
wint_t _ungetwch_nolock(
   wint_t c
);
```

Parameters

c

Character to be pushed.

Return Value

Both functions return the character *c* if successful. If there is an error, **_ungetch** returns a value of **EOF** and **_ungetwch** returns **WEOF**.

Remarks

These functions push the character *c* back to the console, causing *c* to be the next character read by **_getch** or **_getche** (or **_getwch** or **_getwche**). **_ungetch** and **_ungetwch** fail if they are called more than once before the next read. The *c* argument may not be **EOF** (or **WEOF**).

The versions with the **_nolock** suffix are identical except that they are not protected from interference by other threads. They may be faster since they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_ungettch	**_ungetch**	**_ungetch**	**_ungetwch**
_ungettch_nolock	**_ungetch_nolock**	**_ungetch_nolock**	**_ungetwch_nolock**

Requirements

Routine	Required header	Compatibility
_ungetch, **_ungetch_nolock**	<conio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_ungetwch, **_ungetwch_nolock**	<conio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_ungetch.c
// compile with: /c
// In this program, a white-space delimited
```

```
// token is read from the keyboard. When the program
// encounters a delimiter, it uses _ungetch to replace
// the character in the keyboard buffer.
//

#include <conio.h>
#include <ctype.h>
#include <stdio.h>

int main( void )
{
   char buffer[100];
   int count = 0;
   int ch;

   ch = _getche();
   while( isspace( ch ) )      // Skip preceding white space.
      ch = _getche();
   while( count < 99 )         // Gather token.
   {
      if( isspace( ch ) )      // End of token.
         break;
      buffer[count++] = (char)ch;
      ch = _getche();
   }
   _ungetch( ch );            // Put back delimiter.
   buffer[count] = '\0';      // Null terminate the token.
   printf( "\ntoken = %s\n", buffer );
}
```

Input

```
White
```

Output

```
token = White
```

See Also

Reference

Console and Port I/O

_cscanf, _cscanf_l, _cwscanf, _cwscanf_l

_getch, _getwch

unlink

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _unlink instead.

_unlink, _wunlink

Delete a file.

```
int _unlink(
   const char *filename
);
int _wunlink(
   const wchar_t *filename
);
```

Parameters

filename

Name of file to remove.

Return Value

Each of these functions returns 0 if successful. Otherwise, the function returns –1 and sets **errno** to **EACCES**, which means the path specifies a read-only file, or to **ENOENT**, which means the file or path is not found or the path specified a directory.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

Remarks

The **_unlink** function deletes the file specified by *filename*. **_wunlink** is a wide-character version of **_unlink**; the *filename* argument to **_wunlink** is a wide-character string. These functions behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tunlink	**_unlink**	**_unlink**	**_wunlink**

Requirements

Routine	Required header	Compatibility
_unlink	<io.h> and <stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wunlink	<io.h> or <wchar.h>	Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Code Example

This program uses _unlink to delete CRT_UNLINK.TXT.

```
// crt_unlink.c

#include <stdio.h>

int main( void )
{
   if( _unlink( "crt_unlink.txt" ) == -1 )
      perror( "Could not delete 'CRT_UNLINK.TXT'" );
   else
      printf( "Deleted 'CRT_UNLINK.TXT'\n" );
}
```

Input: crt_unlink.txt

```
This file will be deleted.
```

Sample Output

```
Deleted 'CRT_UNLINK.TXT'
```

.NET Framework Equivalent

System::IO::File::Delete

See Also

Reference

File Handling

_close

remove, _wremove

_unlock_file

Unlocks a file, allowing other processes to access the file.

```
void _unlock_file(
   FILE* file
);
```

Parameters

file

File handle.

Remarks

The **_unlock_file** function unlocks the file specified by *file*. Unlocking a file allows access to the file by other processes. This function should not be called unless **_lock_file** was previously called on the *file* pointer. Calling **_unlock_file** on a file that isn't locked may result in a deadlock. For an example, see _lock_file.

Requirements

Routine	Required header	Compatibility
_unlock_file	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::IO::FileStream::Lock

See Also

Reference

File Handling
_creat, _wcreat
_open, _wopen
_lock_file

_utime, _utime32 _utime64, _wutime, _wutime32, _wutime64

Set the file modification time.

```
int _utime(
   const char *filename,
   struct _utimbuf *times
);
int _utime32(
   const char *filename,
   struct __utimbuf32 *times
);
int _utime64(
   const char *filename,
   struct __utimbuf64 *times
);
int _wutime(
   const wchar_t *filename,
   struct _utimbuf *times
);
int _wutime32(
   const wchar_t *filename,
   struct __utimbuf32 *times
);
int _wutime64(
   const wchar_t *filename,
   struct __utimbuf64 *times
);
```

Parameters

filename

Pointer to a string that contains the path or filename.

times

Pointer to stored time values.

Return Value

Each of these functions returns 0 if the file-modification time was changed. A return value of –1 indicates an error. If an invalid parameter is passed, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and **errno** is set to one of the following values:

EACCES

Path specifies directory or read-only file

EINVAL

Invalid *times* argument

EMFILE

Too many open files (the file must be opened to change its modification time)

ENOENT

Path or filename not found

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

The date can be changed for a file if the change date is after midnight, January 1, 1970, and before the end date of the function used. In Visual C++ 2005, **_utime** and **_wutime** now use a 64-bit time value, so the end date is 23:59:59, December 31, 3000, UTC. If **_USE_32BIT_TIME_T** is defined to force the old behavior, the end date is 03:14:07 January 19, 2038, UTC. **_utime32** or **_wutime32** use a 32-bit time type regardless of whether **_USE_32BIT_TIME_T** is defined, and always have the earlier end date, and **_utime64** or **_wutime64** always use the 64-bit time type, so always support the later end date.

Remarks

The **_utime** function sets the modification time for the file specified by *filename*. The process must have write access to the file in order to change the time. In the Windows operating system, you can change the access time and the modification time in the **_utimbuf** structure. If *times* is a **NULL** pointer, the modification time is set to the current local time. Otherwise, *times* must point to a structure of type **_utimbuf**, defined in SYS\UTIME.H.

The **_utimbuf** structure stores file access and modification times used by **_utime** to change file-modification dates. The structure has the following fields, which are both of type **time_t**:

actime

Time of file access

modtime

Time of file modification

In Visual C++ 2005, specific versions of the **_utimbuf** structure (**_utimebuf32** and **__utimbuf64**) are defined using the 32-bit and 64-bit versions of the time type. These are used in the 32-bit and 64-bit specific versions of this function. **_utimbuf** itself by default uses a 64-bit time type unless **_USE_32BIT_TIME_T** is defined.

_utime is identical to **_futime** except that the *filename* argument of **_utime** is a filename or a path to a file, rather than a file descriptor of an open file.

_wutime is a wide-character version of **_utime**; the *filename* argument to **_wutime** is a wide-character string. These functions behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tutime	**_utime**	**_utime**	**_wutime**
_tutime32	**_utime32**	**_utime32**	**_wutime32**
_tutime64	**_utime64**	**_utime64**	**_wutime64**

Requirements

Routine	Required headers	Optional headers	Compatibility
_utime, **_utime32**, **_utime64**	<sys/utime.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_utime64	<sys/utime.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wutime	<utime.h> or <wchar.h>	<errno.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

This program uses **_utime** to set the file-modification time to the current time.

```
// crt_utime.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/utime.h>
#include <time.h>
```

```
int main( void )
{
   struct tm tma = {0}, tmm = {0};
   struct _utimbuf ut;

   // Fill out the accessed time structure
   tma.tm_hour = 12;
   tma.tm_isdst = 0;
   tma.tm_mday = 15;
   tma.tm_min = 0;
   tma.tm_mon = 0;
   tma.tm_sec = 0;
   tma.tm_year = 103;

   // Fill out the modified time structure
   tmm.tm_hour = 12;
   tmm.tm_isdst = 0;
   tmm.tm_mday = 15;
   tmm.tm_min = 0;
   tmm.tm_mon = 0;
   tmm.tm_sec = 0;
   tmm.tm_year = 102;

   // Convert tm to time_t
   ut.actime = mktime(&tma);
   ut.modtime = mktime(&tmm);

   // Show file time before and after
   system( "dir crt_utime.c" );
   if( _utime( "crt_utime.c", &ut ) == -1 )
      perror( "_utime failed\n" );
   else
      printf( "File time modified\n" );
   system( "dir crt_utime.c" );
}
```

Sample Output

```
Volume in drive C has no label.
 Volume Serial Number is 9CAC-DE74

 Directory of C:\test

01/09/2003  05:38 PM               935 crt_utime.c
               1 File(s)            935 bytes
               0 Dir(s)  20,742,955,008 bytes free
File time modified
 Volume in drive C has no label.
 Volume Serial Number is 9CAC-DE74

 Directory of C:\test

01/15/2002  12:00 PM               935 crt_utime.c
               1 File(s)            935 bytes
               0 Dir(s)  20,742,955,008 bytes free
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Time Management
asctime, _wasctime
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32
_ftime, _ftime32, _ftime64

_futime, _futime32, _futime64
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
_stat, _wstat Functions
time, _time32, _time64

va_arg, va_end, va_start

Access variable-argument lists.

```
type va_arg(
      va_list arg_ptr,
      type
);
void va_end(
   va_list arg_ptr
);
void va_start(
   va_list arg_ptr
);  // (UNIX version)
void va_start(
   va_list arg_ptr,
      prev_param
); // (ANSI version)
```

Parameters

type

Type of argument to be retrieved.

arg_ptr

Pointer to list of arguments.

prev_param

Parameter preceding first optional argument (ANSI only).

Return Value

va_arg returns the current argument; **va_start** and **va_end** do not return values.

Remarks

The **va_arg**, **va_end**, and **va_start** macros provide a portable way to access the arguments to a function when the function takes a variable number of arguments. Two versions of the macros are available: The macros defined in STDARG.H conform to the ANSI C standard, and the macros defined in VARARGS.H are compatible with the UNIX System V definition. The macros are:

va_alist

Name of parameter to called function (UNIX version only)

va_arg

Macro to retrieve current argument

va_dcl

Declaration of **va_alist** (UNIX version only)

va_end

Macro to reset *arg_ptr*

va_list

typedef for pointer to list of arguments defined in STDIO.H

va_start

Macro to set *arg_ptr* to beginning of list of optional arguments (UNIX version only)

Both versions of the macros assume that the function takes a fixed number of required arguments, followed by a variable number of optional arguments. The required arguments are declared as ordinary parameters to the function and can be

accessed through the parameter names. The optional arguments are accessed through the macros in STDARG.H or VARARGS.H, which set a pointer to the first optional argument in the argument list, retrieve arguments from the list, and reset the pointer when argument processing is completed.

The ANSI C standard macros, defined in STDARG.H, are used as follows:

- All required arguments to the function are declared as parameters in the usual way. **va_dcl** is not used with the STDARG.H macros.
- **va_start** sets *arg_ptr* to the first optional argument in the list of arguments passed to the function. The argument *arg_ptr* must have **va_list** type. The argument *prev_param* is the name of the required parameter immediately preceding the first optional argument in the argument list. If *prev_param* is declared with the register storage class, the macro's behavior is undefined. **va_start** must be used before **va_arg** is used for the first time.
- **va_arg** retrieves a value of *type* from the location given by *arg_ptr* and increments *arg_ptr* to point to the next argument in the list, using the size of *type* to determine where the next argument starts. **va_arg** can be used any number of times within the function to retrieve arguments from the list.
- After all arguments have been retrieved, **va_end** resets the pointer to **NULL**.

The UNIX System V macros, defined in VARARGS.H, operate somewhat differently:

- Any required arguments to the function can be declared as parameters in the usual way.
- The last (or only) parameter to the function represents the list of optional arguments. This parameter must be named **va_alist** (not to be confused with **va_list**, which is defined as the type of **va_alist**).
- **va_dcl** appears after the function definition and before the opening left brace of the function. This macro is defined as a complete declaration of the **va_alist** parameter, including the terminating semicolon; therefore, no semicolon should follow **va_dcl**.
- Within the function, **va_start** sets *arg_ptr* to the beginning of the list of optional arguments passed to the function. **va_start** must be used before **va_arg** is used for the first time. The argument *arg_ptr* must have **va_list** type.
- **va_arg** retrieves a value of *type* from the location given by *arg_ptr* and increments *arg_ptr* to point to the next argument in the list, using the size of *type* to determine where the next argument starts. **va_arg** can be used any number of times within the function to retrieve the arguments from the list.
- After all arguments have been retrieved, **va_end** resets the pointer to **NULL**.

When compiled with /clr (Common Language Runtime Compilation), programs using these macros may generate unexpected results because of differences between native and common language runtime type systems. Consider this program:

```
#include <stdio.h>
#include <stdarg.h>

void testit ( int i, ...)
{
   va_list argptr;
   va_start(argptr, i);

   if ( i == 0 ) {
      int n = va_arg( argptr, int );
      printf( "%d\n", n );
   } else {
      char *s = va_arg( argptr, char* );
      printf( "%s\n", s);
   }
}

int main()
{
   testit( 0, 0xFFFFFFFF ); // 1st problem: 0xffffffff is not an int
   testit( 1, NULL );       // 2nd problem: NULL is not a char*
}
```

Note that **testit** expects its second parameter to be either an **int** or a **char***. The arguments being passed are 0xffffffff (an

unsigned int, not an **int**) and **NULL** (actually an **int**, not a **char***). When compiled for native code, the program produces the output

```
-1
(null)
```

However, when compiled with **/clr:pure**, the type mismatches cause the program to generate an exception. The solution is to use explicit casts:

```
int main()
{
   testit( 0, (int)0xFFFFFFFF ); // cast unsigned to int
   testit( 1, (char*)NULL );     // cast int to char*
}
```

Requirements

Routine	Required header	Optional headers	Compatibility
va_arg	<stdio.h> and <stdarg.h>	<varargs.h>*	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
va_end	<stdio.h> and <stdarg.h>	<varargs.h>*	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
va_start	<stdio.h> and <stdarg.h>	<varargs.h>*	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility in the Introduction.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_va.c
/* The program below illustrates passing a variable
 * number of arguments using the following macros:
 *      va_start            va_arg              va_end
 *      va_list             va_dcl (UNIX only)
 */

#include <stdio.h>
#define ANSI            /* Comment out for UNIX version     */
#ifdef ANSI             /* ANSI compatible version          */
#include <stdarg.h>
int average( int first, ... );
#else                   /* UNIX compatible version          */
#include <varargs.h>
int average( va_list );
#endif

int main( void )
{
   /* Call with 3 integers (-1 is used as terminator). */
   printf( "Average is: %d\n", average( 2, 3, 4, -1 ) );

   /* Call with 4 integers. */
```

```
   printf( "Average is: %d\n", average( 5, 7, 9, 11, -1 ) );

   /* Call with just -1 terminator. */
   printf( "Average is: %d\n", average( -1 ) );
}

/* Returns the average of a variable list of integers. */
#ifdef ANSI             /* ANSI compatible version    */
int average( int first, ... )
{
   int count = 0, sum = 0, i = first;
   va_list marker;

   va_start( marker, first );     /* Initialize variable arguments. */
   while( i != -1 )
   {
      sum += i;
      count++;
      i = va_arg( marker, int);
   }
   va_end( marker );              /* Reset variable arguments.      */
   return( sum ? (sum / count) : 0 );
}
#else       /* UNIX compatible version must use old-style definition.  */
int average( va_alist )
va_dcl
{
   int i, count, sum;
   va_list marker;

   va_start( marker );            /* Initialize variable arguments. */
   for( sum = count = 0; (i = va_arg( marker, int)) != -1; count++ )
      sum += i;
   va_end( marker );              /* Reset variable arguments.      */
   return( sum ? (sum / count) : 0 );
}
#endif
```

Output

```
Average is: 3
Average is: 8
Average is: 0
```

.NET Framework Equivalent

System::ParamArrayAttribute Class

See Also

Reference

Argument Access

vfprintf, _vfprintf_l, vfwprintf, _vfwprintf_l

_vcprintf, _vcprintf_l, _vcwprintf, _vcwprintf_l

Write formatted output to the console using a pointer to a list of arguments. More secure versions are available, see _vcprintf_s, _vcprintf_s_l, _vcwprintf_s, _vcwprintf_s_l.

```
int _vcprintf(
   const char* format,
   va_list argptr
);
int _vcprintf_l(
   const char* format,
   locale_t locale,
   va_list argptr
);
int _vcwprintf(
   const wchar_t* format,
   va_list argptr
);
int _vcwprintf_l(
   const wchar_t* format,
   locale_t locale,
   va_list argptr
);
```

Parameters

format

Format specification.

argptr

Pointer to list of arguments.

locale

The locale to use.

For more information, see Format Specifications.

Return Value

The number of characters written, or a negative value if an output error occurs. If *format* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and -1 is returned.

Remarks

Each of these functions takes a pointer to an argument list, then formats and writes the given data to the console. **_vcwprintf** is the wide-character version of **_vcprintf**. It takes a wide-character string as an argument.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

Security Note
Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_vtcprintf	**_vcprintf**	**_vcprintf**	**_vcwprintf**
_vtcprintf_l	**_vcprintf_l**	**_vcprintf_l**	**_vcwprintf_l**

Requirements

Routine	Required header	Optional headers	Compatibility
_vcprintf, _vcprintf_l	<conio.h> and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_vcwprintf, _vcwprintf_l	<conio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_vcprintf.cpp
// compile with: /c
#include <conio.h>
#include <stdarg.h>

// An error formatting function used to print to the console.
int eprintf(const char* format, ...)
{
  va_list args;
  va_start(args, format);
  return _vcprintf(format, args);
}

int main()
{
   eprintf("  (%d:%d): Error %s%d : %s\n", 10, 23, "C", 2111,
           "<some error text>");
   eprintf("  (Related to symbol '%s' defined on line %d).\n",
           "<symbol>", 5 );
}
```

Output

```
(10,23): Error C2111 : <some error text>
(Related to symbol '<symbol>' defined on line 5).
```

.NET Framework Equivalent

System::Console::Write

See Also

Reference

Stream I/O
vprintf Functions
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_end, va_start

_vcprintf_p, _vcprintf_p_l, _vcwprintf_p, _vcwprintf_p_l

Write formatted output to the console using a pointer to a list of arguments, with support for positional parameters in the format string.

```
int _vcprintf_p(
   const char* format,
   va_list argptr
);
int _vcprintf_p_l(
   const char* format,
   locale_t locale,
   va_list argptr
);
int _vcwprintf_p(
   const wchar_t* format,
   va_list argptr
);
int _vcwprintf_p_l(
   const wchar_t* format,
   locale_t locale,
   va_list argptr
);
```

Parameters

format

Format specification.

argptr

Pointer to list of arguments.

locale

The locale to use.

Return Value

The number of characters written, or a negative value if an output error occurs. If *format* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and -1 is returned.

Remarks

Each of these functions takes a pointer to an argument list, then formats and writes the given data to the console using the **_putch** function (**_putwch** for **_vcwprintf_p**). **_vcwprintf_p** is the wide-character version of **_vcprintf_p**. It takes a wide-character string as an argument.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

Each *argument* (if any) is converted and output according to the corresponding format specification in *format*. The format specification supports positional parameters, which allows specifying the order in which the arguments are used in the format string. For more information, see printf_p Positional Parameters.

These functions do not translate line-feed characters into carriage return–line feed (CR-LF) combinations when output.

Security Note
Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

These functions validate the input pointer and the format string. If *format* or *argument* is **NULL**, or if the format string contains invalid formatting characters, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

Tchar.h routine	_UNICODE and _MBCS not defined	_MBCS defined	_UNICODE defined
_vtcprintf_p	**_vcprintf_p**	**_vcprintf_p**	**_vcwprintf_p**
_vtcprintf_p_l	**_vcprintf_p_l**	**_vcprintf_p_l**	**_vcwprintf_p_l**

Requirements

Routine	Required header	Compatibility
_vcprintf_p, **_vcprintf_p_l**	<conio.h> and <stdarg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_vcwprintf_p, **_vcwprintf_p_l**	<conio.h> and <stdarg.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_vcprintf_p.c
// compile with: /c
#include <conio.h>
#include <stdarg.h>

// An error formatting function used to print to the console.
int eprintf(const char* format, ...)
{
  va_list args;
  va_start(args, format);
  return _vcprintf_p(format, args);
}

int main()
{
   int n = eprintf("parameter 2 = %2$d; parameter 1 = %1$s\r\n",
      "one", 222);
   _cprintf_s("%d characters printed\r\n");
}
```

Output

```
parameter 2 = 222; parameter 1 = one
38 characters printed
```

See Also

Reference

Console and Port I/O
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l
va_arg, va_end, va_start
printf_p Positional Parameters

_vcprintf_s, _vcprintf_s_l, _vcwprintf_s, _vcwprintf_s_l

Write formatted output to the console using a pointer to a list of arguments. These are versions of _vcprintf, _vcprintf_l, _vcwprintf, _vcwprintf_l with security enhancements as described in Security Enhancements in the CRT.

```
int _vcprintf(
   const char* format,
   va_list argptr
);
int _vcprintf(
   const char* format,
   locale_t locale,
   va_list argptr
);
int _vcwprintf_s(
   const wchar_t* format,
   va_list argptr
);
int _vcwprintf_s_l(
   const wchar_t* format,
   locale_t locale,
   va_list argptr
);
```

Parameters

format

Format specification.

argptr

Pointer to list of arguments.

locale

The locale to use.

For more information, see Format Specifications.

Return Value

The number of characters written, or a negative value if an output error occurs.

Like the non-secure version of these functions, if *format* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. Additionally, unlike the non-secure version of these functions, if *format* does not specify a valid format, an invalid parameter exception is generated. If execution is allowed to continue, these functions return

Remarks

Each of these functions takes a pointer to an argument list, then formats and writes the given data to the console. **_vcwprintf_s** is the wide-character version of **_vcprintf_s**. It takes a wide-character string as an argument.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

Security Note
Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_vtcprintf_s	**_vcprintf_s**	**_vcprintf_s**	**_vcwprintf_s**
_vtcprintf_s_l	**_vcprintf_s_l**	**_vcprintf_s_l**	**_vcwprintf_s_l**

Requirements

Routine	Required header	Optional headers	Compatibility
_vcprintf_s, **_vcprintf_s_l**	<conio.h> and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_vcwprintf_s, **_vcwprintf_s_l**	<conio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_vcprintf_s.cpp
#include <conio.h>
#include <stdarg.h>

// An error formatting function used to print to the console.
int eprintf_s(const char* format, ...)
{
  va_list args;
  va_start(args, format);
  return _vcprintf_s(format, args);
}

int main()
{
   eprintf_s("  (%d:%d): Error %s%d : %s\n", 10, 23, "C", 2111,
           "<some error text>");
   eprintf_s("  (Related to symbol '%s' defined on line %d).\n",
           "<symbol>", 5 );
}
```

Sample Output

```
(10,23): Error C2111 : <some error text>
(Related to symbol '<symbol>' defined on line 5).
```

.NET Framework Equivalent

System::Console::Write

See Also

Reference

Stream I/O
vprintf Functions
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_end, va_start

vfprintf, _vfprintf_l, vfwprintf, _vfwprintf_l

Write formatted output using a pointer to a list of arguments. More secure versions of these functions exist; see vfprintf_s, _vfprintf_s_l, vfwprintf_s, _vfwprintf_s_l.

```
int vfprintf(
   FILE *stream,
   const char *format,
   va_list argptr
);
int _vfprintf_l(
   FILE *stream,
   const char *format,
   locale_t locale,
      va_list argptr
);
int vfwprintf(
      FILE *stream,
   const wchar_t *format,
      va_list argptr
);
int _vfwprintf_l(
      FILE *stream,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
```

Parameters

stream

Pointer to **FILE** structure.

format

Format specification.

argptr

Pointer to list of arguments.

locale

The locale to use.

For more information, see Format Specifications.

Return Value

vfprintf and **vfwprintf** return the number of characters written, not including the terminating null character, or a negative value if an output error occurs. If either *stream* or *format* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions takes a pointer to an argument list, then formats and writes the given data to *stream*.

vfwprintf is the wide-character version of **vfprintf**; the two functions behave identically if the stream is opened in ANSI mode. **vfprintf** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Security Note

Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_vftprintf	**vfprintf**	**vfprintf**	**vfwprintf**
_vftprintf_l	**_vfprintf_l**	**_vfprintf_l**	**_vfwprintf_l**

Requirements

Routine	Required header	Optional headers	Compatibility
vfprintf, **_vfprintf_l**	<stdio.h> and <stdarg.h>	<varargs.h>*	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
vfwprintf, **_vfwprintf_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O
vprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_end, va_start

_vfprintf_p, _vfprintf_p_l, _vfwprintf_p, _vfwprintf_p_l

Write formatted output using a pointer to a list of arguments, with the ability to specify the order that arguments are used in the format string.

```
int _vfprintf_p(
   FILE *stream,
   const char *format,
   va_list argptr
);
int _vfprintf_p_l(
   FILE *stream,
   const char *format,
   locale_t locale,
   va_list argptr
);
int _vfwprintf_p(
   FILE *stream,
   const wchar_t *format,
   va_list argptr
);
int _vfwprintf_p_l(
   FILE *stream,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
```

Parameters

stream

Pointer to **FILE** structure.

format

Format specification.

argptr

Pointer to list of arguments.

locale

The locale to use.

For more information, see Format Specifications.

Return Value

_vfprintf_p and **_vfwprintf_p** return the number of characters written, not including the terminating null character, or a negative value if an output error occurs.

Remarks

Each of these functions takes a pointer to an argument list, then formats and writes the given data to *stream*. These functions differ from the **_vfprint_s** and **_vfwprint_s** versions only in that they support positional parameters. For more information, see printf_p Positional Parameters.

_vfwprintf_p is the wide-character version of **_vprintf_p**; the two functions behave identically if the stream is opened in ANSI mode. **_vprintf_p** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Security Note

Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

If either *stream* or *format* is a null pointer, or if the format string contains invalid formatting characters, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_vftprintf_p	**_vfprintf_p**	**_vfprintf_p**	**_vfwprintf_p**
_vftprintf_p_l	**_vfprintf_p_l**	**_vfprintf_p_l**	**_vfwprintf_p_l**

Requirements

Routine	Required header	Optional headers	Compatibility
_vfprintf_p, **_vfprintf_p_l**	<stdio.h> and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003.
_vfwprintf_p, **_vfwprintf_p_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003.

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O
vprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_end, va_start
printf_p Positional Parameters
_fprintf_p, _fprintf_p_l, _fwprintf_p, _fwprintf_p_l
_vsprintf_p, _vsprintf_p_l, _vswprintf_p, _vswprintf_p_l
_sprintf_p, _sprintf_p_l, _swprintf_p, _swprintf_p_l

vfprintf_s, _vfprintf_s_l, vfwprintf_s, _vfwprintf_s_l

Write formatted output using a pointer to a list of arguments. These are versions of vfprintf, _vfprintf_l, vfwprintf, _vfwprintf_l with security enhancements as described in Security Enhancements in the CRT.

```
int vfprintf_s(
   FILE *stream,
   const char *format,
   va_list argptr
);
int _vfprintf_s_l(
   FILE *stream,
   const char *format,
   locale_t locale,
   va_list argptr
);
int vfwprintf_s(
   FILE *stream,
   const wchar_t *format,
   va_list argptr
);
int _vfwprintf_s_l(
   FILE *stream,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
```

Parameters

stream

Pointer to **FILE** structure.

format

Format specification.

argptr

Pointer to list of arguments.

locale

The locale to use.

For more information, see Format Specifications.

Return Value

vfprintf_s and **vfwprintf_s** return the number of characters written, not including the terminating null character, or a negative value if an output error occurs. If either *stream* or *format* is a null pointer, or if the format string contains invalid formatting characters, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions takes a pointer to an argument list, then formats and writes the given data to *stream*.

These functions differ from the non-secure versions only in that the secure versions check that the *format* string contains valid formatting characters.

vfwprintf_s is the wide-character version of **vfprintf_s**; the two functions behave identically if the stream is opened in ANSI mode. **vfprintf_s** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the

current thread locale.

Security Note
Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_vftprintf_s	**vfprintf_s**	**vfprintf_s**	**vfwprintf_s**
_vftprintf_s_l	**_vfprintf_s_l**	**_vfprintf_s_l**	**_vfwprintf_s_l**

Requirements

Routine	Required header	Optional headers	Compatibility
vfprintf_s, **_vfprintf_s_l**	<stdio.h> and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
vfwprintf_s, **_vfwprintf_s_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O
vprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_end, va_start

vprintf Functions

Each of the **vprintf** functions takes a pointer to an argument list, then formats and writes the given data to a particular destination. The functions differ in the parameter validation performed, whether the functions take wide or single-byte character strings, the output destination, and the support for specifying the order in which parameters are used in the format string.

_vcprintf, _vcwprintf	vfprintf, vfwprintf
_vfprintf_p, _vfprintf_p_l, _vfwprintf_p, _vfwprintf_p_l	vfprintf_s, _vfprintf_s_l, vfwprintf_s, _vfwprintf_s_l
vprintf, vwprintf	_vprintf_p, _vprintf_p_l, _vwprintf_p, _vwprintf_p_l
vprintf_s, _vprintf_s_l, vwprintf_s, _vwprintf_s_l	vsprintf, vswprintf
_vsprintf_p, _vsprintf_p_l, _vswprintf_p, _vswprintf_p_l	vsprintf_s, _vsprintf_s_l, vswprintf_s, _vswprintf_s_l
_vscprintf, _vscprintf_l, _vscwprintf, _vscwprintf_l	_vsnprintf, _vsnwprintf

Remarks

The **vprintf** functions are similar to their counterpart functions as listed in the following table. However, each **vprintf** function accepts a pointer to an argument list, whereas each of the counterpart functions accepts an argument list.

These functions format data for output to destinations as follows.

Function	Counterpart function	Output destination	Parameter Validation	Positional Parameter Support
_vcprintf	_cprintf	console	Check for null.	no
_vcwprintf	_cwprintf	console	Check for null.	no
vfprintf	fprintf	*Stream*	Check for null.	no
vfprintf_p	fprintf_p	*Stream*	Check for null and valid format.	yes
vfprintf_s	fprintf_s	*Stream*	Check for null and valid format.	no
vfwprintf	fwprintf	*Stream*	Check for null.	no
vfwprintf_p	fwprintf_p	*Stream*	Check for null and valid format.	yes
vfwprintf_s	fwprintf_s	*Stream*	Check for null and valid format.	no
vprintf	printf	**Stdout**	Check for null.	no
vprintf_p	printf_p	**Stdout**	Check for null and valid format.	yes
vprintf_s	printf_s	**Stdout**	Check for null and valid format.	no

vwprintf	wprintf	**Stdout**	Check for null.	no
vwprintf_p	wprintf_p	**Stdout**	Check for null and valid format.	yes
vwprintf_s	wprintf_s	**Stdout**	Check for null and valid format.	no
vsprintf	sprintf	memory pointed to by *buffer*	Check for null.	no
vsprintf_p	sprintf_p	memory pointed to by *buffer*	Check for null and valid format.	yes
vsprintf_s	sprintf_s	memory pointed to by *buffer*	Check for null and valid format.	no
vswprintf	swprintf	memory pointed to by *buffer*	Check for null.	no
vswprintf_p	swprintf_p	memory pointed to by *buffer*	Check for null and valid format.	yes
vswprintf_s	swprintf_s	memory pointed to by *buffer*	Check for null and valid format.	no
_vscprintf	_vscprintf	memory pointed to by *buffer*	Check for null.	no
_vscwprintf	_vscwprintf	memory pointed to by *buffer*	Check for null.	no
_vsnprintf	_snprintf	memory pointed to by *buffer*	Check for null.	no
_vsnwprintf	_snwprintf	memory pointed to by *buffer*	Check for null.	no

The *argptr* argument has type **va_list**, which is defined in VARARGS.H and STDARG.H. The *argptr* variable must be initialized by **va_start,** and may be reinitialized by subsequent **va_arg** calls; *argptr* then points to the beginning of a list of arguments that are converted and transmitted for output according to the corresponding specifications in the *format* argument. *format* has the same form and function as the *format* argument for printf. None of these functions invokes **va_end**. For a more complete description of each **vprintf** function, see the description of its counterpart function as listed in the preceding table.

_vsnprintf differs from **vsprintf** in that it writes no more than *count* bytes to *buffer*.

The versions of these functions with the **w** infix in the name are wide-character versions of the corresponding functions without the **w** infix; in each of these wide-character functions, *buffer* and *format* are wide-character strings. Otherwise, each wide-character function behaves identically to its SBCS counterpart function.

The versions of these functions with **_s** and **_p** suffixes are the more secure versions. These versions validate the format strings and will generate an exception if the format string is not well formed (for example, if invalid formatting characters are used).

The versions of these functions with the **_p** suffix provide the ability to specify the order in which the supplied arguments are substituted in the format string. For more information, see printf_p Positional Parameters.

For **vsprintf**, **vswprintf**, **_vsnprintf** and **_vsnwprintf**, if copying occurs between strings that overlap, the behavior is undefined.

Security Note
Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns. If using the secure versi ons of these functions (either the **_s** or **_p** suffixes), a user-supplied format string could trigger an invalid parameter exceptio n if the user-supplied string contains invalid formatting characters.

See Also

Reference

Stream I/O

fprintf, _fprintf_l, fwprintf, _fwprintf_l

printf, _printf_l, wprintf, _wprintf_l

sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l

va_arg, va_end, va_start

vprintf, _vprintf_l, vwprintf, _vwprintf_l

Write formatted output using a pointer to a list of arguments. More secure versions of these functions are available, see vprintf_s, _vprintf_s_l, vwprintf_s, _vwprintf_s_l.

```
int vprintf(
   const char *format,
      va_list argptr
);
int _vprintf_l(
      const char *format,
   locale_t locale,
   va_list argptr
);
int vwprintf(
   const wchar_t *format,
   va_list argptr
);
int _vwprintf_l(
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
```

Parameters

format

Format specification.

argptr

Pointer to list of arguments.

locale

The locale to use.

Return Value

vprintf and **vwprintf** return the number of characters written, not including the terminating null character, or a negative value if an output error occurs. If *format* is a null pointer, or if the format string contains invalid formatting characters, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions takes a pointer to an argument list, then formats and writes the given data to **stdout**.

vwprintf is the wide-character version of **vprintf**; the two functions behave identically if the stream is opened in ANSI mode. **vprintf** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Security Note
Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns. The risk of using user-defined strings is much less in Visual C++ 2005, because invalid format strings are detected and result in an error.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_vtprintf	**vprintf**	**vprintf**	**vwprintf**

_vtprintf_l	_vprintf_l	_vprintf_l	_vwprintf_l

Requirements

Routine	Required header	Optional headers	Compatibility
vprintf, **_vprintf_l**	<stdio.h> and <stdarg.h>	<varargs.h>*	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
vwprintf, **_vwprintf_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::Console::Write

See Also

Reference

Stream I/O
vprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_end, va_start

_vprintf_p, _vprintf_p_l, _vwprintf_p, _vwprintf_p_l

Write formatted output using a pointer to a list of arguments, with the ability to specify the order in which the arguments are used.

```
int _vprintf_p(
   const char *format,
   va_list argptr
);
int _vprintf_p_l(
   const char *format,
   locale_t locale,
   va_list argptr
);
int _vwprintf_p(
   const wchar_t *format,
   va_list argptr
);
int _vwprintf_p_l(
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
```

Parameters

format

Format specification.

argptr

Pointer to list of arguments.

locale

The locale to use.

Return Value

_vprintf_p and **_vwprintf_p** return the number of characters written, not including the terminating null character, or a negative value if an output error occurs.

Remarks

Each of these functions takes a pointer to an argument list, then formats and writes the given data to **stdout**. These functions differ from **vprintf_s** and **vwprintf_s** only in that they support the ability to specify the order in which the arguments are used. For more information, see printf_p Positional Parameters.

_vwprintf_p is the wide-character version of **_vprintf_p**; the two functions behave identically if the stream is opened in ANSI mode. **_vprintf_p** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Security Note
Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

If *format* is a null pointer, or if the format string contains invalid formatting characters, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined

_vtprintf_p	**_vprintf_p**	**_vprintf_p**	**_vwprintf_p**
_vtprintf_p_l	**_vprintf_p_l**	**_vprintf_p_l**	**_vwprintf_p_l**

Requirements

Routine	Required header	Optional headers	Compatibility
_vprintf_p, **_vprintf_p_l**	<stdio.h> and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_vwprintf_p, **_vwprintf_p_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::Console::Write

See Also

Reference

Stream I/O
vprintf Functions
_fprintf_p, _fprintf_p_l, _fwprintf_p, _fwprintf_p_l
_printf_p, _printf_p_l, _wprintf_p, _wprintf_p_l
_sprintf_p, _sprintf_p_l, _swprintf_p, _swprintf_p_l
vsprintf_s, _vsprintf_s_l, vswprintf_s, _vswprintf_s_l
va_arg, va_end, va_start
_vfprintf_p, _vfprintf_p_l, _vfwprintf_p, _vfwprintf_p_l
_printf_p, _printf_p_l, _wprintf_p, _wprintf_p_l
printf_p Positional Parameters

vprintf_s, _vprintf_s_l, vwprintf_s, _vwprintf_s_l

Write formatted output using a pointer to a list of arguments. These are versions of vprintf, _vprintf_l, vwprintf, _vwprintf_l with security enhancements as described in Security Enhancements in the CRT.

```
int vprintf_s(
   const char *format,
   va_list argptr
);
int _vprintf_s_l(
   const char *format,
   locale_t locale,
   va_list argptr
);
int vwprintf_s(
   const wchar_t *format,
   va_list argptr
);
int _vwprintf_s_l(
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
```

Parameters

format

Format specification.

argptr

Pointer to list of arguments.

locale

The locale to use.

Return Value

vprintf_s and **vwprintf_s** return the number of characters written, not including the terminating null character, or a negative value if an output error occurs. If *format* is a null pointer, or if the format string contains invalid formatting characters, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions takes a pointer to an argument list, then formats and writes the given data to **stdout**.

The secure versions of these functions differ from **vprintf** and **vwprintf** only in that the secure versions check that the format string contains valid formatting characters.

vwprintf_s is the wide-character version of **vprintf_s**; the two functions behave identically if the stream is opened in ANSI mode. **vprintf_s** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Security Note
Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined

_vtprintf_s	**vprintf_s**	**vprintf_s**	**vwprintf_s**
_vtprintf_s_l	**_vprintf_s_l**	**_vprintf_s_l**	**_vwprintf_s_l**

Requirements

Routine	Required header	Optional headers	Compatibility
vprintf_s, **_vprintf_s_l**	<stdio.h> and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
vwprintf_s, **_vwprintf_s_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility in the Introduction.

.NET Framework Equivalent

System::Console::Write

See Also

Reference

Stream I/O
vprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_end, va_start

_vscprintf, _vscprintf_l, _vscwprintf, _vscwprintf_l

Returns the number of characters in the formatted string using a pointer to a list of arguments.

```
int _vscprintf(
   const char *format,
      va_list argptr
);
int _vscprintf_l(
      const char *format,
   locale_t locale,
      va_list argptr
);
int _vscwprintf(
      const wchar_t *format,
   va_list argptr
);
int _vscwprintf_l(
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
```

Parameters

format

Format-control string.

argptr

Pointer to list of arguments.

locale

The locale to use.

For more information, see Format Specifications.

Return Value

_vscprintf returns the number of characters that would be generated if the string pointed to by the list of arguments was printed or sent to a file or buffer using the specified formatting codes. The value returned does not include the terminating null character. **_vscwprintf** performs the same function for wide characters.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

If *format* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

Remarks

Each *argument* (if any) is converted according to the corresponding format specification in *format*. The format consists of ordinary characters and has the same form and function as the *format* argument for printf.

Security Note
Ensure that if *format* is a user-defined string, it is null terminated and has the correct number and type of parameters. For m ore information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_vsctprintf	**_vscprintf**	**_vscprintf**	**_vscwprintf**

_vsctprintf_l	**_vscprintf_l**	**_vscprintf_l**	**_vscwprintf_l**

Requirements

Routine	Required header	Compatibility
_vscprintf, **_vscprintf_l**	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_vscwprintf, **_vscwprintf_l**	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for vsprintf.

See Also

Reference

Stream I/O
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
scanf, _scanf_l, wscanf, _wscanf_l
sscanf, _sscanf_l, swscanf, _swscanf_l
vprintf Functions

_vscprintf_p, _vscprintf_p_l, _vscwprintf_p, _vscwprintf_p_l

Returns the number of characters in the formatted string using a pointer to a list of arguments, with the ability to specify the order in which the arguments are used.

```
int _vscprintf_p(
   const char *format,
   va_list argptr
);
int _vscprintf_p _l(
   const char *format,
   locale_t locale,
   va_list argptr
);
int _vscwprintf_p (
   const wchar_t *format,
   va_list argptr
);
int _vscwprintf_p _l(
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
```

Parameters

format

Format-control string.

argptr

Pointer to list of arguments.

locale

The locale to use.

Return Value

_vscprintf_p returns the number of characters that would be generated if the string pointed to by the list of arguments was printed or sent to a file or buffer using the specified formatting codes. The value returned does not include the terminating null character. **_vscwprintf_p** performs the same function for wide characters.

Remarks

These functions differ from **_vscprintf** and **_vscwprintf** only in that they support the ability to specify the order in which the arguments are used. For more information, see printf_p Positional Parameters.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

If *format* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

Security Note
Ensure that if *format* is a user-defined string, it is null terminated and has the correct number and type of parameters. For m ore information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_vsctprintf_p	_vscprintf_p	_vscprintf_p	_vscwprintf_p

_vsctprintf_p_l	**_vscprintf_p_l**	**_vscprintf_p_l**	**_vscwprintf_p_l**

Requirements

Routine	Required header	Compatibility
_vscprintf_p, **_vscprintf_p_l**	<stdio.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_vscwprintf_p, **_vscwprintf_p_l**	<stdio.h> or <wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

See the example for vsprintf.

See Also

Reference

vprintf Functions
_scprintf_p, _scprintf_p_l, _scwprintf_p, _scwprintf_p_l
_vscprintf, _vscprintf_l, _vscwprintf, _vscwprintf_l

vsnprintf, _vsnprintf, _vsnprintf_l, _vsnwprintf, _vsnwprintf_l

Write formatted output using a pointer to a list of arguments. These functions are deprecated because more secure versions are available; see vsnprintf_s, _vsnprintf_s, _vsnprintf_s_l, _vsnwprintf_s, _vsnwprintf_s_l.

```
int vsnprintf(
   char *buffer,
   size_t count,
   const char *format,
   va_list argptr
);
int _vsnprintf(
   char *buffer,
   size_t count,
   const char *format,
   va_list argptr
);
int _vsnprintf_l(
   char *buffer,
   size_t count,
   const char *format,
   locale_t locale,
   va_list argptr
);
int _vsnwprintf(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format,
   va_list argptr
);
int _vsnwprintf_l(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
template <size_t size>
int vsnprintf(
   char (&buffer)[size],
   size_t count,
   const char *format,
   va_list argptr
); // C++ only
template <size_t size>
int _vsnprintf(
   char (&buffer)[size],
   size_t count,
   const char *format,
   va_list argptr
); // C++ only
template <size_t size>
int _vsnprintf_l(
   char (&buffer)[size],
   size_t count,
   const char *format,
   locale_t locale,
   va_list argptr
); // C++ only
template <size_t size>
int _vsnwprintf(
   wchar_t (&buffer)[size],
   size_t count,
   const wchar_t *format,
```

```
   va_list argptr
); // C++ only
template <size_t size>
int _vsnwprintf_l(
   wchar_t (&buffer)[size],
   size_t count,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
); // C++ only
```

Parameters

buffer

Storage location for output.

count

Maximum number of characters to write.

format

Format specification.

argptr

Pointer to list of arguments.

locale

The locale to use.

Return Value

vsnprintf, **_vsnprintf**, and **_vsnwprintf** return the number of characters written if the number of characters to write is less than or equal to *count*; if the number of characters to write is greater than *count*, these functions return -1 indicating that output has been truncated. The return value does not include the terminating null, if one is written.

If *buffer* or *format* is **NULL**, or if count is less than or equal to zero, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

Remarks

Each of these functions takes a pointer to an argument list, then formats the data, and writes up to *count* characters to the memory pointed to by *buffer*. If there is room at the end (that is, if the number of characters to write is less than *count*), the buffer will be null-terminated.

Security Note
Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

Note
To ensure that there is room for the terminating null, be sure that *count* is strictly less than the buffer length and initialize the buffer to null prior to calling the function.

vsnprintf is identical to **_vsnprintf**. **vsnprintf** is included for compliance to the ANSI standard; **vnsprintf** is retained for backward compatibility.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_vsntprintf	**_vsnprintf**	**_vsnprintf**	**_vsnwprintf**

_vsntprintf_l	_vsnprintf_l	_vsnprintf_l	_vsnwprintf_l

Requirements

Routine	Required header	Optional headers	Compatibility
vsnprintf	<stdio.h> and <stdarg.h>	<varargs.h>*	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_vsnprintf, **vsnprintf_l**	<stdio.h> and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_vsnwprintf, **_vsnwprintf_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_vsnprintf.cpp
// compile with: /W3
#include <stdio.h>
#include <wtypes.h>

void FormatOutput(LPCSTR formatstring, ...)
{
   int nSize = 0;
   char buff[10];
   memset(buff, 0, sizeof(buff));
   va_list args;
   va_start(args, formatstring);
   nSize = vsnprintf( buff, sizeof(buff) - 1, formatstring, args); // C4996
// Note: vsnprintf is deprecated; consider vsnprintf_s instead
   printf("nSize: %d, buff: %s\n", nSize, buff);
}

int main() {
   FormatOutput("%s %s", "Hi", "there");
   FormatOutput("%s %s", "Hi", "there!");
   FormatOutput("%s %s", "Hi", "there!!");
}
```

Output

```
nSize: 8, buff: Hi there
nSize: 9, buff: Hi there!
nSize: -1, buff: Hi there!
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Stream I/O
vprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l

va_arg, va_end, va_start

vsnprintf_s, _vsnprintf_s, _vsnprintf_s_l, _vsnwprintf_s, _vsnwprintf_s_l

Write formatted output using a pointer to a list of arguments. These are versions of vsnprintf, _vsnprintf, _vsnprintf_l, _vsnwprintf, _vsnwprintf_l with security enhancements as described in Security Enhancements in the CRT.

```
int vsnprintf_s(
   char *buffer,
   size_t sizeOfBuffer,
   size_t count,
   const char *format,
   va_list argptr
);
int _vsnprintf_s(
   char *buffer,
   size_t sizeOfBuffer,
   size_t count,
   const char *format,
   va_list argptr
);
int _vsnprintf_s_l(
   char *buffer,
   size_t sizeOfBuffer,
   size_t count,
   const char *format,
   locale_t locale,
   va_list argptr
);
int _vsnwprintf_s(
   wchar_t *buffer,
   size_t sizeOfBuffer,
   size_t count,
   const wchar_t *format,
   va_list argptr
);
int _vsnwprintf_s_l(
   wchar_t *buffer,
   size_t sizeOfBuffer,
   size_t count,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
template <size_t size>
int _vsnprintf_s(
   char (&buffer)[size],
   size_t count,
   const char *format,
   va_list argptr
); // C++ only
template <size_t size>
int _vsnwprintf_s(
   wchar_t (&buffer)[size],
   size_t count,
   const wchar_t *format,
   va_list argptr
); // C++ only
```

Parameters

buffer

Storage location for output.

sizeOfBuffer

The size of the *buffer* for output.

count

Maximum number of characters to write (not including the terminating null), or _TRUNCATE.

format

Format specification.

argptr

Pointer to list of arguments.

locale

The locale to use.

Return Value

vsnprintf_s, **_vsnprintf_s** and **_vsnwprintf_s** return the number of characters written, not including the terminating null, or a negative value if an output error occurs. **vsnprintf_s** is identical to **_vsnprintf_s**. **vsnprintf_s** is included for compliance to the ANSI standard. **_vnsprintf** is retained for backward compatibility.

If the storage required to store the data and a terminating null exceeds *sizeOfBuffer*, the invalid parameter handler is invoked, as described in Parameter Validation, unless *count* is _TRUNCATE, in which case as much of the string as will fit in *buffer* is written and -1 returned. If execution continues after the invalid parameter handler, these functions set *buffer* to an empty string, set **errno** to **ERANGE**, and return -1.

If *buffer* or *format* is a **NULL** pointer, or if *count* is less than or equal to zero, the invalid parameter handler is invoked. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1.

Error Conditions

Condition	**Return**	**errno**
buffer is **NULL**	-1	**EINVAL**
format is **NULL**	-1	**EINVAL**
count <= 0	-1	**EINVAL**
sizeOfBuffer too small (and *count* != **_TRUNCATE**)	-1 (and *buffer* set to an empty string)	**ERANGE**

Remarks

Each of these functions takes a pointer to an argument list, then formats and writes up to *count* characters of the given data to the memory pointed to by *buffer* and appends a terminating null.

If *count* is _TRUNCATE, then these functions write as much of the string as will fit in *buffer* while leaving room for a terminating null. If the entire string (with terminating null) fits in *buffer*, then these functions return the number of characters written (not including the terminating null); otherwise, these functions return -1 to indicate that truncation occurred.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Security Note
Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

Note
To ensure that there is room for the terminating null, be sure that *count* is strictly less than the buffer length, or use **_TRUNCATE**.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their

newer, secure counterparts. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_vsntprintf_s	**_vsnprintf_s**	**_vsnprintf_s**	**_vsnwprintf_s**
_vsntprintf_s_l	**_vsnprintf_s_l**	**_vsnprintf_s_l**	**_vsnwprintf_s_l**

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

Requirements

Routine	Required header	Optional headers	Compatibility
vsnprintf_s	<stdio.h> and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_vsnprintf_s, **_vsnprintf_s_l**	<stdio.h> and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_vsnwprintf_s, **_vsnwprintf_s_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_vsnprintf_s.cpp
#include <stdio.h>
#include <wtypes.h>

void FormatOutput(LPCSTR formatstring, ...)
{
   int nSize = 0;
   char buff[10];
   memset(buff, 0, sizeof(buff));
   va_list args;
   va_start(args, formatstring);
   nSize = vsnprintf_s( buff, sizeof(buff), _TRUNCATE, formatstring, args);
   printf("nSize: %d, buff: %s\n", nSize, buff);
}

int main() {
   FormatOutput("%s %s", "Hi", "there");
   FormatOutput("%s %s", "Hi", "there!");
   FormatOutput("%s %s", "Hi", "there!!");
}
```

Output

```
nSize: 8, buff: Hi there
nSize: 9, buff: Hi there!
nSize: -1, buff: Hi there!
```

See Also

Reference

Stream I/O

vprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_end, va_start

vsprintf, _vsprintf_l, vswprintf, _vswprintf_l, __vswprintf_l

Write formatted output using a pointer to a list of arguments. These functions are deprecated because more secure versions are available; see vsprintf_s, _vsprintf_s_l, vswprintf_s, _vswprintf_s_l.

```
int vsprintf(
    char *buffer,
   const char *format,
      va_list argptr
);
int _vsprintf_l(
      char *buffer,
   const char *format,
   locale_t locale,
   va_list argptr
);
int vswprintf(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format,
   va_list argptr
);
int _vswprintf_l(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
int __vswprintf_l(
   wchar_t *buffer,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
template <size_t size>
int vsprintf(
   char (&buffer)[size],
   const char *format,
   va_list argptr
); // C++ only
template <size_t size>
int _vsprintf_l(
   char (&buffer)[size],
   const char *format,
   locale_t locale,
   va_list argptr
); // C++ only
template <size_t size>
int vswprintf(
   wchar_t (&buffer)[size],
   size_t count,
   const wchar_t *format,
   va_list argptr
); // C++ only
template <size_t size>
int _vswprintf_l(
   wchar_t (&buffer)[size],
   size_t count,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
); // C++ only
```

Parameters

buffer

Storage location for output.

count

Maximum number of characters to store, in the **UNICODE** version of this function.

format

Format specification.

argptr

Pointer to list of arguments.

locale

The locale to use.

Return Value

vsprintf and **vswprintf** return the number of characters written, not including the terminating null character, or a negative value if an output error occurs. If *buffer* or *format* is a null pointer, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions takes a pointer to an argument list, and then formats and writes the given data to the memory pointed to by **buffer**.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Security Note
Using **vsprintf**, here is no way to limit the number of characters written, which means that code using this function is suscep tible to buffer overruns. Use _vsnprintf instead, or call _vscprintf to determine how large a buffer is needed. Also, ensure that **format** is not a user-defined string. For more information, see Avoiding Buffer Overruns.

In Visual C++ 2005, **vswprintf** conforms to the ISO C Standard, which requires the second parameter, *count*, of type **size_t**. To force the old nonstandard behavior, define **_CRT_NON_CONFORMING_SWPRINTFS.** The old behavior may not be in a future version, so code should be changed to use the new conformant behavior.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	**_UNICODE & _MBCS not defined**	**_MBCS defined**	**_UNICODE defined**
_vstprintf	**vsprintf**	**vsprintf**	**vswprintf**
_vstprintf_l	**_vsprintf_l**	**_vsprintf_l**	**_vswprintf_l**

Requirements

Routine	**Required header**	**Optional headers**	**Compatibility**
vsprintf, **_vsprintf_l**	<stdio.h> and <stdarg.h>	<varargs.h>*	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Editi on, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Profession al, Windows Server 2003

vswprintf, _vswprintf_l	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_vsprintf.c
// compile with: /W3
// This program uses vsprintf to write to a buffer.
// The size of the buffer is determined by _vscprintf.

#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>

void test( char * format, ... )
{
    va_list args;
    int     len;
    char    *buffer;

    // retrieve the variable arguments
    va_start( args, format );

    len = _vscprintf( format, args ) // _vscprintf doesn't count
                                + 1; // terminating '\0'

    buffer = (char*)malloc( len * sizeof(char) );

    vsprintf( buffer, format, args ); // C4996
    // Note: vsprintf is deprecated; consider using vsprintf_s instead
    puts( buffer );

    free( buffer );
}

int main( void )
{
   test( "%d %c %d", 123, '<', 456 );
   test( "%s", "This is a string" );
}
```

Output

```
123 < 456
This is a string
```

.NET Framework Equivalent

System::String::Format

See Also

Reference

Stream I/O
vprintf Functions
Format Specification Fields: printf and wprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_end, va_start

_vsprintf_p, _vsprintf_p_l, _vswprintf_p, _vswprintf_p_l

Write formatted output using a pointer to a list of arguments, with the ability to specify the order in which the arguments are used.

```
int _vsprintf_p(
   char *buffer,
   size_t numberOfElements,
   const char *format,
   va_list argptr
);
int _vsprintf_p_l(
   char *buffer,
   size_t numberOfElements,
   const char *format,
   locale_t locale,
   va_list argptr
);
int _vswprintf_p(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format,
   va_list argptr
);
int _vswprintf_p_l(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
```

Parameters

buffer

Storage location for output.

numberOfElements

Size of *buffer* in characters.

count

Maximum number of characters to store, in the UNICODE version of this function.

format

Format specification.

argptr

Pointer to list of arguments.

locale

The locale to use.

Return Value

_vsprintf_p and **_vswprintf_p** return the number of characters written, not including the terminating null character, or a negative value if an output error occurs.

Remarks

Each of these functions takes a pointer to an argument list, and then formats and writes the given data to the memory pointed to by *buffer*.

These functions differ from the **vsprintf_s** and **vswprintf_s** only in that they support positional parameters. For more

information, see printf_p Positional Parameters.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

If the *buffer* or *format* parameters are NULL pointers, if *count* is zero, or if the format string contains invalid formatting characters, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_vstprintf_p	**_vsprintf_p**	**_vsprintf_p**	**_vswprintf_p**
_vstprintf_p_l	**_vsprintf_p_l**	**_vsprintf_p_l**	**_vswprintf_p_l**

Requirements

Routine	Required header	Optional headers	Compatibility
_vsprintf_p, **_vsprintf_p_l**	<stdio.h> and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_vswprintf_p, **_vswprintf_p_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt__vsprintf_p.c
// This program uses vsprintf_p to write to a buffer.
// The size of the buffer is determined by _vscprintf_p.

#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>


void example( char * format, ... )
{
    va_list  args;
    int      len;
    char     *buffer = NULL;

    va_start( args, format );

    // _vscprintf doesn't count the
    // null terminating string so we add 1.
    len = _vscprintf_p( format, args ) + 1;

    // Allocate memory for our buffer
    buffer = (char*)malloc( len * sizeof(char) );
    if (buffer)
    {
        _vsprintf_p( buffer, len, format, args );
        puts( buffer );
        free( buffer );
    }
}
```

```
int main( void )
{
    // First example
    example( "%2$d %1$c %3$d", '<', 123, 456 );

    // Second example
    example( "%s", "This is a string" );
}
```

Output

```
123 < 456
This is a string
```

.NET Framework Equivalent

System::String::Format

See Also

Reference

Stream I/O
vprintf Functions
Format Specification Fields: printf and wprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_end, va_start

vsprintf_s, _vsprintf_s_l, vswprintf_s, _vswprintf_s_l

Write formatted output using a pointer to a list of arguments. These are versions of vsprintf, _vsprintf_l, vswprintf, _vswprintf_l, __vswprintf_l with security enhancements as described in Security Enhancements in the CRT.

```
int vsprintf_s(
   char *buffer,
   size_t numberOfElements,
   const char *format,
   va_list argptr
);
int _vsprintf_s_l(
   char *buffer,
   size_t numberOfElements,
   const char *format,
   locale_t locale,
   va_list argptr
);
int vswprintf_s(
   wchar_t *buffer,
   size_t numberOfElements,
   const wchar_t *format,
   va_list argptr
);
int _vswprintf_s_l(
   wchar_t *buffer,
   size_t numberOfElements,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
template <size_t size>
int vsprintf_s(
   char (&buffer)[size],
   const char *format,
   va_list argptr
); // C++ only
template <size_t size>
int vswprintf_s(
   wchar_t (&buffer)[size],
   const wchar_t *format,
   va_list argptr
); // C++ only
```

Parameters

buffer

Storage location for output.

numberOfElements

Size of *buffer*.

format

Format specification.

argptr

Pointer to list of arguments.

locale

The locale to use.

Return Value

vsprintf_s and **vswprintf_s** return the number of characters written, not including the terminating null character, or a negative value if an output error occurs. If *buffer* or *format* is a null pointer, if count is zero, or if the format string contains invalid formatting characters, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions takes a pointer to an argument list, and then formats and writes the given data to the memory pointed to by *buffer*.

In Visual C++ 2005, **vswprintf_s** conforms to the ISO C Standard for **vswprintf**, which requires the second parameter, *count*, of type **size_t**.

These functions differ from the non-secure versions only in that the secure versions support positional parameters. For more information, see printf_p Positional Parameters.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_vstprintf_s	**vsprintf_s**	**vsprintf_s**	**vswprintf_s**
_vstprintf_s_l	**_vsprintf_s_l**	**_vsprintf_s_l**	**_vswprintf_s_l**

Requirements

Routine	Required header	Optional headers	Compatibility
vsprintf_s, **_vsprintf_s_l**	<stdio.h> and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
vswprintf_s, **_vswprintf_s_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_vsprintf_s.c
// This program uses vsprintf_s to write to a buffer.
// The size of the buffer is determined by _vscprintf.

#include <stdlib.h>
#include <stdarg.h>

void test( char * format, ... )
{
   va_list args;
   int len;
   char * buffer;

   va_start( args, format );
   len = _vscprintf( format, args ) // _vscprintf doesn't count
```

```
                                + 1; // terminating '\0'
   buffer = malloc( len * sizeof(char) );
   vsprintf_s( buffer, len, format, args );
   puts( buffer );
   free( buffer );
}

int main( void )
{
   test( "%d %c %d", 123, '<', 456 );
   test( "%s", "This is a string" );
}
```

Output

```
123 < 456
This is a string
```

.NET Framework Equivalent

System::String::Format

See Also

Reference

Stream I/O
vprintf Functions
Format Specification Fields: printf and wprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_end, va_start

wcrtomb

Convert a wide character into its multibyte character representation. This function is deprecated because a more secure version is available; see wcrtomb_s.

```
size_t wcrtomb(
   char *mbchar,
   wchar_t wchar,
   mbstate_t *mbstate
);
template <size_t size>
size_t wcrtomb(
   char (&mbchar)[size],
   wchar_t wchar,
   mbstate_t *mbstate
); // C++ only
```

Parameters

[out] *mbchar*

The resulting multibyte converted character.

[in] *wchar*

A wide character to convert.

[in] *mbstate*

A pointer to an **mbstate_t** object.

Return Value

Returns the number of bytes required to represent the converted multibyte character, otherwise a -1 if an error occurs.

Remarks

The **wcrtomb** function converts a wide character, beginning in the specified conversion state contained in *mbstate*, from the value contained in *wchar*, into the address represented by *mbchar*. The return value is the number of bytes required to represent the corresponding multibyte character, but it will not return more than **MB_CUR_MAX** bytes.

If *mbstate* is null, the internal **mbstate_t** object containing the conversion state of *mbchar* is used. If the character sequence *wchar* does not have a corresponding multibyte character representation, a -1 is returned and the **errno** is set to **EILSEQ**.

The **wcrtomb** function differs from wctomb, _wctomb_l by its restartability. The conversion state is stored in *mbstate* for subsequent calls to the same or other restartable functions. Results are undefined when mixing the use of restartable and nonrestartable functions. For example, an application would use **wcsrlen** rather than **wcslen**, if a subsequent call to **wcsrtombs** were used instead of **wcstombs**.

In C++, this function has a template overload that invokes the newer, secure counterparts of this function. For more information, see Secure Template Overloads.

Exceptions

The **wcrtomb** function is multithread safe as long as no function in the current thread calls **setlocale** while this function is executing and while the *mbstate* is null.

Example

```
// crt_wcrtomb.c
// compile with: /W3
// This program converts a wide character
// to its corresponding multibyte character.

#include <string.h>
#include <stdio.h>
#include <wchar.h>
```

```
int main( void )
{
    size_t      sizeOfCovertion = 0;
    mbstate_t   mbstate;
    char        mbStr = 0;
    wchar_t*    wcStr = L"Q";

    // Reset to initial conversion state
    memset(&mbstate, 0, sizeof(mbstate));

    sizeOfCovertion = wcrtomb(&mbStr, *wcStr, &mbstate); // C4996
    // Note: wcrtomb is deprecated; consider using wcrtomb_s instead
    if (sizeOfCovertion > 0)
    {
        printf("The corresponding wide character \"");
        wprintf(L"%s\"", wcStr);
        printf(" was converted to the \"%c\" ", mbStr);
        printf("multibyte character.\n");
    }
    else
    {
        printf("No corresponding multibyte character "
               "was found.\n");
    }
}
```

Output

```
The corresponding wide character "Q" was converted to the "Q" multibyte character.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

Requirements

Routine	Required header	Compatibility
wcrtomb	<wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

See Also

Reference

Data Conversion
Locale
Interpretation of Multibyte-Character Sequences
mbsinit

wcrtomb_s

Convert a wide character into its multibyte character representation. A version of wcrtomb with security enhancements as described in Security Enhancements in the CRT.

```
errno_t wcrtomb_s(
   size_t *pReturnValue,
   char *mbchar,
   size_t sizeOfmbchar,
   wchar_t *wchar,
   mbstate_t *mbstate
);
template <size_t size>
errno_t wcrtomb_s(
   size_t *pReturnValue,
   char (&mbchar)[size],
   wchar_t *wchar,
   mbstate_t *mbstate
); // C++ only
```

Parameters

[out] *pReturnValue*

Returns the number of bytes written or -1 if an error occurred.

[out] *mbchar*

The resulting multibyte converted character.

[in] *sizeOfmbchar*

The size of the *mbchar* variable in bytes.

[in] *wchar*

A wide character to convert.

[in] *mbstate*

A pointer to an **mbstate_t** object.

Return Value

Returns zero or an **errno** value if an error occurs.

Remarks

The **wcrtomb_s** function converts a wide character, beginning in the specified conversion state contained in *mbstate*, from the value contained in *wchar*, into the address represented by *mbchar*. The *pReturnValue* value will be the number of bytes converted, but no more than **MB_CUR_MAX** bytes, or an -1 if an error occurred.

If *mbstate* is null, the internal **mbstate_t** conversion state is used. If the character contained in *wchar* does not have a corresponding multibyte character, the value of *pReturnValue* will be -1 and the function will return the **errno** value of **EILSEQ**.

The **wcrtomb_s** function differs from wctomb_s, _wctomb_s_l by its restartability. The conversion state is stored in *mbstate* for subsequent calls to the same or other restartable functions. Results are undefined when mixing the use of restartable and nonrestartable functions. For example, an application would use **wcsrlen** rather than **wcslen**, if a subsequent call to **wcsrtombs_s** were used instead of **wcstombs_s.**

In C++, using this function is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Exceptions

The **wcrtomb_s** function is multithread safe as long as no function in the current thread calls **setlocale** while this function is executing and the *mbstate* is null.

Example

```
// crt_wcrtomb_s.c
// This program converts a wide character
// to its corresponding multibyte character.
//

#include <string.h>
#include <stdio.h>
#include <wchar.h>

int main( void )
{
    errno_t     returnValue;
    size_t      pReturnValue;
    mbstate_t   mbstate;
    size_t      sizeOfmbStr = 1;
    char        mbchar = 0;
    wchar_t*    wchar = L"Q\0";

    // Reset to initial conversion state
    memset(&mbstate, 0, sizeof(mbstate));

    returnValue = wcrtomb_s(&pReturnValue, &mbchar, sizeof(char),
                            *wchar, &mbstate);
    if (returnValue == 0) {
        printf("The corresponding wide character \"");
        wprintf(L"%s\"", wchar);
        printf(" was converted to a the \"%c\" ", mbchar);
        printf("multibyte character.\n");
    }
    else
    {
        printf("No corresponding multibyte character "
               "was found.\n");
    }
}
```

Output

```
The corresponding wide character "Q" was converted to a the "Q" multibyte character.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

Requirements

Routine	Required header	Compatibility
wcrtomb_s	<wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

See Also

Reference

Data Conversion
Locale
Interpretation of Multibyte-Character Sequences
mbsinit

wcsrtombs

Convert a wide character string to its multibyte character string representation. This function is deprecated because a more secure version is available; see wcsrtombs_s.

```
size_t wcsrtombs(
   char *mbstr,
   const wchar_t **wcstr,
   sizeof count,
   mbstate_t *mbstate
);
template <size_t size>
size_t wcsrtombs(
   char (&mbstr)[size],
   const wchar_t **wcstr,
   sizeof count,
   mbstate_t *mbstate
); // C++ only
```

Parameters

[out] *mbstr*

The resulting converted multibyte character string's address location.

[in] *wcstr*

Indirectly points to the location of the wide character string to be converted.

[in] *count*

The number of character to be converted.

[in] *mbstate*

A pointer to an **mbstate_t** conversion state object.

Return Value

Returns the number of bytes successfully converted, not including the null terminating null byte (if any), otherwise a -1 if an error occurred.

Remarks

The **wcsrtombs** function converts a string of wide characters, beginning in the specified conversion state contained in *mbstate*, from the values indirect pointed to in *wcstr*, into the address of *mbstr*. The conversion will continue for each character until: after a null terminating wide character is encountered, when a non corresponding character is encountered or when the next character would exceed the limit contained in *count*. If **wcsrtombs** encounters the wide-character null character (L'\0') either before or when *count* occurs, it converts it to an 8-bit 0 and stops.

Thus, the multibyte character string at *mbstr* is null-terminated only if **wcsrtombs** encounters a wide character null character during conversion. If the sequences pointed to by *wcstr* and *mbstr* overlap, the behavior of **wcsrtombs** is undefined. **wcsrtombs** is affected by the LC_TYPE category of the current locale.

The **wcsrtombs** function differs from wcstombs, _wcstombs_l by its restartability. The conversion state is stored in *mbstate* for subsequent calls to the same or other restartable functions. Results are undefined when mixing the use of restartable and nonrestartable functions. For example, an application would use **wcsrlen** rather than **wcslen**, if a subsequent call to **wcsrtombs** were used instead of **wcstombs**.

If the *mbstr* argument is **NULL**, **wcsrtombs** returns the required size in bytes of the destination string. If *mbstate* is null, the internal **mbstate_t** conversion state is used. If the character sequence *wchar* does not have a corresponding multibyte character representation, a -1 is returned and the **errno** is set to **EILSEQ**.

In C++, this function has a template overload that invokes the newer, secure counterpart of this function. For more information, see Secure Template Overloads.

Exceptions

The **wcsrtombs** function is multithread safe as long as no function in the current thread calls **setlocale** while this function is executing and the *mbstate* is null.

Example

```
// crt_wcsrtombs.cpp
// compile with: /W3
// This code example converts a wide
// character string into a multibyte
// character string.

#include <stdio.h>
#include <memory.h>
#include <wchar.h>
#include <errno.h>

#define MB_BUFFER_SIZE 100

int main()
{
    const wchar_t   wcString[] =
                    {L"Every good boy does fine."};
    const wchar_t   *wcsIndirectString = wcString;
    char            mbString[MB_BUFFER_SIZE];
    size_t          countConverted;
    mbstate_t       mbstate;

    // Reset to initial shift state
    ::memset((void*)&mbstate, 0, sizeof(mbstate));

    countConverted = wcsrtombs(mbString, &wcsIndirectString,
                               MB_BUFFER_SIZE, &mbstate); // C4996
    // Note: wcsrtombs is deprecated; consider using wcsrtombs_s
    if (errno == EILSEQ)
    {
        printf( "An encoding error was detected in the string.\n" );
    }
    else
    {
        printf( "The string was successfuly converted.\n" );
    }
}
```

Output

```
The string was successfuly converted.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

Requirements

Routine	Required header	Compatibility
wcsrtombs	<wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

See Also

Reference

Data Conversion
Locale
Interpretation of Multibyte-Character Sequences
wcrtomb
wcrtomb_s
wctomb, _wctomb_l
wcstombs, _wcstombs_l

mbsinit

wcsrtombs_s

Convert a wide character string to its multibyte character string representation. A version of wcsrtombs with security enhancements as described in Security Enhancements in the CRT.

```
errno_t wcsrtombs_s(
   size_t *pReturnValue,
   char *mbstr,
   size_t sizeInBytes,
   const wchar_t **wcstr,
   sizeof count,
   mbstate_t *mbstate
);
template <size_t size>
errno_t wcsrtombs_s(
   size_t *pReturnValue,
   char (&mbstr)[size],
   const wchar_t **wcstr,
   sizeof count,
   mbstate_t *mbstate
); // C++ only
```

Parameters

[out] *pReturnValue*

The number of characters converted.

[out] *mbstr*

The address of a buffer for the resulting converted multibyte character string.

[out] *sizeInBytes*

The size in bytes of the *mbstr* buffer.

[in] *wcstr*

Points to the wide character string to be converted.

[in] *count*

The maximum number of bytes to be stored in the *mbstr* buffer, or _TRUNCATE.

[in] *mbstate*

A pointer to an **mbstate_t** conversion state object.

Return Value

Zero if successful, an error code on failure.

Error condition	Return value and errno
mbstr is **NULL** and *sizeInBytes* > 0	**EINVAL**
wcstr is **NULL**	**EINVAL**
The destination buffer is too small to contain the converted string (unless *count* is **_TRUNCATE**; see Remarks below)	**ERANGE**

If any of these conditions occurs, the invalid parameter exception is invoked as described in Parameter Validation . If execution is allowed to continue, the function returns an error code and sets **errno** as indicated in the table.

Remarks

The **wcsrtombs_s** function converts a string of wide characters pointed to by *wcstr* into multibyte characters stored in the buffer pointed to by *mbstr*, using the conversion state contained in *mbstate*. The conversion will continue for each character until one of these conditions is met:

- A null wide character is encountered
- A wide character that cannot be converted is encountered
- The number of bytes stored in the *mbstr* buffer equals *count*.

The destination string is always null-terminated (even in the case of an error).

If *count* is the special value _TRUNCATE, then **wcsrtombs_s** converts as much of the string as will fit into the destination buffer, while still leaving room for a null terminator.

If **wcsrtombs_s** successfully converts the source string, it puts the size in bytes of the converted string, including the null terminator, into **pReturnValue* (provided *pReturnValue* is not **NULL**). This occurs even if the *mbstr* argument is **NULL** and provides a way to determine the required buffer size. Note that if *mbstr* is **NULL**, *count* is ignored.

If **wcsrtombs_s** encounters a wide character it cannot convert to a multibyte character, it puts -1 in **pReturnValue*, sets the destination buffer to an empty string, sets *errno* to **EILSEQ**, and returns **EILSEQ**.

If the sequences pointed to by *wcstr* and *mbstr* overlap, the behavior of **wcsrtombs_s** is undefined. **wcsrtombs_s** is affected by the LC_TYPE category of the current locale.

Security Note
Ensure that *wcstr* and *mbstr* do not overlap, and that *count* correctly reflects the number of wide characters to convert.

The **wcsrtombs_s** function differs from wcstombs_s, _wcstombs_s_l by its restartability. The conversion state is stored in *mbstate* for subsequent calls to the same or other restartable functions. Results are undefined when mixing the use of restartable and nonrestartable functions. For example, an application would use **wcsrlen** rather than **wcslen**, if a subsequent call to **wcsrtombs_s** were used instead of **wcstombs_s.**

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Exceptions

The **wcsrtombs_s** function is multithread safe as long as no function in the current thread calls **setlocale** while this function is executing and the *mbstate* is null.

Example

```
// crt_wcsrtombs_s.cpp
//
// This code example converts a wide
// character string into a multibyte
// character string.
//

#include <stdio.h>
#include <memory.h>
#include <wchar.h>
#include <errno.h>

#define MB_BUFFER_SIZE 100

void main()
{
    const wchar_t   wcString[] =
                    {L"Every good boy does fine."};
    const wchar_t   *wcsIndirectString = wcString;
    char            mbString[MB_BUFFER_SIZE];
    size_t          countConverted;
    errno_t         err;
    mbstate_t       mbstate;
```

```
    // Reset to initial shift state
    ::memset((void*)&mbstate, 0, sizeof(mbstate));

    err = wcsrtombs_s(&countConverted, mbString, MB_BUFFER_SIZE,
                      &wcsIndirectString, MB_BUFFER_SIZE, &mbstate);
    if (err == EILSEQ)
    {
        printf( "An encoding error was detected in the string.\n" );
    }
    else
    {
        printf( "The string was successfully converted.\n" );
    }
}
```

Output

```
The string was successfully converted.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

Requirements

Routine	Required header	Compatibility
wcsrtombs_s	<wchar.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

See Also

Reference

Data Conversion
Locale
Interpretation of Multibyte-Character Sequences
wcrtomb
wcrtomb_s
wctomb, _wctomb_l
wcstombs, _wcstombs_l
mbsinit

wcstombs, _wcstombs_l

Converts a sequence of wide characters to a corresponding sequence of multibyte characters. These functions are deprecated because more secure versions are available; see wcstombs_s, _wcstombs_s_l.

```
size_t wcstombs(
   char *mbstr,
   const wchar_t *wcstr,
   size_t count
);
size_t _wcstombs_l(
   char *mbstr,
   const wchar_t *wcstr,
   size_t count,
   _locale_t locale
);
template <size_t size>
size_t wcstombs(
   char (&mbstr)[size],
   const wchar_t *wcstr,
   size_t count
); // C++ only
template <size_t size>
size_t _wcstombs_l(
   char (&mbstr)[size],
   const wchar_t *wcstr,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

mbstr

The address of a sequence of multibyte characters.

wcstr

The address of a sequence of wide characters.

count

The maximum number of bytes that can be stored in the multibyte output string.

locale

The locale to use.

Return Value

If **wcstombs** successfully converts the multibyte string, it returns the number of bytes written into the multibyte output string, excluding the terminating **NULL** (if any). If the *mbstr* argument is **NULL**, **wcstombs** returns the required size in bytes of the destination string. If **wcstombs** encounters a wide character it cannot convert to a multibyte character, it returns –1 cast to type **size_t** and sets **errno** to **EILSEQ**.

Remarks

The **wcstombs** function converts the wide-character string pointed to by *wcstr* to the corresponding multibyte characters and stores the results in the *mbstr* array. The *count* parameter indicates the maximum number of bytes that can be stored in the multibyte output string (that is, the size of *mbstr*). In general, it is not known how many bytes will be required when converting a wide-character string. Some wide characters will require only one byte in the output string; others require two. If there are two bytes in the multibyte output string for every wide character in the input string (including the wide character **NULL**), the result is guaranteed to fit.

If **wcstombs** encounters the wide-character null character (L'\0') either before or when *count* occurs, it converts it to an 8-bit 0 and stops. Thus, the multibyte character string at *mbstr* is null-terminated only if **wcstombs** encounters a wide-character null character during conversion. If the sequences pointed to by *wcstr* and *mbstr* overlap, the behavior of **wcstombs** is undefined.

If the *mbstr* argument is **NULL**, **wcstombs** returns the required size in bytes of the destination string.

wcstombs validates its parameters. If *wcstr* is **NULL**, or if *count* is greater than **INT_MAX**, this function invokes the invalid parameter handler, as described in Parameter Validation . If execution is allowed to continue, the function sets **errno** to **EINVAL** and returns -1.

wcstombs uses the current locale for any locale-dependent behavior; **_wcstombs_l** is identical except that it uses the locale passed in instead.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Requirements

Routine	Required header	Compatibility
wcstombs	<stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wcstombs_l	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

This program illustrates the behavior of the **wcstombs** function.

```
// crt_wcstombs.c
// compile with: /W3
// This example demonstrates the use
// of wcstombs, which converts a string
// of wide characters to a string of
// multibyte characters.

#include <stdlib.h>
#include <stdio.h>

#define BUFFER_SIZE 100

int main( void )
{
    size_t  count;
    char    *pMBBuffer = (char *)malloc( BUFFER_SIZE );
    wchar_t *pWCBuffer = L"Hello, world.";

    printf("Convert wide-character string:\n" );

    count = wcstombs(pMBBuffer, pWCBuffer, BUFFER_SIZE ); // C4996
    // Note: wcstombs is deprecated; consider using wcstombs_s instead
    printf("   Characters converted: %u\n",
            count );
    printf("    Multibyte character: %s\n\n",
           pMBBuffer );

    free(pMBBuffer);
}
```

Output

```
Convert wide-character string:
   Characters converted: 13
    Multibyte character: Hello, world.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Data Conversion
Locale
_mbclen, mblen, _mblen_l
mbstowcs, _mbstowcs_l
mbtowc, _mbtowc_l
wctomb, _wctomb_l
WideCharToMultiByte

wcstombs_s, _wcstombs_s_l

Converts a sequence of wide characters to a corresponding sequence of multibyte characters. A version of wcstombs, _wcstombs_l with security enhancements as described in Security Enhancements in the CRT.

```
errno_t wcstombs_s(
   size_t *pReturnValue,
   char *mbstr,
   size_t sizeInBytes,
   const wchar_t *wcstr,
   size_t count
);
errno_t _wcstombs_s_l(
   size_t *pReturnValue,
   char *mbstr,
   size_t sizeInBytes,
   const wchar_t *wcstr,
   size_t count,
   _locale_t locale
);
template <size_t size>
errno_t wcstombs_s(
   size_t *pReturnValue,
   char (&mbstr)[size],
   const wchar_t *wcstr,
   size_t count
); // C++ only
template <size_t size>
errno_t _wcstombs_s_l(
   size_t *pReturnValue,
   char (&mbstr)[size],
   const wchar_t *wcstr,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

[out] *pReturnValue*

The number of characters converted.

[out] *mbstr*

The address of a buffer for the resulting converted multibyte character string.

[in] *sizeInBytes*

The size in bytes of the *mbstr* buffer.

[in] *wcstr*

Points to the wide character string to be converted.

[in] *count*

The maximum number of bytes to be stored in the *mbstr* buffer, or _TRUNCATE.

[in] *locale*

The locale to use.

Return Value

Zero if successful, an error code on failure.

Error condition	Return value and errno

mbstr is **NULL** and *sizeInBytes* > 0	**EINVAL**
wcstr is **NULL**	**EINVAL**
The destination buffer is too small to contain the converted string (unless *count* is **_TRUNCATE**; see Remarks below)	**ERANGE**

If any of these conditions occurs, the invalid parameter exception is invoked as described in Parameter Validation . If execution is allowed to continue, the function returns an error code and sets **errno** as indicated in the table.

Remarks

The **wcstombs_s** function converts a string of wide characters pointed to by *wcstr* into multibyte characters stored in the buffer pointed to by *mbstr*. The conversion will continue for each character until one of these conditions is met:

- A null wide character is encountered
- A wide character that cannot be converted is encountered
- The number of bytes stored in the *mbstr* buffer equals *count*.

The destination string is always null-terminated (even in the case of an error).

If *count* is the special value _TRUNCATE, then **wcstombs_s** converts as much of the string as will fit into the destination buffer, while still leaving room for a null terminator.

If **wcstombs_s** successfully converts the source string, it puts the size in bytes of the converted string, including the null terminator, into **pReturnValue* (provided *pReturnValue* is not **NULL**). This occurs even if the *mbstr* argument is **NULL** and provides a way to determine the required buffer size. Note that if *mbstr* is **NULL**, *count* is ignored.

If **wcstombs_s** encounters a wide character it cannot convert to a multibyte character, it puts 0 in **pReturnValue*, sets the destination buffer to an empty string, sets *errno* to **EILSEQ**, and returns **EILSEQ**.

If the sequences pointed to by *wcstr* and *mbstr* overlap, the behavior of **wcstombs_s** is undefined.

Security Note
Ensure that *wcstr* and *mbstr* do not overlap, and that *count* correctly reflects the number of wide characters to convert.

wcstombs_s uses the current locale for any locale-dependent behavior; **_wcstombs_s_l** is identical to **wcstombs** except that it uses the locale passed in instead.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Requirements

Routine	Required header	Compatibility
wcstombs_s	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003.

For additional compatibility information, see Compatibility in the Introduction.

Example

This program illustrates the behavior of the **wcstombs_s** function.

```
// crt_wcstombs_s.c
// This example converts a wide character
// string to a multibyte character string.
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
```

```
#define BUFFER_SIZE 100

int main( void )
{
    size_t   i;
    char      *pMBBuffer = (char *)malloc( BUFFER_SIZE );
    wchar_t*pWCBuffer = L"Hello, world.";

    printf( "Convert wide-character string:\n" );

    // Conversion
    wcstombs_s(&i, pMBBuffer, (size_t)BUFFER_SIZE,
               pWCBuffer, (size_t)BUFFER_SIZE );

    // Output
    printf("   Characters converted: %u\n", i);
    printf("    Multibyte character: %s\n\n",
     pMBBuffer );

    // Free multibyte character buffer
    if (pMBBuffer)
    {
    free(pMBBuffer);
    }
}
```

Output

```
Convert wide-character string:
   Characters converted: 14
    Multibyte character: Hello, world.
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Data Conversion
Locale
_mbclen, mblen, _mblen_l
mbstowcs, _mbstowcs_l
mbtowc, _mbtowc_l
wctomb_s, _wctomb_s_l
WideCharToMultiByte

wctob

Determines if a wide character corresponds to a multibyte character and returns its multibyte character representation.

```
int wctob(
   wint_t wchar
);
```

Parameters

wchar

Value to translate.

Return Value

If **wctob** successfully converts a wide character, it returns its multibyte character representation, only if the multibyte character is exactly one byte long. If **wctob** encounters a wide character it cannot convert to a multibyte character or the multibyte character is not exactly one byte long, it returns a –1.

Remarks

The **wctob** function converts a wide character contained in *wchar* to the corresponding multibyte character passed by the return **int** value, if the multibyte character is exactly one byte long.

If **wctob** was unsuccessful and no corresponding multibyte character was found, the function sets **errno** to **EILSEQ** and returns -1.

Requirements

Routine	Required header	Compatibility
wctob	<wchar.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

This program illustrates the behavior of the **wcstombs** function.

```
// crt_wctob.c
#include <stdio.h>
#include <wchar.h>

int main( void )
{
    int     bChar = 0;
    wint_t  wChar = 0;

    // Set the corresponding wide character to exactly one byte.
    wChar = (wint_t)'A';

    bChar = wctob( wChar );
    if (bChar == WEOF)
    {
        printf( "No corresponding multibyte character was found.\n");
    }
    else
    {
        printf( "Determined the corresponding multibyte character to"
                " be \"%c\".\n", bChar);
    }
}
```

Output

```
Determined the corresponding multibyte character to be "A".
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Data Conversion
Locale
_mbclen, mblen, _mblen_l
mbstowcs, _mbstowcs_l
mbtowc, _mbtowc_l
wctomb, _wctomb_l
WideCharToMultiByte

wctomb, _wctomb_l

Convert a wide character to the corresponding multibyte character. These functions are deprecated because more secure versions are available; see wctomb_s, _wctomb_s_l.

```
int wctomb(
   char *mbchar,
   wchar_t wchar
);
int _wctomb_l(
   char *mbchar,
   wchar_t wchar,
   _locale_t locale
);
```

Parameters

mbchar

The address of a multibyte character.

wchar

A wide character.

Return Value

If **wctomb** converts the wide character to a multibyte character, it returns the number of bytes (which is never greater than **MB_CUR_MAX**) in the wide character. If *wchar* is the wide-character null character (L'\0'), **wctomb** returns 1. If the target pointer *mbchar* is NULL, **wctomb** returns 0. If the conversion is not possible in the current locale, **wctomb** returns –1 and **errno** is set to **EILSEQ**.

Remarks

The **wctomb** function converts its *wchar* argument to the corresponding multibyte character and stores the result at *mbchar*. You can call the function from any point in any program. **wctomb** uses the current locale for any locale-dependent behavior; **_wctomb_l** is identical to **wctomb** except that it uses the locale passed in instead.

wctomb validates its parameters. If *mbchar* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns -1.

Requirements

Routine	Required header	Compatibility
wctomb	<stdlib.h>	ANSI, Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

This program illustrates the behavior of the wctomb function.

```
// crt_wctomb.cpp
// compile with: /W3
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   int i;
   wchar_t wc = L'a';
   char *pmb = (char *)malloc( MB_CUR_MAX );
```

```
    printf( "Convert a wide character:\n" );
    i = wctomb( pmb, wc ); // C4996
    // Note: wctomb is deprecated; consider using wctomb_s
    printf( "   Characters converted: %u\n", i );
    printf( "   Multibyte character: %.1s\n\n", pmb );
  }
```

Output

```
Convert a wide character:
   Characters converted: 1
   Multibyte character: a
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Data Conversion
Locale
_mbclen, mblen, _mblen_l
mbstowcs, _mbstowcs_l
mbtowc, _mbtowc_l
wcstombs, _wcstombs_l
WideCharToMultiByte

wctomb_s, _wctomb_s_l

Converts a wide character to the corresponding multibyte character. A version of wctomb, _wctomb_l with security enhancements as described in Security Enhancements in the CRT.

```
errno_t wctomb_s(
   int *pRetValue,
   char *mbchar,
   size_t sizeInBytes,
   wchar_t wchar
);
errno_t _wctomb_s_l(
   int *pRetValue,
   char *mbchar,
   size_t sizeInBytes,
   wchar_t wchar,
   _locale_t locale
);
```

Parameters

[out] *pRetValue*

The number of bytes, or a code indicating the result.

[out] *mbchar*

The address of a multibyte character.

[in] *sizeInBytes*

Size of the buffer *mbchar*.

[in] *wchar*

A wide character.

[in] *locale*

The locale to use.

Return Value

Zero if successful, an error code on failure.

Error Conditions

mbchar	***sizeInBytes***	**Return value**	***pRetValue***
NULL	>0	**EINVAL**	not modified
any	>**INT_MAX**	**EINVAL**	not modified
any	too small	**EINVAL**	not modified

If any of the above error conditions occurs, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **wctomb** returns **EINVAL** and sets **errno** to **EINVAL**.

Remarks

The **wctomb_s** function converts its *wchar* argument to the corresponding multibyte character and stores the result at *mbchar*. You can call the function from any point in any program.

If **wctomb_s** converts the wide character to a multibyte character, it puts the number of bytes (which is never greater than **MB_CUR_MAX**) in the wide character into the integer pointed to by *pRetValue*. If *wchar* is the wide-character null character (L'\0'), **wctomb_s** fills *pRetValue* with 1. If the target pointer *mbchar* is NULL, **wctomb_s** puts 0 in *pRetValue*. If the conversion is not possible in the current locale, **wctomb_s** puts –1 in *pRetValue*.

wctomb_s uses the current locale for locale-dependent information; **_wctomb_s_l** is identical except that it uses the locale passed in instead.

Requirements

Routine	Required header	Compatibility
wctomb_s	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003
_wctomb_s_l	<stdlib.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

This program illustrates the behavior of the **wctomb** function.

```
// crt_wctomb_s.cpp
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int i;
    wchar_t wc = L'a';
    char *pmb = (char *)malloc( MB_CUR_MAX );

    printf_s( "Convert a wide character:\n" );
    wctomb_s( &i, pmb, MB_CUR_MAX, wc );
    printf_s( "   Characters converted: %u\n", i );
    printf_s( "   Multibyte character: %.1s\n\n", pmb );
}
```

Output

```
Convert a wide character:
   Characters converted: 1
   Multibyte character: a
```

.NET Framework Equivalent

Not applicable. To call the standard C function, use **PInvoke**. For more information, see Platform Invoke Examples.

See Also

Reference

Data Conversion
Locale
_mbclen, mblen, _mblen_l
mbstowcs, _mbstowcs_l
mbtowc, _mbtowc_l
wcstombs, _wcstombs_l
WideCharToMultiByte

wctrans

Determines a mapping from one set of character codes to another.

```
wctrans_t wctrans(
   const char *property
);
```

Parameters

property

A string that specifies one of the valid transformations.

Return Value

If the **LC_CTYPE** category of the current locale does not define a mapping whose name matches the property string *property*, the function returns zero. Otherwise, it returns a nonzero value suitable for use as the second argument to a subsequent call to towctrans.

Remarks

This function determines a mapping from one set of character codes to another.

The following pairs of calls have the same behavior in all locales, but it is possible to define additional mappings even in the "C" locale:

Function	Same As
tolower(*c* **)**	**towctrans(** *c* **, wctrans("towlower"))**
towupper(*c* **)**	**towctrans(** *c* **, wctrans("toupper"))**

Requirements

Routine	Required Header	Compatibility
wctrans	<wctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt_wctrans.cpp
// compile with: /EHsc
// This example determines a mapping from one set of character
// codes to another.

#include <wchar.h>
#include <wctype.h>
#include <stdio.h>
#include <iostream>

int main()
{
    wint_t c = 'a';
    printf_s("%d\n",c);

    wctrans_t i = wctrans("toupper");
    printf_s("%d\n",i);

    wctrans_t ii = wctrans("towlower");
    printf_s("%d\n",ii);

    wchar_t wc = towctrans(c, i);
```

```
    printf_s("%d\n",wc);
}
```

Output

```
97
1
0
65
```

See Also

Reference

Data Conversion

setlocale, _wsetlocale

wctype

Determines a classification rule for wide-character codes.

```
wctype_t wctype(
   const char * property
);
```

Parameters

property

Property string.

Return Value

If the **LC_CTYPE** category of the current locale does not define a classification rule whose name matches the property string *property*, the function returns zero. Otherwise, it returns a nonzero value suitable for use as the second argument to a subsequent call to towctrans.

Remarks

The function determines a classification rule for wide-character codes. The following pairs of calls have the same behavior in all locales (but an implementation can define additional classification rules even in the "C" locale):

Function	Same as
iswalnum(*c* **)**	**iswctype(** *c* **, wctype("alnum"))**
iswalpha(*c* **)**	**iswctype(** *c* **, wctype("alpha"))**
iswcntrl(*c* **)**	**iswctype(** *c* **, wctype("cntrl"))**
iswdigit(*c* **)**	**iswctype(** *c* **, wctype("digit"))**
iswgraph(*c* **)**	**iswctype(** *c* **, wctype("graph"))**
iswlower(*c* **)**	**iswctype(** *c* **, wctype("lower"))**
iswprint(*c* **)**	**iswctype(** *c* **, wctype("print"))**
iswpunct(*c* **)**	**iswctype(** *c* **, wctype("punct"))**
iswspace(*c* **)**	**iswctype(** *c* **, wctype("space"))**
iswupper(*c* **)**	**iswctype(** *c* **, wctype("upper"))**
iswxdigit(*c* **)**	**iswctype(** *c* **, wctype("xdigit"))**

Requirements

Routine	Required header	Compatibility
wctype	<wctype.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

See Also

Reference

write

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _write instead.

_write

Writes data to a file.

```
int _write(
   int fd,
   const void *buffer,
   unsigned int count
);
```

Parameters

fd

File descriptor of file into which data is written.

buffer

Data to be written.

count

Number of bytes.

Return Value

If successful, **_write** returns the number of bytes actually written. If the actual space remaining on the disk is less than the size of the buffer the function is trying to write to the disk, **_write** fails and does not flush any of the buffer's contents to the disk. A return value of –1 indicates an error. If invalid parameters are passed, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the function returns -1 and **errno** is set to one of three values: **EBADF**, which means the file descriptor is invalid or the file is not opened for writing; **ENOSPC**, which means there is not enough space left on the device for the operation; or **EINVAL**, which means that *buffer* was a null pointer.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

If the file is opened in text mode, each linefeed character is replaced with a carriage return – linefeed pair in the output. The replacement does not affect the return value.

Remarks

The **_write** function writes *count* bytes from *buffer* into the file associated with *fd*. The write operation begins at the current position of the file pointer (if any) associated with the given file. If the file is open for appending, the operation begins at the current end of the file. After the write operation, the file pointer is increased by the number of bytes actually written.

When writing to files opened in text mode, **_write** treats a CTRL+Z character as the logical end-of-file. When writing to a device, **_write** treats a CTRL+Z character in the buffer as an output terminator.

Requirements

Routine	Required header	Compatibility
_write	<io.h>	Windows 95, Windows 98, Windows 98 Second Edition, Windows Millennium Edition, Windows NT 4.0, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows Server 2003

For additional compatibility information, see Compatibility in the Introduction.

Example

```
// crt__write.c
//
// This program opens a file for output and uses _write to write
// some bytes to the file.

#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <share.h>

char buffer[] = "This is a test of '_write' function";

int main( void )
{
   int         fileHandle = 0;
   unsigned    bytesWritten = 0;

   if ( _sopen_s(&fileHandle, "write.o", _O_RDWR | _O_CREAT,
                 _SH_DENYNO, _S_IREAD | _S_IWRITE) )
      return -1;

   if (( bytesWritten = _write( fileHandle, buffer, sizeof( buffer ))) == -1 )
   {
      switch(errno)
      {
         case EBADF:
            perror("Bad file descriptor!");
            break;
         case ENOSPC:
            perror("No space left on device!");
            break;
         case EINVAL:
            perror("Invalid parameter: buffer was NULL!");
            break;
         default:
            // An unrelated error occured
            perror("Unexpected error!");
      }
   }
   else
   {
      printf_s( "Wrote %u bytes to file.\n", bytesWritten );
   }
   _close( fileHandle );
}
```

Output

```
Wrote 36 bytes to file.
```

See Also

Reference

Low-Level I/O
fwrite
_open, _wopen
_read

wcsicoll

This POSIX function is deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant _wcsicoll instead.

y0, y1, yn

These POSIX functions are deprecated beginning in Visual C++ 2005. Use the ISO C++ conformant Bessel Functions: _y0, _y1, _yn instead.

Generic-Text Mappings

To simplify writing code for international markets, generic-text mappings are defined in TCHAR.H for:

- Data types
- Constants and global variables
- Routine mappings

For more information, see Using Generic-Text Mappings. Generic-text mappings are Microsoft extensions that are not ANSI compatible.

See Also

Reference

Data Type Mappings

A Sample Generic-Text Program

Data Type Mappings

These data-type mappings are defined in TCHAR.H and depend on whether the constant **_UNICODE** or **_MBCS** has been defined in your program.

For related information, see Using TCHAR.H Data Types with _MBCS Code.

Generic-Text Data Type Mappings

Generic-text data type name	SBCS (_UNICODE, _MBCS not defined)	_MBCS defined	_UNICODE defined
_TCHAR	**char**	**char**	**wchar_t**
_tfinddata_t	**_finddata_t**	**_finddata_t**	**_wfinddata_t**
_tfinddata64_t	**__finddata64_t**	**__finddata64_t**	**__wfinddata64_t**
_tfinddatai64_t	**_finddatai64_t**	**_finddatai64_t**	**_wfinddatai64_t**
_TINT	**int**	**int**	**wint_t**
_TSCHAR	**signed char**	**signed char**	**wchar_t**
_TUCHAR	**unsigned char**	**unsigned char**	**wchar_t**
_TXCHAR	**char**	**unsigned char**	**wchar_t**
_T or **_TEXT**	No effect (removed by preprocessor)	No effect (removed by preprocessor)	**L** (converts following character or string to its Unicode counterpart)

See Also

Reference

Generic-Text Mappings
Constant and Global Variable Mappings
Routine Mappings
A Sample Generic-Text Program
Using Generic-Text Mappings

Constant and Global Variable Mappings

These generic-text constant, global variable, and standard-type mappings are defined in TCHAR.H and depend on whether the constant **_UNICODE** or **_MBCS** has been defined in your program.

Generic-Text Constant and Global Variable Mappings

Generic-text - object name	SBCS (_UNICODE, _MBCS not defined)	_MBCS defined	_UNICODE defined
_TEOF	**EOF**	**EOF**	**WEOF**
_tenviron	**_environ**	**_environ**	**_wenviron**
_tpgmptr	**_pgmptr**	**_pgmptr**	**_wpgmptr**

See Also

Reference

Generic-Text Mappings
Data Type Mappings
Routine Mappings
A Sample Generic-Text Program
Using Generic-Text Mappings

Routine Mappings

The generic-text routine mappings are defined in TCHAR.H. **_tccpy** and **_tclen** map to functions in the MBCS model; they are mapped to macros or inline functions in the SBCS and Unicode models for completeness. For information on a generic text routine, see the help topic about the corresponding **SBCS**-, **_MBCS**-, or **_UNICODE**-related routine.

More specific information about individual routines listed in the left column in the following table is not available in this documentation. However, you can easily look up the information on a corresponding **SBCS**-, **_MBCS**-, or **_UNICODE**-related routine. Use the **Search** command on the **Help** menu to look up any generic-text routine listed below.

For related information, see Generic-Text Mappings in TCHAR.H.

Generic-Text Routine Mappings

Generic-text routine name	SBCS (_UNICODE & MBCS not defined)	_MBCS defined	_UNICODE defined
_cgetts	_cgets	_cgets	_cgetws
_cgetts_s	_cgets_s	_cgets_s	_cgetws_s
_cputts	_cputs	_cputs	_cputws
_fgettc	fgetc	fgetc	fgetwc
_fgettchar	_fgetchar	_fgetchar	_fgetwchar
_fgetts	fgets	fgets	fgetws
_fputtc	fputc	fputc	fputwc
_fputtchar	_fputchar	_fputchar	_fputwchar
_fputts	fputs	fputs	fputws
_ftprintf	fprintf	fprintf	fwprintf
_ftprintf_s	fprintf_s	fprintf_s	fwprintf_s
_ftscanf	fscanf	fscanf	fwscanf
_ftscanf_s	fscanf_s	fscanf_s	fwscanf_s
_gettc	getc	getc	getwc
_gettch	_getch	_getch	_getwch
_gettchar	getchar	getchar	getwchar
_gettche	_getche	_getche	_getwche
_getts	gets	gets	getws
_getts_s	gets_s	gets_s	getws_s
_istalnum	isalnum	_ismbcalnum	iswalnum

_istalpha	**isalpha**	**_ismbcalpha**	**iswalpha**
_istascii	**isascii**	**isascii**	**iswascii**
_istcntrl	**iscntrl**	**iscntrl**	**iswcntrl**
_istdigit	**isdigit**	**_ismbcdigit**	**iswdigit**
_istgraph	**isgraph**	**_ismbcgraph**	**iswgraph**
_istlead	Always returns false	**_ismbblead**	Always returns false
_istleadbyte	Always returns false	**isleadbyte**	Always returns false
_istlegal	Always returns true	**_ismbclegal**	Always returns true
_istlower	**islower**	**_ismbclower**	**iswlower**
_istprint	**isprint**	**_ismbcprint**	**iswprint**
_istpunct	**ispunct**	**_ismbcpunct**	**iswpunct**
_istspace	**isspace**	**_ismbcspace**	**iswspace**
_istupper	**isupper**	**_ismbcupper**	**iswupper**
_istxdigit	**isxdigit**	**isxdigit**	**iswxdigit**
_itot	**_itoa**	**_itoa**	**_itow**
_itot_s	**_itoa_s**	**_itoa_s**	**_itow_s**
_ltot	**_ltoa**	**_ltoa**	**_ltow**
_ltot_s	**_ltoa_s**	**_ltoa_s**	**_ltow_s**
_puttc	**putc**	**putc**	**putwc**
_puttch	**_putch**	**_putch**	**_putwch**
_puttchar	**putchar**	**putchar**	**putwchar**
_putts	**puts**	**puts**	**_putws**
_sctprintf	**_scprintf**	**_scprintf**	**_scwprintf**
_sntprintf	**_snprintf**	**_snprintf**	**_snwprintf**
_sntprintf_s	**_snprintf_s**	**_snprintf_s**	**_snwprintf_s**
_sntscanf	**_snscanf**	**_snscanf**	**_snwscanf**
_sntscanf_s	**_snscanf_s**	**_snscanf_s**	**_snwscanf_s**

_stprintf	**sprintf**	**sprintf**	**swprintf**
_stprintf_s	**sprintf_s**	**sprintf_s**	**swprintf_s**
_stscanf	**sscanf**	**sscanf**	**swscanf**
_stscanf_s	**sscanf_s**	**sscanf_s**	**swscanf_s**
_taccess	**_access**	**_access**	**_waccess**
_taccess_s	**_access_s**	**_access_s**	**_waccess_s**
_tasctime	**asctime**	**asctime**	**_wasctime**
_tasctime_s	**asctime_s**	**asctime_s**	**_wasctime_s**
_tccmp	Maps to macro or inline function	**_mbsncmp**	Maps to macro or inline function
_tccpy	Maps to macro or inline function	**_mbccpy**	Maps to macro or inline function
_tccpy_s	**strcpy_s**	**_mbccpy_s**	**wcscpy_s**
_tchdir	**_chdir**	**_chdir**	**_wchdir**
_tclen	Maps to macro or inline function	**_mbclen**	Maps to macro or inline function
_tchmod	**_chmod**	**_chmod**	**_wchmod**
_tcprintf	**_cprintf**	**_cprintf**	**_cwprintf**
_tcprintf_s	**_cprintf_s**	**_cprintf_s**	**_cwprintf_s**
_tcreat	**_creat**	**_creat**	**_wcreat**
_tcscanf	**_cscanf**	**_cscanf**	**_cwscanf**
_tcscanf_s	**_cscanf_s**	**_cscanf_s**	**_cwscanf_s**
_tcscat	**strcat**	**_mbscat**	**wcscat**
_tcscat_s	**strcat_s**	**_mbscat_s**	**wcscat_s**
_tcschr	**strchr**	**_mbschr**	**wcschr**
_tcsclen	**strlen**	**_mbslen**	**wcslen**
_tcsclen_s	**strlen_s**	**_mbslen_s**	**wcslen_s**
_tcscmp	**strcmp**	**_mbscmp**	**wcscmp**
_tcscoll	**strcoll**	**_mbscoll**	**wcscoll**
_tcscpy	**strcpy**	**_mbscpy**	**wcscpy**

_tcscpy_s	**strcpy_s**	**_mbscpy_s**	**wcscpy_s**
_tcscspn	**strcspn**	**_mbscspn**	**wcscspn**
_tcsdec	**_strdec**	**_mbsdec**	**_wcsdec**
_tcsdup	**_strdup**	**_mbsdup**	**_wcsdup**
_tcserror	**strerror**	**strerror**	**_wcserror**
_tcserror_s	**strerror_s**	**strerror_s**	**_wcserror_s**
_tcsftime	**strftime**	**strftime**	**wcsftime**
_tcsicmp	**_stricmp**	**_mbsicmp**	**_wcsicmp**
_tcsicoll	**_stricoll**	**_mbsicoll**	**_wcsicoll**
_tcsinc	**_strinc**	**_mbsinc**	**_wcsinc**
_tcslen	**strlen**	**strlen**	**wcslen**
_tcslen_s	**strlen_s**	**strlen_s**	**wcslen_s**
_tcslwr	**_strlwr**	**_mbslwr**	**_wcslwr**
_tcslwr_s	**_strlwr_s**	**_mbslwr_s**	**_wcslwr_s**
_tcsnbcnt	**_strncnt**	**_mbsnbcnt**	**_wcsncnt**
_tcsncat	**strncat**	**_mbsnbcat**	**wcsncat**
_tcsncat_s	**strncat_s**	**_mbsnbcat_s**	**wcsncat_s**
_tcsnccat	**strncat**	**_mbsncat**	**wcsncat**
_tcsnccmp	**strncmp**	**_mbsncmp**	**wcsncmp**
_tcsnccmp_s	**strncmp_s**	**_mbsncmp_s**	**wcsncmp_s**
_tcsnccoll	**_strncoll**	**_mbsncoll**	**_wcsncoll**
_tcsncmp	**strncmp**	**_mbsnbcmp**	**wcsncmp**
_tcsnccnt	**_strncnt**	**_mbsnccnt**	**_wcsncnt**
_tcsnccpy	**strncpy**	**_mbsncpy**	**wcsncpy**
_tcsnccpy_s	**strncpy_s**	**_mbsncpy_s**	**wcsncpy_s**
_tcsncicmp	**_strnicmp**	**_mbsnicmp**	**_wcsnicmp**
_tcsncicoll	**_strnicoll**	**_mbsnicoll**	**_wcsnicoll**

_tcsncpy	**strncpy**	**_mbsnbcpy**	**wcsncpy**
_tcsncpy_s	**strncpy_s**	**_mbsnbcpy_s**	**wcsncpy_s**
_tcsncset	**_strnset**	**_mbsnset**	**_wcsnset**
_tcsnextc	**_strnextc**	**_mbsnextc**	**_wcsnextc**
_tcsnicmp	**_strnicmp**	**_mbsnbicmp**	**_wcsnicmp**
_tcsnicoll	**_strnicoll**	**_mbsnbicoll**	**_wcsnicoll**
_tcsninc	**_strninc**	**_mbsninc**	**_wcsninc**
_tcsnccnt	**_strncnt**	**_mbsnccnt**	**_wcsncnt**
_tcsnset	**_strnset**	**_mbsnbset**	**_wcsnset**
_tcspbrk	**strpbrk**	**_mbspbrk**	**wcspbrk**
_tcsspnp	**_strspnp**	**_mbsspnp**	**_wcsspnp**
_tcsrchr	**strrchr**	**_mbsrchr**	**wcsrchr**
_tcsrev	**_strrev**	**_mbsrev**	**_wcsrev**
_tcsset	**_strset**	**_mbsset**	**_wcsset**
_tcsspn	**strspn**	**_mbsspn**	**wcsspn**
_tcsstr	**strstr**	**_mbsstr**	**wcsstr**
_tcstod	**strtod**	**strtod**	**wcstod**
_tcstoi64	**_strtoi64**	**_strtoi64**	**_wcstoi64**
_tcstok	**strtok**	**_mbstok**	**wcstok**
_tcstok_s	**strtok_s**	**_mbstok_s**	**wcstok_s**
_tcstol	**strtol**	**strtol**	**wcstol**
_tcstoui64	**_strtoui64**	**_strtoui64**	**_wcstoui64**
_tcstoul	**strtoul**	**strtoul**	**wcstoul**
_tcsupr	**_strupr**	**_mbsupr**	**_wcsupr**
_tcsupr_s	**_strupr_s**	**_mbsupr_s**	**_wcsupr_s**
_tcsxfrm	**strxfrm**	**strxfrm**	**wcsxfrm**

_tctime	**ctime**	**ctime**	**_wctime**
_tctime_s	**ctime_s**	**ctime_s**	**_wctime_s**
_tctime32	**_ctime32**	**_ctime32**	**_wctime32**
_tctime32_s	**_ctime32_s**	**_ctime32_s**	**_wctime32_s**
_tctime64	**_ctime64**	**_ctime64**	**_wctime64**
_tctime64_s	**_ctime64_s**	**_ctime64_s**	**_wctime64_s**
_texecl	**_execl**	**_execl**	**_wexecl**
_texecle	**_execle**	**_execle**	**_wexecle**
_texeclp	**_execlp**	**_execlp**	**_wexeclp**
_texeclpe	**_execlpe**	**_execlpe**	**_wexeclpe**
_texecv	**_execv**	**_execv**	**_wexecv**
_texecve	**_execve**	**_execve**	**_wexecve**
_texecvp	**_execvp**	**_execvp**	**_wexecvp**
_texecvpe	**_execvpe**	**_execvpe**	**_wexecvpe**
_tfdopen	**_fdopen**	**_fdopen**	**_wfdopen**
_tfindfirst	**_findfirst**	**_findfirst**	**_wfindfirst**
_tfindnext	**_findnext**	**_findnext**	**_wfindnext**
_tfindnext32	**_findnext32**	**_findnext32**	**_wfindnext32**
_tfindnext64	**_findnext64**	**_findnext64**	**_wfindnext64**
_tfindnexti64	**_findnexti64**	**_findnexti64**	**_wfindnexti64**
_tfindnexti6432	**_findnexti6432**	**_findnexti6432**	**_wfindnexti6432**
_tfindnext32i64	**_findnext32i64**	**_findnext32i64**	**_wfindnext32i64**
_tfopen	**fopen**	**fopen**	**_wfopen**
_tfopen_s	**fopen_s**	**fopen_s**	**_wfopen_s**
_tfreopen	**freopen**	**freopen**	**_wfreopen**
_tfreopen_s	**freopen_s**	**freopen_s**	**_wfreopen_s**
_tfsopen	**_fsopen**	**_fsopen**	**_wfsopen**

_tfullpath	_fullpath	_fullpath	_wfullpath
_tgetcwd	_getcwd	_getcwd	_wgetcwd
_tgetdcwd	_getdcwd	_getdcwd	_wgetdcwd
_tgetenv	getenv	getenv	_wgetenv
_tgetenv_s	getenv_s	getenv_s	_wgetenv_s
_tmain	main	main	wmain
_tmakepath	_makepath	_makepath	_wmakepath
_tmakepath_s	_makepath_s	_makepath_s	_wmakepath_s
_tmkdir	_mkdir	_mkdir	_wmkdir
_tmktemp	_mktemp	_mktemp	_wmktemp
_tmktemp_s	_mktemp_s	_mktemp_s	_wmktemp_s
_topen	_open	_open	_wopen
_topen_s	_open_s	_open_s	_wopen_s
_totlower	tolower	_mbctolower	towlower
_totupper	toupper	_mbctoupper	towupper
_tperror	perror	perror	_wperror
_tpopen	_popen	_popen	_wpopen
_tprintf	printf	printf	wprintf
_tprintf_s	printf_s	printf_s	wprintf_s
_tputenv	_putenv	_putenv	_wputenv
_tputenv_s	_putenv_s	_putenv_s	_wputenv_s
_tremove	remove	remove	_wremove
_trename	rename	rename	_wrename
_trmdir	_rmdir	_rmdir	_wrmdir
_tsearchenv	_searchenv	_searchenv	_wsearchenv
_tsearchenv_s	_searchenv_s	_searchenv_s	_wsearchenv_s
_tscanf	scanf	scanf	wscanf

_tscanf_s	**scanf_s**	**scanf_s**	**wscanf_s**
_tsetlocale	**setlocale**	**setlocale**	**_wsetlocale**
_tsopen	**_sopen**	**_sopen**	**_wsopen**
_tsopen_s	**_sopen_s**	**_sopen_s**	**_wsopen_s**
_tspawnl	**_spawnl**	**_spawnl**	**_wspawnl**
_tspawnle	**_spawnle**	**_spawnle**	**_wspawnle**
_tspawnlp	**_spawnlp**	**_spawnlp**	**_wspawnlp**
_tspawnlpe	**_spawnlpe**	**_spawnlpe**	**_wspawnlpe**
_tspawnv	**_spawnv**	**_spawnv**	**_wspawnv**
_tspawnve	**_spawnve**	**_spawnve**	**_wspawnve**
_tspawnvp	**_spawnvp**	**_spawnvp**	**_wspawnvp**
_tspawnvpe	**_spawnvpe**	**_spawnvpe**	**_wspawnvpe**
_tsplitpath	**_splitpath**	**_splitpath**	**_wsplitpath**
_tstat	**_stat**	**_stat**	**_wstat**
_tstat32	**_stat32**	**_stat32**	**_wstat32**
_tstati32	**_stati32**	**_stati32**	**_wstati32**
_tstat64	**_stat64**	**_stat64**	**_wstat64**
_tstati64	**_stati64**	**_stati64**	**_wstati64**
_tstof	**atof**	**atof**	**_wtof**
_tstoi	**atoi**	**atoi**	**_wtoi**
_tstoi64	**_atoi64**	**_atoi64**	**_wtoi64**
_tstol	**atol**	**atol**	**_wtol**
_tstrdate	**_strdate**	**_strdate**	**_wstrdate**
_tstrdate_s	**_strdate_s**	**_strdate_s**	**_wstrdate_s**
_tstrtime	**_strtime**	**_strtime**	**_wstrtime**
_tstrtime_s	**_strtime_s**	**_strtime_s**	**_wstrtime_s**
_tsystem	**system**	**system**	**_wsystem**

_ttempnam	**_tempnam**	**_tempnam**	**_wtempnam**
_ttmpnam	**tmpnam**	**tmpnam**	**_wtmpnam**
_ttmpnam_s	**tmpnam_s**	**tmpnam_s**	**_wtmpnam_s**
_ttoi	**atoi**	**atoi**	**_wtoi**
_ttoi64	**_atoi64**	**_atoi64**	**_wtoi64**
_ttol	**atol**	**atol**	**_wtol**
_tunlink	**_unlink**	**_unlink**	**_wunlink**
_tutime	**_utime**	**_utime**	**_wutime**
_tutime32	**_utime32**	**_utime32**	**_wutime32**
_tutime64	**_utime64**	**_utime64**	**_wutime64**
_tWinMain	**WinMain**	**WinMain**	**wWinMain**
_ui64tot	**_ui64toa**	**_ui64toa**	**_ui64tow**
_ui64tot_s	**_ui64toa_s**	**_ui64toa_s**	**_ui64tow_s**
_ultot	**_ultoa**	**_ultoa**	**_ultow**
_ultot_s	**_ultoa_s**	**_ultoa_s**	**_ultow_s**
_ungettc	**ungetc**	**ungetc**	**ungetwc**
_ungettch	**_ungetch**	**_ungetch**	**_ungetwch**
_vftprintf	**vfprintf**	**vfprintf**	**vfwprintf**
_vftprintf_s	**vfprintf_s**	**vfprintf_s**	**vfwprintf_S**
_vsctprintf	**_vscprintf**	**_vscprintf**	**_vscwprintf**
_vsctprintf_s	**_vscprintf_s**	**_vscprintf_s**	**_vscwprintf_S**
_vsntprintf	**_vsnprintf**	**_vsnprintf**	**_vsnwprintf**
_vsntprintf_s	**_vsnprintf_s**	**_vsnprintf_s**	**_vsnwprintf_s**
_vstprintf	**vsprintf**	**vsprintf**	**vswprintf**
_vstprintf_s	**vsprintf_s**	**vsprintf_s**	**vswprintf_s**
_vtprintf	**vprintf**	**vprintf**	**vwprintf**

_vtprintf_s	**vprintf_s**	**vprintf_s**	**vwprintf_s**

See Also

Reference

Generic-Text Mappings
Data Type Mappings
Constant and Global Variable Mappings
A Sample Generic-Text Program
Using Generic-Text Mappings

Language and Country/Region Strings

The *locale* argument to the setlocale function takes the following form:

```
locale  "lang[_country_region[.code_page]]"
            | ".code_page"
            | ""
            | NULL
```

This appendix lists the language strings and Country/Region strings available to **setlocale**. For information on code pages, see Code Pages.

See Also

Other Resources

Run-Time Library Reference

Language Strings

The following language strings are recognized by **setlocale**. Any language not supported by the operating system is not accepted by **setlocale**. The three-letter language-string codes are only valid in Windows 98, Windows Me, Windows NT, Windows 2000, and Windows XP.

Primary language	Sublanguage	Language string
Chinese	Chinese	"chinese"
Chinese	Chinese (simplified)	"chinese-simplified" or "chs"
Chinese	Chinese (traditional)	"chinese-traditional" or "cht"
Czech	Czech	"csy" or "czech"
Danish	Danish	"dan" or "danish"
Dutch	Dutch (default)	"dutch" or "nld"
Dutch	Dutch (Belgium)	"belgian", "dutch-belgian", or "nlb"
English	English (default)	"english"
English	English (Australia)	"australian", "ena", or "english-aus"
English	English (Canada)	"canadian", "enc", or "english-can"
English	English (New Zealand)	"english-nz" or "enz"
English	English (United Kingdom)	"eng", "english-uk", or "uk"
English	English (United States)	"american", "american english", "american-english", "english-american", "english-us", "english-usa", "enu", "us", or "usa"
Finnish	Finnish	"fin" or "finnish"
French	French (default)	"fra" or "french"
French	French (Belgium)	"frb" or "french-belgian"
French	French (Canada)	"frc" or "french-canadian"
French	French (Switzerland)	"french-swiss" or "frs"
German	German (default)	"deu" or "german"
German	German (Austria)	"dea" or "german-austrian"

German	German (Switzerland)	"des", "german-swiss", or "swiss"
Greek	Greek	"ell" or "greek"
Hungarian	Hungarian	"hun" or "hungarian"
Icelandic	Icelandic	"icelandic" or "isl"
Italian	Italian (default)	"ita" or "italian"
Italian	Italian (Switzerland)	"italian-swiss" or "its"
Japanese	Japanese	"japanese" or "jpn"
Korean	Korean	"kor" or "korean"
Norwegian	Norwegian (default)	"norwegian"
Norwegian	Norwegian (Bokmal)	"nor" or "norwegian-bokmal"
Norwegian	Norwegian (Nynorsk)	"non" or "norwegian-nynorsk"
Polish	Polish	"plk" or "polish"
Portuguese	Portuguese (default)	"portuguese" or "ptg"
Portuguese	Portuguese (Brazil)	"portuguese-brazilian" or "ptb"
Russian	Russian (default)	"rus" or "russian"
Slovak	Slovak	"sky" or "slovak"
Spanish	Spanish (default)	"esp" or "spanish"
Spanish	Spanish (Mexico)	"esm" or "spanish-mexican"
Spanish	Spanish (Modern)	"esn" or "spanish-modern"
Swedish	Swedish	"sve" or "swedish"
Turkish	Turkish	"trk" or "turkish"

See Also

Reference

Language and Country/Region Strings

Country/Region Strings

The following is a list of Country/Region strings recognized by **setlocale**. Strings for countries/regions that are not supported by the operating system are not accepted by **setlocale**. Three-letter Country/Region-name codes are from the International Organization for Standardization and International Electrotechnical Commission (ISO/IEC) specification 3166.

Country/Region	Country/Region string
Australia	"aus" or "australia"
Austria	"aut" or "austria"
Belgium	"bel" or "belgium"
Brazil	"bra" or "brazil"
Canada	"can" or "canada"
China	"china", "chn", "pr china", or "pr-china"
Czech Republic	"cze" or "czech"
Denmark	"dnk" or "denmark"
Finland	"fin" or "finland"
France	"fra" or "france"
Germany	"deu" or "germany"
Greece	"grc" or "greece"
Hong Kong SAR	"hkg", "hong kong", or "hong-kong"
Hungary	"hun" or "hungary"
Iceland	"iceland" or "isl"
Ireland	"irl" or "ireland"
Italy	"ita" or "italy"
Japan	"jpn" or "japan"
Korea	"kor" or "korea"
Mexico	"mex" or "mexico"
The Netherlands	"nld", "holland", or "netherlands"
New Zealand	"nzl", "new zealand", "new-zealand", or "nz"
Norway	"nor" or "norway"

Poland	"pol" or "poland"
Portugal	"prt" or "portugal"
Russia	"rus" or "russia"
Singapore	"sgp" or "singapore"
Slovakia	"svk" or "slovak"
Spain	"esp" or "spain"
Sweden	"swe" or "sweden"
Switzerland	"che" or "switzerland"
Taiwan	"twn" or "taiwan"
Turkey	"tur" or "turkey"
United Kingdom	"gbr", "britain", "england", "great britain", "uk", "united kingdom", or "united-kingdom"
United States	"usa", "america", "united states", "united-states", or "us"

See Also

Reference

Language and Country/Region Strings

Standard C++ Library Reference

A C++ program can call on a large number of functions from this conforming implementation of the Standard C++ Library. These functions perform essential services such as input and output and provide efficient implementations of frequently used operations.

For more information about Visual C++ run-time libraries, see C Run-Time Libraries.

In This Section

Breaking Changes in the Standard C++ Library

This section describes the breaking changes in Visual C++ 2005.

Standard C++ Library Overview

Provides an overview of the Microsoft implementation of the Standard C++ Library.

iostream Programming

Provides an overview of iostream programming.

Standard C++ Library Header Files

Provides links to reference topics discussing the Standard C++ Library header files.

Standard Template Library Samples

Provides links to sample code showing how to use the Standard Template Library.

Related Sections

Standard C++ Library Changes

Describes changes to the Standard C++ Library implementation.

Visual C++ Reference

Provides links to topics describing the C and C++ language references, the libraries provided with Visual C++, the Visual C++ Extensibility Object Model, and the Microsoft Macro Assembler (MASM).

Visual C++ Samples

Provides links to sample code showing the capabilities of Visual C++ and the libraries and technologies it supports.

See Also

Other Resources

Visual C++ Libraries Reference

Breaking Changes in the Standard C++ Library

This section describes the breaking changes in Visual C++ 2005.

The exception Class

The exception class (located in the <exception> header) has been moved to the **std** namespace. In previous versions, this class was in the global namespace.

To resolve any errors indicating that the **exception** class cannot be found, add the following **using** statement to your code:

```
using namespace std;
```

Changes to valarray::resize()

When calling valarray::resize**()**, the contents of the **valarray** will be lost and will be replaced by default values. The **resize()** method is intended to reinitialize the **valarray** rather than grow it dynamically like a **vector**.

Debug Iterators

Applications built with a debug version of the C-Runtime Library and which use iterators incorrectly might begin to see asserts at runtime. To disable these asserts, you must define **_HAS_ITERATOR_DEBUGGING** to **0**. For more information, see Debug Iterator Support.

See Also

Concepts

What's New in Visual C++

New Standard C++ Library Features

Other Resources

Standard C++ Library Reference

Previous Product Changes

Standard C++ Library Overview

All C++ library entities are declared or defined in one or more standard headers. To make use of a library entity in a program, write an include directive that names the relevant standard header. The Standard C++ Library consists of 50 required headers. This implementation also includes two additional headers, **<hash_map>** and **<hash_set>**, that are not required by the C++ Standard, for a total of 52 headers. These 52 C++ library headers (with the additional 18 Standard C headers) constitute a hosted implementation of the C++ library.

<algorithm>	<bitset>	<cassert>
<cctype>	<cerrno>	<cfloat>
<ciso646>	<climits>	<clocale>
<cmath>	<complex>	<csetjmp>
<csignal>	<cstdarg>	<cstddef>
<cstdio>	<cstdlib>	<cstring>
<ctime>	<cwchar>	<cwctype>
<deque>	<exception>	<fstream>
<functional>	<hash_map>	<hash_set>
<iomanip>	<ios>	<iosfwd>
<iostream>	<iso646.h>	<istream>
<iterator>	<limits>	<list>
<locale>	<map>	<memory>
<new>	<numeric>	<ostream>
<queue>	<set>	<sstream>
<stack>	<stdexcept>	<streambuf>
<string>	<strstream>	<utility>
<valarray>	<vector>	

A freestanding implementation of the C++ library provides only a subset of these headers:

<cstddef>	<cstdlib> (declaring at least the functions **abort**, **atexit**, and **exit**)
<exception>	<limits>
<new>	<cstdarg>

The C++ library headers have two broader subdivisions:

- iostreams conventions.
- Standard Template Library conventions.

This section contains the following sections:

- Using C++ Library Headers
- C++ Library Conventions
- iostreams Conventions
- C++ Program Startup and Termination
- Safe Libraries: Standard C++ Library
- Checked Iterators
- Debug Iterator Support
- Standard Template Library
- Thread Safety in the Standard C++ Library
- The stdext Namespace

For more information about Visual C++ run-time libraries, see C Run-Time Libraries.

See Also

Other Resources

Standard C++ Library Reference

Using C++ Library Headers

You include the contents of a standard header by naming it in an include directive.

```
#include <iostream>   // include I/O facilities
```

You can include the standard headers in any order, a standard header more than once, or two or more standard headers that define the same macro or the same type. Do not include a standard header within a declaration. Do not define macros that have the same names as keywords before you include a standard header.

A C++ library header includes any other C++ library headers it needs to define needed types. (Always include explicitly any C++ library headers needed in a translation unit, however, lest you guess wrong about its actual dependencies.) A Standard C header never includes another standard header. A standard header declares or defines only the entities described for it in this document.

Every function in the library is declared in a standard header. Unlike in Standard C, the standard header never provides a masking macro with the same name as the function that masks the function declaration and achieves the same effect. For more information on masking macros, see C++ Library Conventions.

All names other than **operator delete** and **operator new** in the C++ library headers are defined in the **std** namespace, or in a namespace nested within the **std** namespace. You refer to the name **cin**, for example, as **std::cin**. Note, however, that macro names are not subject to namespace qualification, so you always write **__STD_COMPLEX** without a namespace qualifier.

In some translation environments, including a C++ library header may hoist external names declared in the **std** namespace into the global namespace as well, with individual **using** declarations for each of the names. Otherwise, the header does *not* introduce any library names into the current namespace.

The C++ Standard requires that the C Standard headers declare all external names in namespace **std**, then hoist them into the global namespace with individual **using** declarations for each of the names. But in some translation environments the C Standard headers include no namespace declarations, declaring all names directly in the global namespace. Thus, the most portable way to deal with namespaces is to follow two rules:

- To assuredly declare in namespace **std** an external name that is traditionally declared in <stdlib.h>, for example, include the header <cstdlib>. Know that the name might also be declared in the global namespace.
- To assuredly declare in the global namespace an external name declared in <stdlib.h>, include the header <stdlib.h> directly. Know that the name might also be declared in namespace **std**.

Thus, if you want to call **std::abort** to cause abnormal termination, you should include <cstdlib>. If you want to call **abort**, you should include <stdlib.h>.

Alternatively, you can write the declaration:

```
using namespace std;
```

which brings all library names into the current namespace. If you write this declaration immediately after all include directives, you hoist the names into the global namespace. You can subsequently ignore namespace considerations in the remainder of the translation unit. You also avoid most dialect differences across different translation environments.

Unless specifically indicated otherwise, you may not define names in the **std** namespace, or in a namespace nested within the **std** namespace, within your program.

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

C++ Library Conventions

The C++ library obeys much the same conventions as the Standard C Library, plus a few more outlined here.

An implementation has certain latitude in how it declares types and functions in the C++ library:

- Names of functions in the Standard C library may have either extern #"C++" or extern "C" linkage. Include the appropriate Standard C header rather than declare a library entity inline.
- A member function name in a library class may have additional function signatures over those listed in this document. You can be sure that a function call described here behaves as expected, but you cannot reliably take the address of a library member function. (The type may not be what you expect.)
- A library class may have undocumented (nonvirtual) base classes. A class documented as derived from another class may, in fact, be derived from that class through other undocumented classes.
- A type defined as a synonym for some integer type may be the same as one of several different integer types.
- A bitmask type can be implemented as either an integer type or an enumeration. In either case, you can perform bitwise operations (such as **AND** and **OR**) on values of the same bitmask type. The elements *A* and *B* of a bitmask type are nonzero values such that *A* & *B* is zero.
- A library function that has no exception specification can throw an arbitrary exception, unless its definition clearly restricts such a possibility.

On the other hand, there are some restrictions:

- The Standard C Library uses no masking macros. Only specific function signatures are reserved, not the names of the functions themselves.
- A library function name outside a class will not have additional, undocumented, function signatures. You can reliably take its address.
- Base classes and member functions described as virtual are assuredly virtual, while those described as nonvirtual are assuredly nonvirtual.
- Two types defined by the C++ library are always different unless this document explicitly suggests otherwise.
- Functions supplied by the library, including the default versions of replaceable functions, can throw *at most* those exceptions listed in any exception specification. No destructors supplied by the library throw exceptions. Functions in the Standard C Library may propagate an exception, as when **qsort** calls a comparison function that throws an exception, but they do not otherwise throw exceptions.

See Also
Reference
Standard C++ Library Overview
Thread Safety in the Standard C++ Library

iostreams Conventions

The iostreams headers support conversions between text and encoded forms, and input and output to external files:

<fstream>	<iomanip>
<ios>	<iosfwd>
<iostream>	<istream>
<ostream>	<sstream>
<streambuf>	<strstream>

The simplest use of iostreams requires only that you include the header <iostream>. You can then extract values from cin, to read the standard input. The rules for doing so are outlined in the description of the class basic_istream Class. You can also insert values to cout, to write the standard output. The rules for doing so are outlined in the description of the class basic_ostream Class. Format control common to both extractors and insertors is managed by the class basic_ios Class. Manipulating this format information in the guise of extracting and inserting objects is the province of several manipulators.

You can perform the same iostreams operations on files that you open by name, using the classes declared in <fstream>. To convert between iostreams and objects of class basic_string Class, use the classes declared in <sstream>. To do the same with C strings, use the classes declared in <strstream>.

The remaining headers provide support services, typically of direct interest to only the most advanced users of the iostreams classes.

See Also

Reference

Standard C++ Library Overview
iostream Programming
Thread Safety in the Standard C++ Library

C++ Program Startup and Termination

A C++ program performs the same operations as a C program does at program startup and at program termination, plus a few more outlined here.

Before the target environment calls the function **main**, and after it stores any constant initial values you specify in all objects that have static duration, the program executes any remaining constructors for such static objects. The order of execution is not specified between translation units, but you can nevertheless assume that some iostreams objects are properly initialized for use by these static constructors. These control text streams are:

- cin — for standard input.
- cout — for standard output.
- cerr — for unbuffered standard error output.
- clog — for buffered standard error output.

You can also use these objects within the destructors called for static objects, during program termination.

As with C, returning from **main** or calling **exit** calls all functions registered with **atexit** in reverse order of registry. An exception thrown from such a registered function calls **terminate**.

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

Safe Libraries: Standard C++ Library

Beginning with Visual C++ 2005, several enhancements have been made to the libraries that ship with Visual C++, including the Standard C++ Library, to make them more secure.

Several methods in the Standard C++ Library have been identified as potentially unsafe because they could lead to a buffer overrun or other code defect. The use of these methods is discouraged, and new, safer methods have been created to replace them. These new methods all end in **_s**.

Several enhancements have also been made to make iterators and algorithms more secure. For more information, see Checked Iterators and Debug Iterator Support.

Remarks

The following table lists the Standard C++ Library methods that are potentially unsafe, as well as their safer equivalent:

Potentially unsafe method	**Safer equivalent**
basic_string::copy	basic_string::_Copy_s
basic_istream::read	basic_istream::_Read_s
basic_istream::readsome	basic_istream::_Readsome_s
basic_streambuf::sgetn	basic_streambuf::_Sgetn_s
basic_streambuf::xsgetn	basic_streambuf::_Xsgetn_s
char_traits::copy	char_traits::_Copy_s
char_traits::move	char_traits::_Move_s
ctype::narrow	ctype::_Narrow_s
ctype::do_narrow	ctype::_Do_narrow_s
ctype::widen	ctype::_Widen_s
ctype::do_widen	ctype::_Do_widen_s

If you call any one of the potentially unsafe methods above, or if you use iterators incorrectly, the compiler will generate Compiler Warning (level 1) C4996. For information on how to disable these warnings, see _SCL_SECURE_NO_WARNINGS.

In This Section

_SCL_SECURE_NO_WARNINGS

Checked Iterators

Debug Iterator Support

See Also

Reference

Standard C++ Library Overview

_SCL_SECURE_NO_WARNINGS

Calling any one of the potentially unsafe methods in the Standard C++ Library will result in Compiler Warning (level 1) C4996. To disable this warning, define the macro **_SCL_SECURE_NO_WARNINGS** in your code:

```
#define _SCL_SECURE_NO_WARNINGS
```

Remarks

Other ways to disable warning C4996 include:

- Using the /D (Preprocessor Definitions) compiler option:

  ```
  cl /D_SCL_SECURE_NO_WARNINGS [other compiler options] myfile.cpp
  ```

- Using the /w compiler option:

  ```
  cl /wd4996 [other compiler options] myfile.cpp
  ```

- Using the #pragma warning directive:

  ```
  #pragma warning(disable:4996)
  ```

Also, you can manually change the level of warning C4996 with the **/w<l><n>** compiler option. For example, to set warning C4996 to level 4:

```
cl /w44996 [other compiler options] myfile.cpp
```

For more information, see /w, /Wn, /WX, /Wall, /wln, /wdn, /wen, /won (Warning Level).

See Also

Reference

Safe Libraries: Standard C++ Library

Checked Iterators

Checked iterators ensure that you do not overwrite the bounds of your container.

Checked iterators apply to release builds and debug builds. See Debug Iterator Support for more information on using iterators when compiling in debug mode.

Remarks

For information on how to disable warnings generated by checked iterators, see _SCL_SECURE_NO_WARNINGS.

The following symbols are for use with the checked iterators feature.

_SECURE_SCL

If defined as 1, unsafe iterator use causes a runtime error. If defined as 0, checked iterators are disabled. The exact behavior of the runtime error depends on the value of **_SECURE_SCL_THROWS**. The default value for **_SECURE_SCL** is 1, meaning checked iterators are enabled by default.

_SECURE_SCL_THROWS

If defined as 1, an out of range iterator use causes an exception at runtime. If defined as 0, the program is terminated by calling invalid_parameter. The default value for **_SECURE_SCL_THROWS** is 0, meaning the program will be terminated by default. Requires **_SECURE_SCL** to also be defined.

When **_SECURE_SCL**=1 is defined:

- All standard iterators (**vector::iterator**, for example) are checked.
- The checked form of an algorithm will be used, for standard functions with checked forms (see list below).
- If an output iterator is a checked iterator:
  - You will get checked behavior on calls to the standard function (**std::copy**, for example).
  - You will get checked behavior on calls to a checked function (**stdext::checked_copy**, for example).
  - You will get checked behavior on calls to an unchecked function (**stdext::unchecked_copy**, for example).
- If the output iterator is an unchecked iterator (an array, for example):
  - Calls to the standard function (**std::copy**, for example) will result in compiler warnings.
  - Calls to the checked function (**stdext::checked_copy**, for example) will result in compiler warnings.
  - You will get unchecked behavior on calls to an unchecked function (**stdext::unchecked_copy**, for example).
- The following functions will generate a runtime error if there is an access outside the bounds of the container:

basic_string::operator[]	bitset::operator[]	deque::back	deque::front
deque::operator[]	list::back	list::front	queue::back
queue::front	valarray::operator[]	vector::back	vector::front
vector::operator[]			

When **_SECURE_SCL**=0 is defined:

- All standard iterators are unchecked (iterators can move beyond the container boundaries, leading to undefined behavior).
- The unchecked form of a function will be used, for standard functions with checked forms (see list below).
- If an output iterator is a checked iterator:
  - You will get checked behavior on calls to the standard function (**std::copy**, for example).

    - You will get checked behavior on calls to a checked function (**stdext::checked_copy**, for example).
    - You will get checked behavior on calls to an unchecked function (**stdext::unchecked_copy**, for example).
- If an output iterator is an unchecked iterator:
    - You will get unchecked behavior on calls to the standard function (**std::copy**, for example).
    - Calls to a checked function (**stdext::checked_copy**, for example) will result in compiler warnings.
    - You will get unchecked behavior on calls to an unchecked function (**stdext::unchecked_copy**, for example).

A checked iterator refers to an iterator that will throw an exception or call invalid_parameter if you attempt to move past the boundaries of the container. For more information about invalid_parameter, see Parameter Validation.

A checked algorithm enforces the use of a checked destination iterator. If passed an unchecked destination iterator, a checked algorithm will compile, but with warnings. See _SCL_SECURE_NO_WARNINGS for information on how to disable these warnings.

There are two iterator adaptors that support checked iterators:

- checked_array_iterator Class
- checked_iterator Class

The following algorithms enforce the use of a checked iterator as output iterator. This is useful when you want to compile with **_SECURE_SCL**=0, and where you identify some code where you want to enforce the use of checked iterators. The following algorithms are all defined in the **stdext** namespace.

Note
The following algorithms are Microsoft extensions to the Standard C++ Library. Code implemented using these algorithms will not be portable.

checked_adjacent_difference	checked_copy	checked_copy_backward
checked_fill_n	checked_generate_n	checked_merge
checked_partial_sum	checked_remove_copy	checked_remove_copy_if
checked_replace_copy	checked_replace_copy_if	checked_reverse_copy
checked_rotate_copy	checked_set_difference	checked_set_intersection
checked_set_symmetric_difference	checked_set_union	checked_uninitialized_copy
checked_uninitialized_fill_n	checked_unique_copy	

The following algorithms allow the use of an unchecked iterator as output iterator. This is useful when you want to compile with **_SECURE_SCL**=1, and where you want to selectively allow the use of unchecked iterators. The following algorithms are all defined in the **stdext** namespace.

Note
The following algorithms are Microsoft extensions to the Standard C++ Library. Code implemented using these algorithms will not be portable.

unchecked_adjacent_difference	unchecked_copy	unchecked_copy_backward
unchecked_fill_n	unchecked_generate_n	unchecked_merge
unchecked_partial_sum	unchecked_remove_copy	unchecked_remove_copy_if

unchecked_replace_copy	unchecked_replace_copy_if	unchecked_reverse_copy
unchecked_rotate_copy	unchecked_set_difference	unchecked_set_intersection
unchecked_set_symmetric_difference	unchecked_set_union	unchecked_uninitialized_copy
unchecked_uninitialized_fill_n	unchecked_unique_copy	

Example

When compiling with **_SECURE_SCL** 1, a runtime error will occur if you attempt to access an element outside the bounds of the container with the indexing operator of certain classes (see above).

```
// checked_iterators_1.cpp
// compile with: /EHsc
#define _SECURE_SCL 1
#define _SECURE_SCL_THROWS 1
#include <vector>
#include <iostream>
using namespace std;
int main() {
   vector<int> v;
   v.push_back(67);
   int i = v[0];
   cout << i << endl;

   try {
      i = v[1];
   }
   catch (std::out_of_range) {
      cout << "invalid container access" << endl;
   }
};
```

Output

```
67
invalid container access
```

When compiling with **_SECURE_SCL** 1, a runtime error will occur if you attempt to access an element with front or back of certain classes (see above), when the container is empty.

```
// checked_iterators_2.cpp
// compile with: /EHsc
#define _SECURE_SCL 1
#define _SECURE_SCL_THROWS 1

#include <vector>
#include <iostream>

int main() {
   using namespace std;
   vector <int> v1;
   try {
      int& i = v1.front();
   }
   catch (std::out_of_range) {
      cout << "vector is empty!!" << endl;
   }
}
```

Output

```
vector is empty!!
```

When compiling with **_SECURE_SCL** 1, the compiler enforces the use of a checked iterator on standard algorithms that also have checked and unchecked versions (see above).

```
// checked_iterators_3.cpp
// compile with: /EHsc /W3
#define _SECURE_SCL 1
#include <algorithm>
#include <iostream>
using namespace std;
using namespace stdext;
int main() {
    int a[] = { 1, 2, 3 };
    int *b = new int[10];
    int c[10];

    copy(a, a + 3, b);   // C4996 unchecked iterator
    copy(a, a + 3, checked_array_iterator<int*>(c, _countof(c)));  // OK

    delete[] b;
}
```

The following example sets **_SECURE_SCL** to 0, thus disabling checked iterators. With checked iterators disabled, you must take extra caution to ensure you do not iterate beyond the bounds of your container.

```
// checked_iterators_4.cpp
// compile with: /EHsc
#define _SECURE_SCL 0
#include <vector>
#include <iostream>
using namespace std;
int main() {
   vector<int> v;
   v.push_back(67);
   int i = v[0];
   cout << i << endl;
};
```

Output

```
67
```

See Also

Reference

Standard C++ Library Overview

Debug Iterator Support

_SECURE_SCL

Defines whether Checked Iterators are enabled. If defined as 1, unsafe iterator use causes a runtime error. If defined as 0, checked iterators are disabled. The exact behavior of the runtime error depends on the value of _SECURE_SCL_THROWS. The default value for **_SECURE_SCL** is 1, meaning checked iterators are enabled by default.

Remarks

To enable checked iterators, set **_SECURE_SCL** to 1:

```
#define _SECURE_SCL 1
```

To disable checked iterators, set **_SECURE_SCL** to 0:

```
#define _SECURE_SCL 0
```

For information on how to disable warnings about checked iterators, see _SCL_SECURE_NO_WARNINGS.

See Also

Reference

Checked Iterators

Debug Iterator Support

Safe Libraries: Standard C++ Library

_SECURE_SCL_THROWS

Defines whether incorrect use of Checked Iterators causes an exception or a program termination. If defined as 1, an out of range iterator use causes an exception at runtime. If defined as 0, the program is terminated by calling **invalid_parameter**. The default value for **_SECURE_SCL_THROWS** is 0, meaning the program will be terminated by default. Requires _SECURE_SCL to also be defined.

Remarks

To enable checked iterators to throw an exception on incorrect use, set **_SECURE_SCL** to 1 and **_SECURE_SCL_THROWS** to 1:

```
#define _SECURE_SCL 1
#define _SECURE_SCL_THROWS 1
```

To enable checked iterators to terminate your program on incorrect use, set **_SECURE_SCL** to 1 and **_SECURE_SCL_THROWS** to 0:

```
#define _SECURE_SCL 1
#define _SECURE_SCL_THROWS 0
```

To disable checked iterators, set **_SECURE_SCL** to 0.

See Also

Reference

Checked Iterators
Debug Iterator Support
Safe Libraries: Standard C++ Library

Debug Iterator Support

The Visual C++ runtime library now detects incorrect iterator use and will assert and display a dialog box at run time. To enable debug iterator support, a program must be compiled with a debug version of a C run time library (see C Run-Time Libraries for more information).

See Checked Iterators for more information on using iterators.

The C++ standard describes which member functions cause iterators to a container to become invalid. Two examples are:

- Erasing an element from a container causes iterators to the element to become invalid.
- Increasing the size of a **vector** (push or insert) causes iterators into the **vector** container become invalid.

Example

Compile the following program in debug mode and at runtime it will assert and terminate.

```
/* compile with /D_DEBUG /EHsc /MDd */
#include <vector>
#include <iostream>

int main() {
   std::vector<int> v ;

   v.push_back(10);
   v.push_back(15);
   v.push_back(20);

   std::vector<int>::iterator i = v.begin();
   ++i;

   std::vector<int>::iterator j = v.end();
   --j;

   std::cout<<*j<<'\n';

   v.insert(i,25);

   std::cout<<*j<<'\n'; // Using an old iterator after an insert
}
```

The symbol _HAS_ITERATOR_DEBUGGING can be used to turn off the iterator debugging feature in a debug build. The following program will not assert:

```
// iterator_debugging.cpp
// compile with: /D_DEBUG /EHsc /MDd
#define _HAS_ITERATOR_DEBUGGING 0
#include <vector>
#include <iostream>

int main() {
   std::vector<int> v ;

   v.push_back(10);
   v.push_back(15);
   v.push_back(20);

   std::vector<int>::iterator i = v.begin();
   ++i;

   std::vector<int>::iterator j = v.end();
   --j;

   std::cout<<*j<<'\n';
```

```
   v.insert(i,25);

   std::cout<<*j<<'\n'; // Using an old iterator after an insert
}
```

Output

```
20
-572662307
```

An assert will also occur if you attempt to use an iterator before it is initialized:

```
/* compile with /EHsc /MDd */
#include <string>
using namespace std;
int main() {
   string::iterator i1, i2;
   if (i1 == i2)
      ;
}
```

The following code example will cause an assertion because the two iterators to the **for_each** algorithm are incompatible. All algorithms check to see that the iterators supplied to them are referencing the same container.

```
/* compile with /D_DEBUG /EHsc /MDd */
#include <algorithm>
#include <vector>
using namespace std;

// The function object multiplies an element by a Factor
template <class Type>
class MultValue
{
private:
    Type Factor;   // The value to multiply by
public:
    // Constructor initializes the value to multiply by
    MultValue(const Type& val ) : Factor(val) { }

    // The function call for the element to be multiplied
    void operator()(Type& elem) const
    {
        elem *= Factor;
    }
};

int main()
{
    vector<int> v1;
    vector<int> v2;

    v1.push_back(10);
    v1.push_back(20);

    v2.push_back(10);
    v2.push_back(20);

    // This next line will assert because v1 and v2 are
    // incompatible.
    for_each(v1.begin(), v2.end(), MultValue<int>(-2));
}
```

Debug iterator checking also implies /Zc:forScope, such that, an iterator variable declared in a for loop will not be in scope when the for loop scope ends.

```
// debug_iterator.cpp
// compile with: /EHsc /MDd
#include <vector>
#include <iostream>
int main() {
   std::vector<int> v ;

   v.push_back(10);
   v.push_back(15);
   v.push_back(20);

   for (std::vector<int>::iterator i = v.begin() ; i != v.end(); ++i)
   ;
   --i;   // C2065
}
```

Debug iterators have non-trivial destructors. If a destructor does not run, for whatever reason, it could lead to access violations and data corruption. Consider the following example:

```
/* compile with: /D_DEBUG /EHsc /MDd */
#include <vector>
struct base {
   // FIX: uncomment the next line
   // virtual ~base() {}
};

struct derived : base {
   std::vector<int>::iterator m_iter;
   derived( std::vector<int>::iterator iter ) : m_iter( iter ) {}
   ~derived() {}
};

int main() {
   std::vector<int> vect( 10 );
   base * pb = new derived( vect.begin() );
   delete pb;  // doesn't call ~derived()
   // access violation
}
```

See Also

Reference

Standard C++ Library Overview

_HAS_ITERATOR_DEBUGGING

Defines whether the iterator debugging feature is enabled in a debug build. By default, iterator debugging is enabled. For more information, see Debug Iterator Support.

Remarks

To enable iterator debugging in debug builds, set **_HAS_ITERATOR_DEBUGGING** to 1:

```
#define _HAS_ITERATOR_DEBUGGING 1
```

_HAS_ITERATOR_DEBUGGING cannot be set to 1 in retail builds.

To disable iterator debugging in debug builds, set **_HAS_ITERATOR_DEBUGGING** to 0:

```
#define _HAS_ITERATOR_DEBUGGING 0
```

See Also

Reference

Debug Iterator Support
Checked Iterators
Safe Libraries: Standard C++ Library

Standard Template Library

The Standard Template Library (STL) establishes uniform standards for the application of iterators to STL containers or other sequences that you define, by STL algorithms or other functions that you define.

In This Section

STL Containers

Iterators

Algorithms

Allocators

Function Objects

See Also

Reference

Standard C++ Library Overview

STL Containers

The Standard Template Library (STL) provides several containers for storing collections of related objects. The containers are all template classes, allowing you to specify what objects are allowed in the containers. This topic provides an overview of the STL containers to help you decide which container is best for your needs.

Containers in STL can be divided into three categories, sequence containers, associative containers, and container adapters. These categories, and the collections that belong to each category, are summarized here. More detailed information can be found in the reference documentation for the containers.

Sequence Containers

Sequence containers maintain the original ordering of inserted elements. This allows you to specify where to insert the element in the container.

The **deque** (double-ended queue) container allows for fast insertions and deletions at the beginning and end of the container. You can also randomly access any element quickly.

The **list** container allows for fast insertions and deletions anywhere in the container, but you cannot randomly access an element in the container.

The **vector** container behaves like an array, but will automatically grow as required.

For more information on the sequence containers, consult the following table:

deque Class	list Class	vector Class

Associative Containers

The defining characteristic of associative containers is that elements are inserted in a pre-defined order, such as sorted ascending.

The associative containers can be grouped into two subsets: maps and sets. A **map**, sometimes referred to as a dictionary, consists of a key/value pair. The key is used to order the sequence, and the value is somehow associated with that key. For example, a **map** might contain keys representing every unique word in a text and values representing the number of times that word appears in the text. A **set** is simply an ascending container of unique elements.

Both **map** and **set** only allow one instance of a key or element to be inserted into the container. If multiple instances of elements are required, use **multimap** or **multiset**.

Both maps and sets support bidirectional iterators. For more information on iterators, see Iterators.

While not officially part of the STL standard, **hash_map** and **hash_set** are commonly used to improve searching times. These containers store their elements as a hash table, with each table entry containing a bidirectional linked list of elements. To ensure the fastest search times, make sure that the hashing algorithm for your elements returns evenly distributed hash values.

For more information on the associative containers, consult the following table:

hash_map Class	hash_multimap Class	hash_multiset Class
hash_set Class	map Class	multimap Class
multiset Class	set Class	

Container Adapters

The container adapters are simply variations of the above containers. The container adapters do not support iterators.

The **priority_queue** container organized such that the element with the highest value is always first in the queue.

The **queue** container follows FIFO (first in, first out) semantics. The first element inserted (pushed) into the queue is the first to be removed (popped).

The **stack** container follows LIFO (last in, first out) semantics. The last element to be inserted (pushed) on the stack is the first element to be removed (popped).

Since container adapters do not support iterators, they cannot be used with the STL algorithms. For more information on algorithms, see Algorithms.

For more information on the container adapters, consult the following table:

priority_queue Class	queue Class	stack Class

Requirements for Container Elements

Elements inserted into an STL container can be of any object type that supplies a public copy constructor, a public destructor, and a public assignment operator. The destructor may not throw an exception. Furthermore, associative containers such as **set** and **map** must have a public comparison operator defined, which is operator< by default. Some operations on containers might also require a public default constructor and a public equivalence operator.

Accessing Container Elements

The elements of containers are accessed using iterators. For more information, see Iterators.

Note
You can also use for each, in to iterate over STL collections. For more information, see How to: Iterate Over STL Collection with for each.

See Also

Reference

Standard Template Library

<sample container>

<sample container>

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Shows you the structure of the container headers in the Standard C++ Library.

```
#include <sample container> // Nonfunctional header
```

See Also

Other Resources

<sample container> Members

<sample container> Operators

For more information about the operators in <sample container>, see <sample container>.

operator!=

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Overloads **operator!=** to compare two objects of template class Container.

```
template<class Ty>
   bool operator!=(
      const Container <Ty>& _Left,
      const Container <Ty>& _Right
      );
```

Return Value

Returns **!**(*_Left* **==** *_Right*).

See Also

Reference

<sample container>

operator== (<sample container>)

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Overloads **operator==** to compare two objects of template class Container.

```
template<class Ty>
   bool operator==(
      const Container <Ty>& _Left,
      const Container <Ty>& _Right
      );
```

Return Value

Returns *_Left*.size **==** *_Right*.**size && equal**(*_Left*. begin, *_Left*. end, *_Right*.**begin**).

See Also

Reference

<sample container>

operator< (<sample container>)

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Overloads **operator<** to compare two objects of template class Container.

```
template<class Ty>
   bool operator<(
      const Container <Ty>& _Left,
      const Container <Ty>& _Right
   );
```

Return Value

Returns **lexicographical_compare**(*_Left*. begin, *_Left*. end, *_Right*.**begin**, *_Right*.**end**).

See Also

Reference

<sample container>

operator<= (<sample container>)

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Overloads **operator<=** to compare two objects of template class Container.

```
template<class Ty>
   bool operator<=(
      const Container <Ty>& _Left,
      const Container <Ty>& _Right
      );
```

Return Value

Returns **!**(*_Right* < *_Left*).

See Also

Reference

<sample container>

operator> (<sample container>)

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Overloads **operator>** to compare two objects of template class Container.

```
template<class Ty>
   bool operator*gt;(
      const Container <Ty>& _Left,
      const Container <Ty>& _Right
   );
```

Return Value

Returns *_Right* < *_Left*.

See Also

Reference

<sample container>

operator>=

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Overloads **operator>=** to compare two objects of template class Container.

```
template<class Ty>
   bool operator>=(
      const Container <Ty>& _Left,
      const Container <Ty>& _Right
   );
```

Return Value

Returns **!**(*_Left* < *_Right*).

See Also

Reference

<sample container>

<sample container> Specialized Template Functions

For more information about the specialized template functions in <sample container>, see <sample container>.

swap (<sample container>)

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Executes _*Left*.swap(_*Right*).

```
template<class Ty>
   void swap(
      Container <Ty>& _Left,
      Container <Ty>& _Right
   );
```

See Also

Reference

<sample container>

<sample container> Classes

For more information about the classes in <sample container>, see <sample container>.

Sample Container Class

Note
This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Describes an object that controls a varying-length sequence of elements, typically of type **Ty**. The sequence is stored in different ways, depending on the actual container.

A container constructor or member function may find occasion to call the constructor **Ty**(**const Ty&**) or the function **Ty::operator=**(**const Ty&**). If such a call throws an exception, the container object is obliged to maintain its integrity, and to rethrow any exception it catches. You can safely swap, assign to, erase, or destroy a container object after it throws one of these exceptions. In general, however, you cannot otherwise predict the state of the sequence controlled by the container object.

A few additional caveats:

- If the expression **~Ty** throws an exception, the resulting state of the container object is undefined.
- If the container stores an allocator object *al*, and *al* throws an exception other than as a result of a call to *al*.**allocate**, the resulting state of the container object is undefined.
- If the container stores a function object *comp*, to determine how to order the controlled sequence, and *comp* throws an exception of any kind, the resulting state of the container object is undefined.

The container classes defined by STL satisfy several additional requirements, as described in the following paragraphs.

Container template class list provides deterministic, and useful, behavior even in the presence of the exceptions described above. For example, if an exception is thrown during the insertion of one or more elements, the container is left unaltered and the exception is rethrown.

For *all* the container classes defined by STL, if an exception is thrown during calls to the following member functions:

```
<A NAME="vclrfcontainerinsert"></A>insert // single element inserted
<A NAME="vclrfcontainerpushback"></A>push_back
<A NAME="vclrfcontainerpushfront"></A>push_front
```

the container is left unaltered and the exception is rethrown.

For *all* the container classes defined by STL, no exception is thrown during calls to the following member functions:

```
<A NAME="vclrfcontainerpopback"></A>pop_back
<A NAME="vclrfcontainerpopfront"></A>pop_front
```

The member function erase throws an exception only if a copy operation (assignment or copy construction) throws an exception.

Moreover, no exception is thrown while copying an iterator returned by a member function.

The member function swap makes additional promises for *all* container classes defined by STL:

- The member function throws an exception only if the container stores an allocator object al, and `al` throws an exception when copied.
- References, pointers, and iterators that designate elements of the controlled sequences being swapped remain valid.

An object of a container class defined by STL allocates and frees storage for the sequence it controls through a stored object of type **Alloc**, which is typically a template parameter. Such an allocator object must have the same external interface as an object of class **allocator<Ty>**. In particular, **Alloc** must be the same type as **Alloc::rebind<value_type>::other**

For *all* container classes defined by STL, the member function **Alloc get_allocator const;** returns a copy of the stored allocator object. Note that the stored allocator object is *not* copied when the container object is assigned. All constructors initialize the value stored in **allocator**, to **Alloc** if the constructor contains no allocator parameter.

According to the C++ Standard, a container class defined by STL can assume that:

- All objects of class **Alloc** compare equal.
- Type **Alloc::const_pointer** is the same as **const Ty ***.
- Type **Alloc::const_reference** is the same as **const Ty&**.
- Type **Alloc::pointer** is the same as **Ty ***.
- Type **Alloc::reference** is the same as **Ty&**.

In this implementation, however, containers do not make such simplifying assumptions. Thus, they work properly with allocator objects that are more ambitious:

- All objects of class **Alloc** does not need to compare equal. (You can maintain multiple pools of storage.)
- Type **Alloc::const_pointer** does not need to be the same as **const Ty ***. (A const pointer can be a class.)
- Type **Alloc::pointer** does not need to be the same as **Ty ***. (A pointer can be a class.)

Requirements

Header: <sample container>

See Also

Reference

<sample container>

Sample Container Members

Note
This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Reference

Typedefs

const_iterator	Describes an object that can serve as a constant iterator for the controlled sequence.
const_reference	Describes an object that can serve as a constant reference to an element of the controlled sequence.
const_reverse_iterator	Describes an object that can serve as a constant reverse iterator for the controlled sequence.
difference_type	Describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.
iterator	Describes an object that can serve as an iterator for the controlled sequence.
reference	Describes an object that can serve as a reference to an element of the controlled sequence.
reverse_iterator	Describes an object that can serve as a reverse iterator for the controlled sequence.
size_type	Describes an object that can represent the length of any controlled sequence.
value_type	Acts a synonym for the template parameter **Ty**.

Member Functions

begin	Returns an iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).
clear	Calls erase(begin, end).
empty	Returns **true** for an empty controlled sequence.
end	Returns an iterator that points just beyond the end of the sequence.
erase	Erases an element.
max_size	Returns the length of the longest sequence that the object can control, in constant time regardless of the length of the controlled sequence.
rbegin	Returns a reverse iterator that points just beyond the end of the controlled sequence, designating the beginning of the reverse sequence.
rend	The member function returns a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence), designating the end of the reverse sequence.
size	Returns the length of the controlled sequence, in constant time regardless of the length of the controlled sequence.
swap	Swaps the controlled sequences between ***this** and *_Right*.

Sample Container Typedefs

For more information about the typedefs in the sample container class, see Sample Container Class.

Container Class::const_iterator

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Describes an object that can serve as a constant iterator for the controlled sequence.

```
typedef T6 const_iterator;
```

Remarks

It is described here as a synonym for the unspecified type **T6**.

See Also

Reference

Sample Container Class

Container Class::const_reference

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Describes an object that can serve as a constant reference to an element of the controlled sequence.

```
typedef T3 const_reference;
```

Remarks

It is described here as a synonym for the unspecified type **T3** (typically **Alloc::const_reference**).

See Also

Reference

Sample Container Class

Container Class::const_reverse_iterator

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Describes an object that can serve as a constant reverse iterator for the controlled sequence.

```
typedef T8 const_reverse_iterator;
```

Remarks

It is described here as a synonym for the unspecified type **T8** (typically reverse_iterator <const_iterator>).

See Also

Reference

Sample Container Class

Container Class::difference_type

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.

```
typedef T1 difference_type;
```

Remarks

It is described here as a synonym for the unspecified type **T1** (typically **Alloc::difference_type**).

See Also

Reference

Sample Container Class

Container Class::iterator

Note
This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Describes an object that can serve as an iterator for the controlled sequence.

```
typedef T5 iterator;
```

Remarks

It is described here as a synonym for the unspecified type **T5**. An object of type **iterator** can be cast to an object of type const_iterator.

See Also

Reference

Sample Container Class

Container Class::reference

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Describes an object that can serve as a reference to an element of the controlled sequence.

```
typedef T2 reference;
```

Remarks

It is described here as a synonym for the unspecified type **T2** (typically **Alloc::reference**). An object of type **reference** can be cast to an object of type const_reference.

See Also

Reference

Sample Container Class

Container Class::reverse_iterator

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Describes an object that can serve as a reverse iterator for the controlled sequence.

```
typedef T7 reverse_iterator;
```

Remarks

It is described here as a synonym for the unspecified type **T7** (typically **reverse_iterator** <iterator>).

See Also

Reference

Sample Container Class

Container Class::size_type

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Describes an object that can represent the length of any controlled sequence.

```
typedef T0 size_type;
```

Remarks

It is described here as a synonym for the unspecified type **T0** (typically **Alloc::size_type**).

See Also

Reference

Sample Container Class

Container Class::value_type

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Acts a synonym for the template parameter **Ty**.

```
typedef T4 value_type;
```

Remarks

It is described here as a synonym for the unspecified type **T4** (typically **Alloc::value_type**).

See Also

Reference

Sample Container Class

Sample Container Member Functions

For more information about the member functions in the sample container class, see Sample Container Class.

Container Class::begin

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Returns an iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

```
const_iterator begin( ) const;
iterator begin( );
```

See Also

Reference

Sample Container Class

Container Class::clear

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Calls erase(begin, end).

```
void clear( );
```

See Also

Reference

Sample Container Class

Container Class::empty

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Returns **true** for an empty controlled sequence.

```
bool empty( ) const;
```

See Also

Reference

Sample Container Class

Container Class::end

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Returns an iterator that points just beyond the end of the sequence.

```
const_iterator end( ) const;
iterator end( );
```

See Also

Reference

Sample Container Class

Container Class::erase

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Erases an element.

```
iterator erase(
   iterator _Where
);
iterator erase(
   iterator _First,
   iterator _Last
);
```

Remarks

The first member function removes the element of the controlled sequence pointed to by *_Where*. The second member function removes the elements of the controlled sequence in the range [*_First*, *_Last*). Both return an iterator that designates the first element remaining beyond any elements removed, or end if no such element exists.

The member functions throw an exception only if a copy operation throws an exception.

See Also

Reference

Sample Container Class

Container Class::max_size

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Returns the length of the longest sequence that the object can control, in constant time regardless of the length of the controlled sequence.

```
size_type max_size( ) const;
```

See Also

Reference

Sample Container Class

Container Class::rbegin

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Returns a reverse iterator that points just beyond the end of the controlled sequence, designating the beginning of the reverse sequence.

```
const_reverse_iterator rbegin( ) const;
reverse_iterator rbegin( );
```

See Also

Reference

Sample Container Class

Container Class::rend

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

The member function returns a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence), designating the end of the reverse sequence.

```
const_reverse_iterator rend( ) const;
reverse_iterator rend( );
```

See Also

Reference

Sample Container Class

Container Class::size

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Returns the length of the controlled sequence, in constant time regardless of the length of the controlled sequence.

```
size_type size( ) const;
```

See Also

Reference

Sample Container Class

Container Class::swap

Note

This topic is in the Visual C++ documentation as a nonfunctional example of containers used in the Standard C++ Library. For more information, see STL Containers.

Swaps the controlled sequences between ***this** and *_Right*.

```
void swap(
   Container& _Right
);
```

Remarks

If **get_allocator ==** *_Right*.**get_allocator**, it does so in constant time. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

See Also

Reference

Sample Container Class

Iterators

The STL facilities make widespread use of iterators to mediate among the various algorithms and the sequences upon which they act. The name of an iterator type (or its prefix) indicates the category of iterators required for that type. In order of increasing power, the categories are summarized here as:

- Output. An output iterator *X* can only have a value *V* stored indirect on it, after which it *must* be incremented before the next store, as in (**X*++ = *V*), (**X* = *V*, ++*X*), or (**X* = *V*, *X*++).
- Input. An input iterator *X* can represent a singular value that indicates end of sequence. If an input iterator does not compare equal to its end-of-sequence value, it can have a value *V* accessed indirect on it any number of times, as in (*V* = **X*). To progress to the next value or end of sequence, you increment it, as in ++*X*, *X*++, or (*V* = **X*++). Once you increment any copy of an input iterator, none of the other copies can safely be compared, dereferenced, or incremented thereafter.
- Forward. A forward iterator *X* can take the place of an output iterator for writing or an input iterator for reading. You can, however, read (through *V* = **X*) what you just wrote (through **X* = *V*) through a forward iterator. You can also make multiple copies of a forward iterator, each of which can be dereferenced and incremented independently.
- Bidirectional. A bidirectional iterator *X* can take the place of a forward iterator. You can, however, also decrement a bidirectional iterator, as in --*X*, *X*--, or (*V* = **X*--).
- Random access. A random-access iterator *X* can take the place of a bidirectional iterator. You can also perform much the same integer arithmetic on a random-access iterator that you can on an object pointer. For *N*, an integer object, you can write *x*[*N*], *x* + *N*, *x* - *N*, and *N* + *X*.

Note that an object pointer can take the place of a random-access iterator or any other iterator. All iterators can be assigned or copied. They are assumed to be lightweight objects and are often passed and returned by value, not by reference. Note also that none of the operations previously described can throw an exception when performed on a valid iterator.

The hierarchy of iterator categories can be summarized by showing three sequences. For write-only access to a sequence, you can use any of:

```
output iterator
   -> forward iterator
   -> bidirectional iterator
   -> random-access iterator
```

The right arrow means "can be replaced by." Any algorithm that calls for an output iterator should work nicely with a forward iterator, for example, but *not* the other way around.

For read-only access to a sequence, you can use any of:

```
input iterator
   -> forward iterator
   -> bidirectional iterator
   -> random-access iterator
```

An input iterator is the weakest of all categories, in this case.

Finally, for read/write access to a sequence, you can use any of:

```
forward iterator
   -> bidirectional iterator
   -> random-access iterator
```

An object pointer can always serve as a random-access iterator, so it can serve as any category of iterator if it supports the proper read/write access to the sequence it designates.

An iterator **Iterator** other than an object pointer must also define the member types required by the specialization **iterator_traits<Iterator>**. Note that these requirements can be met by deriving **Iterator** from the public base class iterator.

This "algebra" of iterators is fundamental to practically everything else in the Standard Template Library. It is important to understand the promises and limitations of each iterator category to see how iterators are used by containers and algorithms in the STL.

Note
You can also use for each, in to iterate over STL collections. For more information, see How to: Iterate Over STL Collection with for each.

Visual C++ now offers checked iterators and debug iterators to ensure that you do not overwrite the bounds of your container. For more information, see Checked Iterators and Debug Iterator Support.

See Also

Reference

Standard Template Library

Thread Safety in the Standard C++ Library

Algorithms

Algorithms are a fundamental part of the Standard Template Library. Algorithms do not work with containers themselves but rather with iterators. Therefore, the same algorithm can be used by most if not all of the STL containers. This section discusses the conventions and terminology of the STL algorithms.

Remarks

The descriptions of the algorithm template functions employ several shorthand phrases:

- The phrase "in the range [*A*, *B*)" means the sequence of zero or more discrete values beginning with *A* up to but not including *B*. A range is valid only if *B* is reachable from *A;* you can store *A* in an object *N* (*N* = *A*), increment the object zero or more times (++*N*), and have the object compare equal to *B* after a finite number of increments (N == B*)*.
- The phrase "each *N* in the range [*A*, *B*)" means that *N* begins with the value *A* and is incremented zero or more times until it equals the value *B*. The case *N* == *B* is not in the range.
- The phrase "the lowest value of *N* in the range [*A*, *B*) such that *X*" means that the condition *X* is determined for each *N* in the range [*A*, *B*) until the condition *X* is met.
- The phrase "the highest value of *N* in the range [*A*, *B*) such that *X* means that *X* is determined for each *N* in the range [*A*, *B*). The function stores in *K* a copy of *N* each time the condition *X* is met. If any such store occurs, the function replaces the final value of *N*, which equals *B*, with the value of *K*. For a bidirectional or random-access iterator, however, it can also mean that *N* begins with the highest value in the range and is decremented over the range until the condition *X* is met.
- Expressions such as *X* - *Y*, where *X* and *Y* can be iterators other than random-access iterators, are intended in the mathematical sense. The function does not necessarily evaluate operator**-** if it must determine such a value. The same is also true for expressions such as *X* + *N* and *X* - *N*, where *N* is an integer type.

Several algorithms make use of a predicate that performs a pairwise comparison, such as with **operator==**, to yield a **bool** result. The predicate function **operator==**, or any replacement for it, must not alter either of its operands. It must yield the same **bool** result every time it is evaluated, and it must yield the same result if a copy of either operand is substituted for the operand.

Several algorithms make use of a predicate that must impose a strict weak ordering on pairs of elements from a sequence. For the predicate *pr*(*X*, *Y*):

- Strict means that *pr*(*X*, *X*) is false.
- Weak means that *X* and *Y* have an equivalent ordering if !*pr*(*X*, *Y*) && !*pr*(*Y*, *X*) (*X* == *Y* does not need to be defined).
- Ordering means that *pr*(*X*, *Y*) && *pr*(*Y*, Z) implies *pr*(*X*, Z).

Some of these algorithms implicitly use the predicate *X* < *Y*. Other predicates that typically satisfy the strict weak ordering requirement are *X* > *Y*, **less**(*X*, *Y*), and **greater**(*X*, *Y*). Note, however, that predicates such as *X* <= *Y* and *X* >= *Y* do not satisfy this requirement.

A sequence of elements designated by iterators in the range [*First*, *Last*) is a sequence ordered by operator**<** if, for each *N* in the range [0, *Last* - *First*) and for each *M* in the range (N, *Last* - *First*) the predicate !(*(*First* + *M*) < *(*First* + *N*)) is true. (Note that the elements are sorted in ascending order.) The predicate function **operator<**, or any replacement for it, must not alter either of its operands. It must yield the same **bool** result every time it is evaluated, and it must yield the same result if a copy of either operand is substituted for the operand. Moreover, it must impose a strict weak ordering on the operands it compares.

A sequence of elements designated by iterators in the range [*First*, *Last*) is a heap ordered by **operator<** if, for each *N* in the range [1, *Last* - *First*) the predicate !(**First* < *(*First* + *N*)) is true. (The first element is the largest.) Its internal structure is otherwise known only to the template functions make_heap, pop_heap, and push_heap. As with an ordered sequence, the predicate function **operator<**, or any replacement for it, must not alter either of its operands, and it must impose a strict weak ordering on the operands it compares. It must yield the same **bool** result every time it is evaluated, and it must yield the same result if a copy of either operand is substituted for the operand.

The STL algorithms are located in the <algorithm> and <numeric> header files.

See Also

Reference

Standard Template Library

Thread Safety in the Standard C++ Library

Allocators

Allocators are used by the Standard Template Library to handle the allocation and deallocation of elements stores in containers. All STL containers have a template argument of type **allocator<Type>**, where *Type* represents the type of the container element. For example, the vector class is declared as follows:

```
template <
    class Type,
    class Allocator = allocator<Type>
>
class vector
```

The Standard Template Library provides a default implementation for an allocator. In most cases, this default allocator should be sufficient. For more information on the default allocator, see allocator Class.

Writing Your Own Allocator

The default allocator uses **new** and **delete** to allocate and deallocate memory. If you want to use a different method of memory allocation, such as using shared memory, then you must create your own allocator.

Any allocator used with STL containers must implement the following type definitions:

const_pointer	**rebind**
const_reference	**reference**
difference_type	**size_type**
pointer	**value_type**

In addition, any allocator used with STL containers must implement the following methods:

Constructor	**deallocate**
Copy constructor	**destroy**
Destructor	**max_size**
address	**operator==**
allocate	**operator!=**
construct	

For more information on these type definitions and methods, see allocator Class.

See Also

Reference

Standard Template Library

Function Objects

A *function object*, or *functor*, is any type that implements operator(). This operator is referred to as the *call operator* or sometimes the *application operator*. The Standard Template Library uses function objects primarily as sorting criteria for containers and in algorithms.

Function objects provide two main advantages over a straight function call. The first is that a function object can contain state. The second is that a function object is a type and therefore can be used as a template parameter.

Creating a Function Object

To create a function object, create a type and implement operator(), such as:

```
class Functor
{
public:
    int operator()(int a, int b)
    {
        return a < b;
    }
};

int main()
{
    Functor f;
    int a = 5;
    int b = 7;
    int ans = f(a, b);
}
```

The last line of the **main** function shows how you call the function object. This call looks like a call to a function, but it is actually calling operator() of the Functor type. This similarity between calling a function object and a function is how the term function object came about.

Function Objects and Containers

The Standard Template Library contains several function objects in the <functional> header file. One use of these function objects is as a sorting criterion for containers. For example, the **set** container is declared as follows:

```
template <
    class Key,
    class Traits=less<Key>,
    class Allocator=allocator<Key> >
class set
```

The second template argument is the function object **less**. This function object returns **true** if the first parameter passed to it is less than the second parameter passed. Since some containers sort their elements, the container needs a way of comparing two elements, and this is accomplished using the function object. You can define your own sorting criteria for containers by creating a function object and specifying it in the template list for the container.

Function Objects and Algorithms

Another use of functional objects is in algorithms. For example, the **remove_if** algorithm is declared as follows:

```
template<class ForwardIterator, class Predicate>
    ForwardIterator remove_if(
        ForwardIterator _First,
        ForwardIterator _Last,
        Predicate _Pred
    );
```

The last argument to **remove_if** is a function object that returns a boolean value (a *predicate*). If the result of the function

object is **true**, then the element is removed from the container being accessed by the iterators *_First* and *_Last*. You can use any of the function objects declared in the <functional> header for the argument *_Pred* or you can create your own.

See Also

Reference

Standard Template Library

Thread Safety in the Standard C++ Library

The following thread safety rules apply to the Standard C++ Library:

Container Classes and complex

The container classes are vector, deque, list, queue, stack, priority_queue, valarray, map, hash_map, multimap, hash_multimap, set, hash_set, multiset, hash_multiset, basic_string, and bitset.

A single object is thread safe for reading from multiple threads. For example, given an object A, it is safe to read A from thread 1 and from thread 2 simultaneously.

If a single object is being written to by one thread, then all reads and writes to that object on the same or other threads must be protected. For example, given an object A, if thread 1 is writing to A, then thread 2 must be prevented from reading from or writing to A.

It is safe to read and write to one instance of a type even if another thread is reading or writing to a different instance of the same type. For example, given objects A and B of the same type, it is safe if A is being written in thread 1 and B is being read in thread 2.

iostream Classes

The iostream classes follow the same rules as the container classes, with one exception. It is safe to write to an object from multiple threads. For example, thread 1 can write to cout at the same time as thread 2. However, this can result in the output from the two threads being intermixed.

Note
Reading from a stream buffer is not considered to be a read operation. It should be considered as a write operation, because this changes the state of the class.

See Also

Reference

Standard C++ Library Overview

The stdext Namespace

Members of the <hash_map> and <hash_set> header files are not currently part of the ISO C++ standard. Therefore, these types and members have been moved from the **std** namespace to namespace **stdext**, to remain conformant with the C++ standard.

When compiling with /Ze, which is the default, the compiler will warn on the use of **std** for members of the <hash_map> and <hash_set> header files. To disable the warning, use the warning pragma.

To have the compiler generate an error for the use of **std** for members of the <hash_map> and <hash_set> header files with **/Ze**, add the following directive before #include'ing any Standard C++ Library header files.

```
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
```

When compiling with **/Za**, the compiler will generate an error.

See Also

Reference

Standard C++ Library Overview

iostream Programming

This section provides a general description of the iostream classes and then describes output streams, input streams, and input/output streams. The end of the section provides information about advanced iostream programming.

There is also a discussion on Thread Safety in the Standard C++ Library and the stdext namespace.

In This Section

What a Stream Is

Output Streams

Input Streams

Input/Output Streams

Custom Manipulators with Arguments

Deriving Your Own Stream Classes

See Also

Reference

iostreams Conventions

Other Resources

Standard C++ Library Reference

What a Stream Is

Like C, C++ does not have built-in input/output capability. All C++ compilers, however, come bundled with a systematic, object-oriented I/O package, known as the iostream classes. The stream is the central concept of the iostream classes. You can think of a stream object as a smart file that acts as a source and destination for bytes. A stream's characteristics are determined by its class and by customized insertion and extraction operators.

Through device drivers, the disk operating system deals with the keyboard, screen, printer, and communication ports as extended files. The iostream classes interact with these extended files. Built-in classes support reading from and writing to memory with syntax identical to that for disk I/O, which makes it easy to derive stream classes.

In This Section

Input/Output Alternatives

See Also

Reference

iostream Programming

Input/Output Alternatives

Visual C++ provides several alternatives for I/O programming:

- C run-time library direct, unbuffered I/O.
- ANSI C run-time library stream I/O.
- Console and port direct I/O.
- Microsoft Foundation Class Library.
- Microsoft Standard C++ Library.

The iostream classes are useful for buffered, formatted text I/O. They are also useful for unbuffered or binary I/O if you need a C++ programming interface and decide not to use the Microsoft Foundation Class (MFC) library. The iostream classes are an object-oriented I/O alternative to the C run-time functions.

You can use iostream classes with the Microsoft Windows operating system. String and file streams work without restrictions, but the character-mode stream objects **cin**, **cout**, **cerr**, and **clog** are inconsistent with the Windows graphical user interface. You can also derive custom stream classes that interact directly with the Windows environment.

See Also
Reference
What a Stream Is

Output Streams

An output stream object is a destination for bytes. The three most important output stream classes are ostream, ofstream, and ostrstream.

The **ostream** class, through the derived class basic_ostream, supports the predefined stream objects:

- **cout** standard output
- **cerr** standard error with limited buffering
- **clog** similar to **cerr** but with full buffering

Objects are rarely constructed from **ostream**; predefined objects are generally used. In some cases, you can reassign predefined objects after program startup. The **ostream** class, which can be configured for buffered or unbuffered operation, is best suited to sequential text-mode output. All functionality of the base class, **ios**, is included in **ostream**. If you construct an object of class **ostream**, you must specify a **streambuf** object to the constructor.

The **ofstream** class supports disk file output. If you need an output-only disk, construct an object of class **ofstream**. You can specify whether **ofstream** objects accept binary or text-mode data when constructing the **ofstream** object or when calling the **open** member function of the object. Many formatting options and member functions apply to **ofstream** objects, and all functionality of the base classes **ios** and **ostream** is included.

If you specify a filename in the constructor, that file is automatically opened when the object is constructed. Otherwise, you can use the **open** member function after invoking the default constructor.

Like the run-time function **sprintf**, the **ostrstream** class supports output to in-memory strings. To create a string in memory using I/O stream formatting, construct an object of class **ostrstream**. Because **ostrstream** objects are write-only, your program must access the resulting string through a pointer to **char**.

In This Section

Constructing Output Stream Objects

Using Insertion Operators and Controlling Format

Output File Stream Member Functions

The Effects of Buffering

Binary Output Files

Overloading the << Operator for Your Own Classes

Writing Your Own Manipulators Without Arguments

See Also

Reference

iostream Programming

Constructing Output Stream Objects

If you use only the predefined **cout**, **cerr**, or **clog** objects, you do not need to construct an output stream. You must use constructors for:

- Output File Stream Constructors
- Output String Stream Constructors

Output File Stream Constructors

You can construct an output file stream in one of two ways:

- Use the default constructor, and then call the **open** member function.

```
ofstream myFile; // Static or on the stack
myFile.open( "filename" );

ofstream* pmyFile = new ofstream; // On the heap
pmyFile->open( "filename" );
```

- Specify a filename and mode flags in the constructor call.

```
ofstream myFile( "filename", ios_base::out);
```

Output String Stream Constructors

To construct an output string stream, you can use one of two ostrstream constructors. One dynamically allocates its own storage, and the other requires the address and size of a preallocated buffer.

- The dynamic constructor is used in the following way:

```
char* sp;
ostrstream myString;
mystring << "this is a test" << ends;
sp = myString.str();  // Get a pointer to the string
```

  The **ends** "manipulator" adds the necessary terminating null character to the string.

- The constructor that requires the preallocated buffer is used in the following way:

```
char s[32];
ostrstream myString( s, sizeof( s ) );
myString << "this is a test" << ends; // Text stored in s
```

See Also
Reference
Output Streams

Using Insertion Operators and Controlling Format

This topic shows how to control format and how to create insertion operators for your own classes. The insertion (**<<**) operator, which is preprogrammed for all standard C++ data types, sends bytes to an output stream object. Insertion operators work with predefined "manipulators," which are elements that change the default format of integer arguments.

You can control the format with the following options:

- Output Width
- Alignment
- Precision
- Radix

Output Width

To align output, you specify the output width for each item by placing the **setw** manipulator in the stream or by calling the **width** member function. This example right-aligns the values in a column at least 10 characters wide:

```
// output_width.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
   double values[] = { 1.23, 35.36, 653.7, 4358.24 };
   for( int i = 0; i < 4; i++ )
   {
      cout.width(10);
      cout << values[i] << '\n';
   }
}
```

Output

```
      1.23
     35.36
     653.7
   4358.24
```

Leading blanks are added to any value fewer than 10 characters wide.

To pad a field, use the **fill** member function, which sets the value of the padding character for fields that have a specified width. The default is a blank. To pad the column of numbers with asterisks, modify the previous **for** loop as follows:

```
for( int i = 0; i < 4; i++ )
{
   cout.width( 10 );
   cout.fill( '*' );
   cout << values[i] << endl;
}
```

The **endl** manipulator replaces the newline character (`'\n'`). The output looks like this:

```
******1.23
*****35.36
*****653.7
***4358.24
```

To specify widths for data elements in the same line, use the **setw** manipulator:

```
// setw.cpp
// compile with: /EHsc
#include <iostream>
#include <iomanip>
```

```
using namespace std;

int main( )
{
   double values[] = { 1.23, 35.36, 653.7, 4358.24 };
   char *names[] = { "Zoot", "Jimmy", "Al", "Stan" };
   for( int i = 0; i < 4; i++ )
      cout << setw( 6 )  << names[i]
           << setw( 10 ) << values[i] << endl;
}
```

Output

The **width** member function is declared in <iostream>. If you use **setw** or any other manipulator with arguments, you must include <iomanip>. In the output, strings are printed in a field of width 6 and integers in a field of width 10:

```
  Zoot      1.23
 Jimmy     35.36
    Al     653.7
  Stan   4358.24
```

Neither **setw** nor **width** truncates values. If formatted output exceeds the width, the entire value prints, subject to the stream's precision setting. Both **setw** and **width** affect the following field only. Field width reverts to its default behavior (the necessary width) after one field has been printed. However, the other stream format options remain in effect until changed.

Alignment

Output streams default to right-aligned text. To left-align the names in the previous example and right-align the numbers, replace the **for** loop as follows:

```
for ( int i = 0; i < 4; i++ )
   cout << setiosflags( ios::left )
        << setw( 6 )  << names[i]
        << resetiosflags( ios::left )
        << setw( 10 ) << values[i] << endl;
```

The output looks like this:

```
Zoot        1.23
Jimmy      35.36
Al         653.7
Stan     4358.24
```

The left-align flag is set by using the setiosflags manipulator with the left enumerator. This enumerator is defined in the ios class, so its reference must include the **ios::** prefix. The resetiosflags manipulator turns off the left-align flag. Unlike **width** and **setw**, the effect of **setiosflags** and **resetiosflags** is permanent.

Precision

The default value for floating-point precision is six. For example, the number 3466.9768 prints as 3466.98. To change the way this value prints, use the setprecision manipulator. The manipulator has two flags: fixed and scientific. If fixed is set, the number prints as 3466.976800. If **scientific** is set, it prints as 3.4669773+003.

To display the floating-point numbers shown in Alignment with one significant digit, replace the **for** loop as follows:

```
for ( int i = 0; i < 4; i++ )
   cout << setiosflags( ios::left )
        << setw( 6 )
        << names[i]
        << resetiosflags( ios::left )
        << setw( 10 )
        << setprecision( 1 )
        << values[i]
        << endl;
```

The program prints this list:

```
Zoot          1
Jimmy     4e+001
Al        7e+002
Stan      4e+003
```

To eliminate scientific notation, insert this statement before the **for** loop:

```
cout << setiosflags( ios::fixed );
```

With fixed notation, the program prints with one digit after the decimal point.

```
Zoot         1.2
Jimmy       35.4
Al         653.7
Stan      4358.2
```

If you change the **ios::fixed** flag to **ios::scientific**, the program prints this:

```
Zoot    1.2e+000
Jimmy   3.5e+001
Al      6.5e+002
Stan    4.4e+003
```

Again, the program prints one digit after the decimal point. If either **ios::fixed** or **ios::scientific** is set, the precision value determines the number of digits after the decimal point. If neither flag is set, the precision value determines the total number of significant digits. The **resetiosflags** manipulator clears these flags.

Radix

The **dec**, **oct**, and **hex** manipulators set the default radix for input and output. For example, if you insert the **hex** manipulator into the output stream, the object correctly translates the internal data representation of integers into a hexadecimal output format. The numbers are displayed with digits a through f in lower case if the uppercase flag is clear (the default); otherwise, they are displayed in upper case. The default radix is **dec** (decimal).

See Also

Reference

Output Streams

Output File Stream Member Functions

Output stream member functions have three types: those that are equivalent to manipulators, those that perform unformatted write operations, and those that otherwise modify the stream state and have no equivalent manipulator or insertion operator. For sequential, formatted output, you might use only insertion operators and manipulators. For random-access binary disk output, you use other member functions, with or without insertion operators.

The open Function for Output Streams

To use an output file stream (ofstream), you must associate that stream with a specific disk file in the constructor or the **open** function. If you use the **open** function, you can reuse the same stream object with a series of files. In either case, the arguments describing the file are the same.

When you open the file associated with an output stream, you generally specify an **open_mode** flag. You can combine these flags, which are defined as enumerators in the **ios** class, with the bitwise OR (|) operator. See ios_base::openmode for a list of the enumerators.

Three common output stream situations involve mode options:

- Creating a file. If the file already exists, the old version is deleted.

```
ostream ofile( "FILENAME" );  // Default is ios::out
ofstream ofile( "FILENAME", ios::out ); // Equivalent to above
```

- Appending records to an existing file or creating one if it does not exist.

```
ofstream ofile( "FILENAME", ios::app );
```

- Opening two files, one at a time, on the same stream.

```
ofstream ofile();
ofile.open( "FILE1", ios::in );
// Do some output
ofile.close(); // FILE1 closed
ofile.open( "FILE2", ios::in );
// Do some more output
ofile.close(); // FILE2 closed
// When ofile goes out of scope it is destroyed.
```

The put Function

The **put** function writes one character to the output stream. The following two statements are the same by default, but the second is affected by the stream's format arguments:

```
cout.put( 'A' ); // Exactly one character written
cout << 'A'; // Format arguments 'width' and 'fill' apply
```

The write Function

The **write** function writes a block of memory to an output file stream. The length argument specifies the number of bytes written. This example creates an output file stream and writes the binary value of the `Date` structure to it:

```
// write_function.cpp
// compile with: /EHsc
#include <fstream>
using namespace std;

struct Date
{
   int mo, da, yr;
```

```
};

int main( )
{
   Date dt = { 6, 10, 92 };
   ofstream tfile( "date.dat" , ios::binary );
   tfile.write( (char *) &dt, sizeof dt );
}
```

The **write** function does not stop when it reaches a null character, so the complete class structure is written. The function takes two arguments: a **char** pointer and a count of characters to write. Note the required cast to **char*** before the address of the structure object.

The seekp and tellp Functions

An output file stream keeps an internal pointer that points to the position where data is to be written next. The **seekp** member function sets this pointer and thus provides random-access disk file output. The **tellp** member function returns the file position. For examples that use the input stream equivalents to **seekp** and **tellp**, see The seekg and tellg Functions.

The close Function for Output Streams

The **close** member function closes the disk file associated with an output file stream. The file must be closed to complete all disk output. If necessary, the **ofstream** destructor closes the file for you, but you can use the **close** function if you need to open another file for the same stream object.

The output stream destructor automatically closes a stream's file only if the constructor or the **open** member function opened the file. If you pass the constructor a file descriptor for an already-open file or use the **attach** member function, you must close the file explicitly.

Error Processing Functions

Use these member functions to test for errors while writing to a stream:

Function	**Return value**
bad	Returns **true** if there is an unrecoverable error.
fail	Returns **true** if there is an unrecoverable error or an "expected" condition, such as a conversion error, or if the file is not found. Processing can often resume after a call to **clear** with a zero argument.
good	Returns **true** if there is no error condition (unrecoverable or otherwise) and the end-of-file flag is not set.
eof	Returns **true** on the end-of-file condition.
clear	Sets the internal error state. If called with the default arguments, it clears all error bits.
rdstate	Returns the current error state.

The **!** operator is overloaded to perform the same function as the **fail** function. Thus the expression:

```
if( !cout)...
```

is equivalent to:

```
if( cout.fail() )...
```

The **void*()** operator is overloaded to be the opposite of the **!** operator; thus the expression:

```
if( cout)...
```

is equal to:

```
if( !cout.fail() )...
```

The **void*()** operator is not equivalent to **good** because it does not test for the end of file.

See Also

Reference

Output Streams

The Effects of Buffering

The following example shows the effects of buffering. You might expect the program to print `please wait`, wait 5 seconds, and then proceed. It will not necessarily work this way, however, because the output is buffered.

```
// effects_buffering.cpp
// compile with: /EHsc
#include <iostream>
#include <time.h>
using namespace std;

int main( )
{
   time_t tm = time( NULL ) + 5;
   cout << "Please wait...";
   while ( time( NULL ) < tm )
      ;
   cout << "\nAll done" << endl;
}
```

To make the program work logically, the **cout** object must empty itself when the message is to appear. To flush an **ostream** object, send it the **flush** manipulator:

```
cout << "Please wait..." << flush;
```

This step flushes the buffer, ensuring the message prints before the wait. You can also use the **endl** manipulator, which flushes the buffer and outputs a carriage return–linefeed, or you can use the **cin** object. This object (with the **cerr** or **clog** objects) is usually tied to the **cout** object. Thus, any use of **cin** (or of the **cerr** or **clog** objects) flushes the **cout** object.

See Also

Reference

Output Streams

Binary Output Files

Streams were originally designed for text, so the default output mode is text. In text mode, the newline character (hexadecimal 10) expands to a carriage return–linefeed (16-bit only). The expansion can cause problems, as shown here:

```
// binary_output_files.cpp
// compile with: /EHsc
#include <fstream>
using namespace std;
int iarray[2] = { 99, 10 };
int main( )
{
    ofstream os( "test.dat" );
    os.write( (char *) iarray, sizeof( iarray ) );
}
```

You might expect this program to output the byte sequence { 99, 0, 10, 0 }; instead, it outputs { 99, 0, 13, 10, 0 }, which causes problems for a program expecting binary input. If you need true binary output, in which characters are written untranslated, you could specify binary output by using the ofstream constructor mode argument:

```
// binary_output_files2.cpp
// compile with: /EHsc
#include <fstream>
using namespace std;
int iarray[2] = { 99, 10 };

int main()
{
   ofstream ofs ( "test.dat", ios_base::binary );

   // Exactly 8 bytes written
   ofs.write( (char*)&iarray[0], sizeof(int)*2 );
}
```

See Also

Reference

Output Streams

Overloading the << Operator for Your Own Classes

Output streams use the insertion (**<<**) operator for standard types. You can also overload the **<<** operator for your own classes.

Example

The **write** function example showed the use of a `Date` structure. A date is an ideal candidate for a C++ class in which the data members (month, day, and year) are hidden from view. An output stream is the logical destination for displaying such a structure. This code displays a date using the **cout** object:

```
Date dt( 1, 2, 92 );
cout << dt;
```

To get **cout** to accept a `Date` object after the insertion operator, overload the insertion operator to recognize an **ostream** object on the left and a `Date` on the right. The overloaded **<<** operator function must then be declared as a friend of class `Date` so it can access the private data within a `Date` object.

```
// overload_date.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Date
{
    int mo, da, yr;
public:
    Date(int m, int d, int y)
    {
        mo = m; da = d; yr = y;
    }
    friend ostream& operator<<(ostream& os, const Date& dt);
};

ostream& operator<<(ostream& os, const Date& dt)
{
    os << dt.mo << '/' << dt.da << '/' << dt.yr;
    return os;
}

int main()
{
    Date dt(5, 6, 92);
    cout << dt;
}
```

Output

```
5/6/92
```

Remarks

The overloaded operator returns a reference to the original **ostream** object, which means you can combine insertions:

```
cout << "The date is" << dt << flush;
```

See Also

Reference

Output Streams

Writing Your Own Manipulators Without Arguments

Writing manipulators that do not use arguments requires neither class derivation nor use of complex macros. Suppose your printer requires the pair <ESC>[to enter bold mode. You can insert this pair directly into the stream:

```
cout << "regular " << '\033' << '[' << "boldface" << endl;
```

Or you can define the **bold** manipulator, which inserts the characters:

```
ostream& bold( ostream& os ) {
    return os << '\033' << '[';
}
cout << "regular " << bold << "boldface" << endl;
```

The globally defined **bold** function takes an **ostream** reference argument and returns the **ostream** reference. It is not a member function or a friend because it does not need access to any private class elements. The **bold** function connects to the stream because the stream's **<<** operator is overloaded to accept that type of function, using a declaration that looks something like this:

```
_Myt& operator<<(ios_base& (__cdecl *_Pfn)(ios_base&))
{
   // call ios_base manipulator
   (*_Pfn)(*(ios_base *)this);
   return (*this);
}
```

You can use this feature to extend other overloaded operators. In this case, it is incidental that **bold** inserts characters into the stream. The function is called when it is inserted into the stream, not necessarily when the adjacent characters are printed. Thus, printing could be delayed because of the stream's buffering.

See Also

Reference

Output Streams

Input Streams

An input stream object is a source of bytes. The three most important input stream classes are istream, ifstream, and istrstream.

The istream class is best used for sequential text-mode input. You can configure objects of class istream for buffered or unbuffered operation. All functionality of the base class, ios, is included in istream. You will rarely construct objects from class istream. Instead, you will generally use the predefined **cin** object, which is actually an object of class ostream. In some cases, you can assign **cin** to other stream objects after program startup.

The ifstream class supports disk file input. If you need an input-only disk file, construct an object of class ifstream. You can specify binary or text-mode data. If you specify a filename in the constructor, the file is automatically opened when the object is constructed. Otherwise, you can use the **open** function after invoking the default constructor. Many formatting options and member functions apply to ifstream objects. All functionality of the base classes ios and istream is included in ifstream.

Like the library function **sscanf**, the istrstream class supports input from in-memory strings. To extract data from a character array that has a null terminator, allocate and initialize the string, then construct an object of class istrstream.

In This Section

Constructing Input Stream Objects

Using Extraction Operators

Testing for Extraction Errors

Input Stream Manipulators

Input Stream Member Functions

Overloading the >> Operator for Your Own Classes

See Also

Reference

iostream Programming

Constructing Input Stream Objects

If you use only the **cin** object, you do not need to construct an input stream. You must construct an input stream if you use:

- Input File Stream Constructors
- Input String Stream Constructors

Input File Stream Constructors

There are two ways to create an input file stream:

- Use the **void** argument constructor, then call the **open** member function:

```
ifstream myFile; // On the stack
myFile.open( "filename" );

ifstream* pmyFile = new ifstream; // On the heap
pmyFile->open( "filename" );
```

- Specify a filename and mode flags in the constructor invocation, thereby opening the file during the construction process:

```
ifstream myFile( "filename" );
```

Input String Stream Constructors

Input string stream constructors require the address of preallocated, preinitialized storage:

```
char s[] = "123.45";
double amt;
istrstream myString( s );
myString >> amt; // Amt should contain 123.45
```

See Also
Reference
Input Streams

Using Extraction Operators

The extraction operator (**>>**), which is preprogrammed for all standard C++ data types, is the easiest way to get bytes from an input stream object.

Formatted text input extraction operators depend on white space to separate incoming data values. This is inconvenient when a text field contains multiple words or when commas separate numbers. In such a case, one alternative is to use the unformatted input member function istream::getline to read a block of text with white space included, then parse the block with special functions. Another method is to derive an input stream class with a member function such as `GetNextToken`, which can call istream members to extract and format character data.

See Also

Reference

Input Streams

Testing for Extraction Errors

Output error processing functions, discussed in Error Processing Functions, apply to input streams. Testing for errors during extraction is important. Consider this statement:

```
cin >> n;
```

If `n` is a signed integer, a value greater than 32,767 (the maximum allowed value, or MAX_INT) sets the stream's **fail** bit, and the **cin** object becomes unusable. All subsequent extractions result in an immediate return with no value stored.

See Also
Reference
Input Streams

Input Stream Manipulators

Many manipulators, such as setprecision, are defined for the **ios** class and thus apply to input streams. Few manipulators, however, actually affect input stream objects. Of those that do, the most important are the radix manipulators, **dec**, **oct**, and **hex**, which determine the conversion base used with numbers from the input stream.

On extraction, the **hex** manipulator enables processing of various input formats. For example, c, C, 0xc, 0xC, 0Xc, and 0XC are all interpreted as the decimal integer 12. Any character other than 0 through 9, A through F, a through f, x, and X terminates the numeric conversion. Thus the sequence `"124n5"` is converted to the number 124 with the basic_ios::fail bit set.

See Also

Reference

Input Streams

Input Stream Member Functions

Input stream member functions are used for disk input. The member functions include:

- The open Function for Input Streams
- The get Function
- The getline Function
- The read Function
- The seekg and tellg Functions
- The close Function for Input Streams

The open Function for Input Streams

If you are using an input file stream (ifstream), you must associate that stream with a specific disk file. You can do this in the constructor, or you can use the **open** function. In either case, the arguments are the same.

You generally specify an ios_base::openmode flag when you open the file associated with an input stream (the default mode is **ios::in**). For a list of the **open_mode** flags, see The open Function. The flags can be combined with the bitwise OR (|) operator.

To read a file, first use the **fail** member function to determine whether it exists:

```
istream ifile( "FILENAME" );
if ( ifile.fail() )
// The file does not exist ...
```

The get Function

The unformatted **get** member function works like the **>>** operator with two exceptions. First, the **get** function includes white-space characters, whereas the extractor excludes white space when the skipws flag is set (the default). Second, the **get** function is less likely to cause a tied output stream (**cout**, for example) to be flushed.

A variation of the **get** function specifies a buffer address and the maximum number of characters to read. This is useful for limiting the number of characters sent to a specific variable, as this example shows:

```
// ioo_get_function.cpp
// compile with: /EHsc
// Type up to 24 characters and a terminating character.
// Any remaining characters can be extracted later.
#include <iostream>
using namespace std;

int main()
{
   char line[25];
   cout << " Type a line terminated by carriage return\n>";
   cin.get( line, 25 );
   cout << line << endl;
}
```

Input

```
1234
```

Sample Output

```
1234
```

The getline Function

The **getline** member function is similar to the **get** function. Both functions allow a third argument that specifies the terminating character for input. The default value is the newline character. Both functions reserve one character for the

required terminating character. However, **get** leaves the terminating character in the stream and **getline** removes the terminating character.

The following example specifies a terminating character for the input stream:

```
// getline_func.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
   char line[100];
   cout << " Type a line terminated by 't'" << endl;
   cin.getline( line, 100, 't' );
   cout << line;
}
```

Input

```
test
```

The read Function

The **read** member function reads bytes from a file to a specified area of memory. The length argument determines the number of bytes read. If you do not include that argument, reading stops when the physical end of file is reached or, in the case of a text-mode file, when an embedded **EOF** character is read.

This example reads a binary record from a payroll file into a structure:

```
#include <fstream>
#include <iostream>
using namespace std;

int main()
{
   struct
   {
      double salary;
      char name[23];
   } employee;

   ifstream is( "payroll" );
   if( is ) {  // ios::operator void*()
      is.read( (char *) &employee, sizeof( employee ) );
      cout << employee.name << ' ' << employee.salary << endl;
   }
   else {
      cout << "ERROR: Cannot open file 'payroll'." << endl;
   }
}
```

The program assumes that the data records are formatted exactly as specified by the structure with no terminating carriage-return or linefeed characters.

The seekg and tellg Functions

Input file streams keep an internal pointer to the position in the file where data is to be read next. You set this pointer with the **seekg** function, as shown here:

```
#include <iostream>
#include <fstream>
using namespace std;

int main( )
{
   char ch;
```

```
   ifstream tfile( "payroll" );
   if( tfile ) {
      tfile.seekg( 8 );        // Seek 8 bytes in (past salary)
      while ( tfile.good() ) { // EOF or failure stops the reading
         tfile.get( ch );
         if( !ch ) break;      // quit on null
         cout << ch;
      }
   }
   else {
      cout << "ERROR: Cannot open file 'payroll'." << endl;
   }
}
```

To use **seekg** to implement record-oriented data management systems, multiply the fixed-length record size by the record number to obtain the byte position relative to the end of the file, and then use the **get** object to read the record.

The **tellg** member function returns the current file position for reading. This value is of type **streampos**, a **typedef** defined in <iostream>. The following example reads a file and displays messages showing the positions of spaces.

```
#include <fstream>
#include <iostream>
using namespace std;

int main( )
{
   char ch;
   ifstream tfile( "payroll" );
   if( tfile ) {
       while ( tfile.good( ) ) {
          streampos here = tfile.tellg();
          tfile.get( ch );
          if ( ch == ' ' )
             cout << "\nPosition " << here << " is a space";
       }
   }
   else {
      cout << "ERROR: Cannot open file 'payroll'." << endl;
   }
}
```

The close Function for Input Streams

The **close** member function closes the disk file associated with an input file stream and frees the operating system file handle. The ifstream destructor closes the file for you, but you can use the **close** function if you need to open another file for the same stream object.

See Also
Reference
Input Streams

Overloading the >> Operator for Your Own Classes

Input streams use the extraction (**>>**) operator for the standard types. You can write similar extraction operators for your own types; your success depends on using white space precisely.

Here is an example of an extraction operator for the `Date` class presented earlier:

```
istream& operator>> ( istream& is, Date& dt )
{
   is >> dt.mo >> dt.da >> dt.yr;
   return is;
}
```

See Also

Reference

Input Streams

Input/Output Streams

An iostream object is a source and/or a destination for bytes. The two most important I/O stream classes, both derived from iostream, are fstream and strstream. These classes inherit the functionality of the istream and ostream classes described previously.

The fstream class supports disk file input and output. If you need to read from and write to a particular disk file in the same program, construct an fstream object. An fstream object is a single stream with two logical substreams, one for input and one for output. Although the underlying buffer contains separately designated positions for reading and writing, those positions are tied together.

The strstream class supports input and output of in-memory strings.

See Also

Reference

iostream Programming

Custom Manipulators with Arguments

This section describes how to create output stream manipulators with one or more arguments, and how to use manipulators for nonoutput streams.

In This Section

Output Stream Manipulators with One Argument (**int** or **long**)

Other One-Argument Output Stream Manipulators

See Also

Reference

iostream Programming

Output Stream Manipulators with One Argument (int or long)

The iostream class library provides a set of macros for creating parameterized manipulators. Manipulators with a single **int** or **long** argument are a special case. To create an output stream manipulator that accepts a single **int** or **long** argument (like **setw**), you must use the _Smanip macro, which is defined in <iomanip>. This example defines a `fillblank` manipulator that inserts a specified number of blanks into the stream:

Example

```
// output_stream_manip.cpp
// compile with: /GR /EHsc
#include <iostream>
#include <iomanip>
using namespace std;

void fb( ios_base& os, int l )
{
   ostream *pos = dynamic_cast<ostream*>(&os);
   if (pos)
   {
      for( int i=0; i < l; i++ )
      (*pos) << ' ';
   };
}

_Smanip<int>
   __cdecl fillblank(int no)
   {
   return (_Smanip<int>(&fb, no));
   }

int main( )
{
   cout << "10 blanks follow" << fillblank( 10 ) << ".\n";
}
```

See Also

Reference

Custom Manipulators with Arguments

Other One-Argument Output Stream Manipulators

The following example uses a class `money`, which is a **long** type. The `setpic` manipulator attaches a formatting "picture" string to the class that can be used by the overloaded stream insertion operator of the class `money`. The picture string is stored as a static variable in the `money` class rather than as data member of a stream class, so you do not have to derive a new output stream class.

Example

```
// one_arg_output.cpp
// compile with: /GR /EHsc
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

typedef char* charp;

class money
{
private:
    long value;
    static char *szCurrentPic;
public:
    money( long val ) { value = val; }
    friend ostream& operator << ( ostream& os, money m ) {
        // A more complete function would merge the picture
        // with the value rather than simply appending it
        os << m.value << '[' << money::szCurrentPic << ']';
        return os;
    }
    static void setpic( char* szPic ) {
        money::szCurrentPic = new char[strlen( szPic ) + 1];
        strcpy_s( money::szCurrentPic, strlen( szPic ) + 1, szPic );
    }
};

char *money::szCurrentPic;  // Static pointer to picture

void fb( ios_base& os, char * somename )
{
   money::setpic(somename);
/*
   ostream *pos = dynamic_cast<ostream*>(&os);
   if (pos)
   {
      for( int i=0; i < l; i++ )
      (*pos) << ' ';
   };
*/
}

_Smanip<charp>
   __cdecl setpic(char * somename)
   {
   return (_Smanip<charp>(&fb, somename));
   }

int main( )
{
    money amt = (long)35235.22;
    cout << setiosflags( ios::fixed );
    cout << setpic( "###,###,###.##" ) << "amount = " << amt << endl;
}
```

See Also

Reference

Custom Manipulators with Arguments

Deriving Your Own Stream Classes

Like any C++ class, a stream class can be derived to add new member functions, data members, or manipulators. If you need an input file stream that tokenizes its input data, for example, you can derive from the ifstream class. This derived class can include a member function that returns the next token by calling its base class's public member functions or extractors. You may need new data members to hold the stream object's state between operations, but you probably won't need to use the base class's protected member functions or data members.

For the straightforward stream class derivation, you only have to write the necessary constructors and the new member functions.

In This Section

The streambuf Class

Why Derive a Custom streambuf Class?

See Also

Reference

iostream Programming

The streambuf Class

Unless you plan to make major changes to the iostream library, you do not need to work much with the streambuf class, which does most of the work for the other stream classes. In most cases, you will create a modified output stream by deriving only a new streambuf class and connecting it to the ostream class.

See Also

Reference

Deriving Your Own Stream Classes

Why Derive a Custom streambuf Class?

Existing output streams communicate to the file system and to in-memory strings. You can create streams that address a memory-mapped video screen, a window as defined by Microsoft Windows, a new physical device, and so on. A simpler method is to alter the byte stream as it goes to a file system device.

See Also

Reference

Deriving Your Own Stream Classes

Standard C++ Library Header Files

The following Standard C++ Library header files are documented:

<algorithm>	<bitset>	<complex>
<deque>	<exception>	<fstream>
<functional>	<hash_map>	<hash_set>
<iomanip>	<ios>	<iosfwd>
<iostream>	<iso646.h>	<istream>
<iterator>	<limits>	<list>
<locale>	<map>	<memory>
<new>	<numeric>	<ostream>
<queue>	<set>	<sstream>
<stack>	<stdexcept>	<streambuf>
<string>	<strstream>	<utility>
<valarray>	<vector>	

In addition, the following C++ wrappers are documented:

<cassert>	<cctype>	<cerrno>
<cfloat>	<ciso646>	<climits>
<clocale>	<cmath>	<csetjmp>
<csignal>	<cstdarg>	<cstddef>
<cstdio>	<cstdlib>	<cstring>
<ctime>	<cwchar>	<cwctype>

For more information about Standard C++ Library header files, see the Standard C++ Library Overview.

See Also

Other Resources

Standard C++ Library Reference

<algorithm>

Defines Standard Template Library (STL) container template functions that perform algorithms.

For a list of all members of this header, see <algorithm> Members.

```
#include <algorithm>
```

Remarks

The STL algorithms are generic because they can operate on a variety of data structures. The data structures that they can operate on include not only the STL container classes such as vector and list, but also program-defined data structures and arrays of elements that satisfy the requirements of a particular algorithm. STL algorithms achieve this level of generality by accessing and traversing the elements of a container indirectly through iterators. STL algorithms process iterator ranges that are typically specified by their beginning or ending positions. The ranges referred to must be valid in the sense that all pointers in the ranges must be dereferenceable and within the sequences of each range, the last position must be reachable from the first by incrementation.

The STL algorithms extend the actions supported by the operations and member functions of each STL container and allow working, for example, with different types of container objects at the same time. Two suffixes have been used to convey information about the purpose of the algorithms.

- The **_if** suffix indicates that the algorithm is used with function objects operating on the values of the elements rather than on the values of the elements themselves. The **find_if** algorithm looks for elements whose values satisfy the criterion specified by a function object, and the **find** algorithm searches for a particular value.
- The **copy** suffix indicates that the algorithm not only manipulates the values of the elements but also copies the modified values into a destination range. The **reverse** algorithm reverses the order of the elements within a range, and the **reverse_copy** algorithm also copies the result into a destination range.

STL algorithms are often classified into groups that indicate something about their purpose or requirements. These include modifying algorithms that change the value of elements as compared with nonmodifying algorithms that do not. Mutating algorithms change the order of elements, but not the values of their elements. Removing algorithms can eliminate elements from a range or a copy of a range. Sorting algorithms reorder the elements in a range in various ways and sorted range algorithms only act on algorithms whose elements have been sorted in a particular way.

The STL numeric algorithms that are provided for numerical processing have their own header file <numeric>, and function objects and adaptors are defined in the header <functional>. Function objects that return Boolean values are known as predicates. The default binary predicate is the comparison **operator<**. In general, the elements being ordered need to be less than comparable so that, given any two elements, it can be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering among the nonequivalent elements.

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<algorithm> Members

Standard C++ Library Header Files

<algorithm> Members

Reference

Functions

adjacent_find	Searches for two adjacent elements that are either equal or satisfy a specified condition.
binary_search	Tests whether there is an element in a sorted range that is equal to a specified value or t hat is equivalent to it in a sense specified by a binary predicate.
checked_copy	Same as **copy**, but enforces the use of a checked iterator as output iterator.
checked_copy_backward	Same as **copy_backward**, but enforces the use of a checked iterator as output iterator.
checked_fill_n	Same as **fill_n**, but enforces the use of a checked iterator as output iterator.
checked_generate_n	Same as **generate_n**, but enforces the use of a checked iterator as output iterator.
checked_merge	Same as **merge**, but enforces the use of a checked iterator as output iterator.
checked_remove_copy	Same as **remove_copy**, but enforces the use of a checked iterator as output iterator.
checked_remove_copy_if	Same as **remove_copy_if** but enforces the use of a checked iterator as output iterator.
checked_replace_copy	Same as **replace_copy**, but enforces the use of a checked iterator as output iterator.
checked_replace_copy_if	Same as **replace_copy_if** but enforces the use of a checked iterator as output iterator.
checked_reverse_copy	Same as **reverse_copy** but enforces the use of a checked iterator as output iterator.
checked_rotate_copy	Same as **rotate_copy** but enforces the use of a checked iterator as output iterator.
checked_set_difference	Same as **set_difference** but enforces the use of a checked iterator as output iterator.
checked_set_intersection	Same as **set_intersection** but enforces the use of a checked iterator as output iterator.
checked_set_symmetric_difference	Same as **set_symmetric_difference** but enforces the use of a checked iterator as outpu t iterator.
checked_set_union	Same as **set_union** but enforces the use of a checked iterator as output iterator.
checked_unique_copy	Same as **unique_copy** but enforces the use of a checked iterator as output iterator.
copy	Assigns the values of elements from a source range to a destination range, iterating thro ugh the source sequence of elements and assigning them new positions in a forward dir ection.
copy_backward	Assigns the values of elements from a source range to a destination range, iterating thro ugh the source sequence of elements and assigning them new positions in a backward d irection.
count	Returns the number of elements in a range whose values match a specified value.

count_if	Returns the number of elements in a range whose values match a specified condition.
equal	Compares two ranges element by element either for equality or equivalence in a sense s pecified by a binary predicate.
equal_range	Finds a pair of positions in an ordered range, the first less than or equivalent to the positi on of a specified element and the second greater than the element's position, where the sense of equivalence or ordering used to establish the positions in the sequence may be specified by a binary predicate.
fill	Assigns the same new value to every element in a specified range.
fill_n	Assigns a new value to a specified number of elements in a range beginning with a parti cular element.
find	Locates the position of the first occurrence of an element in a range that has a specified value.
find_end	Looks in a range for the last subsequence that is identical to a specified sequence or that is equivalent in a sense specified by a binary predicate.
find_first_of	Searches for the first occurrence of any of several values within a target range or for the first occurrence of any of several elements that are equivalent in a sense specified by a bi nary predicate to a specified set of the elements.
find_if	Locates the position of the first occurrence of an element in a range that satisfies a specif ied condition.
for_each	Applies a specified function object to each element in a forward order within a range and returns the function object.
generate	Assigns the values generated by a function object to each element in a range.
generate_n	Assigns the values generated by a function object to a specified number of element is a r ange and returns to the position one past the last assigned value.
includes	Tests whether one sorted range contains all the elements contained in a second sorted r ange, where the ordering or equivalence criterion between elements may be specified by a binary predicate.
inplace_merge	Combines the elements from two consecutive sorted ranges into a single sorted range, where the ordering criterion may be specified by a binary predicate.
iter_swap	Exchanges two values referred to by a pair of specified iterators.
lexicographical_compare	Compares element by element between two sequences to determine which is lesser of t he two.
lower_bound	Finds the position of the first element in an ordered range that has a value less than or e quivalent to a specified value, where the ordering criterion may be specified by a binary predicate.
make_heap	Converts elements from a specified range into a heap in which the first element is the lar gest and for which a sorting criterion may be specified with a binary predicate.

max	Compares two objects and returns the larger of the two, where the ordering criterion may be specified by a binary predicate.
max_element	Finds the first occurrence of largest element in a specified range where the ordering criterion may be specified by a binary predicate.
merge	Combines all the elements from two sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.
min	Compares two objects and returns the lesser of the two, where the ordering criterion may be specified by a binary predicate.
min_element	Finds the first occurrence of smallest element in a specified range where the ordering criterion may be specified by a binary predicate.
mismatch	Compares two ranges element by element either for equality or equivalent in a sense specified by a binary predicate and locates the first position where a difference occurs.
next_permutation	Reorders the elements in a range so that the original ordering is replaced by the lexicographically next greater permutation if it exists, where the sense of next may be specified with a binary predicate.
nth_element	Partitions a range of elements, correctly locating the *n*th element of the sequence in the range so that all the elements in front of it are less than or equal to it and all the elements that follow it in the sequence are greater than or equal to it.
partial_sort	Arranges a specified number of the smaller elements in a range into a nondescending order or according to an ordering criterion specified by a binary predicate.
partial_sort_copy	Copies elements from a source range into a destination range where the source elements are ordered by either less than or another specified binary predicate.
partition	Classifies elements in a range into two disjoint sets, with those elements satisfying a unary predicate preceding those that fail to satisfy it.
pop_heap	Removes the largest element from the front of a heap to the next-to-last position in the range and then forms a new heap from the remaining elements.
prev_permutation	Reorders the elements in a range so that the original ordering is replaced by the lexicographically next greater permutation if it exists, where the sense of next may be specified with a binary predicate.
push_heap	Adds an element that is at the end of a range to an existing heap consisting of the prior elements in the range.
random_shuffle	Rearranges a sequence of *N* elements in a range into one of *N*! possible arrangements selected at random.
remove	Eliminates a specified value from a given range without disturbing the order of the remaining elements and returning the end of a new range free of the specified value.
remove_copy	Copies elements from a source range to a destination range, except that elements of a specified value are not copied, without disturbing the order of the remaining elements and returning the end of a new destination range.

remove_copy_if	Copies elements from a source range to a destination range, except that satisfying a predicate are not copied, without disturbing the order of the remaining elements and returning the end of a new destination range.
remove_if	Eliminates elements that satisfy a predicate from a given range without disturbing the order of the remaining elements and returning the end of a new range free of the specified value.
replace	Examines each element in a range and replaces it if it matches a specified value.
replace_copy	Examines each element in a source range and replaces it if it matches a specified value while copying the result into a new destination range.
replace_copy_if	Examines each element in a source range and replaces it if it satisfies a specified predicate while copying the result into a new destination range.
replace_if	Examines each element in a range and replaces it if it satisfies a specified predicate.
reverse	Reverses the order of the elements within a range.
reverse_copy	Reverses the order of the elements within a source range while copying them into a destination range
rotate	Exchanges the elements in two adjacent ranges.
rotate_copy	Exchanges the elements in two adjacent ranges within a source range and copies the result to a destination range.
search	Searches for the first occurrence of a sequence within a target range whose elements are equal to those in a given sequence of elements or whose elements are equivalent in a sense specified by a binary predicate to the elements in the given sequence.
search_n	Searches for the first subsequence in a range that of a specified number of elements having a particular value or a relation to that value as specified by a binary predicate.
set_difference	Unites all of the elements that belong to one sorted source range, but not to a second sorted source range, into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.
set_intersection	Unites all of the elements that belong to both sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.
set_symmetric_difference	Unites all of the elements that belong to one, but not both, of the sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.
set_union	Unites all of the elements that belong to at least one of two sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.
sort	Arranges the elements in a specified range into a nondescending order or according to an ordering criterion specified by a binary predicate.
sort_heap	Converts a heap into a sorted range.

stable_partition	Classifies elements in a range into two disjoint sets, with those elements satisfying a unary predicate preceding those that fail to satisfy it, preserving the relative order of equivalent elements.
stable_sort	Arranges the elements in a specified range into a nondescending order or according to an ordering criterion specified by a binary predicate and preserves the relative ordering of equivalent elements.
swap	Exchanges the values of the elements between two types of objects, assigning the contents of the first object to the second object and the contents of the second to the first.
swap_ranges	Exchanges the elements of one range with the elements of another, equal sized range.
transform	Applies a specified function object to each element in a source range or to a pair of elements from two source ranges and copies the return values of the function object into a destination range.
unchecked_copy	Same as **copy**, but enforces the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.
unchecked_copy_backward	Same as **copy_backward**, but enforces the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.
unchecked_fill_n	Same as **fill_n**, but enforces the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.
unchecked_generate_n	Same as **generate_n**, but enforces the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.
unchecked_merge	Same as **merge**, but enforces the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.
unchecked_remove_copy	Same as **remove_copy**, but enforces the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.
unchecked_remove_copy_if	Same as **remove_copy_if**, but enforces the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.
unchecked_replace_copy	Same as **replace_copy**, but enforces the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.
unchecked_replace_copy_if	Same as **replace_copy_if**, but enforces the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.
unchecked_reverse_copy	Same as **reverse_copy**, but enforces the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.
unchecked_rotate_copy	Same as **rotate_copy**, but enforces the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.
unchecked_set_difference	Same as **set_difference**, but enforces the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.
unchecked_set_intersection	Same as **set_intersection**, but enforces the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.

unchecked_set_symmetric_difference	Same as **set_symmetric_difference**, but enforces the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.
unchecked_set_union	Same as **set_union**, but enforces the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.
unchecked_unique_copy	Same as **unique_copy**, but enforces the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.
unique	Removes duplicate elements that are adjacent to each other in a specified range.
unique_copy	Copies elements from a source range into a destination range except for the duplicate elements that are adjacent to each other.
upper_bound	Finds the position of the first element in an ordered range that has a value that is greater than a specified value, where the ordering criterion may be specified by a binary predicate.

See Also

Reference

<algorithm>
Thread Safety in the Standard C++ Library
Standard Template Library

<algorithm> Functions

For more information about the functions in <algorithm>, see <algorithm> Members.

See Also

Reference

Standard Template Library

adjacent_find

Searches for two adjacent elements that are either equal or satisfy a specified condition.

```
template<class ForwardIterator>
   ForwardIterator adjacent_find(
      ForwardIterator _First,
      ForwardIterator _Last
      );
template<class ForwardIterator , class BinaryPredicate>
   ForwardIterator adjacent_find(
      ForwardIterator _First,
      ForwardIterator _Last,
            BinaryPredicate _Comp
   );
```

Parameters

_First

A forward iterator addressing the position of the first element in the range to be searched.

_Last

A forward iterator addressing the position one past the final element in the range to be searched.

_Comp

The binary predicate giving the condition to be satisfied by the values of the adjacent elements in the range being searched.

Return Value

A forward iterator to the first element of the adjacent pair that are either equal to each other (in the first version) or that satisfy the condition given by the binary predicate (in the second version), provided that such a pair of elements is found. Otherwise, an iterator pointing to *_Last* is returned.

Remarks

The **adjacent_find** algorithm is a nonmutating sequence algorithm. The range to be searched must be valid; all pointers must be dereferenceable and the last position is reachable from the first by incrementation. The time complexity of the algorithm is linear in the number of elements contained in the range.

The **operator==** used to determine the match between elements must impose an equivalence relation between its operands.

Example

```
// alg_adj_fnd.cpp
// compile with: /EHsc
#include <list>
#include <algorithm>
#include <iostream>

// Returns whether second element is twice the first
bool twice (int elem1, int elem2 )
{
   return elem1 * 2 == elem2;
}

int main( )
{
   using namespace std;
   list <int> L;
   list <int>::iterator Iter;
   list <int>::iterator result1, result2;

   L.push_back( 50 );
   L.push_back( 40 );
```

```
    L.push_back( 10 );
    L.push_back( 20 );
    L.push_back( 20 );

    cout << "L = ( " ;
    for ( Iter = L.begin( ) ; Iter != L.end( ) ; Iter++ )
       cout << *Iter << " ";
    cout << ")" << endl;

    result1 = adjacent_find( L.begin( ), L.end( ) );
    if ( result1 == L.end( ) )
       cout << "There are not two adjacent elements that are equal."
            << endl;
    else
       cout << "There are two adjacent elements that are equal."
            << "\n They have a value of "
            <<  *( result1 ) << "." << endl;

    result2 = adjacent_find( L.begin( ), L.end( ), twice );
    if ( result2 == L.end( ) )
       cout << "There are not two adjacent elements where the "
            << " second is twice the first." << endl;
    else
       cout << "There are two adjacent elements where "
            << "the second is twice the first."
            << "\n They have values of " << *(result2++);
       cout << " & " << *result2 << "." << endl;
 }
```

Output

```
L = ( 50 40 10 20 20 )
There are two adjacent elements that are equal.
 They have a value of 20.
There are two adjacent elements where the second is twice the first.
 They have values of 10 & 20.
```

Requirements

Header: <algorithm>

See Also

Reference

Nonpredicate Version of adjacent_find

Predicate Version of adjacent_find

Standard Template Library

Other Resources

<algorithm> Members

binary_search

Tests whether there is an element in a sorted range that is equal to a specified value or that is equivalent to it in a sense specified by a binary predicate.

```
template<class ForwardIterator, class Type>
   bool binary_search(
      ForwardIterator _First,
      ForwardIterator _Last,
      const Type& _Val
   );
template<class ForwardIterator, class Type, class BinaryPredicate>
   bool binary_search(
      ForwardIterator _First,
      ForwardIterator _Last,
      const Type& _Val,
      BinaryPredicate _Comp
   );
```

Parameters

_First

A forward iterator addressing the position of the first element in the range to be searched.

_Last

A forward iterator addressing the position one past the final element in the range to be searched.

_Val

The value required to be matched by the value of the element or that must satisfy the condition with the element value specified by the binary predicate.

_Comp

User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

true if an element is found in the range that is equal or equivalent to the specified value; otherwise, **false**.

Remarks

The sorted source range referenced must be valid; all pointers must be dereferenceable and, within the sequence, the last position must be reachable from the first by incrementation.

The sorted range must each be arranged as a precondition to the application of the **binary_search** algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The source ranges are not modified by **binary_search**.

The value types of the forward iterators need to be less-than comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements

The complexity of the algorithm is logarithmic for random-access iterators and linear otherwise, with the number of steps proportional to (*_Last* – *_First*).

Example

```
// alg_bin_srch.cpp
// compile with: /EHsc
#include <list>
#include <vector>
#include <algorithm>
#include <iostream>
```

```
// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
   if (elem1 < 0)
      elem1 = - elem1;
   if (elem2 < 0)
      elem2 = - elem2;
   return elem1 < elem2;
}

int main( )
{
   using namespace std;

   list <int> L;
   list <int>::iterator Iter;
   bool b1, b2;

   L.push_back( 50 );
   L.push_back( 10 );
   L.push_back( 30 );
   L.push_back( 20 );
   L.push_back( 25 );
   L.push_back( 5 );

   L.sort( );

   cout << "L = ( " ;
   for ( Iter = L.begin( ) ; Iter != L.end( ) ; Iter++ )
      cout << *Iter << " ";
   cout << ")" << endl;

   b1 = binary_search( L.begin( ), L.end( ), 10 );
   if  ( b1 )
      cout << "There is an element in list L with a value equal to 10."
           << endl;
   else
      cout << "There is no element in list L with a value equal to 10."
           << endl;
   // a binary_search under the binary predicate greater
   L.sort ( greater<int> ( ) );
   b2 = binary_search( L.begin( ), L.end( ), 10 , greater<int> ( ) );
   if  ( b2 )
      cout << "There is an element in list L with a value equivalent to 10 "
           << "under greater than." << endl;
   else
      cout << "No element in list L with a value equivalent to 10 "
           << "under greater than." << endl;

   // a binary_search under the user-defined binary predicate mod_lesser
   vector <int> v1;
   vector <int>::iterator Iter1;
   int i;
   for ( i = -2 ; i <= 4 ; i++ )
   {
      v1.push_back( i );
   }
   sort ( v1.begin ( ) , v1.end ( ) , mod_lesser );

   cout << "Ordered under mod_lesser, vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   bool b3 = binary_search( v1.begin( ), v1.end( ), -3 , mod_lesser );
   if ( b3 )
      cout << "There is an element with a value equivalent to -3 "
```

```
            << "under mod_lesser." << endl;
    else
       cout << "There is not an element with a value equivalent to -3 "
            << "under mod_lesser." << endl;
 }
```

Output

```
L = ( 5 10 20 25 30 50 )
There is an element in list L with a value equal to 10.
There is an element in list L with a value equivalent to 10 under greater than.
Ordered under mod_lesser, vector v1 = ( 0 -1 1 -2 2 3 4 )
There is an element with a value equivalent to -3 under mod_lesser.
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

checked_copy

Same as copy, but enforces the use of a checked iterator as output iterator. **checked_copy** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator, class OutputIterator>
   OutputIterator checked_copy(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _DestBeg
   );
```

Parameters

_First

An input iterator addressing the position of the first element in the source range.

_Last

An input iterator addressing the position that is one past the final element in the source range.

_DestBeg

An output iterator addressing the position of the first element in the destination range.

Return Value

An output iterator addressing the position that is one past the final element in the destination range, that is, the iterator addresses *_Result* + (*_Last* – *_First*).

Remarks

See copy for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

checked_copy_backward

Same as copy_backward, but enforces the use of a checked iterator as output iterator. **checked_copy_backward** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
   BidirectionalIterator2 checked_copy_backward(
      BidirectionalIterator1 _First,
      BidirectionalIterator1 _Last,
      BidirectionalIterator2 _DestEnd
   );
```

Parameters

_First

A bidirectional iterator addressing the position of the first element in the source range.

_Last

A bidirectional iterator addressing the position that is one past the final element in the source range.

_DestEnd

A bidirectional iterator addressing the position of the one past the final element in the destination range.

Return Value

An output iterator addressing the position that is one past the final element in the destination range, that is, the iterator addresses _ *DestEnd* – (_*Last* – _*First*).

Remarks

See copy_backward for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

checked_fill_n

Same as fill_n, but enforces the use of a checked iterator as output iterator. **checked_fill_n** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class OutputIterator, class Size, class Type>
   void checked_fill_n(
      OutputIterator _First,
      Size _Count,
      const Type& _Val
   );
```

Parameters

_First

An output iterator addressing the position of the first element in the range to be assigned the value *_Val*.

_Count

A signed or unsigned integer type specifying the number of elements to be assigned the value.

_Val

The value to be assigned to elements in the range [*_First*, *_First* + *_Count*).

Remarks

See fill_n for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

checked_generate_n

Same as generate_n, but enforces the use of a checked iterator as output iterator. **checked_generate_n** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class OutputIterator, class Size, class Generator>
   void checked_generate_n(
      OutputIterator _First,
      Size _Count,
      Generator _Gen
   );
```

Parameters

_First

An output iterator addressing the position of first element in the range to which values are to be assigned.

_Count

A signed or unsigned integer type specifying the number of elements to be assigned a value by the generator function.

_Gen

A function object that is called with no arguments that is used to generate the values to be assigned to each of the elements in the range.

Remarks

See generate_n for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

checked_merge

Same as merge, but enforces the use of a checked iterator as output iterator. **checked_merge** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
   OutputIterator checked_merge(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPred
icate>
   OutputIterator checked_merge(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
      BinaryPredicate _Comp
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be combined and sorted into a single range.

_Last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be combined and sorted into a single range.

_First2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be combined and sorted into a single range.

_Last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be combined and sorted into a single range.

_Result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be combined into a single sorted range.

_Comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range.

Remarks

See merge for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

checked_remove_copy

Same as remove_copy, but enforces the use of a checked iterator as output iterator. **checked_remove_copy** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator, class OutputIterator, class Type>
   OutputIterator checked_remove_copy(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      const Type& _Val
   );
```

Parameters

_First

An input iterator addressing the position of the first element in the range from which elements are being removed.

_Last

An input iterator addressing the position one past the final element in the range from which elements are being removed.

_Result

An output iterator addressing the position of the first element in the destination range to which elements are being removed.

_Val

The value that is to be removed from the range.

Return Value

A forward iterator addressing the new end position of the destination range, one past the final element of the copy of the remnant sequence free of the specified value.

Remarks

See remove_copy for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

checked_remove_copy_if

Same as remove_copy_if but enforces the use of a checked iterator as output iterator. **checked_remove_copy_if** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator, class OutputIterator, class Predicate>
   OutputIterator checked_remove_copy_if(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      Predicate _Pred
   );
```

Parameters

_First

An input iterator addressing the position of the first element in the range from which elements are being removed.

_Last

An input iterator addressing the position one past the final element in the range from which elements are being removed.

_Result

An output iterator addressing the position of the first element in the destination range to which elements are being removed.

_Pred

The unary predicate that must be satisfied is the value of an element is to be replaced.

Return Value

A forward iterator addressing the new end position of the destination range, one past the final element of the remnant sequence free of the elements satisfying the predicate.

Remarks

See remove_copy_if for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

checked_replace_copy

Same as replace_copy, but enforces the use of a checked iterator as output iterator. **checked_replace_copy** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator, class OutputIterator, class Type>
   OutputIterator checked_replace_copy(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      const Type& _OldVal,
      const Type& _NewVal
   );
```

Parameters

_First

An input iterator pointing to the position of the first element in the range from which elements are being replaced.

_Last

An input iterator pointing to the position one past the final element in the range from which elements are being replaced.

_Result

An output iterator pointing to the first element in the destination range to where the altered sequence of elements is being copied.

_OldVal

The old value of the elements being replaced.

_NewVal

The new value being assigned to the elements with the old value.

Return Value

An output iterator pointing to the position one past the final element in the destination range to where the altered sequence of elements is being copied.

Remarks

See replace_copy for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

checked_replace_copy_if

Same as replace_copy_if but enforces the use of a checked iterator as output iterator. **checked_replace_copy_if** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator, class OutputIterator, class Predicate, class Type>
   OutputIterator checked_replace_copy_if(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      Predicate _Pred,
      const Type& _Val
   );
```

Parameters

_First

An input iterator pointing to the position of the first element in the range from which elements are being replaced.

_Last

An input iterator pointing to the position one past the final element in the range from which elements are being replaced.

_Result

An output iterator pointing to the first element in the destination range to where the altered sequence of elements is being copied.

_OldVal

The old value of the elements being replaced.

_NewVal

The new value being assigned to the elements with the old value.

Return Value

An output iterator pointing to the position one past the final element in the destination range to where the altered sequence of elements is being copied.

Remarks

See replace_copy_if for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

checked_reverse_copy

Same as reverse_copy but enforces the use of a checked iterator as output iterator. **checked_reverse_copy** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class BidirectionalIterator, class OutputIterator>
   OutputIterator checked_reverse_copy(
      BidirectionalIterator _First,
      BidirectionalIterator _Last,
      OutputIterator _Result
   );
```

Parameters

_First

A bidirectional iterator pointing to the position of the first element in the source range within which the elements are being permuted.

_Last

A bidirectional iterator pointing to the position one past the final element in the source range within which the elements are being permuted.

_Result

An output iterator pointing to the position of the first element in the destination range to which elements are being copied.

Return Value

An output iterator pointing to the position one past the final element in the destination range to where the altered sequence of elements is being copied.

Remarks

See reverse_copy for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

checked_rotate_copy

Same as rotate_copy but enforces the use of a checked iterator as output iterator. **checked_rotate_copy** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class ForwardIterator, class OutputIterator>
   OutputIterator checked_rotate_copy(
      ForwardIterator _First,
      ForwardIterator _Middle,
      ForwardIterator _Last,
      OutputIterator _Result
   );
```

Parameters

_First

A forward iterator addressing the position of the first element in the range to be rotated.

_Middle

A forward iterator defining the boundary within the range that addresses the position of the first element in the second part of the range whose elements are to be exchanged with those in the first part of the range.

_ Last

A forward iterator addressing the position one past the final element in the range to be rotated.

_Result

An output iterator addressing the position of the first element in the destination range.

Return Value

An output iterator addressing the position one past the final element in the destination range.

Remarks

See rotate_copy for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

checked_set_difference

Same as set_difference but enforces the use of a checked iterator as output iterator. **checked_set_difference** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
   OutputIterator checked_set_difference(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPred
icate>
   OutputIterator checked_set_difference(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result,
      BinaryPredicate _Comp
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

_Last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

_First2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

_Last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

_Result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the difference of the two source ranges.

_Comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range representing the difference of the two source ranges.

Remarks

See set_difference for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

checked_set_intersection

Same as set_intersection but enforces the use of a checked iterator as output iterator. **checked_set_intersection** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
   OutputIterator checked_set_intersection(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPred
icate>
   OutputIterator checked_set_intersection(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result,
      BinaryPredicate _Comp
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

_Last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

_First2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

_Last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

_Result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the intersection of the two source ranges.

_Comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range representing the intersection of the two source ranges.

Remarks

See set_intersection for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

checked_set_symmetric_difference

Same as set_symmetric_difference but enforces the use of a checked iterator as output iterator. **checked_set_symmetric_difference** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
   OutputIterator checked_set_symmetric_difference(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPred
icate>
   OutputIterator checked_set_symmetric_difference(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result,
      BinaryPredicate _Comp
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

_Last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

_First2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

_Last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

_Result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the symmetric difference of the two source ranges.

_Comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range representing the symmetric difference of the two source ranges.

Remarks

See set_symmetric_difference for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

checked_set_union

Same as set_union but enforces the use of a checked iterator as output iterator. **checked_set_union** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
   OutputIterator checked_set_union(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPred
icate>
   OutputIterator checked_set_union(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result,
      BinaryPredicate _Comp
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

_Last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

_First2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

_Last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

_Result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the union of the two source ranges.

_Comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range representing the union of the two source ranges.

Remarks

See set_union for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

checked_unique_copy

Same as unique_copy but enforces the use of a checked iterator as output iterator. **checked_unique_copy** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator, class OutputIterator>
   OutputIterator checked_unique_copy(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result
   );
template<class InputIterator, class OutputIterator, class BinaryPredicate>
   OutputIterator checked_unique_copy(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      BinaryPredicate _Comp,
   );
```

Parameters

_First

A forward iterator addressing the position of the first element in the source range to be copied.

_Last

A forward iterator addressing the position one past the final element in the source range to be copied.

_Result

An output iterator addressing the position of the first element in the destination range that is receiving the copy with consecutive duplicates removed.

_Comp

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

An output iterator addressing the position one past the final element in the destination range that is receiving the copy with consecutive duplicates removed.

Remarks

See unique_copy for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

copy

Assigns the values of elements from a source range to a destination range, iterating through the source sequence of elements and assigning them new positions in a forward direction.

```
template<class InputIterator, class OutputIterator>
   OutputIterator copy(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _DestBeg
   );
```

Parameters

_First

An input iterator addressing the position of the first element in the source range.

_Last

An input iterator addressing the position that is one past the final element in the source range.

_DestBeg

An output iterator addressing the position of the first element in the destination range.

Return Value

An output iterator addressing the position that is one past the final element in the destination range, that is, the iterator addresses *_Result* + (*_Last* – *_First*).

Remarks

The source range must be valid and there must be sufficient space at the destination to hold all the elements being copied.

Because the algorithm copies the source elements in order beginning with the first element, the destination range can overlap with the source range provided the *_Last* position of the source range is not contained in the destination range. **copy** can be used to shift elements to the left but not the right, unless there is no overlap between the source and destination ranges. To shift to the right any number of positions, use the copy_backward algorithm.

The **copy** algorithm only modifies values pointed to by the iterators, assigning new values to elements in the destination range. It cannot be used to create new elements and cannot insert elements into an empty container directly.

copy has two related forms:

- checked_copy
- unchecked_copy

For information on how these functions behave, see Checked Iterators.

Example

```
// alg_copy.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
   using namespace std;
   vector <int> v1, v2;
   vector <int>::iterator Iter1, Iter2;

   int i;
   for ( i = 0 ; i <= 5 ; i++ )
      v1.push_back( 10 * i );
```

```
    int ii;
    for ( ii = 0 ; ii <= 10 ; ii++ )
       v2.push_back( 3 * ii );

    cout << "v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
       cout << *Iter2 << " ";
    cout << ")" << endl;

    // To copy the first 3 elements of v1 into the middle of v2
    copy( v1.begin( ), v1.begin( ) + 3, v2.begin( ) + 4 );

    cout << "v2 with v1 insert = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
       cout << *Iter2 << " ";
    cout << ")" << endl;

    // To shift the elements inserted into v2 two positions
    // to the left
    copy( v2.begin( )+4, v2.begin( ) + 7, v2.begin( ) + 2 );

    cout << "v2 with shifted insert = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
       cout << *Iter2 << " ";
    cout << ")" << endl;
  }
```

For another sample showing how to use copy, see accumulate, copy, and vector::push_back.

Output

```
v1 = ( 0 10 20 30 40 50 )
v2 = ( 0 3 6 9 12 15 18 21 24 27 30 )
v2 with v1 insert = ( 0 3 6 9 0 10 20 21 24 27 30 )
v2 with shifted insert = ( 0 3 0 10 20 10 20 21 24 27 30 )
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

copy_backward

Assigns the values of elements from a source range to a destination range, iterating through the source sequence of elements and assigning them new positions in a backward direction.

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
   BidirectionalIterator2 copy_backward(
      BidirectionalIterator1 _First,
      BidirectionalIterator1 _Last,
      BidirectionalIterator2 _DestEnd
   );
```

Parameters

_First

A bidirectional iterator addressing the position of the first element in the source range.

_Last

A bidirectional iterator addressing the position that is one past the final element in the source range.

_DestEnd

A bidirectional iterator addressing the position of the one past the final element in the destination range.

Return Value

An output iterator addressing the position that is one past the final element in the destination range, that is, the iterator addresses _ *DestEnd* – (_*Last* – _*First*).

Remarks

The source range must be valid and there must be sufficient space at the destination to hold all the elements being copied.

The **copy_backward** algorithm imposes more stringent requirements than that the copy algorithm. Both its input and output iterators must be bidirectional.

The **copy_backward** algorithm is the only Standard Template Library algorithm designating the output range with an iterator pointing to the end of the destination range.

Because the algorithm copies the source elements in order beginning with the last element, the destination range can overlap with the source range provided the _*First* position of the source range is not contained in the destination range. **copy_backward** can be used to shift elements to the right but not the left, unless there is no overlap between the source and destination ranges. To shift to the left any number of positions, use the copy algorithm.

The **copy_backward** algorithm only modifies values pointed to by the iterators, assigning new values to elements in the destination range. It cannot be used to create new elements and cannot insert elements into an empty container directly.

copy_backward has two related forms:

- checked_copy_backward
- unchecked_copy_backward

For information on how these functions behave, see Checked Iterators.

Example

```
// alg_copy_bkwd.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
   using namespace std;
   vector <int> v1, v2;
```

```
    vector <int>::iterator Iter1, Iter2;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
       v1.push_back( 10 * i );

    int ii;
    for ( ii = 0 ; ii <= 10 ; ii++ )
       v2.push_back( 3 * ii );

    cout << "v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
       cout << *Iter2 << " ";
    cout << ")" << endl;

    // To copy_backward the first 3 elements of v1 into the middle of v2
    copy_backward( v1.begin( ), v1.begin( ) + 3, v2.begin( ) + 7 );

    cout << "v2 with v1 insert = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
       cout << *Iter2 << " ";
    cout << ")" << endl;

    // To shift the elements inserted into v2 two positions
    // to the right
    copy_backward( v2.begin( )+4, v2.begin( ) + 7, v2.begin( ) + 9 );

    cout << "v2 with shifted insert = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
       cout << *Iter2 << " ";
    cout << ")" << endl;
 }
```

Output

```
v1 = ( 0 10 20 30 40 50 )
v2 = ( 0 3 6 9 12 15 18 21 24 27 30 )
v2 with v1 insert = ( 0 3 6 9 0 10 20 21 24 27 30 )
v2 with shifted insert = ( 0 3 6 9 0 10 0 10 20 27 30 )
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

count

Returns the number of elements in a range whose values match a specified value.

```
template<class InputIterator, class Type>
   typename iterator_traits<InputIterator>::difference_type count(
      InputIterator _First,
      InputIterator _Last,
      const Type& _Val
   );
```

Parameters

_First

An input iterator addressing the position of the first element in the range to be traversed.

_Last

An input iterator addressing the position one past the final element in the range to be traversed.

_Val

The value of the elements to be counted.

Return Value

The difference type of the **InputIterator** that counts the number of elements in the range [*_First*, *_Last*) that have value *_Val*.

Remarks

The **operator==** used to determine the match between an element and the specified value must impose an equivalence relation between its operands.

This algorithm is generalized to count elements that satisfy any predicate with the template function count_if.

Example

```
// alg_count.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter;

    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(10);
    v1.push_back(40);
    v1.push_back(10);

    cout << "v1 = ( " ;
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    vector<int>::iterator::difference_type result;
    result = count(v1.begin(), v1.end(), 10);
    cout << "The number of 10s in v2 is: " << result << "." << endl;
}
```

Output

```
v1 = ( 10 20 10 40 10 )
The number of 10s in v2 is: 3.
```

Requirements

Header: <algorithm>

See Also

Reference

count (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

count_if

Returns the number of elements in a range whose values satisfy a specified condition.

```
template<class InputIterator, class Predicate>
   typename iterator_traits<InputIterator>::difference_type count_if(
      InputIterator _First,
      InputIterator _Last,
      Predicate _Pred
   );
```

Parameters

_First

An input iterator addressing the position of the first element in the range to be searched.

_Last

An input iterator addressing the position one past the final element in the range to be searched.

_Pred

User-defined predicate function object that defines the condition to be satisfied if an element is to be counted. A predicate takes single argument and returns **true** or **false**.

Return Value

The number of elements that satisfy the condition specified by the predicate.

Remarks

This template function is a generalization of the algorithm count, replacing the predicate "equals a specific value" with any predicate.

Example

```
// alg_count_if.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

bool greater10(int value)
{
    return value >10;
}

int main()
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter;

    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(10);
    v1.push_back(40);
    v1.push_back(10);

    cout << "v1 = ( ";
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    vector<int>::iterator::difference_type result1;
    result1 = count_if(v1.begin(), v1.end(), greater10);
```

```
    cout << "The number of elements in v1 greater than 10 is: "
         << result1 << "." << endl;
}
```

Output

```
v1 = ( 10 20 10 40 10 )
The number of elements in v1 greater than 10 is: 2.
```

Requirements

Header: <algorithm>

See Also

Reference

count_if (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

equal

Compares two ranges element by element either for equality or equivalence in a sense specified by a binary predicate.

```
template<class InputIterator1, class InputIterator2>
   bool equal(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2
      );
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
   bool equal(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      BinaryPredicate _Comp
      );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first range to be tested.

_Last1

An input iterator addressing the position one past the final element in the first range to be tested.

_First2

An input iterator addressing the position of the first element in the second range to be tested.

_Comp

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

true if and only if the ranges are identical or equivalent under the binary predicate when compared element by element; otherwise, **false**.

Remarks

The range to be searched must be valid; all pointers must be dereferenceable and the last position is reachable from the first by incrementation.

The time complexity of the algorithm is linear in the number of elements contained in the range.

The **operator==** used to determine the equality between elements must impose an equivalence relation between its operands.

Example

```
// alg_equal.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

// Return whether second element is twice the first
bool twice ( int elem1, int elem2 )
{
   return elem1 * 2 == elem2;
}

int main( )
{
   using namespace std;
```

```
vector <int> v1, v2, v3;
vector <int>::iterator Iter1, Iter2, Iter3;

int i;
for ( i = 0 ; i <= 5 ; i++ )
{
   v1.push_back( 5 * i );
}

int ii;
for ( ii = 0 ; ii <= 5 ; ii++ )
{
   v2.push_back( 5 * ii );
}

int iii;
for ( iii = 0 ; iii <= 5 ; iii++ )
{
   v3.push_back( 10 * iii );
}

cout << "v1 = ( " ;
for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
   cout << *Iter1 << " ";
cout << ")" << endl;

cout << "v2 = ( " ;
for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
   cout << *Iter2 << " ";
cout << ")" << endl;

cout << "v3 = ( " ;
for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
   cout << *Iter3 << " ";
cout << ")" << endl;

// Testing v1 and v2 for equality under identity
bool b;
b = equal( v1.begin( ), v1.end( ), v2.begin( ) );

if ( b )
   cout << "The vectors v1 and v2 are equal under equality."
        << endl;
else
   cout << "The vectors v1 and v2 are not equal under equality."
        << endl;

// Testing v1 and v3 for equality under identity
bool c;
c = equal( v1.begin( ), v1.end( ), v3.begin( ) );

if ( c )
   cout << "The vectors v1 and v3 are equal under equality."
        << endl;
else
   cout << "The vectors v1 and v3 are not equal under equality."
        << endl;

// Testing v1 and v3 for equality under twice
bool d;
d = equal( v1.begin( ), v1.end( ), v3.begin( ), twice );

if ( d )
   cout << "The vectors v1 and v3 are equal under twice."
        << endl;
else
   cout << "The vectors v1 and v3 are not equal under twice."
        << endl;
```

```
}
```

Output

```
v1 = ( 0 5 10 15 20 25 )
v2 = ( 0 5 10 15 20 25 )
v3 = ( 0 10 20 30 40 50 )
The vectors v1 and v2 are equal under equality.
The vectors v1 and v3 are not equal under equality.
The vectors v1 and v3 are equal under twice.
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

equal_range

Finds a pair of positions in an ordered range, the first less than or equivalent to the position of a specified element and the second greater than the element's position, where the sense of equivalence or ordering used to establish the positions in the sequence may be specified by a binary predicate.

```
template<class ForwardIterator, class Type>
   pair<ForwardIterator, ForwardIterator> equal_range(
      ForwardIterator _First,
      ForwardIterator _Last,
      const Type& _Val
   );
template<class ForwardIterator, class Type, class Pr>
   pair<ForwardIterator, ForwardIterator> equal_range(
      ForwardIterator _First,
      ForwardIterator _Last,
      const Type& _Val,
      BinaryPredicate _Comp
   );
```

Parameters

_First

A forward iterator addressing the position of the first element in the range to be searched.

_Last

A forward iterator addressing the position one past the final element in the range to be searched.

_Val

The value in the ordered range that needs to be equivalent to the value of the element addressed by the first component of the pair returned and that needs to be less than the value of the element addressed by the second component of that pair returns.

_Comp

User-defined predicate function object that is **true** when the left-hand argument is less than the right-hand argument. The user-defined predicate function should return **false** when its arguments are equivalent.

Return Value

A pair of forward iterators addressing two positions in an ordered range in which the first component of the pair refers to the position where an element is or would be if it had a value that is less than or equivalent to a specified value and the second component of the pair refers to the first position where an element has a value that is greater than the value specified, where the sense of equivalence or ordering may be specified by a binary predicate.

Alternately, the pair of forward iterators may be described as specify a subrange, contained within the range searched, in which all of the elements are equivalent to the specified value in the sense defined by the binary predicate used.

Remarks

The first component of the pair of the algorithm returns lower_bound, and the second component returns upper_bound.

The subrange defined by the pair of iterators returned by the **equal_range** algorithm contains the equivalence class, in the standard set-theoretic sense, of the element whose value is specified as a parameter.

The sorted source range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position must be reachable from the first by incrementation.

The sorted range must each be arranged as a precondition to the application of the **equal_range** algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The range is not modified by the algorithm **merge**.

The value types of the forward iterators need be less-than comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other.

This results in an ordering between the nonequivalent elements

The complexity of the algorithm is logarithmic for random-access iterators and linear otherwise, with the number of steps proportional to (*_Last1* – *_First1*).

Example

```
// alg_equal_range.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      //For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
   if ( elem1 < 0 )
      elem1 = - elem1;
   if ( elem2 < 0 )
      elem2 = - elem2;
   return elem1 < elem2;
}

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1;
   pair < vector <int> :: iterator , vector <int> :: iterator > Result1, Result2, Result3;

   // Constructing vectors v1a & v1b with default less than ordering
   int i;
   for ( i = -1 ; i <= 4 ; i++ )
   {
      v1.push_back(  i );
   }

   int ii;
   for ( ii =-3 ; ii <= 0 ; ii++ )
   {
      v1.push_back( ii );
   }

   sort ( v1.begin ( ) , v1.end ( ) );
   cout << "Original vector v1 with range sorted by the\n "
        <<  "binary predicate less than is  v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // Constructing vectors v2 with range sorted by greater
   vector <int> v2 ( v1 );
   vector <int>::iterator Iter2;
   sort ( v2.begin ( ) , v2.end ( ) , greater<int> ( ) );

   cout << "Original vector v2 with range sorted by the\n "
        <<  "binary predicate greater is    v2 = ( " ;
   for ( Iter2 = v2.begin ( ) ; Iter2 != v2.end ( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << "))." << endl;

   // Constructing vectors v3 with range sorted by mod_lesser
   vector <int> v3 ( v1 );
   vector <int>::iterator Iter3;
   sort ( v3.begin ( ) , v3.end ( ) , mod_lesser );

   cout << "Original vector v3 with range sorted by the\n "
        << "binary predicate mod_lesser is v3 = ( " ;
```

```
    for ( Iter3 = v3.begin ( ) ; Iter3 != v3.end ( ) ; Iter3++ )
       cout << *Iter3 << " ";
    cout << "))." << endl << endl;

    // equal_range of 3 in v1 with default binary predicate less <int> ( )
    Result1 = equal_range ( v1.begin ( ) , v1.end ( ) , 3 );
    cout << "The lower_bound in v1 for the element with a value of 3 is: "
         << *Result1.first << "." << endl;
    cout << "The upper_bound in v1 for the element with a value of 3 is: "
         << *Result1.second << "." << endl;
    cout << "The equivalence class for the element with a value of 3 in"
         << "\n v1 includes the elements: ( ";
    for ( Iter1 = Result1.first ; Iter1 != Result1.second ; Iter1++ )
       cout << *Iter1 << " ";
    cout << "))." << endl << endl;

    // equal_range of 3 in v2 with the binary predicate greater <int> ( )
    Result2 = equal_range ( v2.begin ( ) , v2.end ( ) , 3 , greater <int> ( ) );
    cout << "The lower_bound in v2 for the element with a value of 3 is: "
         << *Result2.first << "." << endl;
    cout << "The upper_bound in v2 for the element with a value of 3 is: "
         << *Result2.second << "." << endl;
    cout << "The equivalence class for the element with a value of 3 in"
         << "\n v2 includes the elements: ( ";
    for ( Iter2 = Result2.first ; Iter2 != Result2.second ; Iter2++ )
       cout << *Iter2 << " ";
    cout << "))." << endl << endl;

    // equal_range of 3 in v3 with the binary predicate mod_lesser
    Result3 = equal_range ( v3.begin ( ) , v3.end ( ) , 3 ,mod_lesser );
    cout << "The lower_bound in v3 for the element with a value of 3 is: "
         << *Result3.first << "." << endl;
    cout << "The upper_bound in v3 for the element with a value of 3 is: "
         << *Result3.second << "." << endl;
    cout << "The equivalence class for the element with a value of 3 in"
         << "\n v3 includes the elements: ( ";
    for ( Iter3 = Result3.first ; Iter3 != Result3.second ; Iter3++ )
       cout << *Iter3 << " ";
    cout << "))." << endl << endl;
  }
```

Output

```
Original vector v1 with range sorted by the
 binary predicate less than is  v1 = ( -3 -2 -1 -1 0 0 1 2 3 4 ).
Original vector v2 with range sorted by the
 binary predicate greater is    v2 = ( 4 3 2 1 0 0 -1 -1 -2 -3 ).
Original vector v3 with range sorted by the
 binary predicate mod_lesser is v3 = ( 0 0 -1 -1 1 -2 2 -3 3 4 ).

The lower_bound in v1 for the element with a value of 3 is: 3.
The upper_bound in v1 for the element with a value of 3 is: 4.
The equivalence class for the element with a value of 3 in
 v1 includes the elements: ( 3 ).

The lower_bound in v2 for the element with a value of 3 is: 3.
The upper_bound in v2 for the element with a value of 3 is: 2.
The equivalence class for the element with a value of 3 in
 v2 includes the elements: ( 3 ).

The lower_bound in v3 for the element with a value of 3 is: -3.
The upper_bound in v3 for the element with a value of 3 is: 4.
The equivalence class for the element with a value of 3 in
 v3 includes the elements: ( -3 3 ).
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

fill

Assigns the same new value to every element in a specified range.

```
template<class ForwardIterator, class Type>
   void fill(
      ForwardIterator _First,
      ForwardIterator _Last,
      const Type& _Val
   );
```

Parameters

_First

A forward iterator addressing the position of the first element in the range to be traversed.

_Last

A forward iterator addressing the position one past the final element in the range to be traversed.

_Val

The value to be assigned to elements in the range [*_First*, *_Last*).

Remarks

The destination range must be valid; all pointers must be dereferenceable, and the last position is reachable from the first by incrementation. The complexity is linear with the size of the range.

Example

```
// alg_fill.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1;

   int i;
   for ( i = 0 ; i <= 9 ; i++ )
   {
      v1.push_back( 5 * i );
   }

   cout << "Vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // Fill the last 5 positions with a value of 2
   fill( v1.begin( ) + 5, v1.end( ), 2 );

   cout << "Modified v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;
}
```

Output

```
Vector v1 = ( 0 5 10 15 20 25 30 35 40 45 )
```

```
Modified v1 = ( 0 5 10 15 20 2 2 2 2 2 )
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

fill_n

Assigns a new value to a specified number of elements in a range beginning with a particular element.

```
template<class OutputIterator, class Size, class Type>
   void fill_n(
      OutputIterator _First,
      Size _Count,
      const Type& _Val
   );
```

Parameters

_First

An output iterator addressing the position of the first element in the range to be assigned the value *_Val*.

_Count

A signed or unsigned integer type specifying the number of elements to be assigned the value.

_Val

The value to be assigned to elements in the range [*_First*, *_First* + *_Count*).

Remarks

The destination range must be valid; all pointers must be dereferenceable, and the last position is reachable from the first by incrementation. The complexity is linear with the size of the range.

fill_n has two related forms:

- checked_fill_n
- unchecked_fill_n

For information on how these functions behave, see Checked Iterators.

Example

```
// alg_fill_n.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1;

   int i;
   for ( i = 0 ; i <= 9 ; i++ )
      v1.push_back( 5 * i );

   cout << "Vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // Fill the last 5 positions with a value of 2
   fill_n( v1.begin( ) + 5, 5, 2 );

   cout << "Modified v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;
```

```
}
```

Output

```
Vector v1 = ( 0 5 10 15 20 25 30 35 40 45 )
Modified v1 = ( 0 5 10 15 20 2 2 2 2 2 )
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

find (<algorithm>)

Locates the position of the first occurrence of an element in a range that has a specified value.

```
template<class InputIterator, class Type>
   InputIterator find(
      InputIterator _First,
      InputIterator _Last,
      const Type& _Val
   );
```

Parameters

_First

An input iterator addressing the position of the first element in the range to be searched for the specified value.

_Last

An input iterator addressing the position one past the final element in the range to be searched for the specified value.

_Val

The value to be searched for.

Return Value

An input iterator addressing the first occurrence of the specified value in the range being searched. If no such value exists in the range, the iterator returned addresses the last position of the range, one past the final element.

Remarks

The **operator==** used to determine the match between an element and the specified value must impose an equivalence relation between its operands.

Example

```
// alg_find.cpp
// compile with: /EHsc
#include <list>
#include <algorithm>
#include <iostream>

int main() {
   using namespace std;

   list <int> L;
   list <int>::iterator Iter;
   list <int>::iterator result;

   L.push_back( 40 );
   L.push_back( 20 );
   L.push_back( 10 );
   L.push_back( 30 );
   L.push_back( 10 );

   cout << "L = ( " ;
   for ( Iter = L.begin( ) ; Iter != L.end( ) ; Iter++ )
      cout << *Iter << " ";
   cout << ")" << endl;

   result = find( L.begin( ), L.end( ), 10 );
   if  ( result == L.end( ) )
      cout << "There is no 10 in list L.";
   else {
      cout << "There is a 10 in list L";
      if ( ++result != L.end() )
```

```
            cout << " and it is followed by a " << *result << ".";
    }
    cout << endl;
}
```

Output

```
L = ( 40 20 10 30 10 )
There is a 10 in list L and it is followed by a 30.
```

Requirements

Header: <algorithm>

See Also

Reference

find (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

find_end

Looks in a range for the last subsequence that is identical to a specified sequence or that is equivalent in a sense specified by a binary predicate.

```
template<class ForwardIterator1, class ForwardIterator2>
   ForwardIterator1 find_end(
      ForwardIterator1 _First1,
      ForwardIterator1 _Last1,
      ForwardIterator2 _First2,
      ForwardIterator2 _Last2
   );
template<class ForwardIterator1, class ForwardIterator2, class Pr>
   ForwardIterator1 find_end(
      ForwardIterator1 _First1,
      ForwardIterator1 _Last1,
      ForwardIterator2 _First2,
      ForwardIterator2 _Last2,
      BinaryPredicate _Comp
   );
```

Parameters

_First1

A forward iterator addressing the position of the first element in the range to be searched.

_Last1

A forward iterator addressing the position one past the final element in the range to be searched.

_First2

A forward iterator addressing the position of the first element in the range to be searched.

_Last2

A forward iterator addressing the position one past the final element in the range to be searched.

_Comp

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

A forward iterator addressing the position of the first element of the last subsequence that matches the specified sequence or that is equivalent in a sense specified by a binary predicate.

Remarks

The **operator==** used to determine the match between an element and the specified value must impose an equivalence relation between its operands.

The ranges referenced must be valid; all pointers must be dereferenceable and, within each sequence, the last position is reachable from the first by incrementation.

Example

```
// alg_find_end.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>

// Return whether second element is twice the first
bool twice ( int elem1, int elem2 )
```

```
{
   return 2 * elem1 == elem2;
}

int main( )
{
   using namespace std;
   vector <int> v1, v2;
   list <int> L1;
   vector <int>::iterator Iter1, Iter2;
   list <int>::iterator L1_Iter, L1_inIter;

   int i;
   for ( i = 0 ; i <= 5 ; i++ )
   {
      v1.push_back( 5 * i );
   }
   for ( i = 0 ; i <= 5 ; i++ )
   {
      v1.push_back( 5 * i );
   }

   int ii;
   for ( ii = 1 ; ii <= 4 ; ii++ )
   {
      L1.push_back( 5 * ii );
   }

   int iii;
   for ( iii = 2 ; iii <= 4 ; iii++ )
   {
      v2.push_back( 10 * iii );
   }

   cout << "Vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   cout << "List L1 = ( " ;
   for ( L1_Iter = L1.begin( ) ; L1_Iter!= L1.end( ) ; L1_Iter++ )
      cout << *L1_Iter << " ";
   cout << ")" << endl;

   cout << "Vector v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
      cout << ")" << endl;

   // Searching v1 for a match to L1 under identity
   vector <int>::iterator result1;
   result1 = find_end ( v1.begin( ), v1.end( ), L1.begin( ), L1.end( ) );

   if ( result1 == v1.end( ) )
      cout << "There is no match of L1 in v1."
           << endl;
   else
      cout << "There is a match of L1 in v1 that begins at "
           << "position "<< result1 - v1.begin( ) << "." << endl;

   // Searching v1 for a match to L1 under the binary predicate twice
   vector <int>::iterator result2;
   result2 = find_end ( v1.begin( ), v1.end( ), v2.begin( ), v2.end( ), twice );

   if ( result2 == v1.end( ) )
      cout << "There is no match of L1 in v1."
           << endl;
   else
```

```
        cout << "There is a sequence of elements in v1 that "
             << "are equivalent to those\n in v2 under the binary "
             << "predicate twice and that begins at position "
             << result2 - v1.begin( ) << "." << endl;
  }
```

Output

```
Vector v1 = ( 0 5 10 15 20 25 0 5 10 15 20 25 )
List L1 = ( 5 10 15 20 )
Vector v2 = ( 20 30 40 )
There is a match of L1 in v1 that begins at position 7.
There is a sequence of elements in v1 that are equivalent to those
 in v2 under the binary predicate twice and that begins at position 8.
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

find_first_of

Searches for the first occurrence of any of several values within a target range or for the first occurrence of any of several elements that are equivalent in a sense specified by a binary predicate to a specified set of the elements.

```
template<class ForwardIterator1, class ForwardIterator2>
   ForwardIterator1 find_first_of(
      ForwardIterator1 _First1,
      ForwardIterator1 _Last1,
      ForwardIterator2 _First2,
      ForwardIterator2 _Last2
   );
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
   ForwardIterator1 find_first_of(
      ForwardIterator1 _First1,
      ForwardIterator1 _Last1,
      ForwardIterator2 _First2,
      ForwardIterator2 _Last2,
      BinaryPredicate _Comp
   );
```

Parameters

_First1

A forward iterator addressing the position of the first element in the range to be searched.

_Last1

A forward iterator addressing the position one past the final element in the range to be searched.

_First2

A forward iterator addressing the position of the first element in the range to be matched.

_Last2

A forward iterator addressing the position one past the final element in the range to be matched.

_Comp

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

A forward iterator addressing the position of the first element of the first subsequence that matches the specified sequence or that is equivalent in a sense specified by a binary predicate.

Remarks

The **operator==** used to determine the match between an element and the specified value must impose an equivalence relation between its operands.

The ranges referenced must be valid; all pointers must be dereferenceable and, within each sequence, the last position is reachable from the first by incrementation.

Example

```
// alg_find_first_of.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>

// Return whether second element is twice the first
bool twice ( int elem1, int elem2 )
```

```
{
   return 2 * elem1 == elem2;
}

int main( )
{
   using namespace std;
   vector <int> v1, v2;
   list <int> L1;
   vector <int>::iterator Iter1, Iter2;
   list <int>::iterator L1_Iter, L1_inIter;

   int i;
   for ( i = 0 ; i <= 5 ; i++ )
   {
      v1.push_back( 5 * i );
   }
   for ( i = 0 ; i <= 5 ; i++ )
   {
      v1.push_back( 5 * i );
   }

   int ii;
   for ( ii = 3 ; ii <= 4 ; ii++ )
   {
      L1.push_back( 5 * ii );
   }

   int iii;
   for ( iii = 2 ; iii <= 4 ; iii++ )
   {
      v2.push_back( 10 * iii );
   }

   cout << "Vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   cout << "List L1 = ( " ;
   for ( L1_Iter = L1.begin( ) ; L1_Iter!= L1.end( ) ; L1_Iter++ )
      cout << *L1_Iter << " ";
   cout << ")" << endl;

   cout << "Vector v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
      cout << ")" << endl;

   // Searching v1 for first match to L1 under identity
   vector <int>::iterator result1;
   result1 = find_first_of ( v1.begin( ), v1.end( ), L1.begin( ), L1.end( ) );

   if ( result1 == v1.end( ) )
      cout << "There is no match of L1 in v1."
           << endl;
   else
      cout << "There is at least one match of L1 in v1"
           << "\n and the first one begins at "
           << "position "<< result1 - v1.begin( ) << "." << endl;

   // Searching v1 for a match to L1 under the binary predicate twice
   vector <int>::iterator result2;
   result2 = find_first_of ( v1.begin( ), v1.end( ), v2.begin( ), v2.end( ), twice );

   if ( result2 == v1.end( ) )
      cout << "There is no match of L1 in v1."
           << endl;
```

```
    else
       cout << "There is a sequence of elements in v1 that "
            << "are equivalent\n to those in v2 under the binary "
            << "predicate twice\n and the first one begins at position "
            << result2 - v1.begin( ) << "." << endl;
  }
```

Output

```
Vector v1 = ( 0 5 10 15 20 25 0 5 10 15 20 25 )
List L1 = ( 15 20 )
Vector v2 = ( 20 30 40 )
There is at least one match of L1 in v1
 and the first one begins at position 3.
There is a sequence of elements in v1 that are equivalent
 to those in v2 under the binary predicate twice
 and the first one begins at position 2.
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

find_if

Locates the position of the first occurrence of an element in a range that satisfies a specified condition.

```
template<class InputIterator, class Predicate>
   InputIterator find_if(
      InputIterator _First,
      InputIterator _Last,
      Predicate _Pred
   );
```

Parameters

_First

An input iterator addressing the position of the first element in the range to be searched.

_Last

An input iterator addressing the position one past the final element in the range to be searched.

_Pred

User-defined predicate function object that defines the condition to be satisfied by the element being searched for. A predicate takes single argument and returns **true** or **false**.

Return Value

An input iterator that addresses the first element in the range that satisfies the condition specified by the predicate.

Remarks

This template function is a generalization of the algorithm find, replacing the predicate "equals a specific value" with any predicate.

Example

```
// alg_find_if.cpp
// compile with: /EHsc
#include <list>
#include <algorithm>
#include <iostream>

bool greater10 ( int value )
{
   return value >10;
}

int main( )
{
   using namespace std;

   list <int> L;
   list <int>::iterator Iter;
   list <int>::iterator result;

   L.push_back( 5 );
   L.push_back( 10 );
   L.push_back( 15 );
   L.push_back( 20 );
   L.push_back( 10 );

   cout << "L = ( " ;
   for ( Iter = L.begin( ) ; Iter != L.end( ) ; Iter++ )
      cout << *Iter << " ";
   cout << ")" << endl;
```

```
    result = find_if( L.begin( ), L.end( ), &greater10 );
    if ( result == L.end( ) )
       cout << "There is no element greater than 10 in list L."
            << endl;
    else
    {
       result++;
       cout << "There is an element greater than 10 in list L,"
            << "\n and it is followed by a "
            <<  *(result) << "." << endl;
    }
  }
```

Output

```
L = ( 5 10 15 20 10 )
There is an element greater than 10 in list L,
 and it is followed by a 20.
```

Requirements

Header: <algorithm>

See Also

Reference

find_if (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

for_each

Applies a specified function object to each element in a forward order within a range and returns the function object.

```
template<class InputIterator, class Function>
   Function for_each(
      InputIterator _First,
      InputIterator _Last,
      Function _Func
      );
```

Parameters

_First

An input iterator addressing the position of the first element in the range to be operated on.

_Last

An input iterator addressing the position one past the final element in the range operated on.

_Func

User-defined function object that is applied to each element in the range.

Return Value

A copy of the function object after it has been applied to all of the elements in the range.

Remarks

The algorithm **for_each** is very flexible, allowing the modification of each element within a range in different, user-specified ways. Templatized functions may be reused in a modified form by passing different parameters. User-defined functions may accumulate information within an internal state that the algorithm may return after processing all of the elements in the range.

The range referenced must be valid; all pointers must be dereferenceable and, within the sequence, the last position must be reachable from the first by incrementation.

The complexity is linear with at most (*_Last* – *_First*) comparisons.

Example

```
// alg_for_each.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

// The function object multiplies an element by a Factor
template <class Type>
class MultValue
{
private:
   Type Factor;   // The value to multiply by
public:
   // Constructor initializes the value to multiply by
   MultValue ( const Type& _Val ) : Factor ( _Val ) {
   }

   // The function call for the element to be multiplied
   void operator ( ) ( Type& elem ) const
   {
      elem *= Factor;
   }
};

// The function object to determine the average
class Average
```

```
{
private:
   long num;      // The number of elements
   long sum;      // The sum of the elements
public:
   // Constructor initializes the value to multiply by
   Average ( ) : num ( 0 ) , sum ( 0 )
   {
   }

   // The function call to process the next elment
   void operator ( ) ( int elem ) \
   {
      num++;      // Increment the element count
      sum += elem;   // Add the value to the partial sum
   }

   // return Average
   operator double ( )
   {
      return  static_cast <double> (sum) /
      static_cast <double> (num);
   }
};

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1;

   // Constructing vector v1
   int i;
   for ( i = -4 ; i <= 2 ; i++ )
   {
      v1.push_back(  i );
   }

   cout << "Original vector  v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;

   // Using for_each to multiply each element by a Factor
   for_each ( v1.begin ( ) , v1.end ( ) , MultValue<int> ( -2 ) );

   cout << "Multiplying the elements of the vector v1\n "
        <<  "by the factor -2 gives:\n v1mod1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;

   // The function object is templatized and so can be
   // used again on the elements with a different Factor
   for_each (v1.begin ( ) , v1.end ( ) , MultValue<int> (5 ) );

   cout << "Multiplying the elements of the vector v1mod\n "
        <<  "by the factor 5 gives:\n v1mod2 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;

   // The local state of a function object can accumulate
   // information about a sequence of actions that the
   // return value can make available, here the Average
   double avemod2 = for_each ( v1.begin ( ) , v1.end ( ) ,
      Average ( ) );
   cout << "The average of the elements of v1 is:\n Average ( v1mod2 ) = "
```

```
        << avemod2 << "." << endl;
  }
```

Output

```
Original vector  v1 = ( -4 -3 -2 -1 0 1 2 ).
Multiplying the elements of the vector v1
 by the factor -2 gives:
 v1mod1 = ( 8 6 4 2 0 -2 -4 ).
Multiplying the elements of the vector v1mod
 by the factor 5 gives:
 v1mod2 = ( 40 30 20 10 0 -10 -20 ).
The average of the elements of v1 is:
 Average ( v1mod2 ) = 10.
```

Requirements

Header: <algorithm>

See Also

Reference

for_each (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

generate

Assigns the values generated by a function object to each element in a range.

```
template<class ForwardIterator, class Generator>
   void generate(
      ForwardIterator _First,
      ForwardIterator _Last,
      Generator _Gen
   );
```

Parameters

_First

A forward iterator addressing the position of the first element in the range to which values are to be assigned.

_Last

A forward iterator addressing the position one past the final element in the range to which values are to be assigned.

_Gen

A function object that is called with no arguments that is used to generate the values to be assigned to each of the elements in the range.

Remarks

The function object is invoked for each element in the range and does not need to return the same value each time it is called. It may, for example, read from a file or refer to and modify a local state. The generator's result type must be convertible to the value type of the forward iterators for the range.

The range referenced must be valid; all pointers must be dereferenceable and, within the sequence, the last position must be reachable from the first by incrementation.

The complexity is linear, with exactly (*_Last* – *_First*) calls to the generator being required.

Example

```
// alg_generate.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>
#include <ostream>

int main( )
{
   using namespace std;

   // Assigning random values to vector integer elements
   vector <int> v1 ( 5 );
   vector <int>::iterator Iter1;
   deque <int> deq1 ( 5 );
   deque <int>::iterator d1_Iter;

   generate ( v1.begin ( ), v1.end ( ) , rand );

   cout << "Vector v1 is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;

   // Assigning random values to deque integer elements
   generate ( deq1.begin ( ), deq1.end ( ) , rand );
```

```
    cout << "Deque deq1 is ( " ;
    for ( d1_Iter = deq1.begin( ) ; d1_Iter != deq1.end( ) ; d1_Iter++ )
       cout << *d1_Iter << " ";
    cout << "))." << endl;
  }
```

Output

```
Vector v1 is ( 41 18467 6334 26500 19169 ).
Deque deq1 is ( 15724 11478 29358 26962 24464 ).
```

Requirements

Header: <algorithm>

See Also

Reference

generate (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

generate_n

Assigns the values generated by a function object to a specified number of element in a range and returns to the position one past the last assigned value.

```
template<class OutputIterator, class Size, class Generator>
   void generate_n(
      OutputIterator _First,
      Size _Count,
      Generator _Gen
   );
```

Parameters

_First

An output iterator addressing the position of first element in the range to which values are to be assigned.

_Count

A signed or unsigned integer type specifying the number of elements to be assigned a value by the generator function.

_Gen

A function object that is called with no arguments that is used to generate the values to be assigned to each of the elements in the range.

Remarks

The function object is invoked for each element in the range and does not need to return the same value each time it is called. It may, for example, read from a file or refer to and modify a local state. The generator's result type must be convertible to the value type of the forward iterators for the range.

The range referenced must be valid; all pointers must be dereferenceable and, within the sequence, the last position must be reachable from the first by incrementation.

The complexity is linear, with exactly *_Count* calls to the generator being required.

generate_n has two related forms:

- checked_generate_n
- unchecked_generate_n

For information on how these functions behave, see Checked Iterators.

Example

```
// alg_generate_n.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>
#include <ostream>

int main() {
   using namespace std;

   // Assigning random values to vector integer elements
   vector <int> v1 ( 5 );
   vector <int>::iterator Iter1;
   deque <int> deq1 ( 5 );
   deque <int>::iterator d1_Iter;

   generate_n ( v1.begin ( ), 5 , rand );

   cout << "Vector v1 is ( " ;
```

```
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << ")." << endl;

    // Assigning random values to deque integer elements
    generate_n ( deq1.begin ( ), 3 , rand );

    cout << "Deque deq1 is ( " ;
    for ( d1_Iter = deq1.begin( ) ; d1_Iter != deq1.end( ) ; d1_Iter++ )
       cout << *d1_Iter << " ";
    cout << ")." << endl;
 }
```

Output

```
Vector v1 is ( 41 18467 6334 26500 19169 ).
Deque deq1 is ( 15724 11478 29358 0 0 ).
```

Requirements

Header: <algorithm>

See Also

Reference

generate_n (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

includes

Tests whether one sorted range contains all the elements contained in a second sorted range, where the ordering or equivalence criterion between elements may be specified by a binary predicate.

```
template<class InputIterator1, class InputIterator2>
   bool includes(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last1
   );
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
   bool includes(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last1,
      BinaryPredicate _Comp
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be tested for whether all the elements of the second are contained in the first.

_Last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be tested for whether all the elements of the second are contained in the first.

_First2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be tested for whether all the elements of the second are contained in the first.

_Last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be tested for whether all the elements of the second are contained in the first.

_Comp

User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

true if the first sorted range contains all the elements in the second sorted range; otherwise, **false**.

Remarks

Another way to think of this test is that it determined whether the second source range is a subset of the first source range.

The sorted source ranges referenced must be valid; all pointers must be dereferenceable and, within each sequence, the last position must be reachable from the first by incrementation.

The sorted source ranges must each be arranged as a precondition to the application of the algorithm includes in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The source ranges are not modified by the algorithm **merge**.

The value types of the input iterators need be less-than comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. More precisely, the algorithm tests whether all the elements in the first sorted range under a specified binary predicate have equivalent ordering to those in the second sorted range.

The complexity of the algorithm is linear with at most 2 * ((*_Last1* – *_First1*) – (*_Last2* – *_First2*)) – 1 comparisons for nonempty source ranges.

Example

```
// alg_includes.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser (int elem1, int elem2 )
{
   if ( elem1 < 0 )
      elem1 = - elem1;
   if ( elem2 < 0 )
      elem2 = - elem2;
   return elem1 < elem2;
}

int main( )
{
   using namespace std;
   vector <int> v1a, v1b;
   vector <int>::iterator Iter1a,  Iter1b;

   // Constructing vectors v1a & v1b with default less-than ordering
   int i;
   for ( i = -2 ; i <= 4 ; i++ )
   {
      v1a.push_back(  i );
   }

   int ii;
   for ( ii =-2 ; ii <= 3 ; ii++ )
   {
      v1b.push_back(  ii  );
   }

   cout << "Original vector v1a with range sorted by the\n "
        << "binary predicate less than is v1a = ( " ;
   for ( Iter1a = v1a.begin( ) ; Iter1a != v1a.end( ) ; Iter1a++ )
      cout << *Iter1a << " ";
   cout << ")." << endl;

   cout << "Original vector v1b with range sorted by the\n "
        <<  "binary predicate less than is v1b = ( " ;
   for ( Iter1b = v1b.begin ( ) ; Iter1b != v1b.end ( ) ; Iter1b++ )
      cout << *Iter1b << " ";
   cout << ")." << endl;

   // Constructing vectors v2a & v2b with ranges sorted by greater
   vector <int> v2a ( v1a ) , v2b ( v1b );
   vector <int>::iterator Iter2a,  Iter2b;
   sort ( v2a.begin ( ) , v2a.end ( ) , greater<int> ( ) );
   sort ( v2b.begin ( ) , v2b.end ( ) , greater<int> ( ) );
   v2a.pop_back ( );

   cout << "Original vector v2a with range sorted by the\n "
        <<  "binary predicate greater is v2a = ( " ;
   for ( Iter2a = v2a.begin ( ) ; Iter2a != v2a.end ( ) ; Iter2a++ )
      cout << *Iter2a << " ";
   cout << ")." << endl;

   cout << "Original vector v2b with range sorted by the\n "
        <<  "binary predicate greater is v2b = ( " ;
   for ( Iter2b = v2b.begin ( ) ; Iter2b != v2b.end ( ) ; Iter2b++ )
```

```
      cout << *Iter2b << " ";
   cout << ")." << endl;

   // Constructing vectors v3a & v3b with ranges sorted by mod_lesser
   vector <int> v3a ( v1a ), v3b ( v1b ) ;
   vector <int>::iterator Iter3a, Iter3b;
   reverse (v3a.begin( ), v3a.end( ) );
   v3a.pop_back ( );
   v3a.pop_back ( );
   sort ( v3a.begin ( ) , v3a.end ( ) , mod_lesser );
   sort ( v3b.begin ( ) , v3b.end ( ) , mod_lesser );

   cout << "Original vector v3a with range sorted by the\n "
        <<  "binary predicate mod_lesser is v3a = ( " ;
   for ( Iter3a = v3a.begin ( ) ; Iter3a != v3a.end ( ) ; Iter3a++ )
      cout << *Iter3a << " ";
   cout << ")." << endl;

   cout << "Original vector v3b with range sorted by the\n "
        <<  "binary predicate mod_lesser is v3b = ( " ;
   for ( Iter3b = v3b.begin ( ) ; Iter3b != v3b.end ( ) ; Iter3b++ )
      cout << *Iter3b << " ";
   cout << ")." << endl;

   // To test for inclusion under an asscending order
   // with the default binary predicate less <int> ( )
   bool Result1;
   Result1 = includes ( v1a.begin ( ) , v1a.end ( ) ,
      v1b.begin ( ) , v1b.end ( ) );
   if ( Result1 )
      cout << "All the elements in vector v1b are "
           << "contained in vector v1a." << endl;
   else
      cout << "At least one of the elements in vector v1b "
           << "is not contained in vector v1a." << endl;

   // To test for inclusion under descending
   // order specify binary predicate greater<int>( )
   bool Result2;
   Result2 = includes ( v2a.begin ( ) , v2a.end ( ) ,
      v2b.begin ( ) , v2b.end ( ) , greater <int> ( ) );
   if ( Result2 )
      cout << "All the elements in vector v2b are "
           << "contained in vector v2a." << endl;
   else
      cout << "At least one of the elements in vector v2b "
           << "is not contained in vector v2a." << endl;

   // To test for inclusion under a user
   // defined binary predicate mod_lesser
   bool Result3;
   Result3 = includes ( v3a.begin ( ) , v3a.end ( ) ,
      v3b.begin ( ) , v3b.end ( ) , mod_lesser );
   if ( Result3 )
      cout << "All the elements in vector v3b are "
           << "contained under mod_lesser in vector v3a."
           << endl;
   else
      cout << "At least one of the elements in vector v3b is "
           << " not contained under mod_lesser in vector v3a."
           << endl;
}
```

Output

```
Original vector v1a with range sorted by the
 binary predicate less than is v1a = ( -2 -1 0 1 2 3 4 ).
Original vector v1b with range sorted by the
```

```
 binary predicate less than is v1b = ( -2 -1 0 1 2 3 ).
Original vector v2a with range sorted by the
 binary predicate greater is v2a = ( 4 3 2 1 0 -1 ).
Original vector v2b with range sorted by the
 binary predicate greater is v2b = ( 3 2 1 0 -1 -2 ).
Original vector v3a with range sorted by the
 binary predicate mod_lesser is v3a = ( 0 1 2 3 4 ).
Original vector v3b with range sorted by the
 binary predicate mod_lesser is v3b = ( 0 -1 1 -2 2 3 ).
All the elements in vector v1b are contained in vector v1a.
At least one of the elements in vector v2b is not contained in vector v2a.
At least one of the elements in vector v3b is  not contained under mod_lesser in vector
v3a.
```

Requirements

Header: <algorithm>

See Also

Reference

includes (STL Samples)

Predicate Version of includes

Standard Template Library

Other Resources

<algorithm> Members

inplace_merge

Combines the elements from two consecutive sorted ranges into a single sorted range, where the ordering criterion may be specified by a binary predicate.

```
template<class BidirectionalIterator>
   void inplace_merge(
      BidirectionalIterator _First,
      BidirectionalIterator _Middle,
      BidirectionalIterator _Last
      );
template<class BidirectionalIterator, class Pr>
   void inplace_merge(
      BidirectionalIterator _First,
      BidirectionalIterator _Middle,
      BidirectionalIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

_First

A bidirectional iterator addressing the position of the first element in the first of two consecutive sorted ranges to be combined and sorted into a single range.

_Middle

A bidirectional iterator addressing the position of the first element in the second of two consecutive sorted ranges to be combined and sorted into a single range.

_Last

A bidirectional iterator addressing the position one past the last element in the second of two consecutive sorted ranges to be combined and sorted into a single range.

_Comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Remarks

The sorted consecutive ranges referenced must be valid; all pointers must be dereferenceable and, within each sequence, the last position must be reachable from the first by incrementation.

The sorted consecutive ranges must each be arranged as a precondition to the application of the **inplace_merge** algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges. The operation is stable as the relative order of elements within each range is preserved. When there are equivalent elements in both source ranges, the element is the first range precedes the element from the second in the combined range.

The complexity depends on the available memory as the algorithm allocates memory to a temporary buffer. If sufficient memory is available, the best case is linear with (*_Last* – *_First*) – 1 comparisons; if no auxiliary memory is available, the worst case is *N* log*(N)*, where *N* = (*_Last* – *_First*).

Example

```
// alg_inplace_merge.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      //For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
```

```
{
   if ( elem1 < 0 )
      elem1 = - elem1;
   if ( elem2 < 0 )
      elem2 = - elem2;
   return elem1 < elem2;
}

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1, Iter2, Iter3;

   // Constructing vector v1 with default less-than ordering
   int i;
   for ( i = 0 ; i <= 5 ; i++ )
   {
      v1.push_back( i );
   }

   int ii;
   for ( ii =-5 ; ii <= 0 ; ii++ )
   {
      v1.push_back(  ii  );
   }

   cout << "Original vector v1 with subranges sorted by the\n "
        <<  "binary predicate less than is  v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // Constructing vector v2 with ranges sorted by greater
   vector <int> v2 ( v1 );
   vector <int>::iterator break2;
   break2 = find ( v2.begin ( ) , v2.end ( ) , -5 );
   sort ( v2.begin ( ) , break2 , greater<int> ( ) );
   sort ( break2 , v2.end ( ) , greater<int> ( ) );
   cout << "Original vector v2 with subranges sorted by the\n "
        << "binary predicate greater is v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << ")" << endl;

   // Constructing vector v3 with ranges sorted by mod_lesser
   vector <int> v3 ( v1 );
   vector <int>::iterator break3;
   break3 = find ( v3.begin ( ) , v3.end ( ) , -5 );
   sort ( v3.begin ( ) , break3 , mod_lesser );
   sort ( break3 , v3.end ( ) , mod_lesser );
   cout << "Original vector v3 with subranges sorted by the\n "
        << "binary predicate mod_lesser is v3 = ( " ;
   for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
      cout << *Iter3 << " ";
   cout << ")" << endl;

   vector <int>::iterator break1;
   break1 = find (v1.begin ( ) , v1.end ( ) , -5 );
   inplace_merge ( v1.begin( ), break1, v1.end( ) );
   cout << "Merged inplace with default order,\n vector v1mod = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // To merge inplace in descending order, specify binary
   // predicate greater<int>( )
   inplace_merge ( v2.begin( ), break2 , v2.end( ) , greater<int>( ) );
```

```
    cout << "Merged inplace with binary predicate greater specified,\n "
         << "vector v2mod = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
       cout << *Iter2 << " ";
    cout << ")" << endl;

    // Applying a user defined (UD) binary predicate mod_lesser
    inplace_merge ( v3.begin( ), break3, v3.end( ), mod_lesser );
    cout << "Merged inplace with binary predicate mod_lesser specified,\n "
         << "vector v3mod = ( " ; ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
       cout << *Iter3 << " ";
    cout << ")" << endl;
  }
```

Output

```
Original vector v1 with subranges sorted by the
 binary predicate less than is  v1 = ( 0 1 2 3 4 5 -5 -4 -3 -2 -1 0 )
Original vector v2 with subranges sorted by the
 binary predicate greater is v2 = ( 5 4 3 2 1 0 0 -1 -2 -3 -4 -5 )
Original vector v3 with subranges sorted by the
 binary predicate mod_lesser is v3 = ( 0 1 2 3 4 5 0 -1 -2 -3 -4 -5 )
Merged inplace with default order,
 vector v1mod = ( -5 -4 -3 -2 -1 0 0 1 2 3 4 5 )
Merged inplace with binary predicate greater specified,
 vector v2mod = ( 5 4 3 2 1 0 0 -1 -2 -3 -4 -5 )
Merged inplace with binary predicate mod_lesser specified,
 vector v3mod = ( 0 0 1 -1 2 -2 3 -3 4 -4 5 -5 )
```

Requirements

Header: <algorithm>

See Also

Reference

inplace_merge (STL Samples)

Predicate Version of inplace_merge

Standard Template Library

Other Resources

<algorithm> Members

iter_swap

Exchanges two values referred to by a pair of specified iterators.

```
template<class ForwardIterator1, class ForwardIterator2>
   void iter_swap(
      ForwardIterator1 _Left,
      ForwardIterator2 _Right
   );
```

Parameters

_Left

One of the forward iterators whose value is to be exchanged.

_Right

The second of the forward iterators whose value is to be exchanged.

Remarks

swap should be used in preference to i**ter_swap**, which was included in the C++ Standard for backward compatibility. If *Fit1* and *Fit2* are forward iterators, then **iter_swap** (*Fit1*, *Fit2*), is equivalent to **swap** (**Fit1*, **Fit2*).

The value types of the input forward iterators must have the same value.

Example

```
// alg_iter_swap.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>
#include <ostream>

using namespace std;
class CInt;
ostream& operator<<( ostream& osIn, const CInt& rhs );

class CInt
{
public:
   CInt( int n = 0 ) : m_nVal( n ){}
   CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){}
   CInt&   operator=( const CInt& rhs ) { m_nVal =
   rhs.m_nVal; return *this; }
   bool operator<( const CInt& rhs ) const
      { return ( m_nVal < rhs.m_nVal );}
   friend ostream& operator<<( ostream& osIn, const CInt& rhs );

private:
   int m_nVal;
};

inline ostream& operator<<( ostream& osIn, const CInt& rhs )
{
   osIn << "CInt(" << rhs.m_nVal << ")";
   return osIn;
}

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
   if ( elem1 < 0 )
      elem1 = - elem1;
```

```
   if ( elem2 < 0 )
      elem2 = - elem2;
   return elem1 < elem2;
};

int main( )
{
   CInt c1 = 5, c2 = 1, c3 = 10;
   deque<CInt> deq1;
   deque<CInt>::iterator d1_Iter;

   deq1.push_back ( c1 );
   deq1.push_back ( c2 );
   deq1.push_back ( c3 );

   cout << "The original deque of CInts is deq1 = (";
   for ( d1_Iter = deq1.begin( ); d1_Iter != --deq1.end( ); d1_Iter++ )
      cout << " " << *d1_Iter << ",";
   d1_Iter = --deq1.end( );
   cout << " " << *d1_Iter << " )." << endl;

   // Exchanging first and last elements with iter_swap
   iter_swap ( deq1.begin ( ) , --deq1.end ( ) );

   cout << "The deque of CInts with first & last elements swapped is:\n deq1 = (";
   for ( d1_Iter = deq1.begin( ); d1_Iter != --deq1.end( ); d1_Iter++ )
      cout << " " << *d1_Iter << ",";
   d1_Iter = --deq1.end( );
   cout << " " << *d1_Iter << " )." << endl;

   // Swapping back first and last elements with swap
   swap ( *deq1.begin ( ) , *(deq1.end ( ) -1 ) );

   cout << "The deque of CInts with first & last elements swapped back is:\n deq1 = (";
   for ( d1_Iter = deq1.begin( ); d1_Iter != --deq1.end( ); d1_Iter++ )
      cout << " " << *d1_Iter << ",";
   d1_Iter = --deq1.end( );
   cout << " " << *d1_Iter << " )." << endl;

   // Swapping a vector element with a deque element
   vector <int> v1;
   vector <int>::iterator Iter1;
   deque <int> deq2;
   deque <int>::iterator d2_Iter;

   int i;
   for ( i = 0 ; i <= 3 ; i++ )
   {
      v1.push_back( i );
   }

   int ii;
   for ( ii = 4 ; ii <= 5 ; ii++ )
   {
      deq2.push_back( ii );
   }

   cout << "Vector v1 is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;

   cout << "Deque deq2 is ( " ;
   for ( d2_Iter = deq2.begin( ) ; d2_Iter != deq2.end( ) ; d2_Iter++ )
      cout << *d2_Iter << " ";
   cout << ")." << endl;

   iter_swap ( v1.begin ( ) , deq2.begin ( ) );
```

```
    cout << "After exchanging first elements,\n vector v1 is: v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << "))." << endl << " & deque deq2 is: deq2 = ( ";
    for ( d2_Iter = deq2.begin( ) ; d2_Iter != deq2.end( ) ; d2_Iter++ )
       cout << *d2_Iter << " ";
    cout << "))." << endl;
  }
```

Output

```
The original deque of CInts is deq1 = ( CInt(5), CInt(1), CInt(10) ).
The deque of CInts with first & last elements swapped is:
 deq1 = ( CInt(10), CInt(1), CInt(5) ).
The deque of CInts with first & last elements swapped back is:
 deq1 = ( CInt(5), CInt(1), CInt(10) ).
Vector v1 is ( 0 1 2 3 ).
Deque deq2 is ( 4 5 ).
After exchanging first elements,
 vector v1 is: v1 = ( 4 1 2 3 ).
 & deque deq2 is: deq2 = ( 0 5 ).
```

Requirements

Header: <algorithm>

See Also

Reference

iter_swap (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

lexicographical_compare

Compares element by element between two sequences to determine which is lesser of the two.

```
template<class InputIterator1, class InputIterator2>
   bool lexicographical_compare(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2
   );
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
   bool lexicographical_compare(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      BinaryPredicate _Comp
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first range to be compared.

_Last1

An input iterator addressing the position one past the final element in the first range to be compared.

_First2

An input iterator addressing the position of the first element in the second range to be compared.

_Last2

An input iterator addressing the position one past the final element in the second range to be compared.

_Comp

User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

true if the first range is lexicographically less than the second range; otherwise **false**.

Remarks

A lexicographical comparison between sequences compares them element by element until:

- It finds two corresponding elements unequal, and the result of their comparison is taken as the result of the comparison between sequences.
- No inequalities are found, but one sequence has more elements than the other, and the shorter sequence is considered less than the longer sequence.
- No inequalities are found and the sequences have the same number of elements, and so the sequences are equal and the result of the comparison is false.

Example

```
// alg_lex_comp.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
```

```
// Return whether second element is twice the first
bool twice ( int elem1, int elem2 )
{
   return 2 * elem1 < elem2;
}

int main( )
{
   using namespace std;
   vector <int> v1, v2;
   list <int> L1;
   vector <int>::iterator Iter1, Iter2;
   list <int>::iterator L1_Iter, L1_inIter;

   int i;
   for ( i = 0 ; i <= 5 ; i++ )
   {
      v1.push_back( 5 * i );
   }
   int ii;
   for ( ii = 0 ; ii <= 6 ; ii++ )
   {
      L1.push_back( 5 * ii );
   }

   int iii;
   for ( iii = 0 ; iii <= 5 ; iii++ )
   {
      v2.push_back( 10 * iii );
   }

   cout << "Vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   cout << "List L1 = ( " ;
   for ( L1_Iter = L1.begin( ) ; L1_Iter!= L1.end( ) ; L1_Iter++ )
      cout << *L1_Iter << " ";
   cout << ")" << endl;

   cout << "Vector v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
      cout << ")" << endl;

   // Self lexicographical_comparison of v1 under identity
   bool result1;
   result1 = lexicographical_compare (v1.begin( ), v1.end( ),
                  v1.begin( ), v1.end( ) );
   if ( result1 )
      cout << "Vector v1 is lexicographically_less than v1." << endl;
   else
      cout << "Vector v1 is not lexicographically_less than v1." << endl;

   // lexicographical_comparison of v1 and L2 under identity
   bool result2;
   result2 = lexicographical_compare (v1.begin( ), v1.end( ),
                  L1.begin( ), L1.end( ) );
   if ( result2 )
      cout << "Vector v1 is lexicographically_less than L1." << endl;
   else
      cout << "Vector v1 is lexicographically_less than L1." << endl;

   bool result3;
   result3 = lexicographical_compare (v1.begin( ), v1.end( ),
                  v2.begin( ), v2.end( ), twice );
```

```
    if ( result3 )
       cout << "Vector v1 is lexicographically_less than v2 "
            << "under twice." << endl;
    else
       cout << "Vector v1 is not lexicographically_less than v2 "
            << "under twice." << endl;
  }
```

Output

```
Vector v1 = ( 0 5 10 15 20 25 )
List L1 = ( 0 5 10 15 20 25 30 )
Vector v2 = ( 0 10 20 30 40 50 )
Vector v1 is not lexicographically_less than v1.
Vector v1 is lexicographically_less than L1.
Vector v1 is not lexicographically_less than v2 under twice.
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

lower_bound

Finds the position of the first element in an ordered range that has a value greater than or equivalent to a specified value, where the ordering criterion may be specified by a binary predicate.

```
template<class ForwardIterator, class Type>
   ForwardIterator lower_bound(
      ForwardIterator _First,
      ForwardIterator _Last,
      const Type& _Val
   );
template<class ForwardIterator, class Type, class BinaryPredicate>
   ForwardIterator lower_bound(
      ForwardIterator _First,
      ForwardIterator _Last,
      const Type& _Val,
      BinaryPredicate _Comp
   );
```

Parameters

_First

A forward iterator addressing the position of the first element in the range to be searched.

_Last

A forward iterator addressing the position one past the final element in the range to be searched.

_Val

The value whose first position or possible first position is being searched for in the ordered range.

_Comp

User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

A forward iterator at the position of the first element in an ordered range with a value that is greater than or equivalent to a specified value, where the equivalence is specified with a binary predicate.

Remarks

The sorted source range referenced must be valid; all iterators must be dereferenceable and within the sequence the last position must be reachable from the first by incrementation.

A sorted range is a precondition of using **lower_bound** and where the ordering is the same as specified by with binary predicate.

The range is not modified by the algorithm **lower_bound**.

The value types of the forward iterators need be less-than comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements

The complexity of the algorithm is logarithmic for random-access iterators and linear otherwise, with the number of steps proportional to (*_Last1* – *_First1*).

Example

```
// alg_lower_bound.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>
```

```
// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
   if ( elem1 < 0 )
      elem1 = - elem1;
   if ( elem2 < 0 )
      elem2 = - elem2;
   return elem1 < elem2;
}

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1, Result1;

   // Constructing vectors v1a & v1b with default less than ordering
   int i;
   for ( i = -1 ; i <= 4 ; i++ )
   {
      v1.push_back(  i );
   }

   int ii;
   for ( ii =-3 ; ii <= 0 ; ii++ )
   {
      v1.push_back(  ii  );
   }

   sort ( v1.begin ( ) , v1.end ( ) );
   cout << "Original vector v1 with range sorted by the\n "
        << "binary predicate less than is  v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;

   // Constructing vectors v2 with range sorted by greater
   vector <int> v2 ( v1 );
   vector <int>::iterator Iter2, Result2;
   sort ( v2.begin ( ) , v2.end ( ) , greater<int> ( ) );

   cout << "Original vector v2 with range sorted by the\n "
        << "binary predicate greater is    v2 = ( " ;
   for ( Iter2 = v2.begin ( ) ; Iter2 != v2.end ( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << ")." << endl;

   // Constructing vectors v3 with range sorted by mod_lesser
   vector <int> v3 ( v1 );
   vector <int>::iterator Iter3, Result3;
   sort ( v3.begin ( ) , v3.end ( ) , mod_lesser );

   cout << "Original vector v3 with range sorted by the\n "
        <<  "binary predicate mod_lesser is v3 = ( " ;
   for ( Iter3 = v3.begin ( ) ; Iter3 != v3.end ( ) ; Iter3++ )
      cout << *Iter3 << " ";
   cout << ")." << endl;

   // lower_bound of 3 in v1 with default binary predicate less <int> ( )
   Result1 = lower_bound ( v1.begin ( ) , v1.end ( ) , 3 );
   cout << "The lower_bound in v2 for the element with a value of 3 is: "
        << *Result1 << "." << endl;

   // lower_bound of 3 in v2 with the binary predicate greater <int> ( )
   Result2 = lower_bound ( v2.begin ( ) , v2.end ( ) , 3, greater <int> ( ) );
   cout << "The lower_bound in v2 for the element with a value of 3 is: "
        << *Result2 << "." << endl;
```

```
    // lower_bound of 3 in v3 with the binary predicate  mod_lesser
    Result3 = lower_bound ( v3.begin ( ) , v3.end ( ) , 3,  mod_lesser  );
    cout << "The lower_bound in v3 for the element with a value of 3 is: "
         << *Result3 << "." << endl;
}
```

Output

```
Original vector v1 with range sorted by the
 binary predicate less than is  v1 = ( -3 -2 -1 -1 0 0 1 2 3 4 ).
Original vector v2 with range sorted by the
 binary predicate greater is    v2 = ( 4 3 2 1 0 0 -1 -1 -2 -3 ).
Original vector v3 with range sorted by the
 binary predicate mod_lesser is v3 = ( 0 0 -1 -1 1 -2 2 -3 3 4 ).
The lower_bound in v2 for the element with a value of 3 is: 3.
The lower_bound in v2 for the element with a value of 3 is: 3.
The lower_bound in v3 for the element with a value of 3 is: -3.
```

Requirements

Header: <algorithm>

See Also

Reference

lower_bound (STL Samples)

Predicate Version of lower_bound

Standard Template Library

Other Resources

<algorithm> Members

make_heap

Converts elements from a specified range into a heap in which the first element is the largest and for which a sorting criterion may be specified with a binary predicate.

```
template<class RandomAccessIterator>
   void make_heap(
      RandomAccessIterator _First,
      RandomAccessIterator _Last
   );
template<class RandomAccessIterator, class BinaryPredicate>
   void make_heap(
      RandomAccessIterator _First,
      RandomAccessIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

_First

A random-access iterator addressing the position of the first element in the range to be converted into a heap.

_Last

A random-access iterator addressing the position one past the final element in the range to be converted into a heap.

_Comp

User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

Heaps have two properties:

- The first element is always the largest.
- Elements may be added or removed in logarithmic time.

Heaps are an ideal way to implement priority queues and they are used in the implementation of the Standard Template Library container adaptor priority_queue Class.

The complexity is linear, requiring 3 * (*_Last* – *_First*) comparisons.

Example

```
// alg_make_heap.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

int main() {
   using namespace std;
   vector <int> v1, v2;
   vector <int>::iterator Iter1, Iter2;

   int i;
   for ( i = 0 ; i <= 9 ; i++ )
      v1.push_back( i );

   random_shuffle( v1.begin( ), v1.end( ) );

   cout << "Vector v1 is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
```

```
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // Make v1 a heap with default less than ordering
   make_heap ( v1.begin( ), v1.end( ) );
   cout << "The heaped version of vector v1 is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // Make v1 a heap with greater than ordering
   make_heap ( v1.begin( ), v1.end( ), greater<int>( ) );
   cout << "The greater-than heaped version of v1 is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;
}
```

Sample Output

```
Vector v1 is ( 8 1 9 2 0 5 7 3 4 6 ).
The heaped version of vector v1 is ( 9 6 8 4 1 5 7 3 2 0 ).
The greater-than heaped version of v1 is ( 0 1 5 2 6 8 7 3 4 9 ).
```

Requirements

Header: <algorithm>

See Also

Reference

heap

Standard Template Library

Other Resources

<algorithm> Members

max

Compares two objects and returns the larger of the two, where the ordering criterion may be specified by a binary predicate.

```
template<class Type>
   const Type& max(
      const Type& _Left,
      const Type& _Right
   );
template<class Type, class Pr>
   const Type& max(
      const Type& _Left,
      const Type& _Right,
      BinaryPredicate _Comp
   );
```

Parameters

_Left

The first of the two objects being compared.

_Right

The second of the two objects being compared.

_Comp

A binary predicate used to compare the two objects.

Return Value

The greater of the two objects unless neither is greater, in which case it returns the first of the two objects.

Remarks

The max algorithm is unusual in having objects passed as parameters. Most Standard Template Library algorithms operate on a range of elements whose position is specified by iterators passes as parameters.

Example

```
// alg_max.cpp
// compile with: /EHsc
#include <vector>
#include <set>
#include <algorithm>
#include <iostream>
#include <ostream>

using namespace std;
class CInt;
ostream& operator<<( ostream& osIn, const CInt& rhs );

class CInt
{
public:
   CInt( int n = 0 ) : m_nVal( n ){}
   CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){}
   CInt&   operator=( const CInt& rhs ) {m_nVal =
   rhs.m_nVal; return *this;}
   bool operator<( const CInt& rhs ) const
      {return ( m_nVal < rhs.m_nVal );}
   friend ostream& operator<<( ostream& osIn, const CInt& rhs );

private:
   int m_nVal;
};
```

```
inline ostream& operator<<( ostream& osIn, const CInt& rhs )
{
   osIn << "CInt( " << rhs.m_nVal << " )";
   return osIn;
}

// Return whether modulus of elem1 is greater than modulus of elem2
bool mod_greater ( int elem1, int elem2 )
{
   if ( elem1 < 0 )
      elem1 = - elem1;
   if ( elem2 < 0 )
      elem2 = - elem2;
   return elem1 > elem2;
};

int main( )
{
   // Comparing integers directly using the max algorithm
   int a = 6, b = -7, c = 7;
   const int& result1 = max ( a, b, mod_greater );
   const int& result2 = max ( b, c );

   cout << "The mod_greater of the integers 6 & -7 is: "
        << result1 << "." << endl;
   cout << "The larger of the integers -7 & 7 is: "
        << result2 << "." << endl;
   cout << endl;

   // Comparing set containers with elements of type CInt
   // using the max algorithm
   CInt c1 = 1, c2 = 2, c3 = 3;
   set<CInt> s1, s2, s3;
   set<CInt>::iterator s1_Iter, s2_Iter, s3_Iter;

   s1.insert ( c1 );
   s1.insert ( c2 );
   s2.insert ( c2 );
   s2.insert ( c3 );

   cout << "s1 = (";
   for ( s1_Iter = s1.begin( ); s1_Iter != --s1.end( ); s1_Iter++ )
      cout << " " << *s1_Iter << ",";
   s1_Iter = --s1.end( );
   cout << " " << *s1_Iter << " )." << endl;

   cout << "s2 = (";
   for ( s2_Iter = s2.begin( ); s2_Iter != --s2.end( ); s2_Iter++ )
      cout << " " << *s2_Iter << ",";
   s2_Iter = --s2.end( );
   cout << " " << *s2_Iter << " )." << endl;

   s3 = max ( s1, s2 );
   cout << "s3 = max ( s1, s2 ) = (";
   for ( s3_Iter = s3.begin( ); s3_Iter != --s3.end( ); s3_Iter++ )
      cout << " " << *s3_Iter << ",";
   s3_Iter = --s3.end( );
   cout << " " << *s3_Iter << " )." << endl << endl;

   // Comparing vectors with integer elements using the max algorithm
   vector <int> v1, v2, v3, v4, v5;
   vector <int>::iterator Iter1, Iter2, Iter3, Iter4, Iter5;

   int i;
   for ( i = 0 ; i <= 2 ; i++ )
   {
      v1.push_back( i );
```

```
    }

    int ii;
    for ( ii = 0 ; ii <= 2 ; ii++ )
    {
       v2.push_back( ii );
    }

    int iii;
    for ( iii = 0 ; iii <= 2 ; iii++ )
    {
       v3.push_back( 2 * iii );
    }

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << "))." << endl;

    cout << "Vector v2 is ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
       cout << *Iter2 << " ";
    cout << "))." << endl;

    cout << "Vector v3 is ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
       cout << *Iter3 << " ";
    cout << "))." << endl;

    v4 = max ( v1, v2 );
    v5 = max ( v1, v3 );

    cout << "Vector v4 = max (v1,v2) is ( " ;
    for ( Iter4 = v4.begin( ) ; Iter4 != v4.end( ) ; Iter4++ )
       cout << *Iter4 << " ";
    cout << "))." << endl;

    cout << "Vector v5 = max (v1,v3) is ( " ;
    for ( Iter5 = v5.begin( ) ; Iter5 != v5.end( ) ; Iter5++ )
       cout << *Iter5 << " ";
    cout << "))." << endl;
  }
```

Output

```
The mod_greater of the integers 6 & -7 is: 6.
The larger of the integers -7 & 7 is: 7.

s1 = ( CInt( 1 ), CInt( 2 ) ).
s2 = ( CInt( 2 ), CInt( 3 ) ).
s3 = max ( s1, s2 ) = ( CInt( 2 ), CInt( 3 ) ).

Vector v1 is ( 0 1 2 ).
Vector v2 is ( 0 1 2 ).
Vector v3 is ( 0 2 4 ).
Vector v4 = max (v1,v2) is ( 0 1 2 ).
Vector v5 = max (v1,v3) is ( 0 2 4 ).
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

max_element

Finds the first occurrence of largest element in a specified range where the ordering criterion may be specified by a binary predicate.

```
template<class ForwardIterator>
   ForwardIterator max_element(
      ForwardIterator _First,
      ForwardIterator _Last
   );
template<class ForwardIterator, class BinaryPredicate>
   ForwardIterator max_element(
      ForwardIterator _First,
      ForwardIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

_First

A forward iterator addressing the position of the first element in the range to be searched for the largest element.

_Last

A forward iterator addressing the position one past the final element in the range to be searched for the largest element.

_Comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

A forward iterator addressing the position of the first occurrence of the largest element in the range searched.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within each sequence the last position is reachable from the first by incrementation.

The complexity is linear: (*_Last* – *_First*) – 1 comparisons are required for a nonempty range.

Example

```
// alg_max_element.cpp
// compile with: /EHsc
#include <vector>
#include <set>
#include <algorithm>
#include <iostream>
#include <ostream>

using namespace std;
class CInt;
ostream& operator<<( ostream& osIn, const CInt& rhs );

class CInt
{
public:
   CInt( int n = 0 ) : m_nVal( n ){}
   CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){}
   CInt& operator=( const CInt& rhs ) {m_nVal =
   rhs.m_nVal; return *this;}
   bool operator<( const CInt& rhs ) const
      {return ( m_nVal < rhs.m_nVal );}
```

```
   friend ostream& operator<<( ostream& osIn, const CInt& rhs );

private:
   int m_nVal;
};

inline ostream& operator<<(ostream& osIn, const CInt& rhs)
{
   osIn << "CInt( " << rhs.m_nVal << " )";
   return osIn;
}

// Return whether modulus of elem1 is greater than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
   if ( elem1 < 0 )
      elem1 = - elem1;
   if ( elem2 < 0 )
      elem2 = - elem2;
   return elem1 < elem2;
};

int main( )
{
   // Searching a set container with elements of type CInt
   // for the maximum element
   CInt c1 = 1, c2 = 2, c3 = -3;
   set<CInt> s1;
   set<CInt>::iterator s1_Iter, s1_R1_Iter, s1_R2_Iter;

   s1.insert ( c1 );
   s1.insert ( c2 );
   s1.insert ( c3 );

   cout << "s1 = (";
   for ( s1_Iter = s1.begin( ); s1_Iter != --s1.end( ); s1_Iter++ )
      cout << " " << *s1_Iter << ",";
   s1_Iter = --s1.end( );
   cout << " " << *s1_Iter << " )." << endl;

   s1_R1_Iter = max_element ( s1.begin ( ) , s1.end ( ) );

   cout << "The largest element in s1 is: " << *s1_R1_Iter << endl;
   cout << endl;

   // Searching a vector with elements of type int for the maximum
   // element under default less than & mod_lesser binary predicates
   vector <int> v1;
   vector <int>::iterator v1_Iter, v1_R1_Iter, v1_R2_Iter;

   int i;
   for ( i = 0 ; i <= 3 ; i++ )
   {
      v1.push_back( i );
   }

   int ii;
   for ( ii = 1 ; ii <= 4 ; ii++ )
   {
      v1.push_back( - 2 * ii );
   }

   cout << "Vector v1 is ( " ;
   for ( v1_Iter = v1.begin( ) ; v1_Iter != v1.end( ) ; v1_Iter++ )
      cout << *v1_Iter << " ";
   cout << ")." << endl;

   v1_R1_Iter = max_element ( v1.begin ( ) , v1.end ( ) );
```

```
    v1_R2_Iter = max_element ( v1.begin ( ) , v1.end ( ), mod_lesser);

    cout << "The largest element in v1 is: " << *v1_R1_Iter << endl;
    cout << "The largest element in v1 under the mod_lesser"
         << "\n binary predicate is: " << *v1_R2_Iter << endl;
  }
```

Output

```
s1 = ( CInt( -3 ), CInt( 1 ), CInt( 2 ) ).
The largest element in s1 is: CInt( 2 )

Vector v1 is ( 0 1 2 3 -2 -4 -6 -8 ).
The largest element in v1 is: 3
The largest element in v1 under the mod_lesser
 binary predicate is: -8
```

Requirements

Header: <algorithm>

See Also

Reference

Nonpredicate Version of max_element

Predicate Version of max_element

Standard Template Library

Other Resources

<algorithm> Members

merge

Combines all of the elements from two sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
   OutputIterator merge(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPred
icate>
   OutputIterator merge(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
      BinaryPredicate _Comp
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be combined and sorted into a single range.

_Last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be combined and sorted into a single range.

_First2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be combined and sorted into a single range.

_Last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be combined and sorted into a single range.

_Result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be combined into a single sorted range.

_Comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range.

Remarks

The sorted source ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation.

The destination range should not overlap either of the source ranges and should be large enough to contain the destination range.

The sorted source ranges must each be arranged as a precondition to the application of the **merge** algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The operation is stable as the relative order of elements within each range is preserved in the destination range. The source ranges are not modified by the algorithm **merge**.

The value types of the input iterators need be less-than comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. When there are equivalent elements in both source ranges, the elements in the first range precede the elements from the second source range in the destination range.

The complexity of the algorithm is linear with at most (*_Last1* – *_First1*) – (*_Last2* – *_First2*) – 1 comparisons.

The list class provides a member function merge to merge the elements of two lists.

merge has two related forms:

checked_merge

unchecked_merge

For information on how these functions behave, see Checked Iterators.

Example

```
// alg_merge.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>   // For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 ) {
   if (elem1 < 0)
      elem1 = - elem1;
   if (elem2 < 0)
      elem2 = - elem2;
   return elem1 < elem2;
}

int main() {
   using namespace std;
   vector <int> v1a, v1b, v1 ( 12 );
   vector <int>::iterator Iter1a,  Iter1b, Iter1;

   // Constructing vector v1a and v1b with default less than ordering
   int i;
   for ( i = 0 ; i <= 5 ; i++ )
      v1a.push_back(  i );

   int ii;
   for ( ii =-5 ; ii <= 0 ; ii++ )
      v1b.push_back(  ii  );

   cout << "Original vector v1a with range sorted by the\n "
        << "binary predicate less than is  v1a = ( " ;
   for ( Iter1a = v1a.begin( ) ; Iter1a != v1a.end( ) ; Iter1a++ )
      cout << *Iter1a << " ";
   cout << ")." << endl;

   cout << "Original vector v1b with range sorted by the\n "
        << "binary predicate less than is  v1b = ( " ;
   for ( Iter1b = v1b.begin ( ) ; Iter1b != v1b.end ( ) ; Iter1b++ )
      cout << *Iter1b << " ";
   cout << ")." << endl;

   // Constructing vector v2 with ranges sorted by greater
   vector <int> v2a ( v1a ) , v2b ( v1b ) ,  v2 ( v1 );
   vector <int>::iterator Iter2a,  Iter2b, Iter2;
```

```
    sort ( v2a.begin ( ) , v2a.end ( ) , greater<int> ( ) );
    sort ( v2b.begin ( ) , v2b.end ( ) , greater<int> ( ) );

    cout << "Original vector v2a with range sorted by the\n "
         <<  "binary predicate greater is   v2a =  ( " ;
    for ( Iter2a = v2a.begin ( ) ; Iter2a != v2a.end ( ) ; Iter2a++ )
       cout << *Iter2a << " ";
    cout << ")." << endl;

    cout << "Original vector v2b with range sorted by the\n "
         <<  "binary predicate greater is   v2b =  ( " ;
    for ( Iter2b = v2b.begin ( ) ; Iter2b != v2b.end ( ) ; Iter2b++ )
       cout << *Iter2b << " ";
    cout << ")." << endl;

    // Constructing vector v3 with ranges sorted by mod_lesser
    vector <int> v3a ( v1a ), v3b ( v1b ) ,  v3 ( v1 );
    vector <int>::iterator Iter3a,  Iter3b, Iter3;
    sort ( v3a.begin ( ) , v3a.end ( ) , mod_lesser );
    sort ( v3b.begin ( ) , v3b.end ( ) , mod_lesser );

    cout << "Original vector v3a with range sorted by the\n "
         << "binary predicate mod_lesser is   v3a =  ( " ;
    for ( Iter3a = v3a.begin ( ) ; Iter3a != v3a.end ( ) ; Iter3a++ )
       cout << *Iter3a << " ";
    cout << ")." << endl;

    cout << "Original vector v3b with range sorted by the\n "
         << "binary predicate mod_lesser is   v3b =  ( " ;
    for ( Iter3b = v3b.begin ( ) ; Iter3b != v3b.end ( ) ; Iter3b++ )
       cout << *Iter3b << " ";
    cout << ")." << endl;

    // To merge inplace in ascending order with default binary
    // predicate less <int> ( )
    merge ( v1a.begin ( ) , v1a.end ( ) , v1b.begin ( ) , v1b.end ( ) , v1.begin ( ) );
    cout << "Merged inplace with default order,\n vector v1mod =  ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << ")." << endl;

    // To merge inplace in descending order, specify binary
    // predicate greater<int>( )
    merge ( v2a.begin ( ) , v2a.end ( ) , v2b.begin ( ) , v2b.end ( ) ,
        v2.begin ( ) ,  greater <int> ( ) );
    cout << "Merged inplace with binary predicate greater specified,\n "
         << "vector v2mod  = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
       cout << *Iter2 << " ";
    cout << ")." << endl;

    // Applying A user-defined (UD) binary predicate mod_lesser
    merge ( v3a.begin ( ) , v3a.end ( ) , v3b.begin ( ) , v3b.end ( ) ,
        v3.begin ( ) ,  mod_lesser );
    cout << "Merged inplace with binary predicate mod_lesser specified,\n "
         << "vector v3mod  = ( " ; ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
       cout << *Iter3 << " ";
    cout << ")." << endl;
  }
```

Output

```
Original vector v1a with range sorted by the
 binary predicate less than is  v1a = ( 0 1 2 3 4 5 ).
Original vector v1b with range sorted by the
 binary predicate less than is  v1b = ( -5 -4 -3 -2 -1 0 ).
Original vector v2a with range sorted by the
```

```
 binary predicate greater is   v2a =  ( 5 4 3 2 1 0 ).
Original vector v2b with range sorted by the
 binary predicate greater is   v2b =  ( 0 -1 -2 -3 -4 -5 ).
Original vector v3a with range sorted by the
 binary predicate mod_lesser is   v3a =  ( 0 1 2 3 4 5 ).
Original vector v3b with range sorted by the
 binary predicate mod_lesser is   v3b =  ( 0 -1 -2 -3 -4 -5 ).
Merged inplace with default order,
 vector v1mod =  ( -5 -4 -3 -2 -1 0 0 1 2 3 4 5 ).
Merged inplace with binary predicate greater specified,
 vector v2mod  = ( 5 4 3 2 1 0 0 -1 -2 -3 -4 -5 ).
Merged inplace with binary predicate mod_lesser specified,
 vector v3mod  = ( 0 0 1 -1 2 -2 3 -3 4 -4 5 -5 ).
```

Requirements

Header: <algorithm>

See Also

Reference

merge (STL Samples)

Predicate Version of merge

Standard Template Library

Other Resources

<algorithm> Members

min

Compares two objects and returns the lesser of the two, where the ordering criterion may be specified by a binary predicate.

```
template<class Type>
   const Type& min(
      const Type& _Left,
      const Type& _Right
   );
template<class Type, class Pr>
   const Type& min(
      const Type& _Left,
      const Type& _Right,
      BinaryPredicate _Comp
   );
```

Parameters

_Left

The first of the two objects being compared.

_Right

The second of the two objects being compared.

_Comp

A binary predicate used to compare the two objects.

Return Value

The lesser of the two objects unless neither is lesser, in which case it returns the first of the two objects.

Remarks

The **min** algorithm is unusual in having objects passed as parameters. Most Standard Template Library algorithms operate on a range of elements whose position is specified by iterators passed as parameters.

Example

```
// alg_min.cpp
// compile with: /EHsc
#include <vector>
#include <set>
#include <algorithm>
#include <iostream>
#include <ostream>

using namespace std;
class CInt;
ostream& operator<<( ostream& osIn, const CInt& rhs );

class CInt
{
public:
   CInt( int n = 0 ) : m_nVal( n ){}
   CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){}
   CInt& operator=( const CInt& rhs ) {m_nVal =
   rhs.m_nVal; return *this;}
   bool operator<( const CInt& rhs ) const
      {return ( m_nVal < rhs.m_nVal );}
   friend ostream& operator<<(ostream& osIn, const CInt& rhs);

private:
   int m_nVal;
};
```

```
inline ostream& operator<<( ostream& osIn, const CInt& rhs )
{
   osIn << "CInt( " << rhs.m_nVal << " )";
   return osIn;
}

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
   if ( elem1 < 0 )
      elem1 = - elem1;
   if ( elem2 < 0 )
      elem2 = - elem2;
   return elem1 < elem2;
};

int main( )
{
   // Comparing integers directly using the min algorithm with
   // binary predicate mod_lesser & with default less than
   int a = 6, b = -7, c = 7 ;
   const int& result1 = min ( a, b, mod_lesser );
   const int& result2 = min ( b, c );

   cout << "The mod_lesser of the integers 6 & -7 is: "
        << result1 << "." << endl;
   cout << "The lesser of the integers -7 & 7 is: "
        << result2 << "." << endl;
   cout << endl;

   // Comparing set containers with elements of type CInt
   // using the min algorithm
   CInt c1 = 1, c2 = 2, c3 = 3;
   set<CInt> s1, s2, s3;
   set<CInt>::iterator s1_Iter, s2_Iter, s3_Iter;

   s1.insert ( c1 );
   s1.insert ( c2 );
   s2.insert ( c2 );
   s2.insert ( c3 );

   cout << "s1 = (";
   for ( s1_Iter = s1.begin( ); s1_Iter != --s1.end( ); s1_Iter++ )
      cout << " " << *s1_Iter << ",";
   s1_Iter = --s1.end( );
   cout << " " << *s1_Iter << " )." << endl;

   cout << "s2 = (";
   for ( s2_Iter = s2.begin( ); s2_Iter != --s2.end( ); s2_Iter++ )
      cout << " " << *s2_Iter << ",";
   s2_Iter = --s2.end( );
   cout << " " << *s2_Iter << " )." << endl;

   s3 = min ( s1, s2 );
   cout << "s3 = min ( s1, s2 ) = (";
   for ( s3_Iter = s3.begin( ); s3_Iter != --s3.end( ); s3_Iter++ )
      cout << " " << *s3_Iter << ",";
   s3_Iter = --s3.end( );
   cout << " " << *s3_Iter << " )." << endl << endl;

   // Comparing vectors with integer elements using min algorithm
   vector <int> v1, v2, v3, v4, v5;
   vector <int>::iterator Iter1, Iter2, Iter3, Iter4, Iter5;

   int i;
   for ( i = 0 ; i <= 2 ; i++ )
   {
```

```
      v1.push_back( i );
   }

   int ii;
   for ( ii = 0 ; ii <= 2 ; ii++ )
   {
      v2.push_back( ii );
   }

   int iii;
   for ( iii = 0 ; iii <= 2 ; iii++ )
   {
      v3.push_back( 2 * iii );
   }

   cout << "Vector v1 is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;

   cout << "Vector v2 is ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << ")." << endl;

   cout << "Vector v3 is ( " ;
   for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
      cout << *Iter3 << " ";
   cout << ")." << endl;

   v4 = min ( v1, v2 );
   v5 = min ( v1, v3 );

   cout << "Vector v4 = min ( v1,v2 ) is ( " ;
   for ( Iter4 = v4.begin( ) ; Iter4 != v4.end( ) ; Iter4++ )
      cout << *Iter4 << " ";
   cout << ")." << endl;

   cout << "Vector v5 = min ( v1,v3 ) is ( " ;
   for ( Iter5 = v5.begin( ) ; Iter5 != v5.end( ) ; Iter5++ )
      cout << *Iter5 << " ";
   cout << ")." << endl;
}
```

Output

```
The mod_lesser of the integers 6 & -7 is: 6.
The lesser of the integers -7 & 7 is: -7.

s1 = ( CInt( 1 ), CInt( 2 ) ).
s2 = ( CInt( 2 ), CInt( 3 ) ).
s3 = min ( s1, s2 ) = ( CInt( 1 ), CInt( 2 ) ).

Vector v1 is ( 0 1 2 ).
Vector v2 is ( 0 1 2 ).
Vector v3 is ( 0 2 4 ).
Vector v4 = min ( v1,v2 ) is ( 0 1 2 ).
Vector v5 = min ( v1,v3 ) is ( 0 1 2 ).
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

min_element

Finds the first occurrence of smallest element in a specified range where the ordering criterion may be specified by a binary predicate.

```
template<class ForwardIterator>
   ForwardIterator min_element(
      ForwardIterator _First,
      ForwardIterator _Last
   );
template<class ForwardIterator, class BinaryPredicate>
   ForwardIterator min_element(
      ForwardIterator _First,
      ForwardIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

_First

A forward iterator addressing the position of the first element in the range to be searched for the largest element.

_Last

A forward iterator addressing the position one past the final element in the range to be searched for the largest element.

_Comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

A forward iterator addressing the position of the first occurrence of the smallest element in the range searched.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within each sequence the last position is reachable from the first by incrementation.

The complexity is linear: (*_Last* – *_First*) – 1 comparisons are required for a nonempty range.

Example

```
// alg_min_element.cpp
// compile with: /EHsc
#include <vector>
#include <set>
#include <algorithm>
#include <iostream>
#include <ostream>

using namespace std;
class CInt;
ostream& operator<<( ostream& osIn, const CInt& rhs );

class CInt
{
public:
   CInt( int n = 0 ) : m_nVal( n ){}
   CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){}
   CInt& operator=( const CInt& rhs ) {m_nVal =
   rhs.m_nVal; return *this;}
   bool operator<( const CInt& rhs ) const
      {return ( m_nVal < rhs.m_nVal );}
```

```
   friend ostream& operator<<( ostream& osIn, const CInt& rhs );

private:
   int m_nVal;
};

inline ostream& operator<<( ostream& osIn, const CInt& rhs )
{
   osIn << "CInt( " << rhs.m_nVal << " )";
   return osIn;
}

// Return whether modulus of elem1 is greater than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
   if ( elem1 < 0 )
      elem1 = - elem1;
   if ( elem2 < 0 )
      elem2 = - elem2;
   return elem1 < elem2;
};

int main()
{
   // Searching a set container with elements of type CInt
   // for the minimum element
   CInt c1 = 1, c2 = 2, c3 = -3;
   set<CInt> s1;
   set<CInt>::iterator s1_Iter, s1_R1_Iter, s1_R2_Iter;

   s1.insert ( c1 );
   s1.insert ( c2 );
   s1.insert ( c3 );

   cout << "s1 = (";
   for ( s1_Iter = s1.begin( ); s1_Iter != --s1.end( ); s1_Iter++ )
      cout << " " << *s1_Iter << ",";
   s1_Iter = --s1.end( );
   cout << " " << *s1_Iter << " )." << endl;

   s1_R1_Iter = min_element ( s1.begin ( ) , s1.end ( ) );

   cout << "The smallest element in s1 is: " << *s1_R1_Iter << endl;
   cout << endl;

   // Searching a vector with elements of type int for the maximum
   // element under default less than & mod_lesser binary predicates
   vector <int> v1;
   vector <int>::iterator v1_Iter, v1_R1_Iter, v1_R2_Iter;

   int i;
   for ( i = 0 ; i <= 3 ; i++ )
   {
      v1.push_back( i );
   }

   int ii;
   for ( ii = 1 ; ii <= 4 ; ii++ )
   {
      v1.push_back( - 2 * ii );
   }

   cout << "Vector v1 is ( " ;
   for ( v1_Iter = v1.begin( ) ; v1_Iter != v1.end( ) ; v1_Iter++ )
      cout << *v1_Iter << " ";
   cout << "))." << endl;

   v1_R1_Iter = min_element ( v1.begin ( ) , v1.end ( ) );
```

```
    v1_R2_Iter = min_element ( v1.begin ( ) , v1.end ( ), mod_lesser);

    cout << "The smallest element in v1 is: " << *v1_R1_Iter << endl;
    cout << "The smallest element in v1 under the mod_lesser"
         << "\n binary predicate is: " << *v1_R2_Iter << endl;
  }
```

Output

```
s1 = ( CInt( -3 ), CInt( 1 ), CInt( 2 ) ).
The smallest element in s1 is: CInt( -3 )

Vector v1 is ( 0 1 2 3 -2 -4 -6 -8 ).
The smallest element in v1 is: -8
The smallest element in v1 under the mod_lesser
 binary predicate is: 0
```

Requirements

Header: <algorithm>

See Also

Reference

min_element (STL Samples)

Predicate Version of min_element

Standard Template Library

Other Resources

<algorithm> Members

mismatch

Compares two ranges element by element either for equality or equivalent in a sense specified by a binary predicate and locates the first position where a difference occurs.

```
template<class InputIterator1, class InputIterator2>
   pair<InputIterator1, InputIterator2> mismatch(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2
    );
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
   pair<InputIterator1, InputIterator2> mismatch(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2
      BinaryPredicate _Comp
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first range to be tested.

_Last1

An input iterator addressing the position one past the final element in the first range to be tested.

_First2

An input iterator addressing the position of the first element in the second range to be tested.

_Comp

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

A pair of iterators addressing the positions of the mismatch in the two ranges, the first component iterator to the position in the first range and the second component iterator to the position in the second range If there is no difference between the elements in the ranges compared or if the binary predicate in the second version is satisfied by all element pairs from the two ranges, then the first component iterator points to the position one past the final element in the first range and the second component iterator to position one past the final element tested in the second range.

Remarks

The range to be searched must be valid; all pointers must be dereferenceable and the last position is reachable from the first by incrementation.

The time complexity of the algorithm is linear in the number of elements contained in the range.

The **operator==** used to determine the equality between elements must impose an equivalence relation between its operands.

Example

```
// alg_mismatch.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>

// Return whether second element is twice the first
bool twice ( int elem1, int elem2 )
{
   return elem1 * 2 == elem2;
```

```
}

int main( )
{
   using namespace std;
   vector <int> v1, v2;
   list <int> L1;
   vector <int>::iterator Iter1, Iter2;
   list <int>::iterator L1_Iter, L1_inIter;

   int i;
   for ( i = 0 ; i <= 5 ; i++ )
   {
      v1.push_back( 5 * i );
   }

   int ii;
   for ( ii = 0 ; ii <= 7 ; ii++ )
   {
      L1.push_back( 5 * ii );
   }

   int iii;
   for ( iii = 0 ; iii <= 5 ; iii++ )
   {
      v2.push_back( 10 * iii );
   }

   cout << "Vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   cout << "List L1 = ( " ;
   for ( L1_Iter = L1.begin( ) ; L1_Iter!= L1.end( ) ; L1_Iter++ )
      cout << *L1_Iter << " ";
   cout << ")" << endl;

   cout << "Vector v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
      cout << ")" << endl;

   // Testing v1 and L1 for mismatch under identity
   pair<vector <int>::iterator, list <int>::iterator> results1;
   results1 = mismatch (v1.begin( ), v1.end( ), L1.begin( ));

   if ( results1.first == v1.end( ) )
      cout << "The two ranges do not differ."
           << endl;
   else
      cout << "The first mismatch is between "
           << *results1.first << " & " << *results1.second
           << endl;

   // Modifying L1
   L1_inIter = L1.begin( );
   L1_inIter++;
   L1_inIter++;
   L1.insert(L1_inIter, 100);
   cout << "Modified L1 = ( " ;
   for ( L1_Iter = L1.begin( ) ; L1_Iter!= L1.end( ) ; L1_Iter++ )
      cout << *L1_Iter << " ";
   cout << ")" << endl;

   // Testing v1 with modified L1 for mismatch under identity
   results1 = mismatch ( v1.begin( ), v1.end( ), L1.begin( ) );
```

```
    if ( results1.first == v1.end( ) )
       cout << "The two ranges do not differ."
            << endl;
    else
       cout << "The first mismatch is between "
            << *results1.first << " & " << *results1.second
            << endl;

    // Test v1 and v2 for mismatch under the binary predicate twice
    pair<vector <int>::iterator, vector <int>::iterator> results2;
    results2 = mismatch ( v1.begin( ), v1.end( ), v2.begin( ), twice );

    if ( results2.first == v1.end( ) )
       cout << "The two ranges do not differ under the binary "
            << "predicate twice." << endl;
    else
       cout << "The first mismatch is between "
            << *results2.first << " & " << *results2.second
            << endl;
 }
```

Output

```
Vector v1 = ( 0 5 10 15 20 25 )
List L1 = ( 0 5 10 15 20 25 30 35 )
Vector v2 = ( 0 10 20 30 40 50 )
The two ranges do not differ.
Modified L1 = ( 0 5 100 10 15 20 25 30 35 )
The first mismatch is between 10 & 100
The two ranges do not differ under the binary predicate twice.
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

next_permutation

Reorders the elements in a range so that the original ordering is replaced by the lexicographically next greater permutation if it exists, where the sense of next may be specified with a binary predicate.

```
template<class BidirectionalIterator>
   bool next_permutation(
      BidirectionalIterator _First,
      BidirectionalIterator _Last
   );
template<class BidirectionalIterator, class BinaryPredicate>
   bool next_permutation(
      BidirectionalIterator _First,
      BidirectionalIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

_First

A bidirectional iterator pointing to the position of the first element in the range to be permuted.

_Last

A bidirectional iterator pointing to the position one past the final element in the range to be permuted.

_Comp

User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

true if the lexicographically next permutation exists and has replaced the original ordering of the range; otherwise **false**, in which case the ordering is transformed into the lexicographically smallest permutation.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The default binary predicate is less than and the elements in the range must be less than comparable to insure that the next permutation is well defined.

The complexity is linear with at most (*_Last* – *_First*)/2 swaps.

Example

```
// alg_next_perm.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>
#include <ostream>

using namespace std;
class CInt;
ostream& operator<<( ostream& osIn, const CInt& rhs );

class CInt
{
public:
   CInt( int n = 0 ) : m_nVal( n ){}
   CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){}
   CInt&   operator=( const CInt& rhs ) {m_nVal =
   rhs.m_nVal; return *this;}
```

```
   bool operator<( const CInt& rhs ) const
      { return ( m_nVal < rhs.m_nVal );}
   friend   ostream& operator<<( ostream& osIn, const CInt& rhs );

private:
   int m_nVal;
};

inline ostream& operator<<( ostream& osIn, const CInt& rhs )
{
   osIn << "CInt( " << rhs.m_nVal << " )";
   return osIn;
}

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
   if ( elem1 < 0 )
      elem1 = - elem1;
   if ( elem2 < 0 )
      elem2 = - elem2;
   return elem1 < elem2;
};

int main( )
{
   // Reordering the elements of type CInt in a deque
   // using the prev_permutation algorithm
   CInt c1 = 5, c2 = 1, c3 = 10;
   bool deq1Result;
   deque<CInt> deq1, deq2, deq3;
   deque<CInt>::iterator d1_Iter;

   deq1.push_back ( c1 );
   deq1.push_back ( c2 );
   deq1.push_back ( c3 );

   cout << "The original deque of CInts is deq1 = (";
   for ( d1_Iter = deq1.begin( ); d1_Iter != --deq1.end( ); d1_Iter++ )
      cout << " " << *d1_Iter << ",";
   d1_Iter = --deq1.end( );
   cout << " " << *d1_Iter << " )." << endl;

   deq1Result = next_permutation ( deq1.begin ( ) , deq1.end ( ) );

   if ( deq1Result )
      cout << "The lexicographically next permutation "
           << "exists and has\nreplaced the original "
           << "ordering of the sequence in deq1." << endl;
   else
      cout << "The lexicographically next permutation doesn't "
           << "exist\n and the lexicographically "
           << "smallest permutation\n has replaced the "
           << "original ordering of the sequence in deq1." << endl;

   cout << "After one application of next_permutation,\n deq1 = (";
   for ( d1_Iter = deq1.begin( ); d1_Iter != --deq1.end( ); d1_Iter++ )
      cout << " " << *d1_Iter << ",";
   d1_Iter = --deq1.end( );
   cout << " " << *d1_Iter << " )." << endl << endl;

   // Permuting vector elements with binary function mod_lesser
   vector <int> v1;
   vector <int>::iterator Iter1;

   int i;
   for ( i = -3 ; i <= 3 ; i++ )
   {
```

```
      v1.push_back( i );
   }

   cout << "Vector v1 is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   next_permutation ( v1.begin ( ) , v1.end ( ) , mod_lesser );

   cout << "After the first next_permutation, vector v1 is:\n v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   int iii = 1;
   while ( iii <= 5 ) {
      next_permutation ( v1.begin ( ) , v1.end ( ) , mod_lesser );
      cout << "After another next_permutation of vector v1,\n v1 =   ( " ;
      for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ;Iter1 ++ )
         cout << *Iter1  << " ";
      cout << "))." << endl;
      iii++;
   }
}
```

Output

```
The original deque of CInts is deq1 = ( CInt( 5 ), CInt( 1 ), CInt( 10 ) ).
The lexicographically next permutation exists and has
replaced the original ordering of the sequence in deq1.
After one application of next_permutation,
 deq1 = ( CInt( 5 ), CInt( 10 ), CInt( 1 ) ).

Vector v1 is ( -3 -2 -1 0 1 2 3 ).
After the first next_permutation, vector v1 is:
 v1 = ( -3 -2 -1 0 1 3 2 ).
After another next_permutation of vector v1,
 v1 =   ( -3 -2 -1 0 2 1 3 ).
After another next_permutation of vector v1,
 v1 =   ( -3 -2 -1 0 2 3 1 ).
After another next_permutation of vector v1,
 v1 =   ( -3 -2 -1 0 3 1 2 ).
After another next_permutation of vector v1,
 v1 =   ( -3 -2 -1 0 3 2 1 ).
After another next_permutation of vector v1,
 v1 =   ( -3 -2 -1 1 0 2 3 ).
```

Requirements

Header: <algorithm>

See Also

Reference

next_permutation (STL Samples)

Predicate Version of next_permutation

Standard Template Library

Other Resources

<algorithm> Members

nth_element

Partitions a range of elements, correctly locating the *n*th element of the sequence in the range so that all the elements in front of it are less than or equal to it and all the elements that follow it in the sequence are greater than or equal to it.

```
template<class RandomAccessIterator>
   void nth_element(
      RandomAccessIterator _First,
      RandomAccessIterator _Nth,
      RandomAccessIterator _Last
   );
template<class RandomAccessIterator, class BinaryPredicate>
   void nth_element(
      RandomAccessIterator _First,
      RandomAccessIterator _Nth,
      RandomAccessIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

_First

A random-access iterator addressing the position of the first element in the range to be partitioned.

_Nth

A random-access iterator addressing the position of element to be correctly ordered on the boundary of the partition.

_Last

A random-access iterator addressing the position one past the final element in the range to be partitioned.

_Comp

User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The **nth_element** algorithm does not guarantee that elements in the sub-ranges either side of the *n*th element are sorted. It thus makes fewer guarantees than **partial_sort**, which orders the elements in the range below some chosen element, and may be used as a faster alternative to **partial_sort** when the ordering of the lower range is not required.

Elements are equivalent, but not necessarily equal, if neither is less than the other.

The average of a sort complexity is linear with respect to *_Last* – *_First*.

Example

```
// alg_nth_elem.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether first element is greater than the second
bool UDgreater ( int elem1, int elem2 ) {
   return elem1 > elem2;
}

int main() {
   using namespace std;
   vector <int> v1;
```

```
   vector <int>::iterator Iter1;

   int i;
   for ( i = 0 ; i <= 5 ; i++ )
      v1.push_back( 3 * i );

   int ii;
   for ( ii = 0 ; ii <= 5 ; ii++ )
      v1.push_back( 3 * ii + 1 );

   int iii;
   for ( iii = 0 ; iii <= 5 ; iii++ )
      v1.push_back( 3 * iii +2 );

   cout << "Original vector:\n v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   nth_element(v1.begin( ), v1.begin( ) + 3, v1.end( ) );
   cout << "Position 3 partitioned vector:\n v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // To sort in descending order, specify binary predicate
   nth_element( v1.begin( ), v1.begin( ) + 4, v1.end( ),
          greater<int>( ) );
   cout << "Position 4 partitioned (greater) vector:\n v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   random_shuffle( v1.begin( ), v1.end( ) );
   cout << "Shuffled vector:\n v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // A user-defined (UD) binary predicate can also be used
   nth_element( v1.begin( ), v1.begin( ) + 5, v1.end( ), UDgreater );
   cout << "Position 5 partitioned (UDgreater) vector:\n v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;
}
```

Sample Output

```
Original vector:
 v1 = ( 0 3 6 9 12 15 1 4 7 10 13 16 2 5 8 11 14 17 )
Position 3 partitioned vector:
 v1 = ( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 )
Position 4 partitioned (greater) vector:
 v1 = ( 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 )
Shuffled vector:
 v1 = ( 5 16 8 15 17 6 10 0 13 2 9 12 3 4 7 1 11 14 )
Position 5 partitioned (UDgreater) vector:
 v1 = ( 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 )
```

Requirements

Header: <algorithm>

See Also

Reference

nth_element (STL Samples)

Predicate Version of nth_element

partial_sort

Arranges a specified number of the smaller elements in a range into a nondescending order or according to an ordering criterion specified by a binary predicate.

```
template<class RandomAccessIterator>
   void partial_sort(
      RandomAccessIterator _First,
      RandomAccessIterator _SortEnd,
      RandomAccessIterator _Last
   );
template<class RandomAccessIterator, class BinaryPredicate>
   void partial_sort(
      RandomAccessIterator _First,
      RandomAccessIterator _SortEnd,
      RandomAccessIterator _Last
      BinaryPredicate _Comp
   );
```

Parameters

_First

A random-access iterator addressing the position of the first element in the range to be sorted.

_SortEnd

A random-access iterator addressing the position one past the final element in the subrange to be sorted.

_Last

A random-access iterator addressing the position one past the final element in the range to be partially sorted.

_Comp

User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Elements are equivalent, but not necessarily equal, if neither is less than the other. The **sort** algorithm is not stable and does not guarantee that the relative ordering of equivalent elements will be preserved. The algorithm **stable_sort** does preserve this original ordering.

The average of a sort complexity is $O(N \log N)$, where $N = _Last - _First$.

Example

```
// alg_partial_sort.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether first element is greater than the second
bool UDgreater ( int elem1, int elem2 )
{
   return elem1 > elem2;
}

int main( )
{
   using namespace std;
   vector <int> v1;
```

```
    vector <int>::iterator Iter1;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
       v1.push_back( 2 * i );
    }

    int ii;
    for ( ii = 0 ; ii <= 5 ; ii++ )
    {
       v1.push_back( 2 * ii +1 );
    }

    cout << "Original vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << ")" << endl;

    partial_sort(v1.begin( ), v1.begin( ) + 6, v1.end( ) );
    cout << "Partially sorted vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << ")" << endl;

    // To partially sort in descending order, specify binary predicate
    partial_sort(v1.begin( ), v1.begin( ) + 4, v1.end( ), greater<int>( ) );
    cout << "Partially resorted (greater) vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << ")" << endl;

    // A user-defined (UD) binary predicate can also be used
    partial_sort(v1.begin( ), v1.begin( ) + 8, v1.end( ),
  UDgreater );
    cout << "Partially resorted (UDgreater) vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << ")" << endl;
  }
```

Output

```
Original vector:
 v1 = ( 0 2 4 6 8 10 1 3 5 7 9 11 )
Partially sorted vector:
 v1 = ( 0 1 2 3 4 5 10 8 6 7 9 11 )
Partially resorted (greater) vector:
 v1 = ( 11 10 9 8 0 1 2 3 4 5 6 7 )
Partially resorted (UDgreater) vector:
 v1 = ( 11 10 9 8 7 6 5 4 0 1 2 3 )
```

Requirements

Header: <algorithm>

See Also

Reference

partial_sort (STL Samples)

Predicate Version of partial_sort

Standard Template Library

Other Resources

<algorithm> Members

partial_sort_copy

Copies elements from a source range into a destination range where the source elements are ordered by either less than or another specified binary predicate.

```
template<class InputIterator, class RandomAccessIterator>
   RandomAccessIterator partial_sort_copy(
      InputIterator _First1,
      InputIterator _Last1,
      RandomAccessIterator _First2,
      RandomAccessIterator _Last2
   );
template<class InputIterator, class RandomAccessIterator, class BinaryPredicate>
   RandomAccessIterator partial_sort_copy(
      InputIterator _First1,
      InputIterator _Last1,
      RandomAccessIterator _First2,
      RandomAccessIterator _Last2,
      BinaryPredicate _Comp
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the source range.

_Last1

An input iterator addressing the position one past the final element in the source range.

_First2

A random-access iterator addressing the position of the first element in the sorted destination range.

_Last2

A random-access iterator addressing the position one past the final element in the sorted destination range.

_Comp

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

A random-access iterator addressing the element in the destination range one position beyond the last element inserted from the source range.

Remarks

The source and destination ranges must not overlap and must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation.

The binary predicate must provide a strict weak ordering so that elements that are not equivalent are ordered, but elements that are equivalent are not. Two elements are equivalent under less than, but not necessarily equal, if neither is less than the other.

Example

```
// alg_partial_sort_copy.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <functional>
#include <iostream>
```

```
int main() {
    using namespace std;
    vector<int> v1, v2;
    list<int> list1;
    vector<int>::iterator iter1, iter2;
    list<int>::iterator list1_Iter, list1_inIter;

    int i;
    for (i = 0; i <= 9; i++)
        v1.push_back(i);

    random_shuffle(v1.begin(), v1.end());

    list1.push_back(60);
    list1.push_back(50);
    list1.push_back(20);
    list1.push_back(30);
    list1.push_back(40);
    list1.push_back(10);

    cout << "Vector v1 = ( " ;
    for (iter1 = v1.begin(); iter1 != v1.end(); iter1++)
        cout << *iter1 << " ";
    cout << ")" << endl;

    cout << "List list1 = ( " ;
    for (list1_Iter = list1.begin();
         list1_Iter!= list1.end();
         list1_Iter++)
        cout << *list1_Iter << " ";
    cout << ")" << endl;

    // Copying a partially sorted copy of list1 into v1
    vector<int>::iterator result1;
    result1 = partial_sort_copy(list1.begin(), list1.end(),
             v1.begin(), v1.begin() + 3);

    cout << "List list1 Vector v1 = ( " ;
    for (iter1 = v1.begin() ; iter1 != v1.end() ; iter1++)
        cout << *iter1 << " ";
    cout << ")" << endl;
    cout << "The first v1 element one position beyond"
         << "\n the last L 1 element inserted was " << *result1
         << "." << endl;

    // Copying a partially sorted copy of list1 into v2
    int ii;
    for (ii = 0; ii <= 9; ii++)
        v2.push_back(ii);

    random_shuffle(v2.begin(), v2.end());
    vector<int>::iterator result2;
    result2 = partial_sort_copy(list1.begin(), list1.end(),
             v2.begin(), v2.begin() + 6);

    cout << "List list1 into Vector v2 = ( " ;
    for (iter2 = v2.begin() ; iter2 != v2.end(); iter2++)
        cout << *iter2 << " ";
    cout << ")" << endl;
    cout << "The first v2 element one position beyond"
         << "\n the last L 1 element inserted was " << *result2
         << "." << endl;
}
```

Sample Output

```
Vector v1 = ( 8 1 9 2 0 5 7 3 4 6 )
List list1 = ( 60 50 20 30 40 10 )
```

```
List list1 Vector v1 = ( 10 20 30 2 0 5 7 3 4 6 )
The first v1 element one position beyond
 the last L 1 element inserted was 2.
List list1 into Vector v2 = ( 10 20 30 40 50 60 1 8 5 2 )
The first v2 element one position beyond
 the last L 1 element inserted was 1.
```

Requirements

Header: <algorithm>

See Also

Reference

partial_sort_copy (STL Samples)

Predicate Version of partial_sort_copy

Standard Template Library

Other Resources

<algorithm> Members

partition

Classifies elements in a range into two disjoint sets, with those elements satisfying a unary predicate preceding those that fail to satisfy it.

```
template<class BidirectionalIterator, class Predicate>
   BidirectionalIterator partition(
      BidirectionalIterator _First,
      BidirectionalIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

_First

A bidirectional iterator addressing the position of the first element in the range to be partitioned.

_Last

A bidirectional iterator addressing the position one past the final element in the range to be partitioned.

_Comp

User-defined predicate function object that defines the condition to be satisfied if an element is to be classified. A predicate takes a single argument and returns **true** or **false**.

Return Value

A bidirectional iterator addressing the position of the first element in the range to not satisfy the predicate condition.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Elements *a* and *b* are equivalent, but not necessarily equal, if both *Pr* (*a*, *b*) is false and *Pr* (*b*, *a*) if false, where *Pr* is the parameter-specified predicate. The **partition** algorithm is not stable and does not guarantee that the relative ordering of equivalent elements will be preserved. The algorithm **stable_ partition** does preserve this original ordering.

The complexity is linear: there are (*_Last* – *_First*) applications of *_Comp* and at most (*_Last* – *_First*)/2 swaps.

Example

```
// alg_partition.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

bool greater5 ( int value ) {
   return value >5;
}

int main( ) {
   using namespace std;
   vector <int> v1, v2;
   vector <int>::iterator Iter1, Iter2;

   int i;
   for ( i = 0 ; i <= 10 ; i++ )
   {
      v1.push_back( i );
   }
   random_shuffle( v1.begin( ), v1.end( ) );

   cout << "Vector v1 is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
```

```
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // Partition the range with predicate greater10
   partition ( v1.begin( ), v1.end( ), greater5 );
   cout << "The partitioned set of elements in v1 is: ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;
}
```

Sample Output

```
Vector v1 is ( 10 1 9 2 0 5 7 3 4 6 8 ).
The partitioned set of elements in v1 is: ( 10 8 9 6 7 5 0 3 4 2 1 ).
```

Requirements

Header: <algorithm>

See Also

Reference

partition (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

pop_heap

Removes the largest element from the front of a heap to the next-to-last position in the range and then forms a new heap from the remaining elements.

```
template<class RandomAccessIterator>
   void pop_heap(
      RandomAccessIterator _First,
      RandomAccessIterator _Last
   );
template<class RandomAccessIterator, class BinaryPredicate>
   void pop_heap(
      RandomAccessIterator _First,
      RandomAccessIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

_First

A random-access iterator addressing the position of the first element in the heap.

_Last

A random-access iterator addressing the position one past the final element in the heap.

_Comp

User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

The **pop_heap** algorithm is the inverse of the operation performed by the push_heap algorithm, in which an element at the next-to-last position of a range is added to a heap consisting of the prior elements in the range, in the case when the element being added to the heap is larger than any of the elements already in the heap.

Heaps have two properties:

- The first element is always the largest.
- Elements may be added or removed in logarithmic time.

Heaps are an ideal way to implement priority queues and they are used in the implementation of the Standard Template Library container adaptor priority_queue Class.

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The range excluding the newly added element at the end must be a heap.

The complexity is logarithmic, requiring at most log (*_Last* – *_First*) comparisons.

Example

```
// alg_pop_heap.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

int main( )  {
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1, Iter2;
```

```
   int i;
   for ( i = 1 ; i <= 9 ; i++ )
      v1.push_back( i );

   // Make v1 a heap with default less than ordering
   random_shuffle( v1.begin( ), v1.end( ) );
   make_heap ( v1.begin( ), v1.end( ) );
   cout << "The heaped version of vector v1 is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // Add an element to the back of the heap
   v1.push_back( 10 );
   push_heap( v1.begin( ), v1.end( ) );
   cout << "The reheaped v1 with 10 added is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // Remove the largest element from the heap
   pop_heap( v1.begin( ), v1.end( ) );
   cout << "The heap v1 with 10 removed is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl << endl;

   // Make v1 a heap with greater-than ordering with a 0 element
   make_heap ( v1.begin( ), v1.end( ), greater<int>( ) );
   v1.push_back( 0 );
   push_heap( v1.begin( ), v1.end( ), greater<int>( ) );
   cout << "The 'greater than' reheaped v1 puts the smallest "
        << "element first:\n ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // Application of pop_heap to remove the smallest element
   pop_heap( v1.begin( ), v1.end( ), greater<int>( ) );
   cout << "The 'greater than' heaped v1 with the smallest element\n "
        << "removed from the heap is: ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;
}
```

Sample Output

```
The heaped version of vector v1 is ( 9 5 8 4 1 6 7 2 3 ).
The reheaped v1 with 10 added is ( 10 9 8 4 5 6 7 2 3 1 ).
The heap v1 with 10 removed is ( 9 5 8 4 1 6 7 2 3 10 ).

The 'greater than' reheaped v1 puts the smallest element first:
 ( 0 1 6 3 2 8 7 4 9 10 5 ).
The 'greater than' heaped v1 with the smallest element
 removed from the heap is: ( 1 2 6 3 5 8 7 4 9 10 0 ).
```

Requirements

Header: <algorithm>

See Also

Reference

heap

Standard Template Library

Other Resources

<algorithm> Members

prev_permutation

Reorders the elements in a range so that the original ordering is replaced by the lexicographically previous greater permutation if it exists, where the sense of previous may be specified with a binary predicate.

```
template<class BidirectionalIterator>
   bool prev_permutation(
      BidirectionalIterator _First,
      BidirectionalIterator _Last
   );
template<class BidirectionalIterator, class BinaryPredicate>
   bool prev_permutation(
      BidirectionalIterator _First,
      BidirectionalIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

_First

A bidirectional iterator pointing to the position of the first element in the range to be permuted.

_Last

A bidirectional iterator pointing to the position one past the final element in the range to be permuted.

_Comp

User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

true if the lexicographically previous permutation exists and has replaced the original ordering of the range; otherwise **false**, in which case the ordering is transformed into the lexicographically largest permutation.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The default binary predicate is less than and the elements in the range must be less-than comparable to ensure that the previous permutation is well defined.

The complexity is linear, with at most (*_Last* – *_First*)/2 swaps.

Example

```
// alg_prev_perm.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>
#include <ostream>

using namespace std;
class CInt;
ostream& operator<<( ostream& osIn, const CInt& rhs );

class CInt {
public:
   CInt( int n = 0 ) : m_nVal( n ){}
   CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){}
   CInt&   operator=( const CInt& rhs ) {m_nVal =
   rhs.m_nVal; return *this;}
   bool operator<( const CInt& rhs ) const
```

```
      {return ( m_nVal < rhs.m_nVal );}
   friend ostream& operator<<( ostream& osIn, const CInt& rhs );

private:
   int m_nVal;
};

inline ostream& operator<<( ostream& osIn, const CInt& rhs ) {
   osIn << "CInt( " << rhs.m_nVal << " )";
   return osIn;
}

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser (int elem1, int elem2 ) {
   if ( elem1 < 0 )
      elem1 = - elem1;
   if ( elem2 < 0 )
      elem2 = - elem2;
   return elem1 < elem2;
};

int main() {
   // Reordering the elements of type CInt in a deque
   // using the prev_permutation algorithm
   CInt c1 = 1, c2 = 5, c3 = 10;
   bool deq1Result;
   deque<CInt> deq1, deq2, deq3;
   deque<CInt>::iterator d1_Iter;

   deq1.push_back ( c1 );
   deq1.push_back ( c2 );
   deq1.push_back ( c3 );

   cout << "The original deque of CInts is deq1 = (";
   for ( d1_Iter = deq1.begin( ); d1_Iter != --deq1.end( ); d1_Iter++ )
      cout << " " << *d1_Iter << ",";
   d1_Iter = --deq1.end( );
   cout << " " << *d1_Iter << " )." << endl;

   deq1Result = prev_permutation ( deq1.begin ( ) , deq1.end ( ) );

   if ( deq1Result )
      cout << "The lexicographically previous permutation "
           << "exists and has \nreplaced the original "
           << "ordering of the sequence in deq1." << endl;
   else
      cout << "The lexicographically previous permutation doesn't "
           << "exist\n and the lexicographically "
           << "smallest permutation\n has replaced the "
           << "original ordering of the sequence in deq1." << endl;

   cout << "After one application of prev_permutation,\n deq1 = (";
   for ( d1_Iter = deq1.begin( ); d1_Iter != --deq1.end( ); d1_Iter++ )
      cout << " " << *d1_Iter << ",";
   d1_Iter = --deq1.end( );
   cout << " " << *d1_Iter << " )." << endl << endl;

   // Permutating vector elements with binary function mod_lesser
   vector <int> v1;
   vector <int>::iterator Iter1;

   int i;
   for ( i = -3 ; i <= 3 ; i++ )
      v1.push_back( i );

   cout << "Vector v1 is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
```

```
    cout << "))." << endl;

    prev_permutation ( v1.begin ( ) , v1.end ( ) , mod_lesser );

    cout << "After the first prev_permutation, vector v1 is:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << "))." << endl;

    int iii = 1;
    while ( iii <= 5 ) {
       prev_permutation ( v1.begin ( ) , v1.end ( ) , mod_lesser );
       cout << "After another prev_permutation of vector v1,\n v1 =   ( " ;
       for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ;Iter1 ++ )
          cout << *Iter1  << " ";
       cout << "))." << endl;
       iii++;
    }
  }
```

Output

```
The original deque of CInts is deq1 = ( CInt( 1 ), CInt( 5 ), CInt( 10 ) ).
The lexicographically previous permutation doesn't exist
 and the lexicographically smallest permutation
 has replaced the original ordering of the sequence in deq1.
After one application of prev_permutation,
 deq1 = ( CInt( 10 ), CInt( 5 ), CInt( 1 ) ).

Vector v1 is ( -3 -2 -1 0 1 2 3 ).
After the first prev_permutation, vector v1 is:
 v1 = ( -3 -2 0 3 2 1 -1 ).
After another prev_permutation of vector v1,
 v1 =   ( -3 -2 0 3 -1 2 1 ).
After another prev_permutation of vector v1,
 v1 =   ( -3 -2 0 3 -1 1 2 ).
After another prev_permutation of vector v1,
 v1 =   ( -3 -2 0 2 3 1 -1 ).
After another prev_permutation of vector v1,
 v1 =   ( -3 -2 0 2 -1 3 1 ).
After another prev_permutation of vector v1,
 v1 =   ( -3 -2 0 2 -1 1 3 ).
```

Requirements

Header: <algorithm>

See Also

Reference

prev_permutation (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

push_heap

Adds an element that is at the end of a range to an existing heap consisting of the prior elements in the range.

```
template<class RandomAccessIterator>
   void push_heap(
      RandomAccessIterator _First,
      RandomAccessIterator _Last
   );
template<class RandomAccessIterator, class BinaryPredicate>
   void push_heap(
      RandomAccessIterator _First,
      RandomAccessIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

_First

A random-access iterator addressing the position of the first element in the heap.

_Last

A random-access iterator addressing the position one past the final element in the range to be converted into a heap.

_Comp

User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

The element must first be pushed back to the end of an existing heap and then the algorithm is used to add this element to the existing heap.

Heaps have two properties:

- The first element is always the largest.
- Elements may be added or removed in logarithmic time.

Heaps are an ideal way to implement priority queues and they are used in the implementation of the Standard Template Library container adaptor priority_queue Class.

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The range excluding the newly added element at the end must be a heap.

The complexity is logarithmic, requiring at most log (*_Last* – *_First*) comparisons.

Example

```
// alg_push_heap.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

int main( ) {
   using namespace std;
   vector <int> v1, v2;
   vector <int>::iterator Iter1, Iter2;

   int i;
   for ( i = 1 ; i <= 9 ; i++ )
```

```
      v1.push_back( i );

   random_shuffle( v1.begin( ), v1.end( ) );

   cout << "Vector v1 is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;

   // Make v1 a heap with default less than ordering
   make_heap ( v1.begin( ), v1.end( ) );
   cout << "The heaped version of vector v1 is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;

   // Add an element to the heap
   v1.push_back( 10 );
   cout << "The heap v1 with 10 pushed back is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;

   push_heap( v1.begin( ), v1.end( ) );
   cout << "The reheaped v1 with 10 added is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl << endl;

   // Make v1 a heap with greater than ordering
   make_heap ( v1.begin( ), v1.end( ), greater<int>( ) );
   cout << "The greater-than heaped version of v1 is\n ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;

   v1.push_back(0);
   cout << "The greater-than heap v1 with 11 pushed back is\n ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;

   push_heap( v1.begin( ), v1.end( ), greater<int>( ) );
   cout << "The greater than reheaped v1 with 11 added is\n ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;
}
```

Sample Output

```
Vector v1 is ( 9 2 7 3 1 6 8 4 5 ).
The heaped version of vector v1 is ( 9 5 8 4 1 6 7 2 3 ).
The heap v1 with 10 pushed back is ( 9 5 8 4 1 6 7 2 3 10 ).
The reheaped v1 with 10 added is ( 10 9 8 4 5 6 7 2 3 1 ).

The greater-than heaped version of v1 is
 ( 1 2 6 3 5 8 7 4 10 9 ).
The greater-than heap v1 with 11 pushed back is
 ( 1 2 6 3 5 8 7 4 10 9 0 ).
The greater than reheaped v1 with 11 added is
 ( 0 1 6 3 2 8 7 4 10 9 5 ).
```

Requirements

Header: <algorithm>

See Also

Reference

heap

Standard Template Library

Other Resources

<algorithm> Members

random_shuffle

Rearranges a sequence of *N* elements in a range into one of *N*! possible arrangements selected at random.

```
template<class RandomAccessIterator>
   void random_shuffle(
      RandomAccessIterator _First,
      RandomAccessIterator _Last
      );
template<class RandomAccessIterator, class RandomNumberGenerator>
   void random_shuffle(
      RandomAccessIterator _First,
      RandomAccessIterator _Last,
      RandomNumberGenerator& _Rand
   );
```

Parameters

_First

A random-access iterator addressing the position of the first element in the range to be rearranged.

_Last

A random-access iterator addressing the position one past the final element in the range to be rearranged.

_Rand

A special function object called a random number generator.

Remarks

The two versions of the function differ in how they generate random numbers. The first version uses an internal random number generator and the second a random number generator function object that is explicitly passed and can accept a seed value.

Example

```
// alg_random_shuffle.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

int main( ) {
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1, Iter2;

   int i;
   for ( i = 1 ; i <= 9 ; i++ )
      v1.push_back( i );

   random_shuffle( v1.begin( ), v1.end( ) );
   cout << "The original version of vector v1 is: ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // Shuffled once
   random_shuffle( v1.begin( ), v1.end( ));
   push_heap( v1.begin( ), v1.end( ) );
   cout << "Vector v1 after one shuffle is:       ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;
```

```
   // Shuffled again
   random_shuffle( v1.begin( ), v1.end( ));
   push_heap( v1.begin( ), v1.end( ) );
   cout << "Vector v1 after another shuffle is:   ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;
}
```

Sample Output

```
The original version of vector v1 is: ( 9 2 7 3 1 6 8 4 5 ).
Vector v1 after one shuffle is:       ( 1 4 7 9 2 5 8 6 3 ).
Vector v1 after another shuffle is:   ( 3 2 8 5 4 9 6 7 1 ).
```

Requirements

Header: <algorithm>

See Also

Reference

Predicate Version of random_shuffle

random_shuffle (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

remove

Eliminates a specified value from a given range without disturbing the order of the remaining elements and returning the end of a new range free of the specified value.

```
template<class ForwardIterator, class Type>
   ForwardIterator remove(
      ForwardIterator _First,
      ForwardIterator _Last,
      const Type& _Val
   );
```

Parameters

_First

A forward iterator addressing the position of the first element in the range from which elements are being removed.

_Last

A forward iterator addressing the position one past the final element in the range from which elements are being removed.

_Val

The value that is to be removed from the range.

Return Value

A forward iterator addressing the new end position of the modified range, one past the final element of the remnant sequence free of the specified value.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The order of the elements not removed remains stable.

The **operator==** used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear; there are (*_Last* – *_First*) comparisons for equality.

The list class has a more efficient member function version of **remove**, which also relinks pointers.

Example

```
// alg_remove.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main( ) {
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1, Iter2, new_end;

   int i;
   for ( i = 0 ; i <= 9 ; i++ )
      v1.push_back( i );

   int ii;
   for ( ii = 0 ; ii <= 3 ; ii++ )
      v1.push_back( 7 );

   random_shuffle ( v1.begin( ), v1.end( ) );
   cout << "Vector v1 is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
```

```
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // Remove elements with a value of 7
   new_end = remove ( v1.begin( ), v1.end( ), 7 );

   cout << "Vector v1 with value 7 removed is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // To change the sequence size, use erase
   v1.erase (new_end, v1.end( ) );

   cout << "Vector v1 resized with value 7 removed is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;
}
```

Sample Output

```
Vector v1 is ( 7 1 9 2 0 7 7 3 4 6 8 5 7 7 ).
Vector v1 with value 7 removed is ( 1 9 2 0 3 4 6 8 5 6 8 5 7 7 ).
Vector v1 resized with value 7 removed is ( 1 9 2 0 3 4 6 8 5 ).
```

Requirements

Header: <algorithm>

See Also

Reference

remove (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

remove_copy

Copies elements from a source range to a destination range, except that elements of a specified value are not copied, without disturbing the order of the remaining elements and returning the end of a new destination range.

```
template<class InputIterator, class OutputIterator, class Type>
   OutputIterator remove_copy(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      const Type& _Val
   );
```

Parameters

_First

An input iterator addressing the position of the first element in the range from which elements are being removed.

_Last

An input iterator addressing the position one past the final element in the range from which elements are being removed.

_Result

An output iterator addressing the position of the first element in the destination range to which elements are being removed.

_Val

The value that is to be removed from the range.

Return Value

A forward iterator addressing the new end position of the destination range, one past the final element of the copy of the remnant sequence free of the specified value.

Remarks

The source and destination ranges referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

There must be enough space in the destination range to contain the remnant elements that will be copied after elements of the specified value are removed.

The order of the elements not removed remains stable.

The **operator==** used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear; there are (*_Last* – *_First*) comparisons for equality and at most (*_Last* – *_First*) assignments.

remove_copy has two related forms:

- checked_remove_copy
- unchecked_remove_copy

For information on how these functions behave, see Checked Iterators.

Example

```
// alg_remove_copy.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
   using namespace std;
   vector <int> v1, v2(10);
```

```
   vector <int>::iterator Iter1, Iter2, new_end;

   int i;
   for ( i = 0 ; i <= 9 ; i++ )
      v1.push_back( i );

   int ii;
   for ( ii = 0 ; ii <= 3 ; ii++ )
      v1.push_back( 7 );

   random_shuffle (v1.begin( ), v1.end( ) );
   cout << "The original vector v1 is:     ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // Remove elements with a value of 7
   new_end = remove_copy ( v1.begin( ), v1.end( ), v2.begin( ), 7 );

   cout << "Vector v1 is left unchanged as ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   cout << "Vector v2 is a copy of v1 with the value 7 removed:\n ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << "))." << endl;
}
```

Sample Output

```
The original vector v1 is:     ( 7 1 9 2 0 7 7 3 4 6 8 5 7 7 ).
Vector v1 is left unchanged as ( 7 1 9 2 0 7 7 3 4 6 8 5 7 7 ).
Vector v2 is a copy of v1 with the value 7 removed:
 ( 1 9 2 0 3 4 6 8 5 0 ).
```

Requirements

Header: <algorithm>

See Also

Reference

remove_copy (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

remove_copy_if

Copies elements from a source range to a destination range, except that satisfying a predicate are not copied, without disturbing the order of the remaining elements and returning the end of a new destination range.

```
template<class InputIterator, class OutputIterator, class Predicate>
   OutputIterator remove_copy_if(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      Predicate _Pred
   );
```

Parameters

_First

An input iterator addressing the position of the first element in the range from which elements are being removed.

_Last

An input iterator addressing the position one past the final element in the range from which elements are being removed.

_Result

An output iterator addressing the position of the first element in the destination range to which elements are being removed.

_Pred

The unary predicate that must be satisfied is the value of an element is to be replaced.

Return Value

A forward iterator addressing the new end position of the destination range, one past the final element of the remnant sequence free of the elements satisfying the predicate.

Remarks

The source range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

There must be enough space in the destination range to contain the remnant elements that will be copied after elements of the specified value are removed.

The order of the elements not removed remains stable.

The **operator==** used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear: there are (*_Last* – *_First*) comparisons for equality and at most (*_Last* – *_First*) assignments.

remove_copy_if has two related forms:

- checked_remove_copy_if
- unchecked_remove_copy_if

For information on how these functions behave, see Checked Iterators.

Example

```
// alg_remove_copy_if.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

bool greater6 ( int value ) {
   return value >6;
}
```

```
int main() {
   using namespace std;
   vector <int> v1, v2(10);
   vector <int>::iterator Iter1, Iter2, new_end;

   int i;
   for ( i = 0 ; i <= 9 ; i++ )
      v1.push_back( i );

   int ii;
   for ( ii = 0 ; ii <= 3 ; ii++ )
      v1.push_back( 7 );

   random_shuffle ( v1.begin( ), v1.end( ) );
   cout << "The original vector v1 is:      ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // Remove elements with a value greater than 6
   new_end = remove_copy_if ( v1.begin( ), v1.end( ),
      v2.begin( ), greater6 );

   cout << "After the appliation of remove_copy_if to v1,\n "
        << "vector v1 is left unchanged as ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   cout << "Vector v2 is a copy of v1 with values greater "
        << "than 6 removed:\n ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != new_end ; Iter2++ )
      cout << *Iter2 << " ";
   cout << "))." << endl;
}
```

Sample Output

```
The original vector v1 is:      ( 7 1 9 2 0 7 7 3 4 6 8 5 7 7 ).
After the appliation of remove_copy_if to v1,
 vector v1 is left unchanged as ( 7 1 9 2 0 7 7 3 4 6 8 5 7 7 ).
Vector v2 is a copy of v1 with values greater than 6 removed:
 ( 1 2 0 3 4 6 5 ).
```

Requirements

Header: <algorithm>

See Also

Reference

remove_copy_if (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

remove_if

Eliminates elements that satisfy a predicate from a given range without disturbing the order of the remaining elements and returning the end of a new range free of the specified value.

```
template<class ForwardIterator, class Predicate>
   ForwardIterator remove_if(
      ForwardIterator _First,
      ForwardIterator _Last,
      Predicate _Pred
      );
```

Parameters

_First

A forward iterator pointing to the position of the first element in the range from which elements are being removed.

_Last

A forward iterator pointing to the position one past the final element in the range from which elements are being removed.

_Pred

The unary predicate that must be satisfied is the value of an element is to be replaced.

Return Value

A forward iterator addressing the new end position of the modified range, one past the final element of the remnant sequence free of the specified value.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The order of the elements not removed remains stable.

The **operator==** used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear: there are (*_Last* – *_First*) comparisons for equality.

List has a more efficient member function version of remove which relinks pointers.

Example

```
// alg_remove_if.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

bool greater6 ( int value ) {
   return value >6;
}

int main( ) {
   using namespace std;
   vector <int> v1, v2;
   vector <int>::iterator Iter1, Iter2, new_end;

   int i;
   for ( i = 0 ; i <= 9 ; i++ )
      v1.push_back( i );

   int ii;
   for ( ii = 0 ; ii <= 3 ; ii++ )
      v1.push_back( 7 );
```

```
   random_shuffle ( v1.begin( ), v1.end( ) );
   cout << "Vector v1 is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;

   // Remove elements satisfying predicate greater6
   new_end = remove_if (v1.begin( ), v1.end( ), greater6 );

   cout << "Vector v1 with elements satisfying greater6 removed is\n ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;

   // To change the sequence size, use erase
   v1.erase (new_end, v1.end( ) );

   cout << "Vector v1 resized elements satisfying greater6 removed is\n ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;
}
```

Sample Output

```
Vector v1 is ( 7 1 9 2 0 7 7 3 4 6 8 5 7 7 ).
Vector v1 with elements satisfying greater6 removed is
 ( 1 2 0 3 4 6 5 3 4 6 8 5 7 7 ).
Vector v1 resized elements satisfying greater6 removed is
 ( 1 2 0 3 4 6 5 ).
```

Requirements

Header: <algorithm>

See Also

Reference

remove_if (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

replace

Examines each element in a range and replaces it if it matches a specified value.

```
template<class ForwardIterator, class Type>
   void replace(
      ForwardIterator _First,
      ForwardIterator _Last,
      const Type& _OldVal,
      const Type& _NewVal
   );
```

Parameters

_First

A forward iterator pointing to the position of the first element in the range from which elements are being replaced.

_Last

A forward iterator pointing to the position one past the final element in the range from which elements are being replaced.

_OldVal

The old value of the elements being replaced.

_NewVal

The new value being assigned to the elements with the old value.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The order of the elements not replaced remains stable.

The **operator==** used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear; there are (*_Last* – *_First*) comparisons for equality and at most (*_Last* – *_First*) assignments of new values.

Example

```
// alg_replace.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main( ) {
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1;

   int i;
   for ( i = 0 ; i <= 9 ; i++ )
      v1.push_back( i );

   int ii;
   for ( ii = 0 ; ii <= 3 ; ii++ )
      v1.push_back( 7 );

   random_shuffle (v1.begin( ), v1.end( ) );
   cout << "The original vector v1 is:\n ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;
```

```
   // Replace elements with a value of 7 with a value of 700
   replace (v1.begin( ), v1.end( ), 7 , 700);

   cout << "The vector v1 with a value 700 replacing that of 7 is:\n ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;
}
```

Sample Output

```
The original vector v1 is:
 ( 7 1 9 2 0 7 7 3 4 6 8 5 7 7 ).
The vector v1 with a value 700 replacing that of 7 is:
 ( 700 1 9 2 0 700 700 3 4 6 8 5 700 700 ).
```

Requirements

Header: <algorithm>

See Also

Reference

replace (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

replace_copy

Examines each element in a source range and replaces it if it matches a specified value while copying the result into a new destination range.

```
template<class InputIterator, class OutputIterator, class Type>
   OutputIterator replace_copy(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      const Type& _OldVal,
      const Type& _NewVal
   );
```

Parameters

_First

An input iterator pointing to the position of the first element in the range from which elements are being replaced.

_Last

An input iterator pointing to the position one past the final element in the range from which elements are being replaced.

_Result

An output iterator pointing to the first element in the destination range to where the altered sequence of elements is being copied.

_OldVal

The old value of the elements being replaced.

_NewVal

The new value being assigned to the elements with the old value.

Return Value

An output iterator pointing to the position one past the final element in the destination range to where the altered sequence of elements is being copied.

Remarks

The source and destination ranges referenced must not overlap and must both be valid: all pointers must be dereferenceable and within the sequences the last position is reachable from the first by incrementation.

The order of the elements not replaced remains stable.

The **operator==** used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear: there are (*_Last* – *_First*) comparisons for equality and at most (*_Last* – *_First*) assignments of new values.

replace_copy has two related forms:

- checked_replace_copy
- unchecked_replace_copy

For information on how these functions behave, see Checked Iterators.

Example

```
// alg_replace_copy.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
```

```
#include <iostream>

int main( ) {
   using namespace std;
   vector <int> v1;
   list <int> L1 (15);
   vector <int>::iterator Iter1;
   list <int>::iterator L_Iter1;

   int i;
   for ( i = 0 ; i <= 9 ; i++ )
      v1.push_back( i );

   int ii;
   for ( ii = 0 ; ii <= 3 ; ii++ )
      v1.push_back( 7 );

   random_shuffle ( v1.begin( ), v1.end( ) );

   int iii;
   for ( iii = 0 ; iii <= 15 ; iii++ )
      v1.push_back( 1 );

   cout << "The original vector v1 is:\n ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // Replace elements in one part of a vector with a value of 7
   // with a value of 70 and copy into another part of the vector
   replace_copy ( v1.begin( ), v1.begin( ) + 14,v1.end( ) -15, 7 , 70);

   cout << "The vector v1 with a value 70 replacing that of 7 is:\n ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // Replace elements in a vector with a value of 70
   // with a value of 1 and copy into a list
   replace_copy ( v1.begin( ), v1.begin( ) + 14,L1.begin( ), 7 , 1);

   cout << "The list copy L1 of v1 with the value 0 replacing "
        << "that of 7 is:\n ( " ;
   for ( L_Iter1 = L1.begin( ) ; L_Iter1 != L1.end( ) ; L_Iter1++ )
      cout << *L_Iter1 << " ";
   cout << "))." << endl;
}
```

Sample Output

```
The original vector v1 is:
 ( 7 1 9 2 0 7 7 3 4 6 8 5 7 7 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ).
The vector v1 with a value 70 replacing that of 7 is:
 ( 7 1 9 2 0 7 7 3 4 6 8 5 7 7 1 70 1 9 2 0 70 70 3 4 6 8 5 70 70 1 ).
The list copy L1 of v1 with the value 0 replacing that of 7 is:
 ( 1 1 9 2 0 1 1 3 4 6 8 5 1 1 0 ).
```

Requirements

Header: <algorithm>

See Also

Reference

replace_copy (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

replace_copy_if

Examines each element in a source range and replaces it if it satisfies a specified predicate while copying the result into a new destination range.

```
template<class InputIterator, class OutputIterator, class Predicate, class Type>
   OutputIterator replace_copy_if(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      Predicate _Pred,
      const Type& _Val
   );
```

Parameters

_First

An input iterator pointing to the position of the first element in the range from which elements are being replaced.

_Last

An input iterator pointing to the position one past the final element in the range from which elements are being replaced.

_Result

An output iterator pointing to the position of the first element in the destination range to which elements are being copied.

_Pred

The unary predicate that must be satisfied is the value of an element is to be replaced.

_Val

The new value being assigned to the elements whose old value satisfies the predicate.

Return Value

An output iterator pointing to the position one past the final element in the destination range to where the altered sequence of elements is being copied.

Remarks

The source and destination ranges referenced must not overlap and must both be valid: all pointers must be dereferenceable and within the sequences the last position is reachable from the first by incrementation.

The order of the elements not replaced remains stable.

The **operator==** used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear; there are (*_Last* – *_First*) comparisons for equality and at most (*_Last* – *_First*) assignments of new values.

replace_copy_if has two related forms:

- checked_replace_copy_if
- unchecked_replace_copy_if

For information on how these functions behave, see Checked Iterators.

Example

```
// alg_replace_copy_if.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
```

```
bool greater6 ( int value ) {
   return value >6;
}

int main( ) {
   using namespace std;
   vector <int> v1;
   list <int> L1 (13);
   vector <int>::iterator Iter1;
   list <int>::iterator L_Iter1;

   int i;
   for ( i = 0 ; i <= 9 ; i++ )
      v1.push_back( i );

   int ii;
   for ( ii = 0 ; ii <= 3 ; ii++ )
      v1.push_back( 7 );

   random_shuffle ( v1.begin( ), v1.end( ) );

   int iii;
   for ( iii = 0 ; iii <= 13 ; iii++ )
      v1.push_back( 1 );

   cout << "The original vector v1 is:\n ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // Replace elements with a value of 7 in the 1st half of a vector
   // with a value of 70 and copy it into the 2nd half of the vector
   replace_copy_if ( v1.begin( ), v1.begin( ) + 14,v1.end( ) -14,
      greater6 , 70);

   cout << "The vector v1 with values of 70 replacing those greater"
        << "\n than 6 in the 1st half & copied into the 2nd half is:\n ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // Replace elements in a vector with a value of 70
   // with a value of 1 and copy into a list
   replace_copy_if ( v1.begin( ), v1.begin( ) + 13,L1.begin( ),
      greater6 , -1 );

   cout << "A list copy of vector v1 with the value -1\n replacing "
        << "those greater than 6 is:\n ( " ;
   for ( L_Iter1 = L1.begin( ) ; L_Iter1 != L1.end( ) ; L_Iter1++ )
      cout << *L_Iter1 << " ";
   cout << "))." << endl;
}
```

Sample Output

```
The original vector v1 is:
 ( 7 1 9 2 0 7 7 3 4 6 8 5 7 7 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ).
The vector v1 with values of 70 replacing those greater
 than 6 in the 1st half & copied into the 2nd half is:
 ( 7 1 9 2 0 7 7 3 4 6 8 5 7 7 70 1 70 2 0 70 70 3 4 6 70 5 70 70 ).
A list copy of vector v1 with the value -1
 replacing those greater than 6 is:
 ( -1 1 -1 2 0 -1 -1 3 4 6 -1 5 -1 ).
```

Requirements

Header: <algorithm>

See Also

Reference

replace_copy_if (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

replace_if

Examines each element in a range and replaces it if it satisfies a specified predicate.

```
template<class ForwardIterator, class Predicate, class Type>
   void replace_if(
      ForwardIterator _First,
      ForwardIterator _Last,
      Predicate _Pred,
      const Type& _Val
   );
```

Parameters

_First

A forward iterator pointing to the position of the first element in the range from which elements are being replaced.

_Last

An iterator pointing to the position one past the final element in the range from which elements are being replaced.

_Pred

The unary predicate that must be satisfied is the value of an element is to be replaced.

_Val

The new value being assigned to the elements whose old value satisfies the predicate.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The order of the elements not replaced remains stable.

The algorithm **replace_if** is a generalization of the algorithm **replace**, allowing any predicate to be specified, rather than equality to a specified constant value.

The **operator==** used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear: there are (*_Last* – *_First*) comparisons for equality and at most (*_Last* – *_First*) assignments of new values.

Example

```
// alg_replace_if.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

bool greater6 ( int value ) {
   return value >6;
}

int main( ) {
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1;

   int i;
   for ( i = 0 ; i <= 9 ; i++ )
      v1.push_back( i );

   int ii;
   for ( ii = 0 ; ii <= 3 ; ii++ )
```

```
      v1.push_back( 7 );

   random_shuffle ( v1.begin( ), v1.end( ) );
   cout << "The original vector v1 is:\n ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;

   // Replace elements satisfying the predicate greater6
   // with a value of 70
   replace_if ( v1.begin( ), v1.end( ), greater6 , 70);

   cout << "The vector v1 with a value 70 replacing those\n "
        << "elements satisfying the greater6 predicate is:\n ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;
}
```

Sample Output

```
The original vector v1 is:
 ( 7 1 9 2 0 7 7 3 4 6 8 5 7 7 ).
The vector v1 with a value 70 replacing those
 elements satisfying the greater6 predicate is:
 ( 70 1 70 2 0 70 70 3 4 6 70 5 70 70 ).
```

Requirements

Header: <algorithm>

See Also

Reference

replace_if (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

reverse

Reverses the order of the elements within a range.

```
template<class BidirectionalIterator>
   void reverse(
      BidirectionalIterator _First,
      BidirectionalIterator _Last
   );
```

Parameters

_First

A bidirectional iterator pointing to the position of the first element in the range within which the elements are being permuted.

_Last

A bidirectional iterator pointing to the position one past the final element in the range within which the elements are being permuted.

Remarks

The source range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Example

```
// alg_reverse.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main( ) {
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1;

   int i;
   for ( i = 0 ; i <= 9 ; i++ )
   {
      v1.push_back( i );
   }

   cout << "The original vector v1 is:\n ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // Reverse the elements in the vector
   reverse (v1.begin( ), v1.end( ) );

   cout << "The modified vector v1 with values reversed is:\n ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;
}
```

Output

```
The original vector v1 is:
 ( 0 1 2 3 4 5 6 7 8 9 ).
The modified vector v1 with values reversed is:
 ( 9 8 7 6 5 4 3 2 1 0 ).
```

Requirements

Header: <algorithm>

See Also

Reference

reverse (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

reverse_copy

Reverses the order of the elements within a source range while copying them into a destination range

```
template<class BidirectionalIterator, class OutputIterator>
   OutputIterator reverse_copy(
      BidirectionalIterator _First,
      BidirectionalIterator _Last,
      OutputIterator _Result
   );
```

Parameters

_First

A bidirectional iterator pointing to the position of the first element in the source range within which the elements are being permuted.

_Last

A bidirectional iterator pointing to the position one past the final element in the source range within which the elements are being permuted.

_Result

An output iterator pointing to the position of the first element in the destination range to which elements are being copied.

Return Value

An output iterator pointing to the position one past the final element in the destination range to where the altered sequence of elements is being copied.

Remarks

The source and destination ranges referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

reverse_copy has two related forms:

- checked_reverse_copy
- unchecked_reverse_copy

For information on how these functions behave, see Checked Iterators.

Example

```
// alg_reverse_copy.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main( ) {
   using namespace std;
   vector <int> v1, v2( 10 );
   vector <int>::iterator Iter1, Iter2;

   int i;
   for ( i = 0 ; i <= 9 ; i++ )
   {
      v1.push_back( i );
   }

   cout << "The original vector v1 is:\n ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;
```

```
    // Reverse the elements in the vector
    reverse_copy (v1.begin( ), v1.end( ), v2.begin( ) );

    cout << "The copy v2 of the reversed vector v1 is:\n ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
       cout << *Iter2 << " ";
    cout << ")." << endl;

    cout << "The original vector v1 remains unmodified as:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << ")." << endl;
 }
```

Output

```
The original vector v1 is:
 ( 0 1 2 3 4 5 6 7 8 9 ).
The copy v2 of the reversed vector v1 is:
 ( 9 8 7 6 5 4 3 2 1 0 ).
The original vector v1 remains unmodified as:
 ( 0 1 2 3 4 5 6 7 8 9 ).
```

Requirements

Header: <algorithm>

See Also

Reference

reverse_copy (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

rotate

Exchanges the elements in two adjacent ranges.

```
template<class ForwardIterator>
   void rotate(
      ForwardIterator _First,
      ForwardIterator _Middle,
      ForwardIterator _Last
   );
```

Parameters

_First

A forward iterator addressing the position of the first element in the range to be rotated.

_Middle

A forward iterator defining the boundary within the range that addresses the position of the first element in the second part of the range whose elements are to be exchanged with those in the first part of the range.

_ Last

A forward iterator addressing the position one past the final element in the range to be rotated.

Remarks

The ranges referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The complexity is linear with at most (*_Last* – *_First*) swaps.

Example

```
// alg_rotate.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>

int main( ) {
   using namespace std;
   vector <int> v1;
   deque <int> d1;
   vector <int>::iterator v1Iter1;
   deque<int>::iterator d1Iter1;

   int i;
   for ( i = -3 ; i <= 5 ; i++ )
   {
      v1.push_back( i );
   }

   int ii;
   for ( ii =0 ; ii <= 5 ; ii++ )
   {
      d1.push_back( ii );
   }

   cout << "Vector v1 is ( " ;
   for ( v1Iter1 = v1.begin( ) ; v1Iter1 != v1.end( ) ;v1Iter1 ++ )
      cout << *v1Iter1  << " ";
   cout << "))." << endl;

   rotate ( v1.begin ( ) , v1.begin ( ) + 3 , v1.end ( ) );
```

```
    cout << "After rotating, vector v1 is ( " ;
    for ( v1Iter1 = v1.begin( ) ; v1Iter1 != v1.end( ) ;v1Iter1 ++ )
       cout << *v1Iter1  << " ";
    cout << "))." << endl;

    cout << "The original deque d1 is ( " ;
    for ( d1Iter1 = d1.begin( ) ; d1Iter1 != d1.end( ) ;d1Iter1 ++ )
       cout << *d1Iter1  << " ";
    cout << "))." << endl;

    int iii = 1;
    while ( iii <= d1.end ( ) - d1.begin ( ) ) {
       rotate ( d1.begin ( ) , d1.begin ( ) + 1 , d1.end ( ) );
       cout << "After the rotation of a single deque element to the back,\n d1 is   ( " ;
       for ( d1Iter1 = d1.begin( ) ; d1Iter1 != d1.end( ) ;d1Iter1 ++ )
          cout << *d1Iter1  << " ";
       cout << "))." << endl;
       iii++;
    }
 }
```

Output

```
Vector v1 is ( -3 -2 -1 0 1 2 3 4 5 ).
After rotating, vector v1 is ( 0 1 2 3 4 5 -3 -2 -1 ).
The original deque d1 is ( 0 1 2 3 4 5 ).
After the rotation of a single deque element to the back,
 d1 is   ( 1 2 3 4 5 0 ).
After the rotation of a single deque element to the back,
 d1 is   ( 2 3 4 5 0 1 ).
After the rotation of a single deque element to the back,
 d1 is   ( 3 4 5 0 1 2 ).
After the rotation of a single deque element to the back,
 d1 is   ( 4 5 0 1 2 3 ).
After the rotation of a single deque element to the back,
 d1 is   ( 5 0 1 2 3 4 ).
After the rotation of a single deque element to the back,
 d1 is   ( 0 1 2 3 4 5 ).
```

Requirements

Header: <algorithm>

See Also

Reference

rotate (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

rotate_copy

Exchanges the elements in two adjacent ranges within a source range and copies the result to a destination range.

```
template<class ForwardIterator, class OutputIterator>
   OutputIterator rotate_copy(
      ForwardIterator _First,
      ForwardIterator _Middle,
      ForwardIterator _Last,
      OutputIterator _Result
   );
```

Parameters

_First

A forward iterator addressing the position of the first element in the range to be rotated.

_Middle

A forward iterator defining the boundary within the range that addresses the position of the first element in the second part of the range whose elements are to be exchanged with those in the first part of the range.

_ Last

A forward iterator addressing the position one past the final element in the range to be rotated.

_Result

An output iterator addressing the position of the first element in the destination range.

Return Value

An output iterator addressing the position one past the final element in the destination range.

Remarks

The ranges referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The complexity is linear with at most (*_Last* – *_First*) swaps.

rotate_copy has two related forms:

- checked_rotate_copy
- unchecked_rotate_copy

For information on how these functions behave, see Checked Iterators.

Example

```
// alg_rotate_copy.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>

int main() {
   using namespace std;
   vector <int> v1 , v2 ( 9 );
   deque <int> d1 , d2 ( 6 );
   vector <int>::iterator v1Iter , v2Iter;
   deque<int>::iterator d1Iter , d2Iter;

   int i;
   for ( i = -3 ; i <= 5 ; i++ )
      v1.push_back( i );
```

```
    int ii;
    for ( ii =0 ; ii <= 5 ; ii++ )
       d1.push_back( ii );

    cout << "Vector v1 is ( " ;
    for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ;v1Iter ++ )
       cout << *v1Iter  << " ";
    cout << "))." << endl;

    rotate_copy ( v1.begin ( ) , v1.begin ( ) + 3 , v1.end ( ) , v2.begin ( ) );
    cout << "After rotating, the vector v1 remains unchanged as:\n v1 = ( " ;
    for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ;v1Iter ++ )
       cout << *v1Iter  << " ";
    cout << "))." << endl;

    cout << "After rotating, the copy of vector v1 in v2 is:\n v2 = ( " ;
    for ( v2Iter = v2.begin( ) ; v2Iter != v2.end( ) ;v2Iter ++ )
       cout << *v2Iter  << " ";
    cout << "))." << endl;

    cout << "The original deque d1 is ( " ;
    for ( d1Iter = d1.begin( ) ; d1Iter != d1.end( ) ;d1Iter ++ )
       cout << *d1Iter  << " ";
    cout << "))." << endl;

    int iii = 1;
    while ( iii <= d1.end ( ) - d1.begin ( ) )
    {
       rotate_copy ( d1.begin ( ) , d1.begin ( ) + 1 , d1.end ( ) , d2.begin ( ) );
       cout << "After the rotation of a single deque element to the back,\n d2 is   ( " ;
       for ( d2Iter = d2.begin( ) ; d2Iter != d2.end( ) ;d2Iter ++ )
          cout << *d2Iter  << " ";
       cout << "))." << endl;
       iii++;
    }
  }
```

Output

```
Vector v1 is ( -3 -2 -1 0 1 2 3 4 5 ).
After rotating, the vector v1 remains unchanged as:
 v1 = ( -3 -2 -1 0 1 2 3 4 5 ).
After rotating, the copy of vector v1 in v2 is:
 v2 = ( 0 1 2 3 4 5 -3 -2 -1 ).
The original deque d1 is ( 0 1 2 3 4 5 ).
After the rotation of a single deque element to the back,
 d2 is   ( 1 2 3 4 5 0 ).
After the rotation of a single deque element to the back,
 d2 is   ( 1 2 3 4 5 0 ).
After the rotation of a single deque element to the back,
 d2 is   ( 1 2 3 4 5 0 ).
After the rotation of a single deque element to the back,
 d2 is   ( 1 2 3 4 5 0 ).
After the rotation of a single deque element to the back,
 d2 is   ( 1 2 3 4 5 0 ).
After the rotation of a single deque element to the back,
 d2 is   ( 1 2 3 4 5 0 ).
```

Requirements

Header: <algorithm>

See Also

Reference

rotate_copy (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

search

Searches for the first occurrence of a sequence within a target range whose elements are equal to those in a given sequence of elements or whose elements are equivalent in a sense specified by a binary predicate to the elements in the given sequence.

```
template<class ForwardIterator1, class ForwardIterator2>
   ForwardIterator1 search(
      ForwardIterator1 _First1,
      ForwardIterator1 _Last1,
      ForwardIterator2 _First2,
      ForwardIterator2 _Last2
   );
template<class ForwardIterator1, class ForwardIterator2, class Pr>
   ForwardIterator1 search(
      ForwardIterator1 _First1,
      ForwardIterator1 _Last1,
      ForwardIterator2 _First2,
      ForwardIterator2 _Last2
      BinaryPredicate _Comp
   );
```

Parameters

_First1

A forward iterator addressing the position of the first element in the range to be searched.

_Last1

A forward iterator addressing the position one past the final element in the range to be searched.

_First2

A forward iterator addressing the position of the first element in the range to be matched.

_Last2

A forward iterator addressing the position one past the final element in the range to be matched.

_Comp

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

A forward iterator addressing the position of the first element of the first subsequence that matches the specified sequence or that is equivalent in a sense specified by a binary predicate.

Remarks

The **operator==** used to determine the match between an element and the specified value must impose an equivalence relation between its operands.

The ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position is reachable from the first by incrementation.

Average complexity is linear with respect to the size of the searched range, and worst case complexity is also linear with respect to the size of the sequence being searched for.

Example

```
// alg_search.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
```

```
// Return whether second element is twice the first
bool twice (int elem1, int elem2 )
{
   return 2 * elem1 == elem2;
}

int main( ) {
   using namespace std;
   vector <int> v1, v2;
   list <int> L1;
   vector <int>::iterator Iter1, Iter2;
   list <int>::iterator L1_Iter, L1_inIter;

   int i;
   for ( i = 0 ; i <= 5 ; i++ )
   {
      v1.push_back( 5 * i );
   }
   for ( i = 0 ; i <= 5 ; i++ )
   {
      v1.push_back( 5 * i );
   }

   int ii;
   for ( ii = 4 ; ii <= 5 ; ii++ )
   {
      L1.push_back( 5 * ii );
   }

   int iii;
   for ( iii = 2 ; iii <= 4 ; iii++ )
   {
      v2.push_back( 10 * iii );
   }

   cout << "Vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   cout << "List L1 = ( " ;
   for ( L1_Iter = L1.begin( ) ; L1_Iter!= L1.end( ) ; L1_Iter++ )
      cout << *L1_Iter << " ";
   cout << ")" << endl;

   cout << "Vector v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
      cout << ")" << endl;

   // Searching v1 for first match to L1 under identity
   vector <int>::iterator result1;
   result1 = search (v1.begin( ), v1.end( ), L1.begin( ), L1.end( ) );

   if ( result1 == v1.end( ) )
      cout << "There is no match of L1 in v1."
           << endl;
   else
      cout << "There is at least one match of L1 in v1"
           << "\n and the first one begins at "
           << "position "<< result1 - v1.begin( ) << "." << endl;

   // Searching v1 for a match to L1 under the binary predicate twice
   vector <int>::iterator result2;
   result2 = search  (v1.begin( ), v1.end( ), v2.begin( ), v2.end( ), twice );

   if ( result2 == v1.end( ) )
```

```
        cout << "There is no match of L1 in v1."
             << endl;
    else
        cout << "There is a sequence of elements in v1 that "
             << "are equivalent\n to those in v2 under the binary "
             << "predicate twice\n and the first one begins at position "
             << result2 - v1.begin( ) << "." << endl;
  }
```

Output

```
Vector v1 = ( 0 5 10 15 20 25 0 5 10 15 20 25 )
List L1 = ( 20 25 )
Vector v2 = ( 20 30 40 )
There is at least one match of L1 in v1
 and the first one begins at position 4.
There is a sequence of elements in v1 that are equivalent
 to those in v2 under the binary predicate twice
 and the first one begins at position 2.
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

search_n

Searches for the first subsequence in a range that of a specified number of elements having a particular value or a relation to that value as specified by a binary predicate.

```
template<class ForwardIterator1, class Diff2, class Type>
   ForwardIterator1 search_n(
      ForwardIterator1 _First1,
      ForwardIterator1 _Last1,
      Size2 _Count,
      const Type& _Val
   );
template<class ForwardIterator1, class Size2, class Type, class BinaryPredicate>
   ForwardIterator1 search_n(
      ForwardIterator1 _First1,
      ForwardIterator1 _Last1,
      Size2 _Count,
      const Type& _Val,
      BinaryPredicate _Comp
   );
```

Parameters

_First1

A forward iterator addressing the position of the first element in the range to be searched.

_Last1

A forward iterator addressing the position one past the final element in the range to be searched.

_Count

The size of the subsequence being searched for.

_Val

The value of the elements in the sequence being searched for.

_Comp

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

A forward iterator addressing the position of the first element of the first subsequence that matches the specified sequence or that is equivalent in a sense specified by a binary predicate.

Remarks

The **operator==** used to determine the match between an element and the specified value must impose an equivalence relation between its operands.

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Complexity is linear with respect to the size of the searched.

Example

```
// alg_search_n.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>

// Return whether second element is twice the first
```

```
bool twice ( int elem1, int elem2 )
{
   return 2 * elem1 == elem2;
}

int main( )
{
   using namespace std;
   vector <int> v1, v2;
   vector <int>::iterator Iter1;

   int i;
   for ( i = 0 ; i <= 5 ; i++ )
   {
      v1.push_back( 5 * i );
   }

   for ( i = 0 ; i <= 2 ; i++ )
   {
      v1.push_back( 5  );
   }

   for ( i = 0 ; i <= 5 ; i++ )
   {
      v1.push_back( 5 * i );
   }

   for ( i = 0 ; i <= 2 ; i++ )
   {
      v1.push_back( 5  );
   }

   cout << "Vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // Searching v1 for first match to (5 5 5) under identity
   vector <int>::iterator result1;
   result1 = search_n ( v1.begin( ), v1.end( ), 3, 5 );

   if ( result1 == v1.end( ) )
      cout << "There is no match for a sequence ( 5 5 5 ) in v1."
           << endl;
   else
      cout << "There is at least one match of a sequence ( 5 5 5 )"
           << "\n in v1 and the first one begins at "
           << "position "<< result1 - v1.begin( ) << "." << endl;

   // Searching v1 for first match to (5 5 5) under twice
   vector <int>::iterator result2;
   result2 = search_n (v1.begin( ), v1.end( ), 3, 5 );

   if ( result2 == v1.end( ) )
      cout << "There is no match for a sequence ( 5 5 5 ) in v1"
           << " under the equivalence predicate twice." << endl;
   else
      cout << "There is a match of a sequence ( 5 5 5 ) "
           << "under the equivalence\n predicate twice"
           << "in v1 and the first one begins at "
           << "position "<< result1 - v1.begin( ) << "." << endl;
}
```

Output

```
Vector v1 = ( 0 5 10 15 20 25 5 5 5 0 5 10 15 20 25 5 5 5 )
There is at least one match of a sequence ( 5 5 5 )
 in v1 and the first one begins at position 6.
```

```
There is a match of a sequence ( 5 5 5 ) under the equivalence
 predicate twicein v1 and the first one begins at position 6.
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

set_difference

Unites all of the elements that belong to one sorted source range, but not to a second sorted source range, into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
   OutputIterator set_difference(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPred
icate>
   OutputIterator set_difference(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result,
      BinaryPredicate _Comp
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

_Last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

_First2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

_Last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

_ Result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the difference of the two source ranges.

_Comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range representing the difference of the two source ranges.

Remarks

The sorted source ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation.

The destination range should not overlap either of the source ranges and should be large enough to contain the destination

range.

The sorted source ranges must each be arranged as a precondition to the application of the **set_difference** algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The operation is stable as the relative order of elements within each range is preserved in the destination range. The source ranges are not modified by the algorithm merge.

The value types of the input iterators need be less-than-comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. When there are equivalent elements in both source ranges, the elements in the first range precede the elements from the second source range in the destination range. If the source ranges contain duplicates of an element such that there are more in the first source range than in the second, then the destination range will contain the number by which the occurrences of those elements in the first source range exceed the occurrences of those elements in the second source range.

The complexity of the algorithm is linear with at most 2 * ((*_Last1* – *_First1*) – (*_Last2* – *_First2*)) – 1 comparisons for nonempty source ranges.

set_difference has two related forms:

- checked_set_difference
- unchecked_set_difference

For information on how these functions behave, see Checked Iterators.

Example

```
// alg_set_diff.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser (int elem1, int elem2 )
{
   if (elem1 < 0)
      elem1 = - elem1;
   if (elem2 < 0)
      elem2 = - elem2;
   return elem1 < elem2;
}

int main( )
{
   using namespace std;
   vector <int> v1a, v1b, v1 ( 12 );
   vector <int>::iterator Iter1a,  Iter1b, Iter1, Result1;

   // Constructing vectors v1a & v1b with default less-than ordering
   int i;
   for ( i = -1 ; i <= 4 ; i++ )
   {
      v1a.push_back(  i );
   }

   int ii;
   for ( ii =-3 ; ii <= 0 ; ii++ )
   {
      v1b.push_back(  ii  );
   }

   cout << "Original vector v1a with range sorted by the\n "
        <<  "binary predicate less than is  v1a = ( " ;
   for ( Iter1a = v1a.begin( ) ; Iter1a != v1a.end( ) ; Iter1a++ )
      cout << *Iter1a << " ";
```

```
cout << ")." << endl;

cout << "Original vector v1b with range sorted by the\n "
     <<  "binary predicate less than is  v1b = ( " ;
for ( Iter1b = v1b.begin ( ) ; Iter1b != v1b.end ( ) ; Iter1b++ )
   cout << *Iter1b << " ";
cout << ")." << endl;

// Constructing vectors v2a & v2b with ranges sorted by greater
vector <int> v2a ( v1a ) , v2b ( v1b ) ,  v2 ( v1 );
vector <int>::iterator Iter2a, Iter2b, Iter2, Result2;
sort ( v2a.begin ( ) , v2a.end ( ) , greater<int> ( ) );
sort ( v2b.begin ( ) , v2b.end ( ) , greater<int> ( ) );

cout << "Original vector v2a with range sorted by the\n "
     <<  "binary predicate greater is   v2a =  ( " ;
for ( Iter2a = v2a.begin ( ) ; Iter2a != v2a.end ( ) ; Iter2a++ )
   cout << *Iter2a << " ";
cout << ")." << endl;

cout << "Original vector v2b with range sorted by the\n "
     <<  "binary predicate greater is   v2b =  ( " ;
for ( Iter2b = v2b.begin ( ) ; Iter2b != v2b.end ( ) ; Iter2b++ )
   cout << *Iter2b << " ";
cout << ")." << endl;

// Constructing vectors v3a & v3b with ranges sorted by mod_lesser
vector <int> v3a ( v1a ), v3b ( v1b ) ,  v3 ( v1 );
vector <int>::iterator Iter3a,  Iter3b, Iter3, Result3;
sort ( v3a.begin ( ) , v3a.end ( ) , mod_lesser );
sort ( v3b.begin ( ) , v3b.end ( ) , mod_lesser  );

cout << "Original vector v3a with range sorted by the\n "
     <<  "binary predicate mod_lesser is   v3a =  ( " ;
for ( Iter3a = v3a.begin ( ) ; Iter3a != v3a.end ( ) ; Iter3a++ )
   cout << *Iter3a << " ";
cout << ")." << endl;

cout << "Original vector v3b with range sorted by the\n "
     <<  "binary predicate mod_lesser is   v3b =  ( " ;
for ( Iter3b = v3b.begin ( ) ; Iter3b != v3b.end ( ) ; Iter3b++ )
   cout << *Iter3b << " ";
cout << ")." << endl;

// To combine into a difference in asscending
// order with the default binary predicate less <int> ( )
Result1 = set_difference ( v1a.begin ( ) , v1a.end ( ) ,
   v1b.begin ( ) , v1b.end ( ) , v1.begin ( ) );
cout << "Set_difference of source ranges with default order,"
     << "\n vector v1mod =  ( " ;
for ( Iter1 = v1.begin( ) ; Iter1 != Result1 ; Iter1++ )
   cout << *Iter1 << " ";
cout << ")." << endl;

// To combine into a difference in descending
// order specify binary predicate greater<int>( )
Result2 = set_difference ( v2a.begin ( ) , v2a.end ( ) ,
   v2b.begin ( ) , v2b.end ( ) ,v2.begin ( ) , greater <int> ( ) );
cout << "Set_difference of source ranges with binary"
     << "predicate greater specified,\n vector v2mod  = ( " ;
for ( Iter2 = v2.begin( ) ; Iter2 != Result2 ; Iter2++ )
   cout << *Iter2 << " ";
cout << ")." << endl;

// To combine into a difference applying a user
// defined binary predicate mod_lesser
Result3 = set_difference (  v3a.begin ( ) , v3a.end ( ) ,
   v3b.begin ( ) , v3b.end ( ) , v3.begin ( ) , mod_lesser );
```

```
    cout << "Set_difference of source ranges with binary "
         << "predicate mod_lesser specified,\n vector v3mod  = ( " ; ;
    for ( Iter3 = v3.begin( ) ; Iter3 != Result3 ; Iter3++ )
       cout << *Iter3 << " ";
    cout << "))." << endl;
  }
```

Output

```
Original vector v1a with range sorted by the
 binary predicate less than is  v1a = ( -1 0 1 2 3 4 ).
Original vector v1b with range sorted by the
 binary predicate less than is  v1b = ( -3 -2 -1 0 ).
Original vector v2a with range sorted by the
 binary predicate greater is   v2a =  ( 4 3 2 1 0 -1 ).
Original vector v2b with range sorted by the
 binary predicate greater is   v2b =  ( 0 -1 -2 -3 ).
Original vector v3a with range sorted by the
 binary predicate mod_lesser is   v3a =  ( 0 -1 1 2 3 4 ).
Original vector v3b with range sorted by the
 binary predicate mod_lesser is   v3b =  ( 0 -1 -2 -3 ).
Set_difference of source ranges with default order,
 vector v1mod =  ( 1 2 3 4 ).
Set_difference of source ranges with binarypredicate greater specified,
 vector v2mod  = ( 4 3 2 1 ).
Set_difference of source ranges with binary predicate mod_lesser specified,
 vector v3mod  = ( 1 4 ).
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

set_intersection

Unites all of the elements that belong to both sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
   OutputIterator set_intersection(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPred
icate>
   OutputIterator set_intersection(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result,
      BinaryPredicate _Comp
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

_Last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

_First2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

_Last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

_ Result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the intersection of the two source ranges.

_Comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range representing the intersection of the two source ranges.

Remarks

The sorted source ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation.

The destination range should not overlap either of the source ranges and should be large enough to contain the destination

range.

The sorted source ranges must each be arranged as a precondition to the application of the merge algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The operation is stable as the relative order of elements within each range is preserved in the destination range. The source ranges are not modified by the algorithm.

The value types of the input iterators need be less-than comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. When there are equivalent elements in both source ranges, the elements in the first range precede the elements from the second source range in the destination range. If the source ranges contain duplicates of an element, then the destination range will contain the maximum number of those elements that occur in both source ranges.

The complexity of the algorithm is linear with at most 2 * ((*_Last1* – *_First1*) + (*_Last2* – *_First2*)) – 1 comparisons for nonempty source ranges.

set_intersection has two related forms:

- checked_set_intersection
- unchecked_set_intersection

For information on how these functions behave, see Checked Iterators.

Example

```
// alg_set_intersection.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>   // For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser (int elem1, int elem2 ) {
   if ( elem1 < 0 )
      elem1 = - elem1;
   if ( elem2 < 0 )
      elem2 = - elem2;
   return elem1 < elem2;
}

int main() {
   using namespace std;
   vector <int> v1a, v1b, v1 ( 12 );
   vector <int>::iterator Iter1a,  Iter1b, Iter1, Result1;

   // Constructing vectors v1a & v1b with default less than ordering
   int i;
   for ( i = -1 ; i <= 3 ; i++ )
      v1a.push_back( i );

   int ii;
   for ( ii =-3 ; ii <= 1 ; ii++ )
      v1b.push_back( ii );

   cout << "Original vector v1a with range sorted by the\n "
        <<  "binary predicate less than is  v1a = ( " ;
   for ( Iter1a = v1a.begin( ) ; Iter1a != v1a.end( ) ; Iter1a++ )
      cout << *Iter1a << " ";
   cout << "))." << endl;

   cout << "Original vector v1b with range sorted by the\n "
        <<  "binary predicate less than is  v1b = ( " ;
   for ( Iter1b = v1b.begin ( ) ; Iter1b != v1b.end ( ) ; Iter1b++ )
      cout << *Iter1b << " ";
   cout << "))." << endl;
```

```
    // Constructing vectors v2a & v2b with ranges sorted by greater
    vector <int> v2a ( v1a ) , v2b ( v1b ) , v2 ( v1 );
    vector <int>::iterator Iter2a, Iter2b, Iter2, Result2;
    sort ( v2a.begin ( ) , v2a.end ( ) , greater<int> ( ) );
    sort ( v2b.begin ( ) , v2b.end ( ) , greater<int> ( ) );

    cout << "Original vector v2a with range sorted by the\n "
         << "binary predicate greater is   v2a =  ( " ;
    for ( Iter2a = v2a.begin ( ) ; Iter2a != v2a.end ( ) ; Iter2a++ )
       cout << *Iter2a << " ";
    cout << ")." << endl;

    cout << "Original vector v2b with range sorted by the\n "
         << "binary predicate greater is   v2b =  ( " ;
    for ( Iter2b = v2b.begin ( ) ; Iter2b != v2b.end ( ) ; Iter2b++ )
       cout << *Iter2b << " ";
    cout << ")." << endl;

    // Constructing vectors v3a & v3b with ranges sorted by mod_lesser
    vector <int> v3a ( v1a ), v3b ( v1b ) , v3 ( v1 );
    vector <int>::iterator Iter3a,  Iter3b, Iter3, Result3;
    sort ( v3a.begin ( ) , v3a.end ( ) , mod_lesser );
    sort ( v3b.begin ( ) , v3b.end ( ) , mod_lesser );

    cout << "Original vector v3a with range sorted by the\n "
         <<  "binary predicate mod_lesser is   v3a =  ( " ;
    for ( Iter3a = v3a.begin ( ) ; Iter3a != v3a.end ( ) ; Iter3a++ )
       cout << *Iter3a << " ";
    cout << ")." << endl;

    cout << "Original vector v3b with range sorted by the\n "
            <<  "binary predicate mod_lesser is   v3b =  ( " ;
    for ( Iter3b = v3b.begin ( ) ; Iter3b != v3b.end ( ) ; Iter3b++ )
       cout << *Iter3b << " ";
    cout << ")." << endl;

    // To combine into an intersection in asscending order with the
    // default binary predicate less <int> ( )
    Result1 = set_intersection ( v1a.begin ( ) , v1a.end ( ) ,
       v1b.begin ( ) , v1b.end ( ) , v1.begin ( ) );
    cout << "Intersection of source ranges with default order,"
         << "\n vector v1mod =  ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != Result1 ; ++Iter1 )
       cout << *Iter1 << " ";
    cout << ")." << endl;

    // To combine into an intersection in descending order, specify
    // binary predicate greater<int>( )
    Result2 = set_intersection ( v2a.begin ( ) , v2a.end ( ) ,
       v2b.begin ( ) , v2b.end ( ) ,v2.begin ( ) , greater <int> ( ) );
    cout << "Intersection of source ranges with binary predicate"
         << " greater specified,\n vector v2mod  = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != Result2 ; ++Iter2 )
       cout << *Iter2 << " ";
    cout << ")." << endl;

    // To combine into an intersection applying a user-defined
    // binary predicate mod_lesser
    Result3 = set_intersection ( v3a.begin ( ) , v3a.end ( ) ,
       v3b.begin ( ) , v3b.end ( ) , v3.begin ( ) , mod_lesser );
    cout << "Intersection of source ranges with binary predicate "
         << "mod_lesser specified,\n vector v3mod  = ( " ; ;
    for ( Iter3 = v3.begin( ) ; Iter3 != Result3 ; ++Iter3 )
       cout << *Iter3 << " ";
    cout << ")." << endl;
 }
```

Output

```
Original vector v1a with range sorted by the
 binary predicate less than is  v1a = ( -1 0 1 2 3 ).
Original vector v1b with range sorted by the
 binary predicate less than is  v1b = ( -3 -2 -1 0 1 ).
Original vector v2a with range sorted by the
 binary predicate greater is   v2a =  ( 3 2 1 0 -1 ).
Original vector v2b with range sorted by the
 binary predicate greater is   v2b =  ( 1 0 -1 -2 -3 ).
Original vector v3a with range sorted by the
 binary predicate mod_lesser is   v3a =  ( 0 -1 1 2 3 ).
Original vector v3b with range sorted by the
 binary predicate mod_lesser is   v3b =  ( 0 -1 1 -2 -3 ).
Intersection of source ranges with default order,
 vector v1mod =  ( -1 0 1 ).
Intersection of source ranges with binary predicate greater specified,
 vector v2mod  = ( 1 0 -1 ).
Intersection of source ranges with binary predicate mod_lesser specified,
 vector v3mod  = ( 0 -1 1 2 3 ).
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

set_symmetric_difference

Unites all of the elements that belong to one, but not both, of the sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
   OutputIterator set_symmetric_difference(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPred
icate>
   OutputIterator set_symmetric_difference(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result,
      BinaryPredicate _Comp
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

_Last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

_First2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

_Last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

_ Result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the symmetric difference of the two source ranges.

_Comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range representing the symmetric difference of the two source ranges.

Remarks

The sorted source ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation.

The destination range should not overlap either of the source ranges and should be large enough to contain the destination

range.

The sorted source ranges must each be arranged as a precondition to the application of the **merge** algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The operation is stable as the relative order of elements within each range is preserved in the destination range. The source ranges are not modified by the algorithm merge.

The value types of the input iterators need be less-than comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. When there are equivalent elements in both source ranges, the elements in the first range precede the elements from the second source range in the destination range. If the source ranges contain duplicates of an element, then the destination range will contain the absolute value of the number by which the occurrences of those elements in the one of the source ranges exceeds the occurrences of those elements in the second source range.

The complexity of the algorithm is linear with at most 2 * ((*_Last1* – *_First1*) – (*_Last2* – *_First2*)) – 1 comparisons for nonempty source ranges.

set_symmetric_difference has two related forms:

- checked_set_symmetric_difference
- unchecked_set_symmetric_difference

For information on how these functions behave, see Checked Iterators.

Example

```
// alg_set_sym_diff.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser (int elem1, int elem2 )
{
   if ( elem1 < 0 )
      elem1 = - elem1;
   if ( elem2 < 0 )
      elem2 = - elem2;
   return elem1 < elem2;
}

int main( )
{
   using namespace std;
   vector <int> v1a, v1b, v1 ( 12 );
   vector <int>::iterator Iter1a,  Iter1b, Iter1, Result1;

   // Constructing vectors v1a & v1b with default less-than ordering
   int i;
   for ( i = -1 ; i <= 4 ; i++ )
   {
      v1a.push_back(  i );
   }

   int ii;
   for ( ii =-3 ; ii <= 0 ; ii++ )
   {
      v1b.push_back(  ii  );
   }

   cout << "Original vector v1a with range sorted by the\n "
        <<  "binary predicate less than is  v1a = ( " ;
   for ( Iter1a = v1a.begin( ) ; Iter1a != v1a.end( ) ; Iter1a++ )
      cout << *Iter1a << " ";
```

```
cout << ")." << endl;

cout << "Original vector v1b with range sorted by the\n "
     <<  "binary predicate less than is  v1b = ( " ;
for ( Iter1b = v1b.begin ( ) ; Iter1b != v1b.end ( ) ; Iter1b++ )
   cout << *Iter1b << " ";
cout << ")." << endl;

// Constructing vectors v2a & v2b with ranges sorted by greater
vector <int> v2a ( v1a ) , v2b ( v1b ) ,  v2 ( v1 );
vector <int>::iterator Iter2a, Iter2b, Iter2, Result2;
sort ( v2a.begin ( ) , v2a.end ( ) , greater<int> ( ) );
sort ( v2b.begin ( ) , v2b.end ( ) , greater<int> ( ) );

cout << "Original vector v2a with range sorted by the\n "
     <<  "binary predicate greater is   v2a =  ( " ;
for ( Iter2a = v2a.begin ( ) ; Iter2a != v2a.end ( ) ; Iter2a++ )
   cout << *Iter2a << " ";
cout << ")." << endl;

cout << "Original vector v2b with range sorted by the\n "
     <<  "binary predicate greater is   v2b =  ( " ;
for ( Iter2b = v2b.begin ( ) ; Iter2b != v2b.end ( ) ; Iter2b++ )
   cout << *Iter2b << " ";
cout << ")." << endl;

// Constructing vectors v3a & v3b with ranges sorted by mod_lesser
vector <int> v3a ( v1a ), v3b ( v1b ) ,  v3 ( v1 );
vector <int>::iterator Iter3a, Iter3b, Iter3, Result3;
sort ( v3a.begin ( ) , v3a.end ( ) , mod_lesser );
sort ( v3b.begin ( ) , v3b.end ( ) , mod_lesser  );

cout << "Original vector v3a with range sorted by the\n "
     <<  "binary predicate mod_lesser is   v3a =  ( " ;
for ( Iter3a = v3a.begin ( ) ; Iter3a != v3a.end ( ) ; Iter3a++ )
   cout << *Iter3a << " ";
cout << ")." << endl;

cout << "Original vector v3b with range sorted by the\n "
     <<  "binary predicate mod_lesser is   v3b =  ( " ;
for ( Iter3b = v3b.begin ( ) ; Iter3b != v3b.end ( ) ; Iter3b++ )
   cout << *Iter3b << " ";
cout << ")." << endl;

// To combine into a symmetric difference in ascending
// order with the default binary predicate less <int> ( )
Result1 = set_symmetric_difference ( v1a.begin ( ) , v1a.end ( ) ,
   v1b.begin ( ) , v1b.end ( ) , v1.begin ( ) );
cout << "Set_symmetric_difference of source ranges with default order,"
     << "\n vector v1mod =  ( " ;
for ( Iter1 = v1.begin( ) ; Iter1 != Result1 ; Iter1++ )
   cout << *Iter1 << " ";
cout << ")." << endl;

// To combine into a symmetric difference in descending
// order, specify binary predicate greater<int>( )
Result2 = set_symmetric_difference ( v2a.begin ( ) , v2a.end ( ) ,
   v2b.begin ( ) , v2b.end ( ) ,v2.begin ( ) , greater <int> ( ) );
cout << "Set_symmetric_difference of source ranges with binary"
     << "predicate greater specified,\n vector v2mod  = ( " ;
for ( Iter2 = v2.begin( ) ; Iter2 != Result2 ; Iter2++ )
   cout << *Iter2 << " ";
cout << ")." << endl;

// To combine into a symmetric difference applying a user
// defined binary predicate mod_lesser
Result3 = set_symmetric_difference ( v3a.begin ( ) , v3a.end ( ) ,
   v3b.begin ( ) , v3b.end ( ) , v3.begin ( ) , mod_lesser );
```

```
    cout << "Set_symmetric_difference of source ranges with binary "
         << "predicate mod_lesser specified,\n vector v3mod  = ( " ; ;
    for ( Iter3 = v3.begin( ) ; Iter3 != Result3 ; Iter3++ )
       cout << *Iter3 << " ";
    cout << "))." << endl;
  }
```

Output

```
Original vector v1a with range sorted by the
 binary predicate less than is  v1a = ( -1 0 1 2 3 4 ).
Original vector v1b with range sorted by the
 binary predicate less than is  v1b = ( -3 -2 -1 0 ).
Original vector v2a with range sorted by the
 binary predicate greater is   v2a =  ( 4 3 2 1 0 -1 ).
Original vector v2b with range sorted by the
 binary predicate greater is   v2b =  ( 0 -1 -2 -3 ).
Original vector v3a with range sorted by the
 binary predicate mod_lesser is   v3a =  ( 0 -1 1 2 3 4 ).
Original vector v3b with range sorted by the
 binary predicate mod_lesser is   v3b =  ( 0 -1 -2 -3 ).
Set_symmetric_difference of source ranges with default order,
 vector v1mod =  ( -3 -2 1 2 3 4 ).
Set_symmetric_difference of source ranges with binarypredicate greater specified,
 vector v2mod  = ( 4 3 2 1 -2 -3 ).
Set_symmetric_difference of source ranges with binary predicate mod_lesser specified,
 vector v3mod  = ( 1 4 ).
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

set_union

Unites all of the elements that belong to at least one of two sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
   OutputIterator set_union(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPred
icate>
   OutputIterator set_union(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result,
      BinaryPredicate _Comp
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

_Last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

_First2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

_Last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

_ Result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the union of the two source ranges.

_Comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range representing the union of the two source ranges.

Remarks

The sorted source ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation.

The destination range should not overlap either of the source ranges and should be large enough to contain the destination

range.

The sorted source ranges must each be arranged as a precondition to the application of the **merge** algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The operation is stable as the relative order of elements within each range is preserved in the destination range. The source ranges are not modified by the algorithm **merge**.

The value types of the input iterators need be less-than comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. When there are equivalent elements in both source ranges, the elements in the first range precede the elements from the second source range in the destination range. If the source ranges contain duplicates of an element, then the destination range will contain the maximum number of those elements that occur in both source ranges.

The complexity of the algorithm is linear with at most 2 * ((*_Last1* – *_First1*) – (*_Last2* – *_First2*)) – 1 comparisons.

set_union has two related forms:

- checked_set_union
- unchecked_set_union

For information on how these functions behave, see Checked Iterators.

Example

```
// alg_set_union.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
   if ( elem1 < 0 )
      elem1 = - elem1;
   if ( elem2 < 0 )
      elem2 = - elem2;
   return elem1 < elem2;
}

int main( )
{
   using namespace std;
   vector <int> v1a, v1b, v1 ( 12 );
   vector <int>::iterator Iter1a, Iter1b, Iter1, Result1;

   // Constructing vectors v1a & v1b with default less than ordering
   int i;
   for ( i = -1 ; i <= 3 ; i++ )
   {
      v1a.push_back(  i );
   }

   int ii;
   for ( ii =-3 ; ii <= 1 ; ii++ )
   {
      v1b.push_back(  ii  );
   }

   cout << "Original vector v1a with range sorted by the\n "
        <<  "binary predicate less than is  v1a = ( " ;
   for ( Iter1a = v1a.begin( ) ; Iter1a != v1a.end( ) ; Iter1a++ )
      cout << *Iter1a << " ";
   cout << "))." << endl;

   cout << "Original vector v1b with range sorted by the\n "
```

```
     <<  "binary predicate less than is  v1b = ( " ;
for ( Iter1b = v1b.begin ( ) ; Iter1b != v1b.end ( ) ; Iter1b++ )
   cout << *Iter1b << " ";
cout << ")." << endl;

// Constructing vectors v2a & v2b with ranges sorted by greater
vector <int> v2a ( v1a ) , v2b ( v1b ) , v2 ( v1 );
vector <int>::iterator Iter2a,  Iter2b, Iter2, Result2;
sort ( v2a.begin ( ) , v2a.end ( ) , greater<int> ( ) );
sort ( v2b.begin ( ) , v2b.end ( ) , greater<int> ( ) );

cout << "Original vector v2a with range sorted by the\n "
     <<  "binary predicate greater is   v2a =  ( " ;
for ( Iter2a = v2a.begin ( ) ; Iter2a != v2a.end ( ) ; Iter2a++ )
   cout << *Iter2a << " ";
cout << ")." << endl;

cout << "Original vector v2b with range sorted by the\n "
     <<  "binary predicate greater is   v2b =  ( " ;
for ( Iter2b = v2b.begin ( ) ; Iter2b != v2b.end ( ) ; Iter2b++ )
   cout << *Iter2b << " ";
cout << ")." << endl;

// Constructing vectors v3a & v3b with ranges sorted by mod_lesser
vector <int> v3a ( v1a ), v3b ( v1b ) ,  v3 ( v1 );
vector <int>::iterator Iter3a, Iter3b, Iter3, Result3;
sort ( v3a.begin ( ) , v3a.end ( ) , mod_lesser );
sort ( v3b.begin ( ) , v3b.end ( ) , mod_lesser  );

cout << "Original vector v3a with range sorted by the\n "
     <<  "binary predicate mod_lesser is   v3a =  ( " ;
for ( Iter3a = v3a.begin ( ) ; Iter3a != v3a.end ( ) ; Iter3a++ )
   cout << *Iter3a << " ";
cout << ")." << endl;

cout << "Original vector v3b with range sorted by the\n "
     <<  "binary predicate mod_lesser is   v3b =  ( " ;
for ( Iter3b = v3b.begin ( ) ; Iter3b != v3b.end ( ) ; Iter3b++ )
   cout << *Iter3b << " ";
cout << ")." << endl;

// To combine into a union in ascending order with the default
 // binary predicate less <int> ( )
Result1 = set_union ( v1a.begin ( ) , v1a.end ( ) ,
   v1b.begin ( ) , v1b.end ( ) , v1.begin ( ) );
cout << "Union of source ranges with default order,"
     << "\n vector v1mod =  ( " ;
for ( Iter1 = v1.begin( ) ; Iter1 != Result1 ; Iter1++ )
   cout << *Iter1 << " ";
cout << ")." << endl;

// To combine into a union in descending order, specify binary
// predicate greater<int>( )
Result2 = set_union (  v2a.begin ( ) , v2a.end ( ) ,
   v2b.begin ( ) , v2b.end ( ) ,v2.begin ( ) , greater <int> ( ) );
cout << "Union of source ranges with binary predicate greater "
     << "specified,\n vector v2mod  = ( " ;
for ( Iter2 = v2.begin( ) ; Iter2 != Result2 ; Iter2++ )
   cout << *Iter2 << " ";
cout << ")." << endl;

// To combine into a union applying a user-defined
// binary predicate mod_lesser
Result3 = set_union ( v3a.begin ( ) , v3a.end ( ) ,
   v3b.begin ( ) , v3b.end ( ) , v3.begin ( ) , mod_lesser );
cout << "Union of source ranges with binary predicate "
     << "mod_lesser specified,\n vector v3mod  = ( " ; ;
for ( Iter3 = v3.begin( ) ; Iter3 != Result3 ; Iter3++ )
```

```
        cout << *Iter3 << " ";
    cout << ")." << endl;
  }
```

Output

```
Original vector v1a with range sorted by the
 binary predicate less than is  v1a = ( -1 0 1 2 3 ).
Original vector v1b with range sorted by the
 binary predicate less than is  v1b = ( -3 -2 -1 0 1 ).
Original vector v2a with range sorted by the
 binary predicate greater is   v2a =  ( 3 2 1 0 -1 ).
Original vector v2b with range sorted by the
 binary predicate greater is   v2b =  ( 1 0 -1 -2 -3 ).
Original vector v3a with range sorted by the
 binary predicate mod_lesser is   v3a =  ( 0 -1 1 2 3 ).
Original vector v3b with range sorted by the
 binary predicate mod_lesser is   v3b =  ( 0 -1 1 -2 -3 ).
Union of source ranges with default order,
 vector v1mod =  ( -3 -2 -1 0 1 2 3 ).
Union of source ranges with binary predicate greater specified,
 vector v2mod  = ( 3 2 1 0 -1 -2 -3 ).
Union of source ranges with binary predicate mod_lesser specified,
 vector v3mod  = ( 0 -1 1 2 3 ).
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

sort

Arranges the elements in a specified range into a nondescending order or according to an ordering criterion specified by a binary predicate.

```
template<class RandomAccessIterator>
   void sort(
      RandomAccessIterator _First,
      RandomAccessIterator _Last
   );
template<class RandomAccessIterator, class Pr>
   void sort(
      RandomAccessIterator _First,
      RandomAccessIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

_First

A random-access iterator addressing the position of the first element in the range to be sorted.

_Last

A random-access iterator addressing the position one past the final element in the range to be sorted.

_Comp

User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied. This comparator function must impose a strict weak ordering on pairs of elements from the sequence. For more information, see Algorithms.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Elements are equivalent, but not necessarily equal, if neither is less than the other. The **sort** algorithm is not stable and so does not guarantee that the relative ordering of equivalent elements will be preserved. The algorithm **stable_sort** does preserve this original ordering.

The average of a sort complexity is $O(N \log N)$, where $N = _Last - _First$.

Example

```
// alg_sort.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether first element is greater than the second
bool UDgreater ( int elem1, int elem2 )
{
   return elem1 > elem2;
}

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1;

   int i;
```

```
   for ( i = 0 ; i <= 5 ; i++ )
   {
      v1.push_back( 2 * i );
   }

   int ii;
   for ( ii = 0 ; ii <= 5 ; ii++ )
   {
      v1.push_back( 2 * ii + 1 );
   }

   cout << "Original vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   sort( v1.begin( ), v1.end( ) );
   cout << "Sorted vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // To sort in descending order. specify binary predicate
   sort( v1.begin( ), v1.end( ), greater<int>( ) );
   cout << "Resorted (greater) vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // A user-defined (UD) binary predicate can also be used
   sort( v1.begin( ), v1.end( ), UDgreater );
   cout << "Resorted (UDgreater) vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;
}
```

Output

```
Original vector v1 = ( 0 2 4 6 8 10 1 3 5 7 9 11 )
Sorted vector v1 = ( 0 1 2 3 4 5 6 7 8 9 10 11 )
Resorted (greater) vector v1 = ( 11 10 9 8 7 6 5 4 3 2 1 0 )
Resorted (UDgreater) vector v1 = ( 11 10 9 8 7 6 5 4 3 2 1 0 )
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

sort_heap

Converts a heap into a sorted range.

```
template<class RandomAccessIterator>
   void sort_heap(
      RandomAccessIterator _First,
      RandomAccessIterator _Last
   );
template<class RandomAccessIterator, class Pr>
   void sort_heap(
      RandomAccessIterator _First,
      RandomAccessIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

_First

A random-access iterator addressing the position of the first element in the target heap.

_Last

A random-access iterator addressing the position one past the final element in the target heap.

_Comp

User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

Heaps have two properties:

- The first element is always the largest.
- Elements may be added or removed in logarithmic time.

After the application if this algorithm, the range it was applied to is no longer a heap.

This is not a stable sort because the relative order of equivalent elements is not necessarily preserved.

Heaps are an ideal way to implement priority queues and they are used in the implementation of the Standard Template Library container adaptor priority_queue Class.

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The complexity is at most *N* log *N*, where *N* = (*_Last* – *_First*).

Example

```
// alg_sort_heap.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

int main( ) {
   using namespace std;
   vector <int> v1, v2;
   vector <int>::iterator Iter1, Iter2;

   int i;
   for ( i = 1 ; i <= 9 ; i++ )
      v1.push_back( i );
```

```
   random_shuffle( v1.begin( ), v1.end( ) );

   cout << "Vector v1 is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // Sort heap v1 with default less-than ordering
   sort_heap (v1.begin( ), v1.end( ) );
   cout << "The heap v1 becomes the sorted range: ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // Make v1 a heap with greater than ordering
   make_heap ( v1.begin( ), v1.end( ), greater<int>( ) );
   cout << "The greater-than heaped version of v1 is\n ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   sort_heap(v1.begin( ), v1.end( ), greater<int>( ) );
   cout << "The greater-than heap v1 becomes the sorted range\n ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;
}
```

Sample Output

```
Vector v1 is ( 9 2 7 3 1 6 8 4 5 ).
The heap v1 becomes the sorted range: ( 1 2 3 4 5 6 8 7 9 ).
The greater-than heaped version of v1 is
 ( 1 2 3 4 5 6 8 7 9 ).
The greater-than heap v1 becomes the sorted range
 ( 9 8 7 6 5 4 3 2 1 ).
```

Requirements

Header: <algorithm>

See Also

Reference

heap

Standard Template Library

Other Resources

<algorithm> Members

stable_partition

Classifies elements in a range into two disjoint sets, with those elements satisfying a unary predicate preceding those that fail to satisfy it, preserving the relative order of equivalent elements.

```
template<class BidirectionalIterator, class Predicate>
   BidirectionalIterator stable_partition(
      BidirectionalIterator _First,
      BidirectionalIterator _Last,
      Predicate _Pred
   );
```

Parameters

_First

A bidirectional iterator addressing the position of the first element in the range to be partitioned.

_Last

A bidirectional iterator addressing the position one past the final element in the range to be partitioned.

_Pred

User-defined predicate function object that defines the condition to be satisfied if an element is to be classified. A predicate takes single argument and returns **true** or **false**.

Return Value

A bidirectional iterator addressing the position of the first element in the range to not satisfy the predicate condition.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Elements *a* and *b* are equivalent, but not necessarily equal, if both *Pr* (*a*, *b*) is false and *Pr* (*b*, *a*) if false, where *Pr* is the parameter-specified predicate. The **stable_ partition** algorithm is stable and guarantees that the relative ordering of equivalent elements will be preserved. The algorithm **partition** does not necessarily preserve this original ordering.

Example

```
// alg_stable_partition.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

bool greater5 ( int value ) {
   return value >5;
}

int main( ) {
   using namespace std;
   vector <int> v1, v2;
   vector <int>::iterator Iter1, Iter2, result;

   int i;
   for ( i = 0 ; i <= 10 ; i++ )
      v1.push_back( i );

   int ii;
   for ( ii = 0 ; ii <= 4 ; ii++ )
      v1.push_back( 5 );

   random_shuffle(v1.begin( ), v1.end( ) );
```

```
    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << "))." << endl;

    // Partition the range with predicate greater10
    result = stable_partition (v1.begin( ), v1.end( ), greater5 );
    cout << "The partitioned set of elements in v1 is:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << "))." << endl;

    cout << "The first element in v1 to fail to satisfy the"
         << "\n predicate greater5 is: " << *result << "." << endl;
}
```

Sample Output

```
Vector v1 is ( 5 1 9 2 0 5 7 3 4 5 8 5 5 5 10 6 ).
The partitioned set of elements in v1 is:
 ( 9 7 8 10 6 5 1 2 0 5 3 4 5 5 5 5 ).
The first element in v1 to fail to satisfy the
 predicate greater5 is: 5.
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

stable_sort

Arranges the elements in a specified range into a nondescending order or according to an ordering criterion specified by a binary predicate and preserves the relative ordering of equivalent elements.

```
template<class BidirectionalIterator>
   void stable_sort(
      BidirectionalIterator _First,
      BidirectionalIterator _Last
   );
template<class BidirectionalIterator, class BinaryPredicate>
   void stable_sort(
      BidirectionalIterator _First,
      BidirectionalIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

_First

A bidirectional iterator addressing the position of the first element in the range to be sorted.

_Last

A bidirectional iterator addressing the position one past the final element in the range to be sorted.

_Comp

User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Elements are equivalent, but not necessarily equal, if neither is less than the other. The **sort** algorithm is stable and guarantees that the relative ordering of equivalent elements will be preserved.

The run-time complexity of **stable_sort** depends on the amount of memory available, but the best case (given sufficient memory) is $O(N \log N)$ and the worst case is $O(N (\log N)2)$, where N = *_Last* – *First*. Usually, the **sort** algorithm is significantly faster than **stable_sort**.

Example

```
// alg_stable_sort.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether first element is greater than the second
bool UDgreater (int elem1, int elem2 )
{
   return elem1 > elem2;
}

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1;

   int i;
   for ( i = 0 ; i <= 5 ; i++ )
```

```
    {
       v1.push_back( 2 * i );
    }

    for ( i = 0 ; i <= 5 ; i++ )
    {
       v1.push_back( 2 * i  );
    }

    cout << "Original vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << ")" << endl;

    stable_sort(v1.begin( ), v1.end( ) );
    cout << "Sorted vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << ")" << endl;

    // To sort in descending order, specify binary predicate
    stable_sort(v1.begin( ), v1.end( ), greater<int>( ) );
    cout << "Resorted (greater) vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << ")" << endl;

    // A user-defined (UD) binary predicate can also be used
    stable_sort(v1.begin( ), v1.end( ), UDgreater );
    cout << "Resorted (UDgreater) vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << ")" << endl;
 }
```

Output

```
Original vector v1 = ( 0 2 4 6 8 10 0 2 4 6 8 10 )
Sorted vector v1 = ( 0 0 2 2 4 4 6 6 8 8 10 10 )
Resorted (greater) vector v1 = ( 10 10 8 8 6 6 4 4 2 2 0 0 )
Resorted (UDgreater) vector v1 = ( 10 10 8 8 6 6 4 4 2 2 0 0 )
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

swap

Exchanges the values of the elements between two types of objects, assigning the contents of the first object to the second object and the contents of the second to the first.

```
template<class Type>
   void swap(
      Type& _Left,
      Type& _Right
   );
```

Parameters

_Left

The first object to have its elements exchanged.

_Right

The second object to have its elements exchanged.

Remarks

This algorithm is exceptional in the STL in being designed to operate on individual elements rather than on a range of elements.

Example

```
// alg_swap.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

bool greater5 ( int value )
{
   return value >5;
}

int main( )
{
   using namespace std;
   vector <int> v1, v2;
   vector <int>::iterator Iter1, Iter2, result;

   int i;
   for ( i = 0 ; i <= 10 ; i++ )
   {
      v1.push_back( i );
   }

   int ii;
   for ( ii = 0 ; ii <= 4 ; ii++ )
   {
      v2.push_back( 5 );
   }

   cout << "Vector v1 is ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   cout << "Vector v2 is ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << "))." << endl;
```

```
    swap( v1, v2 );

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << "))." << endl;

    cout << "Vector v2 is ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
       cout << *Iter2 << " ";
    cout << "))." << endl;
  }
```

Output

```
Vector v1 is ( 0 1 2 3 4 5 6 7 8 9 10 ).
Vector v2 is ( 5 5 5 5 5 ).
Vector v1 is ( 5 5 5 5 5 ).
Vector v2 is ( 0 1 2 3 4 5 6 7 8 9 10 ).
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

swap_ranges

Exchanges the elements of one range with the elements of another, equal sized range.

```
template<class ForwardIterator1, class ForwardIterator2>
   ForwardIterator2 swap_ranges(
      ForwardIterator1 _First1,
      ForwardIterator1 _Last1,
      ForwardIterator2 _First2
   );
```

Parameters

_First1

A forward iterator pointing to the first position of the first range whose elements are to be exchanged.

_Last1

A forward iterator pointing to one past the final position of the first range whose elements are to be exchanged.

_First2

A forward iterator pointing to the first position of the second range whose elements are to be exchanged.

Return Value

A forward iterator pointing to one past the final position of the second range whose elements are to be exchanged.

Remarks

The ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position is reachable from the first by incrementation. The second range has to be as large as the first range.

The complexity is linear with *_Last1* – *_First1* swaps performed. If elements from containers of the same type are being swapped, them the **swap** member function from that container should be used, because the member function typically has constant complexity.

Example

```
// alg_swap_ranges.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   deque <int> d1;
   vector <int>::iterator v1Iter1;
   deque<int>::iterator d1Iter1;

   int i;
   for ( i = 0 ; i <= 5 ; i++ )
   {
      v1.push_back( i );
   }

   int ii;
   for ( ii =4 ; ii <= 9 ; ii++ )
   {
      d1.push_back( 6 );
   }
```

```
    cout << "Vector v1 is ( " ;
    for ( v1Iter1 = v1.begin( ) ; v1Iter1 != v1.end( ) ;v1Iter1 ++ )
       cout << *v1Iter1  << " ";
    cout << "))." << endl;

    cout << "Deque d1 is  ( " ;
    for ( d1Iter1 = d1.begin( ) ; d1Iter1 != d1.end( ) ;d1Iter1 ++ )
       cout << *d1Iter1  << " ";
    cout << "))." << endl;

    swap_ranges ( v1.begin ( ) , v1.end ( ) , d1.begin ( ) );

    cout << "After the swap_range, vector v1 is ( " ;
    for ( v1Iter1 = v1.begin( ) ; v1Iter1 != v1.end( ) ;v1Iter1 ++ )
       cout << *v1Iter1 << " ";
    cout << "))." << endl;

    cout << "After the swap_range deque d1 is   ( " ;
    for ( d1Iter1 = d1.begin( ) ; d1Iter1 != d1.end( ) ;d1Iter1 ++ )
       cout << *d1Iter1 << " ";
    cout << "))." << endl;
  }
```

Output

```
Vector v1 is ( 0 1 2 3 4 5 ).
Deque d1 is  ( 6 6 6 6 6 6 ).
After the swap_range, vector v1 is ( 6 6 6 6 6 6 ).
After the swap_range deque d1 is   ( 0 1 2 3 4 5 ).
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

transform

Applies a specified function object to each element in a source range or to a pair of elements from two source ranges and copies the return values of the function object into a destination range.

```
template<class InputIterator, class OutputIterator, class UnaryFunction>
   OutputIterator transform(
      InputIterator _First1,
      InputIterator _Last1,
      OutputIterator _Result,
      UnaryFunction _Func
      );
template<class InputIterator1, class InputIterator2, class OutputIterator,
   class BinaryFunction>
   OutputIterator transform(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      OutputIterator _Result,
      BinaryFunction _Func
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first source range to be operated on.

_Last1

An input iterator addressing the position one past the final element in the first source range operated on.

_First2

An input iterator addressing the position of the first element in the second source range to be operated on.

_Result

An output iterator addressing the position of the first element in the destination range.

_Func

User-defined unary function object used in the first version of the algorithm that is applied to each element in the first source range or A user-defined (UD) binary function object used in the second version of the algorithm that is applied pairwise, in a forward order, to the two source ranges.

Return Value

An output iterator addressing the position one past the final element in the destination range that is receiving the output elements transformed by the function object.

Remarks

The ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation. The destination range must be large enough to contain the transformed source range.

If *_Result* is set equal to *_First1* in the first version of the algorithm, then the source and destination ranges will be the same and the sequence will be modified in place. But the *_Result* may not address a position within the range [*_First1* +1, *_Last1*).

The complexity is linear with at most (*_Last1* – *_First1*) comparisons.

Example

```
// alg_transform.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
```

```
#include <functional>
#include <iostream>

// The function object multiplies an element by a Factor
template <class Type>
class MultValue
{
   private:
      Type Factor;   // The value to multiply by
   public:
      // Constructor initializes the value to multiply by
      MultValue ( const Type& _Val ) : Factor ( _Val ) {
      }

      // The function call for the element to be multiplied
      int operator ( ) ( Type& elem ) const
      {
         return elem * Factor;
      }
};

int main( )
{
   using namespace std;
   vector <int> v1, v2 ( 7 ), v3 ( 7 );
   vector <int>::iterator Iter1, Iter2 , Iter3;

   // Constructing vector v1
   int i;
   for ( i = -4 ; i <= 2 ; i++ )
   {
      v1.push_back(  i );
   }

   cout << "Original vector  v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;

   // Modifying the vector v1 in place
   transform (v1.begin ( ) , v1.end ( ) , v1.begin ( ) , MultValue<int> ( 2 ) );
   cout << "The elements of the vector v1 multiplied by 2 in place gives:"
        << "\n v1mod = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")." << endl;

   // Using transform to multiply each element by a factor of 5
   transform ( v1.begin ( ) , v1.end ( ) , v2.begin ( ) , MultValue<int> ( 5 ) );

   cout << "Multiplying the elements of the vector v1mod\n "
        <<  "by the factor 5 & copying to v2 gives:\n v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << ")." << endl;

   // The second version of transform used to multiply the
   // elements of the vectors v1mod & v2 pairwise
   transform ( v1.begin ( ) , v1.end ( ) ,  v2.begin ( ) , v3.begin ( ) ,
      multiplies <int> ( ) );

   cout << "Multiplying elements of the vectors v1mod and v2 pairwise "
        <<  "gives:\n v3 = ( " ;
   for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
      cout << *Iter3 << " ";
   cout << ")." << endl;
}
```

Output

```
Original vector  v1 = ( -4 -3 -2 -1 0 1 2 ).
The elements of the vector v1 multiplied by 2 in place gives:
 v1mod = ( -8 -6 -4 -2 0 2 4 ).
Multiplying the elements of the vector v1mod
 by the factor 5 & copying to v2 gives:
 v2 = ( -40 -30 -20 -10 0 10 20 ).
Multiplying elements of the vectors v1mod and v2 pairwise gives:
 v3 = ( 320 180 80 20 0 20 80 ).
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

unchecked_copy

Same as copy, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined. **unchecked_copy** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator, class OutputIterator>
   OutputIterator unchecked_copy(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _DestBeg
   );
```

Parameters

_First

An input iterator addressing the position of the first element in the source range.

_Last

An input iterator addressing the position that is one past the final element in the source range.

_DestBeg

An output iterator addressing the position of the first element in the destination range.

Return Value

An output iterator addressing the position that is one past the final element in the destination range, that is, the iterator addresses *_Result* + (*_Last* – *_First*).

Remarks

See copy for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

unchecked_copy_backward

Same as copy_backward, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined. **unchecked_copy_backward** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
   BidirectionalIterator2 unchecked_copy_backward(
      BidirectionalIterator1 _First,
      BidirectionalIterator1 _Last,
      BidirectionalIterator2 _DestEnd
   );
```

Parameters

_First

A bidirectional iterator addressing the position of the first element in the source range.

_Last

A bidirectional iterator addressing the position that is one past the final element in the source range.

_DestEnd

A bidirectional iterator addressing the position of the one past the final element in the destination range.

Return Value

An output iterator addressing the position that is one past the final element in the destination range, that is, the iterator addresses _ *DestEnd* – (_*Last* – _*First*).

Remarks

See copy_backward for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

unchecked_fill_n

Same as fill_n, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined. **unchecked_fill_n** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class OutputIterator, class Size, class Type>
   void unchecked_fill_n(
      OutputIterator _First,
      Size _Count,
      const Type& _Val
   );
```

Parameters

_First

An output iterator addressing the position of the first element in the range to be assigned the value *_Val*.

_Count

A signed or unsigned integer type specifying the number of elements to be assigned the value.

_Val

The value to be assigned to elements in the range [*_First*, *_First* + *_Count*).

Remarks

See fill_n for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

unchecked_generate_n

Same as generate_n, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined. **unchecked_generate_n** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class OutputIterator, class Size, class Generator>
   void unchecked_generate_n(
      OutputIterator _First,
      Size _Count,
      Generator _Gen
   );
```

Parameters

_First

An output iterator addressing the position of first element in the range to which values are to be assigned.

_Count

A signed or unsigned integer type specifying the number of elements to be assigned a value by the generator function.

_Gen

A function object that is called with no arguments that is used to generate the values to be assigned to each of the elements in the range.

Remarks

See generate_n for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

unchecked_merge

Same as merge, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined. **unchecked_merge** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
   OutputIterator unchecked_merge(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPred
icate>
   OutputIterator checked_merge(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
      BinaryPredicate _Comp
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be combined and sorted into a single range.

_Last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be combined and sorted into a single range.

_First2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be combined and sorted into a single range.

_Last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be combined and sorted into a single range.

_Result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be combined into a single sorted range.

_Comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range.

Remarks

See merge for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

unchecked_remove_copy

Same as remove_copy, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined. **unchecked_remove_copy** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator, class OutputIterator, class Type>
   OutputIterator unchecked_remove_copy(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      const Type& _Val
   );
```

Parameters

_First

An input iterator addressing the position of the first element in the range from which elements are being removed.

_Last

An input iterator addressing the position one past the final element in the range from which elements are being removed.

_Result

An output iterator addressing the position of the first element in the destination range to which elements are being removed.

_Val

The value that is to be removed from the range.

Return Value

A forward iterator addressing the new end position of the destination range, one past the final element of the copy of the remnant sequence free of the specified value.

Remarks

See remove_copy for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

unchecked_remove_copy_if

Same as remove_copy_if, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined. **unchecked_remove_copy_if** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator, class OutputIterator, class Predicate>
   OutputIterator unchecked_remove_copy_if(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      Predicate _Pred
   );
```

Parameters

_First

An input iterator addressing the position of the first element in the range from which elements are being removed.

_Last

An input iterator addressing the position one past the final element in the range from which elements are being removed.

_Result

An output iterator addressing the position of the first element in the destination range to which elements are being removed.

_Pred

The unary predicate that must be satisfied is the value of an element is to be replaced.

Return Value

A forward iterator addressing the new end position of the destination range, one past the final element of the remnant sequence free of the elements satisfying the predicate.

Remarks

See remove_copy_if for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

unchecked_replace_copy

Same as replace_copy, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined. **unchecked_replace_copy** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator, class OutputIterator, class Type>
   OutputIterator unchecked_replace_copy(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      const Type& _OldVal,
      const Type& _NewVal
   );
```

Parameters

_First

An input iterator pointing to the position of the first element in the range from which elements are being replaced.

_Last

An input iterator pointing to the position one past the final element in the range from which elements are being replaced.

_Result

An output iterator pointing to the first element in the destination range to where the altered sequence of elements is being copied.

_OldVal

The old value of the elements being replaced.

_NewVal

The new value being assigned to the elements with the old value.

Return Value

An output iterator pointing to the position one past the final element in the destination range to where the altered sequence of elements is being copied.

Remarks

See replace_copy for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

unchecked_replace_copy_if

Same as replace_copy_if, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined. **unchecked_replace_copy_if** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator, class OutputIterator, class Predicate, class Type>
   OutputIterator unchecked_replace_copy_if(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      Predicate _Pred,
      const Type& _Val
   );
```

Parameters

_First

An input iterator pointing to the position of the first element in the range from which elements are being replaced.

_Last

An input iterator pointing to the position one past the final element in the range from which elements are being replaced.

_Result

An output iterator pointing to the first element in the destination range to where the altered sequence of elements is being copied.

_OldVal

The old value of the elements being replaced.

_NewVal

The new value being assigned to the elements with the old value.

Return Value

An output iterator pointing to the position one past the final element in the destination range to where the altered sequence of elements is being copied.

Remarks

See replace_copy_if for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

unchecked_reverse_copy

Same as reverse_copy, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined. **unchecked_reverse_copy** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class BidirectionalIterator, class OutputIterator>
   OutputIterator unchecked_reverse_copy(
      BidirectionalIterator _First,
      BidirectionalIterator _Last,
      OutputIterator _Result
   );
```

Parameters

_First

A bidirectional iterator pointing to the position of the first element in the source range within which the elements are being permuted.

_Last

A bidirectional iterator pointing to the position one past the final element in the source range within which the elements are being permuted.

_Result

An output iterator pointing to the position of the first element in the destination range to which elements are being copied.

Return Value

An output iterator pointing to the position one past the final element in the destination range to where the altered sequence of elements is being copied.

Remarks

See reverse_copy for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

unchecked_rotate_copy

Same as rotate_copy, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined. **unchecked_rotate_copy** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class ForwardIterator, class OutputIterator>
   OutputIterator unchecked_rotate_copy(
      ForwardIterator _First,
      ForwardIterator _Middle,
      ForwardIterator _Last,
      OutputIterator _Result
   );
```

Parameters

_First

A forward iterator addressing the position of the first element in the range to be rotated.

_Middle

A forward iterator defining the boundary within the range that addresses the position of the first element in the second part of the range whose elements are to be exchanged with those in the first part of the range.

_ Last

A forward iterator addressing the position one past the final element in the range to be rotated.

_Result

An output iterator addressing the position of the first element in the destination range.

Return Value

An output iterator addressing the position one past the final element in the destination range.

Remarks

See rotate_copy for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

unchecked_set_difference

Same as set_difference, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined. **unchecked_set_difference** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
   OutputIterator unchecked_set_difference(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPred
icate>
   OutputIterator unchecked_set_difference(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result,
      BinaryPredicate _Comp
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

_Last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

_First2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

_Last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

_Result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the difference of the two source ranges.

_Comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range representing the difference of the two source ranges.

Remarks

See set_difference for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

unchecked_set_intersection

Same as set_intersection, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined. **unchecked_set_intersection** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
   OutputIterator unchecked_set_intersection(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPred
icate>
   OutputIterator unchecked_set_intersection(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result,
      BinaryPredicate _Comp
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

_Last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

_First2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

_Last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

_Result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the intersection of the two source ranges.

_Comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range representing the intersection of the two source ranges.

Remarks

See set_intersection for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

unchecked_set_symmetric_difference

Same as set_symmetric_difference, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined. **unchecked_set_symmetric_difference** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
   OutputIterator unchecked_set_symmetric_difference(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPred
icate>
   OutputIterator unchecked_set_symmetric_difference(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result,
      BinaryPredicate _Comp
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

_Last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

_First2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

_Last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

_Result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the symmetric difference of the two source ranges.

_Comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range representing the symmetric difference of the two source ranges.

Remarks

See set_symmetric_difference for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

unchecked_set_union

Same as set_union, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined. **unchecked_set_union** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
   OutputIterator unchecked_set_union(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPred
icate>
   OutputIterator unchecked_set_union(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result,
      BinaryPredicate _Comp
   );
```

Parameters

_First1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

_Last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

_First2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

_Last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

_Result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the union of the two source ranges.

_Comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range representing the union of the two source ranges.

Remarks

See set_union for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

unchecked_unique_copy

Same as unique_copy, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined. **unchecked_unique_copy** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator, class OutputIterator>
   OutputIterator unchecked_unique_copy(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result
   );
template<class InputIterator, class OutputIterator, class BinaryPredicate>
   OutputIterator unchecked_unique_copy(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      BinaryPredicate _Comp,
   );
```

Parameters

_First

A forward iterator addressing the position of the first element in the source range to be copied.

_Last

A forward iterator addressing the position one past the final element in the source range to be copied.

_Result

An output iterator addressing the position of the first element in the destination range that is receiving the copy with consecutive duplicates removed.

_Comp

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

An output iterator addressing the position one past the final element in the destination range that is receiving the copy with consecutive duplicates removed.

Remarks

See unique_copy for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

unique (<algorithm>)

Removes duplicate elements that are adjacent to each other in a specified range.

```
template<class ForwardIterator>
   ForwardIterator unique(
      ForwardIterator _First,
      ForwardIterator _Last
   );
template<class ForwardIterator, class Pr>
   ForwardIterator unique(
      ForwardIterator _First,
      ForwardIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

_First

A forward iterator addressing the position of the first element in the range to be scanned for duplicate removal.

_Last

A forward iterator addressing the position one past the final element in the range to be scanned for duplicate removal.

_Comp

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

A forward iterator to the new end of the modified sequence that contains no consecutive duplicates, addressing the position one past the last element not removed.

Remarks

Both forms of the algorithm remove the second duplicate of a consecutive pair of equal elements.

The operation of the algorithm is stable so that the relative order of the undeleted elements is not changed.

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation. he number of elements in the sequence is not changed by the algorithm **unique** and the elements beyond the end of the modified sequence are dereferenceable but not specified.

The complexity is linear, requiring (*_Last* – *_First*) – 1 comparisons.

List provides a more efficient member function unique, which may perform better.

These algorithms cannot be used on an associative container.

Example

```
// alg_unique.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
#include <ostream>

using namespace std;

// Return whether modulus of elem1 is equal to modulus of elem2
bool mod_equal ( int elem1, int elem2 )
{
   if ( elem1 < 0 )
```

```
      elem1 = - elem1;
   if ( elem2 < 0 )
      elem2 = - elem2;
   return elem1 == elem2;
};

int main( )
{
   vector <int> v1;
   vector <int>::iterator v1_Iter1, v1_Iter2, v1_Iter3,
         v1_NewEnd1, v1_NewEnd2, v1_NewEnd3;

   int i;
   for ( i = 0 ; i <= 3 ; i++ )
   {
      v1.push_back( 5 );
      v1.push_back( -5 );
   }

   int ii;
   for ( ii = 0 ; ii <= 3 ; ii++ )
   {
      v1.push_back( 4 );
   }
   v1.push_back( 7 );

   cout << "Vector v1 is ( " ;
   for ( v1_Iter1 = v1.begin( ) ; v1_Iter1 != v1.end( ) ; v1_Iter1++ )
      cout << *v1_Iter1 << " ";
   cout << "))." << endl;

   // Remove consecutive duplicates
   v1_NewEnd1 = unique ( v1.begin ( ) , v1.end ( ) );

   cout << "Removing adjacent duplicates from vector v1 gives\n ( " ;
   for ( v1_Iter1 = v1.begin( ) ; v1_Iter1 != v1_NewEnd1 ; v1_Iter1++ )
      cout << *v1_Iter1 << " ";
   cout << "))." << endl;

   // Remove consecutive duplicates under the binary prediate mod_equals
   v1_NewEnd2 = unique ( v1.begin ( ) , v1_NewEnd1 , mod_equal );

   cout << "Removing adjacent duplicates from vector v1 under the\n "
        << " binary predicate mod_equal gives\n ( " ;
   for ( v1_Iter2 = v1.begin( ) ; v1_Iter2 != v1_NewEnd2 ; v1_Iter2++ )
      cout << *v1_Iter2 << " ";
   cout << "))." << endl;

   // Remove elements if preceded by an element that was greater
   v1_NewEnd3 = unique ( v1.begin ( ) , v1_NewEnd2, greater<int>( ) );

   cout << "Removing adjacent elements satisfying the binary\n "
        << " predicate mod_equal from vector v1 gives ( " ;
   for ( v1_Iter3 = v1.begin( ) ; v1_Iter3 != v1_NewEnd3 ; v1_Iter3++ )
      cout << *v1_Iter3 << " ";
   cout << "))." << endl;
}
```

Output

```
Vector v1 is ( 5 -5 5 -5 5 -5 5 -5 4 4 4 4 7 ).
Removing adjacent duplicates from vector v1 gives
 ( 5 -5 5 -5 5 -5 5 -5 4 7 ).
Removing adjacent duplicates from vector v1 under the
  binary predicate mod_equal gives
 ( 5 4 7 ).
Removing adjacent elements satisfying the binary
  predicate mod_equal from vector v1 gives ( 5 7 ).
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

unique_copy

Copies elements from a source range into a destination range except for the duplicate elements that are adjacent to each other.

```
template<class InputIterator, class OutputIterator>
   OutputIterator unique_copy(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result
   );
template<class InputIterator, class OutputIterator, class BinaryPredicate>
   OutputIterator unique_copy(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      BinaryPredicate _Comp,
      );
```

Parameters

_First

A forward iterator addressing the position of the first element in the source range to be copied.

_Last

A forward iterator addressing the position one past the final element in the source range to be copied.

_Result

An output iterator addressing the position of the first element in the destination range that is receiving the copy with consecutive duplicates removed.

_Comp

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

An output iterator addressing the position one past the final element in the destination range that is receiving the copy with consecutive duplicates removed.

Remarks

Both forms of the algorithm remove the second duplicate of a consecutive pair of equal elements.

The operation of the algorithm is stable so that the relative order of the undeleted elements is not changed.

The ranges referenced must be valid; all pointers must be dereferenceable and within a sequence the last position is reachable from the first by incrementation.

The complexity is linear, requiring (*_Last* – *_First*) comparisons.

unique_copy has two related forms:

- checked_unique_copy
- unchecked_unique_copy

For information on how these functions behave, see Checked Iterators.

Example

```
// alg_unique_copy.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
```

```
#include <iostream>
#include <ostream>

using namespace std;

// Return whether modulus of elem1 is equal to modulus of elem2
bool mod_equal ( int elem1, int elem2 ) {
   if ( elem1 < 0 )
      elem1 = - elem1;
   if ( elem2 < 0 )
      elem2 = - elem2;
   return elem1 == elem2;
};

int main() {
   vector <int> v1;
   vector <int>::iterator v1_Iter1, v1_Iter2,
         v1_NewEnd1, v1_NewEnd2;

   int i;
   for ( i = 0 ; i <= 1 ; i++ ) {
      v1.push_back( 5 );
      v1.push_back( -5 );
   }

   int ii;
   for ( ii = 0 ; ii <= 2 ; ii++ )
      v1.push_back( 4 );
   v1.push_back( 7 );

   int iii;
   for ( iii = 0 ; iii <= 5 ; iii++ )
      v1.push_back( 10 );

   cout << "Vector v1 is\n ( " ;
   for ( v1_Iter1 = v1.begin( ) ; v1_Iter1 != v1.end( ) ; v1_Iter1++ )
      cout << *v1_Iter1 << " ";
   cout << "))." << endl;

   // Copy first half to second, removing consecutive duplicates
   v1_NewEnd1 = unique_copy ( v1.begin ( ) , v1.begin ( ) + 8, v1.begin ( ) + 8 );

   cout << "Copying the first half of the vector to the second half\n "
        << "while removing adjacent duplicates gives\n ( " ;
   for ( v1_Iter1 = v1.begin( ) ; v1_Iter1 != v1_NewEnd1 ; v1_Iter1++ )
      cout << *v1_Iter1 << " ";
   cout << "))." << endl;

   int iv;
   for ( iv = 0 ; iv <= 7 ; iv++ )
      v1.push_back( 10 );

   // Remove consecutive duplicates under the binary prediate mod_equals
   v1_NewEnd2 = unique_copy ( v1.begin ( ) , v1.begin ( ) + 14,
      v1.begin ( ) + 14 , mod_equal );

   cout << "Copying the first half of the vector to the second half\n "
        << " removing adjacent duplicates under mod_equals gives\n ( " ;
   for ( v1_Iter2 = v1.begin( ) ; v1_Iter2 != v1_NewEnd2 ; v1_Iter2++ )
      cout << *v1_Iter2 << " ";
   cout << "))." << endl;
}
```

Output

```
Vector v1 is
 ( 5 -5 5 -5 4 4 4 7 10 10 10 10 10 10 ).
Copying the first half of the vector to the second half
```

```
 while removing adjacent duplicates gives
 ( 5 -5 5 -5 4 4 4 7 5 -5 5 -5 4 7 ).
Copying the first half of the vector to the second half
  removing adjacent duplicates under mod_equals gives
 ( 5 -5 5 -5 4 4 4 7 5 -5 5 -5 4 7 5 4 7 5 4 7 ).
```

Requirements

Header: <algorithm>

See Also

Reference

Standard Template Library

Other Resources

<algorithm> Members

upper_bound

Finds the position of the first element in an ordered range that has a value that is greater than a specified value, where the ordering criterion may be specified by a binary predicate.

```
template<class ForwardIterator, class Type>
   ForwardIterator upper_bound(
      ForwardIterator _First,
      ForwardIterator _Last,
      const Type& _Val
   );
template<class ForwardIterator, class Type, class Pr>
   ForwardIterator upper_bound(
      ForwardIterator _First,
      ForwardIterator _Last,
      const Type& _Val,
      BinaryPredicate _Comp
   );
```

Parameters

_First

The position of the first element in the range to be searched.

_Last

The position one past the final element in the range to be searched.

_Val

The value in the ordered range that needs to be exceeded by the value of the element addressed by the iterator returned.

_Comp

User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

A forward iterator to the position of the first element that has a value greater than a specified value.

Remarks

The sorted source range referenced must be valid; all iterators must be dereferenceable and within the sequence the last position must be reachable from the first by incrementation.

A sorted range is a precondition of the use of **upper_bound** and where the ordering criterion is the same as specified by the binary predicate.

The range is not modified by **upper_bound**.

The value types of the forward iterators need be less-than comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements

The complexity of the algorithm is logarithmic for random-access iterators and linear otherwise, with the number of steps proportional to (*_Last1* – *_First1*).

Example

```
// alg_upper_bound.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>
```

```
// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
   if ( elem1 < 0 )
      elem1 = - elem1;
   if ( elem2 < 0 )
      elem2 = - elem2;
   return elem1 < elem2;
}

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1, Result1;

   // Constructing vectors v1a & v1b with default less-than ordering
   int i;
   for ( i = -1 ; i <= 4 ; i++ )
   {
      v1.push_back(  i );
   }

   int ii;
   for ( ii =-3 ; ii <= 0 ; ii++ )
   {
      v1.push_back(  ii  );
   }

   sort ( v1.begin ( ) , v1.end ( ) );
   cout << "Original vector v1 with range sorted by the\n "
        << "binary predicate less than is  v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << "))." << endl;

   // Constructing vectors v2 with range sorted by greater
   vector <int> v2 ( v1 );
   vector <int>::iterator Iter2, Result2;
   sort ( v2.begin ( ) , v2.end ( ) , greater<int> ( ) );

   cout << "Original vector v2 with range sorted by the\n "
        <<  "binary predicate greater is    v2 = ( " ;
   for ( Iter2 = v2.begin ( ) ; Iter2 != v2.end ( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << "))." << endl;

   // Constructing vectors v3 with range sorted by mod_lesser
   vector <int> v3 ( v1 );
   vector <int>::iterator Iter3, Result3;
   sort ( v3.begin ( ) , v3.end ( ) , mod_lesser );

   cout << "Original vector v3 with range sorted by the\n "
        <<  "binary predicate mod_lesser is v3 = ( " ;
   for ( Iter3 = v3.begin ( ) ; Iter3 != v3.end ( ) ; Iter3++ )
      cout << *Iter3 << " ";
   cout << "))." << endl;

   // upper_bound of 3 in v1 with default binary predicate less <int> ( )
   Result1 = upper_bound ( v1.begin ( ) , v1.end ( ) , 3 );
   cout << "The upper_bound in v2 for the element with a value of 3 is: "
        << *Result1 << "." << endl;

   // upper_bound of 3 in v2 with the binary predicate greater <int> ( )
   Result2 = upper_bound ( v2.begin ( ) , v2.end ( ) , 3, greater <int> ( ) );
   cout << "The upper_bound in v2 for the element with a value of 3 is: "
        << *Result2 << "." << endl;
```

```
    // upper_bound of 3 in v3 with the binary predicate  mod_lesser
    Result3 = upper_bound ( v3.begin ( ) , v3.end ( ) , 3,mod_lesser );
    cout << "The upper_bound in v3 for the element with a value of 3 is: "
         << *Result3 << "." << endl;
}
```

Output

```
Original vector v1 with range sorted by the
 binary predicate less than is  v1 = ( -3 -2 -1 -1 0 0 1 2 3 4 ).
Original vector v2 with range sorted by the
 binary predicate greater is    v2 = ( 4 3 2 1 0 0 -1 -1 -2 -3 ).
Original vector v3 with range sorted by the
 binary predicate mod_lesser is v3 = ( 0 0 -1 -1 1 -2 2 -3 3 4 ).
The upper_bound in v2 for the element with a value of 3 is: 4.
The upper_bound in v2 for the element with a value of 3 is: 2.
The upper_bound in v3 for the element with a value of 3 is: 4.
```

Requirements

Header: <algorithm>

See Also

Reference

Predicate Version of upper_bound

upper_bound (STL Samples)

Standard Template Library

Other Resources

<algorithm> Members

<bitset>

Defines the template class bitset and two supporting template functions for representing and manipulating fixed-size sequences of bits.

For a list of all members of this header, see <bitset> Members.

```
#include <bitset>
```

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<bitset> Members

Standard C++ Library Header Files

<bitset> Members

Reference

Operators

operator&	Performs a bitwise AND between two bitsets.
operator<<	Inserts a text representation of the bit sequence into the standard output stream.
operator>>	Inserts a text representation of the bit sequence into the standard output stream.
operator^	Performs a bitwise EXCLUSIVE-OR between two bitsets.
operator|	Performs a bitwise OR between two bitsets.

Classes

bitset Class	The template class describes a type of object that stores a sequence consisting of a fixed number of bits that provide a compact way of keeping flags for a set of items or conditions.

See Also

Reference

<bitset>

Thread Safety in the Standard C++ Library

<bitset> Operators

For information about the operators in <bitset>, see <bitset> Members.

operator& (<bitset>)

Performs a bitwise **AND** between two bitsets.

```
template <size_t size>
bitset<size) operator&(
   const bitset<size>& _Left,
   const bitset<size)& _Right
);
```

Parameters

_Left

The first of the two bitsets whose respective elements are to be combined with the bitwise **AND**.

_Right

The second of the two valarrays whose respective elements are to be combined with the bitwise **AND**.

Return Value

A bitset whose elements are the result of performing the **AND** operation on the corresponding elements of *_Left* and *_Right*.

Example

```
// bitset_and.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>
#include <string>

using namespace std;

int main()
{
   bitset<4> b1 ( string("0101") );
   bitset<4> b2 ( string("0011") );
   bitset<4> b3 = b1 & b2;
   cout << "bitset 1: " << b1 << endl;
   cout << "bitset 2: " << b2 << endl;
   cout << "bitset 3: " << b3 << endl;
}
```

Output

```
bitset 1: 0101
bitset 2: 0011
bitset 3: 0001
```

Requirements

Header: <bitset>

See Also

Other Resources

<bitset> Members

operator<< (<bitset>)

Inserts a text representation of the bit sequence into the output stream.

```
template<class CharType, class Traits, size_t N>
   basic_ostream<CharType, Traits>& operator<< (
      basic_ostream<CharType, Traits>& ostr,
      const bitset<N>& _Right
      );
```

Parameters

_Right

An object of type **bitset<N>** that is to be inserted into the output stream as a string.

Return Value

A text representation of the bit sequence in **ostr**.

Remarks

The template function overloads **operator<<**, allowing a bitset to be written out without first converting it into a string. The template function effectively executes:

ostr << *_Right*.to_string <**CharType**, **Traits**, **allocator**<**CharType**> > ( )

Example

```
// bitset_op_insert.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 9 );

   // bitset inserted into output stream directly
   cout << "The ordered set of bits in the bitset<5> b1(9)"
        << "\n can be output with the overloaded << as: ( "
        << b1 << " )" << endl;

   // Compare converting bitset to a string before
   // inserting it into the output steam
   string s1;
   s1 =  b1.template to_string<char,
      char_traits<char>, allocator<char> >( );
   cout << "The string returned from the bitset b1"
        << "\n by the member function to_string( ) is: "
        << s1 << "." << endl;
}
```

Output

```
The ordered set of bits in the bitset<5> b1(9)
 can be output with the overloaded << as: ( 01001 )
The string returned from the bitset b1
 by the member function to_string( ) is: 01001.
```

Requirements

Header: <bitset>

See Also

Other Resources

<bitset> Members

operator>> (<bitset>)

Reads a string of bit characters into a bitset.

```
template<class CharType, class Traits, size_t Bits>
   basic_istream<CharType, Traits>& operator>> (
      basic_istream<CharType, Traits>& _Istr,
      bitset<N>& _Right
   );
```

Parameters

_Istr

The string that is entered into the input stream to be inserted into the bitset.

_Right

The bitset that is receiving the bits from the input stream.

Return Value

The template function returns the string *_Istr*.

Remarks

The template function overloads **operator>>** to store in the bitset *_Right* the value bitset(*str*), where *str* is an object of type basic_string<**CharType**, **Traits**, **allocator**<**CharType**> >**&** extracted from *_Istr*.

The template function extracts elements from *_Istr* and inserts them into the bitset until:

- All the bit elements have been extracted from the input stream and stored in the bitset.
- The bitset is filled up with bits from the input stream.
- An input element is encountered that is neither 0 nor 1.

Example

```
#include <bitset>
#include <iostream>
#include <string>

using namespace std;
int main()
{

   bitset<5> b1;
   cout << "Enter string of (0 or 1) bits for input into bitset<5>.\n"
        << "Try bit string of length less than or equal to 5,\n"
        << " (for example: 10110): ";
   cin >>  b1;

   cout << "The ordered set of bits entered from the "
        << "keyboard\n has been input into bitset<5> b1 as: ( "
        << b1 << " )" << endl;

   // Truncation due to longer string of bits than length of bitset
   bitset<2> b3;
   cout << "Enter string of bits (0 or 1) for input into bitset<2>.\n"
        << " Try bit string of length greater than 2,\n"
        << " (for example: 1011): ";
   cin >>  b3;

   cout << "The ordered set of bits entered from the "
        << "keyboard\n has been input into bitset<2> b3 as: ( "
        << b3 << " )" << endl;
```

```
    // Flushing the input stream
    char buf[100];
    cin.getline(&buf[0], 99);

    // Truncation with non-bit value
    bitset<5> b2;
    cout << "Enter a string for input into  bitset<5>.\n"
         << " that contains a character than is NOT a 0 or a 1,\n "
         << " (for example: 10k01): ";
    cin >>  b2;

    cout << "The string entered from the keyboard\n"
         << " has been input into bitset<5> b2 as: ( "
         << b2 << " )" << endl;
}
```

Input

```
10110
1011
10k10
```

Requirements

Header: <bitset>

See Also

Other Resources

<bitset> Members

operator^ (<bitset>)

Performs a bitwise **EXCLUSIVE-OR** between two bitsets.

```
template <size_t size>
bitset<size) operator^(
   const bitset<size>& _Left,
   const bitset<size)& _Right
);
```

Parameters

_Left

The first of the two bitsets whose respective elements are to be combined with the bitwise **EXCLUSIVE-OR**.

_Right

The second of the two valarrays whose respective elements are to be combined with the bitwise **EXCLUSIVE-OR**.

Return Value

A bitset whose elements are the result of performing the **EXCLUSIVE-OR** operation on the corresponding elements of *_Left* and *_Right*.

Example

```
// bitset_xor.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>
#include <string>

using namespace std;

int main()
{
   bitset<4> b1 ( string("0101") );
   bitset<4> b2 ( string("0011") );
   bitset<4> b3 = b1 ^ b2;
   cout << "bitset 1: " << b1 << endl;
   cout << "bitset 2: " << b2 << endl;
   cout << "bitset 3: " << b3 << endl;
}
```

Output

```
bitset 1: 0101
bitset 2: 0011
bitset 3: 0110
```

Requirements

Header: <bitset>

See Also

Other Resources

<bitset> Members

operator| (<bitset>)

Performs a bitwise **OR** between two bitsets.

```
template <size_t size>
bitset<size) operator|(
   const bitset<size>& _Left,
   const bitset<size)& _Right
);
```

Parameters

_Left

The first of the two bitsets whose respective elements are to be combined with the bitwise **OR**.

_Right

The second of the two valarrays whose respective elements are to be combined with the bitwise **OR**.

Return Value

A bitset whose elements are the result of performing the **OR** operation on the corresponding elements of *_Left* and *_Right*.

Example

```
// bitset_or.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>
#include <string>

using namespace std;

int main()
{
   bitset<4> b1 ( string("0101") );
   bitset<4> b2 ( string("0011") );
   bitset<4> b3 = b1 | b2;
   cout << "bitset 1: " << b1 << endl;
   cout << "bitset 2: " << b2 << endl;
   cout << "bitset 3: " << b3 << endl;
}
```

Output

```
bitset 1: 0101
bitset 2: 0011
bitset 3: 0111
```

Requirements

Header: <bitset>

See Also

Other Resources

<bitset> Members

<bitset> Classes

For information about the classes in <bitset>, see <bitset> Members.

bitset Class

Describes a type of object that stores a sequence consisting of a fixed number of bits that provide a compact way of keeping flags for a set of items or conditions. The bitset class supports operations on objects of type bitset that contain a collection of bits and provide constant-time access to each bit.

For a list of all members of this type, see bitset Members.

```
template <size_t N>
   class bitset
```

Parameters

N

Specifies the number of bits in the bitset object with a nonzero integer of type **size_t** that must be known at compile time.

Remarks

Unlike the similar vector<bool> Class, the bitset class does not have iterators and is not an Standard Template Library container. It also differs from vector<bool> by being of some specific size that is fixed at compile time in accordance with the size specified by the template parameter **N** when the **bitset<N>** is declared.

A bit is set if its value is 1 and reset if its value is 0. To flip or toggle a bit is to change its value from 1 to 0 or from 0 to 1. The **N** bits in a bitset are indexed by integer values from 0 to **N** - 1, where 0 indexes the first bit position and **N** - 1 the final bit position.

Requirements

Header: <bitset>

See Also

Other Resources

bitset Members

<bitset> Members

bitset Members

Reference

Constructors

bitset	Constructs an object of class **bitset<N>** and initializes the bits to zero, to some specified value, or to values obtained from characters in a string.

Typedefs

element_type	A type that is a synonym for the data type **bool** and can be used to reference element bits in a **bitset**.

Member Functions

any	The member function tests whether any bit in the sequence is set to 1.
count	The member function returns the number of bits set in the bit sequence.
flip	Toggles the value of all the bits in a **bitset** or toggles a single bit at a specified position.
none	Tests if no bit has been set to 1 in a **bitset** object.
reset	Resets all the bits in a **bitset** to 0 or resets a bit at a specified position to 0.
set	Sets all the bits in a **bitset** to 1 or sets a bit at a specified position to 1.
size	Returns the number of bits in a **bitset** object.
test	Tests whether the bit at a specified position in a **bitset** is set to 1.
to_string	Converts a **bitset** object to a string representation.
to_ulong	Converts a **bitset** object to the integer that would generate the sequence of bits contained if used to initialize the **bitset**.

Member Classes

reference	A proxy class that provides references to bits contained in a **bitset** that is used to access and manipulate the individual bits as a helper class for the **operator[]** of class **bitset**.

Operators

operator!=	Tests a target **bitset** for inequality with a specified **bitset**.
operator&=	Performs a bitwise combination of bitsets with the logical **AND** operation.
operator<<	Shifts the bits in a **bitset** to the left a specified number of positions and returns the result to a new **bitset**.
operator<<=	Shifts the bits in a **bitset** to the left a specified number of positions and returns the result to the targeted **bitset**.
operator==	Tests a target **bitset** for equality with a specified **bitset**.
operator>>	Shifts the bits in a **bitset** to the right a specified number of positions and returns the result to a new **bitset**.
operator>>=	Shifts the bits in a **bitset** to the right a specified number of positions and returns the result to the targeted **bitset**.

operator[]	Returns a reference to a bit at a specified position in a **bitset** if the **bitset** is modifiable; otherwise, it returns the value of the bit at that position.
operator^=	Performs a bitwise combination of bitsets with the exclusive **OR** operation.
operator|=	Performs a bitwise combination of bitsets with the inclusive **OR** operation.
operator~	Toggles all the bits in a target **bitset** and returns the result.

See Also

Reference

bitset Class

bitset Typedefs

For information about the typedefs in the bitset class, see bitset Members.

bitset::element_type

A type that is a synonym for the data type **bool** and can be used to reference element bits in a bitset.

```
typedef bool element_type;
```

Example

```
// bitset_elem_type.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<3> b1 ( 2 );
   cout << "Original bitset b1(6) is: ( "<< b1 << " )"
        << endl;

   //Compare two ways to reference bits in a bitset
   bool b;
   bitset<5>::element_type e;

   b = b1.test ( 2 );
   if ( b )
      cout << "The bit at position 2 of bitset b1"
           << "has a value of 1." << endl;
   else
      cout << "The bit at position 2 of bitset b1"
           << "has a value of 0." << endl;
   b1[2] = 1;
   cout << "Bitset b1 modified by b1[2] = 1 is: ( "<< b1 << " )"
        << endl;

   e = b1.test ( 2 );
   if ( e )
      cout << "The bit at position 2 of bitset b1"
           << "has a value of 1." << endl;
   else
      cout << "The bit at position 2 of bitset b1"
           << "has a value of 0." << endl;
}
```

Output

```
Original bitset b1(6) is: ( 010 )
The bit at position 2 of bitset b1has a value of 0.
Bitset b1 modified by b1[2] = 1 is: ( 110 )
The bit at position 2 of bitset b1has a value of 1.
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset Member Functions

For information about the member functions in the bitset class, see bitset Members.

bitset::any

Tests whether any bit in the sequence is set to 1.

```
bool any( ) const;
```

Return Value

true if any bit in the bitset is set to 1; **false** if all the bits are 0.

Example

```
// bitset_any.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 6 );
   bool b, rb;

   cout << "The original bitset b1( 6 ) is: ( "<< b1 << " )"
        << endl;

   b = b1.any ( );

   if ( b )
      cout << "At least one of the bits in bitset is set to 1."
           << endl;
   else
      cout << "None of the bits in bitset are set to 1."
           << endl;

   bitset<5> rb1;
   rb1 = b1.reset ( );

   cout << "The reset bitset is: ( "<< b1 << " )"
        << endl;

   rb = rb1.any ( );

   if ( rb )
      cout << "At least one of the bits in the reset bitset "
           << "are set to 1." << endl;
   else
      cout << "None of the bits in bitset b1 are set to 1."
           << endl;
}
```

Output

```
The original bitset b1( 6 ) is: ( 00110 )
At least one of the bits in bitset is set to 1.
The reset bitset is: ( 00000 )
None of the bits in bitset b1 are set to 1.
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset::bitset

Constructs an object of class **bitset<N>** and initializes the bits to zero, or to some specified value, or to values obtained from characters in a string.

```
bitset( );
bitset(
   unsigned long _Val
);
template<class CharType, class Traits, class Allocator>
   explicit bitset(
      const basic_string<CharType, Traits, Allocator>& _Str,
      typename basic_string<CharType, Traits, Allocator>::size_type
                  _Pos = 0,
      typename basic_string<CharType, Traits, Allocator>::size_type
                  _Count = basic_string<CharType, Traits, Alloc>::npos
      );
```

Parameters

_Val

The positive integer whose base two representation is used to initialize the bits in the bitset being constructed.

_Str

The string of zeros and ones used to initialize the bitset bit values.

_Pos

The position of the character in the string, counting from left to right and beginning with zero, used to initialize the first bit in the bitset.

_Count

The number of characters in the string used to provide initial values for the bits in the bitset.

Remarks

Three member functions can be used to construct objects of class **bitset<N>**:

- The first member function constructs an object of class **bitset<N>** and initializes all **N** bits to a default value of zero.
- The second member function constructs an object of class **bitset<N>**, initializing the **N** bits to values that correspond to a base two representation of the positive integer specified in *_Val*. If the length of the base two integer exceeds the number of bits **N** in the bitset, then the first **N** places of the base two integer are used to initialize the bitset bit values. If the number of bits in the bitset exceeds the length of the base two number, then the additional bits are initialized to the default value of zero.
- The third member function constructs an object of class **bitset<N>** and initializes bits from the characters provided in a string of zeroes and ones. If any characters of the string are other than 0 or 1, the constructor throws an object of class invalid argument. If the position specified is beyond the length of the string, then the constructor throws an object of class out_of_range, which is available when the <stdexcept> header is included. The constructor sets only those bits at position *j* in the bitset for which the character in the string at position *pos* **+** *j* is 1.

Example

```
// bitset_bitset.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   // Using the default constructor
   using namespace std;
```

```
    bitset<2> b0;
    cout << "The set of bits in bitset<2> b0 is: ( "
         << b0 << " )." << endl;

    // Using the second member function
    bitset<5> b1 ( 6 );
    cout << "The set of bits in bitset<5> b1( 6 ) is: ( "
         << b1 << " )." << endl;

    // The template parameter N can be an expresssion
    bitset< 2 * sizeof ( int ) > b2;
    cout << "The set of bits in bitset<2 * sizeof ( int ) > b2 is: ( "
         << b2 << " )." << endl;

    // The base two representation will be truncated
    // if its length exceeds the size of the bitset
    bitset<3> b3 ( 6 );
    cout << "The set of bits in bitset<3> b3( 6 ) is ( "
         << b3 << " )." << endl;

    // Using the third member function with the first parameter
    string bitval4 ( "10011" );
    bitset<5> b4 ( bitval4 );
    cout << "The set of bits in bitset<5> b4( bitval4 ) is ( "
         << b4 << " )." << endl;

    // Only part of the string may be used for initialization

    // Starting at position 3 for a length of 6 (100110)
    string bitval5 ("11110011011");
    bitset<6> b5 ( bitval5, 3, 6 );
    cout << "The set of bits in bitset<11> b5( bitval, 3, 6 ) is ( "
         << b5 << " )." << endl;

    // The bits not initialized with part of the string
    // will default to zero
    bitset<11> b6 ( bitval5, 3, 5 );
    cout << "The set of bits in bitset<11> b6( bitval5, 3, 5 ) is ( "
         << b6 << " )." << endl;

    // Starting at position 2 and continue to the end of the string
    bitset<9> b7 ( bitval5, 2 );
    cout << "The set of bits in bitset<9> b7( bitval, 2 ) is ( "
         << b7 << " )." << endl;
}
```

Output

```
The set of bits in bitset<2> b0 is: ( 00 ).
The set of bits in bitset<5> b1( 6 ) is: ( 00110 ).
The set of bits in bitset<2 * sizeof ( int ) > b2 is: ( 00000000 ).
The set of bits in bitset<3> b3( 6 ) is ( 110 ).
The set of bits in bitset<5> b4( bitval4 ) is ( 10011 ).
The set of bits in bitset<11> b5( bitval, 3, 6 ) is ( 100110 ).
The set of bits in bitset<11> b6( bitval5, 3, 5 ) is ( 00000010011 ).
The set of bits in bitset<9> b7( bitval, 2 ) is ( 110011011 ).
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset::count

Returns the number of bits set in the bit sequence.

```
size_t count( ) const;
```

Return Value

The number of bits set in the bit sequence.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// bitset_count.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;

    bitset<5> b1(4);

    cout << "The collection of bits in the original bitset is: ( "
         << b1 << " )" << endl;

    size_t i;
    i = b1.count();
    cout << "The number of bits in the bitset set to 1 is: "
         << i << "." << endl;

    bitset<5> fb1;
    fb1 = b1.flip();

    cout << "The collection of flipped bits in the modified bitset "
         << "is: ( " << b1 << " )" << endl;

    size_t ii;
    ii = fb1.count();
    cout << "The number of bits in the bitset set to 1 is: "
         << ii << "." << endl;
}
```

Output

```
The collection of bits in the original bitset is: ( 00100 )
The number of bits in the bitset set to 1 is: 1.
The collection of flipped bits in the modified bitset is: ( 11011 )
The number of bits in the bitset set to 1 is: 4.
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset::flip

Toggles the value of all the bits in a bitset or toggles a single bit at a specified position.

```
bitset<N>& flip( );
bitset<N>& flip(
   size_t _Pos
);
```

Parameters

_Pos

The position of the bit whose value is to be toggled.

Return Value

A copy of the modified bitset for which the member function was invoked.

Remarks

The second member function throws an out_of_range exception if the position specified as a parameter is greater than the size *N* of the **bitset<***N***>** whose bit was toggled.

Example

```
// bitset_flip.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;
   bitset<5> b1 ( 6 );

   cout << "The collection of bits in the original bitset is: ( "
        << b1 << " )" << endl;

   bitset<5> fb1;
   fb1 = b1.flip ( );

   cout << "After flipping all the bits, the bitset becomes: ( "
        << fb1 << " )" << endl;

   bitset<5> f3b1;
   f3b1 = b1.flip ( 3 );

   cout << "After flipping the fourth bit, the bitset becomes: ( "
        << f3b1 << " )" << endl << endl;

   bitset<5> b2;
   int i;
   for ( i = 0 ; i <= 4 ; i++ )
   {
      b2.flip(i);
      cout << b2 << "  The bit flipped is in position "
           << i << ".\n";
   }
}
```

Output

```
The collection of bits in the original bitset is: ( 00110 )
After flipping all the bits, the bitset becomes: ( 11001 )
After flipping the fourth bit, the bitset becomes: ( 10001 )
```

```
00001  The bit flipped is in position 0.
00011  The bit flipped is in position 1.
00111  The bit flipped is in position 2.
01111  The bit flipped is in position 3.
11111  The bit flipped is in position 4.
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset::none

Tests if no bit has been set to 1 in a bitset object.

```
bool none( ) const;
```

Return Value

true if no bit in the bitset has been set to 1; **false** if at least one bit has been set to 1.

Example

```
// bitset_none.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 6 );
   bool b, rb;

   cout << "Original bitset b1(6) is: ( " << b1 << " )"
        << endl;

   b = b1.none ( );

   if ( b )
      cout << "None of the bits in bitset b1 are set to 1."
           << endl;
   else
      cout << "At least one of the bits in bitset b1 is set to 1."
           << endl;

   bitset<5> rb1;
   rb1 = b1.reset ( );
   rb = rb1.none ( );
   if ( rb )
      cout << "None of the bits in bitset b1 are set to 1."
           << endl;
   else
      cout << "At least one of the bits in bitset b1 is set to 1."
           << endl;
}
```

Output

```
Original bitset b1(6) is: ( 00110 )
At least one of the bits in bitset b1 is set to 1.
None of the bits in bitset b1 are set to 1.
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset::reset

Resets all the bits in a bitset to 0 or resets a bit at a specified position to 0.

```
bitset<N>& reset( );
bitset<N>& reset(
   size_t _Pos
);
```

Parameters

_Pos

The position of the bit in the bitset to be reset to 0.

Return Value

A copy of the bitset for which the member function was invoked.

Remarks

The second member function throws an out_of_range exception if the position specified is greater than the size of the bitset.

Example

```
// bitset_reset.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 13 );
   cout << "The set of bits in bitset<5> b1(13) is: ( "<< b1 << " )"
        << endl;

   bitset<5> b1r3;
   b1r3 = b1.reset( 2 );
   cout << "The collecion of bits obtained from resetting the\n"
        << " third bit of bitset b1 is: ( "<< b1r3 << " )"
        << endl;

   bitset<5> b1r;
   b1r = b1.reset( );
   cout << "The collecion of bits obtained from resetting all\n"
        << " the elements of the bitset b1 is: ( "<< b1r << " )"
        << endl;
}
```

Output

```
The set of bits in bitset<5> b1(13) is: ( 01101 )
The collecion of bits obtained from resetting the
 third bit of bitset b1 is: ( 01001 )
The collecion of bits obtained from resetting all
 the elements of the bitset b1 is: ( 00000 )
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset::set

Sets all the bits in a bitset to 1 or sets a bit at a specified position to 1.

```
bitset<N>& set( );
bitset<N>& set(
   size_t _Pos,
   bool _Val = true
);
```

Parameters

_Pos

The position of the bit in the bitset to be set to assigned a value.

_Val

The value to be assigned to the bit at the position specified.

Return Value

A copy of the bitset for which the member function was invoked.

Remarks

The second member function throws an out_of_range exception if the position specified is greater than the size of the bitset.

Example

```
// bitset_set.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 6 );
   cout << "The set of bits in bitset<5> b1(6) is: ( "<< b1 << " )"
        << endl;

   bitset<5> b1s0;
   b1s0 = b1.set( 0 );
   cout << "The collecion of bits obtained from setting the\n"
        << " zeroth bit of bitset b1 is: ( "<< b1s0 << " )"
        << endl;

   bitset<5> bs1;
   bs1 = b1.set( );
   cout << "The collecion of bits obtained from setting all the\n"
        << " elements of the bitset b1 is: ( "<< bs1 << " )"
        << endl;
}
```

Output

```
The set of bits in bitset<5> b1(6) is: ( 00110 )
The collecion of bits obtained from setting the
 zeroth bit of bitset b1 is: ( 00111 )
The collecion of bits obtained from setting all the
 elements of the bitset b1 is: ( 11111 )
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset::size

Returns the number of bits in a bitset object.

```
size_t size( ) const;
```

Return Value

The number of bits, *N*, in a **bitset<***N***>**.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// bitset_size.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main()
{
    using namespace std;

    bitset<5> b1(6);
    size_t i;

    cout << "The set of bits in bitset<5> b1( 6 ) is: ( "<< b1 << " )"
         << endl;

    i = b1.size();

    cout << "The number of bits in bitset b1 is: " << i << "."
         << endl;
}
```

Output

```
The set of bits in bitset<5> b1( 6 ) is: ( 00110 )
The number of bits in bitset b1 is: 5.
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset::test

Tests whether the bit at a specified position in a bitset is set to 1.

```
bool test(
   size_t _Pos,
);
```

Parameters

_Pos

The position of the bit in the bitset to be tested for its value.

Return Value

true if the bit specified by the argument position is set to 1; otherwise, **false**.

Remarks

The member function throws an out_of_range exception that is the size of the bitset is less than or equal to the position specified in the argument.

Example

```
// bitset_test.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 6 );
   bool b;
   bitset<5>::element_type e;

   cout << "Original bitset b1(6) is: ( "<< b1 << " )"
        << endl;
   cout << "Note: positions in a bitset are numbered\n"
         << " from the right starting with 0.\n" << endl;

   b = b1.test ( 3 );

   if ( b )
      cout << "The bit at position 3 of bitset b1 "
           << "has a value of 1." << endl;
   else
      cout << "The bit at position 3 of bitset b1 "
           << "has a value of 0." << endl;

   b1[3] = 1;
   cout << "Bitset b1 modified by b1[3] = 1 is: ( "<< b1 << " )"
        << endl;

   e = b1.test ( 3 );
   if ( e )
      cout << "The bit at position 3 of bitset b1 "
           << "has a value of 1." << endl;
   else
      cout << "The bit at position 3 of bitset b1 "
           << "has a value of 0." << endl;
}
```

Output

```
Original bitset b1(6) is: ( 00110 )
Note: positions in a bitset are numbered
 from the right starting with 0.

The bit at position 3 of bitset b1 has a value of 0.
Bitset b1 modified by b1[3] = 1 is: ( 01110 )
The bit at position 3 of bitset b1 has a value of 1.
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset::to_string

Converts a bitset object to a string representation.

```
template<class CharType, class Traits, class Alloc>
   basic_string<CharType, Traits, Alloc> to_string ( ) const;
```

Return Value

A string object of class basic_string, where each bit set in the bitset has a corresponding character of 1, and a character of 0 if the bit is unset.

Example

```
// bitset_to_string.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 7 );

   cout << "The ordered set of bits in the bitset<5> b1( 7 )"
        << "\n that was generated by the number 7 is: ( "
        << b1 << " )" << endl;

   string s1;
   s1 =  b1.template to_string<char,
   char_traits<char>, allocator<char> >( );
   cout << "The string returned from the bitset b1"
        << "\n by the member function to_string( ) is: "
        << s1 << "." << endl;
}
```

Output

```
The ordered set of bits in the bitset<5> b1( 7 )
 that was generated by the number 7 is: ( 00111 )
The string returned from the bitset b1
 by the member function to_string( ) is: 00111.
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset::to_ulong

Converts a bitset object to the integer that would generate the sequence of bits contained if used to initialize the bitset.

```
unsigned long to_ulong( ) const;
```

Return Value

An integer that would generate the bits in a bitset if used in the initialization of the bitset.

Remarks

Applying the member function would return the integer that has the same sequence of 1 and 0 digits as is found in sequence of bits contained in the bitset.

The member function throws **overflow_error** if any bit in the bit sequence has a bit value that cannot be represented as a value of type **unsigned long**.

Example

```
// bitset_to_ulong.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 7 );

   cout << "The ordered set of bits in the bitset<5> b1( 7 )"
        << "\n that was generated by the number 7 is: ( "
        << b1 << " )" << endl;

   unsigned long int i;
   i = b1.to_ulong( );
   cout << "The integer returned from the bitset b1,"
        << "\n by the member function to_long( ), that"
        << "\n generated the bits as a base two number is: "
        << i << "." << endl;
}
```

Output

```
The ordered set of bits in the bitset<5> b1( 7 )
 that was generated by the number 7 is: ( 00111 )
The integer returned from the bitset b1,
 by the member function to_long( ), that
 generated the bits as a base two number is: 7.
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

Member Classes

For more information on the member classes in the bitset class, see bitset Class.

bitset::reference

A proxy class that provides references to bits contained in a bitset that is used to access and manipulate the individual bits as a helper class for the **operator[]** of class bitset.

```
class reference {
   friend class bitset<N>;
   public:
      reference& operator=(
         bool _Val
            );
      reference& operator=(
         const reference& _Bitref
            );
      bool operator~( ) const;
      operator bool( ) const;
      reference& flip( );
};
```

Parameters

_Val

The value of the object of type **bool** to be assigned to a bit in a bitset.

_Bitref

A reference of the form *x [i]* to the bit at position *i* in bitset *x*.

Return Value

A reference to the bit in the bitset specified by the argument position for the first, second, and fifth member functions of class reference, and **true** or **false**, to reflect the value of the modified bit in the bitset for the third and fourth member functions of class reference.

Remarks

The class reference exists only as a helper class for the bitset **operator[]**. The member class describes an object that can access an individual bit within a bitset. Let *b* be an object of type **bool**, *x* and *y* objects of type **bitset<***N***>**, and *i* and *j* valid positions within such an object. The notation *x [i]* references the bit at position *i* in bitset *x*. The member functions of class reference provide, in order, the following operations:

Operation	Definition
x[*i*] = *b*	Stores **bool** value *b* at bit position *i* in bitset *x*.
x[*i*] = *y*[*j*]	Stores the value of the bit *y*[*j*] at bit position *i* in bitset *x*.
b = ~*x*[*i*]	Stores the flipped value of the bit *x*[*i*] in **bool** *b*.
b = *x*[*i*]	Stores the value of the bit *x*[*i*] in **bool** *b*.
x[*i*].**flip**( )	Stores the flipped value of the bit *x*[*i*] back at bit position *i* in *x*.

Example

```
// bitset_reference.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;
```

```
    bitset<5> b1 ( 2 );
    bitset<5> b2 ( 6 );
    cout << "The initialized bitset<5> b1( 2 ) is: ( "<< b1 << " )."
         << endl;
    cout << "The initialized bitset<5> b2( 6 ) is: ( "<< b2 << " )."
         << endl;

    // Example of x [i] = b storing bool b at bit position i
    // in bitset x
    b1[ 0 ] = true;
    cout << "The bitset<5> b1 with the bit at position 0 set to 1"
         << " is: ( "<< b1 << " )" << endl;

    // Example of x [i] = y [j] storing the bool value of the
    // bit at position j in bitset y at bit position i in bitset x
    b2 [4] = b1 [0];      // b1 [0] = true
    cout << "The bitset<5> b2 with the bit at position 4 set to the "
         << "value\n of the bit at position 0 of the bit in "
         << "bitset<5> b1 is: ( "<<  b2  << " )" << endl;

    // Example of b = ~x [i] flipping the value of the bit at
    // position i of bitset x and storing the value in an
    // object b of type bool
    bool b = ~b2 [4];      // b2 [4] = false
    if ( b )
       cout << "The value of the object b = ~b2 [4] "
            << "of type bool is true." << endl;
    else
       cout << "The value of the object b = ~b2 [4] "
            << "of type bool is false." << endl;

    // Example of b = x [i] storing the value of the bit at
    // position i of bitset x in the object b of type bool
    b = b2 [4];
    if ( b )
       cout << "The value of the object b = b2 [4] "
            << "of type bool is true." << endl;
    else
       cout << "The value of the object b = b2 [4] "
            << "of type bool is false." << endl;

    // Example of x [i] . flip ( ) toggling the value of the bit at
    // position i of bitset x
    cout << "Before flipping the value of the bit at position 4 in "
         << "bitset b2,\n it is ( "<<  b2  << " )." << endl;
    b2 [4].flip( );
    cout << "After flipping the value of the bit at position 4 in "
         << "bitset b2,\n it becomes ( "<<  b2  << " )." << endl;
    bool c;
    c = b2 [4].flip( );
    cout << "After a second toggle, the value of the position 4"
         << " bit in b2 is now: " << c << ".";
  }
```

Output

```
The initialized bitset<5> b1( 2 ) is: ( 00010 ).
The initialized bitset<5> b2( 6 ) is: ( 00110 ).
The bitset<5> b1 with the bit at position 0 set to 1 is: ( 00011 )
The bitset<5> b2 with the bit at position 4 set to the value
 of the bit at position 0 of the bit in bitset<5> b1 is: ( 10110 )
The value of the object b = ~b2 [4] of type bool is false.
The value of the object b = b2 [4] of type bool is true.
Before flipping the value of the bit at position 4 in bitset b2,
 it is ( 10110 ).
After flipping the value of the bit at position 4 in bitset b2,
 it becomes ( 00110 ).
After a second toggle, the value of the position 4 bit in b2 is now: 1.
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Thread Safety in the Standard C++ Library

Other Resources

bitset Members

bitset Operators

For information about the operators in the bitset class, see bitset Class.

bitset::operator!=

Tests a target bitset for inequality with a specified bitset.

```
bool operator !=(
   const bitset<N>& _Right
) const;
```

Parameters

_Right

The bitset that is to be compared to the target bitset for inequality.

Return Value

true if the bitsets are different; **false** if they are the same.

Remarks

Bitsets must be of the same size to be tested for inequality by the member operator function.

Example

```
// bitset_op_NE.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 7 );
   bitset<5> b2 ( 7 );
   bitset<5> b3 ( 2 );
   bitset<4> b4 ( 7 );

   if ( b1 != b2 )
      cout << "Bitset b1 is different from bitset b2." << endl;
   else
      cout << "Bitset b1 is the same as bitset b2." << endl;

   if ( b1 != b3 )
      cout << "Bitset b1 is different from bitset b3." << endl;
   else
      cout << "Bitset b1 is the same as bitset b3." << endl;

   // This would cause an error because bitsets must have the
   // same size to be tested
   // if ( b1 != b4 )
   //   cout << "Bitset b1 is different from bitset b4." << endl;
   // else
   //   cout << "Bitset b1 is the same as bitset b4." << endl;
}
```

Output

```
Bitset b1 is the same as bitset b2.
Bitset b1 is different from bitset b3.
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset::operator&=

Performs a bitwise combination of bitsets with the logical **AND** operation.

```
bitset<N>& operator&=(
   const bitset<N>& _Right
);
```

Parameters

_Right

The bitset that is to be combined bitwise with the target bitset.

Return Value

The modified target bitset that results from the bitwise **AND** operation with the bitset specified as a parameter.

Remarks

Two bits combined by the **AND** operator return **true** if each bit is true; otherwise, their combination returns **false**.

Bitsets must be of the same size to be combined bitwise with the **AND** operator by the member operator function.

Example

```
// bitset_op_bitwise.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 7 );
   bitset<5> b2 ( 11 );
   bitset<4> b3 ( 7 );

   cout << "The target bitset b1 is:    ( "<< b1 << " )." << endl;
   cout << "The parameter bitset b2 is: ( "<< b2 << " )." << endl;
   cout << endl;

   b1 &= b2;
   cout << "After bitwise AND combination,\n"
        << " the target bitset b1 becomes:   ( "<< b1 << " )."
        << endl;

   // Note that the parameter-specified bitset is unchanged
   cout << "The parameter bitset b2 remains: ( "<< b2 << " )."
        << endl;

   // The following would cause an error because the bisets
   // must be of the same size to be combined
   // b1 &= b3;
}
```

Output

```
The target bitset b1 is:    ( 00111 ).
The parameter bitset b2 is: ( 01011 ).

After bitwise AND combination,
 the target bitset b1 becomes:   ( 00011 ).
The parameter bitset b2 remains: ( 01011 ).
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset::operator<<

Shifts the bits in a bitset to the left a specified number of positions and returns the result to a new bitset.

```
bitset<N> operator<<(
   const bitset<N>& _Pos
);
```

Parameters

_Pos

The number of positions to the left that the bits in the bitset are to be shifted.

Return Value

The modified bitset with the bits shifted to the left the required number of positions.

Remarks

The member operator function returns **bitset**(***this**) **<<= pos,** where <<= shifts the bits in a bitset to the left a specified number of positions and returns the result to the targeted bitset.

Example

```
// bitset_op_LS.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 7 );

   cout << "The bitset b1 is: ( "<< b1 << " )." << endl;

   bitset<5> b2;
   b2 = b1 << 2;

   cout << "After shifting the bits 2 positions to the left,\n"
        << " the bitset b2 is: ( "<< b2 << " )."
        << endl;

   bitset<5> b3 = b2 >> 1;

   cout << "After shifting the bits 1 position to the right,\n"
        << " the bitset b3 is: ( " << b3 << " )."
        << endl;
}
```

Output

```
The bitset b1 is: ( 00111 ).
After shifting the bits 2 positions to the left,
 the bitset b2 is: ( 11100 ).
After shifting the bits 1 position to the right,
 the bitset b3 is: ( 01110 ).
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset::operator<<=

Shifts the bits in a bitset to the left a specified number of positions and returns the result to the targeted bitset.

```
bitset<N>& operator<<=(
   const bitset<N>& _Pos
);
```

Parameters

_Pos

The number of positions to the left the bits in the bitset are to be shifted.

Return Value

The targeted bitset modified so that the bits have been shifted to the left the required number of positions.

Remarks

If no element exists to shift into the position, the function clears the bit to a value of 0.

Example

```
// bitset_op_LSE.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;
   bitset<5> b1 ( 7 );
   cout << "The target bitset b1 is: ( "<< b1 << " )." << endl;
   b1 <<= 2;
   cout << "After shifting the bits 2 positions to the left,\n"
        << " the target bitset b1 becomes: ( "<< b1 << " )."
        << endl;
}
```

Output

```
The target bitset b1 is: ( 00111 ).
After shifting the bits 2 positions to the left,
 the target bitset b1 becomes: ( 11100 ).
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset::operator==

Tests a target bitset for equality with a specified bitset.

```
bool operator ==(
   const bitset<N>& _Right
) const;
```

Parameters

_Right

The bitset that is to be compared to the target bitset for equality.

Return Value

true if the bitsets are the same; **false** if they are different.

Remarks

Bitsets must be of the same size to be tested for equality by the member operator function.

Example

```
// bitset_op_EQ.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;
   bitset<5> b1 ( 7 );
   bitset<5> b2 ( 7 );
   bitset<5> b3 ( 2 );
   bitset<4> b4 ( 7 );

   if ( b1 == b2 )
      cout << "Bitset b1 is the same as bitset b2." << endl;
   else
      cout << "Bitset b1 is different from bitset b2." << endl;

   if ( b1 == b3 )
      cout << "Bitset b1 is the same as bitset b3." << endl;
   else
      cout << "Bitset b1 is different from bitset b3." << endl;

   // This would cause an error because bitsets must have the
   // same size to be tested
   // if ( b1 == b4 )
   //   cout << "Bitset b1 is the same as bitset b4." << endl;
   // else
   //   cout << "Bitset b1 is different from bitset b4." << endl;
}
```

Output

```
Bitset b1 is the same as bitset b2.
Bitset b1 is different from bitset b3.
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset::operator>>

Shifts the bits in a bitset to the right a specified number of positions and returns the result to a new bitset.

```
bitset<N> operator>>(
   const bitset<N>& _Pos
);
```

Parameters

_Pos

The number of positions to the right the bits in the bitset are to be shifted.

Return Value

A new bitset where the bits have been shifted to the right the required number of positions relative to the targeted bitset.

Example

```
// bitset_op_RS.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;
   bitset<5> b1 ( 7 );
   cout << "The bitset b1 is: ( "<< b1 << " )." << endl;

   bitset<5> b2;
   b2 = b1 << 2;

   cout << "After shifting the bits 2 positions to the left,\n"
        << " the bitset b2 is: ( "<< b2 << " )."
        << endl;
   bitset<5> b3 = b2 >> 1;

   cout << "After shifting the bits 1 position to the right,\n"
        << " the bitset b3 is: ( " << b3 << " )."
        << endl;
}
```

Output

```
The bitset b1 is: ( 00111 ).
After shifting the bits 2 positions to the left,
 the bitset b2 is: ( 11100 ).
After shifting the bits 1 position to the right,
 the bitset b3 is: ( 01110 ).
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset::operator>>=

Shifts the bits in a bitset to the right a specified number of positions and returns the result to the targeted bitset.

```
bitset<N>& operator>>=(
   const bitset<N>& _Pos
);
```

Parameters

_Pos

The number of positions to the right the bits in the bitset are to be shifted.

Return Value

The targeted bitset modified so that the bits have been shifted to the right the required number of positions.

Remarks

If no element exists to shift into the position, the function clears the bit to a value of 0.

Example

```
// bitset_op_RSE.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;
   bitset<5> b1 ( 28 );
   cout << "The target bitset b1 is: ( "<< b1 << " )." << endl;

   b1 >>= 2;
   cout << "After shifting the bits 2 positions to the right,\n"
        << " the target bitset b1 becomes: ( "<< b1 << " )."
        << endl;
}
```

Output

```
The target bitset b1 is: ( 11100 ).
After shifting the bits 2 positions to the right,
 the target bitset b1 becomes: ( 00111 ).
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset::operator[]

Returns a reference to a bit at a specified position in a bitset if the bitset is modifiable; otherwise, it returns the value of the bit at that position.

```
bool operator[](
   size_type _Pos
) const;
reference operator[](
   size_type _Pos
);
```

Parameters

_Pos

The position locating the bit within the bitset.

Remarks

When compiling with _SECURE_SCL 1, a runtime error will occur if you attempt to access an element outside the bounds of the bitset. See Checked Iterators for more information.

Example

```
// bitset_op_REF.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;
   bool b;
   bitset<5> b1 ( 6 );
   cout << "The initialized bitset<5> b1( 2 ) is: ( "<< b1 << " )."
        << endl;

   int i;
   for ( i = 0 ; i <= 4 ; i++ )
   {
      b = b1[ i ];
      cout << "  The bit in position "
           << i << " is " << b << ".\n";
   }
}
```

Output

```
The initialized bitset<5> b1( 2 ) is: ( 00110 ).
  The bit in position 0 is 0.
  The bit in position 1 is 1.
  The bit in position 2 is 1.
  The bit in position 3 is 0.
  The bit in position 4 is 0.
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset::operator^=

Performs a bitwise combination of bitsets with the exclusive **OR** operation.

```
bitset<N>& operator^=(
   const bitset<N>& _Right
);
```

Parameters

_Right

The bitset that is to be combined bitwise with the target bitset.

Return Value

The modified target bitset that results from the bitwise exclusive **OR** operation with the bitset specified as a parameter.

Remarks

Two bits combined by the exclusive **OR** operator return **true** if at least one, but not both, of the bits is **true**; otherwise, their combination returns **false**.

Bitsets must be of the same size to be combined bitwise with the exclusive **OR** operator by the member operator function.

Example

```
// bitset_op_bitwiseOR.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;
   bitset<5> b1 ( 7 );
   bitset<5> b2 ( 11 );
   bitset<4> b3 ( 7 );

   cout << "The target bitset b1 is:    ( "<< b1 << " )." << endl;
   cout << "The parameter bitset b2 is: ( "<< b2 << " )." << endl;
   cout << endl;

   b1 ^= b2;
   cout << "After bitwise exclusive OR combination,\n"
        << " the target bitset b1 becomes:   ( "<< b1 << " )."
        << endl;

   // Note that the parameter-specified bitset in unchanged
   cout << "The parameter bitset b2 remains: ( "<< b2 << " )."
        << endl;

   // The following would cause an error because the bisets
   // must be of the same size to be combined
   // b1 |= b3;
}
```

Output

```
The target bitset b1 is:    ( 00111 ).
The parameter bitset b2 is: ( 01011 ).

After bitwise exclusive OR combination,
 the target bitset b1 becomes:   ( 01100 ).
The parameter bitset b2 remains: ( 01011 ).
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset::operator|=

Performs a bitwise combination of bitsets with the inclusive **OR** operation.

```
bitset<N>& operator|=(
   const bitset<N>& _Right
);
```

Parameters

_Right

The bitset that is to be combined bitwise with the target bitset.

Return Value

The modified target bitset that results from the bitwise inclusive **OR** operation with the bitset specified as a parameter.

Remarks

Two bits combined by the inclusive **OR** operator return **true** if at least one of the bits is **true**; if both bits are **false**, their combination returns **false**.

Bitsets must be of the same size to be combined bitwise with the inclusive **OR** operator by the member operator function.

Example

```
// bitset_op_BIO.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 7 );
   bitset<5> b2 ( 11 );
   bitset<4> b3 ( 7 );

   cout << "The target bitset b1 is:    ( "<< b1 << " )." << endl;
   cout << "The parameter bitset b2 is: ( "<< b2 << " )." << endl;
   cout << endl;

   b1 |= b2;
   cout << "After bitwise inclusive OR combination,\n"
        << " the target bitset b1 becomes:   ( "<< b1 << " )."
        << endl;

   // Note that the parameter-specified bitset in unchanged
   cout << "The parameter bitset b2 remains: ( "<< b2 << " )."
        << endl;

   // The following would cause an error because the bisets
   // must be of the same size to be combined
   // b1 |= b3;
}
```

Output

```
The target bitset b1 is:    ( 00111 ).
The parameter bitset b2 is: ( 01011 ).

After bitwise inclusive OR combination,
 the target bitset b1 becomes:   ( 01111 ).
The parameter bitset b2 remains: ( 01011 ).
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

bitset::operator~

Toggles all the bits in a target bitset and returns the result.

```
bitset<N> operator~( );
```

Return Value

The bitset with all its bits toggled with respect to the targeted bitset.

Example

```
// bitset_op_toggle.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
#include <bitset>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 7 );
   bitset<5> b2;
   b2 = ~b1;

   cout << "Bitset b1 is: ( "<< b1 << " )." << endl;
   cout << "Bitset b2 = ~b1 is: ( "<< b2 << " )." << endl;

   // These bits could also be flipped using the flip member function
   bitset<5> b3;
   b3 = b1.flip( );
   cout << "Bitset b3 = b1.flip( ) is: ( "<< b2 << " )." << endl;
}
```

Output

```
Bitset b1 is: ( 00111 ).
Bitset b2 = ~b1 is: ( 11000 ).
Bitset b3 = b1.flip( ) is: ( 11000 ).
```

Requirements

Header: <bitset>

See Also

Reference

bitset Class

Other Resources

bitset Members

<cassert>

Effectively includes the Standard C library header <assert.h>.

```
#include <assert.h>
```

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

Other Resources

Standard C++ Library Header Files

<cctype>

Defines the macros traditionally defined in the Standard C library header <ctype.h>.

```
#if <TRADITIONAL C HEADERS>
   #include <ctype.h>
namespace std {
   using ::isalnum;
   using ::isalpha;
   using ::iscntrl;
   using ::isdigit;
   using ::isgraph;
   using ::islower;
   using ::isprint;
   using ::ispunct;
   using ::isspace;
   using ::isupper;
   using ::isxdigit;
   using ::tolower;
   using ::toupper;
}
#endif
```

Remarks

Including this header also ensures that the names declared with external linkage in the Standard C library header are declared in the **std** namespace. In this implementation, the names may or may not also be declared in the global namespace, depending on the specific translation environment.

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

Other Resources

Standard C++ Library Header Files

<cerrno>

Effectively includes the Standard C library header <errno.h>.

```
#include <errno.h>
namespace std {
   using ::errno;
}
```

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

Other Resources

Standard C++ Library Header Files

<cfloat>

Effectively includes the Standard C library header <float.h>.

```
#include <float.h>
```

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

Other Resources

Standard C++ Library Header Files

<ciso646>

Effectively includes the standard header <iso646.h>.

```
#include <iso646.h>
```

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

Other Resources

Standard C++ Library Header Files

<climits>

Effectively includes the Standard C library header <limits.h>.

```
#include <limits.h>
```

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

Other Resources

Standard C++ Library Header Files

<clocale>

Defines the macros traditionally defined in the Standard C library header <locale.h>.

```
#if <TRADITIONAL C HEADERS>
   #include <locale.h>
namespace std {
   using ::lconv;
   using ::localeconv;
   using ::setlocale;
}
#endif
```

Remarks

Including this header also ensures that the names declared with external linkage in the Standard C library header are declared in the **std** namespace. In this implementation, the names may or may not also be declared in the global namespace, depending on the specific translation environment.

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

Other Resources

Standard C++ Library Header Files

<cmath>

Defines the macros traditionally defined in the Standard C library header <math.h>.

```
#if <TRADITIONAL C HEADERS>
   #include <math.h>
namespace std {
   using ::abs;
   using ::acos;
   using ::acosf;
   using ::acosl;
   using ::asin;
   using ::asinf;
   using ::asinl;
   using ::atan;
   using ::atan2;
   using ::atan2f;
   using ::atan2l;
   using ::atanf;
   using ::atanl;
   using ::ceil;
   using ::ceilf;
   using ::ceill;
   using ::cos;
   using ::cosf;
   using ::cosh;
   using ::coshf;
   using ::coshl;
   using ::cosl;
   using ::exp;
   using ::expf;
   using ::expl;
   using ::fabs;
   using ::fabsf;
   using ::fabsl;
   using ::floor;
   using ::floorf;
   using ::floorl;
   using ::fmod;
   using ::fmodf;
   using ::fmodl;
   using ::frexp;
   using ::frexpf;
   using ::frexpl;
   using ::ldexp;
   using ::ldexpf;
   using ::ldexpl;
   using ::log;
   using ::log10;
   using ::log10f;
   using ::log10l;
   using ::logf;
   using ::logl;
   using ::modf;
   using ::modff;
   using ::modfl;
   using ::pow;
   using ::powf;
   using ::powl;
   using ::sin;
   using ::sinf;
   using ::sinh;
   using ::sinhf;
   using ::sinhl;
```

```
    using ::sinl;
    using ::sqrt;
    using ::sqrtf;
    using ::sqrtl;
    using ::tan;
    using ::tanf;
    using ::tanh;
    using ::tanhf;
    using ::tanhl;
    using ::tanl;
}
#endif
```

Remarks

Including this header also ensures that the names declared with external linkage in the Standard C library header are declared in the **std** namespace. In this implementation, the names may or may not also be declared in the global namespace, depending on the specific translation environment.

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

Other Resources

Standard C++ Library Header Files

<complex>

Defines the container template class complex and its supporting templates.

For a list of all members of this header, see <complex> Members.

```
#include <complex>
```

Remarks

A complex number is an ordered pair of real numbers. In purely geometrical terms, the complex plane is the real, two-dimensional plane. The special qualities of the complex plane that distinguish it from the real plane are due to its having an additional algebraic structure. This algebraic structure has two fundamental operations:

- Addition defined as $(a, b) + (c, d) = (a + c, b + d)$
- Multiplication defined as $(a, b) * (c, d) = (ac - bd, ad + bc)$

The set of complex numbers with the operations of complex addition and complex multiplication are a field in the standard algebraic sense:

- The operations of addition and multiplication are commutative and associative and multiplication distributes over addition exactly as it does with real addition and multiplication on the field of real numbers.
- The complex number $(0, 0)$ is the additive identity and $(1, 0)$ is the multiplicative identity.
- The additive inverse for a complex number (a, b) is $(-a, -b)$, and the multiplicative inverse for all such complex numbers except $(0, 0)$ is

  $(a/(a^2 + b^2), -b/(a^2 + b^2)$

By representing a complex number $z = (a, b)$ in the form $z = a + bi$, where $i^2 = -1$, the rules for the algebra of the set of real numbers can be applied to the set of complex numbers and to their components. For example:

$(1 + 2i) * (2 + 3i) \quad = 1*(2 + 3i) + 2i*(2 + 3i) = (2 + 3i) + (4i + 6i^2)$

$\quad = (2 - 6) + (3 + 4)i = -4 + 7i$

The system of complex numbers is a field, but it is not an ordered field. There is no ordering of the complex numbers as there is for the field or real numbers and its subsets, so inequalities cannot be applied to complex numbers as they are to real numbers which is an ordered field.

There are three common forms of representing a complex number z:

- Cartesian: $z = a + bi$
- Polar: $z = r$ (cos + $i$sin)
- Exponential: $z = r$ * exp()

The terms used in these standard representations of a complex number are referred to as follows:

- The real Cartesian component or real part a.
- The imaginary Cartesian component or imaginary part b.
- The modulus or absolute value of a complex number P.
- The argument or phase angle .

Unless otherwise specified, functions that can return multiple values are required to return a principal value for their arguments greater than –pi and less than or equal to +pi to keep them single valued. All angles need to be expressed in radians, where there are 2 pi radians (360 degrees) in a circle.

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<complex> Members

Standard C++ Library Header Files

<complex> Members

Reference

Functions

abs	Extracts the modulus of a complex number.
arg	Extracts the argument from a complex number.
conj	Returns the complex conjugate of a complex number.
cos	Returns the cosine of a complex number.
cosh	Returns the hyperbolic cosine of a complex number.
exp	Returns the exponential function of a complex number.
imag	Extracts the imaginary component of a complex number.
log	Returns the natural logarithm of a complex number.
log10	Returns the base 10 logarithm of a complex number.
norm	Extracts the norm of a complex number.
polar	Returns the complex number, which corresponds to a specified modulus and argument, in Cartesian form.
pow	Evaluates the complex number obtained by raising a base that is a complex number to the power of another complex number.
real	Extracts the real component of a complex number.
sin	Returns the sine of a complex number.
sinh	Returns the hyperbolic sine of a complex number.
sqrt	Returns the square root of a complex number.
tan	Returns the tangent of a complex number.
tanh	Returns the hyperbolic tangent of a complex number.

Operators

operator!=	Tests for inequality between two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.
operator*	Multiplies two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.
operator+	Adds two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.

operator-	Subtracts two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.
operator/	Divides two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.
operator<<	A template function that inserts a complex number into the output stream.
operator==	Tests for equality between two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.
operator>>	A template function that extracts a complex value from the input stream.

Classes

complex<double>	The explicitly specialized template class describes an object that stores an ordered pair of objects both of type **double**, first representing the real part of a complex number and the second representing the imaginary part.
complex<float>	The explicitly specialized template class describes an object that stores an ordered pair of objects both of type **float**, first representing the real part of a complex number and the second representing the imaginary part.
complex<long double>	The explicitly specialized template class describes an object that stores an ordered pair of objects both of type **long double**, first representing the real part of a complex number and the second representing the imaginary part.
complex	The template class describes an object used to represent the complex number system and perform complex arithmetic operations.

See Also

Reference

<complex>

Thread Safety in the Standard C++ Library

<complex> Functions

For information about the functions in <complex>, see <complex> Members.

abs

Extracts the modulus of a complex number.

```
template<class Type>
   Type abs(
      const complex<Type>& _ComplexNum
   );
```

Parameters

_ComplexNum

The complex number whose modulus is to be determined.

Return Value

The modulus of a complex number.

Remarks

The modulus of a complex number is a measure of the length of the vector representing the complex number. The modulus of a complex number *a* + *bi* is **sqrt***(a*2 + *b*2*)*, written |*a* + *bi*|. The norm of a complex number *a* + *bi* is *(a*2 + *b*2*)*, so the modulus of a complex number is the square root of its norm.

Example

```
// complex_abs.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Complex numbers can be entered in polar form with
   // modulus and argument parameter inputs but are
   // stored in Cartesian form as real & imag coordinates
   complex <double> c1 ( polar ( 5.0 ) );   // Default argument = 0
   complex <double> c2 ( polar ( 5.0 , pi / 6 ) );
   complex <double> c3 ( polar ( 5.0 , 13 * pi / 6 ) );
   cout << "c1 = polar ( 5.0 ) = " << c1 << endl;
   cout << "c2 = polar ( 5.0 , pi / 6 ) = " << c2 << endl;
   cout << "c3 = polar ( 5.0 , 13 * pi / 6 ) = " << c3 << endl;

   // The modulus and argument of a complex number can be recovered
   // using abs & arg member functions
   double absc1 = abs ( c1 );
   double argc1 = arg ( c1 );
   cout << "The modulus of c1 is recovered from c1 using: abs ( c1 ) = "
        << absc1 << endl;
   cout << "Argument of c1 is recovered from c1 using:\n arg ( c1 ) = "
        << argc1 << " radians, which is " << argc1 * 180 / pi
        << " degrees." << endl;

   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is recovered from c2 using: abs ( c2 ) = "
        << absc2 << endl;
   cout << "Argument of c2 is recovered from c2 using:\n arg ( c2 ) = "
        << argc2 << " radians, which is " << argc2 * 180 / pi
        << " degrees." << endl;

   // Testing if the principal angles of c2 and c3 are the same
```

```
    if ( (arg ( c2 ) <= ( arg ( c3 ) + .00000001) ) ||
         (arg ( c2 ) >= ( arg ( c3 ) - .00000001) ) )
       cout << "The complex numbers c2 & c3 have the "
            << "same principal arguments."<< endl;
    else
       cout << "The complex numbers c2 & c3 don't have the "
            << "same principal arguments." << endl;
  }
```

Output

```
c1 = polar ( 5.0 ) = (5,0)
c2 = polar ( 5.0 , pi / 6 ) = (4.33013,2.5)
c3 = polar ( 5.0 , 13 * pi / 6 ) = (4.33013,2.5)
The modulus of c1 is recovered from c1 using: abs ( c1 ) = 5
Argument of c1 is recovered from c1 using:
 arg ( c1 ) = 0 radians, which is 0 degrees.
The modulus of c2 is recovered from c2 using: abs ( c2 ) = 5
Argument of c2 is recovered from c2 using:
 arg ( c2 ) = 0.523599 radians, which is 30 degrees.
The complex numbers c2 & c3 have the same principal arguments.
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

arg

Extracts the argument from a complex number.

```
template<class Type>
   Type arg(
      const complex<Type>& _ComplexNum
      );
```

Parameters

_ComplexNum

The complex number whose argument is to be determined.

Return Value

The argument of the complex number.

Remarks

The argument is the angle that the complex vector makes with the positive real axis in the complex plane. For a complex number *a* + *bi*, the argument is equal to arctan(*b/a*). The angle has a positive sense when measured in a counterclockwise direction from the positive real axis and a negative sense when measured in a clockwise direction. The principal values are greater than –pi and less than or equal to +pi.

Example

```
// complex_arg.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Complex numbers can be entered in polar form with
   // modulus and argument parameter inputs but are
   // stored in Cartesian form as real & imag coordinates
   complex <double> c1 ( polar ( 5.0 ) );   // Default argument = 0
   complex <double> c2 ( polar ( 5.0 , pi / 6 ) );
   complex <double> c3 ( polar ( 5.0 , 13 * pi / 6 ) );
   cout << "c1 = polar ( 5.0 ) = " << c1 << endl;
   cout << "c2 = polar ( 5.0 , pi / 6 ) = " << c2 << endl;
   cout << "c3 = polar ( 5.0 , 13 * pi / 6 ) = " << c3 << endl;

   // The modulus and argument of a complex number can be rcovered
   // using abs & arg member functions
   double absc1 = abs ( c1 );
   double argc1 = arg ( c1 );
   cout << "The modulus of c1 is recovered from c1 using: abs ( c1 ) = "
        << absc1 << endl;
   cout << "Argument of c1 is recovered from c1 using:\n arg ( c1 ) = "
        << argc1 << " radians, which is " << argc1 * 180 / pi
        << " degrees." << endl;

   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is recovered from c2 using: abs ( c2 ) = "
        << absc2 << endl;
   cout << "Argument of c2 is recovered from c2 using:\n arg ( c2 ) = "
        << argc2 << " radians, which is " << argc2 * 180 / pi
        << " degrees." << endl;
```

```
    // Testing if the principal angles of c2 and c3 are the same
    if ( (arg ( c2 ) <= ( arg ( c3 ) + .00000001) ) ||
         (arg ( c2 ) >= ( arg ( c3 ) - .00000001) ) )
       cout << "The complex numbers c2 & c3 have the "
            << "same principal arguments."<< endl;
    else
       cout << "The complex numbers c2 & c3 don't have the "
            << "same principal arguments." << endl;
  }
```

Output

```
c1 = polar ( 5.0 ) = (5,0)
c2 = polar ( 5.0 , pi / 6 ) = (4.33013,2.5)
c3 = polar ( 5.0 , 13 * pi / 6 ) = (4.33013,2.5)
The modulus of c1 is recovered from c1 using: abs ( c1 ) = 5
Argument of c1 is recovered from c1 using:
 arg ( c1 ) = 0 radians, which is 0 degrees.
The modulus of c2 is recovered from c2 using: abs ( c2 ) = 5
Argument of c2 is recovered from c2 using:
 arg ( c2 ) = 0.523599 radians, which is 30 degrees.
The complex numbers c2 & c3 have the same principal arguments.
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

conj

Returns the complex conjugate of a complex number.

```
template<class Type>
   complex<Type> conj(
      const complex<Type>& _ComplexNum
   );
```

Parameters

_ComplexNum

The complex number whose complex conjugate is being returned.

Return Value

The complex conjugate of the input complex number.

Remarks

The complex conjugate of a complex number $a + bi$ is $a - bi$. The product of a complex number and its conjugate is the norm of the number $a2 + b2$.

Example

```
// complex_conj.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;

   complex <double> c1 ( 4.0 , 3.0 );
   cout << "The complex number c1 = " << c1 << endl;

   double dr1 = real ( c1 );
   cout << "The real part of c1 is real ( c1 ) = "
        << dr1 << "." << endl;

   double di1 = imag ( c1 );
   cout << "The imaginary part of c1 is imag ( c1 ) = "
        << di1 << "." << endl;

   complex <double> c2 = conj ( c1 );
   cout << "The complex conjugate of c1 is c2 = conj ( c1 )= "
        << c2 << endl;

   double dr2 = real ( c2 );
   cout << "The real part of c2 is real ( c2 ) = "
        << dr2 << "." << endl;

   double di2 = imag ( c2 );
   cout << "The imaginary part of c2 is imag ( c2 ) = "
        << di2 << "." << endl;

   // The real part of the product of a complex number
   // and its conjugate is the norm of the number
   complex <double> c3 = c1 * c2;
   cout << "The norm of (c1 * conj (c1) ) is c1 * c2 = "
        << real( c3 ) << endl;
}
```

Output

```
The complex number c1 = (4,3)
The real part of c1 is real ( c1 ) = 4.
The imaginary part of c1 is imag ( c1 ) = 3.
The complex conjugate of c1 is c2 = conj ( c1 )= (4,-3)
The real part of c2 is real ( c2 ) = 4.
The imaginary part of c2 is imag ( c2 ) = -3.
The norm of (c1 * conj (c1) ) is c1 * c2 = 25
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

cos

Returns the cosine of a complex number.

```
template<class Type>
   complex<Type> cos(
      const complex<Type>& _ComplexNum
   );
```

Parameters

_ComplexNum

The complex number whose cosine is being determined.

Return Value

The complex number that is the cosine of the input complex number.

Remarks

Identities defining the complex cosines:

cos (*z*) = (1/2)*(exp (*iz*) + exp (-*iz*))

cos (*z*) = cos (*a* + *bi*) = cos (*a*) cosh (*b*) - isin (*a*) sinh (*b*)

Example

```
// complex_cos.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   complex <double> c1 ( 3.0 , 4.0 );
   cout << "Complex number c1 = " << c1 << endl;

   // Values of cosine of a complex number c1
   complex <double> c2 = cos ( c1 );
   cout << "Complex number c2 = cos ( c1 ) = " << c2 << endl;
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

   // Cosines of the standard angles in the first
   // two quadrants of the complex plane
   vector <complex <double> > v1;
   vector <complex <double> >::iterator Iter1;
   complex <double> vc1  ( polar (1.0, pi / 6) );
   v1.push_back( cos ( vc1 ) );
   complex <double> vc2  ( polar (1.0, pi / 3) );
   v1.push_back( cos ( vc2 ) );
   complex <double> vc3  ( polar (1.0, pi / 2) );
   v1.push_back( cos ( vc3) );
   complex <double> vc4  ( polar (1.0, 2 * pi / 3) );
   v1.push_back( cos ( vc4 ) );
   complex <double> vc5  ( polar (1.0, 5 * pi / 6) );
   v1.push_back( cos ( vc5 ) );
   complex <double> vc6  ( polar (1.0,  pi ) );
   v1.push_back( cos ( vc6 ) );
```

```
    cout << "The complex components cos (vci), where abs (vci) = 1"
         << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << endl;
 }
```

Output

```
Complex number c1 = (3,4)
Complex number c2 = cos ( c1 ) = (-27.0349,-3.85115)
The modulus of c2 is: 27.3079
The argument of c2 is: -3.00009 radians, which is -171.893 degrees.

The complex components cos (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(0.730543,-0.39695)
(1.22777,-0.469075)
(1.54308,1.21529e-013)
(1.22777,0.469075)
(0.730543,0.39695)
(0.540302,-1.74036e-013)
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

cosh

Returns the hyperbolic cosine of a complex number.

```
template<class Type>
   complex<Type> cosh(
      const complex<Type>& _ComplexNum
   );
```

Parameters

_ComplexNum

The complex number whose hyperbolic cosine is being determined.

Return Value

The complex number that is the hyperbolic cosine of the input complex number.

Remarks

Identities defining the complex hyperbolic cosines:

cos (*z*) = (1/2)*(exp (*z*) + exp (-*z*))

cos (*z*) = cosh (*a* + *bi*) = cosh (*a*) cos (*b*) + isinh (*a*) sin (*b*)

Example

```
// complex_cosh.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   complex <double> c1 ( 3.0 , 4.0 );
   cout << "Complex number c1 = " << c1 << endl;

   // Values of cosine of a complex number c1
   complex <double> c2 = cosh ( c1 );
   cout << "Complex number c2 = cosh ( c1 ) = " << c2 << endl;
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

   // Hyperbolic cosines of the standard angles
   // in the first two quadrants of the complex plane
   vector <complex <double> > v1;
   vector <complex <double> >::iterator Iter1;
   complex <double> vc1  ( polar (1.0, pi / 6) );
   v1.push_back( cosh ( vc1 ) );
   complex <double> vc2  ( polar (1.0, pi / 3) );
   v1.push_back( cosh ( vc2 ) );
   complex <double> vc3  ( polar (1.0, pi / 2) );
   v1.push_back( cosh ( vc3) );
   complex <double> vc4  ( polar (1.0, 2 * pi / 3) );
   v1.push_back( cosh ( vc4 ) );
   complex <double> vc5  ( polar (1.0, 5 * pi / 6) );
   v1.push_back( cosh ( vc5 ) );
   complex <double> vc6  ( polar (1.0,  pi ) );
   v1.push_back( cosh ( vc6 ) );
```

```
    cout << "The complex components cosh (vci), where abs (vci) = 1"
         << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << endl;
 }
```

Output

```
Complex number c1 = (3,4)
Complex number c2 = cosh ( c1 ) = (-6.58066,-7.58155)
The modulus of c2 is: 10.0392
The argument of c2 is: -2.28564 radians, which is -130.957 degrees.

The complex components cosh (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(1.22777,0.469075)
(0.730543,0.39695)
(0.540302,-8.70178e-014)
(0.730543,-0.39695)
(1.22777,-0.469075)
(1.54308,2.43059e-013)
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

exp

Returns the exponential function of a complex number.

```
template<class Type>
   complex<Type> exp(
      const complex<Type>& _ComplexNum
      );
```

Parameters

_ComplexNum

The complex number whose exponential is being determined.

Return Value

The complex number that is the exponential of the input complex number.

Example

```
// complex_exp.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main() {
   using namespace std;
   double pi = 3.14159265359;
   complex <double> c1 ( 1 , pi/6 );
   cout << "Complex number c1 = " << c1 << endl;

   // Value of exponential of a complex number c1:
   // note the argument of c2 is determined by the
   // imaginary part of c1 & the modulus by the real part
   complex <double> c2 = exp ( c1 );
   cout << "Complex number c2 = exp ( c1 ) = " << c2 << endl;
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

   // Exponentials of the standard angles
   // in the first two quadrants of the complex plane
   vector <complex <double> > v1;
   vector <complex <double> >::iterator Iter1;
   complex <double> vc1  ( 0.0 , -pi );
   v1.push_back( exp ( vc1 ) );
   complex <double> vc2  ( 0.0, -2 * pi / 3 );
   v1.push_back( exp ( vc2 ) );
   complex <double> vc3  ( 0.0, 0.0 );
   v1.push_back( exp ( vc3 ) );
   complex <double> vc4  ( 0.0, pi / 3 );
   v1.push_back( exp ( vc4 ) );
   complex <double> vc5  ( 0.0 , 2 * pi / 3 );
   v1.push_back( exp ( vc5 ) );
   complex <double> vc6  ( 0.0, pi );
   v1.push_back( exp ( vc6 ) );

   cout << "The complex components exp (vci), where abs (vci) = 1"
        << "\n& arg (vci) = i * pi / 3 of the vector v1 are:\n" ;
   for ( Iter1 = v1.begin() ; Iter1 != v1.end() ; Iter1++ )
      cout <<  ( * Iter1 ) << "\n     with argument = "
           << ( 180/pi ) * arg ( *Iter1 )
```

```
            << " degrees\n     modulus = "
            << abs ( * Iter1 ) << endl;
  }
```

Output

```
Complex number c1 = (1,0.523599)
Complex number c2 = exp ( c1 ) = (2.3541,1.35914)
The modulus of c2 is: 2.71828
The argument of c2 is: 0.523599 radians, which is 30 degrees.

The complex components exp (vci), where abs (vci) = 1
& arg (vci) = i * pi / 3 of the vector v1 are:
(-1,2.06823e-013)
     with argument = 180 degrees
     modulus = 1
(-0.5,-0.866025)
     with argument = -120 degrees
     modulus = 1
(1,0)
     with argument = 0 degrees
     modulus = 1
(0.5,0.866025)
     with argument = 60 degrees
     modulus = 1
(-0.5,0.866025)
     with argument = 120 degrees
     modulus = 1
(-1,-2.06823e-013)
     with argument = -180 degrees
     modulus = 1
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

imag

Extracts the imaginary component of a complex number.

```
template<class Type>
   Type imag(
      const complex<Type>& _ComplexNum
      );
```

Parameters

_ComplexNum

The complex number whose real part is to be extracted.

Return Value

The imaginary part of the complex number as a global function.

Remarks

This template function cannot be used to modify the real part of the complex number. To change the real part, a new complex number must be assigned the component value.

Example

```
// complexc_imag.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   complex <double> c1 ( 4.0 , 3.0 );
   cout << "The complex number c1 = " << c1 << endl;

   double dr1 = real ( c1 );
   cout << "The real part of c1 is real ( c1 ) = "
        << dr1 << "." << endl;

   double di1 = imag ( c1 );
   cout << "The imaginary part of c1 is imag ( c1 ) = "
        << di1 << "." << endl;
}
```

Output

```
The complex number c1 = (4,3)
The real part of c1 is real ( c1 ) = 4.
The imaginary part of c1 is imag ( c1 ) = 3.
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

log

Returns the natural logarithm of a complex number.

```
template<class Type>
   complex<Type> log(
      const complex<Type>& _ComplexNum
   );
```

Parameters

_ComplexNum

The complex number whose natural logarithm is being determined.

Return Value

The complex number that is the natural logarithm of the input complex number.

Remarks

The branch cuts are along the negative real axis.

Example

```
// complex_log.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main() {
   using namespace std;
   double pi = 3.14159265359;
   complex <double> c1 ( 3.0 , 4.0 );
   cout << "Complex number c1 = " << c1 << endl;

   // Values of log of a complex number c1
   complex <double> c2 = log ( c1 );
   cout << "Complex number c2 = log ( c1 ) = " << c2 << endl;
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

   // log of the standard angles
   // in the first two quadrants of the complex plane
   vector <complex <double> > v1;
   vector <complex <double> >::iterator Iter1;
   complex <double> vc1  ( polar (1.0, pi / 6) );
   v1.push_back( log ( vc1 ) );
   complex <double> vc2  ( polar (1.0, pi / 3) );
   v1.push_back( log ( vc2 ) );
   complex <double> vc3  ( polar (1.0, pi / 2) );
   v1.push_back( log ( vc3) );
   complex <double> vc4  ( polar (1.0, 2 * pi / 3) );
   v1.push_back( log ( vc4 ) );
   complex <double> vc5  ( polar (1.0, 5 * pi / 6) );
   v1.push_back( log ( vc5 ) );
   complex <double> vc6  ( polar (1.0,  pi ) );
   v1.push_back( log ( vc6 ) );

   cout << "The complex components log (vci), where abs (vci) = 1 "
        << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
   for ( Iter1 = v1.begin() ; Iter1 != v1.end() ; Iter1++ )
```

```
        cout << *Iter1 << " " << endl;
}
```

Sample Output

```
Complex number c1 = (3,4)
Complex number c2 = log ( c1 ) = (1.60944,0.927295)
The modulus of c2 is: 1.85746
The argument of c2 is: 0.522706 radians, which is 29.9489 degrees.

The complex components log (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(0,0.523599)
(0,1.0472)
(0,1.5708)
(0,2.0944)
(0,2.61799)
(0,-3.14159)
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

log10

Returns the base 10 logarithm of a complex number.

```
template<class Type>
   complex<Type> log10(
      const complex<Type>& _ComplexNum
      );
```

Parameters

_ComplexNum

The complex number whose base 10 logarithm is being determined.

Return Value

The complex number that is the base 10 logarithm of the input complex number.

Remarks

The branch cuts are along the negative real axis.

Example

```
// complex_log10.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main() {
   using namespace std;
   double pi = 3.14159265359;
   complex <double> c1 ( 3.0 , 4.0 );
   cout << "Complex number c1 = " << c1 << endl;

   // Values of log10 of a complex number c1
   complex <double> c2 = log10 ( c1 );
   cout << "Complex number c2 = log10 ( c1 ) = " << c2 << endl;
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

   // log10 of the standard angles
   // in the first two quadrants of the complex plane
   vector <complex <double> > v1;
   vector <complex <double> >::iterator Iter1;
   complex <double> vc1  ( polar (1.0, pi / 6) );
   v1.push_back( log10 ( vc1 ) );
   complex <double> vc2  ( polar (1.0, pi / 3) );
   v1.push_back( log10 ( vc2 ) );
   complex <double> vc3  ( polar (1.0, pi / 2) );
   v1.push_back( log10 ( vc3) );
   complex <double> vc4  ( polar (1.0, 2 * pi / 3) );
   v1.push_back( log10 ( vc4 ) );
   complex <double> vc5  ( polar (1.0, 5 * pi / 6) );
   v1.push_back( log10 ( vc5 ) );
   complex <double> vc6  ( polar (1.0,  pi ) );
   v1.push_back( log10 ( vc6 ) );

   cout << "The complex components log10 (vci), where abs (vci) = 1"
        << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
```

```
        cout << *Iter1 << endl;
}
```

Sample Output

```
Complex number c1 = (3,4)
Complex number c2 = log10 ( c1 ) = (0.69897,0.402719)
The modulus of c2 is: 0.806686
The argument of c2 is: 0.522706 radians, which is 29.9489 degrees.

The complex components log10 (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(0,0.227396)
(0,0.454792)
(0,0.682188)
(0,0.909584)
(0,1.13698)
(0,-1.36438)
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

norm

Extracts the norm of a complex number.

```
template<class Type>
   Type norm(
      const complex<Type>& _ComplexNum
      );
```

Parameters

_ComplexNum

The complex number whose norm is to be determined.

Return Value

The norm of a complex number.

Remarks

The norm of a complex number *a* + *bi* is *(a*2 + *b2)*. The norm of a complex number is the square of its modulus. The modulus of a complex number is a measure of the length of the vector representing the complex number. The modulus of a complex number *a* + *bi* is **sqrt***(a*2 + *b2)*, written |*a* + *bi*|.

Example

```
// complex_norm.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Complex numbers can be entered in polar form with
   // modulus and argument parameter inputs but are
   // stored in Cartesian form as real & imag coordinates
   complex <double> c1 ( polar ( 5.0 ) );   // Default argument = 0
   complex <double> c2 ( polar ( 5.0 , pi / 6 ) );
   complex <double> c3 ( polar ( 5.0 , 13 * pi / 6 ) );
   cout << "c1 = polar ( 5.0 ) = " << c1 << endl;
   cout << "c2 = polar ( 5.0 , pi / 6 ) = " << c2 << endl;
   cout << "c3 = polar ( 5.0 , 13 * pi / 6 ) = " << c3 << endl;

   if ( (arg ( c2 ) <= ( arg ( c3 ) + .00000001) ) ||
        (arg ( c2 ) >= ( arg ( c3 ) - .00000001) ) )
      cout << "The complex numbers c2 & c3 have the "
           << "same principal arguments."<< endl;
   else
      cout << "The complex numbers c2 & c3 don't have the "
           << "same principal arguments." << endl;

   // The modulus and argument of a complex number can be recovered
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is recovered from c2 using: abs ( c2 ) = "
        << absc2 << endl;
   cout << "Argument of c2 is recovered from c2 using:\n arg ( c2 ) = "
        << argc2 << " radians, which is " << argc2 * 180 / pi
        << " degrees." << endl;

   // The norm of a complex number is the square of its modulus
   double normc2 = norm ( c2 );
```

```
    double sqrtnormc2 = sqrt ( normc2 );
    cout << "The norm of c2 given by: norm ( c2 ) = " << normc2 << endl;
    cout << "The modulus of c2 is the square root of the norm: "
         << "sqrt ( normc2 ) = " << sqrtnormc2 << ".";
}
```

Output

```
c1 = polar ( 5.0 ) = (5,0)
c2 = polar ( 5.0 , pi / 6 ) = (4.33013,2.5)
c3 = polar ( 5.0 , 13 * pi / 6 ) = (4.33013,2.5)
The complex numbers c2 & c3 have the same principal arguments.
The modulus of c2 is recovered from c2 using: abs ( c2 ) = 5
Argument of c2 is recovered from c2 using:
 arg ( c2 ) = 0.523599 radians, which is 30 degrees.
The norm of c2 given by: norm ( c2 ) = 25
The modulus of c2 is the square root of the norm: sqrt ( normc2 ) = 5.
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

polar

Returns the complex number, which corresponds to a specified modulus and argument, in Cartesian form.

```
template<class Type>
   complex<Type> polar(
      const Type& _Modulus,
      const Type& _Argument = 0
   );
```

Parameters

_Modulus

The modulus of the complex number being input.

_Argument

The argument of the complex number being input.

Return Value

Cartesian form of the complex number specified in polar form.

Remarks

The polar form of a complex number provides the modulus *r* and the argument , where these parameters are related to the real and imaginary Cartesian components *a* and *b* by the equations *a* = r * cos ( ) and b = *r* * sin ().

Example

```
// complex_polar.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Complex numbers can be entered in polar form with
   // modulus and argument parameter inputs but are
   // stored in Cartesian form as real & imag coordinates
   complex <double> c1 ( polar ( 5.0 ) );   // Default argument = 0
   complex <double> c2 ( polar ( 5.0 , pi / 6 ) );
   complex <double> c3 ( polar ( 5.0 , 13 * pi / 6 ) );
   cout << "c1 = polar ( 5.0 ) = " << c1 << endl;
   cout << "c2 = polar ( 5.0 , pi / 6 ) = " << c2 << endl;
   cout << "c3 = polar ( 5.0 , 13 * pi / 6 ) = " << c3 << endl;

   if ( (arg ( c2 ) <= ( arg ( c3 ) + .00000001) ) ||
        (arg ( c2 ) >= ( arg ( c3 ) - .00000001) ) )
      cout << "The complex numbers c2 & c3 have the "
           << "same principal arguments."<< endl;
   else
      cout << "The complex numbers c2 & c3 don't have the "
           << "same principal arguments." << endl;

   // the modulus and argument of a complex number can be rcovered
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is recovered from c2 using: abs ( c2 ) = "
        << absc2 << endl;
   cout << "Argument of c2 is recovered from c2 using:\n arg ( c2 ) = "
        << argc2 << " radians, which is " << argc2 * 180 / pi
        << " degrees." << endl;
```

```
}
```

Output

```
c1 = polar ( 5.0 ) = (5,0)
c2 = polar ( 5.0 , pi / 6 ) = (4.33013,2.5)
c3 = polar ( 5.0 , 13 * pi / 6 ) = (4.33013,2.5)
The complex numbers c2 & c3 have the same principal arguments.
The modulus of c2 is recovered from c2 using: abs ( c2 ) = 5
Argument of c2 is recovered from c2 using:
 arg ( c2 ) = 0.523599 radians, which is 30 degrees.
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

pow

Evaluates the complex number obtained by raising a base that is a complex number to the power of another complex number.

```
template<class Type>
   complex<Type> pow(
      const complex<Type>& _Base,
      int _Power
   );
template<class Type>
   complex<Type> pow(
      const complex<Type>& _Base,
      const Type& _Power
   );
template<class Type>
   complex<Type> pow(
      const complex<Type>& _Base,
      const complex<Type>& _Power
   );
template<class Type>
   complex<Type> pow(
      const Type& _Base,
      const complex<Type>& _Power
      );
```

Parameters

_Base

The complex number or number that is of the parameter type for the complex number that is the base to be raised to a power by the member function.

_Power

The integer or complex number or number that is of the parameter type for the complex number that is the power that the base is to be raised to by the member function.

Return Value

The complex number obtained by raising the specified base to the specified power.

Remarks

The functions each effectively convert both operands to the return type, and then return the converted **left** to the power **right**.

The branch cut is along the negative real axis.

Example

```
// complex_pow.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // First member function
   // type complex<double> base & type integer power
   complex <double> cb1 ( 3 , 4);
   int cp1 = 2;
   complex <double> ce1 = pow ( cb1 ,cp1 );

   cout << "Complex number for base cb1 = " << cb1 << endl;
   cout << "Integer for power = " << cp1 << endl;
```

```
    cout << "Complex number returned from complex base and integer power:"
         << "\n ce1 = cb1 ^ cp1 = " << ce1 << endl;
    double absce1 = abs ( ce1 );
    double argce1 = arg ( ce1 );
    cout << "The modulus of ce1 is: " << absce1 << endl;
    cout << "The argument of ce1 is: "<< argce1 << " radians, which is "
         << argce1 * 180 / pi << " degrees." << endl << endl;

    // Second member function
    // type complex<double> base & type double power
    complex <double> cb2 ( 3 , 4 );
    double cp2 = pi;
    complex <double> ce2 = pow ( cb2 ,cp2 );

    cout << "Complex number for base cb2 = " << cb2 << endl;
    cout << "Type double for power cp2 = pi = " << cp2 << endl;
    cout << "Complex number returned from complex base and double power:"
         << "\n ce2 = cb2 ^ cp2 = " << ce2 << endl;
    double absce2 = abs ( ce2 );
    double argce2 = arg ( ce2 );
    cout << "The modulus of ce2 is: " << absce2 << endl;
    cout << "The argument of ce2 is: "<< argce2 << " radians, which is "
         << argce2 * 180 / pi << " degrees." << endl << endl;

    // Third member function
    // type complex<double> base & type complex<double> power
    complex <double> cb3 ( 3 , 4 );
    complex <double> cp3 ( -2 , 1 );
    complex <double> ce3 = pow ( cb3 ,cp3 );

    cout << "Complex number for base cb3 = " << cb3 << endl;
    cout << "Complex number for power cp3= " << cp3 << endl;
    cout << "Complex number returned from complex base and complex power:"
         << "\n ce3 = cb3 ^ cp3 = " << ce3 << endl;
    double absce3 = abs ( ce3 );
    double argce3 = arg ( ce3 );
    cout << "The modulus of ce3 is: " << absce3 << endl;
    cout << "The argument of ce3 is: "<< argce3 << " radians, which is "
         << argce3 * 180 / pi << " degrees." << endl << endl;

    // Fourth member function
    // type double base & type complex<double> power
    double cb4 = pi;
    complex <double> cp4 ( 2 , -1 );
    complex <double> ce4 = pow ( cb4 ,cp4 );

    cout << "Type double for base cb4 = pi = " << cb4 << endl;
    cout << "Complex number for power cp4 = " << cp4 << endl;
    cout << "Complex number returned from double base and complex power:"
         << "\n ce4 = cb4 ^ cp4 = " << ce4 << endl;
    double absce4 = abs ( ce4 );
    double argce4 = arg ( ce4 );
    cout << "The modulus of ce4 is: " << absce4 << endl;
    cout << "The argument of ce4 is: "<< argce4 << " radians, which is "
         << argce4 * 180 / pi << " degrees." << endl << endl;
  }
```

Output

```
Complex number for base cb1 = (3,4)
Integer for power = 2
Complex number returned from complex base and integer power:
 ce1 = cb1 ^ cp1 = (-7,24)
The modulus of ce1 is: 25
The argument of ce1 is: 1.85459 radians, which is 106.26 degrees.

Complex number for base cb2 = (3,4)
Type double for power cp2 = pi = 3.14159
Complex number returned from complex base and double power:
```

```
 ce2 = cb2 ^ cp2 = (-152.915,35.5475)
The modulus of ce2 is: 156.993
The argument of ce2 is: 2.91318 radians, which is 166.913 degrees.

Complex number for base cb3 = (3,4)
Complex number for power cp3= (-2,1)
Complex number returned from complex base and complex power:
 ce3 = cb3 ^ cp3 = (0.0153517,-0.00384077)
The modulus of ce3 is: 0.0158249
The argument of ce3 is: -0.245153 radians, which is -14.0462 degrees.

Type double for base cb4 = pi = 3.14159
Complex number for power cp4 = (2,-1)
Complex number returned from double base and complex power:
 ce4 = cb4 ^ cp4 = (4.07903,-8.98725)
The modulus of ce4 is: 9.8696
The argument of ce4 is: -1.14473 radians, which is -65.5882 degrees.
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

real

Extracts the real component of a complex number.

```
template<class Type>
   Type real(
      const complex<Type>& _ComplexNum
      );
```

Parameters

_ComplexNum

The complex number whose real part is to be extracted.

Return Value

The real part of the complex number as a global function.

Remarks

This template function cannot be used to modify the real part of the complex number. To change the real part, a new complex number must be assigned the component value.

Example

```
// complex_real.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   complex <double> c1 ( 4.0 , 3.0 );
   cout << "The complex number c1 = " << c1 << endl;

   double dr1 = real ( c1 );
   cout << "The real part of c1 is real ( c1 ) = "
        << dr1 << "." << endl;

   double di1 = imag ( c1 );
   cout << "The imaginary part of c1 is imag ( c1 ) = "
        << di1 << "." << endl;
}
```

Output

```
The complex number c1 = (4,3)
The real part of c1 is real ( c1 ) = 4.
The imaginary part of c1 is imag ( c1 ) = 3.
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

sin

Returns the sine of a complex number.

```
template<class Type>
   complex<Type> sin(
      const complex<Type>& _ComplexNum
      );
```

Parameters

_ComplexNum

The complex number whose sine is being determined.

Return Value

The complex number that is the sine of the input complex number.

Remarks

Identities defining the complex sines:

sin (z) = (1/2i)*(exp (iz) – exp (-iz))

sin (z) = sin (a + bi) = sin (a) cosh (b) + icos (a) sinh (b)

Example

```
// complex_sin.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   complex <double> c1 ( 3.0 , 4.0 );
   cout << "Complex number c1 = " << c1 << endl;

   // Values of sine of a complex number c1
   complex <double> c2 = sin ( c1 );
   cout << "Complex number c2 = sin ( c1 ) = " << c2 << endl;
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

   // sines of the standard angles in the first
   // two quadrants of the complex plane
   vector <complex <double> > v1;
   vector <complex <double> >::iterator Iter1;
   complex <double> vc1  ( polar ( 1.0, pi / 6 ) );
   v1.push_back( sin ( vc1 ) );
   complex <double> vc2  ( polar ( 1.0, pi / 3 ) );
   v1.push_back( sin ( vc2 ) );
   complex <double> vc3  ( polar ( 1.0, pi / 2 ) );
   v1.push_back( sin ( vc3 ) );
   complex <double> vc4  ( polar ( 1.0, 2 * pi / 3 ) );
   v1.push_back( sin ( vc4 ) );
   complex <double> vc5  ( polar ( 1.0, 5 * pi / 6 ) );
   v1.push_back( sin ( vc5 ) );
   complex <double> vc6  ( polar ( 1.0, pi ) );
   v1.push_back( sin ( vc6 ) );
```

```
    cout << "The complex components sin (vci), where abs (vci) = 1"
         << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << endl;
 }
```

Output

```
Complex number c1 = (3,4)
Complex number c2 = sin ( c1 ) = (3.85374,-27.0168)
The modulus of c2 is: 27.2903
The argument of c2 is: -1.42911 radians, which is -81.882 degrees.

The complex components sin (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(0.85898,0.337596)
(0.670731,0.858637)
(-1.59572e-013,1.1752)
(-0.670731,0.858637)
(-0.85898,0.337596)
(-0.841471,-1.11747e-013)
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

sinh

Returns the hyperbolic sine of a complex number.

```
template<class Type>
   complex<Type> sinh(
      const complex<Type>& _ComplexNum
      );
```

Parameters

_ComplexNum

The complex number whose hyperbolic sine is being determined.

Return Value

The complex number that is the hyperbolic sine of the input complex number.

Remarks

Identities defining the complex hyperbolic sines:

sinh (z) = (1/2)*(exp (z) – exp (-z))

sinh (z) = sinh (a + bi) = sinh (a) cos (b) + $i$cosh (a) sin (b)

Example

```
// complex_sinh.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   complex <double> c1 ( 3.0 , 4.0 );
   cout << "Complex number c1 = " << c1 << endl;

   // Values of sine of a complex number c1
   complex <double> c2 = sinh ( c1 );
   cout << "Complex number c2 = sinh ( c1 ) = " << c2 << endl;
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

   // Hyperbolic sines of the standard angles in
   // the first two quadrants of the complex plane
   vector <complex <double> > v1;
   vector <complex <double> >::iterator Iter1;
   complex <double> vc1  ( polar ( 1.0, pi / 6 ) );
   v1.push_back( sinh ( vc1 ) );
   complex <double> vc2  ( polar ( 1.0, pi / 3 ) );
   v1.push_back( sinh ( vc2 ) );
   complex <double> vc3  ( polar ( 1.0, pi / 2 ) );
   v1.push_back( sinh ( vc3) );
   complex <double> vc4  ( polar ( 1.0, 2 * pi / 3 ) );
   v1.push_back( sinh ( vc4 ) );
   complex <double> vc5  ( polar ( 1.0, 5 * pi / 6 ) );
   v1.push_back( sinh ( vc5 ) );
   complex <double> vc6  ( polar ( 1.0, pi ) );
   v1.push_back( sinh ( vc6 ) );
```

```
    cout << "The complex components sinh (vci), where abs (vci) = 1"
         << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << endl;
 }
```

Output

```
Complex number c1 = (3,4)
Complex number c2 = sinh ( c1 ) = (-6.54812,-7.61923)
The modulus of c2 is: 10.0464
The argument of c2 is: -2.28073 radians, which is -130.676 degrees.

The complex components sinh (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(0.858637,0.670731)
(0.337596,0.85898)
(-5.58735e-014,0.841471)
(-0.337596,0.85898)
(-0.858637,0.670731)
(-1.1752,-3.19145e-013)
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

sqrt

Calculates the square root of a complex number.

```
template<class Type>
   complex<Type> sqrt(
      const complex<Type>& _ComplexNum
      );
```

Parameters

_ComplexNum

The complex number whose square root is to be found.

Return Value

The square root of a complex number.

Remarks

The square root will have a phase angle in the half-open interval (-pi/2, pi/2].

The branch cuts in the complex plane are along the negative real axis.

The square root of a complex number will have a modulus that is the square root of the input number and an argument that is one-half that of the input number.

Example

```
// complex_sqrt.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Complex numbers can be entered in polar form with
   // modulus and argument parameter inputs but are
   // stored in Cartesian form as real & imag coordinates
   complex <double> c1 ( polar ( 25.0 , pi / 2 ) );
   complex <double> c2 = sqrt ( c1 );
   cout << "c1 = polar ( 5.0 ) = " << c1 << endl;
   cout << "c2 = sqrt ( c1 ) = " << c2 << endl;


   // The modulus and argument of a complex number can be recovered
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is recovered from c2 using: abs ( c2 ) = "
        << absc2 << endl;
   cout << "Argument of c2 is recovered from c2 using:\n arg ( c2 ) = "
        << argc2 << " radians, which is " << argc2 * 180 / pi
        << " degrees." << endl;

   // The modulus and argument of c2 can be directly calculated
   absc2 = sqrt( abs ( c1 ) );
   argc2 = 0.5 * arg ( c1 );
   cout << "The modulus of c2 = sqrt( abs ( c1 ) ) =" << absc2 << endl;
   cout << "The argument of c2 = ( 1 / 2 ) * arg ( c1 ) ="
        << argc2 << " radians,\n which is " << argc2 * 180 / pi
        << " degrees." << endl;
}
```

Output

```
c1 = polar ( 5.0 ) = (-2.58529e-012,25)
c2 = sqrt ( c1 ) = (3.53553,3.53553)
The modulus of c2 is recovered from c2 using: abs ( c2 ) = 5
Argument of c2 is recovered from c2 using:
 arg ( c2 ) = 0.785398 radians, which is 45 degrees.
The modulus of c2 = sqrt( abs ( c1 ) ) =5
The argument of c2 = ( 1 / 2 ) * arg ( c1 ) =0.785398 radians,
 which is 45 degrees.
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

Functions tan

Returns the tangent of a complex number.

```
template<class Type>
   complex<Type> tan(
      const complex<Type>& _ComplexNum
      );
```

Parameters

_ComplexNum

The complex number whose tangent is being determined.

Return Value

The complex number that is the tangent of the input complex number.

Remarks

Identities defining the complex cotangent:

tan (*z*) = sin (*z*) / cos (*z*) = (exp (*iz*) – exp (-*iz*)) / *i*(exp (*iz*) + exp (-*iz*))

Example

```
// complex_tan.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   complex <double> c1 ( 3.0 , 4.0 );
   cout << "Complex number c1 = " << c1 << endl;

   // Values of cosine of a complex number c1
   complex <double> c2 = tan ( c1 );
   cout << "Complex number c2 = tan ( c1 ) = " << c2 << endl;
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

   // Hyperbolic tangent of the standard angles
   // in the first two quadrants of the complex plane
   vector <complex <double> > v1;
   vector <complex <double> >::iterator Iter1;
   complex <double> vc1  ( polar ( 1.0, pi / 6 ) );
   v1.push_back( tan ( vc1 ) );
   complex <double> vc2  ( polar ( 1.0, pi / 3 ) );
   v1.push_back( tan ( vc2 ) );
   complex <double> vc3  ( polar ( 1.0, pi / 2 ) );
   v1.push_back( tan ( vc3) );
   complex <double> vc4  ( polar ( 1.0, 2 * pi / 3 ) );
   v1.push_back( tan ( vc4 ) );
   complex <double> vc5  ( polar ( 1.0, 5 * pi / 6 ) );
   v1.push_back( tan ( vc5 ) );
   complex <double> vc6  ( polar ( 1.0,  pi ) );
   v1.push_back( tan ( vc6 ) );

   cout << "The complex components tan (vci), where abs (vci) = 1"
```

```
         << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
    for ( Iter1 = v1.begin() ; Iter1 != v1.end() ; Iter1++ )
       cout << *Iter1 << endl;
 }
```

Output

```
Complex number c1 = (3,4)
Complex number c2 = tan ( c1 ) = (-0.000187346,0.999356)
The modulus of c2 is: 0.999356
The argument of c2 is: 1.57098 radians, which is 90.0107 degrees.

The complex components tan (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(0.713931,0.85004)
(0.24356,0.792403)
(-4.34302e-014,0.761594)
(-0.24356,0.792403)
(-0.713931,0.85004)
(-1.55741,-7.08476e-013)
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

tanh

Returns the hyperbolic tangent of a complex number.

```
template<class Type>
   complex<Type> tanh(
      const complex<Type>& _ComplexNum
      );
```

Parameters

_ComplexNum

The complex number whose hyperbolic tangent is being determined.

Return Value

The complex number that is the hyperbolic tangent of the input complex number.

Remarks

Identities defining the complex hyperbolic cotangent:

tanh (*z*) = sinh (*z*) / cosh (*z*) = (exp (*z*) – exp (-*z*)) / (exp (*z*) + exp (-*z*))

Example

```
// complex_tanh.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   complex <double> c1 ( 3.0 , 4.0 );
   cout << "Complex number c1 = " << c1 << endl;

   // Values of cosine of a complex number c1
   complex <double> c2 = tanh ( c1 );
   cout << "Complex number c2 = tanh ( c1 ) = " << c2 << endl;
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

   // Hyperbolic tangents of the standard angles
   // in the first two quadrants of the complex plane
   vector <complex <double> > v1;
   vector <complex <double> >::iterator Iter1;
   complex <double> vc1  ( polar ( 1.0, pi / 6 ) );
   v1.push_back( tanh ( vc1 ) );
   complex <double> vc2  ( polar ( 1.0, pi / 3 ) );
   v1.push_back( tanh ( vc2 ) );
   complex <double> vc3  ( polar ( 1.0, pi / 2 ) );
   v1.push_back( tanh ( vc3 ) );
   complex <double> vc4  ( polar ( 1.0, 2 * pi / 3 ) );
   v1.push_back( tanh ( vc4 ) );
   complex <double> vc5  ( polar ( 1.0, 5 * pi / 6 ) );
   v1.push_back( tanh ( vc5 ) );
   complex <double> vc6  ( polar ( 1.0, pi ) );
   v1.push_back( tanh ( vc6 ) );

   cout << "The complex components tanh (vci), where abs (vci) = 1"
```

```
         << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << endl;
 }
```

Output

```
Complex number c1 = (3,4)
Complex number c2 = tanh ( c1 ) = (1.00071,0.00490826)
The modulus of c2 is: 1.00072
The argument of c2 is: 0.00490474 radians, which is 0.281021 degrees.

The complex components tanh (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(0.792403,0.24356)
(0.85004,0.713931)
(-3.54238e-013,1.55741)
(-0.85004,0.713931)
(-0.792403,0.24356)
(-0.761594,-8.68604e-014)
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

<complex> Operators

For more information about operators in <complex>, see <complex> Members.

operator!= (<complex>)

Tests for inequality between two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.

```
template<class Type>
   bool operator!=(
      const complex<Type>& _Left,
      const complex<Type>& _Right
   );
template<class Type>
   bool operator!=(
      const complex<Type>& _Left,
      const Type& _Right
   );
template<class Type>
   bool operator!=(
      const Type& _Left,
      const complex<Type>& _Right
   );
```

Parameters

_Left

A complex number or object of its parameter type to be tested for inequality.

_Right

A complex number or object of its parameter type to be tested for inequality.

Return Value

true if the numbers are not equal; **false** if numbers are equal.

Remarks

Two complex numbers are equal if and only if their real parts are equal and their imaginary parts are equal. Otherwise, they are unequal.

The operation is overloaded so that comparison tests can be executed without the conversion of the data to a particular format.

Example

```
// complex_op_NE.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> compared with type complex<double>
   complex <double> cl1 ( polar (3.0 , pi / 6 ) );
   complex <double> cr1a ( polar (3.0 , pi /6 ) );
   complex <double> cr1b ( polar (2.0 , pi / 3 ) );

   cout << "The left-side complex number is cl1 = " << cl1 << endl;
   cout << "The 1st right-side complex number is cr1a = " << cr1a << endl;
   cout << "The 2nd right-side complex number is cr1b = " << cr1b << endl;
   if ( cl1 != cr1a )
      cout << "The complex numbers cl1 & cr1a are not equal." << endl;
   else
      cout << "The complex numbers cl1 & cr1a are equal." << endl;
```

```
    if ( cl1 != cr1b )
       cout << "The complex numbers cl1 & cr1b are not equal." << endl;
    else
       cout << "The complex numbers cl1 & cr1b are equal." << endl;
    cout << endl;

    // Example of the second member function
    // type complex<int> compared with type int
    complex <int> cl2a ( 3 , 4 );
    complex <int> cl2b ( 5 ,0 );
    int cr2a =3;
    int cr2b =5;

    cout << "The 1st left-side complex number is cl2a = " << cl2a << endl;
    cout << "The 1st right-side complex number is cr2a = " << cr2a << endl;
    if ( cl2a != cr2a )
       cout << "The complex numbers cl2a & cr2a are not equal." << endl;
    else
       cout << "The complex numbers cl2a & cr2a are equal." << endl;

    cout << "The 2nd left-side complex number is cl2b = " << cl2b << endl;
    cout << "The 2nd right-side complex number is cr2b = " << cr2b << endl;
    if ( cl2b != cr2b )
       cout << "The complex numbers cl2b & cr2b are not equal." << endl;
    else
       cout << "The complex numbers cl2b & cr2b are equal." << endl;
    cout << endl;

    // Example of the third member function
    // type double compared with type complex<double>
    double cl3a =3;
    double cl3b =5;
    complex <double> cr3a ( 3 , 4 );
    complex <double> cr3b ( 5 ,0 );

    cout << "The 1st left-side complex number is cl3a = " << cl3a << endl;
    cout << "The 1st right-side complex number is cr3a = " << cr3a << endl;
    if ( cl3a != cr3a )
       cout << "The complex numbers cl3a & cr3a are not equal." << endl;
    else
       cout << "The complex numbers cl3a & cr3a are equal." << endl;

    cout << "The 2nd left-side complex number is cl3b = " << cl3b << endl;
    cout << "The 2nd right-side complex number is cr3b = " << cr3b << endl;
    if ( cl3b != cr3b )
       cout << "The complex numbers cl3b & cr3b are not equal." << endl;
    else
       cout << "The complex numbers cl3b & cr3b are equal." << endl;
    cout << endl;
  }
```

Output

```
The left-side complex number is cl1 = (2.59808,1.5)
The 1st right-side complex number is cr1a = (2.59808,1.5)
The 2nd right-side complex number is cr1b = (1,1.73205)
The complex numbers cl1 & cr1a are equal.
The complex numbers cl1 & cr1b are not equal.

The 1st left-side complex number is cl2a = (3,4)
The 1st right-side complex number is cr2a = 3
The complex numbers cl2a & cr2a are not equal.
The 2nd left-side complex number is cl2b = (5,0)
The 2nd right-side complex number is cr2b = 5
The complex numbers cl2b & cr2b are equal.

The 1st left-side complex number is cl3a = 3
The 1st right-side complex number is cr3a = (3,4)
The complex numbers cl3a & cr3a are not equal.
```

```
The 2nd left-side complex number is cl3b = 5
The 2nd right-side complex number is cr3b = (5,0)
The complex numbers cl3b & cr3b are equal.
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

operator* (<complex>)

Multiplies two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.

```
template<class Type>
   complex<Type> operator*(
      const complex<Type>& _Left,
      const complex<Type>& _Right
   );
template<class Type>
   complex<Type> operator*(
      const complex<Type>& _Left,
      const Type& _Right
   );
template<class Type>
   complex<Type> operator*(
      const Type& _Left,
      const complex<Type>& _Right
   );
```

Parameters

_Left

The first of two complex numbers or a number that is of the parameter type for a complex number that is to be multiplied by the * operation.

_Right

The second of two complex numbers or a number that is of the parameter type for a complex number that is to be multiplied by the * operation.

Return Value

The complex number that results from the multiplication of the two numbers whose value and type are specified by the parameter inputs.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

```
// complex_op_mult.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> times type complex<double>
   complex <double> cl1 ( polar (3.0 , pi / 6 ) );
   complex <double> cr1 ( polar (2.0 , pi / 3 ) );
   complex <double> cs1 = cl1 * cr1;

   cout << "The left-side complex number is cl1 = " << cl1 << endl;
   cout << "The right-side complex number is cr1 = " << cr1 << endl;
   cout << "Product of two complex numbers is: cs1 = " << cs1 << endl;
   double abscs1 = abs ( cs1 );
   double argcs1 = arg ( cs1 );
   cout << "The modulus of cs1 is: " << abscs1 << endl;
   cout << "The argument of cs1 is: "<< argcs1 << " radians, which is "
```

```
        << argcs1 * 180 / pi << " degrees." << endl << endl;

    // Example of the second member function
    // type complex<double> times type double
    complex <double> cl2 ( polar ( 3.0 , pi / 6 ) );
    double cr2 =5;
    complex <double> cs2 = cl2 * cr2;

    cout << "The left-side complex number is cl2 = " << cl2 << endl;
    cout << "The right-side complex number is cr2 = " << cr2 << endl;
    cout << "Product of two complex numbers is: cs2 = " << cs2 << endl;
    double abscs2 = abs ( cs2 );
    double argcs2 = arg ( cs2 );
    cout << "The modulus of cs2 is: " << abscs2 << endl;
    cout << "The argument of cs2 is: "<< argcs2 << " radians, which is "
        << argcs2 * 180 / pi << " degrees." << endl << endl;

    // Example of the third member function
    // type double times type complex<double>
    double cl3 = 5;
    complex <double> cr3 ( polar (3.0 , pi / 6 ) );
    complex <double> cs3 = cl3 * cr3;

    cout << "The left-side complex number is cl3 = " << cl3 << endl;
    cout << "The right-side complex number is cr3 = " << cr3 << endl;
    cout << "Product of two complex numbers is: cs3 = " << cs3 << endl;
    double abscs3 = abs ( cs3 );
    double argcs3 = arg ( cs3 );
    cout << "The modulus of cs3 is: " << abscs3 << endl;
    cout << "The argument of cs3 is: "<< argcs3 << " radians, which is "
        << argcs3 * 180 / pi << " degrees." << endl << endl;
}
```

Sample Output

The following is output on x86.

```
The left-side complex number is cl1 = (2.59808,1.5)
The right-side complex number is cr1 = (1,1.73205)
Product of two complex numbers is: cs1 = (-6.20837e-013,6)
The modulus of cs1 is: 6
The argument of cs1 is: 1.5708 radians, which is 90 degrees.

The left-side complex number is cl2 = (2.59808,1.5)
The right-side complex number is cr2 = 5
Product of two complex numbers is: cs2 = (12.9904,7.5)
The modulus of cs2 is: 15
The argument of cs2 is: 0.523599 radians, which is 30 degrees.

The left-side complex number is cl3 = 5
The right-side complex number is cr3 = (2.59808,1.5)
Product of two complex numbers is: cs3 = (12.9904,7.5)
The modulus of cs3 is: 15
The argument of cs3 is: 0.523599 radians, which is 30 degrees.
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

operator+ (<complex>)

Adds two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.

```
template<class Type>
   complex<Type> operator+(
      const complex<Type>& _Left,
      const complex<Type>& _Right
   );
template<class Type>
   complex<Type> operator+(
      const complex<Type>& _Left,
      const Type& _Right
      );
template<class Type>
   complex<Type> operator+(
      const Type& _Left,
      const complex<Type>& _Right
      );
template<class Type>
   complex<Type> operator+(
      const complex<Type>& _Left
      );
```

Parameters

_Left

The first of two complex numbers or a number that is of the parameter type for a complex number that is to be added by the + operation.

_Right

The second of two complex numbers or a number that is of the parameter type for a complex number that is to be added by the + operation.

Return Value

The complex number that results from the addition of the two numbers whose value and type are specified by the parameter inputs.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format. The unary operator returns *_Left*.

Example

```
// complex_op_add.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> plus type complex<double>
   complex <double> cl1 ( 3.0 , 4.0 );
   complex <double> cr1 ( 2.0 , 5.0 );
   complex <double> cs1 = cl1 + cr1;

   cout << "The left-side complex number is cl1 = " << cl1 << endl;
   cout << "The right-side complex number is cr1 = " << cr1 << endl;
   cout << "The sum of the two complex numbers is: cs1 = " << cs1 << endl;
```

```
    double abscs1 = abs ( cs1 );
    double argcs1 = arg ( cs1 );
    cout << "The modulus of cs1 is: " << abscs1 << endl;
    cout << "The argument of cs1 is: "<< argcs1 << " radians, which is "
         << argcs1 * 180 / pi << " degrees." << endl << endl;

    // Example of the second member function
    // type complex<double> plus type double
    complex <double> cl2 ( 3.0 , 4.0 );
    double cr2 =5.0;
    complex <double> cs2 = cl2 + cr2;

    cout << "The left-side complex number is cl2 = " << cl2 << endl;
    cout << "The right-side complex number is cr2 = " << cr2 << endl;
    cout << "The sum of the two complex numbers is: cs2 = " << cs2 << endl;
    double abscs2 = abs ( cs2 );
    double argcs2 = arg ( cs2 );
    cout << "The modulus of cs2 is: " << abscs2 << endl;
    cout << "The argument of cs2 is: "<< argcs2 << " radians, which is "
         << argcs2 * 180 / pi << " degrees." << endl << endl;

    // Example of the third member function
    // type double plus type complex<double>
    double cl3 = 5.0;
    complex <double> cr3 ( 3.0 , 4.0 );
    complex <double> cs3 = cl3 + cr3;

    cout << "The left-side complex number is cl3 = " << cl3 << endl;
    cout << "The right-side complex number is cr3 = " << cr3 << endl;
    cout << "The sum of the two complex numbers is: cs3 = " << cs3 << endl;
    double abscs3 = abs ( cs3 );
    double argcs3 = arg ( cs3 );
    cout << "The modulus of cs3 is: " << abscs3 << endl;
    cout << "The argument of cs3 is: "<< argcs3 << " radians, which is "
         << argcs3 * 180 / pi << " degrees." << endl << endl;

    // Example of the fourth member function
    // plus type complex<double>
    complex <double> cr4 ( 3.0 , 4.0 );
    complex <double> cs4 = + cr4;

    cout << "The right-side complex number is cr4 = " << cr4 << endl;
    cout << "The result of the unary application of + to the right-side"
         << "\n complex number is: cs4 = " << cs4 << endl;
    double abscs4 = abs ( cs4 );
    double argcs4 = arg ( cs4 );
    cout << "The modulus of cs4 is: " << abscs4 << endl;
    cout << "The argument of cs4 is: "<< argcs4 << " radians, which is "
         << argcs4 * 180 / pi << " degrees." << endl << endl;
}
```

Output

```
The left-side complex number is cl1 = (3,4)
The right-side complex number is cr1 = (2,5)
The sum of the two complex numbers is: cs1 = (5,9)
The modulus of cs1 is: 10.2956
The argument of cs1 is: 1.0637 radians, which is 60.9454 degrees.

The left-side complex number is cl2 = (3,4)
The right-side complex number is cr2 = 5
The sum of the two complex numbers is: cs2 = (8,4)
The modulus of cs2 is: 8.94427
The argument of cs2 is: 0.463648 radians, which is 26.5651 degrees.

The left-side complex number is cl3 = 5
The right-side complex number is cr3 = (3,4)
The sum of the two complex numbers is: cs3 = (8,4)
The modulus of cs3 is: 8.94427
```

```
The argument of cs3 is: 0.463648 radians, which is 26.5651 degrees.

The right-side complex number is cr4 = (3,4)
The result of the unary application of + to the right-side
 complex number is: cs4 = (3,4)
The modulus of cs4 is: 5
The argument of cs4 is: 0.927295 radians, which is 53.1301 degrees.
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

operator- (<complex>)

Subtracts two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.

```
template<class Type>
   complex<Type> operator-(
      const complex<Type>& _Left,
      const complex<Type>& _Right
   );
template<class Type>
   complex<Type> operator-(
      const complex<Type>& _Left,
      const Type& _Right
      );
template<class Type>
   complex<Type> operator-(
      const Type& _Left,
      const complex<Type>& _Right
      );
template<class Type>
   complex<Type> operator-(
      const complex<Type>& _Left
      );
```

Parameters

_Left

The first of two complex numbers or a number that is of the parameter type for a complex number that is to be subtracted by the - operation.

_Right

The second of two complex numbers or a number that is of the parameter type for a complex number that is to be subtracted by the - operation.

Return Value

The complex number that results from the subtraction of *_Right* from *_Left*, the two numbers whose values are specified by the parameter inputs.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

The unary operator changes the sign of a complex number and returns a value whose real part is the negative of the real part of the number input and whose imaginary part is the negative of the imaginary part of the number input.

Example

```
// complex_op_sub.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> minus type complex<double>
   complex <double> cl1 ( 3.0 , 4.0 );
   complex <double> cr1 ( 2.0 , 5.0 );
   complex <double> cs1 = cl1 - cr1;
```

```
    cout << "The left-side complex number is cl1 = " << cl1 << endl;
    cout << "The right-side complex number is cr1 = " << cr1 << endl;
    cout << "Difference of two complex numbers is: cs1 = " << cs1 << endl;
    double abscs1 = abs ( cs1 );
    double argcs1 = arg ( cs1 );
    cout << "The modulus of cs1 is: " << abscs1 << endl;
    cout << "The argument of cs1 is: "<< argcs1 << " radians, which is "
         << argcs1 * 180 / pi << " degrees." << endl << endl;

    // Example of the second member function
    // type complex<double> minus type double
    complex <double> cl2 ( 3.0 , 4.0 );
    double cr2 =5.0;
    complex <double> cs2 = cl2 - cr2;

    cout << "The left-side complex number is cl2 = " << cl2 << endl;
    cout << "The right-side complex number is cr2 = " << cr2 << endl;
    cout << "Difference of two complex numbers is: cs2 = " << cs2 << endl;
    double abscs2 = abs ( cs2 );
    double argcs2 = arg ( cs2 );
    cout << "The modulus of cs2 is: " << abscs2 << endl;
    cout << "The argument of cs2 is: "<< argcs2 << " radians, which is "
         << argcs2 * 180 / pi << " degrees." << endl << endl;

    // Example of the third member function
    // type double minus type complex<double>
    double cl3 = 5.0;
    complex <double> cr3 ( 3.0 , 4.0 );
    complex <double> cs3 = cl3 - cr3;

    cout << "The left-side complex number is cl3 = " << cl3 << endl;
    cout << "The right-side complex number is cr3 = " << cr3 << endl;
    cout << "Difference of two complex numbers is: cs3 = " << cs3 << endl;
    double abscs3 = abs ( cs3 );
    double argcs3 = arg ( cs3 );
    cout << "The modulus of cs3 is: " << abscs3 << endl;
    cout << "The argument of cs3 is: "<< argcs3 << " radians, which is "
         << argcs3 * 180 / pi << " degrees." << endl << endl;

    // Example of the fourth member function
    // minus type complex<double>
    complex <double> cr4 ( 3.0 , 4.0 );
    complex <double> cs4 = - cr4;

    cout << "The right-side complex number is cr4 = " << cr4 << endl;
    cout << "The result of the unary application of - to the right-side"
         << "\n complex number is: cs4 = " << cs4 << endl;
    double abscs4 = abs ( cs4 );
    double argcs4 = arg ( cs4 );
    cout << "The modulus of cs4 is: " << abscs4 << endl;
    cout << "The argument of cs4 is: "<< argcs4 << " radians, which is "
         << argcs4 * 180 / pi << " degrees." << endl << endl;
  }
```

Output

```
The left-side complex number is cl1 = (3,4)
The right-side complex number is cr1 = (2,5)
Difference of two complex numbers is: cs1 = (1,-1)
The modulus of cs1 is: 1.41421
The argument of cs1 is: -0.785398 radians, which is -45 degrees.

The left-side complex number is cl2 = (3,4)
The right-side complex number is cr2 = 5
Difference of two complex numbers is: cs2 = (-2,4)
The modulus of cs2 is: 4.47214
The argument of cs2 is: 2.03444 radians, which is 116.565 degrees.

The left-side complex number is cl3 = 5
```

```
The right-side complex number is cr3 = (3,4)
Difference of two complex numbers is: cs3 = (2,-4)
The modulus of cs3 is: 4.47214
The argument of cs3 is: -1.10715 radians, which is -63.4349 degrees.

The right-side complex number is cr4 = (3,4)
The result of the unary application of - to the right-side
 complex number is: cs4 = (-3,-4)
The modulus of cs4 is: 5
The argument of cs4 is: -2.2143 radians, which is -126.87 degrees.
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

operator/ (<complex>)

Divides two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.

```
template<class Type>
   complex<Type> operator*(
      const complex<Type>& _Left,
      const complex<Type>& _Right
   );
template<class Type>
   complex<Type> operator*(
      const complex<Type>& _Left,
      const Type& _Right
   );
template<class Type>
   complex<Type> operator*(
      const Type& _Left,
      const complex<Type>& _Right
   );
```

Parameters

_Left

A complex number or a number that is of the parameter type for a complex number that is the numerator to be divided by the denominator with the / operation.

_Right

A complex number or a number that is of the parameter type for a complex number that is the denominator to be used to divide the numerator with the / operation.

Return Value

The complex number that results from the division of the numerator by the denominator, the values of which are specified by the parameter inputs.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

```
// complex_op_div.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> divided by type complex<double>
   complex <double> cl1 ( polar ( 3.0 , pi / 6 ) );
   complex <double> cr1 ( polar ( 2.0 , pi / 3 ) );
   complex <double> cs1 = cl1 / cr1;

   cout << "The left-side complex number is cl1 = " << cl1 << endl;
   cout << "The right-side complex number is cr1 = " << cr1 << endl;
   cout << "The quotient of the two complex numbers is: cs1 = cl1 /cr1 = "
        << cs1 << endl;
   double abscs1 = abs ( cs1 );
   double argcs1 = arg ( cs1 );
   cout << "The modulus of cs1 is: " << abscs1 << endl;
```

```
    cout << "The argument of cs1 is: "<< argcs1 << " radians, which is "
         << argcs1 * 180 / pi << " degrees." << endl << endl;

    // example of the second member function
    // type complex<double> divided by type double
    complex <double> cl2 ( polar (3.0 , pi / 6 ) );
    double cr2 =5;
    complex <double> cs2 = cl2 / cr2;

    cout << "The left-side complex number is cl2 = " << cl2 << endl;
    cout << "The right-side complex number is cr2 = " << cr2 << endl;
    cout << "The quotient of the two complex numbers is: cs2 = cl2 /cr2 = "
         << cs2 << endl;
    double abscs2 = abs ( cs2 );
    double argcs2 = arg ( cs2 );
    cout << "The modulus of cs2 is: " << abscs2 << endl;
    cout << "The argument of cs2 is: "<< argcs2 << " radians, which is "
         << argcs2 * 180 / pi << " degrees." << endl << endl;

    // Example of the third member function
    // type double divided by type complex<double>
    double cl3 = 5;
    complex <double> cr3 ( polar ( 3.0 , pi / 6 ) );
    complex <double> cs3 = cl3 / cr3;

    cout << "The left-side complex number is cl3 = " << cl3 << endl;
    cout << "The right-side complex number is cr3 = " << cr3 << endl;
    cout << "The quotient of the two complex numbers is: cs3 = cl3 /cr2 = "
         << cs3 << endl;
    double abscs3 = abs ( cs3 );
    double argcs3 = arg ( cs3 );
    cout << "The modulus of cs3 is: " << abscs3 << endl;
    cout << "The argument of cs3 is: "<< argcs3 << " radians, which is "
         << argcs3 * 180 / pi << " degrees." << endl << endl;
}
```

Output

```
The left-side complex number is cl1 = (2.59808,1.5)
The right-side complex number is cr1 = (1,1.73205)
The quotient of the two complex numbers is: cs1 = cl1 /cr1 = (1.29904,-0.75)
The modulus of cs1 is: 1.5
The argument of cs1 is: -0.523599 radians, which is -30 degrees.

The left-side complex number is cl2 = (2.59808,1.5)
The right-side complex number is cr2 = 5
The quotient of the two complex numbers is: cs2 = cl2 /cr2 = (0.519615,0.3)
The modulus of cs2 is: 0.6
The argument of cs2 is: 0.523599 radians, which is 30 degrees.

The left-side complex number is cl3 = 5
The right-side complex number is cr3 = (2.59808,1.5)
The quotient of the two complex numbers is: cs3 = cl3 /cr2 = (1.44338,-0.833333)
The modulus of cs3 is: 1.66667
The argument of cs3 is: -0.523599 radians, which is -30 degrees.
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

operator<< (<complex>)

Inserts a complex number specified into the output stream.

```
template<class Type, class Elem, class Traits>
   basic_ostream<Elem, Traits>&
      operator<< (
         basic_ostream<Elem, Traits>& _Ostr,
         const complex<Type>& _Right
            );
```

Parameters

_Ostr

The output stream into which the complex number is being entered.

_Right

The complex number to be entered into the output stream

Return Value

Writes the value of the specified complex number to the *_Ostr* in a Cartesian format: (*real part, imaginary part*).

Remarks

The output stream is overloaded so that it will accept any form of a complex number, and its default output format is the Cartesian format.

Example

```
// complex_op_insert.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   complex <double> c1 ( 3.0 , 4.0 );
   cout << "Complex number c1 = " << c1 << endl;

   complex <double> c2  ( polar ( 2.0 , pi / 6 ) );
   cout << "Complex number c2 = " << c2 << endl;

   // To display in polar form
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;
}
```

Output

```
Complex number c1 = (3,4)
Complex number c2 = (1.73205,1)
The modulus of c2 is: 2
The argument of c2 is: 0.523599 radians, which is 30 degrees.
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

operator== (<complex>)

Tests for equality between two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.

```
template<class Type>
   bool operator==(
      const complex<Type>& _Left,
      const complex<Type>& _Right
   );
template<class Type>
   bool operator==(
      const complex<Type>& _Left,
      const Type& _Right
   );
template<class Type>
   bool operator==(
      const Type& _Left,
      const complex<Type>& _Right
   );
```

Parameters

_Left

A complex number or object of its parameter type to be tested for inequality.

_Right

A complex number or object of its parameter type to be tested for inequality.

Return Value

true if the numbers are equal; **false** if numbers are not equal.

Remarks

Two complex numbers are equal if and only if their real parts are equal and their imaginary parts are equal. Otherwise, they are unequal.

The operation is overloaded so that comparison tests can be executed without the conversion of the data to a particular format.

Example

```
// complex_op_EQ.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> compared with type complex<double>
   complex <double> cl1 ( polar ( 3.0 , pi / 6 ) );
   complex <double> cr1a ( polar ( 3.0 , pi /6 ) );
   complex <double> cr1b ( polar ( 2.0 , pi / 3 ) );

   cout << "The left-side complex number is cl1 = " << cl1 << endl;
   cout << "The 1st right-side complex number is cr1a = " << cr1a << endl;
   cout << "The 2nd right-side complex number is cr1b = " << cr1b << endl;
   if ( cl1 == cr1a )
      cout << "The complex numbers cl1 & cr1a are equal." << endl;
   else
      cout << "The complex numbers cl1 & cr1a are not equal." << endl;
```

```
    if ( cl1 == cr1b )
       cout << "The complex numbers cl1 & cr1b are equal." << endl;
    else
       cout << "The complex numbers cl1 & cr1b are not equal." << endl;
    cout << endl;

    // Example of the second member function
    // type complex<int> compared with type int
    complex <int> cl2a ( 3 , 4 );
    complex <int> cl2b ( 5 ,0 );
    int cr2a =3;
    int cr2b =5;

    cout << "The 1st left-side complex number is cl2a = " << cl2a << endl;
    cout << "The 1st right-side complex number is cr2a = " << cr2a << endl;
    if ( cl2a == cr2a )
       cout << "The complex numbers cl2a & cr2a are equal." << endl;
    else
       cout << "The complex numbers cl2a & cr2a are not equal." << endl;

    cout << "The 2nd left-side complex number is cl2b = " << cl2b << endl;
    cout << "The 2nd right-side complex number is cr2b = " << cr2b << endl;
    if ( cl2b == cr2b )
       cout << "The complex numbers cl2b & cr2b are equal." << endl;
    else
       cout << "The complex numbers cl2b & cr2b are not equal." << endl;
    cout << endl;

    // Example of the third member function
    // type double compared with type complex<double>
    double cl3a =3;
    double cl3b =5;
    complex <double> cr3a (3 , 4 );
    complex <double> cr3b (5 ,0 );

    cout << "The 1st left-side complex number is cl3a = " << cl3a << endl;
    cout << "The 1st right-side complex number is cr3a = " << cr3a << endl;
    if ( cl3a == cr3a )
       cout << "The complex numbers cl3a & cr3a are equal." << endl;
    else
       cout << "The complex numbers cl3a & cr3a are not equal." << endl;

    cout << "The 2nd left-side complex number is cl3b = " << cl3b << endl;
    cout << "The 2nd right-side complex number is cr3b = " << cr3b << endl;
    if ( cl3b == cr3b )
       cout << "The complex numbers cl3b & cr3b are equal." << endl;
    else
       cout << "The complex numbers cl3b & cr3b are not equal." << endl;
    cout << endl;
 }
```

Output

```
The left-side complex number is cl1 = (2.59808,1.5)
The 1st right-side complex number is cr1a = (2.59808,1.5)
The 2nd right-side complex number is cr1b = (1,1.73205)
The complex numbers cl1 & cr1a are equal.
The complex numbers cl1 & cr1b are not equal.

The 1st left-side complex number is cl2a = (3,4)
The 1st right-side complex number is cr2a = 3
The complex numbers cl2a & cr2a are not equal.
The 2nd left-side complex number is cl2b = (5,0)
The 2nd right-side complex number is cr2b = 5
The complex numbers cl2b & cr2b are equal.

The 1st left-side complex number is cl3a = 3
The 1st right-side complex number is cr3a = (3,4)
The complex numbers cl3a & cr3a are not equal.
```

```
The 2nd left-side complex number is cl3b = 5
The 2nd right-side complex number is cr3b = (5,0)
The complex numbers cl3b & cr3b are equal.
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

operator>> (<complex>)

Extracts a complex value from the input stream.

```
template<class Type, class Elem, class Traits>
   basic_istream<Elem, Traits>&
      operator>>(
         basic_istream<Elem, Traits>& _Istr,
         complex<Type>& _Right
      );
```

Parameters

_Istr

The input stream from which the complex number is being extracted.

_Right

The complex number that is being extracted from the input stream.

Return Value

Reads the value of the specified complex number from *_Istr* and returns it into *_Right*.

Remarks

The valid input formats are

- *(real part, imaginary part)*
- *(real part)*
- *real part*

Example

```
// complex_op_extract.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   complex <double> c2;

   cout << "Input a complex number ( try: 2.0 ): ";
   cin >> c2;
   cout << c2 << endl;
}
```

Input

```
2.0
```

Requirements

Header: <complex>

See Also

Other Resources

<complex> Members

Classes/Specializations

For more information about classes/specializations in <complex>, see <complex> Members.

complex<double>

Describes an object that stores an ordered pair of objects both of type **double**, the first representing the real part of a complex number and the second representing the imaginary part.

```
template<>
   class complex<double> {
public:
   complex(
      double _RealVal = 0,
      double _ImagVal = 0
   );

   complex(
      const complex<float>& _ComplexNum
   );
   explicit complex(
      const complex<long double>& _ComplexNum
      );
   // rest same as template class complex
};
```

Parameters

_RealVal

The value of type **double** for the real part of the complex number being constructed.

_ImagVal

The value of type **double** for the imaginary part of the complex number being constructed.

_ComplexNum

The complex number of type **float** or of type **long double** whose real and imaginary parts are used to initialize a complex number of type **double** being constructed.

Return Value

A complex number of type **double**.

Remarks

The explicit specialization of the template class complex to a complex class of type **double** differs from the template class only in the constructors it defines. The conversion from **float** to **double** is allowed to be implicit, but the conversion from **long double** to **double** is required to be **explicit**. The use of **explicit** rules out the initiation with type conversion using assignment syntax.

For more information on the template class **complex**, see complex Class. For a list of members of the template class **complex**, see complex Members.

Example

```
// complex_comp_dbl.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // The first constructor specifies real & imaginary parts
   complex <double> c1 ( 4.0 , 5.0 );
   cout << "Specifying initial real & imaginary parts,\n"
        << " as type double gives c1 = " << c1 << endl;
```

```
    // The second constructor initializes values of the real &
    // imaginary parts using those of complex number of type float
    complex <float> c2float ( 4.0 , 5.0 );
    complex <double> c2double ( c2float );
    cout << "Implicit conversion from type float to type double,"
         << "\n gives c2double = " << c2double << endl;

    // The third constructor initializes values of the real &
    // imaginary parts using those of a complex number
    // of type long double
    complex <long double> c3longdouble ( 4.0 , 5.0 );
    complex <double> c3double ( c3longdouble );
    cout << "Explicit conversion from type float to type double,"
         << "\n gives c3longdouble = " << c3longdouble << endl;

    // The modulus and argument of a complex number can be recovered
    double absc3 = abs ( c3longdouble );
    double argc3 = arg ( c3longdouble );
    cout << "The modulus of c3 is recovered from c3 using: abs ( c3 ) = "
         << absc3 << endl;
    cout << "Argument of c3 is recovered from c3 using:\n arg ( c3 ) = "
         << argc3 << " radians, which is " << argc3 * 180 / pi
         << " degrees." << endl;
}
```

Output

```
Specifying initial real & imaginary parts,
 as type double gives c1 = (4,5)
Implicit conversion from type float to type double,
 gives c2double = (4,5)
Explicit conversion from type float to type double,
 gives c3longdouble = (4,5)
The modulus of c3 is recovered from c3 using: abs ( c3 ) = 6.40312
Argument of c3 is recovered from c3 using:
 arg ( c3 ) = 0.896055 radians, which is 51.3402 degrees.
```

Requirements

Header: <complex>

See Also

Reference

complex Class

Thread Safety in the Standard C++ Library

Other Resources

<complex> Members

complex Members

complex<float>

Describes an object that stores an ordered pair of objects both of type **float**, the first representing the real part of a complex number and the second representing the imaginary part.

```
template<>
   class complex<float> {
public:
   complex(
      double _RealVal = 0,
      double _ImagVal = 0
   );

   complex(
      const complex<double>& _ComplexNum
   );
   explicit complex(
      const complex<long double>& _ComplexNum
      );
   // rest same as template class complex
};
```

Parameters

_RealVal

The value of type **float** for the real part of the complex number being constructed.

_ImagVal

The value of type **float** for the imaginary part of the complex number being constructed.

_ComplexNum

The complex number of type **double** or of type **long double** whose real and imaginary parts are used to initialize a complex number of type **float** being constructed.

Return Value

A complex number of type **float**.

Remarks

The explicit specialization of the template class complex to a complex class of type **float** differs from the template class only in the constructors it defines. The conversion from **float** to **double** is allowed to be implicit, but the less safe conversion from **float** to **long double** is required to be **explicit**. The use of **explicit** rules out the initiation with type conversion using assignment syntax.

For more information on the template class **complex**, see complex Class. For a list of members of the template class **complex**, see complex Members.

Example

```
// complex_comp_flt.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // The first constructor specifies real & imaginary parts
   complex <float> c1 ( 4.0 , 5.0 );
   cout << "Specifying initial real & imaginary parts,\n"
        << " as type float gives c1 = " << c1 << endl;
```

```
    // The second constructor initializes values of the real &
    // imaginary parts using those of complex number of type double
    complex <double> c2double ( 1.0 , 3.0 );
    complex <float> c2float ( c2double );
    cout << "Implicit conversion from type double to type float,"
         << "\n gives c2float = " << c2float << endl;

    // The third constructor initializes values of the real &
    // imaginary parts using those of a complex number
    // of type long double
    complex <long double> c3longdouble ( 3.0 , 4.0 );
    complex <float> c3float ( c3longdouble );
    cout << "Explicit conversion from type long double to type float,"
         << "\n gives c3float = " << c3float << endl;

    // The modulus and argument of a complex number can be recovered
    double absc3 = abs ( c3float);
    double argc3 = arg ( c3float);
    cout << "The modulus of c3 is recovered from c3 using: abs ( c3 ) = "
         << absc3 << endl;
    cout << "Argument of c3 is recovered from c3 using:\n arg ( c3 ) = "
         << argc3 << " radians, which is " << argc3 * 180 / pi
         << " degrees." << endl;
}
```

Output

```
Specifying initial real & imaginary parts,
 as type float gives c1 = (4,5)
Implicit conversion from type double to type float,
 gives c2float = (1,3)
Explicit conversion from type long double to type float,
 gives c3float = (3,4)
The modulus of c3 is recovered from c3 using: abs ( c3 ) = 5
Argument of c3 is recovered from c3 using:
 arg ( c3 ) = 0.927295 radians, which is 53.1301 degrees.
```

Requirements

Header: <complex>

See Also

Reference

complex Class

Thread Safety in the Standard C++ Library

Other Resources

<complex> Members

complex Members

complex<long double>

Describes an object that stores an ordered pair of objects both of type **long double**, the first representing the real part of a complex number and the second representing the imaginary part.

```
template<>
   class complex<long double> {
public:
   complex(
      double _RealVal = 0,
      double _ImagVal = 0
   );

   complex(
      const complex<float>& _ComplexNum
   );
   explicit complex(
      const complex<double>& _ComplexNum
      );
   // rest same as template class complex
};
```

Parameters

_RealVal

The value of type **long double** for the real part of the complex number being constructed.

_ImagVal

The value of type **long double** for the imaginary part of the complex number being constructed.

_ComplexNum

The complex number of type **double** or of type **float** whose real and imaginary parts are used to initialize a complex number of type **long double** being constructed.

Return Value

A complex number of type **long double**.

Remarks

The explicit specialization of the template class complex to a complex class of type **long double** differs from the template class only in the constructors it defines. The conversion from **long double** to **float** is allowed to be implicit, but the conversion from **double** to **long double** is required to be **explicit**. The use of **explicit** rules out the initiation with type conversion using assignment syntax.

For more information on the template class **complex**, see complex Class. For a list of members of the template class **complex**, see complex Members.

Example

```
// complex_comp_ld.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // The first constructor specifies real & imaginary parts
   complex <long double> c1 ( 4.0 , 5.0 );
   cout << "Specifying initial real & imaginary parts,\n"
        << " as type float gives c1 = " << c1 << endl;
```

```
    // The second constructor initializes values of the real &
    // imaginary parts using those of complex number of type float
    complex <float> c2float ( 1.0 , 3.0 );
    complex <long double> c2longdouble ( c2float );
    cout << "Implicit conversion from type float to type long double,"
         << "\n gives c2longdouble = " << c2longdouble << endl;

    // The third constructor initializes values of the real &
    // imaginary parts using those of a complex number
    // of type double
    complex <double> c3double ( 3.0 , 4.0 );
    complex <long double> c3longdouble ( c3double );
    cout << "Implicit conversion from type long double to type float,"
         << "\n gives c3longdouble = " << c3longdouble << endl;

    // The modulus and argument of a complex number can be recovered
    double absc3 = abs ( c3longdouble );
    double argc3 = arg ( c3longdouble );
    cout << "The modulus of c3 is recovered from c3 using: abs ( c3 ) = "
         << absc3 << endl;
    cout << "Argument of c3 is recovered from c3 using:\n arg ( c3 ) = "
         << argc3 << " radians, which is " << argc3 * 180 / pi
         << " degrees." << endl;
}
```

Output

```
Specifying initial real & imaginary parts,
 as type float gives c1 = (4,5)
Implicit conversion from type float to type long double,
 gives c2longdouble = (1,3)
Implicit conversion from type long double to type float,
 gives c3longdouble = (3,4)
The modulus of c3 is recovered from c3 using: abs ( c3 ) = 5
Argument of c3 is recovered from c3 using:
 arg ( c3 ) = 0.927295 radians, which is 53.1301 degrees.
```

Requirements

Header: <complex>

See Also

Reference

complex Class

Thread Safety in the Standard C++ Library

Other Resources

<complex> Members

complex Members

<complex> Classes

For more information about classes in <complex>, see <complex> Members.

complex Class

The template class describes an object that stores two objects of type **Type**, one that represents the real part of a complex number and one that represents the imaginary part.

For a list of all members of this type, see complex Members.

```
template<class Type>
   class complex
```

Remarks

An object of class **Type**:

- Has a public default constructor, destructor, copy constructor, and assignment operator with conventional behavior.
- Can be assigned integer or floating-point values, or type cast to such values with conventional behavior.
- Defines the arithmetic operators and math functions, as needed, that are defined for the floating-point types with conventional behavior.

In particular, no subtle differences may exist between copy construction and default construction followed by assignment. None of the operations on objects of class **Type** may throw exceptions.

Explicit specializations of template class complex exist for the three floating-point types. In this implementation, a value of any other type **Type** is typecast to **double** for actual calculations, with the **double** result assigned back to the stored object of type **Type** .

Requirements

Header: <complex>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

complex Members

<complex> Members

complex Members

Reference

Constructors

complex	Constructs a complex number with specified real and imaginary parts or as a copy of some other complex number.

Typedefs

value_type	A type that represents the data type used to represent the real and imaginary parts of a complex number.

Member Functions

imag	Extracts the imaginary component of a complex number.
real	Extracts the real component of a complex number.

Operators

operator*=	Multiplies a target complex number by a factor, which may be complex or be the same type as are the real and imaginary parts of the complex number.
operator+=	Adds a number to a target complex number, where the number added may be complex or of the same type as are the real and imaginary parts of the complex number to which it is added.
operator-=	Subtracts a number from a target complex number, where the number subtracted may be complex or of the same type as are the real and imaginary parts of the complex number to which it is added.
operator/=	Divides a target complex number by a divisor, which may be complex or be the same type as are the real and imaginary parts of the complex number.
operator=	Assigns a number to a target complex number, where the number assigned may be complex or of the same type as are the real and imaginary parts of the complex number to which it is being assigned.

See Also

Reference

complex Class

Thread Safety in the Standard C++ Library

complex Typedefs

For information about the typedefs in the complex class, see complex Members.

complex::value_type

A type that represents the data type used to represent the real and imaginary parts of a complex number.

```
typedef Type value_type;
```

Remarks

value_type is a synonym for the class complex **Type** template parameter.

Example

```
// complex_valuetype.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   complex <double>::value_type a = 3, b = 4;

   complex <double> c1 ( a , b );
   cout << "Specifying initial real & imaginary parts"
      << "\nof type value_type: "
      << "c1 = " << c1 << "." << endl;
}
```

Output

```
Specifying initial real & imaginary parts
of type value_type: c1 = (3,4).
```

Requirements

Header: <complex>

See Also

Reference

complex Class

Other Resources

complex Members

complex Member Functions

For information about the member functions in the complex class, see complex Members.

complex::complex

Constructs a complex number with specified real and imaginary parts or as a copy of some other complex number.

```
complex(
    const Type& _RealVal = 0,
    const Type& _ImagVal = 0
);

template<class Other>
   complex(
      const complex<Other>& _ComplexNum
      );
```

Parameters

_RealVal

The value of the real part used to initialize the complex number being constructed.

_ImagVal

The value of the imaginary part used to initialize the complex number being constructed.

_ComplexNum

The complex number whose real and imaginary parts are used to initialize the complex number being constructed.

Remarks

The first constructor initializes the stored real part to *_RealVal* and the stored imaginary part to *_Imagval*. The second constructor initializes the stored real part to *_ComplexNum*.**real**() and the stored imaginary part to *_ComplexNum*.**imag**().

In this implementation, if a translator does not support member template functions, the template:

```
template<class Other>
   complex(const complex<Other>& right);
```

is replaced with:

```
   complex(const complex& right);
```

which is the copy constructor.

Example

```
// complex_complex.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // The first constructor specifies real & imaginary parts
   complex <double> c1 ( 4.0 , 5.0 );
   cout << "Specifying initial real & imaginary parts,"
        << "c1 = " << c1 << endl;

   // The second constructor initializes values of the real &
   // imaginary parts using those of another complex number
   complex <double> c2 ( c1 );
   cout << "Initializing with the real and imaginary parts of c1,"
```

```
        << " c2 = " << c2 << endl;

    // Complex numbers can be initialized in polar form
    // but will be stored in Cartesian form
    complex <double> c3 ( polar ( sqrt( (double)8 ) , pi / 4 ) );
    cout << "c3 = polar ( sqrt ( 8 ) , pi / 4 ) = " << c3 << endl;

    // The modulus and argument of a complex number can be recovered
    double absc3 = abs ( c3 );
    double argc3 = arg ( c3 );
    cout << "The modulus of c3 is recovered from c3 using: abs ( c3 ) = "
         << absc3 << endl;
    cout << "Argument of c3 is recovered from c3 using:\n arg ( c3 ) = "
         << argc3 << " radians, which is " << argc3 * 180 / pi
         << " degrees." << endl;
}
```

Output

```
Specifying initial real & imaginary parts,c1 = (4,5)
Initializing with the real and imaginary parts of c1, c2 = (4,5)
c3 = polar ( sqrt ( 8 ) , pi / 4 ) = (2,2)
The modulus of c3 is recovered from c3 using: abs ( c3 ) = 2.82843
Argument of c3 is recovered from c3 using:
 arg ( c3 ) = 0.785398 radians, which is 45 degrees.
```

Requirements

Header: <complex>

See Also

Reference

complex Class

Other Resources

complex Members

complex::imag

Extracts the imaginary component of a complex number.

```
Type imag( ) const;
```

Return Value

The imaginary part of the complex number.

Remarks

For a complex number *a* + *bi*, the imaginary part or component is *Im(a + bi) = b*.

Example

```
// complex_imag.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;

   complex <double> c1 ( 4.0 , 3.0 );
   cout << "The complex number c1 = " << c1 << endl;

   double dr1 = c1.real ( );
   cout << "The real part of c1 is c1.real ( ) = "
        << dr1 << "." << endl;

   double di1 = c1.imag ( );
   cout << "The imaginary part of c1 is c1.imag ( ) = "
        << di1 << "." << endl;
}
```

Output

```
The complex number c1 = (4,3)
The real part of c1 is c1.real ( ) = 4.
The imaginary part of c1 is c1.imag ( ) = 3.
```

Requirements

Header: <complex>

See Also

Reference

complex Class

Other Resources

complex Members

complex::real

Extracts the real component of a complex number.

```
Type real( ) const;
```

Return Value

The real part of the complex number.

Remarks

For a complex number *a* + *bi*, the real part or component is *Re(a + bi) = a*.

Example

```
// complex_class_real.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;

   complex <double> c1 ( 4.0 , 3.0 );
   cout << "The complex number c1 = " << c1 << endl;

   double dr1 = c1.real ( );
   cout << "The real part of c1 is c1.real ( ) = "
        << dr1 << "." << endl;

   double di1 = c1.imag ( );
   cout << "The imaginary part of c1 is c1.imag ( ) = "
        << di1 << "." << endl;
}
```

Output

```
The complex number c1 = (4,3)
The real part of c1 is c1.real ( ) = 4.
The imaginary part of c1 is c1.imag ( ) = 3.
```

Requirements

Header: <complex>

See Also

Reference

complex Class

Other Resources

complex Members

complex Operators

For more information about the operators in the complex class, see complex Members.

complex::operator*=

Multiplies a target complex number by a factor, which may be complex or be the same type as are the real and imaginary parts of the complex number.

```
template<class Other>
   complex& operator*=(
      const complex<Other>& _Right
   );
complex& operator*=(
   const Type& _Right
);
```

Parameters

_Right

A complex number or a number that is of the same type as the parameter of the target complex number.

Return Value

A complex number that has been multiplied by the number specified as a parameter.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

```
// complex_op_me.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main() {
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> multiplied by type complex<double>
   complex <double> cl1 ( polar ( 3.0 , pi / 6 ) );
   complex <double> cr1 ( polar ( 2.0 , pi / 3 ) );
   cout << "The left-side complex number is cl1 = " << cl1 << endl;
   cout << "The right-side complex number is cr1 = " << cr1 << endl;

   complex <double> cs1 = cl1 * cr1;
   cout << "Quotient of two complex numbers is: cs1 = cl1 * cr1 = "
        << cs1 << endl;

   // This is equivalent to the following operation
   cl1  *= cr1;
   cout << "Quotient of two complex numbers is also: cl1 *= cr1 = "
        << cl1 << endl;

   double abscl1 = abs ( cl1 );
   double argcl1 = arg ( cl1 );
   cout << "The modulus of cl1 is: " << abscl1 << endl;
   cout << "The argument of cl1 is: "<< argcl1 << " radians, which is "
        << argcl1 * 180 / pi << " degrees." << endl << endl;

   // Example of the second member function
   // type complex<double> multiplied by type double
   complex <double> cl2 ( polar ( 3.0 , pi / 6 ) );
   double cr2 = 5.0;
   cout << "The left-side complex number is cl2 = " << cl2 << endl;
```

```
    cout << "The right-side complex number is cr2 = " << cr2 << endl;

    complex <double> cs2 = cl2 * cr2;
    cout << "Quotient of two complex numbers is: cs2 = cl2 * cr2 = "
         << cs2 << endl;

    // This is equivalent to the following operation
    cl2  *= cr2;
    cout << "Quotient of two complex numbers is also: cl2 *= cr2 = "
         << cl2 << endl;

    double abscl2 = abs ( cl2 );
    double argcl2 = arg ( cl2 );
    cout << "The modulus of cl2 is: " << abscl2 << endl;
    cout << "The argument of cl2 is: "<< argcl2 << " radians, which is "
         << argcl2 * 180 / pi << " degrees." << endl;
}
```

Sample Output

```
The left-side complex number is cl1 = (2.59808,1.5)
The right-side complex number is cr1 = (1,1.73205)
Quotient of two complex numbers is: cs1 = cl1 * cr1 = (-6.21281e-013,6)
Quotient of two complex numbers is also: cl1 *= cr1 = (-6.21281e-013,6)
The modulus of cl1 is: 6
The argument of cl1 is: 1.5708 radians, which is 90 degrees.

The left-side complex number is cl2 = (2.59808,1.5)
The right-side complex number is cr2 = 5
Quotient of two complex numbers is: cs2 = cl2 * cr2 = (12.9904,7.5)
Quotient of two complex numbers is also: cl2 *= cr2 = (12.9904,7.5)
The modulus of cl2 is: 15
The argument of cl2 is: 0.523599 radians, which is 30 degrees.
```

Requirements

Header: <complex>

See Also

Reference

complex Class

Other Resources

complex Members

complex::operator+=

Adds a number to a target complex number, where the number added may be complex or of the same type as are the real and imaginary parts of the complex number to which it is added.

```
template<class Other>
   complex& operator+=(
      const complex<Other>& _Right
   );
complex& operator+=(
      const Type& _Right
      );
```

Parameters

_Right

A complex number or a number that is of the same type as the parameter of the target complex number.

Return Value

A complex number that has had the number specified as a parameter added.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

```
// complex_op_pe.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> added to type complex<double>
   complex <double> cl1 ( 3.0 , 4.0 );
   complex <double> cr1 ( 2.0 , -1.0 );
   cout << "The left-side complex number is cl1 = " << cl1 << endl;
   cout << "The right-side complex number is cr1 = " << cr1 << endl;

   complex <double> cs1 = cl1 + cr1;
   cout << "The sum of the two complex numbers is: cs1 = cl1 + cr1 = "
        << cs1 << endl;

   // This is equivalent to the following operation
   cl1 += cr1;
   cout << "The complex number cr1 added to the complex number cl1 is:"
        << "\n cl1 += cr1 = " << cl1 << endl;

   double abscl1 = abs ( cl1 );
   double argcl1 = arg ( cl1 );
   cout << "The modulus of cl1 is: " << abscl1 << endl;
   cout << "The argument of cl1 is: "<< argcl1 << " radians, which is "
        << argcl1 * 180 / pi << " degrees." << endl << endl;

   // Example of the second member function
   // type double added to type complex<double>
   complex <double> cl2 ( -2 , 4 );
   double cr2 =5.0;
```

```
    cout << "The left-side complex number is cl2 = " << cl2 << endl;
    cout << "The right-side complex number is cr2 = " << cr2 << endl;

    complex <double> cs2 = cl2 + cr2;
    cout << "The sum of the two complex numbers is: cs2 = cl2 + cr2 = "
         << cs2 << endl;

    // This is equivalent to the following operation
    cl2 += cr2;
    cout << "The complex number cr2 added to the complex number cl2 is:"
         << "\n cl2 += cr2 = " << cl2 << endl;

    double abscl2 = abs ( cl2 );
    double argcl2 = arg ( cl2 );
    cout << "The modulus of cl2 is: " << abscl2 << endl;
    cout << "The argument of cl2 is: "<< argcl2 << " radians, which is "
         << argcl2 * 180 / pi << " degrees." << endl << endl;
}
```

Output

```
The left-side complex number is cl1 = (3,4)
The right-side complex number is cr1 = (2,-1)
The sum of the two complex numbers is: cs1 = cl1 + cr1 = (5,3)
The complex number cr1 added to the complex number cl1 is:
 cl1 += cr1 = (5,3)
The modulus of cl1 is: 5.83095
The argument of cl1 is: 0.54042 radians, which is 30.9638 degrees.

The left-side complex number is cl2 = (-2,4)
The right-side complex number is cr2 = 5
The sum of the two complex numbers is: cs2 = cl2 + cr2 = (3,4)
The complex number cr2 added to the complex number cl2 is:
 cl2 += cr2 = (3,4)
The modulus of cl2 is: 5
The argument of cl2 is: 0.927295 radians, which is 53.1301 degrees.
```

Requirements

Header: <complex>

See Also

Reference

complex Class

Other Resources

complex Members

complex::operator-=

Subtracts a number from a target complex number, where the number subtracted may be complex or of the same type as are the real and imaginary parts of the complex number to which it is added.

```
template<class Other>
   complex& operator-=(
      const complex<Other>& _ComplexNum
   );
complex& operator-=(
   const Type& _RealPart
);
```

Parameters

_ComplexNum

A complex number to be subtracted from the target complex number.

_RealPart

A real number to be subtracted from the target complex number.

Return Value

A complex number that has had the number specified as a parameter subtracted from it.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

```
// complex_op_se.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> subtracted from type complex<double>
   complex <double> cl1 ( 3.0 , 4.0 );
   complex <double> cr1 ( 2.0 , -1.0 );
   cout << "The left-side complex number is cl1 = " << cl1 << endl;
   cout << "The right-side complex number is cr1 = " << cr1 << endl;

   complex <double> cs1 = cl1 - cr1;
   cout << "The difference between the two complex numbers is:"
        << "\n cs1 = cl1 - cr1 = " << cs1 << endl;

   // This is equivalent to the following operation
   cl1 -= cr1;
   cout << "Complex number cr1 subtracted from complex number cl1 is:"
        << "\n cl1 -= cr1 = " << cl1 << endl;

   double abscl1 = abs ( cl1 );
   double argcl1 = arg ( cl1 );
   cout << "The modulus of cl1 is: " << abscl1 << endl;
   cout << "The argument of cl1 is: "<< argcl1 << " radians, which is "
        << argcl1 * 180 / pi << " degrees." << endl << endl;

   // Example of the second member function
```

```
    // type double subtracted from type complex<double>
    complex <double> cl2 ( 2.0 , 4.0 );
    double cr2 = 5.0;
    cout << "The left-side complex number is cl2 = " << cl2 << endl;
    cout << "The right-side complex number is cr2 = " << cr2 << endl;

    complex <double> cs2 = cl2 - cr2;
    cout << "The difference between the two complex numbers is:"
         << "\n cs2 = cl2 - cr2 = " << cs2 << endl;

    // This is equivalent to the following operation
    cl2  -= cr2;
    cout << "Complex number cr2 subtracted from complex number cl2 is:"
         << "\n cl2 -= cr2 = " << cl2 << endl;

    double abscl2 = abs ( cl2 );
    double argcl2 = arg ( cl2 );
    cout << "The modulus of cl2 is: " << abscl2 << endl;
    cout << "The argument of cl2 is: "<< argcl2 << " radians, which is "
         << argcl2 * 180 / pi << " degrees." << endl << endl;
}
```

Output

```
The left-side complex number is cl1 = (3,4)
The right-side complex number is cr1 = (2,-1)
The difference between the two complex numbers is:
 cs1 = cl1 - cr1 = (1,5)
Complex number cr1 subtracted from complex number cl1 is:
 cl1 -= cr1 = (1,5)
The modulus of cl1 is: 5.09902
The argument of cl1 is: 1.3734 radians, which is 78.6901 degrees.

The left-side complex number is cl2 = (2,4)
The right-side complex number is cr2 = 5
The difference between the two complex numbers is:
 cs2 = cl2 - cr2 = (-3,4)
Complex number cr2 subtracted from complex number cl2 is:
 cl2 -= cr2 = (-3,4)
The modulus of cl2 is: 5
The argument of cl2 is: 2.2143 radians, which is 126.87 degrees.
```

Requirements

Header: <complex>

See Also

Reference

complex Class

Other Resources

complex Members

complex::operator/=

Divides a target complex number by a divisor, which may be complex or be the same type as are the real and imaginary parts of the complex number.

```
template<class Other>
   complex& operator/=(
      const complex<Other>& _ComplexNum
   );
complex& operator/=(
   const Type& _RealPart
   );
```

Parameters

_ComplexNum

A complex number to be subtracted from the target complex number.

_RealPart

A real number to be subtracted from the target complex number.

Return Value

A complex number that has been divided by the number specified as a parameter.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

```
// complex_op_de.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> divided by type complex<double>
   complex <double> cl1 ( polar (3.0 , pi / 6 ) );
   complex <double> cr1 ( polar (2.0 , pi / 3 ) );
   cout << "The left-side complex number is cl1 = " << cl1 << endl;
   cout << "The right-side complex number is cr1 = " << cr1 << endl;

   complex <double> cs1 = cl1 / cr1;
   cout << "The quotient of the two complex numbers is: cs1 = cl1 /cr1 = "
        << cs1 << endl;

   // This is equivalent to the following operation
   cl1 /= cr1;
   cout << "Quotient of two complex numbers is also: cl1 /= cr1 = "
        << cl1 << endl;

   double abscl1 = abs ( cl1 );
   double argcl1 = arg ( cl1 );
   cout << "The modulus of cl1 is: " << abscl1 << endl;
   cout << "The argument of cl1 is: "<< argcl1 << " radians, which is "
        << argcl1 * 180 / pi << " degrees." << endl << endl;

   // Example of the second member function
```

```
    // type complex<double> divided by type double
    complex <double> cl2 ( polar (3.0 , pi / 6 ) );
    double cr2 =5;
    cout << "The left-side complex number is cl2 = " << cl2 << endl;
    cout << "The right-side complex number is cr2 = " << cr2 << endl;

    complex <double> cs2 = cl2 / cr2;
    cout << "The quotient of the two complex numbers is: cs2 /= cl2 cr2 = "
         << cs2 << endl;

    // This is equivalent to the following operation
    cl2 /= cr2;
    cout << "Quotient of two complex numbers is also: cl2 = /cr2 = "
         << cl2 << endl;

    double abscl2 = abs ( cl2 );
    double argcl2 = arg ( cl2 );
    cout << "The modulus of cl2 is: " << abscl2 << endl;
    cout << "The argument of cl2 is: "<< argcl2 << " radians, which is "
         << argcl2 * 180 / pi << " degrees." << endl << endl;
}
```

Output

```
The left-side complex number is cl1 = (2.59808,1.5)
The right-side complex number is cr1 = (1,1.73205)
The quotient of the two complex numbers is: cs1 = cl1 /cr1 = (1.29904,-0.75)
Quotient of two complex numbers is also: cl1 /= cr1 = (1.29904,-0.75)
The modulus of cl1 is: 1.5
The argument of cl1 is: -0.523599 radians, which is -30 degrees.

The left-side complex number is cl2 = (2.59808,1.5)
The right-side complex number is cr2 = 5
The quotient of the two complex numbers is: cs2 /= cl2 cr2 = (0.519615,0.3)
Quotient of two complex numbers is also: cl2 = /cr2 = (0.519615,0.3)
The modulus of cl2 is: 0.6
The argument of cl2 is: 0.523599 radians, which is 30 degrees.
```

Requirements

Header: <complex>

See Also

Reference

complex Class

Other Resources

complex Members

complex::operator=

Assigns a number to a target complex number, where the number assigned may be complex or of the same type as are the real and imaginary parts of the complex number to which it is being assigned.

```
template<class Other>
   complex& operator=(
      const complex<Other>& _Right
   );
complex& operator=(
   const Type& _Right
);
```

Parameters

_Right

A complex number or a number that is of the same type as the parameter of the target complex number.

Return Value

A complex number that has been assigned the number specified as a parameter.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

```
// complex_op_as.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> assigned to type complex<double>
   complex <double> cl1 ( 3.0 , 4.0 );
   complex <double> cr1 ( 2.0 , -1.0 );
   cout << "The left-side complex number is cl1 = " << cl1 << endl;
   cout << "The right-side complex number is cr1 = " << cr1 << endl;

   cl1  = cr1;
   cout << "The complex number cr1 assigned to the complex number cl1 is:"
        << "\n cl1 = cr1 = " << cl1 << endl;

   // Example of the second member function
   // type double assigned to type complex<double>
   complex <double> cl2 ( -2 , 4 );
   double cr2 =5.0;
   cout << "The left-side complex number is cl2 = " << cl2 << endl;
   cout << "The right-side complex number is cr2 = " << cr2 << endl;

   cl2 = cr2;
   cout << "The complex number cr2 assigned to the complex number cl2 is:"
        << "\n cl2 = cr2 = " << cl2 << endl;

   cl2 = complex<double>(3.0, 4.0);
   cout << "The complex number (3, 4) assigned to the complex number cl2 is:"
        << "\n cl2 = " << cl2 << endl;
}
```

Output

```
The left-side complex number is cl1 = (3,4)
The right-side complex number is cr1 = (2,-1)
The complex number cr1 assigned to the complex number cl1 is:
 cl1 = cr1 = (2,-1)
The left-side complex number is cl2 = (-2,4)
The right-side complex number is cr2 = 5
The complex number cr2 assigned to the complex number cl2 is:
 cl2 = cr2 = (5,0)
The complex number (3, 4) assigned to the complex number cl2 is:
 cl2 = (3,4)
```

Requirements

Header: <complex>

See Also

Reference

complex Class

Other Resources

complex Members

<csetjmp>

Defines the macros traditionally defined in the Standard C library header <setjmp.h>.

```
#if <TRADITIONAL C HEADERS>
   #include <setjmp.h>
namespace std {
   using ::jmp_buf;
   using ::longjmp;
   using ::setjmp;
}
#endif
```

Remarks

Including this header also ensures that the names declared with external linkage in the Standard C library header are declared in the **std** namespace. In this implementation, the names may or may not also be declared in the global namespace, depending on the specific translation environment.

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

Other Resources

Standard C++ Library Header Files

<csignal>

Defines the macros traditionally defined in the Standard C library header <signal.h>.

```
#if <TRADITIONAL C HEADERS>
   #include <signal.h>
namespace std {
   using ::sig_atomic_t;
   using ::raise;
   using ::signal;
}
#endif
```

Remarks

Including this header also ensures that the names declared with external linkage in the Standard C library header are declared in the **std** namespace. In this implementation, the names may or may not also be declared in the global namespace, depending on the specific translation environment.

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

Other Resources

Standard C++ Library Header Files

<cstdarg>

Defines the macros traditionally defined in the Standard C library header <stdarg.h>.

```
#if <TRADITIONAL C HEADERS>
   #include <stdarg.h>
namespace std {
   using ::va_list;
}
#endif
```

Remarks

Including this header also ensures that the names declared with external linkage in the Standard C library header are declared in the **std** namespace. In this implementation, the names may or may not also be declared in the global namespace, depending on the specific translation environment.

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

Other Resources

Standard C++ Library Header Files

<cstddef>

Defines the macros traditionally defined in the Standard C library header <stddef.h>.

```
#if <TRADITIONAL C HEADERS>
   #include <stddef.h>
namespace std {
   using ::ptrdiff_t;
   using ::size_t;
}
#endif
```

Remarks

Including this header also ensures that the names declared with external linkage in the Standard C library header are declared in the **std** namespace. In this implementation, the names may or may not also be declared in the global namespace, depending on the specific translation environment.

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

Other Resources

Standard C++ Library Header Files

<cstdio>

Defines the macros traditionally defined in the Standard C library header <stdio.h>.

```
#if <TRADITIONAL C HEADERS>
   #include <stdio.h>
namespace std {
   using ::fclose;
   using ::feof;
   using ::ferror;
   using ::fflush;
   using ::fgetc;
   using ::fgetpos;
   using ::fgets;
   using ::FILE;
   using ::clearerr;
   using ::fopen;
   using ::fprintf;
   using ::fpos_t;
   using ::fputc;
   using ::fputs;
   using ::fread;
   using ::freopen;
   using ::fscanf;
   using ::fseek;
   using ::fsetpos;
   using ::ftell;
   using ::fwrite;
   using ::getc;
   using ::getchar;
   using ::gets;
   using ::perror;
   using ::putc;
   using ::putchar;
   using ::printf;
   using ::puts;
   using ::remove;
   using ::rename;
   using ::rewind;
   using ::scanf;
   using ::setbuf;
   using ::setvbuf;
   using ::size_t;
   using ::sprintf;
   using ::sscanf;
   using ::tmpfile;
   using ::tmpnam;
   using ::ungetc;
   using ::vfprintf;
   using ::vprintf;
   using ::vsprintf;
}
#endif
```

Remarks

Including this header also ensures that the names declared with external linkage in the Standard C library header are declared in the **std** namespace. In this implementation, the names may or may not also be declared in the global namespace, depending on the specific translation environment.

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

Other Resources

Standard C++ Library Header Files

<cstdlib>

Defines the macros traditionally defined in the Standard C library header <stdlib.h>.

```
#if <TRADITIONAL C HEADERS>
   #include <stdlib.h>
namespace std {
   using ::abort;
   using ::abs;
   using ::atexit;
   using ::atof;
   using ::atoi;
   using ::atol;
   using ::bsearch;
   using ::calloc;
   using ::div;
   using ::div_t;
   using ::exit;
   using ::free;
   using ::getenv;
   using ::labs;
   using ::ldiv;
   using ::ldiv_t;
   using ::malloc;
   using ::mblen;
   using ::mbstowcs;
   using ::mbtowc;
   using ::qsort;
   using ::rand;
   using ::realloc;
   using ::size_t;
   using ::srand;
   using ::strtod;
   using ::strtol;
   using ::strtoul;
   using ::system;
   using ::wcstombs;
   using ::wctomb;
  }
#endif
```

Remarks

Including this header also ensures that the names declared with external linkage in the Standard C library header are declared in the **std** namespace. In this implementation, the names may or may not also be declared in the global namespace, depending on the specific translation environment.

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

Other Resources

Standard C++ Library Header Files

<cstring>

Defines the macros traditionally defined in the Standard C library header <string.h>.

```
#if <TRADITIONAL C HEADERS>
   #include <string.h>
namespace std {
   using ::memchr;
   using ::memcmp;
   using ::memcpy;
   using ::memmove;
   using ::memset;
   using ::size_t;
   using ::strcat;
   using ::strchr;
   using ::strcmp;
   using ::strcoll;
   using ::strcpy;
   using ::strcspn;
   using ::strerror;
   using ::strlen;
   using ::strncat;
   using ::strncmp;
   using ::strncpy;
   using ::strpbrk;
   using ::strrchr;
   using ::strspn;
   using ::strstr;
   using ::strtok;
   using ::strxfrm;
}
#endif
```

Remarks

Including this header also ensures that the names declared with external linkage in the Standard C library header are declared in the **std** namespace. In this implementation, the names may or may not also be declared in the global namespace, depending on the specific translation environment.

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

Other Resources

Standard C++ Library Header Files

<ctime>

Defines the macros traditionally defined in the Standard C Library header <time.h>.

```
#if <TRADITIONAL C HEADERS>
   #include <time.h>
namespace std {
   using ::asctime;
   using ::clock;
   using ::clock_t;
   using ::ctime;
   using ::difftime;
   using ::gmtime;
   using ::localtime;
   using ::mktime;
   using ::size_t;
   using ::strftime;
   using ::time;
   using ::time_t;
   using ::tm;
}
#endif
```

Remarks

Including this header also ensures that the names declared with external linkage in the Standard C Library header are declared in the **std** namespace. In this implementation, the names may or may not also be declared in the global namespace, depending on the specific translation environment.

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

Other Resources

Standard C++ Library Header Files

<cwchar>

Defines the macros traditionally defined in the Standard C library header <wchar.h>.

```
#if <TRADITIONAL C HEADERS>
   #include <wchar.h>
namespace std {
   using ::btowc;
   using ::fgetwc;
   using ::fgetws;
   using ::fputwc;
   using ::fputws;
   using ::fwide;
   using ::fwprintf;
   using ::fwscanf;
   using ::getwc;
   using ::getwchar;
   using ::mbrlen;
   using ::mbrtowc;
   using ::mbsinit;
   using ::mbsrtowcs;
   using ::mbstate_t;
   using ::putwc;
   using ::putwchar;
   using ::size_t;
   using ::swprintf;
   using ::swscanf;
   using ::tm;
   using ::ungetwc;
   using ::vfwprintf;
   using ::vswprintf;
   using ::vwprintf;
   using ::wcrtomb;
   using ::wcscat;
   using ::wcschr;
   using ::wcscmp;
   using ::wcscoll;
   using ::wcscpy;
   using ::wcscspn;
   using ::wcsftime;
   using ::wcslen;
   using ::wcsncat;
   using ::wcsncmp;
   using ::wcsncpy;
   using ::wcspbrk;
   using ::wcsrchr;
   using ::wcsrtombs;
   using ::wcsspn;
   using ::wcsstr;
   using ::wcstod;
   using ::wcstok;
   using ::wcstol;
   using ::wcstoul;
   using ::wcsxfrm;
   using ::wctob;
   using ::wint_t;
   using ::wmemchr;
   using ::wmemcmp;
   using ::wmemcpy;
   using ::wmemmove;
   using ::wmemset;
   using ::wprintf;
   using ::wscanf;
}
```

```
#endif
```

Remarks

Including this header also ensures that the names declared with external linkage in the Standard C library header are declared in the **std** namespace. In this implementation, the names may or may not also be declared in the global namespace, depending on the specific translation environment.

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

Other Resources

Standard C++ Library Header Files

<cwctype>

Defines the macros traditionally defined in the Standard C library header <wctype.h>.

```
#if <TRADITIONAL C HEADERS>
   #include <wctype.h>
namespace std {
   using ::iswalnum;
   using ::iswalpha;
   using ::iswcntrl;
   using ::iswctype;
   using ::iswdigit;
   using ::iswgraph;
   using ::iswlower;
   using ::iswprint;
   using ::iswpunct;
   using ::iswspace;
   using ::iswupper;
   using ::iswxdigit;
   using ::towctrans;
   using ::towlower;
   using ::towupper;
   using ::wctrans;
   using ::wctrans_t;
   using ::wctype;
   using ::wctype_t;
   using ::wint_t;
}

#undef iswalnum
#undef iswalpha
#undef iswcntrl
#undef iswctype
#undef iswdigit
#undef iswgraph
#undef iswlower
#undef iswprint
#undef iswpunct
#undef iswspace
#undef iswupper
#undef iswxdigit
#undef towctrans
#undef towlower
#undef towupper
#undef wctrans
#undef wctype

#endif
```

Remarks

Including this header also ensures that the names declared with external linkage in the Standard C library header are declared in the **std** namespace. In this implementation, the names may or may not also be declared in the global namespace, depending on the specific translation environment.

See Also

Reference

Standard C++ Library Overview

Thread Safety in the Standard C++ Library

Other Resources

Standard C++ Library Header Files

<deque>

Defines the container template class deque and several supporting templates.

For a list of all members of this header, see <deque> Members.

```
#include <deque>
```

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<deque> Members

Standard C++ Library Header Files

<deque> Members

Reference

Operators

operator!=	Tests if the deque object on the left side of the operator is not equal to the deque object on the right side.
operator<	Tests if the deque object on the left side of the operator is less than the deque object on the right side.
operator<=	Tests if the deque object on the left side of the operator is less than or equal to the deque object on the right side.
operator==	Tests if the deque object on the left side of the operator is equal to the deque object on the right side.
operator>	Tests if the deque object on the left side of the operator is greater than the deque object on the right side.
operator>=	Tests if the deque object on the left side of the operator is greater than or equal to the deque object on the right side.

Functions

swap (<deque>)	Exchanges the elements of two deques.

Classes

deque Class	A template class of sequence containers that arrange elements of a given type in a linear arrangement and, like vectors, allow fast random access to any element and efficient insertion and deletion at the back of the container.

See Also

Reference

<deque>
Thread Safety in the Standard C++ Library
Standard Template Library

<deque> Operators

For more information about the operators in <deque>, see <deque> Members.

See Also

Reference

Standard Template Library

operator!= (<deque>)

Tests if the deque object on the left side of the operator is not equal to the deque object on the right side.

```
bool operator!=(
   const deque<Type, Allocator>& _Left,
   const deque<Type, Allocator>& _Right
);
```

Parameters

_Left

An object of type **deque**.

_Right

An object of type **deque**.

Return Value

true if the deque objects are not equal; **false** if the deque objects are equal.

Remarks

The comparison between deque objects is based on a pairwise comparison of their elements. Two deque objects are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// deque_op_ne.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1, c2;

   c1.push_back( 1 );
   c2.push_back( 2 );

   if ( c1 != c2 )
      cout << "The deques are not equal." << endl;
   else
      cout << "The deques are equal." << endl;
}
```

Output

```
The deques are not equal.
```

Requirements

Header: <deque>

See Also

Reference

Standard Template Library

Other Resources

<deque> Members

operator< (<deque>)

Tests if the deque object on the left side of the operator is less than the deque object on the right side.

```
bool operator<(
   const deque<Type, Allocator>& _Left,
   const deque<Type, Allocator>& _Right
);
```

Parameters

_Left

An object of type **deque**.

_Right

An object of type **deque**.

Return Value

true if the deque on the left side of the operator is less than and not equal to the deque on the right side of the operator; otherwise **false**.

Remarks

The comparison between deque objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// deque_op_lt.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1, c2;

   c1.push_back( 1 );
   c1.push_back( 2 );
   c1.push_back( 4 );

   c2.push_back( 1 );
   c2.push_back( 3 );

   if ( c1 < c2 )
      cout << "Deque c1 is less than deque c2." << endl;
   else
      cout << "Deque c1 is not less than deque c2." << endl;
}
```

Output

```
Deque c1 is less than deque c2.
```

Requirements

Header: <deque>

See Also

Reference

deque::operator== and deque::operator<

Standard Template Library

Other Resources

operator<= (<deque>)

Tests if the deque object on the left side of the operator is less than or equal to the deque object on the right side.

```
bool operator<=(
   const deque<Type, Allocator>& _Left,
   const deque<Type, Allocator>& _Right
);
```

Parameters

_Left

An object of type **deque**.

_Right

An object of type **deque**.

Return Value

true if the deque on the left side of the operator is less than or equal to the deque on the right side of the operator; otherwise **false**.

Remarks

The comparison between deque objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// deque_op_le.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1, c2;

   c1.push_back( 1 );
   c1.push_back( 2 );
   c1.push_back( 4 );

   c2.push_back( 1 );
   c2.push_back( 3 );

   if ( c1 <= c2 )
      cout << "Deque c1 is less than or equal to deque c2." << endl;
   else
      cout << "Deque c1 is greater than deque c2." << endl;
}
```

Output

```
Deque c1 is less than or equal to deque c2.
```

Requirements

Header: <deque>

See Also

Reference

Standard Template Library

Other Resources

<deque> Members

operator== (<deque>)

Tests if the deque object on the left side of the operator is equal to the deque object on the right side.

```
bool operator==(
   const deque<Type, Allocator>& _Left,
   const deque<Type, Allocator>& _Right
);
```

Parameters

_Left

An object of type **deque**.

_Right

An object of type **deque**.

Return Value

true if the deque on the left side of the operator is equal to the deque on the right side of the operator; otherwise **false**.

Remarks

The comparison between deque objects is based on a pairwise comparison of their elements. Two deques are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// deque_op_eq.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1, c2;

   c1.push_back( 1 );
   c2.push_back( 1 );

   if ( c1 == c2 )
      cout << "The deques are equal." << endl;
   else
      cout << "The deques are not equal." << endl;

   c1.push_back( 1 );
   if ( c1 == c2 )
      cout << "The deques are equal." << endl;
   else
      cout << "The deques are not equal." << endl;
}
```

Output

```
The deques are equal.
The deques are not equal.
```

Requirements

Header: <deque>

See Also

Reference

deque::operator== and deque::operator<

operator> (<deque>)

Tests if the deque object on the left side of the operator is greater than the deque object on the right side.

```
bool operator>(
   const deque<Type, Allocator>& _Left,
   const deque<Type, Allocator>& _Right
);
```

Parameters

_Left

An object of type **deque**.

_Right

An object of type **deque**.

Return Value

true if the deque on the left side of the operator is greater than the deque on the right side of the operator; otherwise **false**.

Remarks

The comparison between deque objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// deque_op_gt.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1, c2;

   c1.push_back( 1 );
   c1.push_back( 3 );
   c1.push_back( 1 );

   c2.push_back( 1 );
   c2.push_back( 2 );
   c2.push_back( 2 );

   if ( c1 > c2 )
      cout << "Deque c1 is greater than deque c2." << endl;
   else
      cout << "Deque c1 is not greater than deque c2." << endl;
}
```

Output

```
Deque c1 is greater than deque c2.
```

Requirements

Header: <deque>

See Also

Reference

Standard Template Library

Other Resources

<deque> Members

operator>= (<deque>)

Tests if the deque object on the left side of the operator is greater than or equal to the deque object on the right side.

```
bool operator>=(
   const deque<Type, Allocator>& _Left,
   const deque<Type, Allocator>& _Right
);
```

Parameters

_Left

An object of type **deque**.

_Right

An object of type **deque**.

Return Value

true if the deque on the left side of the operator is greater than or equal to the deque on the right side of the operator; otherwise **false**.

Remarks

The comparison between deque objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// deque_op_ge.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1, c2;

   c1.push_back( 1 );
   c1.push_back( 3 );
   c1.push_back( 1 );

   c2.push_back( 1 );
   c2.push_back( 2 );
   c2.push_back( 2 );

   if ( c1 >= c2 )
      cout << "Deque c1 is greater than or equal to deque c2." << endl;
   else
      cout << "Deque c1 is less than deque c2." << endl;
}
```

Output

```
Deque c1 is greater than or equal to deque c2.
```

Requirements

Header: <deque>

See Also

Reference

Standard Template Library

Other Resources

<deque> Functions

For more information about the operators in <deque>, see <deque> Members.

See Also

Reference

Standard Template Library

swap (<deque>)

Exchanges the elements of two deques.

```
void swap(
   deque<Type, Allocator>& _Left,
   deque<Type, Allocator>& _Right,
);
```

Parameters

_Left

An object of type **deque**.

_Right

An object of type **deque**.

Example

See the example for deque::swap.

Requirements

Header: <deque>

See Also

Reference

Standard Template Library

Other Resources

<deque> Members

<deque> Classes

For more information about the classes in <deque>, see <deque> Members.

See Also

Reference

Standard Template Library

deque Class

The Standard Template Library (STL) sequence container deque arranges elements of a given type in a linear arrangement and, like vectors, allow fast random access to any element and efficient insertion and deletion at the back of the container. However, unlike a vector, the deque class also supports efficient insertion and deletion at the front of the container.

For a list of all members of this type, see deque Class Members.

```
template <
   class Type,
   class Allocator=allocator<Type>
>
      class deque
```

Parameters

Type

The element data type to be stored in the deque.

Allocator

The type that represents the stored allocator object that encapsulates details about the deque's allocation and deallocation of memory. This argument is optional, and the default value is **allocator<Type>**.

Remarks

The choice of container type should be based in general on the type of searching and inserting required by the application. Vectors should be the preferred container for managing a sequence when random access to any element is at a premium and insertions or deletions of elements are only required at the end of a sequence. The performance of the list container is superior when efficient insertions and deletions (in constant time) at any location within the sequence is at a premium. Such operations in the middle of the sequence require element copies and assignments proportional to the number of elements in the sequence (linear time).

Deque reallocation occurs when a member function must insert or erase elements of the sequence:

- If an element is inserted into an empty sequence, or if an element is erased to leave an empty sequence, then iterators earlier returned by begin and end become invalid.
- If an element is inserted at the first position of the deque, then all iterators, but no references, that designate existing elements become invalid.
- If an element is inserted at the end of the deque, then end and all iterators, but no references, that designate existing elements become invalid.
- If an element is erased at the front of the deque, only that iterator and references to the erased element become invalid.
- If the last element is erased from the end of the deque, only that iterator to the final element and references to the erased element become invalid.

Otherwise, inserting or erasing an element invalidates all iterators and references.

Requirements

Header: <deque>

See Also

Reference

Thread Safety in the Standard C++ Library
Standard Template Library

Other Resources

<deque> Members
deque Class Members

deque Class Members

Reference

Constructors

deque	Constructs a **deque** of a specific size or with elements of a specific value or with a specific **allocator** or as a copy of all or part of some other **deque**.

Typedefs

allocator_type	A type that represents the **allocator** class for the **deque** object.
const_iterator	A type that provides a random-access iterator that can access and read a **const** element in the **deque**.
const_pointer	A type that provides a pointer to a **const** element in a **deque**.
const_reference	A type that provides a reference to a **const** element stored in a **deque** for reading and performing **const** operations.
const_reverse_iterator	A type that provides a random-access iterator that can read any **const** element in the **deque**.
difference_type	A type that provides the difference between two iterators that refer to elements within the same **deque**.
iterator	A type that provides a random-access iterator that can read or modify any element in a **deque**.
pointer	A type that provides a pointer to an element in a **deque**.
reference	A type that provides a reference to an element stored in a **deque**.
reverse_iterator	A type that provides a random-access iterator that can read or modify an element in a reversed **deque**.
size_type	A type that counts the number of elements in a **deque**.
value_type	A type that represents the data type stored in a **deque**.

Member Functions

assign	Erases elements from a **deque** and copies a new set of elements to the target **deque**.
at	Returns a reference to the element at a specified location in the **deque**.
back	Returns a reference to the last element of the **deque**.
begin	Returns an iterator addressing the first element in the **deque**.
clear	Erases all the elements of a **deque**.
empty	Tests if a **deque** is empty.
end	Returns an iterator that addresses the location succeeding the last element in a **deque**.
erase	Removes an element or a range of elements in a **deque** from specified positions.
front	Returns a reference to the first element in a **deque**.

get_allocator	Returns a copy of the **allocator** object used to construct the **deque**.
insert	Inserts an element or a number of elements or a range of elements into the **deque** at a specified position.
max_size	Returns the maximum length of the **deque**.
pop_back	Deletes the element at the end of the **deque**.
pop_front	Deletes the element at the beginning of the **deque**.
push_back	Adds an element to the end of the **deque**.
push_front	Adds an element to the beginning of the **deque**.
rbegin	Returns an iterator to the first element in a reversed **deque**.
rend	Returns an iterator that points just beyond the last element in a reversed **deque**.
resize	Specifies a new size for a **deque**.
size	Returns the number of elements in the **deque**.
swap	Exchanges the elements of two **deque**s.

Operators

operator[]	Returns a reference to the **deque** element at a specified position.

See Also

Reference

deque Class

Thread Safety in the Standard C++ Library

Standard Template Library

deque Typedefs

For more information about the typedefs in deque, see deque Class Members.

See Also

Reference

Standard Template Library

deque::allocator_type

A type that represents the allocator class for the deque object.

```
typedef Allocator allocator_type
```

Remarks

allocator_type is a synonym for the template parameter **Allocator**.

Example

See the example for get_allocator.

Requirements

Header: <deque>

See Also

Reference

deque Class

Standard Template Library

Other Resources

deque Class Members

deque::const_iterator

A type that provides a random-access iterator that can access and read a **const** element in the deque.

```
typedef implementation-defined const_iterator;
```

Remarks

A type **const_iterator** cannot be used to modify the value of an element.

Example

See the example for back.

Requirements

Header: <deque>

See Also

Reference

deque Class

Standard Template Library

Other Resources

deque Class Members

deque::const_pointer

A type that provides a pointer to a **const** element in a deque.

```
typedef typename Allocator::const_pointer const_pointer;
```

Remarks

A type **const_pointer** cannot be used to modify the value of an element. An iterator is more commonly used to access a deque element.

Requirements

Header: <deque>

See Also

Reference

deque Class

Standard Template Library

Other Resources

deque Class Members

deque::const_reference

A type that provides a reference to a **const** element stored in a deque for reading and performing **const** operations.

```
typedef typename Allocator::const_reference const_reference;
```

Remarks

A type **const_reference** cannot be used to modify the value of an element.

Example

```
// deque_const_ref.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );

   const deque <int> c2 = c1;
   const int &i = c2.front( );
   const int &j = c2.back( );
   cout << "The first element is " << i << endl;
   cout << "The second element is " << j << endl;

   // The following line would cause an error as c2 is const
   // c2.push_back( 30 );
}
```

Output

```
The first element is 10
The second element is 20
```

Requirements

Header: <deque>

See Also

Reference

deque Class

Standard Template Library

Other Resources

deque Class Members

deque::const_reverse_iterator

A type that provides a random-access iterator that can read any **const** element in the deque.

```
typedef reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type **const_reverse_iterator** cannot modify the value of an element and is used to iterate through the deque in reverse.

Example

See the example for rbegin for an example of how to declare and use an iterator.

Requirements

Header: <deque>

See Also

Reference

deque Class

Standard Template Library

Other Resources

deque Class Members

deque::difference_type

A type that provides the difference between two iterators that refer to elements within the same deque.

```
typedef implementation-defined difference_type;
```

Remarks

A **difference_type** can also be described as the number of elements between two pointers.

Example

```
// deque_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <deque>
#include <algorithm>

int main( )
{
   using namespace std;

   deque <int> c1;
   deque <int>::iterator c1_Iter, c2_Iter;

   c1.push_back( 30 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1.push_back( 10 );
   c1.push_back( 30 );
   c1.push_back( 20 );

   c1_Iter = c1.begin( );
   c2_Iter = c1.end( );

   deque <int>::difference_type df_typ1, df_typ2, df_typ3;

   df_typ1 = count( c1_Iter, c2_Iter, 10 );
   df_typ2 = count( c1_Iter, c2_Iter, 20 );
   df_typ3 = count( c1_Iter, c2_Iter, 30 );
   cout << "The number '10' is in c1 collection " << df_typ1 << " times.\n";
   cout << "The number '20' is in c1 collection " << df_typ2 << " times.\n";
   cout << "The number '30' is in c1 collection " << df_typ3 << " times.\n";
}
```

Output

```
The number '10' is in c1 collection 1 times.
The number '20' is in c1 collection 2 times.
The number '30' is in c1 collection 3 times.
```

Requirements

Header: <deque>

See Also

Reference

deque Class

Standard Template Library

Other Resources

deque Class Members

deque::iterator

A type that provides a random-access iterator that can read or modify any element in a deque.

```
typedef implementation-defined iterator;
```

Remarks

A type **iterator** can be used to modify the value of an element.

Example

See the example for begin.

Requirements

Header: <deque>

See Also

Reference

deque Class

Standard Template Library

Other Resources

deque Class Members

deque::pointer

A type that provides a pointer to an element in a deque.

```
typedef typename Allocator::pointer pointer;
```

Remarks

A type **pointer** can be used to modify the value of an element. An iterator is more commonly used to access a deque element.

Requirements

Header: <deque>

See Also

Reference

deque Class

Standard Template Library

Other Resources

deque Class Members

deque::reference

A type that provides a reference to an element stored in a deque.

```
typedef typename Allocator::reference reference;
```

Example

```
// deque_reference.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );

   const int &i = c1.front( );
   int &j = c1.back( );
   cout << "The first element is " << i << endl;
   cout << "The second element is " << j << endl;
}
```

Output

```
The first element is 10
The second element is 20
```

Requirements

Header: <deque>

See Also

Reference

deque Class

Standard Template Library

Other Resources

deque Class Members

deque::reverse_iterator

A type that provides a random-access iterator that can read or modify an element in a reversed deque.

```
typedef reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type **reverse_iterator** is use to iterate through the deque.

Example

See the example for rbegin.

Requirements

Header: <deque>

See Also

Reference

deque Class

Standard Template Library

Other Resources

deque Class Members

deque::size_type

A type that counts the number of elements in a deque.

```
typedef implementation-defined size_type;
```

Example

See the example for size.

Requirements

Header: <deque>

See Also

Reference

deque Class

Standard Template Library

Other Resources

deque Class Members

deque::value_type

A type that represents the data type stored in a deque.

```
typedef Type value_type;
```

Remarks

value_type is a synonym for the template parameter **Type**.

Example

```
// deque_value_type.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>
int main( )
{
   using namespace std;
   deque<int>::value_type AnInt;
   AnInt = 44;
   cout << AnInt << endl;
}
```

Output

```
44
```

Requirements

Header: <deque>

See Also

Reference

deque Class

Standard Template Library

Other Resources

deque Class Members

deque Member Functions

For more information about the member functions in deque, see deque Class Members.

See Also

Reference

Standard Template Library

deque::assign

Erases elements from a deque and copies a new set of elements to the target deque.

```
template<class InputIterator>
   void assign(
      InputIterator _First,
      InputIterator _Last
   );
void assign(
   size_type _Count,
   const Type& _Val
);
```

Parameters

_First

Position of the first element in the range of elements to be copied from the argument deque.

_Last

Position of the first element beyond the range of elements to be copied from the argument deque.

_Count

The number of copies of an element being inserted into the deque.

_Val

The value of the element being inserted into the deque.

Remarks

After erasing any existing elements in the target deque, **assign** either inserts a specified range of elements from the original deque or from some other deque into the target deque or inserts copies of a new element of a specified value into the target deque.

Example

```
// deque_assign.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1, c2;
   deque <int>::const_iterator cIter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c2.push_back( 40 );
   c2.push_back( 50 );
   c2.push_back( 60 );

   cout << "c1 =";
   for ( cIter = c1.begin( ); cIter != c1.end( ); cIter++ )
      cout << " " << *cIter;
   cout << endl;

   c1.assign( ++c2.begin( ), c2.end( ) );
   cout << "c1 =";
   for ( cIter = c1.begin( ); cIter != c1.end( ); cIter++ )
      cout << " " << *cIter;
```

```
    cout << endl;

    c1.assign( 7, 4 );
    cout << "c1 =";
    for ( cIter = c1.begin( ); cIter != c1.end( ); cIter++ )
       cout << " " << *cIter;
    cout << endl;
  }
```

Output

```
c1 = 10 20 30
c1 = 50 60
c1 = 4 4 4 4 4 4 4
```

Requirements

Header: <deque>

See Also

Reference

deque Class

deque::assign and deque::swap

Standard Template Library

Other Resources

deque Class Members

deque::at

Returns a reference to the element at a specified location in the deque.

```
reference at(
   size_type _Pos
);
const_reference at(
   size_type _Pos
) const;
```

Parameters

_Pos

The subscript (or position number) of the element to reference in the deque.

Return Value

If *_Pos* is greater than the size of the deque, **at** throws an exception.

Return Value

If the return value of **at** is assigned to a **const_reference**, the deque object cannot be modified. If the return value of **at** is assigned to a **reference**, the deque object can be modified.

Example

```
// deque_at.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );

   const int& i = c1.at( 0 );
   int& j = c1.at( 1 );
   cout << "The first element is " << i << endl;
   cout << "The second element is " << j << endl;
}
```

Output

```
The first element is 10
The second element is 20
```

Requirements

Header: <deque>

See Also

Reference

deque Class

deque::operator[] and deque::at

Standard Template Library

Other Resources

deque Class Members

deque::back

Returns a reference to the last element of the deque.

```
reference back( );
const_reference back( ) const;
```

Return Value

The last element of the deque. If the deque is empty, the return value is undefined.

Remarks

If the return value of **back** is assigned to a **const_reference**, the deque object cannot be modified. If the return value of **back** is assigned to a **reference**, the deque object can be modified.

When compiling with _SECURE_SCL 1, a runtime error will occur if you attempt to access an element in an empty deque. See Checked Iterators for more information.

Example

```
// deque_back.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   c1.push_back( 11 );

   int& i = c1.back( );
   const int& ii = c1.front( );

   cout << "The last integer of c1 is " << i << endl;
   i--;
   cout << "The next-to-last integer of c1 is " << ii << endl;
}
```

Output

```
The last integer of c1 is 11
The next-to-last integer of c1 is 10
```

Requirements

Header: <deque>

See Also

Reference

deque Class

deque::front and deque::back

Standard Template Library

Other Resources

deque Class Members

deque::begin

Returns an iterator addressing the first element in the deque.

```
const_iterator begin( ) const;
iterator begin( );
```

Return Value

A random-access iterator addressing the first element in the deque or to the location succeeding an empty deque.

Remarks

If the return value of **begin** is assigned to a **const_iterator**, the deque object cannot be modified. If the return value of **begin** is assigned to an **iterator**, the deque object can be modified.

Example

```
// deque_begin.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;
   deque <int>::iterator c1_Iter;
   deque <int>::const_iterator c1_cIter;

   c1.push_back( 1 );
   c1.push_back( 2 );

   c1_Iter = c1.begin( );
   cout << "The first element of c1 is " << *c1_Iter << endl;

   *c1_Iter = 20;
   c1_Iter = c1.begin( );
   cout << "The first element of c1 is now " << *c1_Iter << endl;

   // The following line would be an error because iterator is const
   // *c1_cIter = 200;
}
```

Output

```
The first element of c1 is 1
The first element of c1 is now 20
```

Requirements

Header: <deque>

See Also

Reference

deque Class

deque::begin and deque::end

Standard Template Library

Other Resources

deque Class Members

deque::clear

Erases all the elements of a deque.

```
void clear( );
```

Example

```
// deque_clear.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   cout << "The size of the deque is initially " << c1.size( ) << endl;
   c1.clear( );
   cout << "The size of the deque after clearing is " << c1.size( ) << endl;
}
```

Output

```
The size of the deque is initially 3
The size of the deque after clearing is 0
```

Requirements

Header: <deque>

See Also

Reference

deque Class

deque::erase and deque::clear

Standard Template Library

Other Resources

deque Class Members

deque::deque

Constructs a deque of a specific size or with elements of a specific value or with a specific allocator or as a copy of all or part of some other deque.

```
deque( );
explicit deque(
   const Allocator& _Al
);
explicit deque(
   size_type _Count
);
deque(
   size_type _Count,
   const Type& _Val
);
deque(
   size_type _Count,
   const Type& _Val,
   const Allocator& _Al
);
deque(
   const _deque& _Right
);
template<class InputIterator>
   deque(
      InputIterator _First,
      InputIterator _Last
   );
template<class InputIterator>
   deque(
      InputIterator _First,
      InputIterator _Last,
      const Allocator& _Al
   );
```

Parameters

_Al

The allocator class to use with this object.

_Count

The number of elements in the constructed deque.

_Val

The value of the elements in the constructed deque.

_Right

The deque of which the constructed deque is to be a copy.

_First

Position of the first element in the range of elements to be copied.

_Last

Position of the first element beyond the range of elements to be copied.

Remarks

All constructors store an allocator object (*_Al*) and initialize the deque.

The first two constructors specify an empty initial deque, the second specifying the allocator type (*_Al*) to be used.

The third constructor specifies a repetition of a specified number (*_Count*) of elements of the default value for class **Type**.

The fourth and fifth constructors specify a repetition of (*_Count*) elements of value *_Val*.

The sixth constructor specifies a copy of the deque *_Right*.

The last two constructors copy the range [*_First, _Last*) of a deque.

None of the constructors perform any interim reallocations.

Example

```
// deque_deque.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int>::iterator c1_Iter, c2_Iter, c3_Iter, c4_Iter, c5_Iter, c6_Iter;

   // Create an empty deque c0
   deque <int> c0;

   // Create a deque c1 with 3 elements of default value 0
   deque <int> c1( 3 );

   // Create a deque c2 with 5 elements of value 2
   deque <int> c2( 5, 2 );

   // Create a deque c3 with 3 elements of value 1 and with the
   // allocator of deque c2
   deque <int> c3( 3, 1, c2.get_allocator( ) );

   // Create a copy, deque c4, of deque c2
   deque <int> c4( c2 );

   // Create a deque c5 by copying the range c4[_First, _Last)
   c4_Iter = c4.begin( );
   c4_Iter++;
   c4_Iter++;
   deque <int> c5( c4.begin( ), c4_Iter );

   // Create a deque c6 by copying the range c4[_First, _Last) and
   // c2 with the allocator of deque
   c4_Iter = c4.begin( );
   c4_Iter++;
   c4_Iter++;
   c4_Iter++;
   deque <int> c6( c4.begin( ), c4_Iter, c2.get_allocator( ) );

   cout << "c1 = ";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << *c1_Iter << " ";
   cout << endl;

   cout << "c2 = ";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << *c2_Iter << " ";
   cout << endl;

   cout << "c3 = ";
   for ( c3_Iter = c3.begin( ); c3_Iter != c3.end( ); c3_Iter++ )
      cout << *c3_Iter << " ";
   cout << endl;

   cout << "c4 = ";
   for ( c4_Iter = c4.begin( ); c4_Iter != c4.end( ); c4_Iter++ )
      cout << *c4_Iter << " ";
   cout << endl;
```

```
    cout << "c5 = ";
    for ( c5_Iter = c5.begin( ); c5_Iter != c5.end( ); c5_Iter++ )
       cout << *c5_Iter << " ";
    cout << endl;

    cout << "c6 = ";
    for ( c6_Iter = c6.begin( ); c6_Iter != c6.end( ); c6_Iter++ )
       cout << *c6_Iter << " ";
    cout << endl;
 }
```

Output

```
c1 = 0 0 0
c2 = 2 2 2 2 2
c3 = 1 1 1
c4 = 2 2 2 2 2
c5 = 2 2
c6 = 2 2 2
```

Requirements

Header: <deque>

See Also

Reference

deque Class

Standard Template Library

Other Resources

deque Class Members

deque::empty

Tests if a deque is empty.

```
bool empty( ) const;
```

Return Value

true if the deque is empty; **false** if the deque is not empty.

Example

```
// deque_empty.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   if ( c1.empty( ) )
      cout << "The deque is empty." << endl;
   else
      cout << "The deque is not empty." << endl;
}
```

Output

```
The deque is not empty.
```

Requirements

Header: <deque>

See Also

Reference

deque Class

Standard Template Library

Other Resources

deque Class Members

deque::end

Returns an iterator that addresses the location succeeding the last element in a deque.

```
const_iterator end( ) const;
iterator end( );
```

Return Value

A random-access iterator that addresses the location succeeding the last element in a deque. If the deque is empty, then deque::end == deque::begin.

Remarks

end is used to test whether an iterator has reached the end of its deque.

Example

```
// deque_end.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;
   deque <int>::iterator c1_Iter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   c1_Iter = c1.end( );
   c1_Iter--;
   cout << "The last integer of c1 is " << *c1_Iter << endl;

   c1_Iter--;
   *c1_Iter = 400;
   cout << "The new next-to-last integer of c1 is " << *c1_Iter << endl;

   // If a const iterator had been declared instead with the line:
   // deque <int>::const_iterator c1_Iter;
   // an error would have resulted when inserting the 400

   cout << "The deque is now:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
}
```

Output

```
The last integer of c1 is 30
The new next-to-last integer of c1 is 400
The deque is now: 10 400 30
```

Requirements

Header: <deque>

See Also

Reference

deque Class

deque::begin and deque::end

Standard Template Library

Other Resources

deque Class Members

deque::erase

Removes an element or a range of elements in a deque from specified positions.

```
iterator erase(
   iterator _Where
);
iterator erase(
   iterator _First,
   iterator _Last
);
```

Parameters

_Where

Position of the element to be removed from the deque.

_First

Position of the first element removed from the deque.

_Last

Position just beyond the last element removed from the deque.

Return Value

A random-access iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the deque if no such element exists.

Remarks

For more information on **erase**, see deque::erase and deque::clear.

erase never throws an exception.

Example

```
// deque_erase.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;
   deque <int>::iterator Iter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1.push_back( 40 );
   c1.push_back( 50 );
   cout << "The initial deque is: ";
   for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
      cout << *Iter << " ";
   cout << endl;
   c1.erase( c1.begin( ) );
   cout << "After erasing the first element, the deque becomes:  ";
   for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
      cout << *Iter << " ";
   cout << endl;
   Iter = c1.begin( );
   Iter++;
   c1.erase( Iter, c1.end( ) );
   cout << "After erasing all elements but the first, deque becomes: ";
```

```
    for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
       cout << *Iter << " ";
    cout << endl;
 }
```

Output

```
The initial deque is: 10 20 30 40 50
After erasing the first element, the deque becomes:  20 30 40 50
After erasing all elements but the first, deque becomes: 20
```

Requirements

Header: <deque>

See Also

Reference

deque Class

deque::erase and deque::clear

Standard Template Library

Other Resources

deque Class Members

deque::front

Returns a reference to the first element in a deque.

```
reference front( );
const_reference front( ) const;
```

Return Value

If the deque is empty, the return is undefined.

Remarks

If the return value of **front** is assigned to a **const_reference**, the deque object cannot be modified. If the return value of **front** is assigned to a **reference**, the deque object can be modified.

When compiling with _SECURE_SCL 1, a runtime error will occur if you attempt to access an element in an empty deque. See Checked Iterators for more information.

Example

```
// deque_front.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   c1.push_back( 11 );

   int& i = c1.front( );
   const int& ii = c1.front( );

   cout << "The first integer of c1 is " << i << endl;
   i++;
   cout << "The second integer of c1 is " << ii << endl;
}
```

Output

```
The first integer of c1 is 10
The second integer of c1 is 11
```

Requirements

Header: <deque>

See Also

Reference

deque Class

deque::front and deque::back

Standard Template Library

Other Resources

deque Class Members

deque::get_allocator

Returns a copy of the allocator object used to construct the deque.

```
Allocator get_allocator( ) const;
```

Return Value

The allocator used by the deque.

Remarks

Allocators for the deque class specify how the class manages storage. The default allocators supplied with STL container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```
// deque_get_allocator.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   // The following lines declare objects that use the default allocator.
   deque <int> c1;
   deque <int, allocator<int> > c2 = deque <int, allocator<int> >( allocator<int>( ) );

   // c3 will use the same allocator class as c1
   deque <int> c3( c1.get_allocator( ) );

   deque <int>::allocator_type xlst = c1.get_allocator( );
   // You can now call functions on the allocator class used by c1
}
```

Requirements

Header: <deque>

See Also

Reference

deque Class

Standard Template Library

Other Resources

deque Class Members

deque::insert

Inserts an element or a number of elements or a range of elements into the deque at a specified position.

```
iterator insert(
   iterator _Where,
   const Type& _Val
);
void insert(
   iterator _Where,
   size_type _Count,
   const Type& _Val
);
template<class InputIterator>
   void insert(
      iterator _Where,
      InputIterator _First,
      InputIterator _Last
   );
```

Parameters

_Where

The position in the target deque where the first element is inserted.

_Val

The value of the element being inserted into the deque.

_Count

The number of elements being inserted into the deque.

_First

The position of the first element in the range of elements in the argument deque to be copied.

_Last

The position of the first element beyond the range of elements in the argument deque to be copied.

Return Value

The first **insert** function returns an iterator that points to the position where the new element was inserted into the deque.

Remarks

Any insertion operation can be expensive.

Example

```
// deque_insert.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1, c2;
   deque <int>::iterator Iter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c2.push_back( 40 );
   c2.push_back( 50 );
   c2.push_back( 60 );
```

```
    cout << "[ c1 = ";
    for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
       cout << *Iter << " ";
    cout << "]" << endl;

    Iter = c1.begin( );
    Iter++;
    c1.insert( Iter, 100 );
    cout << "[ c1 = ";
    for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
       cout << *Iter << " ";
    cout << "]" << endl;

    Iter = c1.begin( );
    Iter++;
    Iter++;
    c1.insert( Iter, 2, 200 );

    cout << "[ c1 = ";
    for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
       cout << *Iter << " ";
    cout << "]" << endl;

    c1.insert( ++c1.begin( ), c2.begin( ),--c2.end( ) );

    cout << "[ c1 = ";
    for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
       cout << *Iter << " ";
    cout << "]" << endl;
 }
```

Output

```
[ c1 = 10 20 30 ]
[ c1 = 10 100 20 30 ]
[ c1 = 10 100 200 200 20 30 ]
[ c1 = 10 40 50 100 200 200 20 30 ]
```

Requirements

Header: <deque>

See Also

Reference

deque Class

deque::insert (STL Samples)

Standard Template Library

Other Resources

deque Class Members

deque::max_size

Returns the maximum length of the deque.

```
size_type max_size( ) const;
```

Return Value

The maximum possible length of the deque.

Example

```
// deque_max_size.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;
   deque <int>::size_type i;

   i = c1.max_size( );
   cout << "The maximum possible length of the deque is " << i << "." << endl;
}
```

Sample Output

```
The maximum possible length of the deque is 1073741823.
```

Requirements

Header: <deque>

See Also

Reference

deque Class

Standard Template Library

Other Resources

deque Class Members

deque::pop_back

Deletes the element at the end of the deque.

```
void pop_back( );
```

Remarks

The last element must not be empty. **pop_back** never throws an exception.

Example

```
// deque_pop_back.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 1 );
   c1.push_back( 2 );
   cout << "The first element is: " << c1.front( ) << endl;
   cout << "The last element is: " << c1.back( ) << endl;

   c1.pop_back( );
   cout << "After deleting the element at the end of the deque, the "
      "last element is: " << c1.back( ) << endl;
}
```

Output

```
The first element is: 1
The last element is: 2
After deleting the element at the end of the deque, the last element is: 1
```

Requirements

Header: <deque>

See Also

Reference

deque Class

deque::push_back and deque::pop_back

Standard Template Library

Other Resources

deque Class Members

deque::pop_front

Deletes the element at the beginning of the deque.

```
void pop_front( );
```

Remarks

The first element must not be empty. **pop_front** never throws an exception.

Example

```
// deque_pop_front.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 1 );
   c1.push_back( 2 );
   cout << "The first element is: " << c1.front( ) << endl;
   cout << "The second element is: " << c1.back( ) << endl;

   c1.pop_front( );
   cout << "After deleting the element at the beginning of the "
      "deque, the first element is: " << c1.front( ) << endl;
}
```

Output

```
The first element is: 1
The second element is: 2
After deleting the element at the beginning of the deque, the first element is: 2
```

Requirements

Header: <deque>

See Also

Reference

deque Class

deque::push_front and deque::pop_front

Standard Template Library

Other Resources

deque Class Members

deque::push_back

Adds an element to the end of the deque.

```
void push_back(
   const Type& _Val
);
```

Parameters

_Val

The element added to the end of the deque.

Remarks

If an exception is thrown, the deque is left unaltered and the exception is rethrown.

Example

```
// deque_push_back.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 1 );
   if ( c1.size( ) != 0 )
      cout << "Last element: " << c1.back( ) << endl;

   c1.push_back( 2 );
   if ( c1.size( ) != 0 )
      cout << "New last element: " << c1.back( ) << endl;
}
```

Output

```
Last element: 1
New last element: 2
```

Requirements

Header: <deque>

See Also

Reference

deque Class

deque::push_back and deque::pop_back

Standard Template Library

Other Resources

deque Class Members

deque::push_front

Adds an element to the beginning of the deque.

```
void push_front(
   const Type& _Val
);
```

Parameters

_Val

The element added to the beginning of the deque.

Remarks

If an exception is thrown, the deque is left unaltered and the exception is rethrown.

Example

```
// deque_push_front.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_front( 1 );
   if ( c1.size( ) != 0 )
      cout << "First element: " << c1.front( ) << endl;

   c1.push_front( 2 );
   if ( c1.size( ) != 0 )
      cout << "New first element: " << c1.front( ) << endl;
}
```

Output

```
First element: 1
New first element: 2
```

Requirements

Header: <deque>

See Also

Reference

deque Class

deque::push_front and deque::pop_front

Standard Template Library

Other Resources

deque Class Members

deque::rbegin

Returns an iterator to the first element in a reversed deque.

```
const_reverse_iterator rbegin( ) const;
reverse_iterator rbegin( );
```

Return Value

A reverse random-access iterator addressing the first element in a reversed deque or addressing what had been the last element in the unreversed deque.

Remarks

rbegin is used with a reversed deque just as begin is used with a deque.

If the return value of **rbegin** is assigned to a **const_reverse_iterator**, the deque object cannot be modified. If the return value of **rbegin** is assigned to a **reverse_iterator**, the deque object can be modified.

rbegin can be used to iterate through a deque backwards.

Example

```
// deque_rbegin.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;
   deque <int>::iterator c1_Iter;
   deque <int>::reverse_iterator c1_rIter;

   // If the following line had replaced the line above, an error
   // would have resulted in the line modifying an element
   // (commented below) because the iterator would have been const
   // deque <int>::const_reverse_iterator c1_rIter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   c1_rIter = c1.rbegin( );
   cout << "Last element in the deque is " << *c1_rIter << "." << endl;

   cout << "The deque contains the elements: ";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << *c1_Iter << " ";
   cout << "in that order.";
   cout << endl;

   // rbegin can be used to iterate through a deque in reverse order
   cout << "The reversed deque is: ";
   for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
      cout << *c1_rIter << " ";
   cout << endl;

   c1_rIter = c1.rbegin( );
   *c1_rIter = 40;  // This would have caused an error if a
                    // const_reverse iterator had been declared as
                    // noted above
   cout << "Last element in deque is now " << *c1_rIter << "." << endl;
}
```

Output

```
Last element in the deque is 30.
The deque contains the elements: 10 20 30 in that order.
The reversed deque is: 30 20 10
Last element in deque is now 40.
```

Requirements

Header: <deque>

See Also

Reference

deque Class

deque::rbegin and deque::rend

Standard Template Library

Other Resources

deque Class Members

deque::rend

Returns an iterator that addresses the location succeeding the last element in a reversed deque.

```
const_reverse_iterator rend( ) const;
reverse_iterator rend( );
```

Return Value

A reverse random-access iterator that addresses the location succeeding the last element in a reversed deque (the location that had preceded the first element in the unreversed deque).

Remarks

rend is used with a reversed deque just as end is used with a deque.

If the return value of **rend** is assigned to a **const_reverse_iterator**, the deque object cannot be modified. If the return value of **rend** is assigned to a **reverse_iterator**, the deque object can be modified.

rend can be used to test whether a reverse iterator has reached the end of its deque.

The value returned by **rend** should not be dereferenced.

Example

```
// deque_rend.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;

   deque <int> c1;
   deque <int>::iterator c1_Iter;
   deque <int>::reverse_iterator c1_rIter;
   // If the following line had replaced the line above, an error
   // would have resulted in the line modifying an element
   // (commented below) because the iterator would have been const
   // deque <int>::const_reverse_iterator c1_rIter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   c1_rIter = c1.rend( );
   c1_rIter --; // Decrementing a reverse iterator moves it forward
                // in the deque (to point to the first element here)
   cout << "The first element in the deque is: " << *c1_rIter << endl;

   cout << "The deque is: ";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << *c1_Iter << " ";
   cout << endl;

   // rend can be used to test if an iteration is through all of
   // the elements of a reversed deque
   cout << "The reversed deque is: ";
   for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
      cout << *c1_rIter << " ";
   cout << endl;

   c1_rIter = c1.rend( );
   c1_rIter--; // Decrementing the reverse iterator moves it backward
               // in the reversed deque (to the last element here)
```

```
    *c1_rIter = 40; // This modification of the last element would
                    // have caused an error if a const_reverse
                    // iterator had been declared (as noted above)
    cout << "The modified reversed deque is: ";
    for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
       cout << *c1_rIter << " ";
    cout << endl;
  }
```

Output

```
The first element in the deque is: 10
The deque is: 10 20 30
The reversed deque is: 30 20 10
The modified reversed deque is: 30 20 40
```

Requirements

Header: <deque>

See Also

Reference

deque Class

deque::rbegin and deque::rend

Standard Template Library

Other Resources

deque Class Members

deque::resize

Specifies a new size for a deque.

```
void resize(
   size_type _Newsize
);
void resize(
   size_type _Newsize,
   Type _Val
);
```

Parameters

_Newsize

The new size of the deque.

_Val

The value of the new elements to be added to the deque if the new size is larger that the original size. If the value is omitted, the new elements are assigned the default value for the class.

Remarks

If the deque's size is less than the requested size, *_Newsize*, elements are added to the deque until it reaches the requested size.

If the deque's size is larger than the requested size, the elements closest to the end of the deque are deleted until the deque reaches the size *_Newsize*.

If the present size of the deque is the same as the requested size, no action is taken.

size reflects the current size of the deque.

Example

```
// deque_resize.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   c1.resize( 4,40 );
   cout << "The size of c1 is: " << c1.size( ) << endl;
   cout << "The value of the last element is " << c1.back( ) << endl;

   c1.resize( 5 );
   cout << "The size of c1 is now: " << c1.size( ) << endl;
   cout << "The value of the last element is now " << c1.back( ) << endl;

   c1.resize( 2 );
   cout << "The reduced size of c1 is: " << c1.size( ) << endl;
   cout << "The value of the last element is now " << c1.back( ) << endl;
}
```

Output

```
The size of c1 is: 4
The value of the last element is 40
```

```
The size of c1 is now: 5
The value of the last element is now 0
The reduced size of c1 is: 2
The value of the last element is now 20
```

Requirements

Header: <deque>

See Also

Reference

deque Class

deque::size and deque::resize

Standard Template Library

Other Resources

deque Class Members

deque::size

Returns the number of elements in the deque.

```
size_type size( ) const;
```

Return Value

The current length of the deque.

Example

```
// deque_size.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;
   deque <int>::size_type i;

   c1.push_back( 1 );
   i = c1.size( );
   cout << "The deque length is " << i << "." << endl;

   c1.push_back( 2 );
   i = c1.size( );
   cout << "The deque length is now " << i << "." << endl;
}
```

Output

```
The deque length is 1.
The deque length is now 2.
```

Requirements

Header: <deque>

See Also

Reference

deque Class

deque::size and deque::resize

Standard Template Library

Other Resources

deque Class Members

deque::swap

Exchanges the elements of two deques.

```
void swap(
   deque<Type, Allocator>& _Right
);
friend void swap(
   deque<Type, Allocator>& _Left,
   deque<Type, Allocator>& _Right
)
template<class Type, class Allocator>
void swap(
   deque< Type, Allocator>& _Left,
   deque< Type, Allocator>& _Right
);
```

Parameters

_Right

The deque providing the elements to be swapped, or the deque whose elements are to be exchanged with those of the deque *_Left*.

_Left

A deque whose elements are to be exchanged with those of the deque *_Right*.

Example

```
// deque_swap.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1, c2, c3;
   deque <int>::iterator c1_Iter;

   c1.push_back( 1 );
   c1.push_back( 2 );
   c1.push_back( 3 );
   c2.push_back( 10 );
   c2.push_back( 20 );
   c3.push_back( 100 );

   cout << "The original deque c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   c1.swap( c2 );

   cout << "After swapping with c2, deque c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   swap( c1,c3 );

   cout << "After swapping with c3, deque c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;
```

```
    swap<>(c1, c2);
    cout << "After swapping with c2, deque c1 is:";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
       cout << " " << *c1_Iter;
    cout << endl;
 }
```

Output

```
The original deque c1 is: 1 2 3
After swapping with c2, deque c1 is: 10 20
After swapping with c3, deque c1 is: 100
After swapping with c2, deque c1 is: 1 2 3
```

Requirements

Header: <deque>

See Also

Reference

deque Class

deque::assign and deque::swap

Standard Template Library

Other Resources

deque Class Members

deque Operators

For more information about the operators in deque, see deque Class Members.

See Also

Reference

Standard Template Library

deque::operator[]

Returns a reference to the deque element at a specified position.

```
reference operator[](
   size_type _Pos
);
const_reference operator[](
   size_type _Pos
) const;
```

Parameters

_Pos

The position of the deque element to be referenced.

Return Value

A reference to the element whose position is specified in the argument. If the position specified is greater than the size of the deque, the result is undefined.

Remarks

If the return value of **operator[]** is assigned to a **const_reference**, the deque object cannot be modified. If the return value of **operator[]** is assigned to a **reference**, the deque object can be modified.

When compiling with _SECURE_SCL 1, a runtime error will occur if you attempt to access an element outside the bounds of the deque. See Checked Iterators for more information.

Example

```
// deque_op_ref.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );
   cout << "The first integer of c1 is " << c1[0] << endl;
   int& i = c1[1];
   cout << "The second integer of c1 is " << i << endl;

}
```

Output

```
The first integer of c1 is 10
The second integer of c1 is 20
```

Requirements

Header: <deque>

See Also

Reference

deque Class

deque::operator[] and deque::at

Standard Template Library

Other Resources

deque Class Members

<exception> Header

Defines several types and functions related to the handling of exceptions. Exception handling is used in situations in which the system can recover from an error. It provides a means for control to be returned from a function to the program. The objective of incorporating exception handling is to increase the program's robustness while providing a way to recover from an error in an orderly fashion.

For a list of all members of this header, see <exception> Members.

```
#include <exception>
```

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<exception> Members

Standard C++ Library Header Files

<exception> Members

Reference

Typedefs

terminate_handler	A type that describes a pointer to a function suitable for use as a **terminate_handler**.
unexpected_handler	A type that describes a pointer to a function suitable for use as an **unexpected_handler**.

Functions

set_terminate	Establishes a new **terminate_handler** to be called at the termination of the program.
set_unexpected	Establishes a new **unexpected_handler** to be when an unexpected exception is encountered.
terminate	Calls a terminate handler.
uncaught_exception	Returns **true** only if a thrown exception is being currently processed.
unexpected	Calls an unexpected handler.

Classes

bad_exception Class	The class describes an exception that can be thrown from an **unexpected_handler**.
exception Class	The class serves as the base class for all exceptions thrown by certain expressions and by the Standard C++ Library.

See Also

Reference

<exception> Header

Thread Safety in the Standard C++ Library

<exception> Typedefs

For information about the typedefs in <exception>, see <exception> Members.

terminate_handler

The type describes a pointer to a function suitable for use as a **terminate_handler**.

```
typedef void ( *terminate_handler )( );
```

Remarks

The type describes a pointer to a function suitable for use as a terminate handler.

Example

See set_terminate for an example of the use of **terminate_handler**.

Requirements

Header: <exception>

See Also

Other Resources

<exception> Members

unexpected_handler

The type describes a pointer to a function suitable for use as an **unexpected_handler**.

```
typedef void (*unexpected_handler)( );
```

Example

See set_unexpected for an example of the use of **unexpected_handler**.

Requirements

Header: <exception>

See Also

Other Resources

<exception> Members

<exception> Functions

For information about the functions in <exception>, see <exception>.

set_terminate (<exception>)

Establishes a new **terminate_handler** to be called at the termination of the program.

```
terminate_handler
   set_terminate(
      terminate_handler _Pnew
   ) throw( );
```

Parameters

_Pnew

The function to be called at termination.

Return Value

The address of the previous function that used to be called at termination.

Remarks

The function establishes a new terminate_handler as the function **_Pnew*. Thus, *_Pnew* must not be a null pointer. The function returns the address of the previous terminate handler.

Example

```
// exception_set_terminate.cpp
// compile with: /EHsc /c
#include<exception>
#include<iostream>

using namespace std;

void termfunction( )
{
   cout << "I'll be back." << endl;
   abort( );
}

int main( )
{
   terminate_handler oldHand = set_terminate(termfunction);

   // Throwing an unhandled exception would also terminate the program
   throw bad_alloc( );

   // The program could also be explicitely terminated with:
   // terminate( );
}
```

Output

```
I'll be back.

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

Requirements

Header: <exception>

See Also

Other Resources

<exception> Members

set_unexpected (<exception>)

Establishes a new **unexpected_handler** to be when an unexpected exception is encountered.

```
unexpected_handler
   set_unexpected(
      unexpected_handler _Pnew
   ) throw( );
```

Parameters

_Pnew

The function to be called at

Return Value

The address of the previous **unexpected_handler**.

Remarks

_Pnew must not be a null pointer.

The C++ Standard requires that **unexpected** is called when a function throws an exception that is not on its throw list. The current implementation does not support this. The following example calls **unexpected** directly, which then calls the **unexpected_handler**.

Example

```
// exception_set_unexpected.cpp
// compile with: /c /EHsc
#include<exception>
#include<iostream>

using namespace std;

void unfunction( )
{
   cout << "I'll be back." << endl;
   terminate( );
}

int main( )
{
   unexpected_handler oldHand = set_unexpected( unfunction );
   unexpected( );
}
```

Output

```
I'll be back.

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

Requirements

Header: <exception>

See Also

Other Resources

<exception> Members

terminate (<exception>)

Calls a terminate handler.

```
void terminate( );
```

Remarks

The function calls a terminate handler, a function of type **void**. If **terminate** is called directly by the program, the terminate handler is the one most recently set by a call to set_terminate. If **terminate** is called for any of several other reasons during evaluation of a throw expression, the terminate handler is the one in effect immediately after evaluating the throw expression.

A terminate handler may not return to its caller. At program startup, the terminate handler is a function that calls **abort**.

Example

See set_unexpected for an example of the use of **terminate**.

Requirements

Header: <exception>

See Also

Other Resources

<exception> Members

uncaught_exception

Returns **true** only if a thrown exception is being currently processed.

```
bool uncaught_exception( );
```

Return Value

Returns **true** after completing evaluation of a throw expression and before completing initialization of the exception declaration in the matching handler or calling unexpected as a result of the throw expression. In particular, **uncaught_exception** will return **true** when called from a destructor that is being invoked during an exception unwind. On devices, **uncaught_exception** is only supported on Windows CE 5.00 and higher versions, including Windows Mobile 2005 platforms.

Example

```
// exception_uncaught_exception.cpp
// compile with: /EHsc
#include <exception>
#include <iostream>
#include <string>

class Test
{
public:
   Test( std::string msg ) : m_msg( msg )
   {
      std::cout << "In Test::Test(\"" << m_msg << "\")" << std::endl;
   }
   ~Test( )
   {
      std::cout << "In Test::~Test(\"" << m_msg << "\")" << std::endl
         << "        std::uncaught_exception( ) = "
         << std::uncaught_exception( )
         << std::endl;
   }
private:
    std::string m_msg;
};

// uncaught_exception will be true in the destructor
// for the object created inside the try block because
// the destructor is being called as part of the unwind.

int main( void )
   {
      Test t1( "outside try block" );
      try
      {
         Test t2( "inside try block" );
         throw 1;
      }
      catch (...) {
   }
}
```

Output

```
In Test::Test("outside try block")
In Test::Test("inside try block")
In Test::~Test("inside try block")
        std::uncaught_exception( ) = 1
In Test::~Test("outside try block")
        std::uncaught_exception( ) = 0
```

Requirements

Header: <exception>

See Also

Other Resources

<exception> Members

unexpected (<exception>)

Calls the unexpected handler.

```
void unexpected( );
```

Remarks

The C++ Standard requires that **unexpected** is called when a function throws an exception that is not on its throw list. The current implementation does not support this. The example calls **unexpected** directly, which calls the unexpected handler.

The function calls an unexpected handler, a function of type **void**. If **unexpected** is called directly by the program, the unexpected handler is the one most recently set by a call to set_unexpected.

An unexpected handler may not return to its caller. It may terminate execution by:

- Throwing an object of a type listed in the exception specification or an object of any type if the unexpected handler is called directly by the program.
- Throwing an object of type bad_exception.
- Calling terminate, **abort** or **exit**(**int**).

At program startup, the unexpected handler is a function that calls terminate.

Example

See set_unexpected for an example of the use of **unexpected.**

Requirements

Header: <exception>

See Also

Other Resources

<exception> Members

<exception> Classes

For information about the classes in <exception>, see <exception> Members

bad_exception Class

The class describes an exception that can be thrown from an unexpected handler.

```
class bad_exception
   : public exception {};
```

Remarks

unexpected will throw a **bad_exception** instead of terminating or instead of calling another function specified with set_unexpected if **bad_exception** is included in the throw list of a function.

The value returned by **what** is an implementation-defined C string. None of the member functions throw any exceptions.

For a list of members inherited by the **bad_exception** class, see exception Class.

Example

See set_unexpected for an example of the use of unexpected throwing a **bad_exception**.

Requirements

Header: <exception>

See Also

Reference

exception Class

Thread Safety in the Standard C++ Library

Other Resources

<exception> Members

exception Class

The class serves as the base class for all exceptions thrown by certain expressions and by the Standard C++ Library.

```
class exception {
public:
   exception( );
   exception(const char *const&);
   exception(const char *const&, int);
   exception(const exception&);
   exception& operator=(const exception&);
   virtual ~exception( );
   virtual const char *what( ) const;
};
```

Remarks

Specifically, this base class is the root of the standard exception classes defined in <stdexcept>. The C string value returned by **what** is left unspecified by the default constructor, but may be defined by the constructors for certain derived classes as an implementation-defined C string. None of the member functions throw any exceptions.

The int parameter allows you to specify that no memory should be allocated. The value of the int is ignored.

Example

For examples of the use of the standard exception classes that inherit from the **exception** class, see any of the classes defined in <stdexcept>.

Requirements

Header: <exception>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<exception> Members

<fstream>

Defines several classes that support iostreams operations on sequences stored in external files.

For a list of all members of this header, see <fstream> Members.

```
#include <fstream>
```

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

<fstream> Members
Standard C++ Library Header Files

<fstream> Members

Reference

Typedefs

filebuf	A type **basic_filebuf** specialized on **char** template parameters.
fstream	A type **basic_fstream** specialized on **char** template parameters.
ifstream	A type **basic_ifstream** specialized on **char** template parameters.
ofstream	A type **basic_ofstream** specialized on **char** template parameters.
wfstream	A type **basic_fstream** specialized on **wchar_t** template parameters.
wifstream	A type **basic_ifstream** specialized on **wchar_t** template parameters.
wofstream	A type **basic_ofstream** specialized on **wchar_t** template parameters.
wfilebuf	A type **basic_filebuf** specialized on **wchar_t** template parameters.

Classes

basic_filebuf	The template class describes a stream buffer that controls the transmission of elements of type **Elem**, whose c haracter traits are determined by the class **Tr**, to and from a sequence of elements stored in an external file.
basic_fstream	The template class describes an object that controls insertion and extraction of elements and encoded objects u sing a stream buffer of class basic_filebuf<**Elem**, **Tr**>, with elements of type **Elem**, whose character traits are d etermined by the class **Tr**.
basic_ifstream	The template class describes an object that controls extraction of elements and encoded objects from a stream buffer of class basic_filebuf<**Elem**, **Tr**>, with elements of type **Elem**, whose character traits are determined by the class **Tr**.
basic_ofstream	The template class describes an object that controls insertion of elements and encoded objects into a stream b uffer of class basic_filebuf<**Elem**, **Tr**>, with elements of type **Elem**, whose character traits are determined by t he class **Tr**.

See Also

Reference

<iostream>
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

<fstream> Typedefs

For information about the typedefs in <fstream>, see <fstream>.

See Also

Reference

iostream Programming

iostreams Conventions

filebuf

A type **basic_filebuf** specialized on **char** template parameters.

```
typedef basic_filebuf<char, char_traits<char> > filebuf;
```

Remarks

The type is a synonym for template class basic_filebuf, specialized for elements of type **char** with default character traits.

Requirements

Header: <fstream>

See Also

Reference

basic_filebuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_filebuf Members
<fstream> Members

fstream

A type **basic_fstream** specialized on **char** template parameters.

```
typedef basic_fstream<char, char_traits<char> > fstream;
```

Remarks

The type is a synonym for template class basic_fstream, specialized for elements of type **char** with default character traits.

Requirements

Header: <fstream>

See Also

Reference

basic_fstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_fstream Members
<fstream> Members

ifstream

A type **basic_ifstream** specialized on **char** template parameters.

```
typedef basic_ifstream<char, char_traits<char> > ifstream;
```

Remarks

The type is a synonym for template class basic_ifstream, specialized for elements of type **char** with default character traits.

Requirements

Header: <fstream>

See Also

Reference

basic_ifstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ifstream Members
<fstream> Members

ofstream

A type **basic_ofstream** specialized on **char** template parameters.

```
typedef basic_ofstream<char, char_traits<char> > ofstream;
```

Remarks

The type is a synonym for template class basic_ofstream, specialized for elements of type **char** with default character traits.

Requirements

Header: <fstream>

See Also

Reference

basic_ofstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ofstream Members
<fstream> Members

wfstream

A type **basic_fstream** specialized on **wchar_t** template parameters.

```
typedef basic_fstream<wchar_t, char_traits<wchar_t> > wfstream;
```

Remarks

The type is a synonym for template class basic_fstream, specialized for elements of type **wchar_t** with default character traits.

Requirements

Header: <fstream>

See Also

Reference

basic_fstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_fstream Members
<fstream> Members

wifstream

A type **basic_ifstream** specialized on **wchar_t** template parameters.

```
typedef basic_ifstream<wchar_t, char_traits<wchar_t> > wifstream;
```

Remarks

The type is a synonym for template class basic_ifstream, specialized for elements of type **wchar_t** with default character traits.

Requirements

Header: <fstream>

See Also

Reference

basic_ifstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ifstream Members
<fstream> Members

wofstream

A type **basic_ofstream** specialized on **wchar_t** template parameters.

```
typedef basic_ofstream<wchar_t, char_traits<wchar_t> > wofstream;
```

Remarks

The type is a synonym for template class basic_ofstream, specialized for elements of type **wchar_t** with default character traits.

Requirements

Header: <fstream>

See Also

Reference

basic_ofstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ofstream Members
<fstream> Members

wfilebuf

A type **basic_filebuf** specialized on **wchar_t** template parameters.

```
typedef basic_filebuf<wchar_t, char_traits<wchar_t> > wfilebuf;
```

Remarks

The type is a synonym for template class basic_filebuf, specialized for elements of type **wchar_t** with default character traits.

Requirements

Header: <fstream>

See Also

Reference

basic_filebuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_filebuf Members
<fstream> Members

<fstream> Classes

For information about the classes in <fstream>, see <fstream> Members

See Also

Reference

iostream Programming

iostreams Conventions

basic_filebuf Class

Describes a stream buffer that controls the transmission of elements of type *Elem*, whose character traits are determined by the class *Tr*, to and from a sequence of elements stored in an external file.

For a list of all members of this type, see basic_filebuf Members.

```
template <class Elem, class Tr = char_traits<Elem> >
    class basic_filebuf : public basic_streambuf<Elem, Tr>
```

Parameters

Elem

The basic element of the file buffer.

Tr

The traits of the basic element of the file buffer (usually **char_traits**<*Elem*>).

Remarks

The template class describes a stream buffer that controls the transmission of elements of type *Elem*, whose character traits are determined by the class *Tr*, to and from a sequence of elements stored in an external file.

Note

Objects of type **basic_filebuf** are created with an internal buffer of type **char *** regardless of the **char_type** specified by the type parameter *Elem*. This means that a Unicode string (containing **wchar_t** characters) will be converted to an ANSI string (containing **char** characters) before it is written to the internal buffer. To store Unicode strings in the buffer, create a new buffer of type **wchar_t** and set it using the basic_streambuf::pubsetbuf**()** method. To see an example that demonstrates this behavior, see below.

An object of class **basic_filebuf**<*Elem*, *Tr*> stores a file pointer, which designates the **FILE** object that controls the stream associated with an open file. It also stores pointers to two file conversion facets for use by the protected member functions overflow and underflow. For more information, see basic_filebuf::open.

Example

The following example demonstrates how to force an object of type **basic_filebuf<wchar_t>** to store Unicode characters in its internal buffer by calling the **pubsetbuf()** method.

```
// unicode_basic_filebuf.cpp
// compile with: /EHsc

#include <iostream>
#include <string>
#include <fstream>
#include <iomanip>
#include <memory.h>
#include <string.h>

#define IBUFSIZE 16

using namespace std;

void hexdump(const string& filename);

int main()
{
    wchar_t* wszHello = L"Hello World";
    wchar_t wBuffer[128];

    basic_filebuf<wchar_t> wOutFile;
```

```
    // Open a file, wcHello.txt, then write to it, then dump the
    // file's contents in hex
    wOutFile.open("wcHello.txt",
        ios_base::out | ios_base::trunc | ios_base::binary);
    if(!wOutFile.is_open())
    {
        cout << "Error Opening wcHello.txt\n";
        return -1;
    }
    wOutFile.sputn(wszHello, (streamsize)wcslen(wszHello));
    wOutFile.close();
    cout << "Hex Dump of wcHello.txt - note that output is ANSI chars:\n";
    hexdump(string("wcHello.txt"));

    // Open a file, wwHello.txt, then set the internal buffer of
    // the basic_filebuf object to be of type wchar_t, then write
    // to the file and dump the file's contents in hex
    wOutFile.open("wwHello.txt",
        ios_base::out | ios_base::trunc | ios_base::binary);
    if(!wOutFile.is_open())
    {
        cout << "Error Opening wwHello.txt\n";
        return -1;
    }
    wOutFile.pubsetbuf(wBuffer, (streamsize)128);
    wOutFile.sputn(wszHello, (streamsize)wcslen(wszHello));
    wOutFile.close();
    cout << "\nHex Dump of wwHello.txt - note that output is wchar_t chars:\n";
    hexdump(string("wwHello.txt"));

    return 0;
}

// dump contents of filename to stdout in hex
void hexdump(const string& filename)
{
    fstream ifile(filename.c_str(),
        ios_base::in | ios_base::binary);
    char *ibuff = new char[IBUFSIZE];
    char *obuff = new char[(IBUFSIZE*2)+1];
    int i;

    if(!ifile.is_open())
    {
        cout << "Cannot Open " << filename.c_str()
             << " for reading\n";
        return;
    }
    if(!ibuff || !obuff)
    {
        cout << "Cannot Allocate buffers\n";
        ifile.close();
        return;
    }

    while(!ifile.eof())
    {
        memset(obuff,0,(IBUFSIZE*2)+1);
        memset(ibuff,0,IBUFSIZE);
        ifile.read(ibuff,IBUFSIZE);

        // corner case where file is exactly a multiple of
        // 16 bytes in length
        if(ibuff[0] == 0 && ifile.eof())
            break;

        for(i = 0; i < IBUFSIZE; i++)
        {
```

```
            if(ibuff[i] >= ' ')
                obuff[i] = ibuff[i];
            else
                obuff[i] = '.';

            cout << setfill('0') << setw(2) << hex
                 << (int)ibuff[i] << ' ';
        }
        cout << "  " << obuff << endl;
    }
    ifile.close();
}
```

Output

```
Hex Dump of wcHello.txt - note that output is ANSI chars:
48 65 6c 6c 6f 20 57 6f 72 6c 64 00 00 00 00 00   Hello World.....

Hex Dump of wwHello.txt - note that output is wchar_t chars:
48 00 65 00 6c 00 6c 00 6f 00 20 00 57 00 6f 00   H.e.l.l.o. .W.o.
72 00 6c 00 64 00 00 00 00 00 00 00 00 00 00 00   r.l.d...........
```

Requirements

Header: <fstream>

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

<fstream> Members
basic_filebuf Members

basic_filebuf Members

The following tables list the members of the basic_filebuf Class. For a list of members inherited by the **basic_filebuf** class, see basic_streambuf Members.

Reference

Constructors

basic_filebuf	Constructs an object of type **basic_filebuf**.

Typedefs

char_type	Associates a type name with the *Elem* template parameter.
int_type	Makes this type within **basic_filebuf**'s scope equivalent to the type of the same name in the *Tr* scope.
off_type	Makes this type within **basic_filebuf**'s scope equivalent to the type of the same name in the *Tr* scope.
pos_type	Makes this type within **basic_filebuf**'s scope equivalent to the type of the same name in the *Tr* scope.
traits_type	Associates a type name with the *Tr* template parameter.

Member Functions

close	Closes a file.
is_open	Indicates whether a file is open.
open	Opens a file.
overflow	A protected virtual function that can be called when a new character is inserted into a full buffer.
pbackfail	The protected virtual member function tries to put back an element into the input stream, then make it the current e lement (pointed to by the next pointer).
seekoff	The protected virtual member function tries to alter the current positions for the controlled streams.
seekpos	The protected virtual member function tries to alter the current positions for the controlled streams.
setbuf	The protected virtual member function performs an operation particular to each derived stream buffer.
sync	Protected, virtual function tries to synchronize the controlled streams with any associated external streams.
uflow	Protected, virtual function to extract the current element from the input stream.
underflow	Protected, virtual function to extract the current element from the input stream.

See Also

Reference

basic_filebuf Class
basic_streambuf Class
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_filebuf Typedefs

For information about the typedefs in the basic_filebuf class, see basic_filebuf Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_filebuf::char_type

Associates a type name with the **Elem** template parameter.

```
typedef Elem char_type;
```

Requirements

Header: <fstream>

See Also

Reference

basic_filebuf Class

iostream Programming

iostreams Conventions

Other Resources

basic_filebuf Members

basic_filebuf::int_type

Makes this type within basic_filebuf's scope equivalent to the type of the same name in the **Tr** scope.

```
typedef typename traits_type::int_type int_type;
```

Requirements

Header: <fstream>

See Also

Reference

basic_filebuf Class

iostream Programming

iostreams Conventions

Other Resources

basic_filebuf Members

basic_filebuf::off_type

Makes this type within basic_filebuf's scope equivalent to the type of the same name in the **Tr** scope.

```
typedef typename traits_type::off_type off_type;
```

Requirements

Header: <fstream>

See Also

Reference

basic_filebuf Class

iostream Programming

iostreams Conventions

Other Resources

basic_filebuf Members

basic_filebuf::pos_type

Makes this type within basic_filebuf's scope equivalent to the type of the same name in the **Tr** scope.

```
typedef typename traits_type::pos_type pos_type;
```

Requirements

Header: <fstream>

See Also

Reference

basic_filebuf Class

iostream Programming

iostreams Conventions

Other Resources

basic_filebuf Members

basic_filebuf::traits_type

Associates a type name with the **Tr** template parameter.

```
typedef Tr traits_type;
```

Requirements

Header: <fstream>

See Also

Reference

basic_filebuf Class

iostream Programming

iostreams Conventions

Other Resources

basic_filebuf Members

basic_filebuf Member Functions

For information about the member functions in the gslice class, see basic_filebuf Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_filebuf::basic_filebuf

Constructs an object of type **basic_filebuf**.

```
basic_filebuf();
```

Remarks

The constructor stores a null pointer in all the pointers controlling the input buffer and the output buffer. It also stores a null pointer in the file pointer.

Requirements

Header: <fstream>

See Also

Reference

basic_filebuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_filebuf Members

basic_filebuf::close

Closes a file.

```
basic_filebuf *close( );
```

Return Value

The member function returns a null pointer if the file pointer is a null pointer.

Remarks

close calls **fclose**(**fp**). If that function returns a nonzero value, the function returns a null pointer. Otherwise, it returns **this** to indicate that the file was successfully closed.

For a wide stream, if any insertions have occurred since the stream was opened, or since the last call to **streampos**, the function calls overflow. It also inserts any sequence needed to restore the initial conversion state, by using the file conversion facet **fac** to call **fac.unshift** as needed. Each element **byte** of type **char** thus produced is written to the associated stream designated by the file pointer **fp** as if by successive calls of the form **fputc**(**byte**, **fp**). If the call to **fac.unshift** or any write fails, the function does not succeed.

Example

The following sample assumes two files in the current directory: basic_filebuf_close.txt (contents is "testing") and iotest.txt (contents is "ssss").

```
// basic_filebuf_close.cpp
// compile with: /EHsc
#include <fstream>
#include <iostream>

int main() {
   using namespace std;
   ifstream file;
   basic_ifstream <wchar_t> wfile;
   char c;
   // Open and close with a basic_filebuf
   file.rdbuf()->open( "basic_filebuf_close.txt", ios::in );
   file >> c;
   cout << c << endl;
   file.rdbuf( )->close( );

   // Open/close directly
   file.open( "iotest.txt" );
   file >> c;
   cout << c << endl;
   file.close( );

   // open a file with a wide character name
   wfile.open( L"iotest.txt" );

   // Open and close a nonexistent with a basic_filebuf
   file.rdbuf()->open( "ziotest.txt", ios::in );
   cout << file.fail() << endl;
   file.rdbuf( )->close( );

   // Open/close directly
   file.open( "ziotest.txt" );
   cout << file.fail() << endl;
   file.close( );
}
```

Output

t
s
0
1

Requirements

Header: <fstream>

See Also

Reference

basic_filebuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_filebuf Members

basic_filebuf::is_open

Indicates whether a file is open.

```
bool is_open( );
```

Return Value

true if the file pointer is not a null pointer.

Example

```
// basic_filebuf_is_open.cpp
// compile with: /EHsc
#include <fstream>
#include <iostream>

int main( )
{
   using namespace std;
   ifstream file;
   cout << boolalpha << file.rdbuf( )->is_open( ) << endl;

   file.open( "basic_filebuf_is_open.cpp" );
   cout << file.rdbuf( )->is_open( ) << endl;
}
```

Output

```
false
true
```

Requirements

Header: <fstream>

See Also

Reference

basic_filebuf Class

iostream Programming

iostreams Conventions

Other Resources

basic_filebuf Members

basic_filebuf::open

Opens a file.

```
basic_filebuf *open(
    const char *_Filename,
    ios_base::openmode _Mode,
    int _Prot = (int)ios_base::_Openprot
);
basic_filebuf *open(
    const char *_Filename,
    ios_base::openmode _Mode
);
basic_filebuf *open(
    const wchar_t *_Filename,
    ios_base::openmode _Mode,
    int _Prot = (int)ios_base::_Openprot
);
basic_filebuf *open(
    const wchar_t *_Filename,
    ios_base::openmode _Mode
);
```

Parameters

_Filename

The name of the file to open.

_Mode

One of the enumerations in ios_base::openmode.

_Prot

The default file opening protection.

Return Value

If the file pointer is a null pointer, the function returns a null pointer. Otherwise, it returns **this**.

Remarks

The member function opens the file with filename *filename*, by calling fopen(*filename*, **strmode**). **strmode** is determined from **mode &**~(ate & | binary):

- **ios_base::in** becomes **"r"** (open existing file for reading).
- ios_base::out or **ios_base::out | ios_base::trunc** becomes **"w"** (truncate existing file or create for writing).
- **ios_base::out | app** becomes **"a"** (open existing file for appending all writes).
- **ios_base::in | ios_base::out** becomes **"r+"** (open existing file for reading and writing).
- **ios_base::in | ios_base::out | ios_base::trunc** becomes **"w+"** (truncate existing file or create for reading and writing).
- **ios_base::in | ios_base::out | ios_base::app** becomes **"a+"** (open existing file for reading and for appending all writes).

If **mode & ios_base::binary** is nonzero, the function appends **b** to **strmode** to open a binary stream instead of a text stream. It then stores the value returned by **fopen** in the file pointer **fp**. If **mode & ios_base::ate** is nonzero and the file pointer is not a null pointer, the function calls **fseek**(**fp**, 0, **SEEK_END**) to position the stream at end of file. If that positioning operation fails, the function calls close(**fp**) and stores a null pointer in the file pointer.

If the file pointer is not a null pointer, the function determines the file conversion facet: **use_facet**<**codecvt**<**Elem**, **char**, **traits_type::**state_type> >(getloc), for use by underflow and overflow.

If the file pointer is a null pointer, the function returns a null pointer. Otherwise, it returns **this**.

Example

See basic_filebuf::close for an example that uses **open**.

Requirements

Header: <fstream>

See Also

Reference

basic_filebuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_filebuf Members

basic_filebuf::overflow

Called when a new character is inserted into a full buffer.

```
virtual int_type overflow(
   int_type _Meta = traits_type::eof
);
```

Parameters

_Meta

The character to insert into the buffer or **traits_type::eof**.

Return Value

If the function cannot succeed, it returns **traits_type::eof**. Otherwise, it returns **traits_type::**not_eof(*_Meta*).

Remarks

If *_Meta* **!= traits_type::**eof, the protected virtual member function endeavors to insert the element **ch = traits_type::**to_char_type(*_Meta*) into the output buffer. It can do so in various ways:

- If a write position is available, it can store the element into the write position and increment the next pointer for the output buffer.
- It can make a write position available by allocating new or additional storage for the output buffer.
- It can convert any pending output in the output buffer, followed by **ch**, by using the file conversion facet **fac** to call **fac.out** as needed. Each element `ch` of type *char* thus produced is written to the associated stream designated by the file pointer **fp** as if by successive calls of the form **fputc**(**ch**, **fp**). If any conversion or write fails, the function does not succeed.

Requirements

Header: <fstream>

See Also

Reference

basic_filebuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_filebuf Members

basic_filebuf::pbackfail

Tries to put back an element into the input stream, then make it the current element (pointed to by the next pointer).

```
virtual int_type pbackfail(
      int_type _Meta = traits_type::eof
);
```

Parameters

_Meta

The character to insert into the buffer, or **traits_type::eof**.

Return Value

If the function cannot succeed, it returns **traits_type::eof**. Otherwise, it returns **traits_type::**not_eof(*_Meta*).

Remarks

The protected virtual member function puts back an element into the input buffer and then makes it the current element (pointed to by the next pointer). If *_Meta* **== traits_type::**eof, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by **ch = traits_type::**to_char_type(*_Meta*). The function can put back an element in various ways:

- If a putback position is available, and the element stored there compares equal to **ch**, it can decrement the next pointer for the input buffer.
- If the function can make a **putback** position available, it can do so, set the next pointer to point at that position, and store **ch** in that position.
- If the function can push back an element onto the input stream, it can do so, such as by calling **ungetc** for an element of type **char**.

Requirements

Header: <fstream>

See Also

Reference

basic_filebuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_filebuf Members

basic_filebuf::seekoff

Tries to alter the current positions for the controlled streams.

```
virtual pos_type seekoff(
   off_type _Off,
   ios_base::seekdir _Way,
   ios_base::openmode _Which = ios_base::in | ios_base::out
);
```

Parameters

_Off

The position to seek for relative to *_Way*.

_Way

The starting point for offset operations. See seekdir for possible values.

_Which

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

Returns the new position or an invalid stream position.

Remarks

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class basic_filebuf<*Elem*, *Tr*>, a stream position can be represented by an object of type **fpos_t**, which stores an offset and any state information needed to parse a wide stream. Offset zero designates the first element of the stream. (An object of type pos_type stores at least an **fpos_t** object.)

For a file opened for both reading and writing, both the input and output streams are positioned in tandem. To switch between inserting and extracting, you must call either pubseekoff or pubseekpos. Calls to **pubseekoff** (and hence to **seekoff**) have various limitations for text streams, binary streams, and wide streams.

If the file pointer **fp** is a null pointer, the function fails. Otherwise, it endeavors to alter the stream position by calling **fseek**(**fp**, *_Off*, *_Way*). If that function succeeds and the resulting position **fposn** can be determined by calling **fgetpos**(**fp**, **&fposn**), the function succeeds. If the function succeeds, it returns a value of type **pos_type** containing **fposn**. Otherwise, it returns an invalid stream position.

Requirements

Header: <fstream>

See Also

Reference

basic_filebuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_filebuf Members

basic_filebuf::seekpos

Tries to alter the current positions for the controlled streams.

```
virtual pos_type seekpos(
   pos_type _Sp,
   ios_base::openmode _Which = ios_base::in | ios_base::out
);
```

Parameters

_Sp

The position to seek for.

_Which

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

If the file pointer **fp** is a null pointer, the function fails. Otherwise, it endeavors to alter the stream position by calling **fsetpos**(**fp**, **&fposn**), where **fposn** is the **fpos_t** object stored in **pos**. If that function succeeds, the function returns **pos**. Otherwise, it returns an invalid stream position. To determine if the stream position is invalid, compare the return value with `pos_type(off_type(-1))`.

Remarks

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class basic_filebuf<**Elem**, **Tr**>, a stream position can be represented by an object of type **fpos_t**, which stores an offset and any state information needed to parse a wide stream. Offset zero designates the first element of the stream. (An object of type **pos_type** stores at least an **fpos_t** object.)

For a file opened for both reading and writing, both the input and output streams are positioned in tandem. To switch between inserting and extracting, you must call either pubseekoff or pubseekpos. Calls to **pubseekoff** (and hence to **seekoff**) have various limitations for text streams, binary streams, and wide streams.

For a wide stream, if any insertions have occurred since the stream was opened, or since the last call to **streampos**, the function calls overflow. It also inserts any sequence needed to restore the initial conversion state, by using the file conversion facet **fac** to call **fac.unshift** as needed. Each element **byte** of type **char** thus produced is written to the associated stream designated by the file pointer **fp** as if by successive calls of the form **fputc**(**byte**, **fp**). If the call to **fac.unshift** or any write fails, the function does not succeed.

Requirements

Header: <fstream>

See Also

Reference

basic_filebuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_filebuf Members

basic_filebuf::setbuf

Performs an operation particular to each derived stream buffer.

```
virtual basic_streambuf *setbuf(
   char_type *_Buffer,
   streamsize _Count
);
```

Parameters

_Buffer

Pointer to a buffer.

_Count

Size of the buffer.

Return Value

The protected member function returns zero if the file pointer `fp` is a null pointer.

Remarks

setbuf calls **setvbuf**(**fp**, (**char** *)*_Buffer*, **_IOFBF**, *_Count* * **sizeof** (**Elem**)) to offer the array of *_Count* elements beginning at *_Buffer* as a buffer for the stream. If that function returns a nonzero value, the function returns a null pointer. Otherwise, it returns **this** to signal success.

Requirements

Header: <fstream>

See Also

Reference

basic_filebuf Class

iostream Programming

iostreams Conventions

Other Resources

basic_filebuf Members

basic_filebuf::sync

Tries to synchronize the controlled streams with any associated external streams.

```
int sync( );
```

Return Value

Returns zero if the file pointer **fp** is a null pointer. Otherwise, it returns zero only if calls to both overflow and **fflush**(**fp**) succeed in flushing any pending output to the stream.

Requirements

Header: <fstream>

See Also

Reference

basic_filebuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_filebuf Members

basic_filebuf::underflow

Extracts the current element from the input stream.

```
virtual int_type underflow( );
```

Return Value

If the function cannot succeed, it returns **traits_type::**eof. Otherwise, it returns **ch**, converted as described in the Remarks section.

Remarks

The protected virtual member function endeavors to extract the current element **ch** from the input stream, and return the element as **traits_type::**to_int_type(**ch**). It can do so in various ways:

- If a read position is available, it takes **ch** as the element stored in the read position and advances the next pointer for the input buffer.
- It can read one or more elements of type **char**, as if by successive calls of the form **fgetc**(**fp**), and convert them to an element **ch** of type **Elem** by using the file conversion facet fac to call **fac.in** as needed. If any read or conversion fails, the function does not succeed.

Requirements

Header: <fstream>

See Also

Reference

basic_filebuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_filebuf Members

basic_fstream Class

Describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class basic_filebuf<*Elem*, *Tr*>, with elements of type *Elem*, whose character traits are determined by the class *Tr*.

For a list of all members of this type, see basic_fstream Members.

```
template <class Elem, class Tr = char_traits<Elem> >
    class basic_fstream : public basic_iostream<Elem, Tr>
```

Parameters

Elem

The basic element of the file buffer.

Tr

The traits of the basic element of the file buffer (usually **char_traits**<*Elem*>).

Remarks

The object stores an object of class **basic_filebuf**<*Elem*, *Tr*>.

Note

The get pointer and put pointer of an fstream object are **NOT** independent of each other. If the get pointer moves, so does th e put pointer.

Example

The following example demonstrates how to create a **basic_fstream** object that can be read from and written to.

```
// basic_fstream_class.cpp
// compile with: /EHsc

#include <fstream>
#include <iostream>

using namespace std;

int main(int argc, char **argv)
{
    fstream fs("C:\\fstream.txt", ios::in | ios::out | ios::trunc);
    if (!fs.bad())
    {
        // Write to the file.
        fs << "Writing to a basic_fstream object..." << endl;
        fs.close();

        // Dump the contents of the file to cout.
        fs.open("C:\\fstream.txt", ios::in);
        cout << fs.rdbuf();
        fs.close();
    }
}
```

Output

```
Writing to a basic_fstream object...
```

Requirements

Header: <fstream>

See Also

Reference

Thread Safety in the Standard C++ Library

iostream Programming

iostreams Conventions

Other Resources

<fstream> Members

basic_fstream Members

basic_fstream Members

The following table lists the members of the basic_fstream Class. For a list of members inherited by the **basic_fstream** class, see basic_iostream Class.

Reference

Constructors

basic_fstream	Constructs an object of type **basic_fstream**.

Member Functions

close	Closes a file.
is_open	Determines if a file is open.
open	Opens a file.
rdbuf	Returns the address of the stored stream buffer, of type pointer to basic_filebuf<*Elem*, *Tr*>.

See Also

Reference

basic_fstream Class
basic_iostream Class
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

basic_fstream Member Functions

For information about the member functions in the basic_fstream class, see basic_fstream Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_fstream::basic_fstream

Constructs an object of type **basic_fstream**.

```
basic_fstream( );
explicit basic_fstream(
    const char *_Filename,
    ios_base::openmode _Mode = ios_base::in | ios_base::out,
    int _Prot = (int)ios_base::_Openprot
);
explicit basic_fstream(
    const wchar_t *_Filename,
    ios_base::openmode _Mode = ios_base::in | ios_base::out,
    int _Prot = (int)ios_base::_Openprot
);
```

Parameters

_Filename

The name of the file to open.

_Mode

One of the enumerations in ios_base::openmode.

_Prot

The default file opening protection.

Remarks

The first constructor initializes the base class by calling basic_iostream(**sb**), where **sb** is the stored object of class basic_filebuf<**Elem**, **Tr**>. It also initializes **sb** by calling **basic_filebuf**<**Elem**, **Tr**>.

The second and third constructors initializes the base class by calling **basic_iostream**(**sb**). It also initializes **sb** by calling **basic_filebuf**<**Elem**, **Tr**>, and then **sb.**open(*_Filename*, *_Mode*). If the latter function returns a null pointer, the constructor calls setstate(**failbit**).

Example

See streampos for an example that uses **basic_fstream**.

Requirements

Header: <fstream>

See Also

Reference

basic_fstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_fstream Members

basic_fstream::close

Closes a file.

```
void close( );
```

Remarks

The member function calls rdbuf **->** close.

Example

See basic_filebuf::close for an example of how to use **close**.

Requirements

Header: <fstream>

See Also

Reference

basic_fstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_fstream Members

basic_fstream::is_open

Determines if a file is open.

```
bool is_open( );
```

Return Value

true if the file is open, **false** otherwise.

Remarks

The member function returns rdbuf **->**is_open.

Example

See basic_filebuf::is_open for an example of how to use **is_open**.

Requirements

Header: <fstream>

See Also

Reference

basic_fstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_fstream Members

basic_fstream::open

Opens a file.

```
void open(
    const char *_Filename,
    ios_base::openmode _Mode = ios_base::in | ios_base::out,
    int _Prot = (int)ios_base::_Openprot
);
void open(
    const char *_Filename,
    ios_base::openmode _Mode
);
void open(
    const wchar_t *_Filename,
    ios_base::openmode _Mode = ios_base::in | ios_base::out,
    int _Prot = (int)ios_base::_Openprot

);
void open(
    const wchar_t *_Filename,
    ios_base::openmode _Mode
);
```

Parameters

_Filename

The name of the file to open.

_Mode

One of the enumerations in ios_base::openmode.

_Prot

The default file opening protection.

Remarks

The member function calls rdbuf **->** open(*_Filename*, *_Mode*). If that function returns a null pointer, the function calls setstate(**failbit**).

Example

See basic_filebuf::open for an example of how to use **open**.

Requirements

Header: <fstream>

See Also

Reference

basic_fstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_fstream Members

basic_fstream::rdbuf

Returns the address of the stored stream buffer, of type pointer to basic_filebuf<**Elem**, **Tr**>.

```
basic_filebuf<Elem, Tr> *rdbuf( ) const
```

Return Value

The address of the stored stream buffer.

Example

See basic_filebuf::close for an example of how to use **rdbuf**.

Requirements

Header: <fstream>

See Also

Reference

basic_fstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_fstream Members

basic_ifstream Class

Describes an object that controls extraction of elements and encoded objects from a stream buffer of class basic_filebuf<*Elem*, *Tr*>, with elements of type *Elem*, whose character traits are determined by the class *Tr*.

For a list of all members of this type, see basic_ifstream Members.

```
template <class Elem, class Tr = char_traits<Elem> >
    class basic_ifstream : public basic_istream<Elem, Tr>
```

Parameters

Elem

The basic element of the file buffer.

Tr

The traits of the basic element of the file buffer (usually **char_traits**<*Elem*>).

Remarks

The object stores an object of class **basic_filebuf**<*Elem*, *Tr*>.

Example

The following example shows how to read in text from a file. To create the file, see the example for basic_ofstream Class.

```
// basic_ifstream_class.cpp
// compile with: /EHsc

#include <fstream>
#include <iostream>

using namespace std;

int main(int argc, char **argv)
{
    ifstream ifs("C:\\ofstream.txt");
    if (!ifs.bad())
    {
        // Dump the contents of the file to cout.
        cout << ifs.rdbuf();
        ifs.close();
    }
}
```

Output

```
Writing to a basic_ofstream object...
```

Requirements

Header: <fstream>

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

<fstream> Members
basic_ifstream Members

basic_ifstream Members

The following table lists the members of the basic_ifstream Class. For a list of members inherited by the **basic_ifstream** class, see basic_istream Members.

Reference

Constructors

basic_ifstream	Initializes a new instance of a **basic_ifstream** object.

Member Functions

close	Closes a file.
is_open	Determines if a file is open.
open	Opens a file.
rdbuf	Returns the address of the stored stream buffer.

See Also

Reference

basic_ifstream Class
basic_istream Class
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

basic_istream Members

basic_ifstream Member Functions

For information about the member functions in the basic_ifstream class, see basic_ifstream Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_ifstream::basic_ifstream

Constructs an object of type **basic_ifstream**.

```
basic_ifstream( );
explicit basic_ifstream(
    const char *_Filename,
    ios_base::openmode _Mode = ios_base::in,
    int _Prot = (int)ios_base::_Openprot
);
explicit basic_ifstream(
    const wchar_t *_Filename,
    ios_base::openmode _Mode = ios_base::in,
    int _Prot = (int)ios_base::_Openprot
);
```

Parameters

_Filename

The name of the file to open.

_Mode

One of the enumerations in ios_base::openmode.

_Prot

The default file opening protection.

Remarks

The first constructor initializes the base class by calling basic_istream(**sb**), where **sb** is the stored object of class basic_filebuf<*Elem*, *Tr*>. It also initializes **sb** by calling **basic_filebuf**<*Elem*, *Tr*>.

The second and third constructors initializes the base class by calling **basic_istream**(**sb**). It also initializes **sb** by calling basic_filebuf<*Elem*, *Tr*>, then **sb**.open(*_Filename*, *_Mode* | **ios_base::in**). If the latter function returns a null pointer, the constructor calls setstate(**failbit**).

Example

The following example shows how to read in text from a file. To create the file, see the example for basic_ofstream::basic_ofstream.

```
// basic_ifstream_ctor.cpp
// compile with: /EHsc

#include <fstream>
#include <iostream>

using namespace std;

int main(int argc, char **argv)
{
    ifstream ifs("C:\\ofstream.txt");
    if (!ifs.bad())
    {
        // Dump the contents of the file to cout.
        cout << ifs.rdbuf();
        ifs.close();
    }
}
```

Output

```
Writing to a basic_ofstream object...
```

Requirements

Header: <fstream>

See Also

Reference

basic_ifstream Class

iostream Programming

iostreams Conventions

Other Resources

basic_ifstream Members

basic_ifstream::close

Closes a file.

```
void close( );
```

Remarks

The member function calls rdbuf **->** close.

Example

See basic_filebuf::close for an example that uses **close**.

Requirements

Header: <fstream>

See Also

Reference

basic_ifstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ifstream Members

basic_ifstream::is_open

Determines if a file is open.

```
bool is_open( );
```

Return Value

true if the file is open, **false** otherwise.

Remarks

The member function returns rdbuf **->** is_open.

Example

See basic_filebuf::is_open for an example that uses **is_open**.

Requirements

Header: <fstream>

See Also

Reference

basic_ifstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ifstream Members

basic_ifstream::open

Opens a file.

```
void open(
    const char *_Filename,
    ios_base::openmode _Mode = ios_base::in,
    int _Prot = (int)ios_base::_Openprot
);
void open(
    const char *_Filename,
    ios_base::openmode _Mode
);
void open(
    const wchar_t *_Filename,
    ios_base::openmode _Mode = ios_base::in,
    int _Prot = (int)ios_base::_Openprot
);
void open(
    const wchar_t *_Filename,
    ios_base::openmode _Mode
);
```

Parameters

_Filename

The name of the file to open.

_Mode

One of the enumerations in ios_base::openmode.

_Prot

The default file opening protection.

Remarks

The member function calls rdbuf **->** open(*_Filename*, *_Mode* | **ios_base::in**). If open fails, the function calls setstate(**failbit**), which may throw an ios_base::failure exception.

Example

See basic_filebuf::open for an example that uses **open**.

Requirements

Header: <fstream>

See Also

Reference

basic_ifstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ifstream Members

basic_ifstream::rdbuf

Returns the address of the stored stream buffer.

```
basic_filebuf<Elem, Tr> *rdbuf( ) const
```

Return Value

A pointer to a basic_filebuf object representing the stored stream buffer.

Example

See basic_filebuf::close for an example that uses **rdbuf**.

Requirements

Header: <fstream>

See Also

Reference

basic_ifstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ifstream Members

basic_ofstream Class

Describes an object that controls insertion of elements and encoded objects into a stream buffer of class basic_filebuf<*Elem*, *Tr*>, with elements of type *Elem*, whose character traits are determined by the class *Tr*.

For a list of all members of this type, see basic_ofstream Members.

```
template <class Elem, class Tr = char_traits<Elem> >
    class basic_ofstream : public basic_ostream<Elem, Tr>
```

Parameters

Elem

The basic element of the file buffer.

Tr

The traits of the basic element of the file buffer (usually **char_traits**<*Elem*>).

Remarks

When the **wchar_t** specialization of **basic_ofstream** writes to the file, if the file is opened in text mode it will write a MBCS sequence. The internal representation will use a buffer of **wchar_t** characters.

The object stores an object of class **basic_filebuf**<*Elem*, *Tr*>.

Example

The following example shows how to create a **basic_ofstream** object and write text to it.

```
// basic_ofstream_class.cpp
// compile with: /EHsc
#include <fstream>

using namespace std;

int main(int argc, char **argv)
{
    ofstream ofs("C:\\ofstream.txt");
    if (!ofs.bad())
    {
        ofs << "Writing to a basic_ofstream object..." << endl;
        ofs.close();
    }
}
```

Requirements

Header: <fstream>

See Also

Reference

basic_ostream Class
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

<fstream> Members
basic_ofstream Members

basic_ofstream Members

The following table lists the members of the basic_ofstream Class. For a list of members inherited by the **basic_ofstream** class, see basic_ostream Members.

Reference

Constructors

basic_ofstream	Creates an object of type **basic_ofstream**.

Member Functions

close	Closes a file.
is_open	Determines if a file is open.
open	Opens a file.
rdbuf	Returns the address of the stored stream buffer.

See Also

Reference

basic_ofstream Class
basic_ostream Class
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

basic_ostream Members

basic_ofstream Member Functions

For information about the member functions in the basic_ofstream class, see basic_ofstream Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_ofstream::basic_ofstream

Creates an object of type basic_ofstream.

```
basic_ofstream( );
explicit basic_ofstream(
    const char *_Filename,
    ios_base::openmode _Mode = ios_base::out,
    int _Prot = (int)ios_base::_Openprot
);
explicit basic_ofstream(
    const wchar_t *_Filename,
    ios_base::openmode _Mode = ios_base::out,
    int _Prot = (int)ios_base::_Openprot
);
```

Parameters

_Filename

The name of the file to open.

_Mode

One of the enumerations in ios_base::openmode.

_Prot

The default file opening protection.

Remarks

The first constructor initializes the base class by calling basic_ostream(**sb**), where **sb** is the stored object of class basic_filebuf<*Elem*, *Tr*>. It also initializes **sb** by calling **basic_filebuf**<*Elem*, *Tr*>.

The second and third constructors initializes the base class by calling **basic_ostream**(**sb**). It also initializes **sb** by calling **basic_filebuf**<*Elem*, *Tr*> and then **sb**.open(*_Filename*, *_Mode* | **ios_base::out**). If the latter function returns a null pointer, the constructor calls setstate(**failbit**).

Example

The following example shows how to create a **basic_ofstream** object and write text to it.

```
// basic_ofstream_ctor.cpp
// compile with: /EHsc
#include <fstream>

using namespace std;

int main(int argc, char **argv)
{
    ofstream ofs("C:\\ofstream.txt");
    if (!ofs.bad())
    {
        ofs << "Writing to a basic_ofstream object..." << endl;
        ofs.close();
    }
}
```

Requirements

Header: <fstream>

See Also

Reference

basic_ofstream Class

iostream Programming
iostreams Conventions
Other Resources
basic_ofstream Members

basic_ofstream::close

Closes a file.

```
void close( );
```

Remarks

The member function calls rdbuf **->** close.

Example

See basic_filebuf::close for an example that uses **close**.

Requirements

Header: <fstream>

See Also

Reference

basic_ofstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ofstream Members

basic_ofstream::is_open

Indicates whether a file is open.

```
bool is_open( );
```

Return Value

true if the file is open, **false** otherwise.

Remarks

The member function returns rdbuf **->** is_open.

Example

```
// basic_ofstream_is_open.cpp
// compile with: /EHsc
#include <fstream>
#include <iostream>

int main( )
{
   using namespace std;
   ifstream file;
   // Open and close with a basic_filebuf
   file.rdbuf( )->open( "basic_ofstream_is_open.txt", ios::in );
   file.close( );
   if (file.is_open())
      cout << "it's open" << endl;
   else
      cout << "it's closed" << endl;
}
```

Output

```
it's closed
```

Requirements

Header: <fstream>

See Also

Reference

basic_ofstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ofstream Members

basic_ofstream::open

Opens a file.

```
void open(
    const char *_Filename,
    ios_base::openmode _Mode = ios_base::out,
    int _Prot = (int)ios_base::_Openprot
);
void open(
   const char *_Filename,
   ios_base::openmode _Mode
);
void open(
    const wchar_t *_Filename,
    ios_base::openmode _Mode = ios_base::out,
    int _Prot = (int)ios_base::_Openprot
);
void open(
   const wchar_t *_Filename,
   ios_base::openmode _Mode
);
```

Parameters

_Filename

The name of the file to open.

_Mode

One of the enumerations in ios_base::openmode.

_Prot

The defaut file opening protection.

Remarks

The member function calls rdbuf **->** open(*_Filename*, *_Mode* | **ios_base::out**). If that function returns a null pointer, the function calls setstate(**failbit**).

Example

See basic_filebuf::open for an example that uses **open**.

Requirements

Header: <fstream>

See Also

Reference

basic_ofstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ofstream Members

basic_ofstream::rdbuf

Returns the address of the stored stream buffer.

```
basic_filebuf<Elem, Tr> *rdbuf( ) const
```

Return Value

Returns the address of the stored stream buffer.

Example

See basic_filebuf::close for an example that uses **rdbuf**.

Requirements

Header: <fstream>

See Also

Reference

basic_ofstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ofstream Members

<functional>

Defines Standard Template Library (STL) functions that help construct function objects, also known as functors, and their binders.

For a list of all members of this header, see <functional> Members.

```
#include <functional>
```

Remarks

Algorithms require two types of function objects: unary and binary. Unary function objects require one argument, and binary function objects require two arguments. A function object and function pointers can be passed as a predicate to an algorithm, but function objects are also adaptable and increase the scope, flexibility, and efficiency of the STL. If, for example, a value needed to be bound to a function before being passed to an algorithm, then a function pointer could not be used. Function adaptors convert function pointers into adaptable function objects that can be bound to a value. The header <functional> also contains member function adaptors that allow member functions to be called as adaptable function objects. Functions are adaptable if they have nested type declarations specifying their argument and return types. The C++ Standard requires that this adaptability is implemented by having all standard object classes inherit from the unary_function or binary_function base classes. Function objects and their adaptors allow the STL to upgrade existing applications and help integrate the STL into the C++ programming environment.

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

Standard C++ Library Header Files

<functional> Members

Reference

Classes

binary_function	An empty base class that defines types that may be inherited by derived class that provides a binary function object.
binary_negate	A template class providing a member function that negates the return value of a specified binary function.
binder1st	A template class providing a constructor that converts a binary function object into a unary function object by binding the first argument of the binary function to a specified value.
binder2nd	A template class providing a constructor that converts a binary function object into a unary function object by binding the second argument of the binary function to a specified value.
const_mem_fun_ref_t	An adapter class that allows a const member function that takes no arguments to be called as a unary function object when initialized with a reference argument.
const_mem_fun_t	An adapter class that allows a const member function that takes no arguments to be called as a unary function object when initialized with a pointer argument.
const_mem_fun1_ref_t	An adapter class that allows a const member function that takes a single argument to be called as a binary function object when initialized with a reference argument.
const_mem_fun1_t	An adapter class that allows a const member function that takes a single argument to be called as a binary function object when initialized with a pointer argument.
divides	The class provides a predefined function object that performs the arithmetic operation of division on elements of a specified value type.
logical_and	The class provides a predefined function object that performs the logical operation of conjunction on elements of a specified value type and tests for the truth or falsity of the result.
logical_not	The class provides a predefined function object that performs the logical operation of negation on elements of a specified value type and tests for the truth or falsity of the result.
logical_or	The class provides a predefined function object that performs the logical operation of disjunction on elements of a specified value type and tests for the truth or falsity of the result.
mem_fun_ref_t	An adapter class that allows a **non_const** member function that takes no arguments to be called as a unary function object when initialized with a reference argument.
mem_fun_t	An adapter class that allows a **non_const** member function that takes no arguments to be called as a unary function object when initialized with a pointer argument.
mem_fun1_ref_t	An adapter class that allows a **non_const** member function that takes a single argument to be called as a binary function object when initialized with a reference argument.
mem_fun1_t	An adapter class that allows a **non_const** member function that takes a single argument to be called as a binary function object when initialized with a pointer argument.
minus	The class provides a predefined function object that performs the arithmetic operation of subtraction on elements of a specified value type.

modulus	The class provides a predefined function object that performs the arithmetic operation of modulus on elements of a specified value type.
multiplies	The class provides a predefined function object that performs the arithmetic operation of multiplication on elements of a specified value type.
negate	The class provides a predefined function object that returns the negative of an element value.
plus	The class provides a predefined function object that performs the arithmetic operation of addition on elements of a specified value type.
unary_function	An empty base class that defines types that may be inherited by derived class that provides a unary function object.
unary_negate	A template class providing a member function that negates the return value of a specified unary function.

Functions

bind1st	A helper template function that creates an adaptor to convert a binary function object into a unary function object by binding the first argument of the binary function to a specified value.
bind2nd	A helper template function that creates an adaptor to convert a binary function object into a unary function object by binding the second argument of the binary function to a specified value.
equal_to	A binary predicate that tests whether a value of a specified type is equal to another value of that type.
greater_equal	A binary predicate that tests whether a value of a specified type is greater than or equal to another value of that type.
greater	A binary predicate that tests whether a value of a specified type is greater than another value of that type.
less_equal	A binary predicate that tests whether a value of a specified type is less than or equal to another value of that type.
less	A binary predicate that tests whether a value of a specified type is less than another value of that type.
mem_fun_ref	A helper template function used to construct function object adaptors for member functions when initialized with reference arguments.
mem_fun	Helper template functions used to construct function object adaptors for member functions when initialized with pointer arguments.
not_equal_to	A binary predicate that tests whether a value of a specified type is not equal to another value of that type.
not1	Returns the complement of a unary predicate.
not2	Returns the complement of a binary predicate.

pointer_to_binary_function	Converts a binary function pointer into an adaptable binary function.
pointer_to_unary_function	Converts a unary function pointer into an adaptable unary function.
ptr_fun	A helper template function used to convert unary and binary function pointers, respectively, into u nary and binary adaptable functions.

See Also

Reference

<functional>
Thread Safety in the Standard C++ Library
Standard Template Library

<functional> Classes

For information about the classes in <functional>, see <functional> Members.

See Also

Reference

Standard Template Library

binary_function Class

An empty base class that defines types that may be inherited by derived class that provides a binary function object.

```
template<class Arg1, class Arg2, class Result>
   struct binary_function {
      typedef Arg1 first_argument_type;
      typedef Arg2 second_argument_type;
      typedef Result result_type;
   };
```

Remarks

The template class serves as a base for classes that define a member function of the form:

result_type operator()(**const first_argument_type&**,

const second_argument_type&) **const**

All such binary functions can refer to their first argument type as **first_argument_type**, their second argument type as **second_argument_type**, and their return type as ***result_type***.

Example

```
// functional_binary_function.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

template <class Type> class average:
binary_function<Type, Type, Type>
{
public:
   result_type operator( ) ( first_argument_type a,
                  second_argument_type b )
   {
      return (result_type) ( ( a + b ) / 2 );
   }
};

int main( )
{
   vector <double> v1, v2, v3 ( 6 );
   vector <double>::iterator Iter1, Iter2, Iter3;

   for ( int i = 1 ; i <= 6 ; i++ )
      v1.push_back( 11.0 / i );

   for ( int j = 0 ; j <= 5 ; j++ )
      v2.push_back( -2.0 * j );

   cout << "The vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   cout << "The vector v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << ")" << endl;
```

```
    // Finding the element-wise averages of the elements of v1 & v2
    transform ( v1.begin( ),  v1.end( ), v2.begin( ), v3.begin ( ),
       average<double>( ) );

    cout << "The element-wise averages are: ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
       cout << *Iter3 << " ";
    cout << ")" << endl;
 }
```

Output

```
The vector v1 = ( 11 5.5 3.66667 2.75 2.2 1.83333 )
The vector v2 = ( -0 -2 -4 -6 -8 -10 )
The element-wise averages are: ( 5.5 1.75 -0.166667 -1.625 -2.9 -4.08333 )
```

Requirements

Header: <functional>

See Also

Reference

binary_function Structure

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

binary_negate Class

A template class providing a member function that negates the return value of a specified binary function.

```
template<class Operation>
   class binary_negate
      : public binary_function <
         typename Operation::first_argument_type,
         typename Operation::second_argument_type,
         bool>
   {
   public:
   explicit binary_negate(
      const Operation& _Func
      );
   bool operator()(
      const typename Operation::first_argument_type& _Left,
      const typename Operation::second_argument_type& _Right
      ) const;
   };
```

Parameters

_Func

The binary function to be negated.

_Left

The left operand of the binary function to be negated.

_Right

The right operand of the binary function to be negated.

Return Value

The negation of the binary function.

Remarks

The template class stores a copy of a binary function object *_Func*. It defines its member function **operator()** as returning **!***_Func(_Left, _Right)*.

The constructor of **binary_negate** is rarely used directly. The helper function not2 is usually preferred to declare and use the **binary_negator** adaptor predicate.

Example

```
// functional_binary_negate.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <cstdlib>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1;

   int i;
   v1.push_back( 6262 );
   v1.push_back( 6262 );
   for ( i = 0 ; i < 5 ; i++ )
   {
```

```
      v1.push_back( rand( ) );
   }

   cout << "Original vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // To sort in ascending order,
   // use default binary predicate less<int>( )
   sort( v1.begin( ), v1.end( ) );
   cout << "Sorted vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // To sort in descending order,
   // use the binary_negate function
   sort( v1.begin( ), v1.end( ),
   binary_negate<less<int> >(less<int>( ) ) );

   // The helper function not2 could also have been used
   // in the above line and is usually preferred for convenience
   // sort( v1.begin( ), v1.end( ), not2(less<int>( ) ) );

   cout << "Resorted vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;
}
```

Output

```
Original vector v1 = ( 6262 6262 41 18467 6334 26500 19169 )
Sorted vector v1 = ( 41 6262 6262 6334 18467 19169 26500 )
Resorted vector v1 = ( 26500 19169 18467 6334 6262 6262 41 )
```

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

binder1st Class

A template class providing a constructor that converts a binary function object into a unary function object by binding the first argument of the binary function to a specified value.

```
template<class Operation>
class binder1st
   : public unary_function <
      typename Operation::second_argument_type,
      typename Operation::result_type>
  {
   public:
   typedef typename Operation::second_argument_type argument_type;
   typedef typename Operation::result_type result_type;
   binder1st(
      const Operation & _Func,
      const typename Operation::first_argument_type& _Left
   );
   result_type operator()(
      const argument_type& _Right
      ) const;
   result_type operator()(
   const argument_type& _Right
   ) const;
   protected:
   Operation op;
   typename Operation::first_argument_type value;
   };
```

Parameters

_Func

The binary function object to be converted to a unary function object.

_Left

The value to which the first argument of the binary function object is to be bound.

_Right

The value of the argument that the adapted binary object compares to the fixed value of the second argument.

Return Value

The unary function object that results from binding the first argument of the binary function object to the value *_Left*.

Remarks

The template class stores a copy of a binary function object *_Func* in **op**, and a copy of *_Left* in **value**. It defines its member function **operator()** as returning **op**(**value**, *_Right*).

If *_Func* is an object of type **Operation** and *c* is a constant , then bind1st (*_Func*, *c*) is equivalent to the **binder1st** class constructor **binder1st**<**Operation**> (*_Func*, *c*) and more convenient.

Example

```
// functional_binder1st.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
```

```
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 5; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( ";
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    // Count the number of integers > 10 in the vector
    vector<int>::iterator::difference_type result1;
    result1 = count_if(v1.begin(), v1.end(),
        binder1st<less<int> >(less<int>(), 10));
    cout << "The number of elements in v1 greater than 10 is: "
         << result1 << "." << endl;

    // Compare use of binder2nd fixing 2nd argument:
    // count the number of integers < 10 in the vector
    vector<int>::iterator::difference_type result2;
    result2 = count_if(v1.begin(), v1.end(),
        binder2nd<less<int> >(less<int>(), 10));
    cout << "The number of elements in v1 less than 10 is: "
         << result2 << "." << endl;
}
```

Output

```
The vector v1 = ( 0 5 10 15 20 25 )
The number of elements in v1 greater than 10 is: 3.
The number of elements in v1 less than 10 is: 2.
```

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

binder2nd Class

A template class providing a constructor that converts a binary function object into a unary function object by binding the second argument of the binary function to a specified value.

```
template<class Operation>
   class binder2nd
      : public unary_function <
         typename Operation::first_argument_type,
         typename Operation::result_type>
   {
   public:
   typedef typename Operation::first_argument_type argument_type;
   typedef typename Operation::result_type result_type;
   binder2nd(
      const Operation& _Func,
      const typename Operation::second_argument_type& _Right
      );
   result_type operator()(
      const argument_type& _Left
      ) const;
   result_type operator()(
   argument_type& _Left
      ) const;
   protected:
   Operation op;
   typename Operation::second_argument_type value;
   };
```

Parameters

_Func

The binary function object to be converted to a unary function object.

_Right

The value to which the second argument of the binary function object is to be bound.

_Left

The value of the argument that the adapted binary object compares to the fixed value of the second argument.

Return Value

The unary function object that results from binding the second argument of the binary function object to the value *_Right*.

Remarks

The template class stores a copy of a binary function object *_Func* in **op**, and a copy of *_Right* in **value**. It defines its member function **operator()** as returning **op**(*_Left*, **value**).

If *_Func* is an object of type **Operation** and c is a constant, then bind2nd (*_Func*, *c*) is equivalent to the **binder2nd** class constructor **binder2nd**<**Operation**> (*_Func*, *c*) and more convenient.

Example

```
// functional_binder2nd.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
```

```
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 5; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( ";
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    // Count the number of integers > 10 in the vector
    vector<int>::iterator::difference_type result1;
    result1 = count_if(v1.begin(), v1.end(),
        binder2nd<greater<int> >(greater<int>(), 10));
    cout << "The number of elements in v1 greater than 10 is: "
         << result1 << "." << endl;

    // Compare using binder1st fixing 1st argument:
    // count the number of integers < 10 in the vector
    vector<int>::iterator::difference_type result2;
    result2 = count_if(v1.begin(), v1.end(),
        binder1st<greater<int> >(greater<int>(), 10));
    cout << "The number of elements in v1 less than 10 is: "
         << result2 << "." << endl;
}
```

Output

```
The vector v1 = ( 0 5 10 15 20 25 )
The number of elements in v1 greater than 10 is: 3.
The number of elements in v1 less than 10 is: 2.
```

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

const_mem_fun_ref_t Class

An adapter class that allows a **const** member function that takes no arguments to be called as a unary function object when initialized with a reference argument.

```
template<class Result, class Type>
   struct const_mem_fun_ref_t
      : public unary_function<Type, Result>
   {
   explicit const_mem_fun_t(Result ( Type::* _Pm)( ) const );
      Result operator()(
         const Type& _Left
      ) const;
   };
```

Parameters

_Pm

A pointer to the member function of class **Type** to be converted to a function object.

_Left

The object that the *_Pm* member function is called on.

Return Value

An adaptable unary function.

Remarks

The template class stores a copy of *_Pm*, which must be a pointer to a member function of class **Type**, in a private member object. It defines its member function **operator()** as returning (**_Left**.* *_Pm*)() **const**.

Example

The constructor of **const_mem_fun_ref_t** is not usually used directly; the helper function **mem_fun_ref** is used to adapt member functions. See mem_fun_ref for an example of how to use member function adaptors.

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

const_mem_fun_t Class

An adapter class that allows a const member function that takes no arguments to be called as a unary function object when initialized with a reference argument.

```
template<class Result, class Type>
   struct const_mem_fun_t : public unary_function <Type *, Result>
   {
   explicit const_mem_fun_t( Result ( Type::* _Pm )( ) const );
      Result operator()(
         const Type* _Pleft
      ) const;
   };
```

Parameters

_Pm

A pointer to the member function of class **Type** to be converted to a function object.

_Pleft

The object that the *_Pm* member function is called on.

Return Value

An adaptable unary function.

Remarks

The template class stores a copy of *_Pm*, which must be a pointer to a member function of class **Type**, in a private member object. It defines its member function **operator()** as returning (*_Pleft*->* *_Pm*)() **const**.

Example

The constructor of **const_mem_fun_t** is not usually used directly; the helper function **mem_fun** is used to adapt member functions. See mem_fun for an example of how to use member function adaptors.

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

const_mem_fun1_ref_t Class

An adapter class that allows a **const** member function that takes a single argument to be called as a binary function object when initialized with a reference argument.

```
template<class Result, class Type, class Arg>
   struct const_mem_fun1_ref_t
      : public binary_function<Type, Arg, Result> {
   explicit const_mem_fun1_ref_t( Result (Type::*_Pm )( Arg ) const );
      Result operator()(
         const Type& _Left,
         Arg _Right
            ) const;
   };
```

Parameters

_Pm

A pointer to the member function of class **Type** to be converted to a function object.

_Left

The **const** object that the *_Pm* member function is called on.

_Right

The argument that is being given to *_Pm*.

Return Value

An adaptable binary function.

Remarks

The template class stores a copy of *_Pm*, which must be a pointer to a member function of class **Type**, in a private member object. It defines its member function **operator()** as returning (*_Left*.* *_Pm*)(*_Right*) **const**.

Example

The constructor of **const_mem_fun1_ref_t** is not usually used directly; the helper function **mem_fun_ref** is used to adapt member functions. See mem_fun_ref for examples of how to use member function adaptors.

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

const_mem_fun1_t Class

An adapter class that allows a **const** member function that takes a single argument to be called as a binary function object when initialized with a pointer argument.

```
template<class Result, class Type, class Arg>
   struct const_mem_fun1_t
      : public binary_function<const Type *, Arg, Result>
   {
   explicit const_mem_fun1_t( Result ( Type::* _Pm )( Arg ) const );
      Result operator()(
         const Type* _Pleft,
         Arg _Right
      ) const;
   };
```

Parameters

_Pm

A pointer to the member function of class **Type** to be converted to a function object.

_Pleft

The **const** object that the *_Pm* member function is called on.

_Right

The argument that is being given to *_Pm*.

Return Value

An adaptable binary function.

Remarks

The template class stores a copy of *_Pm*, which must be a pointer to a member function of class **Type**, in a private member object. It defines its member function **operator()** as returning (**_Pleft**->* *Pm)(***Right**) **const**.

Example

The constructor of **const_mem_fun1_t** is not usually used directly; the helper function **mem_fun** is used to adapt member functions. See mem_fun for an example of how to use member function adaptors.

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

divides Class

The class provides a predefined function object that performs the arithmetic operation of division on elements of a specified value type.

```
template<class Type>
   struct divides : public binary_function <Type, Type, Type>
   {
      Type operator()(
         const Type& _Left,
         const Type& _Right ) const;
   };
```

Parameters

_Left

A number that is of the parameter type **Type** that is to be divided by the number *_Right*.

_Right

A number that is of the parameter type **Type** that is to divide the number *_Left*.

Return Value

The quotient *_Left* / *_Right*.

Example

```
// functional_divides.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
   vector <double> v1, v2, v3 (6);
   vector <double>::iterator Iter1, Iter2, Iter3;

   int i;
   for ( i = 0 ; i <= 5 ; i++ )
   {
      v1.push_back( 7.0 * i );
   }

   int j;
   for ( j = 1 ; j <= 6 ; j++ )
   {
      v2.push_back( 2.0 * j);
   }

   cout << "The vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   cout << "The vector v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << ")" << endl;

   // Finding the element-wise quotients of the elements of v1 & v2
```

```
    transform ( v1.begin( ), v1.end( ), v2.begin( ), v3.begin ( ),
       divides<double>( ) );

    cout << "The element-wise quotients are: ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
       cout << *Iter3 << " ";
    cout << ")" << endl;
  }
```

Output

```
The vector v1 = ( 0 7 14 21 28 35 )
The vector v2 = ( 2 4 6 8 10 12 )
The element-wise quotients are: ( 0 1.75 2.33333 2.625 2.8 2.91667 )
```

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

logical_and Class

The class provides a predefined function object that performs the logical operation of conjunction on elements of a specified value type and tests for the truth or falsity of the result.

```
template<class Type>
   struct logical_and : public binary_function<Type, Type, bool>
   {
      bool operator()(
         const Type& _Left,
         const Type& _Right
      ) const;
   };
```

Parameters

_Left

The left operand of type **Type** in the conjunction to be tested.

_Right

The right operand of type **Type** in the conjunction to be tested.

Return Value

true if and only if *_Left* is **true** and *_Right* is **true**; otherwise **false**.

Example

```
// functional_logical_and.cpp
// compile with: /EHsc
#include <deque>
#include <algorithm>
#include <functional>
#include <iostream>

int main( )
{
   using namespace std;
   deque<bool> d1, d2, d3( 7 );
   deque<bool>::iterator iter1, iter2, iter3;

   int i;
   for ( i = 0 ; i < 7 ; i++ )
   {
      d1.push_back((bool)((rand() % 2) != 0));
   }

   int j;
   for ( j = 0 ; j < 7 ; j++ )
   {
      d2.push_back((bool)((rand() % 2 ) != 0));
   }

   cout << boolalpha;    // boolalpha I/O flag on

   cout << "Original deque:\n d1 = ( " ;
   for ( iter1 = d1.begin( ) ; iter1 != d1.end( ) ; iter1++ )
      cout << *iter1 << " ";
   cout << ")" << endl;

   cout << "Original deque:\n d2 = ( " ;
   for ( iter2 = d2.begin( ) ; iter2 != d2.end( ) ; iter2++ )
      cout << *iter2 << " ";
   cout << ")" << endl;
```

```
    // To find element-wise conjunction of the truth values
    // of d1 & d2, use the logical_and function object
    transform( d1.begin( ), d1.end( ), d2.begin( ),
       d3.begin( ), logical_and<bool>( ) );
    cout << "The deque which is the conjuction of d1 & d2 is:\n d3 = ( " ;
    for ( iter3 = d3.begin( ) ; iter3 != d3.end( ) ; iter3++ )
       cout << *iter3 << " ";
    cout << ")" << endl;
  }
```

Output

```
Original deque:
 d1 = ( true true false false true false false )
Original deque:
 d2 = ( false false false true true true true )
The deque which is the conjuction of d1 & d2 is:
 d3 = ( false false false false true false false )
```

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

logical_not Class

The class provides a predefined function object that performs the logical operation of negation on elements of a specified value type and tests for the truth or falsity of the result.

```
template<class Type>
   struct logical_not : public unary_function<Type, bool>
   {
      bool operator()(
         const Type& _Left
      ) const;
   };
```

Parameters

_Left

The operand of type **Type** in the negation to be tested.

Return Value

true if and only if *_Left* is **false**; **false** if and only if *_Left* is **true.**

Example

```
// functional_logical_not.cpp
// compile with: /EHsc
#include <deque>
#include <algorithm>
#include <functional>
#include <iostream>

int main( )
{
   using namespace std;
   deque<bool> d1, d2 ( 7 );
   deque<bool>::iterator iter1, iter2;

   int i;
   for ( i = 0 ; i < 7 ; i++ )
   {
      d1.push_back((bool)((i % 2) != 0));
   }

   cout << boolalpha;    // boolalpha I/O flag on

   cout << "Original deque:\n d1 = ( " ;
   for ( iter1 = d1.begin( ) ; iter1 != d1.end( ) ; iter1++ )
      cout << *iter1 << " ";
   cout << ")" << endl;

   // To flip all the truth values of the elements,
   // use the logical_not function object
   transform( d1.begin( ), d1.end( ), d2.begin( ),logical_not<bool>( ) );
   cout << "The deque with its values negated is:\n d2 = ( " ;
   for ( iter2 = d2.begin( ) ; iter2 != d2.end( ) ; iter2++ )
      cout << *iter2 << " ";
   cout << ")" << endl;
}
```

Output

```
Original deque:
 d1 = ( false true false true false true false )
The deque with its values negated is:
 d2 = ( true false true false true false true )
```

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

logical_or Class

The class provides a predefined function object that performs the logical operation of disjunction on elements of a specified value type and tests for the truth or falsity of the result.

```
template<class Type>
   struct logical_or : public binary_function<Type, Type, bool>
   {
      bool operator()(
         const Type& _Left,
         const Type& _Right
      ) const;
   };
```

Parameters

_Left

The left operand of type **Type** in the disjunction to be tested.

_Right

The right operand of type **Type** in the disjunction to be tested.

Return Value

true if *_Left* is **true** or *_Right* is **true**; **false** if and only if *_Left* is **false** and *_Right* is **false**.

Example

```
// functional_logical_or.cpp
// compile with: /EHsc
#include <deque>
#include <algorithm>
#include <functional>
#include <iostream>

int main( )
{
   using namespace std;
   deque <bool> d1, d2, d3( 7 );
   deque <bool>::iterator iter1, iter2, iter3;

   int i;
   for ( i = 0 ; i < 7 ; i++ )
   {
      d1.push_back((bool)((rand() % 2) != 0));
   }

   int j;
   for ( j = 0 ; j < 7 ; j++ )
   {
      d2.push_back((bool)((rand() % 2) != 0));
   }

   cout << boolalpha;    // boolalpha I/O flag on

   cout << "Original deque:\n d1 = ( " ;
   for ( iter1 = d1.begin( ) ; iter1 != d1.end( ) ; iter1++ )
      cout << *iter1 << " ";
   cout << ")" << endl;

   cout << "Original deque:\n d2 = ( " ;
   for ( iter2 = d2.begin( ) ; iter2 != d2.end( ) ; iter2++ )
      cout << *iter2 << " ";
   cout << ")" << endl;
```

```
    // To find element-wise disjunction of the truth values
    // of d1 & d2, use the logical_or function object
    transform( d1.begin( ), d1.end( ), d2.begin( ),
       d3.begin( ), logical_or<bool>( ) );
    cout << "The deque which is the disjuction of d1 & d2 is:\n d3 = ( " ;
    for ( iter3 = d3.begin( ) ; iter3 != d3.end( ) ; iter3++ )
       cout << *iter3 << " ";
    cout << ")" << endl;
  }
```

Output

```
Original deque:
 d1 = ( true true false false true false false )
Original deque:
 d2 = ( false false false true true true true )
The deque which is the disjuction of d1 & d2 is:
 d3 = ( true true false true true true true )
```

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

mem_fun_t Class

An adapter class that allows a **non_const** member function that takes no arguments to be called as a unary function object when initialized with a pointer argument.

```
template<class Result, class Type>
   struct mem_fun_t : public unary_function<Type *, Result> {
      explicit mem_fun_t(Result ( Type::* _Pm )( ) );
      Result operator()( Type* _Pleft ) const;
   };
```

Parameters

_Pm

A pointer to the member function of class **Type** to be converted to a function object.

_Pleft

The object that the *_Pm* member function is called on.

Return Value

An adaptable unary function.

Remarks

The template class stores a copy of *_Pm*, which must be a pointer to a member function of class **Type**, in a private member object. It defines its member function **operator()** as returning (*_Pleft*->* *_Pm*)( ).

Example

The constructor of **mem_fun_t** is not usually used directly; the helper function **mem_fun** is used to adapt member functions. See mem_fun for an example of how to use member function adaptors.

Requirements

Header: <functional>

See Also

Reference

<functional>

Thread Safety in the Standard C++ Library

Standard Template Library

mem_fun_ref_t Class

An adapter class that allows a **non_const** member function that takes no arguments to be called as a unary function object when initialized with a reference argument.

```
template<class Result, class Type>
   struct mem_fun_ref_t : public unary_function<Type, Result> {
      explicit mem_fun_t(
         Result ( Type::*_Pm )( )
         );
      Result operator()( Type& _Left ) const;
   };
```

Parameters

_Pm

A pointer to the member function of class **Type** to be converted to a function object.

_Left

The object that the *_Pm* member function is called on.

Return Value

An adaptable unary function.

Remarks

The template class stores a copy of *_Pm*, which must be a pointer to a member function of class **Type**, in a private member object. It defines its member function **operator()** as returning (**_Left**.* *_Pm*)( ).

Example

The constructor of **mem_fun_ref_t** is not usually used directly; the helper function **mem_fun_ref** is used to adapt member functions. See mem_fun_ref for an example of how to use member function adaptors.

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

mem_fun1_t Class

An adapter class that allows a **non_const** member function that takes a single argument to be called as a binary function object when initialized with a pointer argument.

```
template<class Result, class Type, class Arg>
   struct mem_fun1_t : public binary_function<Type *, Arg, Result> {
      explicit mem_fun1_t(
         Result (Type::* _Pm )( Arg )
         );
      Result operator()(
         Type* _Pleft,
         Arg _Right
         ) const;
   };
```

Parameters

_Pm

A pointer to the member function of class **Type** to be converted to a function object.

_Pleft

The object that the *_Pm* member function is called on.

_Right

The argument that is being given to *_Pm*.

Return Value

An adaptable binary function.

Remarks

The template class stores a copy of *_Pm*, which must be a pointer to a member function of class **Type**, in a private member object. It defines its member function **operator()** as returning (**_Pleft**->* *_Pm*)(**_Right**).

Example

The constructor of **mem_fun1_t** is not usually used directly; the helper function **mem_fun** is used to adapt member functions. See mem_fun for an example of how to use member function adaptors.

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

mem_fun1_ref_t Class

An adapter class that allows a **non_const** member function that takes a single argument to be called as a binary function object when initialized with a reference argument.

```
template<class Result, class Type, class Arg>
   struct mem_fun1_ref_t : public binary_function<Type, Arg, Result> {
      explicit mem_fun1_ref_t(Result (
         Type::* _Pm )( Arg )
         );
      Result operator()(
         Type& _Left,
         Arg _Right
         ) const;
   };
```

Parameters

_Pm

A pointer to the member function of class **Type** to be converted to a function object.

_Left

The object that the *_Pm* member function is called on.

_Right

The argument that is being given to *_Pm*.

Return Value

An adaptable binary function.

Remarks

The template class stores a copy of *_Pm*, which must be a pointer to a member function of class **Type**, in a private member object. It defines its member function **operator()** as returning (**_Left**.* *_Pm*)(**_Right**).

Example

The constructor of **mem_fun1_ref_t** is not usually used directly; the helper function **mem_fun_ref** is used to adapt member functions. See mem_fun_ref for an example of how to use member function adaptors.

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

minus Class

The class provides a predefined function object that performs the arithmetic operation of subtraction on elements of a specified value type.

```
template<class Type>
   struct minus : public binary_function <Type, Type, Type>
   {
      Type operator()(
         const Type& _Left,
         const Type& _Right
      ) const;
   };
```

Parameters

_Left

A number that is of the parameter type **Type** from which another number is to be subtracted by the function object.

_Right

A number that is of the parameter type **Type** that is to be subtracted from another by the function object.

Return Value

The sum *_Left* - *_Right*.

Example

```
// functional_minus.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
   vector <int> v1, v2, v3 ( 6 );
   vector <int>::iterator Iter1, Iter2, Iter3;

   int i;
   for ( i = 0 ; i <= 5 ; i++ )
   {
      v1.push_back( 4 * i + 1);
   }

   int j;
   for ( j = 0 ; j <= 5 ; j++ )
   {
      v2.push_back( 3 * j - 1);
   }

   cout << "The vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   cout << "The vector v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << ")" << endl;
```

```
    // Finding the element-wise diference of the elements of v1 & v2
    transform ( v1.begin( ),  v1.end( ), v2.begin( ), v3.begin ( ),
       minus<int>( ) );

    cout << "The element-wise differences between v1 and v2 are: ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
       cout << *Iter3 << " ";
    cout << ")" << endl;
 }
```

Output

```
The vector v1 = ( 1 5 9 13 17 21 )
The vector v2 = ( -1 2 5 8 11 14 )
The element-wise differences between v1 and v2 are: ( 2 3 4 5 6 7 )
```

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

modulus Class

The class provides a predefined function object that performs the modular operation of calculating the remainder after division between elements.

```
template<class Type>
   struct modulus : public binary_function <Type, Type, Type>
   {
      Type operator()(
         const Type& _Left,
         const Type& _Right
      ) const;
   };
```

Parameters

_Left

A number that is of the parameter type **Type** into which another number is to be divided by the function object.

_Right

A number that is of the parameter type **Type** that is to be divided into another by the function object.

Return Value

The remainder of the division *_Left* / *_Right*.

Remarks

The **modulus**<**Type**> operator is restricted to integral types for the basic data types, or to any user-defined type that implements **operator%**.

Example

```
// functional_modulus.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
   vector <int> v1, v2, v3 ( 6 );
   vector <int>::iterator Iter1, Iter2, Iter3;

   int i;
   for ( i = 1 ; i <= 6 ; i++ )
   {
      v1.push_back( 5 * i );
   }

   int j;
   for ( j = 1 ; j <= 6 ; j++ )
   {
      v2.push_back( 3 * j );
   }

   cout << "The vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   cout << "The vector v2 = ( " ;
```

```
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
       cout << *Iter2 << " ";
    cout << ")" << endl;

    // Finding the element-wise remainders of the elements of v1 & v2
    transform (v1.begin( ),  v1.end( ), v2.begin( ), v3.begin ( ),
       modulus<int>() );

    cout << "The element-wise remainders of the modular division\n are: ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
       cout << *Iter3 << " ";
    cout << ")" << endl;
 }
```

Output

```
The vector v1 = ( 5 10 15 20 25 30 )
The vector v2 = ( 3 6 9 12 15 18 )
The element-wise remainders of the modular division
 are: ( 2 4 6 8 10 12 )
```

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

multiplies Class

The class provides a predefined function object that performs the arithmetic operation of multiplication on elements of a specified value type.

```
template<class Type>
   struct multiplies : public binary_function <Type, Type, Type>
   {
      Type operator()(
         const Type& _Left,
         const Type& _Right
      ) const;
   };
```

Parameters

_Left

A number that is of the parameter type **Type** that is to be multiplied by the function object.

_Right

A number that is of the parameter type **Type** that is to be multiplied by the function object.

Return Value

The product of the multiplication *_Left* * *_Right*.

Example

```
// functional_multiplies.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
   vector <int> v1, v2, v3 ( 6 );
   vector <int>::iterator Iter1, Iter2, Iter3;

   int i;
   for ( i = 1 ; i <= 6 ; i++ )
   {
      v1.push_back( 2 * i );
   }

   int j;
   for ( j = 1 ; j <= 6 ; j++ )
   {
      v2.push_back( 3 * j );
   }

   cout << "The vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   cout << "The vector v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << ")" << endl;
```

```
    // Finding the element-wise products of the elements of v1 & v2
    transform ( v1.begin( ),  v1.end( ), v2.begin( ), v3.begin ( ),
       multiplies<int>( ) );

    cout << "The element-wise products of vectors V1 & v2\n are: ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
       cout << *Iter3 << " ";
    cout << ")" << endl;
 }
```

Output

```
The vector v1 = ( 2 4 6 8 10 12 )
The vector v2 = ( 3 6 9 12 15 18 )
The element-wise products of vectors V1 & v2
 are: ( 6 24 54 96 150 216 )
```

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

negate Class

The class provides a predefined function object that returns the negative of an element value.

```
template<class Type>
   struct negate : public unary_function<Type, Type>
   {
      Type operator()(
         const Type& _Left
      ) const;
   };
```

Parameters

_Left

The value of an element whose negative is to be returned.

Return Value

The negative of the parameter value: - *_Left*.

Example

```
// functional_negate.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
   vector <int> v1, v2 ( 8 );
   vector <int>::iterator Iter1, Iter2;

   int i;
   for ( i = -2 ; i <= 5 ; i++ )
   {
      v1.push_back( 5 * i );
   }

   cout << "The vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // Finding the element-wise negatives of the vector v1
   transform ( v1.begin( ),  v1.end( ), v2.begin( ), negate<int>( ) );

   cout << "The negated elements of the vector = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << ")" << endl;
}
```

Output

```
The vector v1 = ( -10 -5 0 5 10 15 20 25 )
The negated elements of the vector = ( 10 5 0 -5 -10 -15 -20 -25 )
```

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

plus Class

The class provides a predefined function object that performs the arithmetic operation of addition on elements of a specified value type.

```
template<class Type>
   struct plus : public binary_function <Type, Type, Type>
   {
      Type operator()(
         const Type& _Left,
         const Type& _Right
      ) const;
   };
```

Parameters

_Left

A number that is of the parameter type **Type** that is to be added by the function object.

_Right

A number that is of the parameter type **Type** that is to be added by the function object.

Return Value

The sum *_Left* + *_Right*.

Example

```
// functional_plus.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
   vector <double> v1, v2, v3 ( 6 );
   vector <double>::iterator Iter1, Iter2, Iter3;

   int i;
   for ( i = 0 ; i <= 5 ; i++ )
      v1.push_back( 4 * i );

   int j;
   for ( j = 0 ; j <= 5 ; j++ )
      v2.push_back( -2.0 * j - 4 );

   cout << "The vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   cout << "The vector v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << ")" << endl;

   // Finding the element-wise sums of the elements of v1 & v2
   transform (v1.begin( ), v1.end( ), v2.begin( ), v3.begin ( ), plus<double>( ) );

   cout << "The element-wise sums are: ( " ;
```

```
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
       cout << *Iter3 << " ";
    cout << ")" << endl;
 }
```

Output

```
The vector v1 = ( 0 4 8 12 16 20 )
The vector v2 = ( -4 -6 -8 -10 -12 -14 )
The element-wise sums are: ( -4 -2 0 2 4 6 )
```

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

unary_function Class

An empty base class that defines types that may be inherited by derived class that provides a unary function object.

```
template<class Arg, class Result>
   struct unary_function {
      typedef Arg argument_type;
      typedef Result result_type;
   };
```

Remarks

The template class serves as a base for classes that define a member function of the form **result_type operator()**(**const argument_type&**) **const**.

All such derived unary functions can refer to their sole argument type as **argument_type** and their return type as **result_type**.

Example

```
// functional_unary_function.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

// Creation of a user-defined function object
// that inherits from the unary_function base class
class greaterthan10: unary_function<int, bool>
{
public:
    result_type operator()(argument_type i)
    {
        return (result_type)(i > 10);
    }
};

int main()
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 5; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( " ;
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    vector<int>::iterator::difference_type result1;
    result1 = count_if(v1.begin(), v1.end(), greaterthan10());
    cout << "The number of elements in v1 greater than 10 is: "
         << result1 << "." << endl;
}
```

Output

```
The vector v1 = ( 0 5 10 15 20 25 )
```

```
The number of elements in v1 greater than 10 is: 3.
```

Requirements

Header: <functional>

See Also

Reference

unary_function<> Structure

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

unary_negate Class

A template class providing a member function that negates the return value of a specified unary function.

```
template<class Predicate>
   class unary_negate
      : public unary_function<
         typename Predicate::argument_type,
         bool>
   {
   public:
   explicit unary_negate(
      const Predicate& _Func
   );
   bool operator()(
      const typename Predicate::argument_type& _Left ) const;
   };
```

Parameters

_Func

The unary function to be negated.

_Left

The operand of the unary function to be negated.

Return Value

The negation of the unary function.

Remarks

The template class stores a copy of a unary function object *_Func*. It defines its member function **operator()** as returning **!***_Func(_Left)*.

The constructor of **unary_negate** is rarely used directly. The helper function not1 provides an easier way to declare and use the **unary_negator** adaptor predicate.

Example

```
// functional_unary_negate.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 7; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( ";
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;
```

```
    vector<int>::iterator::difference_type result1;
    // Count the elements greater than 10
    result1 = count_if(v1.begin(), v1.end(), bind2nd(greater<int>(), 10));
    cout << "The number of elements in v1 greater than 10 is: "
         << result1 << "." << endl;

    vector<int>::iterator::difference_type result2;
    // Use the negator to count the elements less than or equal to 10
    result2 = count_if(v1.begin(), v1.end(),
        unary_negate<binder2nd <greater<int> > >(bind2nd(greater<int>(),10)));

    // The following helper function not1 also works for the above line
    // not1(bind2nd(greater<int>(), 10)));

    cout << "The number of elements in v1 not greater than 10 is: "
         << result2 << "." << endl;
}
```

Output

```
The vector v1 = ( 0 5 10 15 20 25 30 35 )
The number of elements in v1 greater than 10 is: 5.
The number of elements in v1 not greater than 10 is: 3.
```

Requirements

Header: <functional>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<functional> Members

<functional> Functions

For information about the functions in <functional>, see <functional> Members.

See Also

Reference

Standard Template Library

bind1st

A helper template function that creates an adaptor to convert a binary function object into a unary function object by binding the first argument of the binary function to a specified value.

```
template<class Operation, class Type>
   binder1st <Operation> bind1st(
      const Operation& _Func,
      const Type& _Left
   );
```

Parameters

_Func

The binary function object to be converted to a unary function object.

_Left

The value to which the first argument of the binary function object is to be bound.

Return Value

The unary function object that results from binding the first argument of the binary function object to the value *_Left*.

Remarks

Function binders are a kind of function adaptor and, because they return function objects, can be used in certain types of function composition to construct more complicated and powerful expressions.

If *_Func* is an object of type **Operation** and *c* is a constant , then **bind1st** (*_Func*, *c*) is equivalent to the binder1st class constructor **binder1st**<**Operation**> (*_Func*, *c*) and is more convenient.

Example

```
// functional_bind1st.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

// Creation of a user-defined function object
// that inherits from the unary_function base class
class greaterthan5: unary_function<int, bool>
{
public:
    result_type operator()(argument_type i)
    {
        return (result_type)(i > 5);
    }
};

int main()
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 5; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( " ;
```

```
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    // Count the number of integers > 10 in the vector
    vector<int>::iterator::difference_type result1a;
    result1a = count_if(v1.begin(), v1.end(), bind1st(less<int>(), 10));
    cout << "The number of elements in v1 greater than 10 is: "
         << result1a << "." << endl;

    // Compare: counting the number of integers > 15 in the vector
    // with a user defined function object
    vector<int>::iterator::difference_type result1b;
    result1b = count_if(v1.begin(), v1.end(), greaterthan5());
    cout << "The number of elements in v1 greater than 15 is: "
         << result1b << "." << endl;

    // Count the number of integers < 10 in the vector
    vector<int>::iterator::difference_type result2;
    result2 = count_if(v1.begin(), v1.end(), bind2nd(less<int>(), 10));
    cout << "The number of elements in v1 less than 10 is: "
         << result2 << "." << endl;
}
```

Output

```
The vector v1 = ( 0 5 10 15 20 25 )
The number of elements in v1 greater than 10 is: 3.
The number of elements in v1 greater than 15 is: 4.
The number of elements in v1 less than 10 is: 2.
```

Requirements

Header: <functional>

See Also

Reference

Standard Template Library

Other Resources

<functional> Members

bind2nd

A helper template function that creates an adaptor to convert a binary function object into a unary function object by binding the second argument of the binary function to a specified value.

```
template<class Operation, class Type>
   binder2nd <Operation> bind2nd(
      const Operation& _Func,
      const Type& _Right
   );
```

Parameters

_Func

The binary function object to be converted to a unary function object.

_Right

The value to which the second argument of the binary function object is to be bound.

Return Value

The unary function object that results from binding the second argument of the binary function object to the value *_Right*.

Remarks

Function binders are a kind of function adaptor and, because they return function objects, can be used in certain types of function composition to construct more complicated and powerful expressions.

If *_Func* is an object of type **Operation** and *c* is a constant, then **bind2nd** (*_Func*, *c*) is equivalent to the binder2nd class constructor **binder2nd<Operation>** (*_Func*, *c*) and more convenient.

Example

```
// functional_bind2nd.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

// Creation of a user-defined function object
// that inherits from the unary_function base class
class greaterthan15: unary_function<int, bool>
{
public:
    result_type operator()(argument_type i)
    {
        return (result_type)(i > 15);
    }
};

int main()
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 5; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( ";
```

```
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    // Count the number of integers > 10 in the vector
    vector<int>::iterator::difference_type result1a;
    result1a = count_if(v1.begin(), v1.end(), bind2nd(greater<int>(), 10));
    cout << "The number of elements in v1 greater than 10 is: "
         << result1a << "." << endl;

    // Compare counting the number of integers > 15 in the vector
    // with a user-defined function object
    vector<int>::iterator::difference_type result1b;
    result1b = count_if(v1.begin(), v1.end(), greaterthan15());
    cout << "The number of elements in v1 greater than 15 is: "
         << result1b << "." << endl;

    // Count the number of integers < 10 in the vector
    vector<int>::iterator::difference_type result2;
    result2 = count_if(v1.begin(), v1.end(), bind1st(greater<int>(), 10));
    cout << "The number of elements in v1 less than 10 is: "
         << result2 << "." << endl;
}
```

Output

```
The vector v1 = ( 0 5 10 15 20 25 )
The number of elements in v1 greater than 10 is: 3.
The number of elements in v1 greater than 15 is: 2.
The number of elements in v1 less than 10 is: 2.
```

Requirements

Header: <functional>

See Also

Reference

Standard Template Library

Other Resources

<functional> Members

greater

A binary predicate that tests whether a value of a specified type is greater than another value of that type.

```
template<class Type>
   struct greater : public binary_function <Type, Type, bool>
   {
      bool operator()(
         const Type& _Left,
         const Type& _Right
      ) const;
   };
```

Parameters

_Left

The left operand of type **Type** in the inequality to be tested.

_Right

The right operand of type **Type** in the inequality to be tested.

Return Value

true if *_Left* > *_Right*; **false** if *_Left* <= *_Right*.

Remarks

The binary predicate **greater**<**Type**> provides a strict weak ordering of a set of element values of type **Type** into equivalence classes if and only if this **Type** satisfies the standard mathematical requirements for being so ordered. The specializations for any pointer type yield a total ordering of elements in that all elements of distinct values are ordered with respect to each other.

Example

```
// functional_greater.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <cstdlib>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1;

   int i;
   for ( i = 0 ; i < 8 ; i++ )
   {
      v1.push_back( rand( ) );
   }

   cout << "Original vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // To sort in ascending order,
   // use default binary predicate less<int>( )
   sort( v1.begin( ), v1.end( ) );
   cout << "Sorted vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;
```

```
    // To sort in descending order,
    // specify binary predicate greater<int>( )
    sort( v1.begin( ), v1.end( ), greater<int>( ) );
    cout << "Resorted vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << ")" << endl;
  }
```

Output

```
Original vector v1 = ( 41 18467 6334 26500 19169 15724 11478 29358 )
Sorted vector v1 = ( 41 6334 11478 15724 18467 19169 26500 29358 )
Resorted vector v1 = ( 29358 26500 19169 18467 15724 11478 6334 41 )
```

Requirements

Header: <functional>

See Also

Reference

Standard Template Library

Other Resources

<functional> Members

greater_equal

A binary predicate that tests whether a value of a specified type is greater than or equal to another value of that type.

```
template<class Type>
   struct greater_equal : public binary_function <Type, Type, bool>
   {
      bool operator()(
         const Type& _Left,
         const Type& _Right
      ) const;
   };
```

Parameters

_Left

The left operand of type **Type** in the inequality to be tested.

_Right

The right operand of type **Type** in the inequality to be tested.

Return Value

true if *_Left* >= *_Right*; **false** if *_Left* < *_Right*.

Remarks

The binary predicate **greater_equal**<**Type**> provides a strict weak ordering of a set of element values of type **Type** into equivalence classes if and only if this **Type** satisfies the standard mathematical requirements for being so ordered. The specializations for any pointer type yield a total ordering of elements in that all elements of distinct values are ordered with respect to each other.

Example

```
// functional_greater_equal.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <cstdlib>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1;

   int i;
   v1.push_back( 6262 );
   v1.push_back( 6262 );
   for ( i = 0 ; i < 5 ; i++ )
   {
      v1.push_back( rand( ) );
   }

   cout << "Original vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // To sort in ascending order,
   // use default binary predicate less<int>( )
   sort( v1.begin( ), v1.end( ) );
   cout << "Sorted vector v1 = ( " ;
```

```
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << ")" << endl;

    // To sort in descending order,
    // specify binary predicate greater_equal<int>( )
    sort( v1.begin( ), v1.end( ), greater_equal<int>( ) );
    cout << "Resorted vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << ")" << endl;
  }
```

Output

```
Original vector v1 = ( 6262 6262 41 18467 6334 26500 19169 )
Sorted vector v1 = ( 41 6262 6262 6334 18467 19169 26500 )
Resorted vector v1 = ( 26500 19169 18467 6334 6262 6262 41 )
```

Requirements

Header: <functional>

See Also

Reference

Standard Template Library

Other Resources

<functional> Members

less

A binary predicate that tests whether a value of a specified type is less than another value of that type.

```
template<class Type>
   struct less : public binary_function <Type, Type, bool>
   {
      bool operator()(
         const Type& _Left,
         const Type& _Right
      ) const;
   };
```

Parameters

_Left

The left operand of type **Type** in the inequality to be tested.

_Right

The right operand of type **Type** in the inequality to be tested.

Return Value

true if *_Left* < *_Right*; **false** if *_Left* >= *_Right*.

Remarks

The binary predicate **less**<**Type**> provides a strict weak ordering of a set of element values of type **Type** into equivalence classes if and only if this **Type** satisfies the standard mathematical requirements for being so ordered. The specializations for any pointer type yield a total ordering of elements in that all elements of distinct values are ordered with respect to each other.

Example

```
// functional_less.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

struct MyStruct {
   MyStruct(int i) : m_i(i){}

   bool operator < (const MyStruct & rhs) const {
      return m_i < rhs.m_i;
   }

   int m_i;
};

int main() {
   using namespace std;
   vector <MyStruct> v1;
   vector <MyStruct>::iterator Iter1;
   vector <MyStruct>::reverse_iterator rIter1;

   int i;
   for ( i = 0 ; i < 7 ; i++ )
       v1.push_back( MyStruct(rand()));

   cout << "Original vector v1 = ( " ;
   for ( Iter1 = v1.begin() ; Iter1 != v1.end() ; Iter1++ )
cout << Iter1->m_i << " ";
   cout << ")" << endl;
```

```
    // To sort in ascending order,
    sort( v1.begin( ), v1.end( ), less<MyStruct>());

    cout << "Sorted vector v1 = ( " ;
    for ( Iter1 = v1.begin() ; Iter1 != v1.end() ; Iter1++ )
  cout << Iter1->m_i << " ";
    cout << ")" << endl;
  }
```

Output

```
Original vector v1 = ( 41 18467 6334 26500 19169 15724 11478 )
Sorted vector v1 = ( 41 6334 11478 15724 18467 19169 26500 )
```

Requirements

Header: <functional>

See Also

Reference

Standard Template Library

Other Resources

<functional> Members

less_equal

A binary predicate that tests whether a value of a specified type is less than or equal to another value of that type.

```
template<class Type>
   struct less_equal : public binary_function <Type, Type, bool>
   {
      bool operator()(
         const Type& _Left,
         const Type& _Right
      ) const;
   };
```

Parameters

_Left

The left operand of type **Type** in the inequality to be tested.

_Right

The right operand of type **Type** in the inequality to be tested.

Return Value

true if *_Left* <= *_Right*; **false** if *_Left* > *_Right*.

Remarks

The binary predicate **less_equal**<**Type**> provides a strict weak ordering of a set of element values of type **Type** into equivalence classes if and only if this **Type** satisfies the standard mathematical requirements for being so ordered. The specializations for any pointer type yield a total ordering of elements in that all elements of distinct values are ordered with respect to each other.

Example

```
// functional_less_equal.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <cstdlib>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1;
   vector <int>::reverse_iterator rIter1;

   int i;
   for ( i = 0 ; i < 5 ; i++ )
   {
      v1.push_back( rand( ) );
   }
   for ( i = 0 ; i < 3 ; i++ )
   {
      v1.push_back( 2836 );
   }

   cout << "Original vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // To sort in ascending order,
```

```
    // use the binary predicate less_equal<int>( )
    sort( v1.begin( ), v1.end( ), less_equal<int>( ) );
    cout << "Sorted vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
       cout << *Iter1 << " ";
    cout << ")" << endl;
  }
```

Output

```
Original vector v1 = ( 41 18467 6334 26500 19169 2836 2836 2836 )
Sorted vector v1 = ( 41 2836 2836 2836 6334 18467 19169 26500 )
```

Requirements

Header: <functional>

See Also

Reference

Standard Template Library

Other Resources

<functional> Members

mem_fun

Helper template functions used to construct function object adaptors for member functions when initialized with pointer arguments.

```
template<class Result, class Type>
   mem_fun_t<Result, Type> mem_fun (
      Result(Type::* _Pm )( )
);

template<class Result, class Type, class Arg>
   mem_fun1_t<Result, Type, Arg> mem_fun(
      Result (Type::* _Pm )( Arg )
);

template<class Result, class Type>
   const_mem_fun_t<Result, Type>
      mem_fun(Result (Type::* _Pm )( ) const
);

template<class Result, class Type, class Arg>
   const_mem_fun1_t<Result, Type, Arg>
      mem_fun(Result (Type::* _Pm )( Arg ) const
);
```

Parameters

_Pm

A pointer to the member function of class **Type** to be converted to a function object.

Return Value

A **const** or **non_const** function object of type **mem_fun_t** or **mem_fun1_t**.

Example

```
// functional_mem_fun.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

class StoreVals
{
    int val;
public:
    StoreVals() { val = 0; }
    StoreVals(int j) { val = j; }

    bool display() { cout << val << " "; return true; }
    int squareval() { val *= val; return val; }
    int lessconst(int k) {val -= k; return val; }
};

int main( )
{
    vector<StoreVals *> v1;

    StoreVals sv1(5);
    v1.push_back(&sv1);
    StoreVals sv2(10);
    v1.push_back(&sv2);
```

```
    StoreVals sv3(15);
    v1.push_back(&sv3);
    StoreVals sv4(20);
    v1.push_back(&sv4);
    StoreVals sv5(25);
    v1.push_back(&sv5);

    cout << "The original values stored are: " ;
    for_each(v1.begin(), v1.end(), mem_fun<bool, StoreVals>(&StoreVals::display));
    cout << endl;

    // Use of mem_fun calling member function through a pointer
    // square each value in the vector using squareval ()
    for_each(v1.begin(), v1.end(), mem_fun<int, StoreVals>(&StoreVals::squareval));
    cout << "The squared values are: " ;
    for_each(v1.begin(), v1.end(), mem_fun<bool, StoreVals>(&StoreVals::display));
    cout << endl;

    // Use of mem_fun1 calling member function through a pointer
    // subtract 5 from each value in the vector using lessconst ()
    for_each(v1.begin(), v1.end(),
        bind2nd (mem_fun1<int, StoreVals,int>(&StoreVals::lessconst), 5));
    cout << "The squared values less 5 are: " ;
    for_each(v1.begin(), v1.end(), mem_fun<bool, StoreVals>(&StoreVals::display));
    cout << endl;
}
```

Output

```
The original values stored are: 5 10 15 20 25
The squared values are: 25 100 225 400 625
The squared values less 5 are: 20 95 220 395 620
```

Requirements

Header: <functional>

See Also

Reference

Standard Template Library

Other Resources

<functional> Members

mem_fun_ref

Helper template functions used to construct function object adaptors for member functions when initialized with reference arguments.

```
template<class Result, class Type>
   mem_fun_ref_t<Result, Type> mem_fun_ref(
      Result (Type::*_Pm )( )
   );
template<class Result, class Type, class Arg>
   mem_fun1_ref_t<Result, Type, Arg>
      mem_fun_ref(
         Result (Type::*_Pm )( Arg )
      );
template<class Result, class Type>
   const_mem_fun_ref_t<Result, Type>
      mem_fun_ref(
         Result Type::*_Pm )( ) const
      );
template<class Result, class Type, class Arg>
   const_mem_fun1_ref_t<Result, Type, Arg>
      mem_fun_ref(
         Result ( _Type::*_Pm )( Arg ) const
      );
```

Parameters

_ Pm

A pointer to the member function of class **Type** to be converted to a function object.

Return Value

A **const** or **non_const** function object of type **mem_fun_ref_t** or **mem_fun1_ref_t**.

Example

```
// functional_mem_fun_ref.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

class NumVals
   {
   int val;
   public:
   NumVals ( ) { val = 0; }
   NumVals ( int j ) { val = j; }

   bool display ( ) { cout << val << " "; return true; }
   bool isEven ( ) { return ( bool )  !( val %2 ); }
   bool isPrime ( )
   {
      for ( int i = 2;  i <= ( val / 2 ); i++ )
         if ( !( val % i ) ) return false;
      return true;
   }
};

int main( )
{
   vector <NumVals> v1 ( 13 ), v2 ( 13 );
```

```
    vector <NumVals>::iterator v1_Iter, v2_Iter;
    int i, k;

    for ( i = 0; i < 13; i++ ) v1 [ i ] = NumVals ( i+1 );
    for ( k = 0; k < 13; k++ ) v2 [ k ] = NumVals ( k+1 );

    cout << "The original values stored in v1 are: " ;
    for_each( v1.begin( ), v1.end( ),
    mem_fun_ref ( &NumVals::display ) );
    cout << endl;

    // Use of mem_fun_ref calling member function through a reference
    // remove the primes in the vector using isPrime ( )
    v1_Iter = remove_if ( v1.begin( ),  v1.end( ),
       mem_fun_ref ( &NumVals::isPrime ) );
    cout << "With the primes removed, the remaining values in v1 are: " ;
    for_each( v1.begin( ), v1_Iter,
    mem_fun_ref ( &NumVals::display ) );
    cout << endl;

    cout << "The original values stored in v2 are: " ;
    for_each( v2.begin( ), v2.end( ),
    mem_fun_ref ( &NumVals::display ) );
    cout << endl;

    // Use of mem_fun_ref calling member function through a reference
    // remove the even numbers in the vector v2 using isEven ( )
    v2_Iter = remove_if ( v2.begin( ),  v2.end( ),
       mem_fun_ref ( &NumVals::isEven ) );
    cout << "With the even numbers removed, the remaining values are: " ;
    for_each( v2.begin( ),  v2_Iter,
    mem_fun_ref ( &NumVals::display ) );
    cout << endl;
  }
```

Output

```
The original values stored in v1 are: 1 2 3 4 5 6 7 8 9 10 11 12 13
With the primes removed, the remaining values in v1 are: 4 6 8 9 10 12
The original values stored in v2 are: 1 2 3 4 5 6 7 8 9 10 11 12 13
With the even numbers removed, the remaining values are: 1 3 5 7 9 11 13
```

Requirements

Header: <functional>

See Also

Reference

Standard Template Library

Other Resources

<functional> Members

not1

Returns the complement of a unary predicate.

```
template<class UnaryPredicate>
   binary_negate<UnaryPredicate> not1(
      const UnaryPredicate& _Pred
   );
```

Parameters

_Pred

The unary predicate to be negated.

Return Value

A unary predicate that is the negation of the unary predicate modified.

Remarks

If a **unary_negate** is constructed from a unary predicate **Pred**(*x*), then it returns **! Pred**(*x*).

Example

```
// functional_not1.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 7; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( ";
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    vector<int>::iterator::difference_type result1;
    // Count the elements greater than 10
    result1 = count_if(v1.begin(), v1.end(), bind2nd(greater<int>(), 10));
    cout << "The number of elements in v1 greater than 10 is: "
         << result1 << "." << endl;

    vector<int>::iterator::difference_type result2;
    // Use the negator to count the elements less than or equal to 10
    result2 = count_if(v1.begin(), v1.end(),
        not1(bind2nd(greater<int>(), 10)));

    cout << "The number of elements in v1 not greater than 10 is: "
         << result2 << "." << endl;
}
```

Output

```
The vector v1 = ( 0 5 10 15 20 25 30 35 )
The number of elements in v1 greater than 10 is: 5.
The number of elements in v1 not greater than 10 is: 3.
```

Requirements

Header: <functional>

See Also

Reference

Standard Template Library

Other Resources

<functional> Members

not2

Returns the complement of a binary predicate.

```
template<class BinaryPredicate>
   binary_negate<BinaryPredicate> not2(
      const BinaryPredicate& _Func
   );
```

Parameters

_Func

The binary predicate to be negated.

Return Value

A binary predicate that is the negation of the binary predicate modified.

Remarks

If a **binary_negate** is constructed from a binary predicate **BinPred**(*x*, *y*), then it returns ! **BinPred**(*x*, *y*).

Example

```
// functional_not2.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <cstdlib>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1;

   int i;
   v1.push_back( 6262 );
   v1.push_back( 6262 );
   for ( i = 0 ; i < 5 ; i++ )
   {
      v1.push_back( rand( ) );
   }

   cout << "Original vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // To sort in ascending order,
   // use default binary predicate less<int>( )
   sort( v1.begin( ), v1.end( ) );
   cout << "Sorted vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // To sort in descending order,
   // use the binary_negate helper function not2
   sort( v1.begin( ), v1.end( ), not2(less<int>( ) ) );
   cout << "Resorted vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
```

```
    cout << ")" << endl;
}
```

Output

```
Original vector v1 = ( 6262 6262 41 18467 6334 26500 19169 )
Sorted vector v1 = ( 41 6262 6262 6334 18467 19169 26500 )
Resorted vector v1 = ( 26500 19169 18467 6334 6262 6262 41 )
```

Requirements

Header: <functional>

See Also

Reference

Standard Template Library

Other Resources

<functional> Members

not_equal_to

A binary predicate that tests whether a value of a specified type is not equal to another value of that type.

```
template<class Type>
   struct not_equal_to : public binary_function<Type, Type, bool>
   {
      bool operator()(
         const Type& _Left,
         const Type& _Right
      ) const;
   };
```

Parameters

_Left

The left operand of type **Type** in the equality to be tested.

_Right

The right operand of type **Type** in the equality to be tested.

Return Value

true if *_Left* does not equal *_Right* ; **false** if *_Left* equals *_Right*.

Remarks

The objects of type **Type** must be equality comparable. This requires that the **operator!=** defined on the set of objects satisfies the mathematical properties of an equivalence relation All the built-in numeric and pointer types satisfy this requirement.

Example

```
// functional_not_equal_to.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
   vector <double> v1, v2, v3 (6);
   vector <double>::iterator Iter1, Iter2, Iter3;

   int i;
   for ( i = 0 ; i <= 5 ; i+=2 )
   {
      v1.push_back( 2.0 *i );
      v1.push_back( 2.0 * i + 1.0 );
   }

   int j;
   for ( j = 0 ; j <= 5 ; j+=2 )
   {
      v2.push_back( - 2.0 * j );
      v2.push_back( 2.0 * j + 1.0 );
   }

   cout << "The vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;
```

```
    cout << "The vector v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
       cout << *Iter2 << " ";
    cout << ")" << endl;

    // Testing for the element-wise equality between v1 & v2
    transform ( v1.begin( ), v1.end( ), v2.begin( ), v3.begin ( ),
       not_equal_to<double>( ) );

    cout << "The result of the element-wise not_equal_to comparsion\n"
       << "between v1 & v2 is: ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
       cout << *Iter3 << " ";
    cout << ")" << endl;
 }
```

Output

```
The vector v1 = ( 0 1 4 5 8 9 )
The vector v2 = ( -0 1 -4 5 -8 9 )
The result of the element-wise not_equal_to comparsion
between v1 & v2 is: ( 0 0 1 0 1 0 )
```

Requirements

Header: <functional>

See Also

Reference

Standard Template Library

Other Resources

<functional> Members

pointer_to_binary_function

Converts a binary function pointer into an adaptable binary function.

```
template<class Arg1, class Arg2, class Result>
   class pointer_to_binary_function
   : public binary_function <Arg1, Arg2, Result>
   {
   public:
   explicit pointer_to_binary_function(
      Result (*_pfunc )( Arg1, Arg2 )
      );
      Result operator()(
         const Arg1 _Left,
         const Arg2 _Right
      ) const;
   };
```

Parameters

_pfunc

The binary function to be converted.

_Left

The left object that the **_pfunc* is called on.

_Right

The right object that the **_pfunc* is called on.

Return Value

The template class stores a copy of **_pfunc**. It defines its member function **operator()** as returning (***_pfunc**)(*_Left*, *_Right*).

Remarks

A binary function pointer is a function object and may be passed to any Standard Template Library algorithm that is expecting a binary function as a parameter, but it is not adaptable. To use it with an adaptor, such as binding a value to it or using it with a negator, it must be supplied with the nested types **first_argument_type**, **second_argument_type**, and **result_type** that make such an adaptation possible. The conversion by **pointer_to_binary_function** allows the function adaptors to work with binary function pointers.

Example

The constructor of **pointer_to_binary_function** is rarely used directly. See the helper function ptr_fun for an example of how to declare and use the **pointer_to_binary_function** adaptor predicate.

Requirements

Header: <functional>

See Also

Reference

Standard Template Library

Other Resources

<functional> Members

pointer_to_unary_function

Converts a unary function pointer into an adaptable unary function.

```
template<class Arg, class Result>
class pointer_to_unary_function
   : public unary_function<Arg, Result>
   {
   public:
      explicit pointer_to_unary_function(Result (
                  *_pfunc)(Arg)
      );
      Result operator()(
         const Arg _Left
      ) const;
   };
```

Parameters

_pfunc

The binary function to be converted.

_Left

The object that the **_pfunc* is called on.

Return Value

The template class stores a copy of **_pfunc**. It defines its member function **operator()** as returning (***_pfunc**)(*_Left*).

Remarks

A unary function pointer is a function object and may be passed to any Standard Template Library algorithm that is expecting a unary function as a parameter, but it is not adaptable. To use it with an adaptor, such as binding a value to it or using it with a negator, it must be supplied with the nested types **argument_type** and **result_type** that make such an adaptation possible. The conversion by **pointer_to_unary_function** allows the function adaptors to work with binary function pointers.

Example

The constructor of **pointer_to_unary_function** is rarely used directly. See the helper function ptr_fun for an example of how to declare and use the **pointer_to_unary_function** adaptor predicate.

Requirements

Header: <functional>

See Also

Reference

Standard Template Library

Other Resources

<functional> Members

ptr_fun

Helper template functions used to convert unary and binary function pointers, respectively, into unary and binary adaptable functions.

```
template<class Arg, class Result>
   pointer_to_unary_function<Arg, Result, Result (*)(Arg)>
      ptr_fun(Result (*_pfunc)(Arg));
template<class Arg1, class Arg2, class Result>
   pointer_to_binary_function<Arg1, Arg2, Result, Result (*)(Arg1, Arg2)>
      ptr_fun(Result (*_pfunc)(Arg1, Arg2));
```

Parameters

_pfunc

The unary or binary function pointer to be converted to an adaptable function.

Return Value

The first template function returns the unary function pointer_to_unary_function <**Arg**, **Result**>(*_pfunc*).

The second template function returns binary function pointer_to_binary_function <**Arg1**, **Arg2**, **Result**>(*_pfunc*).

Remarks

A function pointer is a function object and may be passed to any Standard Template Library algorithm that is expecting a function as a parameter, but it is not adaptable. To use it with an adaptor, such as binding a value to it or using it with a negator, it must be supplied with the nested types that make such an adaptation possible. The conversion of unary and binary function pointers by the **ptr_fun** helper function allows the function adaptors to work with unary and binary function pointers.

Example

```
// functional_ptr_fun.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <cstring>
#include <iostream>

int main( )
{
   using namespace std;
   vector <char*> v1;
   vector <char*>::iterator Iter1, RIter;

   v1.push_back ( "Open" );
   v1.push_back ( "up" );
   v1.push_back ( "the" );
   v1.push_back ( "pearly" );
   v1.push_back ( "gates" );

   cout << "Original sequence contains: " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << endl;

   // To search the sequence for "pearly"
   // use a pointer_to_function conversion
   RIter = find_if( v1.begin( ), v1.end( ),
      not1 ( bind2nd (ptr_fun ( strcmp ), "pearly" ) ) );

   if ( RIter != v1.end( ) )
   {
      cout << "The search for 'pearly' was successful.\n";
```

```
        cout << "The next character string is: "
             << *++RIter << "." << endl;
    }
  }
```

Output

```
Original sequence contains: Open up the pearly gates
The search for 'pearly' was successful.
The next character string is: gates.
```

Requirements

Header: <functional>

See Also

Reference

Standard Template Library

Other Resources

<functional> Members

<hash_map>

Defines the container template classes hash_map and hash_multimap and their supporting templates.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

For a list of all members of this header, see <hash_map> Members.

```
#include <hash_map>
```

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<hash_map> Members

Standard C++ Library Header Files

<hash_map> Members

Reference

Operators

Hash_map version	Hash_multimap version	Description
operator!= (hash_map)	operator!= (hash_multimap)	Tests if the hash_map or hash_multimap object on the left side of the operator is not equal to the hash_map or hash_multimap object on the right side.
operator< (hash_map)	operator< (hash_multimap)	Tests if the hash_map or hash_multimap object on the left side of the operator is less than the hash_map or hash_multimap object on the right side.
operator<= (hash_map)	operator<= (hash_multimap)	Tests if the hash_map or hash_multimap object on the left side of the operator is less than or equal to the hash_map or hash_multimap object on the right side.
operator== (hash_map)	operator== (hash_multimap)	Tests if the hash_map or hash_multimap object on the left side of the operator is equal to the hash_map or hash_multimap object on the right side.
operator> (hash_map)	operator> (hash_multimap)	Tests if the hash_map or hash_multimap object on the left side of the operator is greater than the hash_map or hash_multimap object on the right side.
operator>= (hash_map)	operator>= (hash_multimap)	Tests if the hash_map or hash_multimap object on the left side of the operator is greater than or equal to the hash_map or hash_multimap object on the right side.

Specialized Template Functions

Hash_map version	Hash_multimap version	Description
swap (hash_map)	swap (hash_multimap)	Exchanges the elements of two hash_maps or hash_multimaps.

Classes

hash_compare Class	Describes an object that can be used by any of the hash associative containers — hash_map, hash_multimap, hash_set, or hash_multiset — as a default **Traits** parameter object to order and hash the elements they contain.
value_compare Class	Provides a function object that can compare the elements of a hash_map by comparing the values of their keys to determine their relative order in the hash_map.
hash_map Class	Used for the storage and fast retrieval of data from a collection in which each element is a pair that has a sort key whose value is unique and an associated data value.
hash_multimap Class	Used for the storage and fast retrieval of data from a collection in which each element is a pair that has a sort key whose value need not be unique and an associated data value.

See Also

Reference

<hash_map>
Thread Safety in the Standard C++ Library
Standard Template Library

<hash_map> Operators

For more information about the operators in <hash_map>, see <hash_map> Members.

See Also

Reference

Standard Template Library

operator!= (hash_map)

Tests if the hash_map object on the left side of the operator is not equal to the hash_map object on the right side.

```
bool operator!=(
   const hash_map <Key, Type, Traits, Allocator>& _Left,
   const hash_map <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_map**.

_Right

An object of type **hash_map**.

Return Value

true if the hash_maps are not equal; **false** if hash_maps are equal.

Remarks

The comparison between hash_map objects is based on a pairwise comparison of their elements. Two hash_maps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_op_ne.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1, hm2, hm3;
   int i;
   typedef pair <int, int> Int_Pair;

   for ( i = 0 ; i < 3 ; i++ )
   {
      hm1.insert ( Int_Pair ( i, i ) );
      hm2.insert ( Int_Pair ( i, i * i ) );
      hm3.insert ( Int_Pair ( i, i ) );
   }

   if ( hm1 != hm2 )
      cout << "The hash_maps hm1 and hm2 are not equal." << endl;
   else
      cout << "The hash_maps hm1 and hm2 are equal." << endl;

   if ( hm1 != hm3 )
      cout << "The hash_maps hm1 and hm3 are not equal." << endl;
   else
      cout << "The hash_maps hm1 and hm3 are equal." << endl;
}
```

Output

```
The hash_maps hm1 and hm2 are not equal.
The hash_maps hm1 and hm3 are equal.
```

Requirements

Header: <hash_map>

See Also

Reference

Standard Template Library

Other Resources

<hash_map> Members

operator< (hash_map)

Tests if the hash_map object on the left side of the operator is less than the hash_map object on the right side.

```
bool operator<(
   const hash_map <Key, Type, Traits, Allocator>& _Left,
   const hash_map <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_map**.

_Right

An object of type **hash_map**.

Return Value

true if the hash_map on the left side of the operator is strictly less than the hash_map on the right side of the operator; otherwise **false**.

Remarks

The comparison between hash_map objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_op_lt.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map<int, int> hm1, hm2, hm3;
   hash_map <int, int>::iterator hm1_Iter, hm2_Iter, hm3_Iter;
   int i;
   typedef pair<int, int> Int_Pair;

   for ( i = 1 ; i < 4 ; i++ )
   {
      hm1.insert ( Int_Pair ( i , i ) );
      hm2.insert ( Int_Pair ( i , i + 1 ) );
      hm3.insert ( Int_Pair ( i + 1 , i ) );
   }

   cout  << "The elements of hash_map hm1 are:";
   for ( hm1_Iter= hm1.begin( ) ; hm1_Iter!= hm1.end( ) ; hm1_Iter++)
       cout << "( " << hm1_Iter-> first << ", " << hm1_Iter->second << " ) ";
   cout << "." << endl;

   cout  << "The elements of hash_map hm2 are:";
   for ( hm2_Iter= hm2.begin( ) ; hm2_Iter!= hm2.end( ) ; hm2_Iter++)
       cout << "( " << hm2_Iter-> first << ", " << hm2_Iter->second << " ) ";
   cout << "." << endl;

   cout  << "The elements of hash_map hm3 are:";
```

```
    for ( hm3_Iter= hm3.begin( ) ; hm3_Iter!= hm3.end( ) ; hm3_Iter++)
        cout << "( " << hm3_Iter-> first << ", " << hm3_Iter->second << " ) ";
    cout << "." << endl;


    if ( hm1 < hm2 )
       cout << "The hash_map hm1 is less than the hash_map hm2." << endl;
    else
       cout << "The hash_map hm1 is not less than the hash_map hm2." << endl;

    if ( hm1 < hm3 )
       cout << "The hash_map hm1 is less than the hash_map hm3." << endl;
    else
       cout << "The hash_map hm1 is not less than the hash_map hm3." << endl;
 }
```

Output

```
The elements of hash_map hm1 are:( 1, 1 ) ( 2, 2 ) ( 3, 3 ) .
The elements of hash_map hm2 are:( 1, 2 ) ( 2, 3 ) ( 3, 4 ) .
The elements of hash_map hm3 are:( 2, 1 ) ( 3, 2 ) ( 4, 3 ) .
The hash_map hm1 is less than the hash_map hm2.
The hash_map hm1 is less than the hash_map hm3.
```

Requirements

Header: <hash_map>

See Also

Reference

Standard Template Library

Other Resources

<hash_map> Members

operator<= (hash_map)

Tests if the hash_map object on the left side of the operator is less than or equal to the hash_map object on the right side.

```
bool operator<=(
   const hash_map <Key, Type, Traits, Allocator>& _Left,
   const hash_map <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_map**.

_Right

An object of type **hash_map**.

Return Value

true if the hash_map on the left side of the operator is less than or equal to the hash_map on the right side of the operator; otherwise **false**.

Remark

The comparison between hash_map objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_op_le.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1, hm2, hm3, hm4;
   int i;
   typedef pair <int, int> Int_Pair;

   for ( i = 1 ; i < 3 ; i++ )
   {
      hm1.insert ( Int_Pair ( i, i ) );
      hm2.insert ( Int_Pair ( i, i * i ) );
      hm3.insert ( Int_Pair ( i, i - 1 ) );
      hm4.insert ( Int_Pair ( i, i ) );
   }

   if ( hm1 <= hm2 )
      cout << "The hash_map hm1 is less than or equal to the hash_map hm2." << endl;
   else
      cout << "The hash_map hm1 is greater than the hash_map hm2." << endl;

   if ( hm1 <= hm3 )
      cout << "The hash_map hm1 is less than or equal to the hash_map hm3." << endl;
   else
      cout << "The hash_map hm1 is greater than the hash_map hm3." << endl;

   if ( hm1 <= hm4 )
```

```
        cout << "The hash_map hm1 is less than or equal to the hash_map hm4." << endl;
    else
        cout << "The hash_map hm1 is greater than the hash_map hm4." << endl;
}
```

Output

```
The hash_map hm1 is less than or equal to the hash_map hm2.
The hash_map hm1 is greater than the hash_map hm3.
The hash_map hm1 is less than or equal to the hash_map hm4.
```

Requirements

Header: <hash_map>

See Also

Reference

Standard Template Library

Other Resources

<hash_map> Members

operator== (hash_map)

Tests if the hash_map object on the left side of the operator is equal to the hash_map object on the right side.

```
bool operator==(
   const hash_map <Key, Type, Traits, Allocator>& _Left,
   const hash_map <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_map**.

_Right

An object of type **hash_map**.

Return Value

true if the hash_map on the left side of the operator is equal to the hash_map on the right side of the operator; otherwise **false**.

Remarks

The comparison between hash_map objects is based on a pairwise comparison of their elements. Two hash_maps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_op_eq.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1, hm2, hm3;
   int i;
   typedef pair <int, int> Int_Pair;

   for ( i = 0 ; i < 3 ; i++ )
   {
      hm1.insert ( Int_Pair ( i, i ) );
      hm2.insert ( Int_Pair ( i, i * i ) );
      hm3.insert ( Int_Pair ( i, i ) );
   }

   if ( hm1 == hm2 )
      cout << "The hash_maps hm1 and hm2 are equal." << endl;
   else
      cout << "The hash_maps hm1 and hm2 are not equal." << endl;

   if ( hm1 == hm3 )
      cout << "The hash_maps hm1 and hm3 are equal." << endl;
   else
      cout << "The hash_maps hm1 and hm3 are not equal." << endl;
}
```

Output

```
The hash_maps hm1 and hm2 are not equal.
The hash_maps hm1 and hm3 are equal.
```

Requirements

Header: <hash_map>

See Also

Reference

Standard Template Library

Other Resources

<hash_map> Members

operator> (hash_map)

Tests if the hash_map object on the left side of the operator is greater than the hash_map object on the right side.

```
bool operator>(
   const hash_map <Key, Type, Traits, Allocator>& _Left,
   const hash_map <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_map**.

_Right

An object of type **hash_map**.

Return Value

true if the hash_map on the left side of the operator is greater than the hash_map on the right side of the operator; otherwise **false**.

Remarks

The comparison between hash_map objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_op_gt.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1, hm2, hm3;
   int i;
   typedef pair <int, int> Int_Pair;

   for ( i = 0 ; i < 3 ; i++ )
   {
      hm1.insert ( Int_Pair ( i, i ) );
      hm2.insert ( Int_Pair ( i, i * i ) );
      hm3.insert ( Int_Pair ( i, i - 1 ) );
   }

   if ( hm1 > hm2 )
      cout << "The hash_map hm1 is greater than the hash_map hm2." << endl;
   else
      cout << "The hash_map hm1 is not greater than the hash_map hm2." << endl;

   if ( hm1 > hm3 )
      cout << "The hash_map hm1 is greater than the hash_map hm3." << endl;
   else
      cout << "The hash_map hm1 is not greater than the hash_map hm3." << endl;
}
```

Output

```
The hash_map hm1 is not greater than the hash_map hm2.
The hash_map hm1 is greater than the hash_map hm3.
```

Requirements

Header: <hash_map>

See Also

Reference

Standard Template Library

Other Resources

<hash_map> Members

operator>= (hash_map)

Tests if the hash_map object on the left side of the operator is greater than or equal to the hash_map object on the right side.

```
bool operator>=(
   const hash_map <Key, Type, Traits, Allocator>& _Left,
   const hash_map <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_map**.

_Right

An object of type **hash_map**.

Return Value

true if the hash_map on the left side of the operator is greater than or equal to the hash_map on the right side of the list; otherwise **false**.

Remark

The comparison between hash_map objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_op_ge.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1, hm2, hm3, hm4;
   int i;
   typedef pair <int, int> Int_Pair;

   for ( i = 1 ; i < 3 ; i++ )
   {
      hm1.insert ( Int_Pair ( i, i ) );
      hm2.insert ( Int_Pair ( i, i * i ) );
      hm3.insert ( Int_Pair ( i, i - 1 ) );
      hm4.insert ( Int_Pair ( i, i ) );
   }

   if ( hm1 >= hm2 )
      cout << "The hash_map hm1 is greater than or equal to the hash_map hm2." << endl;
   else
      cout << "The hash_map hm1 is less than the hash_map hm2." << endl;

   if ( hm1 >= hm3 )
      cout << "The hash_map hm1 is greater than or equal to the hash_map hm3." << endl;
   else
      cout << "The hash_map hm1 is less than the hash_map hm3." << endl;

   if ( hm1 >= hm4 )
```

```
        cout << "The hash_map hm1 is greater than or equal to the hash_map hm4." << endl;
    else
        cout << "The hash_map hm1 is less than the hash_map hm4." << endl;
}
```

Output

```
The hash_map hm1 is less than the hash_map hm2.
The hash_map hm1 is greater than or equal to the hash_map hm3.
The hash_map hm1 is greater than or equal to the hash_map hm4.
```

Requirements

Header: <hash_map>

See Also

Reference

Standard Template Library

Other Resources

<hash_map> Members

operator!= (hash_multimap)

Tests if the hash_multimap object on the left side of the operator is not equal to the hash_multimap object on the right side.

```
bool operator!=(
   const hash_multimap <Key, Type, Traits, Allocator>& _Left,
   const hash_multimap <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_multimap**.

_Right

An object of type **hash_multimap**.

Return Value

true if the hash_multimaps are not equal; **false** if hash_multimaps are equal.

Remarks

The comparison between hash_multimap objects is based on a pairwise comparison of their elements. Two hash_multimaps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_op_ne.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1, hm2, hm3;
   int i;
   typedef pair <int, int> Int_Pair;

   for ( i = 0 ; i < 3 ; i++ )
   {
      hm1.insert ( Int_Pair ( i, i ) );
      hm2.insert ( Int_Pair ( i, i * i ) );
      hm3.insert ( Int_Pair ( i, i ) );
   }

   if ( hm1 != hm2 )
      cout << "The hash_multimaps hm1 and hm2 are not equal." << endl;
   else
      cout << "The hash_multimaps hm1 and hm2 are equal." << endl;

   if ( hm1 != hm3 )
      cout << "The hash_multimaps hm1 and hm3 are not equal." << endl;
   else
      cout << "The hash_multimaps hm1 and hm3 are equal." << endl;
}
```

Output

```
The hash_multimaps hm1 and hm2 are not equal.
The hash_multimaps hm1 and hm3 are equal.
```

Requirements

Header: <hash_map>

See Also

Reference

Standard Template Library

Other Resources

<hash_map> Members

operator< (hash_multimap)

Tests if the hash_multimap object on the left side of the operator is less than the hash_multimap object on the right side.

```
bool operator<(
   const hash_multimap <Key, Type, Traits, Allocator>& _Left,
   const hash_multimap <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_multimap**.

_Right

An object of type **hash_multimap**.

Return Value

true if the hash_multimap on the left side of the operator is strictly less than the hash_multimap on the right side of the operator; otherwise **false**.

Remarks

The comparison between hash_multimap objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_op_lt.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap < int, int > hm1, hm2, hm3;
   int i;
   typedef pair < int, int > Int_Pair;

   for ( i = 1 ; i < 3 ; i++ )
   {
      hm1.insert ( Int_Pair ( i, i ) );
      hm2.insert ( Int_Pair ( i, i * i ) );
      hm3.insert ( Int_Pair ( i, i - 1 ) );
   }

   if ( hm1 < hm2 )
      cout << "The hash_multimap hm1 is less than the hash_multimap hm2." << endl;
   else
      cout << "The hash_multimap hm1 is not less than the hash_multimap hm2." << endl;

   if ( hm1 < hm3 )
      cout << "The hash_multimap hm1 is less than the hash_multimap hm3." << endl;
   else
      cout << "The hash_multimap hm1 is not less than the hash_multimap hm3." << endl;
}
```

Output

```
The hash_multimap hm1 is less than the hash_multimap hm2.
The hash_multimap hm1 is not less than the hash_multimap hm3.
```

Requirements

Header: <hash_map>

See Also

Reference

Standard Template Library

Other Resources

<hash_map> Members

operator<= (hash_multimap)

Tests if the hash_multimap object on the left side of the operator is less than or equal to the hash_multimap object on the right side.

```
bool operator<=(
   const hash_multimap <Key, Type, Traits, Allocator>& _Left,
   const hash_multimap <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_multimap**.

_Right

An object of type **hash_multimap**.

Return Value

true if the hash_multimap on the left side of the operator is less than or equal to the hash_multimap on the right side of the operator; otherwise **false**.

Remark

The comparison between hash_multimap objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_op_le.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1, hm2, hm3, hm4;
   int i;
   typedef pair <int, int> Int_Pair;

   for ( i = 1 ; i < 3 ; i++ )
   {
      hm1.insert ( Int_Pair ( i, i ) );
      hm2.insert ( Int_Pair ( i, i * i ) );
      hm3.insert ( Int_Pair ( i, i - 1 ) );
      hm4.insert ( Int_Pair ( i, i ) );
   }

   if ( hm1 <= hm2 )
      cout << "The hash_multimap hm1 is less than or equal to the hash_multimap hm2." << en
dl;
   else
      cout << "The hash_multimap hm1 is greater than the hash_multimap hm2." << endl;

   if ( hm1 <= hm3 )
      cout << "The hash_multimap hm1 is less than or equal to the hash_multimap hm3." << en
dl;
   else
```

```
      cout << "The hash_multimap hm1 is greater than the hash_multimap hm3." << endl;

   if ( hm1 <= hm4 )
      cout << "The hash_multimap hm1 is less than or equal to the hash_multimap hm4." << en
dl;
   else
      cout << "The hash_multimap hm1 is greater than the hash_multimap hm4." << endl;
}
```

Output

```
The hash_multimap hm1 is less than or equal to the hash_multimap hm2.
The hash_multimap hm1 is greater than the hash_multimap hm3.
The hash_multimap hm1 is less than or equal to the hash_multimap hm4.
```

Requirements

Header: <hash_map>

See Also

Reference

Standard Template Library

Other Resources

<hash_map> Members

operator== (hash_multimap)

Tests if the hash_multimap object on the left side of the operator is equal to the hash_multimap object on the right side.

```
bool operator==(
   const hash_multimap <Key, Type, Traits, Allocator>& _Left,
   const hash_multimap <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_multimap**.

_Right

An object of type **hash_multimap**.

Return Value

true if the hash_multimap on the left side of the operator is equal to the hash_multimap on the right side of the operator; otherwise **false**.

Remarks

The comparison between hash_multimap objects is based on a pairwise comparison of their elements. Two hash_multimaps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_op_eq.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap<int, int> hm1, hm2, hm3;
   int i;
   typedef pair<int, int> Int_Pair;

   for (i = 0; i < 3; i++)
   {
      hm1.insert(Int_Pair(i, i));
      hm2.insert(Int_Pair(i, i*i));
      hm3.insert(Int_Pair(i, i));
   }

   if ( hm1 == hm2 )
      cout << "The hash_multimaps hm1 and hm2 are equal." << endl;
   else
      cout << "The hash_multimaps hm1 and hm2 are not equal." << endl;

   if ( hm1 == hm3 )
      cout << "The hash_multimaps hm1 and hm3 are equal." << endl;
   else
      cout << "The hash_multimaps hm1 and hm3 are not equal." << endl;
}
```

Output

```
The hash_multimaps hm1 and hm2 are not equal.
The hash_multimaps hm1 and hm3 are equal.
```

Requirements

Header: <hash_map>

See Also

Reference

Standard Template Library

Other Resources

<hash_map> Members

operator> (hash_multimap)

Tests if the hash_multimap object on the left side of the operator is greater than the hash_multimap object on the right side.

```
bool operator>(
   const hash_multimap <Key, Type, Traits, Allocator>& _Left,
   const hash_multimap <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_multimap**.

_Right

An object of type **hash_multimap**.

Return Value

true if the hash_multimap on the left side of the operator is greater than the hash_multimap on the right side of the operator; otherwise **false**.

Remarks

The comparison between hash_multimap objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_op_gt.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1, hm2, hm3;
   int i;
   typedef pair <int, int> Int_Pair;

   for ( i = 0 ; i < 3 ; i++ )
   {
      hm1.insert ( Int_Pair ( i, i ) );
      hm2.insert ( Int_Pair ( i, i * i ) );
      hm3.insert ( Int_Pair ( i, i - 1 ) );
   }

   if ( hm1 > hm2 )
      cout << "The hash_multimap hm1 is greater than the hash_multimap hm2." << endl;
   else
      cout << "The hash_multimap hm1 is not greater than the hash_multimap hm2." << endl;

   if ( hm1 > hm3 )
      cout << "The hash_multimap hm1 is greater than the hash_multimap hm3." << endl;
   else
      cout << "The hash_multimap hm1 is not greater than the hash_multimap hm3." << endl;
}
```

Output

```
The hash_multimap hm1 is not greater than the hash_multimap hm2.
The hash_multimap hm1 is greater than the hash_multimap hm3.
```

Requirements

Header: <hash_map>

See Also

Reference

Standard Template Library

Other Resources

<hash_map> Members

operator>= (hash_multimap)

Tests if the hash_multimap object on the left side of the operator is greater than or equal to the hash_multimap object on the right side.

```
bool operator>=(
   const hash_multimap <Key, Type, Traits, Allocator>& _Left,
   const hash_multimap <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_multimap**.

_Right

An object of type **hash_multimap**.

Return Value

true if the hash_multimap on the left side of the operator is greater than or equal to the hash_multimap on the right side of the list; otherwise **false**.

Remark

The comparison between hash_multimap objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_op_ge.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1, hm2, hm3, hm4;
   int i;
   typedef pair <int, int> Int_Pair;

   for ( i = 1 ; i < 3 ; i++ )
   {
      hm1.insert ( Int_Pair ( i, i ) );
      hm2.insert ( Int_Pair ( i, i * i ) );
      hm3.insert ( Int_Pair ( i, i - 1 ) );
      hm4.insert ( Int_Pair ( i, i ) );
   }

   if ( hm1 >= hm2 )
      cout << "The hash_multimap hm1 is greater than or equal to the hash_multimap hm2." <<
endl;
   else
      cout << "The hash_multimap hm1 is less than the hash_multimap hm2." << endl;

   if ( hm1 >= hm3 )
      cout << "The hash_multimap hm1 is greater than or equal to the hash_multimap hm3." <<
endl;
   else
```

```
      cout << "The hash_multimap hm1 is less than the hash_multimap hm3." << endl;

   if ( hm1 >= hm4 )
      cout << "The hash_multimap hm1 is greater than or equal to the hash_multimap hm4." <<
endl;
   else
      cout << "The hash_multimap hm1 is less than the hash_multimap hm4." << endl;
}
```

Output

```
The hash_multimap hm1 is less than the hash_multimap hm2.
The hash_multimap hm1 is greater than or equal to the hash_multimap hm3.
The hash_multimap hm1 is greater than or equal to the hash_multimap hm4.
```

Requirements

Header: <hash_map>

See Also

Reference

Standard Template Library

Other Resources

<hash_map> Members

<hash_map> Specialized Template Functions

For more information about the specialized template functions in <hash_map>, see <hash_map> Members.

See Also

Reference

Standard Template Library

swap (hash_map)

Exchanges the elements of two hash_maps.

```
void swap(
   hash_map <Key, Traits, Alloctor>& _Left,
   hash_map <Key, Traits, Allocator>& _Right
);
```

Parameters

_Right

The hash_map whose elements are to be exchanged with those of the map *_Left*.

_Left

The hash_map whose elements are to be exchanged with those of the map *_Right*.

Remarks

The template function is an algorithm specialized on the container class hash_map to execute the member function *_Left*.swap(*_Right*). This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function, **template <class T> void swap(T&, T&)**, in the algorithm header file works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for the member function hash_map::swapor an example of the use of the template version of **swap**.

Requirements

Header: <hash_map>

See Also

Reference

Standard Template Library

Other Resources

<hash_map> Members

swap (hash_multimap)

Exchanges the elements of two hash_multimaps.

```
void swap(
   hash_multimap <Key, Traits, Alloctor>& _Left,
   hash_multimap <Key, Traits, Allocator>& _Right
);
```

Parameters

_Right

The hash_multimap whose elements are to be exchanged with those of the map *_Left*.

_Left

The hash_multimap whose elements are to be exchanged with those of the map *_Right*.

Remarks

The template function is an algorithm specialized on the container class hash_multimap to execute the member function *_Left*.swap(*_Right*). This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function, **template <class T> void swap(T&, T&)**, in the algorithm header file works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for the member function hash_multimap::swap for an example of the use of the template version of **swap**.

Requirements

Header: <hash_map>

See Also

Reference

Standard Template Library

Other Resources

<hash_map> Members

<hash_map> Classes

For more information about the classes in <hash_map>, see <hash_map> Members.

See Also

Reference

Standard Template Library

hash_compare Class

The template class describes an object that can be used by any of the hash associative containers — hash_map, hash_multimap, hash_set, or hash_multiset — as a default **Traits** parameter object to order and hash the elements they contain.

```
template<class Key, class Traits = less<Key> >
   class hash_compare
   {
   Traits comp;
public:
   const size_t bucket_size = 4;
   const size_t min_buckets = 8;
   hash_compare( );
   hash_compare( Traits pred );
   size_t operator( )( const Key& Key ) const;
   bool operator( )(
      const Key& _Key1,
      const Key& _Key2
   ) const;
   };
```

Remarks

Each hash associative container stores a hash traits object of type **Traits** (a template parameter). You can derive a class from a specialization of hash_compare to selectively override certain functions and objects, or you can supply your own version of this class if you meet certain minimum requirements. Specifically, for an object hash_comp of type **hash_compare<Key, Traits>**, the following behavior is required by the above containers:

- For all values *_Key* of type **Key**, the call **hash_comp**(*_Key*) serves as a hash function, which yields a distribution of values of type **size_t**. The function supplied by hash_compare returns *_Key*.
- For any value *_Key1* of type **Key** that precedes *_Key2* in the sequence and has the same hash value (value returned by the hash function), **hash_comp**(*_Key2*, *_Key1*) is false. The function must impose a total ordering on values of type **Key**. The function supplied by hash_compare returns *comp*(*_Key2*, *_Key1*), where *comp* is a stored object of type **Traits** that you can specify when you construct the object hash_comp. For the default **Traits** parameter type **less<Key>**, sort keys never decrease in value.
- The integer constant **bucket_size** specifies the mean number of elements per "bucket" (hash-table entry) that the container should try not to exceed. It must be greater than zero. The value supplied by hash_compare is 4.
- The integer constant **min_buckets** specifies the minimum number of buckets to maintain in the hash table. It must be a power of two and greater than zero. The value supplied by hash_compare is 8.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See examples for hash_map::hash_map, hash_multimap::hash_multimap, hash_set::hash_set, and hash_multiset::hash_multiset, for examples of how to declare and use hash_compare.

Requirements

Header: <hash_map>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<hash_map> Members

<hash_set> Members

Classes value_compare Class

Provides a function object that can compare the elements of a hash_map by comparing the values of their keys to determine their relative order in the hash_map.

```
class value_compare
   : public binary_function<value_type, value_type, bool>
{
public:
   bool operator( )( const value_type& _Left,
      const value_type& _Right ) const
      {
         return ( comp( _Left.first, _Right.first ) );
      }
protected:
   value_compare( key_compare c ) : comp (c) { }
   key_compare comp;
};
```

Remarks

The comparison criteria provided by value_compare between **value_types** of whole elements contained by a hash_map is induced from a comparison between the keys of the respective elements by the auxiliary class construction. The member function operator uses the object **comp** of type **key_compare** stored in the function object provided by value_compare to compare the sort-key components of two elements.

For hash_sets and hash_multisets, which are simple containers where the key values are identical to the element values, value_compare is equivalent to **key_compare**; for hash_maps and hash_multimaps they are not, because the value of the type **pair** elements is not identical to the value of the element's key.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for hash_map::value_comp for an example of how to declare and use value_compare.

Requirements

Header: <hash_map>

See Also

Reference

binary_function Class
Thread Safety in the Standard C++ Library
Standard Template Library

Other Resources

<hash_map> Members

hash_map Class

Stores and retrieves data quickly from a collection in which each element is a pair that has a sort key whose value is unique and an associated data value.

For a list of all members of this type, see hash_map Members.

```
template <
   class Key,
   class Type,
   class Traits=hash_compare<Key, less<Key> >,
      class Allocator=allocator<pair <const Key, Type> >
>
class hash_map
```

Parameters

Key

The key data type to be stored in the hash_map.

Type

The element data type to be stored in the hash_map.

Traits

The type which includes two function objects, one of class compare able to compare two element values as sort keys to determine their relative order and a hash function that is a unary predicate mapping key values of the elements to unsigned integers of type **size_t**. This argument is optional, and **hash_compare<***Key***, less<***Key***> >** is the default value.

Allocator

The type that represents the stored allocator object that encapsulates details about the hash_map's allocation and deallocation of memory. This argument is optional, and the default value is **allocator<pair <const** *Key*, *Type***> >**.

Remarks

The hash_map is:

- An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value.
- Reversible, because it provides a bidirectional iterator to access its elements.
- Hashed, because its elements are grouped into buckets based on the value of a hash function applied to the key values of the elements.
- Unique in the sense that each of its elements must have a unique key.
- A pair associative container, because its element data values are distinct from its key values.
- A template class, because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

The main advantage of hashing over sorting is greater efficiency; a successful hashing performs insertions, deletions, and finds in constant average time as compared with a time proportional to the logarithm of the number of elements in the container for sorting techniques. The value of an element in a hash_map, but not its associated key value, may be changed directly. Instead, key values associated with old elements must be deleted and new key values associated with new elements inserted.

The choice of container type should be based in general on the type of searching and inserting required by the application. Hashed associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient when used with a well-designed hash function, performing them in a time that is on average constant and not dependent on the number of elements in the container. A well-designed hash function produces a uniform distribution of hashed values and minimizes the number of collisions, where a collision is said to occur when distinct key values are mapped into the same hashed value. In the worst case, with the worst possible hash function, the

number of operations is proportional to the number of elements in the sequence (linear time).

The hash_map should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. A model for this type of structure is an ordered list of uniquely occurring keywords with associated string values providing, say, definitions. If, instead, the words had more than one correct definition, so that keys were not unique, then a hash_multimap would be the container of choice. If, on the other hand, just the list of words were being stored, then a hash_set would be the correct container. If multiple occurrences of the words were allowed, then a hash_multiset would be the appropriate container structure.

The hash_map orders the sequence it controls by calling a stored hash **Traits** object of class value_compare. This stored object may be accessed by calling the member function key_comp. Such a function object must behave the same as an object of class hash_compare<*Key*, *less*<*Key*> >. Specifically, for all values *_Key* of type **Key**, the call **Traits**(*_Key*) yields a distribution of values of type **size_t**.

In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate *f*(*x*,*y*) is a function object that has two argument objects *x* and *y* and a return value of **true** or **false**. An ordering imposed on a hash_map is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects x and y are defined to be equivalent when both *f*(*x*,*y*) and *f*(*y*,*x*) are false. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

The actual order of elements in the controlled sequence depends on the hash function, the ordering function, and the current size of the hash table stored in the container object. You cannot determine the current size of the hash table, so you cannot in general predict the order of elements in the controlled sequence. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

The iterator provided by the hash_map class is a bidirectional iterator, but the class member functions insert and hash_map have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own set of requirements, and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal set of functionality, but it is enough to be able to talk meaningfully about a range of iterators *[_First, _Last)* in the context of the class member functions.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Requirements

Header: <hash_map>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<hash_map> Members

hash_map Members

hash_map Members

Reference

Constructors

hash_map	Constructs a **hash_map** that is empty or that is a copy of all or part of some other **hash_map**.

Typedefs

allocator_type	A type that represents the **allocator** class for the **hash_map** object.
const_iterator	A type that provides a bidirectional iterator that can read a **const** element in the **hash_map**.
const_pointer	A type that provides a pointer to a **const** element in a **hash_map**.
const_reference	A type that provides a reference to a **const** element stored in a **hash_map** for reading and performing **const** operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any **const** element in the **hash_map**.
difference_type	A signed integer type that can be used to represent the number of elements of a **hash_map** in a range between elements pointed to by iterators.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a **hash_map**.
key_compare	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the **hash_map**.
key_type	A type describes the sort key object that constitutes each element of the **hash_map**.
mapped_type	A type that represents the data type stored in a **hash_map**.
pointer	A type that provides a pointer to an element in a **hash_map**.
reference	A type that provides a reference to an element stored in a **hash_map**.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed **hash_map**.
size_type	An unsigned integer type that can represent the number of elements in a **hash_map**.
value_type	A type that provides a function object that can compare two elements as sort keys to determine their relative order in the **hash_map**.

Member Functions

begin	Returns an iterator addressing the first element in the **hash_map**.
clear	Erases all the elements of a **hash_map**.
count	Returns the number of elements in a **hash_map** whose key matches a parameter-specified key.
empty	Tests if a **hash_map** is empty.
end	Returns an iterator that addresses the location succeeding the last element in a **hash_map**.

equal_range	Returns a pair of iterators, respectively, to the first element in a **hash_map** with a key that is greater than a specified key and to the first element in the **hash_map** with a key that is equal to or greater than the key.
erase	Removes an element or a range of elements in a **hash_map** from specified positions
find	Returns an iterator addressing the location of an element in a **hash_map** that has a key equivalent to a specified key.
get_allocator	Returns a copy of the **allocator** object used to construct the **hash_map**.
insert	Inserts an element or a range of elements into a **hash_map**.
key_comp	Returns an iterator to the first element in a **hash_map** with a key value that is equal to or greater than that of a specified key.
lower_bound	Returns an iterator to the first element in a **hash_map** with a key value that is equal to or greater than that of a specified key.
max_size	Returns the maximum length of the **hash_map**.
operator[]	Inserts an element into a **hash_map** with a specified key value.
rbegin	Returns an iterator addressing the first element in a reversed **hash_map**.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed **hash_map**.
size	Returns the number of elements in the **hash_map**.
swap	Exchanges the elements of two **hash_map**s.
upper_bound	Returns an iterator to the first element in a **hash_map** that with a key value that is greater than that of a specified key.
value_comp	Retrieves a copy of the comparison object used to order element values in a **hash_map**.

Operators

operator[]	Inserts an element into a **hash_map** with a specified key value.

See Also

Reference

hash_map Class
Thread Safety in the Standard C++ Library
Standard Template Library

hash_map Typedefs

For information about the typedefs in the hash_map class, see hash_map Members.

See Also

Reference

Standard Template Library

hash_map::allocator_type

A type that represents the allocator class for the hash_map object.

```
typedef Allocator allocator_type
```

Remark

allocator_type is a synonym for the **Allocator** template parameter.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See example for get_allocator for an example using **allocator_type**.

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::const_iterator

A type that provides a bidirectional iterator that can read a **const** element in the hash_map.

```
typedef implementation-defined const_iterator;
```

Remarks

A type **const_iterator** cannot be used to modify the value of an element.

The **const_iterator** defined by hash_map points to elements that are objects of value_type, that is of type **pair**<**const Key, Type**>, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a **const_iterator** *clter* pointing to an element in a hash_map, use the **->** operator.

To access the value of the key for the element, use *clter* **-> first**, which is equivalent to (**clter*)**.first**. To access the value of the mapped datum for the element, use *clter* **-> second**, which is equivalent to (**clter*)**.second**.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See example for begin for an example using **const_iterator**.

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::const_pointer

A type that provides a pointer to a **const** element in a hash_map.

```
typedef typename Allocator::const_pointer const_pointer;
```

Remarks

A type **const_pointer** cannot be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a hash_map object.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::const_reference

A type that provides a reference to a **const** element stored in a hash_map for reading and performing **const** operations.

```
typedef typename Allocator::const_reference const_reference;
```

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_const_ref.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map<int, int> hm1;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );

   // Declare and initialize a const_reference &Ref1
   // to the key of the first element
   const int &Ref1 = ( hm1.begin( ) -> first );

   // The following line would cause an error because the
   // non-const_reference cannot be used to access the key
   // int &Ref1 = ( hm1.begin( ) -> first );

   cout << "The key of the first element in the hash_map is "
        << Ref1 << "." << endl;

   // Declare and initialize a reference &Ref2
   // to the data value of the first element
   int &Ref2 = ( hm1.begin( ) -> second );

   cout << "The data value of the first element in the hash_map is "
        << Ref2 << "." << endl;
}
```

Output

```
The key of the first element in the hash_map is 1.
The data value of the first element in the hash_map is 10.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::const_reverse_iterator

A type that provides a bidirectional iterator that can read any **const** element in the hash_map.

```
typedef reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type **const_reverse_iterator** cannot modify the value of an element and is use to iterate through the hash_map in reverse.

The **const_reverse_iterator** defined by hash_map points to elements that are objects of value_type, that is of type **pair**<**const Key, Type**>, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a **const_reverse_iterator** *crIter* pointing to an element in a hash_map, use the **->** operator.

To access the value of the key for the element, use *crIter* -> **first**, which is equivalent to (**crIter*)**.first**. To access the value of the mapped datum for the element, use *crIter* -> **second**, which is equivalent to (**crIter*).**first**.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for rend for an example of how to declare and use the **const_reverse_iterator**.

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::difference_type

A signed integer type that can be used to represent the number of elements of a hash_map in a range between elements pointed to by iterators.

```
typedef implementation-defined difference_type;
```

Remark

The **difference_type** is the type returned when subtracting or incrementing through iterators of the container. The **difference_type** is typically used to represent the number of elements in the range *[_First, _Last)* between the iterators *_First* and *_Last*, includes the element pointed to by *_First* and the range of elements up to, but not including, the element pointed to by *_Last*.

Note that although **difference_type** is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as set, subtraction between iterators is only supported by random-access iterators provided by a random access container, such as vector.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_diff_type.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <iostream>
#include <hash_map>
#include <algorithm>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 3, 20 ) );

   // The following won't insert, because map keys are unique
   hm1.insert ( Int_Pair ( 2, 30 ) );

   hash_map <int, int>::iterator hm1_Iter, hm1_bIter, hm1_eIter;
   hm1_bIter = hm1.begin( );
   hm1_eIter = hm1.end( );

   // Count the number of elements in a hash_map
   hash_map <int, int>::difference_type  df_count = 0;
   hm1_Iter = hm1.begin( );
   while ( hm1_Iter != hm1_eIter)
   {
      df_count++;
      hm1_Iter++;
   }

   cout << "The number of elements in the hash_map hm1 is: "
        << df_count << "." << endl;

   cout  << "The keys of the mapped elements are:";
   for ( hm1_Iter= hm1.begin( ) ; hm1_Iter!= hm1.end( ) ;
         hm1_Iter++)
      cout << " " << hm1_Iter-> first;
```

```
    cout << "." << endl;

    cout  << "The values of the mapped elements are:";
    for ( hm1_Iter= hm1.begin( ) ; hm1_Iter!= hm1.end( ) ;
          hm1_Iter++)
       cout << " " << hm1_Iter-> second;
    cout << "." << endl;
  }
```

Output

```
The number of elements in the hash_map hm1 is: 3.
The keys of the mapped elements are: 1 2 3.
The values of the mapped elements are: 10 20 20.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::iterator

A type that provides a bidirectional iterator that can read or modify any element in a hash_map.

```
typedef implementation-defined iterator;
```

Remarks

The **iterator** defined by hash_map points to elements that are objects of value_type, that is of type **pair<const Key, Type>,** whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference an **iterator** *Iter* pointing to an element in a multimap, use the **->** operator.

To access the value of the key for the element, use *Iter* -> **first**, which is equivalent to (**Iter*).**first**. To access the value of the mapped datum for the element, use *Iter* -> **second**, which is equivalent to (**Iter*).**second**.

A type **iterator** can be used to modify the value of an element.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See example for begin for an example of how to declare and use the **iterator**.

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::key_comp

Retrieves a copy of the comparison object used to order keys in a hash_map.

```
key_compare key_comp( ) const;
```

Return Value

Returns the function object that a hash_map uses to order its elements.

Remarks

The stored object defines the member function

bool operator(**const Key&** *_Left*, **const Key&** *_Right*);

that returns **true** if *_Left* precedes and is not equal to *_Right* in the sort order.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_key_comp.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;

   hash_map <int, int, hash_compare<int, less<int> > > hm1;
   hash_map <int, int, hash_compare<int, less<int> > >::key_compare
      kc1 = hm1.key_comp( ) ;

   // Operator stored in kc1 tests order & returns bool value
   bool result1 = kc1( 2, 3 ) ;
   if( result1 == true )
   {
      cout << "kc1( 2,3 ) returns value of true,"
           << "\n where kc1 is the function object of hm1"
           << " of type key_compare." << endl;
   }
   else
   {
      cout << "kc1( 2,3 ) returns value of false"
           << "\n where kc1 is the function object of hm1"
           << " of type key_compare." << endl;
   }

   hash_map <int, int, hash_compare<int, greater<int> > > hm2;
   hash_map <int, int, hash_compare<int, greater<int> > >
      ::key_compare kc2 = hm2.key_comp( );

   // Operator stored in kc2 tests order & returns bool value
   bool result2 = kc2( 2, 3 ) ;
   if( result2 == true )
   {
      cout << "kc2( 2,3 ) returns value of true,"
           << "\n where kc2 is the function object of hm2"
           << " of type key_compare." << endl;
   }
   else
```

```
    {
       cout << "kc2( 2,3 ) returns value of false,"
            << "\n where kc2 is the function object of hm2"
            << " of type key_compare." << endl;
    }
  }
```

Output

```
kc1( 2,3 ) returns value of true,
 where kc1 is the function object of hm1 of type key_compare.
kc2( 2,3 ) returns value of false,
 where kc2 is the function object of hm2 of type key_compare.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::key_type

A type describes the sort key object that constitutes each element of the hash_map.

```
typedef Key key_type;
```

Remarks

key_type is a synonym for the template parameter **Key**.

For more information on **Key**, see the Remarks section of the hash_map Class topic.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See example for value_type for an example of how to declare and use **key_type**.

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::mapped_type

A type that represents the data type stored in a hash_map.

```
typedef Type mapped_type;
```

Remarks

The type **mapped_type** is a synonym for the template parameter **Type**.

For more information on **Type** see the hash_map Class topic.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See example for value_type for an example of how to declare and use **key_type**.

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::pointer

A type that provides a pointer to an element in a hash_map.

```
typedef typename Allocator::pointer pointer;
```

Remarks

A type **pointer** can be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a hash_map object.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::reference

A type that provides a reference to an element stored in a hash_map.

```
typedef typename Allocator::reference reference;
```

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_reference.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );

   // Declare and initialize a const_reference &Ref1
   // to the key of the first element
   const int &Ref1 = ( hm1.begin( ) -> first );

   // The following line would cause an error as the
   // non-const_reference cannot be used to access the key
   // int &Ref1 = ( hm1.begin( ) -> first );

   cout << "The key of first element in the hash_map is "
        << Ref1 << "." << endl;

   // Declare and initialize a reference &Ref2
   // to the data value of the first element
   int &Ref2 = ( hm1.begin( ) -> second );

   cout << "The data value of first element in the hash_map is "
        << Ref2 << "." << endl;

   // The non-const_reference can be used to modify the
   // data value of the first element
   Ref2 = Ref2 + 5;
   cout << "The modified data value of first element is "
        << Ref2 << "." << endl;
}
```

Output

```
The key of first element in the hash_map is 1.
The data value of first element in the hash_map is 10.
The modified data value of first element is 15.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::reverse_iterator

A type that provides a bidirectional iterator that can read or modify an element in a reversed hash_map.

```
typedef reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type **reverse_iterator** cannot modify the value of an element and is use to iterate through the hash_map in reverse.

The **reverse_iterator** defined by hash_map points to elements that are objects of value_type, that is of type **pair<const Key, Type>**, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a **reverse_iterator** *rIter* pointing to an element in a hash_map, use the -> operator.

To access the value of the key for the element, use *rIter* -> **first**, which is equivalent to (**rIter*).**first**. To access the value of the mapped datum for the element, use *rIter* -> **second**, which is equivalent to (**rIter*).**first**.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See example for rbegin for an example of how to declare and use **reverse_iterator**.

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::size_type

An unsigned integer type that can represent the number of elements in a hash_map.

```
typedef implementation defined size_type;
```

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See example for size for an example of how to declare and use **size_type**

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::value_type

A type that represents the type of object stored in a hash_map.

```
typedef pair<const Key, Type> value_type;
```

Remarks

value_type is declared to be **pair** <**const** key_type, mapped_type> and not **pair <key_type, mapped_type>** because the keys of an associative container may not be changed using a nonconstant iterator or reference.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_value_type.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   typedef pair <const int, int> cInt2Int;
   hash_map <int, int> hm1;
   hash_map <int, int> :: key_type key1;
   hash_map <int, int> :: mapped_type mapped1;
   hash_map <int, int> :: value_type value1;
   hash_map <int, int> :: iterator pIter;

   // value_type can be used to pass the correct type
   // explicitely to avoid implicit type conversion
   hm1.insert ( hash_map <int, int> :: value_type ( 1, 10 ) );

   // Compare other ways to insert objects into a hash_map
   hm1.insert ( cInt2Int ( 2, 20 ) );
   hm1[ 3 ] = 30;

   // Initializing key1 and mapped1
   key1 = ( hm1.begin( ) -> first );
   mapped1 = ( hm1.begin( ) -> second );

   cout << "The key of first element in the hash_map is "
        << key1 << "." << endl;

   cout << "The data value of first element in the hash_map is "
        << mapped1 << "." << endl;

   // The following line would cause an error because
   // the value_type is not assignable
   // value1 = cInt2Int ( 4, 40 );

   cout  << "The keys of the mapped elements are:";
   for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
      cout << " " << pIter -> first;
   cout << "." << endl;

   cout  << "The values of the mapped elements are:";
   for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
      cout << " " << pIter -> second;
   cout << "." << endl;
```

```
}
```

Output

```
The key of first element in the hash_map is 1.
The data value of first element in the hash_map is 10.
The keys of the mapped elements are: 1 2 3.
The values of the mapped elements are: 10 20 30.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map Member Functions

For information about the functions in the hash_map class, see hash_map Members.

See Also

Reference

Standard Template Library

hash_map::begin

Returns an iterator addressing the first element in the hash_map.

```
const_iterator begin( ) const;
iterator begin( );
```

Return Value

A bidirectional iterator addressing the first element in the hash_map or the location succeeding an empty hash_map.

Remark

If the return value of **begin** is assigned to a **const_iterator**, the elements in the hash_map object cannot be modified. If the return value of **begin** is assigned to an **iterator**, the elements in the hash_map object can be modified.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_begin.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;

   hash_map <int, int> :: iterator hm1_Iter;
   hash_map <int, int> :: const_iterator hm1_cIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 0, 0 ) );
   hm1.insert ( Int_Pair ( 1, 1 ) );
   hm1.insert ( Int_Pair ( 2, 4 ) );

   hm1_cIter = hm1.begin ( );
   cout << "The first element of hm1 is "
        << hm1_cIter -> first << "." << endl;

   hm1_Iter = hm1.begin ( );
   hm1.erase ( hm1_Iter );

   // The following 2 lines would err because the iterator is const
   // hm1_cIter = hm1.begin ( );
   // hm1.erase ( hm1_cIter );

   hm1_cIter = hm1.begin( );
   cout << "The first element of hm1 is now "
        << hm1_cIter -> first << "." << endl;
}
```

Output

```
The first element of hm1 is 0.
The first element of hm1 is now 1.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::clear

Erases all the elements of a hash_map.

```
void clear( );
```

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// hash_map_clear.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map<int, int> hm1;
    hash_map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(1, 1));
    hm1.insert(Int_Pair(2, 4));

    i = hm1.size();
    cout << "The size of the hash_map is initially "
         << i << "." << endl;

    hm1.clear();
    i = hm1.size();
    cout << "The size of the hash_map after clearing is "
         << i << "." << endl;
}
```

Output

```
The size of the hash_map is initially 2.
The size of the hash_map after clearing is 0.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::count

Returns the number of elements in a hash_map whose key matches a parameter-specified key.

```
size_type count(
   const Key& _Key
) const;
```

Parameters

_Key

The key value of the elements to be matched from the hash_map.

Return Value

1 if the hash_map contains an element whose sort key matches the parameter key; 0 if the hash_map doesn't contain an element with a matching key.

Remarks

The member function returns the number of elements *x* in the range

[**lower_bound** (*_Key*), **upper_bound** (*_Key*))

which is 0 or 1 in the case of hash_map, which is a unique associative container.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// hash_map_count.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_map<int, int> hm1;
    hash_map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair (1, 1));
    hm1.insert(Int_Pair (2, 1));
    hm1.insert(Int_Pair (1, 4));
    hm1.insert(Int_Pair (2, 1));

    // Keys must be unique in hash_map, so duplicates are ignored
    i = hm1.count(1);
    cout << "The number of elements in hm1 with a sort key of 1 is: "
         << i << "." << endl;

    i = hm1.count(2);
    cout << "The number of elements in hm1 with a sort key of 2 is: "
         << i << "." << endl;

    i = hm1.count(3);
    cout << "The number of elements in hm1 with a sort key of 3 is: "
         << i << "." << endl;
}
```

Output

```
The number of elements in hm1 with a sort key of 1 is: 1.
The number of elements in hm1 with a sort key of 2 is: 1.
The number of elements in hm1 with a sort key of 3 is: 0.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::empty

Tests if a hash_map is empty.

```
bool empty( ) const;
```

Return Value

true if the hash_map is empty; **false** if the hash_map is nonempty.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_empty.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1, hm2;

   typedef pair <int, int> Int_Pair;
   hm1.insert ( Int_Pair ( 1, 1 ) );

   if ( hm1.empty( ) )
      cout << "The hash_map hm1 is empty." << endl;
   else
      cout << "The hash_map hm1 is not empty." << endl;

   if ( hm2.empty( ) )
      cout << "The hash_map hm2 is empty." << endl;
   else
      cout << "The hash_map hm2 is not empty." << endl;
}
```

Output

```
The hash_map hm1 is not empty.
The hash_map hm2 is empty.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::end

Returns an iterator that addresses the location succeeding the last element in a hash_map.

```
const_iterator end( ) const;
iterator end( );
```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a hash_map. If the hash_map is empty, then hash_map::end == hash_map::begin.

Remarks

end is used to test whether an iterator has reached the end of its hash_map.

The value returned by **end** should not be dereferenced.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_end.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;

   hash_map <int, int> :: iterator hm1_Iter;
   hash_map <int, int> :: const_iterator hm1_cIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_cIter = hm1.end( );
   hm1_cIter--;
   cout << "The value of last element of hm1 is "
        << hm1_cIter -> second << "." << endl;

   hm1_Iter = hm1.end( );
   hm1_Iter--;
   hm1.erase ( hm1_Iter );

   // The following 2 lines would err because the iterator is const
   // hm1_cIter = hm1.end ( );
   // hm1_cIter--;
   // hm1.erase ( hm1_cIter );

   hm1_cIter = hm1.end( );
   hm1_cIter--;
   cout << "The value of last element of hm1 is now "
        << hm1_cIter -> second << "." << endl;
}
```

Output

```
The value of last element of hm1 is 30.
The value of last element of hm1 is now 20.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::equal_range

Returns a pair of iterators respectively to the first element in a hash_map with a key that is greater than a specified key and to the first element in the hash_map with a key that is equal to or greater than the key.

```
pair <const_iterator, const_iterator> equal_range (
   const Key& _Key
) const;
pair <iterator, iterator> equal_range (
   const Key& _Key
);
```

Parameters

_Key

The argument key value to be compared with the sort key of an element from the hash_map being searched.

Return Value

A pair of iterators such that the first is the lower_bound of the key and the second is the upper_bound of the key.

To access the first iterator of a pair *pr* returned by the member function, use *pr*.**first** and to dereference the lower bound iterator, use *(*pr*.**first**). To access the second iterator of a pair *pr* returned by the member function, use *pr*.**second** and to dereference the upper bound iterator, use *(*pr*.**second**).

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_equal_range.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   typedef hash_map <int, int> IntMap;
   IntMap hm1;
   hash_map <int, int> :: const_iterator hm1_RcIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   pair <IntMap::const_iterator, IntMap::const_iterator> p1, p2;
   p1 = hm1.equal_range( 2 );

   cout << "The lower bound of the element with "
        << "a key of 2 in the hash_map hm1 is: "
        << p1.first -> second << "." << endl;

   cout << "The upper bound of the element with "
        << "a key of 2 in the hash_map hm1 is: "
        << p1.second -> second << "." << endl;

   // Compare the upper_bound called directly
   hm1_RcIter = hm1.upper_bound( 2 );
```

```
    cout << "A direct call of upper_bound( 2 ) gives "
         << hm1_RcIter -> second << "," << endl
         << " matching the 2nd element of the pair"
         << " returned by equal_range( 2 )." << endl;

    p2 = hm1.equal_range( 4 );

    // If no match is found for the key,
    // both elements of the pair return end( )
    if ( ( p2.first == hm1.end( ) ) && ( p2.second == hm1.end( ) ) )
       cout << "The hash_map hm1 doesn't have an element "
              << "with a key less than 40." << endl;
    else
       cout << "The element of hash_map hm1 with a key >= 40 is: "
            << p1.first -> first << "." << endl;
 }
```

Output

```
The lower bound of the element with a key of 2 in the hash_map hm1 is: 20.
The upper bound of the element with a key of 2 in the hash_map hm1 is: 30.
A direct call of upper_bound( 2 ) gives 30,
 matching the 2nd element of the pair returned by equal_range( 2 ).
The hash_map hm1 doesn't have an element with a key less than 40.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::erase

Removes an element or a range of elements in a hash_map from specified positions or removes elements that match a specified key.

```
iterator erase(
   iterator _Where
);
iterator erase(
   iterator _First,
   iterator _Last
);
size_type erase(
   const key_type& _Key
);
```

Parameters

_Where

Position of the element to be removed from the hash_map.

_First

Position of the first element removed from the hash_map.

_Last

Position just beyond the last element removed from the hash_map.

_Key

The key value of the elements to be removed from the hash_map.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the hash_map if no such element exists.

For the third member function, returns the number of elements that have been removed from the hash_map.

Remarks

The member functions never throw an exception.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// hash_map_erase.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_map<int, int> hm1, hm2, hm3;
    hash_map<int, int> :: iterator pIter, Iter1, Iter2;
    int i;
    hash_map<int, int>::size_type n;
    typedef pair<int, int> Int_Pair;
```

```
    for (i = 1; i < 5; i++)
    {
        hm1.insert(Int_Pair (i, i));
        hm2.insert(Int_Pair (i, i*i));
        hm3.insert(Int_Pair (i, i-1));
    }

    // The 1st member function removes an element at a given position
    Iter1 = ++hm1.begin();
    hm1.erase(Iter1);

    cout << "After the 2nd element is deleted, the hash_map hm1 is:";
    for (pIter = hm1.begin(); pIter != hm1.end(); pIter++)
        cout << " " << pIter -> second;
    cout << "." << endl;

    // The 2nd member function removes elements
    // in the range [_First, _Last)
    Iter1 = ++hm2.begin();
    Iter2 = --hm2.end();
    hm2.erase(Iter1, Iter2);

    cout << "After the middle two elements are deleted, "
         << "the hash_map hm2 is:";
    for (pIter = hm2.begin(); pIter != hm2.end(); pIter++)
        cout << " " << pIter -> second;
    cout << "." << endl;

    // The 3rd member function removes elements with a given _Key
    n = hm3.erase(2);

    cout << "After the element with a key of 2 is deleted,\n"
         << "the hash_map hm3 is:";
    for (pIter = hm3.begin(); pIter != hm3.end(); pIter++)
        cout << " " << pIter -> second;
    cout << "." << endl;

    // The 3rd member function returns the number of elements removed
    cout << "The number of elements removed from hm3 is: "
         << n << "." << endl;

    // The dereferenced iterator can also be used to specify a key
    Iter1 = ++hm3.begin();
    hm3.erase(Iter1);

    cout << "After another element with a key equal to that"
         << endl;
    cout  << "of the 2nd element is deleted, "
          << "the hash_map hm3 is:";
    for (pIter = hm3.begin(); pIter != hm3.end(); pIter++)
        cout << " " << pIter -> second;
    cout << "." << endl;
}
```

Output

```
After the 2nd element is deleted, the hash_map hm1 is: 1 3 4.
After the middle two elements are deleted, the hash_map hm2 is: 1 16.
After the element with a key of 2 is deleted,
the hash_map hm3 is: 0 2 3.
The number of elements removed from hm3 is: 1.
After another element with a key equal to that
of the 2nd element is deleted, the hash_map hm3 is: 0 3.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::find

Returns an iterator addressing the location of an element in a hash_map that has a key equivalent to a specified key.

```
iterator find(
   const Key& _Key
);
const_iterator find(
   const Key& _Key
) const;
```

Parameters

_Key

The key value to be matched by the sort key of an element from the hash_map being searched.

Return Value

An iterator that addresses the location of an element with a specified key, or the location succeeding the last element in the hash_map if no match is found for the key.

Remarks

find returns an iterator that addresses an element in the hash_map whose sort key is equivalent to the argument key under a binary predicate that induces an ordering based on a less than comparability relation.

If the return value of **find** is assigned to a const_iterator, the hash_map object cannot be modified. If the return value of **find** is assigned to an iterator, the hash_map object can be modified

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_find.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;
   hash_map <int, int> :: const_iterator hm1_AcIter, hm1_RcIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_RcIter = hm1.find( 2 );
   cout << "The element of hash_map hm1 with a key of 2 is: "
        << hm1_RcIter -> second << "." << endl;

   // If no match is found for the key, end( ) is returned
   hm1_RcIter = hm1.find( 4 );

   if ( hm1_RcIter == hm1.end( ) )
      cout << "The hash_map hm1 doesn't have an element "
           << "with a key of 4." << endl;
   else
      cout << "The element of hash_map hm1 with a key of 4 is: "
           << hm1_RcIter -> second << "." << endl;
```

```
    // The element at a specific location in the hash_map can be found
    // using a dereferenced iterator addressing the location
    hm1_AcIter = hm1.end( );
    hm1_AcIter--;
    hm1_RcIter = hm1.find( hm1_AcIter -> first );
    cout << "The element of hm1 with a key matching "
         << "that of the last element is: "
         << hm1_RcIter -> second << "." << endl;
}
```

Output

```
The element of hash_map hm1 with a key of 2 is: 20.
The hash_map hm1 doesn't have an element with a key of 4.
The element of hm1 with a key matching that of the last element is: 30.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::get_allocator

Returns a copy of the allocator object used to construct the hash_map.

```
Allocator get_allocator( ) const;
```

Return Value

The allocator used by the hash_map.

Remarks

Allocators for the hash_map class specify how the class manages storage. The default allocators supplied with STL container classes is sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_get_allocator.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int>::allocator_type hm1_Alloc;
   hash_map <int, int>::allocator_type hm2_Alloc;
   hash_map <int, double>::allocator_type hm3_Alloc;
   hash_map <int, int>::allocator_type hm4_Alloc;

   // The following lines declare objects
   // that use the default allocator.
   hash_map <int, int> hm1;
   hash_map <int, int> hm2;
   hash_map <int, double> hm3;

   hm1_Alloc = hm1.get_allocator( );
   hm2_Alloc = hm2.get_allocator( );
   hm3_Alloc = hm3.get_allocator( );

   cout << "The number of integers that can be allocated"
        << endl << "before free memory is exhausted: "
        << hm2.max_size( ) << "." << endl;

   cout << "The number of doubles that can be allocated"
        << endl << "before free memory is exhausted: "
        << hm3.max_size( ) <<  "." << endl;

   // The following line creates a hash_map hm4
   // with the allocator of hash_map hm1.
   hash_map <int, int> hm4( less<int>( ), hm1_Alloc );

   hm4_Alloc = hm4.get_allocator( );

   // Two allocators are interchangeable if
   // storage allocated from each can be
   // deallocated with the other
   if( hm1_Alloc == hm4_Alloc )
   {
      cout << "The allocators are interchangeable."
```

```
            << endl;
    }
    else
    {
       cout << "The allocators are not interchangeable."
            << endl;
    }
}
```

Sample Output

The following output is for x86.

```
The number of integers that can be allocated
before free memory is exhausted: 536870911.
The number of doubles that can be allocated
before free memory is exhausted: 268435455.
The allocators are interchangeable.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::hash_map

Constructs a hash_map that is empty or that is a copy of all or part of some other hash_map.

```
hash_map( );
explicit hash_map(
   const Traits& _Comp
);
explicit hash_map(
   const Traits& _Comp,
   const Allocator& _Al
);
hash_map(
   const _hash_map& _Right
);
template<class InputIterator>
   hash_map(
      InputIterator _First,
      InputIterator _Last
   );
template<class InputIterator>
   hash_map(
      InputIterator _First,
      InputIterator _Last,
      const Traits& _Comp
   );
template<class InputIterator>
   hash_map(
      InputIterator _First,
      InputIterator _Last,
      const Traits& _Comp,
      const Allocator& _Al
   );
```

Parameters

_Al

The storage allocator class to be used for this hash_map object, which defaults to **Allocator**.

_Comp

The comparison function of type **const Traits** used to order the elements in the hash_map, which defaults to **hash_compare**.

_Right

The hash_map of which the constructed map is to be a copy.

_First

The position of the first element in the range of elements to be copied.

_Last

The position of the first element beyond the range of elements to be copied.

Remarks

All constructors store a type of allocator object that manages memory storage for the hash_map and that can later be returned by calling get_allocator. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their hash_map.

All constructors store a function object of type **Traits** that is used to establish an order among the keys of the hash_map and that can later be returned by calling key_comp.

The first three constructors specify an empty initial hash_map, the second, in addition, specifying the type of comparison function (*_Comp*) to be used in establishing the order of the elements and the third explicitly specifying the allocator type (*_Al*) to be used. The keyword **explicit** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the hash_map *_Right*.

The last three constructors copy the range [*_First*, *_Last*) of a hash_map with increasing explicitness in specifying the type of comparison function of class **Traits** and allocator.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_hash_map.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <ostream>
#include <hash_map>
#include <iostream>
#include <fstream>
#include <string.h>
using namespace std;

typedef char * MyStr;

struct MyInt {
   int i;
   friend ostream& operator<<(ostream& ii, MyInt& jj);
   MyInt(int j = 0) {
      i = j;
   }
};

struct greater_str {
   bool operator()(const MyStr & x, const MyStr & y) const {
      if ( strcmp(x, y) < 0)
         return true;

      return false;
   }
};


struct less_str {
   bool operator()(const MyStr & x, const MyStr & y) const {
      if ( strcmp(x, y) > 0)
         return true;

      return false;
   }
};

ostream& operator<<(ostream& o, MyInt& j) {
   o << j.i;
   return o;
}

int main() {
   using namespace stdext;
   typedef pair <MyStr, MyInt> Int_Pair;
   hash_map <MyStr, MyInt>::iterator hm1_Iter, hm3_Iter, hm4_Iter, hm5_Iter, hm6_Iter;
   hash_map <MyStr, MyInt, hash_compare <MyStr, greater_str > >::iterator hm2_Iter;

   // Create an empty hash_map hm0 of key type integer
   hash_map <MyStr, MyInt> hm0;

   // Create an empty hash_map hm1 with the key comparison
   // function of less than, then insert 4 elements
```

```
   hash_map <MyStr, MyInt, hash_compare <MyStr, less_str > > hm1;

   hm1.insert( Int_Pair( "one",  0) );
   hm1.insert( Int_Pair( "two", 10 ) );
   hm1.insert( Int_Pair( "three", 20 ) );
   hm1.insert( Int_Pair( "four", 30 ) );
   hm1.insert( Int_Pair( "five", 40 ) );

   // Create an empty hash_map hm2 with the key comparison
   // function of geater than, then insert 2 elements
   hash_map <MyStr, MyInt, hash_compare <MyStr, greater_str > > hm2;
   hm2.insert( Int_Pair( "one", 10 ) );
   hm2.insert( Int_Pair( "two", 20 ) );

   // Create a hash_map hm3 with the
   // allocator of hash_map hm1
   hash_map <MyStr, MyInt>::allocator_type hm1_Alloc;
   hm1_Alloc = hm1.get_allocator( );
   hash_map <MyStr, MyInt, hash_compare <MyStr, less_str > > hm3( hash_compare <MyStr, less
_str > (), hm1_Alloc );
   hm3.insert( Int_Pair( "three", 30 ) );


   // Create a copy, hash_map hm4, of hash_map hm1
   hash_map <MyStr, MyInt, hash_compare <MyStr, less_str > > hm4( hm1 );

   // Create a hash_map hm5 by copying the range hm1[_First, _Last)
   hash_map <MyStr, MyInt>::const_iterator hm1_bcIter, hm1_ecIter;
   hm1_bcIter = hm1.begin();
   hm1_ecIter = hm1.begin();
   hm1_ecIter++;
   hm1_ecIter++;
   hash_map <MyStr, MyInt> hm5(hm1_bcIter, hm1_ecIter);

   // Create a hash_map hm6 by copying the range hm4[_First, _Last)
   // and with the allocator of hash_map hm2
   hash_map <MyStr, MyInt>::allocator_type hm2_Alloc;
   hm2_Alloc = hm2.get_allocator( );
   hash_map <MyStr, MyInt, hash_compare <MyStr, less_str > > hm6( hm4.begin( ), ++hm4.begin
( ), hash_compare <MyStr, less_str > ( ), hm2_Alloc);

   cout << "hm1 =";
   for ( hm1_Iter = hm1.begin(); hm1_Iter != hm1.end(); hm1_Iter++ )
      cout << " " << hm1_Iter -> second;
   cout << endl;

   cout << "hm2 =";
   for ( hm2_Iter = hm2.begin( ); hm2_Iter != hm2.end( ); hm2_Iter++ )
      cout << " " << hm2_Iter -> second;
   cout << endl;

   cout << "hm3 =";
   for ( hm3_Iter = hm3.begin( ); hm3_Iter != hm3.end( ); hm3_Iter++ )
      cout << " " << hm3_Iter -> second;
   cout << endl;

   cout << "hm4 =";
   for ( hm4_Iter = hm4.begin( ); hm4_Iter != hm4.end( ); hm4_Iter++ )
      cout << " " << hm4_Iter -> second;
   cout << endl;

   cout << "hm5 =";
   for ( hm5_Iter = hm5.begin( ); hm5_Iter != hm5.end( ); hm5_Iter++ )
      cout << " " << hm5_Iter -> second;
   cout << endl;

   cout << "hm6 =";
   for ( hm6_Iter = hm6.begin( ); hm6_Iter != hm6.end( ); hm6_Iter++ )
```

```
        cout << " " << hm6_Iter -> second;
    cout << endl;
}
```

Output

```
hm1 = 10 20 0 30 40
hm2 = 10 20
hm3 = 30
hm4 = 10 20 0 30 40
hm5 = 10 20
hm6 = 10
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::insert

Inserts an element or a range of elements into a hash_map.

```
pair <iterator, bool> insert(
   const value_type& _Val
);
iterator insert(
   iterator _Where,
   const value_type& _Val
);
template<class InputIterator>
   void insert(
      InputIterator _First,
      InputIterator _Last
   );
```

Parameters

_Val

The value of an element to be inserted into the hash_map unless the hash_map already contains that element (or, more generally, an element whose key is equivalently ordered).

_Where

A hint regarding the place to start searching for the correct point of insertion.

_First

The position of the first element to be copied from a hash_map.

_Last

The position just beyond the last element to be copied from a hash_map.

Return Value

The first **insert** member function returns a pair whose bool component returns true if an insertion was make and false if the hash_map already contained an element whose key had an equivalent value in the ordering, and whose iterator component returns the address where a new element was inserted or where the element was already located.

To access the iterator component of a pair *pr* returned by this member function, use *pr*.**first**, and to dereference it, use *(*pr*.**first**). To access the **bool** component of a pair *pr* returned by this member function, use *pr*.**second**, and to dereference it, use *(*pr*.**second**).

The second **insert** member function, the hint version, returns an iterator that points to the position where the new element was inserted into the hash_map.

Remarks

The value_type of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Insertion can occur in amortized constant time for the hint version of insert, instead of logarithmic time, if the insertion point immediately follows *_Where*.

The third member function inserts the sequence of element values into a hash_map corresponding to each element addressed by an iterator of in the range *[First, Last)* of a specified set.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_insert.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
```

```
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int>::iterator hm1_pIter, hm2_pIter;

   hash_map <int, int> hm1, hm2;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );
   hm1.insert ( Int_Pair ( 4, 40 ) );

   cout << "The original key values of hm1 =";
   for ( hm1_pIter = hm1.begin( ); hm1_pIter != hm1.end( ); hm1_pIter++ )
      cout << " " << hm1_pIter -> first;
   cout << "." << endl;

   cout << "The original hash_mapped values of hm1 =";
   for ( hm1_pIter = hm1.begin( ); hm1_pIter != hm1.end( ); hm1_pIter++ )
      cout << " " << hm1_pIter -> second;
   cout << "." << endl;

   pair< hash_map<int,int>::iterator, bool > pr;
   pr = hm1.insert ( Int_Pair ( 1, 10 ) );

   if( pr.second == true )
   {
      cout << "The element 10 was inserted in hm1 successfully."
           << endl;
   }
   else
   {
      cout << "The element 10 already exists in hm1\n with a key value of"
           << " ( (pr.first) -> first ) = " << ( pr.first ) -> first
           << "." << endl;
   }

   // The hint version of insert
   hm1.insert( --hm1.end( ), Int_Pair ( 5, 50 ) );

   cout << "After the insertions, the key values of hm1 =";
   for ( hm1_pIter = hm1.begin( ); hm1_pIter != hm1.end( ); hm1_pIter++ )
      cout << " " << hm1_pIter -> first;
   cout << "," << endl;

   cout << " and the mapped values of hm1 =";
   for ( hm1_pIter = hm1.begin( ); hm1_pIter != hm1.end( ); hm1_pIter++ )
      cout << " " << hm1_pIter -> second;
   cout << "." << endl;

   hm2.insert ( Int_Pair ( 10, 100 ) );

   // The templatized version inserting a range
   hm2.insert( ++hm1.begin( ), --hm1.end( ) );

   cout << "After the insertions, the key values of hm2 =";
   for ( hm2_pIter = hm2.begin( ); hm2_pIter != hm2.end( ); hm2_pIter++ )
      cout << " " << hm2_pIter -> first;
   cout << "," << endl;

   cout << " and the mapped values of hm2 =";
   for ( hm2_pIter = hm2.begin( ); hm2_pIter != hm2.end( ); hm2_pIter++ )
      cout << " " << hm2_pIter -> second;
```

```
    cout << "." << endl;
  }
```

Output

```
The original key values of hm1 = 1 2 3 4.
The original hash_mapped values of hm1 = 10 20 30 40.
The element 10 already exists in hm1
 with a key value of ( (pr.first) -> first ) = 1.
After the insertions, the key values of hm1 = 1 3 5 2 4,
 and the mapped values of hm1 = 10 30 50 20 40.
After the insertions, the key values of hm2 = 2 3 5 10,
 and the mapped values of hm2 = 20 30 50 100.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::key_compare

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the map.

```
typedef Traits key_compare;
```

Remarks

key_compare is a synonym for the template parameter **Traits**.

For more information on **Traits** see the hash_map Class topic.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See example for key_comp for an example of how to declare and use **key_compare**.

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::lower_bound

Returns an iterator to the first element in a hash_map with a key value that is equal to or greater than that of a specified key.

```
iterator lower_bound(
   const Key& _Key
);
const_iterator lower_bound(
   const Key& _Key
) const;
```

Parameters

_Key

The argument key value to be compared with the sort key of an element from the hash_map being searched.

Return Value

An iterator or const_iterator that addresses the location of an element in a hash_map that with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the hash_map if no match is found for the key.

If the return value of **lower_bound** is assigned to a **const_iterator**, the hash_map object cannot be modified. If the return value of **lower_bound** is assigned to a **iterator**, the hash_map object can be modified.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_lower_bound.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;
   hash_map <int, int> :: const_iterator hm1_AcIter, hm1_RcIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_RcIter = hm1.lower_bound( 2 );
   cout << "The first element of hash_map hm1 with a key of 2 is: "
        << hm1_RcIter -> second << "." << endl;

   // If no match is found for the key, end( ) is returned
   hm1_RcIter = hm1. lower_bound ( 4 );

   if ( hm1_RcIter == hm1.end( ) )
      cout << "The hash_map hm1 doesn't have an element "
           << "with a key of 4." << endl;
   else
      cout << "The element of hash_map hm1 with a key of 4 is: "
           << hm1_RcIter -> second << "." << endl;

   // An element at a specific location in the hash_map can be
```

```
    // found using a dereferenced iterator addressing the location
    hm1_AcIter = hm1.end( );
    hm1_AcIter--;
    hm1_RcIter = hm1. lower_bound ( hm1_AcIter -> first );
    cout << "The element of hm1 with a key matching "
         << "that of the last element is: "
         << hm1_RcIter -> second << "." << endl;
}
```

Output

```
The first element of hash_map hm1 with a key of 2 is: 20.
The hash_map hm1 doesn't have an element with a key of 4.
The element of hm1 with a key matching that of the last element is: 30.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::max_size

Returns the maximum length of the hash_map.

```
size_type max_size( ) const;
```

Return Value

The maximum possible length of the hash_map.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_max_size.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;
   hash_map <int, int> :: size_type i;

   i = hm1.max_size( );
   cout << "The maximum possible length "
        << "of the hash_map is " << i << "."
        << endl << "(Magnitude is machine specific.)";
}
```

Sample Output

The following output is for x86.

```
The maximum possible length of the hash_map is 536870911.
(Magnitude is machine specific.)
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::rbegin

Returns an iterator addressing the first element in a reversed hash_map.

```
const_reverse_iterator rbegin( ) const;
reverse_iterator rbegin( );
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed hash_map or addressing what had been the last element in the unreversed hash_map.

Remarks

rbegin is used with a reversed hash_map just as begin is used with a hash_map.

If the return value of **rbegin** is assigned to a const_reverse_iterator, then the hash_map object cannot be modified. If the return value of **rbegin** is assigned to a reverse_iterator, then the hash_map object can be modified.

rbegin can be used to iterate through a hash_map backwards.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_rbegin.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;

   hash_map <int, int> :: iterator hm1_Iter;
   hash_map <int, int> :: reverse_iterator hm1_rIter;
   hash_map <int, int> :: const_reverse_iterator hm1_crIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_rIter = hm1.rbegin( );
   cout << "The first element of the reversed hash_map hm1 is "
        << hm1_rIter -> first << "." << endl;

   // begin can be used to start an iteration
   // through a hash_map in a forward order
   cout << "The hash_map is: ";
   for ( hm1_Iter = hm1.begin( ) ; hm1_Iter != hm1.end( ); hm1_Iter++)
      cout << hm1_Iter -> first << " ";
      cout << "." << endl;

   // rbegin can be used to start an iteration
   // through a hash_map in a reverse order
   cout << "The reversed hash_map is: ";
   for ( hm1_rIter = hm1.rbegin( ) ; hm1_rIter != hm1.rend( ); hm1_rIter++)
      cout << hm1_rIter -> first << " ";
      cout << "." << endl;
```

```
    // A hash_map element can be erased by dereferencing to its key
    hm1_rIter = hm1.rbegin( );
    hm1.erase ( hm1_rIter -> first );

    hm1_rIter = hm1.rbegin( );
    cout << "After the erasure, the first element "
         << "in the reversed hash_map is "
         << hm1_rIter -> first << "." << endl;
}
```

Output

```
The first element of the reversed hash_map hm1 is 3.
The hash_map is: 1 2 3 .
The reversed hash_map is: 3 2 1 .
After the erasure, the first element in the reversed hash_map is 2.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::rend

Returns an iterator that addresses the location succeeding the last element in a reversed hash_map.

```
const_reverse_iterator rend( ) const;
reverse_iterator rend( );
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed hash_map (the location that had preceded the first element in the unreversed hash_map).

Remarks

rend is used with a reversed hash_map just as end is used with a hash_map.

If the return value of **rend** is assigned to a const_reverse_iterator, then the hash_map object cannot be modified. If the return value of **rend** is assigned to a reverse_iterator, then the hash_map object can be modified.

rend can be used to test to whether a reverse iterator has reached the end of its hash_map.

The value returned by **rend** should not be dereferenced.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_rend.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;

   hash_map <int, int> :: iterator hm1_Iter;
   hash_map <int, int> :: reverse_iterator hm1_rIter;
   hash_map <int, int> :: const_reverse_iterator hm1_crIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_rIter = hm1.rend( );
   hm1_rIter--;
   cout << "The last element of the reversed hash_map hm1 is "
        << hm1_rIter -> first << "." << endl;

   // begin can be used to start an iteration
   // through a hash_map in a forward order
   cout << "The hash_map is: ";
   for ( hm1_Iter = hm1.begin( ) ; hm1_Iter != hm1.end( );
   hm1_Iter++)
      cout << hm1_Iter -> first << " ";
      cout << "." << endl;

   // rbegin can be used to start an iteration
   // through a hash_map in a reverse order
   cout << "The reversed hash_map is: ";
   for ( hm1_rIter = hm1.rbegin( ) ; hm1_rIter != hm1.rend( );
```

```
        hm1_rIter++)
        cout << hm1_rIter -> first << " ";
        cout << "." << endl;

    // A hash_map element can be erased by dereferencing to its key
    hm1_rIter = --hm1.rend( );
    hm1.erase ( hm1_rIter -> first );

    hm1_rIter = hm1.rend( );
    hm1_rIter--;
    cout << "After the erasure, the last element "
         << "in the reversed hash_map is "
         << hm1_rIter -> first << "." << endl;
}
```

Output

```
The last element of the reversed hash_map hm1 is 1.
The hash_map is: 1 2 3 .
The reversed hash_map is: 3 2 1 .
After the erasure, the last element in the reversed hash_map is 2.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::size

Returns the number of elements in the hash_map.

```
size_type size( ) const;
```

Return Value

The current length of the hash_map.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// hash_map_size.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map<int, int> hm1, hm2;
    hash_map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(1, 1));
    i = hm1.size();
    cout << "The hash_map length is " << i << "." << endl;

    hm1.insert(Int_Pair(2, 4));
    i = hm1.size();
    cout << "The hash_map length is now " << i << "." << endl;
}
```

Output

```
The hash_map length is 1.
The hash_map length is now 2.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::swap

Exchanges the elements of two hash_maps.

```
void swap(
   hash_map& _Right
);
```

Parameters

_Right

The argument hash_map providing the elements to be swapped with the target hash_map.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two hash_maps whose elements are being exchanged.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_swap.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1, hm2, hm3;
   hash_map <int, int>::iterator hm1_Iter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );
   hm2.insert ( Int_Pair ( 10, 100 ) );
   hm2.insert ( Int_Pair ( 20, 200 ) );
   hm3.insert ( Int_Pair ( 30, 300 ) );

   cout << "The original hash_map hm1 is:";
   for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++ )
      cout << " " << hm1_Iter -> second;
   cout   << "." << endl;

   // This is the member function version of swap
   // hm2 is said to be the argument hash_map;
   // hm1 is said to be the target hash_map
   hm1.swap( hm2 );

   cout << "After swapping with hm2, hash_map hm1 is:";
   for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++ )
      cout << " " << hm1_Iter -> second;
   cout  << "." << endl;

   // This is the specialized template version of swap
   swap( hm1, hm3 );

   cout << "After swapping with hm3, hash_map hm1 is:";
   for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++ )
      cout << " " << hm1_Iter -> second;
```

```
    cout   << "." << endl;
}
```

Output

```
The original hash_map hm1 is: 10 20 30.
After swapping with hm2, hash_map hm1 is: 100 200.
After swapping with hm3, hash_map hm1 is: 300.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::upper_bound

Returns an iterator to the first element in a hash_map that with a key having a value that is greater than that of a specified key.

```
iterator upper_bound(
   const Key& _Key
);
const_iterator upper_bound(
   const Key& _Key
) const;
```

Parameters

_Key

The argument key value to be compared with the sort key value of an element from the hash_map being searched.

Return Value

An iterator or const_iterator that addresses the location of an element in a hash_map that with a key that is greater than the argument key, or that addresses the location succeeding the last element in the hash_map if no match is found for the key.

If the return value is assigned to a **const_iterator**, the hash_map object cannot be modified. If the return value is assigned to an **iterator**, the hash_map object can be modified.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_upper_bound.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;
   hash_map <int, int> :: const_iterator hm1_AcIter, hm1_RcIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_RcIter = hm1.upper_bound( 2 );
   cout << "The first element of hash_map hm1 with a key "
        << "greater than 2 is: "
        << hm1_RcIter -> second << "." << endl;

   // If no match is found for the key, end is returned
   hm1_RcIter = hm1. upper_bound ( 4 );

   if ( hm1_RcIter == hm1.end( ) )
      cout << "The hash_map hm1 doesn't have an element "
           << "with a key greater than 4." << endl;
   else
      cout << "The element of hash_map hm1 with a key > 4 is: "
           << hm1_RcIter -> second << "." << endl;

   // The element at a specific location in the hash_map can be found
```

```
    // using a dereferenced iterator addressing the location
    hm1_AcIter = hm1.begin( );
    hm1_RcIter = hm1. upper_bound ( hm1_AcIter -> first );
    cout << "The 1st element of hm1 with a key greater than "
         << "that\n of the initial element of hm1 is: "
         << hm1_RcIter -> second << "." << endl;
}
```

Output

```
The first element of hash_map hm1 with a key greater than 2 is: 30.
The hash_map hm1 doesn't have an element with a key greater than 4.
The 1st element of hm1 with a key greater than that
 of the initial element of hm1 is: 20.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map::value_comp

Returns a function object that determines the order of elements in a hash_map by comparing their key values.

```
value_compare value_comp( ) const;
```

Return Value

Returns the comparison function object that a hash_map uses to order its elements.

Remarks

For a hash_map *m*, if two elements *e*1*(k*1, *d*1*)* and *e*2*(k*2, *d*2*)* are objects of type value_type, where *k*1 and *k*2 are their keys of type key_type and *d*1 and *d*2 are their data of type mapped_type, then *m*.**value_comp***( )(e*1, *e*2*)* is equivalent to *m*.**key_comp***( ) (k*1, *k*2*)*. A stored object defines the member function

bool operator(**value_type&** *_Left*, **value_type&** *_Right*)**;**

which returns **true** if the key value of *_Left* precedes and is not equal to the key value of *_Right* in the sort order.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_value_comp.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;

   hash_map <int, int, hash_compare<int, less<int> > > hm1;
   hash_map <int, int, hash_compare<int, less<int> > >
   ::value_compare vc1 = hm1.value_comp( );
   pair< hash_map<int,int>::iterator, bool > pr1, pr2;

   pr1= hm1.insert ( hash_map <int, int> :: value_type ( 1, 10 ) );
   pr2= hm1.insert ( hash_map <int, int> :: value_type ( 2, 5 ) );

   if( vc1( *pr1.first, *pr2.first ) == true )
   {
      cout << "The element ( 1,10 ) precedes the element ( 2,5 )."
           << endl;
   }
   else
   {
      cout << "The element ( 1,10 ) does not precede the element ( 2,5 )."
           << endl;
   }

   if( vc1 ( *pr2.first, *pr1.first ) == true )
   {
      cout << "The element ( 2,5 ) precedes the element ( 1,10 )."
           << endl;
   }
   else
   {
      cout << "The element ( 2,5 ) does not precede the element ( 1,10 )."
           << endl;
   }
```

```
}
```

Output

```
The element ( 1,10 ) precedes the element ( 2,5 ).
The element ( 2,5 ) does not precede the element ( 1,10 ).
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_map Operators

For information about the operators in the hash_map class, see hash_map Members.

See Also

Reference

Standard Template Library

hash_map::operator[]

Inserts an element into a hash_map with a specified key value.

```
Type& operator[](
   const Key& _Key
);
```

Parameters

_Key

The key value of the element that is to be inserted.

Return Value

A reference to the data value of the inserted element.

Remarks

If the argument key value is not found, then it is inserted along with the default value of the data type.

operator[]may be used to insert elements into a hash_map m using **m[**_*Key*_**] =** *DataValue*; where *DataValue* is the value of the **mapped_type** of the element with a key value of _*Key*.

When using **operator[]** to insert elements, the returned reference does not indicate whether an insertion is changing a preexisting element or creating a new one. The member functions find and insert can be used to determine whether an element with a specified key is already present before an insertion.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_map_op_ref.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   typedef pair <const int, int> cInt2Int;
   hash_map <int, int> hm1;
   hash_map <int, int> :: iterator pIter;

   // Insert a data value of 10 with a key of 1
   // into a hash_map using the operator[] member function
   hm1[ 1 ] = 10;

   // Compare other ways to insert objects into a hash_map
   hm1.insert ( hash_map <int, int> :: value_type ( 2, 20 ) );
   hm1.insert ( cInt2Int ( 3, 30 ) );

   cout  << "The keys of the mapped elements are:";
   for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
      cout << " " << pIter -> first;
   cout << "." << endl;

   cout  << "The values of the mapped elements are:";
   for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
      cout << " " << pIter -> second;
   cout << "." << endl;

   // If the key already exists, operator[]
```

```
    // changes the value of the datum in the element
    hm1[ 2 ] = 40;

    // operator[] will also insert the value of the data
    // type's default constructor if the value is unspecified
    hm1[5];

    cout  << "The keys of the mapped elements are now:";
    for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
       cout << " " << pIter -> first;
    cout << "." << endl;

    cout  << "The values of the mapped elements are now:";
    for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
       cout << " " << pIter -> second;
    cout << "." << endl;
 }
```

Output

```
The keys of the mapped elements are: 1 2 3.
The values of the mapped elements are: 10 20 30.
The keys of the mapped elements are now: 1 2 3 5.
The values of the mapped elements are now: 10 40 30 0.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_map Class

Standard Template Library

Other Resources

hash_map Members

hash_multimap Class

The container class hash_multimap is an extension of the Standard Template Library and is used for the storage and fast retrieval of data from a collection in which each element is a pair that has a sort key whose value need not be unique and an associated data value.

For a list of all members of this type, see hash_multimap Members.

```
template <
   class Key,
   class Type,
   class Traits=hash_compare<Key, less<Key> >,
      class Allocator=allocator<pair <const Key, Type> >
>
class hash_multimap
```

Parameters

Key

The key data type to be stored in the hash_multimap.

Type

The element data type to be stored in the hash_multimap.

Traits

The type that includes two function objects, one of class **Traits** that is able to compare two element values as sort keys to determine their relative order and a hash function that is a unary predicate mapping key values of the elements to unsigned integers of type **size_t**. This argument is optional, and the **hash_compare**<*Key*, *less*<*Key*> > is the default value.

Allocator

The type that represents the stored allocator object that encapsulates details about the hash_multimap's allocation and deallocation of memory. This argument is optional, and the default value is **allocator**<**pair** <**const** *Key*, *Type*> >.

Remarks

The hash_multimap is:

- An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value.
- Reversible, because it provides a bidirectional iterator to access its elements.
- Hashed, because its elements are grouped into buckets based on the value of a hash function applied to the key values of the elements.
- Multiple, because its elements do not need to have a unique keys, so that one key value may have many element data values associated with it.
- A pair associative container, because its element values are distinct from its key values.
- A template class, because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

The main advantage of hashing over sorting is greater efficiency; a successful hashing performs insertions, deletions, and finds in constant average time as compared with a time proportional to the logarithm of the number of elements in the container for sorting techniques. The value of an element in a hash_multimap, but not its associated key value, may be changed directly. Instead, key values associated with old elements must be deleted and new key values associated with new elements inserted.

The choice of container type should be based in general on the type of searching and inserting required by the application. Hashed associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient when used with a well-designed hash function, performing them in a time that is on average constant and not dependent on the number of elements in the container. A well-designed hash function

produces a uniform distribution of hashed values and minimizes the number of collisions, where a collision is said to occur when distinct key values are mapped into the same hashed value. In the worst case, with the worst possible hash function, the number of operations is proportional to the number of elements in the sequence (linear time).

The hash_multimap should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. A model for this type of structure is an ordered list of key words with associated string values providing, say, definitions, where the words were not always uniquely defined. If, instead, the keywords were uniquely defined so that keys were unique, then a hash_map would be the container of choice. If, on the other hand, just the list of words were being stored, then a hash_set would be the correct container. If multiple occurrences of the words were allowed, then a hash_multiset would be the appropriate container structure.

The hash_multimap orders the sequence it controls by calling a stored hash **Traits** object of type value_compare. This stored object may be accessed by calling the member function key_comp. Such a function object must behave the same as an object of class hash_compare<*Key*, **less**<*Key*> >. Specifically, for all values *_Key* of type **Key**, the call **Traits***(_Key)* yields a distribution of values of type **size_t**.

In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the non-equivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate *f*(*x*,*y*) is a function object that has two argument objects x and y and a return value of true or false. An ordering imposed on a hash_multimap is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects x and y are defined to be equivalent when both *f*(*x*,*y*) and *f*(*y*,*x*) are false. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

The actual order of elements in the controlled sequence depends on the hash function, the ordering function, and the current size of the hash table stored in the container object. You cannot determine the current size of the hash table, so you cannot in general predict the order of elements in the controlled sequence. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

The iterator provided by the hash_multimap class is a bidirectional iterator, but the class member functions insert and hash_multimap have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own hash_multimap of requirements, and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal hash_multimap of functionality, but it is enough to be able to talk meaningfully about a range of iterators *[_First, _Last)* in the context of the member functions.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Requirements

Header: <hash_map>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<hash_map> Members

hash_multimap Members

hash_multimap Members

Reference

Constructors

hash_multimap	Constructs a list of a specific size or with elements of a specific value or with a specific **allocator** or as a copy of some other **hash_multimap**.

Typedefs

allocator_type	A type that represents the **allocator** class for the **hash_multimap** object.
const_iterator	A type that provides a bidirectional iterator that can read a **const** element in the **hash_multimap**.
const_pointer	A type that provides a pointer to a **const** element in a **hash_multimap**.
const_reference	A type that provides a reference to a **const** element stored in a **hash_multimap** for reading and performing **const** operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any **const** element in the **hash_multimap**.
difference_type	A signed integer type that can be used to represent the number of elements of a **hash_multimap** in a range between elements pointed to by iterators.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a **hash_multimap**.
key_compare	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the **hash_multimap**.
key_type	A type that describes the sort key object that constitutes each element of the **hash_multimap**.
mapped_type	A type that represents the data type stored in a **hash_multimap**.
pointer	A type that provides a pointer to an element in a **hash_multimap**.
reference	A type that provides a reference to an element stored in a **hash_multimap**.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed **hash_multimap**.
size_type	An unsigned integer type that can represent the number of elements in a **hash_multimap**.
value_type	A type that provides a function object that can compare two elements as sort keys to determine their relative order in the **hash_multimap**.

Member Functions

begin	Returns an iterator addressing the first element in the **hash_multimap**.
clear	Erases all the elements of a **hash_multimap**.
count	Returns the number of elements in a **hash_multimap** whose key matches a parameter-specified key.
empty	Tests if a **hash_multimap** is empty.

end	Returns an iterator that addresses the location succeeding the last element in a **hash_multimap**.
equal_range	Returns an iterator that addresses the location succeeding the last element in a **hash_multimap**.
erase	Removes an element or a range of elements in a **hash_multimap** from specified positions
find	Returns an iterator addressing the location of an element in a **hash_multimap** that has a key equivalent to a sp ecified key.
get_allocator	Returns a copy of the **allocator** object used to construct the **hash_multimap**.
insert	Inserts an element or a range of elements into the **hash_multimap** at a specified position.
key_comp	Retrieves a copy of the comparison object used to order keys in a **hash_multimap**.
lower_bound	Returns an iterator to the first element in a **hash_multimap** that with a key value that is equal to or greater than that of a specified key.
max_size	Returns the maximum length of the **hash_multimap**.
rbegin	Returns an iterator addressing the first element in a reversed **hash_multimap**.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed **hash_multimap**.
size	Specifies a new size for a **hash_multimap**.
swap	Exchanges the elements of two **hash_multimap**s.
upper_bound	Returns an iterator to the first element in a **hash_multimap** that with a key value that is greater than that of a s pecified key.
value_comp	Retrieves a copy of the comparison object used to order element values in a **hash_multimap**.

See Also

Reference

hash_multimap Class
Thread Safety in the Standard C++ Library
Standard Template Library

hash_multimap Typedefs

For information about the typedefs in the hash_multimap class, see hash_multimap Members.

See Also

Reference

Standard Template Library

hash_multimap::allocator_type

A type that represents the allocator class for the hash_multimap object.

```
typedef Allocator allocator_type
```

Remarks

allocator_type is a synonym for the template parameter **Allocator**.

For more information on **Allocator**, see the Remarks section of the hash_multimap Class topic.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for get_allocator for an example using **allocator_type**.

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::const_iterator

A type that provides a bidirectional iterator that can read a **const** element in the hash_multimap.

```
typedef implementation-defined const_iterator;
```

Remarks

A type **const_iterator** cannot be used to modify the value of an element.

The **const_iterator** defined by hash_multimap points to objects of value_type, which are of type **pair**<**const Key, Type**>. The value of the key is available through the first member pair, and the value of the mapped element is available through the second member of the pair.

To dereference a **const_iterator** *clter* pointing to an element in a hash_multimap, use the **->** operator.

To access the value of the key for the element, use *clter* -> **first**, which is equivalent to (**clter*).**first**. To access the value of the mapped datum for the element, use *clter* -> **second**, which is equivalent to (**clter*).**first**.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for begin for an example using **const_iterator**.

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::const_pointer

A type that provides a pointer to a **const** element in a hash_multimap.

```
typedef typename Allocator::const_pointer const_pointer;
```

Remarks

A type **const_pointer** cannot be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a hash_multimap object.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::const_reference

A type that provides a reference to a **const** element stored in a hash_multimap for reading and performing **const** operations.

```
typedef typename Allocator::const_reference const_reference;
```

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_const_ref.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap<int, int> hm1;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );

   // Declare and initialize a const_reference &Ref1
   // to the key of the first element
   const int &Ref1 = ( hm1.begin( ) -> first );

   // The following line would cause an error because the
   // non-const_reference cannot be used to access the key
   // int &Ref1 = ( hm1.begin( ) -> first );

   cout << "The key of first element in the hash_multimap is "
        << Ref1 << "." << endl;

   // Declare and initialize a reference &Ref2
   // to the data value of the first element
   int &Ref2 = ( hm1.begin() -> second );

   cout << "The data value of 1st element in the hash_multimap is "
        << Ref2 << "." << endl;
}
```

Output

```
The key of first element in the hash_multimap is 1.
The data value of 1st element in the hash_multimap is 10.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::const_reverse_iterator

A type that provides a bidirectional iterator that can read any **const** element in the hash_multimap.

```
typedef reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type **const_reverse_iterator** cannot modify the value of an element and is use to iterate through the hash_multimap in reverse.

The **const_reverse_iterator** defined by hash_multimap points to objects of value_type, which are of type **pair**<**const Key, Type>**, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a **const_reverse_iterator** *crIter* pointing to an element in a hash_multimap, use the **->** operator.

To access the value of the key for the element, use *crIter* -> **first**, which is equivalent to (**crIter*).**first**. To access the value of the mapped datum for the element, use *crIter* -> **second**, which is equivalent to (**crIter*).**first**.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for rend for an example of how to declare and use the **const_reverse_iterator**.

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::difference_type

A signed integer type that can be used to represent the number of elements of a hash_multimap in a range between elements pointed to by iterators.

```
typedef implementation-defined difference_type;
```

Remarks

The **difference_type** is the type returned when subtracting or incrementing through iterators of the container. The **difference_type** is typically used to represent the number of elements in the range *[_First, _Last)* between the iterators *_First* and *_Last*, includes the element pointed to by *_First* and the range of elements up to, but not including, the element pointed to by *_Last*.

Note that although **difference_type** is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as set, subtraction between iterators is only supported by random-access iterators provided by a random-access container such as vector.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_diff_type.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <iostream>
#include <hash_map>
#include <algorithm>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 3, 20 ) );

   // The following will insert, because map keys
   // do not need to be unique
   hm1.insert ( Int_Pair ( 2, 30 ) );

   hash_multimap <int, int>::iterator hm1_Iter, hm1_bIter, hm1_eIter;
   hm1_bIter = hm1.begin( );
   hm1_eIter = hm1.end( );

   // Count the number of elements in a hash_multimap
   hash_multimap <int, int>::difference_type  df_count = 0;
   hm1_Iter = hm1.begin( );
   while ( hm1_Iter != hm1_eIter)
   {
      df_count++;
      hm1_Iter++;
   }

   cout << "The number of elements in the hash_multimap hm1 is: "
        << df_count << "." << endl;

   cout  << "The keys of the mapped elements are:";
   for ( hm1_Iter= hm1.begin( ) ; hm1_Iter!= hm1.end( ) ;
      hm1_Iter++)
```

```
      cout << " " << hm1_Iter-> first;
   cout << "." << endl;

   cout  << "The values of the mapped elements are:";
   for ( hm1_Iter= hm1.begin( ) ; hm1_Iter!= hm1.end( ) ;
      hm1_Iter++)
      cout << " " << hm1_Iter-> second;
   cout << "." << endl;
}
```

Output

```
The number of elements in the hash_multimap hm1 is: 4.
The keys of the mapped elements are: 1 2 2 3.
The values of the mapped elements are: 10 30 20 20.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::iterator

A type that provides a bidirectional iterator that can read or modify any element in a hash_multimap.

```
typedef implementation-defined iterator;
```

Remarks

The **iterator** defined by hash_multimap points to objects of value_type, which are of type **pair**<**const Key, Type**>, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference an **iterator** *Iter* pointing to an element in a hash_multimap, use the **->** operator.

To access the value of the key for the element, use *Iter* -> **first**, which is equivalent to (**Iter*).**first**. To access the value of the mapped datum for the element, use *Iter* -> **second**, which is equivalent to (**Iter*).**first**.

A type **iterator** can be used to modify the value of an element.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for begin for an example of how to declare and use **iterator**.

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::key_compare

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the hash_multimap.

```
typedef Traits key_compare;
```

Remarks

key_compare is a synonym for the template parameter **Traits**.

For more information on **Traits** see the hash_multimap Class topic.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for key_comp for an example of how to declare and use **key_compare**.

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::key_type

A type that describes the sort key object that constitutes each element of the hash_multimap.

```
typedef Key key_type;
```

Remarks

key_type is a synonym for the template parameter **Key**.

For more information on **Key**, see the Remarks section of the hash_multimap Class topic.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for value_type for an example of how to declare and use **key_compare**.

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::mapped_type

A type that represents the data type stored in a hash_multimap.

```
typedef Type mapped_type;
```

Remarks

mapped_type is a synonym for the template parameter **Type**.

For more information on **Type** see the hash_multimap Class topic.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for value_type for an example of how to declare and use **key_type**.

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::pointer

A type that provides a pointer to an element in a hash_multimap.

```
typedef typename Allocator::pointer pointer;
```

Remarks

A type **pointer** can be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a hash_multimap object.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::reference

A type that provides a reference to an element stored in a hash_multimap.

```
typedef typename Allocator::reference reference;
```

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_reference.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );

   // Declare and initialize a const_reference &Ref1
   // to the key of the first element
   const int &Ref1 = ( hm1.begin( ) -> first );

   // The following line would cause an error as the
   // non-const_reference cannot be used to access the key
   // int &Ref1 = ( hm1.begin( ) -> first );

   cout << "The key of first element in the hash_multimap is "
        << Ref1 << "." << endl;

   // Declare and initialize a reference &Ref2
   // to the data value of the first element
   int &Ref2 = ( hm1.begin( ) -> second );

   cout << "The data value of first element in the hash_multimap is "
        << Ref2 << "." << endl;

   //The non-const_reference can be used to modify the
   //data value of the first element
   Ref2 = Ref2 + 5;
   cout << "The modified data value of first element is "
        << Ref2 << "." << endl;
}
```

Output

```
The key of first element in the hash_multimap is 1.
The data value of first element in the hash_multimap is 10.
The modified data value of first element is 15.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::reverse_iterator

A type that provides a bidirectional iterator that can read or modify an element in a reversed hash_multimap.

```
typedef reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type **reverse_iterator** is used to iterate through the hash_multimap in reverse.

The **reverse_iterator** defined by hash_multimap points to objects of value_type, which are of type **pair**<**const Key, Type**>. The value of the key is available through the first member pair and the value of the mapped element is available through the second member of the pair.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for rbegin for an example of how to declare and use **reverse_iterator**.

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::size_type

An unsigned integer type that counts the number of elements in a hash_multimap.

```
typedef implementation-defined size_type;
```

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for size for an example of how to declare and use **size_type**

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::value_type

A type that represents the type of object stored in a hash_multimap.

```
typedef pair<const Key, Type> value_type;
```

Remarks

value_type is declared to be **pair** <**const** key_type, mapped_type> and not **pair** <**key_type**, **mapped_type**> because the keys of an associative container may not be changed using a nonconstant iterator or reference.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_value_type.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   typedef pair <const int, int> cInt2Int;
   hash_multimap <int, int> hm1;
   hash_multimap <int, int> :: key_type key1;
   hash_multimap <int, int> :: mapped_type mapped1;
   hash_multimap <int, int> :: value_type value1;
   hash_multimap <int, int> :: iterator pIter;

   // value_type can be used to pass the correct type
   // explicitely to avoid implicit type conversion
   hm1.insert ( hash_multimap <int, int> :: value_type ( 1, 10 ) );

   // Compare another way to insert objects into a hash_multimap
   hm1.insert ( cInt2Int ( 2, 20 ) );

   // Initializing key1 and mapped1
   key1 = ( hm1.begin( ) -> first );
   mapped1 = ( hm1.begin( ) -> second );

   cout << "The key of first element in the hash_multimap is "
        << key1 << "." << endl;

   cout << "The data value of first element in the hash_multimap is "
        << mapped1 << "." << endl;

   // The following line would cause an error because
   // the value_type is not assignable
   // value1 = cInt2Int ( 4, 40 );

   cout  << "The keys of the mapped elements are:";
   for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
      cout << " " << pIter -> first;
   cout << "." << endl;

   cout  << "The values of the mapped elements are:";
   for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
      cout << " " << pIter -> second;
   cout << "." << endl;
}
```

Output

```
The key of first element in the hash_multimap is 1.
The data value of first element in the hash_multimap is 10.
The keys of the mapped elements are: 1 2.
The values of the mapped elements are: 10 20.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap Member Functions

For information about the functions in the hash_multimap class, see hash_multimap Members.

See Also

Reference

Standard Template Library

hash_multimap::begin

Returns an iterator addressing the first element in the hash_multimap.

```
const_iterator begin( ) const;
iterator begin( );
```

Return Value

A bidirectional iterator addressing the first element in the hash_multimap or the location succeeding an empty hash_multimap.

Remarks

If the return value of **begin** is assigned to a **const_iterator**, the elements in the hash_multimap object cannot be modified. If the return value of **begin** is assigned to an **iterator**, the elements in the hash_multimap object can be modified.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_begin.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;

   hash_multimap <int, int> :: iterator hm1_Iter;
   hash_multimap <int, int> :: const_iterator hm1_cIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 0, 0 ) );
   hm1.insert ( Int_Pair ( 1, 1 ) );
   hm1.insert ( Int_Pair ( 2, 4 ) );

   hm1_cIter = hm1.begin ( );
   cout << "The first element of hm1 is " << hm1_cIter -> first
        << "." << endl;

   hm1_Iter = hm1.begin ( );
   hm1.erase ( hm1_Iter );

   // The following 2 lines would err because the iterator is const
   // hm1_cIter = hm1.begin ( );
   // hm1.erase ( hm1_cIter );

   hm1_cIter = hm1.begin( );
   cout << "The first element of hm1 is now " << hm1_cIter -> first
        << "." << endl;
}
```

Output

```
The first element of hm1 is 0.
The first element of hm1 is now 1.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::clear

Erases all the elements of a hash_multimap.

```
void clear( );
```

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// hash_multimap_clear.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, int> hm1;
    hash_multimap<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(1, 1));
    hm1.insert(Int_Pair(2, 4));

    i = hm1.size();
    cout << "The size of the hash_multimap is initially "
         << i  << "." << endl;

    hm1.clear();
    i = hm1.size();
    cout << "The size of the hash_multimap after clearing is "
         << i << "." << endl;
}
```

Output

```
The size of the hash_multimap is initially 2.
The size of the hash_multimap after clearing is 0.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::count

Returns the number of elements in a hash_multimap whose key matches a parameter-specified key.

```
size_type count(
   const Key& _Key
) const;
```

Parameters

_Key

The key of the elements to be matched from the hash_multimap.

Return Value

1 if the hash_multimap contains an element whose sort key matches the parameter key; 0 if the hash_multimap doesn't contain an element with a matching key.

Remarks

The member function returns the number of elements in the range

[lower_bound (_*Key*), upper_bound (_*Key*))

which have a key value _*Key*.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// hash_multimap_count.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, int> hm1;
    hash_multimap<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(1, 1));
    hm1.insert(Int_Pair(2, 1));
    hm1.insert(Int_Pair(1, 4));
    hm1.insert(Int_Pair(2, 1));

    // Elements do not need to have unique keys in hash_multimap,
    // so duplicates are allowed and counted
    i = hm1.count(1);
    cout << "The number of elements in hm1 with a sort key of 1 is: "
         << i << "." << endl;

    i = hm1.count(2);
    cout << "The number of elements in hm1 with a sort key of 2 is: "
         << i << "." << endl;

    i = hm1.count(3);
    cout << "The number of elements in hm1 with a sort key of 3 is: "
         << i << "." << endl;
```

```
}
```

Output

```
The number of elements in hm1 with a sort key of 1 is: 2.
The number of elements in hm1 with a sort key of 2 is: 2.
The number of elements in hm1 with a sort key of 3 is: 0.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::empty

Tests if a hash_multimap is empty.

```
bool empty( ) const;
```

Return Value

true if the hash_multimap is empty; **false** if the hash_multimap is nonempty.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_empty.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1, hm2;

   typedef pair <int, int> Int_Pair;
   hm1.insert ( Int_Pair ( 1, 1 ) );

   if ( hm1.empty( ) )
      cout << "The hash_multimap hm1 is empty." << endl;
   else
      cout << "The hash_multimap hm1 is not empty." << endl;

   if ( hm2.empty( ) )
      cout << "The hash_multimap hm2 is empty." << endl;
   else
      cout << "The hash_multimap hm2 is not empty." << endl;
}
```

Output

```
The hash_multimap hm1 is not empty.
The hash_multimap hm2 is empty.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::end

Returns an iterator that addresses the location succeeding the last element in a hash_multimap.

```
const_iterator end( ) const;
iterator end( );
```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a hash_multimap. If the hash_multimap is empty, then hash_multimap::end == hash_multimap::begin.

Remarks

end is used to test whether an iterator has reached the end of its hash_multimap.

The value returned by **end** should not be dereferenced.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_end.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;

   hash_multimap <int, int> :: iterator hm1_Iter;
   hash_multimap <int, int> :: const_iterator hm1_cIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_cIter = hm1.end( );
   hm1_cIter--;
   cout << "The value of last element of hm1 is "
        << hm1_cIter -> second << "." << endl;

   hm1_Iter = hm1.end( );
   hm1_Iter--;
   hm1.erase ( hm1_Iter );

   // The following 2 lines would err because the iterator is const
   // hm1_cIter = hm1.end ( );
   // hm1_cIter--;
   // hm1.erase ( hm1_cIter );

   hm1_cIter = hm1.end( );
   hm1_cIter--;
   cout << "The value of last element of hm1 is now "
        << hm1_cIter -> second << "." << endl;
}
```

Output

```
The value of last element of hm1 is 30.
The value of last element of hm1 is now 20.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::equal_range

Returns a pair of iterators respectively to the first element in a hash_multimap with a key that is greater than a specified key and to the first element in the hash_multimap with a key that is equal to or greater than the key.

```
pair <const_iterator, const_iterator> equal_range (
   const Key& _Key
) const;
pair <iterator, iterator> equal_range (
   const Key& _Key
);
```

Parameters

_Key

The argument key to be compared with the sort key of an element from the hash_multimap being searched.

Return Value

A pair of iterators such that the first is the lower_bound of the key and the second is the upper_bound of the key.

To access the first iterator of a pair *pr* returned by the member function, use *pr*.**first** and to dereference the lower bound iterator, use *(*pr*.**first**). To access the second iterator of a pair *pr* returned by the member function, use *pr*.**second** and to dereference the upper bound iterator, use *(*pr*.**second**).

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_equal_range.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   typedef hash_multimap <int, int> IntMMap;
   IntMMap hm1;
   hash_multimap <int, int> :: const_iterator hm1_RcIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   pair <IntMMap::const_iterator, IntMMap::const_iterator> p1, p2;
   p1 = hm1.equal_range( 2 );

   cout << "The lower bound of the element with "
        << "a key of 2\n in the hash_multimap hm1 is: "
        << p1.first -> second << "." << endl;

   cout << "The upper bound of the element with "
        << "a key of 2\n in the hash_multimap hm1 is: "
        << p1.second -> second << "." << endl;

   // Compare the upper_bound called directly
   hm1_RcIter = hm1.upper_bound( 2 );
```

```
    cout << "A direct call of upper_bound( 2 ) gives "
         << hm1_RcIter -> second << "," << endl
         << " matching the 2nd element of the pair"
         << " returned by equal_range( 2 )." << endl;

    p2 = hm1.equal_range( 4 );

    // If no match is found for the key,
    // both elements of the pair return end( )
    if ( ( p2.first == hm1.end( ) ) && ( p2.second == hm1.end( ) ) )
       cout << "The hash_multimap hm1 doesn't have an element "
            << "with a key less than 4." << endl;
    else
       cout << "The element of hash_multimap hm1 with a key >= 40 is: "
            << p1.first -> first << "." << endl;
 }
```

Output

```
The lower bound of the element with a key of 2
 in the hash_multimap hm1 is: 20.
The upper bound of the element with a key of 2
 in the hash_multimap hm1 is: 30.
A direct call of upper_bound( 2 ) gives 30,
 matching the 2nd element of the pair returned by equal_range( 2 ).
The hash_multimap hm1 doesn't have an element with a key less than 4.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::erase

Removes an element or a range of elements in a hash_multimap from specified positions or removes elements that match a specified key.

```
iterator erase(
   iterator _Where
);
iterator erase(
   iterator _First,
   iterator _Last
);
size_type erase(
   const key_type& _Key
);
```

Parameters

_Where

Position of the element to be removed from the hash_multimap.

_First

Position of the first element removed from the hash_multimap.

_Last

Position just beyond the last element removed from the hash_multimap.

_Key

The key of the elements to be removed from the hash_multimap.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the hash_multimap if no such element exists.

For the third member function, returns the number of elements that have been removed from the hash_multimap.

Remarks

The member functions never throw an exception.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// hash_multimap_erase.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, int> hm1, hm2, hm3;
    hash_multimap<int, int> :: iterator pIter, Iter1, Iter2;
    int i;
    hash_multimap<int, int>::size_type n;
    typedef pair<int, int> Int_Pair;
```

```
    for (i = 1; i < 5; i++)
    {
        hm1.insert(Int_Pair (i, i) );
        hm2.insert(Int_Pair (i, i*i) );
        hm3.insert(Int_Pair (i, i-1) );
    }

    // The 1st member function removes an element at a given position
    Iter1 = ++hm1.begin();
    hm1.erase(Iter1);

    cout << "After the 2nd element is deleted, "
         << "the hash_multimap hm1 is:";
    for (pIter = hm1.begin(); pIter != hm1.end(); pIter++)
        cout << " " << pIter -> second;
    cout << "." << endl;

    // The 2nd member function removes elements
    // in the range [_First, _Last)
    Iter1 = ++hm2.begin();
    Iter2 = --hm2.end();
    hm2.erase(Iter1, Iter2);

    cout << "After the middle two elements are deleted, "
         << "the hash_multimap hm2 is:";
    for (pIter = hm2.begin(); pIter != hm2.end(); pIter++)
        cout << " " << pIter -> second;
    cout << "." << endl;

    // The 3rd member function removes elements with a given _Key
    hm3.insert(Int_Pair (2, 5));
    n = hm3.erase(2);

    cout << "After the element with a key of 2 is deleted,\n"
         << " the hash_multimap hm3 is:";
    for (pIter = hm3.begin(); pIter != hm3.end(); pIter++)
        cout << " " << pIter -> second;
    cout << "." << endl;

    // The 3rd member function returns the number of elements removed
    cout << "The number of elements removed from hm3 is: "
         << n << "." << endl;

    // The dereferenced iterator can also be used to specify a key
    Iter1 = ++hm3.begin();
    hm3.erase(Iter1);

    cout << "After another element with a key equal to that of the"
         << endl;
    cout  << " 2nd element is deleted, "
          << "the hash_multimap hm3 is:";
    for (pIter = hm3.begin(); pIter != hm3.end(); pIter++)
        cout << " " << pIter -> second;
    cout << "." << endl;
}
```

Output

```
After the 2nd element is deleted, the hash_multimap hm1 is: 1 3 4.
After the middle two elements are deleted, the hash_multimap hm2 is: 1 16.
After the element with a key of 2 is deleted,
 the hash_multimap hm3 is: 0 2 3.
The number of elements removed from hm3 is: 2.
After another element with a key equal to that of the
 2nd element is deleted, the hash_multimap hm3 is: 0 3.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::find

Returns an iterator addressing the first location of an element in a hash_multimap that has a key equivalent to a specified key.

```
iterator find(
   const Key& _Key
);
const_iterator find(
   const Key& _Key
) const;
```

Parameters

_Key

The key to be matched by the sort key of an element from the hash_multimap being searched.

Return Value

An iterator that addresses the first location of an element with a specified key, or the location succeeding the last element in the hash_multimap if no match is found for the key.

Remarks

The member function returns an iterator that addresses an element in the hash_multimap whose sort key is **equivalent** to the argument key under a binary predicate that induces an ordering based on a less than comparability relation.

If the return value of **find** is assigned to a **const_iterator**, the hash_multimap object cannot be modified. If the return value of **find** is assigned to an **iterator**, the hash_multimap object can be modified.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_find.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <iostream>
#include <hash_map>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;
   hash_multimap <int, int> :: const_iterator hm1_AcIter, hm1_RcIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_RcIter = hm1.find( 2 );
   cout << "The element of hash_multimap hm1 with a key of 2 is: "
        << hm1_RcIter -> second << "." << endl;

   hm1_RcIter = hm1.find( 3 );
   cout << "The first element of hash_multimap hm1 with a key of 3 is: "
        << hm1_RcIter -> second << "." << endl;

   // If no match is found for the key, end( ) is returned
   hm1_RcIter = hm1.find( 4 );

   if ( hm1_RcIter == hm1.end( ) )
      cout << "The hash_multimap hm1 doesn't have an element "
```

```
            << "with a key of 4." << endl;
    else
       cout << "The element of hash_multimap hm1 with a key of 4 is: "
            << hm1_RcIter -> second << "." << endl;

    // The element at a specific location in the hash_multimap can be
    // found using a dereferenced iterator addressing the location
    hm1_AcIter = hm1.end( );
    hm1_AcIter--;
    hm1_RcIter = hm1.find( hm1_AcIter -> first );
    cout << "The first element of hm1 with a key matching"
         << endl << "that of the last element is: "
         << hm1_RcIter -> second << "." << endl;

    // Note that the first element with a key equal to
    // the key of the last element is not the last element
    if ( hm1_RcIter == --hm1.end( ) )
       cout << "This is the last element of hash_multimap hm1."
            << endl;
    else
       cout << "This is not the last element of hash_multimap hm1."
            << endl;
 }
```

Output

```
The element of hash_multimap hm1 with a key of 2 is: 20.
The first element of hash_multimap hm1 with a key of 3 is: 30.
The hash_multimap hm1 doesn't have an element with a key of 4.
The first element of hm1 with a key matching
that of the last element is: 30.
This is not the last element of hash_multimap hm1.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::get_allocator

Returns a copy of the allocator object used to construct the hash_multimap.

```
Allocator get_allocator( ) const;
```

Return Value

The allocator used by the hash_multimap.

Remarks

Allocators for the hash_multimap class specify how the class manages storage. The default allocators supplied with STL container classes is sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_get_allocator.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int>::allocator_type hm1_Alloc;
   hash_multimap <int, int>::allocator_type hm2_Alloc;
   hash_multimap <int, double>::allocator_type hm3_Alloc;
   hash_multimap <int, int>::allocator_type hm4_Alloc;

   // The following lines declare objects
   // that use the default allocator.
   hash_multimap <int, int> hm1;
   hash_multimap <int, int> hm2;
   hash_multimap <int, double> hm3;

   hm1_Alloc = hm1.get_allocator( );
   hm2_Alloc = hm2.get_allocator( );
   hm3_Alloc = hm3.get_allocator( );

   cout << "The number of integers that can be allocated"
        << endl << " before free memory is exhausted: "
        << hm2.max_size( ) << "." << endl;

   cout << "The number of doubles that can be allocated"
        << endl << " before free memory is exhausted: "
        << hm3.max_size( ) <<  "." << endl;

   // The following line creates a hash_multimap hm4
   // with the allocator of hash_multimap hm1.
   hash_multimap <int, int> hm4( less<int>( ), hm1_Alloc );

   hm4_Alloc = hm4.get_allocator( );

   // Two allocators are interchangeable if
   // storage allocated from each can be
   // deallocated by the other
   if( hm1_Alloc == hm4_Alloc )
   {
```

```
        cout << "The allocators are interchangeable."
             << endl;
    }
    else
    {
        cout << "The allocators are not interchangeable."
             << endl;
    }
}
```

Sample Output

The following output is for x86.

```
The number of integers that can be allocated
 before free memory is exhausted: 536870911.
The number of doubles that can be allocated
 before free memory is exhausted: 268435455.
The allocators are interchangeable.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::hash_multimap

Constructs a hash_multimap that is empty or that is a copy of all or part of some other hash_multimap.

```
multimap( );
explicit hash_multimap(
   const Compare& _Comp
);
explicit hash_multimap(
   const Compare& _Comp,
   const Allocator& _Al
);
map(
   const _ hash_multimap& _Right
);
template<class InputIterator>
   hash_multimap(
      InputIterator _First,
      InputIterator _Last
   );
template<class InputIterator>
   hash_multimap(
      InputIterator _First,
      InputIterator _Last,
      const Compare& _Comp
   );
template<class InputIterator>
   hash_multimap(
      InputIterator _First,
      InputIterator _Last,
      const Compare& _Comp,
      const Allocator& _Al
   );
```

Parameters

_Al

The storage allocator class to be used for this hash_multimap object, which defaults to **Allocator**.

_Comp

The comparison function of type **const Traits** used to order the elements in the map, which defaults to **Traits**.

_Right

The map of which the constructed set is to be a copy.

_First

The position of the first element in the range of elements to be copied.

_Last

The position of the first element beyond the range of elements to be copied.

Remarks

All constructors store a type of allocator object that manages memory storage for the hash_multimap and that can later be returned by calling get_allocator. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their hash_multimap.

All constructors store a function object of type **Traits** that is used to establish an order among the keys of the hash_multimap and that can later be returned by calling key_comp.

The first three constructors specify an empty initial hash_multimap, the second specifying the type of comparison function

(*_Comp*) to be used in establishing the order of the elements and the third explicitly specifying the allocator type (*_Al*) to be used. The keyword **explicit** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the hash_multimap *_Right*.

The last three constructors copy the range [*_First, _Last*) of a map with increasing explicitness in specifying the type of comparison function of class **Traits** and allocator.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_hash_multimap.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   typedef pair <int, int> Int_Pair;
   hash_multimap <int, int>::iterator hm1_Iter, hm3_Iter, hm4_Iter,
      hm5_Iter, hm6_Iter;
   hash_multimap <int, int, hash_compare <int, greater<int> >
      >::iterator hm2_Iter;

   // Create an empty hash_multimap hm0 of key type integer
   hash_multimap <int, int> hm0;

   // Create an empty hash_multimap hm1 with the key comparison
   // function of less than, then insert 4 elements
   hash_multimap <int, int, hash_compare <int, less<int> > > hm1;
   hm1.insert( Int_Pair( 1, 10 ) );
   hm1.insert( Int_Pair( 2, 20 ) );
   hm1.insert( Int_Pair( 3, 30 ) );
   hm1.insert( Int_Pair( 4, 40 ) );

   // Create an empty hash_multimap hm2 with the key comparison
   // function of greater than, then insert 2 elements
   hash_multimap <int, int, hash_compare <int, greater<int> > > hm2;
   hm2.insert( Int_Pair( 1, 10 ) );
   hm2.insert( Int_Pair( 2, 20 ) );

   // Create a hash_multimap hm3 with the
   // allocator of hash_multimap hm1
   hash_multimap <int, int>::allocator_type hm1_Alloc;
   hm1_Alloc = hm1.get_allocator( );
   hash_multimap <int, int> hm3( hash_compare <int, less<int> > ( ),
      hm1_Alloc );
   hm3.insert( Int_Pair( 3, 30 ) );

   // Create a copy, hash_multimap hm4, of hash_multimap hm1
   hash_multimap <int, int> hm4( hm1 );

   // Create a hash_multimap hm5 by copying the range hm1[_First, _Last)
   hash_multimap <int, int>::const_iterator hm1_bcIter, hm1_ecIter;
   hm1_bcIter = hm1.begin( );
   hm1_ecIter = hm1.begin( );
   hm1_ecIter++;
   hm1_ecIter++;
   hash_multimap <int, int> hm5( hm1_bcIter, hm1_ecIter );

   // Create a hash_multimap hm6 by copying the range hm4[_First, _Last)
   // and with the allocator of hash_multimap hm2
   hash_multimap <int, int>::allocator_type hm2_Alloc;
   hm2_Alloc = hm2.get_allocator( );
```

```
    hash_multimap <int, int> hm6(hm4.begin( ), ++hm4.begin( ), less<int>( ),
       hm2_Alloc);

    cout << "hm1 = ";
    for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++ )
       cout << hm1_Iter -> second << " ";
    cout << endl;

    cout << "hm2 = ";
    for ( hm2_Iter = hm2.begin( ); hm2_Iter != hm2.end( ); hm2_Iter++ )
    cout << hm2_Iter -> second << " ";
    cout << endl;

    cout << "hm3 = ";
    for ( hm3_Iter = hm3.begin( ); hm3_Iter != hm3.end( ); hm3_Iter++ )
       cout << hm3_Iter -> second << " ";
    cout << endl;

    cout << "hm4 = ";
    for ( hm4_Iter = hm4.begin( ); hm4_Iter != hm4.end( ); hm4_Iter++ )
       cout << hm4_Iter -> second << " ";
    cout << endl;

    cout << "hm5 = ";
    for ( hm5_Iter = hm5.begin( ); hm5_Iter != hm5.end( ); hm5_Iter++ )
       cout << hm5_Iter -> second << " ";
    cout << endl;

    cout << "hm6 = ";
    for ( hm6_Iter = hm6.begin( ); hm6_Iter != hm6.end( ); hm6_Iter++ )
       cout << hm6_Iter -> second << " ";
    cout << endl;
 }
```

Output

```
hm1 = 10 20 30 40
hm2 = 20 10
hm3 = 30
hm4 = 10 20 30 40
hm5 = 10 20
hm6 = 10
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::insert

Inserts an element or a range of elements into a hash_multimap.

```
iterator insert(
   const value_type& _Val
);
iterator insert(
   iterator _Where,
   const value_type& _Val
);
template<class InputIterator>
   void insert(
      InputIterator _First,
      InputIterator _Last
   );
```

Parameters

_Val

The value of an element to be inserted into the hash_multimap unless the hash_multimap already contains that element or, more generally, an element whose key is equivalently ordered.

_Where

A hint regarding the place to start searching for the correct point of insertion.

_First

The position of the first element to be copied from a map.

_Last

The position just beyond the last element to be copied from a map.

Return Value

The first **insert** member function returns a pair whose bool component returns true if an insertion was make and false if the map already contained an element whose key had an equivalent value in the ordering, and whose iterator component returns the address where a new element was inserted or where the element was already located.

The second **insert** member function, the hint version, returns an iterator that points to the position where the new element was inserted into the map.

Remarks

The value_type of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Insertion can occur in amortized constant time for the hint version of insert, instead of logarithmic time, if the insertion point immediately follows *_Where*.

The third member function inserts the sequence of element values into a map corresponding to each element addressed by an iterator of in the range *[_First, _Last)* of a specified set.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_insert.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
```

```
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int>::iterator hm1_pIter, hm2_pIter;

   hash_multimap <int, int> hm1, hm2;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   cout << "The original key values of hm1 =";
   for ( hm1_pIter = hm1.begin( ); hm1_pIter != hm1.end( );
      hm1_pIter++ )
      cout << " " << hm1_pIter -> first;
   cout << "." << endl;

   cout << "The original mapped values of hm1 =";
   for ( hm1_pIter = hm1.begin( ); hm1_pIter != hm1.end( );
      hm1_pIter++ )
      cout << " " << hm1_pIter -> second;
   cout << "." << endl;

   hm1.insert ( Int_Pair ( 1, 10 ) );

   // The hint version of insert
   hm1.insert( --hm1.end( ), Int_Pair ( 4, 40 )  );

   cout << "After the insertions, the key values of hm1 =";
   for ( hm1_pIter = hm1.begin( ); hm1_pIter != hm1.end( );
      hm1_pIter++ )
      cout << " " << hm1_pIter -> first;
   cout << "," << endl;

   cout << " and the mapped values of hm1 =";
   for ( hm1_pIter = hm1.begin( ); hm1_pIter != hm1.end( );
      hm1_pIter++ )
      cout << " " << hm1_pIter -> second;
   cout << "." << endl;

   hm2.insert ( Int_Pair ( 10, 100 ) );

   // The templatized version inserting a range
   hm2.insert( ++hm1.begin( ), --hm1.end( ) );

   cout << "After the insertions, the key values of hm2 =";
   for ( hm2_pIter = hm2.begin( ); hm2_pIter != hm2.end( );
      hm2_pIter++ )
      cout << " " << hm2_pIter -> first;
   cout << "," << endl;

   cout << " and the mapped values of hm2 =";
   for ( hm2_pIter = hm2.begin( ); hm2_pIter != hm2.end( );
      hm2_pIter++ )
      cout << " " << hm2_pIter -> second;
   cout << "." << endl;
}
```

Output

```
The original key values of hm1 = 1 2 3.
The original mapped values of hm1 = 10 20 30.
After the insertions, the key values of hm1 = 1 1 3 2 4,
 and the mapped values of hm1 = 10 10 30 20 40.
After the insertions, the key values of hm2 = 1 2 3 10,
 and the mapped values of hm2 = 10 20 30 100.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::key_comp

Retrieves a copy of the comparison object used to order keys in a hash_multimap.

```
key_compare key_comp( ) const;
```

Return Value

Returns the function object that a hash_multimap uses to order its elements.

Remarks

The stored object defines the member function

bool operator(const Key& *_Left* **, const Key&** *_Right* **);**

which returns **true** if *_Left* precedes and is not equal to *_Right* in the sort order.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_key_comp.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;

   hash_multimap <int, int, hash_compare<int, less<int> > > hm1;
   hash_multimap <int, int, hash_compare<int, less<int> >
      >::key_compare kc1 = hm1.key_comp( ) ;
   bool result1 = kc1( 2, 3 ) ;
   if( result1 == true )
   {
      cout << "kc1( 2,3 ) returns value of true,\n"
           << "where kc1 is the function object of hm1.\n"
           << endl;
   }
   else
   {
      cout << "kc1( 2,3 ) returns value of false,\n"
           << "where kc1 is the function object of hm1.\n"
           << endl;
   }

   hash_multimap <int, int, hash_compare<int, greater<int> > > hm2;
   hash_multimap <int, int, hash_compare<int, greater<int> >
      >::key_compare kc2 = hm2.key_comp( );
   bool result2 = kc2( 2, 3 ) ;
   if( result2 == true )
   {
      cout << "kc2( 2,3 ) returns value of true,\n"
           << "where kc2 is the function object of hm2."
           << endl;
   }
   else
   {
      cout << "kc2( 2,3 ) returns value of false,\n"
           << "where kc2 is the function object of hm2."
           << endl;
```

```
    }
  }
```

Output

```
kc1( 2,3 ) returns value of true,
where kc1 is the function object of hm1.

kc2( 2,3 ) returns value of false,
where kc2 is the function object of hm2.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::lower_bound

Returns an iterator to the first element in a hash_multimap with a key that is equal to or greater than a specified key.

```
iterator lower_bound(
   const Key& _Key
);
const_iterator lower_bound(
   const Key& _Key
) const;
```

Parameters

_Key

The argument key to be compared with the sort key of an element from the hash_multimap being searched.

Return Value

An iterator or const_iterator that addresses the location of an element in a hash_multimap with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the hash_multimap if no match is found for the key.

If the return value of **lower_bound** is assigned to a **const_iterator**, the hash_multimap object cannot be modified. If the return value of **lower_bound** is assigned to an **iterator**, the hash_multimap object can be modified.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_lower_bound.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;
   hash_multimap <int, int> :: const_iterator hm1_AcIter,
      hm1_RcIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_RcIter = hm1.lower_bound( 2 );
   cout << "The element of hash_multimap hm1 with a key of 2 is: "
        << hm1_RcIter -> second << "." << endl;

   hm1_RcIter = hm1.lower_bound( 3 );
   cout << "The first element of hash_multimap hm1 with a key of 3 is: "
        << hm1_RcIter -> second << "." << endl;

   // If no match is found for the key, end( ) is returned
   hm1_RcIter = hm1.lower_bound( 4 );

   if ( hm1_RcIter == hm1.end( ) )
      cout << "The hash_multimap hm1 doesn't have an element "
```

```
            << "with a key of 4." << endl;
    else
       cout << "The element of hash_multimap hm1 with a key of 4 is: "
            << hm1_RcIter -> second << "." << endl;

    // The element at a specific location in the hash_multimap can be
    // found using a dereferenced iterator addressing the location
    hm1_AcIter = hm1.end( );
    hm1_AcIter--;
    hm1_RcIter = hm1.lower_bound( hm1_AcIter -> first );
    cout << "The first element of hm1 with a key matching"
         << endl << " that of the last element is: "
         << hm1_RcIter -> second << "." << endl;

    // Note that the first element with a key equal to
    // the key of the last element is not the last element
    if ( hm1_RcIter == --hm1.end( ) )
       cout << "This is the last element of hash_multimap hm1."
            << endl;
    else
       cout << "This is not the last element of hash_multimap hm1."
            << endl;
 }
```

Output

```
The element of hash_multimap hm1 with a key of 2 is: 20.
The first element of hash_multimap hm1 with a key of 3 is: 30.
The hash_multimap hm1 doesn't have an element with a key of 4.
The first element of hm1 with a key matching
 that of the last element is: 30.
This is not the last element of hash_multimap hm1.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::max_size

Returns the maximum length of the hash_multimap.

```
size_type max_size( ) const;
```

Return Value

The maximum possible length of the hash_multimap.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_max_size.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;
   hash_multimap <int, int> :: size_type i;

   i = hm1.max_size( );
   cout << "The maximum possible length "
        << "of the hash_multimap is " << i << "." << endl;
}
```

Sample Output

The following output is for x86.

```
The maximum possible length of the hash_multimap is 536870911.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::rbegin

Returns an iterator addressing the first element in a reversed hash_multimap.

```
const_reverse_iterator rbegin( ) const;
reverse_iterator rbegin( );
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed hash_multimap or addressing what had been the last element in the unreversed hash_multimap.

Remarks

rbegin is used with a reversed hash_multimap just as begin is used with a hash_multimap.

If the return value of **rbegin** is assigned to a **const_reverse_iterator**, then the hash_multimap object cannot be modified. If the return value of **rbegin** is assigned to a **reverse_iterator**, then the hash_multimap object can be modified.

rbegin can be used to iterate through a hash_multimap backwards.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_rbegin.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;

   hash_multimap <int, int> :: iterator hm1_Iter;
   hash_multimap <int, int> :: reverse_iterator hm1_rIter;
   hash_multimap <int, int> :: const_reverse_iterator hm1_crIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_rIter = hm1.rbegin( );
   cout << "The first element of the reversed hash_multimap hm1 is "
        << hm1_rIter -> first << "." << endl;

   // begin can be used to start an iteration
   // through a hash_multimap in a forward order
   cout << "The hash_multimap is: ";
   for ( hm1_Iter = hm1.begin( ) ; hm1_Iter != hm1.end( ); hm1_Iter++)
      cout << hm1_Iter -> first << " ";
      cout << "." << endl;

   // rbegin can be used to start an iteration
   // through a hash_multimap in a reverse order
   cout << "The reversed hash_multimap is: ";
   for ( hm1_rIter = hm1.rbegin( ) ; hm1_rIter != hm1.rend( ); hm1_rIter++)
      cout << hm1_rIter -> first << " ";
      cout << "." << endl;
```

```
    // A hash_multimap element can be erased by dereferencing its key
    hm1_rIter = hm1.rbegin( );
    hm1.erase ( hm1_rIter -> first );

    hm1_rIter = hm1.rbegin( );
    cout << "After the erasure, the first element"
         << "\n in the reversed hash_multimap is "
         << hm1_rIter -> first << "." << endl;
}
```

Output

```
The first element of the reversed hash_multimap hm1 is 3.
The hash_multimap is: 1 2 3 .
The reversed hash_multimap is: 3 2 1 .
After the erasure, the first element
 in the reversed hash_multimap is 2.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::rend

Returns an iterator that addresses the location succeeding the last element in a reversed hash_multimap.

```
const_reverse_iterator rend( ) const;
reverse_iterator rend( );
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed hash_multimap (the location that had preceded the first element in the unreversed hash_multimap).

Remarks

rend is used with a reversed hash_multimap just as end is used with a hash_multimap.

If the return value of **rend** is assigned to a const_reverse_iterator, then the hash_multimap object cannot be modified. If the return value of **rend** is assigned to a reverse_iterator, then the hash_multimap object can be modified.

rend can be used to test to whether a reverse iterator has reached the end of its hash_multimap.

The value returned by **rend** should not be dereferenced.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_rend.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;

   hash_multimap <int, int> :: iterator hm1_Iter;
   hash_multimap <int, int> :: reverse_iterator hm1_rIter;
   hash_multimap <int, int> :: const_reverse_iterator hm1_crIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_rIter = hm1.rend( );
   hm1_rIter--;
   cout << "The last element of the reversed hash_multimap hm1 is "
        << hm1_rIter -> first << "." << endl;

   // begin can be used to start an iteration
   // through a hash_multimap in a forward order
   cout << "The hash_multimap is: ";
   for ( hm1_Iter = hm1.begin( ) ; hm1_Iter != hm1.end( ); hm1_Iter++)
      cout << hm1_Iter -> first << " ";
      cout << "." << endl;

   // rbegin can be used to start an iteration
   // through a hash_multimap in a reverse order
   cout << "The reversed hash_multimap is: ";
   for ( hm1_rIter = hm1.rbegin( ) ; hm1_rIter != hm1.rend( ); hm1_rIter++)
      cout << hm1_rIter -> first << " ";
```

```
        cout << "." << endl;

    // A hash_multimap element can be erased by dereferencing its key
    hm1_rIter = --hm1.rend( );
    hm1.erase ( hm1_rIter -> first );

    hm1_rIter = hm1.rend( );
    hm1_rIter--;
    cout << "After the erasure, the last element "
         << "in the reversed hash_multimap is "
         << hm1_rIter -> first << "." << endl;
}
```

Output

```
The last element of the reversed hash_multimap hm1 is 1.
The hash_multimap is: 1 2 3 .
The reversed hash_multimap is: 3 2 1 .
After the erasure, the last element in the reversed hash_multimap is 2.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::size

Returns the number of elements in the hash_multimap.

```
size_type size( ) const;
```

Return Value

The current length of the hash_multimap.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// hash_multimap_size.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, int> hm1, hm2;
    hash_multimap<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(1, 1));
    i = hm1.size();
    cout << "The hash_multimap length is " << i << "." << endl;

    hm1.insert(Int_Pair(2, 4));
    i = hm1.size();
    cout << "The hash_multimap length is now " << i << "." << endl;
}
```

Output

```
The hash_multimap length is 1.
The hash_multimap length is now 2.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::swap

Exchanges the elements of two hash_multimaps.

```
void swap(
   hash_multimap& _Right
);
```

Parameters

_Right

The hash_multimap providing the elements to be swapped or the hash_multimap whose elements are to be exchanged with those of the hash_multimap.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two hash_multimaps whose elements are being exchanged.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_swap.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1, hm2, hm3;
   hash_multimap <int, int>::iterator hm1_Iter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );
   hm2.insert ( Int_Pair ( 10, 100 ) );
   hm2.insert ( Int_Pair ( 20, 200 ) );
   hm3.insert ( Int_Pair ( 30, 300 ) );

   cout << "The original hash_multimap hm1 is:";
   for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++ )
      cout << " " << hm1_Iter -> second;
   cout   << "." << endl;

   // This is the member function version of swap
   hm1.swap( hm2 );

   cout << "After swapping with hm2, hash_multimap hm1 is:";
   for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++ )
      cout << " " << hm1_Iter -> second;
   cout  << "." << endl;

   // This is the specialized template version of swap
   swap( hm1, hm3 );

   cout << "After swapping with hm3, hash_multimap hm1 is:";
   for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++ )
      cout << " " << hm1_Iter -> second;
   cout   << "." << endl;
```

```
}
```

Output

```
The original hash_multimap hm1 is: 10 20 30.
After swapping with hm2, hash_multimap hm1 is: 100 200.
After swapping with hm3, hash_multimap hm1 is: 300.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::upper_bound

Returns an iterator to the first element in a hash_multimap with a key that is greater than a specified key.

```
iterator upper_bound(
   const Key& _Key
);
const_iterator upper_bound(
   const Key& _Key
) const;
```

Parameters

_Key

The argument key to be compared with the sort key of an element from the hash_multimap being searched.

Return Value

An iterator or const_iterator that addresses the location of an element in a hash_multimap with a key that is greater than the argument key, or that addresses the location succeeding the last element in the hash_multimap if no match is found for the key.

If the return value of **upper_bound** is assigned to a **const_iterator**, the hash_multimap object cannot be modified. If the return value of **upper_bound** is assigned to a **iterator**, the hash_multimap object can be modified.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_upper_bound.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;
   hash_multimap <int, int> :: const_iterator hm1_AcIter, hm1_RcIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );
   hm1.insert ( Int_Pair ( 3, 40 ) );

   hm1_RcIter = hm1.upper_bound( 1 );
   cout << "The 1st element of hash_multimap hm1 with "
        << "a key greater than 1 is: "
        << hm1_RcIter -> second << "." << endl;

   hm1_RcIter = hm1.upper_bound( 2 );
   cout << "The first element of hash_multimap hm1\n with a key "
        << " greater than 2 is: "
        << hm1_RcIter -> second << "." << endl;

   // If no match is found for the key, end( ) is returned
   hm1_RcIter = hm1.lower_bound( 4 );

   if ( hm1_RcIter == hm1.end( ) )
```

```
      cout << "The hash_multimap hm1 doesn't have an element "
           << "with a key of 4." << endl;
   else
      cout << "The element of hash_multimap hm1 with a key of 4 is: "
           << hm1_RcIter -> second << "." << endl;

   // The element at a specific location in the hash_multimap can be
   // found using a dereferenced iterator addressing the location
   hm1_AcIter = hm1.begin( );
   hm1_RcIter = hm1.upper_bound( hm1_AcIter -> first );
   cout << "The first element of hm1 with a key greater than"
        << endl << " that of the initial element of hm1 is: "
        << hm1_RcIter -> second << "." << endl;
}
```

Output

```
The 1st element of hash_multimap hm1 with a key greater than 1 is: 20.
The first element of hash_multimap hm1
 with a key  greater than 2 is: 40.
The hash_multimap hm1 doesn't have an element with a key of 4.
The first element of hm1 with a key greater than
 that of the initial element of hm1 is: 20.
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

hash_multimap::value_comp

The member function returns a function object that determines the order of elements in a hash_multimap by comparing their key values.

```
value_compare value_comp( ) const;
```

Return Value

Returns the comparison function object that a hash_multimap uses to order its elements.

Remarks

For a hash_multimap *m*, if two elements *e*1(*k*1, *d*1) and *e*2(*k*2, *d*2) are objects of type value_type, where *k*1 and *k*2 are their keys of type key_type and *d*1 and *d*2 are their data of type mapped_type, then *m*.**value_comp**( )(*e*1, *e*2) is equivalent to *m*.**key_comp**( ) (*k*1, *k*2). A stored object defines the member function

bool operator(**value_type&** *_Left*, **value_type&** *_Right*);

which returns **true** if the key value of *_Left* precedes and is not equal to the key value of *_Right* in the sort order.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multimap_value_comp.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;

   hash_multimap <int, int, hash_compare<int, less<int> > > hm1;
   hash_multimap <int, int, hash_compare<int, less<int> >
      >::value_compare vc1 = hm1.value_comp( );
   hash_multimap <int,int>::iterator Iter1, Iter2;

   Iter1= hm1.insert ( hash_multimap <int, int> :: value_type ( 1, 10 ) );
   Iter2= hm1.insert ( hash_multimap <int, int> :: value_type ( 2, 5 ) );

   if( vc1( *Iter1, *Iter2 ) == true )
   {
      cout << "The element ( 1,10 ) precedes the element ( 2,5 )."
           << endl;
   }
   else
   {
      cout << "The element ( 1,10 ) does "
           << "not precede the element ( 2,5 )."
           << endl;
   }

   if( vc1( *Iter2, *Iter1 ) == true )
   {
      cout << "The element ( 2,5 ) precedes the element ( 1,10 )."
           << endl;
   }
   else
   {
      cout << "The element ( 2,5 ) does "
```

```
            << "not precede the element ( 1,10 )."
            << endl;
    }
}
```

Output

```
The element ( 1,10 ) precedes the element ( 2,5 ).
The element ( 2,5 ) does not precede the element ( 1,10 ).
```

Requirements

Header: <hash_map>

See Also

Reference

hash_multimap Class

Standard Template Library

Other Resources

hash_multimap Members

<hash_set>

Defines the container template classes hash_set and hash_multiset and their supporting templates.

For a list of all members of this header, see <hash_set> Members.

```
#include <hash_set>
```

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<hash_set> Members

Standard C++ Library Header Files

<hash_set> Members

Reference

Operators

Hash_set version	Hash_multiset version	Description
operator!= (hash_set)	operator!= (hash_multiset)	Tests if the hash_set or hash_multiset object on the left side of the operator is not equal to the hash_set or hash_multiset object on the right side.
operator< (hash_set)	operator< (hash_multiset)	Tests if the hash_set or hash_multiset object on the left side of the operator is less than the hash_set or hash_multiset object on the right side.
operator<= (hash_set)	operator<= (hash_multiset)	Tests if the hash_set or hash_multiset object on the left side of the operator is less than or equal to the hash_set or hash_multiset object on the right side.
operator== (hash_set)	operator== (hash_multiset)	Tests if the hash_set or hash_multiset object on the left side of the operator is equal to the hash_set or hash_multiset object on the right side.
operator> (hash_set)	operator> (hash_multiset)	Tests if the hash_set or hash_multiset object on the left side of the operator is greater than the hash_set or hash_multiset object on the right side.
operator>= (hash_set)	operator>= (hash_multiset)	Tests if the hash_set or hash_multiset object on the left side of the operator is greater than or equal to the hash_set or hash_multiset object on the right side.

Specialized Template Functions

Hash_set version	Hash_multiset version	Description
swap (hash_set)	swap (hash_multiset)	Exchanges the elements of two hash_sets or hash_multisets.

Classes

hash_compare Class	Describes an object that can be used by any of the hash associative containers — hash_map, hash_multimap, hash_set, or hash_multiset — as a default **Traits** parameter object to order and hash the elements they contain.
hash_set Class	Used for the storage and fast retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values.
hash_multiset Class	Used for the storage and fast retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values.

See Also

Reference

<hash_set>
Thread Safety in the Standard C++ Library
Standard Template Library

<hash_set> Operators

For more information about the operators in <hash_set>, see <hash_set> Members.

See Also

Reference

Standard Template Library

operator!= (hash_set)

Tests if the hash_set object on the left side of the operator is not equal to the hash_set object on the right side.

```
bool operator!=(
   const hash_set <Key, Traits, Allocator>& _Left,
   const hash_set <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_set**.

_Right

An object of type **hash_set**.

Return Value

true if the hash_sets are not equal; **false** if hash_sets are equal.

Remarks

The comparison between hash_set objects is based on a pairwise comparison between their elements. Two hash_sets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_op_ne.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1, hs2, hs3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      hs1.insert ( i );
      hs2.insert ( i * i );
      hs3.insert ( i );
   }

   if ( hs1 != hs2 )
      cout << "The hash_sets hs1 and hs2 are not equal." << endl;
   else
      cout << "The hash_sets hs1 and hs2 are equal." << endl;

   if ( hs1 != hs3 )
      cout << "The hash_sets hs1 and hs3 are not equal." << endl;
   else
      cout << "The hash_sets hs1 and hs3 are equal." << endl;
}
```

Output

```
The hash_sets hs1 and hs2 are not equal.
```

```
The hash_sets hs1 and hs3 are equal.
```

Requirements

Header: <hash_set>

See Also

Reference

Standard Template Library

Other Resources

<hash_set> Members

operator< (hash_set)

Tests if the hash_set object on the left side of the operator is less than the hash_set object on the right side.

```
bool operator<(
   const hash_set <Key, Traits, Allocator>& _Left,
   const hash_set <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_set**.

_Right

An object of type **hash_set**.

Return Value

true if the hash_set on the left side of the operator is strictly less than the hash_set on the right side of the operator; otherwise **false**.

Remarks

The comparison between hash_set objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_op_lt.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1, hs2, hs3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      hs1.insert ( i );
      hs2.insert ( i * i );
      hs3.insert ( i - 1 );
   }

   if ( hs1 < hs2 )
      cout << "The hash_set hs1 is less than the hash_set hs2." << endl;
   else
      cout << "Hash_set hs1 is not less than hash_set hs2." << endl;

   if ( hs1 < hs3 )
      cout << "The hash_set hs1 is less than the hash_set hs3." << endl;
   else
      cout << "Hash_set hs1 is not less than hash_set hs3." << endl;
}
```

Output

```
The hash_set hs1 is less than the hash_set hs2.
Hash_set hs1 is not less than hash_set hs3.
```

Requirements

Header: <hash_set>

See Also

Reference

Standard Template Library

Other Resources

<hash_set> Members

operator<= (hash_set)

Tests if the hash_set object on the left side of the operator is less than or equal to the hash_set object on the right side.

```
bool operator!<=(
   const hash_set <Key, Traits, Allocator>& _Left,
   const hash_set <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_set**.

_Right

An object of type **hash_set**.

Return Value

true if the hash_set on the left side of the operator is less than or equal to the hash_set on the right side of the operator; otherwise **false**.

Remarks

The comparison between hash_set objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_op_le.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1, hs2, hs3, hs4;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      hs1.insert ( i );
      hs2.insert ( i * i );
      hs3.insert ( i - 1 );
      hs4.insert ( i );
   }

   if ( hs1 <= hs2 )
      cout << "Hash_set hs1 is less than or equal to hash_set hs2." << endl;
   else
      cout << "Hash_set hs1 is greater than hash_set hs2." << endl;

   if ( hs1 <= hs3 )
      cout << "Hash_set hs1 is less than or equal to hash_set hs3." << endl;
   else
      cout << "Hash_set hs1 is greater than hash_set hs3." << endl;

   if ( hs1 <= hs4 )
      cout << "Hash_set hs1 is less than or equal to hash_set hs4." << endl;
```

```
    else
       cout << "Hash_set hs1 is greater than hash_set hs4." << endl;
 }
```

Output

```
Hash_set hs1 is less than or equal to hash_set hs2.
Hash_set hs1 is greater than hash_set hs3.
Hash_set hs1 is less than or equal to hash_set hs4.
```

Requirements

Header: <hash_set>

See Also

Reference

Standard Template Library

Other Resources

<hash_set> Members

operator== (hash_set)

Tests if the hash_set object on the left side of the operator is equal to the hash_set object on the right side.

```
bool operator!==(
   const hash_set <Key, Traits, Allocator>& _Left,
   const hash_set <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_set**.

_Right

An object of type **hash_set**.

Return Value

true if the hash_set on the left side of the operator is equal to the hash_set on the right side of the operator; otherwise **false**.

Remarks

The comparison between hash_set objects is based on a pairwise comparison of their elements. Two hash_sets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_op_eq.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> s1, s2, s3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i );
   }

   if ( s1 == s2 )
      cout << "The hash_sets s1 and s2 are equal." << endl;
   else
      cout << "The hash_sets s1 and s2 are not equal." << endl;

   if ( s1 == s3 )
      cout << "The hash_sets s1 and s2 are equal." << endl;
   else
      cout << "The hash_sets s1 and s2 are not equal." << endl;
}
```

Output

```
The hash_sets s1 and s2 are not equal.
```

```
The hash_sets s1 and s2 are equal.
```

Requirements

Header: <hash_set>

See Also

Reference

Standard Template Library

Other Resources

<hash_set> Members

operator> (hash_set)

Tests if the hash_set object on the left side of the operator is greater than the hash_set object on the right side.

```
bool operator>(
   const hash_set <Key, Traits, Allocator>& _Left,
   const hash_set <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_set**.

_Right

An object of type **hash_set**.

Return Value

true if the hash_set on the left side of the operator is greater than the hash_set on the right side of the operator; otherwise **false**.

Remarks

The comparison between hash_set objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_op_gt.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1, hs2, hs3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      hs1.insert ( i );
      hs2.insert ( i * i );
      hs3.insert ( i - 1 );
   }

   if ( hs1 > hs2 )
      cout << "Hash_set hs1 is greater than hash_set hs2." << endl;
   else
      cout << "Hash_set hs1 is not greater than hash_set hs2." << endl;

   if ( hs1 > hs3 )
      cout << "Hash_set hs1 is greater than hash_set hs3." << endl;
   else
      cout << "Hash_set hs1 is not greater than hash_set hs3." << endl;
}
```

Output

```
Hash_set hs1 is not greater than hash_set hs2.
Hash_set hs1 is greater than hash_set hs3.
```

Requirements

Header: <hash_set>

See Also

Reference

Standard Template Library

Other Resources

<hash_set> Members

operator>= (hash_set)

Tests if the hash_set object on the left side of the operator is greater than or equal to the hash_set object on the right side.

```
bool operator!>=(
   const hash_set <Key, Traits, Allocator>& _Left,
   const hash_set <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_set**.

_Right

An object of type **hash_set**.

Return Value

true if the hash_set on the left side of the operator is greater than or equal to the hash_set on the right side of the list; otherwise **false**.

Remarks

The comparison between hash_set objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_op_ge.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1, hs2, hs3, hs4;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      hs1.insert ( i );
      hs2.insert ( i * i );
      hs3.insert ( i - 1 );
      hs4.insert ( i );
   }

   if ( hs1 >= hs2 )
      cout << "hs1 greater than or equal to hs2" << endl;
   else
      cout << "hs1 less than hs2" << endl;

   if ( hs1 >= hs3 )
      cout << "hs1 greater than or equal to hs3" << endl;
   else
      cout << "hs1 less than hs3" << endl;

   if ( hs1 >= hs4 )
      cout << "hs1 greater than or equal to hs4" << endl;
```

```
    else
       cout << "hs1 less than hs4" << endl;
 }
```

Output

```
hs1 less than hs2
hs1 greater than or equal to hs3
hs1 greater than or equal to hs4
```

Requirements

Header: <hash_set>

See Also

Reference

Standard Template Library

Other Resources

<hash_set> Members

operator!= (hash_multiset)

Tests if the hash_multiset object on the left side of the operator is not equal to the hash_multiset object on the right side.

```
bool operator!=(
   const hash_multiset <Key, Traits, Allocator>& _Left,
   const hash_multiset <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_multiset**.

_Right

An object of type **hash_multiset**.

Return Value

true if the hash_multisets are not equal; **false** if hash_multisets are equal.

Remarks

The comparison between hash_multiset objects is based on a pairwise comparison between their elements. Two hash_multisets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hashset_op_ne.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hs1, hs2, hs3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      hs1.insert ( i );
      hs2.insert ( i * i );
      hs3.insert ( i );
   }

   if ( hs1 != hs2 )
      cout << "The hash_multisets hs1 and hs2 are not equal." << endl;
   else
      cout << "The hash_multisets hs1 and hs2 are equal." << endl;

   if ( hs1 != hs3 )
      cout << "The hash_multisets hs1 and hs3 are not equal." << endl;
   else
      cout << "The hash_multisets hs1 and hs3 are equal." << endl;
}
```

Output

```
The hash_multisets hs1 and hs2 are not equal.
The hash_multisets hs1 and hs3 are equal.
```

Requirements

Header: <hash_set>

See Also

Reference

Standard Template Library

Other Resources

<hash_set> Members

operator< (hash_multiset)

Tests if the hash_multiset object on the left side of the operator is less than the hash_multiset object on the right side.

```
bool operator<(
   const hash_multiset <Key, Traits, Allocator>& _Left,
   const hash_multiset <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_multiset**.

_Right

An object of type **hash_multiset**.

Return Value

true if the hash_multiset on the left side of the operator is strictly less than the hash_multiset on the right side of the operator; otherwise **false**.

Remarks

The comparison between hash_multiset objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_multiset_op_lt.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hs1, hs2, hs3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      hs1.insert ( i );
      hs2.insert ( i * i );
      hs3.insert ( i - 1 );
   }

   if ( hs1 < hs2 )
      cout << "The hash_multiset hs1 is less than "
           << "the hash_multiset hs2." << endl;
   else
      cout << "The hash_multiset hs1 is not less than "
           << "the hash_multiset hs2." << endl;

   if ( hs1 < hs3 )
      cout << "The hash_multiset hs1 is less than "
           << "the hash_multiset hs3." << endl;
   else
      cout << "The hash_multiset hs1 is not less than "
           << "the hash_multiset hs3." << endl;
```

```
}
```

Output

```
The hash_multiset hs1 is less than the hash_multiset hs2.
The hash_multiset hs1 is not less than the hash_multiset hs3.
```

Requirements

Header: <hash_set>

See Also

Reference

Standard Template Library

Other Resources

<hash_set> Members

operator<= (hash_multiset)

Tests if the hash_multiset object on the left side of the operator is less than or equal to the hash_multiset object on the right side.

```
bool operator!<=(
   const hash_multiset <Key, Traits, Allocator>& _Left,
   const hash_multiset <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_multiset**.

_Right

An object of type **hash_multiset**.

Return Value

true if the hash_multiset on the left side of the operator is less than or equal to the hash_multiset on the right side of the operator; otherwise **false**.

Remarks

The comparison between hash_multiset objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_op_ne.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hs1, hs2, hs3, hs4;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      hs1.insert ( i );
      hs2.insert ( i * i );
      hs3.insert ( i - 1 );
      hs4.insert ( i );
   }

   if ( hs1 <= hs2 )
      cout << "The hash_multiset hs1 is less than "
           << "or equal to the hash_multiset hs2." << endl;
   else
      cout << "The hash_multiset hs1 is greater than "
           << "the hash_multiset hs2." << endl;

   if ( hs1 <= hs3 )
      cout << "The hash_multiset hs1 is less than "
           << "or equal to the hash_multiset hs3." << endl;
   else
```

```
      cout << "The hash_multiset hs1 is greater than "
           << "the hash_multiset hs3." << endl;

   if ( hs1 <= hs4 )
      cout << "The hash_multiset hs1 is less than "
           << "or equal to the hash_multiset hs4." << endl;
   else
      cout << "The hash_multiset hs1 is greater than "
           << "the hash_multiset hs4." << endl;
}
```

Output

```
The hash_multiset hs1 is less than or equal to the hash_multiset hs2.
The hash_multiset hs1 is greater than the hash_multiset hs3.
The hash_multiset hs1 is less than or equal to the hash_multiset hs4.
```

Requirements

Header: <hash_set>

See Also

Reference

Standard Template Library

Other Resources

<hash_set> Members

operator== (hash_multiset)

Tests if the hash_multiset object on the left side of the operator is equal to the hash_multiset object on the right side.

```
bool operator!==(
   const hash_multiset <Key, Traits, Allocator>& _Left,
   const hash_multiset <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_multiset**.

_Right

An object of type **hash_multiset**.

Return Value

true if the hash_multiset on the left side of the operator is equal to the hash_multiset on the right side of the operator; otherwise **false**.

Remarks

The comparison between hash_multiset objects is based on a pairwise comparison of their elements. Two hash_multisets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_op_eq.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> s1, s2, s3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i );
   }

   if ( s1 == s2 )
      cout << "The hash_multisets s1 and s2 are equal." << endl;
   else
      cout << "The hash_multisets s1 and s2 are not equal." << endl;

   if ( s1 == s3 )
      cout << "The hash_multisets s1 and s2 are equal." << endl;
   else
      cout << "The hash_multisets s1 and s2 are not equal." << endl;
}
```

Output

```
The hash_multisets s1 and s2 are not equal.
The hash_multisets s1 and s2 are equal.
```

Requirements

Header: <hash_set>

See Also

Reference

Standard Template Library

Other Resources

<hash_set> Members

operator> (hash_multiset)

Tests if the hash_multiset object on the left side of the operator is greater than the hash_multiset object on the right side.

```
bool operator>(
   const hash_multiset <Key, Traits, Allocator>& _Left,
   const hash_multiset <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_multiset**.

_Right

An object of type **hash_multiset**.

Return Value

true if the hash_multiset on the left side of the operator is greater than the hash_multiset on the right side of the operator; otherwise **false**.

Remarks

The comparison between hash_multiset objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_op_gt.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hs1, hs2, hs3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      hs1.insert ( i );
      hs2.insert ( i * i );
      hs3.insert ( i - 1 );
   }

   if ( hs1 > hs2 )
      cout << "The hash_multiset hs1 is greater than "
           << "the hash_multiset hs2." << endl;
   else
      cout << "The hash_multiset hs1 is not greater "
           << "than the hash_multiset hs2." << endl;

   if ( hs1 > hs3 )
      cout << "The hash_multiset hs1 is greater than "
           << "the hash_multiset hs3." << endl;
   else
      cout << "The hash_multiset hs1 is not greater than "
           << "the hash_multiset hs3." << endl;
```

```
}
```

Output

```
The hash_multiset hs1 is not greater than the hash_multiset hs2.
The hash_multiset hs1 is greater than the hash_multiset hs3.
```

Requirements

Header: <hash_set>

See Also

Reference

Standard Template Library

Other Resources

<hash_set> Members

operator>= (hash_multiset)

Tests if the hash_multiset object on the left side of the operator is greater than or equal to the hash_multiset object on the right side.

```
bool operator!>=(
   const hash_multiset <Key, Traits, Allocator>& _Left,
   const hash_multiset <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **hash_multiset**.

_Right

An object of type **hash_multiset**.

Return Value

true if the hash_multiset on the left side of the operator is greater than or equal to the hash_multiset on the right side of the list; otherwise **false**.

Remarks

The comparison between hash_multiset objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_op_ge.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hs1, hs2, hs3, hs4;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      hs1.insert ( i );
      hs2.insert ( i * i );
      hs3.insert ( i - 1 );
      hs4.insert ( i );
   }

   if ( hs1 >= hs2 )
      cout << "The hash_multiset hs1 is greater than "
           << "or equal to the hash_multiset hs2." << endl;
   else
      cout << "The hash_multiset hs1 is less than "
           << "the hash_multiset hs2." << endl;

   if ( hs1 >= hs3 )
      cout << "The hash_multiset hs1 is greater than "
           << "or equal to the hash_multiset hs3." << endl;
   else
```

```
      cout << "The hash_multiset hs1 is less than "
           << "the hash_multiset hs3." << endl;

   if ( hs1 >= hs4 )
      cout << "The hash_multiset hs1 is greater than "
           << "or equal to the hash_multiset hs4." << endl;
   else
      cout << "The hash_multiset hs1 is less than "
           << "the hash_multiset hs4." << endl;
}
```

Output

```
The hash_multiset hs1 is less than the hash_multiset hs2.
The hash_multiset hs1 is greater than or equal to the hash_multiset hs3.
The hash_multiset hs1 is greater than or equal to the hash_multiset hs4.
```

Requirements

Header: <hash_set>

See Also

Reference

Standard Template Library

Other Resources

<hash_set> Members

<hash_set> Specialized Template Functions

For more information about the specialized template functions in <hash_set>, see <hash_set> Members.

See Also

Reference

Standard Template Library

swap (hash_set)

Exchanges the elements of two hash_sets.

```
void swap(
   hash_multiset <Key, Traits, Allocator>& _Left,
   hash_multiset <Key, Traits, Allocator>& _Right
);
```

Parameters

_Right

The hash_set providing the elements to be swapped, or the hash_set whose elements are to be exchanged with those of the hash_set *_Left*.

_Left

The hash_set whose elements are to be exchanged with those of the hash_set *_Right*.

Remarks

The **swap** template function is an algorithm specialized on the container class hash_set to execute the member function *_Left*.swap(*_Right*). This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function

template <class T> void swap(T&, T&),

in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the code example for the member class hash_set::swap for an example that uses the template version of **swap**.

Requirements

Header: <hash_set>

See Also

Reference

Standard Template Library

Other Resources

<hash_set> Members

swap (hash_multiset)

Exchanges the elements of two hash_multisets.

```
void swap(
   hash_multiset <Key, Traits, Allocator>& _Left,
   hash_multiset <Key, Traits, Allocator>& _Right
);
```

Parameters

_Right

The hash_multiset providing the elements to be swapped, or the hash_multiset whose elements are to be exchanged with those of the hash_multiset *_Left*.

_Left

The hash_multiset whose elements are to be exchanged with those of the hash_multiset *_Right*.

Remarks

The **swap** template function is an algorithm specialized on the container class hash_multiset to execute the member function *_Left*.swap(*_Right*). This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function

template <class T> void swap(T&, T&),

in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the code example for the member class hash_multiset::swap for an example that uses the template version of **swap**.

Requirements

Header: <hash_set>

See Also

Reference

Standard Template Library

Other Resources

<hash_set> Members

<hash_set> Classes

For more information about the classes in <hash_set>, see <hash_set> Members.

See Also

Reference

Standard Template Library

hash_compare Class

The template class describes an object that can be used by any of the hash associative containers — hash_map, hash_multimap, hash_set, or hash_multiset — as a default **Traits** parameter object to order and hash the elements they contain.

```
template<class Key, class Traits = less<Key> >
   class hash_compare
   {
   Traits comp;
public:
   const size_t bucket_size = 4;
   const size_t min_buckets = 8;
   hash_compare( );
   hash_compare( Traits pred );
   size_t operator( )( const Key& Key ) const;
   bool operator( )(
      const Key& _Key1,
      const Key& _Key2
   ) const;
   };
```

Remarks

Each hash associative container stores a hash traits object of type **Traits** (a template parameter). You can derive a class from a specialization of hash_compare to selectively override certain functions and objects, or you can supply your own version of this class if you meet certain minimum requirements. Specifically, for an object hash_comp of type **hash_compare<Key, Traits>**, the following behavior is required by the above containers:

- For all values *_Key* of type **Key**, the call **hash_comp**(*_Key*) serves as a hash function, which yields a distribution of values of type **size_t**. The function supplied by hash_compare returns *_Key*.
- For any value *_Key1* of type **Key** that precedes *_Key2* in the sequence and has the same hash value (value returned by the hash function), **hash_comp**(*_Key2*, *_Key1*) is false. The function must impose a total ordering on values of type **Key**. The function supplied by hash_compare returns *comp*(*_Key2*, *_Key1*), where *comp* is a stored object of type **Traits** that you can specify when you construct the object hash_comp. For the default **Traits** parameter type **less<Key>**, sort keys never decrease in value.
- The integer constant **bucket_size** specifies the mean number of elements per "bucket" (hash-table entry) that the container should try not to exceed. It must be greater than zero. The value supplied by hash_compare is 4.
- The integer constant **min_buckets** specifies the minimum number of buckets to maintain in the hash table. It must be a power of two and greater than zero. The value supplied by hash_compare is 8.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See examples for hash_map::hash_map, hash_multimap::hash_multimap, hash_set::hash_set, and hash_multiset::hash_multiset, for examples of how to declare and use hash_compare.

Requirements

Header: <hash_map>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<hash_map> Members

<hash_set> Members

hash_set Class

The container class hash_set is an extension of the Standard Template Library (STL) and is used for the storage and fast retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values.

For a list of all members of this type, see hash_set Members.

```
template <
   class Key,
   class Traits=hash_compare<Key, less<Key> >,
      class Allocator=allocator<Key>
>
class hash_set
```

Parameters

Key

The element data type to be stored in the hash_set.

Traits

The type which includes two function objects, one of class compare that is a binary predicate able to compare two element values as sort keys to determine their relative order and a hash function that is a unary predicate mapping key values of the elements to unsigned integers of type **size_t**. This argument is optional, and the **hash_compare**<*Key*, **less**<*Key*> > is the default value.

Allocator

The type that represents the stored allocator object that encapsulates details about the hash_set's allocation and deallocation of memory. This argument is optional, and the default value is **allocator**<*Key*>.

Remarks

The hash_set is:

- An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value. Further, it is a simple associative container because its element values are its key values.
- Reversible, because it provides a bidirectional iterator to access its elements.
- Hashed, because its elements are grouped into buckets based on the value of a hash function applied to the key values of the elements.
- Unique in the sense that each of its elements must have a unique key. Because hash_set is also a simple associative container, its elements are also unique.
- A template class because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

The main advantage of hashing over sorting is greater efficiency; a successful hashing performs insertions, deletions, and finds in constant average time as compared with a time proportional to the logarithm of the number of elements in the container for sorting techniques. The value of an element in a set may not be changed directly. Instead, you must delete old values and insert elements with new values.

The choice of container type should be based in general on the type of searching and inserting required by the application. Hashed associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient when used with a well-designed hash function, performing them in a time that is on average constant and not dependent on the number of elements in the container. A well-designed hash function produces a uniform distribution of hashed values and minimizes the number of collisions, where a collision is said to occur when distinct key values are mapped into the same hashed value. In the worst case, with the worst possible hash function, the number of operations is proportional to the number of elements in the sequence (linear time).

The hash_set should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. The elements of a hash_set are unique and serve as their own sort keys. A model for this type of

structure is an ordered list of, say, words in which the words may occur only once. If multiple occurrences of the words were allowed, then a hash_multiset would be the appropriate container structure. If unique definitions were attached as values to the list of key words, then a hash_map would be an appropriate structure to contain this data. If instead the definitions were not unique, then a hash_multimap would be the container of choice.

The hash_set orders the sequence it controls by calling a stored hash **Traits** object of type value_compare. This stored object may be accessed by calling the member function key_comp. Such a function object must behave the same as an object of class *hash_compare<Key, less<Key> >*. Specifically, for all values *_Key* of type Key, the call Trait(*_Key*) yields a distribution of values of type size_t.

In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the non-equivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate *f*(*x*,*y*) is a function object that has two argument objects x and y and a return value of true or false. An ordering imposed on a hash_set is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects *x* and *y* are defined to be equivalent when both *f*(*x*,*y*) and *f*(*y*,*x*) are false. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

The actual order of elements in the controlled sequence depends on the hash function, the ordering function, and the current size of the hash table stored in the container object. You cannot determine the current size of the hash table, so you cannot in general predict the order of elements in the controlled sequence. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

The iterator provided by the hash_set class is a bidirectional iterator, but the class member functions insert and hash_set have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own hash_set of requirements, and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal hash_set of functionality, but it is enough to be able to talk meaningfully about a range of iterators [*_First*, *_Last*) in the context of the class member functions.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Requirements

Header: <hash_set>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<hash_set> Members

hash_set Members

hash_set Members

Reference

Constructors

hash_set	Constructs a **hash_set** that is empty or that is a copy of all or part of some other **hash_set**.

Typedefs

allocator_type	A type that represents the **allocator** class for the **hash_set** object.
const_iterator	A type that provides a bidirectional iterator that can read a **const** element in the **hash_set**.
const_pointer	A type that provides a pointer to a **const** element in a **hash_set**.
const_reference	A type that provides a reference to a **const** element stored in a **hash_set** for reading and performing **c onst** operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any **const** element in the **hash_set**.
difference_type	A signed integer type that can be used to represent the number of elements of a **hash_set** in a range be tween elements pointed to by iterators.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a **hash_set**.
key_compare	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the **hash_set**.
key_type	A type that describes an object stored as an element of a **hash_set** in its capacity as sort key.
pointer	A type that provides a pointer to an element in a **hash_set**.
reference	A type that provides a reference to an element stored in a **hash_set**.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed **hash_set**.
size_type	An unsigned integer type that can represent the number of elements in a **hash_set**.
value_compare	A type that provides two function objects, a binary predicate of class compare that can compare two ele ment values of a **hash_set** to determine their relative order and a unary predicate that hashes the elem ents.
value_type	A type that describes an object stored as an element of a **hash_set** in its capacity as a value.

Member Functions

begin	Returns an iterator that addresses the first element in the **hash_set**.
clear	Erases all the elements of a **hash_set**.
count	Returns the number of elements in a **hash_set** whose key matches a parameter-specified key.
empty	Tests if a **hash_set** is empty.
end	Returns an iterator that addresses the location succeeding the last element in a **hash_set**.

equal_range	Returns a pair of iterators respectively to the first element in a **hash_set** with a key that is greater than a specified key and to the first element in the **hash_set** with a key that is equal to or greater than the key.
erase	Removes an element or a range of elements in a **hash_set** from specified positions or removes elements that match a specified key.
find	Returns an iterator addressing the location of an element in a **hash_set** that has a key equivalent to a specified key.
get_allocator	Returns a copy of the **allocator** object used to construct the **hash_set**.
insert	Inserts an element or a range of elements into a **hash_set**.
key_comp	Retrieves a copy of the comparison object used to order keys in a **hash_set**.
lower_bound	Returns an iterator to the first element in a **hash_set** with a key that is equal to or greater than a specified key.
max_size	Returns the maximum length of the **hash_set**.
rbegin	Returns an iterator addressing the first element in a reversed **hash_set**.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed **hash_set**.
size	Returns the number of elements in the **hash_set**.
swap	Exchanges the elements of two **hash_set**s.
upper_bound	Returns an iterator to the first element in a **hash_set** that with a key that is equal to or greater than a specified key.
value_comp	Retrieves a copy of the hash traits object used to hash and order element key values in a **hash_set**.

See Also

Reference

hash_set Class
Thread Safety in the Standard C++ Library
Standard Template Library

hash_set Typedefs

For information about the typedefs in the hash_set class, see hash_set Members.

See Also

Reference

Standard Template Library

hash_set::allocator_type

A type that represents the allocator class for the hash_set object.

```
typedef Allocator allocator_type
```

Remarks

allocator_type is a synonym for the template parameter **Allocator**.

For more information on **Allocator**, see the Remarks section of the hash_set Class topic.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See example for get_allocator for an example that uses **allocator_type**.

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::const_iterator

A type that provides a bidirectional iterator that can read a **const** element in the hash_set.

```
typedef implementation-defined const_iterator;
```

Remarks

A type **const_iterator** cannot be used to modify the value of an element.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See example for begin for an example that uses **const_iterator**.

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::const_pointer

A type that provides a pointer to a **const** element in a hash_set.

```
typedef typename Allocator::const_pointer const_pointer;
```

Remarks

A type **const_pointer** cannot be used to modify the value of an element.

In most cases, a const_iterator should be used to access the elements in a **const** hash_set object.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::const_reference

A type that provides a reference to a **const** element stored in a hash_set for reading and performing **const** operations.

```
typedef typename Allocator::const_reference const_reference;
```

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_const_ref.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;

   hs1.insert( 10 );
   hs1.insert( 20 );

   // Declare and initialize a const_reference &Ref1
   // to the 1st element
   const int &Ref1 = *hs1.begin( );

   cout << "The first element in the hash_set is "
        << Ref1 << "." << endl;

   // The following line would cause an error because the
   // const_reference cannot be used to modify the hash_set
   // Ref1 = Ref1 + 5;
}
```

Output

```
The first element in the hash_set is 10.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::const_reverse_iterator

A type that provides a bidirectional iterator that can read any **const** element in the hash_set.

```
typedef reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type **const_reverse_iterator** cannot modify the value of an element and is use to iterate through the hash_set in reverse.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for rend for an example of how to declare and use the **const_reverse_iterator**

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::difference_type

A signed integer type that can be used to represent the number of elements of a hash_set in a range between elements pointed to by iterators.

```
typedef implementation-defined difference_type;
```

Remarks

The **difference_type** is the type returned when subtracting or incrementing through iterators of the container. The **difference_type** is typically used to represent the number of elements in the range [*_First*, *_Last*) between the iterators *_First* and *_Last*, includes the element pointed to by *_First* and the range of elements up to, but not including, the element pointed to by *_Last*.

Note that although **difference_type** is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as set, subtraction between iterators is only supported by random-access iterators provided by a random access container, such as vector or deque.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_diff_type.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <iostream>
#include <hash_set>
#include <algorithm>

int main( )
{
   using namespace std;
   using namespace stdext;

   hash_set <int> hs1;
   hash_set <int>::iterator hs1_Iter, hs1_bIter, hs1_eIter;

   hs1.insert( 20 );
   hs1.insert( 10 );
   hs1.insert( 20 );   // Won't insert as hash_set elements are unique

   hs1_bIter = hs1.begin( );
   hs1_eIter = hs1.end( );

   hash_set <int>::difference_type   df_typ5, df_typ10, df_typ20;

   df_typ5 = count( hs1_bIter, hs1_eIter, 5 );
   df_typ10 = count( hs1_bIter, hs1_eIter, 10 );
   df_typ20 = count( hs1_bIter, hs1_eIter, 20 );

   // The keys, and hence the elements, of a hash_set are unique,
   // so there is at most one of a given value
   cout << "The number '5' occurs " << df_typ5
        << " times in hash_set hs1.\n";
   cout << "The number '10' occurs " << df_typ10
        << " times in hash_set hs1.\n";
   cout << "The number '20' occurs " << df_typ20
        << " times in hash_set hs1.\n";

   // Count the number of elements in a hash_set
   hash_set <int>::difference_type  df_count = 0;
   hs1_Iter = hs1.begin( );
   while ( hs1_Iter != hs1_eIter)
```

```
    {
       df_count++;
       hs1_Iter++;
    }

    cout << "The number of elements in the hash_set hs1 is: "
         << df_count << "." << endl;
 }
```

Output

```
The number '5' occurs 0 times in hash_set hs1.
The number '10' occurs 1 times in hash_set hs1.
The number '20' occurs 1 times in hash_set hs1.
The number of elements in the hash_set hs1 is: 2.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::iterator

A type that provides a bidirectional iterator that can read or modify any element in a hash_set.

```
typedef implementation-defined iterator;
```

Remarks

A type **iterator** can be used to modify the value of an element.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for begin for an example of how to declare and use **iterator**.

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::key_compare

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the hash_set.

```
typedef Traits key_compare;
```

Remarks

key_compare is a synonym for the template parameter **Traits**.

For more information on **Traits** see the hash_set Class topic.

Note that both **key_compare** and value_compare are synonyms for the template parameter **Traits**. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for key_comp for an example of how to declare and use **key_compare**.

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::key_type

A type that describes an object stored as an element of a hash_set in its capacity as sort key.

```
typedef Key key_type;
```

Remarks

key_type is a synonym for the template parameter **Key**.

For more information on **Key**, see the Remarks section of the hash_set Class topic.

Note that both **key_type** and value_type are synonyms for the template parameter **Key**. Both types are provided for the hash_set and hash_multiset classes, where they are identical, for compatibility with the hash_map and hash_multimap classes, where they are distinct.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for value_type for an example of how to declare and use **key_type**.

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::pointer

A type that provides a pointer to an element in a hash_set.

```
typedef typename Allocator::pointer pointer;
```

Remarks

A type **pointer** can be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a hash_set object.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::reference

A type that provides a reference to an element stored in a hash_set.

```
typedef typename Allocator::reference reference;
```

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_reference.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;

   hs1.insert( 10 );
   hs1.insert( 20 );

   // Declare and initialize a reference &Ref1 to the 1st element
   int &Ref1 = *hs1.begin( );

   cout << "The first element in the hash_set is "
        << Ref1 << "." << endl;

   // The value of the 1st element of the hash_set can be changed
   // by operating on its (non-const) reference
   Ref1 = Ref1 + 5;

   cout << "The first element in the hash_set is now "
        << *hs1.begin() << "." << endl;
}
```

Output

```
The first element in the hash_set is 10.
The first element in the hash_set is now 15.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::reverse_iterator

A type that provides a bidirectional iterator that can read or modify an element in a reversed hash_set.

```
typedef reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type **reverse_iterator** is use to iterate through the hash_set in reverse.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for rbegin for an example of how to declare and use **reverse_iterator**.

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::size_type

An unsigned integer type that can represent the number of elements in a hash_set.

```
typedef implementation-defined size_type;
```

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for size for an example of how to declare and use **size_type**

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::value_compare

A type that provides two function objects, a binary predicate of class compare that can compare two element values of a hash_set to determine their relative order and a unary predicate that hashes the elements.

```
typedef Traits value_compare;
```

Remarks

value_compare is a synonym for the template parameter **Traits**.

For more information on **Traits** see the hash_set Class topic.

Note that both key_compare and **value_compare** are synonyms for the template parameter **Traits**. Both types are provided for the hash_set and hash_multiset classes, where they are identical, for compatibility with the hash_map and hash_multimap classes, where they are distinct.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for value_comp for an example of how to declare and use **value_compare**.

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::value_type

A type that describes an object stored as an element of a hash_set in its capacity as a value.

```
typedef Key value_type;
```

Remark

value_type is a synonym for the template parameter **Key**.

Note that both key_type and **value_type** are synonyms for the template parameter **Key**. Both types are provided for the set and hash_set classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

For more information on **Key**, see the Remarks section of the hash_set Class topic.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_value_type.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int>::iterator hs1_Iter;

   hash_set <int> :: value_type hsvt_Int;   // Declare value_type
   hsvt_Int = 10;             // Initialize value_type

   hash_set <int> :: key_type hskt_Int;   // Declare key_type
   hskt_Int = 20;             // Initialize key_type

   hs1.insert( hsvt_Int );         // Insert value into hs1
   hs1.insert( hskt_Int );         // Insert key into hs1

   // A hash_set accepts key_types or value_types as elements
   cout << "The hash_set has elements:";
   for ( hs1_Iter = hs1.begin( ) ; hs1_Iter != hs1.end( ); hs1_Iter++)
      cout << " " << *hs1_Iter;
   cout << "." << endl;
}
```

Output

```
The hash_set has elements: 10 20.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set Member Functions

For information about the functions in the hash_set class, see hash_set Members.

See Also

Reference

Standard Template Library

hash_set::begin

Returns an iterator that addresses the first element in the hash_set.

```
const_iterator begin( ) const;
iterator begin( );
```

Return Value

A bidirectional iterator addressing the first element in the hash_set or the location succeeding an empty hash_set.

Remarks

If the return value of **begin** is assigned to a **const_iterator**, the elements in the hash_set object cannot be modified. If the return value of **begin** is assigned to an **iterator**, the elements in the hash_set object can be modified.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_begin.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int>::iterator hs1_Iter;
   hash_set <int>::const_iterator hs1_cIter;

   hs1.insert( 1 );
   hs1.insert( 2 );
   hs1.insert( 3 );

   hs1_Iter = hs1.begin( );
   cout << "The first element of hs1 is " << *hs1_Iter << endl;

   hs1_Iter = hs1.begin( );
   hs1.erase( hs1_Iter );

   // The following 2 lines would err because the iterator is const
   // hs1_cIter = hs1.begin( );
   // hs1.erase( hs1_cIter );

   hs1_cIter = hs1.begin( );
   cout << "The first element of hs1 is now " << *hs1_cIter << endl;
}
```

Output

```
The first element of hs1 is 1
The first element of hs1 is now 2
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

hash_set::clear

Erases all the elements of a hash_set.

```
void clear( );
```

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_clear.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;

   hs1.insert( 1 );
   hs1.insert( 2 );

   cout << "The size of the hash_set is initially " << hs1.size( )
        << "." << endl;

   hs1.clear( );
   cout << "The size of the hash_set after clearing is "
        << hs1.size( ) << "." << endl;
}
```

Output

```
The size of the hash_set is initially 2.
The size of the hash_set after clearing is 0.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::count

Returns the number of elements in a hash_set whose key matches a parameter-specified key.

```
size_type count(
   const Key& _Key
) const;
```

Parameters

_Key

The key of the elements to be matched from the hash_set.

Return Value

1 if the hash_set contains an element whose sort key matches the parameter key.

0 if the hash_set does not contain an element with a matching key.

Remarks

The member function returns the number of elements in the following range:

[**lower_bound** (*_Key*), **upper_bound** (*_Key*)).

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// hash_set_count.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set<int> hs1;
    hash_set<int>::size_type i;

    hs1.insert(1);
    hs1.insert(1);

    // Keys must be unique in hash_set, so duplicates are ignored.
    i = hs1.count(1);
    cout << "The number of elements in hs1 with a sort key of 1 is: "
         << i << "." << endl;

    i = hs1.count(2);
    cout << "The number of elements in hs1 with a sort key of 2 is: "
         << i << "." << endl;
}
```

Output

```
The number of elements in hs1 with a sort key of 1 is: 1.
The number of elements in hs1 with a sort key of 2 is: 0.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::empty

Tests if a hash_set is empty.

```
bool empty( ) const;
```

Return Value

true if the hash_set is empty; **false** if the hash_set is nonempty.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_empty.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1, hs2;
   hs1.insert ( 1 );

   if ( hs1.empty( ) )
      cout << "The hash_set hs1 is empty." << endl;
   else
      cout << "The hash_set hs1 is not empty." << endl;

   if ( hs2.empty( ) )
      cout << "The hash_set hs2 is empty." << endl;
   else
      cout << "The hash_set hs2 is not empty." << endl;
}
```

Output

```
The hash_set hs1 is not empty.
The hash_set hs2 is empty.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::end

Returns an iterator that addresses the location succeeding the last element in a hash_set.

```
const_iterator end( ) const;
iterator end( );
```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a hash_set. If the hash_set is empty, then hash_set::end == hash_set::begin.

Remarks

end is used to test whether an iterator has reached the end of its hash_set. The value returned by **end** should not be dereferenced.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_end.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int> :: iterator hs1_Iter;
   hash_set <int> :: const_iterator hs1_cIter;

   hs1.insert( 1 );
   hs1.insert( 2 );
   hs1.insert( 3 );

   hs1_Iter = hs1.end( );
   hs1_Iter--;
   cout << "The last element of hs1 is " << *hs1_Iter << endl;

   hs1.erase( hs1_Iter );

   // The following 3 lines would err because the iterator is const:
   // hs1_cIter = hs1.end( );
   // hs1_cIter--;
   // hs1.erase( hs1_cIter );

   hs1_cIter = hs1.end( );
   hs1_cIter--;
   cout << "The last element of hs1 is now " << *hs1_cIter << endl;
}
```

Output

```
The last element of hs1 is 3
The last element of hs1 is now 2
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::equal_range

Returns a pair of iterators respectively to the first element in a hash_set with a key that is greater than a specified key and to the first element in the hash_set with a key that is equal to or greater than the key.

```
pair <const_iterator, const_iterator> equal_range (
   const Key& _Key
) const;
pair <iterator, iterator> equal_range (
   const Key& _Key
);
```

Parameters

_Key

The argument key to be compared with the sort key of an element from the hash_set being searched.

Return Value

A pair of iterators where the first is the lower_bound of the key and the second is the upper_bound of the key.

To access the first iterator of a pair pr returned by the member function, use *pr*.**first**, and to dereference the lower bound iterator, use *(*pr*.**first**). To access the second iterator of a pair *pr* returned by the member function, use *pr*.**second**, and to dereference the upper bound iterator, use *(*pr*.**second**).

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_equal_range.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   typedef hash_set<int> IntHSet;
   IntHSet hs1;
   hash_set <int> :: const_iterator hs1_RcIter;

   hs1.insert( 10 );
   hs1.insert( 20 );
   hs1.insert( 30 );

   pair <IntHSet::const_iterator, IntHSet::const_iterator> p1, p2;
   p1 = hs1.equal_range( 20 );

   cout << "The upper bound of the element with "
        << "a key of 20 in the hash_set hs1 is: "
        << *(p1.second) << "." << endl;

   cout << "The lower bound of the element with "
        << "a key of 20 in the hash_set hs1 is: "
        << *(p1.first) << "." << endl;

   // Compare the upper_bound called directly
   hs1_RcIter = hs1.upper_bound( 20 );
   cout << "A direct call of upper_bound( 20 ) gives "
        << *hs1_RcIter << "," << endl
```

```
         << "matching the 2nd element of the pair"
         << " returned by equal_range( 20 )." << endl;

   p2 = hs1.equal_range( 40 );

   // If no match is found for the key,
   // both elements of the pair return end( )
   if ( ( p2.first == hs1.end( ) ) && ( p2.second == hs1.end( ) ) )
      cout << "The hash_set hs1 doesn't have an element "
           << "with a key less than 40." << endl;
   else
      cout << "The element of hash_set hs1 with a key >= 40 is: "
           << *(p1.first) << "." << endl;
}
```

Output

```
The upper bound of the element with a key of 20 in the hash_set hs1 is: 30.
The lower bound of the element with a key of 20 in the hash_set hs1 is: 20.
A direct call of upper_bound( 20 ) gives 30,
matching the 2nd element of the pair returned by equal_range( 20 ).
The hash_set hs1 doesn't have an element with a key less than 40.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::erase

Removes an element or a range of elements in a hash_set from specified positions or removes elements that match a specified key.

```
iterator erase(
   iterator _Where
);
iterator erase(
   iterator _First,
   iterator _Last
);
size_type erase(
   const key_type& _Key
);
```

Parameters

_Where

Position of the element to be removed from the hash_set.

_First

Position of the first element removed from the hash_set.

_Last

Position just beyond the last element removed from the hash_set.

_Key

The key of the elements to be removed from the hash_set.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the hash_set if no such element exists. For the third member function, the number of elements that have been removed from the hash_set.

Remarks

The member functions never throw an exception.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// hash_set_erase.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_set<int> hs1, hs2, hs3;
    hash_set<int>::iterator pIter, Iter1, Iter2;
    int i;
    hash_set<int>::size_type n;
```

```
    for (i = 1; i < 5; i++)
    {
        hs1.insert (i);
        hs2.insert (i * i);
        hs3.insert (i - 1);
    }

    // The 1st member function removes an element at a given position
    Iter1 = ++hs1.begin();
    hs1.erase(Iter1);

    cout << "After the 2nd element is deleted, the hash_set hs1 is:";
    for (pIter = hs1.begin(); pIter != hs1.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;

    // The 2nd member function removes elements
    // in the range [_First, _Last)
    Iter1 = ++hs2.begin();
    Iter2 = --hs2.end();
    hs2.erase(Iter1, Iter2);

    cout << "After the middle two elements are deleted, "
         << "the hash_set hs2 is:";
    for (pIter = hs2.begin(); pIter != hs2.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;

    // The 3rd member function removes elements with a given _Key
    n = hs3.erase(2);

    cout << "After the element with a key of 2 is deleted, "
         << "the hash_set hs3 is:";
    for (pIter = hs3.begin(); pIter != hs3.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;

    // The 3rd member function returns the number of elements removed
    cout << "The number of elements removed from hs3 is: "
         << n << "." << endl;

    // The dereferenced iterator can also be used to specify a key
    Iter1 = ++hs3.begin();
    hs3.erase(Iter1);

    cout << "After another element (unique for hash_set) with a key "
         << endl;
    cout  << "equal to that of the 2nd element is deleted, "
          << "the hash_set hs3 is:";
    for (pIter = hs3.begin(); pIter != hs3.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;
}
```

Output

```
After the 2nd element is deleted, the hash_set hs1 is: 1 3 4.
After the middle two elements are deleted, the hash_set hs2 is: 1 16.
After the element with a key of 2 is deleted, the hash_set hs3 is: 0 1 3.
The number of elements removed from hs3 is: 1.
After another element (unique for hash_set) with a key
equal to that of the 2nd element is deleted, the hash_set hs3 is: 0 3.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class
Standard Template Library
Other Resources
hash_set Members

hash_set::find

Returns an iterator addressing the location of an element in a hash_set that has a key equivalent to a specified key.

```
iterator find(
   const Key& _Key
);
const_iterator find(
   const Key& _Key
) const;
```

Parameters

_Key

The argument key to be matched by the sort key of an element from the hash_set being searched.

Return Value

An **iterator** or **const_iterator** that addresses the location of an element equivalent to a specified key or that addresses the location succeeding the last element in the hash_set if no match is found for the key.

Remarks

The member function returns an iterator that addresses an element in the hash_set whose sort key is **equivalent** to the argument key under a binary predicate that induces an ordering based on a less-than comparability relation.

If the return value of **find** is assigned to a **const_iterator**, the hash_set object cannot be modified. If the return value of **find** is assigned to an **iterator**, the hash_set object can be modified.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_find.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int> :: const_iterator hs1_AcIter, hs1_RcIter;

   hs1.insert( 10 );
   hs1.insert( 20 );
   hs1.insert( 30 );

   hs1_RcIter = hs1.find( 20 );
   cout << "The element of hash_set hs1 with a key of 20 is: "
        << *hs1_RcIter << "." << endl;

   hs1_RcIter = hs1.find( 40 );

   // If no match is found for the key, end( ) is returned
   if ( hs1_RcIter == hs1.end( ) )
      cout << "The hash_set hs1 doesn't have an element "
           << "with a key of 40." << endl;
   else
      cout << "The element of hash_set hs1 with a key of 40 is: "
           << *hs1_RcIter << "." << endl;

   // The element at a specific location in the hash_set can be found
```

```
    // by using a dereferenced iterator addressing the location
    hs1_AcIter = hs1.end( );
    hs1_AcIter--;
    hs1_RcIter = hs1.find( *hs1_AcIter );
    cout << "The element of hs1 with a key matching "
         << "that of the last element is: "
         << *hs1_RcIter << "." << endl;
}
```

Output

```
The element of hash_set hs1 with a key of 20 is: 20.
The hash_set hs1 doesn't have an element with a key of 40.
The element of hs1 with a key matching that of the last element is: 30.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::get_allocator

Returns a copy of the allocator object used to construct the hash_set.

```
Allocator get_allocator( ) const;
```

Return Value

The allocator used by the hash_set to manage memory, which is the template parameter **Allocator**.

For more information on **Allocator**, see the Remarks section of the hash_set Class topic.

Remarks

Allocators for the hash_set class specify how the class manages storage. The default allocators supplied with STL container classes is sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_get_allocator.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;

   // The following lines declare objects
   // that use the default allocator.
   hash_set <int, hash_compare <int, less<int> > > hs1;
   hash_set <int,  hash_compare <int, greater<int> > > hs2;
   hash_set <double, hash_compare <double,
      less<double> >, allocator<double> > hs3;

   hash_set <int, hash_compare <int,
      greater<int> > >::allocator_type hs2_Alloc;
   hash_set <double>::allocator_type hs3_Alloc;
   hs2_Alloc = hs2.get_allocator( );

   cout << "The number of integers that can be allocated"
        << endl << "before free memory is exhausted: "
        << hs1.max_size( ) << "." << endl;

   cout << "The number of doubles that can be allocated"
        << endl << "before free memory is exhausted: "
        << hs3.max_size( ) <<  "." << endl;

   // The following lines create a hash_set hs4
   // with the allocator of hash_set hs1.
   hash_set <int>::allocator_type hs4_Alloc;
   hash_set <int> hs4;
   hs4_Alloc = hs2.get_allocator( );

   // Two allocators are interchangeable if
   // storage allocated from each can be
   // deallocated by the other
   if( hs2_Alloc == hs4_Alloc )
   {
      cout << "The allocators are interchangeable."
```

```
            << endl;
    }
    else
    {
       cout << "The allocators are not interchangeable."
            << endl;
    }
}
```

Sample Output

The following output is for x86.

```
The number of integers that can be allocated
before free memory is exhausted: 1073741823.
The number of doubles that can be allocated
before free memory is exhausted: 536870911.
The allocators are interchangeable.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::hash_set

Constructs a hash_set that is empty or that is a copy of all or part of some other hash_set.

```
hash_set( );
explicit hash_set(
   const Traits& _Comp
);
explicit hash_set(
   const Traits& _Comp,
   const Allocator& _Al
);
hash_set(
   const _hash_set& _Right
);
template<class InputIterator>
   hash_set(
      InputIterator _First,
      InputIterator _Last
   );
template<class InputIterator>
   hash_set(
      InputIterator _First,
      InputIterator _Last,
      const Traits& _Comp
   );
template<class InputIterator>
   hash_set(
      InputIterator _First,
      InputIterator _Last,
      const Traits& _Comp,
      const Allocator& _Al
   );
```

Parameters

_Al

The storage allocator class to be used for this hash_set object, which defaults to **Allocator**.

_Comp

The comparison function of type **const Traits** used to order the elements in the hash_set, which defaults to **hash_compare**.

_Right

The hash_set of which the constructed hash_set is to be a copy.

_First

The position of the first element in the range of elements to be copied.

_Last

The position of the first element beyond the range of elements to be copied.

Remarks

All constructors store a type of allocator object that manages memory storage for the hash_set and that can later be returned by calling get_allocator. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their hash_sets.

All constructors store a function object of type **Traits** that is used to establish an order among the keys of the hash_set and that can later be returned by calling key_comp. For more information on **Traits** see the hash_set Class topic.

The first three constructors specify an empty initial hash_set, the second specifying the type of comparison function (*_Comp*) to

be used in establishing the order of the elements and the third explicitly specifying the allocator type (*_Al*) to be used. The key word **explicit** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the hash_set *_Right*.

The last three constructors copy the range [*_First*, *_Last*) of a hash_set with increasing explicitness in specifying the type of comparison function of class Traits and allocator.

The actual order of elements in a hash_set container depends on the hash function, the ordering function and the current size of the hash table and cannot, in general, be predicted as it could with the set container, where it was determined by the ordering function alone.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_hash_set.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int>::iterator hs1_Iter, hs3_Iter, hs4_Iter,
      hs5_Iter, hs6_Iter;
   hash_set <int, hash_compare <int, greater<int> > >::iterator
      hs2_Iter;

   // Create an empty hash_set hs0 of key type integer
   hash_set <int> hs0;

   // Create an empty hash_set hs1 with the key comparison
   // function of less than, then insert 4 elements
   hash_set <int, hash_compare <int, less<int> > > hs1;
   hs1.insert( 10 );
   hs1.insert( 20 );
   hs1.insert( 30 );
   hs1.insert( 40 );

   // Create an empty hash_set hs2 with the key comparison
   // function of geater than, then insert 2 elements
   hash_set <int, hash_compare <int, greater<int> > > hs2;
   hs2.insert(10);
   hs2.insert(20);

   // Create a hash_set hs3 with the
   // allocator of hash_set hs1
   hash_set <int>::allocator_type hs1_Alloc;
   hs1_Alloc = hs1.get_allocator( );
   hash_set <int> hs3( hash_compare <int, less<int> >( ),
      hs1_Alloc );
   hs3.insert( 30 );

   // Create a copy, hash_set hs4, of hash_set hs1
   hash_set <int> hs4( hs1 );

   // Create a hash_set hs5 by copying the range hs1[_First, _Last)
   hash_set <int>::const_iterator hs1_bcIter, hs1_ecIter;
   hs1_bcIter = hs1.begin( );
   hs1_ecIter = hs1.begin( );
   hs1_ecIter++;
   hs1_ecIter++;
   hash_set <int> hs5( hs1_bcIter, hs1_ecIter );

   // Create a hash_set hs6 by copying the range hs4[_First, _Last)
```

```
    // and with the allocator of hash_set hs2
    hash_set <int>::allocator_type hs2_Alloc;
    hs2_Alloc = hs2.get_allocator( );
    hash_set <int> hs6( hs4.begin( ), ++hs4.begin( ),
       less<int>( ), hs2_Alloc );

    cout << "hs1 = ";
    for ( hs1_Iter = hs1.begin( ); hs1_Iter != hs1.end( );
          hs1_Iter++ )
       cout << *hs1_Iter << " ";
    cout << endl;

    cout << "hs2 = " ;
    for ( hs2_Iter = hs2.begin( ); hs2_Iter != hs2.end( );
          hs2_Iter++ )
       cout << *hs2_Iter << " ";
    cout << endl;

    cout << "hs3 = ";
    for ( hs3_Iter = hs3.begin( ); hs3_Iter != hs3.end( );
          hs3_Iter++ )
       cout << *hs3_Iter << " ";
    cout << endl;

    cout << "hs4 = ";
    for ( hs4_Iter = hs4.begin( ); hs4_Iter != hs4.end( );
          hs4_Iter++ )
       cout << *hs4_Iter << " ";
    cout << endl;

    cout << "hs5 = ";
    for ( hs5_Iter = hs5.begin( ); hs5_Iter != hs5.end( );
          hs5_Iter++ )
       cout << *hs5_Iter << " ";
    cout << endl;

    cout << "hs6 = ";
    for ( hs6_Iter = hs6.begin( ); hs6_Iter != hs6.end( );
          hs6_Iter++ )
       cout << *hs6_Iter << " ";
    cout << endl;
  }
```

Output

```
hs1 = 10 20 30 40
hs2 = 20 10
hs3 = 30
hs4 = 10 20 30 40
hs5 = 10 20
hs6 = 10
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::insert

Inserts an element or a range of elements into a **hash_set**.

```
pair<iterator, bool> insert(
   const value_type& _Val
);
iterator insert(
   iterator _Where,
   const value_type& _Val
);
template<class InputIterator>
   void insert(
      InputIterator _First,
      InputIterator _Last
   );
```

Parameters

_Val

The value of an element to be inserted into the **hash_set** unless the **hash_set** already contains that element or, more generally, an element whose key is equivalently ordered.

_Where

The place to start searching for the correct point of insertion. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows *_Where*.)

_First

The position of the first element to be copied from a **hash_set**.

_Last

The position just beyond the last element to be copied from a **hash_set**.

Return Value

The first **insert** member function returns a pair whose **bool** component returns **true** if an insertion was make and **false** if the **hash_set** already contained an element whose key had an equivalent value in the ordering, and whose iterator component returns the address where a new element was inserted or where the element was already located.

To access the iterator component of a pair *pr* returned by this member function, use *pr*.**first** and to dereference it, use *(*pr*.**first**). To access the **bool** component of a pair *pr* returned by this member function, use *pr*.**second**, and to dereference it, use *(*pr*.**second**).

The second **insert** member function returns an iterator that points to the position where the new element was inserted into the **hash_set**.

Remarks

The third member function inserts the sequence of element values into a **hash_set** corresponding to each element addressed by an iterator of in the range [*_First*, *_Last*) of a specified **hash_set**.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_insert.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
```

```
    using namespace std;
    using namespace stdext;
    hash_set <int>::iterator hs1_pIter, hs2_pIter;

    hash_set <int, hash_compare <int, less<int> > > hs1, hs2;
    hs1.insert( 10 );
    hs1.insert( 20 );
    hs1.insert( 30 );
    hs1.insert( 40 );

    cout << "The original hs1 =";
    for ( hs1_pIter = hs1.begin( ); hs1_pIter != hs1.end( );
          hs1_pIter++ )
       cout << " " << *hs1_pIter;
    cout << "." << endl;

    pair< hash_set<int>::iterator, bool > pr;
    pr = hs1.insert( 10 );

    if(pr.second == true)
    {
       cout << "The element 10 was inserted in hs1 successfully."
            << endl;
    }
    else
    {
       cout << "The element 10 already exists in hs1 and"
            << " *( pr.first ) = " << *( pr.first ) << "."
            << endl;
    }

    hs1.insert( --hs1.end( ), 50 );

    cout << "After the insertions, hs1 =";
    for ( hs1_pIter = hs1.begin( ); hs1_pIter != hs1.end( );
          hs1_pIter++ )
       cout << " " << *hs1_pIter;
    cout << "." << endl;

    hs2.insert( 100 );
    hs2.insert( ++hs1.begin( ), --hs1.end( ) );

    cout << "hs2 =";
    for ( hs2_pIter = hs2.begin( ); hs2_pIter != hs2.end( );
          hs2_pIter++ )
       cout << " " << *hs2_pIter;
    cout << "." << endl;
 }
```

Output

```
The original hs1 = 10 20 30 40.
The element 10 already exists in hs1 and *( pr.first ) = 10.
After the insertions, hs1 = 10 20 30 40 50.
hs2 = 20 30 40 100.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::key_comp

Retrieves a copy of the hash traits object used to hash and order element key values in a hash_set.

```
key_compare key_comp( ) const;
```

Return Value

Returns the function object that a hash_set uses to order its elements, which is the template parameter **Traits**.

For more information on **Traits** see the hash_set Class topic.

Remarks

The stored object defines the member function:

bool operator(**const Key&** *_xVal*, **const Key&** *_yVal*);

which returns **true** if *_xVal* precedes and is not equal to *_yVal* in the sort order.

Note that both key_compare and value_compare are synonyms for the template parameter **Traits**. Both types are provided for the hash_set and hash_multiset classes, where they are identical, for compatibility with the hash_map and hash_multimap classes, where they are distinct.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_key_comp.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;

   hash_set <int, hash_compare < int, less<int> > >hs1;
   hash_set<int, hash_compare < int, less<int> > >::key_compare kc1
          = hs1.key_comp( ) ;
   bool result1 = kc1( 2, 3 ) ;
   if( result1 == true )
   {
      cout << "kc1( 2,3 ) returns value of true, "
           << "where kc1 is the function object of hs1."
           << endl;
   }
   else
   {
      cout << "kc1( 2,3 ) returns value of false "
           << "where kc1 is the function object of hs1."
        << endl;
   }

   hash_set <int, hash_compare < int, greater<int> > > hs2;
   hash_set<int, hash_compare < int, greater<int> > >::key_compare
         kc2 = hs2.key_comp( ) ;
   bool result2 = kc2( 2, 3 ) ;
   if(result2 == true)
   {
      cout << "kc2( 2,3 ) returns value of true, "
           << "where kc2 is the function object of hs2."
           << endl;
```

```
    }
    else
    {
       cout << "kc2( 2,3 ) returns value of false, "
            << "where kc2 is the function object of hs2."
            << endl;
    }
 }
```

Output

```
kc1( 2,3 ) returns value of true, where kc1 is the function object of hs1.
kc2( 2,3 ) returns value of false, where kc2 is the function object of hs2.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::lower_bound

Returns an iterator to the first element in a hash_set with a key that is equal to or greater than a specified key.

```
const_iterator lower_bound(
   const Key& _Key
) const;
iterator lower_bound(
   const Key& _Key
);
```

Parameters

_Key

The argument key to be compared with the sort key of an element from the hash_set being searched.

Return Value

An **iterator** or **const_iterator** that addresses the location of an element in a hash_set that with a key that is equal to or greater than the argument key or that addresses the location succeeding the last element in the hash_set if no match is found for the key.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_lower_bound.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int> :: const_iterator hs1_AcIter, hs1_RcIter;

   hs1.insert( 10 );
   hs1.insert( 20 );
   hs1.insert( 30 );

   hs1_RcIter = hs1.lower_bound( 20 );
   cout << "The element of hash_set hs1 with a key of 20 is: "
        << *hs1_RcIter << "." << endl;

   hs1_RcIter = hs1.lower_bound( 40 );

   // If no match is found for the key, end( ) is returned
   if ( hs1_RcIter == hs1.end( ) )
      cout << "The hash_set hs1 doesn't have an element "
           << "with a key of 40." << endl;
   else
      cout << "The element of hash_set hs1 with a key of 40 is: "
           << *hs1_RcIter << "." << endl;

   // An element at a specific location in the hash_set can be found
   // by using a dereferenced iterator that addresses the location
   hs1_AcIter = hs1.end( );
   hs1_AcIter--;
   hs1_RcIter = hs1.lower_bound( *hs1_AcIter );
```

```
    cout << "The element of hs1 with a key matching "
         << "that of the last element is: "
         << *hs1_RcIter << "." << endl;
}
```

Output

```
The element of hash_set hs1 with a key of 20 is: 20.
The hash_set hs1 doesn't have an element with a key of 40.
The element of hs1 with a key matching that of the last element is: 30.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::max_size

Returns the maximum length of the hash_set.

```
size_type max_size( ) const;
```

Return Value

The maximum possible length of the hash_set.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_max_size.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int>::size_type i;

   i = hs1.max_size( );
   cout << "The maximum possible length "
        << "of the hash_set is " << i << "." << endl;
}
```

Sample Output

The following output is for x86.

```
The maximum possible length of the hash_set is 1073741823.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::rbegin

Returns an iterator addressing the first element in a reversed hash_set.

```
const_reverse_iterator rbegin( ) const;
reverse_iterator rbegin( );
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed hash_set or addressing what had been the last element in the unreversed hash_set.

Remarks

rbegin is used with a reversed hash_set just as begin is used with a hash_set.

If the return value of **rbegin** is assigned to a **const_reverse_iterator**, then the hash_set object cannot be modified. If the return value of **rbegin** is assigned to a **reverse_iterator**, then the hash_set object can be modified.

rbegin can be used to iterate through a hash_set backwards.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_rbegin.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int>::iterator hs1_Iter;
   hash_set <int>::reverse_iterator hs1_rIter;

   hs1.insert( 10 );
   hs1.insert( 20 );
   hs1.insert( 30 );

   hs1_rIter = hs1.rbegin( );
   cout << "The first element in the reversed hash_set is "
        << *hs1_rIter << "." << endl;

   // begin can be used to start an iteration
   // throught a hash_set in a forward order
   cout << "The hash_set is: ";
   for ( hs1_Iter = hs1.begin( ) ; hs1_Iter != hs1.end( );
         hs1_Iter++ )
      cout << *hs1_Iter << " ";
   cout << endl;

   // rbegin can be used to start an iteration
   // throught a hash_set in a reverse order
   cout << "The reversed hash_set is: ";
   for ( hs1_rIter = hs1.rbegin( ) ; hs1_rIter != hs1.rend( );
         hs1_rIter++ )
      cout << *hs1_rIter << " ";
   cout << endl;

   // A hash_set element can be erased by dereferencing to its key
```

```
    hs1_rIter = hs1.rbegin( );
    hs1.erase ( *hs1_rIter );

    hs1_rIter = hs1.rbegin( );
    cout << "After the erasure, the first element "
         << "in the reversed hash_set is "<< *hs1_rIter << "."
         << endl;
}
```

Output

```
The first element in the reversed hash_set is 30.
The hash_set is: 10 20 30
The reversed hash_set is: 30 20 10
After the erasure, the first element in the reversed hash_set is 20.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::rend

Returns an iterator that addresses the location succeeding the last element in a reversed hash_set.

```
const_reverse_iterator rend( ) const;
reverse_iterator rend( );
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed hash_set (the location that had preceded the first element in the unreversed hash_set).

Remarks

rend is used with a reversed hash_set just as end is used with a hash_set.

If the return value of **rend** is assigned to a **const_reverse_iterator**, then the hash_set object cannot be modified. If the return value of **rend** is assigned to a **reverse_iterator**, then the hash_set object can be modified. The value returned by **rend** should not be dereferenced.

rend can be used to test to whether a reverse iterator has reached the end of its hash_set.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_rend.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int>::iterator hs1_Iter;
   hash_set <int>::reverse_iterator hs1_rIter;
   hash_set <int>::const_reverse_iterator hs1_crIter;

   hs1.insert( 10 );
   hs1.insert( 20 );
   hs1.insert( 30 );

   hs1_rIter = hs1.rend( );
   hs1_rIter--;
   cout << "The last element in the reversed hash_set is "
        << *hs1_rIter << "." << endl;

   // end can be used to terminate an iteration
   // throught a hash_set in a forward order
   cout << "The hash_set is: ";
   for ( hs1_Iter = hs1.begin( ) ; hs1_Iter != hs1.end( );
         hs1_Iter++ )
      cout << *hs1_Iter << " ";
   cout << "." << endl;

   // rend can be used to terminate an iteration
   // through a hash_set in a reverse order
   cout << "The reversed hash_set is: ";
   for ( hs1_rIter = hs1.rbegin( ) ; hs1_rIter != hs1.rend( );
         hs1_rIter++ )
      cout << *hs1_rIter << " ";
```

```
    cout << "." << endl;

    hs1_rIter = hs1.rend( );
    hs1_rIter--;
    hs1.erase ( *hs1_rIter );

    hs1_rIter = hs1.rend( );
    hs1_rIter--;
    cout << "After the erasure, the last element in the "
         << "reversed hash_set is " << *hs1_rIter << "."
         << endl;
}
```

Output

```
The last element in the reversed hash_set is 10.
The hash_set is: 10 20 30 .
The reversed hash_set is: 30 20 10 .
After the erasure, the last element in the reversed hash_set is 20.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::size

Returns the number of elements in the hash_set.

```
size_type size( ) const;
```

Return Value

The current length of the hash_set.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_size.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int> :: size_type i;

   hs1.insert( 1 );
   i = hs1.size( );
   cout << "The hash_set length is " << i << "." << endl;

   hs1.insert( 2 );
   i = hs1.size( );
   cout << "The hash_set length is now " << i << "." << endl;
}
```

Output

```
The hash_set length is 1.
The hash_set length is now 2.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::swap

Exchanges the elements of two hash_sets.

```
void swap(
   hash_set& _Right
);
```

Parameters

_Right

The argument hash_set providing the elements to be swapped with the target hash_set.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two hash_sets whose elements are being exchanged.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_swap.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1, hs2, hs3;
   hash_set <int>::iterator hs1_Iter;

   hs1.insert( 10 );
   hs1.insert( 20 );
   hs1.insert( 30 );
   hs2.insert( 100 );
   hs2.insert( 200 );
   hs3.insert( 300 );

   cout << "The original hash_set hs1 is:";
   for ( hs1_Iter = hs1.begin( ); hs1_Iter != hs1.end( );
         hs1_Iter++ )
         cout << " " << *hs1_Iter;
   cout   << "." << endl;

   // This is the member function version of swap
   hs1.swap( hs2 );

   cout << "After swapping with hs2, list hs1 is:";
   for ( hs1_Iter = hs1.begin( ); hs1_Iter != hs1.end( );
         hs1_Iter++ )
         cout << " " << *hs1_Iter;
   cout  << "." << endl;

   // This is the specialized template version of swap
   swap( hs1, hs3 );

   cout << "After swapping with hs3, list hs1 is:";
   for ( hs1_Iter = hs1.begin( ); hs1_Iter != hs1.end( );
         hs1_Iter++ )
         cout << " " << *hs1_Iter;
```

```
    cout   << "." << endl;
}
```

Output

```
The original hash_set hs1 is: 10 20 30.
After swapping with hs2, list hs1 is: 100 200.
After swapping with hs3, list hs1 is: 300.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::upper_bound

Returns an iterator to the first element in a hash_set that with a key that is greater than a specified key.

```
const_iterator upper_bound(
   const Key& _Key
) const;
iterator upper_bound(
   const Key& _Key
);
```

Parameters

_Key

The argument key to be compared with the sort key of an element from the hash_set being searched.

Return Value

An **iterator** or **const_iterator** that addresses the location of an element in a hash_set that with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the hash_set if no match is found for the key.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_upper_bound.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int> :: const_iterator hs1_AcIter, hs1_RcIter;

   hs1.insert( 10 );
   hs1.insert( 20 );
   hs1.insert( 30 );

   hs1_RcIter = hs1.upper_bound( 20 );
   cout << "The first element of hash_set hs1 with a key greater "
        << "than 20 is: " << *hs1_RcIter << "." << endl;

   hs1_RcIter = hs1.upper_bound( 30 );

   // If no match is found for the key, end( ) is returned
   if ( hs1_RcIter == hs1.end( ) )
      cout << "The hash_set hs1 doesn't have an element "
           << "with a key greater than 30." << endl;
   else
      cout << "The element of hash_set hs1 with a key > 40 is: "
           << *hs1_RcIter << "." << endl;

   // An element at a specific location in the hash_set can be found
   // by using a dereferenced iterator addressing the location
   hs1_AcIter = hs1.begin( );
   hs1_RcIter = hs1.upper_bound( *hs1_AcIter );
   cout << "The first element of hs1 with a key greater than "
```

```
        << endl << "that of the initial element of hs1 is: "
        << *hs1_RcIter << "." << endl;
}
```

Output

```
The first element of hash_set hs1 with a key greater than 20 is: 30.
The hash_set hs1 doesn't have an element with a key greater than 30.
The first element of hs1 with a key greater than
that of the initial element of hs1 is: 20.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_set::value_comp

Retrieves a copy of the comparison object used to order element values in a hash_set.

```
value_compare value_comp( ) const;
```

Return Value

Returns the function object that a hash_set uses to order its elements, which is the template parameter **Compare**.

For more information on **Compare**, see the Remarks section of the hash_set Class topic.

Remarks

The stored object defines the member function:

bool operator(**const Key&** *_xVal*, **const Key&** *_yVal*);

which returns **true** if *_xVal* precedes and is not equal to *_yVal* in the sort order.

Note that both value_compare and key_compare are synonyms for the template parameter **Compare**. Both types are provided for the hash_set and hash_multiset classes, where they are identical, for compatibility with the hash_map and hash_multimap classes, where they are distinct.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_set_value_comp.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;

   hash_set <int, hash_compare < int, less<int> > > hs1;
   hash_set <int, hash_compare < int, less<int> >  >::value_compare
      vc1 = hs1.value_comp( );
   bool result1 = vc1( 2, 3 );
   if( result1 == true )
   {
      cout << "vc1( 2,3 ) returns value of true, "
           << "where vc1 is the function object of hs1."
           << endl;
   }
   else
   {
      cout << "vc1( 2,3 ) returns value of false, "
           << "where vc1 is the function object of hs1."
           << endl;
   }

   hash_set <int, hash_compare < int, greater<int> > > hs2;
   hash_set<int, hash_compare < int, greater<int> > >::value_compare
      vc2 = hs2.value_comp( );
   bool result2 = vc2( 2, 3 );
   if( result2 == true )
   {
      cout << "vc2( 2,3 ) returns value of true, "
           << "where vc2 is the function object of hs2."
           << endl;
```

```
    }
    else
    {
       cout << "vc2( 2,3 ) returns value of false, "
            << "where vc2 is the function object of hs2."
            << endl;
    }
 }
```

Output

```
vc1( 2,3 ) returns value of true, where vc1 is the function object of hs1.
vc2( 2,3 ) returns value of false, where vc2 is the function object of hs2.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_set Class

Standard Template Library

Other Resources

hash_set Members

hash_multiset Class

The container class hash_multiset is an extension of the Standard Template Library and is used for the storage and fast retrieval of data from a collection in which the values of the elements contained serve as the key values and are not required to be unique.

For a list of all members of this type, see hash_multiset Members.

```
template <
   class Key,
   class Traits=hash_compare<Key, less<Key> >,
      class Allocator=allocator<Key>
>
class hash_multiset
```

Parameters

Key

The element data type to be stored in the hash_multiset.

Traits

The type which includes two function objects, one of class compare that is a binary predicate able to compare two element values as sort keys to determine their relative order and a hash function that is a unary predicate mapping key values of the elements to unsigned integers of type **size_t**. This argument is optional, and the **hash_compare**<*Key*, **less**<*Key*> > is the default value.

Allocator

The type that represents the stored allocator object that encapsulates details about the hash_multiset's allocation and deallocation of memory. This argument is optional, and the default value is **allocator**<*Key*>.

Remarks

The hash_multiset is:

- An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value. Further, it is a simple associative container because its element values are its key values.
- Reversible, because it provides a bidirectional iterator to access its elements.
- Hashed, because its elements are grouped into buckets based on the value of a hash function applied to the key values of the elements.
- Unique in the sense that each of its elements must have a unique key. Because hash_multiset is also a simple associative container, its elements are also unique.
- A template class because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

The main advantage of hashing over sorting is greater efficiency: a successful hashing performs insertions, deletions, and finds in constant average time as compared with a time proportional to the logarithm of the number of elements in the container for sorting techniques. The value of an element in a set may not be changed directly. Instead, you must delete old values and insert elements with new values.

The choice of container type should be based in general on the type of searching and inserting required by the application. Hashed associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient when used with a well-designed hash function, performing them in a time that is on average constant and not dependent on the number of elements in the container. A well-designed hash function produces a uniform distribution of hashed values and minimizes the number of collisions, where a collision is said to occur when distinct key values are mapped into the same hashed value. In the worst case, with the worst possible hash function, the number of operations is proportional to the number of elements in the sequence (linear time).

The hash_multiset should be the associative container of choice when the conditions associating the values with their keys are

satisfies by the application. The elements of a hash_multiset may be multiple and serve as their own sort keys, so keys are not unique. A model for this type of structure is an ordered list of, say, words in which the words may occur more than once. Had multiple occurrences of the words not been allowed, then a hash_set would have been the appropriate container structure. If unique definitions were attached as values to the list of unique keywords, then a hash_map would be an appropriate structure to contain this data. If instead the definitions were not unique, then a hash_multimap would be the container of choice.

The hash_multiset orders the sequence it controls by calling a stored hash traits object of type value_compare. This stored object may be accessed by calling the member function key_comp. Such a function object must behave the same as an object of class **hash_compare**<*Key*, **less**<*Key*> >. Specifically, for all values *Key* of type **Key**, the call **Trait**(*Key*) yields a distribution of values of type **size_t**.

In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate *f*(*x*,*y*) is a function object that has two argument objects x and y and a return value of true or false. An ordering imposed on a hash_multiset is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects x and y are defined to be equivalent when both *f*(*x*,*y*) and *f*(*y*,*x*) are false. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

The actual order of elements in the controlled sequence depends on the hash function, the ordering function, and the current size of the hash table stored in the container object. You cannot determine the current size of the hash table, so you cannot in general predict the order of elements in the controlled sequence. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

The iterator provided by the hash_multiset class is a bidirectional iterator, but the class member functions insert and hash_multiset have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own hash_multiset of requirements, and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal hash_multiset of functionality, but it is enough to be able to talk meaningfully about a range of iterators [*_First*, *_Last*) in the context of the class member functions.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Requirements

Header: <hash_set>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<hash_set> Members

hash_multiset Members

hash_multiset Members

Reference

Constructors

hash_multiset	Constructs a **hash_multiset** that is empty or that is a copy of all or part of some other **hash_multiset**.

Typedefs

allocator_type	A type that represents the **allocator** class for the **hash_multiset** object.
const_iterator	A type that provides a bidirectional iterator that can read a **const** element in the **hash_multiset**.
const_pointer	A type that provides a pointer to a **const** element in a **hash_multiset**.
const_reference	A type that provides a reference to a **const** element stored in a **hash_multiset** for reading and perform ing **const** operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any **const** element in the **hash_multiset**.
difference_type	A signed integer type that provides the difference between two iterators that address elements within t he same **hash_multiset**.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a **hash_multiset**.
key_compare	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the **hash_multiset**.
key_type	A type that provides a function object that can compare sort keys to determine the relative order of two elements in the **hash_multiset**.
pointer	A type that provides a pointer to an element in a **hash_multiset**.
reference	A type that provides a reference to an element stored in a **hash_multiset**.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed **hash_mul tiset**.
size_type	An unsigned integer type that can represent the number of elements in a **hash_multiset**.
value_compare	A type that provides two function objects, a binary predicate of class compare that can compare two ele ment values of a **hash_multiset** to determine their relative order and a unary predicate that hashes the elements.
value_type	A type that describes an object stored as an element of a **hash_multiset** in its capacity as a value.

Member Functions

begin	Returns an iterator that addresses the first element in the **hash_multiset**.
clear	Erases all the elements of a **hash_multiset**.
count	Returns the number of elements in a **hash_multiset** whose key matches a parameter-specified key
empty	Tests if a **hash_multiset** is empty.

end	Returns an iterator that addresses the location succeeding the last element in a **hash_multiset**.
equal_range	Returns a pair of iterators respectively to the first element in a **hash_multiset** with a key that is greater than a s pecified key and to the first element in the **hash_multiset** with a key that is equal to or greater than the key.
erase	Removes an element or a range of elements in a **hash_multiset** from specified positions or removes elements t hat match a specified key.
find	Returns an iterator addressing the location of an element in a **hash_multiset** that has a key equivalent to a spec ified key.
get_allocator	Returns a copy of the **allocator** object used to construct the **hash_multiset**.
insert	Inserts an element or a range of elements into a **hash_multiset**.
key_comp	Retrieves a copy of the comparison object used to order keys in a **hash_multiset**.
lower_bound	Returns an iterator to the first element in a **hash_multiset** with a key that is equal to or greater than a specified key.
max_size	Returns the maximum length of the **hash_multiset**.
rbegin	Returns an iterator addressing the first element in a reversed **hash_multiset**.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed **hash_multiset**.
size	Returns the number of elements in the **hash_multiset**.
swap	Exchanges the elements of two **hash_multiset**s.
upper_bound	Returns an iterator to the first element in a **hash_multiset** that with a key that is equal to or greater than a speci fied key.
value_comp	Retrieves a copy of the hash traits object used to hash and order element key values in a **hash_multiset**.

See Also

Reference

hash_multiset Class
Thread Safety in the Standard C++ Library
Standard Template Library

hash_multiset Typedefs

For information about the typedefs in the hash_multiset class, see hash_multiset Members.

See Also

Reference

Standard Template Library

hash_multiset::allocator_type

A type that represents the allocator class for the hash_multiset object.

```
typedef Allocator allocator_type
```

Remark

allocator_type is a synonym for the template parameter **Allocator**.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

For more information on **Allocator**, see the Remarks section of the hash_multiset Class.

Example

See example for get_allocator for an example using **allocator_type**

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::const_iterator

A type that provides a bidirectional iterator that can read a **const** element in the hash_multiset.

```
typedef implementation-defined const_iterator;
```

Remarks

A type **const_iterator** cannot be used to modify the value of an element.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See example for begin for an example using **const_iterator**.

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::const_pointer

A type that provides a pointer to a **const** element in a hash_multiset.

```
typedef typename Allocator::const_pointer const_pointer;
```

Remarks

A type **const_pointer** cannot be used to modify the value of an element.

In most cases, a const_iterator should be used to access the elements in a **const** hash_multiset object.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::const_reference

A type that provides a reference to a **const** element stored in a hash_multiset for reading and performing **const** operations.

```
typedef typename Allocator::const_reference const_reference;
```

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_const_reference.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;

   hms1.insert( 10 );
   hms1.insert( 20 );

   // Declare and initialize a const_reference &Ref1
   // to the 1st element
   const int &Ref1 = *hms1.begin( );

   cout << "The first element in the hash_multiset is "
        << Ref1 << "." << endl;

   // The following line would cause an error because the
   // const_reference cannot be used to modify the hash_multiset
   // Ref1 = Ref1 + 5;
}
```

Output

```
The first element in the hash_multiset is 10.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::const_reverse_iterator

A type that provides a bidirectional iterator that can read any **const** element in the hash_multiset.

```
typedef reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type **const_reverse_iterator** cannot modify the value of an element and is use to iterate through the hash_multiset in reverse.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See the example for rend for an example of how to declare and use the **const_reverse_iterator**.

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::difference_type

A signed integer type that provides the difference between two iterators that address elements within the same hash_multiset.

```
typedef implementation-defined difference_type;
```

Remarks

The **difference_type** is the type returned when subtracting or incrementing through iterators of the container. The **difference_type** is typically used to represent the number of elements in the range [*_First*, *_Last*) between the iterators *_First* and *_Last*, includes the element pointed to by *_First* and the range of elements up to, but not including, the element pointed to by *_Last*.

Note that although **difference_type** is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as set. Subtraction between iterators is only supported by random-access iterators provided by a random-access container such as vector or deque.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_diff_type.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <iostream>
#include <hash_set>
#include <algorithm>

int main( )
{
   using namespace std;
   using namespace stdext;

   hash_multiset <int> hms1;
   hash_multiset <int>::iterator hms1_Iter, hms1_bIter, hms1_eIter;

   hms1.insert( 20 );
   hms1.insert( 10 );

   // hash_multiset elements need not be unique
   hms1.insert( 20 );

   hms1_bIter = hms1.begin( );
   hms1_eIter = hms1.end( );

   hash_multiset <int>::difference_type   df_typ5, df_typ10,
        df_typ20;

   df_typ5 = count( hms1_bIter, hms1_eIter, 5 );
   df_typ10 = count( hms1_bIter, hms1_eIter, 10 );
   df_typ20 = count( hms1_bIter, hms1_eIter, 20 );

   // The keys & hence the elements of a hash_multiset
   // need not be unique and may occur multiple times
   cout << "The number '5' occurs " << df_typ5
        << " times in hash_multiset hms1.\n";
   cout << "The number '10' occurs " << df_typ10
        << " times in hash_multiset hms1.\n";
   cout << "The number '20' occurs " << df_typ20
        << " times in hash_multiset hms1.\n";

   // Count the number of elements in a hash_multiset
   hash_multiset <int>::difference_type  df_count = 0;
```

```
    hms1_Iter = hms1.begin( );
    while ( hms1_Iter != hms1_eIter)
    {
       df_count++;
       hms1_Iter++;
    }

    cout << "The number of elements in the hash_multiset hms1 is "
         << df_count << "." << endl;
 }
```

Output

```
The number '5' occurs 0 times in hash_multiset hms1.
The number '10' occurs 1 times in hash_multiset hms1.
The number '20' occurs 2 times in hash_multiset hms1.
The number of elements in the hash multiset hms1 is 3.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::iterator

A type that provides a bidirectional iterator that can read or modify any element in a hash_multiset.

```
typedef implementation-defined iterator;
```

Remarks

A type **iterator** can be used to modify the value of an element.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See example for begin for an example of how to declare and use **iterator**.

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::key_compare

A type that provides two function objects, a binary predicate of class compare that can compare two element values of a hash_multiset to determine their relative order and a unary predicate that hashes the elements.

```
typedef Traits key_compare;
```

Remarks

key_compare is a synonym for the template parameter **Traits**.

For more information on **Traits** see the hash_multiset Class topic.

Note that both **key_compare** and value_compare are synonyms for the template parameter **Traits**. Both types are provided for the hash_set and hash_multiset classes, where they are identical, for compatibility with the hash_map and hash_multimap classes, where they are distinct.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See example for key_comp for an example of how to declare and use **key_compare**.

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::key_type

A type that provides a function object that can compare sort keys to determine the relative order of two elements in the hash_multiset.

```
typedef Key key_type;
```

Remarks

key_type is a synonym for the template parameter **Key**.

Note that both **key_type** and value_type are synonyms for the template parameter **Key**. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

For more information on **Key**, see the Remarks section of the hash_multiset Class topic.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See example for value_type for an example of how to declare and use **key_type**.

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::pointer

A type that provides a pointer to an element in a hash_multiset.

```
typedef typename Allocator::pointer pointer;
```

Remarks

A type **pointer** can be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a multiset object.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::reference

A type that provides a reference to an element stored in a hash_multiset.

```
typedef typename Allocator::reference reference;
```

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_reference.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;

   hms1.insert( 10 );
   hms1.insert( 20 );

   // Declare and initialize a reference &Ref1 to the 1st element
   int &Ref1 = *hms1.begin( );

   cout << "The first element in the hash_multiset is "
        << Ref1 << "." << endl;

   // The value of the 1st element of the hash_multiset can be
   // changed by operating on its (non const) reference
   Ref1 = Ref1 + 5;

   cout << "The first element in the hash_multiset is now "
        << *hms1.begin() << "." << endl;
}
```

Output

```
The first element in the hash_multiset is 10.
The first element in the hash_multiset is now 15.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::reverse_iterator

A type that provides a bidirectional iterator that can read or modify an element in a reversed hash_multiset.

```
typedef reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type **reverse_iterator** is use to iterate through the hash_multiset in reverse.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See example for rbegin for an example of how to declare and use **reverse_iterator**.

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::size_type

An unsigned integer type that can represent the number of elements in a hash_multiset.

```
typedef implementation-defined size_type;
```

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See example for size for an example of how to declare and use **size_type**

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::value_compare

A type that provides two function objects, a binary predicate of class compare that can compare two element values of a hash_multiset to determine their relative order and a unary predicate that hashes the elements.

```
typedef Traits value_compare;
```

Remarks

value_compare is a synonym for the template parameter **Traits**.

For more information on **Traits** see the hash_multiset Class topic.

Note that both key_compare and **value_compare** are synonyms for the template parameter **Traits**. Both types are provided for the classes set and multiset, where they are identical, for compatibility with the classes map and multimap, where they are distinct.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

See example for value_comp for an example of how to declare and use **value_compare**.

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::value_type

A type that describes an object stored as an element as a hash_multiset in its capacity as a value.

```
typedef Key value_type;
```

Remark

value_type is a synonym for the template parameter **Key**.

For more information on **Key**, see the Remarks section of the hash_multiset Class topic.

Note that both key_type and **value_type** are synonyms for the template parameter **Key**. Both types are provided for the classes hash_set and hash_multiset, where they are identical, for compatibility with the classes hash_map and hash_multimap, where they are distinct.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_value_type.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int>::iterator hms1_Iter;

   // Declare value_type
   hash_multiset <int> :: value_type hmsvt_Int;

   hmsvt_Int = 10;   // Initialize value_type

   // Declare key_type
   hash_multiset <int> :: key_type hmskt_Int;
   hmskt_Int = 20;             // Initialize key_type

   hms1.insert( hmsvt_Int );         // Insert value into s1
   hms1.insert( hmskt_Int );         // Insert key into s1

   // A hash_multiset accepts key_types or value_types as elements
   cout << "The hash_multiset has elements:";
   for ( hms1_Iter = hms1.begin() ; hms1_Iter != hms1.end( );
         hms1_Iter++)
      cout << " " << *hms1_Iter;
      cout << "." << endl;
}
```

Output

```
The hash_multiset has elements: 10 20.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

hash_multiset Member Functions

For information about the functions in the hash_multiset class, see hash_multiset Members.

See Also

Reference

Standard Template Library

hash_multiset::begin

Returns an iterator that addresses the first element in the hash_multiset.

```
const_iterator begin( ) const;
iterator begin( );
```

Return Value

A bidirectional iterator addressing the first element in the hash_multiset or the location succeeding an empty hash_multiset.

Remarks

If the return value of **begin** is assigned to a **const_iterator**, the elements in the hash_multiset object cannot be modified. If the return value of **begin** is assigned to an **iterator**, the elements in the hash_multiset object can be modified.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_begin.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int>::iterator hms1_Iter;
   hash_multiset <int>::const_iterator hms1_cIter;

   hms1.insert( 1 );
   hms1.insert( 2 );
   hms1.insert( 3 );

   hms1_Iter = hms1.begin( );
   cout << "The first element of hms1 is " << *hms1_Iter << endl;

   hms1_Iter = hms1.begin( );
   hms1.erase( hms1_Iter );

   // The following 2 lines would err because the iterator is const
   // hms1_cIter = hms1.begin( );
   // hms1.erase( hms1_cIter );

   hms1_cIter = hms1.begin( );
   cout << "The first element of hms1 is now " << *hms1_cIter << endl;
}
```

Output

```
The first element of hms1 is 1
The first element of hms1 is now 2
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

hash_multiset::clear

Erases all the elements of a hash_multiset.

```
void clear( );
```

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_clear.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;

   hms1.insert( 1 );
   hms1.insert( 2 );

   cout << "The size of the hash_multiset is initially " << hms1.size( )
        << "." << endl;

   hms1.clear( );
   cout << "The size of the hash_multiset after clearing is "
        << hms1.size( ) << "." << endl;
}
```

Output

```
The size of the hash_multiset is initially 2.
The size of the hash_multiset after clearing is 0.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::count

Returns the number of elements in a hash_multiset whose key matches a parameter-specified key.

```
size_type count(
   const Key& _Key
) const;
```

Parameters

_Key

The key of the elements to be matched from the hash_multiset.

Return Value

1 if the hash_multiset contains an element whose sort key matches the parameter key.

0 if the hash_multiset does not contain an element with a matching key.

Remarks

The member function returns the number of elements in the following range:

[**lower_bound** (*_Key*), **upper_bound** (*_Key*)).

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// hash_multiset_count.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset<int> hms1;
    hash_multiset<int>::size_type i;

    hms1.insert(1);
    hms1.insert(1);

    // Keys do not need to be unique in hash_multiset,
    // so duplicates may exist.
    i = hms1.count(1);
    cout << "The number of elements in hms1 with a sort key of 1 is: "
         << i << "." << endl;

    i = hms1.count(2);
    cout << "The number of elements in hms1 with a sort key of 2 is: "
         << i << "." << endl;
}
```

Output

```
The number of elements in hms1 with a sort key of 1 is: 2.
The number of elements in hms1 with a sort key of 2 is: 0.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::empty

Tests if a hash_multiset is empty.

```
bool empty( ) const;
```

Return Value

true if the hash_multiset is empty; **false** if the hash_multiset is nonempty.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_empty.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1, hms2;
   hms1.insert ( 1 );

   if ( hms1.empty( ) )
      cout << "The hash_multiset hms1 is empty." << endl;
   else
      cout << "The hash_multiset hms1 is not empty." << endl;

   if ( hms2.empty( ) )
      cout << "The hash_multiset hms2 is empty." << endl;
   else
      cout << "The hash_multiset hms2 is not empty." << endl;
}
```

Output

```
The hash_multiset hms1 is not empty.
The hash_multiset hms2 is empty.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::end

Returns an iterator that addresses the location succeeding the last element in a hash_multiset.

```
const_iterator end( ) const;
iterator end( );
```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a hash_multiset. If the hash_multiset is empty, then hash_multiset::end == hash_multiset::begin.

Remarks

end is used to test whether an iterator has reached the end of its hash_multiset. The value returned by **end** should not be dereferenced.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_end.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int> :: iterator hms1_Iter;
   hash_multiset <int> :: const_iterator hms1_cIter;

   hms1.insert( 1 );
   hms1.insert( 2 );
   hms1.insert( 3 );

   hms1_Iter = hms1.end( );
   hms1_Iter--;
   cout << "The last element of hms1 is " << *hms1_Iter << endl;

   hms1.erase( hms1_Iter );

   // The following 3 lines would err because the iterator is const
   // hms1_cIter = hms1.end( );
   // hms1_cIter--;
   // hms1.erase( hms1_cIter );

   hms1_cIter = hms1.end( );
   hms1_cIter--;
   cout << "The last element of hms1 is now " << *hms1_cIter << endl;
}
```

Output

```
The last element of hms1 is 3
The last element of hms1 is now 2
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::equal_range

Returns a pair of iterators respectively to the first element in a hash_multiset with a key that is greater than a specified key and to the first element in the hash_multiset with a key that is equal to or greater than the key.

```
pair <const_iterator, const_iterator> equal_range (
   const Key& _Key
) const;
pair <iterator, iterator> equal_range (
   const Key& _Key
);
```

Parameters

_Key

The argument key to be compared with the sort key of an element from the hash_multiset being searched.

Return Value

A pair of iterators where the first is the lower_bound of the key and the second is the upper_bound of the key.

To access the first iterator of a pair *pr* returned by the member function, use *pr*.**first** and to dereference the lower bound iterator, use *(*pr*.**first**). To access the second iterator of a pair *pr* returned by the member function, use *pr*.**second** and to dereference the upper bound iterator, use *(*pr*.**second**).

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_equal_range.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   typedef hash_multiset<int> IntHSet;
   IntHSet hms1;
   hash_multiset <int> :: const_iterator hms1_RcIter;

   hms1.insert( 10 );
   hms1.insert( 20 );
   hms1.insert( 30 );

   pair <IntHSet::const_iterator, IntHSet::const_iterator> p1, p2;
   p1 = hms1.equal_range( 20 );

   cout << "The upper bound of the element with "
        << "a key of 20\nin the hash_multiset hms1 is: "
        << *(p1.second) << "." << endl;

   cout << "The lower bound of the element with "
        << "a key of 20\nin the hash_multiset hms1 is: "
        << *(p1.first) << "." << endl;

   // Compare the upper_bound called directly
   hms1_RcIter = hms1.upper_bound( 20 );
   cout << "A direct call of upper_bound( 20 ) gives "
        << *hms1_RcIter << "," << endl
        << "matching the 2nd element of the pair"
        << " returned by equal_range( 20 )." << endl;
```

```
    p2 = hms1.equal_range( 40 );

    // If no match is found for the key,
    // both elements of the pair return end( )
    if ( ( p2.first == hms1.end( ) )
       && ( p2.second == hms1.end( ) ) )
       cout << "The hash_multiset hms1 doesn't have an element "
            << "with a key less than 40." << endl;
    else
       cout << "The element of hash_multiset hms1"
            << "with a key >= 40 is: "
            << *(p1.first) << "." << endl;
}
```

Output

```
The upper bound of the element with a key of 20
in the hash_multiset hms1 is: 30.
The lower bound of the element with a key of 20
in the hash_multiset hms1 is: 20.
A direct call of upper_bound( 20 ) gives 30,
matching the 2nd element of the pair returned by equal_range( 20 ).
The hash_multiset hms1 doesn't have an element with a key less than 40.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::erase

Removes an element or a range of elements in a hash_multiset from specified positions or removes elements that match a specified key.

```
iterator erase(
   iterator _Where
);
iterator erase(
   iterator _First,
   iterator _Last
);
size_type erase(
   const key_type& _Key
);
```

Parameters

_Where

Position of the element to be removed from the hash_multiset.

_First

Position of the first element removed from the hash_multiset.

_Last

Position just beyond the last element removed from the hash_multiset.

_Key

The key of the elements to be removed from the hash_multiset.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the hash_multiset if no such element exists. For the third member function, the number of elements that have been removed from the hash_multiset.

Remarks

The member functions never throw an exception.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// hash_multiset_erase.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_multiset<int> hms1, hms2, hms3;
    hash_multiset<int> :: iterator pIter, Iter1, Iter2;
    int i;
    hash_multiset<int>::size_type n;
```

```
    for (i = 1; i < 5; i++)
    {
        hms1.insert(i);
        hms2.insert(i * i);
        hms3.insert(i - 1);
    }

    // The 1st member function removes an element at a given position
    Iter1 = ++hms1.begin();
    hms1.erase(Iter1);

    cout << "After the 2nd element is deleted,\n "
         << "the hash_multiset hms1 is:" ;
    for (pIter = hms1.begin(); pIter != hms1.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;

    // The 2nd member function removes elements
    // in the range [_First, _Last)
    Iter1 = ++hms2.begin();
    Iter2 = --hms2.end();
    hms2.erase(Iter1, Iter2);

    cout << "After the middle two elements are deleted,\n "
         << "the hash_multiset hms2 is:" ;
    for (pIter = hms2.begin(); pIter != hms2.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;

    // The 3rd member function removes elements with a given _Key
    n = hms3.erase(2);

    cout << "After the element with a key of 2 is deleted,\n "
         << "the hash_multiset hms3 is:" ;
    for (pIter = hms3.begin(); pIter != hms3.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;

    // The 3rd member function returns the number of elements removed
    cout << "The number of elements removed from hms3 is: "
         << n << "." << endl;

    // The dereferenced iterator can also be used to specify a key
    Iter1 = ++hms3.begin();
    hms3.erase(Iter1);

    cout << "After another element with a key "
         << "equal to that of the 2nd element\n is deleted, "
         << "the hash_multiset hms3 is:" ;
    for (pIter = hms3.begin(); pIter != hms3.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;
}
```

Output

```
After the 2nd element is deleted,
 the hash_multiset hms1 is: 1 3 4.
After the middle two elements are deleted,
 the hash_multiset hms2 is: 1 16.
After the element with a key of 2 is deleted,
 the hash_multiset hms3 is: 0 1 3.
The number of elements removed from hms3 is: 1.
After another element with a key equal to that of the 2nd element
 is deleted, the hash_multiset hms3 is: 0 3.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::find

Returns an iterator addressing the location of an element in a hash_multiset that has a key equivalent to a specified key.

```
iterator find(
   const Key& _Key
);
const_iterator find(
   const Key& _Key
) const;
```

Parameters

_Key

The argument key to be matched by the sort key of an element from the hash_multiset being searched.

Return Value

An iterator or const_iterator that addresses the location of an element equivalent to a specified key or that addresses the location succeeding the last element in the hash_multiset if no match is found for the key.

Remarks

The member function returns an iterator that addresses an element in the hash_multiset whose sort key is **equivalent** to the argument key under a binary predicate that induces an ordering based on a less-than comparability relation.

If the return value of **find** is assigned to a **const_iterator**, the hash_multiset object cannot be modified. If the return value of **find** is assigned to an **iterator**, the hash_multiset object can be modified.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_find.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int> :: const_iterator hms1_AcIter, hms1_RcIter;

   hms1.insert( 10 );
   hms1.insert( 20 );
   hms1.insert( 30 );

   hms1_RcIter = hms1.find( 20 );
   cout << "The element of hash_multiset hms1 with a key of 20 is: "
        << *hms1_RcIter << "." << endl;

   hms1_RcIter = hms1.find( 40 );

   // If no match is found for the key, end( ) is returned
   if ( hms1_RcIter == hms1.end( ) )
      cout << "The hash_multiset hms1 doesn't have an element "
           << "with a key of 40." << endl;
   else
      cout << "The element of hash_multiset hms1 with a key of 40 is: "
           << *hms1_RcIter << "." << endl;

   // The element at a specific location in the hash_multiset can be found
```

```
    // by using a dereferenced iterator addressing the location
    hms1_AcIter = hms1.end( );
    hms1_AcIter--;
    hms1_RcIter = hms1.find( *hms1_AcIter );
    cout << "The element of hms1 with a key matching "
         << "that of the last element is: "
         << *hms1_RcIter << "." << endl;
}
```

Output

```
The element of hash_multiset hms1 with a key of 20 is: 20.
The hash_multiset hms1 doesn't have an element with a key of 40.
The element of hms1 with a key matching that of the last element is: 30.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::get_allocator

Returns a copy of the allocator object used to construct the hash_multiset.

```
Allocator get_allocator( ) const;
```

Return Value

The allocator used by the hash_multiset to manage memory, which is the class's template parameter **Allocator**.

For more information on **Allocator**, see the Remarks section of the hash_multiset Class topic.

Remarks

Allocators for the hash_multiset class specify how the class manages storage. The default allocators supplied with STL container classes is sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_get_allocator.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;

   // The following lines declare objects
   // that use the default allocator.
   hash_multiset <int, hash_compare <int, less<int> > > hms1;
   hash_multiset <int, hash_compare <int, greater<int> > > hms2;
   hash_multiset <double, hash_compare <double,
      less<double> >, allocator<double> > hms3;

   hash_multiset <int, hash_compare <int,
      greater<int> > >::allocator_type hms2_Alloc;
   hash_multiset <double>::allocator_type hms3_Alloc;
   hms2_Alloc = hms2.get_allocator( );

   cout << "The number of integers that can be allocated"
        << endl << "before free memory is exhausted: "
        << hms1.max_size( ) << "." << endl;

   cout << "The number of doubles that can be allocated"
        << endl << "before free memory is exhausted: "
        << hms3.max_size( ) <<  "." << endl;

   // The following lines create a hash_multiset hms4
   // with the allocator of hash_multiset hms1.
   hash_multiset <int>::allocator_type hms4_Alloc;
   hash_multiset <int> hms4;
   hms4_Alloc = hms2.get_allocator( );

   // Two allocators are interchangeable if
   // storage allocated from each can be
   // deallocated by the other
   if( hms2_Alloc == hms4_Alloc )
   {
```

```
        cout << "The allocators are interchangeable."
             << endl;
    }
    else
    {
        cout << "The allocators are not interchangeable."
             << endl;
    }
}
```

Sample Output

The following output is for x86.

```
The number of integers that can be allocated
before free memory is exhausted: 1073741823.
The number of doubles that can be allocated
before free memory is exhausted: 536870911.
The allocators are interchangeable.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::hash_multiset

Constructs a hash_multiset that is empty or that is a copy of all or part of some other hash_multiset.

```
hash_multiset( );
explicit hash_multiset(
   const Traits& _Comp
);
explicit hash_multiset(
   const Traits& _Comp,
   const Allocator& _Al
);
hash_multiset(
   const _hash_multiset& _Right
);
template<class InputIterator>
   hash_multiset(
      InputIterator _First,
      InputIterator _Last
   );
template<class InputIterator>
   hash_multiset(
      InputIterator _First,
      InputIterator _Last,
      const Traits& _Comp
   );
template<class InputIterator>
   hash_multiset(
      InputIterator _First,
      InputIterator _Last,
      const Traits& _Comp,
      const Allocator& _Al
   );
```

Parameters

_Al

The storage allocator class to be used for this hash_multiset object, which defaults to **Allocator**.

_Comp

The comparison function of type **const Traits** used to order the elements in the hash_multiset, which defaults to **hash_compare**.

_Right

The hash_multiset of which the constructed hash_multiset is to be a copy.

_First

The position of the first element in the range of elements to be copied.

_Last

The position of the first element beyond the range of elements to be copied.

Remarks

All constructors store a type of allocator object that manages memory storage for the hash_multiset and that can later be returned by calling get_allocator. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their hash_multisets.

All constructors store a function object of type **Traits** that is used to establish an order among the keys of the hash_multiset and that can later be returned by calling key_comp. For more information on **Traits** see the hash_multiset Class topic.

The first three constructors specify an empty initial hash_multiset, the second specifying the type of comparison function (*_Comp*) to be used in establishing the order of the elements and the third explicitly specifying the allocator type (*_Al*) to be used. The keyword **explicit** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the hash_multiset *_Right*.

The last three constructors copy the range [*_First*, *_Last*) of a hash_multiset with increasing explicitness in specifying the type of comparison function of class Compare and allocator.

The actual order of elements in a hashed set container depends on the hash function, the ordering function and the current size of the hash table and cannot, in general, be predicted as it could with the set container, where it was determined by the ordering function alone.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_hash_multiset.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int>::iterator hms1_Iter, hms3_Iter, hms4_Iter,
      hms5_Iter, hms6_Iter;
   hash_multiset <int, hash_compare <int, greater<int> > >::iterator
      hms2_Iter;

   // Create an empty hash_multiset hs0 of key type integer
   hash_multiset <int> hs0;

   // Create an empty hash_multiset hms1 with the key comparison
   // function of less than, then insert 4 elements
   hash_multiset <int, hash_compare <int, less<int> > > hms1;
   hms1.insert( 10 );
   hms1.insert( 20 );
   hms1.insert( 30 );
   hms1.insert( 40 );

   // Create an empty hash_multiset hms2 with the key comparison
   // function of geater than, then insert 2 elements
   hash_multiset <int, hash_compare <int, greater<int> > > hms2;
   hms2.insert( 10 );
   hms2.insert( 20 );

   // Create a hash_multiset hms3 with the
   // allocator of hash_multiset hms1
   hash_multiset <int>::allocator_type hms1_Alloc;
   hms1_Alloc = hms1.get_allocator( );
   hash_multiset <int> hms3( hash_compare <int, less<int> >( ),
      hms1_Alloc );
   hms3.insert( 30 );

   // Create a copy, hash_multiset hms4, of hash_multiset hms1
   hash_multiset <int> hms4( hms1 );

   // Create a hash_multiset hms5 by copying the range hms1[_First, _Last)
   hash_multiset <int>::const_iterator hms1_bcIter, hms1_ecIter;
   hms1_bcIter = hms1.begin( );
   hms1_ecIter = hms1.begin( );
   hms1_ecIter++;
   hms1_ecIter++;
   hash_multiset <int> hms5( hms1_bcIter, hms1_ecIter );
```

```
    // Create a hash_multiset hms6 by copying the range hms4[_First, _Last)
    // and with the allocator of hash_multiset hms2
    hash_multiset <int>::allocator_type hms2_Alloc;
    hms2_Alloc = hms2.get_allocator( );
    hash_multiset <int> hms6( hms4.begin( ), ++hms4.begin( ),
       less<int>( ), hms2_Alloc );

    cout << "hms1 = ";
    for ( hms1_Iter = hms1.begin( ); hms1_Iter != hms1.end( );
          hms1_Iter++ )
       cout << *hms1_Iter << " ";
    cout << endl;

    cout << "hms2 = " ;
    for ( hms2_Iter = hms2.begin( ); hms2_Iter != hms2.end( );
          hms2_Iter++ )
       cout << *hms2_Iter << " ";
    cout << endl;

    cout << "hms3 = ";
    for ( hms3_Iter = hms3.begin( ); hms3_Iter != hms3.end( );
          hms3_Iter++ )
       cout << *hms3_Iter << " ";
    cout << endl;

    cout << "hms4 = ";
    for ( hms4_Iter = hms4.begin( ); hms4_Iter != hms4.end( );
          hms4_Iter++ )
       cout << *hms4_Iter << " ";
    cout << endl;

    cout << "hms5 = ";
    for ( hms5_Iter = hms5.begin( ); hms5_Iter != hms5.end( );
          hms5_Iter++ )
       cout << *hms5_Iter << " ";
    cout << endl;

    cout << "hms6 = ";
    for ( hms6_Iter = hms6.begin( ); hms6_Iter != hms6.end( );
          hms6_Iter++ )
       cout << *hms6_Iter << " ";
    cout << endl;
  }
```

Output

```
hms1 = 10 20 30 40
hms2 = 20 10
hms3 = 30
hms4 = 10 20 30 40
hms5 = 10 20
hms6 = 10
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::insert

Inserts an element or a range of elements into a hash_multiset.

```
pair <iterator, bool> insert(
   const value_type& _Val
);
iterator insert(
   iterator _Where,
   const value_type& _Val
);
template<class InputIterator>
   void insert(
      InputIterator _First,
      InputIterator _Last
   );
```

Parameters

_Val

The value of an element to be inserted into the hash_multiset unless the hash_multiset already contains that element or, more generally, an element whose key is equivalently ordered.

_Where

The place to start searching for the correct point of insertion. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows *_Where*.)

_First

The position of the first element to be copied from a hash_multiset.

_Last

The position just beyond the last element to be copied from a hash_multiset.

Return Value

The first **insert** member function returns a pair whose bool component returns **true** if an insertion was make and **false** if the hash_multiset already contained an element whose key had an equivalent value in the ordering, and whose iterator component returns the address where a new element was inserted or where the element was already located.

To access the iterator component of a pair *pr* returned by this member function, use *pr*.**first** and to dereference it, use *(*pr*.**first**). To access the **bool** component of a pair *pr* returned by this member function, use *pr*.**second**, and to dereference it, use *(*pr*.**second**).

The second **insert** member function returns an iterator that points to the position where the new element was inserted into the hash_multiset.

Remarks

The third member function inserts the sequence of element values into a hash_multiset corresponding to each element addressed by an iterator of in the range [*_First*, *_Last*) of a specified hash_multiset.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_insert.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
```

```
    using namespace std;
    using namespace stdext;
    hash_multiset <int>::iterator hms1_pIter, hms2_pIter;

    hash_multiset <int, hash_compare <int, less<int> > > hms1, hms2;
    hms1.insert( 10 );
    hms1.insert( 20 );
    hms1.insert( 30 );
    hms1.insert( 40 );

    cout << "The original hms1 =";
    for ( hms1_pIter = hms1.begin( ); hms1_pIter != hms1.end( );
          hms1_pIter++ )
       cout << " " << *hms1_pIter;
    cout << "." << endl;


    hms1.insert( 20 );
    hms1.insert( --hms1.end( ), 50 );

    cout << "After the insertions, hms1 =";
    for ( hms1_pIter = hms1.begin( ); hms1_pIter != hms1.end( );
          hms1_pIter++ )
       cout << " " << *hms1_pIter;
    cout << "." << endl;

    hms2.insert( 100 );
    hms2.insert( ++hms1.begin( ), --hms1.end( ) );

    cout << "hms2 =";
    for ( hms2_pIter = hms2.begin( ); hms2_pIter != hms2.end( );
          hms2_pIter++ )
       cout << " " << *hms2_pIter;
    cout << "." << endl;
 }
```

Output

```
The original hms1 = 10 20 30 40.
After the insertions, hms1 = 10 20 20 30 40 50.
hms2 = 20 20 30 40 100.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::key_comp

Retrieves a copy of the comparison object used to order keys in a hash_multiset.

```
key_compare key_comp( ) const;
```

Return Value

Returns the hash_multiset template parameter **Traits**, which contains function objects that are used to hash and to order the elements of the container.

For more information on **Traits** see the hash_multiset Class topic.

Remarks

The stored object defines a member function:

bool operator(**const Key&** *_xVal*, **const Key&** *_yVal*);

which returns **true** if *_xVal* precedes and is not equal to *_yVal* in the sort order.

Note that both key_compare and value_compare are synonyms for the template parameter **Traits**. Both types are provided for the hash_multiset and hash_multiset classes, where they are identical, for compatibility with the hash_map and hash_multimap classes, where they are distinct.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_key_comp.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;

   hash_multiset <int, hash_compare < int, less<int> > >hms1;
   hash_multiset<int, hash_compare < int, less<int> > >::key_compare kc1
          = hms1.key_comp( ) ;
   bool result1 = kc1( 2, 3 ) ;
   if( result1 == true )
   {
      cout << "kc1( 2,3 ) returns value of true, "
           << "where kc1 is the function object of hms1."
           << endl;
   }
   else
   {
      cout << "kc1( 2,3 ) returns value of false "
           << "where kc1 is the function object of hms1."
        << endl;
   }

   hash_multiset <int, hash_compare < int, greater<int> > > hms2;
   hash_multiset<int, hash_compare < int, greater<int> > >::key_compare
         kc2 = hms2.key_comp( ) ;
   bool result2 = kc2( 2, 3 ) ;
   if( result2 == true )
   {
      cout << "kc2( 2,3 ) returns value of true, "
           << "where kc2 is the function object of hms2."
```

```
            << endl;
    }
    else
    {
       cout << "kc2( 2,3 ) returns value of false, "
            << "where kc2 is the function object of hms2."
            << endl;
    }
 }
```

Output

```
kc1( 2,3 ) returns value of true, where kc1 is the function object of hms1.
kc2( 2,3 ) returns value of false, where kc2 is the function object of hms2.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::lower_bound

Returns an iterator to the first element in a hash_multiset with a key that is equal to or greater than a specified key.

```
const_iterator lower_bound(
   const Key& _Key
) const;
iterator lower_bound(
   const Key& _Key
);
```

Parameters

_Key

The argument key to be compared with the sort key of an element from the hash_multiset being searched.

Return Value

An iterator or const_iterator that addresses the location of the first element in a hash_multiset with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the hash_multiset if no match is found for the key.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_lower_bound.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main() {
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int> :: const_iterator hms1_AcIter, hms1_RcIter;

   hms1.insert( 10 );
   hms1.insert( 20 );
   hms1.insert( 30 );

   hms1_RcIter = hms1.lower_bound( 20 );
   cout << "The element of hash_multiset hms1 with a key of 20 is: "
        << *hms1_RcIter << "." << endl;

   hms1_RcIter = hms1.lower_bound( 40 );

   // If no match is found for the key, end( ) is returned
   if ( hms1_RcIter == hms1.end( ) )
      cout << "The hash_multiset hms1 doesn't have an element "
           << "with a key of 40." << endl;
   else
      cout << "The element of hash_multiset hms1 with a key of 40 is: "
           << *hms1_RcIter << "." << endl;

   // An element at a specific location in the hash_multiset can be found
   // by using a dereferenced iterator that addresses the location
   hms1_AcIter = hms1.end( );
   hms1_AcIter--;
   hms1_RcIter = hms1.lower_bound( *hms1_AcIter );
   cout << "The element of hms1 with a key matching "
```

```
        << "that of the last element is: "
        << *hms1_RcIter << "." << endl;
}
```

Output

```
The element of hash_multiset hms1 with a key of 20 is: 20.
The hash_multiset hms1 doesn't have an element with a key of 40.
The element of hms1 with a key matching that of the last element is: 30.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::max_size

Returns the maximum length of the hash_multiset.

```
size_type max_size( ) const;
```

Return Value

The maximum possible length of the hash_multiset.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_max_size.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int>::size_type i;

   i = hms1.max_size( );
   cout << "The maximum possible length "
        << "of the hash_multiset is " << i << "." << endl;
}
```

Sample Output

The following output is for x86.

```
The maximum possible length of the hash_multiset is 1073741823.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::rbegin

Returns an iterator addressing the first element in a reversed hash_multiset.

```
const_reverse_iterator rbegin( ) const;
reverse_iterator rbegin( );
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed hash_multiset or addressing what had been the last element in the unreversed hash_multiset.

Remarks

rbegin is used with a reversed hash_multiset just as begin is used with a hash_multiset.

If the return value of **rbegin** is assigned to a **const_reverse_iterator**, then the hash_multiset object cannot be modified. If the return value of **rbegin** is assigned to a **reverse_iterator**, then the hash_multiset object can be modified.

rbegin can be used to iterate through a hash_multiset backwards.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_rbegin.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int>::iterator hms1_Iter;
   hash_multiset <int>::reverse_iterator hms1_rIter;

   hms1.insert( 10 );
   hms1.insert( 20 );
   hms1.insert( 30 );

   hms1_rIter = hms1.rbegin( );
   cout << "The first element in the reversed hash_multiset is "
        << *hms1_rIter << "." << endl;

   // begin can be used to start an iteration
   // throught a hash_multiset in a forward order
   cout << "The hash_multiset is: ";
   for ( hms1_Iter = hms1.begin( ) ; hms1_Iter != hms1.end( );
         hms1_Iter++ )
      cout << *hms1_Iter << " ";
   cout << endl;

   // rbegin can be used to start an iteration
   // throught a hash_multiset in a reverse order
   cout << "The reversed hash_multiset is: ";
   for ( hms1_rIter = hms1.rbegin( ) ; hms1_rIter != hms1.rend( );
         hms1_rIter++ )
      cout << *hms1_rIter << " ";
   cout << endl;

   // A hash_multiset element can be erased by dereferencing to its key
```

```
    hms1_rIter = hms1.rbegin( );
    hms1.erase ( *hms1_rIter );

    hms1_rIter = hms1.rbegin( );
    cout << "After the erasure, the first element "
         << "in the reversed hash_multiset is "<< *hms1_rIter << "."
         << endl;
}
```

Output

```
The first element in the reversed hash_multiset is 30.
The hash_multiset is: 10 20 30
The reversed hash_multiset is: 30 20 10
After the erasure, the first element in the reversed hash_multiset is 20.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::rend

Returns an iterator that addresses the location succeeding the last element in a reversed hash_multiset.

```
const_reverse_iterator rend( ) const;
reverse_iterator rend( );
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed hash_multiset (the location that had preceded the first element in the unreversed hash_multiset).

Remarks

rend is used with a reversed hash_multiset just as end is used with a hash_multiset.

If the return value of **rend** is assigned to a **const_reverse_iterator**, then the hash_multiset object cannot be modified. If the return value of **rend** is assigned to a **reverse_iterator**, then the hash_multiset object can be modified. The value returned by **rend** should not be dereferenced.

rend can be used to test to whether a reverse iterator has reached the end of its hash_multiset.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_rend.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int>::iterator hms1_Iter;
   hash_multiset <int>::reverse_iterator hms1_rIter;
   hash_multiset <int>::const_reverse_iterator hms1_crIter;

   hms1.insert( 10 );
   hms1.insert( 20 );
   hms1.insert( 30 );

   hms1_rIter = hms1.rend( );
   hms1_rIter--;
   cout << "The last element in the reversed hash_multiset is "
        << *hms1_rIter << "." << endl;

   // end can be used to terminate an iteration
   // through a hash_multiset in a forward order
   cout << "The hash_multiset is: ";
   for ( hms1_Iter = hms1.begin( ) ; hms1_Iter != hms1.end( );
         hms1_Iter++ )
      cout << *hms1_Iter << " ";
   cout << "." << endl;

   // rend can be used to terminate an iteration
   // throught a hash_multiset in a reverse order
   cout << "The reversed hash_multiset is: ";
   for ( hms1_rIter = hms1.rbegin( ) ; hms1_rIter != hms1.rend( );
         hms1_rIter++ )
      cout << *hms1_rIter << " ";
```

```
    cout << "." << endl;

    hms1_rIter = hms1.rend( );
    hms1_rIter--;
    hms1.erase ( *hms1_rIter );

    hms1_rIter = hms1.rend( );
    hms1_rIter--;
    cout << "After the erasure, the last element in the "
         << "reversed hash_multiset is " << *hms1_rIter << "."
         << endl;
}
```

Output

```
The last element in the reversed hash_multiset is 10.
The hash_multiset is: 10 20 30 .
The reversed hash_multiset is: 30 20 10 .
After the erasure, the last element in the reversed hash_multiset is 20.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::size

Returns the number of elements in the hash_multiset.

```
size_type size( ) const;
```

Return Value

The current length of the hash_multiset.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_size.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int> :: size_type i;

   hms1.insert( 1 );
   i = hms1.size( );
   cout << "The hash_multiset length is " << i << "." << endl;

   hms1.insert( 2 );
   i = hms1.size( );
   cout << "The hash_multiset length is now " << i << "." << endl;
}
```

Output

```
The hash_multiset length is 1.
The hash_multiset length is now 2.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::swap

Exchanges the elements of two hash_multisets.

```
void swap(
   hash_multiset& _Right
);
```

Parameters

_Right

The argument hash_multiset providing the elements to be swapped with the target hash_multiset.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two hash_multisets whose elements are being exchanged.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_swap.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1, hms2, hms3;
   hash_multiset <int>::iterator hms1_Iter;

   hms1.insert( 10 );
   hms1.insert( 20 );
   hms1.insert( 30 );
   hms2.insert( 100 );
   hms2.insert( 200 );
   hms3.insert( 300 );

   cout << "The original hash_multiset hms1 is:";
   for ( hms1_Iter = hms1.begin( ); hms1_Iter != hms1.end( );
         hms1_Iter++ )
         cout << " " << *hms1_Iter;
   cout   << "." << endl;

   // This is the member function version of swap
   hms1.swap( hms2 );

   cout << "After swapping with hms2, list hms1 is:";
   for ( hms1_Iter = hms1.begin( ); hms1_Iter != hms1.end( );
         hms1_Iter++ )
         cout << " " << *hms1_Iter;
   cout  << "." << endl;

   // This is the specialized template version of swap
   swap( hms1, hms3 );

   cout << "After swapping with hms3, list hms1 is:";
   for ( hms1_Iter = hms1.begin( ); hms1_Iter != hms1.end( );
         hms1_Iter++ )
         cout << " " << *hms1_Iter;
```

```
    cout   << "." << endl;
}
```

Output

```
The original hash_multiset hms1 is: 10 20 30.
After swapping with hms2, list hms1 is: 100 200.
After swapping with hms3, list hms1 is: 300.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::upper_bound

Returns an iterator to the first element in a hash_multiset with a key that is greater than a specified key.

```
const_iterator upper_bound(
   const Key& _Key
) const;
iterator upper_bound(
   const Key& _Key
);
```

Parameters

_Key

The argument key to be compared with the sort key of an element from the hash_multiset being searched.

Return Value

An iterator or const_iterator that addresses the location of the first element in a hash_multiset with a key greater than the argument key, or that addresses the location succeeding the last element in the hash_multiset if no match is found for the key.

Remarks

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_upper_bound.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int> :: const_iterator hms1_AcIter, hms1_RcIter;

   hms1.insert( 10 );
   hms1.insert( 20 );
   hms1.insert( 30 );

   hms1_RcIter = hms1.upper_bound( 20 );
   cout << "The first element of hash_multiset hms1" << endl
        << " with a key greater than 20 is: "
        << *hms1_RcIter << "." << endl;

   hms1_RcIter = hms1.upper_bound( 30 );

   // If no match is found for the key, end( ) is returned
   if ( hms1_RcIter == hms1.end( ) )
      cout << "The hash_multiset hms1 doesn't have an element "
           << "\n with a key greater than 30." << endl;
   else
      cout << "The element of hash_multiset hms1"
           << "with a key > 40 is: "
           << *hms1_RcIter << "." << endl;

   // An element at a specific location in the hash_multiset can be
   // found by using a dereferenced iterator addressing the location
   hms1_AcIter = hms1.begin( );
   hms1_RcIter = hms1.upper_bound( *hms1_AcIter );
```

```
    cout << "The first element of hms1\n with a key greater than "
         << endl << "that of the initial element of hms1 is: "
         << *hms1_RcIter << "." << endl;
  }
```

Output

```
The first element of hash_multiset hms1
 with a key greater than 20 is: 30.
The hash_multiset hms1 doesn't have an element
 with a key greater than 30.
The first element of hms1
 with a key greater than
that of the initial element of hms1 is: 20.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

hash_multiset::value_comp

Retrieves a copy of the comparison object used to order element values in a hash_multiset.

```
value_compare value_comp( ) const;
```

Return Value

Returns the hash_multiset template parameter **Traits**, which contains function objects that are used to hash and to order elements of the container.

For more information on **Traits** see the hash_multiset Class topic.

Remarks

The stored object defines a member function:

bool operator(**const Key&** *_xVal*, **const Key&** *_yVal*);

which returns **true** if *_xVal* precedes and is not equal to *_yVal* in the sort order.

Note that both key_compare and value_compare are synonyms for the template parameter **Traits**. Both types are provided for the hash_multiset and hash_multiset classes, where they are identical, for compatibility with the hash_map and hash_multimap classes, where they are distinct.

In Visual C++ .NET 2003, members of the <hash_map> and <hash_set> header files are no longer in the std namespace, but rather have been moved into the stdext namespace. See The stdext Namespace for more information.

Example

```
// hash_multiset_value_comp.cpp
// compile with: /EHsc
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;

   hash_multiset <int, hash_compare < int, less<int> > > hms1;
   hash_multiset <int, hash_compare < int, less<int> > >::value_compare
      vc1 = hms1.value_comp( );
   bool result1 = vc1( 2, 3 );
   if( result1 == true )
   {
      cout << "vc1( 2,3 ) returns value of true, "
           << "where vc1 is the function object of hms1."
           << endl;
   }
   else
   {
      cout << "vc1( 2,3 ) returns value of false, "
           << "where vc1 is the function object of hms1."
           << endl;
   }

   hash_multiset <int, hash_compare < int, greater<int> > > hms2;
   hash_multiset<int, hash_compare < int, greater<int> > >::
           value_compare vc2 = hms2.value_comp( );
   bool result2 = vc2( 2, 3 );
   if( result2 == true )
   {
      cout << "vc2( 2,3 ) returns value of true, "
           << "where vc2 is the function object of hms2."
```

```
            << endl;
    }
    else
    {
       cout << "vc2( 2,3 ) returns value of false, "
            << "where vc2 is the function object of hms2."
            << endl;
    }
  }
```

Output

```
vc1( 2,3 ) returns value of true, where vc1 is the function object of hms1.
vc2( 2,3 ) returns value of false, where vc2 is the function object of hms2.
```

Requirements

Header: <hash_set>

See Also

Reference

hash_multiset Class

Standard Template Library

Other Resources

hash_multiset Members

<iomanip>

Define several manipulators that each take a single argument.

For a list of all members of this header, see <iomanip> Members.

```
#include <iomanip>
```

Remarks

Each of these manipulators returns an unspecified type, called **T1** through **T6**, that overloads both **basic_istream**<**Elem**, **Tr**>**::**operator>> and **basic_ostream**<**Elem**, **Tr**>**::**operator<<.

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

<iomanip> Members
Standard C++ Library Header Files

<iomanip> Members

Reference

Manipulators

resetiosflags	Clears the specified flags.
setbase	Set base for integers.
setfill	Sets the character that will be used to fill spaces in a right-justified display.
setiosflags	Sets the specified flags.
setprecision	Sets the precision for floating-point values.
setw	Specifies the width of the display field.

See Also

Reference

<iomanip>
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

<iomanip> Manipulators

For information about the manipulators in <iomanip>, see <iomanip> Members.

See Also

Reference

iostream Programming

iostreams Conventions

resetiosflags

Clears the specified flags.

```
T1 resetiosflags(
   ios_base::fmtflags _Mask
);
```

Parameters

_Mask

The flags to clear.

Return Value

The manipulator returns an object that, when extracted from or inserted into the stream **str**, calls **str**.setf(**ios_base::**fmtflags, *_Mask*), and then returns **str**.

Example

See setw for an example of using **resetiosflags**.

Requirements

Header: <iomanip>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<iomanip> Members

setbase

Set base for integers.

```
T3 setbase(
   int _Base
);
```

Parameters

_Base

The number base.

Return Value

The manipulator returns an object that, when extracted from or inserted into the stream **str**, calls **str**.**setf**(**mask**, ios_base::basefield), and then returns **str**. Here, **mask** is determined as follows:

- If *_Base* is 8, then **mask** is **ios_base::**oct.
- If *_Base* is 10, then mask is **ios_base::**dec.
- If *_Base* is 16, then **mask** is **ios_base::**hex.
- If *_Base* is any other value, then mask is **ios_base::**fmtflags(0).

Example

See setw for an example of using **setbase**.

Requirements

Header: <iomanip>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<iomanip> Members

setfill

Sets the character that will be used to fill spaces in a right-justified display.

```
template<class Elem>
   T4 setfill(
      Elem _Ch
      );
```

Parameters

_Ch

The character that will be used to fill spaces in a right-justified display.

Return Value

The template manipulator returns an object that, when extracted from or inserted into the stream **str**, calls **str**.fill(*_Ch*), and then returns **str**. The type **Elem** must be the same as the element type for the stream **str**.

Example

See setw for an example of using **setfill**.

Requirements

Header: <iomanip>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<iomanip> Members

setiosflags

Sets the specified flags.

```
T2 setiosflags(
   ios_base::fmtflags _Mask
);
```

Parameters

_Mask

The flags to set.

Return Value

The manipulator returns an object that, when extracted from or inserted into the stream **str**, calls **str**.setf(*_Mask*), and then returns **str**.

Example

See setw for an example of using **setiosflags**.

Requirements

Header: <iomanip>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<iomanip> Members

setprecision

Sets the precision for floating-point values.

```
T5 setprecision(
   streamsize _Prec
);
```

Parameters

_Prec

The precision for floating-point values.

Return Value

The manipulator returns an object that, when extracted from or inserted into the stream **str**, calls **str**.precision(*_Prec*), and then returns **str**.

Example

See setw for an example of using **setprecision**.

Requirements

Header: <iomanip>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<iomanip> Members

setw

Specifies the width of the display field.

```
T6 setw(
   streamsize _Wide
);
```

Parameters

_Wide

The width of the display field.

Return Value

The manipulator returns an object that, when extracted from or inserted into the stream **str**, calls **str**.width(*_Wide*), then returns **str**.

Example

```
// iomanip_setw.cpp
// compile with: /EHsc
// Defines the entry point for the console application.
//
// Sample use of the following manipulators:
//   resetiosflags
//   setiosflags
//   setbase
//   setfill
//   setprecision
//   setw

#include <iostream>
#include <iomanip>

using namespace std;

const double   d1 = 1.23456789;
const double   d2 = 12.3456789;
const double   d3 = 123.456789;
const double   d4 = 1234.56789;
const double   d5 = 12345.6789;
const long      l1 = 16;
const long      l2 = 256;
const long      l3 = 1024;
const long      l4 = 4096;
const long      l5 = 65536;
int         base = 10;

void DisplayDefault( )
{
   cout << endl << "default display" << endl;
   cout << "d1 = " << d1 << endl;
   cout << "d2 = " << d2 << endl;
   cout << "d3 = " << d3 << endl;
   cout << "d4 = " << d4 << endl;
   cout << "d5 = " << d5 << endl;
}

void DisplayWidth( int n )
{
   cout << endl << "fixed width display set to " << n << ".\n";
   cout << "d1 = " << setw(n) << d1 << endl;
   cout << "d2 = " << setw(n) << d2 << endl;
   cout << "d3 = " << setw(n) << d3 << endl;
```

```
   cout << "d4 = " << setw(n) << d4 << endl;
   cout << "d5 = " << setw(n) << d5 << endl;
}

void DisplayLongs( )
{
   cout << setbase(10);
   cout << endl << "setbase(" << base << ")" << endl;
   cout << setbase(base);
   cout << "l1 = " << l1 << endl;
   cout << "l2 = " << l2 << endl;
   cout << "l3 = " << l3 << endl;
   cout << "l4 = " << l4 << endl;
   cout << "l5 = " << l5 << endl;
}

int main( int argc, char* argv[] )
{
   DisplayDefault( );

   cout << endl << "setprecision(" << 3 << ")" << setprecision(3);
   DisplayDefault( );

   cout << endl << "setprecision(" << 12 << ")" << setprecision(12);
   DisplayDefault( );

   cout << setiosflags(ios_base::scientific);
   cout << endl << "setiosflags(" << ios_base::scientific << ")";
   DisplayDefault( );

   cout << resetiosflags(ios_base::scientific);
   cout << endl << "resetiosflags(" << ios_base::scientific << ")";
   DisplayDefault( );

   cout << endl << "setfill('" << 'S' << "')" << setfill('S');
   DisplayWidth(15);
   DisplayDefault( );

   cout << endl << "setfill('" << ' ' << "')" << setfill(' ');
   DisplayWidth(15);
   DisplayDefault( );

   cout << endl << "setprecision(" << 8 << ")" << setprecision(8);
   DisplayWidth(10);
   DisplayDefault( );

   base = 16;
   DisplayLongs( );

   base = 8;
   DisplayLongs( );

   base = 10;
   DisplayLongs( );

   return   0;
}
```

Output

```
default display
d1 = 1.23457
d2 = 12.3457
d3 = 123.457
d4 = 1234.57
d5 = 12345.7

setprecision(3)
```

```
default display
d1 = 1.23
d2 = 12.3
d3 = 123
d4 = 1.23e+003
d5 = 1.23e+004

setprecision(12)
default display
d1 = 1.23456789
d2 = 12.3456789
d3 = 123.456789
d4 = 1234.56789
d5 = 12345.6789

setiosflags(4096)
default display
d1 = 1.234567890000e+000
d2 = 1.234567890000e+001
d3 = 1.234567890000e+002
d4 = 1.234567890000e+003
d5 = 1.234567890000e+004

resetiosflags(4096)
default display
d1 = 1.23456789
d2 = 12.3456789
d3 = 123.456789
d4 = 1234.56789
d5 = 12345.6789

setfill('S')
fixed width display set to 15.
d1 = SSSSS1.23456789
d2 = SSSSS12.3456789
d3 = SSSSS123.456789
d4 = SSSSS1234.56789
d5 = SSSSS12345.6789

default display
d1 = 1.23456789
d2 = 12.3456789
d3 = 123.456789
d4 = 1234.56789
d5 = 12345.6789

setfill(' ')
fixed width display set to 15.
d1 =      1.23456789
d2 =      12.3456789
d3 =      123.456789
d4 =      1234.56789
d5 =      12345.6789

default display
d1 = 1.23456789
d2 = 12.3456789
d3 = 123.456789
d4 = 1234.56789
d5 = 12345.6789

setprecision(8)
fixed width display set to 10.
d1 =  1.2345679
d2 =  12.345679
d3 =  123.45679
d4 =  1234.5679
d5 =  12345.679

default display
d1 = 1.2345679
d2 = 12.345679
```

```
d3 = 123.45679
d4 = 1234.5679
d5 = 12345.679

setbase(16)
l1 = 10
l2 = 100
l3 = 400
l4 = 1000
l5 = 10000

setbase(8)
l1 = 20
l2 = 400
l3 = 2000
l4 = 10000
l5 = 200000

setbase(10)
l1 = 16
l2 = 256
l3 = 1024
l4 = 4096
l5 = 65536
```

Requirements

Header: <iomanip>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<iomanip> Members

<ios>

Defines several types and functions basic to the operation of iostreams. This header is typically included for you by another iostream headers; you rarely include it directly.

For a list of all members of this header, see <ios> Members.

```
#include <ios>
```

Remarks

A large group of functions are manipulators. A manipulator declared in <ios> alters the values stored in its argument object of class ios_base. Other manipulators perform actions on streams controlled by objects of a type derived from this class, such as a specialization of one of the template classes basic_istream or basic_ostream. For example, noskipws(**str**) clears the format flag **ios_base::skipws** in the object **str**, which can be of one of these types.

You can also call a manipulator by inserting it into an output stream or extracting it from an input stream, because of special insertion and extraction operations supplied for the classes derived from **ios_base**. For example:

```
istr >> noskipws;
```

calls noskipws(**istr**).

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

<ios> Members
Standard C++ Library Header Files

<ios> Members

Reference

Typedefs

ios	Supports the ios class from the old iostream library.
streamoff	Supports internal operations.
streampos	Holds the current position of the buffer pointer or file pointer.
streamsize	Specifies the size of the stream.
wios	Supports the wios class from the old iostream library.
wstreampos	Holds the current position of the buffer pointer or file pointer.

Manipulators

boolalpha	Specifies that variables of type bool appear as **true** or **false** in the stream.
dec	Specifies that integer variables appear in base 10 notation.
fixed	Specifies that a floating-point number is displayed in fixed-decimal notation.
hex	Specifies that integer variables appear in base 16 notation.
internal	Causes a number's sign to be left justified and the number to be right justified.
left	Causes text that is not as wide as the output width to appear in the stream flush with the left margin.
noboolalpha	Specifies that variables of type bool appear as 1 or 0 in the stream.
noshowbase	Turns off indicating the notational base in which a number is displayed.
noshowpoint	Displays only the whole-number part of floating-point numbers whose fractional part is zero.
noshowpos	Causes positive numbers to not be explicitly signed.
noskipws	Cause spaces to be read by the input stream.
nounitbuf	Causes output to be buffered and processed when the buffer is full.
nouppercase	Specifies that hexadecimal digits and the exponent in scientific notation appear in lowercase.
oct	Specifies that integer variables appear in base 8 notation.
right	Causes text that is not as wide as the output width to appear in the stream flush with the right margin.
scientific	Causes floating point numbers to be displayed using scientific notation.
showbase	Indicates the notational base in which a number is displayed.

showpoint	Displays the whole-number part of a floating-point number and digits to the right of the decimal point even when the fractional part is zero.
showpos	Causes positive numbers to be explicitly signed.
skipws	Cause spaces to not be read by the input stream.
unitbuf	Causes output to be processed when the buffer is not empty.
uppercase	Specifies that hexadecimal digits and the exponent in scientific notation appear in uppercase.

Classes

basic_ios	The template class describes the storage and member functions common to both input streams (of template class basic_istream) and output streams (of template class basic_ostream) that depend on the template parameters.
fpos	The template class describes an object that can store all the information needed to restore an arbitrary file-position indicator within any stream.
ios_base	The class describes the storage and member functions common to both input and output streams that do not depend on the template parameters.

See Also

Reference

<ios>
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

<ios> Typedefs

For information about the typedefs in <ios>, see <ios> Members.

See Also

Reference

iostream Programming

iostreams Conventions

ios

Supports the ios class from the old iostream library.

```
typedef basic_ios<char, char_traits<char> > ios;
```

Remarks

The type is a synonym for template class basic_ios, specialized for elements of type **char** with default character traits.

Requirements

Header: <ios>

See Also

Reference

basic_ios Class
iostream Programming
iostreams Conventions

Other Resources

basic_ios Members
<ios> Members

streamoff

Supports internal operations.

```
typedef T1 streamoff;
```

Remarks

The type is a signed integer type **T1** that describes an object that can store a byte offset involved in various stream positioning operations. Its representation has at least 32 value bits. It is not necessarily large enough to represent an arbitrary byte position within a stream. The value **streamoff(-1)** generally indicates an erroneous offset.

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

streampos

Holds the current position of the buffer pointer or file pointer.

```
typedef fpos<mbstate_t> streampos;
```

Remarks

The type is a synonym for fpos<**mbstate_t**>.

Example

```
// ios_streampos.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
   using namespace std;

   ofstream x( "iostream.txt" );
   x << "testing";
   streampos y = x.tellp( );
   cout << y << endl;
}
```

Output

```
7
```

Requirements

Header: <ios>

See Also

Reference

fpos Class

iostream Programming

iostreams Conventions

Other Resources

fpos Members

<ios> Members

streamsize

Specifies the size of the stream

```
typedef T2 streamsize;
```

Remarks

The type is a signed integer type **T2** that describes an object that can store a count of the number of elements involved in various stream operations. Its representation has at least 16 bits. It is not necessarily large enough to represent an arbitrary byte position within a stream.

Example

After compiling and running the following program, look at the file test.txt to see the effect of setting **streamsize**.

```
// ios_streamsize.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
   using namespace std;
   char a[16] = "any such text";
   ofstream x( "test.txt" );
   streamsize y = 6;
   x.write( a, y );
}
```

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

wios

Supports the wios class from the old iostream library.

```
typedef basic_ios<wchar_t, char_traits<wchar_t> > wios;
```

Remarks

The type is a synonym for template class basic_ios, specialized for elements of type **wchar_t** with default character traits.

Requirements

Header: <ios>

See Also

Reference

basic_ios Class
iostream Programming
iostreams Conventions

Other Resources

basic_ios Members
<ios> Members

wstreampos

Holds the current position of the buffer pointer or file pointer.

```
typedef fpos<mbstate_t> wstreampos;
```

Remarks

The type is a synonym for fpos<**mbstate_t**>.

Example

```
// ios_wstreampos.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
   using namespace std;
   wofstream xw( "wiostream.txt" );
   xw << L"testing";
   wstreampos y = xw.tellp( );
   cout << y << endl;
}
```

Output

```
7
```

Requirements

Header: <ios>

See Also

Reference

fpos Class

iostream Programming

iostreams Conventions

Other Resources

fpos Members

<ios> Members

<ios> Manipulators

For information about the manipulators in <ios>, see <ios> Members.

See Also

Reference

iostream Programming

iostreams Conventions

boolalpha

Specifies that variables of type bool appear as **true** or **false** in the stream.

```
ios_base& boolalpha(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

By default, variables of type **bool** are displayed as 1 or 0.

boolalpha effectively calls *_Str*.setf(**ios_base::boolalpha**), and then returns *_Str*.

noboolalpha reverses the effect of **boolalpha**.

Example

```
// ios_boolalpha.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   bool b = true;
   cout << b << endl;
   boolalpha( cout );
   cout << b << endl;
   noboolalpha( cout );
   cout << b << endl;
   cout << boolalpha << b << endl;
}
```

Output

```
1
true
1
true
```

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

dec

Specifies that integer variables appear in base 10 notation.

```
ios_base& dec(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

By default, integer variables are displayed in base 10.

dec effectively calls *_Str*.setf(**ios_base::dec, ios_base::basefield**), and then returns *_Str*.

Example

```
// ios_dec.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   int i = 100;

   cout << i << endl;   // Default is base 10
   cout << hex << i << endl;
   dec( cout );
   cout << i << endl;
   oct( cout );
   cout << i << endl;
   cout << dec << i << endl;
}
```

Output

```
100
64
100
144
100
```

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

fixed

Specifies that a floating-point number is displayed in fixed-decimal notation.

```
ios_base& fixed(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

fixed is the default display notation for floating-point numbers. scientific causes floating-point numbers to be displayed using scientific notation.

The manipulator effectively calls *_Str*.setf(**ios_base::fixed**, **ios_base::floatfield**), and then returns *_Str*.

Example

```
// ios_fixed.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   float i = 1.1F;

   cout << i << endl;   // fixed is the default
   cout << scientific << i << endl;
   cout.precision( 1 );
   cout << fixed << i << endl;
}
```

Output

```
1.1
1.100000e+000
1.1
```

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

hex

Specifies that integer variables shall appear in base 16 notation.

```
ios_base& hex(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

By default, integer variables are displayed in base 10 notation. dec and oct also change the way integer variables appear.

The manipulator effectively calls *_Str*.setf(**ios_base::hex**, **ios_base::basefield**), and then returns *_Str*.

Example

See dec for an example of how to use **hex**.

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

internal (Standard C++ Library)

Causes a number's sign to be left justified and the number to be right justified.

```
ios_base& internal(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

showpos causes the sign to display for positive numbers.

The manipulator effectively calls *_Str*.setf(ios_base::internal, ios_base::adjustfield), and then returns *_Str*.

Example

```
// ios_internal.cpp
// compile with: /EHsc
#include <iostream>
#include <iomanip>

int main( void )
{
   using namespace std;
   float i = -123.456F;
   cout.fill( '.' );
   cout << setw( 10 ) << i << endl;
   cout << setw( 10 ) << internal << i << endl;
}
```

Output

```
..-123.456
-..123.456
```

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

left

Causes text that is not as wide as the output width to appear in the stream flush with the left margin.

```
ios_base& left(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

The manipulator effectively calls *_Str*.setf(**ios_base::left**, **ios_base::adjustfield**), and then returns *_Str*.

Example

```
// ios_left.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   double f1= 5.00;
   cout.width( 20 );
   cout << f1 << endl;
   cout << left << f1 << endl;
}
```

Output

```
                    5
5
```

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

noboolalpha

Specifies that variables of type bool appear as 1 or 0 in the stream.

```
ios_base& noboolalpha(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

By default, **noboolalpha** is in effect.

noboolalpha effectively calls *_Str*.unsetf(**ios_base::boolalpha**), and then returns *_Str*.

boolalpha reverses the effect of **noboolalpha**.

Example

See boolalpha for an example of using **noboolalpha**.

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

noshowbase

Turns off indicating the notational base in which a number is displayed.

```
ios_base& noshowbase(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

noshowbase is on by default. Use showbase to indicate the notational base of numbers.

The manipulator effectively calls *_Str*.unsetf(**ios_base::showbase**), and then returns *_Str*.

Example

See showbase for an example of how to use **noshowbase**.

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

noshowpoint

Displays only the whole-number part of floating-point numbers whose fractional part is zero.

```
ios_base& noshowpoint(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

noshowpoint is on by default; use showpoint and precision to display zeros after the decimal point.

The manipulator effectively calls *_Str*.unsetf(**ios_base::showpoint**), and then returns *_Str*.

Example

```
// ios_noshowpoint.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   double f1= 5.000;
   cout << f1 << endl;   // noshowpoint is default
   cout.precision( 4 );
   cout << showpoint << f1 << endl;
   cout << noshowpoint << f1 << endl;
}
```

Output

```
5
5.000
5
```

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

noshowpos

Causes positive numbers to not be explicitly signed.

```
ios_base& noshowpos(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

noshowpos is on by default.

The manipulator effectively calls *_Str*.unsetf(**ios_base::showps**), then returns *_Str*.

Example

See showpos for an example of using **noshowpos**.

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

noskipws

Cause spaces to be read by the input stream.

```
ios_base& noskipws(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

By default, skipws is in effect. When a space is read in the input stream, it signals the end of the buffer.

The manipulator effectively calls *_Str*.unsetf(**ios_base::skipws**), and then returns *_Str*.

Example

```
// ios_noskipws.cpp
// compile with: /EHsc
#include <iostream>
#include <string>

int main() {
   using namespace std;
   string s1, s2, s3;
   cout << "Enter three strings: ";
   cin >> noskipws >> s1 >> s2 >> s3;
   cout << "." << s1  << "." << endl;
   cout << "." << s2 << "." << endl;
   cout << "." << s3 << "." << endl;
}
```

Input

```
1 2 3
```

Sample Output

```
.1.
..
..
```

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

nounitbuf

Causes output to be buffered and processed on when the buffer is full.

```
ios_base& nounitbuf(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

unitbuf causes the buffer to be processed when it is not empty.

The manipulator effectively calls *_Str*.unsetf(**ios_base::unitbuf**), and then returns *_Str*.

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

nouppercase

Specifies that hexadecimal digits and the exponent in scientific notation appear in lowercase.

```
ios_base& nouppercase(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

The manipulator effectively calls *_Str*.unsetf(**ios_base::uppercase**), and then returns *_Str*.

Example

See uppercase for an example of using **nouppercase**.

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

oct (<ios>)

Specifies that integer variables appear in base 8 notation.

```
ios_base& oct(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

By default, integer variables are displayed in base 10 notation. dec and hex also change the way integer variables appear.

The manipulator effectively calls *_Str*.setf(**ios_base::oct**, **ios_base::basefield**), and then returns *_Str*.

Example

See dec for an example of how to use **oct**.

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

right

Causes text that is not as wide as the output width to appear in the stream flush with the right margin.

```
ios_base& right(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

left also modifies the justification of text.

The manipulator effectively calls *_Str*.setf(**ios_base::right**, **ios_base::adjustfield**), and then returns *_Str*.

Example

```
// ios_right.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   double f1= 5.00;
   cout << f1 << endl;
   cout.width( 20 );
   cout << f1 << endl;
   cout.width( 20 );
   cout << left << f1 << endl;
   cout.width( 20 );
   cout << f1 << endl;
   cout.width( 20 );
   cout << right << f1 << endl;
   cout.width( 20 );
   cout << f1 << endl;
}
```

Output

```
5
                    5
5
5
                    5
                    5
```

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

scientific

Causes floating-point numbers to be displayed using scientific notation.

```
ios_base& scientific(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

By default, fixed notation is in effect for floating-point numbers.

The manipulator effectively calls *_Str*.setf(**ios_base::scientific**, **ios_base::floatfield**), and then returns *_Str*.

Example

```
// ios_scientific.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   float i = 100.23F;

   cout << i << endl;
   cout << scientific << i << endl;
}
```

Output

```
100.23
1.002300e+002
```

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

showbase

Indicates the notational base in which a number is displayed.

```
ios_base& showbase(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

The notational base of a number can be changed with dec, oct, or hex.

The manipulator effectively calls *_Str*.setf(**ios_base::showbase**), and then returns *_Str*.

Example

```
// ios_showbase.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   int j = 100;

   cout << showbase << j << endl;   // dec is default
   cout << hex << j << showbase << endl;
   cout << oct << j << showbase << endl;

   cout << dec << j << noshowbase << endl;
   cout << hex << j << noshowbase << endl;
   cout << oct << j << noshowbase << endl;
}
```

Output

```
100
0x64
0144
100
64
144
```

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

showpoint

Displays the whole-number part of a floating-point number and digits to the right of the decimal point even when the fractional part is zero.

```
ios_base& showpoint(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

By default, noshowpoint is in effect.

The manipulator effectively calls *_Str*.setf(**ios_base::showpoint**), and then returns *_Str*.

Example

See noshowpoint for an example of using **showpoint**.

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

showpos

Causes positive numbers to be explicitly signed.

```
ios_base& showpos(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

noshowpos is the default.

The manipulator effectively calls *_Str*.setf(**ios_base::showpos**), and then returns *_Str*.

Example

```
// ios_showpos.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   int i = 1;

   cout << noshowpos << i << endl;   // noshowpos is default
   cout << showpos << i << endl;
}
```

Output

```
1
+1
```

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

skipws

Cause spaces to not be read by the input stream.

```
ios_base& skipws(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

By default, **skipws** is in effect. noskipws will cause spaces to be read from the input stream.

The manipulator effectively calls *_Str*.setf(**ios_base::skipws**), and then returns *_Str*.

Example

```
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   char s1, s2, s3;
   cout << "Enter three characters: ";
   cin >> skipws >> s1 >> s2 >> s3;
   cout << "." << s1  << "." << endl;
   cout << "." << s2 << "." << endl;
   cout << "." << s3 << "." << endl;
}
```

Input

```
1 2 3
```

Sample Output

```
.1.
.2.
.3.
```

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

unitbuf

Causes output to be processed when the buffer is not empty.

```
ios_base& unitbuf(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

Note that **endl** also flushes the buffer.

nounitbuf is in effect by default.

The manipulator effectively calls *_Str*.setf(ios_base::unitbuf), and then returns *_Str*.

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

uppercase

Specifies that hexadecimal digits and the exponent in scientific notation appear in uppercase.

```
ios_base& uppercase(
   ios_base& _Str
);
```

Parameters

_Str

A reference to an object of type ios_base, or to a type that inherits from **ios_base**.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

By default, nouppercase is in effect.

The manipulator effectively calls *_Str*.setf(ios_base::uppercase), and then returns *_Str*.

Example

```
// ios_uppercase.cpp
// compile with: /EHsc
#include <iostream>

int main( void )
{
   using namespace std;

   double i = 1.23e100;
   cout << i << endl;
   cout << uppercase << i << endl;

   int j = 10;
   cout << hex << nouppercase << j << endl;
   cout << hex << uppercase << j << endl;
}
```

Output

```
1.23e+100
1.23E+100
a
A
```

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ios> Members

<ios> Classes

For information about classes in <ios>, see <ios>.

See Also

Reference

iostream Programming

iostreams Conventions

basic_ios Class

The template class describes the storage and member functions common to both input streams (of template class basic_istream) and output streams (of template class basic_ostream) that depend on the template parameters. (The class ios_base describes what is common and not dependent on template parameters. An object of class **basic_ios<class Elem, class Traits>** helps control a stream with elements of type **Elem**, whose character traits are determined by the class **Traits**.

For a list of all members of this type, see basic_ios Members.

```
template <class Elem, class Traits>
   class basic_ios : public ios_base
```

Parameters

Elem

A type.

Traits

A variable of type **char_traits**.

Remarks

An object of class **basic_ios<class Elem, class Traits>** stores:

- A tie pointer to an object of type basic_istream**<Elem, Traits>**.
- A stream buffer pointer to an object of type basic_streambuf**<Elem, Traits >**.
- Formatting information.
- Stream state information in a base object of type ios_base.
- A fill character in an object of type **char_type**.

Requirements

Header: <ios>

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

basic_ios Members
<ios> Members

basic_ios Members

The following tables list the members of the basic_ios Class. For a list of members inherited by the **basic_ios** class, see ios_base Members.

Reference

Constructors

basic_ios	Constructs the **basic_ios** class.

Typedefs

char_type	A synonym for the template parameter *Elem*.
int_type	A synonym for **Traits::int_type**.
off_type	A synonym for **Traits::int_type**.
pos_type	A synonym for **Traits::int_type**.
traits_type	A synonym for the template parameter *Traits*.

Member Functions

bad	Indicates the state of rdstate `() & badbit`.
clear	Clears all error flags.
copyfmt	Copies flags from one stream to another.
eof	Indicates if the end of a stream has been reached.
exceptions	Indicates which exceptions will be thrown by the stream.
fail	Indicates the status of rdstate `() & failbit`.
fill	Specifies or returns the character that will be used when the text is not as wide as the stream.
good	Indicates the state of rdstate `() & goodbit`.
imbue	Changes the locale.
init	Called by **basic_ios** constructors.
narrow	Finds the equivalent char to a given **char_type**.
rdbuf	Routes stream to specified buffer.
rdstate	Reads the state of bits for flags.
setstate	Sets additional flags.
tie	Ensures that one stream is processed before another stream.
widen	Finds the equivalent **char_type** to a given char.

Operators

operator void *	Indicates if the stream is still good.
operator!	Indicates if the stream is not bad.

See Also

Reference

basic_ios Class
ios_base Class
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

ios_base Members

basic_ios Typedefs

For information about the typedefs in the basic_ios class, see basic_ios Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_ios::char_type

A synonym for the template parameter **Elem**.

```
typedef Elem char_type;
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class

iostream Programming

iostreams Conventions

Other Resources

basic_ios Members

basic_ios::int_type

A synonym for **Traits::int_type**.

```
typedef typename Traits::int_type int_type;
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class
iostream Programming
iostreams Conventions

Other Resources

basic_ios Members

basic_ios::off_type

A synonym for **Traits::off_type**.

```
typedef typename Traits::off_type off_type;
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class

iostream Programming

iostreams Conventions

Other Resources

basic_ios Members

basic_ios::pos_type

A synonym for **Traits::pos_type**.

```
typedef typename Traits::pos_type pos_type;
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class

iostream Programming

iostreams Conventions

Other Resources

basic_ios Members

basic_ios::traits_type

A synonym for the template parameter **Traits**.

```
typedef Traits traits_type;
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class

iostream Programming

iostreams Conventions

Other Resources

basic_ios Members

basic_ios Member Functions

For information about the member functions in the basic_ios class, see basic_ios Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_ios::bad

Indicates the state of rdstate **& badbit**.

```
bool bad( ) const;
```

Return Value

1 or **true** if rdstate **& badbit** is nonzero; otherwise 0.

Example

```
// basic_ios_bad.cpp
// compile with: /EHsc
#include <iostream>

int main( void )
{
   using namespace std;
   bool b = cout.bad( );
   cout << b << endl;

   b = cout.good( );
   cout << b << endl;
}
```

Output

```
0
1
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class

iostream Programming

iostreams Conventions

Other Resources

basic_ios Members

basic_ios::basic_ios

Constructs the basic_ios class.

```
explicit basic_ios(
   basic_streambuf<_Elem,
      _Traits> *_Sb
);
```

Parameters

_Sb

Standard buffer to store input or output elements.

Remarks

The first constructor initializes its member objects by calling init(*_Sb*). The second (protected) constructor leaves its member objects uninitialized. A later call to **init** must initialize the object before it can be safely destroyed.

Requirements

Header: <ios>

See Also

Reference

basic_ios Class

iostream Programming

iostreams Conventions

Other Resources

basic_ios Members

basic_ios::clear

Clears all error flags.

```
void clear(
   iostate _State=goodbit
);
```

Parameters

_State (optional)

The flags you want to set after clearing all flags.

Remarks

The flags are **goodbit**, **failbit**, **eofbit**, and **badbit**. Test for these flags with good, bad, eof, and fail

The member function replaces the stored stream state information with

_State | (rdbuf != 0 ? **goodbit** : **badbit**)

If *_State* **&** exceptions is nonzero, it then throws an object of class failure.

Example

See rdstate for an example of using **clear**.

Requirements

Header: <ios>

See Also

Reference

basic_ios Class
iostream Programming
iostreams Conventions

Other Resources

basic_ios Members

basic_ios::copyfmt

Copies flags from one stream to another.

```
basic_ios& copyfmt(
   const basic_ios& _Right
);
```

Parameters

_Right

The stream whose flags you want to copy.

Return Value

The **this** object for the stream to which you are copying the flags.

Remarks

The member function reports the callback event erase_event. It then copies from *_Right* into ***this** the fill character, the tie pointer, and the formatting information. Before altering the exception mask, it reports the callback event copyfmt_event. If, after the copy is complete, **state &** exceptions is nonzero, the function effectively calls clear with the argument rdstate. It returns ***this**.

Example

```
// basic_ios_copyfmt.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
   using namespace std;
   ofstream x( "test.txt" );
   int i = 10;

   x << showpos;
   cout << i << endl;
   cout.copyfmt( x );
   cout << i << endl;
}
```

Output

```
10
+10
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class

iostream Programming

iostreams Conventions

Other Resources

basic_ios Members

basic_ios::eof

Indicates if the end of a stream has been reached.

```
bool eof( ) const;
```

Return Value

1 or **true** if the end of the stream has been reached, 0 or **false** otherwise.

Remarks

boolalpha specifies the format of a **bool** variable.

The member function returns **true** if rdstate **& eofbit** is nonzero.

Example

```
// basic_ios_eof.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

using namespace std;

int main( int argc, char* argv[] )
{
   fstream   fs;
   int n = 1;
   fs.open( "C:\\basic_ios_eof.txt" );   // an empty file

   cout << fs.eof() << endl;
   fs >> n;   // Read the char in the file
   cout << fs.eof() << endl;
}
```

Output

```
0
1
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class

iostream Programming

iostreams Conventions

Other Resources

basic_ios Members

basic_ios::exceptions

Indicates which exceptions will be thrown by the stream.

```
iostate exceptions( ) const;
void exceptions(
   iostate _Newexcept
);
```

Parameters

_Newexcept

The flags that you want to throw an exception.

Return Value

The flags that are currently specified to thrown an exception for the stream.

Remarks

The first member function returns the stored exception mask. The second member function stores *_Except* in the exception mask and returns its previous stored value. Note that storing a new exception mask can throw an exception just like the call clear(rdstate).

Example

```
// basic_ios_exceptions.cpp
// compile with: /EHsc /GR
#include <iostream>

int main( )
{
   using namespace std;

   cout << cout.exceptions( ) << endl;
   cout.exceptions( ios::eofbit );
   cout << cout.exceptions( ) << endl;
   try
   {
      cout.clear( ios::eofbit );   // Force eofbit on
   }
   catch ( exception &e )
   {
      cout.clear( );
      cout << "Caught the exception." << endl;
      cout << "Exception class: " << typeid(e).name()  << endl;
      cout << "Exception description: " << e.what() << endl;
   }
}
```

Output

```
0
1
Caught the exception.
Exception class: class std::ios_base::failure
Exception description: ios_base::eofbit set
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class

iostream Programming

iostreams Conventions

Other Resources

basic_ios Members

basic_ios::fail

Indicates the status of rdstate **& failbit**.

```
bool fail( ) const;
```

Return Value

1 or **true** if rdstate **& failbit** is nonzero, otherwise 0 or **false**.

Remarks

boolalpha specifies the format of a **bool** variable.

Example

```
// basic_ios_fail.cpp
// compile with: /EHsc
#include <iostream>

int main( void )
{
   using namespace std;
   bool b = cout.fail( );
   cout << b << endl;
}
```

Output

```
0
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class

iostream Programming

iostreams Conventions

Other Resources

basic_ios Members

basic_ios::fill

Specifies or returns the character that will be used when the text is not as wide as the stream.

```
char_type fill( ) const;
char_type fill(
   char_type _Char
);
```

Parameters

_Char

The character you want as the fill character.

Return Value

The current fill character.

Remarks

The first member function returns the stored fill character. The second member function stores *_Char* in the fill character and returns its previous stored value.

Example

```
// basic_ios_fill.cpp
// compile with: /EHsc
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;

   cout << setw( 5 ) << 'a' << endl;

   cout.fill( 'x' );
   cout << setw( 5 ) << 'a' << endl;

   cout << cout.fill( ) << endl;
}
```

Output

```
    a
xxxxa
x
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class

iostream Programming

iostreams Conventions

Other Resources

basic_ios Members

basic_ios::good

Indicates the state of rdstate **& goodbit**.

```
bool good( ) const;
```

Return Value

1 or **true** if rdstate **== goodbit** (no state flags are set), otherwise, 0 or **false**.

Remarks

boolalpha specifies the format of a **bool** variable.

The member function returns true if rdstate **== goodbit** (no state flags are set).

Example

See basic_ios::bad for an example of using **good**.

Requirements

Header: <ios>

See Also

Reference

basic_ios Class
iostream Programming
iostreams Conventions

Other Resources

basic_ios Members

basic_ios::imbue

Changes the locale.

```
locale imbue(
   const locale& _Loc
);
```

Parameters

_Loc

A locale string.

Return Value

The previous locale.

Remarks

If rdbuf is not a null pointer, the member function calls

rdbuf->pubimbue(*_Loc*)

In any case, it returns ios_base::imbue(*_Loc*).

Example

```
// basic_ios_imbue.cpp
// compile with: /EHsc
#include <iostream>
#include <locale>

int main( )
{
   using namespace std;

   cout.imbue( locale( "french_france" ) );
   double x = 1234567.123456;
   cout << x << endl;
}
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class

iostream Programming

iostreams Conventions

Other Resources

basic_ios Members

basic_ios::init

Called by basic_ios constructors.

```
void init(
   basic_streambuf<Elem,Traits> *_Sb
);
```

Parameters

_Sb

Standard buffer to store input or output elements.

Remarks

The member function stores values in all member objects, so that:

- rdbuf returns *_Sb*.
- tie returns a null pointer.
- rdstate returns goodbit if *_Sb* is nonzero; otherwise, it returns badbit.
- exceptions returns **goodbit**.
- flags returns skipws | dec.
- width returns 0.
- precision returns 6.
- fill returns the space character.
- getloc returns **locale::classic**.
- iword returns zero, and pword returns a null pointer for all argument values.

Requirements

Header: <ios>

See Also

Reference

basic_ios Class
iostream Programming
iostreams Conventions

Other Resources

basic_ios Members

basic_ios::narrow

Finds the equivalent char to a given **char_type**.

```
char narrow(
   char_type _Char,
   char _Default = '\0'
);
```

Parameters

_Char

The **char** to convert.

_Default

The **char** that you want returned if no equivalent is found.

Return Value

The equivalent **char** to a given **char_type**.

Remarks

The member function returns use_facet**< ctype**<**E**> >(getloc( )). **narrow**(*_Char*, *_Default*).

Example

```
// basic_ios_narrow.cpp
// compile with: /EHsc
#include <ios>
#include <iostream>
#include <wchar.h>

int main( )
{
   using namespace std;
   wchar_t *x = L"test";
   char y[10];
   cout << x[0] << endl;
   wcout << x << endl;
   y[0] = wcout.narrow( x[0] );
   cout << y[0] << endl;
}
```

Output

```
116
test
t
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class

iostream Programming

iostreams Conventions

Other Resources

basic_ios Members

basic_ios::rdbuf

Routes stream to specified buffer.

```
basic_streambuf<Elem, Traits> *rdbuf( ) const;
basic_streambuf<Elem, Traits> *rdbuf(
   basic_streambuf<E, T> *_Sb
);
```

Parameters

_Sb

A stream.

Remarks

The first member function returns the stored stream buffer pointer.

The second member function stores *_Sb* in the stored stream buffer pointer and returns the previously stored value.

Example

```
// basic_ios_rdbuf.cpp
// compile with: /EHsc
#include <ios>
#include <iostream>
#include <fstream>

int main( )
{
   using namespace std;
   ofstream file( "rdbuf.txt" );
   streambuf *x = cout.rdbuf( file.rdbuf( ) );
   cout << "test" << endl;   // Goes to file
   cout.rdbuf(x);
   cout << "test2" << endl;
}
```

Output

```
test2
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class

iostream Programming

iostreams Conventions

Other Resources

basic_ios Members

basic_ios::rdstate

Reads the state of bits for flags.

```
iostate rdstate( ) const;
```

Return Value

The stored stream state information.

Example

```
// basic_ios_rdstate.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>
using namespace std;

void TestFlags( ios& x )
{
   cout << ( x.rdstate( ) & ios::badbit ) << endl;
   cout << ( x.rdstate( ) & ios::failbit ) << endl;
   cout << ( x.rdstate( ) & ios::eofbit ) << endl;
   cout << endl;
}

int main( )
{
   fstream x( "c:\test.txt", ios::out );
   x.clear( );
   TestFlags( x );
   x.clear( ios::badbit | ios::failbit | ios::eofbit );
   TestFlags( x );
}
```

Output

```
0
0
0

4
2
1
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class

iostream Programming

iostreams Conventions

Other Resources

basic_ios Members

basic_ios::setstate

Sets additional flags.

```
void setstate(
   iostate _State
);
```

Parameters

_State

Additional flags to set.

Remarks

The member function effectively calls clear(*_State* | rdstate).

Example

```
// basic_ios_setstate.cpp
// compile with: /EHsc
#include <ios>
#include <iostream>

int main( )
{
   using namespace std;
   bool b = cout.bad( );
   cout << b << endl;   // Good
   cout.clear( ios::badbit );
   b = cout.bad( );
   // cout.clear( );
   cout << b << endl;   // Is bad, good
   b = cout.fail( );
   cout << b << endl;   // Not failed
   cout.setstate( ios::failbit );
   b = cout.fail( );
   cout.clear( );
   cout << b << endl;   // Is failed, good
}
```

Output

```
0
1
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class

iostream Programming

iostreams Conventions

Other Resources

basic_ios Members

basic_ios::tie

Ensures that one stream is processed before another stream.

```
basic_ostream<Elem, Traits> *tie( ) const;
basic_ostream<Elem, Traits> *tie(
   basic_ostream<E, T> *_Str
);
```

Parameters

_Str

A stream.

Return Value

The first member function returns the stored tie pointer. The second member function stores *_Str* in the tie pointer and returns its previous stored value.

Remarks

tie causes two streams to be synchronized, such that, operations on one stream occur after operations on the other stream are complete.

Example

In this example, by tying cin to cout, it is guaranteed that the "Enter a number:" string will go to the console before the number itself is extracted from cin. This eliminates the possibility that the "Enter a number:" string is still sitting in the buffer when the number is read, so that we are certain that the user actually has some prompt to respond to. By default, cin and cout are tied.

```
#include <ios>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
   cin.tie( &cout );
   cout << "Enter a number:";
   cin >> i;
}
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class

iostream Programming

iostreams Conventions

Other Resources

basic_ios Members

basic_ios::widen

Finds the equivalent **char_type** to a given **char**.

```
char_type widen(
   char _Char
);
```

Parameters

_Char

The character to convert.

Return Value

Finds the equivalent **char_type** to a given **char**.

Remarks

The member function returns use_facet< **ctype**<**E**> >(getloc). **widen**(*_Char*).

Example

```
// basic_ios_widen.cpp
// compile with: /EHsc
#include <ios>
#include <iostream>
#include <wchar.h>

int main( )
{
   using namespace std;
   char *z = "Hello";
   wchar_t y[2] = {0,0};
   cout << z[0] << endl;
   y[0] = wcout.widen( z[0] );
   wcout << &y[0] << endl;
}
```

Output

```
H
H
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class

iostream Programming

iostreams Conventions

Other Resources

basic_ios Members

basic_ios Operators

For more information about the operators in basic_ios class, see basic_ios Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_ios::operator void *

Indicates if the stream is still good.

```
operator void *( ) const;
```

Return Value

The operator returns a null pointer only if fail.

Example

```
// basic_ios_opgood.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   cout << (bool)(&cout != 0) << endl;   // Stream is still good
}
```

Output

```
1
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class
iostream Programming
iostreams Conventions

Other Resources

basic_ios Members

basic_ios::operator!

Indicates if the stream is not bad.

```
bool operator!( ) const;
```

Return Value

If the stream is bad, returns fail.

Example

```
// basic_ios_opbad.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   cout << !cout << endl;   // Stream is not bad
}
```

Output

```
0
```

Requirements

Header: <ios>

See Also

Reference

basic_ios Class
iostream Programming
iostreams Conventions

Other Resources

basic_ios Members

fpos Class

The template class describes an object that can store all the information needed to restore an arbitrary file-position indicator within any stream. An object of class fpos<**St**> effectively stores at least two member objects:

- A byte offset, of type streamoff.
- A conversion state, for use by an object of class basic_filebuf, of type **St**, typically **mbstate_t**.

It can also store an arbitrary file position, for use by an object of class basic_filebuf, of type **fpos_t**. For an environment with limited file size, however, **streamoff** and **fpos_t** may sometimes be used interchangeably. For an environment with no streams that have a state-dependent encoding, **mbstate_t** may actually be unused. Therefore, the number of member objects stored may vary.

For a list of all members of this type, see fpos Members.

```
template <class Statetype>
   class fpos
```

Parameters

Statetype

State information.

Requirements

Header: <ios>

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

fpos Members
<ios> Members

fpos Members

Reference

Constructors

fpos	Create an object that contains information about a position (offset) in a stream.

Member Functions

state	Sets or returns the conversion state.

Operators

operator!=	Tests file-position indicators for inequality.
operator+	Increments a file-position indicator.
operator+=	Increments a file-position indicator.
operator-	Decrements a file-position indicator.
operator-=	Decrements a file-position indicator.
operator==	Tests file-position indicators for equality.
operator streamoff	Cast object of type **fpos** to object of type **streamoff**.

See Also

Reference

fpos Class
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

fpos Member Functions

For information about the member functions in the fpos class, see fpos Members.

See Also

Reference

iostream Programming

iostreams Conventions

fpos::fpos

Create an object that contains information about a position (offset) in a stream.

```
fpos(
   streamoff _Off=0
);
fpos(
   Statetype _State,
   fpos_t _Filepos
);
```

Parameters

_Off

The offset into the stream.

_State

The starting state of the **fpos** object.

_Filepos

The offset into the stream.

Remarks

The first constructor stores the offset *_Off*, relative to the beginning of file and in the initial conversion state (if that matters). If *_Off* is -1, the resulting object represents an invalid stream position.

The second constructor stores a zero offset and the object *_State*.

Requirements

Header: <ios>

See Also

Reference

fpos Class

iostream Programming

iostreams Conventions

Other Resources

fpos Members

fpos::state

Sets or returns the conversion state.

```
Statetype state( ) const;
void state(
   Statetype _State
);
```

Parameters

_State

The new conversion state.

Return Value

The conversion state.

Remarks

The first member function returns the value stored in the **St** member object. The second member function stores *_State* in the **St** member object.

Example

```
// fpos_state.cpp
// compile with: /EHsc
#include <ios>
#include <iostream>
#include <fstream>

int main() {
   using namespace std;
   streamoff s;
   ifstream file( "fpos_state.txt" );

   fpos<mbstate_t> f = file.tellg( );
   char ch;
   while ( !file.eof( ) )
      file.get( ch );
   s = f;
   cout << f.state( ) << endl;
   f.state( 9 );
   cout << f.state( ) << endl;
}
```

Input: fpos_state.txt

```
testing
```

Output

```
0
9
```

Requirements

Header: <ios>

See Also

Reference

fpos Class

iostream Programming

iostreams Conventions

Other Resources

fpos Operators

For more information about the operators in the fpos class, see fpos Members.

See Also

Reference

iostream Programming

iostreams Conventions

fpos::operator!=

Tests file-position indicators for inequality.

```
bool operator!=(
   const fpos& _Right
) const;
```

Parameters

_Right

The file-position indicator against which to compare.

Return Value

true if the file-position indicators are not equal, otherwise **false**.

Remarks

The member function returns **!**(***this ==** *_Right*).

Example

```
// fpos_op_neq.cpp
// compile with: /EHsc
#include <fstream>
#include <iostream>

int main( )
{
   using namespace std;

   fpos<int> pos1, pos2;
   ifstream file;
   char c;

   // Compare two fpos object
   if ( pos1 != pos2 )
      cout << "File position pos1 and pos2 are not equal" << endl;
   else
      cout << "File position pos1 and pos2 are equal" << endl;

   file.open( "fpos_op_neq.txt" );
   file.seekg( 0 );   // Goes to a zero-based position in the file
   pos1 = file.tellg( );
   file.get( c);
   cout << c << endl;

   // Increment pos1
   pos1 += 1;
   file.get( c );
   cout << c << endl;

   pos1 = file.tellg( ) - fpos<int>( 2);
   file.seekg( pos1 );
   file.get( c );
   cout << c << endl;

   // Increment pos1
   pos1 = pos1 + fpos<int>( 1 );
   file.get(c);
   cout << c << endl;

   pos1 -= fpos<int>( 2 );
   file.seekg( pos1 );
```

```
    file.get( c );
    cout << c << endl;

    file.close( );
}
```

Input: fpos_op_neq.txt

```
987654321
```

Output

```
File position pos1 and pos2 are equal
9
8
9
8
8
```

Requirements

Header: <ios>

See Also

Reference

fpos Class

iostream Programming

iostreams Conventions

Other Resources

fpos Members

fpos::operator+

Increments a file-position indicator.

```
fpos operator+(
   streamoff _Off
) const;
```

Parameters

_Off

The offset by which you want to increment the file-position indicator.

Return Value

The position in the file.

Remarks

The member function returns **fpos(*this) +=** *_Off*.

Example

See operator!= for a sample of using **operator+**.

Requirements

Header: <ios>

See Also

Reference

fpos Class
iostream Programming
iostreams Conventions

Other Resources

fpos Members

fpos::operator+=

Increments a file-position indicator.

```
fpos& operator+=(
   streamoff _Off
);
```

Parameters

_Off

The offset by which you want to increment the file-position indicator.

Return Value

The position in the file.

Remarks

The member function adds *_Off* to the stored offset member object and then returns ***this**. For positioning within a file, the result is generally valid only for binary streams that do not have a state-dependent encoding.

Example

See operator!= for a sample of using **operator+=**.

Requirements

Header: <ios>

See Also

Reference

fpos Class

iostream Programming

iostreams Conventions

Other Resources

fpos Members

fpos::operator-

Decrements a file-position indicator.

```
streamoff operator-(
   const fpos& _Right
) const;
fpos operator-(
   streamoff _Off
) const;
```

Parameters

_Right

File position.

_Off

Stream offset.

Return Value

The first member function returns **(streamoff)*this - (streamoff)***_Right*. The second member function returns **fpos(*this) -=** *_Off*.

Example

See operator!= for a sample of using **operator-**.

Requirements

Header: <ios>

See Also

Reference

fpos Class

iostream Programming

iostreams Conventions

Other Resources

fpos Members

fpos::operator-=

Decrements a file-position indicator.

```
fpos& operator-=(
   streamoff _Off
);
```

Parameters

_Off

Stream offset.

Return Value

The member function returns **fpos(*this) -=** *_Off*.

Remarks

For positioning within a file, the result is generally valid only for binary streams that do not have a state-dependent encoding.

Example

See operator!= for a sample of using **operator-=**.

Requirements

Header: <ios>

See Also

Reference

fpos Class
iostream Programming
iostreams Conventions

Other Resources

fpos Members

fpos::operator==

Tests file-position indicators for equality.

```
bool operator==(
   const fpos& _Right
) const;
```

Parameters

_Right

The file-position indicator against which to compare.

Return Value

true if the file-position indicators are equal; otherwise **false**.

Remarks

The member function returns **(streamoff)*this == (streamoff)***_Right*.

Example

See operator!= for a sample of using **operator+=**.

Requirements

Header: <ios>

See Also

Reference

fpos Class
iostream Programming
iostreams Conventions

Other Resources

fpos Members

fpos::operator streamoff

Cast object of type **fpos** to object of type **streamoff**.

```
operator streamoff( ) const;
```

Remarks

The member function returns the stored offset member object and any additional offset stored as part of the **fpos_t** member object.

Example

```
// fpos_op_streampos.cpp
// compile with: /EHsc
#include <ios>
#include <iostream>
#include <fstream>

int main( )
{
   using namespace std;
   streamoff s;
   ofstream file( "rdbuf.txt");

   fpos<mbstate_t> f = file.tellp( );
   // Is equivalent to ..
   // streampos f = file.tellp( );
   s = f;
   cout << s << endl;
}
```

Output

```
0
```

Requirements

Header: <ios>

See Also

Reference

fpos Class

iostream Programming

iostreams Conventions

Other Resources

fpos Members

ios_base Class

The class describes the storage and member functions common to both input and output streams that do not depend on the template parameters. (The template class basic_ios describes what is common and is dependent on template parameters.)

An object of class ios_base stores formatting information, which consists of:

- Format flags in an object of type fmtflags.
- An exception mask in an object of type iostate.
- A field width in an object of type **int**.
- A display precision in an object of type **int**.
- A locale object in an object of type **locale**.
- Two extensible arrays, with elements of type **long** and **void** pointer.

An object of class ios_base also stores stream state information, in an object of type iostate, and a callback stack.

For a list of all members of this type, see ios_base Members.

Requirements

Header: <ios>

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

ios_base Members
<ios> Members

ios_base Members

Reference

Constructors

ios_base	Constructs **ios_base** objects.

Typedefs

event	Specifies event types.
event_callback	Describes a function passed to register_call.
fmtflags	Constants to specify the appearance of output.
iostate	Defines constants describing the state of a stream.
openmode	Describes how to interact with a stream.
seekdir	Specifies starting point for offset operations.

Member Functions

failure	The member class serves as the base class for all exceptions thrown by the member function clear in templat e class basic_ios.
flags	Sets or returns the current flag settings.
getloc	Returns the stored locale object.
imbue	Changes the locale.
Init	Creates the standard iostream objects when constructed.
iword	Assigns a value to be stored as an **iword**.
precision	Specifies the number of digits to display in a floating-point number.
pword	Assigns a value to be stored as a **pword**.
register_callback	Specifies a callback function.
setf	Sets the specified flags.
sync_with_stdio	Ensures that iostream and C run-time library operations occur in the order that they appear in source code.
unsetf	Causes the specified flags to be off.
width	Sets the length of the output stream.
xalloc	Specifies that a variable shall be part of the stream.

Operators

operator=	The assignment operator for **ios_base** objects.

See Also

Reference

ios_base Class
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

ios_base Typedefs

For information about the typedefs in the ios_base class, see ios_base Members.

See Also

Reference

iostream Programming

iostreams Conventions

ios_base::event

Specifies event types.

```
enum event (copyfmt_event, erase_event, imbue_event);
```

Remarks

The type is an enumerated type **T5** that describes an object that can store the callback event used as an argument to a function registered with register_callback. The distinct event values are:

- **copyfmt_event**, to identify a callback that occurs near the end of a call to copyfmt, just before the exception mask is copied.
- **erase_event**, to identify a callback that occurs at the beginning of a call to copyfmt, or at the beginning of a call to the destructor for ***this**.
- **imbue_event**, to identify a callback that occurs at the end of a call to imbue, just before the function returns.

Example

See register_callback for an example.

Requirements

Header: <ios>

See Also

Reference

iostream Programming

iostreams Conventions

ios_base::event_callback

Describes a function passed to register_call.

```
typedef void (__cdecl *event_callback)(
   event _E,
   ios_base& _Base,
   int _I
);
```

Parameters

_E

The event.

_Base

The stream in which the event was called.

_I

A user-defined number.

Remarks

The type describes a pointer to a function that can be registered with register_callback. This type of function must not throw an exception.

Example

See register_call for an example that uses **event_callback**.

Requirements

Header: <ios>

See Also

Reference

ios_base Class
iostream Programming
iostreams Conventions

Other Resources

ios_base Members

ios_base::fmtflags

Constants to specify the appearance of output.

```
typedef T1 fmtflags;
static const fmtflags boolalpha, dec, fixed, hex,
   internal, left, oct, right, scientific,
   showbase, showpoint, showpos, skipws, unitbuf,
   uppercase, adjustfield, basefield, floatfield;
```

Remarks

Supports the manipulators in ios.

The type is a bitmask type **T1** that describes an object that can store format flags. The distinct flag values (elements) are:

- **dec**, to insert or extract integer values in decimal format.
- **hex**, to insert or extract integer values in hexadecimal format.
- **oct**, to insert or extract integer values in octal format.
- **showbase**, to insert a prefix that reveals the base of a generated integer field.
- **internal**, to pad to a field width as needed by inserting fill characters at a point internal to a generated numeric field. (For information on setting the field width, see setw).
- **left**, to pad to a field width as needed by inserting fill characters at the end of a generated field (left justification).
- **right**, to pad to a field width as needed by inserting fill characters at the beginning of a generated field (right justification).
- **boolalpha**, to insert or extract objects of type **bool** as names (such as **true** and **false**) rather than as numeric values.
- **fixed**, to insert floating-point values in fixed-point format (with no exponent field).
- **scientific**, to insert floating-point values in scientific format (with an exponent field).
- **showpoint**, to insert a decimal point unconditionally in a generated floating-point field.
- **showpos**, to insert a plus sign in a nonnegative generated numeric field.
- **skipws**, to skip leading white space before certain extractions.
- **unitbuf**, to flush output after each insertion.
- **uppercase**, to insert uppercase equivalents of lowercase letters in certain insertions.

In addition, several useful values are:

- **adjustfield**, a bitmask defined as **internal** | **left** | **right**
- **basefield**, defined as **dec** | **hex** | **oct**
- **floatfield**, defined as **fixed** | **scientific**

For examples of functions that modify these format flags, see <ios> Members (under Manipulators) and <iomanip> Members.

Requirements

Header: <ios>

See Also

Reference

ios_base Class

iostream Programming

iostreams Conventions

Other Resources

ios_base::iostate

Defines constants describing the state of a stream.

```
typedef T2 iostate;
static const iostate badbit, eofbit, failbit, goodbit;
```

Remarks

The type is a bitmask type **T2** that describes an object that can store stream state information. The distinct flag values (elements) are:

- **badbit**, to record a loss of integrity of the stream buffer.
- **eofbit**, to record end-of-file while extracting from a stream.
- **failbit**, to record a failure to extract a valid field from a stream.

In addition, a useful value is **goodbit**, where none of the previously mentioned bits are set.

Requirements

Header: <ios>

See Also

Reference

ios_base Class
iostream Programming
iostreams Conventions

Other Resources

ios_base Members

ios_base::openmode

Describes how to interact with a stream.

```
typedef T3 openmode;
static const openmode app, ate, binary, in, out, trunc;
```

Remarks

The type is a **bitmask type T3** that describes an object that can store the opening mode for several iostreams objects. The distinct flag values (elements) are:

- **app**, to seek to the end of a stream before each insertion.
- **ate**, to seek to the end of a stream when its controlling object is first created.
- **binary**, to read a file as a binary stream, rather than as a text stream.
- **in**, to permit extraction from a stream.
- **out**, to permit insertion to a stream.
- **trunc**, to delete contents of an existing file when its controlling object is created.

Example

```
// ios_base_openmode.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main ( )
{
   using namespace std;
   fstream file;
   file.open( "rm.txt", ios_base::out | ios_base::trunc );

   file << "testing";
}
```

Requirements

Header: <ios>

See Also

Reference

ios_base Class

iostream Programming

iostreams Conventions

Other Resources

ios_base Members

ios_base::seekdir

Specifies starting point for offset operations.

```
typedef T4 seekdir;
static const seekdir beg, cur, end;
```

Remarks

The type is an enumerated type **T4** that describes an object that can store the seek mode used as an argument to the member functions of several iostream classes. The distinct flag values are:

- **beg**, to seek (alter the current read or write position) relative to the beginning of a sequence (array, stream, or file).
- **cur**, to seek relative to the current position within a sequence.
- **end**, to seek relative to the end of a sequence.

Example

```
// ios_base_seekdir.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main ( )
{
   using namespace std;
   fstream file;
   file.open( "rm.txt", ios_base::out | ios_base::trunc );

   file << "testing";
   file.seekp( 0, ios_base::beg );
   file << "a";
   file.seekp( 0, ios_base::end );
   file << "a";
}
```

Contents of File

```
aestinga
```

Requirements

Header: <ios>

See Also

Reference

ios_base Class
iostream Programming
iostreams Conventions

Other Resources

ios_base Members

ios_base Member Functions

For information about the member functions in the ios_base class, see ios_base Members.

See Also

Reference

iostream Programming

iostreams Conventions

ios_base::failure

The member class serves as the base class for all exceptions thrown by the member function clear in template class basic_ios.

```
class failure : public exception {
public:
   explicit failure(
       const string& _Message)
   };
```

Remarks

The value returned by the member function **what()** is **_Message.**c_str**()**.

Example

```
// ios_base_failure.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main ( )
{
   using namespace std;
   fstream file;
   file.exceptions(ios::failbit);
   try
   {
      file.open( "rm.txt", ios_base::in );
      // Opens nonexistent file for reading
   }
   catch( ios_base::failure f )
   {
      cout << "Caught an exception: " << f.what() << endl;
   }
}
```

Output

```
Caught an exception: ios_base::failbit set
```

Requirements

Header: <ios>

See Also

Reference

ios_base Class

iostream Programming

iostreams Conventions

Other Resources

ios_base Members

ios_base::flags

Sets or returns the current flag settings.

```
fmtflags flags( ) const;
fmtflags flags(
      fmtflags _Fmtfl
);
```

Parameters

_Fmtfl

The new **fmtflags** setting.

Return Value

The previous or current **fmtflags** setting.

Remarks

See the manipulators section for ios for a list of the flags.

The first member function returns the stored format flags. The second member function stores *_Fmtfl* in the format flags and returns its previous stored value.

Example

```
// ios_base_flags.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main ( )
{
   using namespace std;
   cout << cout.flags( ) << endl;
   cout.flags( ios::dec | ios::boolalpha );
   cout << cout.flags( );
}
```

Output

```
513
16896
```

Requirements

Header: <ios>

See Also

Reference

ios_base Class

iostream Programming

iostreams Conventions

Other Resources

ios_base Members

ios_base::getloc

Returns the stored locale object.

```
locale getloc( ) const;
```

Return Value

The stored locale object.

Example

```
// ios_base_getlock.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   cout << cout.getloc( ).name( ).c_str( ) << endl;
}
```

Output

```
C
```

Requirements

Header: <ios>

See Also

Reference

ios_base Class
iostream Programming
iostreams Conventions

Other Resources

ios_base Members

ios_base::imbue

Changes the locale.

```
locale imbue(
   const locale& _Loc
);
```

Parameters

_Loc

The new locale setting.

Return Value

The previous locale.

Remarks

The member function stores *_Loc* in the locale object and then reports the callback event and **imbue_event**. It returns the previous stored value.

Example

See basic_ios::imbue for a sample.

Requirements

Header: <ios>

See Also

Reference

ios_base Class
iostream Programming
iostreams Conventions

Other Resources

ios_base Members

ios_base::Init

Creates the standard iostream objects when constructed.

```
class Init { };
```

Remarks

The nested class describes an object whose construction ensures that the standard iostreams objects are properly constructed, even before the execution of a constructor for an arbitrary static object.

Requirements

Header: <ios>

See Also

Reference

ios_base Class
iostream Programming
iostreams Conventions

Other Resources

ios_base Members

ios_base::ios_base

Constructs ios_base objects.

```
ios_base( );
```

Remarks

The (protected) constructor does nothing. A later call to **basic_ios::**init must initialize the object before it can be safely destroyed. Thus, the only safe use for class ios_base is as a base class for template class basic_ios.

Requirements

Header: <ios>

See Also

Reference

ios_base Class
iostream Programming
iostreams Conventions

Other Resources

ios_base Members

ios_base::iword

Assigns a value to be stored as an **iword**.

```
long& iword(
   int _Idx
);
```

Parameters

_Idx

The index of the value to store as a **pword**.

Remarks

The member function returns a reference to element *_Idx* of the extensible array with elements of type **long**. All elements are effectively present and initially store the value zero. The returned reference is invalid after the next call to **iword** for the object, after the object is altered by a call to **basic_ios::**copyfmt, or after the object is destroyed.

If *_Idx* is negative or if unique storage is unavailable for the element, the function calls setstate**(badbit)** and returns a reference that might not be unique.

To obtain a unique index, for use across all objects of type **ios_base**, call xalloc.

Example

See xalloc for a sample of how to use **iword**.

Requirements

Header: <ios>

See Also

Reference

ios_base Class

iostream Programming

iostreams Conventions

Other Resources

ios_base Members

ios_base::precision

Specifies the number of digits to display in a floating-point number.

```
streamsize precision( ) const;
streamsize precision(
   streamsize _Prec
);
```

Parameters

_Prec

The number of significant digits to display, or the number of digits after the decimal point in fixed notation.

Return Value

The first member function returns the stored display precision. The second member function stores *_Prec* in the display precision and returns its previous stored value.

Remarks

Floating-point numbers are displayed in fixed notation with fixed.

Example

```
// ios_base_precision.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   float i = 31.31234F;

   cout.precision( 3 );
   cout << i << endl;          // display three significant digits
   cout << fixed << i << endl; // display three digits after decimal
                               // point
}
```

Output

```
31.3
31.312
```

Requirements

Header: <ios>

See Also

Reference

ios_base Class

iostream Programming

iostreams Conventions

Other Resources

ios_base Members

ios_base::pword

Assigns a value to be stored as a **pword**.

```
void *& pword(
   int _Idx
);
```

Parameters

_Idx

The index of the value to store as a **pword**.

Remarks

The member function returns a reference to element *_Idx* of the extensible array with elements of type **void** pointer. All elements are effectively present and initially store the null pointer. The returned reference is invalid after the next call to **pword** for the object, after the object is altered by a call to **basic_ios::**copyfmt, or after the object is destroyed.

If *_Idx* is negative, or if unique storage is unavailable for the element, the function calls setstate**(badbit)** and returns a reference that might not be unique.

To obtain a unique index, for use across all objects of type **ios_base**, call xalloc.

Example

See xalloc for an example of using **pword**.

Requirements

Header: <ios>

See Also

Reference

ios_base Class

iostream Programming

iostreams Conventions

Other Resources

ios_base Members

ios_base::register_callback

Specifies a callback function.

```
void register_callback(
   event_callback _Pfn,
   int _Idx
);
```

Parameters

_Pfn

Pointer to the callback function.

_Idx

A user-defined number.

Remarks

The member function pushes the pair {*_Pfn*, *_Idx*} onto the stored callback stack callback stack. When a callback event **ev** is reported, the functions are called, in reverse order of registry, by the expression ***_Pfn*)(**ev**, ***this**, *_Idx*).

Example

```
// ios_base_register_callback.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

using namespace std;

void callback1( ios_base::event e, ios_base& stream, int arg )
{
   cout << "in callback1" << endl;
   switch ( e )
   {
      case ios_base::erase_event:
         cout << "an erase event" << endl;
         break;
      case ios_base::imbue_event:
         cout << "an imbue event" << endl;
         break;
      case ios_base::copyfmt_event:
         cout << "an copyfmt event" << endl;
         break;
   };
}

void callback2( ios_base::event e, ios_base& stream, int arg )
{
   cout << "in callback2" << endl;
   switch ( e )
   {
      case ios_base::erase_event:
         cout << "an erase event" << endl;
         break;
      case ios_base::imbue_event:
         cout << "an imbue event" << endl;
         break;
      case ios_base::copyfmt_event:
         cout << "an copyfmt event" << endl;
         break;
   };
}
```

```
int main( )
{
   // Make sure the imbue will not throw an exception
   // assert( setlocale( LC_ALL, "german" )!=NULL );

   cout.register_callback( callback1, 0 );
   cin.register_callback( callback2, 0 );

   try
   {
      // If no exception because the locale's not found,
      // generate an imbue_event on callback1
      cout.imbue(locale("german"));
   }
   catch(...)
   {
      cout << "exception" << endl;
   }

   // This will
   // (1) erase_event on callback1
   // (2) copyfmt_event on callback2
   cout.copyfmt(cin);

   // We get two erase events from callback2 at the end because
// both cin and cout have callback2 registered when cin and cout
   // are destroyed at the end of program.
}
```

Output

```
in callback1
an imbue event
in callback1
an erase event
in callback2
an copyfmt event
in callback2
an erase event
in callback2
an erase event
```

Requirements

Header: <ios>

See Also

Reference

ios_base Class

iostream Programming

iostreams Conventions

Other Resources

ios_base Members

ios_base::setf

Sets the specified flags.

```
void setf(
   fmtflags _Mask
);
fmtflags setf(
      fmtflags _Mask,
   fmtflags _Unset
);
```

Parameters

_Mask

The flags to turn on.

_Unset

The flags to turn off.

Return Value

The previous format flags

Remarks

The first member function effectively calls flags(*_Mask* | *_Flags*) (set selected bits) and then returns the previous format flags. The second member function effectively calls **flags**(*_Mask* **& fmtfl, flags & ~***_Mask*) (replace selected bits under a mask) and then returns the previous format flags.

Example

```
// ios_base_setf.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   int i = 10;
   cout << i << endl;

   cout.unsetf( ios_base::dec );
   cout.setf( ios_base::hex );
   cout << i << endl;

   cout.setf( ios_base::dec );
   cout << i << endl;
   cout.setf( ios_base::hex, ios_base::dec );
   cout << i << endl;
}
```

Output

```
10
a
10
a
```

Requirements

Header: <ios>

See Also

Reference

ios_base Class
iostream Programming
iostreams Conventions
Other Resources
ios_base Members

ios_base::sync_with_stdio

Ensures that iostream and C run-time library operations occur in the order that they appear in source code.

```
static bool sync_with_stdio(
   bool _Sync=true
);
```

Parameters

_Sync

Whether all streams are in sync with **stdio**.

Return Value

Previous setting for this function.

Remarks

The static member function stores a **stdio** sync flag, which is initially **true**. When **true**, this flag ensures that operations on the same file are properly synchronized between the iostreams functions and those defined in the Standard C++ Library. Otherwise, synchronization may or may not be guaranteed, but performance may be improved. The function stores *_Sync* in the **stdio** sync flag and returns its previous stored value. You can call it reliably only before performing any operations on the standard streams.

Requirements

Header: <ios>

See Also

Reference

ios_base Class
iostream Programming
iostreams Conventions

Other Resources

ios_base Members

ios_base::unsetf

Causes the specified flags to be off.

```
void unsetf(
   fmtflags _Mask
);
```

Parameters

_Mask

The flags that you want off.

Remarks

The member function effectively calls flags(**~***_Mask* **& flags**) (clear selected bits).

Example

See ios_base::setf for a sample of using **unsetf**.

Requirements

Header: <ios>

See Also

Reference

ios_base Class

iostream Programming

iostreams Conventions

Other Resources

ios_base Members

ios_base::width

Sets the length of the output stream.

```
streamsize width( ) const;
streamsize width(
   streamsize _Wide
);
```

Parameters

_Wide

The desired size of the output stream.

Return Value

The current width setting.

Remarks

The first member function returns the stored field width. The second member function stores *_Wide* in the field width and returns its previous stored value.

Example

```
// ios_base_width.cpp
// compile with: /EHsc
#include <iostream>

int main( ) {
   using namespace std;

   cout.width( 20 );
   cout << cout.width( ) << endl;
   cout << cout.width( ) << endl;
}
```

Output

```
                    20
0
```

Requirements

Header: <ios>

See Also

Reference

ios_base Class

iostream Programming

iostreams Conventions

Other Resources

ios_base Members

ios_base::xalloc

Specifies that a variable is part of the stream.

```
static int xalloc( );
```

Return Value

The static member function returns a stored static value, which it increments on each call.

Remarks

You can use the return value as a unique index argument when calling the member functions iword or pword.

Example

```
// ios_base_xalloc.cpp
// compile with: /EHsc
// Lets you store user-defined information.
// iword, jword, xalloc
#include <iostream>

int main( )
{
   using namespace std;

   static const int i = ios_base::xalloc();
   static const int j = ios_base::xalloc();
   cout.iword( i ) = 11;
   cin.iword( i ) = 13;
   cin.pword( j ) = "testing";
   cout << cout.iword( i ) << endl;
   cout << cin.iword( i ) << endl;
   cout << ( char * )cin.pword( j ) << endl;
}
```

Output

```
11
13
testing
```

Requirements

Header: <ios>

See Also

Reference

ios_base Class

iostream Programming

iostreams Conventions

Other Resources

ios_base Members

ios_base Operators

For more information about the operators in the ios_base class, see ios_base Members.

See Also

Reference

iostream Programming

iostreams Conventions

ios_base::operator=

The assignment operator for ios_base objects.

```
ios_base& operator=(
   const ios_base& _Right
);
```

Parameters

_Right

An object of type **ios_base**.

Return Value

The object being assigned to.

Remarks

The operator copies the stored formatting information, making a new copy of any extensible arrays. It then returns ***this**. Note that the callback stack is not copied.

This operator is only used by classes derived from **ios_base**.

Requirements

Header: <ios>

See Also

Reference

ios_base Class
iostream Programming
iostreams Conventions

Other Resources

ios_base Members

<iosfwd>

Declares forward references to several template classes used throughout iostreams. All such template classes are defined in other standard headers. You include this header explicitly only when you need one of its declarations, but not its definition.

```
#include <iosfwd>
```

Typedefs

```
typedef T1 streamoff;
typedef T2 streamsize;
typedef fpos streampos;

// wchar_t TYPE DEFINITIONS
typedef basic_ios<char, char_traits<char> > ios;
typedef basic_streambuf<char, char_traits<char> > streambuf;
typedef basic_istream<char, char_traits<char> > istream;
typedef basic_ostream<char, char_traits<char> > ostream;
typedef basic_iostream<char, char_traits<char> > iostream;
typedef basic_stringbuf<char, char_traits<char> > stringbuf;
typedef basic_istringstream<char, char_traits<char> > istringstream;
typedef basic_ostringstream<char, char_traits<char> > ostringstream;
typedef basic_stringstream<char, char_traits<char> > stringstream;
typedef basic_filebuf<char, char_traits<char> > filebuf;
typedef basic_ifstream<char, char_traits<char> > ifstream;
typedef basic_ofstream<char, char_traits<char> > ofstream;
typedef basic_fstream<char, char_traits<char> > fstream;

// wchar_t TYPE DEFINITIONS
typedef basic_ios<wchar_t, char_traits<wchar_t> > wios;
typedef basic_streambuf<wchar_t, char_traits<wchar_t> > wstreambuf;
typedef basic_istream<wchar_t, char_traits<wchar_t> > wistream;
typedef basic_ostream<wchar_t, char_traits<wchar_t> > wostream;
typedef basic_iostream<wchar_t, char_traits<wchar_t> > wiostream;
typedef basic_stringbuf<wchar_t, char_traits<wchar_t> > wstringbuf;
typedef basic_istringstream<wchar_t, char_traits<wchar_t> > wistringstream;
typedef basic_ostringstream<wchar_t, char_traits<wchar_t> > wostringstream;
typedef basic_stringstream<wchar_t, char_traits<wchar_t> > wstringstream;
typedef basic_filebuf<wchar_t, char_traits<wchar_t> > wfilebuf;
typedef basic_ifstream<wchar_t, char_traits<wchar_t> > wifstream;
typedef basic_ofstream<wchar_t, char_traits<wchar_t> > wofstream;
typedef basic_fstream<wchar_t, char_traits<wchar_t> > wfstream;
};
```

Forward Declarations/Template Classes

```
template<class Elem>
class char_traits;

class char_traits<char>;

class char_traits<wchar_t>;

template<class Elem, class Tr = char_traits<Elem> >
class basic_ios;

template<class Elem, class Tr = char_traits<Elem> >
class istreambuf_iterator;

template<class Elem, class Tr = char_traits<Elem> >
class ostreambuf_iterator;

template<class Elem, class Tr = char_traits<Elem> >
```

```
class basic_streambuf;

template<class Elem, class Tr = char_traits<Elem> >
class basic_istream;

template<class Elem, class Tr = char_traits<Elem> >
class basic_ostream;

template<class Elem, class Tr = char_traits<Elem> >
class basic_iostream;

template<class Elem, class Tr = char_traits<Elem> >
class basic_stringbuf;

template<class Elem, class Tr = char_traits<Elem> >
class basic_istringstream;

template<class Elem, class Tr = char_traits<Elem> >
class basic_ostringstream;

template<class Elem, class Tr = char_traits<Elem> >
class basic_stringstream;

template<class Elem, class Tr = char_traits<Elem> >
class basic_filebuf;

template<class Elem, class Tr = char_traits<Elem> >
class basic_ifstream;

template<class Elem, class Tr = char_traits<Elem> >
class basic_ofstream;

template<class Elem, class Tr = char_traits<Elem> >
class basic_fstream;
```

See Also

Reference

Thread Safety in the Standard C++ Library

iostream Programming

iostreams Conventions

Other Resources

Standard C++ Library Header Files

<iostream>

Declares objects that control reading from and writing to the standard streams. This is often the only header you need to include to perform input and output from a C++ program.

For a list of all members of this header, see <iostream> Members.

```
#include <iostream>
```

Remarks

The objects fall into two groups:

- cin, cout, cerr, and clog are byte oriented, performing conventional byte-at-a-time transfers.
- wcin, wcout, wcerr, and wclog are wide oriented, translating to and from the wide characters that the program manipulates internally.

Once you perform certain operations on a stream, such as the standard input, you cannot perform operations of a different orientation on the same stream. Therefore, a program cannot operate interchangeably on both cin and wcin, for example.

All the objects declared in this header share a peculiar property — you can assume they are constructed before any static objects you define, in a translation unit that includes <iostream>. Equally, you can assume that these objects are not destroyed before the destructors for any such static objects you define. (The output streams are, however, flushed during program termination.) Therefore, you can safely read from or write to the standard streams before program startup and after program termination.

This guarantee is not universal, however. A static constructor may call a function in another translation unit. The called function cannot assume that the objects declared in this header have been constructed, given the uncertain order in which translation units participate in static construction. To use these objects in such a context, you must first construct an object of class ios_base::Init.

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

<iostream> Members
Standard C++ Library Header Files

<iostream> Members

Reference

Global Stream Objects

cerr	Specifies the **cerr** global stream.
cin	Specifies the **cin** global stream.
clog	Specifies the **clog** global stream.
cout	Specifies the **cout** global stream.
wcerr	Specifies the **wcerr** global stream.
wcin	Specifies the **wcin** global stream.
wclog	Specifies the **wclog** global stream.
wcout	Specifies the **wcout** global stream.

See Also

Reference

<iostream>
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Global Stream Objects

For information about the global stream objects in <iostream>, see <iostream> Members.

See Also

Reference

iostream Programming
iostreams Conventions

cerr

Specifies the **cerr** global stream.

```
extern ostream cerr;
```

Return Value

An ostream object.

Remarks

The object controls unbuffered insertions to the standard error output as a byte stream. Once the object is constructed, the expression **cerr.**flags **&** unitbuf is nonzero.

Example

```
// iostream_cerr.cpp
// compile with: /EHsc
// By default, cerr and clog are the same as cout
#include <iostream>
#include <fstream>

using namespace std;

void TestWide( )
{
   int i = 0;
   wcout << L"Enter a number: ";
   wcin >> i;
   wcerr << L"test for wcerr" << endl;
   wclog << L"test for wclog" << endl;
}

int main( )
{
   int i = 0;
   cout << "Enter a number: ";
   cin >> i;
   cerr << "test for cerr" << endl;
   clog << "test for clog" << endl;
   TestWide( );
}
```

Input

```
3
1
```

Sample Output

```
Enter a number: 3
test for cerr
test for clog
Enter a number: 1
test for wcerr
test for wclog
```

Requirements

Header: <iostream>

See Also

Reference

ostream

iostream Programming
iostreams Conventions
Other Resources
<iostream> Members

cin

Specifies the **cin** global stream.

```
extern istream cin;
```

Return Value

An istream object.

Remarks

The object controls extractions from the standard input as a byte stream. Once the object is constructed, the call **cin.**tie returns **&**cout.

Example

In this example, **cin** sets the fail bit on the stream when it encounters non-numeric characters. The program clears the fail bit and strips the invalid character from the stream to proceed.

```
// iostream_cin.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main()
{
   int x;
   cout << "enter choice:";
   cin >> x;
   while (x < 1 || x > 4)
   {
      cout << "Invalid choice, try again:";
      cin >> x;
      // not a numeric character, probably
      // clear the failure and pull off the non-numeric character
      if (cin.fail())
      {
         cin.clear();
         char c;
         cin >> c;
      }
   }
}
```

Input

2

Requirements

Header: <iostream>

See Also

Reference

istream

iostream Programming

iostreams Conventions

Other Resources

<iostream> Members

clog

Specifies the **clog** global stream.

```
extern ostream clog;
```

Return Value

An ostream object.

Remarks

The object controls buffered insertions to the standard error output as a byte stream.

Example

See cerr for an example of using **clog**.

Requirements

Header: <iostream>

See Also

Reference

ostream
iostream Programming
iostreams Conventions

Other Resources

<iostream> Members

cout

Specifies the **cout** global stream.

```
extern ostream cout;
```

Return Value

An ostream object.

Remarks

The object controls insertions to the standard output as a byte stream.

Example

See cerr for an example of using **cout**.

Requirements

Header: <iostream>

See Also

Reference

ostream
iostream Programming
iostreams Conventions

Other Resources

<iostream> Members

wcerr

Specifies the **wcerr** global stream.

```
extern wostream wcerr;
```

Return Value

A wostream object.

Remarks

The object controls unbuffered insertions to the standard error output as a wide stream. Once the object is constructed, the expression **wcerr.**flags **&** unitbuf is nonzero.

Example

See cerr for an example of using **wcerr**.

Requirements

Header: <iostream>

See Also

Reference

wostream

iostream Programming

iostreams Conventions

Other Resources

<iostream> Members

wcin

Specifies the **wcin** global stream.

```
extern wistream wcin;
```

Return Value

A wistream object.

Remarks

The object controls extractions from the standard input as a wide stream. Once the object is constructed, the call **wcin.**tie returns **&**wcout.

Example

See cerr for an example of using **wcin**.

Requirements

Header: <iostream>

See Also

Reference

wistream

iostream Programming

iostreams Conventions

Other Resources

<iostream> Members

wclog

Specifies the **wclog** global stream.

```
extern wostream wclog;
```

Return Value

A wostream object.

Remarks

The object controls buffered insertions to the standard error output as a wide stream.

Example

See cerr for an example of using **wclog**.

Requirements

Header: <iostream>

See Also

Reference

wostream
iostream Programming
iostreams Conventions

Other Resources

<iostream> Members

wcout

Specifies the **wcout** global stream.

```
extern wostream wcout;
```

Return Value

A wostream object.

Remarks

The object controls insertions to the standard output as a wide stream.

Example

See cerr for an example of using **wcout**.

Requirements

Header: <iostream>

See Also

Reference

wostream
iostream Programming
iostreams Conventions

Other Resources

<iostream> Members

<iso646.h>

Provides readable alternatives to certain operators or punctuators. The standard header <iso646.h> is available even in a freestanding implementation.

For a list of all members of this header, see <iso646.h> Members.

```
#include <iso646.h>
```

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<iso646.h> Members

Standard C++ Library Header Files

<iso646.h> Members

Reference

Macros

and	An alternative to the && operator.
and_eq	An alternative to the &= operator.
bitand	An alternative to the & operator.
bitor	An alternative to the | operator.
compl	An alternative to the ~ operator.
not	An alternative to the ! operator.
not_eq	An alternative to the != operator.
or	An alternative to the || operator.
or_eq	An alternative to the |= operator.
xor	An alternative to the ^ operator.
xor_eq	An alternative to the ^= operator.

See Also

Reference

<iso646.h>

Thread Safety in the Standard C++ Library

<iso646.h> Macros

For information about the macros in <iso646.h>, see <iso646.h> Members.

and

An alternative to the && operator.

```
#define and &&
```

Remarks

The macro yields the operator &&.

Example

```
// iso646_and.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   bool a = true, b = false, result;

   boolalpha(cout);

   result= a && b;
   cout << result << endl;

   result= a and b;
   cout << result << endl;
}
```

Output

```
false
false
```

Requirements

Header: <iso646.h>

See Also

Other Resources

<iso646.h> Members

and_eq

An alternative to the &= operator.

```
#define and_eq &=
```

Remarks

The macro yields the operator &=.

Example

```
// iso646_and_eq.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   int a = 3, b = 2, result;

   result= a &= b;
   cout << result << endl;

   result= a and_eq b;
   cout << result << endl;
}
```

Output

```
2
2
```

Requirements

Header: <iso646.h>

See Also

Other Resources

<iso646.h> Members

bitand

An alternative to the & operator.

```
#define bitand &
```

Remarks

The macro yields the operator

Example

```
// iso646_bitand.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   int a = 1, b = 2, result;


   result = a & b;
   cout << result << endl;

   result= a bitand b;
   cout << result << endl;
}
```

Output

```
0
0
```

Requirements

Header: <iso646.h>

See Also

Other Resources

<iso646.h> Members

bitor

An alternative to the | operator.

```
#define bitor |
```

Remarks

The macro yields the operator |.

Example

```
// iso646_bitor.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   int a = 1, b = 2, result;


   result = a | b;
   cout << result << endl;

   result= a bitor b;
   cout << result << endl;
}
```

Output

```
3
3
```

Requirements

Header: <iso646.h>

See Also

Other Resources

<iso646.h> Members

compl

An alternative to the ~ operator.

```
#define compl ~
```

Remarks

The macro yields the operator ~.

Example

```
// iso646_compl.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   int a = 1, result;

   result = ~a;
   cout << result << endl;

   result= compl(a);
   cout << result << endl;
}
```

Output

```
-2
-2
```

Requirements

Header: <iso646.h>

See Also

Other Resources

<iso646.h> Members

not

An alternative to the ! operator.

```
#define not !
```

Remarks

The macro yields the operator !.

Example

```
// iso646_not.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   int a = 0;

   if (!a)
      cout << "a is zero" << endl;

   if (not(a))
      cout << "a is zero" << endl;
}
```

Output

```
a is zero
a is zero
```

Requirements

Header: <iso646.h>

See Also

Other Resources

<iso646.h> Members

not_eq

An alternative to the != operator.

```
#define not_eq !=
```

Remarks

The macro yields the operator !=.

Example

```
// iso646_not_eq.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   int a = 0, b = 1;

   if (a != b)
      cout << "a is not equal to b" << endl;

   if (a not_eq b)
      cout << "a is not equal to b" << endl;
}
```

Output

```
a is not equal to b
a is not equal to b
```

Requirements

Header: <iso646.h>

See Also

Other Resources

<iso646.h> Members

or

An alternative to the || operator.

```
#define or ||
```

Remarks

The macro yields the operator ||.

Example

```
// iso646_or.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   bool a = true, b = false, result;

   boolalpha(cout);

   result= a || b;
   cout << result << endl;

   result= a or b;
   cout << result << endl;
}
```

Output

```
true
true
```

Requirements

Header: <iso646.h>

See Also

Other Resources

<iso646.h> Members

or_eq

An alternative to the |= operator.

```
#define or_eq |=
```

Remarks

The macro yields the operator |=.

Example

```
// iso646_oreq.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   int a = 3, b = 2, result;

   result= a |= b;
   cout << result << endl;

   result= a or_eq b;
   cout << result << endl;
}
```

Output

```
3
3
```

Requirements

Header: <iso646.h>

See Also

Other Resources

<iso646.h> Members

xor

An alternative to the ^ operator.

```
#define xor ^
```

Remarks

The macro yields the operator ^.

Example

```
// iso646_xor.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   int a = 3, b = 2, result;

   result= a ^ b;
   cout << result << endl;

   result= a xor_eq b;
   cout << result << endl;
}
```

Output

```
1
1
```

Requirements

Header: <iso646.h>

See Also

Other Resources

<iso646.h> Members

xor_eq

An alternative to the ^= operator.

```
#define xor_eq ^=
```

Remarks

The macro yields the operator ^=.

Example

```
// iso646_xor_eq.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   int a = 3, b = 2, result;

   result= a ^= b;
   cout << result << endl;

   a = 3;
   b = 2;

   result= a xor_eq b;
   cout << result << endl;
}
```

Output

```
1
1
```

Requirements

Header: <iso646.h>

See Also

Other Resources

<iso646.h> Members

<istream>

Defines the template class basic_istream, which mediates extractions for the iostreams, and the template class basic_iostream, which mediates both insertions and extractions. The header also defines a related manipulator. This header file is typically included for you by another iostreams header; you rarely have to include it directly.

For a list of all members of this header, see <istream> Members.

```
#include <istream>
```

See Also

Reference

Thread Safety in the Standard C++ Library

iostream Programming

iostreams Conventions

Other Resources

<istream> Members

<istream> Members

Reference

Typedefs

iostream	A type **basic_iostream** specialized on **char**.
istream	A type **basic_istream** specialized on **char**.
wiostream	A type **basic_iostream** specialized on **wchar**.
wistream	A type **basic_istream** specialized on **wchar**.

Manipulators

ws	Skips white space in the stream.

Operators

operator>>	Extracts characters and strings from the stream.

Classes

basic_iostream	A stream class that can do both input and output.
basic_istream	The template class describes an object that controls extraction of elements and encoded objects from a stream buffer with elements of type **Elem**, also known as char_type, whose character traits are determined by the class **Tr**, also known as traits_type.

See Also

Reference

<istream>
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

<istream> Typedefs

For information about the typedefs in <istream>, see <istream> Members.

See Also

Reference

iostream Programming

iostreams Conventions

iostream

A type **basic_iostream** specialized on **char**.

```
typedef basic_iostream<char, char_traits<char> > iostream;
```

Remarks

The type is a synonym for template class basic_iostream, specialized for elements of type **char** with default character traits.

Requirements

Header: <istream>

See Also

Reference

basic_iostream Class
iostream Programming
iostreams Conventions

Other Resources

<istream> Members

istream

A type **basic_istream** specialized on **char**.

```
typedef basic_istream<char, char_traits<char> > istream;
```

Remarks

The type is a synonym for template class basic_istream, specialized for elements of type **char** with default character traits.

Requirements

Header: <istream>

See Also

Reference

basic_istream Class
iostream Programming
iostreams Conventions

Other Resources

<istream> Members

wiostream

A type **basic_iostream** specialized on **wchar_t**.

```
typedef basic_iostream<wchar_t, char_traits<wchar_t> > wiostream;
```

Remarks

The type is a synonym for template class basic_iostream, specialized for elements of type **wchar_t** with default character traits.

Requirements

Header: <istream>

See Also

Reference

basic_iostream Class
iostream Programming
iostreams Conventions

Other Resources

<istream> Members

wistream

A type **basic_istream** specialized on **wchar_t**.

```
typedef basic_istream<wchar_t, char_traits<wchar_t> > wistream;
```

Remarks

The type is a synonym for template class basic_istream, specialized for elements of type **wchar_t** with default character traits.

Requirements

Header: <istream>

See Also

Reference

basic_istream Class
iostream Programming
iostreams Conventions

Other Resources

<istream> Members

<istream> Manipulators

For information about the manipulators in <istream>, see <istream> Members.

See Also

Reference

iostream Programming
iostreams Conventions

basic_iostream Class

A stream class that can do both input and output.

```
template <class Elem, class Tr = char_traits<Elem> >
   class basic_iostream : public basic_istream<Elem, Tr>,
      public basic_ostream<Elem, Tr> {
public:
   explicit basic_iostream(basic_streambuf<Elem, Tr>& *_Strbuf);
   virtual ~basic_iostream( );
   };
```

Remarks

The template class describes an object that controls insertions, through its base class basic_ostream<*Elem*, *Tr*>, and extractions, through its base class basic_istream<*Elem*, *Tr*>. The two objects share a common virtual base class basic_ios<*Elem*, *Tr*>. They also manage a common stream buffer, with elements of type *Elem*, whose character traits are determined by the class *Tr*. The constructor initializes its base classes through **basic_istream**(**strbuf**) and **basic_ostream**(**strbuf**).

Requirements

Header: <istream>

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

<istream> Members

ws

Skips white space in the stream.

```
template class<Elem, Tr>
   basic_istream<Elem, Tr>& ws(
   basic_istream<Elem, Tr>& _Istr
);
```

Parameters

_Istr

A stream.

Return Value

The stream.

Remarks

The manipulator extracts and discards any elements `ch` for which use_facet< **ctype**<**Elem**> >(getloc). **is**(**ctype**<**Elem**>::**space**, **ch**) is true.

The function calls setstate(**eofbit**) if it encounters end of file while extracting elements. It returns *_Istr*.

Example

See operator>> for an example of using **ws**.

Requirements

Header: <istream>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<istream> Members

<istream> Operators

For information about the operators in <istream>, see <istream> Members.

See Also

Reference

iostream Programming

iostreams Conventions

operator>> (<istream>)

Extracts characters and strings from the stream.

```
template<class Elem, class Tr>
   basic_istream<Elem, Tr>& operator>>(
      basic_istream<Elem, Tr>& _Istr,
      Elem *_Str
      );
template<class Elem, class Tr>
   basic_istream<Elem, Tr>& operator>>(
      basic_istream<Elem, Tr>& _Istr,
      Elem& _Ch
      );
template<class Tr>
   basic_istream<char, Tr>& operator>>(
      basic_istream<char, Tr>& _Istr,
      signed char *_Str
);
template<class Tr>
   basic_istream<char, Tr>& operator>>(
      basic_istream<char, Tr>& _Istr,
      signed char& _Ch
);
template<class Tr>
   basic_istream<char, Tr>& operator>>(
      basic_istream<char, Tr>& _Istr,
      unsigned char *_Str
);
template<class Tr>
   basic_istream<char, Tr>& operator>>(
      basic_istream<char, Tr>& _Istr,
      unsigned char& _Ch
);
```

Parameters

_Ch

A character.

_Istr

A stream.

_Str

A string.

Return Value

The stream

Remarks

The **basic_istream** class also defines several extraction operators. For more information, see basic_istream::operator>>.

The template function:

```
template<class Elem, class Tr>
   basic_istream<Elem, Tr>& operator>>(
      basic_istream<Elem, Tr>& _Istr, Elem *_str);
```

extracts up to *N* - 1 elements and stores them in the array beginning at *_Str*. If *_Istr*.width is greater than zero, *N* is *_Istr*.**width**; otherwise, it is the size of the largest array of **Elem** that can be declared. The function always stores **Elem** after any extracted elements it stores. Extraction stops early on end of file, on a character with value **Elem**(0) (which is not extracted), or on any

element (which is not extracted) that would be discarded by ws. If the function extracts no elements, it calls _*Istr*.setstate(**failbit**). In any case, it calls _*Istr*.**width**(0) and returns _*Istr*.

Security Note Use a null-terminated string. The null-terminated string must not exceed the size of the destination buffer. For more information, see Avoiding Buffer Overruns.

The template function:

```
template<class Elem, class Tr>
   basic_istream<Elem, Tr>& operator>>(
      basic_istream<Elem, Tr>& _Istr, char& _Ch);
```

extracts an element, if possible, and stores it in _*Ch*. Otherwise, it calls **is**.setstate(**failbit**). In any case, it returns _*Istr*.

The template function:

```
template<class Tr>
   basic_istream<char, Tr>& operator>>(
      basic_istream<char, Tr>& _Istr, signed char *_Str);
```

returns _*Istr* >> (**char** *)_*Str*.

The template function:

```
template<class Tr>
   basic_istream<char, Tr>& operator>>(
      basic_istream<char, Tr>& _Istr, signed char& _Ch);
```

returns _*Istr* >> (**char&**)_*Ch*.

The template function:

```
template<class Tr>
   basic_istream<char, Tr>& operator>>(
      basic_istream<char, Tr>& _Istr, unsigned char *_Str);
```

returns _*Istr* >> (**char** *)_*Str*.

The template function:

```
template<class Tr>
   basic_istream<char, Tr>& operator>>(
      basic_istream<char, Tr>& _Istr, unsigned char& _Ch);
```

returns _*Istr* >> (**char&**)_*Ch*.

Example

```
// istream_op_extract.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
   ws( cin );
   char c[10];

   cin.width( 9 );
   cin >> c;
   cout << c << endl;
}
```

Input

```
1234567890
```

Output

```
12345678
```

Requirements

Header: <istream>

See Also

Reference

basic_istream::operator>>

iostream Programming

iostreams Conventions

Other Resources

<istream> Members

<istream> Classes

For information about the classes in <istream>, see <istream> Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_istream Class

Describes an object that controls extraction of elements and encoded objects from a stream buffer with elements of type *Elem*, also known as char_type, whose character traits are determined by the class *Tr*, also known as traits_type.

For a list of all members of this type, see basic_istream Members.

```
template <class Elem, class Tr = char_traits<Elem> >
   class basic_istream
      : virtual public basic_ios<Elem, Tr>
```

Remarks

Most of the member functions that overload operator>> are formatted input functions. They follow the pattern:

```
     iostate state = goodbit;
     const sentry ok(*this);
     if (ok)
         {try
             {<extract elements and convert
             accumulate flags in state
             store a successful conversion> }
         catch (...)
             {try
                 {setstate(badbit); }
             catch (...)
                 {}
             if ((exceptions( ) & badbit) != 0)
                 throw; }}
     setstate(state);
     return (*this);
```

Many other member functions are unformatted input functions. They follow the pattern:

```
     iostate state = goodbit;
     count = 0;    // the value returned by gcount
     const sentry ok(*this, true);
     if (ok)
         {try
             {<extract elements and deliver
             count extracted elements in count
             accumulate flags in state> }
         catch (...)
             {try
                 {setstate(badbit); }
             catch (...)
                 {}
             if ((exceptions( ) & badbit) != 0)
                 throw; }}
     setstate(state);
```

Both groups of functions call setstate(**eofbit**) if they encounter end of file while extracting elements.

An object of class **basic_istream**<*Elem*, *Tr*> stores:

- A virtual public base object of class basic_ios<*Elem*, *Tr*> .
- An extraction count for the last unformatted input operation (called **count** in the previous code).

Example

See the example for basic_ifstream Class to learn more about input streams.

Requirements

Header: <istream>

See Also

Reference

Thread Safety in the Standard C++ Library

iostream Programming

iostreams Conventions

Other Resources

basic_istream Members

<istream> Members

basic_istream Members

The following tables list the members of the basic_istream Class. For a list of members inherited by the **basic_istream** class, see basic_ios Members.

Reference

Constructors

basic_istream	Constructs an object of type **basic_istream**.

Member Functions

gcount	Returns the number of characters read during the last unformatted input.
get	Reads one or more characters from the input stream.
getline	Reads a line from the input stream.
ignore	Causes a number of elements to be skipped from the current read position.
peek	Returns the next character to be read.
putback	Puts a specified character into the stream.
read	Reads a specified number of characters from the stream and stores them in an array. Deprecated. Use basic_istream::_Read_s instead.
_Read_s	Reads a specified number of characters from the stream and stores them in an array.
readsome	Read from buffer only. Deprecated. Use basic_istream::_Readsome_s instead.
_Readsome_s	Read from buffer only.
seekg	Moves the read position in a stream.
sentry	The nested class describes an object whose declaration structures the formatted input functions and the unform atted input functions.
sync	Synchronizes the input device associated with the stream with the stream's buffer.
tellg	Reports the current read position in the stream.
unget	Puts the most recently read character back into the stream.

Operators

operator>>	Calls a function on the input stream or reads formatted data from the input stream.

See Also

Reference

basic_istream Class
basic_ios Class
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

basic_ios Members

basic_istream Member Functions

For information about the functions in the basic_stream class, see basic_istream Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_istream::basic_istream

Constructs an object of type **basic_istream**.

```
explicit basic_istream(
    basic_streambuf<Elem, Tr> *_Strbuf,
    bool _Isstd = false
);
```

Parameters

_Strbuf

An object of type basic_streambuf.

_Isstd

true if this is a standard stream; otherwise, **false**.

Remarks

The constructor initializes the base class by calling init(_*Strbuf*). It also stores zero in the extraction count. For more information on this extraction count, see the Remarks section of the basic_istream Class overview topic.

Example

See the example for basic_ifstream::basic_ifstream to learn more about input streams.

Requirements

Header: <istream>

See Also

Reference

basic_istream Class
iostream Programming
iostreams Conventions

Other Resources

basic_istream Members

basic_istream::gcount

Returns the number of characters read during the last unformatted input.

```
streamsize gcount( ) const;
```

Return Value

The extraction count.

Remarks

Use basic_istream::get to read unformatted characters.

Example

```
// basic_istream_gcount.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
   cout << "Type the letter 'a': ";

   ws( cin );
   char c[10];

   cin.get( &c[0],9 );
   cout << c << endl;

   cout << cin.gcount( ) << endl;
}
```

Input

```
a
```

Sample Output

```
Type the letter 'a':
a
1
```

Requirements

Header: <istream>

See Also

Reference

basic_istream Class
iostream Programming
iostreams Conventions

Other Resources

basic_istream Members

basic_istream::get

Reads one or more characters from the input stream.

```
int_type get( );
basic_istream& get(
   char_type& _Ch
);
basic_istream& get(char_type *_Str,
   streamsize _Count);
basic_istream& get(
   char_type *_Str,
   streamsize _Count,
   char_type _Delim
);
basic_istream& get(
   basic_streambuf<Elem, Tr> *_Strbuf
);
basic_istream& get(
   basic_streambuf<Elem, Tr> *_Strbuf,
   char_type _Delim
);
```

Parameters

_Count

The number of characters to read from **strbuf**.

_Delim

The character that should terminate the read if it is encountered before *_Count*.

_Str

A string in which to write.

_Strbuf

A buffer in which to write.

Return Value

The parameterless form of get returns the element read as an integer or end of file. The remaining forms return the stream (***this**).

Remarks

The first of these unformatted input functions extracts an element, if possible, as if by returning **rdbuf**->**sbumpc**. Otherwise, it returns **traits_type::**eof. If the function extracts no element, it calls setstate(**failbit**).

The second function extracts the int_type element **meta** the same way. If **meta** compares equal to **traits_type::eof**, the function calls **setstate**(**failbit**). Otherwise, it stores **traits_type::**to_char_type(**meta**) in *_Ch*. The function returns ***this**.

The third function returns **get**(*_Str*, *_Count*, **widen**('**n**')).

The fourth function extracts up to *_Count* - 1 elements and stores them in the array beginning at *_Str*. It always stores **char_type** after any extracted elements it stores. In order of testing, extraction stops:

- At end of file.
- After the function extracts an element that compares equal to *_Delim*, in which case the element is put back to the controlled sequence.
- After the function extracts *_Count* - 1 elements.

If the function extracts no elements, it calls **setstate**(**failbit**). In any case, it returns ***this**.

The fifth function returns **get**(**strbuf**, **widen**('**n**')).

The sixth function extracts elements and inserts them in **strbuf**. Extraction stops on end-of-file or on an element that compares equal to *_Delim*, which is not extracted. It also stops, without extracting the element in question, if an insertion fails or throws an exception (which is caught but not rethrown). If the function extracts no elements, it calls **setstate**(**failbit**). In any case, the function returns ***this**.

Example

```
// basic_istream_get.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
   char c[10];

   c[0] = cin.get( );
   cin.get( c[1] );
   cin.get( &c[2],3 );
   cin.get( &c[4], 4, '7' );

   cout << c << endl;
}
```

Input

```
11
```

Output

```
11
```

Requirements

Header: <istream>

See Also

Reference

basic_istream Class

iostream Programming

iostreams Conventions

Other Resources

basic_istream Members

basic_istream::getline

Gets a line from the input stream.

```
basic_istream& getline(
   char_type *_Str,
   streamsize _Count
);
basic_istream& getline(
   char_type *_Str,
   streamsize _Count,
   char_type _Delim
);
```

Parameters

_Count

The number of characters to read from **strbuf**.

_Delim

The character that should terminate the read if it is encountered before *_Count*.

_Str

A string in which to write.

Return Value

The stream (***this**).

Remarks

The first of these unformatted input functions returns **getline**(*_Str*, *_Count*, **widen**('**\n**')).

The second function extracts up to *_Count* - 1 elements and stores them in the array beginning at *_Str*. It always stores the string termination character after any extracted elements it stores. In order of testing, extraction stops:

- At end of file.
- After the function extracts an element that compares equal to *_Delim*, in which case the element is neither put back nor appended to the controlled sequence.
- After the function extracts *_Count* - 1 elements.

If the function extracts no elements or *_Count* - 1 elements, it calls setstate(**failbit**). In any case, it returns ***this**.

Example

```
// basic_istream_getline.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
   char c[10];

   cin.getline( &c[0], 5, '2' );
   cout << c << endl;
}
```

Input

```
12
```

Output

1

Requirements

Header: <istream>

See Also

Reference

basic_istream Class

iostream Programming

iostreams Conventions

Other Resources

basic_istream Members

basic_istream::ignore

Causes a number of elements to be skipped from the current read position.

```
basic_istream& ignore(
   streamsize _Count = 1,
   int_type _Delim = traits_type::eof( )
);
```

Parameters

_Count

The number of elements to skip from the current read position.

_Delim

The element that, if encountered before count, causes **ignore** to return and allowing all elements after *_Delim* to be read.

Return Value

The stream (***this**).

Remarks

The unformatted input function extracts up to *_Count* elements and discards them. If *_Count* equals **numeric_limits<int>::max**, however, it is taken as arbitrarily large. Extraction stops early on end of file or on an element *_Ch* such that **traits_type::**to_int_type(*_Ch*) compares equal to *_Delim* (which is also extracted). The function returns ***this**.

Example

```
// basic_istream_ignore.cpp
// compile with: /EHsc
#include <iostream>
int main( )
{
   using namespace std;
   char chararray[10];
   cout << "Type 'abcdef': ";
   cin.ignore( 5, 'c' );
   cin >> chararray;
   cout << chararray;
}
```

Input

```
abcdef
```

Sample Output

```
def
```

Requirements

Header: <istream>

See Also

Reference

basic_istream Class

iostream Programming

iostreams Conventions

Other Resources

basic_istream Members

basic_istream::peek

Returns the next character to be read.

```
int_type peek( );
```

Return Value

The next character that will be read.

Remarks

The unformatted input function extracts an element, if possible, as if by returning **rdbuf** ->sgetc. Otherwise, it returns **traits_type::**eof.

Example

```
// basic_istream_peek.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
   char c[10], c2;
   cout << "Type 'abcde': ";

   c2 = cin.peek( );
   cin.getline( &c[0], 9 );

   cout << c2 << " " << c << endl;
}
```

Input

```
abcde
```

Sample Output

```
Type 'abcde': abcde
a abcde
```

Requirements

Header: <istream>

See Also

Reference

basic_istream Class
iostream Programming
iostreams Conventions

Other Resources

basic_istream Members

basic_istream::putback

Puts a specified character into the stream.

```
basic_istream& putback(
   char_type _Ch
);
```

Parameters

_Ch

A character to put back into the stream.

Return Value

The stream (***this**).

Remarks

The unformatted input function puts back *_Ch*, if possible, as if by calling rdbuf->sputbackc. If rdbuf is a null pointer, or if the call to **sputbackc** returns **traits_type::**eof, the function calls setstate(**badbit**). In any case, it returns ***this**.

Example

```
// basic_istream_putback.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
   char c[10], c2, c3;

   c2 = cin.get( );
   c3 = cin.get( );
   cin.putback( c2 );
   cin.getline( &c[0], 9 );
   cout << c << endl;
}
```

Input

```
qw
```

Output

```
q
```

Requirements

Header: <istream>

See Also

Reference

basic_istream Class

iostream Programming

iostreams Conventions

Other Resources

basic_istream Members

basic_istream::read

Reads a specified number of characters from the stream and stores them in an array.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct. Consider using basic_istream::_Read_s instead.

```
basic_istream& read(
   char_type *_Str,
   streamsize _Count
);
```

Parameters

_Str

The array in which to read the characters.

_Count

The number of characters to read.

Return Value

The stream (***this**).

Remarks

The unformatted input function extracts up to `count` elements and stores them in the array beginning at *_Str*. Extraction stops early on end of file, in which case the function calls setstate(**failbit**). In any case, it returns ***this**.

Example

```
// basic_istream_read.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main()
{
    char c[10];
    int count = 5;

    cout << "Type 'abcde': ";

    // Note: cin::read is potentially unsafe, consider
    // using cin::_Read_s instead.
    cin.read(&c[0], count);
    c[count] = 0;

    cout << c << endl;
}
```

Input

```
abcde
```

Sample Output

```
Type 'abcde': abcde
abcde
```

Requirements

Header: <istream>

See Also

Reference

basic_istream Class
iostream Programming
iostreams Conventions

Other Resources

basic_istream Members

basic_istream::_Read_s

Reads a specified number of characters from the stream and stores them in an array.

```
basic_istream& _Read_s(
    char_type *_Str,
    size_t _Str_size,
    streamsize _Count
);
```

Parameters

_Str

The array in which to read the characters.

_Str_size

The size of *_Str*.

_Count

The number of characters to read.

Return Value

The stream (***this**).

Remarks

The unformatted input function extracts up to `count` elements and stores them in the array beginning at *_Str*. Extraction stops early on end of file, in which case the function calls setstate(**failbit**). In any case, it returns ***this**.

Example

```
// basic_istream__Read_s.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main()
{
    const int c_size = 10;
    char c[c_size];
    int count = 5;

    cout << "Type 'abcde': ";

    cin._Read_s(&c[0], c_size, count);
    c[count] = 0;

    cout << c << endl;
}
```

Input

```
abcde
```

Sample Output

```
Type 'abcde': abcde
abcde
```

Requirements

Header: <istream>

See Also

Reference

basic_istream Class

iostream Programming

iostreams Conventions

Safe Libraries: Standard C++ Library

Other Resources

basic_istream Members

basic_istream::readsome

Read from buffer only.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct. Consider using basic_istream::_Readsome_s instead.

```
streamsize readsome(
   char_type *_Str,
   streamsize _Count
);
```

Parameters

_Str

The array in which to read the characters.

_Count

The number of characters to read.

Return Value

The count of items in the buffer.

Remarks

The unformatted input function extracts up to `count` elements and stores them in the array beginning at *_Str*. If good is false, the function calls setstate(**failbit**). Otherwise, it assigns the value of **rdbuf**->in_avail to *N*. If *N* < 0, the function calls **setstate**(**eofbit**). Otherwise, it replaces the value stored in *N* with the smaller of *_Count* and *N*, and then calls read(*_Str*, *N*). In any case, the function returns gcount.

Example

```
// basic_istream_readsome.cpp
// compile with: /EHsc /W3
#include <iostream>
using namespace std;

int main( )
{
   char c[10];
   int count = 5;

   cout << "Type 'abcdefgh': ";

   // Can read from buffer or console
   // Note: cin::read is potentially unsafe, consider
   // using cin::_Read_s instead.
   cin.read(&c[0], 2);

   // Can only read from buffer, not from console
   // Note: cin::readsome is potentially unsafe, consider
   // using cin::_Readsome_s instead.
   cin.readsome(&c[0], count);  // C4996
   c[count] = 0;
   cout << c << endl;
}
```

Input

```
abcdefgh
```

Sample Output

```
Type 'abcdefgh': abcdefgh
```

```
cdefg
```

Requirements

Header: <istream>

See Also

Reference

basic_istream Class

iostream Programming

iostreams Conventions

Other Resources

basic_istream Members

basic_istream::_Readsome_s

Read from buffer only.

```
streamsize _Readsome_s(
    char_type *_Str,
    size_t _Str_size,
    streamsize _Count
);
```

Parameters

_Str

The array in which to read the characters.

_Str_size

The size of *_Str*.

_Count

The number of characters to read.

Return Value

The count of items in the buffer.

Remarks

The unformatted input function extracts up to `count` elements and stores them in the array beginning at *_Str*. If good is false, the function calls setstate(**failbit**). Otherwise, it assigns the value of **rdbuf**->in_avail to *N*. If *N* < 0, the function calls **setstate**(**eofbit**). Otherwise, it replaces the value stored in *N* with the smaller of *_Count* and *N*, and then calls read(*_Str*, *N*). In any case, the function returns gcount.

Example

```
// basic_istream__Readsome_s.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
    const int c_size = 10;
    char c[c_size];
    int count = 5;

    cout << "Type 'abcdefgh': ";

    // Can read from buffer or console
    cin._Read_s(&c[0], c_size, 2);

    // Can only read from buffer, not from console
    cin._Readsome_s(&c[0], c_size, count);
    c[count] = 0;
    cout << c << endl;
}
```

Input

```
abcdefgh
```

Sample Output

```
Type 'abcdefgh': abcdefgh
cdefg
```

Requirements

Header: <istream>

See Also

Reference

basic_istream Class

iostream Programming

iostreams Conventions

Safe Libraries: Standard C++ Library

Other Resources

basic_istream Members

basic_istream::seekg

Moves the read position in a stream.

```
basic_istream& seekg(
   pos_type _Pos
);
basic_istream& seekg(
   off_type _Off,
   ios_base::seekdir _Way
);
```

Parameters

_Off

An offset to move the read pointer relative to *way*.

_Pos

The absolute position in which to move the read pointer.

_Way

One of the ios_base::seekdir enumerations.

Return Value

The stream (***this**).

Remarks

If fail is false, the first member function calls **newpos** = rdbuf -> pubseekpos(*_Pos*), for some **pos_type** temporary object **newpos**. If **fail** is false, the second function calls **newpos** = **rdbuf** -> pubseekoff(*_Off*, *_Way*). In either case, if (**off_type**)**newpos** == (**off_type**)(-1) (the positioning operation fails), the function calls **istr**.setstate(**failbit**). Both functions return ***this**.

Example

```
// basic_istream_seekg.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main ( )
{
   using namespace std;
   ifstream file;
   char c, c1;

   file.open( "basic_istream_seekg.txt" );
   file.seekg(2);   // chars to skip
   file >> c;
   cout << c << endl;

   file.seekg( 0, ios_base::beg );
   file >> c;
   cout << c << endl;

   file.seekg( -1, ios_base::end );
   file >> c1;
   cout << c1 << endl;
}
```

Input: basic_istream_seekg.txt

```
0123456789
```

Output

```
2
0
9
```

Requirements

Header: <istream>

See Also

Reference

basic_istream Class

iostream Programming

iostreams Conventions

Other Resources

basic_istream Members

basic_istream::sentry

The nested class describes an object whose declaration structures the formatted and unformatted input functions.

```
class sentry {
public:
   explicit sentry(
      basic_istream& _Istr,
      bool _Noskip = false
   );
   operator bool( ) const;
};
```

Remarks

If _*Istr*.good is true, the constructor:

- Calls _*Istr*.tie -> flush if _*Istr*.**tie** is not a null pointer
- Effectively calls ws(_*Istr*) if _*Istr*.flags **&** skipws is nonzero

If, after any such preparation, _*Istr*.**good** is false, the constructor calls _*Istr*.setstate(**failbit**). In any case, the constructor stores the value returned by _*Istr*.**good** in **status**. A later call to **operator bool** delivers this stored value.

Requirements

Header: <istream>

See Also

Reference

basic_istream Class
iostream Programming
iostreams Conventions

Other Resources

basic_istream Members

basic_istream::sync

Synchronizes the input device associated with the stream with the stream's buffer.

```
int sync( );
```

Return Value

If rdbuf is a null pointer, the function returns -1. Otherwise, it calls **rdbuf** ->pubsync. If that returns -1, the function calls setstate(**badbit**) and returns -1. Otherwise, the function returns zero.

Requirements

Header: <istream>

See Also

Reference

basic_istream Class
iostream Programming
iostreams Conventions

Other Resources

basic_istream Members

basic_istream::tellg

Reports the current read position in the stream.

```
pos_type tellg( );
```

Return Value

The current position in the stream.

Remarks

If fail is false, the member function returns rdbuf -> pubseekoff(0, **cur**, **in**). Otherwise, it returns **pos_type**(-1).

Example

```
// basic_istream_tellg.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main()
{
    using namespace std;
    ifstream file;
    char c;
    streamoff i;

    file.open("basic_istream_tellg.txt");
    i = file.tellg();
    file >> c;
    cout << c << " " << i << endl;

    i = file.tellg();
    file >> c;
    cout << c << " " << i << endl;
}
```

Input: basic_istream_tellg.txt

```
0123456789
```

Program Output

```
0 0
1 1
```

Requirements

Header: <istream>

See Also

Reference

basic_istream Class

iostream Programming

iostreams Conventions

Other Resources

basic_istream Members

basic_istream::unget

Puts the most recently read character back into the stream.

```
basic_istream& unget( );
```

Return Value

The stream (***this**).

Remarks

The unformatted input function puts back the previous element in the stream, if possible, as if by calling **rdbuf** ->sungetc. If rdbuf is a null pointer, or if the call to **sungetc** returns **traits_type::**eof, the function calls setstate(**badbit**). In any case, it returns ***this**.

Example

```
// basic_istream_unget.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
   char c[10], c2;

   cout << "Type 'abc': ";
   c2 = cin.get( );
   cin.unget( );
   cin.getline( &c[0], 9 );
   cout << c << endl;
}
```

Input

```
abc
```

Sample Output

```
Type 'abc': abc
abc
```

Requirements

Header: <istream>

See Also

Reference

basic_istream Class

iostream Programming

iostreams Conventions

Other Resources

basic_istream Members

basic_istream Operators

For information about the functions in the basic_istream class, see basic_istream Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_istream::operator>>

Calls a function on the input stream or reads formatted data from the input stream.

```
basic_istream& operator>>(
   basic_istream& (*_Pfn)(basic_istream&)
);
basic_istream& operator>>(
   ios_base& (*_Pfn)(ios_base&)
);
basic_istream& operator>>(
   basic_ios<Elem, Tr>& (*_Pfn)(basic_ios<Elem, Tr>&))
;
basic_istream& operator>>(
   basic_streambuf<Elem, Tr> *_Strbuf
);
basic_istream& operator>>(
   bool& _Val
);
basic_istream& operator>>(
   short& _Val
);
basic_istream& operator>>(
   unsigned short& _Val
);
basic_istream& operator>>(
   int& _Val
);
basic_istream& operator>>(
   unsigned int& _Val
);
basic_istream& operator>>(
   long& _Val
);
basic_istream& operator>>(
   unsigned long& _Val
);
basic_istream& operator>>(
   void *& _Val
);
basic_istream& operator>>(
   float& _Val
);
basic_istream& operator>>(
   double& _Val
);
basic_istream& operator>>(
   long double& _Val
);
```

Parameters

_Pfn

A function pointer.

_Strbuf

An object of type **stream_buf**.

_Val

The value to read from the stream.

Return Value

The stream (***this**).

Remarks

The **<istream>** header also defines several global extraction operators. For more information, see operator>> (<istream>).

The first member function ensures that an expression of the form **istr** >> **ws** calls ws(**istr**), and then returns ***this**. The second and third functions ensure that other manipulators, such as hex, behave similarly. The remaining functions constitute the formatted input functions.

The function:

```
basic_istream& operator>>(
    basic_streambuf<Elem, Tr> *_Strbuf);
```

extracts elements, if *_Strbuf* is not a null pointer, and inserts them in *_Strbuf*. Extraction stops on end of file. It also stops without extracting the element in question, if an insertion fails or throws an exception (which is caught but not rethrown). If the function extracts no elements, it calls setstate(**failbit**). In any case, the function returns ***this**.

The function:

```
basic_istream& operator>>(bool& _Val);
```

extracts a field and converts it to a Boolean value by calling use_facet <**num_get**<**Elem**, **InIt**>(getloc). get(**InIt**(rdbuf), **Init**(0), ***this**, **getloc**, *_Val*). Here, **InIt** is defined as istreambuf_iterator<**Elem**, **Tr**>. The function returns ***this**.

The functions:

```
basic_istream& operator>>(short& _Val);
basic_istream& operator>>(unsigned short& _Val);
basic_istream& operator>>(int& _Val);
basic_istream& operator>>(unsigned int& _Val);
basic_istream& operator>>(long& _Val);
basic_istream& operator>>(unsigned long& _Val);
basic_istream& operator>>(void *& _Val);
```

each extract a field and convert it to a numeric value by calling **use_facet**<**num_get**<**Elem**, **InIt**>(**getloc**). get(**InIt**(**rdbuf**), **Init**(0), ***this**, **getloc**, *_Val*). Here, **InIt** is defined as **istreambuf_iterator**<**Elem**, **Tr**>, and *_Val* has type **long**, **unsigned long**, or **void *** as needed.

If the converted value cannot be represented as the type of *_Val*, the function calls setstate(**failbit**). In any case, the function returns ***this**.

The functions:

```
basic_istream& operator>>(float& _Val);
basic_istream& operator>>(double& _Val);
basic_istream& operator>>(long double& _Val);
```

each extract a field and convert it to a numeric value by calling **use_facet**<**num_get**<**Elem**, **InIt**>(**getloc**). **get**(**InIt**(**rdbuf**), **Init**(0), ***this**, **getloc**, *_Val*). Here, **InIt** is defined as **istreambuf_iterator**<**Elem**, **Tr**>, and *_Val* has type **double** or **long double** as needed.

If the converted value cannot be represented as the type of *_Val*, the function calls **setstate**(**failbit**). In any case, it returns ***this**.

Example

```
// istream_basic_istream_op_is.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

ios_base& hex2( ios_base& ib )
{
   ib.unsetf( ios_base::dec );
```

```
   ib.setf( ios_base::hex );
   return ib;
}

basic_istream<char, char_traits<char> >& somefunc(basic_istream<char, char_traits<char> > &
i)
{
   if ( i == cin )
   {
      cerr << "i is cin" << endl;
   }
   return i;
}

int main( )
{
   int i = 0;
   cin >> somefunc;
   cin >> i;
   cout << i << endl;
   cin >> hex2;
   cin >> i;
   cout << i << endl;
}
```

Input

```
10
10
```

Sample Output

```
i is cin
10
10
10
16
```

Requirements

Header: <istream>

See Also

Reference

basic_istream Class
operator>> (<istream>)
iostream Programming
iostreams Conventions

Other Resources

basic_istream Members

<iterator>

Defines the iterator primitives, predefined iterators and stream iterators, as well as several supporting templates. The predefined iterators include insert and reverse adaptors. There are three classes of insert iterator adaptors: front, back, and general. They provide insert semantics rather than the overwrite semantics that the container member function iterators provide.

For a list of all members of this header, see <iterator> Members.

```
#include <iterator>
```

Remarks

Iterators are a generalization of pointers, abstracting from their requirements in a way that allows a C++ program to work with different data structures in a uniform manner. Iterators act as intermediaries between containers and generic algorithms. Instead of operating on specific data types, algorithms are defined to operate on a range specified by a type of iterator. Any data structure that satisfies the requirements of the iterator may then be operated on by the algorithm. There are five types or categories of iterator, each with its own set of requirements and resulting functionality:

- Output: forward moving, may store but not retrieve values, provided by ostream and inserter.
- Input: forward moving, may retrieve but not store values, provided by istream.
- Forward: forward moving, may store and retrieve values.
- Bidirectional: forward and backward moving, may store and retrieve values, provided by list, set, multiset, map, and multimap.
- Random access: elements accessed in any order, may store and retrieve values, provided by vector, deque, string, and array.

Iterators that have greater requirements and so more powerful access to elements may be used in place of iterators with fewer requirements. For example, if a forward iterator is called for, then a random-access iterator may used instead.

See Also

Reference

Thread Safety in the Standard C++ Library
Standard Template Library

Other Resources

<iterator> Members
Standard C++ Library Header Files

<iterator> Members

Reference

Functions

advance	Increments an iterator by a specified number of positions.
back_inserter	Creates an iterator that can insert elements at the back of a specified container.
distance	Determines the number of increments between the positions addressed by two iterators.
front_inserter	Creates an iterator that can insert elements at the front of a specified container.
inserter	An iterator adaptor that adds a new element to a container at a specified point of insertion.

Operators

operator!=	Tests if the iterator object on the left side of the operator is not equal to the iterator object on the right side.
operator==	Tests if the iterator object on the left side of the operator is equal to the iterator object on the right side.
operator<	Tests if the iterator object on the left side of the operator is less than the iterator object on the right side.
operator<=	Tests if the iterator object on the left side of the operator is less than or equal to the iterator object on the right side.
operator>	Tests if the iterator object on the left side of the operator is greater than the iterator object on the right side.
operator>=	Tests if the iterator object on the left side of the operator is greater than or equal to the iterator object on the right side.
operator+	Adds an offset to an iterator and returns the new **reverse_iterator** addressing the inserted element at the new offset position.
operator-	Subtracts one iterator from another and returns the difference.

Classes

back_insert_iterator	The template class describes an output iterator object. It inserts elements into a container of type **Container**, which it accesses through the protected **pointer** object it stores called container.
bidirectional_iterator_tag	A class that provides a return type for an **iterator_category** function that represents a bidirectional iterator.
checked_array_iterator	A class that accesses an array using a random access, checked iterator.
checked_iterator	A class that transforms iterators into checked iterators.
front_insert_iterator	The template class describes an output iterator object. It inserts elements into a container of type **Container**, which it accesses through the protected **pointer** object it stores called container.
forward_iterator_tag	A class that provides a return type for an **iterator_category** function that represents a forward iterator.
input_iterator_tag	A class that provides a return type for an **iterator_category** function that represents a bidirectional iterator.

insert_iterator	The template class describes an output iterator object. It inserts elements into a container of type **Container**, which it accesses through the protected **pointer** object it stores called container. It als o stores the protected **iterator** object, of class **Container::iterator**, called **iter**.
istream_iterator	The template class describes an input iterator object. It extracts objects of class **Ty** from an input st ream, which it accesses through an object it stores, of type pointer to **basic_istream**<**Elem**, **Tr**>.
istreambuf_iterator	The template class describes an input iterator object. It inserts elements of class **Elem** into an outp ut stream buffer, which it accesses through an object it stores, of type **pointer** to **basic_streamb uf**<**Elem**, **Tr**>.
iterator	The template class is used as a base type for all iterators.
iterator_traits	A template helper class providing critical types that are associated with different iterator types so t hat they can be referred to in the same way.
ostream_iterator	The template class describes an output iterator object. It inserts objects of class **Type** into an outp ut stream, which it accesses through an object it stores, of type **pointer** to **basic_ostream**<**Elem**, **Tr**>.
ostreambuf_iterator Class	The template class describes an output iterator object. It inserts elements of class **Elem** into an ou tput stream buffer, which it accesses through an object it stores, of type pointer to **basic_streamb uf**<**Elem**, **Tr**>.
output_iterator_tag	A class that provides a return type for **iterator_category** function that represents an output iterat or.
random_access_iterator_tag	A class that provides a return type for **iterator_category** function that represents a random-acce ss iterator.
reverse_iterator	The template class describes an object that behaves like a random-access iterator, only in reverse.

See Also

Reference

<iterator>

Thread Safety in the Standard C++ Library

Standard Template Library

<iterator> Functions

For information about the functions in <iterator>, see <iterator> Members.

See Also

Reference

Standard Template Library

advance

Increments an iterator by a specified number of positions.

```
template<class InputIterator, class Distance>
   void advance(
      InputIterator& _InIt,
      Distance _Off
   );
```

Parameters

_InIt

The iterator that is to be incremented and that must satisfy the requirements for an input iterator.

_Off

An integral type that is convertible to the iterator's difference type and that specifies the number of increments the position of the iterator is to be advanced.

Remarks

The range advanced through must be nonsingular, where the iterators must be dereferenceable or past the end.

If the **InputIterator** satisfies the requirements for a bidirectional iterator type, then *_Off* may be negative. If **InputIterator** is an input or forward iterator type, *_Off* must be nonnegative.

The advance function has constant complexity when **InputIterator** satisfies the requirements for a random-access iterator; otherwise, it has linear complexity and so is potentially expensive.

Example

```
// iterator_advance.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   list<int> L;
   for ( i = 1 ; i < 9 ; ++i )
   {
      L.push_back ( i );
   }
   list <int>::iterator L_Iter, LPOS = L.begin ( );

   cout << "The list L is: ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
      cout << *L_Iter << " ";
   cout << ")." << endl;

   cout << "The iterator LPOS initially points to the first element: "
        << *LPOS << "." << endl;

   advance ( LPOS , 4 );
   cout << "LPOS is advanced 4 steps forward to point"
        << " to the fifth element: "
        << *LPOS << "." << endl;

   advance ( LPOS , -3 );
   cout << "LPOS is moved 3 steps back to point to the "
        << "2nd element: " << *LPOS << "." << endl;
```

```
}
```

Output

```
The list L is: ( 1 2 3 4 5 6 7 8 ).
The iterator LPOS initially points to the first element: 1.
LPOS is advanced 4 steps forward to point to the fifth element: 5.
LPOS is moved 3 steps back to point to the 2nd element: 2.
```

Requirements

Header: <iterator>

See Also

Reference

advance (STL Samples)

Standard Template Library

Other Resources

<iterator> Members

back_inserter

Creates an iterator that can insert elements at the back of a specified container.

```
template<class Container>
   back_insert_iterator<Container> back_inserter(
      Container& _Cont
      );
```

Parameters

_Cont

The container into which the back insertion is to be executed.

Return Value

A **back_insert_iterator** associated with the container object *_Cont*.**Remarks**

Within the Standard Template Library, the argument must refer to one of the three sequence containers that have the member function **push_back**: deque Class, list Class, or vector Class.

Example

```
// iterator_back_inserter.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for ( i = 0 ; i < 3 ; ++i )
   {
      vec.push_back ( i );
   }

   vector <int>::iterator vIter;
   cout << "The initial vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   // Insertions can be done with template function
   back_insert_iterator<vector<int> > backiter ( vec );
   *backiter = 30;
   backiter++;
   *backiter = 40;

   // Alternatively, insertions can be done with the
   // back_insert_iterator member function
   back_inserter ( vec ) = 500;
   back_inserter ( vec ) = 600;

   cout << "After the insertions, the vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;
}
```

Output

```
The initial vector vec is: ( 0 1 2 ).
After the insertions, the vector vec is: ( 0 1 2 30 40 500 600 ).
```

Requirements

Header: <iterator>

See Also

Reference

Standard Template Library

Other Resources

<iterator> Members

distance

Determines the number of increments between the positions addressed by two iterators.

```
template<class InputIterator>
   typename iterator_traits<InputIterator>::difference_type
      distance(
         InputIterator _First,
         InputIterator _Last
            );
```

Parameters

_First

The first iterator whose distance from the second is to be determined.

_Last

The second iterator whose distance from the first is to be determined.

Return Value

The number of times that *_First* must be incremented until it equal *_Last*.

Remarks

The distance function has constant complexity when **InputIterator** satisfies the requirements for a random-access iterator; otherwise, it has linear complexity and so is potentially expensive.

Example

```
// iterator_distance.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   list<int> L;
   for ( i = -1 ; i < 9 ; ++i )
   {
      L.push_back ( 2 * i );
   }
   list <int>::iterator L_Iter, LPOS = L.begin ( );

   cout << "The list L is: ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++ )
      cout << *L_Iter << " ";
   cout << ")." << endl;

   cout << "The iterator LPOS initially points to the first element: "
        << *LPOS << "." << endl;

   advance ( LPOS , 7 );
   cout << "LPOS is advanced 7 steps forward to point "
        << " to the eighth element: "
        << *LPOS << "." << endl;

   list<int>::difference_type Ldiff ;
   Ldiff = distance ( L.begin ( ) , LPOS );
   cout << "The distance from L.begin( ) to LPOS is: "
        << Ldiff << "." << endl;
```

```
}
```

Output

```
The list L is: ( -2 0 2 4 6 8 10 12 14 16 ).
The iterator LPOS initially points to the first element: -2.
LPOS is advanced 7 steps forward to point  to the eighth element: 12.
The distance from L.begin( ) to LPOS is: 7.
```

Requirements

Header: <iterator>

See Also

Reference

distance (STL Samples)

Standard Template Library

Other Resources

<iterator> Members

front_inserter

Creates an iterator that can insert elements at the front of a specified container.

```
template<class Container>
   front_insert_iterator<Container> front_inserter(
      Container& _Cont
   );
```

Parameters

_Cont

The container object whose front is having an element inserted.

Return Value

A **front_insert_iterator** associated with the container object *_Cont*.

Remarks

The member function front_insert_iterator of the front_insert_iterator class may also be used.

Within the Standard Template Library, the argument must refer to one of the two sequence containers that have the member function **push_back**: deque Class or list Class.

Example

```
// iterator_front_inserter.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
   list <int>::iterator L_Iter;

   list<int> L;
   for ( i = -1 ; i < 9 ; ++i )
   {
      L.push_back ( i );
   }

   cout << "The list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
      cout << *L_Iter << " ";
   cout << ")." << endl;

   // Using the template function to insert an element
   front_insert_iterator< list < int> > Iter(L);
   *Iter = 100;

   // Alternatively, you may use the front_insert member function
   front_inserter ( L ) = 200;

   cout << "After the front insertions, the list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
      cout << *L_Iter << " ";
   cout << ")." << endl;
}
```

Output

```
The list L is:
 ( -1 0 1 2 3 4 5 6 7 8 ).
After the front insertions, the list L is:
 ( 200 100 -1 0 1 2 3 4 5 6 7 8 ).
```

Requirements

Header: <iterator>

See Also

Reference

Standard Template Library

Other Resources

<iterator> Members

inserter

An iterator adaptor that adds a new element to a container at a specified point of insertion.

```
template<class Container, class Iterator>
   insert_iterator<Container> inserter(
      Container& _Cont,
      Iterator _It
      );
```

Parameters

_Cont

The container to which new elements are to be added.

_It

An iterator locating the point of insertion.

Return Value

An insert iterator addressing the new element inserted.

Remarks

Within the Standard Template Library, the sequence and sorted associative containers may be used as a container object *_Cont* with the templatized inserter.

Example

```
// iterator_inserter.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
   list <int>::iterator L_Iter;

   list<int> L;
   for (i = 2 ; i < 5 ; ++i )
   {
      L.push_back ( 10 * i );
   }

   cout << "The list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++ )
      cout << *L_Iter << " ";
   cout << ")." << endl;

   // Using the template version to insert an element
   insert_iterator<list <int> > Iter( L, L.begin ( ) );
   *Iter = 1;

   // Alternatively, using the member function to insert an element
   inserter ( L, L.end ( ) ) = 500;

   cout << "After the insertions, the list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
      cout << *L_Iter << " ";
   cout << ")." << endl;
}
```

Output

```
The list L is:
 ( 20 30 40 ).
After the insertions, the list L is:
 ( 1 20 30 40 500 ).
```

Requirements

Header: <iterator>

See Also

Reference

Standard Template Library

Other Resources

<iterator> Members

<iterator> Operators

For information about the operators in <iterator>, see <iterator> Members.

See Also

Reference

Standard Template Library

operator!= (<iterator>)

Tests if the iterator object on the left side of the operator is not equal to the iterator object on the right side.

```
template<class RandomIterator>
   bool operator!=(
      const reverse_iterator<RandomIterator>& _Left,
      const reverse_iterator<RandomIterator>& _Right
   );
template<class Type, class CharType, class Traits, class Distance>
   bool operator!=(
      const istream_iterator<Type, CharType, Traits, Distance>& _Left,
      const istream_iterator<Type, CharType, Traits, Distance>& _Right
   );
template<class CharType, class Tr>
   bool operator!=(
      const istreambuf_iterator<CharType, Traits>& _Left,
      const istreambuf_iterator<CharType, Traits>& _Right
);
```

Parameters

_Left

An object of type **iterator**.

_Right

An object of type **iterator**.

Return Value

true if the iterator objects are not equal; **false** if the iterator objects are equal.

Remarks

One iterator object is equal to another if they address the same elements in a container. If two iterators point to different elements in a container, then they are not equal.

Example

```
// iterator_op_ne.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for ( i = 1 ; i < 9 ; ++i )
   {
      vec.push_back ( i );
   }

   vector <int>::iterator vIter;

   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;

   // Initializing reverse_iterators to the last element
   vector <int>::reverse_iterator rVPOS1 = vec.rbegin ( ),
```

```
            rVPOS2 = vec.rbegin ( );

    cout << "The iterator rVPOS1 initially points to the first "
         << "element\n in the reversed sequence: "
         << *rVPOS1 << "." << endl;

    if ( rVPOS1 != rVPOS2 )
       cout << "The iterators are not equal." << endl;
    else
       cout << "The iterators are equal." << endl;

    rVPOS1++;
    cout << "The iterator rVPOS1 now points to the second "
         << "element\n in the reversed sequence: "
         << *rVPOS1 << "." << endl;

    if ( rVPOS1 != rVPOS2 )
       cout << "The iterators are not equal." << endl;
    else
       cout << "The iterators are equal." << endl;
 }
```

Output

```
The vector vec is: ( 1 2 3 4 5 6 7 8 ).
The iterator rVPOS1 initially points to the first element
 in the reversed sequence: 8.
The iterators are equal.
The iterator rVPOS1 now points to the second element
 in the reversed sequence: 7.
The iterators are not equal.
```

Requirements

Header: <iterator>

See Also

Reference

Standard Template Library

Other Resources

<iterator> Members

operator== (<iterator>)

Tests if the iterator object on the left side of the operator is equal to the iterator object on the right side.

```
template<class RandomIterator>
   bool operator==(
      const reverse_iterator<RandomIterator>& _Left,
      const reverse_iterator<RandomIterator>& _Right
   );
template<class Type, class CharType, class Traits, class Distance>
   bool operator==(
      const istream_iterator<Type, CharType, Traits, Distance>& _Left,
      const istream_iterator<Type, CharType, Traits, Distance>& _Right
   );
template<class CharType, class Tr>
   bool operator==(
      const istreambuf_iterator<CharType, Traits>& _Left,
      const istreambuf_iterator<CharType, Traits>& _Right
);
```

Parameters

_Left

An object of type **iterator**.

_Right

An object of type **iterator**.

Return Value

true if the iterator objects are equal; **false** if the iterator objects are not equal.

Remarks

One iterator object is equal to another if they address the same elements in a container. If two iterators point to different elements in a container, then they are not equal.

Example

```
// iterator_op_eq.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for ( i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( 2 * i );
   }

   vector <int>::iterator vIter;

   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   // Initializing reverse_iterators to the last element
   vector <int>::reverse_iterator rVPOS1 = vec.rbegin ( ),
```

```
            rVPOS2 = vec.rbegin ( );

    cout << "The iterator rVPOS1 initially points to the first "
         << "element\n in the reversed sequence: "
         << *rVPOS1 << "." << endl;

    if ( rVPOS1 == rVPOS2 )
       cout << "The iterators are equal." << endl;
    else
       cout << "The iterators are not equal." << endl;

    rVPOS1++;
    cout << "The iterator rVPOS1 now points to the second "
         << "element\n in the reversed sequence: "
         << *rVPOS1 << "." << endl;

    if ( rVPOS1 == rVPOS2 )
       cout << "The iterators are equal." << endl;
    else
       cout << "The iterators are not equal." << endl;
 }
```

Output

```
The vector vec is: ( 2 4 6 8 10 ).
The iterator rVPOS1 initially points to the first element
 in the reversed sequence: 10.
The iterators are equal.
The iterator rVPOS1 now points to the second element
 in the reversed sequence: 8.
The iterators are not equal.
```

Requirements

Header: <iterator>

See Also

Reference

Standard Template Library

Other Resources

<iterator> Members

operator< (<iterator>)

Tests if the iterator object on the left side of the operator is less than the iterator object on the right side.

```
template<class RandomIterator>
   bool operator<(
      const reverse_iterator<RandomIterator>& _Left,
      const reverse_iterator<RandomIterator>& _Right
   );
```

Parameters

_Left

An object of type **iterator**.

_Right

An object of type **iterator**.

Return Value

true if the iterator on the left side of the expression is less than the iterator on the right side of the expression; **false** if it is greater than or equal to the iterator on the right.

Remarks

One iterator object is less than another if it addresses an element that occurs earlier in the container than the element addressed by the other iterator object. One iterator object is not less than another if it addresses either the same element as the other iterator object or an element that occurs later in the container than the element addressed by the other iterator object.

Example

```
// iterator_op_lt.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for ( i = 0 ; i < 6 ; ++i )
   {
      vec.push_back ( 2 * i );
   }

   vector <int>::iterator vIter;

   cout << "The initial vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
      cout << *vIter << " ";
   cout << "))." << endl;

   // Initializing reverse_iterators to the last element
   vector <int>::reverse_iterator rVPOS1 = vec.rbegin ( ),
           rVPOS2 = vec.rbegin ( );

   cout << "The iterators rVPOS1& rVPOS2 initially point to the "
           << "first element\n in the reversed sequence: "
           << *rVPOS1 << "." << endl;

   if ( rVPOS1 < rVPOS2 )
```

```
      cout << "The iterator rVPOS1 is less than"
              << " the iterator rVPOS2." << endl;
   else
      cout << "The iterator rVPOS1 is not less than"
              << " the iterator rVPOS2." << endl;

   rVPOS2++;
   cout << "The iterator rVPOS2 now points to the second "
           << "element\n in the reversed sequence: "
           << *rVPOS2 << "." << endl;

   if ( rVPOS1 < rVPOS2 )
      cout << "The iterator rVPOS1 is less than"
              << " the iterator rVPOS2." << endl;
   else
      cout << "The iterator rVPOS1 is not less than"
              << " the iterator rVPOS2." << endl;
}
```

Output

```
The initial vector vec is: ( 0 2 4 6 8 10 ).
The iterators rVPOS1& rVPOS2 initially point to the first element
 in the reversed sequence: 10.
The iterator rVPOS1 is not less than the iterator rVPOS2.
The iterator rVPOS2 now points to the second element
 in the reversed sequence: 8.
The iterator rVPOS1 is less than the iterator rVPOS2.
```

Requirements

Header: <iterator>

See Also

Reference

Standard Template Library

Other Resources

<iterator> Members

operator<= (<iterator>)

Tests if the iterator object on the left side of the operator is less than or equal to the iterator object on the right side.

```
template<class RandomIterator>
   bool operator<=(
      const reverse_iterator<RandomIterator>& _Left,
      const reverse_iterator<RandomIterator>& _Right
   );
```

Parameters

_Left

An object of type iterator.

_Right

An object of type iterator.

Return Value

true if the iterator on the left side of the expression is less than or equal to the iterator on the right side of the expression; **false** if it is greater than the iterator on the right.

Remarks

One iterator object is less than or equal to another if it addresses the same element or an element that occurs earlier in the container than the element addressed by the other iterator object. One iterator object is greater than another if it addresses an element that occurs later in the container than the element addressed by the other iterator object.

Example

```
// iterator_op_le.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 0 ; i < 6 ; ++i )  {
      vec.push_back ( 2 * i );
      }

   vector <int>::iterator vIter;

   cout << "The initial vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
      cout << *vIter << " ";
   cout << "))." << endl;

   vector <int>::reverse_iterator rVPOS1 = vec.rbegin ( ) + 1,
           rVPOS2 = vec.rbegin ( );

   cout << "The iterator rVPOS1 initially points to the "
           << "second element\n in the reversed sequence: "
           << *rVPOS1 << "." << endl;

   cout << "The iterator rVPOS2 initially points to the "
           << "first element\n in the reversed sequence: "
           << *rVPOS2 << "." << endl;
```

```
    if ( rVPOS1 <= rVPOS2 )
       cout << "The iterator rVPOS1 is less than or "
               << "equal to the iterator rVPOS2." << endl;
    else
       cout << "The iterator rVPOS1 is greater than "
               << "the iterator rVPOS2." << endl;

    rVPOS2++;
    cout << "The iterator rVPOS2 now points to the second "
            << "element\n in the reversed sequence: "
            << *rVPOS2 << "." << endl;

    if ( rVPOS1 <= rVPOS2 )
       cout << "The iterator rVPOS1 is less than or "
               << "equal to the iterator rVPOS2." << endl;
    else
       cout << "The iterator rVPOS1 is greater than "
               << "the iterator rVPOS2." << endl;
 }
```

Output

```
The initial vector vec is: ( 0 2 4 6 8 10 ).
The iterator rVPOS1 initially points to the second element
 in the reversed sequence: 8.
The iterator rVPOS2 initially points to the first element
 in the reversed sequence: 10.
The iterator rVPOS1 is greater than the iterator rVPOS2.
The iterator rVPOS2 now points to the second element
 in the reversed sequence: 8.
The iterator rVPOS1 is less than or equal to the iterator rVPOS2.
```

Requirements

Header: <iterator>

See Also

Reference

Standard Template Library

Other Resources

<iterator> Members

operator> (<iterator>)

Tests if the iterator object on the left side of the operator is greater than the iterator object on the right side.

```
template<class RandomIterator>
   bool operator>(
      const reverse_iterator<RandomIterator>& _Left,
      const reverse_iterator<RandomIterator>& _Right
   );
```

Parameters

_Left

An object of type iterator.

_Right

An object of type iterator.

Return Value

true if the iterator on the left side of the expression is greater than the iterator on the right side of the expression; **false** if it is less than or equal to the iterator on the right.

Remarks

One iterator object is greater than another if it addresses an element that occurs later in the container than the element addressed by the other iterator object. One iterator object is not greater than another if it addresses either the same element as the other iterator object or an element that occurs earlier in the container than the element addressed by the other iterator object.

Example

```
// iterator_op_gt.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 0 ; i < 6 ; ++i )  {
      vec.push_back ( 2 * i );
      }

   vector <int>::iterator vIter;

   cout << "The initial vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   vector <int>::reverse_iterator rVPOS1 = vec.rbegin ( ),
           rVPOS2 = vec.rbegin ( );

   cout << "The iterators rVPOS1 & rVPOS2 initially point to "
           << "the first element\n in the reversed sequence: "
           << *rVPOS1 << "." << endl;

   if ( rVPOS1 > rVPOS2 )
      cout << "The iterator rVPOS1 is greater than "
              << "the iterator rVPOS2." << endl;
```

```
    else
       cout << "The iterator rVPOS1 is less than or "
               << "equal to the iterator rVPOS2." << endl;

    rVPOS1++;
    cout << "The iterator rVPOS1 now points to the second "
            << "element\n in the reversed sequence: "
            << *rVPOS1 << "." << endl;

    if ( rVPOS1 > rVPOS2 )
       cout << "The iterator rVPOS1 is greater than "
               << "the iterator rVPOS2." << endl;
    else
       cout << "The iterator rVPOS1 is less than or "
               << "equal to the iterator rVPOS2." << endl;
 }
```

Output

```
The initial vector vec is: ( 0 2 4 6 8 10 ).
The iterators rVPOS1 & rVPOS2 initially point to the first element
 in the reversed sequence: 10.
The iterator rVPOS1 is less than or equal to the iterator rVPOS2.
The iterator rVPOS1 now points to the second element
 in the reversed sequence: 8.
The iterator rVPOS1 is greater than the iterator rVPOS2.
```

Requirements

Header: <iterator>

See Also

Reference

Standard Template Library

Other Resources

<iterator> Members

operator>= (<iterator>)

Tests if the iterator object on the left side of the operator is greater than or equal to the iterator object on the right side.

```
template<class RandomIterator>
   bool operator>=(
      const reverse_iterator<RandomIterator>& _Left,
      const reverse_iterator<RandomIterator>& _Right
   );
```

Parameters

_Left

An object of type iterator.

_Right

An object of type iterator.

Return Value

true if the iterator on the left side of the expression is greater than or equal to the iterator on the right side of the expression; **false** if it is less than the iterator on the right.

Remarks

One iterator object is greater than or equal to another if it addresses the same element or an element that occurs later in the container than the element addressed by the other iterator object. One iterator object is less than another if it addresses an element that occurs earlier in the container than the element addressed by the other iterator object.

Example

```
// iterator_op_ge.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 0 ; i < 6 ; ++i )  {
      vec.push_back ( 2 * i );
      }

   vector <int>::iterator vIter;

   cout << "The initial vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
      cout << *vIter << " ";
   cout << "))." << endl;

   vector <int>::reverse_iterator rVPOS1 = vec.rbegin ( ),
           rVPOS2 = vec.rbegin ( ) + 1;

   cout << "The iterator rVPOS1 initially points to the "
           << "first element\n in the reversed sequence: "
           << *rVPOS1 << "." << endl;

   cout << "The iterator rVPOS2 initially points to the "
           << "second element\n in the reversed sequence: "
           << *rVPOS2 << "." << endl;
```

```
    if ( rVPOS1 >= rVPOS2 )
       cout << "The iterator rVPOS1 is greater than or "
               << "equal to the iterator rVPOS2." << endl;
    else
       cout << "The iterator rVPOS1 is less than "
               << "the iterator rVPOS2." << endl;

    rVPOS1++;
    cout << "The iterator rVPOS1 now points to the second "
            << "element\n in the reversed sequence: "
            << *rVPOS1 << "." << endl;

    if ( rVPOS1 >= rVPOS2 )
       cout << "The iterator rVPOS1 is greater than or "
               << "equal to the iterator rVPOS2." << endl;
    else
       cout << "The iterator rVPOS1 is less than "
               << "the iterator rVPOS2." << endl;
 }
```

Output

```
The initial vector vec is: ( 0 2 4 6 8 10 ).
The iterator rVPOS1 initially points to the first element
 in the reversed sequence: 10.
The iterator rVPOS2 initially points to the second element
 in the reversed sequence: 8.
The iterator rVPOS1 is less than the iterator rVPOS2.
The iterator rVPOS1 now points to the second element
 in the reversed sequence: 8.
The iterator rVPOS1 is greater than or equal to the iterator rVPOS2.
```

Requirements

Header: <iterator>

See Also

Reference

Standard Template Library

Other Resources

<iterator> Members

operator+ (<iterator>)

Adds an offset to an iterator and returns a **reverse_iterator** addressing the inserted element at the new offset position.

```
template<class RandomIterator>
   reverse_iterator<RandomIterator> operator+(
      typename reverse_iterator<Iterator>::difference_type _Off,
      const reverse_iterator<RandomIterator>& _Right
   );
```

Parameters

_Off

The number of positions the const reverse_iterator is to be offset.

_Right

The const reverse_iterator that is to be offset.

Return Value

The sum *_Right* + *_Off*.

Example

```
// iterator_op_insert.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 0 ; i < 6 ; ++i )  {
      vec.push_back ( 2 * i );
      }

   vector <int>::iterator vIter;

   cout << "The initial vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   vector <int>::reverse_iterator rVPOS1 = vec.rbegin ( );

   cout << "The iterator rVPOS1 initially points to "
           << "the first element\n in the reversed sequence: "
           << *rVPOS1 << "." << endl;

   vector<int>::difference_type diff = 4;
   rVPOS1 = diff +rVPOS1;

   cout << "The iterator rVPOS1 now points to the fifth "
           << "element\n in the reversed sequence: "
           << *rVPOS1 << "." << endl;
}
```

Output

```
The initial vector vec is: ( 0 2 4 6 8 10 ).
```

```
The iterator rVPOS1 initially points to the first element
 in the reversed sequence: 10.
The iterator rVPOS1 now points to the fifth element
 in the reversed sequence: 2.
```

Requirements

Header: <iterator>

See Also

Reference

Standard Template Library

Other Resources

<iterator> Members

operator- (<iterator>)

Subtracts one iterator from another and returns the difference.

```
template<class RandomIterator>
   typename reverse_iterator<RandomIterator>::difference_type operator-(
      const reverse_iterator<RandomIterator>& _Left,
      const reverse_iterator<RandomIterator>& _Right
   );
```

Parameters

_Left

An iterator that serves as the minuend from which another iterator is to be subtracted in forming the difference.

_Right

An iterator that serves as the subtrahend that is to be subtracted from other iterator in forming the difference.

Return Value

The difference between two iterators: *_Left* - *_Right*.

Remarks

The difference equals the minuend minus the subtrahend.

Example

```
// iterator_op_sub.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 0 ; i < 6 ; ++i )
   {
      vec.push_back ( 2 * i );
   }

   vector <int>::iterator vIter;

   cout << "The initial vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
      cout << *vIter << " ";
   cout << "))." << endl;

   vector <int>::reverse_iterator rVPOS1 = vec.rbegin ( ),
          rVPOS2 = vec.rbegin ( );

   cout << "The iterators rVPOS1 & rVPOS2 initially point to "
        << "the first element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

   for (i = 1; i < 5; ++i)
   {
      rVPOS2++;
   }
   cout << "The iterator rVPOS2 now points to the fifth "
        << "element\n in the reversed sequence: "
```

```
        << *rVPOS2 << "." << endl;

    vector<int>::difference_type diff = rVPOS2 - rVPOS1;
    cout << "The difference: rVPOS2 - rVPOS1= "
        << diff << "." << endl;
}
```

Output

```
The initial vector vec is: ( 0 2 4 6 8 10 ).
The iterators rVPOS1 & rVPOS2 initially point to the first element
 in the reversed sequence: 10.
The iterator rVPOS2 now points to the fifth element
 in the reversed sequence: 2.
The difference: rVPOS2 - rVPOS1= 4.
```

Requirements

Header: <iterator>

See Also

Reference

Standard Template Library

Other Resources

<iterator> Members

<iterator> Classes

For information about the classes in <iterator>, see <iterator> Members.

See Also

Reference

Standard Template Library

back_insert_iterator Class

Describes an iterator adaptor that satisfies the requirements of an output iterator. It inserts, rather than overwrites, elements into the back end of a sequence and thus provides semantics that are different from the overwrite semantics provided by the iterators of the C++ sequence containers. The back_insert_iterator class is templatized on the type of container.

For a list of all members of this type, see back_insert_iterator Members.

```
template <class Container>
class back_insert_iterator
```

Parameters

Container

The type of container into the back of which elements are to be inserted by a **back_insert_iterator**.

Remarks

The container must satisfy the requirements for a back insertion sequence where is it possible to insert elements at the end of the sequence in amortized constant time. STL sequence containers defined by the deque Class, list Class and vector Class provide the needed **push_back** member function and satisfy these requirements. These three containers as well as strings may each be adapted to use with **back_insert_iterator**s. A **back_insert_iterator** must always be initialized with its container.

Requirements

Header: <iterator>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

back_insert_iterator Members

<iterator> Members

back_insert_iterator Members

Reference

Constructors

back_insert_iterator	Constructs a **back_insert_iterator** that inserts elements after the last element in a container.

Typedefs

container_type	A type that provides a container for the **back_insert_iterator**.
reference	A type that provides a reference for the **back_insert_iterator**.

Operators

operator*	Dereferencing operator used to implement the output iterator expression *i = x for a back insertion.
operator++	Increments the **back_insert_iterator** to the next location into which a value may be stored.
operator=	Assignment operator used to implement the output iterator expression *i = x for a back insertion.

See Also

Reference

back_insert_iterator Class
Thread Safety in the Standard C++ Library
Standard Template Library

back_insert_iterator Typedefs

For information about the typedefs in the back_insert_iterator class, see back_insert_iterator Members.

See Also

Reference

Standard Template Library

back_insert_iterator::container_type

A type that provides a container for the **back_insert_iterator**.

```
typedef Container container_type;
```

Remarks

The type is a synonym for the template parameter **Container**.

Example

```
// back_insert_iterator_container_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 4 ; ++i )
   {
      vec.push_back (  i );
   }

   vector <int>::iterator vIter;
   cout << "The original vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << "))." << endl;

   back_insert_iterator<vector<int> >::container_type vec1 = vec;
   back_inserter ( vec1 ) = 40;

   cout << "After the insertion, the vector is: ( ";
   for ( vIter = vec1.begin ( ) ; vIter != vec1.end ( ); vIter++)
      cout << *vIter << " ";
   cout << "))." << endl;
}
```

Output

```
The original vector vec is: ( 1 2 3 ).
After the insertion, the vector is: ( 1 2 3 40 ).
```

Requirements

Header: <iterator>

See Also

Reference

back_insert_iterator Class

Standard Template Library

Other Resources

back_insert_iterator Members

back_insert_iterator::reference

A type that provides a reference for the **back_insert_iterator**.

```
typedef typename Container::reference reference;
```

Remarks

The type describes a reference to an element of the sequence controlled by the associated container.

Example

```
// back_insert_iterator_reference.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 4 ; ++i )
   {
      vec.push_back ( i );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   back_insert_iterator<vector<int> >::reference
        RefLast = *(vec.end ( ) - 1 );
   cout << "The last element in the vector vec is: "
        << RefLast << "." << endl;
}
```

Output

```
The vector vec is: ( 1 2 3 ).
The last element in the vector vec is: 3.
```

Requirements

Header: <iterator>

See Also

Reference

back_insert_iterator Class

Standard Template Library

Other Resources

back_insert_iterator Members

back_insert_iterator Member Functions

For information about the functions in the back_insert_iterator class, see back_insert_iterator Members.

See Also

Reference

Standard Template Library

back_insert_iterator::back_insert_iterator

Constructs a **back_insert_iterator** that inserts elements after the last element in a container.

```
explicit back_insert_iterator(
   Container& _Cont
);
```

Parameters

_Cont

The container that the **back_insert_iterator** is to insert an element into.

Return Value

A **back_insert_iterator** for the parameter container.

Example

```
// back_insert_iterator_back_insert_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for ( i = 1 ; i < 4 ; ++i )
   {
      vec.push_back ( i );
   }

   vector <int>::iterator vIter;
   cout << "The initial vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   // Insertions with member function
   back_inserter ( vec ) = 40;
   back_inserter ( vec ) = 50;

   // Alternatively, insertions can be done with template function
   back_insert_iterator<vector<int> > backiter ( vec );
   *backiter = 600;
   backiter++;
   *backiter = 700;

   cout << "After the insertions, the vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;
}
```

Output

```
The initial vector vec is: ( 1 2 3 ).
After the insertions, the vector vec is: ( 1 2 3 40 50 600 700 ).
```

Requirements

Header: <iterator>

See Also

Reference

back_insert_iterator Class

Standard Template Library

Other Resources

back_insert_iterator Members

back_insert_iterator Operators

For information about the operators in the back_insert_iterator class, see back_insert_iterator Class.

See Also

Reference

Standard Template Library

back_insert_iterator::operator*

Dereferencing operator used to implement the output iterator expression **i* = *x*.

```
back_insert_iterator& operator*( );
```

Return Value

A reference to the element inserted at the back of the container.

Remarks

Used to implement the output iterator expression ***Iter** = **value**. If **Iter** is an iterator that addresses an element in a sequence, then ***Iter** = **value** replaces that element with value and does not change the total number of elements in the sequence.

Example

```
// back_insert_iterator_back_insert.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 4 ; ++i )
   {
      vec.push_back ( i );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   back_insert_iterator<vector<int> > backiter ( vec );
   *backiter = 10;
   backiter++;      // Increment to the next element
   *backiter = 20;
   backiter++;

   cout << "After the insertions, the vector vec becomes: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;
}
```

Output

```
The vector vec is: ( 1 2 3 ).
After the insertions, the vector vec becomes: ( 1 2 3 10 20 ).
```

Requirements

Header: <iterator>

See Also

Reference

back_insert_iterator Class

Standard Template Library

Other Resources

back_insert_iterator Members

back_insert_iterator::operator++

Increments the **back_insert_iterator** to the next location into which a value may be stored.

```
back_insert_iterator& operator++( );
back_insert_iterator operator++( int );
```

Return Value

A **back_insert_iterator** addressing the next location into which a value may be stored.

Remarks

Both preincrementation and postincrementation operators return the same result.

Example

```
// back_insert_iterator_op_incre.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 3 ; ++i )
   {
      vec.push_back ( 10 * i );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   back_insert_iterator<vector<int> > backiter ( vec );
   *backiter = 30;
   backiter++;      // Increment to the next element
   *backiter = 40;
   backiter++;

   cout << "After the insertions, the vector vec becomes: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;
}
```

Output

```
The vector vec is: ( 10 20 ).
After the insertions, the vector vec becomes: ( 10 20 30 40 ).
```

Requirements

Header: <iterator>

See Also

Reference

back_insert_iterator Class

Standard Template Library

Other Resources

back_insert_iterator Members

back_insert_iterator::operator=

Assignment operator used to implement the output iterator expression **i = x*.

```
back_insert_iterator& operator=(
   typename Container::const_reference _Val
);
```

Parameters

_Val

The value to be inserted into the container.

Return Value

A reference to the last element inserted at the back of the container

Remarks

The member function evaluates **container.push_back**(*_Val*).

Example

```
// back_insert_iterator_op_assign.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 4 ; ++i )
   {
      vec.push_back ( i );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << "))." << endl;

   back_insert_iterator<vector<int> > backiter ( vec );
   *backiter = 10;
   backiter++;      // Increment to the next element
   *backiter = 20;
   backiter++;

   cout << "After the insertions, the vector vec becomes: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << "))." << endl;
}
```

Output

```
The vector vec is: ( 1 2 3 ).
After the insertions, the vector vec becomes: ( 1 2 3 10 20 ).
```

Requirements

Header: <iterator>

See Also

Reference

back_insert_iterator Class

Standard Template Library

Other Resources

back_insert_iterator Members

bidirectional_iterator_tag Class

A class that provides a return type for **iterator_category** function that represents a bidirectional iterator.

```
struct bidirectional_iterator_tag
   : public forward_iterator_tag {};
```

Remarks

The category tag classes are used as compile tags for algorithm selection. The template function needs to find the most specific category of its iterator argument, so that it can use the most efficient algorithm at compile time. For every iterator of type **Iterator**, **iterator_traits**<**Iterator**>::**iterator_category** must be defined to be the most specific category tag that describes the iterator's behavior.

The type is the same as **iterator**<**Iter**>::**iterator_category** when **Iter** describes an object that can serve as a bidirectional iterator.

Example

See random_access_iterator_tag for an example of how to use **bidirectional_iterator_tag**.

Requirements

Header: <iterator>

See Also

Reference

forward_iterator_tag Class
Thread Safety in the Standard C++ Library
Standard Template Library

Other Resources

<iterator> Members

checked_array_iterator Class

The **checked_array_iterator** class allows you to transform an array or pointer into a checked iterator.

For a list of all members of this type, see checked_array_iterator Members.

Note

This class is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be portable. For an example demonstrating how to write code that does not require the use of this class, see the second example below.

```
#include <iterator>
```

Remarks

This class is defined in the stdext namespace.

For more information on the checked iterator feature, see Checked Iterators.

Example

The following sample shows how to define and use a checked array iterator.

If the destination is not large enough to hold all the elements being copied, such as would be the case if you changed the line:

```
copy(a, a + 5, checked_array_iterator<int*>(b, 5));
```

to

```
copy(a, a + 5, checked_array_iterator<int*>(b, 4));
```

A runtime error will occur.

```
// checked_array_iterator_overview.cpp
// compile with: /EHsc
#include <algorithm>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
   int a[]={0, 1, 2, 3, 4};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b, 5));

   cout << "(";
   for (int i = 0 ; i < 5 ; i++)
      cout << " " << b[i];
   cout << " )" << endl;

   // constructor example
   checked_array_iterator<int*> checked_out_iter(b, 5);
   copy(a, a + 5, checked_out_iter);

   cout << "(";
   for (int i = 0 ; i < 5 ; i++)
      cout << " " << b[i];
   cout << " )" << endl;
}
```

Output

```
( 0 1 2 3 4 )
( 0 1 2 3 4 )
```

To avoid the need for the **checked_array_iterator** class when using Standard C++ Library algorithms, consider using a **vector** instead of a dynamically allocated array. The following example demonstrates how to do this.

```
// checked_array_iterator_2.cpp
// compile with: /EHsc

#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    std::vector<int> v(10);
    int *arr = new int[10];
    for (int i = 0; i < 10; ++i)
    {
        v[i] = i;
        arr[i] = i;
    }

    // std::copy(v.begin(), v.end(), arr); will result in
    // warning C4996. To avoid this warning while using int *,
    // use the Microsoft extension checked_array_iterator.
    std::copy(v.begin(), v.end(),
              stdext::checked_array_iterator<int *>(arr, 10));

    // Instead of using stdext::checked_array_iterator and int *,
    // consider using std::vector to encapsulate the array. This will
    // result in no warnings, and the code will be portable.
    std::vector<int> arr2(10);    // Similar to int *arr = new int[10];
    std::copy(v.begin(), v.end(), arr2.begin());

    for (int j = 0; j < arr2.size(); ++j)
    {
        cout << " " << arr2[j];
    }
    cout << endl;

    return 0;
}
```

Output

```
 0 1 2 3 4 5 6 7 8 9
```

See Also

Reference

Standard Template Library

Other Resources

<iterator> Members

checked_array_iterator Members

checked_array_iterator Members

Reference

Constructors

checked_array_iterator	Constructs a default **checked_array_iterator** or a **checked_array_iterator** from an underlying iterator.

Typedefs

difference_type	A type that provides the difference between two **checked_array_iterator**s referring to elements within the same container.
pointer	A type that provides a pointer to an element addressed by a **checked_array_iterator**.
reference	A type that provides a reference to an element addressed by a **checked_array_iterator**.

Member Functions

base	Recovers the underlying iterator from its **checked_array_iterator**.

Operators

operator==	Tests two **checked_array_iterator**s for equality.
operator!=	Tests two **checked_array_iterator**s for inequality.
operator<	Tests if the **checked_array_iterator** on the left side of the operator is less than the **checked_array_iterator** on the right side.
operator>	Tests if the **checked_array_iterator** on the left side of the operator is greater than the **checked_array_iterator** on the right side.
operator<=	Tests if the **checked_array_iterator** on the left side of the operator is less than or equal to the **checked_array_iterator** on the right side.
operator>=	Tests if the **checked_array_iterator** on the left side of the operator is greater than or equal to the **checked_array_iterator** on the right side.
operator*	Returns the element that a **checked_array_iterator** addresses.
operator->	Returns a pointer to the element addressed by the **checked_array_iterator**.
operator++	Increments the **checked_array_iterator** to the next element.
operator--	Decrements the **checked_array_iterator** to the previous element.
operator+=	Adds a specified offset to a **checked_array_iterator**.
operator+	Adds an offset to an iterator and returns the new **checked_array_iterator** addressing the inserted element at the new offset position.
operator-=	Decrements a specified offset from a **checked_array_iterator**.
operator-	Decrements an offset from an iterator and returns the new **checked_array_iterator** addressing the inserted element at the new offset position.

operator[]	Returns a reference to an element offset from the element addressed by a **checked_array_iterator** by a specified number of positions.

See Also

Reference

checked_array_iterator Class
Standard Template Library

checked_array_iterator Typedefs

For information about the typedefs in the checked_array_iterator class, see checked_array_iterator Class.

See Also

Reference

Standard Template Library

checked_array_iterator::difference_type

A type that provides the difference between two **checked_array_iterator**s referring to elements within the same container.

```
typedef typename iterator_traits<_Pointer>::difference_type difference_type;
```

Remarks

The **checked_array_iterator** difference type is the same as the iterator difference type.

See checked_array_iterator::operator[] for a code sample.

For more information, see Checked Iterators.

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

Standard Template Library

Other Resources

checked_array_iterator Members

checked_array_iterator::pointer

A type that provides a pointer to an element addressed by a **checked_array_iterator**.

```
typedef typename iterator_traits<_Pointer>::pointer pointer;
```

Remarks

See checked_array_iterator::operator* for a code sample.

For more information, see Checked Iterators.

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

Standard Template Library

Other Resources

checked_array_iterator Members

checked_array_iterator::reference

A type that provides a reference to an element addressed by a **checked_array_iterator**.

```
typedef typename iterator_traits<_Pointer>::reference reference;
```

Remarks

See checked_array_iterator::operator[] for a code sample.

For more information, see Checked Iterators.

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

Standard Template Library

Other Resources

checked_array_iterator Members

checked_array_iterator Member Functions

For information about the functions in the checked_array_iterator class, see checked_array_iterator Members.

See Also

Reference

Standard Template Library

checked_array_iterator::base

Recovers the underlying iterator from its **checked_array_iterator**.

```
_Pointer base() const;
```

Remarks

For more information, see Checked Iterators.

Example

```
// checked_array_iterators_base.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main() {
   using namespace std;

   int V1[10];

   for (int i = 0; i < 10 ; i++)
      V1[i] = i;

   int* bpos;

   stdext::checked_array_iterator<int*> rpos(V1, 10);
   rpos++;

   bpos = rpos.base ( );
   cout << "The iterator underlying rpos is bpos & it points to: "
        << *bpos << "." << endl;
}
```

Output

```
The iterator underlying rpos is bpos & it points to: 1.
```

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

Standard Template Library

Other Resources

checked_array_iterator Members

checked_array_iterator::checked_array_iterator

Constructs a default **checked_array_iterator** or a **checked_array _iterator** from an underlying iterator.

```
checked_array_iterator( );
checked_array_iterator(
   Pointer_Type ptr,
   size_t size,
   size_t index = 0
);
```

Parameters

ptr

A pointer to the array.

size

The size of the array.

index

(Optional) An element in the array, to initialize the iterator. By default, the iterator is initialized to the first element in the array.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_array_iterators_ctor.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
   int a[] = {0, 1, 2, 3, 4};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   for (int i = 0 ; i < 5 ; i++)
      cout << b[i] << " ";
   cout << endl;

   checked_array_iterator<int*> checked_output_iterator(b,5);
   copy (a, a + 5, checked_output_iterator);
   for (int i = 0 ; i < 5 ; i++)
      cout << b[i] << " ";
   cout << endl;

   checked_array_iterator<int*> checked_output_iterator2(b,5,3);
   cout << *checked_output_iterator2 << endl;
}
```

Output

```
0 1 2 3 4
0 1 2 3 4
3
```

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

Standard Template Library

Other Resources

checked_array_iterator Members

checked_array_iterator Operators

For information about the operators in the checked_array_iterator class, see checked_array_iterator Class.

See Also

Reference

Standard Template Library

checked_array_iterator::operator==

Tests two **checked_array_iterator**s for equality.

```
bool operator==(
   const checked_array_iterator& _Right
);
```

Parameters

_Right

The **checked_array_iterator** against which to check for equality.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_array_iterators_opeq.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
   int a[] = {0, 1, 2, 3, 4};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   checked_array_iterator<int*> checked_output_iterator(b,5);
   checked_array_iterator<int*> checked_output_iterator2(b,5);

   if (checked_output_iterator2 == checked_output_iterator)
      cout << "checked_array_iterators are equal" << endl;
   else
      cout << "checked_array_iterators are not equal" << endl;

   copy (a, a + 5, checked_output_iterator);
   checked_output_iterator++;

   if (checked_output_iterator2 == checked_output_iterator)
      cout << "checked_array_iterators are equal" << endl;
   else
      cout << "checked_array_iterators are not equal" << endl;
}
```

Output

```
checked_array_iterators are equal
checked_array_iterators are not equal
```

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

Standard Template Library

Other Resources

checked_array_iterator Members

checked_array_iterator::operator!=

Tests two **checked_array_iterator**s for inequality.

```
bool operator!=(
   const checked_array_iterator& _Right
);
```

Parameters

_Right

The **checked_array_iterator** against which to check for inequality.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_array_iterators_opneq.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
   int a[] = {0, 1, 2, 3, 4};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   checked_array_iterator<int*> checked_output_iterator(b,5);
   checked_array_iterator<int*> checked_output_iterator2(b,5);

   if (checked_output_iterator2 != checked_output_iterator)
      cout << "checked_array_iterators are not equal" << endl;
   else
      cout << "checked_array_iterators are equal" << endl;

   copy (a, a + 5, checked_output_iterator);
   checked_output_iterator++;

   if (checked_output_iterator2 != checked_output_iterator)
      cout << "checked_array_iterators are not equal" << endl;
   else
      cout << "checked_array_iterators are equal" << endl;
}
```

Output

```
checked_array_iterators are equal
checked_array_iterators are not equal
```

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

Standard Template Library

Other Resources

checked_array_iterator Members

checked_array_iterator::operator<

Tests if the **checked_array_iterator** on the left side of the operator is less than the **checked_array_iterator** on the right side.

```
bool operator<(
   const checked_array_iterator& _Right
);
```

Parameters

_Right

The **checked_array_iterator** against which to check for inequality.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_array_iterators_oplt.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
   int a[] = {0, 1, 2, 3, 4};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   checked_array_iterator<int*> checked_output_iterator(b,5);
   checked_array_iterator<int*> checked_output_iterator2(b,5);

   if (checked_output_iterator2 < checked_output_iterator)
      cout << "checked_output_iterator2 is less than checked_output_iterator" << endl;
   else
      cout << "checked_output_iterator2 is not less than checked_output_iterator" << endl;

   copy (a, a + 5, checked_output_iterator);
   checked_output_iterator++;

   if (checked_output_iterator2 < checked_output_iterator)
      cout << "checked_output_iterator2 is less than checked_output_iterator" << endl;
   else
      cout << "checked_output_iterator2 is not less than checked_output_iterator" << endl;
}
```

Output

```
checked_output_iterator2 is not less than checked_output_iterator
checked_output_iterator2 is less than checked_output_iterator
```

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

Standard Template Library

Other Resources

checked_array_iterator Members

checked_array_iterator::operator>

Tests if the **checked_array_iterator** on the left side of the operator is greater than the **checked_array_iterator** on the right side.

```
bool operator>(
   const checked_array_iterator& _Right
);
```

Parameters

_Right

The **checked_array_iterator** to compare against.

Remarks

See checked_array_iterator::operator< for a code sample.

For more information, see Checked Iterators.

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

Standard Template Library

Other Resources

checked_array_iterator Members

checked_array_iterator::operator<=

Tests if the **checked_array_iterator** on the left side of the operator is less than or equal to the **checked_array_iterator** on the right side.

```
bool operator<=(
   const checked_array_iterator& _Right
);
```

Parameters

_Right

The **checked_array_iterator** to compare against.

Remarks

See checked_array_iterator::operator>= for a code sample.

For more information, see Checked Iterators.

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

Standard Template Library

Other Resources

checked_array_iterator Members

checked_array_iterator::operator>=

Tests if the **checked_array_iterator** on the left side of the operator is greater than or equal to the **checked_array_iterator** on the right side.

```
bool operator>=(
   const checked_array_iterator& _Right
);
```

Parameters

_Right

The **checked_array_iterator** to compare against.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_array_iterators_opgteq.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
   int a[] = {0, 1, 2, 3, 4};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   checked_array_iterator<int*> checked_output_iterator(b,5);
   checked_array_iterator<int*> checked_output_iterator2(b,5);

   if (checked_output_iterator2 >= checked_output_iterator)
      cout << "checked_output_iterator2 is greater than or equal to checked_output_iterator
" << endl;
   else
      cout << "checked_output_iterator2 is less than checked_output_iterator" << endl;

   copy (a, a + 5, checked_output_iterator);
   checked_output_iterator++;

   if (checked_output_iterator2 >= checked_output_iterator)
      cout << "checked_output_iterator2 is greater than or equal to checked_output_iterator
" << endl;
   else
      cout << "checked_output_iterator2 is less than checked_output_iterator" << endl;
}
```

Output

```
checked_output_iterator2 is greater than or equal to checked_output_iterator
checked_output_iterator2 is less than checked_output_iterator
```

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

checked_array_iterator::operator*

Returns the element that a **checked_array_iterator** addresses.

```
reference operator*( ) const;
```

Return Value

The value of the element addressed by the **checked_array_iterator**.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_array_iterator_pointer.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <utility>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
   int a[] = {0, 1, 2, 3, 4};
   int b[5];
   pair<int, int> c[1];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   for (int i = 0 ; i < 5 ; i++)
      cout << b[i] << endl;

    c[0].first = 10;
    c[0].second = 20;

   checked_array_iterator<int*> checked_output_iterator(b,5);
   checked_array_iterator<int*>::pointer p = &(*checked_output_iterator);
   checked_array_iterator<pair<int, int>*> chk_c(c, 1);
   checked_array_iterator<pair<int, int>*>::pointer p_c = &(*chk_c);

   cout << "b[0] = " << *p << endl;
   cout << "c[0].first = " << p_c->first << endl;
}
```

Output

```
0
1
2
3
4
b[0] = 0
c[0].first = 10
```

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

checked_array_iterator::operator->

Returns a pointer to the element addressed by the **checked_array_iterator**.

```
pointer operator->( ) const;
```

Return Value

A pointer to the element addressed by the **checked_array_iterator**.

Remarks

See checked_array_iterator::pointer for a code sample.

For more information, see Checked Iterators.

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

Standard Template Library

Other Resources

checked_array_iterator Members

checked_array_iterator::operator++

Increments the **checked_array_iterator** to the next element.

```
checked_array_iterator& operator++();
checked_array_iterator operator++(int);
```

Return Value

The first operator returns the preincremented **checked_array_iterator** and the second, the postincrement operator, returns a copy of the incremented **checked_array_iterator**.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_array_iterators_op_plus_plus.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace stdext;
   using namespace std;
   int a[] = {6, 3, 77, 199, 222};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   checked_array_iterator<int*> checked_output_iterator(b,5);

   cout << *checked_output_iterator << endl;
   ++checked_output_iterator;
   cout << *checked_output_iterator << endl;
   checked_output_iterator++;
   cout << *checked_output_iterator << endl;
}
```

Output

```
6
3
77
```

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

Standard Template Library

Other Resources

checked_array_iterator Members

checked_array_iterator::operator--

Decrements the **checked_array_iterator** to the previous element.

```
checked_array_iterator& operator--( );
checked_array_iterator operator--( int );
```

Return Value

The first operator returns the predecremented **checked_array_iterator** and the second, the postdecrement operator, returns a copy of the decremented **checked_array_iterator**.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_array_iterators_op_minus_minus.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace stdext;
   using namespace std;
   int a[] = {6, 3, 77, 199, 222};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   checked_array_iterator<int*> checked_output_iterator(b,5);

   cout << *checked_output_iterator << endl;
   checked_output_iterator++;
   cout << *checked_output_iterator << endl;
   checked_output_iterator--;
   cout << *checked_output_iterator << endl;
}
```

Output

```
6
3
6
```

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

Standard Template Library

Other Resources

checked_array_iterator Members

checked_array_iterator::operator+=

Adds a specified offset to a **checked_array_iterator**.

```
checked_array_iterator& operator+=(
   difference_type _Off
);
```

Parameters

_Off

The offset by which to increment the iterator.

Return Value

A reference to the element addressed by the **checked_array_iterator**.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_array_iterators_op_plus_eq.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace stdext;
   using namespace std;
   int a[] = {6, 3, 77, 199, 222};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   checked_array_iterator<int*> checked_output_iterator(b,5);

   cout << *checked_output_iterator << endl;
   checked_output_iterator += 3;
   cout << *checked_output_iterator << endl;
}
```

Output

```
6
199
```

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

Standard Template Library

Other Resources

checked_array_iterator Members

checked_array_iterator::operator+

Adds an offset to an iterator and returns the new **checked_array_iterator** addressing the inserted element at the new offset position.

```
checked_array_iterator operator+(
   difference_type _Off
) const;
```

Parameters

_Off

The offset to be added to the **checked_array_iterator**.

Return Value

A **checked_array_iterator** addressing the offset element.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_array_iterators_op_plus.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace stdext;
   using namespace std;
   int a[] = {6, 3, 77, 199, 222};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   checked_array_iterator<int*> checked_output_iterator(b,5);

   cout << *checked_output_iterator << endl;
   checked_output_iterator = checked_output_iterator + 3;
   cout << *checked_output_iterator << endl;
}
```

Output

```
6
199
```

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

Standard Template Library

Other Resources

checked_array_iterator Members

checked_array_iterator::operator-=

Decrements a specified offset from a **checked_array_iterator**.

```
checked_array_iterator& operator-=(
   difference_type _Off
);
```

Parameters

_Off

The offset by which to increment the iterator.

Return Value

A reference to the element addressed by the **checked_array_iterator**.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_array_iterators_op_minus_eq.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace stdext;
   using namespace std;
   int a[] = {6, 3, 77, 199, 222};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   checked_array_iterator<int*> checked_output_iterator(b,5);

   checked_output_iterator += 3;
   cout << *checked_output_iterator << endl;
   checked_output_iterator -= 2;
   cout << *checked_output_iterator << endl;
}
```

Output

```
199
3
```

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

Standard Template Library

Other Resources

checked_array_iterator Members

checked_array_iterator::operator-

Decrements an offset from an iterator and returns the new **checked_array_iterator** addressing the inserted element at the new offset position.

```
checked_array_iterator operator-(
   difference_type _Off
) const;
```

Parameters

_Off

The offset to be decremented from the **checked_array_iterator**.

Return Value

A **checked_array_iterator** addressing the offset element.

Remarks

See checked_array_iterator::operator- for a code sample.

For more information, see Checked Iterators.

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

Standard Template Library

Other Resources

checked_array_iterator Members

checked_array_iterator::operator[]

Returns a reference to an element offset from the element addressed by a **checked_array_iterator** by a specified number of positions.

```
reference operator[](
   difference_type _Off
) const;
```

Parameters

_Off

The offset from the **checked_array_iterator** address.

Return Value

The reference to the element offset.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_array_iterators_op_diff.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace std;
   int V1[10];

   for (int i = 0; i < 10 ; i++)
      V1[i] = i;

   // Declare a difference type for a parameter
   stdext::checked_array_iterator<int*>::difference_type diff = 2;

   stdext::checked_array_iterator<int*> VChkIter(V1, 10);

   stdext::checked_array_iterator<int*>::reference refrpos = VChkIter [diff];

   cout << refrpos + 1 << endl;
}
```

Output

```
3
```

Requirements

Header: <iterator>

See Also

Reference

checked_array_iterator Class

Standard Template Library

Other Resources

checked_array_iterator Members

checked_iterator Class

The **checked_iterator** class allows you to transform an iterator into a checked iterator.

For a list of all members of this type, see checked_iterator Members.

Note
This class is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be portable.

```
template <class _Cont, class _Iter = typename _Cont::iterator>
    class checked_iterator : public iterator<
        typename iterator_traits<_Iter>::iterator_category,
        typename iterator_traits<_Iter>::value_type,
        typename iterator_traits<_Iter>::difference_type,
        typename iterator_traits<_Iter>::pointer,
        typename iterator_traits<_Iter>::reference>
;
```

Remarks

This class is defined in the stdext namespace.

This iterator adapter should only be used when you have a non-checked iterator. All Standard C++ Library containers will return a checked iterator if **_SECURE_SCL** is set to 1, so there is no need to use **checked_iterator** with **std::vector<T>::iterator**, for example. However, if you have a custom container which returns iterators that are non-checked, then use **checked_iterator**.

If your code relies heavily on non-Standard C++ Library containers and iterators and you do not want to use **checked_iterator**, then consider disabling these warnings using _SCL_SECURE_NO_WARNINGS or the other methods described in that topic.

For more information, see Checked Iterators.

Example

The following code sample shows how to use a function that enforces the use of checked iterators without defining **_SECURE_SCL**. This requires you to explicitly define checked iterators, for all iterators used by checked functions.

```
// checked_iterators.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <numeric>
#include <functional>
#include <iostream>

int main() {
   using namespace std;
   vector<int> V1( 10 ), V2( 10 );
   vector<int>::iterator VIter1, VIter2, VIterend, VIterend2;
   stdext::checked_iterator< vector<int> > VChkIterend(V1, V1.begin());
   stdext::checked_iterator< vector<int> > VChkIterend2(V2, V2.begin());


   list<int> L1;
   stdext::checked_iterator< list<int> > LChkIterend(L1, L1.begin());
   list<int>::iterator LIter1, LIterend;

   int t;
   for ( t = 1 ; t <= 10 ; t++ )
      L1.push_back( t );

   cout << "The input list L1 is:\n ( " ;
```

```
    for ( LIter1 = L1.begin() ; LIter1 != L1.end() ; LIter1++ )
       cout << *LIter1 << " ";
    cout << "))." << endl;

    // The first member function for the partial sums of
    // elements in a list output to a vector
    stdext::checked_iterator< vector<int> > V1chkbegin(V1, V1.begin());
    VChkIterend = stdext::checked_partial_sum ( L1.begin(), L1.end(), V1chkbegin );

    cout << "The output vector containing the partial sums is:\n ( " ;
    for ( VIter1 = V1.begin( ) ; VIter1 != VChkIterend.base() ; VIter1++ )
       cout << *VIter1 << " ";
    cout << "))." << endl;

    // The second member function used to compute
    // the partial product of the elements in a list
    stdext::checked_iterator< vector<int> > V2chkbegin(V2, V2.begin());
    VChkIterend2 = stdext::checked_partial_sum ( L1.begin(), L1.end(), V2chkbegin ,
       multiplies<int>( ) );

    cout << "The output vector with the partial products is:\n ( " ;
    for ( VIter2 = V2.begin() ; VIter2 != VChkIterend2.base() ; VIter2++ )
       cout << *VIter2 << " ";
    cout << "))." << endl;

    // Computation of partial sums in place
    stdext::checked_iterator< list<int> > L1chkbegin(L1, L1.begin());
    LChkIterend = stdext::checked_partial_sum ( L1.begin ( ) , L1.end ( ) , L1chkbegin );
    cout << "The in place output stdext::checked_partial_sum list L1 is:\n ( " ;
    for ( LIter1 = L1.begin( ) ; LIter1 != LChkIterend.base() ; LIter1++ )
       cout << *LIter1 << " ";
    cout << "))." << endl;
 }
```

Output

```
The input list L1 is:
 ( 1 2 3 4 5 6 7 8 9 10 ).
The output vector containing the partial sums is:
 ( 1 3 6 10 15 21 28 36 45 55 ).
The output vector with the partial products is:
 ( 1 2 6 24 120 720 5040 40320 362880 3628800 ).
The in place output stdext::checked_partial_sum list L1 is:
 ( 1 3 6 10 15 21 28 36 45 55 ).
```

See Also

Reference

<iterator>

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator Members

Reference

Constructors

checked_iterator	Constructs a default **checked_iterator** or a **checked_iterator** from an underlying iterator.

Typedefs

difference_type	A type that provides the difference between two **checked_iterator**s referring to elements within the same container.
pointer	A type that provides a pointer to an element addressed by a **checked_iterator**.
reference	A type that provides a reference to an element addressed by a **checked_iterator**.

Member Functions

base	Recovers the underlying iterator from its **checked_iterator**.

Operators

operator==	Tests two **checked_iterator**s for equality.
operator!=	Tests two **checked_iterator**s for inequality.
operator<	Tests if the **checked_iterator** on the left side of the operator is less than the **checked_iterator** on the right side.
operator>	Tests if the **checked_iterator** on the left side of the operator is greater than the **checked_iterator** on the right side.
operator<=	Tests if the **checked_iterator** on the left side of the operator is less than or equal to the **checked_iterator** on the right side.
operator>=	Tests if the **checked_iterator** on the left side of the operator is greater than or equal to the **checked_iterator** on the right side.
operator*	Returns the element that a **checked_iterator** addresses.
operator->	Returns a pointer to the element addressed by the **checked_iterator**.
operator++	Increments the **checked_iterator** to the next element.
operator--	Decrements the **checked_iterator** to the previous element.
operator+=	Adds a specified offset to a **checked_iterator**.
operator+	Adds an offset to an iterator and returns the new **checked_iterator** addressing the inserted element at the new offset position.
operator-=	Decrements a specified offset from a **checked_iterator**.
operator-	Decrements an offset from an iterator and returns the new **checked_iterator** addressing the inserted element at the new offset position.
operator[]	Returns a reference to an element offset from the element addressed by a **checked_iterator** by a specified number of positions.

See Also

Reference

checked_iterator Class
Thread Safety in the Standard C++ Library
Standard Template Library

checked_iterator Typedefs

For information about the typedefs in the checked_iterator class, see checked_iterator Class.

See Also

Reference

Standard Template Library

checked_iterator::difference_type

A type that provides the difference between two **checked_iterator**s referring to elements within the same container.

```
typedef typename iterator_traits<_Iter>::difference_type difference_type;
```

Remarks

The **checked_iterator** difference type is the same as the iterator difference type.

For more information, see Checked Iterators.

Example

See checked_iterator::operator[] for an example of how to declare and use **difference_type**.

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator::pointer

A type that provides a pointer to an element addressed by a **checked_iterator**.

```
typedef typename iterator_traits<_Iter>::pointer pointer;
```

Remarks

For more information, see Checked Iterators.

Example

```
// checked_iterator_pointer.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <utility>
#include <iostream>

int main() {
   using namespace std;

   typedef vector<pair<int,int> > pVector;
   pVector vec;
   vec.push_back( pVector::value_type( 1,2 ) );
   vec.push_back( pVector::value_type( 3,4 ) );
   vec.push_back( pVector::value_type( 5,6 ) );

   stdext::checked_iterator<pVector > pvIter(vec, vec.begin ( ));
   stdext::checked_iterator<pVector >::pointer p = &(*pvIter);

   cout << "The first pair is:\n" << "(" << p -> first << ", "
        << p -> second << ")" << endl;
}
```

Output

```
The first pair is:
(1, 2)
```

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator::reference

A type that provides a reference to an element addressed by a **checked_iterator**.

```
typedef typename iterator_traits<_Iter>::reference reference;
```

Remarks

For more information, see Checked Iterators.

Example

See checked_iterator::operator[] for an example of how to declare and use a **reference**.

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator Member Functions

For information about the functions in the checked_iterator class, see checked_iterator Members.

See Also

Reference

Standard Template Library

checked_iterator::base

Recovers the underlying iterator from its **checked_iterator**.

```
InputIterator base( ) const;
```

Remarks

For more information, see Checked Iterators.

Example

```
// checked_iterators_base.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main() {
   using namespace std;

   vector<int> V1;

   for (int i = 0; i < 10 ; i++)
      V1.push_back(i);

   vector <int>::iterator bpos;

   stdext::checked_iterator< vector<int> > rpos(V1, V1.begin());
   rpos++;

   bpos = rpos.base ( );
   cout << "The iterator underlying rpos is bpos & it points to: "
        << *bpos << "." << endl;
}
```

Output

```
The iterator underlying rpos is bpos & it points to: 1.
```

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator::checked_iterator

Constructs a default **checked_iterator** or a **checked_iterator** from an underlying iterator.

```
checked_iterator( );
checked_iterator(
   Container& _C,
   InputIterator _P
);
checked_iterator(
    const checked_iterator<Container, InputIterator> &_Right
);
template <class Iter2>
    checked_iterator(
        const checked_iterator<Container, Iter2> &_Right
);
```

Remarks

For more information, see Checked Iterators.

Example

```
// checked_iterators_ctor.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   std::vector<int> V1( 10 );
   stdext::checked_iterator< std::vector<int> > VChkIter1(V1, V1.begin());
   stdext::checked_iterator< std::vector<int> > VChkIter2;
}
```

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator Operators

For information about the operators in the checked_iterator class, see checked_iterator Class.

See Also

Reference

Standard Template Library

checked_iterator::operator==

Tests two **checked_iterator**s for equality.

```
template <class Iter2>
    bool operator==(
        const checked_iterator<Container, Iter2>& _Right) const;
```

Parameters

_Right

The **checked_iterator** against which to check for equality.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_iterators_opeq.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace std;
   vector<int> V1( 10 );
   stdext::checked_iterator< vector<int> > VChkIterend(V1, V1.begin());
   stdext::checked_iterator< vector<int> > VChkIterend2(V1, V1.begin());

   if (VChkIterend == VChkIterend2)
      cout << "checked iterators are equal" << endl;
   else
      cout << "checked iterators are not equal" << endl;

   VChkIterend++;
   if (VChkIterend == VChkIterend2)
      cout << "checked iterators are equal" << endl;
   else
      cout << "checked iterators are not equal" << endl;
}
```

Output

```
checked iterators are equal
checked iterators are not equal
```

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator::operator!=

Tests two **checked_iterator**s for inequality.

```
template <class Iter2>
    bool operator!=(
        const checked_iterator<Container, Iter2>& _Right) const;
```

Parameters

_Right

The **checked_iterator** against which to check for inequality.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_iterators_opneq.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace std;
   vector<int> V1( 10 );
   stdext::checked_iterator< vector<int> > VChkIterend(V1, V1.begin());
   stdext::checked_iterator< vector<int> > VChkIterend2(V1, V1.begin());

   if (VChkIterend != VChkIterend2)
      cout << "checked iterators are not equal" << endl;
   else
      cout << "checked iterators are equal" << endl;

   VChkIterend++;

   if (VChkIterend != VChkIterend2)
      cout << "checked iterators are not equal" << endl;
   else
      cout << "checked iterators are equal" << endl;
}
```

Output

```
checked iterators are equal
checked iterators are not equal
```

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator::operator<

Tests if the **checked_iterator** on the left side of the operator is less than the **checked_iterator** on the right side.

```
template <class Iter2>
    bool operator<(
        const checked_iterator<Container, Iter2>& _Right) const;
```

Parameters

_Right

The **checked_iterator** against which to check for inequality.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_iterators_oplt.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace std;
   vector<int> V1( 10 );
   stdext::checked_iterator< vector<int> > VChkIter1(V1, V1.begin());
   stdext::checked_iterator< vector<int> > VChkIter2(V1, V1.begin());

   VChkIter1++;
   VChkIter2++;

   if (VChkIter1 < VChkIter2)
      cout << "VChkIter1 is less than VChkIter2" << endl;
   else
      cout << "VChkIter1 is not less than VChkIter2" << endl;

   VChkIter2++;

   if (VChkIter1 < VChkIter2)
      cout << "VChkIter1 is less than VChkIter2" << endl;
   else
      cout << "VChkIter1 is not less than VChkIter2" << endl;
}
```

Output

```
VChkIter1 is not less than VChkIter2
VChkIter1 is less than VChkIter2
```

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator::operator>

Tests if the **checked_iterator** on the left side of the operator is greater than the **checked_iterator** on the right side.

```
template <class Iter2>
    bool operator>(
        const checked_iterator<Container, Iter2>& _Right) const;
```

Parameters

_Right

The **checked_iterator** to compare against.

Remarks

See checked_iterator::operator< for a code sample.

For more information, see Checked Iterators.

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator::operator<=

Tests if the **checked_iterator** on the left side of the operator is less than or equal to the **checked_iterator** on the right side.

```
template <class Iter2>
    bool operator<=(
        const checked_iterator<Container, Iter2>& _Right) const;
```

Parameters

_Right

The **checked_iterator** to compare against.

Remarks

See checked_iterator::operator>= for a code sample.

For more information, see Checked Iterators.

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator::operator>=

Tests if the **checked_iterator** on the left side of the operator is greater than or equal to the **checked_iterator** on the right side.

```
template <class Iter2>
    bool operator>=(
        const checked_iterator<Container, Iter2>& _Right) const;
```

Parameters

_Right

The **checked_iterator** to compare against.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_iterators_opgteq.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace std;
   vector<int> V1( 10 );
   stdext::checked_iterator< vector<int> > VChkIter1(V1, V1.begin());
   stdext::checked_iterator< vector<int> > VChkIter2(V1, V1.begin());

   VChkIter1++;
   VChkIter2++;

   if (VChkIter1 >= VChkIter2)
      cout << "VChkIter1 is greater than or equal to VChkIter2" << endl;
   else
      cout << "VChkIter1 is not greater than or equal to VChkIter2" << endl;

   VChkIter2++;

   if (VChkIter1 >= VChkIter2)
      cout << "VChkIter1 is greater than or equal to VChkIter2" << endl;
   else
      cout << "VChkIter1 is not greater than or equal to VChkIter2" << endl;
}
```

Output

```
VChkIter1 is greater than or equal to VChkIter2
VChkIter1 is not greater than or equal to VChkIter2
```

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator::operator*

Returns the element that a **checked_iterator** addresses.

```
reference operator*( ) const;
```

Return Value

The value of the element addressed by the **checked_iterator**.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_iterators_op_pointer.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace std;
   vector<int> V1;
   for (int i = 0 ; i < 10 ; i++)
      V1.push_back(i);

   stdext::checked_iterator< vector<int> > VChkIter1(V1, V1.begin());

   cout << *VChkIter1 << endl;
   VChkIter1++;
   cout << *VChkIter1 << endl;
}
```

Output

```
0
1
```

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator::operator->

Returns a pointer to the element addressed by the **checked_iterator**.

```
pointer operator->( ) const;
```

Return Value

A pointer to the element addressed by the **checked_iterator**.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_iterator_ptrto.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <utility>
#include <iostream>

int main() {
   using namespace std;

   typedef vector<pair<int,int> > pVector;
   pVector vec;
   vec.push_back(pVector::value_type(1,2));
   vec.push_back(pVector::value_type(3,4));
   vec.push_back(pVector::value_type(5,6));

   stdext::checked_iterator<pVector > pvIter(vec, vec.begin ( ));
   stdext::checked_iterator<pVector > pvIterEnd(vec, vec.end ( ));
   cout << "The vector vec of integer pairs is:\n( ";
   for ( ; pvIter != pvIterEnd; pvIter++)
      cout << "( " << pvIter -> first << ", " << pvIter -> second << ") ";
   cout << ")" << endl << endl;
}
```

Output

```
The vector vec of integer pairs is:
( ( 1, 2) ( 3, 4) ( 5, 6) )
```

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator::operator++

Increments the **checked_iterator** to the next element.

```
checked_iterator& operator++( );
checked_iterator operator++( int );
```

Return Value

The first operator returns the preincremented **checked_iterator** and the second, the postincrement operator, returns a copy of the incremented **checked_iterator**.

See checked_iterator::operator>= for a code sample.

Remarks

For more information, see Checked Iterators.

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator::operator--

Decrements the **checked_iterator** to the previous element.

```
checked_iterator& operator--( );
checked_iterator operator--( int );
```

Return Value

The first operator returns the predecremented **checked_iterator** and the second, the postdecrement operator, returns a copy of the decremented **checked_iterator**.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_iterators_op_minus_minus.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace std;
   vector<int> V1;

   for (int i = 0; i < 10 ; i++)
      V1.push_back(i);

   stdext::checked_iterator< vector<int> > VChkIter(V1, V1.end());

   VChkIter--;
   cout << *VChkIter << endl;
   VChkIter--;
   cout << *VChkIter << endl;
}
```

Output

```
9
8
```

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator::operator+=

Adds a specified offset to a **checked_iterator**.

```
checked_iterator& operator+=(
   difference_type _Off
);
```

Parameters

_Off

The offset by which to increment the iterator.

Return Value

A reference to the element addressed by the **checked_iterator**.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_iterators_op_plus_eq.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace std;
   vector<int> V1;

   for (int i = 0; i < 10 ; i++)
      V1.push_back(i);

   stdext::checked_iterator< vector<int> > VChkIter(V1, V1.begin());

   cout << *VChkIter << endl;
   VChkIter += 4;
   cout << *VChkIter << endl;
}
```

Output

```
0
4
```

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator::operator+

Adds an offset to an iterator and returns the new **checked_iterator** addressing the inserted element at the new offset position.

```
checked_iterator operator+(
   difference_type _Off
) const;
```

Parameters

_Off

The offset to be added to the **checked_iterator**.

Return Value

A **checked_iterator** addressing the offset element.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_iterators_op_plus.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace std;
   vector<int> V1;

   for (int i = 0; i < 10 ; i++)
      V1.push_back(i);

   stdext::checked_iterator< vector<int> > VChkIter(V1, V1.begin());

   cout << *VChkIter << endl;
   VChkIter = VChkIter + 5;
   cout << *VChkIter << endl;
}
```

Output

```
0
5
```

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator::operator-=

Decrements a specified offset from a **checked_iterator**.

```
checked_iterator& operator-=(
   difference_type _Off
);
```

Parameters

_Off

The offset by which to increment the iterator.

Return Value

A reference to the element addressed by the **checked_iterator**.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_iterators_op_minus_eq.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace std;
   vector<int> V1;

   for (int i = 0; i < 10 ; i++)
      V1.push_back(i);

   stdext::checked_iterator< vector<int> > VChkIter(V1, V1.end());

   VChkIter -= 1;
   cout << *VChkIter << endl;
   VChkIter -= 4;
   cout << *VChkIter << endl;
}
```

Output

```
9
5
```

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator::operator-

Decrements an offset from an iterator and returns the new **checked_iterator** addressing the inserted element at the new offset position.

```
checked_iterator operator-(
   difference_type _Off
) const;
```

Parameters

_Off

The offset to be decremented from the **checked_iterator**.

Return Value

A **checked_iterator** addressing the offset element.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_iterators_op_minus.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace std;
   vector<int> V1;

   for (int i = 0; i < 10 ; i++)
      V1.push_back(i);

   stdext::checked_iterator< vector<int> > VChkIter(V1, V1.end());

   VChkIter = VChkIter - 1;
   cout << *VChkIter << endl;
   VChkIter = VChkIter - 4;
   cout << *VChkIter << endl;
}
```

Output

```
9
5
```

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

checked_iterator::operator[]

Returns a reference to an element offset from the element addressed by a **checked_iterator** by a specified number of positions.

```
reference operator[](
   difference_type _Off
) const;
```

Parameters

_Off

The offset from the **checked_iterator** address.

Return Value

The reference to the element offset.

Remarks

For more information, see Checked Iterators.

Example

```
// checked_iterators_op_diff.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace std;
   vector<int> V1;

   for (int i = 0; i < 10 ; i++)
      V1.push_back(i);

   // Declare a difference type for a parameter
   stdext::checked_iterator<vector<int> >::difference_type diff = 2;

   stdext::checked_iterator< vector<int> > VChkIter(V1, V1.begin());

   stdext::checked_iterator<vector<int> >::reference refrpos = VChkIter [diff];

   cout << refrpos + 1 << endl;
}
```

Output

```
3
```

Requirements

Header: <iterator>

See Also

Reference

checked_iterator Class

Standard Template Library

Other Resources

checked_iterator Members

forward_iterator_tag Class

A class that provides a return type for **iterator_category** function that represents a forward iterator.

```
struct forward_iterator_tag
   : public input_iterator_tag {};
```

Remarks

The category tag classes are used as compile tags for algorithm selection. The template function needs to find out what is the most specific category of its iterator argument so that it can use the most efficient algorithm at compile time. For every iterator of type **Iterator**, **iterator_traits**<**Iterator**>**::iterator_category** must be defined to be the most specific category tag that describes the iterator's behavior.

The type is the same as **iterator**<**Iter**>**::iterator_category** when **Iter** describes an object that can serve as a forward iterator.

Example

See iterator_traits or random_access_iterator_tag for an example of how to use the **iterator_tag**s.

Requirements

Header: <iterator>

See Also

Reference

input_iterator_tag Class
Thread Safety in the Standard C++ Library
Standard Template Library

Other Resources

<iterator> Members

front_insert_iterator Class

Describes an iterator adaptor that satisfies the requirements of an output iterator. It inserts, rather than overwrites, elements into the front of a sequence and thus provides semantics that are different from the overwrite semantics provided by the iterators of the C++ sequence containers. The **front_insert_iterator** class is templatized on the type of container.

For a list of all members of this type, see front_insert_iterator Members.

```
template <class Container>
class front_insert_iterator
```

Parameters

Container

The type of container into the front of which elements are to be inserted by a **front_insert_iterator**.

Remarks

The container must satisfy the requirements for a front insertion sequence where is it possible to insert elements at the beginning of the sequence in amortized constant time. The Standard Template Library sequence containers defined by the deque Class and list Class provide the needed **push_front** member function and satisfy these requirements. By contrast, sequence containers defined by the vector Class do not satisfy these requirements and cannot be adapted to use with **front_insert_iterator**s. A **front_insert_iterator** must always be initialized with its container.

Requirements

Header: <iterator>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

front_insert_iterator Members

<iterator> Members

front_insert_iterator Members

Reference

Constructors

front_insert_iterator	Creates an iterator that can insert elements at the front of a specified container object.

Typedefs

container_type	A type that represents the container into which a front insertion is to be made.
reference	A type that provides a reference to an element in a sequence controlled by the associated container.

Operators

operator*	Dereferencing operator used to implement the output iterator expression *i = x for a front insertion.
operator++	Increments the **front_insert_iterator** to the next location into which a value may be stored.
operator=	Assignment operator used to implement the output iterator expression *i = x for a front insertion.

See Also

Reference

front_insert_iterator Class
Thread Safety in the Standard C++ Library
Standard Template Library

front_insert_iterator Typedefs

For information about the typedefs in the front_insert_iterator class, see front_insert_iterator Members.

See Also

Reference

Standard Template Library

front_insert_iterator::container_type

A type that represents the container into which a front insertion is to be made.

```
typedef Container container_type;
```

Remarks

The type is a synonym for the template parameter **Container**.

Example

```
// front_insert_iterator_container_type.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   list<int> L1;
   front_insert_iterator<list<int> >::container_type L2 = L1;
   front_inserter ( L2 ) = 20;
   front_inserter ( L2 ) = 10;
   front_inserter ( L2 ) = 40;

   list <int>::iterator vIter;
   cout << "The list L2 is: ( ";
   for ( vIter = L2.begin ( ) ; vIter != L2.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;
}
```

Output

```
The list L2 is: ( 40 10 20 ).
```

Requirements

Header: <iterator>

See Also

Reference

front_insert_iterator Class

Standard Template Library

Other Resources

front_insert_iterator Members

front_insert_iterator::reference

A type that provides a reference to an element in a sequence controlled by the associated container.

```
typedef typename Container::reference reference;
```

Example

```
// front_insert_iterator_reference.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   list<int> L;
   front_insert_iterator<list<int> > fiivIter( L );
   *fiivIter = 10;
   *fiivIter = 20;
   *fiivIter = 30;

   list<int>::iterator LIter;
   cout << "The list L is: ( ";
   for ( LIter = L.begin ( ) ; LIter != L.end ( ); LIter++)
      cout << *LIter << " ";
   cout << "))." << endl;

   front_insert_iterator<list<int> >::reference
        RefFirst = *(L.begin ( ));
   cout << "The first element in the list L is: "
        << RefFirst << "." << endl;
}
```

Output

```
The list L is: ( 30 20 10 ).
The first element in the list L is: 30.
```

Requirements

Header: <iterator>

See Also

Reference

front_insert_iterator Class

Standard Template Library

Other Resources

front_insert_iterator Members

front_insert_iterator Member Functions

For information about the functions in the front_insert_iterator class, see front_insert_iterator Members.

See Also

Reference

Standard Template Library

front_insert_iterator::front_insert_iterator

Creates an iterator that can insert elements at the front of a specified container object.

```
explicit front_insert_iterator(
   Container& _Cont
);
```

Parameters

_Cont

The container object into which the **front_insert_iterator** is to insert elements.

Return Value

A **front_insert_iterator** for the parameter container object.

Example

```
// front_insert_iterator_front_insert_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
   list <int>::iterator L_Iter;

   list<int> L;
   for (i = -1 ; i < 9 ; ++i )
   {
      L.push_back ( 2 * i );
   }

   cout << "The list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
      cout << *L_Iter << " ";
   cout << ")." << endl;

   // Using the member function to insert an element
   front_inserter ( L ) = 20;

   // Alternatively, one may use the template function
   front_insert_iterator< list < int> > Iter(L);
   *Iter = 30;

   cout << "After the front insertions, the list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
      cout << *L_Iter << " ";
   cout << ")." << endl;
}
```

Output

```
The list L is:
 ( -2 0 2 4 6 8 10 12 14 16 ).
After the front insertions, the list L is:
 ( 30 20 -2 0 2 4 6 8 10 12 14 16 ).
```

Requirements

Header: <iterator>

See Also

Reference

front_insert_iterator Class

Standard Template Library

Other Resources

front_insert_iterator Members

front_insert_iterator Operators

For information about the operators in front_insert_iterator class, see front_insert_iterator Class.

See Also

Reference

Standard Template Library

front_insert_iterator::operator*

Dereferences the insert iterator returning the element it addresses.

```
front_insert_iterator& operator*( );
```

Return Value

The member function returns the value of the element addressed.

Remarks

Used to implement the output iterator expression ***Iter** = **value**. If **Iter** is an iterator that addresses an element in a sequence, then ***Iter** = **value** replaces that element with value and does not change the total number of elements in the sequence.

Example

```
// front_insert_iterator_deref.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
   list <int>::iterator L_Iter;

   list<int> L;
   for ( i = -1 ; i < 9 ; ++i )
   {
      L.push_back ( 2 * i );
   }

   cout << "The list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
      cout << *L_Iter << " ";
   cout << ")." << endl;

   front_insert_iterator< list < int> > Iter(L);
   *Iter = 20;

   // Alternatively, you may use
   front_inserter ( L ) = 30;

   cout << "After the front insertions, the list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
      cout << *L_Iter << " ";
   cout << ")." << endl;
}
```

Output

```
The list L is:
 ( -2 0 2 4 6 8 10 12 14 16 ).
After the front insertions, the list L is:
 ( 30 20 -2 0 2 4 6 8 10 12 14 16 ).
```

Requirements

Header: <iterator>

See Also

Reference

front_insert_iterator Class
Standard Template Library

Other Resources

front_insert_iterator Members

front_insert_iterator::operator++

Increments the **back_insert_iterator** to the next location into which a value may be stored.

```
front_insert_iterator& operator++( );
front_insert_iterator operator++( int );
```

Return Value

A **front_insert_iterator** addressing the next location into which a value may be stored.

Remarks

Both preincrementation and postincrementation operators return the same result.

Example

```
// front_insert_iterator_op_incre.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   list<int> L1;
   front_insert_iterator<list<int> > iter ( L1 );
   *iter = 10;
   iter++;
   *iter = 20;
   iter++;
   *iter = 30;
   iter++;

   list <int>::iterator vIter;
   cout << "The list L1 is: ( ";
   for ( vIter = L1.begin ( ) ; vIter != L1.end ( ); vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;
}
```

Output

```
The list L1 is: ( 30 20 10 ).
```

Requirements

Header: <iterator>

See Also

Reference

front_insert_iterator Class

Standard Template Library

Other Resources

front_insert_iterator Members

front_insert_iterator::operator=

Assignment operator used to implement the output iterator expression **i = x*.

```
front_insert_iterator& operator=(
   typename Container::const_reference _Val
);
```

Parameters

_Val

The value to be assigned to the element.

Return Value

A reference to the last element inserted at the front of the container.

Remarks

The member function evaluates **container.push_front**(*_Val*).

Example

```
// front_insert_iterator_op_assign.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   list<int> L1;
   front_insert_iterator<list<int> > iter ( L1 );
   *iter = 10;
   iter++;
   *iter = 20;
   iter++;
   *iter = 30;
   iter++;

   list <int>::iterator vIter;
   cout << "The list L1 is: ( ";
   for ( vIter = L1.begin ( ) ; vIter != L1.end ( ); vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;
}
```

Output

```
The list L1 is: ( 30 20 10 ).
```

Requirements

Header: <iterator>

See Also

Reference

front_insert_iterator Class

Standard Template Library

Other Resources

front_insert_iterator Members

insert_iterator Class

Describes an iterator adaptor that satisfies the requirements of an output iterator. It inserts, rather than overwrites, elements into a sequence and thus provides semantics that are different from the overwrite semantics provided by the iterators of the C++ sequence and associative containers. The insert_iterator class is templatized on the type of container being adapted.

For a list of all members of this type, see insert_iterator Members.

```
template <class Container>
class insert_iterator
```

Parameters

Container

The type of container into which elements are to be inserted by an **insert_iterator**.

Remarks

The container of type **Container** must satisfy the requirements for a variable-sized container and have a two-argument insert member function where the parameters are of type **Container::iterator** and **Container::value_type** and that returns a type **Container::iterator**. Standard Template Library sequence and sorted associative containers satisfy these requirements and can be adapted to use with **insert_iterator**s. For associative containers, the position argument is treated as a hint, which has the potential to improve or degrade performance depending on how good the hint is. An **insert_iterator** must always be initialized with its container.

Requirements

Header: <iterator>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

insert_iterator Members

<iterator> Members

insert_iterator Members

Reference

Constructors

insert_iterator	Constructs an **insert_iterator** that inserts an element into a specified position in a container.

Typedefs

container_type	A type that represents the container into which a general insertion is to be made.
reference	A type that provides a reference to an element in a sequence controlled by the associated container.

Operators

operator*	Dereferencing operator used to implement the output iterator expression *i = x for a general insertion.
operator++	Increments the **insert_iterator** to the next location into which a value may be stored.
operator=	Assignment operator used to implement the output iterator expression *i = x for a general insertion.

See Also

Reference

insert_iterator Class
Thread Safety in the Standard C++ Library
Standard Template Library

insert_iterator Typedefs

For information about the typedefs in the insert_iterator class, see insert_iterator Members.

See Also

Reference

Standard Template Library

insert_iterator::container_type

A type that represents the container into which a general insertion is to be made.

```
typedef Container container_type;
```

Remarks

The type is a synonym for the template parameter **Container**.

Example

```
// insert_iterator_container_type.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   list<int> L1;
   insert_iterator<list<int> >::container_type L2 = L1;
   inserter ( L2, L2.end ( ) ) = 20;
   inserter ( L2, L2.end ( ) ) = 10;
   inserter ( L2, L2.begin ( ) ) = 40;

   list <int>::iterator vIter;
   cout << "The list L2 is: ( ";
   for ( vIter = L2.begin ( ) ; vIter != L2.end ( ); vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;
}
```

Output

```
The list L2 is: ( 40 20 10 ).
```

Requirements

Header: <iterator>

See Also

Reference

insert_iterator Class

Standard Template Library

Other Resources

insert_iterator Members

insert_iterator::reference

A type that provides a reference to an element in a sequence controlled by the associated container.

```
typedef typename Container::reference reference;
```

Remarks

The type describes a reference to an element of the sequence controlled by the associated container.

Example

```
// insert_iterator_container_reference.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   list<int> L;
   insert_iterator<list<int> > iivIter( L , L.begin ( ) );
   *iivIter = 10;
   *iivIter = 20;
   *iivIter = 30;


   list<int>::iterator LIter;
   cout << "The list L is: ( ";
   for ( LIter = L.begin ( ) ; LIter != L.end ( ); LIter++ )
      cout << *LIter << " ";
   cout << ")." << endl;

   insert_iterator<list<int> >::reference
        RefFirst = *(L.begin ( ));
   cout << "The first element in the list L is: "
        << RefFirst << "." << endl;
}
```

Output

```
The list L is: ( 10 20 30 ).
The first element in the list L is: 10.
```

Requirements

Header: <iterator>

See Also

Reference

insert_iterator Class

Standard Template Library

Other Resources

insert_iterator Members

insert_iterator Member Functions

For information about the functions in the insert_iterator class, see insert_iterator Members.

See Also

Reference

Standard Template Library

insert_iterator::insert_iterator

Constructs an **insert_iterator** that inserts an element into a specified position in a container.

```
insert_iterator(
   Container& _Cont,
   typename Container::iterator _It
);
```

Parameters

_Cont

The container into which the **insert_iterator** is to insert elements.

_It

The position for the insertion.

Remarks

All containers have the insert member function called by the **insert_iterator**. For associative containers the position parameter is merely a suggestion. The inserter function provides a convenient way to insert to values.

Example

```
// insert_iterator_insert_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>


int main( )
{
   using namespace std;
   int i;
   list <int>::iterator L_Iter;

   list<int> L;
   for (i = 1 ; i < 4 ; ++i )
   {
      L.push_back ( 10 * i );
   }

   cout << "The list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
      cout << *L_Iter << " ";
   cout << ")." << endl;

   // Using the member function to insert an element
   inserter ( L, L.begin ( ) ) = 2;

   // Alternatively, you may use the template version
   insert_iterator< list < int> > Iter(L, L.end ( ) );
   *Iter = 300;

   cout << "After the insertions, the list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++ )
      cout << *L_Iter << " ";
   cout << ")." << endl;
}
```

Output

```
The list L is:
```

```
 ( 10 20 30 ).
After the insertions, the list L is:
 ( 2 10 20 30 300 ).
```

Requirements

Header: <iterator>

See Also

Reference

insert_iterator Class

Standard Template Library

Other Resources

insert_iterator Members

insert_iterator Operators

For information about the operators in the insert_iterator class, see insert_iterator Class.

See Also

Reference

Standard Template Library

insert_iterator::operator*

Dereferences the insert iterator returning the element is addresses.

```
insert_iterator& operator*( );
```

Return Value

The member function returns the value of the element addressed.

Remarks

Used to implement the output iterator expression ***Iter** = **value**. If **Iter** is an iterator that addresses an element in a sequence, then ***Iter** = **value** replaces that element with value and does not change the total number of elements in the sequence.

Example

```
// insert_iterator_op_deref.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>


int main( )
{
   using namespace std;
   int i;
   list <int>::iterator L_Iter;

   list<int> L;
   for (i = 0 ; i < 4 ; ++i )
   {
      L.push_back ( 2 * i );
   }

   cout << "The original list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++ )
      cout << *L_Iter << " ";
   cout << ")." << endl;

   insert_iterator< list < int> > Iter(L, L.begin ( ) );
   *Iter = 10;
   *Iter = 20;
   *Iter = 30;

   cout << "After the insertions, the list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++ )
      cout << *L_Iter << " ";
   cout << ")." << endl;
}
```

Output

```
The original list L is:
 ( 0 2 4 6 ).
After the insertions, the list L is:
 ( 10 20 30 0 2 4 6 ).
```

Requirements

Header: <iterator>

See Also

Reference

insert_iterator Class
Standard Template Library
Other Resources
insert_iterator Members

insert_iterator::operator++

Increments the **insert_iterator** to the next location into which a value may be stored.

```
insert_iterator& operator++( );
insert_iterator& operator++( int );
```

Parameters

A **insert_iterator** addressing the next location into which a value may be stored.

Remarks

Both preincrementation and postincrementation operators return the same result.

Example

```
// insert_iterator_op_incr.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 5 ; ++i )
   {
      vec.push_back (  i );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is:\n ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++ )
      cout << *vIter << " ";
   cout << "))." << endl;

   insert_iterator<vector<int> > ii ( vec, vec.begin ( ) );
   *ii = 30;
   ii++;
   *ii = 40;
   ii++;
   *ii = 50;

   cout << "After the insertions, the vector vec becomes:\n ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++ )
      cout << *vIter << " ";
   cout << "))." << endl;
}
```

Output

```
The vector vec is:
 ( 1 2 3 4 ).
After the insertions, the vector vec becomes:
 ( 30 40 50 1 2 3 4 ).
```

Requirements

Header: <iterator>

See Also

Reference

insert_iterator Class

Standard Template Library

Other Resources

insert_iterator Members

insert_iterator::operator=

Assignment operator used to implement the output iterator expression **i = x*.

```
insert_iterator& operator=(
   typename Container::const_reference _Val
);
```

Parameters

_Val

The value to be assigned to the element.

Return Value

A reference to the element inserted into the container.

Remarks

The member function evaluates **Iter** = **container**. **insert**(**Iter**, *_Val*), then returns ***this**.

Example

```
// insert_iterator_op_assign.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
   list <int>::iterator L_Iter;

   list<int> L;
   for (i = 0 ; i < 4 ; ++i )
   {
      L.push_back ( 2 * i );
   }

   cout << "The original list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++ )
      cout << *L_Iter << " ";
   cout << ")." << endl;

   insert_iterator< list < int> > Iter(L, L.begin ( ) );
   *Iter = 10;
   *Iter = 20;
   *Iter = 30;

   cout << "After the insertions, the list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++ )
      cout << *L_Iter << " ";
   cout << ")." << endl;
}
```

Output

```
The original list L is:
 ( 0 2 4 6 ).
After the insertions, the list L is:
 ( 10 20 30 0 2 4 6 ).
```

Requirements

Header: <iterator>

See Also

Reference

insert_iterator Class

Standard Template Library

Other Resources

insert_iterator Members

input_iterator_tag Class

A class that provides a return type for **iterator_category** function that represents an input iterator.

```
struct input_iterator_tag {};
```

Remarks

The category tag classes are used as compile tags for algorithm selection. The template function needs to find the most specific category of its iterator argument so that it can use the most efficient algorithm at compile time. For every iterator of type **Iterator**, **iterator_traits**<**Iterator**>**::iterator_category** must be defined to be the most specific category tag that describes the iterator's behavior.

The type is the same as **iterator**<**Iter**>**::iterator_category** when **Iter** describes an object that can serve as an input iterator.

Example

See iterator_traits or random_access_iterator_tag for an example of how to use **iterator_tag**s.

Requirements

Header: <iterator>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<iterator> Members

istream_iterator Class

Describes an input iterator object. It extracts objects of class **Type** from an input stream, which it accesses through an object it stores, of type **pointer** to **basic_istream**<**CharType**, **Traits**>.

For a list of all members of this type, see istream_iterator Members.

```
template <
   class Type
      class CharType = char
   class Traits = char_traits<CharType>
   class Distance= ptrdiff_t
>
class istream_iterator :
public iterator<input_iterator_tag, Type, Distance, const Type *, const Type &>
```

Parameters

Type

The type of object to be extracted from the input stream.

CharType

The type that represents the character type for the **istream_iterator**. This argument is optional and the default value is **char**.

Traits

The type that represents the character type for the **istream_iterator**. This argument is optional and the default value is **char_traits**<*CharType*>.

Distance

A signed integral type that represents the difference type for the **istream_iterator**. This argument is optional and the default value is **ptrdiff_t**.

After constructing or incrementing an object of class istream_iterator with a nonnull stored pointer, the object attempts to extract and store an object of type **Type** from the associated input stream. If the extraction fails, the object effectively replaces the stored pointer with a null pointer, thus making an end-of-sequence indicator.

Requirements

Header: <iterator>

See Also

Reference

iterator Class

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

istream_iterator Members

<iterator> Members

istream_iterator Members

The following tables list the members of the istream_iterator Class. For a list of members inherited by the **istream_iterator** class, see iterator Class.

Reference

Constructors

istream_iterator	Constructs either an end-of-stream iterator as the default **istream_iterator** or a **istream_iterator** initialized to the iterator's stream type from which it reads.

Typedefs

char_type	A type that provides for the character type of the **istream_iterator**.
istream_type	A type that provides for the stream type of the **istream_iterator**.
traits_type	A type that provides for the character traits type of the **istream_iterator**.

Operators

operator*	The dereferencing operator returns the stored object of type **Type** addressed by the **istream_iterator**.
operator->	Returns the value of a member, if any.
operator++	Either extracts an incremented object from the input stream or copies the object before incrementing it and returns the copy.

See Also

Reference

istream_iterator Class
iterator Class
Thread Safety in the Standard C++ Library
Standard Template Library

istream_iterator Typedefs

For information about the typedefs in the istream_iterator class, see istream_iterator Members.

See Also

Reference

Standard Template Library

istream_iterator::char_type

A type that provides for the character type of the **istream_iterator**.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **Chartype**.

Example

```
// istream_iterator_char_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   typedef istream_iterator<int>::char_type CHT1;
   typedef istream_iterator<int>::traits_type CHTR1;

   // Standard iterator interface for reading
   // elements from the input stream:
   cout << "Enter integers separated by spaces & then\n"
        << " any character ( try example: '2 4 f' ): ";

   // istream_iterator for reading int stream
   istream_iterator<int, CHT1, CHTR1> intRead ( cin );

   // End-of-stream iterator
   istream_iterator<int, CHT1, CHTR1> EOFintRead;

   while ( intRead != EOFintRead )
   {
      cout << "Reading:  " << *intRead << endl;
      ++intRead;
   }
   cout << endl;
}
```

Input

```
2 4 f
```

Sample Output

```
Enter integers separated by spaces & then
 any character ( try example: '2 4 f' ): 2 4 f
Reading:  2
Reading:  4
```

Requirements

Header: <iterator>

See Also

Reference

istream_iterator Class

Standard Template Library

Other Resources

istream_iterator Members

istream_iterator::istream_type

A type that provides for the stream type of the **istream_iterator**.

```
typedef basic_istream<CharType, Traits> istream_type;
```

Remarks

The type is a synonym for **basic_istream**<**CharType**, **Traits**>.

Example

See istream_iterator for an example of how to declare and use **istream_type**.

Requirements

Header: <iterator>

See Also

Reference

istream_iterator Class

Standard Template Library

Other Resources

istream_iterator Members

istream_iterator::traits_type

A type that provides for the character traits type of the **istream_iterator**.

```
typedef Traits traits_type;
```

Remarks

The type is a synonym for the template parameter **Traits**.

Example

```
// istream_iterator_traits_type.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

int main( )
{
   using namespace std;

   typedef istream_iterator<int>::char_type CHT1;
   typedef istream_iterator<int>::traits_type CHTR1;

   // Standard iterator interface for reading
   // elements from the input stream:
   cout << "Enter integers separated by spaces & then\n"
        << " any character ( try example: '10 20 a' ): ";

   // istream_iterator for reading int stream
   istream_iterator<int, CHT1, CHTR1> intRead ( cin );

   // End-of-stream iterator
   istream_iterator<int, CHT1, CHTR1> EOFintRead;

   while ( intRead != EOFintRead )
   {
      cout << "Reading:  " << *intRead << endl;
      ++intRead;
   }
   cout << endl;
}
```

Input

```
10 20 a
```

Sample Output

```
Enter integers separated by spaces & then
 any character ( try example: '10 20 a' ): 10 20 a
Reading:  10
Reading:  20
```

Requirements

Header: <iterator>

See Also

Reference

istream_iterator Class

Standard Template Library

Other Resources

istream_iterator Members

istream_iterator Member Functions

For information about the functions in the istream_iterator class, see istream_iterator Members.

See Also

Reference

Standard Template Library

istream_iterator::istream_iterator

Constructs either an end-of-stream iterator as the default **istream_iterator** or a **istream_iterator** initialized to the iterator's stream type from which it reads.

```
istream_iterator( );
istream_iterator(
   istream_type& _Istr
);
```

Parameters

_Istr

The input stream to be read use to initialize the **istream_iterator**.

Remarks

The First constructor initializes the input stream pointer with a null pointer and creates an end-of-stream iterator. The second constructor initializes the input stream pointer with *&_Istr*, then attempts to extract and store an object of type **Type**.

The end-of-stream iterator can be use to test whether an **istream_iterator** has reached the end of a stream.

Example

```
// istream_iterator_istream_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <algorithm>
#include <iostream>

int main( )
{
   using namespace std;

   // Used in conjunction with copy algorithm
   // to put elements into a vector read from cin
   vector<int> vec ( 4 );
   vector <int>::iterator Iter;

   cout << "Enter 4 integers separated by spaces & then\n"
        << " a character ( try example: '2 4 6 8 a' ): ";
   istream_iterator<int> intvecRead ( cin );

   // Default constructor will test equal to end of stream
   // for delimiting source range of vecor
   copy ( intvecRead , istream_iterator<int>( ) , vec.begin ( ) );
   cin.clear ( );

   cout << "vec = ";
   for ( Iter = vec.begin( ) ; Iter != vec.end( ) ; Iter++ )
      cout << *Iter << " "; cout << endl;
}
```

Input

```
2 4 6 8 a
```

Sample Output

```
Enter 4 integers separated by spaces & then
 a character ( try example: '2 4 6 8 a' ): 2 4 6 8 a
vec = 2 4 6 8
```

Requirements

Header: <iterator>

See Also

Reference

istream_iterator Class

Standard Template Library

Other Resources

istream_iterator Members

istream_iterator Operators

For information about the operators in istream_iterator class, see istream_iterator Class.

See Also

Reference

Standard Template Library

istream_iterator::operator*

The dereferencing operator returns the stored object of type **Type** addressed by the **istream_iterator**.

```
const Type& operator*( ) const;
```

Return Value

The stored object of type **Type**.

Example

```
// istream_iterator_operator.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <algorithm>
#include <iostream>

int main( )
{
   using namespace std;

   cout << "Enter integers separated by spaces & then\n"
        << " a character ( try example: '2 4 6 8 a' ): ";

   // istream_iterator from stream cin
   istream_iterator<int> intRead ( cin );

   // End-of-stream iterator
   istream_iterator<int> EOFintRead;

   while ( intRead != EOFintRead )
   {
      cout << "Reading:  " << *intRead << endl;
      ++intRead;
   }
   cout << endl;
}
```

Input

```
2 4 6 8 a
```

Sample Output

```
Enter integers separated by spaces & then
 a character ( try example: '2 4 6 8 a' ): 2 4 6 8 a
Reading:  2
Reading:  4
Reading:  6
Reading:  8
```

Requirements

Header: <iterator>

See Also

Reference

istream_iterator Class

Standard Template Library

Other Resources

istream_iterator Members

istream_iterator::operator->

Returns the value of a member, if any.

```
const Type* operator->( ) const;
```

Return Value

The value of a member, if any.

Remarks

i -> is equivalent to (**i*).*m*

The operator returns **&**this**.

Example

```
// istream_iterator_operator_vm.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>
#include <complex>

using namespace std;

int main( )
{
   cout << "Enter complex numbers separated by spaces & then\n"
        << " a character pair ( try example: '(1,2) (3,4) (a,b)' ): ";

   // istream_iterator from stream cin
   istream_iterator< complex<double> > intRead ( cin );

   // End-of-stream iterator
   istream_iterator<complex<double> > EOFintRead;

   while ( intRead != EOFintRead )
   {
      cout << "Reading the real part: " << intRead ->real( ) << endl;
      cout << "Reading the imaginary part: " << intRead ->imag( ) << endl;
      ++intRead;
   }
   cout << endl;
}
```

Input

```
(1,2) (3,4) (a,b)
```

Sample Output

```
Enter complex numbers separated by spaces & then
 a character pair ( try example: '(1,2) (3,4) (a,b)' ): (1,2) (3,4) (a,b)
Reading the real part: 1
Reading the imaginary part: 2
Reading the real part: 3
Reading the imaginary part: 4
```

Requirements

Header: <iterator>

See Also

Reference

istream_iterator Class

istream_iterator::operator++

Either extracts an incremented object from the input stream or copies the object before incrementing it and returns the copy.

```
istream_iterator<Type, CharType, Traits, Distance>& operator++( );
istream_iterator<Type, CharType, Traits, Distance> operator++( int );
```

Return Value

The first member operator returns a reference to the incremented object of type **Type** extracted from the input stream and the second member function returns a copy of the object.

Example

```
// istream_iterator_operator_incr.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <algorithm>
#include <iostream>

int main( )
{
   using namespace std;

   cout << "Enter integers separated by spaces & then\n"
        << " a character ( try example: '2 4 6 8 a' ): ";

   // istream_iterator from stream cin
   istream_iterator<int> intRead ( cin );

   // End-of-stream iterator
   istream_iterator<int> EOFintRead;

   while ( intRead != EOFintRead )
   {
      cout << "Reading:  " << *intRead << endl;
      ++intRead;
   }
   cout << endl;
}
```

Input

```
2 4 6 8 a
```

Sample Output

```
Enter integers separated by spaces & then
 a character ( try example: '2 4 6 8 a' ): 2 4 6 8 a
Reading:  2
Reading:  4
Reading:  6
Reading:  8
```

Requirements

Header: <iterator>

See Also

Reference

istream_iterator Class

Standard Template Library

Other Resources

istream_iterator Members

istreambuf_iterator Class

The template class istreambuf_iterator describes an input iterator object that extracts character elements from an input stream buffer, which it accesses through an object it stores, of type pointer to **basic_streambuf**<**CharType**, **Traits**>.

For a list of all members of this type, see istreambuf_iterator Members.

```
template <
      class CharType
   class Traits = char_traits<CharType>
>
class istreambuf_iterator
: public iterator<input_iterator_tag, CharType, typename Traits::off_type, CharType *, Char
Type&>
```

Parameters

CharType

The type that represents the character type for the istreambuf_iterator.

Traits

The type that represents the character type for the istreambuf_iterator. This argument is optional and the default value is **char_traits**<*CharType*>.

Remarks

The ostreambuf_iterator class must satisfy the requirements for an input iterator.

After constructing or incrementing an object of class istreambuf_iterator with a non-null stored pointer, the object effectively attempts to extract and store an object of type **CharType** from the associated input stream. The extraction may be delayed, however, until the object is actually dereferenced or copied. If the extraction fails, the object effectively replaces the stored pointer with a null pointer, thus making an end-of-sequence indicator.

Requirements

Header: <iterator>

See Also

Reference

iterator Class

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

istreambuf_iterator Members

<iterator> Members

istreambuf_iterator Members

The following tables list the members of the istreambuf_iterator Class. For a list of members inherited by the **istreambuf_iterator** class, see iterator Class.

Reference

Constructors

istreambuf_iterator	Constructs an **istreambuf_iterator** that is initialized to read characters from the input stream.

Typedefs

char_type	A type that provides for the character type of the **ostreambuf_iterator**.
int_type	A type that provides an integer type for an **istreambuf_iterator**.
istream_type	A type that provides for the stream type of the **istream_iterator**.
streambuf_type	A type that provides for the stream type of the **istreambuf_iterator**.
traits_type	A type that provides for the character traits type of the **istream_iterator**.

Member Functions

equal	Tests for equality between two input stream buffer iterators.

Operators

operator*	The dereferencing operator returns the next character in the stream.
operator++	Either returns the next character from the input stream or copies the object before incrementing it and returns the copy.
operator->	Returns the value of a member, if any.

See Also

Reference

istreambuf_iterator Class
iterator Class
Thread Safety in the Standard C++ Library
Standard Template Library

istreambuf_iterator Typedefs

For information about the typedefs in the istreambuf_iterator class, see istreambuf_iterator Members.

See Also

Reference

Standard Template Library

istreambuf_iterator::char_type

A type that provides for the character type of the **ostreambuf_iterator**.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

Example

```
// istreambuf_iterator_char_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>
#include <algorithm>

int main( )
{
   using namespace std;

   typedef istreambuf_iterator<char>::char_type CHT1;
   typedef istreambuf_iterator<char>::traits_type CHTR1;

   cout << "(Try the example: 'So many dots to be done'\n"
        << " then an Enter key to insert into the output,\n"
        << " & use a ctrl-Z Enter key combination to exit): ";

   // istreambuf_iterator for input stream
   istreambuf_iterator< CHT1, CHTR1> charInBuf ( cin );
   ostreambuf_iterator<char> charOut ( cout );

   // Used in conjunction with replace_copy algorithm
   // to insert into output stream and replace spaces
   // with dot-separators
   replace_copy ( charInBuf , istreambuf_iterator<char>( ),
        charOut , ' ' , '.' );
}
```

Input

```
So many dots to be done
```

Sample Output

```
(Try the example: 'So many dots to be done'
 then an Enter key to insert into the output,
 & use a ctrl-Z Enter key combination to exit): So many dots to be done
So.many.dots.to.be.done
^Z
```

Requirements

Header: <iterator>

See Also

Reference

istreambuf_iterator Class

Standard Template Library

Other Resources

istreambuf_iterator Members

istreambuf_iterator::int_type

A type that provides an integer type for an **istreambuf_iterator**.

```
typedef typename Traits::int_type int_type;
```

Remarks

The type is a synonym for **Traits::int_type**.

Example

```
// istreambuf_iterator_int_type.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

int main( )
{
   using namespace std;
   istreambuf_iterator<char>::int_type inttype1 = 100;
   cout << "The inttype1 = " << inttype1 << "." << endl;
}
```

Output

```
The inttype1 = 100.
```

Requirements

Header: <iterator>

See Also

Reference

istreambuf_iterator Class

Standard Template Library

Other Resources

istreambuf_iterator Members

istreambuf_iterator::istream_type

A type that provides for the stream type of the **istreambuf_iterator**.

```
typedef basic_istream<CharType, Traits> istream_type;
```

Remarks

The type is a synonym for **basic_istream**<**CharType**, **Traits**>.

Example

See istreambuf_iterator for an example of how to declare and use **istream_type**.

Requirements

Header: <iterator>

See Also

Reference

istreambuf_iterator Class

Standard Template Library

Other Resources

istreambuf_iterator Members

istreambuf_iterator::streambuf_type

A type that provides for the stream type of the istreambuf_iterator.

```
typedef basic_streambuf<CharType, Traits> streambuf_type;
```

Remarks

The type is a synonym for **basic_streambuf**<**CharType**, **Traits**>.

Example

See istreambuf_iterator for an example of how to declare and use **istreambuf_type**.

Requirements

Header: <iterator>

See Also

Reference

istreambuf_iterator Class

Standard Template Library

Other Resources

istreambuf_iterator Members

istreambuf_iterator::traits_type

A type that provides for the character traits type of the **istream_iterator**.

```
typedef Traits traits_type;
```

Remarks

The type is a synonym for the template parameter **Traits**.

Example

```
// istreambuf_iterator_traits_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>
#include <algorithm>

int main( )
{
   using namespace std;

   typedef istreambuf_iterator<char>::char_type CHT1;
   typedef istreambuf_iterator<char>::traits_type CHTR1;

   cout << "(Try the example: 'So many dots to be done'\n"
        << " then an Enter key to insert into the output,\n"
        << " & use a ctrl-Z Enter key combination to exit): ";

   // istreambuf_iterator for input stream
   istreambuf_iterator< CHT1, CHTR1> charInBuf ( cin );
   ostreambuf_iterator<char> charOut ( cout );

   // Used in conjunction with replace_copy algorithm
   // to insert into output stream and replace spaces
   // with dot-separators
   replace_copy ( charInBuf , istreambuf_iterator<char>( ),
        charOut , ' ' , '.' );
}
```

Input

```
So many dots to be done
```

Sample Output

```
(Try the example: 'So many dots to be done'
 then an Enter key to insert into the output,
 & use a ctrl-Z Enter key combination to exit): So many dots to be done
So.many.dots.to.be.done
^Z
```

Requirements

Header: <iterator>

See Also

Reference

istreambuf_iterator Class

Standard Template Library

Other Resources

istreambuf_iterator Members

istreambuf_iterator Member Functions

For information about the functions in the istreambuf_iterator class, see istreambuf_iterator Members.

See Also

Reference

Standard Template Library

istreambuf_iterator::equal

Tests for equivalence between two input stream buffer iterators.

```
bool equal(
   const istreambuf_iterator& _Right
) const;
```

Parameters

_Right

The iterator for which to check for equality.

Return Value

true if both **istreambuf_iterator**s are end-of-stream iterators or if neither is an end-of-stream iterator; otherwise **false**.

Remarks

A range is defined by the **istreambuf_iterator** to the current position and the end-of-stream iterator, but since all non-end-of stream iterators are equivalent under the **equal** member function, it is not possible to define any subranges using **istreambuf_iterator**s. The **==** and **!=** operators have the same semantics.

Example

```
// istreambuf_iterator_equal.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

int main( )
{
   using namespace std;

   cout << "(Try the example: 'Hello world!'\n"
        << " then an Enter key to insert into the output,\n"
        << " & use a ctrl-Z Enter key combination to exit): ";

   istreambuf_iterator<char> charReadIn1 ( cin );
   istreambuf_iterator<char> charReadIn2 ( cin );

   bool b1 = charReadIn1.equal ( charReadIn2 );

   if (b1)
      cout << "The iterators are equal." << endl;
   else
      cout << "The iterators are not equal." << endl;
}
```

Input

```
Hello world!
```

Sample Output

```
(Try the example: 'Hello world!'
 then an Enter key to insert into the output,
 & use a ctrl-Z Enter key combination to exit): Hello world!
The iterators are equal.
```

Requirements

Header: <iterator>

See Also

Reference

istreambuf_iterator Class
Standard Template Library
Other Resources
istreambuf_iterator Members

istreambuf_iterator::istreambuf_iterator

Constructs an istreambuf_iterator that is initialized to read characters from the input stream.

```
istreambuf_iterator(
   streambuf_type* _Strbuf = 0
) throw( );
istreambuf_iterator(
   istream_type& _Istr
) throw( );
```

Parameters

_Strbuf

The input stream buffer to which the **istreambuf_iterator** is being attached.

_Istr

The input stream to which the **istreambuf_iterator** is being attached.

Remarks

The first constructor initializes the input stream-buffer pointer with *_Strbuf*. The second constructor initializes the input stream-buffer pointer with *_Istr*.**rdbuf**, and then eventually attempts to extract and store an object of type **CharType**.

Example

```
// istreambuf_iterator_istreambuf_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <algorithm>
#include <iostream>

int main( )
{
   using namespace std;

   // Following declarations will not compile:
   istreambuf_iterator<char>::istream_type &istrm = cin;
   istreambuf_iterator<char>::streambuf_type *strmbf = cin.rdbuf( );

   cout << "(Try the example: 'Oh what a world!'\n"
      << " then an Enter key to insert into the output,\n"
      << " & use a ctrl-Z Enter key combination to exit): ";
   istreambuf_iterator<char> charReadIn ( cin );
   ostreambuf_iterator<char> charOut ( cout );

   // Used in conjunction with replace_copy algorithm
   // to insert into output stream and replace spaces
   // with hyphen-separators
   replace_copy ( charReadIn , istreambuf_iterator<char>( ),
      charOut , ' ' , '-' );
}
```

Input

```
Oh what a world!
```

Sample Output

```
(Try the example: 'Oh what a world!'
 then an Enter key to insert into the output,
 & use a ctrl-Z Enter key combination to exit): Oh what a world!
Oh-what-a-world!
^Z
```

Requirements

Header: <iterator>

See Also

Reference

istreambuf_iterator Class

Standard Template Library

Other Resources

istreambuf_iterator Members

istreambuf_iterator Operators

For information about the operators in istreambuf_iterator class, see istreambuf_iterator Class.

See Also

Reference

Standard Template Library

istreambuf_iterator::operator*

The dereferencing operator returns the next character in the stream.

```
const CharType& operator*( ) const;
```

Return Value

A reference to the stored object of type **CharType**.

Example

```
// istreambuf_iterator_operator_deref.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

int main( )
{
   using namespace std;

   cout << "Type string of characters & enter to output it,\n"
      << " with stream buffer iterators,(try: 'I'll be back.')\n"
      << " repeat as many times as desired,\n"
      << " then keystroke ctrl-Z Enter to exit program: ";
   istreambuf_iterator<char> inpos ( cin );
   istreambuf_iterator<char> endpos;
   ostreambuf_iterator<char> outpos ( cout );
   while ( inpos != endpos )
   {
      *outpos = *inpos;   //Put value of outpos equal to inpos
      ++inpos;
      ++outpos;
   }
}
```

Input

```
I'll be back.
```

Sample Output

```
Type string of characters & enter to output it,
 with stream buffer iterators,(try: 'I'll be back.')
 repeat as many times as desired,
 then keystroke ctrl-Z Enter to exit program: I'll be back.
I'll be back.
^Z
```

Requirements

Header: <iterator>

See Also

Reference

istreambuf_iterator Class

Standard Template Library

Other Resources

istreambuf_iterator Members

istreambuf_iterator::operator++

Either returns the next character from the input stream or copies the object before incrementing it and returns the copy.

```
istreambuf_iterator& operator++( );
istreambuf_iterator operator++( int );
```

Return Value

An **istreambuf_iterator** or a reference to an **istreambuf_iterator**.

Remarks

The first operator eventually attempts to extract and store an object of type **CharType** from the associated input stream. The second operator makes a copy of the object, increments the object, and then returns the copy.

Example

```
// istreambuf_iterator_operator_incr.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

int main( )
{
   using namespace std;

   cout << "Type string of characters & enter to output it,\n"
      << " with stream buffer iterators,(try: 'I'll be back.')\n"
      << " repeat as many times as desired,\n"
      << " then keystroke ctrl-Z Enter to exit program: ";
   istreambuf_iterator<char> inpos ( cin );
   istreambuf_iterator<char> endpos;
   ostreambuf_iterator<char> outpos ( cout );
   while ( inpos != endpos )
   {
      *outpos = *inpos;
      ++inpos;   //Increment istreambuf_iterator
      ++outpos;
   }
}
```

Input

```
I'll be back.
```

Sample Output

```
Type string of characters & enter to output it,
 with stream buffer iterators,(try: 'I'll be back.')
 repeat as many times as desired,
 then keystroke ctrl-Z Enter to exit program: I'll be back.
I'll be back.
^Z
```

Requirements

Header: <iterator>

See Also

Reference

istreambuf_iterator Class

Standard Template Library

Other Resources

istreambuf_iterator Members

istreambuf_iterator::operator->

Returns the value of a member, if any.

```
const Elem *operator->( ) const;
```

Return Value

The operator returns **&**this**.

Requirements

Header: <iterator>

See Also

Reference

istreambuf_iterator Class

Standard Template Library

Other Resources

istreambuf_iterator Members

iterator Class

An empty base class used to ensure that a user-defined iterator class works properly with **iterator_trait**s.

```
template<class Category, class Type, class Distance = ptrdiff_t
   class Pointer = Type*, class Reference = Type&>
   struct iterator {
   typedef Category iterator_category;
   typedef Type value_type;
   typedef Distance difference_type;
   typedef Pointer pointer;
   typedef Reference reference;
   };
```

Remarks

The template class serves as a base type for all iterators. It defines the member types

- **iterator_category** (a synonym for the template parameter **Category**).
- **value_type** (a synonym for the template parameter **Type**).
- **difference_type** (a synonym for the template parameter **Distance**).
- **pointer** (a synonym for the template parameter **Pointer**).
- **reference** (a synonym for the template parameter **Reference**).

Note that **value_type** should not be a constant type even if **pointer** points at an object of const **Type** and reference designates an object of const **Type**.

Example

See iterator_traits for an example of how to declare and use the types in the iterator base class.

Requirements

Header: <iterator>

See Also

Reference

<iterator>

Thread Safety in the Standard C++ Library

Standard Template Library

iterator_traits Class

A template helper class used to specify all the critical type definitions that an iterator should have.

```
template<class Iterator>
   struct iterator_traits {
   typedef typename Iterator::iterator_category iterator_category;
   typedef typename Iterator::value_type value_type;
   typedef typename Iterator::difference_type difference_type;
   typedef typename Iterator::pointer pointer;
   typedef typename Iterator::reference reference;
   };
template<class Type>
   struct iterator_traits<Type*> {
   typedef random_access_iterator_tag iterator_category;
   typedef Type value_type;
   typedef ptrdiff_t difference_type;
   typedef Type *pointer;
   typedef Type& reference;
   };
template<class Type>
   struct iterator_traits<const Type*> {
   typedef random_access_iterator_tag iterator_category;
   typedef Type value_type;
   typedef ptrdiff_t difference_type;
   typedef const Type *pointer;
   typedef const Type& reference;
   };
```

Remarks

The template class defines the member types

- **iterator_category**: a synonym for **Iterator::iterator_category**.
- **value_type**: a synonym for **Iterator::value_type**.
- **difference_type**: a synonym for **Iterator::difference_type**.
- **pointer**: a synonym for **Iterator::pointer**.
- **reference**: a synonym for **Iterator::reference**.

The partial specializations determine the critical types associated with an object pointer of type **Type *** or const **Type ***.

In this implementation you can also use several template functions that do not make use of partial specialization:

```
template<class Category, class Type, class Diff>
C _Iter_cat(const iterator<Category, Ty, Diff>&);
template<class Ty>
    random_access_iterator_tag _Iter_cat(const Ty *);

template<class Category, class Ty, class Diff>
Ty *_Val_type(const iterator<Category, Ty, Diff>&);
template<class Ty>
    Ty *_Val_type(const Ty *);

template<class Category, class Ty, class Diff>
Diff *_Dist_type(const iterator<Category, Ty, Diff>&);
template<class Ty>
    ptrdiff_t *_Dist_type(const Ty *);
```

which determine several of the same types more indirectly. You use these functions as arguments on a function call. Their sole purpose is to supply a useful template class parameter to the called function.

Example

```
// iterator_traits.cpp
// compile with: /EHsc
#include <iostream>
#include <iterator>
#include <vector>
#include <list>

using namespace std;

template< class it >
void
function( it i1, it i2 )
{
   iterator_traits<it>::iterator_category cat;
   cout << typeid( cat ).name( ) << endl;
   while ( i1 != i2 )
   {
      iterator_traits<it>::value_type x;
      x = *i1;
      cout << x << " ";
      i1++;
   };
   cout << endl;
};

int main( )
{
   vector<char> vc( 10,'a' );
   list<int> li( 10 );
   function( vc.begin( ), vc.end( ) );
   function( li.begin( ), li.end( ) );
}
```

Output

```
struct std::random_access_iterator_tag
a a a a a a a a a a
struct std::bidirectional_iterator_tag
0 0 0 0 0 0 0 0 0 0
```

Requirements

Header: <iterator>

See Also

Reference

<iterator>

Thread Safety in the Standard C++ Library

Standard Template Library

ostream_iterator Class

The template class ostream_iterator describes an output iterator object that writes successive elements onto the output stream with the extraction **operator <<**.

For a list of all members of this type, see ostream_iterator Members.

```
template <
   class Type
      class CharType = char
   class Traits = char_traits<CharType>
>
class ostream_iterator
```

Parameters

Type

The type of object to be inserted into the output stream.

CharType

The type that represents the character type for the **ostream_iterator**. This argument is optional and the default value is **char**.

Traits

The type that represents the character type for the **ostream_iterator**. This argument is optional and the default value is **char_traits**<*CharType*>.

The ostream_iterator class must satisfy the requirements for an output iterator. Algorithms can be written directly to output streams using an **ostream_iterator**.

Requirements

Header: <iterator>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

ostream_iterator Members

<iterator> Members

ostream_iterator Members

Reference

Constructors

ostream_iterator	Constructs an **ostream_iterator** that is initialized and delimited to write to the output stream.

Typedefs

char_type	A type that provides for the character type of the **ostream_iterator**.
ostream_type	A type that provides for the stream type of the **ostream_iterator**.
traits_type	A type that provides for the character traits type of the **ostream_iterator**.

Operators

operator*	Dereferencing operator used to implement the output iterator expression *i = x.
operator++	A nonfunctional increment operator that returns an **ostream_iterator** to the same object it addressed before the operation was called.
operator=	Assignment operator used to implement the output iterator expression *i = x for writing to an output stream.

See Also

Reference

ostream_iterator Class
Thread Safety in the Standard C++ Library
Standard Template Library

ostream_iterator Typedefs

For information about the typedefs in the ostream_iterator class, see ostream_iterator Members.

See Also

Reference

Standard Template Library

ostream_iterator::char_type

A type that provides for the character type of the iterator.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

Example

```
// ostream_iterator_char_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   typedef ostream_iterator<int>::char_type CHT1;
   typedef ostream_iterator<int>::traits_type CHTR1;

   // ostream_iterator for stream cout
   // with new line delimiter:
    ostream_iterator<int, CHT1, CHTR1> intOut ( cout , "\n" );

   // Standard iterator interface for writing
   // elements to the output stream:
   cout << "The integers written to the output stream\n"
        << "by intOut are:" << endl;
   *intOut = 10;
   *intOut = 20;
   *intOut = 30;
}
```

Output

```
The integers written to the output stream
by intOut are:
10
20
30
```

Requirements

Header: <iterator>

See Also

Reference

ostream_iterator Class

Standard Template Library

Other Resources

ostream_iterator Members

ostream_iterator::ostream_type

A type that provides for the stream type of the iterator.

```
typedef basic_ostream<CharType, Traits> ostream_type;
```

Remarks

The type is a synonym for basic_ostream<*CharType*, *Traits*>, a stream class of the iostream hierarchy that defines objects that can be used for writing.

Example

See ostream_iterator for an example of how to declare and use **ostream_type**.

Requirements

Header: <iterator>

See Also

Reference

ostream_iterator Class

Standard Template Library

Other Resources

ostream_iterator Members

ostream_iterator::traits_type

A type that provides for the character traits type of the iterator.

```
typedef Traits traits_type;
```

Remarks

The type is a synonym for the template parameter **Traits**.

Example

```
// ostream_iterator_traits_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // The following not OK, but are just the default values:
   typedef ostream_iterator<int>::char_type CHT1;
   typedef ostream_iterator<int>::traits_type CHTR1;

   // ostream_iterator for stream cout
   // with new line delimiter:
    ostream_iterator<int, CHT1, CHTR1> intOut ( cout , "\n" );

   // Standard iterator interface for writing
   // elements to the output stream:
   cout << "The integers written to output stream\n"
        << "by intOut are:" << endl;
   *intOut = 1;
   *intOut = 10;
   *intOut = 100;
}
```

Output

```
The integers written to output stream
by intOut are:
1
10
100
```

Requirements

Header: <iterator>

See Also

Reference

ostream_iterator Class

Standard Template Library

Other Resources

ostream_iterator Members

ostream_iterator Member Functions

For information about the functions in the ostream_iterator class, see ostream_iterator Members.

See Also

Reference

Standard Template Library

ostream_iterator::ostream_iterator

Constructs an **ostream_iterator** that is initialized and delimited to write to the output stream.

```
ostream_iterator(
   ostream_type& _Ostr
);
ostream_iterator(
   ostream_type& _Ostr,
   const CharType* _Delimiter
);
```

Parameters

_Ostr

The output stream of type ostream_iterator::ostream_type to be iterated over.

_Delimiter

The delimiter that is inserted into the output stream between values.

Remarks

The first constructor initializes the output stream pointer with *&_Ostr*. The delimiter string pointer designates an empty string.

The second constructor initializes the output stream pointer with *&_Ostr* and the delimiter string pointer with *_Delimiter*.

Example

```
// ostream_iterator_ostream_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // ostream_iterator for stream cout
   ostream_iterator<int> intOut ( cout , "\n" );
   *intOut = 10;
   intOut++;
   *intOut = 20;
   intOut++;

   int i;
   vector<int> vec;
   for ( i = 1 ; i < 7 ; ++i )
   {
      vec.push_back (  i );
   }

   // Write elements to standard output stream
   cout << "Elements output without delimiter: ";
   copy ( vec.begin ( ), vec.end ( ),
          ostream_iterator<int> ( cout ) );
   cout << endl;

   // Write elements with delimiter " : " to output stream
   cout << "Elements output with delimiter: ";
   copy ( vec.begin ( ), vec.end ( ),
          ostream_iterator<int> ( cout, " : " ) );
   cout << endl;
}
```

Output

```
10
20
Elements output without delimiter: 123456
Elements output with delimiter: 1 : 2 : 3 : 4 : 5 : 6 :
```

Requirements

Header: <iterator>

See Also

Reference

ostream_iterator Class

Standard Template Library

Other Resources

ostream_iterator Members

ostream_iterator Operators

For information about the operators in the ostream_iterator class, see ostream_iterator Class.

See Also

Reference

Standard Template Library

ostream_iterator::operator*

Dereferencing operator used to implement the output iterator expression **ii = x*.

```
ostream_iterator<Type, CharType, Traits>& operator*( );
```

Return Value

A reference to the **ostream_iterator**.

Remarks

The requirements for an output iterator that the **ostream_iterator** must satisfy require only the expression **ii = t* be valid and says nothing about the **operator** or the **operator=** on their own. The member operator in this implementation returns ***this**.

Example

```
// ostream_iterator_op_deref.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // ostream_iterator for stream cout
   // with new line delimiter
   ostream_iterator<int> intOut ( cout , "\n" );

   // Standard iterator interface for writing
   // elements to the output stream
   cout << "Elements written to output stream:" << endl;
   *intOut = 10;
   intOut++;      // No effect on iterator position
   *intOut = 20;
   *intOut = 30;
}
```

Output

```
Elements written to output stream:
10
20
30
```

Requirements

Header: <iterator>

See Also

Reference

ostream_iterator Class

Standard Template Library

Other Resources

ostream_iterator Members

ostream_iterator::operator++

A nonfunctional increment operator that returns an **ostream_iterator** to the same object it addressed before the operation was called.

```
ostream_iterator<Type, CharType, Traits>& operator++( );
ostream_iterator<Type, CharType, Traits>& operator++( int );
```

Return Value

A reference to the **ostream_iterator**.

Remarks

These member operators both return ***this**.

Example

```
// ostream_iterator_op_incr.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // ostream_iterator for stream cout
   // with new line delimiter
   ostream_iterator<int> intOut ( cout , "\n" );

   // standard iterator interface for writing
   // elements to the output stream
   cout << "Elements written to output stream:" << endl;
   *intOut = 10;
   intOut++;      // No effect on iterator position
   *intOut = 20;
   *intOut = 30;
}
```

Output

```
Elements written to output stream:
10
20
30
```

Requirements

Header: <iterator>

See Also

Reference

ostream_iterator Class

Standard Template Library

Other Resources

ostream_iterator Members

ostream_iterator::operator=

Assignment operator used to implement the output_iterator expression *i = x for writing to an output stream.

```
ostream_iterator<Type, CharType, Traits>& operator=(
   const Type& _Val
);
```

Parameters

_Val

The value of the object of type *Type* to be inserted into the output stream.

Return Value

The operator inserts *_Val* into the output stream associated with the object, followed by the delimiter specified in the ostream_iterator constructor (if any), and then returns a reference to the **ostream_iterator**.

Remarks

The requirements for an output iterator that the **ostream_iterator** must satisfy require only the expression **ii = t* be valid and says nothing about the operator or the operator= on their own. This member operator returns ***this**.

Example

```
// ostream_iterator_op_assign.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // ostream_iterator for stream cout
   // with new line delimiter
   ostream_iterator<int> intOut ( cout , "\n" );

   // Standard iterator interface for writing
   // elements to the output stream
   cout << "Elements written to output stream:" << endl;
   *intOut = 10;
   intOut++;      // No effect on iterator position
   *intOut = 20;
   *intOut = 30;
}
```

Output

```
Elements written to output stream:
10
20
30
```

Requirements

Header: <iterator>

See Also

Reference

ostream_iterator Class

Standard Template Library

Other Resources

ostream_iterator Members

ostreambuf_iterator Class

The template class ostreambuf_iterator describes an output iterator object that writes successive character elements onto the output stream with the extraction **operator>>**. The **ostreambuf_iterator**s differ from those of the ostream_iterator Class in having characters instead of a generic type at the type of object being inserted into the output stream.

For a list of all members of this type, see ostreambuf_iterator Members.

```
template <
      class CharType = char
   class Traits = char_traits<CharType>
>
```

Parameters

CharType

The type that represents the character type for the ostreambuf_iterator. This argument is optional and the default value is **char**.

Traits

The type that represents the character type for the ostreambuf_iterator. This argument is optional and the default value is **char_traits**<*CharType*>.

Remarks

The ostreambuf_iterator class must satisfy the requirements for an output iterator. Algorithms can be written directly to output streams using an **ostreambuf_iterator**. The class provides a low-level stream iterator that allows access to the raw (unformatted) I/O stream in the form of characters and the ability to bypass the buffering and character translations associated with the high-level stream iterators.

Requirements

Header: <iterator>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

ostreambuf_iterator Members

<iterator> Members

ostreambuf_iterator Members

Reference

Constructors

ostreambuf_iterator	Constructs an **ostreambuf_iterator** that is initialized to write characters to the output stream.

Typedefs

char_type	A type that provides for the character type of the **ostreambuf_iterator**.
ostream_type	A type that provides for the stream type of the **ostream_iterator**.
streambuf_type	A type that provides for the stream type of the **ostreambuf_iterator**.
traits_type	A type that provides for the character traits type of the **ostream_iterator**.

Member Functions

failed	Tests for failure of an insertion into the output stream buffer.

Operators

operator*	Dereferencing operator used to implement the output iterator expression *i = x.
operator++	A nonfunctional increment operator that returns an **ostreambuf_iterator** to the same object it addressed before t he operation was called.
operator=	The operator inserts a character into the associated stream buffer.

See Also

Reference

ostreambuf_iterator Class
Thread Safety in the Standard C++ Library
Standard Template Library

ostreambuf_iterator Typedefs

For information about the typedefs in the ostreambuf_iterator class, see ostreambuf_iterator Members.

See Also

Reference

Standard Template Library

ostreambuf_iterator::char_type

A type that provides for the character type of the **ostreambuf_iterator**.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

Example

```
// ostreambuf_iterator_char_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   typedef ostreambuf_iterator<char>::char_type CHT1;
   typedef ostreambuf_iterator<char>::traits_type CHTR1;

   // ostreambuf_iterator for stream cout
   // with new line delimiter:
    ostreambuf_iterator< CHT1, CHTR1> charOutBuf ( cout );

   // Standard iterator interface for writing
   // elements to the output streambuf:
   cout << "The characters written to the output stream\n"
        << " by charOutBuf are: ";
   *charOutBuf = 'O';
   charOutBuf++;
   *charOutBuf = 'U';
   charOutBuf++;
   *charOutBuf = 'T';
   charOutBuf++;
   cout << "." << endl;
}
```

Output

```
The characters written to the output stream
 by charOutBuf are: OUT.
```

Requirements

Header: <iterator>

See Also

Reference

ostreambuf_iterator Class

Standard Template Library

Other Resources

ostreambuf_iterator Members

ostreambuf_iterator::ostream_type

A type that provides for the stream type of the **ostream_iterator**.

```
typedef basic_ostream<CharType, Traits> ostream_type;
```

Remarks

The type is a synonym for **basic_ostream**<**CharType**, **Traits**>

Example

See ostreambuf_iterator for an example of how to declare and use **ostream_type**.

Requirements

Header: <iterator>

See Also

Reference

ostreambuf_iterator Class

Standard Template Library

Other Resources

ostreambuf_iterator Members

ostreambuf_iterator::streambuf_type

A type that provides for the stream type of the **ostreambuf_iterator**.

```
typedef basic_streambuf<CharType, Traits> streambuf_type;
```

Remarks

The type is a synonym for **basic_streambuf**<**CharType**, **Traits**>, a stream class for I/O buffers that becomes **streambuf** when specialized to character type **char**.

Example

See ostreambuf_iterator for an example of how to declare and use **streambuf_type**.

Requirements

Header: <iterator>

See Also

Reference

ostreambuf_iterator Class

Standard Template Library

Other Resources

ostreambuf_iterator Members

ostreambuf_iterator::traits_type

A type that provides for the character traits type of the **ostream_iterator**.

```
typedef Traits traits_type;
```

Remarks

The type is a synonym for the template parameter **Traits**.

Example

```
// ostreambuf_iterator_traits_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   typedef ostreambuf_iterator<char>::char_type CHT1;
   typedef ostreambuf_iterator<char>::traits_type CHTR1;

   // ostreambuf_iterator for stream cout
   // with new line delimiter:
    ostreambuf_iterator< CHT1, CHTR1> charOutBuf ( cout );

   // Standard iterator interface for writing
   // elements to the output streambuf:
   cout << "The characters written to the output stream\n"
        << " by charOutBuf are: ";
   *charOutBuf = 'O';
   charOutBuf++;
   *charOutBuf = 'U';
   charOutBuf++;
   *charOutBuf = 'T';
   charOutBuf++;
   cout << "." << endl;
}
```

Output

```
The characters written to the output stream
 by charOutBuf are: OUT.
```

Requirements

Header: <iterator>

See Also

Reference

ostreambuf_iterator Class

Standard Template Library

Other Resources

ostreambuf_iterator Members

ostreambuf_iterator Member Functions

For information about the functions in the ostreambuf_iterator class, see ostreambuf_iterator Members.

See Also

Reference

Standard Template Library

ostreambuf_iterator::failed

Tests for failure of an insertion into the output stream buffer.

```
bool failed( ) const throw( );
```

Return Value

true if no insertion into the output stream buffer has failed earlier; otherwise **false**.

Remarks

The member function returns **true** if, in any prior use of member **operator=**, the call to **subf**_->**sputc** returned **eof**.

Example

```
// ostreambuf_iterator_failed.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // ostreambuf_iterator for stream cout
   ostreambuf_iterator<char> charOut ( cout );

   *charOut = 'a';
   charOut ++;
   *charOut  = 'b';
   charOut ++;
   *charOut = 'c';
   cout << " are characters output individually." << endl;

   bool b1 = charOut.failed ( );
   if (b1)
       cout << "At least one insertion failed." << endl;
   else
       cout << "No insertions failed." << endl;
}
```

Output

```
abc are characters output individually.
No insertions failed.
```

Requirements

Header: <iterator>

See Also

Reference

ostreambuf_iterator Class

Standard Template Library

Other Resources

ostreambuf_iterator Members

ostreambuf_iterator::ostreambuf_iterator

Constructs an **ostreambuf_iterator** that is initialized to write characters to the output stream.

```
ostreambuf_iterator(
   streambuf_type* _Strbuf
) throw( );
ostreambuf_iterator(
   ostream_type& _Ostr
) throw( );
```

Parameters

_Strbuf

The output streambuf object used to initialize the output stream-buffer pointer.

_Ostr

The output stream object used to initialize the output stream-buffer pointer.

Remarks

The first constructor initializes the output stream-buffer pointer with *_Strbuf*.

The second constructor initializes the output stream-buffer pointer with *_Ostr*.**rdbuf**. The stored pointer must not be a null pointer.

Example

```
// ostreambuf_iterator_ostreambuf_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // ostreambuf_iterator for stream cout
   ostreambuf_iterator<char> charOut ( cout );

   *charOut = 'O';
   charOut ++;
   *charOut  = 'U';
   charOut ++;
   *charOut = 'T';
   cout << " are characters output individually." << endl;

   ostreambuf_iterator<char> strOut ( cout );
   string str = "These characters are being written to the output stream.\n ";
   copy ( str.begin ( ), str. end ( ), strOut );
}
```

Output

```
OUT are characters output individually.
These characters are being written to the output stream.
```

Requirements

Header: <iterator>

See Also

Reference

ostreambuf_iterator Class

ostreambuf_iterator Operators

For information about the operators in the ostreambuf_iterator class, see ostreambuf_iterator Class.

See Also

Reference

Standard Template Library

ostreambuf_iterator::operator*

A nonfunctional dereferencing operator used to implement the output iterator expression *i = x.

```
ostreambuf_iterator& operator*( );
```

Return Value

The ostreambuf iterator object.

Remarks

This operator functions only in the output iterator expression *i = x to output characters to stream buffer. Applied to an ostreambuf iterator, it returns the iterator; ***iter** returns **iter**,

Example

```
// ostreambuf_iterator_op_deref.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // ostreambuf_iterator for stream cout
   // with new line delimiter
   ostreambuf_iterator<char> charOutBuf ( cout );

   // Standard iterator interface for writing
   // elements to the output stream
   cout << "Elements written to output stream:" << endl;
   *charOutBuf = 'O';
   charOutBuf++;   // no effect on iterator position
   *charOutBuf = 'U';
   *charOutBuf = 'T';
}
```

Output

```
Elements written to output stream:
OUT
```

Requirements

Header: <iterator>

See Also

Reference

ostreambuf_iterator Class

Standard Template Library

Other Resources

ostreambuf_iterator Members

ostreambuf_iterator::operator++

A nonfunctional increment operator that returns an ostream iterator to the same character it addressed before the operation was called.

```
ostreambuf_iterator& operator++( );
Implementation-defined operator++( int );
```

Return Value

A reference to the character originally addressed or to an implementation-defined object that is convertible to **ostreambuf_iterator**<**CharType**, **Traits**>.

Remarks

The operator is used to implement the output iterator expression *i = x.

Example

```
// ostreambuf_iterator_op_incr.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // ostreambuf_iterator for stream cout
   // with new line delimiter
   ostreambuf_iterator<char> charOutBuf ( cout );

   // Standard iterator interface for writing
   // elements to the output stream
   cout << "Elements written to output stream:" << endl;
   *charOutBuf = 'O';
   charOutBuf++;      // No effect on iterator position
   *charOutBuf = 'U';
   *charOutBuf = 'T';
}
```

Output

```
Elements written to output stream:
OUT
```

Requirements

Header: <iterator>

See Also

Reference

ostreambuf_iterator Class

Standard Template Library

Other Resources

ostreambuf_iterator Members

ostreambuf_iterator::operator=

The operator inserts a character into the associated stream buffer.

```
ostreambuf_iterator& operator=(
   CharType _Char
);
```

Parameters

_Char

The character to be inserted into the stream buffer.

Return Value

A reference to the character inserted into the stream buffer.

Remarks

Assignment operator used to implement the output iterator expression *i = x for writing to an output stream.

Example

```
// ostreambuf_iterator_op_assign.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // ostreambuf_iterator for stream cout
   // with new line delimiter
   ostreambuf_iterator<char> charOutBuf ( cout );

   // Standard iterator interface for writing
   // elements to the output stream
   cout << "Elements written to output stream:" << endl;
   *charOutBuf = 'O';
   charOutBuf++;      // No effect on iterator position
   *charOutBuf = 'U';
   *charOutBuf = 'T';
}
```

Output

```
Elements written to output stream:
OUT
```

Requirements

Header: <iterator>

See Also

Reference

ostreambuf_iterator Class

Standard Template Library

Other Resources

ostreambuf_iterator Members

output_iterator_tag Class

A class that provides a return type for **iterator_category** function that represents an output iterator.

```
struct output_iterator_tag {};
```

Remarks

The category tag classes are used as compile tags for algorithm selection. The template function needs to find the most specific category of its iterator argument so that it can use the most efficient algorithm at compile time. For every iterator of type **Iterator**, **iterator_traits**<**Iterator**>**::iterator_category** must be defined to be the most specific category tag that describes the iterator's behavior.

The type is the same as **iterator**<**Iter**>**::iterator_category** when **Iter** describes an object that can serve as a output iterator.

This tag is not parameterized on the **value_type** or **difference_type** for the iterator, as with the other iterator tags, because output iterators do not have either a **value_type** or a **difference_type**.

Example

See iterator_traits or random_access_iterator_tag for an example of how to use **iterator_tag**s.

Requirements

Header: <iterator>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<iterator> Members

random_access_iterator_tag Class

A class that provides a return type for **iterator_category** function that represents a random-access iterator.

```
struct random_access_iterator_tag
   : public bidirectional_iterator_tag {};
```

Remarks

The category tag classes are used as compile tags for algorithm selection. The template function needs to find the most specific category of its iterator argument so that it can use the most efficient algorithm at compile time. For every iterator of type **Iterator**, **iterator_traits**<**Iterator**>**::iterator_category** must be defined to be the most specific category tag that describes the iterator's behavior.

The type is the same as **iterator**<**Iter**>**::iterator_category** when **Iter** describes an object that can serve as a random-access iterator.

Example

```
// iterator_rait.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>
#include <list>

using namespace std;

int main( )
{
   vector<int> vi;
   vector<char> vc;
   list<char> lc;
   iterator_traits<vector<int>:: iterator>::iterator_category cati;
   iterator_traits<vector<char>:: iterator>::iterator_category catc;
   iterator_traits<list<char>:: iterator>::iterator_category catlc;

   // These are both random-access iterators
   cout << "The type of iterator for vector<int> is "
       << "identified by the tag:\n "
       << typeid ( cati ).name( ) << endl;
   cout << "The type of iterator for vector<char> is "
       << "identified by the tag:\n "
       << typeid ( catc ).name( ) << endl;
   if ( typeid ( cati ) == typeid( catc ) )
      cout << "The iterators are the same." << endl << endl;
   else
      cout << "The iterators are not the same." << endl << endl;

   // But the list iterator is bidirectinal, not random access
   cout << "The type of iterator for list<char> is "
       << "identified by the tag:\n "
       << typeid (catlc).name( ) << endl;

   // cout << ( typeid ( vi.begin( ) ) == typeid( vc.begin( ) ) ) << endl;
   if ( typeid ( vi.begin( ) ) == typeid( vc.begin( ) ) )
      cout << "The iterators are the same." << endl;
   else
      cout << "The iterators are not the same." << endl;
   // A random-access iterator is a bidirectional iterator.
   cout << ( void* ) dynamic_cast< iterator_traits<list<char>:: iterator>
          ::iterator_category* > ( &catc ) << endl;
}
```

Sample Output

The following output is for x86.

```
The type of iterator for vector<int> is identified by the tag:
 struct std::random_access_iterator_tag
The type of iterator for vector<char> is identified by the tag:
 struct std::random_access_iterator_tag
The iterators are the same.

The type of iterator for list<char> is identified by the tag:
 struct std::bidirectional_iterator_tag
The iterators are not the same.
0012FF3B
```

Requirements

Header: <iterator>

See Also

Reference

bidirectional_iterator_tag Class

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<iterator> Members

reverse_iterator Class

The template class is an iterator adaptor that describes a reverse iterator object that behaves like a random-access or bidirectional iterator, only in reverse. It enables the backward traversal of a range.

For a list of all members of this type, see reverse_iterator Members.

```
template <class Iterator>
```

Parameters

Iterator

The type that represents the iterator to be adapted to operate in reverse.

Remarks

Existing Standard Template Library containers also define **reverse_iterator** and **const_reverse_iterator** types and have member functions **rbegin** and **rend** that return reverse iterators. These iterators have overwrite semantics. The **reverse_iterator** adaptor supplements this functionality as offers insert semantics and can also be used with streams.

The **reverse_iterator**s that require a bidirectional iterator must not call any of the member functions **operator+=**, **operator+**, **operator-=**, **operator-**, or **operator[]**, which may only be used with random-access iterators.

If the range of an iterator is [_*First*, _Last), where the square bracket on the left indicates the inclusion on _*First* and the parenthesis on the right indicates the inclusion of elements up to _*Left* but excluding _*Left* itself. The same elements are included in the reversed sequence [**rev** – _*First*, **rev** – _*Left*) so that if _*Left* is the one-past-the-end element in a sequence, then the first element **rev** – _*First* in the reversed sequence points to *(_*Left* – 1). The identity which relates all reverse iterators to their underlying iterators is:

&*(**reverse_iterator** (*i*)) == &*(*i* – 1).

In practice, this means that in the reversed sequence the reverse_iterator will refer to the element one position beyond (to the right of) the element that the iterator had referred to in the original sequence. So if an iterator addressed the element 6 in the sequence (2, 4, 6, 8), then the **reverse_iterator** will address the element 4 in the reversed sequence (8, 6, 4, 2).

Requirements

Header: <iterator>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

reverse_iterator Members

<iterator> Members

reverse_iterator Members

Reference

Constructors

reverse_iterator	Constructs a default **reverse_iterator** or a **reverse_iterator** from an underlying iterator.

Typedefs

difference_type	A type that provides the difference between two **reverse_iterator**s referring to elements within the same container.
iterator_type	A type that provides the underlying iterator for a **reverse_iterator**.
pointer	A type that provides a pointer to an element addressed by a **reverse_iterator**.
reference	A type that provides a reference to an element addressed by a **reverse_iterator**.

Member Functions

base	Recovers the underlying iterator from its **reverse_iterator**.

Operators

operator*	Returns the element that a **reverse_iterator** addresses.
operator+	Adds an offset to an iterator and returns the new **reverse_iterator** addressing the inserted element at the new offset position.
operator++	Increments the **reverse_iterator** to the next element.
operator+=	Adds a specified offset from a **reverse_iterator**.
operator-	Subtracts an offset from a **reverse_iterator** and returns a **reverse_iterator** addressing the element at the offset position.
operator--	Decrements the **reverse_iterator** to the previous element.
operator-=	Subtracts a specified offset from a **reverse_iterator**.
operator->	Returns a pointer to the element addressed by the **reverse_iterator**.
operator[]	Returns a reference to an element offset from the element addressed by a **reverse_iterator** by a specified number of positions.

See Also

Reference

reverse_iterator Class
Thread Safety in the Standard C++ Library
Standard Template Library

reverse_iterator Typedefs

For information about the typedefs in the reverse_iterator class, see reverse_iterator Members.

See Also

Reference

Standard Template Library

reverse_iterator::difference_type

A type that provides the difference between two **reverse_iterator**s referring to elements within the same container.

```
typedef typename iterator_traits<RandomIterator>::difference_type
   difference_type;
```

Remarks

The **reverse_iterator** difference type is the same as the iterator difference type.

The type is a synonym for the iterator trait typename **iterator_traits**<**RandomIterator**>**::pointer**.

Example

See reverse_iterator::operator[] for an example of how to declare and use **difference_type**.

Requirements

Header: <iterator>

See Also

Reference

reverse_iterator Class

Standard Template Library

Other Resources

reverse_iterator Members

reverse_iterator::iterator_type

A type that provides the underlying iterator for a **reverse_iterator**.

```
typedef Iterator iterator_type;
```

Remarks

The type is a synonym for the template parameter **Iterator**.

Example

See reverse_iterator::base for an example of how to declare and use **iterator_type**.

Requirements

Header: <iterator>

See Also

Reference

reverse_iterator Class

Standard Template Library

Other Resources

reverse_iterator Members

reverse_iterator::pointer

A type that provides a pointer to an element addressed by a **reverse_iterator**.

```
typedef typename iterator_traits<Iterator>::pointer
   pointer;
```

Remarks

The type is a synonym for the iterator trait typename **iterator_traits**<*Random-access iterator*>**::pointer**.

Example

```
// reverse_iterator_pointer.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <utility>
#include <iostream>

int main( )
{
   using namespace std;

   typedef vector<pair<int,int> > pVector;
   pVector vec;
   vec.push_back( pVector::value_type( 1,2 ) );
   vec.push_back( pVector::value_type( 3,4 ) );
   vec.push_back( pVector::value_type( 5,6 ) );

   pVector::iterator pvIter;
   cout << "The vector vec of integer pairs is:\n" << "( ";
   for ( pvIter = vec.begin ( ) ; pvIter != vec.end ( ); pvIter++)
      cout << "( " << pvIter -> first << ", " << pvIter -> second << ") ";
   cout << ")" << endl;

   pVector::reverse_iterator rpvIter;
   cout << "\nThe vector vec reversed is:\n" << "( ";
   for ( rpvIter = vec.rbegin( ) ; rpvIter != vec.rend( ); rpvIter++)
      cout << "( " << rpvIter -> first << ", " << rpvIter -> second << ") ";
   cout << ")" << endl;

   pVector::iterator pos = vec.begin ( );
   pos++;
   cout << "\nThe iterator pos points to:\n"
        << "( " << pos -> first << ", "
        << pos -> second << " )" << endl;

   pVector::reverse_iterator rpos (pos);
   cout << "\nThe iterator rpos points to:\n"
        << "( " << rpos -> first << ", "
        << rpos -> second << " )" << endl;
}
```

Output

```
The vector vec of integer pairs is:
( ( 1, 2) ( 3, 4) ( 5, 6) )

The vector vec reversed is:
( ( 5, 6) ( 3, 4) ( 1, 2) )

The iterator pos points to:
```

```
( 3, 4 )

The iterator rpos points to:
( 1, 2 )
```

Requirements

Header: <iterator>

See Also

Reference

reverse_iterator Class

Standard Template Library

Other Resources

reverse_iterator Members

reverse_iterator::reference

A type that provides a reference to an element addressed by a reverse_iterator.

```
typedef typename iterator_traits<Iterator>::reference
   reference;
```

Remarks

The type is a synonym for the iterator trait typename **iterator_traits**<*Random-access iterator*>**::reference**.

Example

See reverse_iterator::operator[] or reverse_iterator::operator* for examples of how to declare and use **reference**.

Requirements

Header: <iterator>

See Also

Reference

reverse_iterator Class

Standard Template Library

Other Resources

reverse_iterator Members

reverse_iterator Member Functions

For information about the functions in the reverse_iterator class, see reverse_iterator Members.

See Also

Reference

Standard Template Library

reverse_iterator::base

Recovers the underlying iterator from its **reverse_iterator**.

```
RandomIterator base( ) const;
```

Return Value

The iterator underlying the **reverse_iterator**.

Remarks

The identity that relates all reverse iterators to their underlying iterators is:

&*(**reverse_iterator** (*i*)) == &*(*i* – 1).

In practice, this means that in the reversed sequence the **reverse_iterator** will refer to the element one position beyond (to the right of) the element that the iterator had referred to in the original sequence. So if an iterator addressed the element 6 in the sequence (2, 4, 6, 8), then the **reverse_iterator** will address the element 4 in the reversed sequence (8, 6, 4, 2).

Example

```
// reverse_iterator_base.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for ( i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( 2 * i );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
      cout << *rvIter << " ";
   cout << ")." << endl;

   vector <int>::iterator pos, bpos;
   pos = find ( vec.begin ( ), vec.end ( ), 6 );
   cout << "The iterator pos points to: " << *pos << "." << endl;

   typedef reverse_iterator<vector<int>::iterator>::iterator_type it_vec_int_type;

   reverse_iterator<it_vec_int_type> rpos ( pos );
   cout << "The reverse_iterator rpos points to: " << *rpos
        << "." << endl;

   bpos = rpos.base ( );
   cout << "The iterator underlying rpos is bpos & it points to: "
        << *bpos << "." << endl;
```

```
}
```

Output

```
The vector vec is: ( 2 4 6 8 10 ).
The vector vec reversed is: ( 10 8 6 4 2 ).
The iterator pos points to: 6.
The reverse_iterator rpos points to: 4.
The iterator underlying rpos is bpos & it points to: 6.
```

Requirements

Header: <iterator>

See Also

Reference

reverse_iterator Class

Standard Template Library

Other Resources

reverse_iterator Members

reverse_iterator::reverse_iterator

Constructs a default **reverse_iterator** or a **reverse_iterator** from an underlying iterator.

```
reverse_iterator( );
explicit reverse_iterator(
   RandomIterator _Right
);
template<class Type>
   reverse_iterator(
      const reverse_iterator<Type>& _Right
      );
```

Parameters

_Right

The iterator that is to be adapted to a **reverse_iterator**.

Return Value

A default **reverse_iterator** or a **reverse_iterator** adapting an underlying iterator.

Remarks

The identity which relates all reverse iterators to their underlying iterators is:

&*(**reverse_iterator** (*i*)) == &*(*i* – 1).

In practice, this means that in the reversed sequence the reverse_iterator will refer to the element one position beyond (to the right of) the element that the iterator had referred to in the original sequence. So if an iterator addressed the element 6 in the sequence (2, 4, 6, 8), then the **reverse_iterator** will address the element 4 in the reversed sequence (8, 6, 4, 2).

Example

```
// reverse_iterator_reverse_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for ( i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( i );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
      cout << *rvIter << " ";
   cout << ")." << endl;

   vector <int>::iterator pos;
   pos = find ( vec.begin ( ), vec.end ( ), 4 );
```

```
    cout << "The iterator pos = " << *pos << "." << endl;

    vector <int>::reverse_iterator rpos ( pos );
    cout << "The reverse_iterator rpos = " << *rpos
         << "." << endl;
}
```

Output

```
The vector vec is: ( 1 2 3 4 5 ).
The vector vec reversed is: ( 5 4 3 2 1 ).
The iterator pos = 4.
The reverse_iterator rpos = 3.
```

Requirements

Header: <iterator>

See Also

Reference

reverse_iterator Class

Standard Template Library

Other Resources

reverse_iterator Members

reverse_iterator Operators

For information about the operators in reverse_iterator class, see reverse_iterator Class.

See Also

Reference

Standard Template Library

reverse_iterator::operator*

Returns the element that a reverse_iterator addresses.

```
reference operator*( ) const;
```

Return Value

The value of the elements addressed by the reverse_iterator.

Remarks

The operator returns *(**current** – 1).

Example

```
// reverse_iterator_op_ref.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( 2 * i );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++ )
      cout << *vIter << " ";
   cout << "))." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
      cout << *rvIter << " ";
   cout << "))." << endl;

   vector <int>::iterator pos, bpos;
   pos = find ( vec.begin ( ), vec.end ( ), 6 );

   // Declare a difference type for a parameter
   // declare a reference return type
   reverse_iterator<vector<int>::iterator>::reference refpos = *pos;
   cout << "The iterator pos points to: " << refpos << "." << endl;
}
```

Output

```
The vector vec is: ( 2 4 6 8 10 ).
The vector vec reversed is: ( 10 8 6 4 2 ).
The iterator pos points to: 6.
```

Requirements

Header: <iterator>

See Also

Reference

reverse_iterator Class

Standard Template Library

Other Resources

reverse_iterator Members

reverse_iterator::operator+

Adds an offset to an iterator and returns the new **reverse_iterator** addressing the inserted element at the new offset position.

```
reverse_iterator operator+(
   difference_type _Off
) const;
```

Parameters

_Off

The offset to be added to the reverse iterator.

Return Value

A **reverse_iterator** addressing the offset element.

Remarks

This member function may only be used if the **reverse_iterator** satisfies the requirements for a random-access iterator.

Example

```
// reverse_iterator_op_add.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( 2 * i );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
      cout << *rvIter << " ";
   cout << ")." << endl;

   // Initializing reverse_iterators to the first element
   vector <int>::reverse_iterator rVPOS1 = vec.rbegin ( );

   cout << "The iterator rVPOS1 initially points to the first "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

   vector <int>::reverse_iterator rVPOS2 =rVPOS1 + 2; // offset added
   cout << "After the +2 offset, the iterator rVPOS2 points\n"
        << " to the 3rd element in the reversed sequence: "
        << *rVPOS2 << "." << endl;
}
```

Output

```
The vector vec is: ( 2 4 6 8 10 ).
The vector vec reversed is: ( 10 8 6 4 2 ).
The iterator rVPOS1 initially points to the first element
 in the reversed sequence: 10.
After the +2 offset, the iterator rVPOS2 points
 to the 3rd element in the reversed sequence: 6.
```

Requirements

Header: <iterator>

See Also

Reference

reverse_iterator Class

Standard Template Library

Other Resources

reverse_iterator Members

reverse_iterator::operator++

Increments the reverse_iterator to the previous element.

```
reverse_iterator& operator++( );
reverse_iterator operator++( int );
```

Return Value

The first operator returns the preincremented **reverse_iterator** and the second, the postincrement operator, returns a copy of the incremented **reverse_iterator**.

Remarks

This member function may only be used if the **reverse_iterator** satisfies the requirements for a bidirectional iterator.

Example

```
// reverse_iterator_op_incr.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for ( i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( 2 * i - 1 );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
      cout << *vIter << " ";
   cout << "))." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
      cout << *rvIter << " ";
   cout << "))." << endl;

   // Initializing reverse_iterators to the last element
   vector <int>::reverse_iterator rVPOS1 = vec.rbegin( );

   cout << "The iterator rVPOS1 initially points to the first "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

   rVPOS1++;  // postincrement, preincrement: ++rVPSO1

   cout << "After incrementing, the iterator rVPOS1 points\n"
        << " to the second element in the reversed sequence: "
        << *rVPOS1 << "." << endl;
}
```

Output

```
The vector vec is: ( 1 3 5 7 9 ).
The vector vec reversed is: ( 9 7 5 3 1 ).
```

```
The iterator rVPOS1 initially points to the first element
 in the reversed sequence: 9.
After incrementing, the iterator rVPOS1 points
 to the second element in the reversed sequence: 7.
```

Requirements

Header: <iterator>

See Also

Reference

reverse_iterator Class

Standard Template Library

Other Resources

reverse_iterator Members

reverse_iterator::operator+=

Adds a specified offset from a reverse_iterator.

```
reverse_iterator& operator+=(
   difference_type _Off
);
```

Parameters

_Off

The offset by which to increment the iterator.

Return Value

A reference to the element addressed by the **reverse_iterator**.

Example

```
// reverse_iterator_op_addoff.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( 2 * i );
   }

   vector <int>::iterator vIter;

   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
      cout << *rvIter << " ";
   cout << ")." << endl;

   // Initializing reverse_iterators to the last element
   vector <int>::reverse_iterator rVPOS1 = vec.rbegin ( );

   cout << "The iterator rVPOS1 initially points to the first "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

   rVPOS1+=2;   // addition of an offset
   cout << "After the +2 offset, the iterator rVPOS1 now points\n"
        << " to the third element in the reversed sequence: "
        << *rVPOS1 << "." << endl;
}
```

Output

```
The vector vec is: ( 2 4 6 8 10 ).
```

```
The vector vec reversed is: ( 10 8 6 4 2 ).
The iterator rVPOS1 initially points to the first element
 in the reversed sequence: 10.
After the +2 offset, the iterator rVPOS1 now points
 to the third element in the reversed sequence: 6.
```

Requirements

Header: <iterator>

See Also

Reference

reverse_iterator Class

Standard Template Library

Other Resources

reverse_iterator Members

reverse_iterator::operator-

Subtracts an offset from a **reverse_iterator** and returns a **reverse_iterator** addressing the element at the offset position.

```
reverse_iterator operator-(
   difference_type _Off
) const;
```

Parameters

_Off

The offset to be subtracted from the reverse_iterator.

Return Value

A **reverse_iterator** addressing the offset element.

Remarks

This member function may only be used if the **reverse_iterator** satisfies the requirements for a random-access iterator.

Example

```
// reverse_iterator_op_sub.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for ( i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( 3 * i );
   }

   vector <int>::iterator vIter;

   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
      cout << *rvIter << " ";
   cout << ")." << endl;

   // Initializing reverse_iterators to the first element
   vector <int>::reverse_iterator rVPOS1 = vec.rend ( ) - 1;

   cout << "The iterator rVPOS1 initially points to the last "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

   vector <int>::reverse_iterator rVPOS2 =rVPOS1 - 2; // offset subtracted
   cout << "After the -2 offset, the iterator rVPOS2 points\n"
        << " to the 2nd element from the last in the reversed sequence: "
        << *rVPOS2 << "." << endl;
}
```

Output

```
The vector vec is: ( 3 6 9 12 15 ).
The vector vec reversed is: ( 15 12 9 6 3 ).
The iterator rVPOS1 initially points to the last element
 in the reversed sequence: 3.
After the -2 offset, the iterator rVPOS2 points
 to the 2nd element from the last in the reversed sequence: 9.
```

Requirements

Header: <iterator>

See Also

Reference

reverse_iterator Class

Standard Template Library

Other Resources

reverse_iterator Members

reverse_iterator::operator--

Decrements the reverse_iterator to the previous element.

```
reverse_iterator& operator--( );
reverse_iterator operator--( int );
```

Return Value

The first operator returns the predecremented **reverse_iterator** and the second, the postdecrement operator, returns a copy of the decremented **reverse_iterator**.

Remarks

This member function may only be used if the **reverse_iterator** satisfies the requirements for a bidirectional iterator.

Example

```
// reverse_iterator_op_decr.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( 2 * i - 1 );
   }

   vector <int>::iterator vIter;

   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
      cout << *vIter << " ";
   cout << "))." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
      cout << *rvIter << " ";
   cout << "))." << endl;

   // Initializing reverse_iterators to the first element
   vector <int>::reverse_iterator rVPOS1 = vec.rend ( ) - 1;

   cout << "The iterator rVPOS1 initially points to the last "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;
   rVPOS1--;  // postdecrement, predecrement: --rVPSO1

   cout << "After the decrement, the iterator rVPOS1 points\n"
        << " to the next-to-last element in the reversed sequence: "
        << *rVPOS1 << "." << endl;
}
```

Output

```
The vector vec is: ( 1 3 5 7 9 ).
The vector vec reversed is: ( 9 7 5 3 1 ).
```

```
The iterator rVPOS1 initially points to the last element
 in the reversed sequence: 1.
After the decrement, the iterator rVPOS1 points
 to the next-to-last element in the reversed sequence: 3.
```

Requirements

Header: <iterator>

See Also

Reference

reverse_iterator Class

Standard Template Library

Other Resources

reverse_iterator Members

reverse_iterator::operator-=

Subtracts a specified offset from a **reverse_iterator**.

```
reverse_iterator& operator-=(
   difference_type _Off
);
```

Parameters

_Off

The offset to be subtracted from the **reverse_iterator**.

Remarks

This member function may only be used if the **reverse_iterator** satisfies the requirements for a random-access iterator.

The operator evaluates **current** + *_Off*. then returns ***this**.

Example

```
// reverse_iterator_op_suboff.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( 3 * i );
   }

   vector <int>::iterator vIter;

   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
      cout << *rvIter << " ";
   cout << ")." << endl;

   // Initializing reverse_iterators to the first element
   vector <int>::reverse_iterator rVPOS1 = vec.rend ( ) - 1;

   cout << "The iterator rVPOS1 initially points to the last "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

   rVPOS1-=2;      // Subtraction of an offset
   cout << "After the -2 offset, the iterator rVPOS1 now points\n"
        << " to the 2nd element from the last in the reversed sequence: "
        << *rVPOS1 << "." << endl;
}
```

Output

```
The vector vec is: ( 3 6 9 12 15 ).
The vector vec reversed is: ( 15 12 9 6 3 ).
The iterator rVPOS1 initially points to the last element
 in the reversed sequence: 3.
After the -2 offset, the iterator rVPOS1 now points
 to the 2nd element from the last in the reversed sequence: 9.
```

Requirements

Header: <iterator>

See Also

Reference

reverse_iterator Class

Standard Template Library

Other Resources

reverse_iterator Members

reverse_iterator::operator->

Returns a pointer to the element addressed by the **reverse_iterator**.

```
pointer operator->( ) const;
```

Return Value

A pointer to the element addressed by the **reverse_iterator**.

Remarks

The operator returns **&**this**.

Example

```
// reverse_iterator_ptrto.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <utility>
#include <iostream>

int main( )
{
   using namespace std;

   typedef vector<pair<int,int> > pVector;
   pVector vec;
   vec.push_back(pVector::value_type(1,2));
   vec.push_back(pVector::value_type(3,4));
   vec.push_back(pVector::value_type(5,6));

   pVector::iterator pvIter;
   cout << "The vector vec of integer pairs is:\n( ";
   for ( pvIter = vec.begin ( ) ; pvIter != vec.end ( ); pvIter++)
      cout << "( " << pvIter -> first << ", " << pvIter -> second << ") ";
   cout << ")" << endl << endl;

   pVector::reverse_iterator rpvIter;
   cout << "The vector vec reversed is:\n( ";
   for ( rpvIter = vec.rbegin( ) ; rpvIter != vec.rend( ); rpvIter++ )
      cout << "( " << rpvIter -> first << ", " << rpvIter -> second << ") ";
   cout << ")" << endl << endl;

   pVector::iterator pos = vec.begin ( );
   pos++;
   cout << "The iterator pos points to:\n( " << pos -> first << ", "
   << pos -> second << " )" << endl << endl;

   pVector::reverse_iterator rpos (pos);

   // Use operator -> with return type: why type int and not int*?
   int fint = rpos -> first;
   int sint = rpos -> second;

   cout << "The reverse_iterator rpos points to:\n( " << fint << ", "
   << sint << " )" << endl;
}
```

Output

```
The vector vec of integer pairs is:
( ( 1, 2) ( 3, 4) ( 5, 6) )
```

```
The vector vec reversed is:
( ( 5, 6) ( 3, 4) ( 1, 2) )

The iterator pos points to:
( 3, 4 )

The reverse_iterator rpos points to:
( 1, 2 )
```

Requirements

Header: <iterator>

See Also

Reference

reverse_iterator Class

Standard Template Library

Other Resources

reverse_iterator Members

reverse_iterator::operator[]

Returns a reference to an element offset from the element addressed by a **reverse_iterator** by a specified number of positions.

```
reference operator[](
   difference_type _Off
) const;
```

Parameters

_Off

The offset from the **reverse_iterator** address.

Return Value

The reference to the element offset.

Remarks

The operator returns ***(*this** + *_Off*).

Example

```
// reverse_iterator_ret_ref.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( 2 * i );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
      cout << *rvIter << " ";
   cout << ")." << endl;

   vector <int>::iterator pos;
   pos = find ( vec.begin ( ), vec.end ( ), 8 );
   reverse_iterator<vector<int>::iterator> rpos ( pos );

   // Declare a difference type for a parameter
   reverse_iterator<vector<int>::iterator>::difference_type diff = 2;

   cout << "The iterator pos points to: " << *pos << "." << endl;
   cout << "The iterator rpos points to: " << *rpos << "." << endl;

   // Declare a reference return type & use operator[]
```

```
    reverse_iterator<vector<int>::iterator>::reference refrpos = rpos [diff];
    cout << "The iterator rpos now points to: " << refrpos << "." << endl;
}
```

Output

```
The vector vec is: ( 2 4 6 8 10 ).
The vector vec reversed is: ( 10 8 6 4 2 ).
The iterator pos points to: 8.
The iterator rpos points to: 6.
The iterator rpos now points to: 2.
```

Requirements

Header: <iterator>

See Also

Reference

reverse_iterator Class

Standard Template Library

Other Resources

reverse_iterator Members

<limits>

Defines the template class **numeric_limits** and two enumerations concerning floating-point representations and rounding.

For a list of all members of this header, see <limits> Members.

```
#include <limits>
```

Remarks

Explicit specializations of the **numeric_limits** class describe many properties of the fundamental types, including the character, integer, and floating-point types and **bool** that are implementation defined rather than fixed by the rules of the C++ language. Properties described in <limits> include accuracy, minimum and maximum sized representations, rounding, and signaling type errors.

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<limits> Members

Standard C++ Library Header Files

<limits> Members

Reference

Enumerations

float_denorm_style	The enumeration describes the various methods that an implementation can choose for representing a de normalized floating-point value — one too small to represent as a normalized value:
float_round_style	The enumeration describes the various methods that an implementation can choose for rounding a floatin g-point value to an integer value.

Classes

numeric_limits Class	The template class describes arithmetic properties of built-in numerical types.

See Also

Reference

<limits>

Thread Safety in the Standard C++ Library

<limits> Enumerations

For information about the enumerations in <limits>, see <limits> Members.

float_denorm_style

The enumeration describes the various methods that an implementation can choose for representing a denormalized floating-point value — one too small to represent as a normalized value:

```
enum float_denorm_style {
   denorm_indeterminate = -1,
   denorm_absent = 0,
   denorm_present = 1
   };
```

Return Value

The enumeration returns:

- **denorm_indeterminate** if the presence or absence of denormalized forms cannot be determined at translation time.
- **denorm_absent** if denormalized forms are absent.
- **denorm_present** if denormalized forms are present.

Example

See numeric_limits::has_denorm for an example in which the values of this enumeration may be accessed.

Requirements

Header: <limits>

See Also

Other Resources

<limits> Members

float_round_style

The enumeration describes the various methods that an implementation can choose for rounding a floating-point value to an integer value.

```
enum float_round_style {
   round_indeterminate = -1,
   round_toward_zero = 0,
   round_to_nearest = 1,
   round_toward_infinity = 2,
   round_toward_neg_infinity = 3
   };
```

Return Value

The enumeration returns:

- **round_indeterminate** if the rounding method cannot be determined.
- **round_toward_zero** if the round toward zero.
- **round_to_nearest** if the round to nearest integer.
- **round_toward_infinity** if the round away from zero.
- **round_toward_neg_infinity** if the round to more negative integer.

Example

See numeric_limits::round_style for an example in which the values of this enumeration may be accessed.

Requirements

Header: <limits>

See Also

Other Resources

<limits> Members

<limits> Classes

For information about the classes in <limits>, see <limits> Members.

numeric_limits Class

The template class describes arithmetic properties of built-in numerical types.

For a list of all members of this type, see numeric_limits Members.

```
template<class Type> class numeric_limits
```

Parameters

Type

The fundamental element data type whose properties are being tested or queried or set.

Remarks

The header defines explicit specializations for the types **wchar_t**, **bool**, **char**, **signed char**, **unsigned char**, **short**, **unsigned short**, **int**, **unsigned int**, **long**, **unsigned long**, **float**, **double**, and **long double**. For these explicit specializations, the member numeric_limits::is_specialized is **true**, and all relevant members have meaningful values. The program can supply additional explicit specializations. Most member functions of the class describe or test possible implementations of **float**.

For an arbitrary specialization, no members have meaningful values. A member object that does not have a meaningful value stores zero (or false) and a member function that does not return a meaningful value returns **Type**(0).

Requirements

Header: <limits>

See Also

Reference

Members of the numeric_limits Class

Thread Safety in the Standard C++ Library

Other Resources

numeric_limits Members

<limits> Members

numeric_limits Members

Reference

Static Functions

denorm_min	Returns the smallest nonzero denormalized value.
digits	Returns the number of radix digits that the type can represent without loss of precision.
digits10	Returns the number of decimal digits that the type can represent without loss of precision.
epsilon	Returns the difference between 1 and the smallest value greater than 1 that the data type can represent.
has_denorm	Tests whether a type allows denormalized values.
has_denorm_loss	Tests whether loss of accuracy is detected as a denormalization loss rather than as an inexact result.
has_infinity	Tests whether a type has a representation for positive infinity.
has_quiet_NaN	Tests whether a type has a representation for a quiet not a number (NAN), which is nonsignaling.
has_signaling_NaN	Tests whether a type has a representation for signaling not a number (NAN).
infinity	The representation for positive infinity for a type, if available.
is_bounded	Tests if the set of values that a type may represent is finite.
is_exact	Tests if the calculations done on a type are free of rounding errors.
is_iec559	Tests if a type conforms to IEC 559 standards.
is_integer	Tests if a type has an integer representation.
is_modulo	Tests if a type has a modulo representation.
is_signed	Tests if a type has a signed representation.
is_specialized	Tests if a type has an explicit specialization defined in the template class **numeric_limits**.
max	Returns the maximum finite value for a type.
max_exponent	Returns the maximum positive integral exponent that the floating-point type can represent as a finite value when a base of radix is raised to that power.
max_exponent10	Returns the maximum positive integral exponent that the floating-point type can represent as a finite value when a base of ten is raised to that power.
min	Returns the minimum normalized value for a type.
min_exponent	Returns the maximum negative integral exponent that the floating-point type can represent as a finite valu e when a base of radix is raised to that power.

min_exponent10	Returns the maximum negative integral exponent that the floating-point type can represent as a finite value when a base of ten is raised to that power.
quiet_NaN	Returns the representation of a quiet not a number (NAN) for the type.
radix	Returns the integral base, referred to as radix, used for the representation of a type.
round_error	Returns the maximum rounding error for the type.
round_style	Returns a value that describes the various methods that an implementation can choose for rounding a floating-point value to an integer value.
signaling_NaN	Returns the representation of a signaling not a number (NAN) for the type.
tinyness_before	Tests whether a type can determine that a value is too small to represent as a normalized value before rounding it.
traps	Tests whether trapping that reports on arithmetic exceptions is implemented for a type.

See Also

Reference

strstreambuf Class

Thread Safety in the Standard C++ Library

numeric_limits Static Functions

For information about the static functions in the numeric_limits class, see numeric_limits Members.

numeric_limits::denorm_min

Returns the smallest nonzero denormalized value.

```
static Type denorm_min( ) throw( );
```

Return Value

The smallest nonzero denormalized value.

Remarks

long double is the same as **double** for the C++ compiler.

The function returns the minimum value for the type, which is the same as min if has_denorm is not equal to **denorm_present**.

Example

```
// numeric_limits_denorm_min.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The smallest nonzero denormalized value\n for float "
        << "objects is: " << numeric_limits<float>::denorm_min( )
        << endl;
   cout << "The smallest nonzero denormalized value\n for double "
        << "objects is: " << numeric_limits<double>::denorm_min( )
        << endl;
   cout << "The smallest nonzero denormalized value\n for long double "
        << "objects is: " << numeric_limits<long double>::denorm_min( )
        << endl;

   // A smaller value will round to zero
   cout << numeric_limits<float>::denorm_min( )/2 <<endl;
   cout << numeric_limits<double>::denorm_min( )/2 <<endl;
   cout << numeric_limits<long double>::denorm_min( )/2 <<endl;
}
```

Output

```
The smallest nonzero denormalized value
 for float objects is: 1.4013e-045
The smallest nonzero denormalized value
 for double objects is: 4.94066e-324
The smallest nonzero denormalized value
 for long double objects is: 4.94066e-324
0
0
0
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::digits

Returns the number of radix digits that the type can represent without loss of precision.

```
static const int digits = 0;
```

Return Value

The number of radix digits that the type can represent without loss of precision.

Remarks

The member stores the number of radix digits that the type can represent without change, which is the number of bits other than any sign bit for a predefined integer type, or the number of mantissa digits for a predefined floating-point type.

Example

```
// numeric_limits_digits_min.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << numeric_limits<float>::digits <<endl;
   cout << numeric_limits<double>::digits <<endl;
   cout << numeric_limits<long double>::digits <<endl;
   cout << numeric_limits<int>::digits <<endl;
   cout << numeric_limits<__int64>::digits <<endl;
}
```

Output

```
24
53
53
31
63
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::digits10

Returns the number of decimal digits that the type can represent without loss of precision.

```
static const int digits10 = 0;
```

Return Value

The number of decimal digits that the type can represent without loss of precision.

Example

```
// numeric_limits_digits10.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << numeric_limits<float>::digits10 <<endl;
   cout << numeric_limits<double>::digits10 <<endl;
   cout << numeric_limits<long double>::digits10 <<endl;
   cout << numeric_limits<int>::digits10 <<endl;
   cout << numeric_limits<__int64>::digits10 <<endl;
   float f = (float)99999999;
   cout.precision ( 10 );
   cout << "The float is; " << f << endl;
}
```

Output

```
6
15
15
9
18
The float is; 100000000
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::epsilon

The function returns the difference between 1 and the smallest value greater than 1 that is representable for the data type.

```
static Type epsilon( ) throw( );
```

Return Value

The difference between 1 and the smallest value greater than 1 that is representable for the data type.

Remarks

The value is FLT_EPSILON for type **float**. **epsilon** for a type is the smallest positive floating-point number *N* such that *N* + **epsilon** + *N* is representable.

Example

```
// numeric_limits_epsilon.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The difference between 1 and the smallest "
        << "value greater than 1\n for float objects is: "
        << numeric_limits<float>::epsilon( )
        << endl;
   cout << "The difference between 1 and the smallest "
        << "value greater than 1\n for double objects is: "
        << numeric_limits<double>::epsilon( )
        << endl;
   cout << "The difference between 1 and the smallest "
        << "value greater than 1\n for long double objects is: "
        << numeric_limits<long double>::epsilon( )
        << endl;
}
```

Output

```
The difference between 1 and the smallest value greater than 1
 for float objects is: 1.19209e-007
The difference between 1 and the smallest value greater than 1
 for double objects is: 2.22045e-016
The difference between 1 and the smallest value greater than 1
 for long double objects is: 2.22045e-016
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::has_denorm

Tests whether a type allows denormalized values.

```
static const float_denorm_style has_denorm = denorm_absent;
```

Return Value

An enumeration value of type **const float_denorm_style**, indicating whether the type allows denormalized values.

Remarks

The member stores **denorm_present** for a floating-point type that has denormalized values, effectively a variable number of exponent bits.

Example

```
// numeric_limits_has_denorm.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects allow denormalized values: "
        << numeric_limits<float>::has_denorm
        << endl;
   cout << "Whether double objects allow denormalized values: "
        << numeric_limits<double>::has_denorm
        << endl;
   cout << "Whether long int objects allow denormalized values: "
        << numeric_limits<long int>::has_denorm
        << endl;
}
```

Output

```
Whether float objects allow denormalized values: 1
Whether double objects allow denormalized values: 1
Whether long int objects allow denormalized values: 0
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::has_denorm_loss

Tests whether loss of accuracy is detected as a denormalization loss rather than as an inexact result.

```
static const bool has_denorm_loss = false;
```

Return Value

true if the loss of accuracy is detected as a denormalization loss; **false** if not.

Remarks

The member stores true for a type that determines whether a value has lost accuracy because it is delivered as a denormalized result (too small to represent as a normalized value) or because it is inexact (not the same as a result not subject to limitations of exponent range and precision), an option with IEC 559 floating-point representations that can affect some results.

Example

```
// numeric_limits_has_denorm_loss.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects can detect denormalized loss: "
        << numeric_limits<float>::has_denorm_loss
        << endl;
   cout << "Whether double objects can detect denormalized loss: "
        << numeric_limits<double>::has_denorm_loss
        << endl;
   cout << "Whether long int objects can detect denormalized loss: "
        << numeric_limits<long int>::has_denorm_loss
        << endl;
}
```

Output

```
Whether float objects can detect denormalized loss: 1
Whether double objects can detect denormalized loss: 1
Whether long int objects can detect denormalized loss: 0
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::has_infinity

Tests whether a type has a representation for positive infinity.

```
static const bool has_infinity = false;
```

Return Value

true if the type has a representation for positive infinity; **false** if not.

Remarks

The member returns **true** if is_iec559 is **true**.

Example

```
// numeric_limits_has_infinity.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects have infinity: "
        << numeric_limits<float>::has_infinity
        << endl;
   cout << "Whether double objects have infinity: "
        << numeric_limits<double>::has_infinity
        << endl;
   cout << "Whether long int objects have infinity: "
        << numeric_limits<long int>::has_infinity
        << endl;
}
```

Output

```
Whether float objects have infinity: 1
Whether double objects have infinity: 1
Whether long int objects have infinity: 0
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::has_quiet_NaN

Tests whether a type has a representation for a quiet not a number (NAN), which is nonsignaling.

```
static const bool has_quiet_NaN = false;
```

Return Value

true if the **type** has a representation for a quiet NAN; **false** if not.

Remarks

A quiet NAN is an encoding for not a number, which does not signal its presence in an expression. The return value is **true** if is_iec559 is true.

Example

```
// numeric_limits_has_quiet_nan.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects have quiet_NaN: "
        << numeric_limits<float>::has_quiet_NaN
        << endl;
   cout << "Whether double objects have quiet_NaN: "
        << numeric_limits<double>::has_quiet_NaN
        << endl;
   cout << "Whether long int objects have quiet_NaN: "
        << numeric_limits<long int>::has_quiet_NaN
        << endl;
}
```

Output

```
Whether float objects have quiet_NaN: 1
Whether double objects have quiet_NaN: 1
Whether long int objects have quiet_NaN: 0
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::has_signaling_NaN

Tests whether a type has a representation for signaling not a number (NAN).

```
static const bool has_signaling_NaN = false;
```

Return Value

true if the type has a representation for a signaling NAN; **false** if not.

Remarks

A signaling NAN is an encoding for not a number, which signals its presence in an expression. The return value is **true** is_iec559 is true.

Example

```
// numeric_limits_has_signaling_nan.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects have a signaling_NaN: "
        << numeric_limits<float>::has_signaling_NaN
        << endl;
   cout << "Whether double objects have a signaling_NaN: "
        << numeric_limits<double>::has_signaling_NaN
        << endl;
   cout << "Whether long int objects have a signaling_NaN: "
        << numeric_limits<long int>::has_signaling_NaN
        << endl;
}
```

Output

```
Whether float objects have a signaling_NaN: 1
Whether double objects have a signaling_NaN: 1
Whether long int objects have a signaling_NaN: 0
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::infinity

The representation of positive infinity for a type, if available.

```
static Type infinity( ) throw( );
```

Return Value

The representation of positive infinity for a type, if available.

Remarks

The return value is meaningful only if has_infinity is **true**.

Example

```
// numeric_limits_infinity.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << numeric_limits<float>::has_infinity <<endl;
   cout << numeric_limits<double>::has_infinity<<endl;
   cout << numeric_limits<long double>::has_infinity <<endl;
   cout << numeric_limits<int>::has_infinity <<endl;
   cout << numeric_limits<__int64>::has_infinity <<endl;

   cout << "The representation of infinity for type float is: "
        << numeric_limits<float>::infinity( ) <<endl;
   cout << "The representation of infinity for type double is: "
        << numeric_limits<double>::infinity( ) <<endl;
   cout << "The representation of infinity for type long double is: "
        << numeric_limits<long double>::infinity( ) <<endl;
}
```

Output

```
1
1
1
0
0
The representation of infinity for type float is: 1.#INF
The representation of infinity for type double is: 1.#INF
The representation of infinity for type long double is: 1.#INF
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::is_bounded

Tests if the set of values that a type may represent is finite.

```
static const bool is_bounded = false;
```

Return Value

true if the type has a bounded set of representable values; **false** if not.

Remarks

All predefined types have a bounded set of representable values and return **true**.

Example

```
// numeric_limits_is_bounded.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects have bounded set "
        << "of representable values: "
        << numeric_limits<float>::is_bounded
        << endl;
   cout << "Whether double objects have bounded set "
        << "of representable values: "
        << numeric_limits<double>::is_bounded
        << endl;
   cout << "Whether long int objects have bounded set "
        << "of representable values: "
        << numeric_limits<long int>::is_bounded
        << endl;
   cout << "Whether unsigned char objects have bounded set "
        << "of representable values: "
        << numeric_limits<unsigned char>::is_bounded
        << endl;
}
```

Output

```
Whether float objects have bounded set of representable values: 1
Whether double objects have bounded set of representable values: 1
Whether long int objects have bounded set of representable values: 1
Whether unsigned char objects have bounded set of representable values: 1
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::is_exact

Tests if the calculations done on a type are free of rounding errors.

```
static const bool is_exact = false;
```

Return Value

true if the calculations are free of rounding errors; **false** if not.

Remarks

All predefined integer types have exact representations for their values and return **false**. A fixed-point or rational representation is also considered exact, but a floating-point representation is not.

Example

```
// numeric_limits_is_exact.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects have calculations "
        << "free of rounding errors: "
        << numeric_limits<float>::is_exact
        << endl;
   cout << "Whether double objects have calculations "
        << "free of rounding errors: "
        << numeric_limits<double>::is_exact
        << endl;
   cout << "Whether long int objects have calculations "
        << "free of rounding errors: "
        << numeric_limits<long int>::is_exact
        << endl;
   cout << "Whether unsigned char objects have calculations "
        << "free of rounding errors: "
        << numeric_limits<unsigned char>::is_exact
        << endl;
}
```

Output

```
Whether float objects have calculations free of rounding errors: 0
Whether double objects have calculations free of rounding errors: 0
Whether long int objects have calculations free of rounding errors: 1
Whether unsigned char objects have calculations free of rounding errors: 1
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::is_iec559

Tests if a type conforms to IEC 559 standards.

```
static const bool is_iec559 = false;
```

Return Value

true if the type conforms to the IEC 559 standards; **false** if not.

Remarks

The IEC 559 is an international standard for representing floating-point values and is also known as IEEE 754 in the USA.

Example

```
// numeric_limits_is_iec559.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects conform to iec559 standards: "
        << numeric_limits<float>::is_iec559
        << endl;
   cout << "Whether double objects conform to iec559 standards: "
        << numeric_limits<double>::is_iec559
        << endl;
   cout << "Whether int objects conform to iec559 standards: "
        << numeric_limits<int>::is_iec559
        << endl;
   cout << "Whether unsigned char objects conform to iec559 standards: "
        << numeric_limits<unsigned char>::is_iec559
        << endl;
}
```

Output

```
Whether float objects conform to iec559 standards: 1
Whether double objects conform to iec559 standards: 1
Whether int objects conform to iec559 standards: 0
Whether unsigned char objects conform to iec559 standards: 0
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::is_integer

Tests if a type has an integer representation.

```
static const bool is_integer = false;
```

Return Value

true if the type has an integer representation; **false** if not.

Remarks

All predefined integer types have an integer representation.

Example

```
// numeric_limits_is_integer.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects have an integral representation: "
        << numeric_limits<float>::is_integer
        << endl;
   cout << "Whether double objects have an integral representation: "
        << numeric_limits<double>::is_integer
        << endl;
   cout << "Whether int objects have an integral representation: "
        << numeric_limits<int>::is_integer
        << endl;
   cout << "Whether unsigned char objects have an integral representation: "
        << numeric_limits<unsigned char>::is_integer
        << endl;
}
```

Output

```
Whether float objects have an integral representation: 0
Whether double objects have an integral representation: 0
Whether int objects have an integral representation: 1
Whether unsigned char objects have an integral representation: 1
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::is_modulo

Tests if a **type** has a modulo representation.

```
static const bool is_modulo = false;
```

Return Value

true if the type has a modulo representation; **false** if not.

Remarks

A modulo representation is a representation where all results are reduced modulo some value. All predefined unsigned integer types have a modulo representation.

Example

```
// numeric_limits_is_modulo.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects have a modulo representation: "
        << numeric_limits<float>::is_modulo
        << endl;
   cout << "Whether double objects have a modulo representation: "
        << numeric_limits<double>::is_modulo
        << endl;
   cout << "Whether signed char objects have a modulo representation: "
        << numeric_limits<signed char>::is_modulo
        << endl;
   cout << "Whether unsigned char objects have a modulo representation: "
        << numeric_limits<unsigned char>::is_modulo
        << endl;
}
```

Output

```
Whether float objects have a modulo representation: 0
Whether double objects have a modulo representation: 0
Whether signed char objects have a modulo representation: 1
Whether unsigned char objects have a modulo representation: 1
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::is_signed

Tests if a type has a signed representation.

```
static const bool is_signed = false;
```

Return Value

true if the type has a signed representation; **false** if not.

Remarks

The member stores true for a type that has a signed representation, which is the case for all predefined floating-point and signed integer types.

Example

```
// numeric_limits_is_signaled.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects have a signed representation: "
        << numeric_limits<float>::is_signed
        << endl;
   cout << "Whether double objects have a signed representation: "
        << numeric_limits<double>::is_signed
        << endl;
   cout << "Whether signed char objects have a signed representation: "
        << numeric_limits<signed char>::is_signed
        << endl;
   cout << "Whether unsigned char objects have a signed representation: "
        << numeric_limits<unsigned char>::is_signed
        << endl;
}
```

Output

```
Whether float objects have a signed representation: 1
Whether double objects have a signed representation: 1
Whether signed char objects have a signed representation: 1
Whether unsigned char objects have a signed representation: 0
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::is_specialized

Tests if a type has an explicit specialization defined in the template class **numeric_limits**.

```
static const bool is_specialized = false;
```

Return Value

true if the type has an explicit specialization defined in the template class; **false** if not.

Remarks

All scalar types other than pointers have an explicit specialization defined for template class **numeric_limits**.

Example

```
// numeric_limits_is_specialized.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects have an explicit "
        << "specialization in the class: "
        << numeric_limits<float>::is_specialized
        << endl;
   cout << "Whether float* objects have an explicit "
        << "specialization in the class: "
        << numeric_limits<float*>::is_specialized
        << endl;
   cout << "Whether int objects have an explicit "
        << "specialization in the class: "
        << numeric_limits<int>::is_specialized
        << endl;
   cout << "Whether int* objects have an explicit "
        << "specialization in the class: "
        << numeric_limits<int*>::is_specialized
        << endl;
}
```

Output

```
Whether float objects have an explicit specialization in the class: 1
Whether float* objects have an explicit specialization in the class: 0
Whether int objects have an explicit specialization in the class: 1
Whether int* objects have an explicit specialization in the class: 0
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::max

Returns the maximum finite value for a type.

```
static Type max( ) throw( );
```

Return Value

The maximum finite value for a type.

Remarks

The maximum finite value is INT_MAX for type **int** and FLT_MAX for type **float**. The return value is meaningful if is_bounded is **true**.

Example

```
// numeric_limits_max.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main() {
   cout << "The maximum value for type float is:  "
        << numeric_limits<float>::max( )
        << endl;
   cout << "The maximum value for type double is:  "
        << numeric_limits<double>::max( )
        << endl;
   cout << "The maximum value for type int is:  "
        << numeric_limits<int>::max( )
        << endl;
   cout << "The maximum value for type short int is:  "
        << numeric_limits<short int>::max( )
        << endl;
}
```

Output

```
The maximum value for type float is:  3.40282e+038
The maximum value for type double is:  1.79769e+308
The maximum value for type int is:  2147483647
The maximum value for type short int is:  32767
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::max_exponent

Returns the maximum positive integral exponent that the floating-point type can represent as a finite value when a base of radix is raised to that power.

```
static const int max_exponent = 0;
```

Return Value

The maximum integral radix-based exponent representable by the type.

Remarks

The member function return is meaningful only for floating-point types. The **max_exponent** is the value FLT_MAX_EXP for type **float**.

Example

```
// numeric_limits_max_exponent.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The maximum radix-based exponent for type float is:  "
        << numeric_limits<float>::max_exponent
        << endl;
   cout << "The maximum radix-based exponent for type double is:  "
        << numeric_limits<double>::max_exponent
        << endl;
   cout << "The maximum radix-based exponent for type long double is:  "
        << numeric_limits<long double>::max_exponent
        << endl;
}
```

Output

```
The maximum radix-based exponent for type float is:  128
The maximum radix-based exponent for type double is:  1024
The maximum radix-based exponent for type long double is:  1024
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::max_exponent10

Returns the maximum positive integral exponent that the floating-point type can represent as a finite value when a base of ten is raised to that power.

```
static const int max_exponent10 = 0;
```

Return Value

The maximum integral base 10 exponent representable by the type.

Remarks

The member function return is meaningful only for floating-point types. The **max_exponent** is the value FLT_MAX_10 for type **float**.

Example

```
// numeric_limits_max_exponent10.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The maximum base 10 exponent for type float is:  "
           << numeric_limits<float>::max_exponent10
           << endl;
   cout << "The maximum base 10 exponent for type double is:  "
           << numeric_limits<double>::max_exponent10
           << endl;
   cout << "The maximum base 10 exponent for type long double is:  "
           << numeric_limits<long double>::max_exponent10
           << endl;
}
```

Output

```
The maximum base 10 exponent for type float is:  38
The maximum base 10 exponent for type double is:  308
The maximum base 10 exponent for type long double is:  308
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::min

Returns the minimum normalized value for a type.

```
static Type min( ) throw( );
```

Return Value

The minimum normalized value for the type.

Remarks

The minimum normalized value is INT_MIN for type **int** and FLT_MIN for type **float**. The return value is meaningful if is_bounded is **true** or if is_signed is **false**.

Example

```
// numeric_limits_min.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The minimum value for type float is:  "
        << numeric_limits<float>::min( )
        << endl;
   cout << "The minimum value for type double is:  "
        << numeric_limits<double>::min( )
        << endl;
   cout << "The minimum value for type int is:  "
        << numeric_limits<int>::min( )
        << endl;
   cout << "The minimum value for type short int is:  "
        << numeric_limits<short int>::min( )
        << endl;
}
```

Output

```
The minimum value for type float is:  1.17549e-038
The minimum value for type double is:  2.22507e-308
The minimum value for type int is:  -2147483648
The minimum value for type short int is:  -32768
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::min_exponent

Returns the maximum negative integral exponent that the floating-point type can represent as a finite value when a base of radix is raised to that power.

```
static const int min_exponent = 0;
```

Return Value

The minimum integral radix-based exponent representable by the type.

Remarks

The member function is meaningful only for floating-point types. The **min_exponent** is the value FLT_MIN_EXP for type **float**.

Example

```
// numeric_limits_min_exponent.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The minimum radix-based exponent for type float is:  "
        << numeric_limits<float>::min_exponent
        << endl;
   cout << "The minimum radix-based exponent for type double is:  "
        << numeric_limits<double>::min_exponent
        << endl;
   cout << "The minimum radix-based exponent for type long double is:  "
         << numeric_limits<long double>::min_exponent
        << endl;
}
```

Output

```
The minimum radix-based exponent for type float is:  -125
The minimum radix-based exponent for type double is:  -1021
The minimum radix-based exponent for type long double is:  -1021
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::min_exponent10

Returns the maximum negative integral exponent that the floating-point type can represent as a finite value when a base of ten is raised to that power.

```
static const int min_exponent10 = 0;
```

Return Value

The minimum integral base 10 exponent representable by the type.

Remarks

The member function is meaningful only for floating-point types. The **min_exponent10** is the value FLT_MIN_10_EXP for type **float**.

Example

```
// numeric_limits_min_exponent10.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The minimum base 10 exponent for type float is:  "
        << numeric_limits<float>::min_exponent10
        << endl;
   cout << "The minimum base 10 exponent for type double is:  "
        << numeric_limits<double>::min_exponent10
        << endl;
   cout << "The minimum base 10 exponent for type long double is:  "
        << numeric_limits<long double>::min_exponent10
        << endl;
}
```

Output

```
The minimum base 10 exponent for type float is:  -37
The minimum base 10 exponent for type double is:  -307
The minimum base 10 exponent for type long double is:  -307
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::quiet_NaN

Returns the representation of a quiet not a number (NAN) for the type.

```
static Type quiet_NaN( ) throw( );
```

Return Value

The representation of a quiet NAN for the type.

Remarks

The return value is meaningful only if has_quiet_NaN is **true**.

Example

```
// numeric_limits_quiet_nan.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The quiet NaN for type float is:  "
        << numeric_limits<float>::quiet_NaN( )
        << endl;
   cout << "The quiet NaN for type int is:  "
        << numeric_limits<int>::quiet_NaN( )
        << endl;
   cout << "The quiet NaN for type long double is:  "
        << numeric_limits<long double>::quiet_NaN( )
        << endl;
}
```

Output

```
The quiet NaN for type float is:  1.#QNAN
The quiet NaN for type int is:  0
The quiet NaN for type long double is:  1.#QNAN
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::radix

Returns the integral base, referred to as radix, used for the representation of a type.

```
static const int radix = 0;
```

Return Value

The integral base for the representation of the type.

Remarks

The base is 2 for the predefined integer types, and the base to which the exponent is raised, or FLT_RADIX, for the predefined floating-point types.

Example

```
// numeric_limits_radix.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The base for type float is:  "
        << numeric_limits<float>::radix
        << endl;
   cout << "The base for type int is:  "
        << numeric_limits<int>::radix
        << endl;
   cout << "The base for type long double is:  "
        << numeric_limits<long double>::radix
        << endl;
}
```

Output

```
The base for type float is:  2
The base for type int is:  2
The base for type long double is:  2
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::round_error

Returns the maximum rounding error for the type.

```
static Type round_error( ) throw( );
```

Return Value

The maximum rounding error for the type.

Example

```
// numeric_limits_round_error.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The maximum rounding error for type float is:  "
        << numeric_limits<float>::round_error( )
        << endl;
   cout << "The maximum rounding error for type int is:  "
        << numeric_limits<int>::round_error( )
        << endl;
   cout << "The maximum rounding error for type long double is:  "
        << numeric_limits<long double>::round_error( )
        << endl;
}
```

Output

```
The maximum rounding error for type float is:  0.5
The maximum rounding error for type int is:  0
The maximum rounding error for type long double is:  0.5
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::round_style

Returns a value that describes the various methods that an implementation can choose for rounding a floating-point value to an integer value.

```
static const float_round_style round_style = round_toward_zero;
```

Return Value

A value from the **float_round_style** enumeration that describes the rounding style.

Remarks

The member stores a value that describes the various methods that an implementation can choose for rounding a floating-point value to an integer value.

The round style is hard coded in this implementation, so even if the program starts up with a different rounding mode, that value will not change.

Example

```
// numeric_limits_round_style.cpp
// compile with: /EHsc
#include <iostream>
#include <float.h>
#include <limits>

using namespace std;

int main( )
{
   cout << "The rounding style for a double type is: "
        << numeric_limits<double>::round_style << endl;
   _controlfp_s(NULL,_RC_DOWN,_MCW_RC );
   cout << "The rounding style for a double type is now: "
        << numeric_limits<double>::round_style << endl;
   cout << "The rounding style for an int type is: "
        << numeric_limits<int>::round_style << endl;
}
```

Output

```
The rounding style for a double type is: 1
The rounding style for a double type is now: 1
The rounding style for an int type is: 0
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::signaling_NaN

Returns the representation of a signaling not a number (NAN) for the type.

```
static Type signaling_NaN( ) throw( );
```

Return Value

The representation of a signaling NAN for the type.

Remarks

The return value is meaningful only if has_signaling_NaN is **true**.

Example

```
// numeric_limits_signaling_nan.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The signaling NaN for type float is:  "
        << numeric_limits<float>::signaling_NaN( )
        << endl;
   cout << "The signaling NaN for type int is:  "
        << numeric_limits<int>::signaling_NaN( )
        << endl;
   cout << "The signaling NaN for type long double is:  "
        << numeric_limits<long double>::signaling_NaN( )
        << endl;
}
```

Sample Output

The following is output on x86.

```
The signaling NaN for type float is:  1.#QNAN
The signaling NaN for type int is:  0
The signaling NaN for type long double is:  1.#QNAN
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::tinyness_before

Tests whether a type can determine that a value is too small to represent as a normalized value before rounding it.

```
static const bool tinyness_before = false;
```

Return Value

true if the type can detect tiny values before rounding; **false** if it cannot.

Remarks

Types that can detect tinyness where included as an option with IEC 559 floating-point representations and its implementation can affect some results.

Example

```
// numeric_limits_tinyness_before.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float types can detect tinyness before rounding: "
        << numeric_limits<float>::tinyness_before
        << endl;
   cout << "Whether double types can detect tinyness before rounding: "
        << numeric_limits<double>::tinyness_before
        << endl;
   cout << "Whether long int types can detect tinyness before rounding: "
        << numeric_limits<long int>::tinyness_before
        << endl;
   cout << "Whether unsigned char types can detect tinyness before rounding: "
        << numeric_limits<unsigned char>::tinyness_before
        << endl;
}
```

Output

```
Whether float types can detect tinyness before rounding: 1
Whether double types can detect tinyness before rounding: 1
Whether long int types can detect tinyness before rounding: 0
Whether unsigned char types can detect tinyness before rounding: 0
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

numeric_limits::traps

Tests whether trapping that reports on arithmetic exceptions is implemented for a type.

```
static const bool traps = false;
```

Return Value

true if trapping is implemented for the type; **false** if it is not.

Example

```
// numeric_limits_traps.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float types have implemented trapping: "
        << numeric_limits<float>::traps
        << endl;
   cout << "Whether double types have implemented trapping: "
        << numeric_limits<double>::traps
        << endl;
   cout << "Whether long int types have implemented trapping: "
        << numeric_limits<long int>::traps
        << endl;
   cout << "Whether unsigned char types have implemented trapping: "
        << numeric_limits<unsigned char>::traps
        << endl;
}
```

Output

```
Whether float types have implemented trapping: 1
Whether double types have implemented trapping: 1
Whether long int types have implemented trapping: 0
Whether unsigned char types have implemented trapping: 0
```

Requirements

Header: <limits>

See Also

Reference

strstreambuf Class

Other Resources

numeric_limits Members

<list> Header

Defines the container template class list and several supporting templates.

For a list of all members of this header, see <list> Members.

```
#include <list>
```

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<list> Members

Standard C++ Library Header Files

<list> Members

Reference

Operators

operator!=	Tests if the list object on the left side of the operator is not equal to the list object on the right side.
operator<	Tests if the list object on the left side of the operator is less than the list object on the right side.
operator<=	Tests if the list object on the left side of the operator is less than or equal to the list object on the right side.
operator==	Tests if the list object on the left side of the operator is equal to the list object on the right side.
operator>	Tests if the list object on the left side of the operator is greater than the list object on the right side.
operator>=	Tests if the list object on the left side of the operator is greater than or equal to the list object on the right side.

Classes

list Class	A template class of sequence containers that maintain their elements in a linear arrangement and allow efficient insertions and deletions at any location within the sequence.

See Also

Reference

<list> Header
Thread Safety in the Standard C++ Library
Standard Template Library

<list> Operators

For information about the operators in <list>, see <list> Members.

See Also

Reference

Standard Template Library

operator!= (<list>)

Tests if the list object on the left side of the operator is not equal to the list object on the right side.

```
bool operator!=(
   const list<Type, Allocator>& _Left,
   const list<Type, Allocator>& _Right
);
```

Parameters

_Left

An object of type **list**.

_Right

An object of type **list**.

Return Value

true if the lists are not equal; **false** if the lists are equal.

Remarks

The comparison between list objects is based on a pairwise comparison of their elements. Two lists are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// list_op_ne.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1, c2;
   c1.push_back( 1 );
   c2.push_back( 2 );

   if ( c1 != c2 )
      cout << "Lists not equal." << endl;
   else
      cout << "Lists equal." << endl;
}
```

Output

```
Lists not equal.
```

Requirements

Header: <list>

See Also

Reference

Standard Template Library

Other Resources

<list> Members

operator< (<list>)

Tests if the list object on the left side of the operator is less than the list object on the right side.

```
bool operator<(
   const list<Type, Allocator>& _Left,
   const list<Type, Allocator>& _Right
);
```

Parameters

_Left

An object of type **list**.

_Right

An object of type **list**.

Return Value

true if the list on the left side of the operator is less than but not equal to the list on the right side of the operator; otherwise **false**.

Remarks

The comparison between list objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// list_op_lt.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1, c2;
   c1.push_back( 1 );
   c1.push_back( 2 );
   c1.push_back( 4 );

   c2.push_back( 1 );
   c2.push_back( 3 );

   if ( c1 < c2 )
      cout << "List c1 is less than list c2." << endl;
   else
      cout << "List c1 is not less than list c2." << endl;
}
```

Output

```
List c1 is less than list c2.
```

Requirements

Header: <list>

See Also

Reference

Standard Template Library

Other Resources

<list> Members

operator<= (<list>)

Tests if the list object on the left side of the operator is less than or equal to the list object on the right side.

```
bool operator<=(
   const list<Type, Allocator>& _Left,
   const list<Type, Allocator>& _Right
);
```

Parameters

_Left

An object of type **list**.

_Right

An object of type **list**.

Return Value

true if the list on the left side of the operator is less than or equal to the list on the right side of the operator; otherwise **false**.

Remarks

The comparison between list objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// list_op_le.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1, c2;
   c1.push_back( 1 );
   c1.push_back( 2 );
   c1.push_back( 4 );

   c2.push_back( 1 );
   c2.push_back( 3 );

   if ( c1 <= c2 )
      cout << "List c1 is less than or equal to list c2." << endl;
   else
      cout << "List c1 is greater than list c2." << endl;
}
```

Output

```
List c1 is less than or equal to list c2.
```

Requirements

Header: <list>

See Also

Reference

Standard Template Library

Other Resources

<list> Members

operator== (<list>)

Tests if the list object on the left side of the operator is equal to the list object on the right side.

```
bool operator==(
   const list<Type, Allocator>& _Left,
   const list<Type, Allocator>& _Right
);
```

Parameters

_Left

An object of type **list**.

_Right

An object of type **list**.

Return Value

true if the list on the left side of the operator is equal to the list on the right side of the operator; otherwise **false**.

Remarks

The comparison between list objects is based on a pairwise comparison of their elements. Two lists are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// list_op_eq.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
int main( )
{
   using namespace std;

   list <int> c1, c2;
   c1.push_back( 1 );
   c2.push_back( 1 );

   if ( c1 == c2 )
      cout << "The lists are equal." << endl;
   else
      cout << "The lists are not equal." << endl;
}
```

Output

```
The lists are equal.
```

Requirements

Header: <list>

See Also

Reference

Standard Template Library

Other Resources

<list> Members

operator> (<list>)

Tests if the list object on the left side of the operator is greater than the list object on the right side.

```
bool operator>(
   const list<Type, Allocator>& _Left,
   const list<Type, Allocator>& _Right
);
```

Parameters

_Left

An object of type **list**.

_Right

An object of type **list**.

Return Value

true if the list on the left side of the operator is greater than the list on the right side of the operator; otherwise **false**.

Remarks

The comparison between list objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// list_op_gt.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
int main( )
{
   using namespace std;
   list <int> c1, c2;
   c1.push_back( 1 );
   c1.push_back( 3 );
   c1.push_back( 1 );

   c2.push_back( 1 );
   c2.push_back( 2 );
   c2.push_back( 2 );

   if ( c1 > c2 )
      cout << "List c1 is greater than list c2." << endl;
   else
      cout << "List c1 is not greater than list c2." << endl;
}
```

Output

```
List c1 is greater than list c2.
```

Requirements

Header: <list>

See Also

Reference

Standard Template Library

Other Resources

<list> Members

operator>= (<list>)

Tests if the list object on the left side of the operator is greater than or equal to the list object on the right side.

```
bool operator>=(
   const list<Type, Allocator>& _Left,
   const list<Type, Allocator>& _Right
);
```

Parameters

_Left

An object of type **list**.

_Right

An object of type **list**.

Return Value

true if the list on the left side of the operator is greater than or equal to the list on the right side of the operator; otherwise **false**.

Remarks

The comparison between list objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// list_op_ge.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1, c2;
   c1.push_back( 1 );
   c1.push_back( 3 );
   c1.push_back( 1 );

   c2.push_back( 1 );
   c2.push_back( 2 );
   c2.push_back( 2 );

   if ( c1 >= c2 )
      cout << "List c1 is greater than or equal to list c2." << endl;
   else
      cout << "List c1 is less than list c2." << endl;
}
```

Output

```
List c1 is greater than or equal to list c2.
```

Requirements

Header: <list>

See Also

Reference

Standard Template Library

Other Resources

<list> Members

<list> Classes

For information about the classes in <list>, see <list> Members.

See Also

Reference

Standard Template Library

list Class

The STL list class is a template class of sequence containers that maintain their elements in a linear arrangement and allow efficient insertions and deletions at any location within the sequence. The sequence is stored as a bidirectional linked list of elements, each containing a member of some type *Type*.

For a list of all members of this type, see list Class Members.

```
template <
   class Type,
   class Allocator=allocator<Type>
>
class list
```

Parameters

Type

The element data type to be stored in the list.

Allocator

The type that represents the stored allocator object that encapsulates details about the list's allocation and deallocation of memory. This argument is optional, and the default value is **allocator**<*Type*>.

Remarks

The choice of container type should be based in general on the type of searching and inserting required by the application. Vectors should be the preferred container for managing a sequence when random access to any element is at a premium and insertions or deletions of elements are only required at the end of a sequence. The performance of the class deque container is superior when random access is needed and insertions and deletions at both the beginning and the end of a sequence are at a premium.

The list member functions merge, reverse, unique, remove, and remove_if have been optimized for operation on list objects and offer a high-performance alternative to their generic counterparts.

List reallocation occurs when a member function must insert or erase elements of the list. In all such cases, only iterators or references that point at erased portions of the controlled sequence become invalid.

Include the STL standard header <list> to define the container template class list and several supporting templates.

Requirements

Header: <list>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

list Class Members

<list> Members

list Class Members

Reference

Constructors

list	Constructs a list of a specific size or with elements of a specific value or with a specific **allocator** or as a copy of some other list.

Typedefs

allocator_type	A type that represents the **allocator** class for a list object.
const_iterator	A type that provides a bidirectional iterator that can read a **const** element in a list.
const_pointer	A type that provides a pointer to a **const** element in a list.
const_reference	A type that provides a reference to a **const** element stored in a list for reading and performing **const** operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any **const** element in a list.
difference_type	A type that provides the difference between two iterators that refer to elements within the same list.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a list.
pointer	A type that provides a pointer to an element in a list.
reference	A type that provides a reference to a **const** element stored in a list for reading and performing **const** operations.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed list.
size_type	A type that counts the number of elements in a list.
value_type	A type that represents the data type stored in a list.

Member Functions

assign	Erases elements from a list and copies a new set of elements to the target list.
back	Returns a reference to the last element of a list.
begin	Returns an iterator addressing the first element in a list.
clear	Erases all the elements of a list.
empty	Tests if a list is empty.
end	Returns an iterator that addresses the location succeeding the last element in a list.
erase	Removes an element or a range of elements in a list from specified positions.
front	Returns a reference to the first element in a list.
get_allocator	Returns a copy of the **allocator** object used to construct a list.

insert	Inserts an element or a number of elements or a range of elements into a list at a specified position.
max_size	Returns the maximum length of a list.
merge	Removes the elements from the argument list, inserts them into the target list, and orders the new, combined set of elements in ascending order or in some other specified order.
pop_back	Deletes the element at the end of a list.
pop_front	Deletes the element at the beginning of a list.
push_back	Adds an element to the end of a list.
push_front	Adds an element to the beginning of a list.
rbegin	Returns an iterator addressing the first element in a reversed list.
remove	Erases elements in a list that match a specified value.
remove_if	Erases elements from the list for which a specified predicate is satisfied.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed list.
resize	Specifies a new size for a list.
reverse	Reverses the order in which the elements occur in a list.
size	Returns the number of elements in a list.
sort	Arranges the elements of a list in ascending order or with respect to some other order relation.
splice	Removes elements from the argument list and inserts them into the target list.
swap	Exchanges the elements of two lists.
unique	Removes adjacent duplicate elements or adjacent elements that satisfy some other binary predicate from the list.

See Also

Reference

list Class
Thread Safety in the Standard C++ Library
Standard Template Library

list Typedefs

For information about the typedefs in the list class, see list Class Members.

See Also

Reference

Standard Template Library

list::allocator_type

A type that represents the allocator class for a list object.

```
typedef Allocator allocator_type
```

Remarks

allocator_type is a synonym for the template parameter **Allocator.**

Example

See the example for get_allocator.

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::const_iterator

A type that provides a bidirectional iterator that can read a **const** element in a list.

```
typedef implementation-defined const_iterator;
```

Remarks

A type **const_iterator** cannot be used to modify the value of an element.

Example

See the example for back.

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::const_pointer

A type that provides a pointer to a **const** element in a list.

```
typedef typename Allocator::const_pointer const_pointer;
```

Remarks

A type **const_pointer** cannot be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a list object.

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::const_reference

A type that provides a reference to a **const** element stored in a list for reading and performing **const** operations.

```
typedef typename Allocator::const_reference const_reference;
```

Remarks

A type **const_reference** cannot be used to modify the value of an element.

Example

```
// list_const_ref.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );

   const list <int> c2 = c1;
   const int &i = c2.front( );
   const int &j = c2.back( );
   cout << "The first element is " << i << endl;
   cout << "The second element is " << j << endl;

   // The following line would cause an error because c2 is const
   // c2.push_back( 30 );
}
```

Output

```
The first element is 10
The second element is 20
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::const_reverse_iterator

A type that provides a bidirectional iterator that can read any **const** element in a list.

```
typedef reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type **const_reverse_iterator** cannot modify the value of an element and is used to iterate through the list in reverse.

Example

See the example for rbegin.

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::difference_type

A signed integer type that can be used to represent the number of elements of a list in a range between elements pointed to by iterators.

```
typedef implementation-defined difference_type;
```

Remarks

The **difference_type** is the type returned when subtracting or incrementing through iterators of the container. The **difference_type** is typically used to represent the number of elements in the range [*_First*, *_Last*) between the iterators *_First* and *_Last*, includes the element pointed to by *_First* and the range of elements up to, but not including, the element pointed to by *_Last*.

Note that although **difference_type** is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers like set, subtraction between iterators is only supported by random-access iterators provided by a random-access container, such as vector Class.

Example

```
// list_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <list>
#include <algorithm>

int main( )
{
   using namespace std;

   list <int> c1;
   list <int>::iterator   c1_Iter, c2_Iter;

   c1.push_back( 30 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1.push_back( 10 );
   c1.push_back( 30 );
   c1.push_back( 20 );

   c1_Iter = c1.begin( );
   c2_Iter = c1.end( );

    list <int>::difference_type df_typ1, df_typ2, df_typ3;

   df_typ1 = count( c1_Iter, c2_Iter, 10 );
   df_typ2 = count( c1_Iter, c2_Iter, 20 );
   df_typ3 = count( c1_Iter, c2_Iter, 30 );
   cout << "The number '10' is in c1 collection " << df_typ1 << " times.\n";
   cout << "The number '20' is in c1 collection " << df_typ2 << " times.\n";
   cout << "The number '30' is in c1 collection " << df_typ3 << " times.\n";
}
```

Output

```
The number '10' is in c1 collection 1 times.
The number '20' is in c1 collection 2 times.
The number '30' is in c1 collection 3 times.
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::iterator

A type that provides a bidirectional iterator that can read or modify any element in a list.

```
typedef implementation-defined iterator;
```

Remarks

A type **iterator** can be used to modify the value of an element.

Example

See the example for begin.

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::pointer

A type that provides a pointer to an element in a list.

```
typedef typename Allocator::pointer pointer;
```

Remarks

A type **pointer** can be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a list object.

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::reference

A type that provides a reference to an element stored in a list.

```
typedef typename Allocator::reference reference;
```

Example

```
// list_ref.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );

   int &i = c1.front( );
   int &j = c1.back( );
   cout << "The first element is " << i << endl;
   cout << "The second element is " << j << endl;
}
```

Output

```
The first element is 10
The second element is 20
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::reverse_iterator

A type that provides a bidirectional iterator that can read or modify an element in a reversed list.

```
typedef reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type **reverse_iterator** is used to iterate through the list in reverse.

Example

See the example for rbegin.

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::size_type

A type that counts the number of elements in a list.

```
typedef implementation-defined size_type;
```

Example

See the example for size.

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::value_type

A type that represents the data type stored in a list.

```
typedef Type value_type;
```

Remarks

value_type is a synonym for the template parameter **Type**.

Example

```
// list_value_type.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list<int>::value_type AnInt;
   AnInt = 44;
   cout << AnInt << endl;
}
```

Output

```
44
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list Member Functions

For information about the member functions in the list class, see list Class Members.

See Also

Reference

Standard Template Library

list::assign

Erases elements from a list and copies a new set of elements to a target list.

```
template<class InputIterator>
   void assign(
      InputIterator _First,
      InputIterator _Last
   );

void assign(
   size_type _Count,
   const Type& _Val
);
```

Parameters

_First

Position of the first element in the range of elements to be copied from the argument list.

_Last

Position of the first element just beyond the range of elements to be copied from the argument list.

_Count

The number of copies of an element being inserted into the list.

_Val

The value of the element being inserted into the list.

Remarks

After erasing any existing elements in the target list, **assign** either inserts a specified range of elements from the original list or from some other list into the target list or inserts copies of a new element of a specified value into the target list

Example

```
// list_assign.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list<int> c1, c2;
   list<int>::const_iterator cIter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c2.push_back( 40 );
   c2.push_back( 50 );
   c2.push_back( 60 );

   cout << "c1 =";
   for ( cIter = c1.begin( ); cIter != c1.end( ); cIter++ )
      cout << " " << *cIter;
   cout << endl;

   c1.assign( ++c2.begin( ), c2.end( ) );
   cout << "c1 =";
   for ( cIter = c1.begin( ); cIter != c1.end( ); cIter++ )
      cout << " " << *cIter;
```

```
    cout << endl;

    c1.assign( 7, 4 );
    cout << "c1 =";
    for ( cIter = c1.begin( ); cIter != c1.end( ); cIter++ )
       cout << " " << *cIter;
    cout << endl;
 }
```

Output

```
c1 = 10 20 30
c1 = 50 60
c1 = 4 4 4 4 4 4 4
```

Requirements

Header: <list>

See Also

Reference

list Class

list::assign (STL Samples)

Standard Template Library

Other Resources

list Class Members

list::back

Returns a reference to the last element of a list.

```
reference back( );
const_reference back( ) const;
```

Return Value

The last element of the list. If the list is empty, the return value is undefined.

Remarks

If the return value of **back** is assigned to a **const_reference**, the list object cannot be modified. If the return value of **back** is assigned to a **reference**, the list object can be modified.

When compiling with _SECURE_SCL 1, a runtime error will occur if you attempt to access an element in an empty list. See Checked Iterators for more information.

Example

```
// list_back.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 10 );
   c1.push_back( 11 );

   int& i = c1.back( );
   const int& ii = c1.front( );

   cout << "The last integer of c1 is " << i << endl;
   i--;
   cout << "The next-to-last integer of c1 is " << ii << endl;
}
```

Output

```
The last integer of c1 is 11
The next-to-last integer of c1 is 10
```

Requirements

Header: <list>

See Also

Reference

list Class

list::back and list::front

Standard Template Library

Other Resources

list Class Members

list::begin

Returns an iterator addressing the first element in a list.

```
const_iterator begin( ) const;
iterator begin( );
```

Return Value

A bidirectional iterator addressing the first element in the list or to the location succeeding an empty list.

Remarks

If the return value of **begin** is assigned to a **const_iterator**, the elements in the list object cannot be modified. If the return value of **begin** is assigned to an **iterator**, the elements in the list object can be modified.

Example

```
// list_begin.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;
   list <int>::const_iterator c1_cIter;

   c1.push_back( 1 );
   c1.push_back( 2 );

   c1_Iter = c1.begin( );
   cout << "The first element of c1 is " << *c1_Iter << endl;

   *c1_Iter = 20;
   c1_Iter = c1.begin( );
   cout << "The first element of c1 is now " << *c1_Iter << endl;

   // The following line would be an error because iterator is const
   // *c1_cIter = 200;
}
```

Output

```
The first element of c1 is 1
The first element of c1 is now 20
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::clear

Erases all the elements of a list.

```
void clear( );
```

Example

```
// list_clear.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main() {
   using namespace std;
   list <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   cout << "The size of the list is initially " << c1.size( ) << endl;
   c1.clear( );
   cout << "The size of list after clearing is " << c1.size( ) << endl;
}
```

Output

```
The size of the list is initially 3
The size of list after clearing is 0
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::empty

Tests if a list is empty.

```
bool empty( ) const;
```

Return Value

true if the list is empty; **false** if the list is not empty.

Example

```
// list_empty.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 10 );
   if ( c1.empty( ) )
      cout << "The list is empty." << endl;
   else
      cout << "The list is not empty." << endl;
}
```

Output

```
The list is not empty.
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::end

Returns an iterator that addresses the location succeeding the last element in a list.

```
const_iterator end( ) const;
iterator end( );
```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a list. If the list is empty, then **list::end == list::begin**.

Remarks

end is used to test whether an iterator has reached the end of its list.

Example

```
// list_end.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   c1_Iter = c1.end( );
   c1_Iter--;
   cout << "The last integer of c1 is " << *c1_Iter << endl;

   c1_Iter--;
   *c1_Iter = 400;
   cout << "The new next-to-last integer of c1 is "
        << *c1_Iter << endl;

   // If a const iterator had been declared instead with the line:
   // list <int>::const_iterator c1_Iter;
   // an error would have resulted when inserting the 400

   cout << "The list is now:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
}
```

Output

```
The last integer of c1 is 30
The new next-to-last integer of c1 is 400
The list is now: 10 400 30
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::erase

Removes an element or a range of elements in a list from specified positions.

```
iterator erase(
   iterator _Where
);
iterator erase(
   iterator _First,
   iterator _Last
);
```

Parameters

_Where

Position of the element to be removed from the list.

_First

Position of the first element removed from the list.

_Last

Position just beyond the last element removed from the list.

Return Value

A bidirectional iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the list if no such element exists.

Remarks

No reallocation occurs, so iterators and references become invalid only for the erased elements.

erase never throws an exception.

Example

```
// list_erase.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator Iter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1.push_back( 40 );
   c1.push_back( 50 );
   cout << "The initial list is:";
   for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
      cout << " " << *Iter;
   cout << endl;

   c1.erase( c1.begin( ) );
   cout << "After erasing the first element, the list becomes:";
   for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
      cout << " " << *Iter;
   cout << endl;
   Iter = c1.begin( );
   Iter++;
   c1.erase( Iter, c1.end( ) );
```

```
    cout << "After erasing all elements but the first, the list becomes: ";
    for (Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
       cout << " " << *Iter;
    cout << endl;
 }
```

Output

```
The initial list is: 10 20 30 40 50
After erasing the first element, the list becomes: 20 30 40 50
After erasing all elements but the first, the list becomes:  20
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::front

Returns a reference to the first element in a list.

```
reference front( );
const_reference front( ) const;
```

Return Value

If the list is empty, the return is undefined.

Remarks

If the return value of **front** is assigned to a **const_reference**, the list object cannot be modified. If the return value of **front** is assigned to a **reference**, the list object can be modified.

When compiling with _SECURE_SCL 1, a runtime error will occur if you attempt to access an element in an empty list. See Checked Iterators for more information.

Example

```
// list_front.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main() {
   using namespace std;
   list <int> c1;

   c1.push_back( 10 );

   int& i = c1.front();
   const int& ii = c1.front();

   cout << "The first integer of c1 is " << i << endl;
   i++;
   cout << "The first integer of c1 is " << ii << endl;
}
```

Output

```
The first integer of c1 is 10
The first integer of c1 is 11
```

Requirements

Header: <list>

See Also

Reference

list Class

list::back and list::front

Standard Template Library

Other Resources

list Class Members

list::get_allocator

Returns a copy of the allocator object used to construct a list.

```
Allocator get_allocator( ) const;
```

Return Value

The allocator used by the list.

Remarks

Allocators for the list class specify how the class manages storage. The default allocators supplied with STL container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```
// list_get_allocator.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   // The following lines declare objects
   // that use the default allocator.
   list <int> c1;
   list <int, allocator<int> > c2 = list <int, allocator<int> >( allocator<int>( ) );

   // c3 will use the same allocator class as c1
   list <int> c3( c1.get_allocator( ) );

   list<int>::allocator_type xlst = c1.get_allocator( );
   // You can now call functions on the allocator class used by c1
}
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::insert

Inserts an element or a number of elements or a range of elements into a list at a specified position.

```
iterator insert(
   iterator _Where,
   const Type& _Val
);
void insert(
   iterator _Where,
   size_type _Count,
   const Type& _Val
);
template<class InputIterator>
   void insert(
      iterator _Where,
      InputIterator _First,
      InputIterator _Last
   );
```

Parameters

_Where

The position in the target list where the first element is inserted.

_Val

The value of the element being inserted into the list.

_Count

The number of elements being inserted into the list.

_First

The position of the first element in the range of elements in the argument list to be copied.

_Last

The position of the first element beyond the range of elements in the argument list to be copied.

Return Value

The first **insert** function returns an iterator that points to the position where the new element was inserted into the list.

Remarks

Any insertion operation can be expensive.

Example

```
// list_class_insert.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1, c2;
   list <int>::iterator Iter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c2.push_back( 40 );
   c2.push_back( 50 );
   c2.push_back( 60 );
```

```
    cout << "c1 =";
    for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
       cout << " " << *Iter;
    cout << endl;

    Iter = c1.begin( );
    Iter++;
    c1.insert( Iter, 100 );
    cout << "c1 =";
    for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
       cout << " " << *Iter;
    cout << endl;

    Iter = c1.begin( );
    Iter++;
    Iter++;
    c1.insert( Iter, 2, 200 );

    cout << "c1 =";
    for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
       cout << " " << *Iter;
    cout << endl;

    c1.insert( ++c1.begin( ), c2.begin( ),--c2.end( ) );

    cout << "c1 =";
    for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
       cout << " " << *Iter;
    cout << endl;
 }
```

Output

```
c1 = 10 20 30
c1 = 10 100 20 30
c1 = 10 100 200 200 20 30
c1 = 10 40 50 100 200 200 20 30
```

Requirements

Header: <list>

See Also

Reference

list Class

list::insert (STL Samples)

Standard Template Library

Other Resources

list Class Members

list::list

Constructs a list of a specific size or with elements of a specific value or with a specific allocator or as a copy of all or part of some other list.

```
list( );
explicit list(
   const Allocator& _Al
);
explicit list(
   size_type _Count
);
list(
   size_type _Count,
   const Type& _Val
);
list(
   size_type _Count,
   const Type& _Val,
   const Allocator& _Al
);
list(
   const _list& _Right
);
template<class InputIterator>
   list(
      InputIterator _First,
      InputIterator _Last
   );
template<class InputIterator >
   list(
      InputIterator _First,
      InputIterator _Last,
      const Allocator& _Al
   );
```

Parameters

_Al

The allocator class to use with this object.

_Count

The number of elements in the list constructed.

_Val

The value of the elements in the list constructed.

_Right

The list of which the constructed list is to be a copy.

_First

The position of the first element in the range of elements to be copied.

_Last

The position of the first element beyond the range of elements to be copied.

Remarks

All constructors store an allocator object (*_Al*) and initialize the list.

get_allocator returns a copy of the allocator object used to construct a list.

The first two constructors specify an empty initial list, the second specifying the allocator type (*_Al*) to be used.

The third constructor specifies a repetition of a specified number (*_Count*) of elements of the default value for class **Type**.

The fourth and fifth constructors specify a repetition of (*_Count*) elements of value *_Val*.

The sixth constructor specifies a copy of the list *_Right*.

The last two constructors copy the range [*_First, _Last*) of a list.

None of the constructors perform any interim reallocations.

Example

```
// list_class_list.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int>::iterator c1_Iter, c2_Iter, c3_Iter, c4_Iter, c5_Iter, c6_Iter;

   // Create an empty list c0
   list <int> c0;

   // Create a list c1 with 3 elements of default value 0
   list <int> c1( 3 );

   // Create a list c2 with 5 elements of value 2
   list <int> c2( 5, 2 );

   // Create a list c3 with 3 elements of value 1 and with the
   // allocator of list c2
   list <int> c3( 3, 1, c2.get_allocator( ) );

   // Create a copy, list c4, of list c2
   list <int> c4(c2);

   // Create a list c5 by copying the range c4[_First, _Last)
   c4_Iter = c4.begin( );
   c4_Iter++;
   c4_Iter++;
   list <int> c5( c4.begin( ), c4_Iter );

   // Create a list c6 by copying the range c4[_First, _Last) and with
   // the allocator of list c2
   c4_Iter = c4.begin( );
   c4_Iter++;
   c4_Iter++;
   c4_Iter++;
   list <int> c6( c4.begin( ), c4_Iter, c2.get_allocator( ) );

   cout << "c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   cout << "c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;
```

```
    cout << "c3 =";
    for ( c3_Iter = c3.begin( ); c3_Iter != c3.end( ); c3_Iter++ )
       cout << " " << *c3_Iter;
    cout << endl;

    cout << "c4 =";
    for ( c4_Iter = c4.begin( ); c4_Iter != c4.end( ); c4_Iter++ )
       cout << " " << *c4_Iter;
    cout << endl;

    cout << "c5 =";
    for ( c5_Iter = c5.begin( ); c5_Iter != c5.end( ); c5_Iter++ )
       cout << " " << *c5_Iter;
    cout << endl;

    cout << "c6 =";
    for ( c6_Iter = c6.begin( ); c6_Iter != c6.end( ); c6_Iter++ )
       cout << " " << *c6_Iter;
    cout << endl;
 }
```

Output

```
c1 = 0 0 0
c2 = 2 2 2 2 2
c3 = 1 1 1
c4 = 2 2 2 2 2
c5 = 2 2
c6 = 2 2 2
```

Requirements

Header: <list>

See Also

Reference

list Class

list::list (STL Samples)

Standard Template Library

Other Resources

list Class Members

list::max_size

Returns the maximum length of a list.

```
size_type max_size( ) const;
```

Return Value

The maximum possible length of the list.

Example

```
// list_max_size.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::size_type i;

   i = c1.max_size( );
   cout << "Maximum possible length of the list is " << i << "." << endl;
}
```

Sample Output

The following output is for x86.

```
Maximum possible length of the list is 1073741823.
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::merge

Removes the elements from the argument list, inserts them into the target list, and orders the new, combined set of elements in ascending order or in some other specified order.

```
void merge(
   list<Type, Allocator>& _Right
);
template<class Traits>
   void merge(
      list<Type, Allocator>& _Right,
      Traits _Comp
      );
```

Parameters

_Right

The argument list to be merged with the target list.

_Comp

The comparison operator used to order the elements of the target list.

Remarks

The argument list *_Right* is merged with the target list.

Both argument and target lists must be ordered with the same comparison relation by which the resulting sequence is to be ordered. The default order for the first member function is ascending order. The second member function imposes the user-specified comparison operation *_Comp* of class **Traits**.

Example

```
// list_merge.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1, c2, c3;
   list <int>::iterator c1_Iter, c2_Iter, c3_Iter;

   c1.push_back( 3 );
   c1.push_back( 6 );
   c2.push_back( 2 );
   c2.push_back( 4 );
   c3.push_back( 5 );
   c3.push_back( 1 );

   cout << "c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   cout << "c2 =";
```

```
    for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
       cout << " " << *c2_Iter;
    cout << endl;

    c2.merge( c1 );  // Merge c1 into c2 in (default) ascending order
    c2.sort( greater<int>( ) );
    cout << "After merging c1 with c2 and sorting with >: c2 =";
    for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
       cout << " " << *c2_Iter;
    cout << endl;

    cout << "c3 =";
    for ( c3_Iter = c3.begin( ); c3_Iter != c3.end( ); c3_Iter++ )
       cout << " " << *c3_Iter;
    cout << endl;

    c2.merge( c3, greater<int>( ) );
    cout << "After merging c3 with c2 according to the '>' comparison relation: c2 =";
    for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
       cout << " " << *c2_Iter;
    cout << endl;
 }
```

Output

```
c1 = 3 6
c2 = 2 4
After merging c1 with c2 and sorting with >: c2 = 6 4 3 2
c3 = 5 1
After merging c3 with c2 according to the '>' comparison relation: c2 = 6 5 4 3 2 1
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::pop_back

Deletes the element at the end of a list.

```
void pop_back( );
```

Remarks

The last element must not be empty. **pop_back** never throws an exception.

Example

```
// list_pop_back.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 1 );
   c1.push_back( 2 );
   cout << "The first element is: " << c1.front( ) << endl;
   cout << "The last element is: " << c1.back( ) << endl;

   c1.pop_back( );
   cout << "After deleting the element at the end of the list, "
           "the last element is: " << c1.back( ) << endl;
}
```

Output

```
The first element is: 1
The last element is: 2
After deleting the element at the end of the list, the last element is: 1
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::pop_front

Deletes the element at the beginning of a list.

```
void pop_front( );
```

Remarks

The first element must not be empty. **pop_front** never throws an exception.

Example

```
// list_pop_front.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 1 );
   c1.push_back( 2 );
   cout << "The first element is: " << c1.front( ) << endl;
   cout << "The second element is: " << c1.back( ) << endl;

   c1.pop_front( );
   cout << "After deleting the element at the beginning of the list, "
         "the first element is: " << c1.front( ) << endl;
}
```

Output

```
The first element is: 1
The second element is: 2
After deleting the element at the beginning of the list, the first element is: 2
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::push_back

Adds an element to the end of a list.

```
void push_back(
   const Type& _Val
);
```

Parameters

_Val

The element added to the end of the list.

Remarks

If an exception is thrown, the list is left unaltered and the exception is rethrown.

Example

```
// list_push_back.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 1 );
   if ( c1.size( ) != 0 )
      cout << "Last element: " << c1.back( ) << endl;

   c1.push_back( 2 );
   if ( c1.size( ) != 0 )
      cout << "New last element: " << c1.back( ) << endl;
}
```

Output

```
Last element: 1
New last element: 2
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::push_front

Adds an element to the beginning of a list.

```
void push_front(
   const Type& _Val
);
```

Parameters

_Val

The element added to the beginning of the list.

Remarks

If an exception is thrown, the list is left unaltered and the exception is rethrown.

Example

```
// list_push_front.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_front( 1 );
   if ( c1.size( ) != 0 )
      cout << "First element: " << c1.front( ) << endl;

   c1.push_front( 2 );
   if ( c1.size( ) != 0 )
      cout << "New first element: " << c1.front( ) << endl;
}
```

Output

```
First element: 1
New first element: 2
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::rbegin

Returns an iterator addressing the first element in a reversed list.

```
const_reverse_iterator rbegin( ) const;
reverse_iterator rbegin( );
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed list (or addressing what had been the last element in the unreversed list).

Remarks

rbegin is used with a reversed list just as begin is used with a list.

If the return value of **rbegin** is assigned to a **const_reverse_iterator**, the list object cannot be modified. If the return value of **rbegin** is assigned to a **reverse_iterator**, the list object can be modified.

rbegin can be used to iterate through a list backwards.

Example

```
// list_rbegin.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;
   list <int>::reverse_iterator c1_rIter;

   // If the following line replaced the line above, *c1_rIter = 40;
   // (below) would be an error
   //list <int>::const_reverse_iterator c1_rIter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1_rIter = c1.rbegin( );
   cout << "The last element in the list is " << *c1_rIter << "." << endl;

   cout << "The list is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   // rbegin can be used to start an iteration through a list in
   // reverse order
   cout << "The reversed list is:";
   for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
      cout << " " << *c1_rIter;
   cout << endl;
```

```
    c1_rIter = c1.rbegin( );
    *c1_rIter = 40;
    cout << "The last element in the list is now " << *c1_rIter << "." << endl;
}
```

Output

```
The last element in the list is 30.
The list is: 10 20 30
The reversed list is: 30 20 10
The last element in the list is now 40.
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::remove

Erases elements in a list that match a specified value.

```
void remove(
   const Type& _Val
);
```

Parameters

_Val

The value which, if held by an element, will result in that element's removal from the list.

Remarks

The order of the elements remaining is not affected.

Example

```
// list_remove.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter, c2_Iter;

   c1.push_back( 5 );
   c1.push_back( 100 );
   c1.push_back( 5 );
   c1.push_back( 200 );
   c1.push_back( 5 );
   c1.push_back( 300 );

   cout << "The initial list is c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   list <int> c2 = c1;
   c2.remove( 5 );
   cout << "After removing elements with value 5, the list becomes c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;
}
```

Output

```
The initial list is c1 = 5 100 5 200 5 300
After removing elements with value 5, the list becomes c2 = 100 200 300
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::remove_if

Erases elements from a list for which a specified predicate is satisfied.

```
template<class Predicate>
void remove_if(
   Predicate _Pred
)
```

Parameters

_Pred

The unary predicate which, if satisfied by an element, results in the deletion of that element from the list.

Example

```
// list_remove_if.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

template <class T> class is_odd : public std::unary_function<T, bool>
{
public:
   bool operator( ) ( T& val )
   {
   return ( val % 2 ) == 1;
   }
};

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter, c2_Iter;

   c1.push_back( 3 );
   c1.push_back( 4 );
   c1.push_back( 5 );
   c1.push_back( 6 );
   c1.push_back( 7 );
   c1.push_back( 8 );

   cout << "The initial list is c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   list <int> c2 = c1;
   c2.remove_if( is_odd<int>( ) );

   cout << "After removing the odd elements, "
        << "the list becomes c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;
```

```
}
```

Output

```
The initial list is c1 = 3 4 5 6 7 8
After removing the odd elements, the list becomes c2 = 4 6 8
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::rend

Returns an iterator that addresses the location succeeding the last element in a reversed list.

```
const_reverse_iterator rend( ) const;
reverse_iterator rend( );
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed list (the location that had preceded the first element in the unreversed list).

Remarks

rend is used with a reversed list just as end is used with a list.

If the return value of **rend** is assigned to a **const_reverse_iterator**, the list object cannot be modified. If the return value of **rend** is assigned to a **reverse_iterator**, the list object can be modified.

rend can be used to test to whether a reverse iterator has reached the end of its list.

The value returned by **rend** should not be dereferenced.

Example

```
// list_rend.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;
   list <int>::reverse_iterator c1_rIter;

   // If the following line had replaced the line above, an error would
   // have resulted in the line modifying an element (commented below)
   // because the iterator would have been const
   // list <int>::const_reverse_iterator c1_rIter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   c1_rIter = c1.rend( );
   c1_rIter --;  // Decrementing a reverse iterator moves it forward in
                 // the list (to point to the first element here)
   cout << "The first element in the list is: " << *c1_rIter << endl;

   cout << "The list is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   // rend can be used to test if an iteration is through all of the
```

```
    // elements of a reversed list
    cout << "The reversed list is:";
    for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
       cout << " " << *c1_rIter;
    cout << endl;

    c1_rIter = c1.rend( );
    c1_rIter--;  // Decrementing the reverse iterator moves it backward
                 // in the reversed list (to the last element here)

    *c1_rIter = 40;  // This modification of the last element would have
                     // caused an error if a const_reverse iterator had
                     // been declared (as noted above)

    cout << "The modified reversed list is:";
    for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
       cout << " " << *c1_rIter;
    cout << endl;
 }
```

Output

```
The first element in the list is: 10
The list is: 10 20 30
The reversed list is: 30 20 10
The modified reversed list is: 30 20 40
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::resize

Specifies a new size for a list.

```
void resize(
   size_type _Newsize
);
void resize(
   size_type _Newsize,
   Type _Val
);
```

Parameters

_Newsize

The new size of the list.

_Val

The value of the new elements to be added to the list if the new size is larger that the original size. If the value is omitted, the new elements are assigned the default value for the class.

Remarks

If the list's size is less than the requested size, *_Newsize*, elements are added to the list until it reaches the requested size.

If the list's size is larger than the requested size, the elements closest to the end of the list are deleted until the list reaches the size *_Newsize*.

If the present size of the list is the same as the requested size, no action is taken.

size reflects the current size of the list.

Example

```
// list_resize.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   c1.resize( 4,40 );
   cout << "The size of c1 is " << c1.size( ) << endl;
   cout << "The value of the last element is " << c1.back( ) << endl;

   c1.resize( 5 );
   cout << "The size of c1 is now " << c1.size( ) << endl;
   cout << "The value of the last element is now " << c1.back( ) << endl;
```

```
    c1.resize( 2 );
    cout << "The reduced size of c1 is: " << c1.size( ) << endl;
    cout << "The value of the last element is now " << c1.back( ) << endl;
}
```

Output

```
The size of c1 is 4
The value of the last element is 40
The size of c1 is now 5
The value of the last element is now 0
The reduced size of c1 is: 2
The value of the last element is now 20
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::reverse

Reverses the order in which the elements occur in a list.

```
void reverse( );
```

Example

```
// list_reverse.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   cout << "c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   c1.reverse( );
   cout << "Reversed c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;
}
```

Output

```
c1 = 10 20 30
Reversed c1 = 30 20 10
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::size

Returns the number of elements in a list.

```
size_type size( ) const;
```

Return Value

The current length of the list.

Example

```
// list_size.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::size_type i;

   c1.push_back( 1 );
   i = c1.size( );
   cout << "List length is " << i << "." << endl;

   c1.push_back( 2 );
   i = c1.size( );
   cout << "List length is now " << i << "." << endl;
}
```

Output

```
List length is 1.
List length is now 2.
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::sort

Arranges the elements of a list in ascending order or with respect to some other user-specified order relation.

```
void sort( );
template<class Traits>
   void sort(
      Traits _Comp
   );
```

Parameters

_Comp

The comparison operator used to order successive elements.

Remarks

The first member function puts the elements in ascending order by default.

The member template function orders the elements according to the user-specified comparison operation *_Comp* of class **Traits**.

Example

```
// list_sort.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;

   c1.push_back( 20 );
   c1.push_back( 10 );
   c1.push_back( 30 );

   cout << "Before sorting: c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   c1.sort( );
   cout << "After sorting c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   c1.sort( greater<int>( ) );
   cout << "After sorting with 'greater than' operation, c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;
}
```

Output

```
Before sorting: c1 = 20 10 30
After sorting c1 = 10 20 30
After sorting with 'greater than' operation, c1 = 30 20 10
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::splice

Removes elements from the argument list and inserts them into the target list.

```
void splice(
   iterator _Where,
   list<Allocator>& _Right
);
void splice(
   iterator _Where,
   list<Allocator>& _Right,
   iterator _First
);
void splice(
   iterator _Where,
   list<Allocator>& & _Right,
   iterator _First,
   iterator _Last
);
```

Parameters

_Where

The position in the target list before which the elements of the argument list are to be inserted.

_Right

The argument list that is to be inserted into the target list.

_First

The first element in the range to be inserted from the argument list.

_Last

The first element beyond the range to be inserted from the argument list.

Remarks

The first member function inserts all elements in the argument list before the element located at *_Where* in the target list. It also removes all elements from the argument list.

The second member function removes the element pointed to by *_First* in the argument list and inserts it before the element in the target list pointed to by *_Where*.

The third member function inserts the range designated by [*_First*, *_Last*) from the argument list before the element in the target list pointed to by *_Where*. It also removes the range inserted from the argument list.

In all cases, only iterators or references that point at spliced elements become invalid.

Example

```
// list_splice.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
```

```
{
   using namespace std;
   list <int> c1, c2, c3, c4;
   list <int>::iterator c1_Iter, c2_Iter, w_Iter, f_Iter, l_Iter;

   c1.push_back( 10 );
   c1.push_back( 11 );
   c2.push_back( 12 );
   c2.push_back( 20 );
   c2.push_back( 21 );
   c3.push_back( 30 );
   c3.push_back( 31 );
   c4.push_back( 40 );
   c4.push_back( 41 );
   c4.push_back( 42 );

   cout << "c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   cout << "c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;

   w_Iter = c2.begin( );
   w_Iter++;
   c2.splice( w_Iter,c1 );
   cout << "After splicing c1 into c2: c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;

   f_Iter = c3.begin( );
   c2.splice( w_Iter,c3, f_Iter );
   cout << "After splicing the first element of c3 into c2: c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;

   f_Iter = c4.begin( );
   l_Iter = c4.end( );
   l_Iter--;
   c2.splice( w_Iter,c4, f_Iter, l_Iter );
   cout << "After splicing a range of c4 into c2: c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;
}
```

Output

```
c1 = 10 11
c2 = 12 20 21
After splicing c1 into c2: c2 = 12 10 11 20 21
After splicing the first element of c3 into c2: c2 = 12 10 11 30 20 21
After splicing a range of c4 into c2: c2 = 12 10 11 30 40 41 20 21
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::swap

Exchanges the elements of two lists.

```
void swap(
   list<Type, Allocator>& _Right
);
friend void swap(
   list<Type, Allocator>& _Left,
   list<Type, Allocator>& _Right
)
```

Parameters

_Right

The list providing the elements to be swapped, or the list whose elements are to be exchanged with those of the list *_Left*.

_Left

A list whose elements are to be exchanged with those of the list *_Right*.

Example

```
// list_swap.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1, c2, c3;
   list <int>::iterator c1_Iter;

   c1.push_back( 1 );
   c1.push_back( 2 );
   c1.push_back( 3 );
   c2.push_back( 10 );
   c2.push_back( 20 );
   c3.push_back( 100 );

   cout << "The original list c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   c1.swap( c2 );

   cout << "After swapping with c2, list c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   swap( c1,c3 );

   cout << "After swapping with c3, list c1 is:";
```

```
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
       cout << " " << *c1_Iter;
    cout << endl;
 }
```

Output

```
The original list c1 is: 1 2 3
After swapping with c2, list c1 is: 10 20
After swapping with c3, list c1 is: 100
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

list::unique

Removes adjacent duplicate elements or adjacent elements that satisfy some other binary predicate from a list.

```
void unique( );
template<class BinaryPredicate>
      void unique(
      BinaryPredicate _Pred
   );
```

Parameters

_Pred

The binary predicate used to compare successive elements.

Remarks

This function assumes that the list is sorted, so that all duplicate elements are adjacent. Duplicates that are not adjacent will not be deleted.

The first member function removes every element that compares equal to its preceding element.

The second member function removes every element that satisfies the predicate function *_Pred* when compared with its preceding element.

Example

```
// list_unique.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter, c2_Iter,c3_Iter;
   not_equal_to<int> mypred;

   c1.push_back( -10 );
   c1.push_back( 10 );
   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 20 );
   c1.push_back( -10 );

   cout << "The initial list is c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   list <int> c2 = c1;
   c2.unique( );
   cout << "After removing successive duplicate elements, c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;
```

```
    list <int> c3 = c2;
    c3.unique( mypred );
    cout << "After removing successive unequal elements, c3 =";
    for ( c3_Iter = c3.begin( ); c3_Iter != c3.end( ); c3_Iter++ )
       cout << " " << *c3_Iter;
    cout << endl;
 }
```

Output

```
The initial list is c1 = -10 10 10 20 20 -10
After removing successive duplicate elements, c2 = -10 10 20 -10
After removing successive unequal elements, c3 = -10 -10
```

Requirements

Header: <list>

See Also

Reference

list Class

Standard Template Library

Other Resources

list Class Members

<locale>

Defines template classes and functions that C++ programs can use to encapsulate and manipulate different cultural conventions regarding the representation and formatting of numeric, monetary, and calendric data, including internationalization support for character classification and string collation.

For a list of all members of this header, see <locale> Members.

```
#include <locale>
```

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<locale> Members

<locale> Members

Reference

Functions

has_facet	Tests if a particular facet is stored in a specified locale.
isalnum	Tests whether an element in a locale is an alphabetic or a numeric character.
isalpha	Tests whether an element in a locale is alphabetic character.
iscntrl	Tests whether an element in a locale is a control character.
isdigit	Tests whether an element in a locale is a numeric character.
isgraph	Tests whether an element in a locale is an alphanumeric or punctuation character.
islower	Tests whether an element in a locale is lower case.
isprint	Tests whether an element in a locale is a printable character.
ispunct	Tests whether an element in a locale is a punctuation character.
isspace	Tests whether an element in a locale is a whitespace character.
isupper	Tests whether an element in a locale is upper case.
isxdigit	Tests whether an element in a locale is a character used to represent a hexadecimal number.
tolower	Converts a character to lower case.
toupper	Converts a character to upper case.
use_facet	Returns a reference to a facet of a specified type stored in a locale.

Classes

codecvt	A template class that provides a facet used to convert between internal and external character encodings .
codecvt_base	A base class for the codecvt class that is used to define an enumeration type referred to as **result**, used a s the return type for the facet member functions to indicate the result of a conversion.
codecvt_byname	A derived template class that describes an object that can serve as a collate facet of a given locale, enabli ng the retrieval of information specific to a cultural area concerning conversions.
collate	A collate template class that provides a facet that handles string sorting conventions.
collate_byname	A derived template class that describes an object that can serve as a collate facet of a given locale, enabli ng the retrieval of information specific to a cultural area concerning string sorting conventions.
ctype	A template class that provides a facet that is used to classify characters, convert from upper- and lowerc ase and between the native character set and that set used by the locale.

ctype<char>	A class that is an explicit specialization of template class **ctype<CharType>** to type **char**, describing an object that can serve as a locale facet to characterize various properties of a character of type **char**.
ctype_base	A base class for the ctype class that is used to define enumeration types used to classify or test characters either individually or within entire ranges.
ctype_byname	A derived template class that describes an object that can serve as a ctype facet of a given locale, enabling the classification of characters and conversion of characters between case and native and locale specified character sets.
locale	A class that describes a locale object that encapsulates culture-specific information as a set of facets that collectively define a specific localized environment.
messages	A template class that describes an object that can serve as a locale facet to retrieve localized messages from a catalog of internationalized messages for a given locale.
messages_base	A base class that describes an **int** type for the catalog of messages.
messages_byname	A derived template class that describes an object that can serve as a message facet of a given locale, enabling the retrieval of localized messages.
money_base	A base class for the ctype class that is used to define enumeration types used to classify or test characters either individually or within entire ranges.
money_get	A template class that describes an object that can serve as a locale facet to control conversions of sequences of type **CharType** to monetary values.
money_put	A template class that describes an object that can serve as a locale facet to control conversions of monetary values to sequences of type **CharType**.
moneypunct	A template class that describes an object that can serve as a locale facet to describe the sequences of type **CharType** used to represent a monetary input field or a monetary output field.
moneypunct_byname	A derived template class that describes an object that can serve as a moneypunct facet of a given locale enabling the formatting monetary input or output fields.
num_get	A template class that describes an object that can serve as a locale facet to control conversions of sequences of type **CharType** to numeric values.
num_put	A template class that describes an object that can serve as a locale facet to control conversions of numeric values to sequences of type **CharType**.
numpunct	A template class that describes an object that can serve as a local facet to describe the sequences of type **CharType** used to represent information about the formatting and punctuation of numeric and Boolean expressions.
numpunct_byname	A derived template class that describes an object that can serve as a moneypunct facet of a given locale enabling the formatting and punctuation of numeric and Boolean expressions.
time_base	A class that serves as a base class for facets of template class time_get, defining just the enumerated type dateorder and several constants of this type.
time_get	A template class that describes an object that can serve as a locale facet to control conversions of sequences of type **CharType** to time values.

time_get_byname	A derived template class that describes an object that can serve as a locale facet of type time_get<**CharType**, **InputIterator**>.
time_put	A template class that describes an object that can serve as a locale facet to control conversions of time values to sequences of type **CharType**.
time_put_byname	A derived template class that describes an object that can serve as a locale facet of type **time_put**<**CharType**, **OutputIterator**>.

See Also

Reference

<locale>

Thread Safety in the Standard C++ Library

<locale> Functions

For information about the functions in <locale>, see <locale> Members.

has_facet

Tests if a particular facet is stored in a specified locale.

```
template<class Facet>
   bool has_facet(
      const locale& _Loc
   );
```

Parameters

_Loc

The locale to be tested for the presence of a facet.

Return Value

true if the locale has the facet tested for; **false** if it does not.

Remarks

The template function is useful for checking whether nonmandatory facets are listed in a locale before **use_facet** is called to avoid the exception that would be thrown if it were not present.

Example

```
// locale_has_facet.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result = has_facet <ctype<char> > ( loc );
   cout << result << endl;
}
```

Output

```
1
```

Requirements

Header: <locale>

See Also

Other Resources

<locale> Members

isalnum

Tests whether an element in a locale is an alphabetic or a numeric character.

```
template<Class CharType>
   bool isalnum(
      CharType _Ch,
      const locale& _Loc
   )
```

Parameters

_Ch

The alphanumeric element to be tested.

_Loc

The locale containing the alphanumeric element to be tested.

Return Value

true if the element tested is alphanumeric; **false** if it is not.

Example

```
// locale_isalnum.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = isalnum ( 'L', loc);
   bool result2 = isalnum ( '@', loc);
   bool result3 = isalnum ( '3', loc);

   if ( result1 )
      cout << "The character 'L' in the locale is "
           << "alphanumeric." << endl;
   else
      cout << "The character 'L' in the locale is "
           << " not alphanumeric." << endl;

   if ( result2 )
      cout << "The character '@' in the locale is "
           << "alphanumeric." << endl;
   else
      cout << "The character '@' in the locale is "
           << " not alphanumeric." << endl;

   if ( result3 )
      cout << "The character '3' in the locale is "
           << "alphanumeric." << endl;
   else
      cout << "The character '3' in the locale is "
           << " not alphanumeric." << endl;
}
```

Output

```
The character 'L' in the locale is alphanumeric.
The character '@' in the locale is  not alphanumeric.
```

```
The character '3' in the locale is alphanumeric.
```

Requirements

Header: <locale>

See Also

Other Resources

<locale> Members

isalpha

Tests whether an element in a locale is an alphabetic character.

```
template<Class CharType>
   bool isalpha(
      CharType _Ch,
      const locale& _Loc
   )
```

Parameters

_Ch

The element to be tested.

_Loc

The locale containing the alphabetic element to be tested.

Return Value

true if the element tested is alphabetic; **false** if it is not.

Remarks

The template function returns use_facet<ctype<**CharType**> >(*_Loc*).is(**ctype**<**CharType**>::**alpha**, *_Ch*).

Example

```
// locale_isalpha.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = isalpha ( 'L', loc);
   bool result2 = isalpha ( '@', loc);
   bool result3 = isalpha ( '3', loc);

   if ( result1 )
      cout << "The character 'L' in the locale is "
           << "alphabetic." << endl;
   else
      cout << "The character 'L' in the locale is "
           << " not alphabetic." << endl;

   if ( result2 )
      cout << "The character '@' in the locale is "
           << "alphabetic." << endl;
   else
      cout << "The character '@' in the locale is "
           << " not alphabetic." << endl;

   if ( result3 )
      cout << "The character '3' in the locale is "
           << "alphabetic." << endl;
   else
      cout << "The character '3' in the locale is "
           << " not alphabetic." << endl;
}
```

Output

```
The character 'L' in the locale is alphabetic.
The character '@' in the locale is  not alphabetic.
The character '3' in the locale is  not alphabetic.
```

Requirements

Header: <locale>

See Also

Other Resources

<locale> Members

iscntrl

Tests whether an element in a locale is a control character.

```
template<Class CharType>
   bool iscntrl(
      CharType _Ch,
      const locale& _Loc
   )
```

Parameters

_Ch

The element to be tested.

_Loc

The locale containing the element to be tested.

Return Value

true if the element tested is a control character; **false** if it is not.

Remarks

The template function returns use_facet<ctype<**CharType**> >(*_Loc*).is(**ctype**<**CharType**>::**cntrl**, *_Ch*).

Example

```
// locale_iscntrl.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = iscntrl ( 'L', loc );
   bool result2 = iscntrl ( '\n', loc );
   bool result3 = iscntrl ( '\t', loc );

   if ( result1 )
      cout << "The character 'L' in the locale is "
           << "a control character." << endl;
   else
      cout << "The character 'L' in the locale is "
           << " not a control character." << endl;

   if ( result2 )
      cout << "The character-set 'backslash-n' in the locale\n is "
           << "a control character." << endl;
   else
      cout << "The character-set 'backslash-n' in the locale\n is "
           << " not a control character." << endl;

   if ( result3 )
      cout << "The character-set 'backslash-t' in the locale\n is "
           << "a control character." << endl;
   else
      cout << "The character-set 'backslash-n' in the locale \n is "
           << " not a control character." << endl;
}
```

Output

```
The character 'L' in the locale is  not a control character.
The character-set 'backslash-n' in the locale
 is a control character.
The character-set 'backslash-t' in the locale
 is a control character.
```

Requirements

Header: <locale>

See Also

Other Resources

<locale> Members

isdigit

Tests whether an element in a locale is a numeric character.

```
template<Class CharType>
   bool isdigit(
      CharType _Ch,
      const locale& _Loc
   )
```

Parameters

_Ch

The element to be tested.

_Loc

The locale containing the element to be tested.

Return Value

true if the element tested is a numeric character; **false** if it is not.

Remarks

The template function returns use_facet<ctype<**CharType**> >(*_Loc*).is(**ctype**<**CharType**>::**digit**, *_Ch*).

Example

```
// locale_is_digit.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = isdigit ( 'L', loc );
   bool result2 = isdigit ( '@', loc );
   bool result3 = isdigit ( '3', loc );

   if ( result1 )
      cout << "The character 'L' in the locale is "
           << "a numeric character." << endl;
   else
      cout << "The character 'L' in the locale is "
           << " not a numeric character." << endl;

   if ( result2 )
      cout << "The character '@' in the locale is "
           << "a numeric character." << endl;
   else
      cout << "The character '@' in the locale is "
           << " not a numeric character." << endl;

   if ( result3 )
      cout << "The character '3' in the locale is "
           << "a numeric character." << endl;
   else
      cout << "The character '3' in the locale is "
           << " not a numeric character." << endl;
}
```

Output

```
The character 'L' in the locale is  not a numeric character.
The character '@' in the locale is  not a numeric character.
The character '3' in the locale is a numeric character.
```

Requirements

Header: <locale>

See Also

Other Resources

<locale> Members

isgraph

Tests whether an element in a locale is an alphanumeric or punctuation character.

```
template<Class CharType>
   bool isgraph(
      CharType _Ch,
      const locale& _Loc
   )
```

Parameters

_Ch

The element to be tested.

_Loc

The locale containing the element to be tested.

Return Value

true if the element tested is an alphanumeric or a punctuation character; **false** if it is not.

Remarks

The template function returns use_facet<ctype<**CharType**> >(*_Loc*).is(**ctype**<**CharType**>::**graph**, *_Ch*).

Example

```
// locale_is_graph.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = isgraph ( 'L', loc );
   bool result2 = isgraph ( '\t', loc );
   bool result3 = isgraph ( '.', loc );

   if ( result1 )
      cout << "The character 'L' in the locale is\n "
           << "an alphanumeric or punctuation character." << endl;
   else
      cout << "The character 'L' in the locale is\n "
           << " not an alphanumeric or punctuation character." << endl;

   if ( result2 )
      cout << "The character 'backslash-t' in the locale is\n "
           << "an alphanumeric or punctuation character." << endl;
   else
      cout << "The character 'backslash-t' in the locale is\n "
           << "not an alphanumeric or punctuation character." << endl;

   if ( result3 )
      cout << "The character '.' in the locale is\n "
           << "an alphanumeric or punctuation character." << endl;
   else
      cout << "The character '.' in the locale is\n "
           << " not an alphanumeric or punctuation character." << endl;
}
```

Output

```
The character 'L' in the locale is
 an alphanumeric or punctuation character.
The character 'backslash-t' in the locale is
 not an alphanumeric or punctuation character.
The character '.' in the locale is
 an alphanumeric or punctuation character.
```

Requirements

Header: <locale>

See Also

Other Resources

<locale> Members

islower

Tests whether an element in a locale is lower case.

```
template<Class CharType>
   bool islower(
      CharType _Ch,
      const locale& _Loc
   )
```

Parameters

_Ch

The element to be tested.

_Loc

The locale containing the element to be tested.

Return Value

true if the element tested is a lowercase character; **false** if it is not.

Remarks

The template function returns use_facet<ctype<**CharType**> >(*_Loc*). is(**ctype**<**CharType**>::**lower**, *_Ch*).

Example

```
// locale_islower.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = islower ( 'L', loc );
   bool result2 = islower ( 'n', loc );
   bool result3 = islower ( '3', loc );

   if ( result1 )
      cout << "The character 'L' in the locale is "
           << "a lowercase character." << endl;
   else
      cout << "The character 'L' in the locale is "
           << " not a lowercase character." << endl;

   if ( result2 )
      cout << "The character 'n' in the locale is "
           << "a lowercase character." << endl;
   else
      cout << "The character 'n' in the locale is "
           << " not a lowercase character." << endl;

   if ( result3 )
      cout << "The character '3' in the locale is "
           << "a lowercase character." << endl;
   else
      cout << "The character '3' in the locale is "
           << " not a lowercase character." << endl;
}
```

Output

```
The character 'L' in the locale is  not a lowercase character.
The character 'n' in the locale is a lowercase character.
The character '3' in the locale is  not a lowercase character.
```

Requirements

Header: <locale>

See Also

Other Resources

<locale> Members

isprint

Tests whether an element in a locale is a printable character.

```
template<Class CharType>
   bool isprint(
      CharType _Ch,
      const locale& _Loc
   )
```

Parameters

_Ch

The element to be tested.

_Loc

The locale containing the element to be tested.

Return Value

true if the element tested is a printable; **false** if it is not.

Remarks

The template function returns use_facet<ctype<**CharType**> >(*_Loc*).is(**ctype**<**CharType**>::**print**, *_Ch*).

Example

```
// locale_isprint.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc ( "German_Germany" );

   bool result1 = isprint ( 'L', loc );
   if ( result1 )
      cout << "The character 'L' in the locale is "
           << "a printable character." << endl;
   else
      cout << "The character 'L' in the locale is "
           << " not a printable character." << endl;

   bool result2 = isprint( '\t', loc );
   if ( result2 )
      cout << "The character 'backslash-t' in the locale is "
           << "a printable character." << endl;
   else
      cout << "The character 'backslash-t' in the locale is "
           << " not a printable character." << endl;

   bool result3 = isprint( '\n', loc );
   if ( result3 )
      cout << "The character 'backslash-n' in the locale is "
           << "a printable character." << endl;
   else
      cout << "The character 'backslash-n' in the locale is "
           << " not a printable character." << endl;
}
```

Output

```
The character 'L' in the locale is a printable character.
The character 'backslash-t' in the locale is a printable character.
The character 'backslash-n' in the locale is  not a printable character.
```

Requirements

Header: <locale>

See Also

Other Resources

<locale> Members

ispunct

Tests whether an element in a locale is a punctuation character.

```
template<Class CharType>
   bool ispunct(
      CharType _Ch,
      const locale& _Loc
   )
```

Parameters

_Ch

The element to be tested.

_Loc

The locale containing the element to be tested.

Return Value

true if the element tested is a punctuation character; **false** if it is not.

Remarks

The template function returns use_facet<ctype<**CharType**> >(*_Loc*).is(**ctype**<**CharType**>::**punct**, *_Ch*).

Example

```
// locale_ispunct.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = ispunct ( 'L', loc );
   bool result2 = ispunct ( ';', loc );
   bool result3 = ispunct ( '*', loc );

   if ( result1 )
      cout << "The character 'L' in the locale is "
           << "a punctuation character." << endl;
   else
      cout << "The character 'L' in the locale is "
           << " not a punctuation character." << endl;

   if ( result2 )
      cout << "The character ';' in the locale is "
           << "a punctuation character." << endl;
   else
      cout << "The character ';' in the locale is "
           << " not a punctuation character." << endl;

   if ( result3 )
      cout << "The character '*' in the locale is "
           << "a punctuation character." << endl;
   else
      cout << "The character '*' in the locale is "
           << " not a punctuation character." << endl;
}
```

Output

```
The character 'L' in the locale is  not a punctuation character.
The character ';' in the locale is a punctuation character.
The character '*' in the locale is a punctuation character.
```

Requirements

Header: <locale>

See Also

Other Resources

<locale> Members

isspace

Tests whether an element in a locale is a whitespace character.

```
template<Class CharType>
   bool isspace(
      CharType _Ch,
      const locale& _Loc
   )
```

Parameters

_Ch

The element to be tested.

_Loc

The locale containing the element to be tested.

Return Value

true if the element tested is a whitespace character; **false** if it is not.

Remarks

The template function returns use_facet<ctype<**CharType**> >(*_Loc*).is(**ctype**<**CharType**>::**space**, *_Ch*).

Example

```
// locale_isspace.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = isspace ( 'L', loc );
   bool result2 = isspace ( '\n', loc );
   bool result3 = isspace ( ' ', loc );

   if ( result1 )
      cout << "The character 'L' in the locale is "
           << "a whitespace character." << endl;
   else
      cout << "The character 'L' in the locale is "
           << " not a whitespace character." << endl;

   if ( result2 )
      cout << "The character 'backslash-n' in the locale is "
           << "a whitespace character." << endl;
   else
      cout << "The character 'backslash-n' in the locale is "
           << " not a whitespace character." << endl;

   if ( result3 )
      cout << "The character ' ' in the locale is "
           << "a whitespace character." << endl;
   else
      cout << "The character ' ' in the locale is "
           << " not a whitespace character." << endl;
}
```

Output

```
The character 'L' in the locale is  not a whitespace character.
The character 'backslash-n' in the locale is a whitespace character.
The character ' ' in the locale is a whitespace character.
```

Requirements

Header: <locale>

See Also

Other Resources

<locale> Members

isupper

Tests whether an element in a locale is in upper case.

```
template<Class CharType>
   bool isupper(
      CharType _Ch,
      const locale& _Loc
   )
```

Parameters

_Ch

The element to be tested.

_Loc

The locale containing the element to be tested.

Return Value

true if the element tested is an uppercase character; **false** if it is not.

Remarks

The template function returns use_facet<ctype<**CharType**> >(*_Loc*).is(**ctype**<**CharType**>::**upper**, *_Ch*).

Example

```
// locale_isupper.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = isupper ( 'L', loc );
   bool result2 = isupper ( 'n', loc );
   bool result3 = isupper ( '3', loc );

   if ( result1 )
      cout << "The character 'L' in the locale is "
           << "a uppercase character." << endl;
   else
      cout << "The character 'L' in the locale is "
           << " not a uppercase character." << endl;

   if ( result2 )
      cout << "The character 'n' in the locale is "
           << "a uppercase character." << endl;
   else
      cout << "The character 'n' in the locale is "
           << " not a uppercase character." << endl;

   if ( result3 )
      cout << "The character '3' in the locale is "
           << "a uppercase character." << endl;
   else
      cout << "The character '3' in the locale is "
           << " not a uppercase character." << endl;
}
```

Output

```
The character 'L' in the locale is a uppercase character.
The character 'n' in the locale is  not a uppercase character.
The character '3' in the locale is  not a uppercase character.
```

Requirements

Header: <locale>

See Also

Other Resources

<locale> Members

isxdigit

Tests whether an element in a locale is a character used to represent a hexadecimal number.

```
template<Class CharType>
   bool isxdigit(
      CharType _Ch,
      const locale& _Loc
   )
```

Parameters

_Ch

The element to be tested.

_Loc

The locale containing the element to be tested.

Return Value

true if the element tested is a character used to represent a hexadecimal number; **false** if it is not.

Remarks

The template function returns use_facet<ctype< **CharType**> >(*_Loc*).is(**ctype**<**CharType**>::**xdigit**, *_Ch*).

Hexadecimal digits use base 16 to represent numbers, using the numbers 0 through 9 plus case-insensitive letters A through F to represent the decimal numbers 0 through 15.

Example

```
// locale_isxdigit.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = isxdigit ( '5', loc );
   bool result2 = isxdigit ( 'd', loc );
   bool result3 = isxdigit ( 'q', loc );

   if ( result1 )
      cout << "The character '5' in the locale is "
           << "a hexidecimal digit-character." << endl;
   else
      cout << "The character '5' in the locale is "
           << " not a hexidecimal digit-character." << endl;

   if ( result2 )
      cout << "The character 'd' in the locale is "
           << "a hexidecimal digit-character." << endl;
   else
      cout << "The character 'd' in the locale is "
           << " not a hexidecimal digit-character." << endl;

   if ( result3 )
      cout << "The character 'q' in the locale is "
           << "a hexidecimal digit-character." << endl;
   else
      cout << "The character 'q' in the locale is "
           << " not a hexidecimal digit-character." << endl;
```

```
}
```

Output

```
The character '5' in the locale is a hexidecimal digit-character.
The character 'd' in the locale is a hexidecimal digit-character.
The character 'q' in the locale is  not a hexidecimal digit-character.
```

Requirements

Header: <locale>

See Also

Other Resources

<locale> Members

tolower

Converts a character to lower case.

```
template<Class CharType>
   CharType tolower(
      CharType _Ch,
      const locale& _Loc
   )
```

Parameters

_Ch

The character to be converted to lower case.

_Loc

The locale containing the character to be converted.

Return Value

The character converted to lower case.

Remarks

The template function returns use_facet<ctype<**CharType**> >(*_Loc*).tolower(*_Ch*).

Example

```
// locale_tolower.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   char result1 = tolower ( 'H', loc );
   cout << "The lower case of 'H' in the locale is: "
        << result1 << "." << endl;
   char result2 = tolower ( 'h', loc );
   cout << "The lower case of 'h' in the locale is: "
        << result2 << "." << endl;
   char result3 = tolower ( '$', loc );
   cout << "The lower case of '$' in the locale is: "
        << result3 << "." << endl;
}
```

Output

```
The lower case of 'H' in the locale is: h.
The lower case of 'h' in the locale is: h.
The lower case of '$' in the locale is: $.
```

Requirements

Header: <locale>

See Also

Other Resources

<locale> Members

toupper

Converts a character to upper case.

```
template<Class CharType>
   CharType toupper(
      CharType _Ch,
      const locale& _Loc
   )
```

Parameters

_Ch

The character to be converted to upper case.

_Loc

The locale containing the character to be converted.

Return Value

The character converted to upper case.

Remarks

The template function returns use_facet<ctype<**CharType**> >(*_Loc*).toupper(*_Ch*).

Example

```
// locale_toupper.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   char result1 = toupper ( 'h', loc );
   cout << "The upper case of 'h' in the locale is: "
        << result1 << "." << endl;
   char result2 = toupper ( 'H', loc );
   cout << "The upper case of 'H' in the locale is: "
        << result2 << "." << endl;
   char result3 = toupper ( '$', loc );
   cout << "The upper case of '$' in the locale is: "
        << result3 << "." << endl;
}
```

Output

```
The upper case of 'h' in the locale is: H.
The upper case of 'H' in the locale is: H.
The upper case of '$' in the locale is: $.
```

Requirements

Header: <locale>

See Also

Other Resources

<locale> Members

use_facet

Returns a reference to a facet of a specified type stored in a locale.

```
template<class Facet>
   const Facet& use_facet(
      const locale& _Loc
   );
```

Parameters

_Loc

The const locale containing the type of facet being referenced.

Return Value

A reference to the facet of class **Facet** contained within the argument locale.

Remarks

The reference to the facet returned by the template function remains valid as long as any copy of the containing locale exists. If no such facet object of class **Facet** is listed in the argument locale, the function throws a **bad_cast** exception.

Example

```
// locale_use_facet.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc1 ( "German_Germany" ), loc2 ( "English_Australia" );
   bool result1 = use_facet<ctype<char> > ( loc1 ).is(
   ctype_base::alpha, 'a'
);
   bool result2 = use_facet<ctype<char> > ( loc2 ).is( ctype_base::alpha, '!'
   );

   if ( result1 )
      cout << "The character 'a' in locale loc1 is alphabetic."
           << endl;
   else
      cout << "The character 'a' in locale loc1 is not alphabetic."
           << endl;

   if ( result2 )
      cout << "The character '!' in locale loc2 is alphabetic."
           << endl;
   else
      cout << "The character '!' in locale loc2 is not alphabetic."
           << endl;
}
```

Output

```
The character 'a' in locale loc1 is alphabetic.
The character '!' in locale loc2 is not alphabetic.
```

Requirements

Header: <locale>

See Also

Other Resources

<locale> Classes

For information about the classes in <locale>, see <locale> Members.

codecvt Class

A template class that describes an object that can serve as a locale facet that is able to control conversions between a sequence of values used to encode characters within the program and a sequence of values used to encode characters outside the program.

For a list of all members of this type, see codecvt Members.

```
template <
   class CharType,
   class Byte,
   class StateType,
>
class codecvt : public codecvt_base
```

Parameters

CharType

The type used within a program to encode characters.

Byte

A type used to encode characters outside a program.

StateType

A type that can be used to represent intermediate states of a conversion between internal and external types of character representations.

Remarks

The internal encoding uses a representation with a fixed number of bytes per character, usually either type **char** or type **wchar_t**.

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id.**

The template versions of do_in and do_out always return **codecvt_base::noconv**. The Standard C++ Library defines an explicit specialization, however, that is more useful:

template<> **codecvt**<**wchar_t**, **char**, **mbstate_t**>

which converts between **wchar_t** and **char** sequences.

Requirements

Header: <locale>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

codecvt Members

<locale> Members

codecvt Members

The following tables list the members of the codecvt Class. For a list of members inherited by the **codecvt** class, see codecvt_base Class.

Reference

Constructors

codecvt	The constructor for objects of class **codecvt** that serves as a locale facet to handle conversions.

Typedefs

extern_type	A character type that is used for external representations.
intern_type	A character type that is used for internal representations.
state_type	A character type that is used to represent intermediate states during conversions between internal and external re presentations.

Member Functions

always_noconv	Tests whether no conversions need be done.
do_always_noconv	A virtual function called to test whether no conversions need be done.
do_encoding	A virtual function that tests if the encoding of the **Byte** stream is state dependent, whether the ratio betwee n the **Byte**s used and the **CharType**s produced is constant, and, if so, determines the value of that ratio.
do_in	A virtual function called to convert a sequence of internal **Byte**s to a sequence of external **CharType**s.
do_length	A virtual function that determines how many **Byte**s from a given sequence of external **Byte**s produce not more than a given number of internal **CharType**s and returns that number of **Byte**s.
do_max_length	A virtual function that returns the maximum number of external Bytes necessary to produce one internal **C harType**.
do_out	A virtual function called to convert a sequence of internal **CharType**s to a sequence of external Bytes.
do_unshift	A virtual function called to provide the **Byte**s needed in a state-dependent conversion to complete the last character in a sequence of **Byte**s.
in	Converts an external representation of a sequence of **Byte**s to an internal representation of a sequence of **CharType**s.
length	Determines how many **Byte**s from a given sequence of external **Byte**s produce not more than a given num ber of internal **CharType**s and returns that number of **Byte**s.
encoding	Tests if the encoding of the **Byte** stream is state dependent, whether the ratio between the **Byte**s used and the **CharType**s produced is constant, and, if so, determines the value of that ratio.
max_length	Returns the maximum number of external **Byte**s necessary to produce one internal **CharType**.
out	Converts a sequence of internal **CharType**s to a sequence of external **Byte**s.
unshift	Provides the external **Byte**s needed in a state-dependent conversion to complete the last character in the s equence of **Byte**s.

See Also

Reference

codecvt Class

codecvt_base Class

Thread Safety in the Standard C++ Library

codecvt Typedefs

For information about the typedefs in the codecvt class, see codecvt Members.

codecvt::extern_type

A character type that is used for external representations.

```
typedef Byte extern_type;
```

Remarks

The type is a synonym for the template parameter **Byte**.

Requirements

Header: <locale>

See Also

Reference

codecvt Class

Other Resources

codecvt Members

codecvt::intern_type

A character type that is used for internal representations.

```
typedef CharType intern_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

Requirements

Header: <locale>

See Also

Reference

codecvt Class

Other Resources

codecvt Members

codecvt::state_type

A character type that is used to represent intermediate states during conversions between internal and external representations.

```
typedef StateType state_type;
```

Remarks

The type is a synonym for the template parameter **StateType**.

Requirements

Header: <locale>

See Also

Reference

codecvt Class

Other Resources

codecvt Members

codecvt Member Functions

For information about the member functions in the codecvt class, see codecvt Members.

codecvt::always_noconv

Tests whether no conversions need be done.

```
bool always_noconv( ) const throw( );
```

Return Value

A Boolean value that is **true** if no conversions need be done; **false** is at least one needs to be done.

Remarks

The member function returns do_always_noconv.

Example

```
// codecvt_always_noconv.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = use_facet<codecvt<char, char, mbstate_t> >
      ( loc ).always_noconv( );

   if ( result1 )
      cout << "No conversion is needed." << endl;
   else
      cout << "At least one conversion is required." << endl;

   bool result2 = use_facet<codecvt<wchar_t, char, mbstate_t> >
      ( loc ).always_noconv( );

   if ( result2 )
      cout << "No conversion is needed." << endl;
   else
      cout << "At least one conversion is required." << endl;
}
```

Output

```
No conversion is needed.
At least one conversion is required.
```

Requirements

Header: <locale>

See Also

Reference

codecvt Class

Other Resources

codecvt Members

codecvt::codecvt

The constructor for objects of class codecvt that serves as a locale facet to handle conversions.

```
explicit codecvt(
   size_t _Refs = 0
);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 0: These values are not defined.

The constructor initializes its **locale::facet** base object with **locale::**facet(*_Refs*).

Requirements

Header: <locale>

See Also

Reference

codecvt Class

Other Resources

codecvt Members

codecvt::do_always_noconv

A virtual function called to test whether no conversions need be done.

```
virtual bool do_always_noconv( ) const throw( );
```

Return Value

The protected virtual member function returns **true** only if every call to do_in or do_out returns **noconv**.

The template version always returns **true**.

Example

See the example for always_noconv, which calls **do_always_noconv**.

Requirements

Header: <locale>

See Also

Reference

codecvt Class

Other Resources

codecvt Members

codecvt::do_encoding

A virtual function that tests if the encoding of the **Byte** stream is state dependent, whether the ratio between the **Byte**s used and the **CharType**s produced is constant and, if so, determines the value of that ratio.

```
virtual int do_encoding( ) const throw( );
```

Return Value

The protected virtual member function returns:

- –1, if the encoding of sequences of type **extern_type** is state dependent.
- 0, if the encoding involves sequences of varying lengths.
- *N*, if the encoding involves only sequences of length *N*

Example

See the example for encoding, which calls **do_encoding**.

Requirements

Header: <locale>

See Also

Reference

codecvt Class

Other Resources

codecvt Members

codecvt::do_in

A virtual function called to convert a sequence of internal **Byte**s to a sequence of external **CharType**s.

```
virtual result do_in(
   StateType& _State,
   const Byte* _First1,
   const Byte* _Last1,
   const Byte*& _Next1,
   CharType* _First2,
   CharType* _Last2,
   CharType*& _Next2,
);
```

Parameters

_State

The conversion state that is maintained between calls to the member function.

_First1

Pointer to the beginning of the sequence to be converted.

_Last1

Pointer to the end of the sequence to be converted.

_Next1

Pointer beyond the end of the converted sequence, to the first unconverted character.

_First2

Pointer to the beginning of the converted sequence.

_Last2

Pointer to the end of the converted sequence.

_Next2

Pointer to the **CharType** that comes after the last converted **CharType**, to the first unaltered character in the destination sequence.

Return Value

A return that indicates the success, partial success, or failure of the operation. The function returns:

- **codecvt_base::error** if the source sequence is ill formed.
- **codecvt_base::noconv** if the function performs no conversion.
- **codecvt_base::ok** if the conversion succeeds.
- **codecvt_base::partial** if the source is insufficient or if the destination is not large enough, for the conversion to succeed.

Remarks

_State must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value as needed to reflect the current state of a successful conversion. Its stored value is otherwise unspecified.

Example

See the example for in, which calls **do_in**.

Requirements

Header: <locale>

See Also

Reference

codecvt Class

Other Resources

codecvt Members

codecvt::do_length

A virtual function that determines how many **Byte**s from a given sequence of external **Byte**s produce not more than a given number of internal **CharType**s and returns that number of **Byte**s.

```
virtual int do_length(
   const StateType& _State,
   const Byte* _First1,
   const Byte* _Last1,
   size_t _Len2
) const throw( );
```

Parameters

_State

The conversion state that is maintained between calls to the member function.

_First1

Pointer to the beginning of the external sequence.

_Last1

Pointer to the end of the external sequence.

_Len2

The maximum number of **Byte**s that can be returned by the member function.

Return Value

An integer that represents a count of the maximum number of conversions, not greater than *_Len2*, defined by the external source sequence at [*_First1*, *_Last1*).

Remarks

The protected virtual member function effectively calls **do_in**(*_State*, *_First1*, *_Last1*, *_Next1*, *_Buf*, *_Buf* + *_Len2*, *_Next2*) for *_State* (a copy of state), some buffer *_Buf*, and pointers *_Next1*and *_Next2*.

It then returns *_Next2* – **buf**. Thus, it counts the maximum number of conversions, not greater than *_Len2*, defined by the source sequence at [*_First1*, *_Last1*).

The template version always returns the lesser of *_Last1* – *_First1* and *_Len2*.

Example

See the example for length, which calls **do_length**.

Requirements

Header: <locale>

See Also

Reference

codecvt Class

Other Resources

codecvt Members

codecvt::do_max_length

A virtual function that returns the maximum number of external **Byte**s necessary to produce one internal **CharType**.

```
virtual int do_max_length( ) const throw( );
```

Return Value

The maximum number of **Byte**s necessary to produce one **CharType**.

Remarks

The protected virtual member function returns the largest permissible value that can be returned by do_length(*_First1*, *_Last1*, 1) for arbitrary valid values of *_First1* and *_Last1*.

Example

See the example for max_length, which calls **do_max_length**.

Requirements

Header: <locale>

See Also
Reference
codecvt Class
Other Resources
codecvt Members

codecvt::do_out

A virtual function called to convert a sequence of internal **CharType**s to a sequence of external **Byte**s.

```
virtual result do_out(
   StateType& _State,
   const CharType* _First1,
   const CharType* _Last1,
   const CharType*& _Next1,
   Byte* _First2,
   Byte* _Last2,
   Byte*& _Next2
) const;
```

Parameters

_State

The conversion state that is maintained between calls to the member function.

_First1

Pointer to the beginning of the sequence to be converted.

_Last1

Pointer to the end of the sequence to be converted.

_Next1

Reference to a pointer to the first unconverted **CharType**, after the last **CharType** converted.

_First2

Pointer to the beginning of the converted sequence.

_Last2

Pointer to the end of the converted sequence.

_Next2

Reference to a pointer to the first unconverted **Byte**, after the last **Byte** converted.

Return Value

The function returns:

- **codecvt_base::error** if the source sequence is ill formed.
- **codecvt_base::noconv** if the function performs no conversion.
- **codecvt_base::ok** if the conversion succeeds.
- **codecvt_base::partial** if the source is insufficient or if the destination is not large enough for the conversion to succeed.

Remarks

_State must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value as needed to reflect the current state of a successful conversion. Its stored value is otherwise unspecified.

Example

See the example for out, which calls **do_out**.

Requirements

Header: <locale>

See Also

Reference

codecvt Class

Other Resources

codecvt Members

codecvt::do_unshift

A virtual function called to provide the **Byte**s needed in a state-dependent conversion to complete the last character in a sequence of **Byte**s.

```
virtual result do_unshift(
   StateType& _State,
   Byte* _First2,
   Byte* _Last2,
   Byte* _Next2
);
```

Parameters

_State

The conversion state that is maintained between calls to the member function.

_First2

Pointer to the first position in the destination range.

_Last2

Pointer to the last position in the destination range.

_Next2

Pointer to the first unaltered element in the destination sequence.

Return Value

The function returns:

- **codecvt_base::error** if *_State* represents an invalid state
- **codecvt_base::noconv** if the function performs no conversion
- **codecvt_base::ok** if the conversion succeeds
- **codecvt_base::partial** if the destination is not large enough for the conversion to succeed

Remarks

The protected virtual member function tries to convert the source element **CharType**(0) to a destination sequence that it stores within [*_First2*, *_Last2*), except for the terminating element **Byte**(0). It always stores in *_Next2* a pointer to the first unaltered element in the destination sequence.

_State must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value as needed to reflect the current state of a successful conversion. Typically, converting the source element **CharType**(0) leaves the current state in the initial conversion state.

Example

See the example for unshift, which calls **do_unshift**.

Requirements

Header: <locale>

See Also

Reference

codecvt Class

Other Resources

codecvt Members

codecvt::encoding

Tests if the encoding of the **Byte** stream is state dependent, whether the ratio between the **Byte**s used and the **CharType**s produced is constant, and, if so, determines the value of that ratio.

```
int encoding( ) const throw( );
```

Return Value

If the return value is positive then that value is the constant number of **Byte** characters required to produce the **CharType** character.

The protected virtual member function returns:

- –1, if the encoding of sequences of type **extern_type** is state dependent.
- 0, if the encoding involves sequences of varying lengths.
- *N*, if the encoding involves only sequences of length *N*.

Remarks

The member function returns do_encoding.

Example

```
// codecvt_encoding.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   int result1 = use_facet<codecvt<char, char, mbstate_t> > ( loc ).encoding ( );
   cout << result1 << endl;
   result1 = use_facet<codecvt<wchar_t, char, mbstate_t> > ( loc ).encoding( );
   cout << result1 << endl;
   result1 = use_facet<codecvt<char, wchar_t, mbstate_t> > ( loc ).encoding( );
   cout << result1 << endl;
}
```

Output

```
1
0
1
```

Requirements

Header: <locale>

See Also

Reference

codecvt Class

Other Resources

codecvt Members

codecvt::in

Converts an external representation of a sequence of **Byte**s to an internal representation of a sequence of **CharType**s.

```
result in(
   StateType& _State,
   const Byte* _First1,
   const Byte* _Last1,
   const Byte*& _Next1,
   CharType* _First2,
   CharType* _Last2,
   CharType*& _Next2,
);
```

Parameters

_State

The conversion state that is maintained between calls to the member function.

_First1

Pointer to the beginning of the sequence to be converted.

_Last1

Pointer to the end of the sequence to be converted.

_Next1

Pointer beyond the end of the converted sequence to the first unconverted character.

_First2

Pointer to the beginning of the converted sequence.

_Last2

Pointer to the end of the converted sequence.

_Next2

Pointer to the **CharType** that comes after the last converted **Chartype** to the first unaltered character in the destination sequence.

Return Value

A return that indicates the success, partial success or failure of the operation. The function returns:

- **codecvt_base::error** if the source sequence is ill formed.
- **codecvt_base::noconv** if the function performs no conversion.
- **codecvt_base::ok** if the conversion succeeds.
- **codecvt_base::partial** if the source is insufficient or if the destination is not large enough for the conversion to succeed.

Remarks

_State must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value, as needed, to reflect the current state of a successful conversion. After a partial conversion, *_State* must be set so as to allow the conversion to resume when new characters arrive.

The member function returns do_in(*_State*, *_First1*, *_Last1*, *_Next1*, *First2*, *_Llast2*, *_Next2*).

Example

```
// codecvt_in.cpp
// compile with: /EHsc
#define _INTL
```

```
#include <locale>
#include <iostream>
using namespace std;
#define LEN 90
int main( )
{
   char* pszExt = "This is the string to be converted!";
   wchar_t pwszInt [LEN+1];
   memset(&pwszInt[0], 0, (sizeof(wchar_t))*(LEN+1));
   const char* pszNext;
   wchar_t* pwszNext;
   mbstate_t state = {0};
   locale loc("C");//English_Britain");//German_Germany
   int res = use_facet<codecvt<wchar_t, char, mbstate_t> >
     ( loc ).in( state,
          pszExt, &pszExt[strlen(pszExt)], pszNext,
          pwszInt, &pwszInt[strlen(pszExt)], pwszNext );
   pwszInt[strlen(pszExt)] = 0;
   wcout << ( (res!=codecvt_base::error) ? L"It worked! " : L"It didn't work! " )
   << L"The converted string is:\n ["
   << &pwszInt[0]
   << L"]" << endl;
   exit(-1);
}
```

Output

```
It worked! The converted string is:
 [This is the string to be converted!]
```

Requirements

Header: <locale>

See Also

Reference

codecvt Class

Other Resources

codecvt Members

codecvt::length

Determines how many **Byte**s from a given sequence of external **Byte**s produce not more than a given number of internal **CharType**s and returns that number of **Byte**s.

```
int length(
   const StateType& _State,
   const Byte* _First1,
   const Byte* _Last1,
   size_t _Len2
) const throw( );
```

Parameters

_State

The conversion state that is maintained between calls to the member function.

_First1

Pointer to the beginning of the external sequence.

_Last1

Pointer to the end of the external sequence.

_Len2

The maximum number of Bytes that can be returned by the member function.

Return Value

An integer that represents a count of the maximum number of conversions, not greater than *_Len2*, defined by the external source sequence at [*_First1*, *_Last1*).

Remarks

The member function returns do_length(*_State*, *_First1*, *_Last1*, *_Len2*).

Example

```
// codecvt_length.cpp
// compile with: /EHsc
#define _INTL
#include <locale>
#include <iostream>
using namespace std;
#define LEN 90
int main( )
{
   char* pszExt = "This is the string whose length is to be measured!";
   mbstate_t state = {0};
   locale loc("C");//English_Britain");//German_Germany
   int res = use_facet<codecvt<wchar_t, char, mbstate_t> >
     ( loc ).length( state,
          pszExt, &pszExt[strlen(pszExt)], LEN );
   cout << "The length of the string is: ";
   wcout << res;
   cout << "." << endl;
   exit(-1);
}
```

Output

```
The length of the string is: 50.
```

Requirements

Header: <locale>

See Also

Reference

codecvt Class

Other Resources

codecvt Members

codecvt::max_length

Returns the maximum number of external **Byte**s necessary to produce one internal **CharType**.

```
int max_length( ) const throw( );
```

Return Value

The maximum number of **Byte**s necessary to produce one **CharType**.

Remarks

The member function returns do_max_length.

Example

```
// codecvt_max_length.cpp
// compile with: /EHsc
#define _INTL
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc( "C");//English_Britain" );//German_Germany
   int res = use_facet<codecvt<char, char, mbstate_t> >
     ( loc ).max_length( );
   wcout << res << endl;
}
```

Output

```
1
```

Requirements

Header: <locale>

See Also

Reference

codecvt Class

Other Resources

codecvt Members

codecvt::out

Converts a sequence of internal **CharType**s to a sequence of external **Byte**s.

```
result out(
   StateType& _State,
   const CharType* _First1,
   const CharType* _Last1,
   const CharType*& _Next1,
   Byte* _First2,
   Byte* _Last2,
   Byte*& _Next2
);
```

Parameters

_State

The conversion state that is maintained between calls to the member function.

_First1

Pointer to the beginning of the sequence to be converted.

_Last1

Pointer to the end of the sequence to be converted.

_Next1

Reference to a pointer to the first unconverted **CharType** after the last **CharType** converted.

_First2

Pointer to the beginning of the converted sequence.

_Last2

Pointer to the end of the converted sequence.

_Next2

Reference to a pointer to the first unconverted **Byte** after the last converted **Byte**.

Return Value

The member function returns do_out(*_State*, *_First1*, *_Last1*, *_Next1*, *_First2*, *_Last2*, *_Next2*).

Remarks

For more information, see codecvt::do_out.

Example

```
// codecvt_out.cpp
// compile with: /EHsc
#define _INTL
#include <locale>
#include <iostream>
#include <wchar.h>
using namespace std;
#define LEN 90
int main( )
{
   char pszExt[LEN+1];
   wchar_t *pwszInt = L"This is the wchar_t string to be converted.";
   memset( &pszExt[0], 0, ( sizeof( char ) )*( LEN+1 ) );
   char* pszNext;
   const wchar_t* pwszNext;
   mbstate_t state;
```

```
    locale loc("C");//English_Britain");//German_Germany
    int res = use_facet<codecvt<wchar_t, char, mbstate_t> >
       ( loc ).out( state,
       pwszInt, &pwszInt[wcslen( pwszInt )], pwszNext ,
       pszExt, &pszExt[wcslen( pwszInt )], pszNext );
    pszExt[wcslen( pwszInt )] = 0;
    cout << ( ( res!=codecvt_base::error ) ? "It worked: " : "It didn't work: " )
    << "The converted string is:\n ["
    << &pszExt[0]
    << "]" << endl;
  }
```

Output

```
It worked: The converted string is:
 [This is the wchar_t string to be converted.]
```

Requirements

Header: <locale>

See Also

Reference

codecvt Class

Other Resources

codecvt Members

codecvt::unshift

Provides the **Byte**s needed in a state-dependent conversion to complete the last character in a sequence of **Byte**s.

```
result unshift(
   StateType& _State,
   Byte* _First2,
   Byte* _Last2,
   Byte* _Next2
);
```

Parameters

_State

The conversion state that is maintained between calls to the member function.

_First2

Pointer to the first position in the destination range.

_Last2

Pointer to the last position in the destination range.

_Next2

Pointer to the first unaltered element in the destination sequence.

Return Value

The function returns:

- **codecvt_base::error** if state represents an invalid state.
- **codecvt_base::noconv** if the function performs no conversion.
- **codecvt_base::ok** if the conversion succeeds.
- **codecvt_base::partial** if the destination is not large enough for the conversion to succeed.

Remarks

The protected virtual member function tries to convert the source element **CharType**(0) to a destination sequence that it stores within [*_First2*, *_Last2*), except for the terminating element **Byte**(0). It always stores in *_Next2* a pointer to the first unaltered element in the destination sequence.

_State must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value, as needed, to reflect the current state of a successful conversion. Typically, converting the source element **CharType**(0) leaves the current state in the initial conversion state.

The member function returns do_unshift(*_State*, *_First2*, *_Last2*, *_Next2*).

Requirements

Header: <locale>

See Also

Reference

codecvt Class

Other Resources

codecvt Members

codecvt_base Class

A base class for the codecvt class that is used to define an enumeration type referred to as **result**, used as the return type for the facet member functions to indicate the result of a conversion.

```
class codecvt_base {
public:
   enum result {ok, partial, error, noconv};
};
```

Remarks

The class describes an enumeration common to all specializations of template class codecvt. The enumeration result describes the possible return values from do_in or do_out:

- **ok** if the conversion between internal and external character encodings succeeds.
- **partial** if the destination is not large enough for the conversion to succeed.
- **error** if the source sequence is ill formed.
- **noconv** if the function performs no conversion.

Requirements

Header: <locale>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<locale> Members

codecvt_byname Class

A derived template class that describes an object that can serve as a collate facet of a given locale, enabling the retrieval of information specific to a cultural area concerning conversions.

```
template<Class CharType, class Byte, class StateType>
   class codecvt_byname: public codecvt< CharType, Byte, StateType> {
public:
   explicit codecvt_byname(
      const char* _Locname,
      size_t _Refs = 0
   );
protected:
   ~codecvt_byname( );
};
```

Remarks

Byname facets are automatically created when a named locale is constructed.

Its behavior is determined by the named locale *_Locname*. The constructor initializes its base object with codecvt<**CharType**, **Byte**, **StateType**>(*_Refs*).

Requirements

Header: <locale>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<locale> Members

collate Class

A template class that describes an object that can serve as a locale facet to control the ordering and grouping of characters within a string, comparisons between them and the hashing of strings.

For a list of all members of this type, see collate Members.

```
template <class CharType >
 class collate : public locale::facet
```

Parameters

CharType

The type used within a program to encode characters.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id.** In some languages, characters are grouped and treated as a single character, and in others, individual characters are treated as if they were two characters. The collating services provided by the collate class provide the way to sort these cases.

Requirements

Header: <locale>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

collate Members

<locale> Members

collate Members

The following tables list the members of the collate Class. For a list of members inherited by the **collate** class, see facet Class.

Reference

Constructors

collate	The constructor for objects of class **collate** that serves as a locale facet to handle string sorting conventions.

Typedefs

char_type	A type that describes a character of type **CharType**.
string_type	A type that describes a string of type **basic_string** containing characters of type **CharType**.

Member Functions

compare	Compares two character sequences according to their facet-specific rules for equality or inequality.
do_compare	A virtual function called to compare two character sequences according to their facet-specific rules for equality or inequality.
do_hash	A virtual function called to determine the hash value of sequences according to their facet-specific rules.
do_transform	A virtual function called to convert a character sequence from a locale to a string that may be used in lexicographical comparisons with other character sequences similarly converted from the same locale.
hash	Determines the hash value of sequence according to their facet-specific rules.
transform	Converts a character sequence from a locale to a string that may be used in lexicographical comparisons with other character sequences similarly converted from the same locale.

See Also

Reference

collate Class
facet Class
Thread Safety in the Standard C++ Library

collate Typedefs

For information about the typedefs in the collate class, see collate Members.

collate::char_type

A type that describes a character of type **CharType**.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

Requirements

Header: <locale>

See Also

Reference

collate Class

Other Resources

collate Members

collate::string_type

A type that describes a string of type **basic_string** containing characters of type **CharType**.

```
typedef basic_string<CharType> string_type;
```

Remarks

The type describes a specialization of template class basic_string whose objects can store copies of the source sequence.

Example

For an example of how to declare and use **string_type**, see transform.

Requirements

Header: <locale>

See Also

Reference

collate Class

Other Resources

collate Members

collate Member Functions

For information about the member functions in the collate class, see collate Members.

collate::collate

The constructor for objects of class collate that serves as a locale facet to handle string sorting conventions.

```
explicit collate(
   size_t _Refs = 0
);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 0: These values are not defined.

The constructor initializes its base object with **locale::**facet(*_Refs*).

Requirements

Header: <locale>

See Also

Reference

collate Class

Other Resources

collate Members

collate::compare

Compares two character sequences according to their facet-specific rules for equality or inequality.

```
int compare(
   const CharType* _First1,
   const CharType* _Last1,
   const CharType* _First2,
   const CharType* _Last2
) const;
```

Parameters

_First1

Pointer to the first element in the first sequence to be compared.

_Last1

Pointer to the last element in the first sequence to be compared.

_First2

Pointer to the first element in the second sequence to be compared.

_Last2

Pointer to the last element in the second sequence to be compared.

Return Value

The member function returns:

- –1 if the first sequence compares less than the second sequence.
- +1 if the second sequence compares less than the first sequence.
- 0 if the sequences are equivalent.

Remarks

The first sequence compares less if it has the smaller element in the earliest unequal pair in the sequences, or, if no unequal pairs exist, but the first sequence is shorter.

The member function returns do_compare(*_First1*, *_Last1*, *_First2*, *_Last2*).

Example

```
// collate_compare.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;

int main() {
   locale loc ( "German_germany" );
   _TCHAR * s1 = _T("Das ist wei\x00dfzz."); // \x00df is the German sharp-s, it comes befo
re z in the German alphabet
   _TCHAR * s2 = _T("Das ist weizzz.");
   int result1 = use_facet<collate<_TCHAR> > ( loc ).
      compare ( s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
   cout << result1 << endl;

   locale loc2 ( "C" );
   int result2 = use_facet<collate<_TCHAR> > ( loc2 ).
      compare (s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
   cout << result2 << endl;
```

```
}
```

Sample Output

```
-1
1
```

Requirements

Header: <locale>

See Also

Reference

collate Class

Other Resources

collate Members

collate::do_compare

A virtual function called to compare two character sequences according to their facet-specific rules for equality or inequality.

```
virtual int do_compare(
   const CharType* _First1,
   const CharType* _Last1,
   const CharType* _First2,
   const CharType* _Last2
) const;
```

Parameters

_First1

Pointer to the first element in the first sequence to be compared.

_Last1

Pointer to the last element in the first sequence to be compared.

_First2

Pointer to the first element in the second sequence to be compared.

_Last2

Pointer to the last element in the second sequence to be compared.

Return Value

The member function returns:

- -1 if the first sequence compares less than the second sequence.
- +1 if the second sequence compares less than the first sequence.
- 0 if the sequences are equivalent.

Remarks

The protected virtual member function compares the sequence at [*_First1*, *Last1)* with the sequence at *[_First2*, *_Last2*). It compares values by applying **operator<** between pairs of corresponding elements of type **CharType**. The first sequence compares less if it has the smaller element in the earliest unequal pair in the sequences or if no unequal pairs exist but the first sequence is shorter.

Example

See the example for collate::compare, which calls **do_compare**.

Requirements

Header: <locale>

See Also

Reference

collate Class

Other Resources

collate Members

collate::do_hash

A virtual function called to determine the hash value of sequences according to their facet-specific rules.

```
virtual long do_hash(
   const CharType* _First,
   const CharType* _Last
) const;
```

Parameters

_First

A pointer to the first character in the sequence whose has value is to be determined.

_Last

A pointer to the last character in the sequence whose has value is to be determined.

Return Value

A hash value of type **long** for the sequence.

Remarks

A hash value can be useful, for example, in distributing sequences pseudo-randomly across an array of lists.

Example

See the example for hash, which calls **do_hash**.

Requirements

Header: <locale>

See Also

Reference

collate Class

Other Resources

collate Members

collate::do_transform

A virtual function called to convert a character sequence from a locale to a string that may be used in lexicographical comparisons with other character sequences similarly converted from the same locale.

```
virtual string_type do_transform(
   const CharType* _First,
   const CharType* _Last
) const;
```

Parameters

_First

A pointer to the first character in the sequence to be converted.

_Last

A pointer to the last character in the sequence to be converted.

Return Value

A string that is the transformed character sequence.

Remarks

The protected virtual member function returns an object of class string_type whose controlled sequence is a copy of the sequence [*_First*, *_Last*). If a class derived from collate<**CharType**> overrides do_compare, it should also override **do_transform** to match. When passed to **collate::compare**, two transformed strings should yield the same result that you would get from passing the untransformed strings to compare in the derived class.

Example

See the example for transform, which calls **do_transform**.

Requirements

Header: <locale>

See Also

Reference

collate Class

Other Resources

collate Members

collate::hash

Determines the hash value of sequence according to their facet-specific rules.

```
long hash(
   const CharType* _First,
   const CharType* _Last
) const;
```

Parameters

_First

A pointer to the first character in the sequence whose has value is to be determined.

_Last

A pointer to the last character in the sequence whose has value is to be determined.

Return Value

A hash value of type **long** for the sequence.

Remarks

The member function returns do_hash(*_First*, *_Last*).

A hash value can be useful, for example, in distributing sequences pseudo-randomly across an array of lists.

Example

```
// collate_hash.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;

int main( )
{
   locale loc ( "German_germany" );
   _TCHAR * s1 = _T("\x00dfzz abc."); // \x00df is the German sharp-s (looks like beta), it
comes before z in the alphabet
   _TCHAR * s2 = _T("zzz abc."); // \x00df is the German sharp-s (looks like beta), it come
s before z in the alphabet

   long r1 = use_facet< collate<_TCHAR> > ( loc ).
      hash (s1, &s1[_tcslen( s1 )-1 ]);
   long r2 =  use_facet< collate<_TCHAR> > ( loc ).
      hash (s2, &s2[_tcslen( s2 )-1 ] );
   cout << r1 << " " << r2 << endl;
}
```

Output

```
541187293 551279837
```

Requirements

Header: <locale>

See Also

Reference

collate Class

Other Resources

collate Members

collate::transform

Converts a character sequence from a locale to a string that may be used in lexicographical comparisons with other character sequences similarly converted from the same locale.

```
string_type transform(
   const CharType* _First,
   const CharType* _Last
) const;
```

Parameters

_First

A pointer to the first character in the sequence to be converted.

_Last

A pointer to the last character in the sequence to be converted.

Return Value

A string that contains the transformed character sequence.

Remarks

The member function returns do_transform(*_First*, *_Last*).

Example

```
// collate_transform.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   _TCHAR* s1 = _T("\x00dfzz abc.");
   // \x00df is the German sharp-s (looks like beta),
   // it comes before z in the alphabet
   _TCHAR* s2 = _T("zzz abc.");

   collate<_TCHAR>::string_type r1;   // OK for typedef?
   r1 = use_facet< collate<_TCHAR> > ( loc ).
      transform (s1, &s1[_tcslen( s1 )-1 ]);

   cout << r1 << endl;

   basic_string<_TCHAR> r2 = use_facet< collate<_TCHAR> > ( loc ).
      transform (s2, &s2[_tcslen( s2 )-1 ]);

   cout << r2 << endl;

   int result1 = use_facet<collate<_TCHAR> > ( loc ).compare
      (s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );

   cout << _tcscmp(r1.c_str( ),r2.c_str( )) << result1
      << _tcscmp(s1,s2) <<endl;
}
```

Sample Output

```
????????     ?
????
```

```
???????       ?
????
-1-11
```

Requirements

Header: <locale>

See Also

Reference

collate Class

Other Resources

collate Members

collate_byname Class

A derived template class that describes an object that can serve as a collate facet of a given locale, enabling the retrieval of information specific to a cultural area concerning string sorting conventions.

```
template<Class CharType>
   class collate_byname : public collate<Elem> {
public:
   explicit collate_byname(
      const char* _Locname,
      size_t _Refs = 0
   );
protected:
   ~collate_byname( );
};
```

Remarks

The template class describes an object that can serve as a locale facet of type collate<**CharType**>. Its behavior is determined by the named locale *_Locname*. The constructor initializes its base object with collate<**CharType**>(*_Refs*).

Requirements

Header: <locale>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<locale> Members

ctype Class

A class that provides a facet that is used to classify characters, convert from upper and lower cases, and convert between the native character set and that set used by the locale.

For a list of all members of this type, see ctype Members.

```
template <class CharType>
class ctype : public ctype_base
```

Parameters

CharType

The type used within a program to encode characters.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id.** Classification criteria are provided a nested bitmask type in the base class ctype_base.

The Standard C++ Library defines two explicit specializations of this template class:

- ctype<**char**>, an explicit specialization whose differences are described separately.
- **ctype**<**wchar_t**>, which treats elements as wide characters.

Other specializations of template class **ctype**<**CharType**>:

- Convert a value ***ch*** of type **CharType** to a value of type **char** with the expression (**char**)**ch**.
- Convert a value ***byte*** of type **char** to a value of type **CharType** with the expression **CharType** (**byte**).

All other operations are performed on **char** values in the same way as for the explicit specialization **ctype**<**char**>.

Requirements

Header: <locale>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

ctype Members

<locale> Members

ctype Members

The following tables list the members of the ctype Class. For a list of members inherited by the **ctype** class, see ctype_base Class.

Reference

Constructors

ctype	Constructor for objects of class **ctype** that serve as locale facets for characters.

Typedefs

char_type	A type that describes a character used by a locale.

Member Functions

do_is	A virtual function called to test whether a single character has a particular attribute, or classify the attributes of each character in a range and stores them in an array.
do_narrow	A virtual function called to convert a character of type **CharType** used by a locale to the corresponding character of type **char** in the native character set. Deprecated. Use ctype::_Do_narrow_s instead.
_Do_narrow_s	A virtual function called to convert a character of type **CharType** used by a locale to the corresponding character of type **char** in the native character set.
do_scan_is	A virtual function called to locate the first character in a range that matches a specified mask.
do_scan_not	A virtual function called to locate the first character in a range that does not match a specified mask.
do_tolower	A virtual function called to convert a character or a range of characters to their lower case.
do_toupper	A virtual function called to convert a character or a range of characters to upper case.
do_widen	A virtual function called to converts a character of type **char** in the native character set to the corresponding character of type **CharType** used by a locale. Deprecated. Use ctype::_Do_widen_s instead.
_Do_widen_s	A virtual function called to converts a character of type **char** in the native character set to the corresponding character of type **CharType** used by a locale.
is	Tests whether a single character has a particular attribute, or classifies the attributes of each character in a range and stores them in an array.
narrow	Converts a character of type **CharType** used by a locale to the corresponding character of type char in the native character set. Deprecated. Use ctype::_Narrow_s instead.
_Narrow_s	Converts a character of type **CharType** used by a locale to the corresponding character of type char in the native character set.
scan_is	Locates the first character in a range that matches a specified mask.
scan_not	Locates the first character in a range that does not match a specified mask.
tolower	Converts a character or a range of characters to lower case.
toupper	Converts a character or a range of characters to upper case.

widen	Converts a character of type **char** in the native character set to the corresponding character of type **CharType** used by a locale. Deprecated. Use ctype::_Widen_s instead.
_Widen_s	Converts a character of type **char** in the native character set to the corresponding character of type **CharType** used by a locale.

See Also

Reference

ctype Class
ctype_base Class
Thread Safety in the Standard C++ Library

ctype Typedefs

For information about the typedefs in the ctype class, see ctype Members.

ctype::char_type

A type that describes a character used by a locale.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

Example

See the member function widen for an example that uses **char_type** as a return value.

Requirements

Header: <locale>

See Also

Reference

ctype Class

Other Resources

ctype Members

ctype Member Functions

For information about the member functions in the ctype class, see ctype Members.

ctype::ctype

Constructor for objects of class ctype that serve as locale facets for characters.

```
explicit ctype(
   size_t _Refs = 0
);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 0: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its **locale::facet** base object with **locale::**facet(*_Refs*).

Requirements

Header: <locale>

See Also

Reference

ctype Class

Other Resources

ctype Members

ctype::do_is

A virtual function called to test whether a single character has a particular attribute, or classify the attributes of each character in a range and stores them in an array.

```
virtual bool do_is(
   Mask _Maskval,
   CharType _Ch
) const;
virtual const CharType *do_is(
   const CharType* _First,
   const CharType* _Last,
   mask* _Dest
) const;
```

Parameters

_MaskVal

The mask value for which the character is to be tested.

_Ch

The character whose attributes are to be tested.

_First

A pointer to the first character in the range whose attributes are to be classified.

_Last

A pointer to the last character in the range whose attributes are to be classified.

_Dest

A pointer to the beginning of the array where the mask values characterizing the attributes of each of the characters are to be stored.

Return Value

The first member function returns a Boolean value that is **true** if the character tested has the attribute described by the mask value; **false** if it fails to have the attribute.

The second member function returns an array containing the mask values characterizing the attributes of each of the characters in the range.

Remarks

The mask values classifying the attributes of the characters are provided by the class ctype_base, from which ctype derives. The first member function can accept expressions for its first parameter referred to as bitmasks and formed from the combination of mask values by the logical bitwise operators (| , & , ^ , ~).

Example

See the example for is, which calls **do_is**.

Requirements

Header: <locale>

See Also

Reference

ctype Class

Other Resources

ctype Members

ctype::do_narrow

A virtual function called to convert a character of type **CharType** used by a locale to the corresponding character of type **char** in the native character set.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct. Consider using ctype::_Do_narrow_s instead.

```
virtual char do_narrow(
   CharType _Ch,
   char _Dfault
) const;
virtual const CharType *do_narrow(
   const CharType* _First,
   const CharType* _Last,
   char dflt, char* _Dest
) const;
```

Parameters

_Ch

The character of type **Chartype** used by the locale to be converted.

_Dfault

The default value to be assigned by the member function to characters of type **CharType** that do not have counterpart characters of type **char**.

_First

A pointer to the first character in the range of characters to be converted.

_Last

A pointer to the last character in the range of characters to be converted.

_Dest

A const pointer to the first character of type **char** in the destination range that stores the converted range of characters.

Return Value

The first protected member function returns the native character of type char that corresponds to the parameter character of type **CharType** or *_Dfault* if not counterpart is defined.

The second protected member function returns a pointer to the destination range of native characters converted from characters of type **CharType**.

Remarks

The second protected member template function stores in *_Dest* [*I*] the value **do_narrow**(*_First* [*I*], *_Dfault*), for *I* in the interval [0, *_Last* – *_First*).

Example

See the example for narrow, which calls **do_narrow**.

Requirements

Header: <locale>

See Also

Reference

ctype Class

Other Resources

ctype Members

ctype::_Do_narrow_s

A virtual function called to convert a character of type **CharType** used by a locale to the corresponding character of type **char** in the native character set.

```
virtual const CharType *_Do_narrow_s(
    const CharType *_First,
    const CharType *_Last,
    char _Dfault,
    char *_Dest,
    size_t _Dest_size
) const;
```

Parameters

_First

A pointer to the first character in the range of characters to be converted.

_Last

A pointer to the last character in the range of characters to be converted.

_Dfault

The default value to be assigned by the member function to characters of type **CharType** that do not have counterpart characters of type **char**.

_Dest

A const pointer to the first character of type **char** in the destination range that stores the converted range of characters.

_Dest_size

The size of *_Dest* in bytes.

Return Value

This protected member function returns a pointer to the destination range of native characters converted from characters of type **CharType**.

Remarks

This protected member template function stores in *_Dest* [*I*] the value **_Do_narrow_s**(*_First* [*I*], *_Dfault*), for *I* in the interval [0, *_Last* – *_First*).

Example

See the example for ctype::_Narrow_s, which calls **_Do_narrow_s**.

Requirements

Header: <locale>

See Also

Reference

ctype Class

Safe Libraries: Standard C++ Library

Other Resources

ctype Members

ctype::do_scan_is

A virtual function called to locate the first character in a range that matches a specified mask.

```
virtual const CharType *do_scan_is(
   mask _MaskVal,
   const CharType* _First,
   const CharType* _Last,
) const;
```

Parameters

_MaskVal

The mask value to be matched by a character.

_First

A pointer to the first character in the range to be scanned.

_Last

A pointer to the last character in the range to be scanned.

Return Value

A pointer to the first character in a range that does match a specified mask. If no such value exists, the function returns *_Last*.

Remarks

The protected member function returns the smallest pointer *_Ptr* in the range [*_First*, *_Last*) for which do_is(*_MaskVal*, **_Ptr*) is true.

Example

See the example for scan_is, which calls **do_scan_is**.

Requirements

Header: <locale>

See Also

Reference

ctype Class

Other Resources

ctype Members

ctype::do_scan_not

A virtual function called to locate the first character in a range that does not match a specified mask.

```
virtual const CharType *do_scan_not(
   mask _MaskVal,
   const CharType* _First,
   const CharType* _Last
) const;
```

Parameters

_MaskVal

The mask value not to be matched by a character.

_First

A pointer to the first character in the range to be scanned.

_Last

A pointer to the last character in the range to be scanned.

Return Value

A pointer to the first character in a range that doesn't match a specified mask. If no such value exists, the function returns *_Last*.

Remarks

The protected member function returns the smallest pointer *_Ptr* in the range [*_First*, *_Last*) for which do_is(*_MaskVal*, **_Ptr*) is false.

Example

See the example for scan_not, which calls **do_scan_not**.

Requirements

Header: <locale>

See Also

Reference

ctype Class

Other Resources

ctype Members

ctype::do_tolower

A virtual function called to convert a character or a range of characters to lower case.

```
virtual CharType do_tolower(
   CharType _Ch
) const;
virtual const CharType *do_tolower(
   const CharType* _First,
   const CharType* _Last,
) const;
```

Parameters

_Ch

The character to be converted to lower case.

_First

A pointer to the first character in the range of characters whose cases are to be converted.

_Last

A pointer to the last character in the range of characters whose cases are to be converted.

Return Value

The first protected member function returns the lowercase form of the parameter character. If no lowercase form exists, it returns the parameter character. The second protected member function returns a constant pointer to the first character in the converted range of characters.

Remarks

The second protected member template function replaces each element *_First* [*I*], for *I* in the interval [0, *_Last* – *_First*), with **do_tolower**(*_First* [*I*]).

Example

See the example for tolower, which calls **do_tolower**.

Requirements

Header: <locale>

See Also

Reference

ctype Class

Other Resources

ctype Members

ctype::do_toupper

A virtual function called to convert a character or a range of characters to upper case.

```
virtual CharType do_toupper(
   CharType _Ch
) const;
virtual const CharType *do_toupper(
   CharType* _First,
   CharType* _Last
) const;
```

Parameters

_Ch

The character to be converted to upper case.

_First

A pointer to the first character in the range of characters whose cases are to be converted.

_Last

A pointer to the last character in the range of characters whose cases are to be converted.

Return Value

The first protected member function returns the uppercase form of the parameter character. If no uppercase form exists, it returns the parameter character. The second protected member function returns a constant pointer to the first character in the converted range of characters.

Remarks

The second protected member template function replaces each element *_First* [*I*], for *I* in the interval [0, *_Last* – *_First*), with **do_toupper**(*_First* [*I*]).

Example

See the example for toupper, which calls **do_toupper**.

Requirements

Header: <locale>

See Also

Reference

ctype Class

Other Resources

ctype Members

ctype::do_widen

A virtual function called to converts a character of type **char** in the native character set to the corresponding character of type **CharType** used by a locale.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct. Consider using ctype::_Do_widen_s instead.

```
virtual CharType do_widen(
   char _Byte
) const;
virtual const char *do_widen(
   const CharType* _First,
   const CharType* _Last,
   CharType* _Dest
) const;
```

Parameters

_Byte

The character of type **char** in the native character set to be converted.

_First

A pointer to the first character in the range of characters to be converted.

_Last

A pointer to the last character in the range of characters to be converted.

_Dest

A pointer to the first character of type **CharType** in the destination range that stores the converted range of characters.

Return Value

The first protected member function returns the character of type **CharType** that corresponds to the parameter character of native type **char**.

The second protected member function returns a pointer to the destination range of characters of type **CharType** used by a locale converted from native characters of type **char**.

Remarks

The second protected member template function stores in *_Dest* [*I*] the value **do_widen**(*_First* [*I*]), for *I* in the interval [0, *_Last* - *_First*).

Example

See the example for widen, which calls **do_widen**.

Requirements

Header: <locale>

See Also

Reference

ctype Class

Other Resources

ctype Members

ctype::_Do_widen_s

A virtual function called to converts a character of type **char** in the native character set to the corresponding character of type **CharType** used by a locale.

```
virtual const char *_Do_widen_s(
    const char *_First,
    const char *_Last,
    CharType *_Dest,
    size_t _Dest_size
) const;
```

Parameters

_First

A pointer to the first character in the range of characters to be converted.

_Last

A pointer to the last character in the range of characters to be converted.

_Dest

A pointer to the first character of type **CharType** in the destination range that stores the converted range of characters.

_Dest_size

The size of *_Dest*. If **CharType** is **char**, then this is in bytes. If **CharType** is **wchar_t**, then this is in words.

Return Value

This protected member function returns a pointer to the destination range of characters of type **CharType** used by a locale converted from native characters of type **char**.

Remarks

This protected member template function stores in *_Dest* [*I*] the value **_Do_widen_s**(*_First* [*I*]), for *I* in the interval [0, *_Last* - *_First*).

Example

See the example for ctype::_Widen_s, which calls **_Do_widen_s**.

Requirements

Header: <locale>

See Also

Reference

ctype Class

Safe Libraries: Standard C++ Library

Other Resources

ctype Members

ctype::is

Tests whether a single character has a particular attribute or classifies the attributes of each character in a range and stores them in an array.

```
bool is(
   Mask _MaskVal,
   CharType _Ch
) const;
const CharType *is(
   const CharType* _First,
   const CharType* _Last,
   mask* _Dest
) const;
```

Parameters

_MaskVal

The mask value for which the character is to be tested.

_Ch

The character whose attributes are to be tested.

_First

A pointer to the first character in the range whose attributes are to be classified.

_Last

A pointer to the last character in the range whose attributes are to be classified.

_Dest

A pointer to the beginning of the array where the mask values characterizing the attributes of each of the characters are to be stored.

Return Value

The first member function returns a Boolean value that is **true** if the character tested has the attribute described by the mask value; **false** if it fails to have the attribute.

The second member function returns an array containing the mask values characterizing the attributes of each of the characters in the range.

Remarks

The mask values classifying the attributes of the characters are provided by the class ctype_base Class, from which ctype derives. The first member function can accept expressions for its first parameter referred to as bitmasks and formed from the combination of mask values by the logical bitwise operators (| , & , ^ , ~).

Example

```
// ctype_is.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main() {
   locale loc1 ( "German_Germany" ), loc2 ( "English_Australia" );

   if (use_facet<ctype<char> > ( loc1 ).is( ctype_base::alpha, 'a' ))
      cout << "The character 'a' in locale loc1 is alphabetic."
           << endl;
   else
      cout << "The character 'a' in locale loc1 is not alphabetic."
```

```
            << endl;

    if (use_facet<ctype<char> > ( loc2 ).is( ctype_base::alpha, '!' ))
       cout << "The character '!' in locale loc2 is alphabetic."
            << endl;
    else
       cout << "The character '!' in locale loc2 is not alphabetic."
            << endl;

    char *string = "Hello, my name is John!";
    ctype<char>::mask maskarray[30];
    use_facet<ctype<char> > ( loc2 ).is(
       string, string + strlen(string), maskarray );
    for (unsigned int i = 0; i < strlen(string); i++) {
       cout << string[i] << ": "
            << (maskarray[i] & ctype_base::alpha ? "alpha"
                                                 : "not alpha")
            << endl;;
    };
  }
```

Output

```
The character 'a' in locale loc1 is alphabetic.
The character '!' in locale loc2 is not alphabetic.
H: alpha
e: alpha
l: alpha
l: alpha
o: alpha
,: not alpha
 : not alpha
m: alpha
y: alpha
 : not alpha
n: alpha
a: alpha
m: alpha
e: alpha
 : not alpha
i: alpha
s: alpha
 : not alpha
J: alpha
o: alpha
h: alpha
n: alpha
!: not alpha
```

Requirements

Header: <locale>

See Also

Reference

ctype Class

Other Resources

ctype Members

ctype::narrow

Converts characters of type **CharType** used by a locale to the corresponding characters of type **char** in the native character set.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct. Consider using ctype::_Narrow_s instead.

```
char narrow(
   CharType _Ch,
   char _Dfault
) const;
const CharType* narrow(
   const CharType* _First,
   const CharType* _Last,
   char _Dfault,
   char* _Dest
) const;
```

Parameters

_Ch

The character of type **Chartype** used by the locale to be converted.

_Dfault

The default value to be assigned by the member function to characters of type **CharType** that do not have counterpart characters of type **char**.

_First

A pointer to the first character in the range of characters to be converted.

_Last

A pointer to the character immediately following the last character in the range of characters to be converted.

_Dest

A const pointer to the first character of type **char** in the destination range that stores the converted range of characters.

Return Value

The first member function returns the native character of type **char** that corresponds to the parameter character of type **CharType** *_Dfault* if not counterpart is defined.

The second member function returns a pointer to the destination range of native characters converted from characters of type **CharType**.

Remarks

The first member function returns do_narrow(*_Ch*, *_Dfault*). The second member function returns do_narrow (*_First*, *_Last*, *_Dfault*, *_Dest*). Only the basic source characters are guaranteed to have a unique inverse image **CharType** under **narrow**. For these basic source characters, the following invariant holds: **narrow** (widen (**c**), 0) == **c**.

Example

```
// ctype_narrow.cpp
// compile with: /EHsc /W3
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc1 ( "english" );
   wchar_t *str1 = L"\x0392fhello everyone";
   char str2 [16];
```

```
    // Note:: ctype::narrow is potentially unsafe, consider
    // using ctype::_Narrow_s instead.
    bool result1 = (use_facet<ctype<wchar_t> > ( loc1 ).narrow
       ( str1, str1 + wcslen(str1), 'X', &str2[0] ) != 0);  // C4996
    str2[wcslen(str1)] = '\0';
    wcout << str1 << endl;
    cout << &str2[0] << endl;
}
```

Output

```
Xhello everyone
```

Requirements

Header: <locale>

See Also

Reference

ctype Class

Other Resources

ctype Members

ctype::_Narrow_s

Converts characters of type **CharType** used by a locale to the corresponding characters of type **char** in the native character set.

```
const CharType *_Narrow_s(
    const CharType *_First,
    const CharType *_Last,
    char _Dfault,
    char *_Dest,
    size_t _Dest_size
) const;
```

Parameters

_First

A pointer to the first character in the range of characters to be converted.

_Last

A pointer to the character immediately following the last character in the range of characters to be converted.

_Dfault

The default value to be assigned by the member function to characters of type **CharType** that do not have counterpart characters of type **char**.

_Dest

A const pointer to the first character of type **char** in the destination range that stores the converted range of characters.

_Dest_size

The size of *_Dest* in bytes.

Return Value

This member function returns a pointer to the destination range of native characters converted from characters of type **CharType**.

Remarks

This member function returns _Do_narrow_s(*_First*, *_Last*, *_Dfault*, *_Dest*, *_Dest_size*).

Example

```
// ctype__Narrow_s.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
    locale loc1("english");
    wchar_t *str1 = L"\x0392fhello everyone";
    const int str2_size = 16;
    char str2[str2_size];
    bool result1 = (use_facet<ctype<wchar_t> >(loc1)._Narrow_s
        (str1, str1 + wcslen(str1), 'X', &str2[0], str2_size) != 0);
    str2[wcslen(str1)] = '\0';
    wcout << str1 << endl;
    cout << &str2[0] << endl;
}
```

Output

```
Xhello everyone
```

Requirements

Header: <locale>

See Also

Reference

ctype Class

Safe Libraries: Standard C++ Library

Other Resources

ctype Members

ctype::scan_is

Locates the first character in a range that matches a specified mask.

```
const CharType *scan_is(
   mask _MaskVal,
   const CharType* _First,
   const CharType* _Last,
) const;
```

Parameters

_MaskVal

The mask value to be matched by a character.

_First

A pointer to the first character in the range to be scanned.

_Last

A pointer to the last character in the range to be scanned.

Return Value

A pointer to the first character in a range that does match a specified mask. If no such value exists, the function returns *_Last*.

Remarks

The member function returns do_scan_is(*_MaskVal*, *_First*, *_Last*).

Example

```
// ctype_scan_is.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc1 ( "German_Germany" );

   char *string = "Hello, my name is John!";

   const char* i = use_facet<ctype<char> > ( loc1 ).scan_is
      ( ctype_base::punct, &string[0], &string[strlen(&string[0])-1] );
   cout << "The first punctuation is \"" << *i << "\" at position: "
      << i - string << endl;
}
```

Output

```
The first punctuation is "," at position: 5
```

Requirements

Header: <locale>

See Also

Reference

ctype Class

Other Resources

ctype Members

ctype::scan_not

Locates the first character in a range that does not match a specified mask.

```
const CharType *scan_not(
   mask _MaskVal,
   const CharType* _First,
   const CharType* _Last,
) const;
```

Parameters

_MaskVal

The mask value not to be matched by a character.

_First

A pointer to the first character in the range to be scanned.

_Last

A pointer to the last character in the range to be scanned.

Return Value

A pointer to the first character in a range that does not match a specified mask. If no such value exists, the function returns *_Last*.

Remarks

The member function returns do_scan_not(*_MaskVal*, *_First*, *_Last*).

Example

```
// ctype_scan_not.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc1 ( "German_Germany" );

   char *string = "Hello, my name is John!";

   const char* i = use_facet<ctype<char> > ( loc1 ).scan_not
      ( ctype_base::alpha, &string[0], &string[strlen(&string[0])-1] );
   cout << "First nonalpha character is \"" << *i << "\" at position: "
      << i - string << endl;
}
```

Output

```
First nonalpha character is "," at position: 5
```

Requirements

Header: <locale>

See Also

Reference

ctype Class

Other Resources

ctype Members

ctype::tolower

Converts a character or a range of characters to lower case.

```
CharType tolower(
   CharType _Ch
) const;
const CharType *tolower(
   CharType* _First,
   const CharType* _Last,
) const;
```

Parameters

_Ch

The character to be converted to lower case.

_First

A pointer to the first character in the range of characters whose cases are to be converted.

_Last

A pointer to the last character in the range of characters whose cases are to be converted.

Return Value

The first member function returns the lowercase form of the parameter character. If no lowercase form exists, it returns the parameter character.

The second member function returns a constant pointer to the last character in the converted range of characters.

Remarks

The first member function returns do_tolower(*_Ch*). The second member function returns do_tolower(*_First*, *_Last*).

Example

```
// ctype_tolower.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc1 ( "German_Germany" );

   char string[] = "HELLO, MY NAME IS John!";

   use_facet<ctype<char> > ( loc1 ).tolower
      ( &string[0], &string[strlen(&string[0])-1] );
   cout << "The lowercase string is: " << string << endl;
}
```

Output

```
The lowercase string is: hello, my name is john!
```

Requirements

Header: <locale>

See Also

Reference

ctype Class

Other Resources

ctype::toupper

Converts a character or a range of characters to upper case.

```
CharType toupper(
   CharType _Ch
) const;
const CharType *toupper(
   CharType* _First,
   const CharType* _Last
) const;
```

Parameters

_Ch

The character to be converted to uppercase.

_First

A pointer to the first character in the range of characters whose cases are to be converted.

_Last

A pointer to the last character in the range of characters whose cases are to be converted.

Return Value

The first member function returns the uppercase form of the parameter character. If no uppercase form exists, it returns the parameter character.

The second member function returns a constant pointer to the last character in the converted range of characters.

Remarks

The first member function returns do_toupper(*_Ch*). The second member function returns do_toupper(*_First*, *_Last*).

Example

```
// ctype_toupper.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc1 ( "German_Germany" );

   char string[] = "Hello, my name is John!";

   use_facet<ctype<char> > ( loc1 ).toupper
      ( &string[0], &string[strlen(&string[0])-1] );
   cout << "The uppercase string is: " << string << endl;
}
```

Output

```
The uppercase string is: HELLO, MY NAME IS JOHN!
```

Requirements

Header: <locale>

See Also

Reference

ctype Class

Other Resources

ctype::widen

Converts a character of type **char** in the native character set to the corresponding character of type **CharType** used by a locale.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct. Consider using ctype::_Widen_s instead.

```
CharType widen(
   char _Byte
) const;
const char *widen(
   char* _First,
   char* _Last,
   CharType* _Dest
) const;
```

Parameters

_Byte

The character of type char in the native character set to be converted.

_First

A pointer to the first character in the range of characters to be converted.

_Last

A pointer to the character immediately following the last character in the range of characters to be converted.

_Dest

A pointer to the first character of type **CharType** in the destination range that stores the converted range of characters.

Return Value

The first member function returns the character of type **CharType** that corresponds to the parameter character of native type **char**.

The second member function returns a pointer to the destination range of characters of type **CharType** used by a locale converted from native characters of type **char**.

Remarks

The first member function returns do_widen(*_Byte*). The second member function returns do_widen(*_First*, *_Last*, *_Dest*).

Example

```
// ctype_widen.cpp
// compile with: /EHsc /W3
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc1 ( "English" );
   char *str1 = "Hello everyone!";
   wchar_t str2 [16];
   // Note:: ctype::widen is potentially unsafe, consider
   // using ctype::_Widen_s instead.
   bool result1 = (use_facet<ctype<wchar_t> > ( loc1 ).widen
      ( str1, str1 + strlen(str1), &str2[0] ) != 0);  // C4996
   str2[strlen(str1)] = '\0';
   cout << str1 << endl;
   wcout << &str2[0] << endl;

   ctype<wchar_t>::char_type charT;
```

```
    charT = use_facet<ctype<char> > ( loc1 ).widen( 'a' );
}
```

Output

```
Hello everyone!
Hello everyone!
```

Requirements

Header: <locale>

See Also

Reference

ctype Class

Other Resources

ctype Members

ctype::_Widen_s

Converts a character of type **char** in the native character set to the corresponding character of type **CharType** used by a locale.

```
const char *_Widen_s(
    const char *_First,
    const char *_Last,
    CharType *_Dest,
    size_t _Dest_size
) const;
```

Parameters

_First

A pointer to the first character in the range of characters to be converted.

_Last

A pointer to the character immediately following the last character in the range of characters to be converted.

_Dest

A pointer to the first character of type **CharType** in the destination range that stores the converted range of characters.

_Dest_size

The size of *_Dest*. If **CharType** is **char**, then this is in bytes. If **CharType** is **wchar_t**, then this is in words.

Return Value

This member function returns a pointer to the destination range of characters of type **CharType** used by a locale converted from native characters of type **char**.

Remarks

This member function returns _Do_widen_s(*_First*, *_Last*, *_Dest*, *_DestSize*).

Example

```
// ctype__Widen_s.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
    locale loc1("English");
    char *str1 = "Hello everyone!";
    const int str2_size = 16;
    wchar_t str2[str2_size];
    bool result1 = (use_facet<ctype<wchar_t> >(loc1)._Widen_s
        (str1, str1 + strlen(str1), &str2[0], str2_size) != 0);
    str2[strlen(str1)] = '\0';
    cout << str1 << endl;
    wcout << &str2[0] << endl;

    ctype<wchar_t>::char_type charT;
    charT = use_facet<ctype<char> >(loc1).widen('a');
}
```

Output

```
Hello everyone!
Hello everyone!
```

Requirements

Header: <locale>

See Also

Reference

ctype Class

Safe Libraries: Standard C++ Library

Other Resources

ctype Members

ctype<char> Class

The class is an explicit specialization of template class **ctype<CharType>** to type **char**, describing an object that can serve as a locale facet to characterize various properties of a character of type **char**.

```
template<>
   class ctype<char>
   : public locale::facet, public ctype_base {
public:
   typedef char char_type;
   explicit ctype(
      const mask *tab = 0,
      bool del = false,
      size_t refs = 0
   );
   bool is(
      mask maskval,
      char ch
   ) const;
   const char *is(
      const char *first,
      const char *last,
      mask *dest
  ) const;
   const char *scan_is(
      mask maskval,
      const char *first,
      const char *last
   ) const;
   const char *scan_not(
      mask maskval,
      const char *first,
      const char *last
   ) const;
   char toupper(char ch) const;
   const char *toupper(char *first, char *last) const;
   char tolower(char ch) const;
   const char *tolower(char *first, char *last) const;
   char widen(char byte) const;
   const char *widen(char *first, char *last, char *dest) const;
   char narrow(char ch, char dflt) const;
   const char *narrow(const char *first,
        const char *last, char dflt, char *dest) const;
   static locale::id id;
protected:
   ~ctype( );
   //other protected members
    };
```

Remarks

The explicit specialization differs from the template class in several ways:

- An object of class ctype<**char**> stores a pointer to the first element of a ctype mask table, an array of UCHAR_MAX + 1 elements of type **ctype_base::mask**. It also stores a Boolean object that indicates whether the array should be deleted (using **operator delete[]**) when the ctype<**Elem**> object is destroyed.
- Its sole public constructor lets you specify **tab**, the ctype mask table, and **del**, the Boolean object that is true if the array should be deleted when the ctype<**char**> object is destroyed, as well as the reference-count parameter refs.
- The protected member function **table** returns the stored ctype mask table.
- The static member object **table_size** specifies the minimum number of elements in a ctype mask table.

- The protected static member function **classic_table**(returns the ctype mask table appropriate to the "C" locale.
- There are no protected virtual member functions do_is, do_scan_is, or do_scan_not. The corresponding public member functions perform the equivalent operations themselves.

The member functions do_narrow and do_widen copy elements unaltered.

Requirements

Header: <locale>

See Also

Reference

facet Class

ctype_base Class

Thread Safety in the Standard C++ Library

Other Resources

<locale> Members

ctype_base Class

The class serves as a base class for facets of template class ctype. A base class for the ctype class that is used to define enumeration types used to classify or test characters either individually or within entire ranges.

```
class ctype_base {
public:
    enum mask {
        space, print, cntrl, upper, lower, digit,
        punct, xdigit, alpha, alnum, graph
   };
```

Remarks

It defines an enumeration mask. Each enumeration constant characterizes a different way to classify characters, as defined by the functions with similar names declared in the header <ctype.h>. The constants are:

- **space** (function isspace)
- **print** (function isprint)
- **cntrl** (function iscntrl)
- **upper** (function isupper)
- **lower** (function islower)
- **digit** (function isdigit)
- **punct** (function ispunct)
- **xdigit** (function isxdigit)
- **alpha** (function isalpha)
- **alnum** (function isalnum)
- **graph** (function isgraph)

You can characterize a combination of classifications by ORing these constants. In particular, it is always true that **alnum** == (**alpha** | **digit**) and **graph** == (**alnum** | **punct**).

Requirements

Header: <locale>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<locale> Members

ctype_byname Class

The derived template class describes an object that can serve as a ctype facet of a given locale, enabling the classification of characters and conversion of characters between case and native and locale specified character sets.

```
template<class CharType>
    class ctype_byname : public ctype<CharType> {
public:
   explicit ctype_byname(
      const char* _Locname,
      size_t _Refs = 0
   );
protected:
   ~ctype_byname( );
   };
```

Remarks

Its behavior is determined by the named locale *_Locname*. The constructor initializes its base object with ctype<**CharType**> (*_Refs*) or the equivalent for base class **ctype**<**char**>.

Requirements

Header: <locale>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<locale> Members

locale Class

The class that describes a locale object that encapsulates culture-specific information as a set of facets that collectively define a specific localized environment.

For a list of all members of this type, see locale Members.

Remarks

A facet is a pointer to an object of a class derived from class facet that has a public object of the form:

```
static locale::id id;
```

You can define an open-ended set of these facets. You can also construct a locale object that designates an arbitrary number of facets.

Predefined groups of these facets represent the locale categories traditionally managed in the Standard C Library by the function **setlocale**.

Category **collate** (LC_COLLATE) includes the facets:

```
collate<char>
collate<wchar_t>
```

Category **ctype** (LC_CTYPE) includes the facets:

```
ctype<char>
ctype<wchar_t>
codecvt<char, char, mbstate_t>
codecvt<wchar_t, char, mbstate_t>
```

Category **monetary** (LC_MONETARY) includes the facets:

```
moneypunct<char, false>
moneypunct<wchar_t, false>
moneypunct<char, true>
moneypunct<wchar_t, true>
money_get<char, istreambuf_iterator<char> >
money_get<wchar_t, istreambuf_iterator<wchar_t> >
money_put<char, ostreambuf_iterator<char> >
money_put<wchar_t, ostreambuf_iterator<wchar_t> >
```

Category **numeric** (LC_NUMERIC) includes the facets:

```
num_get<char, istreambuf_iterator<char> >
num_get<wchar_t, istreambuf_iterator<wchar_t> >
num_put<char, ostreambuf_iterator<char> >
num_put<wchar_t, ostreambuf_iterator<wchar_t> >
numpunct<char>
numpunct<wchar_t>
```

Category **time** (LC_TIME) includes the facets:

```
time_get<char, istreambuf_iterator<char> >
time_get<wchar_t, istreambuf_iterator<wchar_t> >
time_put<char, ostreambuf_iterator<char> >
time_put<wchar_t, ostreambuf_iterator<wchar_t> >
```

Category **messages** (LC_MESSAGES) includes the facets:

```
messages<char>
messages<wchar_t>
```

(The last category is required by Posix, but not the C Standard.)

Some of these predefined facets are used by the iostreams classes, to control the conversion of numeric values to and from text sequences.

An object of class locale also stores a locale name as an object of class string. Using an invalid locale name to construct a locale facet or a locale object throws an object of class runtime_error. The stored locale name is `"*"` if the locale object cannot be certain that a C-style locale corresponds exactly to that represented by the object. Otherwise, you can establish a matching locale within the Standard C Library, for the locale object **loc**, by calling **setlocale**(LC_ALL, **loc.**name**().c_str()**).

In this implementation, you can also call the static member function:

```
static locale empty( );
```

to construct a locale object that has no facets. It is also a transparent locale; if the template functions has_facet and use_facet cannot find the requested facet in a transparent locale, they consult first the global locale and then, if that is transparent, the classic locale. Thus, you can write:

```
cout.imbue(locale::empty( ));
```

Subsequent insertions to cout are mediated by the current state of the global locale. You can even write:

```
locale loc(locale::empty( ), locale::classic( ),
    locale::numeric);
cout.imbue(loc);
```

Numeric formatting rules for subsequent insertions to `cout` remain the same as in the C locale, even as the global locale supplies changing rules for inserting dates and monetary amounts.

Requirements

Header: <locale>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

locale Members

<locale> Members

locale Members

Reference

Constructors

locale	Creates a locale, or a copy of a locale, or a copy of locale where a facet or a category has been replaced by a facet or category from another locale.

Typedefs

category	An integer type that provides bitmask values to denote standard facet families.

Member Functions

combine	Inserts a facet from a specified locale into a target locale.
name	Returns the stored locale name.

Static Functions

classic	The static member function returns a locale object that represents the classic C locale.
global	Resets the default local for the program.

Operators

operator!=	Tests two locales for inequality.
operator( )	Compares two **basic_string** objects.
operator==	Tests two locales for equality.

Classes

facet	A class that serves as the base class for all locale facets.
id	The member class provides a unique facet identification used as an index for looking up facets in a locale.

See Also

Reference

locale Class

Thread Safety in the Standard C++ Library

locale Typedefs

For information about the typedefs in the locale class, see locale Members.

locale::category

An integer type that provides bitmask values to denote standard facet families.

```
typedef int category;
   static const category none, collate, ctype, monetary,
   numeric, time, messages, all;
```

Remarks

The type is a synonym for an **int** type that can represent a group of distinct elements of a bitmask type local to class locale or can be used to represent any of the corresponding C locale categories. The elements are:

- **collate**, corresponding to the C category LC_COLLATE
- **ctype**, corresponding to the C category LC_CTYPE
- **monetary**, corresponding to the C category LC_MONETARY
- **numeric**, corresponding to the C category LC_NUMERIC
- **time**, corresponding to the C category LC_TIME
- **messages**, corresponding to the Posix category LC_MESSAGES

In addition, two useful values are:

- **none**, corresponding to none of the C categories
- **all**, corresponding to the C union of all categories LC_ALL

You can represent an arbitrary group of categories by using **OR** with these constants, as in **monetary** | **time**.

Requirements

Header: <locale>

See Also

Reference

locale Class

Other Resources

locale Members

locale Static Functions

For information about the static functions in the locale class, see locale Members.

locale::classic

The static member function returns a locale object that represents the classic C locale.

```
static const locale& classic( );
```

Return Value

A reference to the C locale.

Remarks

The classic C locale is the U.S. English ASCII locale within the Standard C Library that is implicitly used in programs that are not internationalized.

Example

```
// locale_classic.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
#include <locale>

using namespace std;

int main( )
{
   locale loc1( "german" );
   locale loc2 = locale::global( loc1 );
   cout << "The name of the previous locale is: " << loc2.name( )
        << "." << endl;
   cout << "The name of the current locale is: " << loc1.name( )
        << "." << endl;

   if (loc2 == locale::classic( ) )
      cout << "The previous locale was classic." << endl;
   else
      cout << "The previous locale was not classic." << endl;

   if (loc1 == locale::classic( ) )
      cout << "The current locale is classic." << endl;
   else
      cout << "The current locale is not classic." << endl;
}
```

Output

```
The name of the previous locale is: C.
The name of the current locale is: German_Germany.1252.
The previous locale was classic.
The current locale is not classic.
```

Requirements

Header: <locale>

See Also

Reference

locale Class

Other Resources

locale Members

locale::global

Resets the default local for the program.

```
static locale global(
   const locale& _Loc
);
```

Parameters

_Loc

The locale to be used as the default locale by the program.

Return Value

The previous locale before the default locale was reset.

Remarks

At program startup, the global locale is the same as the classic locale.

Example

```
// locale_global.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;

int main( )
{
   locale loc ( "German_germany" );
   locale loc1;
   cout << "The initial locale is: " << loc1.name( ) << endl;
   locale loc2 = locale::global ( loc );
   locale loc3;
   cout << "The current locale is: " << loc3.name( ) << endl;
   cout << "The previous locale was: " << loc2.name( ) << endl;
}
```

Output

```
The initial locale is: C
The current locale is: German_Germany.1252
The previous locale was: C
```

Requirements

Header: <locale>

See Also

Reference

locale Class

Other Resources

locale Members

locale Member Functions

For information about the member functions in the locale class, see locale Members.

locale::combine

Inserts a facet from a specified locale into a target locale.

```
template<class Facet>
   locale combine(
      const locale& _Loc
   ) const;
```

Parameters

_Loc

The locale containing the facet to be inserted into the target locale.

Return Value

The member function returns a locale object that replaces in or adds to ***this** the facet **Facet** listed in *_Loc*.

Example

```
// locale_combine.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;

int main() {
   locale loc ( "German_germany" );
   _TCHAR * s1 = _T("Das ist wei\x00dfzz."); // \x00df is the German sharp-s; it comes befo
re z in the German alphabet
   _TCHAR * s2 = _T("Das ist weizzz.");
   int result1 = use_facet<collate<_TCHAR> > ( loc ).
      compare (s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
   cout << isalpha (_T ( '\x00df' ), loc ) << result1 << endl;

   locale loc2 ( "C" );
   int result2 = use_facet<collate<_TCHAR> > ( loc2 ).
      compare (s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
   cout << isalpha (_T ( '\x00df' ), loc2 )  << result2 << endl;

   locale loc3 = loc2.combine<collate<_TCHAR> > (loc);
   int result3 = use_facet<collate<_TCHAR> > ( loc3 ).
      compare (s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
   cout << isalpha (_T ( '\x00df' ), loc3 ) << result3 << endl;
}
```

Sample Output

```
1-1
01
0-1
```

Requirements

Header: <locale>

See Also

Reference

locale Class

Other Resources

locale Members

locale::locale

Creates a locale, or a copy of a locale, or a copy of locale where a facet or a category has been replaced by a facet or category from another locale.

```
locale( );
explicit locale(
   const char* _Locname
);
locale(
   const locale& _Loc,
   const locale& _Other,
   category _Cat
);
locale(
   const locale& _Loc,
   const char* _Locname,
   category _Cat
);
template<class Facet>
   locale(
      const locale& _Loc,
      Facet* _Fac
   );
```

Parameters

_Locname

Name of a locale.

_Loc

A locale that is to be copied in constructing the new locale.

_Other

A locale from which to select a category.

_Cat

The category to be substituted into the constructed locale.

_Fac

The facet to be substituted into the constructed locale.

Remarks

The first constructor initializes the object to match the global locale. The second constructor initializes all the locale categories to have behavior consistent with the locale name locname. The remaining constructors copy *_Loc*, with the exceptions noted:

locale(**const locale**& *_Loc*, **const locale**& *_Other*, **category** *_Cat*);

replaces from *_Other* those facets corresponding to a category *C* for which *C* & *_Cat* is nonzero.

locale(**const locale**& *_Loc*, **const char*** *_Locname*, **category** *_Cat*);

replaces from **locale**(*_Locname*, *_All*) those facets corresponding to a category *C* for which *C* & *_Cat* is nonzero.

template<**class Facet**> **locale**(**const locale**& *_Loc*, **Facet*** *_Fac*);

replaces in (or adds to) *_Loc* the facet *_Fac*, if *_Fac* is not a null pointer.

If a locale name *_Locname* is a null pointer or otherwise invalid, the function throws runtime_error.

Example

```
// locale_locale.cpp
// compile with: /EHsc
```

```
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;

int main( ) {

   // Second constructor
   locale loc ( "German_germany" );
   _TCHAR * s1 = _T("Das ist wei\x00dfzz."); // \x00df is the German sharp-s, it comes befo
re z in the German alphabet
   _TCHAR * s2 = _T("Das ist weizzz.");
   int result1 = use_facet<collate<_TCHAR> > ( loc ).
      compare (s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
   cout << isalpha (_T ( '\x00df' ), loc ) << result1 << endl;

   // The first (default) constructor
   locale loc2;
   int result2 = use_facet<collate<_TCHAR> > ( loc2 ).
      compare (s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
   cout << isalpha (_T ( '\x00df' ), loc2 )  << result2 << endl;

   // Third constructor
   locale loc3 (loc2,loc, _M_COLLATE );
   int result3 = use_facet<collate<_TCHAR> > ( loc3 ).
      compare (s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
   cout << isalpha (_T ( '\x00df' ), loc3 ) << result3 << endl;

   // Fourth constructor
   locale loc4 (loc2, "German_Germany", _M_COLLATE );
   int result4 = use_facet<collate<_TCHAR> > ( loc4 ).
      compare (s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
   cout << isalpha (_T ( '\x00df' ), loc4 ) << result4 << endl;
}
```

Sample Output

```
1-1
01
0-1
0-1
```

Requirements

Header: <locale>

See Also

Reference

locale Class

Other Resources

locale Members

locale::name

Returns the stored locale name.

```
string name( ) const;
```

Return Value

A string giving the name of the locale.

Example

```
// locale_name.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
#include <locale>

using namespace std;

int main( )
{
   locale loc1( "german" );
   locale loc2 = locale::global( loc1 );
   cout << "The name of the previous locale is: "
        << loc2.name( ) << "." << endl;
   cout << "The name of the current locale is: "
        << loc1.name( ) << "." << endl;
}
```

Output

```
The name of the previous locale is: C.
The name of the current locale is: German_Germany.1252.
```

Requirements

Header: <locale>

See Also

Reference

locale Class

Other Resources

locale Members

locale Classes

For information about the classes in the locale class, see locale Members.

id Class

The member class provides a unique facet identification used as an index for looking up facets in a locale.

```
class id {
protected:
   id( );
private:
   id(const id&)              // not defined
   void operator=(const id&)  // not defined
   };
```

Remarks

The member class describes the static member object required by each unique locale facet. Note that you cannot copy or assign an object of class **id**.

Requirements

Header: <locale>

See Also

Reference

locale Class

Thread Safety in the Standard C++ Library

Other Resources

locale Members

facet Class

A class that serves as the base class for all locale facets.

```
class facet {
protected:
   explicit facet(
      size_t _Refs = 0
);
   virtual ~facet( );
private:
   facet(const facet&)           // not defined
   void operator=(const facet&)  // not defined
   };
```

Remarks

Note that you cannot copy or assign an object of class facet. You can construct and destroy objects derived from class **locale::facet** but not objects of the base class proper. Typically, you construct an object *_Myfac* derived from facet when you construct a locale, as in **locale loc**(**locale::classic**( ), **new** *_Myfac*);

In such cases, the constructor for the base class facet should have a zero *_Refs* argument. When the object is no longer needed, it is deleted. Thus, you supply a nonzero *_Refs* argument only in those rare cases where you take responsibility for the lifetime of the object.

Requirements

Header: <locale>

See Also

Reference

locale Class

Thread Safety in the Standard C++ Library

Other Resources

locale Members

locale Operators

For information about the operators in the locale class, see locale Members.

locale::operator!=

Tests two locales for inequality.

```
bool operator!=(
   const locale& _Right
) const;
```

Parameters

_Right

One of the locales to be tested for inequality.

Return Value

A Boolean value that is **true** if the locales are not copies of the same locale; **false** if the locales are copies of the same locale.

Remarks

Two locales are equal if they are the same locale, if one is a copy of the other, or if they have identical names.

Example

```
// locale_op_ne.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
#include <locale>

using namespace std;

int main( )
{
   locale loc1( "German_Germany" );
   locale loc2( "German_Germany" );
   locale loc3( "English" );

   if ( loc1 != loc2 )
      cout << "locales loc1 (" << loc1.name( )
      << ") and\n loc2 (" << loc2.name( ) << ") are not equal." << endl;
   else
      cout << "locales loc1 (" << loc1.name( )
      << ") and\n loc2 (" << loc2.name( ) << ") are equal." << endl;

   if ( loc1 != loc3 )
      cout << "locales loc1 (" << loc1.name( )
      << ") and\n loc3 (" << loc3.name( ) << ") are not equal." << endl;
   else
      cout << "locales loc1 (" << loc1.name( )
      << ") and\n loc3 (" << loc3.name( ) << ") are equal." << endl;
}
```

Output

```
locales loc1 (German_Germany.1252) and
 loc2 (German_Germany.1252) are equal.
locales loc1 (German_Germany.1252) and
 loc3 (English_United States.1252) are not equal.
```

Requirements

Header: <locale>

See Also

Reference

locale Class

Other Resources

locale Members

locale::operator()

Compares two **basic_string** objects.

```
template<Class CharType, class Traits, class Allocator>
   bool operator()(
      const basic_string<Elem, Traits, Allocator >& _Left,
      const basic_string<Elem, Traits, Allocator >& _Right
   );
```

Parameters

_Left

The left string.

_Right

The right string.

Return Value

The member function returns:

- –1 if the first sequence compares less than the second sequence.
- +1 if the second sequence compares less than the first sequence.
- 0 if the sequences are equivalent.

Remarks

The member function effectively executes:

```
const collate<CharType>& fac = use_fac<collate<CharType> >(*this);
return (fac.compare(_Left.begin( ),_Left.end( ),_Right.begin( ),_Right.end( )) < 0);
```

Thus, you can use a locale object as a function object.

Example

```
// locale_op_compare.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
#include <locale>

int main( )
{
   using namespace std;
   wchar_t *sa = L"ztesting";
   wchar_t *sb = L"\0x00DFtesting";
   basic_string<wchar_t> a( sa );
   basic_string<wchar_t> b( sb );

   locale loc( "German_Germany" );
   cout << loc( a,b ) << endl;

   const collate<wchar_t>& fac = use_facet<collate<wchar_t> >( loc );
   cout << ( fac.compare( sa, sa + a.length( ),
       sb, sb + b.length( ) ) < 0) << endl;
}
```

Output

```
0
```

0

Requirements

Header: <locale>

See Also

Reference

locale Class

Other Resources

locale Members

locale::operator==

Tests two locales for equality.

```
bool operator==(
   const locale& _Right
) const;
```

Parameters

_Right

One of the locales to be tested for equality.

Return Value

A Boolean value that is **true** if the locales are copies of the same locale; **false** if the locales are not copies of the same locale.

Remarks

Two locales are equal if they are the same locale, if one is a copy of the other, or if they have identical names.

Example

```
// locale_op_eq.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
#include <locale>

using namespace std;

int main( )
{
   locale loc1( "German_Germany" );
   locale loc2( "German_Germany" );
   locale loc3( "English" );

   if ( loc1 == loc2 )
      cout << "locales loc1 (" << loc1.name( )
      << ")\n and loc2 (" << loc2.name( ) << ") are equal."
      << endl;
   else
      cout << "locales loc1 (" << loc1.name( )
      << ")\n and loc2 (" << loc2.name( ) << ") are not equal."
      << endl;

   if ( loc1 == loc3 )
      cout << "locales loc1 (" << loc1.name( )
      << ")\n and loc3 (" << loc3.name( ) << ") are equal."
      << endl;
   else
      cout << "locales loc1 (" << loc1.name( )
      << ")\n and loc3 (" << loc3.name( ) << ") are not equal."
      << endl;
}
```

Output

```
locales loc1 (German_Germany.1252)
 and loc2 (German_Germany.1252) are equal.
locales loc1 (German_Germany.1252)
 and loc3 (English_United States.1252) are not equal.
```

Requirements

Header: <locale>

See Also

Reference

locale Class

Other Resources

locale Members

messages Class

The template class describes an object that can serve as a locale facet to retrieve localized messages from a catalog of internationalized messages for a given locale.

Currently, while the messages class is implemented, there are no messages.

For a list of all members of this type, see messages Members.

```
template <class CharType>
class messages : public messages_base
```

Parameters

CharType

The type used within a program to encode characters in a locale.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id.**

This facet basically opens a catalog of messages defined in the base class messages_base, retrieves the information required, and closes the catalog.

Requirements

Header: <locale>

See Also

Reference

messages_base Class

Thread Safety in the Standard C++ Library

Other Resources

messages Members

<locale> Members

messages Members

The following tables list the members of the messages Class. For a list of members inherited by the **messages** class, see messages_base Class.

Reference

Constructors

messages	The message facet constructor function.

Typedefs

char_type	A character type that is used display messages.
string_type	A type that describes a string of type **basic_string** containing characters of type **CharType**.

Member Functions

close	Closes the message catalog.
do_close	A virtual function called to lose the message catalog.
do_get	A virtual function called to retrieve the message catalog.
do_open	A virtual function called to open the message catalog.
get	Retrieves the message catalog.
open	Opens the message catalog.

See Also

Reference

messages Class
messages_base Class
Thread Safety in the Standard C++ Library

messages Typedefs

For information about the typedefs in the messages class, see messages Members.

messages::char_type

A character type that is used display messages.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

Requirements

Header: <locale>

See Also

Reference

messages Class

Other Resources

messages Members

messages::string_type

A type that describes a string of type **basic_string** containing characters of type **CharType**.

```
typedef basic_string<CharType> string_type;
```

Remarks

The type describes a specialization of template class basic_string whose objects can store copies of the message sequences.

Requirements

Header: <locale>

See Also

Reference

messages Class

Other Resources

messages Members

messages Member Functions

For information about the member functions in the messages class, see messages Members.

messages::close

Closes the message catalog.

```
void close(
   catalog _Catval
) const;
```

Parameters

_Catval

The catalog to be closed.

Remarks

The member function calls do_close(*_Catval*).

Requirements

Header: <locale>

See Also

Reference

messages Class

Other Resources

messages Members

messages::do_close

A virtual function called to lose the message catalog.

```
virtual void do_close(
   catalog _Catval
) const;
```

Parameters

_Catval

The catalog to be closed.

Remarks

The protected member function closes the message catalog *_Catval*, which must have been opened by an earlier call to do_open.

_Catval must be obtained from a previously opened catalog that is not closed.

Example

See the example for close, which calls **do_close**.

Requirements

Header: <locale>

See Also

Reference

messages Class

Other Resources

messages Members

messages::do_get

A virtual function called to retrieve the message catalog.

```
virtual string_type do_get(
   catalog _Catval,
   int _Set,
   int _Message,
   const string_type& _Dfault
) const;
```

Parameters

_Catval

The identification value specifying the message catalog to be searched.

_Set

The first identified used to locate a message in a message catalog.

_Message

The second identified used to locate a message in a message catalog.

_Dfault

The string to be returned on failure.

Return Value

It returns a copy of *_Dfault* on failure. Otherwise, it returns a copy of the specified message sequence.

Remarks

The protected member function tries to obtain a message sequence from the message catalog *_Catval*. It may make use of *_Set*, *_Message*, and *_Dfault* in doing so.

Example

See the example for get, which calls **do_get**.

Requirements

Header: <locale>

See Also

Reference

messages Class

Other Resources

messages Members

messages::do_open

A virtual function called to open the message catalog.

```
virtual catalog do_open(
   const string& _Catname,
   const locale& _Loc
) const;
```

Parameters

_Catname

The name of the catalog to be searched.

_Loc

The locale being searched for in the catalog.

Return Value

It returns a value that compares less than zero on failure. Otherwise, the returned value can be used as the first argument on a later call to get.

Remarks

The protected member function tries to open a message catalog whose name is *_Catname*. It may make use of the locale *_Loc* in doing so

The return value should be used as the argument on a later call to close.

Example

See the example for open, which calls **do_open**.

Requirements

Header: <locale>

See Also

Reference

messages Class

Other Resources

messages Members

messages::get

Retrieves the message catalog.

```
string_type get(
   catalog _CatVal,
   int _Set,
   int _Message,
   const string_type& _Dfault
) const;
```

Parameters

_Catval

The identification value specifying the message catalog to be searched.

_Set

The first identified used to locate a message in a message catalog.

_Message

The second identified used to locate a message in a message catalog.

_Dfault

The string to be returned on failure.

Return Value

It returns a copy of *_Dfault* on failure. Otherwise, it returns a copy of the specified message sequence.

Remarks

The member function returns do_get(*_Catval*, *_Set*, *_Message*, *_Dfault*).

Requirements

Header: <locale>

See Also

Reference

messages Class

Other Resources

messages Members

messages::messages

The message facet constructor function.

```
explicit messages(
   size_t _Refs = 0
);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 0: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with **locale::**facet(*_Refs*).

Requirements

Header: <locale>

See Also

Reference

messages Class

Other Resources

messages Members

messages::open

Opens the message catalog.

```
catalog open(
   const string& _Catname,
   const locale& _Loc
) const;
```

Parameters

_Catname

The name of the catalog to be searched.

_Loc

The locale being searched for in the catalog.

Return Value

It returns a value that compares less than zero on failure. Otherwise, the returned value can be used as the first argument on a later call to get.

Remarks

The member function returns do_open(*_Catname*, *_Loc*).

Requirements

Header: <locale>

See Also

Reference

messages Class

Other Resources

messages Members

messages_base Class

The base class describes an **int** type for the catalog of messages.

```
class messages_base {
    typedef int catalog;
};
```

Remarks

The type catalog is a synonym for type **int** that describes the possible return values from messages::do_open.

Requirements

Header: <locale>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<locale> Members

messages_byname Class

The derived template class describes an object that can serve as a message facet of a given locale, enabling the retrieval of localized messages.

```
template<class Elem>
   class messages_byname : public messages<CharType> {
public:
   explicit messages_byname(
      const char* _Locname,
      size_t _Refs = 0
);
protected:
    ~messages_byname( );
    };
```

Remarks

Its behavior is determined by the named locale *_Locname*. The constructor initializes its base object with messages<**CharType**>(*_Refs*).

Requirements

Header: <locale>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<locale> Members

money_base Class

The class describes an enumeration and a structure common to all specializations of template class moneypunct.

```
class money_base {
    enum part {none, sign, space, symbol, value};
    struct pattern {
        char field[4];
        };
    };
```

Remarks

The enumeration **part** describes the possible values in elements of the array field in the structure pattern. The values of **part** are:

- **none** to match zero or more spaces or generate nothing.
- **sign** to match or generate a positive or negative sign.
- **space** to match zero or more spaces or generate a space.
- **symbol** to match or generate a currency symbol.
- **value** to match or generate a monetary value.

Requirements

Header: <locale>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<locale> Members

money_get Class

The template class describes an object that can serve as a locale facet to control conversions of sequences of type **CharType** to monetary values.

For a list of all members of this type, see money_get Members.

```
template<
   class CharType,
   class InputIterator = istreambuf_iterator<CharType>
>
class money_get : public locale::facet
```

Parameters

CharType

The type used within a program to encode characters in a locale.

InputIterator

The type of iterator from which the get functions read their input.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id.**

Requirements

Header: <locale>

See Also

Reference

facet Class

Thread Safety in the Standard C++ Library

Other Resources

money_get Members

<locale> Members

money_get Members

The following tables list the members of the money_get Class. For a list of members inherited by the **money_get** class, see facet Class.

Reference

Constructors

money_get	The constructor for objects of type **money_get** that are used to extract numerical values from sequences representing monetary values.

Typedefs

char_type	A type that is used to describe a character used by a locale.
iter_type	A type that describes an input iterator.
string_type	A type that describes a string containing characters of type **CharType**.

Member Functions

do_get	A virtual function called to extracts a numerical value from a character sequence that represents a monetary value.
get	Extracts a numerical value from a character sequence that represents a monetary value.

See Also

Reference

money_get Class
facet Class
Thread Safety in the Standard C++ Library

money_get Typedefs

For information about the typedefs in the money_get class, see money_get Members.

money_get::char_type

A type that is used to describe a character used by a locale.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

Requirements

Header: <locale>

See Also

Reference

money_get Class

Other Resources

money_get Members

money_get::iter_type

A type that describes an input iterator.

```
typedef InputIterator iter_type;
```

Remarks

The type is a synonym for the template parameter **InputIterator**.

Requirements

Header: <locale>

See Also

Reference

money_get Class

Other Resources

money_get Members

money_get::string_type

A type that describes a string containing characters of type **CharType**.

```
typedef basic_string<CharType> string_type;
```

Remarks

The type describes a specialization of template class basic_string.

Requirements

Header: <locale>

See Also

Reference

money_get Class

Other Resources

money_get Members

money_get Member Functions

For information about the member functions in the money_get class, see money_get Members.

money_get::do_get

Virtual function called to extracts a numerical value from a character sequence that represents a monetary value.

```
virtual iter_type do_get(
   iter_type _First,
   iter_type _Last,
   bool _Intl,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   long double& _Val
) const
virtual iter_type do_get(
   iter_type _First,
   iter_type _Last,
   bool _Intl,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   string_type& _Val
) const
```

Parameters

_First

Input iterator addressing the beginning of the sequence to be converted.

_Last

Input iterator addressing the end of the sequence to be converted.

_Intl

A Boolean value indicating the type of currency symbol expected in the sequence: **true** if international, **false** if domestic.

_Iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

_State

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded or not.

_Val

A string storing the converted sequence.

Return Value

An input iterator addressing the first element beyond the monetary input field.

Remarks

The first virtual protected member function tries to match sequential elements beginning at first in the sequence [*_First*, *_Last*) until it has recognized a complete, nonempty monetary input field. If successful, it converts this field to a sequence of one or more decimal digits, optionally preceded by a minus sign (–), to represent the amount and stores the result in the string_type object *_Val*. It returns an iterator designating the first element beyond the monetary input field. Otherwise, the function stores an empty sequence in *_Val* and sets **ios_base::failbit** in *_State*. It returns an iterator designating the first element beyond any prefix of a valid monetary input field. In either case, if the return value equals *_Last*, the function sets **ios_base::eofbit** in *_State*.

The second virtual protected member function behaves the same as the first, except that if successful it converts the optionally signed digit sequence to a value of type **long double** and stores that value in *_Val*.

The format of a monetary input field is determined by the locale facet **fac** returned by the effective call use_facet <moneypunct<**CharType**, **intl**>>(**iosbase**.getloc).

Specifically:

- **fac**.neg_format determines the order in which components of the field occur.

- **fac**.curr_symbol determines the sequence of elements that constitutes a currency symbol.
- **fac**.positive_sign determines the sequence of elements that constitutes a positive sign.
- **fac**.negative_sign determines the sequence of elements that constitutes a negative sign.
- **fac**.grouping determines how digits are grouped to the left of any decimal point.
- **fac**.thousands_sep determines the element that separates groups of digits to the left of any decimal point.
- **fac**.decimal_point determines the element that separates the integer digits from the fraction digits.
- **fac**.frac_digits determines the number of significant fraction digits to the right of any decimal point.

If the sign string (**fac**.**negative_sign** or **fac**.**positive_sign**) has more than one element, only the first element is matched where the element equal to **money_base::sign** appears in the format pattern (**fac**.**neg_format**). Any remaining elements are matched at the end of the monetary input field. If neither string has a first element that matches the next element in the monetary input field, the sign string is taken as empty and the sign is positive.

If **iosbase**.flags &showbase is nonzero, the string **fac**.**curr_symbol** must match where the element equal to **money_base::symbol** appears in the format pattern. Otherwise, if **money_base::symbol** occurs at the end of the format pattern, and if no elements of the sign string remain to be matched, the currency symbol is not matched. Otherwise, the currency symbol is optionally matched.

If no instances of **fac**.**thousands_sep** occur in the value portion of the monetary input field (where the element equal to **money_base::value** appears in the format pattern), no grouping constraint is imposed. Otherwise, any grouping constraints imposed by **fac**.**grouping** is enforced. Note that the resulting digit sequence represents an integer whose low-order **fac**.**frac_digits** decimal digits are considered to the right of the decimal point.

Arbitrary white space is matched where the element equal to **money_base::space** appears in the format pattern, if it appears other than at the end of the format pattern. Otherwise, no internal white space is matched. An element *ch* is considered white space if use_facet <ctype<**CharType**> >(**iosbase**.getloc).is(**ctype_base::space**, *ch*) is **true**.

Example

See the example for get, which calls **do_get**.

Requirements

Header: <locale>

See Also

Reference

money_get Class

Other Resources

money_get Members

money_get::get

Extracts a numerical value from a character sequence that represents a monetary value.

```
iter_type get(
   iter_type _First,
   iter_type _Last,
   bool _Intl,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   long double& _Val
) const;
iter_type get(
   iter_type _First,
   iter_type _Last,
   bool _Intl,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   string_type& _Val
) const;
```

Parameters

_First

Input iterator addressing the beginning of the sequence to be converted.

_Last

Input iterator addressing the end of the sequence to be converted.

_Intl

A Boolean value indicating the type of currency symbol expected in the sequence: **true** if international, **false** if domestic.

_Iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required

_State

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

_Val

A string storing the converted sequence.

Return Value

An input iterator addressing the first element beyond the monetary input field.

Remarks

Both member functions return do_get(*_First*, *_Last*, *_Intl*, *_Iosbase*, *_State*, *_Val*).

Example

```
// money_get_get.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;

int main( )
{
   locale loc( "german_germany" );

   basic_stringstream< char > psz;
```

```
    psz << use_facet<moneypunct<char, 1> >(loc).curr_symbol() << "-1.000,56";
    basic_stringstream< char > psz2;
    psz2 << "-100056" << use_facet<moneypunct<char, 1> >(loc).curr_symbol();

    ios_base::iostate st = 0;
    long double fVal;

    psz.flags( psz.flags( )|ios_base::showbase );
    // Which forced the READING the currency symbol
    psz.imbue(loc);
    use_facet < money_get < char > >( loc ).
       get( basic_istream<char>::_Iter( psz.rdbuf( ) ),
            basic_istream<char>::_Iter( 0 ), true, psz, st, fVal );

    if ( st & ios_base::failbit )
       cout << "money_get(" << psz.str( ) << ", intl = 1) FAILED"
            << endl;
    else
       cout << "money_get(" << psz.str( ) << ", intl = 1) = "
            << fVal/100.0 << endl;

    use_facet < money_get < char > >( loc ).
       get(basic_istream<char>::_Iter(psz2.rdbuf( )),
           basic_istream<char>::_Iter(0), false, psz2, st, fVal);

    if ( st & ios_base::failbit )
       cout << "money_get(" << psz2.str( ) << ", intl = 0) FAILED"
            << endl;
    else
       cout << "money_get(" << psz2.str( ) << ", intl = 0) = "
            << fVal/100.0 << endl;
};
```

Sample Output

If you run Windows 2000 or earlier, you will get this output:

```
money_get(DEM-1.000,56, intl = 1) = -1000.56
money_get(-100056DM, intl = 0) = -1000.56
```

If you run Windows XP, you will get this output:

```
money_get(EUR-1.000,56, intl = 1) = -1000.56
money_get(-100056EUR, intl = 0) = -1000.56
```

Requirements

Header: <locale>

See Also

Reference

money_get Class

Other Resources

money_get Members

money_get::money_get

The constructor for objects of type **money_get** that are used to extract numerical values from sequences representing monetary values.

```
explicit money_get(
   size_t _Refs = 0
);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 0: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with **locale::**facet(*_Refs*).

Requirements

Header: <locale>

See Also

Reference

money_get Class

Other Resources

money_get Members

money_put Class

The template class describes an object that can serve as a locale facet to control conversions of monetary values to sequences of type **CharType**.

For a list of all members of this type, see money_put Members.

```
template<
   class CharType,
   class OutputIterator = ostreambuf_iterator<CharType>
>
class money_put : public locale::facet
```

Parameters

CharType

The type used within a program to encode characters in a locale.

OutputIterator

The type of iterator to which the monetary put functions write their output.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id.**

Requirements

Header: <locale>

See Also

Reference

facet Class

Thread Safety in the Standard C++ Library

Other Resources

money_put Members

<locale> Members

money_put Members

The following tables list the members of the money_put Class. For a list of members inherited by the **money_put** class, see facet Class.

Reference

Constructors

money_put	The constructor for objects of type **money_put**.

Typedefs

char_type	A type that is used to describe a character used by a locale.
iter_type	A type that describes an output iterator.
string_type	A type that describes a string containing characters of type **CharType**.

Member Functions

do_put	A virtual function called to convert either number or a string to a character sequence that represents a monetary value.
put	Converts either number or a string to a character sequence that represents a monetary value.

See Also

Reference

money_put Class
facet Class
Thread Safety in the Standard C++ Library

money_put Typedefs

For information about the typedefs in the money_put class, see money_put Members.

money_put::char_type

A type that is used to describe a character used by a locale.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

Requirements

Header: <locale>

See Also

Reference

money_put Class

Other Resources

money_put Members

money_put::iter_type

A type that describes an output iterator.

```
typedef InIt iter_type;
```

Remarks

The type is a synonym for the template parameter **OutputIterator.**

Requirements

Header: <locale>

See Also

Reference

money_put Class

Other Resources

money_put Members

money_put::string_type

A type that describes a string containing characters of type **CharType**.

```
typedef basic_string<CharType> string_type;
```

Remarks

The type describes a specialization of template class basic_string whose objects can store sequences of elements from the source sequence.

Requirements

Header: <locale>

See Also

Reference

money_put Class

Other Resources

money_put Members

money_put Member Functions

For information about the member functions in the money_put class, see money_put Members.

money_put::do_put

A virtual function called to convert either number or a string to a character sequence that represents a monetary value.

```
virtual iter_type do_put(
   iter_type _Next,
   bool _Intl,
   ios_base& _Iosbase,
   CharType _Fill,
   string_type& _Val
) const;
virtual iter_type do_put(
   iter_type _Next,
   bool _Intl,
   ios_base& _Iosbase,
   CharType _Fill,
   long double& _Val
) const;
```

Parameters

_Next

An iterator addressing the first element of the inserted string.

_Intl

A Boolean value indicating the type of currency symbol expected in the sequence: **true** if international, **false** if domestic.

_Iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required

_Fill

A character which is used for spacing.

_Val

A string object to be converted.

Return Value

An output iterator the addresses the position one beyond the last element produced.

Remarks

The first virtual protected member function generates sequential elements beginning at *_Next* to produce a monetary output field from the string_type object *_Val*. The sequence controlled by *_Val* must begin with one or more decimal digits, optionally preceded by a minus sign (–), which represents the amount. The function returns an iterator designating the first element beyond the generated monetary output field.

The second virtual protected member function behaves the same as the first, except that it effectively first converts *_Val* to a sequence of decimal digits, optionally preceded by a minus sign, then converts that sequence as above.

The format of a monetary output field is determined by the locale facet fac returned by the (effective) call use_facet <moneypunct<**CharType**, **intl**> >(**iosbase**.getloc).

Specifically:

- **fac**.pos_format determines the order in which components of the field are generated for a nonnegative value.
- **fac**.neg_format determines the order in which components of the field are generated for a negative value.
- **fac**.curr_symbol determines the sequence of elements to generate for a currency symbol.
- **fac**.positive_sign determines the sequence of elements to generate for a positive sign.
- **fac**.negative_sign determines the sequence of elements to generate for a negative sign.

- **fac**.grouping determines how digits are grouped to the left of any decimal point.
- **fac**.thousands_sep determines the element that separates groups of digits to the left of any decimal point.
- **fac**.decimal_point determines the element that separates the integer digits from any fraction digits.
- **fac**.frac_digits determines the number of significant fraction digits to the right of any decimal point.

If the sign string (**fac**.**negative_sign** or **fac**.**positive_sign**) has more than one element, only the first element is generated where the element equal to **money_base::sign** appears in the format pattern (**fac**.**neg_format** or **fac**.**pos_format**). Any remaining elements are generated at the end of the monetary output field.

If **iosbase**.flags & showbase is nonzero, the string **fac**.**curr_symbol** is generated where the element equal to **money_base::symbol** appears in the format pattern. Otherwise, no currency symbol is generated.

If no grouping constraints are imposed by **fac**.**grouping** (its first element has the value CHAR_MAX), then no instances of **fac**.**thousands_sep** are generated in the value portion of the monetary output field (where the element equal to **money_base::value** appears in the format pattern). If **fac**.**frac_digits** is zero, then no instance of **fac**.**decimal_point** is generated after the decimal digits. Otherwise, the resulting monetary output field places the low-order **fac**.**frac_digits** decimal digits to the right of the decimal point.

Padding occurs as for any numeric output field, except that if **iosbase**.**flags** & **iosbase**.internal is nonzero, any internal padding is generated where the element equal to **money_base::space** appears in the format pattern, if it does appear. Otherwise, internal padding occurs before the generated sequence. The padding character is **fill**.

The function calls **iosbase**.**width**(0) to reset the field width to zero.

Example

See the example for put, where the virtual member function is called by **put**.

Requirements

Header: <locale>

See Also

Reference

money_put Class

Other Resources

money_put Members

money_put::money_put

The constructor for objects of type **money_put**.

```
explicit money_put(
   size_t _Refs = 0
);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: the lifetime of the object is managed by the locales that contain it.
- 1: the lifetime of the object must be manually managed.
- > 0: these values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with **locale::**facet(*_Refs*).

Requirements

Header: <locale>

See Also

Reference

money_put Class

Other Resources

money_put Members

money_put::put

Converts either number or a string to a character sequence that represents a monetary value.

```
iter_type put(
   iter_type _Next,
   bool _Intl,
   ios_base& _Iosbase,
   CharType _Fill,
   string_type& _Val
) const;
iter_type put(
   iter_type _Next,
   bool _Intl,
   ios_base& _Iosbase,
   CharType _Fill,
   long double& _Val
) const;
```

Parameters

_Next

An iterator addressing the first element of the inserted string.

_Intl

A Boolean value indicating the type of currency symbol expected in the sequence: **true** if international, **false** if domestic.

_Iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required

_Fill

A character which is used for spacing.

_Val

A string object to be converted.

Return Value

An output iterator the addresses the position one beyond the last element produced.

Remarks

Both member functions return do_put(*_Next*, *_Intl*, *_Iosbase*, *_Fill*, *_Val*).

Example

```
// money_put_put.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
//   locale loc( "german_germany" );
   locale loc( "english_canada" );
   basic_stringstream<char> psz, psz2;
   ios_base::iostate st = 0;

   psz2.imbue( loc );
   psz2.flags( psz2.flags( )|ios_base::showbase ); // force the printing of the currency sy
mbol
   use_facet < money_put < char > >(loc).put(basic_ostream<char>::_Iter( psz2.rdbuf( ) ), t
rue, psz2, st, 100012);
```

```
    if (st & ios_base::failbit)
       cout << "money_put( ) FAILED" << endl;
    else
       cout << "money_put( ) = \"" << psz2.rdbuf( )->str( ) <<"\""<< endl;

    st = 0;
 };
```

Output

```
money_put( ) = "CAD1,000.12"
```

Requirements

Header: <locale>

See Also

Reference

money_put Class

Other Resources

money_put Members

moneypunct Class

The template class describes an object that can serve as a locale facet to describe the sequences of type **CharType** used to represent a monetary input field or a monetary output field. If the template parameter **Intl** is **true**, international conventions are observed.

For a list of all members of this type, see moneypunct Members.

```
template<class CharType, bool Intl>
class moneypunct
```

Parameters

CharType

The type used within a program to encode characters.

Intl

A flag specifying whether international conventions are to be observed.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id.**

The const static object intl stores the value of the template parameter **Intl**.

Requirements

Header: <locale>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

moneypunct Members

<locale> Members

moneypunct Members

Reference

Constructors

moneypunct	Constructor of objects of type **moneypunct**.

Typedefs

char_type	A type that is used to describe a character used by a locale.
string_type	A type that describes a string containing characters of type **CharType**.

Member Functions

curr_symbol	Returns a locale-specific sequence of elements to use as a currency symbol.
decimal_point	Returns a locale-specific sequence of elements to use as a decimal point symbol.
do_curr_symbol	A protected virtual member function that returns a locale-specific sequence of elements to use as a currenc y symbol.
do_decimal_point	A protected virtual member function that is called to return a locale-specific sequence of elements to use as a decimal point symbol.
do_frac_digits	The protected virtual member function returns a locale-specific count of the number of digits to display to t he right of any decimal point.
do_grouping	The protected virtual member function returns a locale-specific rule for determining how digits are groupe d to the left of any decimal point.
do_neg_format	A protected virtual member function that is called to return a locale-specific rule for formatting outputs wit h negative amounts.
do_negative_sign	A protected virtual member function that is called to return a locale-specific sequence of elements to use as a negative sign symbol.
do_pos_format	A protected virtual member function that is called to return a locale-specific rule for formatting outputs wit h positive amounts.
do_positive_sign	A protected virtual member function that is called to return a locale-specific sequence of elements to use as a positive sign symbol.
do_thousands_sep	A protected virtual member function that is called to return a locale-specific sequence of elements to use as a thousands separator symbol.
frac_digits	Returns a locale-specific count of the number of digits to display to the right of any decimal point.
grouping	Returns a locale-specific rule for determining how digits are grouped to the left of any decimal point.
neg_format	Returns a locale-specific rule for formatting outputs with negative amounts.
negative_sign	Returns a locale-specific sequence of elements to use as a negative sign symbol.
pos_format	Returns a locale-specific rule for formatting outputs with positive amounts.

positive_sign	Returns a locale-specific sequence of elements to use as a positive sign symbol.
thousands_sep	Returns a locale-specific sequence of elements to use as a thousands separator symbol.

See Also

Reference

moneypunct Class

Thread Safety in the Standard C++ Library

moneypunct Typedefs

For information about the typedefs in the moneypunct class, see moneypunct Members.

moneypunct::char_type

A type that is used to describe a character used by a locale.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct Members

moneypunct::string_type

A type that describes a string containing characters of type **CharType**.

```
typedef basic_string<CharType> string_type;
```

Remarks

The type describes a specialization of template class basic_string whose objects can store copies of the punctuation sequences.

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct Members

moneypunct Member Functions

For information about the member functions in the moneypunct class, see moneypunct Members.

moneypunct::curr_symbol

Returns a locale-specific sequence of elements to use as a currency symbol.

```
string_type curr_symbol( ) const;
```

Return Value

A string containing the currency symbol.

Remarks

The member function returns do_curr_symbol.

Example

```
// moneypunct_curr_symbol.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "german_germany" );

   const moneypunct < char, true > &mpunct = use_facet < moneypunct < char, true > >(loc);
   cout << loc.name( ) << " international currency symbol "<<  mpunct.curr_symbol( ) << end
l;

   const moneypunct < char, false> &mpunct2 = use_facet < moneypunct < char, false> >(loc);
   cout << loc.name( ) << " domestic currency symbol "<<  mpunct2.curr_symbol( ) << endl;
};
```

Sample Output

If you run Windows 2000 or earlier, you will get this output:

```
German_Germany.1252 international currency symbol DEM
German_Germany.1252 domestic currency symbol DM
```

If you run Windows XP, you will get this output:

```
German_Germany.1252 international currency symbol EUR
German_Germany.1252 domestic currency symbol
```

This assumes that you have not changed the default currency settings.

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct Members

moneypunct::decimal_point

Returns a locale-specific sequence of elements to use as a decimal point symbol.

```
CharType decimal_point( ) const;
```

Return Value

A locale-specific sequence of elements to use as a decimal point symbol.

Remarks

The member function returns do_decimal_point.

Example

```
// moneypunct_decimal_pt.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc("german_germany");

   const moneypunct < char, true > &mpunct = use_facet
      < moneypunct < char, true > >(loc);
   cout << loc.name( ) << " international decimal point "
        << mpunct.decimal_point( ) << endl;

   const moneypunct < char, false> &mpunct2 = use_facet
      < moneypunct < char, false> >(loc);
   cout << loc.name( ) << " domestic decimal point "
        << mpunct2.decimal_point( ) << endl;
}
```

Output

```
German_Germany.1252 international decimal point ,
German_Germany.1252 domestic decimal point ,
```

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct Members

moneypunct::do_curr_symbol

A protected virtual member function that returns a locale-specific sequence of elements to use as a currency symbol.

```
string_type do_curr_symbol( ) const;
```

Return Value

A locale-specific sequence of elements to use as a decimal point symbol.

Example

See the example for curr_symbol, where the virtual member function is called by **curr_symbol**.

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct Members

moneypunct::do_decimal_point

A protected virtual member function that returns a locale-specific sequence of elements to use as a decimal point symbol.

```
CharType do_decimal_point( ) const;
```

Return Value

A locale-specific sequence of elements to use as a decimal point symbol.

Example

See the example for decimal_point, where the virtual member function is called by **decimal_point**.

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct Members

moneypunct::do_frac_digits

A protected virtual member function that returns a locale-specific count of the number of digits to display to the right of any decimal point.

```
int do_frac_digits( ) const;
```

Return Value

A locale-specific count of the number of digits to display to the right of any decimal point.

Example

See the example for frac_digits, where the virtual member function is called by **frac_digits**.

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct Members

moneypunct::do_grouping

A protected virtual member function that returns a locale-specific rule for determining how digits are grouped to the left of any decimal point.

```
string do_grouping( ) const;
```

Return Value

A locale-specific rule for determining how digits are grouped to the left of any decimal point.

Example

See the example for grouping, where the virtual member function is called by **grouping**.

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct Members

moneypunct::do_neg_format

A protected virtual member function that is called to return a locale-specific rule for formatting outputs with negative amounts.

```
pattern do_neg_format( ) const;
```

Return Value

The protected virtual member function returns a locale-specific rule for determining how to generate a monetary output field for a negative amount. Each of the four elements of **pattern::field** can have the values:

- **none** to match zero or more spaces or generate nothing.
- **sign** to match or generate a positive or negative sign.
- **space** to match zero or more spaces or generate a space.
- **symbol** to match or generate a currency symbol.
- **value** to match or generate a monetary value.

Components of a monetary output field are generated and components of a monetary input field are matched in the order in which these elements appear in **pattern::field**. Each of the values **sign**, **symbol**, **value**, and either **none** or **space** must appear exactly once. The value **none** must not appear first. The value space **must** not appear first or last. If **Intl** is true, the order is **symbol**, **sign**, **none**, then **value**.

The template version of **moneypunct**<**CharType**, **Intl**> returns {**money_base::symbol**, **money_base::sign**, **money_base::value**, **money_base::none**}.

Example

See the example for neg_format, where the virtual member function is called by **neg_format**.

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct Members

moneypunct::do_negative_sign

A protected virtual member function that is called to return a locale-specific sequence of elements to use as a negative sign symbol.

```
string_type do_negative_sign( ) const;
```

Return Value

A locale-specific sequence of elements to use as a negative sign.

Example

See the example for negative_sign, where the virtual member function is called by **negative_sign**.

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct Members

moneypunct::do_pos_format

A protected virtual member function that is called to return a locale-specific rule for formatting outputs with positive amounts.

```
pattern do_pos_format( ) const;
```

Return Value

The protected virtual member function returns a locale-specific rule for determining how to generate a monetary output field for a positive amount. (It also determines how to match the components of a monetary input field.) The encoding is the same as for do_neg_format.

The template version of moneypunct<**CharType**, **InputIterator**> returns {**money_base::symbol**, **money_base::sign**, **money_base::value**, **money_base::none**}.

Example

See the example for pos_format, where the virtual member function is called by **pos_format**.

Requirements

Header: <locale>

See Also
Reference
moneypunct Class
Other Resources
moneypunct Members

moneypunct::do_positive_sign

A protected virtual member function that returns a locale-specific sequence of elements to use as a positive sign.

```
string_type do_positive_sign( ) const;
```

Return Value

A locale-specific sequence of elements to use as a positive sign.

Example

See the example for positive_sign, where the virtual member function is called by **positive_sign**.

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct Members

moneypunct::do_thousands_sep

A protected virtual member function that returns a locale-specific element to use as a group separator to the left of any decimal point.

```
CharType do_thousands_sep( ) const;
```

Return Value

A locale-specific element to use as a group separator to the left of any decimal point.

Example

See the example for thousands_sep, where the virtual member function is called by **thousands_sep**.

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct Members

moneypunct::frac_digits

Returns a locale-specific count of the number of digits to display to the right of any decimal point.

```
int frac_digits( ) const;
```

Return Value

A locale-specific count of the number of digits to display to the right of any decimal point.

Remarks

The member function returns do_frac_digits.

Example

```
// moneypunct_frac_digits.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "german_germany" );

   const moneypunct <char, true> &mpunct =
       use_facet <moneypunct <char, true> >(loc);
   for (unsigned int i = 0; i <mpunct.grouping( ).length( ); i++ )
   {
      cout << loc.name( ) << " international grouping:\n the "
           << i <<"th group to the left of the radix character "
           << "is of size " << (int)(mpunct.grouping ( )[i])
           << endl;
   }
   cout << loc.name( ) << " international frac_digits\n to the right"
        << " of the radix character: "
        << mpunct.frac_digits ( ) << endl << endl;

   const moneypunct <char, false> &mpunct2 =
       use_facet <moneypunct <char, false> >(loc);
   for (unsigned int i = 0; i <mpunct2.grouping( ).length( ); i++ )
   {
      cout << loc.name( ) << " domestic grouping:\n the "
           << i <<"th group to the left of the radix character "
           << "is of size " << (int)(mpunct2.grouping ( )[i])
           << endl;
   }
   cout << loc.name( ) << " domestic frac_digits\n to the right"
        << " of the radix character: "
        << mpunct2.frac_digits ( ) << endl << endl;
}
```

Output

```
German_Germany.1252 international grouping:
 the 0th group to the left of the radix character is of size 3
German_Germany.1252 international frac_digits
 to the right of the radix character: 2

German_Germany.1252 domestic grouping:
 the 0th group to the left of the radix character is of size 3
German_Germany.1252 domestic frac_digits
 to the right of the radix character: 2
```

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct Members

moneypunct::grouping

Returns a locale-specific rule for determining how digits are grouped to the left of any decimal point.

```
string grouping( ) const;
```

Return Value

A locale-specific rule for determining how digits are grouped to the left of any decimal point.

Remarks

The member function returns do_grouping.

Example

```
// moneypunct_grouping.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "german_germany" );

   const moneypunct <char, true> &mpunct =
       use_facet <moneypunct <char, true> >( loc );
   for (unsigned int i = 0; i <mpunct.grouping( ).length( ); i++ )
   {
      cout << loc.name( ) << " international grouping:\n the "
           << i <<"th group to the left of the radix character "
           << "is of size " << (int)(mpunct.grouping ( )[i])
           << endl;
   }
   cout << loc.name( ) << " international frac_digits\n to the right"
        << " of the radix character: "
        << mpunct.frac_digits ( ) << endl << endl;

   const moneypunct <char, false> &mpunct2 =
       use_facet <moneypunct <char, false> >( loc );
   for (unsigned int i = 0; i <mpunct2.grouping( ).length( ); i++ )
   {
      cout << loc.name( ) << " domestic grouping:\n the "
           << i <<"th group to the left of the radix character "
           << "is of size " << (int)(mpunct2.grouping ( )[i])
           << endl;
   }
   cout << loc.name( ) << " domestic frac_digits\n to the right"
        << " of the radix character: "
        << mpunct2.frac_digits ( ) << endl << endl;
}
```

Output

```
German_Germany.1252 international grouping:
 the 0th group to the left of the radix character is of size 3
German_Germany.1252 international frac_digits
 to the right of the radix character: 2

German_Germany.1252 domestic grouping:
 the 0th group to the left of the radix character is of size 3
German_Germany.1252 domestic frac_digits
 to the right of the radix character: 2
```

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct Members

moneypunct::moneypunct

Constructor of objects of type **moneypunct**.

```
explicit moneypunct(
   size_t _Refs = 0
   );
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 0: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with **locale::**facet(*_Refs*).

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct Members

moneypunct::neg_format

Returns a locale-specific rule for formatting outputs with negative amounts.

```
pattern neg_format( ) const;
```

Return Value

A locale-specific rule for formatting outputs with negative amounts.

Remarks

The member function returns do_neg_format.

Example

```
// moneypunct_neg_format.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>

using namespace std;

int main( ) {
   locale loc( "german_germany" );

   const moneypunct <char, true> &mpunct =
      use_facet <moneypunct <char, true > >(loc);
   cout << loc.name( ) << " international negative number format: "
        << mpunct.neg_format( ).field[0]
        << mpunct.neg_format( ).field[1]
        << mpunct.neg_format( ).field[2]
        << mpunct.neg_format( ).field[3] << endl;

   const moneypunct <char, false> &mpunct2 =
      use_facet <moneypunct <char, false> >(loc);
   cout << loc.name( ) << " domestic negative number format: "
        << mpunct2.neg_format( ).field[0]
        << mpunct2.neg_format( ).field[1]
        << mpunct2.neg_format( ).field[2]
        << mpunct2.neg_format( ).field[3] << endl;
}
```

Sample Output

```
German_Germany.1252 international negative number format: $+vx
German_Germany.1252 domestic negative number format: +v $
```

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct Members

moneypunct::negative_sign

Returns a locale-specific sequence of elements to use as a negative sign symbol.

```
string_type negative_sign( ) const;
```

Return Value

Returns a locale-specific sequence of elements to use as a negative sign symbol.

Remarks

The member function returns do_negative_sign.

Example

```
// moneypunct_neg_sign.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>

using namespace std;

int main( )
{
   locale loc( "german_germany" );

   const moneypunct <char, true> &mpunct =
      use_facet <moneypunct <char, true> >(loc);
   cout << loc.name( ) << " international negative sign: "
        << mpunct.negative_sign( ) << endl;

   const moneypunct <char, false> &mpunct2 =
      use_facet <moneypunct <char, false> >(loc);
   cout << loc.name( ) << " domestic negative sign: "
        << mpunct2.negative_sign( ) << endl;

   locale loc2( "French" );

   const moneypunct <char, true> &mpunct3 =
      use_facet <moneypunct <char, true> >(loc2);
   cout << loc2.name( ) << " international negative sign: "
        << mpunct3.negative_sign( ) << endl;

   const moneypunct <char, false> &mpunct4 =
      use_facet <moneypunct <char, false> >(loc2);
   cout << loc2.name( ) << " domestic negative sign: "
        << mpunct4.negative_sign( ) << endl;
};
```

Output

```
German_Germany.1252 international negative sign: -
German_Germany.1252 domestic negative sign: -
French_France.1252 international negative sign: -
French_France.1252 domestic negative sign: -
```

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct Members

moneypunct::pos_format

Returns a locale-specific rule for formatting outputs with positive amounts.

```
pattern pos_format( ) const;
```

Return Value

A locale-specific rule for formatting outputs with positive amounts.

Remarks

The member function returns do_pos_format.

Example

```
// moneypunct_pos_format.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>

using namespace std;

int main() {
   locale loc( "german_germany" );

   const moneypunct <char, true> &mpunct =
      use_facet <moneypunct <char, true> >(loc);
   cout << loc.name( ) << " international positive number format: "
        << mpunct.neg_format( ).field[0]
        << mpunct.pos_format( ).field[1]
        << mpunct.pos_format( ).field[2]
        << mpunct.pos_format( ).field[3] << endl;

   const moneypunct <char, false> &mpunct2 =
      use_facet <moneypunct <char, false> >(loc);
   cout << loc.name( ) << " domestic positive number format: "
        << mpunct2.neg_format( ).field[0]
        << mpunct2.pos_format( ).field[1]
        << mpunct2.pos_format( ).field[2]
        << mpunct2.pos_format( ).field[3] << endl;
}
```

Sample Output

```
German_Germany.1252 international positive number format: $+vx
German_Germany.1252 domestic positive number format: +v $
```

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct Members

moneypunct::positive_sign

Returns a locale-specific sequence of elements to use as a positive sign symbol.

```
string_type positive_sign( ) const;
```

Return Value

A locale-specific sequence of elements to use as a positive sign symbol.

Remarks

The member function returns do_positive_sign.

Example

```
// moneypunct_pos_sign.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>

using namespace std;

int main( )
{
   locale loc( "german_germany" );

   const moneypunct <char, true> &mpunct =
      use_facet <moneypunct <char, true > >(loc);
   cout << loc.name( ) << " international positive sign:"
        << mpunct.positive_sign( ) << endl;

   const moneypunct <char, false> &mpunct2 =
      use_facet <moneypunct <char, false> >(loc);
   cout << loc.name( ) << " domestic positive sign:"
        << mpunct2.positive_sign( ) << endl;

   locale loc2( "French" );

   const moneypunct <char, true> &mpunct3 =
      use_facet <moneypunct <char, true> >(loc2);
   cout << loc2.name( ) << " international positive sign:"
        << mpunct3.positive_sign( ) << endl;

   const moneypunct <char, false> &mpunct4 =
      use_facet <moneypunct <char, false> >(loc2);
   cout << loc2.name( ) << " domestic positive sign:"
        << mpunct4.positive_sign( ) << endl;
};
```

Output

```
German_Germany.1252 international positive sign:
German_Germany.1252 domestic positive sign:
French_France.1252 international positive sign:
French_France.1252 domestic positive sign:
```

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct Members

moneypunct::thousands_sep

Returns a locale-specific sequence of elements to use as a thousands separator symbol.

```
CharType thousands_sep( ) const;
```

Return Value

A locale-specific sequence of elements to use as a thousands separator

Remarks

The member function returns do_thousands_sep.

Example

```
// moneypunct_thou_sep.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "german_germany" );

   const moneypunct <char, true> &mpunct =
       use_facet <moneypunct <char, true > >(loc);
   cout << loc.name( ) << " international thousands separator: "
        << mpunct.thousands_sep( ) << endl;

   const moneypunct <char, false> &mpunct2 =
      use_facet <moneypunct <char, false> >(loc);
   cout << loc.name( ) << " domestic thousands separator: "
        << mpunct2.thousands_sep( ) << endl << endl;

   locale loc2( "english_canada" );

   const moneypunct <char, true> &mpunct3 =
       use_facet <moneypunct <char, true> >(loc2);
   cout << loc2.name( ) << " international thousands separator: "
        << mpunct3.thousands_sep( ) << endl;

   const moneypunct <char, false> &mpunct4 =
      use_facet <moneypunct <char, false> >(loc2);
   cout << loc2.name( ) << " domestic thousands separator: "
        << mpunct4.thousands_sep( ) << endl;
}
```

Output

```
German_Germany.1252 international thousands separator: .
German_Germany.1252 domestic thousands separator: .

English_Canada.1252 international thousands separator: ,
English_Canada.1252 domestic thousands separator: ,
```

Requirements

Header: <locale>

See Also

Reference

moneypunct Class

Other Resources

moneypunct_byname Class

A derived template class that describes an object that can serve as a moneypunct facet of a given locale, enabling the formatting monetary input field or monetary output fields.

```
template<class CharType, bool Intl>
   class moneypunct_byname
        : public moneypunct< CharType, Intl> {
public:
   explicit moneypunct_byname(
      const char *_Locname,
      size_t _Refs = 0);
protected:
   ~moneypunct_byname( );
   };
```

Remarks

Its behavior is determined by the named locale *_Locname*. The constructor initializes its base object with moneypunct<**CharType**, **Intl**>(*_Refs*).

Requirements

Header: <locale>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<locale> Members

num_get Class

A template class that describes an object that can serve as a locale facet to control conversions of sequences of type **CharType** to numeric values.

For a list of all members of this type, see num_get Members.

```
template<
   class CharType,
   class InputIterator = istreambuf_iterator<CharType>
>
class num_get : public locale::facet
```

Parameters

CharType

The type used within a program to encode characters in a locale.

InputIterator

The type of iterator from which the numeric get functions read their input.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id.**

Requirements

Header: <locale>

See Also

Reference

facet Class

Thread Safety in the Standard C++ Library

Other Resources

num_get Members

<locale> Members

num_get Members

The following tables list the members of the num_get Class. For a list of members inherited by the **num_get** class, see facet Class.

Reference

Constructors

num_get	The constructor for objects of type **num_get** that are used to extract numerical values from sequences.

Typedefs

char_type	A type that is used to describe a character used by a locale.
iter_type	A type that describes an input iterator.

Member Functions

do_get	A virtual function that is called to extracts a numerical or Boolean value from a character sequence.
get	Extracts a numerical or Boolean value from a character sequence.

See Also

Reference

num_get Class
facet Class
Thread Safety in the Standard C++ Library

num_get Typedefs

For information about the typedefs in the num_get class, see num_get Members.

num_get::char_type

A type that is used to describe a character used by a locale.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

Requirements

Header: <locale>

See Also

Reference

num_get Class

Other Resources

num_get Members

num_get::iter_type

A type that describes an input iterator.

```
typedef InputIterator iter_type;
```

Remarks

The type is a synonym for the template parameter **InputIterator**.

Requirements

Header: <locale>

See Also

Reference

num_get Class

Other Resources

num_get Members

num_get Member Functions

For information about the member functions in the num_get class, see num_get Members.

num_get::do_get

A virtual function that is called to extracts a numerical or Boolean value from a character sequence.

```
virtual iter_type do_get (
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   long& _Val
) const;
virtual iter_type do_get (
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   unsigned long& _Val
) const;
virtual iter_type do_get (
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   double& _Val
) const;
virtual iter_type do_get (
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   long double& _Val
) const;
virtual iter_type do_get (
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   void *& _Val
) const;
virtual iter_type do_get (
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   bool& _Val
) const;
```

Parameters

_First

The beginning of the range of characters from which to read the number.

_Last

The end of the range of characters from which to read the number.

_Iosbase

The ios_base whose flags are used by the conversion.

_State

The state to which failbit (see ios_base::iostate) is added upon failure.

_Val

The value that was read.

Return Value

The iterator after the value has been read.

Example

See the example for get, where the virtual member function is called by **do_get**.

Requirements

Header: <locale>

See Also

Reference

num_get Class

Other Resources

num_get Members

num_get::get

Extracts a numerical or Boolean value from a character sequence.

```
iter_type get(
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   long& _Val
) const;
iter_type get(
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   unsigned long& _Val
) const;
iter_type get(
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   double& _Val
) const;
iter_type get(
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   long double& _Val
) const;
iter_type get(
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   void *& _Val
) const;
iter_type get(
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   bool& _Val
) const;
```

Parameters

_First

The beginning of the range of characters from which to read the number.

_Last

The end of the range of characters from which to read the number.

_Iosbase

The ios_base whose flags are used by the conversion.

_State

The state to which failbit (see ios_base::iostate) is added upon failure.

_Val

The value that was read.

Return Value

The iterator after the value has been read.

Remarks

All member functions return do_get(*_First*, *_Last*, *_Iosbase*, *_State*, *_Val*).

The first virtual protected member function tries to match sequential elements beginning at first in the sequence [*_First*, *_Last*) until it has recognized a complete, nonempty integer input field. If successful, it converts this field to its equivalent value as type **long** and stores the result in *_Val*. It returns an iterator designating the first element beyond the numeric input field. Otherwise, the function stores nothing in *_Val* and sets **ios_base::failbit** in *_State*. It returns an iterator designating the first element beyond any prefix of a valid integer input field. In either case, if the return value equals **last**, the function sets **ios_base::eofbit** in *_State*.

The integer input field is converted by the same rules used by the scan functions for matching and converting a series of **char** elements from a file. Each such **char** element is assumed to map to an equivalent element of type **CharType** by a simple, one-to-one mapping. The equivalent scan conversion specification is determined as follows:

- If **iosbase**.flags & **ios_base::basefield** == **ios_base::**oct, the conversion specification is **lo**.
- If **iosbase.flags** & **ios_base::basefield** == **ios_base::**hex, the conversion specification is **lx**.
- If **iosbase.flags** & **ios_base::basefield** == 0, the conversion specification is **li**.
- Otherwise, the conversion specification is **ld**.

The format of an integer input field is further determined by the locale facet **fac** returned by the call use_facet <numpunct<**Elem**>(**iosbase**.getloc). Specifically:

- **fac**.grouping determines how digits are grouped to the left of any decimal point.
- **fac**.thousands_sep determines the sequence that separates groups of digits to the left of any decimal point.

If no instances of **fac**.**thousands_sep** occur in the numeric input field, no grouping constraint is imposed. Otherwise, any grouping constraints imposed by **fac**.**grouping** is enforced and separators are removed before the scan conversion occurs.

The second virtual protected member function:

```
virtual iter_type do_get(iter_type _First, iter_type _Last,
    ios_base& _Iosbase, ios_base::iostate& _State,
        unsigned long& _Val) const;
```

behaves the same as the first, except that it replaces a conversion specification of **ld** with **lu**. If successful, it converts the numeric input field to a value of type **unsigned long** and stores that value in *_Val*.

The third virtual protected member function:

```
virtual iter_type do_get(iter_type _First, iter_type _Last,
    ios_base& _Iosbase, ios_base::iostate& _State,
        double& _Val) const;
```

behaves the same as the first, except that it tries to match a complete, nonempty floating-point input field. **fac**.decimal_point determines the sequence that separates the integer digits from the fraction digits. The equivalent scan conversion specifier is **lf**.

The fourth virtual protected member function:

```
virtual iter_type do_get(iter_type _First, iter_type _Last,
    ios_base& _Iosbase, ios_base::iostate& _State,
        long double& _Val) const;
```

behaves the same the third, except that the equivalent scan conversion specifier is **Lf**.

The fifth virtual protected member function:

```
virtual iter_type do_get(iter_type _First, iter_type _Last,
    ios_base& _Iosbase, ios_base::iostate& _State,
        void *& _Val) const;
```

behaves the same the first, except that the equivalent scan conversion specifier is **p**.

The sixth virtual protected member function:

```
virtual iter_type do_get(iter_type _First, iter_type _Last,
    ios_base& _Iosbase, ios_base::iostate& _State,
        bool& _Val) const;
```

behaves the same as the first, except that it tries to match a complete, nonempty boolean input field. If successful it converts the Boolean input field to a value of type **bool** and stores that value in *_Val*.

A boolean input field takes one of two forms. If **iosbase**.**flags** & **ios_base::**boolalpha is **false**, it is the same as an integer input field, except that the converted value must be either 0 (for **false**) or 1 (for **true**). Otherwise, the sequence must match either **fac**.falsename (for **false**), or **fac**.truename (for **true**).

Example

```
// num_get_get.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "german_germany" );

   basic_stringstream<char> psz, psz2;
   psz << "-1000,56";

   ios_base::iostate st = 0;
   long double fVal;
   cout << use_facet <numpunct <char> >(loc).thousands_sep( ) << endl;


   psz.imbue( loc );
   use_facet <num_get <char> >
   (loc).get( basic_istream<char>::_Iter( psz.rdbuf( ) ),
           basic_istream<char>::_Iter(0), psz, st, fVal );

   if ( st & ios_base::failbit )
      cout << "money_get( ) FAILED" << endl;
   else
      cout << "money_get( ) = " << fVal << endl;
}
```

Output

```
.
money_get( ) = -1000.56
```

Requirements

Header: <locale>

See Also

Reference

num_get Class

Other Resources

num_get Members

num_get::num_get

The constructor for objects of type **num_get** that are used to extract numerical values from sequences.

```
explicit num_get(
   size_t _Refs = 0
);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 0: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with **locale::**facet(*_Refs*).

Requirements

Header: <locale>

See Also

Reference

num_get Class

Other Resources

num_get Members

num_put Class

A template class that describes an object that can serve as a locale facet to control conversions of numeric values to sequences of type **CharType**.

For a list of all members of this type, see num_put Members.

```
template<
   class CharType,
   class OutputIterator = ostreambuf_iterator<CharType>
>
class num_put : public locale::facet
```

Parameters

CharType

The type used within a program to encode characters in a locale.

OutputIterator

The type of iterator to which the numeric put functions write their output.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id.**

Requirements

Header: <locale>

See Also

Reference

facet Class

Thread Safety in the Standard C++ Library

Other Resources

num_put Members

<locale> Members

num_put Members

The following tables list the members of the num_put Class. For a list of members inherited by the **num_put** class, see facet Class.

Reference

Constructors

num_put	The constructor for objects of type **num_put**.

Typedefs

char_type	A type that is used to describe a character used by a locale.
iter_type	A type that describes an output iterator.

Member Functions

do_put	A virtual function that is called to convert a number into a sequence of **CharType**s that represents the number formatted for a given locale.
put	Converts a number into a sequence of **CharType**s which represents the number formatted for a given locale.

See Also

Reference

num_put Class
facet Class
Thread Safety in the Standard C++ Library

num_put Typedefs

For information about the typedefs in the num_put class, see num_put Members.

num_put::char_type

A type that is used to describe a character used by a locale.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

Requirements

Header: <locale>

See Also

Reference

num_put Class

Other Resources

num_put Members

num_put::iter_type

A type that describes an output iterator.

```
typedef InIt iter_type;
```

Remarks

The type is a synonym for the template parameter **OutputIterator.**

Requirements

Header: <locale>

See Also

Reference

num_put Class

Other Resources

num_put Members

num_put Member Functions

For information about the member functions in the num_put class, see num_put Members.

num_put::do_put

A virtual function that is called to convert a number into a sequence of **CharType**s that represents the number formatted for a given locale.

```
virtual iter_type do_put(
   iter_type _Next,
   ios_base& _Iosbase,
   CharType _Fill,
   long _Val
) const;
virtual iter_type do_put(
   iter_type _Next,
   ios_base& _Iosbase,
   CharType _Fill,
   unsigned long _Val
) const;
virtual iter_type do_put(
   iter_type _Next,
   ios_base& _Iosbase,
   CharType _Fill,
   double _Val
) const;
virtual iter_type do_put(
   iter_type _Next,
   ios_base& _Iosbase,
   CharType _Fill,
   long double _Val
) const;
virtual iter_type do_put(
   iter_type _Next,
   ios_base& _Iosbase,
   CharType _Fill,
   const void* _Val
) const;
virtual iter_type do_put(
   iter_type _Next,
   ios_base& _Iosbase,
   CharType _Fill,
   bool _Val
) const;
```

Parameters

_Next

An iterator addressing the first element of the inserted string.

_Iosbase

Specified the stream which contains locale with the numpunct facet used to punctuate the output and flags for formatting the output.

_Fill

A character that is used for spacing.

_Val

The number or Boolean type that is to be output.

Return Value

An output iterator the addresses the position one beyond the last element produced.

Remarks

The first virtual protected member function generates sequential elements beginning at *_Next* to produce an integer output field from the value of *_Val*. The function returns an iterator designating the next place to insert an element beyond the generated integer output field.

The integer output field is generated by the same rules used by the print functions for generating a series of **char** elements to a file. Each such char element is assumed to map to an equivalent element of type **CharType** by a simple, one-to-one mapping. Where a print function pads a field with either spaces or the digit 0, however, **do_put** instead uses **fill**. The equivalent print conversion specification is determined as follows:

- If **iosbase**.flags & **ios_base::basefield** == **ios_base::**oct, the conversion specification is **lo**.
- If **iosbase.flags** & **ios_base::basefield** == **ios_base::**hex, the conversion specification is **lx**.
- Otherwise, the conversion specification is **ld**.

If **iosbase**.width is nonzero, a field width of this value is prepended. The function then calls **iosbase**.**width**(0) to reset the field width to zero.

Padding occurs only if the minimum number of elements *N* required to specify the output field is less than **iosbase**.width. Such padding consists of a sequence of *N* – **width** copies of **fill**. Padding then occurs as follows:

- If **iosbase**.**flags** & **ios_base::adjustfield** == **ios_base::**left, the flag **–** is prepended. (Padding occurs after the generated text.)
- If **iosbase.flags** & **ios_base::adjustfield** == **ios_base::**internal, the flag **0** is prepended. (For a numeric output field, padding occurs where the print functions pad with 0.)
- Otherwise, no additional flag is prepended. (Padding occurs before the generated sequence.)

Finally:

- If **iosbase**.**flags** & **ios_base::**showpos is nonzero, the flag **+** is prepended to the conversion specification.
- If **iosbase**.**flags** & **ios_base::**showbase is nonzero, the flag **#** is prepended to the conversion specification.

The format of an integer output field is further determined by the locale facet **fac** returned by the call use_facet <numpunct<**Elem**>(**iosbase**.getloc). Specifically:

- **fac**.grouping determines how digits are grouped to the left of any decimal point
- **fac**.thousands_sep determines the sequence that separates groups of digits to the left of any decimal point

If no grouping constraints are imposed by **fac**.**grouping** (its first element has the value CHAR_MAX), then no instances of **fac**.**thousands_sep** are generated in the output field. Otherwise, separators are inserted after the print conversion occurs.

The second virtual protected member function:

```
virtual iter_type do_put(iter_type _Next, ios_base& _Iosbase,
    CharType _Fill, unsigned long _Val) const;
```

behaves the same as the first, except that it replaces a conversion specification of **ld** with **lu**.

The third virtual protected member function:

```
virtual iter_type do_put(iter_type _Next, ios_base& _Iosbase,
    CharType _Fill, double _Val) const;
```

behaves the same as the first, except that it produces a floating-point output field from the value of **val**.**fac**.decimal_point determines the sequence that separates the integer digits from the fraction digits. The equivalent print conversion specification is determined as follows:

- If **iosbase**.**flags** & **ios_base::floatfield** == **ios_base::**fixed, the conversion specification is **lf**.
- If **iosbase**.**flags** & **ios_base::floatfield** == **ios_base::**scientific, the conversion specification is **le**. If **iosbase**.**flags** & **ios_base::**uppercase is nonzero, **e** is replaced with **E**.
- Otherwise, the conversion specification is **lg**. If **iosbase**.**flags** & **ios_base::uppercase** is nonzero, **g** is replaced with **G**.

If **iosbase**.**flags** & **ios_base::fixed** is nonzero or if **iosbase**.precision is greater than zero, a precision with the value **iosbase**.**precision** is prepended to the conversion specification. Any padding behaves the same as for an integer output field. The padding character is **fill**. Finally:

- If **iosbase**.**flags** & **ios_base::**showpos is nonzero, the flag **+** is prepended to the conversion specification.
- If **iosbase**.**flags** & **ios_base::**showpoint is nonzero, the flag **#** is prepended to the conversion specification.

The fourth virtual protected member function:

```
virtual iter_type do_put(iter_type _Next, ios_base& _Iosbase,
    CharType _Fill, long double _Val) const;
```

behaves the same the third, except that the qualifier **l** in the conversion specification is replaced with **L**.

The fifth virtual protected member function:

```
virtual iter_type do_put(iter_type _Next, ios_base& _Iosbase,
    CharType _Fill, const void *_Val) const;
```

behaves the same the first, except that the conversion specification is `p`**,** plus any qualifier needed to specify padding.

The sixth virtual protected member function:

```
virtual iter_type do_put(iter_type _Next, ios_base& _Iosbase,
    CharType _Fill, bool _Val) const;
```

behaves the same as the first, except that it generates a Boolean output field from *_Val*.

A Boolean output field takes one of two forms. If **iosbase**.**flags** & **ios_base::**boolalpha is **false**, the member function returns **do_put**(*_Next*, *_Iosbase*, *_Fill*, (**long**)*_Val*), which typically produces a generated sequence of either 0 (for **false**) or 1 (for **true**). Otherwise, the generated sequence is either **fac**.falsename) (for **false**), or **fac**.truename (for **true**).

Example

See the example for put, which calls **do_put**.

Requirements

Header: <locale>

See Also

Reference

num_put Class

Other Resources

num_put Members

num_put::num_put

The constructor for objects of type **num_put**.

```
explicit num_put(
   size_t _Refs = 0
);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 0: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with **locale::**facet(*_Refs*).

Requirements

Header: <locale>

See Also

Reference

num_put Class

Other Resources

num_put Members

num_put::put

Converts a number into a sequence of **CharType**s that represents the number formatted for a given locale.

```
iter_type put(
   iter_type _Next,
   ios_base& _Iosbase,
   CharType _Fill,
   long _Val
) const;
iter_type put(
   iter_type _Next,
   ios_base& _Iosbase,
   CharType _Fill,
   unsigned long _Val
) const;
iter_type put(
   iter_type _Next,
   ios_base& _Iosbase,
   CharType _Fill,
   double _Val
) const;
iter_type put(
   iter_type _Next,
   ios_base& _Iosbase,
   CharType _Fill,
   long double _Val
) const;
iter_type put(
   iter_type _Next,
   ios_base& _Iosbase,
   CharType _Fill,
   const void* _Val
) const;
iter_type put(
   iter_type _Next,
   ios_base& _Iosbase,
   CharType _Fill,
   bool _Val
) const;
```

Parameters

_Next

An iterator addressing the first element of the inserted string.

_Iosbase

Specified the stream that contains locale with the numpunct facet used to punctuate the output and flags for formatting the output.

_Fill

A character that is used for spacing.

_Val

The number or Boolean type that is to be output.

Return Value

An output iterator the addresses the position one beyond the last element produced.

Remarks

All member functions return do_put(*_Next*, *_Iosbase*, *_Fill*, *_Val*).

Example

```
// num_put_put.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "german_germany" );
   basic_stringstream<char> psz2;
   ios_base::iostate st = 0;
   long double fVal;
   cout << "The thousands separator is: "
        << use_facet < numpunct <char> >(loc).thousands_sep( )
        << endl;

   psz2.imbue( loc );
   use_facet < num_put < char > >
      ( loc ).put(basic_ostream<char>::_Iter(psz2.rdbuf( ) ),
                  psz2, ' ', fVal=1000.67);

   if ( st & ios_base::failbit )
      cout << "num_put( ) FAILED" << endl;
   else
      cout << "num_put( ) = " << psz2.rdbuf( )->str( ) << endl;
}
```

Output

```
The thousands separator is: .
num_put( ) = 1.000,67
```

Requirements

Header: <locale>

See Also

Reference

num_put Class

Other Resources

num_put Members

numpunct Class

A template class that describes an object that can serve as a local facet to describe the sequences of type **CharType** used to represent information about the formatting and punctuation of numeric and Boolean expressions.

For a list of all members of this type, see numpunct Members.

```
template <class CharType>
class numpunct : public locale::facet
```

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id.**

Requirements

Header: <locale>

See Also

Reference

facet Class

Thread Safety in the Standard C++ Library

Other Resources

numpunct Members

<locale> Members

numpunct Members

The following tables list the members of the numpunct Class. For a list of members inherited by the **numpunct** class, see facet Class.

Reference

Constructors

numpunct	The constructor for objects of type **numpunct**.

Typedefs

char_type	A type that is used to describe a character used by a locale.
string_type	A type that describes a string containing characters of type **CharType**.

Member Functions

decimal_point	Returns a locale-specific element to use as a decimal point.
do_decimal_point	A protected virtual member function that is called to return a locale-specific element to use as a decimal point.
do_falsename	A protected virtual member function that is called to return a string to use as a text representation of the value **false**.
do_grouping	A protected virtual member function that is called to return a locale-specific rule for determining how digits are grouped to the left of any decimal point.
do_truename	A protected virtual member function that is called to return a string to use as a text representation of the value **true**.
do_thousands_sep	A protected virtual member function that is called to return a locale-specific element to use as a thousands separator.
falsename	Returns a string to use as a text representation of the value **false**.
grouping	Returns a locale-specific rule for determining how digits are grouped to the left of any decimal point.
thousands_sep	Returns a locale-specific element to use as a thousands separator.
truename	Returns a string to use as a text representation of the value **true**.

See Also

Reference

numpunct Class
facet Class
Thread Safety in the Standard C++ Library

numpunct Typedefs

For information about the typedefs in the numpunct class, see numpunct Members.

numpunct::char_type

A type that is used to describe a character used by a locale.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType.**

Requirements

Header: <locale>

See Also

Reference

numpunct Class

Other Resources

numpunct Members

numpunct::string_type

A type that describes a string containing characters of type **CharType**.

```
typedef basic_string<CharType> string_type;
```

Remarks

The type describes a specialization of template class basic_string whose objects can store copies of the punctuation sequences.

Requirements

Header: <locale>

See Also

Reference

numpunct Class

Other Resources

numpunct Members

numpunct Member Functions

For information about the member functions in the numpunct class, see numpunct Members.

numpunct::decimal_point

Returns a locale-specific element to use as a decimal point.

```
CharType decimal_point( ) const;
```

Return Value

A locale-specific element to use as a decimal point.

Remarks

The member function returns do_decimal_point.

Example

```
// numpunct_decimal_point.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "german_germany" );

   const numpunct <char> &npunct =
   use_facet <numpunct <char> >( loc);
   cout << loc.name( ) << " decimal point "<<
   npunct.decimal_point( ) << endl;
   cout << loc.name( ) << " thousands separator "
   << npunct.thousands_sep( ) << endl;
};
```

Output

```
German_Germany.1252 decimal point ,
German_Germany.1252 thousands separator .
```

Requirements

Header: <locale>

See Also

Reference

numpunct Class

Other Resources

numpunct Members

numpunct::do_decimal_point

A protected virtual member function that is called to return a locale-specific element to use as a decimal point.

```
CharType do_decimal_point( ) const;
```

Return Value

A locale-specific element to use as a decimal point.

Example

See the example for decimal_point, where the virtual member function is called by **decimal_point**.

Requirements

Header: <locale>

See Also

Reference

numpunct Class

Other Resources

numpunct Members

numpunct::do_falsename

The protected virtual member function returns a sequence to use as a text representation of the value **false**.

```
string_type do_falsename( ) const;
```

Return Value

A string containing a sequence to use as a text representation of the value **false**.

Remarks

The member function returns the string "false" to represent the value **false** in all locales.

Example

See the example for falsename, where the virtual member function is called by **falsename**.

Requirements

Header: <locale>

See Also

Reference

numpunct Class

Other Resources

numpunct Members

numpunct::do_grouping

A protected virtual member function that is called to return a locale-specific rule for determining how digits are grouped to the left of any decimal point.

```
string do_grouping( ) const;
```

Return Value

A locale-specific rule for determining how digits are grouped to the left of any decimal point.

Remarks

The protected virtual member function returns a locale-specific rule for determining how digits are grouped to the left of any decimal point. The encoding is the same as for **lconv::grouping**.

Example

See the example for grouping, where the virtual member function is called by **grouping**.

Requirements

Header: <locale>

See Also

Reference

numpunct Class

Other Resources

numpunct Members

numpunct::do_thousands_sep

A protected virtual member function that is called to return a locale-specific element to use as a thousands separator.

```
CharType do_thousands_sep( ) const;
```

Return Value

Returns a locale-specific element to use as a thousands separator.

Remarks

The protected virtual member function returns a locale-specific element of type **CharType** to use as a group separator to the left of any decimal point.

Example

See the example for thousands_sep, where the virtual member function is called by **thousands_sep**.

Requirements

Header: <locale>

See Also

Reference

numpunct Class

Other Resources

numpunct Members

numpunct::do_truename

A protected virtual member function that is called to return a string to use as a text representation of the value **true**.

```
string_type do_truename( ) const;
```

Remarks

A string to use as a text representation of the value **true**.

All locales return a string "true" to represent the value **true**.

Example

See the example for truename, where the virtual member function is called by **truename**.

Requirements

Header: <locale>

See Also

Reference

numpunct Class

Other Resources

numpunct Members

numpunct::falsename

Returns a string to use as a text representation of the value **false**.

```
string_type falsename( ) const;
```

Return Value

A string containing a sequence of **CharType**s to use as a text representation of the value **false**.

Remarks

The member function returns the string "false" to represent the value **false** in all locales.

The member function returns do_falsename.

Example

```
// numpunct_falsename.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "English" );

   const numpunct <char> &npunct = use_facet <numpunct <char> >( loc );
   cout << loc.name( ) << " truename "<< npunct.truename( ) << endl;
   cout << loc.name( ) << " falsename "<< npunct.falsename( ) << endl;

   locale loc2( "French" );
   const numpunct <char> &npunct2 = use_facet <numpunct <char> >(loc2);
   cout << loc2.name( ) << " truename "<< npunct2.truename( ) << endl;
   cout << loc2.name( ) << " falsename "<< npunct2.falsename( ) << endl;
}
```

Output

```
English_United States.1252 truename true
English_United States.1252 falsename false
French_France.1252 truename true
French_France.1252 falsename false
```

Requirements

Header: <locale>

See Also

Reference

numpunct Class

Other Resources

numpunct Members

numpunct::grouping

Returns a locale-specific rule for determining how digits are grouped to the left of any decimal point.

```
string grouping( ) const;
```

Return Value

A locale-specific rule for determining how digits are grouped to the left of any decimal point.

Remarks

The member function returns do_grouping.

Example

```
// numpunct_grouping.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "german_germany");

   const numpunct <char> &npunct =
       use_facet < numpunct <char> >( loc );
   for (unsigned int i = 0; i < npunct.grouping( ).length( ); i++)
   {
      cout << loc.name( ) << " international grouping:\n the "
           << i <<"th group to the left of the radix character "
           << "is of size " << (int)(npunct.grouping ( )[i])
           << endl;
   }
}
```

Output

```
German_Germany.1252 international grouping:
 the 0th group to the left of the radix character is of size 3
```

Requirements

Header: <locale>

See Also

Reference

numpunct Class

Other Resources

numpunct Members

numpunct::numpunct

The constructor for objects of type **numpunct**.

```
explicit numpunct(
   size_t _Refs = 0
);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 0: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with **locale::**facet(*_Refs*).

Requirements

Header: <locale>

See Also

Reference

numpunct Class

Other Resources

numpunct Members

numpunct::thousands_sep

Returns a locale-specific element to use as a thousands separator.

```
CharType thousands_sep( ) const;
```

Return Value

A locale-specific element to use as a thousands separator.

Remarks

The member function returns do_thousands_sep.

Example

```
// numpunct_thou_sep.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "german_germany" );

   const numpunct <char> &npunct =
   use_facet < numpunct < char > >( loc );
   cout << loc.name( ) << " decimal point "<<
   npunct.decimal_point( ) << endl;
   cout << loc.name( ) << " thousands separator "
   << npunct.thousands_sep( ) << endl;
};
```

Output

```
German_Germany.1252 decimal point ,
German_Germany.1252 thousands separator .
```

Requirements

Header: <locale>

See Also

Reference

numpunct Class

Other Resources

numpunct Members

numpunct::truename

Returns a string to use as a text representation of the value **true**.

```
string_type falsename( ) const;
```

Return Value

A string to use as a text representation of the value **true**.

Remarks

The member function returns do_truename.

All locales return a string "true" to represent the value **true**.

Example

```
// numpunct_truename.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "English" );

   const numpunct < char> &npunct = use_facet <numpunct <char> >( loc );
   cout << loc.name( ) << " truename "<< npunct.truename( ) << endl;
   cout << loc.name( ) << " falsename "<< npunct.falsename( ) << endl;

   locale loc2("French");
   const numpunct <char> &npunct2 = use_facet <numpunct <char> >( loc2 );
   cout << loc2.name( ) << " truename "<< npunct2.truename( ) << endl;
   cout << loc2.name( ) << " falsename "<< npunct2.falsename( ) << endl;
}
```

Output

```
English_United States.1252 truename true
English_United States.1252 falsename false
French_France.1252 truename true
French_France.1252 falsename false
```

Requirements

Header: <locale>

See Also

Reference

numpunct Class

Other Resources

numpunct Members

numpunct_byname Class

The derived template class describes an object that can serve as a moneypunct facet of a given locale enabling the formatting and punctuation of numeric and Boolean expressions.

```
template<Class CharType>
   class numpunct_byname : public numpunct<Elem> {
public:
   explicit numpunct_byname(
      const char* _Locname,
      size_t _Refs = 0
   );
protected:
   ~numpunct_byname( );
   };
```

Remarks

Its behavior is determined by the named locale locname. The constructor initializes its base object with numpunct<**CharType**> (*_Refs*).

Requirements

Header: <locale>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<locale> Members

time_base Class

The class serves as a base class for facets of template class time_get, defining just the enumerated type **dateorder** and several constants of this type.

```
class time_base {
public:
   enum dateorder {no_order, dmy, mdy, ymd, ydm};
   };
```

Remarks

Each constant characterizes a different way to order the components of a date. The constants are:

- **no_order** specifies no particular order.
- **dmy** specifies the order day, month, then year, as in 2 December 1979.
- **mdy** specifies the order month, day, then year, as in December 2, 1979.
- **ymd** specifies the order year, month, then day, as in 1979/12/2.
- **ydm** specifies the order year, day, then month, as in 1979: 2 Dec.

Requirements

Header: <locale>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<locale> Members

time_get Class

The template class describes an object that can serve as a locale facet to control conversions of sequences of type **CharType** to time values.

For a list of all members of this type, see time_get Members.

```
template <
   class CharType,
   class InputIterator = istreambuf_iterator<CharType>
>
class time_get : public time_base
```

Parameters

CharType

The type used within a program to encode characters.

InputIterator

The iterator from which the time values are read.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id.**

Requirements

Header: <locale>

See Also

Reference

time_base Class

Thread Safety in the Standard C++ Library

Other Resources

time_get Members

<locale> Members

time_get Members

The following tables list the members of the time_get Class. For a list of members inherited by the **time_get** class, see time_base Class.

Reference

Constructors

time_get	The constructor for objects of type **time_get**.

Typedefs

char_type	A type that is used to describe a character used by a locale.
iter_type	A type that describes an input iterator.

Member Functions

date_order	Returns the date order used by a facet.
do_date_order	A protected virtual member function that is called to return the date order used by a facet.
do_get_date	A protected virtual member function that is called to parse a string as the date produced by the *x* specifier for **strftime**.
do_get_monthname	A protected virtual member function that is called to parse a string as the name of the month.
do_get_time	A protected virtual member function that is called to parse a string as the date produced by the *X* specifier for **strftime**.
do_get_weekday	A protected virtual member function that is called to parse a string as the name of the day of the week.
do_get_year	A protected virtual member function that is called to parses a string as the name of the year.
get_date	Parses a string as the date produced by the *x* specifier for **strftime**.
get_monthname	Parses a string as the name of the month.
get_time	Parses a string as the date produced by the *X* specifier for **strftime**.
get_weekday	Parses a string as the name of the day of the week.
get_year	Parses a string as the name of the year.

See Also

Reference

time_get Class
time_base Class
Thread Safety in the Standard C++ Library

time_get Typedefs

For information about the typedefs in the time_get class, see time_get Members.

time_get::char_type

A type that is used to describe a character used by a locale.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

Requirements

Header: <locale>

See Also

Reference

time_get Class

Other Resources

time_get Members

time_get::iter_type

A type that describes an input iterator.

```
typedef InputIterator iter_type;
```

Remarks

The type is a synonym for the template parameter **InputIterator**.

Requirements

Header: <locale>

See Also

Reference

time_get Class

Other Resources

time_get Members

time_get Member Functions

For information about the member functions in the time_get class, see time_get Members.

time_get::date_order

Returns the date order used by a facet.

```
dateorder date_order( ) const;
```

Return Value

The date order used by a facet.

Remarks

The member function returns do_date_order.

Example

```
// time_get_date_order.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
void po( char *p )
{
   locale loc( p );

   time_get <char>::dateorder order = use_facet <time_get <char> >( loc ).date_order ( );
   cout << loc.name( );
   switch (order){
      case time_base::dmy: cout << "(day, month, year)" << endl;
      break;
      case time_base::mdy: cout << "(month, day, year)" << endl;
      break;
      case time_base::ydm: cout << "(year, day, month)" << endl;
      break;
      case time_base::ymd: cout << "(year, month, day)"<< endl;
      break;
      case time_base::no_order: cout << "(no_order)"<< endl;
      break;
   }
}

int main( )
{
   po( "C" );
   po( "german" );
   po( "English_Britain" );
}
```

Output

```
C(month, day, year)
German_Germany.1252(day, month, year)
English_United Kingdom.1252(day, month, year)
```

Requirements

Header: <locale>

See Also

Reference

time_get Class

Other Resources

time_get Members

time_get::do_date_order

A protected virtual member function that is called to return the date order used by a facet.

```
virtual dateorder do_date_order( ) const;
```

Return Value

The date order used by a facet.

Remarks

The virtual protected member function returns a value of type **time_base::dateorder**, which describes the order in which date components are matched by do_get_date. In this implementation, the value is **time_base::mdy**, corresponding to dates of the form December 2, 1979.

Example

See the example for date_order, which calls **do_date_order**.

Requirements

Header: <locale>

See Also

Reference

time_get Class

Other Resources

time_get Members

time_get::do_get_date

A protected virtual member function that is called to parse a string as the date produced by the *x* specifier for **strftime**.

```
virtual iter_type do_get_date(
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   tm* _Pt
) const;
```

Parameters

_First

Input iterator addressing the beginning of the sequence to be converted.

_Last

Input iterator addressing the end of the sequence to be converted.

_Iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

_State

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

_Pt

A pointer to where the date information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The virtual protected member function tries to match sequential elements beginning at first in the sequence [*_First*, *_Last*) until it has recognized a complete, nonempty date input field. If successful, it converts this field to its equivalent value as the components **tm::tm_mon**, **tm::tm_day**, and **tm::tm_year**, and stores the results in *_Pt*->**tm_mon**, *_Pt*->**tm_day** and *_Pt*->**tm_year**, respectively. It returns an iterator designating the first element beyond the date input field. Otherwise, the function sets *_Iosbase***::failbit** in *_State*. It returns an iterator designating the first element beyond any prefix of a valid date input field. In either case, if the return value equals *_Last*, the function sets **ios_base::eofbit** in *_State*.

The format for the date input field is locale dependent. For the default locale, the date input field has the form MMM DD, YYYY, where:

- MMM is matched by calling get_monthname, giving the month.
- DD is a sequence of decimal digits whose corresponding numeric value must be in the range [1, 31], giving the day of the month.
- YYYY is matched by calling get_year, giving the year.

The literal spaces and commas must match corresponding elements in the input sequence.

Example

See the example for get_date, which calls **do_get_date**.

Requirements

Header: <locale>

See Also

Reference

time_get Class

Other Resources

time_get Members

time_get::do_get_monthname

A protected virtual member function that is called to parse a string as the name of the month.

```
virtual iter_type do_get_monthname(
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   tm* _Pt
) const;
```

Parameters

_First

Input iterator addressing the beginning of the sequence to be converted.

_Last

Input iterator addressing the end of the sequence to be converted.

_Iosbase

Unused.

_State

An output parameter that sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

_Pt

A pointer to where the month information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The virtual protected member function tries to match sequential elements beginning at first in the sequence [*_First*, *_Last*) until it has recognized a complete, nonempty month input field. If successful, it converts this field to its equivalent value as the component **tm::tm_mon**, and stores the result in *_Pt*->**tm_mon**. It returns an iterator designating the first element beyond the month input field. Otherwise, the function sets **ios_base::failbit** in *_State*. It returns an iterator designating the first element beyond any prefix of a valid month input field. In either case, if the return value equals *_Last*, the function sets **ios_base::eofbit** in *_State*.

The month input field is a sequence that matches the longest of a set of locale-specific sequences, such as Jan, January, Feb, February, and so on. The converted value is the number of months since January.

Example

See the example for get_monthname, which calls **do_get_monthname**.

Requirements

Header: <locale>

See Also

Reference

time_get Class

Other Resources

time_get Members

time_get::do_get_time

A protected virtual member function that is called to parse a string as the date produced by the *X* specifier for **strftime**.

```
virtual iter_type do_get_time(
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   tm* _Pt
) const;
```

Parameters

_First

Input iterator addressing the beginning of the sequence to be converted.

_Last

Input iterator addressing the end of the sequence to be converted.

_Iosbase

Unused.

_State

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

_Pt

A pointer to where the date information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The virtual protected member function tries to match sequential elements beginning at first in the sequence [*_First*, *_Last*) until it has recognized a complete, nonempty time input field. If successful, it converts this field to its equivalent value as the components **tm::tm_hour**, **tm::tm_min**, and **tm::tm_sec**, and stores the results in *_Pt*->**tm_hour**, *_Pt*->**tm_min**, and *_Pt*->**tm_sec**, respectively. It returns an iterator designating the first element beyond the time input field. Otherwise, the function sets **ios_base::failbit** in *_State*. It returns an iterator designating the first element beyond any prefix of a valid time input field. In either case, if the return value equals *_Last*, the function sets **ios_base::eofbit** in *_State*.

In this implementation, the time input field has the form HH:MM:SS, where:

- HH is a sequence of decimal digits whose corresponding numeric value must be in the range [0, 24), giving the hour of the day.
- MM is a sequence of decimal digits whose corresponding numeric value must be in the range [0, 60), giving the minutes past the hour.
- SS is a sequence of decimal digits whose corresponding numeric value must be in the range [0, 60), giving the seconds past the minute.

The literal colons must match corresponding elements in the input sequence.

Example

See the example for get_time, which calls **do_get_time**.

Requirements

Header: <locale>

See Also

Reference

time_get Class

Other Resources

time_get Members

time_get::do_get_weekday

A protected virtual member function that is called to parse a string as the name of the day of the week.

```
virtual iter_type do_get_weekday(
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   tm* _Pt
) const;
```

Parameters

_First

Input iterator addressing the beginning of the sequence to be converted.

_Last

Input iterator addressing the end of the sequence to be converted.

_Iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

_State

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

_Pt

A pointer to where the weekday information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The virtual protected member function tries to match sequential elements beginning at *_First* in the sequence [*_First*, *_Last*) until it has recognized a complete, nonempty weekday input field. If successful, it converts this field to its equivalent value as the component **tm::tm_wday**, and stores the result in *_Pt*->**tm_wday**. It returns an iterator designating the first element beyond the weekday input field. Otherwise, the function sets **ios_base::failbit** in *_State*. It returns an iterator designating the first element beyond any prefix of a valid weekday input field. In either case, if the return value equals *_Last*, the function sets **ios_base::eofbit** in *_State*.

The weekday input field is a sequence that matches the longest of a set of locale-specific sequences, such as Sun, Sunday, Mon, Monday, and so on. The converted value is the number of days since Sunday.

Example

See the example for get_weekday, which calls **do_get_weekday**.

Requirements

Header: <locale>

See Also

Reference

time_get Class

Other Resources

time_get Members

time_get::do_get_year

A protected virtual member function that is called to parses a string as the name of the year.

```
virtual iter_type do_get_year(
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   tm* _Pt
) const;
```

Parameters

_First

Input iterator addressing the beginning of the sequence to be converted.

_Last

Input iterator addressing the end of the sequence to be converted.

_Iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

_State

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

_Pt

A pointer to where the year information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The virtual protected member function tries to match sequential elements beginning at *_First* in the sequence [*_First*, *_Last*) until it has recognized a complete, nonempty year input field. If successful, it converts this field to its equivalent value as the component **tm::tm_year**, and stores the result in *_Pt*->**tm_year**. It returns an iterator designating the first element beyond the year input field. Otherwise, the function sets **ios_base::failbit** in *_State*. It returns an iterator designating the first element beyond any prefix of a valid year input field. In either case, if the return value equals *_Last*, the function sets **ios_base::eofbit** in *_State*.

The year input field is a sequence of decimal digits whose corresponding numeric value must be in the range [1900, 2036). The stored value is this value minus 1900. In this implementation, values in the range [69, 136) represent the range of years [1969, 2036). Values in the range [0, 69) are also permissible, but may represent either the range of years [1900, 1969) or [2000, 2069), depending on the specific translation environment.

Example

See the example for get_year, which calls **do_get_year**.

Requirements

Header: <locale>

See Also

Reference

time_get Class

Other Resources

time_get Members

time_get::get_date

Parses a string as the date produced by the *x* specifier for **strftime**.

```
iter_type get_date(
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   tm* _Pt
) const;
```

Parameters

_First

Input iterator addressing the beginning of the sequence to be converted.

_Last

Input iterator addressing the end of the sequence to be converted.

_Iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

_State

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

_Pt

A pointer to where the date information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The member function returns do_get_date(*_First*, *_Last*, *_Iosbase*, *_State*, *_Pt*).

Note that months are counted from 0 to 11.

Example

```
// time_get_get_date.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
int main( )
{
   locale loc;
   basic_stringstream< char > pszGetF, pszPutF, pszGetI, pszPutI;
   ios_base::iostate st = 0;
   struct tm t;
   memset(&t, 0, sizeof(struct tm));

   pszGetF << "July 4, 2000";
   pszGetF.imbue( loc );
   basic_istream<char>::_Iter i = use_facet <time_get<char> >
   (loc).get_date(basic_istream<char>::_Iter(pszGetF.rdbuf( ) ),
            basic_istream<char>::_Iter(0), pszGetF, st, &t);

   if ( st & ios_base::failbit )
      cout << "time_get("<< pszGetF.rdbuf( )->str( )<< ") FAILED on char: " << *i << endl;
```

```
    else

       cout << "time_get("<< pszGetF.rdbuf( )->str( )<< ") ="
       << "\ntm_sec: " << t.tm_sec
       << "\ntm_min: " << t.tm_min
       << "\ntm_hour: " << t.tm_hour
       << "\ntm_mday: " << t.tm_mday
       << "\ntm_mon: " << t.tm_mon
       << "\ntm_year: " << t.tm_year
       << "\ntm_wday: " << t.tm_wday
       << "\ntm_yday: " << t.tm_yday
       << "\ntm_isdst: " << t.tm_isdst
       << endl;
  }
```

Output

```
time_get(July 4, 2000) =
tm_sec: 0
tm_min: 0
tm_hour: 0
tm_mday: 4
tm_mon: 6
tm_year: 100
tm_wday: 0
tm_yday: 0
tm_isdst: 0
```

Requirements

Header: <locale>

See Also

Reference

time_get Class

Other Resources

time_get Members

time_get::get_monthname

Parses a string as the name of the month.

```
iter_type get_monthname(
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   tm* _Pt
) const;
```

Parameters

_First

Input iterator addressing the beginning of the sequence to be converted.

_Last

Input iterator addressing the end of the sequence to be converted.

_Iosbase

Unused.

_State

An output parameter that sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

_Pt

A pointer to where the month information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The member function returns do_get_monthname(*_First*, *_Last*, *_Iosbase*, *_State*, *_Pt*).

Example

```
// time_get_get_monthname.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
int main( )
{
   locale loc ( "French" );
   basic_stringstream<char> pszGetF, pszPutF, pszGetI, pszPutI;
   ios_base::iostate st = 0;
   struct tm t;
   memset( &t, 0, sizeof( struct tm ) );

   pszGetF << "juillet";
   pszGetF.imbue( loc );
   basic_istream<char>::_Iter i = use_facet <time_get <char> >
   (loc).get_monthname(basic_istream<char>::_Iter(pszGetF.rdbuf( )),
              basic_istream<char>::_Iter(0), pszGetF, st, &t);

   if (st & ios_base::failbit)
      cout << "time_get("<< pszGetF.rdbuf( )->str( )<< ") FAILED on char: " << *i << endl;
```

```
    else

       cout << "time_get("<< pszGetF.rdbuf( )->str( )<< ") ="
       << "\ntm_sec: " << t.tm_sec
       << "\ntm_min: " << t.tm_min
       << "\ntm_hour: " << t.tm_hour
       << "\ntm_mday: " << t.tm_mday
       << "\ntm_mon: " << t.tm_mon
       << "\ntm_year: " << t.tm_year
       << "\ntm_wday: " << t.tm_wday
       << "\ntm_yday: " << t.tm_yday
       << "\ntm_isdst: " << t.tm_isdst
       << endl;
  }
```

Output

```
time_get(juillet) =
tm_sec: 0
tm_min: 0
tm_hour: 0
tm_mday: 0
tm_mon: 6
tm_year: 0
tm_wday: 0
tm_yday: 0
tm_isdst: 0
```

Requirements

Header: <locale>

See Also

Reference

time_get Class

Other Resources

time_get Members

time_get::get_time

Parses a string as the date produced by the *X* specifier for **strftime**.

```
iter_type get_time(
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   tm* _Pt
) const;
```

Parameters

_First

Input iterator addressing the beginning of the sequence to be converted.

_Last

Input iterator addressing the end of the sequence to be converted.

_Iosbase

Unused.

_State

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

_Pt

A pointer to where the date information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The member function returns do_get_time(*_First*, *_Last*, *_Iosbase*, *_State*, *_Pt*).

Example

```
// time_get_get_time.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
int main( )
{
   locale loc;
   basic_stringstream<char> pszGetF, pszPutF, pszGetI, pszPutI;
   ios_base::iostate st = 0;
   struct tm t;
   memset( &t, 0, sizeof( struct tm ) );

   pszGetF << "11:13:20";
   pszGetF.imbue( loc );
   basic_istream<char>::_Iter i = use_facet
      <time_get <char> >
      (loc).get_time(basic_istream<char>::_Iter(pszGetF.rdbuf( )),
               basic_istream<char>::_Iter(0), pszGetF, st, &t);

   if (st & ios_base::failbit)
      cout << "time_get::get_time("<< pszGetF.rdbuf( )->str( )<< ") FAILED on char: " << *i
<< endl;
```

```
    else

       cout << "time_get::get_time("<< pszGetF.rdbuf( )->str( )<< ") ="
       << "\ntm_sec: " << t.tm_sec
       << "\ntm_min: " << t.tm_min
       << "\ntm_hour: " << t.tm_hour
       << endl;
  }
```

Output

```
time_get::get_time(11:13:20) =
tm_sec: 20
tm_min: 13
tm_hour: 11
```

Requirements

Header: <locale>

See Also

Reference

time_get Class

Other Resources

time_get Members

time_get::get_weekday

Parses a string as the name of the day of the week.

```
iter_type get_weekday(
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   tm* _Pt
) const;
```

Parameters

_First

Input iterator addressing the beginning of the sequence to be converted.

_Last

Input iterator addressing the end of the sequence to be converted.

_Iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

_State

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

_Pt

A pointer to where the weekday information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The member function returns do_get_weekday(*_First*, *_Last*, *_Iosbase*, *_State*, *_Pt*).

Example

```
// time_get_get_weekday.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
int main( )
{
   locale loc ( "French" );
   basic_stringstream< char > pszGetF, pszPutF, pszGetI, pszPutI;
   ios_base::iostate st = 0;
   struct tm t;
   memset( &t, 0, sizeof( struct tm ) );

   pszGetF << "mercredi";
   pszGetF.imbue(loc);
   basic_istream<char>::_Iter i = use_facet
      <time_get<char> >
      (loc).get_weekday(basic_istream<char>::_Iter(pszGetF.rdbuf( )),
               basic_istream<char>::_Iter(0), pszGetF, st, &t);

   if (st & ios_base::failbit)
      cout << "time_get::get_time("<< pszGetF.rdbuf( )->str( )<< ") FAILED on char: " << *i
<< endl;
```

```
    else

       cout << "time_get::get_time("<< pszGetF.rdbuf( )->str( )<< ") ="
       << "\ntm_wday: " << t.tm_wday
       << endl;
  }
```

Output

```
time_get::get_time(mercredi) =
tm_wday: 3
```

Requirements

Header: <locale>

See Also

Reference

time_get Class

Other Resources

time_get Members

time_get::get_year

Parses a string as the name of the year.

```
iter_type get_year(
   iter_type _First,
   iter_type _Last,
   ios_base& _Iosbase,
   ios_base::iostate& _State,
   tm* _Pt
) const;
```

Parameters

_First

Input iterator addressing the beginning of the sequence to be converted.

_Last

Input iterator addressing the end of the sequence to be converted.

_Iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

_State

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

_Pt

A pointer to where the year information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The member function returns do_get_year(*_First*, *_Last*, *_Iosbase*, *_State*, *_Pt*).

Example

```
// time_get_get_year.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
int main( )
{
   locale loc;
   basic_stringstream<char> pszGetF, pszPutF, pszGetI, pszPutI;
   ios_base::iostate st = 0;
   struct tm t;
   memset( &t, 0, sizeof( struct tm ) );

   pszGetF << "1928";

   pszGetF.imbue( loc );
   basic_istream<char>::_Iter i = use_facet
      <time_get<char> >
      (loc).get_year(basic_istream<char>::_Iter(pszGetF.rdbuf( )),
               basic_istream<char>::_Iter(0), pszGetF, st, &t);

   if (st & ios_base::failbit)
      cout << "time_get::get_year("<< pszGetF.rdbuf( )->str( )<< ") FAILED on char: " << *i
```

```
  << endl;
     else

        cout << "time_get::get_year("<< pszGetF.rdbuf( )->str( )<< ") ="
        << "\ntm_year: " << t.tm_year
        << endl;
  }
```

Output

```
time_get::get_year(1928) =
tm_year: 28
```

Requirements

Header: <locale>

See Also

Reference

time_get Class

Other Resources

time_get Members

time_get::time_get

The constructor for objects of type **time_get**.

```
explicit time_get(
   size_t _Refs = 0
);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 0: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with **locale::**facet(*_Refs*).

Requirements

Header: <locale>

See Also

Reference

time_get Class

Other Resources

time_get Members

time_get_byname Class

The derived template class describes an object that can serve as a locale facet of type **time_get**<**CharType**, **InputIterator**>.

```
template<class Elem, class InIt>
   class time_get_byname : public time_get<Elem, InIt> {
public:
   explicit time_get_byname(
      const char* _Locname,
      size_t _Refs = 0
   );
protected:
    ~time_get_byname( );
    };
```

Requirements

Its behavior is determined by the named locale *_Locname*. The constructor initializes its base object with time_get<**CharType**, **InputIterator**>(*_Refs*).

Requirements

Header: <locale>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<locale> Members

time_put Class

The template class describes an object that can serve as a locale facet to control conversions of time values to sequences of type **CharType**.

For a list of all members of this type, see time_put Members.

```
template <
   class CharType,
   class OutputIterator = ostreambuf_iterator<CharType>
>
class time_put : public locale::facet
```

Parameters

CharType

The type used within a program to encode characters.

OutputIterator

The type of iterator into which the time put functions write their output.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id.**

Requirements

Header: <locale>

See Also

Reference

time_base Class

Thread Safety in the Standard C++ Library

Other Resources

time_put Members

<locale> Members

time_put Members

The following tables list the members of the time_put Class. For a list of members inherited by the **time_put** class, see time_base Class.

Reference

Constructors

time_put	The constructor for objects of type **time_put**.

Typedefs

char_type	A type that is used to describe a character used by a locale.
iter_type	A type that describes an output iterator.

Member Functions

do_put	A virtual function that outputs time and date information as a sequence of **CharType**s.
put	Outputs time and date information as a sequence of **CharType**s.

See Also

Reference

time_put Class
time_base Class
Thread Safety in the Standard C++ Library

time_put Typedefs

For information about the typedefs in the time_put class, see time_put Members.

time_put::char_type

A type that is used to describe a character used by a locale.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

Requirements

Header: <locale>

See Also

Reference

time_put Class

Other Resources

time_put Members

time_put::iter_type

A type that describes an output iterator.

```
typedef OutputIterator iter_type;
```

Remarks

The type is a synonym for the template parameter **OutputIterator**.

Requirements

Header: <locale>

See Also

Reference

time_put Class

Other Resources

time_put Members

time_put Member Functions

For information about the member functions in the time_put class, see time_put Members.

time_put::do_put

A virtual function that outputs time and date information as a sequence of **CharType**s.

```
virtual iter_type do_put(
   iter_type _Next,
   ios_base& _Iosbase,
   char_type _Fill,
   const tm* _Pt,
   char _Fmt,
   char _Mod = 0
) const;
```

Parameters

_Next

An output iterator where the sequence of characters representing time and date are to be inserted.

_Iosbase

Unused.

_Fill

The character of type **CharType** used for spacing.

_Pt

The time and date information being output.

_Fmt

The format of the output. See strftime, wcsftime, _strftime_l, _wcsftime_l for valid values.

_Mod

A modifier for the format. See strftime, wcsftime, _strftime_l, _wcsftime_l for valid values.

Return Value

An iterator to the first position after the last element inserted.

Remarks

The virtual protected member function generates sequential elements beginning at *_Next* from time values stored in the object **_Pt*, of type **tm**. The function returns an iterator designating the next place to insert an element beyond the generated output.

The output is generated by the same rules used by **strftime**, with a last argument of *_Pt*, for generating a series of **char** elements into an array. Each such **char** element is assumed to map to an equivalent element of type **CharType** by a simple, one-to-one mapping. If *_Mod* equals zero, the effective format is "%F", where F is replaced by *_Fmt*. Otherwise, the effective format is "%MF", where M is replaced by *_Mod*.

The parameter *_Fill* is not used.

Example

See the example for put, which calls **do_put**.

Requirements

Header: <locale>

See Also

Reference

time_put Class

Other Resources

time_put Members

time_put::put

Outputs time and date information as a sequence of **CharType**s.

```
iter_type put(
   iter_type _Next,
   ios_base& _Iosbase,
   char_type _Fill,
   const tm* _Pt,
   char _Fmt,
   char _Mod = 0
) const;
iter_type put(
   iter_type _Next,
   ios_base& _Iosbase,
   char_type _Fill,
   const tm* _Pt,
   const CharType* _First,
   const CharType* _Last
) const;
```

Parameters

_Next

An output iterator where the sequence of characters representing time and date are to be inserted.

_Iosbase

Unused.

_Fill

The character of type **CharType** used for spacing.

_Pt

The time and date information being output.

_Fmt

The format of the output. See strftime, wcsftime, _strftime_l, _wcsftime_l for valid values.

_Mod

A modifier for the format. See strftime, wcsftime, _strftime_l, _wcsftime_l for valid values.

_First

The beginning of the formatting string for the output. See strftime, wcsftime, _strftime_l, _wcsftime_l for valid values.

_Last

The end of the formatting string for the output. See strftime, wcsftime, _strftime_l, _wcsftime_l for valid values.

Return Value

An iterator to the first position after the last element inserted.

Remarks

The first member function returns do_put(*_Next*, *_Iosbase*, *_Fill*, *_Pt*, *_Fmt*, *_Mod*). The second member function copies to **_Next* ++ any element in the interval [*_First*, *_Last*) other than a percent (%). For a percent followed by a character *C* in the interval [*_First*, *_Last*), the function instead evaluates *_Next* = **do_put**(*_Next*, *_Iosbase*, *_Fill*, *_Pt*, *C*, 0) and skips past *C*. If, however, *C* is a qualifier character from the set EOQ#, followed by a character *C2* in the interval [*_First*, *_Last*), the function instead evaluates *_Next* = **do_put**(*_Next*, *_Iosbase*, *_Fill*, *_Pt*, *C2*, *C*) and skips past *C2*.

Example

```
// time_put_put.cpp
```

```
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
int main( )
{
   locale loc;
   basic_stringstream<char> pszPutI;
   ios_base::iostate st = 0;
   struct tm t;
   memset( &t, 0, sizeof( struct tm ) );

   t.tm_hour = 5;
   t.tm_min = 30;
   t.tm_sec = 40;
   t.tm_year = 00;
   t.tm_mday = 4;
   t.tm_mon = 6;

   pszPutI.imbue( loc );
   char *pattern = "x: %X %x";
   use_facet <time_put <char> >
   (loc).put(basic_ostream<char>::_Iter(pszPutI.rdbuf( )),
          pszPutI, ' ', &t, pattern, pattern+strlen(pattern));

      cout << "num_put( ) = " << pszPutI.rdbuf( )->str( ) << endl;

      char strftimebuf[255];
      strftime(&strftimebuf[0], 255, pattern, &t);
      cout << "strftime( ) = " << &strftimebuf[0] << endl;
}
```

Output

```
num_put( ) = x: 05:30:40 07/04/00
strftime( ) = x: 05:30:40 07/04/00
```

Requirements

Header: <locale>

See Also

Reference

time_put Class

Other Resources

time_put Members

time_put::time_put

Constructor for objects of type **time_put**.

```
explicit time_put(
   size_t _Refs = 0
);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 0: These values are not defined.

The constructor initializes its base object with **locale::**facet(*_Refs*).

Requirements

Header: <locale>

See Also

Reference

time_put Class

Other Resources

time_put Members

time_put_byname Class

The derived template class describes an object that can serve as a locale facet of type **time_put**< **CharType**, **OutputIterator** >.

```
template<class CharType, class OutputIterator>
    class time_put_byname : public time_put<CharType, OutputIterator> {
public:
   explicit time_put_byname(
      const char* _Locname,
      size_t _Refs = 0
   );
protected:
    ~time_put_byname( );
    };
```

Remarks

Its behavior is determined by the named locale *_Locname*. The constructor initializes its base object with time_put<**CharType**, **OutputIterator**>(*_Refs*).

Requirements

Header: <locale>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<locale> Members

<map>

Defines the container template classes map and multimap and their supporting templates.

For a list of all members of this header, see <map> Members.

```
#include <map>
```

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<map> Members

Standard C++ Library Header Files

<map> Members

Reference

Operators

Map version	Multimap version	Description
operator!= (map)	operator!= (multimap)	Tests if the map or multimap object on the left side of the operator is not equal to the map or multimap object on the right side.
operator< (map)	operator< (multimap)	Tests if the map or multimap object on the left side of the operator is less than the map or multimap object on the right side.
operator<= (map)	operator<= (multimap)	Tests if the map or multimap object on the left side of the operator is less than or equal to the map or multimap object on the right side.
operator== (map)	operator== (multimap)	Tests if the map or multimap object on the left side of the operator is equal to the map or multimap object on the right side.
operator> (map)	operator> (multimap)	Tests if the map or multimap object on the left side of the operator is greater than the map or multimap object on the right side.
operator>= (map)	operator>= (multimap)	Tests if the map or multimap object on the left side of the operator is greater than or equal to the map or multimap object on the right side.

Specialized Template Functions

Map version	Multimap version	Description
swap (map)	swap (multimap)	Exchanges the elements of two maps or multimaps.

Classes

value_compare Class	Provides a function object that can compare the elements of a map by comparing the values of their keys to determine their relative order in the map.
map Class	Used for the storage and retrieval of data from a collection in which the each of the elements has a unique key with which the data is automatically ordered.
multimap Class	Used for the storage and retrieval of data from a collection in which the each of the elements has a key with which the data is automatically ordered and the keys do not need to have unique values.

See Also

Reference

<map>
Thread Safety in the Standard C++ Library
Standard Template Library

<map> Operators

For more information about the operators in <map>, see <map> Members.

See Also

Reference

Standard Template Library

operator!= (map)

Tests if the map object on the left side of the operator is not equal to the map object on the right side.

```
bool operator!=(
   const map <Key, Type, Traits, Allocator>& _Left,
   const map <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **map**.

_Right

An object of type **map**.

Return Value

true if the maps are not equal; **false** if maps are equal.

Remarks

The comparison between map objects is based on a pairwise comparison of their elements. Two maps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// map_op_ne.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1, m2, m3;
   int i;
   typedef pair <int, int> Int_Pair;

   for ( i = 0 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i ) );
   }

   if ( m1 != m2 )
      cout << "The maps m1 and m2 are not equal." << endl;
   else
      cout << "The maps m1 and m2 are equal." << endl;

   if ( m1 != m3 )
      cout << "The maps m1 and m3 are not equal." << endl;
   else
      cout << "The maps m1 and m3 are equal." << endl;
}
```

Output

```
The maps m1 and m2 are not equal.
The maps m1 and m3 are equal.
```

Requirements

Header: <map>

See Also

Reference

Standard Template Library

Other Resources

<map> Members

operator< (map)

Tests if the map object on the left side of the operator is less than the map object on the right side.

```
bool operator<(
   const map <Key, Type, Traits, Allocator>& _Left,
   const map <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **map**.

_Right

An object of type **map**.

Return Value

true if the map on the left side of the operator is strictly less than the map on the right side of the operator; otherwise **false**.

Remarks

The comparison between map objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// map_op_lt.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map<int, int> m1, m2, m3;
   int i;
   typedef pair<int, int> Int_Pair;

   for ( i = 1 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i - 1 ) );
   }

   if ( m1 < m2 )
      cout << "The map m1 is less than the map m2." << endl;
   else
      cout << "The map m1 is not less than the map m2." << endl;

   if ( m1 < m3 )
      cout << "The map m1 is less than the map m3." << endl;
   else
      cout << "The map m1 is not less than the map m3." << endl;
}
```

Output

```
The map m1 is less than the map m2.
The map m1 is not less than the map m3.
```

Requirements

Header: <map>

See Also

Reference

Standard Template Library

Other Resources

<map> Members

operator<= (map)

Tests if the map object on the left side of the operator is less than or equal to the map object on the right side.

```
bool operator<=(
   const map <Key, Type, Traits, Allocator>& _Left,
   const map <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **map**.

_Right

An object of type **map**.

Return Value

true if the map on the left side of the operator is less than or equal to the map on the right side of the operator; otherwise **false**.

Remark

The comparison between map objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// map_op_le.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1, m2, m3, m4;
   int i;
   typedef pair <int, int> Int_Pair;

   for ( i = 1 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i - 1 ) );
      m4.insert ( Int_Pair ( i, i ) );
   }

   if ( m1 <= m2 )
      cout << "The map m1 is less than or equal to the map m2." << endl;
   else
      cout << "The map m1 is greater than the map m2." << endl;

   if ( m1 <= m3 )
      cout << "The map m1 is less than or equal to the map m3." << endl;
   else
      cout << "The map m1 is greater than the map m3." << endl;

   if ( m1 <= m4 )
      cout << "The map m1 is less than or equal to the map m4." << endl;
   else
      cout << "The map m1 is greater than the map m4." << endl;
}
```

Output

```
The map m1 is less than or equal to the map m2.
The map m1 is greater than the map m3.
The map m1 is less than or equal to the map m4.
```

Requirements

Header: <map>

See Also

Reference

Standard Template Library

Other Resources

<map> Members

operator== (map)

Tests if the map object on the left side of the operator is equal to the map object on the right side.

```
bool operator==(
   const map <Key, Type, Traits, Allocator>& _Left,
   const map <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **map**.

_Right

An object of type **map**.

Return Value

true if the map on the left side of the operator is equal to the map on the right side of the operator; otherwise **false**.

Remarks

The comparison between map objects is based on a pairwise comparison of their elements. Two maps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// map_op_eq.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map < int, int > m1, m2, m3;
   int i;
   typedef pair < int, int > Int_Pair;

   for ( i = 0 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i ) );
   }

   if ( m1 == m2 )
      cout << "The maps m1 and m2 are equal." << endl;
   else
      cout << "The maps m1 and m2 are not equal." << endl;

   if ( m1 == m3 )
      cout << "The maps m1 and m3 are equal." << endl;
   else
      cout << "The maps m1 and m3 are not equal." << endl;
}
```

Output

```
The maps m1 and m2 are not equal.
The maps m1 and m3 are equal.
```

Requirements

Header: <map>

See Also

Reference

Standard Template Library

Other Resources

<map> Members

operator> (map)

Tests if the map object on the left side of the operator is greater than the map object on the right side.

```
bool operator>(
   const map <Key, Type, Traits, Allocator>& _Left,
   const map <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **map**.

_Right

An object of type **map**.

Return Value

true if the map on the left side of the operator is greater than the map on the right side of the operator; otherwise **false**.

Remarks

The comparison between map objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// map_op_gt.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map < int, int > m1, m2, m3;
   int i;
   typedef pair < int, int > Int_Pair;

   for ( i = 0 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i - 1 ) );
   }

   if ( m1 > m2 )
      cout << "The map m1 is greater than the map m2." << endl;
   else
      cout << "The map m1 is not greater than the map m2." << endl;

   if ( m1 > m3 )
      cout << "The map m1 is greater than the map m3." << endl;
   else
      cout << "The map m1 is not greater than the map m3." << endl;
}
```

Output

```
The map m1 is not greater than the map m2.
The map m1 is greater than the map m3.
```

Requirements

Header: <map>

See Also

Reference

Standard Template Library

Other Resources

<map> Members

operator>= (map)

Tests if the map object on the left side of the operator is greater than or equal to the map object on the right side.

```
bool operator>=(
   const map <Key, Type, Traits, Allocator>& _Left,
   const map <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **map**.

_Right

An object of type **map**.

Return Value

true if the map on the left side of the operator is greater than or equal to the map on the right side of the list; otherwise **false**.

Remark

The comparison between map objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// map_op_ge.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map < int, int > m1, m2, m3, m4;
   int i;
   typedef pair < int, int > Int_Pair;

   for ( i = 1 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i - 1 ) );
      m4.insert ( Int_Pair ( i, i ) );
   }

   if ( m1 >= m2 )
      cout << "Map m1 is greater than or equal to map m2." << endl;
   else
      cout << "The map m1 is less than the map m2." << endl;

   if ( m1 >= m3 )
      cout << "Map m1 is greater than or equal to map m3." << endl;
   else
      cout << "The map m1 is less than the map m3." << endl;

   if ( m1 >= m4 )
      cout << "Map m1 is greater than or equal to map m4." << endl;
   else
      cout << "The map m1 is less than the map m4." << endl;
}
```

Output

```
The map m1 is less than the map m2.
Map m1 is greater than or equal to map m3.
Map m1 is greater than or equal to map m4.
```

Requirements

Header: <map>

See Also

Reference

Standard Template Library

Other Resources

<map> Members

operator!= (multimap)

Tests if the multimap object on the left side of the operator is not equal to the multimap object on the right side.

```
bool operator!=(
   const multimap <Key, Type, Traits, Allocator>& _Left,
   const multimap <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **multimap**.

_Right

An object of type **multimap**.

Return Value

true if the multimaps are not equal; **false** if multimaps are equal.

Remarks

The comparison between multimap objects is based on a pairwise comparison of their elements. Two multimaps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// multimap_op_ne.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1, m2, m3;
   int i;
   typedef pair <int, int> Int_Pair;

   for ( i = 0 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i ) );
   }

   if ( m1 != m2 )
      cout << "The multimaps m1 and m2 are not equal." << endl;
   else
      cout << "The multimaps m1 and m2 are equal." << endl;

   if ( m1 != m3 )
      cout << "The multimaps m1 and m3 are not equal." << endl;
   else
      cout << "The multimaps m1 and m3 are equal." << endl;
}
```

Output

```
The multimaps m1 and m2 are not equal.
The multimaps m1 and m3 are equal.
```

Requirements

Header: <map>

See Also

Reference

Standard Template Library

Other Resources

<map> Members

operator< (multimap)

Tests if the multimap object on the left side of the operator is less than the multimap object on the right side.

```
bool operator<(
   const multimap <Key, Type, Traits, Allocator>& _Left,
   const multimap <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **multimap**.

_Right

An object of type **multimap**.

Return Value

true if the multimap on the left side of the operator is strictly less than the multimap on the right side of the operator; otherwise **false**.

Remarks

The comparison between multimap objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// multimap_op_lt.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap < int, int > m1, m2, m3;
   int i;
   typedef pair < int, int > Int_Pair;

   for ( i = 1 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i - 1 ) );
   }

   if ( m1 < m2 )
      cout << "The multimap m1 is less than the multimap m2." << endl;
   else
      cout << "The multimap m1 is not less than the multimap m2." << endl;

   if ( m1 < m3 )
      cout << "The multimap m1 is less than the multimap m3." << endl;
   else
      cout << "The multimap m1 is not less than the multimap m3." << endl;
}
```

Output

```
The multimap m1 is less than the multimap m2.
The multimap m1 is not less than the multimap m3.
```

Requirements

Header: <map>

See Also

Reference

Standard Template Library

Other Resources

<map> Members

operator<= (multimap)

Tests if the multimap object on the left side of the operator is less than or equal to the multimap object on the right side.

```
bool operator<=(
   const multimap <Key, Type, Traits, Allocator>& _Left,
   const multimap <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **multimap**.

_Right

An object of type **multimap**.

Return Value

true if the multimap on the left side of the operator is less than or equal to the multimap on the right side of the operator; otherwise **false**.

Remark

The comparison between multimap objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// multimap_op_le.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1, m2, m3, m4;
   int i;
   typedef pair <int, int> Int_Pair;

   for ( i = 1 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i - 1 ) );
      m4.insert ( Int_Pair ( i, i ) );
   }

   if ( m1 <= m2 )
      cout << "m1 is less than or equal to m2" << endl;
   else
      cout << "m1 is greater than m2" << endl;

   if ( m1 <= m3 )
      cout << "m1 is less than or equal to m3" << endl;
   else
      cout << "m1 is greater than m3" << endl;

   if ( m1 <= m4 )
      cout << "m1 is less than or equal to m4" << endl;
   else
      cout << "m1 is greater than m4" << endl;
}
```

Output

```
m1 is less than or equal to m2
m1 is greater than m3
m1 is less than or equal to m4
```

Requirements

Header: <map>

See Also

Reference

Standard Template Library

Other Resources

<map> Members

operator== (multimap)

Tests if the multimap object on the left side of the operator is equal to the multimap object on the right side.

```
bool operator==(
   const multimap <Key, Type, Traits, Allocator>& _Left,
   const multimap <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **multimap**.

_Right

An object of type **multimap**.

Return Value

true if the multimap on the left side of the operator is equal to the multimap on the right side of the operator; otherwise **false**.

Remarks

The comparison between multimap objects is based on a pairwise comparison of their elements. Two multimaps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// multimap_op_eq.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap<int, int> m1, m2, m3;
   int i;
   typedef pair<int, int> Int_Pair;

   for (i = 0; i < 3; i++)
   {
      m1.insert(Int_Pair(i, i));
      m2.insert(Int_Pair(i, i*i));
      m3.insert(Int_Pair(i, i));
   }

   if ( m1 == m2 )
      cout << "m1 and m2 are equal" << endl;
   else
      cout << "m1 and m2 are not equal" << endl;

   if ( m1 == m3 )
      cout << "m1 and m3 are equal" << endl;
   else
      cout << "m1 and m3 are not equal" << endl;
}
```

Output

```
m1 and m2 are not equal
m1 and m3 are equal
```

Requirements

Header: <map>

See Also

Reference

Standard Template Library

Other Resources

<map> Members

operator> (multimap)

Tests if the multimap object on the left side of the operator is greater than the multimap object on the right side.

```
bool operator>(
   const multimap <Key, Type, Traits, Allocator>& _Left,
   const multimap <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **multimap**.

_Right

An object of type **multimap**.

Return Value

true if the multimap on the left side of the operator is greater than the multimap on the right side of the operator; otherwise **false**.

Remarks

The comparison between multimap objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// multimap_op_gt.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap < int, int > m1, m2, m3;
   int i;
   typedef pair < int, int > Int_Pair;

   for ( i = 0 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i - 1 ) );
   }

   if ( m1 > m2 )
      cout << "The multimap m1 is greater than the multimap m2." << endl;
   else
      cout << "Multimap m1 is not greater than multimap m2." << endl;

   if ( m1 > m3 )
      cout << "The multimap m1 is greater than the multimap m3." << endl;
   else
      cout << "The multimap m1 is not greater than the multimap m3." << endl;
}
```

Output

```
Multimap m1 is not greater than multimap m2.
The multimap m1 is greater than the multimap m3.
```

Requirements

Header: <map>

See Also

Reference

Standard Template Library

Other Resources

<map> Members

operator>= (multimap)

Tests if the multimap object on the left side of the operator is greater than or equal to the multimap object on the right side.

```
bool operator>=(
   const multimap <Key, Type, Traits, Allocator>& _Left,
   const multimap <Key, Type, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **multimap**.

_Right

An object of type **multimap**.

Return Value

true if the multimap on the left side of the operator is greater than or equal to the multimap on the right side of the list; otherwise **false**.

Remark

The comparison between multimap objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// multimap_op_ge.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap < int, int > m1, m2, m3, m4;
   int i;
   typedef pair < int, int > Int_Pair;

   for ( i = 1 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i - 1 ) );
      m4.insert ( Int_Pair ( i, i ) );
   }

   if ( m1 >= m2 )
      cout << "The multimap m1 is greater than or equal to the multimap m2." << endl;
   else
      cout << "The multimap m1 is less than the multimap m2." << endl;

   if ( m1 >= m3 )
      cout << "The multimap m1 is greater than or equal to the multimap m3." << endl;
   else
      cout << "The multimap m1 is less than the multimap m3." << endl;

   if ( m1 >= m4 )
      cout << "The multimap m1 is greater than or equal to the multimap m4." << endl;
   else
      cout << "The multimap m1 is less than the multimap m4." << endl;
}
```

Output

```
The multimap m1 is less than the multimap m2.
The multimap m1 is greater than or equal to the multimap m3.
The multimap m1 is greater than or equal to the multimap m4.
```

Requirements

Header: <map>

See Also

Reference

Standard Template Library

Other Resources

<map> Members

<map> Specialized Template Functions

For more information about specialized template functions in <map>, see <map> Members.

See Also

Reference

Standard Template Library

swap (map)

Exchanges the elements of two maps.

```
template<class _Key, class _Ty, class _Pr, class _Alloc>
   void swap(
   map<Key, Traits, Compare, Alloctor >& _Left,
   map<Key, Traits, Compare, Alloctor >& _Right
)
```

Parameters

_Right

The map providing the elements to be swapped, or the map whose elements are to be exchanged with those of the map *_Left*.

_Left

The map whose elements are to be exchanged with those of the map *_Right*.

Remarks

The template function is an algorithm specialized on the container class map to execute the member function *_Left*.swap(*_Right*). This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function, **template** <**class T**> **void swap**(**T&**, **T&**), in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

Example

See the code example for member function map::swap for an example that uses the template version of **swap**.

Requirements

Header: <map>

See Also

Reference

Standard Template Library

Other Resources

<map> Members

swap (multimap)

Exchanges the elements of two multimaps.

```
template<class _Key, class _Ty, class _Pr, class _Alloc>
void swap(
   multimap<Key, Traits, Compare, Alloctor >& _Left,
   multimap<Key, Traits, Compare, Alloctor >& _Right
)
```

Parameters

_Right

The multimap providing the elements to be swapped, or the multimap whose elements are to be exchanged with those of the multimap *_Left*.

_Left

The multimap whose elements are to be exchanged with those of the multimap *_Right*.

Remarks

The template function is an algorithm specialized on the container class map to execute on the container class multimap to execute the member function *_Left*.swap (*_Right*). This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function, **template** <**class T**> **void swap**(**T&**, **T&**), in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

Example

See the code example for member function multimap::swap for an example that uses the template version of **swap**.

Requirements

Header: <map>

See Also

Reference

Standard Template Library

Other Resources

<map> Members

<map> Classes

For more information about classes in <map>, see <map> Members.

See Also

Reference

Standard Template Library

value_compare Class (<map>)

Provides a function object that can compare the elements of a map by comparing the values of their keys to determine their relative order in the map.

```
class value_compare : public binary_function<value_type, value_type, bool>
{
friend class _Tmap_traits<_Kty, _Ty, _Pr, _Alloc, _Mfl>;
public:
   bool operator()(const value_type& _Left, const value_type& _Right) const;
   value_compare(key_compare _Pred) : comp(_Pred);
   protected:
      key_compare comp;
};
```

Remarks

The comparison criterion provided by **value_compare** between **value_types** of whole elements contained by a map is induced from a comparison between the keys of the respective elements by the auxiliary class construction. The member function operator uses the object **comp** of type **key_compare** stored in the function object provided by **value_compare** to compare the sort-key components of two elements.

For sets and multisets, which are simple containers where the key values are identical to the element values, **value_compare** is equivalent to **key_compare**; for maps and multimaps they are not, as the value of the type **pair** elements is not identical to the value of the element's key.

Example

See example for value_comp for an example of how to declare and use **value_compare**.

Requirements

Header: <map>

See Also

Reference

binary_function Class

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<map> Members

map Class

The STL map class is used for the storage and retrieval of data from a collection in which the each element is a pair that has both a data value and a sort key. The value of the key is unique and is used to order the data automatically. The value of an element in a map, but not its associated key value, may be changed directly. Instead, key values associated with old elements must be deleted and new key values associated with new elements inserted.

For a list of all members of this type, see map Members.

```
template <
   class Key,
   class Type,
   class Traits = less<Key>,
      class Allocator=allocator<pair <const Key, Type> >
>
class map
```

Parameters

Key

The key data type to be stored in the map.

Type

The element data type to be stored in the map.

Traits

The type that provides a function object that can compare two element values as sort keys to determine their relative order in the map. This argument is optional and the binary predicate **less**<*Key*> is the default value.

Allocator

The type that represents the stored allocator object that encapsulates details about the map's allocation and deallocation of memory. This argument is optional and the default value is **allocator**<**pair** <**const** *Key, Type*> >.

Remarks

The STL map class is:

- An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value.
- Reversible, because it provides bidirectional iterators to access its elements.
- Sorted, because its elements are ordered by key values within the container in accordance with a specified comparison function.
- Unique in the sense that each of its elements must have a unique key.
- A pair associative container, because its element data values are distinct from its key values.
- A template class, because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

The iterator provided by the map class is a bidirectional iterator, but the class member functions insert and map have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own set of requirements and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal set of functionality, but it is enough to be able to talk meaningfully about a range of iterators *[_First, _Last)* in the context of the class's member functions.

The choice of container type should be based in general on the type of searching and inserting required by the application.

Associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient, performing them in a time that is on average proportional to the logarithm of the number of elements in the container. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

The map should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. A model for this type of structure is an ordered list of uniquely occurring key words with associated string values providing, say, definitions. If, instead, the words had more than one correct definition, so that keys were not unique, then a multimap would be the container of choice. If, on the other hand, just the list of words were being stored, then a set would be the correct container. If multiple occurrences of the words were allowed, then a multiset would be the appropriate container structure.

The map orders the sequence it controls by calling a stored function object of type key_compare. This stored object is a comparison function that may be accessed by calling the member function key_comp. In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the non-equivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate *f*(*x*,*y*) is a function object that has two argument objects *x* and *y* and a return value of **true** or **false**. An ordering imposed on a set is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects *x* and *y* are defined to be equivalent when both *f*(*x*,*y*) and *f*(*y*,*x*) are false. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

Requirements

Header: <map>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

map Members

<map> Members

map Members

Reference

Constructors

map	Constructs a list of a specific size or with elements of a specific value or with a specific **allocator** or as a copy of some other map.

Typedefs

allocator_type	A type that represents the **allocator** class for the map object.
const_iterator	A type that provides a bidirectional iterator that can read a **const** element in the map.
const_pointer	A type that provides a pointer to a **const** element in a map.
const_reference	A type that provides a reference to a **const** element stored in a map for reading and performing **const** operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any **const** element in the map.
difference_type	A signed integer type that can be used to represent the number of elements of a map in a range between elements pointed to by iterators.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a map.
key_compare	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the map.
key_type	A type that describes the sort key object which constitutes each element of the map.
mapped_type	A type that represents the data type stored in a map.
pointer	A type that provides a pointer to a **const** element in a map.
reference	A type that provides a reference to an element stored in a map.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed map.
size_type	An unsigned integer type that can represent the number of elements in a map
value_type	A type that provides a function object that can compare two elements as sort keys to determine their relative order in the map.

Member Functions

begin	Returns an iterator addressing the first element in the map.
clear	Erases all the elements of a map.
count	Returns the number of elements in a map whose key matches a parameter-specified key.
empty	Tests if a map is empty.
end	Returns an iterator that addresses the location succeeding the last element in a map.

equal_range	Returns an iterator that addresses the location succeeding the last element in a map.
erase	Removes an element or a range of elements in a map from specified positions
find	Returns an iterator addressing the location of an element in a map that has a key equivalent to a specified key.
get_allocator	Returns a copy of the **allocator** object used to construct the map.
insert	Inserts an element or a range of elements into the map at a specified position.
key_comp	Retrieves a copy of the comparison object used to order keys in a map.
lower_bound	Returns an iterator to the first element in a map that with a key value that is equal to or greater than that of a sp ecified key.
max_size	Returns the maximum length of the map.
rbegin	Returns an iterator addressing the first element in a reversed map.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed map.
size	Returns the number of elements in the map.
swap	Exchanges the elements of two maps.
upper_bound	Returns an iterator to the first element in a map that with a key value that is greater than that of a specified key.
value_comp	Retrieves a copy of the comparison object used to order element values in a map.

Operators

operator[]	Inserts an element into a map with a specified key value.

See Also

Reference

map Class
Thread Safety in the Standard C++ Library
Standard Template Library

map Typedefs

For information about the typedefs in the map class, see map Members.

See Also

Reference

Standard Template Library

map::allocator_type

A type that represents the allocator class for the map object.

```
typedef Allocator allocator_type
```

Remark

allocator_type is a synonym for the template parameter **Allocator**.

For more information on **Allocator**, see the Remarks section of the map Class topic.

Example

See example for get_allocator for an example that uses **allocator_type**.

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::const_iterator

A type that provides a bidirectional iterator that can read a **const** element in the map.

```
typedef implementation-defined const_iterator;
```

Remarks

A type **const_iterator** cannot be used to modify the value of an element.

The **const_iterator** defined by map points to elements that are objects of value_type, that is of type **pair**<**const Key**, **Type**>, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a **const_iterator** *clter* pointing to an element in a map, use the **->** operator.

To access the value of the key for the element, use *clter* -> **first**, which is equivalent to (**clter*).**first**.

To access the value of the mapped datum for the element, use *clter* -> **second**, which is equivalent to (**clter*). **second**.

Example

See example for begin for an example that uses **const_iterator**.

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::const_pointer

A type that provides a pointer to a **const** element in a map.

```
typedef typename Allocator::const_pointer const_pointer;
```

Remarks

A type **const_pointer** cannot be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a map object.

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::const_reference

A type that provides a reference to a **const** element stored in a map for reading and performing **const** operations.

```
typedef typename Allocator::const_reference const_reference;
```

Example

```
// map_const_ref.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );

   // Declare and initialize a const_reference &Ref1
   // to the key of the first element
   const int &Ref1 = ( m1.begin( ) -> first );

   // The following line would cause an error as the
   // non-const_reference cannot be used to access the key
   // int &Ref1 = ( m1.begin( ) -> first );

   cout << "The key of first element in the map is "
        << Ref1 << "." << endl;

   // Declare and initialize a reference &Ref2
   // to the data value of the first element
   int &Ref2 = ( m1.begin( ) -> second );

   cout << "The data value of first element in the map is "
        << Ref2 << "." << endl;
}
```

Output

```
The key of first element in the map is 1.
The data value of first element in the map is 10.
```

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::const_reverse_iterator

A type that provides a bidirectional iterator that can read any **const** element in the map.

```
typedef reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type **const_reverse_iterator** cannot modify the value of an element and is use to iterate through the map in reverse.

The **const_reverse_iterator** defined by map points to elements that are objects of value_type, that is of type **pair**<**const Key**, **Type**>, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a **const_reverse_iterator** *crIter* pointing to an element in a map, use the **->** operator.

To access the value of the key for the element, use *crIter* -> **first**, which is equivalent to (***crIter*).**first**.

To access the value of the mapped datum for the element, use *crIter* -> **second**, which is equivalent to (***crIter*).**first**.

Example

See the example for rend for an example of how to declare and use **const_reverse_iterator**.

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::difference_type

A signed integer type that can be used to represent the number of elements of a map in a range between elements pointed to by iterators.

```
typedef implementation-defined difference_type;
```

Remarks

The **difference_type** is the type returned when subtracting or incrementing through iterators of the container. The **difference_type** is typically used to represent the number of elements in the range *[_First, _Last)* between the iterators *_First* and *_Last*, includes the element pointed to by *_First* and the range of elements up to, but not including, the element pointed to by *_Last*.

Note that although **difference_type** is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as set, subtraction between iterators is only supported by random access iterators provided by a random access container such as vector.

Example

```
// map_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <map>
#include <algorithm>

int main( )
{
   using namespace std;
   map <int, int> m1;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 3, 20 ) );
   m1.insert ( Int_Pair ( 2, 30 ) );

   map <int, int>::iterator m1_Iter, m1_bIter, m1_eIter;
   m1_bIter = m1.begin( );
   m1_eIter = m1.end( );

   // Count the number of elements in a map
   map <int, int>::difference_type  df_count = 1;
   m1_Iter = m1.begin( );
   while ( m1_Iter != m1_eIter)
   {
      df_count++;
      m1_Iter++;
   }

   cout << "The number of elements in the map m1 is: "
        << df_count << "." << endl;
}
```

Output

```
The number of elements in the map m1 is: 4.
```

Requirements

Header: <map>

See Also

Reference

map Class
Standard Template Library
Other Resources
map Members

map::iterator

A type that provides a bidirectional iterator that can read or modify any element in a map.

```
typedef implementation-defined iterator;
```

Remarks

The **iterator** defined by map points to elements that are objects of value_type, that is of type **pair**<**const Key**, **Type**>, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference an **iterator** *Iter* pointing to an element in a map, use the **->** operator.

To access the value of the key for the element, use *Iter* -> **first**, which is equivalent to (**Iter*).**first**. To access the value of the mapped datum for the element, use *Iter* -> **second**, which is equivalent to (**Iter*).**second**.

Example

See example for begin for an example of how to declare and use **iterator**.

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::key_compare

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the map.

```
typedef Traits key_compare;
```

Remarks

key_compare is a synonym for the template parameter **Traits**.

For more information on **Traits** see the map Class topic.

Example

See example for key_comp for an example of how to declare and use **key_compare**.

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::key_type

A type that describes the sort key object that constitutes each element of the map.

```
typedef Key key_type;
```

Remarks

key_type is a synonym for the template parameter **Key**.

For more information on **Key**, see the Remarks section of the map Class topic.

Example

See example for value_type for an example of how to declare and use **key_type**.

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::mapped_type

A type that represents the data type stored in a map.

```
typedef Type mapped_type;
```

Remarks

The type **mapped_type** is a synonym for the class's **Type** template parameter.

For more information on **Type** see the map Class topic.

Example

See example for value_type for an example of how to declare and use **mapped_type**.

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::pointer

A type that provides a pointer to an element in a map.

```
typedef typename Allocator::pointer pointer;
```

Remarks

A type **pointer** can be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a map object.

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::reference

A type that provides a reference to an element stored in a map.

```
typedef typename Allocator::reference reference;
```

Example

```
// map_reference.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );

   // Declare and initialize a const_reference &Ref1
   // to the key of the first element
   const int &Ref1 = ( m1.begin( ) -> first );

   // The following line would cause an error because the
   // non-const_reference cannot be used to access the key
   // int &Ref1 = ( m1.begin( ) -> first );

   cout << "The key of first element in the map is "
        << Ref1 << "." << endl;

   // Declare and initialize a reference &Ref2
   // to the data value of the first element
   int &Ref2 = ( m1.begin( ) -> second );

   cout << "The data value of first element in the map is "
        << Ref2 << "." << endl;

   //The non-const_reference can be used to modify the
   //data value of the first element
   Ref2 = Ref2 + 5;
   cout << "The modified data value of first element is "
        << Ref2 << "." << endl;
}
```

Output

```
The key of first element in the map is 1.
The data value of first element in the map is 10.
The modified data value of first element is 15.
```

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::reverse_iterator

A type that provides a bidirectional iterator that can read or modify an element in a reversed map.

```
typedef reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type **reverse_iterator** cannot modify the value of an element and is use to iterate through the map in reverse.

The **reverse_iterator** defined by map points to elements that are objects of value_type, that is of type **pair**<**const Key**, **Type**>, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a **reverse_iterator** *rIter* pointing to an element in a map, use the **->** operator.

To access the value of the key for the element, use *rIter* -> **first**, which is equivalent to (**rIter*).**first**. To access the value of the mapped datum for the element, use *rIter* -> **second**, which is equivalent to (**rIter*).**first**.

Example

See example for rbegin for an example of how to declare and use **reverse_iterator**.

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::size_type

An unsigned integer type that can represent the number of elements in a map.

```
typedef implementation-defined size_type;
```

Example

See the example for size for an example of how to declare and use **size_type**.

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::value_type

A type that represents the type of object stored as an element in a map.

```
typedef pair<const Key, Type> value_type;
```

Remark

value_type is declared to be **pair** <**const** key_type, mapped_type> and not simply **pair** <**key_type**, **mapped_typ***e*> because the keys of an associative container may not be changed using a nonconstant iterator or reference.

Example

```
// map_value_type.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   typedef pair <const int, int> cInt2Int;
   map <int, int> m1;
   map <int, int> :: key_type key1;
   map <int, int> :: mapped_type mapped1;
   map <int, int> :: value_type value1;
   map <int, int> :: iterator pIter;

   // value_type can be used to pass the correct type
   // explicitly to avoid implicit type conversion
   m1.insert ( map <int, int> :: value_type ( 1, 10 ) );

   // Compare other ways to insert objects into a map
   m1.insert ( cInt2Int ( 2, 20 ) );
   m1[ 3 ] = 30;

   // Initializing key1 and mapped1
   key1 = ( m1.begin( ) -> first );
   mapped1 = ( m1.begin( ) -> second );

   cout << "The key of first element in the map is "
        << key1 << "." << endl;

   cout << "The data value of first element in the map is "
        << mapped1 << "." << endl;

   // The following line would cause an error because
   // the value_type is not assignable
   // value1 = cInt2Int ( 4, 40 );

   cout  << "The keys of the mapped elements are:";
   for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
      cout << " " << pIter -> first;
   cout << "." << endl;

   cout  << "The values of the mapped elements are:";
   for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
      cout << " " << pIter -> second;
   cout << "." << endl;
}
```

Output

```
The key of first element in the map is 1.
```

```
The data value of first element in the map is 10.
The keys of the mapped elements are: 1 2 3.
The values of the mapped elements are: 10 20 30.
```

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map Member Functions

For information about the functions in the map class, see map Members.

See Also

Reference

Standard Template Library

map::begin

Returns an iterator addressing the first element in the map.

```
const_iterator begin( ) const;
iterator begin( );
```

Return Value

A bidirectional iterator addressing the first element in the map or the location succeeding an empty map.

Remark

If the return value of **begin** is assigned to a **const_iterator**, the elements in the map object cannot be modified. If the return value of **begin** is assigned to an **iterator**, the elements in the map object can be modified.

Example

```
// map_begin.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;

   map <int, int> :: iterator m1_Iter;
   map <int, int> :: const_iterator m1_cIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 0, 0 ) );
   m1.insert ( Int_Pair ( 1, 1 ) );
   m1.insert ( Int_Pair ( 2, 4 ) );

   m1_cIter = m1.begin ( );
   cout << "The first element of m1 is " << m1_cIter -> first << endl;

   m1_Iter = m1.begin ( );
   m1.erase ( m1_Iter );

   // The following 2 lines would err because the iterator is const
   // m1_cIter = m1.begin ( );
   // m1.erase ( m1_cIter );

   m1_cIter = m1.begin( );
   cout << "The first element of m1 is now " << m1_cIter -> first << endl;
}
```

Output

```
The first element of m1 is 0
The first element of m1 is now 1
```

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::clear

Erases all the elements of a map.

```
void clear( );
```

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// map_clear.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    map<int, int> m1;
    map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    m1.insert(Int_Pair(1, 1));
    m1.insert(Int_Pair(2, 4));

    i = m1.size();
    cout << "The size of the map is initially "
         << i << "." << endl;

    m1.clear();
    i = m1.size();
    cout << "The size of the map after clearing is "
         << i << "." << endl;
}
```

Output

```
The size of the map is initially 2.
The size of the map after clearing is 0.
```

Requirements

Header: <map>

See Also

Reference

map Class

map::max_size, map::clear, map::erase, and map::size

Standard Template Library

Other Resources

map Members

map::count

Returns the number of elements in a map whose key matches a parameter-specified key.

```
size_type count(
   const Key& _Key
) const;
```

Parameters

_Key

The key value of the elements to be matched from the map.

Return Value

1 if the map contains an element whose sort key matches the parameter key; 0 if the map does not contain an element with a matching key.

Remarks

The member function returns the number of elements *x* in the range

[**lower_bound** (*_Key*), **upper_bound** (*_Key*))

which is 0 or 1 in the case of map, which is a unique associative container.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// map_count.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    map<int, int> m1;
    map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    m1.insert(Int_Pair(1, 1));
    m1.insert(Int_Pair(2, 1));
    m1.insert(Int_Pair(1, 4));
    m1.insert(Int_Pair(2, 1));

    // Keys must be unique in map, so duplicates are ignored
    i = m1.count(1);
    cout << "The number of elements in m1 with a sort key of 1 is: "
         << i << "." << endl;

    i = m1.count(2);
    cout << "The number of elements in m1 with a sort key of 2 is: "
         << i << "." << endl;

    i = m1.count(3);
    cout << "The number of elements in m1 with a sort key of 3 is: "
         << i << "." << endl;
}
```

Output

```
The number of elements in m1 with a sort key of 1 is: 1.
```

```
The number of elements in m1 with a sort key of 2 is: 1.
The number of elements in m1 with a sort key of 3 is: 0.
```

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::empty

Tests if a map is empty.

```
bool empty( ) const;
```

Return Value

true if the map is empty; **false** if the map is nonempty.

Example

```
// map_empty.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1, m2;

   typedef pair <int, int> Int_Pair;
   m1.insert ( Int_Pair ( 1, 1 ) );

   if ( m1.empty( ) )
      cout << "The map m1 is empty." << endl;
   else
      cout << "The map m1 is not empty." << endl;

   if ( m2.empty( ) )
      cout << "The map m2 is empty." << endl;
   else
      cout << "The map m2 is not empty." << endl;
}
```

Output

```
The map m1 is not empty.
The map m2 is empty.
```

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::end

Returns an iterator that addresses the location succeeding the last element in a map.

```
const_iterator end( ) const;
iterator end( );
```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a map. If the map is empty, then map::end == map::begin.

Remarks

end is used to test whether an iterator has reached the end of its map.

The value returned by **end** should not be dereferenced.

Example

```
// map_end.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;

   map <int, int> :: iterator m1_Iter;
   map <int, int> :: const_iterator m1_cIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_cIter = m1.end( );
   m1_cIter--;
   cout << "The value of the last element of m1 is:\n"
        << m1_cIter -> second << endl;

   m1_Iter = m1.end( );
   m1_Iter--;
   m1.erase ( m1_Iter );

   // The following 2 lines would err because the iterator is const
   // m1_cIter = m1.end ( );
   // m1_cIter--;
   // m1.erase ( m1_cIter );

   m1_cIter = m1.end( );
   m1_cIter--;
   cout << "The value of the last element of m1 is now:\n"
        << m1_cIter -> second << endl;
}
```

Output

```
The value of the last element of m1 is:
30
The value of the last element of m1 is now:
20
```

Requirements

Header: <map>

See Also

Reference

map Class

map::insert, map::find, and map::end

Standard Template Library

Other Resources

map Members

map::equal_range

Returns a pair of iterators respectively to the first element in a map with a key that is greater than a specified key and to the first element in the map with a key that is equal to or greater than the key.

```
pair <const_iterator, const_iterator> equal_range (
   const Key& _Key
) const;
pair <iterator, iterator> equal_range (
   const Key& _Key
);
```

Parameters

_Key

The argument key value to be compared with the sort key of an element from the map being searched.

Return Value

A pair of iterators such that the first is the lower_bound of the key and the second is the upper_bound of the key.

To access the first iterator of a pair *pr* returned by the member function, use *pr*.**first**, and to dereference the lower bound iterator, use *(*pr*.**first**). To access the second iterator of a pair *pr* returned by the member function, use *pr*.**second**, and to dereference the upper bound iterator, use *(*pr*.**second**).

Example

```
// map_equal_range.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   typedef map <int, int, less<int> > IntMap;
   IntMap m1;
   map <int, int> :: const_iterator m1_RcIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   pair <IntMap::const_iterator, IntMap::const_iterator> p1, p2;
   p1 = m1.equal_range( 2 );

   cout << "The lower bound of the element with "
        << "a key of 2 in the map m1 is: "
        << p1.first -> second << "." << endl;

   cout << "The upper bound of the element with "
        << "a key of 2 in the map m1 is: "
        << p1.second -> second << "." << endl;

   // Compare the upper_bound called directly
   m1_RcIter = m1.upper_bound( 2 );

   cout << "A direct call of upper_bound( 2 ) gives "
        << m1_RcIter -> second << "," << endl
        << " matching the 2nd element of the pair"
        << " returned by equal_range( 2 )." << endl;

   p2 = m1.equal_range( 4 );
```

```
    // If no match is found for the key,
    // both elements of the pair return end( )
    if ( ( p2.first == m1.end( ) ) && ( p2.second == m1.end( ) ) )
       cout << "The map m1 doesn't have an element "
            << "with a key less than 40." << endl;
    else
       cout << "The element of map m1 with a key >= 40 is: "
            << p2.first -> first << "." << endl;
 }
```

Output

```
The lower bound of the element with a key of 2 in the map m1 is: 20.
The upper bound of the element with a key of 2 in the map m1 is: 30.
A direct call of upper_bound( 2 ) gives 30,
 matching the 2nd element of the pair returned by equal_range( 2 ).
The map m1 doesn't have an element with a key less than 40.
```

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::erase

Removes an element or a range of elements in a map from specified positions or removes elements that match a specified key.

```
iterator erase(
   iterator _Where
);
iterator erase(
   iterator _First,
   iterator _Last
);
size_type erase(
   const key_type& _Key
);
```

Parameters

_Where

Position of the element to be removed from the map.

_First

Position of the first element removed from the map.

_Last

Position just beyond the last element removed from the map.

_Key

The key value of the elements to be removed from the map.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the map if no such element exists.

Note
This return type does not conform to the C++ standard.

For the third member function, returns the number of elements that have been removed from the map.

Remarks

In some instances, this method might throw an out_of_range exception. This behavior does not conform to the C++ standard.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// map_erase.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    map<int, int> m1, m2, m3;
    map<int, int>::iterator pIter, Iter1, Iter2;
    int i;
    map<int, int>::size_type n;
    typedef pair<int, int> Int_Pair;
```

```
    for (i = 1; i < 5; i++)
    {
        m1.insert(Int_Pair(i, i));
        m2.insert(Int_Pair(i, i*i));
        m3.insert(Int_Pair(i, i-1));
    }

    // The 1st member function removes an element at a given position
    Iter1 = ++m1.begin();
    m1.erase(Iter1);

    cout << "After the 2nd element is deleted, the map m1 is:";
    for (pIter = m1.begin(); pIter != m1.end(); pIter++)
        cout << " " << pIter->second;
    cout << "." << endl;

    // The 2nd member function removes elements
    // in the range [_First, _Last)
    Iter1 = ++m2.begin();
    Iter2 = --m2.end();
    m2.erase(Iter1, Iter2);

    cout << "After the middle two elements are deleted,"
         << " the map m2 is:";
    for (pIter = m2.begin(); pIter != m2.end(); pIter++)
        cout << " " << pIter->second;
    cout << "." << endl;

    // The 3rd member function removes elements with a given _Key
    n = m3.erase(2);

    cout << "After the element with a key of 2 is deleted,\n"
         << "the map m3 is:";
    for (pIter = m3.begin(); pIter != m3.end(); pIter++)
        cout << " " << pIter->second;
    cout << "." << endl;

    // The 3rd member function returns the number of elements removed
    cout << "The number of elements removed from m3 is: "
         << n << "." << endl;

    // The dereferenced iterator can also be used to specify a key
    Iter1 = ++m3.begin();
    m3.erase(Iter1);

    cout << "After another element with a key equal to that"
         << endl;
    cout  << "of the 2nd element is deleted, "
          << "the map m3 is:";
    for (pIter = m3.begin(); pIter != m3.end(); pIter++)
        cout << " " << pIter->second;
    cout << "." << endl;
}
```

Output

```
After the 2nd element is deleted, the map m1 is: 1 3 4.
After the middle two elements are deleted, the map m2 is: 1 16.
After the element with a key of 2 is deleted,
the map m3 is: 0 2 3.
The number of elements removed from m3 is: 1.
After another element with a key equal to that
of the 2nd element is deleted, the map m3 is: 0 3.
```

Requirements

Header: <map>

See Also

Reference

map Class

map::max_size, map::clear, map::erase, and map::size

Standard Template Library

Other Resources

map Members

map::find

Returns an iterator addressing the location of an element in a map that has a key equivalent to a specified key.

```
iterator find(
   const Key& _Key
);
const_iterator find(
   const Key& _Key
) const;
```

Parameters

_Key

The key value to be matched by the sort key of an element from the map being searched.

Return Value

An iterator that addresses the location of an element with a specified key, or the location succeeding the last element in the map if no match is found for the key.

Remarks

The member function returns an iterator that addresses an element in the map whose sort key is equivalent to the argument key under a binary predicate that induces an ordering based on a less than comparability relation.

If the return value of **find** is assigned to a **const_iterator**, the map object cannot be modified. If the return value of **find** is assigned to an **iterator**, the map object can be modified

Example

```
// map_find.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;
   map <int, int> :: const_iterator m1_AcIter, m1_RcIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_RcIter = m1.find( 2 );
   cout << "The element of map m1 with a key of 2 is: "
        << m1_RcIter -> second << "." << endl;

   // If no match is found for the key, end( ) is returned
   m1_RcIter = m1.find( 4 );

   if ( m1_RcIter == m1.end( ) )
      cout << "The map m1 doesn't have an element "
           << "with a key of 4." << endl;
   else
      cout << "The element of map m1 with a key of 4 is: "
           << m1_RcIter -> second << "." << endl;

   // The element at a specific location in the map can be found
   // using a dereferenced iterator addressing the location
   m1_AcIter = m1.end( );
   m1_AcIter--;
```

```
    m1_RcIter = m1.find( m1_AcIter -> first );
    cout << "The element of m1 with a key matching "
         << "that of the last element is: "
         << m1_RcIter -> second << "." << endl;
}
```

Output

```
The element of map m1 with a key of 2 is: 20.
The map m1 doesn't have an element with a key of 4.
The element of m1 with a key matching that of the last element is: 30.
```

Requirements

Header: <map>

See Also

Reference

map Class

map::insert, map::find, and map::end

Standard Template Library

Other Resources

map Members

map::get_allocator

Returns a copy of the allocator object used to construct the map.

```
Allocator get_allocator( ) const;
```

Return Value

The allocator used by the map.

Remarks

Allocators for the map class specify how the class manages storage. The default allocators supplied with STL container classes is sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```
// map_get_allocator.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int>::allocator_type m1_Alloc;
   map <int, int>::allocator_type m2_Alloc;
   map <int, double>::allocator_type m3_Alloc;
   map <int, int>::allocator_type m4_Alloc;

   // The following lines declare objects
   // that use the default allocator.
   map <int, int> m1;
   map <int, int, allocator<int> > m2;
   map <int, double, allocator<double> > m3;

   m1_Alloc = m1.get_allocator( );
   m2_Alloc = m2.get_allocator( );
   m3_Alloc = m3.get_allocator( );

   cout << "The number of integers that can be allocated\n"
        << "before free memory is exhausted: "
        << m2.max_size( ) << ".\n" << endl;

   cout << "The number of doubles that can be allocated\n"
        << "before free memory is exhausted: "
        << m3.max_size( ) <<  ".\n" << endl;

   // The following line creates a map m4
   // with the allocator of map m1.
   map <int, int> m4( less<int>( ), m1_Alloc );

   m4_Alloc = m4.get_allocator( );

   // Two allocators are interchangeable if
   // storage allocated from each can be
   // deallocated with the other
   if( m1_Alloc == m4_Alloc )
   {
      cout << "The allocators are interchangeable." << endl;
   }
   else
   {
      cout << "The allocators are not interchangeable." << endl;
   }
```

```
}
```

Sample Output

The following output is for x86.

```
The number of integers that can be allocated
before free memory is exhausted: 536870911.

The number of doubles that can be allocated
before free memory is exhausted: 268435455.

The allocators are interchangeable.
```

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::insert

Inserts an element or a range of elements into a map.

```
pair <iterator, bool> insert(
   const value_type& _Val
);
iterator insert(
   iterator _Where,
   const value_type& _Val
);
template<class InputIterator>
   void insert(
      InputIterator _First,
      InputIterator _Last
   );
```

Parameters

_Val

The value of an element to be inserted into the map unless the map already contains that element or, more generally, an element whose key is equivalently ordered.

_Where

A hint regarding the place to start searching for the correct point of insertion.

_First

The position of the first element to be copied from a map.

_Last

The position just beyond the last element to be copied from a map.

Return Value

The first **insert** member function returns a pair whose bool component returns true if an insertion was made and false if the map already contained an element whose key had an equivalent value in the ordering, and whose iterator component returns the address where a new element was inserted or where the element was already located.

To access the iterator component of a pair *pr* returned by this member function, use *pr*.**first**, and to dereference it, use *(*pr*.**first**). To access the **bool** component of a pair *pr* returned by this member function, use *pr*.**second**.

The second **insert** member function, the hint version, returns an iterator that points to the position where the new element was inserted into the map.

Remarks

The value_type of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Insertion can occur in amortized constant time for the hint version of insert, instead of logarithmic time, if the insertion point immediately follows *_Where*.

The third member function inserts the sequence of element values into a map corresponding to each element addressed by an iterator of in the range *[_First, _Last)* of a specified set.

Example

```
// map_insert.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( ) {
   using namespace std;
```

```
    map <int, int>::iterator m1_pIter, m2_pIter;

    map <int, int> m1, m2;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 3, 30 ) );
    m1.insert ( Int_Pair ( 4, 40 ) );

    cout << "The original key values of m1 =";
    for ( m1_pIter = m1.begin( ); m1_pIter != m1.end( ); m1_pIter++ )
       cout << " " << m1_pIter -> first;
    cout << "." << endl;

    cout << "The original mapped values of m1 =";
    for ( m1_pIter = m1.begin( ); m1_pIter != m1.end( ); m1_pIter++ )
       cout << " " << m1_pIter -> second;
    cout << "." << endl;

    pair< map<int,int>::iterator, bool > pr;
    pr = m1.insert ( Int_Pair ( 1, 10 ) );

    if( pr.second == true ) {
       cout << "The element 10 was inserted in m1 successfully." << endl;
    }
    else {
       cout << "Key number 1 already exists in m1\n"
            << "with an associated value of ( (pr.first) -> second ) = "
            << ( pr.first ) -> second
            << "." << endl;
    }

    // The hint version of insert
    m1.insert( --m1.end( ), Int_Pair ( 5, 50 ) );

    cout << "After the insertions, the key values of m1 =";
    for ( m1_pIter = m1.begin( ); m1_pIter != m1.end( ); m1_pIter++ )
       cout << " " << m1_pIter -> first;
    cout << "," << endl;

    cout << "and the mapped values of m1 =";
    for ( m1_pIter = m1.begin( ); m1_pIter != m1.end( ); m1_pIter++ )
       cout << " " << m1_pIter -> second;
    cout << "." << endl;

    m2.insert ( Int_Pair ( 10, 100 ) );

    // The templatized version inserting a range
    m2.insert( ++m1.begin( ), --m1.end( ) );

    cout << "After the insertions, the key values of m2 =";
    for ( m2_pIter = m2.begin( ); m2_pIter != m2.end( ); m2_pIter++ )
       cout << " " << m2_pIter -> first;
    cout << "," << endl;

    cout << "and the mapped values of m2 =";
    for ( m2_pIter = m2.begin( ); m2_pIter != m2.end( ); m2_pIter++ )
       cout << " " << m2_pIter -> second;
    cout << "." << endl;
  }
```

Output

```
The original key values of m1 = 1 2 3 4.
The original mapped values of m1 = 10 20 30 40.
Key number 1 already exists in m1
with an associated value of ( (pr.first) -> second ) = 10.
```

```
After the insertions, the key values of m1 = 1 2 3 4 5,
and the mapped values of m1 = 10 20 30 40 50.
After the insertions, the key values of m2 = 2 3 4 10,
and the mapped values of m2 = 20 30 40 100.
```

Requirements

Header: <map>

See Also

Reference

map Class

map::insert, map::find, and map::end

Standard Template Library

Other Resources

map Members

map::key_comp

Retrieves a copy of the comparison object used to order keys in a map.

```
key_compare key_comp( ) const;
```

Return Value

Returns the function object that a map uses to order its elements.

Remarks

The stored object defines the member function

bool operator(**const Key&** *_Left*, **const Key&** *_Right*);

which returns **true** if *_Left* precedes and is not equal to *_Right* in the sort order.

Example

```
// map_key_comp.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;

   map <int, int, less<int> > m1;
   map <int, int, less<int> >::key_compare kc1 = m1.key_comp( ) ;
   bool result1 = kc1( 2, 3 ) ;
   if( result1 == true )
   {
      cout << "kc1( 2,3 ) returns value of true, "
           << "where kc1 is the function object of m1."
           << endl;
   }
   else
   {
      cout << "kc1( 2,3 ) returns value of false "
           << "where kc1 is the function object of m1."
           << endl;
   }

   map <int, int, greater<int> > m2;
   map <int, int, greater<int> >::key_compare kc2 = m2.key_comp( );
   bool result2 = kc2( 2, 3 ) ;
   if( result2 == true )
   {
      cout << "kc2( 2,3 ) returns value of true, "
           << "where kc2 is the function object of m2."
           << endl;
   }
   else
   {
      cout << "kc2( 2,3 ) returns value of false, "
           << "where kc2 is the function object of m2."
           << endl;
   }
}
```

Output

```
kc1( 2,3 ) returns value of true, where kc1 is the function object of m1.
```

```
kc2( 2,3 ) returns value of false, where kc2 is the function object of m2.
```

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::lower_bound

Returns an iterator to the first element in a map with a key value that is equal to or greater than that of a specified key.

```
iterator lower_bound(
   const Key& _Key
);
const_iterator lower_bound(
   const Key& _Key
) const;
```

Parameters

_Key

The argument key value to be compared with the sort key of an element from the map being searched.

Return Value

An **iterator** or **const_iterator** that addresses the location of an element in a map that with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the map if no match is found for the key.

If the return value of **lower_bound** is assigned to a **const_iterator**, the map object cannot be modified. If the return value of **lower_bound** is assigned to an **iterator**, the map object can be modified.

Example

```
// map_lower_bound.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;
   map <int, int> :: const_iterator m1_AcIter, m1_RcIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_RcIter = m1.lower_bound( 2 );
   cout << "The first element of map m1 with a key of 2 is: "
        << m1_RcIter -> second << "." << endl;

   // If no match is found for this key, end( ) is returned
   m1_RcIter = m1. lower_bound ( 4 );

   if ( m1_RcIter == m1.end( ) )
      cout << "The map m1 doesn't have an element "
           << "with a key of 4." << endl;
   else
      cout << "The element of map m1 with a key of 4 is: "
           << m1_RcIter -> second << "." << endl;

   // The element at a specific location in the map can be found
   // using a dereferenced iterator addressing the location
   m1_AcIter = m1.end( );
   m1_AcIter--;
   m1_RcIter = m1. lower_bound ( m1_AcIter -> first );
   cout << "The element of m1 with a key matching "
        << "that of the last element is: "
        << m1_RcIter -> second << "." << endl;
}
```

Output

```
The first element of map m1 with a key of 2 is: 20.
The map m1 doesn't have an element with a key of 4.
The element of m1 with a key matching that of the last element is: 30.
```

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::map

Constructs a map that is empty or that is a copy of all or part of some other map.

```
map( );
explicit map(
   const Traits& _Comp
);
explicit map(
   const Traits& _Comp,
   const Allocator& _Al
);
map(
   const _map& _Right
);
template<class InputIterator>
   map(
      InputIterator _First,
      InputIterator _Last
   );
template<class InputIterator>
   map(
      InputIterator _First,
      InputIterator _Last,
      const Traits& _Comp
   );
template<class InputIterator>
   map(
      InputIterator _First,
      InputIterator _Last,
      const Traits& _Comp,
      const Allocator& _Al
   );
```

Parameters

_Al

The storage allocator class to be used for this map object, which defaults to **Allocator**.

_Comp

The comparison function of type **const Traits** used to order the elements in the map, which defaults to **hash_compare**.

_Right

The map of which the constructed set is to be a copy.

_First

The position of the first element in the range of elements to be copied.

_Last

The position of the first element beyond the range of elements to be copied.

Remarks

All constructors store a type of allocator object that manages memory storage for the map and that can later be returned by calling get_allocator. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their map.

All constructors store a function object of type Traits that is used to establish an order among the keys of the map and that can later be returned by calling key_comp.

The first three constructors specify an empty initial map, the second specifying the type of comparison function (*_Comp*) to be

used in establishing the order of the elements and the third explicitly specifying the allocator type (*_Al*) to be used. The key word **explicit** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the map *_Right*.

The last three constructors copy the range [*_First, _Last*) of a map with increasing explicitness in specifying the type of comparison function of class **Traits** and allocator.

Example

```
// map_map.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   typedef pair <int, int> Int_Pair;
   map <int, int>::iterator m1_Iter, m3_Iter, m4_Iter, m5_Iter, m6_Iter;
   map <int, int, greater<int> >::iterator m2_Iter;

   // Create an empty map m0 of key type integer
   map <int, int> m0;

   // Create an empty map m1 with the key comparison
   // function of less than, then insert 4 elements
   map <int, int, less<int> > m1;
   m1.insert( Int_Pair( 1, 10 ) );
   m1.insert( Int_Pair( 2, 20 ) );
   m1.insert( Int_Pair( 3, 30 ) );
   m1.insert( Int_Pair( 4, 40 ) );

   // Create an empty map m2 with the key comparison
   // function of geater than, then insert 2 elements
   map <int, int, greater<int> > m2;
   m2.insert( Int_Pair( 1, 10 ) );
   m2.insert( Int_Pair( 2, 20 ) );

   // Create a map m3 with the
   // allocator of map m1
   map <int, int>::allocator_type m1_Alloc;
   m1_Alloc = m1.get_allocator( );
   map <int, int> m3( less<int>( ), m1_Alloc );
   m3.insert( Int_Pair( 3, 30 ) );

   // Create a copy, map m4, of map m1
   map <int, int> m4( m1 );

   // Create a map m5 by copying the range m1[_First, _Last)
   map <int, int>::const_iterator m1_bcIter, m1_ecIter;
   m1_bcIter = m1.begin( );
   m1_ecIter = m1.begin( );
   m1_ecIter++;
   m1_ecIter++;
   map <int, int> m5(m1_bcIter, m1_ecIter);

   // Create a map m6 by copying the range m4[_First, _Last)
   // and with the allocator of map m2
   map <int, int>::allocator_type m2_Alloc;
   m2_Alloc = m2.get_allocator( );
   map <int, int> m6( m4.begin( ), ++m4.begin( ), less<int>( ), m2_Alloc);

   cout << "m1 =";
   for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
      cout << " " << m1_Iter -> second;
   cout << endl;

   cout << "m2 =";
```

```
    for ( m2_Iter = m2.begin( ); m2_Iter != m2.end( ); m2_Iter++ )
       cout << " " << m2_Iter -> second;
    cout << endl;

    cout << "m3 =";
    for ( m3_Iter = m3.begin( ); m3_Iter != m3.end( ); m3_Iter++ )
       cout << " " << m3_Iter -> second;
    cout << endl;

    cout << "m4 =";
    for ( m4_Iter = m4.begin( ); m4_Iter != m4.end( ); m4_Iter++ )
       cout << " " << m4_Iter -> second;
    cout << endl;

    cout << "m5 =";
    for ( m5_Iter = m5.begin( ); m5_Iter != m5.end( ); m5_Iter++ )
       cout << " " << m5_Iter -> second;
    cout << endl;

    cout << "m6 =";
    for ( m6_Iter = m6.begin( ); m6_Iter != m6.end( ); m6_Iter++ )
       cout << " " << m6_Iter -> second;
    cout << endl;
 }
```

Output

```
m1 = 10 20 30 40
m2 = 20 10
m3 = 30
m4 = 10 20 30 40
m5 = 10 20
m6 = 10
```

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::max_size

Returns the maximum length of the map.

```
size_type max_size( ) const;
```

Return Value

The maximum possible length of the map.

Example

```
// map_max_size.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;
   map <int, int> :: size_type i;

   i = m1.max_size( );
   cout << "The maximum possible length "
        << "of the map is " << i << "."
        << endl << "(Magnitude is machine specific.)";
}
```

Sample Output

The following output is for x86.

```
The maximum possible length of the map is 536870911.
(Magnitude is machine specific.)
```

Requirements

Header: <map>

See Also

Reference

map Class

map::max_size, map::clear, map::erase, and map::size

Standard Template Library

Other Resources

map Members

map::rbegin

Returns an iterator addressing the first element in a reversed map.

```
const_reverse_iterator rbegin( ) const;
reverse_iterator rbegin( );
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed map or addressing what had been the last element in the unreversed map.

Remarks

rbegin is used with a reversed map just as begin is used with a map.

If the return value of **rbegin** is assigned to a **const_reverse_iterator**, then the map object cannot be modified. If the return value of **rbegin** is assigned to a **reverse_iterator**, then the map object can be modified.

rbegin can be used to iterate through a map backwards.

Example

```
// map_rbegin.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;

   map <int, int> :: iterator m1_Iter;
   map <int, int> :: reverse_iterator m1_rIter;
   map <int, int> :: const_reverse_iterator m1_crIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_rIter = m1.rbegin( );
   cout << "The first element of the reversed map m1 is "
        << m1_rIter -> first << "." << endl;

   // begin can be used to start an iteration
   // through a map in a forward order
   cout << "The map is: ";
   for ( m1_Iter = m1.begin( ) ; m1_Iter != m1.end( ); m1_Iter++)
      cout << m1_Iter -> first << " ";
      cout << "." << endl;

   // rbegin can be used to start an iteration
   // through a map in a reverse order
   cout << "The reversed map is: ";
   for ( m1_rIter = m1.rbegin( ) ; m1_rIter != m1.rend( ); m1_rIter++)
      cout << m1_rIter -> first << " ";
      cout << "." << endl;

   // A map element can be erased by dereferencing to its key
   m1_rIter = m1.rbegin( );
   m1.erase ( m1_rIter -> first );

   m1_rIter = m1.rbegin( );
```

```
    cout << "After the erasure, the first element "
         << "in the reversed map is "
         << m1_rIter -> first << "." << endl;
}
```

Output

```
The first element of the reversed map m1 is 3.
The map is: 1 2 3 .
The reversed map is: 3 2 1 .
After the erasure, the first element in the reversed map is 2.
```

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::rend

Returns an iterator that addresses the location succeeding the last element in a reversed map.

```
const_reverse_iterator rend( ) const;
reverse_iterator rend( );
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed map (the location that had preceded the first element in the unreversed map).

Remarks

rend is used with a reversed map just as end is used with a map.

If the return value of **rend** is assigned to a **const_reverse_iterator**, then the map object cannot be modified. If the return value of **rend** is assigned to a **reverse_iterator**, then the map object can be modified.

rend can be used to test to whether a reverse iterator has reached the end of its map.

The value returned by **rend** should not be dereferenced.

Example

```
// map_rend.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;

   map <int, int> :: iterator m1_Iter;
   map <int, int> :: reverse_iterator m1_rIter;
   map <int, int> :: const_reverse_iterator m1_crIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_rIter = m1.rend( );
   m1_rIter--;
   cout << "The last element of the reversed map m1 is "
        << m1_rIter -> first << "." << endl;

   // begin can be used to start an iteration
   // through a map in a forward order
   cout << "The map is: ";
   for ( m1_Iter = m1.begin( ) ; m1_Iter != m1.end( ); m1_Iter++)
      cout << m1_Iter -> first << " ";
      cout << "." << endl;

   // rbegin can be used to start an iteration
   // through a map in a reverse order
   cout << "The reversed map is: ";
   for ( m1_rIter = m1.rbegin( ) ; m1_rIter != m1.rend( ); m1_rIter++)
      cout << m1_rIter -> first << " ";
      cout << "." << endl;

   // A map element can be erased by dereferencing to its key
   m1_rIter = --m1.rend( );
   m1.erase ( m1_rIter -> first );
```

```
    m1_rIter = m1.rend( );
    m1_rIter--;
    cout << "After the erasure, the last element "
         << "in the reversed map is "
         << m1_rIter -> first << "." << endl;
}
```

Output

```
The last element of the reversed map m1 is 1.
The map is: 1 2 3 .
The reversed map is: 3 2 1 .
After the erasure, the last element in the reversed map is 2.
```

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::size

Returns the number of elements in the map.

```
size_type size( ) const;
```

Return Value

The current length of the map.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// map_size.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    map<int, int> m1, m2;
    map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    m1.insert(Int_Pair(1, 1));
    i = m1.size();
    cout << "The map length is " << i << "." << endl;

    m1.insert(Int_Pair(2, 4));
    i = m1.size();
    cout << "The map length is now " << i << "." << endl;
}
```

Output

```
The map length is 1.
The map length is now 2.
```

Requirements

Header: <map>

See Also

Reference

map Class

map::max_size, map::clear, map::erase, and map::size

Standard Template Library

Other Resources

map Members

map::swap

Exchanges the elements of two maps.

```
void swap(
   map& _Right
);
```

Parameters

_Right

The argument map providing the elements to be swapped with the target map.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two maps whose elements are being exchanged.

Example

```
// map_swap.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1, m2, m3;
   map <int, int>::iterator m1_Iter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );
   m2.insert ( Int_Pair ( 10, 100 ) );
   m2.insert ( Int_Pair ( 20, 200 ) );
   m3.insert ( Int_Pair ( 30, 300 ) );

   cout << "The original map m1 is:";
   for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
      cout << " " << m1_Iter -> second;
   cout   << "." << endl;

   // This is the member function version of swap
   //m2 is said to be the argument map; m1 the target map
   m1.swap( m2 );

   cout << "After swapping with m2, map m1 is:";
   for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
      cout << " " << m1_Iter -> second;
   cout  << "." << endl;

   // This is the specialized template version of swap
   swap( m1, m3 );

   cout << "After swapping with m3, map m1 is:";
   for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
      cout << " " << m1_Iter -> second;
   cout   << "." << endl;
}
```

Output

```
The original map m1 is: 10 20 30.
```

```
After swapping with m2, map m1 is: 100 200.
After swapping with m3, map m1 is: 300.
```

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::upper_bound

Returns an iterator to the first element in a map that with a key having a value that is greater than that of a specified key.

```
iterator upper_bound(
   const Key& _Key
);
const_iterator upper_bound(
   const Key& _Key
) const;
```

Parameters

_Key

The argument key value to be compared with the sort key value of an element from the map being searched.

Return Value

An **iterator** or **const_iterator** that addresses the location of an element in a map that with a key that is greater than the argument key, or that addresses the location succeeding the last element in the map if no match is found for the key.

If the return value is assigned to a **const_iterator**, the map object cannot be modified. If the return value is assigned to a **iterator**, the map object can be modified.

Example

```
// map_upper_bound.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;
   map <int, int> :: const_iterator m1_AcIter, m1_RcIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_RcIter = m1.upper_bound( 2 );
   cout << "The first element of map m1 with a key "
        << "greater than 2 is: "
        << m1_RcIter -> second << "." << endl;

   // If no match is found for the key, end is returned
   m1_RcIter = m1. upper_bound ( 4 );

   if ( m1_RcIter == m1.end( ) )
      cout << "The map m1 doesn't have an element "
           << "with a key greater than 4." << endl;
   else
      cout << "The element of map m1 with a key > 4 is: "
           << m1_RcIter -> second << "." << endl;

   // The element at a specific location in the map can be found
   // using a dereferenced iterator addressing the location
   m1_AcIter = m1.begin( );
   m1_RcIter = m1. upper_bound ( m1_AcIter -> first );
   cout << "The 1st element of m1 with a key greater than\n"
        << "that of the initial element of m1 is: "
        << m1_RcIter -> second << "." << endl;
}
```

Output

```
The first element of map m1 with a key greater than 2 is: 30.
The map m1 doesn't have an element with a key greater than 4.
The 1st element of m1 with a key greater than
that of the initial element of m1 is: 20.
```

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map::value_comp

The member function returns a function object that determines the order of elements in a map by comparing their key values.

```
value_compare value_comp( ) const;
```

Return Value

Returns the comparison function object that a map uses to order its elements.

Remarks

For a map *m*, if two elements *e*1(*k*1, *d*1) and *e*2(*k*2, *d*2) are objects of type **value_type**, where *k*1 and *k*2 are their keys of type **key_type** and *d*1 and *d*2 are their data of type **mapped_type**, then *m*.**value_comp**(*e*1, *e*2) is equivalent to *m*.**key_comp**(*k*1, *k*2). A stored object defines the member function

bool operator(**value_type&** *_Left*, **value_type&** *_Right*);

which returns **true** if the key value of *_Left* precedes and is not equal to the key value of *_Right* in the sort order.

Example

```
// map_value_comp.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;

   map <int, int, less<int> > m1;
   map <int, int, less<int> >::value_compare vc1 = m1.value_comp( );
   pair< map<int,int>::iterator, bool > pr1, pr2;

   pr1= m1.insert ( map <int, int> :: value_type ( 1, 10 ) );
   pr2= m1.insert ( map <int, int> :: value_type ( 2, 5 ) );

   if( vc1( *pr1.first, *pr2.first ) == true )
   {
      cout << "The element ( 1,10 ) precedes the element ( 2,5 )."
           << endl;
   }
   else
   {
      cout << "The element ( 1,10 ) does not precede the element ( 2,5 )."
           << endl;
   }

   if(vc1( *pr2.first, *pr1.first ) == true )
   {
      cout << "The element ( 2,5 ) precedes the element ( 1,10 )."
           << endl;
   }
   else
   {
      cout << "The element ( 2,5 ) does not precede the element ( 1,10 )."
           << endl;
   }
}
```

Output

```
The element ( 1,10 ) precedes the element ( 2,5 ).
The element ( 2,5 ) does not precede the element ( 1,10 ).
```

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

map Operators

For information about the operators in the map class, see map Members.

See Also

Reference

Standard Template Library

map::operator[]

Inserts an element into a map with a specified key value.

```
Type& operator[](
   const Key& _Key
);
```

Parameters

_Key

The key value of the element that is to be inserted.

Return Value

A reference to the data value of the inserted element.

Remarks

If the argument key value is not found, then it is inserted along with the default value of the data type.

operator[] may be used to insert elements into a map *m* using *m*[*_Key*] = *DataValue*; where *DataValue* is the value of the **mapped_type** of the element with a key value of *_Key*.

When using **operator[]** to insert elements, the returned reference does not indicate whether an insertion is changing a pre-existing element or creating a new one. The member functions find and insert can be used to determine whether an element with a specified key is already present before an insertion.

Example

```
// map_op_insert.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   typedef pair <const int, int> cInt2Int;
   map <int, int> m1;
   map <int, int> :: iterator pIter;

   // Insert a data value of 10 with a key of 1
   // into a map using the operator[] member function
   m1[ 1 ] = 10;

   // Compare other ways to insert objects into a map
   m1.insert ( map <int, int> :: value_type ( 2, 20 ) );
   m1.insert ( cInt2Int ( 3, 30 ) );

   cout  << "The keys of the mapped elements are:";
   for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
      cout << " " << pIter -> first;
   cout << "." << endl;

   cout  << "The values of the mapped elements are:";
   for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
      cout << " " << pIter -> second;
   cout << "." << endl;

   // If the key already exists, operator[]
   // changes the value of the datum in the element
   m1[ 2 ] = 40;

   // operator[] will also insert the value of the data
   // type's default constructor if the value is unspecified
```

```
    m1[5];

    cout  << "The keys of the mapped elements are now:";
    for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
       cout << " " << pIter -> first;
    cout << "." << endl;

    cout  << "The values of the mapped elements are now:";
    for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
       cout << " " << pIter -> second;
    cout << "." << endl;
 }
```

Output

```
The keys of the mapped elements are: 1 2 3.
The values of the mapped elements are: 10 20 30.
The keys of the mapped elements are now: 1 2 3 5.
The values of the mapped elements are now: 10 40 30 0.
```

Requirements

Header: <map>

See Also

Reference

map Class

Standard Template Library

Other Resources

map Members

multimap Class

The Standard Template Library multimap class is used for the storage and retrieval of data from a collection in which the each element is a pair that has both a data value and a sort key. The value of the key does not need to be unique and is used to order the data automatically. The value of an element in a multimap, but not its associated key value, may be changed directly. Instead, key values associated with old elements must be deleted and new key values associated with new elements inserted.

For a list of all members of this type, see multimap Members.

```
template <
   class Key,
   class Type,
   class Traits=less<Key>,
      class Allocator=allocator<pair <const Key, Type> >
>
class multimap
```

Parameters

Key

The key data type to be stored in the multimap.

Type

The element data type to be stored in the multimap.

Traits

The type that provides a function object that can compare two element values as sort keys to determine their relative order in the multimap. The binary predicate **less**<*Key*> is the default value.

Allocator

The type that represents the stored allocator object that encapsulates details about the map's allocation and deallocation of memory. This argument is optional and the default value is **allocator**<**pair** <**const** *Key, Type*> >.

Remarks

The STL multimap class is

- An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value.
- Reversible, because it provides bidirectional iterators to access its elements.
- Sorted, because its elements are ordered by key values within the container in accordance with a specified comparison function.
- Multiple, because its elements do not need to have a unique keys, so that one key value may have many element data values associated with it.
- A pair associative container, because its element data values are distinct from its key values.
- A template class, because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

The iterator provided by the map class is a bidirectional iterator, but the class member functions insert and multimap have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own set of requirements and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal set of functionality, but it is enough to be able to talk meaningfully about a range of iterators *[_First, _Last)* in the context of the class's member functions.

The choice of container type should be based in general on the type of searching and inserting required by the application. Associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient, performing them in a time that is on average proportional to the logarithm of the number of elements in the container. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

The multimap should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. A model for this type of structure is an ordered list of key words with associated string values providing, say, definitions, where the words were not always uniquely defined. If, instead, the key words were uniquely defined so that keys were unique, then a map would be the container of choice. If, on the other hand, just the list of words were being stored, then a set would be the correct container. If multiple occurrences of the words were allowed, then a multiset would be the appropriate container structure.

The multimap orders the sequence it controls by calling a stored function object of type key_compare. This stored object is a comparison function that may be accessed by calling the member function key_comp. In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate *f*(*x,y*) is a function object that has two argument objects *x* and *y* and a return value of true or false. An ordering imposed on a set is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects *x* and y are defined to be equivalent when both *f*(*x,y*) and *f*(*y,x*) are false. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

Requirements

Header: <map>

The (**key**, **value**) pairs are stored in a multimap as objects of type **pair**. The pair class requires the header <utility>, which is automatically included by <map>.

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

multimap Members

<map> Members

Standard C++ Library Header Files

multimap Members

Reference

Constructors

multimap	Constructs a **multimap** that is empty or that is a copy of all or part of some other **multimap**.

Typedefs

allocator_type	A type that represents the **allocator** class for the **multimap** object.
const_iterator	A type that provides a bidirectional iterator that can read a **const** element in the **multimap**.
const_pointer	A type that provides a pointer to a **const** element in a **multimap**.
const_reference	A type that provides a reference to a **const** element stored in a **multimap** for reading and performing **const** operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any **const** element in the **multimap**.
difference_type	A signed integer type that can be used to represent the number of elements of a **multimap** in a range between elements pointed to by iterators.
iterator	A type that provides the difference between two iterators that refer to elements within the same **multimap**.
key_compare	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the **multimap**.
key_type	A type that describes the sort key object that constitutes each element of the **multimap**.
mapped_type	A type that represents the data type stored in a **multimap**.
pointer	A type that provides a pointer to a **const** element in a **multimap**.
reference	A type that provides a reference to an element stored in a **multimap**.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed **multimap**.
size_type	An unsigned integer type that provides a pointer to a **const** element in a **multimap**.
value_type	A type that provides a function object that can compare two elements as sort keys to determine their relative order in the **multimap**.

Member Functions

begin	Returns an iterator addressing the first element in the **multimap**.
clear	Erases all the elements of a **multimap**.
count	Returns the number of elements in a **multimap** whose key matches a parameter-specified key.
empty	Tests if a **multimap** is empty.

end	Returns an iterator that addresses the location succeeding the last element in a **multimap**.
equal_range	Returns a pair of iterators respectively to the first element in a **multimap** with a key that is greater than a specified key and to the first element in the **multimap** with a key that is equal to or greater than the key.
erase	Removes an element or a range of elements in a **multimap** from specified positions or removes elements that match a specified key.
find	Returns an iterator addressing the first location of an element in a **multimap** that has a key equivalent to a specified key.
get_allocator	Returns a copy of the **allocator** object used to construct the **multimap**.
insert	Inserts an element or a range of elements into a **multimap**.
key_comp	Retrieves a copy of the comparison object used to order keys in a **multimap**.
lower_bound	Returns an iterator to the first element in a **multimap** that with a key that is equal to or greater than a specified key.
max_size	Returns the maximum length of the **multimap**.
rbegin	Returns an iterator addressing the first element in a reversed **multimap**.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed **multimap**.
size	Returns the number of elements in the **multimap**.
swap	Exchanges the elements of two **multimap**s.
upper_bound	Returns an iterator to the first element in a **multimap** that with a key that is greater than a specified key.
value_comp	The member function returns a function object that determines the order of elements in a **multimap** by comparing their key values.

See Also

Reference

multimap Class
Thread Safety in the Standard C++ Library
Standard Template Library

multimap Typedefs

For information about the typedefs in the multimap class, see multimap Members.

See Also

Reference

Standard Template Library

multimap::allocator_type

A type that represents the allocator class for the multimap object.

```
typedef Allocator allocator_type
```

Remark

allocator_type is a synonym for the template parameter **Allocator**.

Example

See the example for get_allocator for an example using **allocator_type**.

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::const_iterator

A type that provides a bidirectional iterator that can read a **const** element in the multimap.

```
typedef implementation-defined const_iterator;
```

Remarks

A type **const_iterator** cannot be used to modify the value of an element.

The **const_iterator** defined by multimap points to objects of value_type, which are of type **pair**<**const Key**, **Type**>. The value of the key is available through the first member pair and the value of the mapped element is available through the second member of the pair.

To dereference a **const_iterator** *clter* pointing to an element in a multimap, use the **->** operator.

To access the value of the key for the element, use *clter* -> **first**, which is equivalent to (**clter*).**first**. To access the value of the mapped datum for the element, use *clter* -> **second**, which is equivalent to (**clter*).**second**.

Example

See the example for begin for an example using **const_iterator**.

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::const_pointer

A type that provides a pointer to a **const** element in a multimap.

```
typedef typename Allocator::const_pointer const_pointer;
```

Remarks

A type **const_pointer** cannot be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a multimap object.

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::const_reference

A type that provides a reference to a **const** element stored in a multimap for reading and performing **const** operations.

```
typedef typename Allocator::const_reference const_reference;
```

Example

```
// multimap_const_ref.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );

   // Declare and initialize a const_reference &Ref1
   // to the key of the first element
   const int &Ref1 = ( m1.begin( ) -> first );

   // The following line would cause an error because the
   // non-const_reference cannot be used to access the key
   // int &Ref1 = ( m1.begin( ) -> first );

   cout << "The key of the first element in the multimap is "
        << Ref1 << "." << endl;

   // Declare and initialize a reference &Ref2
   // to the data value of the first element
   int &Ref2 = ( m1.begin( ) -> second );

   cout << "The data value of the first element in the multimap is "
        << Ref2 << "." << endl;
}
```

Output

```
The key of the first element in the multimap is 1.
The data value of the first element in the multimap is 10.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::const_reverse_iterator

A type that provides a bidirectional iterator that can read any **const** element in the multimap.

```
typedef reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type **const_reverse_iterator** cannot modify the value of an element and is use to iterate through the multimap in reverse.

The **const_reverse_iterator** defined by multimap points to objects of value_type, which are of type **pair**<**const Key**, **Type**>. The value of the key is available through the first member pair and the value of the mapped element is available through the second member of the pair.

To dereference a **const_reverse_iterator** *crIter* pointing to an element in a multimap, use the **->** operator.

To access the value of the key for the element, use *crIter* -> **first**, which is equivalent to (**crIter*).**first**. To access the value of the mapped datum for the element, use *crIter* -> **second**, which is equivalent to (**crIter*).**first**.

Example

See the example for rend for an example of how to declare and use **const_reverse_iterator**.

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::difference_type

A signed integer type that can be used to represent the number of elements of a multimap in a range between elements pointed to by iterators.

```
typedef implementation-defined difference_type;
```

Remarks

The **difference_type** is the type returned when subtracting or incrementing through iterators of the container. The **difference_type** is typically used to represent the number of elements in the range [*_First*, *_Last*) between the iterators *_First* and *_Last*, includes the element pointed to by *_First* and the range of elements up to, but not including, the element pointed to by *_Last*.

Note that although **difference_type** is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as set, subtraction between iterators is only supported by random-access iterators provided by a random-access container such as vector.

Example

```
// multimap_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <map>
#include <algorithm>

int main( )
{
   using namespace std;
   multimap <int, int> m1;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 3, 20 ) );

   // The following will insert as multimap keys are not unique
   m1.insert ( Int_Pair ( 2, 30 ) );

   multimap <int, int>::iterator m1_Iter, m1_bIter, m1_eIter;
   m1_bIter = m1.begin( );
   m1_eIter = m1.end( );

   // Count the number of elements in a multimap
   multimap <int, int>::difference_type  df_count = 0;
   m1_Iter = m1.begin( );
   while ( m1_Iter != m1_eIter )
   {
      df_count++;
      m1_Iter++;
   }

   cout << "The number of elements in the multimap m1 is: "
        << df_count << "." << endl;
}
```

Output

```
The number of elements in the multimap m1 is: 4.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::iterator

A type that provides a bidirectional iterator that can read or modify any element in a multimap.

```
typedef implementation-defined iterator;
```

Remarks

The **iterator** defined by multimap points to objects of value_type, which are of type **pair**<**const Key**, **Type**>. The value of the key is available through the first member pair and the value of the mapped element is available through the second member of the pair.

To dereference an **iterator** *Iter* pointing to an element in a multimap, use the **->** operator.

To access the value of the key for the element, use *Iter* -> **first**, which is equivalent to (**Iter*).**first**. To access the value of the mapped datum for the element, use *Iter* -> **second**, which is equivalent to (**Iter*).**second**.

A type **iterator** can be used to modify the value of an element.

Example

See the example for begin for an example of how to declare and use **iterator**.

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::key_compare

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the multimap.

```
typedef Traits key_compare;
```

Remarks

key_compare is a synonym for the template parameter **Traits**.

For more information on **Traits** see the multimap Class topic.

Example

See the example for key_comp for an example of how to declare and use **key_compare**.

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::key_type

A type that describes the sort key object that constitutes each element of the multimap.

```
typedef Key key_type;
```

Remarks

key_type is a synonym for the template parameter **Key**.

For more information on **Key**, see the Remarks section of the multimap Class topic.

Example

See the example for value_type for an example of how to declare and use **key_type**.

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::mapped_type

A type that represents the data type stored in a multimap.

```
typedef Type mapped_type;
```

Remarks

mapped_type is a synonym for the template parameter **Type**.

For more information on **Type** see the multimap Class topic.

Example

See the example for value_type for an example of how to declare and use **key_type**.

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::pointer

A type that provides a pointer to an element in a multimap.

```
typedef typename Allocator::pointer pointer;
```

Remarks

A type **pointer** can be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a multimap object.

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::reference

A type that provides a reference to an element stored in a multimap.

```
typedef typename Allocator::reference reference;
```

Example

```
// multimap_ref.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );

   // Declare and initialize a const_reference &Ref1
   // to the key of the first element
   const int &Ref1 = ( m1.begin( ) -> first );

   // The following line would cause an error because the
   // non-const_reference cannot be used to access the key
   // int &Ref1 = ( m1.begin( ) -> first );

   cout << "The key of first element in the multimap is "
        << Ref1 << "." << endl;

   // Declare and initialize a reference &Ref2
   // to the data value of the first element
   int &Ref2 = ( m1.begin( ) -> second );

   cout << "The data value of first element in the multimap is "
        << Ref2 << "." << endl;

   // The non-const_reference can be used to modify the
   // data value of the first element
   Ref2 = Ref2 + 5;
   cout << "The modified data value of first element is "
        << Ref2 << "." << endl;
}
```

Output

```
The key of first element in the multimap is 1.
The data value of first element in the multimap is 10.
The modified data value of first element is 15.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::reverse_iterator

A type that provides a bidirectional iterator that can read or modify an element in a reversed multimap.

```
typedef reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type **reverse_iterator** is use to iterate through the multimap in reverse.

The **reverse_iterator** defined by multimap points to objects of value_type, which are of type **pair**<**const Key**, **Type**>. The value of the key is available through the first member pair and the value of the mapped element is available through the second member of the pair.

To dereference a **reverse_iterator** *rIter* pointing to an element in a multimap, use the -> operator.

To access the value of the key for the element, use *rIter* -> **first**, which is equivalent to (**rIter*).**first**. To access the value of the mapped datum for the element, use *rIter* -> **second**, which is equivalent to (**rIter*).**first**.

Example

See the example for rbegin for an example of how to declare and use **reverse_iterator**.

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::size_type

An unsigned integer type that counts the number of elements in a multimap.

```
typedef implementation-defined size_type;
```

Example

See the example for size for an example of how to declare and use **size_type**

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::value_type

A type that represents the type of object stored as an element in a map.

```
typedef pair<const Key, Type> value_type;
```

Remark

value_type is declared to be **pair** <**const** key_type, mapped_type> and not **pair** <**key_type**, **mapped_type**>, because the keys of an associative container may not be changed using a nonconstant iterator or reference.

Example

```
// multimap_value_type.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   typedef pair <const int, int> cInt2Int;
   multimap <int, int> m1;
   multimap <int, int> :: key_type key1;
   multimap <int, int> :: mapped_type mapped1;
   multimap <int, int> :: value_type value1;
   multimap <int, int> :: iterator pIter;

   // value_type can be used to pass the correct type
   // explicitely to avoid implicit type conversion
   m1.insert ( multimap <int, int> :: value_type ( 1, 10 ) );

   // Compare another way to insert objects into a hash_multimap
   m1.insert ( cInt2Int ( 2, 20 ) );

   // Initializing key1 and mapped1
   key1 = ( m1.begin( ) -> first );
   mapped1 = ( m1.begin( ) -> second );

   cout << "The key of first element in the multimap is "
        << key1 << "." << endl;

   cout << "The data value of first element in the multimap is "
        << mapped1 << "." << endl;

   // The following line would cause an error because
   // the value_type is not assignable
   // value1 = cInt2Int ( 4, 40 );

   cout  << "The keys of the mapped elements are:";
   for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
      cout << " " << pIter -> first;
   cout << "." << endl;

   cout  << "The values of the mapped elements are:";
   for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
      cout << " " << pIter -> second;
   cout << "." << endl;
}
```

Output

```
The key of first element in the multimap is 1.
The data value of first element in the multimap is 10.
```

```
The keys of the mapped elements are: 1 2.
The values of the mapped elements are: 10 20.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap Member Functions

For information about the member functions in the multimap class, see multimap Members.

See Also

Reference

Standard Template Library

multimap::begin

Returns an iterator addressing the first element in the multimap.

```
const_iterator begin( ) const;
iterator begin( );
```

Return Value

A bidirectional iterator addressing the first element in the multimap or the location succeeding an empty multimap.

Remark

If the return value of **begin** is assigned to a **const_iterator**, the elements in the multimap object cannot be modified. If the return value of **begin** is assigned to an **iterator**, the elements in the multimap object can be modified.

Example

```
// multimap_begin.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;

   multimap <int, int> :: iterator m1_Iter;
   multimap <int, int> :: const_iterator m1_cIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 0, 0 ) );
   m1.insert ( Int_Pair ( 1, 1 ) );
   m1.insert ( Int_Pair ( 2, 4 ) );

   m1_cIter = m1.begin ( );
   cout << "The first element of m1 is " << m1_cIter -> first << endl;

   m1_Iter = m1.begin ( );
   m1.erase ( m1_Iter );

   // The following 2 lines would err as the iterator is const
   // m1_cIter = m1.begin ( );
   // m1.erase ( m1_cIter );

   m1_cIter = m1.begin( );
   cout << "First element of m1 is now " << m1_cIter -> first << endl;
}
```

Output

```
The first element of m1 is 0
First element of m1 is now 1
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::clear

Erases all the elements of a multimap.

```
void clear( );
```

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// multimap_clear.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap<int, int> m1;
   multimap<int, int>::size_type i;
   typedef pair<int, int> Int_Pair;

   m1.insert(Int_Pair(1, 1));
   m1.insert(Int_Pair(2, 4));

   i = m1.size();
   cout << "The size of the multimap is initially "
        << i << "." << endl;

   m1.clear();
   i = m1.size();
   cout << "The size of the multimap after clearing is "
        << i << "." << endl;
}
```

Output

```
The size of the multimap is initially 2.
The size of the multimap after clearing is 0.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::count

Returns the number of elements in a multimap whose keys match a parameter-specified key.

```
size_type count(
   const Key& _Key
) const;
```

Parameters

_Key

The key of the elements to be matched from the multimap.

Return Value

The number of elements whose sort keys match the parameter key; 0 if the multimap doesn't contain an element with a matching key.

Remarks

The member function returns the number of elements in the range

[**lower_bound** (*_Key*), **upper_bound** (*_Key*))

that have a key value *_Key*.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// multimap_count.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap<int, int> m1;
    multimap<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    m1.insert(Int_Pair(1, 1));
    m1.insert(Int_Pair(2, 1));
    m1.insert(Int_Pair(1, 4));
    m1.insert(Int_Pair(2, 1));

    // Elements do not need to have unique keys in multimap,
    // so duplicates are allowed and counted
    i = m1.count(1);
    cout << "The number of elements in m1 with a sort key of 1 is: "
         << i << "." << endl;

    i = m1.count(2);
    cout << "The number of elements in m1 with a sort key of 2 is: "
         << i << "." << endl;

    i = m1.count(3);
    cout << "The number of elements in m1 with a sort key of 3 is: "
         << i << "." << endl;
}
```

Output

```
The number of elements in m1 with a sort key of 1 is: 2.
The number of elements in m1 with a sort key of 2 is: 2.
The number of elements in m1 with a sort key of 3 is: 0.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::empty

Tests if a multimap is empty.

```
bool empty( ) const;
```

Return Value

true if the multimap is empty; **false** if the multimap is nonempty.

Example

```
// multimap_empty.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1, m2;

   typedef pair <int, int> Int_Pair;
   m1.insert ( Int_Pair ( 1, 1 ) );

   if ( m1.empty( ) )
      cout << "The multimap m1 is empty." << endl;
   else
      cout << "The multimap m1 is not empty." << endl;

   if ( m2.empty( ) )
      cout << "The multimap m2 is empty." << endl;
   else
      cout << "The multimap m2 is not empty." << endl;
}
```

Output

```
The multimap m1 is not empty.
The multimap m2 is empty.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::end

Returns an iterator that addresses the location succeeding the last element in a multimap.

```
const_iterator end( ) const;
iterator end( );
```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a multimap. If the multimap is empty, then multimap::end == multimap::begin.

Remarks

end is used to test whether an iterator has reached the end of its multimap.

The value returned by **end** should not be dereferenced.

Example

```
// multimap_end.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;

   multimap <int, int> :: iterator m1_Iter;
   multimap <int, int> :: const_iterator m1_cIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_cIter = m1.end( );
   m1_cIter--;
   cout << "The value of last element of m1 is "
        << m1_cIter -> second << endl;

   m1_Iter = m1.end( );
   m1_Iter--;
   m1.erase ( m1_Iter );

   // The following 2 lines would err because the iterator is const
   // m1_cIter = m1.end ( );
   // m1_cIter--;
   // m1.erase ( m1_cIter );

   m1_cIter = m1.end( );
   m1_cIter--;
   cout << "The value of last element of m1 is now "
        << m1_cIter -> second << endl;
}
```

Output

```
The value of last element of m1 is 30
The value of last element of m1 is now 20
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::equal_range

Returns a pair of iterators respectively to the first element in a multimap with a key that is greater than a specified key and to the first element in the multimap with a key that is equal to or greater than the key.

```
pair <const_iterator, const_iterator> equal_range (
   const Key& _Key
) const;
pair <iterator, iterator> equal_range (
   const Key& _Key
);
```

Parameters

_Key

The argument key to be compared with the sort key of an element from the multimap being searched.

Return Value

A pair of iterators such that the first is the lower_bound of the key and the second is the upper_bound of the key.

To access the first iterator of a pair *pr* returned by the member function, use *pr*.**first** and to dereference the lower bound iterator, use *(*pr*.**first**). To access the second iterator of a pair *pr* returned by the member function, use *pr*.**second** and to dereference the upper bound iterator, use *(*pr*.**second**).

Example

```
// multimap_equal_range.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   typedef multimap <int, int, less<int> > IntMMap;
   IntMMap m1;
   multimap <int, int> :: const_iterator m1_RcIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   pair <IntMMap::const_iterator, IntMMap::const_iterator> p1, p2;
   p1 = m1.equal_range( 2 );

   cout << "The lower bound of the element with "
        << "a key of 2 in the multimap m1 is: "
        << p1.first -> second << "." << endl;

   cout << "The upper bound of the element with "
        << "a key of 2 in the multimap m1 is: "
        << p1.second -> second << "." << endl;

   // Compare the upper_bound called directly
   m1_RcIter = m1.upper_bound( 2 );

   cout << "A direct call of upper_bound( 2 ) gives "
        << m1_RcIter -> second << "," << endl
        << " matching the 2nd element of the pair"
        << " returned by equal_range( 2 )." << endl;

   p2 = m1.equal_range( 4 );
```

```
    // If no match is found for the key,
    // both elements of the pair return end( )
    if ( ( p2.first == m1.end( ) ) && ( p2.second == m1.end( ) ) )
       cout << "The multimap m1 doesn't have an element "
            << "with a key less than 4." << endl;
    else
       cout << "The element of multimap m1 with a key >= 40 is: "
            << p1.first -> first << "." << endl;
 }
```

Output

```
The lower bound of the element with a key of 2 in the multimap m1 is: 20.
The upper bound of the element with a key of 2 in the multimap m1 is: 30.
A direct call of upper_bound( 2 ) gives 30,
 matching the 2nd element of the pair returned by equal_range( 2 ).
The multimap m1 doesn't have an element with a key less than 4.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::erase

Removes an element or a range of elements in a multimap from specified positions or removes elements that match a specified key.

```
iterator erase(
   iterator _Where
);
iterator erase(
   iterator _First,
   iterator _Last
);
size_type erase(
   const key_type& _Key
);
```

Parameters

_Where

Position of the element to be removed from the multimap.

_First

Position of the first element removed from the multimap.

_Last

Position just beyond the last element removed from the multimap.

_Key

The key of the elements to be removed from the multimap.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the multimap if no such element exists.

Note
This return type does not conform to the C++ standard.

For the third member function, returns the number of elements that have been removed from the multimap.

Remarks

In some instances, this method might throw an out_of_range exception. This behavior does not conform to the C++ standard.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// multimap_erase.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    multimap<int, int> m1, m2, m3;
    multimap<int, int> :: iterator pIter, Iter1, Iter2;
    int i;
    multimap<int, int>::size_type n;
```

```
    typedef pair<int, int> Int_Pair;

    for (i = 1; i < 5; i++)
    {
        m1.insert(Int_Pair (i, i) );
        m2.insert(Int_Pair (i, i*i) );
        m3.insert(Int_Pair (i, i-1) );
    }

    // The 1st member function removes an element at a given position
    Iter1 = ++m1.begin();
    m1.erase(Iter1);

    cout << "After the 2nd element is deleted, "
         << "the multimap ms1 is:";
    for (pIter = m1.begin(); pIter != m1.end(); pIter++)
        cout << " " << pIter -> second;
    cout << "." << endl;

    // The 2nd member function removes elements
    // in the range [_First, _Last)
    Iter1 = ++m2.begin();
    Iter2 = --m2.end();
    m2.erase(Iter1, Iter2);

    cout << "After the middle two elements are deleted, "
         << "the multimap m2 is:";
    for (pIter = m2.begin(); pIter != m2.end(); pIter++)
        cout << " " << pIter -> second;
    cout << "." << endl;

    // The 3rd member function removes elements with a given _Key
    m3.insert(Int_Pair (2, 5));
    n = m3.erase(2);

    cout << "After the element with a key of 2 is deleted,\n"
         << "the multimap m3 is:";
    for (pIter = m3.begin(); pIter != m3.end(); pIter++)
        cout << " " << pIter -> second;
    cout << "." << endl;

    // The 3rd member function returns the number of elements removed
    cout << "The number of elements removed from m3 is: "
         << n << "." << endl;

    // The dereferenced iterator can also be used to specify a key
    Iter1 = ++m3.begin();
    m3.erase(Iter1);

    cout << "After another element with a key equal to that"
         << endl;
    cout  << "of the 2nd element is deleted, "
          << "the multimap m3 is:";
    for (pIter = m3.begin(); pIter != m3.end(); pIter++)
        cout << " " << pIter -> second;
    cout << "." << endl;
}
```

Output

```
After the 2nd element is deleted, the multimap ms1 is: 1 3 4.
After the middle two elements are deleted, the multimap m2 is: 1 16.
After the element with a key of 2 is deleted,
the multimap m3 is: 0 2 3.
The number of elements removed from m3 is: 2.
After another element with a key equal to that
of the 2nd element is deleted, the multimap m3 is: 0 3.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::find

Returns an iterator addressing the first location of an element in a multimap that has a key equivalent to a specified key.

```
iterator find(
   const Key& _Key
);
const_iterator find(
   const Key& _Key
) const;
```

Parameters

_Key

The key to be matched by the sort key of an element from the multimap being searched.

Return Value

An iterator that addresses the first location of an element with a specified key, or the location succeeding the last element in the multimap if no match is found for the key.

Remarks

The member function returns an iterator that addresses an element in the multimap whose sort key is **equivalent** to the argument key under a binary predicate that induces an ordering based on a less than comparability relation.

If the return value of **find** is assigned to a **const_iterator**, the multimap object cannot be modified. If the return value of **find** is assigned to an **iterator**, the multimap object can be modified.

Example

```
// multimap_find.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;
   multimap <int, int> :: const_iterator m1_AcIter, m1_RcIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_RcIter = m1.find( 2 );
   cout << "The element of multimap m1 with a key of 2 is: "
        << m1_RcIter -> second << "." << endl;

   m1_RcIter = m1.find( 3 );
   cout << "The first element of multimap m1 with a key of 3 is: "
        << m1_RcIter -> second << "." << endl;

   // If no match is found for the key, end( ) is returned
   m1_RcIter = m1.find( 4 );

   if ( m1_RcIter == m1.end( ) )
      cout << "The multimap m1 doesn't have an element "
           << "with a key of 4." << endl;
   else
      cout << "The element of multimap m1 with a key of 4 is: "
           << m1_RcIter -> second << "." << endl;
```

```
    // The element at a specific location in the multimap can be
    // found using a dereferenced iterator addressing the location
    m1_AcIter = m1.end( );
    m1_AcIter--;
    m1_RcIter = m1.find( m1_AcIter -> first );
    cout << "The first element of m1 with a key matching"
         << endl << "that of the last element is: "
         << m1_RcIter -> second << "." << endl;

    // Note that the first element with a key equal to
    // the key of the last element is not the last element
    if ( m1_RcIter == --m1.end( ) )
       cout << "This is the last element of multimap m1."
            << endl;
    else
       cout << "This is not the last element of multimap m1."
            << endl;
}
```

Output

```
The element of multimap m1 with a key of 2 is: 20.
The first element of multimap m1 with a key of 3 is: 20.
The multimap m1 doesn't have an element with a key of 4.
The first element of m1 with a key matching
that of the last element is: 20.
This is not the last element of multimap m1.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::get_allocator

Returns a copy of the allocator object used to construct the multimap.

```
Allocator get_allocator( ) const;
```

Return Value

The allocator used by the multimap.

Remarks

Allocators for the multimap class specify how the class manages storage. The default allocators supplied with STL container classes is sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```
// multimap_get_allocator.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int>::allocator_type m1_Alloc;
   multimap <int, int>::allocator_type m2_Alloc;
   multimap <int, double>::allocator_type m3_Alloc;
   multimap <int, int>::allocator_type m4_Alloc;

   // The following lines declare objects
   // that use the default allocator.
   multimap <int, int> m1;
   multimap <int, int, allocator<int> > m2;
   multimap <int, double, allocator<double> > m3;

   m1_Alloc = m1.get_allocator( );
   m2_Alloc = m2.get_allocator( );
   m3_Alloc = m3.get_allocator( );

   cout << "The number of integers that can be allocated"
        << endl << "before free memory is exhausted: "
        << m2.max_size( ) << ".\n" << endl;

   cout << "The number of doubles that can be allocated"
        << endl << "before free memory is exhausted: "
        << m3.max_size( ) <<  ".\n" << endl;

   // The following line creates a multimap m4
   // with the allocator of multimap m1.
   map <int, int> m4( less<int>( ), m1_Alloc );

   m4_Alloc = m4.get_allocator( );

   // Two allocators are interchangeable if
   // storage allocated from each can be
   // deallocated via the other
   if( m1_Alloc == m4_Alloc )
   {
      cout << "The allocators are interchangeable."
           << endl;
   }
   else
   {
      cout << "The allocators are not interchangeable."
```

```
            << endl;
    }
}
```

Sample Output

The following output is for x86.

```
The number of integers that can be allocated
before free memory is exhausted: 536870911.

The number of doubles that can be allocated
before free memory is exhausted: 268435455.

The allocators are interchangeable.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::insert

Inserts an element or a range of elements into a multimap.

```
iterator insert(
   const value_type& _Val
);
iterator insert(
   iterator _Where,
   const value_type& _Val
);
template<class InputIterator>
   void insert(
      InputIterator _First,
      InputIterator _Last
   );
```

Parameters

_Val

The value of an element to be inserted into the multimap unless the multimap already contains that element or, more generally, an element whose key is equivalently ordered.

_Where

A hint regarding the place to start searching for the correct point of insertion.

_First

The position of the first element to be copied from a map.

_Last

The position just beyond the last element to be copied from a map.

Return Value

The **insert** member functions returns an iterator that points to the position where the new element was inserted into the multimap.

Remarks

The value_type of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Insertion can occur in amortized constant time for the hint version of insert, instead of logarithmic time, if the insertion point immediately follows *_Where*.

The third member function inserts the sequence of element values into a map corresponding to each element addressed by an iterator in the range *[_First, _Last)* of a specified set.

Example

```
// multimap_insert.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int>::iterator m1_pIter, m2_pIter;

   multimap <int, int> m1, m2;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
```

```
    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 3, 30 ) );

    cout << "The original key values of m1 =";
    for ( m1_pIter = m1.begin( ); m1_pIter != m1.end( ); m1_pIter++ )
       cout << " " << m1_pIter -> first;
    cout << "." << endl;

    cout << "The original mapped values of m1 =";
    for ( m1_pIter = m1.begin( ); m1_pIter != m1.end( ); m1_pIter++ )
       cout << " " << m1_pIter -> second;
    cout << "." << endl;

    m1.insert ( Int_Pair ( 1, 10 ) );

    // The hint version of insert
    m1.insert( --m1.end( ), Int_Pair ( 4, 40 )  );

    cout << "After the insertions, the key values of m1 =";
    for ( m1_pIter = m1.begin( ); m1_pIter != m1.end( ); m1_pIter++ )
       cout << " " << m1_pIter -> first;
    cout << "," << endl;

    cout << " and the mapped values of m1 =";
    for ( m1_pIter = m1.begin( ); m1_pIter != m1.end( ); m1_pIter++ )
       cout << " " << m1_pIter -> second;
    cout << "." << endl;

    m2.insert ( Int_Pair ( 10, 100 ) );

    // The templatized version inserting a range
    m2.insert( ++m1.begin( ), --m1.end( ) );

    cout << "After the insertions, the key values of m2 =";
    for ( m2_pIter = m2.begin( ); m2_pIter != m2.end( ); m2_pIter++ )
       cout << " " << m2_pIter -> first;
    cout << "," << endl;

    cout << " and the mapped values of m2 =";
    for ( m2_pIter = m2.begin( ); m2_pIter != m2.end( ); m2_pIter++ )
       cout << " " << m2_pIter -> second;
    cout << "." << endl;
 }
```

Output

```
The original key values of m1 = 1 2 3.
The original mapped values of m1 = 10 20 30.
After the insertions, the key values of m1 = 1 1 2 3 4,
 and the mapped values of m1 = 10 10 20 30 40.
After the insertions, the key values of m2 = 1 2 3 10,
 and the mapped values of m2 = 10 20 30 100.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::key_comp

Retrieves a copy of the comparison object used to order keys in a multimap.

```
key_compare key_comp( ) const;
```

Return Value

Returns the function object that a multimap uses to order its elements.

Remarks

The stored object defines the member function

bool operator(**const Key&** *x*, **const Key&** *y*);

which returns true if *x* strictly precedes *y* in the sort order.

Example

```
// multimap_key_comp.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;

   multimap <int, int, less<int> > m1;
   multimap <int, int, less<int> >::key_compare kc1 = m1.key_comp( ) ;
   bool result1 = kc1( 2, 3 ) ;
   if( result1 == true )
   {
      cout << "kc1( 2,3 ) returns value of true, "
           << "where kc1 is the function object of m1."
           << endl;
   }
   else
   {
      cout << "kc1( 2,3 ) returns value of false "
           << "where kc1 is the function object of m1."
           << endl;
   }

   multimap <int, int, greater<int> > m2;
   multimap <int, int, greater<int> >::key_compare kc2 = m2.key_comp( );
   bool result2 = kc2( 2, 3 ) ;
   if( result2 == true )
   {
      cout << "kc2( 2,3 ) returns value of true, "
           << "where kc2 is the function object of m2."
           << endl;
   }
   else
   {
      cout << "kc2( 2,3 ) returns value of false, "
           << "where kc2 is the function object of m2."
           << endl;
   }
}
```

Output

```
kc1( 2,3 ) returns value of true, where kc1 is the function object of m1.
```

```
kc2( 2,3 ) returns value of false, where kc2 is the function object of m2.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::lower_bound

Returns an iterator to the first element in a multimap that with a key that is equal to or greater than a specified key.

```
iterator lower_bound(
   const Key& _Key
);
const_iterator lower_bound(
   const Key& _Key
) const;
```

Parameters

_Key

The argument key to be compared with the sort key of an element from the multimap being searched.

Return Value

An iterator or **const_iterator** that addresses the location of an element in a multimap that with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the multimap if no match is found for the key.

If the return value of **lower_bound** is assigned to a **const_iterator**, the multimap object cannot be modified. If the return value of **lower_bound** is assigned to an **iterator**, the multimap object can be modified.

Example

```
// multimap_lower_bound.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;
   multimap <int, int> :: const_iterator m1_AcIter, m1_RcIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_RcIter = m1.lower_bound( 2 );
   cout << "The element of multimap m1 with a key of 2 is: "
        << m1_RcIter -> second << "." << endl;

   m1_RcIter = m1.lower_bound( 3 );
   cout << "The first element of multimap m1 with a key of 3 is: "
        << m1_RcIter -> second << "." << endl;

   // If no match is found for the key, end( ) is returned
   m1_RcIter = m1.lower_bound( 4 );

   if ( m1_RcIter == m1.end( ) )
      cout << "The multimap m1 doesn't have an element "
              << "with a key of 4." << endl;
   else
      cout << "The element of multimap m1 with a key of 4 is: "
                << m1_RcIter -> second << "." << endl;

   // The element at a specific location in the multimap can be
   // found using a dereferenced iterator addressing the location
   m1_AcIter = m1.end( );
```

```
    m1_AcIter--;
    m1_RcIter = m1.lower_bound( m1_AcIter -> first );
    cout << "The first element of m1 with a key matching\n"
         << "that of the last element is: "
         << m1_RcIter -> second << "." << endl;

    // Note that the first element with a key equal to
    // the key of the last element is not the last element
    if ( m1_RcIter == --m1.end( ) )
       cout << "This is the last element of multimap m1."
            << endl;
    else
       cout << "This is not the last element of multimap m1."
            << endl;
 }
```

Output

```
The element of multimap m1 with a key of 2 is: 20.
The first element of multimap m1 with a key of 3 is: 20.
The multimap m1 doesn't have an element with a key of 4.
The first element of m1 with a key matching
that of the last element is: 20.
This is not the last element of multimap m1.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::max_size

Returns the maximum length of the multimap.

```
size_type max_size( ) const;
```

Return Value

The maximum possible length of the multimap.

Example

```
// multimap_max_size.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;
   multimap <int, int> :: size_type i;

   i = m1.max_size( );
   cout << "The maximum possible length "
        << "of the multimap is " << i << "." << endl;
}
```

Sample Output

The following output is for x86.

```
The maximum possible length of the multimap is 536870911.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::multimap

Constructs a multimap that is empty or that is a copy of all or part of some other multimap.

```
multimap( );
explicit multimap(
   const Traits& _Comp
);
explicit multimap(
   const Traits& _Comp,
   const Allocator& _Al
);
map(
   const _ multimap& _Right
);
template<class InputIterator>
   multimap(
      InputIterator _First,
      InputIterator _Last
   );
template<class InputIterator>
   multimap(
      InputIterator _First,
      InputIterator _Last,
      const Traits& _Comp
   );
template<class InputIterator>
   multimap(
      InputIterator _First,
      InputIterator _Last,
      const Traits& _Comp,
      const Allocator& _Al
   );
```

Parameters

_Al

The storage allocator class to be used for this multimap object, which defaults to Allocator.

_Comp

The comparison function of type **const Traits** used to order the elements in the map, which defaults to **Traits**.

_Right

The map of which the constructed set is to be a copy.

_First

The position of the first element in the range of elements to be copied.

_Last

The position of the first element beyond the range of elements to be copied.

Remarks

All constructors store a type of allocator object that manages memory storage for the multimap and that can later be returned by calling get_allocator. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their multimap.

All constructors store a function object of type **Traits** that is used to establish an order among the keys of the multimap and that can later be returned by calling key_comp.

The first three constructors specify an empty initial multimap, the second specifying the type of comparison function (*_Comp*)

to be used in establishing the order of the elements and the third explicitly specifying the allocator type (*_Al*) to be used. The key word **explicit** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the multimap *_Right*.

The last three constructors copy the range [*_First, _Last*) of a map with increasing explicitness in specifying the type of comparison function of class **Traits** and allocator.

Example

```
// multimap_ctor.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   typedef pair <int, int> Int_Pair;
   multimap <int, int>::iterator m1_Iter, m3_Iter, m4_Iter, m5_Iter, m6_Iter;
   multimap <int, int, greater<int> >::iterator m2_Iter;

   // Create an empty multimap m0 of key type integer
   multimap <int, int> m0;

   // Create an empty multimap m1 with the key comparison
   // function of less than, then insert 4 elements
   multimap <int, int, less<int> > m1;
   m1.insert( Int_Pair( 1, 10 ) );
   m1.insert( Int_Pair( 2, 20 ) );
   m1.insert( Int_Pair( 3, 30 ) );
   m1.insert( Int_Pair( 4, 40 ) );

   // Create an empty multimap m2 with the key comparison
   // function of geater than, then insert 2 elements
   multimap <int, int, greater<int> > m2;
   m2.insert( Int_Pair( 1, 10 ) );
   m2.insert( Int_Pair( 2, 20 ) );

   // Create a multimap m3 with the
   // allocator of multimap m1
   multimap <int, int>::allocator_type m1_Alloc;
   m1_Alloc = m1.get_allocator( );
   multimap <int, int> m3( less<int>( ), m1_Alloc );
   m3.insert( Int_Pair( 3, 30 ) );

   // Create a copy, multimap m4, of multimap m1
   multimap <int, int> m4( m1 );

   // Create a multimap m5 by copying the range m1[_First, _Last)
   multimap <int, int>::const_iterator m1_bcIter, m1_ecIter;
   m1_bcIter = m1.begin( );
   m1_ecIter = m1.begin( );
   m1_ecIter++;
   m1_ecIter++;
   multimap <int, int> m5( m1_bcIter, m1_ecIter );

   // Create a multimap m6 by copying the range m4[_First, _Last)
   // and with the allocator of multimap m2
   multimap <int, int>::allocator_type m2_Alloc;
   m2_Alloc = m2.get_allocator( );
   multimap <int, int> m6(m4.begin( ), ++m4.begin( ), less<int>( ), m2_Alloc);

   cout << "m1 =";
   for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
      cout << " " << m1_Iter -> second;
   cout << endl;

   cout << "m2 =";
```

```
    for ( m2_Iter = m2.begin( ); m2_Iter != m2.end( ); m2_Iter++ )
       cout << " " << m2_Iter -> second;
    cout << endl;

    cout << "m3 =";
    for ( m3_Iter = m3.begin( ); m3_Iter != m3.end( ); m3_Iter++ )
       cout << " " << m3_Iter -> second;
    cout << endl;

    cout << "m4 =";
    for ( m4_Iter = m4.begin( ); m4_Iter != m4.end( ); m4_Iter++ )
       cout << " " << m4_Iter -> second;
    cout << endl;

    cout << "m5 =";
    for ( m5_Iter = m5.begin( ); m5_Iter != m5.end( ); m5_Iter++ )
       cout << " " << m5_Iter -> second;
    cout << endl;

    cout << "m6 =";
    for ( m6_Iter = m6.begin( ); m6_Iter != m6.end( ); m6_Iter++ )
       cout << " " << m6_Iter -> second;
    cout << endl;
 }
```

Output

```
m1 = 10 20 30 40
m2 = 20 10
m3 = 30
m4 = 10 20 30 40
m5 = 10 20
m6 = 10
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::rbegin

Returns an iterator addressing the first element in a reversed multimap.

```
const_reverse_iterator rbegin( ) const;
reverse_iterator rbegin( );
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed multimap or addressing what had been the last element in the unreversed multimap.

Remarks

rbegin is used with a reversed multimap just as begin is used with a multimap.

If the return value of **rbegin** is assigned to a **const_reverse_iterator**, then the multimap object cannot be modified. If the return value of **rbegin** is assigned to a **reverse_iterator**, then the multimap object can be modified.

rbegin can be used to iterate through a multimap backwards.

Example

```
// multimap_rbegin.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;

   multimap <int, int> :: iterator m1_Iter;
   multimap <int, int> :: reverse_iterator m1_rIter;
   multimap <int, int> :: const_reverse_iterator m1_crIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_rIter = m1.rbegin( );
   cout << "The first element of the reversed multimap m1 is "
        << m1_rIter -> first << "." << endl;

   // begin can be used to start an iteration
   // throught a multimap in a forward order
   cout << "The multimap is: ";
   for ( m1_Iter = m1.begin( ) ; m1_Iter != m1.end( ); m1_Iter++)
      cout << m1_Iter -> first << " ";
      cout << "." << endl;

   // rbegin can be used to start an iteration
   // throught a multimap in a reverse order
   cout << "The reversed multimap is: ";
   for ( m1_rIter = m1.rbegin( ) ; m1_rIter != m1.rend( ); m1_rIter++)
      cout << m1_rIter -> first << " ";
      cout << "." << endl;

   // A multimap element can be erased by dereferencing its key
   m1_rIter = m1.rbegin( );
   m1.erase ( m1_rIter -> first );

   m1_rIter = m1.rbegin( );
```

```
    cout << "After the erasure, the first element "
         << "in the reversed multimap is "
         << m1_rIter -> first << "." << endl;
}
```

Output

```
The first element of the reversed multimap m1 is 3.
The multimap is: 1 2 3 .
The reversed multimap is: 3 2 1 .
After the erasure, the first element in the reversed multimap is 2.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::rend

Returns an iterator that addresses the location succeeding the last element in a reversed multimap.

```
const_reverse_iterator rend( ) const;
reverse_iterator rend( );
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed multimap (the location that had preceded the first element in the unreversed multimap).

Remarks

rend is used with a reversed multimap just as end is used with a multimap.

If the return value of **rend** is assigned to a **const_reverse_iterator**, then the multimap object cannot be modified. If the return value of **rend** is assigned to a **reverse_iterator**, then the multimap object can be modified.

rend can be used to test to whether a reverse iterator has reached the end of its multimap.

The value returned by **rend** should not be dereferenced.

Example

```
// multimap_rend.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;

   multimap <int, int> :: iterator m1_Iter;
   multimap <int, int> :: reverse_iterator m1_rIter;
   multimap <int, int> :: const_reverse_iterator m1_crIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_rIter = m1.rend( );
   m1_rIter--;
   cout << "The last element of the reversed multimap m1 is "
        << m1_rIter -> first << "." << endl;

   // begin can be used to start an iteration
   // throught a multimap in a forward order
   cout << "The multimap is: ";
   for ( m1_Iter = m1.begin( ) ; m1_Iter != m1.end( ); m1_Iter++)
      cout << m1_Iter -> first << " ";
      cout << "." << endl;

   // rbegin can be used to start an iteration
   // throught a multimap in a reverse order
   cout << "The reversed multimap is: ";
   for ( m1_rIter = m1.rbegin( ) ; m1_rIter != m1.rend( ); m1_rIter++)
      cout << m1_rIter -> first << " ";
      cout << "." << endl;

   // A multimap element can be erased by dereferencing to its key
   m1_rIter = --m1.rend( );
   m1.erase ( m1_rIter -> first );
```

```
    m1_rIter = m1.rend( );
    m1_rIter--;
    cout << "After the erasure, the last element "
         << "in the reversed multimap is "
         << m1_rIter -> first << "." << endl;
}
```

Output

```
The last element of the reversed multimap m1 is 1.
The multimap is: 1 2 3 .
The reversed multimap is: 3 2 1 .
After the erasure, the last element in the reversed multimap is 2.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::size

Returns the number of elements in the multimap.

```
size_type size( ) const;
```

Return Value

The current length of the multimap.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// multimap_size.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    multimap<int, int> m1, m2;
    multimap<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    m1.insert(Int_Pair(1, 1));
    i = m1.size();
    cout << "The multimap length is " << i << "." << endl;

    m1.insert(Int_Pair(2, 4));
    i = m1.size();
    cout << "The multimap length is now " << i << "." << endl;
}
```

Output

```
The multimap length is 1.
The multimap length is now 2.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::swap

Exchanges the elements of two multimaps.

```
void swap(
   multimap& _Right
);
```

Parameters

_Right

The multimap providing the elements to be swapped, or the multimap whose elements are to be exchanged with those of the multimap *_Left*.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two multimaps whose elements are being exchanged.

Example

```
// multimap_swap.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1, m2, m3;
   multimap <int, int>::iterator m1_Iter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );
   m2.insert ( Int_Pair ( 10, 100 ) );
   m2.insert ( Int_Pair ( 20, 200 ) );
   m3.insert ( Int_Pair ( 30, 300 ) );

   cout << "The original multimap m1 is:";
   for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
      cout << " " << m1_Iter -> second;
   cout   << "." << endl;

   // This is the member function version of swap
   m1.swap( m2 );

   cout << "After swapping with m2, multimap m1 is:";
   for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
      cout << " " << m1_Iter -> second;
   cout  << "." << endl;

   // This is the specialized template version of swap
   swap( m1, m3 );

   cout << "After swapping with m3, multimap m1 is:";
   for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
      cout << " " << m1_Iter -> second;
   cout   << "." << endl;
}
```

Output

```
The original multimap m1 is: 10 20 30.
After swapping with m2, multimap m1 is: 100 200.
After swapping with m3, multimap m1 is: 300.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::upper_bound

Returns an iterator to the first element in a multimap that with a key that is greater than a specified key.

```
iterator upper_bound(
   const Key& _Key
);
const_iterator upper_bound(
   const Key& _Key
) const;
```

Parameters

_Key

The argument key to be compared with the sort key of an element from the multimap being searched.

Return Value

An iterator or **const_iterator** that addresses the location of an element in a multimap that with a key that is greater than the argument key, or that addresses the location succeeding the last element in the multimap if no match is found for the key.

If the return value is assigned to a **const_iterator**, the multimap object cannot be modified. If the return value is assigned to a **iterator**, the multimap object can be modified.

Example

```
// multimap_upper_bound.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;
   multimap <int, int> :: const_iterator m1_AcIter, m1_RcIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );
   m1.insert ( Int_Pair ( 3, 40 ) );

   m1_RcIter = m1.upper_bound( 1 );
   cout << "The 1st element of multimap m1 with "
        << "a key greater than 1 is: "
        << m1_RcIter -> second << "." << endl;

   m1_RcIter = m1.upper_bound( 2 );
   cout << "The first element of multimap m1 with a key "
        << " greater than 2 is: "
        << m1_RcIter -> second << "." << endl;

   // If no match is found for the key, end( ) is returned
   m1_RcIter = m1.lower_bound( 4 );

   if ( m1_RcIter == m1.end( ) )
      cout << "The multimap m1 doesn't have an element "
           << "with a key of 4." << endl;
   else
      cout << "The element of multimap m1 with a key of 4 is: "
           << m1_RcIter -> second << "." << endl;

   // The element at a specific location in the multimap can be
   // found using a derefenced iterator addressing the location
```

```
    m1_AcIter = m1.begin( );
    m1_RcIter = m1.upper_bound( m1_AcIter -> first );
    cout << "The first element of m1 with a key greater than\n"
         << "that of the initial element of m1 is: "
         << m1_RcIter -> second << "." << endl;
}
```

Output

```
The 1st element of multimap m1 with a key greater than 1 is: 20.
The first element of multimap m1 with a key  greater than 2 is: 30.
The multimap m1 doesn't have an element with a key of 4.
The first element of m1 with a key greater than
that of the initial element of m1 is: 20.
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

multimap::value_comp

The member function returns a function object that determines the order of elements in a multimap by comparing their key values.

```
value_compare value_comp( ) const;
```

Return Value

Returns the comparison function object that a multimap uses to order its elements.

Remarks

For a multimap *m*, if two elements *e*1(*k*1, *d*1) and *e*2(*k*2, *d*2) are objects of type **value_type**, where *k*1 and *k*2 are their keys of type **key_type** and *d*1 and *d*2 are their data of type **mapped_type**, then *m*.**value_comp**(*e*1, *e*2) is equivalent to *m*.**key_comp**(*k*1, *k*2).

Example

```
// multimap_value_comp.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;

   multimap <int, int, less<int> > m1;
   multimap <int, int, less<int> >::value_compare vc1 = m1.value_comp( );
   multimap<int,int>::iterator Iter1, Iter2;

   Iter1= m1.insert ( multimap <int, int> :: value_type ( 1, 10 ) );
   Iter2= m1.insert ( multimap <int, int> :: value_type ( 2, 5 ) );

   if( vc1( *Iter1, *Iter2 ) == true )
   {
      cout << "The element ( 1,10 ) precedes the element ( 2,5 )."
           << endl;
   }
   else
   {
      cout << "The element ( 1,10 ) does "
           << "not precede the element ( 2,5 )."
           << endl;
   }

   if( vc1( *Iter2, *Iter1 ) == true )
   {
      cout << "The element ( 2,5 ) precedes the element ( 1,10 )."
           << endl;
   }
   else
   {
      cout << "The element ( 2,5 ) does "
           << "not precede the element ( 1,10 )."
           << endl;
   }
}
```

Output

```
The element ( 1,10 ) precedes the element ( 2,5 ).
The element ( 2,5 ) does not precede the element ( 1,10 ).
```

Requirements

Header: <map>

See Also

Reference

multimap Class

Standard Template Library

Other Resources

multimap Members

<memory>

Defines a class, an operator, and several templates that help allocate and free objects.

For a list of all members of this header, see <memory> Members.

```
#include <memory>
```

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<memory> Members

Standard C++ Library Header Files

<memory> Members

Reference

Functions

checked_uninitialized_fill_n	Same as **uninitialized_fill_n** but enforces the use of a checked iterator as output iterator.
get_temporary_buffer	Allocates temporary storage for a sequence of elements that does not exceed a specified number of elements.
return_temporary_buffer	Deallocates the temporary memory that was allocated using the **get_temporary_buffer** template function.
unchecked_uninitialized_fill_n	Same as **uninitialized_fill_n** but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.
uninitialized_copy	Copies objects from a specified input range into an uninitialized destination range.
uninitialized_fill	Copies objects of a specified value into an uninitialized destination range.
uninitialized_fill_n	Copies objects of a specified value into specified number of elements an uninitialized destination range.

Operators

operator!=	Tests for inequality between allocator objects of a specified class.
operator==	Tests for equality between allocator objects of a specified class.

Classes

allocator Class	The template class describes an object that manages storage allocation and freeing for arrays of objects of type **Type**.
auto_ptr Class	The template class describes an object that stores a pointer to an allocated object of type **Type** * that ensures the object to which it points gets deleted when its enclosing auto_ptr gets destroyed.
raw_storage_iterator Class	An adaptor class that is provided to enable algorithms to store their results into uninitialized memory.

Specializations

allocator<void> Class	A specialization of the template class allocator to type void, defining the only the member types that make sense in this specialized context.

See Also

Reference

<memory>

Thread Safety in the Standard C++ Library

<memory> Operators

For information about the operators in <memory>, see <memory> Members.

operator!= (<memory>)

Tests for inequality between allocator objects of a specified class.

```
template<class Type>
   bool operator!=(
      allocator<Type>& _Left,
      allocator<Type>& _Right
   );
```

Parameters

_Left

One of the allocator objects to be tested for inequality.

_Right

One of the allocator objects to be tested for inequality.

Return Value

true if the allocator objects are not equal; **false** if allocator objects are equal.

Remarks

Two allocator objects should compare equal only if an object allocated through one can be deallocated through the other. If the value of one object is determined from another by assignment or by construction, the two object should compare equal.

Example

```
// memory_op_me.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   allocator<double> Alloc;
   vector <char>:: allocator_type v1Alloc;

   if ( Alloc != v1Alloc )
      cout << "The allocator objects Alloc & v1Alloc not are equal."
           << endl;
   else
      cout << "The allocator objects Alloc & v1Alloc are equal."
           << endl;
}
```

Output

```
The allocator objects Alloc & v1Alloc are equal.
```

Requirements

Header: <memory>

See Also

Reference

auto_ptr Class

Other Resources

auto_ptr Members

operator== (<memory>)

Tests for equality between allocator objects of a specified class.

```
template<class Type>
   bool operator==(
      allocator<Type>& _Left,
      allocator<Type>& _Right
   );
```

Parameters

_Left

One of the allocator objects to be tested for equality.

_Right

One of the allocator objects to be tested for equality.

Return Value

true if the allocator objects are equal; **false** if allocator objects are not equal.

Remarks

Two allocator objects should compare equal only if an object allocated through one can be deallocated through the other. If the value of one object is determined from another by assignment or by construction, the two objects should compare equal.

Example

```
// memory_op_eq.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   allocator<char> Alloc;
   vector <int>:: allocator_type v1Alloc;

   allocator<char> cAlloc(Alloc);
   allocator<int> cv1Alloc(v1Alloc);

   if ( cv1Alloc == v1Alloc )
      cout << "The allocator objects cv1Alloc & v1Alloc are equal."
           << endl;
   else
      cout << "The allocator objects cv1Alloc & v1Alloc are not equal."
           << endl;

   if ( cAlloc == Alloc )
      cout << "The allocator objects cAlloc & Alloc are equal."
           << endl;
   else
      cout << "The allocator objects cAlloc & Alloc are not equal."
           << endl;
}
```

Output

```
The allocator objects cv1Alloc & v1Alloc are equal.
The allocator objects cAlloc & Alloc are equal.
```

Requirements

Header: <memory>

See Also

Reference

auto_ptr Class

Other Resources

auto_ptr Members

<memory> Functions

For information about the functions in <memory>, see <memory> Members.

checked_uninitialized_copy

Same as uninitialized_copy but enforces the use of a checked iterator as output iterator. This function is defined in the The stdext Namespace namespace.

Note

This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator, class ForwardIterator>
   ForwardIterator unchecked_uninitialized_copy(
      InputIterator _First,
      InputIterator _Last,
      ForwardIterator _Dest
   );

template<class InputIterator, class ForwardIterator, class Allocator>
   ForwardIterator unchecked_uninitialized_copy(
      InputIterator _First,
      InputIterator _Last,
      ForwardIterator _Dest,
      Allocator& _Al
```

Parameters

_First

An input iterator addressing the first element in the source range to be copied.

_Last

An input iterator addressing the last element in the source range to be copied.

_Dest

A forward iterator addressing the first element in the destination range to be copied.

_Al

The allocator class to use with this object. vector::get_allocator returns the allocator class for the object.

Return Value

A forward iterator addressing the position one past the final element in the destination range that is receiving the copy.

Remarks

See uninitialized_copy for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <memory>

See Also

Other Resources

<memory> Members

checked_uninitialized_fill_n

Same as uninitialized_fill_n but enforces the use of a checked iterator as output iterator (*_First*). **checked_uninitialized_fill_n** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class FwdIt, class Size, class Type>
   void checked_uninitialized_fill_n(
      ForwardIterator _First,
      Size _Count,
      const Type& _Val
   );
template<class FwdIt, class Size, class Type, class Allocator>
   void checked_uninitialized_fill_n(
      ForwardIterator _First,
      Size _Count,
      const Type& _Val,
      Allocator& _Al
   );
```

Parameters

_Al

The allocator class to use with this object. vector::get_allocator returns the allocator class for the object.

_First

A forward iterator addressing the first element in the destination range to be initiated.

_Count

The number of elements to be initialized.

_Val

The value to be used to initialize the destination range.

Remarks

See **uninitialized_fill_n** for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <memory>

See Also

Other Resources

<memory> Members

get_temporary_buffer

Allocates temporary storage for a sequence of elements that does not exceed a specified number of elements.

```
template<class Type>
   pair<Type *, ptrdiff_t>
      get_temporary_buffer(
         ptrdiff_t _Count
      );
```

Parameters

_Count

The maximum number of elements requested for which memory is to be allocated.

Return Value

A **pair** whose first component is a pointer to the memory that was allocated, and whose second component gives the size of the buffer, indicating the largest number of elements that it could store.

Remarks

The function makes a request for memory and it may not succeed. If no buffer is allocated, then the function returns a pair, with the second component equal to zero and the first component equal to the null pointer.

This function should only be used for memory that is temporary.

Example

```
// memory_get_temp_buf.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

using namespace std;

int main( )
{
   // Create an array of ints
   int intArray [ ] = { 10, 20, 30, 40, 100, 200, 300, 1000, 2000 };
   int count = sizeof ( intArray ) / sizeof ( int );
   cout << "The number of integers in the array is: "
      << count << "." << endl;

   pair<int *, ptrdiff_t> resultPair;
   resultPair = get_temporary_buffer<int>( count );

   cout << "The number of elements that the allocated memory\n"
        << "could store is given by: resultPair.second = "
        << resultPair.second << "." << endl;
}
```

Output

```
The number of integers in the array is: 9.
The number of elements that the allocated memory
could store is given by: resultPair.second = 9.
```

Requirements

Header: <memory>

See Also

Other Resources

<memory> Members

return_temporary_buffer

Deallocates the temporary memory that was allocated using the **get_temporary_buffer** template function.

```
template<class Type>
   void return_temporary_buffer(
      Type* _Pbuf
   );
```

Parameters

_Pbuf

A pointer to the memory to be deallocated.

Remarks

This function should only be used for memory that is temporary.

Example

```
// memory_ret_temp_buf.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

using namespace std;

int main( )
{
   // Create an array of ints
   int intArray [ ] = { 10, 20, 30, 40, 100, 200, 300 };
   int count = sizeof ( intArray ) / sizeof ( int );
   cout << "The number of integers in the array is: "
         << count << "." << endl;

   pair<int *, ptrdiff_t> resultPair;
   resultPair = get_temporary_buffer<int>( count );

   cout << "The number of elements that the allocated memory\n"
         << " could store is given by: resultPair.second = "
         << resultPair.second << "." << endl;

   int* tempBuffer = resultPair.first;

   // Deallocates memory allocated with get_temporary_buffer
   return_temporary_buffer ( tempBuffer );
}
```

Output

```
The number of integers in the array is: 7.
The number of elements that the allocated memory
 could store is given by: resultPair.second = 7.
```

Requirements

Header: <memory>

See Also

Other Resources

<memory> Members

unchecked_uninitialized_copy

Same as uninitialized_copy but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined. This function is defined in the The stdext Namespace namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator, class ForwardIterator>
   ForwardIterator unchecked_uninitialized_copy(
      InputIterator _First,
      InputIterator _Last,
      ForwardIterator _Dest
   );

template<class InputIterator, class ForwardIterator, class Allocator>
   ForwardIterator unchecked_uninitialized_copy(
      InputIterator _First,
      InputIterator _Last,
      ForwardIterator _Dest,
      Allocator& _Al
   );
```

Parameters

_First

An input iterator addressing the first element in the source range to be copied.

_Last

An input iterator addressing the last element in the source range to be copied.

_Dest

A forward iterator addressing the first element in the destination range to be copied.

_Al

The allocator class to use with this object. vector::get_allocator returns the allocator class for the object.

Return Value

A forward iterator addressing the position one past the final element in the destination range that is receiving the copy.

Remarks

See uninitialized_copy for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <memory>

See Also

Other Resources

<memory> Members

unchecked_uninitialized_fill_n

Same as uninitialized_fill_n but allows the use of an unchecked iterator as output iterator (*_First*) when _SECURE_SCL=1 is defined. **unchecked_uninitialized_fill_n** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class ForwardIterator, class Size, class Type>
   void unchecked_uninitialized_fill_n(
      ForwardIterator _First,
      Size _Count,
      const Type& _Val
   );

template<class ForwardIterator, class Size, class Type, class Allocator>
   void unchecked_uninitialized_fill_n(
      ForwardIterator _First,
      Size _Count,
      const Type& _Val,
      Allocator& _Al
   );
```

Parameters

_Al

The allocator class to use with this object. vector::get_allocator returns the allocator class for the object.

_First

A forward iterator addressing the first element in the destination range to be initiated.

_Count

The number of elements to be initialized.

_Val

The value to be used to initialize the destination range.

Remarks

See uninitialized_fill_n for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <memory>

See Also

Other Resources

<memory> Members

uninitialized_copy

Copies objects from a specified source range into an uninitialized destination range.

```
template<class InputIterator, class ForwardIterator>
   ForwardIterator uninitialized_copy(
      InputIterator _First,
      InputIterator _Last,
      ForwardIterator _Dest
   );
```

Parameters

_First

An input iterator addressing the first element in the source range.

_Last

An input iterator addressing the last element in the source range.

_Dest

A forward iterator addressing the first element in the destination range.

Return Value

A forward iterator addressing the first position beyond the destination range, unless the source range was empty and iterator addresses *_First*.

Remarks

This algorithm allows the decoupling of memory allocation from object construction.

The template function effectively executes:

```
while ( _First!= _Last )
   new ( ( void * )&*_Dest ++)
      iterator_traits<InputIterator>::value_type ( *_First ++ );
return _First;
```

unless the code throws an exception. In that case, all constructed objects are destroyed and the exception is rethrown.

uninitialized_copy has two related forms:

checked_uninitialized_copy

unchecked_uninitialized_copy

For more information on how these functions behave, see Checked Iterators.

Example

```
// memory_uninit_copy.cpp
// compile with: /EHsc /W3
#include <memory>
#include <iostream>

using namespace std;

   class Integer
   {
   public:
      Integer( int x ) : val( x ) {}
      int get( ) { return val; }
   private:
      int val;
   };
```

```
int main( )
{
   int Array[] = { 10, 20, 30, 40 };
   const int N = sizeof( Array ) / sizeof( int );

   int i;
   cout << "The initialized Array contains " << N << " elements: ";
      for (i = 0 ; i < N; i++ )
      {
         cout << " " << Array [ i ];
      }
   cout << endl;

   Integer* ArrayPtr = ( Integer* ) malloc( N * sizeof( int ) );
   Integer* LArrayPtr = uninitialized_copy(
      Array, Array + N, ArrayPtr);  // C4996

   cout << "Address of position after the last element in the array is: "
        << &Array[0] + N << endl;
   cout << "The iterator returned by uninitialized_copy addresses: "
        << ( void* )LArrayPtr << endl;
   cout << "The address just beyond the last copied element is: "
        << ( void* )( ArrayPtr + N ) << endl;

   if ( ( &Array[0] + N ) == ( void* )LArrayPtr )
      cout << "The return value is an iterator "
           << "pointing just beyond the original array." << endl;
   else
      cout << "The return value is an iterator "
           << "not pointing just beyond the original array." << endl;

   if ( ( void* )LArrayPtr == ( void* )( ArrayPtr + N ) )
      cout << "The return value is an iterator "
           << "pointing just beyond the copied array." << endl;
   else
      cout << "The return value is an iterator "
           << "not pointing just beyond the copied array." << endl;

   free ( ArrayPtr );

   cout << "Note that the exact addresses returned will vary\n"
        << "with the memory allocation in individual computers."
        << endl;
}
```

Example Output

```
The initialized Array contains 4 elements: 10 20 30 40
Address of position after the last element in the array is: 0012FED8
The iterator returned by uninitialized_copy addresses: 00311B88
The address just beyond the last copied element is: 00311B88
The return value is an iterator not pointing just beyond the original array.
The return value is an iterator pointing just beyond the copied array.
Note that the exact addresses returned will vary
with  the memory allocation in individual computers.
```

Requirements

Header: <memory>

See Also

Other Resources

<memory> Members

uninitialized_fill

Copies objects of a specified value into an uninitialized destination range.

```
template<class ForwardIterator, class Type>
   void uninitialized_fill(
      ForwardIterator _First,
      ForwardIterator _Last,
      const Type& _Val
   );
```

Parameters

_First

A forward iterator addressing the first element in the destination range that is to be initiated.

_Last

A forward iterator addressing the last element in the destination range that is to be initiated.

_Val

The value to be used to initialize the destination range.

Remarks

This algorithm allows the decoupling of memory allocation from object construction.

The template function effectively executes:

```
while ( _First!= _Last )
   new ( (void *)&*_First ++)
      iterator_traits<ForwardIterator>::value_type ( _Val );
```

unless the code throws an exception. In that case, all constructed objects are destroyed and the exception is rethrown.

Example

```
// memory_uninit_fill.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

using namespace std;

   class Integer {         // No default constructor
   public:
      Integer( int x ) : val( x ) {}
      int get( ) { return val; }
   private:
      int val;
   };

int main( )
{
   const int N = 10;
   Integer val ( 25 );
   Integer* Array = ( Integer* ) malloc( N * sizeof( int ) );
   uninitialized_fill( Array, Array + N, val );
   int i;
   cout << "The initialized Array contains: ";
      for ( i = 0 ; i < N; i++ )
      {
         cout << Array [ i ].get( ) << " ";
      }
```

```
    cout << endl;
}
```

Output

```
The initialized Array contains: 25 25 25 25 25 25 25 25 25 25
```

Requirements

Header: <memory>

See Also

Other Resources

<memory> Members

uninitialized_fill_n

Copies objects of a specified value into specified number of elements into an uninitialized destination range.

```
template<class FwdIt, class Size, class Type>
   void uninitialized_fill_n(
      ForwardIterator _First,
      Size _Count,
      const Type& _Val
   );
```

Parameters

_First

A forward iterator addressing the first element in the destination range to be initiated.

_Count

The number of elements to be initialized.

_Val

The value to be used to initialize the destination range.

Remarks

This algorithm allows the decoupling of memory allocation from object construction.

The template function effectively executes:

```
while ( 0 < count-- )
   new ( ( void * )&*_First ++ )
      iterator_traits<ForwardIterator>::value_type( _Val );
```

unless the code throws an exception. In that case, all constructed objects are destroyed and the exception is rethrown.

uninitialized_fill_n has two related forms:

- checked_uninitialized_fill_n
- unchecked_uninitialized_fill_n

For information on how these functions behave, see Checked Iterators.

Example

```
// memory_uninit_fill_n.cpp
// compile with: /EHsc /W3
#include <memory>
#include <iostream>

using namespace std;

class Integer {   // No default constructor
public:
   Integer( int x ) : val( x ) {}
   int get( ) { return val; }
private:
   int val;
};

int main() {
   const int N = 10;
   Integer val ( 60 );
   Integer* Array = ( Integer* ) malloc( N * sizeof( int ) );
   uninitialized_fill_n( Array, N, val );  // C4996
```

```
    int i;
    cout << "The uninitialized Array contains: ";
    for ( i = 0 ; i < N; i++ )
       cout << Array [ i ].get( ) <<  " ";
 }
```

Output

```
The uninitialized Array contains: 60 60 60 60 60 60 60 60 60 60
```

Requirements

Header: <memory>

See Also

Other Resources

<memory> Members

<memory> Classes

For information about the classes in <memory>, see <memory> Members.

allocator Class

The template class describes an object that manages storage allocation and freeing for arrays of objects of type **Type**. An object of class **allocator** is the default allocator object specified in the constructors for several container template classes in the Standard C++ Library.

For a list of all members of this type, see allocator Members.

```
template <class Type>
class allocator
```

Parameters

Type

The type of object for which storage is being allocated or deallocated.

Remarks

All the Standard Template Library containers have a template parameter that defaults to **allocator**. Template class **allocator** supplies several type definitions that are rather pedestrian. They hardly seem worth defining. But another class with the same members might choose more interesting alternatives. Constructing a container with an allocator object of such a class gives individual control over allocation and freeing of elements controlled by that container.

For example, an allocator object might allocate storage on a private heap. It might allocate storage on a far heap, requiring nonstandard pointers to access the allocated objects. It might also specify, through the type definitions it supplies, that elements be accessed through special accessor objects that manage shared memory, or perform automatic garbage collection. Hence, a class that allocates storage using an allocator object should use these types for declaring pointer and reference objects, as the containers in the Standard C++ Library do.

When you derive from allocator class, you have to provide a rebind struct, whose _*Other* typedef references your newly-derived class.

Thus, an allocator defines the following types:

- pointer behaves like a pointer to **Type**.
- const_pointer behaves like a const pointer to **Type**.
- reference behaves like a reference to **Type**.
- const_reference behaves like a const reference to **Type**.

These **Type**s specify the form that pointers and references must take for allocated elements. (allocator::pointer is not necessarily the same as **Type*** for all allocator objects, even though it has this obvious definition for class **allocator**.)

Requirements

Header: <memory>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

allocator Members

<memory> Members

allocator Members

Reference

Constructors

allocator	Constructors used to create **allocator** objects.

Typedefs

const_pointer	A type that provides a constant pointer to the type of object managed by the allocator.
const_reference	A type that provides a constant reference to type of object managed by the allocator.
difference_type	A signed integral type that can represent the difference between values of pointers to the type of object managed by the allocator.
pointer	A type that provides a pointer to the type of object managed by the allocator.
reference	A type that provides a reference to the type of object managed by the allocator.
size_type	An unsigned integral type that can represent the length of any sequence that an object of template class **allocator** can allocate.
value_type	A type that is managed by the allocator.

Member Functions

address	Finds the address of an object whose value is specified.
allocate	Allocates a block of memory large enough to store at least some specified number of elements.
construct	Constructs a specific type of object at a specified address that is initialized with a specified value.
deallocate	Frees a specified number of objects from storage beginning at a specified position.
destroy	Calls an objects destructor without deallocating the memory where the object was stored.
max_size	Returns the number of elements of type **Type** that could be allocated by an object of class **allocator** before the free memory is used up.
rebind	A structure that enables an allocator for objects of one type to allocate storage for objects of another type.

Operators

operator=	Assigns one **allocator** object to another **allocator** object.

See Also

Reference

allocator Class

Thread Safety in the Standard C++ Library

allocator Typedefs

For information about the typedefs in the allocator class, see allocator Members.

allocator::const_pointer

A type that provides a constant pointer to the type of object managed by the allocator.

```
typedef const Type *const_pointer;
```

Remarks

The pointer type describes an object **ptr** that can designate, through the expression ***ptr**, any const object that an object of template class allocator can allocate.

Example

```
// allocator_const_ptr.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   vector <int> v1;
   vector <int>::iterator v1Iter;
   vector <int>:: allocator_type v1Alloc;

   int i;
   for ( i = 1 ; i <= 7 ; i++ )
   {
      v1.push_back( i * 2 );
   }

   cout << "The original vector v1 is:\n ( " ;
   for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
      cout << *v1Iter << " ";
   cout << "))." << endl;

   allocator<int>::const_pointer v1Ptr;
   const int k = 10;
   v1Ptr = v1Alloc.address( k );

   cout << "The integer's address found has a value of: "
        << *v1Ptr << "." << endl;
}
```

Output

```
The original vector v1 is:
 ( 2 4 6 8 10 12 14 ).
The integer's address found has a value of: 10.
```

Requirements

Header: <memory>

See Also

Reference

allocator Class

Other Resources

allocator Members

allocator::const_reference

A type that provides a constant reference to type of object managed by the allocator.

```
typedef const Type& const_reference;
```

Remarks

The reference type describes an object that can designate any const object that an object of template class allocator can allocate.

Example

```
// allocator_const_ref.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   vector <double> v;
   vector <double> ::iterator vIter, vfIter;
   vector <double> :: allocator_type vAlloc;

   int j;
   for ( j = 1 ; j <= 7 ; j++ )
   {
      v.push_back( 100.0 * j );
   }

   cout << "The original vector v is:\n ( " ;
   for ( vIter = v.begin( ) ; vIter != v.end( ) ; vIter++ )
      cout << *vIter << " ";
   cout << "))." << endl;

   vfIter = v.begin( );
   allocator<double>::const_reference vcref =*vfIter;
   cout << "The value of the element referred to by vref is: "
        << vcref << ",\n the first element in the vector." << endl;

   // const references can have their elements modified,
   // so the following would generate an error:
   // vcref = 150;
   // but the value of the first element could be modified through
   // its nonconst iterator and the const reference would remain valid
   *vfIter = 175;
   cout << "The value of the element referred to by vcref,"
        <<"\n after nofication through its nonconst iterator, is: "
        << vcref << "." << endl;
}
```

Output

```
The original vector v is:
 ( 100 200 300 400 500 600 700 ).
The value of the element referred to by vref is: 100,
 the first element in the vector.
The value of the element referred to by vcref,
 after nofication through its nonconst iterator, is: 175.
```

Requirements

Header: <memory>

See Also

Reference

allocator Class

Other Resources

allocator Members

allocator::difference_type

A signed integral type that can represent the difference between values of pointers to the type of object managed by the allocator.

```
typedef ptrdiff_t difference_type;
```

Remarks

The signed integer type describes an object that can represent the difference between the addresses of any two elements in a sequence that an object of template class allocator can allocate.

Example

```
// allocator_diff_type.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   vector <int> v1;
   vector <int>::iterator v1Iter;
   vector <int>:: allocator_type v1Alloc;

   int i;
   for ( i = 0 ; i <= 7 ; i++ )
   {
      v1.push_back( i * 2 );
   }

   cout << "The original vector v1 is:\n ( " ;
   for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
      cout << *v1Iter << " ";
   cout << ")." << endl;

   allocator<int>::const_pointer v1PtrA, v1PtrB;
   const int kA = 4, kB =12;
   v1PtrA = v1Alloc.address( kA );
   v1PtrB = v1Alloc.address( kB );
   allocator<int>::difference_type v1diff = *v1PtrB - *v1PtrA;

   cout << "Pointer v1PtrA addresses " << *v1PtrA << "." << endl;
   cout << "Pointer v1PtrB addresses " << *v1PtrB <<  "." << endl;
   cout << "The difference between the integer's addresses is: "
        << v1diff << "." << endl;
}
```

Output

```
The original vector v1 is:
 ( 0 2 4 6 8 10 12 14 ).
Pointer v1PtrA addresses 4.
Pointer v1PtrB addresses 12.
The difference between the integer's addresses is: 8.
```

Requirements

Header: <memory>

See Also

Reference

allocator Class

Other Resources

allocator Members

allocator::pointer

A type that provides a pointer to the type of object managed by the allocator.

```
typedef Type *pointer;
```

Remarks

The pointer type describes an object **ptr** that can designate, through the expression ***ptr**, any object that an object of template class allocator can allocate.

Example

```
// allocator_ptr.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   vector <int> v1;
   vector <int>::iterator v1Iter;
   vector <int>:: allocator_type v1Alloc;

   int i;
   for ( i = 1 ; i <= 7 ; i++ )
   {
      v1.push_back( 3 * i );
   }

   cout << "The original vector v1 is:\n ( " ;
   for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
      cout << *v1Iter << " ";
   cout << "))." << endl;

   allocator<int>::const_pointer v1Ptr;
   const int k = 12;
   v1Ptr = v1Alloc.address( k );

   cout << "The integer addressed by v1Ptr has a value of: "
        << "*v1Ptr = " << *v1Ptr << "." << endl;
}
```

Output

```
The original vector v1 is:
 ( 3 6 9 12 15 18 21 ).
The integer addressed by v1Ptr has a value of: *v1Ptr = 12.
```

Requirements

Header: <memory>

See Also

Reference

allocator Class

Other Resources

allocator Members

allocator::reference

A type that provides a reference to the type of object managed by the allocator.

```
typedef Type& reference;
```

Remarks

The reference type describes an object that can designate any object that an object of template class allocator can allocate.

Example

```
// allocator_reference.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   vector <double> v;
   vector <double> ::iterator vIter, vfIter;
   vector <double> :: allocator_type vAlloc;

   int j;
   for ( j = 1 ; j <= 7 ; j++ )
   {
      v.push_back( 100.0 * j );
   }

   cout << "The original vector v is:\n ( " ;
   for ( vIter = v.begin( ) ; vIter != v.end( ) ; vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;

   vfIter = v.begin( );
   allocator<double>::reference vref =*vfIter;
   cout << "The value of the element referred to by vref is: "
        << vref << ",\n the first element in the vector." << endl;

   // nonconst references can have their elements modified
   vref = 150;
   cout << "The element referred to by vref after being modified is: "
        << vref << "." << endl;
}
```

Output

```
The original vector v is:
 ( 100 200 300 400 500 600 700 ).
The value of the element referred to by vref is: 100,
 the first element in the vector.
The element referred to by vref after being modified is: 150.
```

Requirements

Header: <memory>

See Also

Reference

allocator Class

Other Resources

allocator Members

allocator::size_type

An unsigned integral type that can represent the length of any sequence that an object of template class allocator can allocate.

```
typedef size_t size_type;
```

Example

```
// allocator_size_type.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   vector <double> v;
   vector <double> ::iterator vIter;
   vector <double> :: allocator_type vAlloc;

   int j;
   for ( j = 1 ; j <= 7 ; j++ )
   {
      v.push_back( 100.0 * j );
   }

   cout << "The original vector v is:\n ( " ;
   for ( vIter = v.begin( ) ; vIter != v.end( ) ; vIter++ )
      cout << *vIter << " ";
   cout << "))." << endl;

   allocator<double>::size_type vsize;
   vsize = vAlloc.max_size( );

   cout << "The number of doubles that can be allocated before\n"
        << " the free memory in the vector v is used up is: "
        << vsize << "." << endl;
}
```

Sample Output

The following output is for x86.

```
The original vector v is:
 ( 100 200 300 400 500 600 700 ).
The number of doubles that can be allocated before
 the free memory in the vector v is used up is: 536870911.
```

Requirements

Header: <memory>

See Also

Reference

allocator Class

Other Resources

allocator Members

allocator::value_type

A type that is managed by the allocator.

```
typedef Type value_type;
```

Remarks

The type is a synonym for the template parameter **Type**.

Example

```
// allocator_value_type.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>
using namespace std;

int main( )
{
   vector <double> v;
   vector <double> ::iterator vIter, vfIter;
   vector <double> :: allocator_type vAlloc;

   int j;
   for ( j = 1 ; j <= 7 ; j++ )
   {
      v.push_back( 100.0 * j );
   }

   cout << "The original vector v is:\n ( " ;
   for ( vIter = v.begin( ) ; vIter != v.end( ) ; vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;

   vfIter = v.begin( );
   allocator<double>::value_type vecVal = 150.0;
   *vfIter = vecVal;
   cout << "The value of the element addressed by vfIter is: "
        << *vfIter << ",\n the first element in the vector." << endl;

   cout << "The modified vector v is:\n ( " ;
   for ( vIter = v.begin( ) ; vIter != v.end( ) ; vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;
}
```

Output

```
The original vector v is:
 ( 100 200 300 400 500 600 700 ).
The value of the element addressed by vfIter is: 150,
 the first element in the vector.
The modified vector v is:
 ( 150 200 300 400 500 600 700 ).
```

Requirements

Header: <memory>

See Also

Reference

allocator Class

Other Resources

allocator Member Functions

For information about the functions in the allocator class, see allocator Members.

allocator::address

Finds the address of an object whose value is specified.

```
pointer (
   reference _Val
   ) const;
const_pointer (
   const_reference _Val
   ) const;
```

Parameters

_Val

The const or nonconst value of the object whose address is being searched for.

Return Value

A const or nonconst pointer to the object found of, respectively, const or nonconst value.

Remarks

The member functions return the address of *_Val*, in the form that pointers must take for allocated elements.

Example

```
// allocator_address.cpp
// compile with: /EHsc
#include <memory>
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   vector <int> v1;
   vector <int>::iterator v1Iter;
   vector <int>:: allocator_type v1Alloc;

   int i;
   for ( i = 1 ; i <= 7 ; i++ )
   {
      v1.push_back( 2 * i );
   }

   cout << "The original vector v1 is:\n ( " ;
   for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
      cout << *v1Iter << " ";
   cout << "))." << endl;

   allocator<int>::const_pointer v1Ptr;
   const int k = 8;
   v1Ptr = v1Alloc.address( *find(v1.begin( ), v1.end( ), k) );
   // v1Ptr = v1Alloc.address( k );
   cout << "The integer addressed by v1Ptr has a value of: "
        << "*v1Ptr = " << *v1Ptr << "." << endl;
}
```

Output

```
The original vector v1 is:
 ( 2 4 6 8 10 12 14 ).
The integer addressed by v1Ptr has a value of: *v1Ptr = 8.
```

Requirements

Header: <memory>

See Also

Reference

allocator Class

Other Resources

allocator Members

allocator::allocate

Allocates a block of memory large enough to store at least some specified number of elements.

```
template<class Other>
   pointer allocate(
      size_type _Count,
      const Other* _Hint = 0
   );
```

Parameters

_Count

The number of elements for which sufficient storage is to be allocated.

_Hint

A const pointer that may assist the allocator object satisfy the request for storage by locating the address of an object allocated prior to the request.

Return Value

A pointer to the allocated object.

Remarks

The member function allocates storage for an array of count elements of type **Type**, by calling operator new(*_Count*). It returns a pointer to the allocated object. The hint argument helps some allocators in improving locality of reference; a valid choice is the address of an object earlier allocated by the same allocator object and not yet deallocated. To supply no hint, use a null pointer argument instead.

Example

```
// allocator_allocate.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   allocator<int> v1Alloc;

   allocator<int>::pointer v1aPtr;

   v1aPtr = v1Alloc.allocate ( 10 );

   int i;
   for ( i = 0 ; i < 10 ; i++ )
   {
      v1aPtr[ i ] = i;
   }

   for ( i = 0 ; i < 10 ; i++ )
   {
      cout << v1aPtr[ i ] << " ";
   }
   cout << endl;

   v1Alloc.deallocate( v1aPtr, 10 );
}
```

Output

0 1 2 3 4 5 6 7 8 9

Requirements

Header: <memory>

See Also

Reference

allocator Class

Other Resources

allocator Members

allocator::allocator

Constructors used to create allocator objects.

```
allocator( );
allocator(
   const allocator<Type>& _Right
);
template<class Other>
   allocator(
      const allocator<Other>& _Right
      );
```

Parameters

_Right

The allocator object to be copied.

Remarks

The constructor does nothing. In general, however, an allocator object constructed from another allocator object should compare equal to it and permit intermixing of object allocation and freeing between the two allocator objects.

Example

```
// allocator_allocator.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

class Int {
public:
   Int( int i )
   {
      cout << "Constructing " << ( void* )this  << endl;
      x = i;
      bIsConstructed = true;
   };
   ~Int( )
   {
      cout << "Destructing " << ( void* )this << endl;
      bIsConstructed = false;
   };
   Int &operator++( )
   {
      x++;
      return *this;
   };
   int x;
private:
   bool bIsConstructed;
};

int main( )
{
   allocator<double> Alloc;
   vector <Int>:: allocator_type v1Alloc;

   allocator<double> cAlloc(Alloc);
   allocator<Int> cv1Alloc(v1Alloc);

   if ( cv1Alloc == v1Alloc )
```

```
      cout << "The allocator objects cv1Alloc & v1Alloc are equal."
           << endl;
   else
      cout << "The allocator objects cv1Alloc & v1Alloc are not equal."
           << endl;

   if ( cAlloc == Alloc )
      cout << "The allocator objects cAlloc & Alloc are equal."
           << endl;
   else
      cout << "The allocator objects cAlloc & Alloc are not equal."
           << endl;
}
```

Output

```
The allocator objects cv1Alloc & v1Alloc are equal.
The allocator objects cAlloc & Alloc are equal.
```

Requirements

Header: <memory>

See Also

Reference

allocator Class

Other Resources

allocator Members

allocator::construct

Constructs a specific type of object at a specified address that is initialized with a specified value.

```
void construct(
   pointer _Ptr,
   const Type& _Val
);
```

Parameters

_Ptr

A pointer to the location where the object is to be constructed.

_Val

The value with which the object being constructed is to be initialized.

Remarks

The member function is equivalent to **new** ((**void** *) *_Ptr*) **Type** (*_Val*).

Example

```
// allocator_construct.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main( )
{
   vector <int> v1;
   vector <int>::iterator v1Iter;
   vector <int>:: allocator_type v1Alloc;

   int i;
   for ( i = 1 ; i <= 7 ; i++ )
   {
      v1.push_back( 3 * i );
   }

   cout << "The original vector v1 is:\n ( " ;
   for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
      cout << *v1Iter << " ";
   cout << ")." << endl;

   allocator<int>::pointer v1PtrA;
   int kA = 6, kB = 7;
   v1PtrA = v1Alloc.address( *find( v1.begin( ), v1.end( ), kA ) );
   v1Alloc.destroy ( v1PtrA );
   v1Alloc.construct ( v1PtrA , kB );

   cout << "The modified vector v1 is:\n ( " ;
   for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
      cout << *v1Iter << " ";
   cout << ")." << endl;
}
```

Output

```
The original vector v1 is:
```

```
 ( 3 6 9 12 15 18 21 ).
The modified vector v1 is:
 ( 3 7 9 12 15 18 21 ).
```

Requirements

Header: <memory>

See Also

Reference

allocator Class

Other Resources

allocator Members

allocator::deallocate

Frees a specified number of objects from storage beginning at a specified position.

```
void deallocate(
   pointer _Ptr,
   size_type _Count
);
```

Parameters

_Ptr

A pointer to the first object to be deallocated from storage.

_Count

The number of objects to be deallocated from storage.

Remarks

The member function frees storage for the array of count objects of type **Type** beginning at *_Ptr*, by calling **operator delete**(*_Ptr*). The pointer *_Ptr* must have been returned earlier by a call to allocate for an allocator object that compares equal to ***this**, allocating an array object of the same size and type. **deallocate** never throws an exception.

Example

For an example using the member function, see allocator::allocate.

Requirements

Header: <memory>

See Also

Reference

allocator Class

Other Resources

allocator Members

allocator::destroy

Calls an objects destructor without deallocating the memory where the object was stored.

```
void destroy(
   pointer _Ptr
);
```

Parameters

_Ptr

A pointer designating the address of the object to be destroyed.

Remarks

The member function destroys the object designated by *_Ptr*, by calling the destructor *_Ptr* ->**Type**::**~Type**.

Example

```
// allocator_destroy.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main( )
{
   vector <int> v1;
   vector <int>::iterator v1Iter;
   vector <int>:: allocator_type v1Alloc;

   int i;
   for ( i = 1 ; i <= 7 ; i++ )
   {
      v1.push_back( 2 * i );
   }

   cout << "The original vector v1 is:\n ( " ;
   for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
      cout << *v1Iter << " ";
   cout << ")." << endl;

   allocator<int>::pointer v1PtrA;
   int kA = 12, kB = -99;
   v1PtrA = v1Alloc.address( *find(v1.begin( ), v1.end( ), kA) );
   v1Alloc.destroy ( v1PtrA );
   v1Alloc.construct ( v1PtrA , kB );

   cout << "The modified vector v1 is:\n ( " ;
   for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
      cout << *v1Iter << " ";
   cout << ")." << endl;
}
```

Output

```
The original vector v1 is:
 ( 2 4 6 8 10 12 14 ).
The modified vector v1 is:
 ( 2 4 6 8 10 -99 14 ).
```

Requirements

Header: <memory>

See Also

Reference

allocator Class

Other Resources

allocator Members

allocator::max_size

Returns the number of elements of type **Type** that could be allocated by an object of class allocator before the free memory is used up.

```
size_type max_size( ) const;
```

Return Value

The number of elements that could be allocated.

Example

```
// allocator_max_size.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   vector <int> v1;
   vector <int>::iterator v1Iter;
   vector <int>:: allocator_type v1Alloc;

   int i;
   for ( i = 1 ; i <= 7 ; i++ )
   {
      v1.push_back( i );
   }

   cout << "The original vector v1 is:\n ( " ;
   for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
      cout << *v1Iter << " ";
   cout << ")." << endl;

   vector <double> v2;
   vector <double> ::iterator v2Iter;
   vector <double> :: allocator_type v2Alloc;
   allocator<int>::size_type v1size;
   v1size = v1Alloc.max_size( );

   cout << "The number of integers that can be allocated before\n"
        << " the free memory in the vector v1 is used up is: "
        << v1size << "." << endl;

   int ii;
   for ( ii = 1 ; ii <= 7 ; ii++ )
   {
      v2.push_back( ii * 10.0 );
   }

   cout << "The original vector v2 is:\n ( " ;
   for ( v2Iter = v2.begin( ) ; v2Iter != v2.end( ) ; v2Iter++ )
      cout << *v2Iter << " ";
   cout << ")." << endl;
   allocator<double>::size_type v2size;
   v2size = v2Alloc.max_size( );

   cout << "The number of doubles that can be allocated before\n"
        << " the free memory in the vector v2 is used up is: "
        << v2size << "." << endl;
```

```
}
```

Sample Output

The following output is for x86.

```
The original vector v1 is:
 ( 1 2 3 4 5 6 7 ).
The number of integers that can be allocated before
 the free memory in the vector v1 is used up is: 1073741823.
The original vector v2 is:
 ( 10 20 30 40 50 60 70 ).
The number of doubles that can be allocated before
 the free memory in the vector v2 is used up is: 536870911.
```

Requirements

Header: <memory>

See Also

Reference

allocator Class

Other Resources

allocator Members

allocator::rebind

A structure that enables an allocator for objects of one type to allocate storage for objects of another type.

```
template<class _Other>
   struct rebind {
   typedef allocator<_Other> other;
   };
```

Parameters

other

The type of element for which memory is being allocated.

Remarks

This structure is useful for allocating memory for type that differs from the element type of the container being implemented.

The member template class defines the type other. Its sole purpose is to provide the type name **allocator**<_**Other**>, given the type name **allocator**<**Type**>.

For example, given an allocator object **al** of type **A**, you can allocate an object of type _**Other** with the expression:

```
A::rebind<Other>::other(al).allocate(1, (Other *)0)
```

Or, you can name its pointer type by writing the type:

```
A::rebind<Other>::other::pointer
```

Example

```
// allocator_rebind.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

typedef vector<int>::allocator_type IntAlloc;
int main( )
{
   IntAlloc v1Iter;
   vector<int> v1;

   IntAlloc::rebind<char>::other::pointer pszC =
      IntAlloc::rebind<char>::other(v1.get_allocator()).allocate(1, (void *)0);

   int * pInt = v1Iter.allocate(10);
}
```

Requirements

Header: <memory>

See Also

Reference

allocator Class

Other Resources

allocator Members

allocator Operators

For information about the operators in the allocator class, see allocator Class.

allocator::operator=

Assigns one allocator object to another allocator object.

```
template<class Other>
   allocator& operator=(
      const allocator<Other>& _Right
   );
```

Parameters

_Right

An allocator object to be assigned to another such object.

Return Value

A reference to the allocator object

Remarks

The template assignment operator does nothing. In general, however, an allocator object assigned to another allocator object should compare equal to it and permit intermixing of object allocation and freeing between the two allocator objects.

Example

```
// allocator_op_assign.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

class Int {
public:
   Int(int i)
   {
      cout << "Constructing " << ( void* )this  << endl;
      x = i;
      bIsConstructed = true;
   };
   ~Int( ) {
      cout << "Destructing " << ( void* )this << endl;
      bIsConstructed = false;
   };
   Int &operator++( )
   {
      x++;
      return *this;
   };
   int x;
private:
   bool bIsConstructed;
};

int main( )
{
   allocator<Int> Alloc;
   allocator<Int> cAlloc ;
   cAlloc = Alloc;
}
```

Requirements

Header: <memory>

See Also

Reference

allocator Class

Other Resources

allocator Members

auto_ptr Class

The template class describes an object that stores a pointer to an allocated object of type **Type*** that ensures that the object to which it points gets destroyed automatically when control leaves a block.

For a list of all members of this type, see auto_ptr Members.

```
template <class Type>
class auto_ptr
```

Parameters

Type

The type of object for which storage is being allocated or deallocated.

Remarks

The template class describes an object that stores a pointer to an allocated object **myptr** of type **Type ***. The stored pointer must either be null or designate an object allocated by a new expression. An object constructed with a nonnull pointer owns the pointer. It transfers ownership if its stored value is assigned to another object. (It replaces the stored value after a transfer with a null pointer.) The destructor for **auto_ptr**<**Type**> deletes the allocated object if it owns it. Hence, an object of class **auto_ptr**<**Type**> ensures that an allocated object is automatically deleted when control leaves a block, even through a thrown exception. You should not construct two **auto_ptr**<**Type**> objects that own the same object.

You can pass an **auto_ptr**<**Type**> object by value as an argument to a function call. You can return such an object by value as well. Both operations depend on the implicit construction of intermediate objects of class **auto_ptr**<**Type**>**::auto_ptr_ref**<**Other**>, by various subtle conversion rules. You cannot, however, reliably manage a sequence of **auto_ptr**<**Type**> objects with a Standard Template Library container.

Requirements

Header: <memory>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

auto_ptr Members

<memory> Members

auto_ptr Members

Reference

Constructors

auto_ptr	The constructor for objects of type **auto_ptr**.

Typedefs

element_type	The type is a synonym for the template parameter **Type**.

Member Functions

get	The member function returns the stored pointer **myptr**.
release	The member replaces the stored pointer **myptr** with a null pointer and returns the previously stored pointer.
reset	The member function evaluates the expression `delete myptr`, but only if the stored pointer value **myptr** changes as a result of function call. It then replaces the stored pointer with **ptr**.

Operators

operator=	An assignment operator that transfers ownership from one **auto_ptr** object to another.
operator*	The dereferencing operator for objects of type **auto_ptr**.
operator->	The operator for allowing member access.
operator auto_ptr<Other>	Casts from one kind of **auto_ptr** to another kind of **auto_ptr**.
operator auto_ptr_ref<Other>	Casts from an **auto_ptr** to an **auto_ptr_ref**.

See Also

Reference

auto_ptr Class

Thread Safety in the Standard C++ Library

auto_ptr Typedefs

For information about the typedefs in the auto_ptr class, see auto_ptr Members.

auto_ptr::element_type

The type is a synonym for the template parameter **Type**.

```
typedef Type element_type;
```

Requirements

Header: <memory>

See Also

Reference

auto_ptr Class

Other Resources

auto_ptr Members

auto_ptr Member Functions

For information about the functions in the auto_ptr class, see auto_ptr Members.

auto_ptr::auto_ptr

The constructor for objects of type **auto_ptr**.

```
explicit auto_ptr(
   Type* _Ptr = 0
) throw( );
auto_ptr(
   auto_ptr<Type>& _Right
) throw( );
auto_ptr(
   auto_ptr_ref<Type> _Right
) throw( );
template<class Other>
   auto_ptr(
   auto_ptr<Other>& _Right
) throw( );
```

Parameters

_Ptr

The pointer to the object that **auto_ptr** encapsulates.

_Right

The **auto_ptr** object to be copied by the constructor.

Remarks

The first constructor stores *_Ptr* in **myptr**, the stored pointer to the allocated object. The second constructor transfers ownership of the pointer stored in *_Right*, by storing *_Right*.release in **myptr**.

The third constructor behaves the same as the second, except that it stores **right**.*ref*.**release** in **myptr**, where *ref* is the reference stored in *_Right*.

The template constructor behaves the same as the second constructor, provided that a pointer to **Other** can be implicitly converted to a pointer to **Type**.

Example

```
// auto_ptr_auto_ptr.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

class Int
{
public:
   Int(int i)
   {
      cout << "Constructing " << ( void* )this  << endl;
      x = i;
      bIsConstructed = true;
   };
   ~Int( )
   {
      cout << "Destructing " << ( void* )this << endl;
      bIsConstructed = false;
   };
   Int &operator++( )
   {
      x++;
      return *this;
```

```
   };
   int x;
private:
   bool bIsConstructed;
};

void function ( auto_ptr<Int> &pi )
{
   ++( *pi );
   auto_ptr<Int> pi2( pi );
   ++( *pi2 );
   pi = pi2;
}

int main( )
{
   auto_ptr<Int> pi ( new Int( 5 ) );
   cout << pi->x << endl;
   function( pi );
   cout << pi->x << endl;
}
```

Sample Output

```
Constructing 00311AF8
5
7
Destructing 00311AF8
```

Requirements

Header: <memory>

See Also

Reference

auto_ptr Class

Other Resources

auto_ptr Members

auto_ptr::get

The member function returns the stored pointer **myptr**.

```
Type *get( ) const throw( );
```

Return Value

The stored pointer **myptr**.

Example

```
// auto_ptr_get.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>
using namespace std;

class Int
{
public:
   Int(int i)
   {
      x = i;
      cout << "Constructing " << ( void* )this  << " Value: " << x << endl;
   };
   ~Int( )
   {
      cout << "Destructing " << ( void* )this << " Value: " << x << endl;
   };

   int x;

};

int main( )
{
   auto_ptr<Int> pi ( new Int( 5 ) );
   pi.reset( new Int( 6 ) );
   Int* pi2 = pi.get ( );
   Int* pi3 = pi.release ( );
   if (pi2 == pi3)
      cout << "pi2 == pi3" << endl;
   delete pi3;
}
```

Sample Output

```
Constructing 00311AF8 Value: 5
Constructing 00311B88 Value: 6
Destructing 00311AF8 Value: 5
pi2 == pi3
Destructing 00311B88 Value: 6
```

Requirements

Header: <memory>

See Also

Reference

auto_ptr Class

Other Resources

auto_ptr Members

auto_ptr::release

The member replaces the stored pointer **myptr** with a null pointer and returns the previously stored pointer.

```
Type *release( ) throw( );
```

Return Value

The previously stored pointer.

Remarks

The member replaces the stored pointer **myptr** with a null pointer and returns the previously stored pointer.

Example

```
// auto_ptr_release.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>
using namespace std;

class Int
{
public:
   Int( int i )
   {
      x = i;
      cout << "Constructing " << ( void* )this << " Value: " << x << endl;
   };
   ~Int( ) {
      cout << "Destructing " << ( void* )this << " Value: " << x << endl;
   };

   int x;

};

int main( )
{
   auto_ptr<Int> pi ( new Int( 5 ) );
   pi.reset( new Int( 6 ) );
   Int* pi2 = pi.get ( );
   Int* pi3 = pi.release ( );
   if ( pi2 == pi3 )
      cout << "pi2 == pi3" << endl;
   delete pi3;
}
```

Sample Output

```
Constructing 00311AF8 Value: 5
Constructing 00311B88 Value: 6
Destructing 00311AF8 Value: 5
pi2 == pi3
Destructing 00311B88 Value: 6
```

Requirements

Header: <memory>

See Also

Reference

auto_ptr Class

Other Resources

auto_ptr Members

auto_ptr::reset

The member function evaluates the expression **delete myptr**, but only if the stored pointer value **myptr** changes as a result of a function call. It then replaces the stored pointer with **ptr**.

```
void reset(
   Type* _Ptr = 0
);
```

Parameters

_Ptr

The pointer specified to replace the stored pointer **myptr**.

Example

```
// auto_ptr_reset.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

class Int
{
public:
   Int( int i )
   {
      x = i;
      cout << "Constructing " << ( void* )this << " Value: " << x << endl;
   };
   ~Int( )
   {
      cout << "Destructing " << ( void* )this << " Value: " << x << endl;
   };

   int x;
};

int main( )
{
   auto_ptr<Int> pi ( new Int( 5 ) );
   pi.reset( new Int( 6 ) );
   Int* pi2 = pi.get ( );
   Int* pi3 = pi.release ( );
   if ( pi2 == pi3 )
      cout << "pi2 == pi3" << endl;
   delete pi3;
}
```

Sample Output

```
Constructing 00311AF8 Value: 5
Constructing 00311B88 Value: 6
Destructing 00311AF8 Value: 5
pi2 == pi3
Destructing 00311B88 Value: 6
```

Requirements

Header: <memory>

See Also

Reference

auto_ptr Class

Other Resources

auto_ptr Members

auto_ptr Operators

For information about the operators in the auto_ptr class, see auto_ptr Class.

auto_ptr::operator=

An assignment operator that transfers ownership from one **auto_ptr** object to another.

```
template<class Other>
   auto_ptr<Type>& operator=(
      auto_ptr<Other>& _Right
   ) throw( );
auto_ptr<Type>& operator=(
   auto_ptr<>& _Right
) throw( );
```

Parameters

_Right

An object of type **auto_ptr**.

Return Value

A reference to an object of type **auto_ptr**<**Type**>.

Remarks

The assignment evaluates the expression **delete myptr**, but only if the stored pointer **myptr** changes as a result of the assignment. It then transfers ownership of the pointer stored in *_Right*, by storing *_Right*.release in **myptr**. The function returns ***this**.

Example

For an example of the use of the member operator, see auto_ptr::auto_ptr.

Requirements

Header: <memory>

See Also

Reference

auto_ptr Class

Other Resources

auto_ptr Members

auto_ptr::operator*

The dereferencing operator for objects of type **auto_ptr**.

```
Type& operator*( ) const throw( );
```

Return Value

A reference to an object of type **Type** that the pointer owns.

Remarks

The indirection operator returns *get. Hence, the stored pointer must not be null.

Example

For an example of how to use the member function, see auto_ptr::auto_ptr.

Requirements

Header: <memory>

See Also

Reference

auto_ptr Class

Other Resources

auto_ptr Members

auto_ptr::operator->

The operator for allowing member access.

```
Type *operator->( ) const throw( );
```

Return Value

A member of the object that **auto_ptr** owns.

Remarks

The selection operator returns get( ), so that the expression *ap*->**member** behaves the same as (*ap*.**get**( ))->**member**, where *ap* is an object of class **auto_ptr**<**Type**>. Hence, the stored pointer must not be null, and **Type** must be a class, struct, or union type with a **member** member.

Example

For an example of how to use the member function, see auto_ptr::auto_ptr.

Requirements

Header: <memory>

See Also

Reference

auto_ptr Class

Other Resources

auto_ptr Members

auto_ptr::operator auto_ptr<Other>

Casts from one kind of **auto_ptr** to another kind of **auto_ptr**.

```
template<class Other>
   operator auto_ptr<Other>( ) throw( );
```

Return Value

The type cast operator returns **auto_ptr** <**Other**>(***this**).

Example

```
// auto_ptr_op_auto_ptr.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;
int main()
{
   auto_ptr<int> pi ( new int( 5 ) );
   auto_ptr<const int> pc = ( auto_ptr<const int> )pi;
}
```

Requirements

Header: <memory>

See Also

Reference

auto_ptr Class

Other Resources

auto_ptr Members

auto_ptr::operator auto_ptr_ref<Other>

Casts from an **auto_ptr** to an **auto_ptr_ref**.

```
template<class Other>
   operator auto_ptr_ref<Other>( ) throw( );
```

Return Value

The type cast operator returns **auto_ptr_ref**<**Other**>(***this**).

Example

```
// auto_ptr_op_auto_ptr_ref.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

class C{
   public:
   C(int _i) : m_i(_i){
   }
   ~C(){
      cout << "~C:  "<< m_i <<"\n";
   }
   C &operator =(const int &x){
      m_i = x;
      return *this;
   }
   int m_i;
};
void f(auto_ptr<C> arg ){
};
int main()
{
   const auto_ptr<C> ciap ( new C(1) );
   auto_ptr<C> iap ( new C(2) );

   // Error: this implies transfer of ownership of iap's pointer
   // f(ciap);
   f(iap); // compiles, but gives up ownership of pointer

   // here, iap owns a destroyed pointer so the following is bad:
   // *iap = 5; // BOOM

   cout << "main exiting\n";
}
```

Output

```
~C:  2
main exiting
~C:  1
```

Requirements

Header: <memory>

See Also

Reference

auto_ptr Class

Other Resources

auto_ptr Members

raw_storage_iterator Class

An adaptor class that is provided to enable algorithms to store their results into uninitialized memory.

For a list of all members of this type, see raw_storage_iterator Members.

```
template <class OutputIterator, class Type>
class raw_storage_iterator
```

Parameters

OutputIterator

Specifies the output iterator for the object being stored.

Type

The type of object for which storage is being allocated.

Remarks

The class describes an output iterator that constructs objects of type **Type** in the sequence it generates. An object of class **raw_storage_iterator**<**ForwardIterator**, **Type**> accesses storage through a forward iterator object, of class **ForwardIterator**, that you specify when you construct the object. For an object first of class **ForwardIterator**, the expression **&*first** must designate unconstructed storage for the next object (of type **Type**) in the generated sequence.

This adaptor class is used when it is necessary to separate memory allocation and object construction. The **raw_storage_iterator** can be used to copy objects into uninitialized storage, such as memory allocated using the **malloc** function.

Requirements

Header: <memory>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

raw_storage_iterator Members

<memory> Members

raw_storage_iterator Members

Reference

Constructors

raw_storage_iterator	Constructs a raw storage iterator with a specified underlying output iterator.

Typedefs

element_type	Provides a type that describes an element to be stored a raw storage iterator.
iter_type	Provides a type that describes an iterator that underlies a raw storage iterator.

Operators

operator*	A dereferencing operator used to implement the output iterator expression *ii = x.
operator=	An assignment operator used to implement the raw storage iterator expression *i = x for storing in memory.
operator++	Preincrement and postincrement operators for raw storage iterators.

See Also

Reference

raw_storage_iterator Class

Thread Safety in the Standard C++ Library

raw_storage_iterator Typedefs

For information about the typedefs in the raw_storage_iterator class, see raw_storage_iterator Members.

raw_storage_iterator::element_type

Provides a type that describes an element to be stored a raw storage iterator.

```
typedef Type element_type;
```

Remarks

The type is a synonym for the raw_storage_iterator class template parameter **Type**.

Requirements

Header: <memory>

See Also

Reference

raw_storage_iterator Class

Other Resources

raw_storage_iterator Members

raw_storage_iterator::iter_type

Provides a type that describes an iterator that underlies a raw storage iterator.

```
typedef OutputIterator iter_type;
```

Remarks

The type is a synonym for the template parameter **ForwardIterator**.

Requirements

Header: <memory>

See Also

Reference

raw_storage_iterator Class

Other Resources

raw_storage_iterator Members

raw_storage_iterator Member Functions

For information about the functions in the raw_storage_iterator class, see raw_storage_iterator Members.

raw_storage_iterator::raw_storage_iterator

Constructs a raw storage iterator with a specified underlying output iterator.

```
explicit raw_storage_iterator(
   OutputIterator _First
);
```

Parameters

_First

The output iterator that is to underlie the **raw_storage_iterator** object being constructed.

Example

```
// raw_storage_iterator_ctor.cpp
// compile with: /EHsc /W3
#include <iostream>
#include <iterator>
#include <memory>
#include <list>
using namespace std;

class Int
{
public:
   Int(int i)
   {
      cout << "Constructing " << i << endl;
      x = i;
      bIsConstructed = true;
   };
   Int &operator=( int i )
   {
      if (!bIsConstructed)
         cout << "Error! I'm not constructed!\n";
      cout << "Copying " << i << endl;  x = i; return *this;
   };
   int x;
   bool bIsConstructed;
};

int main( void )
{
   std::list<int> l;
   l.push_back( 1 );
   l.push_back( 2 );
   l.push_back( 3 );
   l.push_back( 4 );

   Int *pInt = (Int*)malloc(sizeof(Int)*l.size( ));
   memset (pInt, 0, sizeof(Int)*l.size( ));
   // Hack: make sure bIsConstructed is false

   std::copy( l.begin( ), l.end( ), pInt );  // C4996
   for (unsigned int i = 0; i < l.size( ); i++)
      cout << "array " << i << " = " << pInt[i].x << endl;;

   memset (pInt, 0, sizeof(Int)*l.size( ));
   // hack: make sure bIsConstructed is false

   std::copy( l.begin( ), l.end( ),
      std::raw_storage_iterator<Int*,Int>(pInt));  // C4996
   for (unsigned int i = 0; i < l.size( ); i++ )
```

```
        cout << "array " << i << " = " << pInt[i].x << endl;

    free(pInt);
  }
```

Output

```
Error! I'm not constructed!
Copying 1
Error! I'm not constructed!
Copying 2
Error! I'm not constructed!
Copying 3
Error! I'm not constructed!
Copying 4
array 0 = 1
array 1 = 2
array 2 = 3
array 3 = 4
Constructing 1
Constructing 2
Constructing 3
Constructing 4
array 0 = 1
array 1 = 2
array 2 = 3
array 3 = 4
```

Requirements

Header: <memory>

See Also

Reference

raw_storage_iterator Class

Other Resources

raw_storage_iterator Members

raw_storage_iterator Operators

For information about the operators in the raw_storage_iterator class, see raw_storage_iterator Class.

raw_storage_iterator::operator*

A dereferencing operator used to implement the raw storage iterator expression **ii = x*.

```
raw_storage_iterator<OutputIterator, Type>& operator*( );
```

Return Value

A reference to the raw storage iterator

Remarks

The requirements for an **OutputIterator** are that the raw storage iterator must satisfy require only the expression **ii = t* be valid and that it says nothing about the **operator** or the **operator=** on their own. The member operators in this implementation returns ***this**, so that operator=(**const Type**&) can perform the actual store in an expression, such as **_Ptr = _Val*.

Example

```
// raw_storage_iterator_op_deref.cpp
// compile with: /EHsc
#include <iostream>
#include <iterator>
#include <memory>
#include <list>
using namespace std;

class Int
{
public:
   Int(int i)
   {
      cout << "Constructing " << i << endl;
      x = i;
      bIsConstructed = true;
   };

   Int &operator=(int i)
   {
      if (!bIsConstructed)
         cout << "Not constructed.\n";
      cout << "Copying " << i << endl;
      x = i;
      return *this;
   };

   int x;

private:
   bool bIsConstructed;
};

int main( void)
{
   Int *pInt = ( Int* ) malloc( sizeof( Int ) );
   memset( pInt, 0, sizeof( Int ) ); // Set bIsConstructed to false;
   *pInt = 5;
   raw_storage_iterator< Int*, Int > it( pInt );
   *it = 5;
}
```

Output

```
Not constructed.
```

```
Copying 5
Constructing 5
```

Requirements

Header: <memory>

See Also

Reference

raw_storage_iterator Class

Other Resources

raw_storage_iterator Members

raw_storage_iterator::operator=

Assignment operator used to implement the raw storage iterator expression **i = x* for storing in memory.

```
raw_storage_iterator<OutputIterator, Type>& operator=(
   const Type& _Val
);
```

Parameters

_Val

The value of the object of type **Type** to be inserted into memory.

Return Value

The operator inserts *_Val* into memory, and then returns a reference to the raw storage iterator.

Remarks

The requirements for an **OutputIterator** state that the raw storage iterator must satisfy require only the expression **ii = t* be valid, and that it says nothing about the **operator** or the **operator=** on their own. These member operators return ***this**.

The assignment operator constructs the next object in the output sequence using the stored iterator value first, by evaluating the placement new expression **new** ((**void** *)&***first**) **Type**(*_Val*).

Example

```
// raw_storage_iterator_op_assign.cpp
// compile with: /EHsc
#include <iostream>
#include <iterator>
#include <memory>
#include <list>
using namespace std;

class Int
{
public:
   Int( int i )
   {
      cout << "Constructing " << i << endl;
      x = i;
      bIsConstructed = true;
   };
   Int &operator=( int i )
   {
      if ( !bIsConstructed )
         cout << "Not constructed.\n";
      cout << "Copying " << i << endl; x = i;
      return *this;
   };
   int x;
private:
   bool bIsConstructed;
};

int main( void )
{
   Int *pInt = ( Int* )malloc( sizeof( Int ) );
   memset( pInt, 0, sizeof( Int ) ); // Set bIsConstructed to false;

   *pInt = 5;

   raw_storage_iterator<Int*, Int> it( pInt );
   *it = 5;
}
```

Output

```
Not constructed.
Copying 5
Constructing 5
```

Requirements

Header: <memory>

See Also

Reference

raw_storage_iterator Class

Other Resources

raw_storage_iterator Members

raw_storage_iterator::operator++

Preincrement and postincrement operators for raw storage iterators.

```
raw_storage_iterator<OutputIterator, Type>& operator++( );
raw_storage_iterator<OutputIterator, Type> operator++(Int);
```

Return Value

An raw storage iterator or a reference to an raw storage iterator.

Remarks

The first operator eventually attempts to extract and store an object of type **CharType** from the associated input stream. The second operator makes a copy of the object, increments the object, and then returns the copy.

The first preincrement operator increments the stored output iterator object, and then returns ***this**.

The second postincrement operator makes a copy of ***this**, increments the stored output iterator object, and then returns the copy.

The constructor stores **first** as the output iterator object.

Example

```
// raw_storage_iterator_op_incr.cpp
// compile with: /EHsc
#include <iostream>
#include <iterator>
#include <memory>
#include <list>
using namespace std;

int main( void )
{
   int *pInt = new int[5];
   std::raw_storage_iterator<int*,int> it( pInt );
   for ( int i = 0; i < 5; i++, it++ ) {
      *it = 2 * i;
};

   for ( int i = 0; i < 5; i++ ) cout << "array " << i << " = " << pInt[i] << endl;;

   delete[] pInt;
}
```

Output

```
array 0 = 0
array 1 = 2
array 2 = 4
array 3 = 6
array 4 = 8
```

Requirements

Header: <memory>

See Also

Reference

raw_storage_iterator Class

Other Resources

raw_storage_iterator Members

<memory> Specializations

For information about the specializations in <memory>, see <memory> Members.

allocator<void> Class

A specialization of the template class allocator to type **void**, defining the types that make sense in this context.

```
template<>
   class allocator<void> {
   typedef void *pointer;
   typedef const void *const_pointer;
   typedef void value_type;
   template<class Other>
      struct rebind;
   allocator( );
   template<class Other>
      allocator(
         const allocator<Other>
      );
   template<class Other>
      allocator<void>& operator=(
         const allocator<Other>
      );
   };
```

Remarks

The class explicitly specializes template class allocator for type *void*. Its constructors and assignment operator behave the same as for the template class, but it defines only the following types:

- const_pointer.
- pointer.
- value_type.
- rebind, a nested template class.

Requirements

Header: <memory>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<memory> Members

<new>

Defines several types and functions that control the allocation and freeing of storage under program control. It also defines components for reporting on storage management errors.

For a list of all members of this header, see <new> Members.

```
#include <new>
```

Remarks

Some of the functions declared in this header are replaceable. The implementation supplies a default version, whose behavior is described in this document. A program can, however, define a function with the same signature to replace the default version at link time. The replacement version must satisfy the requirements described in this document.

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<new> Members

Standard C++ Library Header Files

<new> Members

Reference

Objects

nothrow	Provides an object to be used as an argument for the **nothrow** versions of **new** and **delete**.

Typedefs

new_handler	A type that points to a function suitable for use as a new handler.

Functions

set_new_handler	Installs a user function that is called when new fails in its attempt to allocate memory.

Operators

operator delete	The function called by a delete expression to deallocate storage for individual of objects.
operator delete[]	The function called by a delete expression to deallocate storage for an array of objects.
operator new	The function called by a new expression to allocate storage for individual objects.
operator new[]	The function called by a new expression to allocate storage for an array of objects.

Classes

bad_alloc Class	The class describes an exception thrown to indicate that an allocation request did not succeed.
nothrow_t Class	The class is used as a function parameter to operator new to indicate that the function should return a null pointer to report an allocation failure, rather than throw an exception.

See Also

Reference

<new>

Thread Safety in the Standard C++ Library

<new> Objects

For information about the objects in <new>, see <new> Members.

nothrow (<new>)

Provides an object to be used as an argument for the **nothrow** versions of **new** and **delete**.

```
extern const std::nothrow_t nothrow;
```

Remarks

The object is used as a function argument to match the parameter type std::nothrow_t.

Example

See operator new and operator new[] for examples of how **std::nothrow_t** is used as a function parameter.

Requirements

Header: <new>

See Also

Other Resources

<new> Members

<new> Typedefs

For information about the typedefs in <new>, see <new> Members.

new_handler

The type points to a function suitable for use as a new handler.

```
typedef void ( *new_handler )( );
```

Remarks

This type of handler function is called by **operator new** or **operator new[]** when they cannot satisfy a request for additional storage.

Example

See set_new_handler for an example using **new_handler** as a return value.

Requirements

Header: <new>

See Also

Other Resources

<new> Members

<new> Functions

For information about the functions in <new>, see <new> Members.

set_new_handler

Installs a user function that is to be called when **operator new** fails in its attempt to allocate memory.

```
new_handler set_new_handler(
   new_handler _Pnew
) throw( );
```

Parameters

_Pnew

The new_handler to be installed.

Return Value

0 on the first call and the previous **new_handler** on subsequent calls.

Remarks

The function stores *_Pnew* in a static new handler pointer that it maintains, then returns the value previously stored in the pointer. The new handler is used by operator new(**size_t**).

Example

```
// new_set_new_handler.cpp
// compile with: /EHsc
#include<new>
#include<iostream>

using namespace std;
void __cdecl newhandler( )
{
   cout << "The new_handler is called:" << endl;
   throw bad_alloc( );
   return;
}

int main( )
{
   set_new_handler (newhandler);
   try
   {
      while ( 1 )
      {
         new int[5000000];
         cout << "Allocating 5000000 ints." << endl;
      }
   }
   catch ( exception e )
   {
      cout << e.what( ) << " xxx" << endl;
   }
}
```

Sample Output

```
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
```

```
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
The new_handler is called:
bad allocation
```

Requirements

Header: <new>

See Also

Other Resources

<new> Members

<new> Operators

For information about the operators in <new>, see <new> Members.

operator delete (<new>)

The function called by a delete expression to deallocate storage for individual of objects.

```
void operator delete(
   void* _Ptr
) throw( );
void operator delete(
   void *,
   void *
) throw( );
void operator delete(
   void* _Ptr,
   const std::nothrow_t&
) throw( );
```

Parameters

_Ptr

The pointer whose value is to be rendered invalid by the deletion.

Remarks

The first function is called by a delete expression to render the value of *_Ptr* invalid. The program can define a function with this function signature that replaces the default version defined by the Standard C++ Library. The required behavior is to accept a value of *_Ptr* that is null or that was returned by an earlier call to operator new(**size_t**).

The default behavior for a null value of *_Ptr* is to do nothing. Any other value of *_Ptr* must be a value returned earlier by a call as previously described. The default behavior for such a nonnull value of *_Ptr* is to reclaim storage allocated by the earlier call. It is unspecified under what conditions part or all of such reclaimed storage is allocated by a subsequent call to **operator new**(**size_t**), or to any of **calloc**(**size_t**), **malloc**(**size_t**), or **realloc**(**void***, **size_t**).

The second function is called by a placement delete expression corresponding to a new expression of the form **new**(**std::size_t**). It does nothing.

The third function is called by a placement delete expression corresponding to a new expression of the form **new**(**std::size_t**, **const std::nothrow_t&**). The program can define a function with this function signature that replaces the default version defined by the Standard C++ Library. The required behavior is to accept a value of *_Ptr* that is null or that was returned by an earlier call to **operator new**(**size_t**). The default behavior is to evaluate **delete**(*_Ptr*).

Example

See operator new for an example that use **operator delete**.

Requirements

Header: <new>

See Also

Other Resources

<new> Members

operator delete[] (<new>)

The function called by a delete expression to deallocate storage for an array of objects.

```
void operator delete[](
   void* _Ptr
) throw( );
void operator delete[](
   void *,
   void *
) throw( );
void operator delete[](
   void* _Ptr,
   const std::nothrow_t&
) throw( );
```

Parameters

_Ptr

The pointer whose value is to be rendered invalid by the deletion.

Remarks

The first function is called by an **delete[]** expression to render the value of *_Ptr* invalid. The function is replaceable because the program can define a function with this function signature that replaces the default version defined by the Standard C++ Library. The required behavior is to accept a value of *_Ptr* that is null or that was returned by an earlier call to operator new[](**size_t**). The default behavior for a null value of *_Ptr* is to do nothing. Any other value of *_Ptr* must be a value returned earlier by a call as previously described. The default behavior for such a nonnull value of *_Ptr* is to reclaim storage allocated by the earlier call. It is unspecified under what conditions part or all of such reclaimed storage is allocated by a subsequent call to operator new(**size_t**), or to any of **calloc**(**size_t**), **malloc**(**size_t**), or **realloc**(**void***, **size_t**).

The second function is called by a placement **delete[]** expression corresponding to a **new[]** expression of the form **new[]**(**std::size_t**). It does nothing.

The third function is called by a placement delete expression corresponding to a **new[]** expression of the form **new[]**(**std::size_t**, **const std::nothrow_t&**). The program can define a function with this function signature that replaces the default version defined by the Standard C++ Library. The required behavior is to accept a value of *_Ptr* that is null or that was returned by an earlier call to operator **new[]**(**size_t**). The default behavior is to evaluate **delete[]**(*_Ptr*).

Example

See operator new[] for examples of the use of **operator delete[]**.

Requirements

Header: <new>

See Also

Other Resources

<new> Members

operator new (<new>)

The function called by a new-expression to allocate storage for individual objects.

```
void* operator new(
   std::size_t _Count
) throw(bad_alloc);
void* operator new(
   std::size_t _Count,
   const std::nothrow_t&
) throw( );
void* operator new(
   std::size_t _Count,
   void* _Ptr
) throw( );
```

Parameters

_Count

The number of bytes of storage to be allocated.

_Ptr

The pointer to be returned.

Return Value

A pointer to the lowest byte address of the newly-allocated storage. Or *_Ptr*.

Remarks

The first function is called by a new expression to allocate *_Count* bytes of storage suitably aligned to represent any object of that size. The program can define an alternate function with this function signature that replaces the default version defined by the Standard C++ Library and so is replaceable.

The required behavior is to return a nonnull pointer only if storage can be allocated as requested. Each such allocation yields a pointer to storage disjoint from any other allocated storage. The order and contiguity of storage allocated by successive calls is unspecified. The initial stored value is unspecified. The returned pointer points to the start (lowest byte address) of the allocated storage. If count is zero, the value returned does not compare equal to any other value returned by the function.

The default behavior is to execute a loop. Within the loop, the function first attempts to allocate the requested storage. Whether the attempt involves a call to **malloc**(**size_t**) is unspecified. If the attempt is successful, the function returns a pointer to the allocated storage. Otherwise, the function calls the designated new handler. If the called function returns, the loop repeats. The loop terminates when an attempt to allocate the requested storage is successful or when a called function does not return.

The required behavior of a new handler is to perform one of the following operations:

- Make more storage available for allocation and then return.
- Call either **abort** or **exit**(**int**).
- Throw an object of type **bad_alloc.**

The default behavior of a new handler is to throw an object of type **bad_alloc**. A null pointer designates the default new handler.

The order and contiguity of storage allocated by successive calls to **operator new**(**size_t**) is unspecified, as are the initial values stored there.

The second function is called by a placement new expression to allocate *_Count* bytes of storage suitably aligned to represent any object of that size. The program can define an alternate function with this function signature that replaces the default version defined by the Standard C++ Library and so is replaceable.

The default behavior is to return **operator new**(*_Count*) if that function succeeds. Otherwise, it returns a null pointer.

The third function is called by a placement **new** expression, of the form **new** (*args*) T. Here, *args* consists of a single object pointer. This can be useful for constructing an object at a known address. The function returns *_Ptr*.

For information on throwing or nonthrowing behavior of new, see The new and delete Operators.

Example

```
// new_op_new.cpp
// compile with: /EHsc
#include<new>
#include<iostream>

using namespace std;

class MyClass
{
public:
   MyClass( )
   {
      cout << "Construction MyClass." << this << endl;
   };

   ~MyClass( )
   {
      imember = 0; cout << "Destructing MyClass." << this << endl;
   };
   int imember;
};

int main( )
{
   // The first form of new delete
   MyClass* fPtr = new MyClass;
   delete fPtr;

   // The second form of new delete
   char x[sizeof( MyClass )];
   MyClass* fPtr2 = new( &x[0] ) MyClass;
   fPtr2 -> ~MyClass();
   cout << "The address of x[0] is : " << ( void* )&x[0] << endl;

   // The third form of new delete
   MyClass* fPtr3 = new( nothrow ) MyClass;
   delete fPtr3;
}
```

Example Output

```
Construction MyClass.00310840
Destructing MyClass.00310840
Construction MyClass.0012FED4
Destructing MyClass.0012FED4
The address of x[0] is : 0012FED4
Construction MyClass.00310840
Destructing MyClass.00310840
```

Requirements

Header: <new>

See Also

Reference

new operator (STL Samples)

nothrow_t Class

Other Resources

<new> Members

operator new[] (<new>)

The allocation function called by a new expression to allocate storage for an array of objects.

```
void *operator new[](
   std::size_t _Count
)
   throw(std::bad_alloc);
void *operator new[](
   std::size_t _Count,
   const std::nothrow_t&
) throw( );
void *operator new[](
   std::size_t _Count,
   void* _Ptr
) throw( );
```

Parameters

_Count

The number of bytes of storage to be allocated for the array object.

_Ptr

The pointer to be returned.

Return Value

A pointer to the lowest byte address of the newly-allocated storage. Or *_Ptr*.

Remarks

The first function is called by a **new[]** expression to allocate *_Count* bytes of storage suitably aligned to represent any array object of that size or smaller. The program can define a function with this function signature that replaces the default version defined by the Standard C++ Library. The required behavior is the same as for operator new(**size_t**). The default behavior is to return **operator new**(*_Count*).

The second function is called by a placement **new[]** expression to allocate *_Count* bytes of storage suitably aligned to represent any array object of that size. The program can define a function with this function signature that replaces the default version defined by the Standard C++ Library. The default behavior is to return **operator new**(*_Count*) if that function succeeds. Otherwise, it returns a null pointer.

The third function is called by a placement **new[]** expression, of the form **new** (*args*) **T**[**N**]. Here, *args* consists of a single object pointer. The function returns *_Ptr*.

For information on throwing or nonthrowing behavior of new, see The new and delete Operators.

Example

```
// new_op_alloc.cpp
// compile with: /EHsc
#include <new>
#include <iostream>

using namespace std;

class MyClass {
public:
   MyClass() {
      cout << "Construction MyClass." << this << endl;
   };

   ~MyClass() {
      imember = 0; cout << "Destructing MyClass." << this << endl;
      };
   int imember;
```

```
};

int main() {
   // The first form of new delete
   MyClass* fPtr = new MyClass[2];
   delete[ ] fPtr;

   // The second form of new delete
   char x[2 * sizeof( MyClass ) + sizeof(int)];

   MyClass* fPtr2 = new( &x[0] ) MyClass[2];
   fPtr2[1].~MyClass();
   fPtr2[0].~MyClass();
   cout << "The address of x[0] is : " << ( void* )&x[0] << endl;

   // The third form of new delete
   MyClass* fPtr3 = new( nothrow ) MyClass[2];
   delete[ ] fPtr3;
}
```

Sample Output

```
Construction MyClass.00311AEC
Construction MyClass.00311AF0
Destructing MyClass.00311AF0
Destructing MyClass.00311AEC
Construction MyClass.0012FED4
Construction MyClass.0012FED8
Destructing MyClass.0012FED8
Destructing MyClass.0012FED4
The address of x[0] is : 0012FED0
Construction MyClass.00311AEC
Construction MyClass.00311AF0
Destructing MyClass.00311AF0
Destructing MyClass.00311AEC
```

Requirements

Header: <new>

See Also

Reference

nothrow_t Class

Other Resources

<new> Members

<new> Classes

For information about the classes in <new>, see <new> Members.

bad_alloc Class

The class describes an exception thrown to indicate that an allocation request did not succeed.

```
class bad_alloc : public exception {
   bad_alloc(const char *_Message): exception(_Message) {}
   bad_alloc(): exception("bad allocation", 1) {}
   virtual ~bad_alloc() {}
};
```

Remarks

An instance of **bad_alloc** can be constructed from a message or from a message string with no memory allocation.

The value returned by **what** is an implementation-defined C string. None of the member functions throw any exceptions.

Requirements

Header: <new>

Example

```
// bad_alloc.cpp
// compile with: /EHsc
#include<new>
#include<iostream>
using namespace std;

int main() {
   char* ptr;
   try {
      ptr = new char[(~unsigned int((int)0)/2) - 1];
      delete[] ptr;
   }
   catch( bad_alloc &ba) {
      cout << ba.what( ) << endl;
   }
}
```

Sample Output

```
bad allocation
```

Requirements

Header: <new>

See Also

Reference

exception Class

Thread Safety in the Standard C++ Library

Other Resources

<new> Members

nothrow_t Class

The class is used as a function parameter to operator new to indicate that the function should return a null pointer to report an allocation failure, rather than throw an exception.

```
class std::nothrow_t {};
```

Remarks

The class helps the compiler to select the correct version of the constructor. nothrow (<new>) is a synonym for objects of type **std::nothrow_t**.

Example

See operator new and operator new[] for examples of how **std::nothrow_t** is used as a function parameter.

Requirements

Header: <new>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<new> Members

<numeric>

Defines container template functions that perform algorithms provided for numerical processing.

For a list of all members of this header, see <numeric> Members.

```
#include <numeric>
```

Remarks

The algorithms are similar to the Standard Template Library (STL) algorithms and are fully compatible with the STL but are part of the C++ Standard Library rather than the STL. Like the STL algorithms, they are generic because they can operate on a variety of data structures. The data structures that they can operate on include STL container classes, such as vector and list, and program-defined data structures and arrays of elements that satisfy the requirements of a particular algorithm. The algorithms achieve this level of generality by accessing and traversing the elements of a container indirectly through iterators. The algorithms process iterator ranges that are typically specified by their beginning or ending positions. The ranges referred to must be valid in the sense that all pointers in the ranges must be dereferenceable and within the sequences of each range, the last position must be reachable from the first by means of incrementation.

The algorithms extend the actions supported by the operations and member functions of each of the STL containers and allow working, for example, with different types of container objects at the same time.

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<numeric> Members

Standard C++ Library Header Files

<numeric> Members

Reference

Functions

accumulate	Computes the sum of all the elements in a specified range including some initial value by computing successive partial sums or computes the result of successive partial results similarly obtained from using a specified binary operation other than the sum.
adjacent_difference	Computes the successive differences between each element and its predecessor in an input range and outputs the results to a destination range or computes the result of a generalized procedure where the difference operation is replaced by another, specified binary operation.
checked_adjacent_difference	Same as **adjacent_difference**, but enforces the use of a checked iterator on the output iterator.
checked_partial_sum	Same as **partial_sum**, but enforces the use of a checked iterator on the output iterator.
inner_product	Computes the sum of the element-wise product of two ranges and adds it to a specified initial value or computes the result of a generalized procedure where the sum and product binary operations are replaced by other specified binary operations.
partial_sum	Computes a series of sums in an input range from the first element through the *i*th element and stores the result of each such sum in *i*th element of a destination range or computes the result of a generalized procedure where the sum operation is replaced by another specified binary operation.
unchecked_adjacent_difference	Same as **adjacent_difference**, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.
unchecked_partial_sum	Same as **partial_sum**, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.

See Also

Reference

<numeric>

Thread Safety in the Standard C++ Library

Standard Template Library

<numeric> Functions

For information about the functions in <numeric>, see <numeric> Members.

See Also

Reference

Standard Template Library

accumulate

Computes the sum of all the elements in a specified range including some initial value by computing successive partial sums or computes the result of successive partial results similarly obtained from using a specified binary operation other than the sum.

```
template<class InputIterator, class Type>
   Type accumulate(
      InputIterator _First,
      InputIterator _Last,
      Type _Val
   );
template<class InputIterator, class Type, class BinaryOperation>
   Type accumulate(
      InputIterator _First,
      InputIterator _Last,
      Type _Val,
      BinaryOperation _Binary_op
   );
```

Parameters

_First

An input iterator addressing the first element in the range to be summed or combined according to a specified binary operation.

_Last

An input iterator addressing the last element in the range to be summed or combined according to a specified binary operation that is one position beyond the final element actually included in the iterated accumulation.

_Val

An initial value to which each element is in turn added or combined with according to a specified binary operation.

_Binary_op

The binary operation that is to be applied to the each element in the specified range and the result of its previous applications.

Return Value

The sum of *_Val* and all the elements in the specified range for the first template function, or, for the second template function, the result of applying the binary operation specified, instead of the sum operation, to (*PartialResult*, **Iter*), where *PartialResult* is the result of previous applications of the operation and *Iter* is an iterator pointing to an element in the range.

Remarks

The initial value insures that there will be a well-defined result when the range is empty, in which case *_Val* is returned. The binary operation does not need to be associative or commutative. The result is initialized to the initial value *_Val* and then *result* = *_Binary_op* (*result*, ***Iter*) is calculated iteratively through the range, where *Iter* is an iterator pointing to successive element in the range. The range must be valid and the complexity is linear with the size of the range. The return type of the binary operator must be convertible to **Type** to ensure closure during the iteration.

Example

```
// numeric_accum.cpp
// compile with: /EHsc
#include <vector>
#include <numeric>
#include <functional>
#include <iostream>

int main( )
{
   using namespace std;
```

```
vector <int> v1, v2(20);
vector <int>::iterator iter1, iter2;

int i;
for (i = 1; i < 21; i++)
{
   v1.push_back(i);
}

cout << "The original vector v1 is:\n ( " ;
for (iter1 = v1.begin(); iter1 != v1.end(); iter1++)
   cout << *iter1 << " ";
cout << ")." << endl;

// The first member function for the accumulated sum
int total;
total = accumulate(v1.begin(), v1.end(), 0);

cout << "The sum of the integers from 1 to 20 is: "
     << total << "." << endl;

// Constructing a vector of partial sums
int j = 0, partotal;
for (iter1 = v1.begin(); iter1 != v1.end(); iter1++)
{
   partotal = accumulate(v1.begin(), iter1 + 1, 0);
   v2[j] = partotal;
   j++;
}

cout << "The vector of partial sums is:\n ( " ;
for (iter2 = v2.begin(); iter2 != v2.end(); iter2++)
   cout << *iter2 << " ";
cout << ")." << endl << endl;

// The second member function for the accumulated product
vector <int> v3, v4(10);
vector <int>::iterator iter3, iter4;

int s;
for (s = 1; s < 11; s++)
{
   v3.push_back(s);
}

cout << "The original vector v3 is:\n ( " ;
for (iter3 = v3.begin(); iter3 != v3.end(); iter3++)
   cout << *iter3 << " ";
cout << ")." << endl;

int ptotal;
ptotal = accumulate(v3.begin(), v3.end(), 1, multiplies<int>());

cout << "The product of the integers from 1 to 10 is: "
     << ptotal << "." << endl;

// Constructing a vector of partial products
int k = 0, ppartotal;
for (iter3 = v3.begin(); iter3 != v3.end(); iter3++) {
   ppartotal = accumulate(v3.begin(), iter3 + 1, 1, multiplies<int>());
   v4[k] = ppartotal;
   k++;
}

cout << "The vector of partial products is:\n ( " ;
for (iter4 = v4.begin(); iter4 != v4.end(); iter4++)
   cout << *iter4 << " ";
```

```
    cout << "))." << endl;
  }
```

Output

```
The original vector v1 is:
 ( 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ).
The sum of the integers from 1 to 20 is: 210.
The vector of partial sums is:
 ( 1 3 6 10 15 21 28 36 45 55 66 78 91 105 120 136 153 171 190 210 ).

The original vector v3 is:
 ( 1 2 3 4 5 6 7 8 9 10 ).
The product of the integers from 1 to 10 is: 3628800.
The vector of partial products is:
 ( 1 2 6 24 120 720 5040 40320 362880 3628800 ).
```

Requirements

Header: <numeric>

See Also

Reference

accumulate, copy, and vector::push_back

Standard Template Library

Other Resources

<numeric> Members

adjacent_difference

Computes the successive differences between each element and its predecessor in an input range and outputs the results to a destination range or computes the result of a generalized procedure where the difference operation is replaced by another, specified binary operation.

```
template<class InputIterator, class OutIterator>
   OutputIterator adjacent_difference(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result
   );

template<class InputIterator, class OutIterator, class BinaryOperation>
   OutputIterator adjacent_difference(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      BinaryOperation _Binary_op
   );
```

Parameters

_First

An input iterator addressing the first element in the input range whose elements are to be differenced with their respective predecessors or where the pair of values is to be operated on by another specified binary operation.

_Last

An input iterator addressing the last element in the input range whose elements are to be differenced with their respective predecessors or where the pair of values is to be operated on by another specified binary operation.

_Result

An output iterator addressing the first element a destination range where the series of differences or the results of the specified operation is to be stored.

_Binary_op

The binary operation that is to be applied in the generalized operation replacing the operation of subtraction in the differencing procedure.

Return Value

An output iterator addressing the end of the destination range: *_Result* + (*_Last* - *_First*).

Remarks

The output iterator *_Result* is allowed to be the same iterator as the input iterator *_First,* so that **adjacent_difference**s may be computed in place.

For a sequence of values *a*1, *a*2, *a*3, in an input range, the first template function stores successive **partial_difference**s *a*1, *a*2 - *a*1, a3 – *a*2, in the destination range.

For a sequence of values *a*1, *a*2, *a*3, in an input range, the second template function stores successive **partial_difference**s *a*1, *a*2 *_Binary_op a*1, *a*3 *_Binary_op a*2, in the destination range.

The binary operation *_Binary_op* is not required to be either associative or commutative, because the order of operations applies is completely specified.

adjacent_difference has two related forms:

- checked_adjacent_difference
- unchecked_adjacent_difference

If you pass a checked iterator to one of the forms of **adjacent_difference**, you get checked iterator behavior. If you pass an

unchecked iterator, you get unchecked behavior. For more information, see Checked Iterators.

Example

```
// numeric_adj_diff.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <numeric>
#include <functional>
#include <iostream>

int main( )
{
   using namespace std;

   vector<int> V1( 10 ), V2( 10 );
   vector<int>::iterator VIter1, VIter2, VIterend, VIterend2;

   list <int> L1;
   list <int>::iterator LIter1, LIterend, LIterend2;

   int t;
   for ( t = 1 ; t <= 10 ; t++ )
   {
      L1.push_back( t * t );
   }

   cout << "The input list L1 is:\n ( " ;
   for ( LIter1 = L1.begin( ) ; LIter1 != L1.end( ) ; LIter1++ )
      cout << *LIter1 << " ";
   cout << ")." << endl;

   // The first member function for the adjacent_differences of
   // elements in a list output to a vector
   VIterend = adjacent_difference ( L1.begin ( ) , L1.end ( ) ,
      V1.begin ( ) );

   cout << "Output vector containing adjacent_differences is:\n ( " ;
   for ( VIter1 = V1.begin( ) ; VIter1 != VIterend ; VIter1++ )
      cout << *VIter1 << " ";
   cout << ")." << endl;

   // The second member function used to compute
   // the adjacent products of the elements in a list
   VIterend2 = adjacent_difference ( L1.begin ( ) , L1.end ( ) , V2.begin ( ) ,
      multiplies<int>( ) );

   cout << "The output vector with the adjacent products is:\n ( " ;
   for ( VIter2 = V2.begin( ) ; VIter2 != VIterend2 ; VIter2++ )
      cout << *VIter2 << " ";
   cout << ")." << endl;

   // Computation of adjacent_differences in place
   LIterend2 = adjacent_difference ( L1.begin ( ) , L1.end ( ) , L1.begin ( ) );
   cout << "In place output adjacent_differences in list L1 is:\n ( " ;
   for ( LIter1 = L1.begin( ) ; LIter1 != LIterend2 ; LIter1++ )
      cout << *LIter1 << " ";
   cout << ")." << endl;
}
```

Output

```
The input list L1 is:
 ( 1 4 9 16 25 36 49 64 81 100 ).
Output vector containing adjacent_differences is:
 ( 1 3 5 7 9 11 13 15 17 19 ).
The output vector with the adjacent products is:
 ( 1 4 36 144 400 900 1764 3136 5184 8100 ).
```

```
In place output adjacent_differences in list L1 is:
 ( 1 3 5 7 9 11 13 15 17 19 ).
```

Requirements

Header: <numeric>

See Also

Reference

adjacent_difference and vector::push_back

Standard Template Library

Other Resources

<numeric> Members

checked_adjacent_difference

Same as adjacent_difference, but enforces the use of a checked iterator on the output iterator. **checked_adjacent_difference** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator, class OutIterator>
   OutputIterator checked_adjacent_difference(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result
   );

template<class InputIterator, class OutputIterator, class BinaryOperation>
   OutputIterator checked_adjacent_difference(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      BinaryOperation _Binary_op
   );
```

Parameters

_First

An input iterator addressing the first element in the input range whose elements are to be differenced with their respective predecessors or where the pair of values is to be operated on by another specified binary operation.

_Last

An input iterator addressing the last element in the input range whose elements are to be differenced with their respective predecessors or where the pair of values is to be operated on by another specified binary operation.

_Result

An output iterator addressing the first element a destination range where the series of differences or the results of the specified operation is to be stored.

_Binary_op

The binary operation that is to be applied in the generalized operation replacing the operation of subtraction in the differencing procedure.

Return Value

An output iterator addressing the end of the destination range: *_Result* + (*_Last* - *_First*).

Remarks

See adjacent_difference for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <numeric>

See Also

Reference

Standard Template Library

Other Resources

<numeric> Members

checked_partial_sum

Same as partial_sum, but enforces the use of a checked iterator on the output iterator. **checked_partial_sum** is defined in the **stdext** namespace.

Note

This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator, class OutputIterator>
   OutputIterator checked_partial_sum(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result
   );

template<class InputIterator, class OutputIterator, class BinaryOperation>
   OutputIterator checked_partial_sum(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      BinaryOperation _Binary_op
   );
```

Parameters

_First

An input iterator addressing the first element in the range to be partially summed or combined according to a specified binary operation.

_Last

An input iterator addressing the last element in the range to be partially summed or combined according to a specified binary operation that is one position beyond the final element actually included in the iterated accumulation.

_Result

An output iterator addressing the first element a destination range where the series of partial sums or the results of the specified operation is to be stored.

_Binary_op

The binary operation that is to be applied in the generalized operation replacing the operation of sum in the partial sum procedure.

Return Value

An output iterator addressing the end of the destination range: *_Result* + (*_Last* - *_First*),

Remarks

See partial_sum for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <numeric>

See Also

Reference

Standard Template Library

Other Resources

<numeric> Members

inner_product

Computes the sum of the element-wise product of two ranges and adds it to a specified initial value or computes the result of a generalized procedure where the sum and product binary operations are replaced by other specified binary operations.

```
template<class InputIterator1, class InputIterator2, class Type>
   Type inner_product(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      Type _Val
   );

template<class InputIterator1, class InputIterator2, class Type,
   class BinaryOperation1, class BinaryOperation2>
   Type inner_product(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      Type _Val,
      BinaryOperation1 _Binary_op1,
      BinaryOperation2 _Binary_op2
   );
```

Parameters

_First1

An input iterator addressing the first element in the first range whose inner product or generalized inner product with the second range is to be computed.

_Last1

An input iterator addressing the last element in the first range whose inner product or generalized inner product with the second range is to be computed.

_First2

An input iterator addressing the first element in the second range whose inner product or generalized inner product with the first range is to be computed.

_Val

An initial value to which the inner product or generalized inner product between the ranges is to be added.

_Binary_op1

The binary operation that replaces the inner product operation of sum applied to the element-wise products in the generalization of the inner product.

_Binary_op2

The binary operation that replaces the inner product element-wise operation of multiply in the generalization of the inner product.

Return Value

The first member function returns the sum of the element-wise products and adds to it the specified initial value. So for ranges of values *a*i and *b*i, it returns:

$_{Val} + (a_1 * b_1) + (a_2 * b_2) +$

by iteratively replacing *_Val* with *_Val* + (*a_i * *b_i).

The second member function returns:

_Val _Binary_op1 (a_1 *_Binary_op2* b_1) *_Binary_op1* (a_2 *_Binary_op2* b_2) *_Binary_op1*

by iteratively replacing *_Val* with *_Val _Binary_op1* (*a_i *_Binary_op2* *b_i).

Remarks

The initial value ensures that there will be a well-defined result when the range is empty, in which case *_Val* is returned. The binary operations do not need to be associative or commutative. The range must be valid and the complexity is linear with the size of the range. The return type of the binary operator must be convertible to **Type** to ensure closure during the iteration.

Example

```
// numeric_inner_prod.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <numeric>
#include <functional>
#include <iostream>

int main()
{
   using namespace std;

   vector <int> v1, v2(7), v3(7);
   vector <int>::iterator iter1, iter2, iter3;

   int i;
   for (i = 1; i <= 7; i++)
   {
      v1.push_back(i);
   }

   cout << "The original vector v1 is:\n ( " ;
   for (iter1 = v1.begin(); iter1 != v1.end(); iter1++)
      cout << *iter1 << " ";
   cout << ")." << endl;

   list <int> l1, l2(7);
   list <int>::iterator lIter1, lIter2;

   int t;
   for (t = 1; t <= 7; t++)
   {
      l1.push_back(t);
   }

   cout << "The original list l1 is:\n ( " ;
   for (lIter1 = l1.begin(); lIter1 != l1.end(); lIter1++)
      cout << *lIter1 << " ";
   cout << ")." << endl;

   // The first member function for the inner product
   int inprod;
   inprod = inner_product(v1.begin(), v1.end(), l1.begin(), 0);

   cout << "The inner_product of the vector v1 and the list l1 is: "
        << inprod << "." << endl;

   // Constructing a vector of partial inner_products between v1 & l1
   int j = 0, parinprod;
   for (iter1 = v1.begin(); iter1 != v1.end(); iter1++) {
      parinprod = inner_product(v1.begin(), iter1 + 1, l1.begin(), 0);
      v2[j] = parinprod;
      j++;
   }

   cout << "Vector of partial inner_products between v1 & l1 is:\n ( " ;
   for (iter2 = v2.begin(); iter2 != v2.end(); iter2++)
      cout << *iter2 << " ";
   cout << ")." << endl << endl;
```

```
    // The second member function used to compute
    // the product of the element-wise sums
    int inprod2;
    inprod2 = inner_product (v1.begin(), v1.end(),
       l1.begin(), 1, multiplies<int>(), plus<int>());

    cout << "The sum of the element-wise products of v1 and l1 is: "
         << inprod2 << "." << endl;

    // Constructing a vector of partial sums of element-wise products
    int k = 0, parinprod2;
    for (iter1 = v1.begin(); iter1 != v1.end(); iter1++)
    {
       parinprod2 =
          inner_product(v1.begin(), iter1 + 1, l1.begin(), 1,
          multiplies<int>(), plus<int>());
       v3[k] = parinprod2;
       k++;
    }

    cout << "Vector of partial sums of element-wise products is:\n ( " ;
    for (iter3 = v3.begin(); iter3 != v3.end(); iter3++)
       cout << *iter3 << " ";
    cout << "))." << endl << endl;
 }
```

Output

```
The original vector v1 is:
 ( 1 2 3 4 5 6 7 ).
The original list l1 is:
 ( 1 2 3 4 5 6 7 ).
The inner_product of the vector v1 and the list l1 is: 140.
Vector of partial inner_products between v1 & l1 is:
 ( 1 5 14 30 55 91 140 ).

The sum of the element-wise products of v1 and l1 is: 645120.
Vector of partial sums of element-wise products is:
 ( 2 8 48 384 3840 46080 645120 ).
```

Requirements

Header: <numeric>

See Also

Reference

inner_product (STL Samples)

Standard Template Library

Other Resources

<numeric> Members

partial_sum

Computes a series of sums in an input range from the first element through the *i*th element and stores the result of each such sum in the *i*th element of a destination range or computes the result of a generalized procedure where the sum operation is replaced by another specified binary operation.

```
template<class InputIterator, class OutIt>
   OutputIterator partial_sum(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result
   );

template<class InputIterator, class OutIt, class Fn2>
   OutputIterator partial_sum(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      BinaryOperation _Binary_op
   );
```

Parameters

_First

An input iterator addressing the first element in the range to be partially summed or combined according to a specified binary operation.

_Last

An input iterator addressing the last element in the range to be partially summed or combined according to a specified binary operation that is one position beyond the final element actually included in the iterated accumulation.

_Result

An output iterator addressing the first element a destination range where the series of partial sums or the results of the specified operation is to be stored.

_Binary_op

The binary operation that is to be applied in the generalized operation replacing the operation of sum in the partial sum procedure.

Return Value

An output iterator addressing the end of the destination range: *_Result* + (*_Last* - *_First*),

Remarks

The output iterator *_Result* is allowed to be the same iterator as the input iterator *_First*, so that partial sums may be computed in place.

For a sequence of values a_1, a_2, a_3, in an input range, the first template function stores successive partial sums in the destination range, where the *i*th element is given by (((a_1 + a_2) + a_3) a_i).

For a sequence of values a_1, a_2, a_3, in an input range, the second template function stores successive partial sums in the destination range, where the ith element is given by (((a_1 *_Binary_op* a_2) *_Binary_op* a_3) a_i).

The binary operation *_Binary_op* is not required to be either associative or commutative, because the order of operations applies is completely specified.

partial_sum has two related forms:

- checked_partial_sum
- unchecked_partial_sum

If you pass a checked iterator to one of the forms of **partial_sum**, you get checked iterator behavior. If you pass an unchecked iterator, you get unchecked behavior. For more information, see Checked Iterators.

Example

```
// numeric_partial_sum.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <numeric>
#include <functional>
#include <iostream>

int main( )
{
   using namespace std;
   vector<int> V1( 10 ), V2( 10 );
   vector<int>::iterator VIter1, VIter2, VIterend, VIterend2;

   list <int> L1;
   list <int>::iterator LIter1, LIterend;

   int t;
   for ( t = 1 ; t <= 10 ; t++ )
   {
      L1.push_back( t );
   }

   cout << "The input list L1 is:\n ( " ;
   for ( LIter1 = L1.begin( ) ; LIter1 != L1.end( ) ; LIter1++ )
      cout << *LIter1 << " ";
   cout << "))." << endl;

   // The first member function for the partial sums of
   // elements in a list output to a vector
   VIterend = partial_sum ( L1.begin ( ) , L1.end ( ) ,
      V1.begin ( ) );

   cout << "The output vector containing the partial sums is:\n ( " ;
   for ( VIter1 = V1.begin( ) ; VIter1 != VIterend ; VIter1++ )
      cout << *VIter1 << " ";
   cout << "))." << endl;

   // The second member function used to compute
   // the partial product of the elements in a list
   VIterend2 = partial_sum ( L1.begin ( ) , L1.end ( ) , V2.begin ( ) ,
      multiplies<int>( ) );

   cout << "The output vector with the partial products is:\n ( " ;
   for ( VIter2 = V2.begin( ) ; VIter2 != VIterend2 ; VIter2++ )
      cout << *VIter2 << " ";
   cout << "))." << endl;

   // Computation of partial sums in place
   LIterend = partial_sum ( L1.begin ( ) , L1.end ( ) , L1.begin ( ) );
   cout << "The in place output partial_sum list L1 is:\n ( " ;
   for ( LIter1 = L1.begin( ) ; LIter1 != LIterend ; LIter1++ )
      cout << *LIter1 << " ";
   cout << "))." << endl;
}
```

Output

```
The input list L1 is:
 ( 1 2 3 4 5 6 7 8 9 10 ).
The output vector containing the partial sums is:
 ( 1 3 6 10 15 21 28 36 45 55 ).
The output vector with the partial products is:
```

```
 ( 1 2 6 24 120 720 5040 40320 362880 3628800 ).
The in place output partial_sum list L1 is:
 ( 1 3 6 10 15 21 28 36 45 55 ).
```

Requirements

Header: <numeric>

See Also

Reference

partial_sum (STL Samples)

Standard Template Library

Other Resources

<numeric> Members

unchecked_adjacent_difference

Same as adjacent_difference, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined. **unchecked_adjacent_difference** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator, class OutIterator>
   OutputIterator unchecked_adjacent_difference(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result
   );

template<class InputIterator, class OutputIterator, class BinaryOperation>
   OutputIterator unchecked_adjacent_difference(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      BinaryOperation _Binary_op
   );
```

Parameters

_First

An input iterator addressing the first element in the input range whose elements are to be differenced with their respective predecessors or where the pair of values is to be operated on by another specified binary operation.

_Last

An input iterator addressing the last element in the input range whose elements are to be differenced with their respective predecessors or where the pair of values is to be operated on by another specified binary operation.

_Result

An output iterator addressing the first element a destination range where the series of differences or the results of the specified operation is to be stored.

_Binary_op

The binary operation that is to be applied in the generalized operation replacing the operation of subtraction in the differencing procedure.

Return Value

An output iterator addressing the end of the destination range: *_Result* + (*_Last* - *_First*).

Remarks

See adjacent_difference for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <numeric>

See Also

Reference

Standard Template Library

Other Resources

<numeric> Members

unchecked_partial_sum

Same as partial_sum, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined. **unchecked_partial_sum** is defined in the **stdext** namespace.

Note
This algorithm is a Microsoft extension to the Standard C++ Library. Code implemented using this algorithm will not be port able.

```
template<class InputIterator, class OutIterator>
   OutputIterator unchecked_partial_sum(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result
   );

template<class InputIterator, class OutIterator, class BinaryOperation>
   OutputIterator unchecked_partial_sum(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      BinaryOperation _Binary_op
   );
```

Parameters

_First

An input iterator addressing the first element in the range to be partially summed or combined according to a specified binary operation.

_Last

An input iterator addressing the last element in the range to be partially summed or combined according to a specified binary operation that is one position beyond the final element actually included in the iterated accumulation.

_Result

An output iterator addressing the first element a destination range where the series of partial sums or the results of the specified operation is to be stored.

_Binary_op

The binary operation that is to be applied in the generalized operation replacing the operation of sum in the partial sum procedure.

Return Value

An output iterator addressing the end of the destination range: *_Result* + (*_Last* - *_First*),

Remarks

See partial_sum for a code sample.

For more information on checked iterators, see Checked Iterators.

Requirements

Header: <numeric>

See Also

Reference

Standard Template Library

Other Resources

<numeric> Members

<ostream>

Defines the template class basic_ostream, which mediates insertions for the iostreams. The header also defines several related manipulators. (This header is typically included for you by another of the iostreams headers. You rarely need to include it directly.)

For a list of all members of this header, see <ostream> Members.

```
#include <ostream>
```

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

<ostream> Members
Standard C++ Library Header Files

<ostream> Members

Reference

Typedefs

ostream	Creates a type from basic_ostream that is specialized on **char** and **char_traits** specialized on **char**.
wostream	Creates a type from basic_ostream that is specialized on **wchar_t** and **char_traits** specialized on **wchar_t**.

Manipulators

endl	Terminates a line and flushes the buffer.
ends	Terminates a string.
flush	Flushes the buffer.

Operators

operator<<	Writes various types to the stream.

Classes

basic_ostream	The template class describes an object that controls insertion of elements and encoded objects into a stream buffer.

See Also

Reference

<ostream>
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

<ostream> Typedefs

For information about the typedefs in <ostream>, see <ostream> Members.

See Also

Reference

iostream Programming

iostreams Conventions

ostream

Creates a type from basic_ostream that is specialized on **char** and **char_traits** specialized on **char**.

```
typedef basic_ostream<char, char_traits<char> > ostream;
```

Remarks

The type is a synonym for template class basic_ostream, specialized for elements of type **char** with default character traits.

Requirements

Header: <ostream>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ostream> Members

wostream

Creates a type from basic_ostream that is specialized on **wchar_t** and **char_traits** specialized on **wchar_t**.

```
typedef basic_ostream<wchar_t, char_traits<wchar_t> > wostream;
```

Remarks

The type is a synonym for template class basic_ostream, specialized for elements of type **wchar_t** with default character traits.

Requirements

Header: <ostream>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ostream> Members

<ostream> Manipulators

For information about the manipulators in <ostream>, see <ostream> Members.

See Also

Reference

iostream Programming

iostreams Conventions

endl

Terminates a line and flushes the buffer.

```
template class<_Elem, _Tr>
   basic_ostream<_Elem, _Tr>& endl(
      basic_ostream<_Elem, _Tr>& _Ostr
      );
```

Parameters

_Elem

The element type.

_Ostr

An object of type **basic_ostream**.

_Tr

Character traits.

Return Value

An object of type **basic_ostream**.

Remarks

The manipulator calls *_Ostr*.put(*_Ostr*. widen('**\n**')), and then calls *_Ostr*.flush. It returns *_Ostr*.

Example

```
// ostream_endl.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   cout << "testing" << endl;
}
```

Output

```
testing
```

Requirements

Header: <ostream>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ostream> Members

ends (Standard C++ Library)

Terminates a string.

```
template class<_Elem, _Tr>
   basic_ostream<_Elem, _Tr>& ends(
      basic_ostream<_Elem, _Tr>& _Ostr
);
```

Parameters

_Elem

The element type.

_Ostr

An object of type **basic_ostream**.

_Tr

Character traits.

Return Value

An object of type **basic_ostream**.

Remarks

The manipulator calls *_Ostr*.put(*_Elem*('**\0**')). It returns *_Ostr*.

Example

```
// ostream_ends.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   cout << "a";
   cout << "b" << ends;
   cout << "c" << endl;
}
```

Sample Output

```
ab c
```

Requirements

Header: <ostream>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ostream> Members

flush (Standard C++ Library)

Flushes the buffer.

```
template class<_Elem, _Tr>
   basic_ostream<_Elem, _Tr>&
   flush(basic_ostream<_Elem, _Tr>& _Ostr
   );
```

Parameters

_Elem

The element type.

_Ostr

An object of type **basic_ostream**.

_Tr

Character traits.

Return Value

An object of type **basic_ostream**.

Remarks

The manipulator calls *_Ostr*.flush. It returns *_Ostr*.

Example

```
// ostream_flush.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   cout << "testing" << flush;
}
```

Output

```
testing
```

Requirements

Header: <ostream>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<ostream> Members

<ostream> Operators

For more information about the operators in <ostream>, see <ostream> Members.

See Also

Reference

iostream Programming

iostreams Conventions

operator<< (<ostream>)

Writes various types to the stream.

```
template<class _Elem, class _Tr>
   basic_ostream<Elem, _Tr>& operator<<(
      basic_ostream<Elem, _Tr>& _Ostr,
      const Elem *_Str
      );
template<class _Elem, class _Tr>
   basic_ostream<Elem, _Tr>& operator<<(
      basic_ostream<Elem, _Tr>& _Ostr,
      Elem _Ch
      );
template<class _Elem, class _Tr>
   basic_ostream<Elem, _Tr>& operator<<(
      basic_ostream<Elem, _Tr>& _Ostr,
      const char *_Str
      );
template<class _Elem, class _Tr>
   basic_ostream<Elem, _Tr>& operator<< (
      basic_ostream<Elem, _Tr>& _Ostr,
      char _Ch
      );
template<class _Tr>
   basic_ostream<char, _Tr>& operator<< (
      basic_ostream<char, _Tr>& _Ostr,
      const char *_Str
      );
template<class _Tr>
   basic_ostream<char, _Tr>& operator<< (
      basic_ostream<char, _Tr>& _ostr,
      char _Ch
      );
template<class _Tr>
   basic_ostream<char, _Tr>& operator<<(
      basic_ostream<char, _Tr>& _Ostr,
      const signed char *_Str
      );
template<class _Tr>
   basic_ostream<char, _Tr>& operator<<(
      basic_ostream<char, _Tr>& _Ostr,
      signed char _Ch
      );
template<class _Tr>
   basic_ostream<char, _Tr>& operator<<(
      basic_ostream<char, _Tr>& _Ostr,
      const unsigned char *_Str
      );
template<class _Tr>
   basic_ostream<char, _Tr>& operator<<(
      basic_ostream<char, _Tr>& _Ostr,
      unsigned char _Ch
      );
```

Parameters

_Ch

A character.

_Elem

The element type.

_Ostr

A basic_ostream object.

_Str

A character string.

_Tr

Character traits.

Return Value

The stream.

Remarks

The **basic_ostream** class also defines several insertion operators. For more information, see basic_ostream::operator<<.

The template function

```
template<class _Elem, class _Tr>
   basic_ostream<Elem, _Tr>& operator<<(
      basic_ostream<Elem, _Tr>& _ostr,
      const Elem *_Str);
```

determines the length *N* = **traits_type::**length(*_Str*) of the sequence beginning at *_Str*, and inserts the sequence. If *N* < *_Ostr*.width, then the function also inserts a repetition of *_Ostr*.**width** - *N* fill characters. The repetition precedes the sequence if (*_Ostr*.flags **& adjustfield != left**. Otherwise, the repetition follows the sequence. The function returns *_Ostr*.

Security Note Use a null-terminated string. The null-terminated string must not exceed the size of the destination buffer. For more information, see Avoiding Buffer Overruns.

The template function

```
template<class _Elem, class _Tr>
   basic_ostream<Elem, _Tr>& operator<<(
      basic_ostream<Elem, _Tr>& _Ostr,
      Elem _Ch);
```

inserts the element *_Ch*. If 1 < *_Ostr*.**width**, then the function also inserts a repetition of *_Ostr*.**width** - 1 fill characters. The repetition precedes the sequence if (*_Ostr*.**flags & adjustfield != left**. Otherwise, the repetition follows the sequence. It returns *_Ostr*.

The template function

```
template<class _Elem, class _Tr>
   basic_ostream<Elem, _Tr>& operator<<(
      basic_ostream<Elem, _Tr>& _Ostr,
      const char *_Str);
```

behaves the same as

```
template<class _Elem, class _Tr>
   basic_ostream<Elem, _Tr>& operator<<(
      basic_ostream<Elem, _Tr>& _Ostr,
      const Elem *_Str);
```

except that each element *_Ch* of the sequence beginning at *_Str* is converted to an object of type **Elem** by calling *_Ostr*.put(*_Ostr*. widen(*_Ch*)).

The template function

```
template<class _Elem, class _Tr>
   basic_ostream<Elem, _Tr>& operator<<(
      basic_ostream<Elem, _Tr>& _Ostr,
      char _Ch);
```

behaves the same as

```
template<class _Elem, class _Tr>
   basic_ostream<Elem, _Tr>& operator<<(
      basic_ostream<Elem, _Tr>& _Ostr,
      Elem _Ch);
```

except that _*Ch* is converted to an object of type **Elem** by calling _*Ostr*.**put**(_*Ostr*. **widen**(_*Ch*)).

The template function

```
template<class _Tr>
   basic_ostream<char, _Tr>& operator<<(
      basic_ostream<char, _Tr>& _Ostr,
      const char *_Str);
```

behaves the same as

```
template<class _Elem, class _Tr>
   basic_ostream<Elem, _Tr>& operator<<(
      basic_ostream<Elem, _Tr>& _Ostr,
      const Elem *_Str);
```

(It does not have to widen the elements before inserting them.)

The template function

```
template<class _Tr>
   basic_ostream<char, Tr>& operator<<(
      basic_ostream<char, _Tr>& _Ostr,
      char _Ch);
```

behaves the same as

```
template<class _Elem, class _Tr>
   basic_ostream<Elem, _Tr>& operator<<(
      basic_ostream<Elem, _Tr>& _Ostr,
      Elem _Ch);
```

(It does not have to widen _*Ch* before inserting it.)

The template function

```
template<class _Tr>
   basic_ostream<char, _Tr>& operator<<(
      basic_ostream<char, _Tr>& _Ostr,
      const signed char *_Str);
```

returns _*Ostr* << (**const char ***)_*Str*.

The template function

```
template<class _Tr>
   basic_ostream<char, _Tr>& operator<<(
      basic_ostream<char, _Tr>& _Ostr,
      signed char _Ch);
```

returns _*Ostr* << (**char**)_*Ch*.

The template function:

```
template<class _Tr>
```

```
    basic_ostream<char, _Tr>& operator<<(
       basic_ostream<char, _Tr>& _Ostr,
       const unsigned char *_Str);
```

returns *_Ostr* << (**const char ***)*_Str*.

The template function:

```
  template<class _Tr>
     basic_ostream<char, _Tr>& operator<<(
        basic_ostream<char, _Tr>& _Ostr,
        unsigned char _Ch);
```

returns *_Ostr* << (**char**)*_Ch*.

Example

See flush for an example using **operator<<**.

Requirements

Header: <ostream>

See Also

Reference

basic_ostream::operator<<

iostream Programming

iostreams Conventions

Other Resources

<ostream> Members

<ostream> Classes

For information about the classes in <ostream>, see <ostream> Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_ostream Class

This template class describes an object that controls insertion of elements and encoded objects into a stream buffer with elements of type **Elem**, also known as char_type, whose character traits are determined by the class **Tr**, also known as traits_type.

For a list of all members of this type, see basic_ostream Members.

```
template <class _Elem, class _Tr = char_traits<Elem> >
   class basic_ostream
       : virtual public basic_ios<_Elem, _Tr>
```

Parameters

_Elem

A **char_type**.

_Tr

The character **traits_type**.

Remarks

Most of the member functions that overload operator< < are formatted output functions. They follow the pattern:

```
    iostate state = goodbit;
    const sentry ok( *this );
    if ( ok )
       {try
          {<convert and insert elements
          accumulate flags in state> }
       catch ( ... )
          {try
            {setstate( badbit ); }
          catch ( ... )
            {}
          if ( ( exceptions( ) & badbit ) != 0 )
            throw; }}
    width( 0 );    // Except for operator<<(Elem)
    setstate( state );
    return ( *this );
```

Two other member functions are unformatted output functions. They follow the pattern:

```
    iostate state = goodbit;
    const sentry ok( *this );
    if ( !ok )
       state |= badbit;
    else
       {try
          {<obtain and insert elements
          accumulate flags in state> }
       catch ( ... )
          {try
             {setstate( badbit ); }
          catch ( ... )
             {}
          if ( ( exceptions( ) & badbit ) != 0 )
             throw; }}
    setstate( state );
    return ( *this );
```

Both groups of functions call setstate**(badbit)** if they encounter a failure while inserting elements.

An object of class basic_istream<**Elem**, **Tr**> stores only a virtual public base object of class basic_ios**<Elem**, **Tr>**.

Example

See the example for basic_ofstream Class to learn more about output streams.

Requirements

Header: <ostream>

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

basic_ostream Members
<ostream> Members

basic_ostream Members

The following tables list the members of the basic_ostream Class. For a list of members inherited by the **basic_ostream** class, see basic_ios Members.

Reference

Constructors

basic_ostream	Constructs a **basic_ostream** object.

Member Functions

flush	Flushes the buffer.
put	Puts a character in a stream.
seekp	Resets position in output stream.
sentry	The nested class describes an object whose declaration structures the formatted output functions and the unformatted output functions.
tellp	Reports position in output stream.
write	Puts characters in a stream.

Operators

operator<<	Writes to the stream.

See Also

Reference

basic_ostream Class
basic_ios Class
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

basic_ios Members

basic_ostream Member Functions

For more information about the member functions in the basic_ostream class, see basic_ostream Class.

See Also

Reference

iostream Programming

iostreams Conventions

basic_ostream::basic_ostream

Constructs a basic_ostream object.

```
explicit basic_ostream(
   basic_streambuf<Elem, Tr> *_Strbuf
);
```

Parameters

_Strbuf

An object of type **basic_steambuf**.

Remarks

The constructor initializes the base class by calling init(*_Strbuf*).

Example

See the example for basic_ofstream::basic_ofstream to learn more about output streams.

Requirements

Header: <ostream>

See Also

Reference

basic_ostream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ostream Members

basic_ostream::flush

Flushes the buffer.

```
basic_ostream& flush( );
```

Return Value

A reference to the basic_ostream object.

Remarks

If rdbuf is not a null pointer, the function calls **rdbuf->**pubsync. If that returns -1, the function calls setstate(**badbit**). It returns ***this**.

Example

```
// basic_ostream_flush.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   cout << "test";
   cout.flush();
}
```

Output

```
test
```

Requirements

Header: <ostream>

See Also

Reference

basic_ostream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ostream Members

basic_ostream::put

Puts a character in a stream.

```
basic_ostream& put(
   char_type _Ch
);
```

Parameters

_Ch

A character.

Return Value

A reference to the basic_ostream object.

Remarks

The unformatted output function inserts the element *_Ch*. It returns ***this**.

Example

```
// basic_ostream_put.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   cout.put( 'v' );
   cout << endl;
   wcout.put( L'l' );
}
```

Output

```
v
l
```

Requirements

Header: <ostream>

See Also

Reference

basic_ostream Class

iostream Programming

iostreams Conventions

Other Resources

basic_ostream Members

basic_ostream::seekp

Reset position in output stream.

```
basic_ostream& seekp(
   pos_type _Pos
);
basic_ostream& seekp(
   off_type _Off,
   ios_base::seekdir _Way
);
```

Parameters

_Pos

The position in the stream.

_Off

The offset relative to *_Way*.

_Way

One of the ios_base::seekdir enumerations.

Return Value

A reference to the basic_ostream object.

Remarks

If fail is **false**, the first method calls **newpos** = rdbuf-> pubseekpos(*_Pos*), for some **pos_type** temporary object **newpos**. If **fail** is false, the second function calls **newpos = rdbuf->** pubseekoff(*_Off*, *_Way*). In either case, if (**off_type**)**newpos ==** (**off_type**)(-1) (the positioning operation fails), then the function calls **istr.**setstate(**failbit**). Both functions return ***this**.

Example

```
// basic_ostream_seekp.cpp
// compile with: /EHsc
#include <fstream>
#include <iostream>

int main()
{
    using namespace std;
    ofstream x("basic_ostream_seekp.txt");
    streamoff i = x.tellp();
    cout << i << endl;
    x << "testing";
    i = x.tellp();
    cout << i << endl;
    x.seekp(2);   // Put char in third char position in file
    x << " ";

    x.seekp(2, ios::end);   // Put char two after end of file
    x << "z";
    x.close();
}
```

Output

```
0
7
```

Requirements

Header: <ostream>

See Also

Reference

basic_ostream Class

iostream Programming

iostreams Conventions

Other Resources

basic_ostream Members

basic_ostream::sentry

The nested class describes an object whose declaration structures the formatted output functions and the unformatted output functions.

```
class sentry {
public:
   explicit sentry( basic_ostream<Elem, Tr>& _Ostr );
   operator bool( ) const;
   ~sentry( );
private:
   sentry( const sentry&);  // Not defined
   sentry& operator=(const sentry&);  // Not defined
   bool status;
};
```

Remarks

The nested class describes an object whose declaration structures the formatted output functions and the unformatted output functions. If **ostr.**good is **true** and **ostr.**tie is not a null pointer, the constructor calls **ostr.tie->**flush. The constructor then stores the value returned by **ostr.good** in **status**. A later call to **operator bool** delivers this stored value.

If **uncaught_exception** returns **false** and flags **&** unitbuf is nonzero, the destructor calls flush.

Requirements

Header: <ostream>

See Also

Reference

basic_ostream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ostream Members

basic_ostream::tellp

Report position in output stream.

```
pos_type tellp( );
```

Return Value

Position in output stream.

Remarks

If fail is **false**, the member function returns rdbuf**->** pubseekoff(0, **cur**, **in**). Otherwise, it returns **pos_type**(-1).

Example

See seekp for an example using **tellp**.

Requirements

Header: <ostream>

See Also

Reference

basic_ostream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ostream Members

basic_ostream::write

Put characters in a stream.

```
basic_ostream& write(
   const char_type *_Str,
   streamsize _Count
);
```

Parameters

_Count

Count of characters to put into the stream.

_Str

Characters to put into the stream.

Return Value

A reference to the basic_ostream object.

Remarks

The unformatted output function inserts the sequence of *_Count* elements beginning at *_Str*.

Example

See streamsize for an example using **write**.

Requirements

Header: <ostream>

See Also

Reference

basic_ostream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ostream Members

basic_ostream Operators

For more information about the operators in the basic_ostream class, see basic_ostream Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_ostream::operator<<

Writes to the stream.

```
basic_ostream& operator<<(
   basic_ostream& (*_Pfn)(basic_ostream&)
);
basic_ostream& operator<<(
   ios_base& (*_Pfn)(ios_base&)
);
basic_ostream& operator<<(
   basic_ios<Elem, Tr>& (*_Pfn)(basic_ios<Elem, Tr>&)
);
basic_ostream& operator<<(
   basic_streambuf<Elem, Tr> *_Strbuf
);
basic_ostream& operator<<(
   bool _Val
);
basic_ostream& operator<<(
   short _Val
);
basic_ostream& operator<<(
   unsigned short _Val
);
basic_ostream& operator<<(
   int _Val
);
basic_ostream& operator<<(
   unsigned int __w64 _Val
);
basic_ostream& operator<<(
   long _Val
);
basic_ostream& operator<<(
   unsigned long _Val
);
basic_ostream& operator<<(
   float _Val
);
basic_ostream& operator<<(
   double _Val
);
basic_ostream& operator<<(
   long double _Val
);
basic_ostream& operator<<(
   const void *_Val
);
```

Parameters

_Pfn

A function pointer.

_Strbuf

A pointer to a **stream_buf** object.

_Val

An element to write to the stream.

Return Value

A reference to the basic_ostream object.

Remarks

The **<ostream>** header also defines several global insertion operators. For more information, see operator<< (<ostream>).

The first member function ensures that an expression of the form **ostr << endl** calls endl**(ostr)**, and then returns ***this**. The second and third functions ensure that other manipulators, such as hex, behave similarly. The remaining functions are all formatted output functions.

The function

```
basic_ostream& operator<<(basic_streambuf<Elem, Tr> *_Strbuf);
```

extracts elements from *_Strbuf*, if *_Strbuf* is not a null pointer, and inserts them. Extraction stops on end of file, or if an extraction throws an exception (which is rethrown). It also stops, without extracting the element in question, if an insertion fails. If the function inserts no elements, or if an extraction throws an exception, the function calls setstate(**failbit**). In any case, the function returns ***this**.

The function

```
basic_ostream& operator<<(bool _Val);
```

converts `_Val` to a Boolean field and inserts it by calling use_facet**<num_put<Elem, OutIt>** (getloc). put(**OutIt**(rdbuf), ***this**, **getloc**, **val**). Here, **OutIt** is defined as ostreambuf_iterator**<Elem, Tr>**. The function returns ***this**.

The functions

```
basic_ostream& operator<<(short _Val);
basic_ostream& operator<<(unsigned short _Val);
basic_ostream& operator<<(int _Val);
basic_ostream& operator<<(unsigned int __w64 _Val);
basic_ostream& operator<<(long _Val);
basic_ostream& operator<<(unsigned long _Val);
basic_ostream& operator<<(const void *_Val);
```

each convert *_Val* to a numeric field and insert it by calling **use_facet<num_put<Elem, OutIt>**(**getloc**). **put**(**OutIt**(**rdbuf**), ***this**, **getloc**, **val**). Here, **OutIt** is defined as **ostreambuf_iterator<Elem, Tr>**. The function returns ***this**.

The functions

```
basic_ostream& operator<<(float _Val);
basic_ostream& operator<<(double _Val);
basic_ostream& operator<<(long double _Val);
```

each convert `_Val` to a numeric field and insert it by calling **use_facet<num_put<Elem, OutIt>**(**getloc**)**. put**(**OutIt**(**rdbuf**), ***this**, **getloc**, **val**). Here, **OutIt** is defined as **ostreambuf_iterator<Elem, Tr>**. The function returns ***this**.

Example

```
// basic_ostream_op_write.cpp
// compile with: /EHsc
#include <iostream>
#include <string.h>

using namespace std;

ios_base& hex2( ios_base& ib )
{
   ib.unsetf( ios_base::dec );
   ib.setf( ios_base::hex );
   return ib;
}

basic_ostream<char, char_traits<char> >& somefunc(basic_ostream<char, char_traits<char> > &
i)
{
```

```
   if ( i == cout )
   {
      i << "i is cout" << endl;
   }
   return i;
}

class CTxtStreambuf : public basic_streambuf< char, char_traits< char > >
{
public:
   CTxtStreambuf(char *_pszText)
   {
      pszText = _pszText;
      setg(pszText, pszText, pszText+strlen(pszText));
   };
         char *pszText;
};

int main( )
{
   cout << somefunc;
   cout << 21 << endl;

   hex2(cout);
   cout << 21 << endl;

   CTxtStreambuf f("text in streambuf");
   cout << &f << endl;
}
```

Output

```
i is cout
21
15
text in streambuf
```

Requirements

Header: <ostream>

See Also

Reference

basic_ostream Class
operator<< (<ostream>)
iostream Programming
iostreams Conventions

Other Resources

basic_ostream Members

<queue>

Defines the template classes priority_queue and queue and several supporting templates.

For a list of all members of this header, see <queue> Members.

```
#include <queue>
```

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<queue> Members

Standard C++ Library Header Files

<queue> Members

Reference

Operators

operator!= (<queue>)	Tests if the queue object on the left side of the operator is not equal to the queue object on the right side.
operator< (<queue>)	Tests if the queue object on the left side of the operator is less than the queue object on the right side.
operator<= (<queue>)	Tests if the queue object on the left side of the operator is less than or equal to the queue object on the right side.
operator== (<queue>)	Tests if the queue object on the left side of the operator is equal to the queue object on the right side.
operator> (<queue>)	Tests if the queue object on the left side of the operator is greater than the queue object on the right side.
operator>= (<queue>)	Tests if the queue object on the left side of the operator is greater than or equal to the queue object on the right side.

Classes

queue Class	A template container adaptor class that provides a restriction of functionality limiting access to the front and back elements of some underlying container type.
priority_queue Class	A template container adaptor class that provides a restriction of functionality limiting access to the top element of some underlying container type, which is always the largest.

See Also

Reference

<queue>
Thread Safety in the Standard C++ Library
Standard Template Library

<queue> Operators

For more information about the operators in <queue>, see <queue> Members.

See Also

Reference

Standard Template Library

operator!= (<queue>)

Tests if the queue object on the left side of the operator is not equal to the queue object on the right side.

```
bool operator!=(
   const queue <Type, Container>& _Left,
   const queue <Type, Container>& _Right,
);
```

Parameters

_Left

An object of type **queue**.

_Right

An object of type **queue**.

Return Value

true if the queues are not equal; **false** if queues are equal.

Remarks

The comparison between queue objects is based on a pairwise comparison of their elements. Two queues are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// queue_op_ne.cpp
// compile with: /EHsc
#include <queue>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares queues with list base containers
   queue <int, list<int> > q1, q2, q3;

   // The following line would have caused an error because vector
   // does not support pop_front( ) and so cannot be adapted
   // by queue as a base container
   // queue <int, vector<int> > q1, q2, q3;

   q1.push( 1 );
   q2.push( 1 );
   q2.push( 2 );
   q3.push( 1 );

   if ( q1 != q2 )
      cout << "The queues q1 and q2 are not equal." << endl;
   else
      cout << "The queues q1 and q2 are equal." << endl;


   if ( q1 != q3 )
      cout << "The queues q1 and q3 are not equal." << endl;
   else
      cout << "The queues q1 and q3 are equal." << endl;
}
```

Output

```
The queues q1 and q2 are not equal.
The queues q1 and q3 are equal.
```

Requirements

Header: <queue>

See Also

Reference

Standard Template Library

Other Resources

<queue> Members

operator< (<queue>)

Tests if the queue object on the left side of the operator is less than the queue object on the right side.

```
bool operator<(
   const queue <Type, Container>& _Left,
   const queue <Type, Container>& _Right,
);
```

Parameters

_Left

An object of type **queue**.

_Right

An object of type **queue**.

Return Value

true if the queue on the left side of the operator is less than and not equal to the queue on the right side of the operator; otherwise **false**.

Remarks

The comparison between queue objects is based on a pairwise comparison of their elements. The less-than relationship between two queue objects is based on a comparison of the first pair of unequal elements.

Example

```
// queue_op_lt.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares queues with default deque base container
   queue <int> q1, q2, q3;

   q1.push( 1 );
   q1.push( 2 );
   q2.push( 5 );
   q2.push( 10 );
   q3.push( 1 );
   q3.push( 2 );

   if ( q1 < q2 )
      cout << "The queue q1 is less than the queue q2." << endl;
   else
      cout << "The queue q1 is not less than the queue q2." << endl;

   if ( q1 < q3 )
      cout << "The queue q1 is less than the queue q3." << endl;
   else
      cout << "The queue q1 is not less than the queue q3." << endl;
}
```

Output

```
The queue q1 is less than the queue q2.
The queue q1 is not less than the queue q3.
```

Requirements

Header: <queue>

See Also

Reference

Standard Template Library

Other Resources

<queue> Members

operator<= (<queue>)

Tests if the queue object on the left side of the operator is less than or equal to the queue object on the right side.

```
bool operator<=(
   const queue <Type, Container>& _Left,
   const queue <Type, Container>& _Right,
);
```

Parameters

_Left

An object of type **queue**.

_Right

An object of type **queue**.

Return Value

true if the queue on the left side of the operator is strictly less than the queue on the right side of the operator; otherwise **false**.

Remarks

The comparison between queue objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two queue objects is based on a comparison of the first pair of unequal elements.

Example

```
// queue_op_le.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   queue <int> q1, q2, q3;

   q1.push( 5 );
   q1.push( 10 );
   q2.push( 1 );
   q2.push( 2 );
   q3.push( 5 );
   q3.push( 10 );

   if ( q1 <= q2 )
      cout << "The queue q1 is less than or equal to "
           << "the queue q2." << endl;
   else
      cout << "The queue q1 is greater than "
           << "the queue q2." << endl;

   if ( q1 <= q3 )
      cout << "The queue q1 is less than or equal to "
           << "the queue q3." << endl;
   else
      cout << "The queue q1 is greater than "
           << "the queue q3." << endl;
}
```

Output

```
The queue q1 is greater than the queue q2.
The queue q1 is less than or equal to the queue q3.
```

Requirements

Header: <queue>

See Also

Reference

Standard Template Library

Other Resources

<queue> Members

operator== (<queue>)

Tests if the queue object on the left side of the operator is equal to queue object on the right side.

```
bool operator==(
   const queue <Type, Container>& _Left,
   const queue <Type, Container>& _Right,
);
```

Parameters

_Left

An object of type **queue**.

_Right

An object of type **queue**.

Return Value

true if the queues are not equal; **false** if queues are equal.

Remarks

The comparison between queue objects is based on a pairwise comparison of their elements. Two queues are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// queue_op_eq.cpp
// compile with: /EHsc
#include <queue>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares queues with list base containers
   queue <int, list<int> > q1, q2, q3;

   // The following line would have caused an error because vector
   // does not support pop_front( ) and so cannot be adapted
   // by queue as a base container
   // queue <int, vector<int> > q1, q2, q3;

   q1.push( 1 );
   q2.push( 2 );
   q3.push( 1 );

   if ( q1 != q2 )
      cout << "The queues q1 and q2 are not equal." << endl;
   else
      cout << "The queues q1 and q2 are equal." << endl;


   if ( q1 != q3 )
      cout << "The queues q1 and q3 are not equal." << endl;
   else
      cout << "The queues q1 and q3 are equal." << endl;
}
```

Output

```
The queues q1 and q2 are not equal.
```

```
The queues q1 and q3 are equal.
```

Requirements

Header: <queue>

See Also

Reference

Standard Template Library

Other Resources

<queue> Members

operator> (<queue>)

Tests if the queue object on the left side of the operator is greater than the queue object on the right side.

```
bool operator>(
   const queue <Type, Container>& _Left,
   const queue <Type, Container>& _Right,
);
```

Parameters

_Left

An object of type **queue**.

_Right

An object of type **queue**.

Return Value

true if the queue on the left side of the operator is strictly less than the queue on the right side of the operator; otherwise **false**.

Remarks

The comparison between queue objects is based on a pairwise comparison of their elements. The greater-than relationship between two queue objects is based on a comparison of the first pair of unequal elements.

Example

```
// queue_op_gt.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   queue <int> q1, q2, q3;

   q1.push( 1 );
   q1.push( 2 );
   q1.push( 3 );
   q2.push( 5 );
   q2.push( 10 );
   q3.push( 1 );
   q3.push( 2 );

   if ( q1 > q2 )
      cout << "The queue q1 is greater than "
           << "the queue q2." << endl;
   else
      cout << "The queue q1 is not greater than "
           << "the queue q2." << endl;

   if ( q1> q3 )
      cout << "The queue q1 is greater than "
           << "the queue q3." << endl;
   else
      cout << "The queue q1 is not greater than "
           << "the queue q3." << endl;
}
```

Output

```
The queue q1 is not greater than the queue q2.
The queue q1 is greater than the queue q3.
```

Requirements

Header: <queue>

See Also

Reference

Standard Template Library

Other Resources

<queue> Members

operator>= (<queue>)

Tests if the queue object on the left side of the operator is greater than or equal to the queue object on the right side.

```
bool operator>=(
   const queue <Type, Container>& _Left,
   const queue <Type, Container>& _Right,
);
```

Parameters

_Left

An object of type **queue**.

_Right

An object of type **queue**.

Return Value

true if the queue on the left side of the operator is strictly less than the queue on the right side of the operator; otherwise **false**.

Remarks

The comparison between queue objects is based on a pairwise comparison of their elements. Two queues are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// queue_op_ge.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   queue <int> q1, q2, q3;

   q1.push( 1 );
   q1.push( 2 );
   q2.push( 5 );
   q2.push( 10 );
   q3.push( 1 );
   q3.push( 2 );


   if ( q1 >= q2 )
      cout << "The queue q1 is greater than or equal to "
           << "the queue q2." << endl;
   else
      cout << "The queue q1 is less than "
           << "the queue q2." << endl;

   if ( q1>= q3 )
      cout << "The queue q1 is greater than or equal to "
           << "the queue q3." << endl;
   else
      cout << "The queue q1 is less than "
           << "the queue q3." << endl;
}
```

Output

```
The queue q1 is less than the queue q2.
The queue q1 is greater than or equal to the queue q3.
```

Requirements

Header: <queue>

See Also

Reference

Standard Template Library

Other Resources

<queue> Members

<queue> Classes

For more information about the classes in <queue>, see <queue> Members.

See Also

Reference

Standard Template Library

queue Class

A template container adaptor class that provides a restriction of functionality for some underlying container type, limiting access to the front and back elements. Elements can be added at the back or removed from the front, and elements can be inspected at either end of the queue.

For a list of all members of this type, see queue Members.

```
template <
   class Type,
   class Container = deque<Type>
>
class queue
```

Parameters

Type

The element data type to be stored in the queue

Container

The type of the underlying container used to implement the queue.

Remarks

The elements of class **Type** stipulated in the first template parameter of a queue object are synonymous with value_type and must match the type of element in the underlying container class **Container** stipulated by the second template parameter. The **Type** must be assignable, so that it is possible to copy objects of that type and to assign values to variables of that type.

Suitable underlying container classes for queue include deque and list, or any other sequence container that supports the operations of **front**, **back**, **push_back**, and **pop_front**. The underlying container class is encapsulated within the container adaptor, which exposes only the limited set of the sequence container member functions as a public interface.

The queue objects are equality comparable if and only if the elements of class **Type** are equality comparable, and are less-than comparable if and only if the elements of class **Type** are less-than comparable.

There are three types of container adaptors defined by the STL: stack, queue, and priority_queue. Each restricts the functionality of some underlying container class to provide a precisely controlled interface to a standard data structure.

- The stack class supports a last-in, first-out (LIFO) data structure. A good analogue to keep in mind would be a stack of plates. Elements (plates) may be inserted, inspected, or removed only from the top of the stack, which is the last element at the end of the base container. The restriction to accessing only the top element is the reason for using the stack class.
- The queue class supports a first-in, first-out (FIFO) data structure. A good analogue to keep in mind would be people lining up for a bank teller. Elements (people) may be added to the back of the line and are removed from the front of the line. Both the front and the back of a line may be inspected. The restriction to accessing only the front and back elements in this way is the reason for using the queue class.
- The priority_queue class orders its elements so that the largest element is always at the top position. It supports insertion of an element and the inspection and removal of the top element. A good analogue to keep in mind would be people lining up where they are arranged by age, height, or some other criterion.

Requirements

Header: <queue>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

queue Members

queue Members

Reference

Constructors

queue	Constructs a **queue** that is empty or that is a copy of a base container object.

Typedefs

container_type	A type that provides the base container to be adapted by the **queue**.
size_type	An unsigned integer type that can represent the number of elements in a **queue**.
value_type	A type that represents the type of object stored as an element in a **queue**.

Member Functions

back	Returns a reference to the last and most recently added element at the back of the **queue**.
empty	Tests if the **queue** is empty.
front	Returns a reference to the first element at the front of the **queue**.
pop	Removes an element from the front of the **queue**.
push	Adds an element to the back of the **queue**.
size	Returns the number of elements in the **queue**.

See Also

Reference

queue Class

Thread Safety in the Standard C++ Library

Standard Template Library

queue Typedefs

For information about the typedefs in the queue class, see queue Members.

See Also

Reference

Standard Template Library

queue::container_type

A type that provides the base container to be adapted.

```
typedef Container container_type;
```

Remarks

The type is a synonym for the template parameter **Container**. Two STL sequence container classes — the list class and the default deque class — meet the requirements to be used as the base container for a queue object. User-defined types satisfying the requirements may also be used.

For more information on **Container**, see the Remarks section of the queue Class topic.

Example

See the example for queue for an example of how to declare and use **container_type**.

Requirements

Header: <queue>

See Also

Reference

queue Class

Standard Template Library

Other Resources

queue Members

queue::size_type

An unsigned integer type that can represent the number of elements in a queue.

```
typedef typename Container::size_type size_type;
```

Remarks

The type is a synonym for the **size_type** of the base container adapted by the queue.

Example

See the example for queue::front for an example of how to declare and use **size_type**.

Requirements

Header: <queue>

See Also

Reference

queue Class

Standard Template Library

Other Resources

queue Members

queue::value_type

A type that represents the type of object stored as an element in a queue.

```
typedef typename Container::value_type value_type;
```

Remarks

The type is a synonym for the **value_type** of the base container adapted by the queue.

Example

```
// queue_value_type.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
using namespace std;

   // Declares queues with default deque base container
   queue<int>::value_type AnInt;

   AnInt = 69;
   cout << "The value_type is AnInt = " << AnInt << endl;

   queue<int> q1;
   q1.push(AnInt);
   cout << "The element at the front of the queue is "
        << q1.front( ) << "." << endl;
}
```

Output

```
The value_type is AnInt = 69
The element at the front of the queue is 69.
```

Requirements

Header: <queue>

See Also

Reference

queue Class

Standard Template Library

Other Resources

queue Members

queue Member Functions

For information about the functions in the queue class, see queue Members.

See Also

Reference

Standard Template Library

queue::back

Returns a reference to the last and most recently added element at the back of the queue.

```
value_type& back( );
const value_type& back( ) const;
```

Return Value

The last element of the queue. If the queue is empty, the return value is undefined.

Remarks

If the return value of **back** is assigned to a **const_reference**, the queue object cannot be modified. If the return value of **back** is assigned to a **reference**, the queue object can be modified.

When compiling with _SECURE_SCL 1, a runtime error will occur if you attempt to access an element in an empty queue. See Checked Iterators for more information.

Example

```
// queue_back.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   queue <int> q1;

   q1.push( 10 );
   q1.push( 11 );

   int& i = q1.back( );
   const int& ii = q1.front( );

   cout << "The integer at the back of queue q1 is " << i
        << "." << endl;
   cout << "The integer at the front of queue q1 is " << ii
        << "." << endl;
}
```

Output

```
The integer at the back of queue q1 is 11.
The integer at the front of queue q1 is 10.
```

Requirements

Header: <queue>

See Also

Reference

queue Class

queue Functions

Standard Template Library

Other Resources

queue Members

queue::empty

Tests if a queue is empty.

```
bool empty( ) const;
```

Return Value

true if the queue is empty; **false** if the queue is nonempty.

Example

```
// queue_empty.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
using namespace std;

   // Declares queues with default deque base container
   queue <int> q1, q2;

   q1.push( 1 );

   if ( q1.empty( ) )
      cout << "The queue q1 is empty." << endl;
   else
      cout << "The queue q1 is not empty." << endl;

   if ( q2.empty( ) )
      cout << "The queue q2 is empty." << endl;
   else
      cout << "The queue q2 is not empty." << endl;
}
```

Output

```
The queue q1 is not empty.
The queue q2 is empty.
```

Requirements

Header: <queue>

See Also

Reference

queue Class

queue Functions

Standard Template Library

Other Resources

queue Members

queue::front

Returns a reference to the first element at the front of the queue.

```
value_type& front( );
const value_type& front( ) const;
```

Return Value

The first element of the queue. If the queue is empty, the return value is undefined.

Remarks

If the return value of **front** is assigned to a **const_reference**, the queue object cannot be modified. If the return value of **front** is assigned to a **reference**, the queue object can be modified.

The member function returns a **reference** to the first element of the controlled sequence, which must be nonempty.

When compiling with _SECURE_SCL 1, a runtime error will occur if you attempt to access an element in an empty queue. See Checked Iterators for more information.

Example

```
// queue_front.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main() {
   using namespace std;
   queue <int> q1;

   q1.push( 10 );
   q1.push( 20 );
   q1.push( 30 );

   queue <int>::size_type i;
   i = q1.size( );
   cout << "The queue length is " << i << "." << endl;

   int& ii = q1.back( );
   int& iii = q1.front( );

   cout << "The integer at the back of queue q1 is " << ii
        << "." << endl;
   cout << "The integer at the front of queue q1 is " << iii
        << "." << endl;
}
```

Output

```
The queue length is 3.
The integer at the back of queue q1 is 30.
The integer at the front of queue q1 is 10.
```

Requirements

Header: <queue>

See Also

Reference

queue Class

queue Functions

Standard Template Library

Other Resources

queue Members

queue::pop

Removes an element from the front of the queue.

```
void pop( );
```

Remarks

The queue must be nonempty to apply the member function. The top of the queue is the position occupied by the most recently added element and is the last element at the end of the container.

Example

```
// queue_pop.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   queue <int> q1, s2;

   q1.push( 10 );
   q1.push( 20 );
   q1.push( 30 );

   queue <int>::size_type i;
   i = q1.size( );
   cout << "The queue length is " << i << "." << endl;

   i = q1.front( );
   cout << "The element at the front of the queue is "
        << i << "." << endl;

   q1.pop( );

   i = q1.size( );
   cout << "After a pop the queue length is "
        << i << "." << endl;

   i = q1. front ( );
   cout << "After a pop, the element at the front of the queue is "
        << i << "." << endl;
}
```

Output

```
The queue length is 3.
The element at the front of the queue is 10.
After a pop the queue length is 2.
After a pop, the element at the front of the queue is 20.
```

Requirements

Header: <queue>

See Also

Reference

queue Class

queue Functions

Standard Template Library

Other Resources

queue Members

queue::push

Adds an element to the back of the queue.

```
void push(
   const Type& _Val
);
```

Parameters

_Val

The element added to the top of the queue.

Remarks

The top of the queue is the position occupied by the most recently added element and is the last element at the end of the container.

Example

```
// queue_push.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   queue <int> q1;

   q1.push( 10 );
   q1.push( 20 );
   q1.push( 30 );

   queue <int>::size_type i;
   i = q1.size( );
   cout << "The queue length is " << i << "." << endl;

   i = q1.front( );
   cout << "The element at the front of the queue is "
        << i << "." << endl;
}
```

Output

```
The queue length is 3.
The element at the front of the queue is 10.
```

Requirements

Header: <queue>

See Also

Reference

queue Class

queue Functions

Standard Template Library

Other Resources

queue Members

queue::queue

Constructs a queue that is empty or that is a copy of a base container object.

```
queue( );
explicit queue(
   const container_type& _Right
);
```

Parameters

_Right

The **const** container of which the constructed queue is to be a copy.

Remarks

The default base container for queue is deque. You can also specify list as a base container, but you cannot specify vector, because it lacks the required **pop_front** member function.

Example

```
// queue_queue.cpp
// compile with: /EHsc
#include <queue>
#include <vector>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares queue with default deque base container
   queue <char> q1;

   // Explicitly declares a queue with deque base container
   queue <char, deque<char> > q2;

   // These lines don't cause an error, even though they
   // declares a queue with a vector base container
   queue <int, vector<int> > q3;
   q3.push( 10 );
   // but the following would cause an error because vector has
   // no pop_front member function
   // q3.pop( );

   // Declares a queue with list base container
   queue <int, list<int> > q4;

   // The second member function copies elements from a container
   list<int> li1;
   li1.push_back( 1 );
   li1.push_back( 2 );
   queue <int, list<int> > q5( li1 );
   cout << "The element at the front of queue q5 is "
        << q5.front( ) << "." << endl;
   cout << "The element at the back of queue q5 is "
        << q5.back( ) << "." << endl;
}
```

Output

```
The element at the front of queue q5 is 1.
The element at the back of queue q5 is 2.
```

Requirements

Header: <queue>

See Also

Reference

queue Class

Standard Template Library

Other Resources

queue Members

queue::size

Returns the number of elements in the queue.

```
size_type size( ) const;
```

Return Value

The current length of the queue.

Example

```
// queue_size.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   queue <int> q1, q2;
   queue <int>::size_type i;

   q1.push( 1 );
   i = q1.size( );
   cout << "The queue length is " << i << "." << endl;

   q1.push( 2 );
   i = q1.size( );
   cout << "The queue length is now " << i << "." << endl;
}
```

Output

```
The queue length is 1.
The queue length is now 2.
```

Requirements

Header: <queue>

See Also

Reference

queue Class

queue Functions

Standard Template Library

Other Resources

queue Members

priority_queue Class

A template container adaptor class that provides a restriction of functionality limiting access to the top element of some underlying container type, which is always the largest or of the highest priority. New elements can be added to the priority_queue and the top element of the priority_queue can be inspected or removed.

For a list of all members of this type, see priority_queue Members.

```
template <
   class Type,
   class Container=vector<Type>,
      class Compare=less<typename Container::value_type>
>
class priority_queue
```

Parameters

Type

The element data type to be stored in the priority_queue.

Container

The type of the underlying container used to implement the priority_queue.

Compare

The type that provides a function object that can compare two element values as sort keys to determine their relative order in the priority_queue. This argument is optional and the binary predicate **less**<**typename** *Container*::**value_type**> is the default value.

Remarks

The elements of class **Type** stipulated in the first template parameter of a queue object are synonymous with value_type and must match the type of element in the underlying container class **Container** stipulated by the second template parameter. The **Type** must be assignable, so that it is possible to copy objects of that type and to assign values to variables of that type.

The priority_queue orders the sequence it controls by calling a stored function object of class **Traits**. In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense.

Suitable underlying container classes for priority_queue include deque Class and the default vector Class or any other sequence container that supports the operations of **front**, **push_back**, and **pop_front** and a random-access iterator. The underlying container class is encapsulated within the container adaptor, which exposes only the limited set of the sequence container member functions as a public interface.

Adding elements to, removing elements from, and accessing elements in a **priority_queue** all have constant complexity.

There are three types of container adaptors defined by the STL: stack, queue, and priority_queue. Each restricts the functionality of some underlying container class to provide a precisely controlled interface to a standard data structure.

- The stack class supports a last-in, first-out (LIFO) data structure. A good analogue to keep in mind would be a stack of plates. Elements (plates) may be inserted, inspected, or removed only from the top of the stack, which is the last element at the end of the base container. The restriction to accessing only the top element is the reason for using the stack class.
- The queue class supports a first-in, first-out (FIFO) data structure. A good analogue to keep in mind would be people lining up for a bank teller. Elements (people) may be added to the back of the line and are removed from the front of the line. Both the front and the back of a line may be inspected. The restriction to accessing only the front and back elements in this way is the reason fur using the queue class.
- The priority_queue class orders its elements so that the largest element is always at the top position. It supports insertion of an element and the inspection and removal of the top element. A good analogue to keep in mind would be people lining up where they are arranged by age, height, or some other criterion.

Requirements

Header: <queue>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

priority_queue Members

priority_queue Members

Reference

Constructors

priority_queue	Constructs a **priority_queue** that is empty or that is a copy of a range of a base container object or of other **priority_queue**.

Typedefs

container_type	A type that provides the base container to be adapted by a **priority_queue**.
size_type	An unsigned integer type that can represent the number of elements in a **priority_queue**.
value_type	A type that represents the type of object stored as an element in a **priority_queue**.

Member Functions

empty	Tests if the **priority_queue** is empty.
pop	Removes the largest element of the **priority_queue** from the top position.
push	Adds an element to the priority queue based on the priority of the element from operator<.
size	Returns the number of elements in the **priority_queue**.
top	Returns a const reference to the largest element at the top of the **priority_queue**.

See Also

Reference

priority_queue Class
Thread Safety in the Standard C++ Library
Standard Template Library

priority_queue Typedefs

For information about the typedefs in the priority_queue class, see priority_queue Members.

See Also

Reference

Standard Template Library

priority_queue::container_type

A type that provides the base container to be adapted.

```
typedef Container container_type;
```

Remarks

The type is a synonym for the template parameter **Container**. The STL sequence container class deque and the default class vector meet the requirements to be used as the base container for a priority_queue object. User-defined types satisfying the requirements may also be used.

For more information on **Container**, see the Remarks section of the priority_queue Class topic.

Example

See the example for priority_queue for an example of how to declare and use **container_type**.

Requirements

Header: <queue>

See Also

Reference

priority_queue Class

Standard Template Library

Other Resources

priority_queue Members

priority_queue::size_type

An unsigned integer type that can represent the number of elements in a priority_queue.

```
typedef typename Container::size_type size_type;
```

Remarks

The type is a synonym for the **size_type** of the base container adapted by the priority_queue.

Example

See the example for size for an example of how to declare and use **size_type**.

Requirements

Header: <queue>

See Also

Reference

priority_queue Class

Standard Template Library

Other Resources

priority_queue Members

priority_queue::value_type

A type that represents the type of object stored as an element in a priority_queue.

```
typedef typename Container::value_type value_type;
```

Remarks

The type is a synonym for the **value_type** of the base container adapted by the priority_queue.

Example

```
// pqueue_value_type.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares priority_queues with default deque base container
   priority_queue<int>::value_type AnInt;

   AnInt = 69;
   cout << "The value_type is AnInt = " << AnInt << endl;

   priority_queue<int> q1;
   q1.push( AnInt );
   cout << "The element at the top of the priority_queue is "
        << q1.top( ) << "." << endl;
}
```

Output

```
The value_type is AnInt = 69
The element at the top of the priority_queue is 69.
```

Requirements

Header: <queue>

See Also

Reference

priority_queue Class

Standard Template Library

Other Resources

priority_queue Members

priority_queue Member Functions

For information about the functions in the priority_queue class, see priority_queue Members.

See Also

Reference

Standard Template Library

priority_queue::empty

Tests if a priority_queue is empty.

```
bool empty( ) const;
```

Return Value

true if the priority_queue is empty; **false** if the priority_queue is nonempty.

Example

```
// pqueue_empty.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares priority_queues with default deque base container
   priority_queue <int> q1, s2;

   q1.push( 1 );

   if ( q1.empty( ) )
      cout << "The priority_queue q1 is empty." << endl;
   else
      cout << "The priority_queue q1 is not empty." << endl;

   if ( s2.empty( ) )
      cout << "The priority_queue s2 is empty." << endl;
   else
      cout << "The priority_queue s2 is not empty." << endl;
}
```

Output

```
The priority_queue q1 is not empty.
The priority_queue s2 is empty.
```

Requirements

Header: <queue>

See Also

Reference

priority_queue Class
priority_queue Functions
Standard Template Library

Other Resources

priority_queue Members

priority_queue::pop

Removes the largest element of the priority_queue from the top position.

```
void pop( );
```

Remarks

The priority_queue must be nonempty to apply the member function. The top of the priority_queue is always occupied by the largest element in the container.

Example

```
// pqueue_pop.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   priority_queue <int> q1, s2;

   q1.push( 10 );
   q1.push( 20 );
   q1.push( 30 );

   priority_queue <int>::size_type i, iii;
   i = q1.size( );
   cout << "The priority_queue length is " << i << "." << endl;

   const int& ii = q1.top( );
   cout << "The element at the top of the priority_queue is "
        << i << "." << endl;

   q1.pop( );

   iii = q1.size( );
   cout << "After a pop, the priority_queue length is "
        << iii << "." << endl;

   const int& iv = q1.top( );
   cout << "After a pop, the element at the top of the "
        << "priority_queue is " << iv << "." << endl;
}
```

Output

```
The priority_queue length is 3.
The element at the top of the priority_queue is 3.
After a pop, the priority_queue length is 2.
After a pop, the element at the top of the priority_queue is 20.
```

Requirements

Header: <queue>

See Also

Reference

priority_queue Class
priority_queue Functions
Standard Template Library

Other Resources

priority_queue Members

priority_queue::priority_queue

Constructs a priority_queue that is empty or that is a copy of a range of a base container object or of another priority_queue.

```
priority_queue( );
explicit priority_queue(
   const Traits& __Comp
);
priority_queue(
   const Traits& __Comp,
   const container_type& _Cont
);
priority_queue(
   const priority_queue& _Right
);
template<class InputIterator>
   priority_queue(
      InputIterator _First,
      InputIterator _Last
);
template<class InputIterator>
   priority_queue(
      InputIterator _First,
      InputIterator _Last,
      const Traits& __Comp
);
template<class InputIterator>
   priority_queue(
      InputIterator _First,
      InputIterator _Last,
      const Traits& __Comp,
      const container_type& _Cont
);
```

Parameters

__Comp

The comparison function of type **const Traits** used to order the elements in the priority_queue, which defaults to compare function of the base container.

_Cont

The base container of which the constructed priority_queue is to be a copy.

_Right

The priority_queue of which the constructed set is to be a copy.

_First

The position of the first element in the range of elements to be copied.

_Last

The position of the first element beyond the range of elements to be copied.

Remarks

Each of the first three constructors specifies an empty initial priority_queue, the second also specifying the type of comparison function (*_Comp*) to be used in establishing the order of the elements and the third explicitly specifying the **container_type** (*_Cont*) to be used. The keyword **explicit** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the priority_queue *_Right*.

The last three constructors copy the range [*_First, _Last*) of some container and use the values to initialize a priority_queue with increasing explicitness in specifying the type of comparison function of class **Traits** and **container_type**.

Example

```
// pqueue_ctor.cpp
// compile with: /EHsc
#include <queue>
#include <vector>
#include <deque>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function declares priority_queue
   // with a default vector base container
   priority_queue <int> q1;
   cout << "q1 = ( ";
   while ( !q1.empty( ) )
   {
      cout << q1.top( ) << " ";
      q1.pop( );
   }
   cout << ")" << endl;

   // Explicitly declares a priority_queue with nondefault
   // deque base container
   priority_queue <int, deque <int> > q2;
   q2.push( 5 );
   q2.push( 15 );
   q2.push( 10 );
   cout << "q2 = ( ";
   while ( !q2.empty( ) )
   {
      cout << q2.top( ) << " ";
      q2.pop( );
   }
   cout << ")" << endl;

   // This method of printing out the elements of a priority_queue
   // removes the elements from the priority queue, leaving it empty
   cout << "After printing, q2 has " << q2.size( ) << " elements." << endl;

   // The third member function declares a priority_queue
   // with a vector base container and specifies that the comparison
   // function greater is to be used for ordering elements
   priority_queue <int, vector<int>, greater<int> > q3;
   q3.push( 2 );
   q3.push( 1 );
   q3.push( 3 );
   cout << "q3 = ( ";
   while ( !q3.empty( ) )
   {
      cout << q3.top( ) << " ";
      q3.pop( );
   }
   cout << ")" << endl;

   // The fourth member function declares a priority_queue and
   // initializes it with elements copied from another container:
   // first, inserting elements into q1, then copying q1 elements into q4
   q1.push( 100 );
   q1.push( 200 );
   priority_queue <int> q4( q1 );
   cout << "q4 = ( ";
   while ( !q4.empty( ) )
   {
      cout << q4.top( ) << " ";
```

```
      q4.pop( );
   }
   cout << ")" << endl;

   // Creates an auxiliary vector object v5 to be used to initialize q5
   vector <int> v5;
   vector <int>::iterator v5_Iter;
   v5.push_back( 10 );
   v5.push_back( 30 );
   v5.push_back( 20 );
   cout << "v5 = ( " ;
   for ( v5_Iter = v5.begin( ) ; v5_Iter != v5.end( ) ; v5_Iter++ )
      cout << *v5_Iter << " ";
   cout << ")" << endl;

   // The fifth member function declares and
   // initializes a priority_queue q5 by copying the
   // range v5[_First, _Last) from vector v5
   priority_queue <int> q5( v5.begin( ), v5.begin( ) + 2 );
   cout << "q5 = ( ";
   while ( !q5.empty( ) )
   {
      cout << q5.top( ) << " ";
      q5.pop( );
   }
   cout << ")" << endl;

   // The sixth member function declares a priority_queue q6
   // with a comparison function greater and initializes q6
   // by copying the range v5[_First, _Last) from vector v5
   priority_queue <int, vector<int>, greater<int> >
      q6( v5.begin( ), v5.begin( ) + 2 );
   cout << "q6 = ( ";
   while ( !q6.empty( ) )
   {
      cout << q6.top( ) << " ";
      q6.pop( );
   }
   cout << ")" << endl;
}
```

Output

```
q1 = ( )
q2 = ( 15 10 5 )
After printing, q2 has 0 elements.
q3 = ( 1 2 3 )
q4 = ( 200 100 )
v5 = ( 10 30 20 )
q5 = ( 30 10 )
q6 = ( 10 30 )
```

Requirements

Header: <queue>

See Also

Reference

priority_queue Class

Standard Template Library

Other Resources

priority_queue Members

priority_queue::push

Adds an element to the priority queue based on the priority of the element from operator<.

```
void push(
   const Type& _Val
);
```

Parameters

_Val

The element added to the top of the priority_queue.

Remarks

The top of the priority_queue is the position occupied by the largest element in the container.

Example

```
// pqueue_push.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   priority_queue<int> q1;

   q1.push( 10 );
   q1.push( 30 );
   q1.push( 20 );

   priority_queue<int>::size_type i;
   i = q1.size( );
   cout << "The priority_queue length is " << i << "." << endl;

   const int& ii = q1.top( );
   cout << "The element at the top of the priority_queue is "
        << ii << "." << endl;
}
```

Output

```
The priority_queue length is 3.
The element at the top of the priority_queue is 30.
```

Requirements

Header: <queue>

See Also

Reference

priority_queue Class
priority_queue Functions
Standard Template Library

Other Resources

priority_queue Members

priority_queue::size

Returns the number of elements in the priority_queue.

```
size_type size( ) const;
```

Return Value

The current length of the priority_queue.

Example

```
// pqueue_size.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   priority_queue <int> q1, q2;
   priority_queue <int>::size_type i;

   q1.push( 1 );
   i = q1.size( );
   cout << "The priority_queue length is " << i << "." << endl;

   q1.push( 2 );
   i = q1.size( );
   cout << "The priority_queue length is now " << i << "." << endl;
}
```

Output

```
The priority_queue length is 1.
The priority_queue length is now 2.
```

Requirements

Header: <queue>

See Also

Reference

priority_queue Class

priority_queue Functions

Standard Template Library

Other Resources

priority_queue Members

priority_queue::top

Returns a const reference to the largest element at the top of the priority_queue.

```
const value_type& top( ) const;
```

Return Value

A const reference to the largest element, as determined by the **Traits** function, object of the priority_queue.

Remarks

The priority_queue must be nonempty to apply the member function.

Example

```
// pqueue_top.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   priority_queue<int> q1;

   q1.push( 10 );
   q1.push( 30 );
   q1.push( 20 );

   priority_queue<int>::size_type i;
   i = q1.size( );
   cout << "The priority_queue length is " << i << "." << endl;

   const int& ii = q1.top( );
   cout << "The element at the top of the priority_queue is "
        << ii << "." << endl;
}
```

Output

```
The priority_queue length is 3.
The element at the top of the priority_queue is 30.
```

Requirements

Header: <queue>

See Also

Reference

priority_queue Class
priority_queue Functions
Standard Template Library

Other Resources

priority_queue Members

<set>

Defines the container template classes set and multiset and their supporting templates.

For a list of all members of this header, see <set> Members.

```
#include <set>
```

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<set> Members

Standard C++ Library Header Files

<set> Members

Reference

Operators

Set version	Multiset version	Description
operator!= (set)	operator!= (multiset)	Tests if the set or multiset object on the left side of the operator is not equal to the set or multiset object on the right side.
operator< (set)	operator< (multiset)	Tests if the set or multiset object on the left side of the operator is less than the set or multiset object on the right side.
operator<= (set)	operator<= (multiset)	Tests if the set or multiset object on the left side of the operator is less than or equal to the set or multiset object on the right side.
operator== (set)	operator== (multiset)	Tests if the set or multiset object on the left side of the operator is equal to the set or multiset object on the right side.
operator> (set)	operator> (multiset)	Tests if the set or multiset object on the left side of the operator is greater than the set or multiset object on the right side.
operator>= (set)	operator>= (multiset)	Tests if the set or multiset object on the left side of the operator is greater than or equal to the set or multiset object on the right side.

Specialized Template Functions

Set version	Multiset version	Description
swap (set)	swap (multiset)	Exchanges the elements of two sets or multisets.

Classes

set Class	Used for the storage and retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values according to which the data is automatically ordered.
multiset Class	Used for the storage and retrieval of data from a collection in which the values of the elements contained need not be unique and in which they serve as the key values according to which the data is automatically ordered.

See Also

Reference

<set>

Thread Safety in the Standard C++ Library

Standard Template Library

<set> Operators

For more information about the operators in <set>, see <set> Members.

See Also

Reference

Standard Template Library

operator!= (set)

Tests if the set object on the left side of the operator is not equal to the set object on the right side.

```
bool operator!=(
   const set <Key, Traits, Allocator>& _Left,
   const set <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **set**.

_Right

An object of type **set**.

Return Value

true if the sets are not equal; **false** if sets are equal.

Remarks

The comparison between set objects is based on a pairwise comparison between their elements. Two sets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// set_op_ne.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1, s2, s3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i );
   }

   if ( s1 != s2 )
      cout << "The sets s1 and s2 are not equal." << endl;
   else
      cout << "The sets s1 and s2 are equal." << endl;

   if ( s1 != s3 )
      cout << "The sets s1 and s3 are not equal." << endl;
   else
      cout << "The sets s1 and s3 are equal." << endl;
}
```

Output

```
The sets s1 and s2 are not equal.
The sets s1 and s3 are equal.
```

Requirements

Header: <set>

See Also

Reference

Standard Template Library

Other Resources

<set> Members

operator< (set)

Tests if the set object on the left side of the operator is less than the set object on the right side.

```
bool operator<(
   const set <Key, Traits, Allocator>& _Left,
   const set <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **set**.

_Right

An object of type **set**.

Return Value

true if the set on the left side of the operator is strictly less than the set on the right side of the operator; otherwise **false**.

Remarks

The comparison between set objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// set_op_lt.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1, s2, s3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i - 1 );
   }

   if ( s1 < s2 )
      cout << "The set s1 is less than the set s2." << endl;
   else
      cout << "The set s1 is not less than the set s2." << endl;

   if ( s1 < s3 )
      cout << "The set s1 is less than the set s3." << endl;
   else
      cout << "The set s1 is not less than the set s3." << endl;
}
```

Output

```
The set s1 is less than the set s2.
The set s1 is not less than the set s3.
```

Requirements

Header: <set>

See Also

Reference

Standard Template Library

Other Resources

<set> Members

operator<= (set)

Tests if the set object on the left side of the operator is less than or equal to the set object on the right side.

```
bool operator!<=(
   const set <Key, Traits, Allocator>& _Left,
   const set <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **set**.

_Right

An object of type **set**.

Return Value

true if the set on the left side of the operator is less than or equal to the set on the right side of the operator; otherwise **false**.

Remarks

The comparison between set objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// set_op_le.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1, s2, s3, s4;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i - 1 );
      s4.insert ( i );
   }

   if ( s1 <= s2 )
      cout << "Set s1 is less than or equal to the set s2." << endl;
   else
      cout << "The set s1 is greater than the set s2." << endl;

   if ( s1 <= s3 )
      cout << "Set s1 is less than or equal to the set s3." << endl;
   else
      cout << "The set s1 is greater than the set s3." << endl;

   if ( s1 <= s4 )
      cout << "Set s1 is less than or equal to the set s4." << endl;
   else
      cout << "The set s1 is greater than the set s4." << endl;
}
```

Output

```
Set s1 is less than or equal to the set s2.
The set s1 is greater than the set s3.
Set s1 is less than or equal to the set s4.
```

Requirements

Header: <set>

See Also

Reference

Standard Template Library

Other Resources

<set> Members

operator== (set)

Tests if the set object on the left side of the operator is equal to the set object on the right side.

```
bool operator!==(
   const set <Key, Traits, Allocator>& _Left,
   const set <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **set**.

_Right

An object of type **set**.

Return Value

true if the set on the left side of the operator is equal to the set on the right side of the operator; otherwise **false**.

Remarks

The comparison between set objects is based on a pairwise comparison of their elements. Two sets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// set_op_eq.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1, s2, s3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i );
   }

   if ( s1 == s2 )
      cout << "The sets s1 and s2 are equal." << endl;
   else
      cout << "The sets s1 and s2 are not equal." << endl;

   if ( s1 == s3 )
      cout << "The sets s1 and s3 are equal." << endl;
   else
      cout << "The sets s1 and s3 are not equal." << endl;
}
```

Output

```
The sets s1 and s2 are not equal.
The sets s1 and s3 are equal.
```

Requirements

Header: <set>

See Also

Reference

Standard Template Library

Other Resources

<set> Members

operator> (set)

Tests if the set object on the left side of the operator is greater than the set object on the right side.

```
bool operator>(
   const set <Key, Traits, Allocator>& _Left,
   const set <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **set**.

_Right

An object of type **set**.

Return Value

true if the set on the left side of the operator is greater than the set on the right side of the operator; otherwise **false**.

Remarks

The comparison between set objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// set_op_gt.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1, s2, s3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i - 1 );
   }

   if ( s1 > s2 )
      cout << "The set s1 is greater than the set s2." << endl;
   else
      cout << "The set s1 is not greater than the set s2." << endl;

   if ( s1 > s3 )
      cout << "The set s1 is greater than the set s3." << endl;
   else
      cout << "The set s1 is not greater than the set s3." << endl;
}
```

Output

```
The set s1 is not greater than the set s2.
The set s1 is greater than the set s3.
```

Requirements

Header: <set>

See Also

Reference

Standard Template Library

Other Resources

<set> Members

operator>= (set)

Tests if the set object on the left side of the operator is greater than or equal to the set object on the right side.

```
bool operator!>=(
   const set <Key, Traits, Allocator>& _Left,
   const set <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **set**.

_Right

An object of type **set**.

Return Value

true if the set on the left side of the operator is greater than or equal to the set on the right side of the list; otherwise **false**.

Remarks

The comparison between set objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// set_op_ge.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1, s2, s3, s4;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i - 1 );
      s4.insert ( i );
   }

   if ( s1 >= s2 )
      cout << "Set s1 is greater than or equal to set s2." << endl;
   else
      cout << "The set s1 is less than the set s2." << endl;

   if ( s1 >= s3 )
      cout << "Set s1 is greater than or equal to set s3." << endl;
   else
      cout << "The set s1 is less than the set s3." << endl;

   if ( s1 >= s4 )
      cout << "Set s1 is greater than or equal to set s4." << endl;
   else
      cout << "The set s1 is less than the set s4." << endl;
}
```

Output

```
The set s1 is less than the set s2.
Set s1 is greater than or equal to set s3.
Set s1 is greater than or equal to set s4.
```

Requirements

Header: <set>

See Also

Reference

Standard Template Library

Other Resources

<set> Members

operator!= (multiset)

Tests if the multiset object on the left side of the operator is not equal to the multiset object on the right side.

```
bool operator!=(
   const multiset <Key, Traits, Allocator>& _Left,
   const multiset <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **multiset**.

_Right

An object of type **multiset**.

Return Value

true if the sets or multisets are not equal; **false** if sets or multisets are equal.

Remarks

The comparison between multiset objects is based on a pairwise comparison between their elements. Two sets or multisets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// multiset_op_ne.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> s1, s2, s3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i );
   }

   if ( s1 != s2 )
      cout << "The multisets s1 and s2 are not equal." << endl;
   else
      cout << "The multisets s1 and s2 are equal." << endl;

   if ( s1 != s3 )
      cout << "The multisets s1 and s3 are not equal." << endl;
   else
      cout << "The multisets s1 and s3 are equal." << endl;
}
```

Output

```
The multisets s1 and s2 are not equal.
The multisets s1 and s3 are equal.
```

Requirements

Header: <set>

See Also

Reference

Standard Template Library

Other Resources

<set> Members

operator< (multiset)

Tests if the multiset object on the left side of the operator is less than the multiset object on the right side.

```
bool operator<(
   const multiset <Key, Traits, Allocator>& _Left,
   const multiset <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **multiset**.

_Right

An object of type **multiset**.

Return Value

true if the multiset on the left side of the operator is strictly less than the multiset on the right side of the operator; otherwise **false**.

Remarks

The comparison between multiset objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// multiset_op_lt.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> s1, s2, s3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i - 1 );
   }

   if ( s1 < s2 )
      cout << "The multiset s1 is less than "
           << "the multiset s2." << endl;
   else
      cout << "The multiset s1 is not less than "
           << "the multiset s2." << endl;

   if ( s1 < s3 )
      cout << "The multiset s1 is less than "
           << "the multiset s3." << endl;
   else
      cout << "The multiset s1 is not less than "
           << "the multiset s3." << endl;
}
```

Output

```
The multiset s1 is less than the multiset s2.
The multiset s1 is not less than the multiset s3.
```

Requirements

Header: <set>

See Also

Reference

Standard Template Library

Other Resources

<set> Members

operator<= (multiset)

Tests if the multiset object on the left side of the operator is less than or equal to the multiset object on the right side.

```
bool operator!<=(
   const multiset <Key, Traits, Allocator>& _Left,
   const multiset <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **multiset**.

_Right

An object of type **multiset**.

Return Value

true if the multiset on the left side of the operator is less than or equal to the multiset on the right side of the operator; otherwise **false**.

Remarks

The comparison between multiset objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// multiset_op_le.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> s1, s2, s3, s4;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i - 1 );
      s4.insert ( i );
   }

   if ( s1 <= s2 )
      cout << "The multiset s1 is less than "
           << "or equal to the multiset s2." << endl;
   else
      cout << "The multiset s1 is greater than "
           << "the multiset s2." << endl;

   if ( s1 <= s3 )
      cout << "The multiset s1 is less than "
           << "or equal to the multiset s3." << endl;
   else
      cout << "The multiset s1 is greater than "
           << "the multiset s3." << endl;

   if ( s1 <= s4 )
      cout << "The multiset s1 is less than "
           << "or equal to the multiset s4." << endl;
```

```
    else
       cout << "The multiset s1 is greater than "
            << "the multiset s4." << endl;
 }
```

Output

```
The multiset s1 is less than or equal to the multiset s2.
The multiset s1 is greater than the multiset s3.
The multiset s1 is less than or equal to the multiset s4.
```

Requirements

Header: <set>

See Also

Reference

Standard Template Library

Other Resources

<set> Members

operator== (multiset)

Tests if the multiset object on the left side of the operator is equal to the multiset object on the right side.

```
bool operator!==(
   const multiset <Key, Traits, Allocator>& _Left,
   const multiset <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **multiset**.

_Right

An object of type **multiset**.

Return Value

true if the multiset on the left side of the operator is equal to the multiset on the right side of the operator; otherwise **false**.

Remarks

The comparison between multiset objects is based on a pairwise comparison of their elements. Two sets or multisets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// multiset_op_eq.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> s1, s2, s3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i );
   }

   if ( s1 == s2 )
      cout << "The multisets s1 and s2 are equal." << endl;
   else
      cout << "The multisets s1 and s2 are not equal." << endl;

   if ( s1 == s3 )
      cout << "The multisets s1 and s3 are equal." << endl;
   else
      cout << "The multisets s1 and s3 are not equal." << endl;
}
```

Output

```
The multisets s1 and s2 are not equal.
The multisets s1 and s3 are equal.
```

Requirements

Header: <set>

See Also

Reference

Standard Template Library

Other Resources

<set> Members

operator> (multiset)

Tests if the multiset object on the left side of the operator is greater than the multiset object on the right side.

```
bool operator>(
   const multiset <Key, Traits, Allocator>& _Left,
   const multiset <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **multiset**.

_Right

An object of type **multiset**.

Return Value

true if the multiset on the left side of the operator is greater than the multiset on the right side of the operator; otherwise **false**.

Remarks

The comparison between multiset objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// multiset_op_gt.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> s1, s2, s3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i - 1 );
   }

   if ( s1 > s2 )
      cout << "The multiset s1 is greater than "
           << "the multiset s2." << endl;
   else
      cout << "The multiset s1 is not greater "
           << "than the multiset s2." << endl;

   if ( s1 > s3 )
      cout << "The multiset s1 is greater than "
           << "the multiset s3." << endl;
   else
      cout << "The multiset s1 is not greater than "
           << "the multiset s3." << endl;
}
```

Output

```
The multiset s1 is not greater than the multiset s2.
The multiset s1 is greater than the multiset s3.
```

Requirements

Header: <set>

See Also

Reference

Standard Template Library

Other Resources

<set> Members

operator>= (multiset)

Tests if the multiset object on the left side of the operator is greater than or equal to the multiset object on the right side.

```
bool operator!>=(
   const multiset <Key, Traits, Allocator>& _Left,
   const multiset <Key, Traits, Allocator>& _Right
);
```

Parameters

_Left

An object of type **multiset**.

_Right

An object of type **multiset**.

Return Value

true if the multiset on the left side of the operator is greater than or equal to the multiset on the right side of the list; otherwise **false**.

Remarks

The comparison between multiset objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// multiset_op_ge.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> s1, s2, s3, s4;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i - 1 );
      s4.insert ( i );
   }

   if ( s1 >= s2 )
      cout << "The multiset s1 is greater than "
           << "or equal to the multiset s2." << endl;
   else
      cout << "The multiset s1 is less than "
           << "the multiset s2." << endl;

   if ( s1 >= s3 )
      cout << "The multiset s1 is greater than "
           << "or equal to the multiset s3." << endl;
   else
      cout << "The multiset s1 is less than "
           << "the multiset s3." << endl;

   if ( s1 >= s4 )
      cout << "The multiset s1 is greater than "
           << "or equal to the multiset s4." << endl;
```

```
    else
       cout << "The multiset s1 is less than "
            << "the multiset s4." << endl;
 }
```

Output

```
The multiset s1 is less than the multiset s2.
The multiset s1 is greater than or equal to the multiset s3.
The multiset s1 is greater than or equal to the multiset s4.
```

Requirements

Header: <set>

See Also

Reference

Standard Template Library

Other Resources

<set> Members

<set> Specialized Template Functions

For more information about the specialized template functions in <set>, see <set> Members.

See Also

Reference

Standard Template Library

swap (set)

Exchanges the elements of two sets.

```
template<class Key, class Traits, class Allocator>
void swap(
   set< Key, Traits, Allocator>& _Left,
   set< Key, Traits, Allocator>& _Right
);
```

Parameters

_Right

The set providing the elements to be swapped, or the set whose elements are to be exchanged with those of the set *_Left*.

_Left

The set whose elements are to be exchanged with those of the set *_Right*.

Remarks

The template function is an algorithm specialized on the container class set to execute the member function *_Left*.swap(*_Right*). This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function

template <**class T**> **void swap**(**T&**, **T&**)

in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

Example

See the code example for the member class set::swap for an example of the use of the template version of **swap**.

Requirements

Header: <set>

See Also

Reference

Standard Template Library

Other Resources

<set> Members

swap (multiset)

Exchanges the elements of two multisets.

```
template<class Key, class Traits, class Allocator>
void swap(
   multiset< Key, Traits, Allocator>& _Left,
   multiset< Key, Traits, Allocator>& _Right
);
```

Parameters

_Right

The multiset providing the elements to be swapped, or the multiset whose elements are to be exchanged with those of the multiset *_Left*.

_Left

The multiset whose elements are to be exchanged with those of the multiset *_Right*.

Remarks

The template function is an algorithm specialized on the container class multiset to execute the member function *_Left*.swap(*_Right*). This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function

template <**class T**> **void swap**(**T&**, **T&**)

in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

Example

See the code example for the member class multiset::swapfor an example of the use of the template version of **swap**.

Requirements

Header: <set>

See Also

Reference

Standard Template Library

Other Resources

<set> Members

<set> Classes

For information about the classes in <set>, see <set> Members.

See Also

Reference

Standard Template Library

set Class

The STL container class set is used for the storage and retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values according to which the data is automatically ordered. The value of an element in a set may not be changed directly. Instead, you must delete old values and insert elements with new values.

For a list of all members of this type, see set Members.

```
template <
   class Key,
   class Traits=less<Key>,
   class Allocator=allocator<Key>
>
class set
```

Parameters

Key

The element data type to be stored in the set.

Traits

The type that provides a function object that can compare two element values as sort keys to determine their relative order in the set. This argument is optional, and the binary predicate **less** <*Key*> is the default value.

Allocator

The type that represents the stored allocator object that encapsulates details about the set's allocation and deallocation of memory. This argument is optional, and the default value is **allocator**<*Key*>.

Remarks

An STL set is:

- An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value. Further, it is a simple associative container because its element values are its key values.
- Reversible, because it provides a bidirectional iterator to access its elements.
- Sorted, because its elements are ordered by key values within the container in accordance with a specified comparison function.
- Unique in the sense that each of its elements must have a unique key. Since set is also a simple associative container, its elements are also unique.

A set is also described as a template class because the functionality it provides is generic and independent of the specific type of data contained as elements. The data type to be used is, instead, specified as a parameter in the class template along with the comparison function and allocator.

The choice of container type should be based in general on the type of searching and inserting required by the application. Associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient, performing them in a time that is on average proportional to the logarithm of the number of elements in the container. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

The set should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. The elements of a set are unique and serve as their own sort keys. A model for this type of structure is an ordered list of, say, words in which the words may occur only once. If multiple occurrences of the words were allowed, then a multiset would be the appropriate container structure. If unique definitions were attached as values to the list of key words, then a map would be an appropriate structure to contain this data. If instead the definitions were not unique, then a multimap would be the container of choice.

The set orders the sequence it controls by calling a stored function object of type key_compare. This stored object is a comparison function that may be accessed by calling the member function key_comp. In general, the elements need to be merely less than comparable to establish this order so that, given any two elements, it may be determined either that they are

equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate *f*(*x*,*y*) is a function object that has two argument objects *x* and *y* and a return value of **true** or **false**. An ordering imposed on a set is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects *x* and *y* are defined to be equivalent when both *f*(*x*,*y*) and *f*(*y*,*x*) are false. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

The iterator provided by the set class is a bidirectional iterator, but the class member functions insert and set have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own set of requirements, and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal set of functionality, but it is enough to be able to talk meaningfully about a range of iterators [*_First*, *_Last*) in the context of the class's member functions.

Requirements

Header: <set>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

set Members

set Members

Reference

Constructors

set	Constructs a set that is empty or that is a copy of all or part of some other set.

Typedefs

allocator_type	A type that represents the **allocator** class for the set object.
const_iterator	A type that provides a bidirectional iterator that can read a **const** element in the set.
const_pointer	A type that provides a pointer to a **const** element in a set.
const_reference	A type that provides a reference to a **const** element stored in a set for reading and performing **const** operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any **const** element in the set.
difference_type	A signed integer type that can be used to represent the number of elements of a set in a range between elements pointed to by iterators.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a set.
key_compare	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the set.
key_type	The type describes an object stored as an element of a set in its capacity as sort key.
pointer	A type that provides a pointer to an element in a set.
reference	A type that provides a reference to an element stored in a set.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed set.
size_type	An unsigned integer type that can represent the number of elements in a set.
value_compare	The type that provides a function object that can compare two elements to determine their relative order in the set.
value_type	The type describes an object stored as an element of a set in its capacity as a value.

Member Functions

begin	Returns an iterator that addresses the first element in the set.
clear	Erases all the elements of a set.
count	Returns the number of elements in a set whose key matches a parameter-specified key.
empty	Tests if a set is empty.
end	Returns an iterator that addresses the location succeeding the last element in a set.

equal_range	Returns a pair of iterators respectively to the first element in a set with a key that is greater than a specified key and to the first element in the set with a key that is equal to or greater than the key.
erase	Removes an element or a range of elements in a set from specified positions or removes elements that match a specified key.
find	Returns an iterator addressing the location of an element in a set that has a key equivalent to a specified key.
get_allocator	Returns a copy of the **allocator** object used to construct the set.
insert	Inserts an element or a range of elements into a set.
key_comp	Retrieves a copy of the comparison object used to order keys in a set.
lower_bound	Returns an iterator to the first element in a set with a key that is equal to or greater than a specified key.
max_size	Returns the maximum length of the set.
rbegin	Returns an iterator addressing the first element in a reversed set.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed set.
size	Returns the number of elements in the set.
swap	Exchanges the elements of two sets.
upper_bound	Returns an iterator to the first element in a set with a key that is greater than a specified key.
value_comp	Retrieves a copy of the comparison object used to order element values in a set.

See Also

Reference

set Class

Thread Safety in the Standard C++ Library

Standard Template Library

set Typedefs

For information about the typedefs in the set class, see set Members.

See Also

Reference

Standard Template Library

set::allocator_type

A type that represents the allocator class for the set object.

```
typedef Allocator allocator_type
```

Remarks

allocator_type is a synonym for the template parameter Allocator.

Returns the function object that a multiset uses to order its elements, which is the template parameter **Allocator**.

For more information on **Allocator**, see the Remarks section of the set Class topic.

Example

See the example for get_allocator for an example that uses **allocator_type**.

Requirements

Header: <set>

See Also

Reference

set Class

Standard Template Library

Other Resources

set Members

set::const_iterator

A type that provides a bidirectional iterator that can read a **const** element in the set.

```
typedef implementation-defined const_iterator;
```

Remarks

A type **const_iterator** cannot be used to modify the value of an element.

Example

See the example for begin for an example that uses **const_iterator**.

Requirements

Header: <set>

See Also

Reference

set Class

Standard Template Library

Other Resources

set Members

set::const_pointer

A type that provides a pointer to a **const** element in a set.

```
typedef typename Allocator::const_pointer const_pointer;
```

Remarks

A type **const_pointer** cannot be used to modify the value of an element.

In most cases, a const_iterator should be used to access the elements in a const set object.

Requirements

Header: <set>

See Also

Reference

set Class

Standard Template Library

Other Resources

set Members

set::const_reference

A type that provides a reference to a **const** element stored in a set for reading and performing **const** operations.

```
typedef typename Allocator::const_reference const_reference;
```

Example

```
// set_const_ref.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;

   s1.insert( 10 );
   s1.insert( 20 );

   // Declare and initialize a const_reference &Ref1
   // to the 1st element
   const int &Ref1 = *s1.begin( );

   cout << "The first element in the set is "
        << Ref1 << "." << endl;

   // The following line would cause an error because the
   // const_reference cannot be used to modify the set
   // Ref1 = Ref1 + 5;
}
```

Output

```
The first element in the set is 10.
```

Requirements

Header: <set>

See Also

Reference

set Class

Standard Template Library

Other Resources

set Members

set::const_reverse_iterator

A type that provides a bidirectional iterator that can read any **const** element in the set.

```
typedef reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type **const_reverse_iterator** cannot modify the value of an element and is use to iterate through the set in reverse.

Example

See the example for rend for an example of how to declare and use the **const_reverse_iterator**.

Requirements

Header: <set>

See Also

Reference

set Class

Standard Template Library

Other Resources

set Members

set::difference_type

A signed integer type that can be used to represent the number of elements of a set in a range between elements pointed to by iterators.

```
typedef implementation-defined difference_type;
```

Remarks

The **difference_type** is the type returned when subtracting or incrementing through iterators of the container. The **difference_type** is typically used to represent the number of elements in the range *[_First, _Last)* between the iterators *_First* and *_Last*, includes the element pointed to by *_First* and the range of elements up to, but not including, the element pointed to by *_Last*.

Note that although **difference_type** is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as set, subtraction between iterators is only supported by random-access iterators provided by a random-access container such as vector.

Example

```
// set_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <set>
#include <algorithm>

int main( )
{
   using namespace std;

   set <int> s1;
   set <int>::iterator s1_Iter, s1_bIter, s1_eIter;

   s1.insert( 20 );
   s1.insert( 10 );
   s1.insert( 20 );   // won't insert as set elements are unique

   s1_bIter = s1.begin( );
   s1_eIter = s1.end( );

   set <int>::difference_type   df_typ5, df_typ10, df_typ20;

   df_typ5 = count( s1_bIter, s1_eIter, 5 );
   df_typ10 = count( s1_bIter, s1_eIter, 10 );
   df_typ20 = count( s1_bIter, s1_eIter, 20 );

   // the keys, and hence the elements of a set are unique,
   // so there is at most one of a given value
   cout << "The number '5' occurs " << df_typ5
        << " times in set s1.\n";
   cout << "The number '10' occurs " << df_typ10
        << " times in set s1.\n";
   cout << "The number '20' occurs " << df_typ20
        << " times in set s1.\n";

   // count the number of elements in a set
   set <int>::difference_type  df_count = 0;
   s1_Iter = s1.begin( );
   while ( s1_Iter != s1_eIter)
   {
      df_count++;
      s1_Iter++;
   }
```

```
    cout << "The number of elements in the set s1 is: "
         << df_count << "." << endl;
}
```

Output

```
The number '5' occurs 0 times in set s1.
The number '10' occurs 1 times in set s1.
The number '20' occurs 1 times in set s1.
The number of elements in the set s1 is: 2.
```

Requirements

Header: <set>

See Also

Reference

set Class

Standard Template Library

Other Resources

set Members

set::iterator

A type that provides a bidirectional iterator that can read or modify any element in a set.

```
typedef implementation-defined iterator;
```

Remarks

A type **iterator** can be used to modify the value of an element.

Example

See the example for begin for an example of how to declare and use **iterator**.

Requirements

Header: <set>

See Also

Reference

set Class

Standard Template Library

Other Resources

set Members

set::key_compare

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the set.

```
typedef Traits key_compare;
```

Remarks

key_compare is a synonym for the template parameter **Traits**.

For more information on **Traits** see the set Class topic.

Note that both **key_compare** and value_compare are synonyms for the template parameter **Traits**. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

See the example for key_comp for an example of how to declare and use **key_compare**.

Requirements

Header: <set>

See Also

Reference

set Class

Standard Template Library

Other Resources

set Members

set::key_type

A type that describes an object stored as an element of a set in its capacity as sort key.

```
typedef Key key_type;
```

Remarks

key_type is a synonym for the template parameter **Key**.

For more information on **Key**, see the Remarks section of the set Class topic.

Note that both **key_type** and value_type are synonyms for the template parameter **Key**. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

See the example for value_type for an example of how to declare and use **key_type**.

Requirements

Header: <set>

See Also

Reference

set Class

Standard Template Library

Other Resources

set Members

set::pointer

A type that provides a pointer to an element in a set.

```
typedef typename Allocator::pointer pointer;
```

Remarks

A type **pointer** can be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a set object.

Requirements

Header: <set>

See Also

Reference

set Class

Standard Template Library

Other Resources

set Members

set::reference

A type that provides a reference to an element stored in a set.

```
typedef typename Allocator::reference reference;
```

Example

```
// set_reference.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;

   s1.insert( 10 );
   s1.insert( 20 );

   // Declare and initialize a reference &Ref1 to the 1st element
   int &Ref1 = *s1.begin( );

   cout << "The first element in the set is "
        << Ref1 << "." << endl;

   // The value of the 1st element of the set can be changed
   // by operating on its (non const) reference
   Ref1 = Ref1 + 5;

   cout << "The first element in the set is now "
        << *s1.begin( ) << "." << endl;
}
```

Output

```
The first element in the set is 10.
The first element in the set is now 15.
```

Requirements

Header: <set>

See Also

Reference

set Class

Standard Template Library

Other Resources

set Members

set::reverse_iterator

A type that provides a bidirectional iterator that can read or modify an element in a reversed set.

```
typedef reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type **reverse_iterator** is use to iterate through the set in reverse.

Example

See the example for rbegin for an example of how to declare and use **reverse_iterator**.

Requirements

Header: <set>

See Also

Reference

set Class

Standard Template Library

Other Resources

set Members

set::size_type

An unsigned integer type that can represent the number of elements in a set.

```
typedef implementation-defined size_type;
```

Example

See the example for size for an example of how to declare and use **size_type**

Requirements

Header: <set>

See Also

Reference

set Class

Standard Template Library

Other Resources

set Members

set::value_compare

A type that provides a function object that can compare two element values to determine their relative order in the set.

```
typedef Traits value_compare;
```

Remarks

value_compare is a synonym for the template parameter **Traits**.

For more information on **Traits** see the set Class topic.

Note that both key_compare and **value_compare** are synonyms for the template parameter **Traits**. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

See the example for value_comp for an example of how to declare and use **value_compare**.

Requirements

Header: <set>

See Also

Reference

set Class

Standard Template Library

Other Resources

set Members

set::value_type

A type that describes an object stored as an element of a set in its capacity as a value.

```
typedef Key value_type;
```

Remarks

value_type is a synonym for the template parameter **Key**.

For more information on **Key**, see the Remarks section of the set Class topic.

Note that both key_type and **value_type** are synonyms for the template parameter **Key**. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

```
// set_value_type.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int>::iterator s1_Iter;

   set <int>::value_type svt_Int;   // Declare value_type
   svt_Int = 10;            // Initialize value_type

   set <int> :: key_type skt_Int;   // Declare key_type
   skt_Int = 20;             // Initialize key_type

   s1.insert( svt_Int );         // Insert value into s1
   s1.insert( skt_Int );         // Insert key into s1

   // A set accepts key_types or value_types as elements
   cout << "The set has elements:";
   for ( s1_Iter = s1.begin( ) ; s1_Iter != s1.end( ); s1_Iter++)
      cout << " " << *s1_Iter;
   cout << "." << endl;
}
```

Output

```
The set has elements: 10 20.
```

Requirements

Header: <set>

See Also

Reference

set Class

Standard Template Library

Other Resources

set Members

set Member Functions

For information about the functions in the set class, see set Members.

See Also

Reference

Standard Template Library

set::begin

Returns an iterator that addresses the first element in the set.

```
const_iterator begin( ) const;
iterator begin( );
```

Return Value

A bidirectional iterator addressing the first element in the set or the location succeeding an empty set.

Remarks

If the return value of **begin** is assigned to a **const_iterator**, the elements in the set object cannot be modified. If the return value of **begin** is assigned to an **iterator**, the elements in the set object can be modified.

Example

```
// set_begin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int>::iterator s1_Iter;
   set <int>::const_iterator s1_cIter;

   s1.insert( 1 );
   s1.insert( 2 );
   s1.insert( 3 );

   s1_Iter = s1.begin( );
   cout << "The first element of s1 is " << *s1_Iter << endl;

   s1_Iter = s1.begin( );
   s1.erase( s1_Iter );

   // The following 2 lines would err because the iterator is const
   // s1_cIter = s1.begin( );
   // s1.erase( s1_cIter );

   s1_cIter = s1.begin( );
   cout << "The first element of s1 is now " << *s1_cIter << endl;
}
```

Output

```
The first element of s1 is 1
The first element of s1 is now 2
```

Requirements

Header: <set>

See Also

Reference

set Class

set::swap, set::begin, and set::end

Standard Template Library

Other Resources

set Members

set::clear

Erases all the elements of a set.

```
void clear( );
```

Example

```
// set_clear.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;

   s1.insert( 1 );
   s1.insert( 2 );

   cout << "The size of the set is initially " << s1.size( )
        << "." << endl;

   s1.clear( );
   cout << "The size of the set after clearing is "
        << s1.size( ) << "." << endl;
}
```

Output

```
The size of the set is initially 2.
The size of the set after clearing is 0.
```

Requirements

Header: <set>

See Also

Reference

set Class

set::empty and set::clear

Standard Template Library

Other Resources

set Members

set::count

Returns the number of elements in a set whose key matches a parameter-specified key.

```
size_type count(
   const Key& _Key
) const;
```

Parameters

_Key

The key of the elements to be matched from the set.

Return Value

1 if the set contains an element whose sort key matches the parameter key. 0 if the set does not contain an element with a matching key.

Remarks

The member function returns the number of elements in the following range:

[**lower_bound** (*_Key*), **upper_bound** (*_Key*)).

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// set_count.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main()
{
    using namespace std;
    set<int> s1;
    set<int>::size_type i;

    s1.insert(1);
    s1.insert(1);

    // Keys must be unique in set, so duplicates are ignored
    i = s1.count(1);
    cout << "The number of elements in s1 with a sort key of 1 is: "
         << i << "." << endl;

    i = s1.count(2);
    cout << "The number of elements in s1 with a sort key of 2 is: "
         << i << "." << endl;
}
```

Output

```
The number of elements in s1 with a sort key of 1 is: 1.
The number of elements in s1 with a sort key of 2 is: 0.
```

Requirements

Header: <set>

See Also

Reference

set Class

set::count (STL Samples)
Standard Template Library
Other Resources
set Members

set::empty

Tests if a set is empty.

```
bool empty( ) const;
```

Return Value

true if the set is empty; **false** if the set is nonempty.

Example

```
// set_empty.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1, s2;
   s1.insert ( 1 );

   if ( s1.empty( ) )
      cout << "The set s1 is empty." << endl;
   else
      cout << "The set s1 is not empty." << endl;

   if ( s2.empty( ) )
      cout << "The set s2 is empty." << endl;
   else
      cout << "The set s2 is not empty." << endl;
}
```

Output

```
The set s1 is not empty.
The set s2 is empty.
```

Requirements

Header: <set>

See Also

Reference

set Class

set::empty and set::clear

Standard Template Library

Other Resources

set Members

set::end

Returns an iterator that addresses the location succeeding the last element in a set.

```
const_iterator end( ) const;
iterator end( );
```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a set. If the set is empty, then set::end ==set::begin.

Remarks

end is used to test whether an iterator has reached the end of its set. The value returned by **end** should not be dereferenced.

Example

```
// set_end.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int> :: iterator s1_Iter;
   set <int> :: const_iterator s1_cIter;

   s1.insert( 1 );
   s1.insert( 2 );
   s1.insert( 3 );

   s1_Iter = s1.end( );
   s1_Iter--;
   cout << "The last element of s1 is " << *s1_Iter << endl;

   s1.erase( s1_Iter );

   // The following 3 lines would err because the iterator is const
   // s1_cIter = s1.end( );
   // s1_cIter--;
   // s1.erase( s1_cIter );

   s1_cIter = s1.end( );
   s1_cIter--;
   cout << "The last element of s1 is now " << *s1_cIter << endl;
}
```

Output

```
The last element of s1 is 3
The last element of s1 is now 2
```

Requirements

Header: <set>

See Also

Reference

set Class

set::swap, set::begin, and set::end

Standard Template Library

Other Resources

set Members

set::equal_range

Returns a pair of iterators respectively to the first element in a set with a key that is greater than a specified key and to the first element in the set with a key that is equal to or greater than the key.

```
pair <const_iterator, const_iterator> equal_range (
   const Key& _Key
) const;
pair <iterator, iterator> equal_range (
   const Key& _Key
);
```

Parameters

_Key

The argument key to be compared with the sort key of an element from the set being searched.

Return Value

A pair of iterators where the first is the lower_bound of the key and the second is the upper_bound of the key.

To access the first iterator of a pair *pr* returned by the member function, use *pr*.**first**, and to dereference the lower bound iterator, use *(*pr*.**first**). To access the second iterator of a pair *pr* returned by the member function, use *pr*.**second**, and to dereference the upper bound iterator, use *(*pr*.**second**).

Example

```
// set_equal_range.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   typedef set<int, less< int > > IntSet;
   IntSet s1;
   set <int, less< int > > :: const_iterator s1_RcIter;

   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );

   pair <IntSet::const_iterator, IntSet::const_iterator> p1, p2;
   p1 = s1.equal_range( 20 );

   cout << "The upper bound of the element with "
        << "a key of 20 in the set s1 is: "
        << *(p1.second) << "." << endl;

   cout << "The lower bound of the element with "
        << "a key of 20 in the set s1 is: "
        << *(p1.first) << "." << endl;

   // Compare the upper_bound called directly
   s1_RcIter = s1.upper_bound( 20 );
   cout << "A direct call of upper_bound( 20 ) gives "
        << *s1_RcIter << "," << endl
        << "matching the 2nd element of the pair"
        << " returned by equal_range( 20 )." << endl;

   p2 = s1.equal_range( 40 );

   // If no match is found for the key,
   // both elements of the pair return end( )
```

```
    if ( ( p2.first == s1.end( ) ) && ( p2.second == s1.end( ) ) )
       cout << "The set s1 doesn't have an element "
            << "with a key less than 40." << endl;
    else
       cout << "The element of set s1 with a key >= 40 is: "
            << *(p1.first) << "." << endl;
}
```

Output

```
The upper bound of the element with a key of 20 in the set s1 is: 30.
The lower bound of the element with a key of 20 in the set s1 is: 20.
A direct call of upper_bound( 20 ) gives 30,
matching the 2nd element of the pair returned by equal_range( 20 ).
The set s1 doesn't have an element with a key less than 40.
```

Requirements

Header: <set>

See Also

Reference

set Class

set::lower_bound, set::upper_bound, and set::equal_range

Standard Template Library

Other Resources

set Members

set::erase

Removes an element or a range of elements in a set from specified positions or removes elements that match a specified key.

```
iterator erase(
   iterator _Where
);
iterator erase(
   iterator _First,
   iterator _Last
);
size_type erase(
   const key_type& _Key
);
```

Parameters

_Where

Position of the element to be removed from the set.

_First

Position of the first element removed from the set.

_Last

Position just beyond the last element removed from the set.

_Key

The key of the elements to be removed from the set.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the set if no such element exists. For the third member function, the number of elements that have been removed from the set.

Note
This return type does not conform to the C++ standard.

Remarks

In some instances, this method might throw an out_of_range exception. This behavior does not conform to the C++ standard.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// set_erase.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main()
{
    using namespace std;
    set<int> s1, s2, s3;
    set<int>::iterator pIter, Iter1, Iter2;
    int i;
    set<int>::size_type n;

    for (i = 1; i < 5; i++)
    {
```

```
        s1.insert(i);
        s2.insert(i * i);
        s3.insert(i - 1);
    }

    // The 1st member function removes an element at a given position
    Iter1 = ++s1.begin();
    s1.erase(Iter1);

    cout << "After the 2nd element is deleted, the set s1 is:";
    for (pIter = s1.begin(); pIter != s1.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;

    // The 2nd member function removes elements
    // in the range [_First, _Last)
    Iter1 = ++s2.begin();
    Iter2 = --s2.end();
    s2.erase(Iter1, Iter2);

    cout << "After the middle two elements are deleted, "
         << "the set s2 is:";
    for (pIter = s2.begin(); pIter != s2.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;

    // The 3rd member function removes elements with a given _Key
    n = s3.erase(2);

    cout << "After the element with a key of 2 is deleted, "
         << "the set s3 is:";
    for (pIter = s3.begin(); pIter != s3.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;

    // The 3rd member function returns the number of elements removed
    cout << "The number of elements removed from s3 is: "
         << n << "." << endl;

    // The dereferenced iterator can also be used to specify a key
    Iter1 = ++s3.begin();
    s3.erase(Iter1);

    cout << "After another element (unique for set) with a key"
         << endl;
    cout << "equal to that of the 2nd element is deleted, "
         << "the set s3 is:";
    for (pIter = s3.begin(); pIter != s3.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;
}
```

Output

```
After the 2nd element is deleted, the set s1 is: 1 3 4.
After the middle two elements are deleted, the set s2 is: 1 16.
After the element with a key of 2 is deleted, the set s3 is: 0 1 3.
The number of elements removed from s3 is: 1.
After another element (unique for set) with a key
equal to that of the 2nd element is deleted, the set s3 is: 0 3.
```

Requirements

Header: <set>

See Also

Reference

set Class

Standard Template Library

Other Resources

set Members

set::find

Returns an iterator addressing the location of an element in a set that has a key equivalent to a specified key.

```
iterator find(
   const Key& _Key
);
const_iterator find(
   const Key& _Key
) const;
```

Parameters

_Key

The argument key to be matched by the sort key of an element from the set being searched.

Return Value

An iterator or **const_iterator** that addresses the location of an element equivalent to a specified key or that addresses the location succeeding the last element in the set if no match is found for the key.

Remarks

The member function returns an iterator that addresses an element in the set whose sort key is equivalent to the argument key under a binary predicate that induces an ordering based on a less-than comparability relation.

If the return value of **find** is assigned to a **const_iterator**, the set object cannot be modified. If the return value of **find** is assigned to an **iterator**, the set object can be modified.

Example

```
// set_find.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int> :: const_iterator s1_AcIter, s1_RcIter;

   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );

   s1_RcIter = s1.find( 20 );
   cout << "The element of set s1 with a key of 20 is: "
        << *s1_RcIter << "." << endl;

   s1_RcIter = s1.find( 40 );

   // If no match is found for the key, end( ) is returned
   if ( s1_RcIter == s1.end( ) )
      cout << "The set s1 doesn't have an element "
           << "with a key of 40." << endl;
   else
      cout << "The element of set s1 with a key of 40 is: "
           << *s1_RcIter << "." << endl;

   // The element at a specific location in the set can be found
   // by using a dereferenced iterator addressing the location
   s1_AcIter = s1.end( );
   s1_AcIter--;
   s1_RcIter = s1.find( *s1_AcIter );
```

```
    cout << "The element of s1 with a key matching "
         << "that of the last element is: "
         << *s1_RcIter << "." << endl;
}
```

Output

```
The element of set s1 with a key of 20 is: 20.
The set s1 doesn't have an element with a key of 40.
The element of s1 with a key matching that of the last element is: 30.
```

Requirements

Header: <set>

See Also

Reference

set Class

set::find (STL Samples)

Standard Template Library

Other Resources

set Members

set::get_allocator

Returns a copy of the allocator object used to construct the set.

```
Allocator get_allocator( ) const;
```

Return Value

The allocator used by the set to manage memory, which is the template parameter **Allocator**.

For more information on **Allocator**, see the Remarks section of the set Class topic.

Remarks

Allocators for the set class specify how the class manages storage. The default allocators supplied with STL container classes is sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```
// set_get_allocator.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int>::allocator_type s1_Alloc;
   set <int>::allocator_type s2_Alloc;
   set <double>::allocator_type s3_Alloc;
   set <int>::allocator_type s4_Alloc;

   // The following lines declare objects
   // that use the default allocator.
   set <int> s1;
   set <int, allocator<int> > s2;
   set <double, allocator<double> > s3;

   s1_Alloc = s1.get_allocator( );
   s2_Alloc = s2.get_allocator( );
   s3_Alloc = s3.get_allocator( );

   cout << "The number of integers that can be allocated"
        << endl << "before free memory is exhausted: "
        << s2.max_size( ) << "." << endl;

   cout << "\nThe number of doubles that can be allocated"
        << endl << "before free memory is exhausted: "
        << s3.max_size( ) <<  "." << endl;

   // The following line creates a set s4
   // with the allocator of multiset s1.
   set <int> s4( less<int>( ), s1_Alloc );

   s4_Alloc = s4.get_allocator( );

   // Two allocators are interchangeable if
   // storage allocated from each can be
   // deallocated by the other
   if( s1_Alloc == s4_Alloc )
   {
      cout << "\nThe allocators are interchangeable."
           << endl;
   }
   else
```

```
    {
       cout << "\nThe allocators are not interchangeable."
            << endl;
    }
}
```

Sample Output

The following output is for x86.

```
The number of integers that can be allocated
before free memory is exhausted: 1073741823.

The number of doubles that can be allocated
before free memory is exhausted: 536870911.

The allocators are interchangeable.
```

Requirements

Header: <set>

See Also

Reference

set Class

Standard Template Library

Other Resources

set Members

set::insert

Inserts an element or a range of elements into a set.

```
pair <iterator, bool> insert(
   const value_type& _Val
);
iterator insert(
   iterator _Where,
   const value_type& _Val
);
template<class InputIterator>
   void insert(
      InputIterator _First,
      InputIterator _Last
   );
```

Parameters

_Val

The value of an element to be inserted into the set unless the set already contains that element or, more generally, an element whose key is equivalently ordered.

_Where

The place to start searching for the correct point of insertion. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows *_Where*.)

_First

The position of the first element to be copied from a set.

_Last

The position just beyond the last element to be copied from a set.

Return Value

The first **insert** member function returns a pair whose bool component returns **true** if an insertion was make and **false** if the set already contained an element whose key had an equivalent value in the ordering, and whose iterator component returns the address where a new element was inserted or where the element was already located.

To access the iterator component of a pair *pr* returned by this member function, use *pr*.**first**, and to dereference it, use * (*pr*.**first**). To access the **bool** component of a pair *pr* returned by this member function, use *pr*.**second**.

The second **insert** member function returns an iterator that points to the position where the new element was inserted into the set.

Remarks

The third member function inserts the sequence of element values into a set corresponding to each element addressed by an iterator of in the range *[_First, _Last)* of a specified set.

Example

```
// set_insert.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int>::iterator s1_pIter, s2_pIter;

   set <int, less<int> > s1, s2;
   s1.insert( 10 );
```

```
    s1.insert( 20 );
    s1.insert( 30 );
    s1.insert( 40 );

    cout << "The original s1 =";
    for ( s1_pIter = s1.begin( ); s1_pIter != s1.end( ); s1_pIter++ )
       cout << " " << *s1_pIter;
    cout << "." << endl;

    pair< set<int>::iterator, bool > pr;
    pr = s1.insert( 10 );

    if(pr.second == true)
    {
       cout << "The element 10 was inserted in s1 successfully."
            << endl;
    }
    else
    {
       cout << "The element 10 already exists in s1 and"
            << " *( pr.first ) = " << *( pr.first ) << "." << endl;
    }

    s1.insert( --s1.end( ), 50 );

    cout << "After the insertions, s1 =";
    for ( s1_pIter = s1.begin( ); s1_pIter != s1.end( ); s1_pIter++ )
       cout << " " << *s1_pIter;
    cout << "." << endl;

    s2.insert( 100 );
    s2.insert( ++s1.begin( ), --s1.end( ) );

    cout << "s2 =";
    for ( s2_pIter = s2.begin( ); s2_pIter != s2.end( ); s2_pIter++ )
       cout << " " << *s2_pIter;
    cout << "." << endl;
 }
```

Output

```
The original s1 = 10 20 30 40.
The element 10 already exists in s1 and *( pr.first ) = 10.
After the insertions, s1 = 10 20 30 40 50.
s2 = 20 30 40 100.
```

Requirements

Header: <set>

See Also

Reference

set Class

Standard Template Library

Other Resources

set Members

set::key_comp

Retrieves a copy of the comparison object used to order keys in a set.

```
key_compare key_comp( ) const;
```

Return Value

Returns the function object that a set uses to order its elements, which is the template parameter **Traits**.

For more information on **Traits** see the set Class topic.

Remarks

The stored object defines the member function:

bool operator()(**const Key&** *_xVal*, **const Key&** *_yVal*);

which returns **true** if *_xVal* precedes and is not equal to *_yVal* in the sort order.

Note that both key_compare and value_compare are synonyms for the template parameter **Traits**. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

```
// set_key_comp.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;

   set <int, less<int> > s1;
   set<int, less<int> >::key_compare kc1 = s1.key_comp( ) ;
   bool result1 = kc1( 2, 3 ) ;
   if( result1 == true )
   {
      cout << "kc1( 2,3 ) returns value of true, "
           << "where kc1 is the function object of s1."
           << endl;
   }
   else
   {
      cout << "kc1( 2,3 ) returns value of false "
           << "where kc1 is the function object of s1."
           << endl;
   }

   set <int, greater<int> > s2;
   set<int, greater<int> >::key_compare kc2 = s2.key_comp( ) ;
   bool result2 = kc2( 2, 3 ) ;
   if(result2 == true)
   {
      cout << "kc2( 2,3 ) returns value of true, "
           << "where kc2 is the function object of s2."
           << endl;
   }
   else
   {
      cout << "kc2( 2,3 ) returns value of false, "
           << "where kc2 is the function object of s2."
           << endl;
   }
```

```
}
```

Output

```
kc1( 2,3 ) returns value of true, where kc1 is the function object of s1.
kc2( 2,3 ) returns value of false, where kc2 is the function object of s2.
```

Requirements

Header: <set>

See Also

Reference

set Class

set::key_comp and set::value_comp

Standard Template Library

Other Resources

set Members

set::lower_bound

Returns an iterator to the first element in a set with a key that is equal to or greater than a specified key.

```
const_iterator lower_bound(
   const Key& _Key
) const;
iterator lower_bound(
   const Key& _Key
);
```

Parameters

_Key

The argument key to be compared with the sort key of an element from the set being searched.

Return Value

An iterator or **const_iterator** that addresses the location of an element in a set that with a key that is equal to or greater than the argument key or that addresses the location succeeding the last element in the set if no match is found for the key.

Example

```
// set_lower_bound.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int> :: const_iterator s1_AcIter, s1_RcIter;

   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );

   s1_RcIter = s1.lower_bound( 20 );
   cout << "The element of set s1 with a key of 20 is: "
        << *s1_RcIter << "." << endl;

   s1_RcIter = s1.lower_bound( 40 );

   // If no match is found for the key, end( ) is returned
   if ( s1_RcIter == s1.end( ) )
      cout << "The set s1 doesn't have an element "
           << "with a key of 40." << endl;
   else
      cout << "The element of set s1 with a key of 40 is: "
           << *s1_RcIter << "." << endl;

   // The element at a specific location in the set can be found
   // by using a dereferenced iterator that addresses the location
   s1_AcIter = s1.end( );
   s1_AcIter--;
   s1_RcIter = s1.lower_bound( *s1_AcIter );
   cout << "The element of s1 with a key matching "
        << "that of the last element is: "
        << *s1_RcIter << "." << endl;
}
```

Output

```
The element of set s1 with a key of 20 is: 20.
```

```
The set s1 doesn't have an element with a key of 40.
The element of s1 with a key matching that of the last element is: 30.
```

Requirements

Header: <set>

See Also

Reference

set Class

set::lower_bound, set::upper_bound, and set::equal_range

Standard Template Library

Other Resources

set Members

set::max_size

Returns the maximum length of the set.

```
size_type max_size( ) const;
```

Return Value

The maximum possible length of the set.

Example

```
// set_max_size.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int>::size_type i;

   i = s1.max_size( );
   cout << "The maximum possible length "
        << "of the set is " << i << "." << endl;
}
```

Sample Output

The following output is for x86.

```
The maximum possible length of the set is 1073741823.
```

Requirements

Header: <set>

See Also

Reference

set Class

set::max_size (STL Samples)

Standard Template Library

Other Resources

set Members

set::rbegin

Returns an iterator addressing the first element in a reversed set.

```
const_reverse_iterator rbegin( ) const;
reverse_iterator rbegin( );
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed set or addressing what had been the last element in the unreversed set.

Remarks

rbegin is used with a reversed set just as begin is used with a set.

If the return value of **rbegin** is assigned to a **const_reverse_iterator**, then the set object cannot be modified. If the return value of **rbegin** is assigned to a **reverse_iterator**, then the set object can be modified.

rbegin can be used to iterate through a set backwards.

Example

```
// set_rbegin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int>::iterator s1_Iter;
   set <int>::reverse_iterator s1_rIter;

   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );

   s1_rIter = s1.rbegin( );
   cout << "The first element in the reversed set is "
        << *s1_rIter << "." << endl;

   // begin can be used to start an iteration
   // throught a set in a forward order
   cout << "The set is:";
   for ( s1_Iter = s1.begin( ) ; s1_Iter != s1.end( ); s1_Iter++ )
      cout << " " << *s1_Iter;
   cout << endl;

   // rbegin can be used to start an iteration
   // throught a set in a reverse order
   cout << "The reversed set is:";
   for ( s1_rIter = s1.rbegin( ) ; s1_rIter != s1.rend( ); s1_rIter++ )
      cout << " " << *s1_rIter;
   cout << endl;

   // A set element can be erased by dereferencing to its key
   s1_rIter = s1.rbegin( );
   s1.erase ( *s1_rIter );

   s1_rIter = s1.rbegin( );
   cout << "After the erasure, the first element "
        << "in the reversed set is "<< *s1_rIter << "." << endl;
}
```

Output

```
The first element in the reversed set is 30.
The set is: 10 20 30
The reversed set is: 30 20 10
After the erasure, the first element in the reversed set is 20.
```

Requirements

Header: <set>

See Also

Reference

set Class

set::rbegin and set::rend

Standard Template Library

Other Resources

set Members

set::rend

Returns an iterator that addresses the location succeeding the last element in a reversed set.

```
const_reverse_iterator rend( ) const;
reverse_iterator rend( );
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed set (the location that had preceded the first element in the unreversed set).

Remarks

rend is used with a reversed set just as end is used with a set.

If the return value of **rend** is assigned to a **const_reverse_iterator**, then the set object cannot be modified. If the return value of **rend** is assigned to a **reverse_iterator**, then the set object can be modified. The value returned by **rend** should not be dereferenced.

rend can be used to test to whether a reverse iterator has reached the end of its set.

Example

```
// set_rend.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main() {
   using namespace std;
   set <int> s1;
   set <int>::iterator s1_Iter;
   set <int>::reverse_iterator s1_rIter;
   set <int>::const_reverse_iterator s1_crIter;

   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );

   s1_rIter = s1.rend( );
   s1_rIter--;
   cout << "The last element in the reversed set is "
        << *s1_rIter << "." << endl;

   // end can be used to terminate an iteration
   // throught a set in a forward order
   cout << "The set is: ";
   for ( s1_Iter = s1.begin( ) ; s1_Iter != s1.end( ); s1_Iter++ )
      cout << *s1_Iter << " ";
   cout << "." << endl;

   // rend can be used to terminate an iteration
   // throught a set in a reverse order
   cout << "The reversed set is: ";
   for ( s1_rIter = s1.rbegin( ) ; s1_rIter != s1.rend( ); s1_rIter++ )
      cout << *s1_rIter << " ";
   cout << "." << endl;

   s1_rIter = s1.rend( );
   s1_rIter--;
   s1.erase ( *s1_rIter );

   s1_rIter = s1.rend( );
   --s1_rIter;
```

```
    cout << "After the erasure, the last element in the "
         << "reversed set is " << *s1_rIter << "." << endl;
}
```

Output

```
The last element in the reversed set is 10.
The set is: 10 20 30 .
The reversed set is: 30 20 10 .
After the erasure, the last element in the reversed set is 20.
```

Requirements

Header: <set>

See Also

Reference

set Class

set::rbegin and set::rend

Standard Template Library

Other Resources

set Members

set::set

Constructs a set that is empty or that is a copy of all or part of some other set.

```
set( );
explicit set(
   const Traits& _Comp
);
explicit set(
   const Traits& _Comp,
   const Allocator& _Al
);
set(
   const _set& _Right
);
template<class InputIterator>
   set(
      InputIterator _First,
      InputIterator _Last
   );
template<class InputIterator>
   set(
      InputIterator _First,
      InputIterator _Last,
      const Traits& _Comp
   );
template<class InputIterator>
   set(
      InputIterator _First,
      InputIterator _Last,
      const Traits& _Comp,
      const Allocator& _Al
   );
```

Parameters

_Al

The storage allocator class to be used for this set object, which defaults to **Allocator**.

_Comp

The comparison function of type **const Traits** used to order the elements in the set, which defaults to **Compare**.

_Right

The set of which the constructed set is to be a copy.

_First

The position of the first element in the range of elements to be copied.

_Last

The position of the first element beyond the range of elements to be copied.

Remarks

All constructors store a type of allocator object that manages memory storage for the set and that can later be returned by calling get_allocator. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their sets.

All constructors store a function object of type **Traits** that is used to establish an order among the keys of the set and that can later be returned by calling key_comp.

The first three constructors specify an empty initial set, the second specifying the type of comparison function (*_Comp*) to be

used in establishing the order of the elements and the third explicitly specifying the allocator type (*_Al*) to be used. The keyword **explicit** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the set *_Right*.

The last three constructors copy the range [*_First, _Last*) of a set with increasing explicitness in specifying the type of comparison function of class **Traits** and **Allocator**.

Example

```
// set_set.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int>::iterator s1_Iter, s2_Iter, s3_Iter, s4_Iter, s5_Iter, s6_Iter;

   // Create an empty set s0 of key type integer
   set <int> s0;

   // Create an empty set s1 with the key comparison
   // function of less than, then insert 4 elements
   set <int, less<int> > s1;
   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );
   s1.insert( 40 );

   // Create an empty set s2 with the key comparison
   // function of geater than, then insert 2 elements
   set <int, greater<int> > s2;
   s2.insert(10);
   s2.insert(20);

   // Create a set s3 with the
   // allocator of set s1
   set <int>::allocator_type s1_Alloc;
   s1_Alloc = s1.get_allocator( );
   set <int> s3( less<int>( ), s1_Alloc );
   s3.insert( 30 );

   // Create a copy, set s4, of set s1
   set <int> s4( s1 );

   // Create a set s5 by copying the range s1[_First, _Last)
   set <int>::const_iterator s1_bcIter, s1_ecIter;
   s1_bcIter = s1.begin( );
   s1_ecIter = s1.begin( );
   s1_ecIter++;
   s1_ecIter++;
   set <int> s5( s1_bcIter, s1_ecIter );

   // Create a set s6 by copying the range s4[_First, _Last)
   // and with the allocator of set s2
   set <int>::allocator_type s2_Alloc;
   s2_Alloc = s2.get_allocator( );
   set <int> s6( s4.begin( ), ++s4.begin( ), less<int>( ), s2_Alloc );

   cout << "s1 =";
   for ( s1_Iter = s1.begin( ); s1_Iter != s1.end( ); s1_Iter++ )
      cout << " " << *s1_Iter;
   cout << endl;

   cout << "s2 = " << *s2.begin( ) << " " << *++s2.begin( ) << endl;

   cout << "s3 =";
```

```
    for ( s3_Iter = s3.begin( ); s3_Iter != s3.end( ); s3_Iter++ )
       cout << " " << *s3_Iter;
    cout << endl;

    cout << "s4 =";
    for ( s4_Iter = s4.begin( ); s4_Iter != s4.end( ); s4_Iter++ )
       cout << " " << *s4_Iter;
    cout << endl;

    cout << "s5 =";
    for ( s5_Iter = s5.begin( ); s5_Iter != s5.end( ); s5_Iter++ )
       cout << " " << *s5_Iter;
    cout << endl;

    cout << "s6 =";
    for ( s6_Iter = s6.begin( ); s6_Iter != s6.end( ); s6_Iter++ )
       cout << " " << *s6_Iter;
    cout << endl;
 }
```

Output

```
s1 = 10 20 30 40
s2 = 20 10
s3 = 30
s4 = 10 20 30 40
s5 = 10 20
s6 = 10
```

Requirements

Header: <set>

See Also

Reference

set Class

Standard Template Library

Other Resources

set Members

set::size

Returns the number of elements in the set.

```
size_type size( ) const;
```

Return Value

The current length of the set.

Example

```
// set_size.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int> :: size_type i;

   s1.insert( 1 );
   i = s1.size( );
   cout << "The set length is " << i << "." << endl;

   s1.insert( 2 );
   i = s1.size( );
   cout << "The set length is now " << i << "." << endl;
}
```

Output

```
The set length is 1.
The set length is now 2.
```

Requirements

Header: <set>

See Also

Reference

set Class

set::size (STL Samples)

Standard Template Library

Other Resources

set Members

set::swap

Exchanges the elements of two sets.

```
void swap(
   set& _Right
);
```

Parameters

_Right

The argument set providing the elements to be swapped with the target set.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two sets whose elements are being exchanged.

Example

```
// set_swap.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1, s2, s3;
   set <int>::iterator s1_Iter;

   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );
   s2.insert( 100 );
   s2.insert( 200 );
   s3.insert( 300 );

   cout << "The original set s1 is:";
   for ( s1_Iter = s1.begin( ); s1_Iter != s1.end( ); s1_Iter++ )
      cout << " " << *s1_Iter;
   cout   << "." << endl;

   // This is the member function version of swap
   s1.swap( s2 );

   cout << "After swapping with s2, list s1 is:";
   for ( s1_Iter = s1.begin( ); s1_Iter != s1.end( ); s1_Iter++ )
      cout << " " << *s1_Iter;
   cout  << "." << endl;

   // This is the specialized template version of swap
   swap( s1, s3 );

   cout << "After swapping with s3, list s1 is:";
   for ( s1_Iter = s1.begin( ); s1_Iter != s1.end( ); s1_Iter++ )
      cout << " " << *s1_Iter;
   cout   << "." << endl;
}
```

Output

```
The original set s1 is: 10 20 30.
After swapping with s2, list s1 is: 100 200.
After swapping with s3, list s1 is: 300.
```

Requirements

Header: <set>

See Also

Reference

set Class

set::swap, set::begin, and set::end

Standard Template Library

Other Resources

set Members

set::upper_bound

Returns an iterator to the first element in a set that with a key that is greater than a specified key.

```
const_iterator upper_bound(
   const Key& _Key
) const;

iterator upper_bound(
   const Key& _Key
);
```

Parameters

_Key

The argument key to be compared with the sort key of an element from the set being searched.

Return Value

An **iterator** or **const_iterator** that addresses the location of an element in a set that with a key that is greater than the argument key, or that addresses the location succeeding the last element in the set if no match is found for the key.

Example

```
// set_upper_bound.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int> :: const_iterator s1_AcIter, s1_RcIter;

   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );

   s1_RcIter = s1.upper_bound( 20 );
   cout << "The first element of set s1 with a key greater "
        << "than 20 is: " << *s1_RcIter << "." << endl;

   s1_RcIter = s1.upper_bound( 30 );

   // If no match is found for the key, end( ) is returned
   if ( s1_RcIter == s1.end( ) )
      cout << "The set s1 doesn't have an element "
           << "with a key greater than 30." << endl;
   else
      cout << "The element of set s1 with a key > 40 is: "
           << *s1_RcIter << "." << endl;

   // The element at a specific location in the set can be found
   // by using a dereferenced iterator addressing the location
   s1_AcIter = s1.begin( );
   s1_RcIter = s1.upper_bound( *s1_AcIter );
   cout << "The first element of s1 with a key greater than"
        << endl << "that of the initial element of s1 is: "
        << *s1_RcIter << "." << endl;
}
```

Output

```
The first element of set s1 with a key greater than 20 is: 30.
```

```
The set s1 doesn't have an element with a key greater than 30.
The first element of s1 with a key greater than
that of the initial element of s1 is: 20.
```

Requirements

Header: <set>

See Also

Reference

set Class

set::lower_bound, set::upper_bound, and set::equal_range

Standard Template Library

Other Resources

set Members

set::value_comp

Retrieves a copy of the comparison object used to order element values in a set.

```
value_compare value_comp( ) const;
```

Return Value

Returns the function object that a set uses to order its elements, which is the template parameter **Traits**.

For more information on **Traits** see the set Class topic.

Remarks

The stored object defines the member function:

bool operator(**const Key&** *_xVal*, **const Key&** *_yVal*);

which returns **true** if *_xVal* precedes and is not equal to *_yVal* in the sort order.

Note that both value_compare and key_compare are synonyms for the template parameter **Traits**. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

```
// set_value_comp.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;

   set <int, less<int> > s1;
   set <int, less<int> >::value_compare vc1 = s1.value_comp( );
   bool result1 = vc1( 2, 3 );
   if( result1 == true )
   {
      cout << "vc1( 2,3 ) returns value of true, "
           << "where vc1 is the function object of s1."
           << endl;
   }
   else
   {
      cout << "vc1( 2,3 ) returns value of false, "
           << "where vc1 is the function object of s1."
           << endl;
   }

   set <int, greater<int> > s2;
   set<int, greater<int> >::value_compare vc2 = s2.value_comp( );
   bool result2 = vc2( 2, 3 );
   if( result2 == true )
   {
      cout << "vc2( 2,3 ) returns value of true, "
           << "where vc2 is the function object of s2."
           << endl;
   }
   else
   {
      cout << "vc2( 2,3 ) returns value of false, "
           << "where vc2 is the function object of s2."
           << endl;
   }
```

```
}
```

Output

```
vc1( 2,3 ) returns value of true, where vc1 is the function object of s1.
vc2( 2,3 ) returns value of false, where vc2 is the function object of s2.
```

Requirements

Header: <set>

See Also

Reference

set Class

set::key_comp and set::value_comp

Standard Template Library

Other Resources

set Members

multiset Class

The Standard Template Library multiset class is used for the storage and retrieval of data from a collection in which the values of the elements contained need not be unique and in which they serve as the key values according to which the data is automatically ordered. The key value of an element in a multiset may not be changed directly. Instead, old values must be deleted and elements with new values inserted.

For a list of all members of this type, see multiset Members.

```
template <
   class Key,
   class Compare=less<Key>,
   class Allocator=allocator<Key>
>
class multiset
```

Parameters

Key

The element data type to be stored in the multiset.

Compare

The type that provides a function object that can compare two element values as sort keys to determine their relative order in the multiset. The binary predicate **less**<Key> is the default value.

Allocator

The type that represents the stored allocator object that encapsulates details about the multiset's allocation and deallocation of memory. The default value is **allocator**<*Key*>.

Remarks

The STL multiset class is:

- An associative container, which is a variable size container that supports the efficient retrieval of element values based on an associated key value.
- Reversible, because it provides bidirectional iterators to access its elements.
- Sorted, because its elements are ordered by key values within the container in accordance with a specified comparison function.
- Multiple in the sense that its elements do not need to have unique keys, so that one key value can have many element values associated with it.
- A simple associative container because its element values are its key values.
- A template class, because the functionality it provides is generic and so independent of the specific type of data contained as elements. The data type to be used is, instead, specified as a parameter in the class template along with the comparison function and allocator.

The iterator provided by the multiset class is a bidirectional iterator, but the class member functions insert and multiset have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own set of requirements and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal set of functionality, but it is enough to be able to talk meaningfully about a range of iterators [*_First*, *_Last*) in the context of the class's member functions.

The choice of container type should be based in general on the type of searching and inserting required by the application. Associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient, performing them in a time that is on average proportional to the logarithm of the number of elements in the container. Inserting elements invalidates no iterators, and removing elements invalidates only those

iterators that had specifically pointed at the removed elements.

The multiset should be the associative container of choice when the conditions associating the values with their keys are satisfies by the application. The elements of a multiset may be multiple and serve as their own sort keys, so keys are not unique. A model for this type of structure is an ordered list of, say, words in which the words may occur more than once. Had multiple occurrences of the words not been allowed, then a set would have been the appropriate container structure. If unique definitions were attached as values to the list of unique key words, then a map would be an appropriate structure to contain this data. If instead the definitions were not unique, then a multimap would be the container of choice.

The multiset orders the sequence it controls by calling a stored function object of type **Compare**. This stored object is a comparison function that may be accessed by calling the member function key_comp. In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate *f*(*x*,*y*) is a function object that has two argument objects *x* and *y* and a return value of **true** or **false**. An ordering imposed on a set is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects x and y are defined to be equivalent when both *f*(*x*,*y*) and *f*(*y*,*x*) are false. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

Requirements

Header: <set>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

multiset Members

<set> Members

multiset Members

Reference

Constructors

multiset	Constructs a **multiset** that is empty or that is a copy of all or part of some other **multiset**.

Typedefs

allocator_type	A type that represents the **allocator** class for the **multiset** object.
const_iterator	A type that provides a bidirectional iterator that can read a **const** element in the **multiset**.
const_pointer	A type that provides a pointer to a **const** element in a **multiset**.
const_reference	A type that provides a reference to a **const** element stored in a **multiset** for reading and performing **const** operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any **const** element in the **multiset**.
difference_type	A signed integer type that can be used to represent the number of elements of a **multiset** in a range between elements pointed to by iterators.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a **multiset**.
key_compare	A type that provides a function object that can compare two keys to determine the relative order of two elements in the **multiset**.
key_type	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the **multiset**.
pointer	A type that provides a pointer to an element in a **multiset**.
reference	A type that provides a reference to an element stored in a **multiset**.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed **multiset**.
size_type	An unsigned integer type that can represent the number of elements in a **multiset**.
value_compare	The type that provides a function object that can compare two elements as sort keys to determine their relative order in the **multiset**.
value_type	A type that describes an object stored as an element as a **multiset** in its capacity as a value.

Member Functions

begin	Returns an iterator addressing the first element in the **multiset**.
clear	Erases all the elements of a **multiset**.
count	Returns the number of elements in a **multiset** whose key matches a parameter-specified key.
empty	Tests if a **multiset** is empty.
end	Returns an iterator that addresses the location succeeding the last element in a **multiset**.

equal_range	Returns a pair of iterators respectively to the first element in a **multiset** with a key that is greater than a specifie d key and to the first element in the **multiset** with a key that is equal to or greater than the key.
erase	Removes an element or a range of elements in a **multiset** from specified positions or removes elements that m atch a specified key.
find	Returns an iterator addressing the first location of an element in a **multiset** that has a key equivalent to a specifi ed key.
get_allocator	Returns a copy of the **allocator** object used to construct the **multiset**.
insert	Inserts an element or a range of elements into a **multiset**.
key_comp	A type that provides a function object that can compare two sort keys to determine the relative order of two ele ments in the **multiset**.
lower_bound	Returns an iterator to the first element in a **multiset** with a key that is equal to or greater than a specified key.
max_size	Returns the maximum length of the **multiset**.
rbegin	Returns an iterator addressing the first element in a reversed **multiset**.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed **multiset**.
size	A type that counts the number of elements in a **multiset**.
swap	Exchanges the elements of two **multiset**s.
upper_bound	Returns an iterator to the first element in a **multiset** that with a key that is equal to or greater than a specified k ey.
value_comp	Retrieves a copy of the comparison object used to order element values in a **multiset**.

See Also

Reference

multiset Class
Thread Safety in the Standard C++ Library
Standard Template Library

multiset Typedefs

For information about the typedefs in the multiset class, see multiset Members.

See Also

Reference

Standard Template Library

multiset::allocator_type

A type that represents the allocator class for the multiset object

```
typedef Allocator allocator_type
```

Remarks

allocator_type is a synonym for the template parameter **Allocator**.

For more information on **Allocator**, see the Remarks section of the multiset Class topic.

Example

See the example for get_allocator for an example using **allocator_type**

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::const_iterator

A type that provides a bidirectional iterator that can read a **const** element in the multiset.

```
typedef implementation-defined const_iterator;
```

Remarks

A type **const_iterator** cannot be used to modify the value of an element.

Example

See the example for begin for an example using **const_iterator**.

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::const_pointer

A type that provides a pointer to a **const** element in a multiset.

```
typedef typename Allocator::const_pointer const_pointer;
```

Remarks

A type **const_pointer** cannot be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a multiset object.

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::const_reference

A type that provides a reference to a **const** element stored in a multiset for reading and performing **const** operations.

```
typedef typename Allocator::const_reference const_reference;
```

Example

```
// multiset_const_ref.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;

   ms1.insert( 10 );
   ms1.insert( 20 );

   // Declare and initialize a const_reference &Ref1
   // to the 1st element
   const int &Ref1 = *ms1.begin( );

   cout << "The first element in the multiset is "
        << Ref1 << "." << endl;

   // The following line would cause an error because the
   // const_reference cannot be used to modify the multiset
   // Ref1 = Ref1 + 5;
}
```

Output

```
The first element in the multiset is 10.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::const_reverse_iterator

A type that provides a bidirectional iterator that can read any **const** element in the multiset.

```
typedef reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type **const_reverse_iterator** cannot modify the value of an element and is use to iterate through the multiset in reverse.

Example

See the example for rend for an example of how to declare and use the **const_reverse_iterator**.

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::difference_type

A signed integer type that can be used to represent the number of elements of a multiset in a range between elements pointed to by iterators.

```
typedef implementation-defined difference_type;
```

Remarks

The **difference_type** is the type returned when subtracting or incrementing through iterators of the container. The **difference_type** is typically used to represent the number of elements in the range [_*First*, _*Last*) between the iterators _*First* and _*Last*, includes the element pointed to by _*First* and the range of elements up to, but not including, the element pointed to by _*Last*.

Note that although **difference_type** is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers like set, subtraction between iterators is only supported by random-access iterators provided by a random-access container like vector.

Example

```
// multiset_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <set>
#include <algorithm>

int main( )
{
   using namespace std;

   multiset <int> ms1;
   multiset <int>::iterator ms1_Iter, ms1_bIter, ms1_eIter;

   ms1.insert( 20 );
   ms1.insert( 10 );
   ms1.insert( 20 );

   ms1_bIter = ms1.begin( );
   ms1_eIter = ms1.end( );

   multiset <int>::difference_type   df_typ5, df_typ10, df_typ20;

   df_typ5 = count( ms1_bIter, ms1_eIter, 5 );
   df_typ10 = count( ms1_bIter, ms1_eIter, 10 );
   df_typ20 = count( ms1_bIter, ms1_eIter, 20 );

   // The keys, and hence the elements, of a multiset are not unique
   cout << "The number '5' occurs " << df_typ5
        << " times in multiset ms1.\n";
   cout << "The number '10' occurs " << df_typ10
        << " times in multiset ms1.\n";
   cout << "The number '20' occurs " << df_typ20
        << " times in multiset ms1.\n";

   // Count the number of elements in a multiset
   multiset <int>::difference_type  df_count = 0;
   ms1_Iter = ms1.begin( );
   while ( ms1_Iter != ms1_eIter)
   {
      df_count++;
      ms1_Iter++;
   }

   cout << "The number of elements in the multiset ms1 is: "
```

```
        << df_count << "." << endl;
}
```

Output

```
The number '5' occurs 0 times in multiset ms1.
The number '10' occurs 1 times in multiset ms1.
The number '20' occurs 2 times in multiset ms1.
The number of elements in the multiset ms1 is: 3.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::iterator

A type that provides a bidirectional iterator that can read or modify any element in a multiset.

```
typedef implementation-defined iterator;
```

Remarks

A type **iterator** can be used to modify the value of an element.

Example

See example for begin for an example of how to declare and use the **iterator**.

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::key_compare

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the multiset.

```
typedef Compare key_compare;
```

Remarks

key_compare is a synonym for the template parameter **Compare**.

For more information on **Compare**, see the Remarks section of the multiset Class topic.

Example

See the example for key_comp for an example of how to declare and use **key_compare**.

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::key_type

A type that provides a function object that can compare sort keys to determine the relative order of two elements in the multiset.

```
typedef Key key_type;
```

Remarks

key_type is a synonym for the template parameter **Key**.

For more information on **Key**, see the Remarks section of the multiset Class topic.

Example

See the example for value_type for an example of how to declare and use **key_type**.

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::pointer

A type that provides a pointer to an element in a multiset.

```
typedef typename Allocator::pointer pointer;
```

Remarks

A type **pointer** can be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a multiset object.

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::reference

A type that provides a reference to an element stored in a multiset.

```
typedef typename Allocator::reference reference;
```

Example

```
// multiset_ref.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;

   ms1.insert( 10 );
   ms1.insert( 20 );

   // Declare and initialize a reference &Ref1 to the 1st element
   int &Ref1 = *ms1.begin( );

   cout << "The first element in the multiset is "
        << Ref1 << "." << endl;

   // The value of the 1st element of the multiset can be changed
   // by operating on its (non const) reference
   Ref1 = Ref1 + 5;

   cout << "The first element in the multiset is now "
        << *ms1.begin( ) << "." << endl;
}
```

Output

```
The first element in the multiset is 10.
The first element in the multiset is now 15.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::reverse_iterator

A type that provides a bidirectional iterator that can read or modify an element in a reversed multiset.

```
typedef reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type **reverse_iterator** is use to iterate through the multiset in reverse.

Example

See example for rbegin for an example of how to declare and use **reverse_iterator**.

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::size_type

An unsigned integer type that can represent the number of elements in a multiset.

```
typedef implementation defined size_type;
```

Example

See example for size for an example of how to declare and use **size_type**

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::value_compare

The type that provides a function object that can compare two sort keys to determine their relative order in the multiset.

```
typedef Compare value_compare;
```

Remarks

value_compare is a synonym for the template parameter **Compare**.

Note that both key_compare and **value_compare** are synonyms for the template parameter **Compare**. Both types are provided for the classes set and multiset, where they are identical, for compatibility with the classes map and multimap, where they are distinct.

For more information on **Compare**, see the Remarks section of the multiset Class topic.

Example

See the example for value_comp for an example of how to declare and use **value_compare**.

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::value_type

A type that describes an object stored as an element as a multiset in its capacity as a value.

```
typedef Key value_type;
```

Remarks

value_type is a synonym for the template parameter **Key**.

Note that both key_type and **value_type** are synonyms for the template parameter **Key**. Both types are provided for the classes set and multiset, where they are identical, for compatibility with the classes map and multimap, where they are distinct.

For more information on **Key**, see the Remarks section of the topic.

Example

```
// multiset_value_type.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int>::iterator ms1_Iter;

   multiset <int> :: value_type svt_Int;   // Declare value_type
   svt_Int = 10;             // Initialize value_type

   multiset <int> :: key_type skt_Int;   // Declare key_type
   skt_Int = 20;             // Initialize key_type

   ms1.insert( svt_Int );         // Insert value into s1
   ms1.insert( skt_Int );         // Insert key into s1

   // A multiset accepts key_types or value_types as elements
   cout << "The multiset has elements:";
   for ( ms1_Iter = ms1.begin( ) ; ms1_Iter != ms1.end( ); ms1_Iter++ )
      cout << " " << *ms1_Iter;
   cout << "." << endl;
}
```

Output

```
The multiset has elements: 10 20.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset Member Functions

For information about the functions in the multiset class, see multiset Members.

See Also

Reference

Standard Template Library

multiset::begin

Returns an iterator addressing the first element in the multiset.

```
const_iterator begin( ) const;
iterator begin( );
```

Return Value

A bidirectional iterator addressing the first element in the multiset or the location succeeding an empty multiset.

Remark

If the return value of **begin** is assigned to a **const_iterator**, the elements in the multiset object cannot be modified. If the return value of **begin** is assigned to an **iterator**, the elements in the multiset object can be modified.

Example

```
// multiset_begin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int>::iterator ms1_Iter;
   multiset <int>::const_iterator ms1_cIter;

   ms1.insert( 1 );
   ms1.insert( 2 );
   ms1.insert( 3 );

   ms1_Iter = ms1.begin( );
   cout << "The first element of ms1 is " << *ms1_Iter << endl;

   ms1_Iter = ms1.begin( );
   ms1.erase( ms1_Iter );

   // The following 2 lines would err as the iterator is const
   // ms1_cIter = ms1.begin( );
   // ms1.erase( ms1_cIter );

   ms1_cIter = ms1.begin( );
   cout << "The first element of ms1 is now " << *ms1_cIter << endl;
}
```

Output

```
The first element of ms1 is 1
The first element of ms1 is now 2
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::clear

Erases all the elements of a multiset.

```
void clear( );
```

Example

```
// multiset_clear.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;

   ms1.insert( 1 );
   ms1.insert( 2 );

   cout << "The size of the multiset is initially "
        << ms1.size( ) << "." << endl;

   ms1.clear( );
   cout << "The size of the multiset after clearing is "
        << ms1.size( ) << "." << endl;
}
```

Output

```
The size of the multiset is initially 2.
The size of the multiset after clearing is 0.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::count

Returns the number of elements in a multiset whose key matches a parameter-specified key.

```
size_type count(
   const Key& _Key
) const;
```

Parameters

_Key

The key of the elements to be matched from the multiset.

Return Value

The number of elements in the multiset whose sort key matches the parameter key.

Remarks

The member function returns the number of elements *x* in the range

[**lower_bound** (*_Key*), **upper_bound** (*_Key*)).

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// multiset_count.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main()
{
    using namespace std;
    multiset<int> ms1;
    multiset<int>::size_type i;

    ms1.insert(1);
    ms1.insert(1);
    ms1.insert(2);

    // Elements do not need to be unique in multiset,
    // so duplicates are allowed and counted.
    i = ms1.count(1);
    cout << "The number of elements in ms1 with a sort key of 1 is: "
         << i << "." << endl;

    i = ms1.count(2);
    cout << "The number of elements in ms1 with a sort key of 2 is: "
         << i << "." << endl;

    i = ms1.count(3);
    cout << "The number of elements in ms1 with a sort key of 3 is: "
         << i << "." << endl;
}
```

Output

```
The number of elements in ms1 with a sort key of 1 is: 2.
The number of elements in ms1 with a sort key of 2 is: 1.
The number of elements in ms1 with a sort key of 3 is: 0.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::empty

Tests if a multiset is empty.

```
bool empty( ) const;
```

Return Value

true if the multiset is empty; **false** if the multiset is nonempty.

Example

```
// multiset_empty.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1, ms2;
   ms1.insert ( 1 );

   if ( ms1.empty( ) )
      cout << "The multiset ms1 is empty." << endl;
   else
      cout << "The multiset ms1 is not empty." << endl;

   if ( ms2.empty( ) )
      cout << "The multiset ms2 is empty." << endl;
   else
      cout << "The multiset ms2 is not empty." << endl;
}
```

Output

```
The multiset ms1 is not empty.
The multiset ms2 is empty.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::end

Returns an iterator that addresses the location succeeding the last element in a multiset.

```
const_iterator end( ) const;
iterator end( );
```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a multiset. If the multiset is empty, then multiset::end == multiset::begin.

Remarks

end is used to test whether an iterator has reached the end of its multiset. The value returned by **end** should not be dereferenced.

Example

```
// multiset_end.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int> :: iterator ms1_Iter;
   multiset <int> :: const_iterator ms1_cIter;

   ms1.insert( 1 );
   ms1.insert( 2 );
   ms1.insert( 3 );

   ms1_Iter = ms1.end( );
   ms1_Iter--;
   cout << "The last element of ms1 is " << *ms1_Iter << endl;

   ms1.erase( ms1_Iter );

   // The following 3 lines would err as the iterator is const
   // ms1_cIter = ms1.end( );
   // ms1_cIter--;
   // ms1.erase( ms1_cIter );

   ms1_cIter = ms1.end( );
   ms1_cIter--;
   cout << "The last element of ms1 is now " << *ms1_cIter << endl;
}
```

Output

```
The last element of ms1 is 3
The last element of ms1 is now 2
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset::equal_range

Returns a pair of iterators respectively to the first element in a multiset with a key that is greater than a specified key and to the first element in the multiset with a key that is equal to or greater than the key.

```
pair <const_iterator, const_iterator>
   equal_range (
      const Key& _Key
   ) const;

pair <iterator, iterator>
   equal_range (
      const Key& _Key
   );
```

Parameters

_Key

The argument key to be compared with the sort key of an element from the multiset being searched.

Return Value

A pair of iterators such that the first is the lower_bound of the key and the second is the upper_bound of the key.

To access the first iterator of a pair *pr* returned by the member function, use *pr*.**first**, and to dereference the lower bound iterator, use *(*pr*.**first**). To access the second iterator of a pair *pr* returned by the member function, use *pr*.**second**, and to dereference the upper bound iterator, use *(*pr*.**second**).

Example

```
// multiset_equal_range.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   typedef multiset<int, less<int> > IntSet;
   IntSet ms1;
   multiset <int> :: const_iterator ms1_RcIter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );

   pair <IntSet::const_iterator, IntSet::const_iterator> p1, p2;
   p1 = ms1.equal_range( 20 );

   cout << "The upper bound of the element with "
        << "a key of 20 in the multiset ms1 is: "
        << *( p1.second ) << "." << endl;

   cout << "The lower bound of the element with "
        << "a key of 20 in the multiset ms1 is: "
        << *( p1.first ) << "." << endl;

   // Compare the upper_bound called directly
   ms1_RcIter = ms1.upper_bound( 20 );
   cout << "A direct call of upper_bound( 20 ) gives "
        << *ms1_RcIter << "," << endl
        << "matching the 2nd element of the pair"
        << " returned by equal_range( 20 )." << endl;

   p2 = ms1.equal_range( 40 );
```

```
    // If no match is found for the key,
    // both elements of the pair return end( )
    if ( ( p2.first == ms1.end( ) ) && ( p2.second == ms1.end( ) ) )
       cout << "The multiset ms1 doesn't have an element "
               << "with a key less than 40." << endl;
    else
       cout << "The element of multiset ms1 with a key >= 40 is: "
                 << *( p1.first ) << "." << endl;
 }
```

Output

```
The upper bound of the element with a key of 20 in the multiset ms1 is: 30.
The lower bound of the element with a key of 20 in the multiset ms1 is: 20.
A direct call of upper_bound( 20 ) gives 30,
matching the 2nd element of the pair returned by equal_range( 20 ).
The multiset ms1 doesn't have an element with a key less than 40.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::erase

Removes an element or a range of elements in a multiset from specified positions or removes elements that match a specified key.

```
iterator erase(
   iterator _Where
);
iterator erase(
   iterator _First,
   iterator _Last
);
size_type erase(
   const key_type& _Key
);
```

Parameters

_Where

Position of the element to be removed from the multiset.

_First

Position of the first element removed from the multiset.

_Last

Position just beyond the last element removed from the multiset.

_Key

The key of the elements to be removed from the multiset.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the multiset if no such element exists.

Note
This return type does not conform to the C++ standard.

For the third member function, returns the number of elements that have been removed from the multiset.

Remarks

In some instances, this method might throw an out_of_range exception. This behavior does not conform to the C++ standard.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// multiset_erase.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main()
{
    using namespace std;
    multiset<int> ms1, ms2, ms3;
    multiset<int>::iterator pIter, Iter1, Iter2;
    int i;
    multiset<int>::size_type n;
```

```
    for (i = 1; i < 5; i++)
    {
        ms1.insert(i);
        ms2.insert(i * i);
        ms3.insert(i - 1);
    }

    // The 1st member function removes an element at a given position
    Iter1 = ++ms1.begin();
    ms1.erase(Iter1);

    cout << "After the 2nd element is deleted, the multiset ms1 is:" ;
    for (pIter = ms1.begin(); pIter != ms1.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;

    // The 2nd member function removes elements
    // in the range [_First, _Last)
    Iter1 = ++ms2.begin();
    Iter2 = --ms2.end();
    ms2.erase(Iter1, Iter2);

    cout << "After the middle two elements are deleted, "
         << "the multiset ms2 is:" ;
    for (pIter = ms2.begin(); pIter != ms2.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;

    // The 3rd member function removes elements with a given _Key
    ms3.insert(2);
    n = ms3.erase(2);

    cout << "After the element with a key of 2 is deleted,\n"
         << "the multiset ms3 is:" ;
    for (pIter = ms3.begin(); pIter != ms3.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;

    // The 3rd member function returns the number of elements removed
    cout << "The number of elements removed from ms3 is: "
         << n << "." << endl;

    // The dereferenced iterator can also be used to specify a key
    Iter1 = ++ms3.begin();
    ms3.erase(Iter1);

    cout << "After another element with a key"
         << endl;
    cout  << "equal to that of the 2nd element is deleted, "
          << "the multiset ms3 is:" ;
    for (pIter = ms3.begin(); pIter != ms3.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;
}
```

Output

```
After the 2nd element is deleted, the multiset ms1 is: 1 3 4.
After the middle two elements are deleted, the multiset ms2 is: 1 16.
After the element with a key of 2 is deleted,
the multiset ms3 is: 0 1 3.
The number of elements removed from ms3 is: 2.
After another element with a key
equal to that of the 2nd element is deleted, the multiset ms3 is: 0 3.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::find

Returns an iterator addressing the first location of an element in a multiset that has a key equivalent to a specified key.

```
iterator find(
   const Key& _Key
);
const_iterator find(
   const Key& _Key
) const;
```

Parameters

_Key

The key to be matched by the sort key of an element from the multiset being searched.

Return Value

An **iterator** or **const_iterator** that addresses the first location of an element with a specified key, or the location succeeding the last element in the multiset if no match is found for the key.

Remarks

The member function returns an iterator that addresses an element in the multiset whose sort key is equivalent to the argument key under a binary predicate that induces an ordering based on a less than comparability relation.

If the return value of **find** is assigned to a **const_iterator**, the multiset object cannot be modified. If the return value of **find** is assigned to a **iterator**, the multiset object can be modified.

Example

```
// multiset_find.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int> :: const_iterator ms1_AcIter, ms1_RcIter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 20 );

   ms1_RcIter = ms1.find( 20 );
   cout << "The first element of multiset ms1 with a key of 20 is: "
        << *ms1_RcIter << "." << endl;

   ms1_RcIter = ms1.find( 40 );

   // If no match is found for the key, end( ) is returned
   if ( ms1_RcIter == ms1.end( ) )
      cout << "The multiset ms1 doesn't have an element "
               << "with a key of 40." << endl;
   else
      cout << "The element of multiset ms1 with a key of 40 is: "
           << *ms1_RcIter << "." << endl;

   // The element at a specific location in the multiset can be
   // found using a dereferenced iterator addressing the location
   ms1_AcIter = ms1.end( );
   ms1_AcIter--;
   ms1_RcIter = ms1.find( *ms1_AcIter );
```

```
    cout << "The first element of ms1 with a key matching" << endl
         << "that of the last element is: "
         << *ms1_RcIter << "." << endl;

    // Note that the first element with a key equal to
    // the key of the last element is not the last element
    if ( ms1_RcIter == --ms1.end( ) )
       cout << "This is the last element of multiset ms1."
            << endl;
    else
       cout << "This is not the last element of multiset ms1."
            << endl;
}
```

Output

```
The first element of multiset ms1 with a key of 20 is: 20.
The multiset ms1 doesn't have an element with a key of 40.
The first element of ms1 with a key matching
that of the last element is: 20.
This is not the last element of multiset ms1.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::get_allocator

Returns a copy of the allocator object used to construct the multiset.

```
Allocator get_allocator( ) const;
```

Return Value

The allocator used by the multiset.

Remarks

Allocators for the multiset class specify how the class manages storage. The default allocators supplied with STL container classes is sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```
// multiset_get_allocator.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int>::allocator_type ms1_Alloc;
   multiset <int>::allocator_type ms2_Alloc;
   multiset <double>::allocator_type ms3_Alloc;
   multiset <int>::allocator_type ms4_Alloc;

   // The following lines declare objects
   // that use the default allocator.
   multiset <int> ms1;
   multiset <int, allocator<int> > ms2;
   multiset <double, allocator<double> > ms3;

   cout << "The number of integers that can be allocated"
        << endl << "before free memory is exhausted: "
        << ms2.max_size( ) << "." << endl;

   cout << "The number of doubles that can be allocated"
        << endl << "before free memory is exhausted: "
        << ms3.max_size( ) <<  "." << endl;

   // The following lines create a multiset ms4
   // with the allocator of multiset ms1
   ms1_Alloc = ms1.get_allocator( );
   multiset <int> ms4( less<int>( ), ms1_Alloc );
   ms4_Alloc = ms4.get_allocator( );

   // Two allocators are interchangeable if
   // storage allocated from each can be
   // deallocated with the other
   if( ms1_Alloc == ms4_Alloc )
   {
      cout << "Allocators are interchangeable."
           << endl;
   }
   else
   {
      cout << "Allocators are not interchangeable."
           << endl;
   }
}
```

Sample Output

The following output is for x86.

```
The number of integers that can be allocated
before free memory is exhausted: 1073741823.
The number of doubles that can be allocated
before free memory is exhausted: 536870911.
Allocators are interchangeable.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::insert

Inserts an element or a range of elements into a multiset.

```
iterator insert(
   const value_type& _Val
);
iterator insert(
   iterator _Where,
   const value_type& _Val
);
template<class InputIterator>
   void insert(
      InputIterator _First,
      InputIterator _Last
   );
```

Parameters

_Val

The value of an element to be inserted into the multiset unless the multiset already contains that element or, more generally, an element whose key is equivalently ordered.

_Where

The place to start searching for the correct point of insertion. Insertion can occur in amortized constant time instead of logarithmic time, if the insertion point immediately follows *_Where*.

_First

The position of the first element to be copied from a multiset.

_Last

The position just beyond the last element to be copied from a multiset.

Return Value

The **insert** member functions returns an iterator that points to the position where the new element was inserted into the multiset.

Remarks

The third member function inserts the sequence of element values into a multiset corresponding to each element addressed by an iterator in the range [*_First*, *_Last*) of a specified multiset.

Example

```
// multiset_insert.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int>::iterator ms1_pIter, ms2_pIter;

   multiset <int, less<int> > ms1, ms2;
   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );
   ms1.insert( 40 );

   cout << "The original ms1 =";
   for ( ms1_pIter = ms1.begin( ); ms1_pIter != ms1.end( ); ms1_pIter++ )
      cout << " " << *ms1_pIter;
```

```
    cout << "." << endl;

    ms1.insert( 20 );
    ms1.insert( --ms1.end( ), 50 );

    cout << "After the insertions, ms1 =";
    for ( ms1_pIter = ms1.begin( ); ms1_pIter != ms1.end( ); ms1_pIter++ )
       cout << " " << *ms1_pIter;
    cout << "." << endl;

    ms2.insert( 100 );
    ms2.insert( ++ms1.begin( ), --ms1.end( ) );

    cout << "ms2 =";
    for ( ms2_pIter = ms2.begin( ); ms2_pIter != ms2.end( ); ms2_pIter++ )
       cout << " " << *ms2_pIter;
    cout << "." << endl;
 }
```

Output

```
The original ms1 = 10 20 30 40.
After the insertions, ms1 = 10 20 20 30 40 50.
ms2 = 20 20 30 40 100.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::key_comp

Retrieves a copy of the comparison object used to order keys in a multiset.

```
key_compare key_comp( ) const;
```

Return Value

Returns the function object that a multiset uses to order its elements, which is the template parameter **Compare**.

For more information on **Compare**, see the Remarks section of the multiset Class topic.

Remarks

The stored object defines the member function:

bool operator(**const Key&** *x*, **const Key&** *y*);

which returns true if *x* strictly precedes *y* in the sort order.

Note that both key_compare and value_compare are synonyms for the template parameter **Compare**. Both types are provided for the classes set and multiset, where they are identical, for compatibility with the classes map and multimap, where they are distinct.

Example

```
// multiset_key_comp.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;

   multiset <int, less<int> > ms1;
   multiset <int, less<int> >::key_compare kc1 = ms1.key_comp( ) ;
   bool result1 = kc1( 2, 3 ) ;
   if( result1 == true )
   {
      cout << "kc1( 2,3 ) returns value of true, "
           << "where kc1 is the function object of s1."
           << endl;
   }
   else
   {
      cout << "kc1( 2,3 ) returns value of false "
           << "where kc1 is the function object of ms1."
           << endl;
   }

   multiset <int, greater<int> > ms2;
   multiset <int, greater<int> >::key_compare kc2 = ms2.key_comp( ) ;
   bool result2 = kc2( 2, 3 ) ;
   if( result2 == true )
   {
      cout << "kc2( 2,3 ) returns value of true, "
           << "where kc2 is the function object of ms2."
           << endl;
   }
   else
   {
      cout << "kc2( 2,3 ) returns value of false, "
           << "where kc2 is the function object of ms2."
           << endl;
   }
```

```
}
```

Output

```
kc1( 2,3 ) returns value of true, where kc1 is the function object of s1.
kc2( 2,3 ) returns value of false, where kc2 is the function object of ms2.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::lower_bound

Returns an iterator to the first element in a multiset with a key that is equal to or greater than a specified key.

```
const_iterator lower_bound(
   const Key& _Key
) const;
iterator lower_bound(
   const Key& _Key
);
```

Parameters

_Key

The argument key to be compared with the sort key of an element from the multiset being searched.

Return Value

An **iterator** or **const_iterator** that addresses the location of an element in a multiset that with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the multiset if no match is found for the key.

Example

```
// multiset_lower_bound.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int> :: const_iterator ms1_AcIter, ms1_RcIter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );

   ms1_RcIter = ms1.lower_bound( 20 );
   cout << "The element of multiset ms1 with a key of 20 is: "
        << *ms1_RcIter << "." << endl;

   ms1_RcIter = ms1.lower_bound( 40 );

   // If no match is found for the key, end( ) is returned
   if ( ms1_RcIter == ms1.end( ) )
      cout << "The multiset ms1 doesn't have an element "
           << "with a key of 40." << endl;
   else
      cout << "The element of multiset ms1 with a key of 40 is: "
           << *ms1_RcIter << "." << endl;

   // The element at a specific location in the multiset can be
   // found using a derefenced iterator addressing the location
   ms1_AcIter = ms1.end( );
   ms1_AcIter--;
   ms1_RcIter = ms1.lower_bound( *ms1_AcIter );
   cout << "The element of ms1 with a key matching "
        << "that of the last element is: "
        << *ms1_RcIter << "." << endl;
}
```

Output

```
The element of multiset ms1 with a key of 20 is: 20.
The multiset ms1 doesn't have an element with a key of 40.
The element of ms1 with a key matching that of the last element is: 30.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::max_size

Returns the maximum length of the multiset.

```
size_type max_size( ) const;
```

Return Value

The maximum possible length of the multiset.

Example

```
// multiset_max_size.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int>::size_type i;

   i = ms1.max_size( );
   cout << "The maximum possible length "
        << "of the multiset is " << i << "." << endl;
}
```

Sample Output

The following output is for x86.

```
The maximum possible length of the multiset is 1073741823.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::multiset

Constructs a multiset that is empty or that is a copy of all or part of some other multiset.

```
multiset( );
explicit multiset (
   const Compare& _Comp
);
explicit multiset (
   const Compare& _Comp,
   const Allocator& _Al
);
set(
   const _ multiset & _Right
);
template<class InputIterator>
   multiset (
      InputIterator _First,
      InputIterator _Last
   );
template<class InputIterator>
   multiset (
      InputIterator _First,
      InputIterator _Last,
      const Compare& _Comp
   );
template<class InputIterator>
   multiset (
      InputIterator _First,
      InputIterator _Last,
      const Compare& _Comp,
      const Allocator& _Al
   );
```

Parameters

_Al

The storage allocator class to be used for this multiset object, which defaults to **Allocator**.

_Comp

The comparison function of type **const Compare** used to order the elements in the multiset, which defaults to **Compare**.

_Right

The multiset of which the constructed multiset is to be a copy.

_First

The position of the first element in the range of elements to be copied.

_Last

The position of the first element beyond the range of elements to be copied.

Remarks

All constructors store a type of allocator object that manages memory storage for the multiset and that can later be returned by calling get_allocator. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their multiset.

All constructors store a function object of type Compare that is used to establish an order among the keys of the multiset and that can later be returned by calling key_comp.

The first three constructors specify an empty initial multiset, the second specifying the type of comparison function (*_Comp*) to

be used in establishing the order of the elements and the third explicitly specifying the allocator type (*_Al*) to be used. The keyword **explicit** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the multiset *_Right*.

The last three constructors copy the range [*_First, _Last*) of a multiset with increasing explicitness in specifying the type of comparison function and allocator.

Example

```
// multiset_ctor.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int>::iterator ms1_Iter, ms2_Iter, ms3_Iter;
   multiset <int>::iterator ms4_Iter, ms5_Iter, ms6_Iter;

   // Create an empty multiset ms0 of key type integer
   multiset <int> ms0;

   // Create an empty multiset ms1 with the key comparison
   // function of less than, then insert 4 elements
   multiset <int, less<int> > ms1;
   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 20 );
   ms1.insert( 40 );

   // Create an empty multiset ms2 with the key comparison
   // function of geater than, then insert 2 elements
   multiset <int, greater<int> > ms2;
   ms2.insert( 10 );
   ms2.insert( 20 );

   // Create a multiset ms3 with the
   // allocator of multiset ms1
   multiset <int>::allocator_type ms1_Alloc;
   ms1_Alloc = ms1.get_allocator( );
   multiset <int> ms3( less<int>( ), ms1_Alloc );
   ms3.insert( 30 );

   // Create a copy, multiset ms4, of multiset ms1
   multiset <int> ms4( ms1 );

   // Create a multiset ms5 by copying the range ms1[_First, _Last)
   multiset <int>::const_iterator ms1_bcIter, ms1_ecIter;
   ms1_bcIter = ms1.begin( );
   ms1_ecIter = ms1.begin( );
   ms1_ecIter++;
   ms1_ecIter++;
   multiset <int> ms5( ms1_bcIter, ms1_ecIter );

   // Create a multiset ms6 by copying the range ms4[_First, _Last)
   // and with the allocator of multiset ms2
   multiset <int>::allocator_type ms2_Alloc;
   ms2_Alloc = ms2.get_allocator( );
   multiset <int> ms6( ms4.begin( ), ++ms4.begin( ), less<int>( ), ms2_Alloc );

   cout << "ms1 =";
   for ( ms1_Iter = ms1.begin( ); ms1_Iter != ms1.end( ); ms1_Iter++ )
      cout << " " << *ms1_Iter;
   cout << endl;

   cout << "ms2 = " << *ms2.begin( ) << " " << *++ms2.begin( )
   << endl;
```

```
    cout << "ms3 =";
    for ( ms3_Iter = ms3.begin( ); ms3_Iter != ms3.end( ); ms3_Iter++ )
       cout << " " << *ms3_Iter;
    cout << endl;

    cout << "ms4 =";
    for ( ms4_Iter = ms4.begin( ); ms4_Iter != ms4.end( ); ms4_Iter++ )
       cout << " " << *ms4_Iter;
    cout << endl;

    cout << "ms5 =";
    for ( ms5_Iter = ms5.begin( ); ms5_Iter != ms5.end( ); ms5_Iter++ )
       cout << " " << *ms5_Iter;
    cout << endl;

    cout << "ms6 =";
    for ( ms6_Iter = ms6.begin( ); ms6_Iter != ms6.end( ); ms6_Iter++ )
       cout << " " << *ms6_Iter;
    cout << endl;
 }
```

Output

```
ms1 = 10 20 20 40
ms2 = 20 10
ms3 = 30
ms4 = 10 20 20 40
ms5 = 10 20
ms6 = 10
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::rbegin

Returns an iterator addressing the first element in a reversed multiset.

```
const_reverse_iterator rbegin( ) const;
reverse_iterator rbegin( );
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed multiset or addressing what had been the last element in the unreversed multiset.

Remarks

rbegin is used with a reversed multiset just as rbegin is used with a multiset.

If the return value of **rbegin** is assigned to a **const_reverse_iterator**, then the multiset object cannot be modified. If the return value of **rbegin** is assigned to a **reverse_iterator**, then the multiset object can be modified.

rbegin can be used to iterate through a multiset backwards.

Example

```
// multiset_rbegin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int>::iterator ms1_Iter;
   multiset <int>::reverse_iterator ms1_rIter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );

   ms1_rIter = ms1.rbegin( );
   cout << "The first element in the reversed multiset is "
        << *ms1_rIter << "." << endl;

   // begin can be used to start an interation
   // throught a multiset in a forward order
   cout << "The multiset is:";
   for ( ms1_Iter = ms1.begin( ) ; ms1_Iter != ms1.end( ); ms1_Iter++ )
      cout << " " << *ms1_Iter;
   cout << endl;

   // rbegin can be used to start an interation
   // throught a multiset in a reverse order
   cout << "The reversed multiset is:";
   for ( ms1_rIter = ms1.rbegin( ) ; ms1_rIter != ms1.rend( ); ms1_rIter++ )
      cout << " " << *ms1_rIter;
   cout << endl;

   // A multiset element can be erased by dereferencing to its key
   ms1_rIter = ms1.rbegin( );
   ms1.erase ( *ms1_rIter );

   ms1_rIter = ms1.rbegin( );
   cout << "After the erasure, the first element "
        << "in the reversed multiset is "<< *ms1_rIter << "."
        << endl;
```

```
}
```

Output

```
The first element in the reversed multiset is 30.
The multiset is: 10 20 30
The reversed multiset is: 30 20 10
After the erasure, the first element in the reversed multiset is 20.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::rend

Returns an iterator that addresses the location succeeding the last element in a reversed multiset.

```
const_reverse_iterator rend( ) const;
reverse_iterator rend( );
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed multiset (the location that had preceded the first element in the unreversed multiset).

Remarks

rend is used with a reversed multiset just as end is used with a multiset.

If the return value of **rend** is assigned to a **const_reverse_iterator**, then the multiset object cannot be modified. If the return value of **rend** is assigned to a **reverse_iterator**, then the multiset object can be modified.

rend can be used to test to whether a reverse iterator has reached the end of its multiset.

The value returned by **rend** should not be dereferenced.

Example

```
// multiset_rend.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main() {
   using namespace std;
   multiset <int> ms1;
   multiset <int>::iterator ms1_Iter;
   multiset <int>::reverse_iterator ms1_rIter;
   multiset <int>::const_reverse_iterator ms1_crIter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );

   ms1_rIter = ms1.rend( ) ;
   ms1_rIter--;
   cout << "The last element in the reversed multiset is "
        << *ms1_rIter << "." << endl;

   // end can be used to terminate an interation
   // throught a multiset in a forward order
   cout << "The multiset is: ";
   for ( ms1_Iter = ms1.begin( ) ; ms1_Iter != ms1.end( ); ms1_Iter++ )
      cout << *ms1_Iter << " ";
   cout << "." << endl;

   // rend can be used to terminate an interation
   // throught a multiset in a reverse order
   cout << "The reversed multiset is: ";
   for ( ms1_rIter = ms1.rbegin( ) ; ms1_rIter != ms1.rend( ); ms1_rIter++ )
      cout << *ms1_rIter << " ";
   cout << "." << endl;

   ms1_rIter = ms1.rend( );
   ms1_rIter--;
   ms1.erase ( *ms1_rIter );

   ms1_rIter = ms1.rend( );
   --ms1_rIter;
```

```
    cout << "After the erasure, the last element in the "
         << "reversed multiset is " << *ms1_rIter << "." << endl;
}
```

Output

```
The last element in the reversed multiset is 10.
The multiset is: 10 20 30 .
The reversed multiset is: 30 20 10 .
After the erasure, the last element in the reversed multiset is 20.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::size

Returns the number of elements in the multiset.

```
size_type size( ) const;
```

Return Value

The current length of the multiset.

Example

```
// multiset_size.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int> :: size_type i;

   ms1.insert( 1 );
   i = ms1.size( );
   cout << "The multiset length is " << i << "." << endl;

   ms1.insert( 2 );
   i = ms1.size( );
   cout << "The multiset length is now " << i << "." << endl;
}
```

Output

```
The multiset length is 1.
The multiset length is now 2.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::swap

Exchanges the elements of two multisets.

```
void swap(
   multiset& _Right
);
```

Parameters

_Right

The argument multiset providing the elements to be swapped with the target multiset.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two multisets whose elements are being exchanged.

Example

```
// multiset_swap.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1, ms2, ms3;
   multiset <int>::iterator ms1_Iter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );
   ms2.insert( 100 );
   ms2.insert( 200 );
   ms3.insert( 300 );

   cout << "The original multiset ms1 is:";
   for ( ms1_Iter = ms1.begin( ); ms1_Iter != ms1.end( ); ms1_Iter++ )
      cout << " " << *ms1_Iter;
   cout << "." << endl;

   // This is the member function version of swap
   ms1.swap( ms2 );

   cout << "After swapping with ms2, list ms1 is:";
   for ( ms1_Iter = ms1.begin( ); ms1_Iter != ms1.end( ); ms1_Iter++ )
      cout << " " << *ms1_Iter;
   cout << "." << endl;

   // This is the specialized template version of swap
   swap( ms1, ms3 );

   cout << "After swapping with ms3, list ms1 is:";
   for ( ms1_Iter = ms1.begin( ); ms1_Iter != ms1.end( ); ms1_Iter++ )
      cout << " " << *ms1_Iter;
   cout   << "." << endl;
}
```

Output

```
The original multiset ms1 is: 10 20 30.
After swapping with ms2, list ms1 is: 100 200.
After swapping with ms3, list ms1 is: 300.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::upper_bound

Returns an iterator to the first element in a multiset that with a key that is equal to or greater than a specified key.

```
const_iterator upper_bound(
   const Key& _Key
) const;
iterator upper_bound(
   const Key& _Key
);
```

Parameters

_Key

The argument key to be compared with the sort key of an element from the multiset being searched.

Return Value

An **iterator** or **const_iterator** that addresses the location of an element in a multiset that with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the multiset if no match is found for the key.

Example

```
// multiset_upper_bound.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int> :: const_iterator ms1_AcIter, ms1_RcIter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );

   ms1_RcIter = ms1.upper_bound( 20 );
   cout << "The first element of multiset ms1 with a key greater "
           << "than 20 is: " << *ms1_RcIter << "." << endl;

   ms1_RcIter = ms1.upper_bound( 30 );

   // If no match is found for the key, end( ) is returned
   if ( ms1_RcIter == ms1.end( ) )
      cout << "The multiset ms1 doesn't have an element "
              << "with a key greater than 30." << endl;
   else
      cout << "The element of multiset ms1 with a key > 40 is: "
           << *ms1_RcIter << "." << endl;

   // The element at a specific location in the multiset can be
   // found using a dereferenced iterator addressing the location
   ms1_AcIter = ms1.begin( );
   ms1_RcIter = ms1.upper_bound( *ms1_AcIter );
   cout << "The first element of ms1 with a key greater than"
        << endl << "that of the initial element of ms1 is: "
        << *ms1_RcIter << "." << endl;
}
```

Output

```
The first element of multiset ms1 with a key greater than 20 is: 30.
The multiset ms1 doesn't have an element with a key greater than 30.
The first element of ms1 with a key greater than
that of the initial element of ms1 is: 20.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

multiset::value_comp

Retrieves a copy of the comparison object used to order element values in a multiset.

```
value_compare value_comp( ) const;
```

Return Value

Returns the function object that a multiset uses to order its elements, which is the template parameter **Compare**.

For more information on **Compare**, see the Remarks section of the multiset Class topic.

Remarks

The stored object defines the member function:

bool operator(**const Key&** *_xVal*, **const Key&** *_yVal*);

which returns true if *_xVal* precedes and is not equal to *_yVal* in the sort order.

Note that both key_compare and value_compare are synonyms for the template parameter **Compare**. Both types are provided for the classes set and multiset, where they are identical, for compatibility with the classes map and multimap, where they are distinct.

Example

```
// multiset_value_comp.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;

   multiset <int, less<int> > ms1;
   multiset <int, less<int> >::value_compare vc1 = ms1.value_comp( );
   bool result1 = vc1( 2, 3 );
   if( result1 == true )
   {
      cout << "vc1( 2,3 ) returns value of true, "
           << "where vc1 is the function object of ms1."
           << endl;
   }
   else
   {
      cout << "vc1( 2,3 ) returns value of false, "
           << "where vc1 is the function object of ms1."
           << endl;
   }

   set <int, greater<int> > ms2;
   set<int, greater<int> >::value_compare vc2 = ms2.value_comp( );
   bool result2 = vc2( 2, 3 );
   if( result2 == true )
   {
      cout << "vc2( 2,3 ) returns value of true, "
           << "where vc2 is the function object of ms2."
           << endl;
   }
   else
   {
      cout << "vc2( 2,3 ) returns value of false, "
           << "where vc2 is the function object of ms2."
           << endl;
   }
```

```
}
```

Output

```
vc1( 2,3 ) returns value of true, where vc1 is the function object of ms1.
vc2( 2,3 ) returns value of false, where vc2 is the function object of ms2.
```

Requirements

Header: <set>

See Also

Reference

multiset Class

Standard Template Library

Other Resources

multiset Members

<sstream>

Defines several template classes that support iostreams operations on sequences stored in an allocated array object. Such sequences are easily converted to and from objects of template class basic_string.

For a list of all members of this header, see <sstream> Members.

```
#include <sstream>
```

Remarks

Objects of type **char *** can use the functionality in <strstream> for streaming. However, **<strstream>** is deprecated and the use of **<sstream>** is encouraged.

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

<sstream> Members
Standard C++ Library Header Files

<sstream> Members

Reference

Typedefs

istringstream	Creates a type **basic_istringstream** specialized on a **char** template parameter.
ostringstream	Creates a type **basic_ostringstream** specialized on a **char** template parameter.
stringbuf	Creates a type **basic_stringbuf** specialized on a **char** template parameter.
stringstream	Creates a type **basic_stringstream** specialized on a **char** template parameter.
wistringstream	Creates a type **basic_istringstream** specialized on a **wchar_t** template parameter.
wostringstream	Creates a type **basic_ostringstream** specialized on a **wchar_t** template parameter.
wstringbuf	Creates a type **basic_stringbuf** specialized on a **wchar_t** template parameter.
wstringstream	Creates a type **basic_stringstream** specialized on a **wchar_t** template parameter.

Classes

basic_stringbuf	Describes a stream buffer that controls the transmission of elements of type **Elem**, whose character traits are determined by the class **Tr**, to and from a sequence of elements stored in an array object.
basic_istringstream	Describes an object that controls extraction of elements and encoded objects from a stream buffer of class basic_stringbuf<**Elem**, **Tr**, **Alloc**>, with elements of type **Elem**, whose character traits are determined by the class **Tr**, and whose elements are allocated by an allocator of class **Alloc**.
basic_ostringstream	Describes an object that controls insertion of elements and encoded objects into a stream buffer of class basic_stringbuf<**Elem**, **Tr**, **Alloc**>, with elements of type **Elem**, whose character traits are determined by the class **Tr**, and whose elements are allocated by an allocator of class **Alloc**.
basic_stringstream	Describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class basic_stringbuf<**Elem**, **Tr**, **Alloc**>, with elements of type **Elem**, whose character traits are determined by the class **Tr**, and whose elements are allocated by an allocator of class **Alloc**.

See Also

Reference

<sstream>
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

<sstream> Typedefs

For information about the typedefs in <sstream>, see <sstream> Members.

See Also

Reference

iostream Programming

iostreams Conventions

istringstream

Creates a type **basic_istringstream** specialized on a **char** template parameter.

```
typedef basic_istringstream<char> istringstream;
```

Remarks

The type is a synonym for template class basic_istringstream, specialized for elements of type **char**.

Requirements

Header: <sstream>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<sstream> Members

ostringstream

Creates a type **basic_ostringstream** specialized on a **char** template parameter.

```
typedef basic_ostringstream<char> ostringstream;
```

Remarks

The type is a synonym for template class basic_ostringstream, specialized for elements of type **char**.

Requirements

Header: <sstream>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<sstream> Members

stringbuf

Creates a type **basic_stringbuf** specialized on a **char** template parameter.

```
typedef basic_stringbuf<char> stringbuf;
```

Remarks

The type is a synonym for template class basic_stringbuf, specialized for elements of type **char**.

Requirements

Header: <sstream>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<sstream> Members

stringstream

Creates a type **basic_stringstream** specialized on a **char** template parameter.

```
typedef basic_stringstream<char> stringstream;
```

Remarks

The type is a synonym for template class basic_stringstream, specialized for elements of type **char**.

Requirements

Header: <sstream>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<sstream> Members

wistringstream

Creates a type **basic_istringstream** specialized on a **wchar_t** template parameter.

```
typedef basic_istringstream<wchar_t> wistringstream;
```

Remarks

The type is a synonym for template class basic_istringstream, specialized for elements of type **wchar_t**.

Requirements

Header: <sstream>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<sstream> Members

wostringstream

Creates a type **basic_ostringstream** specialized on a **wchar_t** template parameter.

```
typedef basic_ostringstream<wchar_t> wostringstream;
```

Remarks

The type is a synonym for template class basic_ostringstream, specialized for elements of type **wchar_t**.

Requirements

Header: <sstream>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<sstream> Members

wstringbuf

Creates a type **basic_stringbuf** specialized on a **wchar_t** template parameter.

```
typedef basic_stringbuf<wchar_t> wstringbuf;
```

Remarks

The type is a synonym for template class basic_stringbuf, specialized for elements of type **wchar_t**.

Requirements

Header: <sstream>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<sstream> Members

wstringstream

Creates a type **basic_stringstream** specialized on a **wchar_t** template parameter.

```
typedef basic_stringstream<wchar_t> wstringstream;
```

Remarks

The type is a synonym for template class basic_stringstream, specialized for elements of type **wchar_t**.

Requirements

Header: <sstream>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<sstream> Members

<sstream> Classes

For information about the classes in <sstream>, see <sstream> Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_stringbuf Class

Describes a stream buffer that controls the transmission of elements of type *Elem*, whose character traits are determined by the class *Tr*, to and from a sequence of elements stored in an array object.

For a list of all members of this type, see basic_stringbuf Members.

```
template <class Elem, class Tr = char_traits<Elem>,
   class Alloc = allocator<Elem>
>
   class basic_stringbuf : public basic_streambuf<Elem, Tr>
```

Parameters

Alloc

The allocator class.

Elem

The type of the basic element of the string.

Tr

The character traits specialized on the basic element of the string.

Remarks

The object is allocated, extended, and freed as necessary to accommodate changes in the sequence.

An object of class basic_stringbuf<*Elem*, *Tr*, *Alloc*> stores a copy of the **ios_base::**openmode argument from its constructor as its **stringbuf** mode **mode**:

- If `mode & ios_base::in` is nonzero, the input buffer is accessible. For more information, see basic_streambuf Class.
- If `mode & ios_base::out` is nonzero, the output buffer is accessible.

Requirements

Header: <sstream>

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

basic_stringbuf Members
<sstream> Members

basic_stringbuf Members

The following tables list the members of the basic_stringbuf Class. For a list of members inherited by the **basic_stringbuf** class, see basic_streambuf Members.

Reference

Constructors

basic_stringbuf	Constructs an object of type **basic_stringbuf**.

Typedefs

allocator_type	The type is a synonym for the template parameter *Alloc*.
char_type	Associates a type name with the *Elem* template parameter.
int_type	Makes this type within **basic_filebuf**'s scope equivalent to the type of the same name in the *Tr* scope.
off_type	Makes this type within **basic_filebuf**'s scope equivalent to the type of the same name in the *Tr* scope.
pos_type	Makes this type within **basic_filebuf**'s scope equivalent to the type of the same name in the *Tr* scope.
traits_type	Associates a type name with the *Tr* template parameter.

Member Functions

overflow	A protected, virtual function that can be called when a new character is inserted into a full buffer.
pbackfail	The protected virtual member function tries to put back an element into the input buffer, then makes it the current e lement (pointed to by the next pointer).
seekoff	The protected virtual member function tries to alter the current positions for the controlled streams.
seekpos	The protected virtual member function tries to alter the current positions for the controlled streams.
str	Sets or gets the text in a string buffer without changing the write position.
underflow	The protected virtual member function to extract the current element from the input stream.

See Also

Reference

basic_stringbuf Class
basic_streambuf Class
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_stringbuf Typedefs

For information about the typedefs in the basic_filebuf class, see basic_stringbuf Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_stringbuf::allocator_type

The type is a synonym for the template parameter *Alloc*.

```
typedef Alloc allocator_type;
```

Requirements

Header: <sstream>

See Also

Reference

basic_stringbuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_stringbuf Members

basic_stringbuf::char_type

Associates a type name with the **Elem** template parameter.

```
typedef Elem char_type;
```

Requirements

Header: <sstream>

See Also

Reference

basic_stringbuf Class

iostream Programming

iostreams Conventions

Other Resources

basic_stringbuf Members

basic_stringbuf::int_type

Makes this type within basic_filebuf's scope equivalent to the type of the same name in the **Tr** scope.

```
typedef typename traits_type::int_type int_type;
```

Requirements

Header: <sstream>

See Also

Reference

basic_stringbuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_stringbuf Members

basic_stringbuf::off_type

Makes this type within basic_filebuf's scope equivalent to the type of the same name in the **Tr** scope.

```
typedef typename traits_type::off_type off_type;
```

Requirements

Header: <sstream>

See Also

Reference

basic_stringbuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_stringbuf Members

basic_stringbuf::pos_type

Makes this type within basic_filebuf's scope equivalent to the type of the same name in the **Tr** scope.

```
typedef typename traits_type::pos_type pos_type;
```

Requirements

Header: <sstream>

See Also

Reference

basic_stringbuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_stringbuf Members

basic_stringbuf::traits_type

Associates a type name with the **Tr** template parameter.

```
typedef Tr traits_type;
```

Remarks

The type is a synonym for the template parameter **Tr**.

Requirements

Header: <sstream>

See Also

Reference

basic_stringbuf Class

iostream Programming

iostreams Conventions

Other Resources

basic_stringbuf Members

basic_stringbuf Member Functions

For information about the member functions in the basic_stringbuf class, see basic_stringbuf Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_stringbuf::basic_stringbuf

Constructs an object of type **basic_stringbuf**.

```
basic_stringbuf(
   ios_base::openmode _Mode = ios_base::in | ios_base::out
);
basic_stringbuf(
   basic_string<Elem, Tr, Alloc>& _Str,
   ios_base::openmode _Mode = ios_base::in | ios_base::out
);
```

Parameters

_Mode

One of the enumerations in ios_base::openmode.

_Str

An object of type basic_string.

Remarks

The first constructor stores a null pointer in all the pointers controlling the input buffer and the output buffer. For more information, see the Remarks section of the basic_streambuf Class. It also stores *_Mode* as the stringbuf mode. For more information, see the Remarks section of the basic_stringbuf Class.

The second constructor allocates a copy of the sequence controlled by the string object *_Str*. If `_Mode & ios_base::in` is nonzero, it sets the input buffer to start reading at the start of the sequence. If `_Mode & ios_base::out` is nonzero, it sets the output buffer to begin writing at the start of the sequence. It also stores *_Mode* as the stringbuf mode. For more information, see the Remarks section of the basic_stringbuf Class.

Requirements

Header: <sstream>

See Also

Reference

basic_stringbuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_stringbuf Members

basic_stringbuf::overflow

A protected virtual function that can be called when a new character is inserted into a full buffer.

```
virtual int_type overflow(
   int_type _Meta = traits_type::eof( )
);
```

Parameters

_Meta

The character to insert into the buffer, or **traits_type::eof**.

Return Value

If the function cannot succeed, it returns **traits_type::eof**. Otherwise, it returns **traits_type::**not_eof(*_Meta*).

Remarks

If *_Meta* does not compare equal to **traits_type::**eof, the protected virtual member function tries to insert the element **traits_type::**to_char_type(*_Meta*) into the output buffer. It can do so in various ways:

- If a write position is available, it can store the element into the write position and increment the next pointer for the output buffer.
- It can make a write position available by allocating new or additional storage for the output buffer. Extending the output buffer this way also extends any associated input buffer.

Requirements

Header: <sstream>

See Also

Reference

basic_stringbuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_stringbuf Members

basic_stringbuf::pbackfail

The protected virtual member function tries to put back an element into the input buffer, and then make it the current element (pointed to by the next pointer).

```
virtual int_type pbackfail(
      int_type _Meta = traits_type::eof( )
);
```

Parameters

_Meta

The character to insert into the buffer, or **traits_type::eof**.

Return Value

If the function cannot succeed, it returns **traits_type::eof**. Otherwise, it returns **traits_type::**not_eof(*_Meta*).

Remarks

If *_Meta* compares equal to **traits_type::**eof, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by **byte** = **traits_type::**to_char_type(*_Meta*). The function can put back an element in various ways:

- If a putback position is available, and the element stored there compares equal to byte, it can decrement the next pointer for the input buffer.
- If a putback position is available, and if the stringbuf mode permits the sequence to be altered (**mode & ios_base::out** is nonzero), it can store byte into the putback position and decrement the next pointer for the input buffer.

Requirements

Header: <sstream>

See Also

Reference

basic_stringbuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_stringbuf Members

basic_stringbuf::seekoff

The protected virtual member function tries to alter the current positions for the controlled streams.

```
virtual pos_type seekoff(
   off_type _Off,
   ios_base::seekdir _Way,
   ios_base::openmode _Mode = ios_base::in | ios_base::out
);
```

Parameters

_Off

The position to seek for relative to *_Way*.

_Way

The starting point for offset operations. See seekdir for possible values.

_Mode

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

Returns the new position or an invalid stream position.

Remarks

For an object of class basic_stringbuf<**Elem**, **Tr**, **Alloc**>, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence.

The new position is determined as follows:

- If *_Way* == ios_base::beg, the new position is the beginning of the stream plus *_Off*.
- If *_Way* == ios_base::cur, the new position is the current stream position plus *_Off*.
- If *_Way* == ias_base::end, the new position is the end of the stream plus *_Off*.

If **mode & ios_base::in** is nonzero, the function alters the next position to read in the input buffer. If **mode & ios_base::out** is nonzero, the function alters the next position to write in the output buffer. For a stream to be affected, its buffer must exist. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence. If the function affects both stream positions, *_Way* must be **ios_base::beg** or **ios_base::end** and both streams are positioned at the same element. Otherwise (or if neither position is affected), the positioning operation fails.

If the function succeeds in altering either or both of the stream positions, it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position.

Requirements

Header: <sstream>

See Also

Reference

basic_stringbuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_stringbuf Members

basic_stringbuf::seekpos

The protected virtual member function tries to alter the current positions for the controlled streams.

```
virtual pos_type seekpos(
   pos_type _Sp,
   ios_base::openmode _Mode = ios_base::in | ios_base::out
);
```

Parameters

_Sp

The position to seek for.

_Mode

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

If the function succeeds in altering either or both of the stream positions, it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position. To determine if the stream position is invalid, compare the return value with `pos_type(off_type(-1))`.

Remarks

For an object of class basic_stringbuf<**Elem**, **Tr**, **Alloc**>, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence. The new position is determined by *_Sp*.

If **mode & ios_base::in** is nonzero, the function alters the next position to read in the input buffer. If **mode & ios_base::out** is nonzero, the function alters the next position to write in the output buffer. For a stream to be affected, its buffer must exist. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence. Otherwise (or if neither position is affected), the positioning operation fails.

Requirements

Header: <sstream>

See Also

Reference

basic_stringbuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_stringbuf Members

basic_stringbuf::str

Sets or gets the text in a string buffer without changing the write position.

```
basic_string<Elem, Tr, Alloc> str( ) const;
void str(
   basic_string<Elem, Tr, Alloc>& _Newstr
);
```

Parameters

_Newstr

The new string.

Return Value

Returns an object of class basic_string<**Elem**, **Tr**, Alloc**>,** whose controlled sequence is a copy of the sequence controlled by ***this**.

Remarks

The first member function returns an object of class basic_string<**Elem**, **Tr**, **Alloc**>, whose controlled sequence is a copy of the sequence controlled by ***this**. The sequence copied depends on the stored stringbuf mode:

- If **mode & ios_base::out** is nonzero and an output buffer exists, the sequence is the entire output buffer (epptr - pbase elements beginning with **pbase**).
- If **mode & ios_base::in** is nonzero and an input buffer exists, the sequence is the entire input buffer (egptr - eback elements beginning with **eback**).
- Otherwise, the copied sequence is empty.

The second member function deallocates any sequence currently controlled by ***this**. It then allocates a copy of the sequence controlled by *_Newstr*. If **mode & ios_base::in** is nonzero, it sets the input buffer to start reading at the beginning of the sequence. If **mode & ios_base::out** is nonzero, it sets the output buffer to start writing at the beginning of the sequence.

Example

```
// basic_stringbuf_str.cpp
// compile with: /EHsc
#include <iostream>
#include <sstream>

using namespace std;

int main( )
{
   basic_string<char> i( "test" );
   stringstream ss;

   ss.rdbuf( )->str( i );
   cout << ss.str( ) << endl;

   ss << "z";
   cout << ss.str( ) << endl;

   ss.rdbuf( )->str( "be" );
   cout << ss.str( ) << endl;
}
```

Output

```
test
zest
be
```

Requirements

Header: <sstream>

See Also

Reference

basic_stringbuf Class

iostream Programming

iostreams Conventions

Other Resources

basic_stringbuf Members

basic_stringbuf::underflow

Protected, virtual function to extract the current element from the input stream.

```
virtual int_type underflow( );
```

Return Value

If the function cannot succeed, it returns **traits_type::**eof. Otherwise, it returns the current element in the input stream, which are converted.

Remarks

The protected virtual member function tries to extract the current element **byte** from the input buffer, advance the current stream position, and return the element as **traits_type::**to_int_type(**byte**). It can do so in one way: If a read position is available, it takes **byte** as the element stored in the read position and advances the next pointer for the input buffer.

Requirements

Header: <sstream>

See Also

Reference

basic_stringbuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_stringbuf Members

basic_istringstream Class

Describes an object that controls extraction of elements and encoded objects from a stream buffer of class basic_stringbuf<**Elem**, **Tr**, **Alloc**>.

For a list of all members of this type, see basic_istringstream Members.

```
template <
   class Elem,
   class Tr = char_traits<Elem>,
   class Alloc = allocator<Elem>
>
   class basic_istringstream : public basic_istream<Elem, Tr>
```

Parameters

Alloc

The allocator class.

Elem

The type of the basic element of the string.

Tr

The character traits specialized on the basic element of the string.

Remarks

The template class describes an object that controls extraction of elements and encoded objects from a stream buffer of class basic_stringbuf<**Elem**, **Tr**, **Alloc**>, with elements of type **Elem**, whose character traits are determined by the class **Tr**, and whose elements are allocated by an allocator of class **Alloc**. The object stores an object of class basic_stringbuf<**Elem**, **Tr**, **Alloc**>.

Requirements

Header: <sstream>

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

basic_istringstream Members
<sstream> Members

basic_istringstream Members

The following tables list the members of the basic_istringstream Class. For a list of members inherited by the **basic_istringstream** class, see basic_istream Members.

Reference

Constructors

basic_istringstream	Constructs an object of type **basic_istringstream**.

Typedefs

allocator_type	The type is a synonym for the template parameter *Alloc*.

Member Functions

rdbuf	Returns the address of the stored stream buffer of type **pointer** to basic_stringbuf<*Elem*, *Tr*, *Alloc*>.
str	Sets or gets the text in a string buffer without changing the write position.

See Also

Reference

basic_istringstream Class
basic_istream Class
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

basic_istream Members

basic_istringstream Typedefs

For information about the typedefs in the basic_filebuf class, see basic_stringbuf Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_istringstream::allocator_type

The type is a synonym for the template parameter *Alloc*.

```
typedef Alloc allocator_type;
```

Requirements

Header: <sstream>

See Also

Reference

basic_istringstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_istringstream Members

basic_istringstream Member Functions

For information about the member functions in the basic_istringstream class, see basic_istringstream Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_istringstream::basic_istringstream

Constructs an object of type **basic_istringstream**.

```
explicit basic_istringstream(
   ios_base::openmode _Mode = ios_base::in
);
explicit basic_istringstream(
   const basic_string<Elem, Tr, Alloc>& _Str,
   ios_base::openmode _Mode = ios_base::in
);
```

Parameters

_Mode

One of the enumerations in ios_base::openmode.

_Str

An object of type **basic_string**.

Remarks

The first constructor initializes the base class by calling basic_istream(**sb**), where **sb** is the stored object of class basic_stringbuf<**Elem**, **Tr**, **Alloc**>. It also initializes **sb** by calling basic_stringbuf<**Elem**, **Tr**, **Alloc**>(*_Mode* | **ios_base::in**).

The second constructor initializes the base class by calling basic_istream(**sb**). It also initializes **sb** by calling basic_stringbuf<**Elem**, **Tr**, **Alloc**>(*_Str*, *_Mode* | **ios_base::in**).

Requirements

Header: <sstream>

See Also

Reference

basic_istringstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_istringstream Members

basic_istringstream::rdbuf

Returns the address of the stored stream buffer of type **pointer** to basic_stringbuf<**Elem**, **Tr**, **Alloc**>.

```
basic_stringbuf<Elem, Tr, Alloc> *rdbuf( ) const
```

Return Value

The address of the stored stream buffer of type **pointer** to basic_stringbuf<**Elem**, **Tr**, **Alloc**>.

Example

See basic_filebuf::close for an example that uses **rdbuf**.

Requirements

Header: <sstream>

See Also

Reference

basic_istringstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_istringstream Members

basic_istringstream::str

Sets or gets the text in a string buffer without changing the write position.

```
basic_string<Elem, Tr, Alloc> str( ) const;
void str(
   basic_string<Elem, Tr, Alloc>& _Newstr
);
```

Parameters

_Newstr

The new string.

Return Value

Returns an object of class basic_string<**Elem**, **Tr**, **Alloc**>, whose controlled sequence is a copy of the sequence controlled by ***this**.

Remarks

The first member function returns rdbuf -> str. The second member function calls **rdbuf** -> **str**(*_Newstr*).

Example

See basic_stringbuf::str for an example that uses **str**.

Requirements

Header: <sstream>

See Also

Reference

basic_istringstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_istringstream Members

basic_ostringstream Class

Describes an object that controls insertion of elements and encoded objects into a stream buffer of class basic_stringbuf<**Elem**, **Tr**, **Alloc**>.

For a list of all members of this type, see basic_ostringstream Members.

```
template <
   class Elem,
   class Tr = char_traits<Elem>,
   class Alloc = allocator<Elem>
>
   class basic_ostringstream : public basic_ostream<Elem, Tr>
```

Parameters

Alloc

The allocator class.

Elem

The type of the basic element of the string.

Tr

The character traits specialized on the basic element of the string.

Remarks

The class describes an object that controls insertion of elements and encoded objects into a stream buffer, with elements of type **Elem**, whose character traits are determined by the class **Tr**, and whose elements are allocated by an allocator of class **Alloc**. The object stores an object of class basic_stringbuf<**Elem**, **Tr**, **Alloc**>.

Requirements

Header: <sstream>

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

basic_ostringstream Members
<sstream> Members

basic_ostringstream Members

The following tables list the members of the basic_ostringstream Class. For a list of members inherited by the **basic_ostringstream** class, see basic_ostream Members.

Reference

Constructors

basic_ostringstream	Constructs an object of type **basic_ostringstream**.

Typedefs

allocator_type	The type is a synonym for the template parameter *Alloc*.

Member Functions

rdbuf	Returns the address of the stored stream buffer of type **pointer** to basic_stringbuf<*Elem*, *Tr*, *Alloc*>.
str	Sets or gets the text in a string buffer without changing the write position.

See Also

Reference

basic_ostringstream Class
basic_ostream Class
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

basic_ostream Members

basic_ostringstream Typedefs

For information about the typedefs in the basic_filebuf class, see basic_stringbuf Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_ostringstream::allocator_type

The type is a synonym for the template parameter *Alloc*.

```
typedef Alloc allocator_type;
```

Requirements

Header: <sstream>

See Also

Reference

basic_ostringstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ostringstream Members

basic_ostringstream Member Functions

For information about the member functions in the basic_ostringstream class, see basic_ostringstream Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_ostringstream::basic_ostringstream

Constructs an object of type basic_ostringstream.

```
explicit basic_ostringstream(
   ios_base::openmode _Mode = ios_base::out
);
explicit basic_ostringstream(
   const basic_string<Elem, Tr, Alloc>& _Str,
   ios_base::openmode _Mode = ios_base::out
);
```

Parameters

_Mode

One of the enumerations in ios_base::openmode.

_Str

An object of type **basic_string**.

Remarks

The first constructor initializes the base class by calling basic_ostream(**sb**), where **sb** is the stored object of class basic_stringbuf<**Elem**, **Tr**, **Alloc**>. It also initializes **sb** by calling basic_stringbuf<**Elem**, **Tr**, **Alloc**>(*_Mode* | **ios_base::out**).

The second constructor initializes the base class by calling basic_ostream(**sb**). It also initializes **sb** by calling basic_stringbuf<**Elem**, **Tr**, **Alloc**>(*_Str*, *_Mode* | **ios_base::out**).

Requirements

Header: <sstream>

See Also

Reference

basic_ostringstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ostringstream Members

basic_ostringstream::rdbuf

Returns the address of the stored stream buffer of type **pointer** to basic_stringbuf<**Elem**, **Tr**, **Alloc**>.

```
basic_stringbuf<Elem, Tr, Alloc> *rdbuf( ) const
```

Return Value

The address of the stored stream buffer, of type **pointer** to basic_stringbuf<**Elem**, **Tr**, **Alloc**>.

Remarks

The member function returns the address of the stored stream buffer of type **pointer** to basic_stringbuf<**Elem**, **Tr**, **Alloc**>.

Example

See basic_filebuf::close for an example that uses **rdbuf**.

Requirements

Header: <sstream>

See Also

Reference

basic_ostringstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ostringstream Members

basic_ostringstream::str

Sets or gets the text in a string buffer without changing the write position.

```
basic_string<Elem, Tr, Alloc> str( ) const;
void str(
   basic_string<Elem, Tr, Alloc>& _Newstr
);
```

Parameters

_Newstr

The new string.

Return Value

Returns an object of class basic_string<**Elem**, **Tr**, **Alloc**>, whose controlled sequence is a copy of the sequence controlled by ***this**.

Remarks

The first member function returns rdbuf -> str. The second member function calls **rdbuf** -> **str**(*_Newstr*).

Example

See basic_stringbuf::str for an example that uses **str**.

Requirements

Header: <sstream>

See Also

Reference

basic_ostringstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_ostringstream Members

basic_stringstream Class

Describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class basic_stringbuf<**Elem**, **Tr**, **Alloc**>.

For a list of all members of this type, see basic_stringstream Members.

```
template <
   class Elem,
   class Tr = char_traits<Elem>,
   class Alloc = allocator<Elem>
>
   class basic_stringstream : public basic_iostream<Elem, Tr>
```

Parameters

Alloc

The allocator class.

Elem

The type of the basic element of the string.

Tr

The character traits specialized on the basic element of the string.

Remarks

The template class describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class basic_stringbuf<**Elem**, **Tr**, **Alloc**>, with elements of type **Elem**, whose character traits are determined by the class **Tr**, and whose elements are allocated by an allocator of class **Alloc**. The object stores an object of class basic_stringbuf<**Elem**, **Tr**, **Alloc**>.

Requirements

Header: <sstream>

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

basic_stringstream Members
<sstream> Members

basic_stringstream Members

The following tables list the members of the basic_stringstream Class. For a list of members inherited by the **basic_stringstream** class, see basic_iostream Class.

Reference

Constructors

basic_stringstream	Constructs an object of type **basic_stringstream**.

Typedefs

allocator_type	The type is a synonym for the template parameter *Alloc*.

Member Functions

rdbuf	Returns the address of the stored stream buffer of type **pointer** to basic_stringbuf<*Elem*, *Tr*, *Alloc*>.
str	Sets or gets the text in a string buffer without changing the write position.

See Also

Reference

basic_stringstream Class
basic_iostream Class
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

basic_stringstream Typedefs

For information about the typedefs in the basic_filebuf class, see basic_stringbuf Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_stringstream::allocator_type

The type is a synonym for the template parameter *Alloc*.

```
typedef Alloc allocator_type;
```

Requirements

Header: <sstream>

See Also

Reference

basic_stringstream Class

iostream Programming

iostreams Conventions

Other Resources

basic_stringstream Members

basic_stringstream Member Functions

For information about the member functions in the basic_stringstream class, see basic_stringstream Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_stringstream::basic_stringstream

Constructs an object of type **basic_stringstream**.

```
explicit basic_stringstream(
   ios_base::openmode _Mode = ios_base::in | ios_base::out
);
explicit basic_stringstream(
   const basic_string<Elem, Tr, Alloc>& _Str,
   ios_base::openmode _Mode = ios_base::in | ios_base::out
);
```

Parameters

_Mode

One of the enumerations in ios_base::openmode.

_Str

An object of type **basic_string**.

Remarks

The first constructor initializes the base class by calling basic_iostream(**sb**), where **sb** is the stored object of class basic_stringbuf<**Elem**, **Tr**, **Alloc**>. It also initializes **sb** by calling basic_stringbuf<**Elem**, **Tr**, **Alloc**>(*_Mode*).

The second constructor initializes the base class by calling basic_iostream(**sb**). It also initializes **sb** by calling basic_stringbuf<**Elem**, **Tr**, **Alloc**>(*_Str*, *_Mode*).

Requirements

Header: <sstream>

See Also

Reference

basic_stringstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_stringstream Members

basic_stringstream::rdbuf

Returns the address of the stored stream buffer of type **pointer** to basic_stringbuf<**Elem**, **Tr**, **Alloc**>.

```
basic_stringbuf<Elem, Tr, Alloc> *rdbuf( ) const
```

Return Value

The address of the stored stream buffer of type **pointer** to basic_stringbuf<**Elem**, **Tr**, **Alloc**>.

Example

See basic_filebuf::close for an example that uses **rdbuf**.

Requirements

Header: <sstream>

See Also

Reference

basic_stringstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_stringstream Members

basic_stringstream::str

Sets or gets the text in a string buffer without changing the write position.

```
basic_string<Elem, Tr, Alloc> str( ) const;
void str(
   basic_string<Elem, Tr, Alloc>& _Newstr
);
```

Parameters

_Newstr

The new string.

Return Value

Returns an object of class basic_string<**Elem**, **Tr**, **Alloc**>, whose controlled sequence is a copy of the sequence controlled by ***this**.

Remarks

The first member function returns rdbuf -> str. The second member function calls **rdbuf** -> **str**(*_Newstr*).

Example

See basic_stringbuf::str for an example that uses **str**.

Requirements

Header: <sstream>

See Also

Reference

basic_stringstream Class
iostream Programming
iostreams Conventions

Other Resources

basic_stringstream Members

<stack>

Defines the template class stack and two supporting templates.

For a list of all members of this header, see <stack> Members.

```
#include <stack>
```

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<stack> Members

Standard C++ Library Header Files

<stack> Members

Reference

Operators

operator!=	Tests if the stack object on the left side of the operator is not equal to the stack object on the right side.
operator<	Tests if the stack object on the left side of the operator is less than the stack object on the right side.
operator<=	Tests if the stack object on the left side of the operator is less than or equal to the stack object on the right side.
operator==	Tests if the stack object on the left side of the operator is equal to the stack object on the right side.
operator>	Tests if the stack object on the left side of the operator is greater than the stack object on the right side.
operator>=	Tests if the stack object on the left side of the operator is greater than or equal to the stack object on the right side.

Classes

stack Class	A template container adaptor class that provides a restriction of functionality limiting access to the element most recently added to some underlying container type.

See Also

Reference

<stack>
Thread Safety in the Standard C++ Library
Standard Template Library

<stack> Operators

For more information about the operators in <stack>, see <stack> Members.

See Also

Reference

Standard Template Library

operator!= (<stack>)

Tests if the stack object on the left side of the operator is not equal to stack object on the right side.

```
bool operator!=(
   const stack <Type, Container>& _Left,
   const stack <Type, Container>& _Right,
);
```

Parameters

_Left

An object of type **stack**.

_Right

An object of type **stack**.

Return Value

true if the stacks or stacks are not equal; **false** if stacks or stacks are equal.

Remarks

The comparison between stacks objects is based on a pairwise comparison of their elements. Two stacks are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// stack_op_me.cpp
// compile with: /EHsc
#include <stack>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares stacks with vector base containers
   stack <int, vector<int> > s1, s2, s3;

   // The following would have cause an error because stacks with
   // different base containers are not equality comparable
   // stack <int, list<int> >  s3;

   s1.push( 1 );
   s2.push( 2 );
   s3.push( 1 );

   if ( s1 != s2 )
      cout << "The stacks s1 and s2 are not equal." << endl;
   else
      cout << "The stacks s1 and s2 are equal." << endl;

   if ( s1 != s3 )
      cout << "The stacks s1 and s3 are not equal." << endl;
   else
      cout << "The stacks s1 and s3 are equal." << endl;
}
```

Output

```
The stacks s1 and s2 are not equal.
The stacks s1 and s3 are equal.
```

Requirements

Header: <stack>

See Also

Reference

Standard Template Library

Other Resources

<stack> Members

operator< (<stack>)

Tests if the stack object on the left side of the operator is less than the stack object on the right side.

```
bool operator<(
   const stack <Type, Container>& _Left,
   const stack <Type, Container>& _Right
);
```

Parameters

_Left

An object of type **stack**.

_Right

An object of type **stack**.

Return Value

true if the stack on the left side of the operator is less than and not equal to the stack on the right side of the operator; otherwise **false**.

Remarks

The comparison between stack objects is based on a pairwise comparison of their elements. The less-than relationship between two stack objects is based on a comparison of the first pair of unequal elements.

Example

```
// stack_op_lt.cpp
// compile with: /EHsc
#include <stack>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares stacks with list base container
   stack <int, list<int> > s1, s2, s3;

   s1.push( 2 );
   s1.push( 4 );
   s1.push( 6 );
   s1.push( 8 );
   s2.push( 5 );
   s2.push( 10 );
   s3.push( 2 );
   s3.push( 4 );
   s3.push( 6 );
   s3.push( 8 );

   if ( s1 >= s2 )
      cout << "The stack s1 is greater than or equal to "
           << "the stack s2." << endl;
   else
      cout << "The stack s1 is less than "
           << "the stack s2." << endl;

   if ( s1>= s3 )
      cout << "The stack s1 is greater than or equal to "
           << "the stack s3." << endl;
   else
      cout << "The stack s1 is less than "
```

```
            << "the stack s3." << endl;

    // to print out the stack s1 ( by unstacking the elements):
    stack <int>::size_type i_size_s1 = s1.size( );
    cout << "The stack s1 from the top down is: ( ";
    unsigned int i;
    for ( i = 1 ; i <= i_size_s1 ; i++ )
    {
       cout << s1.top( ) << " ";
       s1.pop( );
    }
    cout << "))." << endl;
  }
```

Output

```
The stack s1 is less than the stack s2.
The stack s1 is greater than or equal to the stack s3.
The stack s1 from the top down is: ( 8 6 4 2 ).
```

Requirements

Header: <stack>

See Also

Reference

Standard Template Library

Other Resources

<stack> Members

operator<= (<stack>)

Tests if the stack object on the left side of the operator is less than or equal to the stack object on the right side.

```
bool operator<=(
   const stack <Type, Container>& _Left,
   const stack <Type, Container>& _Right
);
```

Parameters

_Left

An object of type **stack**.

_Right

An object of type **stack**.

Return Value

true if the stack on the left side of the operator is less than or equal to the stack on the right side of the operator; otherwise **false**.

Remarks

The comparison between stack objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two stack objects is based on a comparison of the first pair of unequal elements.

Example

```
// stack_op_le.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares stacks with default deque base container
   stack <int> s1, s2, s3;

   s1.push( 5 );
   s1.push( 10 );
   s2.push( 1 );
   s2.push( 2 );
   s3.push( 5 );
   s3.push( 10 );

   if ( s1 <= s2 )
      cout << "The stack s1 is less than or equal to "
           << "the stack s2." << endl;
   else
      cout << "The stack s1 is greater than "
           << "the stack s2." << endl;

   if ( s1 <= s3 )
      cout << "The stack s1 is less than or equal to "
           << "the stack s3." << endl;
   else
      cout << "The stack s1 is greater than "
           << "the stack s3." << endl;
}
```

Output

```
The stack s1 is greater than the stack s2.
The stack s1 is less than or equal to the stack s3.
```

Requirements

Header: <stack>

See Also

Reference

Standard Template Library

Other Resources

<stack> Members

operator== (<stack>)

Tests if the stack object on the left side of the operator is equal to stack object on the right side.

```
bool operator==(
   const stack <Type, Container>& _Left,
   const stack <Type, Container>& _Right
);
```

Parameters

_Left

An object of type **stack**.

_Right

An object of type **stack**.

Return Value

true if the stacks or stacks are equal; **false** if stacks or stacks are not equal.

Remarks

The comparison between stack objects is based on a pairwise comparison of their elements. Two stacks are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// stack_op_eq.cpp
// compile with: /EHsc
#include <stack>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares stacks with vector base containers
   stack <int, vector<int> > s1, s2, s3;

   // The following would have cause an error because stacks with
   // different base containers are not equality comparable
   // stack <int, list<int> > s3;

   s1.push( 1 );
   s2.push( 2 );
   s3.push( 1 );

   if ( s1 == s2 )
      cout << "The stacks s1 and s2 are equal." << endl;
   else
      cout << "The stacks s1 and s2 are not equal." << endl;

   if ( s1 == s3 )
      cout << "The stacks s1 and s3 are equal." << endl;
   else
      cout << "The stacks s1 and s3 are not equal." << endl;
}
```

Output

```
The stacks s1 and s2 are not equal.
The stacks s1 and s3 are equal.
```

Requirements

Header: <stack>

See Also

Reference

Standard Template Library

Other Resources

<stack> Members

operator> (<stack>)

Tests if the stack object on the left side of the operator is greater than the stack object on the right side.

```
bool operator>(
   const stack <Type, Container>& _Left,
   const stack <Type, Container>& _Right
);
```

Parameters

_Left

An object of type **stack**.

_Right

An object of type **stack**.

Return Value

true if the stack on the left side of the operator is greater than and not equal to the stack on the right side of the operator; otherwise **false**.

Remarks

The comparison between stack objects is based on a pairwise comparison of their elements. The greater-than relationship between two stack objects is based on a comparison of the first pair of unequal elements.

Example

```
// stack_op_gt.cpp
// compile with: /EHsc
#include <stack>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares stacks with list base container
   stack <int, list<int> > s1, s2, s3;

   s1.push( 1 );
   s1.push( 2 );
   s1.push( 3 );
   s2.push( 5 );
   s2.push( 10 );
   s3.push( 1 );
   s3.push( 2 );

   if ( s1 > s2 )
      cout << "The stack s1 is greater than "
           << "the stack s2." << endl;
   else
      cout << "The stack s1 is not greater than "
           << "the stack s2." << endl;

   if ( s1> s3 )
      cout << "The stack s1 is greater than "
           << "the stack s3." << endl;
   else
      cout << "The stack s1 is not greater than "
           << "the stack s3." << endl;
}
```

Output

```
The stack s1 is not greater than the stack s2.
The stack s1 is greater than the stack s3.
```

Requirements

Header: <stack>

See Also

Reference

Standard Template Library

Other Resources

<stack> Members

operator>= (<stack>)

Tests if the stack object on the left side of the operator is greater than or equal to the stack object on the right side.

```
bool operator>=(
   const stack <Type, Container>& _Left,
   const stack <Type, Container>& _Right
);
```

Parameters

_Left

An object of type **stack**.

_Right

An object of type **stack**.

Return Value

true if the stack on the left side of the operator is strictly less than the stack on the right side of the operator; otherwise **false**.

Remarks

The comparison between stack objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two stack objects is based on a comparison of the first pair of unequal elements.

Example

```
// stack_op_ge.cpp
// compile with: /EHsc
#include <stack>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares stacks with list base container
   stack <int, list<int> > s1, s2, s3;

   s1.push( 1 );
   s1.push( 2 );
   s2.push( 5 );
   s2.push( 10 );
   s3.push( 1 );
   s3.push( 2 );

   if ( s1 >= s2 )
      cout << "The stack s1 is greater than or equal to "
           << "the stack s2." << endl;
   else
      cout << "The stack s1 is less than "
           << "the stack s2." << endl;

   if ( s1>= s3 )
      cout << "The stack s1 is greater than or equal to "
           << "the stack s3." << endl;
   else
      cout << "The stack s1 is less than "
           << "the stack s3." << endl;
}
```

Output

```
The stack s1 is less than the stack s2.
The stack s1 is greater than or equal to the stack s3.
```

Requirements

Header: <stack>

See Also

Reference

Standard Template Library

Other Resources

<stack> Members

<stack> Classes

For more information about the classes in <stack>, see <stack> Members.

See Also

Reference

Standard Template Library

stack Class

A template container adaptor class that provides a restriction of functionality limiting access to the element most recently added to some underlying container type. The stack class is used when it is important to be clear that only stack operations are being performed on the container.

For a list of all members of this type, see stack Members.

```
template <
   class Type,
   class Container=deque<Type>
>
class stack
```

Parameters

Type

The element data type to be stored in the stack.

Container

The type of the underlying container used to implement the stack. The default value is the class **deque**<*Type*>.

Remarks

The elements of class **Type** stipulated in the first template parameter of a stack object are synonymous with value_type and must match the type of element in the underlying container class **Container** stipulated by the second template parameter. The **Type** must be assignable, so that it is possible to copy objects of that type and to assign values to variables of that type.

Suitable underlying container classes for stack include deque, list, and vector, or any other sequence container that supports the operations of **back**, **push_back**, and **pop_back**. The underlying container class is encapsulated within the container adaptor, which exposes only the limited set of the sequence container member functions as a public interface.

The stack objects are equality comparable if and only if the elements of class **Type** are equality comparable and are less-than comparable if and only if the elements of class **Type** are less-than comparable.

- The stack class supports a last-in, first-out (LIFO) data structure. A good analogue to keep in mind would be a stack of plates. Elements (plates) may be inserted, inspected, or removed only from the top of the stack, which is the last element at the end of the base container. The restriction to accessing only the top element is the reason for using the stack class.
- The queue class supports a first-in, first-out (FIFO) data structure. A good analogue to keep in mind would be people lining up for a bank teller. Elements (people) may be added to the back of the line and are removed from the front of the line. Both the front and the back of a line may be inspected. The restriction to accessing only the front and back elements in this way is the reason fur using the queue class.
- The priority_queue class orders its elements so that the largest element is always at the top position. It supports insertion of an element and the inspection and removal of the top element. A good analogue to keep in mind would be people lining up where they are arranged by age, height, or some other criterion.

Requirements

Header: <stack>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

stack Members

<stack> Members

stack Members

Reference

Constructors

stack	Constructs a **stack** that is empty or that is a copy of a base container object.

Typedefs

container_type	A type that provides the base container to be adapted by a **stack**.
size_type	An unsigned integer type that can represent the number of elements in a **stack**.
value_type	A type that represents the type of object stored as an element in a **stack**.

Member Functions

empty	Tests if the **stack** is empty.
pop	Removes the element from the top of the **stack**.
push	Adds an element to the top of the **stack**.
size	Returns the number of elements in the **stack**.
top	Returns a reference to an element at the top of the **stack**.

See Also

Reference

stack Class

Thread Safety in the Standard C++ Library

Standard Template Library

stack Typedefs

For information about the typedefs in the stack class, see stack Members.

See Also

Reference

Standard Template Library

stack::container_type

A type that provides the base container to be adapted.

```
typedef Container container_type;
```

Remarks

The type is a synonym for the template parameter **Container**. All three STL sequence container classes — the vector class, list class, and the default class deque — meet the requirements to be used as the base container for a stack object. User-defined types satisfying these requirements may also be used.

For more information on **Container**, see the Remarks section of the stack Class topic.

Example

See the example for stack::stack for an example of how to declare and use **container_type**.

Requirements

Header: <stack>

See Also

Reference

stack Class

Standard Template Library

Other Resources

stack Members

stack::size_type

An unsigned integer type that can represent the number of elements in a stack.

```
typedef typename Container::size_type size_type;
```

Remarks

The type is a synonym for **size_type** of the base container adapted by the stack.

Example

See the example for size for an example of how to declare and use **size_type**.

Requirements

Header: <stack>

See Also

Reference

stack Class

Standard Template Library

Other Resources

stack Members

stack::value_type

A type that represents the type of object stored as an element in a stack.

```
typedef typename Container::value_type value_type;
```

Remarks

The type is a synonym for **value_type** of the base container adapted by the stack.

Example

```
// stack_value_type.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
   using namespace std;
   // Declares stacks with default deque base container
   stack<int>::value_type AnInt;

   AnInt = 69;
   cout << "The value_type is AnInt = " << AnInt << endl;

   stack<int> s1;
   s1.push( AnInt );
   cout << "The element at the top of the stack is "
        << s1.top( ) << "." << endl;
}
```

Output

```
The value_type is AnInt = 69
The element at the top of the stack is 69.
```

Requirements

Header: <stack>

See Also

Reference

stack Class

Standard Template Library

Other Resources

stack Members

stack Member Functions

For information about the functions in the stack class, see stack Members.

See Also

Reference

Standard Template Library

stack::empty

Tests if a stack is empty.

```
bool empty( ) const;
```

Return Value

true if the stack is empty; **false** if the stack is nonempty.

Example

```
// stack_empty.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
   using namespace std;
   // Declares stacks with default deque base container
   stack <int> s1, s2;

   s1.push( 1 );

   if ( s1.empty( ) )
      cout << "The stack s1 is empty." << endl;
   else
      cout << "The stack s1 is not empty." << endl;

   if ( s2.empty( ) )
      cout << "The stack s2 is empty." << endl;
   else
      cout << "The stack s2 is not empty." << endl;
}
```

Output

```
The stack s1 is not empty.
The stack s2 is empty.
```

Requirements

Header: <stack>

See Also

Reference

stack Class

stack::top and stack::empty

Standard Template Library

Other Resources

stack Members

stack::pop

Removes the element from the top of the stack.

```
void pop( );
```

Remarks

The stack must be nonempty to apply the member function. The top of the stack is the position occupied by the most recently added element and is the last element at the end of the container.

Example

```
// stack_pop.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
   using namespace std;
   stack <int> s1, s2;

   s1.push( 10 );
   s1.push( 20 );
   s1.push( 30 );

   stack <int>::size_type i;
   i = s1.size( );
   cout << "The stack length is " << i << "." << endl;

   i = s1.top( );
   cout << "The element at the top of the stack is "
        << i << "." << endl;

   s1.pop( );

   i = s1.size( );
   cout << "After a pop, the stack length is "
        << i << "." << endl;

   i = s1.top( );
   cout << "After a pop, the element at the top of the stack is "
        << i << "." << endl;
}
```

Output

```
The stack length is 3.
The element at the top of the stack is 30.
After a pop, the stack length is 2.
After a pop, the element at the top of the stack is 20.
```

Requirements

Header: <stack>

See Also

Reference

stack Class

Standard Template Library

Other Resources

stack Members

stack::push

Adds an element to the top end of the stack.

```
void push(
   const Type& _Val
);
```

Parameters

_Val

The element added to the top of the stack.

Remarks

The top of the stack is the position occupied by the most recently added element and is the last element at the end of the container.

Example

```
// stack_push.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
   using namespace std;
   stack <int> s1;

   s1.push( 10 );
   s1.push( 20 );
   s1.push( 30 );

   stack <int>::size_type i;
   i = s1.size( );
   cout << "The stack length is " << i << "." << endl;

   i = s1.top( );
   cout << "The element at the top of the stack is "
        << i << "." << endl;
}
```

Output

```
The stack length is 3.
The element at the top of the stack is 30.
```

Requirements

Header: <stack>

See Also

Reference

stack Class

Standard Template Library

Other Resources

stack Members

stack::size

Returns the number of elements in the stack.

```
size_type size( ) const;
```

Return Value

The current length of the stack.

Example

```
// stack_size.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
   using namespace std;
   stack <int> s1, s2;
   stack <int>::size_type i;

   s1.push( 1 );
   i = s1.size( );
   cout << "The stack length is " << i << "." << endl;

   s1.push( 2 );
   i = s1.size( );
   cout << "The stack length is now " << i << "." << endl;
}
```

Output

```
The stack length is 1.
The stack length is now 2.
```

Requirements

Header: <stack>

See Also

Reference

stack Class

stack::size (STL Samples)

Standard Template Library

Other Resources

stack Members

stack::stack

Constructs a stack that is empty or that is a copy of a base container class.

```
stack( );
explicit stack(
   const container_type& _Right
);
```

Parameters

_Right

The container of which the constructed stack is to be a copy.

Example

```
// stack_stack.cpp
// compile with: /EHsc
#include <stack>
#include <vector>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares stack with default deque base container
   stack <char> dsc1;

   //Explicitly declares a stack with deque base container
   stack <char, deque<char> > dsc2;

   // Declares a stack with vector base containers
   stack <int, vector<int> > vsi1;

   // Declares a stack with list base container
   stack <int, list<int> > lsi;

   // The second member function copies elements from a container
   vector<int> v1;
   v1.push_back( 1 );
   stack <int, vector<int> > vsi2( v1 );
   cout << "The element at the top of stack vsi2 is "
        << vsi2.top( ) << "." << endl;
}
```

Output

```
The element at the top of stack vsi2 is 1.
```

Requirements

Header: <stack>

See Also

Reference

stack Class

Standard Template Library

Other Resources

stack Members

stack::top

Returns a reference to an element at the top of the stack.

```
value_type& top( );
const value_type& top( ) const;
```

Return Value

A reference to the last element in the container at the top of the stack.

Remarks

The stack must be nonempty to apply the member function. The top of the stack is the position occupied by the most recently added element and is the last element at the end of the container.

If the return value of **top** is assigned to a **const_reference**, the stack object cannot be modified. If the return value of **top** is assigned to a **reference**, the stack object can be modified.

Example

```
// stack_top.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
   using namespace std;
   stack <int> s1;

   s1.push( 1 );
   s1.push( 2 );

   int& i = s1.top( );
   const int& ii = s1.top( );

   cout << "The top integer of the stack s1 is "
        << i << "." << endl;
   i--;
   cout << "The next integer down is "<< ii << "." << endl;
}
```

Output

```
The top integer of the stack s1 is 2.
The next integer down is 1.
```

Requirements

Header: <stack>

See Also

Reference

stack Class

stack::top and stack::empty

Standard Template Library

Other Resources

stack Members

<stdexcept>

Defines several standard classes used for reporting exceptions. The classes form a derivation hierarchy all derived from class exception and include two general types of exceptions: logical errors and run-time errors. The logical errors are caused programmer mistakes. They derive from the base class logic_error and include:

- **domain_error**
- **invalid_argument**
- **length_error**
- **out_of_range**

The run-time errors occur because of mistakes in either the library functions or in the run-time system. They derive from the base class runtime_error and include:

- **overflow_error**
- **range_error**
- **underflow_error**

For a list of all members of this header, see <stdexcept> Members.

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<stdexcept> Members

Standard C++ Library Header Files

<stdexcept> Members

Reference

Classes

domain_error Class	The class serves as the base class for all exceptions thrown to report a domain error.
invalid_argument Class	The class serves as the base class for all exceptions thrown to report an invalid argument.
length_error Class	The class serves as the base class for all exceptions thrown to report an attempt to generate an object too long to be specified.
logic_error Class	The class serves as the base class for all exceptions thrown to report errors presumably detectable before the program executes, such as violations of logical preconditions.
out_of_range Class	The class serves as the base class for all exceptions thrown to report an argument that is out of its valid range.
overflow_error Class	The class serves as the base class for all exceptions thrown to report an arithmetic overflow.
range_error Class	The class serves as the base class for all exceptions thrown to report a range error.
runtime_error Class	The class serves as the base class for all exceptions thrown to report errors presumably detectable only when the program executes.
underflow_error Class	The class serves as the base class for all exceptions thrown to report an arithmetic underflow.

See Also

Reference

<stdexcept>

Thread Safety in the Standard C++ Library

<stdexcept> Classes

For information about the classes in <stdexcept>, see <stdexcept> Members.

domain_error Class

The class serves as the base class for all exceptions thrown to report a domain error.

```
class domain_error : public logic_error {
public:
   domain_error(const string& message);
   };
```

Remarks

The value returned by **what** is a copy of **message**.data.

Example

```
// domain_error.cpp
// compile with: /EHsc /GR
#include <iostream>

using namespace std;

int main( )
{
   try
   {
      throw domain_error( "Your domain is in error!" );
   }
   catch (exception &e)
   {
      cerr << "Caught: " << e.what( ) << endl;
      cerr << "Type: " << typeid(e).name( ) << endl;
   };
}
```

Sample Output

```
Caught: Your domain is in error!
Type: class std::domain_error
```

Requirements

Header: <stdexcept>

See Also

Reference

logic_error Class

Thread Safety in the Standard C++ Library

Other Resources

<stdexcept> Members

invalid_argument Class

The class serves as the base class for all exceptions thrown to report an invalid argument.

```
class invalid_argument : public logic_error {
public:
   invalid_argument(const string& message);
};
```

Remarks

The value returned by **what** is a copy of **message**.data.

Example

```
// invalid_arg.cpp
// compile with: /EHsc /GR
#include <bitset>
#include <iostream>

using namespace std;

int main( )
{
   try
   {
      bitset< 32 > bitset( string( "11001010101100001b100101010110000") );
   }
   catch ( exception &e )
   {
      cerr << "Caught " << e.what( ) << endl;
      cerr << "Type " << typeid( e ).name( ) << endl;
   };
}
```

Output

```
Caught invalid bitset<N> char
Type class std::invalid_argument
```

Requirements

Header: <stdexcept>

See Also

Reference

logic_error Class

Thread Safety in the Standard C++ Library

Other Resources

<stdexcept> Members

length_error Class

The class serves as the base class for all exceptions thrown to report an attempt to generate an object too long to be specified.

```
class length_error : public logic_error {
public:
   length_error(const string& message);
   };
```

Remarks

The value returned by **what** is a copy of **message**.data.

Example

```
// length_error.cpp
// compile with: /EHsc /GR
#include <vector>
#include <iostream>

using namespace std;

template<class _Ty>
class stingyallocator : public allocator<_Ty>
{
public:
   template <class U>
      struct rebind { typedef stingyallocator<U> other; };
   _SIZT max_size( ) const
   {
         return 10;
   };

};

int main( )
{
   try
   {
      vector<int, stingyallocator< int > > myv;
      for ( int i = 0; i < 11; i++ ) myv.push_back( i );
   }
   catch ( exception &e )
   {
      cerr << "Caught " << e.what( ) << endl;
      cerr << "Type " << typeid( e ).name( ) << endl;
   };
}
```

Sample Output

```
Caught vector<T> too long
Type class std::length_error
```

Requirements

Header: <stdexcept>

See Also

Reference

logic_error Class

Thread Safety in the Standard C++ Library

Other Resources

<stdexcept> Members

logic_error Class

The class serves as the base class for all exceptions thrown to report errors presumably detectable before the program executes, such as violations of logical preconditions.

```
class logic_error : public exception {
public:
   logic_error(const string& message);
};
```

Remarks

The value returned by **what** is a copy of **message**.data.

Example

```
// logic_error.cpp
// compile with: /EHsc /GR
#include <iostream>
using namespace std;

int main( )
{
   try
   {
      throw logic_error( "logic error" );
   }
   catch ( exception &e )
   {
      cerr << "Caught: " << e.what( ) << endl;
      cerr << "Type: " << typeid( e ).name( ) << endl;
   };
}
```

Output

```
Caught: logic error
Type: class std::logic_error
```

Requirements

Header: <stdexcept>

See Also

Reference

exception Class

Thread Safety in the Standard C++ Library

Other Resources

<stdexcept> Members

out_of_range Class

The class serves as the base class for all exceptions thrown to report an argument that is out of its valid range.

```
class out_of_range : public logic_error {
public:
   out_of_range(const string& message);
};
```

Remarks

The value returned by **what** is a copy of **message**.data.

Example

```
// out_of_range.cpp
// compile with: /EHsc /GR
#include <string>
#include <iostream>

using namespace std;

int main() {
// out_of_range
   try {
      string str( "Micro" );
      string rstr( "soft" );
      str.append( rstr, 5, 3 );
      cout << str << endl;
   }
   catch ( exception &e ) {
      cerr << "Caught: " << e.what( ) << endl;
      cerr << "Type: " << typeid( e ).name( ) << endl;
   };
}
```

Sample Output

```
Caught: invalid string position
Type: class std::out_of_range
```

Requirements

Header: <stdexcept>

See Also

Reference

logic_error Class

Thread Safety in the Standard C++ Library

Other Resources

<stdexcept> Members

overflow_error Class

The class serves as the base class for all exceptions thrown to report an arithmetic overflow.

```
class overflow_error : public runtime_error {
public:
   overflow_error(const string& message);
};
```

Remarks

The value returned by **what** is a copy of **message**.data.

Example

```
// overflow_error.cpp
// compile with: /EHsc /GR
#include <bitset>
#include <iostream>

using namespace std;

int main( )
{
   try
   {
      bitset< 33 > bitset;
      bitset[32] = 1;
      bitset[0] = 1;
      unsigned long x = bitset.to_ulong( );
   }
   catch ( exception &e )
   {
      cerr << "Caught " << e.what( ) << endl;
      cerr << "Type " << typeid( e ).name( ) << endl;
   };
}
```

Output

```
Caught bitset<N> overflow
Type class std::overflow_error
```

Requirements

Header: <stdexcept>

See Also

Reference

runtime_error Class

Thread Safety in the Standard C++ Library

Other Resources

<stdexcept> Members

range_error Class

The class serves as the base class for all exceptions thrown to report a range error.

```
class range_error : public runtime_error {
public:
   range_error(const string& message);
};
```

Remarks

The value returned by **what** is a copy of **message**.data.

Example

```
// range_error.cpp
// compile with: /EHsc /GR
#include <iostream>
using namespace std;
int main()
{
   try
   {
      throw range_error( "The range is in error!" );
   }
   catch (exception &e)
   {
      cerr << "Caught: " << e.what( ) << endl;
      cerr << "Type: " << typeid( e ).name( ) << endl;
   };
}
```

Sample Output

```
Caught: The range is in error!
Type: class std::range_error
```

Requirements

Header: <stdexcept>

See Also

Reference

runtime_error Class

Thread Safety in the Standard C++ Library

Other Resources

<stdexcept> Members

runtime_error Class

The class serves as the base class for all exceptions thrown to report errors presumably detectable only when the program executes.

```
class runtime_error : public exception {
public:
   runtime_error(const string& message);
};
```

Remarks

The value returned by **what** is a copy of **message**.data.

Example

```
// runtime_error.cpp
// compile with: /EHsc /GR
#include <iostream>

using namespace std;

int main( )
{
// runtime_error
   try
   {
      locale loc( "test" );
   }
   catch ( exception &e )
   {
      cerr << "Caught " << e.what( ) << endl;
      cerr << "Type " << typeid( e ).name( ) << endl;
   };
}
```

Sample Output

```
Caught bad locale name
Type class std::runtime_error
```

Requirements

Header: <stdexcept>

See Also

Reference

exception Class

Thread Safety in the Standard C++ Library

Other Resources

<stdexcept> Members

underflow_error Class

The class serves as the base class for all exceptions thrown to report an arithmetic underflow.

```
class underflow_error : public runtime_error {
public:
   underflow_error(const string& message);
};
```

Remarks

The value returned by **what** is a copy of **message**.data.

Example

```
// underflow_error.cpp
// compile with: /EHsc /GR
#include <iostream>

using namespace std;

int main( )
{
   try
   {
      throw underflow_error( "The number's a bit small, captain!" );
   }
   catch ( exception &e ) {
      cerr << "Caught: " << e.what( ) << endl;
      cerr << "Type: " << typeid( e ).name( ) << endl;
   };
}
```

Sample Output

```
Caught: The number's a bit small, captain!
Type: class std::underflow_error
```

Requirements

Header: <stdexcept>

See Also

Reference

runtime_error Class

Thread Safety in the Standard C++ Library

Other Resources

<stdexcept> Members

<streambuf>

Include the iostreams standard header <streambuf> to define the template class basic_streambuf, which is basic to the operation of the iostreams classes. This header is typically included for you by another of the iostreams headers; you rarely need to include it directly.

For a list of all members of this header, see <streambuf> Members.

```
#include <streambuf>
```

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

<streambuf> Members
Standard C++ Library Header Files

<streambuf> Members

Reference

Typedefs

streambuf	A specialization of **basic_streambuf** that uses **char** as the template parameters.
wstreambuf	A specialization of **basic_streambuf** that uses **wchar_t** as the template parameters.

Classes

basic_streambuf Class	The template class describes an abstract base class for deriving a stream buffer, which controls the transmission of elements to and from a specific representation of a stream.

See Also

Reference

<streambuf>
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

<streambuf> Typedefs

For information about the template functions in <streambuf>, see <streambuf> Members.

See Also

Reference

iostream Programming

iostreams Conventions

streambuf

A specialization of **basic_streambuf** that uses **char** as the template parameters.

```
typedef basic_streambuf<char, char_traits<char> > streambuf;
```

Remarks

The type is a synonym for the template class basic_streambuf, specialized for elements of type **char** with default character traits.

Requirements

Header: <streambuf>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<streambuf> Members

wstreambuf

A specialization of **basic_streambuf** that uses **wchar_t** as the template parameters.

```
typedef basic_streambuf<wchar_t, char_traits<wchar_t> > wstreambuf;
```

Remarks

The type is a synonym for the template class basic_streambuf, specialized for elements of type **wchar_t** with default character traits.

Requirements

Header: <streambuf>

See Also

Reference

iostream Programming

iostreams Conventions

Other Resources

<streambuf> Members

<streambuf> Classes

For information about the classes in <streambuf>, see <streambuf> Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_streambuf Class

Describes an abstract base class for deriving a stream buffer, which controls the transmission of elements to and from a specific representation of a stream.

For a list of all members of this type, see basic_streambuf Members.

```
template<class Elem, class Tr = char_traits<Elem> >
   class basic_streambuf;
```

Parameters

Elem

A char_type.

Tr

The character traits_type.

Remarks

The template class describes an abstract base class for deriving a stream buffer, which controls the transmission of elements to and from a specific representation of a stream. An object of class **basic_streambuf** helps control a stream with elements of type *Tr*, also known as char_type, whose character traits are determined by the class char_traits, also known as traits_type.

Note

Objects of type **basic_streambuf** are created with an internal buffer of type **char *** regardless of the **char_type** specified by the type parameter *Elem*. This means that a Unicode string (containing **wchar_t** characters) will be converted to an ANSI string (containing **char** characters) before it is written to the internal buffer. To store Unicode strings in the buffer, create a new buffer of type **wchar_t** and set it using the basic_streambuf::pubsetbuf**()** method. To see an example that demonstrates this behavior, see basic_filebuf Class.

Every stream buffer conceptually controls two independent streams: one for extractions (input) and one for insertions (output). A specific representation may, however, make either or both of these streams inaccessible. It typically maintains some relationship between the two streams. What you insert into the output stream of a basic_stringbuf<*Elem*, *Tr*> object, for example, is what you later extract from its input stream. When you position one stream of a basic_filebuf<*Elem*, *Tr*> object, you position the other stream in tandem.

The public interface to template class **basic_streambuf** supplies the operations that are common to all stream buffers, however specialized. The protected interface supplies the operations needed for a specific representation of a stream to do its work. The protected virtual member functions let you tailor the behavior of a derived stream buffer for a specific representation of a stream. Each derived stream buffer in this library describes how it specializes the behavior of its protected virtual member functions. The default behavior for the base class, which is often to do nothing, is described in this topic.

The remaining protected member functions control copying to and from any storage supplied to buffer transmissions to and from streams. An input buffer, for example, is characterized by:

- eback, a pointer to the beginning of the buffer.
- gptr, a pointer to the next element to read.
- egptr, a pointer just past the end of the buffer.

Similarly, an output buffer is characterized by:

- pbase, a pointer to the beginning of the buffer.
- pptr, a pointer to the next element to write.
- epptr, a pointer just past the end of the buffer.

For any buffer, the following protocol is used:

- If the next pointer is null, no buffer exists. Otherwise, all three pointers point into the same sequence. They can be safely compared for order.
- For an output buffer, if the next pointer compares less than the end pointer, you can store an element at the write position designated by the next pointer.
- For an input buffer, if the next pointer compares less than the end pointer, you can read an element at the read position designated by the next pointer.
- For an input buffer, if the beginning pointer compares less than the next pointer, you can put back an element at the putback position designated by the decremented next pointer.

Any protected virtual member functions you write for a class derived from **basic_streambuf**<*Elem*, *Tr*> must cooperate in maintaining this protocol.

An object of class **basic_streambuf**<*Elem*, *Tr*> stores the six pointers previously described. It also stores a locale object in an object of type locale for potential use by a derived stream buffer.

Requirements

Header: <streambuf>

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members
<streambuf> Members

basic_streambuf Members

Reference

Constructors

basic_streambuf	Constructs an object of type **basic_streambuf**.

Typedefs

char_type	Associates a type name with the *Elem* template parameter.
int_type	Associates a type name within **basic_streambuf** scope with the *Elem* template parameter.
off_type	Associates a type name within **basic_streambuf** scope with the *Elem* template parameter.
pos_type	Associates a type name within **basic_streambuf** scope with the *Elem* template parameter.
traits_type	Associates a type name with the *Tr* template parameter.

Member Functions

eback	A protected function that returns a pointer to the beginning of the input buffer.
egptr	A protected function that returns a pointer just past the end of the input buffer.
epptr	A protected function that returns a pointer just past the end of the output buffer.
gbump	A protected function that adds _*Count* to the next pointer for the input buffer.
getloc	Gets the **basic_streambuf** object's locale.
gptr	A protected function that returns a pointer to the next element of the input buffer.
imbue	A protected, virtual function called by pubimbue.
in_avail	Returns the number of elements that are ready to be read from the buffer.
overflow	A protected virtual function that can be called when a new character is inserted into a full buffer.
pbackfail	A protected virtual member function that tries to put back an element into the input stream, then make it the curr ent element (pointed to by the next pointer).
pbase	A protected function that returns a pointer to the beginning of the output buffer.
pbump	A protected function that adds `count` to the next pointer for the output buffer.
pptr	A protected function that returns a pointer to the next element of the output buffer.
pubimbue	Sets the **basic_streambuf** object's locale.
pubseekoff	Calls seekoff, a protected virtual function that is overridden in a derived class.
pubseekpos	Calls seekpos, a protected virtual function that is overridden in a derived class and resets the current pointer positi on.

pubsetbuf	Calls setbuf, a protected virtual function that is overridden in a derived class.
pubsync	Calls sync, a protected virtual function that is overridden in a derived class and updates the external stream associated with this buffer.
sbumpc	Reads and returns the current element, moving the stream pointer.
seekoff	The protected virtual member function tries to alter the current positions for the controlled streams.
seekpos	The protected virtual member function tries to alter the current positions for the controlled streams.
setbuf	The protected virtual member function performs an operation particular to each derived stream buffer.
setg	A protected function that stores *_Gbeg* in the beginning pointer, *_Gnext* in the next pointer, and *_Gend* in the end pointer for the input buffer.
setp	A protected function that stores *_Pbeg* in the beginning pointer and *_Pend* in the end pointer for the output buffer.
sgetc	Returns current element without changing position in the stream.
sgetn	Returns the number of elements read. Deprecated. Use basic_streambuf::_Sgetn_s instead.
_Sgetn_s	Returns the number of elements read.
showmanyc	Protected virtual member function that returns a count of the number of characters that can be extracted from the input stream and ensure that the program will not be subject to an indefinite wait.
snextc	Reads the current element and returns the following element.
sputbackc	Puts a **char_type** in the stream.
sputc	Puts a character into the stream.
sputn	Puts a character string into the stream.
stossc	Move past the current element in the stream.
sungetc	Gets a character from the stream.
sync	A protected virtual function that tries to synchronize the controlled streams with any associated external streams.
uflow	A protected virtual function that extracts the current element from the input stream.
underflow	A protected virtual function that extracts the current element from the input stream.
xsgetn	A protected virtual function that extracts elements from the input stream. Deprecated. Use basic_streambuf::_Xsgetn_s instead.
_Xsgetn_s	A protected virtual function that extracts elements from the input stream.
xsputn	A protected virtual function that inserts elements into the output stream.

See Also

Reference

basic_streambuf Class
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

basic_streambuf Typedefs

For information about the typedefs in the basic_streambuf class, see basic_streambuf Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_streambuf::char_type

Associates a type name with the **Elem** template parameter.

```
typedef Elem char_type;
```

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::int_type

Associates a type name within basic_streambuf scope with one of the types in a template parameter.

```
typedef typename traits_type::int_type int_type;
```

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::off_type

Associates a type name within basic_streambuf scope with one of the types in a template parameter.

```
typedef typename traits_type::off_type off_type;
```

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::pos_type

Associates a type name within basic_streambuf scope with one of the types in a template parameter.

```
typedef typename traits_type::pos_type pos_type;
```

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::traits_type

Associates a type name with the **Tr** template parameter.

```
typedef Tr traits_type;
```

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf Member Functions

For information about the member functions in the basic_streambuf class, see basic_streambuf Members.

See Also

Reference

iostream Programming

iostreams Conventions

basic_streambuf::basic_streambuf

Constructs an object of type **basic_streambuf**.

```
basic_streambuf( );
```

Remarks

The protected constructor stores a null pointer in all pointers controlling the input buffer and the output buffer. It also stores locale::classic in the locale object.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::eback

A protected function that returns a pointer to the beginning of the input buffer.

```
char_type *eback( ) const;
```

Return Value

A pointer to the beginning of the input buffer.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::egptr

A protected function that returns a pointer just past the end of the input buffer.

```
char_type *egptr( ) const;
```

Return Value

A pointer just past the end of the input buffer.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::epptr

A protected function that returns a pointer just past the end of the output buffer.

```
char_type *epptr( ) const;
```

Return Value

A pointer just past the end of the output buffer.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::gbump

A protected function that adds *_Count* to the next pointer for the input buffer.

```
void gbump(
   int _Count
);
```

Parameters

_Count

The amount by which to advance the pointer.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class

iostream Programming

iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::getloc

Gets the basic_streambuf object's locale.

```
locale getloc( ) const;
```

Return Value

The stored locale object.

Remarks

For related information, see ios_base::getloc.

Example

```
// basic_streambuf_getloc.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   cout << cout.rdbuf( )->getloc( ).name( ).c_str( ) << endl;
}
```

Output

```
C
```

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class

iostream Programming

iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::gptr

A protected function that returns a pointer to the next element of the input buffer.

```
char_type *gptr( ) const;
```

Return Value

A pointer to the next element of the input buffer.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::imbue

A protected virtual function called by pubimbue.

```
virtual void imbue(
   const locale &_Loc
);
```

Parameters

_Loc

A reference to a locale.

Remarks

The default behavior is to do nothing.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::in_avail

Returns the number of elements that are ready to be read from the buffer.

```
streamsize in_avail( );
```

Return Value

The number of elements that are ready to be read from the buffer.

Remarks

If a read position is available, the member function returns egptr – gptr. Otherwise, it returns showmanyc.

Example

```
// basic_streambuf_in_avail.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   char c;
   // cin's buffer is empty, in_avail will return 0
   cout << cin.rdbuf( )->in_avail( ) << endl;
   cin >> c;
   cout << cin.rdbuf( )->in_avail( ) << endl;
}
```

Input

```
4
```

Sample Output

```
0
4
1
```

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::overflow

A protected virtual function that can be called when a new character is inserted into a full buffer.

```
virtual int_type overflow(
   int_type _Meta = traits_type::eof( )
);
```

Parameters

_Meta

The character to insert into the buffer, or **traits_type::**eof.

Return Value

If the function cannot succeed, it returns **traits_type::eof** or throws an exception. Otherwise, it returns **traits_type::**not_eof(*_Meta*). The default behavior is to return **traits_type::eof**.

Remarks

If *_Meta* does not compare equal to **traits_type::eof**, the protected virtual member function endeavors to insert the element **traits_type::**to_char_type(*_Meta*) into the output stream. It can do so in various ways:

- If a **write position** is available, it can store the element into the write position and increment the next pointer for the output buffer.
- It can make a write position available by allocating new or additional storage for the output buffer.
- It can make a write position available by writing out, to some external destination, some or all of the elements between the beginning and next pointers for the output buffer.

The virtual overflow function, together with the sync and underflow functions, defines the characteristics of the streambuf-derived class. Each derived class might implement overflow differently, but the interface with the calling stream class is the same.

The **overflow** function is most frequently called by public **streambuf** functions like **sputc** and **sputn** when the put area is full, but other classes, including the stream classes, can call **overflow** anytime.

The function consumes the characters in the put area between the **pbase** and **pptr** pointers and then reinitializes the put area. The **overflow** function must also consume *nCh* (if *nCh* is not **EOF**), or it might choose to put that character in the new put area so that it will be consumed on the next call.

The definition of consume varies among derived classes. For example, the **filebuf** class writes its characters to a file, while the **strstreambuf** class keeps them in its buffer and (if the buffer is designated as dynamic) expands the buffer in response to a call to overflow. This expansion is achieved by freeing the old buffer and replacing it with a new, larger one. The pointers are adjusted as necessary.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::pbackfail

A protected virtual member function that tries to put back an element into the input stream, then make it the current element (pointed to by the next pointer).

```
virtual int_type pbackfail(
      int_type _Meta = traits_type::eof( )
);
```

Parameters

_Meta

The character to insert into the buffer, or **traits_type::**eof.

Return Value

If the function cannot succeed, it returns **traits_type::eof** or throws an exception. Otherwise, it returns some other value. The default behavior is to return **traits_type::eof**.

Remarks

If *_Meta* compares equal to **traits_type::eof**, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by **traits_type::**to_char_type(*_Meta*). The function can put back an element in various ways:

- If a putback position is available, it can store the element into the putback position and decrement the next pointer for the input buffer.
- It can make a putback position available by allocating new or additional storage for the input buffer.
- For a stream buffer with common input and output streams, it can make a putback position available by writing out, to some external destination, some or all of the elements between the beginning and next pointers for the output buffer.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::pbase

A protected function that returns a pointer to the beginning of the output buffer.

```
char_type *pbase( ) const;
```

Return Value

A pointer to the beginning of the output buffer.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::pbump

A protected function that adds *_Count* to the next pointer for the output buffer.

```
void pbump(
   int _Count
);
```

Parameters

_Count

The number of characters by which to move the write position forward.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class

iostream Programming

iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::pptr

A protected function that returns a pointer to the next element of the output buffer.

```
char_type *pptr( ) const;
```

Return Value

A pointer to the next element of the output buffer.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::pubimbue

Sets the basic_streambuf object's locale.

```
locale pubimbue(
   const locale& _Loc
);
```

Parameters

_Loc

A reference to a locale.

Return Value

The previous value stored in the locale object.

Remarks

The member function stores *_Loc* in the locale object and calls imbue.

Example

See basic_ios::imbue for an example that uses **pubimbue**.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::pubseekoff

Calls seekoff, a protected virtual function that is overridden in a derived class.

```
pos_type pubseekoff(
   off_type _Off,
   ios_base::seekdir _Way,
   ios_base::openmode _Which = ios_base::in | ios_base::out
);
```

Parameters

_Off

The position to seek for relative to *_Way*.

_Way

The starting point for offset operations. See seekdir for possible values.

_Which

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

Returns the new position or an invalid stream position (seekoff(*_Off*, *_Way*, *_Which*)).

Remarks

Moves the pointer relative to *_Way*.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::pubseekpos

Calls seekpos, a protected virtual function that is overridden in a derived class, and resets the current pointer position.

```
pos_type pubseekpos(
   pos_type _Sp,
   ios_base::openmode _Which = ios_base::in | ios_base::out
);
```

Parameters

_Sp

The position to seek for.

_Which

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

The new position or an invalid stream position. To determine if the stream position is invalid, compare the return value with `pos_type(off_type(-1))`.

Remarks

The member function returns seekpos(*_Sp*, *_Which*).

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::pubsetbuf

Calls setbuf, a protected virtual function that is overridden in a derived class.

```
basic_streambuf *pubsetbuf(
   char_type *_Buffer,
   streamsize _Count
);
```

Parameters

_Buffer

A pointer to **char_type** for this instantiation.

_Count

The size of the buffer.

Return Value

Returns setbuf(*_Buffer*, *_Count*).

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::pubsync

Calls sync, a protected virtual function that is overridden in a derived class, and updates the external stream associated with this buffer.

```
int pubsync( );
```

Return Value

Returns sync or –1 if failure.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::sbumpc

Reads and returns the current element, moving the stream pointer.

```
int_type sbumpc( );
```

Return Value

The current element.

Remarks

If a read position is available, the member function returns **traits_type::**to_int_type(***gptr**) and increments the next pointer for the input buffer. Otherwise, it returns uflow.

Example

```
// basic_streambuf_sbumpc.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   int i = 0;
   i = cin.rdbuf( )->sbumpc( );
   cout << i << endl;
}
```

Input

```
3
```

Sample Output

```
3
51
```

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::seekoff

A protected virtual member function that tries to alter the current positions for the controlled streams.

```
virtual pos_type seekoff(
   off_type _Off,
   ios_base::seekzdir _Way,
   ios_base::openmode _Which = ios_base::in | ios_base::out
);
```

Parameters

_Off

The position to seek for relative to *_Way*.

_Way

The starting point for offset operations. See seekdir for possible values.

_Which

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

Returns the new position or an invalid stream position (**seekoff** (*_Off*, *_Way*, *_Which*)).

Remarks

The new position is determined as follows:

- If *_Way* == **ios_base::beg**, the new position is the beginning of the stream plus *_Off*.
- If *_Way* == **ios_base::cur**, the new position is the current stream position plus *_Off*.
- If *_Way* == **ios_base::end**, the new position is the end of the stream plus *_Off*.

Typically, if **which & ios_base::in** is nonzero, the input stream is affected, and if **which & ios_base::out** is nonzero, the output stream is affected. Actual use of this parameter varies among derived stream buffers, however.

If the function succeeds in altering the stream position or positions, it returns the resulting stream position or one of the resulting stream positions. Otherwise, it returns an invalid stream position. The default behavior is to return an invalid stream position.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::seekpos

A protected virtual member function that tries to alter the current positions for the controlled streams.

```
virtual pos_type seekpos(
   pos_type _Sp,
   ios_base::openmode _Which = ios_base::in | ios_base::out
);
```

Parameters

_Sp

The position to seek for.

_Which

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

The new position, or an invalid stream position. To determine if the stream position is invalid, compare the return value with `pos_type(off_type(-1))`.

Remarks

The new position is *_Sp*.

Typically, if **which & ios_base::in** is nonzero, the input stream is affected, and if **which & ios_base::out** is nonzero, the output stream is affected. Actual use of this parameter varies among derived stream buffers, however.

If the function succeeds in altering the stream position or positions, it returns the resulting stream position or one of the resulting stream positions. Otherwise, it returns an invalid stream position (-1). The default behavior is to return an invalid stream position.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::setbuf

A protected virtual member function that performs an operation particular to each derived stream buffer.

```
virtual basic_streambuf *setbuf(
   char_type *_Buffer,
   streamsize _Count
);
```

Parameters

_Buffer

Pointer to a buffer.

_Count

Size of the buffer.

Return Value

The default behavior is to return **this**.

Remarks

See basic_filebuf. **setbuf** provides an area of memory for the **streambuf** object to use. How the buffer is used in defined in the derived classes.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::setg

A protected function that stores *_Gbeg* in the beginning pointer, *_Gnext* in the next pointer, and *_Gend* in the end pointer for the input buffer.

```
void setg(
   char_type *_Gbeg,
   char_type *_Gnext,
   char_type *_Gend
);
```

Parameters

_Gbeg

A pointer to the beginning of the buffer.

_Gnext

A pointer to somewhere in the middle of the buffer.

_Gend

A pointer to the end of the buffer.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::setp

A protected function that stores *_Pbeg* in the beginning pointer and *_Pend* in the end pointer for the output buffer.

```
void setp(
   char_type *_Pbeg,
   char_type *_Pend
);
```

Parameters

_Pbeg

A pointer to the beginning of the buffer.

_Pend

A pointer to the end of the buffer.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::sgetc

Returns current element without changing position in the stream.

```
int_type sgetc( );
```

Return Value

The current element.

Remarks

If a read position is available, the member function returns **traits_type::**to_int_type(*gptr). Otherwise, it returns underflow.

Example

```
// basic_streambuf_sgetc.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
   using namespace std;
   ifstream myfile( "basic_streambuf_sgetc.txt", ios::in );

   char i = myfile.rdbuf( )->sgetc( );
   cout << i << endl;
   i = myfile.rdbuf( )->sgetc( );
   cout << i << endl;
}
```

Input: basic_streambuf_sgetc.txt

```
testing
```

Output

```
t
t
```

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::sgetn

Extracts up to *_Count* characters from the input buffer and stores them in the provided buffer *_Ptr*.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct. Consider using basic_streambuf::_Sgetn_s instead.

```
streamsize sgetn(
   char_type *_Ptr,
   streamsize _Count
);
```

Parameters

_Ptr

The buffer to contain the extracted characters.

_Count

The number of elements to read.

Return Value

The number of elements read.

Remarks

The member function returns xsgetn(*_Ptr*, *_Count*).

Example

```
// basic_streambuf_sgetn.cpp
// compile with: /EHsc /W3
#include <iostream>
#include <fstream>

int main()
{
    using namespace std;

    ifstream myfile("basic_streambuf_sgetn.txt", ios::in);
    char a[10];

    // Extract 3 characters from myfile and store them in a.
    // Note: basic_streambuf::sgetn is potentially unsafe, consider
    // using basic_streambuf::_Sgetn_s instead.
    streamsize i = myfile.rdbuf()->sgetn(&a[0], 3);  // C4996
    a[i] = myfile.widen('\0');

    // Display the size and contents of the buffer passed to sgetn.
    cout << i << " " << a << endl;

    // Display the contents of the original input buffer.
    cout << myfile.rdbuf() << endl;
}
```

Input: basic_streambuf_sgetn.txt

```
testing
```

Output

```
3 tes
ting
```

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class

iostream Programming

iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::_Sgetn_s

Extracts up to *_Count* characters from the input buffer and stores them in the provided buffer *_Ptr*.

```
streamsize _Sgetn_s(
    char_type *_Ptr,
    size_t _Ptr_size,
    streamsize _Count
);
```

Parameters

_Ptr

The buffer to contain the extracted characters.

_Ptr_size

The size of *_Ptr*.

_Count

The number of elements to read.

Return Value

The number of elements read.

Remarks

The member function returns basic_streambuf::_Xsgetn_s(*_Ptr*, *_Ptr_size*, *_Count*).

Example

```
// basic_streambuf__Sgetn_s.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main()
{
    using namespace std;

    ifstream myfile("basic_streambuf__Sgetn_s.txt", ios::in);
    const int BUFSIZE = 10;
    char a[BUFSIZE];

    // Extract 3 characters from myfile and store them in a.
    streamsize i = myfile.rdbuf()->_Sgetn_s(&a[0], BUFSIZE, 3);
    a[i] = myfile.widen('\0');

    // Display the size and contents of the buffer passed to sgetn.
    cout << i << " " << a << endl;

    // Display the contents of the original input buffer.
    cout << myfile.rdbuf() << endl;
}
```

Input: basic_streambuf__Sgetn_s.txt

```
testing
```

Output

```
3 tes
ting
```

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions
Safe Libraries: Standard C++ Library

Other Resources

basic_streambuf Members

basic_streambuf::showmanyc

A protected virtual member function that returns a count of the number of characters that can be extracted from the input stream and ensure that the program will not be subject to an indefinite wait.

```
virtual streamsize showmanyc( );
```

Return Value

The default behavior is to return zero.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::snextc

Reads the current element and returns the following element.

```
int_type snextc( );
```

Return Value

The next element in the stream.

Remarks

The member function calls sbumpc and, if that function returns **traits_type::**eof, returns **traits_type::eof**. Otherwise, it returns sgetc.

Example

```
// basic_streambuf_snextc.cpp
// compile with: /EHsc
#include <iostream>
int main( )
{
   using namespace std;
   int i = 0;
   i = cin.rdbuf( )->snextc( );
   // cout << ( int )char_traits<char>::eof << endl;
   cout << i << endl;
}
```

Input

```
aa
```

Sample Output

```
97
```

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::sputbackc

Puts a char_type in the stream.

```
int_type sputbackc(
   char_type _Ch
);
```

Parameters

_Ch

The character.

Return Value

Returns the character or failure.

Remarks

If a putback position is available and *_Ch* compares equal to the character stored in that position, the member function decrements the next pointer for the input buffer and returns **traits_type::**to_int_type(*_Ch*). Otherwise, it returns pbackfail(*_Ch*).

Example

```
// basic_streambuf_sputbackc.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
   using namespace std;

   ifstream myfile( "iotest.txt", ios::in );

   int i = myfile.rdbuf( )->sgetc( );
   cout << ( char )i << endl;
   int j = myfile.rdbuf( )->sputbackc( 'z' );
   if ( j == 'z' )
   {
      cout << "it worked" << endl;
   }
   i = myfile.rdbuf( )->sgetc( );
   cout << ( char )i << endl;
}
```

Input: basic_streambuf_sputbackc.txt

```
testing
```

Sample Output

```
t
it worked
z
```

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class

iostream Programming

iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::sputc

Puts a character into the stream.

```
int_type sputc(
   char_type _Ch
);
```

Parameters

_Ch

The character.

Return Value

Returns the character, if successful.

Remarks

If a **write position** is available, the member function stores *_Ch* in the write position, increments the next pointer for the output buffer, and returns **traits_type::**to_int_type(*_Ch*). Otherwise, it returns overflow(*_Ch*).

Example

```
// basic_streambuf_sputc.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
   using namespace std;

   int i = cout.rdbuf( )->sputc( 'a' );
   cout << endl << ( char )i << endl;
}
```

Output

```
a
a
```

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class

iostream Programming

iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::sputn

Puts a character string into the stream.

```
streamsize sputn(
   const char_type *_Ptr,
   streamsize _Count
);
```

Parameters

_Ptr

The character string.

_Count

The count of characters.

Return Value

The number of characters actually inserted into the stream.

Remarks

The member function returns xsputn(*_Ptr*, *_Count*). See the Remarks section of this member for more information.

Example

```
// basic_streambuf_sputn.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main()
{
    using namespace std;

    streamsize i = cout.rdbuf()->sputn("test", 4);
    cout << endl << i << endl;
}
```

Output

```
test
4
```

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class

iostream Programming

iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::stossc

Move past the current element in the stream.

```
void stossc( );
```

Remarks

The member function calls sbumpc. Note that an implementation is not required to supply this member function.

Example

```
// basic_streambuf_stossc.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
   using namespace std;
   ifstream myfile( "basic_streambuf_stossc.txt", ios::in );

   myfile.rdbuf( )->stossc( );
   char i = myfile.rdbuf( )->sgetc( );
   cout << i << endl;
}
```

Input: basic_streambuf_stossc.txt

```
testing
```

Output

```
e
```

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::sungetc

Gets a character from the stream.

```
int_type sungetc( );
```

Return Value

Returns either the character or failure.

Remarks

If a putback position is available, the member function decrements the next pointer for the input buffer and returns `traits_type::`to_int_type(`*`gptr). Otherwise, it returns pbackfail.

Example

```
// basic_streambuf_sungetc.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
   using namespace std;

   ifstream myfile( "basic_streambuf_sungetc.txt", ios::in );

   // Read and increment
   int i = myfile.rdbuf( )->sbumpc( );
   cout << ( char )i << endl;

   // Read and increment
   i = myfile.rdbuf( )->sbumpc( );
   cout << ( char )i << endl;

   // Decrement, read, and do not increment
   i = myfile.rdbuf( )->sungetc( );
   cout << ( char )i << endl;

   i = myfile.rdbuf( )->sungetc( );
   cout << ( char )i << endl;

   i = myfile.rdbuf( )->sbumpc( );
   cout << ( char )i << endl;
}
```

Input: basic_streambuf_sungetc.txt

```
testing
```

Output

```
t
e
e
t
t
```

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions
Other Resources
basic_streambuf Members

basic_streambuf::sync

A protected virtual function that tries to synchronize the controlled streams with any associated external streams.

```
virtual int sync( );
```

Return Value

If the function cannot succeed, it returns -1. The default behavior is to return zero.

Remarks

sync involves writing out any elements between the beginning and next pointers for the output buffer. It does not involve putting back any elements between the next and end pointers for the input buffer.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::uflow

A protected virtual function that extracts the current element from the input stream.

```
virtual int_type uflow( );
```

Return Value

The current element.

Remarks

The protected virtual member function tries to extract the current element **ch** from the input stream, then advance the current stream position, and return the element as **traits_type::**to_int_type(**ch**). It can do so in various ways:

- If a read position is available, it takes **ch** as the element stored in the read position and advances the next pointer for the input buffer.
- It can read an element directly, from some external source, and deliver it as the value **ch**.
- For a stream buffer with common input and output streams, it can make a read position available by writing out, to some external destination, some or all of the elements between the beginning and next pointers for the output buffer. Or it can allocate new or additional storage for the input buffer. The function then reads in, from some external source, one or more elements.

If the function cannot succeed, it returns **traits_type::**eof, or throws an exception. Otherwise, it returns the current element `ch` in the input stream, converted as described above, and advances the next pointer for the input buffer. The default behavior is to call underflow and, if that function returns **traits_type::eof**, to return **traits_type::eof**. Otherwise, the function returns the current element **ch** in the input stream, converted as previously described, and advances the next pointer for the input buffer.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::underflow

Protected, virtual function to extract the current element from the input stream.

```
virtual int_type underflow( );
```

Return Value

The current element.

Remarks

The protected virtual member function endeavors to extract the current element **ch** from the input stream, without advancing the current stream position, and return it as **traits_type::**to_int_type(**ch**). It can do so in various ways:

- If a read position is available, **ch** is the element stored in the read position. For more information on this, see the Remarks section of the basic_streambuf Class.
- It can make a read position available by allocating new or additional storage for the input buffer, then reading in, from some external source, one or more elements. For more information on this, see the Remarks section of the basic_streambuf Class.

If the function cannot succeed, it returns **traits_type::**eof**()** or throws an exception. Otherwise, it returns the current element in the input stream, converted as previously described. The default behavior is to return **traits_type::eof()**.

The virtual **underflow** function, with the sync and overflow functions, defines the characteristics of the **streambuf**-derived class. Each derived class might implement **underflow** differently, but the interface with the calling stream class is the same.

The **underflow** function is most frequently called by public **streambuf** functions like sgetc and sgetn when the get area is empty, but other classes, including the stream classes, can call **underflow** anytime.

The **underflow** function supplies the get area with characters from the input source. If the get area contains characters, **underflow** returns the first character. If the get area is empty, it fills the get area and returns the next character (which it leaves in the get area). If there are no more characters available, then **underflow** returns **EOF** and leaves the get area empty.

In the **strstreambuf** class, **underflow** adjusts the egptr pointer to access storage that was dynamically allocated by a call to **overflow**.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::xsgetn

Protected, virtual function to extract elements from the input stream.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct. Consider using basic_streambuf::_Xsgetn_s instead.

```
virtual streamsize xsgetn(
   char_type *_Ptr,
   streamsize _Count
);
```

Parameters

_Ptr

The buffer to contain the extracted characters.

_Count

The number of elements to extract.

Return Value

The number of elements extracted.

Remarks

The protected virtual member function extracts up to *_Count* elements from the input stream, as if by repeated calls to sbumpc, and stores them in the array beginning at *_Ptr*. It returns the number of elements actually extracted.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

basic_streambuf::_Xsgetn_s

Protected, virtual function to extract elements from the input stream.

```
virtual streamsize _Xsgetn_s(
    char_type *_Ptr,
    size_t _Ptr_size,
    streamsize _Count
);
```

Parameters

_Ptr

The buffer to contain the extracted characters.

_Ptr_size

The size of *_Ptr*.

_Count

The number of elements to extract.

Return Value

The number of elements extracted.

Remarks

The protected virtual member function extracts up to *_Count* elements from the input stream, as if by repeated calls to sbumpc, and stores them in the array beginning at *_Ptr*. It returns the number of elements actually extracted.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions
Safe Libraries: Standard C++ Library

Other Resources

basic_streambuf Members

basic_streambuf::xsputn

Protected, virtual function to insert elements into the output stream.

```
virtual streamsize xsputn(
   const char_type *_Ptr,
   streamsize _Count
);
```

Parameters

_Ptr

Pointer to elements to insert.

_Count

Number of elements to insert.

Return Value

The number of elements actually inserted into the stream.

Remarks

The protected virtual member function inserts up to *_Count* elements into the output stream, as if by repeated calls to sputc, from the array beginning at *_Ptr*. The insertion of characters into the output stream stops once all *_Count* characters have been written, or if calling `sputc(_Count)` would return `traits::eof()`. It returns the number of elements actually inserted.

Requirements

Header: <streambuf>

See Also

Reference

basic_streambuf Class
iostream Programming
iostreams Conventions

Other Resources

basic_streambuf Members

<string>

Defines the container template class basic_string and various supporting templates.

For a list of all members of this header, see <string> Members.

```
#include <string>
```

Remarks

The C++ language and the Standard C++ Library support two types of strings:

- Null-terminated character arrays often referred to as C strings.
- Template class objects, of type **basic_string**, that handle all **char**-like template arguments.

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<string> Members

Standard C++ Library Header Files

<string> Members

Reference

Typedefs

string	A type that describes a specialization of the template class basic_string with elements of type **char** as a string.
wstring	A type that describes a specialization of the template class basic_string with elements of type **wchar_t** as a wstring.

Operators

operator!=	Tests if the string object on the left side of the operator is not equal to the string object on the right side.
operator==	Tests if the string object on the left side of the operator is equal to the string object on the right side.
operator<	Tests if the string object on the left side of the operator is less than to the string object on the right side.
operator<<	A template function that inserts a string into the output stream.
operator<=	Tests if the string object on the left side of the operator is less than or equal to the string object on the right side.
operator>	Tests if the string object on the left side of the operator is greater than to the string object on the right side.
operator>=	Tests if the string object on the left side of the operator is greater than or equal to the string object on the right side.
operator>>	A template function that extracts a string from the input stream.
operator+	Concatenates two string objects.

Specialized Template Functions

swap	Exchanges the arrays of characters of two strings.

Functions

getline	Extract strings from the input stream line by line.

Classes

basic_string Class	A template class that describes objects that can store a sequence of arbitrary character-like objects.
char_traits Class	A template class that describes attributes associated with a character of type **CharType**

Specializations

char_traits<char> Class	A class that is a specialization of the template class char_traits<**CharType**> to an element of type **char**.
char_traits<wchar_t> Class	A class that is a specialization of the template class char_traits<**CharType**> to an element of type **wchar_t**.

See Also

Reference

<string>

Thread Safety in the Standard C++ Library

<string> Typedefs

For information about the typedefs in <string>, see <string> Members.

string (<string>)

A type that describes a specialization of the template class basic_string with elements of type **char** as a string.

```
typedef basic_string<char> string;
```

Example

```
// string_string.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // Equivalent ways to declare an object
   // of type basic_string <char>
   const basic_string <char> s1 ( "test" );
   string s2 ( "test" );   // Uses the typedef for string

   // comparison between two objects of type basic_string
   if ( s1 == s2 )
      cout << "The strings s1 & s2 are equal." << endl;
   else
      cout << "The strings s1 & s2are not equal." << endl;
}
```

Output

```
The strings s1 & s2 are equal.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

<string> Members

wstring

A type that describes a specialization of the template class basic_string with elements of type **wchar_t** as a wstring.

```
typedef basic_string<wchar_t> wstring;
```

Example

```
// string_wstring.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   // Equivalent ways to declare an object of type
   // basic_string <wchar_t>
   const basic_string <wchar_t> s1 ( L"abc" );
   wstring s2 ( L"abc" );   // Uses the typedef for wstring

   // Comparison between two objects of type basic_string <wchar_t>
   if ( s1 == s2 )
      cout << "The strings s1 & s2 are equal." << endl;
   else
      cout << "The strings s1 & s2 are not equal." << endl;
}
```

Output

```
The strings s1 & s2 are equal.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

<string> Members

<string> Operators

For more information about the operators in <string>, see <string> Members.

operator+ (<string>)

Concatenates two string objects.

```
template<class CharType, class Traits, class Allocator>
   basic_string<CharType, Traits, Allocator> operator+(
      const basic_string<CharType, Traits, Allocator>& _Left,
      const basic_string<CharType, Traits, Allocator>& _Right
   );
template<class CharType, class Traits, class Allocator>
   basic_string<CharType, Traits, Allocator> operator+(
      const basic_string<CharType, Traits, Allocator>& _Left,
      const CharType *_Right
      );
template<class CharType, class Traits, class Allocator>
   basic_string<CharType, Traits, Allocator> operator+(
      const basic_string<CharType, Traits, Allocator>& _Left,
      CharType _Right
      );
template<class CharType, class Traits, class Allocator>
   basic_string<CharType, Traits, Allocator> operator+(
      const CharType *_Left,
      const basic_string<CharType, Traits, Allocator>& _Right
      );
template<class CharType, class Traits, class Allocator>
   basic_string<CharType, Traits, Allocator> operator+(
      CharType _Left,
      const basic_string<CharType, Traits, Allocator>& _Right
      );
```

Parameters

_Left

A C-style string or an object of type **basic_string** to be concatenated.

_Right

A C-style string or an object of type **basic_string** to be concatenated.

Return Value

The string that is the concatenation of the input strings.

Remarks

The functions each overload **operator+** to concatenate two objects of template class basic_string. All effectively return **basic_string**<**CharType**, **Traits**, **Allocator**>(*_Left*).append(*_Right*).

Example

```
// string_op_con.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   // Declaring an object of type basic_string<char>
   string s1 ( "anti" );
   string s2 ( "gravity" );
   cout << "The basic_string s1 = " << s1 << "." << endl;
   cout << "The basic_string s2 = " << s2 << "." << endl;

   // Declaring a C-style string
   char *s3 = "heroine";
```

```
    cout << "The C-style string s3 = " << s3 << "." << endl;

    // Declaring a character constant
    char c1 = '!';
    cout << "The character constant c1 = " << c1 << "." << endl;

    // First member function: concatenates an  object
    // of type basic_string with an object of type basic_string
    string s12 = s1 + s2;
    cout << "The string concatenating s1 & s2 is: " << s12 << endl;

    // Second & fourth member functions: concatenate an object
    // of type basic_string with an object of C-syle string type
    string s1s3 = s1 + s3;
    cout << "The string concatenating s1 & s3 is: " << s1s3 << endl;

    // Third & fifth member functions: concatenate an object
    // of type basic_string with a character constant
    string s1s3c1 = s1s3 + c1;
    cout << "The string concatenating s1 & s3 is: " << s1s3c1 << endl;
}
```

Output

```
The basic_string s1 = anti.
The basic_string s2 = gravity.
The C-style string s3 = heroine.
The character constant c1 = !.
The string concatenating s1 & s2 is: antigravity
The string concatenating s1 & s3 is: antiheroine
The string concatenating s1 & s3 is: antiheroine!
```

Requirements

Header: <string>

See Also

Reference

string::operator+

Other Resources

<string> Members

operator!= (<string>)

Tests if the string object on the left side of the operator is not equal to the string object on the right side.

```
template<class CharType, class Traits, class Allocator>
   bool operator!=(
      const basic_string<CharType, Traits, Allocator>& _Left,
      const basic_string<CharType, Traits, Allocator>& _Right
   );
template<class CharType, class Traits, class Allocator>
   bool operator!=(
      const basic_string<CharType, Traits, Allocator>& _Left,
      const CharType *_Right
      );
template<class CharType, class Traits, class Allocator>
   bool operator!=(
      const CharType *_Left,
      const basic_string<CharType, Traits, Allocator>& _Right
      );
```

Parameters

_Left

A C-style string or an object of type **basic_string** to be compared.

_Right

A C-style string or an object of type **basic_string** to be compared.

Return Value

true if the string object on the left side of the operator is not lexicographically equal to the string object on the right side; otherwise **false**.

Remarks

The comparison between string objects is based on a pairwise lexicographical comparison of their characters. Two strings are equal if they have the same number of characters and their respective character values are the same. Otherwise, they are unequal.

Example

```
// string_op_ne.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // Declaring an objects of type basic_string<char>
   string s1 ( "pluck" );
   string s2 ( "strum" );
   cout << "The basic_string s1 = " << s1 << "." << endl;
   cout << "The basic_string s2 = " << s2 << "." << endl;

   // Declaring a C-style string
   char *s3 = "pluck";
   cout << "The C-style string s3 = " << s3 << "." << endl;

   // First member function: comparison between left-side object
   // of type basic_string & right-side object of type basic_string
   if ( s1 != s2 )
      cout << "The strings s1 & s2 are not equal." << endl;
   else
```

```
      cout << "The strings s1 & s2 are equal." << endl;

   // Second member function: comparison between left-side object
   // of type basic_string & right-side object of C-syle string type
   if ( s1 != s3 )
      cout << "The strings s1 & s3 are not equal." << endl;
   else
      cout << "The strings s1 & s3 are equal." << endl;

   // Third member function: comparison between left-side object
   // of C-syle string type & right-side object of type basic_string
   if ( s3 != s2 )
      cout << "The strings s3 & s2 are not equal." << endl;
   else
      cout << "The strings s3 & s2 are equal." << endl;
}
```

Output

```
The basic_string s1 = pluck.
The basic_string s2 = strum.
The C-style string s3 = pluck.
The strings s1 & s2 are not equal.
The strings s1 & s3 are equal.
The strings s3 & s2 are not equal.
```

Requirements

Header: <string>

See Also

Reference

string::operator!=

Other Resources

<string> Members

operator== (<string>)

Tests if the string object on the left side of the operator is equal to the string object on the right side.

```
template<class CharType, class Traits, class Allocator>
   bool operator==(
      const basic_string<CharType, Traits, Allocator>& _Left,
      const basic_string<CharType, Traits, Allocator>& _Right
      );
template<class CharType, class Traits, class Allocator>
   bool operator==(
      const basic_string<CharType, Traits, Allocator>& _Left,
      const CharType *_Right
      );
template<class CharType, class Traits, class Allocator>
   bool operator==(
      const CharType *_Left,
      const basic_string<CharType, Traits, Allocator>& _Right
      );
```

Parameters

_Left

A C-style string or an object of type **basic_string** to be compared.

_Right

A C-style string or an object of type **basic_string** to be compared.

Return Value

true if the string object on the left side of the operator is lexicographically equal to the string object on the right side; otherwise **false**.

Remarks

The comparison between string objects is based on a pairwise lexicographical comparison of their characters. Two strings are equal if they have the same number of characters and their respective character values are the same. Otherwise, they are unequal.

Example

```
// string_op_eq.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // Declaring an objects of type basic_string<char>
   string s1 ( "pluck" );
   string s2 ( "strum" );
   cout << "The basic_string s1 = " << s1 << "." << endl;
   cout << "The basic_string s2 = " << s2 << "." << endl;

   // Declaring a C-style string
   char *s3 = "pluck";
   cout << "The C-style string s3 = " << s3 << "." << endl;

   // First member function: comparison between left-side object
   // of type basic_string & right-side object of type basic_string
   if ( s1 == s2 )
      cout << "The strings s1 & s2 are equal." << endl;
   else
```

```
      cout << "The strings s1 & s2 are not equal." << endl;

   // Second member function: comparison between left-side object
   // of type basic_string & right-side object of C-syle string type
   if ( s1 == s3 )
      cout << "The strings s1 & s3 are equal." << endl;
   else
      cout << "The strings s1 & s3 are not equal." << endl;

   // Third member function: comparison between left-side object
   // of C-syle string type & right-side object of type basic_string
   if ( s3 == s2 )
      cout << "The strings s3 & s2 are equal." << endl;
   else
      cout << "The strings s3 & s2 are not equal." << endl;
}
```

Output

```
The basic_string s1 = pluck.
The basic_string s2 = strum.
The C-style string s3 = pluck.
The strings s1 & s2 are not equal.
The strings s1 & s3 are equal.
The strings s3 & s2 are not equal.
```

Requirements

Header: <string>

See Also

Reference

string::operator==

Other Resources

<string> Members

operator< (<string>)

Tests if the string object on the left side of the operator is less than to the string object on the right side.

```
template<class CharType, class Traits, class Allocator>
   bool operator<(
      const basic_string<CharType, Traits, Allocator>& _Left,
      const basic_string<CharType, Traits, Allocator>& _Right
      );
template<class CharType, class Traits, class Allocator>
   bool operator<(
      const basic_string<CharType, Traits, Allocator>& _Left,
      const CharType *_Right
      );
template<class CharType, class Traits, class Allocator>
   bool operator<(
      const CharType *_Left,
      const basic_string<CharType, Traits, Allocator>& _Right
      );
```

Parameters

_Left

A C-style string or an object of type **basic_string** to be compared.

_Right

A C-style string or an object of type **basic_string** to be compared.

Return Value

true if the string object on the left side of the operator is lexicographically less than the string object on the right side; otherwise **false**.

Remarks

A lexicographical comparison between strings compares them character by character until:

- It finds two corresponding characters unequal, and the result of their comparison is taken as the result of the comparison between the strings.
- It finds no inequalities, but one string has more characters than the other, and the shorter string is considered less than the longer string.
- It finds no inequalities and finds that the strings have the same number of characters, and so the strings are equal.

Example

```
// string_op_lt.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   // Declaring an objects of type basic_string<char>
   string s1 ( "strict" );
   string s2 ( "strum" );
   cout << "The basic_string s1 = " << s1 << "." << endl;
   cout << "The basic_string s2 = " << s2 << "." << endl;

   // Declaring a C-style string
   char *s3 = "strict";
   cout << "The C-style string s3 = " << s3 << "." << endl;
```

```
    // First member function: comparison between left-side object
    // of type basic_string & right-side object of type basic_string
    if ( s1 < s2 )
       cout << "The string s1 is less than the string s2." << endl;
    else
       cout << "The string s1 is not less than the string s2." << endl;

    // Second member function: comparison between left-hand object
    // of type basic_string & right-hand object of C-syle string type
    if ( s1 < s3 )
       cout << "The string s1 is less than the string s3." << endl;
    else
       cout << "The string s1 is not less than the string s3." << endl;

    // Third member function: comparison between left-hand object
    // of C-syle string type & right-hand object of type basic_string
    if ( s3 < s2 )
       cout << "The string s3 is less than the string s2." << endl;
    else
       cout << "The string s3 is not less than the string s2." << endl;
 }
```

Output

```
The basic_string s1 = strict.
The basic_string s2 = strum.
The C-style string s3 = strict.
The string s1 is less than the string s2.
The string s1 is not less than the string s3.
The string s3 is less than the string s2.
```

Requirements

Header: <string>

See Also

Reference

string::operator<

Other Resources

<string> Members

operator<< (<string>)

A template function that writes a string into the output stream.

```
template<class CharType, class Traits, class Allocator>
   basic_ostream<CharType, Traits>& operator<<(
      basic_ostream<CharType, Traits>& _Ostr,
      const basic_string<CharType, Traits, Allocator>& _Str
);
```

Parameters

_Ostr

The output stream being written to.

_Str

The string to be entered into the output stream.

Return Value

Writes the value of the specified string to the output stream *_Ostr*.

Remarks

The template function overloads **operator<<** to insert an object *_Str* of template class basic_string into the stream *_Ostr*. The function effectively returns *_Ostr*.**write**(*_Str*.c_str, *_Str*.size).

Requirements

Header: <string>

See Also

Reference

string::operator<<

Other Resources

<string> Members

operator<= (in <string>)

Tests if the string object on the left side of the operator is less than or equal to the string object on the right side.

```
template<class CharType, class Traits, class Allocator>
   bool operator<=(
      const basic_string<CharType, Traits, Allocator>& _Left,
      const basic_string<CharType, Traits, Allocator>& _Right
      );
template<class CharType, class Traits, class Allocator>
   bool operator<=(
      const basic_string<CharType, Traits, Allocator>& _Left,
      const CharType *_Right
   );
template<class CharType, class Traits, class Allocator>
   bool operator<=(
      const CharType *_Left,
      const basic_string<CharType, Traits, Allocator>& _Right
   );
```

Parameters

_Left

A C-style string or an object of type **basic_string** to be compared.

_Right

A C-style string or an object of type **basic_string** to be compared.

Return Value

true if the string object on the left side of the operator is lexicographically less than or equal to the string object on the right side; otherwise **false**.

Remarks

A lexicographical comparison between strings compares them character by character until:

- It finds two corresponding characters unequal, and the result of their comparison is taken as the result of the comparison between the strings.
- It finds no inequalities, but one string has more characters than the other, and the shorter string is considered less than the longer string.
- It finds no inequalities and finds that the strings have the same number of characters, so the strings are equal.

Example

```
// string_op_le.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // Declaring an objects of type basic_string<char>
   string s1 ( "strict" );
   string s2 ( "strum" );
   cout << "The basic_string s1 = " << s1 << "." << endl;
   cout << "The basic_string s2 = " << s2 << "." << endl;

   // Declaring a C-style string
   char *s3 = "strict";
   cout << "The C-style string s3 = " << s3 << "." << endl;
```

```
    // First member function: comparison between left-side object
    // of type basic_string & right-side object of type basic_string
    if ( s1 <= s2 )
       cout << "The string s1 is less than or equal to "
            << "the string s2." << endl;
    else
       cout << "The string s1 is greater than "
            << "the string s2." << endl;

    // Second member function: comparison between left-side object
    // of type basic_string & right-side object of C-syle string type
    if ( s1 <= s3 )
       cout << "The string s1 is less than or equal to "
            << "the string s3." << endl;
    else
       cout << "The string s1 is greater than "
            << "the string s3." << endl;

    // Third member function: comparison between left-side object
    // of C-syle string type  & right-side object of type basic_string
    if ( s2 <= s3 )
       cout << "The string s2 is less than or equal to "
            << "the string s3." << endl;
    else
       cout << "The string s2 is greater than "
            << "the string s3." << endl;
 }
```

Output

```
The basic_string s1 = strict.
The basic_string s2 = strum.
The C-style string s3 = strict.
The string s1 is less than or equal to the string s2.
The string s1 is less than or equal to the string s3.
The string s2 is greater than the string s3.
```

Requirements

Header: <string>

See Also

Reference

string::operator<=

Other Resources

<string> Members

operator> (<string>)

Tests if the string object on the left side of the operator is greater than to the string object on the right side.

```
template<class CharType, class Traits, class Allocator>
   bool operator>(
      const basic_string<CharType, Traits, Allocator>& _Left,
      const basic_string<CharType, Traits, Allocator>& _Right
   );
template<class CharType, class Traits, class Allocator>
   bool operator>(
      const basic_string<CharType, Traits, Allocator>& _Left,
      const CharType *_Right
   );
template<class CharType, class Traits, class Allocator>
   bool operator>(
      const CharType *_Left,
      const basic_string<CharType, Traits, Allocator>& _Right
   );
```

Parameters

_Left

A C-style string or an object of type **basic_string** to be compared.

_Right

A C-style string or an object of type **basic_string** to be compared.

Return Value

true if the string object on the left side of the operator is lexicographically greater than the string object on the right side; otherwise **false**.

Remarks

A lexicographical comparison between strings compares them character by character until:

- It finds two corresponding characters unequal, and the result of their comparison is taken as the result of the comparison between the strings.
- It finds no inequalities, but one string has more characters than the other, and the shorter string is considered less than the longer string.
- It finds no inequalities and finds that the strings have the same number of characters, and so the strings are equal.

Example

```
// string_op_gt.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // Declaring an objects of type basic_string<char>
   string s1 ( "strict" );
   string s2 ( "strum" );
   cout << "The basic_string s1 = " << s1 << "." << endl;
   cout << "The basic_string s2 = " << s2 << "." << endl;

   // Declaring a C-style string
   char *s3 = "stricture";
   cout << "The C-style string s3 = " << s3 << "." << endl;
```

```
    // First member function: comparison between left-side object
    // of type basic_string & right-side object of type basic_string
    if ( s1 > s2 )
       cout << "The string s1 is greater than "
            << "the string s2." << endl;
    else
       cout << "The string s1 is not greater than "
            << "the string s2." << endl;

    // Second member function: comparison between left-side object
    // of type basic_string & right-side object of C-syle string type
    if ( s3 > s1 )
       cout << "The string s3 is greater than "
            << "the string s1." << endl;
    else
       cout << "The string s3 is not greater than "
            << "the string s1." << endl;

    // Third member function: comparison between left-side object
    // of C-syle string type & right-side object of type basic_string
    if ( s2 > s3 )
       cout << "The string s2 is greater than "
            << "the string s3." << endl;
    else
       cout << "The string s2 is not greater than "
            << "the string s3." << endl;
 }
```

Output

```
The basic_string s1 = strict.
The basic_string s2 = strum.
The C-style string s3 = stricture.
The string s1 is not greater than the string s2.
The string s3 is greater than the string s1.
The string s2 is greater than the string s3.
```

Requirements

Header: <string>

See Also

Reference

string::operator>

Other Resources

<string> Members

operator>> (<string>)

A template function that reads a string from an input stream.

```
template<class CharType, class Traits, class Allocator>
   basic_istream<CharType, Traits>& operator>>(
      basic_istream<CharType, Traits>& _Istr,
      const basic_string<CharType, Traits, Allocator>& _Right
   );
```

Parameters

_Istr

The input stream used to extract the sequence

_Right

The string that is being extracted from the input stream.

Return Value

Reads the value of the specified string from *_Istr* and returns it into *_Right*.

Remarks

The operator skips the leading white spaces unless the **skipws** flag is set. It reads all the following characters until the next character is a white space or the end of the file is reached.

The template function overloads **operator>>** to replace the sequence controlled by *_Right* with a sequence of elements extracted from the stream *_Istr*. Extraction stops:

- At end of file.
- After the function extracts *_Istr*.**width** elements, if that value is nonzero.

After the function extracts *_Istr*.max_size elements.

- After the function extracts an element *ch* for which use_facet<**ctype**<**CharType**> >(**getloc**). **is**(**ctype**<**CharType**>::**space**, *ch*) is true, in which case the character is put back.

If the function extracts no elements, it calls setstate(**ios_base::failbit**). In any case, it calls **istr**.**width**(0) and returns ***this**.

Example

```
// string_op_read_.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   string c0;
   cout << "Input a string c0 ( try: Fibonacci numbers ): ";
   cin >> c0;
   cout << "The string entered is c0 = " << c0 << endl;
}
```

Input

```
Fibonacci numbers
```

Sample Output

```
Input a string c0 ( try: Fibonacci numbers ): Fibonacci numbers
```

```
The string entered is c0 = Fibonacci
```

Requirements

Header: <string>

See Also

Reference

string::operator>>

Other Resources

<string> Members

operator>= (<string>)

Tests if the string object on the left side of the operator is greater than or equal to the string object on the right side.

```
template<class CharType, class Traits, class Allocator>
   bool operator>=(
      const basic_string<CharType, Traits, Allocator>& _Left,
      const basic_string<CharType, Traits, Allocator>& _Right
   );
template<class CharType, class Traits, class Allocator>
   bool operator>=(
      const basic_string<CharType, Traits, Allocator>& _Left,
      const CharType *_Right
   );
template<class CharType, class Traits, class Allocator>
   bool operator>=(
      const CharType *_Left,
      const basic_string<CharType, Traits, Allocator>& _Right
   );
```

Parameters

_Left

A C-style string or an object of type **basic_string** to be compared.

_Right

A C-style string or an object of type **basic_string** to be compared.

Return Value

true if the string object on the left side of the operator is lexicographically greater than or equal to the string object on the right side; otherwise **false**.

Remarks

A lexicographical comparison between strings compares them character by character until:

- It finds two corresponding characters unequal, and the result of their comparison is taken as the result of the comparison between the strings.
- It finds no inequalities, but one string has more characters than the other, and the shorter string is considered less than the longer string.
- It finds no inequalities and finds the strings have the same number of characters, and so the strings are equal.

Example

```
// string_op_ge.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // Declaring an objects of type basic_string<char>
   string s1 ( "strict" );
   string s2 ( "strum" );
   cout << "The basic_string s1 = " << s1 << "." << endl;
   cout << "The basic_string s2 = " << s2 << "." << endl;

   // Declaring a C-style string
   char *s3 = "stricture";
   cout << "The C-style string s3 = " << s3 << "." << endl;
```

```
    // First member function: comparison between left-side object
    // of type basic_string & right-side object of type basic_string
    if ( s1 >= s2 )
       cout << "The string s1 is greater than or equal to "
            << "the string s2." << endl;
    else
       cout << "The string s1 is less than "
            << "the string s2." << endl;

    // Second member function: comparison between left-side object
    // of type basic_string & right-side object of C-syle string type
    if ( s3 >= s1 )
       cout << "The string s3 is greater than or equal to "
            << "the string s1." << endl;
    else
       cout << "The string s3 is less than "
            << "the string s1." << endl;

    // Third member function: comparison between left-side object
    // of C-syle string type & right-side object of type basic_string
    if ( s2 >= s3 )
       cout << "The string s2 is greater than or equal to "
            << "the string s3." << endl;
    else
       cout << "The string s2 is less than "
            << "the string s3." << endl;
 }
```

Output

```
The basic_string s1 = strict.
The basic_string s2 = strum.
The C-style string s3 = stricture.
The string s1 is less than the string s2.
The string s3 is greater than or equal to the string s1.
The string s2 is greater than or equal to the string s3.
```

Requirements

Header: <string>

See Also

Reference

string::operator>=

Other Resources

<string> Members

<string> Specialized Template Functions

For more information about specialized template functions in <string>, see <string> Members.

swap (<string>)

Exchanges the arrays of characters of two strings.

```
template<class Traits, class Allocator>
   void swap(
      basic_string<CharType, Traits, Allocator>& _Left,
      basic_string<CharType, Traits, Allocator>& _Right
      );
```

Parameters

_Left

One string whose elements are to be swapped with those of another string.

_Right

The other string whose elements are to be swapped with the first string.

Remarks

The template function executes the specialized member function *_Left*.swap(*_Right*) for strings, which guarantees constant complexity.

Example

```
// string_swap.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   // Declaring an object of type basic_string<char>
   string s1 ( "Tweedledee" );
   string s2 ( "Tweedledum" );
   cout << "Before swapping string s1 and s2:" << endl;
   cout << "The basic_string s1 = " << s1 << "." << endl;
   cout << "The basic_string s2 = " << s2 << "." << endl;

   swap ( s1 , s2 );
   cout << "\nAfter swapping string s1 and s2:" << endl;
   cout << "The basic_string s1 = " << s1 << "." << endl;
   cout << "The basic_string s2 = " << s2 << "." << endl;
}
```

Output

```
Before swapping string s1 and s2:
The basic_string s1 = Tweedledee.
The basic_string s2 = Tweedledum.

After swapping string s1 and s2:
The basic_string s1 = Tweedledum.
The basic_string s2 = Tweedledee.
```

Requirements

Header: <string>

See Also

Other Resources

<string> Members

<string> Functions

For information about the functions in <string>, see <string> Members.

getline

Extract strings from the input stream line-by-line.

```
template<class CharType, class Traits, class Allocator>
   basic_istream<CharType, Traits>& getline(
      basic_istream<CharType, Traits>& _Istr,
      basic_string<CharType, Traits, Allocator>& _Str
);

template<class CharType, class Traits, class Allocator>
   basic_istream<CharType, Traits>& getline(
      basic_istream<CharType, Traits>& _Istr,
      basic_string<CharType, Traits, Allocator>& _Str,
      CharType _Delim
);
```

Parameters

_Istr

The input stream from which a string is to be extracted.

_Str

The string into which are read the characters from the input stream.

_Delim

The line delimiter.

Return Value

The first function returns **getline**(*_Istr*, *_Str*, *_Istr*.**widen**('**\n**')).

The second function replaces the sequence controlled by *_Str* with a sequence of elements extracted from the stream *_Istr*.

Remarks

In order of testing, extraction stops:

- At end of file.
- After the function extracts an element that compares equal to **delim**, in which case the element is neither put back nor appended to the controlled sequence.
- After the function extracts `str.`max_size elements, in which case the function calls **setstate**(**ios_base::failbit**).

If the function extracts no elements, it calls **setstate**(**failbit**). In any case, it returns *_Istr*.

Example

See string::getline for an example of using **getline**.

Requirements

Header: <string>

See Also

Reference

string::getline

Other Resources

<string> Members

<string> Classes

For more information about classes in <string>, see <string> Members.

basic_string Class

The sequences controlled by an object of template class basic_string are the Standard C++ string class and are usually referred to as strings, but they should not be confused with the null-terminated C-strings used throughout the Standard C++ Library. The string class is a container that enables the use of strings as normal types, such as using comparison and concatenation operations, iterators, and STL algorithms and copying and assigning with class allocator managed memory.

For a list of all members of this type, see basic_string Members.

```
template <
   class CharType,
   class Traits=char_traits<CharType>,
   class Allocator=allocator<CharType>
>
class basic_string
```

Parameters

CharType

The data type of a single character to be stored in the string. The Standard C++ Library provides two specializations of this template class, with the type definitions string, for elements of type **char**, and wstring, for elements of type **wchar_t**.

Traits

Various important properties of the **CharType** elements in a basic_string specialization are described by the class **Traits**.

Allocator

The type that represents the stored allocator object that encapsulates details about the string's allocation and deallocation of memory. The default value is **allocator**<*Type*>.

Many member functions require an operand sequence of elements. You can specify such an operand sequence several ways.

Element	**Description**
_Ch	One element with character value *_Ch*.
_Count, _Ch	A repetition of *_Count* elements each with value *_Ch*.
_Ptr	A null-terminated sequence, such as a C string, with a **CharType** of type **char**, beginning at *_Ptr* (which must not be a null pointer), where the terminating element is the value **value_type** and is not part of the operand sequence.
_Ptr, _Count	A sequence of *_Count* elements beginning at *_Ptr*.
_Str	The sequence specified by a basic_string object.
_Str, _Off, _Count	The substring of the basic_string object *_Str* with up to *_Count* elements (or through the end of the string, whichever comes first) beginning at position *_Off*.
_First, _Last	A sequence of elements delimited by the iterators *_First* and *_Last*, in the range [*_First, _Last*), which may overlap the sequence controlled by the string object whose member function is being called.

If a position argument (such as *_Off*) is beyond the end of the string on a call to a **basic_string** member function, the function reports an out-of-range error by throwing an object of type out_of_range Class.

If a function is asked to generate a sequence longer than max_size elements, the function reports a length error by throwing an object of type length_error Class.

References, pointers, and iterators that designate elements of the controlled sequence can become invalid after any call to a function that alters the controlled sequence, or after the first call to a non-**const** member function.

Requirements

Header: <string>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

basic_string Members

<string> Members

basic_string Members

Reference

Constructors

basic_string	Constructs a string that is empty or initialized by specific characters or that is a copy of all or part of some other string object or C-string.

Typedefs

allocator_type	A type that represents the **allocator** class for a string object.
const_iterator	A type that provides a random-access iterator that can access and read a **const** element in the string.
const_pointer	A type that provides a pointer to a **const** element in a string.
const_reference	A type that provides a reference to a **const** element stored in a string for reading and performing **const** operations.
const_reverse_iterator	A type that provides a random-access iterator that can read any **const** element in the string.
difference_type	A type that provides the difference between two iterators that refer to elements within the same string.
iterator	A type that provides a random-access iterator that can read or modify any element in a string.
npos	An unsigned integral value initialized to –1 that indicates either "not found" or "all remaining characters" when a search function fails.
pointer	A type that provides a pointer to a character element in a string or character array.
reference	A type that provides a reference to an element stored in a string.
reverse_iterator	A type that provides a random-access iterator that can read or modify an element in a reversed string.
size_type	An unsigned integral type for the number of elements in a string.
traits_type	A type for the character traits of the elements stored in a string.
value_type	A type that represents the type of characters stored in a string.

Member Functions

append	Adds characters to the end of a string.
assign	Assigns new character values to the contents of a string.
at	Returns a reference to the element at a specified location in the string.
begin	Returns an iterator addressing the first element in the string.
c_str	Converts the contents of a string as a C-style, null-terminated, string.
capacity	Returns the largest number of elements that could be stored in a string without increasing the memory allocation of the string.

clear	Erases all elements of a string.
compare	Compares a string with a specified string to determine if the two strings are equal or if one is lexicographicall y less than the other.
copy	Copies at most a specified number of characters from an indexed position in a source string to a target charac ter array. Deprecated. Use basic_string::_Copy_s instead.
_Copy_s	Copies at most a specified number of characters from an indexed position in a source string to a target charac ter array.
data	Converts the contents of a string into an array of characters.
empty	Tests whether the string is contains characters or not.
end	Returns an iterator that addresses the location succeeding the last element in a string.
erase	Removes an element or a range of elements in a string from a specified position.
find	Searches a string in a forward direction for the first occurrence of a substring that matches a specified sequen ce of characters.
find_first_not_of	Searches through a string for the first character that is not any element of a specified string.
find_first_of	Searches through a string for the first character that matches any element of a specified string.
find_last_not_of	Searches through a string for the last character that is not any element of a specified string.
find_last_of	Searches through a string for the last character that is an element of a specified string.
get_allocator	Returns a copy of the **allocator** object used to construct the string.
insert	Inserts an element or a number of elements or a range of elements into the string at a specified position.
length	Returns the current number of elements in a string.
max_size	Returns the maximum number of characters a string could contain.
push_back	Adds an element to the end of the string.
rbegin	Returns an iterator to the first element in a reversed string.
rend	Returns an iterator that points just beyond the last element in a reversed string.
replace	Replaces elements in a string at a specified position with specified characters or characters copied from other ranges or strings or C-strings.
reserve	Sets the capacity of the string to a number at least as great as a specified number.
resize	Specifies a new size for a string, appending or erasing elements as required.
rfind	Searches a string in a backward direction for the first occurrence of a substring that matches a specified sequ ence of characters.

size	Returns the current number of elements in a string.
substr	Copies a substring of at most some number of characters from a string beginning from a specified position.
swap	Exchange the contents of two strings.

Operators

operator+=	Appends characters to a string.
operator=	Assigns new character values to the contents of a string.
operator[]	Provides a reference to the character with a specified index in a string.

See Also

Reference

basic_string Class

Thread Safety in the Standard C++ Library

basic_string Typedefs

For information about the typedefs in the basic_string class, see basic_string Members.

basic_string::allocator_type

A type that represents the allocator class for a string object.

```
typedef Allocator allocator_type;
```

Remarks

The type is a synonym for the template parameter **Allocator**.

Example

```
// basic_string_allocator_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   // The following lines declare objects
   // that use the default allocator.
   string s1;
   basic_string <char>::allocator_type xchar = s1.get_allocator( );
   // You can now call functions on the allocator class xchar used by s1
}
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::const_iterator

A type that provides a random-access iterator that can access and read a **const** element in the string.

```
typedef implementation-defined const_iterator;
```

Remarks

A type **const_iterator** cannot be used to modify the value of a character and is used to iterate through a string in a forward direction.

Example

See the example for begin for an example of how to declare and use **const_iterator**.

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::const_pointer

A type that provides a pointer to a **const** element in a string.

```
typedef typename allocator_type::const_pointer const_pointer;
```

Remarks

The type is a synonym for **allocator_type::const_pointer**.

For type **string**, it is equivalent to **char***.

Pointers that are declared const must be initialized when they are declared. Const pointers always point to the same memory location and may point to constant or nonconstant data.

Example

```
// basic_string_const_ptr.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   basic_string<char>::const_pointer pstr1a = "In Here";
   const char *cstr1c = "Out There";

   cout << "The string pstr1a is: " << pstr1a <<  "." << endl;
   cout << "The C-string cstr1c is: " << cstr1c << "." << endl;
}
```

Output

```
The string pstr1a is: In Here.
The C-string cstr1c is: Out There.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::const_reference

A type that provides a reference to a **const** element stored in a string for reading and performing **const** operations.

```
typedef typename allocator_type::const_reference const_reference;
```

Remarks

A type **const_reference** cannot be used to modify the value of an element.

The type is a synonym for **allocator_type::const_reference**. For string **type**, it is equivalent to const **char&**.

Example

See the example for at for an example of how to declare and use **const_reference**.

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::const_reverse_iterator

A type that provides a random-access iterator that can read any **const** element in the string.

```
typedef reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type **const_reverse_iterator** cannot modify the value of a character and is used to iterate through a string in reverse.

Example

See the example for rbegin for an example of how to declare and use **const_reverse_iterator**.

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::difference_type

A type that provides the difference between two iterators that refer to elements within the same string.

```
typedef implementation-defined difference_type;
```

Remarks

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.

For type **string**, it is equivalent to **ptrdiff_t**.

Example

```
// basic_string_diff_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ( "quintillion" );
   cout << "The original string str1 is: " << str1 << endl;
   basic_string <char>::size_type indexChFi, indexChLi;

   indexChFi = str1.find_first_of ( "i" );
   indexChLi = str1.find_last_of ( "i" );
   basic_string<char>::difference_type diffi = indexChLi - indexChFi;

   cout << "The first character i is at position: "
        << indexChFi << "." << endl;
   cout << "The last character i is at position: "
        << indexChLi << "." << endl;
   cout << "The difference is: " << diffi << "." << endl;
}
```

Output

```
The original string str1 is: quintillion
The first character i is at position: 2.
The last character i is at position: 8.
The difference is: 6.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::iterator

A type that provides a random-access iterator that can access and read a **const** element in the string.

```
typedef implementation-defined iterator;
```

Remarks

A type **iterator** can be used to modify the value of a character and is used to iterate through a string in a forward direction.

Example

See the example for begin for an example of how to declare and use **iterator**.

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::npos

An unsigned integral value initialized to –1 that indicates either "not found" or "all remaining characters" when a search function fails.

```
static const size_type npos = -1;
```

Remarks

When the return value is to be checked for the **npos** value, it might not work unless the return value is of type size_type and not either **int** or **unsigned**.

Example

See the example for find for an example of how to declare and use **npos**.

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::pointer

A type that provides a pointer to a character element in a string or character array.

```
typedef typename allocator_type::pointer pointer;
```

Remarks

The type is a synonym for **allocator_type::pointer**.

For type **string**, it is equivalent to **char***.

Example

```
// basic_string_pointer.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   basic_string<char>::pointer pstr1a = "In Here";
   char *cstr1b = "Out There";
   cout << "The string pstr1a is: " << pstr1a <<  "." << endl;
   cout << "The C-string cstr1b is: " << cstr1b << "." << endl;
}
```

Output

```
The string pstr1a is: In Here.
The C-string cstr1b is: Out There.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::reference

A type that provides a reference to an element stored in a string.

```
typedef typename allocator_type::reference reference;
```

Remarks

A type **reference** can be used to modify the value of an element.

The type is a synonym for **allocator_type::reference**.

For type **string**, it is equivalent to **chr&**.

Example

See the example for at for an example of how to declare and use **reference**.

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::reverse_iterator

A type that provides a reference to an element stored in a string.

```
typedef reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type **reverse_iterator** can be used to modify the value of a character and is used to iterate through a string in reverse.

Example

See the example for rbegin for an example of how to declare and use **reverse_iterator**.

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::size_type

An unsigned integer type that can represent the number of elements and indices in a string.

```
typedef implementation-defined size_type;
```

Remarks

It is equivalent to **allocator_type::size_type**.

For type **string**, it is equivalent to **size_t**.

Example

```
// basic_string_size_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ( "Hello world" );

   basic_string <char>::size_type sizeStr1, capStr1;
   sizeStr1 = str1.size (  );
   capStr1 = str1.capacity (  );

   cout << "The current size of string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The capacity of string str1 is: " << capStr1
         << "." << endl;
}
```

Output

```
The current size of string str1 is: 11.
The capacity of string str1 is: 15.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::traits_type

A type for the character traits of the elements stored in a string.

```
typedef Traits traits_type;
```

Remarks

The type is a synonym for the second template parameter **Traits**.

For type **string**, it is equivalent to **char_traits<char>**.

Example

See the example for copy for an example of how to declare and use **traits_type**.

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::value_type

A type that represents the type of characters stored in a string.

```
typedef typename allocator_type::value_type value_type;
```

Remarks

It is equivalent to **traits_type::char_type** and is equivalent to **char** for objects of type **string**.

Example

```
// basic_string_value_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   basic_string<char>::value_type ch1 = 'G';

   char ch2 = 'H';

   cout << "The character ch1 is: " << ch1 << "." << endl;
   cout << "The character ch2 is: " << ch2 << "." << endl;
}
```

Output

```
The character ch1 is: G.
The character ch2 is: H.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string Member Functions

For information about the member functions in the basic_string class, see basic_string Members.

basic_string::append

Adds characters to the end of a string.

```
basic_string& append(
   const value_type* _Ptr
);
basic_string& append(
   const value_type* _Ptr,
   size_type _Count
);
basic_string& append(
   const basic_string& _Str,
   size_type _Off,
   size_type _Count
);
basic_string& append(
   const basic_string& _Str
);
basic_string& append(
   size_type _Count,
   value_type _Ch
);
template<class InputIterator>
   basic_string& append(
      InputIterator _First,
      InputIterator _Last
   );
```

Parameters

_Ptr

The C-string to be appended.

_Str

The string whose characters are to be appended.

_Off

The index of the part of the source string supplying the characters to be appended.

_Count

The number of characters to be appended, at most, from the source string.

_Ch

The character value to be appended.

_First

An input iterator addressing the first element in the range to be appended.

_Last

An input iterator addressing the position of the one beyond the last element in the range to be appended.

Return Value

A reference to the string object that is being appended with the characters passed by the member function.

Remarks

Characters may be appended to a string using the operator+= or the member functions **append** or push_back. **operator+=** appends single-argument values while the multiple-argument **append** member function allows a specific part of a string to be specified for adding.

Example

```
// basic_string_append.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function
   // appending a C-string to a string
   string str1a ( "Hello " );
   cout << "The original string str1 is: " << str1a << endl;
   const char *cstr1a = "Out There ";
   cout << "The C-string cstr1a is: " << cstr1a << endl;
   str1a.append ( cstr1a );
   cout << "Appending the C-string cstr1a to string str1 gives: "
        << str1a << "." << endl << endl;

   // The second member function
   // appending part of a C-string to a string
   string str1b ( "Hello " );
   cout << "The string str1b is: " << str1b << endl;
   const char *cstr1b = "Out There ";
   cout << "The C-string cstr1b is: " << cstr1b << endl;
   str1b.append ( cstr1b , 3 );
   cout << "Appending the 1st part of the C-string cstr1b "
        << "to string str1 gives: " << str1b << "."
        << endl << endl;

   // The third member function
   // appending part of one string to another
   string str1c ( "Hello " ), str2c ( "Wide World " );
   cout << "The string str2c is: " << str2c << endl;
   str1c.append ( str2c , 5 , 5 );
   cout << "The appended string str1 is: "
        << str1c << "." << endl << endl;

   // The fourth member function
   // appending one string to another in two ways,
   // comparing append and operator [ ]
   string str1d ( "Hello " ), str2d ( "Wide " ), str3d ( "World " );
   cout << "The  string str2d is: " << str2d << endl;
   str1d.append ( str2d );
   cout << "The appended string str1d is: "
        << str1d << "." << endl;
   str1d += str3d;
   cout << "The doubly appended strig str1 is: "
        << str1d << "." << endl << endl;

   // The fifth member function
   // appending characters to a string
   string str1e ( "Hello " );
   str1e.append ( 4 , '!' );
   cout << "The string str1 appended with exclamations is: "
        << str1e << endl << endl;

   // The sixth member function
   // appending a range of one string to another
   string str1f ( "Hello " ), str2f ( "Wide World " );
   cout << "The string str2f is: " << str2f << endl;
   str1f.append ( str2f.begin ( ) + 5 , str2f.end ( ) - 1 );
   cout << "The appended string str1 is: "
        << str1f << "." << endl << endl;
}
```

Output

```
The original string str1 is: Hello
The C-string cstr1a is: Out There
Appending the C-string cstr1a to string str1 gives: Hello Out There .

The string str1b is: Hello
The C-string cstr1b is: Out There
Appending the 1st part of the C-string cstr1b to string str1 gives: Hello Out.

The string str2c is: Wide World
The appended string str1 is: Hello World.

The  string str2d is: Wide
The appended string str1d is: Hello Wide .
The doubly appended strig str1 is: Hello Wide World .

The string str1 appended with exclamations is: Hello !!!!

The string str2f is: Wide World
The appended string str1 is: Hello World.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

basic_string::append (STL Samples)

Other Resources

basic_string Members

basic_string::assign

Assigns new character values to the contents of a string.

```
basic_string& assign(
   const value_type* _Ptr
);
basic_string& assign(
   const value_type* _Ptr,
   size_type _Count
);
basic_string& assign(
   const basic_string& _Str,
   size_type off,
   size_type _Count
);
basic_string& assign(
   const basic_string& _Str
);
basic_string& assign(
   size_type _Count,
   value_type _Ch
);
template<class InIt>
   basic_string& assign(
      InputIterator _First,
      InputIterator _Last
   );
```

Parameters

_Ptr

A pointer to the characters of the C-string to be assigned to the target string.

_Count

The number of characters to be appended, at most, from the source string.

_Str

The source string whose characters are to be assigned to the target string.

_Ch

The character value to be assigned.

_First

An input iterator addressing the first character in the range of the source string to be assigned to the target range.

_Last

An input iterator addressing the one beyond the last character in the range of the source string to be assigned to the target range.

off

The position at which new characters will start to be assigned.

Return Value

A reference to the string object that is being assigned new characters by the member function.

Remarks

The strings can be assigned new character values. The new value can be either a string and C-string or a single character. The operator= may be used if the new value can be described by a single parameter; otherwise the member function **assign**, which has multiple parameters, can be used to specify which part of the string is to be assigned to a target string.

Example

```
// basic_string_assign.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function assigning the
   // characters of a C-string to a string
   string str1a;
   const char *cstr1a = "Out There";
   cout << "The C-string cstr1a is: " << cstr1a <<  "." << endl;
   str1a.assign ( cstr1a );
   cout << "Assigning the C-string cstr1a to string str1 gives: "
        << str1a << "." << endl << endl;

   // The second member function assigning a specific
   // number of the of characters a C-string to a string
   string  str1b;
   const char *cstr1b = "Out There";
   cout << "The C-string cstr1b is: " << cstr1b << endl;
   str1b.assign ( cstr1b , 3 );
   cout << "Assigning the 1st part of the C-string cstr1b "
        << "to string str1 gives: " << str1b << "."
        << endl << endl;

   // The third member function assigning a specific number
   // of the characters from one string to another string
   string str1c ( "Hello " ), str2c ( "Wide World " );
   cout << "The string str2c is: " << str2c << endl;
   str1c.assign ( str2c , 5 , 5 );
   cout << "The newly assigned string str1 is: "
        << str1c << "." << endl << endl;

   // The fourth member function assigning the characters
   // from one string to another string in two equivalent
   // ways, comparing the assign and operator =
   string str1d ( "Hello" ), str2d ( "Wide" ), str3d ( "World" );
   cout << "The original string str1 is: " << str1d << "." << endl;
   cout << "The string str2d is: " << str2d << endl;
   str1d.assign ( str2d );
   cout << "The string str1 newly assigned with string str2d is: "
        << str1d << "." << endl;
   cout << "The string str3d is: " << str3d << "." << endl;
   str1d = str3d;
   cout << "The string str1 reassigned with string str3d is: "
        << str1d << "." << endl << endl;

   // The fifth member function assigning a specific
   // number of characters of a certain value to a string
   string str1e ( "Hello " );
   str1e.assign ( 4 , '!' );
   cout << "The string str1 assigned with eclamations is: "
        << str1e << endl << endl;

   // The sixth member function assigning the value from
   // the range of one string to another string
   string str1f ( "Hello " ), str2f ( "Wide World " );
   cout << "The string str2f is: " << str2f << endl;
   str1f.assign ( str2f.begin ( ) + 5 , str2f.end ( ) - 1 );
   cout << "The string str1 assigned a range of string str2f is: "
        << str1f << "." << endl << endl;
}
```

Output

```
The C-string cstr1a is: Out There.
Assigning the C-string cstr1a to string str1 gives: Out There.

The C-string cstr1b is: Out There
Assigning the 1st part of the C-string cstr1b to string str1 gives: Out.

The string str2c is: Wide World
The newly assigned string str1 is: World.

The original string str1 is: Hello.
The string str2d is: Wide
The string str1 newly assigned with string str2d is: Wide.
The string str3d is: World.
The string str1 reassigned with string str3d is: World.

The string str1 assigned with eclamations is: !!!!

The string str2f is: Wide World
The string str1 assigned a range of string str2f is: World.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::at

Provides a reference to the character with a specified index in a string.

```
const_reference at(
   size_type _Off
) const;
reference at(
   size_type _Off
);
```

Parameters

_Off

The index of the position of the element to be referenced.

Return Value

A reference to the character of the string at the position specified by the parameter index.

Remarks

The first element of the string has an index of zero and the following elements are indexed consecutively by the positive integers, so that a string of length *n* has an *n*th element indexed by the number *n* – 1.

The member operator[] is faster than the member function **at** for providing read and write access to the elements of a string.

The member **operator[]** does not check whether the index passed as a parameter is valid but the member function **at** does and so should be used if the validity is not certain. An invalid index, which is an index less that zero or greater than or equal to the size of the string, passed to the member function **at** throws an out_of_range Class exception. An invalid index passed to the **operator[]** results in undefined behavior, but the index equal to the length of the string is a valid index for const strings and the operator returns the null-character when passed this index.

The reference returned may be invalidated by string reallocations or modifications for the non-**const** strings.

Example

```
// basic_string_at.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ( "Hello world" ), str2 ( "Goodbye world" );
   const string  cstr1 ( "Hello there" ), cstr2 ( "Goodbye now" );
   cout << "The original string str1 is: " << str1 << endl;
   cout << "The original string str2 is: " << str2 << endl;

   // Element access to the non const strings
   basic_string <char>::reference refStr1 = str1 [6];
   basic_string <char>::reference refStr2 = str2.at ( 3 );

   cout << "The character with an index of 6 in string str1 is: "
        << refStr1 << "." << endl;
   cout << "The character with an index of 3 in string str2 is: "
        << refStr2 << "." << endl;

   // Element access to the const strings
   basic_string <char>::const_reference crefStr1 = cstr1 [ cstr1.length ( ) ];
   basic_string <char>::const_reference crefStr2 = cstr2.at ( 8 );

   if ( crefStr1 == '\0' )
      cout << "The null character is returned as a valid reference."
           << endl;
```

```
    else
       cout << "The null character is not returned." << endl;
    cout << "The character with index 8 in the const string cstr2 is: "
         << crefStr2 << "." << endl;
 }
```

Output

```
The original string str1 is: Hello world
The original string str2 is: Goodbye world
The character with an index of 6 in string str1 is: w.
The character with an index of 3 in string str2 is: d.
The null character is returned as a valid reference.
The character with index 8 in the const string cstr2 is: n.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::basic_string

Constructs a string that is empty or initialized by specific characters, or that is a copy of all or part of some other string object or C-string.

```
basic_string(
   const value_type* _Ptr,
   size_type _Count,
   const allocator_type& _Al = Allocator ( )
);
basic_string(
   const value_type* _Ptr,
   const allocator_type& _Al = Allocator ( )
);
basic_string(
   const basic_string& _Right,
   size_type _Roff = 0,
   size_type _Count = npos,
   const allocator_type& _Al = Allocator ( )
);
basic_string(
   size_type _Count,
   value_type _Ch,
   const allocator_type& _Al = Allocator ( )
);
explicit basic_string(
   const allocator_type& _Al = Allocator ( )
);
```

Parameters

_Ptr

The C-string whose characters are to be used to initialize the string being constructed. This value cannot be a null pointer.

_Al

The storage allocator class to be used for the string object being constructed, which defaults to **Allocator**.

_Count

The number of characters to be initialized.

_Right

The string whose characters are to be used to initialize the string being constructed.

_Roff

The index of a character in a string that is the first to be used to initialize character values for the string being constructed.

_Ch

The character value to be copied into the string being constructed.

Return Value

A reference to the string object that is being constructed by the member functions.

Remarks

The constructors for class basic_string create and initialize strings as follows:

- The first and second constructors create a string that is initialized by all or part of a C-string.
- The third member function creates a string that is initialized by all or part of an object of type **basic_string**.
- The fourth member function creates a string that is initialized by a specific number of characters of a parameter stipulated value.

- The fifth member function creates an empty string.

Example

```
// basic_string_ctor.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function initializing with a C-string
   const char *cstr1a = "Hello Out There.";
   basic_string <char> str1a ( cstr1a , 5);
   cout << "The string initialized by C-string cstr1a is: "
        << str1a << "." << endl;

   // The second member function initializing with a string
   string  str2a ( "How Do You Do?" );
   basic_string <char> str2b ( str2a , 7 , 7 );
   cout << "The string initialized by part of the string cstr2a is: "
        << str2b << "." << endl;

   // The third member function initializing a string
   // with a number of characters of a specific value
   basic_string <char> str3a ( 5, '9' );
   cout << "The string initialized by five number 9s is: "
        << str3a << endl;

   // The fourth member function creates an empty string
   // and string with a specified allocator
   basic_string <char> str4a;
   string str4b;
   basic_string <char> str4c ( str4b.get_allocator( ) );
   if (str4c.empty ( ) )
      cout << "The string str4c is empty." << endl;
   else
      cout << "The string str4c is not empty." << endl;

   // The fifth member function initializes a string from
   // another range of characters
   string str5a ( "Hello World" );
   basic_string <char> str5b ( str5a.begin ( ) + 5 , str5a.end ( ) );
   cout << "The string initialized by another range is: "
        << str5b << "." << endl;
}
```

Output

```
The string initialized by C-string cstr1a is: Hello.
The string initialized by part of the string cstr2a is: You Do?.
The string initialized by five number 9s is: 99999
The string str4c is empty.
The string initialized by another range is:  World.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::begin

Returns an iterator addressing the first element in the string.

```
const_iterator begin( ) const;
iterator begin( );
```

Return Value

A random-access iterator that addresses the first element of the sequence or just beyond the end of an empty sequence.

Remark

If the return value of **begin** is assigned to a **const_iterator**, the string object cannot be modified. If the return value of **begin** is assigned to an **iterator**, the string object can be modified.

Example

```
// basic_string_begin.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( ) {
   using namespace std;
   string str1 ( "No way out." ), str2;
   basic_string <char>::iterator strp_Iter, str1_Iter, str2_Iter;
   basic_string <char>::const_iterator str1_cIter;

   str1_Iter = str1.begin ( );
   cout << "The first character of the string str1 is: "
        << *str1_Iter << endl;
   cout << "The full original string str1 is: " << str1 << endl;

   // The dereferenced iterator can be used to modify a character
   *str1_Iter = 'G';
   cout << "The first character of the modified str1 is now: "
        << *str1_Iter << endl;
   cout << "The full modified string str1 is now: " << str1 << endl;

   // The following line would be an error because iterator is const
   // *str1_cIter = 'g';

   // For an empty string, begin is equivalent to end
   if (  str2.begin ( ) == str2.end ( ) )
      cout << "The string str2 is empty." << endl;
   else
      cout << "The string str2 is not empty." << endl;
}
```

Output

```
The first character of the string str1 is: N
The full original string str1 is: No way out.
The first character of the modified str1 is now: G
The full modified string str1 is now: Go way out.
The string str2 is empty.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::c_str

Converts the contents of a string as a C-style, null-terminated string.

```
const value_type *c_str( ) const;
```

Return Value

A pointer to the C-style version of the invoking string. The pointer value is not valid after calling a non-const function, including the destructor, in the basic_string class on the object.

Remarks

Objects of type string belonging to the C++ template class basic_string<char> are not necessarily null terminated. The null character ' \0 ' is used as a special character in a C-string to mark the end of the string but has no special meaning in an object of type string and may be a part of the string just like any other character. There is an automatic conversion from const **char*** into strings, but the string class does not provide for automatic conversions from C-style strings to objects of type **basic_string<char>**.

The returned C-style string should not be modified, as this could invalidate the pointer to the string, or deleted, as the string has a limited lifetime and is owned by the class string.

Example

```
// basic_string_c_str.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   string  str1 ( "Hello world" );
   cout << "The original string object str1 is: "
        << str1 << endl;
   cout << "The length of the string object str1 = "
        << str1.length ( ) << endl << endl;

   // Converting a string to an array of characters
   const char *ptr1 = 0;
   ptr1= str1.data ( );
   cout << "The modified string object ptr1 is: " << ptr1
        << endl;
   cout << "The length of character array str1 = "
        << strlen ( ptr1) << endl << endl;

   // Converting a string to a C-style string
   const char *c_str1 = str1.c_str ( );
   cout << "The C-style string c_str1 is: " << c_str1
        << endl;
   cout << "The length of C-style string str1 = "
        << strlen ( c_str1) << endl << endl;
}
```

Output

```
The original string object str1 is: Hello world
The length of the string object str1 = 11

The modified string object ptr1 is: Hello world
The length of character array str1 = 11

The C-style string c_str1 is: Hello world
The length of C-style string str1 = 11
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::capacity

Returns the largest number of elements that could be stored in a string without increasing the memory allocation of the string.

```
size_type capacity( ) const;
```

Return Value

The size of storage currently allocated in memory to hold the string.

Remarks

The member function returns the storage currently allocated to hold the controlled sequence, a value at least as large as size.

Example

```
// basic_string_capacity.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string  str1 ("Hello world");
   cout << "The original string str1 is: " << str1 << endl;

   // The size and length member functions differ in name only
   basic_string <char>::size_type sizeStr1, lenStr1;
   sizeStr1 = str1.size ( );
   lenStr1 = str1.length ( );

   basic_string <char>::size_type capStr1, max_sizeStr1;
   capStr1 = str1.capacity ( );
   max_sizeStr1 = str1.max_size ( );

   // Compare size, length, capacity & max_size of a string
   cout << "The current size of original string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The current length of original string str1 is: "
        << lenStr1 << "." << endl;
   cout << "The capacity of original string str1 is: "
        << capStr1 << "." << endl;
   cout << "The max_size of original string str1 is: "
        << max_sizeStr1 << "." << endl << endl;

   str1.erase ( 6, 5 );
   cout << "The modified string str1 is: " << str1 << endl;

   sizeStr1 = str1.size (  );
   lenStr1 = str1.length (  );
   capStr1 = str1.capacity (  );
   max_sizeStr1 = str1.max_size (  );

   // Compare size, length, capacity & max_size of a string
   // after erasing part of the original string
   cout << "The current size of modified string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The current length of modified string str1 is: "
        << lenStr1 << "." << endl;
   cout << "The capacity of modified string str1 is: "
        << capStr1 << "." << endl;
   cout << "The max_size of modified string str1 is: "
        << max_sizeStr1 << "." << endl;
}
```

Sample Output

The following output is for x86.

```
The original string str1 is: Hello world
The current size of original string str1 is: 11.
The current length of original string str1 is: 11.
The capacity of original string str1 is: 15.
The max_size of original string str1 is: 4294967294.

The modified string str1 is: Hello
The current size of modified string str1 is: 6.
The current length of modified string str1 is: 6.
The capacity of modified string str1 is: 15.
The max_size of modified string str1 is: 4294967294.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::clear

Erases all elements of a string.

```
void clear( );
```

Remarks

The string on which the member function is called will be empty.

Example

```
// basic_string_clear.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string  str1 ("Hello world"), str2;
   basic_string <char>::iterator str_Iter;
   cout << "The original string str1 is: ";
   for ( str_Iter = str1.begin( ); str_Iter != str1.end( ); str_Iter++ )
      cout << *str_Iter;
   cout << endl;

   str1.clear ( );
   cout << "The modified string str1 is: ";
   for ( str_Iter = str1.begin( ); str_Iter != str1.end( ); str_Iter++ )
      cout << *str_Iter;
   cout << endl;

   //For an empty string, begin is equivalent to end
   if ( str1.begin ( ) == str1.end ( ) )
      cout << "Nothing printed above because "
           << "the string str1 is empty." << endl;
   else
      cout << "The string str1 is not empty." << endl;
}
```

Output

```
The original string str1 is: Hello world
The modified string str1 is:
Nothing printed above because the string str1 is empty.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::compare

Performs a case sensitive comparison with a specified string to determine if the two strings are equal or if one is lexicographically less than the other.

```
int compare(
   const basic_string& _Str
) const;
int compare(
   size_type _Pos1,
   size_type _Num1,
   const basic_string& _Str
);
int compare(
   size_type _Pos1,
   size_type _Num1,
   const basic_string& _Str,
   size_type _Off,
   size_type _Count
);
int compare(
   const value_type* _Ptr
) const;
int compare(
   size_type _Pos1,
   size_type _Num1,
   const value_type* _Ptr
) const;
int compare(
   size_type _Pos1,
   size_type _Num1,
   const value_type* __Ptr
   size_type _Num2 = npos
) const;
```

Parameters

_Str

The string that is to be compared to the operand string.

_Pos1

The index of the operand string at which the comparison begins.

_Num1

The maximum number of characters from the operand string to be compared.

_Num2

The maximum number of characters from the parameter string to be compared.

_Off

The index of the parameter string at which the comparison begins.

_Count

The maximum number of characters from the parameter string to be compared.

_Ptr

The C-string to be compared to the operand string.

Return Value

A negative value if the operand string is less than the parameter string; zero if the two strings are equal; or a positive value if

the operand string is greater than the parameter string.

Remarks

The **compare** member functions compare either all or part of the parameter and operand strings depending on which in used.

The comparison performed is case sensitive.

Example

```
// basic_string_compare.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function compares
   // an operand string to a parameter string
   int comp1;
   string s1o ( "CAB" );
   string s1p ( "CAB" );
   cout << "The operand string is: " << s1o << endl;
   cout << "The parameter string is: " << s1p << endl;
   comp1 = s1o.compare ( s1p );
   if ( comp1 < 0 )
      cout << "The operand string is less than "
           << "the parameter string." << endl;
   else if ( comp1 == 0 )
      cout << "The operand string is equal to "
           << "the parameter string." << endl;
   else
      cout << "The operand string is greater than "
           << "the parameter string." << endl;
   cout << endl;

   // The second member function compares part of
   // an operand string to a parameter string
   int comp2a, comp2b;
   string s2o ( "AACAB" );
   string s2p ( "CAB" );
   cout << "The operand string is: " << s2o << endl;
   cout << "The parameter string is: " << s2p << endl;
   comp2a = s2o.compare (  2 , 3 , s2p );
   if ( comp2a < 0 )
      cout << "The last three characters of "
           << "the operand string\n are less than "
           << "the parameter string." << endl;
   else if ( comp2a == 0 )
      cout << "The last three characters of "
           << "the operand string\n are equal to "
           << "the parameter string." << endl;
   else
      cout << "The last three characters of "
           << "the operand string\n is greater than "
           << "the parameter string." << endl;

   comp2b = s2o.compare (  0 , 3 , s2p );
   if ( comp2b < 0 )
      cout << "The first three characters of "
           << "the operand string\n are less than "
           << "the parameter string." << endl;
   else if ( comp2b == 0 )
      cout << "The first three characters of "
           << "the operand string\n are equal to "
           << "the parameter string." << endl;
   else
      cout << "The first three characters of "
```

```
        << "the operand string\n is greater than "
        << "the parameter string." << endl;
cout << endl;

// The third member function compares part of
// an operand string to part of a parameter string
int comp3a;
string s3o ( "AACAB" );
string s3p ( "DCABD" );
cout << "The operand string is: " << s3o << endl;
cout << "The parameter string is: " << s3p << endl;
comp3a = s3o.compare (  2 , 3 , s3p , 1 , 3 );
if ( comp3a < 0 )
   cout << "The three characters from position 2 of "
        << "the operand string are less than\n "
        << "the 3 characters parameter string "
        << "from position 1." << endl;
else if ( comp3a == 0 )
   cout << "The three characters from position 2 of "
        << "the operand string are equal to\n "
        << "the 3 characters parameter string "
        << "from position 1." << endl;
else
   cout << "The three characters from position 2 of "
        << "the operand string is greater than\n "
        << "the 3 characters parameter string "
        << "from position 1." << endl;
cout << endl;

// The fourth member function compares
// an operand string to a parameter C-string
int comp4a;
string s4o ( "ABC" );
const char* cs4p = "DEF";
cout << "The operand string is: " << s4o << endl;
cout << "The parameter C-string is: " << cs4p << endl;
comp4a = s4o.compare ( cs4p );
if ( comp4a < 0 )
   cout << "The operand string is less than "
        << "the parameter C-string." << endl;
else if ( comp4a == 0 )
   cout << "The operand string is equal to "
        << "the parameter C-string." << endl;
else
   cout << "The operand string is greater than "
        << "the parameter C-string." << endl;
cout << endl;

// The fifth member function compares part of
// an operand string to a parameter C-string
int comp5a;
string s5o ( "AACAB" );
const char* cs5p = "CAB";
cout << "The operand string is: " << s5o << endl;
cout << "The parameter string is: " << cs5p << endl;
comp5a = s5o.compare (  2 , 3 , s2p );
if ( comp5a < 0 )
   cout << "The last three characters of "
        << "the operand string\n are less than "
        << "the parameter C-string." << endl;
else if ( comp5a == 0 )
   cout << "The last three characters of "
        << "the operand string\n are equal to "
        << "the parameter C-string." << endl;
else
   cout << "The last three characters of "
        << "the operand string\n is greater than "
        << "the parameter C-string." << endl;
```

```
    cout << endl;

    // The sixth member function compares part of
    // an operand string to part of an equal length of
    // a parameter C-string
    int comp6a;
    string s6o ( "AACAB" );
    const char* cs6p = "ACAB";
    cout << "The operand string is: " << s6o << endl;
    cout << "The parameter C-string is: " << cs6p << endl;
    comp6a = s6o.compare (  1 , 3 , cs6p , 3 );
    if ( comp6a < 0 )
       cout << "The 3 characters from position 1 of "
            << "the operand string are less than\n "
            << "the first 3 characters of the parameter C-string."
            << endl;
    else if ( comp6a == 0 )
       cout << "The 3 characters from position 2 of "
            << "the operand string are equal to\n "
            << "the first 3 characters of the parameter C-string."
            <<  endl;
    else
       cout << "The 3 characters from position 2 of "
            << "the operand string is greater than\n "
            << "the first 3 characters of the parameter C-string."
            << endl;
    cout << endl;
  }
```

Output

```
The operand string is: CAB
The parameter string is: CAB
The operand string is equal to the parameter string.

The operand string is: AACAB
The parameter string is: CAB
The last three characters of the operand string
 are equal to the parameter string.
The first three characters of the operand string
 are less than the parameter string.

The operand string is: AACAB
The parameter string is: DCABD
The three characters from position 2 of the operand string are equal to
 the 3 characters parameter string from position 1.

The operand string is: ABC
The parameter C-string is: DEF
The operand string is less than the parameter C-string.

The operand string is: AACAB
The parameter string is: CAB
The last three characters of the operand string
 are equal to the parameter C-string.

The operand string is: AACAB
The parameter C-string is: ACAB
The 3 characters from position 2 of the operand string are equal to
 the first 3 characters of the parameter C-string.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::copy

Copies at most a specified number of characters from an indexed position in a source string to a target character array.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct. Consider using basic_string::_Copy_s instead.

```
size_type copy(
   value_type* _Ptr,
   size_type _Count,
    size_type _Off = 0
) const;
```

Parameters

_Ptr

The target character array to which the elements are to be copied.

_ Count

The number of characters to be copied, at most, from the source string.

_Off

The beginning position in the source string from which copies are to be made.

Return Value

The number of characters actually copied.

Remarks

A null character is not appended to the end of the copy.

Example

```
// basic_string_copy.cpp
// compile with: /EHsc /W3
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ( "Hello World" );
   basic_string <char>::iterator str_Iter;
   char array1 [ 20 ] = { 0 };
   char array2 [ 10 ] = { 0 };
   basic_string <char>:: pointer array1Ptr = array1;
   basic_string <char>:: value_type *array2Ptr = array2;

   cout << "The original string str1 is: ";
   for ( str_Iter = str1.begin( ); str_Iter != str1.end( ); str_Iter++ )
      cout << *str_Iter;
   cout << endl;

   basic_string <char>:: size_type nArray1;
   // Note: string::copy is potentially unsafe, consider
   // using string::_Copy_s instead.
   nArray1 = str1.copy ( array1Ptr , 12 );  // C4996
   cout << "The number of copied characters in array1 is: "
        << nArray1 << endl;
   cout << "The copied characters array1 is: " << array1 << endl;

   basic_string <char>:: size_type nArray2;
   // Note: string::copy is potentially unsafe, consider
   // using string::_Copy_s instead.
```

```
    nArray2 = str1.copy ( array2Ptr , 5 , 6  );  // C4996
    cout << "The number of copied characters in array2 is: "
            << nArray2 << endl;
    cout << "The copied characters array2 is: " << array2Ptr << endl;
  }
```

Output

```
The original string str1 is: Hello World
The number of copied characters in array1 is: 11
The copied characters array1 is: Hello World
The number of copied characters in array2 is: 5
The copied characters array2 is: World
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::_Copy_s

Copies at most a specified number of characters from an indexed position in a source string to a target character array.

```
size_type _Copy_s(
    value_type *_Dest,
    size_type _Dest_size,
    size_type _Count,
    size_type _Off = 0
) const;
```

Parameters

_Dest

The target character array to which the elements are to be copied.

_Dest_size

The size of *_Dest*.

_Count

The number of characters to be copied, at most, from the source string.

_Off

The beginning position in the source string from which copies are to be made.

Return Value

The number of characters actually copied.

Remarks

A null character is not appended to the end of the copy.

Example

```
// basic_string__Copy_s.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    string str1("Hello World");
    basic_string<char>::iterator str_Iter;
    const int array1_size = 20;
    char array1[array1_size] = { 0 };
    const int array2_size = 10;
    char array2[array2_size] = { 0 };
    basic_string<char>:: pointer array1Ptr = array1;
    basic_string<char>:: value_type *array2Ptr = array2;

    cout << "The original string str1 is: ";
    for (str_Iter = str1.begin(); str_Iter != str1.end(); str_Iter++)
        cout << *str_Iter;
    cout << endl;

    basic_string<char>::size_type nArray1;
    nArray1 = str1._Copy_s(array1Ptr, array1_size, 12);
    cout << "The number of copied characters in array1 is: "
         << nArray1 << endl;
    cout << "The copied characters array1 is: " << array1 << endl;

    basic_string<char>:: size_type nArray2;
```

```
    nArray2 = str1._Copy_s(array2Ptr, array2_size, 5, 6);
    cout << "The number of copied characters in array2 is: "
         << nArray2 << endl;
    cout << "The copied characters array2 is: " << array2Ptr << endl;
}
```

Output

```
The original string str1 is: Hello World
The number of copied characters in array1 is: 11
The copied characters array1 is: Hello World
The number of copied characters in array2 is: 5
The copied characters array2 is: World
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Safe Libraries: Standard C++ Library

Other Resources

basic_string Members

basic_string::data

Converts the contents of a string into an array of characters.

```
const value_type *data( ) const;
```

Return Value

A pointer to the first element of the array containing the contents of the string, or, for an empty array, a non-null pointer that cannot be dereferenced.

Remarks

Objects of type string belonging to the C++ template class basic_string <char> are not necessarily null terminated. The return type for **data** is not a valid C-string, because no null character gets appended. The null character ' \0 ' is used as a special character in a C-string to mark the end of the string, but has no special meaning in an object of type string and may be a part of the string object just like any other character.

There is an automatic conversion from const **char*** into strings, but the string class does not provide for automatic conversions from C-style strings to objects of type **basic_string <char>**.

The returned string should not be modified, because this could invalidate the pointer to the string, or deleted, because the string has a limited lifetime and is owned by the class string.

Example

```
// basic_string_data.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   string str1 ( "Hello world" );
   cout << "The original string object str1 is: "
        << str1 << endl;
   cout << "The length of the string object str1 = "
        << str1.length ( ) << endl << endl;

   // Converting a string to an array of characters
   const char *ptr1 = 0;
   ptr1= str1.data ( );
   cout << "The modified string object ptr1 is: " << ptr1
        << endl;
   cout << "The length of character array str1 = "
        << strlen ( ptr1) << endl << endl;

   // Converting a string to a C-style string
   const char *c_str1 = str1.c_str ( );
   cout << "The C-style string c_str1 is: " << c_str1
        << endl;
   cout << "The length of C-style string str1 = "
        << strlen ( c_str1) << endl << endl;
}
```

Output

```
The original string object str1 is: Hello world
The length of the string object str1 = 11

The modified string object ptr1 is: Hello world
The length of character array str1 = 11
```

```
The C-style string c_str1 is: Hello world
The length of C-style string str1 = 11
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::empty

Tests whether the string is contains characters or not.

```
bool empty( ) const;
```

Return Value

true if the string object contains no characters; **false** if it has at least one character.

Remarks

The member function is equivalent to size == 0.

Example

```
// basic_string_empty.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main() {
   using namespace std;

   bool b1, b2;

   string str1 ("Hello world");
   cout << "The original string object str1 is: " << str1 << endl;
   b1 = str1.empty();
   if (b1)
      cout << "The string object str1 is empty." << endl;
   else
      cout << "The string object str1 is not empty." << endl;
   cout << endl;

   // An example of an empty string object
   string str2;
   b2 = str2.empty();
   if (b2)
      cout << "The string object str2 is empty." << endl;
   else
      cout << "The string object str2 is not empty." << endl;
}
```

Output

```
The original string object str1 is: Hello world
The string object str1 is not empty.

The string object str2 is empty.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::end

Returns an iterator that addresses the location succeeding the last element in a string.

```
const_iterator end( ) const;
iterator end( );
```

Return Value

Returns a random-access iterator that addresses the location succeeding the last element in a string.

Remarks

end is often used to test whether an iterator has reached the end of its string. The value returned by **end** should not be dereferenced.

If the return value of **end** is assigned to a **const_iterator**, the string object cannot be modified. If the return value of **end** is assigned to an **iterator**, the string object can be modified.

Example

```
// basic_string_end.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ( "No way out." ), str2;
   basic_string <char>::iterator str_Iter, str1_Iter, str2_Iter;
   basic_string <char>::const_iterator str1_cIter;

   str1_Iter = str1.end ( );
   str1_Iter--;
   str1_Iter--;
   cout << "The last character-letter of the string str1 is: " << *str1_Iter << endl;
   cout << "The full orginal string str1 is: " << str1 << endl;

   // end used to test when an iterator has reached the end of its string
   cout << "The string is now: ";
   for ( str_Iter = str1.begin( ); str_Iter != str1.end( ); str_Iter++ )
      cout << *str_Iter;
   cout << endl;

   // The dereferenced iterator can be used to modify a character
    *str1_Iter = 'T';
   cout << "The last character-letter of the modified str1 is now: "
        << *str1_Iter << endl;
   cout << "The modified string str1 is now: " << str1 << endl;

   // The following line would be an error because iterator is const
   // *str1_cIter = 'T';

   // For an empty string, end is equivalent to begin
   if ( str2.begin( ) == str2.end ( ) )
      cout << "The string str2 is empty." << endl;
   else
      cout << "The stringstr2  is not empty." << endl;
}
```

Output

```
The last character-letter of the string str1 is: t
The full orginal string str1 is: No way out.
```

```
The string is now: No way out.
The last character-letter of the modified str1 is now: T
The modified string str1 is now: No way ouT.
The string str2 is empty.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::erase

Removes an element or a range of elements in a string from a specified position.

```
iterator erase(
   iterator _First,
   iterator _Last
);
iterator erase(
   iterator _It
);
basic_string& erase(
   size_type _Pos = 0,
   size_type _Count = npos
);
```

Parameters

_First

An iterator addressing the position of the first element in the range to be erased.

_Last

An iterator addressing the position one past the last element in the range to be erased.

_It

An iterator addressing the position of the element in the string to be erased.

_Pos

The index of the first character in the string to be removed.

_Count

The number of elements that will be removed if there are as many in the range of the string beginning with *_Pos*.

Return Value

For the first two member functions, an iterator addressing the first character after the last character removed by the member function. For the third member function, a reference to the string object from which the elements have been erased.

Remarks

The third member function returns ***this**.

Example

```
// basic_string_erase.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The 1st member function using a range demarcated
   // by iterators
   string str1 ( "Hello world" );
   basic_string <char>::iterator str1_Iter;
   cout << "The original string object str1 is: "
        << str1 << "." << endl;
   str1_Iter = str1.erase ( str1.begin ( ) + 3 , str1.end ( ) - 1 );
   cout << "The first element after those removed is: "
        << *str1_Iter << "." << endl;
   cout << "The modified string object str1 is: " << str1
```

```
            << "." << endl << endl;

    // The 2nd member function erasing a char pointed to
    // by an iterator
    string str2 ( "Hello World" );
    basic_string <char>::iterator str2_Iter;
    cout << "The original string object str2 is: " << str2
         << "." << endl;
    str2_Iter = str2.erase ( str2.begin ( ) + 5 );
    cout << "The first element after those removed is: "
         << *str2_Iter << "." << endl;
    cout << "The modified string object str2 is: " << str2
         << "." << endl << endl;

    // The 3rd member function erasing a number of chars
    // after a char
    string str3 ( "Hello computer" ), str3m;
    basic_string <char>::iterator str3_Iter;
    cout << "The original string object str3 is: "
         << str3 << "." << endl;
    str3m = str3.erase ( 6 , 8 );
    cout << "The modified string object str3m is: "
         << str3m << "." << endl;
}
```

Output

```
The original string object str1 is: Hello world.
The first element after those removed is: d.
The modified string object str1 is: Held.

The original string object str2 is: Hello World.
The first element after those removed is: W.
The modified string object str2 is: HelloWorld.

The original string object str3 is: Hello computer.
The modified string object str3m is: Hello .
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::find

Searches a string in a forward direction for the first occurrence of a substring that matches a specified sequence of characters.

```
size_type find(
   value_type _Ch,
   size_type _Off = 0
) const;
size_type find(
   const value_type* _Ptr,
   size_type _Off = 0
) const;
size_type find(
   const value_type* _Ptr,
   size_type _Off = 0,
   size_type _Count
) const;
size_type find(
   const basic_string& _Str,
   size_type _Off = 0
) const;
```

Parameters

_Ch

The character value for which the member function is to search.

_Off

Index of the position at which the search is to begin.

_Ptr

The C-string for which the member function is to search.

_Count

The number of characters, counting forward from the first character, in the C-string for which the member function is to search.

_Str

The string for which the member function is to search.

Return Value

The index of the first character of the substring searched for when successful; otherwise **npos**.

Example

```
// basic_string_find.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function
   // searches for a single character in a string
   string str1 ( "Hello Everyone" );
   cout << "The original string str1 is: " << str1 << endl;
   basic_string <char>::size_type indexCh1a, indexCh1b;

   indexCh1a = str1.find ( "e" , 3 );
   if (indexCh1a != string::npos )
```

```
      cout << "The index of the 1st 'e' found after the 3rd"
           << " position in str1 is: " << indexCh1a << endl;
   else
      cout << "The character 'e' was not found in str1 ." << endl;

   indexCh1b = str1.find ( "x" );
   if (indexCh1b != string::npos )
      cout << "The index of the 'x' found in str1 is: "
           << indexCh1b << endl << endl;
   else
      cout << "The Character 'x' was not found in str1."
           << endl << endl;

   // The second member function searches a string
   // for a substring as specified by a C-string
   string str2 ( "Let me make this perfectly clear." );
   cout << "The original string str2 is: " << str2 << endl;
   basic_string <char>::size_type indexCh2a, indexCh2b;

   const char *cstr2 = "perfect";
   indexCh2a = str2.find ( cstr2 , 5 );
   if ( indexCh2a != string::npos )
      cout << "The index of the 1st element of 'perfect' "
           << "after\n the 5th position in str2 is: "
           << indexCh2a << endl;
   else
      cout << "The substring 'perfect' was not found in str2 ."
           << endl;

   const char *cstr2b = "imperfectly";
   indexCh2b = str2.find ( cstr2b , 0 );
   if (indexCh2b != string::npos )
      cout << "The index of the 1st element of 'imperfect' "
           << "after\n the 5th position in str3 is: "
           << indexCh2b << endl;
   else
      cout << "The substring 'imperfect' was not found in str2 ."
           << endl << endl;

   // The third member function searches a string
   // for a substring as specified by a C-string
   string str3 ( "This is a sample string for this program" );
   cout << "The original string str3 is: " << str3 << endl;
   basic_string <char>::size_type indexCh3a, indexCh3b;

   const char *cstr3a = "sample";
   indexCh3a = str3.find ( cstr3a );
   if ( indexCh3a != string::npos )
      cout << "The index of the 1st element of sample "
           << "in str3 is: " << indexCh3a << endl;
   else
      cout << "The substring 'perfect' was not found in str3 ."
           << endl;

   const char *cstr3b = "for";
   indexCh3b = str3.find ( cstr3b , indexCh3a + 1 , 2 );
   if (indexCh3b != string::npos )
      cout << "The index of the next occurrence of 'for' is in "
           << "str3 begins at: " << indexCh3b << endl << endl;
   else
      cout << "There is no next occurrence of 'for' in str3 ."
           << endl << endl;

   // The fourth member function searches a string
   // for a substring as specified by a string
   string str4 ( "clearly this perfectly unclear." );
   cout << "The original string str4 is: " << str4 << endl;
   basic_string <char>::size_type indexCh4a, indexCh4b;
```

```
    string str4a ( "clear" );
    indexCh4a = str4.find ( str4a , 5 );
    if ( indexCh4a != string::npos )
       cout << "The index of the 1st element of 'clear' "
            << "after\n the 5th position in str4 is: "
            << indexCh4a << endl;
    else
       cout << "The substring 'clear' was not found in str4 ."
            << endl;

    string str4b ( "clear" );
    indexCh4b = str4.find ( str4b );
    if (indexCh4b != string::npos )
       cout << "The index of the 1st element of 'clear' "
            << "in str4 is: "
            << indexCh4b << endl;
    else
       cout << "The substring 'clear' was not found in str4 ."
            << endl << endl;
 }
```

Output

```
The original string str1 is: Hello Everyone
The index of the 1st 'e' found after the 3rd position in str1 is: 8
The Character 'x' was not found in str1.

The original string str2 is: Let me make this perfectly clear.
The index of the 1st element of 'perfect' after
 the 5th position in str2 is: 17
The substring 'imperfect' was not found in str2 .

The original string str3 is: This is a sample string for this program
The index of the 1st element of sample in str3 is: 10
The index of the next occurrence of 'for' is in str3 begins at: 24

The original string str4 is: clearly this perfectly unclear.
The index of the 1st element of 'clear' after
 the 5th position in str4 is: 25
The index of the 1st element of 'clear' in str4 is: 0
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::find_first_not_of

Searches through a string for the first character that is not an element of a specified string.

```
size_type find_first_not_of(
   value_type _Ch,
   size_type _Off = 0
) const;
size_type find_first_not_of(
   const value_type* _Ptr,
   size_type _Off = 0
) const;
size_type find_first_not_of(
   const value_type* _Ptr,
   size_type _Off,
   size_type _Count
) const;
size_type find_first_not_of(
   const basic_string& _Str,
   size_type _Off = 0
) const;
```

Parameters

_Ch

The character value for which the member function is to search.

_Off

Index of the position at which the search is to begin.

_Ptr

The C-string for which the member function is to search.

_Count

The number of characters, counting forward from the first character, in the C-string for which the member function is to search.

_Str

The string for which the member function is to search.

Return Value

The index of the first character of the substring searched for when successful; otherwise **npos**.

Example

```
// basic_string_find_first_not_of.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function
   // searches for a single character in a string
   string str1 ( "xddd-1234-abcd" );
   cout << "The original string str1 is: " << str1 << endl;
   basic_string <char>::size_type indexCh1a, indexCh1b;
   static const basic_string <char>::size_type npos = -1;

   indexCh1a = str1.find_first_not_of ( "d" , 2 );
```

```
   if ( indexCh1a != npos )
      cout << "The index of the 1st 'd' found after the 3rd"
           << " position in str1 is: " << indexCh1a << endl;
   else
      cout << "The character 'd' was not found in str1 ." << endl;

   indexCh1b = str1.find_first_not_of  ( "x" );
   if (indexCh1b != npos )
      cout << "The index of the 'non x' found in str1 is: "
           << indexCh1b << endl << endl;
   else
      cout << "The character 'non x' was not found in str1."
           << endl << endl;

   // The second member function searches a string
   // for a substring as specified by a C-string
   string str2 ( "BBB-1111" );
   cout << "The original string str2 is: " << str2 << endl;
   basic_string <char>::size_type indexCh2a, indexCh2b;

   const char *cstr2 = "B1";
   indexCh2a = str2.find_first_not_of ( cstr2 , 6 );
   if ( indexCh2a != npos )
      cout << "The index of the 1st occurrence of an "
           << "element of 'B1' in str2 after\n the 6th "
           << "position is: " << indexCh2a << endl;
   else
      cout << "Elements of the substring 'B1' were not"
           << "\n found in str2 after the 6th position."
           << endl;

   const char *cstr2b = "B2";
   indexCh2b = str2.find_first_not_of ( cstr2b );
   if ( indexCh2b != npos )
      cout << "The index of the 1st element of 'B2' "
           << "after\n the 0th position in str2 is: "
           << indexCh2b << endl << endl;
   else
      cout << "The substring 'B2' was not found in str2 ."
           << endl << endl << endl;

   // The third member function searches a string
   // for a substring as specified by a C-string
   string str3 ( "444-555-GGG" );
   cout << "The original string str3 is: " << str3 << endl;
   basic_string <char>::size_type indexCh3a, indexCh3b;

   const char *cstr3a = "45G";
   indexCh3a = str3.find_first_not_of ( cstr3a );
   if ( indexCh3a != npos )
      cout << "The index of the 1st occurrence of an "
           << "element in str3\n other than one of the "
           << "characters in '45G' is: " << indexCh3a
           << endl;
   else
      cout << "Elements in str3 contain only characters "
           << " in the string '45G'. "
           << endl;

   const char *cstr3b = "45G";
   indexCh3b = str3.find_first_not_of ( cstr3b , indexCh3a + 1 , 2 );
   if ( indexCh3b != npos )
      cout << "The index of the second occurrence of an "
           << "element of '45G' in str3\n after the 0th "
           << "position is: " << indexCh3b << endl << endl;
   else
      cout << "Elements in str3 contain only characters "
           << " in the string  '45G'. "
```

```
            << endl  << endl;

    // The fourth member function searches a string
    // for a substring as specified by a string
    string str4 ( "12-ab-12-ab" );
    cout << "The original string str4 is: " << str4 << endl;
    basic_string <char>::size_type indexCh4a, indexCh4b;

    string str4a ( "ba3" );
    indexCh4a = str4.find_first_not_of ( str4a , 5 );
    if (indexCh4a != npos )
       cout << "The index of the 1st non occurrence of an "
            << "element of 'ba3' in str4 after\n the 5th "
            << "position is: " << indexCh4a << endl;
    else
       cout << "Elements other than those in the substring"
            << " 'ba3' were not found in the string str4."
            << endl;

    string str4b ( "12" );
    indexCh4b = str4.find_first_not_of ( str4b  );
    if (indexCh4b != npos )
       cout << "The index of the 1st non occurrence of an "
            << "element of '12' in str4 after\n the 0th "
            << "position is: " << indexCh4b << endl;
    else
       cout << "Elements other than those in the substring"
            << " '12' were not found in the string str4."
            << endl;
  }
```

Output

```
The original string str1 is: xddd-1234-abcd
The index of the 1st 'd' found after the 3rd position in str1 is: 4
The index of the 'non x' found in str1 is: 1

The original string str2 is: BBB-1111
Elements of the substring 'B1' were not
 found in str2 after the 6th position.
The index of the 1st element of 'B2' after
 the 0th position in str2 is: 3

The original string str3 is: 444-555-GGG
The index of the 1st occurrence of an element in str3
 other than one of the characters in '45G' is: 3
The index of the second occurrence of an element of '45G' in str3
 after the 0th position is: 7

The original string str4 is: 12-ab-12-ab
The index of the 1st non occurrence of an element of 'ba3' in str4 after
 the 5th position is: 5
The index of the 1st non occurrence of an element of '12' in str4 after
 the 0th position is: 2
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::find_first_of

Searches through a string for the first character that matches any element of a specified string.

```
size_type find_first_not_of(
   value_type _Ch,
   size_type _Off = 0
) const;
size_type find_first_of(
   const value_type* _Ptr,
   size_type _Off = 0
   ) const;
size_type find_first_of(
   const value_type* _Ptr,
   size_type _Off,
   size_type _Count
) const;
size_type find_first_of(
   const basic_string& _Str,
   size_type _Off = 0
) const;
```

Parameters

_Ch

The character value for which the member function is to search.

_Off

Index of the position at which the search is to begin.

_Ptr

The C-string for which the member function is to search.

_Count

The number of characters, counting forward from the first character, in the C-string for which the member function is to search.

_Str

The string for which the member function is to search.

Return Value

The index of the first character of the substring searched for when successful; otherwise **npos**.

Example

```
// basic_string_find_first_of.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function
   // searches for a single character in a string
   string str1 ( "abcd-1234-abcd-1234" );
   cout << "The original string str1 is: " << str1 << endl;
   basic_string <char>::size_type indexCh1a, indexCh1b;
   static const basic_string <char>::size_type npos = -1;

   indexCh1a = str1.find_first_of ( "d" , 5 );
```

```
   if ( indexCh1a != npos )
      cout << "The index of the 1st 'd' found after the 5rd"
           << " position in str1 is: " << indexCh1a << endl;
   else
      cout << "The character 'd' was not found in str1 ." << endl;

   indexCh1b = str1.find_first_of ( "x" );
   if ( indexCh1b != npos )
      cout << "The index of the 'x' found in str1 is: "
           << indexCh1b << endl << endl;
   else
      cout << "The character 'x' was not found in str1."
           << endl << endl;

   // The second member function searches a string
   // for a substring as specified by a C-string
   string str2 ( "ABCD-1234-ABCD-1234" );
   cout << "The original string str2 is: " << str2 << endl;
   basic_string <char>::size_type indexCh2a, indexCh2b;

   const char *cstr2 = "B1";
   indexCh2a = str2.find_first_of ( cstr2 , 6 );
   if ( indexCh2a != npos )
      cout << "The index of the 1st occurrence of an "
           << "element of 'B1' in str2 after\n the 6th "
           << "position is: " << indexCh2a << endl;
   else
      cout << "Elements of the substring 'B1' were not "
           << "found in str2 after the 10th position."
           << endl;

   const char *cstr2b = "D2";
   indexCh2b = str2.find_first_of ( cstr2b );
   if ( indexCh2b != npos )
      cout << "The index of the 1st element of 'D2' "
           << "after\n the 0th position in str2 is: "
           << indexCh2b << endl << endl;
   else
      cout << "The substring 'D2' was not found in str2 ."
           << endl << endl << endl;

   // The third member function searches a string
   // for a substring as specified by a C-string
   string str3 ( "123-abc-123-abc-456-EFG-456-EFG" );
   cout << "The original string str3 is: " << str3 << endl;
   basic_string <char>::size_type indexCh3a, indexCh3b;

   const char *cstr3a = "5G";
   indexCh3a = str3.find_first_of ( cstr3a );
   if ( indexCh3a != npos )
      cout << "The index of the 1st occurrence of an "
           << "element of '5G' in str3 after\n the 0th "
           << "position is: " << indexCh3a << endl;
   else
      cout << "Elements of the substring '5G' were not "
           << "found in str3\n after the 0th position."
           << endl;

   const char *cstr3b = "5GF";
   indexCh3b = str3.find_first_of  ( cstr3b , indexCh3a + 1 , 2 );
   if (indexCh3b != npos )
      cout << "The index of the second occurrence of an "
           << "element of '5G' in str3\n after the 0th "
           << "position is: " << indexCh3b << endl << endl;
   else
      cout << "Elements of the substring '5G' were not "
           << "found in str3\n after the first occurrrence."
           << endl << endl;
```

```
    // The fourth member function searches a string
    // for a substring as specified by a string
    string str4 ( "12-ab-12-ab" );
    cout << "The original string str4 is: " << str4 << endl;
    basic_string <char>::size_type indexCh4a, indexCh4b;

    string str4a ( "ba3" );
    indexCh4a = str4.find_first_of ( str4a , 5 );
    if ( indexCh4a != npos )
       cout << "The index of the 1st occurrence of an "
            << "element of 'ba3' in str4 after\n the 5th "
            << "position is: " << indexCh4a << endl;
    else
       cout << "Elements of the substring 'ba3' were not "
            << "found in str4\n after the 0th position."
            << endl;

    string str4b ( "a2" );
    indexCh4b = str4.find_first_of ( str4b );
    if ( indexCh4b != npos )
       cout << "The index of the 1st occurrence of an "
            << "element of 'a2' in str4 after\n the 0th "
            << "position is: " << indexCh4b << endl;
    else
       cout << "Elements of the substring 'a2' were not "
            << "found in str4\n after the 0th position."
            << endl;
  }
```

Output

```
The original string str1 is: abcd-1234-abcd-1234
The index of the 1st 'd' found after the 5rd position in str1 is: 13
The character 'x' was not found in str1.

The original string str2 is: ABCD-1234-ABCD-1234
The index of the 1st occurrence of an element of 'B1' in str2 after
 the 6th position is: 11
The index of the 1st element of 'D2' after
 the 0th position in str2 is: 3

The original string str3 is: 123-abc-123-abc-456-EFG-456-EFG
The index of the 1st occurrence of an element of '5G' in str3 after
 the 0th position is: 17
The index of the second occurrence of an element of '5G' in str3
 after the 0th position is: 22

The original string str4 is: 12-ab-12-ab
The index of the 1st occurrence of an element of 'ba3' in str4 after
 the 5th position is: 9
The index of the 1st occurrence of an element of 'a2' in str4 after
 the 0th position is: 1
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

basic_string::find_first_of (STL Samples)

Other Resources

basic_string Members

basic_string::find_last_not_of

Searches through a string for the last character that is not any element of a specified string.

```
size_type find_last_not_of(
   value_type _Ch,
   size_type _Off = npos
) const;
size_type find_last_not_of(
   const value_type* _Ptr,
   size_type _Off = npos
) const;
size_type find_last_not_of(
   const value_type* _Ptr,
   size_type _Off,
   size_type _Count
) const;
size_type find_last_not_of(
   const basic_string& _Str,
   size_type _Off = npos
) const;
```

Parameters

_Ch

The character value for which the member function is to search.

_Off

Index of the position at which the search is to finish.

_Ptr

The C-string for which the member function is to search.

_Count

The number of characters, counting forward from the first character, in the C-string for which the member function is to search.

_Str

The string for which the member function is to search.

Return Value

The index of the first character of the substring searched for when successful; otherwise **npos**.

Example

```
// basic_string_find_last_not_of.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function
   // searches for a single character in a string
   string str1 ( "dddd-1dd4-abdd" );
   cout << "The original string str1 is: " << str1 << endl;
   basic_string <char>::size_type indexCh1a, indexCh1b;
   static const basic_string <char>::size_type npos = -1;

   indexCh1a = str1.find_last_not_of ( "d" , 7 );
```

```
   if ( indexCh1a != npos )
      cout << "The index of the last non 'd'\n found before the "
           << "7th position in str1 is: " << indexCh1a << endl;
   else
      cout << "The non 'd' character was not found ." << endl;

   indexCh1b = str1.find_last_not_of  ( "d" );
   if ( indexCh1b != npos )
      cout << "The index of the non 'd' found in str1 is: "
           << indexCh1b << endl << endl;
   else
      cout << "The Character 'non x' was not found in str1."
           << endl << endl;

   // The second member function searches a string
   // for a substring as specified by a C-string
   string str2 ( "BBB-1111" );
   cout << "The original string str2 is: " << str2 << endl;
   basic_string <char>::size_type indexCh2a, indexCh2b;

   const char *cstr2 = "B1";
   indexCh2a = str2.find_last_not_of  ( cstr2 , 6 );
   if ( indexCh2a != npos )
      cout << "The index of the last occurrence of a "
           << "element\n not of 'B1' in str2 before the 6th "
           << "position is: " << indexCh2a << endl;
   else
      cout << "Elements not of the substring 'B1' were not "
           << "\n found in str2 before the 6th position."
           << endl;

   const char *cstr2b = "B-1";
   indexCh2b = str2.find_last_not_of  ( cstr2b );
   if ( indexCh2b != npos )
      cout << "The index of the last element not "
           << "in 'B-1'\n is: "
           << indexCh2b << endl << endl;
   else
      cout << "The elements of the substring 'B-1' were "
           << "not found in str2 ."
           << endl << endl;

   // The third member function searches a string
   // for a substring as specified by a C-string
   string str3 ( "444-555-GGG" );
   cout << "The original string str3 is: " << str3 << endl;
   basic_string <char>::size_type indexCh3a, indexCh3b;

   const char *cstr3a = "45G";
   indexCh3a = str3.find_last_not_of ( cstr3a );
   if ( indexCh3a != npos )
      cout << "The index of the last occurrence of an "
           << "element in str3\n other than one of the "
           << "characters in '45G' is: " << indexCh3a
           << endl;
   else
      cout << "Elements in str3 contain only characters "
           << " in the string  '45G'. "
           << endl;

   const char *cstr3b = "45G";
   indexCh3b = str3.find_last_not_of ( cstr3b , 6 , indexCh3a - 1 );
   if (indexCh3b != npos )
      cout << "The index of the penultimate occurrence of an "
           << "element\n not in '45G' in str3 is: "
           << indexCh3b << endl << endl;
   else
      cout << "Elements in str3 contain only characters "
```

```
            << " in the string '45G'. "
            << endl  << endl;

    // The fourth member function searches a string
    // for a substring as specified by a string
    string str4 ( "12-ab-12-ab" );
    cout << "The original string str4 is: " << str4 << endl;
    basic_string <char>::size_type indexCh4a, indexCh4b;

    string str4a ( "b-a" );
    indexCh4a = str4.find_last_not_of  ( str4a , 5 );
    if ( indexCh4a != npos )
       cout << "The index of the last occurrence of an "
            << "element not\n in 'b-a' in str4 before the 5th "
            << "position is: " << indexCh4a << endl;
    else
       cout << "Elements other than those in the substring"
            << " 'b-a' were not found in the string str4."
            << endl;

    string str4b ( "12" );
    indexCh4b = str4.find_last_not_of ( str4b  );
    if ( indexCh4b != npos )
       cout << "The index of the last occurrence of an "
            << "element not in '12'\n in str4 before the end "
            << "position is: " << indexCh4b << endl;
    else
       cout << "Elements other than those in the substring"
            << " '12'\n were not found in the string str4."
            << endl;
  }
```

Output

```
The original string str1 is: dddd-1dd4-abdd
The index of the last non 'd'
 found before the 7th position in str1 is: 5
The index of the non 'd' found in str1 is: 11

The original string str2 is: BBB-1111
The index of the last occurrence of a element
 not of 'B1' in str2 before the 6th position is: 3
The elements of the substring 'B-1' were not found in str2 .

The original string str3 is: 444-555-GGG
The index of the last occurrence of an element in str3
 other than one of the characters in '45G' is: 7
The index of the penultimate occurrence of an element
 not in '45G' in str3 is: 3

The original string str4 is: 12-ab-12-ab
The index of the last occurrence of an element not
 in 'b-a' in str4 before the 5th position is: 1
The index of the last occurrence of an element not in '12'
 in str4 before the end position is: 10
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::find_last_of

Searches through a string for the last character that matches any element of a specified string.

```
size_type find_last_of(
   value_type _Ch,
   size_type _Off = npos
) const;

size_type find_last_of(
   const value_type* _Ptr,
   size_type _Off = npos
) const;

size_type find_last_of(
   const value_type* _Ptr,
   size_type _Off,
   size_type _Count
) const;

size_type find_last_of(
   const basic_string& _Str,
   size_type _Off = npos
) const;
```

Parameters

_Ch

The character value for which the member function is to search.

_Off

Index of the position at which the search is to finish.

_Ptr

The C-string for which the member function is to search.

_Count

The number of characters, counting forward from the first character, in the C-string for which the member function is to search.

_Str

The string for which the member function is to search.

Return Value

The index of the last character of the substring searched for when successful; otherwise **npos**.

Example

```
// basic_string_find_last_of.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function
   // searches for a single character in a string
   string str1 ( "abcd-1234-abcd-1234" );
   cout << "The original string str1 is: " << str1 << endl;
   basic_string <char>::size_type indexCh1a, indexCh1b;
```

```
static const basic_string <char>::size_type npos = -1;

indexCh1a = str1.find_last_of ( "d" , 14 );
if ( indexCh1a != npos )
   cout << "The index of the last 'd' found before the 14th"
        << " position in str1 is: " << indexCh1a << endl;
else
   cout << "The character 'd' was not found in str1 ." << endl;

indexCh1b = str1.find_first_of ( "x" );
if ( indexCh1b != npos )
   cout << "The index of the 'x' found in str1 is: "
        << indexCh1b << endl << endl;
else
   cout << "The character 'x' was not found in str1."
        << endl << endl;

// The second member function searches a string
// for a substring as specified by a C-string
string str2 ( "ABCD-1234-ABCD-1234" );
cout << "The original string str2 is: " << str2 << endl;
basic_string <char>::size_type indexCh2a, indexCh2b;

const char *cstr2 = "B1";
indexCh2a = str2.find_last_of  ( cstr2 , 12 );
if (indexCh2a != npos )
   cout << "The index of the last occurrence of an "
        << "element of 'B1' in str2 before\n the 12th "
        << "position is: " << indexCh2a << endl;
else
   cout << "Elements of the substring 'B1' were not "
        << "found in str2 before the 12th position."
        << endl;

const char *cstr2b = "D2";
indexCh2b = str2.find_last_of  ( cstr2b );
if ( indexCh2b != npos )
   cout << "The index of the last element of 'D2' "
        << "after\n the 0th position in str2 is: "
        << indexCh2b << endl << endl;
else
   cout << "The substring 'D2' was not found in str2 ."
        << endl << endl << endl;

// The third member function searches a string
// for a substring as specified by a C-string
string str3 ( "456-EFG-456-EFG" );
cout << "The original string str3 is: " << str3 << endl;
basic_string <char>::size_type indexCh3a;

const char *cstr3a = "5E";
indexCh3a = str3.find_last_of ( cstr3a , 8 , 8 );
if ( indexCh3a != npos )
   cout << "The index of the last occurrence of an "
        << "element of '5E' in str3 before\n the 8th "
        << "position is: " << indexCh3a << endl << endl;
else
   cout << "Elements of the substring '5G' were not "
        << "found in str3\n before the 8th position."
        << endl << endl;

// The fourth member function searches a string
// for a substring as specified by a string
string str4 ( "12-ab-12-ab" );
cout << "The original string str4 is: " << str4 << endl;
basic_string <char>::size_type indexCh4a, indexCh4b;

string str4a ( "ba3" );
```

```
    indexCh4a = str4.find_last_of  ( str4a , 8 );
    if ( indexCh4a != npos )
       cout << "The index of the last occurrence of an "
            << "element of 'ba3' in str4 before\n the 8th "
            << "position is: " << indexCh4a << endl;
    else
       cout << "Elements of the substring 'ba3' were not "
            << "found in str4\n after the 0th position."
            << endl;

    string str4b ( "a2" );
    indexCh4b = str4.find_last_of ( str4b  );
    if ( indexCh4b != npos )
       cout << "The index of the last occurrence of an "
            << "element of 'a2' in str4 before\n the 0th "
            << "position is: " << indexCh4b << endl;
    else
       cout << "Elements of the substring 'a2' were not "
            << "found in str4\n after the 0th position."
            << endl;
  }
```

Output

```
The original string str1 is: abcd-1234-abcd-1234
The index of the last 'd' found before the 14th position in str1 is: 13
The character 'x' was not found in str1.

The original string str2 is: ABCD-1234-ABCD-1234
The index of the last occurrence of an element of 'B1' in str2 before
 the 12th position is: 11
The index of the last element of 'D2' after
 the 0th position in str2 is: 16

The original string str3 is: 456-EFG-456-EFG
The index of the last occurrence of an element of '5E' in str3 before
 the 8th position is: 4

The original string str4 is: 12-ab-12-ab
The index of the last occurrence of an element of 'ba3' in str4 before
 the 8th position is: 4
The index of the last occurrence of an element of 'a2' in str4 before
 the 0th position is: 9
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::get_allocator

Returns a copy of the allocator object used to construct the string.

```
allocator_type get_allocator( ) const;
```

Return Value

The allocator used by the string.

Remarks

The member function returns the stored allocator object.

Allocators for the string class specify how the class manages storage. The default allocators supplied with container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```
// basic_string_get_allocator.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   // The following lines declare objects
   // that use the default allocator.
   string s1;
   basic_string <char> s2;
   basic_string <char, char_traits< char >, allocator< char > > s3;

   // s4 will use the same allocator class as s1
   basic_string <char> s4( s1.get_allocator ( ) );

   basic_string <char>::allocator_type xchar = s1.get_allocator( );
   // You can now call functions on the allocator class xchar used by s1
}
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::insert

Inserts an element or a number of elements or a range of elements into the string at a specified position.

```
basic_string& insert(
   size_type _P0,
   const value_type* _Ptr
);
basic_string& insert(
   size_type _P0,
   const value_type* _Ptr,
   size_type _Count
);
basic_string& insert(
   size_type _P0,
   const basic_string& _Str
);
basic_string& insert(
   size_type _P0,
   const basic_string& _Str,
   size_type _Off,
   size_type _Count
);
basic_string& insert(
   size_type _P0,
   size_type _Count,
   value_type _Ch
);
iterator insert(
   iterator _It,
   value_type _Ch = value_type( )
);
template<class InputIterator>
   void insert(
      iterator _It,
      InputIterator _First,
      InputIterator _Last
   );
void insert(
   iterator _It,
   size_type _Count,
   value_type _Ch
);
```

Parameters

_P0

The index of the position behind the point of insertion the new characters.

_Ptr

The C-string to be wholly or partly inserted into the string.

_Count

The number of characters to be inserted.

_Str

The string to be wholly or partly inserted into the target string.

_Off

The index of the part of the source string supplying the characters to be appended.

_Ch

The character value of the elements to be inserted.

_It

An iterator addressing the position behind which a character is to be inserted.

_First

An input iterator addressing the first element in the source range to be inserted.

_Last

An input iterator addressing the position of the one beyond the last element in the source range to be inserted.

Return Value

Either a reference to the string object that is being assigned new characters by the member function or, in the case of individual character insertions, an iterator addressing the position of the character inserted, or none, depending on the particular member function.

Example

```
// basic_string_insert.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function inserting a C-string
   // at a given position
   basic_string <char> str1a ( "way" );
   const char *cstr1a = "a";
   str1a.insert ( 0, cstr1a );
   cout << "The string with a C-string inserted at position 0 is: "
        << str1a << "." << endl;

   // The second member function inserting a C-string
   // at a given position for a specified number of elements
   basic_string <char> str2a ( "Good" );
   const char *cstr2a = "Bye Bye Baby";
   str2a.insert ( 4, cstr2a ,3 );
   cout << "The string with a C-string inserted at the end is: "
        << str2a << "." << endl;

   // The third member function inserting a string
   // at a given position
   basic_string <char> str3a ( "Bye" );
   string str3b ( "Good" );
   str3a.insert ( 0, str3b );
   cout << "The string with a string inserted at position 0 is: "
        << str3a << "." << endl;

   // The fourth member function inserting part of
   // a string at a given position
   basic_string <char> str4a ( "Good " );
   string str4b ( "Bye Bye Baby" );
   str4a.insert ( 5, str4b , 8 , 4 );
   cout << "The string with part of a string inserted at position 4 is: "
        << str4a << "." << endl;


   // The fifth member function inserts a number of characters
   // at a specified position in the string
   string str5 ( "The number is: ." );
   str5.insert ( 15 , 3 , '3' );
   cout << "The string with characters inserted is: "
        << str5 << endl;
```

```
    // The sixth member function inserts a character
    // at a specified position in the string
    string str6 ( "ABCDFG" );
    basic_string <char>::iterator str6_Iter = ( str6.begin ( ) + 4 );
    str6.insert ( str6_Iter , 'e' );
    cout << "The string with a character inserted is: "
         << str6 << endl;

    // The seventh member function inserts a range
    // at a specified position in the string
    string str7a ( "ABCDHIJ" );
    string str7b ( "abcdefgh" );
    basic_string <char>::iterator str7a_Iter = (str7a.begin ( ) + 4 );
    str7a.insert ( str7a_Iter , str7b.begin ( ) + 4 , str7b.end ( ) -1 );
    cout << "The string with a character inserted from a range is: "
         << str7a << endl;

    // The eigth member function inserts a number of
    // characters at a specified position in the string
    string str8 ( "ABCDHIJ" );
    basic_string <char>::iterator str8_Iter = ( str8.begin ( ) + 4 );
    str8.insert ( str8_Iter , 3 , 'e' );
    cout << "The string with a character inserted from a range is: "
         << str8 << endl;
}
```

Output

```
The string with a C-string inserted at position 0 is: away.
The string with a C-string inserted at the end is: GoodBye.
The string with a string inserted at position 0 is: GoodBye.
The string with part of a string inserted at position 4 is: Good Baby.
The string with characters inserted is: The number is: 333.
The string with a character inserted is: ABCDeFG
The string with a character inserted from a range is: ABCDefgHIJ
The string with a character inserted from a range is: ABCDeeeHIJ
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::length

Returns the current number of elements in a string.

```
size_type length( ) const;
```

Remarks

The member function is the same as size.

Example

```
// basic_string_length.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ("Hello world");
   cout << "The original string str1 is: " << str1 << endl;

   // The size and length member functions differ in name only
   basic_string <char>::size_type sizeStr1, lenStr1;
   sizeStr1 = str1.size ( );
   lenStr1 = str1.length ( );

   basic_string <char>::size_type capStr1, max_sizeStr1;
   capStr1 = str1.capacity ( );
   max_sizeStr1 = str1.max_size ( );

   // Compare size, length, capacity & max_size of a string
   cout << "The current size of original string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The current length of original string str1 is: "
        << lenStr1 << "." << endl;
   cout << "The capacity of original string str1 is: "
        << capStr1 << "." << endl;
   cout << "The max_size of original string str1 is: "
        << max_sizeStr1 << "." << endl << endl;

   str1.erase ( 6, 5 );
   cout << "The modified string str1 is: " << str1 << endl;

   sizeStr1 = str1.size ( );
   lenStr1 = str1.length ( );
   capStr1 = str1.capacity ( );
   max_sizeStr1 = str1.max_size ( );

   // Compare size, length, capacity & max_size of a string
   // after erasing part of the original string
   cout << "The current size of modified string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The current length of modified string str1 is: "
        << lenStr1 << "." << endl;
   cout << "The capacity of modified string str1 is: "
        << capStr1 << "." << endl;
   cout << "The max_size of modified string str1 is: "
        << max_sizeStr1 << "." << endl;
}
```

Sample Output

The following output is for x86.

```
The original string str1 is: Hello world
The current size of original string str1 is: 11.
The current length of original string str1 is: 11.
The capacity of original string str1 is: 15.
The max_size of original string str1 is: 4294967294.

The modified string str1 is: Hello
The current size of modified string str1 is: 6.
The current length of modified string str1 is: 6.
The capacity of modified string str1 is: 15.
The max_size of modified string str1 is: 4294967294.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::max_size

Returns the maximum number of characters a string could contain.

```
size_type max_size( ) const;
```

Return Value

The maximum number of characters a string could contain.

Remarks

A exception of type length_error Class is thrown when an operation produces a string with a length greater than the maximum size.

Example

```
// basic_string_max_size.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ("Hello world");
   cout << "The original string str1 is: " << str1 << endl;

   // The size and length member functions differ in name only
   basic_string <char>::size_type sizeStr1, lenStr1;
   sizeStr1 = str1.size ( );
   lenStr1 = str1.length ( );

   basic_string <char>::size_type capStr1, max_sizeStr1;
   capStr1 = str1.capacity ( );
   max_sizeStr1 = str1.max_size ( );

   // Compare size, length, capacity & max_size of a string
   cout << "The current size of original string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The current length of original string str1 is: "
        << lenStr1 << "." << endl;
   cout << "The capacity of original string str1 is: "
        << capStr1 << "." << endl;
   cout << "The max_size of original string str1 is: "
        << max_sizeStr1 << "." << endl << endl;

   str1.erase ( 6, 5 );
   cout << "The modified string str1 is: " << str1 << endl;

   sizeStr1 = str1.size ( );
   lenStr1 = str1.length ( );
   capStr1 = str1.capacity ( );
   max_sizeStr1 = str1.max_size ( );

   // Compare size, length, capacity & max_size of a string
   // after erasing part of the original string
   cout << "The current size of modified string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The current length of modified string str1 is: "
        << lenStr1 << "." << endl;
   cout << "The capacity of modified string str1 is: "
        << capStr1 << "." << endl;
   cout << "The max_size of modified string str1 is: "
        << max_sizeStr1 << "." << endl;
```

```
}
```

Sample Output

The following output is for x86.

```
The original string str1 is: Hello world
The current size of original string str1 is: 11.
The current length of original string str1 is: 11.
The capacity of original string str1 is: 15.
The max_size of original string str1 is: 4294967294.

The modified string str1 is: Hello
The current size of modified string str1 is: 6.
The current length of modified string str1 is: 6.
The capacity of modified string str1 is: 15.
The max_size of modified string str1 is: 4294967294.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::push_back

Adds an element to the end of the string.

```
void push_back(
   value_type _Ch
);
```

Parameters

_Ch

The character to be added to the end of the string.

Remarks

The member function effectively calls insert(end, *_Ch*).

Example

```
// basic_string_push_back.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ( "abc" );
   basic_string <char>::iterator str_Iter, str1_Iter;

   cout << "The original string str1 is: ";
   for ( str_Iter = str1.begin( ); str_Iter != str1.end( ); str_Iter++ )
      cout << *str_Iter;
   cout << endl;

   // str1.push_back ( 'd' );
   str1_Iter = str1.end ( );
   str1_Iter--;
   cout << "The last character-letter of the modified str1 is now: "
        << *str1_Iter << endl;

   cout << "The modified string str1 is: ";
   for ( str_Iter = str1.begin( ); str_Iter != str1.end( ); str_Iter++ )
      cout << *str_Iter;
   cout << endl;
}
```

Output

```
The original string str1 is: abc
The last character-letter of the modified str1 is now: c
The modified string str1 is: abc
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::rbegin

Returns an iterator to the first element in a reversed string.

```
const_reverse_iterator rbegin( ) const;
reverse_iterator rbegin( );
```

Return Value

Returns a random-access iterator to the first element in a reversed string, addressing what would be the last element in the corresponding unreversed string.

Remarks

rbegin is used with a reversed string just as begin is used with a string.

If the return value of **rbegin** is assigned to a **const_reverse_iterator**, the string object cannot be modified. If the return value of **rbegin** is assigned to a **reverse_iterator**, the string object can be modified.

rbegin can be used to initialize an iteration through a string backwards.

Example

```
// basic_string_rbegin.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ( "Able was I ere I saw Elba" ), str2;
   basic_string <char>::reverse_iterator str_rIter, str1_rIter, str2_rIter;
   basic_string <char>::const_reverse_iterator str1_rcIter;

   str1_rIter = str1.rbegin ( );
   // str1_rIter--;
   cout << "The first character-letter of the reversed string str1 is: "
        << *str1_rIter << endl;
   cout << "The full reversed string str1 is:\n ";
   for ( str_rIter = str1.rbegin( ); str_rIter != str1.rend( ); str_rIter++ )
      cout << *str_rIter;
   cout << endl;

   // The dereferenced iterator can be used to modify a character
    *str1_rIter = 'A';
   cout << "The first character-letter of the modified str1 is now: "
        << *str1_rIter << endl;
   cout << "The full modified reversed string str1 is now:\n ";
   for ( str_rIter = str1.rbegin( ); str_rIter != str1.rend( ); str_rIter++ )
      cout << *str_rIter;
   cout << endl;

   // The following line would be an error because iterator is const
   // *str1_rcIter = 'A';

   // For an empty string, begin is equivalent to end
   if ( str2.rbegin( ) == str2.rend ( ) )
      cout << "The string str2 is empty." << endl;
   else
      cout << "The stringstr2  is not empty." << endl;
}
```

Output

```
The first character-letter of the reversed string str1 is: a
The full reversed string str1 is:
 ablE was I ere I saw elbA
The first character-letter of the modified str1 is now: A
The full modified reversed string str1 is now:
 AblE was I ere I saw elbA
The string str2 is empty.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::rend

Returns an iterator that addresses the location succeeding the last element in a reversed string.

```
const_reverse_iterator rend( ) const;
reverse_iterator rend( );
```

Return Value

A reverse random-access iterator that addresses the location succeeding the last element in a reversed string.

Remarks

rend is used with a reversed string just as end is used with a string.

If the return value of **rend** is assigned to a **const_reverse_iterator**, the string object cannot be modified. If the return value of **rend** is assigned to a **reverse_iterator**, the string object can be modified.

rend can be used to test whether a reverse iterator has reached the end of its string.

The value returned by **rend** should not be dereferenced.

Example

```
// basic_string_rend.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ("Able was I ere I saw Elba"), str2;
   basic_string <char>::reverse_iterator str_rIter, str1_rIter, str2_rIter;
   basic_string <char>::const_reverse_iterator str1_rcIter;

   str1_rIter = str1.rend ( );
   str1_rIter--;
   cout << "The last character-letter of the reversed string str1 is: "
        << *str1_rIter << endl;
   cout << "The full reversed string str1 is:\n ";
   for ( str_rIter = str1.rbegin( ); str_rIter != str1.rend( ); str_rIter++ )
      cout << *str_rIter;
   cout << endl;

   // The dereferenced iterator can be used to modify a character
    *str1_rIter = 'o';
   cout << "The last character-letter of the modified str1 is now: "
        << *str1_rIter << endl;
   cout << "The full modified reversed string str1 is now:\n ";
   for ( str_rIter = str1.rbegin( ); str_rIter != str1.rend( ); str_rIter++ )
      cout << *str_rIter;
   cout << endl;

   // The following line would be an error because iterator is const
   // *str1_rcIter = 'T';

   // For an empty string, end is equivalent to begin
   if ( str2.rbegin( ) == str2.rend ( ) )
      cout << "The string str2 is empty." << endl;
   else
      cout << "The stringstr2  is not empty." << endl;
}
```

Output

```
The last character-letter of the reversed string str1 is: A
The full reversed string str1 is:
 ablE was I ere I saw elbA
The last character-letter of the modified str1 is now: o
The full modified reversed string str1 is now:
 ablE was I ere I saw elbo
The string str2 is empty.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::replace

Replaces elements in a string at a specified position with specified characters or characters copied from other ranges or strings or C-strings.

```
basic_string& replace(
   size_type _Pos1,
   size_type _Num1,
   const value_type* _Ptr
);
basic_string& replace(
   size_type _Pos1,
   size_type _Num1,
   const basic_string& _Str
);
basic_string& replace(
   size_type _Pos1,
   size_type _Num1,
   const value_type* _Ptr,
   size_type _Num2
);
basic_string& replace(
   size_type _Pos1,
   size_type _Num1,
   const basic_string& _Str,
   size_type _Pos2,
   size_type _Num2
);
basic_string& replace(
   size_type _Pos1,
   size_type _Num1,
   size_type _Count,
   value_type _Ch
);
basic_string& replace(
   iterator _First0,
   iterator _Last0,
   const value_type* _Ptr
);
basic_string& replace(
   iterator _First0,
   iterator _Last0,
   const basic_string& _Str
);
basic_string& replace(
   iterator _First0,
   iterator _Last0,
   const value_type* _Ptr,
   size_type _Num2
);
basic_string& replace(
   iterator _First0,
   iterator _Last0,
   size_type _Num2,
   value_type _Ch
);
template<class InputIterator>
   basic_string& replace(
      iterator _First0,
      iterator _Last0,
      InputIterator _First,
      InputIterator _Last
   );
```

Parameters

_Str

The string that is to be a source of characters for the operand string.

_Pos1

The index of the operand string at which the replacement begins.

_Num1

The maximum number of characters to be replaced in the operand string.

_Pos2

The index of the parameter string at which the copying begins.

_Num2

The maximum number of characters to be used from the parameter C-string.

_Ptr

The C-string that is to be a source of characters for the operand string.

_Ch

The character to be copied into the operand string.

_First0

An iterator addressing the first character to be removed in the operand string.

_Last0

An iterator addressing the last character to be removed in the operand string.

_First

An iterator addressing the first character to be copied in the parameter string.

_Last

An iterator addressing the last character to be copied in the parameter string.

Return Value

The operand string with the replacement made.

Example

```
// basic_string_replace.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first two member functions replace
   // part of the operand string with
   // characters from a parameter string or C-string
   string result1a, result1b;
   string s1o ( "AAAAAAAA" );
   string s1p ( "BBB" );
   const char* cs1p = "CCC";
   cout << "The operand string s1o is: " << s1o << endl;
   cout << "The parameter string s1p is: " << s1p << endl;
   cout << "The parameter C-string cs1p is: " << cs1p << endl;
   result1a = s1o.replace ( 1 , 3 , s1p );
   cout << "The result of s1o.replace ( 1 , 3 , s1p )\n is "
        << "the string: " << result1a << "." << endl;
   result1b = s1o.replace ( 5 , 3 , cs1p );
   cout << "The result of s1o.replace ( 5 , 3 , cs1p )\n is "
```

```
     << "the string: " << result1b << "." << endl;
cout << endl;

// The third & fourth member function replace
// part of the operand string with characters
// form part of a parameter string or C-string
string result2a, result2b;
string s2o ( "AAAAAAAA" );
string s2p ( "BBB" );
const char* cs2p = "CCC";
cout << "The operand string s2o is: " << s2o << endl;
cout << "The parameter string s1p is: " << s2p << endl;
cout << "The parameter C-string cs2p is: " << cs2p << endl;
result2a = s2o.replace ( 1 , 3 , s2p , 1 , 2 );
cout << "The result of s2o.replace (1, 3, s2p, 1, 2)\n is "
     << "the string: " << result2a << "." << endl;
result2b = s2o.replace ( 4 , 3 , cs2p , 1 );
cout << "The result of s2o.replace (4 ,3 ,cs2p)\n is "
     << "the string: " << result2b << "." << endl;
cout << endl;

// The fifth member function replaces
// part of the operand string with characters
string result3a;
string s3o ( "AAAAAAAA" );
char ch3p = 'C';
cout << "The operand string s3o is: " << s3o << endl;
cout << "The parameter character c1p is: " << ch3p << endl;
result3a = s3o.replace ( 1 , 3 , 4 , ch3p );
cout << "The result of s3o.replace(1, 3, 4, ch3p)\n is "
     << "the string: " << result3a << "." << endl;
cout << endl;

// The sixth & seventh member functions replace
// part of the operand string, delineated with iterators,
// with a parameter string or C-string
string s4o ( "AAAAAAAA" );
string s4p ( "BBB" );
const char* cs4p = "CCC";
cout << "The operand string s4o is: " << s4o << endl;
cout << "The parameter string s4p is: " << s4p << endl;
cout << "The parameter C-string cs4p is: " << cs4p << endl;
basic_string<char>::iterator IterF0, IterL0;
IterF0 = s4o.begin ( );
IterL0 = s4o.begin ( ) + 3;
string result4a, result4b;
result4a = s4o.replace ( IterF0 , IterL0 , s4p );
cout << "The result of s1o.replace (IterF0, IterL0, s4p)\n is "
     << "the string: " << result4a << "." << endl;
result4b = s4o.replace ( IterF0 , IterL0 , cs4p );
cout << "The result of s4o.replace (IterF0, IterL0, cs4p)\n is "
     << "the string: " << result4b << "." << endl;
cout << endl;

// The 8th member function replaces
// part of the operand string delineated with iterators
// with a number of characters from a parameter C-string
string s5o ( "AAAAAAAF" );
const char* cs5p = "CCCBB";
cout << "The operand string s5o is: " << s5o << endl;
cout << "The parameter C-string cs5p is: " << cs5p << endl;
basic_string<char>::iterator IterF1, IterL1;
IterF1 = s5o.begin ( );
IterL1 = s5o.begin ( ) + 4;
string result5a;
result5a = s5o.replace ( IterF1 , IterL1 , cs5p , 4 );
cout << "The result of s5o.replace (IterF1, IterL1, cs4p ,4)\n is "
     << "the string: " << result5a << "." << endl;
```

```
    cout << endl;

    // The 9th member function replaces
    // part of the operand string delineated with iterators
    // with specified characters
    string s6o ( "AAAAAAAG" );
    char ch6p = 'q';
    cout << "The operand string s6o is: " << s6o << endl;
    cout << "The parameter character ch6p is: " << ch6p << endl;
    basic_string<char>::iterator IterF2, IterL2;
    IterF2 = s6o.begin ( );
    IterL2 = s6o.begin ( ) + 3;
    string result6a;
    result6a = s6o.replace ( IterF2 , IterL2 , 4 , ch6p );
    cout << "The result of s6o.replace (IterF1, IterL1, 4, ch6p)\n is "
         << "the string: " << result6a << "." << endl;
    cout << endl;

    // The 10th member function replaces
    // part of the operand string delineated with iterators
    // with part of a parameter string delineated with iterators
    string s7o ( "OOOOOOO" );
    string s7p ( "PPPP" );
    cout << "The operand string s7o is: " << s7o << endl;
    cout << "The parameter string s7p is: " << s7p << endl;
    basic_string<char>::iterator IterF3, IterL3, IterF4, IterL4;
    IterF3 = s7o.begin ( ) + 1;
    IterL3 = s7o.begin ( ) + 3;
    IterF4 = s7p.begin ( );
    IterL4 = s7p.begin ( ) + 2;
    string result7a;
    result7a = s7o.replace ( IterF3 , IterL3 , IterF4 , IterL4 );
    cout << "The result of s7o.replace (IterF3 ,IterL3 ,IterF4 ,IterL4)\n is "
         << "the string: " << result7a << "." << endl;
    cout << endl;
  }
```

Output

```
The operand string s1o is: AAAAAAAA
The parameter string s1p is: BBB
The parameter C-string cs1p is: CCC
The result of s1o.replace ( 1 , 3 , s1p )
 is the string: ABBBAAAA.
The result of s1o.replace ( 5 , 3 , cs1p )
 is the string: ABBBACCC.

The operand string s2o is: AAAAAAAA
The parameter string s1p is: BBB
The parameter C-string cs2p is: CCC
The result of s2o.replace (1, 3, s2p, 1, 2)
 is the string: ABBAAAA.
The result of s2o.replace (4 ,3 ,cs2p)
 is the string: ABBAC.

The operand string s3o is: AAAAAAAA
The parameter character c1p is: C
The result of s3o.replace(1, 3, 4, ch3p)
 is the string: ACCCCAAAA.

The operand string s4o is: AAAAAAAA
The parameter string s4p is: BBB
The parameter C-string cs4p is: CCC
The result of s1o.replace (IterF0, IterL0, s4p)
 is the string: BBBAAAAA.
The result of s4o.replace (IterF0, IterL0, cs4p)
 is the string: CCCAAAAA.

The operand string s5o is: AAAAAAAF
The parameter C-string cs5p is: CCCBB
```

```
The result of s5o.replace (IterF1, IterL1, cs4p ,4)
 is the string: CCCBAAAF.

The operand string s6o is: AAAAAAAG
The parameter character ch6p is: q
The result of s6o.replace (IterF1, IterL1, 4, ch6p)
 is the string: qqqqAAAAG.

The operand string s7o is: OOOOOOO
The parameter string s7p is: PPPP
The result of s7o.replace (IterF3 ,IterL3 ,IterF4 ,IterL4)
 is the string: OPPOOOO.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::reserve

Sets the capacity of the string to a number at least as great as a specified number.

```
void reserve(
   size_type _Count = 0
);
```

Parameters

_Count

The number of characters for which memory is being reserved.

Remarks

Having sufficient capacity is important because reallocations is a time-consuming process and invalidates all references, pointers, and iterators that refer to characters in a string.

The concept of capacity for objects of type strings is the same as for objects of type vector. Unlike vector, the member function **reserve** may be called to shrink the capacity of an object. The request is nonbinding and may or may not happen. As the default value for the parameter is zero, a call of **reserve** is a non-binding request to shrink the capacity of the string to fit the number of characters currently in the string. The capacity is never reduced below the current number of characters.

Calling **reserve** is the only possible way to shrink the capacity of a string. However, as noted above, this request is nonbinding and may not happen.

Example

```
// basic_string_reserve.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ("Hello world");
   cout << "The original string str1 is: " << str1 << endl;

   basic_string <char>::size_type sizeStr1, sizerStr1;
   sizeStr1 = str1.size ( );
   basic_string <char>::size_type capStr1, caprStr1;
   capStr1 = str1.capacity ( );

   // Compare size & capacity of the original string
   cout << "The current size of original string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The capacity of original string str1 is: "
        << capStr1 << "." << endl << endl;

   // Compare size & capacity of the string
   // with added capacity
   str1.reserve ( 40 );
   sizerStr1 = str1.size ( );
   caprStr1 = str1.capacity ( );

   cout << "The string str1with augmented capacity is: "
        << str1 << endl;
   cout << "The current size of string str1 is: "
        << sizerStr1 << "." << endl;
   cout << "The new capacity of string str1 is: "
        << caprStr1 << "." << endl << endl;

   // Compare size & capacity of the string
   // with downsized capacity
```

```
    str1.reserve ( );
    basic_string <char>::size_type sizedStr1;
    basic_string <char>::size_type capdStr1;
    sizedStr1 = str1.size ( );
    capdStr1 = str1.capacity ( );

    cout << "The string str1 with downsized capacity is: "
         << str1 << endl;
    cout << "The current size of string str1 is: "
         << sizedStr1 << "." << endl;
    cout << "The reduced capacity of string str1 is: "
         << capdStr1 << "." << endl << endl;
}
```

Output

```
The original string str1 is: Hello world
The current size of original string str1 is: 11.
The capacity of original string str1 is: 15.

The string str1with augmented capacity is: Hello world
The current size of string str1 is: 11.
The new capacity of string str1 is: 47.

The string str1 with downsized capacity is: Hello world
The current size of string str1 is: 11.
The reduced capacity of string str1 is: 47.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::resize

Specifies a new size for a string, appending or erasing elements as required.

```
void resize(
   size_type _Count,
   value_type _Ch = value_type( )
);
```

Parameters

_Count

The new size of the string.

_Ch

The value that appended characters are initialized with if additional elements are required.

Remarks

If the resulting size exceeds the maximum number of characters, the form throws **length_error**.

Example

```
// basic_string_resize.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string  str1 ( "Hello world" );
   cout << "The original string str1 is: " << str1 << endl;

   basic_string <char>::size_type sizeStr1;
   sizeStr1 = str1.size ( );
   basic_string <char>::size_type capStr1;
   capStr1 = str1.capacity ( );

   // Compare size & capacity of the original string
   cout << "The current size of original string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The capacity of original string str1 is: "
        << capStr1 << "." << endl << endl;

   // Use resize to increase size by 2 elements: exclamations
   str1.resize ( str1.size ( ) + 2 , '!' );
   cout << "The resized string str1 is: " << str1 << endl;

   sizeStr1 = str1.size ( );
   capStr1 = str1.capacity ( );

   // Compare size & capacity of a string after resizing
   cout << "The current size of resized string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The capacity of resized string str1 is: "
        << capStr1 << "." << endl << endl;

   // Use resize to increase size by 20 elements:
   str1.resize ( str1.size ( ) + 20 );
   cout << "The resized string str1 is: " << str1 << endl;

   sizeStr1 = str1.size ( );
   capStr1 = str1.capacity ( );
```

```
    // Compare size & capacity of a string after resizing
    // note capacity increases automatically as required
    cout << "The current size of modified string str1 is: "
         << sizeStr1 << "." << endl;
    cout << "The capacity of modified string str1 is: "
         << capStr1 << "." << endl << endl;

    // Use resize to downsize by 28 elements:
    str1.resize ( str1.size ( ) - 28 );
    cout << "The downsized string str1 is: " << str1 << endl;

    sizeStr1 = str1.size (  );
    capStr1 = str1.capacity (  );

    // Compare size & capacity of a string after downsizing
    cout << "The current size of downsized string str1 is: "
         << sizeStr1 << "." << endl;
    cout << "The capacity of downsized string str1 is: "
         << capStr1 << "." << endl;
}
```

Sample Output

```
The original string str1 is: Hello world
The current size of original string str1 is: 11.
The capacity of original string str1 is: 15.

The resized string str1 is: Hello world!!
The current size of resized string str1 is: 13.
The capacity of resized string str1 is: 15.

The resized string str1 is: Hello world!!
The current size of modified string str1 is: 33.
The capacity of modified string str1 is: 47.

The downsized string str1 is: Hello
The current size of downsized string str1 is: 5.
The capacity of downsized string str1 is: 47.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

basic_string::size and basic_string::resize

Other Resources

basic_string Members

basic_string::rfind

Searches a string in a backward direction for the first occurrence of a substring that matches a specified sequence of characters.

```
size_type rfind(
   value_type _Ch,
   size_type _Off = npos
) const;
size_type rfind(
   const value_type* _Ptr,
   size_type _Off = npos
) const;
size_type rfind(
   const value_type* _Ptr,
   size_type _Off = npos,
   size_type _Count
) const;
size_type rfind(
   const basic_string& _Str,
   size_type _Off = npos
) const;
```

Parameters

_Ch

The character value for which the member function is to search.

_Off

Index of the position at which the search is to begin.

_Ptr

The C-string for which the member function is to search.

_Count

The number of characters, counting forward from the first character, in the C-string for which the member function is to search.

_Str

The string for which the member function is to search.

Return Value

The index of the last occurrence, when searched backwards, of the first character of the substring when successful; otherwise **npos**.

Example

```
// basic_string_rfind.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function
   // searches for a single character in a string
   string str1 ( "Hello Everyone" );
   cout << "The original string str1 is: " << str1 << endl;
   basic_string <char>::size_type indexCh1a, indexCh1b;
   static const basic_string <char>::size_type npos = -1;
```

```
indexCh1a = str1.rfind ( "e" , 9 );
if ( indexCh1a != npos )
   cout << "The index of the 1st 'e' found before the 9th"
        << " position in str1 is: " << indexCh1a << endl;
else
   cout << "The character 'e' was not found in str1 ." << endl;

indexCh1b = str1.rfind ( "x" );
if ( indexCh1b != npos )
   cout << "The index of the 'x' found in str1 is: "
        << indexCh1b << endl << endl;
else
   cout << "The character 'x' was not found in str1."
        << endl << endl;

// The second member function searches a string
// for a substring as specified by a C-string
string str2 ( "Let me make this perfectly clear." );
cout << "The original string str2 is: " << str2 << endl;
basic_string <char>::size_type indexCh2a, indexCh2b;

const char *cstr2 = "perfect";
indexCh2a = str2.rfind ( cstr2 , 30 );
if ( indexCh2a != npos )
   cout << "The index of the 1st element of 'perfect' "
        << "before\n the 30th position in str2 is: "
        << indexCh2a << endl;
else
   cout << "The substring 'perfect' was not found in str2 ."
        << endl;

const char *cstr2b = "imperfectly";
indexCh2b = str2.rfind ( cstr2b , 30 );
if ( indexCh2b != npos )
   cout << "The index of the 1st element of 'imperfect' "
        << "before\n the 5th position in str3 is: "
        << indexCh2b << endl;
else
   cout << "The substring 'imperfect' was not found in str2 ."
        << endl << endl;

// The third member function searches a string
// for a substring as specified by a C-string
string str3 ( "It is a nice day. I am happy." );
cout << "The original string str3 is: " << str3 << endl;
basic_string <char>::size_type indexCh3a, indexCh3b;

const char *cstr3a = "nice";
indexCh3a = str3.rfind ( cstr3a );
if ( indexCh3a != npos )
   cout << "The index of the 1st element of 'nice' "
        << "in str3 is: " << indexCh3a << endl;
else
   cout << "The substring 'nice' was not found in str3 ."
        << endl;

const char *cstr3b = "am";
indexCh3b = str3.rfind ( cstr3b , indexCh3a + 25 , 2 );
if ( indexCh3b != npos )
   cout << "The index of the next occurrance of 'am' in "
        << "str3 begins at: " << indexCh3b << endl << endl;
else
   cout << "There is no next occurrence of 'am' in str3 ."
        << endl << endl;

// The fourth member function searches a string
// for a substring as specified by a string
```

```
    string str4 ( "This perfectly unclear." );
    cout << "The original string str4 is: " << str4 << endl;
    basic_string <char>::size_type indexCh4a, indexCh4b;

    string str4a ( "clear" );
    indexCh4a = str4.rfind ( str4a , 15 );
    if (indexCh4a != npos )
       cout << "The index of the 1st element of 'clear' "
            << "before\n the 15th position in str4 is: "
            << indexCh4a << endl;
    else
       cout << "The substring 'clear' was not found in str4 "
            << "before the 15th position." << endl;

    string str4b ( "clear" );
    indexCh4b = str4.rfind ( str4b );
    if ( indexCh4b != npos )
       cout << "The index of the 1st element of 'clear' "
            << "in str4 is: "
            << indexCh4b << endl;
    else
       cout << "The substring 'clear' was not found in str4 ."
            << endl << endl;
 }
```

Output

```
The original string str1 is: Hello Everyone
The index of the 1st 'e' found before the 9th position in str1 is: 8
The character 'x' was not found in str1.

The original string str2 is: Let me make this perfectly clear.
The index of the 1st element of 'perfect' before
 the 30th position in str2 is: 17
The substring 'imperfect' was not found in str2 .

The original string str3 is: It is a nice day. I am happy.
The index of the 1st element of 'nice' in str3 is: 8
The index of the next occurrance of 'am' in str3 begins at: 20

The original string str4 is: This perfectly unclear.
The substring 'clear' was not found in str4 before the 15th position.
The index of the 1st element of 'clear' in str4 is: 17
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::size

Returns the current number of elements in a string.

```
size_type size( ) const;
```

Return Value

The length of the string.

Example

```
// basic_string_size.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ("Hello world");
   cout << "The original string str1 is: " << str1 << endl;

   // The size and length member functions differ in name only
   basic_string <char>::size_type sizeStr1, lenStr1;
   sizeStr1 = str1.size (  );
   lenStr1 = str1.length (  );

   basic_string <char>::size_type capStr1, max_sizeStr1;
   capStr1 = str1.capacity (  );
   max_sizeStr1 = str1.max_size (  );

   // Compare size, length, capacity & max_size of a string
   cout << "The current size of original string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The current length of original string str1 is: "
        << lenStr1 << "." << endl;
   cout << "The capacity of original string str1 is: "
        << capStr1 << "." << endl;
   cout << "The max_size of original string str1 is: "
        << max_sizeStr1 << "." << endl << endl;

   str1.erase ( 6, 5 );
   cout << "The modified string str1 is: " << str1 << endl;

   sizeStr1 = str1.size ( );
   lenStr1 = str1.length ( );
   capStr1 = str1.capacity ( );
   max_sizeStr1 = str1.max_size ( );

   // Compare size, length, capacity & max_size of a string
   // after erasing part of the original string
   cout << "The current size of modified string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The current length of modified string str1 is: "
        << lenStr1 << "." << endl;
   cout << "The capacity of modified string str1 is: "
        << capStr1 << "." << endl;
   cout << "The max_size of modified string str1 is: "
        << max_sizeStr1 << "." << endl;
}
```

Sample Output

The following output is for x86.

```
The original string str1 is: Hello world
The current size of original string str1 is: 11.
The current length of original string str1 is: 11.
The capacity of original string str1 is: 15.
The max_size of original string str1 is: 4294967294.

The modified string str1 is: Hello
The current size of modified string str1 is: 6.
The current length of modified string str1 is: 6.
The capacity of modified string str1 is: 15.
The max_size of modified string str1 is: 4294967294.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

basic_string::size and basic_string::resize

Other Resources

basic_string Members

basic_string::substr

Copies a substring of at most some number of characters from a string beginning from a specified position.

```
basic_string substr(
   size_type _Off = 0,
   size_type _Count = npos
) const;
```

Parameters

_Off

An index locating the element at the position from which the copy of the string is made, with a default value of 0.

_Count

The number of characters that are to be copied if they are present.

Return Value

A substring object that is a copy of elements of the string operand beginning at the position specified by the first argument.

Example

```
// basic_string_substr.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   string  str1 ("Heterological paradoxes are persistent.");
   cout << "The original string str1 is: \n " << str1
        << endl << endl;

   basic_string <char> str2 = str1.substr ( 6 , 7 );
   cout << "The substring str1 copied is: " << str2
        << endl << endl;

   basic_string <char> str3 = str1.substr (  );
   cout << "The default substring str3 is: \n " << str3
        <<  "\n which is the entire original string." << endl;
}
```

Output

```
The original string str1 is:
 Heterological paradoxes are persistent.

The substring str1 copied is: logical

The default substring str3 is:
 Heterological paradoxes are persistent.
 which is the entire original string.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::swap

Exchange the contents of two strings.

```
void swap(
   basic_string& _Str
);
```

Parameters

_Str

The source string whose elements are to be exchanged with those in the destination string.

Remarks

If the strings being swapped have the same allocator object, the **swap** member function:

- Occurs in constant time.
- Throws no exceptions.
- Invalidates no references, pointers, or iterators that designate elements in the two strings.

Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

Example

```
// basic_string_swap.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // Declaring an objects of type basic_string<char>
   string s1 ( "Tweedledee" );
   string s2 ( "Tweedledum" );
   cout << "Before swapping string s1 and s2:" << endl;
   cout << " The basic_string s1 = " << s1 << "." << endl;
   cout << " The basic_string s2 = " << s2 << "." << endl;

   s1.swap ( s2 );
   cout << "After swapping string s1 and s2:" << endl;
   cout << " The basic_string s1 = " << s1 << "." << endl;
   cout << " The basic_string s2 = " << s2 << "." << endl;
}
```

Output

```
Before swapping string s1 and s2:
 The basic_string s1 = Tweedledee.
 The basic_string s2 = Tweedledum.
After swapping string s1 and s2:
 The basic_string s1 = Tweedledum.
 The basic_string s2 = Tweedledee.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string Operators

For more information about the operators in the basic_string class, see basic_string Members.

basic_string::operator+=

Appends characters to a string.

```
basic_string& operator+=(
   value_type _Ch
);
basic_string& operator+=(
   const value_type* _Ptr
);
basic_string& operator+=(
   const basic_string& _Right
);
```

Parameters

_Ch

The character to be appended.

_Ptr

The characters of the C-string to be appended.

_Right

The characters of the string to be appended.

Return Value

A reference to the string object that is being appended with the characters passed by the member function.

Remarks

Characters may be appended to a string using the **operator+=** or the member functions append or push_back. The **operator+=** appends single-argument values while the multiple argument append member function allows a specific part of a string to be specified for adding.

Example

```
// basic_string_op_app.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function
   // appending a single character to a string
   string str1a ( "Hello" );
   cout << "The original string str1 is: " << str1a << endl;
   str1a +=  '!' ;
   cout << "The string str1 appended with an exclamation is: "
        << str1a << endl << endl;

   // The second member function
   // appending a C-string to a string
   string  str1b ( "Hello " );
   const char *cstr1b = "Out There";
   cout << "The C-string cstr1b is: " << cstr1b << endl;
   str1b +=  cstr1b;
   cout << "Appending the C-string cstr1b to string str1 gives: "
        << str1b << "." << endl << endl;

   // The third member function
```

```
    // appending one string to another in two ways,
    // comparing append and operator [ ]
    string str1d ( "Hello " ), str2d ( "Wide " ), str3d ( "World" );
    cout << "The string str2d is: " << str2d << endl;
    str1d.append ( str2d );
    cout << "The appended string str1d is: "
         << str1d << "." << endl;
    str1d += str3d;
    cout << "The doubly appended strig str1 is: "
         << str1d << "." << endl << endl;
}
```

Output

```
The original string str1 is: Hello
The string str1 appended with an exclamation is: Hello!

The C-string cstr1b is: Out There
Appending the C-string cstr1b to string str1 gives: Hello Out There.

The string str2d is: Wide
The appended string str1d is: Hello Wide .
The doubly appended strig str1 is: Hello Wide World.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::operator=

Assigns new character values to the contents of a string.

```
basic_string& operator=(
   value_type _Ch
);
basic_string& operator=(
   const value_type* _Ptr
);
basic_string& operator=(
   const basic_string& _Right
);
```

Parameters

_Ch

The character value to be assigned.

_Ptr

A pointer to the characters of the C-string to be assigned to the target string.

_Right

The source string whose characters are to be assigned to the target string.

Return Value

A reference to the string object that is being assigned new characters by the member function.

Remarks

The strings may be assigned new character values. The new value may be either a string and C-string or a single character. The **operator=** may be used if the new value can be described by a single parameter, otherwise the member function assign, which has multiple parameters, may be used to specify which part of the string is to be assigned to a target string.

Example

```
// basic_string_op_assign.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function assigning a
   // character of a certain value to a string
   string str1a ( "Hello " );
   str1a = '0';
   cout << "The string str1 assigned with the zero character is: "
        << str1a << endl << endl;

   // The second member function assigning the
   // characters of a C-string to a string
   string  str1b;
   const char *cstr1b = "Out There";
   cout << "The C-string cstr1b is: " << cstr1b <<  "." << endl;
   str1b = cstr1b;
   cout << "Assigning the C-string cstr1a to string str1 gives: "
        << str1b << "." << endl << endl;

   // The third member function assigning the characters
   // from one string to another string in two equivalent
```

```
    // ways, comparing the assign and operator =
    string str1c ( "Hello" ), str2c ( "Wide" ), str3c ( "World" );
    cout << "The original string str1 is: " << str1c << "." << endl;
    cout << "The string str2c is: " << str2c << "." << endl;
    str1c.assign ( str2c );
    cout << "The string str1 newly assigned with string str2c is: "
         << str1c << "." << endl;
    cout << "The string str3c is: " << str3c << "." << endl;
    str1c = str3c;
    cout << "The string str1 reassigned with string str3c is: "
         << str1c << "." << endl << endl;
}
```

Output

```
The string str1 assigned with the zero character is: 0

The C-string cstr1b is: Out There.
Assigning the C-string cstr1a to string str1 gives: Out There.

The original string str1 is: Hello.
The string str2c is: Wide.
The string str1 newly assigned with string str2c is: Wide.
The string str3c is: World.
The string str1 reassigned with string str3c is: World.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

basic_string::operator[]

Provides a reference to the character with a specified index in a string.

```
const_reference operator[](
   size_type _Off
) const;
reference operator[](
   size_type _Off
);
```

Parameters

_Off

The index of the position of the element to be referenced.

Return Value

A reference to the character of the string at the position specified by the parameter index.

Remarks

The first element of the string has an index of zero, and the following elements are indexed consecutively by the positive integers, so that a string of length *n* has an *n*th element indexed by the number *n* - 1.

operator[] is faster than the member function at for providing read and write access to the elements of a string.

operator[] does not check whether the index passed as a parameter is valid, but the member function **at** does and so should be used in the validity is not certain. An invalid index (an index less that zero or greater than or equal to the size of the string) passed to the member function **at** throws an out_of_range Class exception. An invalid index passed to **operator[]** results in undefined behavior, but the index equal to the length of the string is a valid index for const strings and the operator returns the null character when passed this index.

The reference returned may be invalidated by string reallocations or modifications for the non-**const** strings.

When compiling with _SECURE_SCL 1, a runtime error will occur if you attempt to access an element outside the bounds of the string. See Checked Iterators for more information.

Example

```
// basic_string_op_ref.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ( "Hello world" ), str2 ( "Goodbye world" );
   const string cstr1 ( "Hello there" ), cstr2 ( "Goodbye now" );
   cout << "The original string str1 is: " << str1 << endl;
   cout << "The original string str2 is: " << str2 << endl;

   // Element access to the non-const strings
   basic_string <char>::reference refStr1 = str1 [6];
   basic_string <char>::reference refStr2 = str2.at ( 3 );

   cout << "The character with an index of 6 in string str1 is: "
        << refStr1 << "." << endl;
   cout << "The character with an index of 3 in string str2 is: "
        << refStr2 << "." << endl;

   // Element access to the const strings
   basic_string <char>::const_reference crefStr1 = cstr1 [ cstr1.length ( ) ];
   basic_string <char>::const_reference crefStr2 = cstr2.at ( 8 );
```

```
    if ( crefStr1 == '\0' )
       cout << "The null character is returned as a valid reference."
            << endl;
    else
       cout << "The null character is not returned." << endl;
    cout << "The character with index of 8 in the const string cstr2 is: "
         << crefStr2 << "." << endl;
 }
```

Output

```
The original string str1 is: Hello world
The original string str2 is: Goodbye world
The character with an index of 6 in string str1 is: w.
The character with an index of 3 in string str2 is: d.
The null character is returned as a valid reference.
The character with index of 8 in the const string cstr2 is: n.
```

Requirements

Header: <string>

See Also

Reference

basic_string Class

Other Resources

basic_string Members

char_traits Class

The char_traits class describes attributes associated with a character.

For a list of all members of this type, see char_traits Members.

```
template <
   class CharType
> struct char_traits;
```

Parameters

CharType

The element data type.

Remarks

The template class describes various character traits for type **CharType**. The template class basic_string as well as several iostream template classes, including basic_ios, use this information to manipulate elements of type **CharType**. Such an element type must not require explicit construction or destruction. It must supply a default constructor, a copy constructor, and an assignment operator, with the expected semantics. A bitwise copy must have the same effect as an assignment. None of the member functions of class char_traits can throw exceptions.

Requirements

Header: <string>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

char_traits Members

<string> Members

char_traits Members

Reference

Typedefs

char_type	A type of character.
int_type	An integer type that can represent a character of type **char_type** or an end-of-file (EOF) character.
off_type	An integer type that can represent offsets between positions in a stream.
pos_type	An integer type that can represent positions in a stream.
state_type	A type that represents the conversion state in for multibyte characters in a stream.

Member Functions

assign	Assigns one character value to another.
compare	Compares up to a specified number of characters in two strings.
copy	Copies a specified number of characters from one string to another. Deprecated. Use char_traits::_Copy_s instead.
_Copy_s	Copies a specified number of characters from one string to another.
eof	Returns the end-of-file (EOF) character.
eq	Tests whether two **char_type** characters are equal.
eq_int_type	Tests whether two characters represented as **int_type**s are equal.
find	Searches for the first occurrence of a specified character in a range of characters.
length	Returns the length of a string.
lt	Tests whether one character is less than another.
move	Copies a specified number of characters in a sequence to another, possible overlapping, sequence. Deprecated. Use char_traits::_Move_s instead.
_Move_s	Copies a specified number of characters in a sequence to another, possible overlapping, sequence.
not_eof	Tests whether a character is the end-of-file (EOF) character.
to_char_type	Converts an **int_type** character to the corresponding **char_type** character and returns the result.
to_int_type	Converts a **char_type** character to the corresponding **int_type** character and returns the result.

See Also

Reference

char_traits Class

Thread Safety in the Standard C++ Library

char_traits Typedefs

For information about the typedefs in the char_traits class, see char_traits Members.

char_traits::char_type

A type of character.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

Example

See the example for copy for an example of how to declare and use **char_type**.

Requirements

Header: <string>

See Also

Reference

char_traits Class

Other Resources

char_traits Members

char_traits::int_type

An integer type that can represent a character of type **char_type** or an end-of-file (EOF) character.

```
typedef implementation-defined int_type;
```

Remarks

It must be possible to type cast a value of type **CharType** to **int_type** then back to **CharType** without altering the original value.

Example

See the example for eq_int_type for an example of how to declare and use **int_type**.

Requirements

Header: <string>

See Also

Reference

char_traits Class

Other Resources

char_traits Members

char_traits::off_type

An integer type that can represent offsets between positions in a stream.

```
typedef implementation-defined off_type;
```

Remarks

The type is a signed integer that describes an object that can store a byte offset involved in various stream positioning operations. It is typically a synonym for streamoff, but it has essentially the same properties as that type.

Requirements

Header: <string>

See Also

Reference

char_traits Class

Other Resources

char_traits Members

char_traits::pos_type

An integer type that can represent positions in a stream.

```
typedef implementation-defined pos_type;
```

Remarks

The type describes an object that can store all the information needed to restore an arbitrary file-position indicator within a stream. It is typically a synonym for streampos, but in any case it has essentially the same properties as that type.

Requirements

Header: <string>

See Also

Reference

char_traits Class

Other Resources

char_traits Members

char_traits::state_type

A type that represents the conversion state for multibyte characters in a stream.

```
typedef implementation-defined state_type;
```

Remarks

The type describes an object that can represent a conversion state. It is typically a synonym for **mbstate_t**, but in any case it has essentially the same properties as that type.

Requirements

Header: <string>

See Also

Reference

char_traits Class

Other Resources

char_traits Members

char_traits Member Functions

For information about the functions in the char_traits class, see char_traits Members.

char_traits::assign

Assigns one character value to another or to a range of elements in a string.

```
static void assign(
   char_type& _CharTo,
   const char_type& _CharFrom
);
static char_type *assign(
   char_type* _StrTo,
   size_t _Num,
   char_type _CharFrom
);
```

Parameters

_ CharFrom

The character whose value is to be assigned.

_CharTo

The element that is to be assigned the character value.

_StrTo

The string or character array whose initial elements are to be assigned character values.

_Num

The number of elements that are going to be assigned values.

Return Value

The second member function returns a pointer to the string whose first *_Num* elements have been assigned values of *_CharFrom*.

Example

```
// char_traits_assign.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function assigning
   // one character value to another character
   char ChTo = 't';
   const char ChFrom = 'f';
   cout << "The initial characters ( ChTo , ChFrom ) are: ( "
        << ChTo << " , " << ChFrom << " )." << endl;
   char_traits<char>::assign ( ChTo , ChFrom );
   cout << "After assigning, the characters ( ChTo , ChFrom ) are: ( "
        << ChTo << " , " << ChFrom << " )." << endl << endl;

   // The second member function assigning
   // character values to initial part of a string
   char_traits<char>::char_type s1[] = "abcd-1234-abcd";
   char_traits<char>::char_type* result1;
   cout << "The target string s1 is: " << s1 << endl;
   result1 = char_traits<char>::assign ( s1 , 4 , 'f' );
   cout << "The result1 = assign ( s1 , 4 , 'f' ) is: "
        << result1 << endl;
}
```

Output

```
The initial characters ( ChTo , ChFrom ) are: ( t , f ).
After assigning, the characters ( ChTo , ChFrom ) are: ( f , f ).

The target string s1 is: abcd-1234-abcd
The result1 = assign ( s1 , 4 , 'f' ) is: ffff-1234-abcd
```

Requirements

Header: <string>

See Also

Reference

char_traits Class

Other Resources

char_traits Members

char_traits::compare

Compares up to a specified number of characters in two strings.

```
static int compare(
   const char_type* _Str1,
   const char_type* _Str2,
   size_t _Num
);
```

Parameters

_Str1

The first of two strings to be compared to each other.

_Str2

The second of two strings to be compared to each other.

_Num

The number of elements in the strings to be compared.

Return Value

A negative value if the first string is less than the second string, 0 if the two strings are equal, or a positive value if the first string is greater than the second string.

Remarks

The comparison between the strings is made element by element, first testing for equality and then, if a pair of elements in the sequence tests not equal, they are tested for less than.

If two strings compare equal over a range but one is longer than the other, then the shorter of the two is less than the longer one.

Example

```
// char_traits_compare.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main() {
   using namespace std;

   char_traits<char>::char_type* s1 = "CAB";
   char_traits<char>::char_type* s2 = "ABC";
   char_traits<char>::char_type* s3 = "ABC";
   char_traits<char>::char_type* s4 = "ABCD";

   cout << "The string s1 is: " << s1 << endl;
   cout << "The string s2 is: " << s2 << endl;
   cout << "The string s3 is: " << s3 << endl;
   cout << "The string s4 is: " << s4 << endl;

   int comp1, comp2, comp3, comp4;
   comp1 = char_traits<char>::compare ( s1 , s2 , 2 );
   comp2 = char_traits<char>::compare ( s2 , s3 , 3 );
   comp3 = char_traits<char>::compare ( s3 , s4 , 4 );
   comp4 = char_traits<char>::compare ( s4 , s3 , 4 );
   cout << "compare ( s1 , s2 , 2 ) = " << comp1 << endl;
   cout << "compare ( s2 , s3 , 3 ) = " << comp2 << endl;
   cout << "compare ( s3 , s4 , 4 ) = " << comp3 << endl;
   cout << "compare ( s4 , s3 , 4 ) = " << comp4 << endl;
}
```

Sample Output

```
The string s1 is: CAB
The string s2 is: ABC
The string s3 is: ABC
The string s4 is: ABCD
compare ( s1 , s2 , 2 ) = 1
compare ( s2 , s3 , 3 ) = 0
compare ( s3 , s4 , 4 ) = -1
compare ( s4 , s3 , 4 ) = 1
```

Requirements

Header: <string>

See Also

Reference

char_traits Class

Other Resources

char_traits Members

char_traits::copy

Copies a specified number of characters from one string to another.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct. Consider using char_traits::_Copy_s instead.

```
static char_type *copy(
   char_type* _To,
   const char_type* _From,
   size_t _Num
);
```

Parameters

_To

The element at the beginning of the string or character array targeted to receive the copied sequence of characters.

_From

The element at the beginning of the source string or character array to be copied.

_Num

The number of elements to be copied.

Return Value

The first element copied into the string or character array targeted to receive the copied sequence of characters.

Remarks

The source and destination character sequences must not overlap.

Example

```
// char_traits_copy.cpp
// compile with: /EHsc /W3
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   char_traits<char>::char_type s1[] = "abcd-1234-abcd";
   char_traits<char>::char_type s2[] = "ABCD-1234";
   char_traits<char>::char_type* result1;
   cout << "The source string is: " << s1 << endl;
   cout << "The destination string is: " << s2 << endl;
   // Note: char_traits::copy is potentially unsafe, consider
   // using char_traits::_Copy_s instead.
   result1 = char_traits<char>::copy ( s1 , s2 , 4 );  // C4996
   cout << "The result1 = copy ( s1 , s2 , 4 ) is: "
        << result1 << endl;
}
```

Output

```
The source string is: abcd-1234-abcd
The destination string is: ABCD-1234
The result1 = copy ( s1 , s2 , 4 ) is: ABCD-1234-abcd
```

Requirements

Header: <string>

See Also

Reference

char_traits Class

Other Resources

char_traits Members

char_traits::_Copy_s

Copies a specified number of characters from one string to another.

```
static char_type *_Copy_s(
    char_type *_Dest,
    size_t _Dest_size,
    const char_type *_From,
    size_t _Count
);
```

Parameters

_Dest

The element at the beginning of the string or character array targeted to receive the copied sequence of characters.

_Dest_size

The size of *_Dest*. If **char_type** is **char**, then this size is in bytes. If **char_type** is **wchar_t**, then this size is in words.

_From

The element at the beginning of the source string or character array to be copied.

_Count

The number of elements to be copied.

Return Value

The first element copied into the string or character array targeted to receive the copied sequence of characters.

Remarks

The source and destination character sequences must not overlap.

Example

```
// char_traits__Copy_s.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    char_traits<char>::char_type s1[] = "abcd-1234-abcd";
    char_traits<char>::char_type s2[] = "ABCD-1234";
    char_traits<char>::char_type* result1;
    cout << "The source string is: " << s1 << endl;
    cout << "The destination string is: " << s2 << endl;
    result1 = char_traits<char>::_Copy_s(s1,
        char_traits<char>::length(s1), s2, 4);
    cout << "The result1 = _Copy_s(s1, "
         << "char_traits<char>::length(s1), s2, 4) is: "
         << result1 << endl;
}
```

Output

```
The source string is: abcd-1234-abcd
The destination string is: ABCD-1234
The result1 = _Copy_s(s1, char_traits<char>::length(s1), s2, 4) is: ABCD-1234-abcd
```

Requirements

Header: <string>

See Also

Reference

char_traits Class

Safe Libraries: Standard C++ Library

Other Resources

char_traits Members

char_traits::eof

Returns the end-of-file (EOF) character.

```
static int_type eof();
```

Return Value

The EOF character.

Remarks

A value that represents end of file (such as **EOF** or **WEOF**).

The C++ standard states that this value must not correspond to a valid **char_type** value. The Visual C++ compiler enforces this constraint for type **char**, but not for type **wchar_t**. The example below demonstrates this.

Example

```
// char_traits_eof.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main()
{
    using namespace std;

    char_traits<char>::char_type ch1 = 'x';
    char_traits<char>::int_type int1;
    int1 = char_traits<char>::to_int_type(ch1);
    cout << "char_type ch1 is '" << ch1 << "' and corresponds to int_type "
         << int1 << "." << endl << endl;

    char_traits<char>::int_type int2 = char_traits<char>::eof();
    cout << "The eof marker for char_traits<char> is: " << int2 << endl;

    char_traits<wchar_t>::int_type int3 = char_traits<wchar_t>::eof();
    cout << "The eof marker for char_traits<wchar_t> is: " << int3 << endl;
}
```

Output

```
char_type ch1 is 'x' and corresponds to int_type 120.

The eof marker for char_traits<char> is: -1
The eof marker for char_traits<wchar_t> is: 65535
```

Requirements

Header: <string>

See Also

Reference

char_traits Class

Other Resources

char_traits Members

char_traits::eq

Tests whether two **char_type** characters are equal.

```
static bool eq(
   const char_type& _Ch1,
   const char_type& _Ch2
);
```

Parameters

_Ch1

The first of two characters to be tested for equality.

_Ch2

The second of two characters to be tested for equality.

Return Value

true if the first character is equal to the second character; otherwise **false**.

Example

```
// char_traits_eq.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   char_traits<char>::char_type ch1 =  'x';
   char_traits<char>::char_type ch2 =  'y';
   char_traits<char>::char_type ch3 =  'x';

   // Testing for equality
   bool b1 = char_traits<char>::eq ( ch1 , ch2 );
   if ( b1 )
      cout << "The character ch1 is equal "
           << "to the character ch2." << endl;
   else
      cout << "The character ch1 is not equal "
           << "to the character ch2." << endl;

   // An equivalent and alternatively test procedure
   if ( ch1 == ch3 )
      cout << "The character ch1 is equal "
           << "to the character ch3." << endl;
   else
      cout << "The character ch1 is not equal "
           << "to the character ch3." << endl;
}
```

Output

```
The character ch1 is not equal to the character ch2.
The character ch1 is equal to the character ch3.
```

Requirements

Header: <string>

See Also

Reference

char_traits Class

Other Resources

char_traits Members

char_traits::eq_int_type

Tests whether two characters represented as **int_type**s are equal or not.

```
static bool eq_int_type(
   const int_type& _Ch1,
   const int_type& _Ch2
);
```

Parameters

_Ch1

The first of the two characters to be tested for equality as **int_type**s.

_Ch2

The second of the two characters to be tested for equality as **int_type**s.

Return Value

true if the first character is equal to the second character; otherwise **false**.

Example

```
// char_traits_eq_int_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   char_traits<char>::char_type ch1 =  'x';
   char_traits<char>::char_type ch2 =  'y';
   char_traits<char>::char_type ch3 =  'x';

   // Converting from char_type to int_type
   char_traits<char>::int_type int1, int2 , int3;
   int1 =char_traits<char>:: to_int_type ( ch1 );
   int2 =char_traits<char>:: to_int_type ( ch2 );
   int3 =char_traits<char>:: to_int_type ( ch3 );

   cout << "The char_types and corresponding int_types are:"
        << "\n    ch1 = " << ch1 << " corresponding to int1 = "
        << int1 << "."
        << "\n    ch2 = " << ch2 << " corresponding to int1 = "
        << int2 << "."
        << "\n    ch3 = " << ch3 << " corresponding to int1 = "
        << int3 << "." << endl << endl;

   // Testing for equality of int_type representations
   bool b1 = char_traits<char>::eq_int_type ( int1 , int2 );
   if ( b1 )
      cout << "The int_type representation of character ch1\n "
           << "is equal to the int_type representation of ch2."
           << endl;
   else
      cout << "The int_type representation of character ch1\n is "
           << "not equal to the int_type representation of ch2."
           << endl;

   // An equivalent and alternatively test procedure
   if ( int1 == int3 )
      cout << "The int_type representation of character ch1\n "
```

```
            << "is equal to the int_type representation of ch3."
            << endl;
    else
       cout << "The int_type representation of character ch1\n is "
            << "not equal to the int_type representation of ch3."
            << endl;
  }
```

Output

```
The char_types and corresponding int_types are:
    ch1 = x corresponding to int1 = 120.
    ch2 = y corresponding to int1 = 121.
    ch3 = x corresponding to int1 = 120.

The int_type representation of character ch1
 is not equal to the int_type representation of ch2.
The int_type representation of character ch1
 is equal to the int_type representation of ch3.
```

Requirements

Header: <string>

See Also

Reference

char_traits Class

Other Resources

char_traits Members

char_traits::find

Searches for the first occurrence of a specified character in a range of characters.

```
static const char_type* find(
   const char_type* _Str,
   size_t _Num,
   const char_type& _Ch
);
```

Parameters

_Str

The first character in the string to be searched.

_Num

The number of positions, counting from the first, in the range to be searched.

_Ch

The character to be searched for in the range.

Return Value

A pointer to the first occurrence of the specified character in the range if a match is found; otherwise, a null pointer.

Example

```
// char_traits_find.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   const char* s1 = "f2d-1234-abcd";
   const char* result1;
   cout << "The string to be searched is: " << s1 << endl;

   // Searching for a 'd' in the first 6 positions of string s1
   result1 = char_traits<char>::find ( s1 , 6 , 'd');
   cout << "The character searched for in s1 is: "
        << *result1 << endl;
   cout << "The string beginning with the first occurrence\n "
        << "of the character 'd' is: " << result1 << endl;

   // When no match is found the NULL value is returned
   const char* result2;
   result2 = char_traits<char>::find ( s1 , 3 , 'a');
   if ( result2 == NULL )
      cout << "The result2 of the search is NULL." << endl;
   else
      cout << "The result2 of the search  is: " << result1
           << endl;
}
```

Output

```
The string to be searched is: f2d-1234-abcd
The character searched for in s1 is: d
The string beginning with the first occurrence
 of the character 'd' is: d-1234-abcd
The result2 of the search is NULL.
```

Requirements

Header: <string>

See Also

Reference

char_traits Class

Other Resources

char_traits Members

char_traits::length

Returns the length of a string.

```
static size_t length(
   const char_type* _Str
);
```

Parameters

_Str

The C-string whose length is to be measured.

Return Value

The number of elements in the sequence being measured, not including the null terminator.

Example

```
// char_traits_length.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   const char* str1= "Hello";
   cout << "The C-string str1 is: " << str1 << endl;

   size_t lenStr1;
   lenStr1 = char_traits<char>::length ( str1 );
   cout << "The length of C-string str1 is: "
        << lenStr1 << "." << endl;
}
```

Output

```
The C-string str1 is: Hello
The length of C-string str1 is: 5.
```

Requirements

Header: <string>

See Also

Reference

char_traits Class

Other Resources

char_traits Members

char_traits::lt

Tests whether one character is less than another.

```
static bool lt(
   const char_type& _Ch1,
   const char_type& _Ch2
);
```

Parameters

_Ch1

The first of two characters to be tested for less than.

_Ch2

The second of two characters to be tested for less than.

Return Value

true if the first character is less than the second character; otherwise **false**.

Example

```
// char_traits_lt.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   char_traits<char>::char_type ch1 =  'x';
   char_traits<char>::char_type ch2 =  'y';
   char_traits<char>::char_type ch3 =  'z';

   // Testing for less than
   bool b1 = char_traits<char>::lt ( ch1 , ch2 );
   if ( b1 )
      cout << "The character ch1 is less than "
           << "the character ch2." << endl;
   else
      cout << "The character ch1 is not less "
           << "than the character ch2." << endl;

   // An equivalent and alternatively test procedure
   if ( ch3 <  ch2 )
      cout << "The character ch3 is less than "
           << "the character ch2." << endl;
   else
      cout << "The character ch3 is not less "
           << "than the character ch2." << endl;
}
```

Output

```
The character ch1 is less than the character ch2.
The character ch3 is not less than the character ch2.
```

Requirements

Header: <string>

See Also

Reference

char_traits Class

Other Resources

char_traits Members

char_traits::move

Copies a specified number of characters in a sequence to another, possibly overlapping sequence.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct. Consider using char_traits::_Move_s instead.

```
static char_type *move(
   char_type* _To,
   const char_type* _From,
   size_t _Num
);
```

Parameters

_To

The element at the beginning of the string or character array targeted to receive the copied sequence of characters.

_From

The element at the beginning of the source string or character array to be copied.

_Num

The number of elements to be copied from the source string.

Return Value

The first element *_To* copied into the string or character array targeted to receive the copied sequence of characters.

Remarks

The source and destination may overlap.

Example

```
// char_traits_move.cpp
// compile with: /EHsc /W3
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   char_traits<char>::char_type sFrom1[] =  "abcd-1234-abcd";
   char_traits<char>::char_type sTo1[] =  "ABCD-1234";
   char_traits<char>::char_type* result1;
   cout << "The source string sFrom1 is: " << sFrom1 << endl;
   cout << "The destination stringsTo1 is: " << sTo1 << endl;
   // Note: char_traits::move is potentially unsafe, consider
   // using char_traits::_Move_s instead.
   result1 = char_traits<char>::move ( sTo1 ,  sFrom1 , 4 );  // C4996
   cout << "The result1 = move ( sTo1 , sFrom1 , 4 ) is: "
        << result1 << endl << endl;

   // When source and destination overlap
   char_traits<char>::char_type sToFrom2[] = "abcd-1234-ABCD";
   char_traits<char>::char_type* result2;
   cout << "The source/destination string sToFrom2 is: "
        << sToFrom2 << endl;
   const char* findc = char_traits<char>::find ( sToFrom2 , 4 , 'c' );
   // Note: char_traits::move is potentially unsafe, consider
   // using char_traits::_Move_s instead.
   result2 = char_traits<char>::move ( sToFrom2 , findc , 8 );  // C4996
   cout << "The result2 = move ( sToFrom2 , findc , 8 ) is: "
        << result2 << endl;
```

```
}
```

Output

```
The source string sFrom1 is: abcd-1234-abcd
The destination stringsTo1 is: ABCD-1234
The result1 = move ( sTo1 , sFrom1 , 4 ) is: abcd-1234

The source/destination string sToFrom2 is: abcd-1234-ABCD
The result2 = move ( sToFrom2 , findc , 8 ) is: cd-1234-4-ABCD
```

Requirements

Header: <string>

See Also

Reference

char_traits Class

Other Resources

char_traits Members

char_traits::_Move_s

Copies a specified number of characters in a sequence to another, possibly overlapping sequence.

```
static char_type *_Move_s(
    char_type *_Dest,
    size_t _Dest_size,
    const char_type *_From,
    size_t _Count
);
```

Parameters

_Dest

The element at the beginning of the string or character array targeted to receive the copied sequence of characters.

_Dest_size

The size of *_Dest*. If **char_type** is **char**, then this is in bytes. If **char_type** is **wchar_t**, then this is in words.

_From

The element at the beginning of the source string or character array to be copied.

_Count

The number of elements to be copied from the source string.

Return Value

The first element *_Dest* copied into the string or character array targeted to receive the copied sequence of characters.

Remarks

The source and destination may overlap.

Example

```
// char_traits__Move_s.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    char_traits<char>::char_type sFrom1[] =  "abcd-1234-abcd";
    char_traits<char>::char_type sTo1[] =  "ABCD-1234";
    char_traits<char>::char_type* result1;
    cout << "The source string sFrom1 is: " << sFrom1 << endl;
    cout << "The destination stringsTo1 is: " << sTo1 << endl;
    result1 = char_traits<char>::_Move_s(sTo1,
        char_traits<char>::length(sTo1), sFrom1, 4);
    cout << "The result1 = _Move_s(sTo1, "
         << "char_traits<char>::length(sTo1), sFrom1, 4) is: "
         << result1 << endl << endl;

    // When source and destination overlap
    char_traits<char>::char_type sToFrom2[] = "abcd-1234-ABCD";
    char_traits<char>::char_type* result2;
    cout << "The source/destination string sToFrom2 is: "
         << sToFrom2 << endl;
    const char* findc = char_traits<char>::find(sToFrom2, 4, 'c');
    result2 = char_traits<char>::_Move_s(sToFrom2,
        char_traits<char>::length(sToFrom2), findc, 8);
    cout << "The result2 = _Move_s(sToFrom2, "
```

```
        << "char_traits<char>::length(sToFrom2), findc, 8) is: "
         << result2 << endl;
}
```

Output

```
The source string sFrom1 is: abcd-1234-abcd
The destination stringsTo1 is: ABCD-1234
The result1 = _Move_s(sTo1, char_traits<char>::length(sTo1), sFrom1, 4) is: abcd-1234

The source/destination string sToFrom2 is: abcd-1234-ABCD
The result2 = _Move_s(sToFrom2, char_traits<char>::length(sToFrom2), findc, 8) is: cd-1
234-4-ABCD
```

Requirements

Header: <string>

See Also

Reference

char_traits Class

Safe Libraries: Standard C++ Library

Other Resources

char_traits Members

char_traits::not_eof

Tests whether a character is not the end-of-file (EOF) character or is the EOF.

```
static int_type not_eof(
   const int_type& _Ch
);
```

Parameters

_Ch

The character represented as an **int_type** to be tested for whether it is the EOF character or not.

Return Value

The **int_type** representation of the character tested, if the **int_type** of the character is not equal to that of the EOF character.

If the character **int_type** value is equal to the EOF **int_type** value, then **false**.

Example

```
// char_traits_not_eof.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( ) {
   using namespace std;

   char_traits<char>::char_type ch1 =  'x';
   char_traits<char>::int_type int1;
   int1 = char_traits<char>:: to_int_type ( ch1 );
   cout << "The char_type ch1 is " << ch1
        << " corresponding to int_type: "
        << int1 << "." << endl;

   // EOF member function
   char_traits <char>::int_type int2 = char_traits<char>::eof ( );
   cout << "The eofReturn is: " << int2 << endl;

   // Testing for EOF or another character
   char_traits <char>::int_type eofTest1, eofTest2;
   eofTest1 = char_traits<char>::not_eof ( int1 );
   if ( !eofTest1 )
      cout << "The eofTest1 indicates ch1 is an EOF character."
               << endl;
   else
      cout << "The eofTest1 returns: " << eofTest1
           << ", which is the character: "
           <<  char_traits<char>::to_char_type ( eofTest1 )
           << "." << endl;

   eofTest2 = char_traits<char>::not_eof ( int2 );
   if ( !eofTest2 )
      cout << "The eofTest2 indicates int2 is an EOF character."
           << endl;
   else
      cout << "The eofTest1 returns: " << eofTest2
           << ", which is the character: "
           <<  char_traits<char>::to_char_type ( eofTest2 )
           << "." << endl;
}
```

Output

```
The char_type ch1 is x corresponding to int_type: 120.
The eofReturn is: -1
The eofTest1 returns: 120, which is the character: x.
The eofTest2 indicates int2 is an EOF character.
```

Requirements

Header: <string>

See Also

Reference

char_traits Class

Other Resources

char_traits Members

char_traits::to_char_type

Converts an **int_type** character to the corresponding **char_type** character and returns the result.

```
static char_type to_char_type(
   const int_type& _Ch
);
```

Parameters

_Ch

The **int_type** character to be represented as a **char_type**.

Return Value

The **char_type** character corresponding to the **int_type** character.

A value of *_Ch* that cannot be represented as such yields an unspecified result.

Remarks

The conversion operations to_int_type and **to_char_type** are inverse to each other, so that:

to_int_type (**to_char_type** (*x*)) == *x*

for any **int_type** *x* and

to_char_type (**to_int_type** (*x*)) == *x*

for any **char_type** *x*.

Example

```
// char_traits_to_char_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   char_traits<char>::char_type ch1 =  'a';
   char_traits<char>::char_type ch2 =  'b';
   char_traits<char>::char_type ch3 =  'a';

   // Converting from char_type to int_type
   char_traits<char>::int_type int1, int2 , int3;
   int1 =char_traits<char>:: to_int_type ( ch1 );
   int2 =char_traits<char>:: to_int_type ( ch2 );
   int3 =char_traits<char>:: to_int_type ( ch3 );

   cout << "The char_types and corresponding int_types are:"
        << "\n    ch1 = " << ch1 << " corresponding to int1 = "
        << int1 << "."
        << "\n    ch2 = " << ch2 << " corresponding to int1 = "
        << int2 << "."
        << "\n    ch3 = " << ch3 << " corresponding to int1 = "
        << int3 << "." << endl << endl;

   // Converting from int_type back to char_type
   char_traits<char>::char_type rec_ch1;
   rec_ch1 = char_traits<char>:: to_char_type ( int1);
   char_traits<char>::char_type rec_ch2;
   rec_ch2 = char_traits<char>:: to_char_type ( int2);

   cout << "The recovered char_types and corresponding int_types are:"
```

```
         << "\n    recovered ch1 = " << rec_ch1 << " from int1 = "
         << int1 << "."
         << "\n    recovered ch2 = " << rec_ch2 << " from int2 = "
         << int2 << "." << endl << endl;

    // Testing that the conversions are inverse operations
    bool b1 = char_traits<char>::eq ( rec_ch1 , ch1 );
    if ( b1 )
       cout << "The recovered char_type of ch1"
            << " is equal to the original ch1." << endl;
    else
       cout << "The recovered char_type of ch1"
            << " is not equal to the original ch1." << endl;

    // An equivalent and alternatively test procedure
    if ( rec_ch2 == ch2 )
       cout << "The recovered char_type of ch2"
            << " is equal to the original ch2." << endl;
    else
       cout << "The recovered char_type of ch2"
            << " is not equal to the original ch2." << endl;
 }
```

Output

```
The char_types and corresponding int_types are:
    ch1 = a corresponding to int1 = 97.
    ch2 = b corresponding to int1 = 98.
    ch3 = a corresponding to int1 = 97.

The recovered char_types and corresponding int_types are:
    recovered ch1 = a from int1 = 97.
    recovered ch2 = b from int2 = 98.

The recovered char_type of ch1 is equal to the original ch1.
The recovered char_type of ch2 is equal to the original ch2.
```

Requirements

Header: <string>

See Also

Reference

char_traits Class

Other Resources

char_traits Members

char_traits::to_int_type

Converts a **char_type** character to the corresponding **int_type** character and returns the result.

```
static int_type to_int_type(
   const char_type& _Ch
);
```

Parameters

_Ch

The **char_type** character to be represented as an **int_type**.

Return Value

The **int_type** character corresponding to the **char_type** character.

Remarks

The conversion operations **to_int_type** and to_char_type are inverse to each other, so that:

to_int_type (**to_char_type** (*x*)) == *x*

for any **int_type** *x*, and

to_char_type (**to_int_type** (*x*)) == *x*

for any **char_type** *x*.

Example

```
// char_traits_to_int_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   char_traits<char>::char_type ch1 = 'a';
   char_traits<char>::char_type ch2 = 'b';
   char_traits<char>::char_type ch3 = 'a';

   // Converting from char_type to int_type
   char_traits<char>::int_type int1, int2 , int3;
   int1 =char_traits<char>:: to_int_type ( ch1 );
   int2 =char_traits<char>:: to_int_type ( ch2 );
   int3 =char_traits<char>:: to_int_type ( ch3 );

   cout << "The char_types and corresponding int_types are:"
        << "\n    ch1 = " << ch1 << " corresponding to int1 = "
        << int1 << "."
        << "\n    ch2 = " << ch2 << " corresponding to int1 = "
        << int2 << "."
        << "\n    ch3 = " << ch3 << " corresponding to int1 = "
        << int3 << "." << endl << endl;

   // Converting from int_type back to char_type
   char_traits<char>::char_type rec_ch1;
   rec_ch1 = char_traits<char>:: to_char_type ( int1);
   char_traits<char>::char_type rec_ch2;
   rec_ch2 = char_traits<char>:: to_char_type ( int2);

   cout << "The recovered char_types and corresponding int_types are:"
        << "\n    recovered ch1 = " << rec_ch1 << " from int1 = "
        << int1 << "."
        << "\n    recovered ch2 = " << rec_ch2 << " from int2 = "
```

```
         << int2 << "." << endl << endl;

    // Testing that the conversions are inverse operations
    bool b1 = char_traits<char>::eq ( rec_ch1 , ch1 );
    if ( b1 )
       cout << "The recovered char_type of ch1"
            << " is equal to the original ch1." << endl;
    else
       cout << "The recovered char_type of ch1"
            << " is not equal to the original ch1." << endl;

    // An equivalent and alternatively test procedure
    if ( rec_ch2 == ch2 )
       cout << "The recovered char_type of ch2"
            << " is equal to the original ch2." << endl;
    else
       cout << "The recovered char_type of ch2"
            << " is not equal to the original ch2." << endl;
 }
```

Output

```
The char_types and corresponding int_types are:
    ch1 = a corresponding to int1 = 97.
    ch2 = b corresponding to int1 = 98.
    ch3 = a corresponding to int1 = 97.

The recovered char_types and corresponding int_types are:
    recovered ch1 = a from int1 = 97.
    recovered ch2 = b from int2 = 98.

The recovered char_type of ch1 is equal to the original ch1.
The recovered char_type of ch2 is equal to the original ch2.
```

Requirements

Header: <string>

See Also

Reference

char_traits Class

Other Resources

char_traits Members

<string> Specializations

For information about the specializations in <string>, see <string> Members.

char_traits<char> Class

A class that is a specialization of the template class **char_traits<CharType>** to an element of type **char**.

```
template<> class char_traits<char>;
```

Remarks

Specialization allows the class to take advantage of library functions that manipulate objects of this type **char**.

Example

See the typedefs and member functions of the template class char_traits Class<**CharType**> for examples involving elements of type **char**.

Requirements

Header: <string>

See Also

Other Resources

<string> Members

char_traits<wchar_t> Class

A class that is a specialization of the template class **char_traits<CharType>** to an element of type **wchar_t**.

```
template<> class char_traits<wchar_t>;
```

Remarks

Specialization allows the class to take advantage of library functions that manipulate objects of this type **char**.

Requirements

Header: <string>

See Also

Reference

char_traits Class

Thread Safety in the Standard C++ Library

Other Resources

<string> Members

<strstream>

Defines several classes that support iostreams operations on sequences stored in an allocated array of **char** object. Such sequences are easily converted to and from C strings.

For a list of all members of this header, see <strstream> Members.

```
#include <strstream>
```

Remarks

Objects of type **strstream** work with **char** *, which are C strings. Use <sstream> to work with objects of type basic_string.

Note
The classes in **<strstream>** are deprecated. Consider using the classes in **<sstream>** instead.

See Also

Reference

Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

<strstream> Members
Standard C++ Library Header Files

<strstream> Members

Reference

Classes

strstreambuf Class	The class describes a stream buffer that controls the transmission of elements to and from a sequence of elements stored in a **char** array object.
istrstream Class	The class describes an object that controls extraction of elements and encoded objects from a stream buffer of class strstreambuf.
ostrstream Class	The class describes an object that controls insertion of elements and encoded objects into a stream buffer of class strstreambuf.
strstream Class	The class describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class strstreambuf.

See Also

Reference

<strstream>
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

<strstream> Classes

For information about the classes in <strstream>, see <strstream> Members.

See Also

Reference

iostream Programming

iostreams Conventions

strstreambuf Class

Describes a stream buffer that controls the transmission of elements to and from a sequence of elements stored in a **char** array object.

For a list of all members of this type, see strstreambuf Members.

```
class strstreambuf : public streambuf
```

Remarks

Depending on how the object is constructed, it can be allocated, extended, and freed as necessary to accommodate changes in the sequence.

An object of class **strstreambuf** stores several bits of mode information as its **strstreambuf** mode. These bits indicate whether the controlled sequence:

- Has been allocated and needs to be freed eventually.
- Is modifiable.
- Is extendable by reallocating storage.
- Has been frozen and hence needs to be unfrozen before the object is destroyed, or freed (if allocated) by an agency other than the object.

A controlled sequence that is frozen cannot be modified or extended, regardless of the state of these separate mode bits.

The object also stores pointers to two functions that control **strstreambuf** allocation. If these are null pointers, the object devises its own method of allocating and freeing storage for the controlled sequence.

Note
This class is deprecated. Consider using stringbuf or wstringbuf instead.

Requirements

Header: <strstream>

See Also

Reference

streambuf
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

strstreambuf Members
<strstream> Members

strstreambuf Members

The following table lists the members of the strstreambuf Class. For a list of members inherited by the **strstreambuf** class, see streambuf.

Reference

Constructors

strstreambuf	Constructs an object of type **strstreambuf**.

Member Functions

freeze	Causes a stream buffer to be unavailable through stream buffer operations.
pcount	Returns a count of the number of elements written to the controlled sequence.
overflow	A protected virtual function that can be called when a new character is inserted into a full buffer.
pbackfail	A protected virtual member function that tries to put back an element into the input stream, and then make it the current element (pointed to by the next pointer).
seekoff	A protected virtual member function that tries to alter the current positions for the controlled streams.
seekpos	A protected virtual member function that tries to alter the current positions for the controlled streams.
str	Calls freeze, and then returns a pointer to the beginning of the controlled sequence.
underflow	A protected virtual function to extract the current element from the input stream.

See Also

Reference

streambuf

Thread Safety in the Standard C++ Library

iostream Programming

iostreams Conventions

Other Resources

<strstream> Members

strstreambuf Member Functions

For information about the member functions in the strstreambuf class, see strstreambuf Members.

See Also

Reference

iostream Programming

iostreams Conventions

strstreambuf::freeze

Causes a stream buffer to be unavailable through stream buffer operations.

```
void freeze(
   bool _Freezeit = true
);
```

Parameters

_Freezeit

A **bool** indicating whether you want the stream to be frozen.

Remarks

If *_Freezeit* is true, the function alters the stored **strstreambuf** mode to make the controlled sequence frozen. Otherwise, it makes the controlled sequence not frozen.

str implies **freeze**.

Note

A frozen buffer will not be freed during **strstreambuf** destruction. You must unfreeze the buffer before it is freed to avoid a memory leak.

Example

```
// strstreambuf_freeze.cpp
// compile with: /EHsc

#include <iostream>
#include <strstream>

using namespace std;

void report(strstream &x)
{
    if (!x.good())
        cout << "stream bad" << endl;
    else
        cout << "stream good" << endl;
}

int main()
{
    strstream x;

    x << "test1";
    cout << "before freeze: ";
    report(x);

    // Calling str freezes stream.
    cout.write(x.rdbuf()->str(), 5) << endl;
    cout << "after freeze: ";
    report(x);

    // Stream is bad now, wrote on frozen stream
    x << "test1.5";
    cout << "after write to frozen stream: ";
    report(x);

    // Unfreeze stream, but it is still bad
    x.rdbuf()->freeze(false);
    cout << "after unfreezing stream: ";
```

```
    report(x);

    // Clear stream
    x.clear();
    cout << "after clearing stream: ";
    report(x);

    x << "test3";
    cout.write(x.rdbuf()->str(), 10) << endl;

    // Clean up.  Failure to unfreeze stream will cause a
    // memory leak.
    x.rdbuf()->freeze(false);
}
```

Output

```
before freeze: stream good
test1
after freeze: stream good
after write to frozen stream: stream bad
after unfreezing stream: stream bad
after clearing stream: stream good
test1test3
```

Requirements

Header: <strstream>

See Also

Reference

strstreambuf Class

iostream Programming

iostreams Conventions

Other Resources

strstreambuf Members

strstreambuf::overflow

A protected virtual function that can be called when a new character is inserted into a full buffer.

```
virtual int overflow(
   int _Meta = EOF
);
```

Parameters

_Meta

The character to insert into the buffer, or **EOF**.

Return Value

If the function cannot succeed, it returns **EOF**. Otherwise, if *_Meta* == **EOF**, it returns some value other than **EOF**. Otherwise, it returns *_Meta*.

Remarks

If *_Meta* != **EOF**, the protected virtual member function tries to insert the element (**char**)*_Meta* into the output buffer. It can do so in various ways:

- If a write position is available, it can store the element into the write position and increment the next pointer for the output buffer.
- If the stored strstreambuf mode says the controlled sequence is modifiable, extendable, and not frozen, the function can make a write position available by allocating new for the output buffer. Extending the output buffer this way also extends any associated input buffer.

Requirements

Header: <strstream>

See Also

Reference

strstreambuf Class
iostream Programming
iostreams Conventions

Other Resources

strstreambuf Members

strstreambuf::pbackfail

A protected virtual member function that tries to put back an element into the input stream, and then makes it the current element (pointed to by the next pointer).

```
virtual int pbackfail(
   int _Meta = EOF
);
```

Parameters

_Meta

The character to insert into the buffer, or **EOF**.

Return Value

If the function cannot succeed, it returns **EOF**. Otherwise, if *_Meta* == **EOF**, it returns some value other than **EOF**. Otherwise, it returns *_Meta*.

Remarks

The protected virtual member function tries to put back an element into the input buffer, and then make it the current element (pointed to by the next pointer).

If *_Meta* == **EOF**, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by **ch** = (**char**)*_Meta*. The function can put back an element in various ways:

- If a putback position is available, and the element stored there compares equal to **ch**, it can decrement the next pointer for the input buffer.
- If a putback position is available, and if the strstreambuf mode says the controlled sequence is modifiable, the function can store **ch** into the putback position and decrement the next pointer for the input buffer.

Requirements

Header: <strstream>

See Also

Reference

strstreambuf Class
iostream Programming
iostreams Conventions

Other Resources

strstreambuf Members

strstreambuf::pcount

Returns a count of the number of elements written to the controlled sequence.

```
streamsize pcount( );
```

Return Value

A count of the number of elements written to the controlled sequence.

Remarks

Specifically, if pptr is a null pointer, the function returns zero. Otherwise, it returns **pptr** – pbase.

Example

```
// strstreambuf_pcount.cpp
// compile with: /EHsc
#include <iostream>
#include <strstream>
using namespace std;

int main( )
{
   strstream x;
   x << "test1";
   cout << x.rdbuf( )->pcount( ) << endl;
   x << "test2";
   cout << x.rdbuf( )->pcount( ) << endl;
}
```

Output

```
5
10
```

Requirements

Header: <strstream>

See Also

Reference

strstreambuf Class

iostream Programming

iostreams Conventions

Other Resources

strstreambuf Members

strstreambuf::seekoff

A protected virtual member function that tries to alter the current positions for the controlled streams.

```
virtual streampos seekoff(
   streamoff _Off,
   ios_base::seekdir _Way,
   ios_base::openmode _Which = ios_base::in | ios_base::out
);
```

Parameters

_Off

The position to seek for relative to *_Way*.

_Way

The starting point for offset operations. See seekdir for possible values.

_Which

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

If the function succeeds in altering either or both stream positions, it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position.

Remarks

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class strstreambuf, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence.

The new position is determined as follows:

- If *_Way* == **ios_base::beg**, the new position is the beginning of the stream plus *_Off*.
- If *_Way* == **ios_base::cur**, the new position is the current stream position plus *_Off*.
- If *_Way* == **ios_base::end**, the new position is the end of the stream plus *_Off*.

If *_Which* & **ios_base::in** is nonzero and the input buffer exist, the function alters the next position to read in the input buffer. If *_Which* & **ios_base::out** is also nonzero, *_Way* != **ios_base::cur**, and the output buffer exists, the function also sets the next position to write to match the next position to read.

Otherwise, if *_Which* & **ios_base::out** is nonzero and the output buffer exists, the function alters the next position to write in the output buffer. Otherwise, the positioning operation fails. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence.

Requirements

Header: <strstream>

See Also

Reference

strstreambuf Class
iostream Programming
iostreams Conventions

Other Resources

strstreambuf Members

strstreambuf::seekpos

A protected virtual member function that tries to alter the current positions for the controlled streams.

```
virtual streampos seekpos(
   streampos _Sp,
   ios_base::openmode _Which = ios_base::in | ios_base::out
);
```

Parameters

_Sp

The position to seek for.

_Which

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

If the function succeeds in altering either or both stream positions, it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position. To determine if the stream position is invalid, compare the return value with `pos_type(off_type(-1))`.

Remarks

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class strstreambuf, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence. The new position is determined by *_Sp*.

If *_Which* & **ios_base::in** is nonzero and the input buffer exists, the function alters the next position to read in the input buffer. If *_Which* & **ios_base::out** is nonzero and the output buffer exists, the function also sets the next position to write to match the next position to read. Otherwise, if *_Which* & **ios_base::out** is nonzero and the output buffer exists, the function alters the next position to write in the output buffer. Otherwise, the positioning operation fails. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence.

Requirements

Header: <strstream>

See Also

Reference

strstreambuf Class
iostream Programming
iostreams Conventions

Other Resources

strstreambuf Members

strstreambuf::str

Calls freeze, and then returns a pointer to the beginning of the controlled sequence.

```
char *str( );
```

Return Value

A pointer to the beginning of the controlled sequence.

Remarks

No terminating null element exists, unless you explicitly insert one.

Example

See strstreambuf::freeze for a sample that uses **str**.

Requirements

Header: <strstream>

See Also

Reference

strstreambuf Class
iostream Programming
iostreams Conventions

Other Resources

strstreambuf Members

strstreambuf::strstreambuf

Constructs an object of type **strstreambuf**.

```
explicit strstreambuf(
   streamsize _Count = 0
);
strstreambuf(
   void ( *_Allocfunc )( size_t ),
   void ( *_Freefunc )( void * )
);
strstreambuf(
   char *_Getptr,
   streamsize _Count,
   char *_Putptr = 0
);
strstreambuf(
   signed char *_Getptr,
   streamsize _Count,
   signed char *_Putptr = 0
);
strstreambuf(
   unsigned char *_Getptr,
   streamsize _Count,
   unsigned char *_Putptr = 0
);
strstreambuf(
   const char *_Getptr,
   streamsize _Count
);
strstreambuf(
   const signed char *_Getptr,
   streamsize _Count
);
strstreambuf(
   const unsigned char *_Getptr,
   streamsize _Count
);
```

Parameters

_Allocfunc

The function used to allocate buffer memory.

_Count

Determines the length of the buffer pointed to by *_Getptr*. If *_Getptr* is not an argument (first constructor form), a suggested allocation size for the buffers.

_Freefunc

The function used to free buffer memory.

_Getptr

A buffer used for input.

_Putptr

A buffer used for output.

Remarks

The first constructor stores a null pointer in all the pointers controlling the input buffer, the output buffer, and strstreambuf allocation. It sets the stored strstreambuf mode to make the controlled sequence modifiable and extendable. It also accepts *_Count* as a suggested initial allocation size.

The second constructor behaves like the first, except that it stores *_Allocfunc* as the pointer to the function to call to allocate storage and *_Freefunc* as the pointer to the function to call to free that storage.

The three constructors:

```
        strstreambuf(char *_Getptr, streamsize count,
    char *putptr = 0);
strstreambuf(signed char *_Getptr, streamsize count,
    signed char *putptr = 0);
strstreambuf(unsigned char *_Getptr, streamsize count,
    unsigned char *putptr = 0);
```

also behave like the first, except that *_Getptr* designates the array object used to hold the controlled sequence. (Hence, it must not be a null pointer.) The number of elements *N* in the array is determined as follows:

- If (*_Count* > 0), then *N* is **count**.
- If (*_Count* == 0), then *N* is **strlen**((**const char** *)*_Getptr*).
- If (*_Count* < 0), then *N* is **INT_MAX**.

If *_Putptr* is a null pointer, the function establishes just an input buffer by executing:

```
setg(_Getptr, _Getptr, _Getptr + N);
```

Otherwise, it establishes both input and output buffers by executing:

```
setg(_Getptr, _Getptr, _Putptr);
setp(_Putptr, _Getptr + N);
```

In this case, *_Putptr* must be in the interval [*_Getptr*, *_Getptr* + *N*].

Finally, the three constructors:

```
        strstreambuf(const char *_Getptr, streamsize _Count);
strstreambuf(const signed char *_Getptr, streamsize _Count);
strstreambuf(const unsigned char *_Getptr, streamsize _Count);
```

all behave the same as:

```
streambuf( (char *)_Getptr, _Count );
```

except that the stored mode makes the controlled sequence neither modifiable nor extendable.

Requirements

Header: <strstream>

See Also

Reference

strstreambuf Class

iostream Programming

iostreams Conventions

Other Resources

strstreambuf Members

strstreambuf::underflow

A protected virtual function to extract the current element from the input stream.

```
virtual int underflow( );
```

Return Value

If the function cannot succeed, it returns **EOF**. Otherwise, it returns the current element in the input stream, converted as described above.

Remarks

The protected virtual member function endeavors to extract the current element **ch** from the input buffer, then advance the current stream position, and return the element as (**int**)(**unsigned char**)**ch**. It can do so in only one way: if a read position is available, it takes **ch** as the element stored in the read position and advances the next pointer for the input buffer.

Requirements

Header: <strstream>

See Also

Reference

strstreambuf Class
iostream Programming
iostreams Conventions

Other Resources

strstreambuf Members

istrstream Class

Describes an object that controls extraction of elements and encoded objects from a stream buffer of class strstreambuf.

For a list of all members of this type, see istrstream Members.

```
class istrstream : public istream
```

Remarks

The object stores an object of class **strstreambuf**.

Note
This class is deprecated. Consider using istringstream or wistringstream instead.

Requirements

Header: <strstream>

See Also

Reference

istream
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

istrstream Members
<strstream> Members

istrstream Members

The following table lists the members of the istrstream Class. For a list of members inherited by the **istrstream** class, see istream.

Reference

Constructors

istrstream	Constructs an object of type **istrstream**.

Member Functions

rdbuf	Returns a pointer to the stream's associated **strstreambuf** object.
str	Calls freeze, and then returns a pointer to the beginning of the controlled sequence.

See Also

Reference

istrstream Class
istream
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

istrstream Member Functions

For information about the member functions in the istrstream class, see istrstream Members.

See Also

Reference

iostream Programming

iostreams Conventions

istrstream::istrstream

Constructs an object of type **istrstream**.

```
explicit istrstream(
   const char *_Ptr
);
explicit istrstream(
   char *_Ptr
);
istrstream(
   const char *_Ptr,
   streamsize _Count
);
istrstream(
   char *_Ptr,
   streamsize _Count
);
```

Parameters

_Count

The length of the buffer (*_Ptr*).

_Ptr

The contents with which the buffer is initialized.

Remarks

All the constructors initialize the base class by calling istream(**sb**), where **sb** is the stored object of class strstreambuf. The first two constructors also initialize **sb** by calling **strstreambuf**((**const char** *)*_Ptr*, 0). The remaining two constructors instead call **strstreambuf**((**const char** *)*_Ptr*, *_Count*).

Requirements

Header: <strstream>

See Also

Reference

istrstream Class

iostream Programming

iostreams Conventions

Other Resources

istrstream Members

istrstream::rdbuf

Returns a pointer to the stream's associated strstreambuf object.

```
strstreambuf *rdbuf( ) const
```

Return Value

A pointer to the stream's associated strstreambuf object.

Remarks

The member function returns the address of the stored stream buffer, of type pointer to strstreambuf.

Example

See strstreambuf::pcount for a sample that uses **rdbuf**.

Requirements

Header: <strstream>

See Also

Reference

istrstream Class
iostream Programming
iostreams Conventions

Other Resources

istrstream Members

istrstream::str

Calls freeze, and then returns a pointer to the beginning of the controlled sequence.

```
char *str( );
```

Return Value

A pointer to the beginning of the controlled sequence.

Remarks

The member function returns rdbuf -> str.

Example

See strstream::str for a sample that uses **str**.

Requirements

Header: <strstream>

See Also

Reference

istrstream Class
iostream Programming
iostreams Conventions

Other Resources

istrstream Members

ostrstream Class

Describes an object that controls insertion of elements and encoded objects into a stream buffer of class strstreambuf.

For a list of all members of this type, see ostrstream Members.

```
class ostrstream : public ostream
```

Remarks

The object stores an object of class **strstreambuf**.

Note
This class is deprecated. Consider using ostringstream or wostringstream instead.

Requirements

Header: <strstream>

See Also

Reference

ostream
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

ostrstream Members
<strstream> Members

ostrstream Members

The following table lists the members of the ostrstream Class. For a list of members inherited by the **ostrstream** class, see ostream.

Reference

Constructors

ostrstream	Constructs an object of type **ostrstream**.

Member Functions

freeze	Causes a stream buffer to be unavailable through stream buffer operations.
pcount	Returns a count of the number of elements written to the controlled sequence.
rdbuf	Returns a pointer to the stream's associated **strstreambuf** object.
str	Calls freeze, and then returns a pointer to the beginning of the controlled sequence.

See Also

Reference

ostrstream Class
ostream
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

ostrstream Member Functions

For information about the member functions in the ostrstream class, see ostrstream Members.

See Also

Reference

iostream Programming

iostreams Conventions

ostrstream::freeze

Causes a stream buffer to be unavailable through stream buffer operations.

```
void freeze(
   bool _Freezeit = true
);
```

Parameters

_Freezeit

A **bool** indicating whether you want the stream to be frozen.

Remarks

The member function calls rdbuf -> freeze(*_Freezeit*).

Example

See strstream::freeze for an example that uses **freeze**.

Requirements

Header: <strstream>

See Also

Reference

ostrstream Class
iostream Programming
iostreams Conventions

Other Resources

ostrstream Members

ostrstream::ostrstream

Constructs an object of type **ostrstream**.

```
ostrstream( );
ostrstream(
   char *_Ptr,
   streamsize _Count,
   ios_base::openmode _Mode = ios_base::out
);
```

Parameters

_Ptr

The buffer.

_Count

The size of the buffer in bytes.

_Mode

The input and output mode of the buffer. See ios_base::openmode for more information.

Remarks

Both constructors initialize the base class by calling ostream(**sb**), where **sb** is the stored object of class strstreambuf. The first constructor also initializes **sb** by calling **strstreambuf**. The second constructor initializes the base class one of two ways:

- If *_Mode* & **ios_base::app**== 0, then *_Ptr* must designate the first element of an array of *_Count* elements, and the constructor calls **strstreambuf**(*_Ptr*, *_Count*, *_Ptr*).
- Otherwise, *_Ptr* must designate the first element of an array of count elements that contains a C string whose first element is designated by *_Ptr*, and the constructor calls **strstreambuf**(*_Ptr*, *_Count*, *_Ptr* + **strlen**(*_Ptr*)).

Requirements

Header: <strstream>

See Also

Reference

ostrstream Class
iostream Programming
iostreams Conventions

Other Resources

ostrstream Members

ostrstream::pcount

Returns a count of the number of elements written to the controlled sequence.

```
streamsize pcount( ) const;
```

Return Value

The number of elements written to the controlled sequence.

Remarks

The member function returns rdbuf -> pcount.

Example

See strstream::pcount for a sample that uses **pcount**.

Requirements

Header: <strstream>

See Also

Reference

ostrstream Class
iostream Programming
iostreams Conventions

Other Resources

ostrstream Members

ostrstream::rdbuf

Returns a pointer to the stream's associated strstreambuf object.

```
strstreambuf *rdbuf( ) const
```

Return Value

A pointer to the stream's associated strstreambuf object.

Remarks

The member function returns the address of the stored stream buffer of type **pointer** to strstreambuf.

Example

See strstreambuf::pcount for a sample that uses **rdbuf**.

Requirements

Header: <strstream>

See Also

Reference

ostrstream Class
iostream Programming
iostreams Conventions

Other Resources

ostrstream Members

ostrstream::str

Calls freeze, and then returns a pointer to the beginning of the controlled sequence.

```
char *str( );
```

Return Value

A pointer to the beginning of the controlled sequence.

Remarks

The member function returns rdbuf -> str.

Example

See strstream::str for a sample that uses **str**.

Requirements

Header: <strstream>

See Also

Reference

ostrstream Class
iostream Programming
iostreams Conventions

Other Resources

ostrstream Members

strstream Class

Describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class strstreambuf.

For a list of all members of this type, see strstream Members.

```
class strstream : public iostream
```

Remarks

The object stores an object of class **strstreambuf**.

Note
This class is deprecated. Consider using stringstream or wstringstream instead.

Requirements

Header: <strstream>

See Also

Reference

iostream
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

Other Resources

strstream Members
<strstream> Members

strstream Members

The following table lists the members of the strstream Class. For a list of members inherited by the **strstream** class, see iostream.

Reference

Constructors

strstream	Constructs an object of type **strstream**.

Member Functions

freeze	Causes a stream buffer to be unavailable through stream buffer operations.
pcount	Returns a count of the number of elements written to the controlled sequence.
rdbuf	Returns a pointer to the stream's associated **strstreambuf** object.
str	Calls freeze, and then returns a pointer to the beginning of the controlled sequence.

See Also

Reference

strstream Class
iostream
Thread Safety in the Standard C++ Library
iostream Programming
iostreams Conventions

strstream Member Functions

For information about the member functions in the strstream class, see strstream Members.

See Also

Reference

iostream Programming

iostreams Conventions

strstream::freeze

Causes a stream buffer to be unavailable through stream buffer operations.

```
void freeze(
   bool _Freezeit = true
);
```

Parameters

_Freezeit

A **bool** indicating whether you want the stream to be frozen.

Remarks

The member function calls rdbuf -> freeze(*_Freezeit*).

Example

See strstreambuf::freeze for an example that uses **freeze**.

Requirements

Header: <strstream>

See Also

Reference

strstream Class
iostream Programming
iostreams Conventions

Other Resources

strstream Members

strstream::pcount

Returns a count of the number of elements written to the controlled sequence.

```
streamsize pcount( ) const;
```

Return Value

The number of elements written to the controlled sequence.

Remarks

The member function returns rdbuf -> pcount.

Example

See strstreambuf::pcount for a sample of using pcount.

Requirements

Header: <strstream>

See Also

Reference

strstream Class
iostream Programming
iostreams Conventions

Other Resources

strstream Members

strstream::rdbuf

Returns a pointer to the stream's associated strstreambuf object.

```
strstreambuf *rdbuf( ) const
```

Return Value

A pointer to the stream's associated strstreambuf object.

Remarks

The member function returns the address of the stored stream buffer of type **pointer** to strstreambuf.

Example

See strstreambuf::pcount for a sample that uses **rdbuf**.

Requirements

Header: <strstream>

See Also

Reference

strstream Class
iostream Programming
iostreams Conventions

Other Resources

strstream Members

strstream::str

Calls freeze, and then returns a pointer to the beginning of the controlled sequence.

```
char *str( );
```

Return Value

A pointer to the beginning of the controlled sequence.

Remarks

The member function returns rdbuf -> str.

Example

See strstreambuf::str for a sample that uses **str**.

Requirements

Header: <strstream>

See Also

Reference

strstream Class
iostream Programming
iostreams Conventions

Other Resources

strstream Members

strstream::strstream

Constructs an object of type **strstream**.

```
strstream( );
strstream(
   char *_Ptr,
   streamsize _Count,
   ios_base::openmode _Mode = ios_base::in | ios_base::out
);
```

Parameters

_Count

The size of the buffer.

_Mode

The input and output mode of the buffer. See ios_base::openmode for more information.

_Ptr

The buffer.

Remarks

Both constructors initialize the base class by calling streambuf(**sb**), where **sb** is the stored object of class strstreambuf. The first constructor also initializes **sb** by calling strstreambuf. The second constructor initializes the base class one of two ways:

- If *_Mode* & **ios_base::app**== 0, then *_Ptr* must designate the first element of an array of *_Count* elements, and the constructor calls **strstreambuf**(*_Ptr*, *_Count*, *_Ptr*).
- Otherwise, *_Ptr* must designate the first element of an array of count elements that contains a C string whose first element is designated by *_Ptr*, and the constructor calls **strstreambuf**(*_Ptr*, *_Count*, *_Ptr* + **strlen**(*_Ptr*)).

Requirements

Header: <strstream>

See Also

Reference

strstream Class
iostream Programming
iostreams Conventions

Other Resources

strstream Members

<utility>

Defines Standard Template Library (STL) types, functions, and operators that help to construct and manage pairs of objects, which are useful whenever two objects need to be treated as if they were one.

For a list of all members of this header, see <utility> Members.

```
#include <utility>
```

Remarks

Pairs are widely used in the Standard C++ Library. They are required both as the arguments and return values for various functions and as element types for containers such as map class and multimap class. The <utility> header is automatically included by <map> to assist in managing their key/value pair type elements.

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<utility> Members

Standard C++ Library Header Files

<utility> Members

Reference

Functions

make_pair	A template helper function used to construct objects of type **pair**, where the component types are based on the dat a types passed as parameters.

Operators

operator!=	Tests if the pair object on the left side of the operator is not equal to the pair object on the right side.
operator==	Tests if the pair object on the left side of the operator is equal to the pair object on the right side.
operator<	Tests if the pair object on the left side of the operator is less than the pair object on the right side.
operator<=	Tests if the pair object on the left side of the operator is less than or equal to the pair object on the right side.
operator>	Tests if the pair object on the left side of the operator is greater than the pair object on the right side.
operator>=	Tests if the pair object on the left side of the operator is greater than or equal to the pair object on the right side.

Classes

pair	A type that provides for the ability to treat two objects as a single object.

See Also

Reference

<utility>

Thread Safety in the Standard C++ Library

<utility> Functions

For information about the functions in <utility>, see <utility> Members.

make_pair

A template helper function used to construct objects of type **pair**, where the component types are based on the data types passed as parameters.

```
template<class Type1, class Type2>
   pair<Type1, Type2> make_pair(
      Type1 _Val1,
      Type2 _Val2
   );
```

Parameters

_Val1

Value initializing the first element of **pair**.

_Val2

Value initializing the second element of **pair**.

Return Value

The pair object constructed: **pair**<**Type1**, **Type2**>(*_Val1*, *_Val2*).

Remarks

One advantage of **make_pair** is that the types of objects being stored are determined automatically by the compiler and do not need to be explicitly specified by the user.

The **make_pair** helper function also makes it possible to pass two values to a function that requires a pair as an input parameter.

Example

See pair for an example of how to use the helper function **make_pair** to declare and initialize a pair.

Requirements

Header: <utility>

See Also

Reference

make_pair (STL Samples)

Other Resources

<utility> Members

<utility> Operators

For information about the operators in <utility>, see <utility> Members.

operator!= (<utility>)

Tests if the pair object on the left side of the operator is not equal to the pair object on the right side.

```
template<Class Type>
   bool operator!=(
      const Type& _Left,
      const Type& _Right
      );
template<class Type1, class Type2>
   bool operator!=(
      const pair<Type1, Type2>& _Left,
      const pair<Type1, Type2>& _Right
   );
```

Parameters

_Left

An object of type **pair.**

_Right

An object of type **pair**.

Return Value

true if the pairs are not equal; **false** if the pairs are equal.

Remarks

One pair is equal to another pair if each of their respective elements is equal. Two pairs are unequal if either the first or the second element of one is not equal to the corresponding element of the other pair.

Example

```
// utility_op_ne.cpp
// compile with: /EHsc
#include <utility>
#include <iomanip>
#include <iostream>

int main( )
{
   using namespace std;
   pair <int, double> p1, p2, p3;

   p1 = make_pair ( 10, 1.11e-1 );
   p2 = make_pair ( 1000, 1.11e-3 );
   p3 = make_pair ( 10, 1.11e-1 );

   cout.precision ( 3 );
   cout << "The pair p1 is: ( " << p1.first << ", "
        << p1.second << " )." << endl;
   cout << "The pair p2 is: ( " << p2.first << ", "
        << p2.second << " )." << endl;
   cout << "The pair p3 is: ( " << p3.first << ", "
        << p3.second << " )." << endl << endl;

   if ( p1 != p2 )
      cout << "The pairs p1 and p2 are not equal." << endl;
   else
      cout << "The pairs p1 and p2 are equal." << endl;

   if ( p1 != p3 )
      cout << "The pairs p1 and p3 are not equal." << endl;
   else
```

```
        cout << "The pairs p1 and p3 are equal." << endl;
}
```

Output

```
The pair p1 is: ( 10, 0.111 ).
The pair p2 is: ( 1000, 0.00111 ).
The pair p3 is: ( 10, 0.111 ).

The pairs p1 and p2 are not equal.
The pairs p1 and p3 are equal.
```

Requirements

Header: <utility>

See Also

Other Resources

<utility> Members

operator== (<utility>)

Tests if the pair object on the left side of the operator is equal to the pair object on the right side.

```
template<class Type1, class Type2>
   bool operator==(
      const pair<Type1, Type2>& _Left,
      const pair<Type1, Type2>& _Right
);
```

Parameters

_Left

An object of type **pair.**

_Right

An object of type **pair**.

Return Value

true if the pairs are equal; **false** if the **pair**s are not equal.

Remarks

One pair is equal to another pair if each of their respective elements is equal. The function returns *_Left*.**first** == *_Right*.**first** && *_Left*.**second** == *_Right*.**second**. Two pairs are unequal if either the first or the second element of one is not equal to the corresponding element of the other pair.

Example

```
// utility_op_eq.cpp
// compile with: /EHsc
#include <utility>
#include <iomanip>
#include <iostream>

int main( )
{
   using namespace std;
   pair <int, double> p1, p2, p3;

   p1 = make_pair ( 10, 1.11e-1 );
   p2 = make_pair ( 1000, 1.11e-3 );
   p3 = make_pair ( 10, 1.11e-1 );

   cout.precision ( 3 );
   cout << "The pair p1 is: ( " << p1.first << ", "
        << p1.second << " )." << endl;
   cout << "The pair p2 is: ( " << p2.first << ", "
        << p2.second << " )." << endl;
   cout << "The pair p3 is: ( " << p3.first << ", "
        << p3.second << " )." << endl << endl;

   if ( p1 == p2 )
      cout << "The pairs p1 and p2 are equal." << endl;
   else
      cout << "The pairs p1 and p2 are not equal." << endl;

   if ( p1 == p3 )
      cout << "The pairs p1 and p3 are equal." << endl;
   else
      cout << "The pairs p1 and p3 are not equal." << endl;
}
```

Output

```
The pair p1 is: ( 10, 0.111 ).
The pair p2 is: ( 1000, 0.00111 ).
The pair p3 is: ( 10, 0.111 ).

The pairs p1 and p2 are not equal.
The pairs p1 and p3 are equal.
```

Requirements

Header: <utility>

See Also

Other Resources

<utility> Members

operator< (<utility>)

Tests if the pair object on the left side of the operator is less than the pair object on the right side.

```
template<class Type1, class Type2>
   bool operator<(
      const pair<Type1, Type2>& _Left,
      const pair<Type1, Type2>& _Right
      );
```

Parameters

_Left

An object of type **pair** on the left side of the operator.

_Right

An object of type **pair** on the right side of the operator.

Return Value

true if the **pair** on the left side of the operator is strictly less than the **pair** on the right side of the operator; otherwise **false**.

Remarks

The *_Left* **pair** object is said to be strictly less than the *_Right* **pair** object if *_Left* is less than and not equal *_Right*.

In a comparison of pairs, the values' first elements of the two pairs have the highest priority. If they differ, then the result of their comparison is taken as the result of the comparison of the pair. If the values of the first elements are not different, then the values of the second elements are compared and the result of their comparison is taken as the result of the comparison of the pair.

Example

```
// utility_op_lt.cpp
// compile with: /EHsc
#include <utility>
#include <iomanip>
#include <iostream>

int main( )
{
   using namespace std;
   pair <int, double> p1, p2, p3;

   p1 = make_pair ( 10, 2.22e-1 );
   p2 = make_pair ( 100, 1.11e-1 );
   p3 = make_pair ( 10, 1.11e-1 );

   cout.precision ( 3 );
   cout << "The pair p1 is: ( " << p1.first << ", "
        << p1.second << " )." << endl;
   cout << "The pair p2 is: ( " << p2.first << ", "
        << p2.second << " )." << endl;
   cout << "The pair p3 is: ( " << p3.first << ", "
        << p3.second << " )." << endl << endl;

   if ( p1 < p2 )
      cout << "The pair p1 is less than the pair p2." << endl;
   else
      cout << "The pair p1 is not less than the pair p2." << endl;

   if ( p1 < p3 )
      cout << "The pair p1 is less than the pair p3." << endl;
   else
      cout << "The pair p1 is not less than the pair p3." << endl;
```

```
}
```

Output

```
The pair p1 is: ( 10, 0.222 ).
The pair p2 is: ( 100, 0.111 ).
The pair p3 is: ( 10, 0.111 ).

The pair p1 is less than the pair p2.
The pair p1 is not less than the pair p3.
```

Requirements

Header: <utility>

See Also

Other Resources

<utility> Members

operator<= (<utility>)

Tests if the pair object on the left side of the operator is less than or equal to the pair object on the right side.

```
template<Class Type>
   bool operator<=(
      const Type& _Left,
      const Type& _Right
      );
template<class Type1, class Type2>
   bool operator<=(
      const pair<Type1, Type2>& _Left,
      const pair<Type1, Type2>& _Right
      );
```

Parameters

_Left

An object of type **pair** on the left side of the operator.

_Right

An object of type **pair** on the right side of the operator.

Return Value

true if the **pair** on the left side of the operator is less than or equal to the **pair** on the right side of the operator; otherwise **false**.

Remarks

In a comparison of pairs, the values' first elements of the two pairs have the highest priority. If they differ, then the result of their comparison is taken as the result of the comparison of the pair. If the values of the first elements are not different, then the values of the second elements are compared and the result of their comparison is taken as the result of the comparison of the pair.

Example

```
// utility_op_le.cpp
// compile with: /EHsc
#include <utility>
#include <iomanip>
#include <iostream>

int main( )
{
   using namespace std;
   pair <int, double> p1, p2, p3, p4;

   p1 = make_pair ( 10, 2.22e-1 );
   p2 = make_pair ( 100, 1.11e-1 );
   p3 = make_pair ( 10, 1.11e-1 );
   p4 = make_pair ( 10, 2.22e-1 );

   cout.precision ( 3 );
   cout << "The pair p1 is: ( " << p1.first << ", "
        << p1.second << " )." << endl;
   cout << "The pair p2 is: ( " << p2.first << ", "
        << p2.second << " )." << endl;
   cout << "The pair p3 is: ( " << p3.first << ", "
        << p3.second << " )." << endl;
   cout << "The pair p4 is: ( " << p4.first << ", "
        << p4.second << " )." << endl << endl;

   if ( p1 <= p2 )
      cout << "The pair p1 is less than or equal to the pair p2." << endl;
```

```
    else
       cout << "The pair p1 is greater than the pair p2." << endl;

    if ( p1 <= p3 )
       cout << "The pair p1 is less than or equal to the pair p3." << endl;
    else
       cout << "The pair p1 is greater than the pair p3." << endl;

    if ( p1 <= p4 )
       cout << "The pair p1 is less than or equal to the pair p4." << endl;
    else
       cout << "The pair p1 is greater than the pair p4." << endl;
 }
```

Output

```
The pair p1 is: ( 10, 0.222 ).
The pair p2 is: ( 100, 0.111 ).
The pair p3 is: ( 10, 0.111 ).
The pair p4 is: ( 10, 0.222 ).

The pair p1 is less than or equal to the pair p2.
The pair p1 is greater than the pair p3.
The pair p1 is less than or equal to the pair p4.
```

Requirements

Header: <utility>

See Also

Other Resources

<utility> Members

operator> (<utility>)

Tests if the pair object on the left side of the operator is greater than the pair object on the right side.

```
template<Class Type>
   bool operator>(
      const Type& _Left,
      const Type& _Right
      );
template<class Type1, class Type2>
   bool operator>(
      const pair<Type1, Type2>& _Left,
      const pair<Type1, Type2>& _Right
      );
```

Parameters

_Left

An object of type **pair** on the left side of the operator.

_Right

An object of type **pair** on the right side of the operator.

Return Value

true if the **pair** on the left side of the operator is strictly greater than the **pair** on the right side of the operator; otherwise **false**.

Remarks

The *_Left* **pair** object is said to be strictly greater than the *_Right* **pair** object if *_Left* is greater than and not equal *_Right*.

In a comparison of pairs, the values' first elements of the two pairs have the highest priority. If they differ, then the result of their comparison is taken as the result of the comparison of the pair. If the values of the first elements are not different, then the values of the second elements are compared and the result of their comparison is taken as the result of the comparison of the pair.

Example

```
// utility_op_gt.cpp
// compile with: /EHsc
#include <utility>
#include <iomanip>
#include <iostream>

int main( )
{
   using namespace std;
   pair <int, double> p1, p2, p3, p4;

   p1 = make_pair ( 10, 2.22e-1 );
   p2 = make_pair ( 100, 1.11e-1 );
   p3 = make_pair ( 10, 1.11e-1 );
   p4 = make_pair ( 10, 2.22e-1 );

   cout.precision ( 3 );
   cout << "The pair p1 is: ( " << p1.first << ", "
        << p1.second << " )." << endl;
   cout << "The pair p2 is: ( " << p2.first << ", "
        << p2.second << " )." << endl;
   cout << "The pair p3 is: ( " << p3.first << ", "
        << p3.second << " )." << endl;
   cout << "The pair p4 is: ( " << p4.first << ", "
        << p4.second << " )." << endl << endl;

   if ( p1 > p2 )
```

```
      cout << "The pair p1 is greater than the pair p2." << endl;
   else
      cout << "The pair p1 is not greater than the pair p2." << endl;

   if ( p1 > p3 )
      cout << "The pair p1 is greater than the pair p3." << endl;
   else
      cout << "The pair p1 is not greater than the pair p3." << endl;

   if ( p1 > p4 )
      cout << "The pair p1 is greater than the pair p4." << endl;
   else
      cout << "The pair p1 is not greater than the pair p4." << endl;
}
```

Output

```
The pair p1 is: ( 10, 0.222 ).
The pair p2 is: ( 100, 0.111 ).
The pair p3 is: ( 10, 0.111 ).
The pair p4 is: ( 10, 0.222 ).

The pair p1 is not greater than the pair p2.
The pair p1 is greater than the pair p3.
The pair p1 is not greater than the pair p4.
```

Requirements

Header: <utility>

See Also

Other Resources

<utility> Members

operator>= (<utility>)

Tests if the pair object on the left side of the operator is greater than or equal to the pair object on the right side.

```
template<Class Type>
   bool operator>=(
      const Type& _Left,
      const Type& _Right
);
template<class Type1, class Type2>
   bool operator>=(
      const pair<Type1, Type2>& _Left,
      const pair<Type1, Type2>& _Right
);
```

Parameters

_Left

An object of type **pair** on the left side of the operator.

_Right

An object of type **pair** on the right side of the operator.

Return Value

true if the **pair** on the left side of the operator is greater than or equal to the **pair** on the right side of the operator; otherwise **false**.

Remarks

In a comparison of pairs, the values' first elements of the two pairs have the highest priority. If they differ, then the result of their comparison is taken as the result of the comparison of the pair. If the values of the first elements are not different, then the values of the second elements are compared and the result of their comparison is taken as the result of the comparison of the pair.

Example

```
// utility_op_ge.cpp
// compile with: /EHsc
#include <utility>
#include <iomanip>
#include <iostream>

int main( )
{
   using namespace std;
   pair <int, double> p1, p2, p3, p4;

   p1 = make_pair ( 10, 2.22e-1 );
   p2 = make_pair ( 100, 1.11e-1 );
   p3 = make_pair ( 10, 1.11e-1 );
   p4 = make_pair ( 10, 2.22e-1 );

   cout.precision ( 3 );
   cout << "The pair p1 is: ( " << p1.first << ", "
        << p1.second << " )." << endl;
   cout << "The pair p2 is: ( " << p2.first << ", "
        << p2.second << " )." << endl;
   cout << "The pair p3 is: ( " << p3.first << ", "
        << p3.second << " )." << endl;
   cout << "The pair p4 is: ( " << p4.first << ", "
        << p4.second << " )." << endl << endl;

   if ( p1 >= p2 )
      cout << "Pair p1 is greater than or equal to pair p2." << endl;
```

```
    else
       cout << "The pair p1 is less than the pair p2." << endl;

    if ( p1 >= p3 )
       cout << "Pair p1 is greater than or equal to pair p3." << endl;
    else
       cout << "The pair p1 is less than the pair p3." << endl;

    if ( p1 >= p4 )
       cout << "Pair p1 is greater than or equal to pair p4." << endl;
    else
       cout << "The pair p1 is less than the pair p4." << endl;
 }
```

Output

```
The pair p1 is: ( 10, 0.222 ).
The pair p2 is: ( 100, 0.111 ).
The pair p3 is: ( 10, 0.111 ).
The pair p4 is: ( 10, 0.222 ).

The pair p1 is less than the pair p2.
Pair p1 is greater than or equal to pair p3.
Pair p1 is greater than or equal to pair p4.
```

Requirements

Header: <utility>

See Also

Other Resources

<utility> Members

<utility> Classes

For information about the classes in <utility>, see <utility> Members.

pair Class

A class that provides for the ability to treat two objects as a single object.

```
template<class Type1, class Type2>
   struct pair
   {
   typedef Type1 first_type;
   typedef Type2 second_type
   Type1 first;
   Type2 second;
   pair( );
   pair(
      const Type1& __Val1,
      const Type2& __Val2
   );
   template<class Other1, class Other2>
      pair(
         const pair<Other1, Other2>& _Right
            );
   };
```

Parameters

_Val1

Value initializing the first element of **pair**.

_Val2

Value initializing the second element of **pair**.

_Right

A pair whose values are to be used to initialize the elements of another pair.

Return Value

The first (default) constructor initializes first element of the pair to the default of type **Type1** and second element to default of type **Type2**.

The second constructor initializes first element of the pair to *_Val1* and second to *_Val2*.

The third (template) constructor initializes first element of the pair to *_Right*.**first** and second to *_Right*.**second**.

Remarks

The template class stores a pair of objects of type **Type1** and **Type2**, respectively. The type **first_type** is the same as the template parameter **Type1** and the type **second_type** is the same as the template parameter **Type2**. **Type1** and **Type2** each need supply only a default constructor, a single-argument constructor, and a destructor. All members of the type **pair** are public, because the type is declared as a **struct** rather than as a **class**. The two most common uses for a pair are as return types for functions that return two values and as elements for the associative container classes map Class and multimap Class that have both a key and a value type associated with each element. The latter satisfies the requirements for a pair associative container and has a value type of the form **pair**<**const key_type**, **mapped_type**>.

Example

```
// utility_pair.cpp
// compile with: /EHsc
#include <utility>
#include <map>
#include <iomanip>
#include <iostream>

int main( )
{
   using namespace std;
```

```
    // Using the constructor to declare and initialize a pair
    pair <int, double> p1 ( 10, 1.1e-2 );

    // Compare using the helper function to declare and initialize a pair
    pair <int, double> p2;
    p2 = make_pair ( 10, 2.22e-1 );

    // Making a copy of a pair
    pair <int, double> p3 ( p1 );

    cout.precision ( 3 );
    cout << "The pair p1 is: ( " << p1.first << ", "
         << p1.second << " )." << endl;
    cout << "The pair p2 is: ( " << p2.first << ", "
         << p2.second << " )." << endl;
    cout << "The pair p3 is: ( " << p3.first << ", "
         << p3.second << " )." << endl;

    // Using a pair for a map element
    map <int, int> m1;
    map <int, int>::iterator m1_Iter;

    typedef pair <int, int> Map_Int_Pair;

    m1.insert ( Map_Int_Pair ( 1, 10 ) );
    m1.insert ( Map_Int_Pair ( 2, 20 ) );
    m1.insert ( Map_Int_Pair ( 3, 30 ) );

    cout << "The element pairs of the map m1 are:";
    for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
       cout << " ( " << m1_Iter -> first << ", "
            << m1_Iter -> second << " )";
    cout   << "." << endl;

    // Using pair as a return type for a function
    pair< map<int,int>::iterator, bool > pr1, pr2;
    pr1 = m1.insert ( Map_Int_Pair ( 4, 40 ) );
    pr2 = m1.insert ( Map_Int_Pair (1, 10 ) );

    if( pr1.second == true )
    {
       cout << "The element (4,40) was inserted successfully in m1."
            << endl;
    }
    else
    {
       cout << "The element with a key value of\n"
            << " ( (pr1.first) -> first ) = " << ( pr1.first ) -> first
            << " is already in m1,\n so the insertion failed." << endl;
    }

    if( pr2.second == true )
    {
       cout << "The element (1,10) was inserted successfully in m1."
            << endl;
    }
    else
    {
       cout << "The element with a key value of\n"
            << " ( (pr2.first) -> first ) = " << ( pr2.first ) -> first
            << " is already in m1,\n so the insertion failed." << endl;
    }
  }
```

Output

```
The pair p1 is: ( 10, 0.011 ).
```

```
The pair p2 is: ( 10, 0.222 ).
The pair p3 is: ( 10, 0.011 ).
The element pairs of the map m1 are: ( 1, 10 ) ( 2, 20 ) ( 3, 30 ).
The element (4,40) was inserted successfully in m1.
The element with a key value of
 ( (pr2.first) -> first ) = 1 is already in m1,
 so the insertion failed.
```

Requirements

Header: <utility>

See Also

Reference

Pair Logical Operator

Thread Safety in the Standard C++ Library

Other Resources

<utility> Members

<valarray>

Defines the template class valarray and numerous supporting template classes and functions.

For a list of all members of this header, see <valarray> Members.

```
#include <valarray>
```

Remarks

These template classes and functions are permitted unusual latitude in the interest of improved performance. Specifically, any function returning type **valarray**<T1> may return an object of some other type T2. In that case, any function that accepts one or more arguments of type **valarray**<T2> must have overloads that accept arbitrary combinations of those arguments, each replaced with an argument of type T2.

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<valarray> Members

Standard C++ Library Header Files

<valarray> Members

Reference

Functions

abs	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the absolute value of the elements of the input valarray.
acos	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the arccosine of the elements of the input valarray.
asin	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the arcsine of the elements of the input valarray.
atan	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the principal value of the arctangent of the elements of the input valarray.
atan2	Returns a valarray whose elements are equal to the arctangent of the Cartesian components specified by a combination of constants and elements of valarrays.
cos	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the cosine of the elements of the input valarray.
cosh	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the hyperbolic cosine of the elements of the input valarray.
exp	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the natural exponential of the elements of the input valarray.
log	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the natural logarithm of the elements of the input valarray.
log10	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the base 10 or common logarithm of the elements of the input valarray.
pow	Operates on the elements of input valarrays and constants, returning a valarray whose elements are equal to a base specified either by the elements of an input valarray or a constant raised to an exponent specified either by the elements of an input valarray or a constant.
sin	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the sine of the elements of the input valarray.
sinh	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the hyperbolic sine of the elements of the input valarray.
sqrt	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the square root of the elements of the input valarray.
tan	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the tangent of the elements of the input valarray.
tanh	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the hyperbolic tangent of the elements of the input valarray.

Operators

operator!=	Tests whether the corresponding elements of two equally sized valarrays are unequal or whether all the elements of a valarray are unequal a specified value of the valarray's element type.
operator%	Obtains the remainder of dividing the corresponding elements of two equally sized valarrays or of dividing a valarray by a specified value of the valarray's element type or of dividing a specified value by a valarray.
operator&	Obtains the bitwise **AND** between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the element type.
operator&&	Obtains the logical **AND** between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the valarray's element type.
operator>	Tests whether the elements of one valarray are greater than the elements of an equally sized valarray or whether all the elements of a valarray are greater or less than a specified value of the valarray's element type.
operator>>	Right-shifts the bits for each element of a valarray a specified number of positions or by an element-wise amount specified by a second valarray.
operator>=	Tests whether the elements of one valarray are greater than or equal to the elements of an equally sized valarray or whether all the elements of a valarray are greater than or equal to or less than or equal to a specified value.
operator<	Tests whether the elements of one valarray are less than the elements of an equally sized valarray or whether all the elements of a valarray are greater or less than a specified value.
operator<<	Left shifts the bits for each element of a valarray a specified number of positions or by an element-wise amount specified by a second valarray.
operator<=	Tests whether the elements of one valarray are less than or equal to the elements of an equally sized valarray or whether all the elements of a valarray are greater than or equal to or less than or equal to a specified value.
operator*	Obtains the element-wise product between corresponding elements of two equally sized valarrays or of between a valarray a specified value of the valarray's element type.
operator+	Obtains the element-wise sum between corresponding elements of two equally sized valarrays or of between a valarray a specified value of the valarray's element type.
operator-	Obtains the element-wise difference between corresponding elements of two equally sized valarrays or of between a valarray a specified value of the valarray's element type.
operator/	Obtains the element-wise quotient between corresponding elements of two equally sized valarrays or of between a valarray a specified value of the valarray's element type.
operator==	Tests whether the corresponding elements of two equally sized valarrays are equal or whether all the elements of a valarray are equal a specified value of the valarray's element type.
operator^	Obtains the bitwise exclusive **OR** between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the element type.
operator|	Obtains the bitwise **OR** between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the element type.
operator||	Obtains the logical **OR** between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the valarray's element type.

Classes

gslice Class	A utility class to valarray that is used to define multi-dimensional slices of a valarray.
gslice_array Class	An internal, auxiliary template class that supports general slice objects by providing operations between subset arrays defined by the general slice of a valarray.
indirect_array Class	An internal, auxiliary template class that supports objects that are subsets of valarrays by providing operations between subset arrays defined by specifying a subset of indices of a parent valarray.
mask_array Class	An internal, auxiliary template class that supports objects that are subsets of parent valarrays, specified with a Boolean expression, by providing operations between the subset arrays.
slice Class	A utility class to valarray that is used to define one-dimensional, vector-like subsets of a valarray.
slice_array Class	An internal, auxiliary template class that supports slice objects by providing operations between subset arrays defined by the slice of a valarray.
valarray Class	The template class describes an object that controls a sequence of elements of type **Type** that are stored as an array and designed for performing high-speed mathematical operations, optimized for computational performance.

Specializations

valarray<bool> Class	A specialized version of the template class valarray<**Type**> to elements of type **bool**.

See Also

Reference

<valarray>

Thread Safety in the Standard C++ Library

<valarray> Functions

For information about the functions in <valarray>, see <valarray> Members.

abs (<valarray>)

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the absolute value of the elements of the input valarray.

```
template<class Type>
   valarray<Type> abs(
      const valarray<Type>& _Left
   );
```

Parameters

_Left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the absolute value of the elements of the input valarray.

Example

```
// valarray_abs.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> va1 ( 9 ), va2 ( 9 );
   for ( i = 0 ; i < 4 ; i++ )
      va1 [ i ] =  -i;
   for ( i = 4 ; i < 9 ; i++ )
      va1 [ i ] =  i;

   cout << "The initial valarray is: ";
      for (i = 0 ; i < 9 ; i++ )
         cout << va1 [ i ] << " ";
   cout << "." << endl;

   va2 = abs ( va1 );
   cout << "The absolute value of the initial valarray is: ";
      for (i = 0 ; i < 9 ; i++ )
         cout << va2 [ i ] << " ";
   cout << "." << endl;
}
```

Output

```
The initial valarray is: 0 -1 -2 -3 4 5 6 7 8 .
The absolute value of the initial valarray is: 0 1 2 3 4 5 6 7 8 .
```

Requirements

Header: <valarray>

See Also

Reference

abs (STL Samples)

Other Resources

<valarray> Members

acos

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the arccosine of the elements of the input valarray.

```
template<class Type>
   valarray<Type> acos(
      const valarray<Type>& _Left
   );
```

Parameters

_Left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the arccosine of the elements of the input valarray.

Remarks

The units of the returned elements are in radians.

The return value is a principal value between 0 and +pi that is consistent with the cosine value input.

Example

```
// valarray_acos.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1 ( 9 );
   for ( i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  0.25 * i - 1;
   valarray<double> va2 ( 9 );

   cout << "The initial valarray is:";
   for (i = 0 ; i < 9 ; i++ )
      cout << " " << va1 [ i ];
   cout << endl;

   va2 = acos ( va1 );
   cout << "The arccosine of the initial valarray is:\n";
   for (i = 0 ; i < 9 ; i++ )
      cout << setw(10) << va2 [ i ]
         << "  radians, which is  "
         << setw(11) << (180/pi) * va2 [ i ]
         << "  degrees" << endl;
}
```

Output

```
The initial valarray is: -1 -0.75 -0.5 -0.25 0 0.25 0.5 0.75 1
The arccosine of the initial valarray is:
   3.14159  radians, which is          180  degrees
   2.41886  radians, which is       138.59  degrees
    2.0944  radians, which is          120  degrees
   1.82348  radians, which is      104.478  degrees
```

```
    1.5708  radians, which is           90  degrees
   1.31812  radians, which is      75.5225  degrees
    1.0472  radians, which is           60  degrees
  0.722734  radians, which is      41.4096  degrees
         0  radians, which is            0  degrees
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

asin

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the arcsine of the elements of the input valarray.

```
template<class Type>
   valarray<Type> asin(
      const valarray<Type>& _Left
   );
```

Parameters

_Left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the arcsine of the elements of the input valarray.

Remarks

The units of the returned elements are in radians.

The return value is a principal value between +pi/2 and –pi/2 that is consistent with the sine value input.

Example

```
// valarray_asin.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1 ( 9 );
   for ( i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  0.25 * i - 1;
   valarray<double> va2 ( 9 );

   cout << "The initial valarray is:";
   for (i = 0 ; i < 9 ; i++ )
      cout << " " << va1 [ i ];
   cout << endl;

   va2 = asin ( va1 );
   cout << "The arcsine of the initial valarray is:\n";
   for (i = 0 ; i < 9 ; i++ )
      cout << setw(10) << va2 [ i ]
           << "  radians, which is  "
           << setw(11) << (180/pi) * va2 [ i ]
           << "  degrees" << endl;
}
```

Output

```
The initial valarray is: -1 -0.75 -0.5 -0.25 0 0.25 0.5 0.75 1
The arcsine of the initial valarray is:
   -1.5708  radians, which is          -90  degrees
 -0.848062  radians, which is     -48.5904  degrees
 -0.523599  radians, which is          -30  degrees
  -0.25268  radians, which is     -14.4775  degrees
```

```
        0  radians, which is            0  degrees
  0.25268  radians, which is      14.4775  degrees
 0.523599  radians, which is           30  degrees
 0.848062  radians, which is      48.5904  degrees
   1.5708  radians, which is           90  degrees
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

atan

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the principal value of the arctangent of the elements of the input valarray.

```
template<class Type>
   valarray<Type> atan(
      const valarray<Type>& _Left
   );
```

Parameters

_Left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the arctangent of the elements of the input valarray.

Remarks

The units of the returned elements are in radians.

The return value is a principal value between +pi/2 and –pi/2 that is consistent with the tangent value input.

Example

```
// valarray_atan.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1 ( 9 );
   va1 [ 0 ] = -100;
   for ( i = 1 ; i < 8 ; i++ )
      va1 [ i ] =  5 * ( 0.25 * i - 1 );
   va1 [ 8 ] = 100;
   valarray<double> va2 ( 9 );

   cout << "The initial valarray is: ";
   for ( i = 0 ; i < 9 ; i++ )
      cout << va1 [ i ] << " ";
   cout << "." << endl;

   va2 = atan ( va1 );
   cout << "The arcsine of the initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << setw(10) << va2 [ i ]
           << "  radians, which is  "
           << setw(11) << (180/pi) * va2 [ i ]
           << "  degrees" << endl;
   cout << endl;
}
```

Output

```
The initial valarray is: -100 -3.75 -2.5 -1.25 0 1.25 2.5 3.75 100 .
The arcsine of the initial valarray is:
```

```
   -1.5608  radians, which is     -89.4271  degrees
  -1.31019  radians, which is     -75.0686  degrees
  -1.19029  radians, which is     -68.1986  degrees
 -0.896055  radians, which is     -51.3402  degrees
         0  radians, which is            0  degrees
  0.896055  radians, which is      51.3402  degrees
   1.19029  radians, which is      68.1986  degrees
   1.31019  radians, which is      75.0686  degrees
    1.5608  radians, which is      89.4271  degrees
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

atan2

Returns a valarray whose elements are equal to the arctangent of the Cartesian components specified by a combination of constants and elements of valarrays.

```
template<class Type>
   valarray<Type> atan2(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<Type> atan2(
      const valarray<Type> _Left,
      const Type& _Right
   );
template<class Type>
   valarray<Type> atan2(
      const Type& _Left,
      const valarray<Type>& _Right
   );
```

Parameters

_Left

The constant numerical data type or input valarray whose elements provide the values for the y-coordinate of the arctangent argument.

_Right

The constant numerical data type or input valarray whose elements provide the values for the x-coordinate of the arctangent argument.

Return Value

A valarray whose elements *I* are equal to the arctangent of:

- *_Left* [*I*] / *_Right* [*I*] for the first template function.
- *_Left* [*I*] / *_Right* for the second template function.
- *_Left* / *_Right* [*I*] for the third template function.

Remarks

The units of the returned elements are in radians.

This function preserves information about the signs of the components in the argument that is lost by the standard tangent function, and this knowledge of the quadrant enables the return value to be assigned a unique angle between +pi and –pi.

If *_Left* and *_Right* have a different number of elements, the result is undefined.

Example

```
// valarray_atan2.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1y ( 1 , 4 ), va1x ( 1 , 4 );
```

```
    va1x [ 1 ] = -1;
    va1x [ 2 ] = -1;
    va1y [ 2 ] = -1;
    va1y [ 3 ] = -1;
    valarray<double> va2 ( 4 );

    cout << "The initial valarray for the x coordinate is: ( ";
    for ( i = 0 ; i < 4 ; i++ )
       cout << va1x [ i ] << " ";
    cout << "))." << endl;

    cout << "The initial valarray for the y coordinate is: ( ";
    for ( i = 0 ; i < 4 ; i++ )
       cout << va1y [ i ] << " ";
    cout << "))." << endl;

    va2 = atan2 ( va1y , va1x );
    cout << "The atan2 ( y / x ) of the initial valarrays is:\n";
    for ( i = 0 ; i < 4 ; i++ )
       cout << setw( 10 ) << va2 [ i ]
            << "  radians, which is  "
            << setw( 11 ) << ( 180/pi ) * va2 [ i ]
            << "degrees" << endl;
    cout << endl;
  }
```

Output

```
The initial valarray for the x coordinate is: ( 1 -1 -1 1 ).
The initial valarray for the y coordinate is: ( 1 1 -1 -1 ).
The atan2 ( y / x ) of the initial valarrays is:
  0.785398  radians, which is           45degrees
   2.35619  radians, which is          135degrees
  -2.35619  radians, which is         -135degrees
 -0.785398  radians, which is          -45degrees
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

cos (<valarray>)

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the cosine of the elements of the input valarray.

```
template<class Type>
   valarray<Type> cos(
      const valarray<Type>& _Left
   );
```

Parameters

_Left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the absolute value of the elements of the input valarray.

Example

```
// valarray_cos.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1 ( 9 );
   for ( i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  ( pi ) * ( 0.25 * i - 1 );
   valarray<double> va2 ( 9 );

   cout << "The initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << setw( 10 ) << va1 [ i ]
      << "  radians, which is  "
      << setw( 5 ) << ( 180/pi ) * va1 [ i ]
      << "  degrees" << endl;
   cout << endl;

   va2 = cos ( va1 );
   cout << "The cosine of the initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << va2 [ i ] << endl;
}
```

Output

```
The initial valarray is:
  -3.14159  radians, which is   -180  degrees
  -2.35619  radians, which is   -135  degrees
   -1.5708  radians, which is    -90  degrees
 -0.785398  radians, which is    -45  degrees
         0  radians, which is      0  degrees
  0.785398  radians, which is     45  degrees
    1.5708  radians, which is     90  degrees
   2.35619  radians, which is    135  degrees
   3.14159  radians, which is    180  degrees
```

```
The cosine of the initial valarray is:
-1
-0.707107
-1.03412e-013
0.707107
1
0.707107
-1.03412e-013
-0.707107
-1
```

Requirements

Header: <valarray>

See Also

Reference

Trigonometry Functions

Other Resources

<valarray> Members

cosh (<valarray>)

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the hyperbolic cosine of the elements of the input valarray.

```
template<class Type>
   valarray<Type> cosh(
      const valarray<Type>& _Left
   );
```

Parameters

_Left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the hyperbolic cosine of the elements of the input valarray.

Remarks

Identities defining the hyperbolic cosine in terms of exponential function:

cosh (*z*) = (exp (*z*) + exp (-*z*)) / 2

Example

```
// valarray_cosh.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1 ( 9 );
   for ( i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  pi * ( 0.25 * i - 1 );
   valarray<double> va2 ( 9 );

   cout << "The initial valarray is:\n";
   for (i = 0 ; i < 9 ; i++ )
      cout << setw( 10 ) << va1 [ i ]
      << "  radians, which is  "
      << setw( 5 ) << ( 180/pi ) * va1 [ i ]
      << "  degrees" << endl;
   cout << endl;

   va2 = cosh ( va1 );
   cout << "The hyperbolic cosine of the initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << va2 [ i ] << endl;
}
```

Output

```
The initial valarray is:
  -3.14159  radians, which is   -180  degrees
  -2.35619  radians, which is   -135  degrees
   -1.5708  radians, which is    -90  degrees
 -0.785398  radians, which is    -45  degrees
         0  radians, which is      0  degrees
```

```
  0.785398  radians, which is     45  degrees
    1.5708  radians, which is     90  degrees
   2.35619  radians, which is    135  degrees
   3.14159  radians, which is    180  degrees

The hyperbolic cosine of the initial valarray is:
11.592
5.32275
2.50918
1.32461
1
1.32461
2.50918
5.32275
11.592
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

exp (<valarray>)

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the natural exponential of the elements of the input valarray.

```
template<class Type>
   valarray<Type> exp(
      const valarray<Type>& _Left
   );
```

Parameters

_Left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the natural exponential of the elements of the input valarray.

Example

```
// valarray_exp.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   int i;

   valarray<double> va1 ( 9 );
   for ( i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  10 * ( 0.25 * i - 1 );
   valarray<double> va2 ( 9 );

   cout << "Initial valarray:";
   for ( i = 0 ; i < 9 ; i++ )
      cout << " " << va1 [ i ];
   cout << endl;

   va2 = exp ( va1 );
   cout << "The natural exponential of the initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << va2 [ i ] << endl;
}
```

Output

```
Initial valarray: -10 -7.5 -5 -2.5 0 2.5 5 7.5 10
The natural exponential of the initial valarray is:
4.53999e-005
0.000553084
0.00673795
0.082085
1
12.1825
148.413
1808.04
22026.5
```

Requirements

Header: <valarray>

See Also

Reference

exp, log, and log10

Other Resources

<valarray> Members

log (<valarray>)

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the natural logarithm of the elements of the input valarray.

```
template<class Type>
   valarray<Type> log(
      const valarray<Type>& _Left
   );
```

Parameters

_Left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the absolute value of the elements of the input valarray.

Example

```
// valarray_log.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   int i;

   valarray<double> va1 ( 9 );
   for (i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  10 * i;
   valarray<double> va2 ( 9 );

   cout << "Initial valarray:";
   for ( i = 0 ; i < 9 ; i++ )
      cout << " " << va1 [ i ];
   cout << endl;

   va2 = log ( va1 );
   cout << "The natural logarithm of the initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << va2 [ i ] << endl;
}
```

Output

```
Initial valarray: 0 10 20 30 40 50 60 70 80
The natural logarithm of the initial valarray is:
-1.#INF
2.30259
2.99573
3.4012
3.68888
3.91202
4.09434
4.2485
4.38203
```

Requirements

Header: <valarray>

See Also

Reference

exp, log, and log10

Other Resources

<valarray> Members

log10 (<valarray>)

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the base 10 or common logarithm of the elements of the input valarray.

```
template<class Type>
   valarray<Type> log10(
      const valarray<Type>& _Left
   );
```

Parameters

_Left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the common logarithm of the elements of the input valarray.

Example

```
// valarray_log10.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   int i;

   valarray<double> va1 ( 11 );
   for ( i = 0 ; i < 11 ; i++ )
      va1 [ i ] =  10 * i;
   valarray<double> va2 ( 9 );

   cout << "Initial valarray:";
   for (i = 0 ; i < 11 ; i++ )
      cout << " " << va1 [ i ];
   cout << endl;

   va2 = log10 ( va1 );
   cout << "The common logarithm of the initial valarray is:\n";
   for (i = 0 ; i < 11 ; i++ )
      cout << va2 [ i ] << endl;
}
```

Output

```
Initial valarray: 0 10 20 30 40 50 60 70 80 90 100
The common logarithm of the initial valarray is:
-1.#INF
1
1.30103
1.47712
1.60206
1.69897
1.77815
1.8451
1.90309
1.95424
2
```

Requirements

Header: <valarray>

See Also

Reference

exp, log, and log10

Other Resources

<valarray> Members

pow (<valarray>)

Operates on the elements of input valarrays and constants, returning a valarray whose elements are equal to a base specified either by the elements of an input valarray or a constant raised to an exponent specified either by the elements of an input valarray or a constant.

```
template<class Type>
   valarray<Type> pow(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<Type> pow(
      const valarray<Type> _Left,
      const Type& _Right
   );
template<class Type>
   valarray<Type> pow(
      const Type& _Left,
      const valarray<Type>& _Right
   );
```

Parameters

_Left

The input valarray whose elements supply the base for each element to be exponentiated.

_Right

The input valarray whose elements supply the power for each element to be exponentiated.

Return Value

A valarray whose elements *I* are equal to:

- *_Left* [*I*] raised to the power *_Right* [*I*] for the first template function.
- *_Left* [*I*] raised to the power *_Right* for the second template function.
- *_Left* raised to the power *_Right* [*I*] for the third template function.

Remarks

If *_Left* and *_Right* have a different number of elements, the result is undefined.

Example

```
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> vabase ( 6 );
   for ( i = 0 ; i < 6 ; i++ )
      vabase [ i ] =  i/2;
   valarray<double> vaexp ( 6 );
   for ( i = 0 ; i < 6 ; i++ )
      vaexp [ i ] =  2 * i;

   valarray<double> va2 ( 6 );
```

```
    cout << "The initial valarray for the base is: ( ";
       for ( i = 0 ; i < 6 ; i++ )
          cout << vabase [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial valarray for the exponent is: ( ";
       for ( i = 0 ; i < 6 ; i++ )
          cout << vaexp[ i ] << " ";
    cout << ")." << endl;

    va2 = pow ( vabase , vaexp );
    cout << "The power of (n/2) * exp (2n) for n = 0 to n = 5 is: \n";
       for ( i = 0 ; i < 6 ; i++ )
          cout <<  "n = " << i << "\tgives " << va2 [ i ] << endl;
 }
```

Output

```
The initial valarray for the base is: ( 0 0 1 1 2 2 ).
The initial valarray for the exponent is: ( 0 2 4 6 8 10 ).
The power of (n/2) * exp (2n) for n = 0 to n = 5 is:
n = 0   gives 1
n = 1   gives 0
n = 2   gives 1
n = 3   gives 1
n = 4   gives 256
n = 5   gives 1024
```

Requirements

Header: <valarray>

See Also

Reference

sqrt and pow

Other Resources

<valarray> Members

sin (<valarray>)

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the sine of the elements of the input valarray.

```
template<class Type>
   valarray<Type> sin(
      const valarray<Type>& _Left
   );
```

Parameters

_Left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the sine of the elements of the input valarray.

Example

```
// valarray_sin.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1 ( 9 );
   for ( i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  pi * ( 0.25 * i - 1 );
   valarray<double> va2 ( 9 );

   cout << "The initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << setw(10) << va1 [ i ]
      << "   radians, which is   "
      << setw(5) << ( 180/pi ) * va1 [ i ]
      << "   degrees" << endl;
   cout << endl;

   va2 = sin ( va1 );
   cout << "The sine of the initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << va2 [ i ] << endl;
}
```

Output

```
The initial valarray is:
  -3.14159   radians, which is    -180   degrees
  -2.35619   radians, which is    -135   degrees
   -1.5708   radians, which is     -90   degrees
 -0.785398   radians, which is     -45   degrees
         0   radians, which is       0   degrees
  0.785398   radians, which is      45   degrees
    1.5708   radians, which is      90   degrees
   2.35619   radians, which is     135   degrees
   3.14159   radians, which is     180   degrees
```

```
The sine of the initial valarray is:
2.06823e-013
-0.707107
-1
-0.707107
0
0.707107
1
0.707107
-2.06823e-013
```

Requirements

Header: <valarray>

See Also

Reference

Trigonometry Functions

Other Resources

<valarray> Members

sinh (<valarray>)

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the hyperbolic sine of the elements of the input valarray.

```
template<class Type>
   valarray<Type> sinh(
      const valarray<Type>& _Left
   );
```

Parameters

_Left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the hyperbolic sine of the elements of the input valarray.

Remarks

Identities defining the hyperbolic sine in terms of exponential function:

sinh (*z*) = (exp (*z*) – exp (-*z*)) / 2

Example

```
// valarray_sinh.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1 ( 9 );
   for (i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  pi * ( 0.25 * i - 1 );
   valarray<double> va2 ( 9 );

   cout << "The initial valarray is:\n";
   for (i = 0 ; i < 9 ; i++ )
      cout << setw( 10 ) << va1 [ i ]
      << "   radians, which is   "
      << setw( 5 ) << ( 180/pi ) * va1 [ i ]
      << "  degrees" << endl;
   cout << endl;

   va2 = sinh ( va1 );
   cout << "The hyperbolic sine of the initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << va2 [ i ] << endl;
}
```

Output

```
The initial valarray is:
  -3.14159   radians, which is    -180  degrees
  -2.35619   radians, which is    -135  degrees
   -1.5708   radians, which is     -90  degrees
 -0.785398   radians, which is     -45  degrees
         0   radians, which is       0  degrees
```

```
  0.785398   radians, which is      45  degrees
    1.5708   radians, which is      90  degrees
   2.35619   radians, which is     135  degrees
   3.14159   radians, which is     180  degrees

The hyperbolic sine of the initial valarray is:
-11.5487
-5.22797
-2.3013
-0.868671
0
0.868671
2.3013
5.22797
11.5487
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

sqrt (<valarray>)

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the square root of the elements of the input valarray.

```
template<class Type>
   valarray<Type> sqrt(
      const valarray<Type>& _Left
   );
```

Parameters

_Left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the square root of the elements of the input valarray.

Example

```
// valarray_sqrt.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <cmath>

int main( )
{
   using namespace std;
   int i;

   valarray<double> va1 ( 6 );
   for ( i = 0 ; i < 5 ; i++ )
      va1 [ i ] = i * i;

   cout << "The initial valarray is: ( ";
      for ( i = 0 ; i < 5 ; i++ )
         cout << va1 [ i ] << " ";
   cout << ")." << endl;

   valarray<double> va2 = sqrt ( va1 );
   cout << "The square root of the initial valarray is: ( ";
      for ( i = 0 ; i < 5 ; i++ )
         cout << va2 [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial valarray is: ( 0 1 4 9 16 ).
The square root of the initial valarray is: ( 0 1 2 3 4 ).
```

Requirements

Header: <valarray>

See Also

Reference

sqrt and pow

Other Resources

<valarray> Members

tan (<valarray>)

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the tangent of the elements of the input valarray.

```
template<class Type>
   valarray<Type> tan(
      const valarray<Type>& _Left
   );
```

Parameters

_Left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the tangent of the elements of the input valarray.

Example

```
// valarray_tan.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1 ( 9 );
   for ( i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  ( pi/2 ) * ( 0.25 * i - 1 );
   valarray<double> va2 ( 9 );

   cout << "The initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << setw( 10 ) << va1 [ i ]
      << "     radians, which is   "
      << setw( 5 ) << ( 180/pi ) * va1 [ i ]
      << "   degrees" << endl;
   cout << endl;

   va2 = tan ( va1 );
   cout << "The tangent of the initial valarray is:\n ";
   for ( i = 0 ; i < 9 ; i++ )
      cout << va2 [ i ] << endl;
}
```

Output

```
The initial valarray is:
   -1.5708     radians, which is     -90   degrees
   -1.1781     radians, which is   -67.5   degrees
 -0.785398     radians, which is     -45   degrees
 -0.392699     radians, which is   -22.5   degrees
         0     radians, which is       0   degrees
  0.392699     radians, which is    22.5   degrees
  0.785398     radians, which is      45   degrees
    1.1781     radians, which is    67.5   degrees
    1.5708     radians, which is      90   degrees
```

```
The tangent of the initial valarray is:
 9.6701e+012
-2.41421
-1
-0.414214
0
0.414214
1
2.41421
-9.6701e+012
```

Requirements

Header: <valarray>

See Also

Reference

Trigonometry Functions

Other Resources

<valarray> Members

tanh (<valarray>)

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the hyperbolic tangent of the elements of the input valarray.

```
template<class Type>
   valarray<Type> tanh(
      const valarray<Type>& _Left
   );
```

Parameters

_Left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the hyperbolic cosine of the elements of the input valarray.

Remarks

Identities defining the hyperbolic tangent in terms of the exponential function:

tanh (*z*) = sinh (*z*) / cosh (*z*) = (exp (*z*) – exp (-*z*)) / (exp (*z*) + exp (-*z*))

Example

```
// valarray_tanh.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1 ( 9 );
   for ( i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  pi * ( 0.25 * i - 1 );
   valarray<double> va2 ( 9 );

   cout << "The initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << setw( 10 ) << va1 [ i ]
      << "   radians, which is     "
                << setw( 5 ) << ( 180/pi ) * va1 [ i ]
                << "  degrees" << endl;
   cout << endl;

   va2 = tanh ( va1 );
   cout << "The hyperbolic tangent of the initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
     cout << va2 [ i ] << endl;
}
```

Output

```
The initial valarray is:
  -3.14159   radians, which is      -180  degrees
  -2.35619   radians, which is      -135  degrees
   -1.5708   radians, which is       -90  degrees
 -0.785398   radians, which is       -45  degrees
         0   radians, which is         0  degrees
```

```
  0.785398   radians, which is        45  degrees
    1.5708   radians, which is        90  degrees
   2.35619   radians, which is       135  degrees
   3.14159   radians, which is       180  degrees

The hyperbolic tangent of the initial valarray is:
-0.996272
-0.982193
-0.917152
-0.655794
0
0.655794
0.917152
0.982193
0.996272
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

<valarray> Operators

For information about the operators in <valarray>, see <valarray> Members.

operator!= (<valarray>)

Tests whether the corresponding elements of two equally sized valarrays are unequal or whether all the elements of a valarray are unequal a specified value.

```
template<class Type>
   valarray<bool> operator!=(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<bool> operator!=(
      const valarray<Type> _Left,
      const Type& _Right
   );
template<class Type>
   valarray<bool> operator!=(
      const Type& _Left,
      const valarray<Type>& _Right
   );
```

Parameters

_Left

The first of the two valarrays whose elements are to be tested for inequality.

_Right

The second of the two valarrays whose elements are to be tested for inequality.

Return Value

A valarray of Boolean values, each of which is:

- **true** if the corresponding elements are unequal.
- **false** if the corresponding elements are not unequal.

Remarks

The first template operator returns an object of class valarray<bool>, each of whose elements *I* is *_Left*[*I*] != *_Right*[*I*].

The second template operator stores in element *I* *_Left*[*I*] != *_Right*.

The third template operator stores in element *I* *_Left* != *_Right*[*I*].

Example

```
// valarray_op_ne.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<bool> vaNE ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i;
```

```
    cout << "The initial Left valarray is: ( ";
       for ( i = 0 ; i < 10 ; i++ )
          cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
       for ( i = 0 ; i < 10 ; i++ )
          cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaNE = ( vaL != vaR );
    cout << "The element-by-element result of "
         << "the not equal comparison test is the\n valarray: ( ";
       for ( i = 0 ; i < 10 ; i++ )
          cout << vaNE [ i ] << " ";
    cout << ")." << endl;
  }
```

Output

```
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The element-by-element result of the not equal comparison test is the
 valarray: ( 0 0 1 0 1 0 1 0 1 0 ).
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

operator%

Obtains the remainder of dividing the corresponding elements of two equally sized valarrays or of dividing a valarray by a specified value or of dividing a specified value by a valarray.

```
template<class Type>
   valarray<bool> operator%(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<bool> operator%(
      const valarray<Type> _Left,
      const Type& _Right
   );
template<class Type>
   valarray<bool> operator%(
      const Type& _Left,
      const valarray<Type>& _Right
   );
```

Parameters

_Left

A value or valarray that serves as the dividend into which another value or valarray is to be divided.

_Right

A value or valarray that serves as the divisor and that divides another value or valarray.

Return Value

A valarray whose elements are the element-wise remainders of *_Left* divided by *_Right*.

Example

```
// valarray_op_rem.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 6 ), vaR ( 6 );
   valarray<int> vaREM ( 6 );
   for ( i = 0 ; i < 6 ; i += 2 )
      vaL [ i ] =  53;
   for ( i = 1 ; i < 6 ; i += 2 )
      vaL [ i ] =  -67;
   for ( i = 0 ; i < 6 ; i++ )
      vaR [ i ] =  3*i+1;

   cout << "The initial Left valarray is: ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;
```

```
    vaREM = ( vaL % vaR );
    cout << "The remainders from the element-by-element "
         << "division is the\n valarray: ( ";
       for ( i = 0 ; i < 6 ; i++ )
          cout << vaREM [ i ] << " ";
    cout << "))." << endl;
  }
```

Output

```
The initial Left valarray is: ( 53 -67 53 -67 53 -67 ).
The initial Right valarray is: ( 1 4 7 10 13 16 ).
The remainders from the element-by-element division is the
 valarray: ( 0 -3 4 -7 1 -3 ).
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

operator&

Obtains the bitwise **AND** between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the element type.

```
template<class Type>
   valarray<bool> operator&(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<bool> operator&(
      const valarray<Type> _Left,
      const Type& _Right
   );
template<class Type>
   valarray<bool> operator&(
      const Type& _Left,
      const valarray<Type>& _Right
   );
```

Parameters

_Left

The first of the two valarrays whose respective elements are to be combined with the bitwise **AND** or a specified value of the element type to be combined bitwise with each element of a valarray.

_Right

The second of the two valarrays whose respective elements are to be combined with the bitwise **AND** or a specified value of the element type to be combined bitwise with each element of a valarray.

Return Value

A valarray whose elements are the element-wise combination of the bitwise AND operation of *_Left* and *_Right*.

Remarks

A bitwise operation can only be used to manipulate bits in **char** and **int** data types and variants and not on **float**, **double**, **long double**, **void bool** or other, more complex data types.

The bitwise **AND** has the same truth table as the logical **AND** but applies to the data type on the level of the individual bits. The operator&& applies on an element level, counting all nonzero values as true, and the result is a valarray of Boolean values. The bitwise **AND operator&**, by contrast, can result in a valarray of values other than 0 or 1, depending on outcome of the bitwise operation.

Example

```
// valarray_op_bitand.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<int> vaBWA ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i+1;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i;
```

```
    cout << "The initial Left valarray is:  ( ";
       for ( i = 0 ; i < 10 ; i++ )
          cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
       for ( i = 0 ; i < 10 ; i++ )
          cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaBWA = ( vaL & vaR );
    cout << "The element-by-element result of "
         << "the bitwise operator & is the\n valarray: ( ";
       for ( i = 0 ; i < 10 ; i++ )
          cout << vaBWA [ i ] << " ";
    cout << ")." << endl;
  }
```

Output

```
The initial Left valarray is:  ( 0 2 0 4 0 6 0 8 0 10 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The element-by-element result of the bitwise operator & is the
 valarray: ( 0 0 0 0 0 4 0 0 0 8 ).
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

operator&&

Obtains the logical **AND** between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the valarray's element type.

```
template<class Type>
   valarray<bool> operator&&(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<bool> operator&&(
      const valarray<Type> _Left,
      const Type& _Right
   );
template<class Type>
   valarray<bool> operator&&(
      const Type& _Left,
      const valarray<Type>& _Right
   );
```

Parameters

_Left

The first of the two valarrays whose respective elements are to be combined with the logical **AND** or a specified value of the element type to be combined with each element of a valarray.

_Right

The second of the two valarrays whose respective elements are to be combined with the logical **AND** or a specified value of the element type to be combined with each element of a valarray.

Return Value

A valarray whose elements are of type bool and are the element-wise combination of the logical **AND** operation of *_Left* and *_Right*.

Remarks

The logical **AND operator&&** applies on an element level, counting all nonzero values as true, and the result is a valarray of Boolean values. The bitwise version of **AND**, operator&,, by contrast, can result in a valarray of values other than 0 or 1, depending on the outcome of the bitwise operation.

Example

```
// valarray_op_logand.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<bool> vaLAA ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i-1;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial Left valarray is:  ( ";
```

```
      for (i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;

   cout << "The initial Right valarray is: ( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << "))." << endl;

   vaLAA = ( vaL && vaR );
   cout << "The element-by-element result of "
        << "the logical AND operator&& is the\n valarray: ( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << vaLAA [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial Left valarray is:  ( 0 0 0 2 0 4 0 6 0 8 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The element-by-element result of the logical AND operator&& is the
 valarray: ( 0 0 0 1 0 1 0 1 0 1 ).
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

operator> (<valarray>)

Tests whether the elements of one valarray are greater than the elements of an equally sized valarray or whether all the elements of a valarray are greater or less than a specified value.

```
template<class Type>
   valarray<bool> operator>(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<bool> operator>(
      const valarray<Type> _Left,
      const Type& _Right
   );
template<class Type>
   valarray<bool> operator>(
      const Type& _Left,
      const valarray<Type>& _Right
   );
```

Parameters

_Left

The first of the two valarrays whose elements are to be compared or a specified value to be compared with each element of a valarray.

_Right

The second of the two valarrays whose elements are to be compared or a specified value to be compared with each element of a valarray.

Return Value

A valarray of Boolean values, each of which is:

- **true** if the *_Left* element or value is greater than the corresponding *_Right* element or value.
- **false** if the *_Left* element or value is not greater than the corresponding *_Right* element or value.

Remarks

If the number of elements in two valarrays is not equal, the result is undefined.

Example

```
// valarray_op_gt.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<bool> vaNE ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i - 1;

   cout << "The initial Left valarray is: ( ";
```

```
      for (i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << "))." << endl;

   vaNE = ( vaL > vaR );
   cout << "The element-by-element result of "
        << "the greater than comparison test is the\n valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaNE [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( -1 0 1 2 3 4 5 6 7 8 ).
The element-by-element result of the greater than comparison test is the
 valarray: ( 1 1 0 1 0 1 0 1 0 1 ).
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

operator>> (<valarray>)

Right-shifts the bits for each element of a valarray a specified number of positions or by an element-wise amount specified by a second valarray.

```
template<class Type>
   valarray<bool> operator>>(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<bool> operator>>(
      const valarray<Type> _Left,
      const Type& _Right
  );
template<class Type>
   valarray<bool> operator>>(
      const Type& _Left,
      const valarray<Type>& _Right
   );
```

Parameters

_Left

The value to be shifted or the valarray whose elements are to be shifted.

_Right

The value indicating the amount of right shift or valarray whose elements indicate the element-wise amount of right shift.

Return Value

A valarray whose elements have been shifted right by the specified amount.

Remarks

Signed numbers have their signs preserved.

Example

```
// valarray_op_rs.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
   valarray<int> vaNE ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  64;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  -64;
   for ( i = 0 ; i < 8 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial Left valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
```

```
          cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaNE = ( vaL >> vaR );
    cout << "The element-by-element result of "
         << "the right shift is the\n valarray: ( ";
       for ( i = 0 ; i < 8 ; i++ )
          cout << vaNE [ i ] << " ";
    cout << ")." << endl;
  }
```

Output

```
The initial Left valarray is: ( 64 -64 64 -64 64 -64 64 -64 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the right shift is the
 valarray: ( 64 -32 16 -8 4 -2 1 -1 ).
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

operator>= (<valarray>)

Tests whether the elements of one valarray are greater than or equal to the elements of an equally sized valarray or whether all the elements of a valarray are greater than or equal to or less than or equal to a specified value.

```
template<class Type>
   valarray<bool> operator>=(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<bool> operator>=(
      const valarray<Type> _Left,
      const Type& _Right
   );
template<class Type>
   valarray<bool> operator>=(
      const Type& _Left,
      const valarray<Type>& _Right
   );
```

Parameters

_Left

The first of the two valarrays whose elements are to be compared or a specified value to be compared with each element of a valarray.

_Right

The second of the two valarrays whose elements are to be compared or a specified value to be compared with each element of a valarray.

Return Value

A valarray of Boolean values, each of which is:

- **true** if the *_Left* element or value is greater than or equal to the corresponding *_Right* element or value.
- **false** if the *_Left* element or value is less than the corresponding *_Right* element or value.

Remarks

If the number of elements in two valarrays is not equal, the result is undefined.

Example

```
// valarray_op_ge.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<bool> vaNE ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i - 1;

   cout << "The initial Left valarray is: ( ";
```

```
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << "))." << endl;

   vaNE = ( vaL >= vaR );
   cout << "The element-by-element result of "
        << "the greater than or equal test is the\n valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaNE [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( -1 0 1 2 3 4 5 6 7 8 ).
The element-by-element result of the greater than or equal test is the
 valarray: ( 1 1 0 1 0 1 0 1 0 1 ).
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

operator< (<valarray>)

Tests whether the elements of one valarray are less than the elements of an equally sized valarray or whether all the elements of a valarray are greater or less than a specified value.

```
template<class Type>
   valarray<bool> operator<(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<bool> operator<(
      const valarray<Type> _Left,
      const Type& _Right
   );
template<class Type>
   valarray<bool> operator<(
      const Type& _Left,
      const valarray<Type>& _Right
   );
```

Parameters

_Left

The first of the two valarrays whose elements are to be compared or a specified value to be compared with each element of a valarray.

_Right

The second of the two valarrays whose elements are to be compared or a specified value to be compared with each element of a valarray.

Return Value

A valarray of Boolean values, each of which is:

- **true** if the *_Left* element or value is less than the corresponding *_Right* element or value.
- **false** if the *_Left* element or value is not less than the corresponding *_Right* element or value.

Remarks

If the number of elements two valarrays is not equal, the result is undefined.

Example

```
// valarray_op_lt.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<bool> vaNE ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial Left valarray is: ( ";
```

```
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << "))." << endl;

   vaNE = ( vaL < vaR );
   cout << "The element-by-element result of "
        << "the less-than comparson test is the\n valarray: ( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << vaNE [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The element-by-element result of the less-than comparson test is the
 valarray: ( 0 0 1 0 1 0 1 0 1 0 ).
```

Requirements

Header: <valarray>

See Also

Reference

stack::operator<

Other Resources

<valarray> Members

operator<< (<valarray>)

Left shifts the bits for each element of a valarray a specified number of positions or by an element-wise amount specified by a second valarray.

```
template<class Type>
   valarray<bool> operator<<(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<bool> operator<<(
      const valarray<Type> _Left,
      const Type& _Right
   );
template<class Type>
   valarray<bool> operator<<(
      const Type& _Left,
      const valarray<Type>& _Right
   );
```

Parameters

_Left

The value to be shifted or the valarray whose elements are to be shifted.

_Right

The value indicating the amount of left shift or valarray whose elements indicate the element-wise amount of left shift.

Return Value

A valarray whose elements have been shifted left by the specified amount.

Remarks

Signed numbers have their signs preserved.

Example

```
// valarray_op_ls.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
   valarray<int> vaNE ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  1;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  -1;
   for ( i = 0 ; i < 8 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial Left valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
```

```
          cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaNE = ( vaL << vaR );
    cout << "The element-by-element result of "
         << "the left shift is the\n valarray: ( ";
       for ( i = 0 ; i < 8 ; i++ )
          cout << vaNE [ i ] << " ";
    cout << ")." << endl;
  }
```

Output

```
The initial Left valarray is: ( 1 -1 1 -1 1 -1 1 -1 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the left shift is the
 valarray: ( 1 -2 4 -8 16 -32 64 -128 ).
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

operator<= (<valarray>)

Tests whether the elements of one valarray are less than or equal to the elements of an equally sized valarray or whether all the elements of a valarray are greater than or equal to or less than or equal to a specified value.

```
template<class Type>
   valarray<bool> operator<=(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<bool> operator<=(
      const valarray<Type> _Left,
      const Type& _Right
   );
template<class Type>
   valarray<bool> operator<=(
      const Type& _Left,
      const valarray<Type>& _Right
   );
```

Parameters

_Left

The first of the two valarrays whose elements are to be compared or a specified value to be compared with each element of a valarray.

_Right

The second of the two valarrays whose elements are to be compared or a specified value to be compared with each element of a valarray.

Return Value

A valarray of Boolean values, each of which is:

- **true** if the *_Left* element or value is less than or equal to the corresponding *_Right* element or value.
- **false** if the *_Left* element or value is greater than the corresponding *_Right* element or value.

Remarks

If the number of elements two valarrays is not equal, the result is undefined.

Example

```
// valarray_op_le.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<bool> vaNE ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i - 1;

   cout << "The initial Left valarray is: ( ";
```

```
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << "))." << endl;

   vaNE = ( vaL <= vaR );
   cout << "The element-by-element result of "
        << "the less than or equal test is the\n valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaNE [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( -1 0 1 2 3 4 5 6 7 8 ).
The element-by-element result of the less than or equal test is the
 valarray: ( 0 0 1 0 1 0 1 0 1 0 ).
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

operator* (<valarray>)

Obtains the element-wise product between corresponding elements of two equally sized valarrays or of between a valarray a specified value.

```
template<class Type>
   valarray<bool> operator*(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<bool> operator*(
      const valarray<Type> _Left,
      const Type& _Right
   );
template<class Type>
   valarray<bool> operator*(
      const Type& _Left,
      const valarray<Type>& _Right
   );
```

Parameters

_Left

The first of the two valarrays whose elements are to be multiplied or a specified value to be multiplied with each element of a valarray.

_Right

The second of the two valarrays whose elements are to be multiplied or a specified value to be multiplied with each element of a valarray.

Return Value

A valarray whose elements are the element-wise product of *_Left* and *_Right*.

Example

```
// valarray_op_eprod.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
   valarray<int> vaNE ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  2;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  -1;
   for ( i = 0 ; i < 8 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial Left valarray is: ( ";
      for (i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for (i = 0 ; i < 8 ; i++ )
         cout << vaR [ i ] << " ";
```

```
    cout << "))." << endl;

    vaNE = ( vaL * vaR );
    cout << "The element-by-element result of "
         << "the multiplication is the\n valarray: ( ";
       for (i = 0 ; i < 8 ; i++ )
          cout << vaNE [ i ] << " ";
    cout << "))." << endl;
  }
```

Output

```
The initial Left valarray is: ( 2 -1 2 -1 2 -1 2 -1 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the multiplication is the
 valarray: ( 0 -1 4 -3 8 -5 12 -7 ).
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

operator+ (<valarray>)

Obtains the element-wise sum between corresponding elements of two equally sized valarrays or of between a valarray a specified value.

```
template<class Type>
   valarray<bool> operator+(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<bool> operator+(
      const valarray<Type> _Left,
      const Type& _Right
   );
template<class Type>
   valarray<bool> operator+(
      const Type& _Left,
      const valarray<Type>& _Right
   );
```

Parameters

_Left

The first of the two valarrays whose elements are to be added or a specified value to be added with each element of a valarray.

_Right

The second of the two valarrays whose elements are to be added or a specified value to be added with each element of a valarray.

Return Value

A valarray whose elements are the element-wise sum of *_Left* and *_Right*.

Example

```
// valarray_op_esum.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
   valarray<int> vaNE ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  2;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  -1;
   for ( i = 0 ; i < 8 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial Left valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaR [ i ] << " ";
```

```
    cout << "))." << endl;

    vaNE = ( vaL + vaR );
    cout << "The element-by-element result of "
         << "the sum is the\n valarray: ( ";
       for ( i = 0 ; i < 8 ; i++ )
          cout << vaNE [ i ] << " ";
    cout << "))." << endl;
  }
```

Output

```
The initial Left valarray is: ( 2 -1 2 -1 2 -1 2 -1 )).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 )).
The element-by-element result of the sum is the
 valarray: ( 2 0 4 2 6 4 8 6 )).
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

operator- (<valarray>)

Obtains the element-wise difference between corresponding elements of two equally sized valarrays or of between a valarray a specified value.

```
template<class Type>
   valarray<bool> operator-(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<bool> operator-(
      const valarray<Type> _Left,
      const Type& _Right
   );
template<class Type>
   valarray<bool> operator-(
      const Type& _Left,
      const valarray<Type>& _Right
   );
```

Parameters

_Left

A value or valarray that serves as the minuend from which other values or valarrays are to be subtracted in forming the difference.

_Right

A value or valarray that serves as the subtrahend that is to be subtracted from other values or valarrays in forming the difference.

Return Value

A valarray whose elements are the element-wise difference of *_Left* and *_Right*.

Remarks

The arithmetic terminology used in describing a subtraction:

difference = minuend - subtrahend

Example

```
// valarray_op_ediff.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
   valarray<int> vaNE ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  10;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 0 ; i < 8 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial Left valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
```

```
    cout << "))." << endl;

    cout << "The initial Right valarray is: ( ";
       for ( i = 0 ; i < 8 ; i++ )
          cout << vaR [ i ] << " ";
    cout << "))." << endl;

    vaNE = ( vaL - vaR );
    cout << "The element-by-element result of "
         << "the difference is the\n valarray: ( ";
       for (i = 0 ; i < 8 ; i++ )
          cout << vaNE [ i ] << " ";
    cout << "))." << endl;
  }
```

Output

```
The initial Left valarray is: ( 10 0 10 0 10 0 10 0 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the difference is the
 valarray: ( 10 -1 8 -3 6 -5 4 -7 ).
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

operator/ (<valarray>)

Obtains the element-wise quotient between corresponding elements of two equally sized valarrays or of between a valarray a specified value.

```
template<class Type>
   valarray<bool> operator/(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<bool> operator/(
      const valarray<Type> _Left,
      const Type& _Right
   );
template<class Type>
   valarray<bool> operator/(
      const Type& _Left,
      const valarray<Type>& _Right
   );
```

Parameters

_Left

A value or valarray that serves as the dividend into which another value or valarray is to be divided in forming the quotient.

_Right

A value or valarray that serves as the divisor and that divides another value or valarray in forming the quotient.

Return Value

A valarray whose elements are the element-wise quotient of *_Left* divided by *_Right*.

Remarks

The arithmetic terminology used in describing a division:

quotient = dividend / divisor

Example

```
// valarray_op_equo.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<double> vaL ( 6 ), vaR ( 6 );
   valarray<double> vaNE ( 6 );
   for ( i = 0 ; i < 6 ; i += 2 )
      vaL [ i ] =  100;
   for ( i = 1 ; i < 6 ; i += 2 )
      vaL [ i ] =  -100;
   for ( i = 0 ; i < 6 ; i++ )
      vaR [ i ] =  2*i;

   cout << "The initial Left valarray is: ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;
```

```
    cout << "The initial Right valarray is: ( ";
       for ( i = 0 ; i < 6 ; i++ )
          cout << vaR [ i ] << " ";
    cout << "))." << endl;

    vaNE = ( vaL / vaR );
    cout << "The element-by-element result of "
         << "the quotient is the\n valarray: ( ";
       for ( i = 0 ; i < 6 ; i++ )
          cout << vaNE [ i ] << " ";
    cout << "))." << endl;
  }
```

Output

```
The initial Left valarray is: ( 100 -100 100 -100 100 -100 ).
The initial Right valarray is: ( 0 2 4 6 8 10 ).
The element-by-element result of the quotient is the
 valarray: ( 1.#INF -50 25 -16.6667 12.5 -10 ).
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

operator== (<valarray>)

Tests whether the corresponding elements of two equally sized valarrays are equal or whether all the elements of a valarray are equal a specified value.

```
template<class Type>
   valarray<bool> operator==(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<bool> operator==(
      const valarray<Type> _Left,
      const Type& _Right
   );
template<class Type>
   valarray<bool> operator==(
      const Type& _Left,
      const valarray<Type>& _Right
  );
```

Parameters

_Left

The first of the two valarrays whose elements are to be tested for equality.

_Right

The second of the two valarrays whose elements are to be tested for equality.

Return Value

A valarray of Boolean values, each of which is:

- **true** if the corresponding elements are equal.
- **false** if the corresponding elements are not equal.

Remarks

The first template operator returns an object of class valarray<bool>, each of whose elements *I* is *_Left*[*I*] == *_Right*[*I*]. The second template operator stores in element *I* *_Left*[*I*] == *_Right*. The third template operator stores in element *I* *_Left* == *_Right*[*I*].

Example

```
// valarray_op_eq.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<bool> vaNE ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial Left valarray is: ( ";
```

```
      for (i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << "))." << endl;

   vaNE = ( vaL == vaR );
   cout << "The element-by-element result of "
        << "the equality comparison test is the\n valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaNE [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The element-by-element result of the equality comparison test is the
 valarray: ( 1 1 0 1 0 1 0 1 0 1 ).
```

Requirements

Header: <valarray>

See Also

Reference

stack::operator==

Other Resources

<valarray> Members

operator^

Obtains the bitwise exclusive **OR** (**XOR**) between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the element type.

```
template<class Type>
   valarray<bool> operator^(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<bool> operator^(
      const valarray<Type> _Left,
      const Type& _Right
   );
template<class Type>
   valarray<bool> operator^(
      const Type& _Left,
      const valarray<Type>& _Right
   );
```

Parameters

_Left

The first of the two valarrays whose respective elements are to be combined with the bitwise **XOR** or a specified value of the element type to be combined bitwise with each element of a valarray.

_Right

The second of the two valarrays whose respective elements are to be combined with the bitwise **XOR** or a specified value of the element type to be combined bitwise with each element of a valarray.

Return Value

A valarray whose elements are the element-wise combination of the bitwise **XOR** operation of *_Left* and *_Right*.

Remarks

A bitwise operation can only be used to manipulate bits in **char** and **int** data types and variants and not on **float**, **double**, **long double**, **void bool** or other, more complex data types.

The bitwise exclusive **OR** (**XOR**) has the following semantics: Given bits *b*1 and *b*2, *b*1 **XOR** *b*2 is **true** if exactly one of the bits is true; **false** if both bits are false or if both bits are true.

Example

```
// valarray_op_xor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<int> vaLAA ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  1;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 0 ; i < 10 ; i += 3 )
      vaR [ i ] =  i;
   for ( i = 1 ; i < 10 ; i += 3 )
      vaR [ i ] =  i-1;
```

```
    for ( i = 2 ; i < 10 ; i += 3 )
       vaR [ i ] =  i-1;

    cout << "The initial Left valarray is:  ( ";
       for ( i = 0 ; i < 10 ; i++ )
          cout << vaL [ i ] << " ";
    cout << "))." << endl;

    cout << "The initial Right valarray is: ( ";
       for ( i = 0 ; i < 10 ; i++ )
          cout << vaR [ i ] << " ";
    cout << "))." << endl;

    vaLAA = ( vaL ^ vaR );
    cout << "The element-by-element result of "
         << "the bitwise XOR operator^ is the\n valarray: ( ";
            for ( i = 0 ; i < 10 ; i++ )
          cout << vaLAA [ i ] << " ";
    cout << "))." << endl;
  }
```

Output

```
The initial Left valarray is:  ( 1 0 1 0 1 0 1 0 1 0 ).
The initial Right valarray is: ( 0 0 1 3 3 4 6 6 7 9 ).
The element-by-element result of the bitwise XOR operator^ is the
 valarray: ( 1 0 0 3 2 4 7 6 6 9 ).
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

operator|

Obtains the bitwise **OR** between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the element type.

```
template<class Type>
   valarray<bool> operator|(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<bool> operator|(
      const valarray<Type> _Left,
      const Type& _Right
   );
template<class Type>
   valarray<bool> operator|(
      const Type& _Left,
      const valarray<Type>& _Right
   );
```

Parameters

_Left

The first of the two valarrays whose respective elements are to be combined with the bitwise **OR** or a specified value of the element type to be combined bitwise with each element of a valarray.

_Right

The second of the two valarrays whose respective elements are to be combined with the bitwise **OR** or a specified value of the element type to be combined bitwise with each element of a valarray.

Return Value

A valarray whose elements are the element-wise combination of the bitwise **OR** operation of *_Left* and *_Right*.

Remarks

A bitwise operation can only be used to manipulate bits in **char** and **int** data types and variants and not on **float**, **double**, **long double**, **void**, **bool** or other, more complex data types.

The bitwise OR has the same truth table as the logical **OR**, but applies to the data type on the level of the individual bits. Given bits *b*1 and *b*2, *b*1 **OR** *b*2 is **true** if at least one of the bits is true or **false** if both bits are false. The logical **OR** operator|| applies on an element level, counting all nonzero values as **true**, and the result is a valarray of Boolean values. The bitwise OR **operator|**, by contrast, can result in a valarray of values other than 0 or 1, depending on the outcome of the bitwise operation.

Example

```
// valarray_op_bitor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<int> vaLAA ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  1;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 0 ; i < 10 ; i += 3 )
      vaR [ i ] =  i;
```

```
    for ( i = 1 ; i < 10 ; i += 3 )
       vaR [ i ] =  i-1;
    for ( i = 2 ; i < 10 ; i += 3 )
       vaR [ i ] =  i-1;

    cout << "The initial Left valarray is:  ( ";
       for (i = 0 ; i < 10 ; i++ )
          cout << vaL [ i ] << " ";
    cout << "))." << endl;

    cout << "The initial Right valarray is: ( ";
       for (i = 0 ; i < 10 ; i++ )
          cout << vaR [ i ] << " ";
    cout << "))." << endl;

    vaLAA = ( vaL | vaR );
    cout << "The element-by-element result of "
         << "the bitwise OR operator| is the\n valarray: ( ";
       for (i = 0 ; i < 10 ; i++ )
          cout << vaLAA [ i ] << " ";
    cout << "))." << endl;
  }
```

Output

```
The initial Left valarray is:  ( 1 0 1 0 1 0 1 0 1 0 ).
The initial Right valarray is: ( 0 0 1 3 3 4 6 6 7 9 ).
The element-by-element result of the bitwise OR operator| is the
 valarray: ( 1 0 1 3 3 4 7 6 7 9 ).
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

operator||

Obtains the logical **OR** between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the valarray element type.

```
template<class Type>
   valarray<bool> operator||(
      const valarray<Type>& _Left,
      const valarray<Type>& _Right
   );
template<class Type>
   valarray<bool> operator||(
      const valarray<Type> _Left,
      const Type& _Right
   );
template<class Type>
   valarray<bool> operator||(
      const Type& _Left,
      const valarray<Type>& _Right
   );
```

Parameters

_Left

The first of the two valarrays whose respective elements are to be combined with the logical **OR** or a specified value of the element type to be combined with each element of a valarray.

_Right

The second of the two valarrays whose respective elements are to be combined with the logical **OR** or a specified value of the element type to be combined with each element of a valarray.

Return Value

A valarray whose elements are of type **bool** and are the element-wise combination of the logical OR operation of *_Left* and *_Right*.

Remarks

The logical **OR operator||** applies on an element level, counting all nonzero values as **true**, and the result is a valarray of Boolean values. The bitwise version of **OR**, operator| by contrast, can result in a valarray of values other than 0 or 1, depending on outcome of the bitwise operation.

Example

```
// valarray_op_logor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<bool> vaLOR ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i-1;
   for ( i = 0 ; i < 10 ; i += 3 )
      vaR [ i ] =  i;
   for ( i = 1 ; i < 10 ; i += 3 )
      vaR [ i ] =  0;
```

```
    for ( i = 2 ; i < 10 ; i += 3 )
       vaR [ i ] =  0;

    cout << "The initial Left valarray is:  ( ";
       for ( i = 0 ; i < 10 ; i++ )
          cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
       for ( i = 0 ; i < 10 ; i++ )
          cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaLOR = ( vaL || vaR );
    cout << "The element-by-element result of "
         << "the logical OR operator|| is the\n valarray: ( ";
       for ( i = 0 ; i < 10 ; i++ )
          cout << vaLOR [ i ] << " ";
    cout << ")." << endl;
 }
```

Output

```
The initial Left valarray is:  ( 0 0 0 2 0 4 0 6 0 8 ).
The initial Right valarray is: ( 0 0 0 3 0 0 6 0 0 9 ).
The element-by-element result of the logical OR operator|| is the
 valarray: ( 0 0 0 1 0 1 1 1 0 1 ).
```

Requirements

Header: <valarray>

See Also

Other Resources

<valarray> Members

<valarray> Classes

For information about the classes in <valarray>, see <valarray> Members.

gslice Class

A utility class to valarray that is used to define multidimensional subsets of a valarray. If a valarray is regarded as a multidimensional matrix with all elements in an array, then the slice extracts a vectors out of the multidimensional array.

For a list of all members of this type, see gslice Members.

Remarks

The class stores the parameters that characterize an object of type gslice_array. The subset of a valarray is indirectly constructed when an object of class gslice appears as an argument for an object of class valarray**<Type>**. The stored values that specify the subset selected from the parent valarray include:

- A starting index.
- A length vector of class **valarray<size_t>**.
- A stride vector of class **valarray<size_t>**.

The two vectors must have the same length.

If the set defined by a gslice is the subset of a constant valarray, then the gslice is a new valarray. If the set defined by a gslice is the subset of a nonconstant valarray, then the gslice has reference semantics to the original valarray. The evaluation mechanism for nonconstant valarrays saves time and memory.

Operations on valarrays are guaranteed only if the source and destination subsets defined by the gslices are distinct and all indices are valid.

Requirements

Header: <valarray>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

gslice Members

<valarray> Members

gslice Members

Reference

Constructors

gslice	Defines a subset of a **valarray** that consists of multiple slices of the **valarray** that all start at a specified element.

Member Functions

size	Finds the array values specifying the numbers of elements in a general slice of a **valarray**.
start	Finds the starting index of a general slice of a **valarray**.
stride	Finds the distance between elements in a general slice of a **valarray**.

See Also

Reference

gslice Class

Thread Safety in the Standard C++ Library

gslice Member Functions

For information about the member functions in the gslice class, see gslice Members.

gslice::gslice

A utility class to valarray that is used to define multi-dimensional slices of a valarray.

```
gslice( );
gslice(
   size_t _StartIndex,
   const valarray<size_t> _LenArray,
   const valarray<size_t> _IncArray
);
```

Parameters

_StartIndex

The valarray index of the first element in the subset.

_LenArray

An array specifying the number of elements in each slice.

_IncArray

An array specifying the stride in each slice.

Return Value

The default constructor stores zero for the starting index, and zero-length vectors for the length and stride vectors. The second constructor stores *_StartIndex* for the starting index, *_LenArray* for the length array, and *_IncArray* for the stride array.

Remarks

gslice defines a subset of a valarray that consists of multiple slices of the valarray that each start at the same specified element. The ability to use arrays to define multiple slices is the only difference between **gslice** and slice::slice. The first slice has a first element with an index of *_StartIndex*, a number of elements specified by the first element of *_LenArray*, and a stride given by the first element of *_IncArray*. The next set of orthogonal slices has first elements given by the first slice. The second element of *_LenArray* specifies the number of elements. The stride is given by the second element of *_IncArray*. A third dimension of slices would take the elements of the two-dimensional array as the starting elements and proceed analogously

Example

```
// gslice_ctor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> va ( 20 ), vaResult;
   for ( i = 0 ; i < 20 ; i+=1 )
      va [ i ] =  i;

   cout << "The operand valarray va is:" << endl << "(";
   for ( i = 0 ; i < 20 ; i++ )
      cout << " " << va [ i ];
   cout << " )" << endl;

   valarray<size_t> Len ( 2 ), Stride ( 2 );
   Len [0] = 4;
   Len [1] = 4;
   Stride [0] = 7;
   Stride [1] = 4;

   gslice vaGSlice ( 0, Len, Stride );
```

```
    vaResult = va [ vaGSlice ];

    cout << "The valarray for vaGSlice is vaResult:" << endl
         << "va[vaGSlice] = (";

    for ( i = 0 ; i < 8 ; i++ )
       cout << " " << vaResult [ i ];
    cout << ")" << endl;
  }
```

Output

```
The operand valarray va is:
( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 )
The valarray for vaGSlice is vaResult:
va[vaGSlice] = ( 0 4 8 12 7 11 15 19)
```

Requirements

Header: <valarray>

See Also

Reference

gslice Class

Other Resources

gslice Members

gslice::size

Finds the array values specifying the numbers of elements in a general slice of a valarray.

```
const valarray<size_t> size( ) const;
```

Return Value

A valarray specifying the number of elements in each slice of a general slice of a valarray.

Remarks

The member function returns the stored lengths of slices.

Example

```
// gslice_size.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
   size_t sizeVA;

   valarray<int> va ( 20 ), vaResult;
   for ( i = 0 ; i < 20 ; i+=1 )
      va [ i ] =  i;

   cout << "The operand valarray va is:\n ( ";
      for ( i = 0 ; i < 20 ; i++ )
         cout << va [ i ] << " ";
   cout << ")." << endl;

   sizeVA = va.size ( );
   cout << "The size of the valarray is: "
        << sizeVA << "." << endl << endl;

   valarray<size_t> Len ( 2 ), Stride ( 2 );
   Len [0] = 4;
   Len [1] = 4;
   Stride [0] = 7;
   Stride [1] = 4;

   gslice vaGSlice ( 0, Len, Stride );
   vaResult = va [ vaGSlice ];
   const valarray <size_t> sizeGS = vaGSlice.size ( );

   cout << "The valarray for vaGSlice is vaResult:"
        << "\n va[vaGSlice] = ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaResult [ i ] << " ";
   cout << ")." << endl;

   cout << "The size of vaResult is:"
        << "\n vaGSlice.size ( ) = ( ";
      for ( i = 0 ; i < 2 ; i++ )
         cout << sizeGS[ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The operand valarray va is:
 ( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ).
The size of the valarray is: 20.

The valarray for vaGSlice is vaResult:
 va[vaGSlice] = ( 0 4 8 12 7 11 15 19 ).
The size of vaResult is:
 vaGSlice.size ( ) = ( 4 4 ).
```

Requirements

Header: <valarray>

See Also

Reference

gslice Class

Other Resources

gslice Members

gslice::start

Finds the starting index of a general slice of a valarray.

```
size_t start( ) const;
```

Return Value

The starting index of a general slice of a valarray.

Example

```
// gslice_start.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> va ( 20 ), vaResult;
   for (i = 0 ; i < 20 ; i+=1 )
      va [ i ] =  i;

   cout << "The operand valarray va is:\n ( ";
      for ( i = 0 ; i < 20 ; i++ )
         cout << va [ i ] << " ";
   cout << ")." << endl;

   valarray<size_t> Len ( 2 ), Stride ( 2 );
   Len [0] = 4;
   Len [1] = 4;
   Stride [0] = 7;
   Stride [1] = 4;

   gslice vaGSlice ( 0, Len, Stride );
   vaResult = va [ vaGSlice ];
   size_t vaGSstart = vaGSlice.start ( );

   cout << "The valarray for vaGSlice is vaResult:"
        << "\n va[vaGSlice] = ( ";
      for (i = 0 ; i < 8 ; i++ )
         cout << vaResult [ i ] << " ";
   cout << ")." << endl;

   cout << "The index of the first element of vaResult is: "
        << vaGSstart << "." << endl;
}
```

Output

```
The operand valarray va is:
 ( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ).
The valarray for vaGSlice is vaResult:
 va[vaGSlice] = ( 0 4 8 12 7 11 15 19 ).
The index of the first element of vaResult is: 0.
```

Requirements

Header: <valarray>

See Also

Reference

gslice Class

Other Resources

gslice Members

gslice::stride

Finds the distance between elements in a general slice of a valarray.

```
const valarray<size_t> stride( ) const;
```

Return Value

A valarray specifying the distances between elements in each slice of a general slice of a valarray.

Example

```
// gslice_stride.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> va ( 20 ), vaResult;
   for (i = 0 ; i < 20 ; i+=1 )
      va [ i ] =  i;

   cout << "The operand valarray va is:\n ( ";
      for (i = 0 ; i < 20 ; i++ )
         cout << va [ i ] << " ";
   cout << "))." << endl;

   valarray<size_t> Len ( 2 ), Stride ( 2 );
   Len [0] = 4;
   Len [1] = 4;
   Stride [0] = 7;
   Stride [1] = 4;

   gslice vaGSlice ( 0, Len, Stride );
   vaResult = va [ vaGSlice ];
   const valarray <size_t> strideGS = vaGSlice.stride ( );

   cout << "The valarray for vaGSlice is vaResult:"
        << "\n va[vaGSlice] = ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaResult [ i ] << " ";
   cout << "))." << endl;

   cout << "The strides of vaResult are:"
        << "\n vaGSlice.stride ( ) = ( ";
      for ( i = 0 ; i < 2 ; i++ )
         cout << strideGS[ i ] << " ";
   cout << "))." << endl;

}
```

Output

```
The operand valarray va is:
 ( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ).
The valarray for vaGSlice is vaResult:
 va[vaGSlice] = ( 0 4 8 12 7 11 15 19 ).
The strides of vaResult are:
 vaGSlice.stride ( ) = ( 7 4 ).
```

Requirements

Header: <valarray>

See Also

Reference

gslice Class

Other Resources

gslice Members

gslice_array Class

An internal, auxiliary template class that supports general slice objects by providing operations between subset arrays defined by the general slice of a valarray.

```
template<class Type>
   class gslice_array {
public:
   typedef Type value_type;
   void operator=(
      const valarray<Type> x
   ) const;

   void operator=(
      const Type& x
   ) const;

   void operator*=(
      const valarray<Type> x
   ) const;

   void operator/=(
      const valarray<Type> x
   ) const;

   void operator%=(
      const valarray<Type> x
   ) const;

   void operator+=(
      const valarray<Type> x
   ) const;

   void operator-=(
      const valarray<Type> x
   ) const;

   void operator^=(
      const valarray<Type> x
   ) const;

   void operator&=(
      const valarray<Type> x
   ) const;

   void operator|=(
      const valarray<Type> x
   ) const;

   void operator<<=(
      const valarray<Type> x
   ) const;

   void operator>>=(
      const valarray<Type> x
   ) const;

// The rest is private or implementation defined
}
```

Remarks

The class describes an object that stores a reference to an object **va** of class valarray**<Type>**, along with an object **gs** of class gslice which describes the sequence of elements to select from the **valarray<Type>** object.

You construct a **gslice_array<Type>** object only by writing an expression of the form va[gs]. The member functions of class gslice_array then behave like the corresponding function signatures defined for **valarray<Type>**, except that only the sequence of selected elements is affected.

The template class is created indirectly by certain valarray operations and cannot be used directly in the program. An internal auxiliary template class instead is used by the slice subscript operator:

gslice_array<**Type**> **valarray**<**Type**>::**operator[]** (**const gslice&**).

You construct a **gslice_array<Type>** object only by writing an expression of the form **va[gsl]**, for a slice **gsl** of valarray **va**. The member functions of class gslice_array then behave like the corresponding function signatures defined for **valarray<Type>**, except that only the sequence of selected elements is affected. The sequence controlled by the gslice_array is defined by the three parameters of the slice constructor, the index of the first element in the first slice, the number of elements in each slice, and the distance between the elements in each slice.

In the following example:

```
const size_t lv[] = {2, 3};
const size_t dv[] = {7, 2};
const valarray<size_t> len(lv, 2), str(dv, 2);
// va[gslice(3, len, str)] selects elements with
//   indices 3, 5, 7, 10, 12, 14
```

The indices must be valid for the procedure to be valid.

Example

See the example for gslice::gslice for an example of how to declare and use a slice_array.

Requirements

Header: <valarray>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<valarray> Members

indirect_array Class

An internal, auxiliary template class that supports objects that are subsets of valarrays by providing operations between subset arrays defined by specifying a subset of indices of a parent valarray.

```
template<class Type>
   class indirect_array {
public:
   typedef Type value_type;
   void operator=(
      const valarray<Type> x
   ) const;

   void operator=(
      const Type& x
   ) const;

   void operator*=(
      const valarray<Type> x
   ) const;

   void operator/=(
      const valarray<Type> x
   ) const;

   void operator%=(
      const valarray<Type> x
   ) const;

   void operator+=(
      const valarray<Type> x
   ) const;

   void operator-=(
      const valarray<Type> x
   ) const;

   void operator^=(
      const valarray<Type> x
   ) const;

   void operator&=(
      const valarray<Type> x
   ) const;

   void operator|=(
      const valarray<Type> x
   ) const;

   void operator<<=(
      const valarray<Type> x
   ) const;

   void operator>>=(
      const valarray<Type> x
   ) const;

// The rest is private or implementation defined
}
```

Remarks

The class describes an object that stores a reference to an object **va** of class valarray**<Type>**, along with an object **xa** of class **valarray<size_t>**, which describes the sequence of elements to select from the **valarray<Type>** object.

You construct an **indirect_array<Type>** object only by writing an expression of the form **va[xa]**. The member functions of class indirect_array then behave like the corresponding function signatures defined for **valarray<Type>**, except that only the sequence of selected elements is affected.

The sequence consists of **xa.**size elements, where element *I* becomes the index **xa**[*I*] within **va**.

Example:

```
// indirect_array.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> va ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      va [ i ] =  i;
   for ( i = 1 ; i < 10 ; i += 2 )
      va [ i ] =  -1;

   cout << "The initial operand valarray is:  ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << va [ i ] << " ";
   cout << "))." << endl;

   // Select 2nd, 4th & 6th elements
   // and assign a value of 10 to them
   valarray<size_t> indx ( 3 );
   indx [0] = 2;
   indx [1] = 4;
   indx [2] = 6;
   va[indx] = 10;

   cout << "The modified operand valarray is:  ( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << va [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial operand valarray is:  ( 0 -1 2 -1 4 -1 6 -1 8 -1 ).
The modified operand valarray is:  ( 0 -1 10 -1 10 -1 10 -1 8 -1 ).
```

Requirements

Header: <valarray>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<valarray> Members

mask_array Class

An internal, auxiliary template class that supports objects that are subsets of parent valarrays, specified with a Boolean expression, by providing operations between the subset arrays.

```
template<class Type>
   class mask_array {
public:
   typedef Type value_type;
   void operator=(
      const valarray<Type> x
   ) const;

   void operator=(
      const Type& x
   ) const;

   void operator*=(
      const valarray<Type> x
   ) const;

   void operator/=(
      const valarray<Type> x
   ) const;

   void operator%=(
      const valarray<Type> x
   ) const;

   void operator+=(
      const valarray<Type> x
   ) const;

   void operator-=(
      const valarray<Type> x
   ) const;

   void operator^=(
      const valarray<Type> x
   ) const;

   void operator&=(
      const valarray<Type> x
   ) const;

   void operator|=(
      const valarray<Type> x
   ) const;

   void operator<<=(
      const valarray<Type> x
   ) const;

   void operator>>=(
      const valarray<Type> x
   ) const;

// The rest is private or implementation defined
}
```

Remarks

The class describes an object that stores a reference to an object **va** of class valarray**<Type>**, along with an object **ba** of class valarray<bool>, which describes the sequence of elements to select from the **valarray<Type>** object.

You construct a **mask_array<Type>** object only by writing an expression of the form va[ba]. The member functions of class mask_array then behave like the corresponding function signatures defined for **valarray<Type>**, except that only the sequence of selected elements is affected.

The sequence consists of at most **ba.size** elements. An element *J* is included only if **ba**[*J*] is true. Thus, there are as many elements in the sequence as there are true elements in **ba**. If *I* is the index of the lowest true element in **ba**, then **va**[*I*] is element zero in the selected sequence.

Example:

```
// mask_array.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> va ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      va [ i ] =  i;
   for ( i = 1 ; i < 10 ; i += 2 )
      va [ i ] =  -1;

   cout << "The initial operand valarray is:  ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << va [ i ] << " ";
   cout << "))." << endl;

   // Use masked subsets to assign a value of 10
   // to all elements grrater than 3 in value
   va [va > 3 ] = 10;
   cout << "The modified operand valarray is:  ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << va [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial operand valarray is:  ( 0 -1 2 -1 4 -1 6 -1 8 -1 ).
The modified operand valarray is:  ( 0 -1 2 -1 10 -1 10 -1 10 -1 ).
```

Requirements

Header: <valarray>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<valarray> Members

slice Class

A utility class to valarray that is used to define one-dimensional subsets of a parent valarray. If a valarray is regarded as a two-dimensional matrix with all elements in an array, then the slice extracts a vector in one dimension out of the two-dimensional array.

For a list of all members of this type, see slice Members.

Remarks

The class stores the parameters that characterize an object of type slice_array The subset of a valarray is indirectly constructed when an object of class slice appears as an argument for an object of class valarray**<Type>**. The stored values that specify the subset selected from the parent valarray include:

- A starting index in the valarray.
- A total length, or number of elements in the slice.
- A stride, or distance between subsequent indices of elements in the valarray.

If the set defined by a slice is the subset of a constant valarray, then the slice is a new valarray. If the set defined by a slice is the subset of a nonconstant valarray, then the slice has reference semantics to the original valarray. The evaluation mechanism for nonconstant valarrays saves time and memory.

Operations on valarrays are guaranteed only if the source and destination subsets defined by the slices are distinct and all indices are valid.

Requirements

Header: <valarray>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

slice Members

<valarray> Members

slice Members

Reference

Constructors

slice	Defines a subset of a **valarray** that consists of a number of elements that are an equal distance apart and that start at a specified element.

Member Functions

size	Finds the number of elements in a slice of a **valarray**.
start	Finds the starting index of a slice of a **valarray**.
stride	Finds the distance between elements in a slice of a **valarray**.

See Also

Reference

slice Class

Thread Safety in the Standard C++ Library

slice Member Functions

For information about the member functions in the slice class, see slice Members.

slice::size

Finds the number of elements in a slice of a valarray.

```
size_t size( ) const;
```

Return Value

The number of elements in a slice of a valarray.

Example

```
// slice_size.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
   size_t sizeVA, sizeVAR;

   valarray<int> va ( 20 ), vaResult;
   for ( i = 0 ; i < 20 ; i += 1 )
      va [ i ] =  i+1;

   cout << "The operand valarray va is:\n ( ";
      for ( i = 0 ; i < 20 ; i++ )
         cout << va [ i ] << " ";
   cout << "))." << endl;

   sizeVA = va.size ( );
   cout << "The size of the valarray is: "
        << sizeVA << "." << endl << endl;

   slice vaSlice ( 3 , 6 , 3 );
   vaResult = va [ vaSlice ];

   cout << "The slice of valarray va is vaResult = "
        << "va[slice( 3, 6, 3)] =\n ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaResult [ i ] << " ";
   cout << "))." << endl;

   sizeVAR = vaSlice.size ( );
   cout << "The size of slice vaSlice is: "
        << sizeVAR << "." << endl;
}
```

Output

```
The operand valarray va is:
 ( 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ).
The size of the valarray is: 20.

The slice of valarray va is vaResult = va[slice( 3, 6, 3)] =
 ( 4 7 10 13 16 19 ).
The size of slice vaSlice is: 6.
```

Requirements

Header: <valarray>

See Also

Reference

slice Class

Other Resources

slice Members

slice::slice

Defines a subset of a valarray that consists of a number of elements that are an equal distance apart and that start at a specified element.

```
slice( );
slice(
   size_t _StartIndex,
   const valarray<size_t> _Len,
   const valarray<size_t> _Stride
);
```

Parameters

_StartIndex

The valarray index of the first element in the subset.

_Len

The number of elements in the subset.

_Stride

The distance between elements in the subset.

Return Value

The default constructor stores zeros for the starting index, total length, and stride. The second constructor stores *_StartIndex* for the starting index, *_Len* for the total length, and *_Stride* for the stride.

Remarks

The stride may be negative.

Example

```
// slice_ctor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> va ( 20 ), vaResult;
   for ( i = 0 ; i < 20 ; i+=1 )
      va [ i ] =  2 * (i + 1 );

   cout << "The operand valarray va is:\n( ";
      for ( i = 0 ; i < 20 ; i++ )
         cout << va [ i ] << " ";
   cout << "))." << endl;

   slice vaSlice ( 1 , 7 , 3 );
   vaResult = va [ vaSlice ];

   cout << "\nThe slice of valarray va is vaResult:"
        << "\nva[slice( 1, 7, 3)] = ( ";
      for ( i = 0 ; i < 7 ; i++ )
         cout << vaResult [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The operand valarray va is:
( 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 ).

The slice of valarray va is vaResult:
va[slice( 1, 7, 3)] = ( 4 10 16 22 28 34 40 ).
```

Requirements

Header: <valarray>

See Also

Reference

slice Class

Other Resources

slice Members

slice::start

Finds the starting index of a slice of a valarray.

```
size_t start( ) const;
```

Return Value

The starting index of a slice of a valarray.

Example

```
// slice_start.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
   size_t startVAR;

   valarray<int> va ( 20 ), vaResult;
   for ( i = 0 ; i < 20 ; i += 1 )
      va [ i ] = i+1;

   cout << "The operand valarray va is:\n ( ";
      for ( i = 0 ; i < 20 ; i++ )
         cout << va [ i ] << " ";
   cout << "))." << endl;

   slice vaSlice ( 3 , 6 , 3 );
   vaResult = va [ vaSlice ];

   cout << "The slice of valarray va is vaResult = "
        << "va[slice( 3, 6, 3)] =\n ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaResult [ i ] << " ";
   cout << "))." << endl;

   startVAR = vaSlice.start ( );
   cout << "The start index of slice vaSlice is: "
        << startVAR << "." << endl;
}
```

Output

```
The operand valarray va is:
 ( 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ).
The slice of valarray va is vaResult = va[slice( 3, 6, 3)] =
 ( 4 7 10 13 16 19 ).
The start index of slice vaSlice is: 3.
```

Requirements

Header: <valarray>

See Also

Reference

slice Class

Other Resources

slice Members

slice::stride

Finds the distance between elements in a slice of a valarray.

```
size_t stride( ) const;
```

Return Value

The distance between elements in a slice of a valarray.

Example

```
// slice_stride.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
   size_t strideVAR;

   valarray<int> va ( 20 ), vaResult;
   for ( i = 0 ; i < 20 ; i += 1 )
      va [ i ] =  3 * ( i + 1 );

   cout << "The operand valarray va is:\n ( ";
      for ( i = 0 ; i < 20 ; i++ )
         cout << va [ i ] << " ";
   cout << "))." << endl;

   slice vaSlice ( 4 , 5 , 3 );
   vaResult = va [ vaSlice ];

   cout << "The slice of valarray va is vaResult = "
        << "va[slice( 4, 5, 3)] =\n ( ";
      for ( i = 0 ; i < 5 ; i++ )
         cout << vaResult [ i ] << " ";
   cout << "))." << endl;

   strideVAR = vaSlice.stride ( );
   cout << "The stride of slice vaSlice is: "
        << strideVAR << "." << endl;
}
```

Output

```
The operand valarray va is:
 ( 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 ).
The slice of valarray va is vaResult = va[slice( 4, 5, 3)] =
 ( 15 24 33 42 51 ).
The stride of slice vaSlice is: 3.
```

Requirements

Header: <valarray>

See Also

Reference

slice Class

Other Resources

slice Members

slice_array Class

An internal, auxiliary template class that supports slice objects by providing operations between subset arrays defined by the slice of a valarray.

```
template<class Type>
   class slice_array {
public:
   typedef Type value_type;
   void operator=(
      const valarray<Type> x
   ) const;
   void operator=(
      const Type& x
   ) const;
   void operator*=(
      const valarray<Type> x
   ) const;
   void operator/=(
      const valarray<Type> x
   ) const;
   void operator%=(
      const valarray<Type> x
   ) const;
   void operator+=(
      const valarray<Type> x
   ) const;
   void operator-=(
      const valarray<Type> x
   ) const;
   void operator^=(
      const valarray<Type> x
   ) const;
   void operator&=(
      const valarray<Type> x
   ) const;
   void operator|=(
      const valarray<Type> x
   ) const;
   void operator<<=(
      const valarray<Type> x
   ) const;
   void operator>>=(
      const valarray<Type> x
   ) const;
// The rest is private or implementation defined
}
```

Remarks

The class describes an object that stores a reference to an object of class valarray**<Type>**, along with an object of class slice, which describes the sequence of elements to select from the **valarray<Type>** object.

The template class is created indirectly by certain valarray operations and cannot be used directly in the program. An internal, auxiliary template class that is used by the slice subscript operator:

slice_array<**Type**> **valarray**<**Type**::**operator[]** (**slice**).

You construct a **slice_array<Type>** object only by writing an expression of the form va[sl], for a slice **sl** of valarray **va**. The member functions of class slice_array then behave like the corresponding function signatures defined for **valarray<Type>**, except that only the sequence of selected elements is affected. The sequence controlled by the slice_array is defined by the three parameters of the slice constructor, the index of the first element in the slice, the number of elements, and the distance between the elements. A slice_array cut from valarray **va** declared by **va**[**slice**(2, 5, 3)] selects elements with indices 2, 5, 8, 11, and 14 from **va**. The indices must be valid for the procedure to be valid.

Example

See the example for slice::slice for an example of how to declare and use a slice_array.

Requirements

Header: <valarray>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

<valarray> Members

valarray Class

The template class describes an object that controls a sequence of elements of type **Type** that are stored as an array, designed for performing high-speed mathematical operations, and optimized for computational performance.

For a list of all members of this type, see valarray Members.

Remarks

The class is a representation of the mathematical concept of an ordered set of values and the elements are numbered sequentially from zero. The class is described as a near container because it supports some, but not all, of the capabilities that first-class sequence containers, such as vector, support. It differs from template class vector in two important ways:

- It defines numerous arithmetic operations between corresponding elements of **valarray<Type>** objects of the same type and length, such as *xarr* = cos(*yarr*) + sin(*zarr*).
- It defines a variety of interesting ways to subscript a **valarray<Type>** object, by overloading operator[].

An object of class **Type**:

- Has a public default constructor, destructor, copy constructor, and assignment operator, with conventional behavior.
- Defines the arithmetic operators and math functions, as needed, that are defined for the floating-point types, with conventional behavior.

In particular, no subtle differences may exist between copy construction and default construction followed by assignment. None of the operations on objects of class **Type** may throw exceptions.

Requirements

Header: <valarray>

See Also

Reference

Thread Safety in the Standard C++ Library

Other Resources

valarray Members

<valarray> Members

valarray Members

Reference

Constructors

valarray	Constructs a **valarray** of a specific size or with elements of a specific value or as a copy of another **valarray** or subset of another **valarray**.

Typedefs

value_type	A type that represents the type of element stored in a **valarray**.

Member Functions

apply	Applies a specified function to each element of a **valarray**.
cshift	Cyclically shifts all the elements in a **valarray** by a specified number of positions.
max	Finds the largest element in a **valarray**.
min	Finds the smallest element in a **valarray**.
resize	Changes the number of elements in a **valarray** to a specified number, adding or removing elements as required.
shift	Shifts all the elements in a **valarray** by a specified number of positions.
size	Finds the number of elements in a **valarray**.
sum	Determines the sum of all the elements in a **valarray** of nonzero length.

Operators

operator!	A unary operator that obtains the logical **NOT** values of each element in a **valarray**.
operator%=	Obtains the remainder of dividing the elements of an array element-wise either by a specified **valarray** or by a value of the element type.
operator&=	Obtains the bitwise **AND** of elements in an array either with the corresponding elements in a specified **valarray** or with a value of the element type.
operator>>=	Right-shifts the bits for each element of a **valarray** operand a specified number of positions or by an element-wise amount specified by a second **valarray**.
operator<<=	Left-shifts the bits for each element of a **valarray** operand a specified number of positions or by an element-wise amount specified by a second **valarray**.
operator*=	Multiplies the elements of a specified **valarray** or a value of the element type, element-wise, to an operand **valarray**.
operator+	A unary operator that applies a plus to each element in a **valarray**.
operator+=	Adds the elements of a specified **valarray** or a value of the element type, element-wise, to an operand **valarray**.
operator-	A unary operator that applies a minus to each element in a **valarray**.
operator-=	Subtracts the elements of a specified **valarray** or a value of the element type, element-wise, from an operand **valarray**.

operator/=	Divides an operand **valarray** element-wise by the elements of a specified **valarray** or a value of the element type.
operator=	Assigns elements to a **valarray** whose values are specified either directly or as part of some other **valarray** or by a **slice_array**, **gslice_array**, **mask_array**, or **indirect_array**.
operator[]	Returns a reference to an element or its value at specified index or a specified subset.
operator^=	Obtains the element-wise exclusive logical or operator (**XOR**) of an array with either a specified valarray or a value of the element type.
operator|=	Obtains the bitwise **OR** of elements in an array either with the corresponding elements in a specified **valarray** or with a value of the element type.
operator~	A unary operator that obtains the bitwise **NOT** values of each element in a **valarray**.

See Also

Reference

valarray Class

Thread Safety in the Standard C++ Library

valarray Typedefs

For information about the typedefs in the bitset class, see valarray Members.

valarray::value_type

A type that represents the type of element stored in a valarray.

```
typedef Type value_type;
```

Remarks

The type is a synonym for the template parameter **Type**.

Example

```
// valarray_value_type.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
   valarray<int> va ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      va [ i ] =  i;
   for ( i = 1 ; i < 10 ; i += 2 )
      va [ i ] =  -1;

   cout << "The initial operand valarray is:  ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << va [ i ] << " ";
   cout << "))." << endl;

   // value_type declaration and initialization:
   valarray<int>::value_type Right = 10;

   cout << "The decalared value_type Right is: "
           << Right << endl;
   va *= Right;
   cout << "The resulting valarray is:  ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << va [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial operand valarray is:  ( 0 -1 2 -1 4 -1 6 -1 8 -1 ).
The decalared value_type Right is: 10
The resulting valarray is:  ( 0 -10 20 -10 40 -10 60 -10 80 -10 ).
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray Member Functions

For information about the functions in the valarray class, see valarray Members.

valarray::apply

Applies a specified function to each element of a valarray.

```
valarray<Type> apply(
   Type _Func(Type)
) const;
valarray<Type> apply(
   Type _Func(const Type&)
) const;
```

Parameters

_Func(Type)

The function object to be applied to each element of the operand valarray.

_Func(const Type&)

The function object for const to be applied to each element of the operand valarray.

Return Value

A valarray whose elements have had *_Func* applied element-wise to the elements of the operand valarray.

Remarks

The member function returns an object of class valarray**<Type>**, of length size, each of whose elements *I* is **func**((***this**)[*I*]).

Example

```
// valarray_apply.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

using namespace std;

int __cdecl MyApplyFunc( int n )
{
   return n*2;
}

int main( int argc, char* argv[] )
{
   valarray<int> vaR(10), vaApplied(10);
   int i;

   for ( i = 0; i < 10; i += 3 )
      vaR[i] = i;

   for ( i = 1; i < 10; i += 3 )
      vaR[i] = 0;

   for ( i = 2; i < 10; i += 3 )
      vaR[i] = -i;

   cout << "The initial Right valarray is: (";
   for   ( i=0; i < 10; ++i )
      cout << " " << vaR[i];
   cout << " )" << endl;

   vaApplied = vaR.apply( MyApplyFunc );

   cout << "The element-by-element result of "
       << "applying MyApplyFunc to vaR is the\nvalarray: ( ";
   for ( i = 0; i < 10; ++i )
```

```
        cout << " " << vaApplied[i];
    cout << " )" << endl;
}
```

Output

```
The initial Right valarray is: ( 0 0 -2 3 0 -5 6 0 -8 9 )
The element-by-element result of applying MyApplyFunc to vaR is the
valarray: (  0 0 -4 6 0 -10 12 0 -16 18 )
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::cshift

Cyclically shifts all the elements in a valarray by a specified number of positions.

```
valarray<Type> cshift(
   int _Count
) const;
```

Parameters

_Count

The number of places the elements are to be shifted forward.

Return Value

A new valarray in which all the elements have been moved *_Count* positions cyclically toward the front of the valarray, left with respect to their positions in the operand valarray.

Remarks

A positive value of *_Count* shifts the elements cyclically left *_Count* places.

A negative value of *_Count* shifts the elements cyclically right *_Count* places.

Example

```
// valarray_cshift.cpp
// compile with: /EHsc

#include <valarray>
#include <iostream>

int main()
{
    using namespace std;
    int i;

    valarray<int> va1(10), va2(10);
    for (i = 0; i < 10; i+=1)
        va1[i] = i;
    for (i = 0; i < 10; i+=1)
        va2[i] = 10 - i;

    cout << "The operand valarray va1 is: (";
    for (i = 0; i < 10; i++)
        cout << " " << va1[i];
    cout << ")" << endl;

    // A positive parameter shifts elements right
    va1 = va1.cshift(4);
    cout << "The cyclically shifted valarray va1 is:\nva1.cshift (4) = (";
    for (i = 0; i < 10; i++)
        cout << " " << va1[i];
    cout << ")" << endl;

    cout << "The operand valarray va2 is: (";
    for (i = 0; i < 10; i++)
        cout << " " << va2[i];
    cout << ")" << endl;

    // A negative parameter shifts elements left
    va2 = va2.cshift(-4);
    cout << "The cyclically shifted valarray va2 is:\nva2.shift (-4) = (";
    for (i = 0; i < 10; i++)
        cout << " " << va2[i];
    cout << ")" << endl;
```

```
}
```

Output

```
The operand valarray va1 is: ( 0 1 2 3 4 5 6 7 8 9)
The cyclically shifted valarray va1 is:
va1.cshift (4) = ( 4 5 6 7 8 9 0 1 2 3)
The operand valarray va2 is: ( 10 9 8 7 6 5 4 3 2 1)
The cyclically shifted valarray va2 is:
va2.shift (-4) = ( 4 3 2 1 10 9 8 7 6 5)
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::max

Finds the largest element in a valarray.

```
Type max( ) const;
```

Return Value

The maximum value of the elements in the operand valarray.

Remarks

The member function compares values by applying **operator<** or **operator>** between pairs of elements of class **Type**, for which operators must be provided for the element **Type**.

Example

```
// valarray_max.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i, MaxValue;

   valarray<int> vaR ( 10 );
   for ( i = 0 ; i < 10 ; i += 3 )
      vaR [ i ] =  i;
   for ( i = 1 ; i < 10 ; i += 3 )
      vaR [ i ] =  2*i - 1;
   for ( i = 2 ; i < 10 ; i += 3 )
      vaR [ i ] =  10 - i;

   cout << "The operand valarray is: ( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << "))." << endl;

   MaxValue = vaR.max (  );
   cout << "The largest element in the valarray is: "
        << MaxValue  << "." << endl;
}
```

Output

```
The operand valarray is: ( 0 1 8 3 7 5 6 13 2 9 ).
The largest element in the valarray is: 13.
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::min

Finds the smallest element in a valarray.

```
Type min( ) const;
```

Return Value

The minimum value of the elements in the operand valarray.

Remarks

The member function compares values by applying **operator<** or **operator>** between pairs of elements of class **Type**, for which operators must be provided for the element **Type**.

Example

```
// valarray_min.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i, MinValue;

   valarray<int> vaR ( 10 );
   for ( i = 0 ; i < 10 ; i += 3 )
      vaR [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 3 )
      vaR [ i ] =  2*i;
   for ( i = 2 ; i < 10 ; i += 3 )
      vaR [ i ] =  5 - i;

   cout << "The operand valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << "))." << endl;

   MinValue = vaR.min ( );
   cout << "The smallest element in the valarray is: "
        << MinValue  << "." << endl;
}
```

Output

```
The operand valarray is: ( 0 2 3 -3 8 0 -6 14 -3 -9 ).
The smallest element in the valarray is: -9.
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::resize

Changes the number of elements in a valarray to a specified number.

```
void resize(
   size_t _Newsize
);
void resize(
   size_t _Newsize,
   const Type& _Val
);
```

Parameters

_Newsize

The number of elements in the resized valarray.

_Val

The value to be given to the elements of the resized valarray.

Remarks

The first member function initializes elements with their default constructor.

Any pointers or references to elements in the controlled sequence are invalidated.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// valarray_resize.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main()
{
    using namespace std;
    int i;
    size_t size1, sizeNew;

    valarray<int> va1(10);
    for (i = 0; i < 10; i+=1)
        va1[i] = i;

    cout << "The valarray contains ( ";
        for (i = 0; i < 10; i++)
            cout << va1[i] << " ";
    cout << "))." << endl;

    size1 = va1.size();
    cout << "The number of elements in the valarray is: "
         << size1  << "." <<endl << endl;

    va1.resize(15, 10);

    cout << "The valarray contains ( ";
        for (i = 0; i < 15; i++)
            cout << va1[i] << " ";
    cout << "))." << endl;
    sizeNew = va1.size();
    cout << "The number of elements in the resized valarray is: "
         << sizeNew  << "." <<endl << endl;
```

```
}
```

Output

```
The valarray contains ( 0 1 2 3 4 5 6 7 8 9 ).
The number of elements in the valarray is: 10.

The valarray contains ( 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 ).
The number of elements in the resized valarray is: 15.
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::shift

Shifts all the elements in a valarray by a specified number of places.

```
valarray<Type> shift(
   int _Count
) const;
```

Parameters

_Count

The number of places the elements are to be shifted forward.

Return Value

A new valarray in which all the elements have been moved *_Count* positions toward the front of the valarray, left with respect to their positions in the operand valarray.

Remarks

A positive value of *_Count* shifts the elements left *_Count* places, with zero fill.

A negative value of *_Count* shifts the elements right *_Count* places, with zero fill.

Example

```
// valarray_shift.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> va1 ( 10 ), va2 ( 10 );
   for ( i = 0 ; i < 10 ; i += 1 )
      va1 [ i ] =  i;
   for ( i = 0 ; i < 10 ; i += 1 )
      va2 [ i ] = 10 -  i;

   cout << "The operand valarray va1(10) is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << va1 [ i ] << " ";
   cout << "))." << endl;

   // A positive parameter shifts elements left
   va1 = va1.shift ( 4 );
   cout << "The shifted valarray va1 is: va1.shift (4) = ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << va1 [ i ] << " ";
   cout << "))." << endl;

   cout << "The operand valarray va2(10) is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << va2 [ i ] << " ";
   cout << "))." << endl;

   // A negative parameter shifts elements right
   va2 = va2.shift ( - 4 );
   cout << "The shifted valarray va2 is: va2.shift (-4) = ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << va2 [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The operand valarray va1(10) is: ( 0 1 2 3 4 5 6 7 8 9 ).
The shifted valarray va1 is: va1.shift (4) = ( 4 5 6 7 8 9 0 0 0 0 ).
The operand valarray va2(10) is: ( 10 9 8 7 6 5 4 3 2 1 ).
The shifted valarray va2 is: va2.shift (-4) = ( 0 0 0 0 10 9 8 7 6 5 ).
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::size

Finds the number of elements in a valarray.

```
size_t size( ) const;
```

Return Value

The number of elements in the operand valarray.

Example

When compiling this example with the **/Wp64** flag or on a 64-bit platform, compiler warning C4267 will be generated. For more information on this warning, see Compiler Warning (level 3) C4267.

```
// valarray_size.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main()
{
    using namespace std;
    int i;
    size_t size1, size2;

    valarray<int> va1(10), va2(12);
    for (i = 0; i < 10; i += 1)
        va1[i] =  i;
    for (i = 0; i < 10; i += 1)
        va2[i] =  i;

    cout << "The operand valarray va1(10) is: ( ";
        for (i = 0; i < 10; i++)
            cout << va1[i] << " ";
    cout << "))." << endl;

    size1 = va1.size();
    cout << "The number of elements in the valarray va1 is: va1.size = "
         << size1  << "." <<endl << endl;

    cout << "The operand valarray va2(12) is: ( ";
        for (i = 0; i < 10; i++)
            cout << va2[i] << " ";
    cout << "))." << endl;

    size2 = va2.size();
    cout << "The number of elements in the valarray va2 is: va2.size = "
         << size2  << "." << endl << endl;

    // Initializing two more elements to va2
    va2[10] = 10;
    va2[11] = 11;
    cout << "After initializing two more elements,\n "
         << "the operand valarray va2(12) is now: ( ";
        for (i = 0; i < 12; i++)
            cout << va2[i] << " ";
    cout << "))." << endl;
    cout << "The number of elements in the valarray va2 is still: "
         << size2  << "." << endl;
}
```

Output

```
The operand valarray va1(10) is: ( 0 1 2 3 4 5 6 7 8 9 ).
The number of elements in the valarray va1 is: va1.size = 10.

The operand valarray va2(12) is: ( 0 1 2 3 4 5 6 7 8 9 ).
The number of elements in the valarray va2 is: va2.size = 12.

After initializing two more elements,
 the operand valarray va2(12) is now: ( 0 1 2 3 4 5 6 7 8 9 10 11 ).
The number of elements in the valarray va2 is still: 12.
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::sum

Determines the sum of all the elements in a valarray of nonzero length.

```
Type sum( ) const;
```

Return Value

The sum of the elements of the operand valarray.

Remarks

If the length is greater than one, the member function adds values to the sum by applying **operator+=** between pairs of elements of class **Type**, which operator, consequently, needs be provided for elements of type **Type**.

Example

```
// valarray_sum.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
   int sumva = 0;

   valarray<int> va ( 10 );
   for ( i = 0 ; i < 10 ; i+=1 )
      va [ i ] =  i;

   cout << "The operand valarray va (10) is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << va [ i ] << " ";
   cout << "))." << endl;

   sumva = va.sum ( );
   cout << "The sum of elements in the valarray is: "
        << sumva  << "." <<endl;
}
```

Output

```
The operand valarray va (10) is: ( 0 1 2 3 4 5 6 7 8 9 ).
The sum of elements in the valarray is: 45.
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::valarray

Constructs a valarray of a specific size or with elements of a specific value or as a copy of another valarray or subset of another valarray.

```
valarray( );
explicit valarray(
   size_t _Count
);
valarray(
   const Type& _Val,
   size_t _Count
);
valarray(
   const Type* _Ptr,
   size_t _Count
);
valarray(
   const slice_array<Type>& _SliceArray
);
valarray(
   const gslice_array<Type>& _GsliceArray
);
valarray(
   const mask_array<Type>& _MaskArray
);
valarray(
   const indirect_array<Type>& _IndArray
);
```

Parameters

_Count

The number of elements to be in the valarray.

_Val

The value to be used in initializing the elements in the valarray.

_Ptr

Pointer to the values to be used to initialize the elements in the valarray.

_SliceArray

A slice_array whose element values are to be used in initializing the elements of the valarray being constructed.

_GsliceArray

A gslice_array whose element values are to be used in initializing the elements of the valarray being constructed.

_MaskArray

A mask_array whose element values are to be used in initializing the elements of the valarray being constructed.

_IndArray

A indirect_array whose element values are to be used in initializing the elements of the valarray being constructed.

Return Value

The first (default) constructor initializes the object to an empty array. The next three constructors each initialize the object to an array of count elements as follows:

- For explicit valarray(**size_t** *_Count*), each element is initialized with the default constructor.
- For valarray(const **Type&** *_Val*, *_Count*), each element is initialized with *_Val*.

- For valarray(const **Type*** *_Ptr*, *_Count*), the element at position *I* is initialized with *_Ptr*[*I*].

Each remaining constructor initializes the object to a **valarray<Type>** object determined by the subset specified in the argument.

Example

```
// valarray_ctor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   // The second member function
   valarray<int> va ( 10 );
   for ( i = 0 ; i < 10 ; i += 1 )
      va [ i ] =  2 * ( i + 1 );

   cout << "The operand valarray va is:\n(";
   for ( i = 0 ; i < 10 ; i++ )
      cout << " " << va [ i ];
   cout << " )" << endl;

   slice Slice ( 2 , 4 , 3 );

   // The fifth member function
   valarray<int> vaSlice = va [ Slice ];

   cout << "The new valarray initialized from the slice is vaSlice ="
        << "\nva[slice( 2, 4, 3)] = (";
   for ( i = 0 ; i < 3 ; i++ )
      cout << " " << vaSlice [ i ];
   cout << " )" << endl;
}
```

Output

```
The operand valarray va is:
( 2 4 6 8 10 12 14 16 18 20 )
The new valarray initialized from the slice is vaSlice =
va[slice( 2, 4, 3)] = ( 6 12 18 )
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray Operators

For information about the operators in the valarray class, see valarray Class.

valarray::operator!

A unary operator that obtains the logical **NOT** values of each element in a valarray.

```
valarray<bool> operator!( ) const;
```

Return Value

The valarray of Boolean values that are the negation of the element values of the operand valarray.

Remarks

The logical operation **NOT** negates the elements because it converts all zeros into ones and regards all nonzero values as ones and converts them into zeros. The returned valarray of Boolean values is of the same size as the operand valarray.

There is also a bitwise **NOT** valarray::operator~ that negates on the level of individual bits within the binary representation of **char** and **int** elements of a valarray.

Example

```
// valarray_op_lognot.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 );
   valarray<bool> vaNOT ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i-1;

   cout << "The initial valarray is:  ( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;

   vaNOT = !vaL;
   cout << "The element-by-element result of "
        << "the logical NOT operator! is the\n valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaNOT [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial valarray is:  ( 0 0 0 2 0 4 0 6 0 8 ).
The element-by-element result of the logical NOT operator! is the
 valarray: ( 1 1 1 0 1 0 1 0 1 0 ).
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::operator%=

Obtains the remainder of dividing the elements of an array element-wise either by a specified valarray or by a value of the element type.

```
valarray<Type>& operator%=(
   const valarray<Type>& _Right
);
valarray<Type>& operator%=(
   const Type& _Right
);
```

Parameters

_Right

The valarray or value of an element type identical to that of the operand valarray that is to divide, element-wise, the operand valarray.

Return Value

A valarray whose elements are the remainder from the element-wise division of the operand valarray by *_Right*.

Example

```
// valarray_class_op_rem.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 6 ), vaR ( 6 );
   for ( i = 0 ; i < 6 ; i += 2 )
      vaL [ i ] =  53;
   for ( i = 1 ; i < 6 ; i += 2 )
      vaL [ i ] =  -67;
   for ( i = 0 ; i < 6 ; i++ )
      vaR [ i ] =  3*i+1;

   cout << "The initial valarray is: ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;

   cout << "The initial _Right valarray is: ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaR [ i ] << " ";
   cout << "))." << endl;

   vaL %= vaR;
   cout << "The remainders from the element-by-element "
        << "division is the\n valarray: ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial valarray is: ( 53 -67 53 -67 53 -67 ).
The initial _Right valarray is: ( 1 4 7 10 13 16 ).
The remainders from the element-by-element division is the
```

```
valarray: ( 0 -3 4 -7 1 -3 ).
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::operator&=

Obtains the bitwise **AND** of elements in an array either with the corresponding elements in a specified valarray or with a value of the element type.

```
valarray<Type>& operator&=(
   const valarray<Type>& _Right
);
valarray<Type>& operator&=(
   const Type& _Right
);
```

Parameters

_Right

The valarray or value of an element type identical to that of the operand valarray that is to be combined, element-wise, by the logical **AND** with the operand valarray.

Return Value

A valarray whose elements are the element-wise logical **AND** of the operand valarray by *_Right*.

Remarks

A bitwise operation can only be used to manipulate bits in **char** and **int** data types and variants and not on **float**, **double**, **long double**, **void**, **bool**, or other, more complex data types.

The bitwise AND has the same truth table as the logical **AND** but applies to the data type on the level of the individual bits. Given bits *b*1 and *b*2, *b*1 **AND** *b*2 is **true** if both bits are true; **false** if at least one is false.

Example

```
// valarray_class_op_bitand.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i-1;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial valarray is:  ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaL &= vaR;
   cout << "The element-by-element result of "
        << "the logical AND operator&= is the\n valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
```

```
    cout << "))." << endl;
  }
```

Output

```
The initial valarray is:  ( 0 0 0 2 0 4 0 6 0 8 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The element-by-element result of the logical AND operator&= is the
 valarray: ( 0 0 0 2 0 4 0 6 0 8 ).
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::operator>>=

Right-shifts the bits for each element of a valarray operand a specified number of positions or by an element-wise amount specified by a second valarray.

```
valarray<Type>& operator>>=(
   const valarray<Type>& _Right
);
valarray<Type>& operator>>=(
   const Type& _Right
);
```

Parameters

_Right

The value indicating the amount of right shift or valarray whose elements indicate the element-wise amount of right shift.

Return Value

A valarray whose elements have been shifted right the amount specified in *_Right*.

Remarks

Signed numbers have their signs preserved.

Example

```
// valarray_class_op_rs.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  64;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  -64;
   for ( i = 0 ; i < 8 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial operand valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;

   cout << "The _Right valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaR [ i ] << " ";
   cout << "))." << endl;

   vaL >>= vaR;
   cout << "The element-by-element result of "
        << "the right shift is the\n valarray: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial operand valarray is: ( 64 -64 64 -64 64 -64 64 -64 ).
The _Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the right shift is the
 valarray: ( 64 -32 16 -8 4 -2 1 -1 ).
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::operator<<=

Left-shifts the bits for each element of a valarray operand a specified number of positions or by an element-wise amount specified by a second valarray.

```
valarray<Type>& operator<<=(
   const valarray<Type>& _Right
);
valarray<Type>& operator<<=(
   const Type& _Right
);
```

Parameters

_Right

The value indicating the amount of left shift or valarray whose elements indicate the element-wise amount of left shift.

Return Value

A valarray whose elements have been shifted left the amount specified in *_Right*.

Remarks

Signed numbers have their signs preserved.

Example

```
// valarray_class_op_ls.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  1;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  -1;
   for ( i = 0 ; i < 8 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial operand valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;

   cout << "The _Right valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaR [ i ] << " ";
   cout << "))." << endl;

   vaL <<= vaR;
   cout << "The element-by-element result of "
        << "the left shift\n on the operand array is the valarray:\n ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial operand valarray is: ( 1 -1 1 -1 1 -1 1 -1 ).
The _Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the left shift
 on the operand array is the valarray:
 ( 1 -2 4 -8 16 -32 64 -128 ).
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::operator*=

Multiplies the elements of a specified valarray or a value of the element type, element-wise, to an operand valarray.

```
valarray<Type>& operator*=(
   const valarray<Type>& _Right
);
valarray<Type>& operator*=(
   const Type& _Right
);
```

Parameters

_Right

The valarray or value of an element type identical to that of the operand valarray that is to multiply, element-wise, the operand valarray.

Return Value

A valarray whose elements are the element-wise product of the operand valarray and *_Right*.

Example

```
// valarray_op_emult.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  2;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  -1;
   for ( i = 0 ; i < 8 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaR [ i ] << " ";
   cout << "))." << endl;

   vaL *= vaR;
   cout << "The element-by-element result of "
        << "the multiplication is the\n valarray: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial valarray is: ( 2 -1 2 -1 2 -1 2 -1 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the multiplication is the
 valarray: ( 0 -1 4 -3 8 -5 12 -7 ).
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::operator+

A unary operator that applies a plus to each element in a valarray.

```
valarray<Type> operator+( ) const;
```

Return Value

A valarray whose elements are plus those of the operand array.

Example

```
// valarray_op_eplus.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 );
   valarray<int> vaPLUS ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i-1;

   cout << "The initial valarray is:  ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;

   vaPLUS = +vaL;
   cout << "The element-by-element result of "
        << "the operator+ is the\n valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaPLUS [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial valarray is:  ( 0 0 -2 2 -4 4 -6 6 -8 8 ).
The element-by-element result of the operator+ is the
 valarray: ( 0 0 -2 2 -4 4 -6 6 -8 8 ).
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::operator+=

Adds the elements of a specified valarray or a value of the element type, element-wise, to an operand valarray.

```
valarray<Type>& operator+=(
   const valarray<Type>& _Right
);
valarray<Type>& operator+=(
   const Type& _Right
);
```

Parameters

_Right

The valarray or value of an element type identical to that of the operand valarray that is to be added, element-wise, to the operand valarray.

Return Value

A valarray whose elements are the element-wise sum of the operand valarray and *_Right*.

Example

```
// valarray_op_eadd.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  2;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  -1;
   for ( i = 0 ; i < 8 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial valarray is: ( ";
      for (i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;

   cout << "The initial _Right valarray is: ( ";
      for (i = 0 ; i < 8 ; i++ )
         cout << vaR [ i ] << " ";
   cout << "))." << endl;

   vaL += vaR;
   cout << "The element-by-element result of "
        << "the sum is the\n valarray: ( ";
      for (i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial valarray is: ( 2 -1 2 -1 2 -1 2 -1 ).
The initial _Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the sum is the
 valarray: ( 2 0 4 2 6 4 8 6 ).
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::operator-

A unary operator that applies a minus to each element in a valarray.

```
valarray<Type> operator-( ) const;
```

Return Value

A valarray whose elements are minus those of the operand array.

Example

```
// valarray_op_eminus.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 );
   valarray<int> vaMINUS ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i-1;

   cout << "The initial valarray is:  ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;

   vaMINUS = -vaL;
   cout << "The element-by-element result of "
        << "the operator+ is the\n valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaMINUS [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial valarray is:  ( 0 0 -2 2 -4 4 -6 6 -8 8 ).
The element-by-element result of the operator+ is the
 valarray: ( 0 0 2 -2 4 -4 6 -6 8 -8 ).
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::operator-=

Subtracts the elements of a specified valarray or a value of the element type, element-wise, from an operand valarray.

```
valarray<Type>& operator-=(
   const valarray<Type>& _Right
);
valarray<Type>& operator-=(
   const Type& _Right
);
```

Parameters

_Right

The valarray or value of an element type identical to that of the operand valarray that is to be subtracted, element-wise, from the operand valarray.

Return Value

A valarray whose elements are the element-wise difference of the operand valarray and *_Right*.

Example

```
// valarray_op_esub.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  10;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 0 ; i < 8 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial valarray is: ( ";
      for (i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;

   cout << "The initial _Right valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaR [ i ] << " ";
   cout << "))." << endl;

   vaL -= vaR;
   cout << "The element-by-element result of "
        << "the difference is the\n valarray: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial valarray is: ( 10 0 10 0 10 0 10 0 ).
The initial _Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the difference is the
 valarray: ( 10 -1 8 -3 6 -5 4 -7 ).
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::operator/=

Divides an operand valarray element-wise by the elements of a specified valarray or a value of the element type.

```
valarray<Type>& operator/=(
   const valarray<Type>& _Right
);
valarray<Type>& operator/=(
   const Type& _Right
);
```

Parameters

_Right

The valarray or value of an element type identical to that of the operand valarray that is to be divided, element-wise, into the operand valarray.

Return Value

A valarray whose elements are the element-wise quotient of the operand valarray divided by *_Right*.

Example

```
// valarray_op_ediv.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<double> vaL ( 6 ), vaR ( 6 );
   for ( i = 0 ; i < 6 ; i += 2 )
      vaL [ i ] =  100;
   for ( i = 1 ; i < 6 ; i += 2 )
      vaL [ i ] =  -100;
   for ( i = 0 ; i < 6 ; i++ )
      vaR [ i ] =  2*i;

   cout << "The initial valarray is: ( ";
      for (i = 0 ; i < 6 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;

   cout << "The initial Right valarray is: ( ";
      for (i = 0 ; i < 6 ; i++ )
         cout << vaR [ i ] << " ";
   cout << "))." << endl;

   vaL /= vaR;
   cout << "The element-by-element result of "
        << "the quotient is the\n valarray: ( ";
      for (i = 0 ; i < 6 ; i++ )
         cout << vaL [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial valarray is: ( 100 -100 100 -100 100 -100 ).
The initial Right valarray is: ( 0 2 4 6 8 10 ).
The element-by-element result of the quotient is the
 valarray: ( 1.#INF -50 25 -16.6667 12.5 -10 ).
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::operator=

Assigns elements to a valarray whose values are specified either directly or as part of some other valarray or by a slice_array, gslice_array, mask_array, or indirect_array.

```
valarray<Type>& operator=(
   const valarray<Type>& _Right
);
valarray<Type>& operator=(
   const Type& _Val
);
valarray<Type>& operator=(
   const slice_array<Type>& _Slicearray
);
valarray<Type>& operator=(
   const gslice_array<Type>& _Gslicearray
);
valarray<Type>& operator=(
   const mask_array<Type>& _Maskarray
);
valarray<Type>& operator=(
   const indirect_array<Type>& _Indarray
);
```

Parameters

_Right

The valarray to be copied into the operand valarray.

_Val

The value to be assigned to the elements of the operand valarray.

_Slicearray

The slice_array to be copied into the operand valarray.

_Gslicearray

The gslice_array to be copied into the operand valarray.

_Maskarray

The mask_array to be copied into the operand valarray.

_Indarray

The indirect_array to be copied into the operand valarray.

Return Value

The first member operator replaces the controlled sequence with a copy of the sequence controlled by *_Right*.

The second member operator replaces each element of the controlled sequence with a copy of *_Val*.

The remaining member operators replace those elements of the controlled sequence selected by their arguments, which are generated only by operator[].

If the value of a member in the replacement controlled sequence depends on a member in the initial controlled sequence, the result is undefined.

Remarks

If the length of the controlled sequence changes, the result is generally undefined. In this implementation, however, the effect is merely to invalidate any pointers or references to elements in the controlled sequence.

Example

```
// valarray_op_assign.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> va ( 10 ), vaR ( 10 );
   for ( i = 0 ; i < 10 ; i += 1 )
      va [ i ] = i;
   for ( i = 0 ; i < 10 ; i+=1 )
      vaR [ i ] = 10 -  i;

   cout << "The operand valarray va is:";
   for ( i = 0 ; i < 10 ; i++ )
      cout << " " << va [ i ];
   cout << endl;

   cout << "The operand valarray vaR is:";
      for ( i = 0 ; i < 10 ; i++ )
         cout << " " << vaR [ i ];
   cout << endl;

   // Assigning vaR to va with the first member functon
   va = vaR;
   cout << "The reassigned valarray va is:";
   for ( i = 0 ; i < 10 ; i++ )
      cout << " " << va [ i ];
   cout << endl;

   // Assigning elements of value 10 to va
   // with the second member functon
   va = 10;
   cout << "The reassigned valarray va is:";
      for ( i = 0 ; i < 10 ; i++ )
         cout << " " << va [ i ];
   cout << endl;
}
```

Output

```
The operand valarray va is: 0 1 2 3 4 5 6 7 8 9
The operand valarray vaR is: 10 9 8 7 6 5 4 3 2 1
The reassigned valarray va is: 10 9 8 7 6 5 4 3 2 1
The reassigned valarray va is: 10 10 10 10 10 10 10 10 10 10
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::operator[]

Returns a reference to an element or its value at specified index or a specified subset.

```
Type& operator[](
   size_t _Off
);
slice_array<Type> operator[](
   slice _Slicearray
);
gslice_array<Type> operator[](
   const gslice& _Gslicearray
);
mask_array<Type> operator[](
   const valarray<bool>& _Boolarray
);
indirect_array<Type> operator[](
   const valarray<size_t>& _Indarray
);
Type operator[](
   size_t _Off
) const;
valarray<Type> operator[](
      _Slice
) const;
valarray<Type> operator[](
   const gslice& _Gslicearray
) const;
valarray<Type> operator[](
   const valarray<bool>& _Boolarray
) const;
valarray<Type> operator[](
   const valarray<size_t>& _Indarray
) const;
```

Parameters

_Off

The index of the element to be assigned a value.

_Slicearray

A slice_array of a valarray that specifies a subset to be selected or returned to a new valarray.

_Gslicearray

A gslice_array of a valarray that specifies a subset to be selected or returned to a new valarray.

_Boolarray

A bool_array of a valarray that specifies a subset to be selected or returned to a new valarray.

_Indarray

An indirect_array of a valarray that specifies a subset to be selected or returned to a new valarray.

Return Value

A reference to an element or its value at specified index or a specified subset.

Remarks

The member operator is overloaded to provide several ways to select sequences of elements from among those controlled by ***this**. The first group of five member operators work in conjunction with various overloads of operator= (and other assigning operators) to allow selective replacement (slicing) of the controlled sequence. The selected elements must exist.

When compiling with _SECURE_SCL 1, a runtime error will occur if you attempt to access an element outside the bounds of the

valarray. See Checked Iterators for more information.

Example

See the examples for slice::slice and gslice::gslice for an example of how to declare and use the operator.

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::operator^=

Obtains the element-wise exclusive logical or operator (**XOR**) of an array with either a specified valarray or a value of the element type.

```
valarray<Type>& operator|=(
   const valarray<Type>& _Right
);
valarray<Type>& operator|=(
   const Type& _Right
);
```

Parameters

_Right

The valarray or value of an element type identical to that of the operand valarray that is to be combined, element-wise, by the exclusive logical **XOR** with the operand valarray.

Return Value

A valarray whose elements are the element-wise, exclusive logical **XOR** of the operand valarray and *_Right*.

Remarks

The exclusive logical or, referred to as **XOR**, has the following semantics: Given elements *e*1 and *e*2, *e*1 **XOR** *e*2 is **true** if exactly one of the elements is true; **false** if both elements are false or if both elements are true.

Example

```
// valarray_op_exor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  1;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 0 ; i < 10 ; i += 3 )
      vaR [ i ] =  i;
   for ( i = 1 ; i < 10 ; i += 3 )
      vaR [ i ] =  i-1;
   for ( i = 2 ; i < 10 ; i += 3 )
      vaR [ i ] =  i-1;

   cout << "The initial operand valarray is:  ( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The _Right valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaL ^= vaR;
   cout << "The element-by-element result of "
        << "the bitwise XOR operator^= is the\n valarray: ( ";
      for (i = 0 ; i < 10 ; i++ )
```

```
            cout << vaL [ i ] << " ";
    cout << ")." << endl;
  }
```

Output

```
The initial operand valarray is:  ( 1 0 1 0 1 0 1 0 1 0 ).
The _Right valarray is: ( 0 0 1 3 3 4 6 6 7 9 ).
The element-by-element result of the bitwise XOR operator^= is the
 valarray: ( 1 0 0 3 2 4 7 6 6 9 ).
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::operator|=

Obtains the bitwise **OR** of elements in an array either with the corresponding elements in a specified valarray or with a value of the element type.

```
valarray<Type>& operator|=(
   const valarray<Type>& _Right
);
valarray<Type>& operator|=(
   const Type& _Right
);
```

Parameters

_Right

The valarray or value of an element type identical to that of the operand valarray that is to be combined, element-wise, by the bitwise **OR** with the operand valarray.

Return Value

A valarray whose elements are the element-wise bitwise **OR** of the operand valarray by *_Right*.

Remarks

A bitwise operation can only be used to manipulate bits in **char** and **int** data types and variants and not on **float**, **double**, **long double**, **void**, **bool**, or other, more complex data types.

The bitwise **OR** has the same truth table as the logical **OR** but applies to the data type on the level of the individual bits. Given bits *b*1 and *b*2, *b*1 **OR** *b*2 is **true** if at least one of the bits is true; **false** if both bits are false.

Example

```
// valarray_class_op_bitor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  1;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 0 ; i < 10 ; i += 3 )
      vaR [ i ] =  i;
   for ( i = 1 ; i < 10 ; i += 3 )
      vaR [ i ] =  i-1;
   for ( i = 2 ; i < 10 ; i += 3 )
      vaR [ i ] =  i-1;

   cout << "The initial operand valarray is:\n ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The _Right valarray is:\n ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaL |= vaR;
```

```
    cout << "The element-by-element result of "
         << "the logical OR\n operator|= is the valarray:\n ( ";
       for (i = 0 ; i < 10 ; i++ )
          cout << vaL [ i ] << " ";
    cout << "))." << endl;
  }
```

Output

```
The initial operand valarray is:
 ( 1 0 1 0 1 0 1 0 1 0 ).
The _Right valarray is:
 ( 0 0 1 3 3 4 6 6 7 9 ).
The element-by-element result of the logical OR
 operator|= is the valarray:
 ( 1 0 1 3 3 4 7 6 7 9 ).
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

valarray::operator~

A unary operator that obtains the bitwise **NOT** values of each element in a valarray.

```
valarray<Type> operator~( ) const;
```

Return Value

The valarray of Boolean values that are the bitwise **NOT** of the element values of the operand valarray.

Remarks

A bitwise operation can only be used to manipulate bits in **char** and **int** data types and variants and not on **float**, **double**, **long double**, **void**, **bool** or other, more complex data types.

The bitwise **NOT** has the same truth table as the logical **NOT** but applies to the data type on the level of the individual bits. Given bit *b*, ~*b* is true if *b* is false and false if *b* is true. The logical **NOT** operator! applies on an element level, counting all nonzero values as **true**, and the result is a valarray of Boolean values. The bitwise **NOT operator~**, by contrast, can result in a valarray of values other than 0 or 1, depending on outcome of the bitwise operation.

Example

```
// valarray_op_bitnot.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<unsigned short int> vaL1 ( 10 );
   valarray<unsigned short int> vaNOT1 ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL1 [ i ] =  i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL1 [ i ] =  5*i;

   cout << "The initial valarray <unsigned short int> is:  ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL1 [ i ] << " ";
   cout << ")." << endl;

   vaNOT1 = ~vaL1;
   cout << "The element-by-element result of "
        << "the bitwise NOT operator~ is the\n valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaNOT1 [ i ] << " ";
   cout << ")." << endl << endl;

   valarray<int> vaL2 ( 10 );
   valarray<int> vaNOT2 ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL2 [ i ] =  i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL2 [ i ] =  -2 * i;

   cout << "The initial valarray <int> is:  ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL2 [ i ] << " ";
   cout << ")." << endl;

   vaNOT2 = ~vaL2;
   cout << "The element-by-element result of "
```

```
         << "the bitwise NOT operator~ is the\n valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaNOT2 [ i ] << " ";
   cout << ")." << endl;

   // The negative numbers are represented using
   // the two's complement approach, so adding one
   // to the flipped bits returns the negative elements
   vaNOT2 = vaNOT2 + 1;
   cout << "The element-by-element result of "
        << "adding one\n is the negative of the "
        << "original elements the\n valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaNOT2 [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial valarray <unsigned short int> is:  ( 0 5 2 15 4 25 6 35 8 45 ).
The element-by-element result of the bitwise NOT operator~ is the
 valarray: ( 65535 65530 65533 65520 65531 65510 65529 65500 65527 65490 ).

The initial valarray <int> is:  ( 0 -2 2 -6 4 -10 6 -14 8 -18 ).
The element-by-element result of the bitwise NOT operator~ is the
 valarray: ( -1 1 -3 5 -5 9 -7 13 -9 17 ).
The element-by-element result of adding one
 is the negative of the original elements the
 valarray: ( 0 2 -2 6 -4 10 -6 14 -8 18 ).
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Other Resources

valarray Members

<valarray> Specializations

For information about the specializations in <valarray>, see <valarray> Members.

valarray<bool> Class

A specialized version of the template class **valarray<Type>** to elements of type **bool**.

```
class valarray<bool>
```

Example

```
// valarray_bool.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<bool> vaBool ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial Left valarray is: ( ";
   for ( i = 0 ; i < 10 ; i++ )
      cout << vaL [ i ] << " ";
   cout << "))." << endl;

   cout << "The initial Right valarray is: ( ";
   for ( i = 0 ; i < 10 ; i++ )
      cout << vaR [ i ] << " ";
   cout << "))." << endl;

   vaBool = ( vaL < vaR );
   cout << "The result of the less-than comparison "
   << "test is the\n valarray<bool>: ( ";
   for ( i = 0 ; i < 10 ; i++ )
      cout << vaBool [ i ] << " ";
   cout << "))." << endl;
}
```

Output

```
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The result of the less-than comparison test is the
 valarray<bool>: ( 0 0 1 0 1 0 1 0 1 0 ).
```

Requirements

Header: <valarray>

See Also

Reference

valarray Class

Thread Safety in the Standard C++ Library

Other Resources

valarray Members

<vector>

Defines the container template class vector and several supporting templates.

For a list of all members of this header, see <vector> Members.

```
#include <vector>
```

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

<vector> Members

Standard C++ Library Header Files

<vector> Members

Reference

Operators

operator! =	Tests if the vector object on the left side of the operator is not equal to the vector object on the right side.
operator<	Tests if the vector object on the left side of the operator is less than the vector object on the right side.
operator<=	Tests if the vector object on the left side of the operator is less than or equal to the vector object on the right side.
operator==	Tests if the vector object on the left side of the operator is equal to the vector object on the right side.
operator>	Tests if the vector object on the left side of the operator is greater than the vector object on the right side.
operator>=	Tests if the vector object on the left side of the operator is greater than or equal to the vector object on the right side.

Classes

vector Class	A template class of sequence containers that arrange elements of a given type in a linear arrangement and allow fast random access to any element.

Specializations

vector<bool> Class	A full specialization of the template class vector for elements of type **bool** with an allocator for the underlying type used by the specialization.

See Also

Reference

<vector>

Thread Safety in the Standard C++ Library

Standard Template Library

<vector> Operators

For information about the operators in <vector>, see <vector> Members.

See Also

Reference

Standard Template Library

operator!= (<vector>)

Tests if the object on the left side of the operator is not equal to the object on the right side.

```
bool operator!=(
   const vector<Type, Allocator>& _Left,
   const vector<Type, Allocator>& _Right
);
```

Parameters

_Left

An object of type **vector**.

_Right

An object of type **vector**.

Return Value

true if the vectors are not equal; **false** if the vectors are equal.

Remarks

Two vectors are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// vector_op_ne.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   vector <int> v1, v2;
   v1.push_back( 1 );
     v2.push_back( 2 );

   if ( v1 != v2 )
      cout << "Vectors not equal." << endl;
   else
      cout << "Vectors equal." << endl;
}
```

Output

```
Vectors not equal.
```

Requirements

Header: <vector>

See Also

Reference

Standard Template Library

Other Resources

<vector> Members

operator< (<vector>)

Tests if the object on the left side of the operator is less than the object on the right side.

```
bool operator<(
   const vector<Type, Allocator>& _Left,
   const vector<Type, Allocator>& _Right
);
```

Parameters

_Left

An object of type **vector**.

_Right

An object of type **vector**.

Return Value

true if the vector on the left side of the operator is less than the vector on the right side of the operator; otherwise **false**.

Example

```
// vector_op_lt.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   vector <int> v1, v2;
   v1.push_back( 1 );
   v1.push_back( 2 );
   v1.push_back( 4 );

   v2.push_back( 1 );
   v2.push_back( 3 );

   if ( v1 < v2 )
      cout << "Vector v1 is less than vector v2." << endl;
   else
      cout << "Vector v1 is not less than vector v2." << endl;
}
```

Output

```
Vector v1 is less than vector v2.
```

Requirements

Header: <vector>

See Also

Reference

vector::operator<

Standard Template Library

Other Resources

<vector> Members

operator<= (<vector>)

Tests if the object on the left side of the operator is less than or equal to the object on the right side.

```
bool operator<=(
   const vector<Type, Allocator>& _Left,
   const vector<Type, Allocator>& _Right
);
```

Parameters

_Left

An object of type **vector**.

_Right

An object of type **vector**.

Return Value

true if the vector on the left side of the operator is less than or equal to the vector on the right side of the operator; otherwise **false**.

Example

```
// vector_op_le.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   vector <int> v1, v2;
   v1.push_back( 1 );
   v1.push_back( 2 );
   v1.push_back( 4 );

   v2.push_back( 1 );
   v2.push_back( 3 );

   if ( v1 <= v2 )
      cout << "Vector v1 is less than or equal to vector v2." << endl;
   else
      cout << "Vector v1 is greater than vector v2." << endl;
}
```

Output

```
Vector v1 is less than or equal to vector v2.
```

Requirements

Header: <vector>

See Also

Reference

Standard Template Library

Other Resources

<vector> Members

operator== (<vector>)

Tests if the object on the left side of the operator is equal to the object on the right side.

```
bool operator==(
   const vector<Type, Allocator>& _Left,
   const vector<Type, Allocator>& _Right
);
```

Parameters

_Left

An object of type **vector**.

_Right

An object of type **vector**.

Return Value

true if the vector on the left side of the operator is equal to the vector on the right side of the operator; otherwise **false**.

Remarks

Two vectors are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// vector_op_eq.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   vector <int> v1, v2;
   v1.push_back( 1 );
   v2.push_back( 1 );

   if ( v1 == v2 )
      cout << "Vectors equal." << endl;
   else
      cout << "Vectors not equal." << endl;
}
```

Output

```
Vectors equal.
```

Requirements

Header: <vector>

See Also

Reference

vector::operator==

Standard Template Library

Other Resources

<vector> Members

operator> (<vector>)

Tests if the object on the left side of the operator is greater than the object on the right side.

```
bool operator>(
   const vector<Type, Allocator>& _Left,
   const vector<Type, Allocator>& _Right
);
```

Parameters

_Left

An object of type **vector**.

_Right

An object of type **vector**.

Return Value

true if the vector on the left side of the operator is greater than the vector on the right side of the operator; otherwise **false**.

Example

```
// vector_op_gt.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   vector <int> v1, v2;
   v1.push_back( 1 );
   v1.push_back( 3 );
   v1.push_back( 1 );

   v2.push_back( 1 );
   v2.push_back( 2 );
   v2.push_back( 2 );

   if ( v1 > v2 )
      cout << "Vector v1 is greater than vector v2." << endl;
   else
      cout << "Vector v1 is not greater than vector v2." << endl;
}
```

Output

```
Vector v1 is greater than vector v2.
```

Requirements

Header: <vector>

See Also

Reference

Standard Template Library

Other Resources

<vector> Members

operator>= (<vector>)

Tests if the object on the left side of the operator is greater than or equal to the object on the right side.

```
bool operator>=(
   const vector<Type, Allocator>& _Left,
   const vector<Type, Allocator>& _Right
);
```

Parameters

_Left

An object of type **vector**.

_Right

An object of type **vector**.

Return Value

true if the vector on the left side of the operator is greater than or equal to the vector on the right side of the vector; otherwise **false**.

Example

```
// vector_op_ge.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   vector <int> v1, v2;
   v1.push_back( 1 );
   v1.push_back( 3 );
   v1.push_back( 1 );

     v2.push_back( 1 );
   v2.push_back( 2 );
   v2.push_back( 2 );

   if ( v1 >= v2 )
      cout << "Vector v1 is greater than or equal to vector v2." << endl;
   else
      cout << "Vector v1 is less than vector v2." << endl;
}
```

Output

```
Vector v1 is greater than or equal to vector v2.
```

Requirements

Header: <vector>

See Also

Reference

Standard Template Library

Other Resources

<vector> Members

swap (<vector>)

Exchanges the elements of two vectors.

```
template<class Type, class Allocator>
void swap(
   vector<Type, Allocator>& _Left,
   vector<Type, Allocator>& _Right
);
```

Parameters

_Right

The vector providing the elements to be swapped, or the vector whose elements are to be exchanged with those of the vector *_Left*.

_Left

The vector whose elements are to be exchanged with those of the vector *_Right*.

Remarks

The template function is an algorithm specialized on the container class vector to execute the member function *_Left*.vector::swap(*_Right*). These are instances of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function, **template** <**class T**> **void swap**(**T&**, **T&**), in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

Example

See the code example for member function vector::swap for an example that uses the template version of **swap**.

Requirements

Header: <vector>

See Also

Reference

Standard Template Library

Other Resources

<vector> Members

<vector> Classes

For information about the <vector> header file, see <vector> Members.

See Also

Reference

Standard Template Library

vector Class

The STL vector class is a template class of sequence containers that arrange elements of a given type in a linear arrangement and allow fast random access to any element. They should be the preferred container for a sequence when random-access performance is at a premium.

For a list of all members of this type, see vector Members.

```
template <
   class Type,
   class Allocator = allocator<Type>
>
class vector
```

Parameters

Type

The element data type to be stored in the vector

Allocator

The type that represents the stored allocator object that encapsulates details about the vector's allocation and deallocation of memory. This argument is optional and the default value is **allocator**<*Type*>.

Remarks

Vectors allow constant time insertions and deletions at the end of the sequence. Inserting or deleting elements in the middle of a vector requires linear time. The performance of the deque Class container is superior with respect to insertions and deletions at the beginning and end of a sequence. The list Class container is superior with respect to insertions and deletions at any location within a sequence.

Vector reallocation occurs when a member function must increase the sequence contained in the vector object beyond its current storage capacity. Other insertions and erasures may alter various storage addresses within the sequence. In all such cases, iterators or references that point at altered portions of the sequence become invalid. If no reallocation happens, only iterators and references before the insertion/deletion point remain valid.

The vector<bool> Class is a full specialization of the template class vector for elements of type bool with an allocator for the underlying type used by the specialization.

The vector<bool> reference Class is a nested class whose objects are able to provide references to elements (single bits) within a vector<bool> object.

Requirements

Header: <vector>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

vector Members

<vector> Members

vector Members

Reference

Constructors

vector	Constructs a vector of a specific size or with elements of a specific value or with a specific **allocator** or as a copy of some other vector.

Typedefs

allocator_type	A type that represents the **allocator** class for the vector object.
const_iterator	A type that provides a random-access iterator that can read a **const** element in a vector.
const_pointer	A type that provides a pointer to a **const** element in a vector.
const_reference	A type that provides a reference to a **const** element stored in a vector for reading and performing **const** operations.
const_reverse_iterator	A type that provides a random-access iterator that can read any **const** element in the vector.
difference_type	A type that provides the difference between the addresses of two elements in a vector.
iterator	A type that provides a random-access iterator that can read or modify any element in a vector.
pointer	A type that provides a pointer to an element in a vector.
reference	A type that provides a reference to an element stored in a vector.
reverse_iterator	A type that provides a random-access iterator that can read or modify any element in a reversed vector.
size_type	A type that counts the number of elements in a vector.
value_type	A type that represents the data type stored in a vector.

Member Functions

assign	Erases a vector and copies the specified elements to the empty vector.
at	Returns a reference to the element at a specified location in the vector.
back	Returns a reference to the last element of the vector.
begin	Returns a random-access iterator to the first element in the container.
capacity	Returns the number of elements that the vector could contain without allocating more storage.
clear	Erases the elements of the vector.
empty	Tests if the vector container is empty.
end	Returns a random-access iterator that points just beyond the end of the vector.
erase	Removes an element or a range of elements in a vector from specified positions.

front	Returns a reference to the first element in a vector.
get_allocator	Returns an object to the **allocator** class used by a vector.
insert	Inserts an element or a number of elements into the vector at a specified position.
max_size	Returns the maximum length of the vector.
pop_back	Deletes the element at the end of the vector.
push_back	Add an element to the end of the vector.
rbegin	Returns an iterator to the first element in a reversed vector.
rend	Returns an iterator to the end of a reversed vector.
resize	Specifies a new size for a vector.
reserve	Reserves a minimum length of storage for a vector object.
size	Returns the number of elements in the vector.
swap	Exchanges the elements of two vectors.

Operators

operator[]	Returns a reference to the vector element at a specified position.

See Also

Reference

vector Class

Thread Safety in the Standard C++ Library

Standard Template Library

vector Typedefs

For information about the typedefs in the vector class, see vector Members.

See Also

Reference

Standard Template Library

vector::allocator_type

A type that represents the allocator class for the vector object.

```
typedef Allocator allocator_type
```

Remarks

allocator_type is a synonym for the template parameter **Allocator.**

Example

See the example for get_allocator for an example that uses **allocator_type**.

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::const_iterator

A type that provides a random-access iterator that can read a **const** element in a vector.

```
typedef implementation-defined const_iterator;
```

Remarks

A type **const_iterator** cannot be used to modify the value of an element.

Example

See the example for back for an example that uses **const_iterator**.

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::const_pointer

A type that provides a pointer to a **const** element in a vector.

```
typedef typename Allocator::const_pointer const_pointer;
```

Remarks

A type **const_pointer** cannot be used to modify the value of an element.

An iterator is more commonly used to access a vector element.

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::const_reference

A type that provides a reference to a **const** element stored in a vector for reading and performing **const** operations.

```
typedef typename Allocator::const_reference const_reference;
```

Remarks

A type **const_reference** cannot be used to modify the value of an element.

Example

```
// vector_const_ref.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );
   v1.push_back( 20 );

   const vector <int> v2 = v1;
   const int &i = v2.front( );
   const int &j = v2.back( );
   cout << "The first element is " << i << endl;
   cout << "The second element is " << j << endl;

   // The following line would cause an error as v2 is const
   // v2.push_back( 30 );
}
```

Output

```
The first element is 10
The second element is 20
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::const_reverse_iterator

A type that provides a random-access iterator that can read any **const** element in the vector.

```
typedef reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type **const_reverse_iterator** cannot modify the value of an element and is used to iterate through the vector in reverse.

Example

See rbegin for an example of how to declare and use an iterator.

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::difference_type

A type that provides the difference between two iterators that refer to elements within the same vector.

```
typedef implementation-defined difference_type;
```

Remarks

A **difference_type** can also be described as the number of elements between two pointers, because a pointer to an element contains its address.

An iterator is more commonly used to access a vector element.

Example

```
// vector_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <vector>
#include <algorithm>

int main( )
{
   using namespace std;

   vector <int> c1;
   vector <int>::iterator c1_Iter, c2_Iter;

   c1.push_back( 30 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1.push_back( 10 );
   c1.push_back( 30 );
   c1.push_back( 20 );

   c1_Iter = c1.begin( );
   c2_Iter = c1.end( );

   vector <int>::difference_type   df_typ1, df_typ2, df_typ3;

   df_typ1 = count( c1_Iter, c2_Iter, 10 );
   df_typ2 = count( c1_Iter, c2_Iter, 20 );
   df_typ3 = count( c1_Iter, c2_Iter, 30 );
   cout << "The number '10' is in c1 collection " << df_typ1 << " times.\n";
   cout << "The number '20' is in c1 collection " << df_typ2 << " times.\n";
   cout << "The number '30' is in c1 collection " << df_typ3 << " times.\n";
}
```

Output

```
The number '10' is in c1 collection 1 times.
The number '20' is in c1 collection 2 times.
The number '30' is in c1 collection 3 times.
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::iterator

A type that provides a random-access iterator that can read or modify any element in a vector.

```
typedef implementation-defined iterator;
```

Remarks

A type **iterator** can be used to modify the value of an element.

Example

See the example for begin.

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::pointer

A type that provides a pointer to an element in a vector.

```
typedef typename Allocator::pointer pointer;
```

Remarks

A type **pointer** can be used to modify the value of an element.

Example

```
// vector_pointer.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector<int> v;
   v.push_back( 11 );
   v.push_back( 22 );

   vector<int>::pointer ptr = &v[0];
   cout << *ptr << endl;
   ptr++;
   cout << *ptr << endl;
   *ptr = 44;
   cout << *ptr << endl;
}
```

Output

```
11
22
44
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::reference

A type that provides a reference to an element stored in a vector.

```
typedef typename Allocator::reference reference;
```

Example

See at for an example of how to use **reference** in the vector class.

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::reverse_iterator

A type that provides a random-access iterator that can read or modify any element in a reversed vector.

```
typedef reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type **reverse_iterator** is used to iterate through the vector in reverse.

Example

See the example for rbegin.

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::size_type

A type that counts the number of elements in a vector.

```
typedef implementation defined size_type;
```

Example

See the example for capacity.

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::value_type

A type that represents the data type stored in a vector.

```
typedef Type value_type;
```

Remarks

value_type is a synonym for the template parameter **Type**.

Example

```
// vector_value_type.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector<int>::value_type AnInt;
   AnInt = 44;
   cout << AnInt << endl;
}
```

Output

```
44
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector Member Functions

For information about the functions in the vector class, see vector Members.

See Also

Reference

Standard Template Library

vector::assign

Erases a vector and copies the specified elements to the empty vector.

```
void assign(
   size_type _Count,
   const Type& _Val
);
template<class InputIterator>
      void assign(
      InputIterator _First,
      InputIterator _Last
   );
```

Parameters

_First

Position of the first element in the range of elements to be copied.

_Last

Position of the first element beyond the range of elements to be copied.

_Count

The number of copies of an element being inserted into the vector.

_Val

The value of the element being inserted into the vector.

Remarks

After erasing any existing elements in a vector, **assign** either inserts a specified range of elements from the original vector into a vector or inserts copies of a new element of a specified value into a vector.

Example

```
// vector_assign.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector<int> v1, v2, v3;
   vector<int>::iterator iter;

   v1.push_back(10);
   v1.push_back(20);
   v1.push_back(30);
   v1.push_back(40);
   v1.push_back(50);

   cout << "v1 = " ;
   for (iter = v1.begin(); iter != v1.end(); iter++)
      cout << *iter << " ";
   cout << endl;

   v2.assign(v1.begin(), v1.end());
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << *iter << " ";
   cout << endl;
```

```
    v3.assign(7, 4) ;
    cout << "v3 = ";
    for (iter = v3.begin(); iter != v3.end(); iter++)
       cout << *iter << " ";
    cout << endl;
  }
```

Output

```
v1 = 10 20 30 40 50
v2 = 10 20 30 40 50
v3 = 4 4 4 4 4 4 4
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::at

Returns a reference to the element at a specified location in the vector.

```
reference at(
   size_type _Pos
);
const_reference at(
   size_type _Pos
) const;
```

Parameters

_Pos

The subscript or position number of the element to reference in the vector.

Return Value

A reference to the element subscripted in the argument. If *_Off* is greater than the size of the vector, **at** throws an exception.

Remarks

If the return value of **at** is assigned to a **const_reference**, the vector object cannot be modified. If the return value of **at** is assigned to a **reference**, the vector object can be modified.

Example

```
// vector_at.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );
   v1.push_back( 20 );

   const int &i = v1.at( 0 );
   int &j = v1.at( 1 );
   cout << "The first element is " << i << endl;
   cout << "The second element is " << j << endl;
}
```

Output

```
The first element is 10
The second element is 20
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::back

Returns a reference to the last element of the vector.

```
reference back( );
const_reference back( ) const;
```

Return Value

The last element of the vector. If the vector is empty, the return value is undefined.

Remarks

If the return value of **back** is assigned to a **const_reference**, the vector object cannot be modified. If the return value of **back** is assigned to a **reference**, the vector object can be modified.

When compiling with _SECURE_SCL 1, a runtime error will occur if you attempt to access an element in an empty vector. See Checked Iterators for more information.

Example

```
// vector_back.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );
   v1.push_back( 11 );

   int& i = v1.back( );
   const int& ii = v1.front( );

   cout << "The last integer of v1 is " << i << endl;
   i--;
   cout << "The next-to-last integer of v1 is "<< ii << endl;
}
```

Output

```
The last integer of v1 is 11
The next-to-last integer of v1 is 10
```

Requirements

Header: <vector>

See Also

Reference

vector Class

vector::front and vector::back

Standard Template Library

Other Resources

vector Members

vector::begin

Returns a random-access iterator to the first element in the container.

```
const_iterator begin() const;
iterator begin();
```

Return Value

A random-access iterator addressing the first element in the **vector** or to the location succeeding an empty **vector**. You should always compare the value returned with vector::end to ensure it is valid.

Remarks

If the return value of **begin** is assigned to a vector::const_iterator, the **vector** object cannot be modified. If the return value of **begin** is assigned to an vector::iterator, the **vector** object can be modified.

Example

```
// vector_begin.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> c1;
    vector<int>::iterator c1_Iter;
    vector<int>::const_iterator c1_cIter;

    c1.push_back(1);
    c1.push_back(2);

    cout << "The vector c1 contains elements:";
    c1_Iter = c1.begin();
    for (; c1_Iter != c1.end(); c1_Iter++)
    {
        cout << " " << *c1_Iter;
    }
    cout << endl;

    cout << "The vector c1 now contains elements:";
    c1_Iter = c1.begin();
    *c1_Iter = 20;
    for (; c1_Iter != c1.end(); c1_Iter++)
    {
        cout << " " << *c1_Iter;
    }
    cout << endl;

    // The following line would be an error because iterator is const
    // *c1_cIter = 200;
}
```

Output

```
The vector c1 contains elements: 1 2
The vector c1 now contains elements: 20 2
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::capacity

Returns the number of elements that the vector could contain without allocating more storage.

```
size_type capacity( ) const;
```

Return Value

The current length of storage allocated for the vector.

Remarks

The member function resize will be more efficient if sufficient memory is allocated to accommodate it. Use the member function reserve to specify the amount of memory allocated.

Example

```
// vector_capacity.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 1 );
   cout << "The length of storage allocated is "
        << v1.capacity( ) << "." << endl;

   v1.push_back( 2 );
   cout << "The length of storage allocated is now "
        << v1.capacity( ) << "." << endl;
}
```

Output

```
The length of storage allocated is 1.
The length of storage allocated is now 2.
```

Requirements

Header: <vector>

See Also

Reference

vector Class

vector::size and vector::capacity

Standard Template Library

Other Resources

vector Members

vector::clear

Erases the elements of the vector.

```
void clear( );
```

Example

```
// vector_clear.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );
   v1.push_back( 20 );
   v1.push_back( 30 );

   cout << "The size of v1 is " << v1.size( ) << endl;
   v1.clear( );
   cout << "The size of v1 after clearing is " << v1.size( ) << endl;
}
```

Output

```
The size of v1 is 3
The size of v1 after clearing is 0
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::empty

Tests if the vector is empty.

```
bool empty( ) const;
```

Return Value

true if the vector is empty; **false** if the vector is not empty.

Example

```
// vector_empty.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );

   if ( v1.empty( ) )
      cout << "The vector is empty." << endl;
   else
      cout << "The vector is not empty." << endl;
}
```

Output

```
The vector is not empty.
```

Requirements

Header: <vector>

See Also

Reference

vector Class

vector::empty, vector::erase, and vector::push_back

Standard Template Library

Other Resources

vector Members

vector::end

Returns a random-access iterator that points just beyond the end of the vector.

```
iterator end( );
const_iterator end( ) const;
```

Return Value

A random-access iterator to the end of the vector object. If the vector is empty, vector::end == vector::begin.

Remarks

If the return value of **end** is assigned to a variable of type **const_iterator**, the vector object cannot be modified. If the return value of **end** is assigned to a variable of type **iterator**, the vector object can be modified.

Example

```
// vector_end.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>
int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator v1_Iter;

   v1.push_back( 1 );
   v1.push_back( 2 );

   for ( v1_Iter = v1.begin( ) ; v1_Iter != v1.end( ) ; v1_Iter++ )
      cout << *v1_Iter << endl;
}
```

Output

```
1
2
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::erase

Removes an element or a range of elements in a vector from specified positions.

```
iterator erase(
   iterator _Where
);
iterator erase(
   iterator _First,
   iterator _Last
);
```

Parameters

_Where

Position of the element to be removed from the vector.

_First

Position of the first element removed from the vector.

_Last

Position just beyond the last element removed from the vector.

Return Value

An iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the vector if no such element exists.

Example

```
// vector_erase.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter;

   v1.push_back( 10 );
   v1.push_back( 20 );
   v1.push_back( 30 );
   v1.push_back( 40 );
   v1.push_back( 50 );

   cout << "v1 =" ;
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;

   v1.erase( v1.begin( ) );
   cout << "v1 =";
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;

   v1.erase( v1.begin( ) + 1, v1.begin( ) + 3 );
   cout << "v1 =";
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;
}
```

Output

```
v1 = 10 20 30 40 50
v1 = 20 30 40 50
v1 = 20 50
```

Requirements

Header: <vector>

See Also

Reference

vector Class

vector::empty, vector::erase, and vector::push_back

Standard Template Library

Other Resources

vector Members

vector::front

Returns a reference to the first element in a vector.

```
reference front( );
const_reference front( ) const;
```

Return Value

A reference to the first element in the vector object. If the vector is empty, the return is undefined.

Remarks

If the return value of **front** is assigned to a **const_reference**, the vector object cannot be modified. If the return value of **front** is assigned to a **reference**, the vector object can be modified.

When compiling with _SECURE_SCL 1, a runtime error will occur if you attempt to access an element in an empty vector. See Checked Iterators for more information.

Example

```
// vector_front.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );
   v1.push_back( 11 );

   int& i = v1.front( );
   const int& ii = v1.front( );

   cout << "The first integer of v1 is "<< i << endl;
   i++;
   cout << "The second integer of v1 is "<< ii << endl;
}
```

Output

```
The first integer of v1 is 10
The second integer of v1 is 11
```

Requirements

Header: <vector>

See Also

Reference

vector Class

vector::front and vector::back

Standard Template Library

Other Resources

vector Members

vector::get_allocator

Returns a copy of the allocator object used to construct the vector.

```
Allocator get_allocator( ) const;
```

Return Value

The allocator used by the vector.

Remarks

Allocators for the vector class specify how the class manages storage. The default allocators supplied with STL container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```
// vector_get_allocator.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   // The following lines declare objects that use the default allocator.
   vector<int> v1;
   vector<int, allocator<int> > v2 = vector<int, allocator<int> >(allocator<int>( )) ;

   // v3 will use the same allocator class as v1
   vector <int> v3( v1.get_allocator( ) );

   vector<int>::allocator_type xvec = v3.get_allocator( );
   // You can now call functions on the allocator class used by vec
}
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::insert

Inserts an element or a number of elements or a range of elements into the vector at a specified position.

```
iterator insert(
   iterator _Where,
   const Type& _Val
);
void insert(
   iterator _Where,
   size_type _Count,
   const Type& _Val
);
template<class InputIterator>
      void insert(
      iterator _Where,
      InputIterator _First,
      InputIterator _Last
   );
```

Parameters

_Where

The position in the vector where the first element is inserted.

_Val

The value of the element being inserted into the vector.

_Count

The number of elements being inserted into the vector.

_First

The position of the first element in the range of elements to be copied.

_Last

The position of the first element beyond the range of elements to be copied.

Return Value

The first **insert** function returns an iterator that points to the position where the new element was inserted into the vector.

Remarks

Any insertion operation can be expensive, see vector Class for a discussion of **vector** performance.

Example

```
// vector_insert.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter;

   v1.push_back( 10 );
   v1.push_back( 20 );
   v1.push_back( 30 );

   cout << "v1 =" ;
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
```

```
      cout << " " << *Iter;
   cout << endl;

   v1.insert( v1.begin( ) + 1, 40 );
   cout << "v1 =";
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;
   v1.insert( v1.begin( ) + 2, 4, 50 );

   cout << "v1 =";
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;

   v1.insert( v1.begin( )+1, v1.begin( )+2, v1.begin( )+4 );
   cout << "v1 =";
   for (Iter = v1.begin( ); Iter != v1.end( ); Iter++ )
      cout << " " << *Iter;
   cout << endl;
}
```

Output

```
v1 = 10 20 30
v1 = 10 40 20 30
v1 = 10 40 50 50 50 50 20 30
v1 = 10 50 50 40 50 50 50 50 20 30
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::max_size

Returns the maximum length of the vector.

```
size_type max_size( ) const;
```

Return Value

The maximum possible length of the vector.

Example

```
// vector_max_size.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::size_type i;

   i = v1.max_size( );
   cout << "The maximum possible length of the vector is " << i << "." << endl;
}
```

Sample Output

```
The maximum possible length of the vector is 1073741823.
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::pop_back

Deletes the element at the end of the vector.

```
void pop_back( );
```

Example

```
// vector_pop_back.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 1 );
   cout << v1.back( ) << endl;
   v1.push_back( 2 );
   cout << v1.back( ) << endl;
   v1.pop_back( );
   cout << v1.back( ) << endl;
}
```

Output

```
1
2
1
```

Requirements

Header: <vector>

See Also

Reference

vector Class

vector::push_back and vector::pop_back

Standard Template Library

Other Resources

vector Members

vector::push_back

Adds an element to the end of the vector.

```
void push_back(
   const Type& _Val
);
```

Parameters

_Val

The element added to the end of the vector.

Example

```
// vector_push_back.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 1 );
   if ( v1.size( ) != 0 )
      cout << "Last element: " << v1.back( ) << endl;

   v1.push_back( 2 );
   if ( v1.size( ) != 0 )
      cout << "New last element: " << v1.back( ) << endl;
}
```

Output

```
Last element: 1
New last element: 2
```

Requirements

Header: <vector>

See Also

Reference

vector Class
accumulate, copy, and vector::push_back
adjacent_difference and vector::push_back
vector::empty, vector::erase, and vector::push_back
vector::push_back and vector::pop_back
Standard Template Library

Other Resources

vector Members

vector::rbegin

Returns an iterator to the first element in a reversed vector.

```
reverse_iterator rbegin( );
const_reverse_iterator rbegin( ) const;
```

Return Value

A reverse random-access iterator addressing the first element in a reversed vector or addressing what had been the last element in the unreversed vector.

Remarks

If the return value of **rbegin** is assigned to a **const_reverse_iterator**, the vector object cannot be modified. If the return value of **rbegin** is assigned to a **reverse_iterator**, the vector object can be modified.

Example

```
// vector_rbegin.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator v1_Iter;
   vector <int>::reverse_iterator v1_rIter;

   v1.push_back( 1 );
   v1.push_back( 2 );

   v1_Iter = v1.begin( );
   cout << "The first element of vector is "
        << *v1_Iter << "." << endl;

   v1_rIter = v1.rbegin( );
   cout << "The first element of the reversed vector is "
        << *v1_rIter << "." << endl;
}
```

Output

```
The first element of vector is 1.
The first element of the reversed vector is 2.
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::rend

Returns an iterator that addresses the location succeeding the last element in a reversed vector.

```
const_reverse_iterator rend( ) const;
reverse_iterator rend( );
```

Return Value

A reverse random-access iterator that addresses the location succeeding the last element in a reversed vector (the location that had preceded the first element in the unreversed vector).

Remarks

rend is used with a reversed vector just as end is used with a vector.

If the return value of **rend** is assigned to a **const_reverse_iterator**, then the vector object cannot be modified. If the return value of **rend** is assigned to a **reverse_iterator**, then the vector object can be modified.

rend can be used to test to whether a reverse iterator has reached the end of its vector.

The value returned by **rend** should not be dereferenced.

Example

```
// vector_rend.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::reverse_iterator v1_rIter;

   v1.push_back( 1 );
   v1.push_back( 2 );

   for ( v1_rIter = v1.rbegin( ) ; v1_rIter != v1.rend( ) ; v1_rIter++ )
      cout << *v1_rIter << endl;
}
```

Output

```
2
1
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::reserve

Reserves a minimum length of storage for a vector object, allocating space if necessary.

```
void reserve(
   size_type _Count
);
```

Parameters

_Count

The minimum length of storage to be allocated for the vector.

Example

```
// vector_reserve.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   //vector <int>::iterator Iter;

   v1.push_back( 1 );
   cout << "Current capacity of v1 = "
      << v1.capacity( ) << endl;
   v1.reserve( 20 );
   cout << "Current capacity of v1 = "
      << v1.capacity( ) << endl;
}
```

Output

```
Current capacity of v1 = 1
Current capacity of v1 = 20
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::resize

Specifies a new size for a vector.

```
void resize(
   size_type _Newsize
);
void resize(
   size_type _Newsize,
      Type _Val
);
```

Parameters

_Newsize

The new size of the vector.

_Val

The value of new elements added to the vector if the new size is larger that the original size. If the value is omitted, the new objects are assigned the default value.

Remarks

If the container's size is less than the requested size, *_Newsize*, elements are added to the vector until it reaches the requested size. If the container's size is larger than the requested size, the elements closest to the end of the container are deleted until the container reaches the size *_Newsize*. If the present size of the container is the same as the requested size, no action is taken.

size reflects the current size of the vector.

Example

```
// vector_resize.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );
   v1.push_back( 20 );
   v1.push_back( 30 );

   v1.resize( 4,40 );
   cout << "The size of v1 is " << v1.size( ) << endl;
   cout << "The value of the last object is " << v1.back( ) << endl;

   v1.resize( 5 );
   cout << "The size of v1 is now " << v1.size( ) << endl;
   cout << "The value of the last object is now " << v1.back( ) << endl;
}
```

Output

```
The size of v1 is 4
The value of the last object is 40
The size of v1 is now 5
The value of the last object is now 0
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector::size

Returns the number of elements in the vector.

```
size_type size( ) const;
```

Return Value

The current length of the vector.

Example

```
// vector_size.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::size_type i;

   v1.push_back( 1 );
   i = v1.size( );
   cout << "Vector length is " << i << "." << endl;

   v1.push_back( 2 );
   i = v1.size( );
   cout << "Vector length is now " << i << "." << endl;
}
```

Output

```
Vector length is 1.
Vector length is now 2.
```

Requirements

Header: <vector>

See Also

Reference

vector Class

vector::size and vector::capacity

Standard Template Library

Other Resources

vector Members

vector::swap

Exchanges the elements of two vectors.

```
void swap(
   vector& _Right
);
friend void swap(
   vector<Type, Allocator >& _Left,
   vector<Type, Allocator >& _Right
);
```

Parameters

_Right

A vector providing the elements to be swapped, or a vector whose elements are to be exchanged with those of the vector *_Left*.

_Left

A vector whose elements are to be exchanged with those of the vector *_Right*.

Example

```
// vector_swap.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1, v2;

   v1.push_back( 1 );
   v1.push_back( 2 );
   v1.push_back( 3 );

   v2.push_back( 10 );
   v2.push_back( 20 );

   cout << "The number of elements in v1 = " << v1.size( ) << endl;
   cout << "The number of elements in v2 = " << v2.size( ) << endl;
   cout << endl;

   v1.swap( v2 );

   cout << "The number of elements in v1 = " << v1.size( ) << endl;
   cout << "The number of elements in v2 = " << v2.size( ) << endl;
}
```

Output

```
The number of elements in v1 = 3
The number of elements in v2 = 2

The number of elements in v1 = 2
The number of elements in v2 = 3
```

Requirements

Header: <vector>

See Also

Reference

vector Class
Standard Template Library
Other Resources
vector Members

vector::vector

Constructs a vector of a specific size or with elements of a specific value or with a specific allocator or as a copy of all or part of some other vector.

```
vector( );
explicit vector(
   const Allocator& _Al
);
explicit vector(
   size_type _Count
);
vector(
   size_type _Count,
   const Type& _Val
);
vector(
   size_type _Count,
   const Type& _Val,
   const Allocator& _Al
);
vector(
   const _vector& _Right
);
template<class InputIterator>
   vector(
      InputIterator _First,
      InputIterator _Last
   );
template<class InputIterator>
   vector(
      InputIterator _First,
      InputIterator _Last,
      const Allocator& _Al
   );
```

Parameters

_Al

The allocator class to use with this object. get_allocator returns the allocator class for the object.

_Count

The number of elements in the constructed vector.

_Val

The value of the elements in the constructed vector.

_Right

The vector of which the constructed vector is to be a copy.

_First

Position of the first element in the range of elements to be copied.

_Last

Position of the first element beyond the range of elements to be copied.

Remarks

All constructors store an allocator object (*_Al*) and initialize the vector.

The first two constructors specify an empty initial vector. The second explicitly specifies the allocator type (*_Al*) to be used.

The third constructor specifies a repetition of a specified number (*_Count*) of elements of the default value for class *Type*.

The fourth and fifth constructors specify a repetition of (_*Count*) elements of value _*Val*.

The sixth constructor specifies a copy of the vector _*Right*.

The last two constructors copy the range [_*First*, _*Last*) of a vector.

Example

```
// vector_ctor.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int>::iterator v1_Iter, v2_Iter, v3_Iter, v4_Iter, v5_Iter;

   // Create an empty vector v0
   vector <int> v0;

   // Create a vector v1 with 3 elements of default value 0
   vector <int> v1( 3 );

   // Create a vector v2 with 5 elements of value 2
   vector <int> v2( 5, 2);

   // Create a vector v3 with 3 elements of value 1 and with the allocator
   // of vector v2
   vector <int> v3( 3, 1, v2.get_allocator( ) );

   // Create a copy, vector v4, of vector v2
   vector <int> v4( v2 );

   // Create a vector v5 by copying the range v4[_First, _Last)
   vector <int> v5( v4.begin( ) + 1, v4.begin( ) + 3 );

   cout << "v1 =" ;
   for ( v1_Iter = v1.begin( ) ; v1_Iter != v1.end( ) ; v1_Iter++ )
      cout << " " << *v1_Iter;
   cout << endl;

   cout << "v2 =" ;
   for ( v2_Iter = v2.begin( ) ; v2_Iter != v2.end( ) ; v2_Iter++ )
      cout << " " << *v2_Iter;
   cout << endl;

   cout << "v3 =" ;
   for ( v3_Iter = v3.begin( ) ; v3_Iter != v3.end( ) ; v3_Iter++ )
      cout << " " << *v3_Iter;
   cout << endl;

   cout << "v4 =" ;
   for ( v4_Iter = v4.begin( ) ; v4_Iter != v4.end( ) ; v4_Iter++ )
      cout << " " << *v4_Iter;
   cout << endl;

   cout << "v5 =";
   for ( v5_Iter = v5.begin( ) ; v5_Iter != v5.end( ) ; v5_Iter++ )
      cout << " " << *v5_Iter;
   cout << endl;
}
```

Output

```
v1 = 0 0 0
v2 = 2 2 2 2 2
v3 = 1 1 1
v4 = 2 2 2 2 2
```

```
v5 = 2 2
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

vector Operators

For information about the operators in the vector class, see vector Members.

See Also

Reference

Standard Template Library

vector::operator[]

Returns a reference to the vector element at a specified position.

```
reference operator[](
   size_type _Pos
);
const_reference operator[](
   size_type _Pos
) const;
```

Parameters

_Pos

The position of the vector element.

Return Value

If the position specified is greater than the size of the container, the result is undefined.

Remarks

If the return value of **operator[]** is assigned to a **const_reference**, the vector object cannot be modified. If the return value of **operator[]** is assigned to a **reference**, the vector object can be modified.

When compiling with _SECURE_SCL 1, a runtime error will occur if you attempt to access an element outside the bounds of the vector. See Checked Iterators for more information.

Example

```
// vector_op_ref.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );
   v1.push_back( 20 );

   int& i = v1[1];
   cout << "The second integer of v1 is " << i << endl;
}
```

Output

```
The second integer of v1 is 20
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector Members

<vector> Specializations

For information about the specializations in <vector>, see <vector> Members.

See Also

Reference

Standard Template Library

vector<bool> Class

The STL vector<bool> class is a partial specialization of the template vector Class for elements of type bool with an allocator for the underlying type used by the specialization.

For a list of all members of this type, see vector<bool> Members.

```
template<class Allocator> class vector< bool, allocator >
```

Remarks

The vector<bool> reference Class is a nested class whose objects are able to provide references to elements (single bits) within a vector<bool> object.

Requirements

Header: <vector>

See Also

Reference

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

vector<bool> Members

<vector> Members

vector<bool> Members

Reference

Typedefs

const_iterator	A type that describes an object that can serve as a constant random-access iterator for the sequence of Boolean elements contained by the **vector**.
const_pointer	A type that describes an object that can serve as a constant pointer to a Boolean element of the sequence contained by the **vector**.
const_reference	A type that describes an object that can serve as a constant reference to a Boolean element of the sequence contained by the **vector**.
iterator	A type that describes an object that can serve as a random-access iterator for a sequence of Boolean elements contained by a **vector**.
pointer	A type that describes an object that can serve as a constant pointer to a Boolean element of the sequence contained by the **vector**.

Member Functions

flip	Reverses all bits in the **vector**.
swap	Exchanges the elements of two **vector**s with Boolean elements.

Nested Classes

vector<bool> reference Class	A nested class whose objects are able to provide references to elements (single bits) within a **vector<bool>** object.

See Also

Reference

vector<bool> Class
Thread Safety in the Standard C++ Library
Standard Template Library

vector<bool> Typedefs

For information about the typedefs in the vector<bool> class, see vector<bool> Members.

See Also

Reference

Standard Template Library

vector<bool>::const_iterator

A type that describes an object that can serve as a constant random-access iterator for the sequence of Boolean elements contained by the vector.

```
typedef implementation-defined-type const_iterator;
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector<bool> Members

vector<bool>::const_pointer

A type that describes an object that can serve as a constant pointer to a Boolean element of the sequence contained by the vector.

```
typedef implementation-defined-type const_pointer;
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector<bool> Members

vector<bool>::const_reference

A type that describes an object that can serve as a constant reference to a Boolean element of the sequence contained by the vector.

```
typedef implementation-defined-type const_reference;
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector<bool> Members

vector<bool>::iterator

A type that describes an object that can serve as a random-access iterator for a sequence of Boolean elements contained by a vector.

```
typedef implementation-defined-type iterator;
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector<bool> Members

vector<bool>::pointer

A type that describes an object that can serve as a constant pointer to a Boolean element of the sequence contained by the vector.

```
typedef implementation-defined-type pointer;
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector<bool> Members

vector<bool> Member Functions

For information about the member functions in the vector<bool> class, see vector<bool> Members.

See Also

Reference

Standard Template Library

vector<bool>::flip

Reverses all bits in a vector with Boolean elements.

```
void flip( );
```

Example

```
// vector_bool_flip.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   _Bvector vb;

   vb.push_back( 0 );
   vb.push_back( 1 );

   cout << "The vector is: " << vb.front( ) << " " << vb.back( ) << endl;
   vb.flip( );
   cout << "The flipped vector is: " << vb.front( ) << " " << vb.back( ) << endl;
}
```

Output

```
The vector is: 0 1
The flipped vector is: 1 0
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector<bool> Members

vector<bool>::swap

Exchanges the elements of two vectors with Boolean elements.

```
void swap(
   reference _Left,
   reference _Right
);
```

Parameters

_Left

The vector whose elements are to be exchanged with those of the vector *_Right*.

_Right

The vector whose elements are to be exchanged with those of the vector *_Left*.

Example

```
// vector_bool_swap.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   _Bvector v1, v2;

   v1.push_back( 1 );
   v1.push_back( 1 );
   v2.push_back( 0 );
   v2.push_back( 0 );

   cout << "The vector v1 is: " << v1.front( ) << " " << v1.back( ) << endl;
   cout << "The vector v2 is: " << v2.front( ) << " " << v2.back( ) << endl;

   swap( v1,v2 );

   cout << "After swapping, vector v1 is: " << v1.front( ) << " " << v1.back( ) << endl;
   cout << "After swapping, vector v2 is: " << v2.front( ) << " " << v2.back( ) << endl;
}
```

Output

```
The vector v1 is: 1 1
The vector v2 is: 0 0
After swapping, vector v1 is: 0 0
After swapping, vector v2 is: 1 1
```

Requirements

Header: <vector>

See Also

Reference

vector Class

Standard Template Library

Other Resources

vector<bool> Members

Nested Classes

For more information about the nested classes in vector<bool>, see vector<bool> Class.

See Also

Reference

Standard Template Library

vector<bool>::reference Class

The **template<class Allocator> class vector< bool, allocator > reference class** is a nested class in the vector<bool> Class whose objects are able to provide references to elements (single bits) within a vector<bool> object. C++ does not natively allow a reference to bits. But vector<bool> uses only one bit per element and these can be referenced using the objects in the nested class reference.

Requirements

Header: <vector>

See Also

Reference

<vector>

Thread Safety in the Standard C++ Library

Standard Template Library

Other Resources

vector<bool>::reference Members

vector<bool>::reference Members

Reference

Member Functions

flip	Inverts the Boolean value of a vector element.
operator bool	Exchanges the elements of two vectors with Boolean elements.
operator~	Returns a Boolean value that is the opposite of that held by the element referenced.
operator=	Assigns a Boolean value to a bit or the value held by a referenced element to a bit.

See Also

Reference

vector<bool>::reference Class
Thread Safety in the Standard C++ Library
Standard Template Library

vector<bool> reference Member Functions

For information about the member functions in the vector<bool> reference class, see vector<bool> reference Members.

See Also

Reference

Standard Template Library

vector<bool>::reference::flip

Inverts the Boolean value of a vector element.

```
void flip( );
```

Remarks

reference operator~ also inverts the Boolean value of a vector element.

Example

```
// vector_bool_ref_flip.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

class MyAlloc{};
int main( )
{
   using namespace std;
   typedef vector<bool> boolvector;
   boolvector v;

   v.push_back( false );
   boolvector::reference ref1 = v.at( 0 );

   cout << "The original value of the 1st element is: " << bool( ref1 ) << endl;
   ref1.flip( );
   cout << "The value of the 1st element is now: " << ref1 << endl;
}
```

Output

```
The original value of the 1st element is: 0
The value of the 1st element is now: 1
```

Requirements

Header: <vector>

See Also

Reference

vector<bool>::reference Class

Standard Template Library

Other Resources

vector<bool>::reference Members

vector<bool>::reference::operator bool

Casts a variable of vector<bool>::reference.

```
operator bool( ) const;
```

Return Value

The Boolean value of the element of the vector.

Remarks

The vector object cannot be modified by this operator.

Example

```
// vector_bool_ref_opbool.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   typedef vector<bool> boolvector;
   boolvector v;

   v.push_back( false );
   boolvector::reference ref1 = v.at( 0 );
   cout << ref1 << endl;   // ref1 implicitly cast to bool

   bool b1;

   // One form of an explicit cast
   b1 = ref1.operator bool( );
   cout << b1 << endl;

   // Another form of an explicit cast
   b1 = bool( ref1 );
   cout << b1 << endl;
}
```

Output

```
0
0
0
```

Requirements

Header: <vector>

See Also

Reference

vector<bool>::reference Class

Standard Template Library

Other Resources

vector<bool>::reference Members

vector<bool>::reference::operator~

Returns a Boolean value that is the opposite of that held by the element referenced.

```
bool operator~( ) const;
```

Return Value

The inverted Boolean value of the vector element.

Remarks

The vector object cannot be modified by this operator.

flip also inverts the Boolean value of a vector element.

Example

```
// vector_bool_ref_op_opp.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   typedef vector<bool> boolvector;
   boolvector v;
   bool b1 , b2;

   v.push_back( false );
   v.push_back( true );
   boolvector::reference ref1 = v.at( 0 );
   boolvector::reference ref2 = v.at( 1 );

   // There are two ways to invert the value:
   b1 = ref1.operator~( );
   b2 = ~ref2;

   cout << "Value of inverted 1st element obtained is: " << b1 << endl;
   cout << "Value of 1st element in vector is still: " << ref1 << endl;
   cout << endl;
   cout << "Value of inverted 2nd element obtained is: " << b2 << endl;
   cout << "Value of 2nd element in vector is still: " << ref2 << endl;
   cout << endl;
}
```

Output

```
Value of inverted 1st element obtained is: 1
Value of 1st element in vector is still: 0

Value of inverted 2nd element obtained is: 0
Value of 2nd element in vector is still: 1
```

Requirements

Header: <vector>

See Also

Reference

vector<bool>::reference Class

Standard Template Library

Other Resources

vector<bool>::reference Members

vector<bool>::reference::operator=

Assigns a Boolean value to a bit or the value held by a referenced element to a bit.

```
reference& operator=(
   const reference& _Right
);
reference& operator=(
   bool _Val
);
```

Parameters

_Right

The element reference whose value is to be assigned to the bit.

_Val

The Boolean value to be assigned to the bit.

Return Value

The Boolean value either designated or referred to.

Example

```
// vector_bool_ref_op_assign.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

class MyAlloc{};
int main( )
{
   using namespace std;

   typedef vector<bool> boolvector;
   boolvector v;
   bool b1, b2;

   v.push_back( false );
   v.push_back( true );
   boolvector::reference ref1 = v.at( 0 );
   boolvector::reference ref2 = v.at( 1 );
   b1 = ref1;
   b2 = ref2;
   cout << "The initial value of the 1st element was: " << b1 << endl;
   cout << "The initial value of the 2nd element was: " << b2 << endl;

   ref2.operator=( ref1 );
   b2 = ref2;
   cout << "The value of the 2nd element is now: " << b2 << endl;

   ref1.operator=( true );
   b1 = bool( ref1 );
   cout << "The value of the 1st element is now: " << b1 << endl;
}
```

Output

```
The initial value of the 1st element was: 0
The initial value of the 2nd element was: 1
The value of the 2nd element is now: 0
The value of the 1st element is now: 1
```

Requirements

Header: <vector>

See Also

Reference

vector<bool>::reference Class

Standard Template Library

Other Resources

vector<bool>::reference Members

Standard Template Library Samples

The following samples demonstrate the use of the Standard Template Library, a subset of the Standard C++ Library.

abs

accumulate, copy and vector::push_back

adjacent_difference and vector::push_back

adjacent_find

adjacent_find (predicate version)

advance

Basic Math Functions

basic_string::append

basic_string::find_first_of

basic_string::size and basic_string::resize

binary_function Structure

count

count_if

deque::assign and deque::swap

deque::begin and deque::end

deque::erase and deque::clear

deque::front and deque::back

deque::insert

deque::operator[] and deque::at

deque::operator== and deque::operator<

deque::push_back and deque::pop_back

deque::push_front and deque::pop_front

deque::rbegin and deque::rend

deque::size and deque::resize

distance

equal_to

exp, log, and log10

find

find_if

for_each

generate

generate_n

heap

heap (predicate version)

includes

includes (predicate version)

inner_product

inplace_merge

inplace_merge (predicate version)

iter_swap

list::assign

list::back and list::front

list::insert

list::list

lower_bound

lower_bound (predicate version)

make_pair

map::insert, map::find, and map::end

map::max_size, map::clear, map::erase, and map::size

max_element

max_element (predicate version)

Members of the numeric_limits Class

merge

merge (predicate version)

min_element

min_element (predicate version)

new operator

next_permutation

next_permutation (predicate version)

nth_element

nth_element (predicate version)

Pair Logical Operator

partial_sort

partial_sort (predicate version)

partial_sort_copy

partial_sort_copy (predicate version)

partial_sum

partition

prev_permutation

priority_queue functions

queue functions

random_shuffle

random_shuffle (predicate version)

remove

remove_copy

remove_copy_if

remove_if

replace

replace_copy

replace_copy_if

replace_if

reverse

reverse_copy

rotate

rotate_copy

set::key_comp and set::value_comp

set::lower_bound, set::upper_bound, and set::equal_range

set::count

set::empty and set::clear

set::find

set::max_size

set::rbegin and set::rend

set::size

set::swap, set::begin, and set::end

sqrt and pow

stack::operator<

stack::operator==

stack::size

stack::top and stack::empty

string::getline

string::operator!=

string::operator+

string::operator<

string::operator<<

string::operator<=

string::operator==

string::operator>

string::operator>=

string::operator>>

Trigonometry Functions

unary_function<> Structure

upper_bound

upper_bound (predicate version)

vector::operator<

vector::operator==

vector::empty, vector::erase, and vector::push_back

vector::front and vector::back

vector::push_back and vector::pop_back

vector::size and vector::capacity

See Also

Other Resources

Standard C++ Library Reference

abs (STL Samples)

Illustrates how to use the abs Standard Template Library (STL) function in Visual C++.

```
template<class T>
   valarray<T> abs(
      const valarray<T>& x
   );
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The sample declares a valarray of integers and uses the **abs** STL function to get the absolute value for each array element.

Example

```
// abs_main.cpp
// compile with: /EHsc
#include <iostream>                 // for i/o functions
#include <valarray>                 // for valarray
using namespace std;

#define ARRAY_SIZE  10              // array size
typedef valarray<int> INTVALARRAY;  // type for valarray of ints

int main()
{
    int i;
    // Initialize val_array to 0, -1, -2, etc.
    INTVALARRAY val_array(ARRAY_SIZE);
    for (i = 0; i < ARRAY_SIZE; i++)
        val_array[i] = -i;

    // Display the size of val_array.
    cout << "Size of val_array = " << val_array.size() << "\n\n";

    // Display the values of val_array before calling abs().
    cout << "The result of val_array before calling abs():\n";
    for (i = 0; i < ARRAY_SIZE; i++)
    {
        cout << val_array[i];
        if (i < ARRAY_SIZE - 1)
            cout << ", ";
    }
    cout << endl << endl;

    // Display the result of val_array after calling abs().
    INTVALARRAY abs_array = abs(val_array);
    cout << "The result of val_array after calling abs():\n";
    for (i = 0; i < ARRAY_SIZE; i++)
    {
        cout << abs_array[i];
        if (i < ARRAY_SIZE - 1)
            cout << ", ";
    }
    cout << endl;
}
```

Output

```
Size of val_array = 10

The result of val_array before calling abs():
0, -1, -2, -3, -4, -5, -6, -7, -8, -9

The result of val_array after calling abs():
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Requirements

Header: <valarray>

See Also

Concepts

Standard Template Library Samples

accumulate, copy, and vector::push_back

Illustrates how to use the accumulate, copy, and vector::push_back Standard Template Library (STL) functions in Visual C++.

```
template<class InputIterator, class _TYPE> inline
   _TYPE accumulate(
      InputIterator First,
      InputIterator Last,
      _TYPE Init
   )
template<class InputIterator, class _TYPE, class BinaryOperator> inline
   _TYPE accumulate(
      InputIterator First,
      InputIterator Last,
      _TYPE Init,
      BinaryOperator Binary_Op
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **accumulate** STL function initializes an accumulator **acc** with an initial value init and then modifies it with **acc** = **acc** + ***i** or **acc** = *Binary_Op*(**acc**,**i*) for every iterator *i* in the range [*First*, *Last*) in order. Normally, the **accumulate** STL function is used to sum the numeric elements of a vector. However, it can also be used to do other useful work such as concatenating a vector of strings.

Example

```
// accumulate.cpp
// compile with: /EHsc
//
// Description of accumulate(first,last,init)
//                accumulate(first,last,init,binary_op):
//
//    Initializes the accumulator acc with the initial value init
//        acc = init
//    and then modifies it with
//        acc  =  acc + *i
//    or
//        acc = binary_op(acc, *i)
//    for every iterator i in the range [first, last) in order.

// turn off warning about symbols too long for debugger
#pragma warning (disable : 4786)

#include <iostream>
#include <numeric>
#include <functional>
#include <vector>
#include <iterator>
#include <string>

using namespace std;


typedef vector < float > FloatArray;
typedef vector < string > StringArray;
typedef ostream_iterator <float, char, char_traits <char> > FloatOstreamIt;
```

```
int main ()
{
    // a vector of floats
    FloatArray rgFA;

    // an ostream iterator that outputs a float to cout terminated
    // by a space
    FloatOstreamIt OstreamIt(cout," ");

    // Initialize the array to 1,1/2,1/3,...
    for (int i=0; i<10; i++) rgFA.push_back(1.0f/(i+1));

    // Print the array
    copy(rgFA.begin(),rgFA.end(),OstreamIt);
    cout << endl;

    // Sum the array
    cout << "The sum of 1 + 1/2 + 1/3 + ... + 1/10 is "
         << accumulate(rgFA.begin(),rgFA.end(),0.0f)
         << endl;

    // Compute the product of the array
    cout << "The product of 1 * 1/2 * 1/3 * ... * 1/10 is "
         << accumulate(rgFA.begin(),rgFA.end(),1.0f,multiplies<float>())
         << endl;

    // Initialize array of strings
    StringArray rgs;
    rgs.push_back("This ");
    rgs.push_back("is ");
    rgs.push_back("one ");
    rgs.push_back("sentence. ");

    // Concatenate the strings in the array and print the sentence
    cout << "The concatenated vector of strings: "
         << accumulate(rgs.begin(),rgs.end(),string(""))
         << endl;
}
```

Output

```
1 0.5 0.333333 0.25 0.2 0.166667 0.142857 0.125 0.111111 0.1
The sum of 1 + 1/2 + 1/3 + ... + 1/10 is 2.92897
The product of 1 * 1/2 * 1/3 * ... * 1/10 is 2.75573e-007
The concatenated vector of strings: This is one sentence.
```

Requirements

Header: <numeric>

See Also

Concepts

Standard Template Library Samples

adjacent_difference and vector::push_back

Illustrates how to use the adjacent_difference and vector::push_back Standard Template Library (STL) functions in Visual C++.

```
template<class InputIterator, class OutputIterator> inline
   OutputIterator adjacent_difference(
      InputIterator First,
      InputIterator Last,
      OutputIterator Result
   )
template<class InputIterator, class OutputIterator, class BinaryOperator> inline
   OutputIterator adjacent_difference(
      InputIterator First,
      InputIterator Last,
      OutputIterator Result,
      BinaryOperator Binary_Op
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

Assigns to every element referred to by iterator *i* in the range [*Result* + 1, *Result* + (*Last* - *First*)) a value correspondingly equal to *(*First* + (*i* - *Result*)) - *(*First* + (*i* - *Result*) - 1) or *Binary_Op* (*(*First* + (*i* - *Result*)), *(*First* + (i - *Result*) - 1)). *Result* gets the value of **First*.

Example

```
// adj_diff.cpp
// compile with: /EHsc
//
// Description of adjacent_difference(first,last,result)
//                adjacent_difference(first,last,result,binary_op):
//
//    Assigns to every element referred to by iterator i in the range
//    [result + 1, result + (last - first))
//    a value correspondingly equal to
//    *(first + (i - result)) - *(first + (i - result) - 1)
//    or
//    binary_op(*(first + (i - result)), *(first + (i - result) - 1)).
//    Result gets the value of *first.

#include <iostream>
#include <numeric>
#include <functional>
#include <vector>
#include <iterator>

using namespace std;


typedef vector < int > IntegerArray;
typedef ostream_iterator < int, char, char_traits<char> > IntOstreamIt;

int main ()
{
    // an ostream iterator that outputs an int to cout terminated
    // by a space
    IntOstreamIt itOstream(cout," ");

    // Initialize the array
```

```
    // Suppose that you are taking a trip and can measure
    // the miles traveled from your city of origin
    // to the city you are traveling through
    IntegerArray rgIA;
    rgIA.push_back(5661); // San Francisco to Berlin
    rgIA.push_back(7456); // to Cairo
    rgIA.push_back(10995); // to Kolkata
    rgIA.push_back(17019); // to Cape Town
    rgIA.push_back(24394); // to Hong Kong
    rgIA.push_back(30376); // to London
    rgIA.push_back(35758); // to Los Angeles

    // Print the array
    copy(rgIA.begin(),rgIA.end(),itOstream);
    cout << endl;

    // Suppose that you now want the distance between each
    // of the cities that you traveled to. You can easily
    // find it with adjacent_difference()
    IntegerArray rgDifferences(7);
    IntegerArray::iterator itDifferences = rgDifferences.begin();
    adjacent_difference(rgIA.begin(),rgIA.end(),itDifferences);

    // Print the differences
    // Remember that the first item in the differences array is
    // not a difference, but is unused space
    cout << "The adjacent differences are: ";
    copy(rgDifferences.begin()+1,rgDifferences.end(),itOstream);
    cout << endl;

    // Suppose that you now want to know which adjacent differences
    // are greater. If you have [a,b,c], you would like [1,0] if a>b
    // and b<=c.
    // You are using less() rather than greater() because
    // adjacent_difference() reverses the parameters. For example,
    // if a and b are adjacent, adjacent_difference() calls
    // less(b,a). See the explanation at the top of this file
    // for a more exact description.
    IntegerArray rgGT(6);
    IntegerArray::iterator itGT = rgGT.begin();
    adjacent_difference(rgDifferences.begin()+1,
                        rgDifferences.end(),
                        itGT,
                        less<int>());

    // Print the greater thans
    // Remember that the first item in the differences array is
    // not a difference, but is unused space
    cout << "Which adjacent distances are greater:" << endl
         << "(If you have [a,b,c], then you have [1,0] if a>b and b<=c)"
         << endl;
    copy(rgGT.begin()+1,rgGT.end(),itOstream);
    cout << endl;
}
```

Output

```
5661 7456 10995 17019 24394 30376 35758
The adjacent differences are: 1795 3539 6024 7375 5982 5382
Which adjacent distances are greater:
(If you have [a,b,c], then you have [1,0] if a>b and b<=c)
0 0 0 1 1
```

Requirements

Header: <numeric>

See Also

Concepts

advance (STL Samples)

Illustrates how to use the advance Standard Template Library (STL) function in Visual C++.

```
template<class InIt, class Dist>
   void advance(
            InIt& it,
            Dist n
   );
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **advance** STL function accepts two parameters:

- *InIt* — The iterator to advance.
- *Dist* — The number of elements to increment the iterator by.

The **advance** STL function advances the iterator *n* times. If the iterator is a random-access iterator type, the function evaluates the expression as iterator += *n*. Otherwise, it performs each increment by evaluating: ++iterator. If the iterator is an input or forward iterator type, *n* must not be negative.

Example

```
// Advance.cpp
// compile with: /EHsc

#pragma warning (disable:4786)
#include <iostream>
#include <string>
#include <list>

using namespace std ;

typedef list<string> STRLIST;

int main() {
    STRLIST List;
    STRLIST::iterator iList;
    STRLIST::difference_type dTheDiff;

    List.push_back("A1");
    List.push_back("B2");
    List.push_back("C3");
    List.push_back("D4");
    List.push_back("E5");
    List.push_back("F6");
    List.push_back("G7");

    // Print out the list
    iList=List.begin();
    cout << "The list is: ";
    for (int i = 0; i < 7 ; i++, iList++)
        cout << *iList  << "  ";

    // Initialize to the first element"
    iList=List.begin();
    cout << "\n\nAdvance to the 3rd element." << endl;
    advance(iList,2);
```

```
    cout << "The element is " << *iList << endl;
    dTheDiff = distance( List.begin(), iList);
}
```

Output

```
The list is: A1  B2  C3  D4  E5  F6  G7

Advance to the 3rd element.
The element is C3
```

Requirements

Header: <iterator>

See Also

Concepts

Standard Template Library Samples

Basic Math Functions

Illustrates how to use some basic math functions in the Standard Template Library (STL) in Visual C++.

```
template<class _TYPE>
   struct plus : binary_function<_TYPE, _TYPE, _TYPE>
   {
      _TYPE operator()(const _TYPE& _X, const _TYPE& _Y) const
      {return (_X + _Y); }
   };
template<class _TYPE>
   struct minus : binary_function<_TYPE, _TYPE, _TYPE>
   {
      _TYPE operator()(const _TYPE& _X, const _TYPE& _Y) const
      {return (_X - _Y); }
   };
template<class _TYPE>
   struct times : binary_function<_TYPE, _TYPE, _TYPE>
   {
      _TYPE operator()(const _TYPE& _X, const _TYPE& _Y) const
     {return (_X * _Y); }
   };
template<class _TYPE>
   struct divides : binary_function<_TYPE, _TYPE, _TYPE>
   {
      _TYPE operator()(const _TYPE& _X, const _TYPE& _Y) const
      {return (_X / _Y); }
   };
template<class _TYPE>
   struct modulus : binary_function<_TYPE, _TYPE, _TYPE>
   {
      _TYPE operator()(const _TYPE& _X, const _TYPE& _Y) const
      {return (_X % _Y); }
   };
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

This sample uses a class derived from all five basic math structures: plus, minus, multiply, divide, and modulus, using an integer as the templated operand.

Example

```
// mathfunc.cpp
// compile with: /EHsc
//
// Structures: plus<A>    - Adds data type A object to
//                          a class object derived from plus.
//             minus<A>   - Subtracts data type A.
//             multiplies<A>   - Multiplies object by data type A.
//             divides<A> - Divides object by data type A.
//             modulus<A> - Returns object modulo A.

#include <functional>
#include <iostream>

using namespace std ;

class MathOps : public plus<int>, public minus<int>,
                 public multiplies<int>, public divides<int>,
```

```
                    public modulus<int>
  {
  public:
    int value;
    MathOps(){value=0;}
    MathOps(int x){value=x;}
    result_type operator+(second_argument_type add2)
                              {return value + add2;}
    result_type operator-(second_argument_type sub2)
                              {return value - sub2;}
    result_type operator*(second_argument_type mult2)
                              {return value * mult2;}
    result_type operator/(second_argument_type div2)
                              {return value / div2;}
    result_type operator%(second_argument_type mod2)
                              {return value % mod2;}
    friend ostream& operator<<(ostream& os, const MathOps& obj ) ;
  };

  ostream& operator<<(ostream& os, const MathOps& obj )
  {
         os << obj.value ;
         return os ;
  }

  int main(void)
  {
    MathOps one,two,three,four,five,six;

    cout << "Using MathOps class..." << endl ;

    one = 18;
    cout << "one = " << one << endl ;

    two = one + 1;
    cout << "two = one + 1 = " << two << endl ;

    three = two - 2;
    cout << "three = two - 2 = " << three << endl ;

    four = three * 3;
    cout << "four = three * 3 = " << four << endl ;

    five = four / 4;
    cout << "five = four / 4 = " << five << endl ;

    six = five % 5;
    cout << "six = five % 5 = " << six << endl ;
  }
```

Output

```
Using MathOps class...
one = 18
two = one + 1 = 19
three = two - 2 = 17
four = three * 3 = 51
five = four / 4 = 12
six = five % 5 = 2
```

Requirements

Header: <functional>

See Also

Reference

<functional>

Concepts

Standard Template Library Samples

Other Resources

<functional> Members

basic_string::append (STL Samples)

Illustrates how to use the basic_string::append Standard Template Library (STL) function in Visual C++.

```
string& append(
   const basic_string& _X
);
string& append(
   const basic_string& _X,
   size_type pos,
   size_type count
);
string& append(
   const element_type *_S,
   size_type count
);
string& append(
   const element_type *_S
);
string& append(
   size_type count,
   element_type _C
);
string& append(
   Iterator first,
   Iterator last
);
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **append** member functions of the **basic_string::append** elements to the end of the string. The different forms of the function provide alternate ways to specify the source of the elements that will be appended. The **append** functions return a reference to the string to which the elements were appended.

Example

```
// bsappend.cpp
// compile with: /EHsc
//
// Functions:
//    string::append - appends a sequence of elements to the
//                           current string.
//////////////////////////////////////////////////////////////////////

#include <string>
#include <iostream>

using namespace std ;

int main()
{
    string str1("012");
    string str2("345");

    cout << "str1 = " << str1.c_str() << endl;

    // append str2 to str1
    str1.append(str2);
```

```
    cout << "str1 = " << str1.c_str() << endl;

    // append the last 2 items in str2 to str1
    str2 = "567";
    str1.append(str2, 1, 2);    // begin at pos 1, append 2 elements

    cout << "str1 = " << str1.c_str() << endl;

    // append the first 2 items from an array of the element type
    char achTest[] = {'8', '9', 'A'};
    str1.append(achTest, 2);

    cout << "str1 = " << str1.c_str() << endl;

    // append all of a string literal to str1
    char szTest[] = "ABC";
    str1.append(szTest);

    cout << "str1 = " << str1.c_str() << endl;

    // append one item of the element type
    str1.append(1, 'D');

    cout << "str1 = " << str1.c_str() << endl;

    // append str2 to str1 using iterators
    str2 = "EF";
    str1.append (str2.begin(), str2.end());

    cout << "str1 = " << str1.c_str() << endl;
}
```

Output

```
str1 = 012
str1 = 012345
str1 = 01234567
str1 = 0123456789
str1 = 0123456789ABC
str1 = 0123456789ABCD
str1 = 0123456789ABCDEF
```

Requirements

Header: <string>

See Also

Concepts

Standard Template Library Samples

basic_string::find_first_of (STL Samples)

Illustrates how to use the basic_string::find_first_of Standard Template Library (STL) function in Visual C++.

```
size_type find_first_of(
   const basic_string& _X,
   size_type iPos = 0
);
size_type find_first_of(
   const element_type *_S,
   size_type iPos,
   size_type cElementsIn_S
);
size_type find_first_of(
   const element_type *_S,
   size_type iPos = 0
);
size_type find_first_of(
   element_type _C,
   size_type iPos = 0
);
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The member functions each find the first (lowest position) element of the controlled sequence at or after position *iPos* that matches any of the elements in the operand sequence specified by the remaining operands. If it succeeds, it returns the position. Otherwise, the function returns *npos*. The position returned is 0 (zero) based. The *npos* return value is a special value indicating that none of the elements was found.

Example

```
// main.cpp
// compile with: /EHsc
//
// Functions:
//
//    string::find_first_of() - find the first instance in the
//         controlled string of any of the elements specified by the
//         parameters. The search begins at an optionally-supplied
//         position in the controlled string.

#include <string>
#include <iostream>

using namespace std ;

int main()
{
    string str1("Heartbeat");
    string str2("abcde");
    size_t iPos = 0;

    cout << "The string to search is '" << str1.c_str() << "'"
         << endl;

    // find the first instance in str1 of any characters in str2
    iPos = str1.find_first_of (str2, 0);  // 0 is default position
```

```
    cout << "Element in '" << str2.c_str() << "' found at position "
         << iPos << endl;

    // start looking in the third position...
    iPos = str1.find_first_of (str2, 2);

    cout << "Element in '" << str2.c_str() << "' found at position "
         << iPos << endl;

    // use an array of the element type as the set of elements to
    // search for; look for anything after the fourth position
    char achVowels[] = {'a', 'e', 'i', 'o', 'u'};
    iPos = str1.find_first_of (achVowels, 4, sizeof(achVowels));

    cout << "Element in '";
    for (int i = 0; i < sizeof (achVowels); i++)
        cout << achVowels[i];
    cout << "' found at position " << iPos << endl;

    // use a string literal to specify the set of elements
    char szVowels[] = "aeiou";
    iPos = str1.find_first_of (szVowels, 0);  // 0 is default position

    cout << "Element in '" << szVowels << "' found at position "
         << iPos << endl;

    // look for a specific character beginning in the third position
    iPos = str1.find_first_of ('e', 2);

    cout << "'e' found at position " << iPos << endl;
}
```

Output

```
The string to search is 'Heartbeat'
Element in 'abcde' found at position 1
Element in 'abcde' found at position 2
Element in 'aeiou' found at position 6
Element in 'aeiou' found at position 1
'e' found at position 6
```

Requirements

Header: <string>

See Also

Concepts

Standard Template Library Samples

basic_string::size and basic_string::resize

Illustrates how to use the basic_string::size and basic_string::resize Standard Template Library (STL) functions in Visual C++.

```
size_type size( ) const;
   void resize(
      size_type n,
      E c = E( )
   );
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **basic_string::size** STL function returns the length of the sequence. The **basic_string::resize** STL function changes the size to the length specified by the first parameter. If the sequence is made longer, the function appends elements with the value of the second parameter. This value defaults to a null. The output of the sample code shows spaces for the null characters. **operator<<** reads the size of string and outputs each character in the string one at a time.

Example

```
// size.cpp
// compile with: /EHsc
//
// Functions:
//    size()
//    resize() ; Defined in header xstring which is included indirectly.
//////////////////////////////////////////////////////////////////////
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string TestString = "111112222233333444445555";
   cout << "[" << TestString << "]" << endl
        << "size: " << TestString.size() << endl
        << endl;

   TestString.resize(5);
   cout << "[" << TestString << "]" << endl
        << "size: " << TestString.size() << endl
        << endl;

   TestString.resize(10);
   cout << "[" << TestString << "]" << endl
        << "size: " << TestString.size() << endl
        << endl;

   TestString.resize(15,'6');
   cout << "[" << TestString << "]" << endl
        << "size: " << TestString.size() << endl;
}
```

Sample Output

```
[111112222233333444445555]
size: 25

[11111]
```

```
size: 5

[11111     ]
size: 10

[11111     66666]
size: 15
```

Requirements

Header: <string>

See Also

Concepts

Standard Template Library Samples

binary_function Structure

Illustrates how to use the binary_function structure in the Standard Template Library (STL) in Visual C++.

```
template<class _A1, class _A2, class _R>
   struct binary_function
   {
      typedef _A1 first_argument_type;
      typedef _A2 second_argument_type;
      typedef _R result_type;
   };
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **binary_function**<A,B,C> class is used as a base class to allow the user to easily define binary operator functions that take data types A and B as arguments and return data type C objects.

Example

```
// binfunc.cpp
// compile with: /EHsc
//
// Structure used: binary_function<A,B,C> - base
//                 class used to create operator
//                 functions taking data types A
//                 and B and returning data type C.

#include <functional>
#include <iostream>

using namespace std ;

class binary_test : public binary_function<binary_test &,int,float>
{
public:
  float value;
  binary_test(){value=10.0;}
  binary_test(float x){value=x;}
  result_type operator<<(second_argument_type arg2);
};

binary_test::result_type
binary_test::operator<<(binary_test::second_argument_type arg2)
{
  value = (float)(((int)value) << arg2);
  cout << "New value after shift is " << value << endl;
  return value;
}

int main(void)
{
  binary_test item;

  cout << "Begin" << endl;
  item = item << 2;
}
```

Output

```
Begin
New value after shift is 40
```

Requirements

Header: < functional >

See Also

Concepts

Standard Template Library Samples

count (STL Samples)

Illustrates how to use the count Standard Template Library (STL) function in Visual C++.

```
template<class InputIterator, class T> inline
   size_t count(
      InputIterator First,
      InputIterator Last,
      const T& Value
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **count** algorithm counts the number of elements in the range [*First*, *Last* +1) that match *Value* and returns the number of matching elements.

Example

```
// count.cpp
// compile with: /EHsc
//
// Functions:
//
//    count  - Count items in a range that match a value.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <string>
#include <vector>

using namespace std;


int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of strings
    typedef vector<string > StringVector ;

    //Define an iterator for template class vector of strings
    typedef StringVector::iterator StringVectorIt ;

    StringVector NamesVect(VECTOR_SIZE) ;   //vector containing names

    string value("Sea") ;  // stores the value used
                           // to count matching elements

    StringVectorIt start, end, it ;

    ptrdiff_t result = 0 ;   // stores count of elements
                             // that match value.

    // Initialize vector NamesVect
```

```
    NamesVect[0] = "She" ;
    NamesVect[1] = "Sells" ;
    NamesVect[2] = "Sea" ;
    NamesVect[3] = "Shells" ;
    NamesVect[4] = "by" ;
    NamesVect[5] = "the" ;
    NamesVect[6] = "Sea" ;
    NamesVect[7] = "Shore" ;

    start = NamesVect.begin() ;   // location of first
                                  // element of NamesVect

    end = NamesVect.end() ;       // one past the location
                                  // last element of NamesVect

    // print content of NamesVect
    cout << "NamesVect { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // Count the number of elements in the range [first, last +1)
    // that match value.
    result = count(start, end, value) ;

    // print the count of elements that match value
    cout << "Number of elements that match \"Sea\" = "
        << result << endl  ;
}
```

Output

```
NamesVect { She Sells Sea Shells by the Sea Shore  }

Number of elements that match "Sea" = 2
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

count_if (STL Samples)

Illustrates how to use the count_if Standard Template Library (STL) function in Visual C++.

```
template<class InputIterator, class Predicate> inline
   size_t count_if(
      InputIterator First,
      InputIterator Last,
      Predicate P
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **count_if** algorithm counts the number of elements in the range [*First*, *Last*) that cause the predicate to return **true** and returns the number of elements for which the predicate was true.

Example

```
// countif.cpp
// compile with: /EHsc
//
// Functions:
//
//   count_if  - Count items in a range that satisfy a predicate.
//
//   begin     - Returns an iterator that points to the first element in
//               a sequence.
//
//   end       - Returns an iterator that points one past the end of a
//               sequence.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <string>
#include <vector>

using namespace std;


// Return true if string str starts with letter 'S'
int MatchFirstChar( const string& str)
{
    string s("S") ;
    return s == str.substr(0,1) ;
}

int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of strings
    typedef vector<string > StringVector ;

    //Define an iterator for template class vector of strings
```

```
    typedef StringVector::iterator StringVectorIt ;

    StringVector NamesVect(VECTOR_SIZE) ;   //vector containing names

    StringVectorIt start, end, it ;

    ptrdiff_t result = 0 ;   // stores count of elements
                             // that match value.

    // Initialize vector NamesVect
    NamesVect[0] = "She" ;
    NamesVect[1] = "Sells" ;
    NamesVect[2] = "Sea" ;
    NamesVect[3] = "Shells" ;
    NamesVect[4] = "by" ;
    NamesVect[5] = "the" ;
    NamesVect[6] = "Sea" ;
    NamesVect[7] = "Shore" ;

    start = NamesVect.begin() ;   // location of first
                                  // element of NamesVect

    end = NamesVect.end() ;       // one past the location
                                  // last element of NamesVect

    // print content of NamesVect
    cout << "NamesVect { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // Count the number of elements in the range [first, last +1)
    // that start with letter 'S'
    result = count_if(start, end, MatchFirstChar) ;

    // print the count of elements that start with letter 'S'
    cout << "Number of elements that start with letter \"S\" = "
        << result << endl  ;
}
```

Output

```
NamesVect { She Sells Sea Shells by the Sea Shore  }

Number of elements that start with letter "S" = 6
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

deque::assign and deque::swap

Illustrates how to use the deque::assign and deque::swap Standard Template Library (STL) functions in Visual C++.

```
void assign(
   const_iterator First,
   const_iterator Last
);
void assign(
   size_type n,
   const T& x = T( )
);
void swap(
   deque& Dq
);
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The first **assign** member function replaces the sequence controlled by ***this** with the sequence [*First*, *Last*). The second **assign** member function replaces the sequence controlled by ***this** with a repetition of *n* elements of value *x*. The **swap** member function swaps the contents between ***this** and *Dq*.

Example

```
// deque.cpp
// compile with: /EHsc
//
// Functions:
//
//    assign
//    swap
//    begin
//    end

#include <iostream>
#include <deque>

using namespace std;


typedef deque<char >  CHARDEQUE;
void print_contents (CHARDEQUE  deque, char*);

int main()
{
    //create a  with  3 A's
    CHARDEQUE  a(3,'A');

    //create b with 4 B's.
    CHARDEQUE  b(4,'B');

    //print out the contents
    print_contents (a,"a");
    print_contents (b,"b");

    //swap a and b
    a.swap(b);
    print_contents (a,"a");
    print_contents (b,"b");
```

```
    //swap it back
        a.swap(b);
    print_contents (a,"a");
    print_contents (b,"b");

    //assign the contents of b to a
    a.assign(b.begin(),b.end());
    print_contents (a,"a");

    //assign the first two items of b to a
    a.assign(b.begin(),b.begin()+2);
    print_contents (a,"a");

    //assign 3 'Z's to a
    a.assign(3,'Z');
    print_contents (a,"a");

    }

//function to print the contents of deque
void print_contents (CHARDEQUE  deque, char *name)
{
    CHARDEQUE::iterator pdeque;

    cout <<"The contents of "<< name <<" : ";

        for(pdeque = deque.begin();
        pdeque != deque.end();
        pdeque++)
    {
        cout << *pdeque <<" " ;
    }
        cout<<endl;
}
```

Output

```
The contents of a : A A A
The contents of b : B B B B
The contents of a : B B B B
The contents of b : A A A
The contents of a : A A A
The contents of b : B B B B
The contents of a : B B B B
The contents of a : B B
The contents of a : Z Z Z
```

Requirements

Header: <deque>

See Also

Concepts

Standard Template Library Samples

deque::begin and deque::end

Illustrates how to use the deque::begin and deque::end Standard Template Library (STL) functions in Visual C++.

```
const_iterator begin( ) const;
   iterator begin( );
const_iterator end( ) const;
   iterator end( );
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **begin** member function returns a random-access iterator that points at the first element of the sequence or just beyond the end of an empty sequence. The **end** member function returns a random-access iterator that points just beyond the end of the sequence.

Example

```
// begin.cpp
// compile with: /EHsc
//
// Functions:
//
//    begin()
//    end()

#include <iostream>
#include <deque>

using namespace std;

typedef deque<int >  INTDEQUE;

int main()
{

    // Create A and fill it with elements 1,2,3,4 and 5
    // using push_back function

    INTDEQUE  A;
    A.push_back(1);
    A.push_back(2);
    A.push_back(3);
    A.push_back(4);
    A.push_back(5);

    // Print the contents of A using iterator
    // and functions begin() and end()

     INTDEQUE::iterator pi;

    for(pi= A.begin();  pi !=A.end(); pi++)
    {
        cout << *pi <<" " ;
    }
        cout<<endl;
}
```

Output

```
1 2 3 4 5
```

Requirements

Header: <deque>

See Also

Concepts

Standard Template Library Samples

deque::erase and deque::clear

Illustrates how to use the deque::erase and deque::clear Standard Template Library (STL) functions in Visual C++.

```
iterator erase(
   iterator Iter
);
iterator erase(
   iterator First,
   iterator Last
); void clear( ) const;
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The first **erase** member function removes the element of the container pointed to by *Iter*. The second **erase** member function removes the elements of the container in the range [*First*, *Last*). Both return an iterator that designates the first element remaining beyond any elements removed, or **end** if no such element exists. Removing *N* elements causes *N* destructor calls and an assignment for each of the elements between the insertion point and the nearer end of the sequence. Removing an element at either end invalidates only iterators and references that designate the erased elements. Otherwise, erasing an element invalidates all iterators and references. The clear member function calls **erase**(**begin**, **end**).

Example

```
// deque_erase_clear.cpp
// compile with: /EHsc
//
// Functions:
//
//    erase
//    clear
//    begin
//    end
//////////////////////////////////////////////////////////////////////

#include <iostream>
#include <deque>

using namespace std;


typedef deque<int >  INTDEQUE;
void print_contents (INTDEQUE  deque);

int main()
{
    // create a and with elements 1,2,3,4 and 5
    INTDEQUE  a;
    a.push_back(1);
    a.push_back(2);
    a.push_back(3);
    a.push_back(4);
    a.push_back(5);

    //print the contents
    print_contents (a);

    //  erase the second element
    a.erase(a.begin()+1);
```

```
    print_contents (a);

    //erase the last two elements
    a.erase(a.end()-2,a.end());
    print_contents (a);

    //clear a
    a.clear();
    print_contents (a);
}

void print_contents (INTDEQUE  deque) {
    INTDEQUE::iterator pdeque;

    cout <<"The output is: ";

        for(pdeque = deque.begin();
        pdeque != deque.end();
        pdeque++)
    {
        cout << *pdeque <<" " ;
    }
        cout<<endl;
}
```

Output

```
The output is: 1 2 3 4 5
The output is: 1 3 4 5
The output is: 1 3
The output is:
```

Requirements

Header: <deque>

See Also

Concepts

Standard Template Library Samples

deque::front and deque::back

Illustrates how to use the deque::front and deque::back Standard Template Library (STL) functions in Visual C++.

```
reference front( );
   const_reference front( ) const;
reference back( );
   const_reference back( ) const;
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **front** member function returns a reference to the first element of the controlled sequence, which must be nonempty. The **back** member function returns a reference to the last element of the controlled sequence, which must be nonempty.

Example

```
// front.cpp
// compile with: /EHsc
#include <iostream>
#include <deque>
using namespace std;
typedef deque<char >  CHARDEQUE;
void print_contents (CHARDEQUE  deque, char*);

int main()
{
   //create a  with  A, B, C and D
   CHARDEQUE  a;
   a.push_back('A');
   a.push_back('B');
   a.push_back('C');
   a.push_back('D');

   //print out the contents
   print_contents (a,"a");
   cout <<"The first element of a is " <<a.front() <<endl;
   cout <<"The last element of a is " <<a.back() <<endl;

   // modify first and last elements using reference, front, and back
   CHARDEQUE::reference reffront=a.front();
   CHARDEQUE::reference refback=a.back();

   reffront='X';
   refback='Y';
   print_contents (a,"a");
}

// print the contents of deque
void print_contents (CHARDEQUE  deque, char *name)
{
   CHARDEQUE::iterator pdeque;
   cout << "The contents of " << name << ":";
   for (pdeque = deque.begin(); pdeque != deque.end(); pdeque++)
      cout << " " << *pdeque;

   cout<<endl;
}
```

Output

```
The contents of a: A B C D
The first element of a is A
The last element of a is D
The contents of a: X B C Y
```

Requirements

Header: <deque>

See Also

Concepts

Standard Template Library Samples

deque::insert (STL Samples)

Illustrates how to use the deque::insert Standard Template Library (STL) function in Visual C++.

```
iterator insert(
   iterator Iter,
   const T& x = T( )
);
void insert(
   iterator Iter,
   size_type n,
   const T& x
);
void insert(
   iterator Iter,
   const_iterator First,
   const_iterator Last
);
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

Each function inserts a sequence specified by the remaining operands after the element pointed to by *Iter* in the container. The first member function inserts a single element with value *x* and returns an iterator that points to the newly inserted element. The second member function inserts a repetition of *n* elements of value *x*. The third member function inserts the sequence [*First*, *Last*).

Example

```
// insert.cpp
// compile with: /EHsc

#include <iostream>
#include <deque>

using namespace std;


typedef deque<char >  CHARDEQUE;
void print_contents (CHARDEQUE  deque);

int main()
{
    //create a with 3 A's
    CHARDEQUE  a(3,'A');

    //create b with 2 B's.
    CHARDEQUE  b(2,'B');

    //print out the contents
    print_contents (a);
    print_contents (b);

    //insert 'X' to the beginning of a
    a.insert(a.begin(),'X');
    print_contents (a);

    //insert 'Y' to the end of a
    a.insert(a.end(),'Y');
```

```
    print_contents (a);

    //inset 3 'Z's to one item before the end of a
    a.insert(a.end()-1,3,'Z');
    print_contents (a);

    //insert to the end of a from b
    a.insert(a.end(),b.begin(),b.end());
    print_contents (a);

}

//function to print the contents of deque
void print_contents (CHARDEQUE  deque)
{
    CHARDEQUE::iterator pdeque;

    cout <<"The output is: ";

    for(pdeque = deque.begin();
        pdeque != deque.end();
        pdeque++)
    {
        cout << *pdeque <<" " ;
    }
        cout<<endl;
}
```

Output

```
The output is: A A A
The output is: B B
The output is: X A A A
The output is: X A A A Y
The output is: X A A A Z Z Z Y
The output is: X A A A Z Z Z Y B B
```

Requirements

Header: <deque>

See Also

Concepts

Standard Template Library Samples

deque::operator[] and deque::at

Illustrates how to use the deque::operator[] and deque::at Standard Template Library (STL) functions in Visual C++.

```
const_reference operator[](
   size_type Pos
) const;
reference operator[](
   size_type Pos
);
const_reference operator[](
   difference_type _N
) const;
reference operator[](
   difference_type _N
) const;
const_reference at(
   size_type Pos
) const;
reference at(
   size_type Pos
); bool empty( ) const;
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **operator[]** member function returns a reference to the element of the controlled sequence at position *Pos*. If that position is invalid, the behavior is undefined. The **at** member function returns a reference to the element of the controlled sequence at position *Pos*. If that position is invalid, the function throws an object of class **out_of_range**. The **empty** member function returns **true** for an empty controlled sequence.

Example

```
// operator.cpp
// compile with: /EHsc
//
// Functions:
//    operator[]
//    at
//    empty
//    push_back
//    begin
//    end

#include <iostream>
#include <deque>

using namespace std;


typedef deque<char >  CHARDEQUE;
void print_contents (CHARDEQUE  deque, char*);

int main()
{
    //create an empty deque a
    CHARDEQUE  a;

    //check whether it is empty
```

```
    if(a.empty())
        cout<<"a is empty"<<endl;
    else
        cout<<"a is not empty"<<endl;

    //inset A, B, C and D  to a
    a.push_back('A');
    a.push_back('B');
    a.push_back('C');
    a.push_back('D');

    //check again whether a is empty
    if(a.empty())
        cout<<"a is empty"<<endl;
    else
        cout<<"a is not empty"<<endl;

    //print out the contents

    print_contents (a,"a");

    cout <<"The first element of a is " <<a[0] <<endl;
    cout <<"The first element of a is " <<a.at(0) <<endl;

    cout <<"The last element of a is " <<a[a.size()-1] <<endl;
    cout <<"The last element of a is " <<a.at(a.size()-1) <<endl;
}

//function to print the contents of deque
void print_contents (CHARDEQUE  deque, char *name)
{
    CHARDEQUE::iterator pdeque;

    cout <<"The contents of "<< name <<" :";

    for(pdeque = deque.begin();
        pdeque != deque.end();
        pdeque++)
        {
            cout <<" "  << *pdeque;
        }
    cout<<endl;
}
```

Output

```
a is empty
a is not empty
The contents of a : A B C D
The first element of a is A
The first element of a is A
The last element of a is D
The last element of a is D
```

Requirements

Header: < deque>

See Also

Concepts

Standard Template Library Samples

deque::operator== and deque::operator<

Illustrates how to use the deque::operator== and deque::operator< Standard Template Library (STL) functions in Visual C++.

```
template<class T, class A>
   bool operator==(
      const deque <T, A>& Left,
      const deque <T, A>& Right
   );
template<class T, class A>
   bool operator<(
      const deque <T, A>& Left,
      const deque <T, A>& Right
   );
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The first template function overloads **operator==** to compare two objects of template class deque. The function returns *Left*.**size** == *Right*.**size** && **equal**(*Left*.**begin**, *Left*.**end**, *Right*.**begin**). For equality, the number of elements must be equal in both deque objects. The second template function overloads **operator<** to compare two objects of template class deque. The function returns: **lexicographical_compare**(*Left*.**begin**, *Left*.**end**, *Right*.**begin**, *Right*.**end**). Because **lexicographical_compare** is used, the number of elements does not matter while using **operator<**. In the sample code, adding a line of code while creating the *b* object, such as *b*.**push_front**('*D*');, will make *b* greater than *a*.

Example

```
// deque_operators.cpp
// compile with: /EHsc
//
// Functions:
//    ==
//    <

#include <iostream>
#include <deque>

using namespace std;

typedef deque<char >  CHARDEQUE;
void print_contents (CHARDEQUE  deque, char*);

int main()
{
    //create a  with  3 A's
    CHARDEQUE  a(3,'A');
    a.push_front('C');

    //create b with 4 B's.
    CHARDEQUE  b(6,'B');

    //print out the contents
    print_contents (a,"a");
    print_contents (b,"b");

    //compare a and b
    if (a==b)
        cout <<"a is equal to b"<<endl;
    else if(a<b)
```

```
            cout <<"a is less than b"<<endl;
    else
        cout <<"a is greater than b" <<endl;

    //assign the contents of b to a
    a.assign(b.begin(),b.end());
    print_contents (a,"a");
    print_contents (b,"b");

    //compare a and b again
    if (a==b)
        cout <<"a is equal to b"<<endl;
    else if(a<b)
            cout <<"a is less than b"<<endl;
    else
        cout <<"a is greater than b" <<endl;

}

//function to print the contents of deque
void print_contents (CHARDEQUE  deque, char *name)
{
    CHARDEQUE::iterator pdeque;

    cout <<"The contents of "<< name <<" : ";

        for(pdeque = deque.begin();
        pdeque != deque.end();
        pdeque++)
    {
        cout << *pdeque <<" " ;
    }
        cout<<endl;
}
```

Output

```
The contents of a : C A A A
The contents of b : B B B B B B
a is greater than b
The contents of a : B B B B B B
The contents of b : B B B B B B
a is equal to b
```

Requirements

Header: <deque>

See Also

Concepts

Standard Template Library Samples

deque::push_back and deque::pop_back

Illustrates how to use the deque::push_back and deque::pop_back Standard Template Library (STL) functions in Visual C++.

```
void push_back(
   const T& x
);
void pop_back( );
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **push_back** function inserts an element with value *x* at the end of the container deque. The **pop_back** function removes the last element of the container **deque**, which must be nonempty.

Example

```
// pushback.cpp
// compile with: /EHsc

#include <iostream>
#include <deque>

using namespace std;


typedef deque<int >  INTDEQUE;
void printcontents (INTDEQUE  deque);

int main()
{

    INTDEQUE  dequetest;

    dequetest.push_back(1);
    dequetest.push_back(2);
    dequetest.push_back(3);
    printcontents (dequetest);
    dequetest.pop_back();
    printcontents (dequetest);
    dequetest.pop_back();
    printcontents (dequetest);
   }

   //function to print the contents of deque
   void printcontents (INTDEQUE  deque)
   {
       INTDEQUE::iterator pdeque;

       cout <<"The output is:"<<endl;

           for(pdeque = deque.begin();
           pdeque != deque.end();
           pdeque++)
       {
           cout << *pdeque <<endl ;
       }
   }
```

Output

```
The output is:
1
2
3
The output is:
1
2
The output is:
1
```

Requirements

Header: <deque>

See Also

Concepts

Standard Template Library Samples

deque::push_front and deque::pop_front

Illustrates how to use the deque::push_front and deque::pop_front Standard Template Library (STL) functions in Visual C++.

```
void push_front(
   const T& x
);
void pop_front( );
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **deque::push_front** function inserts an element with value *x* at the beginning of the deque container. The **deque::pop_front** function removes the first element of the deque container, which must be nonempty.

Example

```
// pushfron.cpp
// compile with: /EHsc

#include <iostream>
#include <deque>

using namespace std;


typedef deque<int >  INTDEQUE;
void printcontents (INTDEQUE  deque);

int main()
{
    INTDEQUE  dequetest;

    dequetest.push_front(1);
    dequetest.push_front(2);
    dequetest.push_front(3);
    printcontents (dequetest);
    dequetest.pop_front();
    printcontents (dequetest);
    dequetest.pop_front();
    printcontents (dequetest);
}

//function to print the contents of deque
void printcontents (INTDEQUE  deque)
{
    INTDEQUE::iterator pdeque;

    cout <<"The output is:"<<endl;

        for(pdeque = deque.begin();
        pdeque != deque.end();
        pdeque++)
    {
        cout << *pdeque <<endl ;
    }
}
```

Output

```
The output is:
3
2
1
The output is:
2
1
The output is:
1
```

Requirements

Header: <deque>

See Also

Concepts

Standard Template Library Samples

deque::rbegin and deque::rend

Illustrates how to use the deque::rbegin and deque::rend Standard Template Library (STL) functions in Visual C++.

```
const_reverse_iterator rbegin( ) const;
reverse_iterator rbegin( );
const_reverse_iterator rend( ) const;
reverse_iterator rend( );
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **rbegin** member function returns a reverse iterator that points just beyond the end of the controlled sequence. Therefore, it designates the beginning of the reverse sequence. The **rend** member function returns a reverse iterator that points at the first element of the sequence, or just beyond the end of an empty sequence. Therefore, it designates the end of the reverse sequence.

Example

```
// rbegin.cpp
// compile with: /EHsc
//
// Functions:
//    rbegin
//    rend
//    push_back

#include <iostream>
#include <deque>

using namespace std;

typedef deque<int >  INTDEQUE;

int main()
{

    // Create A and fill it with elements 1,2,3,4 and 5
    // using push_back function

    INTDEQUE  A;
    A.push_back(1);
    A.push_back(2);
    A.push_back(3);
    A.push_back(4);
    A.push_back(5);

    // Now print the contents in reverse order using reverse_iterator
    // and functions rbegin() and rend()

    INTDEQUE::reverse_iterator rpi;
    cout << "Contents in reverse order:";
    for(rpi= A.rbegin(); rpi !=A.rend(); rpi++)
        cout << " " << *rpi;
    cout<<endl;
}
```

Output

```
Contents in reverse order: 5 4 3 2 1
```

Requirements

Header: <deque>

See Also

Concepts

Standard Template Library Samples

deque::size and deque::resize

Illustrates how to use the deque::size and deque::resize Standard Template Library (STL) functions in Visual C++.

```
size_type size( ) const;
void resize(
   size_type n,
   T x = T( )
);
size_type max_size( ) const;
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **size** member function returns the length of the controlled sequence. The **resize** member function ensures that **size** henceforth returns *n*. If it must make the controlled sequence longer, it appends elements with value *x*. If no value is supplied, the default value depends upon the type. For example, if it is a deque of **char**s, the default is a blank. If it is a deque of **int**s, the default is zero. The **max_size** member function returns the length of the longest sequence that the object can control.

Example

```
// sizeresi.cpp
// compile with: /EHsc
//
// Functions:
//    size
//    resize
//    max_size
//    begin
//    end

#include <iostream>
#include <deque>

using namespace std;

typedef deque<char >  CHARDEQUE;
void print_contents (CHARDEQUE  deque, char*);

int main()
{
    //create a  with  A, B, C and D
    CHARDEQUE  a;
    a.push_back('A');
    a.push_back('B');
    a.push_back('C');
    a.push_back('D');

    //print out the contents

    print_contents (a,"a");
    cout <<"max_size of a is " <<a.max_size() <<endl;
    cout <<"size of a is " <<a.size() <<endl;

    //let us increase the size to 10
    // and set the new elements to be 'X'
    a.resize(10,'X');
    print_contents (a,"a");
    cout <<"size of a is " <<a.size() <<endl;
```

```
    //let us resize it to 5
    a.resize(5);
    print_contents (a,"a");
    cout <<"size of a is " <<a.size() <<endl;
    cout <<"max_size of a is still " <<a.max_size() <<endl;

    }

//function to print the contents of deque
void print_contents (CHARDEQUE  deque, char *name)
{
    CHARDEQUE::iterator pdeque;

    cout << "The contents of " << name << " :";
    for(pdeque = deque.begin();
        pdeque != deque.end();
        pdeque++)
    {
        cout << " " << *pdeque ;
    }
    cout<<endl;
}
```

Sample Output

The following output is for x86.

```
The contents of a : A B C D
max_size of a is 4294967295
size of a is 4
The contents of a : A B C D X X X X X X
size of a is 10
The contents of a : A B C D X
size of a is 5
max_size of a is still 4294967295
```

Requirements

Header: <deque>

See Also

Concepts

Standard Template Library Samples

distance (STL Samples)

Illustrates how to use the distance Standard Template Library (STL) function in Visual C++.

```
template<class Init, class Dist>
   Dist distance(
      InIt first,
      InIt last,
      Dist& n
   );
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

This function returns the distance between two iterators by determining how many times the first iterator would need to be incremented until it was equal to the last iterator.

Example

```
// distance.cpp
// compile with: /EHsc

#pragma warning (disable:4786)
#include <iostream>
#include <vector>
#include <iterator>
#include <string>

using namespace std;

typedef vector<string > VTRLIST;

int main() {
    VTRLIST Vector;
    VTRLIST::iterator iVector;
    VTRLIST::difference_type dTheDiff;

    Vector.push_back("A1");
    Vector.push_back("B2");
    Vector.push_back("C3");
    Vector.push_back("D4");
    Vector.push_back("E5");
    Vector.push_back("F6");
    Vector.push_back("G7");

    // Print out the list
    iVector=Vector.begin();
    cout << "The list is: ";
    for (int i = 0; i < 7 ; i++, iVector++)
        cout << *iVector  << "  ";

    // Initialize the iterator the first element"
    iVector=Vector.begin();
    cout << "\n\nAdvance to the 3rd element." << endl;
    advance( iVector, 2);
    cout << "The element is " << *iVector << endl;
    dTheDiff = distance( Vector.begin(), iVector);
    cout << "The distance from the beginning is " << dTheDiff << endl;

    cout << "Calculate it in reverse order " << endl;
```

```
    dTheDiff = distance( iVector, Vector.begin());
    cout << "The distance is " << dTheDiff << endl;

    cout << "\nUse distance() to count from the 3rd element to the end."
         << endl;
    dTheDiff = distance( iVector, Vector.end());

    // Note that end() returns one past the end of the sequence
    cout << "The distance is " << dTheDiff << endl;

    cout <<"\nUse distance() to count the total length." << endl;
    dTheDiff = distance( Vector.begin(), Vector.end() );
    cout << "The total distance is " << dTheDiff << endl;
}
```

Output

```
The list is: A1  B2  C3  D4  E5  F6  G7

Advance to the 3rd element.
The element is C3
The distance from the beginning is 2
Calculate it in reverse order
The distance is -2

Use distance() to count from the 3rd element to the end.
The distance is 5

Use distance() to count the total length.
The total distance is 7
```

Requirements

Header: <iterator>

See Also

Concepts

Standard Template Library Samples

equal_to

A binary predicate that tests whether a value of a specified type is equal to another value of that type.

```
template<class Type>
   struct equal_to : public binary_function<Type, Type, bool>
   {
      bool operator()(
         const Type& _Left,
         const Type& _Right
      ) const;
   };
```

Parameters

_Left

The left operand of type **Type** in the equality to be tested.

_Right

The right operand of type **Type** in the equality to be tested.

Return Value

true if *_Left* equals *_Right*; **false** if *_Left* does not equal *_Right*.

Remarks

The objects of type **Type** must be equality comparable. This requires that the **operator==** defined on the set of objects satisfies the mathematical properties of an equivalence relation All the built-in numeric and pointer types satisfy this requirement.

Example

```
// functional_equal_to.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
   vector <double> v1, v2, v3 ( 6 );
   vector <double>::iterator Iter1, Iter2, Iter3;

   int i;
   for ( i = 0 ; i <= 5 ; i+=2 )
   {
      v1.push_back( 2.0 *i );
      v1.push_back( 2.0 * i + 1.0 );
   }

   int j;
   for ( j = 0 ; j <= 5 ; j+=2 )
   {
      v2.push_back( - 2.0 * j );
      v2.push_back( 2.0 * j + 1.0 );
   }

   cout << "The vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;
```

```
    cout << "The vector v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
       cout << *Iter2 << " ";
    cout << ")" << endl;

    // Testing for the element-wise equality between v1 & v2
    transform ( v1.begin( ),  v1.end( ), v2.begin( ), v3.begin ( ),
       equal_to<double>( ) );

    cout << "The result of the element-wise equal_to comparison\n"
       << "between v1 & v2 is: ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
       cout << *Iter3 << " ";
    cout << ")" << endl;
 }
```

Output

```
The vector v1 = ( 0 1 4 5 8 9 )
The vector v2 = ( -0 1 -4 5 -8 9 )
The result of the element-wise equal_to comparison
between v1 & v2 is: ( 1 1 0 1 0 1 )
```

See Also

Other Resources

<functional> Members

exp, log, and log10

Illustrates how to use the exp, log, and log10 Standard Template Library (STL) functions in Visual C++.

```
template<class T>
   valarray<T> exp(
      const valarray<T>& x
   );
template<class T>
   valarray<T> log(
      const valarray<T>& x
   );
template<class T>
   valarray<T> log10(
      const valarray<T>& x
   );
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

Example

```
// explog10.cpp
// compile with: /EHsc
#include <iostream>   // for i/o functions
#include <valarray>   // for valarray
#include <math.h>    // for exp(), log(), and log10()
using namespace std;

#define ARRAY_SIZE  3   // array size

typedef valarray<double> DB_VARRAY;

int main() {
   // Set val_array to contain values 1, 10, 100 for the following test.
   DB_VARRAY val_array(ARRAY_SIZE);
   int i;
   for (i = 0; i < ARRAY_SIZE; i++)
      val_array[i] = pow((double)10, i);

   // Display the size of val_array.
   cout << "Size of val_array = " << val_array.size() << endl;

   // Display the content of val_array before calling exp, log, and
   // log10 functions.
   cout << "val_array values:";
   for (i = 0; i < ARRAY_SIZE; i++)
      cout << "  " << val_array[i];
   cout << endl;

   // rvalue_array to hold the return value from calling the exp,
   // log, and log10 functions.
   DB_VARRAY rvalue_array;

   // ----------------------------------------------------------------
   // exp() - display the content of rvalue_array
   // ----------------------------------------------------------------
   rvalue_array = exp(val_array);
   cout << "After calling exp():";
   for (i = 0; i < ARRAY_SIZE; i++)
```

```
      cout << "   " << rvalue_array[i];
   cout << endl;

   // ----------------------------------------------------------------
   // log() - display the content of rvalue_array
   // ----------------------------------------------------------------
   rvalue_array = log(val_array);
   cout << "After calling log():";
   for (i = 0; i < ARRAY_SIZE; i++)
      cout << "  " << rvalue_array[i];
   cout << endl;

   // ----------------------------------------------------------------
   // log10() - display the content of rvalue_array
   // ----------------------------------------------------------------
   rvalue_array = log10(val_array);
   cout << "After calling log10():";
   for (i = 0; i < ARRAY_SIZE; i++)
      cout << "  " << rvalue_array[i];
   cout << endl;
}
```

Output

```
Size of val_array = 3
val_array values:  1  10  100
After calling exp():   2.71828   22026.5   2.68812e+043
After calling log():  0  2.30259  4.60517
After calling log10():  0  1  2
```

Requirements

Header: <valarray>

See Also

Concepts

Standard Template Library Samples

find (STL Samples)

Illustrates how to use the find Standard Template Library (STL) function in Visual C++.

```
template<class InputIterator, class T> inline
   InputIterator find(
      InputIterator First,
      InputIterator Last,
      const T& Value
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **find** algorithm locates the first element in the range [*First*, *Last*) that matches *Value* and returns the iterator positioned at the first matching element, or *Last* if no such element exists.

Example

```
// find.cpp
// compile with: /EHsc

// disable warning C4786: symbol greater than 255 characters,
// okay to ignore
#pragma warning(disable: 4786)

#include <algorithm>
#include <iostream>

using namespace std;

int main() {
    const int ARRAY_SIZE = 8 ;
    int IntArray[ARRAY_SIZE] = { 1, 2, 3, 4, 4, 5, 6, 7 } ;

    int *location ;   // stores the position of the first
                      // matching element.

    int i ;

    int value = 4 ;

    // print content of IntArray
    cout << "IntArray { " ;
    for(i = 0; i < ARRAY_SIZE; i++)
        cout << IntArray[i] << ", " ;
    cout << " }" << endl ;

    // Find the first element in the range [first, last)
    // that matches value.
    location = find(IntArray, IntArray + ARRAY_SIZE, value) ;

    //print the matching element if any was found
    if (location != IntArray + ARRAY_SIZE)  // matching element found
        cout << "First element that matches " << value
             << " is at location " << location - IntArray << endl;
    else                                    // no matching element was
                                            // found
        cout << "The sequence does not contain any elements"
             << " with value " << value << endl ;
```

```
}
```

Output

```
IntArray { 1, 2, 3, 4, 4, 5, 6, 7,  }
First element that matches 4 is at location 3
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

find_if (STL Samples)

Illustrates how to use the find_if Standard Template Library (STL) function in Visual C++.

```
template<class InputIterator, class T, class Predicate> inline
   InputIterator find_if(
      InputIterator First,
      InputIterator Last,
      Predicate Predicate
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **find_if** algorithm locates the first element in the range [*First*, *Last*) that causes the predicate to return **true** and returns the iterator positioned at that element, or *Last* if no such element was found.

Example

```
// findif.cpp
// compile with: /EHsc

// disable warning C4786: symbol greater than 255 characters,
// okay to ignore
#pragma warning(disable: 4786)

#include <algorithm>
#include <iostream>

using namespace std;

// returns true if n is an odd number
int IsOdd( int n)
{
    return n % 2 ;
}

int main()
{
    const int ARRAY_SIZE = 8 ;
    int IntArray[ARRAY_SIZE] = { 1, 2, 3, 4, 4, 5, 6, 7 } ;
    int *location ;   // stores the position of the first
                      // element that is an odd number
    int i ;

        // print content of IntArray
    cout << "IntArray { " ;
    for(i = 0; i < ARRAY_SIZE; i++)
        cout << IntArray[i] << ", " ;
    cout << " }" << endl ;

    // Find the first element in the range [first, last -1 ]
    // that is an odd number
    location = find_if(IntArray, IntArray + ARRAY_SIZE, IsOdd) ;

    // print the location of the first element
    // that is an odd number
    if (location != IntArray + ARRAY_SIZE)  // first odd element found
        cout << "First odd element " << *location
             << " is at location " << location - IntArray << endl;
```

```
    else         // no odd numbers in the range
        cout << "The sequence does not contain any odd numbers"
             << endl ;
}
```

Output

```
IntArray { 1, 2, 3, 4, 4, 5, 6, 7,  }
First odd element 1 is at location 0
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

for_each (STL Samples)

Illustrates how to use the for_each Standard Template Library (STL) function in Visual C++.

```
template<class InputIterator, class Function> inline
   Function for_each(
      InputIterator First,
      InputIterator Last,
      Function F
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **for_each** algorithm calls **Function** *F* for each element in the range [*First*, *Last*) and returns the input parameter *F*. This function does not modify any elements in the sequence.

Example

```
// foreach.cpp
// compile with: /EHsc
//
// Functions:
//   for_each  - Calls function F for every element in a range.
//
//   begin     - Returns an iterator that points to the first element
//               in a sequence.
//
//   end       - Returns an iterator that points one past the end of
//               a sequence.

// disable warning C4786: symbol greater than 255 characters,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// prints the cube of integer n
void PrintCube(int n)
{
    cout << n * n * n << " " ;
}

int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of integers
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of integer
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;   //vector containing numbers

    IntVectorIt start, end, it ;
```

```
    int i ;

    // Initialize vector Numbers
    for (i = 0; i < VECTOR_SIZE; i++)
        Numbers[i] = i + 1 ;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // for each element in the range [first, last)
    // print the cube of the element
    for_each(start, end, PrintCube) ;
    cout << "\n\n" ;
}
```

Output

```
Numbers { 1 2 3 4 5 6 7 8  }

1 8 27 64 125 216 343 512
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

generate (STL Samples)

Illustrates how to use the generate Standard Template Library (STL) function in Visual C++.

```
template<class InputIterator, class Function> inline
   Function for_each(
      InputIterator First,
      InputIterator Last,
      Function F
   )
template<class ForwardIterator, class Generator> inline
   void generate(
      ForwardIterator First,
      ForwardIterator Last,
      Generator Gen
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **generate** algorithm traverses the range [*First*, *Last* + 1), assigning to each element the value returned by *Gen*. **generate** modifies the elements in the specified range.

Example

```
// generate.cpp
// compile with: /EHsc
//
// Functions:
//   generate - Fill a sequence using a generator function
//   Fibonacci - return the next Fibonacci number in the
//               Fibonacci series.

// disable warning C4786: symbol greater than 255 characters,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// return the next Fibonacci number in the
// Fibonacci series.
int Fibonacci(void)
{
    static int r;
    static int f1 = 0;
    static int f2 = 1;
    r = f1 + f2 ;
    f1 = f2 ;
    f2 = r ;
    return f1 ;
}

int main()
{
    const int VECTOR_SIZE = 8 ;
```

```
    // Define a template class vector of integers
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of integer
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;   //vector containing numbers

    IntVectorIt start, end, it ;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers


    // fill the range [first, last +1) with a series of
    // Fibonnaci numbers using the Fibonacci function
    generate(start, end, Fibonacci) ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;
}
```

Output

```
Numbers { 1 1 2 3 5 8 13 21  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

generate_n (STL Samples)

Illustrates how to use the generate_n Standard Template Library (STL) function in Visual C++.

```
template<class OutputIterator, class Size, class Generator> inline
   void generate_n(
      OutputIterator First,
      Size n,
      Generator Gen
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **generate_n** algorithm traverses the range [*First*, *First* + *n*), assigning to each element the value returned by *Gen*. Note that **generate** modifies the elements in the specified range.

Example

```
// generate_n.cpp
// compile with: /EHsc
//
// Functions:
//   generate_n  - Fill a specified number of elements of a sequence
//                 using a generator function.
//
//   begin       - Returns an iterator that points to the first element
//                 in a sequence.
//
//   end         - Returns an iterator that points one past the end of
//                 a sequence.
//
//   size        - Returns the length of a sequence.

// disable warning C4786: symbol greater than 255 characters,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// return the next Fibonacci number in the
// Fibonacci series.
int Fibonacci(void)
{
    static int r;
    static int f1 = 0;
    static int f2 = 1;
    r = f1 + f2 ;
    f1 = f2 ;
    f2 = r ;
    return f1 ;
}

int main()
{
    const int VECTOR_SIZE = 15 ;
```

```
    // Define a template class vector of integers
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of integer
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;   //vector containing numbers

    IntVectorIt start, end, it ;

    int i ;

    //Initialize vector Numbers
    for(i = 0; i < VECTOR_SIZE; i++)
        Numbers[i] = i * i ;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    cout << "Before calling generate_n" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // fill the specified range with a series of
    // Fibonacci numbers using the Fibonacci function
    generate_n(start + 5, Numbers.size() - 5, Fibonacci) ;

    cout << "After calling generate_n" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;
}
```

Output

```
Before calling generate_n
Numbers { 0 1 4 9 16 25 36 49 64 81 100 121 144 169 196  }

After calling generate_n
Numbers { 0 1 4 9 16 1 1 2 3 5 8 13 21 34 55  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

heap

Illustrates how to use the **heap** Standard Template Library (STL) functions in Visual C++.

```
template<class RandomAccessIterator> inline
   void make_heap(
      RandomAccessIterator First,
      RandomAccessIterator Last
   )
template<class RandomAccessIterator> inline
   void sort_heap(
      RandomAccessIterator First,
      RandomAccessIterator Last
   )
template<class RandomAccessIterator> inline
   void push_heap(
      RandomAccessIterator First,
      RandomAccessIterator Last
   )
template<class RandomAccessIterator> inline
   void pop_heap(
      RandomAccessIterator First,
      RandomAccessIterator Last
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

A heap is a sequence of elements organized like a binary tree. Each heap element corresponds to a tree node. The first value in the sequence [*First*..*Last*) is the root and is the largest value in the heap. Every element in the heap satisfies the following: Every element is less than or equal to its parent. The largest element is stored in the root, and all children hold progressively smaller values. The make_heap function converts the range [*First*..*Last*) into a heap. The sort_heap function sorts a sequence that was created using the **make_heap** function. The push_heap function inserts a new value into the heap. The pop_heap function swaps the first and last elements in the heap specified by [*First*, *Last*), and then reduces the length of the sequence by one before restoring the heap property. The nonpredicate versions of the heap functions use **operator<** for comparisons.

Example

```
// heapfunc.cpp
// compile with: /EHsc
//
// Functions:
//    make_heap : convert a sequence to a heap
//    sort_heap : sort a heap
//    push_heap : insert an element in a heap
//    pop_heap  : remove the top element from a heap

// disable warning C4786: symbol greater than 255 characters,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std;

int main()
```

```
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of int
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of strings
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;

    IntVectorIt it ;

    // Initialize vector Numbers
    Numbers[0] = 4 ;
    Numbers[1] = 10;
    Numbers[2] = 70 ;
    Numbers[3] = 10 ;
    Numbers[4] = 30 ;
    Numbers[5] = 69 ;
    Numbers[6] = 96 ;
    Numbers[7] = 100;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = Numbers.begin(); it != Numbers.end(); it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // convert Numbers into a heap
    make_heap(Numbers.begin(), Numbers.end()) ;

    cout << "After calling make_heap\n" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = Numbers.begin(); it != Numbers.end(); it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // sort the heapified sequence Numbers
    sort_heap(Numbers.begin(), Numbers.end()) ;

    cout << "After calling sort_heap\n" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = Numbers.begin(); it != Numbers.end(); it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    //insert an element in the heap
    Numbers.push_back(7) ;
    push_heap(Numbers.begin(), Numbers.end()) ;

    // you need to call make_heap to re-assert the
    // heap property
    make_heap(Numbers.begin(), Numbers.end()) ;

    cout << "After calling push_heap and make_heap\n" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = Numbers.begin(); it != Numbers.end(); it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // remove the root element from the heap Numbers
```

```
    pop_heap(Numbers.begin(), Numbers.end()) ;

    cout << "After calling pop_heap\n" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = Numbers.begin(); it != Numbers.end(); it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;
}
```

Output

```
Numbers { 4 10 70 10 30 69 96 100  }

After calling make_heap

Numbers { 100 30 96 10 4 69 70 10  }

After calling sort_heap

Numbers { 4 10 10 30 69 70 96 100  }

After calling push_heap and make_heap

Numbers { 100 69 96 30 4 70 10 10 7  }

After calling pop_heap

Numbers { 96 69 70 30 4 7 10 10 100  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

includes (STL Samples)

Illustrates how to use the includes Standard Template Library (STL) function in Visual C++.

```
template<class InputIterator1, class InputIterator2>
   inline bool includes(
      InputIterator1 First1,
      InputIterator1 Last1,
      InputIterator2 First2,
      InputIterator2 Last2
   )
```

Remarks

Note
The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **includes** algorithm searches for one sequence of values in another sequence of values. **includes** returns **true** if every element in the range [*First2*..*Last2*) is in the sequence [*First1*..*Last1*). This version of includes assumes that both sequences are sorted using **operator<**.

Example

```
// includesP.cpp
// compile with: /EHsc

// disable warning C4786: symbol greater than 255 characters,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <string>
#include <vector>
#include <deque>

using namespace std;


int main()
{
    const int VECTOR_SIZE = 5 ;

    // Define a template class vector of strings
    typedef vector<string > StringVector ;

    //Define an iterator for template class vector of strings
    typedef StringVector::iterator StringVectorIt ;

    // Define a template class deque of strings
    typedef deque<string > StringDeque ;

    //Define an iterator for template class deque of strings
    typedef StringDeque::iterator StringDequeIt ;

    StringVector CartoonVector(VECTOR_SIZE) ;
    StringDeque CartoonDeque ;

    StringVectorIt start1, end1, it1 ;
    StringDequeIt start2, end2, it2 ;
```

```
    // Initialize vector Vector1
    CartoonVector[0] = "Aladdin" ;
    CartoonVector[1] = "Jasmine" ;
    CartoonVector[2] = "Mickey" ;
    CartoonVector[3] = "Minnie" ;
    CartoonVector[4] = "Goofy" ;

    start1 = CartoonVector.begin() ;  // location of first
                                      // element of CartoonVector

    end1 = CartoonVector.end() ;  // one past the location last
                                  // element of CartoonVector

    //Initialize list CartoonDeque
    CartoonDeque.push_back("Jasmine") ;
    CartoonDeque.push_back("Aladdin") ;
    CartoonDeque.push_back("Goofy") ;

    start2 = CartoonDeque.begin() ; // location of first
                                    // element of CartoonDeque

    end2 = CartoonDeque.end() ; // one past the location last
                                // element of CartoonDeque

    //sort CartoonVector and CartoonDeque alphabetically
    //includes requires the sequences
    //to be sorted.
    sort(start1, end1) ;
    sort(start2, end2) ;

    // print contents of CartoonVector and CartoonDeque
    cout << "CartoonVector { " ;
    for(it1 = start1; it1 != end1; it1++)
        cout << *it1 << ", " ;
    cout << " }\n" << endl ;
    cout << "CartoonDeque { " ;
    for(it2 = start2; it2 != end2; it2++)
        cout << *it2 << ", " ;
    cout << " }\n" << endl ;

    //Is CartoonDeque a subset of CartoonVector?
    if(includes(start1, end1, start2, end2) )
        cout << "CartoonVector includes CartoonDeque"
        << endl ;
    else
        cout << "CartoonVector does not include CartoonDeque"
        << endl ;

}
```

Output

```
CartoonVector { Aladdin, Goofy, Jasmine, Mickey, Minnie,  }

CartoonDeque { Aladdin, Goofy, Jasmine,  }

CartoonVector includes CartoonDeque
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

inner_product (STL Samples)

Illustrates how to use the inner_product Standard Template Library (STL) function in Visual C++.

```
template<class InputIterator1, class InputIterator2, class T>
   inline T inner_product(
      InputIterator First,
      InputIterator Last,
      InputIterator First2,
      T Init
   )
template<
   class InputIterator1,
   class InputIterator2,
   class T,
   class BinOp1,
   class BinOp2
>
   inline T inner_product(
      InputIterator1 First,
      InputIterator1 Last,
      InputIterator2 First2,
      T Init,
      BinOp1 Binary_Op1,
      BinOp2 Binary_Op2
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

inner_product computes its result by initializing the accumulator **acc** with *Init* and then modifying it with: **acc** = **acc** + (**i*1) * (**i*2) - or - **acc** = *Binary_Op1*(**acc**, *Binary_Op2*(**i*1, **i*2)) for every iterator *i*1 in the range [*First*, *Last*) and iterator *i*2 in the range [*First2*, *First2* + (*Last* - *First*)) in order.

Example

```
// inner_product.cpp
// compile with: /EHsc
//
// Description of
//         inner_product(first,last,first2,init)
//         inner_product(first,last,first2,init,binary_op1,binary_op2):
//
//    Computes its result by initializing the accumulator acc with init
//        acc = init
//    and then modifying it with
//        acc = acc  +  (*i1) * (*i2)
//    or
//        acc = binary_op1(acc, binary_op2(*i1, *i2))
//    for every iterator i1 in the range [first, last) and
//    iterator  i2  in  the  range [first2, first2 + (last - first))
//    in order.
//////////////////////////////////////////////////////////////////////

#include <iostream>
#include <numeric>
#include <functional>
#include <vector>
#include <iterator>
```

```
using namespace std;


typedef vector < int > intArray;
typedef ostream_iterator < int, char, char_traits<char> >
FloatOstreamIt;

int main ()
{
    FloatOstreamIt itOstream(cout," ");

    // Initialize the arrays
    intArray rgF1, rgF2;
    for (int i=1; i<=5; i++) {
        rgF1.push_back(i);
        rgF2.push_back(i*i);
    };

    // Print the arrays

    cout << "Array 1: ";
    copy(rgF1.begin(),rgF1.end(),itOstream);
    cout << endl;
    cout << "Array 2: ";
    copy(rgF2.begin(),rgF2.end(),itOstream);
    cout << endl;

    // Compute the inner_product of the arrays.  This is the
    // sum of the products (S.O.P) of the corresponding elements

    int ip1 = inner_product(rgF1.begin(),rgF1.end(),rgF2.begin(),0);

    cout << "The inner product (S.O.P) of Array1 and Array2 is "
       << ip1
       << endl;

    // Compute the inner_product of the arrays.  This is the
    // product of the sums (P.O.S.) of the corresponding elements
    int ip2 = inner_product(rgF1.begin(),rgF1.end(),rgF2.begin(),1,
                           multiplies<int>(),plus<int>());
    cout << "The inner product (P.O.S.) of Array1 and Array2 is "
       << ip2
       << endl;

}
```

Output

```
Array 1: 1 2 3 4 5
Array 2: 1 4 9 16 25
The inner product (S.O.P) of Array1 and Array2 is 225
The inner product (P.O.S.) of Array1 and Array2 is 86400
```

Requirements

Header: <numeric>

See Also

Concepts

Standard Template Library Samples

inplace_merge (STL Samples)

Illustrates how to use the inplace_merge Standard Template Library (STL) function in Visual C++.

```
template<class BidirectionalIterator>
   inline void inplace_merge(
      BidirectionalIterator First,
      BidirectionalIterator Middle,
      BidirectionalIterator Last
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **inplace_merge** algorithm merges two sorted subsequences: [*First*..*Middle*) and [*Middle*..*Last*) in place into a single sorted sequence [*First*..*Last*). This version assumes that the ranges [*First*..*Middle*) and [*Middle*..*Last*) are sorted using **operator<**. If both ranges contain equal values, the value from the first range will be stored first.

Example

```
// inplace_merge.cpp
// compile with: /EHsc
//
// Functions:
//   inplace_merge : Merge two sorted sub-sequences in place into a
//                   single sorted list.
//
//   begin - Returns an iterator that points to the first element in a
//           sequence.
//
//   end - Returns an iterator that points one past the end of a sequence.
///////////////////////////////////////////////////////////////////////

// disable warning C4786: symbol greater than 255 characters
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std;


int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of int
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of strings
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;

    IntVectorIt start, end, it ;

    // Initialize vector Numbers
```

```
    Numbers[0] = 4 ;
    Numbers[1] = 10;
    Numbers[2] = 70 ;
    Numbers[3] = 10 ;
    Numbers[4] = 30 ;
    Numbers[5] = 69 ;
    Numbers[6] = 96 ;
    Numbers[7] = 100;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    cout << "Before calling inplace_merge\n" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    //merge the elements of Numbers in place
    inplace_merge(start, start + 3, end) ;

    cout << "After calling inplace_merge\n" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;
}
```

Output

```
Before calling inplace_merge

Numbers { 4 10 70 10 30 69 96 100  }

After calling inplace_merge

Numbers { 4 10 10 30 69 70 96 100  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

iter_swap (STL Samples)

Illustrates how to use the iter_swap Standard Template Library (STL) function in Visual C++.

```
template<class ForwardIterator1, class ForwardIterator2> inline
   void iter_swap(
      ForwardIterator1 First,
      ForwardIterator2 Second
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **iter_swap** algorithm swaps two elements represented by two iterators.

Example

```
// iter_swap.cpp
// compile with: /EHsc
//
// Functions:
//   iter_swap  - Swap two elements in a sequence represented by
//                two iterators.
//
//   begin      - Returns an iterator that points to the first element
//                in a sequence.
//
//   end        - Returns an iterator that points one past the end of
//                  a sequence.

// disable warning C4786: symbol greater than 255 characters
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;


// return the next Fibonacci number in the
// Fibonacci series.
int Fibonacci(void)
{
    static int r;
    static int f1 = 0;
    static int f2 = 1;
    r = f1 + f2 ;
    f1 = f2 ;
    f2 = r ;
    return f1 ;
}

int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of integers
    typedef vector<int > IntVector ;
```

```
    //Define an iterator for template class vector of integer
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;   //vector containing numbers

    IntVectorIt start, end, it ;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    // fill the range [first, last +1) with a series of
    // Fibonacci numbers using the Fibonacci function
    generate(start, end, Fibonacci) ;

    cout << "Before calling iter_swap" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // swap the first and last elements of the
    // sequence using iter_swap
    iter_swap(start, end - 1) ;

    cout << "After calling iter_swap" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

}
```

Output

```
Before calling iter_swap
Numbers { 1 1 2 3 5 8 13 21  }

After calling iter_swap
Numbers { 21 1 2 3 5 8 13 1  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

list::assign (STL Samples)

Illustrates how to use the list::assign Standard Template Library (STL) function in Visual C++.

```
void assign(
   const_iterator First,
   const_iterator Last
);
void assign(
   size_type n,
   const T& x = T( )
);
iterator erase(
   iterator It
);
iterator erase(
   iterator First,
   iterator Last
); bool empty( ) const;
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The first member function replaces the sequence controlled by ***this** with the sequence [*First*, *Last*). The second member function replaces the sequence controlled by ***this** with a repetition of n elements of value *x*. The third member function removes the element of the controlled sequence pointed to by *It*. The fourth member function removes the elements of the controlled sequence in the range [*First*, *Last*). Both return an iterator that designates the first element remaining beyond any elements removed, or **end** if no such element exists. The last member function returns **true** for an empty controlled sequence.

Example

```
// assign.cpp
// compile with: /EHsc
//
// Shows various ways to assign and erase elements
//        from a list<T>.
//
// Functions:
//    list::assign
//    list::empty
//    list::erase

#include <list>
#include <iostream>

using namespace std ;

typedef list<int> LISTINT;

int main()
{
    LISTINT listOne;
    LISTINT listAnother;
    LISTINT::iterator i;

    // Add some data
    listOne.push_front (2);
    listOne.push_front (1);
    listOne.push_back (3);
```

```
    listAnother.push_front(4);

    listAnother.assign(listOne.begin(), listOne.end());

    // 1 2 3
    for (i = listAnother.begin(); i != listAnother.end(); ++i)
        cout << *i << " ";
    cout << endl;

    listAnother.assign(4, 1);

    // 1 1 1 1
    for (i = listAnother.begin(); i != listAnother.end(); ++i)
        cout << *i << " ";
    cout << endl;

    listAnother.erase(listAnother.begin());

    // 1 1 1
    for (i = listAnother.begin(); i != listAnother.end(); ++i)
        cout << *i << " ";
    cout << endl;

    listAnother.erase(listAnother.begin(), listAnother.end());
    if (listAnother.empty())
        cout << "All gone\n";
}
```

Output

```
1 2 3
1 1 1 1
1 1 1
All gone
```

Requirements

Header: <list>

See Also

Concepts

Standard Template Library Samples

list::back and list::front

Illustrates how to use the list::back and list::front Standard Template Library (STL) functions in Visual C++.

```
reference back( );
const_reference back( ) const;
reference front( );
const_reference front( ) const;
void pop_back( );
void pop_front( );
void push_back(
   const T& x
);
void push_front(
   const T& x
);
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **back** member function returns a reference to the last element of the controlled sequence. The **front** member function returns a reference to the first element of the controlled sequence. The **pop_back** member function removes the last element of the controlled sequence. The **pop_front** member function removes the first element of the controlled sequence. All these functions require that the controlled sequence be nonempty. The **push_back** member function inserts an element with value *x* at the end of the controlled sequence. The **push_front** member function inserts an element with value *x* at the beginning of the controlled sequence.

Example

```
// listtck.cpp
// compile with: /EHsc
//  This example shows how to use the various stack
//                like functions of list.
//
// Functions:
//    list::back
//    list::front
//    list::pop_back
//    list::pop_front
//    list::push_back
//    list::push_front

#pragma warning (disable:4786)
#include <list>
#include <string>
#include <iostream>

using namespace std ;

typedef list<string> LISTSTR;

int main()
{
    LISTSTR test;

    test.push_back("back");
    test.push_front("middle");
    test.push_front("front");
```

```
    // front
    cout << test.front() << endl;

    // back
    cout << test.back() << endl;

    test.pop_front();
    test.pop_back();

    // middle
    cout << test.front() << endl;
}
```

Output

```
front
back
middle
```

Requirements

Header: <list>

See Also

Concepts

Standard Template Library Samples

list::insert (STL Samples)

Illustrates how to use the list::insert Standard Template Library (STL) function in Visual C++.

```
iterator insert(
   iterator It,
   const T& x = T( )
);
void insert(
   iterator It,
   size_type n,
   const T& x
);
void insert(
   iterator It,
   const_iterator First,
   const_iterator Last
);
void insert(
   iterator It,
   const T *First,
   const T *Last
);
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

Each member functions inserts, before the element pointed to by it in the controlled sequence, a sequence specified by the remaining operands. The first member function inserts a single element with value *x* and returns an iterator that points to the newly inserted element. The second member function inserts a repetition of *n* elements of value *x*. The last two member functions insert the sequence [*First*, *Last*).

Example

```
// list_insert.cpp
// compile with: /EHsc
// Shows the various ways to insert elements into a
//               list<T>.

#include <list>
#include <iostream>

using namespace std ;

typedef list<int> LISTINT;

int main()
{
    int rgTest1[] = {5,6,7};
    int rgTest2[] = {10,11,12};

    LISTINT listInt;
    LISTINT listAnother;
    LISTINT::iterator i;

    // Insert one at a time
    listInt.insert (listInt.begin(), 2);
    listInt.insert (listInt.begin(), 1);
    listInt.insert (listInt.end(), 3);
```

```
    // 1 2 3
    cout << "lintInt:";
    for (i = listInt.begin(); i != listInt.end(); i++)
        cout << " " << *i;
    cout << endl;

    // Insert 3 fours
    listInt.insert (listInt.end(), 3, 4);

    // 1 2 3 4 4 4
    cout << "lintInt:";
    for (i = listInt.begin(); i != listInt.end(); ++i)
        cout << " " << *i;
    cout << endl;

    // Insert an array in there
    listInt.insert (listInt.end(), rgTest1, rgTest1 + 3);

    // 1 2 3 4 4 4 5 6 7
    cout << "lintInt:";
    for (i = listInt.begin(); i != listInt.end(); ++i)
        cout << " " << *i;
    cout << endl;

    // Insert another LISTINT
    listAnother.insert (listAnother.begin(), rgTest2, rgTest2+3);
    listInt.insert (listInt.end(), listAnother.begin(), listAnother.end());

    // 1 2 3 4 4 4 5 6 7 10 11 12
    cout << "lintInt:";
    for (i = listInt.begin(); i != listInt.end(); ++i)
        cout << " " << *i;
    cout << endl;
}
```

Output

```
lintInt: 1 2 3
lintInt: 1 2 3 4 4 4
lintInt: 1 2 3 4 4 4 5 6 7
lintInt: 1 2 3 4 4 4 5 6 7 10 11 12
```

Requirements

Header: <list>

See Also

Concepts

Standard Template Library Samples

list::list (STL Samples)

Illustrates how to use the list::list Standard Template Library (STL) function in Visual C++.

```
explicit list(
   const A& Al = A( )
);
explicit list(
   size_type n,
   const T& v = T( ),
   const A& Al = A( )
);
list(
   const list& x
);
list(
   const_iterator First,
   const_iterator Last,
   const A& Al = A( )
);
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The first constructor specifies an empty initial controlled sequence. The second constructor specifies a repetition of *n* elements of value *x*. The third constructor specifies a copy of the sequence controlled by *x*. The last constructor specifies the sequence [*First*, *Last*). All constructors store the allocator object *Al*, or for the copy constructor, the return value of *x*.**get_allocator**, in the data member **allocator** and initialize the controlled sequence.

Example

```
// list_list.cpp
// compile with: /EHsc
// Demonstrates the different constructors for list<T>

#pragma warning (disable:4786)
#include <list>
#include <string>
#include <iostream>

using namespace std ;

typedef list<string> LISTSTR;

// Try each of the four constructors
int main()
{
    LISTSTR::iterator i;
    LISTSTR test;                   // default constructor

    test.insert(test.end(), "one");
    test.insert(test.end(), "two");

    LISTSTR test2(test);            // construct from another list
    LISTSTR test3(3, "three");      // add several <T>'s
    LISTSTR test4(++test3.begin(),  // add part of another list
             test3.end());

    // Print them all out
```

```
    // one two
    cout << "test:";
    for (i =  test.begin(); i != test.end(); ++i)
        cout << " " << *i;
    cout << endl;

    // one two
    cout << "test:";
    for (i =  test2.begin(); i != test2.end(); ++i)
        cout << " " << *i;
    cout << endl;

    // three three three
    cout << "test:";
    for (i =  test3.begin(); i != test3.end(); ++i)
        cout << " " << *i;
    cout << endl;

    // three three
    cout << "test:";
    for (i =  test4.begin(); i != test4.end(); ++i)
        cout << " " << *i;
    cout << endl;
}
```

Output

```
test: one two
test: one two
test: three three three
test: three three
```

Requirements

Header: <list>

See Also

Concepts

Standard Template Library Samples

lower_bound (STL Samples)

Illustrates how to use the lower_bound Standard Template Library (STL) function in Visual C++.

```
template<class ForwardIterator, class T>
   inline ForwardIterator lower_bound(
      ForwardIterator First,
      ForwardIterator Last,
      const T& Value
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **lower_bound** algorithm returns the first location in the sequence that value can be inserted such that the order of the sequence is maintained. **lower_bound** returns an iterator positioned at the location that value can be inserted in the range [*First*..*Last*), or returns *Last* if no such position exists. **lower_bound** assumes the range [*First*..*Last*) is sorted using **operator<**.

Example

```
// lower_bound.cpp
// compile with: /EHsc
// Illustrates how to use the lower_bound function.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore this warning
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std;


int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of int
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of strings
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;

    IntVectorIt start, end, it, location ;

    // Initialize vector Numbers
    Numbers[0] = 4 ;
    Numbers[1] = 10;
    Numbers[2] = 10 ;
    Numbers[3] = 30 ;
    Numbers[4] = 69 ;
    Numbers[5] = 70 ;
    Numbers[6] = 96 ;
    Numbers[7] = 100;
```

```
    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // return the first location at which 10 can be inserted
    // in Numbers
    location = lower_bound(start, end, 10) ;

    cout << "First location element 10 can be inserted in Numbers is: "
        << location - start << endl ;
}
```

Output

```
Numbers { 4 10 10 30 69 70 96 100  }

First location element 10 can be inserted in Numbers is: 1
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

make_pair (STL Samples)

Illustrates how to use the make_pair Standard Template Library (STL) function in Visual C++.

```
template<class first, class second> inline
   pair<first,
      second> make_pair(
      const first& _X,
      const second& _Y
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **make_pair** STL function creates a pair structure that contains two data elements of any type.

Example

```
// mkpair.cpp
// compile with: /EHsc
// Illustrates how to use the make_pair function.
//
// Functions: make_pair - creates an object pair containing two data
//                        elements of any type.

#include <utility>
#include <iostream>

using namespace std;

/* STL pair data type containing int and float
*/

typedef struct pair<int,float> PAIR_IF;

int main(void)
{
  PAIR_IF pair1=make_pair(18,3.14f);

  cout << pair1.first << "  " << pair1.second << endl;
  pair1.first=10;
  pair1.second=1.0f;
  cout << pair1.first << "  " << pair1.second << endl;
}
```

Output

```
18  3.14
10  1
```

Requirements

Header: <utility>

See Also

Concepts

Standard Template Library Samples

map::insert, map::find, and map::end

Illustrates how to use the map::insert, map::find, and map::end Standard Template Library (STL) functions in Visual C++.

```
iterator map::end( );
iterator map::find(
   const Key& Key
);
pair<iterator, bool>
map::insert(
   const value_type& x
);
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **end** function returns an iterator that points one past the end of a sequence. **find** returns an iterator that designates the first element whose sort key equals *Key*. If no such element exists, the iterator equals **end**. If the key does not already exist, **insert** will add it to the sequence and return **pair**<**iterator**, **true**>. If the key already exists, **insert** does not add it to the sequence and returns **pair** <**iterator**, **false**>. The following sample creates a map of **int**s to **string**s. In this case, the mapping is from digits to their string equivalents (1 -> "One", 2 -> "Two", and so on). The program reads a number from the user, finds the word equivalent for each digit (using the map), and prints the number back as a series of words. For example, if the user enters 25463, the program responds with: Two Five Four Six Three.

Example

```
// map_insert_find_end.cpp
// compile with: /EHsc
#pragma warning(disable:4786)
#include <iostream>
#include <string>
#include <map>

using namespace std;

typedef map<int, string, less<int> > INT2STRING;

int main()
{
   // 1. Create a map of ints to strings
   INT2STRING theMap;
   INT2STRING::iterator theIterator;
   string theString = "";
   unsigned int index;

   // Fill it with the digits 0 - 9, each mapped to its string counterpart
   // Note: value_type is a pair for maps...
   theMap.insert(INT2STRING::value_type(0,"Zero"));
   theMap.insert(INT2STRING::value_type(1,"One"));
   theMap.insert(INT2STRING::value_type(2,"Two"));
   theMap.insert(INT2STRING::value_type(3,"Three"));
   theMap.insert(INT2STRING::value_type(4,"Four"));
   theMap.insert(INT2STRING::value_type(5,"Five"));
   theMap.insert(INT2STRING::value_type(6,"Six"));
   theMap.insert(INT2STRING::value_type(7,"Seven"));
   theMap.insert(INT2STRING::value_type(8,"Eight"));
   theMap.insert(INT2STRING::value_type(9,"Nine"));

   // Read a Number from the user and print it back as words
```

```
    for( ; ; )
    {
       cout << "Enter \"q\" to quit, or enter a Number: ";
       cin >> theString;
       if (theString == "q")
          break;

       // extract each digit from the string, find its corresponding
       // entry in the map (the word equivalent) and print it
       for (index = 0; index < theString.length(); index++)
       {
          theIterator = theMap.find(theString[index] - '0');
          if (theIterator != theMap.end() )   // is 0 - 9
             cout << (*theIterator).second << " ";
          else    // some character other than 0 - 9
             cout << "[err] ";
       }
       cout << endl;
    }
 }
```

Input

```
12
q
```

Sample Output

```
Enter "q" to quit, or enter a Number: 12
One Two
Enter "q" to quit, or enter a Number: q
```

Requirements

Header: <map>

See Also

Concepts

Standard Template Library Samples

map::max_size, map::clear, map::erase, and map::size

Illustrates how to use the map::max_size, map::clear, map::erase, and map::size Standard Template Library (STL) functions in Visual C++.

```
size_type max_size( ) const;
void clear( ) const;
bool empty( ) const;
iterator erase(
   iterator First,
   iterator Last
);
size_type size( ) const;
A::reference operator[](         // A is the allocator
   const Key& Key
);
iterator map::begin( );
iterator map::end( );
iterator map::find(
   const Key& Key
);
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The following sample creates a map of **string**s to **int**s and fills it first with a map of month names to month numbers, and then empties and refills it with a map of weekday names to corresponding **int**s.

Example

```
// map_max_size_etc_sample.cpp
// compile with: /EHsc
//
// Functions:    iterator map::max_size();
//               void clear() const;
//               bool empty() const;
//               iterator erase(iterator first, iterator last);
//               size_type size() const;
//               A::reference operator[](const Key& key);
//               iterator map::begin();
//               iterator map::end();
//               iterator map::find(const Key& key);

#pragma warning(disable:4786)

#include <iostream>
#include <string>
#include <map>

using namespace std ;

typedef map<string, int> STRING2INT;

int main()
{
   STRING2INT MyMap;
   STRING2INT::iterator MyIterator;
   string MyBuffer;

   // print the maximum number of <key,data> pairs that MyMap can hold
```

```
cout << "MyMap is capable of holding " << MyMap.max_size()
     << " <string,int> pairs" << endl;

if (!MyMap.empty())
   cout << "MyMap has " << MyMap.size() << " entries" << endl;
else
   cout << "MyMap is empty" << endl;

cout << "Entering new entries in MyMap" << endl;
// Fill in MyMap with the months of the year, mapped to their number
// January - 1, February - 2, etc. using operator[].
MyMap["January"] = 1;
MyMap["February"] = 2;
MyMap["March"] = 3;
MyMap["April"] = 4;
MyMap["May"] = 5;
MyMap["June"] = 6;
MyMap["July"] = 7;
MyMap["August"] = 8;
MyMap["September"] = 9;
MyMap["October"] = 10;
MyMap["November"] = 11;
MyMap["December"] = 12;

if (!MyMap.empty())
   cout << "MyMap has " << MyMap.size() << " entries" << endl;
else
   cout << "MyMap is empty" << endl;

// Ask the user for a month of the year and print the number
// that corresponds to the month entered
MyIterator = MyMap.end();
while (MyIterator == MyMap.end())
{
   cout << endl << "Enter a Month: ";
   cin >> MyBuffer;
   if ((MyIterator = MyMap.find(MyBuffer)) != MyMap.end())
      cout << (*MyIterator).first << " is Month Number "
           << (*MyIterator).second << endl;
   else
      cout << "Enter a Valid Month (example: March)" << endl;
}

// empty MyMap - note that clear simply calls erase(begin(),end());
MyMap.clear();

if (!MyMap.empty())
   cout << "MyMap has " << MyMap.size() << " entries" << endl;
else
   cout << "MyMap is empty" << endl;

cout << "Entering new entries in MyMap" << endl;
// Fill MyMap with the days of the week, each mapped to an int
MyMap["Monday"] = 1;
MyMap["Tuesday"] = 2;
MyMap["Wednesday"] = 3;
MyMap["Thursday"] = 4;
MyMap["Friday"] = 5;
MyMap["Saturday"] = 6;
MyMap["Sunday"] = 7;

if (!MyMap.empty())
   cout << "MyMap has " << MyMap.size() << " entries" << endl;
else
   cout << "MyMap is empty" << endl;

// Ask the user for a day of the week and print the number
// that corresponds to the day entered
```

```
    MyIterator = MyMap.end();
    while (MyIterator == MyMap.end())
    {
       cout << endl << "Enter a Day of the Week: ";
       cin >> MyBuffer;
       if ((MyIterator = MyMap.find(MyBuffer)) != MyMap.end())
          cout << (*MyIterator).first << " is Day Number "
               << (*MyIterator).second << endl;
       else
          cout << "Enter a Valid Day of the Week(example: Monday)" << endl;
    }

    // Now clear MyMap again - this time using erase instead of clear
    MyMap.erase(MyMap.begin(), MyMap.end());

    if (!MyMap.empty())
       cout << "MyMap has " << MyMap.size() << " entries" << endl;
    else
       cout << "MyMap is empty" << endl;
 }
```

Input

```
May
Monday
```

Sample Output

```
MyMap is capable of holding 134217727 <string,int> pairs
MyMap is empty
Entering new entries in MyMap
MyMap has 12 entries

Enter a Month: May
May is Month Number 5
MyMap is empty
Entering new entries in MyMap
MyMap has 7 entries

Enter a Day of the Week: Monday
Monday is Day Number 1
MyMap is empty
```

Requirements

Header: <map>

See Also

Concepts

Standard Template Library Samples

Members of the numeric_limits Class

Illustrates how to use the members of the numeric_limits class in Visual C++.

To reference one of the members of this class, you need to specify the type that you want to obtain information about and the member name. The return will either be a value of 1 for **true** or 0 for **false**. For example, **numeric_limits**<**int**>::**min** will return the minimum value for an **int**.

Note
Some of the members are valid only for certain types. Please refer to the Online Help to determine if a member is valid only f or a certain type.

Example

```
// Numeric.cpp
// compile with: /EHsc
//
// Variables and Functions:
//
//    has_denorm
//    has_denorm_loss
//    has_infinity
//    has_quiet_NaN
//    has_signaling_NaN
//    is_bounded
//    is_exact
//    is_iec559
//    is_integer
//    is_modulo
//    is_signed
//    is_specialized
//    tinyness_before
//    traps
//    round_style
//    digits
//    digits10
//    max_exponent
//    max_exponent10
//    min_exponent
//    min_exponent10
//    radix;
//    denorm_min()
//    epsilon()
//    infinity()
//    max()
//    min()
//    quiet_ NaN()
//    round_error()
//    signaling_NaN()
/////////////////////////////////////////////////////////////////////

#include <iostream>
#include <limits>

using namespace std;

int main() {
    cout << " 1 The minimum value for char is " <<
        (int)numeric_limits<char>::min() << endl;
    cout << " 2 The minimum value for int is  " <<
        numeric_limits<int>::min() << endl;
    cout << " 3 The maximum value for char is " <<
        (int)numeric_limits<char>::max() << endl;
    cout << " 4 The maximum value for int is  " <<
        numeric_limits<int>::max() << endl;
```

```
cout << " 5 The number of bits to represent a char is " <<
    numeric_limits<char>::digits << endl;
cout << " 6 The number of bits to represent an int is " <<
    numeric_limits<int>::digits << endl;
cout <<" 7 The number of digits representable in base 10 for float is "
     << numeric_limits<float>::digits10 << endl;
cout << " 8 Is a char signed?              " <<
    numeric_limits<char>::is_signed << endl;
cout << " 9 Is an unsigned integer signed? " <<
    numeric_limits<unsigned int>::is_signed << endl;
cout << "10 Is an integer an integer? " <<
    numeric_limits<int>::is_integer << endl;
cout << "11 Is a float an integer?   " <<
    numeric_limits<float>::is_integer << endl;
cout << "12 Is an integer exact? " <<
    numeric_limits<int>::is_exact << endl;
cout << "13 Is a float exact?  " <<
    numeric_limits<float>::is_exact << endl;
cout << "14 The radix for float is            "  <<
    numeric_limits<float>::radix << endl;
cout << "15 The epsilon for float is          " <<
    numeric_limits<float>::epsilon() << endl;
cout << "16 The round error for float is      " <<
    numeric_limits<float>::round_error() << endl;
cout << "17 The minimum exponent for float is " <<
    numeric_limits<float>::min_exponent << endl;
cout << "18 The minimum exponent in base 10   " <<
    numeric_limits<float>::min_exponent10 << endl;
cout << "19 The maximum exponent is           " <<
    numeric_limits<float>::max_exponent << endl;
cout << "20 The maximum exponent in base 10   " <<
    numeric_limits<float>::max_exponent10 << endl;
cout << "21 Can float represent positive infinity?  " <<
    numeric_limits<float>::has_infinity << endl;
cout << "22 Can double represent positive infinity? " <<
    numeric_limits<double>::has_infinity << endl;
cout << "23 Can int represent positive infinity? " <<
    numeric_limits<int>::has_infinity << endl;
cout << "24 Can float represent a NaN?           " <<
    numeric_limits<float>::has_quiet_NaN << endl;
cout << "25 Can float represent a signaling NaN? " <<
    numeric_limits<float>::has_signaling_NaN << endl;
cout << "26 Does float allow denormalized values?   " <<
    numeric_limits<float>::has_denorm << endl;
cout << "27 Does float detect denormalization loss? " <<
    numeric_limits<float>::has_denorm_loss << endl;
cout << "28 Representation of positive infinity for float " <<
    numeric_limits<float>::infinity() << endl;
cout << "29 Representation of quiet NaN for float         " <<
    numeric_limits<float>::quiet_NaN() << endl;
cout << "30 Minimum denormalized number for float         " <<
    numeric_limits<float>::denorm_min() << endl;
cout << "31 Minimum positive denormalized value for float " <<
    numeric_limits<float>::denorm_min() << endl;
cout << "32 Does float adhere to IEC 559 standard?  " <<
    numeric_limits<float>::is_iec559 << endl;
cout << "33 Is float bounded? " <<
    numeric_limits<float>::is_bounded << endl;
cout << "34 Is float modulo?  " <<
    numeric_limits<float>::is_modulo << endl;
cout << "35 Is int modulo?    " <<
    numeric_limits<float>::is_modulo << endl;
cout << "36 Is trapping implemented for float?    " <<
    numeric_limits<float>::traps << endl;
cout << "37 Is tinyness detected before rounding? " <<
    numeric_limits<float>::tinyness_before << endl;
cout << "38 What is the rounding style for float? " <<
    (int)numeric_limits<float>::round_style << endl;
```

```
    cout << "39 What is the rounding style for int? " <<
        (int)numeric_limits<int>::round_style << endl;
    cout << "40 How does a float represent a signaling NaN? " <<
        numeric_limits<float>::signaling_NaN() << endl;
    cout << "41 Is int specialized? " <<
        numeric_limits<float>::is_specialized << endl;
}
```

Output

```
 1 The minimum value for char is -128
 2 The minimum value for int is  -2147483648
 3 The maximum value for char is 127
 4 The maximum value for int is  2147483647
 5 The number of bits to represent a char is 7
 6 The number of bits to represent an int is 31
 7 The number of digits representable in base 10 for float is 6
 8 Is a char signed?              1
 9 Is an unsigned integer signed? 0
10 Is an integer an integer? 1
11 Is a float an integer?   0
12 Is an integer exact? 1
13 Is a float exact?  0
14 The radix for float is            2
15 The epsilon for float is          1.19209e-007
16 The round error for float is      0.5
17 The minimum exponent for float is -125
18 The minimum exponent in base 10   -37
19 The maximum exponent is           128
20 The maximum exponent in base 10   38
21 Can float represent positive infinity?  1
22 Can double represent positive infinity? 1
23 Can int represent positive infinity? 0
24 Can float represent a NaN?           1
25 Can float represent a signaling NaN? 1
26 Does float allow denormalized values?   1
27 Does float detect denormalization loss? 1
28 Representation of positive infinity for float 1.#INF
29 Representation of quiet NaN for float         1.#QNAN
30 Minimum denormalized number for float         1.4013e-045
31 Minimum positive denormalized value for float 1.4013e-045
32 Does float adhere to IEC 559 standard?  1
33 Is float bounded? 1
34 Is float modulo?  0
35 Is int modulo?    0
36 Is trapping implemented for float?    1
37 Is tinyness detected before rounding? 1
38 What is the rounding style for float? 1
39 What is the rounding style for int? 0
40 How does a float represent a signaling NaN? 1.#QNAN
41 Is int specialized? 1
```

Requirements

Header: <limits>

See Also

Concepts

Standard Template Library Samples

merge (STL Samples)

Illustrates how to use the merge Standard Template Library (STL) function in Visual C++.

```
template<
   class InputIterator1,
   class InputIterator2,
   class OutputIterator
> inline
   OutputIterator merge(
      InputIterator1 First1,
      InputIterator1 Last1,
      InputIterator2 First2,
      InputIterator2 Last2,
      OutputIterator Result
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **merge** algorithm merges two sorted sequences: [*First1*..*Last1*) and [*First2*..*Last2*) into a single sorted sequence starting at *Result*. This version assumes that the ranges [*First1*..*Last1*) and [*First2*..*Last2*) are sorted using **operator<**. If both ranges contain equal values, the value from the first range will be stored first. The result of merging overlapping ranges is undefined.

Example

```
// merge.cpp
// compile with: /EHsc
//
// Functions:
//    merge : Merge two sorted sequences
//            into a single sorted list.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <vector>
#include <list>
#include <deque>

using namespace std ;

int main()
{
    const int MAX_ELEMENTS = 8 ;

    // Define a template class vector of int
    typedef vector<int> IntVector ;

    //Define an iterator for template class vector of ints
    typedef IntVector::iterator IntVectorIt ;

    IntVector NumbersVector(MAX_ELEMENTS) ;

    IntVectorIt startv, endv, itv ;

    // Define a template class list of int
```

```
typedef list<int> IntList ;

//Define an iterator for template class list of ints
typedef IntList::iterator IntListIt ;

IntList NumbersList ;

IntListIt first, last, itl ;

// Define a template class deque of int
typedef deque<int> IntDeque ;

//Define an iterator for template class deque of ints
typedef IntDeque::iterator IntDequeIt ;

IntDeque NumbersDeque(2 * MAX_ELEMENTS) ;

IntDequeIt itd ;

// Initialize vector NumbersVector
NumbersVector[0] = 4 ;
NumbersVector[1] = 10;
NumbersVector[2] = 70 ;
NumbersVector[3] = 10 ;
NumbersVector[4] = 30 ;
NumbersVector[5] = 69 ;
NumbersVector[6] = 96 ;
NumbersVector[7] = 100;

startv = NumbersVector.begin() ;   // location of first
                                   // element of NumbersVector

endv = NumbersVector.end() ;  // one past the location
                              // last element of NumbersVector

// sort NumbersVector, merge requires the sequences
// to be sorted
sort(startv, endv) ;

// print content of NumbersVector
cout << "NumbersVector { " ;
for(itv = startv; itv != endv; itv++)
    cout << *itv << " " ;
cout << " }\n" << endl ;

// Initialize vector NumbersList
for(int i = 0; i < MAX_ELEMENTS; i++)
    NumbersList.push_back(i) ;

first = NumbersList.begin() ;   // location of first
                                // element of NumbersList

last = NumbersList.end() ;  // one past the location
                            // last element of NumbersList

// print content of NumbersList
cout << "NumbersList { " ;
for(itl = first; itl != last; itl++)
    cout << *itl << " " ;
cout << " }\n" << endl ;

// merge the elements of NumbersVector
// and NumbersList and place the
// results in NumbersDeque
merge(startv, endv, first, last, NumbersDeque.begin()) ;

cout << "After calling merge\n" << endl ;
```

```
    // print content of NumbersDeque
    cout << "NumbersDeque { " ;
    for(itd = NumbersDeque.begin();
        itd != NumbersDeque.end(); itd++)
        cout << *itd << " " ;
    cout << " }\n" << endl ;
}
```

Output

```
NumbersVector { 4 10 10 30 69 70 96 100  }

NumbersList { 0 1 2 3 4 5 6 7  }

After calling merge

NumbersDeque { 0 1 2 3 4 4 5 6 7 10 10 30 69 70 96 100  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

min_element (STL Samples)

Illustrates how to use the min_element Standard Template Library (STL) function in Visual C++.

```
template<class InputIterator> inline
   InputIterator min_element(
      InputIterator First,
      InputIterator Last
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **min_element** algorithm returns the location of the minimum element in the sequence [*First*, *Last*). The nonpredicate version of **min_element** uses **operator<** for comparisons.

Example

```
// min_element.cpp
// compile with: /EHsc
//
// Functions:
//    min_element - Return the minimum element within a range.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std;


int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of int
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of strings
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;

    IntVectorIt start, end, it, location ;

    // Initialize vector Numbers
    Numbers[0] = 4 ;
    Numbers[1] = 10 ;
    Numbers[2] = 10 ;
    Numbers[3] = 30 ;
    Numbers[4] = 69 ;
    Numbers[5] = 70 ;
    Numbers[6] = 96 ;
    Numbers[7] = 100 ;
```

```
    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // return the minimum element in the Numbers
    location = min_element(start, end) ;

    cout << "The minimum element in Numbers is: "
        << *location << endl ;
}
```

Output

```
Numbers { 4 10 10 30 69 70 96 100  }

The minimum element in Numbers is: 4
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

new operator (STL Samples)

Illustrates how to use operator new in <new>.

```
void *operator new(
   size_t n
)
void *operator new(
   size_t n,
   const nothrow&
)
void *operator new[](
   size_t n
);
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The first **new** operator will attempt to allocate memory and if it fails, will throw an exception. The **new** second operator accepts a second parameter of type nothrow. This parameter indicates that if the allocation fails, it should return NULL and not throw an exception. The **new** third operator will allocate memory for an array of that type and if it fails, will throw an exception.

Example

```
// newop.cpp
// compile with: /EHsc
//
// Functions:
//   void *operator new(size_t n)
//   void *operator new(size_t n, const nothrow&)
//   void *operator new[](size_t n);

#include <new>
#include <iostream>
using namespace std;

class BigClass {
public:
   BigClass() {};
   ~BigClass(){}

#ifdef _WIN64
      double BigArray[0x0fffffff];
#else
      double BigArray[0x0fffffff];
#endif
};

int main() {
   try {
      BigClass * p = new BigClass;
   }

   catch( bad_alloc a) {
      const char * temp = a.what();
      cout << temp << endl;
      cout << "Threw a bad_alloc exception" << endl;
   }

   BigClass * q = new(nothrow) BigClass;
```

```
    if ( q == NULL )
       cout << "Returned a NULL pointer" << endl;

    try {
       BigClass * r[3] = {new BigClass, new BigClass, new BigClass};
    }

    catch( bad_alloc a) {
       const char * temp = a.what();
       cout << temp << endl;
       cout << "Threw a bad_alloc exception" << endl;
    }
 }
```

Sample Output

```
bad allocation
Threw a bad_alloc exception
Returned a NULL pointer
bad allocation
Threw a bad_alloc exception
```

Requirements

Header: <new>

See Also

Concepts

Standard Template Library Samples

next_permutation (STL Samples)

Illustrates how to use the next_permutation Standard Template Library (STL) function in Visual C++.

```
template<class BidirectionalIterator> inline
   bool next_permutation(
      BidirectionalIterator First,
      BidirectionalIterator Last
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **next_permutation** algorithm changes the order of the elements in the range [*First*, *Last*) to the next lexicographic permutation and returns **true**. If there is no **next_permutation**, it arranges the sequence to be the first permutation and returns **false**.

Note

The **next_permutation** algorithm assumes that the sequence is sorted in ascending order using **operator<**. The nonpredicate version uses the **operator<** to order the permutations.

Example

```
// next_permutation.cpp
// compile with: /EHsc
// Illustrates how to use the next_permutation function.
//
// Functions:
//    next_permutation : Change the order of the sequence to the
//                       next lexicograhic permutation.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <functional>

using namespace std ;

int main()
{
    const int VECTOR_SIZE = 3 ;

    // Define a template class vector of strings
    typedef vector<string> StrVector ;

    //Define an iterator for template class vector of strings
    typedef StrVector::iterator StrVectorIt ;

    //Define an ostream iterator for strings
    typedef ostream_iterator<string> StrOstreamIt;

    StrVector Pattern(VECTOR_SIZE) ;
```

```
    StrVectorIt start, end, it ;

    StrOstreamIt outIt(cout, " ") ;

    start = Pattern.begin() ;   // location of first
                                // element of Pattern

    end = Pattern.end() ;       // one past the location last
                                // element of Pattern

    //Initialize vector Pattern
    Pattern[0] = "A" ;
    Pattern[1] = "B" ;
    Pattern[2] = "C" ;

    // print content of Pattern
    cout << "Before calling next_permutation:\n" << "Pattern: " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << "\n\n" ;

    // Generate all possible permutations

    cout << "After calling next_permutation:" << endl ;
    while ( next_permutation(start, end) )
    {
        copy(start, end, outIt) ;
        cout << endl ;
    }
}
```

Output

```
Before calling next_permutation:
Pattern: A B C

After calling next_permutation:
A C B
B A C
B C A
C A B
C B A
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

Nonpredicate Version of adjacent_find

Illustrates how to use the nonpredicate version of the adjacent_find Standard Template Library (STL) function in Visual C++.

```
template<class ForwardIterator> inline
   ForwardIterator adjacent_find(
      ForwardIterator First,
      ForwardIterator Last
   );
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **adjacent_find** algorithm finds consecutive pairs of matching elements in a sequence. The **adjacent_find** algorithm returns an iterator referencing the first consecutive matching element in the range (*First*, *Last*), or *Last* if there are no such elements. Comparison is done using **operator==** in this nonpredicate version of the algorithm.

Example

```
// adfind.cpp
// compile with: /EHsc
// Illustrates how to use the  non-predicate version of
//              adjacent_find function.
//
// Functions:
//   adjacent_find - Locate a matching consecutive sequence in a range.

#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
    const int ARRAY_SIZE = 8 ;
    int IntArray[ARRAY_SIZE] = { 1, 2, 3, 4, 4, 5, 6, 7 } ;

    int *location ;   // stores the position for the first pair
                      // of matching consecutive elements.

    int i ;

    // print content of IntArray
    cout << "IntArray { " ;
    for(i = 0; i < ARRAY_SIZE; i++)
        cout << IntArray[i] << ", " ;
    cout << " }" << endl ;

    // Find the first pair of matching consecutive elements
    // in the range [first, last + 1)
    // This version performs matching using operator==
    location = adjacent_find(IntArray, IntArray + ARRAY_SIZE) ;

    //print the matching consecutive elements if any were found
    if (location != IntArray + ARRAY_SIZE)  // matching consecutive
                                            // elements found
        cout << "Found adjacent pair of matching elements: ("
        << *location << ", " << *(location + 1) << "), " <<
        "at location " << location - IntArray << endl;
    else         // no matching consecutive elements were found
```

```
        cout << "No adjacent pair of matching elements were found"
        << endl ;

  }
```

Output

```
IntArray { 1, 2, 3, 4, 4, 5, 6, 7,  }
Found adjacent pair of matching elements: (4, 4), at location 3
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

Nonpredicate Version of max_element

Illustrates how to use the nonpredicate version of the max_element Standard Template Library (STL) function in Visual C++.

```
template<class InputIterator>
   inline InputIterator max_element(
      InputIterator First,
      InputIterator Last
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **max_element** algorithm returns the location of the maximum element in the sequence [*First*, *Last*). The nonpredicate version of **max_element** uses **operator<** for comparisons.

Example

```
// max_element.cpp
// compile with: /EHsc
// Illustrates how to use the max_element function.
//
// Functions:
//    max_element : Return the maximum element within a range.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std;


int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of int
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of strings
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;

    IntVectorIt start, end, it, location ;

    // Initialize vector Numbers
    Numbers[0] = 4 ;
    Numbers[1] = 10;
    Numbers[2] = 10 ;
    Numbers[3] = 30 ;
    Numbers[4] = 69 ;
    Numbers[5] = 70 ;
    Numbers[6] = 96 ;
    Numbers[7] = 100;
```

```
    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // return the maximum element in the Numbers
    location = max_element(start, end) ;

    cout << "The maximum element in Numbers is: "
        << *location << endl ;
}
```

Output

```
Numbers { 4 10 10 30 69 70 96 100  }

The maximum element in Numbers is: 100
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

nth_element (STL Samples)

Illustrates how to use the nth_element Standard Template Library (STL) function in Visual C++.

```
template<class RandomAccessIterator> inline
   void nth_element(
      RandomAccessIterator First,
      RandomAccessIterator Nth,
      RandomAccessIterator Last
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **nth_element** algorithm partitions the sequence [*First*..*Last*) on the value referenced by *Nth*. All the elements less than or equal to the value are placed before value and all elements greater than value are placed after value in the sequence. The nonpredicate version of **nth_element** uses **operator<** for comparisons.

Example

```
// nth_element.cpp
// compile with: /EHsc
// Illustrates how to use the nth_element function.
//
// Functions:
//   nth_element : Partition the elements in a sequence by its nth
//                 element.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std ;

int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of int
    typedef vector<int> IntVector ;

    //Define an iterator for template class vector of strings
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;

    IntVectorIt start, end, it ;

    // Initialize vector Numbers
    Numbers[0] = 4 ;
    Numbers[1] = 10;
    Numbers[2] = 70 ;
    Numbers[3] = 30 ;
    Numbers[4] = 10;
    Numbers[5] = 69 ;
```

```
    Numbers[6] = 96 ;
    Numbers[7] = 100;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    cout << "Before calling nth_element\n" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // partition the elements by the 4th element
    nth_element(start, start+4, end) ;

    cout << "After calling nth_element\n" << endl ;

    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;
}
```

Output

```
Before calling nth_element

Numbers { 4 10 70 30 10 69 96 100  }

After calling nth_element

Numbers { 4 10 10 30 69 70 96 100  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

Pair Logical Operator

Illustrates how to use the pair logical operator Standard Template Library (STL) function in Visual C++.

Note

Only **operator<** and **operator==** are necessary to define all of the logical operators.

```
template<class _T1, class _T2> inline
   bool operator==(
      const pair<_T1, _T2>& _X,
      const pair<_T1, _T2>& _Y
   ) {return ( _X.first == _Y.first && _X.second == _Y.second ); }
template<class _T1, class _T2> inline
   bool operator<(
      const pair<_T1, _T2>& _X,
      const pair<_T1, _T2>& _Y
   ) {return (
        _X.first < _Y.first || !( _Y.first < _X.first)
        && _X.second < _Y.second
   ); }
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The functions are described in the comments section of the sample.

Example

```
// paircomp.cpp
// compile with: /EHsc
// Illustrates several comparison operators used to compare two
// pair objects.
//
// Functions:
//    operator== - returns true if two pair objects are identical.
//
//    operator!= - returns true if two pair objects are not identical.
//
//    operator<  - returns true for (A < B) if pair object A is less
//                 than pair object B.
//
//    operator<= - returns true for (A <= B) if pair object A is less
//                 than or equal to pair object B.
//
//    operator>  - returns true for (A > B) if pair object A is greater
//                 than pair object B.
//
//    operator>= - returns true for (A >= B) if pair object A is greater
//                 than or equal to pair object B.
/////////////////////////////

#include <iostream>
#include <utility>

using namespace std ;

/* STL pair data type containing int and float */
```

```
typedef struct pair<int, float> PAIR_IF;

int main(void)
{
  PAIR_IF A(10,3.14f);
  PAIR_IF B(18,3.14f);
  PAIR_IF C(10,6.28f);
  PAIR_IF D(10,3.14f);

/* show pair values */

  cout << "A = ( " << A.first << " , " << A.second << " )" << endl;
  cout << "B = ( " << B.first << " , " << B.second << " )" << endl;
  cout << "C = ( " << C.first << " , " << C.second << " )" << endl;
  cout << "D = ( " << D.first << " , " << D.second << " )" << endl;

/* operator== */

  if (A==D)
    cout << "A and D are equal" << endl;
  else
    cout << "A and D are not equal" << endl;

/* operator!= */

  if (B!=C)
    cout << "B and C are not equivalent" << endl;
  else
    cout << "B and C are equivalent" << endl;

/* operator> */

  if (A>C)
    cout << "A is greater than C" << endl;
  else
    cout << "A is not greater than C" << endl;

/* operator>= */

  if (A>=C)
    cout << "A is greater than or equal to C" << endl;
  else
    cout << "A is not greater than or equal to C" << endl;

/* operator< */

  if (C<D)
    cout << "C is less than D" << endl;
  else
    cout << "C is not less than D" << endl;

/* operator<= */

  if (C<D)
    cout << "C is less than or equal to D" << endl;
  else
    cout << "C is not less than or equal to D" << endl;
}
```

Output

```
A = ( 10 , 3.14 )
B = ( 18 , 3.14 )
C = ( 10 , 6.28 )
D = ( 10 , 3.14 )
A and D are equal
B and C are not equivalent
A is not greater than C
```

```
A is not greater than or equal to C
C is not less than D
C is not less than or equal to D
```

Requirements

Header: <utility>

See Also

Concepts

Standard Template Library Samples

partial_sort (STL Samples)

Illustrates how to use the partial_sort Standard Template Library (STL) function in Visual C++.

```
template<class RandomAccessIterator> inline
   void partial_sort(
      RandomAccessIterator First,
      RandomAccessIterator Middle,
      RandomAccessIterator Last
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **partial_sort** algorithm sorts the smallest *n* elements, where *n* = *Middle* - *First* of the sequence [*First*, *Last*). The remaining elements end up in the range [*Middle*..*Last*) in an undefined order. The nonpredicate version of **partial_sort** uses **operator<** for comparisons.

Example

```
// partial_sort.cpp
// compile with: /EHsc
// Illustrates how to use the partial_sort function.
//
// Functions:
//    partial_sort : Sort the smallest N elements in a sequence.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std ;

int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of int
    typedef vector<int> IntVector ;

    //Define an iterator for template class vector of strings
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;

    IntVectorIt start, end, it ;

    // Initialize vector Numbers
    Numbers[0] = 4 ;
    Numbers[1] = 10;
    Numbers[2] = 70 ;
    Numbers[3] = 30 ;
    Numbers[4] = 10;
    Numbers[5] = 69 ;
    Numbers[6] = 96 ;
```

```
    Numbers[7] = 7;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    cout << "Before calling partial_sort\n" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // sort the smallest 4 elements in the sequence
    partial_sort(start, start+4, end) ;

    cout << "After calling partial_sort\n" << endl ;

    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;
}
```

Output

```
Before calling partial_sort

Numbers { 4 10 70 30 10 69 96 7  }

After calling partial_sort

Numbers { 4 7 10 10 70 69 96 30  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

partial_sort_copy (STL Samples)

Illustrates how to use the partial_sort_copy Standard Template Library (STL) function in Visual C++.

```
template<class InputIterator, RandomAccessIterator> inline
   RandomAccessIterator partial_sort(
      InputIterator First1,
      InputIterator Last1,
      RandomAccessIterator First2,
      RandomAccessIterator Last2
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **partial_sort_copy** algorithm sorts the smallest *n* elements, where *n* = **min**((*Last1* - *First1*), (*Last2* - *First2*)), of the sequence [*First1*, *Last1*) and copies the results to the sequence [*First2*, *First2* + *n*]. The nonpredicate version of **partial_sort_copy** uses **operator<** for comparisons.

Example

```
// partial_sort_copy.cpp
// compile with: /EHsc
// Illustrates how to use the partial_sort_copy function.
//
// Functions:
//    partial_sort_copy : Sort the smallest N elements in a sequence
//                        and copy the resulting sequence
//                        to another sequence.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std ;

int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of int
    typedef vector<int> IntVector ;

    //Define an iterator for template class vector of strings
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;
    IntVector Result(4) ;

    IntVectorIt start, end, it ;

    // Initialize vector Numbers
    Numbers[0] = 4 ;
    Numbers[1] = 10;
    Numbers[2] = 70 ;
```

```
    Numbers[3] = 30 ;
    Numbers[4] = 10;
    Numbers[5] = 69 ;
    Numbers[6] = 96 ;
    Numbers[7] = 7;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    cout << "Before calling partial_sort_copy\n" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // sort the smallest 4 elements in the Numbers
    // and copy the results in Result
    partial_sort_copy(start, end, Result.begin(), Result.end()) ;

    cout << "After calling partial_sort_copy\n" << endl ;

    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    cout << "Result { " ;
    for(it = Result.begin(); it != Result.end(); it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;
}
```

Output

```
Before calling partial_sort_copy

Numbers { 4 10 70 30 10 69 96 7  }

After calling partial_sort_copy

Numbers { 4 10 70 30 10 69 96 7  }

Result { 4 7 10 10  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

partial_sum (STL Samples)

Illustrates how to use the partial_sum Standard Template Library (STL) function in Visual C++.

```
template<class InputIterator, class OutputIterator> inline
   OutputIterator partial_sum(
      InputIterator First,
      InputIterator Last,
      OutputIterator First2
   )
template<
   class InputIterator,
   class OutputIterator,
   class BinaryOperator
> inline
   OutputIterator partial_sum(
      InputIterator First,
      InputIterator Last,
      OutputIterator First2,
      BinaryOperator Binary_Op
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

Assigns to every iterator *i* in the range [*Result*,*Result* + (*Last* - *First*)) a value correspondingly equal to ((...(**First* + *(*First* + 1)) + ...) + *(*First* + (*i* - *Result*))) - or - *Binary_Op*(*Binary_Op*(..., *Binary_Op*(**First*, *(*First* + 1)),...), *(*First* + (*i* - *Result*))). In other words, *(*Result* + *i*) = **init** + *(*First* + 0) + *(*First* + 1) + ... + *(*First* + i).

Example

```
// partial_sum.cpp
// compile with: /EHsc
// Demonstrates the use of partial_sum().
//
// Description of partial_sum(first,last,first2,init)
//                partial_sum(first,last,first2,init,binary_op):
//
//    Assigns to every iterator i in the range
//    [result,result + (last - first)) a value correspondingly equal to
//    ((...(*first + *(first + 1)) + ...) + *(first + (i - result)))
//
//     - or -
//
//    binary_op(binary_op(..., binary_op(*first, *(first  +  1)),...),
//    *(first + (i - result)))
//
//    In other words,
//    *(result+i) = init + *(first+0) + *(first+1) + ... + *(first+i)
//////////////////////////////////////////////////////////////////////

#include <iostream>
#include <numeric>
#include <functional>
#include <vector>
#include <iterator>

using namespace std;

typedef vector < int > IntArray;
```

```
typedef ostream_iterator < int, char, char_traits<char> > IntOstreamIt;

int main ()
{
    IntOstreamIt itOstream(cout," ");

    // Initialize the array
    IntArray rgI;
    for (int i=1; i<=10; i++)
      rgI.push_back(i);

    // Print the arrays
    cout << "Array: [ ";
    copy(rgI.begin(),rgI.end(),itOstream);
    cout << "]" << endl;


    // The result array must be at least the same size as the data array
    IntArray rgIresult(rgI.size());

    // Compute the partial sum of the array
    partial_sum(rgI.begin(),rgI.end(),rgIresult.begin());

    // Print the array of partial sums
    cout << "Array of partial sums: [ ";
    copy(rgIresult.begin(),rgIresult.end(),itOstream);
    cout << "]" << endl;

    // Compute the partial product of the array
    partial_sum(rgI.begin(),rgI.end(),rgIresult.begin(),multiplies<int>());

    // Print the array of partial products
    cout << "Array of partial products: [ ";
    partial_sum(rgIresult.begin(),rgIresult.end(),itOstream);
    cout << "]" << endl;
}
```

Output

```
Array: [ 1 2 3 4 5 6 7 8 9 10 ]
Array of partial sums: [ 1 3 6 10 15 21 28 36 45 55 ]
Array of partial products: [ 1 3 9 33 153 873 5913 46233 409113 4037913 ]
```

Requirements

Header: <numeric>

See Also

Concepts

Standard Template Library Samples

partition (STL Samples)

Illustrates how to use the partition Standard Template Library (STL) function in Visual C++.

```
template<class BidirectionalIterator, class Predicate> inline
   BidirectionalIterator partition(
      BidirectionalIterator First,
      BidirectionalIterator Last,
      Predicate Predicate
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **partition** algorithm arranges elements in the range [*First*, *Last*) such that the elements for which *Predicate* returns **true** are before the elements for which predicate returns **false**. The algorithm returns an iterator positioned at the first element for which *Predicate* returns **false**.

Example

```
// partition.cpp
// compile with: /EHsc
// Illustrates how to use the partition function.
//
// Functions:
//   partition  - Partition a range using a predicate.
//
//   begin      - Returns an iterator that points to the first element
//                in a sequence.
//
//   end        - Returns an iterator that points one past the end of
//                a sequence.
//
//   bind2nd    - Returns true for elements for which the condition
//                is true.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of integers
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of integer
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;   //vector containing numbers

    IntVectorIt start, end, it ;
```

```
    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    //Initialize vector Numbers
    Numbers[0] = 6 ;
    Numbers[1] = 20 ;
    Numbers[2] = 10 ;
    Numbers[3] = 15 ;
    Numbers[4] = 12 ;
    Numbers[5] = 7 ;
    Numbers[6] = 9 ;
    Numbers[7] = 10 ;

    cout << "Before calling partition" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // partition the sequence such that all the elements
    // less than 11 appear before all the elements greater than 11
    it =  partition(start, end, bind2nd(less<int>(), 11)) ;

    cout << "After calling partition" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

}
```

Output

```
Before calling partition
Numbers { 6 20 10 15 12 7 9 10  }

After calling partition
Numbers { 6 10 10 9 7 12 15 20  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

Predicate Version of adjacent_find

Illustrates how to use the predicate version of the adjacent_find Standard Template Library (STL) function in Visual C++.

```
template<class ForwardIterator, class BinaryPredicate> inline
   ForwardIterator adjacent_find(
      ForwardIterator First,
      ForwardIterator Last,
      BinaryPredicate Binary_Pred
   ) ;
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **adjacent_find** algorithm finds consecutive pairs of matching elements in a sequence. **adjacent_find** returns an iterator referencing the first consecutive matching element in the range [*First*, *Last*), or last if there are no such elements. Comparison is done using the **binary_pred** function in this version of the algorithm. The **binary_pred** function can be any user-defined function. You could also use one of the binary function objects provided by the STL.

Example

```
// adfind2.cpp
// compile with: /EHsc
// Illustrates how to use the predicate version of
// adjacent_find function.
//
// Functions:
//   adjacent_find  - Locate a consecutive sequence in a range.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <string>
#include <vector>

using namespace std;


int main()
{
    const int VECTOR_SIZE = 5 ;

    // Define a template class vector of strings
    typedef vector<string > StringVector ;

    //Define an iterator for template class vector of strings
    typedef StringVector::iterator StringVectorIt ;

    StringVector NamesVect(VECTOR_SIZE) ;   //vector containing names

    StringVectorIt location ;   // stores the position for the
                                 // first pair of matching
                                 // consecutive elements.

    StringVectorIt start, end, it ;
```

```
    // Initialize vector NamesVect
    NamesVect[0] = "Aladdin" ;
    NamesVect[1] = "Jasmine" ;
    NamesVect[2] = "Mickey" ;
    NamesVect[3] = "Minnie" ;
    NamesVect[4] = "Goofy" ;

    start = NamesVect.begin() ;   // location of first
                                  // element of NamesVect

    end = NamesVect.end() ;       // one past the location
                                  // last element of NamesVect

    // print content of NamesVect
    cout << "NamesVect { " ;
    for(it = start; it != end; it++)
        cout << *it << ", " ;
    cout << " }\n" << endl ;

    // Find the first name that is lexicographically greater
    // than the following name in the range [first, last + 1).
    // This version performs matching using binary predicate
    // function greater<string>
    location = adjacent_find(start, end, greater<string>()) ;

    // print the first pair of strings such that the first name is
    // lexicographically greater than the second.
    if (location != end)
        cout << "(" << *location << ", " << *(location + 1) << ")"
        << " the first pair of strings in NamesVect such that\n"
        << "the first name is lexicographically greater than "
        << "the second\n" << endl ;
    else
        cout << "No consecutive pair of strings found such that\n"
        << "the first name is lexicographically greater than "
        << "the second\n" << endl ;

}
```

Output

```
NamesVect { Aladdin, Jasmine, Mickey, Minnie, Goofy,  }

(Minnie, Goofy) the first pair of strings in NamesVect such that
the first name is lexicographically greater than the second
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

Predicate Versions of heap

Illustrates how to use the predicate versions of the **heap** STL function in Visual C++.

```
template<class RandomAccessIterator, class Compare> inline
   void make_heap(
      RandomAccessIterator First,
      RandomAccessIterator Last,
      Compare Compare
   )
template<class RandomAccessIterator, class Compare> inline
   void sort_heap(
      RandomAccessIterator First,
      RandomAccessIterator Last,
      Compare Compare
   )
template<class RandomAccessIterator, class Compare> inline
   void push_heap(
      RandomAccessIterator First,
      RandomAccessIterator Last,
      Compare Compare
   )
template<class RandomAccessIterator, class Compare> inline
   void pop_heap(
      RandomAccessIterator First,
      RandomAccessIterator Last,
      Compare Compare
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

A heap is a sequence of elements organized like a binary tree. Each heap element corresponds to a tree node. The first value in the sequence [*First*..*Last*) is the root and is ordered by the predicate. For example, if the predicate is greater, every element in the heap satisfies the following statement: Every element is greater than or equal to its parent. The smallest element is stored in the root, and all children hold progressively larger values. The make_heap function converts the range [*First*..*Last*) into a heap. The sort_heap function sorts a sequence that was created using the **make_heap** function. The push_heap function inserts a new value into the heap. The pop_heap function swaps the first and last elements in the heap specified by [*First*, *Last*), and then reduces the length of the sequence by one before restoring the heap property. The predicate versions of the heap functions use the compare function for comparisons.

Example

```
// heap_PVfunctions.cpp
// compile with: /EHsc
// Illustrates how to use the predicate versions
// of the make_heap, sort_heap, push_heap
// and pop_heap functions.
//
// Functions:
//
//    make_heap : Convert a sequence to a heap.
//    sort_heap : Sort a heap.
//    push_heap : Insert an element in a heap.
//    pop_heap  : Remove the top element from a heap.
//////////////////////////////////////////////////////////////////////

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
```

```
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std;


int main()
{
   const int VECTOR_SIZE = 8 ;

   // Define a template class vector of int
   typedef vector<int > IntVector ;

   //Define an iterator for template class vector of strings
   typedef IntVector::iterator IntVectorIt ;

   IntVector Numbers(VECTOR_SIZE) ;
   IntVectorIt it ;

   // Initialize vector Numbers
   Numbers[0] = 4 ;
   Numbers[1] = 10;
   Numbers[2] = 70 ;
   Numbers[3] = 10 ;
   Numbers[4] = 30 ;
   Numbers[5] = 69 ;
   Numbers[6] = 96 ;
   Numbers[7] = 100;

   // print content of Numbers
   cout << "Numbers { " ;
   for(it = Numbers.begin(); it != Numbers.end(); it++)
      cout << *it << " " ;
   cout << " }\n" << endl ;

   // convert Numbers into a heap
   make_heap(Numbers.begin(), Numbers.end(), greater<int>()) ;
   cout << "After calling make_heap\n" << endl ;

   // print content of Numbers
   cout << "Numbers { " ;
   for(it = Numbers.begin(); it != Numbers.end(); it++)
      cout << *it << " " ;
   cout << " }\n" << endl ;

   // sort the heapified sequence Numbers
   sort_heap(Numbers.begin(), Numbers.end(), greater<int>()) ;
   cout << "After calling sort_heap\n" << endl ;

   // print content of Numbers
   cout << "Numbers { " ;
   for(it = Numbers.begin(); it != Numbers.end(); it++)
      cout << *it << " " ;
   cout << " }\n" << endl ;

   make_heap(Numbers.begin(), Numbers.end(), greater<int>()) ;
   //insert an element in the heap
   Numbers.push_back(7) ;
   push_heap(Numbers.begin(), Numbers.end(), greater<int>()) ;
   cout << "After calling push_heap()\n" << endl;

   // print content of Numbers
   cout << "Numbers { " ;
   for(it = Numbers.begin(); it != Numbers.end(); it++)
```

```
      cout << *it << " " ;
   cout << " }\n" << endl ;

   //remove the root element from the heap Numbers
   pop_heap(Numbers.begin(), Numbers.end(), greater<int>()) ;
   cout << "After calling pop_heap\n" << endl ;

   // print content of Numbers
   cout << "Numbers { " ;
   for(it = Numbers.begin(); it != Numbers.end(); it++)
      cout << *it << " " ;
   cout << " }\n" << endl ;
}
```

Output

```
Numbers { 4 10 70 10 30 69 96 100  }

After calling make_heap

Numbers { 4 10 69 10 30 70 96 100  }

After calling sort_heap

Numbers { 100 96 70 69 30 10 10 4  }

After calling push_heap()

Numbers { 4 7 10 30 100 10 70 96 69  }

After calling pop_heap

Numbers { 7 30 10 69 100 10 70 96 4  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

Predicate Version of includes

Illustrates how to use the predicate version of the includes Standard Template Library (STL) function in Visual C++.

```
template<class InputIterator1, class InputIterator2, class Compare>
   inline bool includes(
      InputIterator1 First1,
      InputIterator1 Last1,
      InputIterator2 First2,
      InputIterator2 Last2,
      Compare Compare
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **includes** algorithm searches for one sequence of values in another sequence of values. **includes** returns **true** if every element in the range [*First2*..*Last2*) is in the sequence [*First1*..*Last1*). This version of **includes** assumes that both sequences are sorted using the **compare** function.

Example

```
// includesPV.cpp
// compile with: /EHsc
//
// Illustrates how to use the predicate version of
// includes function.
//
// Functions:
//    includes - Search for one sequence in another.
//    string_compare - Compare strings, return true if
//                        s1 < s2.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <string>
#include <vector>
#include <deque>

using namespace std ;

bool string_compare(const string& s1, const string& s2)
{
    return s1 < s2 ? 1 : 0;
}
int main()
{
    const int VECTOR_SIZE = 5 ;

    // Define a template class vector of strings
    typedef vector<string> StringVector ;

    //Define an iterator for template class vector of strings
    typedef StringVector::iterator StringVectorIt ;
```

```
    // Define a template class deque of strings
    typedef deque<string> StringDeque ;

    //Define an iterator for template class deque of strings
    typedef StringDeque::iterator StringDequeIt ;

    StringVector CartoonVector(VECTOR_SIZE) ;
    StringDeque CartoonDeque ;

    StringVectorIt start1, end1, it1 ;
    StringDequeIt start2, end2, it2 ;

    // Initialize vector Vector1
    CartoonVector[0] = "Aladdin" ;
    CartoonVector[1] = "Jasmine" ;
    CartoonVector[2] = "Mickey" ;
    CartoonVector[3] = "Minnie" ;
    CartoonVector[4] = "Goofy" ;

    start1 = CartoonVector.begin() ;  // location of first
                                      // element of CartoonVector

    end1 = CartoonVector.end() ;  // one past the location last
                                  // element of CartoonVector

    //Initialize list CartoonDeque
    CartoonDeque.push_back("Jasmine") ;
    CartoonDeque.push_back("Aladdin") ;
    CartoonDeque.push_back("Goofy") ;

    start2 = CartoonDeque.begin() ; // location of first
                                    // element of CartoonDeque

    end2 = CartoonDeque.end() ; // one past the location last
                                // element of CartoonDeque


    //sort CartoonVector and CartoonDeque alphabetically
    //includes requires the sequences
    //to be sorted.
    sort(start1, end1, string_compare) ;
    sort(start2, end2, string_compare) ;

    // print contents of CartoonVector and CartoonDeque
    cout << "CartoonVector { " ;
    for(it1 = start1; it1 != end1; it1++)
        cout << *it1 << ", " ;
    cout << " }\n" << endl ;
    cout << "CartoonDeque { " ;
    for(it2 = start2; it2 != end2; it2++)
        cout << *it2 << ", " ;
    cout << " }\n" << endl ;

    //Is CartoonDeque a subset of CartoonVector?
    if(includes(start1, end1, start2, end2, string_compare) )
        cout << "CartoonVector includes CartoonDeque"
        << endl ;
    else
        cout << "CartoonVector does not include CartoonDeque"
        << endl ;

  }
```

Output

```
CartoonVector { Aladdin, Goofy, Jasmine, Mickey, Minnie,  }

CartoonDeque { Aladdin, Goofy, Jasmine,  }
```

```
CartoonVector includes CartoonDeque
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

Predicate Version of inplace_merge

Illustrates how to use the predicate version of the inplace_merge Standard Template Library (STL) function in Visual C++.

```
template<class BidirectionalIterator, class Compare> inline
   void inplace_merge(
      BidirectionalIterator First,
      BidirectionalIterator Middle,
      BidirectionalIterator Last,
      Compare Compare
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **inplace_merge** algorithm merges two sorted subsequences: [*First*..*Middle*) and [*Middle*..*Last*) in place into a single sorted sequence [*First*..*Last*). This version assumes that the ranges [*First*..*Middle*) and [*Middle*..*Last*) are sorted using the **compare** function. If both ranges contain equal values, the value from the first range will be stored first.

Example

```
// inplace_mergeP.cpp
// compile with: /EHsc
// Illustrates how to use the predicate version of
// the inplace_merge function.
//
// Functions:
//    inplace_merge - Merge two sorted sub-sequences in place into a
//                    single sorted list using the compare function.
//
//    begin - Returns an iterator that points to the first element in a
//            sequence.

//    end - Returns an iterator that points one past the end of a sequence.
///////////////////////////////////////////////////////////////////////

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std;


int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of int
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of strings
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;
```

```
    IntVectorIt start, end, it ;

    // Initialize vector Numbers
    Numbers[0] = 4 ;
    Numbers[1] = 10;
    Numbers[2] = 70 ;
    Numbers[3] = 10 ;
    Numbers[4] = 30 ;
    Numbers[5] = 69 ;
    Numbers[6] = 96 ;
    Numbers[7] = 100;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    cout << "Before calling inplace_merge\n" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    //merge the elements of Numbers in place
    inplace_merge(start, start + 3, end, less<int>()) ;

    cout << "After calling inplace_merge\n" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;
}
```

Output

```
Before calling inplace_merge

Numbers { 4 10 70 10 30 69 96 100  }

After calling inplace_merge

Numbers { 4 10 10 30 69 70 96 100  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

Predicate Version of lower_bound

Illustrates how to use the predicate version of the lower_bound Standard Template Library (STL) function in Visual C++.

```
template<class ForwardIterator, class T, class Compare>
   inline ForwardIterator lower_bound(
      ForwardIterator First,
      ForwardIterator Last,
      const T& Value,
      Compare Compare
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **lower_bound** algorithm returns the first location in the sequence that value can be inserted such that the order of the sequence is maintained. **lower_bound** returns an iterator positioned at the location that value can be inserted in the range [*First*..*Last*), or returns *Last* if no such position exists. This version assumes the range [*First*..*Last*) is sorted sequentially using the **compare** function.

Example

```
// lower_boundPV.cpp
// compile with: /EHsc
// Illustrates how to use the lower_bound function.
//
// Functions:
//    lower_bound : Return the lower bound within a range.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore this warning
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std;


int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of int
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of strings
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;

    IntVectorIt start, end, it, location ;

    // Initialize vector Numbers
    Numbers[0] = 4 ;
    Numbers[1] = 10;
    Numbers[2] = 70 ;
```

```
    Numbers[3] = 10 ;
    Numbers[4] = 30 ;
    Numbers[5] = 69 ;
    Numbers[6] = 96 ;
    Numbers[7] = 100;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    //sort Numbers using the function object less<int>()
    //lower_bound assumes that Numbers is sorted
    //using the "compare" (less<int>() in this case)
    //function
    sort(start, end, less<int>()) ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // return the first location at which 10 can be inserted
    // in Numbers
    location = lower_bound(start, end, 10, less<int>()) ;

    cout << "First location element 10 can be inserted in Numbers is: "
        << location - start << endl ;
}
```

Output

```
Numbers { 4 10 10 30 69 70 96 100  }

First location element 10 can be inserted in Numbers is: 1
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

Predicate Version of max_element

Illustrates how to use the predicate version of the max_element Standard Template Library (STL) function in Visual C++.

```
template<class InputIterator, class Compare> inline
   InputIterator max_element(
      InputIterator First,
      InputIterator Last,
      Compare Compare
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **max_element** algorithm returns the location of the maximum element in the sequence [*First*, *Last*). The predicate version of **max_element** uses the **compare** function for comparisons.

Example

```
// max_elementPV.cpp
// compile with: /EHsc
// Illustrates how to use the predicates version
// of the max_element function.
//
// Functions:
//    max_element : Return the maximum element within a range.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std;


int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of int
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of strings
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;

    IntVectorIt start, end, it, location ;

    // Initialize vector Numbers
    Numbers[0] = 4 ;
    Numbers[1] = 10;
    Numbers[2] = 10 ;
    Numbers[3] = 30 ;
    Numbers[4] = 69 ;
    Numbers[5] = 70 ;
```

```
    Numbers[6] = 96 ;
    Numbers[7] = 100;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // return the maximum element in the Numbers
    location = max_element(start, end, less<int>()) ;

    cout << "The maximum element in Numbers is: "
        << *location << endl ;
}
```

Output

```
Numbers { 4 10 10 30 69 70 96 100  }

The maximum element in Numbers is: 100
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

Predicate Version of merge

Illustrates how to use the predicate version of the merge Standard Template Library (STL) function in Visual C++.

```
template<class InputIterator1,
   class InputIterator2,
   class OutputIterator
   class Compare> inline
   OutputIterator merge(
      InputIterator1 First1,
      InputIterator1 Last1,
      InputIterator2 First2,
      InputIterator2 Last2,
            OutputIterator Result,
      Compare Compare
   )
```

Remarks

Note
The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **merge** algorithm merges two sorted sequences: [*First1*..*Last1*) and [*First2*..*Last2*) into a single sorted sequence starting at *Result*. This version assumes that the ranges [*First1*..*Last1*) and [*First2*..*Last2*) are sorted using the **compare** function. If both ranges contain equal values, the value from the first range will be stored first. The result of merging overlapping ranges is undefined.

Example

```
// mergePV.cpp
// compile with: /EHsc
//
// Illustrates how to use predicate version of the merge function.
//
// Functions:
//    merge : Merge two sorted sequences
//            into a single sorted list.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <vector>
#include <list>
#include <deque>

using namespace std ;

int main()
{
    const int MAX_ELEMENTS = 8 ;

    // Define a template class vector of int
    typedef vector<int> IntVector ;

    //Define an iterator for template class vector of ints
    typedef IntVector::iterator IntVectorIt ;

    IntVector NumbersVector(MAX_ELEMENTS) ;
```

```
IntVectorIt startv, endv, itv ;

// Define a template class list of int
typedef list<int> IntList ;

//Define an iterator for template class list of ints
typedef IntList::iterator IntListIt ;

IntList NumbersList ;

IntListIt first, last, itl ;

// Define a template class deque of int
typedef deque<int> IntDeque ;

//Define an iterator for template class deque of ints
typedef IntDeque::iterator IntDequeIt ;

IntDeque NumbersDeque(2 * MAX_ELEMENTS) ;

IntDequeIt itd ;

// Initialize vector NumbersVector
NumbersVector[0] = 4 ;
NumbersVector[1] = 10;
NumbersVector[2] = 70 ;
NumbersVector[3] = 10 ;
NumbersVector[4] = 30 ;
NumbersVector[5] = 69 ;
NumbersVector[6] = 96 ;
NumbersVector[7] = 100;

startv = NumbersVector.begin() ;   // location of first
                                   // element of NumbersVector

endv = NumbersVector.end() ;  // one past the location
                              // last element of NumbersVector

// sort NumbersVector, merge requires the sequences
// to be sorted
sort(startv, endv, less<int>()) ;

// print content of NumbersVector
cout << "NumbersVector { " ;
for(itv = startv; itv != endv; itv++)
    cout << *itv << " " ;
cout << " }\n" << endl ;

// Initialize vector NumbersList
for(int i = 0; i < MAX_ELEMENTS; i++)
    NumbersList.push_back(i) ;

first = NumbersList.begin() ;   // location of first
                                // element of NumbersList

last = NumbersList.end() ;  // one past the location
                            // last element of NumbersList

// print content of NumbersList
cout << "NumbersList { " ;
for(itl = first; itl != last; itl++)
    cout << *itl << " " ;
cout << " }\n" << endl ;

// merge the elements of NumbersVector
// and NumbersList and place the
// results in NumbersDeque
```

```
    merge(startv, endv, first, last, NumbersDeque.begin(), less<int>()) ;

    cout << "After calling merge\n" << endl ;

    // print content of NumbersDeque
    cout << "NumbersDeque { " ;
    for(itd = NumbersDeque.begin();
        itd != NumbersDeque.end(); itd++)
        cout << *itd << " " ;
    cout << " }\n" << endl ;
}
```

Output

```
NumbersVector { 4 10 10 30 69 70 96 100  }

NumbersList { 0 1 2 3 4 5 6 7  }

After calling merge

NumbersDeque { 0 1 2 3 4 4 5 6 7 10 10 30 69 70 96 100  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

Predicate Version of min_element

Illustrates how to use the predicate version of the min_element Standard Template Library (STL) function in Visual C++.

```
template<class InputIterator, class Compare> inline
   InputIterator min_element(
      InputIterator First,
      InputIterator Last,
      Compare Compare
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **min_element** algorithm returns the location of the minimum element in the sequence [*First*, *Last*). The predicate version of **min_element** uses the **compare** function for comparisons.

Example

```
// min_elementPV.cpp
// compile with: /EHsc
// Illustrates how to use the predicates version
// of the min_element function.
//
// Functions:
//    min_element : Return the minimum element within a range.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std;


int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of int
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of strings
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;

    IntVectorIt start, end, it, location ;

    // Initialize vector Numbers
    Numbers[0] = 4 ;
    Numbers[1] = 10;
    Numbers[2] = 10 ;
    Numbers[3] = 30 ;
    Numbers[4] = 69 ;
    Numbers[5] = 70 ;
```

```
    Numbers[6] = 96 ;
    Numbers[7] = 100;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // return the minimum element in the Numbers
    location = min_element(start, end, less<int>()) ;

    cout << "The minimum element in Numbers is: "
        << *location << endl ;
}
```

Output

```
Numbers { 4 10 10 30 69 70 96 100  }

The minimum element in Numbers is: 4
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

Predicate Version of next_permutation

Illustrates how to use the predicate version of the next_permutation Standard Template Library (STL) function in Visual C++.

```
template<class BidirectionalIterator, class Compare> inline
   bool next_permutation(
      BidirectionalIterator First,
      BidirectionalIterator Last,
      Compare Compare
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **next_permutation** algorithm changes the order of the elements in the range [*First*, *Last*) to the next lexicographic permutation and returns **true**. If there is no **next_permutation**, it arranges the sequence to be the first permutation and returns **false**.

Note

The **next_permutation** algorithm assumes that the sequence is sorted in ascending order using the **compare** function. The nonpredicate version uses the **compare** function to order the permutations.

Example

```
// next_permutationPV.cpp
// compile with: /EHsc
// Illustrates how to use the predicate version
// of the next_permutation function.
//
// Functions:
//    next_permutation : Change the order of the sequence to the
//                       next lexicograhic permutation.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <functional>

using namespace std ;

int main()
{
    const int VECTOR_SIZE = 3 ;

    // Define a template class vector of strings
    typedef vector<string> StrVector ;

    //Define an iterator for template class vector of strings
    typedef StrVector::iterator StrVectorIt ;

    //Define an ostream iterator for strings
    typedef ostream_iterator<string> StrOstreamIt;
```

```
    StrVector Pattern(VECTOR_SIZE);

    StrVectorIt start, end, it;

    StrOstreamIt outIt(cout, " ");

   // location of first element of Pattern
    start = Pattern.begin();

   // one past the location last element of Pattern
    end = Pattern.end();

    // Initialize vector Pattern
    Pattern[0] = "K" ;
    Pattern[1] = "A" ;
    Pattern[2] = "L" ;

    // sort the contents of Pattern, required by next_permutation
    sort(start, end, less<string>()) ;

    // print content of Pattern
    cout << "Before calling next_permutation..." << endl << "Pattern:" ;
    for (it = start; it != end; it++)
        cout << " " << *it;
    cout << endl;

    // Generate all possible permutations

    cout << "After calling next_permutation...." << endl;
    while ( next_permutation(start, end, less<string>()) )
    {
        copy(start, end, outIt) ;
        cout << endl ;
    }
}
```

Sample Output

```
Before calling next_permutation:
Pattern: A K L

After calling next_permutation:.
A L K
K A L
K L A
L A K
L K A
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

Predicate Version of nth_element

Illustrates how to use the predicate version of the nth_element Standard Template Library (STL) function in Visual C++.

```
template<class RandomAccessIterator, class Compare>
   inline void nth_element(
      RandomAccessIterator First,
      RandomAccessIterator Nth,
      RandomAccessIterator Last,
      Compare compare
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **nth_element** algorithm partitions the sequence [*First*..*Last*) on the value referenced by *Nth*. All the elements less than or equal to the value are placed before value and all elements greater than value are placed after value in the sequence. The predicate version of **nth_element** uses the **compare** function for comparisons.

Example

```
// nth_elementPV.cpp
// compile with: /GX
// Illustrates how to use the predicate version
// of the nth_element function.
//
// Functions:
//    nth_element : Partition the elements in a sequence by its nth
//                  element.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std;


int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of int
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of strings
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;

    IntVectorIt start, end, it ;

    // Initialize vector Numbers
    Numbers[0] = 4 ;
    Numbers[1] = 10;
    Numbers[2] = 70 ;
```

```
    Numbers[3] = 30 ;
    Numbers[4] = 10;
    Numbers[5] = 69 ;
    Numbers[6] = 96 ;
    Numbers[7] = 100;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    cout << "Before calling nth_element:\n" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // partition the elements by the 4th element
    nth_element(start, start+4, end, less<int>()) ;

    cout << "After calling nth_element:\n" << endl ;

    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;
}
```

Output

```
Before calling nth_element:

Numbers { 4 10 70 30 10 69 96 100  }

After calling nth_element:

Numbers { 4 10 10 30 69 70 96 100  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

Predicate Version of partial_sort

Illustrates how to use the predicate version of the partial_sort Standard Template Library (STL) function in Visual C++.

```
template<class RandomAccessIterator, class Compare> inline
   void partial_sort(
      RandomAccessIterator First,
      RandomAccessIterator Middle,
      RandomAccessIterator Last,
      Compare Compare
   )
```

Remarks

Note
The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **partial_sort** algorithm sorts the smallest *n* elements, where *n* = *Middle* - *First* of the sequence [*First*, *Last*). The remaining elements end up in the range [*Middle*..*Last*) in an undefined order. The predicate version of **partial_sort** uses the **compare** function for comparisons.

Example

```
// partial_sortPV.cpp
// compile with: /EHsc
// Illustrates how to use the predicate version
// of the partial_sort function.
//
// Functions:
//    partial_sort : Sort the smallest N elements in a sequence.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std ;

int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of int
    typedef vector<int> IntVector ;

    //Define an iterator for template class vector of strings
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;

    IntVectorIt start, end, it ;

    // Initialize vector Numbers
    Numbers[0] = 4 ;
    Numbers[1] = 10;
    Numbers[2] = 70 ;
    Numbers[3] = 30 ;
    Numbers[4] = 10;
```

```
    Numbers[5] = 69 ;
    Numbers[6] = 96 ;
    Numbers[7] = 7;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    cout << "Before calling partial_sort\n" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // sort the smallest 4 elements in the sequence
    partial_sort(start, start+4, end, less<int>()) ;

    cout << "After calling partial_sort\n" << endl ;

    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;
}
```

Output

```
Before calling partial_sort

Numbers { 4 10 70 30 10 69 96 7  }

After calling partial_sort

Numbers { 4 7 10 10 70 69 96 30  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

Predicate Version of partial_sort_copy

Illustrates how to use the predicate version of the partial_sort_copy Standard Template Library (STL) function in Visual C++.

```
template<class InputIterator, RandomAccessIterator, class Compare> inline
   RandomAccessIterator partial_sort(
      InputIterator First1,
       InputIterator Last1,
      RandomAccessIterator First2,
      RandomAccessIterator Last2,
      Compare Compare
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **partial_sort_copy** algorithm sorts the smallest *n* elements, where *n* = **min**((*Last1* - *First1*), (*Last2* - *First2*)), of the sequence [*First1*, *Last1*), and copies the results to the sequence [*First2*, *First2* + *n*]. The predicate version of **partial_sort_copy** uses the **compare** function for comparisons.

Example

```
// partial_sort_copyP.cpp
// compile with: /EHsc
// Illustrates how to use the predicate version
// of the partial_sort_copy function.
//
// Functions:
//    partial_sort_copy : Sort the smallest N elements in a sequence
//                        and copy the resulting sequence
//                        to another sequence.
/////////////////////////////////////////////////////////////////////

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std ;

int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of int
    typedef vector<int> IntVector;

    //Define an iterator for template class vector of strings
    typedef IntVector::iterator IntVectorIt;

    IntVector Numbers(VECTOR_SIZE);
    IntVector Result(4);

    IntVectorIt start, end, it;

    // Initialize vector Numbers
```

```
    Numbers[0] = 4;
    Numbers[1] = 10;
    Numbers[2] = 70;
    Numbers[3] = 30;
    Numbers[4] = 10;
    Numbers[5] = 69;
    Numbers[6] = 96;
    Numbers[7] = 7;

    start = Numbers.begin();    // location of first
                                // element of Numbers

    end = Numbers.end();        // one past the location
                                // last element of Numbers

    cout << "Before calling partial_sort_copy\n" << endl;

    // print content of Numbers
    cout << "Numbers { ";
    for(it = start; it != end; it++)
        cout << *it << " ";
    cout << " }\n" << endl;

    // sort the smallest 4 elements in the Numbers
    // and copy the results in Result
    partial_sort_copy(start, end, Result.begin(),Result.end());

    cout << "After calling partial_sort_copy\n" << endl;

    cout << "Numbers { ";
    for(it = start; it != end; it++)
        cout << *it << " ";
    cout << " }\n" << endl;

    cout << "Result { " ;
    for(it = Result.begin(); it != Result.end(); it++)
        cout << *it << " ";
    cout << " }\n" << endl;
}
```

Output

```
Before calling partial_sort_copy

Numbers { 4 10 70 30 10 69 96 7  }

After calling partial_sort_copy

Numbers { 4 10 70 30 10 69 96 7  }

Result { 4 7 10 10  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

Predicate Version of random_shuffle

Illustrates how to use the predicate version of the random_shuffle Standard Template Library (STL) function in Visual C++.

```
template<class RandomAccessIterator, class Predicate> inline
   void random_shuffle(
      RandomAccessIterator First,
      RandomAccessIterator Last,
      Predicate Pred
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **random_shuffle** algorithm shuffles the elements of a sequence [*First*..*Last*) in a random order. The predicate version uses the *Pred* function to generate the indices of the elements to swap. The *Pred* has to be a function object that takes a parameter *n* and returns an integral random number in the range 0 - (*n* - 1). The predicate version of **random_shuffle** uses **operator=** to perform swaps.

Example

```
// random_shufflePV.cpp
// compile with: /EHsc
// Illustrates how to use the predicate version
// of the random_shuffle function.
//
// Functions:
//    random_shuffle: Shuffle the elements in a random order.
//
//      Rand: Given n, generates an integral random number in the
//          in the range 0 - (n - 1).
//////////////////////////////////////////////////////////////////////

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std;


// return an integral random number in the range 0 - (n - 1)
int Rand(int n)
{
    return rand() % n ;
}

int main()
{
   const int VECTOR_SIZE = 8 ;

   // Define a template class vector of int
   typedef vector<int > IntVector ;

   // Define an iterator for template class vector of strings
```

```
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;

    IntVectorIt start, end, it ;

    // Initialize vector Numbers
    Numbers[0] = 4 ;
    Numbers[1] = 10;
    Numbers[2] = 70 ;
    Numbers[3] = 30 ;
    Numbers[4] = 10;
    Numbers[5] = 69 ;
    Numbers[6] = 96 ;
    Numbers[7] = 100;

    // location of first element of Numbers
    start = Numbers.begin();

    // one past the location last element of Numbers
    end = Numbers.end();

    cout << "Before calling random_shuffle:" << endl;

    // print content of Numbers
    cout << "Numbers {";
    for (it = start; it != end; it++)
       cout << " "  << *it;
    cout << " }" << endl;

    // shuffle the elements in a random order.
    // the pointer_to_unary_function adapter converts a function to a
    // function object.
    random_shuffle(start, end, pointer_to_unary_function<int, int>(Rand));

    cout << "After calling random_shuffle:" << endl;

    cout << "Numbers {";
    for (it = start; it != end; it++)
       cout << " " << *it;
    cout << " }" << endl;
  }
```

Output

```
Before calling random_shuffle:
Numbers { 4 10 70 30 10 69 96 100 }
After calling random_shuffle:
Numbers { 10 10 96 70 4 69 100 30 }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

Predicate Version of upper_bound

Illustrates how to use the predicate version of the upper_bound Standard Template Library (STL) function in Visual C++.

```
template<class ForwardIterator, class T, class Compare>
   inline orwardIterator upper_bound(
      ForwardIterator First,
      ForwardIterator Last,
      const T& Value,
      Compare Compare
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **upper_bound** algorithm returns the last location in the sequence that value can be inserted such that the order of the sequence [*First*..*Last*) is maintained. **upper_bound** returns an iterator positioned at the location that value can be inserted in the range [*First*..*Last*), or returns *Last* if no such position exists. This version assumes the range [*First*..*Last*) is sorted sequentially using the **compare** function.

Example

```
// upper_boundPV.cpp
// compile with: /EHsc
// Illustrates how to use the predicate version
// of the upper_bound function.
//
// Functions:
//    upper_bound : Return the upper bound within a range.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std;


int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of int
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of strings
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;

    IntVectorIt start, end, it, location ;

    // Initialize vector Numbers
    Numbers[0] = 4 ;
    Numbers[1] = 10;
```

```
    Numbers[2] = 70 ;
    Numbers[3] = 10 ;
    Numbers[4] = 30 ;
    Numbers[5] = 69 ;
    Numbers[6] = 96 ;
    Numbers[7] = 100;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    //sort Numbers using the function object less<int>()
    //upper_bound assumes that Numbers is sorted
    //using the "compare" (less<int>() in this case)
    //function
    sort(start, end, less<int>()) ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    //return the highest location at which 10 can be inserted
    // in Numbers
    location = upper_bound(start, end, 10, less<int>()) ;

    cout << "Last location  for element 10 in Numbers is: "
        << location - start << endl ;
}
```

Output

```
Numbers { 4 10 10 30 69 70 96 100  }

Last location  for element 10 in Numbers is: 3
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

prev_permutation (STL Samples)

Illustrates how to use the prev_permutation Standard Template Library (STL) function in Visual C++.

```
template<class BidirectionalIterator> inline
   bool prev_permutation(
      BidirectionalIterator First,
      BidirectionalIterator Last
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **prev_permutation** algorithm changes the order of the elements the range [*First*, *Last*), to the previous lexicographic permutation and returns **true**. If there is no **prev_permutation**, it arranges the sequence to be the first permutation and returns **false**.

Note

The **prev_permutation** algorithm assumes the sequence is sorted in descending order using **operator<**. The nonpredicate version uses the **operator<** to order the permutations.

Example

```
// prev_permutation.cpp
// compile with: /EHsc
// Illustrates how to use the prev_permutation
// function.
//
// Functions:
//    prev_permutation : Change the order of the sequence to the
//                       previous lexicographic permutation.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <functional>

using namespace std ;

int main()
{
    const int VECTOR_SIZE = 3;

    // Define a template class vector of strings
    typedef vector<string> StrVector;

    // Define an iterator for template class vector of strings
    typedef StrVector::iterator StrVectorIt;

    // Define an ostream iterator for strings
    typedef ostream_iterator<string>
    StrOstreamIt;
```

```
    StrVector Pattern(VECTOR_SIZE);

    StrVectorIt start, end, it;

    StrOstreamIt outIt(cout, " ");

    // location of first element of Pattern
    start = Pattern.begin();

    // one past the location last element of Pattern
    end = Pattern.end();

    //Initialize vector Pattern
    Pattern[0] = "C";
    Pattern[1] = "B";
    Pattern[2] = "A";

    // print content of Pattern
    cout << "Before calling prev_permutation..." << endl << "Pattern: [";
    for (it = start; it != end; it++)
        cout << " " << *it;
    cout << " ]" << endl;

    // Generate all possible permutations
    cout << "After calling prev_permutation...." << endl;
    while ( prev_permutation(start, end) )
    {
        cout << "[ ";
        copy(start, end, outIt);
        cout << "]" << endl;
    }
}
```

Output

```
Before calling prev_permutation...
Pattern: [ C B A ]
After calling prev_permutation....
[ C A B ]
[ B C A ]
[ B A C ]
[ A C B ]
[ A B C ]
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

priority_queue Functions

Illustrates how to use the priority_queue::push, priority_queue::pop, priority_queue::empty, priority_queue::top, and priority_queue::size Standard Template Library (STL) functions in Visual C++.

```
priority_queue::push( );
priority_queue::pop( );
priority_queue::empty( );
priority_queue::top( );
priority_queue::size( );
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The sample shows the **priority_queue** implementation using deque and vector containers.

Example

```
// priority_queue.cpp
// compile with: /EHsc
//
// Functions:
//    priority_queue::push(), priority_queue::pop(),
//    priority_queue::empty(), priority_queue::top(), queue::size()

#include <iostream>
#include <queue>
#include <deque>
#include <vector>
#include <functional>

using namespace std ;

// Using priority_queue with deque
// Use of function less sorts the items in ascending order
typedef deque<int> INTDQU;
typedef priority_queue<int, INTDQU, less<int> > INTPRQUE;

// Using priority_queue with vector
// Use of function greater sorts the items in descending order
typedef vector<char> CHVECTOR;
typedef priority_queue<char, CHVECTOR, greater<char> > CHPRQUE;

int main(void)
{
    size_t size_q;
    INTPRQUE   q;
    CHPRQUE    p;

    // Insert items in the priority_queue(uses deque)
    q.push(42);
    q.push(100);
    q.push(49);
    q.push(201);

    // Output the item at the top using top()
    cout << q.top() << endl;

    // Output the size of priority_queue
    size_q = q.size();
```

```
    cout << "size of q is:" << size_q << endl;

   // Output items in priority_queue using top()
    // and use pop() to get to next item until
    // priority_queue is empty
    while (!q.empty())
    {
        cout << q.top() << endl;
        q.pop();

    }

// Insert items in the priority_queue(uses vector)
    p.push('c');
    p.push('a');
    p.push('d');
    p.push('m');
    p.push('h');

    // Output the item at the top using top()
    cout << p.top() << endl;

    // Output the size of priority_queue
    size_q = p.size();
    cout << "size of p is:" << size_q << endl;

    // Output items in priority_queue using top()
    // and use pop() to get to next item until
    // priority_queue is empty
    while (!p.empty())
    {
        cout << p.top() << endl;
        p.pop();

    }
}
```

Output

```
201
size of q is:4
201
100
49
42
a
size of p is:5
a
c
d
h
m
```

Requirements

Header: <queue>

See Also

Concepts

Standard Template Library Samples

queue Functions

Illustrates how to use the queue::push, queue::pop, queue::empty, queue::back, queue::front, and queue::size Standard Template Library (STL) functions in Visual C++.

```
queue::push( );
queue::pop( );
queue::empty( );
queue::back( );
queue::front( );
queue::size( );
```

Description

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The sample shows the **queue** implementation using list and deque containers.

Example

```
// queue.cpp
// compile with: /EHsc
//
// Functions:
//    queue::push(), queue::pop(), queue::empty(), queue::back(),
//    queue::front(),queue::size()

#include <list>
#include <iostream>
#include <queue>
#include <deque>

using namespace std ;

// Using queue with list
typedef list<int > INTLIST;
typedef queue<int>  INTQUEUE;

// Using queue with deque
typedef deque<char*> CHARDEQUE;
typedef queue<char*> CHARQUEUE;

int main(void)
{
    size_t size_q;
    INTQUEUE q;
    CHARQUEUE p;

    // Insert items in the queue(uses list)
    q.push(42);
    q.push(100);
    q.push(49);
    q.push(201);

    // Output the size of queue
    size_q = q.size();
    cout << "size of q is:" << size_q << endl;

    // Output items in queue using front()
    // and use pop() to get to next item until
    // queue is empty
```

```
    while (!q.empty())
    {
        cout << q.front() << endl;
        q.pop();
    }

// Insert items in the queue(uses deque)
    p.push("cat");
    p.push("ape");
    p.push("dog");
    p.push("mouse");
    p.push("horse");

    // Output the item inserted last using back()
    cout << p.back() << endl;

    // Output the size of queue
    size_q = p.size();
    cout << "size of p is:" << size_q << endl;

    // Output items in queue using front()
    // and use pop() to get to next item until
    // queue is empty
    while (!p.empty())
    {
        cout << p.front() << endl;
        p.pop();
    }
}
```

Output

```
size of q is:4
42
100
49
201
horse
size of p is:5
cat
ape
dog
mouse
horse
```

Requirements

Header: <queue>

See Also

Concepts

Standard Template Library Samples

random_shuffle (STL Samples)

Illustrates how to use the random_shuffle Standard Template Library (STL) function in Visual C++.

```
template<class RandomAccessIterator> inline
   void random_shuffle(
      RandomAccessIterator First,
      RandomAccessIterator Last
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **random_shuffle** algorithm shuffles the elements of a sequence [*First*..*Last*) in a random order. The nonpredicate version uses an internal random-number generator to generate the indices of the elements to swap. The nonpredicate version of **random_shuffle** uses **operator=** to perform swaps.

Example

```
// random_shuffle.cpp
// compile with: /EHsc
// Illustrates how to use the random_shuffle
// function.
//
// Functions:
//     random_shuffle : Shuffle the elements in a random order.
//////////////////////////////////////////////////////////////////////

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std ;

int main() {
   const int VECTOR_SIZE = 8 ;

   // Define a template class vector of int
   typedef vector<int> IntVector ;

   //Define an iterator for template class vector of strings
   typedef IntVector::iterator IntVectorIt ;

   IntVector Numbers(VECTOR_SIZE) ;

   IntVectorIt start, end, it ;

   // Initialize vector Numbers
   Numbers[0] = 4 ;
   Numbers[1] = 10;
   Numbers[2] = 70 ;
   Numbers[3] = 30 ;
   Numbers[4] = 10;
   Numbers[5] = 69 ;
   Numbers[6] = 96 ;
```

```
    Numbers[7] = 100;

    // location of first element of Numbers
    start = Numbers.begin();

    // one past the location last element of Numbers
    end = Numbers.end();

    cout << "Before calling random_shuffle" << endl ;

    // print content of Numbers
    cout << "Numbers {" ;
    for (it = start; it != end; it++)
       cout << " " << *it;
    cout << " }" << endl ;

    // shuffle the elements in a random order
    random_shuffle(start, end) ;

    cout << "After calling random_shuffle" << endl ;

    cout << "Numbers {" ;
    for (it = start; it != end; it++)
       cout << " " << *it;
    cout << " }" << endl ;
}
```

Sample Output

```
Before calling random_shuffle
Numbers { 4 10 70 30 10 69 96 100 }
After calling random_shuffle
Numbers { 10 10 96 70 4 69 100 30 }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

remove (STL Samples)

Illustrates how to use the remove Standard Template Library (STL) function in Visual C++.

```
template<class ForwardIterator, class Type> inline
   ForwardIterator remove(
      ForwardIterator First,
      ForwardIterator Last,
      const T& Value
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **remove** algorithm removes all elements that match value from the range (*First*, *Last*). It returns an iterator equal to *Last* - *n*, where *n* = number of elements removed. The last *n* elements of the range have undefined values. The size of the container remains the same.

Example

```
// remove.cpp
// compile with: /EHsc
// Illustrates how to use the remove function.
//
// Functions:
//   remove - remove all elements from the sequence that match value.
//   begin - Returns an iterator that points to the first element in a
//           sequence.
//   end - Returns an iterator that points one past the end of a sequence.
//////////////////////////////////////////////////////////////////////

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;


int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of integers
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of integer
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;   //vector containing numbers

    IntVectorIt start, end, it, last;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers
```

```
    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    //Initialize vector Numbers
    Numbers[0] = 10 ;
    Numbers[1] = 20 ;
    Numbers[2] = 10 ;
    Numbers[3] = 15 ;
    Numbers[4] = 12 ;
    Numbers[5] = 7 ;
    Numbers[6] = 9 ;
    Numbers[7] = 10 ;


    cout << "Before calling remove" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // remove all elements from Numbers that match 10
     last = remove(start, end, 10) ;

    cout << "After calling remove" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    //print number of elements removed from Numbers
    cout << "Total number of elements removed from Numbers = "
        << end - last << endl ;

    //print only the valid elements of Number
    cout << "Valid elements of Numbers { " ;
    for(it = start; it != last; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

}
```

Output

```
Before calling remove
Numbers { 10 20 10 15 12 7 9 10  }

After calling remove
Numbers { 20 15 12 7 9 7 9 10  }

Total number of elements removed from Numbers = 3
Valid elements of Numbers { 20 15 12 7 9  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

remove_copy (STL Samples)

Illustrates how to use the remove_copy Standard Template Library (STL) function in Visual C++.

```
template<class ForwardIterator, class OutputIterator, class Type> inline
   OutputIterator remove_copy(
      ForwardIterator First,
      ForwardIterator Last,
      OutputIterator Result,
      const T& Value
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **remove_copy** algorithm copies all elements from the range (*First*, *Last*) to the range starting at *Result*, skipping any element that matches *Value*. It returns an iterator positioned immediately after the last new element.

Example

```
// remove_copy.cpp
// compile with: /EHsc
// Illustrates how to use the remove_copy function.
//
// Functions:
//   remove_copy - copy the elements of a sequence to another sequence
//                 eliminating any elements that match value.
//   begin - Returns an iterator that points to the first element in a
//           sequence.
//   end - Returns an iterator that points one past the end of a sequence.
//////////////////////////////////////////////////////////////////////

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;


int main()
{
   const int MAX_ELEMENTS = 8 ;

   // Define a template class vector of integers
   typedef vector<int > IntVector ;

   //Define an iterator for template class vector of integer
   typedef IntVector::iterator IntVectorIt ;

   //vector containing numbers
   IntVector Numbers(MAX_ELEMENTS), Result(MAX_ELEMENTS) ;

   IntVectorIt start, end, it, last, resultIt ;

   //Initialize vector Numbers
```

```
    Numbers[0] = 10 ;
    Numbers[1] = 20 ;
       Numbers[2] = 10 ;
    Numbers[3] = 15 ;
    Numbers[4] = 12 ;
    Numbers[5] = 7 ;
    Numbers[6] = 9 ;
    Numbers[7] = 10 ;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    resultIt = Result.begin() ; // location of first
                                // element of Result

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
       cout << *it << " " ;
    cout << " }\n" << endl ;

    // copy all elements from Numbers to Result
    // skipping any item that equals 10
    last = remove_copy(start, end, resultIt, 10) ;

    //print number of elements copied to Result
    cout << "Total number of elements copied to Result = "
       << last - resultIt << endl ;

    start = Result.begin() ;   // location of first
                               // element of Result

    end = Result.end() ;       // one past the location
                               // last element of Result

    // print content of Result
    cout << "Result { " ;
    for(it = start; it != end; it++)
       cout << *it << " " ;
    cout << " }\n" << endl ;
 }
```

Output

```
Numbers { 10 20 10 15 12 7 9 10  }

Total number of elements copied to Result = 5
Result { 20 15 12 7 9 0 0 0  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

remove_copy_if (STL Samples)

Illustrates how to use the remove_copy_if Standard Template Library (STL) function in Visual C++.

```
template<class ForwardIterator, class OutputIterator, class Predicate>
   inline OutputIterator remove_copy_if(
      ForwardIterator First,
      ForwardIterator Last,
      OutputIterator Result,
      Predicate Pred
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **remove_copy_if** algorithm copies all elements from the range (*First*, *Last*) to the range starting at *Result*, skipping any element that causes the predicate to return **true**. It returns an iterator positioned immediately after the last new element.

Example

```
// remove_copy_if.cpp
// compile with: /EHsc
// Illustrates how to use the remove_copy_if function.
//
// Functions:
//   remove_copy_if - copy the elements of a sequence to another sequence
//                    eliminating any elements that satisfy a predicate.
//   bind2nd - Returns true for elements for which the condition is true
//   begin - Returns an iterator that points to the first element in a
//           sequence
//   end - Returns an iterator that points one past the end of a sequence
//////////////////////////////////////////////////////////////////////

// disable warning C4786: symbol greater than 255 characters,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;


int main()
{
    const int MAX_ELEMENTS = 8 ;

    // Define a template class vector of integers
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of integer
    typedef IntVector::iterator IntVectorIt ;

    //vector containing numbers
    IntVector Numbers(MAX_ELEMENTS), Result(MAX_ELEMENTS) ;

    IntVectorIt start, end, it, last, resultIt ;
```

```
    //Initialize vector Numbers
    Numbers[0] = 10 ;
    Numbers[1] = 20 ;
    Numbers[2] = 10 ;
    Numbers[3] = 15 ;
    Numbers[4] = 12 ;
    Numbers[5] = 25 ;
    Numbers[6] = 30 ;
    Numbers[7] = 10 ;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    resultIt = Result.begin() ; // location of first
                                // element of Result

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // copy all elements from Numbers to Result
    // skipping any item that >= 25
    last = remove_copy_if(start, end, resultIt,
                          bind2nd(greater_equal<int>(), 25)) ;

    //print number of elements copied to Result
    cout << "Total number of elements copied to Result = "
        << last - resultIt << endl ;

    start = Result.begin() ;   // location of first
                               // element of Result

    end = Result.end() ;       // one past the location
                               // last element of Result

    // print content of Result
    cout << "Result { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;
}
```

Output

```
Numbers { 10 20 10 15 12 25 30 10  }

Total number of elements copied to Result = 6
Result { 10 20 10 15 12 10 0 0  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

remove_if (STL Samples)

Illustrates how to use the remove_if Standard Template Library (STL) function in Visual C++.

```
template<class ForwardIterator, class Predicate> inline
   ForwardIterator remove_if(
      ForwardIterator First,
      ForwardIterator Last,
      Predicate Pred
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **remove_if** algorithm removes all elements from the range (*First*, *Last*) that cause the predicate to return **true**. It returns an iterator equal to *Last* - *n*, where *n* = number of elements removed. The last *n* elements of the range have undefined values. The size of the container remains the same.

Example

```
// remove_if.cpp
// compile with: /EHsc
// Illustrates how to use the remove_if function.
//
// Functions:
//   remove_if - remove all elements from the sequence that
//               satisfies a predicate.
//   bind2nd - Returns true for elements for which the condition is true
//   begin - Returns an iterator that points to the first element in a
//           sequence
//   end - Returns an iterator that points one past the end of a sequence
//////////////////////////////////////////////////////////////////////

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;


int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of integers
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of integer
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;   //vector containing numbers

    IntVectorIt start, end, it, last;

    start = Numbers.begin() ;   // location of first
```

```
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    //Initialize vector Numbers
    Numbers[0] = 10 ;
    Numbers[1] = 20 ;
    Numbers[2] = 10 ;
    Numbers[3] = 15 ;
    Numbers[4] = 12 ;
    Numbers[5] = 7 ;
    Numbers[6] = 9 ;
    Numbers[7] = 10 ;


    cout << "Before calling remove_if" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // remove all elements from Numbers that <= 10
     last = remove_if(start, end, bind2nd(less_equal<int>(), 10) ) ;

    cout << "After calling remove_if" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    //print number of elements removed from Numbers
    cout << "Total number of elements removed from Numbers = "
        << end - last << endl ;

    //print only the valid elements of Number
    cout << "Valid elements of Numbers { " ;
    for(it = start; it != last; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

}
```

Output

```
Before calling remove_if
Numbers { 10 20 10 15 12 7 9 10  }

After calling remove_if
Numbers { 20 15 12 15 12 7 9 10  }

Total number of elements removed from Numbers = 5
Valid elements of Numbers { 20 15 12  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

replace (STL Samples)

Illustrates how to use the replace Standard Template Library (STL) function in Visual C++.

The **replace** algorithm replaces all elements that match *Old_Value* from the range [*First*, *Last*) with *New_Value*.

Example

```
// replace.cpp
// compile with: /EHsc
// Illustrates how to use the replace function.
//
// Functions:
//    replace - Replace all elements from the sequence that match value
//              with another value.

// disable warning C4786: symbol greater than 255 characters,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;


int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of integers
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of integer
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;   //vector containing numbers

    IntVectorIt start, end, it ;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    //Initialize vector Numbers
    Numbers[0] = 10 ;
    Numbers[1] = 20 ;
    Numbers[2] = 10 ;
    Numbers[3] = 15 ;
    Numbers[4] = 12 ;
    Numbers[5] = 7 ;
    Numbers[6] = 9 ;
    Numbers[7] = 10 ;

    cout << "Before calling replace" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;
```

```
    // remove all elements from Numbers that match 10
     replace(start, end, 10, 35) ;

    cout << "After calling replace, to replace all 10's with 35" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

}
```

Output

```
Before calling replace
Numbers { 10 20 10 15 12 7 9 10  }

After calling replace, to replace all 10's with 35
Numbers { 35 20 35 15 12 7 9 35  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

replace_copy (STL Samples)

Illustrates how to use the replace_copy Standard Template Library (STL) function in Visual C++.

```
template<class ForwardIterator, class OutputIterator, class Type> inline
   OutputIterator replace_copy(
      ForwardIterator First,
      ForwardIterator Last,
      OutputIterator Result,
      const T& Old_Value,
      const T& New_Value
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **replace_copy** algorithm copies all elements from the range [*First*, *Last*) to a same-size range starting at result, replacing all occurrences of *Old_Value* with *New_Value* in the resulting sequence. It returns an iterator positioned immediately after the last new element in the resulting sequence.

Example

```
// replace_copy.cpp
// compile with: /EHsc
// Illustrates how to use the replace_copy function.
//
// Functions:
//    replace_copy - Copy the elements of a sequence to another
//                   same-size sequence replacing any elements
//                   that match a value with another value.

// disable warning C4786: symbol greater than 255 characters,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;


int main()
{
    const int MAX_ELEMENTS = 8 ;

    // Define a template class vector of integers
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of integer
    typedef IntVector::iterator IntVectorIt ;

    //vector containing numbers
    IntVector Numbers(MAX_ELEMENTS), Result(MAX_ELEMENTS) ;

    IntVectorIt start, end, it, last, resultIt ;

    //Initialize vector Numbers
    Numbers[0] = 10 ;
```

```
    Numbers[1] = 20 ;
    Numbers[2] = 10 ;
    Numbers[3] = 15 ;
    Numbers[4] = 12 ;
    Numbers[5] = 7 ;
    Numbers[6] = 9 ;
    Numbers[7] = 10 ;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    resultIt = Result.begin() ; // location of first
                                // element of Result

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // copy all elements from Numbers to Result
    // replacing any item that equals 10 with 30
    last = replace_copy(start, end, resultIt, 10, 30) ;

    //print number of elements copied to Result
    cout << "Total number of elements copied to Result = "
        << last - resultIt << endl ;

    start = Result.begin() ;   // location of first
                               // element of Result

    end = Result.end() ;       // one past the location
                               // last element of Result

    // print content of Result
    cout << "Result  { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

}
```

Output

```
Numbers { 10 20 10 15 12 7 9 10  }

Total number of elements copied to Result = 8
Result  { 30 20 30 15 12 7 9 30  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

replace_copy_if (STL Samples)

Illustrates how to use the replace_copy_if Standard Template Library (STL) function in Visual C++.

```
template<
   class ForwardIterator,
   class OutputIterator,
   class Predicate,
   class Type
> inline
   OutputIterator replace_copy_if(
      ForwardIterator First,
      ForwardIterator Last,
      OutputIterator Result,
      Predicate Pred,
      const T& New_Value
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **replace_copy_if** algorithm copies all elements from the range [*First*, *Last*) to a same-size range starting at result, replacing all occurrences that cause the predicate to return **true** with *New_Value* in the resulting sequence. It returns an iterator positioned immediately after the last new element in the resulting sequence.

Example

```
// replace_copy_if.cpp
// compile with: /EHsc
// Illustrates how to use the replace_copy_if function.
//
// Functions:
//    replace_copy_if - Copy the elements of a sequence to another
//                      same-size sequence replacing any elements
//                      that satisfies a predicate, with another value.


// disable warning C4786: symbol greater than 255 characters,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;


int main()
{
    const int MAX_ELEMENTS = 8 ;

    // Define a template class vector of integers
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of integer
    typedef IntVector::iterator IntVectorIt ;

    //vector containing numbers
```

```
    IntVector Numbers(MAX_ELEMENTS), Result(MAX_ELEMENTS) ;

    IntVectorIt start, end, it, last, resultIt ;

    //Initialize vector Numbers
    Numbers[0] = 10 ;
    Numbers[1] = 20 ;
    Numbers[2] = 10 ;
    Numbers[3] = 15 ;
    Numbers[4] = 12 ;
    Numbers[5] = 7 ;
    Numbers[6] = 9 ;
    Numbers[7] = 10 ;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    resultIt = Result.begin() ; // location of first
                                // element of Result

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // copy all elements from Numbers to Result
    // replacing any item that >= 10 with 30
    last = replace_copy_if(start, end, resultIt,
                           bind2nd(greater_equal<int>(), 10), 30) ;

    //print number of elements copied to Result
    cout << "Total number of elements copied to Result = "
        << last - resultIt << endl ;

    start = Result.begin() ;   // location of first
                               // element of Result

    end = Result.end() ;       // one past the location
                               // last element of Result

    // print content of Result
    cout << "Result  { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

}
```

Output

```
Numbers { 10 20 10 15 12 7 9 10  }

Total number of elements copied to Result = 8
Result  { 30 30 30 30 30 7 9 30  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

replace_if (STL Samples)

Illustrates how to use the replace_if Standard Template Library (STL) function in Visual C++.

```
template<class ForwardIterator, class Predicate, class Type> inline
   void replace_if(
      ForwardIterator First,
      ForwardIterator Last,
      Predicate Pred,
      const Type& Value
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **replace_if** algorithm replaces all elements from the range [*First*, *Last*) that cause the predicate to return **true** with *Value*.

Example

```
// replace_if.cpp
// compile with: /EHsc
// Illustrates how to use the replace_if function.
//
// Functions:
//    replace_if - Replace all elements from the sequence that
//                 satisfies a predicate with a specified value.


// disable warning C4786: symbol greater than 255 characters,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;


int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of integers
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of integer
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;   //vector containing numbers

    IntVectorIt start, end, it ;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers
```

```
    //Initialize vector Numbers
    Numbers[0] = 10 ;
    Numbers[1] = 20 ;
    Numbers[2] = 10 ;
    Numbers[3] = 15 ;
    Numbers[4] = 12 ;
    Numbers[5] = 7 ;
    Numbers[6] = 9 ;
    Numbers[7] = 10 ;

    cout << "Before calling replace_if" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    // replace all elements from Numbers that are <= 10 with 4
     replace_if(start, end, bind2nd(less_equal<int>(), 10), 4 ) ;

    cout << "After calling replace_if" << endl ;

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

}
```

Output

```
Before calling replace_if
Numbers { 10 20 10 15 12 7 9 10  }

After calling replace_if
Numbers { 4 20 4 15 12 4 4 4  }
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

reverse (STL Samples)

Illustrates how to use the reverse Standard Template Library (STL) function in Visual C++.

```
template<class BidirectionalIterator> inline
   void reverse(
      BidirectionalIterator First,
      BidirectionalIterator Last
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **reverse** algorithm reverses the order of the elements in the range [*First*, *Last*).

Example

```
// reverse.cpp
// compile with: /EHsc
// Illustrates how to use the reverse function.
//
// Functions:
//    reverse - Reverse the items in a sequence.

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <functional>

using namespace std;


int main()
{
   const int VECTOR_SIZE = 8;

   // Define a template class vector of strings
   typedef vector<string > StrVector;

   //Define an iterator for template class vector of strings
   typedef StrVector::iterator StrVectorIt;

   StrVector Tongue_Twister(VECTOR_SIZE);
   StrVectorIt start, end, it;

   // location of first element of Tongue_Twister
   start = Tongue_Twister.begin();

   // one past the location last element of Tongue_Twister
   end = Tongue_Twister.end();

   //Initialize vector Tongue_Twister
   Tongue_Twister[0] = "she";
   Tongue_Twister[1] = "sells";
   Tongue_Twister[2] = "sea";
```

```
    Tongue_Twister[3] = "shells";
    Tongue_Twister[4] = "by";
    Tongue_Twister[5] = "the";
    Tongue_Twister[6] = "sea";
    Tongue_Twister[7] = "shore";

    cout << "Before calling reverse" << endl;

    // print content of Tongue_Twister
    cout << "Try this Tongue Twister:";
    for (it = start; it != end; it++)
       cout << " " << *it;
    cout << endl;

    // reverse the items in the vector Tongue_Twister
    reverse(start, end);

    cout << "After calling reverse" << endl;

    // print content of Tongue_Twister
    cout << "Now try the reversed Tongue Twister:";
    for (it = start; it != end; it++)
       cout << " " << *it;
 }
```

Output

```
Before calling reverse
Try this Tongue Twister: she sells sea shells by the sea shore
After calling reverse
Now try the reversed Tongue Twister: shore sea the by shells sea sells she
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

reverse_copy (STL Samples)

Illustrates how to use the reverse_copy Standard Template Library (STL) function in Visual C++.

```
template<class BidirectionalIterator, class OutputIterator> inline
   OutputIterator reverse_copy(
      BidirectionalIterator First,
      BidirectionalIterator Last,
      OutputIterator Result
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **reverse_copy** algorithm reverses the order of the elements in the range [*First*, *Last*) and copies the result into a sequence of the same size starting at *Result*. It returns an iterator positioned immediately after the last new element in the resulting sequence.

Example

```
// reverse_copy.cpp
// compile with: /EHsc
// Illustrates how to use the reverse_copy function.
//
// Functions:
//    reverse_copy - Reverse a sequence, copy the results to another
//                   same-sized sequence.
//
//////////////////////////////////////////////////////////////////////

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <functional>

using namespace std ;

int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of strings
    typedef vector<string> StrVector ;

    //Define an iterator for template class vector of strings
    typedef StrVector::iterator StrVectorIt ;

    StrVector Tongue_Twister(VECTOR_SIZE) ;
    StrVector Reversed_Twister(VECTOR_SIZE) ;

    StrVectorIt start, end, it, RTstart, RTend ;

    start = Tongue_Twister.begin() ; // location of first
                                     // element of Tongue_Twister
```

```
    end = Tongue_Twister.end() ;     // one past the location last
                                     // element of Tongue_Twister

    RTstart = Reversed_Twister.begin() ; // location of first
                                         // element of Reversed_Twister

    RTend = Reversed_Twister.end() ; // one past the location last
                                     // element of Reversed_Twister

    //Initialize vector Tongue_Twister
    Tongue_Twister[0] = "she";
    Tongue_Twister[1] = "sells";
    Tongue_Twister[2] = "sea";
    Tongue_Twister[3] = "shells";
    Tongue_Twister[4] = "by";
    Tongue_Twister[5] = "the";
    Tongue_Twister[6] = "sea";
    Tongue_Twister[7] = "shore";

    cout << "Before calling reverse_copy" << endl ;

    // print content of Tongue_Twister
    cout << "Try this Tongue Twister:";
    for(it = start; it != end; it++)
        cout << " " << *it;

    // reverse the items in the vector Tongue_Twister
    // and copy the results to Reversed_Twister
    reverse_copy(start, end, RTstart);

    cout << "\n\nAfter calling reverse_copy" << endl ;

    // print content of Tongue_Twister
    cout << "Tongue_Twister: ";
    for(it = start; it != end; it++)
        cout << *it << " ";

    // print content of Reversed_Twister
    cout << "\n\nNow try the reversed Tongue Twister:" ;
    for(it = RTstart; it != RTend; it++)
        cout << " " << *it;
}
```

Output

```
Before calling reverse_copy
Try this Tongue Twister: she sells sea shells by the sea shore

After calling reverse_copy
Tongue_Twister: she sells sea shells by the sea shore

Now try the reversed Tongue Twister: shore sea the by shells sea sells she
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

rotate (STL Samples)

Illustrates how to use the rotate Standard Template Library (STL) function in Visual C++.

```
template<class ForwardIterator> inline
   void rotate(
      ForwardIterator First,
      ForwardIterator Middle,
      ForwardIterator Last
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **rotate** algorithm rotates the elements in the range [*First*, *Last*) to the right by *n* positions, where *n* = *Middle* - *First*.

Example

```
// rotate.cpp
// compile with: /EHsc
// Illustrates how to use the rotate function.
//
// Functions:
//    rotate - Rotate the items in a sequence by n positions.
//
////////////////////////////////////////////////////////////////////

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <functional>

using namespace std;

int main()
{
   const int VECTOR_SIZE = 8;

   // Define a template class vector of strings
   typedef vector<string> StrVector;

   //Define an iterator for template class vector of strings
   typedef StrVector::iterator StrVectorIt;

   StrVector Tongue_Twister(VECTOR_SIZE);

   StrVectorIt start, end, middle, it;

   // location of first element of Tongue_Twister
   start = Tongue_Twister.begin();

   // one past the location last element of Tongue_Twister
   end = Tongue_Twister.end();
```

```
    // Initialize vector Tongue_Twister
    Tongue_Twister[0] = "she";
    Tongue_Twister[1] = "sells";
    Tongue_Twister[2] = "sea";
    Tongue_Twister[3] = "shells";
    Tongue_Twister[4] = "by";
    Tongue_Twister[5] = "the";
    Tongue_Twister[6] = "sea";
    Tongue_Twister[7] = "shore";

    middle = start + 3;  // start position for rotating elements

    cout << "Before calling rotate" << endl;

    // print content of Tongue_Twister
    cout << "Try this Tongue Twister:";
    for (it = start; it != end; it++)
       cout << " " << *it;

    // rotate the items in the vector Tongue_Twister by 3 positions
    rotate(start, middle, end);

    cout << endl << "After calling rotate" << endl;

    // print content of Tongue_Twister
    cout << "Now try the rotated Tongue Twister:";
    for (it = start; it != end; it++)
       cout << " " << *it;
    cout << endl;
  }
```

Output

```
Before calling rotate
Try this Tongue Twister: she sells sea shells by the sea shore
After calling rotate
Now try the rotated Tongue Twister: shells by the sea shore she sells sea
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

rotate_copy (STL Samples)

Illustrates how to use the rotate_copy Standard Template Library (STL) function in Visual C++.

```
template<class ForwardIterator, class OutputIterator> inline
   OutputIterator rotate_copy(
      BidirectionalIterator First,
      BidirectionalIterator Middle,
      BidirectionalIterator Last,
      OutputIterator Result
   )
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **rotate_copy** algorithm rotates the elements in the range [*First*, *Last*) to the right by *n* positions (where *n* = *Middle* - *First*), and copies the result into a sequence of the same size, starting at result. It returns an iterator positioned immediately after the last new element in the resulting sequence.

Note

The **OutputIterator** should be different from the sequence to be rotated. If they are the same, the result will depend on the implementation.

Example

```
// rotate_copy.cpp
// compile with: /EHsc
// Illustrates how to use the rotate_copy function.
//
// Functions:
//    rotate_copy - Rotate a sequence by n positions, copy the
//                  results to another same sized sequence.
//
///////////////////////////////////////////////////////////////////////

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <functional>

using namespace std ;

int main()
{
   const int VECTOR_SIZE = 8 ;

   // Define a template class vector of strings
   typedef vector<string> StrVector ;

   //Define an iterator for template class vector of strings
   typedef StrVector::iterator StrVectorIt ;

   StrVector Tongue_Twister(VECTOR_SIZE) ;
```

```
    StrVector Rotated_Twister(VECTOR_SIZE) ;

    StrVectorIt start, middle, end, it, RTstart, RTend ;

    // location of first element of Tongue_Twister
    start = Tongue_Twister.begin() ;

    // one past the location last element of Tongue_Twister
    end = Tongue_Twister.end() ;

    // start position for rotating elements
    middle = start + 3 ;

    // location of first element of Rotated_Twister
    RTstart = Rotated_Twister.begin() ;

    // one past the location last element of Rotated_Twister
    RTend = Rotated_Twister.end() ;

    // Initialize vector Tongue_Twister
    Tongue_Twister[0] = "she" ;
    Tongue_Twister[1] = "sells" ;
    Tongue_Twister[2] = "sea" ;
    Tongue_Twister[3] = "shells" ;
    Tongue_Twister[4] = "by";
    Tongue_Twister[5] = "the";
    Tongue_Twister[6] = "sea" ;
    Tongue_Twister[7] = "shore" ;

    cout << "Before calling rotate_copy:" << endl ;

    // print content of Tongue_Twister
    cout << "Try this Tongue Twister:" ;
    for (it = start; it != end; it++)
       cout << " " << *it;

    // rotate the items in the vector Tongue_Twist to the right by
    // 3 positions and copy the results to Rotated_Twister
    rotate_copy(start, middle, end, RTstart) ;

    cout << endl << "After calling rotate_copy:" << endl ;

    // print content of Tongue_Twister
    cout << "Tongue_Twister: " ;
    for (it = start; it != end; it++)
       cout << " " << *it;

    // print content of Rotated_Twister
    cout << endl << "Now try the rotated Tongue Twister:" ;
    for (it = RTstart; it != RTend; it++)
       cout << " " << *it;
    cout << endl;
  }
```

Output

```
Before calling rotate_copy:
Try this Tongue Twister: she sells sea shells by the sea shore
After calling rotate_copy:
Tongue_Twister:  she sells sea shells by the sea shore
Now try the rotated Tongue Twister: shells by the sea shore she sells sea
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

set::key_comp and set::value_comp

Illustrates how to use the set::key_comp and set::value_comp Standard Template Library (STL) functions in Visual C++.

```
template<class _K, class _Pr, class _A>
   class set
   {
      public:
      // Function 1:
      key_compare key_comp( ) const;
      // Function 2:
      value_compare value_comp( ) const;
   }
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **key_comp** function returns the stored function object that determines the order of elements in the controlled sequence. The **value_comp** function returns the same function that the **key_comp** function returns.

Example

```
// SetComp.cpp
// compile with: /EHsc
//
//      Illustrates how to use the key_comp function to obtain a
//      function pointer that is the stored function object that
//      determines the order of elements in the controlled sequence.
//      It also illustrates how to use the value_comp function to
//      obtain a function pointer that is the stored function object
//      that determines the order of the elements in the controlled
//      sequence (same as key_comp result).
//
// Functions:
//
//    key_comp     Returns a function pointer to the function that
//                 determines the order of elements in the controlled
//                 sequence.
//    value_comp   Returns a function pointer to the function that
//                 determines the order of elements in the controlled
//                 sequence (same as key_comp).
//////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <set>
#include <iostream>

using namespace std ;

typedef set<int> SET_INT;

void truefalse(int x)
{
  cout << (x?"True":"False") << endl;
}

int main() {
  SET_INT s1;

  cout << "s1.key_comp()(8,10) returned ";
```

```
    truefalse(s1.key_comp()(8,10));  // True

    cout << "s1.value_comp()(8,10) returned ";
    truefalse(s1.value_comp()(8,10));  // True

    cout << "s1.key_comp()(10,8) returned ";
    truefalse(s1.key_comp()(10,8));  // False

    cout << "s1.value_comp()(10,8) returned ";
    truefalse(s1.value_comp()(10,8));  // False

    cout << "s1.key_comp()(8,8) returned ";
    truefalse(s1.key_comp()(8,8));  // False

    cout << "s1.value_comp()(8,8) returned ";
    truefalse(s1.value_comp()(8,8));   // False
  }
```

Output

```
s1.key_comp()(8,10) returned True
s1.value_comp()(8,10) returned True
s1.key_comp()(10,8) returned False
s1.value_comp()(10,8) returned False
s1.key_comp()(8,8) returned False
s1.value_comp()(8,8) returned False
```

Requirements

Header: <set>

See Also

Concepts

Standard Template Library Samples

set::lower_bound, set::upper_bound, and set::equal_range

Illustrates how to use the set::lower_bound, set::upper_bound, and set::equal_range Standard Template Library (STL) functions in Visual C++.

```
template<class _K, class _Pr, class _A>
   class set
   {
      public:
      // Function 1:
      const_iterator lower_bound(const _K& _Kv) const;
      // Function 2:
      const_iterator upper_bound(const _K& _Kv) const;
      // Function 3:
      _Paircc equal_range(const _K& _Kv) const;
   }
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **lower_bound** function returns an iterator to the earliest element in the controlled sequence that has a key that does not match the value passed to the **lower_bound** function. The **upper_bound** function returns an iterator to the earliest element in the controlled sequence that has a key matching the value passed to the **upper_bound** function. If no such element exists, the function returns **end**. In both cases, the function **set::key_comp**(*key*, *x*) is used to determine if the keys match. The **equal_range** function returns a pair value, where *.First* is the result of the **lower_bound** function, and .second is the result of the **upper_bound** function.

Example

```
// SetBoundRange.cpp
// compile with: /EHsc
//
//      Illustrates how to use the lower_bound function to get an
//      iterator to the earliest element in the controlled sequence
//      that has a key that does not match the value passed to the
//      function. It also illustrates how to use the upper_bound
//      function to get an iterator to the earliest element in the
//      controlled sequence that has a key that matches the value
//      passed to the function. The last thing it illustrates is
//      how to use the equal_range function to get a pair value that
//      contains the lower_bound and upper_bound results of the key.
//
// Functions:
//
//    lower_bound  Returns an iterator whose value does not match the
//                 key passed to the function, or end() if no such
//                 element exists.
//    upper_bound  Returns an iterator whose value matches the key
//                 passed to the function, or end() if no such element
//                 exists.
//    equal_range  Returns a pair of (lower_bound,upper_bound).
///////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <set>
#include <iostream>

using namespace std ;
```

```
typedef set<int> SET_INT;

int main() {
  SET_INT s1;
  SET_INT::iterator i;
  cout << "s1.insert(5)" << endl;
  s1.insert(5);
  cout << "s1.insert(10)" << endl;
  s1.insert(10);
  cout << "s1.insert(15)" << endl;
  s1.insert(15);
  cout << "s1.insert(20)" << endl;
  s1.insert(20);
  cout << "s1.insert(25)" << endl;
  s1.insert(25);

  cout << "s1 -- starting at s1.lower_bound(12)" << endl;
// prints: 15,20,25
  for (i=s1.lower_bound(12);i!=s1.end();i++)
     cout << "s1 has " << *i << " in its set." << endl;

  cout << "s1 -- starting at s1.lower_bound(15)" << endl;
// prints: 15,20,25
  for (i=s1.lower_bound(15);i!=s1.end();i++)
     cout << "s1 has " << *i << " in its set." << endl;

  cout << "s1 -- starting at s1.upper_bound(12)" << endl;
// prints: 15,20,25
  for (i=s1.upper_bound(12);i!=s1.end();i++)
     cout << "s1 has " << *i << " in its set." << endl;

  cout << "s1 -- starting at s1.upper_bound(15)" << endl;
// prints: 20,25
  for (i=s1.upper_bound(15);i!=s1.end();i++)
     cout << "s1 has " << *i << " in its set." << endl;

  cout << "s1 -- s1.equal_range(12)" << endl;
// does not print anything
  for (i=s1.equal_range(12).first;i!=s1.equal_range(12).second;i++)
     cout << "s1 has " << *i << " in its set." << endl;

  cout << "s1 -- s1.equal_range(15)" << endl;
// prints: 15
  for (i=s1.equal_range(15).first;i!=s1.equal_range(15).second;i++)
     cout << "s1 has " << *i << " in its set." << endl;
}
```

Output

```
s1.insert(5)
s1.insert(10)
s1.insert(15)
s1.insert(20)
s1.insert(25)
s1 -- starting at s1.lower_bound(12)
s1 has 15 in its set.
s1 has 20 in its set.
s1 has 25 in its set.
s1 -- starting at s1.lower_bound(15)
s1 has 15 in its set.
s1 has 20 in its set.
s1 has 25 in its set.
s1 -- starting at s1.upper_bound(12)
s1 has 15 in its set.
s1 has 20 in its set.
s1 has 25 in its set.
s1 -- starting at s1.upper_bound(15)
s1 has 20 in its set.
s1 has 25 in its set.
```

```
s1 -- s1.equal_range(12)
s1 -- s1.equal_range(15)
s1 has 15 in its set.
```

Requirements

Header: <set>

See Also

Concepts

Standard Template Library Samples

set::count (STL Samples)

Illustrates how to use the set::count Standard Template Library (STL) function in Visual C++.

```
template<class _K, class _Pr, class _A>
   class set
   {
      public:
      // Function 1:
      size_type count(const _K& _Kv) const;
   }
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **count** function is used to determine the number of elements in the controlled sequence that have a particular key.

Example

```
// SetCount.cpp
// compile with: /EHsc
//
//      Illustrates how to use the count function to determine how
//      many elements have a particular key.
//
// Functions:
//    count        Returns the number of elements in the controlled
//                 sequence that have a particular key.
//////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <set>
#include <iostream>

using namespace std ;

typedef set<int> SET_INT;

void truefalse(int x)
{
  cout << (x?"True":"False") << endl;
}

int main() {
  SET_INT s1;
  SET_INT::_Pairib pib;
  cout << "s1.insert(5)" << endl;
  pib=s1.insert(5);
  cout << "Inserted element: ";
  truefalse(pib.second);  // True

  cout << "s1.insert(5)" << endl;
  pib=s1.insert(5);
  cout << "Inserted element: ";
  truefalse(pib.second);  // True

  cout << "s1.insert(8)" << endl;
  s1.insert(8);

  cout << "s1.insert(12)" << endl;
```

```
    s1.insert(12);

    cout << "s1.count(5) returned ";
    cout << s1.count(5) << endl;    // 1

    cout << "s1.count(9) returned ";
    cout << s1.count(9) << endl;    // 0
  }
```

Output

```
s1.insert(5)
Inserted element: True
s1.insert(5)
Inserted element: False
s1.insert(8)
s1.insert(12)
s1.count(5) returned 1
s1.count(9) returned 0
```

Requirements

Header: <set>

See Also

Concepts

Standard Template Library Samples

set::empty and set::clear

Illustrates how to use the set::empty and set::clear Standard Template Library (STL) functions in Visual C++.

```
template<class _K, class _Pr, class _A>
   class set
   {
      public:
      // Function 1:
      bool empty() const;
      // Function 2:
      void clear();
   }
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **empty** function returns true when there are no elements in the controlled sequence. By calling **erase**(**begin**, **end**), the **clear** function removes all elements in the controlled sequence. For more information on the **erase** function, see set::erase.

Example

```
// SetEmptyClear.cpp
// compile with: /EHsc
//
//      Illustrates how to use the empty function to determine if
//      there are elements in the controlled sequence. It also
//      illustrates how to use the clear function to remove all
//      elements from the controlled sequence.
//
// Functions:
//
//    empty        Returns true if there are no elements in the
//                 controlled sequence.
//    clear        Removes all elements from the controlled sequence.
//////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <set>
#include <iostream>

using namespace std ;

typedef set<int> SET_INT;

void truefalse(int x)
{
  cout << (x?"True":"False") << endl;
}

int main() {
  SET_INT s1;

  cout << "s1.empty() returned ";
  truefalse(s1.empty());  // True

  cout << "s1.insert(5)" << endl;
  s1.insert(5);
  cout << "s1.insert(8)" << endl;
  s1.insert(8);
```

```
    cout << "s1.empty() returned ";
    truefalse(s1.empty());  // False

    cout << "s1.clear()" << endl;
    s1.clear();

    cout << "s1.empty() returned ";
    truefalse(s1.empty());  // True
  }
```

Output

```
s1.empty() returned True
s1.insert(5)
s1.insert(8)
s1.empty() returned False
s1.clear()
s1.empty() returned True
```

Requirements

Header: <set>

See Also

Concepts

Standard Template Library Samples

set::find (STL Samples)

Illustrates how to use the set::find Standard Template Library (STL) function in Visual C++.

```
template<class _K, class _Pr, class _A>
   class set
   {
      public:
      // Function 1:
      const_iterator find(const _K& _Kv) const;
   }
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **find** function is used to locate an element in a controlled sequence. It returns an iterator to the first element in the controlled sequence whose sort key matches its parameter. If no such element exists, the returned iterator equals **end**.

Example

```
// SetFind.cpp
// compile with: /EHsc
//
//      Illustrates how to use the find function to get an iterator
//      that points to the first element in the controlled sequence
//      that has a particular sort key.
//
// Functions:
//
//    find         Returns an iterator that points to the first element
//                 in the controlled sequence that has the same sort key
//                 as the value passed to the find function. If no such
//                 element exists, the iterator equals end().
///////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <set>
#include <iostream>

using namespace std ;

typedef set<int> SET_INT;

void truefalse(int x)
{
  cout << (x?"True":"False") << endl;
}

int main() {
  SET_INT s1;
  cout << "s1.insert(5)" << endl;
  s1.insert(5);
  cout << "s1.insert(8)" << endl;
  s1.insert(8);
  cout << "s1.insert(12)" << endl;
  s1.insert(12);

  SET_INT::iterator it;
  cout << "it=find(8)" << endl;
  it=s1.find(8);
```

```
    cout << "it!=s1.end() returned ";
    truefalse(it!=s1.end());  //  True

    cout << "it=find(6)" << endl;
    it=s1.find(6);
    cout << "it!=s1.end() returned ";
    truefalse(it!=s1.end());  // False
  }
```

Output

```
s1.insert(5)
s1.insert(8)
s1.insert(12)
it=find(8)
it!=s1.end() returned True
it=find(6)
it!=s1.end() returned False
```

Requirements

Header: <set>

See Also

Concepts

Standard Template Library Samples

set::max_size (STL Samples)

Illustrates how to use the set::max_size Standard Template Library (STL) function in Visual C++.

```
template<class _K, class _Pr, class _A>
   class set
   {
      public:
      // Function 1:
      size_type max_size() const;
   }
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **max_size** function is used to determine the maximum number of elements the controlled sequence can contain.

Example

```
// SetMax_size.cpp
// compile with: /EHsc
//
//      Illustrates how to use the max_size function to determine how
//      many elements the controlled sequence can contain.
//
// Functions:
//
//    max_size     Returns the maximum number of elements the controlled
//                 sequence can contain.
//
/////////////////////////////////////////////////////////////////////
#pragma warning(disable:4786)
#include <set>
#include <iostream>
#include <assert.h>

using namespace std;
typedef set<int> SET_INT;

int main()
{
   SET_INT s1;
   cout << "s1.max_size() returned ";
   cout << s1.max_size() << endl;   // 1073741823 [value may vary]

   for (unsigned int x = 0 ; (x < 1000 && x < s1.max_size()) ; x++)

   assert(s1.insert(x).second);
   cout << "s1.size() returned ";
   cout << s1.size() << endl;   // 1000
}
```

Sample Output

The following output is for x86.

```
s1.max_size() returned 1073741823
s1.size() returned 1000
```

Requirements

Header: <set>

See Also

Concepts

Standard Template Library Samples

set::rbegin and set::rend

Illustrates how to use the set::rbegin and set::rend Standard Template Library (STL) functions in Visual C++.

```
template<class _K, class _Pr, class _A>
   class set
   {
      public:
      // Function 1:
      const_reverse_iterator rbegin( ) const;
      // Function 2:
      const_reverse_iterator rend( ) const;
   }
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **rbegin** function returns a reverse bidirectional iterator that points just beyond the end of the controlled sequence. The **rend** function returns a reverse bidirectional iterator that points at the first element of the sequence.

Example

```
// SetRbeginRend.cpp
// compile with: /EHsc
//
//      Illustrates how to use the rbegin function to get a reverse
//      bidirectional iterator that points just beyond the end of
//      the controlled sequence. It also illustrates how to use
//      the rend function to get a reverse bidirectional iterator
//      that points at the first element of the sequence.
//
// Functions:
//
//    rbegin      Returns a reverse bidirectional iterator that points
//                just beyond the end of the controlled sequence.
//    rend        Returns a reverse bidirectional iterator that points
//                at the first element of the sequence.
//
///////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <set>
#include <iostream>

using namespace std ;

typedef set<int> SET_INT;

int main() {
  SET_INT s1;
  SET_INT::reverse_iterator i;
  cout << "s1.insert(5)" << endl;
  s1.insert(5);
  cout << "s1.insert(10)" << endl;
  s1.insert(10);
  cout << "s1.insert(15)" << endl;
  s1.insert(15);
  cout << "s1.insert(20)" << endl;
  s1.insert(20);
```

```
    // displays: 20,15,10,5
    for (i=s1.rbegin();i!=s1.rend();i++)
       cout << "s1 has " << *i << " in its set." << endl;
  }
```

Output

```
s1.insert(5)
s1.insert(10)
s1.insert(15)
s1.insert(20)
s1 has 20 in its set.
s1 has 15 in its set.
s1 has 10 in its set.
s1 has 5 in its set.
```

Requirements

Header: <set>

See Also

Concepts

Standard Template Library Samples

set::size (STL Samples)

Illustrates how to use the set::size Standard Template Library (STL) function in Visual C++.

```
template<class _K, class _Pr, class _A>
   class set
   {
      public:
      // Function 1:
      size_type size() const;
   }
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **size** function is used to determine the number of elements in the controlled sequence.

Example

```
// SetSize.cpp
// compile with: /EHsc
//
//      Illustrates how to use the size function to determine how
//      many elements are in the controlled sequence.
//
// Functions:
//
//    size         Returns the number of elements in the controlled
//                 sequence.
//
///////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <set>
#include <iostream>

using namespace std ;

typedef set<int> SET_INT;

int main() {
  SET_INT s1;

  cout << "s1.size() returned ";
  cout << s1.size() << endl;  // 0

  cout << "s1.insert(5)" << endl;
  s1.insert(5);
  cout << "s1.insert(8)" << endl;
  s1.insert(8);
  cout << "s1.insert(12)" << endl;
  s1.insert(12);

  cout << "s1.size() returned ";
  cout << s1.size() << endl; // 3
}
```

Output

```
s1.size() returned 0
s1.insert(5)
s1.insert(8)
s1.insert(12)
s1.size() returned 3
```

Requirements

Header: <set>

See Also

Concepts

Standard Template Library Samples

set::swap, set::begin, and set::end

Illustrates how to use the set::swap, set::begin, and set::end Standard Template Library (STL) functions in Visual C++.

```
template<class _K, class _Pr, class _A>
   class set
   {
      public:
      // Function 1:
      void swap(_Myt& _X);
      // Function 2:
      friend void swap(_Myt& _X, _Myt& _Y);
      // Function 3:
      const_iterator begin( ) const;
      // Function 4:
      const_iterator end( ) const;
   }
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **swap** function swaps the two controlled sequences. The **begin** function returns a bidirectional iterator that points at the first element of the sequence. The **end** function returns a bidirectional iterator that points just beyond the end of the sequence.

Example

```
// SetSwapBeginEnd.cpp
// compile with: /EHsc
//
//      Illustrates how to use the swap function to exchange the two
//      controlled sequences. It also illustrates how to use the
//      begin function to get a bidirectional iterator that points at
//      the first element of the controlled sequence. Finally, it
//      illustrates how to use the end function to get a bidirectional
//      iterator that points just beyond the end of the controlled
//      sequence.
//
// Functions:
//
//    swap        Exchanges the two controlled sequences.
//    begin       Returns a bidirectional iterator that points at the
//                first element of the controlled sequence.
//    end         Returns a bidirectional iterator that points just
//                beyond the end of the controlled sequence.
//
///////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <set>
#include <iostream>

using namespace std ;

typedef set<int> SET_INT;

int main() {
  SET_INT s1;
  SET_INT s2;
  SET_INT::iterator i;
  cout << "s1.insert(5)" << endl;
```

```
    s1.insert(5);
    cout << "s1.insert(10)" << endl;
    s1.insert(10);
    cout << "s1.insert(15)" << endl;
    s1.insert(15);
    cout << "s2.insert(2)" << endl;
    s2.insert(2);
    cout << "s2.insert(4)" << endl;
    s2.insert(4);

    cout << "swap(s1,s2)" << endl;
    swap(s1,s2);

    // Displays: 2,4
    for (i=s1.begin();i!=s1.end();i++)
       cout << "s1 has " << *i << " in its set." << endl;

    // Displays: 5,10,15
    for (i=s2.begin();i!=s2.end();i++)
       cout << "s2 has " << *i << " in its set." << endl;

    cout << "s1.swap(s2)" << endl;
    s1.swap(s2);

    // Displays: 5,10,15
    for (i=s1.begin();i!=s1.end();i++)
       cout << "s1 has " << *i << " in its set." << endl;

    // Displays: 2,4
    for (i=s2.begin();i!=s2.end();i++)
       cout << "s2 has " << *i << " in its set." << endl;
  }
```

Output

```
s1.insert(5)
s1.insert(10)
s1.insert(15)
s2.insert(2)
s2.insert(4)
swap(s1,s2)
s1 has 2 in its set.
s1 has 4 in its set.
s2 has 5 in its set.
s2 has 10 in its set.
s2 has 15 in its set.
s1.swap(s2)
s1 has 5 in its set.
s1 has 10 in its set.
s1 has 15 in its set.
s2 has 2 in its set.
s2 has 4 in its set.
```

Requirements

Header: <set>

See Also

Concepts

Standard Template Library Samples

sqrt and pow

Illustrates how to use the sqrt and pow Standard Template Library (STL) functions in Visual C++.

```
template<class T>
inline valarray<T> sqrt(
   const valarray<T>& x
);
template<class T>
inline valarray<T> pow(
   const valarray<T>& x,
   const valarray<T>& y
);
template<class T>
inline valarray<T> pow(
   const valarray<T> x,
   const T& y
);
template<class T>
inline valarray<T> pow(
   const T& x,
   const valarray<T>& y
);
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

sqrt returns an object of class **valarray**<**T**>, each of whose elements at index *I* is the square root of *x*[*I*]. **pow** has three template functions. The first template function returns an object of class **valarray**<**T**>, each of whose elements at index *I* is *x*[*I*] raised to the power of *y*[*I*]. The second template function stores in element *I* *x*[*I*] raised to the power of *y*. The third template function stores in element *I* *x* raised to the power of *y*[*I*].

Example

```
// sqrtpow.cpp
// compile with: /EHsc

#include <iostream>                 // for i/o functions
#include <valarray>                 // for valarray
#include <cmath>                    // for sqrt() and pow()

using namespace std ;

#define ARRAY_SIZE  3               // array size

int main()
{
   // Set val_array to contain values 1, 4, 9 for the following test
   valarray<double> val_array(ARRAY_SIZE);
   int i;
   for (i = 0; i < ARRAY_SIZE; i++)
      val_array[i] = (i+1) * (i+1);

   // Display the size of val_array
   cout << "Size of val_array = " << val_array.size() << endl;

   // Display the values of val_array before calling sqrt() and pow().
   cout << "The values in val_array:" << endl << "[";
   for (i = 0; i < ARRAY_SIZE; i++)
```

```
      cout << " " << val_array[i];
   cout << " ]" << endl << endl;

   // Initialize rev_valarray that is the reverse of val_array.
   valarray<double> rev_valarray(ARRAY_SIZE);
   for (i = 0; i < ARRAY_SIZE; i++)
      rev_valarray[i] = val_array[ARRAY_SIZE - i - 1];

   // Display the size of rev_valarray.
   cout << "Size of rev_valarray = " << rev_valarray.size() << endl;

   // Display the values of rev_valarray.
   cout << "The values in rev_valarray:" << endl << "[";
   for (i = 0; i < ARRAY_SIZE; i++)
      cout << " " << rev_valarray[i];
   cout << " ]" << endl << endl;

   // rvalue_array to hold the return value from calling the sqrt() and
   // pow() functions.
   valarray<double> rvalue_array;

   // ----------------------------------------------------------------
   // sqrt() - display the content of rvalue_array
   // ----------------------------------------------------------------

   // Display the result of val_array after calling sqrt().
   rvalue_array = sqrt(val_array);
   cout << "The result of val_array after calling sqrt():"
        << endl << "[";
   for (i = 0; i < ARRAY_SIZE; i++)
      cout << " " << rvalue_array[i];
   cout << " ]" << endl << endl;

   // ----------------------------------------------------------------
   // pow() - display the content of rvalue_array
   // ----------------------------------------------------------------

   // This template function returns an object of class valarray<T>,
   // each of whose elements at I is x[I] raised to the power of y[I].
   rvalue_array = pow(val_array, rev_valarray);
   cout << "The result after calling pow(val_array, rev_valarray):"
        << endl << "[";
   for (i = 0; i < ARRAY_SIZE; i++)
      cout << " " << rvalue_array[i];
   cout << " ]" << endl << endl;

   // This template function stores in element I x[I] raised to the
   // power of y, where y=2.0.
   rvalue_array = pow(val_array, 2.0);
   cout << "The result after calling pow(val_array, 2.0):"
        << endl << "[";
   for (i = 0; i < ARRAY_SIZE; i++)
      cout << " " << rvalue_array[i];
   cout << " ]" << endl << endl;

   // This template function stores in element I x raised to the
   // y[I] power, where x=2.0.
   rvalue_array = pow(2.0, val_array);
   cout << "The result after calling pow(2.0, val_array):"
        << endl << "[";
   for (i = 0; i < ARRAY_SIZE; i++)
      cout << " " << rvalue_array[i];
   cout << " ]" << endl;
}
```

Output

```
Size of val_array = 3
```

```
The values in val_array:
[ 1 4 9 ]

Size of rev_valarray = 3
The values in rev_valarray:
[ 9 4 1 ]

The result of val_array after calling sqrt():
[ 1 2 3 ]

The result after calling pow(val_array, rev_valarray):
[ 1 256 9 ]

The result after calling pow(val_array, 2.0):
[ 1 16 81 ]

The result after calling pow(2.0, val_array):
[ 2 16 512 ]
```

Requirements

Header: <valarray>

See Also

Concepts

Standard Template Library Samples

stack::operator<

Illustrates how to use the stack::operator< Standard Template Library (STL) function in Visual C++.

```
template<class _TYPE, class _C, class _A>
   bool stack::operator<(
      const stack<_TYPE, _C, _A>& _X
   ) const;
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **stack::operator<** function returns true if the stack on the left side of the operator is less than the stack on the right side.

To determine if one stack is less than another stack

1. Compare the bottom-most element (very first element pushed onto the stack).
2. If the elements are different, the stack with the smaller element is less than the stack with the greater element.
3. If the elements are the same and there are more elements, move to the next element in the stack and go back to step 2.
4. If all the elements in the stacks are processed at this point, the stacks are equal.

Example

```
// StackLessThan.cpp
// compile with: /EHsc
// Illustrates how to use the stack::operator<
// function to determine if one stack is less than
// another stack.
//
// Functions:
//
//    operator< :  Returns true if the stack is smaller than the stack
//                 passed as the operand.
//////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <stack>
#include <iostream>

using namespace std ;

typedef stack<double> STACK_DOUBLE;

int main()
{
   STACK_DOUBLE stack1,stack2;

   // Add item 4.0 to Stack1. Stack1 contains 4.0.
   cout << "stack1.push(4.0)  s1=[4.0]" << endl;
   stack1.push(4.0);

   // Add item 3.0 to Stack1. Stack1 contains 3.0(top) and 4.0(bottom).
   cout << "stack1.push(3.0)  s1=[3.0 4.0]" << endl;
   stack1.push(3.0);

   // Add item 4.0 to Stack2. Stack2 contains 4.0 (top=bottom).
   cout << "stack2.push(4.0)  s2=[4.0]" << endl;
   stack2.push(4.0);
```

```
    // Compare if Stack1 is smaller than Stack2. Should return False.
    cout << "stack1<stack2 is " <<
       ((stack1<stack2)? "True": "False") << endl << endl;

    // Add item 6.0 to Stack2. Stack2 contains 6.0(top) and 4.0(bottom).
    cout << "stack2.push(6.0)  s2=[6.0 4.0]" << endl;
    stack2.push(6.0);

    // Compare if Stack1 is smaller than Stack2. Should return True.
    cout << "stack1<stack2 is " <<
       ((stack1<stack2)? "True": "False") << endl << endl;

    // Add item 8.0 to Stack2. Stack2 contains 8.0(top), 6.0 and
    // 4.0(bottom).
    cout << "stack2.push(8.0)  s2=[8.0 6.0 4.0]" << endl;
    stack2.push(8.0);

    // Compare if Stack1 is smaller than Stack2. Should return True.
    cout << "stack1<stack2 is " <<
       ((stack1<stack2)? "True": "False") << endl << endl;

    // Delete item 8.0 from Stack2.
    cout << "stack2.pop()      s2=[6.0 4.0]" << endl;
    stack2.pop();

    // Delete item 6.0 from Stack2.
    cout << "stack2.pop()      s2=[4.0]" << endl;
    stack2.pop();

    // Add item 3.0 to Stack2. Stack2 contains 3.0(top) and 4.0(bottom).
    cout << "stack2.push(3.0)  s2=[3.0 4.0]" << endl;
    stack2.push(3.0);

    // Compare if Stack1 is smaller than Stack2. Should return False.
    cout << "stack1<stack2 is " <<
       ((stack1<stack2)? "True": "False") << endl << endl;

    // Delete item 3.0 from Stack2.
    cout << "stack2.pop()      s2=[4.0]" << endl;
    stack2.pop();

    // Delete item 4.0 from Stack2.
    cout << "stack2.pop()      s2=[]" << endl;
    stack2.pop();

    // Add item 8.0 to Stack2. Stack2 contains 8.0(top=bottom).
    cout << "stack2.push(8.0)  s2=[8.0]" << endl;
    stack2.push(8.0);

    // Compare if Stack1 is smaller than Stack2. Should return True.
    cout << "stack1<stack2 is " <<
       ((stack1<stack2)? "True": "False") << endl << endl;
  }
```

Output

```
stack1.push(4.0)  s1=[4.0]
stack1.push(3.0)  s1=[3.0 4.0]
stack2.push(4.0)  s2=[4.0]
stack1<stack2 is False

stack2.push(6.0)  s2=[6.0 4.0]
stack1<stack2 is True

stack2.push(8.0)  s2=[8.0 6.0 4.0]
stack1<stack2 is True

stack2.pop()      s2=[6.0 4.0]
```

```
stack2.pop()       s2=[4.0]
stack2.push(3.0)   s2=[3.0 4.0]
stack1<stack2 is False

stack2.pop()       s2=[4.0]
stack2.pop()       s2=[]
stack2.push(8.0)   s2=[8.0]
stack1<stack2 is True
```

Requirements

Header: <stack>

See Also

Concepts

Standard Template Library Samples

stack::operator==

Illustrates how to use the stack::operator== Standard Template Library (STL) function in Visual C++.

```
template<class _TYPE, class _C, class _A>
   bool stack::bool operator==(
      const stack<_TYPE, _C, _A>& _X
   ) const;
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **stack::operator==** function returns **true** if both stacks have equal elements arranged in the same sequence. The **stack::operator==** function always returns **false** if the two stacks are of a different size.

Example

```
// StackEqual.cpp
// compile with: /EHsc
// Illustrates how to use the stack::operator==
// function to determine if two stacks are equal.
//
// Functions:
//    operator== :  Returns true if both stacks are the same.
//////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <stack>
#include <iostream>

using namespace std ;

typedef stack<double> STACK_DOUBLE;

int main()
{
   STACK_DOUBLE stack1,stack2;

   // Add item 4.0 to Stack1.
   cout << "stack1.push(4.0)  s1=[4.0]" << endl;
   stack1.push(4.0);

   // Add item 3.0 to Stack1. Current Stack1 contains items 3.0 (top)
   // 4.0 (bottom).
   cout << "stack1.push(3.0)  s1=[3.0 4.0]" << endl;
   stack1.push(3.0);

   // Add item 4.0 to Stack2.
   cout << "stack2.push(4.0)  s2=[4.0]" << endl;
   stack2.push(4.0);

   // Compare Stack1 and Stack2. Should return False.
   cout << "stack1==stack2 is " <<
      ((stack1==stack2)? "True": "False") << endl << endl;

   // Add item 6.0 to Stack2. Current Stack2 contains items 6.0 (top)
   // 4.0 (bottom)
   cout << "stack2.push(6.0)  s2=[6.0 4.0]" << endl;
   stack2.push(6.0);
```

```
    // Compare Stack1 and Stack2. Should return False.
    cout << "stack1==stack2 is " <<
       ((stack1==stack2)? "True": "False") << endl << endl;

    // Keep adding item 8.0 to Stack2. Current Stack2 contains items
    // 8.0 (top), 6.0 and 4.0 (bottom).
    cout << "stack2.push(8.0)  s2=[8.0 6.0 4.0]" << endl;
    stack2.push(8.0);

    // Compare Stack1 and Stack2. Should return False.
    cout << "stack1==stack2 is " <<
       ((stack1==stack2)? "True": "False") << endl << endl;

    // Delete the top item from Stack2. Current Stack2 contains items
    // 6.0 (top) and 4.0 (bottom).
    cout << "stack2.pop()      s2=[6.0 4.0]" << endl;
    stack2.pop();

    // Delete another item from Stack2. Current Stack2 contains item 4.0.
    cout << "stack2.pop()      s2=[4.0]" << endl;
    stack2.pop();

    // Add item 3.0 to Stack2. Current Stack2 contains item 3.0 (top) and
    // 4.0 (bottom).
    cout << "stack2.push(3.0)  s2=[3.0 4.0]" << endl;
    stack2.push(3.0);

    // Compare Stack2 and Stack2. Should return True.
    cout << "stack1==stack2 is " <<
       ((stack1==stack2)? "True": "False") << endl << endl;

    // Delete the top item from Stack2. Current Stack2 contains 4.0.
    cout << "stack2.pop()      s2=[4.0]" << endl;
    stack2.pop();

    // Delete another item from Stack2. Stack2 should be empty.
    cout << "stack2.pop()      s2=[]" << endl;
    stack2.pop();

    // Push item 8.0 to Stack2.
    cout << "stack2.push(8.0)  s2=[8.0]" << endl;
    stack2.push(8.0);

    // Compare Stack1 and Stack2. Should return False.
    cout << "stack1==stack2 is " <<
       ((stack1==stack2)? "True": "False") << endl << endl;
  }
```

Output

```
stack1.push(4.0)  s1=[4.0]
stack1.push(3.0)  s1=[3.0 4.0]
stack2.push(4.0)  s2=[4.0]
stack1==stack2 is False

stack2.push(6.0)  s2=[6.0 4.0]
stack1==stack2 is False

stack2.push(8.0)  s2=[8.0 6.0 4.0]
stack1==stack2 is False

stack2.pop()      s2=[6.0 4.0]
stack2.pop()      s2=[4.0]
stack2.push(3.0)  s2=[3.0 4.0]
stack1==stack2 is True

stack2.pop()      s2=[4.0]
stack2.pop()      s2=[]
stack2.push(8.0)  s2=[8.0]
```

```
stack1==stack2 is False
```

Requirements

Header: <stack>

See Also

Concepts

Standard Template Library Samples

stack::size (STL Samples)

Illustrates how to use the stack::size Standard Template Library (STL) function in Visual C++.

```
template<class _TYPE, class _C, class _A>
   size_type stack::size( ) const;
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **stack::size** function returns the number of elements in the stack. It is okay to call this function with an empty stack; it will return a value of 0.

Example

```
// StackSize.cpp
// compile with: /EHsc
// Illustrates how to use the size function to determine
// the number of elements on the stack.
//
// Functions:
//
//    size :  returns the number of elements in the stack.
//////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <stack>
#include <string>
#include <iostream>

using namespace std ;

typedef stack<string> STACK_STRING;

int main()
{
   STACK_STRING stack1;

   // Check the size of an empty stack. Should return 0.
   cout << "stack1.size() equals " << stack1.size() << endl;

   // Add item "Hello" to Stack1.
   cout << "stack1.push('Hello')" << endl;
   stack1.push("Hello");

   // Add item "This is the second element" to Stack1.
   cout << "stack1.push('This is the second element')" << endl;
   stack1.push("This is the second element");

   // Check the size of Stack1. Should return 2.
   cout << "stack1.size() equals " << stack1.size() << endl << endl;

   // Add item "Third element" to Stack1.
   cout << "stack1.push('Third element')" << endl;
   stack1.push("Third element");

   // Check the size of Stack1. Should return 3.
   cout << "stack1.size() equals " << stack1.size() << endl << endl;

   // Pop "Third element".
```

```
    cout << "stack1.pop()" << endl;
    stack1.pop();

    // Pop "This is the second element".
    cout << "stack1.pop()" << endl;
    stack1.pop();

    // Check the size of Stack1 again. Should return 1.
    cout << "stack1.size() equals " << stack1.size()  << endl << endl;

    // Pop "Hello".
    cout << "stack1.pop()" << endl;
    stack1.pop();

    // Check the size of Stack1. Should return 0.
    cout << "stack1.size() equals " << stack1.size() << endl;
}
```

Output

```
stack1.size() equals 0
stack1.push('Hello')
stack1.push('This is the second element')
stack1.size() equals 2

stack1.push('Third element')
stack1.size() equals 3

stack1.pop()
stack1.pop()
stack1.size() equals 1

stack1.pop()
stack1.size() equals 0
```

Requirements

Header: <stack>

See Also

Concepts

Standard Template Library Samples

stack::top and stack::empty

Illustrates how to use the stack::top and stack::empty STL functions in Visual C++.

```
template<class _TYPE, class _C, class _A>
   value_type& stack::top( );
template<class _TYPE, class _C, class _A>
   const value_type& stack::top( ) const;
template<class _TYPE, class _C, class _A>
   bool stack::empty( ) const;
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **top** function returns the topmost element of the stack. You should ensure that there are one or more elements on the stack before calling the top function. The first version of the top function returns a reference to the element of the top of the stack, allowing you to modify the value. The second function returns a constant reference, ensuring that you do not accidentally modify the stack. The empty function returns **true** if there are no elements in the stack. If there are one or more elements, the function will return **false**. You should use the empty function to verify that there are elements left on the stack before calling the top function.

Example

```
// StackTopEmpty.cpp
// compile with: /EHsc
// Illustrates how to use the top function to
// retrieve the last element of the controlled
// sequence. It also illustrates how to use the
// empty function to loop though the stack.
//
// Functions:
//
//    top   :  returns the top element of the stack.
//    empty :  returns true if the stack has 0 elements.
/////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <stack>
#include <iostream>

using namespace std ;

typedef stack<int> STACK_INT;

int main()
{
   STACK_INT stack1;

   cout << "stack1.empty() returned " <<
      (stack1.empty()? "true": "false") << endl;  // Function 3

   cout << "stack1.push(2)" << endl;
   stack1.push(2);

   if (!stack1.empty())                           // Function 3
      cout << "stack1.top() returned " <<
      stack1.top() << endl;                       // Function 1

   cout << "stack1.push(5)" << endl;
```

```
    stack1.push(5);

    if (!stack1.empty())                          // Function 3
       cout << "stack1.top() returned " <<
       stack1.top() << endl;                      // Function 1

    cout << "stack1.push(11)" << endl;
    stack1.push(11);

    if (!stack1.empty())                          // Function 3
       cout << "stack1.top() returned " <<
       stack1.top() << endl;                      // Function 1

    // Modify the top item. Set it to 6.
    if (!stack1.empty()) {                        // Function 3
       cout << "stack1.top()=6;" << endl;
       stack1.top()=6;                            // Function 1
    }

    // Repeat until stack is empty
    while (!stack1.empty()) {                     // Function 3
       const int& t=stack1.top();                 // Function 2
       cout << "stack1.top() returned " << t << endl;
       cout << "stack1.pop()" << endl;
       stack1.pop();
    }
 }
```

Output

```
stack1.empty() returned true
stack1.push(2)
stack1.top() returned 2
stack1.push(5)
stack1.top() returned 5
stack1.push(11)
stack1.top() returned 11
stack1.top()=6;
stack1.top() returned 6
stack1.pop()
stack1.top() returned 5
stack1.pop()
stack1.top() returned 2
stack1.pop()
```

Requirements

Header: <stack>

See Also

Concepts

Standard Template Library Samples

string::getline

Illustrates how to use the string::getline Standard Template Library (STL) class in Visual C++.

```
template<class _E, class _TYPE, class _A> inline
   basic_istream<_E, _TYPE>& getline(
   basic_istream<_E, _TYPE>& Istream,
   basic_string<_E, _TYPE, _A>& Xstring,
   const _E _D=_TYPE::newline( )
   );
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **getline** function creates a string containing all of the characters from the input stream until one of the following situations occurs: - End of file. - The delimiter is encountered. - **is.max_str** elements have been extracted.

Example

```
// string_getline_sample.cpp
// compile with: /EHsc
// Illustrates how to use the getline function to read a
// line of text from the keyboard.
//
// Functions:
//
//    getline       Returns a string from the input stream.
//////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <string>
#include <iostream>

using namespace std ;

int main()
{
   string s1;
   cout << "Enter a sentence (use <space> as the delimiter): ";
   getline(cin,s1, ' ');
   cout << "You entered: " << s1 << endl;;
}
```

Input

```
test this
```

Sample Output

```
Enter a sentence (use <space> as the delimiter): test this
You entered: test
```

Requirements

Header: <string>

See Also

Concepts

Standard Template Library Samples

string::operator!=

Illustrates how to use the string::operator!= Standard Template Library (STL) function in Visual C++.

```
template<class _E, class _TYPE, class _A> inline
   bool operator!=(
      const basic_string<_E, _TYPE, _A>& LString,
      const _E *RCharArray
   );
template<class _E, class _TYPE, class _A> inline
   bool operator!=(
      const _E *LCharArray,
      const basic_string<_E, _TYPE, _A>& RString
   );
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

There are two versions of the **string::operator!=** function. Both versions are used to compare a null-terminated character array against a basic_string. They perform this operation by negating the result of (**LString==RCharArray**) or (**LCharArray==RString**). For more information on this comparison, see the **string::operator==** function. Notice that this operator does not work with NULL pointer for the character array. You will need to make sure that the character array is not NULL before you pass it to the operator.

Example

```
// StringNotEqual.cpp
// compile with: /EHsc
// Illustrates how to use the operator!= to test for
// non-equality of a basic_string variable and a
// null-terminated string.
//
// Functions:
//
//    operator!=   Returns true if the basic_string and the null-
//                 terminated string are not equal.
/////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <string>
#include <iostream>

using namespace std ;

void trueFalse(int x)
{
  cout << (x? "True": "False") << endl;
}

int main()
{
   string S1="ABC";
   char CP1[]="ABC";
   char CP2[]="DEF";
   char CP3[]="abc";

   cout << "S1 is " << S1 << endl;
   cout << "CP1 is " << CP1 << endl;
   cout << "CP2 is " << CP2 << endl;
```

```
    cout << "CP3 is " << CP3 << endl;

    cout << "S1!=CP1 returned ";
    trueFalse(S1!=CP1);  // False  (calls function 1)

    cout << "S1!=CP2 returned ";
    trueFalse(S1!=CP2);  // True   (calls function 1)

    cout << "S1!=CP3 returned ";
    trueFalse(S1!=CP3);  // True   (calls function 1)

    cout << "CP1!=S1 returned ";
    trueFalse(CP1!=S1);  // False  (calls function 2)

    cout << "CP2!=S1 returned ";
    trueFalse(CP2!=S1);  // True   (calls function 2)

    cout << "CP3!=S1 returned ";
    trueFalse(CP3!=S1);  // True   (calls function 2)
  }
```

Output

```
S1 is ABC
CP1 is ABC
CP2 is DEF
CP3 is abc
S1!=CP1 returned False
S1!=CP2 returned True
S1!=CP3 returned True
CP1!=S1 returned False
CP2!=S1 returned True
CP3!=S1 returned True
```

Requirements

Header: <string>

See Also

Concepts

Standard Template Library Samples

string::operator+

Illustrates how to use the string::operator+ STL function in Visual C++.

```
template<class _E, class _TYPE, class _A> inline
   basic_string<_E, _TYPE, _A>
   operator+(
      const basic_string<_E, _TYPE, _A>& LString,
      const _E *RCharArray
   );
template<class _E, class _TYPE, class _A> inline
   basic_string<_E, _TYPE, _A>
   operator+(
      const _E *LCharArray,
      const basic_string<_E, _TYPE, _A>& RString
   );
template<class _E, class _TYPE, class _A> inline
   basic_string<_E, _TYPE, _A>
   operator+(
      const basic_string<_E, _TYPE, _A>& LString,
      const _E RChar
   );
template<class _E, class _TYPE, class _A> inline
   basic_string<_E, _TYPE, _A>
   operator+(
      const _E LChar,
      const basic_string<_E, _TYPE, _A>& RString
   );
template<class _E, class _TYPE, class _A> inline
   basic_string<_E, _TYPE, _A>
   operator+(
      const basic_string<_E, _TYPE, _A>& LString,
      const basic_string<_E, _TYPE, _A>& RString
   );
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

There are five versions of the **string::operator+** function. Two functions are used to concatenate a null-terminated character array and a basic_string. Two functions are used to concatenate a character and a basic_string. The last function is used to concatenate two basic_string variables.

Example

```
// StringPlus.cpp
// compile with: /EHsc
// Illustrates how to use the operator+ to concatenate
// a null-terminated character array and a basic_string,
// how to concatenate a character and a basic_string,
// and how to concatenate two basic_string variables.
//
// Functions:
//
//    operator+ : Concatenates a null-terminated character array and
//                a basic_string.
//    operator+ : Concatenates a character array and a basic_string.
//    operator+ : Concatenates two basic_string variables.
/////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
```

```
#include <string>
#include <iostream>

using namespace std ;

int main()
{
   string result;
   string S1="ABC";
   string S2="DEF";
   char CP1[]="GHI";
   char C='J';

   cout << "S1 is " << S1 << endl;
   cout << "S2 is " << S2 << endl;
   cout << "CP1 is " << CP1 << endl;
   cout << "C is " << C << endl;

   result=S1+CP1;                          // Function 1  (ABCGHI)
   cout << "S1+CP1 is " << result << endl;

   result=CP1+S1;                          // Function 2  (GHIABC)
   cout << "CP1+S1 is " << result << endl;

   result=S1+S2;                           // Function 3  (ABCDEF)
   cout << "S1+S2 is " << result << endl;

   result=S1+C;                            // Function 4  (ABCJ)
   cout << "S1+C is " << result << endl;

   result=C+S1;                            // Function 5  (JABC)
   cout << "C+S1 is " << result << endl;
}
```

Output

```
S1 is ABC
S2 is DEF
CP1 is GHI
C is J
S1+CP1 is ABCGHI
CP1+S1 is GHIABC
S1+S2 is ABCDEF
S1+C is ABCJ
C+S1 is JABC
```

Requirements

Header: <string>

See Also

Concepts

Standard Template Library Samples

string::operator<

Illustrates how to use the string::operator< Standard Template Library (STL) function in Visual C++.

```
template<class _E, class _TYPE, class _A> inline
   bool operator<(const basic_string<_E, _TYPE, _A>& LString,
   const _E *RCharArray);
template<class _E, class _TYPE, class _A> inline
   bool operator<(const _E *LCharArray,
   const basic_string<_E, _TYPE, _A>& RString);
template<class _E, class _TYPE, class _A> inline
   bool operator<(const basic_string<_E, _TYPE, _A>& LString,
   const basic_string<_E, _TYPE, _A>& RString);
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

There are three versions of the **string::operator<** function. Two of the functions compare a null-terminated character array against a basic_string. They perform this operation by using the basic_string's **string::compare** function and returning a value based on the results of the compare function. The third function compares two basic_string variables, using the **string::compare** function of the first basic_string variable. For more information on the **compare** function, see the **string::compare** function. Note that this operator does not work with NULL pointer for the character array. You will need to make sure that the character array is not NULL before you pass it to the operator.

Example

```
// StringLessThan.cpp
// compile with: /EHsc
// Illustrates how to use the operator< to compare
// a basic_string variable and a null-terminated
// string. It also illustrates how to use the
// operator< to compare two basic_string variables.
//
// Functions:
//
//    operator<    Returns true if the first parameter is less than the
//                 second.
/////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <string>
#include <iostream>

using namespace std ;

void trueFalse(int x)
{
  cout << (x? "True": "False") << endl;
}

int main()
{
   string S1="ABC";
   string S2="ABC";
   string S3="DEF";
   string S4="abc";
   char CP1[]="ABC";
   char CP2[]="DEF";
   char CP3[]="ABCD";
```

```
    cout << "S1 is " << S1 << endl;
    cout << "S2 is " << S2 << endl;
    cout << "S3 is " << S3 << endl;
    cout << "S4 is " << S4 << endl;
    cout << "CP1 is " << CP1 << endl;
    cout << "CP2 is " << CP2 << endl;
    cout << "CP3 is " << CP3 << endl;

    cout << "S1<CP1 returned ";
    trueFalse(S1<CP1);  // False    (calls function 1)

    cout << "S1<CP2 returned ";
    trueFalse(S1<CP2);  // True   (calls function 1)

    cout << "CP1<S1 returned ";
    trueFalse(CP1<S1);  // False    (calls function 2)

    cout << "CP2<S1 returned ";
    trueFalse(CP2<S1);  // False    (calls function 2)

    cout << "S1<S2 returned ";
    trueFalse(S1<S2);   // False    (calls function 3)

    cout << "S1<S3 returned ";
    trueFalse(S1<S3);   // True   (calls function 3)

    cout << "S1<S4 returned ";
    trueFalse(S1<S4);   // True   (calls function 3)

    cout << "S1<CP3 returned ";
    trueFalse(S1<CP3);  // True   (calls function 1)
  }
```

Output

```
S1 is ABC
S2 is ABC
S3 is DEF
S4 is abc
CP1 is ABC
CP2 is DEF
CP3 is ABCD
S1<CP1 returned False
S1<CP2 returned True
CP1<S1 returned False
CP2<S1 returned False
S1<S2 returned False
S1<S3 returned True
S1<S4 returned True
S1<CP3 returned True
```

Requirements

Header: <string>

See Also

Concepts

Standard Template Library Samples

string::operator<<

Illustrates how to use the string::operator<< Standard Template Library (STL) function in Visual C++.

```
template<class _E, class _TYPE, class _A> inline
basic_ostream<_E, _TYPE>&
operator<<( basic_ostream<_E, _TYPE>& OStream,
            const basic_string<_E, _TYPE, _A>& XString);
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **operator<<** is used to insert a string into an output stream.

Example

```
// StringInsertion.cpp
// compile with: /EHsc
// Illustrates how to use the insertion operator
// (operator<<) to insert a string into an output
// stream.
//
// Functions:
//
//    operator<<   Inserts a string into an output stream.
///////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <string>
#include <iostream>

using namespace std ;

int main()
{
    string msg="Hello!  This is the insertion operator.";
    cout << msg << endl;
}
```

Output

```
Hello!  This is the insertion operator.
```

Requirements

Header: <string>

See Also

Concepts

Standard Template Library Samples

string::operator<=

Illustrates how to use the string::operator<= Standard Template Library (STL) function in Visual C++.

```
template<class _E, class _TYPE, class _A> inline
   bool operator<=(const basic_string<_E, _TYPE, _A>& LString,
   const _E *RCharArray);
template<class _E, class _TYPE, class _A> inline
   bool operator<=(const _E * LCharArray,
   const basic_string<_E, _TYPE, _A>& RString);
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

There are two versions of the **string::operator<=** function. Both versions are used to compare a null-terminated character array to a basic_string. They perform this operation by negating the result of (**RString**<**LCharArray or RCharArray**<**LString**). For more information on this comparison, see the **string::operator<** function. Note that this operator does not work with NULL pointer for the character array. You will need to make sure that the character array is not NULL before you pass it to the operator.

Remarks

```
// StringLessEqual.cpp
// compile with: /EHsc
// Illustrates how to use the operator<= to compare
// a basic_string variable to a null-terminated
// string.
//
// Functions:
//
//    operator<=   Returns true if the second parameter is not less
//                 than the first.
//////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <string>
#include <iostream>

using namespace std ;

void trueFalse(int x)
{
  cout << (x? "True": "False") << endl;
}

int main()
{
   string S1="DEF";
   char CP1[]="ABC";
   char CP2[]="DEF";
   char CP3[]="DEFG";
   char CP4[]="def";

   cout << "S1 is " << S1 << endl;
   cout << "CP1 is " << CP1 << endl;
   cout << "CP2 is " << CP2 << endl;
   cout << "CP3 is " << CP3 << endl;
   cout << "CP4 is " << CP4 << endl;
```

```
    cout << "S1<=CP1 returned ";
    trueFalse(S1<=CP1);  // False  (calls function 1)

    cout << "S1<=CP2 returned ";
    trueFalse(S1<=CP2);  // True   (calls function 1)

    cout << "S1<=CP3 returned ";
    trueFalse(S1<=CP3);  // True   (calls function 1)

    cout << "CP1<=S1 returned ";
    trueFalse(CP1<=S1);  // True   (calls function 2)

    cout << "CP2<=S1 returned ";
    trueFalse(CP2<=S1);  // True   (calls function 2)

    cout << "CP4<=S1 returned ";
    trueFalse(CP4<=S1);  // False   (calls function 2)
  }
```

Output

```
S1 is DEF
CP1 is ABC
CP2 is DEF
CP3 is DEFG
CP4 is def
S1<=CP1 returned False
S1<=CP2 returned True
S1<=CP3 returned True
CP1<=S1 returned True
CP2<=S1 returned True
CP4<=S1 returned False
```

Requirements

Header: <string>

See Also

Concepts

Standard Template Library Samples

string::operator==

Illustrates how to use the string::operator== STL function in Visual C++.

```
template<class _E, class _TYPE, class _A> inline
   bool operator==(
      const basic_string<_E, _TYPE, _A>& LString,
      const _E *RCharArray
   );
template<class _E, class _TYPE, class _A> inline
   bool operator==(
   const _E * LCharArray, const basic_string<_E, _TYPE, _A>& RString
   );
template<class _E, class _TYPE, class _A> inline
   bool operator==(
   const basic_string<_E, _TYPE, _A>& LString,
   const basic_string<_E, _TYPE, _A>& RString
   );
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

There are three versions of the **string::operator==** function. The first two functions compare a null-terminated character array against a basic_string. They perform this operation by using the basic_string's **string::compare** function and returning **true** if the compare function returns a 0 value. The third function compares two basic_string variables, using the **string::compare** function of the first basic_string variable. For more information on the **compare** function, see the **string::compare** function. Note that this operator does not work with NULL pointer for the character array. You will need to make sure that the character array is not NULL before you pass it to the operator.

Example

```
// StringEqual.cpp
// compile with: /EHsc
// Illustrates how to use the operator== to test for
// equality of a basic_string variable and a
// null-terminated string. It also illustrates how to
// use the operator== to test for equality of two
// basic_string variables.
//
// Functions:
//
//    operator==   returns true if the basic_string and the null-
//                 terminated string are equal.
//    operator==   returns true if both basic_strings are equal.
//////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <string>
#include <iostream>

using namespace std ;

void trueFalse(int x)
{
  cout << (x? "True": "False") << endl;
}

int main()
{
```

```
    string S1="ABC";
    string S2="ABC";
    string S3="DEF";
    string S4; //This specifies an empty initial-controlled sequence.
    char CP1[]="abc";
    char CP2[]="DEF";
    char *CP3 = NULL;

    cout << "S1 is " << S1 << endl;
    cout << "S2 is " << S2 << endl;
    cout << "S3 is " << S3 << endl;
    cout << "S4 is" << S4 << endl;
    cout << "CP1 is " << CP1 << endl;
    cout << "CP2 is " << CP2 << endl;

    cout << "S1==CP1 returned ";
    trueFalse(S1==CP1);  // False    (calls function 1)

    cout << "S1==CP2 returned ";
    trueFalse(S1==CP2);  // False   (calls function 1)

    cout << "CP1==S1 returned ";
    trueFalse(CP1==S1);  // False    (calls function 2)

    cout << "CP2==S1 returned ";
    trueFalse(CP2==S1);  // False   (calls function 2)

    cout << "S1==S2 returned ";
    trueFalse(S1==S2);   // True    (calls function 3)

    cout << "S1==S3 returned ";
    trueFalse(S1==S3);   // False   (calls function 3)

    cout << "S1==S4 returned ";
    trueFalse(S1==S4);   // False   (calls function 3)

    // Following use of the operator will cause the program to
    // crash since CP3 is NULL.
    // cout << "S1==CP3 returned ";
    // trueFalse(S1==CP3);
  }
```

Output

```
S1 is ABC
S2 is ABC
S3 is DEF
S4 is
CP1 is abc
CP2 is DEF
S1==CP1 returned False
S1==CP2 returned False
CP1==S1 returned False
CP2==S1 returned False
S1==S2 returned True
S1==S3 returned False
S1==S4 returned False
```

Requirements

Header: <string>

See Also

Concepts

Standard Template Library Samples

string::operator>

Illustrates how to use the string::operator> Standard Template Library (STL) function in Visual C++.

```
template<class _E, class _TYPE, class _A> inline
   bool operator>(const basic_string<_E, _TYPE, _A>& LString,
   const _E *RCharArray);
template<class _E, class _TYPE, class _A> inline
   bool operator>(const _E *LCharArray,
   const basic_string<_E, _TYPE, _A>& RString);
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

There are two versions of the **string::operator>** function. Both versions are used to compare a null-terminated character array to a basic_string. They perform this operation by returning the result of (**RString**<**LCharArray or RCharArray**<**LString**). For more information on this comparison, see the **string::operator<** function. Note that this operator does not work with NULL pointer for the character array. You will need to make sure that the character array is not NULL before you pass it to the operator.

Example

```
// StringGreaterThan.cpp
// compile with: /EHsc
// Illustrates how to use the operator> to
// compare a basic_string variable to a
// null-terminated string.
//
// Functions:
//
//    operator>    Returns true if the second parameter is less than
//                 the first.
/////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <string>
#include <iostream>

using namespace std ;

void trueFalse(int x)
{
  cout << (x? "True": "False") << endl;
}

int main()
{
   string S1="ABC";
   char CP1[]="ABC";
   char CP2[]="DEF";
   char CP3[]="ABCD";
   char CP4[]="abc";

   cout << "S1 is " << S1 << endl;
   cout << "CP1 is " << CP1 << endl;
   cout << "CP2 is " << CP2 << endl;
   cout << "CP3 is " << CP3 << endl;
   cout << "CP4 is " << CP4 << endl;
```

```
    cout << "S1>CP1 returned ";
    trueFalse(S1>CP1);  // False  (calls function 1)

    cout << "S1>CP2 returned ";
    trueFalse(S1>CP2);  // False  (calls function 1)

    cout << "S1>CP4 returned ";
    trueFalse(S1>CP4);  // False   (calls function 1)

    cout << "CP1>S1 returned ";
    trueFalse(CP1>S1);  // False  (calls function 2)

    cout << "CP2>S1 returned ";
    trueFalse(CP2>S1);  // True   (calls function 2)

    cout << "CP3>S1 returned ";
    trueFalse(CP3>S1);  // True   (calls function 2)
  }
```

Output

```
S1 is ABC
CP1 is ABC
CP2 is DEF
CP3 is ABCD
CP4 is abc
S1>CP1 returned False
S1>CP2 returned False
S1>CP4 returned False
CP1>S1 returned False
CP2>S1 returned True
CP3>S1 returned True
```

Requirements

Header: <string>

See Also

Concepts

Standard Template Library Samples

string::operator>=

Illustrates how to use the string::operator>= STL function in Visual C++.

```
template<class _E, class _TYPE, class _A> inline
   bool operator>=(const basic_string<_E, _TYPE, _A>& _L,
   const _E *_R);
template<class _E, class _TYPE, class _A> inline
   bool operator>=(const _E * _L,
   const basic_string<_E, _TYPE, _A>& _R);
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

There are two versions of the **string::operator>=** function. Both versions are used to compare a null-terminated character array against a basic_string. They perform this operation by negating the result of (**_L** < **_R**). For more information on this comparison, see the **string::operator<** function.

Example

```
// StringGreaterEqual.cpp
// compile with: /EHsc
//
//     Illustrates how to use the operator>= to compare a
//     basic_string variable and a null-terminated string.
//
// Functions:
//
//     operator>=   Returns true if the first parameter is not less than
//                  the second.
//
////////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <string>
#include <iostream>

using namespace std ;

void truefalse(int x)
{
  cout << (x?"True":"False") << endl;
}


int main()
{
   string S1="ABC";
   char CP1[4]="ABC";
   char CP2[4]="DEF";

   cout << "S1 is " << S1 << endl;
   cout << "CP1 is " << CP1 << endl;
   cout << "CP2 is " << CP2 << endl;

   cout << "S1>=CP1 returned ";
   truefalse(S1>=CP1);  // True   (calls function 1)

   cout << "S1>=CP2 returned ";
   truefalse(S1>=CP2);  // False  (calls function 1)
```

```
    cout << "CP1>=S1 returned ";
    truefalse(CP1>=S1);  // True   (calls function 2)

    cout << "CP2>=S1 returned ";
    truefalse(CP2>=S1);  // True   (calls function 2)
  }
```

Output

```
S1 is ABC
CP1 is ABC
CP2 is DEF
S1>=CP1 returned True
S1>=CP2 returned False
CP1>=S1 returned True
CP2>=S1 returned True
```

Requirements

Header: <string>

See Also

Concepts

Standard Template Library Samples

string::operator>>

Illustrates how to use the string::operator>> Standard Template Library (STL) function in Visual C++.

```
template<class E, class TYPE, class A> inline
   basic_istream<E, TYPE>&
   operator>>(basic_istream<E, TYPE>& InStream,
   basic_string<E, TYPE, A>& String);
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **operator>>** is used to populate a string with the contents of an input stream.

Security Note

This operator copies data from an input source to a variable. If the input is not verified, this could potentially lead to a buffer overrun. For more information, see Avoiding Buffer Overruns.

Example

```
// string_operator_extract_sample.cpp
// compile with: /EHsc
//
// Illustrates how to use the operator>> to extract
// a string from an input stream, populating a string
// variable with the contents.
//
// Functions:
//
//    operator>>  Extracts a string from an input stream.
//////////////////////////////////////////////////////////////////////

#pragma warning(disable:4786)
#include <string>
#include <iostream>

using namespace std ;

int main()
{
   string s1;
   cout << "Enter a word: ";
   cin >> s1;
   cout << "You entered: " << s1 << endl;
}
```

Input

```
test
```

Sample Output

```
Enter a word: test
You entered: test
```

Requirements

Header: <string>

See Also

Concepts

Standard Template Library Samples

Trigonometry Functions

Illustrates how to use trigonometry (sin, cos, tan, and so on) functions in Visual C++.

```
// acos
template<class T>
   inline valarray<T> acos(
      const valarray<T>& x
   );
// asin
template<class T>
   inline valarray<T> asin(
      const valarray<T>& x
   );
// atan
template<class T>
   inline valarray<T> atan(
      const valarray<T>& x
   );
// atan2
template<class T>
   inline valarray<T> atan2(
      const valarray<T>& x,
      const valarray<T>& y
   );
template<class T>
   inline valarray<T> atan2(
      const valarray<T> x,
      const T& y
   );
template<class T>
   inline valarray<T> atan2(
      const T& x,
      const valarray<T>& y
   );
// cos
template<class T>
   inline valarray<T> cos(
      const valarray<T>& x
   );
// cosh
template<class T>
   inline valarray<T> cosh(
      const valarray<T>& x
   );
// sin
template<class T>
   inline valarray<T> sin(
      const valarray<T>& x
   );
// sinh
template<class T>
   inline valarray<T> sinh(
      const valarray<T>& x
   );
// tan
template<class T>
   inline valarray<T> tan(
      const valarray<T>& x
   );
// tanh
template<class T>
   inline valarray<T> tanh(
      const valarray<T>& x
```

```
    );
```

Example

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

```
// trig.cpp
// compile with: /EHsc
// Illustrates the use of STL trigonometry functions.
// Functions:
//  acos, asin, atan, atan2, cos, cosh, sin, sinh, tan, tanh
/////////////////////////////////////////////////////////////////////

#include <iostream>                 // for i/o functions
#include <valarray>                 // for valarray
#include <cmath>                    // for trigonometry functions

using namespace std ;

#define ARRAY_SIZE  3               // array size

int main()
{
   // Initialize val_array to values -1, 0 and 1.
   valarray<double> val_array(ARRAY_SIZE);
   int i;
   for (i = 0; i < ARRAY_SIZE; i++)
      val_array[i] = i - 1;

   // Display the size of val_array.
   cout << "Size of val_array = " << val_array.size() << endl;

   // Display the values of val_array before calling any trigonometry
   // functions.
   cout << "The values in val_array:" << endl << "[";
   for (i = 0; i < ARRAY_SIZE; i++)
      cout << " " << val_array[i];
   cout << " ]" << endl << endl;

   // Initialize rev_valarray that is the reverse of val_array.
   valarray<double> rev_valarray(ARRAY_SIZE);
   for (i = 0; i < ARRAY_SIZE; i++)
      rev_valarray[i] = val_array[ARRAY_SIZE - i - 1];

   // Display the size of rev_valarray.
   cout << "Size of rev_valarray = " << rev_valarray.size() << endl;

   // Display the values of rev_valarray.
   cout << "The values in rev_valarray:" << endl << "[";
   for (i = 0; i < ARRAY_SIZE; i++)
      cout << " " << rev_valarray[i];
   cout << " ]" << endl << endl;

   // rvalue_array to hold the return value from calling the trigonometry
   // functions.
   valarray<double> rvalue_array;

   // ---------------------------------------------------------------
   // acos() - display the result of rvalue_array
   // ---------------------------------------------------------------
   rvalue_array = acos(val_array);
   cout << "The result after calling acos():" << endl << "[";
   for (i = 0; i < ARRAY_SIZE; i++)
      cout << " " << rvalue_array[i];
```

```
cout << " ]" << endl << endl;

// ----------------------------------------------------------------
// asin() - display the result of rvalue_array
// ----------------------------------------------------------------
rvalue_array = asin(val_array);
cout << "The result after calling asin():" << endl << "[";
for (i = 0; i < ARRAY_SIZE; i++)
   cout << " " << rvalue_array[i];
cout << " ]" << endl << endl;

// ----------------------------------------------------------------
// atan() - display the result of rvalue_array
// ----------------------------------------------------------------
rvalue_array = atan(val_array);
cout << "The result after calling atan():" << endl << "[";
for (i = 0; i < ARRAY_SIZE; i++)
   cout << " " << rvalue_array[i];
cout << " ]" << endl << endl;

// ----------------------------------------------------------------
// atan2() - display the result of rvalue_array
// ----------------------------------------------------------------

// This template function returns an object of class valarray<T>,
// each of whose elements at I is the arctangent of x[I] / y[I].
rvalue_array = atan2(val_array, rev_valarray);
cout << "The result after calling atan2(val_array, rev_valarray):"
     << endl << "[";
for (i = 0; i < ARRAY_SIZE; i++)
   cout << " " << rvalue_array[i];
cout << " ]" << endl << endl;

// This template function stores in element I the arctangent of
// x[I] / y.
rvalue_array = atan2(val_array, 3.1416);
cout << "The result after calling atan2(val_array, 3.1416):"
     << endl << "[";
for (i = 0; i < ARRAY_SIZE; i++)
   cout << " " << rvalue_array[i];
cout << " ]" << endl << endl;

// This template function stores in element I the arctangent of
// x / y[I].
rvalue_array = atan2(3.1416, val_array);
cout << "The result after calling atan2(3.1416, val_array):"
     << endl << "[";
for (i = 0; i < ARRAY_SIZE; i++)
   cout << " " << rvalue_array[i];
cout << " ]" << endl << endl;

// ----------------------------------------------------------------
// cos() - display the result of rvalue_array
// ----------------------------------------------------------------
rvalue_array = cos(val_array);
cout << "The result after calling cos():" << endl << "[";
for (i = 0; i < ARRAY_SIZE; i++)
   cout << " " << rvalue_array[i];
cout << " ]" << endl << endl;

// ----------------------------------------------------------------
// cosh() - display the result of rvalue_array
// ----------------------------------------------------------------
rvalue_array = cosh(val_array);
cout << "The result after calling cosh():" << endl << "[";
for (i = 0; i < ARRAY_SIZE; i++)
   cout << " " << rvalue_array[i];
cout << " ]" << endl << endl;
```

```
    // ----------------------------------------------------------------
    // sin() - display the result of val_array
    // ----------------------------------------------------------------
    rvalue_array = sin(val_array);
    cout << "The result after calling sin():" << endl << "[";
    for (i = 0; i < ARRAY_SIZE; i++)
       cout << " " << rvalue_array[i];
    cout << " ]" << endl << endl;

    // ----------------------------------------------------------------
    // sinh() - display the result of val_array
    // ----------------------------------------------------------------
    rvalue_array = sinh(val_array);
    cout << "The result after calling sinh():" << endl << "[";
    for (i = 0; i < ARRAY_SIZE; i++)
       cout << " " << rvalue_array[i];
    cout << " ]" << endl << endl;

    // ----------------------------------------------------------------
    // tan() - display the result of val_array
    // ----------------------------------------------------------------
    rvalue_array = tan(val_array);
    cout << "The result after calling tan():" << endl << "[";
    for (i = 0; i < ARRAY_SIZE; i++)
       cout << " " << rvalue_array[i];
    cout << " ]" << endl << endl;

    // ----------------------------------------------------------------
    // tanh() - display the result of val_array
    // ----------------------------------------------------------------
    rvalue_array = tanh(val_array);
    cout << "The result after calling tanh():" << endl << "[";
    for (i = 0; i < ARRAY_SIZE; i++)
       cout << " " << rvalue_array[i];
    cout << " ]" << endl;
  }
```

Output

```
Size of val_array = 3
The values in val_array:
[ -1 0 1 ]

Size of rev_valarray = 3
The values in rev_valarray:
[ 1 0 -1 ]

The result after calling acos():
[ 3.14159 1.5708 0 ]

The result after calling asin():
[ -1.5708 0 1.5708 ]

The result after calling atan():
[ -0.785398 0 0.785398 ]

The result after calling atan2(val_array, rev_valarray):
[ -0.785398 0 2.35619 ]

The result after calling atan2(val_array, 3.1416):
[ -0.308168 0 0.308168 ]

The result after calling atan2(3.1416, val_array):
[ 1.87896 1.5708 1.26263 ]

The result after calling cos():
[ 0.540302 1 0.540302 ]

The result after calling cosh():
```

```
[ 1.54308 1 1.54308 ]

The result after calling sin():
[ -0.841471 0 0.841471 ]

The result after calling sinh():
[ -1.1752 0 1.1752 ]

The result after calling tan():
[ -1.55741 0 1.55741 ]

The result after calling tanh():
[ -0.761594 0 0.761594 ]
```

Requirements

Header: <valarray>

See Also

Reference

acos
asin
atan
atan2
cosh (<valarray>)
sinh (<valarray>)
tanh (<valarray>)

Concepts

Standard Template Library Samples

unary_function<> Structure

Illustrates how to use the unary_function<> structure in Visual C++.

```
template<class _A, class _R>
   struct unary_function
   {
      typedef _A Argument_Type;
      typedef _R Result_Type;
   };
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

unary_function is used as a base class to better define operator functions in the following format: *Result_Type classname*::**operator***X*(*Argument_Type*).

Example

```
// unary_function.cpp
// compile with: /EHsc
//
// Structure used:
//       unary_function<int, float> - allows us
//       to write operator functions accepting an
//       integer and returning floats.
/////////////////////////////////////////////

#include <functional>
#include <iostream>

using namespace std ;

/* derive class from unary_function in order to use it */

class unary_test : public unary_function<int,float>
{
public:
  float value;
  unary_test(){value=10.0;}
  unary_test(float x){value=x;}
  result_type operator*(argument_type x);
  result_type operator-(argument_type x);
};

/* You can now easily create operators that accept */
/* an int and return a float.                      */

unary_test::result_type unary_test::operator*(unary_test::argument_type x)
{
  float tmp = value * (float)x;
  cout << "Value after * is " << tmp << endl ;
  return value;
}

unary_test::result_type unary_test::operator-(unary_test::argument_type x)
{
  float tmp = value - (float)x;
  cout << "Value after minus is " << tmp << endl ;
  return tmp;
```

```
}

int main(void)
{
  unary_test item;
  unary_test item2(18.0);

  cout << "Begin" << endl ;
  cout.setf(ios::fixed) ;
  item = item * 2;
  item2 = item2 - 5;
}
```

Output

```
Begin
Value after * is 20.000000
Value after minus is 13.000000
```

Requirements

Header: <functional>

See Also

Concepts

Standard Template Library Samples

upper_bound (STL Samples)

Illustrates how to use the upper_bound STL function in Visual C++.

```
template<class ForwardIterator, class T>
   inline ForwardIterator upper_bound(
      ForwardIterator First,
      ForwardIterator Last,
      const T& Value
)
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The **upper_bound** algorithm returns the last location in the sequence that value can be inserted such that the order of the sequence is maintained. **upper_bound** returns an iterator positioned at the location that value can be inserted in the range [*First*..*Last*), or returns last if no such position exists. **upper_bound** assumes the range [*First* ..*Last*) is sorted using **operator<**.

Example

```
// upper_bound_STL.cpp
// compile with: /EHsc
// Illustrates how to use the upper_bound function.
//
// Functions:
//
//    upper_bound : Return the upper bound within a range.
//////////////////////////////////////////////////////////////////////

// disable warning C4786: symbol greater than 255 character,
// okay to ignore
#pragma warning(disable: 4786)

#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std;


int main()
{
    const int VECTOR_SIZE = 8 ;

    // Define a template class vector of int
    typedef vector<int > IntVector ;

    //Define an iterator for template class vector of strings
    typedef IntVector::iterator IntVectorIt ;

    IntVector Numbers(VECTOR_SIZE) ;

    IntVectorIt start, end, it, location ;

    // Initialize vector Numbers
    Numbers[0] = 4 ;
    Numbers[1] = 10;
    Numbers[2] = 10 ;
    Numbers[3] = 30 ;
```

```
    Numbers[4] = 69 ;
    Numbers[5] = 70 ;
    Numbers[6] = 96 ;
    Numbers[7] = 100;

    start = Numbers.begin() ;   // location of first
                                // element of Numbers

    end = Numbers.end() ;       // one past the location
                                // last element of Numbers

    // print content of Numbers
    cout << "Numbers { " ;
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;

    //return the last location at which 10 can be inserted
    // in Numbers
    location = upper_bound(start, end, 10) ;

    cout << "Element 10 can be inserted at index "
        << location - start << endl ;
}
```

Output

```
Numbers { 4 10 10 30 69 70 96 100  }

Element 10 can be inserted at index 3
```

Requirements

Header: <algorithm>

See Also

Concepts

Standard Template Library Samples

vector::operator<

Illustrates how to use the vector::operator< Standard Template Library (STL) function in Visual C++.

```
template<class _TYPE, class _A> inline
   bool operator<(
   const vector<_TYPE, _A>& _X,
   const vector<_TYPE, _A>& _Y
);
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The sample declares an empty vector of IDs, a user-defined type. It initializes and adds four IDs to the vector in random order. It sorts them using the **operator<** defined for ID and generates the newly sorted vector. (Note that it sorts in order of Score, not Name.)

Example

```
// Opless.cpp
// compile with: /EHsc
// Illustrates the defining the < operator to sort vectors
//
// Functions:
//
// operator< - Vector comparison operator.
//
// vector::begin - Returns an iterator to start traversal of the vector.
//
// vector::end - Returns an iterator for the last element of the vector.
//
// vector::iterator - Traverses the vector.
//
// vector::push_back - Appends (inserts) an element to the end of a
// vector, allocating memory for it if necessary.
//
// sort algorithm - Sorts the vector.
//
//////////////////////////////////////////////////////////////////////

// The debugger can't handle symbols more than 255 characters long.
// STL often creates symbols longer than that.
// When symbols are longer than 255 characters, the warning is disabled.
#pragma warning(disable:4786)

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std ;

// The ID class is used for team scoring. It holds each player's name
// and score.
class ID
{
public:
    string Name;
    int Score;
    ID() : Name(""), Score(0) {}
```

```
    ID(string NewName, int NewScore) : Name(NewName), Score(NewScore) {}
};

// In this example, an ID is equivalent only if both name and score match.
bool operator==(const ID& x, const ID& y)
{
    return (x.Name == y.Name) && (x.Score == y.Score);
}

// IDs will be sorted by Score, not by Name.
bool operator<(const ID& x, const ID& y)
{
    return x.Score < y.Score;
}

// Define a template class for a vector of IDs.
typedef vector<ID> NAMEVECTOR;

int main()
{
    // Declare a dynamically allocated vector of IDs.
    NAMEVECTOR theVector;

    // Iterator is used to loop through the vector.
    NAMEVECTOR::iterator theIterator;

    // Create a pseudo-random vector of players and scores.
    theVector.push_back(ID("Karen Palmer", 2));
    theVector.push_back(ID("Ada Campbell", 1));
    theVector.push_back(ID("John Woloschuk", 3));
    theVector.push_back(ID("Grady Leno", 2));

    cout << "Players and scores:" << endl;
    for (theIterator = theVector.begin(); theIterator != theVector.end();
         theIterator++)
        cout << theIterator->Score  << "     "
             << theIterator->Name << endl;
    cout << endl;

    // Sort the vector of players by score.
    sort(theVector.begin(), theVector.end());

    // Output the contents of the vector in its new, sorted order.
    cout << "Players ranked by score:" << endl;
    for (theIterator = theVector.begin(); theIterator != theVector.end();
         theIterator++)
        cout << theIterator->Score  << "     "
             << theIterator->Name << endl;
}
```

Output

```
Players and scores:
2     Karen Palmer
1     Ada Campbell
3     John Woloschuk
2     Grady Leno

Players ranked by score:
1     Ada Campbell
2     Karen Palmer
2     Grady Leno
3     John Woloschuk
```

Requirements

Header: <vector>

See Also

Concepts

Standard Template Library Samples

vector::operator==

Illustrates how to use the vector::operator== Standard Template Library (STL) function in Visual C++.

```
template<class _TYPE, class _A> inline
   bool operator==(
   const vector<_TYPE, _A>& _X,
   const vector<_TYPE, _A>& _Y
);
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The sample declares three empty vectors of a user-defined class called ID, each of which contains a Name string member and a Score integer member. It creates three vectors of IDs and then compares vectors using the **operator==** as defined for ID.

Example

```
// Opequal.cpp
// compile with: /EHsc
// Illustrates how to define the operator== to compare vectors.
//
// Functions:
//
//    vector::operator== - Vector equality comparison.
//    vector::push_back - Appends (inserts) an element to the end of a
//                        vector, allocating memory for it if necessary.
//
//////////////////////////////////////////////////////////////////////

// The debugger can't handle symbols more than 255 characters long.
// STL often creates symbols longer than that.
// When symbols are longer than 255 characters, the warning is disabled.
#pragma warning(disable:4786)

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std ;
using namespace std::rel_ops ;

// The ID class is used for team scoring. It holds each player's name
// and score.
class ID
{
public:
    string Name;
    int Score;

    ID() : Name(""), Score(0) {}
    ID(string NewName, int NewScore) : Name(NewName), Score(NewScore) {}
};

// In this example, an ID is equivalent only if both name and score match.
bool operator==(const ID& x, const ID& y)
{
    return (x.Name == y.Name) && (x.Score == y.Score);
}
```

```
// IDs will be sorted by Score, not by Name.
bool operator<(const ID& x, const ID& y)
{
    return x.Score < y.Score;
}

// Define a template class for a vector of IDs.
typedef vector<ID> NAMEVECTOR;

int main()
{
    // Declare 3 dynamically allocated vectors of names.
    NAMEVECTOR Vector1, Vector2, Vector3;

    // Create 3 short vectors of names.
    Vector1.push_back(ID("Karen Palmer", 2));
    Vector1.push_back(ID("Ada Campbell", 1));

    Vector2.push_back(ID("John Woloschuk", 3));
    Vector2.push_back(ID("Grady Leno", 2));

    Vector3.push_back(ID("Karen Palmer", 2));
    Vector3.push_back(ID("Ada Campbell", 1));

    // Compare Vector1 to Vector2 and show whether they're equivalent.
    Vector1 == Vector2 ? cout << "Vector1 == Vector2"
                       : cout << "Vector1 != Vector2";
    cout << endl;

    // Compare Vector1 to Vector3 and show whether they're equivalent.
    Vector1 == Vector3 ? cout << "Vector1 == Vector3"
                       : cout << "Vector1 != Vector3";
    cout << endl;
}
```

Output

```
Vector1 != Vector2
Vector1 == Vector3
```

Requirements

Header: <vector>

See Also

Concepts

Standard Template Library Samples

vector::empty, vector::erase, and vector::push_back

Illustrates how to use the vector::empty, vector::erase, and vector::push_back STL functions in Visual C++.

```
template<class _TYPE, class _A>
   void vector::push_back(
      const _TYPE& X
   );
template<class _TYPE, class _A>
   iterator vector::erase(
      iterator Iterator
   );
template<class _TYPE, class _A>
   iterator vector::erase(
      iterator First,
      iterator Last
   );
template<class _TYPE, class _A>
    bool vector::empty( ) const;
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The sample declares an empty vector of integers. It adds 10 integers to the vector and then displays the contents of the vector. It deletes the sixth element by using erase, and then displays the contents of the vector again. It deletes the rest of the elements using a different form of **erase** and then displays the vector (now empty) again. The ShowVector routine uses the empty function to determine whether to generate the contents of the vector.

Example

```
// Empty.cpp
// compile with: /EHsc
// Illustrates the vector::empty and vector::erase functions.
// Also demonstrates the vector::push_back function.
//
// Functions:
//
//    vector::empty - Returns true if vector has no elements.
//
//    vector::erase - Deletes elements from a vector (single & range).
//
//    vector::begin - Returns an iterator to start traversal of the
//                    vector.
//
//    vector::end - Returns an iterator for the last element of the
//                  vector.
//
//    vector::push_back - Appends (inserts) an element to the end of a
//                        vector, allocating memory for it if necessary.
//
//    vector::iterator - Traverses the vector.
//
///////////////////////////////////////////////////////////////////////

// The debugger can't handle symbols more than 255 characters long.
// STL often creates symbols longer than that.
// When symbols are longer than 255 characters, the warning is disabled.
#pragma warning(disable:4786)
```

```
#include <iostream>
#include <vector>

using namespace std ;

typedef vector<int> INTVECTOR;

const int ARRAY_SIZE = 10;

void ShowVector(INTVECTOR &theVector);

int main()
{
    // Dynamically allocated vector begins with 0 elements.
    INTVECTOR theVector;

    // Intialize the vector to contain the numbers 0-9.
    for (int cEachItem = 0; cEachItem < ARRAY_SIZE; cEachItem++)
        theVector.push_back(cEachItem);

    // Output the contents of the dynamic vector of integers.
    ShowVector(theVector);

    // Using void iterator erase(iterator Iterator) to
    // delete the 6th element (Index starts with 0).
    theVector.erase(theVector.begin() + 5);

    // Output the contents of the dynamic vector of integers.
    ShowVector(theVector);

    // Using iterator erase(iterator First, iterator Last) to
    // delete a range of elements all at once.
    theVector.erase(theVector.begin(), theVector.end());

    // Show what's left (actually, nothing).
    ShowVector(theVector);
}

// Output the contents of the dynamic vector or display a
// message if the vector is empty.
void ShowVector(INTVECTOR &theVector)
{
    // First see if there's anything in the vector. Quit if so.
    if (theVector.empty())
    {
        cout << "theVector is empty." << endl;
        return;
    }

    // Iterator is used to loop through the vector.
    INTVECTOR::iterator theIterator;

    // Output contents of theVector.
    cout << "theVector [ " ;
    for (theIterator = theVector.begin(); theIterator != theVector.end();
         theIterator++)
    {
        cout << *theIterator;
        if (theIterator != theVector.end()-1) cout << ", ";
                                              // cosmetics for the output
    }
    cout << " ]" << endl ;
}
```

Output

```
theVector [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
theVector [ 0, 1, 2, 3, 4, 6, 7, 8, 9 ]
```

```
theVector is empty.
```

Requirements

Header: <vector>

See Also

Concepts

Standard Template Library Samples

vector::front and vector::back

Illustrates how to use the vector::front and vector::back Standard Template Library (STL) functions in Visual C++.

```
template<class _TYPE, class _A>
   reference vector::front( );
template<class _TYPE, class _A>
   reference vector::back( );
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The sample declares an empty vector of integers with the members [100, 200, 300, 400]. It displays the first element of the vector using **vector::front** to obtain it. It displays the last element of the vector using **vector::back** to obtain it. It displays the number of elements of the vector using **vector::size**. The sample erases the last element of the vector using **vector::end** – 1, and then displays the new last element using **vector::back**. It erases the first element of the vector using **vector::begin**, and then displays the new first element using **vector::front**. Finally, the sample displays the number of elements remaining in the vector using **vector::size**.

Example

```
// frontback.cpp
// compile with: /EHsc
//
// Illustrates the vector::front and vector::back methods.
//
// Functions:
//
// vector::front - Returns reference to first element of vector.
//
// vector::back - Returns reference to last element of vector.
//
// vector::push_back - Appends (inserts) an element to the end of a
//                        vector, allocating memory for it if necessary.
//
// vector::size - Returns number of elements in the vector.
//
// vector::begin - Returns an iterator to start traversal of the vector.
//
// vector::end - Returns an iterator for the last element of the vector.
//
// vector::erase - Deletes elements from a vector (single & range).
//
///////////////////////////////////////////////////////////////////////

// The debugger can't handle symbols more than 255 characters long.
// STL often creates symbols longer than that.
// When symbols are longer than 255 characters, the warning is issued.
#pragma warning(disable:4786)

#include <iostream>
#include <vector>

using namespace std ;

typedef vector<int> INTVECTOR;

const int ARRAY_SIZE = 4;
```

```
int main()
{
    // Dynamically allocated vector begins with 0 elements.
    INTVECTOR theVector;

    // Intialize the array to contain the members [100, 200, 300, 400]
    for (int cEachItem = 0; cEachItem < ARRAY_SIZE; cEachItem++)
        theVector.push_back((cEachItem + 1) * 100);

    cout << "First element: " << theVector.front() << endl;
    cout << "Last element: " << theVector.back() << endl;
    cout << "Elements in vector: " << theVector.size() << endl;

    // Delete the last element of the vector. Remember that the vector
    // is 0-based, so theVector.end() actually points 1 element beyond
    // the end.
    theVector.erase(theVector.end() - 1);

    cout << endl << "After erasing last element, new last element is: "
         << theVector.back() << endl;

    // Delete the first element of the vector.
    theVector.erase(theVector.begin());

    cout << "After erasing first element, new first element is: "
         << theVector.front() << endl;

    cout << "Elements in vector: " << theVector.size() << endl;
}
```

Output

```
First element: 100
Last element: 400
Elements in vector: 4

After erasing last element, new last element is: 300
After erasing first element, new first element is: 200
Elements in vector: 2
```

Requirements

Header: <vector>

See Also

Concepts

Standard Template Library Samples

vector::push_back and vector::pop_back

Illustrates how to use the vector::push_back and vector::pop_back Standard Template Library (STL) functions in Visual C++.

```
template<class _TYPE, class _A>
   void vector::push_back(
      const _TYPE& X
   );
template<class _TYPE, class _A>
   void vector::pop_back();
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The sample declares an empty vector of integers. It adds three integers to the vector, and then deletes one. Finally, it generates the remaining elements in the vector.

Example

```
// Pushpop.cpp
// compile with: /EHsc
// Illustrates how to use the push and pop member
// functions of the vector container.
//
// Functions:
//
// vector::push_back - Appends (inserts) an element to the end of a
// vector, allocating memory for it if necessary.
//
// vector::pop_back -  Erases the last element of the vector.
//
// vector::begin - Returns an iterator to start traversal of the vector.
//
// vector::end - Returns an iterator for the last element of the vector.
//
// vector::iterator - Traverses the vector.
//
///////////////////////////////////////////////////////////////////////

// The debugger can't handle symbols more than 255 characters long.
// STL often creates symbols longer than that.
// When symbols are longer than 255 characters, the warning is disabled.
#pragma warning(disable:4786)

#include <iostream>
#include <vector>

using namespace std ;

typedef vector<int> INTVECTOR;

int main()
{
    // Dynamically allocated vector begins with 0 elements.
    INTVECTOR theVector;

    // Iterator is used to loop through the vector.
    INTVECTOR::iterator theIterator;

    // Add one element to the end of the vector, an int with the value 42.
```

```
    // Allocate memory if necessary.
    theVector.push_back(42) ;

    // Add two more elements to the end of the vector.
    // theVector will contain [ 42, 1, 109 ].
    theVector.push_back(1) ;
    theVector.push_back(109) ;

    // Erase last element in vector.
    theVector.pop_back();

    // Print contents of theVector. Shows [ 42, 1 ]
    cout << "theVector [ " ;
    for (theIterator = theVector.begin(); theIterator != theVector.end();
         theIterator++)
    {
        cout << *theIterator;
        if (theIterator != theVector.end()-1) cout << ", ";

    }
    cout << " ]" << endl ;
}
```

Output

```
theVector [ 42, 1 ]
```

Requirements

Header: <vector>

See Also

Reference

abs (<valarray>)

Concepts

Standard Template Library Samples

vector::size and vector::capacity

Illustrates how to use the vector::size and vector::capacity Standard Template Library (STL) functions in Visual C++.

```
template<class _TYPE, class _A>
   void vector::reserve(
      size_type _N
   );
template<class _TYPE, class _A>
   size_type vector::max_size( ) const;
template<class _TYPE, class _A>
   void vector::resize(
      size_type _N,
      _TYPE _X = _TYPE( )
   );
template<class _TYPE, class _A>
   size_type vector::capacity( ) const;
template<class _TYPE, class _A>
   size_type vector::size( ) const;
```

Remarks

Note

The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.

The sample declares an empty vector of integers. It adds a single integer element to the vector, and then shows information about the vector's size, maximum size, and capacity. It reserves storage for 1,000 elements and displays the same information about the vector. Finally, it resizes the vector to 2,000 elements and then displays the information.

Example

```
// Remax.cpp
// compile with: /EHsc
// Illustrates vector::reserve, vector::max_size,
// vector::resize, vector::resize, and vector::capacity.
//
// Functions:
//
//    vector::max_size - Returns maximum number of elements vector could
//                       hold.
//
//    vector::capacity - Returns number of elements for which memory has
//                       been allocated.
//
//    vector::size - Returns number of elements in the vector.
//
//    vector::resize - Reallocates memory for vector, preserves its
//                     contents if new size is larger than existing size.
//
//    vector::reserve - Allocates elements for vector to ensure a minimum
//                      size, preserving its contents if the new size is
//                      larger than existing size.
//
//    vector::push_back - Appends (inserts) an element to the end of a
//                        vector, allocating memory for it if necessary.
//
///////////////////////////////////////////////////////////////////////

// The debugger cannot handle symbols more than 255 characters long.
// STL often creates symbols longer than that.
// When symbols are longer than 255 characters, the warning is disabled.
#pragma warning(disable:4786)
```

```
#include <iostream>
#include <vector>

using namespace std ;

typedef vector<int> INTVECTOR;

int main()
{
    // Dynamically allocated vector begins with 0 elements.
    INTVECTOR theVector;

    // Add one element to the end of the vector, an int with the value 42.
    theVector.push_back(42) ;

    // Show statistics about vector.
    cout << "theVector's size is: " << theVector.size() << endl;
    cout << "theVector's maximum size is: " << theVector.max_size()
         << endl;
    cout << "theVector's capacity is: " << theVector.capacity() << endl;

    // Ensure there's room for at least 1000 elements.
    theVector.reserve(1000);
    cout << endl << "After reserving storage for 1000 elements:" << endl;
    cout << "theVector's size is: " << theVector.size() << endl;
    cout << "theVector's maximum size is: " << theVector.max_size()
         << endl;
    cout << "theVector's capacity is: " << theVector.capacity() << endl;

    // Ensure there's room for at least 2000 elements.
    theVector.resize(2000);
    cout << endl << "After resizing storage to 2000 elements:" << endl;
    cout << "theVector's size is: " << theVector.size() << endl;
    cout << "theVector's maximum size is: " << theVector.max_size()
         << endl;
    cout << "theVector's capacity is: " << theVector.capacity() << endl;
}
```

Sample Output

The following output is for x86.

```
theVector's size is: 1
theVector's maximum size is: 1073741823
theVector's capacity is: 1

After reserving storage for 1000 elements:
theVector's size is: 1
theVector's maximum size is: 1073741823
theVector's capacity is: 1000

After resizing storage to 2000 elements:
theVector's size is: 2000
theVector's maximum size is: 1073741823
theVector's capacity is: 2000
```

Requirements

Header: <vector>

See Also

Concepts

Standard Template Library Samples

C++ Support Library

The C++ Support Library provides classes that support managed programming in C++.

In This Section

Resource Management Classes

Synchronization (lock Class)

Calling Functions in a Specific Application Domain

com::ptr

The msclr namespace

The **msclr** namespace contains all the classes of the C++ Support Library. For more information on those classes, see C++ Support Library.

See Also

Other Resources

C++ Support Library

Resource Management Classes

These classes provide automatic management of managed classes.

In This Section

auto_gcroot	Embeds a virtual handle in a native type.
auto_handle	Embeds a virtual handle in a managed type.

See Also

Other Resources

C++ Support Library

auto_gcroot

Defines the **auto_gcroot** class and **swap** function.

```
#include <msclr\auto_gcroot.h>
```

Remarks

In this header file:

auto_gcroot Class

swap Function (auto_gcroot)

See Also

Other Resources

C++ Support Library

auto_gcroot Class

Automatic resource management (like auto_ptr Class) which can be used to embed a virtual handle into a native type.

```
template<typename _element_type>
class auto_gcroot;
```

Parameters

_element_type

The managed type to be embedded.

Requirements

Header file <msclr\auto_gcroot.h>

Namespace msclr

See Also

Tasks

How to: Declare Handles in Native Types

Reference

auto_gcroot

auto_handle Class

Other Resources

auto_gcroot Members

auto_gcroot Members

Members of **auto_gcroot**.

In This Section

auto_gcroot::auto_gcroot

auto_gcroot::~auto_gcroot

auto_gcroot::attach

auto_gcroot::get

auto_gcroot::release

auto_gcroot::reset

auto_gcroot::swap

auto_gcroot::operator=

auto_gcroot::operator->

auto_gcroot::operator!

auto_gcroot::operator auto_gcroot

auto_gcroot::operator bool

See Also

Reference

auto_gcroot Class

auto_gcroot

auto_gcroot::auto_gcroot

The **auto_gcroot** constructor.

```
auto_gcroot(
   _element_type _ptr = nullptr
);
auto_gcroot(
   auto_gcroot<_element_type> & _right
);
template<typename _other_type>
auto_gcroot(
   auto_gcroot<_other_type> & _right
);
```

Parameters

_ptr

The object to own.

_right

An existing **auto_gcroot**.

Remarks

When constructing an **auto_gcroot** from an existing **auto_gcroot**, the existing **auto_gcroot** releases its object before transferring ownership of the object to the new **auto_gcroot**.

Example

```
// msl_auto_gcroot_auto_gcroot.cpp
// compile with: /clr
#include <msclr\auto_gcroot.h>

using namespace System;
using namespace msclr;

ref class RefClassA {
protected:
   String^ m_s;
public:
   RefClassA(String^ s) : m_s(s) {
      Console::WriteLine( "in RefClassA constructor: " + m_s );
   }
   ~RefClassA() {
      Console::WriteLine( "in RefClassA destructor: " + m_s );
   }

   virtual void PrintHello() {
      Console::WriteLine( "Hello from {0} A!", m_s );
   }
};

ref class RefClassB : RefClassA {
public:
   RefClassB( String^ s ) : RefClassA( s ) {}
   virtual void PrintHello() new {
      Console::WriteLine( "Hello from {0} B!", m_s );
   }
};

class ClassA { //unmanaged class
private:
   auto_gcroot<RefClassA^> m_a;
```

```
public:
   ClassA() : m_a( gcnew RefClassA( "unmanaged" ) ) {}
   ~ClassA() {} //no need to delete m_a

   void DoSomething() {
      m_a->PrintHello();
   }
};

int main()
{
   {
      ClassA a;
      a.DoSomething();
   } // a.m_a is automatically destroyed as a goes out of scope

   {
      auto_gcroot<RefClassA^> a(gcnew RefClassA( "first" ) );
      a->PrintHello();
   }

   {
      auto_gcroot<RefClassB^> b(gcnew RefClassB( "second" ) );
      b->PrintHello();
      auto_gcroot<RefClassA^> a(b); //construct from derived type
      a->PrintHello();
      auto_gcroot<RefClassA^> a2(a); //construct from same type
      a2->PrintHello();
   }

   Console::WriteLine("done");
}
```

Output

```
in RefClassA constructor: unmanaged
Hello from unmanaged A!
in RefClassA destructor: unmanaged
in RefClassA constructor: first
Hello from first A!
in RefClassA destructor: first
in RefClassA constructor: second
Hello from second B!
Hello from second A!
Hello from second A!
in RefClassA destructor: second
done
```

Requirements

Header file <msclr\auto_gcroot.h>

Namespace msclr

See Also

Reference

auto_gcroot::attach

auto_gcroot::operator=

auto_gcroot::~auto_gcroot

Other Resources

auto_gcroot Members

auto_gcroot::~auto_gcroot

The **auto_gcroot** destructor.

```
~auto_gcroot();
```

Remarks

The destructor also destructs the owned object.

Example

```
// msl_auto_gcroot_dtor.cpp
// compile with: /clr
#include <msclr\auto_gcroot.h>

using namespace System;
using namespace msclr;

ref class ClassA {
public:
   ClassA() { Console::WriteLine( "ClassA constructor" ); }
   ~ClassA() { Console::WriteLine( "ClassA destructor" ); }
};

int main()
{
   // create a new scope for a:
   {
      auto_gcroot<ClassA^> a = gcnew ClassA;
   }
   // a goes out of scope here, invoking its destructor
   // which in turns destructs the ClassA object.

   Console::WriteLine( "done" );
}
```

Output

```
ClassA constructor
ClassA destructor
done
```

Requirements

Header file <msclr\auto_gcroot.h>

Namespace msclr

See Also

Reference

auto_gcroot::release

auto_gcroot::auto_gcroot

Other Resources

auto_gcroot Members

auto_gcroot::attach

Attach **auto_gcroot** to an object.

```
auto_gcroot<_element_type> & attach(
   _element_type _right
);
auto_gcroot<_element_type> & attach(
   auto_gcroot<_element_type> & _right
);
template<typename _other_type>
auto_gcroot<_element_type> & attach(
   auto_gcroot<_other_type> & _right
);
```

Parameters

_right

The object to attach, or an **auto_gcroot** containing the object to attach.

Return Value

The current **auto_gcroot**.

Remarks

If *_right* is an **auto_gcroot**, it releases ownership of its object before the object is attached to the current **auto_gcroot**.

Example

```
// msl_auto_gcroot_attach.cpp
// compile with: /clr
#include <msclr\auto_gcroot.h>

using namespace System;
using namespace msclr;

ref class ClassA {
protected:
   String^ m_s;
public:
   ClassA( String^ s ) : m_s( s ) {
      Console::WriteLine( "in ClassA constructor:" + m_s );
   }
   ~ClassA() {
      Console::WriteLine( "in ClassA destructor:" + m_s );
   }

   virtual void PrintHello() {
      Console::WriteLine( "Hello from {0} A!", m_s );
   }
};

ref class ClassB : ClassA {
public:
   ClassB( String ^ s) : ClassA( s ) {}
   virtual void PrintHello() new {
      Console::WriteLine( "Hello from {0} B!", m_s );
   }
};

int main() {
   auto_gcroot<ClassA^> a( gcnew ClassA( "first" ) );
   a->PrintHello();
   a.attach( gcnew ClassA( "second" ) ); // attach same type
```

```
    a->PrintHello();
    ClassA^ ha = gcnew ClassA( "third" );
    a.attach( ha ); // attach raw handle
    a->PrintHello();
    auto_gcroot<ClassB^> b( gcnew ClassB("fourth") );
    b->PrintHello();
    a.attach( b ); // attach derived type
    a->PrintHello();
  }
```

Output

```
in ClassA constructor:first
Hello from first A!
in ClassA constructor:second
in ClassA destructor:first
Hello from second A!
in ClassA constructor:third
in ClassA destructor:second
Hello from third A!
in ClassA constructor:fourth
Hello from fourth B!
in ClassA destructor:third
Hello from fourth A!
in ClassA destructor:fourth
```

Requirements

Header file <msclr\auto_gcroot.h>

Namespace msclr

See Also

Reference

auto_gcroot::operator=

auto_gcroot::release

Other Resources

auto_gcroot Members

auto_gcroot::get

Gets the contained object.

```
_element_type get() const;
```

Return Value

The contained object.

Example

```
// msl_auto_gcroot_get.cpp
// compile with: /clr
#include <msclr\auto_gcroot.h>

using namespace System;
using namespace msclr;

ref class ClassA {
   String^ m_s;
public:
   ClassA( String^ s ) : m_s( s ){
      Console::WriteLine( "in ClassA constructor:" + m_s );
   }
   ~ClassA() {
      Console::WriteLine( "in ClassA destructor:" + m_s );
   }

   void PrintHello() {
      Console::WriteLine( "Hello from {0} A!", m_s );
   }
};

void PrintA( ClassA^ a ) {
   a->PrintHello();
}

int main() {
   auto_gcroot<ClassA^> a = gcnew ClassA( "first" );
   a->PrintHello();

   ClassA^ a2 = a.get();
   a2->PrintHello();

   PrintA( a.get() );
}
```

Output

```
in ClassA constructor:first
Hello from first A!
Hello from first A!
Hello from first A!
in ClassA destructor:first
```

Requirements

Header file <msclr\auto_gcroot.h>

Namespace msclr

See Also

Other Resources

auto_gcroot Members

auto_gcroot::release

Releases the object from **auto_gcroot** management.

```
_element_type release();
```

Return Value

The released object.

Example

```
// msl_auto_gcroot_release.cpp
// compile with: /clr
#include <msclr\auto_gcroot.h>

using namespace System;
using namespace msclr;

ref class ClassA {
   String^ m_s;
public:
   ClassA( String^ s ) : m_s( s ) {
      Console::WriteLine( "ClassA constructor: " + m_s );
   }
   ~ClassA() {
      Console::WriteLine( "ClassA destructor: " + m_s );
   }

   void PrintHello() {
      Console::WriteLine( "Hello from {0} A!", m_s );
   }
};

int main()
{
   ClassA^ a;

   // create a new scope:
   {
      auto_gcroot<ClassA^> agc1 = gcnew ClassA( "first" );
      auto_gcroot<ClassA^> agc2 = gcnew ClassA( "second" );
      a = agc1.release();
   }
   // agc1 and agc2 go out of scope here

   a->PrintHello();

   Console::WriteLine( "done" );
}
```

Output

```
ClassA constructor: first
ClassA constructor: second
ClassA destructor: second
Hello from first A!
done
```

Requirements

Header file <msclr\auto_gcroot.h>

Namespace msclr

See Also

Reference

auto_gcroot::~auto_gcroot

Other Resources

auto_gcroot Members

auto_gcroot::reset

Destroy the current owned object and optionally take possession of a new object.

```
void reset(
   _element_type _new_ptr = nullptr
);
```

Parameters

_new_ptr

(Optional) The new object.

Example

```
// msl_auto_gcroot_reset.cpp
// compile with: /clr
#include <msclr\auto_gcroot.h>

using namespace System;
using namespace msclr;

ref class ClassA {
   String^ m_s;
public:
   ClassA( String^ s ) : m_s( s ) {
      Console::WriteLine( "ClassA constructor: " + m_s );
   }
   ~ClassA() {
      Console::WriteLine( "ClassA destructor: " + m_s );
   }

   void PrintHello() {
      Console::WriteLine( "Hello from {0} A!", m_s );
   }
};

int main()
{
   auto_gcroot<ClassA^> agc1 = gcnew ClassA( "first" );
   agc1->PrintHello();

   ClassA^ ha = gcnew ClassA( "second" );
   agc1.reset( ha ); // release first object, reference second
   agc1->PrintHello();

   agc1.reset(); // release second object, set to nullptr

   Console::WriteLine( "done" );
}
```

Output

```
ClassA constructor: first
Hello from first A!
ClassA constructor: second
ClassA destructor: first
Hello from second A!
ClassA destructor: second
done
```

Requirements

Header file <msclr\auto_gcroot.h>

Namespace msclr

See Also

Reference

auto_gcroot::release

auto_gcroot::attach

Other Resources

auto_gcroot Members

auto_gcroot::swap

Swaps objects with another **auto_gcroot**.

```
void swap(
   auto_gcroot<_element_type> & _right
);
```

Parameters

_right

The **auto_gcroot** with which to swap objects.

Example

```
// msl_auto_gcroot_swap.cpp
// compile with: /clr
#include <msclr\auto_gcroot.h>

using namespace System;
using namespace msclr;

int main() {
   auto_gcroot<String^> s1 = "string one";
   auto_gcroot<String^> s2 = "string two";

   Console::WriteLine( "s1 = '{0}', s2 = '{1}'",
      s1->ToString(), s2->ToString() );
   s1.swap( s2 );
   Console::WriteLine( "s1 = '{0}', s2 = '{1}'",
      s1->ToString(), s2->ToString() );
}
```

Output

```
s1 = 'string one', s2 = 'string two'
s1 = 'string two', s2 = 'string one'
```

Requirements

Header file <msclr\auto_gcroot.h>

Namespace msclr

See Also

Reference

swap Function (auto_gcroot)

Other Resources

auto_gcroot Members

auto_gcroot::operator=

Assignment operator.

```
auto_gcroot<_element_type> & operator=(
   _element_type _right
);
auto_gcroot<_element_type> & operator=(
   auto_gcroot<_element_type> & _right
);
template<typename _other_type>
auto_gcroot<_element_type> & operator=(
   auto_gcroot<_other_type> & _right
);
```

Parameters

_right

The object or **auto_gcroot** to be assigned to the current **auto_gcroot**.

Return Value

The current **auto_gcroot**, now owning *_right*.

Example

```
// msl_auto_gcroot_operator_equals.cpp
// compile with: /clr
#include <msclr\auto_gcroot.h>

using namespace System;
using namespace msclr;

ref class ClassA {
protected:
   String^ m_s;
public:
   ClassA(String^ s) : m_s(s) {
      Console::WriteLine( "in ClassA constructor: " + m_s );
   }
   ~ClassA() {
      Console::WriteLine( "in ClassA destructor: " + m_s );
   }

   virtual void PrintHello() {
      Console::WriteLine( "Hello from {0} A!", m_s );
   }
};

ref class ClassB : ClassA {
public:
   ClassB( String^ s ) : ClassA( s ) {}
   virtual void PrintHello() new {
      Console::WriteLine( "Hello from {0} B!", m_s );
   }
};

int main()
{
   auto_gcroot<ClassA^> a;
   auto_gcroot<ClassA^> a2(gcnew ClassA( "first" ) );
   a = a2; // assign from same type
   a->PrintHello();

   ClassA^ ha = gcnew ClassA( "second" );
```

```
    a = ha; // assign from raw handle

    auto_gcroot<ClassB^> b(gcnew ClassB( "third" ) );
    b->PrintHello();
    a = b; // assign from derived type
    a->PrintHello();

    Console::WriteLine("done");
  }
```

Output

```
in ClassA constructor: first
Hello from first A!
in ClassA constructor: second
in ClassA destructor: first
in ClassA constructor: third
Hello from third B!
in ClassA destructor: second
Hello from third A!
done
in ClassA destructor: third
```

Requirements

Header file <msclr\auto_gcroot.h>

Namespace msclr

See Also

Reference

auto_gcroot::attach

Other Resources

auto_gcroot Members

auto_gcroot::operator->

The member access operator.

```
_element_type operator->() const;
```

Return Value

The object that is wrapped by **auto_gcroot**.

Example

```
// msl_auto_gcroot_op_arrow.cpp
// compile with: /clr
#include <msclr\auto_gcroot.h>

using namespace System;
using namespace msclr;

ref class ClassA {
protected:
   String^ m_s;
public:
   ClassA( String^ s ) : m_s( s ) {}

   virtual void PrintHello() {
      Console::WriteLine( "Hello from {0} A!", m_s );
   }

   int m_i;
};

int main() {
   auto_gcroot<ClassA^> a( gcnew ClassA( "first" ) );
   a->PrintHello();

   a->m_i = 5;
   Console::WriteLine( "a->m_i = {0}", a->m_i );
}
```

Output

```
Hello from first A!
a->m_i = 5
```

Requirements

Header file <msclr\auto_gcroot.h>

Namespace msclr

See Also

Reference

auto_gcroot::get

Other Resources

auto_gcroot Members

auto_gcroot::operator!

Operator for using **auto_gcroot** in a conditional expression.

```
bool operator!() const;
```

Return Value

true if the wrapped object is invalid; **false** otherwise.

Example

```
// msl_auto_gcroot_operator_not.cpp
// compile with: /clr
#include <msclr\auto_gcroot.h>

using namespace System;
using namespace msclr;

int main() {
   auto_gcroot<String^> s;
   if ( s ) Console::WriteLine( "s is valid" );
   if ( !s ) Console::WriteLine( "s is invalid" );
   s = "something";
   if ( s ) Console::WriteLine( "now s is valid" );
   if ( !s ) Console::WriteLine( "now s is invalid" );
   s.reset();
   if ( s ) Console::WriteLine( "now s is valid" );
   if ( !s ) Console::WriteLine( "now s is invalid" );
}
```

Output

```
s is invalid
now s is valid
now s is invalid
```

Requirements

Header file <msclr\auto_gcroot.h>

Namespace msclr

See Also

Reference

auto_gcroot::operator bool

Other Resources

auto_gcroot Members

auto_gcroot::operator auto_gcroot

Type-cast operator between **auto_gcroot** and compatible types.

```
template<typename _other_type>
operator auto_gcroot<_other_type>();
```

Return Value

The current **auto_gcroot** cast to **auto_gcroot<_other_type>**.

Example

```
// msl_auto_gcroot_op_auto_gcroot.cpp
// compile with: /clr
#include <msclr\auto_gcroot.h>

using namespace System;
using namespace msclr;

ref class ClassA {
protected:
   String^ m_s;
public:
   ClassA( String^ s ) : m_s( s ) {}

   virtual void PrintHello() {
      Console::WriteLine( "Hello from {0} A!", m_s );
   }
};

ref class ClassB : ClassA {
public:
   ClassB( String ^ s) : ClassA( s ) {}
   virtual void PrintHello() new {
      Console::WriteLine( "Hello from {0} B!", m_s );
   }
};

int main() {
   auto_gcroot<ClassB^> b = gcnew ClassB("first");
   b->PrintHello();
   auto_gcroot<ClassA^> a = (auto_gcroot<ClassA^>)b;
   a->PrintHello();
}
```

Output

```
Hello from first B!
Hello from first A!
```

Requirements

Header file <msclr\auto_gcroot.h>

Namespace msclr

See Also

Other Resources

auto_gcroot Members

auto_gcroot::operator bool

Operator for using **auto_gcroot** in a conditional expression.

```
operator bool() const;
```

Return Value

true if the wrapped object is valid; **false** otherwise.

Remarks

This operator actually converts to **_detail_class::_safe_bool** which is safer than **bool** because it cannot be converted to an integral type.

Example

```
// msl_auto_gcroot_operator_bool.cpp
// compile with: /clr
#include <msclr\auto_gcroot.h>

using namespace System;
using namespace msclr;

int main() {
   auto_gcroot<String^> s;
   if ( s ) Console::WriteLine( "s is valid" );
   if ( !s ) Console::WriteLine( "s is invalid" );
   s = "something";
   if ( s ) Console::WriteLine( "now s is valid" );
   if ( !s ) Console::WriteLine( "now s is invalid" );
   s.reset();
   if ( s ) Console::WriteLine( "now s is valid" );
   if ( !s ) Console::WriteLine( "now s is invalid" );
}
```

Output

```
s is invalid
now s is valid
now s is invalid
```

Requirements

Header file <msclr\auto_gcroot.h>

Namespace msclr

See Also

Reference

auto_gcroot::operator!

Other Resources

auto_gcroot Members

swap Function (auto_gcroot)

Swaps objects between one **auto_gcroot** and another.

```
template<typename _element_type>
void swap(
   auto_gcroot<_element_type> & _left,
   auto_gcroot<_element_type> & _right
);
```

Parameters

_left

An **auto_gcroot**.

_right

Another **auto_gcroot**.

Example

```
// msl_swap_auto_gcroot.cpp
// compile with: /clr
#include <msclr\auto_gcroot.h>

using namespace System;
using namespace msclr;

int main() {
   auto_gcroot<String^> s1 = "string one";
   auto_gcroot<String^> s2 = "string two";

   Console::WriteLine( "s1 = '{0}', s2 = '{1}'",
      s1->ToString(), s2->ToString() );
   swap( s1, s2 );
   Console::WriteLine( "s1 = '{0}', s2 = '{1}'",
      s1->ToString(), s2->ToString() );
}
```

Output

```
s1 = 'string one', s2 = 'string two'
s1 = 'string two', s2 = 'string one'
```

Requirements

Header file <msclr\auto_gcroot.h>

Namespace msclr

See Also

Reference

auto_gcroot

auto_gcroot::swap

auto_handle

Defines the **auto_handle** class and **swap** function.

```
#include <msclr\auto_handle.h>
```

Remarks

In this header file:

auto_handle Class

swap Function (auto_handle)

See Also

Other Resources

C++ Support Library

auto_handle Class

Automatic resource management which can be used to embed a virtual handle into a managed type.

```
template<typename _element_type>
ref class auto_handle;
```

Parameters

_element_type

The managed type to be embedded.

Requirements

Header file <msclr\auto_handle.h>

Namespace msclr

See Also

Reference

auto_handle

auto_gcroot Class

Other Resources

auto_handle Members

auto_handle Members

Members of **auto_handle**.

In This Section

auto_handle::auto_handle

auto_handle::~auto_handle

auto_handle::get

auto_handle::release

auto_handle::reset

auto_handle::swap

auto_handle::operator=

auto_handle::operator->

auto_handle::operator!

auto_handle::operator auto_handle

auto_handle::operator bool

See Also

Reference

auto_handle Class

auto_handle

auto_handle::auto_handle

The **auto_handle** constructor.

```
auto_handle();
auto_handle(
   _element_type ^ _ptr
);
auto_handle(
   auto_handle<_element_type> % _right
);
template<typename _other_type>
auto_handle(
   auto_handle<_other_type> % _right
);
```

Parameters

_ptr

The object to own.

_right

An existing **auto_handle**.

Example

```
// msl_auto_handle_auto_handle.cpp
// compile with: /clr
#include "msclr\auto_handle.h"

using namespace System;
using namespace msclr;
ref class RefClassA {
protected:
   String^ m_s;
public:
   RefClassA(String^ s) : m_s(s) {
      Console::WriteLine( "in RefClassA constructor: " + m_s );
   }
   ~RefClassA() {
      Console::WriteLine( "in RefClassA destructor: " + m_s );
   }

   virtual void PrintHello() {
      Console::WriteLine( "Hello from {0} A!", m_s );
   }
};

ref class RefClassB : RefClassA {
public:
   RefClassB( String^ s ) : RefClassA( s ) {}
   virtual void PrintHello() new {
      Console::WriteLine( "Hello from {0} B!", m_s );
   }
};

int main()
{
   {
      auto_handle<RefClassA> a(gcnew RefClassA( "first" ) );
      a->PrintHello();
   }

   {
```

```
        auto_handle<RefClassB> b(gcnew RefClassB( "second" ) );
        b->PrintHello();
        auto_handle<RefClassA> a(b); //construct from derived type
        a->PrintHello();
        auto_handle<RefClassA> a2(a); //construct from same type
        a2->PrintHello();
    }

    Console::WriteLine("done");
  }
```

Output

```
in RefClassA constructor: first
Hello from first A!
in RefClassA destructor: first
in RefClassA constructor: second
Hello from second B!
Hello from second A!
Hello from second A!
in RefClassA destructor: second
done
```

Requirements

Header file <msclr\auto_handle.h>

Namespace msclr

See Also

Reference

auto_handle::operator=

auto_handle::~auto_handle

Other Resources

auto_handle Members

auto_handle::~auto_handle

The **auto_handle** destructor.

```
~auto_handle();
```

Remarks

The destructor also destructs the owned object.

Example

```
// msl_auto_handle_dtor.cpp
// compile with: /clr
#include "msclr\auto_handle.h"

using namespace System;
using namespace msclr;

ref class ClassA {
public:
   ClassA() { Console::WriteLine( "ClassA constructor" ); }
   ~ClassA() { Console::WriteLine( "ClassA destructor" ); }
};

int main()
{
   // create a new scope for a:
   {
      auto_handle<ClassA> a = gcnew ClassA;
   }
   // a goes out of scope here, invoking its destructor
   // which in turns destructs the ClassA object.

   Console::WriteLine( "done" );
}
```

Output

```
ClassA constructor
ClassA destructor
done
```

Requirements

Header file <msclr\auto_handle.h>

Namespace msclr

See Also

Reference

auto_handle::release

auto_handle::auto_handle

Other Resources

auto_handle Members

auto_handle::get

Gets the contained object.

```
_element_type ^ get();
```

Return Value

The contained object.

Example

```
// msl_auto_handle_get.cpp
// compile with: /clr
#include "msclr\auto_handle.h"

using namespace System;
using namespace msclr;

ref class ClassA {
   String^ m_s;
public:
   ClassA( String^ s ) : m_s( s ){
      Console::WriteLine( "in ClassA constructor:" + m_s );
   }
   ~ClassA() {
      Console::WriteLine( "in ClassA destructor:" + m_s );
   }

   void PrintHello() {
      Console::WriteLine( "Hello from {0} A!", m_s );
   }
};

void PrintA( ClassA^ a ) {
   a->PrintHello();
}

int main() {
   auto_handle<ClassA> a = gcnew ClassA( "first" );
   a->PrintHello();

   ClassA^ a2 = a.get();
   a2->PrintHello();

   PrintA( a.get() );
}
```

Output

```
in ClassA constructor:first
Hello from first A!
Hello from first A!
Hello from first A!
in ClassA destructor:first
```

Requirements

Header file <msclr\auto_handle.h>

Namespace msclr

See Also

Other Resources

auto_handle Members

auto_handle::release

Releases the object from **auto_handle** management.

```
_element_type ^ release();
```

Return Value

The released object.

Example

```
// msl_auto_handle_release.cpp
// compile with: /clr
#include <msclr\auto_handle.h>

using namespace System;
using namespace msclr;

ref class ClassA {
   String^ m_s;
public:
   ClassA( String^ s ) : m_s( s ) {
      Console::WriteLine( "ClassA constructor: " + m_s );
   }
   ~ClassA() {
      Console::WriteLine( "ClassA destructor: " + m_s );
   }

   void PrintHello() {
      Console::WriteLine( "Hello from {0} A!", m_s );
   }
};

int main()
{
   ClassA^ a;

   // create a new scope:
   {
      auto_handle<ClassA> agc1 = gcnew ClassA( "first" );
      auto_handle<ClassA> agc2 = gcnew ClassA( "second" );
      a = agc1.release();
   }
   // agc1 and agc2 go out of scope here

   a->PrintHello();

   Console::WriteLine( "done" );
}
```

Output

```
ClassA constructor: first
ClassA constructor: second
ClassA destructor: second
Hello from first A!
done
```

Requirements

Header file <msclr\auto_handle.h>

Namespace msclr

See Also

Reference

auto_handle::~auto_handle

auto_handle::reset

Other Resources

auto_handle Members

auto_handle::reset

Destroy the current owned object and optionally take possession of a new object.

```
void reset(
   _element_type ^ _new_ptr
);
void reset();
```

Parameters

_new_ptr

(Optional) The new object.

Example

```
// msl_auto_handle_reset.cpp
// compile with: /clr
#include <msclr\auto_handle.h>

using namespace System;
using namespace msclr;

ref class ClassA {
   String^ m_s;
public:
   ClassA( String^ s ) : m_s( s ) {
      Console::WriteLine( "ClassA constructor: " + m_s );
   }
   ~ClassA() {
      Console::WriteLine( "ClassA destructor: " + m_s );
   }

   void PrintHello() {
      Console::WriteLine( "Hello from {0} A!", m_s );
   }
};

int main()
{
   auto_handle<ClassA> agc1 = gcnew ClassA( "first" );
   agc1->PrintHello();

   ClassA^ ha = gcnew ClassA( "second" );
   agc1.reset( ha ); // release first object, reference second
   agc1->PrintHello();

   agc1.reset(); // release second object, set to nullptr

   Console::WriteLine( "done" );
}
```

Output

```
ClassA constructor: first
Hello from first A!
ClassA constructor: second
ClassA destructor: first
Hello from second A!
ClassA destructor: second
done
```

Requirements

Header file <msclr\auto_handle.h>

Namespace msclr

See Also

Reference

auto_handle::release

Other Resources

auto_handle Members

auto_handle::swap

Swaps objects with another **auto_handle**.

```
void swap(
   auto_handle<_element_type> % _right
);
```

Parameters

_right

The **auto_handle** with which to swap objects.

Example

```
// msl_auto_handle_swap.cpp
// compile with: /clr
#include <msclr\auto_handle.h>

using namespace System;
using namespace msclr;

int main() {
   auto_handle<String> s1 = "string one";
   auto_handle<String> s2 = "string two";

   Console::WriteLine( "s1 = '{0}', s2 = '{1}'",
      s1->ToString(), s2->ToString() );
   s1.swap( s2 );
   Console::WriteLine( "s1 = '{0}', s2 = '{1}'",
      s1->ToString(), s2->ToString() );
}
```

Output

```
s1 = 'string one', s2 = 'string two'
s1 = 'string two', s2 = 'string one'
```

Requirements

Header file <msclr\auto_handle.h>

Namespace msclr

See Also

Reference

swap Function (auto_handle)

Other Resources

auto_handle Members

auto_handle::operator=

Assignment operator.

```
auto_handle<_element_type> % operator=(
   auto_handle<_element_type> % _right
);
template<typename _other_type>
auto_handle<_element_type> % operator=(
   auto_handle<_other_type> % _right
);
```

Parameters

_right

The **auto_handle** to be assigned to the current **auto_handle**.

Return Value

The current **auto_handle**, now owning *_right*.

Example

```
// msl_auto_handle_op_assign.cpp
// compile with: /clr
#include <msclr\auto_handle.h>

using namespace System;
using namespace msclr;

ref class ClassA {
protected:
   String^ m_s;
public:
   ClassA(String^ s) : m_s(s) {
      Console::WriteLine( "in ClassA constructor: " + m_s );
   }
   ~ClassA() {
      Console::WriteLine( "in ClassA destructor: " + m_s );
   }

   virtual void PrintHello() {
      Console::WriteLine( "Hello from {0} A!", m_s );
   }
};

ref class ClassB : ClassA {
public:
   ClassB( String^ s ) : ClassA( s ) {}
   virtual void PrintHello() new {
      Console::WriteLine( "Hello from {0} B!", m_s );
   }
};

int main()
{
   auto_handle<ClassA> a;
   auto_handle<ClassA> a2(gcnew ClassA( "first" ) );
   a = a2; // assign from same type
   a->PrintHello();

   auto_handle<ClassB> b(gcnew ClassB( "second" ) );
   b->PrintHello();
   a = b; // assign from derived type
   a->PrintHello();
```

```
    Console::WriteLine("done");
  }
```

Output

```
in ClassA constructor: first
Hello from first A!
in ClassA constructor: second
Hello from second B!
in ClassA destructor: first
Hello from second A!
done
in ClassA destructor: second
```

Requirements

Header file <msclr\auto_handle.h>

Namespace msclr

See Also

Other Resources

auto_handle Members

auto_handle::operator->

The member access operator.

```
_element_type ^ operator->();
```

Return Value

The object that is wrapped by **auto_handle**.

Example

```
// msl_auto_handle_op_arrow.cpp
// compile with: /clr
#include <msclr\auto_handle.h>

using namespace System;
using namespace msclr;

ref class ClassA {
protected:
   String^ m_s;
public:
   ClassA( String^ s ) : m_s( s ) {}

   virtual void PrintHello() {
      Console::WriteLine( "Hello from {0} A!", m_s );
   }

   int m_i;
};

int main() {
   auto_handle<ClassA> a( gcnew ClassA( "first" ) );
   a->PrintHello();

   a->m_i = 5;
   Console::WriteLine( "a->m_i = {0}", a->m_i );
}
```

Output

```
Hello from first A!
a->m_i = 5
```

Requirements

Header file <msclr\auto_handle.h>

Namespace msclr

See Also

Reference

auto_handle::get

Other Resources

auto_handle Members

auto_handle::operator!

Operator for using **auto_handle** in a conditional expression.

```
bool operator!();
```

Return Value

true if the wrapped object is invalid; **false** otherwise.

Example

```
// msl_auto_handle_operator_not.cpp
// compile with: /clr
#include <msclr\auto_handle.h>

using namespace System;
using namespace msclr;

int main() {
   auto_handle<String> s1;
   auto_handle<String> s2 = "something";
   if ( s1) Console::WriteLine( "s1 is valid" );
   if ( !s1 ) Console::WriteLine( "s1 is invalid" );
   if ( s2 ) Console::WriteLine( "s2 is valid" );
   if ( !s2 ) Console::WriteLine( "s2 is invalid" );
   s2.reset();
   if ( s2 ) Console::WriteLine( "s2 is now valid" );
   if ( !s2 ) Console::WriteLine( "s2 is now invalid" );
}
```

Output

```
s1 is invalid
s2 is valid
s2 is now invalid
```

Requirements

Header file <msclr\auto_handle.h>

Namespace msclr

See Also

Reference

auto_handle::operator bool

Other Resources

auto_handle Members

auto_handle::operator auto_handle

Type-cast operator between **auto_handle** and compatible types.

```
template<typename _other_type>
operator auto_handle<_other_type>();
```

Return Value

The current **auto_handle** cast to **auto_handle<_other_type>**.

Example

```
// msl_auto_handle_op_auto_handle.cpp
// compile with: /clr
#include <msclr\auto_handle.h>

using namespace System;
using namespace msclr;

ref class ClassA {
protected:
   String^ m_s;
public:
   ClassA( String^ s ) : m_s( s ) {}

   virtual void PrintHello() {
      Console::WriteLine( "Hello from {0} A!", m_s );
   }
};

ref class ClassB : ClassA {
public:
   ClassB( String ^ s) : ClassA( s ) {}
   virtual void PrintHello() new {
      Console::WriteLine( "Hello from {0} B!", m_s );
   }
};

int main() {
   auto_handle<ClassB> b = gcnew ClassB("first");
   b->PrintHello();
   auto_handle<ClassA> a = (auto_handle<ClassA>)b;
   a->PrintHello();
}
```

Output

```
Hello from first B!
Hello from first A!
```

Requirements

Header file <msclr\auto_handle.h>

Namespace msclr

See Also

Other Resources

auto_handle Members

auto_handle::operator bool

Operator for using **auto_handle** in a conditional expression.

```
operator bool();
```

Return Value

true if the wrapped object is valid; **false** otherwise.

Remarks

This operator actually converts to **_detail_class::_safe_bool** which is safer than **bool** because it cannot be converted to an integral type.

Example

```
// msl_auto_handle_operator_bool.cpp
// compile with: /clr
#include <msclr\auto_handle.h>

using namespace System;
using namespace msclr;

int main() {
   auto_handle<String> s1;
   auto_handle<String> s2 = "hi";
   if ( s1 ) Console::WriteLine( "s1 is valid" );
   if ( !s1 ) Console::WriteLine( "s1 is invalid" );
   if ( s2 ) Console::WriteLine( "s2 is valid" );
   if ( !s2 ) Console::WriteLine( "s2 is invalid" );
   s2.reset();
   if ( s2 ) Console::WriteLine( "s2 is now valid" );
   if ( !s2 ) Console::WriteLine( "s2 is now invalid" );
}
```

Output

```
s1 is invalid
s2 is valid
s2 is now invalid
```

Requirements

Header file <msclr\auto_handle.h>

Namespace msclr

See Also

Reference

auto_handle::operator!

Other Resources

auto_handle Members

swap Function (auto_handle)

Swaps objects between one **auto_handle** and another.

```
template<typename _element_type>
void swap(
   auto_handle<_element_type> % _left,
   auto_handle<_element_type> % _right
);
```

Parameters

_left

An **auto_handle**.

_right

Another **auto_handle**.

Example

```
// msl_swap_auto_handle.cpp
// compile with: /clr
#include <msclr\auto_handle.h>

using namespace System;
using namespace msclr;

int main() {
   auto_handle<String> s1 = "string one";
   auto_handle<String> s2 = "string two";

   Console::WriteLine( "s1 = '{0}', s2 = '{1}'",
      s1->ToString(), s2->ToString() );
   swap( s1, s2 );
   Console::WriteLine( "s1 = '{0}', s2 = '{1}'",
      s1->ToString(), s2->ToString() );
}
```

Output

```
s1 = 'string one', s2 = 'string two'
s1 = 'string two', s2 = 'string one'
```

Requirements

Header file <msclr\auto_handle.h>

Namespace msclr

See Also

Reference

auto_handle

auto_handle::swap

Synchronization (lock Class)

Provides a mechanism to automate taking a lock that synchronizes access to an object.

In This Section

lock

See Also

Other Resources

C++ Support Library

lock

Defines the **lock** class that automates synchronization of access to an object.

```
#include <msclr\lock.h>
```

Remarks

In this header file:

lock Class

lock_when Enum

See Also

Other Resources

C++ Support Library

lock Class

This class automates taking a lock for synchronizing access to an object from multiple threads. When constructed it acquires the lock and when destroyed it releases the lock.

```
ref class lock;
```

Remarks

lock is available only for CLR objects and can only be used in CLR code.

Internally, the lock class uses Monitor to synchronize access. See this topic for more detailed information on synchronization.

Requirements

Header file <msclr\lock.h>

Namespace msclr

See Also

Reference

lock

Other Resources

lock Members

lock Members

Members of the lock Class.

In This Section

lock::lock

lock::~lock

lock::acquire

lock::is_locked

lock::release

lock::try_acquire

lock::operator bool

lock::operator==

lock::operator!=

lock::lock

Constructs a **lock** object, optionally waiting to acquire the lock forever, for a specified amount of time, or not at all.

```
template<class T> lock(
   T ^ _object
);
template<class T> lock(
   T ^ _object,
   int _timeout
);
template<class T> lock(
   T ^ _object,
   System::TimeSpan _timeout
);
template<class T> lock(
   T ^ _object,
   lock_later
);
```

Parameters

_object

The object to be locked.

_timeout

Timeout value in milliseconds or as a TimeSpan.

Exceptions

Throws ApplicationException if lock acquisition does not occur before timeout.

Remarks

The first three forms of the constructor attempt to acquire a lock on *_object* within the specified timeout period (or Infinite if none is specified).

The fourth form of the constructor does not acquire a lock on *_object*. **lock_later** is a member of the lock_when Enum. Use lock::acquire or lock::try_acquire to acquire the lock in this case.

The lock will automatically be released when the destructor is called.

_object cannot be ReaderWriterLock. If it is, a compiler error will result.

Example

This example uses a single instance of a class across multiple threads. The class uses a lock on itself to ensure that accesses to its internal data are consistent for each thread. The main application thread uses a lock on the same instance of the class to periodically check to see if any worker threads still exist, and waits to exit until all worker threads have completed their tasks.

```
// msl_lock_lock.cpp
// compile with: /clr
#include <msclr/lock.h>

using namespace System;
using namespace System::Threading;
using namespace msclr;

ref class CounterClass {
private:
   int Counter;

public:
   property int ThreadCount;
```

```
   // function called by multiple threads, use lock to keep Counter consistent
   // for each thread
   void UseCounter() {
      try {
         lock l(this); // wait infinitely

         Console::WriteLine("In thread {0}, Counter = {1}", Thread::CurrentThread->ManagedT
hreadId,
            Counter);

         for (int i = 0; i < 10; i++) {
            Counter++;
            Thread::Sleep(10);
         }

         Console::WriteLine("In thread {0}, Counter = {1}", Thread::CurrentThread->ManagedT
hreadId,
            Counter);

         Counter = 0;
         // lock is automatically released when it goes out of scope and its destructor is
called
      }
      catch (...) {
         Console::WriteLine("Couldn't acquire lock!");
      }

      ThreadCount--;
   }
};

int main() {
   // create a few threads to contend for access to the shared data
   CounterClass^ cc = gcnew CounterClass;
   array<Thread^>^ tarr = gcnew array<Thread^>(5);
   ThreadStart^ startDelegate = gcnew ThreadStart(cc, &CounterClass::UseCounter);
   for (int i = 0; i < tarr->Length; i++) {
      tarr[i] = gcnew Thread(startDelegate);
      cc->ThreadCount++;
      tarr[i]->Start();
   }

   // keep our main thread alive until all worker threads have completed
   lock l(cc, lock_later); // don't lock now, just create the object
   while (true) {
      if (l.try_acquire(50)) { // try to acquire lock, don't throw an exception if can't
         if (0 == cc->ThreadCount) {
            Console::WriteLine("All threads completed.");
            break; // all threads are gone, exit while
         }
         else {
            Console::WriteLine("{0} threads exist, continue waiting...", cc->ThreadCount);
            l.release(); // some threads exist, let them do their work
         }
      }
   }
}
```

Sample Output

```
In thread 3, Counter = 0
In thread 3, Counter = 10
In thread 5, Counter = 0
In thread 5, Counter = 10
In thread 7, Counter = 0
In thread 7, Counter = 10
In thread 4, Counter = 0
In thread 4, Counter = 10
```

```
In thread 6, Counter = 0
In thread 6, Counter = 10
All threads completed.
```

Requirements

Header file <msclr\lock.h>

Namespace msclr

See Also

Reference

lock::~lock

lock::acquire

lock::try_acquire

Other Resources

lock Members

lock::~lock

Destructs a **lock** object.

```
~lock();
```

Remarks

The destructor calls lock::release.

Example

This example uses a single instance of a class across multiple threads. The class uses a lock on itself to ensure that accesses to its internal data are consistent for each thread. The main application thread uses a lock on the same instance of the class to periodically check to see if any worker threads still exist, and waits to exit until all worker threads have completed their tasks.

```
// msl_lock_dtor.cpp
// compile with: /clr
#include <msclr/lock.h>

using namespace System;
using namespace System::Threading;
using namespace msclr;

ref class CounterClass {
private:
   int Counter;

public:
   property int ThreadCount;

   // function called by multiple threads, use lock to keep Counter consistent
   // for each thread
   void UseCounter() {
      try {
         lock l(this); // wait infinitely

         Console::WriteLine("In thread {0}, Counter = {1}", Thread::CurrentThread->ManagedT
hreadId,
            Counter);

         for (int i = 0; i < 10; i++) {
            Counter++;
            Thread::Sleep(10);
         }

         Console::WriteLine("In thread {0}, Counter = {1}", Thread::CurrentThread->ManagedT
hreadId,
            Counter);

         Counter = 0;
         // lock is automatically released when it goes out of scope and its destructor is
called
      }
      catch (...) {
         Console::WriteLine("Couldn't acquire lock!");
      }

      ThreadCount--;
   }
};

int main() {
   // create a few threads to contend for access to the shared data
```

```
    CounterClass^ cc = gcnew CounterClass;
    array<Thread^>^ tarr = gcnew array<Thread^>(5);
    ThreadStart^ startDelegate = gcnew ThreadStart(cc, &CounterClass::UseCounter);
    for (int i = 0; i < tarr->Length; i++) {
       tarr[i] = gcnew Thread(startDelegate);
       cc->ThreadCount++;
       tarr[i]->Start();
    }

    // keep our main thread alive until all worker threads have completed
    lock l(cc, lock_later); // don't lock now, just create the object
    while (true) {
       if (l.try_acquire(50)) { // try to acquire lock, don't throw an exception if can't
          if (0 == cc->ThreadCount) {
             Console::WriteLine("All threads completed.");
             break; // all threads are gone, exit while
          }
          else {
             Console::WriteLine("{0} threads exist, continue waiting...", cc->ThreadCount);
             l.release(); // some threads exist, let them do their work
          }
       }
    }
 }
```

Sample Output

```
In thread 3, Counter = 0
In thread 3, Counter = 10
In thread 5, Counter = 0
In thread 5, Counter = 10
In thread 7, Counter = 0
In thread 7, Counter = 10
In thread 4, Counter = 0
In thread 4, Counter = 10
In thread 6, Counter = 0
In thread 6, Counter = 10
All threads completed.
```

Requirements

Header file <msclr\lock.h>

Namespace msclr

See Also

Reference

lock::lock

Other Resources

lock Members

lock::is_locked

Indicates whether a lock is being held.

```
bool is_locked();
```

Return Value

true if a lock is held, **false** otherwise.

Example

This example uses a single instance of a class across multiple threads. The class uses a lock on itself to ensure that accesses to its internal data are consistent for each thread. The main application thread uses a lock on the same instance of the class to periodically check to see if any worker threads still exist, and waits to exit until all worker threads have completed their tasks.

```
// msl_lock_is_locked.cpp
// compile with: /clr
#include <msclr/lock.h>

using namespace System;
using namespace System::Threading;
using namespace msclr;

ref class CounterClass {
private:
   int Counter;

public:
   property int ThreadCount;

   // function called by multiple threads, use lock to keep Counter consistent
   // for each thread
   void UseCounter() {
      try {
         lock l(this); // wait infinitely

         Console::WriteLine("In thread {0}, Counter = {1}", Thread::CurrentThread->ManagedT
hreadId,
            Counter);

         for (int i = 0; i < 10; i++) {
            Counter++;
            Thread::Sleep(10);
         }

         Console::WriteLine("In thread {0}, Counter = {1}", Thread::CurrentThread->ManagedT
hreadId,
            Counter);

         Counter = 0;
         // lock is automatically released when it goes out of scope and its destructor is
called
      }
      catch (...) {
         Console::WriteLine("Couldn't acquire lock!");
      }

      ThreadCount--;
   }
};

int main() {
   // create a few threads to contend for access to the shared data
```

```
    CounterClass^ cc = gcnew CounterClass;
    array<Thread^>^ tarr = gcnew array<Thread^>(5);
    ThreadStart^ startDelegate = gcnew ThreadStart(cc, &CounterClass::UseCounter);
    for (int i = 0; i < tarr->Length; i++) {
       tarr[i] = gcnew Thread(startDelegate);
       cc->ThreadCount++;
       tarr[i]->Start();
    }

    // keep our main thread alive until all worker threads have completed
    lock l(cc, lock_later); // don't lock now, just create the object
    while (true) {
       l.try_acquire(50); // try to acquire lock, don't throw an exception if can't
       if (l.is_locked()) { // check if we got the lock
          if (0 == cc->ThreadCount) {
             Console::WriteLine("All threads completed.");
             break; // all threads are gone, exit while
          }
          else {
             Console::WriteLine("{0} threads exist, continue waiting...", cc->ThreadCount);
             l.release(); // some threads exist, let them do their work
          }
       }
    }
 }
```

Sample Output

```
In thread 3, Counter = 0
In thread 3, Counter = 10
In thread 5, Counter = 0
In thread 5, Counter = 10
In thread 4, Counter = 0
In thread 4, Counter = 10
In thread 7, Counter = 0
In thread 7, Counter = 10
In thread 6, Counter = 0
In thread 6, Counter = 10
All threads completed.
```

Requirements

Header file <msclr\lock.h>

Namespace msclr

See Also

Reference

lock::operator bool

Other Resources

lock Members

lock::operator bool

Operator for using **lock** in a conditional expression.

```
operator bool();
```

Return Value

true if a lock is held, **false** otherwise.

Remarks

This operator actually converts to **_detail_class::_safe_bool** which is safer than **bool** because it cannot be converted to an integral type.

Example

This example uses a single instance of a class across multiple threads. The class uses a lock on itself to ensure that accesses to its internal data are consistent for each thread. The main application thread uses a lock on the same instance of the class to periodically check to see if any worker threads still exist, and waits to exit until all worker threads have completed their tasks.

```
// msl_lock_op_bool.cpp
// compile with: /clr
#include <msclr/lock.h>

using namespace System;
using namespace System::Threading;
using namespace msclr;

ref class CounterClass {
private:
   int Counter;

public:
   property int ThreadCount;

   // function called by multiple threads, use lock to keep Counter consistent
   // for each thread
   void UseCounter() {
      try {
         lock l(this); // wait infinitely

         Console::WriteLine("In thread {0}, Counter = {1}", Thread::CurrentThread->ManagedT
hreadId,
            Counter);

         for (int i = 0; i < 10; i++) {
            Counter++;
            Thread::Sleep(10);
         }

         Console::WriteLine("In thread {0}, Counter = {1}", Thread::CurrentThread->ManagedT
hreadId,
            Counter);

         Counter = 0;
         // lock is automatically released when it goes out of scope and its destructor is
called
      }
      catch (...) {
         Console::WriteLine("Couldn't acquire lock!");
      }

      ThreadCount--;
```

```
    }
};

int main() {
   // create a few threads to contend for access to the shared data
   CounterClass^ cc = gcnew CounterClass;
   array<Thread^>^ tarr = gcnew array<Thread^>(5);
   ThreadStart^ startDelegate = gcnew ThreadStart(cc, &CounterClass::UseCounter);
   for (int i = 0; i < tarr->Length; i++) {
      tarr[i] = gcnew Thread(startDelegate);
      cc->ThreadCount++;
      tarr[i]->Start();
   }

   // keep our main thread alive until all worker threads have completed
   lock l(cc, lock_later); // don't lock now, just create the object
   while (true) {
      l.try_acquire(50); // try to acquire lock, don't throw an exception if can't
      if (l) { // use bool operator to check for lock
         if (0 == cc->ThreadCount) {
            Console::WriteLine("All threads completed.");
            break; // all threads are gone, exit while
         }
         else {
            Console::WriteLine("{0} threads exist, continue waiting...", cc->ThreadCount);
            l.release(); // some threads exist, let them do their work
         }
      }
   }
}
```

Sample Output

```
In thread 3, Counter = 0
In thread 3, Counter = 10
In thread 5, Counter = 0
In thread 5, Counter = 10
In thread 7, Counter = 0
In thread 7, Counter = 10
In thread 4, Counter = 0
In thread 4, Counter = 10
In thread 6, Counter = 0
In thread 6, Counter = 10
All threads completed.
```

Requirements

Header file <msclr\lock.h>

Namespace msclr

See Also

Reference

lock::is_locked

Other Resources

lock Members

lock::acquire

Acquires a lock on an object, optionally waiting to acquire the lock forever, for a specified amount of time, or not at all.

```
void acquire();
void acquire(
   int _timeout
);
void acquire(
   System::TimeSpan _timeout
);
```

Parameters

_timeout

Timeout value in milliseconds or as a TimeSpan.

Exceptions

Throws ApplicationException if lock acquisition does not occur before timeout.

Remarks

If a timeout value is not supplied, the default timeout is Infinite.

If a lock has already been acquired, this function does nothing.

Example

This example uses a single instance of a class across multiple threads. The class uses a lock on itself to ensure that accesses to its internal data are consistent for each thread. The main application thread uses a lock on the same instance of the class to periodically check to see if any worker threads still exist, and waits to exit until all worker threads have completed their tasks.

```
// msl_lock_acquire.cpp
// compile with: /clr
#include <msclr/lock.h>

using namespace System;
using namespace System::Threading;
using namespace msclr;

ref class CounterClass {
private:
   int Counter;

public:
   property int ThreadCount;

   // function called by multiple threads, use lock to keep Counter consistent
   // for each thread
   void UseCounter() {
      try {
         lock l(this); // wait infinitely

         Console::WriteLine("In thread {0}, Counter = {1}", Thread::CurrentThread->ManagedT
hreadId,
            Counter);

         for (int i = 0; i < 10; i++) {
            Counter++;
            Thread::Sleep(10);
         }

         Console::WriteLine("In thread {0}, Counter = {1}", Thread::CurrentThread->ManagedT
hreadId,
```

```
            Counter);

         Counter = 0;
         // lock is automatically released when it goes out of scope and its destructor is
called
      }
      catch (...) {
         Console::WriteLine("Couldn't acquire lock!");
      }

      ThreadCount--;
   }
};

int main() {
   // create a few threads to contend for access to the shared data
   CounterClass^ cc = gcnew CounterClass;
   array<Thread^>^ tarr = gcnew array<Thread^>(5);
   ThreadStart^ startDelegate = gcnew ThreadStart(cc, &CounterClass::UseCounter);
   for (int i = 0; i < tarr->Length; i++) {
      tarr[i] = gcnew Thread(startDelegate);
      cc->ThreadCount++;
      tarr[i]->Start();
   }

   // keep our main thread alive until all worker threads have completed
   lock l(cc, lock_later); // don't lock now, just create the object
   while (true) {
      if (l.try_acquire(50)) { // try to acquire lock, don't throw an exception if can't
         if (0 == cc->ThreadCount) {
            Console::WriteLine("All threads completed.");
            break; // all threads are gone, exit while
         }
         else {
            Console::WriteLine("{0} threads exist, continue waiting...", cc->ThreadCount);
            l.release(); // some threads exist, let them do their work
         }
      }
   }
}
```

Sample Output

```
In thread 3, Counter = 0
In thread 3, Counter = 10
In thread 5, Counter = 0
In thread 5, Counter = 10
In thread 7, Counter = 0
In thread 7, Counter = 10
In thread 4, Counter = 0
In thread 4, Counter = 10
In thread 6, Counter = 0
In thread 6, Counter = 10
All threads completed.
```

Requirements

Header file <msclr\lock.h>

Namespace msclr

See Also

Reference

lock::try_acquire

Other Resources

lock Members

lock::try_acquire

Acquires a lock on an object, waiting for a specified amount of time and returning a **bool** to report the success of acquisition instead of throwing an exception.

```
bool try_acquire(
   int _timeout_ms
);
bool try_acquire(
   System::TimeSpan _timeout
);
```

Parameters

_timeout

Timeout value in milliseconds or as a TimeSpan.

Return Value

true if lock was acquired, **false** otherwise.

Remarks

If a lock has already been acquired, this function does nothing.

Example

This example uses a single instance of a class across multiple threads. The class uses a lock on itself to ensure that accesses to its internal data are consistent for each thread. The main application thread uses a lock on the same instance of the class to periodically check to see if any worker threads still exist, and waits to exit until all worker threads have completed their tasks.

```
// msl_lock_try_acquire.cpp
// compile with: /clr
#include <msclr/lock.h>

using namespace System;
using namespace System::Threading;
using namespace msclr;

ref class CounterClass {
private:
   int Counter;

public:
   property int ThreadCount;

   // function called by multiple threads, use lock to keep Counter consistent
   // for each thread
   void UseCounter() {
      try {
         lock l(this); // wait infinitely

         Console::WriteLine("In thread {0}, Counter = {1}", Thread::CurrentThread->ManagedT
hreadId,
            Counter);

         for (int i = 0; i < 10; i++) {
            Counter++;
            Thread::Sleep(10);
         }

         Console::WriteLine("In thread {0}, Counter = {1}", Thread::CurrentThread->ManagedT
hreadId,
            Counter);
```

```
         Counter = 0;
         // lock is automatically released when it goes out of scope and its destructor is
called
      }
      catch (...) {
         Console::WriteLine("Couldn't acquire lock!");
      }

      ThreadCount--;
   }
};

int main() {
   // create a few threads to contend for access to the shared data
   CounterClass^ cc = gcnew CounterClass;
   array<Thread^>^ tarr = gcnew array<Thread^>(5);
   ThreadStart^ startDelegate = gcnew ThreadStart(cc, &CounterClass::UseCounter);
   for (int i = 0; i < tarr->Length; i++) {
      tarr[i] = gcnew Thread(startDelegate);
      cc->ThreadCount++;
      tarr[i]->Start();
   }

   // keep our main thread alive until all worker threads have completed
   lock l(cc, lock_later); // don't lock now, just create the object
   while (true) {
      if (l.try_acquire(50)) { // try to acquire lock, don't throw an exception if can't
         if (0 == cc->ThreadCount) {
            Console::WriteLine("All threads completed.");
            break; // all threads are gone, exit while
         }
         else {
            Console::WriteLine("{0} threads exist, continue waiting...", cc->ThreadCount);
            l.release(); // some threads exist, let them do their work
         }
      }
   }
}
```

Sample Output

```
In thread 3, Counter = 0
In thread 3, Counter = 10
In thread 5, Counter = 0
In thread 5, Counter = 10
In thread 7, Counter = 0
In thread 7, Counter = 10
In thread 4, Counter = 0
In thread 4, Counter = 10
In thread 6, Counter = 0
In thread 6, Counter = 10
All threads completed.
```

Requirements

Header file <msclr\lock.h>

Namespace msclr

See Also

Reference

lock::acquire

Other Resources

lock Members

lock::release

Releases a lock.

```
void release();
```

Remarks

If no lock is being held, `release` does nothing.

You do not have to call this function explicitly; when a **lock** object goes out of scope, its destructor calls `release`.

Example

This example uses a single instance of a class across multiple threads. The class uses a lock on itself to ensure that accesses to its internal data are consistent for each thread. The main application thread uses a lock on the same instance of the class to periodically check to see if any worker threads still exist, and waits to exit until all worker threads have completed their tasks.

```
// msl_lock_release.cpp
// compile with: /clr
#include <msclr/lock.h>

using namespace System;
using namespace System::Threading;
using namespace msclr;

ref class CounterClass {
private:
   int Counter;

public:
   property int ThreadCount;

   // function called by multiple threads, use lock to keep Counter consistent
   // for each thread
   void UseCounter() {
      try {
         lock l(this); // wait infinitely

         Console::WriteLine("In thread {0}, Counter = {1}", Thread::CurrentThread->ManagedT
hreadId,
            Counter);

         for (int i = 0; i < 10; i++) {
            Counter++;
            Thread::Sleep(10);
         }

         Console::WriteLine("In thread {0}, Counter = {1}", Thread::CurrentThread->ManagedT
hreadId,
            Counter);

         Counter = 0;
         // lock is automatically released when it goes out of scope and its destructor is
called
      }
      catch (...) {
         Console::WriteLine("Couldn't acquire lock!");
      }

      ThreadCount--;
   }
};
```

```
int main() {
   // create a few threads to contend for access to the shared data
   CounterClass^ cc = gcnew CounterClass;
   array<Thread^>^ tarr = gcnew array<Thread^>(5);
   ThreadStart^ startDelegate = gcnew ThreadStart(cc, &CounterClass::UseCounter);
   for (int i = 0; i < tarr->Length; i++) {
      tarr[i] = gcnew Thread(startDelegate);
      cc->ThreadCount++;
      tarr[i]->Start();
   }

   // keep our main thread alive until all worker threads have completed
   lock l(cc, lock_later); // don't lock now, just create the object
   while (true) {
      if (l.try_acquire(50)) { // try to acquire lock, don't throw an exception if can't
         if (0 == cc->ThreadCount) {
            Console::WriteLine("All threads completed.");
            break; // all threads are gone, exit while
         }
         else {
            Console::WriteLine("{0} threads exist, continue waiting...", cc->ThreadCount);
            l.release(); // some threads exist, let them do their work
         }
      }
   }
}
```

Sample Output

```
In thread 3, Counter = 0
In thread 3, Counter = 10
In thread 5, Counter = 0
In thread 5, Counter = 10
In thread 7, Counter = 0
In thread 7, Counter = 10
In thread 4, Counter = 0
In thread 4, Counter = 10
In thread 6, Counter = 0
In thread 6, Counter = 10
All threads completed.
```

Requirements

Header file <msclr\lock.h>

Namespace msclr

See Also

Reference

lock::~lock

Other Resources

lock Members

lock::operator==

Equality operator.

```
template<class T> bool operator==(
   T t
);
```

Parameters

t

The object to compare for equality.

Return Value

Returns **true** if *t* is the same as the lock's object, **false** otherwise.

Example

```
// msl_lock_op_eq.cpp
// compile with: /clr
#include <msclr/lock.h>

using namespace System;
using namespace System::Threading;
using namespace msclr;

int main () {
   Object^ o1 = gcnew Object;
   lock l1(o1);
   if (l1 == o1) {
      Console::WriteLine("Equal!");
   }
}
```

Output

```
Equal!
```

Requirements

Header file <msclr\lock.h>

Namespace msclr

See Also

Reference

lock::operator!=

Other Resources

lock Members

lock::operator!=

Inequality operator.

```
template<class T> bool operator!=(
   T t
);
```

Parameters

t

The object to compare for inequality.

Return Value

Returns **true** if *t* differs from the lock's object, **false** otherwise.

Example

```
// msl_lock_op_ineq.cpp
// compile with: /clr
#include <msclr/lock.h>

using namespace System;
using namespace System::Threading;
using namespace msclr;

int main () {
   Object^ o1 = gcnew Object;
   Object^ o2 = gcnew Object;
   lock l1(o1);
   if (l1 != o2) {
      Console::WriteLine("Inequal!");
   }
}
```

Output

```
Inequal!
```

Requirements

Header file <msclr\lock.h>

Namespace msclr

See Also

Reference

lock::operator==

Other Resources

lock Members

lock_when Enum

Specifies deferred locking.

```
enum lock_when {
   lock_later
};
```

Remarks

When passed to lock::lock, **lock_later** specifies that the lock is not to be taken now.

Example

This example uses a single instance of a class across multiple threads. The class uses a lock on itself to ensure that accesses to its internal data are consistent for each thread. The main application thread uses a lock on the same instance of the class to periodically check to see if any worker threads still exist, and waits to exit until all worker threads have completed their tasks.

```
// msl_lock_lock_when.cpp
// compile with: /clr
#include <msclr/lock.h>

using namespace System;
using namespace System::Threading;
using namespace msclr;

ref class CounterClass {
private:
   int Counter;

public:
   property int ThreadCount;

   // function called by multiple threads, use lock to keep Counter consistent
   // for each thread
   void UseCounter() {
      try {
         lock l(this); // wait infinitely

         Console::WriteLine("In thread {0}, Counter = {1}", Thread::CurrentThread->ManagedT
hreadId,
            Counter);

         for (int i = 0; i < 10; i++) {
            Counter++;
            Thread::Sleep(10);
         }

         Console::WriteLine("In thread {0}, Counter = {1}", Thread::CurrentThread->ManagedT
hreadId,
            Counter);

         Counter = 0;
         // lock is automatically released when it goes out of scope and its destructor is
called
      }
      catch (...) {
         Console::WriteLine("Couldn't acquire lock!");
      }

      ThreadCount--;
   }
};
```

```
int main() {
   // create a few threads to contend for access to the shared data
   CounterClass^ cc = gcnew CounterClass;
   array<Thread^>^ tarr = gcnew array<Thread^>(5);
   ThreadStart^ startDelegate = gcnew ThreadStart(cc, &CounterClass::UseCounter);
   for (int i = 0; i < tarr->Length; i++) {
      tarr[i] = gcnew Thread(startDelegate);
      cc->ThreadCount++;
      tarr[i]->Start();
   }

   // keep our main thread alive until all worker threads have completed
   lock l(cc, lock_later); // don't lock now, just create the object
   while (true) {
      if (l.try_acquire(50)) { // try to acquire lock, don't throw an exception if can't
         if (0 == cc->ThreadCount) {
            Console::WriteLine("All threads completed.");
            break; // all threads are gone, exit while
         }
         else {
            Console::WriteLine("{0} threads exist, continue waiting...", cc->ThreadCount);
            l.release(); // some threads exist, let them do their work
         }
      }
   }
}
```

Sample Output

```
In thread 3, Counter = 0
In thread 3, Counter = 10
In thread 5, Counter = 0
In thread 5, Counter = 10
In thread 7, Counter = 0
In thread 7, Counter = 10
In thread 4, Counter = 0
In thread 4, Counter = 10
In thread 6, Counter = 0
In thread 6, Counter = 10
All threads completed.
```

Requirements

Header file <msclr\lock.h>

Namespace msclr

See Also

Reference

lock

Calling Functions in a Specific Application Domain

Supports calling functions in a specific application domain.

In This Section

call_in_appdomain Function

See Also

Other Resources

C++ Support Library

call_in_appdomain Function

Executes a function in a specified application domain.

```
template <typename ArgType1, ...typename ArgTypeN>
void call_in_appdomain(
   DWORD appdomainId,
   void (*voidFunc)(ArgType1, ...ArgTypeN) [ ,
   ArgType1 arg1,
   ...
   ArgTypeN argN ]
);
template <typename RetType, typename ArgType1, ...typename ArgTypeN>
RetType call_in_appdomain(
   DWORD appdomainId,
   RetType (*nonvoidFunc)(ArgType1, ...ArgTypeN) [ ,
   ArgType1 arg1,
   ...
   ArgTypeN argN ]
);
```

Parameters

appdomainId

The appdomain in which to call the function.

voidFunc

Pointer to a **void** function that takes N parameters (0 <= N <= 15).

nonvoidFunc

Pointer to a non-**void** function that takes N parameters (0 <= N <= 15).

arg1...argN

Zero to 15 parameters to be passed to *voidFunc* or *nonvoidFunc* in the other appdomain.

Return Value

The result of executing *voidFunc* or *nonvoidFunc* in the specified application domain.

Remarks

The arguments of the function passed to **call_in_appdomain** must not be CLR types.

Example

```
// msl_call_in_appdomain.cpp
// compile with: /clr

// Defines two functions: one takes a parameter and returns nothing,
// the other takes no parameters and returns an int.  Calls both
// functions in the default appdomain and in a newly-created
// application domain using call_in_appdomain.

#include <msclr\appdomain.h>

using namespace System;
using namespace msclr;

void PrintCurrentDomainName( char* format )
{
   String^ s = gcnew String(format);
   Console::WriteLine( s, AppDomain::CurrentDomain->FriendlyName );
}
```

```
int GetDomainId()
{
   return AppDomain::CurrentDomain->Id;
}

int main()
{
   AppDomain^ appDomain1 = AppDomain::CreateDomain( "First Domain" );

   call_in_appdomain( AppDomain::CurrentDomain->Id,
                      &PrintCurrentDomainName,
                      (char*)"default appdomain: {0}" );
   call_in_appdomain( appDomain1->Id,
                      &PrintCurrentDomainName,
                      (char*)"in appDomain1: {0}" );

   int id;
   id = call_in_appdomain( AppDomain::CurrentDomain->Id, &GetDomainId );
   Console::WriteLine( "default appdomain id = {0}", id );
   id = call_in_appdomain( appDomain1->Id, &GetDomainId );
   Console::WriteLine( "appDomain1 id = {0}", id );
}
```

Output

```
default appdomain: msl_call_in_appdomain.exe
in appDomain1: First Domain
default appdomain id = 1
appDomain1 id = 2
```

Requirements

Header file <msclr\appdomain.h>

Namespace msclr

com::ptr

A wrapper for a COM object that can be used as a member of a CLR class. The wrapper also automates lifetime management of the COM object, releasing owned references on the object when its destructor is called. Analogous to CComPtr Class.

```
#include <msclr\com\ptr.h>
```

Remarks

com::ptr Class is defined in the <msclr\com\ptr.h> file.

See Also

Other Resources

C++ Support Library

com::ptr Class

A wrapper for a COM object that can be used as a member of a CLR class. The wrapper also automates lifetime management of the COM object, releasing all owned references on the object when its destructor is called. Analogous to CComPtr Class.

```
template<class _interface_type>
ref class ptr;
```

Parameters

_interface_type

COM interface.

Remarks

A **com::ptr** can also be used as a local function variable to simplify various COM tasks and to automate lifetime management.

A **com::ptr** cannot be used directly as a function parameter; use a % (Tracking Reference) or a handle (^) instead.

A **com::ptr** cannot be directly returned from a function; use a handle instead.

Example

This example implements a CLR class that uses a **com::ptr** to wrap its private member `IXMLDOMDocument` object. Calling the public methods of the class results in calls to the contained `IXMLDOMDocument` object. The sample creates an instance of an XML document, fills it with some simple XML, and does a simplified walk of the nodes in the parsed document tree to print the XML to the console.

```
// comptr.cpp
// compile with: /clr /link msxml2.lib
#include <msxml2.h>
#include <msclr\com\ptr.h>

#import <msxml3.dll> raw_interfaces_only

using namespace System;
using namespace System::Runtime::InteropServices;
using namespace msclr;

// a ref class that uses a com::ptr to contain an
// IXMLDOMDocument object
ref class XmlDocument {
public:
   // construct the internal com::ptr with a null interface
   // and use CreateInstance to fill it
   XmlDocument(String^ progid) {
      m_ptrDoc.CreateInstance(progid);
   }

   void LoadXml(String^ xml) {
      pin_ptr<const wchar_t> pinnedXml = PtrToStringChars(xml);
      BSTR bstr = NULL;

      try {
         // load some XML into the document
         bstr = ::SysAllocString(pinnedXml);
         if (NULL == bstr) {
            throw gcnew OutOfMemoryException;
         }
         VARIANT_BOOL bIsSuccessful = false;
         // use operator -> to call IXMODOMDocument member function
         Marshal::ThrowExceptionForHR(m_ptrDoc->loadXML(bstr, &bIsSuccessful));
      }
      finally {
         ::SysFreeString(bstr);
```

```
      }
   }

   // simplified function to write just the first xml node to the console
   void WriteXml() {
      IXMLDOMNode* pNode = NULL;

      try {
         // the first child of the document is the first real xml node
         Marshal::ThrowExceptionForHR(m_ptrDoc->get_firstChild(&pNode));
         if (NULL != pNode) {
            WriteNode(pNode);
         }
      }
      finally {
         if (NULL != pNode) {
            pNode->Release();
         }
      }
   }

   // note that the destructor will call the com::ptr destructor
   // and automatically release the reference to the COM object

private:
   // simplified function that only writes the node
   void WriteNode(IXMLDOMNode* pNode) {
      BSTR bstr = NULL;

      try {
         // write out the name and text properties
         Marshal::ThrowExceptionForHR(pNode->get_nodeName(&bstr));
         String^ strName = gcnew String(bstr);
         Console::Write("<{0}>", strName);
         ::SysFreeString(bstr);
         bstr = NULL;

         Marshal::ThrowExceptionForHR(pNode->get_text(&bstr));
         Console::Write(gcnew String(bstr));
         ::SysFreeString(bstr);
         bstr = NULL;

         Console::WriteLine("</{0}>", strName);
      }
      finally {
         ::SysFreeString(bstr);
      }
   }

   com::ptr<IXMLDOMDocument> m_ptrDoc;
};

// use the ref class to handle an XML DOM Document object
int main() {
   try {
      // create the class from a progid string
      XmlDocument doc("Msxml2.DOMDocument.3.0");

      // stream some xml into the document
      doc.LoadXml("<word>persnickety</word>");

      // write the document to the console
      doc.WriteXml();
   }
   catch (Exception^ e) {
      Console::WriteLine(e);
   }
}
```

Output

```
<word>persnickety</word>
```

Requirements

Header file <msclr\com\ptr.h>

Namespace msclr::com

See Also

Other Resources

C++ Support Library

ptr Members

ptr Members

Members of **com::ptr**.

In This Section

ptr::ptr

ptr::~ptr

ptr::Attach

ptr::CreateInstance

ptr::Detach

ptr::GetInterface

ptr::QueryInterface

ptr::Release

ptr::operator=

ptr::operator->

ptr::operator!

ptr::operator bool

See Also

Reference

com::ptr Class

ptr::ptr

Constructs a **com::ptr** to wrap a COM object.

```
ptr();
ptr(
   _interface_type * p
);
```

Parameters

P

A COM interface pointer.

Remarks

The no-argument constructor assigns **nullptr** to the underlying object handle. Subsequent calls to the **com::ptr** will validate the internal object and silently fail until an object is actually created or attached.

The one-argument constructor adds a reference to the COM object but does not release the caller's reference, so the caller must call `Release` on the COM object to truly give up control. When the **com::ptr**'s destructor is called it will automatically release its references on the COM object.

Passing **NULL** to this constructor is the same as calling the no-argument version.

Example

This example implements a CLR class that uses a **com::ptr** to wrap its private member `IXMLDOMDocument` object. It demonstrates usage of both versions of the constructor.

```
// comptr_ptr.cpp
// compile with: /clr /link msxml2.lib
#include <msxml2.h>
#include <msclr\com\ptr.h>

#import <msxml3.dll> raw_interfaces_only

using namespace System;
using namespace System::Runtime::InteropServices;
using namespace msclr;

// a ref class that uses a com::ptr to contain an
// IXMLDOMDocument object
ref class XmlDocument {
public:
   // construct the internal com::ptr with a null interface
   // and use CreateInstance to fill it
   XmlDocument(String^ progid) {
      m_ptrDoc.CreateInstance(progid);
   }

   // construct the internal com::ptr with a COM object
   XmlDocument(IXMLDOMDocument* pDoc) : m_ptrDoc(pDoc) {}

   // note that the destructor will call the com::ptr destructor
   // and automatically release the reference to the COM object

private:
   com::ptr<IXMLDOMDocument> m_ptrDoc;
};

// use the ref class to handle an XML DOM Document object
int main() {
   IXMLDOMDocument* pDoc = NULL;
```

```
    try {
       // create an XML DOM document object
       Marshal::ThrowExceptionForHR(CoCreateInstance(CLSID_DOMDocument30, NULL,
          CLSCTX_ALL, IID_IXMLDOMDocument, (void**)&pDoc));
       // construct the ref class with the COM object
       XmlDocument doc1(pDoc);

       // or create the class from a progid string
       XmlDocument doc2("Msxml2.DOMDocument.3.0");
    }
    // doc1 and doc2 destructors are called when they go out of scope
    // and the internal com::ptr releases its reference to the COM object
    catch (Exception^ e) {
       Console::WriteLine(e);
    }
    finally {
       if (NULL != pDoc) {
          pDoc->Release();
       }
    }
 }
```

Requirements

Header file <msclr\com\ptr.h>

Namespace msclr::com

See Also

Reference

ptr::CreateInstance

ptr::~ptr

Other Resources

ptr Members

ptr::~ptr

Destructs a **com::ptr**.

```
~ptr();
```

Remarks

On destruction, the **com::ptr** releases all references it owns to its COM object. Assuming that there are no other references held to the COM object, the COM object will be deleted and its memory freed.

Example

This example implements a CLR class that uses a **com::ptr** to wrap its private member `IXMLDOMDocument` object. In the `main` function, the two `XmlDocument` objects' destructors will be called when they go out of the scope of the `try` block, resulting in the underlying **com::ptr** destructor being called, releasing all owned references to the COM object.

```
// comptr_dtor.cpp
// compile with: /clr /link msxml2.lib
#include <msxml2.h>
#include <msclr\com\ptr.h>

#import <msxml3.dll> raw_interfaces_only

using namespace System;
using namespace System::Runtime::InteropServices;
using namespace msclr;

// a ref class that uses a com::ptr to contain an
// IXMLDOMDocument object
ref class XmlDocument {
public:
   // construct the internal com::ptr with a null interface
   // and use CreateInstance to fill it
   XmlDocument(String^ progid) {
      m_ptrDoc.CreateInstance(progid);
   }

   // construct the internal com::ptr with a COM object
   XmlDocument(IXMLDOMDocument* pDoc) : m_ptrDoc(pDoc) {}

   // note that the destructor will call the com::ptr destructor
   // and automatically release the reference to the COM object

private:
   com::ptr<IXMLDOMDocument> m_ptrDoc;
};

// use the ref class to handle an XML DOM Document object
int main() {
   IXMLDOMDocument* pDoc = NULL;

   try {
      // create an XML DOM document object
      Marshal::ThrowExceptionForHR(CoCreateInstance(CLSID_DOMDocument30, NULL,
         CLSCTX_ALL, IID_IXMLDOMDocument, (void**)&pDoc));
      // construct the ref class with the COM object
      XmlDocument doc1(pDoc);

      // or create the class from a progid string
      XmlDocument doc2("Msxml2.DOMDocument.3.0");
   }
   // doc1 and doc2 destructors are called when they go out of scope
   // and the internal com::ptr releases its reference to the COM object
```

```
    catch (Exception^ e) {
       Console::WriteLine(e);
    }
    finally {
       if (NULL != pDoc) {
          pDoc->Release();
       }
    }
 }
```

Requirements

Header file <msclr\com\ptr.h>

Namespace msclr::com

See Also

Reference

ptr::ptr

ptr::CreateInstance

Other Resources

ptr Members

ptr::Attach

Attaches a COM object to a **com::ptr**.

```
void Attach(
   _interface_type * _right
);
```

Parameters

_right

The COM interface pointer to attach.

Exceptions

If the **com::ptr** already owns a reference to a COM object, `Attach` throws InvalidOperationException.

Remarks

A call to `Attach` references the COM object but does not release the caller's reference to it.

Passing **NULL** to `Attach` results in no action being taken.

Example

This example implements a CLR class that uses a **com::ptr** to wrap its private member `IXMLDOMDocument` object. The `ReplaceDocument` member function first calls `Release` on any previously owned object and then calls `Attach` to attach a new document object.

```
// comptr_attach.cpp
// compile with: /clr /link msxml2.lib
#include <msxml2.h>
#include <msclr\com\ptr.h>

#import <msxml3.dll> raw_interfaces_only

using namespace System;
using namespace System::Runtime::InteropServices;
using namespace msclr;

// a ref class that uses a com::ptr to contain an
// IXMLDOMDocument object
ref class XmlDocument {
public:
   // construct the internal com::ptr with a null interface
   // and use CreateInstance to fill it
   XmlDocument(String^ progid) {
      m_ptrDoc.CreateInstance(progid);
   }

   // replace currently held COM object with another one
   void ReplaceDocument(IXMLDOMDocument* pDoc) {
      // release current document object
      m_ptrDoc.Release();
      // attach the new document object
      m_ptrDoc.Attach(pDoc);
   }

   // note that the destructor will call the com::ptr destructor
   // and automatically release the reference to the COM object

private:
   com::ptr<IXMLDOMDocument> m_ptrDoc;
};
```

```
// unmanaged function that creates a raw XML DOM Document object
IXMLDOMDocument* CreateDocument() {
   IXMLDOMDocument* pDoc = NULL;
   Marshal::ThrowExceptionForHR(CoCreateInstance(CLSID_DOMDocument30, NULL,
      CLSCTX_INPROC_SERVER, IID_IXMLDOMDocument, (void**)&pDoc));
   return pDoc;
}


// use the ref class to handle an XML DOM Document object
int main() {
   IXMLDOMDocument* pDoc = NULL;

   try {
      // create the class from a progid string
      XmlDocument doc("Msxml2.DOMDocument.3.0");

      // get another document object from unmanaged function and
      // store it in place of the one held by our ref class
      pDoc = CreateDocument();
      doc.ReplaceDocument(pDoc);
      // no further need for raw object reference
      pDoc->Release();
      pDoc = NULL;
   }
   catch (Exception^ e) {
      Console::WriteLine(e);
   }
   finally {
      if (NULL != pDoc) {
         pDoc->Release();
      }
   }
}
```

Requirements

Header file <msclr\com\ptr.h>

Namespace msclr::com

See Also

Reference

ptr::operator=

ptr::Release

Other Resources

ptr Members

ptr::CreateInstance

Creates an instance of a COM object within a **com::ptr**.

```
void CreateInstance(
   System::String ^ progid,
   LPUNKNOWN pouter,
   DWORD cls_context
);
void CreateInstance(
   System::String ^ progid,
   LPUNKNOWN pouter
);
void CreateInstance(
   System::String ^ progid
);
void CreateInstance(
   const wchar_t * progid,
   LPUNKNOWN pouter,
   DWORD cls_context
);
void CreateInstance(
   const wchar_t * progid,
   LPUNKNOWN pouter
);
void CreateInstance(
   const wchar_t * progid
);
void CreateInstance(
   REFCLSID rclsid,
   LPUNKNOWN pouter,
   DWORD cls_context
);
void CreateInstance(
   REFCLSID rclsid,
   LPUNKNOWN pouter
);
void CreateInstance(
   REFCLSID rclsid
);
```

Parameters

progid

A ProgID string.

pouter

Pointer to the aggregate object's IUnknown interface (the controlling IUnknown). If *pouter* is not specified, **NULL** is used.

cls_context

Context in which the code that manages the newly created object will run. The values are taken from the CLSCTX enumeration. If *cls_context* is not specified, the value CLSCTX_ALL is used.

rclsid

CLSID associated with the data and code that will be used to create the object.

Exceptions

If the **com::ptr** already owns a reference to a COM object, `CreateInstance` throws InvalidOperationException.

This function calls `CoCreateInstance` and uses ThrowExceptionForHR to convert any error **HRESULT** to an appropriate exception.

Remarks

`CreateInstance` uses `CoCreateInstance` to create a new instance of the specified object, identified either from a ProgID or a CLSID. The **com::ptr** references the newly created object and will automatically release all owned references upon destruction.

Example

This example implements a CLR class that uses a **com::ptr** to wrap its private member `IXMLDOMDocument` object. The class constructors use two different forms of `CreateInstance` to create the document object either from a ProgID or from a CLSID plus a CLSCTX.

```
// comptr_createinstance.cpp
// compile with: /clr /link msxml2.lib
#include <msxml2.h>
#include <msclr\com\ptr.h>

#import <msxml3.dll> raw_interfaces_only

using namespace System;
using namespace System::Runtime::InteropServices;
using namespace msclr;

// a ref class that uses a com::ptr to contain an
// IXMLDOMDocument object
ref class XmlDocument {
public:
   // construct the internal com::ptr with a null interface
   // and use CreateInstance to fill it
   XmlDocument(String^ progid) {
      m_ptrDoc.CreateInstance(progid);
   }
   XmlDocument(REFCLSID clsid, DWORD clsctx) {
      m_ptrDoc.CreateInstance(clsid, NULL, clsctx);
   }

   // note that the destructor will call the com::ptr destructor
   // and automatically release the reference to the COM object

private:
   com::ptr<IXMLDOMDocument> m_ptrDoc;
};

// use the ref class to handle an XML DOM Document object
int main() {
   try {
      // create the class from a progid string
      XmlDocument doc1("Msxml2.DOMDocument.3.0");

      // or from a clsid with specific CLSCTX
      XmlDocument doc2(CLSID_DOMDocument30, CLSCTX_INPROC_SERVER);
   }
   catch (Exception^ e) {
      Console::WriteLine(e);
   }
}
```

Requirements

Header file <msclr\com\ptr.h>

Namespace msclr::com

See Also

Other Resources

ptr Members

ptr::Detach

Gives up ownership of the COM object, returning a pointer to the object.

```
_interface_type * Detach();
```

Return Value

The pointer to the COM object.

If no object is owned, NULL is returned.

Exceptions

Internally, `QueryInterface` is called on the owned COM object and any error **HRESULT** is converted to an exception by ThrowExceptionForHR.

Remarks

`Detach` first adds a reference to the COM object on behalf of the caller and then releases all references owned by the **com::ptr**. The caller must ultimately release the returned object to destroy it.

Example

This example implements a CLR class that uses a **com::ptr** to wrap its private member `IXMLDOMDocument` object. The `DetachDocument` member function calls `Detach` to give up ownership of the COM object and return a pointer to the caller.

```
// comptr_detach.cpp
// compile with: /clr /link msxml2.lib
#include <msxml2.h>
#include <msclr\com\ptr.h>

#import <msxml3.dll> raw_interfaces_only

using namespace System;
using namespace System::Runtime::InteropServices;
using namespace msclr;

// a ref class that uses a com::ptr to contain an
// IXMLDOMDocument object
ref class XmlDocument {
public:
   // construct the internal com::ptr with a null interface
   // and use CreateInstance to fill it
   XmlDocument(String^ progid) {
      m_ptrDoc.CreateInstance(progid);
   }

   // detach the COM object and return it
   // this releases the internal reference to the object
   IXMLDOMDocument* DetachDocument() {
      return m_ptrDoc.Detach();
   }

   // note that the destructor will call the com::ptr destructor
   // and automatically release the reference to the COM object

private:
   com::ptr<IXMLDOMDocument> m_ptrDoc;
};

// unmanaged function that loads XML into a raw XML DOM Document object
HRESULT LoadXml(IXMLDOMDocument* pDoc, BSTR bstrXml) {
   HRESULT hr = S_OK;
   VARIANT_BOOL bSuccess;
```

```
   hr = pDoc->loadXML(bstrXml, &bSuccess);
   if (S_OK == hr && !bSuccess) {
      hr = E_FAIL;
   }
   return hr;
}


// use the ref class to handle an XML DOM Document object
int main() {
   IXMLDOMDocument* pDoc = NULL;
   BSTR bstrXml = NULL;

   try {
      // create the class from a progid string
      XmlDocument doc("Msxml2.DOMDocument.3.0");

      bstrXml = ::SysAllocString(L"<word>persnickety</word>");
      if (NULL == bstrXml) {
         throw gcnew OutOfMemoryException("bstrXml");
      }
      // detach the document object from the ref class
      pDoc = doc.DetachDocument();
      // use unmanaged function and raw object to load xml
      Marshal::ThrowExceptionForHR(LoadXml(pDoc, bstrXml));
      // release document object as the ref class no longer owns it
      pDoc->Release();
      pDoc = NULL;
   }
   catch (Exception^ e) {
      Console::WriteLine(e);
   }
   finally {
      if (NULL != pDoc) {
         pDoc->Release();
      }

   }
}
```

Requirements

Header file <msclr\com\ptr.h>

Namespace msclr::com

See Also

Reference

ptr::Release

ptr::Attach

Other Resources

ptr Members

ptr::GetInterface

Returns a pointer to the owned COM object.

```
_interface_type * GetInterface();
```

Return Value

A pointer to the owned COM object.

Exceptions

Internally, `QueryInterface` is called on the owned COM object and any error **HRESULT** is converted to an exception by ThrowExceptionForHR.

Remarks

The **com::ptr** adds a reference to the COM object on the caller's behalf and also keeps its own reference on the COM object. The caller must ultimately release the reference on the returned object or it will never be destroyed.

Example

This example implements a CLR class that uses a **com::ptr** to wrap its private member `IXMLDOMDocument` object. The `GetDocument` member function uses `GetInterface` to return a pointer to the COM object.

```
// comptr_getinterface.cpp
// compile with: /clr /link msxml2.lib
#include <msxml2.h>
#include <msclr\com\ptr.h>

#import <msxml3.dll> raw_interfaces_only

using namespace System;
using namespace System::Runtime::InteropServices;
using namespace msclr;

// a ref class that uses a com::ptr to contain an
// IXMLDOMDocument object
ref class XmlDocument {
public:
   // construct the internal com::ptr with a null interface
   // and use CreateInstance to fill it
   XmlDocument(String^ progid) {
      m_ptrDoc.CreateInstance(progid);
   }

   // add a reference to and return the COM object
   // but keep an internal reference to the object
   IXMLDOMDocument* GetDocument() {
      return m_ptrDoc.GetInterface();
   }

   // simplified function that only writes the first node
   void WriteDocument() {
      IXMLDOMNode* pNode = NULL;
      BSTR bstr = NULL;

      try {
         // use operator -> to call XML Doc member
         Marshal::ThrowExceptionForHR(m_ptrDoc->get_firstChild(&pNode));
         if (NULL != pNode) {
            // write out the xml
            Marshal::ThrowExceptionForHR(pNode->get_nodeName(&bstr));
            String^ strName = gcnew String(bstr);
            Console::Write("<{0}>", strName);
```

```
            ::SysFreeString(bstr);
            bstr = NULL;

            Marshal::ThrowExceptionForHR(pNode->get_text(&bstr));
            Console::Write(gcnew String(bstr));
            ::SysFreeString(bstr);
            bstr = NULL;

            Console::WriteLine("</{0}>", strName);
         }
      }
      finally {
         if (NULL != pNode) {
            pNode->Release();
         }
         ::SysFreeString(bstr);
      }
   }

   // note that the destructor will call the com::ptr destructor
   // and automatically release the reference to the COM object

private:
   com::ptr<IXMLDOMDocument> m_ptrDoc;
};

// unmanaged function that loads XML into a raw XML DOM Document object
HRESULT LoadXml(IXMLDOMDocument* pDoc, BSTR bstrXml) {
   HRESULT hr = S_OK;
   VARIANT_BOOL bSuccess;
   hr = pDoc->loadXML(bstrXml, &bSuccess);
   if (S_OK == hr && !bSuccess) {
      hr = E_FAIL;
   }
   return hr;
}


// use the ref class to handle an XML DOM Document object
int main() {
   IXMLDOMDocument* pDoc = NULL;
   BSTR bstrXml = NULL;

   try {
      // create the class from a progid string
      XmlDocument doc("Msxml2.DOMDocument.3.0");

      bstrXml = ::SysAllocString(L"<word>persnickety</word>");
      if (NULL == bstrXml) {
         throw gcnew OutOfMemoryException("bstrXml");
      }
      // detach the document object from the ref class
      pDoc = doc.GetDocument();
      // use unmanaged function and raw object to load xml
      Marshal::ThrowExceptionForHR(LoadXml(pDoc, bstrXml));
      // release reference to document object (but ref class still references it)
      pDoc->Release();
      pDoc = NULL;

      // call another function on the ref class
      doc.WriteDocument();
   }
   catch (Exception^ e) {
      Console::WriteLine(e);
   }
   finally {
      if (NULL != pDoc) {
         pDoc->Release();
```

```
        }

    }
}
```

Output

```
<word>persnickety</word>
```

Requirements

Header file <msclr\com\ptr.h>

Namespace msclr::com

See Also

Reference

ptr::QueryInterface

Other Resources

ptr Members

ptr::QueryInterface

Queries the owned COM object for an interface and attaches the result to another **com::ptr**.

```
template<class _other_type>
void QueryInterface(
   ptr<_other_type> % other
);
```

Parameters

other

The **com::ptr** that will get the interface.

Exceptions

Internally, `QueryInterface` is called on the owned COM object and any error **HRESULT** is converted to an exception by ThrowExceptionForHR.

Remarks

Use this method to create a COM wrapper for a different interface of the COM object owned by the current wrapper. This method calls `QueryInterface` through the owned COM object to request a pointer to a specific interface of the COM object and attaches the returned interface pointer to the passed-in **com::ptr**.

Example

This example implements a CLR class that uses a **com::ptr** to wrap its private member `IXMLDOMDocument` object. The `WriteTopLevelNode` member function uses `QueryInterface` to fill a local **com::ptr** with an `IXMLDOMNode` and then passes the **com::ptr** (by tracking reference) to a private member function that writes the node's name and text properties to the console.

```
// comptr_queryinterface.cpp
// compile with: /clr /link msxml2.lib
#include <msxml2.h>
#include <msclr\com\ptr.h>

#import <msxml3.dll> raw_interfaces_only

using namespace System;
using namespace System::Runtime::InteropServices;
using namespace msclr;

// a ref class that uses a com::ptr to contain an
// IXMLDOMDocument object
ref class XmlDocument {
public:
   // construct the internal com::ptr with a null interface
   // and use CreateInstance to fill it
   XmlDocument(String^ progid) {
      m_ptrDoc.CreateInstance(progid);
   }

   void LoadXml(String^ xml) {
      pin_ptr<const wchar_t> pinnedXml = PtrToStringChars(xml);
      BSTR bstr = NULL;

      try {
         // load some XML into our document
         bstr = ::SysAllocString(pinnedXml);
         if (NULL == bstr) {
            throw gcnew OutOfMemoryException;
         }
         VARIANT_BOOL bIsSuccessful = false;
         // use operator -> to call IXMODOMDocument member function
         Marshal::ThrowExceptionForHR(m_ptrDoc->loadXML(bstr, &bIsSuccessful));
```

```
      }
      finally {
         ::SysFreeString(bstr);
      }
   }

   // write the top level node to the console
   void WriteTopLevelNode() {
      com::ptr<IXMLDOMNode> ptrNode;

      // query for the top level node interface
      m_ptrDoc.QueryInterface(ptrNode);
      WriteNode(ptrNode);
   }

   // note that the destructor will call the com::ptr destructor
   // and automatically release the reference to the COM object

private:
   // simplified function that only writes the node
   void WriteNode(com::ptr<IXMLDOMNode> % node) {
      BSTR bstr = NULL;

      try {
         // write out the name and text properties
         Marshal::ThrowExceptionForHR(node->get_nodeName(&bstr));
         String^ strName = gcnew String(bstr);
         Console::Write("<{0}>", strName);
         ::SysFreeString(bstr);
         bstr = NULL;

         Marshal::ThrowExceptionForHR(node->get_text(&bstr));
         Console::Write(gcnew String(bstr));
         ::SysFreeString(bstr);
         bstr = NULL;

         Console::WriteLine("</{0}>", strName);
      }
      finally {
         ::SysFreeString(bstr);
      }
   }

   com::ptr<IXMLDOMDocument> m_ptrDoc;
};

// use the ref class to handle an XML DOM Document object
int main() {
   try {
      // create the class from a progid string
      XmlDocument doc("Msxml2.DOMDocument.3.0");

      // stream some xml into the document
      doc.LoadXml("<word>persnickety</word>");

      // write the document to the console
      doc.WriteTopLevelNode();
   }
   catch (Exception^ e) {
      Console::WriteLine(e);
   }
}
```

Output

```
<#document>persnickety</#document>
```

Requirements

Header file <msclr\com\ptr.h>

Namespace msclr::com

See Also

Reference

ptr::GetInterface

Other Resources

ptr Members

ptr::Release

Releases all owned references on the COM object.

```
void Release();
```

Remarks

Calling this function releases all owned references on the COM object and sets the internal handle to the COM object to **nullptr**. If no other references on the COM object exist, it will be destroyed.

Example

This example implements a CLR class that uses a **com::ptr** to wrap its private member `IXMLDOMDocument` object. The `ReplaceDocument` member function uses `Release` to release any prior document object before attaching the new document.

```
// comptr_release.cpp
// compile with: /clr /link msxml2.lib
#include <msxml2.h>
#include <msclr\com\ptr.h>

#import <msxml3.dll> raw_interfaces_only

using namespace System;
using namespace System::Runtime::InteropServices;
using namespace msclr;

// a ref class that uses a com::ptr to contain an
// IXMLDOMDocument object
ref class XmlDocument {
public:
   // construct the internal com::ptr with a null interface
   // and use CreateInstance to fill it
   XmlDocument(String^ progid) {
      m_ptrDoc.CreateInstance(progid);
   }

   // replace currently held COM object with another one
   void ReplaceDocument(IXMLDOMDocument* pDoc) {
      // release current document object
      m_ptrDoc.Release();
      // attach the new document object
      m_ptrDoc.Attach(pDoc);
   }

   // note that the destructor will call the com::ptr destructor
   // and automatically release the reference to the COM object

private:
   com::ptr<IXMLDOMDocument> m_ptrDoc;
};

// unmanaged function that creates a raw XML DOM Document object
IXMLDOMDocument* CreateDocument() {
   IXMLDOMDocument* pDoc = NULL;
   Marshal::ThrowExceptionForHR(CoCreateInstance(CLSID_DOMDocument30, NULL,
      CLSCTX_INPROC_SERVER, IID_IXMLDOMDocument, (void**)&pDoc));
   return pDoc;
}


// use the ref class to handle an XML DOM Document object
int main() {
   IXMLDOMDocument* pDoc = NULL;
```

```
    try {
       // create the class from a progid string
       XmlDocument doc("Msxml2.DOMDocument.3.0");

       // get another document object from unmanaged function and
       // store it in place of the one held by our ref class
       pDoc = CreateDocument();
       doc.ReplaceDocument(pDoc);
       // no further need for raw object reference
       pDoc->Release();
       pDoc = NULL;
    }
    catch (Exception^ e) {
       Console::WriteLine(e);
    }
    finally {
       if (NULL != pDoc) {
          pDoc->Release();
       }
    }
 }
```

Requirements

Header file <msclr\com\ptr.h>

Namespace msclr::com

See Also

Reference

ptr::Detach

Other Resources

ptr Members

ptr::operator=

Attaches a COM object to a **com::ptr**.

```
ptr<_interface_type> % operator=(
   _interface_type * _right
);
```

Parameters

_right

The COM interface pointer to attach.

Return Value

A tracking reference on the **com::ptr**.

Exceptions

If the **com::ptr** already owns a reference to a COM object, **operator=** throws InvalidOperationException.

Remarks

Assigning a COM object to a **com::ptr** references the COM object but does not release the caller's reference to it.

This operator has the same effect as `Attach`.

Example

This example implements a CLR class that uses a **com::ptr** to wrap its private member `IXMLDOMDocument` object. The `ReplaceDocument` member function first calls `Release` on any previously owned object and then uses `operator=` to attach a new document object.

```
// comptr_op_assign.cpp
// compile with: /clr /link msxml2.lib
#include <msxml2.h>
#include <msclr\com\ptr.h>

#import <msxml3.dll> raw_interfaces_only

using namespace System;
using namespace System::Runtime::InteropServices;
using namespace msclr;

// a ref class that uses a com::ptr to contain an
// IXMLDOMDocument object
ref class XmlDocument {
public:
   // construct the internal com::ptr with a null interface
   // and use CreateInstance to fill it
   XmlDocument(String^ progid) {
      m_ptrDoc.CreateInstance(progid);
   }

   // replace currently held COM object with another one
   void ReplaceDocument(IXMLDOMDocument* pDoc) {
      // release current document object
      m_ptrDoc.Release();
      // attach the new document object
      m_ptrDoc = pDoc;
   }

   // note that the destructor will call the com::ptr destructor
   // and automatically release the reference to the COM object

private:
```

```
   com::ptr<IXMLDOMDocument> m_ptrDoc;
};

// unmanaged function that creates a raw XML DOM Document object
IXMLDOMDocument* CreateDocument() {
   IXMLDOMDocument* pDoc = NULL;
   Marshal::ThrowExceptionForHR(CoCreateInstance(CLSID_DOMDocument30, NULL,
      CLSCTX_INPROC_SERVER, IID_IXMLDOMDocument, (void**)&pDoc));
   return pDoc;
}


// use the ref class to handle an XML DOM Document object
int main() {
   IXMLDOMDocument* pDoc = NULL;

   try {
      // create the class from a progid string
      XmlDocument doc("Msxml2.DOMDocument.3.0");

      // get another document object from unmanaged function and
      // store it in place of the one held by the ref class
      pDoc = CreateDocument();
      doc.ReplaceDocument(pDoc);
      // no further need for raw object reference
      pDoc->Release();
      pDoc = NULL;
   }
   catch (Exception^ e) {
      Console::WriteLine(e);
   }
   finally {
      if (NULL != pDoc) {
         pDoc->Release();
      }
   }
}
```

Requirements

Header file <msclr\com\ptr.h>

Namespace msclr::com

See Also

Reference

ptr::Attach

ptr::Detach

ptr::Release

Other Resources

ptr Members

ptr::operator->

Member access operator, used to call methods on the owned COM object.

```
_detail::smart_com_ptr<_interface_type> operator->();
```

Return Value

A `smart_com_ptr` to the COM object.

Exceptions

Internally, `QueryInterface` is called on the owned COM object and any error **HRESULT** is converted to an exception by ThrowExceptionForHR.

Remarks

This operator allows you to call methods of the owned COM object. It returns a temporary `smart_com_ptr` that automatically handles its own `AddRef` and `Release`.

Example

This example implements a CLR class that uses a **com::ptr** to wrap its private member `IXMLDOMDocument` object. The `WriteDocument` functions uses `operator->` to call the `get_firstChild` member of the document object.

```
// comptr_op_member.cpp
// compile with: /clr /link msxml2.lib
#include <msxml2.h>
#include <msclr\com\ptr.h>

#import <msxml3.dll> raw_interfaces_only

using namespace System;
using namespace System::Runtime::InteropServices;
using namespace msclr;

// a ref class that uses a com::ptr to contain an
// IXMLDOMDocument object
ref class XmlDocument {
public:
   // construct the internal com::ptr with a null interface
   // and use CreateInstance to fill it
   XmlDocument(String^ progid) {
      m_ptrDoc.CreateInstance(progid);
   }

   // add a reference to and return the COM object
   // but keep an internal reference to the object
   IXMLDOMDocument* GetDocument() {
      return m_ptrDoc.GetInterface();
   }

   // simplified function that only writes the first node
   void WriteDocument() {
      IXMLDOMNode* pNode = NULL;
      BSTR bstr = NULL;

      try {
         // use operator -> to call XML Doc member
         Marshal::ThrowExceptionForHR(m_ptrDoc->get_firstChild(&pNode));
         if (NULL != pNode) {
            // write out the xml
            Marshal::ThrowExceptionForHR(pNode->get_nodeName(&bstr));
            String^ strName = gcnew String(bstr);
            Console::Write("<{0}>", strName);
```

```
            ::SysFreeString(bstr);
            bstr = NULL;

            Marshal::ThrowExceptionForHR(pNode->get_text(&bstr));
            Console::Write(gcnew String(bstr));
            ::SysFreeString(bstr);
            bstr = NULL;

            Console::WriteLine("</{0}>", strName);
         }
      }
      finally {
         if (NULL != pNode) {
            pNode->Release();
         }
         ::SysFreeString(bstr);
      }
   }

   // note that the destructor will call the com::ptr destructor
   // and automatically release the reference to the COM object

private:
   com::ptr<IXMLDOMDocument> m_ptrDoc;
};

// unmanaged function that loads XML into a raw XML DOM Document object
HRESULT LoadXml(IXMLDOMDocument* pDoc, BSTR bstrXml) {
   HRESULT hr = S_OK;
   VARIANT_BOOL bSuccess;
   hr = pDoc->loadXML(bstrXml, &bSuccess);
   if (S_OK == hr && !bSuccess) {
      hr = E_FAIL;
   }
   return hr;
}


// use the ref class to handle an XML DOM Document object
int main() {
   IXMLDOMDocument* pDoc = NULL;
   BSTR bstrXml = NULL;

   try {
      // create the class from a progid string
      XmlDocument doc("Msxml2.DOMDocument.3.0");

      bstrXml = ::SysAllocString(L"<word>persnickety</word>");
      if (NULL == bstrXml) {
         throw gcnew OutOfMemoryException("bstrXml");
      }
      // detach the document object from the ref class
      pDoc = doc.GetDocument();
      // use unmanaged function and raw object to load xml
      Marshal::ThrowExceptionForHR(LoadXml(pDoc, bstrXml));
      // release reference to document object (but ref class still references it)
      pDoc->Release();
      pDoc = NULL;

      // call another function on the ref class
      doc.WriteDocument();
   }
   catch (Exception^ e) {
      Console::WriteLine(e);
   }
   finally {
      if (NULL != pDoc) {
         pDoc->Release();
```

```
        }

    }
}
```

Output

```
<word>persnickety</word>
```

Requirements

Header file <msclr\com\ptr.h>

Namespace msclr::com

See Also

Other Resources

ptr Members

ptr::operator!

Operator to determine if the owned COM object is invalid.

```
bool operator!();
```

Return Value

true if the owned COM object is invalid; **false** otherwise.

Remarks

The owned COM object is valid if it is not **nullptr**.

Example

This example implements a CLR class that uses a **com::ptr** to wrap its private member `IXMLDOMDocument` object. The `CreateInstance` member function uses `operator!` to determine if a document object is already owned, and only creates a new instance if the object is invalid.

```
// comptr_op_not.cpp
// compile with: /clr /link msxml2.lib
#include <msxml2.h>
#include <msclr\com\ptr.h>

#import <msxml3.dll> raw_interfaces_only

using namespace System;
using namespace System::Runtime::InteropServices;
using namespace msclr;

// a ref class that uses a com::ptr to contain an
// IXMLDOMDocument object
ref class XmlDocument {
public:
   void CreateInstance(String^ progid) {
      if (!m_ptrDoc) {
         m_ptrDoc.CreateInstance(progid);
         if (m_ptrDoc) {
            Console::WriteLine("DOM Document created.");
         }
      }
   }

   // note that the destructor will call the com::ptr destructor
   // and automatically release the reference to the COM object

private:
   com::ptr<IXMLDOMDocument> m_ptrDoc;
};

// use the ref class to handle an XML DOM Document object
int main() {
   try {
      XmlDocument doc;
      // create the instance from a progid string
      doc.CreateInstance("Msxml2.DOMDocument.3.0");
   }
   catch (Exception^ e) {
      Console::WriteLine(e);
   }
}
```

Output

```
DOM Document created.
```

Requirements

Header file <msclr\com\ptr.h>

Namespace msclr::com

See Also

Reference

ptr::operator bool

Other Resources

ptr Members

ptr::operator bool

Operator for using **com::ptr** in a conditional expression.

```
operator bool();
```

Return Value

true if the owned COM object is valid; **false** otherwise.

Remarks

The owned COM object is valid if it is not **nullptr**.

This operator actually converts to **_detail_class::_safe_bool** which is safer than **bool** because it cannot be converted to an integral type.

Example

This example implements a CLR class that uses a **com::ptr** to wrap its private member `IXMLDOMDocument` object. The `CreateInstance` member function uses `operator bool` after creating the new document object to determine if it is valid and writes to the console if it is.

```
// comptr_op_bool.cpp
// compile with: /clr /link msxml2.lib
#include <msxml2.h>
#include <msclr\com\ptr.h>

#import <msxml3.dll> raw_interfaces_only

using namespace System;
using namespace System::Runtime::InteropServices;
using namespace msclr;

// a ref class that uses a com::ptr to contain an
// IXMLDOMDocument object
ref class XmlDocument {
public:
   void CreateInstance(String^ progid) {
      if (!m_ptrDoc) {
         m_ptrDoc.CreateInstance(progid);
         if (m_ptrDoc) { // uses operator bool
            Console::WriteLine("DOM Document created.");
         }
      }
   }

   // note that the destructor will call the com::ptr destructor
   // and automatically release the reference to the COM object

private:
   com::ptr<IXMLDOMDocument> m_ptrDoc;
};

// use the ref class to handle an XML DOM Document object
int main() {
   try {
      XmlDocument doc;
      // create the instance from a progid string
      doc.CreateInstance("Msxml2.DOMDocument.3.0");
   }
   catch (Exception^ e) {
      Console::WriteLine(e);
   }
}
```

Output

```
DOM Document created.
```

Requirements

Header file <msclr\com\ptr.h>

Namespace msclr::com

See Also

Reference

ptr::operator!

Other Resources

ptr Members

C++ Attributes Reference

Attributes are designed to simplify COM programming and .NET Framework common language runtime development. When you include attributes in your source files, the compiler works with provider DLLs to insert code or modify the code in the generated object files.

In This Section

Attributes by Group

Provides links to attribute reference topics, grouped by function.

Attributes by Usage

Provides links to attribute reference topics, grouped by usage.

Attributes Alphabetical Reference

Provides links to all attribute reference topics.

Related Sections

Attributed Programming

Provides a conceptual overview of attributes.

Reference

Provides links to topics describing the C and C++ language references, the libraries provided with Visual C++, the Visual C++ Extensibility Object Model, and the Microsoft Macro Assembler (MASM).

Attributes Samples

Provides links to attributed versions of the ATL samples that show how to use attributes to simplify COM programming.

Visual C++ Samples

Provides links to sample code showing the capabilities of Visual C++ and the libraries and technologies it supports.

Attributed Programming Concepts

Visual C++ includes a variety of material to help you use attributes in your programs. Attributes, a new feature in Visual C++, are designed to simplify COM programming and .NET Framework common language runtime development. When you include attributes in your source files, the compiler works with provider dynamic-link libraries (DLL) to insert code or modify the code in the generated object files. There are attributes that aid in the creation of .idl files, interfaces, type libraries, and other COM elements. In the integrated development environment (IDE), attributes are supported by the wizards and by the Properties window.

While attributes eliminate some of the detailed coding needed to write COM objects, you need a background in COM fundamentals to best use them.

In This Section

Purpose of Attributes

Presents an overview of attributed programming.

Basic Mechanics of Attributes

Describes how attributes work in your project.

Building an Attributed Program

Provides information about using C++ compiler options in your project.

Attribute Categories

Provides links to the categories of attributes used in Visual C++.

Attributes Walkthroughs

Provides links to walkthroughs showing you how to use attributes in different settings.

Attribute Programmming FAQ

Answers frequently asked questions about attributed programming.

Related Sections

Attributes Reference

Provides links to reference topic describing the individual attributes and their use.

Debugging Injected Code

Describes debugging attributed programs.

__superand __interface

Links to new C++ keywords related to attributed programming.

Attributes Tutorial

Discusses developing a singleton server object using attributed code and ATL.

Adding Functionality

Provides links to topics describing conceptual information about the Visual C++ libraries and topics discussing various coding technologies and techniques.

Purpose of Attributes

Attributes extend C++ in directions not currently possible without breaking the classic structure of the language. Attributes allow providers (separate DLLs) to extend language functionality dynamically. The primary goal of attributes is to simplify the authoring of COM components, in addition to increasing the productivity level of the component developer. Attributes can be applied to nearly any C++ construct, such as classes, data members, or member functions. The following is a highlight of benefits provided by this new technology:

- Exposes a familiar and simple calling convention.
- Uses inserted code, which, unlike macros, is recognized by the debugger.
- Allows easy derivation from base classes without burdensome implementation details.
- Replaces the large amount of IDL code required by a COM component with a few concise attributes.

For example, to implement a simple event sink for a generic ATL class, you could apply the event_receiver attribute to a specific class such as `CMyReceiver`. The **event_receiver** attribute is then compiled by the Visual C++ compiler, which inserts the proper code into the object file.

```
[event_receiver(com)]
class CMyReceiver
{
   void handler1(int i) { ... }
   void handler2(int i, float j) { ... }
}
```

You can then set up the **CMyReceiver** methods `handler1` and `handler2` to handle events (using the intrinsic function __hook) from an event source, which you can create using event_source.

See Also

Other Resources

Attributed Programming Concepts

Basic Mechanics of Attributes

There are three ways to insert attributes into your project. First, you can insert them manually into your source code. Second, you can insert them using the property grid of an object in your project. Finally, you can insert them using the various wizards. For more information on using the Properties window and the various wizards, see Creating and Managing Visual C++ Projects.

Starting with Visual C++ .NET, the compiler recognizes the presence of attributes in a source file and is able to dynamically parse and verify them during compilation.

As before, when the project is built, the compiler parses each C++ source file, producing an object file. However, when the compiler encounters an attribute, it is parsed and syntactically verified. The compiler then dynamically calls an attribute provider to insert code or make other modifications at compile time. The implementation of the provider differs depending on the type of attribute. For example, ATL-related attributes are implemented by Atlprov.dll.

The following figure demonstrates the relationship between the compiler and the attribute provider.

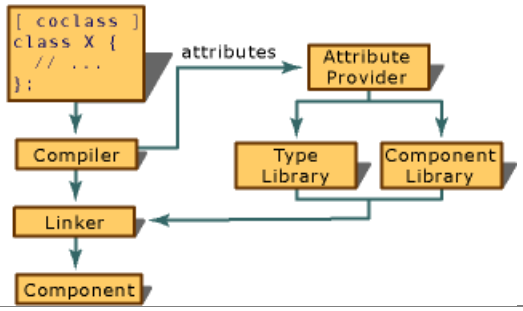

Note

Attribute usage does not alter the contents of the source file. The only time the generated attribute code is visible is during debugging sessions. In addition, for each source file in the project, you can generate a text file that displays the results of the attribute substitution. For more information on this procedure, see /Fx (Merge Injected Code) and Debugging Injected Code.

Like most C++ constructs, attributes have a set of characteristics that defines their proper usage. This is referred to as the context of the attribute and is addressed in the attribute context table for each attribute reference topic. For example, the coclass attribute can only be applied to an existing class or structure, as opposed to the cpp_quote attribute, which can be inserted anywhere within a C++ source file.

See Also

Other Resources

Attributed Programming Concepts

Building an Attributed Program

After you put Visual C++ attributes into your source code, you may want the Visual C++ compiler to produce a type library and .idl file for you. The following linker options help you build .tlb and .idl files:

- /IDLOUT
- /IGNOREIDL
- /MIDL
- /TLBOUT

Some projects contain multiple independent .idl files. These are used to produce two or more .tlb files and optionally bind them into the resource block. This scenario is not currently supported in Visual C++.

In addition, the Visual C++ linker will output all IDL-related attribute information to a single MIDL file. There will be no way to generate two type libraries from a single project.

See Also

Other Resources

Attributed Programming Concepts

Attribute Categories

The attributes reference describes the various types of attributes you can use in Visual C++.

Related Sections

COM Attributes

Injects code to support numerous areas of COM development.

IDL Attributes

Injects code that creates or modifies the associated .idl file from within a source code file without a wizard and without being familiar with the structure and syntax of that file.

ATL Server Attributes

Injects code, based on the ATL Server classes, to create a Web application or XML Web service request handler or a performance monitoring object.

OLE DB Consumer Attributes

Injects code, based on the OLE DB Consumer Templates, to create a working OLE DB consumer that performs tasks such as opening tables, executing commands, and accessing data.

Compiler Attributes

Injects code that performs a variety of common programming tasks.

Attributed Programming

Provides links to conceptual topics describing attributed programming.

Attributes Walkthroughs

These walkthroughs allow you to create programs, using both a text editor and Visual C++ wizards, that use attributes in the code.

In This Section

Creating an ActiveX Control with Attributes

Uses attributes and wizards to create a simple ActiveX control.

Developing a COM DLL with COM Attributes

Creates a COM DLL with attributes using a text editor and wizards.

Related Sections

Simplifying Operations with Database Attributes

Simplifies database operations by using attributes.

Extending Metadata with Custom Attributes

Shows code and attributes that extend the metadata of any given element.

Attributed Programming

Provides links to topics discussing attributed programming.

Adding Functionality

Provides links to topics describing conceptual information about the Visual C++ libraries and topics discussing various coding technologies and techniques.

Attributes Samples

Provides links to attributed versions of the Active Template Library (ATL) samples.

Attributes Tutorial

Discusses developing a singleton server object using attributed code and ATL.

Walkthrough: Creating an ActiveX Control with Attributes

One advantage of programming with attributes is the ease of constructing a basic framework of usable code. With this model, designing an ActiveX control is made easier by using attributes.

In addition to the attributes themselves, several Visual C++ tools take advantage of attributes. When designing an ActiveX control, the ATL Project Wizard and ATL Simple Object Wizard use attributes to develop a basic framework for an ActiveX control. The procedure consists of the following steps:

- Creating the ActiveX Control Project
- Inserting the ATL Control Component
- Adding a Property Using Attributes
- Adding an Event Using Attributes

See Also

Other Resources

Attributes Walkthroughs

Creating the ActiveX Control Project

The first step is to create the project using the Visual C++ integrated development environment. By default, the ATL Project Wizard supplies attribute support, and creates a project that uses them for basic framework implementation.

Procedures

To generate a sample project with the ATL Project Wizard

1. On the **File** menu, click **New**, and then click **Project**.

   The **New Project** dialog box appears.

2. In the Project Types pane, click **Visual C++ Projects**, and in the Templates pane, click the **ATL Project** icon.

3. In the **Name** box, enter **MyAxCtrl** and click **OK**.

   The ATL Project Wizard appears.

4. Click **Application Settings**.

   You will see that the **Attributed** check box and **Dynamic-link library (DLL)** radio button are selected by default.

5. Click **Finish** to close the ATL Project Wizard and generate your project.

A Visual C++ DLL project hosting the ActiveX control is generated. For more information on this wizard, see ATL Project Wizard.

Attributes Used in the MyAxCtrl Project

Before inserting the ATL control component, note that the `atlplus.h` file is added by the ATL Project Wizard (included in `atlbase.h`, which is in turn included by `stdafx.h`). When the implementation for an attribute is needed from an ATL provider, this file provides the information needed by the compiler.

Note
Another class of attributes have an implementation provided by the compiler and do not require any specialized header file. For more information, see COM Attributes and IDL Attributes.

The module Attribute

At this stage of development, the module attribute (located in MyAxCtrl.cpp) is the workhorse of the project. This attribute provides information on various characteristics of the target module.

In this example, the **module** attribute specifies the module type as a DLL, a unique ID for the module (making use of the uuid attribute), and a simple name for the module.

To move to the next step, see Inserting the ATL Control Component.

See Also

Tasks

Adding a Property Using Attributes

Reference

Adding an Event Using Attributes

Concepts

Walkthrough: Creating an ActiveX Control with Attributes

Other Resources

Attributes Walkthroughs

Inserting the ATL Control Component

The next step is to add the ATL control component to your project. The **Add Class** dialog box provides a variety of frameworked components for insertion into a Visual C++ IDE project. One of these components is the ATL control component, located at the top level of the **Add Class** dialog box.

Procedures

To insert the ATL control object

1. In Solution Explorer, right-click the MyAxCtrl project.
2. On the shortcut menu, click **Add**, and then click **Add Class**.

   The **Add Class** dialog box appears.
3. In the Templates pane, click the **ATL Control** icon.
4. Click **Open**.

   The ATL Control Wizard appears.
5. In the **Short Name** box, enter **MyCtl**.

   The remaining boxes are completed for you.
6. Click **Finish** to accept the default choices and add the control to your project.

For more information on this wizard, see ATL Control Wizard.

Attributes Used in the ATL Control Component

Because the ATL control component makes use of attributes, several new attributes are now present in the sample project. These attributes implement the control object and the default control interface.

In MYCTL.H, you will find a new set of attributes (referred to as an attribute block) that implements a control framework, attached to the new `CMyCtl` class. This block contains the following attributes:

- coclass Specifies that the object is a COM object and automatically provides a class factory, auto registration, and an implementation of the **IUnknown** interface.
- threading Specifies that the control is safe to be used in an apartment thread. This is done by changing the generated reference counting and registration code.
- vi_progid Specifies a version-independent form of the ProgID.
- progid Declares a simple ProgID, used by clients of the new control.
- version Specifies the version of the control.
- uuid Declares the CLSID for the control.
- helpstring Specifies a help string, describing the object.
- support_error_info Specifies that the object implements support for detailed, contextual errors.
- registration_script Specifies the custom registration script to be executed for the object.

The other major element added by the ATL Control Wizard is another attribute block (attached to the `IMyCtl` interface), implementing the default interface for the control object. This is the default interface of your control and currently the only implemented interface. The interface is declared in your control's declaration file (in the example, MYCTL.H) and is very simple:

```
__interface IMyCtl: public IDispatch
{
};
```

This code declares a custom interface, `IMyCtl`, which is derived from the standard **IDispatch** interface and contains no

methods.

This block contains the following attributes:

- object Specifies that the following interface definition is a custom interface and should be placed in the .idl file of the project.
- uuid Declares the IID for the `IMyCtl` interface.
- dual Declares the interface as a dual interface (supporting both early and late binding).
- helpstring A short description of the control object and its purpose.
- pointer_default Specifies the characteristics of any embedded pointers. In the sample control, **unique** means that embedded pointers can be **NULL** and there will be no duplicate values.

To move to the next step, see Adding a Property Using Attributes.

See Also

Reference

Adding an Event Using Attributes

Concepts

Walkthrough: Creating an ActiveX Control with Attributes

Creating the ActiveX Control Project

Other Resources

Attributes Walkthroughs

Adding a Property Using Attributes

Adding a property or method to an ActiveX control becomes a quick and simple exercise when using attributes. This topic demonstrates the steps needed to add a simple read-write custom property using attributes, and discusses the changes to the control project.

For demonstration purposes, the sample program adds a custom property, called `Data`, which gets or sets the current value of a numerical value. This property will be stored in a data member of type **short**.

To fully implement the custom property, you need to add the property to the control's default interface and to the class implementing the control. You can do this with the Add Property Wizard and the propget and propput attributes. These attributes implement a standard set of read-write property methods for the control interface. These methods are used by clients of the control to access and set the value (in this case a data string), using standard COM procedures.

To add the custom Data property

1. In Class View, expand **MyAxCtrl** and right-click the `IMyCtl` node.
2. On the shortcut menu, click **Add**, and then click **Add Property**.

   The Add Property Wizard appears.
3. For **Property type**, enter **SHORT**.
4. For **Property name**, enter **Data**.
5. Click **Finish** to implement the custom property.

The resulting modifications are as follows:

- **MyCtl.cpp** The get and set function bodies are added to the `CMyCtl` class. These will be modified later.

```
STDMETHODIMP CMyCtl::get_Data(SHORT* pVal)
{
   // TODO: Add your implementation code here
   return S_OK;
}

STDMETHODIMP CMyCtl::put_Data(SHORT newVal)
{
   // TODO: Add your implementation code here
   return S_OK;
}
```

- **MyCtl.h** The get and set methods are added to the `IMyCtl` interface.

```
[propget, id(1), helpstring("property Data")] HRESULT Data([out, retval] SHORT* pVal);
[propput, id(1), helpstring("property Data")] HRESULT Data([in] SHORT newVal);
```

- **MyCtl.h** The get and set methods declarations are added to the `CMyCtl` class.

```
STDMETHOD(get_Data)(SHORT* pVal);
STDMETHOD(put_Data)( SHORT newVal);
```

To fully implement the Data property, a data member must be added to the control class (storing the current value) and code added to the get/put functions to retrieve and update the property value.

To complete the implementation of the custom Data property

1. In Class View, right-click the `CMyCtl` node.

2. On the shortcut menu, click **Add**, and then click **Add Variable**.

   The Add Variable Wizard appears.

3. For **Variable type**, enter **short**.

4. For **Variable name**, enter **m_iData**.

5. Click **Finish**.

6. In **MyCtl.cpp**, modify the `get_Data` function by replacing the comment with the following code:

```
*pVal= m_iData;
```

7. Modify the `put_Data` function by replacing the comment with the following code:

```
m_iData= newVal;
```

8. Build the solution.

To move to the next step, see Adding an Event Using Attributes.

See Also

Concepts

Walkthrough: Creating an ActiveX Control with Attributes
Creating the ActiveX Control Project
Inserting the ATL Control Component

Other Resources

Attributes Walkthroughs

Adding an Event Using Attributes

Events allow an ActiveX control to communicate important information to its container. Using attributes, adding an event is a simple matter. For detailed information on adding and firing events from an ATL control, see Event Handling in COM.

Conclusion

In the past, implementing the standard features of an ActiveX control could be a tedious task. However, using the attributed programming model, the various wizards of Visual C++, and the Visual C++ compiler itself, implementing a basic ActiveX control now requires little effort, and makes use of code that has been tested and optimized by the Visual C++ team.

See Also

Tasks

Adding a Property Using Attributes

Concepts

Walkthrough: Creating an ActiveX Control with Attributes

Creating the ActiveX Control Project

Inserting the ATL Control Component

Other Resources

Attributes Walkthroughs

Developing a COM DLL with COM Attributes

The following topics demonstrate how to quickly develop a simple COM server (an inproc server or DLL) using various attributes. You can develop the COM server in two ways:

- Walkthrough: Creating a COM Server Using a Text Editor
- Walkthrough: Creating a COM Server Using Wizards

The first walkthrough develops the COM server using a text editor and the command-line tools. The second walkthrough develops the same COM server using the Visual Studio integrated development environment.

Note
The UUID used in this topic is purely for demonstration purposes only. If omitted, the **module** attribute (and other attributes) will automatically generate one for you.

This topic focuses on the following attributes:

coclass	object	threading
dual	out	uuid
emitidl	pointer_default	version
helpstring	progid	vi_progid
module	retval	

For a complete list of supported attributes, see Attributes by Group. For more information on attributed programming, see Attributed Programming.

See Also

Other Resources

Attributes Walkthroughs

Walkthrough: Creating a COM Server Using a Text Editor

Attributes were designed to ease the tedium and complexity of traditional COM programming. Using attributes in your applications can automate or simplify several common COM procedures. In this walkthrough you will develop a simple COM server with a text editor and the command-line tools.

To use attributes in your applications, you must add support for them by defining several related macros (such as _ATL_ATTRIBUTES) and including related header files.

To create the header file

1. Open Notepad or your text editor.
2. Create a new file called MyIncludes.h.
3. Add the following code:

```
#pragma once
#define STRICT
#ifndef _WIN32_WINNT
#define _WIN32_WINNT 0x0400
#endif
#define _ATL_ATTRIBUTES
#define _ATL_APARTMENT_THREADED
#define _ATL_NO_AUTOMATIC_NAMESPACE
#include <atlbase.h>
#include <atlcom.h>
#include <atlwin.h>
#include <atltypes.h>
#include <atlctl.h>
#include <atlhost.h>
using namespace ATL;
```

4. Save the file.

Include files beginning with "atl" bring in ATL support for the application, and with the definition of _ATL_ATTRIBUTES, support for attributed ATL programming is also added.

You will need to create a source file that implements the inproc server.

To create the source file

1. In Notepad, create a new file called MyServer.cpp.
2. Add the following code:

```
#include "MyIncludes.h"
// The module attribute is specified in order to implement DllMain,
// DllRegisterServer and DllUnregisterServer
[ module(dll, name = "MyServer", helpstring = "MyServer 1.0 Type Library") ];
[ emitidl ];
```

3. Save the file.

Using the module attribute saves you the work of implementing the necessary functions required for a COM inproc server. In addition, you do not need to write a .idl file or the .def file (which exports the functions of your server). This is all done automatically by the **module** attribute.

To build the project

- At the command line, enter the following commands:

```
cl /LD MyServer.cpp
regsvr32 MyServer.dll
```

If you encounter any errors, then make sure you have Visual Studio properly installed and the environment variables defined. The Visual Studio environment variables are automatically defined when you run the **Visual Studio Command Prompt**, or they can be set manually by running **vsvars32.bat** found at <Visual Studio Installation Directory>\Common7\Tools\vsvars32.bat.

Building the project successfully registers the server with the operating system.

At this point, the COM server does not expose any functionality to the client. This is because the COM server does not implement any server objects. In the next step, you will add server objects to this server.

To add server objects to the server

1. Open MyServer.cpp and add the following code:

```
/////////////////////////////////////////////////////////////////////////////
// IObject1
[
   object,
   uuid("103FF9D9-8BC9-4ea8-8CD4-C1E627D04358"),
   dual,
   helpstring("IObject1 Interface"),
   pointer_default(unique)
]
__interface IObject1 : IDispatch
{
   HRESULT GetANum([out, retval]int* pInt);
};
/////////////////////////////////////////////////////////////////////////////
// CObject1
[
   coclass,
   threading(apartment),
   vi_progid("MyServer.Object1"),
   progid("MyServer.Object1.1"),
   version(1.0),
   uuid("15615078-523C-43A0-BE6F-651E78A89213"),
   helpstring("Object1 Class")
]
class ATL_NO_VTABLE CObject1 :
   public IObject1
{
public:
   CObject1()
   {
   }
   HRESULT GetANum(int* pInt){
      *pInt = 101;
      return S_OK;
   }
   DECLARE_PROTECT_FINAL_CONSTRUCT()
   HRESULT FinalConstruct()
   {
      return S_OK;
   }
```

```
    void FinalRelease()
    {
    }
};
```

2. Save your changes and examine the new code.

The code creates a simple COM object (`CObject1`) that implements a custom interface (`IObject1`) with a single method. This method (`GetANum`) returns the value of a data member (of type `int`).

Save your changes and rebuild the application, using the commands from the previous step:

```
cl /LD MyServer.cpp
regsvr32 MyServer.dll
```

To determine if the server works, a test client application must be written.

To create a test client application

1. Open Notepad and create a new file named Comtest.cpp.

   Comtest.cpp needs to be in the same folder as MyServer.cpp.

2. Add the following code to Comtest.cpp:

```
#include <iostream>
#include "atlbase.h"
#import "vc80.tlb" no_namespace
using namespace std;
int main()
{
   CoInitialize(NULL);
   {
      CComPtr<IUnknown> spUnknown;
            spUnknown.CoCreateInstance(__uuidof(CObject1));
      CComPtr<IObject1> pI;
      spUnknown.QueryInterface(&pI);
      int res = 0;
      res = pI->GetANum();
      cout << res << endl;
   }
   CoUninitialize();
}
```

For the client to know what the server has to offer, you must import the type library of the server by using the **#import** keyword. Because no specific name was given when the server was created, the compiler generated a type library with a default name (`vc80.tlb`). The following line in the code previously shown imports the library:

```
#import "vc80.tlb" no_namespace
```

When this file is imported into the test application, the client has access to the type information in the COM server. This allows the test application to refer to any types defined within (such as `CObject1` and `IObject1`).

To build and run the test application

1. At the command line, enter the following command:

```
cl /EHsc comtest.cpp
```

2. Run the application by entering the following command:

```
comtest
```

You will see the integer value being printed.

See Also

Tasks

Walkthrough: Creating a COM Server Using Wizards

Reference

Developing a COM DLL with COM Attributes

Walkthrough: Creating a COM Server Using Wizards

In this walkthrough, you will use the Visual Studio integrated development environment (IDE) and the various wizards to create the same ATL COM server.

To create a simple ATL COM server

1. On the **File** menu, click **New**, and then click **Project**.

   The **New Project** dialog box appears.

2. In the Project Types pane, click **Visual C++**, and in the Templates pane, click the **ATL Project** icon.
3. In the **Name** box, enter **MyServer**.
4. Click **OK**.

   The ATL Project Wizard appears.

5. Click **Application Settings**.

   You will see that the **Attributed** check box is not selected by default. Click the check box to select it.

6. Click **Finish** to close the wizard and generate the project.

   The result is an inproc ATL COM server without any server objects.

7. On the **Build** menu, click **Build Solution**.

   Building the project successfully registers the server with the operating system.

To add and implement a server object

1. In Solution Explorer, right-click the **MyServer** project
2. On the shortcut menu, click **Add**, and then click **Add Class**.

   The **Add Class** dialog box appears.

3. In the Templates pane, click the **ATL Simple Object** item and click **Open**.

   The ATL Simple Object Wizard appears.

4. In the **Short Name** text box, enter **Object1**.
5. Click **Finish** to accept the remaining default values.
6. In Class View, right-click the **IObject1** node.
7. On the shortcut menu, click **Add**, and then click **Add Property**.
8. For **Property type**, enter **SHORT**.
9. For **Property name**, enter **GetANum**.
10. Click **Finish**.
11. In the body of the `get_GetANum` method of `CObject1`, replace the comment with the following code:

    ```
    *pVal= 101;
    ```

12. On the **Build** menu, click **Build Solution**.

To create the test client application

1. In Solution Explorer, right-click the solution node and select **Add**, then select **New Project**.

   The **New Project** dialog box appears.

2. In the Project Types pane, click **Visual C++**, and in the **Templates** pane, click the **Win32 Project** icon.
3. In the **Name** text box, enter **COMTest**.
4. Click **OK**.

   The Win32 Application Wizard appears.
5. Click **Application Settings** and select **Console application**.
6. Click **Finish** to close the dialog box and generate the project.
7. In Solution Explorer, double-click the **COMTest.cpp**.
8. Replace the default code with the following:

```
#include "stdafx.h"
#include <iostream>
#include "atlbase.h"
#import "..\MyServer\_MyServer.tlb" no_namespace
using namespace std;
struct OleCom {
      OleCom() { CoInitialize(NULL);}
      ~OleCom() { CoUninitialize(); }
   }olecom;
   int _tmain(int argc, _TCHAR* argv[])
   {
      CComPtr<IUnknown> spUnknown;
      spUnknown.CoCreateInstance(__uuidof(CObject1));
      CComPtr<IObject1> pI;
      spUnknown.QueryInterface(&pI);
      short res = 0;
      pI->get_GetANum(&res);
      cout << res << endl;
      return 0;
   }
```

9. On the **Build** menu, click **Build Solution**.

To run the test application

1. At the command line, change to the COMTest\Debug directory.
2. Run the application by entering the following command:

```
comtest
```

   You will see the integer value being printed.

See Also

Tasks

Walkthrough: Creating a COM Server Using a Text Editor

Reference

Developing a COM DLL with COM Attributes

Attribute Programming FAQ

This topic answers the following frequently asked questions:

- What is an HRESULT?
- When do I have to specify the parameter name for an attribute?
- Can I use comments in an attribute block?
- How do attributes interact with inheritance?
- How can I use attributes in a nonattributed ATL project?
- How can I use an .idl file in an attributed project?
- Can I modify code that is injected by an attribute?
- How can I forward declare an attributed interface?
- Can I use attributes on a class derived from a class that also uses attributes?

What is an HRESULT?

An **HRESULT** is a simple data type that is often used as a return value by attributes and ATL in general. The following table describes the various values. More values are contained in the header file winerror.h.

Name	Description	Value
S_OK	Operation successful	0x00000000
E_UNEXPECTED	Unexpected failure	0x8000FFFF
E_NOTIMPL	Not implemented	0x80004001
E_OUTOFMEMORY	Failed to allocate necessary memory	0x8007000E
E_INVALIDARG	One or more arguments are invalid	0x80070057
E_NOINTERFACE	No such interface supported	0x80004002
E_POINTER	Invalid pointer	0x80004003
E_HANDLE	Invalid handle	0x80070006
E_ABORT	Operation aborted	0x80004004
E_FAIL	Unspecified failure	0x80004005
E_ACCESSDENIED	General access denied error	0x80070005

When do I have to specify the parameter name for an attribute?

In most cases, if the attribute has a single parameter, that parameter is named. This name is not required when inserting the attribute in your code. For example, the following usage of the aggregatable attribute:

```
[coclass, aggregatable(value=allowed)]
class CMyClass
{
// The class declaration
};
```

is exactly the same as:

```
[coclass, aggregatable(allowed)]
class CMyClass
{
// The class declaration
};
```

However, the following attributes have single, unnamed parameters:

call_as	case	cpp_quote
default	defaultvalue	defaultvtable
emitidl	entry	first_is
helpcontext	helpfile	helpstring
helpstringcontext	helpstringdll	id
iid_is	import	importlib
include	includelib	last_is
length_is	max_is	no_injected_text
pointer_default	pragma	restricted
size_is	source	switch_is
switch_type	transmit_as	wire_marshal

Can I use comments in an attribute block?

You can use both single-line and multiple-line comments within an attribute block. However, you cannot use either style of comment within the parentheses holding the parameters to an attribute.

The following is allowed:

```
[ coclass,
   progid("MyClass.CMyClass.1"), /* Multiple-line
                                    comment */
   threading("both") // Single-line comment
]
```

The following is disallowed:

```
[ coclass,
   progid("MyClass.CMyClass.1" /* Multiple-line comment */ ),
   threading("both" // Single-line comment)
]
```

How do attributes interact with inheritance?

You can inherit both attributed and unattributed classes from other classes, which may themselves be attributed or not. The result of deriving from an attributed class is the same as deriving from that class after the attribute provider has transformed its code. Attributes are not transmitted to derived classes through C++ inheritance. An attribute provider only transforms code in the vicinity of its attributes.

How can I use attributes in a nonattributed ATL project?

You may have a nonattributed ATL project, which has an .idl file, and you may want to start adding attributed objects. In this case, use the Add Class Wizard to provide the code.

How can I use an .idl file in an attributed project?

You may have a .idl file that you want to use in your ATL attributed project. In this case, you would use the importidl attribute, compile the .idl file to a .h file (see the MIDL Property Pages in the project's Property Pages dialog box), and then include the .h file in your project.

Can I modify code that is injected by an attribute?

Some attributes inject code into your project. You can see the injected code by using the /Fx compiler option. It is also possible to copy code from the injected file and paste it into your source code. This allows you to modify the behavior of the attribute. However, you may have to modify other parts of your code as well.

The following sample is the result of copying injected code into a source code file:

```
// attr_injected.cpp
// compile with: comsupp.lib
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>

[ module(name="MyLibrary") ];

// ITestTest
[
   object,
   uuid("DADECE00-0FD2-46F1-BFD3-6A0579CA1BC4"),
   dual,
   helpstring("ITestTest Interface"),
   pointer_default(unique)
]

__interface ITestTest : IDispatch {
   [id(1), helpstring("method DoTest")]
   HRESULT DoTest([in] BSTR str);
};

// _ITestTestEvents
[
   uuid("12753B9F-DEF4-49b0-9D52-A79C371F2909"),
   dispinterface,
   helpstring("_ITestTestEvents Interface")
]

__interface _ITestTestEvents {
   [id(1), helpstring("method BeforeChange")] HRESULT BeforeChange([in] BSTR str, [in,out]
VARIANT_BOOL* bCancel);
};

// CTestTest
[
   coclass,
   threading(apartment),
   vi_progid("TestATL1.TestTest"),
   progid("TestATL1.TestTest.1"),
   version(1.0),
   uuid("D9632007-14FA-4679-9E1C-28C9A949E784"),
   // this line would be commented out from original file
   // event_source("com"),
   // this line would be added to support injected code
   source(_ITestTestEvents),
   helpstring("TestTest Class")
]
```

```
class ATL_NO_VTABLE CTestTest : public ITestTest,
// the following base classes support added injected code
public IConnectionPointContainerImpl<CTestTest>,
public IConnectionPointImpl<CTestTest, &__uuidof(::_ITestTestEvents), CComDynamicUnkArray>
{
public:
   CTestTest() {
   }
   // this line would be commented out from original file
   // __event __interface _ITestTestEvents;
   DECLARE_PROTECT_FINAL_CONSTRUCT()
   HRESULT FinalConstruct() {
      return S_OK;
   }

void FinalRelease() {}

public:
   CComBSTR m_value;
   STDMETHOD(DoTest)(BSTR str) {
      VARIANT_BOOL bCancel = FALSE;
      BeforeChange(str,&bCancel);
      if (bCancel) {
          return Error("Error : Someone don't want us to change the value");
      }

     m_value =str;
     return S_OK;
    }
// the following was copied in from the injected code.
HRESULT BeforeChange(::BSTR i1,::VARIANT_BOOL* i2) {
   HRESULT hr = S_OK;
   IConnectionPointImpl<CTestTest, &__uuidof(_ITestTestEvents), CComDynamicUnkArray>* p = t
his;
   VARIANT rgvars[2];
   Lock();
   IUnknown** pp = p->m_vec.begin();
   Unlock();
   while (pp < p->m_vec.end()) {
      if (*pp != NULL) {
         IDispatch* pDispatch = (IDispatch*) *pp;
         ::VariantInit(&rgvars[1]);
         rgvars[1].vt = VT_BSTR;
         V_BSTR(&rgvars[1])= (BSTR) i1;
         ::VariantInit(&rgvars[0]);
         rgvars[0].vt = (VT_BOOL | VT_BYREF);
         V_BOOLREF(&rgvars[0])= (VARIANT_BOOL*) i2;
         DISPPARAMS disp = { rgvars, NULL, 2, 0 };
         VARIANT ret_val;
         hr = __ComInvokeEventHandler(pDispatch, 1, 1, &disp, &ret_val);
         if (FAILED(hr))
            break;
      }
      pp++;
   }
   return hr;
}

BEGIN_CONNECTION_POINT_MAP(CTestTest)
CONNECTION_POINT_ENTRY(__uuidof(::_ITestTestEvents))
END_CONNECTION_POINT_MAP()
// end added code section

// _ITestCtrlEvents Methods
public:
};

int main() {}
```

How can I forward declare an attributed interface?

If you are going to make a forward declaration of an attributed interface, you must apply the same attributes to the forward declaration that you apply to the actual interface declaration. You must also apply the export attribute to your forward declaration.

Can I use attributes on a class derived from a class that also uses attributes?

No, using attributes on a class derived from a class that also uses attributes is not supported.

See Also

Other Resources

Attributed Programming Concepts

Attributes by Group

The C++ attributes are organized into the following functional groups:

Attribute	Description
COM Attributes	Injects code to support numerous areas of COM development and .NET Framework common language runtime development.
IDL Attributes	Lets you modify the .idl file from within a source code file without a wizard and without being familiar with the structure and syntax of that file.
ATL Server Attributes	Injects code, based on the ATL Server classes, to create a Web application or XML Web service request handler, or a performance monitoring object.
OLE DB Consumer Attributes	Injects code, based on the OLE DB Consumer Templates, to create a working OLE DB consumer that performs tasks such as opening tables, executing commands, and accessing data.
Compiler Attributes	Attributes supplied by the Visual C++ compiler.

For information on defining custom attributes for your .NET application, see User-Defined Attributes.

See Also

Other Resources

C++ Attributes Reference
Attributed Programming Concepts
Attributes by Usage
Attributes Alphabetical Reference

COM Attributes

The COM attributes inject code to support numerous areas of COM development and .NET Framework common language runtime development. These areas range from custom interface implementation and support of existing interfaces to supporting stock properties, methods, and events. In addition, support can be found for composite and ActiveX control implementation.

Attribute	Description
aggregatable	Indicates that a control can be aggregated by another control.
aggregates	Indicates that a control aggregates the target class.
coclass	Creates a COM object, which can implement a COM interface.
com_interface_entry	Adds an interface entry to a COM map.
implements_category	Specifies implemented component categories for the class.
perf_counter	Adds performance monitor counter support to a class member variable.
perfmon	Adds performance monitor support to a class.
progid	Defines the ProgID for a control.
rdx	Creates or modifies a registry key.
registration_script	Executes the specified registration script.
requires_category	Specifies required component categories for the class.
soap_handler	Provides the methods necessary for handling SOAP method calls and exposing information about the services offered by this class through SDL.
soap_method	In an XML Web service, exposes the specified member as a public SOAP method.
support_error_info	Supports error reporting for the target object.
synchronize	Synchronizes access to a method.
tag_name	In a request handler, exposes it as a replacement method associated with a tag name.
threading	Specifies the threading model for a COM object.
vi_progid	Defines a version-independent ProgID for a control.

See Also

Other Resources

Attributes by Group

IDL Attributes

Traditionally, maintaining an .idl file meant that you had to:

- Be familiar with the structure and syntax of an .idl file to be able to modify it.
- Rely on a wizard, which would let you modify some aspects of the .idl file.

Now, you can modify the .idl file from within a source code file using Visual C++ IDL attributes. In many cases, Visual C++ IDL attributes have the same name as MIDL attributes. When the name of a Visual C++ IDL attribute and a MIDL attribute are the same, it means that putting the Visual C++ attribute in your source code file will result in an .idl file that contains its namesake MIDL attribute. However, a Visual C++ IDL attribute may not provide all the functionality of a MIDL attribute.

When not used with COM attributes, IDL attributes let you define interfaces. When the source code is compiled, the attributes are used to define the generated .idl file. When used with COM attributes in an ATL project, some IDL attributes, such as **coclass**, cause code to be injected into the project.

Note that idl_quote lets you use MIDL constructs that are not supported in the current version of Visual C++. This and other attributes such as importlib and includelib help you to use existing .idl files in your current Visual C++ project.

Attribute	Description
aggregatable	Indicates that a control can be aggregated by another control.
appobject	Identifies the coclass as an application object, which is associated with a full EXE application, and indicates th at the functions and properties of the coclass are globally available in this type library.
async_uuid	Specifies the UUID that directs the MIDL compiler to define both synchronous and asynchronous versions of a COM interface.
bindable	Indicates that the property supports data binding.
call_as	Enables a nonremotable function to be mapped to a remote function.
case	Used with the switch_type attribute in a union.
coclass	Places class definition into an .idl file as coclass.
control	Specifies that the user-defined type is a control.
cpp_quote	Emits the specified string, without the quote characters, into the generated header file.
defaultbind	Indicates the single, bindable property that best represents the object.
defaultcollelem	Used for Visual Basic code optimization.
defaultvalue	Allows specification of a default value for a typed optional parameter.
default	Indicates that the custom or dispinterface defined within a coclass represents the default programmability in terface.
defaultvtable	Defines an interface as the default vtable interface for a control.
dispinterface	Places an interface in the .idl file as a dispatch interface.
displaybind	Indicates a property that should be displayed to the user as bindable.

dual	Places an interface in the .idl file as a dual interface.
entry	Specifies an exported function or constant in a module by identifying the entry point in the DLL.
first_is	Specifies the index of the first array element to be transmitted.
helpcontext	Specifies a context ID that lets the user view information about this element in the Help file.
helpfile	Sets the name of the Help file for a type library.
helpstringcontext	Specifies the ID of a help topic in an .hlp or .chm file.
helpstringdll	Specifies the name of the DLL to use to perform document string lookup (localization).
helpstring	Specifies a character string that is used to describe the element to which it applies.
hidden	Indicates that the item exists but should not be displayed in a user-oriented browser.
idl_module	Specifies an entry point in a DLL.
idl_quote	Allows you to use attributes or IDL constructs that are not supported in the current version of Visual C++.
id	Specifies a DISPID for a member function (either a property or a method, in an interface or dispinterface).
iid_is	Specifies the IID of the COM interface pointed to by an interface pointer.
immediatebind	Indicates that the database will be notified immediately of all changes to a property of a data-bound object.
importlib	Makes types that have already been compiled into another type library available to the type library being cre ated.
import	Specifies another .idl, .odl, or header file containing definitions you want to reference from your main .idl file.
include	Specifies one or more header files to be included in the generated .idl file.
includelib	Causes an .idl or .h file to be included in the generated .idl file.
in	Indicates that a parameter is to be passed from the calling procedure to the called procedure.
last_is	Specifies the index of the last array element to be transmitted.
lcid	Lets you pass a locale identifier to a function.
length_is	Specifies the number of array elements to be transmitted.
licensed	Indicates that the coclass to which it applies is licensed, and must be instantiated using **IClassFactory2**.
local	Allows you to use the MIDL compiler as a header generator when used in the interface header. When used in an individual function, designates a local procedure for which no stubs are generated.
max_is	Designates the maximum value for a valid array index.
module	Defines the library block in the .idl file.

ms_union	Controls the network data representation alignment of nonencapsulated unions.
no_injected_text	Prevents the compiler from injecting code as a result of attribute use.
nonbrowsable	Indicates that an interface member should not be displayed in a property browser.
noncreatable	Defines an object that cannot be instantiated by itself.
nonextensible	Specifies that the **IDispatch** implementation includes only the properties and methods listed in the interface description and cannot be extended with additional members at run time.
object	Identifies a custom interface; synonymous with custom attribute.
odl	Identifies an interface as an Object Description Language (ODL) interface.
oleautomation	Indicates that an interface is compatible with Automation.
optional	Specifies an optional parameter for a member function.
out	Identifies pointer parameters that are returned from the called procedure to the calling procedure (from the server to the client).
pointer_default	Specifies the default pointer attribute for all pointers except top-level pointers that appear in parameter lists.
pragma	Emits the specified string, without the quote characters, into the generated .idl file.
progid	Specifies the ProgID for a COM object.
propget	Specifies a property accessor (get) function.
propputref	Specifies a property setting function that uses a reference instead of a value.
propput	Specifies a property setting function.
ptr	Designates a pointer as a full pointer.
public	Ensures that a typedef will go into the type library even if it is not referenced from within the .idl file.
range	Specifies a range of allowable values for arguments or fields whose values are set at run time.
readonly	Prohibits assignment to a variable.
ref	Identifies a reference pointer.
requestedit	Indicates that the property supports the **OnRequestEdit** notification.
restricted	Specifies that a library, or member of a module, interface, or dispinterface cannot be called arbitrarily.
retval	Designates the parameter that receives the return value of the member.
size_is	Specifies the size of memory allocated for sized pointers, sized pointers to sized pointers, and single- or mult idimensional arrays.

source	Indicates that a member of a class, property, or method is a source of events.
string	Indicates that the one-dimensional **char**, **wchar_t**, **byte**, or equivalent array or the pointer to such an array must be treated as a string.
switch_is	Specifies the expression or identifier acting as the union discriminant that selects the union member.
switch_type	Identifies the type of the variable used as the union discriminant.
transmit_as	Instructs the compiler to associate a presented type, which client and server applications manipulate, with a t ransmitted type.
uidefault	Indicates that the type information member is the default member for display in the user interface.
unique	Specifies a unique pointer.
usesgetlasterror	Tells the caller that if there is an error when calling that function, the caller can then call **GetLastError** to retri eve the error code.
uuid	Specifies the unique ID for a class or interface.
v1_enum	Directs that the specified enumerated type be transmitted as a 32-bit entity, rather than the 16-bit default.
vararg	Specifies that the function take a variable number of arguments.
vi_progid	Specifies a version-independent form of the ProgID.
wire_marshal	Specifies a data type that will be used for transmission instead of an application-specific data type.

See Also

Concepts

Attribute Limitations

Other Resources

Attributes by Group

ATL Server Attributes

The ATL Server attributes inject code, based on the ATL Server classes, to create a Web application or XML Web service request handler or a performance monitoring object.

Attribute	Description
perf_counter	Apply this attribute to a data member in a perf_object class to expose it as a performance counter.
perf_object	Apply this attribute to a class to define a performance monitor object.
perfmon	Apply this attribute to a class to define a performance manager object.
request_handler	Apply this attribute to a class to expose it as an ATL Server request handler and enable it to handle HTTP requests.
soap_handler	Apply this attribute to a class to provide the methods necessary for handling SOAP method calls and exposing information about the services offered by this class through WSDL.
soap_header	Apply this attribute to a SOAP method in an XML Web service to specify the data member used to hold the value of a SOAP header.
soap_method	Apply this attribute to a method in an XML Web service to expose the specified member as a SOAP method with a corresponding WSDL description.
tag_name	Apply this attribute to a method in a request handler to expose it as a replacement method associated with a tag name.

See Also

Other Resources

Attributes by Group
ATL Server Concepts
ATL Server Reference
ATL Server Samples
ATL Attributes Samples

OLE DB Consumer Attributes

The OLE DB consumer attributes inject code, based on the OLE DB Consumer Templates, to create a working OLE DB consumer that performs tasks such as opening tables, executing commands, and accessing data.

Attribute	Description
db_accessor	Binds columns in a rowset and binds them to the corresponding accessor maps.
db_column	Binds a specified column to the rowset.
db_command	Executes an OLE DB command.
db_param	Associates the specified member variable with an input or output parameter.
db_source	Creates and encapsulates a connection, through a provider, to a data source.
db_table	Opens an OLE DB table.

See Also

Other Resources

Attributes by Group

Compiler Attributes

Compiler attributes provide a variety of functionality.

Attribute	Description
emitidl	Determines whether all subsequent IDL attributes will be processed and placed in the generated .idl file.
event_receiver	Creates an event receiver.
event_source	Creates an event source.
export	Causes a data structure to be placed in the .idl file.
implements	Specifies dispatch interfaces that are forced to be members of the IDL coclass.
importidl	Inserts the specified .idl file into the generated .idl file.
importlib	Makes types that have already been compiled into another type library available to the type library being created.
includelib	Causes an .idl or .h file to be included in the generated .idl file.
library_block	Places a construct inside the .idl file's library block.
no_injected_text	Prevents the compiler from injecting code as a result of attribute use.
satype	Specifies the data type of the **SAFEARRAY**.
version	Identifies a particular version among multiple versions of an interface or class.

See Also

Other Resources

Attributes by Group

Attributes by Usage

This topic lists attributes according to the C++ language elements to which they apply.

If an attribute precedes a Visual C++ element that is not in the attribute's scope, the attribute block is treated as a comment.

Attribute	Description
Module Attributes	Applies to the module attribute.
Interface Attributes	Applies to the __interface C++ keyword.
Class Attributes	Applies to the C++ keyword.
Method Attributes	Applies to the methods in a class, coclass, or interface.
Parameter Attributes	Applies to parameters of a method in a class or interface.
Data Member Attributes	Applies to the data members in a class, coclass, or interface.
Typedef, Enum, Union, and Struct Attributes	Applies to the C++ keywords.
Array Attributes	Applies to arrays or **SAFEARRAY**s.
Stand-Alone Attributes	Operates more like a line of code but does not operate on a C++ keyword. Stand-alone attribute statements require a semicolon at the end of the line.

See Also

Other Resources

C++ Attributes Reference
Attributed Programming Concepts
Attributes by Group
Attributes Alphabetical Reference

Module Attributes

The following attribute can only be applied to the module attribute.

Attribute	Description
helpstringdll	Specifies the name of the DLL to use to perform document string lookup (localization).

See Also

Other Resources

Attributes by Usage

Interface Attributes

The following attributes apply to the interface (or __interface) C++ keyword.

Attribute	Description
async_uuid	Specifies the UUID that directs the MIDL compiler to define both synchronous and asynchronous versions of a COM interface.
custom	Lets you define your own attributes.
dispinterface	Places an interface in the .idl file as a dispatch interface.
dual	Places an interface in the .idl file as a dual interface.
export	Causes a data structure to be placed in the .idl file.
helpcontext	Specifies a context ID that lets the user view information about this element in the Help file.
helpfile	Sets the name of the Help file for a type library.
helpstring	Specifies a character string that is used to describe the element to which it applies.
helpstringcontext	Specifies the ID of a help topic in an .hlp or .chm file.
helpstringdll	Specifies the name of the DLL to use to perform document string lookup (localization).
hidden	Indicates that the item exists but should not be displayed in a user-oriented browser.
library_block	Places a construct inside the .idl file's library block.
local	Allows you to use the MIDL compiler as a header generator when used in the interface header. When used in an individual function, designates a local procedure for which no stubs are generated.
nonextensible	Specifies that the **IDispatch** implementation includes only the properties and methods listed in the interface description and cannot be extended with additional members at run time. This attribute is only valid on a dual interface.
odl	Identifies an interface as an Object Description Language (ODL) interface.
object	Identifies a custom interface.
oleautomation	Indicates that an interface is compatible with Automation.
pointer_default	Specifies the default pointer attribute for all pointers except top-level pointers that appear in parameter lists.
ptr	Designates a pointer as a full pointer.
restricted	Designates which members of the library cannot be called arbitrarily.
uuid	Provides the unique ID for the library

You must observe these rules for defining an interface:

- Default calling convention is __stdcall.

- A GUID is supplied for you if you do not supply one.
- No overloaded methods are allowed.

When not specifying the uuid attribute and using the same interface name in different attribute projects, the same GUID is generated.

See Also

Other Resources

Attributes by Usage

Class Attributes

The following attributes apply to the class C++ keyword.

Attribute	Description
aggregatable	Indicates that the class supports aggregation.
aggregates	Indicates that a control aggregates the target class.
appobject	Identifies the coclass as an application object, which is associated with a full .exe application, and indicates that the functions and properties of the coclass are globally available in this type library.
case	Used with the switch_type attribute in a union.
coclass	Creates an ActiveX control.
com_interface_entry	Adds an interface entry to a COM map.
control	Specifies that the user-defined type is a control.
custom	Lets you define your own attribute.
db_command	Creates an OLE DB command.
db_param	Associates the specified member variable with an input or output parameter and delimits the variable.
db_source	Creates a connection to a data source.
db_table	Opens an OLE DB table.
default	Indicates that the custom or dispinterface defined within a coclass represents the default programmability interface.
defaultvtable	Defines an interface as the default vtable interface for a control.
event_receiver	Creates an event receiver.
event_source	Creates an event source.
helpcontext	Specifies a context ID that lets the user view information about this element in the Help file.
helpfile	Sets the name of the Help file for a type library.
helpstringcontext	Specifies the ID of a help topic in an .hlp or .chm file.
helpstring	Specifies a character string that is used to describe the element to which it applies.
hidden	Indicates that the item exists but should not be displayed in a user-oriented browser.
implements	Specifies dispatch interfaces that are forced to be members of the IDL coclass.
implements_category	Specifies implemented component categories for the class.

module	Defines the library block in the .idl file.
noncreatable	Defines an object that cannot be instantiated by itself.
perf_object	Adds performance monitor object support to a class.
perfmon	Adds performance monitor support to a class.
progid	Defines the ProgID for a control.
registration_script	Executes the specified registration script.
request_handler	Add this attribute to a class to expose it as an ATL Server request handler and enable it to handle HTTP requests.
requestedit	Indicates that the property supports the **OnRequestEdit** notification.
soap_handler	Add this attribute to a class to provide the methods necessary for handling SOAP method calls and exposing information about the services offered by this class through SDL.
source	Specifies the control's source interfaces for connection points on a class. On a property or method, the **source** attribute indicates that the member returns an object or VARIANT that is a source of events.
support_error_info	Supports error reporting for the target object.
threading	Specifies the threading model for a control.
uuid	Specifies the unique ID for a class or interface.
version	Identifies a particular version among multiple versions of a class.
vi_progid	Specifies a version-independent form of the ProgID.

See Also

Other Resources

Attributes by Usage

Method Attributes

The following attributes apply to the methods in a class, coclass, or interface.

Attribute	Description
bindable	Indicates that the property supports data binding.
call_as	Enables a nonremotable function to be mapped to a remote function.
custom	Lets you define your own attribute.
db_column	Binds a specified column to the rowset.
db_command	Creates an OLE DB command.
db_param	Associates the specified member variable with an input or output parameter and delimits the variable.
db_source	Creates a connection to a data source.
db_table	Opens an OLE DB table.
defaultbind	Indicates the single, bindable property that best represents the object.
defaultcollelem	Used for Visual Basic code optimization.
displaybind	Indicates a property that should be displayed to the user as bindable.
helpcontext	Specifies a context ID that lets the user view information about this element in the Help file.
helpfile	Sets the name of the Help file for a type library.
helpstring	Specifies a character string that is used to describe the element to which it applies.
helpstringcontext	Specifies the ID of a help topic in an .hlp or .chm file.
helpstringdll	Specifies the name of the DLL to use to perform document string lookup (localization).
hidden	Indicates that the item exists but should not be displayed in a user-oriented browser.
id	Specifies a DISPID for a member function (either a property or a method, in an interface or dispinterface).
immediatebind	Indicates that the database will be notified immediately of all changes to a property of a data-bound object.
in	Indicates that a parameter is to be passed from the calling procedure to the called procedure.
local	Allows you to use the MIDL compiler as a header generator when used in the interface header. When used in an individual function, designates a local procedure for which no stubs are generated.
nonbrowsable	Indicates that an interface member should not be displayed in a property browser.
propget	Specifies a property accessor function.

propput	Specifies a property-setting function.
propputref	Specifies a property-setting function that uses a reference instead of a value.
ptr	Designates a pointer as a full pointer.
range	Specifies a range of allowable values for arguments or fields whose values are set at run time.
requestedit	Indicates that the property supports the **OnRequestEdit** notification.
restricted	Specifies that a member of a module, interface, or dispinterface cannot be called arbitrarily.
satype	Specifies the data type of the **SAFEARRAY** structure.
soap_method	Apply this attribute to a method in an XML Web service to expose the specified member as a public SOAP m ethod.
source	Specifies the control's source interfaces for connection points on a class. On a property or method, the **sourc e** attribute indicates that the member returns an object or VARIANT that is a source of events.
synchronize	Synchronizes access to the target method.
tag_name	Apply this attribute to a method in a request handler to expose it as a replacement method associated with a tag name.
vararg	Specifies that the function take a variable number of arguments.

See Also

Other Resources

Attributes by Usage

Parameter Attributes

The following attributes apply to parameters of a method in a class or interface.

Attribute	Description
custom	Lets you define your own attribute.
defaultvalue	Allows specification of a default value for a typed optional parameter.
first_is	Specifies the index of the first array element to be transmitted.
iid_is	Specifies the index of the first array element to be transmitted.
immediatebind	Indicates that the database will be notified immediately of all changes to a property of a data-bound object.
in	Indicates that a parameter is to be passed from the calling procedure to the called procedure.
last_is	Specifies the index of the last array element to be transmitted.
lcid	Lets you pass a locale identifier to a function.
length_is	Specifies the number of array elements to be transmitted.
max_is	Designates the maximum value for a valid array index.
optional	Specifies an optional parameter for a member function.
out	Identifies pointer parameters that are returned from the called procedure to the calling procedure (from the se rver to the client).
range	Specifies a range of allowable values for arguments or fields whose values are set at run time.
ref	Identifies a reference pointer.
retval	Designates the parameter that receives the return value of the member.
satype	Specifies the data type of the **SAFEARRAY** structure.
size_is	Specifies the size of memory allocated for sized pointers, sized pointers to sized pointers, and single- or multid imensional arrays.
unique	Specifies a unique pointer.

See Also

Other Resources

Attributes by Usage

Data Member Attributes

The following attributes apply to the data members in a class, coclass, or interface.

Attribute	Description
db_accessor	Groups **db_column** attributes that participate in **IAccessor**-based binding.
db_column	Binds a specified column to the rowset.
db_command	Creates an OLE DB command.
db_param	Associates the specified member variable with an input or output parameter and delimits the variable.
db_source	Creates a connection to a data source.
db_table	Opens an OLE DB table.
defaultbind	Indicates the single, bindable property that best represents the object.
displaybind	Indicates a property that should be displayed to the user as bindable.
id	Specifies a DISPID for a member function (either a property or a method, in an interface or dispinterface).
perf_counter	Adds Performance monitor counter support to a class member variable.
range	Specifies a range of allowable values for arguments or fields whose values are set at run time.
rdx	Creates a registry key or modifies an existing registry key.
readonly	Prohibits assignment to a data member.
requestedit	Indicates that the property supports the **OnRequestEdit** notification.
soap_header	Apply this attribute to a data member in an XML Web service to map the specified member to a SOAP header.

See Also

Other Resources

Attributes by Usage

Typedef, Enum, Union, and Struct Attributes

The following attributes apply to the typedef, struct, and enum C++ keywords.

typedef

Attribute	Description
case	Used with the switch_type attribute in a **union**.
custom	Lets you define your own attribute.
export	Causes a data structure to be placed in the .idl file.
first_is	Specifies the index of the first array element to be transmitted.
helpcontext	Specifies a context ID that lets the user view information about this element in the Help file.
helpfile	Sets the name of the Help file for a type library.
helpstring	Specifies a character string that is used to describe the element to which it applies.
library_block	Places a construct inside the .idl file's library block.
ptr	Designates a pointer as a full pointer.
public	Ensures that a typedef will go into the type library even if it is not referenced from within the .idl file.
ref	Identifies a reference pointer.
switch_is	Specifies the expression or identifier acting as the union discriminant that selects the union member.
switch_type	Identifies the type of the variable used as the union discriminant.
unique	Specifies a unique pointer.
wire_marshal	Specifies a data type that will be used for transmission instead of an application-specific data type.

enum

Attribute	Description
custom	Lets you define your own attribute.
export	Causes a data structure to be placed in the .idl file.
uuid	Specifies the unique ID for a class or interface.
v1_enum	Directs that the specified enumerated type be transmitted as a 32-bit entity, rather than the 16-bit default.

union

Attribute	Description
custom	Lets you define your own attribute.
export	Causes a data structure to be placed in the .idl file.

first_is	Specifies the index of the first array element to be transmitted.
last_is	Specifies the index of the last array element to be transmitted.
length_is	Specifies the number of array elements to be transmitted.
max_is	Designates the maximum value for a valid array index.
size_is	Specifies the size of memory allocated for sized pointers, sized pointers to sized pointers, and single- or multidimensional arrays.
unique	Specifies a unique pointer.
uuid	Specifies the unique ID for a class or interface.

Nonencapsulated union

Attribute	Description
ms_union	Controls the network data representation alignment of nonencapsulated unions.
no_injected_text	Prevents the compiler from injecting code as a result of attribute use.

struct

Attribute	Description
aggregatable	Indicates that the class supports aggregation.
aggregates	Indicates that a control aggregates the target class.
appobject	Identifies the coclass as an application object, which is associated with a full .exe application, and indicates that the functions and properties of the coclass are globally available in this type library.
coclass	Creates an ActiveX control.
com_interface_entry	Adds an interface entry to a COM map.
control	Specifies that the user-defined type is a control.
custom	Lets you define your own attribute.
db_column	Binds a specified column to the rowset.
db_command	Creates an OLE DB command.
db_param	Associates the specified member variable with an input or output parameter and delimits the variable.
db_source	Creates a connection to a data source.
db_table	Opens an OLE DB table.
default	Indicates that the custom or dispinterface defined within a coclass represents the default programmability interface.
defaultvtable	Defines an interface as the default vtable interface for a control.
event_receiver	Creates an event receiver.

event_source	Creates an event source.
export	Causes a data structure to be placed in the .idl file.
first_is	Specifies the index of the first array element to be transmitted.
hidden	Indicates that the item exists but should not be displayed in a user-oriented browser.
implements_category	Specifies implemented component categories for the class.
last_is	Specifies the index of the last array element to be transmitted.
length_is	Specifies the number of array elements to be transmitted.
max_is	Designates the maximum value for a valid array index.
perf_object	Adds Performance monitor object support to a class.
perfmon	Adds performance monitor support to a class.
requires_category	Specifies the required component categories of the target class.
size_is	Specifies the size of memory allocated for sized pointers, sized pointers to sized pointers, and single- or multidimensional arrays.
source	On a class, specifies the COM object's source interfaces for connection points. On a property or method, indicates that the member returns an object or VARIANT that is a source of events.
threading	Specifies the threading model for a COM object.
unique	Specifies a unique pointer.
uuid	Specifies the unique ID for a class or interface.
version	Identifies a particular version among multiple versions of a class.
vi_progid	Specifies a version-independent form of the ProgID.

See Also

Other Resources

Attributes by Usage

Array Attributes

The following attributes apply to arrays or **SAFEARRAY**s.

Attribute	Description
library_block	Places a construct inside the .idl file's library block.
satype	Specifies the data type of the SAFEARRAY structure.
string	Indicates that the one-dimensional **char**, **wchar_t**, **byte** (or equivalent) array or the pointer to such an array must be treated as a string.

See Also

Other Resources

Attributes by Usage

Stand-Alone Attributes

A stand-alone attribute does not operate on a C++ keyword but is more like a line of code. Stand-alone attribute statements require a semicolon at the end of the line.

Attribute	Description
cpp_quote	Emits the specified string, without the quote characters, into the generated header file.
custom	Lets you define your own attribute.
db_command	Creates an OLE DB command.
emitidl	Determines whether all subsequent IDL attributes will be processed and placed in the generated .idl file.
idl_module	Specifies an entry point in a DLL.
idl_quote	Allows you to use IDL constructs that are not supported in the current version of Visual C++ and have them pass through to the generated .idl file.
import	Specifies another .idl, .odl, or .h file containing definitions you want to reference from your main .idl file.
importidl	Inserts the specified .idl file into the generated .idl file
importlib	Makes types that have already been compiled into another type library available to the type library being created.
include	Specifies one or more header files to be included in the generated .idl file.
includelib	Causes an .idl or .h file to be included in the generated .idl file.
library_block	Places a construct inside the .idl file's library block.
module	Defines the library block in the .idl file.
no_injected_text	Prevents the compiler from injecting code as a result of attribute use.
pragma	Emits the specified string, without the quote characters, into the generated .idl file.

See Also

Other Resources

Attributes by Usage

Custom Attributes (C++)

A custom attribute is a strongly typed technique that allows the user to extend metadata. This topic deals with using attributes: specifying parameters and targets.

For information on defining new custom attributes, see,

- User-Defined Attributes
- attribute (deprecated syntax).

See Also

Reference

attribute

Custom Attributes (C++)

Attribute Contexts

C++ attributes can be described using four basic fields: the target they can be applied to (**Applies To**), if they are repeatable or not (**Repeatable**), the required presence of other attributes (**Required Attributes**), and incompatibilities with other attributes (**Invalid Attributes**). These fields are listed in an accompanying table in each attribute's reference topic. Each of these fields is described below.

Applies To

This field describes the different C++ language elements that are legal targets for the specified attribute. For instance, if an attribute specifies "class" in the **Applies To** field, this indicates that the attribute can only be applied to a legal C++ class. If the attribute is applied to a member function of a class, a syntax error would result.

For more information, see Attributes by Usage.

Repeatable

This field states whether the attribute can be repeatedly applied to the same target. The majority of attributes are not repeatable.

Required Attributes

This field lists other attributes that need to be present (that is, applied to the same target) for the specified attribute to function properly. It is uncommon for an attribute to have any entries for this field.

Invalid Attributes

This field lists other attributes that are incompatible with the specified attribute. It is uncommon for an attribute to have any entries for this field.

See Also

Other Resources

C++ Attributes Reference

Attribute Requirements

The requirements listed for C++ attributes describe the project types, compiler settings, and other information necessary for an attribute to work. The categories of information are described below.

Note
Using attributes on a class that derives from a class that also uses attributes is not supported.

Header

This field lists the header files that must be included before an attribute can be used.

Project

This field describes the project types in which an attribute can be used.

Compiler

This field provides the compiler options that must be present for this attribute to be used.

See Also

Concepts

Attribute Contexts

Other Resources

Attributes by Group

Attributes Alphabetical Reference

The following attributes are available in Visual C++.

Attribute	Description
aggregatable	Indicates that a control can be aggregated by another control.
aggregates	Indicates that a control aggregates the target class.
appobject	Identifies the coclass as an application object, which is associated with a full EXE application, and indicates that the functions and properties of the coclass are globally available in this type library.
async_uuid	Specifies the UUID that directs the MIDL compiler to define both synchronous and asynchronous versions of a COM interface.
attribute	Allows you to create a custom attribute.
bindable	Indicates that the property supports data binding.
call_as	Enables a nonremotable function to be mapped to a remote function.
case	Used with the switch_type attribute in a union.
coclass	Creates a COM object, which can implement a COM interface.
com_interface_entry	Adds an interface entry to a COM map.
control	Specifies that the user-defined type is a control.
cpp_quote	Emits the specified string, without the quote characters, into the generated header file.
custom	Lets you define your own attributes.
db_accessor	Binds columns in a rowset and binds them to the corresponding accessor maps.
db_column	Binds a specified column to the rowset.
db_command	Executes an OLE DB command.
db_param	Associates the specified member variable with an input or output parameter.
db_source	Creates and encapsulates a connection, through a provider, to a data source.
db_table	Opens an OLE DB table.
default	Indicates that the custom or dispinterface defined within a coclass represents the default programmability interface.
defaultbind	Indicates the single, bindable property that best represents the object.
defaultcollelem	Used for Visual Basic code optimization.

defaultvalue	Allows specification of a default value for a typed optional parameter.
defaultvtable	Defines an interface as the default vtable interface for a control.
dispinterface	Places an interface in the .idl file as a dispatch interface.
displaybind	Indicates a property that should be displayed to the user as bindable.
dual	Places an interface in the .idl file as a dual interface.
emitidl	Determines whether all subsequent IDL attributes will be processed and placed in the generated .idl file.
entry	Specifies an exported function or constant in a module by identifying the entry point in the DLL.
event_receiver	Creates an event receiver.
event_source	Creates an event source.
export	Causes a data structure to be placed in the .idl file.
first_is	Specifies the index of the first array element to be transmitted.
helpcontext	Specifies a context ID that lets the user view information about this element in the Help file.
helpfile	Sets the name of the Help file for a type library.
helpstring	Specifies the ID of a help topic in an .hlp or .chm file.
helpstringdll	Specifies the name of the DLL to use to perform document string lookup (localization).
hidden	Indicates that the item exists but should not be displayed in a user-oriented browser.
id	Specifies a DISPID for a member function (either a property or a method, in an interface or dispinterface).
idl_module	Specifies an entry point in a DLL.
idl_quote	Allows you to use attributes or IDL constructs that are not supported in the current version of Visual C++.
iid_is	Specifies the IID of the COM interface pointed to by an interface pointer.
immediatebind	Indicates that the database will be notified immediately of all changes to a property of a data-bound object.
implements	Specifies dispatch interfaces that are forced to be members of the IDL coclass.
implements_category	Specifies implemented component categories for the class.
import	Specifies another .idl, .odl, or header file containing definitions you want to reference from your main .idl file.
importidl	Inserts the specified .idl file into the generated .idl file.

importlib	Makes types that have already been compiled into another type library available to the type library being created.
in	Indicates that a parameter is to be passed from the calling procedure to the called procedure.
include	Specifies one or more header files to be included in the generated .idl file.
includelib	Causes an .idl or .h file to be included in the generated .idl file.
last_is	Specifies the index of the last array element to be transmitted.
lcid	Lets you pass a locale identifier to a function.
length_is	Specifies the number of array elements to be transmitted.
library_block	Places a construct inside the .idl file's library block.
licensed	Indicates that the coclass to which it applies is licensed, and must be instantiated using **IClassFactory2**.
local	Allows you to use the MIDL compiler as a header generator when used in the interface header. When used in an individual function, designates a local procedure for which no stubs are generated.
max_is	Designates the maximum value for a valid array index.
module	Defines the library block in the .idl file.
ms_union	Controls the network data representation alignment of nonencapsulated unions.
no_injected_text	Prevents the compiler from injecting code as a result of attribute use.
nonbrowsable	Indicates that an interface member should not be displayed in a property browser.
noncreatable	Defines an object that cannot be instantiated by itself.
nonextensible	Specifies that the **IDispatch** implementation includes only the properties and methods listed in the interface description and cannot be extended with additional members at run time.
object	Identifies a custom interface; synonymous with custom attribute.
odl	Identifies an interface as an Object Description Language (ODL) interface.
oleautomation	Indicates that an interface is compatible with Automation.
optional	Specifies an optional parameter for a member function.
out	Identifies pointer parameters that are returned from the called procedure to the calling procedure (from the server to the client).
perf_counter	Adds performance monitor counter support to a class member variable.
perf_object	Apply this attribute to a class to define a performance monitor object.
perfmon	Adds performance monitor support to a class.

pointer_default	Specifies the default pointer attribute for all pointers except top-level pointers that appear in parameter lists.
pragma	Emits the specified string, without the quote characters, into the generated .idl file.
progid	Specifies the ProgID for a COM object.
propget	Specifies a property accessor (get) function.
propput	Specifies a property setting function.
propputref	Specifies a property setting function that uses a reference instead of a value.
ptr	Designates a pointer as a full pointer.
public	Ensures that a typedef will go into the type library even if it is not referenced from within the .idl file.
range	Specifies a range of allowable values for arguments or fields whose values are set at run time.
rdx	Creates or modifies a registry key.
readonly	Prohibits assignment to a variable.
ref	Identifies a reference pointer.
registration_script	Executes the specified registration script.
request_handler	Apply this attribute to a class to expose it as an ATL Server request handler and enable it to handle HTTP requests.
requestedit	Indicates that the property supports the **OnRequestEdit** notification.
requires_category	Specifies required component categories for the class.
restricted	Specifies that a library, or member of a module, interface, or dispinterface cannot be called arbitrarily.
retval	Designates the parameter that receives the return value of the member.
satype	Specifies the data type of the **SAFEARRAY**.
size_is	Specifies the size of memory allocated for sized pointers, sized pointers to sized pointers, and single- or multidimensional arrays.
soap_handler	Provides the methods necessary for handling SOAP method calls and exposing information about the services offered by this class through SDL.
soap_header	Apply this attribute to a SOAP method in an XML Web service to specify the data member used to hold the value of a SOAP header.
soap_method	Apply this attribute to a method in an XML Web service to expose the specified member as a SOAP method with a corresponding WSDL description.
source	Indicates that a member of a class, property, or method is a source of events.

string	Indicates that the one-dimensional **char**, **wchar_t**, **byte**, or equivalent array or the pointer to such an array must be treated as a string.
support_error_info	Supports error reporting for the target object.
switch_is	Specifies the expression or identifier acting as the union discriminant that selects the union member.
switch_type	Identifies the type of the variable used as the union discriminant.
synchronize	Synchronizes access to a method.
tag_name	In a request handler, exposes it as a replacement method associated with a tag name.
threading	Specifies the threading model for a COM object.
transmit_as	Instructs the compiler to associate a presented type, which client and server applications manipulate, with a transmitted type.
uidefault	Indicates that the type information member is the default member for display in the user interface.
unique	Specifies a unique pointer.
usesgetlasterror	Tells the caller that if there is an error when calling that function, the caller can then call **GetLastError** to retrieve the error code.
uuid	Specifies the unique ID for a class or interface.
v1_enum	Directs that the specified enumerated type be transmitted as a 32-bit entity, rather than the 16-bit default.
vararg	Specifies that the function take a variable number of arguments.
version	Identifies a particular version among multiple versions of an interface or class.
vi_progid	Specifies a version-independent form of the ProgID.
wire_marshal	Specifies a data type that will be used for transmission instead of an application-specific data type.

See Also

Other Resources

C++ Attributes Reference
Attributed Programming Concepts
Attributes by Group
Attributes by Usage

aggregatable

Indicates that the class supports aggregation.

```
[ aggregatable(
      value
) ]
```

Parameters

value (optional)

A parameter to indicate when the COM object can be aggregated:

- **never** The COM object cannot be aggregated.
- **allowed** The COM object can be created directly or it can be aggregated. This is the default.
- **always** The COM object cannot be created directly and can only be aggregated. When you call **CoCreateInstance** for this object, you must specify the aggregating object's **IUnknown** interface (the controlling **IUnknown**).

Remarks

The **aggregatable** C++ attribute has the same functionality as the aggregatable MIDL attribute. This means that the compiler will pass the **aggregatable** attribute through to the generated .idl file.

This attribute requires that the coclass, progid, or vi_progid attribute (or another attribute that implies one of these) also be applied to the same element. If any single attribute is used, the other two are automatically applied. For example, if **progid** is applied, **vi_progid** and **coclass** are also applied.

ATL Projects

If this attribute is used within a project that uses ATL, the behavior of the attribute changes. In addition to the previously described behavior, the attribute also adds one of the following macros to the target class:

Parameter value	Inserted macro
Never	DECLARE_NOT_AGGREGATABLE
Allowed	DECLARE_POLY_AGGREGATABLE
Always	DECLARE_ONLY_AGGREGATABLE

Example

```
// cpp_attr_ref_aggregatable.cpp
// compile with: /LD
#define _ATL_ATTRIBUTES
#include "atlbase.h"
#include "atlcom.h"

[module(name="MyModule")];

[ coclass, aggregatable(allowed),
  uuid("1a8369cc-1c91-42c4-befa-5a5d8c9d2529")]
class CMyClass {};
```

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	No

Required attributes	One or more of the following: **coclass**, **progid**, or **vi_progid**.
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

Aggregation

Other Resources

IDL Attributes

Class Attributes

Typedef, Enum, Union, and Struct Attributes

ATL Samples

aggregates

Indicates that the object aggregates the object specified by the CLSID.

```
[ aggregates(
      clsid,
      variable_name
) ]
```

Parameters

clsid

Specifies the CLSID of the aggregatable object.

variable_name

The name of the variable that is to be inserted. This variable contains the **IUnknown** of the object being aggregated.

Remarks

When applied to an object, the **aggregates** C++ attribute implements an outer wrapper for the object being aggregated (specified by *clsid*).

This attribute requires that the coclass, progid, or vi_progid attribute (or another attribute that implies one of these) also be applied to the same element. If any single attribute is used, the other two are automatically applied. For example, if **progid** is applied, **vi_progid** and **coclass** are also applied.

ATL Projects

If this attribute is used within a project that uses ATL, the behavior of the attribute changes. First, the following entry is added to the COM map of the target object:

```
COM_INTERFACE_ENTRY_AUTOAGGREGATE_BLIND(_m_spAttrXXX, clsid)
```

Second, the DECLARE_GET_CONTROLLING_UNKNOWN macro is also added.

Example

```
// cpp_attr_ref_aggregates.cpp
// compile with: /LD
#define _ATL_ATTRIBUTES
#include "atlbase.h"
#include "atlcom.h"

// requires 'aggregatable.dll'
// see aggregatable attribute to create 'aggregatable.dll'
class DECLSPEC_UUID("1a8369cc-1c91-42c4-befa-5a5d8c9d2529") CMyClass;

[module (name="MYObject")];
[object, uuid("ab006d85-e754-47c5-9ef4-2744ff32a20c")]
__interface IObject
{
};

[ coclass, aggregates(__uuidof(CMyClass)),
  uuid("91cb2c06-8931-432a-baac-206e55c4edfb")]
struct CObject : IObject
{
   int i;
};
```

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	Yes
Required attributes	One or more of the following: **coclass**, **progid**, or **vi_progid**.
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

Aggregation
Aggregatable
COM_INTERFACE_ENTRY_AUTOAGGREGATE_BLIND

Other Resources

COM Attributes
Class Attributes
Typedef, Enum, Union, and Struct Attributes
ATL Samples

appobject

Identifies the coclass as an application object, which is associated with a full .exe application, and indicates that the functions and properties of the coclass are globally available in this type library.

```
[appobject]
```

Remarks

The **appobject** C++ attribute has the same functionality as the appobject MIDL attribute.

Example

The following code shows a simple class definition preceded by an attribute block that includes **appobject**:

```
// cpp_attr_ref_appobject.cpp
// compile with: /LD
#include <windows.h>
[module(name="MyLib", uuid="f1ce17f0-a5df-4d26-95f6-0a122197ac5b")];

[object, uuid="905de6db-7a12-45ab-9f8b-b39f5112f010"]
__interface ICustom {};

[coclass, appobject,uuid="00395340-745f-4b69-bd58-e2921452b9fc"]
class A : public ICustom {
   int i;
};
```

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	No
Required attributes	**coclass**
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes
Class Attributes
Typedef, Enum, Union, and Struct Attributes
ATL Samples

async_uuid

Specifies the UUID that directs the MIDL compiler to define both synchronous and asynchronous versions of a COM interface.

```
[async_uuid (
      uuid
)]
```

Parameters

uuid

A UUID that identifies the version of the interface.

Remarks

The **async_uuid** C++ attribute has the same functionality as the async_uuid MIDL attribute.

Example

```
// cpp_attr_ref_async_uuid.cpp
// compile with: /LD
#include <Windows.h>
[module(name="Test")];
[object, uuid("9e66a290-4365-11d2-a997-00c04fa37ddb"),
async_uuid("e8583106-38fd-487e-912e-4fc8645c677e")]
__interface ICustom {
   HRESULT Custom([in] long l, [out, retval] long *pLong);
};
```

Requirements

Attribute Context

Applies to	**interface**
Repeatable	No
Required attributes	None
Invalid attributes	**dual**, **dispinterface**

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Interface Attributes

ATL Samples

attribute

Allows you to create a custom attribute.

```
[ attribute(
      AllowOn,
   AllowMultiple=boolean,
   Inherited=boolean
) ]
```

Parameters

AllowOn

Specifies the language elements to which the custom attribute can be applied. Default is **System::AttributeTargets::All** (see System::AttributeTargets).

AllowMultiple

Specifies whether the custom attribute can be applied repeatedly to a construct. Default is **FALSE**.

Inherited

Indicates if the attribute is to be inherited by subclasses. The compiler provides no special support for this functionality; it is the job of the attribute consumers (Reflection, for example) to respect this information. If *Inherited* is **TRUE**, the attribute is inherited. If *AllowMultiple* is **TRUE**, the attribute will accumulate on the derived member; if *AllowMultiple* is **FALSE**, the attribute will override (or replace) in inheritance. If *Inherited* is **FALSE**, the attribute will not be inherited. Default is **TRUE**.

Remarks

Note

The **attribute** attribute is now deprecated. Use the common language runtime attribute System.Attribute to directly to create user-defined attirbutes. For more information, see User-Defined Attributes.

You define a custom attribute by placing the **attribute** attribute on a managed class or struct definition. The name of the class is the custom attribute. For example:

```
[ attribute(Parameter) ]
public ref class MyAttr {};
```

defines an attribute called MyAttr that can be applied to function parameters. The class must be public if the attribute is going to be used in other assemblies.

Note

To prevent namespace collisions, all attribute names implicitly end with "Attribute"; in this example, the name of the attribute and class is actually MyAttrAttribute, but MyAttr and MyAttrAttribute can be used interchangeably.

The class's public constructors define the attribute's unnamed parameters. Overloaded constructors allow multiple ways of specifying the attribute, so a custom attribute that is defined the following way:

```
// cpp_attr_ref_attribute.cpp
// compile with: /c /clr
using namespace System;
[ attribute(AttributeTargets::Class) ]   // apply attribute to classes
public ref class MyAttr {
public:
   MyAttr() {}   // Constructor with no parameters
   MyAttr(int arg1) {}   // Constructor with one parameter
};

[MyAttr]
```

```
ref class ClassA {};   // Attribute with no parameters

[MyAttr(123)]
ref class ClassB {};   // Attribute with one parameter
```

The class's public data members and properties are the attribute's optional named parameters:

```
// cpp_attr_ref_attribute_2.cpp
// compile with: /c /clr
using namespace System;
[ attribute(AttributeTargets::Class) ]
ref class MyAttr {
public:
   // Property Priority becomes attribute's named parameter Priority
    property int Priority {
       void set(int value) {}
       int get() { return 0;}
   }
   // Data member Version becomes attribute's named parameter Version
   int Version;
   MyAttr() {}   // constructor with no parameters
   MyAttr(int arg1) {}   // constructor with one parameter
};

[MyAttr(123, Version=2)]
ref class ClassC {};
```

For a list of possible attribute parameter types, see Custom Attributes.

See User-Defined Attributes for a discussion on attribute targets.

The **attribute** attribute has an **AllowMultiple** parameter that specifies whether the custom attribute is single use or multiuse (can appear more than once on the same entity).

```
// cpp_attr_ref_attribute_3.cpp
// compile with: /c /clr
using namespace System;
[ attribute(AttributeTargets::Class, AllowMultiple = true) ]
ref struct MyAttr {
   MyAttr(){}
};   // MyAttr is a multiuse attribute

[MyAttr, MyAttr()]
ref class ClassA {};
```

Custom attribute classes are derived directly or indirectly from AttributeCollection, which makes identifying attribute definitions in metadata fast and easy. The **attribute** attribute implies inheritance from System::Attribute, so explicit derivation is not necessary:

```
[ attribute(Class) ]
ref class MyAttr
```

is equivalent to

```
[ attribute(Class) ]
ref class MyAttr : System::Attribute   // OK, but redundant.
```

attribute is an alias for System.AttributeUsageAttribute (not AttributeAttribute; this is an exception to the attribute naming rule).

Requirements
Attribute Context

Applies to	**ref class**, **ref struct**
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

Example

```
// cpp_attr_ref_attribute_4.cpp
// compile with: /c /clr
using namespace System;
[attribute(AttributeTargets::Class)]
ref struct ABC {
   ABC(Type ^) {}
};

[ABC(String::typeid)]   // typeid operator yields System::Type ^
ref class MyClass {};
```

The **Inherited** named argument specifies whether a custom attribute applied on a base class will show up on reflection of a derived class.

```
// cpp_attr_ref_attribute_5.cpp
// compile with: /clr
using namespace System;
using namespace System::Reflection;

[attribute( AttributeTargets::Method, Inherited=false )]
ref class BaseOnlyAttribute { };

[attribute( AttributeTargets::Method, Inherited=true )]
ref class DerivedTooAttribute { };

ref struct IBase {
public:
   [BaseOnly, DerivedToo]
   virtual void meth() {}
};

// Reflection on Derived::meth will show DerivedTooAttribute
// but not BaseOnlyAttribute.
ref class Derived : public IBase {
public:
   virtual void meth() override {}
};

int main() {
   IBase ^ pIB = gcnew Derived;

   MemberInfo ^ pMI = pIB->GetType( )->GetMethod( "meth" );
   array<Object ^> ^ pObjs = pMI->GetCustomAttributes( true );
   Console::WriteLine( pObjs->Length ) ;
}
```

Output

```
2
```

See Also

Other Resources

bindable

Indicates that the property supports data binding.

```
[bindable]
```

Remarks

The **bindable** C++ attribute has the same functionality as the bindable MIDL attribute. You can use it on properties defined with the propget, propput, or propputref attributes, or you can manually define a bindable method.

The following MFC samples show the use of **bindable**:

- Controls Samples: MFC-Based ActiveX Controls
- CIRC Sample: ActiveX Control
- TESTHELP Sample: ActiveX Control with Tooltips and Help

Example

The following code shows how you can use **bindable** on a property:

```
// cpp_attr_ref_bindable.cpp
// compile with: /LD
#include <windows.h>
[
   uuid("479B29E3-9A2C-11D0-B696-00A0C903487A"),
   dispinterface,
   helpstring("property demo Interface")
]
__interface IPropDemo : IDispatch {

   [propget, id(1), bindable, displaybind, defaultbind, requestedit] HRESULT P1([out, retva
l] long *nSize);
   [propput, id(1), bindable, displaybind, defaultbind, requestedit] HRESULT P1([in] long n
Size);
   [id(3), bindable, propget] HRESULT Object([out, retval] IDispatch **ppObj);
   [id(3), bindable, propputref] HRESULT Object([in] IDispatch* pObj);
   [id(-552), helpstring("method AboutBox")] HRESULT AboutBox();
};

[ module(name="PropDemoLib", uuid="479B29E2-9A2C-11D0-B696-00A0C903487A", version="1.0", he
lpstring="property demo") ];
```

Requirements

Attribute Context

Applies to	Interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

defaultbind

displaybind

immediatebind
requestedit

Other Resources

IDL Attributes
Method Attributes
ATL Samples

call_as

Enables a local function to be mapped to a remote function so that when the remote function is called, the local function is invoked.

```
[ call_as(
      function
) ]
```

Parameters

function

The local function that you want to be called when a remote function is invoked.

Remarks

The **call_as** C++ attribute has the same functionality as the call_as MIDL attribute.

Example

The following code shows how you can use **call_as** to map a nonremotable function (**f1**) to a remotable function (**Remf1**):

```
// cpp_attr_ref_call_as.cpp
// compile with: /LD
#include "unknwn.h"
[module(name="MyLib")];
[dual, uuid("00000000-0000-0000-0000-000000000001")]
__interface IMInterface {
   [local] HRESULT f1 ( int i );
   [call_as(f1)] HRESULT Remf1 ( int i );
};
```

Requirements

Attribute Context

Applies to	Interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

local (C++)

Other Resources

IDL Attributes

Method Attributes

ATL Samples

case (C++)

Used with the switch_type attribute in a **union**.

```
[ case(
      value
) ]
```

Parameters

value

A possible input value for which you want to provide processing. The type of **value** can be one of the following types:

- **int**
- **char**
- **boolean**
- **enum**

or an identifier of such a type.

Remarks

The **case** C++ attribute has the same functionality as the **case** MIDL attribute. This attribute is only used with the switch_type attribute.

Example

The following code shows a use of the **case** attribute:

```
// cpp_attr_ref_case.cpp
// compile with: /LD
#include <unknwn.h>
[export]
struct SizedValue2 {
   [switch_type(char), switch_is(kind)] union {
      [case(1), string]
          wchar_t* wval;
      [default, string]
          char* val;
   };
    char kind;
};
[module(name="ATLFIRELib")];
```

Requirements

Attribute Context

Applies to	Member of a **class** or **struct**
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Typedef, Enum, Union, and Struct Attributes
Class Attributes
ATL Samples

coclass

Creates a COM object, which can implement a COM interface.

```
[coclass]
```

Remarks

The **coclass** C++ attribute places a coclass construct in the generated .idl file.

When defining a coclass, you can also specify the uuid, version, threading, vi_progid, and progid attributes. If any one of them is not specified, it will be generated.

If two header files contain classes with the **coclass** attribute and don't specify a GUID, the compiler will use the same GUID for both classes, and that will result in a MIDL error. Therefore, you should use the **uuid** attribute when you use **coclass**.

ATL Projects

When this attribute precedes a class or structure definition in an ATL project, it:

- Injects code or data to support auto registration for the object.
- Injects code or data to support a COM class factory for the object.
- Injects code or data to implement **IUnknown** and make the object a COM-creatable object.

Specifically, the following base classes are added to the target object:

- CComCoClass Class provides the default class factory and aggregation model for the object.
- CComObjectRootEx Class has a template based on the threading model class specified by the threading attribute. If the **threading** attribute is not specified, the default threading model is apartment.
- IProvideClassInfo2Impl is added if the noncreatable attribute is not specified for the target object.

Finally, any dual interface that is not defined using embedded IDL is replaced with the corresponding IDispatchImpl class. If the dual interface is defined in embedded IDL, the particular interface in the base list is not modified.

The **coclass** attribute also makes the following functions available via injected code, or in the case of **GetObjectCLSID**, as a static method in the base class **CComCoClass**:

- **UpdateRegistry** registers the class factories of the target class.
- **GetObjectCLSID**, which is related to registration, can also be used to obtain the CLSID of the target class.
- **GetObjectFriendlyName** by default returns a string of the format "<*target class name*> **Object**". If this function is already present, it is not added. Add this function to the target class to return a friendlier name than the one automatically generated.
- **GetProgID**, which is related to registration, returns the string specified with the progid attribute.
- **GetVersionIndependentProgID** has the same functionality as **GetProgID**, but it returns the string specified with vi_progid.

The following changes, which are related to the COM map, are made to the target class:

- A COM map is added with entries for all interfaces the target class derives from and all entries specified by the COM Interface Entry Points attribute or those required by the aggregates attribute.
- An OBJECT_ENTRY_AUTO macro is inserted into the COM map. This macro is similar to OBJECT_ENTRY in terms of functionality but does not need to be part of the COM map of the target class.

The name of the coclass generated in the .idl file for the class will have the same name as the class. For example, and referring to the following sample, to access the class ID for a coclass CMyClass, in a client through the MIDL-generated header file, use CLSID_CMyClass.

Example

The following code shows how to use the **coclass** attribute:

```
// cpp_attr_ref_coclass1.cpp
// compile with: /LD
#include "unknwn.h"
[module(name="MyLib")];

[ object, uuid("00000000-0000-0000-0000-000000000001") ]
__interface I {
   HRESULT func();
};

[coclass, progid("MyCoClass.coclass.1"), vi_progid("MyCoClass.coclass"),
appobject, uuid("9E66A294-4365-11D2-A997-00C04FA37DDB")]
class CMyClass : public I {};
```

The following sample shows how to override the default implementation of a function that appears in the code injected by the **coclass** attribute. See /Fx for more information on viewing injected code. Any base classes or interfaces that you use for a class will be appear in the injected code. Further, if a class is included by default in the injected code and you explicitly specify that class as a base for your coclass, the attribute provider will use the form specified in your code.

```
// cpp_attr_ref_coclass2.cpp
// compile with: /LD
#include <atlbase.h>
#include <atlcom.h>
#include <atlwin.h>
#include <atltypes.h>
#include <atlctl.h>
#include <atlhost.h>
#include <atlplus.h>

[module(name="MyLib")];

[object, uuid("00000000-0000-0000-0000-000000000000")]
__interface bb {};

[coclass, uuid("00000000-0000-0000-0000-000000000001")]
class CMyClass : public bb {
public:
   // by adding the definition of UpdateRegistry to your code,
   // the function will not be included in the injected code
   static HRESULT WINAPI UpdateRegistry(BOOL bRegister) {
      // you can add to the default implementation
      CRegistryVirtualMachine rvm;
      HRESULT hr;
      if (FAILED(hr = rvm.AddStandardReplacements()))
         return hr;
      rvm.AddReplacement(_T("FriendlyName"), GetObjectFriendlyName());
      return rvm.VMUpdateRegistry(GetOpCodes(), GetOpcodeStringVals(),
         GetOpcodeDWORDVals(), GetOpcodeBinaryVals(), bRegister);
   }
};
```

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	No
Required attributes	None

Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

appobject

Other Resources

IDL Attributes

COM Attributes

Class Attributes

Typedef, Enum, Union, and Struct Attributes

com_interface_entry (C++)

Adds an interface entry into the COM map of the target class.

```
[ com_interface_entry(
      com_interface_entry
  ) ]
```

Parameters

com_interface_entry

A string containing the actual text of the entry. For a list of possible values, see COM_INTERFACE_ENTRY Macros.

Remarks

The **com_interface_entry** C++ attribute inserts the unabridged contents of a character string into the COM interface map of the target object. If the attribute is applied once to the target object, the entry is inserted into the beginning of the existing interface map. If the attribute is applied repeatedly to the same target object, the entries are inserted at the beginning of the interface map in the order they are received.

This attribute requires that the coclass, progid, or vi_progid attribute (or another attribute that implies one of these) also be applied to the same element. If any single attribute is used, the other two are automatically applied. For example, if **progid** is applied, **vi_progid** and **coclass** are also applied.

Because the first usage of **com_interface_entry** causes the new interface to be inserted at the beginning of the interface map, it must be one of the following COM_INTERFACE_ENTRY types:

- COM_INTERFACE_ENTRY
- COM_INTERFACE_ENTRY_IID
- COM_INTERFACE_ENTRY2
- COM_INTERFACE_ENTRY2_IID

Additional usages of the **com_interface_entry** attribute can use all supported COM_INTERFACE_ENTRY types.

This restriction is necessary because ATL uses the first entry in the interface map as the identity **IUnknown**; therefore, the entry must be a valid interface. For example, the following code sample is invalid because the first entry in the interface map does not specify an actual COM interface.

```
[ coclass, com_interface_entry =
    "COM_INTERFACE_ENTRY_NOINTERFACE(IDebugTest)"
]
   class CMyClass
   {
   };
```

Example

The following code adds two entries to the existing COM interface map of **CMyBaseClass**. The first is a standard interface, and the second hides the **IDebugTest** interface.

```
// cpp_attr_ref_com_interface_entry.cpp
// compile with: /LD
#define _ATL_ATTRIBUTES
#include "atlbase.h"
#include "atlcom.h"

[module (name ="ldld")];

[ object,
  uuid("7dbebed3-d636-4917-af62-c767a720a5b9")]
__interface IDebugTest{};
```

```
[ object,
  uuid("2875ceac-f94b-4087-8e13-d13dc167fcfc")]
__interface IMyClass{};

[ coclass,
  com_interface_entry ("COM_INTERFACE_ENTRY (IMyClass)"),
  com_interface_entry ("COM_INTERFACE_ENTRY_NOINTERFACE(IDebugTest)"),
  uuid("b85f8626-e76e-4775-b6a0-4826a9e94af2")
]

class CMyClass: public IMyClass, public IDebugTest
{
};
```

The resulting COM object map for **CMyBaseClass** is as follows:

```
    BEGIN_COM_MAP(CMyClass)
        COM_INTERFACE_ENTRY (IMyClass)
        COM_INTERFACE_ENTRY_NOINTERFACE(IDebugTest)
        COM_INTERFACE_ENTRY(IMyClass)
        COM_INTERFACE_ENTRY2(IDispatch, IMyClass)
        COM_INTERFACE_ENTRY(IDebugTest)
        COM_INTERFACE_ENTRY(IProvideClassInfo)
    END_COM_MAP()
```

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	Yes
Required attributes	One or more of the following: **coclass**, **progid**, or **vi_progid**.
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

COM Attributes
Class Attributes
Typedef, Enum, Union, and Struct Attributes
ATL Samples

control

Specifies that the user-defined type is a control.

```
[control]
```

Remarks

The **control** attribute implies the coclass attribute. The **control** C++ attribute has the same functionality as the control MIDL attribute.

Example

```
// cpp_attr_ref_control.cpp
// compile with: /LD
#include <windows.h>
[module(name="Test", control=true)];

[object, uuid("9e66a290-4365-11d2-a997-00c04fa37ddb")]
__interface ICustom {
   HRESULT Custom([in] long l, [out, retval] long *pLong);
};

[coclass, control, appobject, uuid("9e66a294-4365-11d2-a997-00c04fa37ddb")]
class CTest : public ICustom {};
```

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes
Class Attributes
Typedef, Enum, Union, and Struct Attributes
ATL Samples

cpp_quote

Emits the specified string, without the quote characters, into the generated .idl file.

```
[ cpp_quote(
   "statement"
) ];
```

Parameters

statement

A C instruction.

Remarks

The **cpp_quote** C++ attribute is useful if you want to put a preprocessor directive in an .idl file.

You can also use **cpp_quote** and generate an .h file as part of the MIDL compilation. For example, if you have a C++ header file that uses C++ IDL attributes but cannot use this file for some task, then you can compile it to create a MIDL-generated .h file, which you should be able to use.

The **cpp_quote** attribute has the same functionality as the cpp_quote MIDL attribute.

Example

See the example for dual for an example use how to use **cpp_quote**.

Requirements

Attribute Context

Applies to	Anywhere
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Stand-Alone Attributes

ATL Samples

custom (C++)

Defines metadata for an object in the type library.

```
[ custom(
      uuid,
   value
) ];
```

Parameters

uuid

A unique ID.

value

A value that can be put into a variant.

Remarks

The **custom** C++ attribute will cause information to be placed into the type library. You will need a tool that reads the custom value from type library.

The **custom** attribute has the same functionality as the custom MIDL attribute.

Requirements

Attribute Context

Applies to	Non-COM **interface**, **class**, **enum**s, **idl_module** methods, interface members, interface parameters, **typedef**s, **union**s, **struct**s
Repeatable	Yes
Required attributes	**coclass** (when used on class)
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes
Stand-Alone Attributes
Typedef, Enum, Union, and Struct Attributes
Parameter Attributes
Method Attributes
Class Attributes
Interface Attributes
ATL Samples

db_accessor

Groups **db_column** attributes that participate in **IAccessor**-based binding.

```
[ db_accessor(
      num,
   auto
) ]
```

Parameters

num

Specifies the accessor number (a zero-based integer index). You must specify accessor numbers in increasing order, using integers or defined values.

auto

A Boolean value that specifies whether the accessor is automatically retrieved (**TRUE**) or not retrieved (**FALSE**).

Remarks

db_accessor defines the underlying OLE DB accessor for subsequent **db_column** and **db_param** attributes within the same class or function. **db_accessor** is usable at member level and is used to group **db_column** attributes that participate in OLE DB **IAccessor**-based binding. It is used in conjunction with either the **db_table** or **db_command** attributes. Calling this attribute is similar to calling the BEGIN_ACCESSOR and END_ACCESSOR macros.

db_accessor generates a rowset and binds it to the corresponding accessor maps. If you do not call **db_accessor**, accessor 0 will automatically be generated, and all column bindings will be mapped to this accessor block.

db_accessor groups database column bindings into one or more accessors. For a discussion of the scenarios in which you need to use multiple accessors, see Using Multiple Accessors on a Rowset. Also see "User Record Support for Multiple Accessors" in User Records.

When the consumer attribute provider applies this attribute to a class, the compiler will rename the class to _*YourClassName*Accessor, where *YourClassName* is the name you gave the class, and the compiler will also create a class called *YourClassName,* which derives from _*YourClassName*Accessor. In Class View, you will see both classes.

Example

The following example uses **db_accessor** to group columns in the Orders table from the Northwind database into two accessors. Accessor 0 is an automatic accessor, and accessor 1 is not.

```
// cpp_attr_ref_db_accessor.cpp
// compile with: /LD /link /OPT:NOREF
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atldbcli.h>

[ db_command(L"SELECT LastName, FirstName FROM Orders") ]
class CEmployees {
public:
   [ db_accessor(0, TRUE) ];
   [ db_column("1") ] LONG m_OrderID;
   [ db_column("2") ] TCHAR m_CustomerID[6];
   [ db_column("4") ] DBTIMESTAMP m_OrderDate;

   [ db_accessor(1, FALSE) ];
   [ db_column("8") ] CURRENCY m_Freight;
};
```

Requirements

Attribute Context

Applies to	Attribute blocks

Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

OLE DB Consumer Attributes

ATL Samples

db_column

Binds a specified column to a variable in the rowset.

```
[ db_column(
      ordinal,
   dbtype,
   precision,
   scale,
   status,
   length
) ]
```

Parameters

ordinal

The ordinal column number (**DBCOLUMNINFO** ordinal) or column name (ANSI or Unicode string) corresponding to a field in the rowset to which to bind data. If you use numbers, you can skip consecutive ordinals (for example: 1, 2, 3, 5). The name may contain spaces if the OLE DB provider you use supports it. For example, you can use either of the following formats:

```
[db_column("2")] TCHAR szCity[30];
[db_column(L"city_name")] TCHAR szCity[30];
```

dbtype (optional)

An OLE DB Type Indicator for the column entry.

precision (optional)

The precision to be used for the column entry. For details, see the description of the *bPrecision* element of the DBBINDING structure

scale (optional)

The scale to be used for the column entry. For details, see the description of *bScale* element of the DBBINDING structure

status (optional)

A member variable used to hold the status of this column. The status indicates whether the column value is a data value or some other value, such as **NULL**. For possible values, see Status in the *OLE DB Programmer's Reference*.

length (optional)

A member variable used to hold the size of the column in bytes.

Remarks

db_column binds the specified table column to a variable in the rowset. It delimits member data that can participate in OLE DB **IAccessor**-based binding. This attribute sets up the column map normally defined using the OLE DB consumer macros BEGIN_COLUMN_MAP, END_COLUMN_MAP, and COLUMN_ENTRY. These manipulate the OLE DB DBBINDING structure to bind the specified column. Each member you mark with the **db_column** attribute will occupy one entry in the column map in the form of a column entry. Therefore, you call this attribute where you would put the column map, that is, in the command or table class.

Use **db_column** in conjunction with either the db_table or db_command attributes.

When the consumer attribute provider applies this attribute to a class, the compiler will rename the class to _*YourClassName*Accessor, where *YourClassName* is the name you gave the class, and the compiler will also create a class called *YourClassName*, which derives from _*YourClassName*Accessor. In Class View, you will see both classes.

For examples of this attribute used in an application, see the samples AtlAgent, MultiRead, and MantaWeb.

Example

This sample binds a column in a table to a **long** data member and specifies status and length fields.

```
// db_column_1.cpp
// compile with: /LD
#include <atlbase.h>
#include <atlplus.h>
#include <atldbcli.h>

[ db_command(L"Select * from Products") ]
class CProducts {
   DBSTATUS m_dwProductIDStatus;
   DBLENGTH m_dwProductIDLength;

   [ db_column("1", status="m_dwProductIDStatus", length="m_dwProductIDLength") ] LONG m_Pr
oductID;
};
```

This sample binds four columns to a **long**, a character string, a timestamp, and a **DB_NUMERIC** integer, in that order.

```
// db_column_2.cpp
// compile with: /LD
#include <atlbase.h>
#include <atlplus.h>
#include <atldbcli.h>

[ db_command(L"Select * from Products") ]
class CProducts {
   [db_column("1")] LONG m_OrderID;
   [db_column("2")] TCHAR m_CustomerID[6];
   [db_column("4")] DB_NUMERIC m_OrderDate;
   [db_column("7", dbtype="DBTYPE_NUMERIC")] DB_NUMERIC m_ShipVia;
};
```

Requirements

Attribute Context

Applies to	**class**, **struct**, member, method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

OLE DB Consumer Attributes

Class Attributes

ATL Samples

db_command

Creates an OLE DB command.

```
[ db_command(
      command,
   name,
   source_name,
   hresult,
   bindings,
   bulk_fetch)
]
```

Parameters

command

A command string containing the text of an OLE DB command. A simple example is:

```
[ db_command ( command = "Select * from Products" ) ]
```

The *command* syntax is as follows:

```
binding parameter block 1
   OLE DB command
binding parameter block 2
   continuation of OLE DB command
binding parameter block 3
...
```

A *binding parameter block* is defined as follows:

([*bindtype* **]** *szVar1* [, *szVar2* [, *nVar3* [, ...]]] **)**

where:

(marks the start of the data binding block.

[*bindtype* **]** is one of the following case-insensitive strings:

- **[db_column]** binds each of the member variables to a column in a rowset.
- **[bindto]** (same as **[db_column]**).
- **[in]** binds member variables as input parameters.
- **[out]** binds member variables as output parameters.
- **[in,out]** binds member variables as input/output parameters.

SzVarX resolves to a member variable within the current scope.

) marks the end of the data binding block.

If the command string contains one or more specifiers such as [in], [out], or [in/out], **db_command** builds a parameter map.

If the command string contains one or more parameters such as [db_column] or [bindto], **db_command** generates a rowset and an accessor map to service these bound variables. See db_accessor for more information.

Note
[*bindtype*] syntax and the *bindings* parameter are not valid when using **db_command** at the class level.

Here are some examples of binding parameter blocks. The following example binds the `m_au_fname` and `m_au_lname` data members to the `au_fname` and `au_lname` columns, respectively, of the authors table in the pubs database:

```
TCHAR m_au_fname[21];
TCHAR m_au_lname[41];
TCHAR m_state[3] = 'CA';

[db_command (
   command = "SELECT au_fname([bindto]m_au_fname), au_lname([bindto]m_au_lname) " \
   "FROM dbo.authors " \
   "WHERE state = ?([in]m_state)")
```

]

name (optional)

The name of the handle you use to work with the rowset. If you specify *name*, **db_command** generates a class with the specified *name*, which can be used to traverse the rowset or to execute multiple action queries. If you do not specify *name*, it will not be possible to return more than one row of results to the user.

source_name (optional)

The **CSession** variable or instance of a class that has the **db_source** attribute applied to it on which the command executes. See db_source.

db_command checks to ensure that the variable used for *source_name* is valid, so the specified variable should be in function or global scope.

hresult (optional)

Identifies the variable that will receive the **HRESULT** of this database command. If the variable does not exist, it will be automatically injected by the attribute.

bindings (optional)

Allows you to separate the binding parameters from the OLE DB command.

If you specify a value for *bindings*, **db_command** will parse the associated value and will not parse the [*bindtype*] parameter. This usage allows you to use OLE DB provider syntax. To disable parsing, without binding parameters, specify **Bindings=""**.

If you do not specify a value for *bindings*, **db_command** will parse the binding parameter block, looking for '**(**', followed by **[***bindtype***]** in brackets, followed by one or more previously declared C++ member variables, followed by '**)**'. All text between the parentheses will be stripped from the resulting command, and these parameters will be used to construct column and parameter bindings for this command.

bulk_fetch(optional)

An integer value that specifies the number of rows to fetch.

The default value is 1, which specifies single row fetching (the rowset will be of type CRowset).

A value greater than 1 specifies bulk row fetching. Bulk row fetching refers to the ability of bulk rowsets to fetch multiple row handles (the rowset will be of type CBulkRowset and will call **SetRows** with the specified number of rows).

If *bulk_fetch* is less than one, **SetRows** will return zero.

Remarks

db_command creates a CCommand object, which is used by an OLE DB consumer to execute a command.

You can use **db_command** with either class or function scope; the main difference is the scope of the **CCommand** object. With function scope, data such as bindings terminate at function end. Both class and function scope usages involve the OLE DB Consumer Template class **CCommand<>**, but the template arguments differ for the function and class cases. In the function case, bindings will be made to an **Accessor** that comprises local variables, while the class usage will infer a **CAccessor**-derived class as the argument. When used as a class attribute, **db_command** works in conjunction with **db_column**.

db_command can be used to execute commands that do not return a result set.

When the consumer attribute provider applies this attribute to a class, the compiler will rename the class to _*YourClassName*Accessor, where *YourClassName* is the name you gave the class, and the compiler will also create a class called *YourClassName,* which derives from _*YourClassName*Accessor. In Class View, you will see both classes.

For sample applications that use this attribute, see the MantaWeb Sample and the OnlineAddressBook Sample.

Example

This sample defines a command that selects the first and last names from a table where the state column matches 'CA'. **db_command** creates and reads a rowset on which you can call wizard-generated functions such as OpenAll and CloseAll, as well as **CRowset** member functions such as MoveNext.

Note that this code requires you to provide your own connection string that connects to the pubs database. For information on how to do this in the development environment, see How to: Connect to a Database from Server Explorer and How to: Add New Data Connections in Server Explorer/Database Explorer.

```
// db_command.h
#include <atlbase.h>
#include <atlplus.h>
#include <atldbcli.h>

#pragma once

[  db_source(L"your connection string"),
   db_command(L" \
      SELECT au_lname, au_fname \
      FROM dbo.authors \
      WHERE state = 'CA'")  ]

struct CAuthors {
   // In order to fix several issues with some providers, the code below may bind
   // columns in a different order than reported by the provider

   DBSTATUS m_dwau_lnameStatus;
   DBSTATUS m_dwau_fnameStatus;
   DBLENGTH m_dwau_lnameLength;
   DBLENGTH m_dwau_fnameLength;

   [ db_column("au_lname", status="m_dwau_lnameStatus", length="m_dwau_lnameLength") ] TCHA
R m_au_lname[41];
   [ db_column("au_fname", status="m_dwau_fnameStatus", length="m_dwau_fnameLength") ] TCHA
R m_au_fname[21];

   [ db_param("7", paramtype="DBPARAMIO_INPUT") ] TCHAR m_state[3];


   void GetRowsetProperties(CDBPropSet* pPropSet) {
      pPropSet->AddProperty(DBPROP_CANFETCHBACKWARDS, true, DBPROPOPTIONS_OPTIONAL);
      pPropSet->AddProperty(DBPROP_CANSCROLLBACKWARDS, true, DBPROPOPTIONS_OPTIONAL);
   }
};
```

```
// db_command.cpp
// compile with: /c
#include "db_command.h"

int main(int argc, _TCHAR* argv[]) {
   HRESULT hr = CoInitialize(NULL);

   // Instantiate rowset
   CAuthors rs;

   // Open rowset and move to first row
   strcpy_s(rs.m_state, sizeof(rs.m_state), _T("CA"));
   hr = rs.OpenAll();
   hr = rs.MoveFirst();

   // Iterate through the rowset
   while( SUCCEEDED(hr) && hr != DB_S_ENDOFROWSET ) {
      // Print out the column information for each row
      printf("First Name: %s, Last Name: %s\n", rs.m_au_fname, rs.m_au_lname);
      hr = rs.MoveNext();
   }
```

```
   rs.CloseAll();
   CoUninitialize();
}
```

This sample uses **db_source** on a data source class `CMySource`, and **db_command** on command classes `CCommand1` and `CCommand2`.

```
// db_command_2.cpp
// compile with: /c
#include <atlbase.h>
#include <atlplus.h>
#include <atldbcli.h>
// class usage for both db_source and db_command

[  db_source(L"your connection string"),
   db_command(L" \
      SELECT au_lname, au_fname \
      FROM dbo.authors \
      WHERE state = 'CA'")  ]
struct CMySource {
   HRESULT OpenDataSource() {
      return S_OK;
   }
};

[db_command(command = "SELECT * FROM Products")]
class CCommand1 {};

[db_command(command = "SELECT FNAME, LNAME FROM Customers")]
class CCommand2 {};

int main() {
   CMySource s;
   HRESULT hr = s.OpenDataSource();
   if (SUCCEEDED(hr)) {
      CCommand1 c1;
      hr = c1.Open(s);

      CCommand2 c2;
      hr = c2.Open(s);
   }

   s.CloseDataSource();
}
```

Requirements

Attribute Context

Applies to	**class**, **struct**, member, method, local
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

OLE DB Consumer Attributes

Stand-Alone Attributes

ATL Samples

db_param

Associates the specified member variable with an input or output parameter and delimits the variable.

```
[ db_param(
      ordinal,
   paramtype="DBPARAMIO_INPUT",
   dbtype,
   precision,
   scale,
   status,
   length
) ]
```

Parameters

ordinal

The column number (**DBCOLUMNINFO** ordinal) corresponding to a field in the rowset to which to bind data.

paramtype (optional)

The type to set for the parameter. Providers support only parameter I/O types that are supported by the underlying data source. The type is a combination of one or more **DBPARAMIOENUM** values:

- **DBPARAMIO_INPUT** An input parameter.
- **DBPARAMIO_OUTPUT** An output parameter.
- **DBPARAMIO_NOTPARAM** The accessor has no parameters. Setting **eParamIO** to this value in row accessors reminds the user that parameters are ignored.

dbtype (optional)

An OLE DB Type Indicator for the column entry.

precision (optional)

The precision to be used for the column entry. For details, see the description of **bPrecision** element of the DBBINDING structure

scale (optional)

The scale to be used for the column entry. For details, see the description of **bScale** element of the DBBINDING structure

status (optional)

A member variable used to hold the status of this column. The status indicates whether the column value is a data value or some other value, such as **NULL**. For possible values, see Status in the *OLE DB Programmer's Reference*.

length (optional)

A member variable used to hold the size of the column in bytes.

Remarks

db_param defines parameters that you use in commands; therefore you use it with **db_command**. For example, you can use **db_param** to bind parameters in SQL queries or stored procedures. Parameters in a stored procedure are denoted by question marks (?), and you should bind the data members in the order in which the parameters appear.

db_param delimits member data that can participate in OLE DB **ICommandWithParameters**-based binding. It sets the parameter type (input or output), OLE DB type, precision, scale, status, and length for the specified parameter. This attribute inserts the OLE DB consumer macros BEGIN_PARAM_MAP ... END_PARAM_MAP. Each member you mark with the **db_param** attribute will occupy one entry in the map in the form of a COLUMN_ENTRY.

db_param is used in conjunction with either the db_table or db_command attributes.

When the consumer attribute provider applies this attribute to a class, the compiler will rename the class to _*YourClassName*Accessor, where *YourClassName* is the name you gave the class, and the compiler will also create a class

called *YourClassName*, which derives from _*YourClassName*Accessor. In Class View, you will see both classes.

Example

The following example creates a command class based on the SalesbyYear stored procedure in the Northwind database. It associates the first parameter in the stored procedure with the `m_RETURN_VALUE` variable, and defines it as an output parameter. It associates the last two (input) parameters with `m_Beginning_Date` and `m_Ending_Date`.

The following example associates the `nOutput` variable with an output parameter.

```
// db_param.cpp
// compile with: /LD
#include <atlbase.h>
#include <atlplus.h>
#include <atldbcli.h>

[ db_source(L"my_connection_string"),
  db_command(L"{ ? = CALL dbo.\"Sales by Year\"(?,?) }")
]
struct CSalesbyYear {
   DBSTATUS m_dwShippedDateStatus;
   DBSTATUS m_dwOrderIDStatus;
   DBSTATUS m_dwSubtotalStatus;
   DBSTATUS m_dwYearStatus;

   DBLENGTH m_dwShippedDateLength;
   DBLENGTH m_dwOrderIDLength;
   DBLENGTH m_dwSubtotalLength;
   DBLENGTH m_dwYearLength;

   // Bind columns
   [ db_column("1", status="m_dwShippedDateStatus", length="m_dwShippedDateLength") ] DBTIM
ESTAMP m_ShippedDate;
   [ db_column("2", status="m_dwOrderIDStatus", length="m_dwOrderIDLength") ] LONG m_OrderI
D;
   [ db_column("3", status="m_dwSubtotalStatus", length="m_dwSubtotalLength") ] CURRENCY m_
Subtotal;
   [ db_column("4", status="m_dwYearStatus", length="m_dwYearLength") ] TCHAR m_Year[31];

   // Bind parameters
   [ db_param("1", paramtype="DBPARAMIO_OUTPUT") ] LONG m_RETURN_VALUE;
   [ db_param("2", paramtype="DBPARAMIO_INPUT") ] DBTIMESTAMP m_Beginning_Date;
   [ db_param("3", paramtype="DBPARAMIO_INPUT") ] DBTIMESTAMP m_Ending_Date;
};
```

For an example of this attribute used in an application, see the MantaWeb sample.

Requirements

Attribute Context

Applies to	**class**, **struct**, member, method, local
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

OLE DB Consumer Attributes

ATL Samples

db_source

Creates a connection to a data source.

```
[ db_source(
      db_source,
   name,
   hresult
) ]
```

Parameters

db_source

The connection string used to connect to the data source. For the format of the connection string, see Connection Strings and Data Links in the Microsoft Data Access Components (MDAC) SDK.

name (optional)

When you use **db_source** on a class, *name* is an instance of a data source object that has the **db_source** attribute applied to it (see example 1). When you use **db_source** inline in a method implementation, *name* is a variable (local to the method) that can be used to access the data source (see example 2). You pass this *name* to the *source_name* parameter of **db_command** to associate the data source with a command.

hresult (optional)

Identifies the variable that will receive the **HRESULT** of this database command. If the variable does not exist, it will be automatically injected by the attribute.

Remarks

db_source creates a CDataSource and a CSession object, which together represent a connection with an OLE DB consumer data source.

When you use **db_source** on a class, the **CSession** object becomes a member of the class.

When you use **db_source** in a method, the injected code will be executed within method scope, and the **CSession** object is created as a local variable.

db_source adds data source properties to a class or within a method. It is used in conjunction with **db_command** (which takes the **db_source** *name* parameter as its *source_name* parameter).

When the consumer attribute provider applies this attribute to a class, the compiler will rename the class to _*YourClassName*Accessor, where *YourClassName* is the name you gave the class, and the compiler will also create a class called *YourClassName,* which derives from _*YourClassName*Accessor. In Class View, you will see both classes.

For an example of this attribute used in an application, see the samples AtlAgent and MultiRead.

Example

This sample calls **db_source** on a class to create a connection to the data source `ds` using the Northwind database. `ds` is a handle for the data source, which can be used internally to the `CMyCommand` class.

```
// db_source_1.cpp
// compile with: /LD
#include <atlbase.h>
#include <atlplus.h>
#include <atldbcli.h>

[
  db_source(L"my_connection_string", name="ds"),
  db_command(L"select * from Products")
]
class CMyCommand {};
```

Requirements

Attribute Context

Applies to	**class**, **struct**, member, method, local
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

OLE DB Consumer Attributes

ATL Samples

db_table

Opens an OLE DB table.

```
[ db_table(
      db_table,
   name,
   source_name,
   hresult
) ]
```

Parameters

db_table

A string specifying the name of a database table (such as "Products").

name (optional)

The name of the handle you use to work with the table. You must specify this parameter if you want to return more than one row of results. **db_table** generates a variable with the specified *name* that can be used to traverse the rowset or execute multiple action queries.

source_name (optional)

The **CSession** variable or instance of a class that has the **db_source** attribute applied to it on which the command executes. See db_source.

hresult (optional)

Identifies the variable that will receive the **HRESULT** of this database command. If the variable does not exist, it will be automatically injected by the attribute.

Remarks

db_table creates a CTable object, which is used by an OLE DB consumer to open a table. You can use this attribute only at the class level; you cannot use it inline. Use **db_column** to bind table columns to variables; use **db_param** to delimit (set the parameter type and so on) of parameters.

When the consumer attribute provider applies this attribute to a class, the compiler will rename the class to _*YourClassName*Accessor, where *YourClassName* is the name you gave the class, and the compiler will also create a class called *YourClassName*, which derives from _*YourClassName*Accessor. In Class View, you will see both classes.

Example

The following example opens the Products table for use by `CProducts`.

```
// db_table.cpp
// compile with: /LD
#include <atlbase.h>
#include <atlplus.h>
#include <atldbcli.h>

[ db_table(L"dbo.Products") ]
class CProducts {
   [ db_column("1") ] LONG m_ProductID;
};
```

For an example of this attribute used in an application, see the samples AtlAgent and MultiRead.

Requirements

Attribute Context

Applies to	**class**, **struct**

Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

OLE DB Consumer Attributes

ATL Samples

default (C++)

Indicates that the custom or dispinterface defined within a coclass represents the default programmability interface.

```
[ default(
      interface1,
   interface2
) ]
```

Parameters

interface1

The default interface that will be made available to scripting environments that create an object based on the class defined with the **default** attribute.

If no default interface is specified, the first occurrence of a nonsource interface is used as the default.

interface2(optional)

The default source interface. You must also specify this interface with the source attribute.

If no default source interface is specified, the first source interface is used as the default.

Remarks

The **default** C++ attribute has the same functionality as the default MIDL attribute. The **default** attribute is also used with the case attribute.

Example

The following code shows how **default** is used on the definition of a coclass to specify **ICustomDispatch** as the default programmability interface:

```
// cpp_attr_ref_default.cpp
// compile with: /LD
#include "windows.h"
[module(name="MyLibrary")];

[object, uuid("9E66A290-4365-11D2-A997-00C04FA37DDB")]
__interface ICustom {
   HRESULT Custom([in] long l, [out, retval] long *pLong);
};

[dual, uuid("9E66A291-4365-11D2-A997-00C04FA37DDB")]
__interface IDual {
   HRESULT Dual([in] long l, [out, retval] long *pLong);
};

[object, uuid("9E66A293-4365-11D2-A997-00C04FA37DDB")]
__interface ICustomDispatch : public IDispatch {
   HRESULT Dispatch([in] long l, [out, retval] long *pLong);
};

[   coclass,
   default(ICustomDispatch),
   source(IDual),
   uuid("9E66A294-4365-11D2-A997-00C04FA37DDB")
]
class CClass : public ICustom, public IDual, public ICustomDispatch {
   HRESULT Custom(long l, long *pLong) { return(S_OK); }
   HRESULT Dual(long l, long *pLong) { return(S_OK); }
   HRESULT Dispatch(long l, long *pLong) { return(S_OK); }
};

int main() {
```

```
#if 0 // Can't instantiate without implementations of IUnknown/IDispatch
   CClass *pClass = new CClass;

   long llong;

   pClass->custom(1, &llong);
   pClass->dual(1, &llong);
   pClass->dispinterface(1, &llong);
   pClass->dispatch(1, &llong);

   delete pClass;
#endif
   return(0);
}
```

The source attribute also has an example of how to use **default**.

Requirements

Attribute Context

Applies to	**class**, **struct**, data member
Repeatable	No
Required attributes	**coclass** (when applied to **class** or **struct**)
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

coclass

Other Resources

IDL Attributes

Class Attributes

ATL Samples

defaultbind

Indicates the single, bindable property that best represents the object.

```
[defaultbind]
```

Remarks

The **defaultbind** C++ attribute has the same functionality as the defaultbind MIDL attribute.

Example

See the example for bindable for an example of how to use **defaultbind**.

Requirements

Attribute Context

Applies to	Interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

displaybind
immediatebind
requestedit

Other Resources

IDL Attributes
Method Attributes
Data Member Attributes
ATL Samples

defaultcollelem

Used for Visual Basic code optimization.

```
[defaultcollelem]
```

Remarks

The **defaultcollelem** C++ attribute has the same functionality as the defaultcollelem MIDL attribute.

Example

The following code shows an interface method using the **defaultcollelem** attribute:

```
// cpp_attr_ref_defaultcollelem.cpp
// compile with: /LD
#include <unknwn.h>
[module(name="MyLib")];
[object, uuid("00000000-0000-0000-0000-000000000001")]
__interface IMyForm
{
   [propget, id(1), bindable, defaultcollelem, displaybind,
   defaultbind, requestedit] HRESULT P1([out, retval] long *nSize);
   [propput, id(1), bindable, defaultcollelem, displaybind,
   defaultbind, requestedit] HRESULT P1([in] long nSize);
};
```

Requirements

Attribute Context

Applies to	Interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Method Attributes

ATL Samples

defaultvalue

Allows specification of a default value for a typed optional parameter.

```
[ defaultvalue= value ]
```

Parameters

value

The default value for the parameter.

Remarks

The **defaultvalue** C++ attribute has the same functionality as the defaultvalue MIDL attribute.

Example

The following code shows an interface method using the **defaultvalue** attribute:

```
// cpp_attr_ref_defaultvalue.cpp
// compile with: /LD
#include <windows.h>

[export] typedef long HRESULT;
[export, ptr, string] typedef unsigned char * MY_STRING_TYPE;

[  uuid("479B29EE-9A2C-11D0-B696-00A0C903487A"),
   dual, oleautomation,
   helpstring("IFireTabCtrl Interface"),
   helpcontext(122), pointer_default(unique) ]

__interface IFireTabCtrl : IDispatch {
   [bindable, propget] HRESULT get_Size([out, retval, defaultvalue("33")] long *nSize);
   [bindable, propput] HRESULT put_Size([in] int nSize);
};

[ module(name="ATLFIRELib", uuid="479B29E1-9A2C-11D0-B696-00A0C903487A",
      version="1.0", helpstring="ATLFire 1.0 Type Library") ];
```

Requirements

Attribute Context

Applies to	Interface parameter
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

out (C++)
retval
in (C++)
pointer_default
unique (C++)

Other Resources

IDL Attributes

defaultvtable

Defines an interface as the default vtable interface for a COM object.

```
[ defaultvtable(
      interface
) ]
```

Parameters

interface

The designated interface that you want to have the default vtable for the COM object.

Remarks

The **defaultvtable** C++ attribute has the same functionality as the defaultvtable MIDL attribute.

Example

The following code shows attributes on a class that use **defaultvtable** to specify a default interface:

```
// cpp_attr_ref_defaultvtable.cpp
// compile with: /LD
#include <unknwn.h>
[module(name="MyLib")];

[object, uuid("00000000-0000-0000-0000-000000000001")]
__interface IMyI1 {
   HRESULT x();
};

[object, uuid("00000000-0000-0000-0000-000000000002")]
__interface IMyI2 {
   HRESULT x();
};

[object, uuid("00000000-0000-0000-0000-000000000003")]
__interface IMyI3 {
   HRESULT x();
};

[coclass, source(IMyI3, IMyI1), default(IMyI3, IMyI2), defaultvtable(IMyI1),
uuid("00000000-0000-0000-0000-000000000004")]
class CMyC3 : public IMyI3 {};
```

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	No
Required attributes	**coclass**
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Class Attributes

dispinterface

Places an interface in the .idl file as a dispatch interface.

```
[dispinterface]
```

Remarks

When the **dispinterface** C++ attribute precedes an interface, it causes the interface to be placed inside the library block in the generated .idl file.

Unless you specify a base class, a dispatch interface will derive from **IDispatch**. You must specify an id for the members of a dispatch interface.

The usage example for dispinterface in the MIDL documentation:

```
dispinterface helloPro
   { interface hello; };
```

is not valid for the **dispinterface** attribute.

Example

See the example for bindable for an example of how to use **dispinterface**.

Requirements

Attribute Context

Applies to	**interface**
Repeatable	No
Required attributes	None
Invalid attributes	**dual**, **object**, **oleautomation**, **local**, **ms_union**

For more information, see Attribute Contexts.

See Also

Reference

uuid (C++ Attributes)
dual
custom (C++)
object (C++)
__interface

Other Resources

IDL Attributes
Attributes by Usage
ATL Samples

displaybind

Indicates a property that should be displayed to the user as bindable.

```
[displaybind]
```

Remarks

The **displaybind** C++ attribute has the same functionality as the displaybind MIDL attribute.

Example

See the example for bindable for an example of how to use **displaybind**.

Requirements

Attribute Context

Applies to	Interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

defaultbind
immediatebind
requestedit

Other Resources

IDL Attributes
Method Attributes
Data Member Attributes
ATL Samples

dual

Places an interface in the .idl file as a dual interface.

```
[dual]
```

Remarks

When the **dual** C++ attribute precedes an interface, it causes the interface to be placed inside the library block in the generated .idl file.

Example

The following code is an attribute block that uses **dual** before an interface definition:

```
// cpp_attr_ref_dual.cpp
// compile with: /LD
#include <windows.h>
[module(name="MyLibrary")];

[uuid("2F5F63F1-16DA-11d2-9E7B-00C04FB926DA"), dual]

__interface IStatic : IDispatch
{
   HRESULT Func1(int i);
   [   propget,
      id(1),
      bindable,
      displaybind,
      defaultbind,
      requestedit
   ]
   HRESULT P1([out, retval] long *nSize);
   [   propput,
      id(1),
      bindable,
      displaybind,
      defaultbind,
      requestedit
   ]
   HRESULT P1([in] long nSize);
};

[cpp_quote("#include file.h")];
```

Requirements

Attribute Context

Applies to	**interface**
Repeatable	No
Required attributes	None
Invalid attributes	**dispinterface**

For more information, see Attribute Contexts.

See Also

Reference

custom (C++)

dispinterface
object (C++)
__interface

Other Resources

IDL Attributes
Attributes by Usage
ATL Samples

emitidl

Determines whether all subsequent IDL attributes will be processed and placed in the generated .idl file.

```
[ emitidl([boolean],
      defaultimports=[boolean]
) ] ;
```

Parameters

boolean

Possible values: **true**, **false**, **forced**, **restricted**, **push**, or **pop**.

- If **true**, any IDL category attributes encountered in a source code file will be placed in the generated .idl file. This is the default setting for **emitidl**.
- If **false**, any IDL category attributes encountered in a source code file will not be placed in the generated .idl file.
- If **restricted**, allows IDL attributes to be in the file without a module attribute. The compiler will not generate an .idl file.
- If **forced**, overrides a subsequent **restricted** attribute, which requires a file to have a **module** attribute if there are IDL attributes in the file.
- **push** lets you save the current **emitidl** settings to an internal **emitidl** stack, and **pop** lets you set **emitidl** to whatever value is at the top of the internal **emitidl** stack.

defaultimports =[*boolean*] (optional)

- If *boolean* is **true**, docobj.idl will be imported into the generated .idl file. Also, if an .idl file with the same name as an .h file that you **#include** into your source code is found in the same directory as the .h file, then the generated .idl file will contain an import statement for that .idl file.
- If *boolean* is **false**, docobj.idl will not be imported into the generated .idl file. You will need to explicitly import .idl files with import.

Remarks

After the **emitidl** C++ attribute is encountered in a source code file, IDL category attributes will be placed in the generated .idl file. If there is no **emitidl** attribute, IDL attributes in the source code file will be output to the generated .idl file.

It is possible to have multiple **emitidl** attributes in a source code file. If `[emitidl(false)];` is encountered in a file without a subsequent `[emitidl(true)];`, then no attributes will be processed into the generated .idl file.

Each time the compiler encounters a new file, **emitidl** is implicitly set to **true**.

Requirements

Attribute Context

Applies to	Anywhere
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Other Resources

Compiler Attributes

Stand-Alone Attributes

ATL Samples

entry

Specifies an exported function or constant in a module by identifying the entry point in the DLL.

```
[ entry(
      id
) ]
```

Parameters

id

The ID of the entry point.

Remarks

The **entry** C++ attribute has the same functionality as the entry MIDL attribute.

Example

See the example for idl_module for an example use of **entry**.

Requirements

Attribute Context

Applies to	**idl_module** attribute
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

ATL Samples

event_receiver

Creates an event receiver (sink).

```
[ event_receiver(
      type
   [, layout_dependent=false]
) ]
```

Parameters

type

An enumeration of one of the following values:

- **native** for unmanaged C/C++ code (default for native classes).
- **com** for COM code. This value requires that you include the following header files:

```
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atlcom.h>
```

layout_dependent

Specify *layout_dependent* only if *type*=**com**. *layout_dependent* is a Boolean:

- **true** means that the signature of the delegates in the event receiver must exactly match those to which they are hooked in the event source. The event receiver handler names must match the names specified in the relevant event source interface. You must use **coclass** when *layout_dependent* is **true**. It is slightly more efficient to specify **true**.
- **false** (default) means that the calling convention and storage class (virtual, static, and others) do not have to match the event method and the handlers; nor do the handler names need to match the event source interface method names.

Remarks

The **event_receiver** C++ attribute specifies that the class or structure to which it is applied will be an event receiver, using the Visual C++ unified event model.

event_receiver is used with the event_source attribute and the __hook and __unhook keywords. Use **event_source** to create event sources. Use **__hook** within an event receiver's methods to associate ("hook") event receiver methods to the events of an event source. Use **__unhook** to dissociate them.

layout_dependent is only specified for COM event receivers (*type*=**com**). The default for *layout_dependent* is **false**.

Note
A templated class or struct cannot contain events.

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	No
Required attributes	**coclass** when *layout_dependent*=**true**
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

event_source

__event

__hook

__unhook

Other Resources

Compiler Attributes

Class Attributes

ATL Samples

event_source

Creates an event source.

```
[ event_source(
      type,
   optimize=[speed | size],
   decorate=[true | false]
) ]
```

Parameters

type

An enumeration of one of the following values:

- **native** for unmanaged C/C++ code (default for unmanaged classes).
- **com** for COM code. You must use **coclass** when *type*=**com**. This value requires that you include the following header files:

```
#define _ATL_ATTRIBUTES
#include <atlbase.h>
#include <atlcom.h>
```

optimize

When *type* is **native**, you can specify **optimize=size**, to indicate that there is 4 bytes of storage (minimum) for all events in a class or **optimize=speed** (the default) to indicate that there is 4 * (# of events) bytes of storage.

decorate

When *type* is **native**, you can specify **decorate=false**, to indicate that the expanded name in the merged (.mrg) file should not include the enclosing class name. /Fx lets you generate .mrg files. **decorate=false**, which is the default, results in fully-qualified type names in the merged file.

Remarks

The **event_source** C++ attribute specifies that the class or structure to which it is applied will be an event source.

event_source is used in conjunction with the event_receiver attribute and the __event keyword. Use **event_receiver** to create event receivers. Use **__event** on methods within the event source to specify those methods as events.

Note
A templated class or struct cannot contain events.

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	No
Required attributes	**coclass** when *type*=**com**
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

event_receiver
__event
__hook
__unhook

Other Resources

Compiler Attributes
Class Attributes
ATL Samples

export

Causes a data structure to be placed in the .idl file.

```
[export]
```

Remarks

The **export** C++ attribute causes a data structure to be placed in the .idl file and to then be available in the type library in a binary-compatible format that makes it available for use with any language.

You cannot apply the **export** attribute to a class even if the class only has public members (the equivalent of a **struct**).

If you export unnamed **enum**s or **struct**s, they will be given names that begin with **__unnamed***x*, where *x* is a sequential number.

The typedefs valid for export are base types, structs, unions, enums, or type identifiers. See typedef for more information.

Example

The following code shows how to use the **export** attribute:

```
// cpp_attr_ref_export.cpp
// compile with: /LD
[module(name="MyLibrary")];

[export]
struct MyStruct {
   int i;
};
```

For other examples, see the MantaWeb Sample and the WeatherService Sample.

Requirements

Attribute Context

Applies to	**union**, **typedef**, **enum**, **struct**, or **interface**
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Other Resources

Compiler Attributes
Typedef, Enum, Union, and Struct Attributes
ATL Samples

first_is

Specifies the index of the first array element to be transmitted.

```
[ first_is(
   "expression"
) ]
```

Parameters

expression

One or more C-language expressions. Empty argument slots are allowed.

Remarks

The **first_is** C++ attribute has the same functionality as the first_is MIDL attribute.

Example

The following code shows various ways to specify a section in an array:

```
// cpp_attr_ref_first_is.cpp
// compile with: /LD
#include "windows.h"
#include "unknwn.h"

[module(name="MyLib")];

[object, uuid(11111111-1111-1111-1111-111111111111)]
__interface b
{
   [id(0), propget, bindable, displaybind, defaultbind,
requestedit] HRESULT get_I([out, retval]long *i);
   HRESULT Proc1([in] short First, [in] short Last,
[first_is(First), last_is(Last), size_is(Last-First)] char Arr1[]);
   HRESULT Proc2([in] short First, [in] short Last,
[last_is(First), size_is(Last)] char Arr2[]);
};
```

Requirements

Attribute Context

Applies to	Field in **struct** or **union**, interface parameter, interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

last_is
max_is
length_is
size_is

Other Resources

IDL Attributes
Typedef, Enum, Union, and Struct Attributes

helpcontext

Specifies a context ID that lets the user view information about this element in the Help file.

```
[ helpcontext(
      id
) ]
```

Parameters

id

The context ID of the help topic. See Help Topics (WinHelp): Context-Sensitive Help for Your Programs for more information on context IDs.

Remarks

The **helpcontext** C++ attribute has the same functionality as the helpcontext MIDL attribute.

Example

See the example for defaultvalue for an example of how to use **helpcontext**.

Requirements

Attribute Context

Applies to	**interface**, **typedef**, **class**, method, property
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

helpfile

helpstring

Other Resources

IDL Attributes

Interface Attributes

Class Attributes

Method Attributes

Typedef, Enum, Union, and Struct Attributes

ATL Samples

helpfile

Sets the name of the Help file for a type library.

```
[ helpfile(
   "filename"
) ]
```

Parameters

filename

The name of the file that contains the help topics.

Remarks

The **helpfile** C++ attribute has the same functionality as the helpfile MIDL attribute.

Example

See the example for module for an example of how to use **helpfile**.

Requirements

Attribute Context

Applies to	**interface**, **typedef**, **class**, method, property
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

helpcontext
helpstring

Other Resources

IDL Attributes
Interface Attributes
Class Attributes
Method Attributes
Typedef, Enum, Union, and Struct Attributes
ATL Samples

helpstring

Specifies a character string that is used to describe the element to which it applies.

```
[ helpstring(
   "string"
) ]
```

Parameters

string

The text of the help string.

Remarks

The **helpstring** C++ attribute has the same functionality as the helpstring MIDL attribute.

Example

See the example for defaultvalue for an example of how to use **helpstring**.

Requirements

Attribute Context

Applies to	**interface**, **typedef**, **class**, method, property
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

helpfile
helpcontext

Other Resources

IDL Attributes
Interface Attributes
Class Attributes
Method Attributes
Typedef, Enum, Union, and Struct Attributes
ATL Samples

helpstringcontext

Specifies the ID of a help topic in an .hlp or .chm file.

```
[ helpstringcontext(
      contextID
) ]
```

Parameters

contextID

A 32-bit Help context identifier in the Help file.

Remarks

The **helpstringcontext** C++ attribute has the same functionality as the helpstringcontext ODL attribute.

Example

```
// cpp_attr_ref_helpstringcontext.cpp
// compile with: /LD
#include <unknwn.h>
[module(name="MyLib")];

[   object,
   helpstring("help string"),
   helpstringcontext(1),
   uuid="11111111-1111-1111-1111-111111111111"
]
__interface IMyI
{
   HRESULT xx();
};
```

Requirements

Attribute Context

Applies to	**class**, **interface**, interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

module (C++)

Other Resources

IDL Attributes

Interface Attributes

Class Attributes

Method Attributes

ATL Samples

helpstringdll

Specifies the name of the DLL to use to perform document string lookup (localization).

```
[ helpstringdll(
   "string"
) ]
```

Parameters

string

The DLL to use to perform document string lookup.

Remarks

The **helpstringdll** C++ attribute has the same functionality as the helpstringdll MIDL attribute.

Example

```
// cpp_attr_ref_helpstringdll.cpp
// compile with: /LD
#include <unknwn.h>
[module(name="MyLib", helpstringdll="xx.dll")];

[object, uuid("00000000-0000-0000-0000-000000000001")]
__interface IMyI
{
   HRESULT xxx();
};
```

Requirements

Attribute Context

Applies to	**class**, **interface**, interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Other Resources

IDL Attributes
Interface Attributes
Class Attributes
Method Attributes
ATL Samples

hidden

Indicates that the item exists but should not be displayed in a user-oriented browser.

```
[hidden]
```

Remarks

The **hidden** C++ attribute has the same functionality as the hidden MIDL attribute.

Example

See the example for bindable for an example of how to use **hidden**.

Requirements

Attribute Context

Applies to	**interface**, **class**, **struct**, method, property
Repeatable	No
Required attributes	**coclass** (when applied to **class** or **struct**)
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Other Resources

IDL Attributes
Interface Attributes
Class Attributes
Method Attributes
ATL Samples

id

Specifies a *dispid* parameter for a member function (either a property or a method, in an interface or dispinterface).

```
[ id(
      dispid
) ]
```

Parameters

dispid

The dispatch ID for the interface method.

Remarks

The **id** C++ attribute has the same functionality as the id MIDL attribute.

Example

See the example for bindable for an example of how to use **id**.

Requirements

Attribute Context

Applies to	Interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

defaultvalue
in (C++)
out (C++)

Other Resources

IDL Attributes
Method Attributes
Data Member Attributes
ATL Samples

idl_module

Specifies an entry point in a .dll file.

```
[ idl_module (
   name=module_name,
      dllname=dll,
      uuid="uuid",
      helpstring="help text",
      helpstringcontext=helpcontextID,
      helpcontext=helpcontext,
      hidden,
      restricted
) ]
function declaration
```

Parameters

name

A user-defined name for the code block that will appear in the .idl file.

dllname (optional)

The .dll file that contains the export.

uuid (optional)

A unique ID.

helpstring (optional)

A character string used to describe the type library.

helpstringcontext (optional)

The ID of a help topic in an .hlp or .chm file.

helpcontext (optional)

The Help ID for this type library.

hidden (optional)

A parameter that prevents the library from being displayed. See the hidden MIDL attribute for more information.

restricted (optional)

Members of the library cannot be arbitrarily called. See the restricted MIDL attribute for more information.

function declaration

The function that you will define.

Remarks

The **idl_module** C++ attribute lets you specify the entry point in a .dll file, which allows you to import from a .dll file.

The **idl_module** attribute has functionality similar to the module MIDL attribute.

You can export anything from a COM object that you can export from a .dll file by putting a DLL entry point in the library block of an .idl file.

Your must use **idl_module** in two steps. First, you must define a name/DLL pair. Then, when you use **idl_module** to specify an entry point, specify the name and any additional attributes.

Example

The following code shows how to use the **idl_module** attribute:

```
// cpp_attr_ref_idl_module.cpp
// compile with: /LD
[idl_quote("midl_pragma warning(disable:2461)")];
[module(name="MyLibrary"), idl_module(name="MyLib", dllname="xxx.dll")];
[idl_module(name="MyLib"), entry(4), usesgetlasterror]
void FuncName(int i);
```

Requirements

Attribute Context

Applies to	Anywhere
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

entry

Other Resources

IDL Attributes

Stand-Alone Attributes

ATL Samples

idl_quote

Allows you to use IDL constructs that are not supported in the current version of Visual C++ and have them pass through to the generated .idl file.

```
[ idl_quote(
      text
) ]
```

Parameters

text

The attribute name that you intend the Visual C++ compiler to pass through to the generated .idl file without returning a compiler error.

Remarks

If the **idl_quote** C++ attribute is used as a stand-alone attribute (with a semicolon after the closing bracket), then *text* is placed in the merged .idl file as is. If **idl_quote** is used on a symbol, *text* is placed within the attribute block for that symbol.

Example

The following code shows how you could specify an unsupported attribute (using **in**, which is supported) and how to define and use an undefined .idl construct:

```
// cpp_attr_ref_idl_quote.cpp
// compile with: /LD
#include <unknwn.h>
[module(name="MyLibrary")];

[export]
struct MYFLOT {
   int i;
};

[export]
struct MYDUB {
   int i;
};

[idl_quote("typedef union _S1_TYPE switch (long l1) U1_TYPE { case 1024: \
struct MYFLOT f1; case 2048: struct MYDUB d2; } S1_TYPE;") ];

typedef struct _S1_TYPE {
   long l1;

union {
   MYFLOT f1; MYDUB d2; } U1_TYPE;
} S1_TYPE;

[uuid("2F5F63F1-16DA-11d2-9E7B-00C04FB926DA"), object]
__interface IStatic{
   HRESULT Func1([idl_quote("in")] int i);
   HRESULT func( S1_TYPE* myStruct );
};
```

This code causes MYFLOT and MYDUB and the *text* entry to be placed in the generated .idl file. The *name* parameter forces *text* to be placed before anything that references *name* in the generated .idl file. The *dependencies* parameter forces the dependency list definitions to be placed before *text* in the generated .idl file.

Requirements

Attribute Context

Applies to	Anywhere
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Stand-Alone Attributes

ATL Samples

iid_is

Specifies the IID of the COM interface pointed to by an interface pointer.

```
[ iid_is(
   "expression"
) ]
```

Parameters

expression

A C language expression that specifies an IID of a COM interface pointed to by an interface pointer.

Remarks

The **iid_is** C++ attribute has the same functionality as the iid_is MIDL attribute.

Example

The following code shows the use of **iid_is**:

```
// cpp_attr_ref_iid_is.cpp
// compile with: /LD
#include "wtypes.h"
#include "unknwn.h"
[dispinterface, uuid("00000000-0000-0000-0000-000000000001")]
__interface IFireTabCtrl : IDispatch
{
   [id(1)] HRESULT CreateInstance([in] REFIID riid,[out, iid_is("riid")]
   IUnknown ** ppvObject);
};

[module(name="ATLFIRELib")];
```

Requirements

Attribute Context

Applies to	Interface parameter, data member
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Parameter Attributes

ATL Samples

immediatebind

Indicates that the database will be notified immediately of all changes to a property of a data-bound object.

```
[immediatebind]
```

Remarks

The **immediatebind** C++ attribute has the same functionality as the immediatebind MIDL attribute.

Example

See bindable for an example of how to use **immediatebind**.

Requirements

Attribute Context

Applies to	Interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

defaultbind
displaybind
requestedit

Other Resources

IDL Attributes
Method Attributes
ATL Samples

implements (C++)

Specifies dispatch interfaces that are forced to be members of the IDL coclass.

```
[ implements(
   interfaces={interfaces},
      dispinterfaces={dispinterfaces}
)]
```

Parameters

interfaces

A comma separated list of the interfaces that will be a member of the IDL coclass. A shorthand method for specifying a single interface is **implements(***interface_name***)**.

dispinterfaces

A comma separated list of the dispinterface that will be a member of the IDL coclass. A shorthand method for specifying a single dispinterface is **implements(dispinterfaces** = *dispinterface_name***)**.

Remarks

By default, only COM-interfaces that are base classes of the coclass are added in the IDL coclass. **implements** lets you force other interfaces to be IDL coclass members.

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	Yes
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

Example

The following example is in three parts: an .idl file and its associated .h file, and a C++ file.

Assume the following .idl file, which will be available to the compiler.

```
// attr_implements.idl
import "docobj.idl";
[ version(1.0), uuid(0ed71801-a1b6-3178-af3b-9431fc00185e) ]
library odod
{
   importlib("stdole2.tlb");
   importlib("olepro32.dll");

   [
      object,
      uuid(1AECC9BB-2104-3723-98B8-7CC54722C7DD)
   ]
   interface IBar1 {
      [id(1)] HRESULT bar1();
   };

   [
      dual,
      uuid(1AECCABB-2104-3723-98B8-7CC54722C7DD)
```

```
    ]
    interface IBar2 {
       [id(1)] HRESULT bar2();
    };

    [
       uuid(1AECC9CC-2104-3723-98B8-7CC54722C7DD)
    ]
    dispinterface ISna {
    properties:

    methods:
       [id(1)] HRESULT sna();
    };

    [
       uuid(159A9BBB-E5F1-33F6-BEF5-6CFAD7A5933F),
       version(1.0)
    ]
    coclass CBar {
       interface IBar1;
       interface IBar2;
       dispinterface ISna;
    };
}
```

And the following .h file, which also needs to be available to the compiler.

```
// attr_implements.h
// this ALWAYS GENERATED file contains definitions for the interfaces

/* File created by MIDL compiler version 6.00.0361 */
/* at Mon Feb 28 16:30:53 2005 */
/* Compiler settings for attr_implements.idl:
    Oicf, W1, Zp8, env=Win32 (32b run)
    protocol : dce , ms_ext, c_ext, robust
    error checks: allocation ref bounds_check enum stub_data
    VC __declspec() decoration level:
         __declspec(uuid()), __declspec(selectany), __declspec(novtable)
         DECLSPEC_UUID(), MIDL_INTERFACE()
*/
//@@MIDL_FILE_HEADING(  )

#pragma warning( disable: 4049 )  /* more than 64k source lines */

/* verify that the <rpcndr.h> version is high enough to compile this file*/
#ifndef __REQUIRED_RPCNDR_H_VERSION__
#define __REQUIRED_RPCNDR_H_VERSION__ 475
#endif

#include "rpc.h"
#include "rpcndr.h"

#ifndef __RPCNDR_H_VERSION__
#error this stub requires an updated version of <rpcndr.h>
#endif // __RPCNDR_H_VERSION__

#ifndef __attr_implements_h__
#define __attr_implements_h__

#if defined(_MSC_VER) && (_MSC_VER >= 1020)
#pragma once
#endif

/* Forward Declarations */

#ifndef __IBar1_FWD_DEFINED__
```

```
#define __IBar1_FWD_DEFINED__
typedef interface IBar1 IBar1;
#endif /* __IBar1_FWD_DEFINED__ */

#ifndef __IBar2_FWD_DEFINED__
#define __IBar2_FWD_DEFINED__
typedef interface IBar2 IBar2;
#endif /* __IBar2_FWD_DEFINED__ */

#ifndef __ISna_FWD_DEFINED__
#define __ISna_FWD_DEFINED__
typedef interface ISna ISna;
#endif /* __ISna_FWD_DEFINED__ */

#ifndef __CBar_FWD_DEFINED__
#define __CBar_FWD_DEFINED__

#ifdef __cplusplus
typedef class CBar CBar;
#else
typedef struct CBar CBar;
#endif /* __cplusplus */

#endif /* __CBar_FWD_DEFINED__ */

/* header files for imported files */
#include "docobj.h"

#ifdef __cplusplus
extern "C"{
#endif

void * __RPC_USER MIDL_user_allocate(size_t);
void __RPC_USER MIDL_user_free( void * );

#ifndef __odod_LIBRARY_DEFINED__
#define __odod_LIBRARY_DEFINED__

/* library odod */
/* [uuid][version] */

EXTERN_C const IID LIBID_odod;

#ifndef __IBar1_INTERFACE_DEFINED__
#define __IBar1_INTERFACE_DEFINED__

/* interface IBar1 */
/* [uuid][object] */

EXTERN_C const IID IID_IBar1;

#if defined(__cplusplus) && !defined(CINTERFACE)

    MIDL_INTERFACE("1AECC9BB-2104-3723-98B8-7CC54722C7DD")
    IBar1
    {
    public:
        BEGIN_INTERFACE
        virtual /* [id] */ HRESULT STDMETHODCALLTYPE bar1( void) = 0;

        END_INTERFACE
    };

#else /* C style interface */

    typedef struct IBar1Vtbl
    {
        BEGIN_INTERFACE
```

```
        /* [id] */ HRESULT ( STDMETHODCALLTYPE *bar1 )(
            IBar1 * This);

        END_INTERFACE
    } IBar1Vtbl;

    interface IBar1
    {
        CONST_VTBL struct IBar1Vtbl *lpVtbl;
    };

#ifdef COBJMACROS

#define IBar1_bar1(This)\
    (This)->lpVtbl -> bar1(This)

#endif /* COBJMACROS */

#endif /* C style interface */

/* [id] */ HRESULT STDMETHODCALLTYPE IBar1_bar1_Proxy(
    IBar1 * This);

void __RPC_STUB IBar1_bar1_Stub(
    IRpcStubBuffer *This,
    IRpcChannelBuffer *_pRpcChannelBuffer,
    PRPC_MESSAGE _pRpcMessage,
    DWORD *_pdwStubPhase);

#endif /* __IBar1_INTERFACE_DEFINED__ */

#ifndef __IBar2_INTERFACE_DEFINED__
#define __IBar2_INTERFACE_DEFINED__

/* interface IBar2 */
/* [auto_handle][uuid][dual] */

EXTERN_C const IID IID_IBar2;

#if defined(__cplusplus) && !defined(CINTERFACE)

    MIDL_INTERFACE("1AECCABB-2104-3723-98B8-7CC54722C7DD")
    IBar2
    {
    public:
        BEGIN_INTERFACE
        virtual /* [id] */ HRESULT STDMETHODCALLTYPE bar2( void) = 0;

        END_INTERFACE
    };

#else /* C style interface */

    typedef struct IBar2Vtbl
    {
        BEGIN_INTERFACE

        /* [id] */ HRESULT ( STDMETHODCALLTYPE *bar2 )(
            IBar2 * This);

        END_INTERFACE
    } IBar2Vtbl;

    interface IBar2
    {
        CONST_VTBL struct IBar2Vtbl *lpVtbl;
    };
```

```
#ifdef COBJMACROS

#define IBar2_bar2(This)\
    (This)->lpVtbl -> bar2(This)

#endif /* COBJMACROS */

#endif /* C style interface */

/* [id] */ HRESULT STDMETHODCALLTYPE IBar2_bar2_Proxy(
    IBar2 * This);

void __RPC_STUB IBar2_bar2_Stub(
    IRpcStubBuffer *This,
    IRpcChannelBuffer *_pRpcChannelBuffer,
    PRPC_MESSAGE _pRpcMessage,
    DWORD *_pdwStubPhase);

#endif /* __IBar2_INTERFACE_DEFINED__ */

#ifndef __ISna_DISPINTERFACE_DEFINED__
#define __ISna_DISPINTERFACE_DEFINED__

/* dispinterface ISna */
/* [uuid] */

EXTERN_C const IID DIID_ISna;

#if defined(__cplusplus) && !defined(CINTERFACE)

    MIDL_INTERFACE("1AECC9CC-2104-3723-98B8-7CC54722C7DD")
    ISna : public IDispatch
    {
    };

#else /* C style interface */

    typedef struct ISnaVtbl
    {
        BEGIN_INTERFACE

        HRESULT ( STDMETHODCALLTYPE *QueryInterface )(
            ISna * This,
            /* [in] */ REFIID riid,
            /* [iid_is][out] */ void **ppvObject);

        ULONG ( STDMETHODCALLTYPE *AddRef )(
            ISna * This);

        ULONG ( STDMETHODCALLTYPE *Release )(
            ISna * This);

        HRESULT ( STDMETHODCALLTYPE *GetTypeInfoCount )(
            ISna * This,
            /* [out] */ UINT *pctinfo);

        HRESULT ( STDMETHODCALLTYPE *GetTypeInfo )(
            ISna * This,
            /* [in] */ UINT iTInfo,
            /* [in] */ LCID lcid,
            /* [out] */ ITypeInfo **ppTInfo);

        HRESULT ( STDMETHODCALLTYPE *GetIDsOfNames )(
            ISna * This,
            /* [in] */ REFIID riid,
            /* [size_is][in] */ LPOLESTR *rgszNames,
            /* [in] */ UINT cNames,
```

```
            /* [in] */ LCID lcid,
            /* [size_is][out] */ DISPID *rgDispId);

        /* [local] */ HRESULT ( STDMETHODCALLTYPE *Invoke )(
            ISna * This,
            /* [in] */ DISPID dispIdMember,
            /* [in] */ REFIID riid,
            /* [in] */ LCID lcid,
            /* [in] */ WORD wFlags,
            /* [out][in] */ DISPPARAMS *pDispParams,
            /* [out] */ VARIANT *pVarResult,
            /* [out] */ EXCEPINFO *pExcepInfo,
            /* [out] */ UINT *puArgErr);

        END_INTERFACE
    } ISnaVtbl;

    interface ISna
    {
        CONST_VTBL struct ISnaVtbl *lpVtbl;
    };

#ifdef COBJMACROS

#define ISna_QueryInterface(This,riid,ppvObject)\
    (This)->lpVtbl -> QueryInterface(This,riid,ppvObject)

#define ISna_AddRef(This)\
    (This)->lpVtbl -> AddRef(This)

#define ISna_Release(This)\
    (This)->lpVtbl -> Release(This)

#define ISna_GetTypeInfoCount(This,pctinfo)\
    (This)->lpVtbl -> GetTypeInfoCount(This,pctinfo)

#define ISna_GetTypeInfo(This,iTInfo,lcid,ppTInfo)\
    (This)->lpVtbl -> GetTypeInfo(This,iTInfo,lcid,ppTInfo)

#define ISna_GetIDsOfNames(This,riid,rgszNames,cNames,lcid,rgDispId)\
    (This)->lpVtbl -> GetIDsOfNames(This,riid,rgszNames,cNames,lcid,rgDispId)

#define ISna_Invoke(This,dispIdMember,riid,lcid,wFlags,pDispParams,pVarResult,pExcepInfo,pu
ArgErr)\
    (This)->lpVtbl -> Invoke(This,dispIdMember,riid,lcid,wFlags,pDispParams,pVarResult,pExc
epInfo,puArgErr)

#endif /* COBJMACROS */

#endif /* C style interface */

#endif /* __ISna_DISPINTERFACE_DEFINED__ */

EXTERN_C const CLSID CLSID_CBar;

#ifdef __cplusplus

class DECLSPEC_UUID("159A9BBB-E5F1-33F6-BEF5-6CFAD7A5933F")
CBar;
#endif
#endif /* __odod_LIBRARY_DEFINED__ */

/* Additional Prototypes for ALL interfaces */

/* end of Additional Prototypes */

#ifdef __cplusplus
}
```

```
#endif
#endif
```

In the following program, without implements, IBar1, IBar2, and ISna will not be in the coclass in the generated IDL.

```
// attr_implements.cpp
// compile with: /LD /link /idlout:out.idl
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include "attr_implements.h"   // IDL generated header that contains a definition of the ps
eudo-interface IBar and pseudo-dispinterface  ISna

[module(name = "MyLib")];

[dispinterface, uuid("00000000-0000-0000-0000-000000000001")]
__interface IMyInterface
{
   [id(0)] long x;
   [id(1)] HRESULT func();
};

[
   coclass,
   uuid("00000000-0000-0000-0000-000000000002"),
   implements(interfaces={IBar1,IBar2}, dispinterfaces=ISna)
]
class CMyClass : public IMyInterface,
   public IBar1,
      public IDispatchImpl<IBar2, &__uuidof(IBar2)>,
   public ISna
{
   long _x;
public:
   long get_x() { return _x; }
   void put_x(long x0) { _x = x0; }
   HRESULT func() { return S_OK; }
   HRESULT __stdcall bar1() { return S_OK; }
   HRESULT __stdcall bar2() { return S_OK; }
   HRESULT __stdcall sna() { return S_OK; }

   virtual HRESULT STDMETHODCALLTYPE ISna::Invoke(
            /* [in] */ DISPID dispIdMember,
            /* [in] */ REFIID riid,
            /* [in] */ LCID lcid,
            /* [in] */ WORD wFlags,
            /* [out][in] */ DISPPARAMS *pDispParams,
            /* [out] */ VARIANT *pVarResult,
            /* [out] */ EXCEPINFO *pExcepInfo,
            /* [out] */ UINT *puArgErr)
   {
      HRESULT hr = S_OK;
      if (pDispParams == 0) {
         return DISP_E_BADVARTYPE;
      }
      if (pDispParams->cArgs > 0) {
         return DISP_E_BADPARAMCOUNT;
      }
      if (pVarResult != 0) {
         ::VariantInit(pVarResult);
      }
      switch (dispIdMember) {
      case 1:
         {
            if (pDispParams->cArgs != 0) {
               return DISP_E_BADPARAMCOUNT;
            }
```

```
            hr = this->sna();
            break;
         }
      default:
         return DISP_E_MEMBERNOTFOUND;
      }
      return hr;
   }
   virtual HRESULT STDMETHODCALLTYPE ISna::GetIDsOfNames(
            /* [in] */ REFIID riid,
            /* [size_is][in] */ LPOLESTR *rgszNames,
            /* [in] */ UINT cNames,
            /* [in] */ LCID lcid,
            /* [size_is][out] */ DISPID *rgDispId)
   {
      static LPOLESTR names[] = { L"sna" };
      static DISPID dids[] = { 1 };
      for (unsigned int i = 0; i < cNames; ++i) {
         int fFoundIt = 0;
         for (unsigned int j = 0; j < sizeof(names)/sizeof(LPOLESTR); ++j) {
            if (lstrcmpiW(rgszNames[i], names[j]) == 0) {
               fFoundIt = 1;
               rgDispId[i] = dids[j];
               break;
            }
         }
         if (fFoundIt == 0) {
            return DISP_E_UNKNOWNNAME;
         }
      }
      return S_OK;
   }
   virtual HRESULT STDMETHODCALLTYPE ISna::GetTypeInfoCount(unsigned int*  pctinfo)
   {
      if (pctinfo == NULL) {
         return E_POINTER;
      }
      CComPtr<ITypeInfo> spTypeInfo;
      *pctinfo =
                  (SUCCEEDED(TypeInfoHelper(__uuidof(ISna), 0, &spTypeInfo))) ? 1 : 0;
      return S_OK;
   }
   virtual HRESULT STDMETHODCALLTYPE ISna::GetTypeInfo(unsigned int iTInfo, LCID lcid, ITyp
eInfo** ppTInfo)
   {
      if (iTInfo != 0) {
         return DISP_E_BADINDEX;
      }
      return TypeInfoHelper(__uuidof(ISna), lcid, ppTInfo);
   }
   BEGIN_COM_MAP(CMyClass)
      COM_INTERFACE_ENTRY(IBar1)
      COM_INTERFACE_ENTRY(IBar2)
      COM_INTERFACE_ENTRY(ISna)
   END_COM_MAP()
};
```

See Also

Other Resources

Compiler Attributes

Class Attributes

ATL Samples

implements_category

Specifies the component categories implemented by the target class.

```
[ implements_category(
     implements_category="uuid"
) ]
```

Parameters

implements_category

The ID of the implemented category.

Remarks

The **implements_category** C++ attribute specifies the component categories implemented by the target class. This is done by creating a CATEGORY map and adding separate entries specified by the **implements_category** attribute. For more information, see What are Component Categories and How Do They Work?.

This attribute requires that the coclass, progid, or vi_progid attribute (or another attribute that implies one of these) also be applied to the same element. If any single attribute is used, the other two are automatically applied. For example, if **progid** is applied, **vi_progid** and **coclass** are also applied.

Example

The following code specifies that the following object implements the Control category.

```
// cpp_attr_ref_implements_category.cpp
// compile with: /LD
#define _ATL_ATTRIBUTES
#include "atlbase.h"
#include "atlcom.h"

[module (name="MyLib")];
[ coclass, implements_category("CATID_Control"),
  uuid("20a0d0cc-5172-40f5-99ae-5e032f3205ae")]
class CMyClass {};
```

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	Yes
Required attributes	One of the following: **coclass**, **progid**, or **vi_progid**
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

IMPLEMENTED_CATEGORY

Other Resources

COM Attributes

Class Attributes

ATL Samples

import

Specifies another .idl, .odl, or header file containing definitions you want to reference from your main IDL.

```
[ import(
      idl_file
) ];
```

Parameters

idl_file

The name of an .idl file that you want imported into the type library of the current project.

Remarks

The **import** C++ attribute causes an `#import` statement to be placed below the `import "docobj.idl"` statement in the generated .idl file. The **import** attribute has the same functionality as the import MIDL attribute.

The **import** attribute only places the specified file into the .idl file that will be generated by your project; the **import** attribute does not let you call constructs in the specified file from source code in your project. To call constructs in the specified file from source code in your project, either use #import and the **embedded_idl** attribute or you can include the .h file for the *idl_file*, if a .h file exists.

Example

The following code:

```
// cpp_attr_ref_import.cpp
// compile with: /LD
[module(name="MyLib")];
[import(import.idl)];
```

produces the following code in the generated .idl file:

```
import "docobj.idl";
import "import.idl";

[ uuid(EED3644C-8488-3ECD-BA97-147DB3CDB499), version(1.0) ]
library MyLib {
   importlib("stdole2.tlb");
   importlib("olepro32.dll");
...
```

Requirements

Attribute Context

Applies to	Anywhere
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

importidl
importlib
include (C++)

includelib (C++)

Other Resources

IDL Attributes

Stand-Alone Attributes

ATL Samples

importidl

Inserts the specified .idl file into the generated .idl file.

```
[ importidl(
      idl_file
) ];
```

Parameters

idl_file

Identifies the name of the .idl file that you want to merge with the .idl file that will be generated for your application.

Remarks

The **importidl** C++ attribute places the section outside of the library block (in *idl_file*) into your program's generated .idl file and the library section (in *idl_file*) into the library section of your program's generated .idl file.

You may want to use **importidl**, for example, if you want to use a hand-coded .idl file with your generated .idl file.

Example

```
// cpp_attr_ref_importidl.cpp
// compile with: /LD
[module(name="MyLib")];
[importidl("import.idl")];
```

Requirements

Attribute Context

Applies to	Anywhere
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

import
importlib
include (C++)
includelib (C++)

Other Resources

Compiler Attributes
Stand-Alone Attributes
ATL Samples

importlib

Makes types that have already been compiled into another type library available to the type library being created.

```
[ importlib(
   "tlb_file"
) ];
```

Parameters

tlb_file

The name of a .tlb file, in quotes, that you want imported into the type library of the current project.

Remarks

The **importlib** C++ attribute causes an `importlib` statement to be placed in the library block of the generated .idl file. The **importlib** attribute has the same functionality as the importlib MIDL attribute.

Example

The following code shows an example of how to use **importlib**:

```
// cpp_attr_ref_importlib.cpp
// compile with: /LD
[module(name="MyLib")];
[importlib("importlib.tlb")];
```

Requirements

Attribute Context

Applies to	Anywhere
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

import
importidl
include (C++)
includelib (C++)

Other Resources

Compiler Attributes
Stand-Alone Attributes
ATL Samples

in (C++)

Indicates that a parameter is to be passed from the calling procedure to the called procedure.

```
[in]
```

Remarks

The **in** C++ attribute has the same functionality as the in MIDL attribute.

Example

See bindable for an example of how to use **in**.

Requirements

Attribute Context

Applies to	Interface parameter, interface method
Repeatable	No
Required attributes	None
Invalid attributes	**retval**

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

defaultvalue
id
out (C++)

Other Resources

IDL Attributes
Parameter Attributes
Method Attributes
ATL Samples

include (C++)

Specifies one or more header files to be included in the generated .idl file.

```
[ include(
      header_file
) ];
```

Parameters

header_file

The name of a file that you want included in the generated .idl file.

Remarks

The **include** C++ attribute causes an `#include` statement to be placed below the `import "docobj.idl"` statement in the generated .idl file.

The **include** C++ attribute has the same functionality as the include MIDL attribute.

Example

The following code shows an example of how to use **include**. For this example, the file include.h contains only a #include statement.

```
// cpp_attr_ref_include.cpp
// compile with: /LD
[module(name="MyLib")];
[include(cpp_attr_ref_include.h)];
```

Requirements

Attribute Context

Applies to	Anywhere
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

import
importidl
includelib (C++)
importlib

Other Resources

IDL Attributes
Stand-Alone Attributes
ATL Samples

includelib (C++)

Causes an .idl or .h file to be included in the generated .idl file.

```
[ includelib(
      name.idl
) ];
```

Parameters

name.idl

The name of the .idl file that you want included as part of the generated .idl file.

Remarks

The **includelib** C++ attribute causes an .idl or .h file to be included in the generated .idl file, after the `importlib` statement.

Example

The following code is shown in a .cpp file:

```
// cpp_attr_ref_includelib.cpp
// compile with: /LD
[module(name="MyLib")];
[includelib("includelib.idl")];
```

Requirements

Attribute Context

Applies to	Anywhere
Repeatable	Yes
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

import
importidl
include (C++)
importlib

Other Resources

IDL Attributes
Stand-Alone Attributes
ATL Samples

last_is

Specifies the index of the last array element to be transmitted.

```
[ last_is(
   "expression"
) ]
```

Parameters

expression

One or more C-language expressions. Empty argument slots are allowed.

Remarks

The **last_is** C++ attribute has the same functionality as the last_is MIDL attribute.

Example

See first_is for an example of how to specify a section of an array.

Requirements

Attribute Context

Applies to	Field in **struct** or **union**, interface parameter, interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

first_is
max_is
length_is
size_is

Other Resources

IDL Attributes
Typedef, Enum, Union, and Struct Attributes
Parameter Attributes
ATL Samples

lcid

Lets you pass a locale identifier to a function.

```
[lcid]
```

Remarks

The **lcid** C++ attribute implements the functionality of the lcid MIDL attribute. If you want to implement locale for a library block, use the **lcid=***lcid* parameter to the module attribute.

Example

```
// cpp_attr_ref_lcid.cpp
// compile with: /LD
#include <unknwn.h>
[module(name="MyLibrary")];
typedef long HRESULT;

[dual, uuid("2F5F63F1-16DA-11d2-9E7B-00C04FB926DA")]
__interface IStatic {
   HRESULT MyFunc([in, lcid] long LocaleID, [out, retval] BSTR * ReturnVal);
};
```

Requirements

Attribute Context

Applies to	Interface parameter
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Parameter Attributes

ATL Samples

length_is

Specifies the number of array elements to be transmitted.

```
[ length_is(
   "expression"
) ]
```

Parameters

expression

One or more C-language expressions. Empty argument slots are allowed.

Remarks

The **length_is** C++ attribute has the same functionality as the length_is MIDL attribute.

Example

See first_is for an example of how to specify a section of an array.

Requirements

Attribute Context

Applies to	Field in **struct** or **union**, interface parameter, interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

first_is
max_is
last_is
size_is

Other Resources

IDL Attributes
Typedef, Enum, Union, and Struct Attributes
Parameter Attributes
ATL Samples

library_block

Places a construct inside the IDL library block.

```
[library_block]
```

Remarks

When you place a construct inside the library block, you ensure that it will be passed into the type library, regardless of whether it is referenced. By default, only constructs modified by the coclass, dispinterface, and idl_module attributes are placed in the library block.

Example

In the following code, a custom interface is placed inside the library block.

```
// cpp_attr_ref_library_block.cpp
// compile with: /LD
#include <windows.h>
[module(name="MyLib")];
[object, library_block, uuid("9E66A290-4365-11D2-A997-00C04FA37DDB")]
__interface IMyInterface {
   HRESULT f1();
};
```

Requirements

Attribute Context

Applies to	Anywhere
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Other Resources

Compiler Attributes
Stand-Alone Attributes
ATL Samples

licensed

Indicates that the COM object to which it applies is licensed, and must be instantiated using **IClassFactory2**.

```
[licensed]
```

Remarks

The **licensed** C++ attribute has the same functionality as the licensed MIDL attribute.

Example

```
// cpp_attr_ref_licensed.cpp
// compile with: /LD
#include "unknwn.h"
[object, uuid("00000000-0000-0000-0000-000000000001")]
__interface IMyI : IUnknown {
   HRESULT f();
};

[coclass, version("2.1"), uuid(12345678-1111-2222-3333-123456789012),
licensed, threading(free), progid(some.name)]
class CSample : public IMyI {
public:
   int nSize;
};

[module(name="MyLibrary", version="1.0", helpstring="My Library Block")];
```

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	No
Required attributes	**coclass**
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Class Attributes

ATL Samples

local (C++)

When used in the interface header, allows you to use the MIDL compiler as a header generator. When used in an individual function, designates a local procedure for which no stubs are generated.

```
[local]
```

Remarks

The **local** C++ attribute has the same functionality as the local MIDL attribute.

Example

See call_as for an example of how to use **local**.

Requirements

Attribute Context

Applies to	**interface**, interface method
Repeatable	No
Required attributes	None
Invalid attributes	**dispinterface**

For more information, see Attribute Contexts.

See Also

Reference

call_as

Other Resources

IDL Attributes

Interface Attributes

Method Attributes

ATL Samples

max_is

Designates the maximum value for a valid array index.

```
[ max_is(
      "expression"
) ]
```

Parameters

expression

One or more C-language expressions. Empty argument slots are allowed.

Remarks

The **max_is** C++ attribute has the same functionality as the max_is MIDL attribute.

Requirements

Attribute Context

Applies to	Field in **struct** or **union**, interface parameter, interface method
Repeatable	No
Required attributes	None
Invalid attributes	**size_is**

For more information, see Attribute Contexts.

Example

See first_is for an example of how to specify a section of an array.

See Also

Reference

first_is
last_is
length_is
size_is

Other Resources

IDL Attributes
Typedef, Enum, Union, and Struct Attributes
Parameter Attributes
ATL Samples

module (C++)

Defines the library block in the .idl file.

```
[ module (
      type=dll,
      name=string,
      version=1.0,
   uuid=uuid,
   lcid=integer,
   control=boolean,
   helpstring=string,
      helpstringdll=string,
      helpfile=string,
      helpcontext=integer,
      helpstringcontext=integer,
      hidden=boolean,
      restricted=boolean,
      custom=string,
   resource_name=string,
) ];
```

Parameters

type (optional)

Can be one of the following:

- **dll** Adds functions and classes that allow the resulting DLL to function as a in-process COM server. This is the default value.
- **exe** Adds functions and classes that allow the resulting executable to function as a out of process COM server.
- **service** Adds functions and classes that allow the resulting executable to function as an NT service.
- **unspecified** Disables injection of ATL code related to the module attribute: the injection of ATL Module class, global instance _AtlModule and entry point functions. Does not disable injection of ATL code due to other attributes in the project.

name (optional)

The name of the library block.

version (optional)

The version number you want to assign to the library block. The default value is 1.0.

uuid

The unique ID for the library. If you omit this parameter, an ID will be automatically generated for the library. You may need to retrieve the *uuid* of your library block, which you can do by using the identifier **__uuidof(***libraryname***)**.

lcid

The localization parameter. See lcid for more information.

control (optional)

Specifies that all coclasses in the library are controls.

helpstring

Specifies the type library.

helpstringdll (optional)

Sets the name of the .dll file to use to perform a document string lookup. See helpstringdll for more information.

helpfile (optional)

The name of the Help file for the type library.

helpcontext (optional)

The Help ID for this type library.

helpstringcontext (optional)

See helpstringcontext for more information.

hidden (optional)

Prevents the entire library from being displayed. This usage is intended for use with controls. Hosts need to create a new type library that wraps the control with extended properties. See the hidden MIDL attribute for more information.

restricted (optional)

Members of the library cannot be called arbitrarily. See the restricted MIDL attribute for more information.

custom (optional)

One or more attributes; this is similar to the custom attribute. The first parameter to *custom* is the GUID of the attribute. For example:

```
[module(custom={guid,1}, custom={guid1,2})]
```

resource_name

The string resource ID of the .rgs file used to register the APP ID of the DLL, executable, or service. When the module is of type service, this argument is also used to obtain the ID of the string containing the service name.

Note
Both the .rgs file and the string containing the service name should contain the same numerical value.

Remarks

Unless you specify the **restricted** parameter to emitidl, **module** is required in any program that uses C++ attributes.

A library block will be created if, in addition to the **module** attribute, source code also uses dispinterface, dual, object, or an attribute that implies coclass.

One library block is allowed in an .idl file. Multiple module entries in source code will be merged, with the most recent parameter values being implemented.

If this attribute is used within a project that uses ATL, the behavior of the attribute changes. In addition to the above behavior, the attribute also inserts a global object (called **_AtlModule**) of the correct type and additional support code. If the attribute is standalone, it inserts a class derived from the correct module type. If the attribute is applied to a class, it adds a base class of the correct module type. The correct type is determined by the value of the *type* parameter:

- *type* = **dll**

  CAtlDllModuleT is used as the base class and the standard DLL entry points required for a COM server. These entry points are DllMain, DllRegisterServer, DllUnRegisterServer, DllCanUnloadNow, and DllGetClassObject.

- *type* = **exe**

  CAtlExeModuleT is used as the base class and the standard executable entry point WinMain.

- *type* = **service**

  CAtlServiceModuleT is used as the base class and the standard executable entry point WinMain.

- *type* = **unspecified**

  Disables injection of ATL code related to the module attribute.

Example

The following code shows how to create a library block in the generated .idl file.

```
// cpp_attr_ref_module1.cpp
// compile with: /LD
[module(name="MyLibrary", version="1.2", helpfile="MyHelpFile")];
```

The following code shows that you can provide your own implementation of a function that would appear in the code that was injected as a result of using **module**. See /Fx for more information on viewing injected code. In order to override one of the functions inserted by the **module** attribute, make a class that will contain your implementation of the function and make the **module** attribute apply to that class.

```
// cpp_attr_ref_module2.cpp
// compile with: /LD /link /OPT:NOREF
#include <atlbase.h>
#include <atlcom.h>
#include <atlwin.h>
#include <atltypes.h>
#include <atlctl.h>
#include <atlhost.h>
#include <atlplus.h>

// no semicolon after attribute block
[module(dll, name="MyLibrary", version="1.2", helpfile="MyHelpFile")]
// module attribute now applies to this class
class CMyClass {
public:
BOOL WINAPI DllMain(DWORD dwReason, LPVOID lpReserved) {
   // add your own code here
   return __super::DllMain(dwReason, lpReserved);
   }
};
```

Requirements

Attribute Context

Applies to	Anywhere
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information, see Attribute Contexts.

See Also

Reference

usesgetlasterror
library
helpcontext
helpstring
helpfile
version (C++)

Other Resources

IDL Attributes
Class Attributes
Stand-Alone Attributes
Typedef, Enum, Union, and Struct Attributes
ATL Samples

ms_union

Controls the network data representation alignment of nonencapsulated unions.

```
[ms_union]
```

Remarks

The **ms_union** C++ attribute has the same functionality as the ms_union MIDL attribute.

Example

The following code shows the placement of **ms_union**:

```
// cpp_attr_ref_ms_union.cpp
// compile with: /LD
#include <unknwn.h>
[object, ms_union, uuid("00000000-0000-0000-0000-000000000001")]
__interface IFireTabCtrl {
   HRESULT DisplayString([in, string] char * p1);
};

[export, switch_type(short)] union _WILLIE_UNION_TYPE  {
   [case(24)]
      float fMays;
   [case(25)]
      double dMcCovey;
   [default]
      int x;
 };

[public] typedef _WILLIE_UNION_TYPE WILLIE_UNION_TYPE;

[module(name="ATLFIRELib")];
```

Requirements

Attribute Context

Applies to	Nonencapsulated unions
Repeatable	No
Required attributes	None
Invalid attributes	**dispinterface**

For more information, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Typedef, Enum, Union, and Struct Attributes

ATL Samples

no_injected_text

Prevents the compiler from injecting code as a result of attribute use.

```
[ no_injected_text(
      boolean
) ];
```

Parameters

boolean(optional)

true if you want no code injected, **false** to allow code to be injected. **true** is the default.

Remarks

The most common use of the **no_injected_text** C++ attribute is by the /Fx compiler option, which inserts the **no_injected_text** attribute into the .mrg file.

Requirements

Attribute Context

Applies to	Anywhere
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

Compiler Attributes

ATL Samples

nonbrowsable

Indicates that an interface member should not be displayed in a property browser.

```
[nonbrowsable]
```

Remarks

The **nonbrowsable** C++ attribute has the same functionality as the nonbrowsable MIDL attribute.

Example

```
// cpp_attr_ref_nonbrowsable.cpp
// compile with: /LD
#include <unknwn.h>
[module(name="MyLib")];

[object, helpstring("help string"), helpstringcontext(1),
uuid="11111111-1111-1111-1111-111111111111"]
__interface IMyI
{
   [nonbrowsable] HRESULT xx();
};
```

Requirements

Attribute Context

Applies to	Interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Method Attributes

ATL Samples

noncreatable

Defines an object that cannot be instantiated by itself.

```
[noncreatable]
```

Remarks

The **noncreatable** C++ attribute has the same functionality as the noncreatable MIDL attribute and is automatically passed through to the generated .IDL file by the compiler.

When this attribute is used within a project that uses ATL, the behavior of the attribute changes. In addition to the above behavior, the attribute also injects the OBJECT_ENTRY_NON_CREATEABLE_EX_AUTO macro. This macro indicates to ATL that the object cannot be created externally.

Example

```
// cpp_attr_ref_noncreatable.cpp
// compile with: /LD
#include <unknwn.h>
[module(name="MyLib")];

[object, uuid("11111111-1111-1111-1111-111111111111")]
__interface A
{
};

[coclass, uuid("11111111-1111-1111-1111-111111111112"), noncreatable]
class CMyClass : public A
{
   HRESULT xx();
};
```

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	No
Required attributes	**coclass**
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes
Class Attributes
ATL Samples

nonextensible

Specifies that the **IDispatch** implementation includes only the properties and methods listed in the interface description and cannot be extended with additional members at run time.

```
[nonextensible]
```

Remarks

The **nonextensible** C++ attribute has the same functionality as the nonextensible MIDL attribute.

Use of **nonextensible** also requires the oleautomation attribute.

Example

The following code shows one use of the **nonextensible** attribute:

```
// cpp_attr_ref_nonextensible.cpp
// compile with: /LD
#include "unknwn.h"
[module(name="ATLFIRELib")];
[export] typedef long HRESULT;

[dual, nonextensible, ms_union, oleautomation,
uuid("00000000-0000-0000-0000-000000000001")]
__interface IFireTabCtrl
{
   HRESULT procedure (int i);
};
```

Requirements

Attribute Context

Applies to	**interface**
Repeatable	No
Required attributes	**dual** and **oleautomation**, or **dispinterface**
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Interface Attributes

ATL Samples

object (C++)

Identifies a custom interface.

```
[object]
```

Remarks

When preceding an interface definition, the **object** C++ attribute causes the interface to be placed in the .idl file as a custom interface.

Any interface marked with object must inherit from **IUnknown**. This condition is satisfied if any of the base interfaces inherit from **IUnknown**. If no base interfaces inherit from **IUnknown**, the compiler will cause the interface marked with **object** to derive from **IUnknown**.

Example

See nonbrowsable for an example of how to use **object**.

Requirements

Attribute Context

Applies to	**interface**
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

dual
dispinterface
custom (C++)
__interface

Other Resources

IDL Attributes
Interface Attributes
ATL Samples

odl

Identifies an interface as an Object Description Language (ODL) interface. The MIDL compiler does not require the **odl** attribute; it is recognized only for compatibility with older .odl files.

```
[odl]
```

Remarks

The **odl** C++ attribute has the same functionality as the odl MIDL attribute.

Example

```
// cpp_attr_ref_odl.cpp
// compile with: /LD
#include <unknwn.h>
[module(name="MyLIb")];

[odl, oleautomation, dual, uuid("00000000-0000-0000-0000-000000000001")]
__interface IMyInterface
{
   HRESULT x();
};

[coclass, uuid("00000000-0000-0000-0000-000000000002")]
class cmyClass : public IMyInterface
{
public:
   HRESULT x(){}
};
```

Requirements

Attribute Context

Applies to	**interface**
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Interface Attributes

ATL Samples

oleautomation

Indicates that an interface is compatible with Automation.

```
[oleautomation]
```

Remarks

The **oleautomation** C++ attribute has the same functionality as the oleautomation MIDL attribute.

Example

See the examples for defaultvalue and nonextensible for a sample use of **oleautomation**.

Requirements

Attribute Context

Applies to	**interface**
Repeatable	No
Required attributes	None
Invalid attributes	**dispinterface**

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Interface Attributes

ATL Samples

optional (C++)

Specifies an optional parameter for a member function.

```
[optional]
```

Remarks

The **optional** C++ attribute has the same functionality as the optional MIDL attribute.

Example

The following code shows how **optional** might be used:

```
// cpp_attr_ref_optional.cpp
// compile with: /LD
#include "unknwn.h"
[module(name="ATLFIRELib")];

[dispinterface, uuid("00000000-0000-0000-0000-000000000001")]
__interface IFireTabCtrl : IDispatch
{
   [id(1)] long procedure ([in, optional] VARIANT i);
};
```

Requirements

Attribute Context

Applies to	Interface parameter
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Parameter Attributes

ATL Samples

out (C++)

Identifies pointer parameters that are returned from the called procedure to the calling procedure (from the server to the client).

```
[out]
```

Remarks

The **out** C++ attribute has the same functionality as the out MIDL attribute.

Example

See the example for bindable for a sample use of **out**.

Requirements

Attribute Context

Applies to	Interface parameter
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

defaultvalue

id

Other Resources

IDL Attributes

Parameter Attributes

ATL Samples

perf_counter

Apply this attribute to a data member in a perf_object class to expose it as a performance counter.

```
[ perf_counter(
      namestring,
      helpstring,
      name_res,
      help_res,
      countertype,
      defscale,
      default_counter,
      detail,
      countertype,
      max_counter_size,
      countertype_string
) ]
```

Parameters

namestring

A string containing the name of the performance counter. This name will be visible to users in the Performance Monitoring Console. Alternately, the name can be provided in the form of a resource ID using the *name_res* argument. A performance counter name must be specified using one of these two arguments.

helpstring

A string containing a description of the performance counter. This help string will be visible to users in the Performance Monitoring Console. Alternately, the name can be provided in the form of a resource ID using the *help_res* argument. A help string must be specified using one of these two arguments.

name_res

An integer that specifies the resource ID of the name to for this performance counter. This name will be visible to users in the Performance Monitoring Console. If this argument is provided, the *namestring* argument must be omitted.

help_res

An integer that specifies the resource ID of a description for this performance counter. This help string will be visible to users in the Performance Monitoring Console. If this argument is provided, the *helpstring* argument must be omitted.

countertype

An integer that specifies the counter type. See Performance Data Types for more information. Alternately, the *countertype_string* argument can be used.

defscale

An exponential value that modifies the actual counter value. Default is zero.

default_counter

A Boolean value that specifies that this is the default counter for the object. An object can have only one default counter. Default is **false**.

detail

An integer that specifies the desired detail level of the counter, and is **PERF_DETAIL_NOVICE** by default. Can be any one of these values:

Detail level	Description
PERF_DETAIL_NOVICE	Indicates that this counter may be meaningful to most users. This is the most common counter de tail level.

PERF_DETAIL_ADVANCED	Indicates that this counter is likely to be useful only to advanced users.
PERF_DETAIL_EXPERT	Indicates that this counter is likely to be useful only to the most advanced users.
PERF_DETAIL_WIZARD	Indicates that this counter is not likely to be useful to any users.

max_counter_size

An integer that specifies the maximum amount of space to reserve for the data of a string counter. By default this value is zero, which indicates no maximum value. Numeric counters, those whose counter type includes the flags **PERF_SIZE_DWORD** or **PERF_SIZE_LARGE**, are fixed size and may not specify this parameter.

countertype_string

Specifies the counter type as a string. If this argument is provided, the *countertype* argument must be omitted. This argument can be any one of the following strings:

countertype_string	Performance data types
"counter"	**PERF_COUNTER_COUNTER**
"timer"	**PERF_COUNTER_TIMER**
"bulk_count"	**PERF_COUNTER_BULK_COUNT**
"text"	**PERF_COUNTER_TEXT**
"rawcount"	**PERF_COUNTER_RAWCOUNT**
"value"	**PERF_COUNTER_VALUE**
"rate"	**PERF_COUNTER_RATE**
"fraction"	**PERF_COUNTER_FRACTION**
"base"	**PERF_COUNTER_BASE**
"elapsed"	**PERF_COUNTER_ELAPSED**
"queuelen"	**PERF_COUNTER_QUEUELEN**
"histogram"	**PERF_COUNTER_HISTOGRAM**
"precision"	**PERF_COUNTER_PRECISION**

See Performance Data Types for more information.

Requirements

Attribute Context

Applies to	Data member
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

Requirements

Header	atlperf.h
Project	EXE, DLL
Compiler	/D "_ATL_ATTRIBUTES"

For more information about the meaning of the requirements, see Attribute Requirements.

Remarks

This attribute is applied to a data member that will serve as a performance counter. The class containing the new counter must be defined using the perf_object attribute.

The counter name and help string are required arguments but can be provided in either string or resource form. Both arguments must, however, be provided in the same form. The counter type is also a required argument, and can be specified in either string or integer form.

For the following type of counters, a [perf_counter] entry for Type 1 should always be followed by an entry for a [perf_counter] of Type 2. For example, a PERF_AVERAGE_TIMER entry should be followed by one for PERF_AVERAGE_BASE.

Perf Counter Type 1	**Perf Counter Type 2**
PERF_AVERAGE_BULK	PERF_AVERAGE_BASE
PERF_AVERAGE_TIMER	PERF_AVERAGE_BASE
PERF_RAW_FRACTION	PERF_RAW_BASE
PERF_SAMPLE_COUNTER	PERF_SAMPLE_BASE
PERF_SAMPLE_FRACTION	PERF_SAMPLE_BASE
PERF_COUNTER_MULTI_TIMER	PERF_COUNTER_MULTI_BASE
PERF_COUNTER_MULTI_TIMER_INV	PERF_COUNTER_MULTI_BASE
PERF_100NSEC_MULTI_TIMER	PERF_COUNTER_MULTI_BASE
PERF_100NSEC_MULTI_TIMER_INV	PERF_COUNTER_MULTI_BASE

Example

```
#include <atlperf.h>
[ perf_object( namestring = "MyPerfObject", helpstring = "Object Help") ]
class CMyPerfObject
{
   [ perf_counter(namestring="MyCounter", helpstring="Counter Help",
     countertype_string="rawcount") ]
   ULONG m_nCounter;
};
```

```
#define _ATL_ATTRIBUTES
#include <atlperf.h>
[ perf_object( namestring = "MyPerfObject", helpstring = "Object Help") ]
class CMyPerfObject
{
   [
     perf_counter(namestring = "MyRawFraction",
     helpstring = "Raw Fraction Help",
```

```
      countertype = PERF_RAW_FRACTION)
    ]
    ULONG m_nRF;
    [
       perf_counter(namestring = "MyRawFractionBase",
       helpstring = "Raw Fraction Base Help",
       countertype = PERF_RAW_BASE)
    ]
    ULONG m_nRFBase;
};
```

See the PerfPersist Sample.

See Also

Tasks

PerformanceScribble Sample: Performance Monitoring in an MFC Application

PerformanceCounter Sample: Performance Monitoring in an ATL Server Application

Reference

perf_object

perfmon

Concepts

Performance Monitoring

Other Resources

ATL Server Attributes

COM Attributes

Data Member Attributes

Performance Monitoring Reference

ATL Samples

perf_object

Apply this attribute to a class to define a performance monitor object.

```
[ perf_object(
      name_res,
      help_res,
      namestring,
      helpstring,
      detail,
      no_instances,
      class,
      maxinstnamelen
) ]
```

Parameters

name_res

The name of the performance object, as a resource ID. Not necessary if a name is provided in string form using the *namestring* argument.

help_res

A help string for the performance object, as a resource ID. Not necessary if the help string is provided in string form using the *helpstring* argument.

namestring

The name of the performance object, as a string. Not necessary if a name is provided as a resource ID using the *name_res* argument.

helpstring

A help string for the performance object, as a string. Not necessary if a help string is provided as resource ID using the *help_res* argument.

detail

Specifies the detail level desired for the object. Default is **PERF_DETAIL_NOVICE**. Possible values are:

Detail level	Description
PERF_DETAIL_NOVICE	Indicates that this counter may be meaningful to most users. This is the most common counter de tail level.
PERF_DETAIL_ADVANCED	Indicates that this counter is likely to be useful only to advanced users.
PERF_DETAIL_EXPERT	Indicates that this counter is likely to be useful only to the most advanced users.
PERF_DETAIL_WIZARD	Indicates that this counter is not likely to be useful to any users.

no_instances

Specifies the performance object is instanceless if set to **true**. Default is **false**.

class

Specifies the **CPerfMon**-derived class to which this object applies. By default, the next encountered name is used.

maxinstnamelen

Specifies the maximum length of an instance name. Default is **ATLPERF_DEFAULT_MAXINSTNAMELENGTH**.

Requirements

Header	atlperf.h
Project	EXE, DLL
Compiler	/D "_ATL_ATTRIBUTES"

For more information about the meaning of the requirements, see Attribute Requirements.

Remarks

This attribute is used at the class level to specify a class as a performance monitoring object by adding **CPerfObject** as a base class. See Adding an ATL Performance Monitor Object for instruction on using Visual Studio code wizards to add performance monitoring support to your project.

Example

See the example for the perf_counter attribute and see the PerfPersist Sample.

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Tasks

PerformanceScribble Sample: Performance Monitoring in an MFC Application
PerformanceCounter Sample: Performance Monitoring in an ATL Server Application

Reference

perf_counter
perfmon

Concepts

Performance Monitoring
Performance Monitoring

Other Resources

ATL Server Attributes
COM Attributes
Class Attributes
Performance Monitoring Reference
ATL Samples

perfmon

Apply this attribute to a class to define a performance manager object.

```
[ perfmon(
      name,
   register
) ]
```

Parameters

name

The name to associate with the perfmon object, as a string. This is a required argument.

register

Boolean indicating whether this manager object should register its objects with the Windows performance monitoring mechanisms. Default is **true**.

Requirements

Header	atlperf.h
Project	EXE, DLL
Compiler	/D "_ATL_ATTRIBUTES"

For more information about the meaning of the requirements, see Attribute Requirements.

Remarks

This attribute is used at the class level to classify a class as a performance object manager object by adding **CPerfMon** as a base class. See Adding an ATL Performance Monitor Object for instructions on using Visual Studio code wizards to add performance monitoring support to your project.

Example

```
#include <atlperf.h>
[ perfmon( name = "MyPerformanceManagerObject", register = true) ]
class CMyPerfMon
{
};
```

For another example, see the PerfPersist Sample.

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Tasks

PerformanceScribble Sample: Performance Monitoring in an MFC Application

PerformanceCounter Sample: Performance Monitoring in an ATL Server Application

Reference

perf_object

perf_counter

Concepts

Performance Monitoring

Performance Monitoring

Other Resources

ATL Server Attributes

COM Attributes

Class Attributes

Performance Monitoring Reference

ATL Samples

pointer_default

Specifies the default pointer attribute for all pointers, except top-level pointers that appear in parameter lists.

```
[ pointer_default(
      value
) ]
```

Parameters

value

A value that describes the pointer type: **ptr**, **ref**, or **unique**.

Remarks

The **pointer_default** C++ attribute has the same functionality as the pointer_default MIDL attribute.

Example

See the example for defaultvalue for a sample use of **pointer_default**.

Requirements

Attribute Context

Applies to	**interface**
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Interface Attributes

ATL Samples

pragma

Emits the specified string, without the quote characters, into the generated .idl file.

```
[ pragma(
      pragma_statement
) ];
```

Parameters

pragma_statement

The pragma that you want to go into the generated .idl file.

Remarks

The **pragma** C++ attribute has the same functionality as the pragma MIDL attribute.

Example

```
// cpp_attr_ref_pragma.cpp
// compile with: /LD
#include "unknwn.h"
[module(name="MyLib")];
[pragma(pack(4))];

[dispinterface, uuid("00000000-0000-0000-0000-000000000001")]
__interface A
{
   [id(1)] HRESULT MyMethod ([in, satype("BSTR")] SAFEARRAY **p);
};
```

Requirements

Attribute Context

Applies to	Anywhere
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

pack

Other Resources

IDL Attributes

Stand-Alone Attributes

ATL Samples

progid

Specifies the ProgID for a COM object.

```
[ progid(
      name
) ];
```

Parameters

name

The ProgID representing the object.

ProgIDs present a human-readable version of the class identifier (CLSID) used to identify COM/ActiveX objects.

Remarks

The **progid** C++ attribute lets you specify the ProgID for a COM object. A ProgID has the form *name1.name2.version*. If you do not specify a *version* for a ProgID, the default version is 1. If you do not specify *name1.name2*, the default name is *classname.classname*. If you do not specify **progid** and you do specify **vi_progid**, *name1.name2* are taken from **vi_progid** and the (next sequential number) version is appended.

If an attribute block that uses **progid** does not also use **uuid**, the compiler will check the registry to see if a **uuid** exists for the specified **progid**. If **progid** is not specified, the version (and coclass name, if creating a coclass) will be used to generate a **progid**.

progid implies the **coclass** attribute, that is, if you specify **progid**, it is the same thing as specifying the **coclass** and **progid** attributes.

The **progid** attribute causes a class to be automatically registered under the specified name. The generated .idl file will not display the **progid** value.

When this attribute is used within a project that uses ATL, the behavior of the attribute changes. In addition to the above behavior, the information specified with this attribute is used in the **GetProgID** function, injected by the **coclass** attribute. For more information, see the coclass attribute.

Example

See the example for coclass for a sample use of **progid**.

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

ProgID Key

Other Resources

IDL Attributes

Class Attributes

Typedef, Enum, Union, and Struct Attributes

ATL Samples

propget

Specifies a property accessor function.

```
[propget]
```

Remarks

The **propget** C++ attribute has the same functionality as the propget MIDL attribute.

Example

See the example for bindable for a sample use of **propget**.

Requirements

Attribute Context

Applies to	Method
Repeatable	No
Required attributes	None
Invalid attributes	**propput**, **propputref**

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

propput

propputref

Other Resources

IDL Attributes

Method Attributes

ATL Samples

propput

Specifies a property setting function.

```
[propput]
```

Remarks

The **propput** C++ attribute has the same functionality as the propput MIDL attribute.

Example

See the example for bindable for a sample use of **propput**.

Requirements

Attribute Context

Applies to	Method
Repeatable	No
Required attributes	None
Invalid attributes	**propget**, **propputref**

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

propget

propputref

Other Resources

IDL Attributes

Method Attributes

ATL Samples

propputref

Specifies a property setting function that uses a reference instead of a value.

```
[propputref]
```

Remarks

The **propputref** C++ attribute has the same functionality as the propputref MIDL attribute.

Example

See the example for bindable for a sample use of **propputref**.

Requirements

Attribute Context

Applies to	Method
Repeatable	No
Required attributes	None
Invalid attributes	**propget**, **propput**

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

propget

propput

Other Resources

IDL Attributes

Method Attributes

ATL Samples

ptr

Designates a pointer as a full pointer.

```
[ptr]
```

Remarks

The **ptr** C++ attribute has the same functionality as the ptr MIDL attribute.

Example

See the example for defaultvalue for a sample use of **ptr**.

Requirements

Attribute Context

Applies to	Interface parameter, interface method, **typedef**
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes
Interface Attributes
Method Attributes
Typedef, Enum, Union, and Struct Attributes
ATL Samples

public (C++ Attributes)

Ensures that a typedef will go into the type library even if it is not referenced from within the .idl file.

```
[public]
```

Remarks

The **public** C++ attribute has the same functionality as the public MIDL attribute.

Example

The following code shows how to use the **public** attribute:

```
// cpp_attr_ref_public.cpp
// compile with: /LD
#include "unknwn.h"
[module(name="ATLFIRELib")];
[export, public] typedef long MEMBERID;

[dispinterface, uuid(99999999-9999-9999-9999-000000000000)]
__interface IFireTabCtrl : IDispatch
{
   [id(2)] long procedure ([in, optional] VARIANT i);
};
```

Requirements

Attribute Context

Applies to	**typedef**
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Typedef, Enum, Union, and Struct Attributes

ATL Samples

range (C++)

Specifies a range of allowable values for arguments or fields whose values are set at run time.

```
[ range(
      low,
      high
) ]
```

Parameters

low

The low range value.

high

The high range value.

Remarks

The **range** C++ attribute has the same functionality as the range MIDL attribute.

Example

```
// cpp_attr_ref_range.cpp
// compile with: /LD
#include <unknwn.h>
[module(name="MyLib")];

[object, uuid("9E66A290-4365-11D2-A997-00C04FA37DDB")]
__interface ICustom {
   HRESULT Custom([in] long l, [out, retval] long *pLong);
   HRESULT length_is1([in, range(0, 999)] long f, [in, length_is(f)] char array[10]);
   HRESULT length_is2([in, range(-99, -1)] long f, [in, length_is("f"), size_is(10)] char *
array);
};
```

Requirements

Attribute Context

Applies to	Interface method, interface parameter
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes
Method Attributes
Parameter Attributes
Data Member Attributes
ATL Samples

rdx

Creates a registry key or modifies an existing registry key.

```
[ rdx(
      key,
   valuename=NULL,
   regtype
) ]
```

Parameters

key

The name of the key to be created or opened.

valuename(optional)

Specifies the value field to be set. If a value field with this name does not already exist in the key, it is added.

regtype

The type of registry key being added. Can be one of the following: **text**, **dword**, **binary**, or **CString**.

Remarks

The **rdx** C++ attribute creates or modifies an existing registry key for a COM component. The attribute adds a BEGIN_RDX_MAP macro to the object that implements the target member. **RegistryDataExchange**, a function injected as a result of the BEGIN_RDX_MAP macro, can be used to transfer data between the registry and the data members

This attribute can be used in conjunction with the coclass, progid, or vi_progid attributes or other attributes that implies one of these.

Requirements

Attribute Context

Applies to	**class** or **struct** member
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

Example

The following code adds a registry key called MyValue to the system describing the CMyClass COM component.

```
// cpp_attr_ref_rdx.cpp
// compile with: /LD /link /OPT:NOREF
#define _ATL_ATTRIBUTES
#include "atlbase.h"

[module (name="MyLib")];

class CMyClass {
public:
   CMyClass() {
      strcpy(m_sz, "SomeValue");
   }

   [ rdx(key = "HKCR\\MyApp.MyApp.1", valuename = "MyValue", regtype = "text")]
   char m_sz[256];
```

```
};
```

See Also

Reference

registration_script

Other Resources

COM Attributes

ATL Samples

readonly (C++)

Prohibits assignment to a data member.

```
[readonly]
```

Remarks

The **readonly** C++ attribute has the same functionality as the readonly MIDL attribute.

If you want to prohibit modification of a method parameter, then use the in attribute.

Example

The following code shows a use of the **readonly** attribute:

```
// cpp_attr_ref_readonly.cpp
// compile with: /LD
[idl_quote("midl_pragma warning(disable:2461)")];
#include "unknwn.h"
[module(name="ATLFIRELib")];

[dispinterface, uuid(11111111-1111-1111-1111-111111111111)]
__interface IFireTabCtrl
{
   [readonly, id(1)] int i();
};
```

Requirements

Attribute Context

Applies to	Interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Data Member Attributes

ATL Samples

ref (C++)

Identifies a reference pointer.

```
[ref]
```

Remarks

The **ref** C++ attribute has the same functionality as the ref MIDL attribute.

Example

The following code shows how to use the **ref** attribute:

```
// cpp_attr_ref_ref.cpp
// compile with: /LD
#include <windows.h>
[module(name="ATLFIRELib")];
[dispinterface, uuid("00000000-0000-0000-0000-000000000001")]
__interface IFireTabCtrl
{
   [id(1), unique] char * GetFirstName([in, ref] char * pszFullName );
};
```

Requirements

Attribute Context

Applies to	**typedef**, interface parameter, interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes
Typedef, Enum, Union, and Struct Attributes
Parameter Attributes
ATL Samples

registration_script

Executes the specified custom registration script.

```
[ registration_script(
      script
) ]
```

Parameters

script

The full path to a custom registration script (.rgs) file. A value of **none**, such as `script = "none"`, indicates that the coclass has no registration requirements.

Remarks

The **registration_script** C++ attribute executes the custom registration script specified by **script**. If this attribute is not specified, a standard .rgs file (containing information for registering the component) is used. For more information on .rgs files, see The ATL Registry Component (Registrar).

This attribute requires that the coclass, progid, or vi_progid attribute (or another attribute that implies one of these) also be applied to the same element.

Example

The following code specifies that the component has a registry script called cpp_attr_ref_registration_script.rgs.

```
// cpp_attr_ref_registration_script.cpp
// compile with: /LD
#define _ATL_ATTRIBUTES
#include "atlbase.h"
#include "atlcom.h"

[module (name="REG")];

[object, uuid("d9cd196b-6836-470b-9b9b-5b04b828e5b0")]
__interface IFace {};

// requires "cpp_attr_ref_registration_script.rgs"
// create sample .RGS file "cpp_attr_ref_registration_script.rgs" if it does not exist
[ coclass, registration_script(script="cpp_attr_ref_registration_script.rgs"),
  uuid("50d3ad42-3601-4f26-8cfe-0f1f26f98f67")]
class CMyClass:public IFace {};
```

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	No
Required attributes	One or more of the following: **coclass**, **progid**, or **vi_progid**.
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

rdx

Other Resources

COM Attributes

request_handler

Apply this attribute to a class to expose it as an ATL Server request handler and enable it to handle HTTP requests.

```
[ request_handler(
      name,
   sdl
) ]
```

Parameters

name(optional)

A string specifying the name of this request handler. If not specified, the name of the class is used

sdl(optional)

A string specifying the name of the compiler-generated handler that will return the WSDL for this request handler. This parameter is only valid if the soap_handler attribute is applied to the same class.

Remarks

Use the tag_name attribute on any methods of this class that you want to expose as replacement methods.

The names specified with the *name* and *sdl* parameters can be used in a server response file or a query string to refer to the request handler. For example, the following URL would return the service description for an XML Web service (soap_handler-attributed class):

```
http://servername/virtualpath/dllname.dll?Handler=sdl
```

See soap_handler for more details on the implications of using the *sdl* parameter.

Inheritance of request handlers and chaining of handler maps is not supported.

The **request_handler** attribute will usually add CRequestHandlerT as a base class. If the soap_handler attribute has been applied to this class, CSoapHandler will be added as a base class instead. If the class to which this attribute is applied already derives from IRequestHandler, then no base classes will be added.

Base class added	Situation
CRequestHandlerT	request_handler attribute applied to class.
CSoapHandler	request_handler and soap_handler attributes applied to class.
None	Class already derives from IRequestHandler.

Global code added	Situation
DECLARE_REQUEST_HANDLER("*name*", *ClassName*)	request_handler attribute applied to class.
HANDLER_ENTRY_SDL("*name*", *ClassName*, Gen*Class Name*WSDL)	request_handler and soap_handler attributes applied to class, *sdl* para meter omitted.
HANDLER_ENTRY_SDL("*name*", *ClassName*, *sdl*)	request_handler and soap_handler attributes applied to class, *sdl* para meter specified.

Example

See Attributed Request Handler Code. Also see the following samples:

- DataSetConsumer Sample
- Input Sample

- MantaWeb Sample
- SOAPDataTypes Sample
- SOAPTransport Sample
- SRFSyntax Sample
- WeatherService Sample

Requirements

Attribute Context

Applies to	**coclass**, **class**, **struct**
Repeatable	Yes
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

Attribute Requirements

Header	atlstencil.h
Project	EXE, DLL
Compiler	/D "_ATL_ATTRIBUTES"

For more information about the meaning of the requirements, see Attribute Requirements.

See Also

Tasks

WeatherService Sample: An XML Web Service Created with ATL Server
MantaWeb Sample: Online Personal Information Management

Reference

tag_name
soap_handler

Other Resources

ATL Server Attributes
COM Attributes
Class Attributes
ATL Samples

requestedit

Indicates that the property supports the **OnRequestEdit** notification.

```
[requestedit]
```

Remarks

The **requestedit** C++ attribute has the same functionality as the requestedit MIDL attribute.

Example

See the example for bindable for a sample use of **requestedit**.

Requirements

Attribute Context

Applies to	Interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

defaultbind
displaybind
immediatebind

Other Resources

IDL Attributes
Method Attributes
Data Member Attributes
ATL Samples

requires_category

Specifies the required component categories of the target class.

```
[ requires_category(
      requires_category
  ) ]
```

Parameters

requires_category

The ID of the required category.

Remarks

The **requires_category** C++ attribute specifies the component categories required by the target class. For more information, see REQUIRED_CATEGORY.

This attribute requires that the coclass, progid, or vi_progid attribute (or another attribute that implies one of these) also be applied to the same element.

Example

The following code requires that the object implement the Control category.

```
// cpp_attr_ref_requires_category.cpp
// compile with: /LD
#define _ATL_ATTRIBUTES
#include "atlbase.h"
#include "atlcom.h"

[module (name="MyLibrary")];

[ coclass, requires_category("CATID_Control"),
  uuid("1e1a2436-f3ea-4ff3-80bf-5409370e8144")]
class CMyClass {};
```

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	No
Required attributes	One or more of the following: **coclass**, **progid**, or **vi_progid**.
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

implements_category

Other Resources

COM Attributes

ATL Samples

restricted

Specifies that a member of a module, interface, or dispinterface cannot be called arbitrarily.

```
[ restricted(
      interfaces
) ]
```

Parameters

interfaces

One or more interfaces that may not be called arbitrarily on a COM object. This parameter is only valid when applied to a class.

Remarks

The **restricted** C++ attribute has the same functionality as the restricted MIDL attribute.

Example

The following code shows how to use the **restricted** attribute:

```
// cpp_attr_ref_restricted.cpp
// compile with: /LD
#include "windows.h"
#include "unknwn.h"
[module(name="MyLib")];

[object, uuid("00000000-0000-0000-0000-000000000001")]
__interface a
{
};

[object, uuid("00000000-0000-0000-0000-000000000002")]
__interface b
{
};

[coclass, restricted(a,b), uuid("00000000-0000-0000-0000-000000000003")]
class c : public a, public b
{
};
```

Requirements

Attribute Context

Applies to	Interface method, **interface**, **class**, **struct**
Repeatable	No
Required attributes	**coclass** (when applied to **class** or **struct**)
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes
Interface Attributes
Method Attributes
ATL Samples

retval

Designates the parameter that receives the return value of the member.

```
[retval]
```

Remarks

The **retval** C++ attribute has the same functionality as the retval MIDL attribute.

retval must appear on the last argument in a function's declaration.

Example

See the example for bindable for a sample use of **retval**.

Requirements

Attribute Context

Applies to	Interface parameter, interface method
Repeatable	No
Required attributes	**out**
Invalid attributes	**in**

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes
Parameter Attributes
Method Attributes
ATL Samples

satype

Specifies the data type of the **SAFEARRAY** structure.

```
[ satype(
      data_type
) ]
```

Parameters

data_type

The data type for the **SAFEARRAY** data structure that is being passed as a parameter to an interface method.

Requirements

Attribute Context

Applies to	Interface parameter, interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

Remarks

The **satype** C++ attribute specifies the data type of the **SAFEARRAY**.

Note
A level of indirection is dropped from the **SAFEARRAY** pointer in the generated .idl file from how it is declared in the .cpp file .

Example

```
// cpp_attr_ref_satype.cpp
// compile with: /LD
#include "unknwn.h"
[module(name="MyModule")];
[dispinterface, uuid("00000000-0000-0000-0000-000000000001")]
__interface A {
   [id(1)] HRESULT MyMethod ([in, satype("BSTR")] SAFEARRAY **p);
};
```

See Also

Reference

SAFEARRAY

id

Other Resources

Compiler Attributes

Parameter Attributes

Method Attributes

ATL Samples

size_is

Specify the size of memory allocated for sized pointers, sized pointers to sized pointers, and single- or multidimensional arrays.

```
[ size_is(
   "expression"
) ]
```

Parameters

expression

The size of memory allocated for sized pointers.

Remarks

The **size_is** C++ attribute has the same functionality as the size_is MIDL attribute.

Example

See the example for first_is for a sample of how to specify a section of an array.

Requirements

Attribute Context

Applies to	Field in **struct** or **union**, interface parameter, interface method
Repeatable	No
Required attributes	None
Invalid attributes	**max_is**

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

first_is
last_is
max_is
length_is

Other Resources

IDL Attributes
Typedef, Enum, Union, and Struct Attributes
Parameter Attributes
ATL Samples

soap_handler

Apply this attribute to a class to provide the methods necessary for handling SOAP method calls and exposing information about the services offered by this class through WSDL.

```
[ soap_handler(
      name,
      namespace,
   protocol,
   style,
   use
) ]
```

Parameters

name(optional)

A string specifying the name of the XML Web service. If not specified, **"Service"** is appended to the name of the class and used as the name in the WSDL description of the service.

namespace(optional)

A string specifying the XML namespace to be used to uniquely identify the service, methods, and data types. If not specified, the name of the class is used.

protocol(optional)

A string specifying the protocol by which this XML Web service can be accessed. The only supported value is **"soap"**, which is assumed if not specified.

style(optional)

A string specifying the style of operations provided by the XML Web service. Corresponds to the **wsdl:style** XML attribute. The only supported values are **"document"** and **"rpc"**. If no value is specified, **"rpc"** is used.

use(optional)

A string specifying whether the WSDL message parts should be encoded or define a concrete schema. Corresponds to the **wsdl:use** XML attribute. The only supported values are **"literal"** and **"encoded"**. If no value is specified, **"encoded"** is used.

Note that *style* and *use* are only supported in the following combinations:

```
style = "rpc", use = "encoded"
style = "document", use = "literal"
```

Remarks

To create a fully functional XML Web service, you must apply the request_handler attribute to the same class. **soap_handler** provides the functionality for parsing XML SOAP requests and dispatching the calls to the appropriate methods. **request_handler** provides the functionality to expose the XML Web service as an ATL Server request handler that can handle SOAP method calls through HTTP.

Use the soap_method attribute on any methods of this class that you want to expose as methods of the XML Web service. Use the soap_header attribute on SOAP methods where you want to obtain or attach the value of one or more SOAP headers.

The values specified by the parameters of the **soap_handler** attribute will be used in the generated WSDL description of the XML Web service.

To expose the XML Web service description using a request handler, specify the name of the handler that the compiler should generate for this task using the *sdl* parameter of the request_handler attribute.

Note that the compiler-generated handler that returns the WSDL relies on an instance of the main SOAP handler class to provide it with the information necessary to create its response. When generating its response, it will create an instance of your class on the stack in order to get the information about the SOAP methods and SOAP headers, so it's important that your class does not do any heavy initialization in the constructor or assume that it will only be created when being used to handle SOAP requests.

The response generated by the WSDL handler will be cached in release builds, so the SOAP handler will only be created by the WSDL handler the first time that the XML Web service description is requested.

See request_handler for details of the base classes injected by this attribute.

Example

See the following examples:

- SOAP Server Code
- MantaWeb Sample
- DataSetConsumer Sample
- OnlineAddressBook Sample
- SOAPDataTypes Sample
- SOAPTransport Sample
- WeatherService Sample

Requirements

Attribute Context

Applies to	**coclass**, **class**, **struct**
Repeatable	Yes
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

Attribute Requirements

Header	atlsoap.h
Project	EXE, DLL
Compiler	/D "_ATL_ATTRIBUTES"

For more information about the meaning of the requirements, see Attribute Requirements.

See Also

Tasks

WeatherService Sample: An XML Web Service Created with ATL Server
MantaWeb Sample: Online Personal Information Management

Reference

soap_handler
soap_method
request_handler

Other Resources

ATL Server Attributes
COM Attributes
Class Attributes
ATL Samples

soap_header

Apply this attribute to a SOAP method in an XML Web service to specify the data member used to hold the value of a SOAP header.

```
[ soap_header(
      value,
   required,
   in,
   out
) ]
```

Parameters

value

A string specifying the name of the data member representing the SOAP header.

required(optional)

A Boolean value specifying whether the SOAP header is required. If omitted, default is **false**.

in(optional)

A Boolean value specifying whether the SOAP header should be sent to the server from the client. If omitted, default is **true**.

out(optional)

A Boolean value specifying whether the SOAP header should be sent to the client from the server. If omitted, default is **true**.

Remarks

If *in* is **true**, this attribute injects code to parse the SOAP request for the appropriate header. If the SOAP header is present when the method is called, the value of the corresponding data member will be set to the value of the header.

If *out* is **true**, the attribute will ensure that the value of the data member will be packaged as a SOAP header in the response and sent back to the client when the method returns.

In addition, the attribute adds the information about the specified header to the WSDL generated for the XML Web service.

The type of the data member to which this attribute can be applied is limited to the types supported by ATL Server for SOAP. See ATL Server Web Service Supported Types for the list of allowed C++ data types and their corresponding W3C schema data types.

Memory for SOAP method parameters and members used as SOAP headers must be allocated and freed using the memory manager returned by CSoapRootHandler::GetMemMgr (except for BSTRs, which must be allocated using SysAllocString and related functions). Apart from using **CSoapRootHandler::GetMemMgr** instead of the COM task allocator, COM memory allocation rules apply.

Note that the contents of the data member specified using this attribute are only valid during the method call. If data was put in the data member by the ATL Server code as a result of the call, it will be freed when the method returns.

See soap_handler for details of the effects of this attribute.

Example

See SOAP Server Code and see the SOAPState Sample.

Requirements

Attribute Context

Applies to	Method, coclass method
Repeatable	No
Required attributes	None

Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

Attribute Requirements

Header	atlsoap.h
Project	EXE, DLL
Compiler	/D "_ATL_ATTRIBUTES"

For more information about the meaning of the requirements, see Attribute Requirements.

See Also

Tasks

WeatherService Sample: An XML Web Service Created with ATL Server

Reference

soap_handler
soap_method
request_handler

Other Resources

ATL Server Attributes
COM Attributes
Data Member Attributes
ATL Samples

soap_method

Apply this attribute to a method in an XML Web service to expose the specified member as a SOAP method with a corresponding WSDL description.

```
[ soap_method(
      name
) ]
```

Parameters

name(optional)

A string specifying the WSDL name of the SOAP method corresponding to this member. If omitted, the name is taken from the member to which this attribute has been applied.

Remarks

This attribute injects code to parse the SOAP request. If the request is a call on this method, the injected code will unpack the parameters from the XML request, convert to the C++ parameter types, call the corresponding C++ method, pack the return types into an XML response, and return it to the client.

In addition, this attribute adds the information about the specified method to the WSDL generated for the XML Web service.

The method must have marshaling information provided for its parameters by previously declaring the method as a member of an interface. See ATL Server Web Service Supported Types for the list of supported marshaling attributes.

The types of the parameters and return value on the method to which this attribute can be applied are limited to the types supported by ATL Server for SOAP. See ATL Server Web Service Supported Types for the list of allowed C++ data types and their corresponding W3C schema data types.

Memory for SOAP method parameters and members used as SOAP headers must be allocated and freed using the memory manager returned by CSoapRootHandler::GetMemMgr (except for BSTRs which must be allocated using SysAllocString and related functions). Apart from using **CSoapRootHandler::GetMemMgr** instead of the COM task allocator, COM memory allocation rules apply.

Parameters that are also size_is parameters for in/out arrays are not marshaled (and do not appear in the WSDL). This is an issue when you want to use the same variable for two purposes, that is, sending and receiving a value. For example:

```
HRESULT Meth([in,out] int *var);
```

However if `var` were also a **size_is** parameter, it would not work in a SOAP method. In the following method, `var` is not marshaled at all:

```
HRESULT Meth([in,out] int *var, [out, size_is(*var)] BSTR **bstrArray);
```

In this case, the generated client side method would look like:

```
HRESULT Meth(BSTR** bstrArray, int* bstrArray_nSizeIs);
```

The first parameter, `bstrArray`, is ignored (because it does not appear in the WSDL).

The workaround is to declare an additional parameter that is not a **size_is** parameter, so that it will appear in the WSDL and be marshaled. In the following method, the additional parameter is `OneMoreVar`:

```
HRESULT Meth( [in,out] int * OneMoreVar, [out] int *var,
               [out, size_is(*var)] BSTR **bstrArray );
```

Example

See SOAP Server Code and see the following samples:

- DataSetConsumer Sample
- MantaWeb Sample
- OnlineAddressBook Sample
- SOAPDataTypes Sample
- SOAPState Sample
- WeatherService Sample

Requirements

Attribute Context

Applies to	Method, coclass method
Repeatable	No
Required attributes	None.
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

Attribute Requirements

Header	atlsoap.h
Project	EXE, DLL
Compiler	/D "_ATL_ATTRIBUTES"

For more information about the meaning of the requirements, see Attribute Requirements.

See Also

Tasks

WeatherService Sample: An XML Web Service Created with ATL Server
MantaWeb Sample: Online Personal Information Management

Reference

soap_handler
soap_method
soap_header
request_handler

Other Resources

ATL Server Attributes
COM Attributes
Method Attributes
ATL Samples

source (C++)

On a class, specifies the COM object's source interfaces for connection points. On a property or method, indicates that the member returns an object or VARIANT that is a source of events.

```
[ source(
      interfaces
) ]
```

Parameters

interfaces

One or more interfaces that you specify when you apply the source attribute to a class. This parameter is not used when source is applied to a property or method.

Remarks

The **source** C++ attribute has the same functionality as the source MIDL attribute.

You can use the default attribute to specify the default source interface for an object.

Example

```
// cpp_attr_ref_source.cpp
// compile with: /LD
#include "windows.h"
#include "unknwn.h"
[module(name="MyLib")];

[object, uuid(11111111-1111-1111-1111-111111111111)]
__interface b
{
   [id(0), propget, bindable, displaybind, defaultbind, requestedit]
   HRESULT get_I([out, retval]long *i);
};

[object, uuid(11111111-1111-1111-1111-111111111131)]
__interface c
{
   [id(0), propget, bindable, displaybind, defaultbind, requestedit]
   HRESULT et_I([out, retval]long *i);
};

[coclass, default(c), uuid(11111111-1111-1111-1111-111111111132)]
class N : public b
{
};

[coclass, source(c), default(b, c), uuid(11111111-1111-1111-1111-111111111133)]
class NN : public b
{
};
```

Requirements

Attribute Context

Applies to	**class**, **struct**, **interface**
Repeatable	No
Required attributes	**coclass** (when applied to class or struct)

Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

coclass

Other Resources

IDL Attributes

Class Attributes

Method Attributes

ATL Samples

string (C++)

Indicates that the one-dimensional **char**, **wchar_t**, **byte** (or equivalent) array or the pointer to such an array must be treated as a string.

```
[string]
```

Remarks

The **string** C++ attribute has the same functionality as the string MIDL attribute.

Example

The following code shows how to use **string** on an interface and on a typedef:

```
// cpp_attr_ref_string.cpp
// compile with: /LD
#include "unknwn.h"
[module(name="ATLFIRELib")];
[export, string] typedef char a[21];
[dispinterface, restricted, uuid("00000000-0000-0000-0000-000000000001")]
__interface IFireTabCtrl
{
   [id(1)] HRESULT Method3([in, string] char *pC);
};
```

Requirements

Attribute Context

Applies to	Array or pointer to an array, interface parameter, interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

export

Other Resources

IDL Attributes

Array Attributes

ATL Samples

support_error_info

Implements support for returning detailed errors.

```
[ support_error_info(
   error_interface=uuid
) ]
```

Parameters

error_interface

The identifier of the interface implementing **IErrorInfo**.

Remarks

The **support_error_info** C++ attribute implements support for returning detailed, contextual errors encountered by the target object to the client. For the object to support errors, the methods of the **IErrorInfo** interface must be implemented by the object. For more information, see Supporting IDispatch and IErrorInfo.

This attribute adds the ISupportErrorInfoImpl class as a base class to the target object. This results in a default implementation of **ISupportErrorInfo** and can be used when a single interface generates errors on an object.

Example

The following code adds default support for the **ISupportErrorInfo** interface to the `CMyClass` object.

```
// cpp_attr_ref_support_error_info.cpp
// compile with: /LD
#define _ATL_ATTRIBUTES
#include "atlbase.h"
#include "atlcom.h"

[module (name="mymod")];
[object, uuid("f0b17d66-dc6e-4662-baaf-76758e09c878")]
__interface IMyErrors
{
};

[ coclass, support_error_info("IMyErrors"),
  uuid("854dd392-bdc7-4781-8667-8757936f2a4f") ]
class CMyClass
{
};
```

Requirements

Attribute Context

Applies to	**class**
Repeatable	Yes
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

COM Attributes

Class Attributes

ATL Samples

switch_is

Specifies the expression or identifier acting as the union discriminant that selects the union member.

```
[switch_is]
```

Remarks

The **switch_is** C++ attribute has the same functionality as the switch_is MIDL attribute.

Example

See the case example for a sample use of **switch_is**.

Requirements

Attribute Context

Applies to	**typedef**
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

switch_type

Other Resources

IDL Attributes

Typedef, Enum, Union, and Struct Attributes

ATL Samples

switch_type

Identifies the type of the variable used as the union discriminant.

```
[switch_type(type}]
```

Parameters

type

The switch type, can be an integer, character, Boolean, or enumeration type.

Remarks

The **switch_type** C++ attribute has the same functionality as the switch_type MIDL attribute.

C++ attributes do not support encapsulated unions. Nonencapsulated unions are supported only in the following form:

```
// cpp_attr_ref_switch_type.cpp
// compile with: /LD
#include <windows.h>
[module(name="MyLibrary")];
[ export ]
struct SizedValue2 {
   [switch_type("char"), switch_is(kind)] union {
      [case(1), string]
         wchar_t* wval;
      [default, string]
         char* val;
   };
   char kind;
};
```

Example

See the case example for a sample use of **switch_type**.

Requirements

Attribute Context

Applies to	**typedef**
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

export

Other Resources

IDL Attributes

Typedef, Enum, Union, and Struct Attributes

ATL Samples

synchronize

Synchronizes access to the target method.

```
[synchronize]
```

Remarks

The **synchronize** C++ attribute implements support for synchronizing the target method of an object. Synchronization allows multiple objects to use a common resource (such as a method of a class) by controlling the access of the target method.

The code inserted by this attribute calls the proper **Lock** method (determined by the threading model) at the beginning of the target method. When the method is exited, **Unlock** is automatically called. For more information on these functions, see CComAutoThreadModule::Lock

This attribute requires that the coclass, progid, or vi_progid attribute (or another attribute that implies one of these) also be applied to the same element. If any single attribute is used, the other two are automatically applied. For example, if **progid** is applied, **vi_progid** and **coclass** are also applied.

Example

The following code provides synchronization for the `UpdateBalance` method of the `CMyClass` object.

```
// cpp_attr_ref_synchronize.cpp
// compile with: /LD
#define _ATL_ATTRIBUTES
#include "atlbase.h"
#include "atlcom.h"

[module(name="SYNC")];

[coclass,
 threading(both),
 vi_progid("MyProject.MyClass"),
 progid("MyProject.MyClass.1"),
 uuid("7a7baa0d-59b8-4576-b754-79d07e1d1cc3")
]
class CMyClass {
   float m_nBalance;

   [synchronize]
   void UpdateBalance(float nAdjust) {
      m_nBalance += nAdjust;
   }
};
```

Requirements

Attribute Context

Applies to	Class method, method
Repeatable	No
Required attributes	One or more of the following: **coclass**, **progid**, or **vi_progid**.
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

COM Attributes

tag_name

Apply this attribute to a method in a request handler to expose it as a replacement method associated with a tag name.

```
[ tag_name(
      name,
   parse_func
) ]
```

Parameters

name

A string specifying the tag name for the replacement handler that will invoke this method.

parse_func (optional)

Specifies the name of the function used to parse arguments passed to the method.

Remarks

This attribute is used to classify a method as a replacement method. A method may have multiple **tag_name** attributes.

If *parse_func* is specified, the parse function should have the following signature:

```
HTTP_CODE parse_func(IAtlMemMgr* pMemoryManager, LPCSTR szArg, Type** ppArg);
```

The replacement method should have a corresponding signature:

```
HTTP_CODE methodFunc(Type* pArg);
```

If *parse_func* is omitted, the method should have a signature in one of the following forms:

```
HTTP_CODE MethodName();
HTTP_CODE MethodName(Type* pArg);
```

Where *Type* can be one of the types listed:

Type	Parsing algorithm
bool	DefaultParseBool
char	DefaultParseString
unsigned char	DefaultParseUChar
short	DefaultParseShort
unsigned short	DefaultParseUShort
int	DefaultParseInt
unsigned int	DefaultParseUInt
__int64	DefaultParseInt64
unsigned __int64	DefaultParseUInt64

double	DefaultParseDouble
float	DefaultParseFloat

See Passing An Argument To A Replacement Method for details.

Use of this attribute corresponds to the REPLACEMENT_METHOD_ENTRY macro when the method does not have a parameter, and it corresponds to REPLACEMENT_METHOD_ENTRY_EX when the method does have a parameter.

Example

See Attributed Request Handler Code and see the following samples:

- Input Sample
- MantaWeb Sample
- SRFSyntax Sample

Requirements

Attribute Context

Applies to	Method
Repeatable	Yes
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

Attribute Requirements

Header	atlstencil.h
Project	EXE, DLL
Compiler	/D "_ATL_ATTRIBUTES"

For more information about the meaning of the requirements, see Attribute Requirements.

See Also

Other Resources

ATL Server Attributes
COM Attributes
Method Attributes
ATL Samples

threading (C++)

Specifies the threading model for a COM object.

```
[ threading(
      model=enumeration
) ]
```

Parameters

model (optional)

One of the following threading models:

- **apartment** (apartment threading)
- **neutral** (.NET Framework components with no user interface)
- **single** (simple threading)
- **free** (free threading)
- **both** (apartment and free threading)

The default value is **apartment**.

Remarks

The **threading** C++ attribute does not appear in the generated .idl file but will be used in the implementation of your COM object.

In ATL projects, If the coclass attribute is also present, the threading model specified by *model* is passed as the template parameter to the CComObjectRootEx class, inserted by the **coclass** attribute.

The **threading** attribute also guards access to an event_source.

Example

See the licensed example for a sample use of **threading**.

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	No
Required attributes	**coclass**
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

Neutral Apartments

Concepts

Multithreading

Other Resources

COM Attributes

Typedef, Enum, Union, and Struct Attributes

Class Attributes

ATL Samples

transmit_as

Instructs the compiler to associate a presented type that client and server applications manipulate, with a transmitted type.

```
[ transmit_as(
      type
) ]
```

Parameters

type

Specifies the data type that is transmitted between client and server.

Remarks

The **transmit_as** C++ attribute has the same functionality as the transmit_as MIDL attribute.

Example

The following code shows a use of the **transmit_as** attribute:

```
// cpp_attr_ref_transmit_as.cpp
// compile with: /LD
#include "windows.h"
[module(name="MyLibrary")];

[export] typedef struct _TREE_NODE_TYPE {
unsigned short data;
struct _TREE_NODE_TYPE * left;
struct _TREE_NODE_TYPE * right;
} TREE_NODE_TYPE;

[export] struct PACKED_NODE {
   unsigned short data;   // same as normal node
   int index;   // array index of parent
};

// A left node recursive built array of
// the nodes in the tree.  Can be unpacked with
// that knowledge
[export] typedef struct _TREE_XMIT_TYPE {
   int count;
   [size_is(count)] PACKED_NODE node[];
} TREE_XMIT_TYPE;

[transmit_as(TREE_XMIT_TYPE)] typedef TREE_NODE_TYPE * TREE_TYPE;
```

Requirements

Attribute Context

Applies to	**typedef**
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

export

Other Resources

IDL Attributes

Typedef, Enum, Union, and Struct Attributes

ATL Samples

uidefault

Indicates that the type information member is the default member for display in the user interface.

```
[uidefault]
```

Remarks

The **uidefault** C++ attribute has the same functionality as the uidefault MIDL attribute.

Example

The following code shows a sample of **uidefault**:

```
// cpp_attr_ref_uidefault.cpp
// compile with: /LD
#include "unknwn.h"
[module(name="MyLib")];

[object, uuid("9E66A290-4365-11D2-A997-00C04FA37DDB")]
__interface ICustom{
   HRESULT Custom([in] long l, [out, retval] long *pLong);
   [uidefault]HRESULT id0([in] long l);
   [uidefault]HRESULT id1([in] long l);

   [uidefault, propget] HRESULT get_y(int *y);
   [uidefault, propput] HRESULT put_y(int y);
};
```

Requirements

Attribute Context

Applies to	Interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Method Attributes

ATL Samples

unique (C++)

Specifies a unique pointer.

```
[unique]
```

Remarks

The **unique** C++ attribute has the same functionality as the unique MIDL attribute.

Example

See the ref example for a sample use of **unique**.

Requirements

Attribute Context

Applies to	**typedef**, **struct**, **union**, interface parameter, interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes
Typedef, Enum, Union, and Struct Attributes
Parameter Attributes
ATL Samples

usesgetlasterror

Tells the caller that if there is an error when calling that function, then the caller can then call **GetLastError** to retrieve the error code.

```
[usesgetlasterror]
```

Remarks

The **usesgetlasterror** C++ attribute has the same functionality as the usesgetlasterror MIDL attribute.

Example

See the idl_module example for a sample of how to use **usesgetlasterror**.

Requirements

Attribute Context

Applies to	**module** attribute
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

ATL Samples

uuid (C++ Attributes)

Specifies the unique ID for a class or interface.

```
[ uuid(
   "uuid"
) ]
```

Parameters

uuid

A 128-bit, unique identifier.

Remarks

If the definition of an interface or class does not specify the **uuid** C++ attribute, then the Visual C++ compiler will provide one.

If you do not specify **uuid**, then the compiler will generate the same GUID for interfaces or classes with the same name in different attribute projects on a machine.

You can use Uuidgen.exe or Guidgen.exe to generate your own unique IDs. (To run either of these tools, click **Start** and click **Run** on the menu. Then enter the name of the required tool.)

When used in a project that does not also use ATL, specifying the **uuid** attribute is the same as specifying the uuid __declspec modifier. To retrieve the **uuid** of a class, you can use __uuidof

Example

See the bindable example for a sample use of **uuid**.

Requirements

Attribute Context

Applies to	**class**, **struct**, **interface**, **union**, **enum**
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

uuid

Other Resources

IDL Attributes

Interface Attributes

Class Attributes

Typedef, Enum, Union, and Struct Attributes

ATL Samples

v1_enum

Directs that the specified enumerated type be transmitted as a 32-bit entity rather than the 16-bit default.

```
[v1_enum]
```

Remarks

The **v1_enum** C++ attribute has the same functionality as the v1_enum MIDL attribute.

Example

The following code shows a use of **v1_enum**:

```
// cpp_attr_ref_v1_enum.cpp
// compile with: /LD
[module(name="MyLibrary")];

[export, v1_enum]
enum eList {
   e1 = 1,
   e2 = 2
};
```

Requirements

Attribute Context

Applies to	Enumerated type
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Typedef, Enum, Union, and Struct Attributes

ATL Samples

vararg

Specifies that the function takes a variable number of arguments.

```
[vararg]
```

Remarks

The **vararg** C++ attribute has the same functionality as the vararg MIDL attribute.

Example

The following code shows a use of **vararg**:

```
// cpp_attr_ref_vararg.cpp
// compile with: /LD
#include "unknwn.h"
#include "oaidl.h"
[module(name="MyLibrary")];

[object, uuid("00000000-0000-0000-0000-000000000001")]
__interface X : public IUnknown
{
   [vararg] HRESULT Button([in, satype(VARIANT)]SAFEARRAY *psa);
};
```

Requirements

Attribute Context

Applies to	Interface method
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Method Attributes

ATL Samples

version (C++)

Identifies a particular version among multiple versions of a class.

```
[ version(
   "version"
) ]
```

Parameters

version

The version number of the coclass. If not specified, 1.0 will be placed in the .idl file.

Remarks

The **version** C++ attribute has the same functionality as the version MIDL attribute and is passed through to the generated .idl file.

Example

See the bindable example for a sample use of **version**.

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	No
Required attributes	**coclass**
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

Compiler Attributes
Class Attributes
ATL Samples

vi_progid

Specifies a version-independent form of the ProgID.

```
[ vi_progid(
      name
) ];
```

Parameters

name

The version-independent ProgID representing the object.

ProgIDs present a human-readable version of the class identifier (CLSID) used to identify COM/ActiveX objects.

Remarks

The **vi_progid** C++ attribute lets you specify a version-independent ProgID for a COM object. A ProgID has the form *name1.name2.version*. A version-independent ProgID does not have a *version*. It is possible to specify both the **progid** and the **vi_progid** attributes on a coclass. If you do not specify **vi_progid**, the version-independent ProgID is the value specified by the progid attribute.

vi_progid implies the **coclass** attribute, that is, if you specify **vi_progid**, it is the same thing as specifying the **coclass** and **vi_progid** attributes.

The **vi_progid** attribute causes a class to be automatically registered under the specified name. The generated .idl file will not display the ProgID value.

In ATL projects, If the coclass attribute is also present, the specified ProgID is used by the **GetVersionIndependentProgID** function (inserted by the **coclass** attribute).

Example

See the coclass example for a sample use of **vi_progid**.

Requirements

Attribute Context

Applies to	**class**, **struct**
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Reference

ProgID Key

Other Resources

IDL Attributes

Typedef, Enum, Union, and Struct Attributes

Class Attributes

ATL Samples

wire_marshal

Specifies a data type that will be used for transmission instead of an application-specific data type.

```
[wire_marshal]
```

Remarks

The **wire_marshal** C++ attribute has the same functionality as the wire_marshal MIDL attribute.

Example

The following code shows a use of **wire_marshal**:

```
// cpp_attr_ref_wire_marshal.cpp
// compile with: /LD
#include "windows.h"
[module(name="MyLibrary")];

[export, public] typedef unsigned long _FOUR_BYTE_DATA;

[export] typedef struct _TWO_X_TWO_BYTE_DATA {
   unsigned short low;
   unsigned short high;
} TWO_X_TWO_BYTE_DATA ;

[export, wire_marshal(TWO_X_TWO_BYTE_DATA)] typedef _FOUR_BYTE_DATA FOUR_BYTE_DATA;
```

Requirements

Attribute Context

Applies to	**typedef**
Repeatable	No
Required attributes	None
Invalid attributes	None

For more information about the attribute contexts, see Attribute Contexts.

See Also

Other Resources

IDL Attributes

Typedef, Enum, Union, and Struct Attributes

ATL Samples

C/C++ Languages

This section of the Visual C++ documentation includes references on C, C++, and the C/C++ preprocessor.

In This Section

C Language Reference

Describes the C programming language as implemented in Microsoft C.

C++ Language Reference

Describes the C++ programming language as implemented in Microsoft C++.

C/C++ Preprocessor Reference

Describes the preprocessor as it is implemented in Microsoft C and C++.

Compiler Intrinsics

Describes intrinsic functions available in Microsoft C and C++ for x86, IPF, and x64 architectures. Also the intrinsics for MMX, Streaming SIMD Extensions (SSE), and Streaming SIMD Extensions 2.

Related Sections

Visual C++

Provides links to different areas of the Visual Studio and Visual C++ documentation set.

Reference

Provides links to topics describing the libraries provided with Visual C++, the Visual C++ Extensibility Object Model, and the Microsoft Macro Assembler (MASM).

Samples

Provides links to sample code showing the capabilities of Visual C++ and the libraries and technologies it supports.

C Language Reference

The *C Language Reference* describes the C programming language as implemented in Microsoft C. The book's organization is based on the ANSI C standard with additional material on the Microsoft extensions to the ANSI C standard.

- Organization of the C Language Reference

For additional reference material on C++ and the preprocessor, see:

- C++ Language Reference
- Preprocessor Reference

Compiler and linker options are documented in the C/C++ Building Reference.

See Also

Other Resources

C/C++ Languages

Visual C++ Libraries Reference

Organization of the C Language Reference

- Elements of C
- Program Structure
- Declarations and Types
- Expressions and Assignments
- Statements
- Functions
- C Language Syntax Summary
- Implementation-Defined Behavior

See Also

Reference

C Language Reference

Scope of this Manual

C is a flexible language that leaves many programming decisions up to you. In keeping with this philosophy, C imposes few restrictions in matters such as type conversion. Although this characteristic of the language can make your programming job easier, you must know the language well to understand how programs will behave. This book provides information on the C language components and the features of the Microsoft implementation. The syntax for the C language is from ANSI X3.159-1989, *American National Standard for Information Systems – Programming Language – C* (hereinafter called the ANSI C standard), although it is not part of the ANSI C standard. C Language Syntax Summary provides the syntax and a description of how to read and use the syntax definitions.

This book does not discuss programming with C++. See C++ Language Reference for information about the C++ language.

See Also

Concepts

Organization of the C Language Reference

ANSI Conformance

Microsoft® C conforms to the standard for the C language as set forth in the 9899:1990 edition of the ANSI C standard.

Microsoft extensions to the ANSI C standard are noted in the text and syntax of this book as well as in the online reference. Because the extensions are not a part of the ANSI C standard, their use may restrict portability of programs between systems. By default, the Microsoft extensions are enabled. To disable the extensions, specify the /Za compiler option. With /Za, all non-ANSI code generates errors or warnings.

See Also

Concepts

Organization of the C Language Reference

Elements of C

This section describes the elements of the C programming language, including the names, numbers, and characters used to construct a C program. The ANSI C syntax labels these components tokens.

This section explains how to define tokens and how the compiler evaluates them.

The following topics are discussed:

- Tokens
- Comments
- Keywords
- Identifiers
- Constants
- String literals
- Punctuation and special characters

The section also includes reference tables for Trigraphs, Limits on Floating-Point Constants, C Integer Limits, and Escape Sequences.

Operators are symbols (both single characters and character combinations) that specify how values are to be manipulated. Each symbol is interpreted as a single unit, called a token. For more information, see Operators.

See Also

Reference

C Language Reference

C Tokens

In a C source program, the basic element recognized by the compiler is the "token." A token is source-program text that the compiler does not break down into component elements.

Syntax
token:

keyword

identifier

constant

string-literal

operator

punctuator

Note
See the introduction to C Language Syntax Summary for an explanation of the ANSI syntax conventions.

The keywords, identifiers, constants, string literals, and operators described in this section are examples of tokens. Punctuation characters such as brackets (**[ ]**), braces (**{ }**), parentheses (**( )**), and commas (**,**) are also tokens.

See Also
Concepts
Elements of C

White-Space Characters

Space, tab, linefeed, carriage-return, formfeed, vertical-tab, and newline characters are called "white-space characters" because they serve the same purpose as the spaces between words and lines on a printed page — they make reading easier. Tokens are delimited (bounded) by white-space characters and by other tokens, such as operators and punctuation. When parsing code, the C compiler ignores white-space characters unless you use them as separators or as components of character constants or string literals. Use white-space characters to make a program more readable. Note that the compiler also treats comments as white space.

See Also

Reference

C Tokens

C Comments

A "comment" is a sequence of characters beginning with a forward slash/asterisk combination (**/***) that is treated as a single white-space character by the compiler and is otherwise ignored. A comment can include any combination of characters from the representable character set, including newline characters, but excluding the "end comment" delimiter (***/**). Comments can occupy more than one line but cannot be nested.

Comments can appear anywhere a white-space character is allowed. Since the compiler treats a comment as a single white-space character, you cannot include comments within tokens. The compiler ignores the characters in the comment.

Use comments to document your code. This example is a comment accepted by the compiler:

```
/* Comments can contain keywords such as
   for and while without generating errors. */
```

Comments can appear on the same line as a code statement:

```
printf( "Hello\n" );  /* Comments can go here */
```

You can choose to precede functions or program modules with a descriptive comment block:

```
/* MATHERR.C illustrates writing an error routine
 * for math functions.
 */
```

Since comments cannot contain nested comments, this example causes an error:

```
/* Comment out this routine for testing

   /* Open file */
    fh = _open( "myfile.c", _O_RDONLY );
    .
    .
    .
 */
```

The error occurs because the compiler recognizes the first `*/`, after the words `Open file`, as the end of the comment. It tries to process the remaining text and produces an error when it finds the `*/` outside a comment.

While you can use comments to render certain lines of code inactive for test purposes, the preprocessor directives **#if** and **#endif** and conditional compilation are a useful alternative for this task. For more information, see Preprocessor Directives in the *Preprocessor Reference*.

Microsoft Specific

The Microsoft compiler also supports single-line comments preceded by two forward slashes (**//**). If you compile with /Za (ANSI standard), these comments generate errors. These comments cannot extend to a second line.

```
// This is a valid comment
```

Comments beginning with two forward slashes (**//**) are terminated by the next newline character that is not preceded by an escape character. In the next example, the newline character is preceded by a backslash (****), creating an "escape sequence." This escape sequence causes the compiler to treat the next line as part of the previous line. (For more information, see Escape Sequences.)

```
// my comment \
    i++;
```

Therefore, the `i++;` statement is commented out.

The default for Microsoft C is that the Microsoft extensions are enabled. Use /Za to disable these extensions.

END Microsoft Specific

See Also
Reference
C Tokens

Evaluation of Tokens

When the compiler interprets tokens, it includes as many characters as possible in a single token before moving on to the next token. Because of this behavior, the compiler may not interpret tokens as you intended if they are not properly separated by white space. Consider the following expression:

```
i+++j
```

In this example, the compiler first makes the longest possible operator (`++`) from the three plus signs, then processes the remaining plus sign as an addition operator (`+`). Thus, the expression is interpreted as `(i++) + (j)`, not `(i) + (++j)`. In this and similar cases, use white space and parentheses to avoid ambiguity and ensure proper expression evaluation.

Microsoft Specific

The C compiler treats a CTRL+Z character as an end-of-file indicator. It ignores any text after CTRL+Z.

END Microsoft Specific

See Also

Reference

C Tokens

C Keywords

"Keywords" are words that have special meaning to the C compiler. In translation phases 7 and 8, an identifier cannot have the same spelling and case as a C keyword. (See a description of translation phases in the *Preprocessor Reference*; for information on identifiers, see Identifiers.) The C language uses the following keywords:

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

You cannot redefine keywords. However, you can specify text to be substituted for keywords before compilation by using C preprocessor directives.

Microsoft Specific

The ANSI C standard allows identifiers with two leading underscores to be reserved for compiler implementations. Therefore, the Microsoft convention is to precede Microsoft-specific keyword names with double underscores. These words cannot be used as identifier names. For a description of the ANSI rules for naming identifiers, including the use of double underscores, see Identifiers.

The following keywords and special identifiers are recognized by the Microsoft C compiler:

__asm	**dllimport**2	**__int8**	**naked**2
__based1	**__except**	**__int16**	**__stdcall**
__cdecl	**__fastcall**	**__int32**	**thread**2
__declspec	**__finally**	**__int64**	**__try**
dllexport2	**__inline**	**__leave**	

1. The **__based** keyword has limited uses for 32-bit target compilations.

2. These are special identifiers when used with **__declspec**; their use in other contexts is not restricted.

Microsoft extensions are enabled by default. To ensure that your programs are fully portable, you can disable Microsoft extensions by specifying the /Za option (compile for ANSI compatibility) during compilation. When you do this, Microsoft-specific keywords are disabled.

When Microsoft extensions are enabled, you can use the keywords listed above in your programs. For ANSI compliance, most of these keywords are prefaced by a double underscore. The four exceptions, **dllexport**, **dllimport**, **naked**, and **thread**, are used only with **__declspec** and therefore do not require a leading double underscore. For backward compatibility, single-underscore versions of the rest of the keywords are supported.

END Microsoft Specific

See Also

Concepts

Elements of C

C Identifiers

"Identifiers" or "symbols" are the names you supply for variables, types, functions, and labels in your program. Identifier names must differ in spelling and case from any keywords. You cannot use keywords (either C or Microsoft) as identifiers; they are reserved for special use. You create an identifier by specifying it in the declaration of a variable, type, or function. In this example, `result` is an identifier for an integer variable, and `main` and `printf` are identifier names for functions.

```
#include <stdio.h>

int main()
{
    int result;

    if ( result != 0 )
        printf_s( "Bad file handle\n" );
}
```

Once declared, you can use the identifier in later program statements to refer to the associated value.

A special kind of identifier, called a statement label, can be used in **goto** statements. (Declarations are described in Declarations and Types Statement labels are described in The goto and Labeled Statements.)

Syntax
identifier:

nondigit

identifier nondigit

identifier digit

nondigit: one of

_ a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

digit: one of

0 1 2 3 4 5 6 7 8 9

The first character of an identifier name must be a *nondigit* (that is, the first character must be an underscore or an uppercase or lowercase letter). ANSI allows six significant characters in an external identifier's name and 31 for names of internal (within a function) identifiers. External identifiers (ones declared at global scope or declared with storage class **extern**) may be subject to additional naming restrictions because these identifiers have to be processed by other software such as linkers.

Microsoft Specific

Although ANSI allows 6 significant characters in external identifier names and 31 for names of internal (within a function) identifiers, the Microsoft C compiler allows 247 characters in an internal or external identifier name. If you aren't concerned with ANSI compatibility, you can modify this default to a smaller or larger number using the /H (restrict length of external names) option.

END Microsoft Specific

The C compiler considers uppercase and lowercase letters to be distinct characters. This feature, called "case sensitivity," enables you to create distinct identifiers that have the same spelling but different cases for one or more of the letters. For example, each of the following identifiers is unique:

```
add
ADD
Add
aDD
```

Microsoft Specific

Do not select names for identifiers that begin with two underscores or with an underscore followed by an uppercase letter. The ANSI C standard allows identifier names that begin with these character combinations to be reserved for compiler use. Identifiers with file-level scope should also not be named with an underscore and a lowercase letter as the first two letters. Identifier names that begin with these characters are also reserved. By convention, Microsoft uses an underscore and an uppercase letter to begin macro names and double underscores for Microsoft-specific keyword names. To avoid any naming conflicts, always select identifier names that do not begin with one or two underscores, or names that begin with an underscore followed by an uppercase letter.

END Microsoft Specific

The following are examples of valid identifiers that conform to either ANSI or Microsoft naming restrictions:

```
j
count
temp1
top_of_page
skip12
LastNum
```

Microsoft Specific

Although identifiers in source files are case sensitive by default, symbols in object files are not. Microsoft C treats identifiers within a compilation unit as case sensitive.

The Microsoft linker is case sensitive. You must specify all identifiers consistently according to case.

The "source character set" is the set of legal characters that can appear in source files. For Microsoft C, the source set is the standard ASCII character set. The source character set and execution character set include the ASCII characters used as escape sequences. See Character Constants for information about the execution character set.

END Microsoft Specific

An identifier has "scope," which is the region of the program in which it is known, and "linkage," which determines whether the same name in another scope refers to the same identifier. These topics are explained in Lifetime, Scope, Visibility, and Linkage.

See Also
Concepts
Elements of C

Multibyte and Wide Characters

A multibyte character is a character composed of sequences of one or more bytes. Each byte sequence represents a single character in the extended character set. Multibyte characters are used in character sets such as Kanji.

Wide characters are multilingual character codes that are always 16 bits wide. The type for character constants is **char**; for wide characters, the type is **wchar_t**. Since wide characters are always a fixed size, using wide characters simplifies programming with international character sets.

The wide-character-string literal `L"hello"` becomes an array of six integers of type **wchar_t**.

```
{L'h', L'e', L'l', L'l', L'o', 0}
```

The Unicode specification is the specification for wide characters. The run-time library routines for translating between multibyte and wide characters include **mbstowcs**, **mbtowc**, **wcstombs**, and **wctomb**.

See Also

Reference

C Identifiers

Trigraphs

The source character set of C source programs is contained within the 7-bit ASCII character set but is a superset of the ISO 646-1983 Invariant Code Set. Trigraph sequences allow C programs to be written using only the ISO (International Standards Organization) Invariant Code Set. Trigraphs are sequences of three characters (introduced by two consecutive question marks) that the compiler replaces with their corresponding punctuation characters. You can use trigraphs in C source files with a character set that does not contain convenient graphic representations for some punctuation characters.

The following table shows the nine trigraph sequences. All occurrences in a source file of the punctuation characters in the first column are replaced with the corresponding character in the second column.

Trigraph Sequences

Trigraph	Punctuation Character
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

A trigraph is always treated as a single source character. The translation of trigraphs takes place in the first translation phase, before the recognition of escape characters in string literals and character constants. Only the nine trigraphs shown in the above table are recognized. All other character sequences are left untranslated.

The character escape sequence, **\?**, prevents the misinterpretation of trigraph-like character sequences. (For information about escape sequences, see Escape Sequences.) For example, if you attempt to print the string `What??!` with this **printf** statement

```
printf( "What??!\n" );
```

the string printed is `What|` because `??!` is a trigraph sequence that is replaced with the `|` character. Write the statement as follows to correctly print the string:

```
printf( "What?\?!\n" );
```

In this **printf** statement, a backslash escape character in front of the second question mark prevents the misinterpretation of `??!` as a trigraph.

See Also

Reference

C Identifiers

C Constants

A "constant" is a number, character, or character string that can be used as a value in a program. Use constants to represent floating-point, integer, enumeration, or character values that cannot be modified.

Syntax

constant:

floating-point-constant

integer-constant

enumeration-constant

character-constant

Constants are characterized by having a value and a type. Floating-point, integer, and character constants are discussed in the next three sections. Enumeration constants are described in Enumeration Declarations.

See Also

Concepts

Elements of C

C Floating-Point Constants

A "floating-point constant" is a decimal number that represents a signed real number. The representation of a signed real number includes an integer portion, a fractional portion, and an exponent. Use floating-point constants to represent floating-point values that cannot be changed.

Syntax

floating-point-constant:

fractional-constant exponent-part $_{\text{opt}}$ *floating-suffix* $_{\text{opt}}$

digit-sequence exponent-part floating-suffix $_{\text{opt}}$

fractional-constant:

digit-sequence $_{\text{opt}}$ **.** *digit-sequence*

digit-sequence **.**

exponent-part:

e *sign* $_{\text{opt}}$ *digit-sequence*

E *sign* $_{\text{opt}}$ *digit-sequence*

sign : one of

+ –

digit-sequence:

digit

digit-sequence digit

floating-suffix : one of

f l F L

You can omit either the digits before the decimal point (the integer portion of the value) or the digits after the decimal point (the fractional portion), but not both. You can leave out the decimal point only if you include an exponent. No white-space characters can separate the digits or characters of the constant.

The following examples illustrate some forms of floating-point constants and expressions:

```
15.75
1.575E1   /* = 15.75   */
1575e-2   /* = 15.75   */
-2.5e-3   /* = -0.0025 */
25E-4     /* =  0.0025 */
```

Floating-point constants are positive unless they are preceded by a minus sign (**–**). In this case, the minus sign is treated as a unary arithmetic negation operator. Floating-point constants have type **float**, **double**, **long**, or **long double**.

A floating-point constant without an **f**, **F**, **l**, or **L** suffix has type **double**. If the letter **f** or **F** is the suffix, the constant has type **float**. If suffixed by the letter **l** or **L**, it has type **long double**. For example:

```
100L  /* Has type long double  */
100F  /* Has type float        */
```

Note that the Microsoft C compiler maps **long double** to type **double**. See Storage of Basic Types for information about type **double**, **float**, and **long**.

You can omit the integer portion of the floating-point constant, as shown in the following examples. The number .75 can be expressed in many ways, including the following:

```
.0075e2
```

```
0.075e1
.075e1
75e-2
```

See Also

Reference

C Constants

Limits on Floating-Point Constants

Microsoft Specific

Limits on the values of floating-point constants are given in the following table. The header file FLOAT.H contains this information.

Limits on Floating-Point Constants

Constant	Meaning	Value
FLT_DIGDBL_DIGLDBL_DIG	Number of digits, *q*, such that a floating-point number with *q* decimal digits can be rounded into a floating-point representation and back without loss of precision.	6 15 15
FLT_EPSILONDBL_EPSILONLDBL_EPSILON	Smallest positive number *x*, such that *x* + 1.0 is not equal to 1.0	1.192092896e–07F 2.2204460492503131e–016 2.2204460492503131e–016
FLT_GUARD		0
FLT_MANT_DIGDBL_MANT_DIGLDBL_MANT_DIG	Number of digits in the radix specified by **FLT_RADIX** in the floating-point significand. The radix is 2; hence these values specify bits.	24 53 53
FLT_MAXDBL_MAXLDBL_MAX	Maximum representable floating-point number.	3.402823466e+38F 1.7976931348623158e+308 1.7976931348623158e+308
FLT_MAX_10_EXPDBL_MAX_10_EXPLDBL_MAX_10_EXP	Maximum integer such that 10 raised to that number is a representable floating-point number.	38 308 308
FLT_MAX_EXPDBL_MAX_EXPLDBL_MAX_EXP	Maximum integer such that **FLT_RADIX** raised to that number is a representable floating-point number.	128 1024 1024
FLT_MINDBL_MINLDBL_MIN	Minimum positive value.	1.175494351e–38F 2.2250738585072014e–308 2.2250738585072014e–308
FLT_MIN_10_EXPDBL_MIN_10_EXPLDBL_MIN_10_EXP	Minimum negative integer such that 10 raised to that number is a representable floating-point number.	–37 –307 –307
FLT_MIN_EXPDBL_MIN_EXPLDBL_MIN_EXP	Minimum negative integer such that **FLT_RADIX** raised to that number is a representable floating-point number.	–125 –1021 –1021
FLT_NORMALIZE		0
FLT_RADIX_DBL_RADIX_LDBL_RADIX	Radix of exponent representation.	2 2 2
FLT_ROUNDS_DBL_ROUNDS_LDBL_ROUNDS	Rounding mode for floating-point addition.	1 (near) 1 (near) 1 (near)

Note that the information in the above table may differ in future implementations.

END Microsoft Specific

See Also

Reference

C Floating-Point Constants

C Integer Constants

An "integer constant" is a decimal (base 10), octal (base 8), or hexadecimal (base 16) number that represents an integral value. Use integer constants to represent integer values that cannot be changed.

Syntax

integer-constant:

decimal-constant integer-suffix $_{opt}$

octal-constant integer-suffix $_{opt}$

hexadecimal-constant integer-suffix $_{opt}$

decimal-constant:

nonzero-digit

decimal-constant digit

octal-constant:

0

octal-constant octal-digit

hexadecimal-constant:

0x *hexadecimal-digit*

0X *hexadecimal-digit*

hexadecimal-constant hexadecimal-digit

nonzero-digit: one of

1 2 3 4 5 6 7 8 9

octal-digit: one of

0 1 2 3 4 5 6 7

hexadecimal-digit: one of

0 1 2 3 4 5 6 7 8 9

a b c d e f

A B C D E F

integer-suffix:

unsigned-suffix long-suffix $_{opt}$

long-suffix unsigned-suffix $_{opt}$

unsigned-suffix: one of

u U

long-suffix: one of

l L

64-bit integer-suffix:

i64

Integer constants are positive unless they are preceded by a minus sign (**–**). The minus sign is interpreted as the unary arithmetic negation operator. (See Unary Arithmetic Operators for information about this operator.)

If an integer constant begins with **0x** or **0X**, it is hexadecimal. If it begins with the digit **0**, it is octal. Otherwise, it is assumed to be decimal.

The following lines are equivalent:

```
0x1C   /* = Hexadecimal representation for decimal 28 */
034    /* = Octal representation for decimal 28 */
```

No white-space characters can separate the digits of an integer constant. These examples show valid decimal, octal, and hexadecimal constants.

```
/* Decimal Constants */
10
132
32179

/* Octal Constants */
012
0204
076663

/* Hexadecimal Constants */
0xa or 0xA
0x84
0x7dB3 or 0X7DB3
```

See Also

Reference

C Constants

Integer Types

Every integer constant is given a type based on its value and the way it is expressed. You can force any integer constant to type **long** by appending the letter **l** or **L** to the end of the constant; you can force it to be type **unsigned** by appending **u** or **U** to the value. The lowercase letter **l** can be confused with the digit 1 and should be avoided. Some forms of **long** integer constants follow:

```
/* Long decimal constants */
10L
79L

/* Long octal constants */
012L
0115L

/* Long hexadecimal constants */
0xaL or 0xAL
0X4fL or 0x4FL

/* Unsigned long decimal constant */
776745UL
778866LU
```

The type you assign to a constant depends on the value the constant represents. A constant's value must be in the range of representable values for its type. A constant's type determines which conversions are performed when the constant is used in an expression or when the minus sign (**–**) is applied. This list summarizes the conversion rules for integer constants.

- The type for a decimal constant without a suffix is either **int**, **long int**, or **unsigned long int**. The first of these three types in which the constant's value can be represented is the type assigned to the constant.
- The type assigned to octal and hexadecimal constants without suffixes is **int**, **unsigned int**, **long int**, or **unsigned long int** depending on the size of the constant.
- The type assigned to constants with a **u** or **U** suffix is **unsigned int** or **unsigned long int** depending on their size.
- The type assigned to constants with an **l** or **L** suffix is **long int** or **unsigned long int** depending on their size.
- The type assigned to constants with a **u** or **U** and an **l** or **L** suffix is **unsigned long int**.

See Also

Reference

C Integer Constants

C Integer Limits

Microsoft Specific

The limits for integer types are listed in the following table. These limits are defined in the standard header file LIMITS.H. Microsoft C also permits the declaration of sized integer variables, which are integral types of size 8-, 16-, or 32-bits. For more information on sized integers, see Sized Integer Types.

Limits on Integer Constants

Constant	Meaning	Value
CHAR_BIT	Number of bits in the smallest variable that is not a bit field.	8
SCHAR_MIN	Minimum value for a variable of type **signed char**.	–128
SCHAR_MAX	Maximum value for a variable of type **signed char**.	127
UCHAR_MAX	Maximum value for a variable of type **unsigned char**.	255 (0xff)
CHAR_MIN	Minimum value for a variable of type **char**.	–128; 0 if /J option used
CHAR_MAX	Maximum value for a variable of type **char**.	127; 255 if /J option used
MB_LEN_MAX	Maximum number of bytes in a multicharacter constant.	5
SHRT_MIN	Minimum value for a variable of type **short**.	–32768
SHRT_MAX	Maximum value for a variable of type **short**.	32767
USHRT_MAX	Maximum value for a variable of type **unsigned short**.	65535 (0xffff)
INT_MIN	Minimum value for a variable of type **int**.	–2147483647 – 1
INT_MAX	Maximum value for a variable of type **int**.	2147483647
UINT_MAX	Maximum value for a variable of type **unsigned int**.	4294967295 (0xffffffff)
LONG_MIN	Minimum value for a variable of type **long**.	–2147483647 – 1
LONG_MAX	Maximum value for a variable of type **long**.	2147483647
ULONG_MAX	Maximum value for a variable of type **unsigned long**.	4294967295 (0xffffffff)

If a value exceeds the largest integer representation, the Microsoft compiler generates an error.

END Microsoft Specific

See Also
Reference
C Integer Constants

C Character Constants

A "character constant" is formed by enclosing a single character from the representable character set within single quotation marks (**'** **'**). Character constants are used to represent characters in the execution character set.

Syntax

character-constant:

' *c-char-sequence* **'**

L' *c-char-sequence* **'**

c-char-sequence:

c-char

c-char-sequence c-char

c-char:

Any member of the source character set except the single quotation mark (**'**), backslash (****), or newline character

escape-sequence

escape-sequence:

simple-escape-sequence

octal-escape-sequence

hexadecimal-escape-sequence

simple-escape-sequence: one of

\a \b \f \n \r \t \v

\' \" \\ \?

octal-escape-sequence:

**** *octal-digit*

**** *octal-digit octal-digit*

**** *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence:

\x *hexadecimal-digit*

hexadecimal-escape-sequence hexadecimal-digit

See Also

Reference

C Constants

Character Types

An integer character constant not preceded by the letter **L** has type **int**. The value of an integer character constant containing a single character is the numerical value of the character interpreted as an integer. For example, the numerical value of the character `a` is 97 in decimal and 61 in hexadecimal.

Syntactically, a "wide-character constant" is a character constant prefixed by the letter **L**. A wide-character constant has type **wchar_t**, an integer type defined in the STDDEF.H header file. For example:

```
char    schar =  'x';   /* A character constant          */
wchar_t wchar = L'x';   /* A wide-character constant for
                            the same character           */
```

Wide-character constants are 16 bits wide and specify members of the extended execution character set. They allow you to express characters in alphabets that are too large to be represented by type **char**. See Multibyte and Wide Characters for more information about wide characters.

See Also

Reference

C Character Constants

Execution Character Set

This content often refers to the "execution character set." The execution character set is not necessarily the same as the source character set used for writing C programs. The execution character set includes all characters in the source character set as well as the null character, newline character, backspace, horizontal tab, vertical tab, carriage return, and escape sequences. The source and execution character sets may differ in other implementations.

See Also

Reference

C Character Constants

Escape Sequences

Character combinations consisting of a backslash (****) followed by a letter or by a combination of digits are called "escape sequences." To represent a newline character, single quotation mark, or certain other characters in a character constant, you must use escape sequences. An escape sequence is regarded as a single character and is therefore valid as a character constant.

Escape sequences are typically used to specify actions such as carriage returns and tab movements on terminals and printers. They are also used to provide literal representations of nonprinting characters and characters that usually have special meanings, such as the double quotation mark ("). The following table lists the ANSI escape sequences and what they represent.

Note that the question mark preceded by a backslash (**\?**) specifies a literal question mark in cases where the character sequence would be misinterpreted as a trigraph. See Trigraphs for more information.

Escape Sequences

Escape Sequence	Represents
\a	Bell (alert)
\b	Backspace
\f	Formfeed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\'	Single quotation mark
\ "	Double quotation mark
****	Backslash
\?	Literal question mark
**** *ooo*	ASCII character in octal notation
\x *hh*	ASCII character in hexadecimal notation
\x *hhhh*	Unicode character in hexadecimal notation if this escape sequence is used in a wide-character constant or a Unicode string literal. For example, `WCHAR f = L'\x4e00'` or `WCHAR b[] = L"The Chinese character for one is \x4e00"`.

Microsoft Specific

If a backslash precedes a character that does not appear in the table, the compiler handles the undefined character as the character itself. For example, `\c` is treated as an `c`.

END Microsoft Specific

Escape sequences allow you to send nongraphic control characters to a display device. For example, the ESC character (**\033**) is often used as the first character of a control command for a terminal or printer. Some escape sequences are device-specific. For instance, the vertical-tab and formfeed escape sequences (**\v** and **\f**) do not affect screen output, but they do perform

appropriate printer operations.

You can also use the backslash (****) as a continuation character. When a newline character (equivalent to pressing the RETURN key) immediately follows the backslash, the compiler ignores the backslash and the newline character and treats the next line as part of the previous line. This is useful primarily for preprocessor definitions longer than a single line. For example:

```
#define assert(exp) \
( (exp) ? (void) 0:_assert( #exp, __FILE__, __LINE__ ) )
```

See Also

Reference

C Character Constants

Octal and Hexadecimal Character Specifications

The sequence *****ooo* means you can specify any character in the ASCII character set as a three-digit octal character code. The numerical value of the octal integer specifies the value of the desired character or wide character.

Similarly, the sequence **\x***hhh* allows you to specify any ASCII character as a hexadecimal character code. For example, you can give the ASCII backspace character as the normal C escape sequence (**\b**), or you can code it as **\010** (octal) or **\x008** (hexadecimal).

You can use only the digits 0 through 7 in an octal escape sequence. Octal escape sequences can never be longer than three digits and are terminated by the first character that is not an octal digit. Although you do not need to use all three digits, you must use at least one. For example, the octal representation is **\10** for the ASCII backspace character and **\101** for the letter A, as given in an ASCII chart.

Similarly, you must use at least one digit for a hexadecimal escape sequence, but you can omit the second and third digits. Therefore you could specify the hexadecimal escape sequence for the backspace character as either **\x8**, **\x08**, or **\x008**.

The value of the octal or hexadecimal escape sequence must be in the range of representable values for type **unsigned char** for a character constant and type **wchar_t** for a wide-character constant. See Multibyte and Wide Characters for information on wide-character constants.

Unlike octal escape constants, the number of hexadecimal digits in an escape sequence is unlimited. A hexadecimal escape sequence terminates at the first character that is not a hexadecimal digit. Because hexadecimal digits include the letters **a** through **f**, care must be exercised to make sure the escape sequence terminates at the intended digit. To avoid confusion, you can place octal or hexadecimal character definitions in a macro definition:

```
#define Bell '\x07'
```

For hexadecimal values, you can break the string to show the correct value clearly:

```
"\xabc"    /* one character  */
"\xab" "c" /* two characters */
```

See Also

Reference

C Character Constants

C String Literals

A "string literal" is a sequence of characters from the source character set enclosed in double quotation marks (**" "**). String literals are used to represent a sequence of characters which, taken together, form a null-terminated string. You must always prefix wide-string literals with the letter **L**.

Syntax
string-literal:

" *s-char-sequence* $_{opt}$ **"**

L" *s-char-sequence* $_{opt}$ **"**

s-char-sequence:

s-char

s-char-sequence s-char

s-char:

any member of the source character set except the double quotation mark ("), backslash (\), or newline character

escape-sequence

The example below is a simple string literal:

```
char *amessage = "This is a string literal.";
```

All escape codes listed in the Escape Sequences table are valid in string literals. To represent a double quotation mark in a string literal, use the escape sequence **\"**. The single quotation mark (') can be represented without an escape sequence. The backslash (****) must be followed with a second backslash (****) when it appears within a string. When a backslash appears at the end of a line, it is always interpreted as a line-continuation character.

See Also
Concepts
Elements of C

Type for String Literals

String literals have type array of **char** (that is, **char[ ]**). (Wide-character strings have type array of **wchar_t** (that is, **wchar_t[ ]**).) This means that a string is an array with elements of type **char**. The number of elements in the array is equal to the number of characters in the string plus one for the terminating null character.

See Also

Reference

C String Literals

Storage of String Literals

The characters of a literal string are stored in order at contiguous memory locations. An escape sequence (such as **** or **\"**) within a string literal counts as a single character. A null character (represented by the **\0** escape sequence) is automatically appended to, and marks the end of, each string literal. (This occurs during translation phase 7.) Note that the compiler may not store two identical strings at two different addresses. /GF forces the compiler to place a single copy of identical strings into the executable file.

Remarks

Microsoft Specific

Strings have static storage duration. See Storage Classes for information about storage duration.

END Microsoft Specific

See Also

Reference

C String Literals

String Literal Concatenation

To form string literals that take up more than one line, you can concatenate the two strings. To do this, type a backslash, then press the RETURN key. The backslash causes the compiler to ignore the following newline character. For example, the string literal

```
"Long strings can be bro\
ken into two or more pieces."
```

is identical to the string

```
"Long strings can be broken into two or more pieces."
```

String concatenation can be used anywhere you might previously have used a backslash followed by a newline character to enter strings longer than one line.

To force a new line within a string literal, enter the newline escape sequence (**\n**) at the point in the string where you want the line broken, as follows:

```
"Enter a number between 1 and 100\nOr press Return"
```

Because strings can start in any column of the source code and long strings can be continued in any column of a succeeding line, you can position strings to enhance source-code readability. In either case, their on-screen representation when output is unaffected. For example:

```
printf_s ( "This is the first half of the string, "
           "this is the second half ") ;
```

As long as each part of the string is enclosed in double quotation marks, the parts are concatenated and output as a single string. This concatenation occurs according to the sequence of events during compilation specified by translation phases.

```
"This is the first half of the string, this is the second half"
```

A string pointer, initialized as two distinct string literals separated only by white space, is stored as a single string (pointers are discussed in Pointer Declarations). When properly referenced, as in the following example, the result is identical to the previous example:

```
char *string = "This is the first half of the string, "
               "this is the second half";

printf_s( "%s" , string ) ;
```

In translation phase 6, the multibyte-character sequences specified by any sequence of adjacent string literals or adjacent wide-string literals are concatenated into a single multibyte-character sequence. Therefore, do not design programs to allow modification of string literals during execution. The ANSI C standard specifies that the result of modifying a string is undefined.

See Also

Reference

C String Literals

Maximum String Length

Microsoft Specific

ANSI compatibility requires a compiler to accept up to 509 characters in a string literal after concatenation. The maximum length of a string literal allowed in Microsoft C is approximately 2,048 bytes. However, if the string literal consists of parts enclosed in double quotation marks, the preprocessor concatenates the parts into a single string, and for each line concatenated, it adds an extra byte to the total number of bytes.

For example, suppose a string consists of 40 lines with 50 characters per line (2,000 characters), and one line with 7 characters, and each line is surrounded by double quotation marks. This adds up to 2,007 bytes plus one byte for the terminating null character, for a total of 2,008 bytes. On concatenation, an extra character is added for each of the first 40 lines. This makes a total of 2,048 bytes. Note, however, that if line continuations (\) are used instead of double quotation marks, the preprocessor does not add an extra character for each line.

While an individual quoted string cannot be longer than 2048 bytes, a string literal of roughly 65535 bytes can be constructed by concatenating strings.

END Microsoft Specific

See Also

Reference

C String Literals

Punctuation and Special Characters

The punctuation and special characters in the C character set have various uses, from organizing program text to defining the tasks that the compiler or the compiled program carries out. They do not specify an operation to be performed. Some punctuation symbols are also operators (see Operators). The compiler determines their use from context.

Syntax

punctuator: one of

```
[] () {} * , : = ; ... #
```

These characters have special meanings in C. Their uses are described throughout this book. The pound sign (**#**) can occur only in preprocessing directives.

See Also

Concepts

Elements of C

Program Structure

This section gives an overview of C programs and program execution. Terms and features important to understanding C programs and components are also introduced. Topics discussed include:

- Source files and source programs
- The main function and program execution
- Parsing command-line arguments
- Lifetime, scope, visibility, and linkage
- Name spaces

Because this section is an overview, the topics discussed contain introductory material only. See the cross-referenced information for more detailed explanations.

See Also

Reference

C Language Reference

Source Files and Source Programs

A source program can be divided into one or more "source files," or "translation units." The input to the compiler is called a "translation unit."

Syntax

translation-unit:

external-declaration

translation-unit external-declaration

external-declaration:

function-definition

declaration

Overview of Declarations gives the syntax for the *declaration* nonterminal, and the *Preprocessor Reference* explains how the translation unit is processed.

Note
See the introduction to C Language Syntax Summary, for an explanation of the ANSI syntax conventions.

The components of a translation unit are external declarations that include function definitions and identifier declarations. These declarations and definitions can be in source files, header files, libraries, and other files the program needs. You must compile each translation unit and link the resulting object files to make a program.

A C "source program" is a collection of directives, pragmas, declarations, definitions, statement blocks, and functions. To be valid components of a Microsoft C program, each must have the syntax described in this book, although they can appear in any order in the program (subject to the rules outlined throughout this book). However, the location of these components in a program does affect how variables and functions can be used in a program. (See Lifetime, Scope, Visibility, and Linkage for more information.)

Source files need not contain executable statements. For example, you may find it useful to place definitions of variables in one source file and then declare references to these variables in other source files that use them. This technique makes the definitions easy to find and update when necessary. For the same reason, constants and macros are often organized into separate files called "include files" or "header files" that can be referenced in source files as required. See the *Preprocessor Reference* for information about macros and include files.

See Also

Concepts

Program Structure

Directives to the Preprocessor

A "directive" instructs the C preprocessor to perform a specific action on the text of the program before compilation. Preprocessor directives are fully described in the *Preprocessor Reference*. This example uses the preprocessor directive **#define**:

```
#define MAX 100
```

This statement tells the compiler to replace each occurrence of `MAX` by `100` before compilation. The C compiler preprocessor directives are:

#define	**#endif**	**#ifdef**	**#line**
#elif	**#error**	**#ifndef**	**#pragma**
#else	**#if**	**#include**	**#undef**

See Also

Concepts

Source Files and Source Programs

C Pragmas

Microsoft Specific

A "pragma" instructs the compiler to perform a particular action at compile time. Pragmas vary from compiler to compiler. For example, you can use the **optimize** pragma to set the optimizations to be performed on your program. The Microsoft C pragmas are:

alloc_text	**data_seg**	**inline_recursion**	**setlocale**
auto_inline	**function**	**intrinsic**	**warning**
check_stack	**hdrstop**	**message**	
code_seg	**include_alias**	**optimize**	
comment	**inline_depth**	**pack**	

See Pragma Directives in the *Preprocessor Reference* for a description of the Microsoft C compiler pragmas.

END Microsoft Specific

See Also
Concepts
Source Files and Source Programs

C Declarations and Definitions

A "declaration" establishes an association between a particular variable, function, or type and its attributes. Overview of Declarations gives the ANSI syntax for the *declaration* nonterminal. A declaration also specifies where and when an identifier can be accessed (the "linkage" of an identifier). See Lifetime, Scope, Visibility, and Linkage for information about linkage.

A "definition" of a variable establishes the same associations as a declaration but also causes storage to be allocated for the variable.

For example, the `main`, `find`, and `count` functions and the `var` and `val` variables are defined in one source file, in this order:

```
int main() {}

int var = 0;
double val[MAXVAL];
char find( fileptr ) {}
int count( double f ) {}
```

The variables `var` and `val` can be used in the `find` and `count` functions; no further declarations are needed. But these names are not visible (cannot be accessed) in `main`.

See Also

Concepts

Source Files and Source Programs

Function Declarations and Definitions

Function prototypes establish the name of the function, its return type, and the type and number of its formal parameters. A function definition includes the function body.

Remarks

Both function and variable declarations can appear inside or outside a function definition. Any declaration within a function definition is said to appear at the "internal" or "local" level. A declaration outside all function definitions is said to appear at the "external," "global," or "file scope" level. Variable definitions, like declarations, can appear at the internal level (within a function definition) or at the external level (outside all function definitions). Function definitions always occur at the external level. Function definitions are discussed further in Function Definitions. Function prototypes are covered in Function Prototypes.

See Also

Concepts

Source Files and Source Programs

Blocks

A sequence of declarations, definitions, and statements enclosed within curly braces (**{ }**) is called a "block." There are two types of blocks in C. The "compound statement," a statement composed of one or more statements (see The Compound Statement), is one type of block. The other, the "function definition," consists of a compound statement (the body of the function) plus the function's associated "header" (the function name, return type, and formal parameters). A block within other blocks is said to be "nested."

Note that while all compound statements are enclosed within curly braces, not everything enclosed within curly braces constitutes a compound statement. For example, although the specifications of array, structure, or enumeration elements can appear within curly braces, they are not compound statements.

See Also

Concepts

Source Files and Source Programs

Example Program

The following C source program consists of two source files. It gives an overview of some of the various declarations and definitions possible in a C program. Later sections in this book describe how to write these declarations, definitions, and initializations, and how to use C keywords such as **static** and **extern**. The **printf** function is declared in the C header file STDIO.H.

The `main` and `max` functions are assumed to be in separate files, and execution of the program begins with the `main` function. No explicit user functions are executed before `main`.

```
/*****************************************************************
                    FILE1.C - main function
*****************************************************************/

#define ONE     1
#define TWO     2
#define THREE   3
#include <stdio.h>

int a = 1;                      // Defining declarations
int b = 2;                      //  of external variables

extern int max( int a, int b );  // Function prototype

int main()                      // Function definition
{                               //  for main function
    int c;                      // Definitions for
    int d;                      //  two uninitialized
                                //  local variables

    extern int u;               // Referencing declaration
                                //  of external variable
                                //  defined elsewhere
    static int v;               // Definition of variable
                                //  with continuous lifetime

    int w = ONE, x = TWO, y = THREE;
    int z = 0;

    z = max( x, y );            // Executable statements
    w = max( z, w );
    printf_s( "%d %d\n", z, w );
    return 0;
}

/*****************************************************************
            FILE2.C - definition of max function
*****************************************************************/

int max( int a, int b )         // Note formal parameters are
                                //  included in function header
{
    if( a > b )
        return( a );
    else
        return( b );
}
```

FILE1.C contains the prototype for the `max` function. This kind of declaration is sometimes called a "forward declaration" because the function is declared before it is used. The definition for the `main` function includes calls to `max`.

The lines beginning with `#define` are preprocessor directives. These directives tell the preprocessor to replace the identifiers `ONE`, `TWO`, and `THREE` with the numbers `1`, `2`, and `3`, respectively, throughout FILE1.C. However, the directives do not apply to FILE2.C, which is compiled separately and then linked with FILE1.C. The line beginning with `#include` tells the compiler to

include the file STDIO.H, which contains the prototype for the **printf** function. Preprocessor directives are explained in the *Preprocessor Reference*.

FILE1.C uses defining declarations to initialize the global variables `a` and `b`. The local variables `c` and `d` are declared but not initialized. Storage is allocated for all these variables. The static and external variables, `u` and `v`, are automatically initialized to 0. Therefore only `a`, `b`, `u`, and `v` contain meaningful values when declared because they are initialized, either explicitly or implicitly. FILE2.C contains the function definition for `max`. This definition satisfies the calls to `max` in FILE1.C.

The lifetime and visibility of identifiers are discussed in Lifetime, Scope, Visibility, and Linkage. For more information on functions, see Functions.

See Also

Concepts

Source Files and Source Programs

The main Function and Program Execution

Every C program has a primary (main) function that must be named **main**. If your code adheres to the Unicode programming model, you can use the wide-character version of **main**, **wmain**. The **main** function serves as the starting point for program execution. It usually controls program execution by directing the calls to other functions in the program. A program usually stops executing at the end of **main**, although it can terminate at other points in the program for a variety of reasons. At times, perhaps when a certain error is detected, you may want to force the termination of a program. To do so, use the **exit** function. See the *Run-Time Library Reference* for information on and an example using the exit function.

```
main( int argc, char *argv[ ], char *envp[ ] )
```

Remarks

Functions within the source program perform one or more specific tasks. The **main** function can call these functions to perform their respective tasks. When **main** calls another function, it passes execution control to the function, so that execution begins at the first statement in the function. A function returns control to **main** when a **return** statement is executed or when the end of the function is reached.

You can declare any function, including **main**, to have parameters. The term "parameter" or "formal parameter" refers to the identifier that receives a value passed to a function. See Parameters for information on passing arguments to parameters. When one function calls another, the called function receives values for its parameters from the calling function. These values are called "arguments." You can declare formal parameters to **main** so that it can receive arguments from the command line using this format:

When you want to pass information to the **main** function, the parameters are traditionally named *argc* and *argv*, although the C compiler does not require these names. The types for *argc* and *argv* are defined by the C language. Traditionally, if a third parameter is passed to **main**, that parameter is named *envp*. Examples later in this section show how to use these three parameters to access command-line arguments. The following sections explain these parameters.

See Using wmain for a description of the wide-character version of **main**.

See Also

Reference

main: Program Startup

Using wmain

Microsoft Specific

In the Unicode programming model, you can define a wide-character version of the **main** function. Use **wmain** instead of **main** if you want to write portable code that adheres to the Unicode programming model.

```
wmain( int argc, wchar_t *argv[ ], wchar_t *envp[ ] )
```

Remarks

You declare formal parameters to **wmain** using a similar format to **main**. You can then pass wide-character arguments and, optionally, a wide-character environment pointer to the program. The *argv* and *envp* parameters to **wmain** are of type **wchar_t***. For example:

If your program uses a **main** function, the multibyte-character environment is created by the run-time library at program startup. A wide-character copy of the environment is created only when needed (for example, by a call to the **_wgetenv** or **_wputenv** functions). On the first call to **_wputenv**, or on the first call to **_wgetenv** if an MBCS environment already exists, a corresponding wide-character string environment is created and is then pointed to by the **_wenviron** global variable, which is a wide-character version of the **_environ** global variable. At this point, two copies of the environment (MBCS and Unicode) exist simultaneously and are maintained by the operating system throughout the life of the program.

Similarly, if your program uses a **wmain** function, a wide-character environment is created at program startup and is pointed to by the **_wenviron** global variable. An MBCS (ASCII) environment is created on the first call to **_putenv** or **getenv**, and is pointed to by the **_environ** global variable.

For more information on the MBCS environment, see Internationalization in the *Run-Time Library Reference*.

END Microsoft Specific

See Also
Reference
The main Function and Program Execution

Argument Description

The *argc* parameter in the **main** and **wmain** functions is an integer specifying how many arguments are passed to the program from the command line. Since the program name is considered an argument, the value of *argc* is at least one.

Remarks

The *argv* parameter is an array of pointers to null-terminated strings representing the program arguments. Each element of the array points to a string representation of an argument passed to **main** (or **wmain**). (For information about arrays, see Array Declarations.) The *argv* parameter can be declared either as an array of pointers to type **char** (`char *argv[]`) or as a pointer to pointers to type **char** (`char **argv`). For **wmain**, the *argv* parameter can be declared either as an array of pointers to type **wchar_t** (`wchar_t *argv[]`) or as a pointer to pointers to type **wchar_t** (`wchar_t **argv`).

By convention, *argv***[0]** is the command with which the program is invoked. However, it is possible to spawn a process using CreateProcess and if you use both the first and second arguments (*lpApplicationName* and *lpCommandLine*), *argv***[0]** may not be the executable name; use GetModuleFileName to retrieve the executable name.

The last pointer (`argv[argc]`) is **NULL**. (See getenv in the *Run-Time Library Reference* for an alternative method for getting environment variable information.)

Microsoft Specific

The *envp* parameter is a pointer to an array of null-terminated strings that represent the values set in the user's environment variables. The *envp* parameter can be declared as an array of pointers to **char** (`char *envp[]`) or as a pointer to pointers to **char** (`char **envp`). In a **wmain** function, the *envp* parameter can be declared as an array of pointers to **wchar_t** (`wchar_t *envp[]`) or as a pointer to pointers to **wchar_t** (`wchar_t **envp`). The end of the array is indicated by a **NULL** *pointer. Note that the environment block passed to **main** or **wmain** is a "frozen" copy of the current environment. If you subsequently change the environment via a call to **_putenv** or **_wputenv**, the current environment (as returned by **getenv**/**_wgetenv** and the **_environ** or **_wenviron** variables) will change, but the block pointed to by *envp* will not change. The *envp* parameter is ANSI compatible in C, but not in C++.

END Microsoft Specific

See Also

Reference

The main Function and Program Execution

Expanding Wildcard Arguments

Microsoft Specific

When running a C program, you can use either of the two wildcards — the question mark (?) and the asterisk (*) — to specify filename and path arguments on the command line.

Command-line arguments are handled by a routine called **_setargv** (or **_wsetargv** in the wide-character environment), which by default does not expand wildcards into separate strings in the *argv* string array. You can replace the normal **_setargv** routine with a more powerful version of **_setargv** that does handle wildcards by linking with the Setargv.obj file. If your program uses a **wmain** function, link with Wsetargv.obj.

To link with Setargv.obj or Wsetargv.obj, use the /link option. For example:

```
cl typeit.c /link setargv.obj
```

The wildcards are expanded in the same manner as operating system commands. (See your operating system user's guide if you are unfamiliar with wildcards.)

END Microsoft Specific

See Also

Reference

The main Function and Program Execution

Concepts

Link Options

Parsing C Command-Line Arguments

Microsoft Specific

Microsoft C startup code uses the following rules when interpreting arguments given on the operating system command line:

- Arguments are delimited by white space, which is either a space or a tab.
- A string surrounded by double quotation marks is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument. Note that the caret (**^**) is not recognized as an escape character or delimiter.
- A double quotation mark preceded by a backslash, **\"**, is interpreted as a literal double quotation mark (**"**).
- Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
- If an even number of backslashes is followed by a double quotation mark, then one backslash (****) is placed in the *argv* array for every pair of backslashes (****), and the double quotation mark (**"**) is interpreted as a string delimiter.
- If an odd number of backslashes is followed by a double quotation mark, then one backslash (****) is placed in the *argv* array for every pair of backslashes (****) and the double quotation mark is interpreted as an escape sequence by the remaining backslash, causing a literal double quotation mark (**"**) to be placed in *argv*.

This list illustrates the rules above by showing the interpreted result passed to `argv` for several examples of command-line arguments. The output listed in the second, third, and fourth columns is from the ARGS.C program that follows the list.

Command-Line Input	argv[1]	argv[2]	argv[3]
`"a b c" d e`	`a b c`	`d`	`e`
`"ab\"c" "\\" d`	`ab"c`	`\`	`d`
`a\\\b d"e f"g h`	`a\\\b`	`de fg`	`h`
`a\\\"b c d`	`a\"b`	`c`	`d`
`a\\\\"b c" d e`	`a\\b c`	`d`	`e`

Example

```
// Parsing_C_Commandline_args.c
// ARGS.C illustrates the following variables used for accessing
// command-line arguments and environment variables:
// argc  argv  envp
//

#include <stdio.h>

int main( int argc, // Number of strings in array argv
 char *argv[],      // Array of command-line argument strings
 char **envp )      // Array of environment variable strings
{
    int count;

    // Display each command-line argument.
    printf_s( "\nCommand-line arguments:\n" );
    for( count = 0; count < argc; count++ )
        printf_s( "  argv[%d]   %s\n", count, argv[count] );

    // Display each environment variable.
    printf_s( "\nEnvironment variables:\n" );
    while( *envp != NULL )
        printf_s( "  %s\n", *(envp++) );

    return;
```

```
}
```

Comments

One example of output from this program is:

```
Command-line arguments:
  argv[0]   C:\MSC\TEST.EXE

Environment variables:
  COMSPEC=C:\NT\SYSTEM32\CMD.EXE

  PATH=c:\nt;c:\binb;c:\binr;c:\nt\system32;c:\word;c:\help;c:\msc;c:\;
  PROMPT=[$p]
  TEMP=c:\tmp
  TMP=c:\tmp
  EDITORS=c:\binr
  WINDIR=c:\nt
```

END Microsoft Specific

See Also

Reference

The main Function and Program Execution

Customizing C Command-Line Processing

If your program does not take command-line arguments, you can save a small amount of space by suppressing use of the library routine that performs command-line processing. This routine is called **_setargv** (or **_wsetargv** in the wide-character environment), as described in Expanding Wildcard Arguments. To suppress its use, define a routine that does nothing in the file containing the **main** function and name it **_setargv** (or **_wsetargv** in the wide-character environment). The call to **_setargv** or **_wsetargv** is then satisfied by your definition of **_setargv** or **_wsetargv** , and the library version is not loaded.

Similarly, if you never access the environment table through the *envp* argument, you can provide your own empty routine to be used in place of **_setenvp** (or **_wsetenvp**), the environment-processing routine.

If your program makes calls to the **_spawn** or **_exec** family of routines in the C run-time library, you should not suppress the environment-processing routine, since this routine is used to pass an environment from the spawning process to the new process.

See Also

Reference

The main Function and Program Execution

Lifetime, Scope, Visibility, and Linkage

To understand how a C program works, you must understand the rules that determine how variables and functions can be used in the program. Several concepts are crucial to understanding these rules:

- Lifetime
- Scope and visibility
- Linkage

See Also

Concepts

Program Structure

Lifetime

"Lifetime" is the period during execution of a program in which a variable or function exists. The storage duration of the identifier determines its lifetime.

An identifier declared with the *storage-class-specifier* **static** has static storage duration. Identifiers with static storage duration (also called "global") have storage and a defined value for the duration of a program. Storage is reserved and the identifier's stored value is initialized only once, before program startup. An identifier declared with external or internal linkage also has static storage duration (see Linkage).

An identifier declared without the **static** storage-class specifier has automatic storage duration if it is declared inside a function. An identifier with automatic storage duration (a "local identifier") has storage and a defined value only within the block where the identifier is defined or declared. An automatic identifier is allocated new storage each time the program enters that block, and it loses its storage (and its value) when the program exits the block. Identifiers declared in a function with no linkage also have automatic storage duration.

The following rules specify whether an identifier has global (static) or local (automatic) lifetime:

- All functions have static lifetime. Therefore they exist at all times during program execution. Identifiers declared at the external level (that is, outside all blocks in the program at the same level of function definitions) always have global (static) lifetimes.
- If a local variable has an initializer, the variable is initialized each time it is created (unless it is declared as **static**). Function parameters also have local lifetime. You can specify global lifetime for an identifier within a block by including the **static** storage-class specifier in its declaration. Once declared **static**, the variable retains its value from one entry of the block to the next.

Although an identifier with a global lifetime exists throughout the execution of the source program (for example, an externally declared variable or a local variable declared with the **static** keyword), it may not be visible in all parts of the program. See Scope and Visibility for information about visibility, and see Storage Classes for a discussion of the *storage-class-specifier* nonterminal.

Memory can be allocated as needed (dynamic) if created through the use of special library routines such as **malloc**. Since dynamic memory allocation uses library routines, it is not considered part of the language. See the malloc function in the *Run-Time Library Reference*.

See Also

Concepts

Lifetime, Scope, Visibility, and Linkage

Scope and Visibility

An identifier's "visibility" determines the portions of the program in which it can be referenced — its "scope." An identifier is visible (i.e., can be used) only in portions of a program encompassed by its "scope," which may be limited (in order of increasing restrictiveness) to the file, function, block, or function prototype in which it appears. The scope of an identifier is the part of the program in which the name can be used. This is sometimes called "lexical scope." There are four kinds of scope: function, file, block, and function prototype.

All identifiers except labels have their scope determined by the level at which the declaration occurs. The following rules for each kind of scope govern the visibility of identifiers within a program:

File scope

The declarator or type specifier for an identifier with file scope appears outside any block or list of parameters and is accessible from any place in the translation unit after its declaration. Identifier names with file scope are often called "global" or "external." The scope of a global identifier begins at the point of its definition or declaration and terminates at the end of the translation unit.

Function scope

A label is the only kind of identifier that has function scope. A label is declared implicitly by its use in a statement. Label names must be unique within a function. (For more information about labels and label names, see The goto and Labeled Statements.)

Block scope

The declarator or type specifier for an identifier with block scope appears inside a block or within the list of formal parameter declarations in a function definition. It is visible only from the point of its declaration or definition to the end of the block containing its declaration or definition. Its scope is limited to that block and to any blocks nested in that block and ends at the curly brace that closes the associated block. Such identifiers are sometimes called "local variables."

Function-prototype scope

The declarator or type specifier for an identifier with function-prototype scope appears within the list of parameter declarations in a function prototype (not part of the function declaration). Its scope terminates at the end of the function declarator.

The appropriate declarations for making variables visible in other source files are described in Storage Classes. However, variables and functions declared at the external level with the **static** storage-class specifier are visible only within the source file in which they are defined. All other functions are globally visible.

See Also

Concepts

Lifetime, Scope, Visibility, and Linkage

Summary of Lifetime and Visibility

The following table is a summary of lifetime and visibility characteristics for most identifiers. The first three columns give the attributes that define lifetime and visibility. An identifier with the attributes given by the first three columns has the lifetime and visibility shown in the fourth and fifth columns. However, the table does not cover all possible cases. Refer to Storage Classes for more information.

Summary of Lifetime and Visibility

Attributes: Level	Item	Storage-Class Specifier	Result: Lifetime	Visibility
File scope	Variable definition	**static**	Global	Remainder of source file in which it occurs
	Variable declaration	**extern**	Global	Remainder of source file in which it occurs
	Function prototype or definition	**static**	Global	Single source file
	Function prototype	**extern**	Global	Remainder of source file
Block scope	Variable declaration	**extern**	Global	Block
	Variable definition	**static**	Global	Block
	Variable definition	**auto** or **register**	Local	Block

Example

The following example illustrates blocks, nesting, and visibility of variables:

```
// Lifetime_and_Visibility.c

#include <stdio.h>

int i = 1;  // i defined at external level

int main()  // main function defined at external level
{
    printf_s( "%d\n", i ); // Prints 1 (value of external level i)
    {                                 // Begin first nested block
        int i = 2, j = 3;          // i and j defined at internal level
        printf_s( "%d %d\n", i, j );  // Prints 2, 3
        {                             // Begin second nested block
            int i = 0;                // i is redefined
            printf_s( "%d %d\n", i, j ); // Prints 0, 3
        }                             // End of second nested block
        printf_s( "%d\n", i );        // Prints 2 (outer definition
                                      //  restored)
    }                                 // End of first nested block
    printf_s( "%d\n", i );            // Prints 1 (external level
                                      // definition restored)
    return 0;
}
```

In this example, there are four levels of visibility: the external level and three block levels. The values are printed to the screen as noted in the comments following each statement.

See Also

Concepts

Lifetime, Scope, Visibility, and Linkage

Linkage

Identifier names can refer to different identifiers in different scopes. An identifier declared in different scopes or in the same scope more than once can be made to refer to the same identifier or function by a process called "linkage." Linkage determines the portions of the program in which an identifier can be referenced (its "visibility"). There are three kinds of linkage: internal, external, and no linkage.

See Also

Reference

Using extern to Specify Linkage

Internal Linkage

If the declaration of a file-scope identifier for an object or a function contains the *storage-class-specifier* **static**, the identifier has internal linkage. Otherwise, the identifier has external linkage. See Storage Classes for a discussion of the *storage-class-specifier* nonterminal.

Within one translation unit, each instance of an identifier with internal linkage denotes the same identifier or function. Internally linked identifiers are unique to a translation unit.

See Also

Reference

Using extern to Specify Linkage

External Linkage

If the first declaration at file-scope level for an identifier does not use the **static** storage-class specifier, the object has external linkage.

If the declaration of an identifier for a function has no *storage-class-specifier*, its linkage is determined exactly as if it were declared with the *storage-class-specifier* **extern**. If the declaration of an identifier for an object has file scope and no *storage-class-specifier*, its linkage is external.

An identifier's name with external linkage designates the same function or data object as does any other declaration for the same name with external linkage. The two declarations can be in the same translation unit or in different translation units. If the object or function also has global lifetime, the object or function is shared by the entire program.

See Also

Reference

Using extern to Specify Linkage

No Linkage

If a declaration for an identifier within a block does not include the **extern** storage-class specifier, the identifier has no linkage and is unique to the function.

The following identifiers have no linkage:

- An identifier declared to be anything other than an object or a function
- An identifier declared to be a function parameter
- A block-scope identifier for an object declared without the **extern** storage-class specifier

If an identifier has no linkage, declaring the same name again (in a declarator or type specifier) in the same scope level generates a symbol redefinition error.

See Also

Reference

Using extern to Specify Linkage

Name Spaces

The compiler sets up "name spaces" to distinguish between the identifiers used for different kinds of items. The names within each name space must be unique to avoid conflict, but an identical name can appear in more than one name space. This means that you can use the same identifier for two or more different items, provided that the items are in different name spaces. The compiler can resolve references based on the syntactic context of the identifier in the program.

Note

Do not confuse the limited C notion of a name space with the C++ "namespace" feature. See Namespaces in the C++ *Language Reference* for more information.

This list describes the name spaces used in C.

Statement labels

Named statement labels are part of statements. Definitions of statement labels are always followed by a colon but are not part of **case** labels. Uses of statement labels always immediately follow the keyword **goto**. Statement labels do not have to be distinct from other names or from label names in other functions.

Structure, union, and enumeration tags

These tags are part of structure, union, and enumeration type specifiers and, if present, always immediately follow the reserved words **struct**, **union**, or **enum**. The tag names must be distinct from all other structure, enumeration, or union tags with the same visibility.

Members of structures or unions

Member names are allocated in name spaces associated with each structure and union type. That is, the same identifier can be a component name in any number of structures or unions at the same time. Definitions of component names always occur within structure or union type specifiers. Uses of component names always immediately follow the member-selection operators (**–>** and **.**). The name of a member must be unique within the structure or union, but it does not have to be distinct from other names in the program, including the names of members of different structures and unions, or the name of the structure itself.

Ordinary identifiers

All other names fall into a name space that includes variables, functions (including formal parameters and local variables), and enumeration constants. Identifier names have nested visibility, so you can redefine them within blocks.

Typedef names

Typedef names cannot be used as identifiers in the same scope.

For example, since structure tags, structure members, and variable names are in three different name spaces, the three items named `student` in this example do not conflict. The context of each item allows correct interpretation of each occurrence of `student` in the program. (For information about structures, see Structure Declarations.)

```
struct student {
   char student[20];
   int class;
   int id;
   } student;
```

When `student` appears after the **struct** keyword, the compiler recognizes it as a structure tag. When `student` appears after a member-selection operator (**–>** or **.**), the name refers to the structure member. In other contexts, `student` refers to the structure variable. However, overloading the tag name space is not recommended since it obscures meaning.

See Also

Concepts

Program Structure

Declarations and Types

This section describes the declaration and initialization of variables, functions, and types. The C language includes a standard set of basic data types. You can also add your own data types, called "derived types," by declaring new ones based on types already defined. The following topics are discussed:

- Overview of declarations
- Storage classes
- Type specifiers
- Type qualifiers
- Declarators and variable declarations
- Interpreting more complex declarators
- Initialization
- Storage of basic types
- Incomplete types
- Typedef declarations
- Extended storage-class attributes

See Also

Reference

C Language Reference

Overview of Declarations

A "declaration" specifies the interpretation and attributes of a set of identifiers. A declaration that also causes storage to be reserved for the object or function named by the identifier is called a "definition." C declarations for variables, functions, and types have this syntax:

Syntax

declaration:

declaration-specifiers init-declarator-list $_{opt}$ **;**

declaration-specifiers:

storage-class-specifier attribute-seq $_{opt}$

declaration-specifiers $_{opt}$

/* *attribute-seq* $_{opt}$ is Microsoft specific */

type-specifier declaration-specifiers $_{opt}$

type-qualifier declaration-specifiers $_{opt}$

init-declarator-list:

init-declarator

init-declarator-list , *init-declarator*

init-declarator:

declarator

declarator **=** *initializer*

Note
This syntax for *declaration* is not repeated in the following sections. Syntax in the following sections usually begins with the *declarator* nonterminal.

The declarations in the *init-declarator-list* contain the identifiers being named; *init* is an abbreviation for initializer. The *init-declarator-list* is a comma-separated sequence of declarators, each of which can have additional type information, or an initializer, or both. The *declarator* contains the identifiers, if any, being declared. The *declaration-specifiers* nonterminal consists of a sequence of type and storage-class specifiers that indicate the linkage, storage duration, and at least part of the type of the entities that the declarators denote. Therefore, declarations are made up of some combination of storage-class specifiers, type specifiers, type qualifiers, declarators, and initializers.

Declarations can contain one or more of the optional attributes listed in *attribute-seq*; *seq* is an abbreviation for sequence. These Microsoft-specific attributes perform a variety of functions, which are discussed in detail throughout this book. For a list of these attributes, see C Language Syntax Summary.

In the general form of a variable declaration, *type-specifier* gives the data type of the variable. The *type-specifier* can be a compound, as when the type is modified by **const** or **volatile**. The *declarator* gives the name of the variable, possibly modified to declare an array or a pointer type. For example,

```
int const *fp;
```

declares a variable named `fp` as a pointer to a nonmodifiable (**const**) **int** value. You can define more than one variable in a declaration by using multiple declarators, separated by commas.

A declaration must have at least one declarator, or its type specifier must declare a structure tag, union tag, or members of an enumeration. Declarators provide any remaining information about an identifier. A declarator is an identifier that can be modified with brackets (**[ ]**), asterisks (*****), or parentheses (**( )**) to declare an array, pointer, or function type, respectively. When you declare simple variables (such as character, integer, and floating-point items), or structures and unions of simple variables, the *declarator* is just an identifier. For more information on declarators, see Declarators and Variable Declarations.

All definitions are implicitly declarations, but not all declarations are definitions. For example, variable declarations that begin with the **extern** storage-class specifier are "referencing," rather than "defining" declarations. If an external variable is to be referred to before it is defined, or if it is defined in another source file from the one where it is used, an **extern** declaration is necessary. Storage is not allocated by "referencing" declarations, nor can variables be initialized in declarations.

A storage class or a type (or both) is required in variable declarations. Except for **__declspec**, only one storage-class specifier is allowed in a declaration and not all storage-class specifiers are permitted in every context. The **__declspec** storage class is allowed with other storage-class specifiers, and it is allowed more than once. The storage-class specifier of a declaration affects how the declared item is stored and initialized, and which parts of a program can reference the item.

The *storage-class-specifier* terminals defined in C include **auto**, **extern**, **register**, **static**, and **typedef**. In addition, Microsoft C includes the *storage-class-specifier* terminal **__declspec**. All *storage-class-specifier* terminals except **typedef** and **__declspec** are discussed in Storage Classes. See Typedef Declarations for information about **typedef**. See Extended Storage-Class Attributes for information about **__declspec**.

The location of the declaration within the source program and the presence or absence of other declarations of the variable are important factors in determining the lifetime of variables. There can be multiple redeclarations but only one definition. However, a definition can appear in more than one translation unit. For objects with internal linkage, this rule applies separately to each translation unit, because internally linked objects are unique to a translation unit. For objects with external linkage, this rule applies to the entire program. See Lifetime, Scope, Visibility, and Linkage for more information about visibility.

Type specifiers provide some information about the data types of identifiers. The default type specifier is **int**. For more information, see Type Specifiers. Type specifiers can also define type tags, structure and union component names, and enumeration constants. For more information see Enumeration Declarations, Structure Declarations, and Union Declarations.

There are two *type-qualifier* terminals: **const** and **volatile**. These qualifiers specify additional properties of types that are relevant only when accessing objects of that type through l-values. For more information on **const** and **volatile**, see Type Qualifiers. For a definition of l-values, see L-Value and R-Value Expressions.

See Also

Concepts

Declarations and Types

C Storage Classes

The "storage class" of a variable determines whether the item has a "global" or "local" lifetime. C calls these two lifetimes "static" and "automatic." An item with a global lifetime exists and has a value throughout the execution of the program. All functions have global lifetimes.

Automatic variables, or variables with local lifetimes, are allocated new storage each time execution control passes to the block in which they are defined. When execution returns, the variables no longer have meaningful values.

C provides the following storage-class specifiers:

Syntax
storage-class-specifier:

auto

register

static

extern

typedef

__declspec (*extended-decl-modifier-seq*) /* Microsoft Specific */

Except for **__declspec**, you can use only one *storage-class-specifier* in the *declaration-specifier* in a declaration. If no storage-class specification is made, declarations within a block create automatic objects.

Items declared with the **auto** or **register** specifier have local lifetimes. Items declared with the **static** or **extern** specifier have global lifetimes.

Since **typedef** and **__declspec** are semantically different from the other four *storage-class-specifier* terminals, they are discussed separately. For specific information on **typedef**, see Typedef Declarations. For specific information on **__declspec**, see Extended Storage-Class Attributes.

The placement of variable and function declarations within source files also affects storage class and visibility. Declarations outside all function definitions are said to appear at the "external level." Declarations within function definitions appear at the "internal level."

The exact meaning of each storage-class specifier depends on two factors:

- Whether the declaration appears at the external or internal level
- Whether the item being declared is a variable or a function

Storage-Class Specifiers for External-Level Declarations and Storage-Class Specifiers for Internal-Level Declarations describe the *storage-class-specifier* terminals in each kind of declaration and explain the default behavior when the *storage-class-specifier* is omitted from a variable. Storage-Class Specifiers with Function Declarations discusses storage-class specifiers used with functions.

See Also
Concepts
Declarations and Types

Storage-Class Specifiers for External-Level Declarations

External variables are variables at file scope. They are defined outside any function, and they are potentially available to many functions. Functions can only be defined at the external level and, therefore, cannot be nested. By default, all references to external variables and functions of the same name are references to the same object, which means they have "external linkage." (You can use the **static** keyword to override this. See information later in this section for more details on **static**.)

Variable declarations at the external level are either definitions of variables ("defining declarations"), or references to variables defined elsewhere ("referencing declarations").

An external variable declaration that also initializes the variable (implicitly or explicitly) is a defining declaration of the variable. A definition at the external level can take several forms:

- A variable that you declare with the **static** storage-class specifier. You can explicitly initialize the **static** variable with a constant expression, as described in Initialization. If you omit the initializer, the variable is initialized to 0 by default. For example, these two statements are both considered definitions of the variable `k`.

  ```
  static int k = 16;
  static int k;
  ```

- A variable that you explicitly initialize at the external level. For example, `int j = 3;` is a definition of the variable `j`.

In variable declarations at the external level (that is, outside all functions), you can use the **static** or **extern** storage-class specifier or omit the storage-class specifier entirely. You cannot use the **auto** and **register** *storage-class-specifier* terminals at the external level.

Once a variable is defined at the external level, it is visible throughout the rest of the translation unit. The variable is not visible prior to its declaration in the same source file. Also, it is not visible in other source files of the program, unless a referencing declaration makes it visible, as described below.

The rules relating to **static** include:

- Variables declared outside all blocks without the **static** keyword always retain their values throughout the program. To restrict their access to a particular translation unit, you must use the **static** keyword. This gives them "internal linkage." To make them global to an entire program, omit the explicit storage class or use the keyword **extern** (see the rules in the next list). This gives them "external linkage." Internal and external linkage are also discussed in Linkage.

- You can define a variable at the external level only once within a program. You can define another variable with the same name and the **static** storage-class specifier in a different translation unit. Since each **static** definition is visible only within its own translation unit, no conflict occurs. This provides a useful way to hide identifier names that must be shared among functions of a single translation unit, but not visible to other translation units.

- The **static** storage-class specifier can apply to functions as well. If you declare a function **static**, its name is invisible outside of the file in which it is declared.

The rules for using **extern** are:

- The **extern** storage-class specifier declares a reference to a variable defined elsewhere. You can use an **extern** declaration to make a definition in another source file visible, or to make a variable visible prior to its definition in the same source file. Once you have declared a reference to the variable at the external level, the variable is visible throughout the remainder of the translation unit in which the declared reference occurs.

- For an **extern** reference to be valid, the variable it refers to must be defined once, and only once, at the external level. This definition (without the **extern** storage class) can be in any of the translation units that make up the program.

Example

The example below illustrates external declarations:

```
/*****************************************************************
                        SOURCE FILE ONE
*****************************************************************/
#include <stdio.h>
```

```
extern int i;                // Reference to i, defined below
void next( void );           // Function prototype

int main()
{
    i++;
    printf_s( "%d\n", i );   // i equals 4
    next();
}

int i = 3;                  // Definition of i

void next( void )
{
    i++;
    printf_s( "%d\n", i );  // i equals 5
    other();
}

/*****************************************************************
                      SOURCE FILE TWO
*****************************************************************/
#include <stdio.h>

extern int i;              // Reference to i in
                           // first source file
void other( void )
{
    i++;
    printf_s( "%d\n", i ); // i equals 6
}
```

The two source files in this example contain a total of three external declarations of `i`. Only one declaration is a "defining declaration." That declaration,

```
int i = 3;
```

defines the global variable `i` and initializes it with initial value 3. The "referencing" declaration of `i` at the top of the first source file using **extern** makes the global variable visible prior to its defining declaration in the file. The referencing declaration of `i` in the second source file also makes the variable visible in that source file. If a defining instance for a variable is not provided in the translation unit, the compiler assumes there is an

```
extern int x;
```

referencing declaration and that a defining reference

```
int x = 0;
```

appears in another translation unit of the program.

All three functions, `main`, `next`, and `other`, perform the same task: they increase `i` and print it. The values 4, 5, and 6 are printed.

If the variable `i` had not been initialized, it would have been set to 0 automatically. In this case, the values 1, 2, and 3 would have been printed. See Initialization for information about variable initialization.

See Also

Reference

C Storage Classes

Storage-Class Specifiers for Internal-Level Declarations

You can use any of four *storage-class-specifier* terminals for variable declarations at the internal level. When you omit the *storage-class-specifier* from such a declaration, the default storage class is **auto**. Therefore, the keyword **auto** is rarely seen in a C program.

See Also

Reference

C Storage Classes

The auto Storage-Class Specifier

The **auto** storage-class specifier declares an automatic variable, a variable with a local lifetime. An **auto** variable is visible only in the block in which it is declared. Declarations of **auto** variables can include initializers, as discussed in Initialization. Since variables with **auto** storage class are not initialized automatically, you should either explicitly initialize them when you declare them, or assign them initial values in statements within the block. The values of uninitialized **auto** variables are undefined. (A local variable of **auto** or **register** storage class is initialized each time it comes in scope if an initializer is given.)

An internal **static** variable (a static variable with local or block scope) can be initialized with the address of any external or **static** item, but not with the address of another **auto** item, because the address of an **auto** item is not a constant.

See Also
Reference
auto Keyword

The register Storage-Class Specifier

Microsoft Specific

The Microsoft C/C++ compiler does not honor user requests for register variables. However, for portability all other semantics associated with the **register** keyword are honored by the compiler. For example, you cannot apply the unary address-of operator (**&**) to a register object nor can the **register** keyword be used on arrays.

END Microsoft Specific

See Also

Concepts

Storage-Class Specifiers for Internal-Level Declarations

The static Storage-Class Specifier

A variable declared at the internal level with the **static** storage-class specifier has a global lifetime but is visible only within the block in which it is declared. For constant strings, using **static** is useful because it alleviates the overhead of frequent initialization in often-called functions.

Remarks

If you do not explicitly initialize a **static** variable, it is initialized to 0 by default. Inside a function, **static** causes storage to be allocated and serves as a definition. Internal static variables provide private, permanent storage visible to only a single function.

See Also

Reference

Static (C++)

The extern Storage-Class Specifier

A variable declared with the **extern** storage-class specifier is a reference to a variable with the same name defined at the external level in any of the source files of the program. The internal **extern** declaration is used to make the external-level variable definition visible within the block. Unless otherwise declared at the external level, a variable declared with the **extern** keyword is visible only in the block in which it is declared.

Example

This example illustrates internal- and external-level declarations:

```
// extern_StorageClassSpecified.c
#include <stdio.h>

void other( void );

int main()
{
    // Reference to i, defined below:
    extern int i;

    // Initial value is zero; a is visible only within main:
    static int a;

    // b is stored in a register, if possible:
    register int b = 0;

    // Default storage class is auto:
    int c = 0;

    // Values printed are 1, 0, 0, 0:
    printf_s( "%d\n%d\n%d\n%d\n", i, a, b, c );
    other();
    return;
}

int i = 1;

void other( void )
{
    // Address of global i assigned to pointer variable:
    static int *external_i = &i;

    // i is redefined; global i no longer visible:
    int i = 16;

    // This a is visible only within the other function:
    static int a = 2;

    a += 2;
    // Values printed are 16, 4, and 1:
    printf_s( "%d\n%d\n%d\n", i, a, *external_i );
}
```

In this example, the variable `i` is defined at the external level with initial value 1. An **extern** declaration in the `main` function is used to declare a reference to the external-level `i`. The **static** variable `a` is initialized to 0 by default, since the initializer is omitted. The call to `printf` prints the values 1, 0, 0, and 0.

In the `other` function, the address of the global variable `i` is used to initialize the **static** pointer variable `external_i`. This works because the global variable has **static** lifetime, meaning its address does not change during program execution. Next, the variable `i` is redefined as a local variable with initial value 16. This redefinition does not affect the value of the external-level `i`, which is hidden by the use of its name for the local variable. The value of the global `i` is now accessible only indirectly within this block, through the pointer `external_i`. Attempting to assign the address of the **auto** variable `i` to a pointer does not work, since it may be different each time the block is entered. The variable `a` is declared as a **static** variable and initialized to 2. This `a`

does not conflict with the `a` in `main`, since **static** variables at the internal level are visible only within the block in which they are declared.

The variable `a` is increased by 2, giving 4 as the result. If the `other` function were called again in the same program, the initial value of `a` would be 4. Internal **static** variables keep their values when the program exits and then reenters the block in which they are declared.

See Also

Concepts

Storage-Class Specifiers for Internal-Level Declarations

Storage-Class Specifiers with Function Declarations

You can use either the **static** or the **extern** storage-class specifier in function declarations. Functions always have global lifetimes.

Microsoft Specific

Function declarations at the internal level have the same meaning as function declarations at the external level. This means that a function is visible from its point of declaration throughout the rest of the translation unit even if it is declared at local scope.

END Microsoft Specific

The visibility rules for functions vary slightly from the rules for variables, as follows:

- A function declared to be **static** is visible only within the source file in which it is defined. Functions in the same source file can call the **static** function, but functions in other source files cannot access it directly by name. You can declare another **static** function with the same name in a different source file without conflict.
- Functions declared as **extern** are visible throughout all source files in the program (unless you later redeclare such a function as **static**). Any function can call an **extern** function.
- Function declarations that omit the storage-class specifier are **extern** by default.

Microsoft Specific

Microsoft allows redefinition of an **extern** identifier as **static**.

END Microsoft Specific

See Also

Reference

C Storage Classes

C Type Specifiers

Type specifiers in declarations define the type of a variable or function declaration.

Syntax
type-specifier:

void

char

short

int

long

float

double

signed

unsigned

struct-or-union-specifier

enum-specifier

typedef-name

The **signed char**, **signed int**, **signed short int**, and **signed long int** types, together with their **unsigned** counterparts and **enum**, are called "integral" types. The **float**, **double**, and **long double** type specifiers are referred to as "floating" or "floating-point" types. You can use any integral or floating-point type specifier in a variable or function declaration. If a *type-specifier* is not provided in a declaration, it is taken to be **int**.

The optional keywords **signed** and **unsigned** can precede or follow any of the integral types, except **enum**, and can also be used alone as type specifiers, in which case they are understood as **signed int** and **unsigned int**, respectively. When used alone, the keyword **int** is assumed to be **signed**. When used alone, the keywords **long** and **short** are understood as **long int** and **short int**.

Enumeration types are considered basic types. Type specifiers for enumeration types are discussed in Enumeration Declarations.

The keyword **void** has three uses: to specify a function return type, to specify an argument-type list for a function that takes no arguments, and to specify a pointer to an unspecified type. You can use the **void** type to declare functions that return no value or to declare a pointer to an unspecified type. See Arguments for information on **void** when it appears alone within the parentheses following a function name.

Microsoft Specific

Type checking is now ANSI-compliant, which means that type **short** and type **int** are distinct types. For example, this is a redefinition in the Microsoft C compiler that was accepted by previous versions of the compiler.

```
int   myfunc();
short myfunc();
```

This next example also generates a warning about indirection to different types:

```
int *pi;
short *ps;

ps = pi;  /* Now generates warning */
```

The Microsoft C compiler also generates warnings for differences in sign. For example:

```
signed int *pi;
```

```
unsigned int *pu

pi = pu;  /* Now generates warning */
```

Type **void** expressions are evaluated for side effects. You cannot use the (nonexistent) value of an expression that has type **void** in any way, nor can you convert a **void** expression (by implicit or explicit conversion) to any type except **void**. If you do use an expression of any other type in a context where a **void** expression is required, its value is discarded.

To conform to the ANSI specification, **void**** cannot be used as **int****. Only **void*** can be used as a pointer to an unspecified type.

END Microsoft Specific

You can create additional type specifiers with **typedef** declarations, as described in Typedef Declarations. See Storage of Basic Types for information on the size of each type.

See Also
Concepts
Declarations and Types

Data Type Specifiers and Equivalents

This book generally uses the forms of the type specifiers listed in the following table rather than the long forms, and it assumes that the **char** type is signed by default. Therefore, throughout this book, **char** is equivalent to **signed char**.

Type Specifiers and Equivalents

Type Specifier	Equivalent(s)
signed char1	**char**
signed int	**signed**, **int**
signed short int	**short**, **signed short**
signed long int	**long**, **signed long**
unsigned char	—
unsigned int	**unsigned**
unsigned short int	**unsigned short**
unsigned long int	**unsigned long**
float	—
long double2	—

1 When you make the **char** type unsigned by default (by specifying the /J compiler option), you cannot abbreviate **signed char** as **char**.

2 In 32-bit operating systems, the Microsoft C compiler maps **long double** to type **double**.

Microsoft Specific

You can specify the /J compiler option to change the default **char** type from signed to unsigned. When this option is in effect, **char** means the same as **unsigned char**, and you must use the **signed** keyword to declare a signed character value. If a **char** value is explicitly declared signed, the /J option does not affect it, and the value is sign-extended when widened to an **int** type. The **char** type is zero-extended when widened to **int** type.

END Microsoft Specific

See Also

Reference

C Type Specifiers

Type Qualifiers

Type qualifiers give one of two properties to an identifier. The **const** type qualifier declares an object to be nonmodifiable. The **volatile** type qualifier declares an item whose value can legitimately be changed by something beyond the control of the program in which it appears, such as a concurrently executing thread.

The two type qualifiers, **const** and **volatile**, can appear only once in a declaration. Type qualifiers can appear with any type specifier; however, they cannot appear after the first comma in a multiple item declaration. For example, the following declarations are legal:

```
typedef volatile int VI;
const int ci;
```

These declarations are not legal:

```
typedef int *i, volatile *vi;
float f, const cf;
```

Type qualifiers are relevant only when accessing identifiers as l-values in expressions. See L-Value and R-Value Expressions for information about l-values and expressions.

Syntax

type-qualifier:

constvolatile

The following are legal **const** and **volatile** declarations:

```
int const *p_ci;       /* Pointer to constant int */
int const (*p_ci);     /* Pointer to constant int */
int *const cp_i;       /* Constant pointer to int */
int (*const cp_i);     /* Constant pointer to int */
int volatile vint;     /* Volatile integer        */
```

If the specification of an array type includes type qualifiers, the element is qualified, not the array type. If the specification of the function type includes qualifiers, the behavior is undefined. Neither **volatile** nor **const** affects the range of values or arithmetic properties of the object.

This list describes how to use **const** and **volatile**.

- The **const** keyword can be used to modify any fundamental or aggregate type, or a pointer to an object of any type, or a **typedef**. If an item is declared with only the **const** type qualifier, its type is taken to be **const int**. A **const** variable can be initialized or can be placed in a read-only region of storage. The **const** keyword is useful for declaring pointers to **const** since this requires the function not to change the pointer in any way.
- The compiler assumes that, at any point in the program, a **volatile** variable can be accessed by an unknown process that uses or modifies its value. Therefore, regardless of the optimizations specified on the command line, the code for each assignment to or reference of a **volatile** variable must be generated even if it appears to have no effect.

  If **volatile** is used alone, **int** is assumed. The **volatile** type specifier can be used to provide reliable access to special memory locations. Use **volatile** with data objects that may be accessed or altered by signal handlers, by concurrently executing programs, or by special hardware such as memory-mapped I/O control registers. You can declare a variable as **volatile** for its lifetime, or you can cast a single reference to be **volatile**.
- An item can be both **const** and **volatile**, in which case the item could not be legitimately modified by its own program, but could be modified by some asynchronous process.

See Also

Concepts

Declarations and Types

Declarators and Variable Declarations

The rest of this section describes the form and meaning of declarations for variable types summarized in this list. In particular, the remaining sections explain how to declare the following:

Type of Variable	Description
Simple variables	Single-value variables with integral or floating-point type
Arrays	Variables composed of a collection of elements with the same type
Pointers	Variables that point to other variables and contain variable locations (in the form of addresses) instead of values
Enumeration variables	Simple variables with integral type that hold one value from a set of named integer constants
Structures	Variables composed of a collection of values that can have different types
Unions	Variables composed of several values of different types that occupy the same storage space

A declarator is the part of a declaration that specifies the name that is to be introduced into the program. It can include modifiers such as ***** (pointer-to) and any of the Microsoft calling-convention keywords.

Microsoft Specific

In the declarator

```
__declspec(thread) char *var;
```

`char` is the type specifier, `__declspec(thread)` and `*` are the modifiers, and `var` is the identifier's name.

END Microsoft Specific

You use declarators to declare arrays of values, pointers to values, and functions returning values of a specified type. Declarators appear in the array and pointer declarations described later in this section.

Syntax

declarator:

pointer $_{opt}$ *direct-declarator*

direct-declarator:

identifier

(*declarator* **)**

direct-declarator **[** *constant-expression* $_{opt}$ **]**

direct-declarator **(** *parameter-type-list* **)**

direct-declarator **(** *identifier-list* $_{opt}$ **)**

pointer:

***** *type-qualifier-list* $_{opt}$

***** *type-qualifier-list* $_{opt}$ *pointer*

type-qualifier-list:

type-qualifier

type-qualifier-list type-qualifier

Note

See the syntax for *declaration* in Overview of Declarations or C Language Syntax Summary for the syntax that references a *declarator*.

When a declarator consists of an unmodified identifier, the item being declared has a base type. If an asterisk (*****) appears to the left of an identifier, the type is modified to a pointer type. If the identifier is followed by brackets (**[ ]**), the type is modified to an array type. If the identifier is followed by parentheses, the type is modified to a function type. For more information about interpreting precedence within declarations, see Interpreting More Complex Declarators.

Each declarator declares at least one identifier. A declarator must include a type specifier to be a complete declaration. The type specifier gives the type of the elements of an array type, the type of object addressed by a pointer type, or the return type of a function.

Array and pointer declarations are discussed in more detail later in this section. The following examples illustrate a few simple forms of declarators:

```
int list[20]; // Declares an array of 20 int values named list
char *cp;     // Declares a pointer to a char value
double func( void ); // Declares a function named func, with no
                     // arguments, that returns a double value
int *aptr[10] // Declares an array of 10 pointers
```

Microsoft Specific

The Microsoft C compiler does not limit the number of declarators that can modify an arithmetic, structure, or union type. The number is limited only by available memory.

END Microsoft Specific

See Also

Concepts

Declarations and Types

Simple Variable Declarations

The declaration of a simple variable, the simplest form of a direct declarator, specifies the variable's name and type. It also specifies the variable's storage class and data type.

Storage classes or types (or both) are required on variable declarations. Untyped variables (such as `var;`) generate warnings.

Syntax

declarator:

pointer opt

direct-declarator

direct-declarator:

identifier

identifier:

nondigit

identifier nondigit

identifier digit

For arithmetic, structure, union, enumerations, and void types, and for types represented by **typedef** names, simple declarators can be used in a declaration since the type specifier supplies all the typing information. Pointer, array, and function types require more complicated declarators.

You can use a list of identifiers separated by commas (**,**) to specify several variables in the same declaration. All variables defined in the declaration have the same base type. For example:

```
int x, y;        /* Declares two simple variables of type int */
int const z = 1; /* Declares a constant value of type int */
```

The variables `x` and `y` can hold any value in the set defined by the **int** type for a particular implementation. The simple object `z` is initialized to the value 1 and is not modifiable.

If the declaration of `z` was for an uninitialized static variable or was at file scope, it would receive an initial value of 0, and that value would be unmodifiable.

```
unsigned long reply, flag; /* Declares two variables
                              named reply and flag     */
```

In this example, both the variables, `reply` and `flag`, have **unsigned long** type and hold unsigned integral values.

See Also

Concepts

Declarators and Variable Declarations

C Enumeration Declarations

An enumeration consists of a set of named integer constants. An enumeration type declaration gives the name of the (optional) enumeration tag and defines the set of named integer identifiers (called the "enumeration set," "enumerator constants," "enumerators," or "members"). A variable with enumeration type stores one of the values of the enumeration set defined by that type.

Variables of **enum** type can be used in indexing expressions and as operands of all arithmetic and relational operators. Enumerations provide an alternative to the **#define** preprocessor directive with the advantages that the values can be generated for you and obey normal scoping rules.

In ANSI C, the expressions that define the value of an enumerator constant always have **int** type; thus, the storage associated with an enumeration variable is the storage required for a single **int** value. An enumeration constant or a value of enumerated type can be used anywhere the C language permits an integer expression.

Syntax
enum-specifier:

enum *identifier* $_{\text{opt}}$ **{** *enumerator-list* **}**

enum *identifier*

The optional *identifier* names the enumeration type defined by *enumerator-list*. This identifier is often called the "tag" of the enumeration specified by the list. A type specifier of the form

```
enum identifier { enumerator-list }
```

declares *identifier* to be the tag of the enumeration specified by the *enumerator-list* nonterminal. The *enumerator-list* defines the "enumerator content." The *enumerator-list* is described in detail below.

If the declaration of a tag is visible, subsequent declarations that use the tag but omit *enumerator-list* specify the previously declared enumerated type. The tag must refer to a defined enumeration type, and that enumeration type must be in current scope. Since the enumeration type is defined elsewhere, the *enumerator-list* does not appear in this declaration. Declarations of types derived from enumerations and **typedef** declarations for enumeration types can use the enumeration tag before the enumeration type is defined.

Syntax
enumerator-list:

enumerator

enumerator-list **,** *enumerator*

enumerator:

enumeration-constant

enumeration-constant **=** *constant-expression*

enumeration-constant:

identifier

Each *enumeration-constant* in an *enumeration-list* names a value of the enumeration set. By default, the first *enumeration-constant* is associated with the value 0. The next *enumeration-constant* in the list is associated with the value of (*constant-expression* + 1), unless you explicitly associate it with another value. The name of an *enumeration-constant* is equivalent to its value.

You can use *enumeration-constant* = *constant-expression* to override the default sequence of values. Thus, if *enumeration-constant* = *constant-expression* appears in the *enumerator-list*, the *enumeration-constant* is associated with the value given by *constant-expression*. The *constant-expression* must have **int** type and can be negative.

The following rules apply to the members of an enumeration set:

- An enumeration set can contain duplicate constant values. For example, you could associate the value 0 with two different identifiers, perhaps named `null` and `zero`, in the same set.

- The identifiers in the enumeration list must be distinct from other identifiers in the same scope with the same visibility, including ordinary variable names and identifiers in other enumeration lists.
- Enumeration tags obey the normal scoping rules. They must be distinct from other enumeration, structure, and union tags with the same visibility.

Examples

These examples illustrate enumeration declarations:

```
enum DAY            /* Defines an enumeration type    */
{
    saturday,       /* Names day and declares a       */
    sunday = 0,     /* variable named workday with    */
    monday,         /* that type                      */
    tuesday,
    wednesday,      /* wednesday is associated with 3 */
    thursday,
    friday
} workday;
```

The value 0 is associated with `saturday` by default. The identifier `sunday` is explicitly set to 0. The remaining identifiers are given the values 1 through 5 by default.

In this example, a value from the set `DAY` is assigned to the variable `today`.

```
enum DAY today = wednesday;
```

Note that the name of the enumeration constant is used to assign the value. Since the `DAY` enumeration type was previously declared, only the enumeration tag `DAY` is necessary.

To explicitly assign an integer value to a variable of an enumerated data type, use a type cast:

```
workday = ( enum DAY ) ( day_value - 1 );
```

This cast is recommended in C but is not required.

```
enum BOOLEAN  /* Declares an enumeration data type called BOOLEAN */
{
    false,     /* false = 0, true = 1 */
    true
};

enum BOOLEAN end_flag, match_flag; /* Two variables of type BOOLEAN */
```

This declaration can also be specified as

```
enum BOOLEAN { false, true } end_flag, match_flag;\
```

or as

```
enum BOOLEAN { false, true } end_flag;
enum BOOLEAN match_flag;
```

An example that uses these variables might look like this:

```
if ( match_flag == false )
    {
     .
     .   /* statement */
     .
    }
```

```
    end_flag = true;
```

Unnamed enumerator data types can also be declared. The name of the data type is omitted, but variables can be declared. The variable `response` is a variable of the type defined:

```
enum { yes, no } response;
```

See Also

Reference

C++ Enumeration Declarations

Structure Declarations

A "structure declaration" names a type and specifies a sequence of variable values (called "members" or "fields" of the structure) that can have different types. An optional identifier, called a "tag," gives the name of the structure type and can be used in subsequent references to the structure type. A variable of that structure type holds the entire sequence defined by that type. Structures in C are similar to the types known as "records" in other languages.

Syntax

struct-or-union-specifier:

struct-or-union identifier $_{\text{opt}}$ **{** *struct-declaration-list* **}**

struct-or-union identifier

struct-or-union:

struct

union

struct-declaration-list:

struct-declaration

struct-declaration-list struct-declaration

The structure content is defined to be

struct-declaration:

specifier-qualifier-list struct-declarator-list **;**

specifier-qualifier-list:

type-specifier specifier-qualifier-list $_{\text{opt}}$

type-qualifier specifier-qualifier-list $_{\text{opt}}$

struct-declarator-list:

struct-declarator

struct-declarator-list **,** *struct-declarator*

struct-declarator:

declarator

The declaration of a structure type does not set aside space for a structure. It is only a template for later declarations of structure variables.

A previously defined *identifier* (tag) can be used to refer to a structure type defined elsewhere. In this case, *struct-declaration-list* cannot be repeated as long as the definition is visible. Declarations of pointers to structures and typedefs for structure types can use the structure tag before the structure type is defined. However, the structure definition must be encountered prior to any actual use of the size of the fields. This is an incomplete definition of the type and the type tag. For this definition to be completed, a type definition must appear later in the same scope.

The *struct-declaration-list* specifies the types and names of the structure members. A *struct-declaration-list* argument contains one or more variable or bit-field declarations.

Each variable declared in *struct-declaration-list* is defined as a member of the structure type. Variable declarations within *struct-declaration-list* have the same form as other variable declarations discussed in this section, except that the declarations cannot contain storage-class specifiers or initializers. The structure members can have any variable types except type **void**, an incomplete type, or a function type.

A member cannot be declared to have the type of the structure in which it appears. However, a member can be declared as a pointer to the structure type in which it appears as long as the structure type has a tag. This allows you to create linked lists of structures.

Structures follow the same scoping as other identifiers. Structure identifiers must be distinct from other structure, union, and

enumeration tags with the same visibility.

Each *struct-declaration* in a *struct-declaration-list* must be unique within the list. However, identifier names in a *struct-declaration-list* do not have to be distinct from ordinary variable names or from identifiers in other structure declaration lists.

Nested structures can also be accessed as though they were declared at the file-scope level. For example, given this declaration:

```
struct a
{
    int x;
    struct b
    {
      int y;
    } var2;
} var1;
```

these declarations are both legal:

```
struct a var3;
struct b var4;
```

Examples

These examples illustrate structure declarations:

```
struct employee   /* Defines a structure variable named temp */
{
    char name[20];
    int id;
    long class;
} temp;
```

The `employee` structure has three members: `name`, `id`, and `class`. The `name` member is a 20-element array, and `id` and `class` are simple members with **int** and **long** type, respectively. The identifier `employee` is the structure identifier.

```
struct employee student, faculty, staff;
```

This example defines three structure variables: `student`, `faculty`, and `staff`. Each structure has the same list of three members. The members are declared to have the structure type `employee`, defined in the previous example.

```
struct           /* Defines an anonymous struct and a */
{                /* structure variable named complex  */
    float x, y;
} complex;
```

The `complex` structure has two members with **float** type, `x` and `y`. The structure type has no tag and is therefore unnamed or anonymous.

```
struct sample   /* Defines a structure named x */
{
    char c;
    float *pf;
    struct sample *next;
} x;
```

The first two members of the structure are a **char** variable and a pointer to a **float** value. The third member, `next`, is declared as a pointer to the structure type being defined (`sample`).

Anonymous structures can be useful when the tag named is not needed. This is the case when one declaration defines all structure instances. For example:

```
struct
```

```
{
    int x;
    int y;
} mystruct;
```

Embedded structures are often anonymous.

```
struct somestruct
{
    struct    /* Anonymous structure */
    {
        int x, y;
    } point;
    int type;
} w;
```

Microsoft Specific

The compiler allows an unsized or zero-sized array as the last member of a structure. This can be useful if the size of a constant array differs when used in various situations. The declaration of such a structure looks like this:

struct *identifier***{** *set-of-declarations* *type array-name***[ ];}**;

Unsized arrays can appear only as the last member of a structure. Structures containing unsized array declarations can be nested within other structures as long as no further members are declared in any enclosing structures. Arrays of such structures are not allowed. The **sizeof** operator, when applied to a variable of this type or to the type itself, assumes 0 for the size of the array.

Structure declarations can also be specified without a declarator when they are members of another structure or union. The field names are promoted into the enclosing structure. For example, a nameless structure looks like this:

```
struct s
{
    float y;
    struct
    {
        int a, b, c;
    };
    char str[10];
} *p_s;
.
.
.
p_s->b = 100;  /* A reference to a field in the s structure */
```

See Structure and Union Members for information about structure references.

END Microsoft Specific

See Also

Concepts

Declarators and Variable Declarations

C Bit Fields

In addition to declarators for members of a structure or union, a structure declarator can also be a specified number of bits, called a "bit field." Its length is set off from the declarator for the field name by a colon. A bit field is interpreted as an integral type.

Syntax
struct-declarator:

declarator

type-specifier declarator $_{\text{opt}}$ **:** *constant-expression*

The *constant-expression* specifies the width of the field in bits. The *type-specifier* for the *declarator* must be **unsigned int**, **signed int**, or **int**, and the *constant-expression* must be a nonnegative integer value. If the value is zero, the declaration has no *declarator*. Arrays of bit fields, pointers to bit fields, and functions returning bit fields are not allowed. The optional *declarator* names the bit field. Bit fields can only be declared as part of a structure. The address-of operator (**&**) cannot be applied to bit-field components.

Unnamed bit fields cannot be referenced, and their contents at run time are unpredictable. They can be used as "dummy" fields, for alignment purposes. An unnamed bit field whose width is specified as 0 guarantees that storage for the member following it in the *struct-declaration-list* begins on an **int** boundary.

Bit fields must also be long enough to contain the bit pattern. For example, these two statements are not legal:

```
short a:17;        /* Illegal! */
int long y:33;     /* Illegal! */
```

This example defines a two-dimensional array of structures named `screen`.

```
struct
{
    unsigned short icon : 8;
    unsigned short color : 4;
    unsigned short underline : 1;
    unsigned short blink : 1;
} screen[25][80];
```

The array contains 2,000 elements. Each element is an individual structure containing four bit-field members: `icon`, `color`, `underline`, and `blink`. The size of each structure is two bytes.

Bit fields have the same semantics as the integer type. This means a bit field is used in expressions in exactly the same way as a variable of the same base type would be used, regardless of how many bits are in the bit field.

Microsoft Specific

Bit fields defined as **int** are treated as signed. A Microsoft extension to the ANSI C standard allows **char** and **long** types (both **signed** and **unsigned**) for bit fields. Unnamed bit fields with base type **long**, **short**, or **char** (**signed** or **unsigned**) force alignment to a boundary appropriate to the base type.

Bit fields are allocated within an integer from least-significant to most-significant bit. In the following code

```
struct mybitfields
{
    unsigned short a : 4;
    unsigned short b : 5;
    unsigned short c : 7;
} test;

int main( void );
{
    test.a = 2;
    test.b = 31;
    test.c = 0;
```

```
}
```

the bits would be arranged as follows:

```
00000001 11110010
ccccccb bbbbaaaa
```

Since the 8086 family of processors stores the low byte of integer values before the high byte, the integer `0x01F2` above would be stored in physical memory as `0xF2` followed by `0x01`.

END Microsoft Specific

See Also

Concepts

Structure Declarations

Storage and Alignment of Structures

Microsoft Specific

Structure members are stored sequentially in the order in which they are declared: the first member has the lowest memory address and the last member the highest.

Every data object has an *alignment-requirement*. For structures, the requirement is the largest of its members. Every object is allocated an *offset* so that

offset **%** *alignment-requirement* **==** 0

Adjacent bit fields are packed into the same 1-, 2-, or 4-byte allocation unit if the integral types are the same size and if the next bit field fits into the current allocation unit without crossing the boundary imposed by the common alignment requirements of the bit fields.

To conserve space or to conform to existing data structures, you may want to store structures more or less compactly. The /Zp[*n*] compiler option and the #pragma pack control how structure data is "packed" into memory. When you use the /Zp[*n*] option, where *n* is 1, 2, 4, 8, or 16, each structure member after the first is stored on byte boundaries that are either the alignment requirement of the field or the packing size (*n*), whichever is smaller. Expressed as a formula, the byte boundaries are the

```
min( n, sizeof( item ) )
```

where *n* is the packing size expressed with the /Zp[*n*] option and *item* is the structure member. The default packing size is /Zp8.

To use the **pack** pragma to specify packing other than the packing specified on the command line for a particular structure, give the **pack** pragma, where the packing size is 1, 2, 4, 8, or 16, before the structure. To reinstate the packing given on the command line, specify the **pack** pragma with no arguments.

Bit fields default to size **long** for the Microsoft C compiler. Structure members are aligned on the size of the type or the /Zp[*n*] size, whichever is smaller. The default size is 4.

END Microsoft Specific

See Also

Concepts

Structure Declarations

Union Declarations

A "union declaration" specifies a set of variable values and, optionally, a tag naming the union. The variable values are called "members" of the union and can have different types. Unions are similar to "variant records" in other languages.

Syntax

struct-or-union-specifier:

struct-or-union identifier $_{opt}$ **{** *struct-declaration-list* **}**

struct-or-union identifier

struct-or-union:

struct

union

struct-declaration-list:

struct-declaration

struct-declaration-list struct-declaration

The union content is defined to be

struct-declaration:

specifier-qualifier-list struct-declarator-list **;**

specifier-qualifier-list:

type-specifier specifier-qualifier-list $_{opt}$

type-qualifier specifier-qualifier-list $_{opt}$

struct-declarator-list:

struct-declarator

struct-declarator-list **,** *struct-declarator*

A variable with **union** type stores one of the values defined by that type. The same rules govern structure and union declarations. Unions can also have bit fields.

Members of unions cannot have an incomplete type, type **void**, or function type. Therefore members cannot be an instance of the union but can be pointers to the union type being declared.

A union type declaration is a template only. Memory is not reserved until the variable is declared.

Note
If a union of two types is declared and one value is stored, but the union is accessed with the other type, the results are unreliable. For example, a union of **float** and **int** is declared. A **float** value is stored, but the program later accesses the value as an **int**. In such a situation, the value would depend on the internal storage of **float** values. The integer value would not be reliable.

Examples

The following are examples of unions:

```
union sign   /* A definition and a declaration */
{
    int svar;
    unsigned uvar;
} number;
```

This example defines a union variable with `sign` type and declares a variable named `number` that has two members: `svar`, a

signed integer, and `uvar`, an unsigned integer. This declaration allows the current value of `number` to be stored as either a signed or an unsigned value. The tag associated with this union type is `sign`.

```
union                /* Defines a two-dimensional */
{                    /*  array named screen */
    struct
    {
      unsigned int icon : 8;
      unsigned color : 4;
    } window1;
    int screenval;
} screen[25][80];
```

The `screen` array contains 2,000 elements. Each element of the array is an individual union with two members: `window1` and `screenval`. The `window1` member is a structure with two bit-field members, `icon` and `color`. The `screenval` member is an **int**. At any given time, each union element holds either the **int** represented by `screenval` or the structure represented by `window1`.

Microsoft Specific

Nested unions can be declared anonymously when they are members of another structure or union. This is an example of a nameless union:

```
struct str
{
    int a, b;
    union            / * Unnamed union */
    {
      char c[4];
      long l;
      float f;
   };
   char c_array[10];
} my_str;
.
.
.
my_str.l == 0L;  /* A reference to a field in the my_str union */
```

Unions are often nested within a structure that includes a field giving the type of data contained in the union at any particular time. This is an example of a declaration for such a union:

```
struct x
{
    int type_tag;
    union
    {
      int x;
      float y;
    }
}
```

See Structure and Union Members for information about referencing unions.

END Microsoft Specific

See Also

Concepts

Declarators and Variable Declarations

Storage of Unions

The storage associated with a union variable is the storage required for the largest member of the union. When a smaller member is stored, the union variable can contain unused memory space. All members are stored in the same memory space and start at the same address. The stored value is overwritten each time a value is assigned to a different member. For example:

```
union         /* Defines a union named x */
{
    char *a, b;
    float f[20];
} x;
```

The members of the `x` union are, in order of their declaration, a pointer to a **char** value, a **char** value, and an array of **float** values. The storage allocated for `x` is the storage required for the 20-element array `f`, since `f` is the longest member of the union. Because no tag is associated with the union, its type is unnamed or "anonymous."

See Also

Concepts

Union Declarations

Array Declarations

An "array declaration" names the array and specifies the type of its elements. It can also define the number of elements in the array. A variable with array type is considered a pointer to the type of the array elements.

Syntax

declaration:

declaration-specifiers init-declarator-list $_{opt}$ **;**

init-declarator-list:

init-declarator

init-declarator-list **,** *init-declarator*

init-declarator:

declarator

declarator **=** *initializer*

declarator:

pointer $_{opt}$ *direct-declarator*

direct-declarator:

direct-declarator **[** *constant-expression* $_{opt}$ **]**

Because *constant-expression* is optional, the syntax has two forms:

- The first form defines an array variable. The *constant-expression* argument within the brackets specifies the number of elements in the array. The *constant-expression*, if present, must have integral type, and a value larger than zero. Each element has the type given by *type-specifier*, which can be any type except **void**. An array element cannot be a function type.
- The second form declares a variable that has been defined elsewhere. It omits the *constant-expression* argument in brackets, but not the brackets. You can use this form only if you previously have initialized the array, declared it as a parameter, or declared it as a reference to an array explicitly defined elsewhere in the program.

In both forms, *direct-declarator* names the variable and can modify the variable's type. The brackets (**[ ]**) following *direct-declarator* modify the declarator to an array type.

Type qualifiers can appear in the declaration of an object of array type, but the qualifiers apply to the elements rather than the array itself.

You can declare an array of arrays (a "multidimensional" array) by following the array declarator with a list of bracketed constant expressions in this form:

```
type-specifier declarator [constant-expression] [constant-expression] ...
```

Each *constant-expression* in brackets defines the number of elements in a given dimension: two-dimensional arrays have two bracketed expressions, three-dimensional arrays have three, and so on. You can omit the first constant expression if you have initialized the array, declared it as a parameter, or declared it as a reference to an array explicitly defined elsewhere in the program.

You can define arrays of pointers to various types of objects by using complex declarators, as described in Interpreting More Complex Declarators.

Arrays are stored by row. For example, the following array consists of two rows with three columns each:

```
char A[2][3];
```

The three columns of the first row are stored first, followed by the three columns of the second row. This means that the last

subscript varies most quickly.

To refer to an individual element of an array, use a subscript expression, as described in Postfix Operators.

Examples

These examples illustrate array declarations:

```
float matrix[10][15];
```

The two-dimensional array named `matrix` has 150 elements, each having **float** type.

```
struct {
    float x, y;
} complex[100];
```

This is a declaration of an array of structures. This array has 100 elements; each element is a structure containing two members.

```
extern char *name[];
```

This statement declares the type and name of an array of pointers to **char**. The actual definition of `name` occurs elsewhere.

Microsoft Specific

The type of integer required to hold the maximum size of an array is the size of **size_t**. Defined in the header file STDDEF.H, **size_t** is an **unsigned int** with the range 0x00000000 to 0x7CFFFFFF.

END Microsoft Specific

See Also

Concepts

Declarators and Variable Declarations

Storage of Arrays

The storage associated with an array type is the storage required for all of its elements. The elements of an array are stored in contiguous and increasing memory locations, from the first element to the last.

See Also

Concepts

Array Declarations

Pointer Declarations

A "pointer declaration" names a pointer variable and specifies the type of the object to which the variable points. A variable declared as a pointer holds a memory address.

Syntax

declarator:

pointer $_{opt}$ *direct-declarator*

direct-declarator:

identifier

(*declarator* **)**

direct-declarator **[** *constant-expression* $_{opt}$ **]**

direct-declarator **(** *parameter-type-list* **)**

direct-declarator **(** *identifier-list* $_{opt}$ **)**

pointer:

***** *type-qualifier-list* $_{opt}$

***** *type-qualifier-list* $_{opt}$ *pointer*

type-qualifier-list:

type-qualifier

type-qualifier-list type-qualifier

The *type-specifier* gives the type of the object, which can be any basic, structure, or union type. Pointer variables can also point to functions, arrays, and other pointers. (For information on declaring and interpreting more complex pointer types, refer to Interpreting More Complex Declarators.)

By making the *type-specifier* **void**, you can delay specification of the type to which the pointer refers. Such an item is referred to as a "pointer to **void**" and is written as `void *`. A variable declared as a pointer to **void** can be used to point to an object of any type. However, to perform most operations on the pointer or on the object to which it points, the type to which it points must be explicitly specified for each operation. (Variables of type **char *** and type **void *** are assignment-compatible without a type cast.) Such conversion can be accomplished with a type cast (see Type-Cast Conversions for more information).

The *type-qualifier* can be either **const** or **volatile**, or both. These specify, respectively, that the pointer cannot be modified by the program itself (**const**), or that the pointer can legitimately be modified by some process beyond the control of the program (**volatile**). (See Type Qualifiers for more information on **const** and **volatile**.)

The *declarator* names the variable and can include a type modifier. For example, if *declarator* represents an array, the type of the pointer is modified to be a pointer to an array.

You can declare a pointer to a structure, union, or enumeration type before you define the structure, union, or enumeration type. You declare the pointer by using the structure or union tag as shown in the examples below. Such declarations are allowed because the compiler does not need to know the size of the structure or union to allocate space for the pointer variable.

Examples

The following examples illustrate pointer declarations.

```
char *message; /* Declares a pointer variable named message */
```

The `message` pointer points to a variable with **char** type.

```
int *pointers[10];  /* Declares an array of pointers */
```

The `pointers` array has 10 elements; each element is a pointer to a variable with **int** type.

```
int (*pointer)[10]; /* Declares a pointer to an array of 10 elements */
```

The pointer variable points to an array with 10 elements. Each element in this array has **int** type.

```
int const *x;      /* Declares a pointer variable, x,
                      to a constant value */
```

The pointer `x` can be modified to point to a different **int** value, but the value to which it points cannot be modified.

```
const int some_object = 5 ;
int other_object = 37;
int *const y = &fixed_object;
const volatile *const z = &some_object;
int *const volatile w = &some_object;
```

The variable `y` in these declarations is declared as a constant pointer to an **int** value. The value it points to can be modified, but the pointer itself must always point to the same location: the address of `fixed_object`. Similarly, `z` is a constant pointer, but it is also declared to point to an **int** whose value cannot be modified by the program. The additional specifier `volatile` indicates that although the value of the **const int** pointed to by `z` cannot be modified by the program, it could legitimately be modified by a process running concurrently with the program. The declaration of `w` specifies that the program cannot change the value pointed to and that the program cannot modify the pointer.

```
struct list *next, *previous; /* Uses the tag for list */
```

This example declares two pointer variables, `next` and `previous`, that point to the structure type `list`. This declaration can appear before the definition of the `list` structure type (see the next example), as long as the `list` type definition has the same visibility as the declaration.

```
struct list
{
    char *token;
    int count;
    struct list *next;
} line;
```

The variable `line` has the structure type named `list`. The `list` structure type has three members: the first member is a pointer to a **char** value, the second is an **int** value, and the third is a pointer to another `list` structure.

```
struct id
{
    unsigned int id_no;
    struct name *pname;
} record;
```

The variable `record` has the structure type `id`. Note that `pname` is declared as a pointer to another structure type named `name`. This declaration can appear before the `name` type is defined.

See Also

Concepts

Declarators and Variable Declarations

Storage of Addresses

The amount of storage required for an address and the meaning of the address depend on the implementation of the compiler. Pointers to different types are not guaranteed to have the same length. Therefore, **sizeof(char *)** is not necessarily equal to **sizeof(int *)**.

Microsoft Specific

For the Microsoft C compiler, **sizeof(char *)** is equal to **sizeof(int *)**.

END Microsoft Specific

See Also

Concepts

Pointer Declarations

Based Pointers (C)

Microsoft Specific

__based (C++ Reference)

For the Microsoft 32-bit and 64-bit C compilers, a based pointer is a 32-bit or 64-bit offset from a 32-bit or 64-bit pointer base. Based addressing is useful for exercising control over sections where objects are allocated, thereby decreasing the size of the executable file and increasing execution speed. In general, the form for specifying a based pointer is

```
type __based( base ) declarator
```

The "based on pointer" variant of based addressing enables specification of a pointer as a base. The based pointer, then, is an offset into the memory section starting at the beginning of the pointer on which it is based. Pointers based on pointer addresses are the only form of the **__based** keyword valid in 32-bit and 64-bit compilations. In such compilations, they are 32-bit or 64-bit displacements from a 32-bit or 64-bit base.

One use for pointers based on pointers is for persistent identifiers that contain pointers. A linked list that consists of pointers based on a pointer can be saved to disk, then reloaded to another place in memory, with the pointers remaining valid.

The following example shows a pointer based on a pointer.

```
void *vpBuffer;

struct llist_t
{
    void __based( vpBuffer ) *vpData;
    struct llist_t __based( vpBuffer ) *llNext;
};
```

The pointer `vpBuffer` is assigned the address of memory allocated at some later point in the program. The linked list is relocated relative to the value of `vpBuffer`.

END Microsoft Specific

See Also

Concepts

Declarators and Variable Declarations

C Abstract Declarators

An abstract declarator is a declarator without an identifier, consisting of one or more pointer, array, or function modifiers. The pointer modifier (*) always precedes the identifier in a declarator; array (**[ ]**) and function (**( )**) modifiers follow the identifier. Knowing this, you can determine where the identifier would appear in an abstract declarator and interpret the declarator accordingly. See Interpreting More Complex Declarators for additional information and examples of complex declarators. Generally **typedef** can be used to simplify declarators. See Typedef Declarations.

Abstract declarators can be complex. Parentheses in a complex abstract declarator specify a particular interpretation, just as they do for the complex declarators in declarations.

These examples illustrate abstract declarators:

```
int *         // The type name for a pointer to type int:

int *[3]      // An array of three pointers to int

int (*) [5]   // A pointer to an array of five int

int *()       // A function with no parameter specification
              // returning a pointer to int

// A pointer to a function taking no arguments and
// returning an int

int (*) ( void )

// An array of an unspecified number of constant pointers to
// functions each with one parameter that has type unsigned int
// and an unspecified number of other parameters returning an int

int (*const []) ( unsigned int, ... )
```

Note

The abstract declarator consisting of a set of empty parentheses, **( )**, is not allowed because it is ambiguous. It is impossible t o determine whether the implied identifier belongs inside the parentheses (in which case it is an unmodified type) or before t he parentheses (in which case it is a function type).

See Also

Concepts

Declarators and Variable Declarations

Interpreting More Complex Declarators

You can enclose any declarator in parentheses to specify a particular interpretation of a "complex declarator." A complex declarator is an identifier qualified by more than one array, pointer, or function modifier. You can apply various combinations of array, pointer, and function modifiers to a single identifier. Generally **typedef** may be used to simplify declarations. See Typedef Declarations.

In interpreting complex declarators, brackets and parentheses (that is, modifiers to the right of the identifier) take precedence over asterisks (that is, modifiers to the left of the identifier). Brackets and parentheses have the same precedence and associate from left to right. After the declarator has been fully interpreted, the type specifier is applied as the last step. By using parentheses you can override the default association order and force a particular interpretation. Never use parentheses, however, around an identifier name by itself. This could be misinterpreted as a parameter list.

A simple way to interpret complex declarators is to read them "from the inside out," using the following four steps:

1. Start with the identifier and look directly to the right for brackets or parentheses (if any).
2. Interpret these brackets or parentheses, then look to the left for asterisks.
3. If you encounter a right parenthesis at any stage, go back and apply rules 1 and 2 to everything within the parentheses.
4. Apply the type specifier.

```
char *( *(*var)() )[10];
 ^   ^  ^ ^ ^   ^    ^
 7   6  4 2 1   3    5
```

In this example, the steps are numbered in order and can be interpreted as follows:

1. The identifier `var` is declared as
2. a pointer to
3. a function returning
4. a pointer to
5. an array of 10 elements, which are
6. pointers to
7. **char** values.

Examples

The following examples illustrate other complex declarations and show how parentheses can affect the meaning of a declaration.

```
int *var[5]; /* Array of pointers to int values */
```

The array modifier has higher priority than the pointer modifier, so `var` is declared to be an array. The pointer modifier applies to the type of the array elements; therefore, the array elements are pointers to **int** values.

```
int (*var)[5]; /* Pointer to array of int values */
```

In this declaration for `var`, parentheses give the pointer modifier higher priority than the array modifier, and `var` is declared to be a pointer to an array of five **int** values.

```
long *var( long, long ); /* Function returning pointer to long */
```

Function modifiers also have higher priority than pointer modifiers, so this declaration for `var` declares `var` to be a function returning a pointer to a **long** value. The function is declared to take two **long** values as arguments.

```
long (*var)( long, long ); /* Pointer to function returning long */
```

This example is similar to the previous one. Parentheses give the pointer modifier higher priority than the function modifier, and `var` is declared to be a pointer to a function that returns a **long** value. Again, the function takes two **long** arguments.

```
struct both       /* Array of pointers to functions */
{                 /*   returning structures         */
    int a;
    char b;
} ( *var[5] )( struct both, struct both );
```

The elements of an array cannot be functions, but this declaration demonstrates how to declare an array of pointers to functions instead. In this example, `var` is declared to be an array of five pointers to functions that return structures with two members. The arguments to the functions are declared to be two structures with the same structure type, `both`. Note that the parentheses surrounding `*var[5]` are required. Without them, the declaration is an illegal attempt to declare an array of functions, as shown below:

```
/* ILLEGAL */
struct both *var[5]( struct both, struct both );
```

The following statement declares an array of pointers.

```
unsigned int *(* const *name[5][10] ) ( void );
```

The `name` array has 50 elements organized in a multidimensional array. The elements are pointers to a pointer that is a constant. This constant pointer points to a function that has no parameters and returns a pointer to an unsigned type.

This next example is a function returning a pointer to an array of three **double** values.

```
double ( *var( double (*)[3] ) )[3];
```

In this declaration, a function returns a pointer to an array, since functions returning arrays are illegal. Here `var` is declared to be a function returning a pointer to an array of three **double** values. The function `var` takes one argument. The argument, like the return value, is a pointer to an array of three **double** values. The argument type is given by a complex *abstract-declarator*. The parentheses around the asterisk in the argument type are required; without them, the argument type would be an array of three pointers to **double** values. For a discussion and examples of abstract declarators, see Abstract Declarators.

```
union sign         /* Array of arrays of pointers */
{                  /* to pointers to unions       */
     int x;
     unsigned y;
} **var[5][5];
```

As the above example shows, a pointer can point to another pointer, and an array can contain arrays as elements. Here `var` is an array of five elements. Each element is a five-element array of pointers to pointers to unions with two members.

```
union sign *(*var[5])[5]; /* Array of pointers to arrays
                             of pointers to unions        */
```

This example shows how the placement of parentheses changes the meaning of the declaration. In this example, `var` is a five-element array of pointers to five-element arrays of pointers to unions. For examples of how to use **typedef** to avoid complex declarations, see Typedef Declarations.

See Also

Concepts

Declarations and Types

Initialization

An "initializer" is a value or a sequence of values to be assigned to the variable being declared. You can set a variable to an initial value by applying an initializer to the declarator in the variable declaration. The value or values of the initializer are assigned to the variable.

The following sections describe how to initialize variables of scalar, aggregate, and string types. "Scalar types" include all the arithmetic types, plus pointers. "Aggregate types" include arrays, structures, and unions.

See Also

Concepts

Declarations and Types

Initializing Scalar Types

When initializing scalar types, the value of the *assignment-expression* is assigned to the variable. The conversion rules for assignment apply. (See Type Conversions for information on conversion rules.)

Syntax

declaration:

declaration-specifiers init-declarator-list $_{opt}$ **;**

declaration-specifiers:

storage-class-specifier declaration-specifiers $_{opt}$

type-specifier declaration-specifiers $_{opt}$

type-qualifier declaration-specifiers $_{opt}$

init-declarator-list:

init-declarator

init-declarator-list **,** *init-declarator*

init-declarator:

declarator

declarator **=** *initializer* /* For scalar initialization */

initializer:

assignment-expression

You can initialize variables of any type, provided that you obey the following rules:

- Variables declared at the file-scope level can be initialized. If you do not explicitly initialize a variable at the external level, it is initialized to 0 by default.
- A constant expression can be used to initialize any global variable declared with the **static** *storage-class-specifier*. Variables declared to be **static** are initialized when program execution begins. If you do not explicitly initialize a global **static** variable, it is initialized to 0 by default, and every member that has pointer type is assigned a null pointer.
- Variables declared with the **auto** or **register** storage-class specifier are initialized each time execution control passes to the block in which they are declared. If you omit an initializer from the declaration of an **auto** or **register** variable, the initial value of the variable is undefined. For automatic and register values, the initializer is not restricted to being a constant; it can be any expression involving previously defined values, even function calls.
- The initial values for external variable declarations and for all **static** variables, whether external or internal, must be constant expressions. (For more information, see Constant Expressions.) Since the address of any externally declared or static variable is constant, it can be used to initialize an internally declared **static** pointer variable. However, the address of an **auto** variable cannot be used as a static initializer because it may be different for each execution of the block. You can use either constant or variable values to initialize **auto** and **register** variables.
- If the declaration of an identifier has block scope, and the identifier has external linkage, the declaration cannot have an initialization.

Examples

The following examples illustrate initializations:

```
int x = 10;
```

The integer variable `x` is initialized to the constant expression `10`.

```
register int *px = 0;
```

The pointer `px` is initialized to 0, producing a "null" pointer.

```
const int c = (3 * 1024);
```

This example uses a constant expression `(3 * 1024)` to initialize `c` to a constant value that cannot be modified because of the **const** keyword.

```
int *b = &x;
```

This statement initializes the pointer `b` with the address of another variable, `x`.

```
int *const a = &z;
```

The pointer `a` is initialized with the address of a variable named `z`. However, since it is specified to be a **const**, the variable `a` can only be initialized, never modified. It always points to the same location.

```
int GLOBAL ;

int function( void )
{
    int LOCAL ;
    static int *lp = &LOCAL;   /* Illegal initialization */
    static int *gp = &GLOBAL;  /* Legal initialization   */
    register int *rp = &LOCAL; /* Legal initialization   */
}
```

The global variable `GLOBAL` is declared at the external level, so it has global lifetime. The local variable `LOCAL` has **auto** storage class and only has an address during the execution of the function in which it is declared. Therefore, attempting to initialize the **static** pointer variable `lp` with the address of `LOCAL` is not permitted. The **static** pointer variable `gp` can be initialized to the address of `GLOBAL` because that address is always the same. Similarly, `*rp` can be initialized because `rp` is a local variable and can have a nonconstant initializer. Each time the block is entered, `LOCAL` has a new address, which is then assigned to `rp`.

See Also

Concepts

Initialization

Initializing Aggregate Types

An "aggregate" type is a structure, union, or array type. If an aggregate type contains members of aggregate types, the initialization rules apply recursively.

Syntax

initializer:

{ *initializer-list* **}** /* For aggregate initialization */

{ *initializer-list* **, }**

initializer-list:

initializer

initializer-list **,** *initializer*

The *initializer-list* is a list of initializers separated by commas. Each initializer in the list is either a constant expression or an initializer list. Therefore, initializer lists can be nested. This form is useful for initializing aggregate members of an aggregate type, as shown in the examples in this section. However, if the initializer for an automatic identifier is a single expression, it need not be a constant expression; it merely needs to have appropriate type for assignment to the identifier.

For each initializer list, the values of the constant expressions are assigned, in order, to the corresponding members of the aggregate variable.

If *initializer-list* has fewer values than an aggregate type, the remaining members or elements of the aggregate type are initialized to 0. The initial value of an automatic identifier not explicitly initialized is undefined. If *initializer-list* has more values than an aggregate type, an error results. These rules apply to each embedded initializer list, as well as to the aggregate as a whole.

A structure's initializer is either an expression of the same type, or a list of initializers for its members enclosed in curly braces (**{ }**). Unnamed bit-field members are not initialized.

When a union is initialized, *initializer-list* must be a single constant expression. The value of the constant expression is assigned to the first member of the union.

If an array has unknown size, the number of initializers determines the size of the array, and its type becomes complete. There is no way to specify repetition of an initializer in C, or to initialize an element in the middle of an array without providing all preceding values as well. If you need this operation in your program, write the routine in assembly language.

Note that the number of initializers can set the size of the array:

```
int x[ ] = { 0, 1, 2 }
```

If you specify the size and give the wrong number of initializers, however, the compiler generates an error.

Microsoft Specific

The maximum size for an array is defined by **size_t**. Defined in the header file STDDEF.H, **size_t** is an **unsigned int** with the range 0x00000000 to 0x7CFFFFFF.

END Microsoft Specific

Examples

This example shows initializers for an array.

```
int P[4][3] =
{
    { 1, 1, 1 },
    { 2, 2, 2 },
    { 3, 3, 3,},
    { 4, 4, 4,},
};
```

This statement declares `P` as a four-by-three array and initializes the elements of its first row to 1, the elements of its second row to 2, and so on through the fourth row. Note that the initializer list for the third and fourth rows contains commas after the last constant expression. The last initializer list (`{4, 4, 4,},`) is also followed by a comma. These extra commas are permitted but are not required; only commas that separate constant expressions from one another, and those that separate one initializer list from another, are required.

If an aggregate member has no embedded initializer list, values are simply assigned, in order, to each member of the subaggregate. Therefore, the initialization in the previous example is equivalent to the following:

```
int P[4][3] =
{
   1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4
};
```

Braces can also appear around individual initializers in the list and would help to clarify the example above.

When you initialize an aggregate variable, you must be careful to use braces and initializer lists properly. The following example illustrates the compiler's interpretation of braces in more detail:

```
typedef struct
{
    int n1, n2, n3;
} triplet;

triplet nlist[2][3] =
{
    { {  1, 2, 3 }, {  4, 5, 6 }, {  7, 8, 9 } },  /* Row 1 */
    { { 10,11,12 }, { 13,14,15 }, { 16,17,18 } }   /* Row 2 */
};
```

In this example, `nlist` is declared as a 2-by-3 array of structures, each structure having three members. Row 1 of the initialization assigns values to the first row of `nlist`, as follows:

1. The first left brace on row 1 signals the compiler that initialization of the first aggregate member of `nlist` (that is, `nlist[0]`) is beginning.
2. The second left brace indicates that initialization of the first aggregate member of `nlist[0]` (that is, the structure at `nlist[0][0]`) is beginning.
3. The first right brace ends initialization of the structure `nlist[0][0]`; the next left brace starts initialization of `nlist[0][1]`.
4. The process continues until the end of the line, where the closing right brace ends initialization of `nlist[0]`.

Row 2 assigns values to the second row of `nlist` in a similar way. Note that the outer sets of braces enclosing the initializers on rows 1 and 2 are required. The following construction, which omits the outer braces, would cause an error:

```
triplet nlist[2][3] =  /* THIS CAUSES AN ERROR */
{
     {  1, 2, 3 },{  4, 5, 6 },{  7, 8, 9 },   /* Line 1 */
     { 10,11,12 },{ 13,14,15 },{ 16,17,18 }    /* Line 2 */
};
```

In this construction, the first left brace on line 1 starts the initialization of `nlist[0]`, which is an array of three structures. The values 1, 2, and 3 are assigned to the three members of the first structure. When the next right brace is encountered (after the value 3), initialization of `nlist[0]` is complete, and the two remaining structures in the three-structure array are automatically initialized to 0. Similarly, `{ 4,5,6 }` initializes the first structure in the second row of `nlist`. The remaining two structures of `nlist[1]` are set to 0. When the compiler encounters the next initializer list (`{ 7,8,9 }`), it tries to initialize `nlist[2]`. Since `nlist` has only two rows, this attempt causes an error.

In this next example, the three **int** members of `x` are initialized to 1, 2, and 3, respectively.

```
struct list
{
    int i, j, k;
```

```
    float m[2][3];
} x = {
        1,
        2,
        3,
       {4.0, 4.0, 4.0}
      };
```

In the `list` structure above, the three elements in the first row of `m` are initialized to 4.0; the elements of the remaining row of `m` are initialized to 0.0 by default.

```
union
{
    char x[2][3];
    int i, j, k;
} y = { {
            {'1'},
            {'4'}
        }
      };
```

The union variable `y`, in this example, is initialized. The first element of the union is an array, so the initializer is an aggregate initializer. The initializer list `{'1'}` assigns values to the first row of the array. Since only one value appears in the list, the element in the first column is initialized to the character `1`, and the remaining two elements in the row are initialized to the value 0 by default. Similarly, the first element of the second row of `x` is initialized to the character `4`, and the remaining two elements in the row are initialized to the value 0.

See Also

Concepts

Initialization

Initializing Strings

You can initialize an array of characters (or wide characters) with a string literal (or wide string literal). For example:

```
char code[ ] = "abc";
```

initializes `code` as a four-element array of characters. The fourth element is the null character, which terminates all string literals.

An identifier list can only be as long as the number of identifiers to be initialized. If you specify an array size that is shorter than the string, the extra characters are ignored. For example, the following declaration initializes `code` as a three-element character array:

```
char code[3] = "abcd";
```

Only the first three characters of the initializer are assigned to `code`. The character `d` and the string-terminating null character are discarded. Note that this creates an unterminated string (that is, one without a 0 value to mark its end) and generates a diagnostic message indicating this condition.

The declaration

```
char s[] = "abc", t[3] = "abc";
```

is identical to

```
char s[]  = {'a', 'b', 'c', '\0'},
     t[3] = {'a', 'b', 'c' };
```

If the string is shorter than the specified array size, the remaining elements of the array are initialized to 0.

Microsoft Specific

In Microsoft C, string literals can be up to 2048 bytes in length.

END Microsoft Specific

See Also

Concepts

Initialization

Storage of Basic Types

The following table summarizes the storage associated with each basic type.

Sizes of Fundamental Types

Type	Storage
char, **unsigned char**, **signed char**	1 byte
short, **unsigned short**	2 bytes
int, **unsigned int**	4 bytes
long, **unsigned long**	4 bytes
float	4 bytes
double	8 bytes
long double	8 bytes

The C data types fall into general categories. The "integral types" include **char**, **int**, **short**, **long**, **signed**, **unsigned**, and **enum**. The "floating types" include **float**, **double**, and **long double**. The "arithmetic types" include all floating and integral types.

See Also

Concepts

Declarations and Types

Type char

The **char** type is used to store the integer value of a member of the representable character set. That integer value is the ASCII code corresponding to the specified character.

Microsoft Specific

Character values of type **unsigned char** have a range from 0 to 0xFF hexadecimal. A **signed char** has range 0x80 to 0x7F. These ranges translate to 0 to 255 decimal, and –128 to +127 decimal, respectively. The /J compiler option changes the default from **signed** to **unsigned**.

END Microsoft Specific

See Also

Concepts

Storage of Basic Types

Type int

The size of a signed or unsigned **int** item is the standard size of an integer on a particular machine. For example, in 16-bit operating systems, the **int** type is usually 16 bits, or 2 bytes. In 32-bit operating systems, the **int** type is usually 32 bits, or 4 bytes. Thus, the **int** type is equivalent to either the **short int** or the **long int** type, and the **unsigned int** type is equivalent to either the **unsigned short** or the **unsigned long** type, depending on the target environment. The **int** types all represent signed values unless specified otherwise.

The type specifiers **int** and **unsigned int** (or simply **unsigned**) define certain features of the C language (for instance, the **enum** type). In these cases, the definitions of **int** and unsigned int for a particular implementation determine the actual storage.

Microsoft Specific

Signed integers are represented in two's-complement form. The most-significant bit holds the sign: 1 for negative, 0 for positive and zero. The range of values is given in C Integer Limits, which is taken from the LIMITS.H header file.

END Microsoft Specific

Note
The int and unsigned int type specifiers are widely used in C programs because they allow a particular machine to handle int eger values in the most efficient way for that machine. However, since the sizes of the int and unsigned int types vary, progra ms that depend on a specific int size may not be portable to other machines. To make programs more portable, you can use expressions with the sizeof operator (as discussed in The sizeof Operator) instead of hard-coded data sizes.

See Also

Concepts

Storage of Basic Types

C Sized Integer Types

Microsoft Specific

Microsoft C features support for sized integer types. You can declare 8-, 16-, 32-, or 64-bit integer variables by using the __int*n* type specifier, where *n* is the size, in bits, of the integer variable. The value of *n* can be 8, 16, 32, or 64. The following example declares one variable of each of the four types of sized integers:

```
__int8 nSmall;      // Declares 8-bit integer
__int16 nMedium;    // Declares 16-bit integer
__int32 nLarge;     // Declares 32-bit integer
__int64 nHuge;      // Declares 64-bit integer
```

The first three types of sized integers are synonyms for the ANSI types that have the same size, and are useful for writing portable code that behaves identically across multiple platforms. Note that the __int8 data type is synonymous with type char, __int16 is synonymous with type short, and __int32 is synonymous with type int. The __int64 type has no equivalent ANSI counterpart.

END Microsoft Specific

See Also

Concepts

Storage of Basic Types

Type float

Floating-point numbers use the IEEE (Institute of Electrical and Electronics Engineers) format. Single-precision values with float type have 4 bytes, consisting of a sign bit, an 8-bit excess-127 binary exponent, and a 23-bit mantissa. The mantissa represents a number between 1.0 and 2.0. Since the high-order bit of the mantissa is always 1, it is not stored in the number. This representation gives a range of approximately 3.4E–38 to 3.4E+38 for type float.

You can declare variables as float or double, depending on the needs of your application. The principal differences between the two types are the significance they can represent, the storage they require, and their range. The following table shows the relationship between significance and storage requirements.

Floating-Point Types

Type	Significant digits	Number of bytes
float	6 – 7	4
double	15 – 16	8

Floating-point variables are represented by a mantissa, which contains the value of the number, and an exponent, which contains the order of magnitude of the number.

The following table shows the number of bits allocated to the mantissa and the exponent for each floating-point type. The most significant bit of any float or double is always the sign bit. If it is 1, the number is considered negative; otherwise, it is considered a positive number.

Lengths of Exponents and Mantissas

Type	Exponent length	Mantissa length
float	8 bits	23 bits
double	11 bits	52 bits

Because exponents are stored in an unsigned form, the exponent is biased by half its possible value. For type float, the bias is 127; for type double, it is 1023. You can compute the actual exponent value by subtracting the bias value from the exponent value.

The mantissa is stored as a binary fraction greater than or equal to 1 and less than 2. For types float and double, there is an implied leading 1 in the mantissa in the most-significant bit position, so the mantissas are actually 24 and 53 bits long, respectively, even though the most-significant bit is never stored in memory.

Instead of the storage method just described, the floating-point package can store binary floating-point numbers as denormalized numbers. "Denormalized numbers" are nonzero floating-point numbers with reserved exponent values in which the most-significant bit of the mantissa is 0. By using the denormalized format, the range of a floating-point number can be extended at the cost of precision. You cannot control whether a floating-point number is represented in normalized or denormalized form; the floating-point package determines the representation. The floating-point package never uses a denormalized form unless the exponent becomes less than the minimum that can be represented in a normalized form.

The following table shows the minimum and maximum values you can store in variables of each floating-point type. The values listed in this table apply only to normalized floating-point numbers; denormalized floating-point numbers have a smaller minimum value. Note that numbers retained in 80*x*87 registers are always represented in 80-bit normalized form; numbers can only be represented in denormalized form when stored in 32-bit or 64-bit floating-point variables (variables of type float and type long).

Range of Floating-Point Types

Type	Minimum value	Maximum value
float	1.175494351 E – 38	3.402823466 E + 38
double	2.2250738585072014 E – 308	1.7976931348623158 E + 308

If precision is less of a concern than storage, consider using type float for floating-point variables. Conversely, if precision is the most important criterion, use type double.

Floating-point variables can be promoted to a type of greater significance (from type float to type double). Promotion often occurs when you perform arithmetic on floating-point variables. This arithmetic is always done in as high a degree of precision as the variable with the highest degree of precision. For example, consider the following type declarations:

```
float f_short;
double f_long;
long double f_longer;

f_short = f_short * f_long;
```

In the preceding example, the variable `f_short` is promoted to type double and multiplied by `f_long`; then the result is rounded to type float before being assigned to `f_short`.

In the following example (which uses the declarations from the preceding example), the arithmetic is done in float (32-bit) precision on the variables; the result is then promoted to type double:

```
f_longer = f_short * f_short;
```

See Also

Concepts

Storage of Basic Types

Type double

Double precision values with double type have 8 bytes. The format is similar to the float format except that it has an 11-bit excess-1023 exponent and a 52-bit mantissa, plus the implied high-order 1 bit. This format gives a range of approximately 1.7E–308 to 1.7E+308 for type double.

Microsoft Specific

The double type contains 64 bits: 1 for sign, 11 for the exponent, and 52 for the mantissa. Its range is +/–1.7E308 with at least 15 digits of precision.

END Microsoft Specific

See Also

Concepts

Storage of Basic Types

Type long double

The **long double** type is identical to the double type.

See Also

Concepts

Storage of Basic Types

Incomplete Types

An incomplete type is a type that describes an identifier but lacks information needed to determine the size of the identifier. An "incomplete type" can be:

- A structure type whose members you have not yet specified.
- A union type whose members you have not yet specified.
- An array type whose dimension you have not yet specified.

The void type is an incomplete type that cannot be completed. To complete an incomplete type, specify the missing information. The following examples show how to create and complete the incomplete types.

- To create an incomplete structure type, declare a structure type without specifying its members. In this example, the `ps` pointer points to an incomplete structure type called `student`.

```
struct student *ps;
```

- To complete an incomplete structure type, declare the same structure type later in the same scope with its members specified, as in

```
struct student
{
    int num;
}                   /* student structure now completed */
```

- To create an incomplete array type, declare an array type without specifying its repetition count. For example:

```
char a[];  /* a has incomplete type */
```

- To complete an incomplete array type, declare the same name later in the same scope with its repetition count specified, as in

```
char a[25]; /* a now has complete type */
```

See Also

Concepts

Declarations and Types

Typedef Declarations

A typedef declaration is a declaration with typedef as the storage class. The declarator becomes a new type. You can use typedef declarations to construct shorter or more meaningful names for types already defined by C or for types that you have declared. Typedef names allow you to encapsulate implementation details that may change.

A typedef declaration is interpreted in the same way as a variable or function declaration, but the identifier, instead of assuming the type specified by the declaration, becomes a synonym for the type.

Syntax

declaration:

declaration-specifiers init-declarator-list $_{\text{opt}}$ **;**

declaration-specifiers:

storage-class-specifier declaration-specifiers $_{\text{opt}}$

type-specifier declaration-specifiers $_{\text{opt}}$

type-qualifier declaration-specifiers $_{\text{opt}}$

storage-class-specifier:

typedef

type-specifier:

void

char

short

int

long

float

double

signed

unsigned

struct-or-union-specifier

enum-specifier

typedef-name

typedef-name:

identifier

Note that a typedef declaration does not create types. It creates synonyms for existing types, or names for types that could be specified in other ways. When a typedef name is used as a type specifier, it can be combined with certain type specifiers, but not others. Acceptable modifiers include **const** and **volatile**.

Typedef names share the name space with ordinary identifiers (see Name Spaces for more information). Therefore, a program can have a typedef name and a local-scope identifier by the same name. For example:

```
typedef char FlagType;

int main()
{
}

int myproc( int )
```

```
{
    int FlagType;
}
```

When declaring a local-scope identifier by the same name as a typedef, or when declaring a member of a structure or union in the same scope or in an inner scope, the type specifier must be specified. This example illustrates this constraint:

```
typedef char FlagType;
const FlagType x;
```

To reuse the `FlagType` name for an identifier, a structure member, or a union member, the type must be provided:

```
const int FlagType;  /* Type specifier required */
```

It is not sufficient to say

```
const FlagType;      /* Incomplete specification */
```

because the `FlagType` is taken to be part of the type, not an identifier that is being redeclared. This declaration is taken to be an illegal declaration like

```
int;  /* Illegal declaration */
```

You can declare any type with typedef, including pointer, function, and array types. You can declare a typedef name for a pointer to a structure or union type before you define the structure or union type, as long as the definition has the same visibility as the declaration.

Typedef names can be used to improve code readability. All three of the following declarations of `signal` specify exactly the same type, the first without making use of any typedef names.

```
typedef void fv( int ), (*pfv)( int );  /* typedef declarations */

void ( *signal( int, void (*) (int)) ) ( int );
fv *signal( int, fv * );   /* Uses typedef type */
pfv signal( int, pfv );    /* Uses typedef type */
```

Examples

The following examples illustrate typedef declarations:

```
typedef int WHOLE; /* Declares WHOLE to be a synonym for int */
```

Note that `WHOLE` could now be used in a variable declaration such as `WHOLE i;` or `const WHOLE i;`. However, the declaration `long WHOLE i;` would be illegal.

```
typedef struct club
{
    char name[30];
    int size, year;
} GROUP;
```

This statement declares `GROUP` as a structure type with three members. Since a structure tag, `club`, is also specified, either the typedef name (`GROUP`) or the structure tag can be used in declarations. You must use the struct keyword with the tag, and you cannot use the struct keyword with the typedef name.

```
typedef GROUP *PG; /* Uses the previous typedef name
                      to declare a pointer            */
```

The type `PG` is declared as a pointer to the `GROUP` type, which in turn is defined as a structure type.

```
typedef void DRAWF( int, int );
```

This example provides the type `DRAWF` for a function returning no value and taking two int arguments. This means, for example, that the declaration

```
DRAWF box;
```

is equivalent to the declaration

```
void box( int, int );
```

See Also

Reference

typedef Specifier

C Extended Storage-Class Attributes

Microsoft Specific

More up-to-date information on this topic can be found under __declspec (C++ Reference).

Extended attribute syntax simplifies and standardizes the Microsoft-specific extensions to the C language. The storage-class attributes that use extended attribute syntax include thread, naked, dllimport, and dllexport.

The extended attribute syntax for specifying storage-class information uses the __declspec keyword, which specifies that an instance of a given type is to be stored with a Microsoft-specific storage-class attribute (thread, naked, dllimport, or dllexport). Examples of other storage-class modifiers include the static and extern keywords. However, these keywords are part of the ANSI C standard and as such are not covered by extended attribute syntax.

Syntax
storage-class-specifier:

__declspec (*extended-decl-modifier-seq*) /* Microsoft Specific */

extended-decl-modifier-seq:

extended-decl-modifier $_{\text{opt}}$

extended-decl-modifier-seq extended-decl-modifier

extended-decl-modifier:

thread

naked

dllimport

dllexport

White space separates the declaration modifiers. Note that *extended-decl-modifier-seq* can be empty; in this case, __declspec has no effect.

The thread, naked, dllimport, and dllexport storage-class attributes are a property only of the declaration of the data or function to which they are applied; they do not redefine the type attributes of the function itself. The thread attribute affects data only. The naked attribute affects functions only. The dllimport and dllexport attributes affect functions and data.

END Microsoft Specific

See Also
Concepts
Declarations and Types

DLL Import and Export

Microsoft Specific

The dllimport and dllexport storage-class modifiers are Microsoft-specific extensions to the C language. These modifiers define the DLL's interface to its client (the executable file or another DLL). For specific information about using these modifiers, see dllexport, dllimport.

END Microsoft Specific

See Also

Reference

C Extended Storage-Class Attributes

Naked (C)

Microsoft Specific

The naked storage-class attribute is a Microsoft-specific extension to the C language. The compiler generates code without prolog and epilog code for functions declared with the naked storage-class attribute. Naked functions are useful when you need to write your own prolog/epilog code sequences using inline assembler code. Naked functions are useful for writing virtual device drivers.

For specific information about using the naked attribute, see Naked Functions.

END Microsoft Specific

See Also
Reference
C Extended Storage-Class Attributes

Thread Local Storage

Microsoft Specific

Thread Local Storage (TLS) is the mechanism by which each thread in a given multithreaded process allocates storage for thread-specific data. In standard multithreaded programs, data is shared among all threads of a given process, whereas thread local storage is the mechanism for allocating per-thread data. For a complete discussion of threads, see Processes and Threads in the Platform SDK.

The Microsoft C language includes the extended storage-class attribute, thread, which is used with the __declspec keyword to declare a thread local variable. For example, the following code declares an integer thread local variable and initializes it with a value:

```
__declspec( thread ) int tls_i = 1;
```

These guidelines must be observed when you are declaring statically bound thread local variables:

- The use of **__declspec(thread)** may interfere with delay loading of DLL imports**.**
- You can apply the thread attribute only to data declarations and definitions. It cannot be used on function declarations or definitions. For example, the following code generates a compiler error:

  ```
  #define Thread   __declspec( thread )
  Thread void func();      /* Error */
  ```

- You can specify the thread attribute only on data items with static storage duration. This includes global data (both static and extern) and local static data. You cannot declare automatic data with the thread attribute. For example, the following code generates compiler errors:

  ```
  #define Thread   __declspec( thread )
  void func1()
  {
      Thread int tls_i;            /* Error */
  }

  int func2( Thread int tls_i )    /* Error */
  {
     return tls_i;
  }
  ```

- You must use the thread attribute for the declaration and the definition of thread local data, regardless of whether the declaration and definition occur in the same file or separate files. For example, the following code generates an error:

  ```
  #define Thread   __declspec( thread )
  extern int tls_i;     /* This generates an error, because the   */
  int Thread tls_i;     /* declaration and the definition differ. */
  ```

- You cannot use the thread attribute as a type modifier. For example, the following code generates a compiler error:

  ```
  char *ch __declspec( thread );      /* Error */
  ```

- The address of a thread local variable is not considered constant, and any expression involving such an address is not considered a constant expression. This means that you cannot use the address of a thread local variable as an initializer for a pointer. For example, the compiler flags the following code as an error:

  ```
  #define Thread   __declspec( thread )
  ```

```
Thread int tls_i;
int *p = &tls_i;      /* Error */
```

- C permits initialization of a variable with an expression involving a reference to itself, but only for objects of nonstatic extent. For example:

```
#define Thread   __declspec( thread )
Thread int tls_i = tls_i;             /* Error */
int j = j;                            /* Error */
Thread int tls_i = sizeof( tls_i )    /* Okay  */
```

  Note that a sizeof expression that includes the variable being initialized does not constitute a reference to itself and is allowed.

- The use of **__declspec(thread)** may interfere with delay loading of DLL imports**.**

For more information about using the thread attribute, see Multithreading Topics.

END Microsoft Specific

See Also

Reference

C Extended Storage-Class Attributes

Expressions and Assignments

This section describes how to form expressions and to assign values in the C language. Constants, identifiers, strings, and function calls are all operands that are manipulated in expressions. The C language has all the usual language operators. This section covers those operators as well as operators that are unique to C or Microsoft C. The topics discussed include:

- L-value and r-value expressions
- Constant expressions
- Side effects
- Sequence points
- Operators
- Operator precedence
- Type conversions
- Type casts

See Also

Reference

C Language Reference

Operands and Expressions

An "operand" is an entity on which an operator acts. An "expression" is a sequence of operators and operands that performs any combination of these actions:

- Computes a value
- Designates an object or function
- Generates side effects

Operands in C include constants, identifiers, strings, function calls, subscript expressions, member-selection expressions, and complex expressions formed by combining operands with operators or by enclosing operands in parentheses. The syntax for these operands is given in Primary Expressions.

See Also

Concepts

Expressions and Assignments

C Primary Expressions

The operands in expressions are called "primary expressions."

Syntax

primary-expression:

identifier

constant

string-literal

(*expression* **)**

expression:

assignment-expression

expression **,** *assignment-expression*

See Also

Concepts

Operands and Expressions

Identifiers in Primary Expressions

Identifiers can have integral, **float**, **enum**, **struct**, **union**, array, pointer, or function type. An identifier is a primary expression provided it has been declared as designating an object (in which case it is an l-value) or as a function (in which case it is a function designator). See L-Value and R-Value Expressions for a definition of l-value.

The pointer value represented by an array identifier is not a variable, so an array identifier cannot form the left-hand operand of an assignment operation and therefore is not a modifiable l-value.

An identifier declared as a function represents a pointer whose value is the address of the function. The pointer addresses a function returning a value of a specified type. Thus, function identifiers also cannot be l-values in assignment operations. For more information, see Identifiers.

See Also

Reference

C Primary Expressions

Constants in Primary Expressions

A constant operand has the value and type of the constant value it represents. A character constant has **int** type. An integer constant has **int**, **long**, **unsigned int**, or **unsigned long** type, depending on the integer's size and on the way the value is specified. See Constants for more information.

See Also

Reference

C Primary Expressions

String Literals in Primary Expressions

A "string literal" is a character, wide character, or sequence of adjacent characters enclosed in double quotation marks. Since they are not variables, neither string literals nor any of their elements can be the left-hand operand in an assignment operation. The type of a string literal is an array of **char** (or an array of **wchar_t** for wide-string literals). Arrays in expressions are converted to pointers. See String Literals for more information about strings.

See Also

Reference

C Primary Expressions

Expressions in Parentheses

You can enclose any operand in parentheses without changing the type or value of the enclosed expression. For example, in the expression:

```
( 10 + 5 ) / 5
```

the parentheses around `10 + 5` mean that the value of `10 + 5` is evaluated first and it becomes the left operand of the division (**/**) operator. The result of `( 10 + 5 ) / 5` is 3. Without the parentheses, `10 + 5 / 5` would evaluate to 11.

Although parentheses affect the way operands are grouped in an expression, they cannot guarantee a particular order of evaluation in all cases. For example, neither the parentheses nor the left-to-right grouping of the following expression guarantees what the value of `i` will be in either of the subexpressions:

```
( i++ +1 ) * ( 2 + i )
```

The compiler is free to evaluate the two sides of the multiplication in any order. If the initial value of `i` is zero, the whole expression could be evaluated as either of these two statements:

```
( 0 + 1 + 1 ) * ( 2 + 1 )
( 0 + 1 + 1 ) * ( 2 + 0 )
```

Exceptions resulting from side effects are discussed in Side Effects.

See Also

Reference

C Primary Expressions

L-Value and R-Value Expressions

Expressions that refer to memory locations are called "l-value" expressions. An l-value represents a storage region's "locator" value, or a "left" value, implying that it can appear on the left of the equal sign (**=**). L-values are often identifiers.

Expressions referring to modifiable locations are called "modifiable l-values." A modifiable l-value cannot have an array type, an incomplete type, or a type with the **const** attribute. For structures and unions to be modifiable l-values, they must not have any members with the **const** attribute. The name of the identifier denotes a storage location, while the value of the variable is the value stored at that location.

An identifier is a modifiable l-value if it refers to a memory location and if its type is arithmetic, structure, union, or pointer. For example, if `ptr` is a pointer to a storage region, then `*ptr` is a modifiable l-value that designates the storage region to which `ptr` points.

Any of the following C expressions can be l-value expressions:

- An identifier of integral, floating, pointer, structure, or union type
- A subscript (**[ ]**) expression that does not evaluate to an array
- A member-selection expression (**–>** or **.**)
- A unary-indirection (*****) expression that does not refer to an array
- An l-value expression in parentheses
- A **const** object (a nonmodifiable l-value)

The term "r-value" is sometimes used to describe the value of an expression and to distinguish it from an l-value. All l-values are r-values but not all r-values are l-values.

Microsoft Specific

Microsoft C includes an extension to the ANSI C standard that allows casts of l-values to be used as l-values, as long as the size of the object is not lengthened through the cast. (See Type-Cast Conversions for more information.) The following example illustrates this feature:

```
char *p ;
short  i;
long l;

(long *) p = &l ;       /* Legal cast   */
(long) i = l ;          /* Illegal cast */
```

The default for Microsoft C is that the Microsoft extensions are enabled. Use the /Za compiler option to disable these extensions.

END Microsoft Specific

See Also

Concepts

Operands and Expressions

C Constant Expressions

A constant expression is evaluated at compile time, not run time, and can be used in any place that a constant can be used. The constant expression must evaluate to a constant that is in the range of representable values for that type. The operands of a constant expression can be integer constants, character constants, floating-point constants, enumeration constants, type casts, **sizeof** expressions, and other constant expressions.

Syntax

constant-expression:

  conditional-expression

conditional-expression:

  logical-OR-expression

  logical-OR-expression **?** *expression* **:** *conditional-expression*

expression:

  assignment-expression

  expression **,** *assignment-expression*

assignment-expression:

  conditional-expression

  unary-expression assignment-operator assignment-expression

assignment-operator: one of

  = *= /= %= += –= <<= >>= &= ^= |=

The nonterminals for struct declarator, enumerator, direct declarator, direct-abstract declarator, and labeled statement contain the *constant-expression* nonterminal.

An integral constant expression must be used to specify the size of a bit-field member of a structure, the value of an enumeration constant, the size of an array, or the value of a **case** constant.

Constant expressions used in preprocessor directives are subject to additional restrictions. Consequently, they are known as "restricted constant expressions." A restricted constant expression cannot contain **sizeof** expressions, enumeration constants, type casts to any type, or floating-type constants. It can, however, contain the special constant expression **defined (***identifier***)**.

See Also

Concepts

Operands and Expressions

Expression Evaluation (C)

Expressions involving assignment, unary increment, unary decrement, or calling a function may have consequences incidental to their evaluation (side effects). When a "sequence point" is reached, everything preceding the sequence point, including any side effects, is guaranteed to have been evaluated before evaluation begins on anything following the sequence point.

"Side effects" are changes caused by the evaluation of an expression. Side effects occur whenever the value of a variable is changed by an expression evaluation. All assignment operations have side effects. Function calls can also have side effects if they change the value of an externally visible item, either by direct assignment or by indirect assignment through a pointer.

See Also

Concepts

Operands and Expressions

Side Effects

The order of evaluation of expressions is defined by the specific implementation, except when the language guarantees a particular order of evaluation (as outlined in Precedence and Order of Evaluation). For example, side effects occur in the following function calls:

```
add( i + 1, i = j + 2 );
myproc( getc(), getc() );
```

The arguments of a function call can be evaluated in any order. The expression `i + 1` may be evaluated before `i = j + 2`, or `i = j + 2` may be evaluated before `i + 1`. The result is different in each case. Likewise, it is not possible to guarantee what characters are actually passed to the `myproc`. Since unary increment and decrement operations involve assignments, such operations can cause side effects, as shown in the following example:

```
x[i] = i++;
```

In this example, the value of `x` that is modified is unpredictable. The value of the subscript could be either the new or the old value of `i`. The result can vary under different compilers or different optimization levels.

Since C does not define the order of evaluation of side effects, both evaluation methods discussed above are correct and either may be implemented. To make sure that your code is portable and clear, avoid statements that depend on a particular order of evaluation for side effects.

See Also

Concepts

Expression Evaluation (C)

C Sequence Points

Between consecutive "sequence points" an object's value can be modified only once by an expression. The C language defines the following sequence points:

- Left operand of the logical-AND operator (**&&**). The left operand of the logical-AND operator is completely evaluated and all side effects complete before continuing. If the left operand evaluates to false (0), the other operand is not evaluated.
- Left operand of the logical-OR operator (**||**). The left operand of the logical-OR operator is completely evaluated and all side effects complete before continuing. If the left operand evaluates to true (nonzero), the other operand is not evaluated.
- Left operand of the comma operator. The left operand of the comma operator is completely evaluated and all side effects complete before continuing. Both operands of the comma operator are always evaluated. Note that the comma operator in a function call does not guarantee an order of evaluation.
- Function-call operator. All arguments to a function are evaluated and all side effects complete before entry to the function. No order of evaluation among the arguments is specified.
- First operand of the conditional operator. The first operand of the conditional operator is completely evaluated and all side effects complete before continuing.
- The end of a full initialization expression (that is, an expression that is not part of another expression such as the end of an initialization in a declaration statement).
- The expression in an expression statement. Expression statements consist of an optional expression followed by a semicolon (**;**). The expression is evaluated for its side effects and there is a sequence point following this evaluation.
- The controlling expression in a selection (**if** or **switch**) statement. The expression is completely evaluated and all side effects complete before the code dependent on the selection is executed.
- The controlling expression of a **while** or **do** statement. The expression is completely evaluated and all side effects complete before any statements in the next iteration of the **while** or **do** loop are executed.
- Each of the three expressions of a **for** statement. The expressions are completely evaluated and all side effects complete before any statements in the next iteration of the **for** loop are executed.
- The expression in a **return** statement. The expression is completely evaluated and all side effects complete before control returns to the calling function.

See Also

Concepts

Expression Evaluation (C)

C Operators

The C operators are a subset of the C++ operators.

There are three types of operators. A unary expression consists of either a unary operator prepended to an operand, or the **sizeof** keyword followed by an expression. The expression can be either the name of a variable or a cast expression. If the expression is a cast expression, it must be enclosed in parentheses. A binary expression consists of two operands joined by a binary operator. A ternary expression consists of three operands joined by the conditional-expression operator.

C includes the following unary operators:

Symbol	Name
– ~ !	Negation and complement operators
*** &**	Indirection and address-of operators
sizeof	Size operator
+	Unary plus operator
++ —	Unary increment and decrement operators

Binary operators associate from left to right. C provides the following binary operators:

Symbol	Name
*** / %**	Multiplicative operators
+ –	Additive operators
<< >>	Shift operators
< > <= >= == !=	Relational operators
& | ^	Bitwise operators
&& ||	Logical operators
,	Sequential-evaluation operator

The base operator (**:>**), supported by previous versions of the Microsoft 16-bit C compiler, is described in C Language Syntax Summary.

The conditional-expression operator has lower precedence than binary expressions and differs from them in being right associative.

Expressions with operators also include assignment expressions, which use unary or binary assignment operators. The unary assignment operators are the increment (**++**) and decrement (**—**) operators; the binary assignment operators are the simple-assignment operator (**=**) and the compound-assignment operators. Each compound-assignment operator is a combination of another binary operator with the simple-assignment operator.

See Also

Concepts

Expressions and Assignments

Precedence and Order of Evaluation

The precedence and associativity of C operators affect the grouping and evaluation of operands in expressions. An operator's precedence is meaningful only if other operators with higher or lower precedence are present. Expressions with higher-precedence operators are evaluated first. Precedence can also be described by the word "binding." Operators with a higher precedence are said to have tighter binding.

The following table summarizes the precedence and associativity (the order in which the operands are evaluated) of C operators, listing them in order of precedence from highest to lowest. Where several operators appear together, they have equal precedence and are evaluated according to their associativity. The operators in the table are described in the sections beginning with Postfix Operators. The rest of this section gives general information about precedence and associativity.

Precedence and Associativity of C Operators

Symbol1	Type of Operation	Associativity
[ ] ( ) . –> postfix **++** and postfix **—**	Expression	Left to right
prefix **++** and prefix **— sizeof & * + – ~ !**	Unary	Right to left
typecasts	Unary	Right to left
*** / %**	Multiplicative	Left to right
+ –	Additive	Left to right
<< >>	Bitwise shift	Left to right
< > <= >=	Relational	Left to right
== !=	Equality	Left to right
&	Bitwise-AND	Left to right
^	Bitwise-exclusive-OR	Left to right
|	Bitwise-inclusive-OR	Left to right
&&	Logical-AND	Left to right
||	Logical-OR	Left to right
? :	Conditional-expression	Right to left
= *= /= %= **+= –= <<= >>=&=** **^= |=**	Simple and compound assignment2	Right to left
,	Sequential evaluation	Left to right

1. Operators are listed in descending order of precedence. If several operators appear on the same line or in a group, they have equal precedence.

2. All simple and compound-assignment operators have equal precedence.

An expression can contain several operators with equal precedence. When several such operators appear at the same level in an expression, evaluation proceeds according to the associativity of the operator, either from right to left or from left to right.

The direction of evaluation does not affect the results of expressions that include more than one multiplication (*****), addition (**+**), or binary-bitwise (**& | ^**) operator at the same level. Order of operations is not defined by the language. The compiler is free to evaluate such expressions in any order, if the compiler can guarantee a consistent result.

Only the sequential-evaluation (**,**), logical-AND (**&&**), logical-OR (**||**), conditional-expression (**? :**), and function-call operators constitute sequence points and therefore guarantee a particular order of evaluation for their operands. The function-call operator is the set of parentheses following the function identifier. The sequential-evaluation operator (**,**) is guaranteed to evaluate its operands from left to right. (Note that the comma operator in a function call is not the same as the sequential-evaluation operator and does not provide any such guarantee.) For more information, see Sequence Points.

Logical operators also guarantee evaluation of their operands from left to right. However, they evaluate the smallest number of operands needed to determine the result of the expression. This is called "short-circuit" evaluation. Thus, some operands of the expression may not be evaluated. For example, in the expression

```
x && y++
```

the second operand, `y++`, is evaluated only if `x` is true (nonzero). Thus, `y` is not incremented if `x` is false (0).

Examples

The following list shows how the compiler automatically binds several sample expressions:

Expression	Automatic Binding
`a & b || c`	`(a & b) || c`
`a = b || c`	`a = (b || c)`
`q && r || s--`	`(q && r) || s--`

In the first expression, the bitwise-AND operator (`&`) has higher precedence than the logical-OR operator (`||`), so `a & b` forms the first operand of the logical-OR operation.

In the second expression, the logical-OR operator (`||`) has higher precedence than the simple-assignment operator (`=`), so `b || c` is grouped as the right-hand operand in the assignment. Note that the value assigned to `a` is either 0 or 1.

The third expression shows a correctly formed expression that may produce an unexpected result. The logical-AND operator (`&&`) has higher precedence than the logical-OR operator (`||`), so `q && r` is grouped as an operand. Since the logical operators guarantee evaluation of operands from left to right, `q && r` is evaluated before `s--`. However, if `q && r` evaluates to a nonzero value, `s--` is not evaluated, and `s` is not decremented. If not decrementing `s` would cause a problem in your program, `s--` should appear as the first operand of the expression, or `s` should be decremented in a separate operation.

The following expression is illegal and produces a diagnostic message at compile time:

Illegal Expression	Default Grouping
`p == 0 ? p += 1: p += 2`	`( p == 0 ? p += 1 : p ) += 2`

In this expression, the equality operator (`==`) has the highest precedence, so `p == 0` is grouped as an operand. The conditional-expression operator (`? :`) has the next-highest precedence. Its first operand is `p == 0`, and its second operand is `p += 1`. However, the last operand of the conditional-expression operator is considered to be `p` rather than `p += 2`, since this occurrence of `p` binds more closely to the conditional-expression operator than it does to the compound-assignment operator. A syntax error occurs because `+= 2` does not have a left-hand operand. You should use parentheses to prevent errors of this kind and produce more readable code. For example, you could use parentheses as shown below to correct and clarify the preceding example:

```
( p == 0 ) ? ( p += 1 ) : ( p += 2 )
```

See Also

Reference

C Operators

Usual Arithmetic Conversions

Most C operators perform type conversions to bring the operands of an expression to a common type or to extend short values to the integer size used in machine operations. The conversions performed by C operators depend on the specific operator and the type of the operand or operands. However, many operators perform similar conversions on operands of integral and floating types. These conversions are known as "arithmetic conversions." Conversion of an operand value to a compatible type causes no change to its value.

The arithmetic conversions summarized below are called "usual arithmetic conversions." These steps are applied only for binary operators that expect arithmetic type. The purpose is to yield a common type which is also the type of the result. To determine which conversions actually take place, the compiler applies the following algorithm to binary operations in the expression. The steps below are not a precedence order.

1. If either operand is of type **long double**, the other operand is converted to type **long double**.
2. If the above condition is not met and either operand is of type **double**, the other operand is converted to type **double**.
3. If the above two conditions are not met and either operand is of type **float**, the other operand is converted to type **float**.
4. If the above three conditions are not met (none of the operands are of floating types), then integral conversions are performed on the operands as follows:
   - If either operand is of type **unsigned long**, the other operand is converted to type **unsigned long**.
   - If the above condition is not met and either operand is of type **long** and the other of type **unsigned int**, both operands are converted to type **unsigned long**.
   - If the above two conditions are not met, and either operand is of type **long**, the other operand is converted to type **long**.
   - If the above three conditions are not met, and either operand is of type **unsigned int**, the other operand is converted to type **unsigned int**.
   - If none of the above conditions are met, both operands are converted to type **int**.

The following code illustrates these conversion rules:

```
float   fVal;
double  dVal;
int   iVal;
unsigned long ulVal;

dVal = iVal * ulVal; /* iVal converted to unsigned long
                      * Uses step 4.
                      * Result of multiplication converted to double
                      */
dVal = ulVal + fVal; /* ulVal converted to float
                      * Uses step 3.
                      * Result of addition converted to double
                      */
```

See Also

Reference

C Operators

Postfix Operators

The postfix operators have the highest precedence (the tightest binding) in expression evaluation.

Syntax

postfix-expression:

primary-expression

postfix-expression **[** *expression* **]**

postfix-expression **(** *argument-expression-list* $_{\text{opt}}$ **)**

postfix-expression **.** *identifier*

postfix-expression **–>** *identifier*

postfix-expression **++**

postfix-expression **—**

Operators in this precedence level are the array subscripts, function calls, structure and union members, and postfix increment and decrement operators.

See Also

Reference

C Operators

One-Dimensional Arrays

A postfix expression followed by an expression in square brackets (**[ ]**) is a subscripted representation of an element of an array object. A subscript expression represents the value at the address that is *expression* positions beyond *postfix-expression* when expressed as

```
postfix-expression [ expression ]
```

Usually, the value represented by *postfix-expression* is a pointer value, such as an array identifier, and *expression* is an integral value. However, all that is required syntactically is that one of the expressions be of pointer type and the other be of integral type. Thus the integral value could be in the *postfix-expression* position and the pointer value could be in the brackets in the *expression*, or "subscript," position. For example, this code is legal:

```
// one_dimensional_arrays.c
int sum, *ptr, a[10];
int main() {
   ptr = a;
   sum = 4[ptr];
}
```

Subscript expressions are generally used to refer to array elements, but you can apply a subscript to any pointer. Whatever the order of values, *expression* must be enclosed in brackets (**[ ]**).

The subscript expression is evaluated by adding the integral value to the pointer value, then applying the indirection operator (*****) to the result. (See Indirection and Address-of Operators for a discussion of the indirection operator.) In effect, for a one-dimensional array, the following four expressions are equivalent, assuming that `a` is a pointer and `b` is an integer:

```
a[b]
*(a + b)
*(b + a)
b[a]
```

According to the conversion rules for the addition operator (given in Additive Operators), the integral value is converted to an address offset by multiplying it by the length of the type addressed by the pointer.

For example, suppose the identifier `line` refers to an array of **int** values. The following procedure is used to evaluate the subscript expression `line[ i ]`:

1. The integer value `i` is multiplied by the number of bytes defined as the length of an **int** item. The converted value of `i` represents `i` **int** positions.
2. This converted value is added to the original pointer value (`line`) to yield an address that is offset `i` **int** positions from `line`.
3. The indirection operator is applied to the new address. The result is the value of the array element at that position (intuitively, `line [ i ]`).

The subscript expression `line[0]` represents the value of the first element of line, since the offset from the address represented by `line` is 0. Similarly, an expression such as `line[5]` refers to the element offset five positions from line, or the sixth element of the array.

See Also

Reference

Subscript Operator: []

Multidimensional Arrays (C)

A subscript expression can also have multiple subscripts, as follows:

```
expression1 [expression2] [expression3]...
```

Subscript expressions associate from left to right. The leftmost subscript expression, *expression1***[***expression2***]**, is evaluated first. The address that results from adding *expression1* and *expression2* forms a pointer expression; then *expression3* is added to this pointer expression to form a new pointer expression, and so on until the last subscript expression has been added. The indirection operator (*) is applied after the last subscripted expression is evaluated, unless the final pointer value addresses an array type (see examples below).

Expressions with multiple subscripts refer to elements of "multidimensional arrays." A multidimensional array is an array whose elements are arrays. For example, the first element of a three-dimensional array is an array with two dimensions.

Examples

For the following examples, an array named `prop` is declared with three elements, each of which is a 4-by-6 array of **int** values.

```
int prop[3][4][6];
int i, *ip, (*ipp)[6];
```

A reference to the `prop` array looks like this:

```
i = prop[0][0][1];
```

The example above shows how to refer to the second individual **int** element of `prop`. Arrays are stored by row, so the last subscript varies most quickly; the expression `prop[0][0][2]` refers to the next (third) element of the array, and so on.

```
i = prop[2][1][3];
```

This statement is a more complex reference to an individual element of `prop`. The expression is evaluated as follows:

1. The first subscript, `2`, is multiplied by the size of a 4-by-6 **int** array and added to the pointer value `prop`. The result points to the third 4-by-6 array of `prop`.
2. The second subscript, `1`, is multiplied by the size of the 6-element **int** array and added to the address represented by `prop[2]`.
3. Each element of the 6-element array is an **int** value, so the final subscript, `3`, is multiplied by the size of an **int** before it is added to `prop[2][1]`. The resulting pointer addresses the fourth element of the 6-element array.
4. The indirection operator is applied to the pointer value. The result is the **int** element at that address.

These next two examples show cases where the indirection operator is not applied.

```
ip = prop[2][1];

ipp = prop[2];
```

In the first of these statements, the expression `prop[2][1]` is a valid reference to the three-dimensional array `prop`; it refers to a 6-element array (declared above). Since the pointer value addresses an array, the indirection operator is not applied.

Similarly, the result of the expression `prop[2]` in the second statement `ipp = prop[2];` is a pointer value addressing a two-dimensional array.

See Also

Reference

Subscript Operator: []

Function Call (C)

A "function call" is an expression that includes the name of the function being called or the value of a function pointer and, optionally, the arguments being passed to the function.

Syntax

postfix-expression:

postfix-expression **(** *argument-expression-list* $_{\text{opt}}$ **)**

argument-expression-list:

assignment-expression

argument-expression-list **,** *assignment-expression*

The *postfix-expression* must evaluate to a function address (for example, a function identifier or the value of a function pointer), and *argument-expression-list* is a list of expressions (separated by commas) whose values (the "arguments") are passed to the function. The *argument-expression-list* argument can be empty.

A function-call expression has the value and type of the function's return value. A function cannot return an object of array type. If the function's return type is **void** (that is, the function has been declared never to return a value), the function-call expression also has **void** type. (See Function Calls for more information.)

See Also

Reference

Function Call Operator: ()

Structure and Union Members

A "member-selection expression" refers to members of structures and unions. Such an expression has the value and type of the selected member.

```
      postfix-expression . identifier
postfix-expression -> identifier
```

This list describes the two forms of the member-selection expressions:

1. In the first form, *postfix-expression* represents a value of **struct** or **union** type, and *identifier* names a member of the specified structure or union. The value of the operation is that of *identifier* and is an l-value if *postfix-expression* is an l-value. See L-Value and R-Value Expressions for more information.
2. In the second form, *postfix-expression* represents a pointer to a structure or union, and *identifier* names a member of the specified structure or union. The value is that of *identifier* and is an l-value.

The two forms of member-selection expressions have similar effects.

In fact, an expression involving the member-selection operator (**->**) is a shorthand version of an expression using the period (**.**) if the expression before the period consists of the indirection operator (*****) applied to a pointer value. Therefore,

```
      expression -> identifier
```

is equivalent to

```
      (*
      expression
      ) . identifier
```

when *expression* is a pointer value.

Examples

The following examples refer to this structure declaration. For information about the indirection operator (*****) used in these examples, see Indirection and Address-of Operators.

```
struct pair
{
    int a;
    int b;
    struct pair *sp;
} item, list[10];
```

A member-selection expression for the `item` structure looks like this:

```
item.sp = &item;
```

In the example above, the address of the `item` structure is assigned to the `sp` member of the structure. This means that `item` contains a pointer to itself.

```
(item.sp)->a = 24;
```

In this example, the pointer expression `item.sp` is used with the member-selection operator (**->**) to assign a value to the member `a`.

```
list[8].b = 12;
```

This statement shows how to select an individual structure member from an array of structures.

See Also

Reference

Member Access Operators: . and ->

C Postfix Increment and Decrement Operators

Operands of the postfix increment and decrement operators are scalar types that are modifiable l-values.

Syntax
postfix-expression:

postfix-expression **++**

postfix-expression **—**

The result of the postfix increment or decrement operation is the value of the operand. After the result is obtained, the value of the operand is incremented (or decremented). The following code illustrates the postfix increment operator.

```
if( var++ > 0 )
    *p++ = *q++;
```

In this example, the variable `var` is compared to 0, then incremented. If `var` was positive before being incremented, the next statement is executed. First, the value of the object pointed to by `q` is assigned to the object pointed to by `p`. Then, `q` and `p` are incremented.

See Also
Reference
Postfix Increment and Decrement Operators: ++ and --

C Unary Operators

Unary operators appear before their operand and associate from right to left.

Syntax

unary-expression:

postfix-expression

++ *unary-expression*

-- *unary-expression*

unary-operator cast-expression

sizeof *unary-expression*

sizeof (*type-name* **)**

unary-operator: one of

& * + - ~ !

See Also

Reference

C Operators

Prefix Increment and Decrement Operators

The unary operators (**++** and **—**) are called "prefix" increment or decrement operators when the increment or decrement operators appear before the operand. Postfix increment and decrement has higher precedence than prefix increment and decrement. The operand must have integral, floating, or pointer type and must be a modifiable l-value expression (an expression without the **const** attribute). The result is an l-value.

When the operator appears before its operand, the operand is incremented or decremented and its new value is the result of the expression.

An operand of integral or floating type is incremented or decremented by the integer value 1. The type of the result is the same as the operand type. An operand of pointer type is incremented or decremented by the size of the object it addresses. An incremented pointer points to the next object; a decremented pointer points to the previous object.

Example

This example illustrates the unary prefix decrement operator:

```
if( line[--i] != '\n' )
    return;
```

In this example, the variable `i` is decremented before it is used as a subscript to `line`.

See Also

Reference

C Unary Operators

Indirection and Address-of Operators

The indirection operator (*****) accesses a value indirectly, through a pointer. The operand must be a pointer value. The result of the operation is the value addressed by the operand; that is, the value at the address to which its operand points. The type of the result is the type that the operand addresses.

If the operand points to a function, the result is a function designator. If it points to a storage location, the result is an l-value designating the storage location.

If the pointer value is invalid, the result is undefined. The following list includes some of the most common conditions that invalidate a pointer value.

- The pointer is a null pointer.
- The pointer specifies the address of a local item that is not visible at the time of the reference.
- The pointer specifies an address that is inappropriately aligned for the type of the object pointed to.
- The pointer specifies an address not used by the executing program.

The address-of operator (**&**) gives the address of its operand. The operand of the address-of operator can be either a function designator or an l-value that designates an object that is not a bit field and is not declared with the **register** storage-class specifier.

The result of the address operation is a pointer to the operand. The type addressed by the pointer is the type of the operand.

The address-of operator can only be applied to variables with fundamental, structure, or union types that are declared at the file-scope level, or to subscripted array references. In these expressions, a constant expression that does not include the address-of operator can be added to or subtracted from the address expression.

Examples

The following examples use these declarations:

```
int *pa, x;
int a[20];
double d;
```

This statement uses the address-of operator:

```
pa = &a[5];
```

The address-of operator (**&**) takes the address of the sixth element of the array `a`. The result is stored in the pointer variable `pa`.

```
x = *pa;
```

The indirection operator (*****) is used in this example to access the **int** value at the address stored in `pa`. The value is assigned to the integer variable `x`.

```
if( x == *&x )
    printf( "True\n" );
```

This example prints the word `True`, demonstrating that the result of applying the indirection operator to the address of `x` is the same as `x`.

```
int roundup( void );     /* Function declaration */

int  *proundup  = roundup;
int  *pround  = &roundup;
```

Once the function `roundup` is declared, two pointers to `roundup` are declared and initialized. The first pointer, `proundup`, is

initialized using only the name of the function, while the second, `pround`, uses the address-of operator in the initialization. The initializations are equivalent.

See Also

Reference

Indirection Operator: *
Address-of Operator: &

Unary Arithmetic Operators

The C unary plus, arithmetic-negation, complement, and logical-negation operators are discussed in the following list:

Operator	Description
+	The unary plus operator preceding an expression in parentheses forces the grouping of the enclosed operations. It is used with expressions involving more than one associative or commutative binary operator. The operand must have arithmetic type. The result is the value of the operand. An integral operand undergoes integral promotion. The type of the result is the type of the promoted operand.
–	The arithmetic-negation operator produces the negative (two's complement) of its operand. The operand must be an integral or floating value. This operator performs the usual arithmetic conversions.
~	The bitwise-complement (or bitwise-NOT) operator produces the bitwise complement of its operand. The operand must be of integral type. This operator performs usual arithmetic conversions; the result has the type of the operand after conversion.
!	The logical-negation (logical-NOT) operator produces the value 0 if its operand is true (nonzero) and the value 1 if its operand is false (0). The result has **int** type. The operand must be an integral, floating, or pointer value.

Unary arithmetic operations on pointers are illegal.

Examples

The following examples illustrate the unary arithmetic operators:

```
short x = 987;
    x = -x;
```

In the example above, the new value of `x` is the negative of 987, or –987.

```
unsigned short y = 0xAAAA;
    y = ~y;
```

In this example, the new value assigned to `y` is the one's complement of the unsigned value 0xAAAA, or 0x5555.

```
if( !(x < y) )
```

If `x` is greater than or equal to `y`, the result of the expression is 1 (true). If `x` is less than `y`, the result is 0 (false).

See Also

Reference

Expressions with Unary Operators

The sizeof Operator

The **sizeof** operator gives the amount of storage, in bytes, required to store an object of the type of the operand. This operator allows you to avoid specifying machine-dependent data sizes in your programs.

```
sizeof unary-expression
sizeof ( type-name )
```

Remarks

The operand is either an identifier that is a *unary-expression*, or a type-cast expression (that is, a type specifier enclosed in parentheses). The *unary-expression* cannot represent a bit-field object, an incomplete type, or a function designator. The result is an unsigned integral constant. The standard header STDDEF.H defines this type as **size_t**.

When you apply the **sizeof** operator to an array identifier, the result is the size of the entire array rather than the size of the pointer represented by the array identifier.

When you apply the **sizeof** operator to a structure or union type name, or to an identifier of structure or union type, the result is the number of bytes in the structure or union, including internal and trailing padding. This size may include internal and trailing padding used to align the members of the structure or union on memory boundaries. Thus, the result may not correspond to the size calculated by adding up the storage requirements of the individual members.

If an unsized array is the last element of a structure, the **sizeof** operator returns the size of the structure without the array.

```
buffer = calloc(100, sizeof (int) );
```

This example uses the **sizeof** operator to pass the size of an **int**, which varies among machines, as an argument to a run-time function named **calloc**. The value returned by the function is stored in `buffer`.

```
static char *strings[] ={
          "this is string one",
          "this is string two",
          "this is string three",
         };
const int string_no = ( sizeof strings ) / ( sizeof strings[0] );
```

In this example, `strings` is an array of pointers to **char**. The number of pointers is the number of elements in the array, but is not specified. It is easy to determine the number of pointers by using the **sizeof** operator to calculate the number of elements in the array. The **const** integer value `string_no` is initialized to this number. Because it is a **const** value, `string_no` cannot be modified.

See Also

Reference

C++ Operators

Cast Operators

A type cast provides a method for explicit conversion of the type of an object in a specific situation.

Syntax

cast-expression:

unary-expression

(*type-name* **)** *cast-expression*

The compiler treats *cast-expression* as type *type-name* after a type cast has been made. Casts can be used to convert objects of any scalar type to or from any other scalar type. Explicit type casts are constrained by the same rules that determine the effects of implicit conversions, discussed in Assignment Conversions. Additional restraints on casts may result from the actual sizes or representation of specific types. See Storage of Basic Types for information on actual sizes of integral types. For more information on type casts, see Type-Cast Conversions.

See Also

Reference

Cast Operator: ()

C Multiplicative Operators

The multiplicative operators perform multiplication (*****), division (**/**), and remainder (**%**) operations.

Syntax

multiplicative-expression:

cast-expression

multiplicative-expression ***** *cast-expression*

multiplicative-expression **/** *cast-expression*

multiplicative-expression **%** *cast-expression*

The operands of the remainder operator (**%**) must be integral. The multiplication (*****) and division (**/**) operators can take integral- or floating-type operands; the types of the operands can be different.

The multiplicative operators perform the usual arithmetic conversions on the operands. The type of the result is the type of the operands after conversion.

Note
Since the conversions performed by the multiplicative operators do not provide for overflow or underflow conditions, information may be lost if the result of a multiplicative operation cannot be represented in the type of the operands after conversion.

The C multiplicative operators are described below:

Operator	**Description**
*****	The multiplication operator causes its two operands to be multiplied.
/	The division operator causes the first operand to be divided by the second. If two integer operands are divided and the result is not an integer, it is truncated according to the following rules:
	• The result of division by 0 is undefined according to the ANSI C standard. The Microsoft C compiler generates an error at compile time or run time.
	• If both operands are positive or unsigned, the result is truncated toward 0.
	• If either operand is negative, whether the result of the operation is the largest integer less than or equal to the algebraic quotient or is the smallest integer greater than or equal to the algebraic quotient is implementation defined. (See the Microsoft Specific section below.)
%	The result of the remainder operator is the remainder when the first operand is divided by the second. When the division is inexact, the result is determined by the following rules:
	• If the right operand is zero, the result is undefined.
	• If both operands are positive or unsigned, the result is positive.
	• If either operand is negative and the result is inexact, the result is implementation defined. (See the Microsoft Specific section below.)

Microsoft Specific

In division where either operand is negative, the direction of truncation is toward 0.

If either operation is negative in division with the remainder operator, the result has the same sign as the dividend (the first operand in the expression).

END Microsoft Specific

Examples

The declarations shown below are used for the following examples:

```
int i = 10, j = 3, n;
double x = 2.0, y;
```

This statement uses the multiplication operator:

```
y = x * i;
```

In this case, `x` is multiplied by `i` to give the value 20.0. The result has **double** type.

```
n = i / j;
```

In this example, 10 is divided by 3. The result is truncated toward 0, yielding the integer value 3.

```
n = i % j;
```

This statement assigns `n` the integer remainder, 1, when 10 is divided by 3.

Microsoft Specific

The sign of the remainder is the same as the sign of the dividend. For example:

```
50 % -6 = 2
-50 % 6 = -2
```

In each case, `50` and `2` have the same sign.

END Microsoft Specific

See Also

Reference

Multiplicative Operators: *, /, and %

C Additive Operators

The additive operators perform addition (**+**) and subtraction (**–**).

Syntax
additive-expression:

multiplicative-expression

additive-expression **+** *multiplicative-expression*

additive-expression **–** *multiplicative-expression*

Note
Although the syntax for *additive-expression* includes *multiplicative-expression*, this does not imply that expressions using multiplication are required. See the syntax in C Language Syntax Summary, for *multiplicative-expression*, *cast-expression*, and *unary-expression*.

The operands can be integral or floating values. Some additive operations can also be performed on pointer values, as outlined under the discussion of each operator.

The additive operators perform the usual arithmetic conversions on integral and floating operands. The type of the result is the type of the operands after conversion. Since the conversions performed by the additive operators do not provide for overflow or underflow conditions, information may be lost if the result of an additive operation cannot be represented in the type of the operands after conversion.

See Also
Reference
Additive Operators: + and -

Addition (+)

The addition operator (**+**) causes its two operands to be added. Both operands can be either integral or floating types, or one operand can be a pointer and the other an integer.

When an integer is added to a pointer, the integer value (*i*) is converted by multiplying it by the size of the value that the pointer addresses. After conversion, the integer value represents *i* memory positions, where each position has the length specified by the pointer type. When the converted integer value is added to the pointer value, the result is a new pointer value representing the address *i* positions from the original address. The new pointer value addresses a value of the same type as the original pointer value and therefore is the same as array indexing (see One-Dimensional Arrays and Multidimensional Arrays). If the sum pointer points outside the array, except at the first location beyond the high end, the result is undefined. For more information, see Pointer Arithmetic.

See Also

Reference

C Additive Operators

Subtraction (–)

The subtraction operator (**–**) subtracts the second operand from the first. Both operands can be either integral or floating types, or one operand can be a pointer and the other an integer.

When two pointers are subtracted, the difference is converted to a signed integral value by dividing the difference by the size of a value of the type that the pointers address. The size of the integral value is defined by the type **ptrdiff_t** in the standard include file STDDEF.H. The result represents the number of memory positions of that type between the two addresses. The result is only guaranteed to be meaningful for two elements of the same array, as discussed in Pointer Arithmetic.

When an integer value is subtracted from a pointer value, the subtraction operator converts the integer value (*i*) by multiplying it by the size of the value that the pointer addresses. After conversion, the integer value represents *i* memory positions, where each position has the length specified by the pointer type. When the converted integer value is subtracted from the pointer value, the result is the memory address *i* positions before the original address. The new pointer points to a value of the type addressed by the original pointer value.

See Also

Reference

C Additive Operators

Using the Additive Operators

The following examples, which illustrate the addition and subtraction operators, use these declarations:

```
int i = 4, j;
float x[10];
float *px;
```

These statements are equivalent:

```
px = &x[4 + i];
px = &x[4] + i;
```

The value of `i` is multiplied by the length of a **float** and added to `&x[4]`. The resulting pointer value is the address of `x[8]`.

```
j = &x[i] - &x[i-2];
```

In this example, the address of the third element of `x` (given by `x[i-2]`) is subtracted from the address of the fifth element of `x` (given by `x[i]`). The difference is divided by the length of a **float**; the result is the integer value 2.

See Also

Reference

C Additive Operators

Pointer Arithmetic

Additive operations involving a pointer and an integer give meaningful results only if the pointer operand addresses an array member and the integer value produces an offset within the bounds of the same array. When the integer value is converted to an address offset, the compiler assumes that only memory positions of the same size lie between the original address and the address plus the offset.

This assumption is valid for array members. By definition, an array is a series of values of the same type; its elements reside in contiguous memory locations. However, storage for any types except array elements is not guaranteed to be filled by the same type of identifiers. That is, blanks can appear between memory positions, even positions of the same type. Therefore, the results of adding to or subtracting from the addresses of any values but array elements are undefined.

Similarly, when two pointer values are subtracted, the conversion assumes that only values of the same type, with no blanks, lie between the addresses given by the operands.

See Also

Reference

C Additive Operators

Bitwise Shift Operators

The shift operators shift their first operand left (**<<**) or right (**>>**) by the number of positions the second operand specifies.

Syntax

shift-expression:

additive-expression

shift-expression **<<** *additive-expression* *shift-expression* **>>** *additive-expression*

Both operands must be integral values. These operators perform the usual arithmetic conversions; the type of the result is the type of the left operand after conversion.

For leftward shifts, the vacated right bits are set to 0. For rightward shifts, the vacated left bits are filled based on the type of the first operand after conversion. If the type is **unsigned**, they are set to 0. Otherwise, they are filled with copies of the sign bit. For left-shift operators without overflow, the statement

```
expr1 << expr2
```

is equivalent to multiplication by 2expr2. For right-shift operators,

```
expr1 >> expr2
```

is equivalent to division by 2expr2 if `expr1` is unsigned or has a nonnegative value.

The result of a shift operation is undefined if the second operand is negative, or if the right operand is greater than or equal to the width in bits of the promoted left operand.

Since the conversions performed by the shift operators do not provide for overflow or underflow conditions, information may be lost if the result of a shift operation cannot be represented in the type of the first operand after conversion.

```
unsigned int x, y, z;

x = 0x00AA;
y = 0x5500;

z = ( x << 8 ) + ( y >> 8 );
```

In this example, `x` is shifted left eight positions and `y` is shifted right eight positions. The shifted values are added, giving 0xAA55, and assigned to `z`.

Shifting a negative value to the right yields half the absolute value, rounded down. For example, –253 (binary 11111111 00000011) shifted right one bit produces –127 (binary 11111111 10000001). A *positive* 253 shifts right to produce +126.

Right shifts preserve the sign bit. When a signed integer shifts right, the most-significant bit remains set. When an unsigned integer shifts right, the most-significant bit is cleared.

If 0xF000 is unsigned, the result is 0x7800. If 0xF0000000 is signed, a right shift produces 0xF8000000. Shifting a positive number right 32 times produces 0xF0000000. Shifting a negative number right 32 times produces 0xFFFFFFFF.

See Also

Reference

Shift Operators: >> and <<

C Relational and Equality Operators

The binary relational and equality operators compare their first operand to their second operand to test the validity of the specified relationship. The result of a relational expression is 1 if the tested relationship is true and 0 if it is false. The type of the result is **int**.

Syntax

relational-expression:

shift-expression

relational-expression **<** *shift-expression*

relational-expression **>** *shift-expression*

relational-expression **<=** *shift-expression*

relational-expression **>=** *shift-expression*

equality-expression:

relational-expression

equality-expression **==** *relational-expression*

equality-expression **!=** *relational-expression*

The relational and equality operators test the following relationships:

Operator	Relationship Tested
<	First operand less than second operand
>	First operand greater than second operand
<=	First operand less than or equal to second operand
>=	First operand greater than or equal to second operand
==	First operand equal to second operand
!=	First operand not equal to second operand

The first four operators in the list above have a higher precedence than the equality operators (**==** and **!=**). See the precedence information in the table Precedence and Associativity of C Operators.

The operands can have integral, floating, or pointer type. The types of the operands can be different. Relational operators perform the usual arithmetic conversions on integral and floating type operands. In addition, you can use the following combinations of operand types with the relational and equality operators:

- Both operands of any relational or equality operator can be pointers to the same type. For the equality (**==**) and inequality (**!=**) operators, the result of the comparison indicates whether the two pointers address the same memory location. For the other relational operators (**<**, **>**, **<**=, and **>**=), the result of the comparison indicates the relative position of the two memory addresses of the objects pointed to. Relational operators compare only offsets.

  Pointer comparison is defined only for parts of the same object. If the pointers refer to members of an array, the comparison is equivalent to comparison of the corresponding subscripts. The address of the first array element is "less than" the address of the last element. In the case of structures, pointers to structure members declared later are "greater than" pointers to members declared earlier in the structure. Pointers to the members of the same union are equal.

- A pointer value can be compared to the constant value 0 for equality (**==**) or inequality (**!=**). A pointer with a value of 0 is called a "null" pointer; that is, it does not point to a valid memory location.

- The equality operators follow the same rules as the relational operators, but permit additional possibilities: a pointer can

be compared to a constant integral expression with value 0, or to a pointer to **void**. If two pointers are both null pointers, they compare as equal. Equality operators compare both segment and offset.

Examples

The examples below illustrate relational and equality operators.

```
int x = 0, y = 0;
if ( x < y )
```

Because `x` and `y` are equal, the expression in this example yields the value 0.

```
char array[10];
char *p;

for ( p = array; p < &array[10]; p++ )
    *p = '\0';
```

The fragment in this example sets each element of `array` to a null character constant.

```
enum color { red, white, green } col;
   .
   .
   .
   if ( col == red )
   .
   .
   .
```

These statements declare an enumeration variable named `col` with the tag `color`. At any time, the variable may contain an integer value of 0, 1, or 2, which represents one of the elements of the enumeration set `color`: the color red, white, or green, respectively. If `col` contains 0 when the **if** statement is executed, any statements depending on the **if** will be executed.

See Also

Reference

Relational Operators: <, >, <=, and >=

Equality Operators: == and !=

C Bitwise Operators

The bitwise operators perform bitwise-AND (**&**), bitwise-exclusive-OR (**^**), and bitwise-inclusive-OR (**|**) operations.

Syntax

AND-expression:

equality-expression

AND-expression **&** *equality-expression*

exclusive-OR-expression:

AND-expression

exclusive-OR-expression **^** *AND-expression*

inclusive-OR-expression:

exclusive-OR-expression

inclusive-OR-expression | *exclusive-OR-expression*

The operands of bitwise operators must have integral types, but their types can be different. These operators perform the usual arithmetic conversions; the type of the result is the type of the operands after conversion.

The C bitwise operators are described below:

Operator	Description
&	The bitwise-AND operator compares each bit of its first operand to the corresponding bit of its second operand. If both bi ts are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.
^	The bitwise-exclusive-OR operator compares each bit of its first operand to the corresponding bit of its second operand. I f one bit is 0 and the other bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.
|	The bitwise-inclusive-OR operator compares each bit of its first operand to the corresponding bit of its second operand. If either bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

Examples

These declarations are used for the following three examples:

```
short i = 0xAB00;
short j = 0xABCD;
short n;

n = i & j;
```

The result assigned to `n` in this first example is the same as `i` (0xAB00 hexadecimal).

```
n = i | j;

n = i ^ j;
```

The bitwise-inclusive OR in the second example results in the value 0xABCD (hexadecimal), while the bitwise-exclusive OR in the third example produces 0xCD (hexadecimal).

Microsoft Specific

The results of bitwise operation on signed integers is implementation-defined according to the ANSI C standard. For the Microsoft C compiler, bitwise operations on signed integers work the same as bitwise operations on unsigned integers. For

example, `-16 & 99` can be expressed in binary as

```
  11111111 11110000
& 00000000 01100011
  ___________________
  00000000 01100000
```

The result of the bitwise AND is 96 decimal.

END Microsoft Specific

See Also

Reference

Bitwise AND Operator: &

Bitwise Exclusive OR Operator: ^

Bitwise Inclusive OR Operator: |

C Logical Operators

The logical operators perform logical-AND (**&&**) and logical-OR (**||**) operations.

Syntax

logical-AND-expression:

inclusive-OR-expression

logical-AND-expression **&&** *inclusive-OR-expression*

logical-OR-expression:

logical-AND-expression

logical-OR-expression **||** *logical-AND-expression*

Logical operators do not perform the usual arithmetic conversions. Instead, they evaluate each operand in terms of its equivalence to 0. The result of a logical operation is either 0 or 1. The result's type is **int**.

The C logical operators are described below:

Operator	Description
&&	The logical-AND operator produces the value 1 if both operands have nonzero values. If either operand is equal to 0, the result is 0. If the first operand of a logical-AND operation is equal to 0, the second operand is not evaluated.
||	The logical-OR operator performs an inclusive-OR operation on its operands. The result is 0 if both operands have 0 values. If either operand has a nonzero value, the result is 1. If the first operand of a logical-OR operation has a nonzero value, the second operand is not evaluated.

The operands of logical-AND and logical-OR expressions are evaluated from left to right. If the value of the first operand is sufficient to determine the result of the operation, the second operand is not evaluated. This is called "short-circuit evaluation." There is a sequence point after the first operand. See Sequence Points for more information.

Examples

The following examples illustrate the logical operators:

```
int w, x, y, z;

if ( x < y && y < z )
    printf( "x is less than z\n" );
```

In this example, the `printf` function is called to print a message if `x` is less than `y` and `y` is less than `z`. If `x` is greater than `y`, the second operand (`y < z`) is not evaluated and nothing is printed. Note that this could cause problems in cases where the second operand has side effects that are being relied on for some other reason.

```
printf( "%d" , (x == w || x == y || x == z) );
```

In this example, if `x` is equal to either `w`, `y`, or `z`, the second argument to the `printf` function evaluates to true and the value 1 is printed. Otherwise, it evaluates to false and the value 0 is printed. As soon as one of the conditions evaluates to true, evaluation ceases.

See Also

Reference

Logical AND Operator: &&

Logical OR Operator: ||

Conditional-Expression Operator

C has one ternary operator: the conditional-expression operator (**? :**).

Syntax

conditional-expression:

logical-OR-expression

logical-OR expression **?** *expression* **:** *conditional-expression*

The *logical-OR-expression* must have integral, floating, or pointer type. It is evaluated in terms of its equivalence to 0. A sequence point follows *logical-OR-expression*. Evaluation of the operands proceeds as follows:

- If *logical-OR-expression* is not equal to 0, *expression* is evaluated. The result of evaluating the expression is given by the nonterminal *expression*. (This means *expression* is evaluated only if *logical-OR-expression* is true.)
- If *logical-OR-expression* equals 0, *conditional-expression* is evaluated. The result of the expression is the value of *conditional-expression*. (This means *conditional-expression* is evaluated only if *logical-OR-expression* is false.)

Note that either *expression* or *conditional-expression* is evaluated, but not both.

The type of the result of a conditional operation depends on the type of the *expression* or *conditional-expression* operand, as follows:

- If *expression* or *conditional-expression* has integral or floating type (their types can be different), the operator performs the usual arithmetic conversions. The type of the result is the type of the operands after conversion.
- If both *expression* and *conditional-expression* have the same structure, union, or pointer type, the type of the result is the same structure, union, or pointer type.
- If both operands have type **void**, the result has type **void**.
- If either operand is a pointer to an object of any type, and the other operand is a pointer to **void**, the pointer to the object is converted to a pointer to **void** and the result is a pointer to **void**.
- If either *expression* or *conditional-expression* is a pointer and the other operand is a constant expression with the value 0, the type of the result is the pointer type.

In the type comparison for pointers, any type qualifiers (**const** or **volatile**) in the type to which the pointer points are insignificant, but the result type inherits the qualifiers from both components of the conditional.

Examples

The following examples show uses of the conditional operator:

```
j = ( i < 0 ) ? ( -i ) : ( i );
```

This example assigns the absolute value of `i` to `j`. If `i` is less than 0, `-i` is assigned to `j`. If `i` is greater than or equal to 0, `i` is assigned to `j`.

```
void f1( void );
void f2( void );
int x;
int y;
    .
    .
    .
( x == y ) ? ( f1() ) : ( f2() );
```

In this example, two functions, `f1` and `f2`, and two variables, `x` and `y`, are declared. Later in the program, if the two variables have the same value, the function `f1` is called. Otherwise, `f2` is called.

See Also

Reference

Conditional Operator: ? :

C Assignment Operators

An assignment operation assigns the value of the right-hand operand to the storage location named by the left-hand operand. Therefore, the left-hand operand of an assignment operation must be a modifiable l-value. After the assignment, an assignment expression has the value of the left operand but is not an l-value.

Syntax

assignment-expression:

conditional-expression

unary-expression assignment-operator assignment-expression

assignment-operator: one of

= *= /= %= += −= <<= >>= &= ^= |=

The assignment operators in C can both transform and assign values in a single operation. C provides the following assignment operators:

Operator	Operation Performed
=	Simple assignment
***=**	Multiplication assignment
/=	Division assignment
%=	Remainder assignment
+=	Addition assignment
−=	Subtraction assignment
<<=	Left-shift assignment
>>=	Right-shift assignment
&=	Bitwise-AND assignment
^=	Bitwise-exclusive-OR assignment
|=	Bitwise-inclusive-OR assignment

In assignment, the type of the right-hand value is converted to the type of the left-hand value, and the value is stored in the left operand after the assignment has taken place. The left operand must not be an array, a function, or a constant. The specific conversion path, which depends on the two types, is outlined in detail in Type Conversions.

See Also

Reference

Assignment Operators: =, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, and |=

Simple Assignment (C)

The simple-assignment operator assigns its right operand to its left operand. The value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand. The conversion rules for assignment apply (see Assignment Conversions).

```
double x;
int y;

x = y;
```

In this example, the value of `y` is converted to type **double** and assigned to `x`.

See Also

Reference

C Assignment Operators

C Compound Assignment

The compound-assignment operators combine the simple-assignment operator with another binary operator. Compound-assignment operators perform the operation specified by the additional operator, then assign the result to the left operand. For example, a compound-assignment expression such as

```
expression1 += expression2
```

can be understood as

```
expression1 = expression1 + expression2
```

However, the compound-assignment expression is not equivalent to the expanded version because the compound-assignment expression evaluates *expression1* only once, while the expanded version evaluates *expression1* twice: in the addition operation and in the assignment operation.

The operands of a compound-assignment operator must be of integral or floating type. Each compound-assignment operator performs the conversions that the corresponding binary operator performs and restricts the types of its operands accordingly. The addition-assignment (**+=**) and subtraction-assignment (**-=**) operators can also have a left operand of pointer type, in which case the right-hand operand must be of integral type. The result of a compound-assignment operation has the value and type of the left operand.

```
#define MASK 0xff00

n &= MASK;
```

In this example, a bitwise-inclusive-AND operation is performed on `n` and `MASK`, and the result is assigned to `n`. The manifest constant `MASK` is defined with a #define preprocessor directive.

See Also
Reference
C Assignment Operators

Sequential-Evaluation Operator

The sequential-evaluation operator, also called the "comma operator," evaluates its two operands sequentially from left to right.

Syntax

expression:

assignment-expression

expression **,** *assignment-expression*

The left operand of the sequential-evaluation operator is evaluated as a **void** expression. The result of the operation has the same value and type as the right operand. Each operand can be of any type. The sequential-evaluation operator does not perform type conversions between its operands, and it does not yield an l-value. There is a sequence point after the first operand, which means all side effects from the evaluation of the left operand are completed before beginning evaluation of the right operand. See Sequence Points for more information.

The sequential-evaluation operator is typically used to evaluate two or more expressions in contexts where only one expression is allowed.

Commas can be used as separators in some contexts. However, you must be careful not to confuse the use of the comma as a separator with its use as an operator; the two uses are completely different.

Example

This example illustrates the sequential-evaluation operator:

```
for ( i = j = 1; i + j < 20; i += i, j-- );
```

In this example, each operand of the **for** statement's third expression is evaluated independently. The left operand `i += i` is evaluated first; then the right operand, `j--`, is evaluated.

```
func_one( x, y + 2, z );
func_two( (x--, y + 2), z );
```

In the function call to `func_one`, three arguments, separated by commas, are passed: `x`, `y + 2`, and `z`. In the function call to `func_two`, parentheses force the compiler to interpret the first comma as the sequential-evaluation operator. This function call passes two arguments to `func_two`. The first argument is the result of the sequential-evaluation operation `(x--, y + 2)`, which has the value and type of the expression `y + 2`; the second argument is `z`.

See Also

Reference

Comma Operator: ,

Type Conversions (C)

Type conversions depend on the specified operator and the type of the operand or operators. Type conversions are performed in the following cases:

- When a value of one type is assigned to a variable of a different type or an operator converts the type of its operand or operands before performing an operation
- When a value of one type is explicitly cast to a different type
- When a value is passed as an argument to a function or when a type is returned from a function

A character, a short integer, or an integer bit field, all either signed or not, or an object of enumeration type, can be used in an expression wherever an integer can be used. If an **int** can represent all the values of the original type, then the value is converted to **int**; otherwise, it is converted to **unsigned int**. This process is called "integral promotion." Integral promotions preserve value. That is, the value after promotion is guaranteed to be the same as before the promotion. See Usual Arithmetic Conversions for more information.

See Also

Concepts

Expressions and Assignments

Assignment Conversions

In assignment operations, the type of the value being assigned is converted to the type of the variable that receives the assignment. C allows conversions by assignment between integral and floating types, even if information is lost in the conversion. The conversion method used depends on the types involved in the assignment, as described in Usual Arithmetic Conversions and in the following sections:

- Conversions from Signed Integral Types
- Conversions from Unsigned Integral Types
- Conversions from Floating-Point Types
- Conversions to and from Pointer Types
- Conversions from Other Types

Type qualifiers do not affect the allowability of the conversion although a **const** l-value cannot be used on the left side of the assignment.

See Also

Concepts

Type Conversions (C)

Conversions from Signed Integral Types

When a signed integer is converted to an unsigned integer with equal or greater size and the value of the signed integer is not negative, the value is unchanged. The conversion is made by sign-extending the signed integer. A signed integer is converted to a shorter signed integer by truncating the high-order bits. The result is interpreted as an unsigned value, as shown in this example.

```
int i = -3;
unsigned short u;

u = i;
printf_s( "%hu\n", u );  // Prints 65533
```

No information is lost when a signed integer is converted to a floating value, except that some precision may be lost when a **long int** or **unsigned long int** value is converted to a **float** value.

The following table summarizes conversions from signed integral types. This table assumes that the **char** type is signed by default. If you use a compile-time option to change the default for the **char** type to unsigned, the conversions given in the Conversions from Unsigned Integral Types table for the **unsigned char** type apply instead of the conversions in the following table, Conversions from Signed Integral Types.

Conversions from Signed Integral Types

From	To	Method
char1	**short**	Sign-extend
char	**long**	Sign-extend
char	**unsigned char**	Preserve pattern; high-order bit loses function as sign bit
char	**unsigned short**	Sign-extend to **short**; convert **short** to **unsigned short**
char	**unsigned long**	Sign-extend to **long**; convert **long** to **unsigned long**
char	**float**	Sign-extend to **long**; convert **long** to **float**
char	**double**	Sign-extend to **long**; convert **long** to **double**
char	**long double**	Sign-extend to **long**; convert **long** to **double**
short	**char**	Preserve low-order byte
short	**long**	Sign-extend
short	**unsigned char**	Preserve low-order byte
short	**unsigned short**	Preserve bit pattern; high-order bit loses function as sign bit
short	**unsigned long**	Sign-extend to **long**; convert **long** to **unsigned long**
short	**float**	Sign-extend to **long**; convert **long** to **float**
short	**double**	Sign-extend to **long**; convert **long** to **double**
short	**long double**	Sign-extend to **long**; convert **long** to **double**

long	**char**	Preserve low-order byte
long	**short**	Preserve low-order word
long	**unsigned char**	Preserve low-order byte
long	**unsigned short**	Preserve low-order word
long	**unsigned long**	Preserve bit pattern; high-order bit loses function as sign bit
long	**float**	Represent as **float**. If **long** cannot be represented exactly, some precision is lost.
long	**double**	Represent as **double**. If **long** cannot be represented exactly as a **double**, some precision is lost.
long	**long double**	Represent as **double**. If **long** cannot be represented exactly as a **double**, some precision is lost.

1. All **char** entries assume that the **char** type is signed by default.

Microsoft Specific

For the Microsoft 32-bit C compiler, an integer is equivalent to a **long**. Conversion of an **int** value proceeds the same as for a **long**.

END Microsoft Specific

See Also

Concepts

Assignment Conversions

Conversions from Unsigned Integral Types

An unsigned integer is converted to a shorter unsigned or signed integer by truncating the high-order bits, or to a longer unsigned or signed integer by zero-extending (see the Conversions from Unsigned Integral Types table).

When the value with integral type is demoted to a signed integer with smaller size, or an unsigned integer is converted to its corresponding signed integer, the value is unchanged if it can be represented in the new type. However, the value it represents changes if the sign bit is set, as in the following example.

```
int j;
unsigned short k = 65533;

j = k;
printf_s( "%hd\n", j );   // Prints -3
```

If it cannot be represented, the result is implementation-defined. See Type-Cast Conversions for information on the Microsoft C compiler's handling of demotion of integers. The same behavior results from integer conversion or from type casting the integer.

Unsigned values are converted in a way that preserves their value and is not representable directly in C. The only exception is a conversion from **unsigned long** to **float**, which loses at most the low-order bits. Otherwise, value is preserved, signed or unsigned. When a value of integral type is converted to floating, and the value is outside the range representable, the result is undefined. (See Storage of Basic Types for information about the range for integral and floating-point types.)

The following table summarizes conversions from unsigned integral types.

Conversions from Unsigned Integral Types

From	To	Method
unsigned char	**char**	Preserve bit pattern; high-order bit becomes sign bit
unsigned char	**short**	Zero-extend
unsigned char	**long**	Zero-extend
unsigned char	**unsigned short**	Zero-extend
unsigned char	**unsigned long**	Zero-extend
unsigned char	**float**	Convert to **long**; convert **long** to **float**
unsigned char	**double**	Convert to **long**; convert **long** to **double**
unsigned char	**long double**	Convert to **long**; convert **long** to **double**
unsigned short	**char**	Preserve low-order byte
unsigned short	**short**	Preserve bit pattern; high-order bit becomes sign bit
unsigned short	**long**	Zero-extend
unsigned short	**unsigned char**	Preserve low-order byte
unsigned short	**unsigned long**	Zero-extend
unsigned short	**float**	Convert to **long**; convert **long** to **float**

unsigned short	**double**	Convert to **long**; convert **long** to **double**
unsigned short	**long double**	Convert to **long**; convert **long** to **double**
unsigned long	**char**	Preserve low-order byte
unsigned long	**short**	Preserve low-order word
unsigned long	**long**	Preserve bit pattern; high-order bit becomes sign bit
unsigned long	**unsigned char**	Preserve low-order byte
unsigned long	**unsigned short**	Preserve low-order word
unsigned long	**float**	Convert to **long**; convert **long** to **float**
unsigned long	**double**	Convert directly to **double**
unsigned long	**long double**	Convert to **long**; convert **long** to **double**

Microsoft Specific

For the Microsoft 32-bit C compiler, the **unsigned int** type is equivalent to the **unsigned long** type. Conversion of an **unsigned int** value proceeds in the same way as conversion of an **unsigned long**. Conversions from **unsigned long** values to **float** are not accurate if the value being converted is larger than the maximum positive signed **long** value.

END Microsoft Specific

See Also

Concepts

Assignment Conversions

Conversions from Floating-Point Types

A **float** value converted to a **double** or **long double**, or a **double** converted to a **long double**, undergoes no change in value. A **double** value converted to a **float** value is represented exactly, if possible. Precision may be lost if the value cannot be represented exactly. If the result is out of range, the behavior is undefined. See Limits on Floating-Point Constants for the range of floating-point types.

A floating value is converted to an integral value by first converting to a **long**, then from the **long** value to the specific integral value. The decimal portion of the floating value is discarded in the conversion to a **long**. If the result is still too large to fit into a **long**, the result of the conversion is undefined.

Microsoft Specific

When converting a **double** or **long double** floating-point number to a smaller floating-point number, the value of the floating-point variable is truncated toward zero when an underflow occurs. An overflow causes a run-time error. Note that the Microsoft C compiler maps **long double** to type **double**.

END Microsoft Specific

The following table summarizes conversions from floating types.

Conversions from Floating-Point Types

From	To	Method
float	**char**	Convert to **long**; convert **long** to **char**
float	**short**	Convert to **long**; convert **long** to **short**
float	**long**	Truncate at decimal point. If result is too large to be represented as **long**, result is undefined.
float	**unsigned short**	Convert to **long**; convert **long** to **unsigned short**
float	**unsigned long**	Convert to **long**; convert **long** to **unsigned long**
float	**double**	Change internal representation
float	**long double**	Change internal representation
double	**char**	Convert to **float**; convert **float** to **char**
double	**short**	Convert to **float**; convert **float** to **short**
double	**long**	Truncate at decimal point. If result is too large to be represented as **long**, result is undefined.
double	**unsigned short**	Convert to **long**; convert **long** to **unsigned short**
double	**unsigned long**	Convert to **long**; convert **long** to **unsigned long**
double	**float**	Represent as a **float**. If **double** value cannot be represented exactly as **float**, loss of precision occurs. If value is too large to be represented as **float**, the result is undefined.
long double	**char**	Convert to **float**; convert **float** to **char**

long double	**short**	Convert to **float**; convert **float** to **short**
long double	**long**	Truncate at decimal point. If result is too large to be represented as **long**, result is undefined.
long double	**unsigned short**	Convert to **long**; convert **long** to **unsigned short**
long double	**unsigned long**	Convert to **long**; convert **long** to **unsigned long**
long double	**float**	Represent as a **float**. If **double** value cannot be represented exactly as **float**, loss of precision occurs. If value is too large to be represented as **float**, the result is undefined.
long double	**double**	The **long double** value is treated as **double**.

Conversions from **float**, **double**, or **long double** values to **unsigned long** are not accurate if the value being converted is larger than the maximum positive **long** value.

See Also

Concepts

Assignment Conversions

Conversions to and from Pointer Types

A pointer to one type of value can be converted to a pointer to a different type. However, the result may be undefined because of the alignment requirements and sizes of different types in storage. A pointer to an object can be converted to a pointer to an object whose type requires less or equally strict storage alignment, and back again without change.

A pointer to **void** can be converted to or from a pointer to any type, without restriction or loss of information. If the result is converted back to the original type, the original pointer is recovered.

If a pointer is converted to another pointer with the same type but having different or additional qualifiers, the new pointer is the same as the old except for restrictions imposed by the new qualifier.

A pointer value can also be converted to an integral value. The conversion path depends on the size of the pointer and the size of the integral type, according to the following rules:

- If the size of the pointer is greater than or equal to the size of the integral type, the pointer behaves like an unsigned value in the conversion, except that it cannot be converted to a floating value.
- If the pointer is smaller than the integral type, the pointer is first converted to a pointer with the same size as the integral type, then converted to the integral type.

Conversely, an integral type can be converted to a pointer type according to the following rules:

- If the integral type is the same size as the pointer type, the conversion simply causes the integral value to be treated as a pointer (an unsigned integer).
- If the size of the integral type is different from the size of the pointer type, the integral type is first converted to the size of the pointer, using the conversion paths given in the tables Conversion from Signed Integral Types and Conversion from Unsigned Integral Types. It is then treated as a pointer value.

An integral constant expression with value 0 or such an expression cast to type **void *** can be converted by a type cast, by assignment, or by comparison to a pointer of any type. This produces a null pointer that is equal to another null pointer of the same type, but this null pointer is not equal to any pointer to a function or to an object. Integers other than the constant 0 can be converted to pointer type, but the result is not portable.

See Also

Concepts

Assignment Conversions

Conversions from Other Types

Since an **enum** value is an **int** value by definition, conversions to and from an **enum** value are the same as those for the **int** type. For the Microsoft C compiler, an integer is the same as a **long**.

Microsoft Specific

No conversions between structure or union types are allowed.

Any value can be converted to type **void**, but the result of such a conversion can be used only in a context where an expression value is discarded, such as in an expression statement.

The **void** type has no value, by definition. Therefore, it cannot be converted to any other type, and other types cannot be converted to **void** by assignment. However, you can explicitly cast a value to type **void**, as discussed in Type-Cast Conversions.

END Microsoft Specific

See Also

Concepts

Assignment Conversions

Type-Cast Conversions

You can use type casts to explicitly convert types.

Syntax

cast-expression:

unary expression

(*type-name* **)** *cast-expression*

type-name:

specifier-qualifier-list abstract-declarator $_{opt}$

The *type-name* is a type and *cast-expression* is a value to be converted to that type. An expression with a type cast is not an l-value. The *cast-expression* is converted as though it had been assigned to a variable of type *type-name*. The conversion rules for assignments (outlined in Assignment Conversions) apply to type casts as well. The following table shows the types that can be cast to any given type.

Legal Type Casts

Destination Types	**Potential Sources**
Integral types	Any integer type or floating-point type, or pointer to an object
Floating-point	Any arithmetic type
A pointer to an object, or (**void** *)	Any integer type, (**void** *), a pointer to an object, or a function pointer
Function pointer	Any integral type, a pointer to an object, or a function pointer
A structure, union, or array	None
Void type	Any type

Any identifier can be cast to **void** type. However, if the type specified in a type-cast expression is not **void**, then the identifier being cast to that type cannot be a **void** expression. Any expression can be cast to **void**, but an expression of type **void** cannot be cast to any other type. For example, a function with **void** return type cannot have its return cast to another type.

Note that a **void** * expression has a type pointer to **void**, not type **void**. If an object is cast to **void** type, the resulting expression cannot be assigned to any item. Similarly, a type-cast object is not an acceptable l-value, so no assignment can be made to a type-cast object.

Microsoft Specific

A type cast can be an l-value expression as long as the size of the identifier does not change. For information on l-value expressions, see L-Value and R-Value Expressions.

END Microsoft Specific

You can convert an expression to type **void** with a cast, but the resulting expression can be used only where a value is not required. An object pointer converted to **void** * and back to the original type will return to its original value.

See Also

Concepts

Type Conversions (C)

Function-Call Conversions

The type of conversion performed on the arguments in a function call depends on the presence of a function prototype (forward declaration) with declared argument types for the called function.

If a function prototype is present and includes declared argument types, the compiler performs type checking (see Functions).

If no function prototype is present, only the usual arithmetic conversions are performed on the arguments in the function call. These conversions are performed independently on each argument in the call. This means that a **float** value is converted to a **double**; a **char** or **short** value is converted to an **int**; and an **unsigned char** or **unsigned short** is converted to an **unsigned int**.

See Also

Concepts

Type Conversions (C)

Statements (C)

The statements of a C program control the flow of program execution. In C, as in other programming languages, several kinds of statements are available to perform loops, to select other statements to be executed, and to transfer control. Following a brief overview of statement syntax, this section describes the C statements in alphabetical order:

break statement	if statement
compound statement	null statement
continue statement	return statement
do-while statement	switch statement
expression statement	try-except statement
for statement	try-finally statement
goto and labeled statements	while statement

See Also

Reference

C Language Reference

Overview of C Statements

C statements consist of tokens, expressions, and other statements. A statement that forms a component of another statement is called the "body" of the enclosing statement. Each statement type given by the following syntax is discussed in this section.

Syntax

statement:

labeled-statement

compound-statement

expression-statement

selection-statement

iteration-statement

jump-statement

try-except-statement

/* Microsoft Specific */try-finally-statement /* Microsoft Specific */

Frequently the statement body is a "compound statement." A compound statement consists of other statements that can include keywords. The compound statement is delimited by braces (**{ }**). All other C statements end with a semicolon (**;**). The semicolon is a statement terminator.

The expression statement contains a C expression that can contain the arithmetic or logical operators introduced in Expressions and Assignments. The null statement is an empty statement.

Any C statement can begin with an identifying label consisting of a name and a colon. Since only the **goto** statement recognizes statement labels, statement labels are discussed with **goto**. See The goto and Labeled Statements for more information.

See Also

Concepts

Statements (C)

The C break Statement

The **break** statement terminates the execution of the nearest enclosing **do**, **for**, **switch**, or **while** statement in which it appears. Control passes to the statement that follows the terminated statement.

Syntax
jump-statement:

break;

The **break** statement is frequently used to terminate the processing of a particular case within a **switch** statement. Lack of an enclosing iterative or **switch** statement generates an error.

Within nested statements, the **break** statement terminates only the **do**, **for**, **switch**, or **while** statement that immediately encloses it. You can use a **return** or **goto** statement to transfer control elsewhere out of the nested structure.

This example illustrates the **break** statement:

```
#include <stdio.h>
int main() {
   char c;
   for(;;) {
      printf_s( "\nPress any key, Q to quit: " );

      // Convert to character value
      scanf_s("%c", &c);
      if (c == 'Q')
          break;
   }
} // Loop exits only when 'Q' is pressed
```

See Also
Reference
The break Statement

The Compound Statement

A compound statement (also called a "block") typically appears as the body of another statement, such as the **if** statement. Declarations and Types describes the form and meaning of the declarations that can appear at the head of a compound statement.

Syntax

compound-statement:

{ *declaration-list* $_{opt}$ *statement-list* $_{opt}$ **}**

declaration-list:

declaration

declaration-list declaration

statement-list:

statement

statement-list statement

If there are declarations, they must come before any statements. The scope of each identifier declared at the beginning of a compound statement extends from its declaration point to the end of the block. It is visible throughout the block unless a declaration of the same identifier exists in an inner block.

Identifiers in a compound statement are presumed **auto** unless explicitly declared otherwise with **register**, **static**, or **extern**, except functions, which can only be **extern**. You can leave off the **extern** specifier in function declarations and the function will still be **extern**.

Storage is not allocated and initialization is not permitted if a variable or function is declared in a compound statement with storage class **extern**. The declaration refers to an external variable or function defined elsewhere.

Variables declared in a block with the **auto** or **register** keyword are reallocated and, if necessary, initialized each time the compound statement is entered. These variables are not defined after the compound statement is exited. If a variable declared inside a block has the **static** attribute, the variable is initialized when program execution begins and keeps its value throughout the program. See Storage Classes for information about **static**.

This example illustrates a compound statement:

```
if ( i > 0 )
{
    line[i] = x;
    x++;
    i--;
}
```

In this example, if `i` is greater than 0, all statements inside the compound statement are executed in order.

See Also

Concepts

Statements (C)

The C continue Statement

The **continue** statement passes control to the next iteration of the nearest enclosing **do**, **for**, or **while** statement in which it appears, bypassing any remaining statements in the **do**, **for**, or **while** statement body.

Syntax
jump-statement:

continue;

The next iteration of a **do**, **for**, or **while** statement is determined as follows:

- Within a **do** or a **while** statement, the next iteration starts by reevaluating the expression of the **do** or **while** statement.
- A **continue** statement in a **for** statement causes the first expression of the **for** statement to be evaluated. Then the compiler reevaluates the conditional expression and, depending on the result, either terminates or iterates the statement body. See The for Statement for more information on the **for** statement and its nonterminals.

This is an example of the **continue** statement:

```
while ( i-- > 0 )
{
    x = f( i );
    if ( x == 1 )
        continue;
    y += x * x;
}
```

In this example, the statement body is executed while `i` is greater than 0. First `f(i)` is assigned to `x`; then, if `x` is equal to 1, the **continue** statement is executed. The rest of the statements in the body are ignored, and execution resumes at the top of the loop with the evaluation of the loop's test.

See Also
Reference
The continue Statement

The do-while Statement (C)

The **do-while** statement lets you repeat a statement or compound statement until a specified expression becomes false.

Syntax
iteration-statement:

do *statement* **while (** *expression* **) ;**

The *expression* in a **do-while** statement is evaluated after the body of the loop is executed. Therefore, the body of the loop is always executed at least once.

The *expression* must have arithmetic or pointer type. Execution proceeds as follows:

1. The statement body is executed.
2. Next, *expression* is evaluated. If *expression* is false, the **do-while** statement terminates and control passes to the next statement in the program. If *expression* is true (nonzero), the process is repeated, beginning with step 1.

The **do-while** statement can also terminate when a **break**, **goto**, or **return** statement is executed within the statement body.

This is an example of the **do-while** statement:

```
do
{
    y = f( x );
    x--;
} while ( x > 0 );
```

In this **do-while** statement, the two statements `y = f( x );` and `x--;` are executed, regardless of the initial value of `x`. Then `x > 0` is evaluated. If `x` is greater than 0, the statement body is executed again and `x > 0` is reevaluated. The statement body is executed repeatedly as long as `x` remains greater than 0. Execution of the **do-while** statement terminates when `x` becomes 0 or negative. The body of the loop is executed at least once.

See Also
Reference
The do-while Statement (C++)

The Expression Statement

When an expression statement is executed, the expression is evaluated according to the rules outlined in Expressions and Assignments.

Syntax
expression-statement:

expression $_{\text{opt}}$ **;**

All side effects from the expression evaluation are completed before the next statement is executed. An empty expression statement is called a null statement. See The Null Statement for more information.

These examples demonstrate expression statements.

```
x = ( y + 3 );            /* x is assigned the value of y + 3  */
x++;                      /* x is incremented                  */
x = y = 0;                /* Both x and y are initialized to 0 */
proc( arg1, arg2 );       /* Function call returning void      */
y = z = ( f( x ) + 3 );   /* A function-call expression        */
```

In the last statement, the function-call expression, the value of the expression, which includes any value returned by the function, is increased by 3 and then assigned to both the variables `y` and `z`.

See Also
Concepts
Statements (C)

The C for Statement

The **for** statement lets you repeat a statement or compound statement a specified number of times. The body of a **for** statement is executed zero or more times until an optional condition becomes false. You can use optional expressions within the **for** statement to initialize and change values during the **for** statement's execution.

Syntax
iteration-statement:

for (*init-expression* $_{\text{opt}}$ **;** *cond-expression* $_{\text{opt}}$ **;** *loop-expression* $_{\text{opt}}$ **)***statement*

Execution of a **for** statement proceeds as follows:

1. The *init-expression*, if any, is evaluated. This specifies the initialization for the loop. There is no restriction on the type of *init-expression*.
2. The *cond-expression*, if any, is evaluated. This expression must have arithmetic or pointer type. It is evaluated before each iteration. Three results are possible:
   - If *cond-expression* is true (nonzero), *statement* is executed; then *loop-expression*, if any, is evaluated. The *loop-expression* is evaluated after each iteration. There is no restriction on its type. Side effects will execute in order. The process then begins again with the evaluation of *cond-expression*.
   - If *cond-expression* is omitted, *cond-expression* is considered true, and execution proceeds exactly as described in the previous paragraph. A **for** statement without a *cond-expression* argument terminates only when a **break** or **return** statement within the statement body is executed, or when a **goto** (to a labeled statement outside the **for** statement body) is executed.
   - If *cond-expression* is false (0), execution of the **for** statement terminates and control passes to the next statement in the program.

A **for** statement also terminates when a **break**, **goto**, or **return** statement within the statement body is executed. A **continue** statement in a **for** loop causes *loop-expression* to be evaluated. When a **break** statement is executed inside a **for** loop, *loop-expression* is not evaluated or executed. This statement

```
for( ;; );
```

is the customary way to produce an infinite loop which can only be exited with a **break**, **goto**, or **return** statement.

This example illustrates the **for** statement:

```
for ( i = space = tab = 0; i < MAX; i++ )
{
    if ( line[i] == ' ' )
        space++;
    if ( line[i] == '\t' )
    {
        tab++;
        line[i] = ' ';
    }
}
```

This example counts space (`' '`) and tab (`'\t'`) characters in the array of characters named `line` and replaces each tab character with a space. First `i`, `space`, and `tab` are initialized to 0. Then `i` is compared with the constant `MAX`; if `i` is less than `MAX`, the statement body is executed. Depending on the value of `line[i]`, the body of one or neither of the **if** statements is executed. Then `i` is incremented and tested against `MAX`; the statement body is executed repeatedly as long as `i` is less than `MAX`.

See Also
Concepts
Statements (C)

The goto and Labeled Statements

The **goto** statement transfers control to a label. The given label must reside in the same function and can appear before only one statement in the same function.

Syntax

statement:

labeled-statement

jump-statement

jump-statement:

goto *identifier* **;**

labeled-statement:

identifier **:** *statement*

A statement label is meaningful only to a **goto** statement; in any other context, a labeled statement is executed without regard to the label.

A *jump-statement* must reside in the same function and can appear before only one statement in the same function. The set of *identifier* names following a **goto** has its own name space so the names do not interfere with other identifiers. Labels cannot be redeclared. See Name Spaces for more information.

It is good programming style to use the **break**, **continue**, and **return** statement in preference to **goto** whenever possible. Since the **break** statement only exits from one level of the loop, a **goto** may be necessary for exiting a loop from within a deeply nested loop.

This example demonstrates the **goto** statement:

```
// goto.c
#include <stdio.h>

int main()
{
    int i, j;

    for ( i = 0; i < 10; i++ )
    {
        printf_s( "Outer loop executing. i = %d\n", i );
        for ( j = 0; j < 3; j++ )
        {
            printf_s( " Inner loop executing. j = %d\n", j );
            if ( i == 5 )
                goto stop;
        }
    }

    /* This message does not print: */
    printf_s( "Loop exited. i = %d\n", i );

    stop: printf_s( "Jumped to stop. i = %d\n", i );
}
```

In this example, a **goto** statement transfers control to the point labeled `stop` when `i` equals 5.

See Also

Concepts

Statements (C)

The C if Statement

The **if** statement controls conditional branching. The body of an **if** statement is executed if the value of the expression is nonzero. The syntax for the **if** statement has two forms.

Syntax
selection-statement:

if (*expression* **)** *statement*

if (*expression* **)** *statement* **else** *statement*

In both forms of the **if** statement, the expressions, which can have any value except a structure, are evaluated, including all side effects.

In the first form of the syntax, if *expression* is true (nonzero), *statement* is executed. If *expression* is false, *statement* is ignored. In the second form of syntax, which uses **else**, the second *statement* is executed if *expression* is false. With both forms, control then passes from the **if** statement to the next statement in the program unless one of the statements contains a **break**, **continue**, or **goto**.

The following are examples of the **if** statement:

```
if ( i > 0 )
    y = x / i;
else
{
    x = i;
    y = f( x );
}
```

In this example, the statement `y = x/i;` is executed if `i` is greater than 0. If `i` is less than or equal to 0, `i` is assigned to `x` and `f( x )` is assigned to `y`. Note that the statement forming the **if** clause ends with a semicolon.

When nesting **if** statements and **else** clauses, use braces to group the statements and clauses into compound statements that clarify your intent. If no braces are present, the compiler resolves ambiguities by associating each **else** with the closest **if** that lacks an **else**.

```
if ( i > 0 )            /* Without braces */
    if ( j > i )
        x = j;
    else
        x = i;
```

The **else** clause is associated with the inner **if** statement in this example. If `i` is less than or equal to 0, no value is assigned to `x`.

```
if ( i > 0 )
{                       /* With braces */
    if ( j > i )
        x = j;
}
else
    x = i;
```

The braces surrounding the inner **if** statement in this example make the **else** clause part of the outer **if** statement. If `i` is less than or equal to 0, `i` is assigned to `x`.

See Also
Reference
The if-else Statement

The C Null Statement

A "null statement" is a statement containing only a semicolon; it can appear wherever a statement is expected. Nothing happens when a null statement is executed. The correct way to code a null statement is:

```
;
```

Remarks

Statements such as **do**, **for**, **if**, and **while** require that an executable statement appear as the statement body. The null statement satisfies the syntax requirement in cases that do not need a substantive statement body.

As with any other C statement, you can include a label before a null statement. To label an item that is not a statement, such as the closing brace of a compound statement, you can label a null statement and insert it immediately before the item to get the same effect.

This example illustrates the null statement:

```
for ( i = 0; i < 10; line[i++] = 0 )
     ;
```

In this example, the loop expression of the **for** statement `line[i++] = 0` initializes the first 10 elements of `line` to 0. The statement body is a null statement, since no further statements are necessary.

See Also

Concepts

Statements (C)

The C return Statement

The **return** statement terminates the execution of a function and returns control to the calling function. Execution resumes in the calling function at the point immediately following the call. A **return** statement can also return a value to the calling function. See Return Type for more information.

Syntax
jump-statement:

return *expression* $_{\text{opt}}$ **;**

The value of *expression*, if present, is returned to the calling function. If *expression* is omitted, the return value of the function is undefined. The expression, if present, is converted to the type returned by the function. If the function was declared with return type **void**, a **return** statement containing an expression generates a warning and the expression is not evaluated.

If no **return** statement appears in a function definition, control automatically returns to the calling function after the last statement of the called function is executed. In this case, the return value of the called function is undefined. If a return value is not required, declare the function to have **void** return type; otherwise, the default return type is **int**.

Many programmers use parentheses to enclose the *expression* argument of the **return** statement. However, C does not require the parentheses.

This example demonstrates the **return** statement:

```
void draw( int I, long L );
long sq( int s );
int main()
{
    long y;
    int x;

    y = sq( x );
    draw( x, y );
    return();
}

long sq( int s )
{
    return( s * s );
}

void draw( int I, long L )
{
    /* Statements defining the draw function here */
    return;
}
```

In this example, the `main` function calls two functions: `sq` and `draw`. The `sq` function returns the value of `x * x` to `main`, where the return value is assigned to `y`. The `draw` function is declared as a **void** function and does not return a value. An attempt to assign the return value of `draw` would cause a diagnostic message to be issued.

See Also
Concepts
Statements (C)

The C switch Statement

The **switch** and **case** statements help control complex conditional and branching operations. The **switch** statement transfers control to a statement within its body.

Syntax
selection-statement:

switch (*expression* **)** *statement*

labeled-statement:

case *constant-expression* **:** *statement*

default : *statement*

Control passes to the statement whose **case** *constant-expression* matches the value of **switch (** *expression* **)**. The **switch** statement can include any number of **case** instances, but no two case constants within the same **switch** statement can have the same value. Execution of the statement body begins at the selected statement and proceeds until the end of the body or until a **break** statement transfers control out of the body.

Use of the **switch** statement usually looks something like this:

switch (*expression*)

{

declarations

.

.

.

case *constant-expression* **:**

statements executed if the expression equals the

value of this constant-expression

.

.

.

break;

default :

statements executed if expression does not equal

any case constant-expression

}

You can use the **break** statement to end processing of a particular case within the **switch** statement and to branch to the end of the **switch** statement. Without **break**, the program continues to the next case, executing the statements until a **break** or the end of the statement is reached. In some situations, this continuation may be desirable.

The **default** statement is executed if no **case** *constant-expression* is equal to the value of **switch (** *expression* **)**. If the **default** statement is omitted, and no **case** match is found, none of the statements in the **switch** body are executed. There can be at most one **default** statement. The **default** statement need not come at the end; it can appear anywhere in the body of the **switch** statement. A **case** or **default** label can only appear inside a **switch** statement.

The type of **switch** *expression* and **case** *constant-expression* must be integral. The value of each **case** *constant-expression* must be unique within the statement body.

The **case** and **default** labels of the **switch** statement body are significant only in the initial test that determines where execution starts in the statement body. Switch statements can be nested. Any static variables are initialized before executing into any **switch** statements.

Note

Declarations can appear at the head of the compound statement forming the **switch** body, but initializations included in the declarations are not performed. The **switch** statement transfers control directly to an executable statement within the body, bypassing the lines that contain initializations.

The following examples illustrate **switch** statements:

```
switch( c )
{
    case 'A':
        capa++;
    case 'a':
        lettera++;
    default :
        total++;
}
```

All three statements of the **switch** body in this example are executed if `c` is equal to `'A'` since a **break** statement does not appear before the following case. Execution control is transferred to the first statement (`capa++;`) and continues in order through the rest of the body. If `c` is equal to `'a'`, `lettera` and `total` are incremented. Only `total` is incremented if `c` is not equal to `'A'` or `'a'`.

```
switch( i )
{
    case -1:
        n++;
        break;
    case 0 :
        z++;
        break;
    case 1 :
        p++;
        break;
}
```

In this example, a **break** statement follows each statement of the **switch** body. The **break** statement forces an exit from the statement body after one statement is executed. If `i` is equal to –1, only `n` is incremented. The **break** following the statement `n++;` causes execution control to pass out of the statement body, bypassing the remaining statements. Similarly, if `i` is equal to 0, only `z` is incremented; if `i` is equal to 1, only `p` is incremented. The final **break** statement is not strictly necessary, since control passes out of the body at the end of the compound statement, but it is included for consistency.

A single statement can carry multiple **case** labels, as the following example shows:

```
case 'a' :
case 'b' :
case 'c' :
case 'd' :
case 'e' :
case 'f' :  hexcvt(c);
```

In this example, if *constant-expression* equals any letter between `'a'` and `'f'`, the `hexcvt` function is called.

Microsoft Specific

Microsoft C does not limit the number of case values in a **switch** statement. The number is limited only by the available memory. ANSI C requires at least 257 case labels be allowed in a **switch** statement.

The default for Microsoft C is that the Microsoft extensions are enabled. Use the /Za compiler option to disable these extensions.

END Microsoft Specific

See Also

Reference

The switch Statement

The try-except Statement

Microsoft Specific

The **try-except** statement is a Microsoft extension to the C language that enables applications to gain control of a program when events that normally terminate execution occur. Such events are called exceptions, and the mechanism that deals with exceptions is called structured exception handling.

Exceptions can be either hardware- or software-based. Even when applications cannot completely recover from hardware or software exceptions, structured exception handling makes it possible to display error information and trap the internal state of the application to help diagnose the problem. This is especially useful for intermittent problems that cannot be reproduced easily.

Syntax
try-except-statement:

__try *compound-statement*

__except (*expression* **)** *compound-statement*

The compound statement after the **__try** clause is the guarded section. The compound statement after the **__except** clause is the exception handler. The handler specifies a set of actions to be taken if an exception is raised during execution of the guarded section. Execution proceeds as follows:

1. The guarded section is executed.
2. If no exception occurs during execution of the guarded section, execution continues at the statement after the **__except** clause.
3. If an exception occurs during execution of the guarded section or in any routine the guarded section calls, the**__except** expression is evaluated and the value returned determines how the exception is handled. There are three values:

   EXCEPTION_CONTINUE_SEARCH Exception is not recognized. Continue to search up the stack for a handler, first for containing **try-except** statements, then for handlers with the next highest precedence.

   EXCEPTION_CONTINUE_EXECUTION Exception is recognized but dismissed. Continue execution at the point where the exception occurred.

   EXCEPTION_EXECUTE_HANDLER Exception is recognized. Transfer control to the exception handler by executing the **__except** compound statement, then continue execution at the point the exception occurred.

Because the **__except** expression is evaluated as a C expression, it is limited to a single value, the conditional-expression operator, or the comma operator. If more extensive processing is required, the expression can call a routine that returns one of the three values listed above.

Note
Structured exception handling works with C and C++ source files. However, it is not specifically designed for C++. You can e nsure that your code is more portable by using C++ exception handling. Also, the C++ exception handling mechanism is mu ch more flexible, in that it can handle exceptions of any type.

Note
For C++ programs, C++ exception handling should be used instead of structured exception handling. For more information, see Exception Handling in the *C++ Language Reference*.

Each routine in an application can have its own exception handler. The **__except** expression executes in the scope of the **__try** body. This means it has access to any local variables declared there.

The **__leave** keyword is valid within a **try-except** statement block. The effect of **__leave** is to jump to the end of the **try-except** block. Execution resumes after the end of the exception handler. Although a **goto** statement can be used to accomplish the same result, a **goto** statement causes stack unwinding. The **__leave** statement is more efficient because it does not involve stack unwinding.

Exiting a **try-except** statement using the **longjmp** run-time function is considered abnormal termination. It is illegal to jump

into a **__try** statement, but legal to jump out of one. The exception handler is not called if a process is killed in the middle of executing a **try-except** statement.

Example

Following is an example of an exception handler and a termination handler. See The try-finally Statement for more information about termination handlers.

```
.
.
.
puts("hello");
__try{
   puts("in try");
   __try{
      puts("in try");
      RAISE_AN_EXCEPTION();
   }__finally{
      puts("in finally");
   }
}__except( puts("in filter"), EXCEPTION_EXECUTE_HANDLER ){
   puts("in except");
}
puts("world");
```

This is the output from the example, with commentary added on the right:

```
hello
in try              /* fall into try                     */
in try              /* fall into nested try                */
in filter           /* execute filter; returns 1 so accept  */
in finally          /* unwind nested finally                */
in except           /* transfer control to selected handler */
world               /* flow out of handler                  */
```

END Microsoft Specific

See Also

Reference

try-except Statement

The try-finally Statement

Microsoft Specific

The **try-finally** statement is a Microsoft extension to the C language that enables applications to guarantee execution of cleanup code when execution of a block of code is interrupted. Cleanup consists of such tasks as deallocating memory, closing files, and releasing file handles. The **try-finally** statement is especially useful for routines that have several places where a check is made for an error that could cause premature return from the routine.

try-finally-statement:

__try *compound-statement*

__finally *compound-statement*

The compound statement after the **__try** clause is the guarded section. The compound statement after the **__finally** clause is the termination handler. The handler specifies a set of actions that execute when the guarded section is exited, whether the guarded section is exited by an exception (abnormal termination) or by standard fall through (normal termination).

Control reaches a **__try** statement by simple sequential execution (fall through). When control enters the **__try** statement, its associated handler becomes active. Execution proceeds as follows:

1. The guarded section is executed.
2. The termination handler is invoked.
3. When the termination handler completes, execution continues after the **__finally** statement. Regardless of how the guarded section ends (for example, via a **goto** statement out of the guarded body or via a **return** statement), the termination handler is executed before the flow of control moves out of the guarded section.

The **__leave** keyword is valid within a **try-finally** statement block. The effect of **__leave** is to jump to the end of the **try-finally** block. The termination handler is immediately executed. Although a **goto** statement can be used to accomplish the same result, a **goto** statement causes stack unwinding. The **__leave** statement is more efficient because it does not involve stack unwinding.

Exiting a **try-finally** statement using a **return** statement or the **longjmp** run-time function is considered abnormal termination. It is illegal to jump into a **__try** statement, but legal to jump out of one. All **__finally** statements that are active between the point of departure and the destination must be run. This is called a "local unwind."

The termination handler is not called if a process is killed while executing a **try-finally** statement.

Note
Structured exception handling works with C and C++ source files. However, it is not specifically designed for C++. You can e nsure that your code is more portable by using C++ exception handling. Also, the C++ exception handling mechanism is mu ch more flexible, in that it can handle exceptions of any type.

Note
For C++ programs, C++ exception handling should be used instead of structured exception handling. For more information, see Exception Handling in the *C++ Language Reference*.

See the example for the try-except statement to see how the **try-finally** statement works.

END Microsoft Specific

See Also
Reference
try-finally Statement

The C while Statement

The **while** statement lets you repeat a statement until a specified expression becomes false.

Syntax
iteration-statement:

while (*expression* **)** *statement*

The *expression* must have arithmetic or pointer type. Execution proceeds as follows:

1. The *expression* is evaluated.
2. If *expression* is initially false, the body of the **while** statement is never executed, and control passes from the **while** statement to the next statement in the program.

   If *expression* is true (nonzero), the body of the statement is executed and the process is repeated beginning at step 1.

The **while** statement can also terminate when a **break**, **goto**, or **return** within the statement body is executed. Use the **continue** statement to terminate an iteration without exiting the **while** loop. The **continue** statement passes control to the next iteration of the **while** statement.

This is an example of the **while** statement:

```
while ( i >= 0 )
{
    string1[i] = string2[i];
    i--;
}
```

This example copies characters from `string2` to `string1`. If `i` is greater than or equal to 0, `string2[i]` is assigned to `string1[i]` and `i` is decremented. When `i` reaches or falls below 0, execution of the **while** statement terminates.

See Also
Reference
The while Statement

Functions (C)

The function is the fundamental modular unit in C. A function is usually designed to perform a specific task, and its name often reflects that task. A function contains declarations and statements. This section describes how to declare, define, and call C functions. Other topics discussed are:

- Overview of functions
- Function attributes
- Specifying calling conventions
- Inline functions
- DLL export and import functions
- Naked functions
- Storage class
- Return type
- Arguments
- Parameters

See Also

Reference

C Language Reference

Overview of Functions

Functions must have a definition and should have a declaration, although a definition can serve as a declaration if the declaration appears before the function is called. The function definition includes the function body — the code that executes when the function is called.

A function declaration establishes the name, return type, and attributes of a function that is defined elsewhere in the program. A function declaration must precede the call to the function. This is why the header files containing the declarations for the run-time functions are included in your code before a call to a run-time function. If the declaration has information about the types and number of parameters, the declaration is a prototype. See Function Prototypes for more information.

The compiler uses the prototype to compare the types of arguments in subsequent calls to the function with the function's parameters and to convert the types of the arguments to the types of the parameters whenever necessary.

A function call passes execution control from the calling function to the called function. The arguments, if any, are passed by value to the called function. Execution of a **return** statement in the called function returns control and possibly a value to the calling function.

See Also

Concepts

Functions (C)

Obsolete Forms of Function Declarations and Definitions

The old-style function declarations and definitions use slightly different rules for declaring parameters than the syntax recommended by the ANSI C standard. First, the old-style declarations don't have a parameter list. Second, in the function definition, the parameters are listed, but their types are not declared in the parameter list. The type declarations precede the compound statement constituting the function body. The old-style syntax is obsolete and should not be used in new code. Code using the old-style syntax is still supported, however. This example illustrates the obsolete forms of declarations and definitions:

```
double old_style();           /* Obsolete function declaration */

double alt_style( a , real )  /* Obsolete function definition */
    double *real;
    int a;
{
    return ( *real + a ) ;
}
```

Functions returning an integer or pointer with the same size as an **int** are not required to have a declaration although the declaration is recommended.

To comply with the ANSI C standard, old-style function declarations using an ellipsis now generate an error when compiling with the /Za option and a level 4 warning when compiling with /Ze. For example:

```
void funct1( a, ... )  /* Generates a warning under /Ze or */
int a;                 /* an error when compiling with /Za */
{
}
```

You should rewrite this declaration as a prototype:

```
void funct1( int a, ... )
{
}
```

Old-style function declarations also generate warnings if you subsequently declare or define the same function with either an ellipsis or a parameter with a type that is not the same as its promoted type.

The next section, C Function Definitions, shows the syntax for function definitions, including the old-style syntax. The nonterminal for the list of parameters in the old-style syntax is *identifier-list*.

See Also

Concepts

Overview of Functions

C Function Definitions

A function definition specifies the name of the function, the types and number of parameters it expects to receive, and its return type. A function definition also includes a function body with the declarations of its local variables, and the statements that determine what the function does.

Syntax

translation-unit:

external-declaration

translation-unit external-declaration

external-declaration: /* Allowed only at external (file) scope */

function-definition

declaration

function-definition: /* Declarator here is the function declarator */

declaration-specifiers $_{\text{opt}}$ *attribute-seq* $_{\text{opt}}$ *declarator declaration-list* $_{\text{opt}}$ *compound-statement*

/* *attribute-seq* is Microsoft Specific */

Prototype parameters are:

declaration-specifiers:

storage-class-specifier declaration-specifiers $_{\text{opt}}$

type-specifier declaration-specifiers $_{\text{opt}}$

type-qualifier declaration-specifiers $_{\text{opt}}$

declaration-list:

declaration

declaration-list declaration

declarator:

pointer $_{\text{opt}}$ *direct-declarator*

direct-declarator: /* A function declarator */

direct-declarator **(** *parameter-type-list* **)** /* New-style declarator */

direct-declarator **(** *identifier-list* $_{\text{opt}}$ **)** /* Obsolete-style declarator */

The parameter list in a definition uses this syntax:

parameter-type-list: /* The parameter list */

parameter-list

parameter-list **, ...**

parameter-list:

parameter-declaration

parameter-list **,** *parameter-declaration*

parameter-declaration:

declaration-specifiers declarator

declaration-specifiers abstract declarator $_{\text{opt}}$

The parameter list in an old-style function definition uses this syntax:

identifier-list: /* Used in obsolete-style function definitions and declarations */

identifier

identifier-list **,** *identifier*

The syntax for the function body is:

compound-statement: /* The function body */

{ *declaration-list* $_{\text{opt}}$ *statement-list* $_{\text{opt}}$ **}**

The only storage-class specifiers that can modify a function declaration are **extern** and **static**. The **extern** specifier signifies that the function can be referenced from other files; that is, the function name is exported to the linker. The **static** specifier signifies that the function cannot be referenced from other files; that is, the name is not exported by the linker. If no storage class appears in a function definition, **extern** is assumed. In any case, the function is always visible from the definition point to the end of the file.

The optional *declaration-specifiers* and mandatory *declarator* together specify the function's return type and name. The *declarator* is a combination of the identifier that names the function and the parentheses following the function name. The optional *attribute-seq* nonterminal is a Microsoft-specific feature defined in Function Attributes.

The *direct-declarator* (in the *declarator* syntax) specifies the name of the function being defined and the identifiers of its parameters. If the *direct-declarator* includes a *parameter-type-list*, the list specifies the types of all the parameters. Such a declarator also serves as a function prototype for later calls to the function.

A *declaration* in the *declaration-list* in function definitions cannot contain a *storage-class-specifier* other than **register**. The *type-specifier* in the *declaration-specifiers* syntax can be omitted only if the **register** storage class is specified for a value of **int** type.

The *compound-statement* is the function body containing local variable declarations, references to externally declared items, and statements.

The sections Function Attributes, Storage Class, Return Type, Parameters, and Function Body describe the components of the function definition in detail.

See Also
Concepts
Functions (C)

Function Attributes

Microsoft Specific

The optional *attribute-seq* nonterminal allows you to select a calling convention on a per-function basis. You can also specify functions as **__fastcall** or **__inline**.

END Microsoft Specific

See Also

Reference

C Function Definitions

Specifying Calling Conventions

Microsoft Specific

For information on calling conventions, see Calling Conventions Topics.

END Microsoft Specific

See Also

Concepts

Function Attributes

Inline Functions

Microsoft Specific

The **__inline** keyword tells the compiler to substitute the code within the function definition for every instance of a function call. However, substitution occurs only at the compiler's discretion. For example, the compiler does not inline a function if its address is taken or if it is too large to inline.

For a function to be considered as a candidate for inlining, it must use the new-style function definition.

Use this form to specify an inline function:

```
        __inline type <SUB>opt</SUB> function-definition;
```

The use of inline functions generates faster code and can sometimes generate smaller code than the equivalent function call generates for the following reasons:

- It saves the time required to execute function calls.
- Small inline functions, perhaps three lines or less, create less code than the equivalent function call because the compiler doesn't generate code to handle arguments and a return value.
- Functions generated inline are subject to code optimizations not available to normal functions because the compiler does not perform interprocedural optimizations.

Functions using **__inline** should not be confused with inline assembler code. See Inline Assembler for more information.

END Microsoft Specific

See Also

Reference

inline, __inline, __forceinline

The C Inline Assembler

Microsoft Specific

The inline assembler lets you embed assembly-language instructions directly in your C source programs without extra assembly and link steps. The inline assembler is built into the compiler — you don't need a separate assembler such as the Microsoft Macro Assembler (MASM).

Because the inline assembler doesn't require separate assembly and link steps, it is more convenient than a separate assembler. Inline assembly code can use any C variable or function name that is in scope, so it is easy to integrate it with your program's C code. And because the assembly code can be mixed with C statements, it can do tasks that are cumbersome or impossible in C alone.

The **__asm** keyword invokes the inline assembler and can appear wherever a C statement is legal. It cannot appear by itself. It must be followed by an assembly instruction, a group of instructions enclosed in braces, or, at the very least, an empty pair of braces. The term "**__asm** block" here refers to any instruction or group of instructions, whether or not in braces.

The code below is a simple **__asm** block enclosed in braces. (The code is a custom function prolog sequence.)

```
__asm
{
   push ebp
   mov  ebp, esp
   sub  esp, __LOCAL_SIZE
}
```

Alternatively, you can put **__asm** in front of each assembly instruction:

```
__asm push ebp
__asm mov  ebp, esp
__asm sub  esp, __LOCAL_SIZE
```

Since the **__asm** keyword is a statement separator, you can also put assembly instructions on the same line:

```
__asm push ebp   __asm mov  ebp, esp   __asm sub  esp, __LOCAL_SIZE
```

END Microsoft Specific

See Also
Concepts
Function Attributes

DLL Import and Export Functions

Microsoft Specific

The most complete and up-to-date information on this topic can be found in dllexport, dllimport.

The **dllimport** and **dllexport** storage-class modifiers are Microsoft-specific extensions to the C language. These modifiers explicitly define the DLL's interface to its client (the executable file or another DLL). Declaring functions as **dllexport** eliminates the need for a module-definition (.DEF) file. You can also use the **dllimport** and **dllexport** modifiers with data and objects.

The **dllimport** and **dllexport** storage-class modifiers must be used with the extended attribute syntax keyword, **__declspec**, as shown in this example:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllExport void func();
DllExport int i = 10;
DllExport int j;
DllExport int n;
```

For specific information about the syntax for extended storage-class modifiers, see Extended Storage-Class Attributes.

END Microsoft Specific

See Also

Reference

C Function Definitions

Definitions and Declarations (C)

Microsoft Specific

The DLL interface refers to all items (functions and data) that are known to be exported by some program in the system; that is, all items that are declared as **dllimport** or **dllexport**. All declarations included in the DLL interface must specify either the **dllimport** or **dllexport** attribute. However, the definition can specify only the **dllexport** attribute. For example, the following function definition generates a compiler error:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllImport int func()    /* Error; dllimport prohibited in */
                        /* definition. */
{
   return 1;
}
```

This code also generates an error:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllImport int i = 10;      /* Error; this is a definition. */
```

However, this is correct syntax:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllExport int i = 10;      /* Okay: this is an export definition. */
```

The use of **dllexport** implies a definition, while **dllimport** implies a declaration. You must use the **extern** keyword with **dllexport** to force a declaration; otherwise, a definition is implied.

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

extern DllImport int k;   /* These are correct and imply */
Dllimport int j;          /* a declaration. */
```

END Microsoft Specific

See Also

Concepts

DLL Import and Export Functions

Defining Inline C Functions with dllexport and dllimport

Microsoft Specific

You can define as inline a function with the **dllexport** attribute. In this case, the function is always instantiated and exported, whether or not any module in the program references the function. The function is presumed to be imported by another program.

You can also define as inline a function declared with the **dllimport** attribute. In this case, the function can be expanded (subject to the /Ob (inline) compiler option specification) but never instantiated. In particular, if the address of an inline imported function is taken, the address of the function residing in the DLL is returned. This behavior is the same as taking the address of a non-inline imported function.

Static local data and strings in inline functions maintain the same identities between the DLL and client as they would in a single program (that is, an executable file without a DLL interface).

Exercise care when providing imported inline functions. For example, if you update the DLL, don't assume that the client will use the changed version of the DLL. To ensure that you are loading the proper version of the DLL, rebuild the DLL's client as well.

END Microsoft Specific

See Also

Concepts

DLL Import and Export Functions

Rules and Limitations for dllimport/dllexport

Microsoft Specific

- If you declare a function without the **dllimport** or **dllexport** attribute, the function is not considered part of the DLL interface. Therefore, the definition of the function must be present in that module or in another module of the same program. To make the function part of the DLL interface, you must declare the definition of the function in the other module as **dllexport**. Otherwise, a linker error is generated when the client is built.
- If a single module in your program contains **dllimport** and **dllexport** declarations for the same function, the **dllexport** attribute takes precedence over the **dllimport** attribute. However, a compiler warning is generated. For example:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

   DllImport void func1( void );
   DllExport void func1( void );   /* Warning; dllexport */
                                   /* takes precedence. */
```

- You cannot initialize a static data pointer with the address of a data object declared with the **dllimport** attribute. For example, the following code generates errors:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

   DllImport int i;
   .
   .
   .
   int *pi = &i;                          /* Error */

   void func2()
   {
      static int *pi = &i;                  /* Error */
   }
```

- Initializing a static function pointer with the address of a function declared with **dllimport** sets the pointer to the address of the DLL import thunk (a code stub that transfers control to the function) rather than the address of the function. This assignment does not generate an error message:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

   DllImport void func1( void
   .
   .
   .
   static void ( *pf )( void ) = &func1;   /* No Error */

   void func2()
   {
      static void ( *pf )( void ) = &func1;  /* No Error */
   }
```

- Because a program that includes the **dllexport** attribute in the declaration of an object must provide the definition for that object, you can initialize a global or local static function pointer with the address of a **dllexport** function. Similarly, you can initialize a global or local static data pointer with the address of a **dllexport** data object. For example:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

   DllImport void func1( void );
   DllImport int i;

   DllExport void func1( void );
   DllExport int i;
   .
   .
   .
   int *pi = &i;                            /* Okay */
   static void ( *pf )( void ) = &func1;    /* Okay */

   void func2()
   {
      static int *pi = i;                     /* Okay */
      static void ( *pf )( void ) = &func1;   /* Okay */
   }
```

END Microsoft Specific

See Also

Concepts

DLL Import and Export Functions

Naked Functions

Microsoft Specific

The **naked** storage-class attribute is a Microsoft-specific extension to the C language. For functions declared with the **naked** storage-class attribute, the compiler generates code without prolog and epilog code. You can use this feature to write your own prolog/epilog code sequences using inline assembler code. Naked functions are particularly useful in writing virtual device drivers.

Because the **naked** attribute is only relevant to the definition of a function and is not a type modifier, naked functions use the extended attribute syntax, described in Extended Storage-Class Attributes.

The following example defines a function with the **naked** attribute:

```
__declspec( naked ) int func( formal_parameters )
{
   /* Function body */
}
```

Or, alternatively:

```
#define Naked   __declspec( naked )

Naked int func( formal_parameters )
{
   /* Function body */
}
```

The **naked** attribute affects only the nature of the compiler's code generation for the function's prolog and epilog sequences. It does not affect the code that is generated for calling such functions. Thus, the **naked** attribute is not considered part of the function's type, and function pointers cannot have the **naked** attribute. Furthermore, the **naked** attribute cannot be applied to a data definition. For example, the following code generates errors:

```
__declspec( naked ) int i;  /* Error--naked attribute not */
                            /* permitted on data declarations. */
```

The **naked** attribute is relevant only to the definition of the function and cannot be specified in the function's prototype. The following declaration generates a compiler error:

```
__declspec( naked ) int func();   /* Error--naked attribute not */
                     /* permitted on function declarations.    */   \
```

END Microsoft Specific

See Also

Reference

C Function Definitions

Rules and Limitations for Using Naked Functions

For information on rules and limitations for using naked functions, see the corresponding topic in the C++ language reference: Rules and Limitations for Naked Functions.

See Also

Concepts

Naked Functions

Considerations When Writing Prolog/Epilog Code

Microsoft Specific

Before writing your own prolog and epilog code sequences, it is important to understand how the stack frame is laid out. It is also useful to know how to use the **__LOCAL_SIZE** predefined constant.

C Stack Frame Layout

This example shows the standard prolog code that might appear in a 32-bit function:

```
push     ebp                ; Save ebp
mov      ebp, esp           ; Set stack frame pointer
sub      esp, localbytes    ; Allocate space for locals
push     <registers>        ; Save registers
```

The `localbytes` variable represents the number of bytes needed on the stack for local variables, and the `registers` variable is a placeholder that represents the list of registers to be saved on the stack. After pushing the registers, you can place any other appropriate data on the stack. The following is the corresponding epilog code:

```
pop      <registers>        ; Restore registers
mov      esp, ebp           ; Restore stack pointer
pop      ebp                ; Restore ebp
ret                         ; Return from function
```

The stack always grows down (from high to low memory addresses). The base pointer (`ebp`) points to the pushed value of `ebp`. The local variables area begins at `ebp-2`. To access local variables, calculate an offset from `ebp` by subtracting the appropriate value from `ebp`.

The __LOCAL_SIZE Constant

The compiler provides a constant, **__LOCAL_SIZE**, for use in the inline assembler block of function prolog code. This constant is used to allocate space for local variables on the stack frame in custom prolog code.

The compiler determines the value of **__LOCAL_SIZE**. The value is the total number of bytes of all user-defined local variables and compiler-generated temporary variables. **__LOCAL_SIZE** can be used only as an immediate operand; it cannot be used in an expression. You must not change or redefine the value of this constant. For example:

```
mov      eax, __LOCAL_SIZE          ;Immediate operand--Okay
mov      eax, [ebp - __LOCAL_SIZE]   ;Error
```

The following example of a naked function containing custom prolog and epilog sequences uses **__LOCAL_SIZE** in the prolog sequence:

```
__declspec ( naked ) func()
{
   int i;
   int j;

   __asm      /* prolog */
      {
      push   ebp
      mov      ebp, esp
      sub      esp, __LOCAL_SIZE
      }

   /* Function body */

   __asm      /* epilog */
      {
      mov      esp, ebp
      pop      ebp
```

```
        ret
        }
}
```

END Microsoft Specific

See Also

Concepts

Naked Functions

Storage Class

The storage-class specifier in a function definition gives the function either **extern** or **static** storage class.

Syntax

function-definition:

declaration-specifiers $_{opt}$ *attribute-seq* $_{opt}$ *declarator declaration-list* $_{opt}$ *compound-statement*

/* *attribute-seq* is Microsoft Specific */

declaration-specifiers:

storage-class-specifier declaration-specifiers $_{opt}$

type-specifier declaration-specifiers $_{opt}$

type-qualifier declaration-specifiers $_{opt}$

storage-class-specifier: /* For function definitions */

extern

static

If a function definition does not include a *storage-class-specifier*, the storage class defaults to **extern**. You can explicitly declare a function as **extern**, but it is not required.

If the declaration of a function contains the *storage-class-specifier* **extern**, the identifier has the same linkage as any visible declaration of the identifier with file scope. If there is no visible declaration with file scope, the identifier has external linkage. If an identifier has file scope and no *storage-class-specifier*, the identifier has external linkage. External linkage means that each instance of the identifier denotes the same object or function. See Lifetime, Scope, Visibility, and Linkage for more information about linkage and file scope.

Block-scope function declarations with a storage-class specifier other than **extern** generate errors.

A function with **static** storage class is visible only in the source file in which it is defined. All other functions, whether they are given **extern** storage class explicitly or implicitly, are visible throughout all source files in the program. If **static** storage class is desired, it must be declared on the first occurrence of a declaration (if any) of the function, and on the definition of the function.

Microsoft Specific

When the Microsoft extensions are enabled, a function originally declared without a storage class (or with **extern** storage class) is given **static** storage class if the function definition is in the same source file and if the definition explicitly specifies **static** storage class.

When compiling with the /Ze compiler option, functions declared within a block using the **extern** keyword have global visibility. This is not true when compiling with /Za. This feature should not be relied upon if portability of source code is a consideration.

END Microsoft Specific

See Also

Reference

C Function Definitions

Return Type

The return type of a function establishes the size and type of the value returned by the function and corresponds to the type-specifier in the syntax below:

Syntax

function-definition:

declaration-specifiers $_{\text{opt}}$ *attribute-seq* $_{\text{opt}}$ *declarator declaration-list* $_{\text{opt}}$ *compound-statement*

/* *attribute-seq* is Microsoft Specific */

declaration-specifiers:

storage-class-specifier declaration-specifiers $_{\text{opt}}$

type-specifier declaration-specifiers $_{\text{opt}}$

type-qualifier declaration-specifiers $_{\text{opt}}$

type-specifier:

void

char

short

int

long

float

double

signed

unsigned

struct-or-union-specifier

enum-specifier

typedef-name

The *type-specifier* can specify any fundamental, structure, or union type. If you do not include *type-specifier*, the return type **int** is assumed.

The return type given in the function definition must match the return type in declarations of the function elsewhere in the program. A function returns a value when a **return** statement containing an expression is executed. The expression is evaluated, converted to the return value type if necessary, and returned to the point at which the function was called. If a function is declared with return type **void**, a return statement containing an expression generates a warning and the expression is not evaluated.

The following examples illustrate function return values.

```
typedef struct
{
    char name[20];
    int id;
    long class;
} STUDENT;

/* Return type is STUDENT: */

STUDENT sortstu( STUDENT a, STUDENT b )
{
    return ( (a.id < b.id) ? a : b );
}
```

This example defines the `STUDENT` type with a **typedef** declaration and defines the function `sortstu` to have `STUDENT` return type. The function selects and returns one of its two structure arguments. In subsequent calls to the function, the compiler checks to make sure the argument types are `STUDENT`.

Note

Efficiency would be enhanced by passing pointers to the structure, rather than the entire structure.

```
char *smallstr( char s1[], char s2[] )
{
    int i;

    i = 0;
    while ( s1[i] != '\0' && s2[i] != '\0' )
        i++;
    if ( s1[i] == '\0' )
        return ( s1 );
    else
        return ( s2 );
}
```

This example defines a function returning a pointer to an array of characters. The function takes two character arrays (strings) as arguments and returns a pointer to the shorter of the two strings. A pointer to an array points to the first of the array elements and has its type; thus, the return type of the function is a pointer to type **char**.

You need not declare functions with **int** return type before you call them, although prototypes are recommended so that correct type checking for arguments and return values is enabled.

See Also

Reference

C Function Definitions

Parameters

Arguments are names of values passed to a function by a function call. Parameters are the values the function expects to receive. In a function prototype, the parentheses following the function name contain a complete list of the function's parameters and their types. Parameter declarations specify the types, sizes, and identifiers of values stored in the parameters.

Syntax

function-definition:

declaration-specifiers $_{\text{opt}}$ *attribute-seq* $_{\text{opt}}$ *declarator declaration-list* $_{\text{opt}}$ *compound-statement*

/* *attribute-seq* is Microsoft Specific */

declarator :

pointer $_{\text{opt}}$ *direct-declarator*

direct-declarator:/* A function declarator */

direct-declarator **(** *parameter-type-list* **)** /* New-style declarator */

parameter-type-list: /* A parameter list */

parameter-list

parameter-list **,...**

parameter-list:

parameter-declaration

parameter-list **,** *parameter-declaration*

parameter-declaration:

declaration-specifiers declarator

declaration-specifiers abstract-declarator $_{\text{opt}}$

The *parameter-type-list* is a sequence of parameter declarations separated by commas. The form of each parameter in a parameter list looks like this:

```
[register]  type-specifier [declarator]
```

Function parameters declared with the **auto** attribute generate errors. The identifiers of the parameters are used in the function body to refer to the values passed to the function. You can name the parameters in a prototype, but the names go out of scope at the end of the declaration. Therefore parameter names can be assigned the same way or differently in the function definition. These identifiers cannot be redefined in the outermost block of the function body, but they can be redefined in inner, nested blocks as though the parameter list were an enclosing block.

Each identifier in *parameter-type-list* must be preceded by its appropriate type specifier, as shown in this example:

```
void new( double x, double y, double z )
{
    /* Function body here */
}
```

If at least one parameter occurs in the parameter list, the list can end with a comma followed by three periods (**, ...**). This construction, called the "ellipsis notation," indicates a variable number of arguments to the function. (See Calls with a Variable Number of Arguments for more information.) However, a call to the function must have at least as many arguments as there are parameters before the last comma.

If no arguments are to be passed to the function, the list of parameters is replaced by the keyword **void**. This use of **void** is distinct from its use as a type specifier.

The order and type of parameters, including any use of the ellipsis notation, must be the same in all the function declarations (if any) and in the function definition. The types of the arguments after usual arithmetic conversions must be assignment-

compatible with the types of the corresponding parameters. (See Usual Arithmetic Conversions for information on arithmetic conversions.) Arguments following the ellipsis are not checked. A parameter can have any fundamental, structure, union, pointer, or array type.

The compiler performs the usual arithmetic conversions independently on each parameter and on each argument, if necessary. After conversion, no parameter is shorter than an **int**, and no parameter has **float** type unless the parameter type is explicitly specified as **float** in the prototype. This means, for example, that declaring a parameter as a **char** has the same effect as declaring it as an **int**.

See Also

Reference

C Function Definitions

Function Body

A "function body" is a compound statement containing the statements that specify what the function does.

Syntax

function-definition:

declaration-specifiers $_{\text{opt}}$ *attribute-seq* $_{\text{opt}}$ *declarator declaration-list* $_{\text{opt}}$ *compound-statement*

/* *attribute-seq* is Microsoft Specific */

compound-statement: /* The function body */

{ *declaration-list* $_{\text{opt}}$ *statement-list* $_{\text{opt}}$ **}**

Variables declared in a function body, "local variables," have **auto** storage class unless otherwise specified. When the function is called, storage is created for the local variables and local initializations are performed. Execution control passes to the first statement in *compound-statement* and continues until a **return** statement is executed or the end of the function body is encountered. Control then returns to the point at which the function was called.

A **return** statement containing an expression must be executed if the function is to return a value. The return value of a function is undefined if no **return** statement is executed or if the **return** statement does not include an expression.

See Also

Reference

C Function Definitions

Function Prototypes

A function declaration precedes the function definition and specifies the name, return type, storage class, and other attributes of a function. To be a prototype, the function declaration must also establish types and identifiers for the function's arguments.

Syntax

declaration:

declaration-specifiers attribute-seq $_{opt}$ *init-declarator-list* $_{opt}$**;**

/* *attribute-seq* $_{opt}$ is Microsoft Specific */

declaration-specifiers:

storage-class-specifier declaration-specifiers $_{opt}$

type-specifier declaration-specifiers $_{opt}$

type-qualifier declaration-specifiers $_{opt}$

init-declarator-list:

init-declarator

init-declarator-list **,** *init-declarator*

init-declarator:

declarator

declarator = *initializer*

declarator:

pointer $_{opt}$ *direct-declarator*

direct-declarator: /* A function declarator */

direct-declarator **(** *parameter-type-list* **)** /* New-style declarator */

direct-declarator **(** *identifier-list* $_{opt}$ **)** /* Obsolete-style declarator */

The prototype has the same form as the function definition, except that it is terminated by a semicolon immediately following the closing parenthesis and therefore has no body. In either case, the return type must agree with the return type specified in the function definition.

Function prototypes have the following important uses:

- They establish the return type for functions that return types other than **int**. Although functions that return **int** values do not require prototypes, prototypes are recommended.
- Without complete prototypes, standard conversions are made, but no attempt is made to check the type or number of arguments with the number of parameters.
- Prototypes are used to initialize pointers to functions before those functions are defined.
- The parameter list is used for checking the correspondence of arguments in the function call with the parameters in the function definition.

The converted type of each parameter determines the interpretation of the arguments that the function call places on the stack. A type mismatch between an argument and a parameter may cause the arguments on the stack to be misinterpreted. For example, on a 16-bit computer, if a 16-bit pointer is passed as an argument, then declared as a **long** parameter, the first 32 bits on the stack are interpreted as a **long** parameter. This error creates problems not only with the **long** parameter, but with any parameters that follow it. You can detect errors of this kind by declaring complete function prototypes for all functions.

A prototype establishes the attributes of a function so that calls to the function that precede its definition (or occur in other source files) can be checked for argument-type and return-type mismatches. For example, if you specify the **static** storage-class specifier in a prototype, you must also specify the **static** storage class in the function definition.

Complete parameter declarations (`int a`) can be mixed with abstract declarators (`int`) in the same declaration. For example, the following declaration is legal:

```
int add( int a, int );
```

The prototype can include both the type of, and an identifier for, each expression that is passed as an argument. However, such identifiers have scope only until the end of the declaration. The prototype can also reflect the fact that the number of arguments is variable, or that no arguments are passed. Without such a list, mismatches may not be revealed, so the compiler cannot generate diagnostic messages concerning them. See Arguments for more information on type checking.

Prototype scope in the Microsoft C compiler is now ANSI-compliant when compiling with the /Za compiler option. This means that if you declare a **struct** or **union** tag within a prototype, the tag is entered at that scope rather than at global scope. For example, when compiling with /Za for ANSI compliance, you can never call this function without getting a type mismatch error:

```
void func1( struct S * );
```

To correct your code, define or declare the **struct** or **union** at global scope before the function prototype:

```
struct S;
void func1( struct S * );
```

Under /Ze, the tag is still entered at global scope.

See Also

Concepts

Functions (C)

Function Calls

A function call is an expression that passes control and arguments (if any) to a function and has the form

```
        expression ( expression-list <SUB>opt</SUB> )
```

where *expression* is a function name or evaluates to a function address and *expression-list* is a list of expressions (separated by commas). The values of these latter expressions are the arguments passed to the function. If the function does not return a value, then you declare it to be a function that returns **void**.

If a declaration exists before the function call, but no information is given concerning the parameters, any undeclared arguments simply undergo the usual arithmetic conversions.

Note

The expressions in the function argument list can be evaluated in any order, so arguments whose values may be changed by side effects from another argument have undefined values. The sequence point defined by the function-call operator guarant ees only that all side effects in the argument list are evaluated before control passes to the called function. (Note that the ord er in which arguments are pushed on the stack is a separate matter.) See Sequence Points for more information.

The only requirement in any function call is that the expression before the parentheses must evaluate to a function address. This means that a function can be called through any function-pointer expression.

Example

This example illustrates function calls called from a **switch** statement:

```
int main()
{
    /* Function prototypes */

    long lift( int ), step( int ), drop( int );
    void work( int number, long (*function)(int i) );

    int select, count;
    .
    .
    .
    select = 1;
    switch( select )
    {
        case 1: work( count, lift );
                break;

        case 2: work( count, step );
                break;

        case 3: work( count, drop );
                /* Fall through to next case */
        default:
                break;
    }
}

/* Function definition */

void work( int number, long (*function)(int i) )
{
    int i;
    long j;

    for ( i = j = 0; i < number; i++ )
```

```
            j += ( *function )( i );
}
```

In this example, the function call in `main`,

```
work( count, lift );
```

passes an integer variable, `count`, and the address of the function `lift` to the function `work`. Note that the function address is passed simply by giving the function identifier, since a function identifier evaluates to a pointer expression. To use a function identifier in this way, the function must be declared or defined before the identifier is used; otherwise, the identifier is not recognized. In this case, a prototype for `work` is given at the beginning of the `main` function.

The parameter `function` in `work` is declared to be a pointer to a function taking one **int** argument and returning a **long** value. The parentheses around the parameter name are required; without them, the declaration would specify a function returning a pointer to a **long** value.

The function `work` calls the selected function from inside the **for** loop by using the following function call:

```
( *function )( i );
```

One argument, `i`, is passed to the called function.

See Also

Concepts

Functions (C)

Arguments

The arguments in a function call have this form:

```
        expression ( expression-list <SUB>opt</SUB> )  /* Function call */
```

In a function call, *expression-list* is a list of expressions (separated by commas). The values of these latter expressions are the arguments passed to the function. If the function takes no arguments, *expression-list* should contain the keyword **void**.

An argument can be any value with fundamental, structure, union, or pointer type. All arguments are passed by value. This means a copy of the argument is assigned to the corresponding parameter. The function does not know the actual memory location of the argument passed. The function uses this copy without affecting the variable from which it was originally derived.

Although you cannot pass arrays or functions as arguments, you can pass pointers to these items. Pointers provide a way for a function to access a value by reference. Since a pointer to a variable holds the address of the variable, the function can use this address to access the value of the variable. Pointer arguments allow a function to access arrays and functions, even though arrays and functions cannot be passed as arguments.

The order in which arguments are evaluated can vary under different compilers and different optimization levels. However, the arguments and any side effects are completely evaluated before the function is entered. See Side Effects for information on side effects.

The *expression-list* in a function call is evaluated and the usual arithmetic conversions are performed on each argument in the function call. If a prototype is available, the resulting argument type is compared to the prototype's corresponding parameter. If they do not match, either a conversion is performed, or a diagnostic message is issued. The parameters also undergo the usual arithmetic conversions.

The number of expressions in *expression-list* must match the number of parameters, unless the function's prototype or definition explicitly specifies a variable number of arguments. In this case, the compiler checks as many arguments as there are type names in the list of parameters and converts them, if necessary, as described above. See Calls with a Variable Number of Arguments for more information.

If the prototype's parameter list contains only the keyword **void**, the compiler expects zero arguments in the function call and zero parameters in the definition. A diagnostic message is issued if it finds any arguments.

Example

This example uses pointers as arguments:

```
int main()
{
    /* Function prototype */

    void swap( int *num1, int *num2 );
    int x, y;
    .
    .
    .
    swap( &x, &y );  /* Function call */
}

/* Function definition */

void swap( int *num1, int *num2 )
{
    int t;

    t = *num1;
    *num1 = *num2;
    *num2 = t;
}
```

In this example, the `swap` function is declared in `main` to have two arguments, represented respectively by identifiers `num1` and `num2`, both of which are pointers to **int** values. The parameters `num1` and `num2` in the prototype-style definition are also declared as pointers to **int** type values.

In the function call

```
swap( &x, &y )
```

the address of `x` is stored in `num1` and the address of `y` is stored in `num2`. Now two names, or "aliases," exist for the same location. References to `*num1` and `*num2` in `swap` are effectively references to `x` and `y` in `main`. The assignments within `swap` actually exchange the contents of `x` and `y`. Therefore, no **return** statement is necessary.

The compiler performs type checking on the arguments to `swap` because the prototype of `swap` includes argument types for each parameter. The identifiers within the parentheses of the prototype and definition can be the same or different. What is important is that the types of the arguments match those of the parameter lists in both the prototype and the definition.

See Also

Concepts

Function Calls

Calls with a Variable Number of Arguments

A partial parameter list can be terminated by the ellipsis notation, a comma followed by three periods (**, ...**), to indicate that there may be more arguments passed to the function, but no more information is given about them. Type checking is not performed on such arguments. At least one parameter must precede the ellipsis notation and the ellipsis notation must be the last token in the parameter list. Without the ellipsis notation, the behavior of a function is undefined if it receives parameters in addition to those declared in the parameter list.

To call a function with a variable number of arguments, simply specify any number of arguments in the function call. An example is the **printf** function from the C run-time library. The function call must include one argument for each type name declared in the parameter list or the list of argument types.

All the arguments specified in the function call are placed on the stack unless the **__fastcall** calling convention is specified. The number of parameters declared for the function determines how many of the arguments are taken from the stack and assigned to the parameters. You are responsible for retrieving any additional arguments from the stack and for determining how many arguments are present. The STDARG.H file contains ANSI-style macros for accessing arguments of functions which take a variable number of arguments. Also, the XENIX® - style macros in VARARGS.H are still supported.

This sample declaration is for a function that calls a variable number of arguments:

```
int average( int first, ...);
```

See Also

Concepts

Function Calls

Recursive Functions

Any function in a C program can be called recursively; that is, it can call itself. The number of recursive calls is limited to the size of the stack. See the /STACK (Stack Allocations) (/STACK) linker option for information about linker options that set stack size. Each time the function is called, new storage is allocated for the parameters and for the **auto** and **register** variables so that their values in previous, unfinished calls are not overwritten. Parameters are only directly accessible to the instance of the function in which they are created. Previous parameters are not directly accessible to ensuing instances of the function.

Note that variables declared with **static** storage do not require new storage with each recursive call. Their storage exists for the lifetime of the program. Each reference to such a variable accesses the same storage area.

Example

This example illustrates recursive calls:

```
int factorial( int num );      /* Function prototype */

int main()
{
    int result, number;
    .
    .
    .
    result = factorial( number );
}

int factorial( int num )      /* Function definition */
{
    .
    .
    .
    if ( ( num > 0 ) || ( num <= 10 ) )
        return( num * factorial( num - 1 ) );
}
```

See Also

Concepts

Function Calls

C Language Syntax Summary

This section gives the full description of the C language and the Microsoft-specific C language features. You can use the syntax notation in this section to determine the exact syntax for any language component. The explanation for the syntax appears in the section of this manual where a topic is discussed.

Note
This syntax summary is not part of the ANSI C standard, but is included for information only. Microsoft-specific syntax is noted in comments following the syntax.

See Also

Reference

C Language Reference

Definitions and Conventions

Terminals are endpoints in a syntax definition. No other resolution is possible. Terminals include the set of reserved words and user-defined identifiers.

Nonterminals are placeholders in the syntax and are defined elsewhere in this syntax summary. Definitions can be recursive.

An optional component is indicated by the subscripted opt. For example,

```
{ expression <SUB>opt</SUB> }
```

indicates an optional expression enclosed in braces.

The syntax conventions use different font attributes for different components of the syntax. The symbols and fonts are as follows:

Attribute	Description
nonterminal	Italic type indicates nonterminals.
const	Terminals in bold type are literal reserved words and symbols that must be entered as shown. Characters in this context are always case sensitive.
opt	Nonterminals followed by opt are always optional.
default typeface	Characters in the set described or listed in this typeface can be used as terminals in C statements.

A colon (**:**) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced with the words "one of."

See Also

Reference

C Language Syntax Summary

Lexical Grammar

- Tokens
- Keywords
- Identifiers
- Constants
- String Literals
- Operators
- Punctuators

See Also

Reference

C Language Syntax Summary

Summary of Tokens

token:

keyword

identifier

constant

string-literal

operator

punctuator

preprocessing-token:

header-name

identifier

pp-number

character-constant

string-literal

operator punctuator

each nonwhite-space character that cannot be one of the above

header-name:

< *path-spec* **>** **"** *path spec* **"**

path-spec:

Legal file path

pp-number:

digit

. *digit*

pp-number digit

pp-number nondigit

pp-number **e** *sign*

pp-number **E** *sign*

pp-number **.**

See Also

Reference

Lexical Grammar

Summary of Keywords

keyword : one of

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

See Also

Reference

Lexical Grammar

Summary of Identifiers

identifier:

nondigitidentifier nondigitidentifier digit

nondigit: one of

_ a b c d e f g h i j k l mn o p q r s t u v w x y zA B C D E F G H I J K L MN O P Q R S T U V W X Y Z

digit: one of

0 1 2 3 4 5 6 7 8 9

See Also
Reference
Lexical Grammar

Summary of Constants

constant:

floating-point-constant

integer-constant

enumeration-constant

character-constant

floating-point-constant:

fractional-constant exponent-part $_{\text{opt}}$ *floating-suffix* $_{\text{opt}}$

digit-sequence exponent-part floating-suffix $_{\text{opt}}$

fractional-constant:

digit-sequence $_{\text{opt}}$ **.***digit-sequence*

digit-sequence **.**

exponent-part:

e *sign* $_{\text{opt}}$ *digit-sequence*

E *sign* $_{\text{opt}}$ *digit-sequence*

sign: one of

+ −

digit-sequence:

digit

digit-sequence digit

floating-suffix: one of

f l F L

integer-constant:

decimal-constant integer-suffix $_{\text{opt}}$

octal-constant integer-suffix $_{\text{opt}}$

hexadecimal-constant integer-suffix $_{\text{opt}}$

decimal-constant:

nonzero-digit

decimal-constant digit

octal-constant:

0

octal-constant octal-digit

hexadecimal-constant:

0x *hexadecimal-digit*

0X *hexadecimal-digit*

hexadecimal-constant hexadecimal-digit

nonzero-digit: one of

1 2 3 4 5 6 7 8 9

octal-digit: one of

0 1 2 3 4 5 6 7

hexadecimal-digit: one of

0 1 2 3 4 5 6 7 8 9

a b c d e f

A B C D E F

unsigned-suffix: one of

u U

long-suffix: one of

l L

character-constant:

' *c-char-sequence*

'L' *c-char-sequence* **'**

integer-suffix:

unsigned-suffix long-suffix $_{\text{opt}}$

long-suffix unsigned-suffix $_{\text{opt}}$

c-char-sequence:

c-char

c-char-sequence c-char

c-char:

Any member of the source character set except the single quotation mark ('), backslash (****), or newline character *escape-sequence*

escape-sequence:

simple-escape-sequence

octal-escape-sequence

hexadecimal-escape-sequence

simple-escape-sequence: one of

\a \b \f \n \r \t \v

\' \" \\ \?

octal-escape-sequence:

**** *octal-digit*

**** *octal-digit octal-digit*

**** *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence:

\x *hexadecimal-digit*

hexadecimal-escape-sequence hexadecimal-digit

See Also

Reference

Lexical Grammar

Summary of String Literals

string-literal:

' *s-char-sequence* $_{\text{opt}}$ **'**

L' *s-char-sequence* $_{\text{opt}}$ **'**

s-char-sequence:

s-char

s-char-sequence s-char

s-char:

any member of the source character set except the double-quotation mark ("), backslash (\), or newline character *escape-sequence*

See Also

Reference

Lexical Grammar

Operators (C)

operator: one of

```
[] () . -
>++ — & * + - ~ ! sizeof/ % << >> <> <= >= == != ^ | && !!? := *= /= %= += -
= <<= >>= &= ^= |=, # ##
```

assignment-operator: one of

```
= *= /= %= += -= <<= >>= &= ^= |=
```

See Also

Reference

Lexical Grammar

Punctuators

punctuator: one of

```
[] () {} * , : = ; ... #
```

See Also

Reference

Lexical Grammar

Phrase Structure Grammar

- Expressions
- Declarations
- Statements
- External Definitions

See Also

Reference

C Language Syntax Summary

Summary of Expressions

primary-expression:

identifier

constant

string-literal

(*expression* **)**

expression:

assignment-expression

expression **,** *assignment-expression*

constant-expression:

conditional-expression

conditional-expression:

logical-OR-expression

logical-OR-expression **?** *expression* **:** *conditional-expression*

assignment-expression:

conditional-expression

unary-expression assignment-operator assignment-expression

postfix-expression:

primary-expression

postfix-expression **[** *expression* **]**

postfix-expression **(** *argument-expression-list* $_{\text{opt}}$ **)**

postfix-expression **.** *identifier*

postfix-expression **–>** *identifier*

postfix-expression **++**

postfix-expression **—**

argument-expression-list:

assignment-expression

argument-expression-list **,** *assignment-expression*

unary-expression:

postfix-expression

++ *unary-expression*

— *unary-expression*

unary-operator

cast-expression

sizeof *unary-expression*

sizeof (*type-name* **)**

unary-operator: one of

& * + – ~ !

cast-expression:

unary-expression

(*type-name* **)** *cast-expression*

multiplicative-expression:

cast-expression

multiplicative-expression ***** *cast-expression*

multiplicative-expression **/** *cast-expression*

multiplicative-expression **%** *cast-expression*

additive-expression:

multiplicative-expression

additive-expression **+** *multiplicative-expression*

additive-expression **–** *multiplicative-expression*

shift-expression:

additive-expression

shift-expression **<<** *additive-expression*

shift-expression **>>** *additive-expression*

relational-expression:

shift-expression

relational-expression **<** *shift-expression*

relational-expression **>** *shift-expression relational-expression* **<=** *shift-expression*

relational-expression **>=** *shift-expression*

equality-expression:

relational-expression

equality-expression **==** *relational-expression*

equality-expression **!=** *relational-expression*

AND-expression:

equality-expression

AND-expression **&** *equality-expression*

exclusive-OR-expression:

AND-expression

exclusive-OR-expression **^** *AND-expression*

inclusive-OR-expression:

exclusive-OR-expression

inclusive-OR-expression **|** *exclusive-OR-expression*

logical-AND-expression:

inclusive-OR-expression

logical-AND-expression **&&** *inclusive-OR-expression*

logical-OR-expression:

logical-AND-expression

logical-OR-expression **||** *logical-AND-expression*

See Also

Reference

Phrase Structure Grammar

Summary of Declarations

declaration:

declaration-specifiers attribute-seq $_{opt}$ *init-declarator-list* $_{opt}$**;**

/* *attribute-seq* is Microsoft Specific */

declaration-specifiers:

storage-class-specifier declaration-specifiers $_{opt}$

type-specifier declaration-specifiers $_{opt}$

type-qualifier declaration-specifiers $_{opt}$

attribute-seq : /* *attribute-seq* is Microsoft Specific */

attribute attribute-seq $_{opt}$

attribute : one of /* Microsoft Specific */

__asm	**__fastcall**
__based	**__inline**
__cdecl	**__stdcall**

init-declarator-list:

init-declarator

init-declarator-list **,** *init-declarator*

init-declarator:

declarator

declarator **=** *initializer* /* For scalar initialization */

storage-class-specifier:

auto

register

static

extern

typedef

__declspec (*extended-decl-modifier-seq* **)** /* Microsoft Specific */

type-specifier:

void

char

short

int

__int8 /* Microsoft Specific */

__int16 /* Microsoft Specific */

__int32 /* Microsoft Specific */

__int64 /* Microsoft Specific */

long

float

double

signed

unsigned

struct-or-union-specifier

enum-specifier

typedef-name

type-qualifier:

const

volatile

declarator:

pointer $_{\text{opt}}$ *direct-declarator*

direct-declarator:

identifier

(*declarator* **)**

direct-declarator **[** *constant-expression* $_{\text{opt}}$ **]**

direct-declarator **(** *parameter-type-list* **)** /* New-style declarator */

direct-declarator **(** *identifier-list* $_{\text{opt}}$ **)** /* Obsolete-style declarator */

pointer:

***** *type-qualifier-list* $_{\text{opt}}$

***** *type-qualifier-list* $_{\text{opt}}$ *pointer*

parameter-type-list: /* The parameter list */

parameter-list

parameter-list **, ...**

parameter-list:

parameter-declaration

parameter-list **,** *parameter-declaration*

type-qualifier-list:

type-qualifier

type-qualifier-list type-qualifier

enum-specifier:

enum *identifier* $_{\text{opt}}$ **{** *enumerator-list* **}**

enum *identifier*

enumerator-list:

enumerator

enumerator-list **,** *enumerator*

enumerator:

enumeration-constant

enumeration-constant **=** *constant-expression*

enumeration-constant:

identifier

struct-or-union-specifier:

struct-or-union identifier $_{\text{opt}}$ **{** *struct-declaration-list* **}** *struct-or-union identifier*

struct-or-union:

struct

union

struct-declaration-list:

struct-declaration

struct-declaration-list struct-declaration

struct-declaration:

specifier-qualifier-list struct-declarator-list **;**

specifier-qualifier-list:

type-specifier specifier-qualifier-list $_{\text{opt}}$

type-qualifier specifier-qualifier-list $_{\text{opt}}$

struct-declarator-list:

struct-declarator struct-declarator-list **,** *struct-declarator*

struct-declarator:

declarator

type-specifier declarator $_{\text{opt}}$ **:** *constant-expression*

parameter-declaration:

declaration-specifiers declarator /* Named declarator */

declaration-specifiers abstract-declarator $_{\text{opt}}$ **/*** Anonymous declarator ***/**

identifier-list: **/*** For old-style declarator ***/**

identifier

identifier-list **,** *identifier*

abstract-declarator: **/*** Used with anonymous declarators ***/**

pointer

pointer $_{\text{opt}}$ *direct-abstract-declarator*

direct-abstract-declarator:

(*abstract-declarator* **)**

direct-abstract-declarator $_{\text{opt}}$ **[** *constant-expression* $_{\text{opt}}$ **]**

direct-abstract-declarator $_{\text{opt}}$ **(** *parameter-type-list* $_{\text{opt}}$ **)**

initializer:

assignment-expression

{ *initializer-list* **}** /* For aggregate initialization */

{ *initializer-list* **,** **}**

initializer-list:

initializer

initializer-list **,** *initializer*

type-name:

specifier-qualifier-list abstract-declarator $_{opt}$

typedef-name:

identifier

extended-decl-modifier-seq:/* Microsoft Specific */

extended-decl-modifier $_{opt}$

extended-decl-modifier-seq extended-decl-modifier

extended-decl-modifier: /* Microsoft Specific */

thread

naked

dllimport

dllexport

See Also

Reference

Phrase Structure Grammar

Summary of Statements

statement:

labeled-statement

compound-statement

expression-statement

selection-statement

iteration-statement

jump-statement

try-except-statement /* Microsoft Specific */

try-finally-statement /* Microsoft Specific */

jump-statement:

goto *identifier* **;**

continue ;

break ;

return *expression* $_{opt}$ **;**

compound-statement:

{ *declaration-list* $_{opt}$ *statement-list* $_{opt}$ **}**

declaration-list:

declaration

declaration-list declaration

statement-list:

statement

statement-list statement

expression-statement:

expression $_{opt}$ **;**

iteration-statement:

while (*expression* **)** *statement*

do *statement* **while (** *expression* **) ;**

for (*expression* $_{opt}$ **;** *expression* $_{opt}$ **;** *expression* $_{opt}$ **)** *statement*

selection-statement:

if (*expression* **)** *statement*

if (*expression* **)** *statement* **else** *statement*

switch (*expression* **)** *statement*

labeled-statement:

identifier **:** *statement*

case *constant-expression* **:** *statement*

default : *statement*

try-except-statement: /* Microsoft Specific */

__try *compound-statement*

__except (*expression* **)** *compound-statement*

try-finally-statement: /* Microsoft Specific */

__try *compound-statement*

__finally *compound-statement*

See Also

Reference

Phrase Structure Grammar

External Definitions

translation-unit:

external-declaration

translation-unit external-declaration

external-declaration: /* Allowed only at external (file) scope */

function-definition

declaration

function-definition: /* Declarator here is the function declarator */

declaration-specifiers $_{\text{opt}}$ *declarator declaration-list* $_{\text{opt}}$ *compound-statement*

See Also

Reference

Phrase Structure Grammar

Implementation-Defined Behavior

ANSI X3.159-1989, *American National Standard for Information Systems – Programming Language – C*, contains a section called "Portability Issues." The ANSI section lists areas of the C language that ANSI leaves open to each particular implementation. This section describes how Microsoft C handles these implementation-defined areas of the C language.

This section follows the same order as the ANSI section. Each item covered includes references to the ANSI that explains the implementation-defined behavior.

Note
This section describes the U.S. English-language version of the C compiler only. Implementations of Microsoft C for other languages may differ slightly.

See Also

Reference

C Language Reference

Translation: Diagnostics

ANSI 2.1.1.3 How a diagnostic is identified

Microsoft C produces error messages in the form:

```
filename( line-number ) : diagnostic Cnumber message
```

where *filename* is the name of the source file in which the error was encountered; *line-number* is the line number at which the compiler detected the error; *diagnostic* is either "error" or "warning"; *number* is a unique four-digit number (preceded by a **C**, as noted in the syntax) that identifies the error or warning; *message* is an explanatory message.

See Also

Concepts

Implementation-Defined Behavior

Environment

- Arguments to main
- Interactive Devices

See Also

Concepts

Implementation-Defined Behavior

Arguments to main

ANSI 2.1.2.2.1 The semantics of the arguments to main

In Microsoft C, the function called at program startup is called **main**. There is no prototype declared for **main**, and it can be defined with zero, two, or three parameters:

```
int main( void )
int main( int argc, char *argv[] )
int main( int argc, char *argv[], char *envp[] )
```

The third line above, where **main** accepts three parameters, is a Microsoft extension to the ANSI C standard. The third parameter, **envp**, is an array of pointers to environment variables. The **envp** array is terminated by a null pointer. See The main Function and Program Execution for more information about **main** and **envp**.

The variable **argc** never holds a negative value.

The array of strings ends with **argv[argc]**, which contains a null pointer.

All elements of the **argv** array are pointers to strings.

A program invoked with no command-line arguments will receive a value of one for **argc**, as the name of the executable file is placed in **argv[0]**. (In MS-DOS versions prior to 3.0, the executable-file name is not available. The letter "C" is placed in **argv[0]**.) Strings pointed to by **argv[1]** through **argv[argc – 1]** represent program parameters.

The parameters **argc** and **argv** are modifiable and retain their last-stored values between program startup and program termination.

See Also

Concepts

Environment

Interactive Devices

ANSI 2.1.2.3 What constitutes an interactive device

Microsoft C defines the keyboard and the display as interactive devices.

See Also

Concepts

Environment

Behavior of Identifiers

- Significant Characters Without External Linkage
- Significant Characters with External Linkage
- Uppercase and Lowercase

See Also

Reference

Using extern to Specify Linkage

Significant Characters Without External Linkage

ANSI 3.1.2 The number of significant characters without external linkage

Identifiers are significant to 247 characters. The compiler does not restrict the number of characters you can use in an identifier; it simply ignores any characters beyond the limit.

See Also

Reference

Using extern to Specify Linkage

Significant Characters with External Linkage

ANSI 3.1.2 The number of significant characters with external linkage

Identifiers declared **extern** in programs compiled with Microsoft C are significant to 247 characters. You can modify this default to a smaller number using the /H (restrict length of external names) option.

See Also

Reference

Using extern to Specify Linkage

Uppercase and Lowercase

ANSI 3.1.2 Whether case distinctions are significant

Microsoft C treats identifiers within a compilation unit as case sensitive.

The Microsoft linker is case sensitive. You must specify all identifiers consistently according to case.

See Also

Concepts

Behavior of Identifiers

Characters

- The ASCII Character Set
- Multibyte Characters
- Bits per Character
- Character Sets
- Unrepresented Character Constants
- Wide Characters
- Converting Multibyte Characters
- Range of char Values

See Also

Concepts

Implementation-Defined Behavior

The ASCII Character Set

ANSI 2.2.1 Members of source and execution character sets

The source character set is the set of legal characters that can appear in source files. For Microsoft C, the source character set is the standard ASCII character set.

Warning Because keyboard and console drivers can remap the character set, programs intended for international distribution should check the Country/Region code.

See Also

Concepts

Characters

Multibyte Characters

ANSI 2.2.1.2 Shift states for multibyte characters

Multibyte characters are used by some implementations, including Microsoft C, to represent foreign-language characters not represented in the base character set. However, Microsoft C does not support any state-dependent encodings. Therefore, there are no shift states. See Multibyte and Wide Characters for more information.

See Also

Concepts

Characters

Bits per Character

ANSI 2.2.4.2.1 Number of bits in a character

The number of bits in a character is represented by the manifest constant **CHAR_BIT**. The LIMITS.H file defines **CHAR_BIT** as 8.

See Also

Concepts

Characters

Character Sets

ANSI 3.1.3.4 Mapping members of the source character set

The source character set and execution character set include the ASCII characters listed in the following table. Escape sequences are also shown in the table.

Escape Sequences

Escape Sequence	Character	ASCII Value
\a	Alert/bell	7
\b	Backspace	8
\f	Formfeed	12
\n	Newline	10
\r	Carriage return	13
\t	Horizontal tab	9
\v	Vertical tab	11
\"	Double quotation	34
\'	Single quotation	39
\\	Backslash	92

See Also

Concepts

Characters

Unrepresented Character Constants

ANSI 3.1.3.4 The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant

All character constants or escape sequences can be represented in the extended character set.

See Also

Concepts

Characters

Wide Characters

ANSI 3.1.3.4 The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character

The regular character constant, 'ab' has the integer value (int)0x6162. When there is more than one byte, previously read bytes are shifted left by the value of **CHAR_BIT** and the next byte is compared using the bitwise-OR operator with the low **CHAR_BIT** bits. The number of bytes in the multibyte character constant cannot exceed sizeof(int), which is 4 for 32-bit target code.

The multibyte character constant is read as above and this is converted to a wide-character constant using the **mbtowc** run-time function. If the result is not a valid wide-character constant, an error is issued. In any event, the number of bytes examined by the **mbtowc** function is limited to the value of **MB_CUR_MAX**.

See Also

Concepts

Characters

Converting Multibyte Characters

ANSI 3.1.3.4 The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant

The current locale is the "C" locale by default. It can be changed with the #pragma setlocale.

See Also

Concepts

Characters

Range of char Values

ANSI 3.2.1.1 Whether a "plain" **char** has the same range of values as a **signed char** or an **unsigned char**

All signed character values range from –128 to 127. All unsigned character values range from 0 to 255.

The /J compiler option changes the default from **signed** to **unsigned**.

See Also

Concepts

Characters

Integers

- Range of Integer Values
- Demotion of Integers
- Signed Bitwise Operations
- Remainders
- Right Shifts

See Also

Concepts

Implementation-Defined Behavior

Range of Integer Values

ANSI 3.1.2.5 The representations and sets of values of the various types of integers

Integers contain 32 bits (four bytes). Signed integers are represented in two's-complement form. The most-significant bit holds the sign: 1 for negative, 0 for positive and zero. The values are listed below:

Type	Minimum and Maximum
unsigned short	0 to 65535
signed short	–32768 to 32767
unsigned long	0 to 4294967295
signed long	–2147483648 to 2147483647

See Also

Concepts

Integers

Demotion of Integers

ANSI 3.2.1.2 The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented

When a **long** integer is cast to **a short**, or a **short** is cast to a **char**, the least-significant bytes are retained.

For example, this line

```
short x = (short)0x12345678L;
```

assigns the value 0x5678 to `x`, and this line

```
char y = (char)0x1234;
```

assigns the value 0x34 to `y`.

When signed variables are converted to unsigned and vice versa, the bit patterns remain the same. For example, casting –2 (0xFE) to an unsigned value yields 254 (also 0xFE).

See Also

Concepts

Integers

Signed Bitwise Operations

ANSI 3.3 The results of bitwise operations on signed integers

Bitwise operations on signed integers work the same as bitwise operations on unsigned integers. For example, `-16 & 99` can be expressed in binary as

```
  11111111 11110000
& 00000000 01100011
  ___________________
  00000000 01100000
```

The result of the bitwise AND is 96.

See Also

Concepts

Integers

Remainders

ANSI 3.3.5 The sign of the remainder on integer division

The sign of the remainder is the same as the sign of the dividend. For example,

```
 50 / -6 == -8
 50 % -6 ==  2
-50 /  6 == -8
-50 %  6 == -2
```

See Also

Concepts

Integers

Right Shifts

ANSI 3.3.7 The result of a right shift of a negative-value signed integral type

Shifting a negative value to the right yields half the absolute value, rounded down. For example, –253 (binary 11111111 00000011) shifted right one bit produces –127 (binary 11111111 10000001). A *positive* 253 shifts right to produce +126.

Right shifts preserve the sign bit. When a signed integer shifts right, the most-significant bit remains set. When an unsigned integer shifts right, the most-significant bit is cleared.

If 0xF000 is unsigned, the result is 0x7800.

If 0xF0000000 is signed, a right shift produces 0xF8000000. Shifting a positive number right 32 times produces 0xF0000000. Shifting a negative number right 32 times produces 0xFFFFFFFF.

See Also

Concepts

Integers

Floating-Point Math

- Values
- Casting Integers to Floating-Point Values
- Truncation of Floating-Point Values

See Also

Concepts

Implementation-Defined Behavior

Values

ANSI 3.1.2.5 The representations and sets of values of the various types of floating-point numbers

The **float** type contains 32 bits: 1 for the sign, 8 for the exponent, and 23 for the mantissa. Its range is +/– 3.4E38 with at least 7 digits of precision.

The **double** type contains 64 bits: 1 for the sign, 11 for the exponent, and 52 for the mantissa. Its range is +/– 1.7E308 with at least 15 digits of precision.

The **long double** type contains 80 bits: 1 for the sign, 15 for the exponent, and 64 for the mantissa. Its range is +/– 1.2E4932 with at least 19 digits of precision. With the Microsoft C compiler, the representation of type **long double** is identical to type **double**.

See Also

Concepts

Floating-Point Math

Casting Integers to Floating-Point Values

ANSI 3.2.1.3 The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value

When an integral number is cast to a floating-point value that cannot exactly represent the value, the value is rounded (up or down) to the nearest suitable value.

For example, casting an **unsigned long** (with 32 bits of precision) to a **float** (whose mantissa has 23 bits of precision) rounds the number to the nearest multiple of 256. The **long** values 4,294,966,913 to 4,294,967,167 are all rounded to the **float** value 4,294,967,040.

See Also

Concepts

Floating-Point Math

Truncation of Floating-Point Values

ANSI 3.2.1.4 The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number

When an underflow occurs, the value of a floating-point variable is rounded down to zero. An overflow may cause a run-time error or it may produce an unpredictable value, depending on the optimizations specified.

See Also

Concepts

Floating-Point Math

Arrays and Pointers

- Largest Array Size
- Pointer Subtraction

See Also

Concepts

Implementation-Defined Behavior

Largest Array Size

ANSI 3.3.3.4, 4.1.1 The type of integer required to hold the maximum size of an array — that is, the size of **size_t**

The **size_t** typedef is an **unsigned int**. On Itanium Processor Family (IPF), **size_t** typedef is an **unsigned __int64**.

See Also

Concepts

Arrays and Pointers

Pointer Subtraction

ANSI 3.3.6, 4.1.1 The type of integer required to hold the difference between two pointers to elements of the same array, **ptrdiff_t**

A **ptrdiff_t** is a **signed int**.

See Also

Concepts

Arrays and Pointers

Registers: Availability of Registers

ANSI 3.5.1 The extent to which objects can actually be placed in registers by use of the register storage-class specifier

The compiler does not honor user requests for register variables. Instead, it makes it own choices when optimizing.

See Also

Concepts

Implementation-Defined Behavior

Structures, Unions, Enumerations, and Bit Fields

- Improper Access to a Union
- Padding and Alignment of Structure Members
- Sign of Bit Fields
- Storage of Bit Fields
- The enum Type

See Also

Concepts

Implementation-Defined Behavior

Improper Access to a Union

ANSI 3.3.2.3 A member of a union object is accessed using a member of a different type

If a union of two types is declared and one value is stored, but the union is accessed with the other type, the results are unreliable.

For example, a union of **float** and **int** is declared. A **float** value is stored, but the program later accesses the value as an **int**. In such a situation, the value would depend on the internal storage of **float** values. The integer value would not be reliable.

See Also

Concepts

Structures, Unions, Enumerations, and Bit Fields

Padding and Alignment of Structure Members

ANSI 3.5.2.1 The padding and alignment of members of structures and whether a bit field can straddle a storage-unit boundary

Structure members are stored sequentially in the order in which they are declared: the first member has the lowest memory address and the last member the highest.

Every data object has an alignment-requirement. The alignment-requirement for all data except structures, unions, and arrays is either the size of the object or the current packing size (specified with either /Zp or the **pack** pragma, whichever is less). For structures, unions, and arrays, the alignment-requirement is the largest alignment-requirement of its members. Every object is allocated an offset so that

offset **%** *alignment-requirement* **==** 0

Adjacent bit fields are packed into the same 1-, 2-, or 4-byte allocation unit if the integral types are the same size and if the next bit field fits into the current allocation unit without crossing the boundary imposed by the common alignment requirements of the bit fields.

See Also

Concepts

Structures, Unions, Enumerations, and Bit Fields

Sign of Bit Fields

ANSI 3.5.2.1 Whether a "plain" **int** field is treated as a **signed int** bit field or as an unsigned int bit field

Bit fields can be signed or unsigned. Plain bit fields are treated as signed.

See Also

Concepts

Structures, Unions, Enumerations, and Bit Fields

Storage of Bit Fields

ANSI 3.5.2.1 The order of allocation of bit fields within an int

Bit fields are allocated within an integer from least-significant to most-significant bit. In the following code

```
struct mybitfields
{
   unsigned a : 4;
   unsigned b : 5;
   unsigned c : 7;
} test;

int main( void )
{
   test.a = 2;
   test.b = 31;
   test.c = 0;
}
```

the bits would be arranged as follows:

```
00000001 11110010
ccccccccb bbbbaaaa
```

Since the 80x86 processors store the low byte of integer values before the high byte, the integer 0x01F2 above would be stored in physical memory as 0xF2 followed by 0x01.

See Also

Concepts

Structures, Unions, Enumerations, and Bit Fields

The enum type

ANSI 3.5.2.2 The integer type chosen to represent the values of an enumeration type

A variable declared as **enum** is an **int**.

See Also
Concepts
Structures, Unions, Enumerations, and Bit Fields

Qualifiers: Access to Volatile Objects

ANSI 3.5.5.3 What constitutes an access to an object that has volatile-qualified type

Any reference to a volatile-qualified type is an access.

See Also

Concepts

Implementation-Defined Behavior

Declarators: Maximum number

ANSI 3.5.4 The maximum number of declarators that can modify an arithmetic, structure, or union type

Microsoft C does not limit the number of declarators. The number is limited only by available memory.

See Also

Concepts

Implementation-Defined Behavior

Statements: Limits on Switch Statements

ANSI 3.6.4.2 The maximum number of **case** values in a **switch** statement

Microsoft C does not limit the number of **case** values in a **switch** statement. The number is limited only by available memory.

See Also

Concepts

Implementation-Defined Behavior

Preprocessing Directives

- Character Constants and Conditional Inclusion
- Including Bracketed Filenames
- Including Quoted Filenames
- Character Sequences
- Pragmas
- Default Date and Time

See Also

Concepts

Implementation-Defined Behavior

Character Constants and Conditional Inclusion

ANSI 3.8.1 Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Whether such a character constant can have a negative value

The character set used in preprocessor statements is the same as the execution character set. The preprocessor recognizes negative character values.

See Also

Concepts

Preprocessing Directives

Including Bracketed Filenames

ANSI 3.8.2 The method for locating includable source files

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A "parent" file is the file that has the #include directive in it. Instead, it begins by searching for the file in the directories specified on the compiler command line following /I. If the /I option is not present or fails, the preprocessor uses the INCLUDE environment variable to find any include files within angle brackets. The INCLUDE environment variable can contain multiple paths separated by semicolons (**;**). If more than one directory appears as part of the /I option or within the INCLUDE environment variable, the preprocessor searches them in the order in which they appear.

See Also

Concepts

Preprocessing Directives

Including Quoted Filenames

ANSI 3.8.2 The support for quoted names for includable source files

If you specify a complete, unambiguous path specification for the include file between two sets of double quotation marks (" "), the preprocessor searches only that path specification and ignores the standard directories.

For include files specified as #include "path-spec", directory searching begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file has not been found, the search continues as if the filename were enclosed in angle brackets.

See Also

Concepts

Preprocessing Directives

Character Sequences

ANSI 3.8.2 The mapping of source file character sequences

Preprocessor statements use the same character set as source file statements with the exception that escape sequences are not supported.

Thus, to specify a path for an include file, use only one backslash:

```
#include "path1\path2\myfile"
```

Within source code, two backslashes are necessary:

```
fil = fopen( "path1\\path2\\myfile", "rt" );
```

See Also

Concepts

Preprocessing Directives

Pragmas

ANSI 3.8.6 The behavior on each recognized #pragma directive

The following C Pragmas are defined for the Microsoft C compiler:

alloc_text	**data_seg**	**include_alias**	**setlocale**
auto_inline	**function**	**intrinsic**	**warning**
check_stack	**hdrstop**	**message**	
code_seg	**inline_depth**	**optimize**	
comment	**inline_recursion**	**pack**	

See Also

Concepts

Preprocessing Directives

Default Date and Time

ANSI 3.8.8 The definitions for _DATE_ and _TIME_ when, respectively, the date and time of translation are not available

When the operating system does not provide the date and time of translation, the default values for _DATE_ and _TIME_ are `May 03 1957` and `17:00:00"`.

See Also

Concepts

Preprocessing Directives

Library Functions

- NULL Macro
- Diagnostic Printed by the assert Function
- Character Testing
- Domain Errors
- Underflow of Floating-Point Values
- The fmod Function
- The signal Function
- Default Signals
- Terminating Newline Characters
- Blank Lines
- Null Characters
- File Position in Append Mode
- Truncation of Text Files
- File Buffering
- Zero-Length Files
- Filenames
- File Access Limits
- Deleting Open Files
- Renaming with a Name That Exists
- Reading Pointer Values
- Reading Ranges
- File Position Errors
- Messages Generated by the perror Function
- Allocating Zero Memory
- The abort Function
- The atexit Function
- Environment Names
- The system Function
- The strerror Function
- The Time Zone
- The clock Function

See Also

Concepts

Implementation-Defined Behavior

NULL Macro

ANSI 4.1.5 The null pointer constant to which the macro NULL expands

Several include files define the NULL macro as `((void *)0)`.

See Also

Concepts

Library Functions

Diagnostic Printed by the assert Function

ANSI 4.2 The diagnostic printed by and the termination behavior of the **assert** function

The **assert** function prints a diagnostic message and calls the **abort** routine if the expression is false (0). The diagnostic message has the form

Assertion failed: *expression*, **file** *filename*, **line** *linenumber*

where filename is the name of the source file and linenumber is the line number of the assertion that failed in the source file. No action is taken if expression is true (nonzero).

See Also

Concepts

Library Functions

Character Testing

ANSI 4.3.1 The sets of characters tested for by the **isalnum**, **isalpha**, **iscntrl**, **islower**, **isprint**, and **isupper** functions

The following list describes these functions as they are implemented by the Microsoft C compiler.

Function	Tests For
isalnum	Characters 0 – 9, A–Z, a–z ASCII 48–57, 65–90, 97–122
isalpha	Characters A–Z, a–z ASCII 65–90, 97–122
iscntrl	ASCII 0 –31, 127
islower	Characters a–z ASCII 97–122
isprint	Characters A–Z, a–z, 0 – 9, punctuation, space ASCII 32–126
isupper	Characters A–Z ASCII 65–90

See Also

Concepts

Library Functions

Domain Errors

ANSI 4.5.1 The values returned by the mathematics functions on domain errors

The ERRNO.H file defines the domain error constant **EDOM** as 33. See the help topic for the particular function that caused the error, for information about the return value.

See Also

Concepts

Library Functions

Underflow of Floating-Point Values

ANSI 4.5.1 Whether the mathematics functions set the integer expression **errno** to the value of the macro **ERANGE** on underflow range errors

A floating-point underflow does not set the expression **errno** to **ERANGE**. When a value approaches zero and eventually underflows, the value is set to zero.

See Also

Concepts

Library Functions

The fmod Function

ANSI 4.5.6.4 Whether a domain error occurs or zero is returned when the **fmod** function has a second argument of zero

When the **fmod** function has a second argument of zero, the function returns zero.

See Also
Concepts
Library Functions

The signal Function

ANSI 4.7.1.1 The set of signals for the **signal** function

The first argument passed to **signal** must be one of the symbolic constants described in the *Run-Time Library Reference* for the **signal** function. The information in the *Run-Time Library Reference* also lists the operating mode support for each signal. The constants are also defined in SIGNAL.H.

See Also
Concepts
Library Functions

Default Signals

ANSI 4.7.1.1 If the equivalent of **signal (***sig***, SIG_DFL)** is not executed prior to the call of a signal handler, the blocking of the signal that is performed

Signals are set to their default status when a program begins running.

See Also

Concepts

Library Functions

Terminating Newline Characters

ANSI 4.9.2 Whether the last line of a text stream requires a terminating newline character

Stream functions recognize either new line or end of file as the terminating character for a line.

See Also

Concepts

Library Functions

Blank Lines

ANSI 4.9.2 Whether space characters that are written out to a text stream immediately before a newline character appear when read in

Space characters are preserved.

See Also

Concepts

Library Functions

Null Characters

ANSI 4.9.2 The number of null characters that can be appended to data written to a binary stream

Any number of null characters can be appended to a binary stream.

See Also

Concepts

Library Functions

File Position in Append Mode

ANSI 4.9.3 Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file

When a file is opened in append mode, the file-position indicator initially points to the end of the file.

See Also

Concepts

Library Functions

Truncation of Text Files

ANSI 4.9.3 Whether a write on a text stream causes the associated file to be truncated beyond that point

Writing to a text stream does not truncate the file beyond that point.

See Also

Concepts

Library Functions

File Buffering

ANSI 4.9.3 The characteristics of file buffering

Disk files accessed through standard I/O functions are fully buffered. By default, the buffer holds 512 bytes.

See Also

Concepts

Library Functions

Zero-Length Files

ANSI 4.9.3 Whether a zero-length file actually exists

Files with a length of zero are permitted.

See Also

Concepts

Library Functions

Filenames

ANSI 4.9.3 The rules for composing valid file names

A file specification can include an optional drive letter (always followed by a colon), a series of optional directory names (separated by backslashes), and a filename.

For more information, see Naming a File for more information.

See Also

Concepts

Library Functions

File Access Limits

ANSI 4.9.3 Whether the same file can be open multiple times

Opening a file that is already open is not permitted.

See Also

Concepts

Library Functions

Deleting Open Files

ANSI 4.9.4.1 The effect of the remove function on an open file

The remove function deletes a file. If the file is open, this function fails and returns -1.

See Also

Concepts

Library Functions

Renaming with a Name That Exists

ANSI 4.9.4.2 The effect if a file with the new name exists prior to a call to the **rename** function

If you attempt to rename a file using a name that exists, the **rename** function fails and returns an error code.

See Also

Concepts

Library Functions

Reading Pointer Values

ANSI 4.9.6.2 The input for **%p** conversion in the **fscanf** function

When the **%p** format character is specified, the **fscanf** function converts pointers from hexadecimal ASCII values into the correct address.

See Also

Concepts

Library Functions

Reading Ranges

ANSI 4.9.6.2 The interpretation of a dash (–) character that is neither the first nor the last character in the scanlist for % [conversion in the **fscanf** function

The following line

```
fscanf( fileptr, "%[A-Z]", strptr);
```

reads any number of characters in the range A–Z into the string to which `strptr` points.

See Also

Concepts

Library Functions

File Position Errors

ANSI 4.9.9.1, 4.9.9.4 The value to which the macro **errno** is set by the **fgetpos** or **ftell** function on failure

When **fgetpos** or **ftell** fails, **errno** is set to the manifest constant **EINVAL** if the position is invalid or EBADF if the file number is bad. The constants are defined in ERRNO.H.

See Also

Concepts

Library Functions

Messages Generated by the perror Function

ANSI 4.9.10.4 The messages generated by the **perror** function

The **perror** function generates these messages:

```
0  Error 0
1
2  No such file or directory
3
4
5
6
7  Arg list too long
8  Exec format error
9  Bad file number
10
11
12 Not enough core
13 Permission denied
14
15
16
17 File exists
18 Cross-device link
19
20
21
22 Invalid argument
23
24 Too many open files
25
26
27
28 No space left on device
29
30
31
32
33 Math argument
34 Result too large
35
36 Resource deadlock would occur
```

See Also

Concepts

Library Functions

Allocating Zero Memory

ANSI 4.10.3 The behavior of the **calloc**, **malloc**, or **realloc** function if the size requested is zero

The **calloc**, **malloc**, and **realloc** functions accept zero as an argument. No actual memory is allocated, but a valid pointer is returned and the memory block can be modified later by realloc.

See Also

Concepts

Library Functions

The abort Function

ANSI 4.10.4.1 The behavior of the **abort** function with regard to open and temporary files

The **abort** function does not close files that are open or temporary. It does not flush stream buffers.

See Also

Concepts

Library Functions

The atexit Function

ANSI 4.10.4.3 The status returned by the **atexit** function if the value of the argument is other than zero, EXIT_SUCCESS, or EXIT_FAILURE

The **atexit** function returns zero if successful, or a nonzero value if unsuccessful.

See Also

Concepts

Library Functions

Environment Names

ANSI 4.10.4.4 The set of environment names and the method for altering the environment list used by the **getenv** function

The set of environment names is unlimited.

To change environment variables from within a C program, call the **_putenv** function. To change environment variables from the command line in Windows or Windows NT, use the SET command (for example, SET LIB = D:\ LIBS).

Environment variables set from within a C program exist only as long as their host copy of the operating system command shell is running (CMD.EXE in Windows NT and COMMAND.COM in Windows). For example, the line

```
system( SET LIB = D:\LIBS );
```

would run a copy of the Windows NT command shell (CMD.EXE), set the environment variable LIB, and return to the C program, exiting the secondary copy of CMD.EXE. Exiting that copy of CMD.EXE removes the temporary environment variable LIB.

This example also runs on the Windows platform.

Likewise, changes made by the **_putenv** function last only until the program ends.

See Also

Concepts

Library Functions

The system Function

ANSI 4.10.4.5 The contents and mode of execution of the string by the **system** function

The **system** function executes an internal operating system command, or an .EXE, .COM (.CMD in Windows NT) or .BAT file from within a C program rather than from the command line.

The system function finds the command interpreter, which is typically CMD.EXE in the Windows NT operating system or COMMAND.COM in Windows. The system function then passes the argument string to the command interpreter.

For more information, see system, _wsystem.

See Also
Concepts
Library Functions

The strerror Function

ANSI 4.11.6.2 The contents of the error message strings returned by the **strerror** function

The **strerror** function generates these messages:

```
0   Error 0
1
2   No such file or directory
3
4
5
6
7   Arg list too long
8   Exec format error
9   Bad file number
10
11
12  Not enough core
13  Permission denied
14
15
16
17  File exists
18  Cross-device link
19
20
21
22  Invalid argument
23
24  Too many open files
25
26
27
28  No space left on device
29
30
31
32
33  Math argument
34  Result too large
35
36  Resource deadlock would occur
```

See Also

Concepts

Library Functions

The Time Zone

ANSI 4.12.1 The local time zone and Daylight Saving Time

The local time zone is Pacific Standard Time. Microsoft C supports Daylight Saving Time.

See Also
Concepts
Library Functions

The clock Function

ANSI 4.12.2.1 The era for the **clock** function

The **clock** function's era begins (with a value of 0) when the C program starts to execute. It returns times measured in 1/**CLOCKS_PER_SEC** (which equals 1/1000 for Microsoft C).

See Also

Concepts

Library Functions

C++ Language Reference

This reference explains the C++ programming language as implemented in Microsoft Visual C++. The organization is based on *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup and on the ANSI/ISO C++ International Standard (ISO/IEC FDIS 14882). Microsoft-specific implementations of C++ language features are included.

See the following tables to quickly find a keyword or operator:

- C++ Keywords
- C++ Operators

In This Section

New Language Features in Visual C++

Reference section for syntax that lets you target the common language runtime.

Compatibility and Compliance Issues in Visual C++

Information about where Visual C++ does not agree with the C++ standard, and breaking changes in the compiler.

Lexical Conventions

Fundamental lexical elements of a C++ program: tokens, comments, operators, keywords, punctuators, literals. Also, file translation, operator precedence/associativity.

Basic Concepts

Scope, linkage, program startup and termination, storage classes, and types.

Standard Conversions

Type conversions between built-in, or "fundamental," types. Also, arithmetic conversions and conversions among pointer, reference, and pointer-to-member types.

Expressions

Types and semantics of expressions, reference topics on operators, casting and casting operators, run-time type information.

Statements

Expression, null, compound, selection, iteration, jump, and declaration statements.

Declarations

Storage-class specifiers, function definitions, initializations, enumerations, class, struct, and union declarations, and typedef declarations. Also, inline functions, const keyword, namespaces.

Declarators

The portion of a declaration statement that names an object, type, or function. Abstract declarators, type names, initializers, function declarations and definitions, arrays, references.

Classes, Structures, and Unions

Introduction to classes, structures, and unions. Also, member functions, data members, bit fields, this pointer, nested classes.

Derived Classes

Single and multiple inheritance, virtual functions, multiple base classes, abstract classes, scope rules. Also, the __super and __interface keywords.

Member-Access Control

Controlling access to class members: public, private, and protected keywords. Friend functions and classes.

Special Member Functions

Special functions unique to class types: constructors, destructors, conversion functions, assignment operator, operator new and operator delete functions.

Overloading

Overloaded functions, declaration matching, argument matching. Also, overloaded operators, rules for operator overloading.

Exception Handling

C++ exception handling, structured exception handling (SEH), keywords used in writing exception handling statements.

Templates

Template specifications, function templates, class templates, typename keyword, templates vs. macros, templates and smart pointers.

Event Handling

Declaring events and event handlers.

Microsoft-Specific Modifiers

Modifiers specific to Microsoft C++. Memory addressing, calling conventions, naked functions, extended storage-class attributes (__declspec), __w64.

Inline Assembler

Using assembly language and C++ in __asm blocks.

Compiler COM Support

A reference to Microsoft-specific classes and global functions used to support COM types.

Related Sections

New Language Features in Visual C++

Reference material on using Visual C++ to target the common language runtime.

C/C++ Building Reference

Compiler options, linker options, and other build tools.

C/C++ Preprocessor Reference

Reference material on pragmas, preprocessor directives, predefined macros, and the preprocessor.

C Language Reference

A reference to the C programming language, including Microsoft extensions.

Visual C++ Libraries

A list of links to the reference start pages for the various Visual C++ libraries.

New Language Features in Visual C++

Visual C++ 2005 includes new syntax for writing applications to target the common language runtime. This topic presents an overview of the new syntax.

The new syntax is not part of the ISO/ANSI C++ standard, but is a set of extensions to C++ standardized under the Ecma C++/CLI Standard. For more information on the new syntax, see the C++/CLI Specification.

This syntax replaces the previous syntax, known as Managed Extensions for C++. Managed Extensions for C++ syntax is still available, but as a deprecated feature under the **/clr:oldSyntax** compiler option. To port code using Managed Extensions for C++, see Managed Extensions for C++ Syntax Upgrade Checklist.

The new syntax is in effect by default when you use **/clr**.

For information on other new features in Visual C++, see

- What's New in Visual C++ Compiler, Language, and Tools
- What's New in Visual C++
- C++ Support Library
- Breaking Changes in the Visual C++ 2005 Compiler
- .NET Programming in C++

CLR Data Type Keywords

The following table lists new keywords that have been added to Visual C++. Note that some keywords consist of two words separated by white space. These aggregate keywords are considered keywords despite the fact that, used separately, they have different meanings. The word **ref**, for example, used without **class** is not a keyword and can be used as a regular identifier. Likewise, by itself **class** denotes a native class. But, used together, **ref class** defines a common language runtime (CLR) reference type.

Keyword	Context sensitive	Purpose	Reference
`ref class` `ref struct`	No	Defines a CLR reference class	Classes and Structs (Managed)
`value class` `value struct`	No	Defines a CLR value class	Classes and Structs (Managed)
`interface class` `interface struct`	No	Defines a CLR interface	interface class
`enum class` `enum struct`	No	Defines a CLR enumeration	enum class
`property`	Yes	Defines a CLR property	property
`delegate`	Yes	Defines a CLR delegate	delegate
`event`	Yes	Defines a CLR event	event (Visual C++)

Override Specifiers

The following keywords can be used to qualify override behavior for derivation. The **new** keyword is not new to C++ but is

listed here because of the additional context in which it can be used. Some specifiers are also valid for native programming. For more information, see How to: Declare Override Specifiers in Native Compilations.

Keyword	Context Sensitive	Purpose	Reference
`abstract`	Yes	Indicates functions or classes are abstract	abstract (Visual C++)
`new`	No	Indicates that a function is not an override of a base class version	new (new slot in vtable)
`override`	Yes	Indicates that a method must be an override of a base-class version	override
`sealed`	Yes	Prevents classes from being used as base classes	sealed

Keywords for Generics

The following keywords have been added to support generic types. For more information, see Generics (Visual C++).

Keyword	Context sensitive	Purpose
`generic`	No	Defines a generic type
`where`	Yes	Specifies the constraints of a generic typef

Miscellaneous New Keywords

The following keywords have been added to Visual C++.

Keyword	Context sensitive	Purpose	Reference
`finally`	Yes	Indicates default exception handlings behavior	Exception Handling under /clr
`for each`	No	Enumerate elements of a collection.	for each, in
`gcnew`	No	Allocates types on the garbage-collected heap	gcnew
`initonly`	Yes	Indicates a member can only be initialized at declaration or in a static constructor	initonly
`literal`	Yes	Creates a literal variable	literal (Visual C++)
`nullptr`	No	Indicates that a handle or pointer does not point at an object	nullptr

Non-Keyword Language Constructs

The following are language constructs that, while not official keywords, are fundamental to .NET programming. These constructs are implemented as templates. Most are defined in the cli Namespace.

Keyword	Purpose	Reference
`array`	Type for representing CLR arrays	array (Visual C++)
`interior_ptr`	Points to data inside reference types.	interior_ptr
`pin_ptr`	Points to CLR reference types to temporarily suppress the garbage collection system	pin_ptr
`safe_cast`	Determines and executes the optimal casting method for CLR types	safe_cast
`typeid`	Retrieves a System.Type object describing the given type or object	typeid

New Operators

Two new operators have been added to Visual C++ to support garbage-collected programming.

Operator	Purpose	Reference
`^`	Indicates a handle to an object located on the garbage-collected heap	^ (Handle to Object on Managed Heap)
`%`	Indicates a tracking reference	% (Tracking Reference)

More CLR Constructs

In addition to keywords and language constructs, this reference section includes the following CLR programming constructs.

Topic	Description
__identifier	Enables the use of keywords as identifiers
.NET Framework Equivalents to C++ Native Types	Lists the CLR types used in place of C++ integral types
appdomain **__declspec** modifier	**__declspec** modifier that mandates static and global variables exist per app domain
C-Style Casts with /clr	Describes how C-style casts are interpreted
__clrcall calling convention	Indicates the CLR-compliant calling convention
__cplusplus_cli	Predefined Macros
Custom Attributes (C++)	Describes how to define your own CLR attributes.
Exception Handling under /clr	Provides an overview of exception handling
Explicit Overrides	Demonstrates how member functions can override arbitrary members
Friend Assemblies	Discusses how a client assembly can access all types in an assembly component.

Implicit Boxing	Demonstrates the conditions where values types are boxed
Compiler Support for Type Traits	Discusses how to detect characteristics of types at compile time.
managed, unmanaged pragmas	Demonstrates how managed and unmanaged functions can co-exist in the same module
process **__declspec** modifier	**__declspec** modifier that mandates static and global variables exist per process
Reflection in C++	Demonstrates the CLR version of run-time type information
System::String Handling in Visual C++	Discusses compiler conversion of string literals to String.
Type Forwarding	Allows you to move a type in a shipping assembly to another assembly, such that, client code does not have to be recompiled.
User-Defined Attributes	Demonstrates use-defined attributes
The #using Directive	Imports external assemblies
XML Documentation (Visual C++)	Explains XML-based code documentation using /doc (Process Documentation Comments) (C/C++)

See Also

Reference

Breaking Changes in the Visual C++ 2005 Compiler

Other Resources

.NET Programming in C++

Native and .NET Interoperability

__identifier

Enables the use of Visual C++ keywords as identifiers.

```
__identifier(Visual_C++_keyword)
```

Remarks

The **__identifier** keyword enables the use of Visual C++ keywords as identifiers, allowing a **/clr** program to use identifiers that are Visual C++ reserved words from referenced metadata.

For more information, see /clr (Common Language Runtime Compilation).

Note

Use of the **__identifier** keyword for identifiers that are not keywords is permitted, but strongly discouraged as a matter of st yle.

The **__identifier** keyword is valid with **/clr** and **/clr:oldSyntax**.

Example

In the following example, a C# class (called `template`) is created and consumed in the Visual C++ program:

```
// identifier_template.cs
// compile with: /target:library
public class template {
   public void Run() { }
}
```

```
// keyword__identifier.cpp
// compile with: /clr
#using <identifier_template.dll>

int main() {
   __identifier(template) ^pTemplate = gcnew __identifier(template)();
   pTemplate->Run();
}
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

% (Tracking Reference)

A tracking reference is similar to a C++ reference, to specify that a variable is passed to a function by reference and to create an alternative name for an object. However, the object referenced by a tracking reference can be moved during execution by the common language runtime garbage collector. For more information on C++ references, see References (C++).

Remarks

A tracking reference is updated by the garbage-collected heap in much the same way that a handle to a reference is updated, when the object moves on the garbage-collected heap.

Use a tracking reference to bind to an object that could possibly be on the garbage-collected heap. Use a tracking reference to a handle when binding to an object of a CLR type on the garbage-collected heap.

It is not possible to have a native C++ reference to an object on the garbage-collected heap.

A tracking reference can point to a whole object, data member, or storage location.

A tracking reference can only be declared on the stack. A tracking reference cannot be a member of a class.

Tracking references are valid to value types and handles.

You can have references to native types and native pointers.

When a tracking reference is initialized, it performs the equivalent of a shallow copy, a "pointer" to shared memory, which is the same behavior as when you initialize a native reference. Also like a native reference, when you assign to a tracking reference, a deep copy is performed, meaning that the memory location of the value is changed.

A tracking reference cannot be assigned null. A tracking reference is rebindable, meaning that you can initialize a tracking reference multiple times.

For more information about tracking references, see:

- How to: Pass CLR Types by Reference with Tracking References Tracking References and Interior Pointers
- How to: Use Tracking References and Value Types Tracking References and Unary "Take-Address" Operator
- How to: Use Tracking References and Value Types
- How to: Cannot Use Tracking References and Unary "Take-Address" Operator
- How to: Write Template Functions that Take Native, Value, or Reference Parameters

Example

```
// tracking_reference_1.cpp
// compile with: /clr
ref class MyClass {
public:
   int i;
};

value struct MyStruct {
   int k;
};

int main() {
   MyClass ^ x = gcnew MyClass;
   MyClass ^% y = x;   // tracking reference handle to reference object

   int %ti = x->i;   // tracking reference to member of reference type

   int j = 0;
   int %tj = j;   // tracking reference to object on the stack

   int * pi = new int[2];
   int % ti2 = pi[0];   // tracking reference to object on native heap

   int *% tpi = pi;   // tracking reference to native pointer
```

```
   MyStruct ^ x2 = gcnew MyStruct;
   MyStruct ^% y2 = x2;   // tracking reference to value object

   MyStruct z;
   int %tk = z.k;   // tracking reference to member of value type

   delete[] pi;
}
```

The following sample shows how to bind a tracking reference to an array.

```
// tracking_reference_2.cpp
// compile with: /clr
using namespace System;

int main() {
   array<int> ^ a = gcnew array< Int32 >(5);
   a[0] = 21;
   Console::WriteLine(a[0]);
   array<int> ^% arr = a;
   arr[0] = 222;
   Console::WriteLine(a[0]);
}
```

Output

```
21
222
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

How to: Pass CLR Types by Reference with Tracking References

The following sample shows how to pass CLR types by reference with tracking references.

Example

```
// tracking_reference_handles.cpp
// compile with: /clr
using namespace System;

ref struct City {
private:
   Int16 zip_;

public:
   City (int zip) : zip_(zip) {};
   property Int16 zip {
      Int16 get(void) {
         return zip_;
      }   // get
   }   // property
};

void passByRef (City ^% myCity) {
   // cast needed so this pointer in City struct is "const City"
   if (myCity->zip == 20100)
      Console::WriteLine("zip == 20100");
   else
      Console::WriteLine("zip != 20100");
}

ref class G {
public:
   int i;
};

void Test(int % i) {
   i++;
}

int main() {
   G ^ g1 = gcnew G;
   G ^% g2 = g1;
   g1 -> i = 12;

   Test(g2->i);   // g2->i will be changed in Test2()

   City ^ Milano = gcnew City(20100);
   passByRef(Milano);
}
```

Output

```
zip == 20100
```

See Also

Reference

% (Tracking Reference)

How to: Using Tracking References and Interior Pointers

The following sample shows that taking the address of a tracking reference returns an interior_ptr and how to modify and access data through a tracking reference.

Example

```
// tracking_reference_data.cpp
// compile with: /clr
using namespace System;

public ref class R {
public:
   R(int i) : m_i(i) {
      Console::WriteLine("ctor: R(int)");
   }

   int m_i;
};

class N {
public:
   N(int i) : m_i (i) {
      Console::WriteLine("ctor: N(int i)");
   }

   int m_i;
};

int main() {
   R ^hr = gcnew R('r');
   R ^%thr = hr;
   N n('n');
   N %tn = n;

   // Declare interior pointers
   interior_ptr<R^> iphr = &thr;
   interior_ptr<N> ipn = &tn;

   // Modify data through interior pointer
   (*iphr)->m_i = 1;   // (*iphr)->m_i == thr->m_i
   ipn->m_i = 4;   // ipn->m_i == tn.m_i

   ++thr-> m_i;   // hr->m_i == thr->m_i
   ++tn. m_i;   // n.m_i == tn.m_i

   ++hr-> m_i;   // (*iphr)->m_i == hr->m_i
   ++n. m_i;   // ipn->m_i == n.m_i
}
```

Output

```
ctor: R(int)
ctor: N(int i)
```

The following code sample shows that you can convert between tracking references and interior pointers.

```
// tracking_reference_interior_ptr.cpp
// compile with: /clr
using namespace System;

public ref class R {
public:
   R(int i) : m_i(i) {
      Console::WriteLine("ctor: R(int)");
```

```
   }

   int m_i;
};

class N {
public:
   N(int i) : m_i(i) {
      Console::WriteLine("ctor: N(int i)");
   }

   int m_i;
};

int main() {
   R ^hr = gcnew R('r');
   N n('n');

   R ^%thr = hr;
   N %tn = n;

   // Declare interior pointers
   interior_ptr<R^> iphr = &hr;
   interior_ptr<N> ipn = &n;

   // Modify data through interior pointer
   (*iphr)->m_i = 1;   // (*iphr)-> m_i == thr->m_i
   ipn->m_i = 4;   // ipn->m_i == tn.m_i

   ++thr->m_i;   // hr->m_i == thr->m_i
   ++tn.m_i;   // n.m_i == tn.m_i

   ++hr->m_i;   // (*iphr)->m_i == hr->m_i
   ++n.m_i;   // ipn->m_i == n.m_i
}
```

Output

```
ctor: R(int)
ctor: N(int i)
```

See Also

Reference

% (Tracking Reference)

How to: Use Tracking References and Value Types

The following sample shows simple boxing through a tracking reference to a value type:

Example

```
// tracking_reference_valuetypes_1.cpp
// compile with: /clr
using namespace System;
int main() {
   int i = 10;
   int % j = i;
   Object ^ o = j;   // j is implicitly boxed and assigned to o
}
```

The following sample shows you can have both tracking references and native references to value types.

```
// tracking_reference_valuetypes_2.cpp
// compile with: /clr
using namespace System;
int main() {
   int i = 10;
   int & j = i;
   int % k = j;
   i++;   // 11
   j++;   // 12
   k++;   // 13
   Console::WriteLine(i);
   Console::WriteLine(j);
   Console::WriteLine(k);
}
```

Output

```
13
13
13
```

The following sample shows that tracking references can be used with value types and native types.

```
// tracking_reference_valuetypes_3.cpp
// compile with: /clr
value struct G {
   int i;
};

struct H {
   int i;
};

int main() {
   G g;
   G % v = g;
   v.i = 4;
   System::Console::WriteLine(v.i);
   System::Console::WriteLine(g.i);

   H h;
   H % w = h;
   w.i = 5;
   System::Console::WriteLine(w.i);
   System::Console::WriteLine(h.i);
}
```

Output

```
4
4
5
5
```

The following sample shows that you can bind a tracking reference to a value type on the garbage-collected heap.

```
// tracking_reference_valuetypes_4.cpp
// compile with: /clr
using namespace System;
value struct V {
   int i;
};

void Test(V^ hV) {   // hv boxes another copy of original V on GC heap
   Console::WriteLine("Boxed new copy V: {0}", hV->i);
}

int main() {
   V v;   // V on the stack
   v.i = 1;
   V ^hV1 = v;   // v is boxed and assigned to hV1
   v.i = 2;
   V % trV = *hV1;   // trV is bound to boxed v, the v on the gc heap.
   Console::WriteLine("Original V: {0}, Tracking reference to boxed V: {1}", v.i, trV.i);
   V ^hV2 = trV;   // hv2 boxes another copy of boxed v on the GC heap
   hV2->i = 3;
   Console::WriteLine("Tracking reference to boxed V: {0}", hV2->i);
   Test(trV);
   v.i = 4;
   V ^% trhV = hV1;  // creates tracking reference to boxed type handle
   Console::WriteLine("Original V: {0}, Reference to handle of originally boxed V: {1}", v.
i, trhV->i);
}
```

Output

```
Original V: 2, Tracking reference to boxed V: 1
Tracking reference to boxed V: 3
Boxed new copy V: 1
Original V: 4, Reference to handle of originally boxed V: 1
```

See Also

Reference

% (Tracking Reference)

How to: Cannot Use Tracking References and Unary "Take-Address" Operator

The following sample shows that a tracking reference cannot be used as a unary take-address operator.

Example

```
// tracking_reference_unary.cpp
// compile with: /clr
using namespace System;
ref struct R {
   static R^ operator%(R ^r) {   // C2805 can't declare unary operator%()
      return nullptr;
   }
   static int operator&(R ^r) {  // can declare a unary operator&()
      return 1;
   }
};

int main() {
   int i;
   int* pi1 = &i; // OK: C++ address of operator&()
   int* pi2 = %i; // ERROR: No managed address of operator%()
   int^ hi = %i; // ERROR: No managed address of operator%()
}
```

See Also

Reference

% (Tracking Reference)

How to: Write Template Functions that Take Native, Value, or Reference Parameters

By using a tracking reference in the signature of a template function, you ensure that the function can be called with parameters whose type are native, CLR value, or CLR reference.

Example

```
// tracking_reference_template.cpp
// compile with: /clr
using namespace System;

class Temp {
public:
   // template functions
   template<typename T>
   static int f1(T% tt) {   // works for object in any location
      Console::WriteLine("T %");
      return 0;
   }

   template<typename T>
   static int f2(T& rt) {   // won't work for object on the gc heap
      Console::WriteLine("T &");
      return 1;
   }
};

// Class Defintions
ref struct R {
   int i;
};

int main() {
   R ^hr = gcnew R;
   int i = 1;

   Temp::f1(i); // ok
   Temp::f1(hr->i); // ok
   Temp::f2(i); // ok

   // error can't track object on gc heap with a native reference
   // Temp::f2(hr->i);
}
```

Output

```
T %
T %
T &
```

See Also

Reference

% (Tracking Reference)

^ (Handle to Object on Managed Heap)

Declares a handle to an object on the managed heap.

Remarks

A handle to an object on the managed heap points to the "whole" object, and not to a member of the object.

See gcnew for information on how to create an object on the managed heap.

In Visual C++ 2002 and Visual C++ 2003, __gc * was used to declare an object on the managed heap. The ^ replaces __gc * in the new syntax.

The common language runtime maintains a separate heap on which it implements a precise, asynchronous, compacting garbage collection scheme. To work correctly, it must track all storage locations that can point into this heap at runtime. ^ provides a handle through which the garbage collector can track a reference to an object on the managed heap, thereby being able to update it whenever that object is moved.

Because native C++ pointers (*) and references (&) cannot be tracked precisely, a handle-to object declarator is used.

Member selection through a handle (^) uses the pointer-to-member operator (->).

For more information, see How to: Declare Handles in Native Types.

Example

This sample shows how to create an instance of reference type on the managed heap. This sample also shows that you can initialize one handle with another, resulting in two references to same object on managed, garbage-collected heap. Notice that assigning nullptr to one handle does not mark the object for garbage collection.

```
// mcppv2_handle.cpp
// compile with: /clr
ref class MyClass {
public:
   MyClass() : i(){}
   int i;
   void Test() {
      i++;
      System::Console::WriteLine(i);
   }
};

int main() {
   MyClass ^ p_MyClass = gcnew MyClass;
   p_MyClass->Test();

   MyClass ^ p_MyClass2;
   p_MyClass2 = p_MyClass;

   p_MyClass = nullptr;
   p_MyClass2->Test();
}
```

Output

```
1
2
```

The following sample shows how to declare a handle to an object on the managed heap, where the type of object is a boxed value type. The sample also shows how to get the value type from the boxed object.

```
// mcppv2_handle_2.cpp
// compile with: /clr
using namespace System;

void Test(Object^ o) {
   Int32^ i = dynamic_cast<Int32^>(o);
```

```
   if(i)
      Console::WriteLine(i);
   else
      Console::WriteLine("Not a boxed int");
}

int main() {
   String^ str = "test";
   Test(str);

   int n = 100;
   Test(n);
}
```

Output

```
Not a boxed int
100
```

This sample shows that the common C++ idiom of using a void* pointer to point to an arbitrary object is replaced by Object^, which can hold a handle to any reference class. It also shows that all types, such as arrays and delegates, can be converted to an object handle.

```
// mcppv2_handle_3.cpp
// compile with: /clr
using namespace System;
using namespace System::Collections;
public delegate void MyDel();
ref class MyClass {
public:
   void Test() {}
};

void Test(Object ^ x) {
   Console::WriteLine("Type is {0}", x->GetType());
}

int main() {
   // handle to Object can hold any ref type
   Object ^ h_MyClass = gcnew MyClass;

   ArrayList ^ arr = gcnew ArrayList();
   arr->Add(gcnew MyClass);

   h_MyClass = dynamic_cast<MyClass ^>(arr[0]);
   Test(arr);

   Int32 ^ bi = 1;
   Test(bi);

   MyClass ^ h_MyClass2 = gcnew MyClass;

   MyDel^ DelInst = gcnew MyDel(h_MyClass2, &MyClass::Test);
   Test(DelInst);
}
```

Output

```
Type is System.Collections.ArrayList
Type is System.Int32
Type is MyDel
```

This sample shows that a handle can be dereferenced and that a member can be accessed via a dereferenced handle.

```
// mcppv2_handle_4.cpp
// compile with: /clr
```

```
using namespace System;
value struct DataCollection {
private:
   int Size;
   array<String^>^ x;

public:
   DataCollection(int i) : Size(i) {
      x = gcnew array<String^>(Size);
      for (int i = 0 ; i < Size ; i++)
         x[i] = i.ToString();
   }

   void f(int Item) {
      if (Item >= Size)
      {
         System::Console::WriteLine("Cannot access array element {0}, size is {1}", Item, S
ize);
         return;
      }
      else
         System::Console::WriteLine("Array value: {0}", x[Item]);
   }
};

void f(DataCollection y, int Item) {
   y.f(Item);
}

int main() {
   DataCollection ^ a = gcnew DataCollection(10);
   f(*a, 7);   // dereference a handle, return handle's object
   (*a).f(11);   // access member via dereferenced handle
}
```

Output

```
Array value: 7
Cannot access array element 11, size is 10
```

This sample shows that a native reference (&) can't bind to an int member of a managed type, .as the int may be stored in the garbage collected heap, and native references don't track object movement in the managed heap. The fix is to use a local variable, or to change & to %, making it a tracking reference.

```
// mcppv2_handle_5.cpp
// compile with: /clr
ref struct A {
   void Test(unsigned int &){}
   void Test2(unsigned int %){}
   unsigned int i;
};

int main() {
   A a;
   a.i = 9;
   a.Test(a.i);   // C2664
   a.Test2(a.i);   // OK

   unsigned int j = 0;
   a.Test(j);   // OK
}
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

abstract (Visual C++)

abstract is a context sensitive keyword that can indicate:

- A member can only be defined in a derived type.
- A type cannot be instantiated (can only act as a base type).

Remarks

Marking a function abstract is the same as making it a pure virtual function. Making a member function abstract causes the enclosing class to also be marked abstract.

abstract is also valid when compiling for native targets (without /clr).

See Override Specifiers and Native Compilations for more information.

You can detect at compile time if a type is abstract with **__is_abstract(***type***)**. For more information, see Compiler Support for Type Traits.

abstract is a context-sensitive keyword; see Context-Sensitive Keywords for more information.

Example

The following sample will generate an error because class X is marked **abstract**.

```
// abstract_keyword.cpp
// compile with: /clr
ref class X abstract {
public:
   virtual void f() {}
};

int main() {
   X ^ MyX = gcnew X;   // C3622 cannot instantiate abstract class
}
```

The following sample shows that the compiler will generate an error because a native class is marked **abstract**, when compiled with **/clr.**

```
// abstract_keyword_2.cpp
class X abstract {
public:
   virtual void f() {}
};

int main() {
   X * MyX = new X;   // C3622
}
```

The following sample will generate an error because function f is marked **abstract**.

```
// abstract_keyword_3.cpp
// compile with: /clr
ref class X {
public:
   virtual void f() abstract {}   // C3634
   virtual void g() = 0 {}   // C3634
};
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

array (Visual C++)

The **array** keyword lets you create a dynamic array that is allocated on the common language runtime heap.

```
[qualifiers] [cli::]array<[qualifiers]type1[, dimension]>^ var =
gcnew [cli::]array<type2[, dimension]>(val[,val...])
```

Parameters

dimension [optional]

The number of dimensions of the array. The default is 1. The maximum is 32.

qualifiers [optional]

A storage class specifier. Valid keywords for *qualifier* are: mutable, volatile, const, extern, and static.

val

A comma-separated list of the size of your array or an aggregate initializer.

There should be one size *val* for each *dimension*. For more information, see Multidimension Arrays.

type1

The type of the array variable. Valid types are managed reference types (*type***^**), managed value types (*type*), and native pointers (*type******). A tracking handle is always required after the closing angle bracket (>) in the declaration.

type2

The type of the values initializing the array. Commonly, *type1* and *type2* will be the same types. However, it is possible for the types to be different, as long as there is a conversion from *type2* to *type1*. For example, if *type2* is derived from *type1*.

Like *type1*, valid types are managed reference types, managed value types, and native pointers. However, a tracking handle is not allowed after the closing angle bracket (>).

var

The name of the array variable.

Remarks

The number of elements of the array is not part of the type. A single array variable can refer to arrays of different sizes.

Like standard C++, the indices of a managed array are zero-based, and a managed array is subscripted using ordinary C++ array brackets. Unlike standard C++, subscripting is not a synonym for pointer arithmetic and is not commutative.

A managed array is itself a managed object. It is actually a pointer into the common language runtime heap. As a managed object, it has the same restrictions as a managed class. Most notably, the element type cannot be a native C++ class that is not a POD type.

array is in the cli namespace. See cli Namespace for more information.

All managed arrays inherit from **System::Array**. Any method or property of **System::Array** can be applied directly to the array variable.

When allocating an array whose element type is pointer-to a managed class, the elements are 0-initialized.

When allocating an array whose element type is a value type `V`, the default constructor for `V` is applied to each array element. For more information, see .NET Framework Equivalents to C++ Native Types.

You can detect at compile time if a type is a CLR array with **__is_ref_array(***type***)**. For more information, see Compiler Support for Type Traits.

For more information on arrays, see

- Array Covariance
- How to: Create Single-Dimension Arrays

- How to: Create Multidimension Arrays
- How to: Create Arrays of Managed Arrays (Jagged Arrays)
- How to: Make Typedefs for Managed Arrays
- How to: Use Managed Arrays as Template Type Parameters
- How to: Accept Variable Arguments
- How to: Sort Arrays
- How to: Sort Arrays Using Custom Criteria

Example

```
// clr_array.cpp
// compile with: /clr
ref class MyClass {};
int main() {
   array<MyClass ^> ^ MyArray = gcnew array<MyClass ^>(100);
}
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

Array Covariance

Given reference class D with direct or indirect base class B, an array of type D can be assigned to an array variable of type B.

```
// clr_array_covariance.cpp
// compile with: /clr
using namespace System;

int main() {
   // String derives from Object
   array<Object^>^ oa = gcnew array<String^>(20);
}
```

Remarks

An assignment to an array element shall be assignment-compatible with the dynamic type of the array. An assignment to an array element with incompatible type will cause System::ArrayTypeMismatchException to be thrown.

Array covariance does not apply to arrays of value class type. For example, arrays of Int32 cannot be converted to Object^ arrays, even via boxing.

Example

```
// clr_array_covariance2.cpp
// compile with: /clr
using namespace System;

ref struct Base { int i; };
ref struct Derived  : Base {};
ref struct Derived2 : Base {};
ref struct Derived3 : Derived {};
ref struct Other { short s; };

int main() {
   // Derived* d[] = new Derived*[100];
   array<Derived^> ^ d = gcnew array<Derived^>(100);

   // ok by array covariance
   array<Base ^> ^  b = d;

   // invalid
   // b[0] = new Other;

   // error (runtime exception)
   // b[1] = gcnew Derived2;

   // error (runtime exception),
   // must be "at least" a Derived.
   // b[0] = gcnew Base;

   b[1] = gcnew Derived;
   b[0] = gcnew Derived3;
}
```

See Also

Reference

array (Visual C++)

How to: Create Single-Dimension Arrays

The first sample shows how to create single-dimension arrays of reference, value, and native pointer types. It also shows how to return a single-dimension array from a function and how to pass a single-dimension array as an argument to a function.

The second sample shows how to perform aggregate initialization.

Example

```
// mcppv2_sdarrays.cpp
// compile with: /clr
using namespace System;

#define ARRAY_SIZE 2

value struct MyStruct {
   int m_i;
};

ref class MyClass {
public:
   int m_i;
};

struct MyNativeClass {
   int m_i;
};

// Returns a managed array of a reference type.
array<MyClass^>^ Test0() {
   int i;
   array< MyClass^ >^ local = gcnew array< MyClass^ >(ARRAY_SIZE);

   for (i = 0 ; i < ARRAY_SIZE ; i++) {
      local[i] = gcnew MyClass;
      local[i] -> m_i = i;
   }
   return local;
}


// Returns a managed array of Int32.
array<Int32>^ Test1() {
   int i;
   array< Int32 >^ local = gcnew array< Int32 >(ARRAY_SIZE);

   for (i = 0 ; i < ARRAY_SIZE ; i++)
      local[i] = i + 10;
   return local;
}


// Modifies an array.
void Test2(array< MyNativeClass * >^ local) {
   for (int i = 0 ; i < ARRAY_SIZE ; i++)
      local[i] -> m_i = local[i] -> m_i + 2;
}

int main() {
   int i;

   // Declares an array of user-defined reference types
   // and initializes with a function.
   array< MyClass^ >^ MyClass0;
   MyClass0 = Test0();
```

```
    for (i = 0 ; i < ARRAY_SIZE ; i++)
       Console::WriteLine("MyClass0[{0}] = {1}", i, MyClass0[i] -> m_i);
    Console::WriteLine();


    // Declares an array of value types and initializes with a function.
    array< Int32 >^ IntArray;
    IntArray = Test1();

    for (i = 0 ; i < ARRAY_SIZE ; i++)
       Console::WriteLine("IntArray[{0}] = {1}", i, IntArray[i]);
    Console::WriteLine();


    // Declares and initializes an array of user-defined
    // reference types.
    array< MyClass^ >^ MyClass1 = gcnew array< MyClass^ >(ARRAY_SIZE);
    for (i = 0 ; i < ARRAY_SIZE ; i++) {
       MyClass1[i] = gcnew MyClass;
       MyClass1[i] -> m_i = i + 20;
    }

    for (i = 0 ; i < ARRAY_SIZE ; i++)
       Console::WriteLine("MyClass1[{0}] = {1}", i, MyClass1[i] -> m_i);
    Console::WriteLine();


    // Declares and initializes an array of pointers to a native type.
    array< MyNativeClass * >^ MyClass2 = gcnew array<
       MyNativeClass * >(ARRAY_SIZE);
    for (i = 0 ; i < ARRAY_SIZE ; i++) {
       MyClass2[i] = new MyNativeClass();
       MyClass2[i] -> m_i = i + 30;
    }

    for (i = 0 ; i < ARRAY_SIZE ; i++)
       Console::WriteLine("MyClass2[{0}] = {1}", i, MyClass2[i]->m_i);
    Console::WriteLine();

    Test2(MyClass2);
    for (i = 0 ; i < ARRAY_SIZE ; i++)
       Console::WriteLine("MyClass2[{0}] = {1}", i, MyClass2[i]->m_i);
    Console::WriteLine();

    delete[] MyClass2[0];
    delete[] MyClass2[1];

    // Declares and initializes an array of user-defined value types.
    array< MyStruct >^ MyStruct1 = gcnew array< MyStruct >(ARRAY_SIZE);
    for (i = 0 ; i < ARRAY_SIZE ; i++) {
       MyStruct1[i] = MyStruct();
       MyStruct1[i].m_i = i + 40;
    }

    for (i = 0 ; i < ARRAY_SIZE ; i++)
       Console::WriteLine("MyStruct1[{0}] = {1}", i, MyStruct1[i].m_i);
  }
```

Output

```
MyClass0[0] = 0
MyClass0[1] = 1

IntArray[0] = 10
IntArray[1] = 11

MyClass1[0] = 20
MyClass1[1] = 21
```

```
MyClass2[0] = 30
MyClass2[1] = 31

MyClass2[0] = 32
MyClass2[1] = 33

MyStruct1[0] = 40
MyStruct1[1] = 41
```

Example

The following code sample shows how to perform aggregate initialization on single dimension managed arrays.

```
// mcppv2_sdarrays_aggregate_init.cpp
// compile with: /clr
using namespace System;

ref class G {
public:
   G(int i) {}
};

value class V {
public:
   V(int i) {}
};

class N {
public:
   N(int i) {}
};

int main() {
   // Aggregate initialize a single-dimension managed array.
   array<String^>^ gc1 = gcnew array<String^>{"one", "two", "three"};
   array<String^>^ gc2 = {"one", "two", "three"};

   array<G^>^ gc3 = gcnew array<G^>{gcnew G(0), gcnew G(1), gcnew G(2)};
   array<G^>^ gc4 = {gcnew G(0), gcnew G(1), gcnew G(2)};

   array<Int32>^ value1 = gcnew array<Int32>{0, 1, 2};
   array<Int32>^ value2 = {0, 1, 2};

   array<V>^ value3 = gcnew array<V>{V(0), V(1), V(2)};
   array<V>^ value4 = {V(0), V(1), V(2)};

   array<N*>^ native1 = gcnew array<N*>{new N(0), new N(1), new N(2)};
   array<N*>^ native2 = {new N(0), new N(1), new N(2)};
}
```

See Also

Reference

array (Visual C++)

How to: Create Multidimension Arrays

The first sample shows how to create multi-dimension arrays of reference, value, and native pointer types. It also shows how to return a multi-dimension array from a function and how to pass a multi-dimension array as an argument to a function.

The second sample shows how to perform aggregate initialization on a multi-dimension managed array.

Example

```
// mcppv2_mdarrays.cpp
// compile with: /clr
using namespace System;

#define ARRAY_SIZE 2

ref class MyClass {
public:
   int m_i;
};

// Returns a multidimensional managed array of a reference type.
array<MyClass^, 2>^ Test0() {
   int i, j;
   array< MyClass^, 2 >^ local = gcnew array< MyClass^,
      2 >(ARRAY_SIZE, ARRAY_SIZE);

   for (i = 0 ; i < ARRAY_SIZE ; i++)
      for (j = 0 ; j < ARRAY_SIZE ; j++) {
         local[i,j] = gcnew MyClass;
         local[i,j] -> m_i = i;
      }

      return local;
}

// Returns a managed array of Int32.
array<Int32, 2>^ Test1() {
   int i, j;
   array< Int32,2 >^ local = gcnew array< Int32,2 >(
        ARRAY_SIZE, ARRAY_SIZE);

   for (i = 0 ; i < ARRAY_SIZE ; i++)
         for (j = 0 ; j < ARRAY_SIZE ; j++)
            local[i,j] = i + 10;

   return local;
}

int main() {
   int i, j;

   // Declares multidimensional array of user-defined reference types
   // and initializes in a function.
   array< MyClass^, 2 >^ MyClass0;
   MyClass0 = Test0();

   for (i = 0 ; i < ARRAY_SIZE ; i++)
      for (j = 0 ; j < ARRAY_SIZE ; j++)
         Console::WriteLine("MyClass0[{0}, {1}] = {2}", i, j,
                            MyClass0[i,j] -> m_i);

   Console::WriteLine();

   // Declare an array of value types and initializes in a function.
   array< Int32, 2 >^ IntArray;
   IntArray = Test1();
```

```
   for (i = 0 ; i < ARRAY_SIZE ; i++)
         for (j = 0 ; j < ARRAY_SIZE ; j++)
            Console::WriteLine("IntArray[{0}, {1}] = {2}", i, j,
            IntArray[i,j]);
}
```

Output

```
MyClass0[0, 0] = 0
MyClass0[0, 1] = 0
MyClass0[1, 0] = 1
MyClass0[1, 1] = 1

IntArray[0, 0] = 10
IntArray[0, 1] = 10
IntArray[1, 0] = 11
IntArray[1, 1] = 11
```

Example

This sample shows how to perform aggregate initialization on a multi-dimension managed array.

```
// mcppv2_mdarrays_aggregate_initialization.cpp
// compile with: /clr
using namespace System;

ref class G {
public:
   G(int i) {}
};

value class V {
public:
   V(int i) {}
};

class N {
public:
   N(int i) {}
};

int main() {
   // Aggregate initialize a multidimension managed array.
   array<String^, 2>^ gc1 = gcnew array<String^, 2>{ {"one", "two"},
       {"three", "four"} };
   array<String^, 2>^ gc2 = { {"one", "two"}, {"three", "four"} };

   array<G^, 2>^ gc3 = gcnew array<G^, 2>{ {gcnew G(0), gcnew G(1)},
       {gcnew G(2), gcnew G(3)} };
   array<G^, 2>^ gc4 = { {gcnew G(0), gcnew G(1)}, {gcnew G(2), gcnew G(3)} };

   array<Int32, 2>^ value1 = gcnew array<Int32, 2>{ {0, 1}, {2, 3} };
   array<Int32, 2>^ value2 = { {0, 1}, {2, 3} };

   array<V, 2>^ value3 = gcnew array<V, 2>{ {V(0), V(1)}, {V(2), V(3)} };
   array<V, 2>^ value4 = { {V(0), V(1)}, {V(2), V(3)} };

   array<N*, 2>^ native1 = gcnew array<N*, 2>{ {new N(0), new N(1)},
      {new N(2), new N(3)} };
   array<N*, 2>^ native2 = { {new N(0), new N(1)}, {new N(2), new N(3)} };
}
```

See Also

Reference

array (Visual C++)

How to: Create Arrays of Managed Arrays (Jagged Arrays)

These samples show how to create single-dimension arrays of managed array of reference, value, and native pointer types. It also shows how to return a single-dimension array of managed array from a function and how to pass a single-dimension array as an argument to a function.

Example

```
// mcppv2_array_of_arrays.cpp
// compile with: /clr
using namespace System;

#define ARRAY_SIZE 2

value struct MyStruct {
   int m_i;
};

ref class MyClass {
public:
   int m_i;
};

// Returns an array of managed arrays of a reference type.
array<array<MyClass^>^>^ Test0() {
   int size_of_array = 4;
   array<array<MyClass^>^>^ local = gcnew
      array<array<MyClass^>^>(ARRAY_SIZE);

   for (int i = 0 ; i < ARRAY_SIZE ; i++, size_of_array += 4) {
      local[i] = gcnew array<MyClass^>(size_of_array);
      for (int k = 0; k < size_of_array ; k++) {
         local[i][k] = gcnew MyClass;
         local[i][k] -> m_i = i;
      }
   }

   return local;
}

// Returns a managed array of Int32.
array<array<Int32>^>^ Test1() {
   int i;
   array<array<Int32>^>^ local = gcnew array<array< Int32 >^>(ARRAY_SIZE);

   for (i = 0 ; i < ARRAY_SIZE ; i++) {
      local[i] = gcnew array< Int32 >(ARRAY_SIZE);
         for ( int j = 0 ; j < ARRAY_SIZE ; j++ )
            local[i][j] = i + 10;
   }
   return local;
}

int main() {
   int i, j;

   // Declares an array of user-defined reference types
   // and initializes with a function.
   array< array< MyClass^ >^ >^ MyClass0;
   MyClass0 = Test0();

   for (i = 0 ; i < ARRAY_SIZE ; i++)
      for ( j = 0 ; j < ARRAY_SIZE ; j++ )
         Console::WriteLine("MyClass0[{0}] = {1}", i, MyClass0[i][j] -> m_i);
   Console::WriteLine();
```

```
   // Declares an array of value types and initializes with a function.
   array< array< Int32 >^ >^ IntArray;
   IntArray = Test1();

   for (i = 0 ; i < ARRAY_SIZE ; i++)
      for (j = 0 ; j < ARRAY_SIZE ; j++)
      Console::WriteLine("IntArray[{0}] = {1}", i, IntArray[i][j]);
   Console::WriteLine();

   // Declares and initializes an array of user-defined value types.
   array< MyStruct >^ MyStruct1 = gcnew array< MyStruct >(ARRAY_SIZE);
   for (i = 0 ; i < ARRAY_SIZE ; i++) {
      MyStruct1[i] = MyStruct();
      MyStruct1[i].m_i = i + 40;
   }

   for (i = 0 ; i < ARRAY_SIZE ; i++)
      Console::WriteLine(MyStruct1[i].m_i);
}
```

Output

```
MyClass0[0] = 0
MyClass0[0] = 0
MyClass0[1] = 1
MyClass0[1] = 1

IntArray[0] = 10
IntArray[0] = 10
IntArray[1] = 11
IntArray[1] = 11

40
41
```

Example

The following sample shows how to perform aggregate initialization with jagged arrays.

```
// mcppv2_array_of_arrays_aggregate_init.cpp
// compile with: /clr
using namespace System;
#define ARRAY_SIZE 2
int size_of_array = 4;
int count = 0;

ref class MyClass {
public:
   int m_i;
};

struct MyNativeClass {
   int m_i;
};

int main() {
   // Declares an array of user-defined reference types
   // and performs an aggregate initialization
   array< array< MyClass^ >^ >^ MyClass0 = gcnew array<array<MyClass^>^> {
      gcnew array<MyClass^>{ gcnew MyClass(), gcnew MyClass() },
      gcnew array<MyClass^>{ gcnew MyClass(), gcnew MyClass() }
   };

   for ( int i = 0 ; i < ARRAY_SIZE ; i++, size_of_array += 4 )
      for ( int k = 0 ; k < ARRAY_SIZE ; k++ )
         MyClass0[i][k] -> m_i = i;
```

```
    for ( int i = 0 ; i < ARRAY_SIZE ; i++ )
       for ( int j = 0 ; j < ARRAY_SIZE ; j++ )
          Console::WriteLine("MyClass0[{0}] = {1}", i, MyClass0[i][j] -> m_i);
    Console::WriteLine();

    // Declares an array of value types and performs an aggregate initialization
    array< array< Int32 >^ >^ IntArray = gcnew array<array< Int32 >^> {
       gcnew array<Int32>{1,2},
       gcnew array<Int32>{3,4,5}
    };

    for each ( array<int>^ outer in IntArray ) {
       Console::Write("[");

       for each( int i in outer )
          Console::Write(" {0}", i);

       Console::Write(" ]");
       Console::WriteLine();
    }
    Console::WriteLine();

    // Declares and initializes an array of pointers to a native type.
    array<array< MyNativeClass * >^ > ^ MyClass2 =
       gcnew array<array< MyNativeClass * > ^> {
          gcnew array<MyNativeClass *>{ new MyNativeClass(), new MyNativeClass() },
          gcnew array<MyNativeClass *>{ new MyNativeClass(), new MyNativeClass(), new MyNati
veClass() }
       };

    for each ( array<MyNativeClass *> ^ outer in MyClass2 )
       for each( MyNativeClass* i in outer )
          i->m_i = count++;

    for each ( array<MyNativeClass *> ^ outer in MyClass2 ) {
       Console::Write("[");
       for each( MyNativeClass* i in outer )
          Console::Write(" {0}", i->m_i);
       Console::Write(" ]");
       Console::WriteLine();
    }
    Console::WriteLine();


    // Declare and initialize array of two-dimensional arrays of strings
    array<array<String ^,2> ^> ^gc3 = gcnew array<array<String ^,2> ^>{
       gcnew array<String ^>{ {"a","b"}, {"c", "d"}, {"e","f"} },
       gcnew array<String ^>{ {"g", "h"} }
    };

    for each ( array<String^, 2> ^ outer in gc3 ){
       Console::Write("[");
       for each( String ^ i in outer )
          Console::Write(" {0}", i);
       Console::Write(" ]");
       Console::WriteLine();
    }
 }
```

Output

```
MyClass0[0] = 0
MyClass0[0] = 0
MyClass0[1] = 1
MyClass0[1] = 1

[ 1 2 ]
[ 3 4 5 ]
```

```
[ 0 1 ]
[ 2 3 4 ]

[ a b c d e f ]
[ g h ]
```

See Also

Reference

array (Visual C++)

How to: Make Typedefs for Managed Arrays

This example shows how to make a typedef for a managed array.

Example

```
// mcppv2_typedef_arrays.cpp
// compile with: /clr
using namespace System;
ref class G {};

typedef array<array<G^>^> jagged_array;

int main() {
   jagged_array ^ MyArr = gcnew jagged_array (10);
}
```

See Also

Reference

array (Visual C++)

How to: Use Managed Arrays as Template Type Parameters

This example shows how to use a managed array as a parameter to a template.

Example

```
// mcppv2_template_type_params.cpp
// compile with: /clr
using namespace System;
template <class T>
class TA {
public:
   array<array<T>^>^ f() {
      array<array<T>^>^ larr = gcnew array<array<T>^>(10);
      return larr;
   }
};

int main() {
   int retval = 0;
   TA<array<array<Int32>^>^>* ta1 = new TA<array<array<Int32>^>^>();
   array<array<array<array<Int32>^>^>^>^ larr = ta1->f();
   retval += larr->Length - 10;
   Console::WriteLine("Return Code: {0}", retval);
}
```

Output

```
Return Code: 0
```

See Also

Reference

array (Visual C++)

How to: Accept Variable Arguments

This example shows how functions with a variable number of arguments can be implemented in Visual C++ using the `...` syntax.

The parameter that uses `...` must be the last parameter in the parameter list.

Example

```
// mcppv2_paramarray.cpp
// compile with: /clr
using namespace System;
double average( ... array<Int32>^ arr ) {
   int i = arr->GetLength(0);
   double answer = 0.0;

   for (int j = 0 ; j < i ; j++)
      answer += arr[j];

   return answer / i;
}

int main() {
   Console::WriteLine("{0}", average( 1, 2, 3, 6 ));
}
```

Output

```
3
```

Legacy Code Example

The following example shows how to call a Visual C++ function that takes a variable number of arguments from C#.

```
// mcppv2_paramarray2.cpp
// compile with: /clr:safe /LD
using namespace System;

public ref class C {
public:
   void f( ... array<String^>^ a ) {}
};
```

The function `f` can be called from C# or Visual Basic, for example, as though it were a function that can take a variable number of arguments.

In C#, an argument passed to a *ParamArray* parameter can be called with a variable number of arguments. The following code sample is in C#.

```
// mcppv2_paramarray3.cs
// compile with: /r:mcppv2_paramarray2.dll
// a C# program

public class X {
   public static void Main() {
      // Visual C# will generate a String array to match the
      // ParamArray attribute
      C myc = new C();
      myc.f("hello", "there", "world");
   }
}
```

A call to `f` in Visual C++ can pass an initialized array as well as a variable-length array:

```
// mcpp_paramarray4.cpp
// compile with: /clr
using namespace System;

public ref class C {
public:
   void f( ... array<String^>^ a ) {}
};

int main() {
   C ^ myc = gcnew C();
   myc->f("hello", "world", "!!!");
}
```

See Also

Reference

array (Visual C++)

How to: Sort Arrays

Unlike standard C++ arrays, managed arrays are implicitly derived from an array base-class from which they inherit common behaviors. A prime example is the **Sort** method, which can be used to order the items in any array.

For arrays containing simple intrinsic types, you can call the **Sort** method. The sort criteria can be overridden, which is necessary to sort for arrays of complex types. In this case, the array element type must implement the IComparable::CompareTo method.

Example

```
using namespace System;

int main() {
   array<int>^ a = { 5, 4, 1, 3, 2 };
   Array::Sort( a );
   for (int i=0; i < a->Length; i++)
      Console::Write("{0} ", a[i] );
}
```

See Also

Tasks

How to: Sort Arrays Using Custom Criteria

Reference

Array Class

How to: Sort Arrays Using Custom Criteria

To sort arrays containing simple intrinsic types, you can simply call the **Array::Sort** method, but to sort arrays containing complex types or to override the default sort criteria, you can override the IComparable::CompareTo method.

In the following example, a structure called `Element` is derived from IComparable, and written to provide a CompareTo method that uses the average of two integers as the sort criteria.

Example

```
using namespace System;

value struct Element : public IComparable {
   int v1, v2;

   int CompareTo(Object^ obj) {
      Element^ o = dynamic_cast<Element^>(obj);
      if (o) {
         int thisAverage = (v1 + v2) / 2;
         int thatAverage = (o->v1 + o->v2) / 2;
         if (thisAverage < thatAverage)
            return -1;
         else if (thisAverage > thatAverage)
            return 1;
         return 0;
         }
      else
         throw new ArgumentException
      ("Object must be of type 'Element'");
   }
};

int main() {
   array<Element>^ a = gcnew array<Element>(10);
   Random^ r = new Random;

   for (int i=0; i < a->Length; i++) {
      a[i].v1 = r->Next() % 100;
      a[i].v2 = r->Next() % 100;
   }

   Array::Sort( a );
   for (int i=0; i < a->Length; i++) {
      int v1 = a[i].v1;
      int v2 = a[i].v2;
      int v = (v1 + v2) / 2;
      Console::WriteLine("{0}  (({1}+{2})/2) ", v, v1, v2);
   }
}
```

See Also

Reference

Array Class

C-Style Casts with /clr

When used with CLR types, the compiler attempts to map C-style cast to one of the casts listed below, in the following order:

1. const_cast
2. safe_cast
3. safe_cast plus const_cast
4. static_cast
5. static_cast plus const_cast

If none of the casts listed above is valid, and if the type of the expression and the target type are CLR reference types, C-style cast maps to a runtime-check (castclass MSIL instruction). Otherwise, a C-style cast is considered invalid and the compiler issues an error.

Remarks

A C-style cast is not recommended. When compiling with /clr (Common Language Runtime Compilation), use safe_cast.

The following sample shows a C-style cast that maps to a **const_cast**.

```
// cstyle_casts_1.cpp
// compile with: /clr
using namespace System;

ref struct R {};
int main() {
   const R^ constrefR = gcnew R();
   R^ nonconstR = (R^)(constrefR);
}
```

The following sample shows a C-style cast that maps to a **safe_cast**.

```
// cstyle_casts_2.cpp
// compile with: /clr
using namespace System;
int main() {
   Object ^ o = "hello";
   String ^ s = (String^)o;
}
```

The following sample shows a C-style cast that maps to a **safe_cast** plus **const_cast**.

```
// cstyle_casts_3.cpp
// compile with: /clr
using namespace System;

ref struct R {};
ref struct R2 : public R {};

int main() {
   const R^ constR2 = gcnew R2();
   try {
   R2^ b2DR = (R2^)(constR2);
   }
   catch(InvalidCastException^ e) {
      System::Console::WriteLine("Invalid Exception");
   }
}
```

The following sample shows a C-style cast that maps to a **static_cast**.

```
// cstyle_casts_4.cpp
// compile with: /clr
using namespace System;

struct N1 {};
struct N2 {
   operator N1() {
      return N1();
   }
};

int main() {
   N2 n2;
   N1 n1 ;
   n1 = (N1)n2;
}
```

The following sample shows a C-style cast that maps to a **static_cast** plus **const_cast**.

```
// cstyle_casts_5.cpp
// compile with: /clr
using namespace System;
struct N1 {};

struct N2 {
   operator const N1*() {
      static const N1 n1;
      return &n1;
   }
};

int main() {
   N2 n2;
   N1* n1 = (N1*)(const N1*)n2;   // const_cast + static_cast
}
```

The following sample shows a C-style cast that maps to a run-time check.

```
// cstyle_casts_6.cpp
// compile with: /clr
using namespace System;

ref class R1 {};
ref class R2 {};

int main() {
   R1^ r  = gcnew R1();
   try {
      R2^ rr = ( R2^)(r);
   }
   catch(System::InvalidCastException^ e) {
      Console::WriteLine("Caught expected exception");
   }
}
```

The following sample shows an invalid C-style cast, which causes the compiler to issue an error.

```
// cstyle_casts_7.cpp
// compile with: /clr
using namespace System;
int main() {
   String^s = S"hello";
   int i = (int)s;   // C2440
}
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

Classes and Structs (Managed)

Declares a managed class or struct.

```
class_access ref class name modifier : inherit_access base_type { };
class_access ref struct name modifier : inherit_access base_type {};
class_access value class name modifier : inherit_access base_type {};
class_access value struct name modifier : inherit_access base_type {};
```

Parameters

base_type(optional)

A base type. A **ref class** or **ref struct** can inherit from zero or more managed interfaces and zero or one ref types. A **value class** or **value struct** can only inherit from zero or more managed interfaces.

class_access (optional)

The accessibility of the class or struct outside the assembly. Possible values are **public** and **private**. **private** is the default. Nested classes or structs cannot have a *class_access* specifier.

inherit_access (optional)

The accessibility of *base_type*. The only permitted accessibility for a base type is **public** (**public** is the default).

modifier (optional)

abstract and sealed are valid modifiers.

name

The name of the class or struct.

Remarks

Like their native equivalents, default member accessibility of a **ref class** or **value class** is private and default member accessibility of a **ref struct** or **value struct** is public.

A value type cannot act as a base type.

When a reference type inherits from another reference type, virtual functions in the base class must explicitly be overridden (with override) or hidden (with new (new slot in vtable)). The derived class functions must also be explicitly marked as virtual.

For more information on classes and structs, see

- Instantiating Classes and Structs
- Semantics of the this Pointer in Value and Reference Types
- C++ Stack Semantics for Reference Types
- Classes, Structures, and Unions
- public and private on Native Classes
- Implicitly Abstract Classes
- Define Static Constructors in a Class or Struct
- Copy Constructor May Not Be Generated
- Hide-by-Signature Functions in Reference Types
- Destructors and Finalizers in Visual C++
- Type and Member Visibility
- User-Defined Operators
- User-Defined Conversions

- How to: Wrap Native Class for Use by C#
- Generic Classes (Visual C++)

For information on interfaces, see interface class.

You can detect at compile time if a type is a CLR type with **__is_ref_class (*type*)**, **__is_value_class (*type*)**, or **__is_simple_value_class (*type*)**. For more information, see Compiler Support for Type Traits.

In the development environment, you can get F1 help on these keywords by highlighting the keyword, (**ref class**, for example) and pressing F1.

Example

```
// mcppv2_ref_class.cpp
// compile with: /clr /LD
ref class MyClass {
public:
   int i;

   ref class MyClass2 {
   public:
      int i;
   };

   interface struct MyInterface {
      void f();
   };
};

ref class MyClass2 : MyClass::MyInterface {
public:
   virtual void f() {
      System::Console::WriteLine("test");
   }
};

value struct MyStruct {
   void f() {
      System::Console::WriteLine("test");
   }
};
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

How to: Instantiate Classes and Structs

Reference (ref) types and value types can only be instantiated on the managed heap, not on the stack or on the native heap.

Example

```
// mcppv2_ref_class2.cpp
// compile with: /clr
ref class MyClass {
public:
   int i;

   // nested class
   ref class MyClass2 {
   public:
      int i;
   };

   // nested interface
   interface struct MyInterface {
      void f();
   };
};

ref class MyClass2 : public MyClass::MyInterface {
public:
   virtual void f() {
      System::Console::WriteLine("test");
   }
};

public value struct MyStruct {
   void f() {
      System::Console::WriteLine("test");
   }
};

int main() {
   // instantiate ref type on garbage-collected heap
   MyClass ^ p_MyClass = gcnew MyClass;
   p_MyClass -> i = 4;

   // instantiate value type on garbage-collected heap
   MyStruct ^ p_MyStruct = gcnew MyStruct;
   p_MyStruct -> f();

   // instantiate value type on the stack
   MyStruct p_MyStruct2;
   p_MyStruct2.f();

   // instantiate nested ref type on garbage-collected heap
   MyClass::MyClass2 ^ p_MyClass2 = gcnew MyClass::MyClass2;
   p_MyClass2 -> i = 5;
}
```

Output

```
test
test
```

See Also

Reference

Classes and Structs (Managed)

Semantics of the this Pointer in Value and Reference Types

When defining types using Visual C++, the **this** pointer in a reference type is of type handle. The **this** pointer in a value type is of type interior pointer.

Remarks

One area where the different semantics of the **this** pointer can result in unexpected behavior is when calling a default indexer. See the example in this topic for an example of the correct way to access a default indexer in value and reference types.

For more information, see

- ^ (Handle to Object on Managed Heap)
- interior_ptr
- How to: Use Indexed Properties

Example

This sample shows how to access a type's default indexer in both reference and value types.

```
// semantics_of_this_pointer.cpp
// compile with: /clr
using namespace System;

ref struct A {
   property Double default[Double] {
      Double get(Double data) {
         return data*data;
      }
   }

   A() {
      // accessing default indexer
      Console::WriteLine("{0}", this[3.3]);
   }
};

value struct B {
   property Double default[Double] {
      Double get(Double data) {
         return data*data;
      }
   }
   void Test() {
      // accessing default indexer
      Console::WriteLine("{0}", this->default[3.3]);
   }
};

int main() {
   A ^ mya = gcnew A();
   B ^ myb = gcnew B();
   myb->Test();
}
```

Output

```
10.89
10.89
```

See Also

Reference

Classes and Structs (Managed)

Implicitly Abstract Classes

If a class has interface base type, and if the class does not implement all the interface's member functions, then the class is implicitly abstract, and you cannot instantiate the class.

For more information about abstract classes, see abstract.

```
// mcppv2_ref_class5.cpp
// compile with: /clr
interface struct MyInterface {
   void func1();
   void func2();
};

ref class MyClass : public MyInterface {
public:
   void func1(){}
   // void func2(){}
};

int main() {
   MyClass ^ h_MyClass = gcnew MyClass;   // C2259
                        // uncomment MyClass::func2 to resolve
}
```

See Also

Reference

Classes and Structs (Managed)

C++ Stack Semantics for Reference Types

Prior to Microsoft Visual C++ 2005, an instance of a reference type could only be created using the **new** operator, which created the object on the garbage collected heap. However, beginning in Microsoft Visual C++ 2005, you can create an instance of a reference type using the same syntax that you would use to create an instance of a native type on the stack. So, you do not need to use gcnew) to create an object of a reference type and that when the object goes out of scope, the compiler will call the object's destructor.

Remarks

When you create an instance of a reference type using stack semantics, the compiler does internally create the instance on the garbage collected heap (using **gcnew**).

When the signature or return type of a function includes an instance of a by-value reference type, the function will be marked in the metadata as requiring special handling (with modreq). This special handling is currently only provided by Visual C++ clients; other languages do not currently support consuming functions or data that use reference types created with stack semantics.

One reason to use **gcnew** (dynamic allocation) instead of stack semantics would be if the type has no destructor. Also, using reference types created with stack semantics in function signatures would not be possible if you want your functions to be consumed by languages other than Visual C++.

The compiler will not generate a copy constructor for a reference type. Therefore, if you define a function that uses a by-value reference type in the signature, you must define a copy constructor for the reference type. A copy constructor in a reference type can have the following signature: `R(R%){}`.

The compiler will not generate a default assignment operator for a reference type. An assignment operator allows you to create an object using stack semantics and initialize it with an existing object created using stack semantics. An assignment operator in a reference types can have the following signature: `void operator=( R% ){}`.

If your type's destructor releases critical resources and you use stack semantics for reference types, you do not need to explicitly call the destructor (or call **delete**). For more information on destructors in reference types, see Destructors and Finalizers in Visual C++.

A compiler-generated assignment operator will follow the usual standard C++ rules with the following additions:

- Any non-static data members whose type is a handle to a reference type will be shallow copied (treated like a non-static data member whose type is a pointer).
- Any non-static data member whose type is a value type will be shallow copied.
- Any non-static data member whose type is an instance of a reference type will invoke a call to the reference type’s copy-constructor.

The compiler also provides a % unary operator to convert an instance of a reference type created using stack semantics to its underlying handle type.

The following reference types are not available for use with stack semantics:

- delegate
- array (Visual C++)
- String

Example

The following code sample shows how to declare instances of reference types with stack semantics, how the assignment operator and copy constructor works, and how to initialize a tracking reference with reference type created using stack semantics.

```
// stack_semantics_for_reference_types.cpp
// compile with: /clr
ref class R {
public:
   int i;
```

```
   R(){}

   // assignment operator
   void operator=(R% r) {
      i = r.i;
   }

   // copy constructor
   R(R% r) : i(r.i) {}
};

void Test(R r) {}   // requires copy constructor

int main() {
   R r1;
   r1.i = 98;

   R r2(r1);   // requires copy constructor
   System::Console::WriteLine(r1.i);
   System::Console::WriteLine(r2.i);

   // use % unary operator to convert instance using stack semantics
   // to its underlying handle
   R ^ r3 = %r1;
   System::Console::WriteLine(r3->i);

   Test(r1);

   R r4;
   R r5;
   r5.i = 13;
   r4 = r5;   // requires a user-defined assignment operator
   System::Console::WriteLine(r4.i);

   // initialize tracking reference
   R % r6 = r4;
   System::Console::WriteLine(r6.i);
}
```

Output

```
98
98
98
13
13
```

See Also

Reference

Classes and Structs (Managed)

How to: Define Static Constructors in a Class or Struct

A CLR type, such as a class or struct, can have a static constructor, which can be used to initialize static data members. A static constructor will be called at most once, and will be called before the first time a static member of the type is accessed.

An instance constructor will always run after a static constructor.

The compiler cannot inline a call to a constructor if the class has a static constructor. The compiler cannot inline a call to any member function if the class is a value type, has a static constructor, and does not have an instance constructor. The common language runtime may inline the call, but the compiler cannot.

A static constructor should be defined as a private member function, as the static constructor is only meant to be called by the common language runtime.

For more information on static constructors, see How to: Define an Interface Static Constructor .

Example

```
// mcppv2_ref_class6.cpp
// compile with: /clr
using namespace System;

ref class MyClass {
private:
   static int i = 0;

   static MyClass() {
      Console::WriteLine("in static constructor");
      i = 9;
   }

public:
   static void Test() {
      i++;
      Console::WriteLine(i);
   }
};

int main() {
   MyClass::Test();
   MyClass::Test();
}
```

Output

```
in static constructor
10
11
```

See Also

Reference

Classes and Structs (Managed)

How to: Declare public and private on Native Classes

It is possible for a native type to be referenced from a managed type. For example, a function in a managed type can take a parameter whose type is a native struct. If the managed type and function is public in an assembly, then the native type must also be public.

Example

```
// mcppv2_ref_class3.h
// native type
public struct N {
   N(){}
   int i;
};
```

Next, create the source code file that will consume the native type:

```
// mcppv2_ref_class3.cpp
// compile with: /clr /LD
#include "mcppv2_ref_class3.h"
// public managed type
public ref struct R {
   // public function that takes a native type
   void f(N nn) {}
};
```

Next, compile a client:

```
// mcppv2_ref_class4.cpp
// compile with: /clr
#using "mcppv2_ref_class3.dll"

#include "mcppv2_ref_class3.h"

int main() {
   R ^r = gcnew R;
   N n;
   r->f(n);
}
```

See Also

Reference

Classes and Structs (Managed)

Copy Constructor May Not Be Generated

The C++ standard says that a copy constructor will be called when an object is moved, such that, an object will be created and destroyed at the same address.

However, when compiling with **/clr**, and when a function compiled to MSIL calls a native function, passing one or more native classes by value, and where the native class has a copy constructor and/or destructor, no copy constructor will be called and the object will be destroyed at a different address than where it was created.

This could result in problems if the class has a pointer into itself, or if the code is tracking objects by address.

For more information, see /clr (Common Language Runtime Compilation).

Example

The following sample demonstrates when a copy constructor will not be generated.

```
// breaking_change_no_copy_ctor.cpp
// compile with: /clr
#include<stdio.h>

struct S {
   int i;
   static int n;

   S() : i(n++) {
      printf_s("S object %d being constructed, this=%p\n", i, this);
   }

   S(S const& rhs) : i(n++) {
      printf_s("S object %d being copy constructed from S object "
               "%d, this=%p\n", i, rhs.i, this);
   }

   ~S() {
      printf_s("S object %d being destroyed, this=%p\n", i, this);
   }
};

int S::n = 0;

#pragma managed(push,off)
void f(S s1, S s2) {
   printf_s("in function f\n");
}
#pragma managed(pop)

int main() {
   S s;
   S t;
   f(s,t);
}
```

See Also

Reference

Classes and Structs (Managed)

Hide-by-Signature Functions in Reference Types

In standard C++, a function in a base class will be hidden by a function with the same name in a derived class, even if the derived class function does not have the same number or type of parameters. This is referred to as hide-by-name semantics. In reference types, a function in a base class can only be hidden by a function in a derived type if the name and parameter list are the same. This is called hide-by-signature semantics.

Remarks

A class is considered a hide-by-signature class when all its functions are marked in the metadata as **hidebysig**. By default, all classes created under **/clr** have **hidebysig** functions. However, a class compiled with **/clr:oldSyntax** does not have **hidebysig** functions, they are hide by name functions. When a class has **hidebysig** functions, the compiler does not hide functions by name in any direct base classes. However, once the compiler encounters a hide-by-name class in an inheritance chain, it resumes hide by name behavior.

Using hide-by-signature semantics, when a function is called on an object, the compiler identifies the most derived class containing a function that could satisfy the function call. If there is only one function in the class that could satisfy the call, the compiler calls that function. If there is more than one function in the class that could satisfy the call, the compiler uses overload resolution rules to determine which function to call. For more information on overload rules, see Function Overloading.

A function in a base class may even have a signature that makes it a slightly better match than a function in a derived class, for a given function call. However, if the function was explicitly called on an object of the derived class, the function in the derived class will be called.

Because the return value is not considered part of a function's signature, a base class function will be hidden if it has the same name and takes the same number and type of arguments as a derived class function, but only differs in the type of the return value.

Example

The following sample shows that a function in a base class is not hidden by a function in a derived class.

```
// hide_by_signature_1.cpp
// compile with: /clr
using namespace System;
ref struct Base {
   void Test() {
      Console::WriteLine("Base::Test");
   }
};

ref struct Derived : public Base {
   void Test(int i) {
      Console::WriteLine("Derived::Test");
   }
};

int main() {
   Derived ^ t = gcnew Derived;
   // Test() in the base class will not be hidden
   t->Test();
}
```

Output

```
Base::Test
```

The following sample shows that the Visual C++ compiler will call a function in the most derived class, even if a conversion is required to match one or more of the parameters, and not call a function in a base class that is a better match for the function call.

```
// hide_by_signature_2.cpp
// compile with: /clr
using namespace System;
ref struct Base {
```

```
   void Test2(Single d) {
      Console::WriteLine("Base::Test2");
   }
};

ref struct Derived : public Base {
   void Test2(Double f) {
      Console::WriteLine("Derived::Test2");
   }
};

int main() {
   Derived ^ t = gcnew Derived;
   // Base::Test2 is better match, but the compiler
   // will call a function in derived class if possible
   t->Test2(3.14f);
}
```

Output

```
Derived::Test2
```

The following sample shows that it is still possible to hide a function if the base class has the same signature as the derived class.

```
// hide_by_signature_3.cpp
// compile with: /clr
using namespace System;
ref struct Base {
   int Test4() {
      Console::WriteLine("Base::Test4");
      return 9;
   }
};

ref struct Derived : public Base {
   char Test4() {
      Console::WriteLine("Derived::Test4");
      return 'a';
   }
};

int main() {
   Derived ^ t = gcnew Derived;

   // Base::Test4 is hidden
   int i = t->Test4();
   Console::WriteLine(i);
}
```

Output

```
Derived::Test4
97
```

The following sample defines a component compiled with **/clr:oldSyntax**. Classes defined using Managed Extensions for C++ have hide-by-name member functions.

```
// hide_by_signature_4.cpp
// compile with: /clr:oldSyntax /LD
using namespace System;
public __gc struct Base0 {
   void Test() {
      Console::WriteLine("in Base0::Test");
   }
};
```

```
public __gc struct Base1 : public Base0 {
   void Test(int i) {
      Console::WriteLine("in Base1::Test");
   }
};
```

The following sample consumes the component built in the previous sample. Notice how hide-by-signature functionality is not applied to base classes of types compiled with **/clr:oldSyntax**.

```
// hide_by_signature_5.cpp
// compile with: /clr:oldSyntax /LD
// compile with: /clr
using namespace System;
#using "hide_by_signature_4.dll"

ref struct Derived : public Base1 {
   void Test(int i, int j) {
      Console::WriteLine("Derived::Test");
   }
};

int main() {
   Derived ^ t = gcnew Derived;
   t->Test(8, 8);   // OK
   t->Test(8);   // OK
   t->Test();   // C2661
}
```

See Also

Reference

Classes and Structs (Managed)

Destructors and Finalizers in Visual C++

Destructors in a reference type perform deterministic clean up of your resources. Finalizers clean up unmanaged resources and can be called deterministically by the destructor or non-deterministically by the garbage collector. (For information about destructors in standard C++, see Destructors (C++).)

```
class classname {
   ~classname() {}   // destructor
   !classname() {}   // finalizer
};
```

Parameters

classname

The name of the class for which you are creating the destructor or finalizer.

Remarks

The behavior of destructors in a managed Visual C++ class differs from the behavior in previous releases. For more information on this change, see Changes in Destructor Semantics.

The common language runtime's garbage collector deletes unused managed objects and releases their memory when no longer needed. However, a type may use resources that the garbage collector does not know how to release. These resources are known as unmanaged resources (native file handles, for example). You should release all unmanaged resources in the finalizer. Because managed resources are released non-deterministically by the garbage collector, it is not safe to refer to managed resources in a finalizer because it is possible that the garbage collector has already cleaned up that managed resource.

A Visual C++ finalizer is not the same as the Finalize method (common language runtime documentation uses finalizer and the Finalize method synonymously). The Finalize method is called by the garbage collector, which invokes each finalizer in a class inheritance chain. Unlike Visual C++ destructors, calling a derived class finalizer does not cause the compiler to invoke call the finalizer in all base classes.

Because the Visual C++ compiler provides support for deterministic release of resources, Visual C++ programmers should not attempt to implement the Dispose or Finalize methods. However, if you are familiar with implementing these methods, this is how a Visual C++ finalizer and a destructor that calls the finalizer maps to the Dispose pattern.

```
// Visual C++ code
ref class T {
   ~T() { this->!T(); }   // destructor calls finalizer
   !T() {}   // finalizer
};

// equivalent Dispose pattern
void Dispose(bool disposing) {
   if (disposing) {
      ~T();
   } else {
      !T();
   }
}
```

A managed type may also use managed resources that you would prefer to release deterministically, and not leave to the garbage collector to release at some point after the object is no longer needed (nondeterministically). Deterministically releasing resources can significantly improve performance.

The Visual C++ compiler lets you define a destructor to deterministically clean up objects. All resources that you want to deterministically release should be released in the destructor. If a finalizer is present, you should call your type's finalizer from the destructor, to avoid code duplication.

```
// destructors_finalizers_1.cpp
// compile with: /clr /c
```

```
ref struct A {
   // destructor cleans up all resources
   ~A() {
      // clean up code to release managed resource
      // ...
      // to avoid code duplication
      // call finalizer to release unmanaged resources
      this->!A();
   }

   // finalizer cleans up unmanaged resources
   // destructor or garbage collector will
   // clean up managed resources
   !A() {
      // clean up code to release unmanaged resource
      // ...
   }
};
```

If the code consuming your type does not call the destructor, the garbage collector will eventually release all managed resources.

The presence of a destructor does not imply the presence of a finalizer. However, a finalizer implies that you will define a destructor, and call the finalizer from the destructor. This provides for the deterministic release of unmanaged resources.

Calling the destructor will suppress (with SuppressFinalize) finalization of the object. If the destructor is not called, your type's finalizer will eventually be called by the garbage collector.

Deterministically cleaning up your object's resources by calling the destructor can increase performance compared with letting the common language runtime non-deterministically finalize the object.

Code authored in Visual C++ and compiled with **/clr** will run a type's destructor for the following reasons:

- If an object created using stack semantics goes out of scope. For more information, see C++ Stack Semantics for Reference Types.
- If an exception is thrown during the object's construction.
- If the object is a member in an object whose destructor is running.
- If you call the delete Operator (C++) on a handle (^ (Handle to Object on Managed Heap)).
- If you explicitly call the destructor.

If your type is being consumed by a client authored in another language, the destructor will be called when:

- A call to **Dispose**.
- Calling Dispose(void) on the type.
- If the type goes out of scope in a C# **using** statement.

If you create an object of a reference type on the managed heap (not using stack semantics for reference types), use **try**/**finally** syntax (try-finally Statement) to ensure that an exception does not prevent the destructor from running:

```
// clr_destructors.cpp
// compile with: /clr
ref struct A {
   ~A() {}
};

int main() {
   A ^ MyA = gcnew A;
   try {
      // use MyA
   }
   finally {
      delete MyA;
   }
```

```
}
```

If your type has a destructor, the compiler will generate a Dispose method that implements IDisposable. If a type authored in Visual C++ with a destructor is consumed from another language, calling IDisposable::Dispose on that type will cause the type's destructor to be called. When consumed from a Visual C++ client, you cannot directly call Dispose; call the destructor instead using the delete operator.

If your type has a finalizer, the compiler will generate a Finalize(void) method that overrides **Finalize**.

If a type has either a finalizer or destructor. the compiler will generate a Dispose(bool) method, according to the design pattern (Implementing Finalize and Dispose to Clean Up Unmanaged Resources). You cannot explicitly author or call Dispose(bool) in Visual C++.

If a type has a base class that conforms to the design pattern, the destructors for all base classes will be called when the destructor for the derived class is called. (If your type is authored in Visual C++, the compiler will ensure that your types implement this pattern.) That is, the destructor of a reference class will chain to its bases and members as specified by the C++ standard (first the classes' destructor is run, then the destructors for its members in the reverse order they were constructed, and finally the destructors for its base classes in the reverse order they were constructed).

Destructors and finalizers are not allowed inside value types or interfaces.

A finalizer can only be defined or declared in a reference type. Like a constructor and destructor, a finalizer has no return type.

After an object's finalizer runs, finalizers in any base classes will also be called, beginning with the least derived type. Finalizers for data members are not automatically chained to by a class's finalizer.

If a finalizer deletes a native pointer in a managed type you will need to ensure that references to or through the native pointer are not prematurely collected; call the destructor on the managed type instead of using KeepAlive.

You can detect at compile time if a type has a finalizer or destructor. For more information, see Compiler Support for Type Traits.

Example

This sample shows two types, one with unmanaged resources and one with managed resources that are deterministically released.

```
// destructors_finalizers_2.cpp
// compile with: /clr
#include <vcclr.h>
#include <stdio.h>
using namespace System;
using namespace System::IO;

ref class SystemFileWriter {
   FileStream ^ file;
   array<Byte> ^ arr;
   int bufLen;

public:
   SystemFileWriter(String ^ name) : file(File::Open(name, FileMode::Append)),
                                     arr(gcnew array<Byte>(1024)) {}

   void Flush() {
      file->Write(arr, 0, bufLen);
      bufLen = 0;
   }

   ~SystemFileWriter() {
      Flush();
      delete file;
   }
};

ref class CRTFileWriter {
   FILE * file;
   array<Byte> ^ arr;
   int bufLen;
```

```
   static FILE * getFile(String ^ n) {
      pin_ptr<const wchar_t> name = PtrToStringChars(n);
      FILE * ret = 0;
      _wfopen_s(&ret, name, L"ab");
      return ret;
   }

public:
   CRTFileWriter(String ^ name) : file(getFile(name)), arr(gcnew array<Byte>(1024) ) {}

   void Flush() {
      pin_ptr<Byte> buf = &arr[0];
      fwrite(buf, 1, bufLen, file);
      bufLen = 0;
   }

   ~CRTFileWriter() {
      this->!CRTFileWriter();
   }

   !CRTFileWriter() {
      Flush();
      fclose(file);
   }
};

int main() {
   SystemFileWriter w("systest.txt");
   CRTFileWriter ^ w2 = gcnew CRTFileWriter("crttest.txt");
}
```

See Also

Reference

Classes and Structs (Managed)

Type and Member Visibility

This section contains topics that discuss the visibility of CLR types and CLR type members outside an assembly.

Remarks

This section contains the following topics:

- Type Visibility
- Member Visibility

See Also

Reference

Classes and Structs (Managed)

Type Visibility

You can control the visibility of CLR types, such that, if an assembly is referenced, types in the assembly can be visible or not visible outside the assembly.

Remarks

`public` indicates that a type will be visible to any source files that contain a `#using` directive for its containing assembly. `private` indicates that a type will not be visible to any source files that contain a `#using` directive for its containing assembly. However, private types are visible within the same assembly. The default visibility for a class is `private`.

Prior to Visual C++ 2005, native types had public accessibility by default outside the assembly. For more information, see Breaking Changes in the Visual C++ 2005 Compiler. Enable Compiler Warning (level 1) C4692 to help you see where you are using private native types incorrectly. Use the make_public pragma to give a public accessibility to a native type in a source code file you cannot modify.

For more information, see The #using Directive.

Example

The following sample shows how to declare types specifying their accessibility, and then access those types inside the assembly. However, if a component with private types was referenced with **#using**, only public types in the assembly would be visible.

```
// type_visibility.cpp
// compile with: /clr
using namespace System;
// public type, visible inside and outside assembly
public ref struct Public_Class {
   void Test(){Console::WriteLine("in Public_Class");}
};

// private type, visible inside but not outside assembly
private ref struct Private_Class {
   void Test(){Console::WriteLine("in Private_Class");}
};

// default accessibility is private
ref class Private_Class_2 {
public:
   void Test(){Console::WriteLine("in Private_Class_2");}
};

int main() {
   Public_Class ^ a = gcnew Public_Class;
   a->Test();

   Private_Class ^ b = gcnew Private_Class;
   b->Test();

   Private_Class_2 ^ c = gcnew Private_Class_2;
   c->Test();
}
```

Output

```
in Public_Class
in Private_Class
in Private_Class_2
```

Now, let's rewrite the previous sample so that it is built as a DLL.

```
// type_visibility_2.cpp
// compile with: /clr /LD
using namespace System;
```

```
// public type, visible inside and outside assembly
public ref struct Public_Class {
   void Test(){Console::WriteLine("in Public_Class");}
};

// private type, visible inside but not outside assembly
private ref struct Private_Class {
   void Test(){Console::WriteLine("in Private_Class");}
};

// default accessibility is private
ref class Private_Class_2 {
public:
   void Test(){Console::WriteLine("in Private_Class_2");}
};
```

This client consumes the component built in the previous sample showing how to access types outside the assembly.

```
// type_visibility_3.cpp
// compile with: /clr
#using "type_visibility_2.dll"
int main() {
   Public_Class ^ a = gcnew Public_Class;
   a->Test();

   // private types not accessible outside assembly
   // Private_Class ^ b = gcnew Private_Class;
   // Private_Class_2 ^ c = gcnew Private_Class_2;
}
```

Output

```
in Public_Class
```

See Also

Reference

Type and Member Visibility

Member Visibility

The internal (accessible from within the same assembly) and external (accessible from outside the assembly) for a member of a public class can differ.

Remarks

You can specify the accessibility of a type member using pairs of access specifiers selected from `public`, `protected`, and `private`.

The following table summarizes the effect of the various access specifiers

Specifier	Effect
`public`	Member is accessible inside and outside the assembly. See public (C++) for more information.
`private`	Member is accessible neither inside nor outside the assembly. See private (C++) for more information.
`protected`	Member is accessible inside and outside the assembly, but only to derived types. See protected (C++) for more information.
`internal`	Member is public inside the assembly but private outside the assembly. **internal** is a context sensitive key word. For more information, see Context-Sensitive Keywords.
`public protected -or- protected public`	Member is public inside the assembly but protected outside the assembly.
`private protected -or- protected private`	Member is protected inside the assembly but private outside the assembly.

Example

The following sample shows a public type with members declared with the different accessibilities, and accesses those members from inside the assembly.

```
// type_member_visibility.cpp
// compile with: /clr
using namespace System;
// public type, visible inside and outside the assembly
public ref class Public_Class {
public:
   void Public_Function(){System::Console::WriteLine("in Public_Function");}

private:
   void Private_Function(){System::Console::WriteLine("in Private_Function");}

protected:
   void Protected_Function(){System::Console::WriteLine("in Protected_Function");}

internal:
   void Internal_Function(){System::Console::WriteLine("in Internal_Function");}
```

```
protected public:
   void Protected_Public_Function(){System::Console::WriteLine("in Protected_Public_Functio
n");}

public protected:
   void Public_Protected_Function(){System::Console::WriteLine("in Public_Protected_Functio
n");}

private protected:
   void Private_Protected_Function(){System::Console::WriteLine("in Private_Protected_Funct
ion");}

protected private:
   void Protected_Private_Function(){System::Console::WriteLine("in Protected_Private_Funct
ion");}
};

// a derived type, will call protected functions
ref struct MyClass : public Public_Class {
   void Test() {
      Console::WriteLine("=======================");
      Console::WriteLine("in function of derived class");
      Protected_Function();
      Protected_Private_Function();
      Private_Protected_Function();
      Console::WriteLine("exiting function of derived class");
      Console::WriteLine("=======================");
   }
};

int main() {
   Public_Class ^ a = gcnew Public_Class;
   MyClass ^ b = gcnew MyClass;
   a->Public_Function();
   a->Protected_Public_Function();
   a->Public_Protected_Function();

   // accessible inside but not outside the assembly
   a->Internal_Function();

   // call protected functions
   b->Test();

   // not accessible inside or outside the assembly
   // a->Private_Function();
}
```

Output

```
in Public_Function
in Protected_Public_Function
in Public_Protected_Function
in Internal_Function
=======================
in function of derived class
in Protected_Function
in Protected_Private_Function
in Private_Protected_Function
exiting function of derived class
=======================
```

Now let's build the previous sample as a DLL.

```
// type_member_visibility_2.cpp
// compile with: /clr /LD
using namespace System;
// public type, visible inside and outside the assembly
```

```
public ref class Public_Class {
public:
   void Public_Function(){System::Console::WriteLine("in Public_Function");}

private:
   void Private_Function(){System::Console::WriteLine("in Private_Function");}

protected:
   void Protected_Function(){System::Console::WriteLine("in Protected_Function");}

internal:
   void Internal_Function(){System::Console::WriteLine("in Internal_Function");}

protected public:
   void Protected_Public_Function(){System::Console::WriteLine("in Protected_Public_Functio
n");}

public protected:
   void Public_Protected_Function(){System::Console::WriteLine("in Public_Protected_Functio
n");}

private protected:
   void Private_Protected_Function(){System::Console::WriteLine("in Private_Protected_Funct
ion");}

protected private:
   void Protected_Private_Function(){System::Console::WriteLine("in Protected_Private_Funct
ion");}
};

// a derived type, will call protected functions
ref struct MyClass : public Public_Class {
   void Test() {
      Console::WriteLine("=======================");
      Console::WriteLine("in function of derived class");
      Protected_Function();
      Protected_Private_Function();
      Private_Protected_Function();
      Console::WriteLine("exiting function of derived class");
      Console::WriteLine("=======================");
   }
};
```

The following sample consumes the component created in the previous sample, showing how to access the members from outside the assembly.

```
// type_member_visibility_3.cpp
// compile with: /clr
#using "type_member_visibility_2.dll"
using namespace System;
// a derived type, will call protected functions
ref struct MyClass : public Public_Class {
   void Test() {
      Console::WriteLine("=======================");
      Console::WriteLine("in function of derived class");
      Protected_Function();
      Protected_Public_Function();
      Public_Protected_Function();
      Console::WriteLine("exiting function of derived class");
      Console::WriteLine("=======================");
   }
};

int main() {
   Public_Class ^ a = gcnew Public_Class;
   MyClass ^ b = gcnew MyClass;
   a->Public_Function();
```

```
    // call protected functions
    b->Test();

    // can't be called outside the assembly
    // a->Private_Function();
    // a->Internal_Function();
    // a->Protected_Private_Function();
    // a->Private_Protected_Function();
  }
```

Output

```
in Public_Function
=======================
in function of derived class
in Protected_Function
in Protected_Public_Function
in Public_Protected_Function
exiting function of derived class
=======================
```

See Also

Reference

Type and Member Visibility

User-Defined Operators

User-defined operators for managed types are allowed as static or instance members of managed types, or at global scope. However, only static operators will be accessible via metadata to clients written in a language other than Visual C++.

When a static user-defined operator is in a reference type, one of the parameters of the user-defined operator must be either:

- A handle (*type* ^) to an instance of the enclosing type,
- A reference type indirection (*type*^& or type^%) to a handle to an instance of the enclosing type.

When a static user-defined operator is in a value type, one of the parameters of the user-defined operator must be either:

- Of the same type as the enclosing value type.
- A pointer type indirection (*type*^) to the enclosing type.
- A reference type indirection (*type*% or *type*&) to the enclosing type.
- A reference type indirection (*type*^% or *type*^&) to the handle.
- You can define the following operators:

Operator	**Unary/Binary Forms?**
!	Unary
!=	Binary
%	Binary
&	Unary and Binary
&&	Binary
*	Unary and Binary
+	Unary and Binary
++	Unary
,	Binary
-	Unary and Binary
--	Unary
->	Unary
/	Binary
<	Binary
<<	Binary
<=	Binary
=	Binary

==	Binary
>	Binary
>=	Binary
>>	Binary
^	Binary
false	Unary
true	Unary
|	Binary
||	Binary
~	Unary

Example

```
// mcppv2_user-defined_operators.cpp
// compile with: /clr
using namespace System;
public ref struct X {
   X(int i) : m_i(i) {}
   X() {}

   int m_i;

   // static, binary, user-defined operator
   static X ^ operator + (X^ me, int i) {
      return (gcnew X(me -> m_i + i));
   }

   // instance, binary, user-defined operator
   X^ operator -( int i ) {
      return gcnew X(this->m_i - i);
   }

   // instance, unary, user-defined pre increment operator
   X^ operator ++() {
      return gcnew X(this->m_i++);
   }

   // instance, unary, user-defined post increment operator
   X^ operator ++(int i) {
      return gcnew X(this->m_i++);
   }

   // static, unary user-defined pre and post increment operator
   static X^ operator-- (X^ me) {
      return (gcnew X(me -> m_i - 1));
   }
};

int main() {
   X ^hX = gcnew X(-5);
   System::Console::WriteLine(hX -> m_i);

   hX = hX + 1;
   System::Console::WriteLine(hX -> m_i);
```

```
   hX = hX - (-1);
   System::Console::WriteLine(hX -> m_i);

   ++hX;
   System::Console::WriteLine(hX -> m_i);

   hX++;
   System::Console::WriteLine(hX -> m_i);

   hX--;
   System::Console::WriteLine(hX -> m_i);

   --hX;
   System::Console::WriteLine(hX -> m_i);
}
```

Output

```
-5
-4
-3
-2
-1
-2
-3
```

This sample demonstrates operator synthesis, which is is only available when compiling with **/clr**. Operator synthesis creates the assignment form of a binary operator if one is not defined, and where the left-hand side of the assignment operator has a CLR type.

```
// mcppv2_user-defined_operators_2.cpp
// compile with: /clr
ref struct A {
   A(int n) : m_n(n) {};
   static A^ operator + (A^ r1, A^ r2) {
      return gcnew A( r1->m_n + r2->m_n);
   };
   int m_n;
};

int main() {
   A^ a1 = gcnew A(10);
   A^ a2 = gcnew A(20);

   a1 += a2;   // a1 = a1 + a2   += not defined in source
   System::Console::WriteLine(a1->m_n);
}
```

Output

```
30
```

See Also

Reference

Classes and Structs (Managed)

User-Defined Conversions

This section discusses user-defined conversions (UDC) when one of the types in the conversion is a reference or instance of a value type or reference type.

Remarks

A user-defined conversion can either be implicit or explicit. A UDC should be implicit if the conversion does not result in a loss of information. Otherwise an explicit UDC should be defined.

A native class's constructor can be used to convert a reference or value type to a native class.

For more information about conversions, see Implicit Boxing and Standard Conversions.

This section contains the following topics:

- Convert-From Operators
- Convert-To Operators
- How to: Convert Generic Classes
- How to: Convert with Constructors

Example

```
// mcpp_User_Defined_Conversions.cpp
// compile with: /clr
#include "stdio.h"
ref class R;
class N;

value class V {
   static operator V(R^) {
      return V();
   }
};

ref class R {
public:
   static operator N(R^);
   static operator V(R^) {
      System::Console::WriteLine("in R::operator N");
      return V();
   }
};

class N {
public:
   N(R^) {
      printf("in N::N\n");
   }
};

R::operator N(R^) {
   System::Console::WriteLine("in R::operator N");
   return N(nullptr);
}

int main() {
   // Direct initialization:
   R ^r2;
   N n2(r2);   // direct initialization, calls constructor
   static_cast<N>(r2);   // also direct initialization

   R ^r3;
   // ambiguous V::operator V(R^) and R::operator V(R^)
   // static_cast<V>(r3);
```

```
}
```

Output

```
in N::N
in N::N
```

See Also

Reference

Classes and Structs (Managed)

Convert-From Operators

Convert-from operators create an object of the class in which the operator is defined from an object of some other class.

Standard C++ does not support convert-from operators; standard C++ uses constructors for this purpose. However, when using CLR types, Visual C++ provide syntactic support for calling convert-from operators.

To interoperate well with other CLS-compliant languages, you may wish to wrap each user-defined unary constructor for a given class with a corresponding convert-from operator.

Remarks

Convert-from operators:

- Shall be defined as static functions.
- Can either be implicit (for conversions that do not lose precision such as short-to-int) or explicit, when there may be a loss of precision.
- Shall return an object of the containing class.
- Shall have the "from" type as the sole parameter type.

Example

The following sample shows an implicit and explicit "convert-from", user-defined conversion (UDC) operator.

```
// clr_udc_convert_from.cpp
// compile with: /clr
value struct MyDouble {
   double d;

   MyDouble(int i) {
      d = static_cast<double>(i);
      System::Console::WriteLine("in constructor");
   }

   // Wrap the constructor with a convert-from operator.
   // implicit UDC because conversion cannot lose precision
   static operator MyDouble (int i) {
      System::Console::WriteLine("in operator");
      // call the constructor
      MyDouble d(i);
      return d;
   }

   // an explicit user-defined conversion operator
   static explicit operator signed short int (MyDouble) {
      return 1;
   }
};

int main() {
   int i = 10;
   MyDouble md = i;
   System::Console::WriteLine(md.d);

   // using explicit user-defined conversion operator requires a cast
   unsigned short int j = static_cast<unsigned short int>(md);
   System::Console::WriteLine(j);
}
```

Output

```
in operator
in constructor
10
```

See Also

Reference

User-Defined Conversions

Convert-To Operators

Convert-to operators convert an object of the class in which the operator is defined to some other object.

Example

The following sample shows an implicit, convert-to, user-defined conversion operator:

```
// clr_udc_convert_to.cpp
// compile with: /clr
using namespace System;
value struct MyInt {
   Int32 i;

   // convert MyInt to String^
   static operator String^ ( MyInt val ) {
      return val.i.ToString();
   }

   MyInt(int _i) : i(_i) {}
};

int main() {
   MyInt mi(10);
   String ^s = mi;
   Console::WriteLine(s);
}
```

Output

```
10
```

An explicit user-defined convert-to conversion operator is appropriate for conversions that potentially lose data in some way. To invoke an explicit convert-to operator, a cast must be used.

```
// clr_udc_convert_to_2.cpp
// compile with: /clr
value struct MyDouble {
   double d;
   // convert MyDouble to Int32
   static explicit operator System::Int32 ( MyDouble val ) {
      return (int)val.d;
   }
};

int main() {
   MyDouble d;
   d.d = 10.3;
   System::Console::WriteLine(d.d);
   int i = 0;
   i = static_cast<int>(d);
   System::Console::WriteLine(i);
}
```

Output

```
10.3
10
```

See Also

Reference

User-Defined Conversions

How to: Convert Generic Classes

You can convert a generic class to T.

Example

```
// clr_udc_generics.cpp
// compile with: /clr
generic<class T>
public value struct V {
   T mem;
   static operator T(V v) {
      return v.mem;
   }

   void f(T t) {
      mem = t;
   }
};

int main() {
   V<int> v;
   v.f(42);
   int i = v;
   i += v;
   System::Console::WriteLine(i == (42 * 2) );
}
```

Output

```
True
```

See Also

Reference

User-Defined Conversions

How to: Convert with Constructors

A converting constructor takes a type and uses it to create an object. A converting constructor is called with direct initialization only; casts will not invoke converting constructors.

By default, converting constructors are explicit for CLR types.

Example

```
// clr_udc_converting_constructors.cpp
// compile with: /clr
public ref struct R {
   int m;
   char c;

   R(int i) : m(i) { }
   R(char j) : c(j) { }
};

public value struct V {
   R^ ptr;
   int m;

   V(R^ r) : ptr(r) { }
   V(int i) : m(i) { }
};

int main() {
   R^ r = gcnew R(5);

   System::Console::WriteLine( V(5).m);
   System::Console::WriteLine( V(r).ptr);
}
```

Output

```
5
R
```

In this code sample, an implicit static conversion function does the same thing as an explicit conversion constructor.

```
public value struct V {
   int m;
   V(int i) : m(i) {}
   static operator V(int i) {
      V v(i*100);
      return v;
   }
};

public ref struct R {
   int m;
   R(int i) : m(i) {}
   static operator R^(int i) {
      return gcnew R(i*100);
   }
};

int main() {
   V v(13);   // explicit
   R^ r = gcnew R(12);   // explicit

   System::Console::WriteLine(v.m);
   System::Console::WriteLine(r->m);

   // explicit ctor can't be called here: not ambiguous
```

```
    v = 5;
    r = 20;

    System::Console::WriteLine(v.m);
    System::Console::WriteLine(r->m);
}
```

Output

```
13
12
500
2000
```

See Also

Reference

User-Defined Conversions

initonly

initonly indicates that variable assignment can only occur as part of the declaration or in a static constructor in the same class.

initonly is a context-sensitive keyword; see Context-Sensitive Keywords for more information.

Example

```
// mcpp_initonly.cpp
// compile with: /clr /c
ref struct Y1 {
   initonly
   static int staticConst1;

   initonly
   static int staticConst2 = 0;

   static Y1() {
      staticConst1 = 0;
   }
};
```

See Also

Reference

Classes and Structs (Managed)

cli Namespace

The **cli** namespace provides language elements that do not conflict with user-defined symbols in existing source code. The **cli** namespace is necessary because symbols in the namespace can be used in many contexts, such that, the compiler is not able to track them as Context-Sensitive Keywords.

Remarks

When compiling with **/clr**, `using namespace cli;` is implied.

The following are symbols in the **cli** namespace:

- array (Visual C++)
- interior_ptr
- pin_ptr
- safe_cast

It is possible to use a symbol in the **cli** namespace as a user-defined symbol in your code. However, once you have done so, you will have to explicitly or implicitly qualify your references to the **cli** language element of the same name.

```
// cli_namespace.cpp
// compile with: /clr
using namespace cli;
int main() {
   array<int> ^ MyArray = gcnew array<int>(100);
   int array = 0;

   array<int> ^ MyArray2 = gcnew array<int>(100);   // C2062

   // OK
   cli::array<int> ^ MyArray2 = gcnew cli::array<int>(100);
   ::array<int> ^ MyArray3 = gcnew ::array<int>(100);
}
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

Compiler Support for Type Traits

The compiler's support for type traits allows library writers to determine various characteristics of a type at compile time.

All type traits return false if the specified conditions are not met.

These traits are necessary but not sufficient to implement some of the tr1 type traits like **std::tr1::is_pod**. The compiler intrinsics provide type information that is otherwise unavailable.

Remarks

The compiler supports the following type traits:

Type Trait	Description

__has_assign(*type* **)**	Returns true if the CLR or native type has a copy assignment operator.
	`// has_assign.cpp`
	`// compile with: /clr`
	`ref struct R {`
	`  void operator=(R% r) {}`
	`};`
	
	`int main() {`
	`  System::Console::WriteLine(__has_assign(R));`
	`}`

__has_copy(*type* **)**	Returns true if the CLR or native type has a copy constructor.
	`// has_copy.cpp`
	`// compile with: /clr`
	`ref struct R {`
	`  R(R% r) {}`
	`};`
	
	`int main() {`
	`  System::Console::WriteLine(__has_copy(R));`
	`}`

__has_finalizer(*type* **)**	Returns true if the CLR type has a finalizer. See Destructors and Finalizers in Visual C++ for more information.
	`// has_finalizer.cpp`
	`// compile with: /clr`
	`using namespace System;`
	`ref struct R {`
	`  ~R() {}`
	`protected:`
	`  !R() {}`
	`};`
	
	`int main() {`
	`  Console::WriteLine(__has_finalizer(R));`
	`}`

__has_nothrow_assign(*type* **)**	Returns true if a copy assignment operator has an empty exception specification.
	`// has_nothrow_assign.cpp`
	`#include <stdio.h>`
	`struct S {`
	`void operator=(S& r) throw() {}`
	`};`
	
	`int main() {`
	`__has_nothrow_assign(S) == true ?`
	`printf("true\n") : printf("false\n");`
	`}`

__has_nothrow_constructor(*type* **)**	Returns true if the default constructor has an empty exception specification.
	`// has_nothrow_constructor.cpp`
	`#include <stdio.h>`
	`struct S {`
	`S() throw() {}`
	`};`
	
	`int main() {`
	`__has_nothrow_constructor(S) == true ?`
	`printf("true\n") : printf("false\n");`
	`}`

__has_nothrow_copy(*type* **)**	Returns true if the copy constructor has an empty exception specification.
	`// has_nothrow_copy.cpp`
	`#include <stdio.h>`
	`struct S {`
	`S(S& r) throw() {}`
	`};`
	
	`int main() {`
	`__has_nothrow_copy(S) == true ?`
	`printf("true\n") : printf("false\n");`
	`}`

__has_trivial_assign(*type* **)**	Returns true if the type has a trivial, compiler-generated assignment operator.
	`// has_trivial_assign.cpp`
	`#include <stdio.h>`
	`struct S {};`
	
	`int main() {`
	`__has_trivial_assign(S) == true ?`
	`printf("true\n") : printf("false\n");`
	`}`

__has_trivial_constructor(*type* **)**	Returns true if the type has a trivial, compiler-generated constructor.
	`// has_trivial_constructor.cpp`
	`#include <stdio.h>`
	`struct S {};`
	
	`int main() {`
	`  __has_trivial_constructor(S) == true ?`
	`  printf("true\n") : printf("false\n");`
	`}`

__has_trivial_copy(*type* **)**	Returns true if the type has a trivial, compiler-generated copy constructor.
	`// has_trivial_copy.cpp`
	`#include <stdio.h>`
	`struct S {};`
	
	`int main() {`
	`  __has_trivial_copy(S) == true ?`
	`  printf("true\n") : printf("false\n");`
	`}`

__has_trivial_destructor(*type* **)**	Returns true if the type has a trivial, compiler-generated destructor.
	`// has_trivial_destructor.cpp`
	`#include <stdio.h>`
	`struct S {};`
	
	`int main() {`
	`  __has_trivial_destructor(S) == true ?`
	`  printf("true\n") : printf("false\n");`
	`}`

__has_user_destructor(*type* **)**	Returns true if the CLR or native type has a user-declared destructor.
	`// has_user_destructor.cpp`
	`// compile with: /clr`
	`using namespace System;`
	`ref class R {`
	`  ~R() {}`
	`};`
	
	`int main() {`
	`  Console::WriteLine(__has_user_destructor(R));`
	`}`

__has_virtual_destructor(*type* **)**	Returns true if the type has a virtual destructor.
	
	__has_virtual_destructor also works on CLR types, and any user-defined destructor in a CLR type is a virtual destructor.
	
	`// has_virtual_destructor.cpp`
	`#include <stdio.h>`
	`struct S {`
	`  virtual ~S() {}`
	`};`
	
	`int main() {`
	`  __has_virtual_destructor(S) == true ?`
	`  printf("true\n") : printf("false\n");`
	`}`

__is_abstract(*type* **)**	Returns true if the type is an abstract type. For more information on native abstract types, see abstract (Visual C++).
	
	__is_abstract also works for CLR types. An interface with at least one member is an abstract type, as is a reference type with at least one abstract member. For more information on abstract CLR types, see Abstract Classes (C++)
	
	`// is_abstract.cpp`
	`#include <stdio.h>`
	`struct S {`
	`  virtual void Test() = 0;`
	`};`
	
	`int main() {`
	`  __is_abstract(S) == true ?`
	`  printf("true\n") : printf("false\n");`
	`}`

__is_base_of(*base* **,** *derived***)**	Returns true if the first type is a base class of the second type, of if both types are the same.
	
	__is_base_of also works on CLR types. For example, it will return true if the first type is an interface class and the second type implements the interface.
	
	`// is_base_of.cpp`
	`#include <stdio.h>`
	`struct S {};`
	`struct T : public S {};`
	
	`int main() {`
	`  __is_base_of(S, T) == true ?`
	`  printf("true\n") : printf("false\n");`
	
	`  __is_base_of(S, S) == true ?`
	`  printf("true\n") : printf("false\n");`
	`}`

__is_class(*type* **)**	Returns true if the type is a native class or struct.
	`// is_class.cpp`
	`#include <stdio.h>`
	`struct S {};`
	
	`int main() {`
	`  __is_class(S) == true ?`
	`  printf("true\n") : printf("false\n");`
	`}`

__is_convertible_to(*from* **,** *to* **)**	Returns true if the first type can be converted to the second type.
	`// is_convertible_to.cpp`
	`#include <stdio.h>`
	`struct S {};`
	`struct T : public S {};`
	
	`int main() {`
	`  S * s = new S;`
	`  T * t = new T;`
	`  s = t;`
	`  __is_convertible_to(T, S) == true ?`
	`  printf("true\n") : printf("false\n");`
	`}`

__is_delegate(*type* **)**	Returns true if *type* is a delegate. For more information, see delegate.
	`// is_delegate.cpp`
	`// compile with: /clr`
	`delegate void MyDel();`
	`int main() {`
	`  System::Console::WriteLine(__is_delegate(MyDel));`
	`}`

__is_empty(*type* **)**	Returns true if the type has no instance data members.
	`// is_empty.cpp`
	`#include <stdio.h>`
	`struct S {`
	`  int Test() {}`
	`  static int i;`
	`};`
	`int main() {`
	`  __is_empty(S) == true ?`
	`  printf("true\n") : printf("false\n");`
	`}`

__is_enum(*type* **)**	Returns true if the type is a native enum.
	`// is_enum.cpp`
	`#include <stdio.h>`
	`enum E { a, b };`
	
	`struct S {`
	`  enum E2 { c, d };`
	`};`
	
	`int main() {`
	`  __is_enum(E) == true ?`
	`  printf("true\n") : printf("false\n");`
	
	`  __is_enum(S::E2) == true ?`
	`  printf("true\n") : printf("false\n");`
	`}`

__is_interface_class(*type* **)**	Returns true if passed a CLR interface. For more information, see interface class.
	`// is_interface_class.cpp`
	`// compile with: /clr`
	`using namespace System;`
	`interface class I {};`
	`int main() {`
	`  Console::WriteLine(__is_interface_class(I));`
	`}`

__is_pod(*type* **)**	Returns true if the type is a class or union with no constructor or private or protected non-static members, no base classes, and no virtual functions. See the C++ standard, sections 8.5.1/1, 9/4, and 3.9/10 for more information on PODs.
	__is_pod will return false on fundamental types.
	`// is_pod.cpp`
	`#include <stdio.h>`
	`struct S {};`
	
	`int main() {`
	`  __is_pod(S) == true ?`
	`  printf("true\n") : printf("false\n");`
	`}`

__is_polymorphic(*type* **)**	Returns true if a native type has virtual functions.
	`// is_polymorphic.cpp`
	`#include <stdio.h>`
	`struct S {`
	`  virtual void Test(){}`
	`};`
	
	`int main() {`
	`  __is_polymorphic(S) == true ?`
	`  printf("true\n") : printf("false\n");`
	`}`

__is_ref_array(*type* **)**	Returns true if passed a CLR array. For more information, see array (Visual C++).
	`// is_ref_array.cpp`
	`// compile with: /clr`
	`using namespace System;`
	`int main() {`
	`  array<int>^ x = gcnew array<int>(10);`
	`  Console::WriteLine(__is_ref_array(array<int>));`
	`}`

__is_ref_class(*type* **)**	Returns true if passed a reference class. For more information on user-defined reference types, see Classes and Structs (Managed).
	`// is_ref_class.cpp`
	`// compile with: /clr`
	`using namespace System;`
	`ref class R {};`
	`int main() {`
	`  Console::WriteLine(__is_ref_class(Buffer));`
	`  Console::WriteLine(__is_ref_class(R));`
	`}`

__is_sealed(*type* **)**	Returns true if passed a CLR or native type marked sealed. For more information, see sealed.
	`// is_sealed.cpp`
	`// compile with: /clr`
	`ref class R sealed{};`
	`int main() {`
	`  System::Console::WriteLine(__is_sealed(R));`
	`}`

__is_simple_value_class(*type* **)**	Returns true if passed a value type that contains no references to the garbage-collected heap. For more information on user-defined value types, see Classes and Structs (Managed).
	`// is_simple_value_class.cpp`
	`// compile with: /clr`
	`using namespace System;`
	`ref class R {};`
	`value struct V {};`
	`value struct V2 {`
	`  R ^ r;  // not a simnple value type`
	`};`
	
	`int main() {`
	`  Console::WriteLine(__is_simple_value_class(V));`
	`  Console::WriteLine(__is_simple_value_class(V2));`
	`}`

__is_union(*type* **)**	Returns true if a type is a union.
	`// is_union.cpp`
	`#include <stdio.h>`
	`union A {`
	`  int i;`
	`  float f;`
	`};`
	
	`int main() {`
	`  __is_union(A) == true ?`
	`  printf("true\n") : printf("false\n");`
	`}`

__is_value_class(*type* **)**	Returns true if passed a value type. For more information on user-defined value types, see Classes and Structs (Managed).
	`// is_value_class.cpp`
	`// compile with: /clr`
	`value struct V {};`
	
	`int main() {`
	`  System::Console::WriteLine(__is_value_class(V));`
	`}`

Example

This sample shows how to use a class template to expose a compiler type trait for a **/clr** compilation. For more information, see Managed Templates,

```
// compiler_type_traits.cpp
// compile with: /clr
using namespace System;

template <class T>
ref struct is_class {
   literal bool value = __is_ref_class(T);
};

ref class R {};

int main () {
   if (is_class<R>::value)
      Console::WriteLine("R is a ref class");
   else
      Console::WriteLine("R is not a ref class");
}
```

Output

```
R is a ref class
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

Context-Sensitive Keywords

Some language elements are only recognized in specific contexts. These are known as context-sensitive keywords. Outside the specific context, a context-sensitive keyword can be a user-defined symbol.

Remarks

For readability purposes, you may want to limit your use of context-sensitive keywords as user–defined symbols.

The following are context-sensitive keywords:

- abstract
- delegate
- event
- finally
- for each, in
- initonly
- **internal** (see Member Visibility)
- literal
- override
- property
- sealed
- **where** (part of Generics (Visual C++))

Example

The following sample shows that property, a context-sensitive keyword, can be used to define a property and a variable.

```
// context_sensitive_keywords.cpp
// compile with: /clr
public ref class C {
   int MyInt;
public:
   C() : MyInt(99) {}

   property int Property_Block {
      int get() { return MyInt; }
   }
};

int main() {
   int property = 0;
   C ^ MyC = gcnew C();
   property = MyC->Property_Block;
   System::Console::WriteLine(++property);
}
```

Output

```
100
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

delegate

Defines a reference type that can encapsulate one or more methods with a specific function prototype. Delegates provide the underlying mechanism (acting as a kind of pointer to member function) for events in the common language runtime component model.

```
access delegate function_declaration
```

Parameters

access (optional)

The accessibility of the delegate outside of the assembly can be public or private. The default is private. Inside a class, a delegate can have any accessibility.

function_declaration

The signature of the function that can be bound to the delegate. The return type of a delegate can be any managed type. For interoperability reasons, it is recommended that the return type of a delegate be a CLS type.

To define an unbound delegate, the first parameter in *function_declaration* should be the type of the *this* pointer for the object (for more information, see Unbound Delegates).

Remarks

Delegates are multicast: the "function pointer" can be bound to one or more methods within a managed class. The **delegate** keyword defines a multicast delegate type with a specific method signature.

A delegate can also be bound to a method of a value class, such as a static method.

A delegate has the following characteristics:

- It inherits from **System::MulticastDelegate**.
- It has a constructor that takes two arguments: a pointer to a managed class or **NULL** (in the case of binding to a static method) and a fully qualified method of the specified type.
- It has a method called **Invoke**, whose signature matches the declared signature of the delegate.

When a delegate is invoked, its function(s) are called in the order they were attached.

The return value of a delegate is the return value from its last attached member function.

Delegates cannot be overloaded.

delegate is a context-sensitive keyword; see Context-Sensitive Keywords for more information.

Delegates can be bound or unbound.

When you instantiate a bound delegate, the first argument shall be an object reference. The second argument of a delegate instantiation shall either be the address of a method of a managed class object, or a pointer to a method of a value type. The second argument of a delegate instantiation must name the method with the full class scope syntax and apply the address-of operator.

When you instantiate an unbound delegate, the first argument shall either be the address of a method of a managed class object, or a pointer to a method of a value type. The argument must name the method with the full class scope syntax and apply the address-of operator.

When creating a delegate to a static or global function, only one parameter is required: the function (optionally, the address of the function).

You can detect at compile time if a type is a delegate with **__is_delegate (***type***)**. For more information, see Compiler Support for Type Traits.

For more information on delegates, see

- Unbound Delegates

- How to: Define and Use Delegates
- Delegate to a Member of a Value Class
- Delegate to an Unmanaged Function
- Composed Delegates
- How to: Pass a Delegate^ to a Native Function Expecting a Function Pointer
- Generic Delegates (Visual C++)

Example

The following example shows how to declare, initialize, and invoke delegates.

```
// mcppv2_delegate.cpp
// compile with: /clr
using namespace System;

// declare a delegate
public delegate void MyDel(int i);

ref class A {
public:
   void func1(int i) {
      Console::WriteLine("in func1 {0}", i);
   }

   void func2(int i) {
      Console::WriteLine("in func2 {0}", i);
   }

   static void func3(int i) {
      Console::WriteLine("in static func3 {0}", i);
   }
};

int main () {
   A ^ a = gcnew A;

   // declare a delegate instance
   MyDel^ DelInst;

   // test if delegate is initialized
   if (DelInst)
      DelInst(7);

   // assigning to delegate
   DelInst = gcnew MyDel(a, &A::func1);

   // invoke delegate
   if (DelInst)
      DelInst(8);

   // add a function
   DelInst += gcnew MyDel(a, &A::func2);

   DelInst(9);

   // remove a function
   DelInst -= gcnew MyDel(a, &A::func1);

   // invoke delegate with Invoke
   DelInst->Invoke(10);

   // make delegate to static function
   MyDel ^ StaticDelInst = gcnew MyDel(&A::func3);
   StaticDelInst(11);
```

```
}
```

Output

```
in func1 8
in func1 9
in func2 9
in func2 10
in static func3 11
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

Unbound Delegates

An unbound delegate allows you to pass an instance of the type whose function you want to call when the delegate is called.

Unbound delegates are especially useful if you want to iterate through the objects in a collection (using for each, in) keywords, and call a member function on each instance.

The following is a summary of how to declare, instantiate, and call bound and unbound delegates:

Action	Bound Delegates	Unbound Delegates
Declare	The delegate signature must match the signature of the function you want to call through the delegate.	The first parameter of the delegate signature is the type of `this` for the object you want to call. After the first parameter, the delegate signature must match the signature of the function you want to call through the delegate.
Instantiate	When you instantiate a bound delegate, you can specify an instance function or a global or static member function. To specify an instance function, the first parameter is an instance of the type whose member function you want to call and the second parameter is the address of the function you want to call. If you want to call a global or static member function, you only pass the name of the static member function or the name of a global function.	When instantiating an unbound delegate, you only pass the address of the function you want to call.
Call	When you call a bound delegate, you only pass the parameters required by the delegate signature.	Same as a bound delegate (but remember the first parameter must be an instance of the object containing the function you want to call).

Example

This sample demonstrates the syntax for declaring, instantiating, and calling unbound delegates.

```
// unbound_delegates.cpp
// compile with: /clr
ref struct A {
   A(){}
   A(int i) : m_i(i) {}
   void Print(int i) { System::Console::WriteLine(m_i + i);}

private:
   int m_i;
};

value struct V {
   void Print() { System::Console::WriteLine(m_i);}
   int m_i;
};

delegate void Delegate1(A^, int i);
delegate void Delegate2(A%, int i);

delegate void Delegate3(interior_ptr<V>);
delegate void Delegate4(V%);

delegate void Delegate5(int i);
delegate void Delegate6();

int main() {
   A^ a1 = gcnew A(1);
```

```
   A% a2 = *gcnew A(2);

   Delegate1 ^ Unbound_Delegate1 = gcnew Delegate1(&A::Print);
   // delegate takes a handle
   Unbound_Delegate1(a1, 1);
   Unbound_Delegate1(%a2, 1);

   Delegate2 ^ Unbound_Delegate2 = gcnew Delegate2(&A::Print);
   // delegate takes a tracking reference (must deference the handle)
   Unbound_Delegate2(*a1, 1);
   Unbound_Delegate2(a2, 1);

   // instantiate a bound delegate to an instance member function
   Delegate5 ^ Bound_Del = gcnew Delegate5(a1, &A::Print);
   Bound_Del(1);

   // instantiate value types
   V v1 = {7};
   V v2 = {8};

   Delegate3 ^ Unbound_Delegate3 = gcnew Delegate3(&V::Print);
   Unbound_Delegate3(&v1);
   Unbound_Delegate3(&v2);

   Delegate4 ^ Unbound_Delegate4 = gcnew Delegate4(&V::Print);
   Unbound_Delegate4(v1);
   Unbound_Delegate4(v2);

   Delegate6 ^ Bound_Delegate3 = gcnew Delegate6(v1, &V::Print);
   Bound_Delegate3();
}
```

Output

```
2
3
2
3
2
7
8
7
8
7
```

Example

This sample shows how you can use unbound delegates and the for each, in keywords to iterate through objects in a collection and call a member function on each instance.

```
// unbound_delegates_2.cpp
// compile with: /clr
using namespace System;

ref class RefClass {
   String^ _Str;

public:
   RefClass( String^ str ) : _Str( str ) {}
   void Print() { Console::Write( _Str ); }
};

delegate void PrintDelegate( RefClass^ );

int main() {
   PrintDelegate^ d = gcnew PrintDelegate( &RefClass::Print );

   array< RefClass^ >^ a = gcnew array<RefClass^>( 10 );
```

```
   for ( int i = 0; i < a->Length; ++i )
      a[i] = gcnew RefClass( i.ToString() );

   for each ( RefClass^ R in a )
      d( R );

   Console::WriteLine();
}
```

Output

```
0123456789
```

Example

This sample creates an unbound delegate to a property's accessor functions.

```
// unbound_delegates_3.cpp
// compile with: /clr
ref struct B {
   property int P1 {
      int get() { return m_i; }
      void set(int i) { m_i = i; }
   }

private:
   int m_i;
};

delegate void DelBSet(B^, int);
delegate int DelBGet(B^);

int main() {
   B^ b = gcnew B;

   DelBSet^ delBSet = gcnew DelBSet(&B::P1::set);
   delBSet(b, 11);

   DelBGet^ delBGet = gcnew DelBGet(&B::P1::get);
   System::Console::WriteLine(delBGet(b));
}
```

Output

```
11
```

Example

This sample shows how to invoke a multicast delegate, where one instance is bound and one instance is unbound.

```
// unbound_delegates_4.cpp
// compile with: /clr
ref class R {
public:
   R(int i) : m_i(i) {}

   void f(R ^ r) {
      System::Console::WriteLine("in f(R ^ r)");
   }

   void f() {
      System::Console::WriteLine("in f()");
   }

private:
   int m_i;
```

```
};

delegate void Del(R ^);

int main() {
   R ^r1 = gcnew R(11);
   R ^r2 = gcnew R(12);

   Del^ d = gcnew Del(r1, &R::f);
   d += gcnew Del(&R::f);
   d(r2);
};
```

Output

```
in f(R ^ r)
in f()
```

Example

This sample shows how to create and call an unbound generic delegate.

```
// unbound_delegates_5.cpp
// compile with: /clr
ref struct R {
   R(int i) : m_i(i) {}

   int f(R ^) { return 999; }
   int f() { return m_i + 5; }

   int m_i;
};

value struct V {
   int f(V%) { return 999; }
   int f() { return m_i + 5; }

   int m_i;
};

generic <typename T>
delegate int Del(T t);

generic <typename T>
delegate int DelV(T% t);


int main() {
   R^ hr = gcnew R(7);
   System::Console::WriteLine((gcnew Del<R^>(&R::f))(hr));

   V v;
   v.m_i = 9;
   System::Console::WriteLine((gcnew DelV<V >(&V::f))(v) );
}
```

Output

```
12
14
```

See Also

Reference

delegate

How to: Define and Use Delegates

The following code example demonstrates how to define and use a delegate. The .NET Framework provides a number of delegates, so often it is not necessary to define new delegates.

The following code example defines a delegate called `MyCallback`. This new delegate requires that the event handling code (the function that is called as a result of firing the delegate) have a return type of **void** and takes a String reference.

In the main function, the `MyCallback` delegate is instantiated using a static method defined by `SomeClass`. The delegate then becomes an alternate method of calling this function, as demonstrated by sending the string "single" to the delegate object.

Next, additional instances of `MyCallback` are linked together and then executed with a single call to the delegate object.

Example

```
// use_delegate.cpp
// compile with: /clr
using namespace System;

ref class SomeClass
{
public:
   static void Func(String^ str)
   {
      Console::WriteLine("static SomeClass::Func - {0}", str);
   }
};

ref class OtherClass
{
public:
   OtherClass( Int32 n )
   {
      num = n;
   }

   void Method(String^ str)
   {
      Console::WriteLine("OtherClass::Method - {0}, num = {1}",
         str, num);
   }

   Int32 num;
};

delegate void MyCallback(String^ str);

int main( )
{
   MyCallback^ callback = gcnew MyCallback(SomeClass::Func);
   callback("single");

   callback += gcnew MyCallback(SomeClass::Func);

   OtherClass^ f = gcnew OtherClass(99);
   callback += gcnew MyCallback(f, &OtherClass::Method);

   f = gcnew OtherClass(100);
   callback += gcnew MyCallback(f, &OtherClass::Method);

   callback("chained");

   return 0;
}
```

Output

```
static SomeClass::Func - single
static SomeClass::Func - chained
static SomeClass::Func - chained
OtherClass::Method - chained, num = 99
OtherClass::Method - chained, num = 100
```

See Also

Reference

delegate

How to: Associate Delegates to Members of a Value Class

This code sample shows how to associate a delegate with a member of a value class.

Example

```
// mcppv2_del_mem_value_class.cpp
// compile with: /clr
using namespace System;
public delegate void MyDel();

value class A {
public:
   void func1() {
      Console::WriteLine("test");
   }
};

int main() {
   A a;
   A^ ah = a;
   MyDel^ f = gcnew MyDel(a, &A::func1);   // implicit box of a
   f();
   MyDel^ f2 = gcnew MyDel(ah, &A::func1);
   f2();
}
```

Output

```
test
test
```

See Also

Reference

delegate

How to: Associate Delegates to Unmanaged Functions

To associate a delegate with a native function, you must wrap the native function in a managed type, and declare the function to be invoked through PInvoke.

Example

```
// mcppv2_del_to_umnangd_func.cpp
// compile with: /clr
#pragma unmanaged
extern "C" void printf(const char*, ...);
class A {
public:
   static void func(char* s) {
      printf(s);
   }
};

#pragma managed
public delegate void func(char*);

ref class B {
   A* ap;

public:
   B(A* ap):ap(ap) {}
   void func(char* s) {
      ap->func(s);
   }
};

int main() {
   A* a = new A;
   B^ b = gcnew B(a);
   func^ f = gcnew func(b, &B::func);
   f("hello");
   delete a;
}
```

Output

```
hello
```

See Also

Reference

delegate

How to: Compose Delegates

This topic demonstrates how to compose delegates. A useful property of delegate objects is that they can be composed using the "**+**" operator. A composed delegate calls the two delegates it was composed from. Only delegates of the same type can be composed.

The "**-**" operator can be used to remove a component delegate from a composed delegate.

Example

```
// mcppv2_compose_delegates.cpp
// compile with: /clr
using namespace System;

delegate void MyDelegate(String ^ s);

ref class MyClass {
public:
   static void Hello(String ^ s) {
      Console::WriteLine("Hello, {0}!", s);
   }

   static void Goodbye(String ^ s) {
      Console::WriteLine("  Goodbye, {0}!", s);
   }
};

int main() {

   MyDelegate ^ a = gcnew MyDelegate(MyClass::Hello);
   MyDelegate ^ b = gcnew MyDelegate(MyClass::Goodbye);
   MyDelegate ^ c = a + b;
   MyDelegate ^ d = c - a;

   Console::WriteLine("Invoking delegate a:");
   a("A");
   Console::WriteLine("Invoking delegate b:");
   b("B");
   Console::WriteLine("Invoking delegate c:");
   c("C");
   Console::WriteLine("Invoking delegate d:");
   d("D");
}
```

Output

```
Invoking delegate a:
Hello, A!
Invoking delegate b:
  Goodbye, B!
Invoking delegate c:
Hello, C!
  Goodbye, C!
Invoking delegate d:
  Goodbye, D!
```

See Also

Reference

delegate

How to: Pass a Delegate^ to a Native Function Expecting a Function Pointer

From a managed component you can call a native function with function pointer parameters where the native function then can call the member function of the managed component's delegate.

Example

This sample created the .dll that exports the native function.

```
// delegate_to_native_function.cpp
// compile with: /LD
#include < windows.h >
extern "C" {
   __declspec(dllexport)
   void nativeFunction(void (CALLBACK *mgdFunc)(const char* str)) {
      mgdFunc("Call to Managed Function");
   }
}
```

This sample consumes the .dll and passes a delegate handle to the native function expecting a function pointer.

```
// delegate_to_native_function_2.cpp
// compile with: /clr
using namespace System;
using namespace System::Runtime::InteropServices;

delegate void Del(String ^s);
public ref class A {
public:
   void delMember(String ^s) {
      Console::WriteLine(s);
   }
};

[DllImportAttribute("delegate_to_native_function", CharSet=CharSet::Ansi)]
extern "C" void nativeFunction(Del ^d);

int main() {
   A ^a = gcnew A;
   Del ^d = gcnew Del(a, &A::delMember);
   nativeFunction(d);   // Call to native function
}
```

Output

```
Call to Managed Function
```

See Also

Reference

delegate

enum class

Declares a managed enumeration. An enumeration is a user-defined type consisting of a set of named constants called enumerators.

```
access enum class name [: type] { enumerator-list } var;
access enum struct name [:type] { enumerator-list } var;
```

Parameters

access

The accessibility of the enum. Can be either **public** or **private**.

enumerator-list

A comma-separated list of the identifiers (enumerators) in the enumeration.

name

The name of the enumeration. Anonymous managed enumerations are not allowed.

type (optional)

The underlying type of the *identifiers*. This can be any scalar type, such as signed or unsigned versions of int, short, or long. **bool** or **char** is also allowed.

var (optional)

The name of a variable of the enumeration type.

Remarks

enum class and **enum struct** are equivalent declarations.

There are two types of enums: managed and standard.

A managed enum might be defined as follows,

```
enum class day {sun, mon };
```

and is semantically equivalent to:

```
ref class day {
public:
   static const int sun = 0;
   static const int mon = 1;
};
```

A standard enum might be defined as follows:

```
enum day2 {sun, mon, };
```

and is semantically equivalent to:

```
static const int sun = 0;
static const int mon = 1;
```

Managed enumerator names (*identifiers*) are not injected into the scope where the enumeration is defined; all references to the enumerators must be fully qualified (*name*::*identifier*). For this reason, you cannot define an anonymous managed enum.

The enumerators of a standard enum are strongly injected into the enclosing scope. That is, if there is another symbol with the same name as an enumerator in the enclosing scope, the compiler will generate an error.

In Visual C++ 2002 and Visual C++ 2003, enumerators were weakly injected (visible in the enclosing scope unless there was another identifier with the same name).

If a standard C++ enum is defined (without **class** or **struct**), compiling with /clr will cause the enumeration to be compiled as a managed enum. The enumeration still has the semantics of an unmanaged enumeration. Note, the compiler injects an attribute, Microsoft::VisualC::NativeEnumAttribute, which the Visual C++ compiler recognizes, to identify a programmer's intent for the enum to be a native enum. Other compilers will simply see the standard enum as a managed enum.

A named, standard enum compiled with /clr will be visible in the assembly as a managed enum, and can be consumed by any other managed compiler. However, an unnamed standard enum will not be publicly visible from the assembly.

In Visual C++ 2002 and Visual C++ 2003, a standard enum used as the type in a function parameter:

```
// mcppv2_enum.cpp
// compile with: /clr
enum E { a, b };
void f(E) {System::Console::WriteLine("hi");}

int main() {
   E myi = b;
   f(myi);
}
```

would emit the following in MSIL for the function signature:

```
void f(int32);
```

However, in current versions of the compiler, the standard enum is emitted as a managed enum with a [NativeEnumAttribute] and the following in MSIL for the function signature:

```
void f(E)
```

For more information about native enums, see C++ Enumeration Declarations.

In the development environment, you can get F1 help on these keywords by highlighting the keyword, (**enum class**, for example) and pressing F1.

For more information on CLR enums, see:

- Underlying Type of an Enum
- Managed and Standard Enumerations
- Operators and Enumerations

Example

```
// mcppv2_enum_2.cpp
// compile with: /clr
// managed enum
public enum class m { a, b };

// standard enum
public enum n { c, d };

// unnamed, standard enum
public enum { e, f } o;

int main() {
   // consume managed enum
   m mym = m::b;
   System::Console::WriteLine("no automatic conversion to int: {0}", mym);
   System::Console::WriteLine("convert to int: {0}", (int)mym);

   // consume standard enum
   n myn = d;
```

```
    System::Console::WriteLine(myn);

    // consume standard, unnamed enum
    o = f;
    System::Console::WriteLine(o);
  }
```

Output

```
no automatic conversion to int: b
convert to int: 1
1
1
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

How to: Specify Underlying Types of Enums

By default, the underlying type of an enumeration is int. However, you can specify the type to be signed or unsigned forms of **int**, **short**, **long**, **__int32**, or **__int64**. You can also use **char** or **bool**.

Example

```
// mcppv2_enum_3.cpp
// compile with: /clr
typedef bool MYBOOL;

public enum class day_bool : MYBOOL {sun = true, mon = false, tue = true, wed = false, thu
= true, fri = false, sat = true};
public enum class day_char : char {sun, mon, tue, wed, thu, fri, sat};

int main() {
   // fully qualified names, enumerator not injected into scope
   day_bool a = day_bool::sun, b = day_bool::mon;

   System::Console::WriteLine(a);
   bool c = (bool)a;
   System::Console::WriteLine(c);
   c = (bool)b;
   System::Console::WriteLine(c);

   day_char d = day_char::sun, e = day_char::mon;
   System::Console::WriteLine(d);
   char f = (char)d;
   System::Console::WriteLine(f);
   f = (char)e;
   System::Console::WriteLine(f);
   e = day_char::tue;
   f = (char)e;
   System::Console::WriteLine(f);
}
```

Output

```
True
True
False
sun
0
1
2
```

See Also

Reference

enum class

How to: Convert Between Managed and Standard Enumerations

There is no standard conversion between an enum and an integral type; a cast is required.

Example

```
// mcppv2_enum_4.cpp
// compile with: /clr
enum class day {sun, mon, tue, wed, thu, fri, sat};
enum {sun, mon, tue, wed, thu, fri, sat} day2; // unnamed std enum

int main() {
   day a = day::sun;
   day2 = sun;
   if ((int)a == day2)
   // or...
   // if (a == (day)day2)
      System::Console::WriteLine("a and day2 are the same");
   else
      System::Console::WriteLine("a and day2 are not the same");
}
```

Output

```
a and day2 are the same
```

See Also

Reference

enum class

Operators and Enumerations

This topic discusses the operators that are valid on a managed enum.

Remarks

The following operators are valid on enums:

Operator
== != < > <= >=
+ -
^ & ~
++ --
sizeof

Operators ^ & ~ ++ -- are defined only for enumerations with integral underlying types, not including bool. Both operands must be of the enumeration type.

The compiler does no static or dynamic checking of the result of an enum operation; an operation may result in a value not in the range of the enum's valid enumerators.

Example

```
// mcppv2_enum_5.cpp
// compile with: /clr
enum class E { a, b } e, mask;
int main() {
   if ( e & mask )   // C2451 no E->bool conversion
      ;

   if ( ( e & mask ) != 0 )   // C3063 no operator!= (E, int)
      ;

   if ( ( e & mask ) != E() )   // OK
      ;
}
```

```
// mcppv2_enum_6.cpp
// compile with: /clr
enum class day : int {sun, mon};
enum : bool {sun = true, mon = false} day2;

int main() {
   day a = day::sun, b = day::mon;
   day2 = sun;

   System::Console::WriteLine(sizeof(a));
   System::Console::WriteLine(sizeof(day2));
   a++;
   System::Console::WriteLine(a == b);
}
```

Output

```
4
1
True
```

See Also

Reference

enum class

event (Visual C++)

The **event** keyword lets you declare an event. An event is a way for a class to provide notifications when something of interest happens.

```
modifier event delegate ^ event_name;   // event data member
modifier event delegate ^ event_name
{
      modifier return_value add (delegate ^ name);
      modifier return_value remove(delegate ^ name);
      modifier return_value raise(parameters);
}   // event block
```

Parameters

delegate

The delegate, whose signature the event handler must match.

event_name

The name of the event.

modifier

A modifier that can be used on either the event declaration or on an event accessor method. Possible values are static and virtual.

name

The local name of the delegate parameter.

parameters(optional)

Parameter(s) for the raise method, must match the signature of the delegate.

return_value

The return value of the event accessor method. To be verifiable, the return type must be **void**.

Remarks

An event is an association between a delegate and a member function (event handler) that responds to the triggering of the event and allows clients from any class to register methods that comply with the signature and return type of the underlying delegate.

The delegate can have one or more associated methods that will be called when your code indicates that the event has occurred. An event in one program can be made available to other programs that target the .NET Framework common language runtime. See Raising an Event Defined in a Different Assembly for a sample.

There are two kinds of events declarations:

Event data members

Storage for the event, in the form of a member of the delegate type, is created by the compiler for data member events. An event data member must be declared inside a class. This is also known as a trivial event (see code sample below.)

Event Blocks

Event blocks let you customize the behavior of the add, remove, and raise methods, by implementing add, remove, and raise methods. The signature of the add, remove, and raise methods must match the signature of the delegate. Event block events are not data members and any use as a data member will generate a compiler error. See Defining Event Accessor Methods for a sample.

The return type of the event handler must match the return type of the delegate.

In the .NET Framework, you can treat a data member as if it were a method itself (that is, the **Invoke** method of its corresponding delegate). You must predefine the delegate type for declaring a managed event data member. In contrast, a managed event method implicitly defines the corresponding managed delegate if it is not already defined. See the code

sample at the end of this topic for an example.

When declaring a managed event, you can specify add and remove accessors that will be called when event handlers are added or removed using operators += and -=. The add, remove and raise methods can be called explicitly.

The following steps must be taken in order to create and use events in Visual C++:

1. Create or identify a delegate. If you are defining your own event, you must also ensure that there is a delegate to use with the **event** keyword. If the event is predefined, in the .NET Framework for example, then consumers of the event need only know the name of the delegate.
2. Create a class that contains:
   - An event created from the delegate.
   - (optional) A method that verifies that an instance of the delegate declared with the **event** keyword exists. Otherwise, this logic must be placed in the code that fires the event.
   - Methods that call the event. These methods can be overrides of some base class functionality.

   This class defines the event.
3. Define one or more classes that connect methods to the event. Each of these classes will associate one or more methods with the event in the base class.
4. Use the event:
   - Create an object of the class that contains the event declaration.
   - Create an object of the class that contains the event definition.

event is a context-sensitive keyword; see Context-Sensitive Keywords for more information.

For more information on events, see

- Events in an Interface
- Overriding the Default Access of add, remove, and raise Methods
- Multiple Handlers to an Event
- Managed Virtual Events
- Event Accessor Methods
- Static Events
- Raising an Event Defined in a Different Assembly
- Abstract Events

Example

```
// mcppv2_events.cpp
// compile with: /clr
using namespace System;

// declare delegates
delegate void ClickEventHandler(int, double);
delegate void DblClickEventHandler(String^);

// class that defines events
ref class EventSource {
public:
   event ClickEventHandler^ OnClick;   // declare the event OnClick
   event DblClickEventHandler^ OnDblClick;   // declare OnDblClick

   void FireEvents() {
      // raises events
      OnClick(7, 3.14159);
      OnDblClick("Hello");
   }
```

```
};

// class that defines methods that will called when event occurs
ref class EventReceiver {
public:
   void OnMyClick(int i, double d) {
      Console::WriteLine("OnClick: {0}, {1}", i, d);
   }

   void OnMyDblClick(String^ str) {
      Console::WriteLine("OnDblClick: {0}", str);
   }
};

int main() {
   EventSource ^ MyEventSource = gcnew EventSource();
   EventReceiver^ MyEventReceiver = gcnew EventReceiver();

   // hook handler to event
   MyEventSource->OnClick += gcnew ClickEventHandler(MyEventReceiver, &EventReceiver::OnMyC
lick);
   MyEventSource->OnDblClick += gcnew DblClickEventHandler(MyEventReceiver, &EventReceiver:
:OnMyDblClick);

   // invoke events
   MyEventSource->FireEvents();

   // unhook handler to event
   MyEventSource->OnClick -= gcnew ClickEventHandler(MyEventReceiver, &EventReceiver::OnMyC
lick);
   MyEventSource->OnDblClick -= gcnew DblClickEventHandler(MyEventReceiver, &EventReceiver:
:OnMyDblClick);
}
```

Output

```
OnClick: 7, 3.14159
OnDblClick: Hello
```

The compiler uses the following logic to generate the raise method of a trivial event: If the event has one or more subscribers, calling the raise method implicitly or explicitly will call the delegate. If the delegate's return type is not void and if there are zero event subscribers, the raise method will return the default value for the delegate type. If there are no event subscribers, calling the raise method this will just return and no exception will be raised; if the delegate return type is not void, this will return the default value for the delegate type. For example

```
// trivial_events.cpp
// compile with: /clr /c
using namespace System;
public delegate int Del();
public ref struct C {
   int i;
   event Del^ MyEvent;

   void FireEvent() {
      i = MyEvent();
   }
};

ref struct EventReceiver {
   int OnMyClick() { return 0; }
};

int main() {
   C c;
   c.i = 687;

   c.FireEvent();
```

```
    Console::WriteLine(c.i);
    c.i = 688;

    EventReceiver^ MyEventReceiver = gcnew EventReceiver();
    c.MyEvent += gcnew Del(MyEventReceiver, &EventReceiver::OnMyClick);
    Console::WriteLine(c.i);
}
```

Output

```
0
688
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

How to: Access Events in Interfaces

The accessibility of a raise method is never public and is protected by default. A raise method is private if the event is private, and virtual if the event is declared virtual. If an event is declared in a managed interface, only its **add** and **remove** methods can be implemented.

Example

```
// mcppv2_events2.cpp
// compile with: /clr
using namespace System;

delegate void Del(int, float);

// interface with event and a function to invoke event
interface struct I {
public:
   event Del ^ E;
   void fire(int, float);
};

// class that implements interface event and function
ref class EventSource: public I {
public:
   virtual event Del^ E;
   virtual void fire(int i, float f) {
      E(i, f);
   }
};

// class that defines event handler
ref class EventReceiver {
public:
   void Handler(int i , float f) {
      Console::WriteLine("EventReceiver::Handler");
   }
};

int main () {
   I^ es = gcnew EventSource();
   EventReceiver^ er = gcnew EventReceiver();

   // hook handler to event
   es->E += gcnew Del(er, &EventReceiver::Handler);

   // call event
   es -> fire(1, 3.14);

   // unhook handler from event
   es->E -= gcnew Del(er, &EventReceiver::Handler);
}
```

Output

```
EventReceiver::Handler
```

See Also

Reference

event (Visual C++)

How to: Override Default Access of add, remove, and raise Methods

This sample shows how to override the default access on the add, remove, and raise events methods.

Example

```
// mcppv2_events3.cpp
// compile with: /clr
public delegate void f(int);

public ref struct E {
   f ^ _E;
public:
   void handler(int i) {
      System::Console::WriteLine(i);
   }

   E() {
      _E = nullptr;
   }

   event f^ Event {
      void add(f ^ d) {
         _E += d;
      }
   private:
      void remove(f ^ d) {
        _E -= d;
      }

   protected:
      void raise(int i) {
         if (_E) {
            _E->Invoke(i);
         }
      }
   }

   // a member function to access all evednt methods
   static void Go() {
      E^ pE = gcnew E;
      pE->Event += gcnew f(pE, &E::handler);
      pE->Event(17);   // prints 17
      pE->Event -= gcnew f(pE, &E::handler);
      pE->Event(17);   // no output
   }
};

int main() {
   E::Go();
}
```

Output

```
17
```

See Also

Reference

event (Visual C++)

How to: Add Multiple Handlers to Events

An event receiver, or any other client code can add one or more handlers to an event.

Example

```
// mcppv2_events4.cpp
// compile with: /clr
using namespace System;
#include <stdio.h>

delegate void ClickEventHandler(int, double);
delegate void DblClickEventHandler(String^);

ref class EventSource {
public:
   event ClickEventHandler^ OnClick;
   event DblClickEventHandler^ OnDblClick;

   void FireEvents() {
      OnClick(7, 3.14159);
      OnDblClick("Started");
   }
};

ref struct EventReceiver {
public:
   void Handler1(int x, double y) {
      System::Console::Write("Click(x={0},y={1})\n", x, y);
   };

   void Handler2(String^ s) {
      System::Console::Write("DblClick(s={0})\n", s->ToCharArray());
   }

   void Handler3(String^ s) {
      System::Console::WriteLine("DblClickAgain(s={0})\n", s->ToCharArray());
   }

   void AddHandlers(EventSource^ pES) {
      pES->OnClick +=
         gcnew ClickEventHandler(this,&EventReceiver::Handler1);
      pES->OnDblClick +=
         gcnew DblClickEventHandler(this,&EventReceiver::Handler2);
      pES->OnDblClick +=
         gcnew DblClickEventHandler(this, &EventReceiver::Handler3);
   }

   void RemoveHandlers(EventSource^ pES) {
      pES->OnClick -=
         gcnew ClickEventHandler(this, &EventReceiver::Handler1);
      pES->OnDblClick -=
         gcnew DblClickEventHandler(this, &EventReceiver::Handler2);
      pES->OnDblClick -=
         gcnew DblClickEventHandler(this, &EventReceiver::Handler3);
   }
};

int main() {
   EventSource^ pES = gcnew EventSource;
   EventReceiver^ pER = gcnew EventReceiver;

   // add handlers
   pER->AddHandlers(pES);

   pES->FireEvents();
```

```
    // remove handlers
    pER->RemoveHandlers(pES);
}
```

Output

```
Click(x=7,y=3.14159)
DblClick(s=System.Char[])
DblClickAgain(s=System.Char[])
```

See Also

Reference

event (Visual C++)

How to: Implement Managed Virtual Events

This sample implements virtual, managed events in an interface and class.

Example

```
// mcppv2_events5.cpp
// compile with: /clr
using namespace System;

public delegate void MyDel();
public delegate int MyDel2(int, float);

// managed class with virtual event
ref class IEFace {
public:
   virtual event MyDel ^ E;   // declares three accessors (add, remove, and raise)
};

// managed interface with virtual event
public interface struct IEFace2 {
public:
   event MyDel2 ^ E2;   // declares two accessors (add and remove)
};

// implement virtual events
ref class EventSource : public IEFace, public IEFace2 {
public:
   virtual event MyDel2 ^ E2;

   void Fire_E() {
      E();
   }

   int Fire_E2(int i, float f) {
      try {
         return E2(i, f);
      }
      catch(System::NullReferenceException^) {
         return 0;   // no handlers
      }
   }
};

// class to hold event handlers, the event receiver
public ref struct EventReceiver {
   // first handler
   void H1() {
      Console::WriteLine("In handler H1");
   }

   // second handler
   int H2(int i, float f) {
      Console::WriteLine("In handler H2 with args {0} and {1}", i.ToString(), f.ToString())
;
      return 0;
   }
};

int main() {
   EventSource ^ pE = gcnew EventSource;
   EventReceiver ^ pR = gcnew EventReceiver;

   // add event handlers
   pE->E += gcnew MyDel(pR, &EventReceiver::H1);
   pE->E2 += gcnew MyDel2(pR, &EventReceiver::H2);
```

```
   // raise events
   pE->Fire_E();
   pE->Fire_E2(1, 2.2);

   // remove event handlers
   pE->E -= gcnew MyDel(pR, &EventReceiver::H1);
   pE->E2 -= gcnew MyDel2(pR, &EventReceiver::H2);

   // raise events, but no handlers so, no effect
   pE->Fire_E();
   pE->Fire_E2(1, 2.5);
}
```

Output

```
In handler H1
In handler H2 with args 1 and 2.2
```

Example

Note that you are currently not able to specify that a simple event overrides or hides a base class event. You must define all of the event's accessor functions, and specify the **new** or **override** keyword on each accessor function.

```
// mcppv2_events5_a.cpp
// compile with: /clr /c
delegate void Del();

ref struct A {
   virtual event Del ^E;
   virtual event Del ^E2;
};

ref struct B : A {
   virtual event Del ^E override;   // C3797
   virtual event Del ^E2 new;   // C3797
};

ref struct C : B {
   virtual event Del ^E {   // OK
      void raise() override {}
      void add(Del ^) override {}
      void remove(Del^) override {}
   }

   virtual event Del ^E2 {   // OK
      void raise() new {}
      void add(Del ^) new {}
      void remove(Del^) new {}
   }
};
```

See Also

Reference

event (Visual C++)

How to: Define Event Accessor Methods

The following sample shows how you can define an event's behavior when handlers are added or removed, and for when an event is raised.

Example

```
// mcppv2_events6.cpp
// compile with: /clr
using namespace System;

public delegate void MyDel();
public delegate int MyDel2(int, float);

ref class EventSource {
public:
   MyDel ^ pE;
   MyDel2 ^ pE2;

   event MyDel^ E {
      void add(MyDel^ p) {
         pE = static_cast<MyDel^> (Delegate::Combine(pE, p));
         // cannot refer directly to the event
        // E = static_cast<MyDel^> (Delegate::Combine(pE, p));   // error
      }

      void remove(MyDel^ p) {
         pE = static_cast<MyDel^> (Delegate::Remove(pE, p));
      }

      void raise() {
         if (pE != nullptr)
            pE->Invoke();
      }
   }  // E event block

   event MyDel2^ E2 {
      void add(MyDel2^ p2) {
         pE2 = static_cast<MyDel2^> (Delegate::Combine(pE2, p2));
      }

      void remove(MyDel2^ p2) {
         pE2 = static_cast<MyDel2^> (Delegate::Remove(pE2, p2));
      }

      int raise(int i, float f) {
         if (pE2 != nullptr) {
            return pE2->Invoke(i, f);
         }
         return 1;
      }
   } // E2 event block
};

public ref struct EventReceiver {
   void H1() {
      Console::WriteLine("In event handler H1");
   }

   int H2(int i, float f) {
      Console::WriteLine("In event handler H2 with args {0} and {1}", i.ToString(), f.ToStr
ing());
      return 0;
   }
};
```

```
int main() {
   EventSource ^ pE = gcnew EventSource;
   EventReceiver ^ pR = gcnew EventReceiver;

   // hook event handlers
   pE->E += gcnew MyDel(pR, &EventReceiver::H1);
   pE->E2 += gcnew MyDel2(pR, &EventReceiver::H2);

   // raise events
   pE->E();
   pE->E2::raise(1, 2.2);   // call event through scope path

   // unhook event handlers
   pE->E -= gcnew MyDel(pR, &EventReceiver::H1);
   pE->E2 -= gcnew MyDel2(pR, &EventReceiver::H2);

   // raise events, but no handlers
   pE->E();
   pE->E2::raise(1, 2.5);
}
```

Output

```
In event handler H1
In event handler H2 with args 1 and 2.2
```

See Also

Reference

event (Visual C++)

How to: Define and Use Static Events

The following sample shows how to define and use static events.

Example

```
// mcppv2_events7.cpp
// compile with: /clr
using namespace System;

public delegate void MyDel();
public delegate int MyDel2(int, float);

ref class EventSource {
public:
   static MyDel ^ psE;
   static event MyDel2 ^ E2;   // event keyword, compiler generates add,
                               // remove, and Invoke

   static event MyDel ^ E {
      static void add(MyDel ^ p) {
         psE = static_cast<MyDel^> (Delegate::Combine(psE, p));
      }

      static void remove(MyDel^ p) {
         psE = static_cast<MyDel^> (Delegate::Remove(psE, p));
      }

      static void raise() {
         if (psE != nullptr)   //psE!=0 -> C2679, use nullptr
            psE->Invoke();
      }
   }

   static int Fire_E2(int i, float f) {
      return E2(i, f);
   }
};

public ref struct EventReceiver {
   void H1() {
      Console::WriteLine("In event handler H1");
   }

   int H2(int i, float f) {
      Console::WriteLine("In event handler H2 with args {0} and {1}", i.ToString(), f.ToStr
ing());
      return 0;
   }
};

int main() {
   EventSource^ pE = gcnew EventSource;
   EventReceiver^ pR = gcnew EventReceiver;

   // Called with "this"
   // hook event handlers
   pE->E += gcnew MyDel(pR, &EventReceiver::H1);
   pE->E2 += gcnew MyDel2(pR, &EventReceiver::H2);

   // raise events
   pE->E();
   pE->Fire_E2(11, 11.11);

   // unhook event handlers
   pE->E -= gcnew MyDel(pR, &EventReceiver::H1);
```

```
    pE->E2 -= gcnew MyDel2(pR, &EventReceiver::H2);

    // Not called with "this"
    // hook event handler
    EventSource::E += gcnew MyDel(pR, &EventReceiver::H1);
    EventSource::E2 += gcnew MyDel2(pR, &EventReceiver::H2);

    // raise events
    EventSource::E();
    EventSource::Fire_E2(22, 22.22);

    // unhook event handlers
    EventSource::E -= gcnew MyDel(pR, &EventReceiver::H1);
    EventSource::E2 -= gcnew MyDel2(pR, &EventReceiver::H2);
}
```

Output

```
In event handler H1
In event handler H2 with args 11 and 11.11
In event handler H1
In event handler H2 with args 22 and 22.22
```

See Also

Reference

event (Visual C++)

How to: Raise Events Defined in a Different Assembly

An event and event handler can be defined in one assembly, and consumed by another assembly.

Legacy Code Example

The following sample defines the event:

```
// mcppv2_events8.cpp
// compile with: /LD /clr
using namespace System;

public delegate void Del(String^ s);

public ref class Source {
public:
   event Del^ Event;
   void Fire(String^ s) {
      Event(s);
   }
};
```

The following client code consumes the event:

```
// mcppv2_events9.cpp
// compile with: /clr
#using "mcppv2_events8.dll"
using namespace System;

ref class Receiver {
public:
   void Handler(String^ s) {
      Console::WriteLine(s);
   }
};

int main() {
   Source^ src = gcnew Source;
   Receiver^ rc1 = gcnew Receiver;
   Receiver^ rc2 = gcnew Receiver;
   src -> Event += gcnew Del(rc1, &Receiver::Handler);
   src -> Event += gcnew Del(rc2, &Receiver::Handler);
   src->Fire("hello");
   src -> Event -= gcnew Del(rc1, &Receiver::Handler);
   src -> Event -= gcnew Del(rc2, &Receiver::Handler);
}
```

Output

```
hello
hello
```

See Also

Reference

event (Visual C++)

How to: Implement Abstract Events

The following sample shows how to implement an abstract event.

Example

```
// mcppv2_events10.cpp
// compile with: /clr /W1
using namespace System;
public delegate void Del();
public delegate void Del2(String^ s);

interface struct IEvent {
public:
   // note: in this case, no raised method defined
   event Del^ Event1;

   event Del2^ Event2 {
   public:
      void add(Del2^ _d);
      void remove(Del2^ _d);
      void raise(String^ s);
   }

   void fire();
};

ref class EventSource: public IEvent {
public:
   virtual event Del^ Event1;
   event Del2^ Event2 {
      virtual void add(Del2^ _d) {
         d = safe_cast<Del2^>(System::Delegate::Combine(d, _d));
      }

      virtual void remove(Del2^ _d) {
         d = safe_cast<Del2^>(System::Delegate::Remove(d, _d));
      }

      virtual void raise(String^ s) {
         if (d) {
            d->Invoke(s);
         }
      }
   }

   virtual void fire() {
      return Event1();
   }

private:
   Del2^ d;
};

ref class EventReceiver {
public:
   void func() {
      Console::WriteLine("hi");
   }

   void func(String^ str) {
      Console::WriteLine(str);
   }
};

int main () {
```

```
    IEvent^ es = gcnew EventSource;
    EventReceiver^ er = gcnew EventReceiver;
    es->Event1 += gcnew Del(er, &EventReceiver::func);
    es->Event2 += gcnew Del2(er, &EventReceiver::func);

    es->fire();
    es->Event2("hello from Event2");
    es->Event1 -= gcnew Del(er, &EventReceiver::func);
    es->Event2 -= gcnew Del2(er, &EventReceiver::func);
    es->Event2("hello from Event2");
}
```

Output

```
hi
hello from Event2
```

See Also

Reference

event (Visual C++)

Exception Handling under /clr

You can use both structured exception handling (SEH) and C++ exception handling under **/clr**. Under **/clr**, you can also handle CLR exceptions. A CLR exception is any exception thrown by a managed type. The System::Exception class provides many useful methods for processing CLR exceptions and is recommended as a base class for user-defined exception classes.

Note
Catching exception types derived from an interface is not supported under **/clr**. Also, the common language runtime does not permit you to catch stack overflow exceptions; a stack overflow exception will terminate the process.

In This Section

Basic Concepts in Using Managed Exceptions

Describes throwing exceptions and using **try**/**catch** blocks.

Differences in Exception Handling Behavior Under /CLR

Discusses the differences from the standard behavior of C++ exception handling.

finally

Discusses how to use the finally keyword.

How to: Define and Install a Global Exception Handler

Demonstrates how unhandled exceptions can be captured.

How to: Catch Exceptions in Native Code Thrown from MSIL

Discusses how to catch CLR and C++ exceptions in native code.

How to: Define and Install a Global Exception Handler

Demonstrates how to catch all unhandled exceptions.

Related Sections

Exception Handling

Describes exception handling in C++.

See Also

Concepts

New Language Features in Visual C++

Basic Concepts in Using Managed Exceptions

In this topic, the basic concepts for exception handling in managed applications are discussed:

- Throwing Exceptions Under /clr
- Try/Catch Blocks for CLR Extensions

For more detailed information on differences in exception handling between managed and unmanaged applications, see Differences in Exception Handling Behavior Under Managed Extensions for C++.

Throwing Exceptions Under /clr

The C++ throw expression is extended to throw a handle to a CLR type. The following example creates a custom exception type and then throws an instance of that type:

```
// clr_exception_handling.cpp
// compile with: /clr /c
ref struct MyStruct: public System::Exception {
public:
   int i;
};

void GlobalFunction() {
   MyStruct^ pMyStruct = gcnew MyStruct;
   throw pMyStruct;
}
```

A value type must be boxed before being thrown:

```
// clr_exception_handling_2.cpp
// compile with: /clr /c
value struct MyValueStruct {
   int i;
};

void GlobalFunction() {
   MyValueStruct v = {11};
   throw (MyValueStruct ^)v;
}
```

Try/Catch Blocks for CLR Extensions

The same **try**/**catch** block structure can be used for catching both CLR and native exceptions:

```
// clr_exception_handling_3.cpp
// compile with: /clr
using namespace System;
ref struct MyStruct : public Exception {
public:
   int i;
};

struct CMyClass {
public:
   double d;
};

void GlobalFunction() {
   MyStruct^ pMyStruct = gcnew MyStruct;
   pMyStruct->i = 11;
   throw pMyStruct;
}
```

```
void GlobalFunction2() {
   CMyClass c = {2.0};
   throw c;
}

int main() {
   for ( int i = 1; i >= 0; --i ) {
      try {
         if ( i == 1 )
            GlobalFunction2();
         if ( i == 0 )
            GlobalFunction();
      }
      catch ( CMyClass& catchC ) {
         Console::WriteLine( "In 'catch(CMyClass& catchC)'" );
         Console::WriteLine( catchC.d );
      }
      catch ( MyStruct^ catchException ) {
         Console::WriteLine( "In 'catch(MyStruct^ catchException)'" );
         Console::WriteLine( catchException->i );
      }
   }
}
```

Output

```
In 'catch(CMyClass& catchC)'
2
In 'catch(MyStruct^ catchException)'
11
```

Order of Unwinding for C++ Objects

Unwinding occurs for any C++ objects with destructors that may be on the run-time stack between the throwing function and the handling function. Because CLR types are allocated on the heap, unwinding does not apply to them.

The order of events for a thrown exception is as follows:

1. The runtime walks the stack looking for the appropriate catch clause, or in the case of SEH, an except filter for SEH, to catch the exception. Catch clauses are searched first in lexical order, and then dynamically down the call stack.
2. Once the correct handler is found, the stack is unwound to that point. For each function call on the stack, its local objects are destructed and __finally blocks are executed, from most nested outward.
3. Once the stack is unwound, the catch clause is executed.

Catching Unmanaged Types

When an unmanaged object type is thrown, it is wrapped with an exception of type System::Runtime.InteropServices::SEHException. When searching for the appropriate **catch** clause, there are two possibilities.

- If a native C++ type is encountered, the exception is unwrapped and compared to the type encountered. This comparison allows a native C++ type to be caught in the normal way.
- However, if a **catch** clause of type **SEHException** or any of its base classes is examined first, the clause will intercept the exception. Therefore, you should place all catch clauses that catch native C++ types first before any catch clauses of CLR types.

Note that

```
catch(Object^)
```

and

```
catch(...)
```

will both catch any thrown type including SEH exceptions.

If an unmanaged type is caught by catch(Object^), it will not destroy the thrown object.

When throwing or catching unmanaged exceptions, one of the Visual C++ exception handling compiler options must be used /EHs or /Eha.

See Also

Reference

safe_cast

Exception Handling in Visual C++

Other Resources

Exception Handling under /clr

Differences in Exception Handling Behavior Under /CLR

Basic Concepts in Using Managed Exceptions discusses exception handling in managed applications. In this topic, differences from the standard behavior of exception handling and some restrictions are discussed in detail. For more information, see The _set_se_translator Function.

Jumping Out of a Finally Block

In native C/C++ code, jumping out of a **__finally** block using structured exception handling (SEH) is allowed although it produces a warning. Under /clr, jumping out of a **finally** block causes an error:

```
// clr_exception_handling_4.cpp
// compile with: /clr
int main() {
   try {}
   finally {
      return 0;   // also fails with goto, break, continue
    }
}   // C3276
```

Raising Exceptions Within an Exception Filter

When an exception is raised during the processing of an exception filter within managed code, the exception is caught and treated as if the filter returns 0.

This is in contrast to the behavior in native code where a nested exception is raised, the **ExceptionRecord** field in the **EXCEPTION_RECORD** structure (as returned by GetExceptionInformation) is set, and the **ExceptionFlags** field sets the 0x10 bit. The following example illustrates this difference in behavior:

```
// clr_exception_handling_5.cpp
#include <windows.h>
#include <stdio.h>
#include <assert.h>

#ifndef false
#define false 0
#endif

int *p;

int filter(PEXCEPTION_POINTERS ExceptionPointers) {
   PEXCEPTION_RECORD ExceptionRecord =
                     ExceptionPointers->ExceptionRecord;

   if ((ExceptionRecord->ExceptionFlags & 0x10) == 0) {
      // not a nested exception, throw one
      *p = 0; // throw another AV
   }
   else {
      printf("Caught a nested exception\n");
      return 1;
    }

   assert(false);

   return 0;
}

void f(void) {
   __try {
      *p = 0;   // throw an AV
   }
   __except(filter(GetExceptionInformation())) {
      printf_s("We should execute this handler if "
```

```
                  "compiled to native\n");
    }
}

int main() {
   __try {
      f();
   }
   __except(1) {
      printf_s("The handler in main caught the "
               "exception\n");
    }
}
```

Output

```
Caught a nested exception
We should execute this handler if compiled to native
```

Disassociated Rethrows

/clr does not support rethrowing an exception outside of a catch handler (known as a disassociated rethrow). Exceptions of this type are treated as a standard C++ rethrow. If a disassociated rethrow is encountered when there is an active managed exception, the exception is wrapped as a C++ exception and then rethrown. Exceptions of this type can only be caught as an exception of type System::SEHException.

The following example demonstrates a managed exception rethrown as a C++ exception:

```
// clr_exception_handling_6.cpp
// compile with: /clr
using namespace System;
#include <assert.h>
#include <stdio.h>

void rethrow( void ) {
   // This rethrow is a dissasociated rethrow.
   // The exception would be masked as SEHException.
   throw;
}

int main() {
   try {
      try {
         throw gcnew ApplicationException;
      }
      catch ( ApplicationException^ ) {
         rethrow();
         // If the call to rethrow() is replaced with
         // a throw statement within the catch handler,
         // the rethrow would be a managed rethrow and
         // the exception type would remain
         // System::ApplicationException
      }
   }

    catch ( ApplicationException^ ) {
      assert( false );

      // This will not be executed since the exception
      // will be masked as SEHException.
    }
   catch ( Runtime::InteropServices::SEHException^ ) {
      printf_s("caught an SEH Exception\n" );
    }
}
```

Output

```
caught an SEH Exception
```

Exception Filters and EXCEPTION_CONTINUE_EXECUTION

If a filter returns **EXCEPTION_CONTINUE_EXECUTION** in a managed application, it is treated as if the filter returned **EXCEPTION_CONTINUE_SEARCH**. For more information on these constants, see try-except Statement.

The following example demonstrates this difference:

```
// clr_exception_handling_7.cpp
#include <windows.h>
#include <stdio.h>
#include <assert.h>

int main() {
   int Counter = 0;
   __try {
      __try  {
         Counter -= 1;
         RaiseException (0xe0000000|'seh',
                         0, 0, 0);
         Counter -= 2;
      }
      __except (Counter) {
         // Counter is negative,
         // indicating "CONTINUE EXECUTE"
         Counter -= 1;
      }
    }
    __except(1) {
      Counter -= 100;
   }

   printf_s("Counter=%d\n", Counter);
}
```

Output

```
Counter=-3
```

The _set_se_translator Function

The translator function, set by a call to **_set_se_translator**, affects only catches in unmanaged code. The following example demonstrates this limitation:

```
// clr_exception_handling_8.cpp
// compile with: /clr /EHa
#include <iostream>
#include <windows.h>
#include <eh.h>
#pragma warning (disable: 4101)
using namespace std;
using namespace System;

#define MYEXCEPTION_CODE 0xe0000101

class CMyException {
public:
   unsigned int m_ErrorCode;
   EXCEPTION_POINTERS * m_pExp;

   CMyException() : m_ErrorCode( 0 ), m_pExp( NULL ) {}

   CMyException( unsigned int i, EXCEPTION_POINTERS * pExp )
         : m_ErrorCode( i ), m_pExp( pExp ) {}

   CMyException( CMyException& c ) : m_ErrorCode( c.m_ErrorCode ),
                                     m_pExp( c.m_pExp ) {}
```

```
   friend ostream& operator <<
                 ( ostream& out, const CMyException& inst ) {
      return out <<  "CMyException[\n" <<
             "Error Code: " << inst.m_ErrorCode <<  "]";
    }
};

#pragma unmanaged
void my_trans_func( unsigned int u, PEXCEPTION_POINTERS pExp ) {
   cout <<  "In my_trans_func.\n";
   throw CMyException( u, pExp );
}

#pragma managed
void managed_func() {
   try  {
      RaiseException( MYEXCEPTION_CODE, 0, 0, 0 );
   }
   catch ( CMyException x ) {}
   catch ( ... ) {
      printf_s("This is invoked since "
               "_set_se_translator is not "
               "supported when /clr is used\n" );
    }
}

#pragma unmanaged
void unmanaged_func() {
   try  {
      RaiseException( MYEXCEPTION_CODE,
                      0, 0, 0 );
   }
   catch ( CMyException x ) {
      printf("Caught an SEH exception with "
             "exception code: %x\n", x.m_ErrorCode );
    }
    catch ( ... ) {}
}

// #pragma managed
int main( int argc, char ** argv ) {
   _set_se_translator( my_trans_func );

   // It does not matter whether the translator function
   // is registered in managed or unmanaged code
   managed_func();
   unmanaged_func();
}
```

Output

```
This is invoked since _set_se_translator is not supported when /clr is used
In my_trans_func.
Caught an SEH exception with exception code: e0000101
```

See Also

Reference

safe_cast

Exception Handling in Visual C++

Other Resources

Exception Handling under /clr

finally

In addition to **try** and **catch** clauses, CLR exception handling supports a **finally** clause. The semantics are identical to the **__finally** block in structured exception handling (SEH). A **__finally** block can follow a **try** or **catch** block.

Remarks

The purpose of the **finally** block is to clean up any resources left after the exception occurred. Note that the **finally** block is always executed, even if no exception was thrown. The **catch** block is only executed if a managed exception is thrown within the associated **try** block.

finally is a context-sensitive keyword; see Context-Sensitive Keywords for more information.

Example

The following example demonstrates a simple **finally** block:

```
// keyword__finally.cpp
// compile with: /clr
using namespace System;

ref class MyException: public System::Exception{};

void ThrowMyException() {
   throw gcnew MyException;
}

int main() {
   try {
      ThrowMyException();
   }
   catch ( MyException^ e ) {
      Console::WriteLine(  "in catch" );
      Console::WriteLine( e->GetType() );
   }
   finally {
      Console::WriteLine(  "in finally" );
   }
}
```

Output

```
in catch
MyException
in finally
```

See Also

Other Resources

Exception Handling under /clr

How to: Catch Exceptions in Native Code Thrown from MSIL

In native code, you can catch native C++ exception from MSIL. You can catch CLR exceptions with **__try** and **__except**.

For more information, see Structured Exception Handling (C++) and C++ Exception Handling.

Example

The following sample defines a module with two functions, one that throws a native exception, and another that throws an MSIL exception.

```
// catch_MSIL_in_native.cpp
// compile with: /clr /c
void Test() {
   throw ("error");
}

void Test2() {
   throw (gcnew System::Exception("error2"));
}
```

The following sample defines a module that catches a native and MSIL exception.

```
// catch_MSIL_in_native_2.cpp
// compile with: /clr catch_MSIL_in_native.obj
#include <iostream>
using namespace std;
void Test();
void Test2();

void Func() {
   // catch any exception from MSIL
   // should not catch Visual C++ exceptions like this
   // runtime may not destroy the object thrown
   __try {
      Test2();
   }
   __except(1) {
      cout << "caught an exception" << endl;
   }

}

int main() {
   // catch native C++ exception from MSIL
   try {
      Test();
   }
   catch(char * S) {
      cout << S << endl;
   }
   Func();
}
```

Output

```
error
caught an exception
```

See Also

Other Resources

Exception Handling under /clr

How to: Define and Install a Global Exception Handler

The following code example demonstrates how unhandled exceptions can be captured. The example form contains a button that, when pressed, performs a null reference, causing an exception to be thrown. This functionality represents a typical code failure. The resulting exception is caught by the application-wide exception handler installed by the main function.

This is accomplished by binding a delegate to the ThreadException event. In this case, subsequent exceptions are then sent to the **`App::OnUnhandled`** method.

Example

```
// global_exception_handler.cpp
// compile with: /c /clr
#using <system.dll>
#using <system.drawing.dll>
#using <system.windows.forms.dll>

using namespace System;
using namespace System::Threading;
using namespace System::Drawing;
using namespace System::Windows::Forms;

ref class MyForm : public Form
{
   Button^ b;
public:
   MyForm( )
   {
      b = gcnew Button( );
      b->Text = "Do Null Access";
      b->Size = Drawing::Size(150, 30);
      b->Click += gcnew EventHandler(this, &MyForm::OnClick);
      Controls->Add(b);
   }
   void OnClick(Object^ sender, EventArgs^ args)
   {
      // do something illegal, like call through a null pointer...
      Object^ o = nullptr;
      o->ToString( );
   }
};

ref class App
{
public:
   static void OnUnhandled(Object^ sender, ThreadExceptionEventArgs^ e)
   {
      MessageBox::Show(e->Exception->Message, "Global Exeception");
      Application::ExitThread( );
   }
};

int main()
{
   Application::ThreadException += gcnew
      ThreadExceptionEventHandler(App::OnUnhandled);

   MyForm^ form = gcnew MyForm( );
   Application::Run(form);
}
```

See Also

Other Resources

Exception Handling under /clr

Explicit Overrides

This topic discusses how to explicitly override a base class or interface member. A named (explicit) override should only be used to override a method with a derived method that has a different name.

```
function-declarator = type::function[,type::function] {
/* function definition */
}
function-declarator = function { /* function definition */ }
```

Parameters

function

The function you want to override. It is possible to specify more than one function; separate function names with a comma.

function-declarator

The return type, name, and argument list of the overriding function. Note that the overriding function does not have to have the same name as the function being overridden. It is possible to specify more than one function; separate function names with a comma.

type

The base type that contains the function you want to override.

Remarks

For information related to modifying the behavior of inherited types and inherited type members, see Override Specifiers.

For information about explicit overrides in native code or code compiled with **/clr:oldSyntax**, see Explicit Overrides (C++).

Example

The following sample shows a simple, implicit override and implementation of a member in a base interface, not using explicit overrides.

```
// explicit_override_1.cpp
// compile with: /clr
interface struct I1 {
   virtual void f();
};

ref class X : public I1 {
public:
   virtual void f() {
      System::Console::WriteLine("X::f override of I1::f");
   }
};

int main() {
   I1 ^ MyI = gcnew X;
   MyI -> f();
}
```

Output

```
X::f override of I1::f
```

The following sample shows how to implement all interface members with a common signature, using explicit override syntax.

```
// explicit_override_2.cpp
// compile with: /clr
interface struct I1 {
   virtual void f();
};
```

```
interface struct I2 {
   virtual void f();
};

ref struct X : public I1, I2 {
   virtual void f() = I1::f, I2::f {
      System::Console::WriteLine("X::f override of I1::f and I2::f");
   }
};

int main() {
   I1 ^ MyI = gcnew X;
   I2 ^ MyI2 = gcnew X;
   MyI -> f();
   MyI2 -> f();
}
```

Output

```
X::f override of I1::f and I2::f
X::f override of I1::f and I2::f
```

The following sample shows how a function override can have a different name from the function it is implementing.

```
// explicit_override_3.cpp
// compile with: /clr
interface struct I1 {
   virtual void f();
};

ref class X : public I1 {
public:
   virtual void g() = I1::f {
      System::Console::WriteLine("X::g");
   }
};

int main() {
   I1 ^ a = gcnew X;
   a->f();
}
```

Output

```
X::g
```

The following sample shows an explicit interface implementation that implements a type safe collection.

```
// explicit_override_4.cpp
// compile with: /clr /LD
using namespace System;
ref class R : ICloneable {
   int X;

   virtual Object^ C() sealed = ICloneable::Clone {
      return this->Clone();
   }

public:
   R() : X(0) {}
   R(int x) : X(x) {}

   virtual R^ Clone() {
      R^ r = gcnew R;
      r->X = this->X;
      return r;
```

```
    }
};
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

for each, in

The **for each** statement is used to iterate through a collection of elements.

```
for each (type identifier in expression) {
      statements
}
```

Parameters

type

The type of *identifier*.

identifier

The iteration variable that represents the collection element. When *identifier* is a tracking reference, you can modify the element.

expression

A managed array expression or collection. The compiler must be able to convert the collection element from Object to the *identifier* type.

expression evaluates to a type that implements IEnumerable, IEnumerable, or a type that defines a GetEnumerator method. In the latter case, GetEnumerator should either return a type that implements IEnumerator or declares all the methods defined in IEnumerator.

statements

One or more statements to be executed.

Remarks

The **for each** statement is used to iterate through a collection. It is possible to modify elements in a collection, but you cannot add or delete elements.

The *statements* are executed for each element in the array or collection. After the iteration has been completed for all the elements in the collection, control is transferred to the next statement following the **for each** block.

for each and **in** are context-sensitive keywords; see Context-Sensitive Keywords for more information.

In the development environment, you can get F1 help on by highlighting the keyword, (**for each**) and pressing F1.

For more information, see,

- How to: Iterate Over STL Collection with for each
- How to: Iterate Over Arrays with for each
- How to: Iterate Over a Generic Collection with for each
- for each, in

Example

This sample shows how to iterate through a string with for each.

```
// for_each_string.cpp
// compile with: /clr
using namespace System;

ref struct MyClass {
   property String ^ MyStringProperty;
};

int main() {
   String ^ MyString = gcnew String("abcd");
```

```
    for each ( Char c in MyString )
       Console::Write(c);

    Console::WriteLine();

    MyClass ^ x = gcnew MyClass();
    x->MyStringProperty = "Testing";

    for each( Char c in x->MyStringProperty )
       Console::Write(c);
 }
```

Output

```
abcd
Testing
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

How to: Iterate Over STL Collection with for each

In addition to .NEt Framework collections, the **for each** keyword can also be used to iterate over Standard C++ Library (STL) collections. An STL collection is also known as a container. For more information, see STL Containers.

Example

This sample uses **for each** in a <map>.

```
// for_each_stl.cpp
// compile with: /EHsc
#include <map>
#include <iostream>
using namespace std;

int main() {
   int retval  = 0;
   map<const char*, int> months;

   months["january"] = 31;
   months["february"] = 28;
   months["march"] = 31;
   months["april"] = 30;
   months["may"] = 31;
   months["june"] = 30;
   months["july"] = 31;
   months["august"] = 31;
   months["september"] = 30;
   months["october"] = 31;
   months["november"] = 30;
   months["december"] = 31;

   map<const char*, int> months_30;

   for each( pair<const char*, int> c in months )
      if ( c.second == 30 )
         months_30[c.first] = c.second;

   for each( pair<const char*, int> c in months_30 )
      retval++;

   cout << "Months with 30 days = " << retval << endl;
}
```

Output

```
Months with 30 days = 4
```

This sample uses const& for an iteration variable with STL containers. You can use an & as an iteration variable on any collection of a type that can be declared as a T&.

```
// for_each_stl_2.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>
using namespace std;

int main() {
   int retval = 0;

   vector<int> col(3);
   col[0] = 10;
   col[1] = 20;
   col[2] = 30;
```

```
    for each( const int& c in col )
       retval += c;

    cout << "retval: " << retval << endl;
 }
```

Output

```
retval: 60
```

See Also

Reference

for each, in

How to: Iterate Over Arrays with for each

This topic shows how to use the for each, in keyword on different types of arrays.

Example

This sample shows how to use **for each** on array of reference types. Note that if any dimension of a multi dimensional array is zero, the **for each** loop will not iterate over the array.

```
// for_each_arrays.cpp
// compile with: /clr
using namespace System;
ref struct MyClass {
   void Test() { Console::WriteLine("in MyClass"); }
};

ref struct MyClass2 {
   void Test() { Console::WriteLine("in MyClass2"); }
};

int main() {
   array<MyClass ^> ^ MyArray = gcnew array<MyClass ^>(2);

   int i = 0;
   for each ( MyClass ^ c in MyArray ) {
      Console::Write("{0} = ", i++);
      c -> Test();
   }

   Console::WriteLine();

   array< MyClass2 ^, 2 > ^ MyArray2 = gcnew array< MyClass2 ^, 2 >(2, 2);
   i = 0;
   for each ( MyClass2 ^ c in MyArray2 ) {
      Console::Write("{0} = ", i++);
      c -> Test();
   }

   array< MyClass2 ^, 2 > ^ MyArray3 = gcnew array< MyClass2 ^, 2 >(2, 0);
   i = 0;
   for each ( MyClass2 ^ c in MyArray3 ) {
      Console::Write("{0} = ", i++);
      c -> Test();
   }
}
```

Output

```
0 = in MyClass
1 = in MyClass

0 = in MyClass2
1 = in MyClass2
2 = in MyClass2
3 = in MyClass2
```

This sample shows for each iterating over a ArrayList, which implements IEnumerable.

```
// for_each_arrays_2.cpp
// compile with: /clr
using namespace System;
using namespace System::Collections;

int main() {
   int retval = 0;
```

```
   ArrayList ^ arr = gcnew ArrayList();
   arr->Add(10);
   arr->Add(20);
   arr->Add(30);

   for each ( int c in arr )
      retval += c;
   Console::WriteLine(retval);
}
```

Output

```
60
```

This sample shows how to iterate over an array of arrays.

```
// for_each_arrays_3.cpp
// compile with: /clr
using namespace System;

#define ARRAY_SIZE 2

int main() {
   int i, j;

   // Declares an array of managed arrays of Int32.
   array< array< Int32 > ^ > ^ IntArray = gcnew array<array< Int32 > ^>(ARRAY_SIZE);

   for (i = 0 ; i < ARRAY_SIZE ; i++) {
      IntArray[i] = gcnew array< Int32 >(ARRAY_SIZE);
         for ( int j = 0 ; j < ARRAY_SIZE ; j++ )
            IntArray[i][j] = i + 10;
   }

   for (i = 0 ; i < ARRAY_SIZE ; i++)
      for (int j = 0 ; j < ARRAY_SIZE ; j++)
         Console::WriteLine("IntArray[{0}] = {1}", i, IntArray[i][j]);
   Console::WriteLine();

   for each (array<Int32> ^ xyz in IntArray)
      for each ( int c in xyz )
         Console::WriteLine(c);
}
```

Output

```
IntArray[0] = 10
IntArray[0] = 10
IntArray[1] = 11
IntArray[1] = 11

10
10
11
11
```

See Also

Reference

for each, in

How to: Iterate Over a Generic Collection with for each

The Generics (Visual C++) feature of Visual C++ allows you to create generic collections.

Example

This sample shows how to use **for each** with a simple generic value type collection.

```
// for_each_generics.cpp
// compile with: /clr
using namespace System;
using namespace System::Collections::Generic;

generic <class T>
public value struct MyArray : public IEnumerable<T> {

   MyArray( array<T>^ d ) {
      data = d;
   }

   ref struct enumerator : IEnumerator<T> {
      enumerator( MyArray^ myArr ) {
         colInst = myArr;
         currentIndex = -1;
      }

      virtual bool MoveNext() = IEnumerator<T>::MoveNext {
         if ( currentIndex < colInst->data->Length - 1 ) {
            currentIndex++;
            return true;
         }

         return false;
      }

      virtual property T Current {
         T get() {
            return colInst->data[currentIndex];
         }
      };

      property Object^ CurrentNonGeneric {
         virtual Object^ get() = System::Collections::IEnumerator::Current::get {
            return colInst->data[currentIndex];
         }
      };

      virtual void Reset() {}
      ~enumerator() {}

      MyArray^ colInst;
      int currentIndex;
   };

   array<T>^ data;

   virtual IEnumerator<T>^ GetEnumerator() {
      return gcnew enumerator(*this);
   }
   virtual System::Collections::IEnumerator^ GetEnumeratorNonGeneric() = System::Collection
s::IEnumerable::GetEnumerator {
      return gcnew enumerator(*this);
   }
};

int main() {
```

```
    MyArray<int> col = MyArray<int>( gcnew array<int>{10, 20, 30 } );

    for each ( Object^ c in col )
       Console::WriteLine((int)c);
 }
```

Output

```
10
20
30
```

See Also

Reference

for each, in

How to: Iterate Over a User-Defined Collection with for each

For a class to be a managed collection, it needs a non-private GetEnumerator function that returns a handle to an Enumerator class or an interface. An enumerator class must contain the declaration for non-static MoveNext function and Current property.

Example

Simple user defined collection with reference types.

```
// for_each_user_defined_collections.cpp
// compile with: /clr
using namespace System;
public interface struct IMyEnumerator {
   bool MoveNext();
   property Object^ Current {
      Object^ get();
   }
   void Reset();
};

public ref struct MyArray {

   MyArray( array<int>^ d ) {
      data = d;
   }

   ref struct enumerator : IMyEnumerator {
      enumerator( MyArray^ myArr ) {
         colInst = myArr;
         currentIndex = -1;
      }

      virtual bool MoveNext() {
         if( currentIndex < colInst->data->Length - 1 ) {
            currentIndex++;
            return true;
         }
         return false;
      }

      property Object^ Current {
         virtual Object^ get() {
            return colInst->data[currentIndex];
         }
      };

      virtual void Reset() {}
      ~enumerator() {}

      MyArray^ colInst;
      int currentIndex;
   };

   array<int>^ data;

   IMyEnumerator^ GetEnumerator() {
      return gcnew enumerator(this);
   }
};

int main() {
   int retval = 0;

   MyArray^ col = gcnew MyArray( gcnew array<int>{10, 20, 30 } );

   for each ( Object^ c in col )
```

```
        retval += (int)c;

    retval -= 10 + 20 + 30;

    for each ( int c in gcnew MyArray( gcnew array<int>{10, 20, 30 } ) )
        retval += c;

    retval -= 10 + 20 + 30;

    Console::WriteLine("Return Code: {0}", retval );
    return retval;
  }
```

Output

```
Return Code: 0
```

See Also

Reference

for each, in

Friend Assemblies

You can make all types at namespace or global scope in an assembly component accessible to one or more client assemblies or .netmodules.

Remarks

To make types at namespace or global scope in an assembly component accessible to a client assembly or .netmodule, do the following:

- In the component, specify an assembly attribute: InternalsVisibleToAttribute, and pass the name of the client assembly or .netmodule that will access all types at namespace or global scope in the component. You can specify multiple client assemblies or .netmodules by specifying additional attributes.
- In the client assembly or .netmodule, when referencing the component assembly (with **#using**), pass the **as_friend** attribute. If you specify the **as_friend** attribute for an assembly that does not specify InternalsVisibleToAttribute, a runtime exception will be thrown if you try to access a type at namespace or global scope in the component.

A build error will result if the assembly containing the **InternalsVisibleToAttribute** attribute does not have a strong name, but the client assembly (uses the **as_friend** attribute) does.

Although types at namespace and global scope can be visible to a client assembly or .netmodule, member accessibility is still in effect. For example, you will not be able to access a private member.

Access to all types in an assembly must be explicitly granted. For example, assembly C will not have access to all types in assembly A if assembly C references assembly B and where assembly B has access to all types in assembly A.

For information on how to specify the accessibility of types outside an assembly, see Type Visibility.

For information on signing (giving a strong name) to an assembly built using the Visual C++ compiler, see Strong Name Assemblies (Assembly Signing).

An alternative to using the friend assemblies feature, you can use StrongNameIdentityPermission to restrict access to individual types.

Example

The following sample defines a component that specifies a client assembly that will have access to all types in the component.

```
// friend_assemblies.cpp
// compile with: /clr /LD
using namespace System::Runtime::CompilerServices;
using namespace System;
// an assembly attribute, not bound to a type
[assembly:InternalsVisibleTo("friend_assemblies_2")];

ref class Class1 {
public:
   void Test_Public() {
      Console::WriteLine("Class1::Test_Public");
   }
};
```

Example

The following sample access a private type in the component.

```
// friend_assemblies_2.cpp
// compile with: /clr
#using "friend_assemblies.dll" as_friend

int main() {
   Class1 ^ a = gcnew Class1;
   a->Test_Public();
}
```

Output

```
Class1::Test_Public
```

Example

The following sample defines a component but does not specify a client assembly that will have access to all types in the component.

Note that the component is linked using **/opt:noref**. This ensures that private types are emitted in the component's metadata, which is not required when the **InternalsVisibleTo** attribute is present. For more information, see /OPT (Optimizations).

```
// friend_assemblies_3.cpp
// compile with: /clr /LD /link /opt:noref
using namespace System;

ref class Class1 {
public:
   void Test_Public() {
      Console::WriteLine("Class1::Test_Public");
   }
};
```

Example

The following sample defines a client that tries to access a private type in a component that does not give access to its private types. Due to the behavior of the runtime, if you want to catch the exception, you must attempt to access a private type in helper function.

```
// friend_assemblies_4.cpp
// compile with: /clr
#using "friend_assemblies_3.dll" as_friend
using namespace System;

void Test() {
   Class1 ^ a = gcnew Class1;
}

int main() {
   // to catch this kind of exception, use a helper function
   try {
      Test();
   }
   catch(MethodAccessException ^ e) {
      Console::WriteLine("caught an exception");
   }
}
```

Output

```
caught an exception
```

Example

The following sample shows how to create a strong-name component that specifies a client assembly that will have access to all types in the component.

Note that the component must specify its public key. The following is a suggested sequence of steps to create a key pair and get the public key:

sn -d friend_assemblies.snk

sn -k friend_assemblies.snk

sn -i friend_assemblies.snk friend_assemblies.snk

sn -pc friend_assemblies.snk key.publickey

sn -tp key.publickey

```
// friend_assemblies_5.cpp
// compile with: /clr /LD /link /keyfile:friend_assemblies.snk
using namespace System::Runtime::CompilerServices;
using namespace System;
// an assembly attribute, not bound to a type

[assembly:InternalsVisibleTo("friend_assemblies_6, PublicKey=0024000004800000940000000060200
0000240000525341310004000001000100 0bf45d77fd991f3bff0ef51af48a12d35699e04616f27ba561195a69e
bd3449c345389dc9603d65be8cd1987bc7ea48bdda35ac7d57d3d82c666b7fc1a5b79836d139ef0ac8c4e715434
211660f481612771a9f7059b9b742c3d8af00e01716ed4b872e6f1be0e94863eb5745224f0deaba5b137624d704
9b6f2d87fba639fc5")];

private ref class Class1 {
public:
   void Test_Public() {
      Console::WriteLine("Class1::Test_Public");
   }
};
```

Example

The following sample access a private type in the strong name component.

```
// friend_assemblies_6.cpp
// compile with: /clr /link /keyfile:friend_assemblies.snk
#using "friend_assemblies_5.dll" as_friend

int main() {
   Class1 ^ a = gcnew Class1;
   a->Test_Public();
}
```

Output

```
Class1::Test_Public
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

gcnew

gcnew creates an instance of a managed type (reference or value type) on the garbage collected heap. The result of the evaluation of a **gcnew** expression is a handle (^) to the type being created.

Example

```
// mcppv2_gcnew_1.cpp
// compile with: /clr
ref struct Message {
   System::String ^ sender, ^ receiver, ^ data;
};

int main() {
   Message ^ h_Message  = gcnew Message ;
}
```

It is possible to create an instance of a managed type, where the managed type contains a nested type other than a reference type:

```
// mcppv2_gcnew_2.cpp
// compile with: /clr
ref class MyClass {
public:
   void Test() {}

   value class Value_Nested_Class {
   public:
      int i;
   };
};

int main() {
   MyClass ^ h_MyClass = gcnew MyClass;

   MyClass::Value_Nested_Class y;
   y.i = 32;
   System::Console::WriteLine(y.i);
}
```

Output

```
32
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

Generics (Visual C++)

Visual C++ 2005 introduces support for generic methods and types in Visual C++. In this section, find out how to author your own generic methods and types in Visual C++ and how to use generic types authored in C# or another language in Visual C++. This section also includes a comparison of generics and C++ templates.

In This Section

Overview of Generics in Visual C++

Information about generics are, the motivation for the language feature, definitions of terms used to describe generics, and information about the use of reference types and value types as type parameters for generics.

Generic Functions

Information about defining and using generic functions and methods.

Generic Classes (Visual C++)

Information about defining and using generic classes.

Generic Interfaces (Visual C++)

Information about defining and using generic interfaces.

Generic Delegates (Visual C++)

Information about defining and using generic delegates.

Constraints

Information about using constraints in generic types.

Consuming Generics from Other .NET Languages

Information about using generics defined in .NET assemblies, possibly authored in other languages, in Visual C++.

Generics and Templates

A comparison of generics and templates, when to use each, and how to combine them usefully.

Related Sections

How to: Iterate Over a Generic Collection with for each

Using the for each, in keyword on a generic collection.

Generics in the .NET Framework Class Library (C# Programming Guide)

Discusses .NET Framework class library's **System.Collections.Generic**, namespace, which includes several ready-to-use generic collection classes and associated interfaces.

Introduction to Generics (C# Programming Guide)

The generics section in the C# documentation.

Generics in J#

The generics section in the J# documentation.

Generic Types in Visual Basic

The generics section in the Visual Basic documentation.

Templates Overview

Information on templates in C++.

See Also

Concepts

New Language Features in Visual C++

Overview of Generics in Visual C++

Generics are parameterized types supported by the common language runtime. A parameterized type is a type that is defined with an unknown type parameter that is specified when the generic is used.

Why Generics?

C++ supports templates and both templates and generics support parameterized types to create typed collection classes. However, templates provide compile-time parameterization. You cannot reference an assembly containing a template definition and create new specializations of the template. Once compiled, a specialized template looks like any other class or method. In contrast, generics are emitted in MSIL as a parameterized type known by the runtime to be a parameterized type; source code that references an assembly containing a generic type can create specializations of the generic type. For more information on the comparison of Visual C++ templates and generics, see Generics and Templates.

Generic Functions and Types

Class types, as long as they are managed types, may be generic. An example of this might be a `List` class. The type of object in the list would be the type parameter. If you needed a `List` class for many different types of objects, before generics you might have used a `List` that takes **System::Object** as the item type. But that would allow any object (including objects of the wrong type) to be used in the list. Such a list would be called an untyped collection class. At best, you could check the type at runtime and throw an exception. Or, you might have used a template, which would lose its generic quality once compiled into an assembly. Consumers of your assembly could not create their own specializations of the template. Generics allow you to create typed collection classes, say `List<int>` (read as "List of int") and `List<double>` ("List of double") which would generate a compile-time error if you tried to put a type that the collection was not designed to accept into the typed collection. In addition, these types remain generic after they are compiled.

A description of the syntax of generic classes may be found in Generic Classes (Visual C++). A new namespace, System.Collections.Generic, introduces a set of parameterized collection types including Dictionary, List and LinkedList. See Generics in the .NET Framework Class Library (C# Programming Guide) for more information.

Both instance and static class member functions, delegates, and global functions may also be generic. Generic functions may be necessary if the function's parameters are of an unknown type, or if the function itself must work with generic types. In many cases where **System::Object** may have been used in the past as a parameter for an unknown object type, a generic type parameter may be used instead, allowing for more type-safe code. Any attempt to pass in a type that the function was not designed for would be flagged as an error at compile time. Using **System::Object** as a function parameter, the inadvertent passing of an object that the function wasn't intended to deal with would not be detected, and you would have to cast the unknown object type to a specific type in the function body, and account for the possibility of an InvalidCastException. With a generic, code attempting to pass an object to the function would cause a type conflict so the function body is guaranteed to have the correct type.

The same benefits apply to collection classes built on generics. Collection classes in the past would use **System::Object** to store elements in a collection. Insertion of objects of a type that the collection was not designed for was not flagged at compile time, and often not even when the objects were inserted. Usually, an object would be cast to some other type when it was accessed in the collection. Only when the cast failed would the unexpected type be detected. Generics solves this problem at compile time by detecting any code that inserts a type that doesn't match (or implicitly convert to) the type parameter of the generic collection.

For a description of the syntax, see Generic Functions.

Terminology Used With Generics

Type Parameters

A generic declaration contains one or more unknown types known as *type parameters*. Type parameters are given a name which stands for the type within the body of the generic declaration. The type parameter is used as a type within the body of the generic declaration. The generic declaration for List<T> contains the type parameter T.

Type Arguments

The *type argument* is the actual type used in place of the type parameter when the generic is specialized for a specific type or types. For example, **int** is the type argument in `List<int>`. Value types and handle types are the only types allowed in as a generic type argument.

Constructed Type

A type constructed from a generic type is referred to as a *constructed type*. A type not fully specified, such as `List<T>` is an

open constructed type; a type fully specified, such as `List<double>`, is a *closed constructed type* or *specialized type*. Open constructed types may be used in the definition of other generic types or methods and may not be fully specified until the enclosing generic is itself specified. For example, the following is a use of an open constructed type as a base class for a generic:

```
// generics_overview.cpp
// compile with: /clr /c
generic <typename T>
ref class List {};
generic <typename T>
ref class Queue : public List<T> {};
```

Constraint

A constraint is a restriction on the types that may be used as a type parameter. For example, a given generic class could accept only classes that inherit from a specified class, or implement a specified interface. For more information, see Constraints.

Reference Types and Value Types

Handles types and value types may be used as type arguments. In the generic definition, in which either type may be used, the syntax is that of reference types. For example, the **->** operator is used to access members of the type of the type parameter whether or not the type eventually used is a reference type or a value type. When a value type is used as the type argument, the runtime generates code that uses the value types directly without boxing the value types.

When using a reference type as a generic type argument, use the handle syntax. When using a value type as a generic type argument, use the name of the type directly.

```
// generics_overview_2.cpp
// compile with: /clr
generic <typename T>
ref class GenericType {};
ref class ReferenceType {};
value struct ValueType {};
int main() {
GenericType<ReferenceType^> x;
GenericType<ValueType> y;
}
```

Type Parameters

Type parameters in a generic class are treated like other identifiers. However, because the type is not known, there are restrictions on their use. For example, you cannot use members and methods of the type parameter class unless the type parameter is known to support these members. That is, in order access a member through the type parameter, you must add the type that contains the member to the type parameter's constraint list.

```
// generics_overview_3.cpp
// compile with: /clr
```

```
interface class I {
   void f1();
   void f2();
};

ref struct R : public I {
   virtual void f1() {}
   virtual void f2() {}
   virtual void f3() {}
};

generic <typename T>
where T : I
void f(T t) {
```

```
   t->f1();
   t->f2();
   safe_cast<R^>(t)->f3();
}

int main() {
   f(gcnew R());
}
```

These restrictions apply to operators as well. An unconstrained generic type parameter may not use the **==** and **!=** operators to compare two instances of the type parameter, in case the type does not support these operators. These checks are necessary for generics, but not for templates, because generics may be specialized at runtime with any class that satisfies the constraints, when it is too late to check for the use of invalid members.

A default instance of the type parameter may be created by using the **()** operator. For example:

```
T t = T();
```

where `T` is a type parameter in a generic class or method definition, initializes the variable to its default value. If `T` is a ref class it will be a null pointer; if `T` is a value class, the object is initialized to zero. This is called a *default initializer*.

See Also

Other Resources

Generics (Visual C++)

Generic Functions

A generic function is a function that is declared with type parameters. When called, actual types are used instead of the type parameters.

```
[attributes] [modifiers]
return-type identifier <type-parameter identifier(s)>
[type-parameter-constraints clauses]

([formal-parameters])
{
      function-body
}
```

Remarks

The above syntax includes these terms, defined as follows:

attributes (Optional)

Additional declarative information. For more information on attributes and attribute classes, see attributes.

modifiers (Optional)

A modifier for the function, such as **static**. **virtual** is not allowed since virtual methods may not be generic.

return-type

The type returned by the method. If the return type is void, no return value is required.

identifier

The function name.

type-parameter identifier(s)

Comma-separated identifiers list.

formal-parameters (Optional)

Parameter list.

type-parameter-constraints-clauses

This specifies restrictions on the types that may be used as type arguments, and takes the form specified in Constraints.

function-body

The body of the method, which may refer to the type parameter identifiers.

Generic functions are functions declared with a generic type parameter. They may be methods in a class or struct, or standalone functions. A single generic declaration implicitly declares a family of functions that differ only in the substitution of a different actual type for the generic type parameter.

When called, the generic type parameter is replaced by an actual type. The actual type may be explicitly specified in angled brackets using syntax similar to a template function call. If called without the type parameters, the compiler will attempt to deduce the actual type from the parameters supplied in the function call. If the intended type argument cannot be deduced from the parameters used, the compiler will report an error.

```
// generics_generic_function_1.cpp
// compile with: /clr
generic <typename ItemType>
void G(int i) {}

ref struct A {
   generic <typename ItemType>
   void G(ItemType) {}
```

```
   generic <typename ItemType>
   static void H(int i) {}
};

int main() {
   A myObject;

   // generic function call
   myObject.G<int>(10);

   // generic function call with type parameters deduced
   myObject.G(10);

   // static generic function call
   A::H<int>(10);

   // global generic function call
   G<int>(10);
}
```

Generic functions can be overloaded based on signature or arity, the number of type parameters on a function.. Also, generic functions can be overloaded with non-generic functions of the same name, as long as the functions differ in some type parameters. For example, the following functions can be overloaded:

```
// generics_generic_function_2.cpp
// compile with: /clr /c
ref struct MyClass {
   void MyMythod(int i) {}

   generic <class T>
   void MyMythod(int i) {}

   generic <class T, class V>
   void MyMythod(int i) {}
};
```

Example

The following example uses a generic function to find the first element in an array. It declares `MyClass`, which inherits from the base class `MyBaseClass`. `MyClass` contains a generic function, `MyFunction`, which calls another generic function, `MyBaseClassFunction`, within the base class. In **main**, the generic function, `MyFunction`, is called using different type arguments.

```
// generics_generic_function_3.cpp
// compile with: /clr
using namespace System;

ref class MyBaseClass {
protected:
   generic <class ItemType>
   ItemType MyBaseClassFunction(ItemType item) {
      return item;
   }
};

ref class MyClass: public MyBaseClass {
public:
   generic <class ItemType>
   ItemType MyFunction(ItemType item) {
      return MyBaseClass::MyBaseClassFunction<ItemType>(item);
   }
};

int main() {
   MyClass^ myObj = gcnew MyClass();
```

```
    // Call MyFunction using an int.
    Console::WriteLine("My function returned an int: {0}",
                           myObj->MyFunction<int>(2003));

    // Call MyFunction using a string.
    Console::WriteLine("My function returned a string: {0}",
    myObj->MyFunction<String^>("Hello generic functions!"));
  }
```

Output

```
My function returned an int: 2003
My function returned a string: Hello generic functions!
```

See Also

Other Resources

Generics (Visual C++)

Generic Classes (Visual C++)

A generic class is declared using the following form:

```
[attributes]
generic <class-key type-parameter-identifier(s)>
[constraint-clauses]
[accessibility-modifiers] ref class identifier  [modifiers]
[: base-list]
{
      class-body
} [declarators] [;]
```

Remarks

In the above syntax, the following terms are used:

attributes (optional)

Additional declarative information. For more information on attributes and attribute classes, see Attributes.

class-key

Either `class` or `typename`

type-parameter-identifier(s),

Comma-separated list of identifiers specifying the names of the type parameters.

constraint-clauses

A list (not comma-separated) of **where** clauses specifying the constraints for the type parameters. Takes the form:

`where` *type-parameter-identifier* `:` *constraint-list* `...`

constraint-list

class-or-interface[`,` ...]

accessibility-modifiers

Allowed accessibility modifiers include **public** and **private**.

identifier

The name of the generic class, any valid C++ identifier.

modifiers (optional)

Allowed modifiers include **sealed** and **abstract**.

base-list

A list that contains the one base class and any implemented interfaces, all separated by commas.

class-body,

The body of the class, containing fields, member functions, etc.

declarators

Declarations of any variables of this type. For example: `^`*identifier*[`,` ...]

You can declare generic classes such as these (note that the keyword **class** may be used instead of **typename**). In this example, `ItemType`, `KeyType` and `ValueType` are unknown types that are specified at the point where the type. `HashTable<int, int>` is a constructed type of the generic type `HashTable<KeyType, ValueType>`. A number of different constructed types can be constructed from a single generic type. Constructed types constructed from generic classes are treated like any other ref class type.

```
// generic_classes_1.cpp
// compile with: /clr
using namespace System;
generic <typename ItemType>
ref struct Stack {
   // ItemType may be used as a type here
   void Add(ItemType item) {}
};

generic <typename KeyType, typename ValueType>
ref class HashTable {};

// The keyword class may be used instead of typename:
generic <class ListItem>
ref class List {};

int main() {
   HashTable<int, Decimal>^ g1 = gcnew HashTable<int, Decimal>();
}
```

Both value types (either built-in types such as **int** or **double**, or user-defined value types) and reference types may be used as a generic type argument. The syntax within the generic definition is the same regardless. Syntactically, the unknown type is treated as if it were a reference type. However, the runtime is able to determine that if the type actually used is a value type and substitute the appropriate generated code for direct access to members. Value types used as generic type arguments are not boxed and so do not suffer the performance penalty associated with boxing. The syntax used within the body of the generic should be **T^** and '**->**' instead of '**.**'. Any use of gcnew for the type parameter will be appropriately interpreted by the runtime as the simple creation of a value type if the type argument is a value type.

You can also declare a generic class with Constraints on the types that can be used for the type parameter. In the following example any type used for `ItemType` must implement the `IItem` interface. Attempting to use **int**, for example, which does not implement `IItem`, would produce a compile-time error because the type argument does not satisfy the constraint.

```
// generic_classes_2.cpp
// compile with: /clr /c
interface class IItem {};
generic <class ItemType>
where ItemType : IItem
ref class Stack {};
```

Generic classes in the same namespace cannot be overloaded by only changing the number or the types of type parameters. However, if each class lives in a different namespace, they can be overloaded. For example, consider the following two classes, `MyClass` and `MyClass<ItemType>`, in the namespaces `A` and `B`. The two classes can then be overloaded in a third namespace C:

```
// generic_classes_3.cpp
// compile with: /clr /c
namespace A {
   ref class MyClass {};
}

namespace B {
   generic <typename ItemType>
   ref class MyClass2 { };
}

namespace C {
   using namespace A;
   using namespace B;

   ref class Test {
      static void F() {
         MyClass^ m1 = gcnew MyClass();   // OK
         MyClass2<int>^ m2 = gcnew MyClass2<int>();   // OK
      }
   };
}
```

The base class and base interfaces cannot be type parameters. However, the base class can involve the type parameter as an argument, as in the following case:

```
// generic_classes_4.cpp
// compile with: /clr /c
generic <typename ItemType>
interface class IInterface {};

generic <typename ItemType>
ref class MyClass : IInterface<ItemType> {};
```

Constructors and destructors are executed once for each object instance (as usual); static constructors are executed once for each constructed type.

Fields in Generic Classes

This section demonstrates the use of instance and static fields in generic classes.

Instance Variables

Instance variables of a generic class can have types and variable initializers that include any type parameters from the enclosing class.

Example

In the following example, three different instances of the generic class, MyClass<ItemType>, are created by using the appropriate type arguments (**int**, **double**, and **string**).

```
// generics_instance_fields1.cpp
// compile with: /clr
// Instance fields on generic classes
using namespace System;

generic <typename ItemType>
ref class MyClass {
// Field of the type ItemType:
public :
   ItemType field1;
   // Constructor using a parameter of the type ItemType:
   MyClass(ItemType p) {
     field1 = p;
   }
};

int main() {
   // Instantiate an instance with an integer field:
   MyClass<int>^ myObj1 = gcnew MyClass<int>(123);
   Console::WriteLine("Integer field = {0}", myObj1->field1);

   // Instantiate an instance with a double field:
   MyClass<double>^ myObj2 = gcnew MyClass<double>(1.23);
   Console::WriteLine("Double field = {0}", myObj2->field1);

   // Instantiate an instance with a String field:
   MyClass<String^>^ myObj3 = gcnew MyClass<String^>("ABC");
   Console::WriteLine("String field = {0}", myObj3->field1);
   }
```

Output

```
Integer field = 123
Double field = 1.23
String field = ABC
```

The following example demonstrates using static fields and a static constructor within a generic class.

```
// generics_static2.cpp
// compile with: /clr
using namespace System;

interface class ILog {
   void Write(String^ s);
};

ref class DateTimeLog : ILog {
public:
   virtual void Write(String^ s) {
      Console::WriteLine( "{0}\t{1}", DateTime::Now, s);
   }
};

ref class PlainLog : ILog {
public:
   virtual void Write(String^ s) { Console::WriteLine(s); }
};

generic <typename LogType>
where LogType : ILog
ref class G {
   static LogType s_log;

public:
   G(){}
   void SetLog(LogType log) { s_log = log; }
   void F() { s_log->Write("Test1"); }
   static G() { Console::WriteLine("Static constructor called."); }
};

int main() {
   G<PlainLog^>^ g1 = gcnew G<PlainLog^>();
   g1->SetLog(gcnew PlainLog());
   g1->F();

   G<DateTimeLog^>^ g2 = gcnew G<DateTimeLog^>();
   g2->SetLog(gcnew DateTimeLog());

   // prints date
   // g2->F();
}
```

Sample Output

```
Static constructor called.
Static constructor called.
Static constructor called.
Test1
```

The following example declares a non-generic method, `ProtectData`, inside a generic class, `MyClass<ItemType>`. The method uses the class type parameter `ItemType` in its signature in an open constructed type.

```
// generics_non_generic_methods1.cpp
// compile with: /clr
// Non-generic methods within a generic class.
using namespace System;

generic <typename ItemType>
ref class MyClass {
public:
   String^ name;
   ItemType data;

   MyClass(ItemType x) {
      data = x;
```

```
   }

   // Non-generic method using the type parameter:
   virtual void ProtectData(MyClass<ItemType>^ x) {
      data = x->data;
   }
};

// ItemType defined as String^
ref class MyMainClass: MyClass<String^> {
public:
   // Passing "123.00" to the constructor:
   MyMainClass(): MyClass<String^>("123.00") {
      name = "Jeff Smith";
   }

   virtual void ProtectData(MyClass<String^>^ x) override {
      x->data = String::Format("${0}**", x->data);
   }

   static void Main() {
      MyMainClass^ x1 = gcnew MyMainClass();

      x1->ProtectData(x1);
      Console::WriteLine("Name: {0}", x1->name);
      Console::WriteLine("Amount: {0}", x1->data);
   }
};

int main() {
   MyMainClass::Main();
}
```

Output

```
Name: Jeff Smith
Amount: $123.00**
```

```
// generics_method2.cpp
// compile with: /clr /c
generic <typename Type1>
ref class G {
public:
   // Generic method having a type parameter
   // from the class, Type1, and its own type
   // parameter, Type2
   generic <typename Type2>
   void Method1(Type1 t1, Type2 t2) { F(t1, t2); }

   // Non-generic method:
   // Can use the class type param, Type1, but not Type2.
   void Method2(Type1 t1) { F(t1, t1); }

   void F(Object^ o1, Object^ o2) {}
};
```

The non-generic method is still generic in the sense that it is parameterized by the class's type parameter, but it has no additional type parameters.

All types of methods in generic classes can be generic, including static, instance, and virtual methods.

The following example demonstrates declaring and using generic methods within generic classes:

```
// generics_generic_method2.cpp
// compile with: /clr
using namespace System;
generic <class ItemType>
```

```
ref class MyClass {
public:
   // Declare a generic method member.
   generic <class Type1>
   String^ MyMethod(ItemType item, Type1 t) {
      return String::Concat(item->ToString(), t->ToString());
   }
};

int main() {
   // Create instances using different types.
   MyClass<int>^ myObj1 = gcnew MyClass<int>();
   MyClass<String^>^ myObj2 = gcnew MyClass<String^>();
   MyClass<String^>^ myObj3 = gcnew MyClass<String^>();

   // Calling MyMethod using two integers.
   Console::WriteLine("MyMethod returned: {0}",
            myObj1->MyMethod<int>(1, 2));

   // Calling MyMethod using an integer and a string.
   Console::WriteLine("MyMethod returned: {0}",
            myObj2->MyMethod<int>("Hello #", 1));

   // Calling MyMethod using two strings.
   Console::WriteLine("MyMethod returned: {0}",
       myObj3->MyMethod<String^>("Hello ", "World!"));

   // generic methods can be called without specifying type arguments
   myObj1->MyMethod<int>(1, 2);
   myObj2->MyMethod<int>("Hello #", 1);
   myObj3->MyMethod<String^>("Hello ", "World!");
}
```

Output

```
MyMethod returned: 12
MyMethod returned: Hello #1
MyMethod returned: Hello World!
```

```
// generics_linked_list.cpp
// compile with: /clr
using namespace System;
generic <class ItemType>
ref class LinkedList {
// The node class:
public:
   ref class Node {
   // The link field:
   public:
      Node^ next;
      // The data field:
      ItemType item;
   } ^first, ^current;
};

ref class ListBuilder {
public:
   void BuildIt(LinkedList<double>^ list) {
      /* Build the list */
      double m[5] = {0.1, 0.2, 0.3, 0.4, 0.5};
      Console::WriteLine("Building the list:");

      for (int n=0; n<=4; n++) {
         // Create a new node:
         list->current = gcnew LinkedList<double>::Node();

         // Assign a value to the data field:
         list->current->item = m[n];
```

```
         // Set the link field "next" to be the same as
         // the "first" field:
         list->current->next = list->first;

         // Redirect "first" to the new node:
         list->first = list->current;

         // Display node's data as it builds:
         Console::WriteLine(list->current->item);
      }
   }

   void ReadIt(LinkedList<double>^ list) {
      // Read the list
      // Make "first" the "current" link field:
      list->current = list->first;
      Console::WriteLine("Reading nodes:");

      // Read nodes until current == null:
      while (list->current != nullptr) {
         // Display the node's data field:
         Console::WriteLine(list->current->item);

         // Move to the next node:
         list->current = list->current->next;
      }
   }
};

int main() {
   // Create a list:
   LinkedList<double>^ aList = gcnew LinkedList<double>();

   // Initialize first node:
   aList->first = nullptr;

   // Instantiate the class, build, and read the list:
   ListBuilder^ myListBuilder = gcnew ListBuilder();
   myListBuilder->BuildIt(aList);
   myListBuilder->ReadIt(aList);
}
```

Output

```
Building the list:
0.1
0.2
0.3
0.4
0.5
Reading nodes:
0.5
0.4
0.3
0.2
0.1
```

This example shows declarations of an instance property within a generic class.

```
// generics_generic_properties1.cpp
// compile with: /clr
using namespace System;

generic <typename ItemType>
ref class MyClass {
private:
   property ItemType myField;
```

```
public:
   property ItemType MyProperty {
      ItemType get() {
         return myField;
      }
      void set(ItemType value) {
         myField = value;
      }
   }
};

int main() {
   MyClass<String^>^ c = gcnew MyClass<String^>();
   MyClass<int>^ c1 = gcnew MyClass<int>();

   c->MyProperty = "John";
   c1->MyProperty = 234;

   Console::Write("{0}, {1}", c->MyProperty, c1->MyProperty);
}
```

Output

```
John, 234
```

The next example shows a generic class with an event.

```
// generics_generic_with_event.cpp
// compile with: /clr
// Declare a generic class with an event and
// invoke events.
using namespace System;

// declare delegates
generic <typename ItemType>
delegate void ClickEventHandler(ItemType);

// generic class that defines events
generic <typename ItemType>
ref class EventSource {
public:
   // declare the event OnClick
   event ClickEventHandler<ItemType>^ OnClick;
   void FireEvents(ItemType item) {
      // raises events
      OnClick(item);
   }
};

// generic class that defines methods that will called when
// event occurs
generic <typename ItemType>
ref class EventReceiver {
public:
   void OnMyClick(ItemType item) {
     Console::WriteLine("OnClick: {0}", item);
   }
};

int main() {
   EventSource<String^>^ MyEventSourceString =
                   gcnew EventSource<String^>();
   EventSource<int>^ MyEventSourceInt = gcnew EventSource<int>();
   EventReceiver<String^>^ MyEventReceiverString =
                   gcnew EventReceiver<String^>();
   EventReceiver<int>^ MyEventReceiverInt = gcnew EventReceiver<int>();
```

```
    // hook handler to event
    MyEventSourceString->OnClick += gcnew ClickEventHandler<String^>(
        MyEventReceiverString, &EventReceiver<String^>::OnMyClick);
    MyEventSourceInt->OnClick += gcnew ClickEventHandler<int>(
              MyEventReceiverInt, &EventReceiver<int>::OnMyClick);

    // invoke events
    MyEventSourceString->FireEvents("Hello");
    MyEventSourceInt->FireEvents(112);

    // unhook handler to event
    MyEventSourceString->OnClick -= gcnew ClickEventHandler<String^>(
         MyEventReceiverString, &EventReceiver<String^>::OnMyClick);
    MyEventSourceInt->OnClick -= gcnew ClickEventHandler<int>(
         MyEventReceiverInt, &EventReceiver<int>::OnMyClick);
}
```

The following example declares a generic struct, `MyGenStruct`, with one field, `myField`, and assigns values of different types (**int**, **double**, **String^**) to this field.

```
// generics_generic_struct1.cpp
// compile with: /clr
using namespace System;

generic <typename ItemType>
ref struct MyGenStruct {
public:
   ItemType myField;

   ItemType AssignValue(ItemType item) {
      myField = item;
      return myField;
   }
};

int main() {
   int myInt = 123;
   MyGenStruct<int>^ myIntObj = gcnew MyGenStruct<int>();
   myIntObj->AssignValue(myInt);
   Console::WriteLine("The field is assigned the integer value: {0}",
            myIntObj->myField);

   double myDouble = 0.123;
   MyGenStruct<double>^ myDoubleObj = gcnew MyGenStruct<double>();
   myDoubleObj->AssignValue(myDouble);
   Console::WriteLine("The field is assigned the double value: {0}",
            myDoubleObj->myField);

   String^ myString = "Hello Generics!";
   MyGenStruct<String^>^ myStringObj = gcnew MyGenStruct<String^>();
   myStringObj->AssignValue(myString);
   Console::WriteLine("The field is assigned the string: {0}",
            myStringObj->myField);
}
```

Output

```
The field is assigned the integer value: 123
The field is assigned the double value: 0.123
The field is assigned the string: Hello Generics!
```

Static Variables

On the creation of a new generic type, new instances of any static variables are created and any static constructor for that type is executed.

Static variables can use any type parameters from the enclosing class.

Methods in Generic Classes

Methods in generic classes can be generic themselves; non-generic methods will be implicitly parameterized by the class type parameter.

The following special rules apply to methods within generic classes:

- Methods in generic classes can use type parameters as parameters, return types, or local variables.
- Methods in generic classes can use open or closed constructed types as parameters, return types, or local variables.

Non-Generic Methods in Generic Classes

Methods in generic classes that have no additional type parameters are usually referred to as non-generic although they are implicitly parameterized by the enclosing generic class.

The signature of a non-generic method can include one or more type parameters of the enclosing class, either directly or in an open constructed type. For example:

```
void MyMethod(MyClass<ItemType> x) {}
```

The body of such methods can also use these type parameters.

Generic Methods in Generic Classes

You can declare generic methods in both generic and non-generic classes. For example:

Using Nested Types in Generic Classes

Just as with ordinary classes, you can declare other types inside a generic class. The nested class declaration is implicitly parameterized by the type parameters of the outer class declaration. Thus, a distinct nested class is defined for each constructed outer type. For example, in the declaration,

```
// generic_classes_5.cpp
// compile with: /clr /c
generic <typename ItemType>
ref struct Outer {
   ref class Inner {};
};
```

The type Outer<int>::Inner is not the same as the type Outer<double>::Inner.

As with generic methods in generic classes, additional type parameters can be defined for the nested type. If you use the same type parameter names in the inner and outer class, the inner type parameter will hide the outer type parameter.

```
// generic_classes_6.cpp
// compile with: /clr /c
generic <typename ItemType>
ref class Outer {
   ItemType outer_item;   // refers to outer ItemType

   generic <typename ItemType>
   ref class Inner {
      ItemType inner_item;   // refers to Inner ItemType
   };
};
```

Since there is no way to refer to the outer type parameter, the compiler will produce a warning in this situation.

When constructed nested generic types are named, the type parameter for the outer type is not included in the type parameter list for the inner type, even though the inner type is implicitly parameterized by the outer type's type parameter. In the above case, a name of a constructed type would be Outer<int>::Inner<string>.

The following example demonstrates building and reading a linked list using nested types in generic classes.

Properties, Events, Indexers and Operators in Generic Classes

- Properties, events, indexers and operators can use the type parameters of the enclosing generic class as return values, parameters, or local variables, such as when `ItemType` is a type parameter of a class:

```
public ItemType MyProperty {}
```

- Properties, events, indexers and operators cannot themselves be parameterized.

Generic Structs

The rules for declaring and using generic structs are the same as those for generic classes, except for the differences noted in in the Visual C++ language reference.

See Also

Other Resources

Generics (Visual C++)

Generic Interfaces (Visual C++)

The restrictions that apply to type parameters on classes are the same as those that apply to type parameters on interfaces (see Generic Classes (Visual C++)).

The rules that control function overloading are the same for functions within generic classes or generic interfaces.

Explicit interface member implementations work with constructed interface types in the same way as with simple interface types (see the following examples).

For more information on interfaces, see interface class.

```
[attributes] generic <class-key type-parameter-identifier[, ...]>
[type-parameter-constraints-clauses][accesibility-modifiers] interface class identifier [:
base-list] {   interface-body} [declarators] ;
```

Remarks

attributes (optional)

Additional declarative information. For more information on attributes and attribute classes, see Attributes.

class-key

class or **typename**

type-parameter-identifier(s)

Comma-separated identifiers list.

type-parameter-constraints-clauses

Takes the form specified in Constraints

accessibility-modifiers (optional)

Accessibility modifiers (e.g. **public, private**).

identifier

The interface name.

base-list (optional)

A list that contains one or more explicit base interfaces separated by commas.

interface-body

Declarations of the interface members.

declarators (optional)

Declarations of variables based on this type.

Example

The following example demonstrates how to declare and instantiate a generic interface. In the example, the generic interface `IList<ItemType>` is declared. It is then implemented by two generic classes, `List1<ItemType>` and `List2<ItemType>`, with different implementations.

```
// generic_interface.cpp
// compile with: /clr
using namespace System;

// An exception to be thrown by the List when
// attempting to access elements beyond the
// end of the list.
ref class ElementNotFoundException : Exception {};

// A generic List interface
```

```
generic <typename ItemType>
public interface class IList {
   ItemType MoveFirst();
   bool Add(ItemType item);
   bool AtEnd();
   ItemType Current();
   void MoveNext();
};

// A linked list implementation of IList
generic <typename ItemType>
public ref class List1 : public IList<ItemType> {
   ref class Node {
      ItemType m_item;

   public:
      ItemType get_Item() { return m_item; };
      void set_Item(ItemType value) { m_item = value; };

      Node^ next;

      Node(ItemType item) {
         m_item = item;
         next = nullptr;
      }
   };

   Node^ first;
   Node^ last;
   Node^ current;

   public:
   List1() {
      first = nullptr;
      last = first;
      current = first;
   }

   virtual ItemType MoveFirst() {
      current = first;
      if (first != nullptr)
        return first->get_Item();
      else
         return ItemType();
   }

   virtual bool Add(ItemType item) {
      if (last != nullptr) {
         last->next = gcnew Node(item);
         last = last->next;
      }
      else {
         first = gcnew Node(item);
         last = first;
         current = first;
      }
      return true;
   }

   virtual bool AtEnd() {
      if (current == nullptr )
        return true;
      else
        return false;
   }

   virtual ItemType Current() {
       if (current != nullptr)
```

```
          return current->get_Item();
        else
          throw gcnew ElementNotFoundException();
    }

    virtual void MoveNext() {
       if (current != nullptr)
        current = current->next;
       else
         throw gcnew ElementNotFoundException();
    }
};

// An array implementation of IList
generic <typename ItemType>
ref class List2 : public IList<ItemType> {
   array<ItemType>^ item_array;
   int count;
   int current;

   public:

   List2() {
      // not yet possible to declare an
      // array of a generic type parameter
      item_array = gcnew array<ItemType>(256);
      count = current = 0;
   }

   virtual ItemType MoveFirst() {
      current = 0;
      return item_array[0];
   }

   virtual bool Add(ItemType item) {
      if (count < 256)
         item_array[count++] = item;
      else
        return false;
      return true;
   }

   virtual bool AtEnd() {
      if (current >= count)
        return true;
      else
        return false;
   }

   virtual ItemType Current() {
      if (current < count)
        return item_array[current];
      else
        throw gcnew ElementNotFoundException();
   }

   virtual void MoveNext() {
      if (current < count)
         ++current;
      else
         throw gcnew ElementNotFoundException();
   }
};

// Add elements to the list and display them.
generic <typename ItemType>
void AddStringsAndDisplay(IList<ItemType>^ list, ItemType item1, ItemType item2) {
   list->Add(item1);
```

```
   list->Add(item2);
   for (list->MoveFirst(); ! list->AtEnd(); list->MoveNext())
     Console::WriteLine(list->Current());
}

int main() {
   // Instantiate both types of list.

   List1<String^>^ list1 = gcnew List1<String^>();
   List2<String^>^ list2 = gcnew List2<String^>();

   // Use the linked list implementation of IList.
   AddStringsAndDisplay<String^>(list1, "Linked List", "List1");

   // Use the array implementation of the IList.
   AddStringsAndDisplay<String^>(list2, "Array List", "List2");
}
```

Output

```
Linked List
List1
Array List
List2
```

This example declares a generic interface, `IMyGenIface`, and two non-generic interfaces, `IMySpecializedInt` and `ImySpecializedString`, that specialize `IMyGenIface`. The two specialized interfaces are then implemented by two classes, `MyIntClass` and `MyStringClass`. The example shows how to specialize generic interfaces, instantiate generic and non-generic interfaces, and call the explicitly implemented members on the interfaces.

```
// generic_interface2.cpp
// compile with: /clr
// Specializing and implementing generic interfaces.
using namespace System;

generic <class ItemType>
public interface class IMyGenIface {
   void Initialize(ItemType f);
};

public interface class IMySpecializedInt: public IMyGenIface<int> {
   void Display();
};

public interface class IMySpecializedString: public IMyGenIface<String^> {
   void Display();
};

public ref class MyIntClass: public IMySpecializedInt {
   int myField;

public:
   virtual void Initialize(int f) {
      myField = f;
   }

   virtual void Display() {
      Console::WriteLine("The integer field contains: {0}", myField);
   }
};

public ref struct MyStringClass: IMySpecializedString {
   String^ myField;

public:
   virtual void Initialize(String^ f) {
      myField = f;
    }
```

```
   virtual void Display() {
      Console::WriteLine("The String field contains: {0}", myField);
   }
};

int main() {
   // Instantiate the generic interface.
   IMyGenIface<int>^ myIntObj = gcnew MyIntClass();

   // Instantiate the specialized interface "IMySpecializedInt."
   IMySpecializedInt^ mySpIntObj = (IMySpecializedInt^) myIntObj;

   // Instantiate the generic interface.
   IMyGenIface<String^>^ myStringObj = gcnew MyStringClass();

   // Instantiate the specialized interface "IMySpecializedString."
   IMySpecializedString^ mySpStringObj =
            (IMySpecializedString^) myStringObj;

   // Call the explicitly implemented interface members.
   myIntObj->Initialize(1234);
   mySpIntObj->Display();

   myStringObj->Initialize("My string");
   mySpStringObj->Display();
}
```

Output

```
The integer field contains: 1234
The String field contains: My string
```

See Also

Other Resources

Generics (Visual C++)

Generic Delegates (Visual C++)

You can use generic type parameters with delegates. For more information on delegates, see delegate.

```
[attributes]
generic < [class | typename] type-parameter-identifiers >
[type-parameter-constraints-clauses]
[accessibility-modifiers] delegate result-type identifier
([formal-parameters]);
```

Parameters

attributes (Optional)

Additional declarative information. For more information on attributes and attribute classes, see Attributes.

type-parameter-identifier(s)

Comma-separated list of identifiers for the type parameters.

type-parameter-constraints-clauses

Takes the form specified in Constraints

accessibility-modifiers (Optional)

Accessibility modifiers (e.g. **public**, **private**).

result-type

The return type of the delegate.

identifier

The name of the delegate.

formal-parameters (Optional)

The parameter list of the delegate.

Example

The delegate type parameters are specified at the point where a delegate object is created. Both the delegate and method associated with it must have the same signature. The following is an example of a generic delegate declaration.

```
// generics_generic_delegate1.cpp
// compile with: /clr /c
generic < class ItemType>
delegate ItemType GenDelegate(ItemType p1, ItemType% p2);
```

The following sample shows that

- You cannot use the same delegate object with different constructed types. Create different delegate objects for different types.
- A generic delegate can be associated with a generic method.
- When a generic method is called without specifying type arguments, the compiler tries to infer the type arguments for the call.

```
// generics_generic_delegate2.cpp
// compile with: /clr
generic < class ItemType>
delegate ItemType GenDelegate(ItemType p1, ItemType% p2);

generic < class ItemType>
ref struct MyGenClass {
```

```
   ItemType MyMethod(ItemType i, ItemType % j) {
      return ItemType();
   }
};

ref struct MyClass {
   generic < class ItemType>
   static ItemType MyStaticMethod(ItemType i, ItemType % j) {
      return ItemType();
   }
};

int main() {
   MyGenClass<int> ^ myObj1 = gcnew MyGenClass<int>();
   MyGenClass<double> ^ myObj2 = gcnew MyGenClass<double>();
   GenDelegate<int>^ myDelegate1 =
      gcnew GenDelegate<int>(myObj1, &MyGenClass<int>::MyMethod);

   GenDelegate<double>^ myDelegate2 =
      gcnew GenDelegate<double>(myObj2, &MyGenClass<double>::MyMethod);

   GenDelegate<int>^ myDelegate =
      gcnew GenDelegate<int>(&MyClass::MyStaticMethod<int>);
}
```

The following example declares a generic delegate `GenDelegate<ItemType>`, and then instantiates it by associating it to the method `MyMethod` that uses the type parameter `ItemType`. Two instances of the delegate (an integer and a double) are created and invoked.

```
// generics_generic_delegate.cpp
// compile with: /clr
using namespace System;

// declare generic delegate
generic <typename ItemType>
delegate ItemType GenDelegate (ItemType p1, ItemType% p2);

// Declare a generic class:
generic <typename ItemType>
ref class MyGenClass {
public:
   ItemType MyMethod(ItemType p1, ItemType% p2) {
      p2 = p1;
      return p1;
    }
};

int main() {
   int i = 0, j = 0;
   double m = 0.0, n = 0.0;

   MyGenClass<int>^ myObj1 = gcnew MyGenClass<int>();
   MyGenClass<double>^ myObj2 = gcnew MyGenClass<double>();

   // Instantiate a delegate using int.
   GenDelegate<int>^ MyDelegate1 =
      gcnew GenDelegate<int>(myObj1, &MyGenClass<int>::MyMethod);

   // Invoke the integer delegate using MyMethod.
   i = MyDelegate1(123, j);

   Console::WriteLine(
      "Invoking the integer delegate: i = {0}, j = {1}", i, j);

   // Instantiate a delegate using double.
   GenDelegate<double>^ MyDelegate2 =
      gcnew GenDelegate<double>(myObj2, &MyGenClass<double>::MyMethod);
```

```
    // Invoke the integer delegate using MyMethod.
    m = MyDelegate2(0.123, n);

    Console::WriteLine(
       "Invoking the double delegate: m = {0}, n = {1}", m, n);
 }
```

Output

```
Invoking the integer delegate: i = 123, j = 123
Invoking the double delegate: m = 0.123, n = 0.123
```

See Also

Other Resources

Generics (Visual C++)

Constraints

In generic type or method declarations, you can qualify a type parameter with constraints. A constraint is a requirement that types used as type arguments must satisfy. For example, a constraint might be that the type argument must implement a certain interface or inherit from a specific class.

Constraints are optional; not specifying a constraint on a parameter is equivalent to constraining that parameter to Object.

```
where type-parameter: constraint list
```

Parameters

type-parameter

One of the type parameters, to be constrained.

constraint list,

constraint list is a comma-separated list of constraint specifications. The list can include interfaces to be implemented by the type parameter.

The list can also include a class. For the type argument to satisfy a base class constraint, it must be the same class as the constraint or derive from the constraint.

You can also specify **gcnew()** to indicate the type argument must have a public parameterless constructor; or **ref class** to indicate the type argument must be a reference type, including any class, interface, delegate, or array type; or **value class** to indicate the type argument must be a value type. Any value type except Nullable<T> can be specified.

You can also specify a generic parameter as a constraint. The type argument supplied for the type you are constraining must be or derive from the type of the constraint. This is called a naked type constraint.

Remarks

The constraint clause consists of **where** followed by a type parameter, a colon (**:**), and the constraint, which specifies the nature of the restriction on the type parameter. **where** is a context-sensitive keyword; see Context-Sensitive Keywords for more information. Separate multiple **where** clauses with a space.

Constraints are applied to type parameters to place limitations on the types that can be used as arguments for a generic type or method.

Class and interface constraints specify that the argument types must be or inherit from a specified class or implement a specified interface.

The application of constraints to a generic type or method allows code in that type or method to take advantage of the known features of the constrained types. For example, you can declare a generic class such that the type parameter implements the **IComparable<T>** interface:

```
// generics_constraints_1.cpp
// compile with: /c /clr
using namespace System;
generic <typename T>
where T : IComparable<T>
ref class List {};
```

This constraint requires that a type argument used for `T` implements `IComparable<T>` at compile time. It also allows interface methods, such as **CompareTo**, to be called. No cast is needed on an instance of the type parameter to call interface methods.

Static methods in the type argument's class cannot be called through the type parameter; they can be called only through the actual named type.

A constraint cannot be a value type, including built-in types such as **int** or **double**. Since value types cannot have derived classes, only one class would ever be able to satisfy the constraint. In that case, the generic can be rewritten with the type parameter replaced by the specific value type.

Constraints are required in some cases since the compiler will not allow the use of methods or other features of an unknown type unless the constraints imply that the unknown type supports the methods or interfaces.

Multiple constraints for the same type parameter can be specified in a comma-separated list

```
// generics_constraints_2.cpp
// compile with: /c /clr
using namespace System;
using namespace System::Collections::Generic;
generic <typename T>
where T : List<T>, IComparable<T>
ref class List {};
```

With multiple type parameters, use one **where** clause for each type parameter. For example:

```
// generics_constraints_3.cpp
// compile with: /c /clr
using namespace System;
using namespace System::Collections::Generic;

generic <typename K, typename V>
   where K: IComparable<K>
   where V: IComparable<K>
ref class Dictionary {};
```

To summarize, use constraints in your code according to the following rules:

- If multiple constraints are listed, the constraints may be listed in any order.
- Constraints can also be class types, such as abstract base classes. However, constraints cannot be value types or sealed classes.
- Constraints cannot themselves be type parameters, but they can involve the type parameters in an open constructed type. For example:

  ```
  // generics_constraints_4.cpp
  // compile with: /c /clr
  generic <typename T>
  ref class G1 {};

  generic <typename Type1, typename Type2>
  where Type1 : G1<Type2>   // OK, G1 takes one type parameter
  ref class G2{};
  ```

Example

The following example demonstrates using constraints to call instance methods on type parameters.

```
// generics_constraints_5.cpp
// compile with: /clr
using namespace System;

interface class IAge {
   int Age();
};

ref class MyClass {
public:
   generic <class ItemType> where ItemType : IAge
   bool isSenior(ItemType item) {
      // Because of the constraint,
      // the Age method can be called on ItemType.
      if (item->Age() >= 65)
         return true;
      else
         return false;
   }
```

```
};

ref class Senior : IAge {
public:
   virtual int Age() {
      return 70;
   }
};

ref class Adult: IAge {
public:
   virtual int Age() {
      return 30;
   }
};

int main() {
   MyClass^ ageGuess = gcnew MyClass();
   Adult^ parent = gcnew Adult();
   Senior^ grandfather = gcnew Senior();

   if (ageGuess->isSenior<Adult^>(parent))
      Console::WriteLine("\"parent\" is a senior");
   else
      Console::WriteLine("\"parent\" is not a senior");

   if (ageGuess->isSenior<Senior^>(grandfather))
      Console::WriteLine("\"grandfather\" is a senior");
   else
      Console::WriteLine("\"grandfather\" is not a senior");
}
```

Output

```
"parent" is not a senior
"grandfather" is a senior
```

When a generic type parameter is used as a constraint, it is called a naked type constraint. Naked type constraints are useful when a member function with its own type parameter needs to constrain that parameter to the type parameter of the containing type.

In the following example, T is a naked type constraint in the context of the Add method.

Naked type constraints can also be used in generic class definitions. The usefulness of naked type constraints with generic classes is limited because the compiler can assume nothing about a naked type constraint except that it derives from **Object**. Use naked type constraints on generic classes in scenarios in which you wish to enforce an inheritance relationship between two type parameters.

```
// generics_constraints_6.cpp
// compile with: /clr /c
generic <class T>
ref struct List {
   generic <class U>
   where U : T
   void Add(List<U> items)  {}
};

generic <class A, class B, class C>
where A : C
ref struct SampleClass {};
```

See Also

Other Resources

Generics (Visual C++)

Consuming Generics from Other .NET Languages

Generics authored in one .NET language may be used in other .NET languages. Unlike templates, a generic in a compiled assembly still remains generic. Thus, one may instantiate the generic type in a different assembly and even in a different language than the assembly in which the generic type was defined.

Remarks

For more information, see:

- Introduction to Generics (C# Programming Guide)
- Generic Types in Visual Basic

Example

This example shows a generic class defined in C#.

```
// consuming_generics_from_other_NET_languages.cs
// compile with: /target:library
// a C# program
public class CircularList<ItemType> {
   class ListNode    {
      public ItemType m_item;
      public ListNode next;
      public ListNode(ItemType item) {
         m_item = item;
      }
   }

   ListNode first, last;

   public CircularList() {}

   public void Add(ItemType item) {
      ListNode newnode = new ListNode(item);
      if (first == null) {
         first = last = newnode;
         first.next = newnode;
         last.next = first;
      }
      else {
         newnode.next = first;
         first = newnode;
         last.next = first;
      }
   }

   public void Remove(ItemType item) {
      ListNode iter = first;
      if (first.m_item.Equals( item )) {
         first =
         last.next = first.next;
      }
      for ( ; iter != last ; iter = iter.next )
         if (iter.next.m_item.Equals( item )) {
              if (iter.next == last)
                  last = iter;
              iter.next = iter.next.next;
              return;
          }
   }

   public void PrintAll() {
      ListNode iter = first;
      do {
```

```
            System.Console.WriteLine( iter.m_item );
            iter = iter.next;
         } while (iter != last);
      }
   }
```

Example

This example consumes the assembly authored in C#.

```
// consuming_generics_from_other_NET_languages_2.cpp
// compile with: /clr
#using <consuming_generics_from_other_NET_languages.dll>
using namespace System;
class NativeClass {};
ref class MgdClass {};

int main() {
   CircularList<int>^ circ1 = gcnew CircularList<int>();
   CircularList<MgdClass^>^ circ2 = gcnew CircularList<MgdClass^>();

   for (int i = 0 ; i < 100 ; i += 10)
      circ1->Add(i);
   circ1->Remove(50);
   circ1->PrintAll();
}
```

Output

```
90
80
70
60
40
30
20
10
```

See Also

Other Resources

Generics (Visual C++)

Generics and Templates

Generics and templates are both language features that provide support for parameterized types. However, they are different and have different uses. This topic provides an overview of the many differences.

For more information, see Managed Templates and Templates Overview.

Comparing Templates and Generics

Key differences between generics and C++ templates:

- Generics are generic until the types are substituted for them at runtime. Templates are specialized at compile time so they are not still parameterized types at runtime
- The common language runtime specifically supports generics in MSIL. Because the runtime knows about generics, specific types can be substituted for generic types when referencing an assembly containing a generic type. Templates, in contrast, resolve into ordinary types at compile time and the resulting types may not be specialized in other assemblies.
- Generics specialized in two different assemblies with the same type arguments are the same type. Templates specialized in two different assemblies with the same type arguments are considered by the runtime to be different types.
- Generics are generated as a single piece of executable code which is used for all reference type arguments (this is not true for value types, which have a unique implementation per value type). The JIT compiler knows about generics and is able to optimize the code for the reference or value types that are used as type arguments. Templates generate separate runtime code for each specialization.
- Generics do not allow non-type template parameters, such as `template <int i> C {}`. Templates allow them.
- Generics do not allow explicit specialization (that is, a custom implementation of a template for a specific type). Templates do.
- Generics do not allow partial specialization (a custom implementation for a subset of the type arguments). Templates do.
- Generics do not allow the type parameter to be used as the base class for the generic type. Templates do.
- Generics do not allow type parameters to have default values. Templates do.
- Templates support template-template parameters (e.g. `template<template<class T> class X> class MyClass`), but generics do not.

Combining Templates and Generics

- The basic difference in generics has implications for building applications that combine templates and generics. For example, suppose you have a template class that you want to create a generic wrapper for to expose that template to other languages as a generic. You cannot have the generic take a type parameter that it then passes though to the template, since the template needs to have that type parameter at compile time, but the generic won't resolve the type parameter until runtime. Nesting a template inside a generic won't work either because there's no way to expand the templates at compile time for arbitrary generic types that could be instantiated at runtime.

Example

The following example shows a simple example of using templates and generics together. In this example, the template class passes its parameter through to the generic type. The reverse is not possible.

This idiom could be used when you want to build on an existing generic API with template code that is local to a Visual C++ assembly, or when you need to add an extra layer of parameterization to a generic type, to take advantage of certain features of templates not supported by generics.

```
// templates_and_generics.cpp
// compile with: /clr
using namespace System;

generic <class ItemType>
ref class MyGeneric {
   ItemType m_item;
```

```
public:
   MyGeneric(ItemType item) : m_item(item) {}
   void F() {
      Console::WriteLine("F");
   }
};

template <class T>
public ref class MyRef {
MyGeneric<T>^ ig;

public:
   MyRef(T t) {
      ig = gcnew MyGeneric<T>(t);
      ig->F();
    }
};

int main() {
   // instantiate the template
   MyRef<int>^ mref = gcnew MyRef<int>(11);
}
```

Output

```
F
```

See Also

Other Resources

Generics (Visual C++)

How to: Improve Performance with Generics

With Visual Studio 2005 and the .NET Framework 2.0, support has been added for programming with generics. With generics, you can create reusable code based on a type parameter. The actual type of the type parameter is deferred until called by client code. For more information on generics, see Generics (Visual C++).

This article will discuss how generics can help increase the performance of an application that uses collections.

Example

The .NET Framework comes with many collection classes in the System.Collections namespace. Most of these collections operate on objects of type System.Object. This allows collections to store any type, since all types in the .NET Framework, even value types, derive from **System.Object**. However, there are two drawbacks to this approach.

First, if the collection is storing value types such as integers, the value must be boxed before being added to the collection and unboxed when the value is retrieved from the collection. These are expensive operations.

Second, there is no way to control which types can be added to a collection. It is perfectly legal to add an integer and a string to the same collection, even though this is probably not what was intended. Therefore, in order for your code to be type safe, you have to check that the type retrieved from the collection really is what was expected.

The following code example shows the two main drawbacks of the .NET Framework collections before generics.

```
// perf_pre_generics.cpp
// compile with: /clr

using namespace System;
using namespace System::Collections;

int main()
{
    // This Stack can contain any type.
    Stack ^s = gcnew Stack();

    // Push an integer to the Stack.
    // A boxing operation is performed here.
    s->Push(7);

    // Push a String to the same Stack.
    // The Stack now contains two different data types.
    s->Push("Seven");

    // Pop the items off the Stack.
    // The item is returned as an Object, so a cast is
    // necessary to convert it to its proper type.
    while (s->Count > 0)
    {
        Object ^o = s->Pop();
        if (o->GetType() == Type::GetType("System.String"))
        {
            Console::WriteLine("Popped a String: {0}", (String ^)o);
        }
        else if (o->GetType() == Type::GetType("System.Int32"))
        {
            Console::WriteLine("Popped an int: {0}", (int)o);
        }
        else
        {
            Console::WriteLine("Popped an unknown type!");
        }
    }
}
```

Output

```
Popped a String: Seven
Popped an int: 7
```

The new System.Collections.Generic namespace contains many of the same collections found in the **System.Collections** namespace, but they have been modified to accept generic type parameters. This eliminates the two drawbacks of non-generic collections: the boxing and unboxing of value types and the inability to specify the types to be stored in the collections. Operations on the two collections are identical; they differ only in how they are instantiated.

Compare the example written above with this example that uses a generic Stack collection. On large collections that are frequently accessed, the performance of this example will be significantly greater than the preceding example.

```
// perf_post_generics.cpp
// compile with: /clr

#using <System.dll>

using namespace System;
using namespace System::Collections::Generic;

int main()
{
    // This Stack can only contain integers.
    Stack<int> ^s = gcnew Stack<int>();

    // Push an integer to the Stack.
    // A boxing operation is performed here.
    s->Push(7);
    s->Push(14);

    // You can no longer push a String to the same Stack.
    // This will result in compile time error C2664.
    //s->Push("Seven");

    // Pop an item off the Stack.
    // The item is returned as the type of the collection, so no
    // casting is necessary and no unboxing is performed for
    // value types.
    int i = s->Pop();
    Console::WriteLine(i);

    // You can no longer retrieve a String from the Stack.
    // This will result in compile time error C2440.
    //String ^str = s->Pop();
}
```

Output

```
14
```

See Also

Other Resources

Generics (Visual C++)

Implicit Boxing

The Visual C++ compiler now boxes value types to Object. This is possible because of a compiler-defined conversion to convert value types to **Object**.

Remarks

Boxing and unboxing enable value types to be treated as objects. Value types, including both struct types and built-in types such as int, can be converted to and from the type **Object**.

For more information, see:

- How to: Explicitly Request Boxing
- How to: Use gcnew to Create Value Types and Use Implicit Boxing
- How to: Unbox
- Standard Conversions and Implicit Boxing

Example

The following sample shows how explicit boxing was done with Managed Extensions for C++.

```
// vcmcppv2_explicit_boxing.cpp
// compile with: /clr:oldSyntax
using namespace System;

__gc class A {
public:
   void func(System::Object * o){Console::WriteLine("in A");}
};

__value class V {};

__gc __interface IFace {
   void func();
};

__value class V1 : public IFace   {
public:
   void func() {
      Console::WriteLine("Interface function");
   }
};

int main() {
   // example 1 explicit boxing
   Int32 __box* bi = __box(1);
   Console::WriteLine(bi);

   // example 2 calling a member with explicit boxing
  Int32 n = 10;
   Console::WriteLine(S"xx = {0}", __box(n)->ToString());

   // example 3 explicit boxing for function calls
   A * a = new A;
   a->func(__box(n));

   // example 4  explicit boxing for WriteLine function call
   V v;
   Console::WriteLine(S"Class {0} passed using implicit boxing", __box(v)->ToString());

   // example 5 casting to a base with explicit boxing
   V1 v1;
   IFace * iface = __box(v1);
   iface->func();
```

```
}
```

Output

```
1
xx = 10
in A
Class V passed using implicit boxing
Interface function
```

The following sample shows how implicit boxing works.

```
// vcmcppv2_explicit_boxing2.cpp
// compile with: /clr
using namespace System;

ref class A {
public:
   void func(System::Object^ o){Console::WriteLine("in A");}
};

value class V {};

interface struct IFace {
   void func();
};

value class V1 : public IFace {
public:
   virtual void func() {
      Console::WriteLine("Interface function");
   }
};

value struct V2 {
   // conversion operator to System::Object
   static operator System::Object^(V2 v2) {
      Console::WriteLine("operator System::Object^");
      return (V2^)v2;
   }
};

void func1(System::Object^){Console::WriteLine("in void func1(System::Object^)");}
void func1(V2^){Console::WriteLine("in func1(V2^)");}

void func2(System::ValueType^){Console::WriteLine("in func2(System::ValueType^)");}
void func2(System::Object^){Console::WriteLine("in func2(System::Object^)");}

int main() {
   // example 1 simple implicit boxing
   Int32^ bi = 1;
   Console::WriteLine(bi);

   // example 2 calling a member with implicit boxing
   Int32 n = 10;
   Console::WriteLine("xx = {0}", n.ToString());

   // example 3 implicit boxing for function calls
   A^ a = gcnew A;
   a->func(n);

   // example 4 implicit boxing for WriteLine function call
   V v;
   Console::WriteLine("Class {0} passed using implicit boxing", v);
   Console::WriteLine("Class {0} passed with forced boxing", (V^)(v));   // force boxing

   // example 5 casting to a base with implicit boxing
   V1 v1;
```

```
   IFace ^ iface = v1;
   iface->func();

   // example 6 user-defined conversion preferred over implicit boxing for function-call pa
rameter matching
   V2 v2;
   func1(v2);   // user defined conversion from V2 to System::Object preferred over implici
t boxing
                // Will call void func1(System::Object^);

   func2(v2);   // OK: Calls "static V2::operator System::Object^(V2 v2)"
   func2((V2^)v2);   // Using explicit boxing: calls func2(System::ValueType^)
}
```

Output

```
1
xx = 10
in A
Class V passed using implicit boxing
Class V passed with forced boxing
Interface function
in func1(V2^)
in func2(System::ValueType^)
in func2(System::ValueType^)
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

How to: Explicitly Request Boxing

You can explicitly request boxing by assigning a variable to a variable of type **Object**.

Example

```
// vcmcppv2_explicit_boxing3.cpp
// compile with: /clr
using namespace System;

void f(int i) {
   Console::WriteLine("f(int i)");
}

void f(Object ^o) {
   Console::WriteLine("f(Object^ o)");
}

int main() {
   float i = 22.22;
   i = 323.5654;
   Object ^ O = i;   // forces i to be boxed
   f(i);
   f( (Object^)i );  // boxes i
}
```

Output

```
f(int i)
f(Object^ o)
```

See Also

Reference

Implicit Boxing

How to: Use gcnew to Create Value Types and Use Implicit Boxing

Using gcnew on a value type will create a boxed value type, which can then be placed on the managed, garbage-collected heap.

Example

```
// vcmcppv2_explicit_boxing4.cpp
// compile with: /clr
public value class V {
public:
   int m_i;
   V(int i) : m_i(i) {}
};

public ref struct TC {
   void do_test(V^ v) {
      if (v != nullptr)
         ;
      else
         ;
   }
};

int main() {
   V^ v = gcnew V(42);
   TC^ tc = gcnew TC;
   tc->do_test(v);
}
```

See Also

Reference

Implicit Boxing

How to: Unbox

Shows how to unbox and modify a value.

Example

```
// vcmcppv2_unboxing.cpp
// compile with: /clr
using namespace System;

int main() {
   int ^ i = gcnew int(13);
   int j;
   Console::WriteLine(*i);   // unboxing
   *i = 14;   // unboxing and assignment
   Console::WriteLine(*i);
   j = safe_cast<int>(i);   // unboxing and assignment
   Console::WriteLine(j);
}
```

Output

```
13
14
14
```

See Also

Reference

Implicit Boxing

Standard Conversions and Implicit Boxing

A standard conversion will be chosen by the compiler over a conversion that requires boxing.

Example

```
// clr_implicit_boxing_Std_conversion.cpp
// compile with: /clr
int f3(int ^ i) {   // requires boxing
   return 1;
}

int f3(char c) {   // no boxing required, standard conversion
   return 2;
}

int main() {
   int i = 5;
   System::Console::WriteLine(f3(i));
}
```

Output

```
2
```

See Also

Reference

Implicit Boxing

interface class

Declares a managed interface. For information on native interfaces, see __interface.

```
interface_access interface class name : inherit_access base_interface {};
interface_access interface struct name : inherit_access base_interface {};
```

Parameters

base_interface(optional)

A base interface for interface *name*.

inherit_access

The accessibility of *base_interface*. The only permitted accessibility for a base interface is **public** (**public** is the default).

interface_access

The accessibility of an interface outside the assembly. Possible values are **public** and **private**. **private** is the default. Nested interfaces cannot have an *interface_access* specifier.

name

The name of the interface.

Remarks

interface struct is equivalent to **interface class**.

An interface can contain declarations for functions, events, and properties. All interface members have public accessibility. An interface can also contain static data members, functions, events, and properties, and these static members must be defined in the interface.

An interface defines how a class may be implemented. An interface is not a class and classes can only implement interfaces. When a class defines a function declared in an interface, the function is implemented, not overridden. Therefore, name lookup does not include interface members.

A class or struct that derives from an interface must implement all members of the interface. When implementing interface *name* you must also implement the interfaces in the *base_interface* list.

For more information, see:

- Interface Static Constructor
- Generic Interfaces (Visual C++)

For information on other CLR types, see Classes and Structs.

You can detect at compile time if a type is an interface with **__is_interface_class(*type*)**. For more information, see Compiler Support for Type Traits.

In the development environment, you can get F1 help on these keywords by highlighting the keyword, (**interface class**, for example) and pressing F1.

Example

```
// mcppv2_interface_class.cpp
// compile with: /clr
using namespace System;

public delegate void ClickEventHandler(int, double);

// define interface with nested interface
public interface class Interface_A {
   void Function_1();

   interface class Interface_Nested_A {
      void Function_2();
```

```
   };
};

// interface with a base interface
public interface class Interface_B : Interface_A {
   property int Property_Block;
   event ClickEventHandler^ OnClick;
   static void Function_3() { Console::WriteLine("in Function_3"); }
};

// implement nested interface
public ref class MyClass : public Interface_A::Interface_Nested_A {
public:
   virtual void Function_2() { Console::WriteLine("in Function_2"); }
};

// implement interface and base interface
public ref class MyClass2 : public Interface_B {
private:
   int MyInt;

public:
   // implement non-static function
   virtual void Function_1() { Console::WriteLine("in Function_1"); }

   // implement property
   property int Property_Block {
      virtual int get() { return MyInt; }
      virtual void set(int value) { MyInt = value; }
   }
   // implement event
   virtual event ClickEventHandler^ OnClick;

   void FireEvents() {
      OnClick(7, 3.14159);
   }
};

// class that defines method called when event occurs
ref class EventReceiver {
public:
   void OnMyClick(int i, double d) {
      Console::WriteLine("OnClick: {0}, {1}", i, d);
   }
};

int main() {
   // call static function in an interface
   Interface_B::Function_3();

   // instantiate class that implements nested interface
   MyClass ^ x = gcnew MyClass;
   x->Function_2();

   // instantiate class that implements interface with base interface
   MyClass2 ^ y = gcnew MyClass2;
   y->Function_1();
   y->Property_Block = 8;
   Console::WriteLine(y->Property_Block);

   EventReceiver^ MyEventReceiver = gcnew EventReceiver();

   // hook handler to event
   y->OnClick += gcnew ClickEventHandler(MyEventReceiver, &EventReceiver::OnMyClick);

   // invoke events
   y->FireEvents();
```

```
   // unhook handler to event
   y->OnClick -= gcnew ClickEventHandler(MyEventReceiver, &EventReceiver::OnMyClick);

   // call implemented function via interface handle
   Interface_A^ hi = gcnew MyClass2();
   hi->Function_1();
}
```

Output

```
in Function_3
in Function_2
in Function_1
8
OnClick: 7, 3.14159
in Function_1
```

The following sample shows two ways to implement functions with the same signature declared in multiple interfaces and where those interfaces are used by a class.

```
// mcppv2_interface_class_2.cpp
// compile with: /clr /c
interface class I {
   void Test();
   void Test2();
};

interface class J : I {
   void Test();
   void Test2();
};

ref struct R : I, J {
   // satisfies the requirement to implement Test in both interfaces
   virtual void Test() {}

   // implement both interface functions with explicit overrides
   virtual void A() = I::Test2 {}
   virtual void B() = J::Test2 {}
};
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

How to: Define an Interface Static Constructor

An interface can have a static constructor, which can be used to initialize static data members. A static constructor will be called at most once, and will be called before the first time a static interface member is accessed.

For more information on static constructors, see How to: Define Static Constructors in a Class or Struct.

Example

```
// mcppv2_interface_class2.cpp
// compile with: /clr
using namespace System;

interface struct MyInterface {
   static int i;
   static void Test() {
      Console::WriteLine(i);
   }

   static MyInterface() {
      Console::WriteLine("in MyInterface static constructor");
      i = 99;
   }
};

ref class MyClass : public MyInterface {};

int main() {
   MyInterface::Test();
   MyClass::MyInterface::Test();

   MyInterface ^ mi = gcnew MyClass;
   mi->Test();
}
```

Output

```
in MyInterface static constructor
99
99
99
```

See Also

Reference

interface class

interior_ptr

Declares a pointer to inside a reference type, but not to the object itself. An interior pointer can point to a reference handle, value type, boxed type handle, member of a managed type, or to an element of a managed array.

```
cli::interior_ptr<cv_qualifier type> var = &initializer;
```

Parameters

cv_qualifier

const or **volatile** qualifiers.

initializer

A member of a reference type, element of a managed array, or any other object that you can assign to a native pointer.

type

The type of *initializer*.

var

The name of the **interior_ptr** variable.

Remarks

A native pointer is not able to track an item as its location changes on the managed heap, which results from the garbage collector moving instances of an object. In order for a pointer to correctly refer to the instance, the runtime needs to update the pointer to the newly positioned object.

An **interior_ptr** represents a superset of the functionality of a native pointer. Therefore, anything that can be assigned to a native pointer can also be assigned to an **interior_ptr**. An interior pointer is permitted to perform the same set of operations as native pointers, including comparison and pointer arithmetic.

An interior pointer can only be declared on the stack. An interior pointer cannot be declared as a member of a class.

Since interior pointers exist only on the stack, taking the address of an interior pointer yields an unmanaged pointer.

interior_ptr has an implicit conversion to **bool**, which allows for its use in conditional statements.

For information on how to declare an interior pointer that points into an object that cannot be moved on the garbage-collected heap, see pin_ptr.

interior_ptr is in the cli namespace. See cli Namespace for more information.

For more information on interior pointers, see

- How to: Declare and Use Interior Pointers and Managed Arrays
- How to: Declare Value Types with the interior_ptr Keyword
- How to: Overload Functions with Interior Pointers and Native Pointers
- How to: Declare Interior Pointers with the const Keyword

Example

The following sample shows how to declare and use an interior pointer into a reference type.

```
// interior_ptr.cpp
// compile with: /clr
using namespace System;

ref class MyClass {
public:
   int data;
};
```

```
int main() {
   MyClass ^ h_MyClass = gcnew MyClass;
   h_MyClass->data = 1;
   Console::WriteLine(h_MyClass->data);

   interior_ptr<int> p = &(h_MyClass->data);
   *p = 2;
   Console::WriteLine(h_MyClass->data);

   // alternatively
   interior_ptr<MyClass ^> p2 = &h_MyClass;
   (*p2)->data = 3;
   Console::WriteLine((*p2)->data);
}
```

Output

```
1
2
3
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

How to: Declare and Use Interior Pointers and Managed Arrays

The following sample shows how you can declare and use an interior pointer to an array.

Example

```
// interior_ptr_arrays.cpp
// compile with: /clr
#define SIZE 10

int main() {
   // declare the array
   array<int>^ arr = gcnew array<int>(SIZE);

   // initialize the array
   for (int i = 0 ; i < SIZE ; i++)
      arr[i] = i + 1;

   // create an interior pointer into the array
   interior_ptr<int> ipi = &arr[0];

   System::Console::WriteLine("1st element in arr holds: {0}", arr[0]);
   System::Console::WriteLine("ipi points to memory address whose value is: {0}", *ipi);

   ipi++;
   System::Console::WriteLine("after incrementing ipi, it points to memory address whose va
lue is: {0}", *ipi);
}
```

Output

```
1st element in arr holds: 1
ipi points to memory address whose value is: 1
after incrementing ipi, it points to memory address whose value is: 2
```

See Also

Reference

interior_ptr

How to: Declare Value Types with the interior_ptr Keyword

An **interior_ptr** can be used with a value type.

Example

The following sample shows how to use an **interior_ptr** with a value type.

```
// interior_ptr_value_types.cpp
// compile with: /clr
value struct V {
   V(int i) : data(i){}
   int data;
};

int main() {
   V v(1);
   System::Console::WriteLine(v.data);

   // pointing to a value type
   interior_ptr<V> pv = &v;
   pv->data = 2;

   System::Console::WriteLine(v.data);
   System::Console::WriteLine(pv->data);

   // pointing into a value type
   interior_ptr<int> pi = &v.data;
   *pi = 3;
   System::Console::WriteLine(*pi);
   System::Console::WriteLine(v.data);
   System::Console::WriteLine(pv->data);
}
```

Output

```
1
2
2
3
3
3
```

Example

In a value type, the `this` pointer evaluates to an interior_ptr.

In the body of a non-static member-function of a value type `V`, `this` is an expression of type `interior_ptr<V>` whose value is the address of the object for which the function is called.

```
// interior_ptr_value_types_this.cpp
// compile with: /clr /LD
value struct V {
   int data;
   void f() {
      interior_ptr<V> pv1 = this;
      // V* pv2 = this;   error
   }
};
```

Example

The following sample shows how to use the address-of operator with static members.

The address of a static Visual C++ type member yields a native pointer. The address of a static value type member is a

managed pointer because value type member is allocated on the runtime heap and can be moved by the garbage collector.

```
// interior_ptr_value_static.cpp
// compile with: /clr
using namespace System;
value struct V { int i; };

ref struct G {
   static V v = {22};
   static int i = 23;
   static String^ pS = "hello";
};

int main() {
   interior_ptr<int> p1 = &G::v.i;
   Console::WriteLine(*p1);

   interior_ptr<int> p2 = &G::i;
   Console::WriteLine(*p2);

   interior_ptr<String^> p3 = &G::pS;
   Console::WriteLine(*p3);
}
```

Output

```
22
23
hello
```

See Also

Reference

interior_ptr

How to: Overload Functions with Interior Pointers and Native Pointers

Functions can be overloaded depending on whether the parameter type is an interior pointer or a native pointer.

Example

```
// interior_ptr_overload.cpp
// compile with: /clr
using namespace System;

// C++ class
struct S {
   int i;
};

// managed class
ref struct G {
   int i;
};

// can update unmanaged storage
void f( int* pi ) {
   *pi = 10;
   Console::WriteLine("in f( int* pi )");
}

// can update managed storage
void f( interior_ptr<int> pi ) {
   *pi = 10;
   Console::WriteLine("in f( interior_ptr<int> pi )");
}

int main() {
   S *pS = new S;   // C++ heap
   G ^pG = gcnew G;   // common language runtime heap
   f( &pS->i );
   f( &pG->i );
};
```

Output

```
in f( int* pi )
in f( interior_ptr<int> pi )
```

See Also

Reference

interior_ptr

How to: Declare Interior Pointers with the const Keyword

The following sample shows how to use **const** in the declaration of an interior pointer.

Example

```
// interior_ptr_const.cpp
// compile with: /clr
using namespace System;
value struct V {
   int i;
};

ref struct G {
   V v;
   String ^ msg;
};

interior_ptr<int> f( interior_ptr<V> pv ) {
   return &(pv->i);
}

int main() {
   int n = -1;
   int o = -1;
   interior_ptr<int> pn1 = &n;
   *pn1 = 50;

   V v;
   v.i = 101;
   V * npV = &v;   // ok: &v yields a pointer to the native heap

   interior_ptr<int> pn2 = &n;
   interior_ptr<V> pV = &(v);
   pn2 = f(pV);
   *pn2 = 50;

   G ^pG = gcnew G;
   pV = &(pG->v);   // ok: pV is an interior pointer

   interior_ptr<int const> pn3 = &n;
   // *pn3 = 5;   error because pn3 cannot be dereferenced and changed
   pn3 = &o;   // OK, can change the memory location

   interior_ptr<int> const pn4 = &n;
   *pn4 = 5;   // OK because you can dereference and change pn4
   // pn4 = &o;   error cannot change the memory location

   const interior_ptr<const int> pn5 = &n;
   // *pn5 = 5;   error cannot dereference and change pn5
   // pn5 = &o;   error cannot change the memory location

   const G ^ h_G = gcnew G;   // object is const, cannot modify any members of h_G or call
any non-const methods
   // h_G->msg = "test";   error h_G is const
   interior_ptr<String^ const> int_ptr_G = &(h_G->msg);

   G ^ const h_G2 = gcnew G;   // interior pointers to this obejct cannot be dereferenced a
nd changed
   h_G2->msg = "test";
   interior_ptr<String^ const> int_ptr_G2 = &(h_G->msg);
};
```

See Also

Reference

interior_ptr

literal (Visual C++)

A variable (data member) marked as **literal** in a **/clr** compilation is the native equivalent of a **static const** variable.

Remarks

A data member marked as **literal** must be initialized when declared and the value must be a constant integral, enum, or string type. Conversion from the type of the initialization expression to the type of the static const data-member must not require a user-defined conversion.

No memory is allocated for the literal field at runtime; the compiler only inserts its value in the metadata for the class.

A variable marked **static const** will not be available in metadata to other compilers.

For more information, see Static (C++) and const (C++).

literal is a context-sensitive keyword. See Context-Sensitive Keywords for more information.

Example

This example shows that a **literal** variable implies **static**.

```
// mcppv2_literal.cpp
// compile with: /clr
ref struct X {
   literal int i = 4;
};

int main() {
   int value = X::i;
}
```

The following sample shows the affect of literal in metadata:

```
// mcppv2_literal2.cpp
// compile with: /clr /LD
public ref struct A {
   literal int lit = 0;
   static const int sc = 1;
};
```

Notice the difference in the metadata for `sc` and `lit`: the `modopt` directive is applied to `sc`, meaning it can be ignored by other compilers.

```
.field public static int32 modopt([mscorlib]System.Runtime.CompilerServices.IsConst) sc = i
nt32(0x0000000A)
```

```
.field public static literal int32 lit = int32(0x0000000A)
```

The following sample, authored in C#, references the metadata created in the previous sample and shows the affect of **literal** and **static const** variables:

```
// mcppv2_literal3.cs
// compile with: /reference:mcppv2_literal2.dll
// A C# program
class B {
   public static void Main() {
      // OK
      System.Console.WriteLine(A.lit);
      System.Console.WriteLine(A.sc);

      // C# does not enforce C++ const
      A.sc = 9;
```

```
        System.Console.WriteLine(A.sc);

        // C# enforces const for a literal
        A.lit = 9;   // CS0131

        // you can assign a C++ literal variable to a C# const variable
        const int i = A.lit;
        System.Console.WriteLine(i);

        // but you cannot assign a C++ static const variable
        // to a C# const variable
        const int j = A.sc;   // CS0133
        System.Console.WriteLine(j);
    }
}
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

Managed Templates

Class templates allow you to define a type and create one or more instantiations of the type using different type parameters. For more information on standard C++ class templates, see Class Templates.

You can also create class templates from value or reference types. See Classes and Structs (Managed) for more information about creating value or reference types. However, there are some limitations to creating class templates from managed types:

- It is possible to instantiate a generic type with a managed type template parameter, but you cannot instantiate a managed template with a generic type template parameter. This is because generic types are resolved at runtime. (For more information, see Generics and Templates.)

```
// managed_templates.cpp
// compile with: /clr /c
generic<class T>
ref class R;

template<class T>
ref class Z {
   // instantiate a generic with a template parameter
   R<T>^ r;    // OK
};

generic<class T>
ref class R {
   // cannot instantiate a template with a generic parameter
   Z<T>^ z;   // C3231
};
```

- A generic type or function may not be nested in a managed template.

```
// managed_templates_2.cpp
// compile with: /clr /c
template<class T> public ref class R {
   generic<class T> ref class W {};   // C2959
};
```

- You cannot access templates defined in a referenced assembly with C++ language syntax, but can use reflection. If a template is not instantiated, it's not emitted in the metadata. If a template is instantiated, only referenced member functions will appear in metadata.

```
// managed_templates_3.cpp
// compile with: /clr
// will not appear in metadata
template<class T> public ref class A {};

// will appear in metadata as a specialized type
template<class T> public ref class R {
public:
   // Test is referenced, will appear in metadata
   void Test() {}

   // Test2 is not referenced, will not appear in metadata
   void Test2() {}
};

// will appear in metadata
```

```
generic<class T> public ref class G { };

public ref class S { };

int main() {
   R<int>^ r = gcnew R<int>;
   r->Test();
}
```

- It is also possible to vary the managed modifier of a class from class template to partial specialization and explicit specialization.

```
// managed_templates_4.cpp
// compile with: /clr /c
// class template
// ref class
template <class T>
ref class A {};

// partial template specialization
// value type
template <class T>
value class A <T *> {};

// partial template specialization
// interface
template <class T>
interface class A<T%> {};

// explicit template specialization
// native class
template <>
class A <int> {};
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

new (new slot in vtable)

In a **/clr** compilation, **new** indicates that a virtual member will get a new slot in the vtable; that the function does not override a base class method.

Remarks

new causes the newslot modifier to be added to the IL for the function. For more information about newslot, see:

- Type Members
- MethodInfo.GetBaseDefinition Method
- MethodAttributes Enumeration

The following sample shows the effect of **new**.

Example

```
// newslot.cpp
// compile with: /clr
ref class C {
public:
   virtual void f() {
      System::Console::WriteLine("C::f() called");
   }

   virtual void g() {
      System::Console::WriteLine("C::g() called");
   }
};

ref class D : public C {
public:
   virtual void f() new {
      System::Console::WriteLine("D::f() called");
   }

   virtual void g() override {
      System::Console::WriteLine("D::g() called");
   }
};

ref class E : public D {
public:
   virtual void f() override {
      System::Console::WriteLine("E::f() called");
   }
};

int main() {
   D^ d = gcnew D;
   C^ c = gcnew D;

   c->f();   // calls C::f
   d->f();   // calls D::f

   c->g();   // calls D::g
   d->g();   // calls D::g

   D ^ e = gcnew E;
   e->f();   // calls E::f
}
```

Output

```
C::f() called
```

```
D::f() called
D::g() called
D::g() called
E::f() called
```

Requirements

Compiler option: **/clr**

See Also

Reference

Override Specifiers

nullptr

nullptr indicates that an object handle, interior pointer, or native pointer type does not point to an object. **nullptr** is only valid when compiling with /clr (Common Language Runtime Compilation).

You cannot initialize handles to zero; only **nullptr** can be used. Assignment of constant 0 to an object handle will produce a boxed **Int32** and a cast to **Object^**.

nullptr can be used in the initialization of the following pointer types:

- Native pointer:
- Managed handle:
- Managed interior pointer:

nullptr can be used for reference checking before using a pointer.

nullptr can be used anywhere a handle or a native pointer can be used. **nullptr** can also be used as an argument for functions.

```
// mcpp_nullptr.cpp
// compile with: /clr
value class V {};
ref class G {};
void f(System::Object ^) {}

int main() {
   int *pN = nullptr;
   G ^pG = nullptr;
   V ^pV1 = nullptr;
   interior_ptr<V> pV2 = nullptr;

   if (pN == nullptr) {}
   if (pG == nullptr) {}
   if (pV1 == nullptr) {}
   if (pV2 == nullptr) {}

   f(nullptr);   // calls f(System::Object ^)
}
```

nullptr is equivalent to **Nothing** in Visual Basic and **null** in C#. Function calls among languages that use these null mechanisms for error checking should be interpreted correctly.

nullptr is not a type and is not supported for use with:

- sizeof
- typeid
- **typeobj**
- **throw nullptr** (although **throw (Object ^)nullptr;** will work)

Example

The following sample shows that **nullptr** and zero can be used interchangeably on native pointers.

```
// mcpp_nullptr_1.cpp
// compile with: /clr
class MyClass {
public:
   int i;
};

int main() {
   MyClass * pMyClass = nullptr;
   if ( pMyClass == nullptr)
```

```
      System::Console::WriteLine("pMyClass == nullptr");

   if ( pMyClass == 0)
      System::Console::WriteLine("pMyClass == 0");

   pMyClass = 0;
   if ( pMyClass == nullptr)
      System::Console::WriteLine("pMyClass == nullptr");

   if ( pMyClass == 0)
      System::Console::WriteLine("pMyClass == 0");
}
```

Output

```
pMyClass == nullptr
pMyClass == 0
pMyClass == nullptr
pMyClass == 0
```

The following example shows that **nullptr** is interpreted as a handle to any type or a native pointer to any type. In case of function overloading with handles to different types, an ambiguity error will be generated. The **nullptr** would have to be explicitly cast to a type.

```
// mcpp_nullptr_2.cpp
// compile with: /clr /LD
void f(int *){}
void f(int ^){}

void f_null() {
   f(nullptr);   // C2668
   // try one of the following lines instead
   f((int *) nullptr);
   f((int ^) nullptr);
}
```

The following sample shows that casting **nullptr** is allowed and returns a pointer or handle to the cast type that contains the **nullptr** value.

```
// mcpp_nullptr_3.cpp
// compile with: /clr /LD
using namespace System;
template <typename T>
void f(T) {}   // C2036 cannot deduce template type because nullptr can be any type

int main() {
   f((Object ^) nullptr);   // T = Object^, call f(Object ^)

   // Delete the following line to resolve.
   f(nullptr);

   f(0);   // T = int, call f(int)
}
```

The following sample shows that **nullptr** can be used as a function parameter.

```
// mcpp_nullptr_4.cpp
// compile with: /clr
using namespace System;
void f(Object ^ x) {
   Console::WriteLine("test");
}

int main() {
   f(nullptr);
```

```
}
```

Output

```
test
```

The following sample shows that when handles are declared and not explicitly initialized, they are default initialized to **nullptr**.

```
// mcpp_nullptr_5.cpp
// compile with: /clr
using namespace System;
ref class MyClass {
public:
   void Test() {
      MyClass ^pMyClass;   // gc type
      if (pMyClass == nullptr)
         Console::WriteLine("NULL");
   }
};

int main() {
   MyClass ^ x = gcnew MyClass();
   x -> Test();
}
```

Output

```
NULL
```

The following sample shows that **nullptr** can be assigned to a native pointer when compiling with **/clr**.

```
// mcpp_nullptr_6.cpp
// compile with: /clr
int main() {
   int * i = 0;
   int * j = nullptr;
}
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

override

override indicates that a member of a managed type must override a base class or a base interface member. If there is no member to override, the compiler will generate an error.

override is also valid when compiling for native targets (without **/clr**). See Override Specifiers and Native Compilations for more information.

override is a context-sensitive keyword. See Context-Sensitive Keywords for more information.

Example

```
// override_keyword.cpp
// compile with: /clr /c
ref struct I1 {
   virtual void f();
};

ref struct X : public I1 {
   virtual void f() override {}
};
```

This sample shows that **override** can also be used in native compilations.

```
// override_keyword_2.cpp
// compile with: /c
struct I1 {
   virtual void f();
};

struct X : public I1 {
   virtual void f() override {}
};
```

Requirements

Compiler option: **/clr**

See Also

Reference

Override Specifiers

Override Specifiers

You can modify how inherited types and members of inherited types behave in derived types. For more information, see:

- abstract
- new (new slot in vtable)
- override
- sealed
- Override Specifiers and Native Compilations

abstract and **sealed** are also valid on type declarations, where they do not act as override specifiers.

For information about explicitly overriding base class functions, see Explicit Overrides.

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

How to: Declare Override Specifiers in Native Compilations

sealed, abstract, and override are also available in compilations that do not use /clr.

Example

The following sample shows that **sealed** is valid in native compilations:

```
// sealed_native_keyword.cpp
#include <stdio.h>
__interface I1 {
   virtual void f();
   virtual void g();
};

class X : public I1 {
public:
   virtual void g() sealed {}
};

class Y : public X {
public:

   // the following override generates a compiler error
   virtual void g() {}   // C3248 X::g is sealed!
};
```

Example

The following sample shows that **override** is valid in native compilations:

```
// override_native_keyword.cpp
#include <stdio.h>
__interface I1 {
   virtual void f();
};

class X : public I1 {
public:
   virtual void f() override {}   // OK
   virtual void g() override {}   // C3668 I1::g does not exist
};
```

Example

The following sample shows that **abstract** is valid in native compilations:

```
// abstract_native_keyword.cpp
class X abstract {};

int main() {
   X * MyX = new X;   // C3622 cannot instantiate abstract class
}
```

See Also

Reference

Override Specifiers

pin_ptr

pin_ptr lets you declare a pinning pointer. A pinning pointer is an interior pointer that prevents the object pointed into from moving on the garbage-collected heap (the value of a pinning pointer is not changed by the common language runtime). This is necessary when passing the address of a managed class to an unmanaged function because the address will not change unexpectedly during resolution of the unmanaged function call.

```
[cli::]pin_ptr<cv_qualifier type> var = &initializer;
```

Parameters

where:

cv_qualifier

const or **volatile** qualifiers. A pinning pointer is **volatile** by default. It is redundant but not an error to declare a pinning pointer **volatile**.

initializer

A member of a reference type, element of a managed array, or any other object that you can assign to a native pointer.

type

The type of *initializer*.

var

The name of the **pin_ptr** variable.

Remarks

An object or sub-object of a managed class can be pinned, in which case the common language runtime will not move it during garbage collection. The principal use of this is to pass a pointer to managed data as an actual parameter of an unmanaged function call. During a collection cycle, the runtime will inspect the metadata created for the pinning pointer and will not move the item it points to.

Pinning a sub-object defined in a managed object has the effect of pinning the entire object.

If the pinning pointer is reassigned to point to a new value, the previous instance pointed to is no longer considered pinned.

When a pinning pointer goes out of scope, the object is no longer considered pinned, unless there are other pinning pointers pointing to or into the object. You do not have to explicitly unpin a pointer.

If no pinning pointers point to the object (all pinning pointers went out of scope, were reassigned to point to other objects, or were assigned nullptr), the object is guaranteed not to be pinned.

A pinning pointer can point to a reference handle, value type or boxed type handle, member of a managed type or to an element of a managed array. It cannot point to a reference type.

Note
Taking the address of a **pin_ptr** that points to a native object results in undefined behavior.

Pinning pointers can only be declared as non-static local variables on the stack.

Pinning pointers cannot be used as:

- function parameters
- the return type of a function
- a member of a class
- the target type of a cast.

A **pin_ptr** represents a superset of the functionality of a native pointer. Therefore, anything that can be assigned to a native

pointer can also be assigned to a **pin_ptr**. An interior pointer is permitted to perform the same set of operations as native pointers, including comparison and pointer arithmetic.

pin_ptr is in the cli namespace. See cli Namespace for more information.

For information about interior pointers, see interior_ptr.

For information about pinning pointers, see

- Pinning Pointers and Arrays
- Scope of Pinning Pointers
- Pinning Pointers and Value Types

Example

```
// pin_ptr_1.cpp
// compile with: /clr
using namespace System;
#define SIZE 10

#pragma unmanaged
// native function that initializes an array
void native_function(int* p) {
   for(int i = 0 ; i < 10 ; i++)
    p[i] = i;
}
#pragma managed

public ref class A {
private:
   array<int>^ arr;   // CLR integer array

public:
   A() {
      arr = gcnew array<int>(SIZE);
   }

   void load() {
   pin_ptr<int> p = &arr[0];   // pin pointer to first element in arr
   int* np = p;   // pointer to the first element in arr
   native_function(np);   // pass pointer to native function
   }

   int sum() {
      int total = 0;
      for (int i = 0 ; i < SIZE ; i++)
         total += arr[i];
      return total;
   }
};

int main() {
   A^ a = gcnew A;
   a->load();   // initialize managed array using the native function
   Console::WriteLine(a->sum());
}
```

Output

```
45
```

This sample shows that an interior pointer can be converted to a pinning pointer and that the return type of the & operator is an interior pointer when the operand is on the managed heap.

```
// pin_ptr_2.cpp
// compile with: /clr
using namespace System;
```

```
ref struct G {
   G() : i(1) {}
   int i;
};

ref struct H {
   H() : j(2) {}
   int j;
};

int main() {
   G ^ g = gcnew G;   // g is a whole reference object pointer
   H ^ h = gcnew H;

   interior_ptr<int> l = &(g->i);   // l is interior pointer

   pin_ptr<int> k = &(h->j);   // k is a pinning interior pointer

   k = l;   // ok
   Console::WriteLine(*k);
};
```

Output

```
1
```

This sample shows that a pinning pointer can be cast to another type:

```
// pin_ptr_3.cpp
// compile with: /clr
using namespace System;

ref class ManagedType {
public:
   int i;
};

int main() {
   ManagedType ^mt = gcnew ManagedType;
   pin_ptr< int > pt = &mt->i;
   *pt = 8;
   Console::WriteLine(mt->i);

   char *pc = ( char* ) pt;
   *pc = 255;
   Console::WriteLine(mt->i);
}
```

Output

```
8
255
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

How to: Pin Pointers and Arrays

Pinning a sub-object defined in a managed object has the effect of pinning the entire object. For example, if any element of an array is pinned, then the whole array is also pinned. There are no extensions to the language for declaring a pinned array. To pin an array, declare a pinning pointer to its element type, and pin one of its elements.

Example

```
// pin_ptr_array.cpp
// compile with: /clr
#include <stdio.h>
using namespace System;

int main() {
   array<Byte>^ arr = gcnew array<Byte>(4);
   arr[0] = 'C';
   arr[1] = '+';
   arr[2] = '+';
   arr[3] = '\0';
   pin_ptr<Byte> p = &arr[1];   // entire array is now pinned
   unsigned char * cp = p;

   printf_s("%s\n", cp); // bytes pointed at by cp
                         // will not move during call
}
```

Output

```
++
```

See Also

Reference

pin_ptr

How to: Declare Pinning Pointers and Value Types

A value type can be implicitly boxed. You can then declare a pinning pointer to the value type object itself and use a **pin_ptr** to the boxed value type.

Example

```
// pin_ptr_value.cpp
// compile with: /clr
value struct V {
   int i;
};

int main() {
   V ^ v = gcnew V;   // imnplicit boxing
   v->i=8;
   System::Console::WriteLine(v->i);
   pin_ptr<V> mv = &*v;
   mv->i = 7;
   System::Console::WriteLine(v->i);
   System::Console::WriteLine(mv->i);
}
```

Output

```
8
7
7
```

See Also

Reference

pin_ptr

How to: Define the Scope of Pinning Pointers

An object is pinned only while a pin_ptr points to it. The object is no longer pinned when its pinning pointer goes out of scope, or is set to nullptr. After the pin_ptr goes out of scope, the object that was pinned can be moved in the heap by the garbage collector. Any native pointers that still point to the object will not be updated, and dereferencing one of them could raise an unrecoverable exception.

Example

```
// pin_ptr_scope.cpp
// compile with: /clr
ref class G {
public:
   int i;
};

int * f(G ^ hG) {
   pin_ptr<int> ppi = &(hG->i);
   int * i = ppi;
   return i;
}

int main() {
   G ^ MyG = gcnew G;
   int * i;
   i = f(MyG);   // bad!  object that i points to is no longer pinned
}
```

See Also

Reference

pin_ptr

property

Declare a managed property.

```
modifier property type property_name;   // property data member
modifier property type property_name {   // property block
      modifier void set(type);
      modifier type get();
}
modifier property type property_name[,] {
      modifier void set(type);
      modifier type get();
}
```

Parameters

[,]

The notation for an indexed property. Commas are optional; for each additional parameter you want the accessor methods to take, add a comma.

modifier

A modifier that can be used on either the event declaration or on an event accessor method. Possible values are static and virtual.

property_name

Parameter(s) for the raise method, must match the signature of the delegate.

type

The type of the value represented by the property.

Remarks

The **property** keyword introduces the declaration of a property and can appear in a class, interface, or value type. A property can have a getter function (read only), a setter function (write only), or both (read-write).

A property name cannot match the name of the managed class containing it. The return type of the getter function must match the type of the last parameter of a corresponding setter function.

To client code, a property has the appearance of an ordinary data member, and can be written to or read from using the same syntax as a data member.

The get and set methods need not agree on the **virtual** *modifier*.

The accessibility of the get and set method can differ.

The definition of a property method can appear outside the class body similar to an ordinary method.

The get and the set method for a property shall agree on the **static** modifier.

A property is scalar if its get and set methods fit the following description:

- The get method has no parameters, and has return type *T*.
- The set method has a single parameter of type *T*, and **void** return type.

There shall be only one scalar property declared in a single scope with the same identifier. Scalar properties cannot be overloaded.

When a property data member is declared, the compiler will inject a data member, sometimes called the "backing store", in the class. However, the name of the data member will be of a form, such that, you cannot reference the member in the source as if it were an actual data member of the containing class. Use ildasm.exe to view the metadata for your type and see the compiler generated name for the property's backing store.

Different accessibility is allowed for the accessor methods in a property block. That is, the set method can be public and the get method can be private. However, it is an error for an accessor method to have a less restrictive accessibility than what is on the

declaration of the property itself.

property is a context-sensitive keyword. See Context-Sensitive Keywords for more information.

For more information about properties, see

- Indexed Properties
- Static Properties
- Virtual Properties
- Multidimensional Properties
- Overloading Property Accessor Methods
- How to: Declare Abstract and Sealed Properties

The following example shows the declaration and use of a property data member and a property block. It also shows that a property accessor can be defined out of class.

Example

```
// mcppv2_property.cpp
// compile with: /clr
using namespace System;
public ref class C {
   int MyInt;
public:

   // property data member
   property String ^ Simple_Property;

   // property block
   property int Property_Block {

      int get();

      void set(int value) {
         MyInt = value;
      }
   }
};

int C::Property_Block::get() {
   return MyInt;
}

int main() {
   C ^ MyC = gcnew C();
   MyC->Simple_Property = "test";
   Console::WriteLine(MyC->Simple_Property);

   MyC->Property_Block = 21;
   Console::WriteLine(MyC->Property_Block);
}
```

Output

```
test
21
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

How to: Use Simple Properties

For simple properties, those that merely assign and retrieve a private data member, you do not explicitly define the get and set accessor functions. The compiler will automatically provide accessor functions, given just the data type of the property The code below demonstrates a simple property.

Example

```
// SimpleProperties.cpp
// compile with: /clr
using namespace System;

ref class C {
public:
   property int Size;
};

int main() {
   C^ c = gcnew C;
   c->Size = 111;
   Console::WriteLine("c->Size = {0}", c->Size);
}
```

Output

```
c->Size = 111
```

See Also

Reference

property

How to: Use Indexed Properties

An indexed property typically exposes a data structure that is accessed using a subscript operator.

A default indexed property allows the user to access the data structure via the class name, whereas a user-defined indexed property requires the user to specify the property name to access the data structure.

For information on accessing a default indexer via the **this** pointer, see Semantics of the this Pointer in Value and Reference Types.

For information on consuming an indexer authored in C#, see How to: Consume a C# Indexer.

Example

The following code sample shows how to use default and user defined indexed properties.

```
// mcppv2_property_2.cpp
// compile with: /clr
using namespace System;
public ref class C {
   array<int>^ MyArr;

public:
   C() {
      MyArr = gcnew array<int>(5);
   }

   // default indexer
   property int default[int] {
      int get(int index) {
         return MyArr[index];
      }
      void set(int index, int value) {
         MyArr[index] = value;
      }
   }

   // user-defined indexer
   property int indexer1[int] {
      int get(int index) {
         return MyArr[index];
      }
      void set(int index, int value) {
         MyArr[index] = value;
      }
   }
};

int main() {
   C ^ MyC = gcnew C();

   // use the default indexer
   Console::Write("[ ");
   for (int i = 0 ; i < 5 ; i++) {
      MyC[i] = i;
      Console::Write("{0} ", MyC[i]);
   }

   Console::WriteLine("]");

   // use the user-defined indexer
   Console::Write("[ ");
   for (int i = 0 ; i < 5 ; i++) {
      MyC->indexer1[i] = i * 2;
      Console::Write("{0} ", MyC->indexer1[i]);
   }
```

```
   Console::WriteLine("]");
}
```

Output

```
[ 0 1 2 3 4 ]
[ 0 2 4 6 8 ]
```

Example

This sample shows how to call the default indexer through the **this** pointer.

```
// call_default_indexer_through_this_pointer.cpp
// compile with: /clr /c
value class Position {
public:
   Position(int x, int y) : position(gcnew array<int, 2>(100, 100)) {
      this->default[x, y] = 1;
   }

   property int default[int, int] {
      int get(int x, int y) {
         return position[x, y];
      }

      void set(int x, int y, int value) {}
   }

private:
   array<int, 2> ^ position;
};
```

Example

This sample shows how you can use DefaultMemberAttribute to specify the default indexer.

```
// specify_default_indexer.cpp
// compile with: /LD /clr
using namespace System;
[Reflection::DefaultMember("XXX")]
public ref struct Squares {
   property Double XXX[Double] {
      Double get(Double data) {
         return data*data;
      }
   }
};
```

Example

This sample consumes the metadata created in the previous example.

```
// consume_default_indexer.cpp
// compile with: /clr
#using "specify_default_indexer.dll"
int main() {
   Squares ^ square = gcnew Squares();
   System::Console::WriteLine("{0}", square[3]);
}
```

Output

```
9
```

See Also

Reference

property

How to: Declare and Use Static Properties

This code sample shows how to declare and use a static property. A static property can only access static members of its class.

Example

```
// mcppv2_property_3.cpp
// compile with: /clr
using namespace System;

ref class StaticProperties {
   static int MyInt;
   static int MyInt2;

public:
   static property int Static_Data_Member_Property;

   static property int Static_Block_Property {
      int get() {
         return MyInt;
      }

      void set(int value) {
         MyInt = value;
      }
   }
};

int main() {
   StaticProperties::Static_Data_Member_Property = 96;
   Console::WriteLine(StaticProperties::Static_Data_Member_Property);

   StaticProperties::Static_Block_Property = 47;
   Console::WriteLine(StaticProperties::Static_Block_Property);
}
```

Output

```
96
47
```

See Also

Reference

property

How to: Declare and Use Virtual Properties

This code sample shows how to declare and use virtual properties.

Example

```
// mcppv2_property_4.cpp
// compile with: /clr
using namespace System;
interface struct IEFace {
public:
   property int VirtualProperty1;
   property int VirtualProperty2 {
      int get();
      void set(int i);
   }
};

// implement virtual events
ref class PropImpl : public IEFace {
   int MyInt;
public:
   virtual property int VirtualProperty1;

   virtual property int VirtualProperty2 {
      int get() {
         return MyInt;
      }
      void set(int i) {
         MyInt = i;
      }
   }
};

int main() {
   PropImpl ^ MyPI = gcnew PropImpl();
   MyPI->VirtualProperty1 = 93;
   Console::WriteLine(MyPI->VirtualProperty1);

   MyPI->VirtualProperty2 = 43;
   Console::WriteLine(MyPI->VirtualProperty2);
}
```

Output

```
93
43
```

See Also

Reference

property

How to: Use Multidimensional Properties

Multidimension properties let you define property accessor methods that take a non-standard number of parameters.

Example

```
// mcppv2_property_5.cpp
// compile with: /clr
ref class X {
   double d;
public:
   X() : d(0) {}
   property double MultiDimProp[int, int, int] {
      double get(int, int, int) {
         return d;
      }
      void set(int i, int j, int k, double l) {
         // do something with those ints
         d = l;
      }
   }

   property double MultiDimProp2[int] {
      double get(int) {
         return d;
      }
      void set(int i, double l) {
         // do something with those ints
         d = l;
      }
   }

};

int main() {
   X ^ MyX = gcnew X();
   MyX->MultiDimProp[0,0,0] = 1.1;
   System::Console::WriteLine(MyX->MultiDimProp[0, 0, 0]);
}
```

Output

```
1.1
```

See Also

Reference

property

How to: Overload Property Accessor Methods

It is possible to overload indexed properties.

Example

```
// mcppv2_property_6.cpp
// compile with: /clr
ref class X {
   double d;
public:
   X() : d(0.0) {}
   property double MyProp[int] {
      double get(int i) {
         return d;
      }

      double get(System::String ^ i) {
         return 2*d;
      }

      void set(int i, double l) {
         d = i * l;
      }
   }   // end MyProp definition
};

int main() {
   X ^ MyX = gcnew X();
   MyX->MyProp[2] = 1.7;
   System::Console::WriteLine(MyX->MyProp[1]);
   System::Console::WriteLine(MyX->MyProp["test"]);
}
```

Output

```
3.4
6.8
```

See Also

Reference

property

How to: Declare Abstract and Sealed Properties

Although specified as valid in the ECMA C++/CLI , in the Microsoft Visual C++ 2005 compiler you cannot specify the abstract (Visual C++) or sealed keywords on trivial properties, or on the property declaration of a non-trivial property.

To declare a sealed or abstract property, you must define a non-trivial property and specify the **abstract** or **sealed** keywords on the get and set accessor functions.

Example

```
// properties_abstract_sealed.cpp
// compile with: /clr
ref struct A {
protected:
   int m_i;

public:
   A() { m_i = 87; }

   // define abstract property
   property int Prop_1 {
      virtual int get() abstract;
      virtual void set(int i) abstract;
   }
};

ref struct B : A {
private:
   int m_i;

public:
   B() { m_i = 86; }

   // implement abstract property
   property int Prop_1 {
      virtual int get() override { return m_i; }
      virtual void set(int i) override { m_i = i; }
   }
};

ref struct C {
private:
   int m_i;

public:
   C() { m_i = 87; }

   // define sealed property
   property int Prop_2 {
      virtual int get() sealed { return m_i; }
      virtual void set(int i) sealed { m_i = i; };
   }
};

int main() {
   B b1;
   // call implementation of abstract property
   System::Console::WriteLine(b1.Prop_1);

   C c1;
   // call sealed property
   System::Console::WriteLine(c1.Prop_2);
}
```

Output

86
87

See Also

Reference

property

safe_cast

safe_cast allows you to change the type of an expression and generate verifiable MSIL code.

```
[cli]::safe_cast<type-id>(expression)
```

Parameters

type-id

A handle to a reference or value type, a value type, or a tracking reference to a reference or value type.

source

An expression that evaluates to a handle to a reference or value type, a value type, or a tracking reference to a reference or value type.

Remarks

The expression **safe_cast<***type-id***>(***expression***)** converts the operand expression to an object of type type-id.

The compiler will accept a static_cast in most places that it will accept a **safe_cast**. However, **safe_cast** is guaranteed to produce verifiable MSIL, where as a **static_cast** could produce unverifiable MSIL. See Pure and Verifiable Code and PEVerify Tool (Peverify.exe) for more information on verifiable code.

Like **static_cast**, **safe_cast** invokes user-defined conversions.

safe_cast is a keyword defined inside the cli namespace, which is a compiler-defined namespace. For more information, see cli Namespace.

For more information about casts, see Casting Operators.

safe_cast does not apply a **const_cast** (cast away **const**).

safe_cast is in the cli namespace. See cli Namespace for more information.

For more information on **safe_cas**t, see:

- C-Style Casts with /clr
- How to: Upcast with safe_cast
- How to: Downcast with safe_cast
- How to: Use safe_cast and Generic Types
- How to: Use safe_cast and User-Defined Conversions
- How to: Use safe_cast and Boxing
- How to: Use safe_cast and Unboxing

Example

One example of where the compiler will not accept a **static_cast** but will accept a **safe_cast** is for casts between unrelated interface types. With **safe_cast**, the compiler will not issue a conversion error and will perform a check at runtime to see if the cast is possible

```
// safe_cast.cpp
// compile with: /clr
using namespace System;

interface class I1 {};
interface class I2 {};
interface class I3 {};

ref class X : public I1, public I2 {};
```

```
int main() {
   I1^ i1 = gcnew X;
   I2^ i2 = safe_cast<I2^>(i1);   // OK, I1 and I2 have common type: X
   // I2^ i3 = static_cast<I2^>(i1);   C2440 use safe_cast instead
   try {
      I3^ i4 = safe_cast<I3^>(i1);   // fail at runtime, no common type
   }
   catch(InvalidCastException^) {
      Console::WriteLine("Caught expected exception");
   }
}
```

Output

```
Caught expected exception
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

How to: Upcast with safe_cast

An upcast is a cast from a derived type to one of its base classes. This cast is safe and does not require an explicit cast notation. The following sample shows how to perform an upcast with and without **safe_cast**.

Example

```
// safe_upcast.cpp
// compile with: /clr
using namespace System;
interface class A {
   void Test();
};

ref struct B : public A {
   virtual void Test() {
      Console::WriteLine("in B::Test");
   }

   void Test2() {
      Console::WriteLine("in B::Test2");
   }
};

ref struct C : public B {
   virtual void Test() override {
      Console::WriteLine("in C::Test");
   };
};

interface class I {} ;
value struct V : public I {};

int main() {
   C ^ c = gcnew C;

   // implicit upcast
   B ^ b = c;
   b->Test();
   b->Test2();

   // upcast with safe_cast
   b = nullptr;
   b = safe_cast<B^>(c);

   V ^ pv;

   // implicit upcast
   I^ i = pv;

   // upcast with safe_cast
   i = safe_cast<I^>(pv);
}
```

Output

```
in C::Test
in B::Test2
```

See Also

Reference

safe_cast

How to: Downcast with safe_cast

A downcast is a cast from a base class to a class derived from the base class. A downcast is only safe if the object addressed at runtime is actually addressing a derived class object. Unlike **static_cast**, **safe_cast** performs a dynamic check and throws InvalidCastException if the conversion fails.

Example

```
// safe_downcast.cpp
// compile with: /clr
using namespace System;

interface class A { void Test(); };

ref struct B : public A {
   virtual void Test() {
      Console::WriteLine("in B::Test()");
   }

   void Test2() {
      Console::WriteLine("in B::Test2()");
   }
};

ref struct C : public B {
   virtual void Test() override {
      Console::WriteLine("in C::Test()");
   }
};

interface class I {};

value struct V : public I {};

int main() {
   A^ a = gcnew C();
   a->Test();
   B^ b = safe_cast<B^>(a);
   b->Test();
   b->Test2();

   V v;
   I^ i = v;   // i boxes V
   V^ refv = safe_cast<V^>(i);

   Object^ o = gcnew B;
   A^ a2= safe_cast<A^>(o);
}
```

Output

```
in C::Test()
in C::Test()
in B::Test2()
```

See Also

Reference

safe_cast

How to: Use safe_cast and Generic Types

The following sample shows that **safe_cast** can perform a downcast with a generic type.

Remarks

```
// safe_cast_generic_types.cpp
// compile with: /clr
interface struct I {};

generic<class T> where T:I
ref struct Base {
   T t;
   void test1() {}
};

generic<class T> where T:I
ref struct Derived:public Base <T> {};

ref struct R:public I {};

typedef Base<R^> GBase_R;
typedef Derived<R^> GDerived_R;

int main() {
   GBase_R^ br = gcnew GDerived_R();
   GDerived_R^ dr = safe_cast<GDerived_R^>(br);
}
```

See Also

Reference

safe_cast

How to: Use safe_cast and User-Defined Conversions

The following sample shows that user-defined conversions can be invoked by a **safe_cast**.

Example

```
// safe_cast_udc.cpp
// compile with: /clr
using namespace System;
value struct V;

ref struct R {
   int x;
   R() {
      x = 1;
   }

   R(int argx) {
      x = argx;
   }

   static operator R::V^(R^ r);
};

value struct V {
   int x;
   static operator R^(V& v) {
      Console::WriteLine("in operator R^(V& v)");
      R^ r = gcnew R();
      r->x = v.x;
      return r;
   }

   V(int argx) {
      x = argx;
   }
};

   R::operator V^(R^ r) {
      Console::WriteLine("in operator V^(R^ r)");
      return gcnew V(r->x);
   }

int main() {
   bool fReturnVal = false;
   V v(2);
   R^ r = safe_cast<R^>(v);   // should invoke UDC
   V^ v2 = safe_cast<V^>(r);   // should invoke UDC
}
```

Output

```
in operator R^(V& v)
in operator V^(R^ r)
```

See Also

Reference

safe_cast

How to: Use safe_cast and Boxing

Boxing in defined as a compiler-injected, user-defined conversion. Therefore, **safe_cast** can be used to box a value on the CLR heap.

The following sample shows boxing with simple and user defined value types. A **safe_cast** will box a value type variable that is on the native stack so it can be assigned to a variable on the garbage-collected heap.

Example

```
// safe_cast_boxing.cpp
// compile with: /clr
using namespace System;

interface struct I {};

value struct V : public I {
   int m_x;

   V(int i) : m_x(i) {}
};

int main() {
   // box a value type
   V v(100);
   I^ i = safe_cast<I^>(v);

   int x = 100;
   V^ refv = safe_cast<V^>(v);
   int^ refi = safe_cast<int^>(x);
}
```

The following sample shows that boxing has a higher priority over a user-defined conversion in a **safe_cast** operation:

```
// safe_cast_boxing_2.cpp
// compile with: /clr
static bool fRetval = true;

interface struct I {};
value struct V : public I {
   int x;

   V(int argx) {
      x = argx;
   }

   static operator I^(V v) {
      fRetval = false;
      I^ pi = v;
      return pi;
   }
};

ref struct R {
   R() {}
   R(V^ pv) {}
};

int main() {
   V v(10);
   I^ pv = safe_cast<I^>(v);   // boxing will occur, not UDC "operator I^"
}
```

See Also

Reference

safe_cast

How to: Use safe_cast and Unboxing

Unoxing in defined as a compiler-injected, user-defined conversion. Therefore, **safe_cast** can be used to unbox a value on the CLR heap.

Unboxing is a user-defined conversion, but unlike boxing, unboxing must be explicitly, which means it must be performed by a **static_cast**, C-style cast, or **safe_cast**; unboxing cannot be performed implicitly.

Example

```
// safe_cast_unboxing.cpp
// compile with: /clr
int main() {
   System::Object ^ o = 42;
   int x = safe_cast<int>(o);
}
```

The following sample shows unboxing with value types and primitive types.

```
// safe_cast_unboxing_2.cpp
// compile with: /clr
using namespace System;

interface struct I {};

value struct VI : public I {};

void test1() {
   Object^ o = 5;
   int x = safe_cast<Int32>(o);
}

value struct V {
   int x;
   String^ s;
};

void test2() {
   V localv;
   Object^ o = localv;
   V unboxv = safe_cast<V>(o);
}

void test3() {
   V localv;
   V^ o2 = localv;
   V unboxv2 = safe_cast<V>(o2);
}

void test4() {
   I^ refi = VI();
   VI vi  = safe_cast<VI>(refi);
}

int main() {
   test1();
   test2();
   test3();
   test4();
}
```

See Also

Reference

safe_cast

sealed

sealed is a context sensitive keyword that can indicate:

- A virtual member cannot be overridden.
- A type cannot be used as a base type.

sealed is also valid when compiling for native targets (without **/clr**).

Remarks

See Override Specifiers and Native Compilations for more information.

You can detect at compile time if a type is sealed with **__is_sealed (*type*)**. For more information, see Compiler Support for Type Traits.

sealed is a context-sensitive keyword. See Context-Sensitive Keywords for more information.

Example

This sample shows the effect of **sealed** on a virtual member:

```
// sealed_keyword.cpp
// compile with: /clr
interface struct I1 {
   virtual void f();
   virtual void g();
};

ref class X : I1 {
public:
   virtual void f() {
      System::Console::WriteLine("X::f override of I1::f");
   }

   virtual void g() sealed {
      System::Console::WriteLine("X::f override of I1::g");
   }
};

ref class Y : public X {
public:
   virtual void f() override {
      System::Console::WriteLine("Y::f override of I1::f");
   }

   /*
   // the following override generates a compiler error
   virtual void g() override {
      System::Console::WriteLine("Y::g override of I1::g");
   }
   */
};

int main() {
   I1 ^ MyI = gcnew X;
   MyI -> f();
   MyI -> g();

   I1 ^ MyI2 = gcnew Y;
   MyI2 -> f();
}
```

Output

```
X::f override of I1::f
```

```
X::f override of I1::g
Y::f override of I1::f
```

The following sample shows how to mark a class as sealed:

```
// sealed_keyword_2.cpp
// compile with: /clr
interface struct I1 {
   virtual void f();
};

ref class X sealed : I1 {
public:
   virtual void f() override {}
};

ref class Y : public X {   // C3246 base class X is sealed
public:
   virtual void f() override {}
};
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

System::String Handling in Visual C++

This topic discusses how the Visual C++ compiler processes string literals when compiling with **/clr**.

Remarks

When compiling with **/clr**, the compiler will convert string literals to strings of type String. To preserve backward compatibility with existing code there are two exceptions to this:

- Exception handling. When a string literal is thrown, the compiler will catch it as a string literal.
- Template deduction. When a string literal is passed as a template argument, the compiler will not convert it to a **String**. Note, string literals passed as a generic argument will be promoted to **String**.

The compiler also has built-in support for three operators, which you can override to customize their behavior:

- System::String ^ operator +(System::String, System::String);
- System::String ^ operator +(System::Object, System::String);
- System::String ^ operator +(System::String, System::Object);

When passed a **String**, the compiler will box, if necessary, and then concatenate the object (with ToString) with the string.

When compiling with **/clr:oldSyntax**, string literals will not be converted to **String**.

For more information see:

- C++ String Literals
- /clr (Common Language Runtime Compilation)

Example

```
// string_operators.cpp
// compile with: /clr
using namespace System;

int main() {
   String ^ a = gcnew String("abc");
   String ^ b = "def";   // same as gcnew form
   Object ^ c = gcnew String("ghi");

   char d[100] = "abc";

   // variables of System::String returning a System::String
   Console::WriteLine(a + b);
   Console::WriteLine(a + c);
   Console::WriteLine(c + a);

   // accessing a character in the string
   Console::WriteLine(a[2]);

   // concatenation of three System::Strings
   Console::WriteLine(a + b + c);

   // concatenation of a System::String and string literal
   Console::WriteLine(a + "zzz");

   // you can append to a System::String ^
   Console::WriteLine(a + 1);
   Console::WriteLine(a + 'a');
   Console::WriteLine(a + 3.1);

   // test System::String ^ for equality
   a += b;
   Console::WriteLine(a);
   a = b;
```

```
   if (a == b)
      Console::WriteLine("a and b are equal");

   a = "abc";
   if (a != b)
      Console::WriteLine("a and b are not equal");

   // System:String ^ and tracking reference
   String^% rstr1 = a;
   Console::WriteLine(rstr1);

   // testing an empty System::String ^
   String ^ n;
   if (n == nullptr)
      Console::WriteLine("n is empty");
}
```

Output

```
abcdef
abcghi
ghiabc
c
abcdefghi
abczzz
abc1
abc97
abc3.1
abcdef
a and b are equal
a and b are not equal
abc
n is empty
```

This sample shows that you can overload the compiler-provided operators, and that the compiler will find a function overload based on the **String** type.

```
// string_operators_2.cpp
// compile with: /clr
using namespace System;

// a string^ overload will be favored when calling with a String
void Test_Overload(const char * a) {
   Console::WriteLine("const char * a");
}
void Test_Overload(String ^ a) {
   Console::WriteLine("String ^ a");
}

// overload will be called instead of compiler defined operator
String ^ operator +(String ^ a, String ^ b) {
   return ("overloaded +(String ^ a, String ^ b)");
}

// overload will be called instead of compiler defined operator
String ^ operator +(Object ^ a, String ^ b) {
   return ("overloaded +(Object ^ a, String ^ b)");
}

// overload will be called instead of compiler defined operator
String ^ operator +(String ^ a, Object ^ b) {
   return ("overloaded +(String ^ a, Object ^ b)");
}

int main() {
   String ^ a = gcnew String("abc");
   String ^ b = "def";   // same as gcnew form
   Object ^ c = gcnew String("ghi");
```

```
   char d[100] = "abc";

   Console::WriteLine(a + b);
   Console::WriteLine(a + c);
   Console::WriteLine(c + a);

   Test_Overload("hello");
   Test_Overload(d);
}
```

Output

```
overloaded +(String ^ a, String ^ b)
overloaded +(String ^ a, Object ^ b)
overloaded +(Object ^ a, String ^ b)
String ^ a
const char * a
```

The following sample shows that the compiler distinguishes between native strings and **String** strings.

```
// string_operators_3.cpp
// compile with: /clr
using namespace System;
int func() {
   throw "simple string";   // const char *
};

int func2() {
   throw "string" + "string";   // returns System::String
};

template<typename T>
void func3(T t) {
   Console::WriteLine(T::typeid);
}

int main() {
   try {
      func();
   }
   catch(char * e) {
      Console::WriteLine("char *");
   }

   try {
      func2();
   }
   catch(String^ str) {
      Console::WriteLine("String^ str");
   }

   func3("string");   // const char *
   func3("string" + "string");   // returns System::String
}
```

Output

```
char *
String^ str
System.SByte*
System.String
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

Type Forwarding

Type forwarding allows you to move a type from one assembly (assembly A) into another assembly (assembly B), such that, it is not necessary to recompile clients that consume assembly A.

```
#using "new.dll"
[assembly:TypeForwardedTo(type::typeid)];
```

Parameters

new

The assembly into which you are moving the type definition.

type

The type whose definition you are moving into another assembly.

Remarks

After a component (assembly) ships and is being used by client applications, you can use type forwarding to move a type from the component (assembly) into another assembly, ship the updated component (and any additional assemblies required), and the client applications will still work without being recompiled.

Type forwarding only works for components referenced by existing applications. When you rebuild an application, there must be the appropriate assembly references for any types used in the application.

When forwarding a type (Type A) from an assembly, you must add the **TypeForwardedTo** attribute for that type, as well as an assembly reference. The assembly that you reference must contain one of the following:

- The definition for Type A.
- A **TypeForwardedTo** attribute for Type A, as well as an assembly reference.

Examples of types that can be forwarded include:

- ref classes
- value classes
- enums
- interfaces

You cannot forward the following types:

- Generic types
- Native types
- Nested types (if you want to forward a nested type, you should forward the enclosing type)

You can forward a type to an assembly authored in any language targeting the common language runtime.

So, if a source code file that is used to build assembly A.dll contains a type definition (`ref class MyClass`), and you wanted to move that type definition to assembly B.dll, you would:

1. Move the `MyClass` type definition to a source code file used to build B.dll.
2. Build assembly B.dll
3. Delete the `MyClass` type definition from the source code used to build A.dll, and replace it with the following:

   ```
   #using "B.dll"
   [assembly:TypeForwardedTo(MyClass::typeid)];
   ```

4. Build assembly A.dll.

5. Use A.dll without recompiling client applications.

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

typeid

typeid is used to get the Type for a type at compile time.

```
type::typeid
```

Parameters

type

The name of a type (abstract declarator) for which you want the System::Type object.

Remarks

typeid is similar to getting the System::Type for a type at run time using GetType or GetType. However, typeid only accepts a type name as a parameter. If you want to use an instance of a type to get its System::Type name, use GetType.

typeid must be able to evaluate a type name (type) at compile time, whereas GetType evaluates the type to return at run time.

typeid can take a native type name or common language runtime alias for the native type name; see .NET Framework Equivalents to C++ Native Types for more information.

typeid also works with native types, although it will still return a System::Type. To get a type_info structure, use typeid Operator.

typeid is the sucessor to __typeof in the previous **/clr** syntax.

Example

```
// keyword__typeid.cpp
// compile with: /clr
using namespace System;

ref struct G {
   int i;
};

int main() {
   G ^ pG = gcnew G;
   Type ^ pType = pG->GetType();
   Type ^ pType2 = G::typeid;

   if (pType == pType2)
      Console::WriteLine("typeid and GetType returned the same System::Type");
   Console::WriteLine(G::typeid);

   typedef float* FloatPtr;
   Console::WriteLine(FloatPtr::typeid);
}
```

Output

```
typeid and GetType returned the same System::Type
G
System.Single*
```

The following sample shows that a variable of type System::Type can be used to get the attributes on a type. It also shows that for some types, you will have to create a typedef to use **typeid**.

```
// keyword__typeid_2.cpp
// compile with: /clr
using namespace System;
using namespace System::Security;
using namespace System::Security::Permissions;

typedef int ^ handle_to_int;
```

```
typedef int * pointer_to_int;

public ref class MyClass {};

class MyClass2 {};

[attribute(AttributeTargets::All)]
ref class AtClass {
public:
   AtClass(Type ^) {
      Console::WriteLine("in AtClass Type ^ constructor");
   }
};

[attribute(AttributeTargets::All)]
ref class AtClass2 {
public:
   AtClass2() {
      Console::WriteLine("in AtClass2 constructor");
   }
};

// Apply the AtClass and AtClass2 attributes to class B
[AtClass(MyClass::typeid), AtClass2]
[AttributeUsage(AttributeTargets::All)]
ref class B : Attribute {};

int main() {
   Type ^ MyType = B::typeid;

   Console::WriteLine(MyType->IsClass);

   array<Object^>^ MyArray = MyType -> GetCustomAttributes(true);
   for (int i = 0 ; i < MyArray->Length ; i++ )
      Console::WriteLine(MyArray[i]);

   if (int::typeid != pointer_to_int::typeid)
      Console::WriteLine("int::typeid != pointer_to_int::typeid, as expected");

   if (int::typeid == handle_to_int::typeid)
      Console::WriteLine("int::typeid == handle_to_int::typeid, as expected");
}
```

Sample Output

```
True
in AtClass2 constructor
in AtClass Type ^ constructor
AtClass2
System.AttributeUsageAttribute
AtClass
int::typeid != pointer_to_int::typeid, as expected
int::typeid == handle_to_int::typeid, as expected
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

User-Defined Attributes

This topic Discusses how to create and use custom attributes. Custom attributes let you extend the metadata of a managed element. For more information, see Extending Metadata Using Attributes.

Note
The information and syntax presented in this topic is meant to supersede the information presented in attribute.

Remarks

You can define a custom attribute by defining a type and making Attribute a base class for the type and optionally applying the AttributeUsageAttribute attribute.

For example, in Microsoft Transaction Server (MTS) 1.0, behavior with respect to transactions, synchronization, load balancing, and so on was specified through custom GUIDs inserted into the type library by using the ODL custom attribute. Hence, a client of an MTS server could determine its characteristics by reading the type library. In the .NET Framework, the analog of the type library is metadata, and the analog of the ODL custom attribute is custom attributes. Also, reading the type library is analogous to using reflection on the types.

For more information, see,

- Attribute Targets
- Attribute Parameter Types

For information on signing assemblies in Visual C++, see Strong Name Assemblies (Assembly Signing).

Example

The following sample shows how to define a custom attribute.

```
// user_defined_attributes.cpp
// compile with: /clr /c
using namespace System;

[AttributeUsage(AttributeTargets::All)]
ref struct Attr : public Attribute {
   Attr(bool i){}
   Attr(){}
};

[Attr]
ref class MyClass {};
```

The following example illustrates some important features of custom attributes. For example, this example shows a common usage of the custom attributes: instantiating a server that can fully describe itself to clients.

```
// extending_metadata_b.cpp
// compile with: /clr
using namespace System;
using namespace System::Reflection;

public enum class Access { Read, Write, Execute };

// Defining the Job attribute:
[AttributeUsage(AttributeTargets::Class, AllowMultiple=true )]
public ref class Job : Attribute {
public:
   property int Priority {
      void set( int value ) { m_Priority = value; }
      int get() { return m_Priority; }
   }

   // You can overload constructors to specify Job attribute in different ways
```

```
   Job() { m_Access = Access::Read; }
   Job( Access a ) { m_Access = a; }
   Access m_Access;

protected:
   int m_Priority;
};

interface struct IService {
   void Run();
};

   // Using the Job attribute:
   // Here we specify that QueryService is to be read only with a priority of 2.
   // To prevent namespace collisions, all custom attributes implicitly
   // end with "Attribute".

[Job( Access::Read, Priority=2 )]
ref struct QueryService : public IService {
   virtual void Run() {}
};

// Because we said AllowMultiple=true, we can add multiple attributes
[Job(Access::Read, Priority=1)]
[Job(Access::Write, Priority=3)]
ref struct StatsGenerator : public IService {
   virtual void Run( ) {}
};

int main() {
   IService ^ pIS;
   QueryService ^ pQS = gcnew QueryService;
   StatsGenerator ^ pSG = gcnew StatsGenerator;

   //  use QueryService
   pIS = safe_cast<IService ^>( pQS );

   // use StatsGenerator
   pIS = safe_cast<IService ^>( pSG );

   // Reflection
   MemberInfo ^ pMI = pIS->GetType();
   array <Object ^ > ^ pObjs = pMI->GetCustomAttributes(false);

   // We can now quickly and easily view custom attributes for an
   // Object through Reflection */
   for( int i = 0; i < pObjs->Length; i++ ) {
      Console::Write("Service Priority = ");
      Console::WriteLine(static_cast<Job^>(pObjs[i])->Priority);
      Console::Write("Service Access = ");
      Console::WriteLine(static_cast<Job^>(pObjs[i])->m_Access);
   }
}
```

Sample Output

```
Service Priority = 0
Service Access = Write
Service Priority = 3
Service Access = Write
Service Priority = 1
Service Access = Read
```

The Object^ type replaces the variant data type. The following example defines a custom attribute that takes an array of Object^ as parameters.

Note

Attribute arguments must be compile-time constants; in most cases, they should be constant literals.

See typeid for information on how to return a value of System::Type from a custom attribute block.

```
// extending_metadata_e.cpp
// compile with: /clr /c
using namespace System;
[AttributeUsage(AttributeTargets::Class | AttributeTargets::Method)]
public ref class AnotherAttr : public Attribute {
public:
   AnotherAttr(array<Object^>^) {}
   array<Object^>^ var1;
};

// applying the attribute
[ AnotherAttr( gcnew array<Object ^> { 3.14159, "pi" }, var1 = gcnew array<Object ^> { "a",
"b" } ) ]
public ref class SomeClass {};
```

The runtime requires that the public part of the custom attribute class must be serializable. When authoring custom attributes, named arguments of your custom attribute are limited to compile-time constants. (Think of it as a sequence of bits appended to your class layout in the metadata.)

```
// extending_metadata_f.cpp
// compile with: /clr /c
using namespace System;
ref struct abc {};

[AttributeUsage( AttributeTargets::All )]
ref struct A : Attribute {
   A( Type^ ) {}
   A( String ^ ) {}
   A( int ) {}
};

[A( abc::typeid )]
ref struct B {};
```

Requirements

Compiler option: **/clr**

See Also

Concepts

New Language Features in Visual C++

Attribute Parameter Types

Values passed to attributes must be known to the compiler at compile time. Attribute parameters can be of the following types:

- **bool**
- **char**, **unsigned char**
- **short**, **unsigned short**
- **int**, **unsigned int**
- **long**, **unsigned long**
- **__int64**, **unsigned __int64**
- **float**, **double**
- **wchar_t**
- **char*** or **wchar_t*** or **System::String***
- **System::Type ^**
- **System::Object ^**
- **enum**

Example

```
// attribute_parameter_types.cpp
// compile with: /clr /c
using namespace System;
ref struct AStruct {};

[AttributeUsage(AttributeTargets::ReturnValue)]
ref struct Attr : public Attribute {
   Attr(AStruct ^ i){}
   Attr(bool i){}
   Attr(){}
};

ref struct MyStruct {
   static AStruct ^ x = gcnew AStruct;
   [returnvalue:Attr(x)] int Test() { return 0; }   // C3104
   [returnvalue:Attr] int Test2() { return 0; }   // OK
   [returnvalue:Attr(true)] int Test3() { return 0; }   // OK
};
```

Example

When specifying attributes, all unnamed (positional) arguments must precede any named arguments.

```
// extending_metadata_c.cpp
// compile with: /clr /c
using namespace System;
[AttributeUsage(AttributeTargets::Class)]
ref class MyAttr : public Attribute {
public:
   MyAttr() {}
   MyAttr(int i) {}
   property int Priority;
   property int Version;
};

[MyAttr]
ref class ClassA {};   // No arguments
```

```
[MyAttr(Priority = 1)]
ref class ClassB {};   // Named argument

[MyAttr(123)]
ref class ClassC {};   // Positional argument

[MyAttr(123, Version = 1)]
ref class ClassD {};   // Positional and named
```

Example

Attribute parameters can be one-dimensional arrays of the previous types.

```
// extending_metadata_d.cpp
// compile with: /clr /c
using namespace System;

[AttributeUsage(AttributeTargets::Class)]
public ref struct ABC : public Attribute {
   ABC(array<int>^){}
   array<double> ^ param;
};

[ABC( gcnew array<int> {1,2,3}, param = gcnew array<double>{2.71, 3.14})]
ref struct AStruct{};
```

See Also

Reference

User-Defined Attributes

Attribute Targets

Attribute usage specifiers let you specify attribute targets. Each attribute is defined to apply to certain language elements. For example, an attribute might be defined to apply only to classes and structs. The following list shows the possible syntactic elements on which a custom attribute can be used. Combinations of these values (using logical or) may be used.

To specify attribute target, to pass one or more AttributeTargets enumerators to AttributeUsageAttribute when defining the attribute.

The following is a list of the valid attribute targets:

Target	Sample Usage

All
 	
(applies to all constructs)
	`// attribute_targets_all.cpp`
	`// compile with: /clr /c`
	`using namespace System;`
	`[AttributeUsage(AttributeTargets::All)]`
	`ref class Attr : public Attribute {};`
	
	`[assembly:Attr];`

Assembly
 	
(applies to an assembly as a whole)
	`// attribute_targets_assembly.cpp`
	`// compile with: /clr /c`
	`using namespace System;`
	`[AttributeUsage(AttributeTargets::Assembly)]`
	`ref class Attr : public Attribute {};`
	
	`[assembly:Attr];`

Module
 	
(applies to a module as a whole)
	`// attribute_targets_module.cpp`
	`// compile with: /clr /c`
	`using namespace System;`
	`[AttributeUsage(AttributeTargets::Module)]`
	`ref class Attr : public Attribute {};`
	
	`[module:Attr];`

Class	`// attribute_targets_class.cpp`
	`// compile with: /clr /c`
	`using namespace System;`
	`[AttributeUsage(AttributeTargets::Class)]`
	`ref class Attr : public System::Attribute {};`
	
	`[Attr]  // same as [class:Attr]`
	`ref class MyClass {};`

Struct	`// attribute_targets_struct.cpp`
	`// compile with: /clr /c`
	`using namespace System;`
	`[AttributeUsage(AttributeTargets::Struct)]`
	`ref class Attr : public Attribute {};`
	
	`[Attr]  // same as [struct:Attr]`
	`value struct MyStruct{};`

enum	```// attribute_targets_enum.cpp
	// compile with: /clr /c
	using namespace System;
	[AttributeUsage(AttributeTargets::Enum)]
	ref class Attr : public Attribute {};
	
	[Attr] // same as [enum:Attr]
	enum struct MyEnum{e, d};```

Constructor	```// attribute_targets_constructor.cpp
	// compile with: /clr /c
	using namespace System;
	[AttributeUsage(AttributeTargets::Constructor)]
	ref class Attr : public Attribute {};
	
	ref struct MyStruct{
	  [Attr] MyStruct(){} // same as [constructor:Attr]
	};```

Method	```// attribute_targets_method.cpp
	// compile with: /clr /c
	using namespace System;
	[AttributeUsage(AttributeTargets::Method)]
	ref class Attr : public Attribute {};
	
	ref struct MyStruct{
	  [Attr] void Test(){} // same as [method:Attr]
	};```

Property	```// attribute_targets_property.cpp
	// compile with: /clr /c
	using namespace System;
	[AttributeUsage(AttributeTargets::Property)]
	ref class Attr : public Attribute {};
	
	ref struct MyStruct{
	  [Attr] property int Test; // same as [property:Attr]
	};```

Field	```// attribute_targets_field.cpp
	// compile with: /clr /c
	using namespace System;
	[AttributeUsage(AttributeTargets::Field)]
	ref class Attr : public Attribute {};
	
	ref struct MyStruct{
	  [Attr] int Test; // same as [field:Attr]
	};```

Event	`// attribute_targets_event.cpp`
	`// compile with: /clr /c`
	`using namespace System;`
	`[AttributeUsage(AttributeTargets::Event)]`
	`ref class Attr : public Attribute {};`
	
	`delegate void ClickEventHandler(int, double);`
	
	`ref struct MyStruct{`
	`  [Attr] event ClickEventHandler^ OnClick;  // same as [event:A`
	`ttr]`
	`};`

Interface	`// attribute_targets_interface.cpp`
	`// compile with: /clr /c`
	`using namespace System;`
	`[AttributeUsage(AttributeTargets::Interface)]`
	`ref class Attr : public Attribute {};`
	
	`[Attr]  // same as [event:Attr]`
	`interface struct MyStruct{};`

Parameter	`// attribute_targets_parameter.cpp`
	`// compile with: /clr /c`
	`using namespace System;`
	`[AttributeUsage(AttributeTargets::Parameter)]`
	`ref class Attr : public Attribute {};`
	
	`ref struct MyStruct{`
	`  void Test([Attr] int i);`
	`  void Test2([parameter:Attr] int i);`
	`};`

Delegate	`// attribute_targets_delegate.cpp`
	`// compile with: /clr /c`
	`using namespace System;`
	`[AttributeUsage(AttributeTargets::Delegate)]`
	`ref class Attr : public Attribute {};`
	
	`[Attr] delegate void Test();`
	`[delegate:Attr] delegate void Test2();`

ReturnValue	`// attribute_targets_returnvalue.cpp`
	`// compile with: /clr /c`
	`using namespace System;`
	`[AttributeUsage(AttributeTargets::ReturnValue)]`
	`ref class Attr : public Attribute {};`
	
	`ref struct MyStruct {`
	`  // Note required specifier`
	`  [returnvalue:Attr] int Test() { return 0; }`
	`};`

Typically, an attribute directly precedes the language element to which it applies. In some cases, however, the position of an attribute is not sufficient to determine the attribute's intended target. Consider this example:

```
[Attr] int MyFn(double x)...
```

Syntactically, there is no way to tell if the attribute is intended to apply to the method or to the method's return value (in this

case, it defaults to the method). In such cases, an attribute usage specifier may be used. For example, to make the attribute apply to the return value, use the **returnvalue** specifier, as follows:

```
[returnvalue:Attr] int MyFn(double x)... // applies to return value
```

Attribute usage specifiers are required in the following situations:

- To specify an assembly- or module-level attribute.
- To specify that an attribute applies to a method's return value, not the method:

  ```
  [method:Attr] int MyFn(double x)...     // Attr applies to method
  [returnvalue:Attr] int MyFn(double x)...// Attr applies to return value
  [Attr] int MyFn(double x)...            // default: method
  ```

- To specify that an attribute applies to a property's accessor, not the property:

  ```
  [method:MyAttr(123)] property int Property()
  [property:MyAttr(123)] property int Property()
  [MyAttr(123)] property int get_MyPropy() // default: property
  ```

- To specify that an attribute applies to an event's accessor, not the event:

  ```
  delegate void MyDel();
  ref struct X {
     [field:MyAttr(123)] event MyDel* MyEvent;   //field
     [event:MyAttr(123)] event MyDel* MyEvent;   //event
     [MyAttr(123)] event MyDel* MyEvent;   // default: event
  }
  ```

An attribute usage specifier applies only to the attribute that immediately follows it; that is,

```
[returnvalue:Attr1, Attr2]
```

is different from

```
[returnvalue:Attr1, returnvalue:Attr2]
```

Example

This sample shows how to specify multiple targets.

```
// attribute_targets.cpp
// compile with: /clr /c
using namespace System;
[AttributeUsage(AttributeTargets::Class | AttributeTargets::Struct, AllowMultiple = true )]
ref struct Attr : public Attribute {
   Attr(bool i){}
   Attr(){}
};

[Attr]
ref class MyClass {};

[Attr]
[Attr(true)]
value struct MyStruct {};
```

See Also

Reference

User-Defined Attributes

XML Documentation (Visual C++)

In Visual C++, you can add comments to your source code that will be processed to an .xml file. This file can then be the input to a process that creates documentation for the classes in your code. An .xml file can also be used to support IntelliSense on your component.

XML documentation can be used regardless of whether you compile with or without **/clr**.

Note
In the current release, code comments are not processed on templates or anything containing a template type (for example, a function taking a parameter as a template). Doing so will result in undefined behavior.

For details on creating an .xml file with documentation comments, see the following topics.

For information about	**See**
The compiler options to use	/doc and (optionally) /clr
Tags you can use to provide commonly used functionality in documentation	Recommended Tags for Documentation Comments
The ID strings that the compiler produces to identify the constructs in your code	Processing the .xml File
How to delimit documentation tags	Delimiters for Visual C++ Documentation Tags
Generating an .xml file from one or more .xdc files.	XDCMake Reference
Links to information about XML as it relates to Visual Studio feature areas	XML in Visual Studio

If you need to put XML special characters in the text of a documentation comment, you must use XML entities or a CDATA section.

See Also

Concepts

New Language Features in Visual C++

Recommended Tags for Documentation Comments (Visual C++)

The Visual C++ compiler will process documentation comments in your code and creates an .xdc file for each compiland, and xdcmake.exe will process the .xdc files to an .xml file. Processing the .xml file to create documentation is a detail that needs to be implemented at your site.

Tags are processed on constructs such as types and type members.

Tags must immediately precede types or members.

Note
Documentation comments cannot be applied to a namespace or on out of line definition for properties and events; documentation comments must on the in-class declarations.

The compiler will process any tag that is valid XML. The following tags provide commonly used functionality in user documentation:

<c>	<code>	<example>
<exception>1	<include>1	<list>
<para>	<param>1	<paramref>1
<permission>1	<remarks>	<returns>
<see>1	<seealso>1	<summary>
<value>		

1. Compiler verifies syntax.

In the current release, the Visual C++ compiler does not support **<paramref>**, a tag that is supported by other Visual Studio compilers. Visual C++ may support **<paramref>** in a future release.

See Also

Other Resources

XML Documentation (Visual C++)

<c> (Visual C++)

The <c> tag indicates that text within a description should be marked as code. Use <code> to indicate multiple lines as code.

```
<c>text</c>
```

Parameters

text

The text you want to indicate as code.

Remarks

Compile with /doc to process documentation comments to a file.

Example

```
// xml_c_tag.cpp
// compile with: /doc /LD
// post-build command: xdcmake xml_c_tag.dll

/// Text for class MyClass.
class MyClass {
public:
   int m_i;
   MyClass() : m_i(0) {}

   /// <summary><c>MyMethod</c> is a method in the <c>MyClass</c> class.
   /// </summary>
   int MyMethod(MyClass * a) {
      return a -> m_i;
   }
};
```

See Also

Other Resources

XML Documentation (Visual C++)

<code> (Visual C++)

The <code> tag gives you a way to indicate one or more lines as code.

```
<code>content</code>
```

Parameters

content

The text you want marked as code.

Remarks

Use <c> to indicate a portion of text should be marked as code.

Compile with /doc to process documentation comments to a file.

Example

See the <example> topic for an example of how to use the <code> tag.

See Also

Other Resources

XML Documentation (Visual C++)

<example> (Visual C++)

The <example> tag lets you specify an example of how to use a method or other library member. Commonly, this would also involve use of the <code> tag.

```
<example>description</example>
```

Parameters

description

A description of the code sample.

Remarks

Compile with /doc to process documentation comments to a file.

Example

```
// xml_example_tag.cpp
// compile with: /clr /doc /LD
// post-build command: xdcmake xml_example_tag.dll

/// Text for class MyClass.
public ref class MyClass {
public:
   /// <summary>
   /// GetZero method
   /// </summary>
   /// <example> This sample shows how to call the GetZero method.
   /// <code>
   /// int main()
   /// {
   ///    return GetZero();
   /// }
   /// </code>
   /// </example>
   static int GetZero() {
      return 0;
   }
};
```

See Also

Other Resources

XML Documentation (Visual C++)

<exception> (Visual C++)

The <exception> tag lets you specify which exceptions can be thrown. This tag is applied to a method definition.

```
<exception cref="member">description</exception>
```

Parameters

member

A reference to an exception that is available from the current compilation environment. Using name lookup rules, the compiler checks that the given exception exists, and translates *member* to the canonical element name in the output XML. The compiler issues a warning if it does not find *member*.

Enclose the name in single or double quotation marks.

For information on how to create a cref reference to a generic type, see <see> (Visual C++).

description

A description.

Remarks

Compile with /doc to process documentation comments to a file.

The Visual C++ compiler will attempt to resolve cref references in one pass through the documentation comments. Therefore, if using the C++ lookup rules, a symbol is not found by the compiler the reference will be marked as unresolved. See <seealso> (Visual C++) for more information.

Example

```
// xml_exception_tag.cpp
// compile with: /clr /doc /LD
// post-build command: xdcmake xml_exception_tag.dll
using namespace System;

/// Text for class EClass.
public ref class EClass : public Exception {
   // class definition ...
};

/// <exception cref="System.Exception">Thrown when... .</exception>
public ref class TestClass {
   void Test() {
      try {
      }
      catch(EClass^) {
      }
   }
};
```

See Also

Other Resources

XML Documentation (Visual C++)

<include> (Visual C++)

The <include> tag lets you refer to comments in another file that describe the types and members in your source code. This is an alternative to placing documentation comments directly in your source code file. For example, you can use <include> to insert standard "boilerplate" comments that are used throughout your team or company.

```
<include file='filename' path='tagpath' />
```

Parameters

filename

The name of the file containing the documentation. The file name can be qualified with a path. Enclose the name in single or double quotation marks. The compiler issues a warning if it does not find *filename*.

tagpath

A valid XPath expression that selects the desired node-set contained in the file.

name

The name specifier in the tag that precedes the comments; *name* will have an *id*.

id

The ID for the tag that precedes the comments. Enclose the name in single or double quotation marks.

Remarks

The <include> tag uses the XML XPath syntax. Refer to XPath documentation for ways to customize using <include>.

Compile with /doc to process documentation comments to a file.

Example

This is a multifile example. The first file, which uses <include>, contains the following documentation comments:

```
// xml_include_tag.cpp
// compile with: /clr /doc /LD
// post-build command: xdcmake xml_include_tag.dll

/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test"]/*' />
public ref class Test {
   void TestMethod() {
   }
};

/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test2"]/*' />
public ref class Test2 {
   void Test() {
   }
};
```

The second file, xml_include_tag.doc, contains the following documentation comments:

```
<MyDocs>

<MyMembers name="test">
<summary>
The summary for this type.
</summary>
</MyMembers>

<MyMembers name="test2">
<summary>
The summary for this other type.
```

```
</summary>
</MyMembers>

</MyDocs>
```

Program Output

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>t2</name>
    </assembly>
    <members>
        <member name="T:Test">
            <summary>
The summary for this type.
</summary>
        </member>
        <member name="T:Test2">
            <summary>
The summary for this other type.
</summary>
        </member>
    </members>
</doc>
```

See Also

Other Resources

XML Documentation (Visual C++)

<list> (Visual C++)

The <listheader> block is used to define the heading row of either a table or definition list. When defining a table, you only need to supply an entry for term in the heading.

```
<list type="bullet" | "number" | "table">
   <listheader>
      <term>term</term>
      <description>description</description>
   </listheader>
   <item>
      <term>term</term>
      <description>description</description>
   </item>
</list>
```

Parameters

term

A term to define, which will be defined in *description*.

description

Either an item in a bullet or numbered list or the definition of a *term*.

Remarks

Each item in the list is specified with an <item> block. When creating a definition list, you will need to specify both *term* and *description*. However, for a table, bulleted list, or numbered list, you only need to supply an entry for *description*.

A list or table can have as many <item> blocks as needed.

Compile with /doc to process documentation comments to a file.

Example

```
// xml_list_tag.cpp
// compile with: /doc /LD
// post-build command: xdcmake xml_list_tag.dll
/// <remarks>Here is an example of a bulleted list:
/// <list type="bullet">
/// <item>
/// <description>Item 1.</description>
/// </item>
/// <item>
/// <description>Item 2.</description>
/// </item>
/// </list>
/// </remarks>
class MyClass {};
```

See Also

Other Resources

XML Documentation (Visual C++)

<para> (Visual C++)

The <para> tag is for use inside a tag, such as <summary>, <remarks>, or <returns>, and lets you add structure to the text.

```
<para>content</para>
```

Parameters

content

The text of the paragraph.

Remarks

Compile with /doc to process documentation comments to a file.

Example

See <summary> for an example of using <para>.

See Also

Other Resources

XML Documentation (Visual C++)

<param> (Visual C++)

The <param> tag should be used in the comment for a method declaration to describe one of the parameters for the method.

```
<param name='name'>description</param>
```

Parameters

name

The name of a method parameter. Enclose the name in single or double quotation marks. The compiler issues a warning if it does not find *name*.

description

A description for the parameter.

Remarks

The text for the <param> tag will be displayed in IntelliSense, the Object Browser, and in the Code Comment Web Report.

Compile with /doc to process documentation comments to a file.

Example

```
// xml_param_tag.cpp
// compile with: /clr /doc /LD
// post-build command: xdcmake xml_param_tag.dll
/// Text for class MyClass.
public ref class MyClass {
   /// <param name="Int1">Used to indicate status.</param>
   void MyMethod(int Int1) {
   }
};
```

See Also

Other Resources

XML Documentation (Visual C++)

<paramref> (Visual C++)

The <paramref> tag gives you a way to indicate that a word is a parameter. The .xml file can be processed to format this parameter in some distinct way.

```
<paramref name="name"/>
```

Parameters

name

The name of the parameter to refer to. Enclose the name in single or double quotation marks. The compiler issues a warning if it does not find *name*.

Remarks

Compile with /doc to process documentation comments to a file.

Example

```
// xml_paramref_tag.cpp
// compile with: /clr /doc /LD
// post-build command: xdcmake xml_paramref_tag.dll
/// Text for class MyClass.
public ref class MyClass {
   /// <summary>MyMethod is a method in the MyClass class.
   /// The <paramref name="Int1"/> parameter takes a number.
   /// </summary>
   void MyMethod(int Int1) {}
};
```

See Also

Other Resources

XML Documentation (Visual C++)

<permission> (Visual C++)

The <permission> tag lets you document the access of a member. PermissionSet lets you specify access to a member.

```
<permission cref="member">description</permission>
```

Parameters

member

A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and translates *member* to the canonical element name in the output XML. Enclose the name in single or double quotation marks.

The compiler issues a warning if it does not find *member*.

For information on how to create a cref reference to a generic type, see <see> (Visual C++).

description

A description of the access to the member.

Remarks

Compile with /doc to process documentation comments to a file.

The Visual C++ compiler will attempt to resolve cref references in one pass through the documentation comments. Therefore, if using the C++ lookup rules, a symbol is not found by the compiler the reference will be marked as unresolved. See <seealso> (Visual C++) for more information.

Example

```
// xml_permission_tag.cpp
// compile with: /clr /doc /LD
// post-build command: xdcmake xml_permission_tag.dll
using namespace System;
/// Text for class TestClass.
public ref class TestClass {
   /// <permission cref="System::Security::PermissionSet">Everyone can access this method.<
/permission>
   void Test() {}
};
```

See Also

Other Resources

XML Documentation (Visual C++)

<remarks> (Visual C++)

The <remarks> tag is used to add information about a type, supplementing the information specified with <summary>. This information is displayed in the Object Browser and in the Code Comment Web Report.

```
<remarks>description</remarks>
```

Parameters

description

A description of the member.

Remarks

Compile with /doc to process documentation comments to a file.

Example

```
// xml_remarks_tag.cpp
// compile with: /LD /clr /doc
// post-build command: xdcmake xml_remarks_tag.dll

using namespace System;

/// <summary>
/// You may have some primary information about this class.
/// </summary>
/// <remarks>
/// You may have some additional information about this class.
/// </remarks>
public ref class MyClass {};
```

See Also

Other Resources

XML Documentation (Visual C++)

<returns> (Visual C++)

The <returns> tag should be used in the comment for a method declaration to describe the return value.

```
<returns>description</returns>
```

Parameters

description

A description of the return value.

Remarks

Compile with /doc to process documentation comments to a file.

Example

```
// xml_returns_tag.cpp
// compile with: /LD /clr /doc
// post-build command: xdcmake xml_returns_tag.dll

/// Text for class MyClass.
public ref class MyClass {
public:
   /// <returns>Returns zero.</returns>
   int GetZero() { return 0; }
};
```

See Also

Other Resources

XML Documentation (Visual C++)

<see> (Visual C++)

The <see> tag lets you specify a link from within text. Use <seealso> to indicate text that you might want to appear in a See Also section.

```
<see cref="member"/>
```

Parameters

member

A reference to a member or field that is available to be called from the current compilation environment. Enclose the name in single or double quotation marks.

The compiler checks that the given code element exists and resolves *member* to the element name in the output XML. The compiler issues a warning if it does not find *member*.

Remarks

Compile with /doc to process documentation comments to a file.

See <summary> for an example of using <see>.

The Visual C++ compiler will attempt to resolve cref references in one pass through the documentation comments. Therefore, if using the C++ lookup rules, a symbol is not found by the compiler the reference will be marked as unresolved. See <seealso> (Visual C++) for more information.

Example

The following sample shows how you can make cref reference to a generic type, such that, the compiler will resolve the reference.

```
// xml_see_cref_example.cpp
// compile with: /LD /clr /doc
// the following cref shows how to specify the reference, such that,
// the compiler will resolve the reference
/// <summary cref="C{T}">
/// </summary>
ref class A {};

// the following cref shows another way to specify the reference,
// such that, the compiler will resolve the reference
// <summary cref="C &lt; T &gt;">

// the following cref shows how to hard-code the reference
/// <summary cref="T:C`1">
/// </summary>
ref class B {};

/// <summary cref="A">
/// </summary>
/// <typeparam name="T"></typeparam>
generic<class T>
ref class C {};
```

See Also

Other Resources

XML Documentation (Visual C++)

<seealso> (Visual C++)

The <seealso> tag lets you specify the text that you might want to appear in a See Also section. Use <see> to specify a link from within text.

```
<seealso cref="member"/>
```

Parameters

member

A reference to a member or field that is available to be called from the current compilation environment. Enclose the name in single or double quotation marks.

The compiler checks that the given code element exists and resolves *member* to the element name in the output XML. The compiler issues a warning if it does not find *member*.

For information on how to create a cref reference to a generic type, see <see> (Visual C++).

Remarks

Compile with /doc to process documentation comments to a file.

See <summary> (Visual C++) for an example of using <seealso>.

The Visual C++ compiler will attempt to resolve cref references in one pass through the documentation comments. Therefore, if using the C++ lookup rules, a symbol is not found by the compiler the reference will be marked as unresolved.

Example

In the following sample, an unresolved symbol is referenced in a cref. The XML comment for the cref to B::Test will be `<seealso cref="!:B::Test" />`, whereas the reference to A::Test is well-formed `<seealso cref="M:A.Test" />`.

```
// xml_seealso_tag.cpp
// compile with: /LD /clr /doc
// post-build command: xdcmake xml_seealso_tag.dll

/// Text for class A.
public ref struct A {
   /// <summary><seealso cref="A::Test"/>
   /// <seealso cref="B::Test"/>
   /// </summary>
   void MyMethod(int Int1) {}

   /// text
   void Test() {}
};

/// Text for class B.
public ref struct B {
   void Test() {}
};
```

See Also

Other Resources

XML Documentation (Visual C++)

<summary> (Visual C++)

The <summary> tag should be used to describe a type or a type member. Use <remarks> to add supplemental information to a type description.

```
<summary>description</summary>
```

Parameters

description

A summary of the object.

Remarks

The text for the <summary> tag is the only source of information about the type in IntelliSense, and is also displayed in the Object Browser and in the Code Comment Web Report.

Compile with /doc to process documentation comments to a file.

Example

```
// xml_summary_tag.cpp
// compile with: /LD /clr /doc
// post-build command: xdcmake xml_summary_tag.dll

/// Text for class MyClass.
public ref class MyClass {
public:
   /// <summary>MyMethod is a method in the MyClass class.
   /// <para>Here's how you could make a second paragraph in a description. <see cref="Syst
em::Console::WriteLine"/> for information about output statements.</para>
   /// <seealso cref="MyClass::MyMethod2"/>
   /// </summary>
   void MyMethod(int Int1) {}

   /// text
   void MyMethod2() {}
};
```

See Also

Other Resources

XML Documentation (Visual C++)

<value> (Visual C++)

The <value> tag lets you describe a property and property accessor methods. Note that when you add a property with a code wizard in the Visual Studio integrated development environment, it will add a <summary> tag for the new property. You should then manually add a <value> tag to describe the value that the property represents.

```
<value>property-description</value>
```

Parameters

property-description

A description for the property.

Remarks

Compile with /doc to process documentation comments to a file.

Example

```
// xml_value_tag.cpp
// compile with: /LD /clr /doc
// post-build command: xdcmake xml_value_tag.dll
using namespace System;
/// Text for class Employee.
public ref class Employee {
private:
   String ^ name;
   /// <value>Name accesses the value of the name data member</value>
public:
   property String ^ Name {
      String ^ get() {
         return name;
      }
      void set(String ^ i) {
         name = i;
      }
   }
};
```

See Also

Other Resources

XML Documentation (Visual C++)

.xml File Processing

The compiler generates an ID string for each construct in your code that is tagged to generate documentation. For more information, see Recommended Tags Documentation Comments. The ID string uniquely identifies the construct. Programs that process the .xml file can use the ID string to identify the corresponding .NET Framework metadata or reflection item to which the documentation applies.

The .xml file is not a hierarchical representation of your code, it is a flat list with a generated ID for each element.

The compiler observes the following rules when it generates the ID strings:

- No white space is placed in the string.
- The first part of the ID string identifies the kind of member being identified, with a single character followed by a colon. The following member types are used:

Character	Description
N	namespace You cannot add documentation comments to a namespace, cref references to a namespace are possible.
T	type: class, interface, struct, enum, delegate
D	typedef
F	field
P	property (including indexers or other indexed properties)
M	method (including such special methods as constructors, operators, and so forth)
E	event
!	error string The rest of the string provides information about the error. The Visual C++ compiler generates error informatio n for links that cannot be resolved.

- The second part of the string is the fully qualified name of the item, starting at the root of the namespace. The name of the item, its enclosing type or types, and namespace are separated by periods. If the name of the item itself has periods, they are replaced by the hash-sign ('#'). It is assumed that no item has an hash-sign directly in its name. For example, the fully qualified name of the **String** constructor would be "System.String.#ctor".
- For properties and methods, if there are arguments to the method, the argument list enclosed in parentheses follows. If there are no arguments, no parentheses are present. The arguments are separated by commas. The encoding of each argument follows directly how it is encoded in a .NET Framework signature:
    - Base types. Regular types (ELEMENT_TYPE_CLASS or ELEMENT_TYPE_VALUETYPE) are represented as the fully qualified name of the type.
    - Intrinsic types (for example, ELEMENT_TYPE_I4, ELEMENT_TYPE_OBJECT, ELEMENT_TYPE_STRING, ELEMENT_TYPE_TYPEDBYREF. and ELEMENT_TYPE_VOID) are represented as the fully qualified name of the corresponding full type, for example, **System.Int32** or **System.TypedReference**.
    - ELEMENT_TYPE_PTR is represented as a '*' following the modified type.
    - ELEMENT_TYPE_BYREF is represented as a '@' following the modified type.
    - ELEMENT_TYPE_PINNED is represented as a '^' following the modified type. The Visual C++ compiler never generates this.

- ELEMENT_TYPE_CMOD_REQ is represented as a '|' and the fully qualified name of the modifier class, following the modified type. The Visual C++ compiler never generates this.
- ELEMENT_TYPE_CMOD_OPT is represented as a '!' and the fully qualified name of the modifier class, following the modified type.
- ELEMENT_TYPE_SZARRAY is represented as "[]" following the element type of the array.
- ELEMENT_TYPE_GENERICARRAY is represented as "[?]" following the element type of the array. The Visual C++ compiler never generates this.
- ELEMENT_TYPE_ARRAY is represented as [*lowerbound*:*size*,*lowerbound*:*size*] where the number of commas is the rank - 1, and the lower bounds and size of each dimension, if known, are represented in decimal. If a lower bound or size is not specified, it is simply omitted. If the lower bound and size for a particular dimension are omitted, the ':' is omitted as well. For example, a 2-dimensional array with 1 as the lower bounds and unspecified sizes is [1:,1:].
- ELEMENT_TYPE_FNPTR is represented as "=FUNC:*type*(*signature*)", where *type* is the return type, and *signature* is the arguments of the method. If there are no arguments, the parentheses are omitted. The Visual C++ compiler never generates this.

The following signature components are not represented because they are never used for differentiating overloaded methods:

- calling convention
- return type
- ELEMENT_TYPE_SENTINEL

- For conversion operators only, the return value of the method is encoded as a '~' followed by the return type, as previously encoded.
- For generic types, the name of the type will be followed by a back tick and then a number that indicates the number of generic type parameters. For example,

```
<member name="T:MyClass`2">
```

For a type that is defined as `public class MyClass<T, U>`.

For methods taking generic types as parameters, the generic type parameters are specified as numbers prefaced with back ticks (for example `0,`1). Each number representing a zero-based array notation for the type's generic parameters.

Example

The following examples show how the ID strings for a class and its members would be generated.

```
// xml_id_strings.cpp
// compile with: /clr /doc /LD
///
namespace N {
// "N:N"

   /// <see cref="System" />
   //  <see cref="N:System"/>
   ref class X {
   // "T:N.X"

   protected:
      ///
      !X(){}
      // "M:N.X.Finalize", destructor's representation in metadata

   public:
      ///
      X() {}
      // "M:N.X.#ctor"
```

```
    ///
    static X() {}
    // "M:N.X.#cctor"

    ///
    X(int i) {}
    // "M:N.X.#ctor(System.Int32)"

    ///
    ~X() {}
    // "M:N.X.Dispose", Dispose function representation in metadata

    ///
    System::String^ q;
    // "F:N.X.q"

    ///
    double PI;
    // "F:N.X.PI"

    ///
    int f() { return 1; }
    // "M:N.X.f"

    ///
    int bb(System::String ^ s, int % y, void * z) { return 1; }
    // "M:N.X.bb(System.String,System.Int32@,System.Void*)"

    ///
    int gg(array<short> ^ array1, array< int, 2 >^ IntArray) { return 0; }
    // "M:N.X.gg(System.Int16[], System.Int32[0:,0:])"

    ///
    static X^ operator+(X^ x, X^ xx) { return x; }
   // "M:N.X.op_Addition(N.X,N.X)"

    ///
    property int prop;
    // "M:N.X.prop"

    ///
    property int prop2 {
    // "P:N.X.prop2"

       ///
       int get() { return 0; }
       // M:N.X.get_prop2

       ///
       void set(int i) {}
       // M:N.X.set_prop2(System.Int32)
    }

    ///
    delegate void D(int i);
    // "T:N.X.D"

    ///
    event D ^ d;
    // "E:N.X.d"

    ///
    ref class Nested {};
    // "T:N.X.Nested"

    ///
    static explicit operator System::Int32 (X x) { return 1; }
```

```
        // "M:N.X.op_Explicit(N.X!System.Runtime.CompilerServices.IsByValue)~System.Int32"
    };
}
```

See Also

Other Resources

XML Documentation (Visual C++)

Delimiters for Visual C++ Documentation Tags

The use of documentation tags requires delimiters, which indicate to the compiler where a documentation comment begins and ends.

You can use the following kinds of delimiters with the XML documentation tags:

`///`

This is the form that is shown in documentation examples and used by the Visual C++ project templates.

`/** */`

These are multiline delimiters.

There are some formatting rules when using the `/** */` delimiters:

- For the line that contains the `/**` delimiter, if the remainder of the line is white space, the line is not processed for comments. If the first character is white space, that white space character is ignored and the rest of the line is processed. Otherwise, the entire text of the line after the `/**` delimiter is processed as part of the comment.
- For the line that contains the `*/` delimiter, if there is only white space up to the `*/` delimiter, that line is ignored. Otherwise, the text on the line up to the `*/` delimiter is processed as part of the comment, subject to the pattern-matching rules described in the following bullet.
- For the lines after the one that begins with the `/**` delimiter, the compiler looks for a common pattern at the beginning of each line that consists of optional white space and an asterisk (`*`), followed by more optional white space. If the compiler finds a common set of characters at the beginning of each line, it will ignore that pattern for all lines after the `/**` delimiter, up to and possibly including the line that contains the `*/` delimiter.

Some examples:

- The only part of the following comment that will be processed is the line that begins with `<summary>`. The following two tag formats will produce the same comments:

```
/**
<summary>text</summary>
*/
/** <summary>text</summary> */
```

- The compiler applies a pattern of " * " to ignore at the beginning of the second and third lines.

```
/**
 * <summary>
 *  text </summary>*/
```

- The compiler finds no pattern in this comment because there is no asterisk on the second line. Therefore, all text on the second and third lines, up till the `*/`, will be processed as part of the comment.

```
/**
 * <summary>
   text </summary>*/
```

- The compiler finds no pattern in this comment for two reasons. First, there is no line that begins with a consistent number of spaces before the asterisk. Second, the fifth line begins with a tab, which does not match spaces. Therefore, all text from the second line until the `*/` will be processed as part of the comment.

```
/**
  * <summary>
  * text
```

```
 *  text2
   *  </summary>
*/
```

See Also

Other Resources

XML Documentation (Visual C++)

XDCMake Reference

xdcmake.exe is a program that compiles .xdc files into an .xml file. An .xdc file is created by the Visual C++ compiler for each source code file when source code is compiled with /doc and when the source code file contains documentation comments marked up with XML tags.

To use xdcmake.exe in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see How to: Open Project Property Pages.
2. Open the **Configuration Properties** folder.
3. Click the **XML Document Comments** property page.

Note
xdcmake.exe options at the command line differ from the options when xdcmake.exe is used in the development environment (property pages). For information on using xdcmake.exe in the development environment, see XML Document Generator Tool Property Pages.

Syntax

xdcmake *input_filename options*

Parameters

where:

input_filename

The file name of the .xdc files used as input to xdcmake.exe. Specify one or more .xdc files or use *.xdc to use all .xdc files in the current directory.

options

Zero or more of the following:

Option	Description
/?, /help	Display help for xdcmake.exe.
/assembly:*filename*	Lets you specify the value of the <assembly> tag in the .xml file. By default, the value of the <assembly> tag is the same as the filename of the .xml file.
/nologo	Suppress copyright message.
/out:*filename*	Lets you specify the name of the .xml file. By default, the name of the .xml file is the filename of the first .xdc file processed by xdcmake.exe.

Remarks

Visual Studio will invoke xdcmake.exe automatically when building a project. You can also invoke xdcmake.exe at the command line.

See Recommended Tags for Documentation Comments for more information on adding documentation comments to source code files.

See Also

Other Resources

XML Documentation (Visual C++)

Compatibility and Compliance Issues in Visual C++

- Visual C++ .NET 2003 Enhanced Compiler Conformance
- Breaking Changes in the Visual C++ Compiler
- Nonstandard Behavior

See Also

Other Resources

C++ Language Reference

Visual C++ .NET 2003 Enhanced Compiler Conformance

In the Visual C++ .NET 2003 release, the conformance of the Visual C++ compiler was significantly improved in the following areas:

- Defining Member Templates Outside a Class
- Compiler Correctly Disambiguates Complex Expressions or Types
- Default Initialization of Scalar Types
- Templated User Defined Conversions
- Universal Character Names
- Operator that is Pointer to Function
- Support Aggregate Initialization
- References Supported as Nontype Template Parameters
- Argument Dependent Name (AKA Koenig) Lookup on Functions
- Partial Ordering of Function Templates
- Partial Specialization of Class Templates
- Unicode Support

See Also

Reference

Compatibility and Compliance Issues in Visual C++

Defining Member Templates Outside a Class

In Visual C++ .NET 2003 and later, it is possible to define member templates outside a class, as described in Member Function Templates and Nested Class Templates.

The following sample works in Visual C++ .NET 2003 as specified in the standard:

```
// defining_member_templates_outside_a_class.cpp
// compile with: /LD
template <class T>
struct S
{
   template<class U> void f(U);
};

template<class T> template <class U> void S<T>::f(U)
{   //defined out of line
}
```

See Also

Other Resources

Visual C++ .NET 2003 Enhanced Compiler Conformance

Compiler Correctly Disambiguates Complex Expressions or Types

The following sample works in Visual C++ .NET 2003 as specified in the standard:

```
// compiler_correctly_disambiguates_complex_expressions_or_types.cpp
// Many different variations, this is a simple one
#include <typeinfo>
typedef void functype();
int main()
{
   return !(typeid(functype) == typeid(void()));
   //now these match correctly
}
```

See Also

Other Resources

Visual C++ .NET 2003 Enhanced Compiler Conformance

Default Initialization of Scalar Types

The following sample works in Visual C++ .NET 2003 as specified in the standard:

Example

```
// default_initialization_of_scalar_types.cpp
struct S
{
    int i0;
    int i1;
    S() : i1()   // i1 default initialized to 0, i0 value indeterminate
    {
    }
};

extern "C" int printf_s(const char*, ...);

int main()
{
    // points to unitialized int (value at *pi0 is indeterminate)
    int *pi0 = new int;

    // pointer to a default initialized int (*pi1 == 0)
    int *pi1 = new int();
    S s;
    printf_s("These variables should both be zero: %d and %d\n",
             *pi1, s.i1);
    delete pi0;
    delete pi1;
}
```

Output

```
These variables should both be zero: 0 and 0
```

See Also

Other Resources

Visual C++ .NET 2003 Enhanced Compiler Conformance

Templated User Defined Conversions

In Visual C++ .NET 2003 and later, it is possible to define user defined conversion operators as template member functions. For more information, see Member Function Templates.

See Also

Other Resources

Visual C++ .NET 2003 Enhanced Compiler Conformance

Universal Character Names

The following sample works in Visual C++ .NET 2003 as specified in the standard:

```
// universal_character_names.cpp
int main()
{
   int IdentContainingTwoUCNCharacters\u1234\U00001234 = 0;
}
```

For more information, see Unicode Support.

See Also

Other Resources

Visual C++ .NET 2003 Enhanced Compiler Conformance

Operator That is Pointer to Function

The following sample works in Visual C++ .NET 2003 as specified in the standard:

```
// operator_that_is_pointer_to_function.cpp
// function style call on object will invoke user-defined conversion
// if there is one. See secion 13.3.1.1.2
typedef void(*ptf)();
void func()
{
}
struct S
{
   operator ptf()
   {
      return func;
   }
};

int main()
{
   S s;
   s();//operates as s.operator ptf()()
}
```

See Also

Other Resources

Visual C++ .NET 2003 Enhanced Compiler Conformance

Support Aggregate Initialization

The following sample works in Visual C++ .NET 2003 as specified in the standard:

```
// support_aggregate_initialization1.cpp
struct NotAgg
{
   NotAgg(int)
   {
   }
};

struct Agg
{
   NotAgg mem;
};

int main()
{
   Agg anAggregate = { NotAgg(1) };
}
```

The following sample works in Visual C++ .NET 2003 as specified in the standard:

```
// support_aggregate_initialization2.cpp
// compile with: /EHsc
#include <string>
#include <iostream>
using namespace std;
struct MyStruct
{
   string strA;
   string strB;
} myStrings[2] = { {"a", "b"}, {"c", "d"} };

int main()
{
    cout << myStrings[0].strB << endl;
}
```

See Also

Other Resources

Visual C++ .NET 2003 Enhanced Compiler Conformance

References Supported as Nontype Template Parameters

In Visual C++ .NET 2003 and later, references may be declared as template parameters. For more information, see Template Specifications.

See Also

Other Resources

Visual C++ .NET 2003 Enhanced Compiler Conformance

Argument-Dependent Name (Koenig) Lookup on Functions

The compiler can use argument-dependent name lookup to find the definition of an unqualified function call. Argument-dependent name lookup is also called Koenig lookup. The type of every argument in a function call is defined within a hierarchy of namespaces, classes, structures, unions, or templates. When you specify an unqualified postfix function call, the compiler searches for the function definition in the hierarchy associated with each argument type.

Example

The following sample works in Visual C++ .NET 2003 and later, and behaves as specified in the C++ standard. For a more comprehensive example, see Argument-Dependent (Koenig) Lookup Now Supported. In the sample, the compiler notes that function `f()` takes an argument `x`. Argument `x` is of type `A::X`, which is defined in namespace `A`. The compiler searches namespace `A` and finds a definition for function `f()` that takes an argument of type `A::X`.

```
// argument_dependent_name_koenig_lookup_on_functions.cpp
namespace A
{
   struct X
   {
   };
   void f(const X&)
   {
   }
}
int main()
{
// The compiler finds A::f() in namespace A, which is where
// the type of argument x is defined. The type of x is A::X.
   A::X x;
   f(x);
}
```

See Also

Other Resources

Visual C++ .NET 2003 Enhanced Compiler Conformance

Partial Ordering of Function Templates in Visual C++ .NET 2003

Visual C++ .NET 2003 and later versions provide support for partial ordering of function templates. For complete information, see Partial Ordering of Function Templates.

See Also

Other Resources

Visual C++ .NET 2003 Enhanced Compiler Conformance

Partial Specialization of Class Templates in Visual C++ .NET 2003

Visual C++ .NET 2003 and later versions provide support for partial ordering of function templates. For complete information, see Partial Specialization of Class Templates.

See Also

Other Resources

Visual C++ .NET 2003 Enhanced Compiler Conformance

Unicode Support

In this version of the compiler, Unicode characters that cannot be represented in the current system code page are not allowed. The compiler's support for Unicode characters and universal-character-names, as specified in ISO/ANSI Standard, is as follows:

- Comments
- String and character literals

For more information, see Universal Character Names.

Additional Visual C++ Unicode support is described in Unicode Support in the Compiler and Linker.

See Also

Other Resources

Visual C++ .NET 2003 Enhanced Compiler Conformance

Breaking Changes in the Visual C++ Compiler

This section discusses the breaking changes made to the Visual C++ compiler and language in the current and previous releases:

- Breaking Changes in Visual C++ .NET 2003
- Breaking Changes in the Visual C++ 2005 Compiler

See Also

Reference

Compatibility and Compliance Issues in Visual C++

Breaking Changes in Visual C++ .NET 2003

The Visual C++ compiler has significantly increased its conformance with the C++ standard in the Visual Studio .NET 2003 release. With this conformance comes a number of breaking changes. Many of those changes now result in compiler errors, where no compiler errors were given in previous releases. However, there are also a number of silent, or runtime errors, which are described in this section.

A nonconformance related breaking change is that just-in time debugging for **/clr** images will not work unless you either add DebuggableAttribute or use a configuration (.ini) file; see Making an Image Easier to Debug for more information.

For remaining conformance issues, see Standard Compliance Issues in Visual C++.

- Throwing New Behavior Changed
- String Literals Have Proper Type of const char[]
- Overloading of Function Templates
- Argument Dependent (Koenig) Lookup Now Supported
- Zero Initialization of POD and Scalar Types
- Old-Style Name Mangling for Exported Template Instantiations
- Temporary Objects Cannot be Bound to Non-Const References
- Summary of Compile-Time Breaking Changes

See Also
Reference
Breaking Changes in the Visual C++ Compiler

Throwing New Behavior Changed

A change that occurred in the Visual C++ .NET 2002 compiler (version 13.00.9466) was that the Standard C++ Library's new header file would now throw an exception when a call to new failed.

In Visual C++ 2005, the compiler will, by default, throw an exception when any call to new fails.

For more information, see The new and delete Operators.

See Also

Reference

Breaking Changes in the Visual C++ Compiler

String Literals Have Proper Type of const char[]

String literals now have the type const char [] and are now placed in a read-only section of memory. Changing that memory will now cause an access violation. Code compiled in previous versions using /GF will also cause the access violation.

The following sample compiles and runs in Visual Studio .NET but will fail at run time in Visual Studio .NET 2003:

```
// bc_string_literals_have_proper_type_of_const_char.cpp
// compile with: /c
void f(char *c) {
   c[0] = 'Q';   // Now gives run-time access violation
}

int main() {
   f("TEST");
}
```

In addition, the run-time behavior of code such as the following example has now changed:

```
// bc_string_literals_have_proper_type_of_const_char2.cpp
#include "stdio.h"
void f(const char *) {   // called in Visual Studio .NET 2003
   printf_s("in f(const char *)\n");
}

void f(char *) {   // called in Visual Studio .NET
   printf_s("in f(char *)\n");
}

int main() {
   f("TEST");
}
```

To resolve this error, do not pass string literals to functions where they will be modified.

For code that will be valid in the current and previous versions of Visual C++ in the case where functions are overloaded in this manner:

- Explicitly cast string literals to const char*.
- Define variables on the stack or the heap.

The following code will be valid in the Visual Studio .NET 2003 and Visual Studio .NET versions of Visual C++, causing an access violation in neither:

```
// bc_string_literals_have_proper_type_of_const_char3.cpp
#include <stdio.h>
void f(const char *psz) {
   printf_s("const version\n");
}

void f(char *psz) {
   printf_s("version where we modify it\n");
   psz[0] = 'x';
}

int main() {
   char myStr[] = "TEST";
   f((const char*)"TEST");
   f(myStr);
}
```

See Also

Reference

Breaking Changes in the Visual C++ Compiler

Overloading of Function Templates

In Visual Studio .NET, the compiler treated functions whose signature mapped to an explicit specialization of a template function (even if the function was not preceded by `template<>`) as a specialization. Now, such functions are treated as non-template overloads.

Run-time behavior may change in cases like:

```
// bc_overloading_of_function_templates.cpp
#include <stdio.h>
template<class T>
void f(T)   // called in Visual Studio .NET 2003
{
    printf_s("in void f(T)\n");
}

void f(int)   // called in Visual Studio .NET
// for identical behavior for both compiler versions, use
// template<> void
// f<int>(int)
{
    printf_s("in void f(int)\n");
}

int main()
{
    f<int>(3);
    // Visual C++ .NET calls template function specialization
    // because explicit template arguments were provided.
    // The current compiler will also call specialization because
    // explicit template arguments were provided.
    // But these will call different functions because the previous
    // compiler explicitly specializes on int, and the current
    // compiler does not (creates non-template overload)

    f(4);
    // Visual C++ .NET will call template function specialization
    // because no global non-template overload defined.
    // The current compiler will call the non-template overload.
}
```

For the previous example, note that identical behavior can be achieved by making `f(int)` an explicit specialization instead of what should be an overload. The specialization will be called in both the Visual Studio .NET 2003 and Visual Studio .NET versions of Visual C++.

See Also

Reference

Breaking Changes in the Visual C++ Compiler

Argument-Dependent (Koenig) Lookup Now Supported

Koenig lookup is now fully implemented in the Visual C++ compiler.

For more information, see Argument-Dependent Name (Koenig) Lookup on Functions.

Example

The following sample behaves differently at run time in Visual Studio .NET than it does in Visual Studio .NET 2003

```
// bc_argument_dependent_AKA_Koenig_lookup_now_supported.cpp
// compile with: /W1
#include <stdio.h>

namespace N {
   class X {};

   void f(X *pX) {
      printf_s("in N::X::f\n");
   };   // called if compiled with 7.1
}

void f(void *pv) {
   printf_s("in ::f\n");
};   // called if compiled with 7.0

int main() {
   N::X *pX = 0;
   f(pX);

   // The following lines will result in the same behavior
   // in Visual Studio .NET or Visual Studio .NET 2003
   f((void*)pX);   // Cast pX to void before calling f; calls global f
   ::f(pX);    // Explicitly specify global f
   N::f(pX);   // Explicitly specify f in namespace
}
```

Output

```
in N::X::f
in ::f
in ::f
in N::X::f
```

See Also

Reference

Breaking Changes in the Visual C++ Compiler

Zero Initialization of POD and Scalar Types

POD and scalar types will always be zero initialized if instantiated with the default constructor syntax.

```
struct S {
   void *operator new (unsigned int size, void*p, int i)
   {
      ((S*)p)->i = i; return p;
   }

   int i;
};

struct T
{
   int i;
   char c;
};

class C
{
   T t;
   int i;
public:
   C(): t(), i() {}   // Zero initializes members of class.
};

// Zero initialize members of t.
// t->i == 0 & t->c == 0
T* t = new T();

// Call placement new operator for S, then
// zero initialize members of pS.
// pS->i == 0 & pS->i != 10
S s;
S* pS = new( &s, 10 ) S();

// Zero initialize *pI
// *pI == 0
int* pI = new int();

// Zero initialize members of c
// c.t.i == 0 & c.t.c == 0 & c.i == 0
C c;
```

The Visual Studio .NET behavior ignores the () parentheses after the initialization and will always leave the members uninitialized. To revert to the Visual Studio .NET behavior for types initialized outside of a constructor's initialization list, remove the () parentheses as follows:

```
T* t = new T;   // Members contain uninitialized data.
S s;
S* pS = new( &s, 10 );   // pS->i == 10
int* pI = new int;   // *pI is uninitialized.
```

To revert to the Visual Studio .NET behavior for types initialized inside of a constructor's initialization list, remove initialization from the list as follows.

```
class C
{
   T t;
   int i;
public:
   C() {}   // Members of class are not initialized.
```

```
};
```

See Also

Reference

Breaking Changes in the Visual C++ Compiler

Old-Style Name Mangling for Exported Template Instantiations

One aspect of the Visual C++ conformance work is to allow the overloading of specializations of function templates and regular templates.

For example, the following sample compiles in Visual Studio .NET 2003 but will fail in Visual Studio .NET:

```
// bc_exported_templ_instan.cpp
template<typename T, typename U>
T ConvertTo(const U &u) {
   return static_cast<T>(u);
}

char ConvertTo(const int &i) {
   return static_cast<char>(i);
}

int main() {
   char c1 = ConvertTo(1);
   char c2 = ConvertTo<char, int>(2);
}
```

To support the overloading of specializations of function templates and regular templates, the compiler changed how it creates decorated names for specializations of function templates. An updated decorated name now includes encoding of the template arguments, as well as encoding of the function parameter and return types. For the functions in the previous example, the C++ compiler in Visual Studio .NET 2003 will generate the following name decorations:

```
?ConvertTo@@YADABH@Z      – char ConvertTo(const int &);
??$ConvertTo@DH@@YADABH@Z – char ConvertTo<char, int>(const int &);
```

This should not imply any change in the way that you develop your applications. One area where this change does affect users is if the export is a specialization of a function template from a DLL used in an application that has not been recompiled with the new compiler. For example:

```
// bc_exported_templ_instan2.h
#include <iostream>
#ifdef _DLL_EXPORT
#define DLL_LINKAGE __declspec(dllexport)
#else
#define DLL_LINKAGE __declspec(dllimport)
#endif

template<typename T>
void f(const T &rT) {
   std::cout << "i = " << rT << std::endl;
}
template DLL_LINKAGE void f<int>(const int &);
```

The following code will be compiled to create the DLL:

```
// bc_exported_templ_instan2.cpp
// compile with: /D_DLL_EXPORT /EHsc /LD
#include "bc_exported_templ_instan2.h"
```

If you then do:

```
dumpbin /exports bc_exported_templ_instan2.dll
```

You see that the decorated name of the exported specialization of the function template is `??$f@H@@YAXABH@Z`.

If an existing application has a dependency on this DLL and this application was built with the Visual C++ compiler in Visual Studio .NET (or an earlier version) and they cannot or do not want to rebuild this application, then they will get a run-time error, because the application will be expecting an entry point with the old name. However, the new DLL will only be exporting a function with the new name.

```
// bc_exported_templ_instan3.cpp
// compile with: /EHsc /link bc_exported_templ_instan2.lib
#include "bc_exported_templ_instan2.h"
int main() {
   f(1);
}
```

The fix for this problem is to change how the DLL is built so it will export both the old and new name for the specialization of the function template. This is done by using /export to the link program. For example:

```
cl /D_DLL_EXPORT /EHsc /LD a.cpp /link /export:?f@@YAXABH@Z=??$f@H@@YAXABH@Z
```

This tells the link to create an export for both the specified names but to have them map to the same symbol.

See Also

Reference

Breaking Changes in the Visual C++ Compiler

Temporary Objects Cannot be Bound to Non-Const References

In previous releases of Visual C++, non-const references could be bound to temporary objects. Now, temporary objects can only be bound to const references.

Example

For example, the following sample compiles has different run-time behavior in Visual Studio .NET 2003 compared to Visual Studio .NET:

```
// bc_temp_objects_not_bound_to_nonconst_ref.cpp
// compile with: /EHsc
#include "iostream"
using namespace std;
class C {};

void f(C & c) { cout << "C&" << endl; }
void f(C const & c) { cout << "C const &" << endl; }

int main() {
   f(C());
}
```

Output

```
C const &
```

See Also

Reference

Breaking Changes in the Visual C++ Compiler

Summary of Compile-Time Breaking Changes

This topic summarizes the compile-time errors and warnings that will now be issued on code that compiled without errors or warnings prior to Visual C++ 2003. However, some conformance-related, compile-time breaking changes were introduced in Visual C++ 2005.

- Closing parentheses now required for the defined preprocessor directive (C2004).
- Missing typename keyword and explicit specializations no longer find template parameters from primary template (C2146).
- A protected member (n) can only be accessed through a member function of a class (B) that inherits from the class (A) of which it (n) is a member (C2247).
- Improved accessibility checks in compiler now detect inaccessible base classes (C2247).
- An explicit specialization can not be a copy constructor or copy assignment operator (C2299).
- An exception cannot be caught if the destructor and/or copy constructor is inaccessible (C2316).
- Default arguments on pointers to functions no longer allowed (C2383).
- A static data member cannot be initialized via derived class (C2477).
- The initialization of a typedef is not allowed by the standard and now generates a compiler error (C2513).
- **bool** is now a proper type (C2632).
- A UDC can now create ambiguity with overloaded operators (C2666).
- A qualification conversion is not the same as identity conversion (C2666).
- More expressions are now considered valid null pointer constants (C2668).
- `template<>` is now required in places where the compiler would previously infer it (C2768).
- The explicit specialization of a member function outside the class is not valid if the function has already been explicitly specialized via a template class specialization. (C2910).
- For every explicit specialization, you must chose the parameters of the every explicit specialization, such that, they match the parameters of the primary template (C2912).
- Floating point non-type template parameters are no longer allowed (C2993).
- Class templates are not allowed as template type arguments (C3206).
- Friend function names are no longer introduced into containing namespace (C3767).
- Enhanced pointer-to-member conformance (C3867).
- The compiler will no longer accept extra commas in a macro (C4002).
- An object of POD type constructed with an initializer of the form () will be default-initialized (C4345).
- `typename` is now required if a dependent name is to be treated as a type (C4346).
- Functions that were incorrectly considered template specializations are no longer considered so (C4347).
- Static data members cannot be initialized via derived class (C4356).
- A class template specialization needs to be defined before it was used in a return type (C4686).
- The compiler now reports unreachable code (C4702).
- A friend declaration that declares a new non-template function must be defined (LNK2019).

See Also

Reference

Breaking Changes in the Visual C++ Compiler

Breaking Changes in the Visual C++ 2005 Compiler

This topic discusses the behavior changes in Visual C++ 2005 that can cause code that worked in a previous release to either not compile, or to behave differently at run time.

For more information on new features, see What's New in Visual C++, What's New in the Visual C++ Libraries, and What's New in Visual C++ Compiler, Language, and Tools.

Pointer-to-members now require qualified name and &

Code written for previous versions of the compiler that just used the method name will now give Compiler Error C3867 or Compiler Warning C4867. This diagnostic is required by Standard C++. Now, to create a pointer to a member function, the address of operator (&) must be used with the fully qualified name of the method. Not having to use the & operator and the fully qualified name of the method can lead to logic bugs in code due missing parentheses in function calls. Using the function's name without an argument list results in a function pointer which is convertible to several types. This code would have compiled, leading to unexpected behavior at runtime.

Class must be accessible to a friend declaration

Visual C++ compilers prior to Visual C++ 2005 allowed a friend declaration to a class that was not accessible in the scope of the class containing the declaration. Now, the compiler will give Compiler Error C2248. To resolve this error, change the accessibility of the class specified in the friend declaration. This change was made to comply with the C++ standard.

__int asm 3 now compiles to native

When compiled with **/clr**, `__asm int 3` did not cause native code to be generated; the compiler translated the instruction to a CLR break instruction. In Visual C++ 2005, `__asm int 3` now results in native code generation for the function. If you want a function to cause a break point in your code and if you want that function compiled to MSIL, use __debugbreak. For more information, see __asm and /clr (Common Language Runtime Compilation). This change was made to be more deterministic about when to generate native code versus managed code; inline assembly code should generate native code.

Explicit specialization not allowed as a copy constructor/copy assignment operator

Code that depends on an explicit template specialization for a copy constructor or copy assignment operator will now get Compiler Error C2299. Standard C++ prohibits this. This change was made for conformance reasons, to make code more portable.

Unspecialized class template can't be used as a template argument in a base class list

Using an unspecialized template class name in the base class list for a class definition will result in Compiler Error C3203. It is illegal to use an unspecialized template class name as a template parameter in a base class list. Explicitly add the template type parameters to the template class name when using it as a template parameter in a base class list. This change was made for conformance reasons, to make code more portable.

A using declaration of nested type no longer allowed

Code that has a using declaration to a nested type will now generate Compiler Error C2885. To resolve fully qualify references to nested types, put the type in a namespace, or create a typedef. This change was made for conformance reasons, to make code more portable.

Compiler does not allow **const_cast** to down cast under **/clr:oldSyntax**

Prior to Visual C++ 2005 the Visual C++ compiler allowed the const_cast Operator to down cast when compiling source code that uses Managed Extensions for C++ syntax. Performing a down cast with **const_cast** now results in Compiler Error C2440. To resolve, use the correct cast operator (for more information, see Casting Operators). This change was made for conformance reasons.

Compiler disallows forward declaration of a managed enum

Prior to Visual C++ 2005 the Visual C++ compiler allowed forward declarations of managed enums. Now, declaring and not defining a managed enum when compiling with any form of **/clr** will result in Compiler Error C2599. To resolve, always define managed enums at declaration. This change was made because forward declarations of managed enums were not always guaranteed to work correctly: the compiler cannot correctly identify the underlying type of the enum. Also the C++ Standard does not allow enum declarations.

/YX compiler option is removed

/YX generated automatic pre-compiled headers support. It was used by default from the development environment. If you

remove **/YX** from your build configurations and replace it with nothing, it can result in faster builds. In addition to the possibility of unexpected behavior with **/YX**, it is preferable to use /Yc (Create Precompiled Header File) and /Yu (Use Precompiled Header File), which give you more control on how precompiled headers are used.

/Oa and **/Ow** compiler options are removed

/Ow and **/Oa** compiler options have been removed but will be silently ignored. Use the noalias or restrict **__declspec** modifiers to specify how the compiler does aliasing.

/Op compiler option is removed

/Op compiler option had been removed. Use /fp (Specify Floating-Point Behavior) instead.

/ML and **/MLd** compiler options have been removed

Visual C++ no longer supports single-threaded, statically linked CRT library support. Use **/MT** and **/MTd** instead. See C Run-Time Libraries for more information.

/G3, **/G4**, **/G5**, **/G6**, **/G7**, and **/GB** compiler options have been removed

The compiler now uses a blended model that attempts to create the best output file for all architectures.

/Gf has been removed

Use /GF (Eliminate Duplicate Strings) instead. **/GF** puts pooled strings into a read-only section, which is safer than the writeable section, where **/Gf** added them.

/clr is not compatible with **/MT**

There is no support in the C Runtime Library to statically link to a managed application. All managed applications have to be dynamically linked (/MD). For more information on restrictions on using **/clr**, see /clr Restrictions.

/GS is now on by default

Buffer overflow checking is now on by default. You can turn buffer overrun checking off with **/GS-**. See /GS (Buffer Security Check) for more information.

/Zc:wchar_t now on by default

This is Standard C++ behavior; a **wchar_t** variable will default to the built in type instead of a short unsigned integer. This change will break binary compatibility when client code is linked with libraries that were compiled without **/Zc:wchar_t** (LNK2019). In that case, use **/Zc:wchar_t-** to revert to the old, non-standard behavior. This change was introduced to create conformant code by default.

For more information, see /Zc:wchar_t (wchar_t Is Native Type).

/Zc:forScope now on by default

This is Standard C++ behavior; code that depends on the use of a variable declared in a for loop after the for loop scope has ended will now fail to compile. Use **/Zc:forScope-** to revert to the old, non-standard behavior. This change was introduced to create conformant code by default.

For more information, see /Zc:forScope (Force Conformance in for Loop Scope).

Enforce parameter checking for Visual C++ attributes

Code that passes named attributes to the attribute constructor in quotes when the type is not a string and without quotes when the type is a string will now give Compiler Error C2065 or Compiler Warning (level 1) C4581. Previously all compiler attributes were parsed as strings, and if needed, the compiler inserted the missing quotes. Attribute support was enhanced by adding parameter checking validation. This will prevent unexpected behavior due to incorrect arguments to an attribute constructor.

For this release you cannot have a multi-byte character string (MBCS) in any argument to an attribute which takes an implicit string as an argument, even if the string is quoted (doing so can result in a corrupt .idl file). The workaround is as follows:

```
#define ARG string_with_MBCS_chars
[helpstring(ARG)]
```

Compiler now requires same template specification for multiple declarations of the same type.

If you have a forward declaration of a type so that you can create friends to that type, for example, the template specification of the type must be the same on all declarations for the type. Otherwise, the compiler will issue Compiler Error C2990.

uuid attribute can no longer target managed types

The uuid (C++ Attributes) attribute was allowed on a user-defined attribute using Managed Extensions for C++, but will now generate Compiler Error C3451. Use GuidAttribute instead.

Syntax change for passing managed arrays to custom attributes

The type of the array is no longer deduced from the aggregate initialization list. The compiler now requires you to specify the type of the array as well as the initializer list. The old syntax will now result in Compiler Error C3104. This change was required because the compiler could not always correctly deduce the array type from the aggregate initialization list.

Compiler will not inject int as the default type in declarations

Code that is missing the type in a declaration will no longer default to type int the compiler will generate Compiler Warning C4430 or Compiler Warning (level 4) C4431. Standard C++ does not support a default int and this change will ensure that you get the type you really want.

dynamic_cast has enhanced conformance to the C++ standard.

The C runtime library now does a dynamic_cast runtime check to ensure the compile-time type of the expression being cast refers to a public base class sub-object of either the cast target type (for down-cast) or most-derived object type (for cross-cast). For more information, see Breaking Changes in dynamic_cast.

An rvalue cannot be bound to a non-const reference.

An rvalue cannot be bound to a non-const reference. In previous versions of Visual C++, it was possible to bind an rvalue to a non-const reference in a direct initialization. This code now generates Compiler Warning (level 1) C4350.

Value types no longer have a default constructor emitted, this can cause type initializers to run at different points

Prior to Visual C++ 2005 static constructors (type initializers) in value types were run when an instance of the value type was created. To ensure that static constructors are run, access a static data member or (**/clr:oldSyntax** only) define an instance constructor. Not providing a default constructor for value types was done because the common language runtime does not guarantee it will always call a default constructor. Also, not providing a default constructor for value types improves performance.

Boxed value types are now read only in verifiable (**/clr:safe**) contexts.

The common language runtime no longer allows modifying a boxed value type when compiling a verifiable assembly. The compiler now gives Compiler Warning C4972 when this is detected.

C4792 is only given if you change the value of the underlying value object via boxed value object. The error will not occur if you change a copy of the value object (for example, changing a boxed object)

Native types are private by default outside the assembly

Native types now will not be visible outside the assembly by default. For more information on type visibility outside the assembly, see Type Visibility. This change was primarily driven by the needs of developers using other, case-insensitive languages, when referencing metadata authored in Visual C++.

/clr now accepts new CLR syntax for Visual C++

Prior to Visual C++ 2005, **/clr** compiled Managed Extensions for C++ syntax. **/clr** now compiles the new CLR syntax and **/clr:oldSyntax** compiles Managed Extensions for C++ syntax. For more information on **/clr**, see /clr (Common Language Runtime Compilation). For more information on the new syntax, see New Language Features in Visual C++.

/clr no longer compiles C source code files

Prior to Visual C++ 2005, you could compile C source code files with **/clr**, however this will now result in Command-Line Error D8045. To resolve, change the file extension to .cpp or .cxx, or compile with **/TP** or **/Tp**. See /Tc, /Tp, /TC, /TP (Specify Source File Type) for more information.

MSIL changes when testing for equality

See How to: Test for Equality for more information.

See Also

Reference

Breaking Changes in the Visual C++ Compiler

Nonstandard Behavior

The following topics are some of the known places where the Visual C++ implementation of C++ does not agree with the C++ standard. The section numbers refer to section numbers in the C++ standard.

- Compiler Limits
- ")10.3 (Paragraph 5) Covariant Return Types
- 14 export Keyword on a Template
- 14.6.2 Dependent Names
- 15.4 Function Exception Specifiers
- 16.3.2 The # Operator
- 21.1.1 Character Traits Requirements
- Storage Location of Objects

See Also

Reference

Compatibility and Compliance Issues in Visual C++

Compiler Limits

The C++ standard recommends limits for various language constructs. The following is a list of constructs where the Visual C++ compiler does not implement the recommended limits. The first number is the recommended limit and the second number is the limit implemented by Visual C++:

- Nesting levels of compound statements, iteration control structures, and selection control structures [256] (64).
- Parameters in one macro definition [256] (127).
- Arguments in one macro invocation [256] (127).
- Characters in a character string literal or wide string literal (after concatenation) [65536] (65535).
- Levels of nested class, structure, or union definitions in a single *struct-declaration-list* [256] (16).
- Member initializers in a constructor definition [6144] (approximately 600, memory dependent, can increase with the /Zm compiler option).
- Scope qualifications of one identifier [256] (127).
- Nested external specifications [1024] (10).
- Template arguments in a template declaration [1024] (64).

See Also

Other Resources

Nonstandard Behavior

10.3 (Paragraph 5) Covariant Return Types

Virtual base classes are not supported as covariant return types when the virtual function has a variable number of arguments.

For example, the following sample will not compile:

```
// CovariantReturn.cpp
class A
{
   virtual A* f(int c, ...);   // remove ...
};

class B : virtual A
{
   B* f(int c, ...);   // C2688 remove ...
};
```

See Also

Other Resources

Nonstandard Behavior

14 export Keyword on a Template

The export keyword is not supported on templates. For example, the following sample will not compile:

```
export template <class T> void fun(T);
export template <class T> class A;
```

See Also

Other Resources

Nonstandard Behavior

14.6.2 Dependent Names

The Visual C++ compiler does not currently support binding nondependent names when initially parsing a template. This can cause overloads to be declared after the template (but before the template is instantiated) to be seen.

```
// DependentNames.cpp
#include <stdio.h>
namespace N {
   void f(int) { printf("f(int)\n");}
}

template <class T> void g(T) {
   N::f('a');   // calls f(char) should call f(int)
}

namespace N {
   void f(char) { printf_s("f(char)\n");}
}

int main() {
   g('c');
}
```

Output

```
f(char)
```

See Also

Other Resources

Nonstandard Behavior

15.4 Function Exception Specifiers

Function exception specifiers other than **throw()** are parsed but not used. For example:

```
void f() throw(int); // parsed but not used
void g() throw();    // parsed and used
```

For more information on exception specifications, see Exception Specifications.

See Also

Other Resources

Nonstandard Behavior

16.3.2 The # Operator

Visual C++ does not behave correctly when the # (stringize) operator is used with strings that include escape sequences. In this situation, the compiler will generate Compiler Error C2017.

See Also

Other Resources

Nonstandard Behavior

21.1.1 Character Traits Requirements

The C++ standard states that the value returned by char_traits::eof must not be a valid character value. The Visual C++ compiler enforces this constraint for **char_type<char>::eof**, but not for **char_type<wchar_t>::eof**.

See Also

Other Resources

Nonstandard Behavior

Storage Location of Objects

The C++ standard requires complete C++ objects to have unique storage locations. However with Visual C++, there are cases where types without data members will share a storage location with other types for the lifetime of the object.

See Also

Other Resources

Nonstandard Behavior

Lexical Conventions

This section introduces the fundamental elements of a C++ program. You use these elements, called "lexical elements" or "tokens" to construct statements, definitions, declarations, and so on, which are used to construct complete programs. The following lexical elements are discussed in this section:

- Tokens
- Comments
- Identifiers
- Keywords
- Punctuators
- Operators
- Literals
- Not a Number (NAN) Items

This section also has a topic on the precedence and associativity of C++ operators. For a complete discussion of operators, see Expressions.

See Also

Other Resources

C++ Language Reference

Overview of File Translation

C++ programs, like C programs, consist of one or more files. Each of these files is translated in the following conceptual order (the actual order follows the "as if" rule: translation must occur as if these steps had been followed):

1. Lexical tokenizing. Character mapping and trigraph processing, line splicing, and tokenizing are performed in this translation phase.

2. Preprocessing. This translation phase brings in ancillary source files referenced by **#include** directives, handles "stringizing" and "charizing" directives, and performs token pasting and macro expansion (see Preprocessor Directives in the *Preprocessor Reference* for more information). The result of the preprocessing phase is a sequence of tokens that, taken together, define a "translation unit."

   Preprocessor directives always begin with the number-sign (**#**) character (that is, the first nonwhite-space character on the line must be a number sign). Only one preprocessor directive can appear on a given line. For example:

   ```
   #include <iostream>  // Include text of iostream in
                        //  translation unit.
   #define NDEBUG       // Define NDEBUG (NDEBUG contains empty
                        //  text string).
   ```

3. Code generation. This translation phase uses the tokens generated in the preprocessing phase to generate object code.

   During this phase, syntactic and semantic checking of the source code is performed.

See Phases of Translation in the *Preprocessor Reference* for more information.

The C++ preprocessor is a strict superset of the ANSI C preprocessor, but the C++ preprocessor differs in a few instances. The following list describes several differences between the ANSI C and the C++ preprocessors:

- Single-line comments are supported. See Comments for more information.
- One predefined macro, **__cplusplus**, is defined only for C++. See Predefined Macros in the *Preprocessor Reference* for more information.
- The C preprocessor does not recognize the C++ operators: **.***, **–>***, and **::**. See Operators and Expressions, for more information about operators.

See Also
Reference
Lexical Conventions

C++ Tokens

A token is the smallest element of a C++ program that is meaningful to the compiler. The C++ parser recognizes these kinds of tokens: identifiers, keywords, literals, operators, punctuators, and other separators. A stream of these tokens makes up a translation unit.

Tokens are usually separated by "white space." White space can be one or more:

- Blanks
- Horizontal or vertical tabs
- New lines
- Formfeeds
- Comments

The following are considered tokens:

keywordidentifierconstantoperatorpunctuator

The following are considered preprocessing tokens:

header-nameidentifierpp-numbercharacter-constantstring-literaloperatorpunctuator each nonwhite-space character that cannot be one of the above

The parser separates tokens from the input stream by creating the longest token possible using the input characters in a left-to-right scan. Consider this code fragment:

```
a = i+++j;
```

The programmer who wrote the code might have intended either of these two statements:

```
a = i + (++j)

a = (i++) + j
```

Because the parser creates the longest token possible from the input stream, it chooses the second interpretation, making the tokens `i++`, `+`, and `j`.

See Also

Reference

Lexical Conventions

C++ Comments

A comment is text that the compiler ignores but that is useful for programmers. Comments are normally used to annotate code for future reference. The compiler treats them as white space. You can use comments in testing to make certain lines of code inactive; however, **#if**/**#endif** preprocessor directives work better for this because you can surround code that contains comments but you cannot nest comments.

A C++ comment is written in one of the following ways:

- The `/*` (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the `*/` characters. This syntax is the same as ANSI C.
- The `//` (two slashes) characters, followed by any sequence of characters. A new line not immediately preceded by a backslash terminates this form of comment. Therefore, it is commonly called a "single-line comment."

The comment characters (`/*`, `*/`, and `//`) have no special meaning within a character constant, string literal, or comment. Comments using the first syntax, therefore, cannot be nested.

See Also
Reference
Lexical Conventions

C++ Identifiers

An identifier is a sequence of characters used to denote one of the following:

- Object or variable name
- Class, structure, or union name
- Enumerated type name
- Member of a class, structure, union, or enumeration
- Function or class-member function
- typedef name
- Label name
- Macro name
- Macro parameter

The following characters are legal as the first character of an identifier, or any subsequent character:

```
_ a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
```

The following characters are legal as any character in an identifier except the first:

```
0 1 2 3 4 5 6 7 8 9
```

Microsoft Specific

Only the first 2048 characters of Microsoft C++ identifiers are significant. Names for user-defined types are "decorated" by the compiler to preserve type information. The resultant name, including the type information, cannot be longer than 2048 characters. (See Decorated Names for more information.) Factors that can influence the length of a decorated identifier are:

- Whether the identifier denotes an object of user-defined type or a type derived from a user-defined type.
- Whether the identifier denotes a function or a type derived from a function.
- The number of arguments to a function.

The dollar sign is also a valid identifier in Visual C++.

```
// dollar_sign_identifier.cpp
struct $Y1$ {
   void $Test$() {}
};

int main() {
   $Y1$ $x$;
   $x$.$Test$();
}
```

END Microsoft Specific

The first character of an identifier must be an alphabetic character, either uppercase or lowercase, or an underscore (_). Because C++ identifiers are case sensitive, `fileName` is different from `FileName`.

Identifiers cannot be exactly the same spelling and case as keywords. Identifiers that contain keywords are legal. For example, `Pint` is a legal identifier, even though it contains **int**, which is a keyword.

Use of two sequential underscore characters (__) at the beginning of an identifier, or a single leading underscore followed by

a capital letter, is reserved for C++ implementations in all scopes. You should avoid using one leading underscore followed by a lowercase letter for names with file scope because of possible conflicts with current or future reserved identifiers.

See Also

Reference

Lexical Conventions

C++ Keywords

Keywords are predefined reserved identifiers that have special meanings. They cannot be used as identifiers in your program. The following keywords are reserved for Microsoft C++. Names with leading underscores are Microsoft extensions.

__abstract 2	abstract	__alignof Operator	array
__asm	__assume	__based	bool
__box 2	break	case	catch
__cdecl	char	class	const
const_cast	continue	__declspec	default
__delegate 2	delegate	delete	deprecated 1
dllexport 1	dllimport 1	do	double
dynamic_cast	else	enum	enum class
enum struct	event	__event	__except
explicit	extern	false	__fastcall
__finally	finally	float	for
for each, in	__forceinline	friend	friend_as
__gc 2	gcnew	generic	goto
__hook 3	__identifier	if	__if_exists
__if_not_exists	initonly	__inline	inline
int	__int8	__int16	__int32
__int64	__interface	interface class	interface struct
interior_ptr	__leave	literal	long
__m64	__m128	__m128d	__m128i
__multiple_inheritance	mutable	naked 1	namespace
new	new	__nogc 2	noinline 1
__noop	noreturn 1	nothrow 1	novtable 1
nullptr	operator	__pin 2	private
__property 2	property	property 1	protected

public	__raise	ref struct	ref class
register	reinterpret_cast	return	safecast
__sealed 2	sealed	selectany 1	short
signed	__single_inheritance	sizeof	static
static_cast	__stdcall	struct	__super
switch	template	this	thread 1
throw	true	try	__try/__except, __try/__finally
__try_cast 2	typedef	typeid	typeid
typename	__unaligned	__unhook 3	union
unsigned	using declaration, using directive	uuid 1	__uuidof
value struct	value class	__value 2	virtual
__virtual_inheritance	void	volatile	__w64
__wchar_t, wchar_t	while		

1 Extended attributes for the **__declspec** keyword.

2 Applicable to Managed Extensions for C++ only. This syntax is deprecated in Microsoft Visual C++ 2005.

3 Intrinsic function used in event handling.

See New Language Features in Visual C++ for information on keywords that are new in Visual C++ 2005.

Microsoft Specific

In Microsoft C++, identifiers with two leading underscores are reserved for compiler implementations. Therefore, the Microsoft convention is to precede Microsoft-specific keywords with double underscores. These words cannot be used as identifier names.

Microsoft extensions are enabled by default. To ensure that your programs are fully portable, you can disable Microsoft extensions by specifying the ANSI-compatible /Za command-line option (compile for ANSI compatibility) during compilation. When you do this, Microsoft-specific keywords are disabled.

When Microsoft extensions are enabled, you can use the Microsoft-specific keywords in your programs. For ANSI compliance, these keywords are prefaced by a double underscore. For backward compatibility, single-underscore versions of all the double-underscored keywords except **__except**, **__finally**, **__leave**, and **__try** are supported. In addition, **__cdecl** is available with no leading underscore.

The **__asm** keyword replaces C++ **asm** syntax. **asm** is reserved for compatibility with other C++ implementations, but not implemented. Use**__asm**.

The **__based** keyword has limited uses for 32-bit and 64-bit target compilations.

END Microsoft Specific

See Also

Reference

Lexical Conventions

C++ Operators

C++ Punctuators

Punctuators in C++ have syntactic and semantic meaning to the compiler but do not, of themselves, specify an operation that yields a value. Some punctuators, either alone or in combination, can also be C++ operators or be significant to the preprocessor.

Any of the following characters are considered punctuators:

```
! % ^ & * ( ) - + = { } | ~
[ ] \ ; ' : " < > ? , . / #
```

The punctuators **[ ]**, **( )**, and **{ }** must appear in pairs after translation phase 4.

See Also

Reference

Lexical Conventions

C++ Operators

The following list contains C++ operators organized by category.

Additive

Addition: +	Subtraction: –	

Assignment

Addition Assignment: +=	Assignment: =	Bitwise AND Assignment: &=
Bitwise exclusive OR Assignment: ^=	Bitwise inclusive OR Assignment: |=	Division Assignment: /=
Left shift assignment: <<=	Modulus Assignment: %=	Multiplication Assignment: *=
Right shift assignment: >>=	Subtraction Assignment: –=	

Bitwise

Bitwise AND: &	Bitwise exclusive OR: ^	Bitwise inclusive OR: |

Logical

Logical AND: &&	Logical OR: ||	

Miscellaneous

Comma: ,	Conditional: ? :	Pointer-to-member: .* or –>*
Reference: &	Scope resolution: ::	

Multiplicative

Division: /	Modulus: %	Multiplication: *

Postfix

Cast: ()	Function call: ( )	Member access: . and –>
Postfix decrement: —	Postfix increment: ++	Subscript: [ ]

Relational and Equality

Equality: ==	Greater than or equal to: >=	Greater than: >
Less than or equal to: <=	Less than: <	Not equal: !=

Shift

Left shift: <<	Right shift: >>	

Unary

Address-of: &	delete	Indirection: *
Logical Negation: !	new	One's Complement: ~
Prefix decrement: —	Prefix increment: ++	sizeof
Unary Plus Operator: +	Unary Negation Operator: -	

See Also

Reference

Operator Precedence and Associativity
Operator Overloading
C++ Keywords
Lexical Conventions

Operator Precedence and Associativity

The C++ language includes all C operators and adds several new operators. Operators specify an evaluation to be performed on one of the following:

- One operand (unary operator)
- Two operands (binary operator)
- Three operands (ternary operator)

Operators follow a strict precedence, which defines the evaluation order of expressions containing these operators. Operators associate with either the expression on their left or the expression on their right; this is called "associativity." The following table shows the precedence and associativity of C++ operators (from highest to lowest precedence). Operators in the same segment of the table have equal precedence and are evaluated in the given order in an expression unless explicitly forced by parentheses.

C++ Operator Precedence and Associativity

Operator	Name or Meaning	Associativity
::	Scope resolution	None
.	Member selection (object)	Left to right
–>	Member selection (pointer)	Left to right
[ ]	Array subscript	Left to right
( )	Function call member initialization	Left to right
++	Postfix increment	Left to right
—	Postfix decrement	Left to right
typeid( )	type name	Left to right
const_cast	Type cast (conversion)	Left to right
dynamic_cast	Type cast (conversion)	Left to right
reinterpret_cast	Type cast (conversion)	Left to right
static_cast	Type cast (conversion)	Left to right
sizeof	Size of object or type	Right to left
++	Prefix increment	Right to left
—	Prefix decrement	Right to left
~	One's complement	Right to left
!	Logical not	Right to left
–	Unary minus	Right to left
+	Unary plus	Right to left
&	Address-of	Right to left
*****	Indirection	Right to left
new	Create object	Right to left
delete	Destroy object	Right to left
()	Cast	Right to left

.*	Pointer-to-member (objects)	Left to right
–>*	Pointer-to-member (pointers)	Left to right
*****	Multiplication	Left to right
/	Division	Left to right
%	Modulus	Left to right
+	Addition	Left to right
–	Subtraction	Left to right
<<	Left shift	Left to right
>>	Right shift	Left to right
<	Less than	Left to right
>	Greater than	Left to right
<=	Less than or equal to	Left to right
>=	Greater than or equal to	Left to right
==	Equality	Left to right
!=	Inequality	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise inclusive OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
e1?e2:e3	Conditional	Right to left
=	Assignment	Right to left
***=**	Multiplication assignment	Right to left
/=	Division assignment	Right to left
%=	Modulus assignment	Right to left
+=	Addition assignment	Right to left
–=	Subtraction assignment	Right to left
<<=	Left-shift assignment	Right to left
>>=	Right-shift assignment	Right to left
&=	Bitwise AND assignment	Right to left
|=	Bitwise inclusive OR assignment	Right to left
^=	Bitwise exclusive OR assignment	Right to left
throw expr	throw expression	Right to left

,	Comma	Left to right

See Also

Reference

C++ Operators

Operator Overloading

Literals (C++)

Invariant program elements are called "literals" or "constants." The terms "literal" and "constant" are used interchangeably here. Literals fall into four major categories: integer, character, floating-point, and string literals.

A literal may be any of the following:

```
integer-constant
character-constant
floating-constant
string-literal
```

Examples

```
157 // integer constant
0xFE // integer constant
'c' // character constant
0.2 // floating constant
0.2E-01 // floating constant
"dog" // string literal
```

For more examples, see the following sections, referenced above.

See Also

Reference

Lexical Conventions

C++ Integer Constants

Integer constants are constant data elements that have no fractional parts or exponents. They always begin with a digit. You can specify integer constants in decimal, octal, or hexadecimal form. They can specify signed or unsigned types and long or short types.

Grammar

integer-constant:

decimal-constant integer-suffix $_{opt}$

octal-constant integer-suffix $_{opt}$

hexadecimal-constant integer-suffix $_{opt}$

' *c-char-sequence* **'**

decimal-constant:

nonzero-digit

decimal-constant digit

octal-constant:

0

octal-constant octal-digit

hexadecimal-constant:

0x *hexadecimal-digit*

0X *hexadecimal-digit*

hexadecimal-constant hexadecimal-digit

nonzero-digit: one of

1 2 3 4 5 6 7 8 9

octal-digit: one of

0 1 2 3 4 5 6 7

hexadecimal-digit: one of

0 1 2 3 4 5 6 7 8 9

a b c d e f

A B C D E F

integer-suffix:

unsigned-suffix long-suffix $_{opt}$

long-suffix unsigned-suffix $_{opt}$

unsigned-suffix: one of

u U

long-suffix: one of

l L

64-bit integer-suffix:

i64 LL ll

To specify integer constants using octal or hexadecimal notation, use a prefix that denotes the base. To specify an integer constant of a given integral type, use a suffix that denotes the type.

To specify a decimal constant, begin the specification with a nonzero digit. For example:

```
int i = 157;   // Decimal constant
int j = 0198;  // Not a decimal number; erroneous octal constant
int k = 0365;  // Leading zero specifies octal constant, not decimal
```

To specify an octal constant, begin the specification with 0, followed by a sequence of digits in the range 0 through 7. The digits 8 and 9 are errors in specifying an octal constant. For example:

```
int i = 0377;   // Octal constant
int j = 0397;   // Error: 9 is not an octal digit
```

To specify a hexadecimal constant, begin the specification with `0x` or `0X` (the case of the "x" does not matter), followed by a sequence of digits in the range `0` through `9` and `a` (or `A`) through `f` (or `F`). Hexadecimal digits `a` (or `A`) through `f` (or `F`) represent values in the range 10 through 15. For example:

```
int i = 0x3fff;   // Hexadecimal constant
int j = 0X3FFF;   // Equal to i
```

To specify an unsigned type, use either the **u** or **U** suffix. To specify a long type, use either the **l** or **L** suffix. For example:

```
unsigned uVal = 328u;             // Unsigned value
long lVal = 0x7FFFFFL;            // Long value specified
                                  //  as hex constant
unsigned long ulVal = 0776745ul;  // Unsigned long value
```

To specify a 64-bit integral type, use the LL, ll or i64 suffix. For example,

```
// 64bitsuffix.cpp
#include <stdio.h>
enum MyEnum {
   IntType,
   Int64Type
};

MyEnum f1(int) {
   printf("in f1(int)\n");
   return IntType;
}

MyEnum f1(__int64) {
   printf_s("in f1(__int64)\n");
   return Int64Type;
}

int main() {
   MyEnum t1 = f1(0x1234), t2 = f1(0x1234i64);
}
```

Output

```
in f1(int)
in f1(__int64)
```

See Also

Reference

Literals (C++)

C++ Character Constants

Character constants are one or more members of the "source character set," the character set in which a program is written, surrounded by single quotation marks (`'`). They are used to represent characters in the "execution character set," the character set on the machine where the program executes.

Microsoft Specific

For Microsoft C++, the source and execution character sets are both ASCII.

END Microsoft Specific

There are three kinds of character constants:

- Normal character constants
- Multicharacter constants
- Wide-character constants

Note
Use wide-character constants in place of multicharacter constants to ensure portability.

Character constants are specified as one or more characters enclosed in single quotation marks. For example:

```
char ch = 'x';          // Specify normal character constant.
int mbch = 'ab';        // Specify system-dependent
                        //  multicharacter constant.
wchar_t wcch = L'ab';   // Specify wide-character constant.
```

Note that `mbch` is of type **int**. If it were declared as type **char**, the second byte would not be retained. A multicharacter constant has four meaningful characters; specifying more than four generates an error message.

Characters within a character constant may be any graphic characters in the source character set, except for newline, backslash \, single quote ' and double quote ", or they may be specified using an escape sequence. There are three types of escape sequences: simple, octal and hexadecimal escape sequences.

Simple escape sequences may be any of the following:

```
        \' \" \? \\
\a \b \f \n \r \t \v
```

An octal escape sequence is a backslash followed by a sequence of up to 3 octal digits.

A hexadecimal escape sequence is a backslash, followed by the character x, followed by a sequence of hex digits.

Microsoft C++ supports normal, multicharacter, and wide-character constants. Use wide-character constants to specify members of the extended execution character set (for example, to support an international application). Normal character constants have type **char**, multicharacter constants have type **int**, and wide-character constants have type **wchar_t**. (The type **wchar_t** is defined in the standard include files STDDEF.H, STDLIB.H, and STRING.H. The wide-character functions, however, are prototyped only in STDLIB.H.)

The only difference in specification between normal and wide-character constants is that wide-character constants are preceded by the letter `L`. For example:

```
char schar = 'x';              // Normal character constant
wchar_t wchar = L'\x8119';    // Wide-character constant
```

The following table shows reserved or nongraphic characters that are system dependent or not allowed within character constants. These characters should be represented with escape sequences.

C++ Reserved or Nongraphic Characters

Character	ASCII Representation	ASCII Value	Escape Sequence
Newline	NL (LF)	10 or 0x0a	\n
Horizontal tab	HT	9	\t
Vertical tab	VT	11 or 0x0b	\v
Backspace	BS	8	\b
Carriage return	CR	13 or 0x0d	\r
Formfeed	FF	12 or 0x0c	\f
Alert	BEL	7	\a
Backslash	\	92 or 0x5c	\\
Question mark	?	63 or 0x3f	\?
Single quotation mark	'	39 or 0x27	\'
Double quotation mark	"	34 or 0x22	\"
Octal number	*ooo*	—	*ooo*
Hexadecimal number	*hhh*	—	\x*hhh*
Null character	NUL	0	\0

If the character following the backslash does not specify a legal escape sequence, the result is implementation defined. In Microsoft C++, the character following the backslash is taken literally, as though the escape were not present, and a level 1 warning ("unrecognized character escape sequence") is issued.

Octal escape sequences, specified in the form *ooo*, consist of a backslash and one, two, or three octal characters. Hexadecimal escape sequences, specified in the form \x*hhh*, consist of the characters `\x` followed by a sequence of hexadecimal digits. Unlike octal escape constants, there is no limit on the number of hexadecimal digits in an escape sequence.

Octal escape sequences are terminated by the first character that is not an octal digit, or when three characters are seen. For example:

```
wchar_t och = L'\076a';  // Sequence terminates at a
char    ch = '\233';     // Sequence terminates after 3 characters
```

Similarly, hexadecimal escape sequences terminate at the first character that is not a hexadecimal digit. Because hexadecimal digits include the letters `a` through `f` (and `A` through `F`), make sure the escape sequence terminates at the intended digit.

Because the single quotation mark (`'`) encloses character constants, use the escape sequence `\'` to represent enclosed single quotation marks. The double quotation mark (`"`) can be represented without an escape sequence. The backslash character (\) is a line-continuation character when placed at the end of a line. If you want a backslash character to appear within a character constant, you must type two backslashes in a row (`\\`). (See Phases of Translation in the *Preprocessor Reference* for more information about line continuation.)

See Also

Reference

Literals (C++)

C++ Floating-Point Constants

Floating-point constants specify values that must have a fractional part. These values contain decimal points (**.**) and can contain exponents.

Floating-point constants have a "mantissa," which specifies the value of the number, an "exponent," which specifies the magnitude of the number, and an optional suffix that specifies the constant's type. The mantissa is specified as a sequence of digits followed by a period, followed by an optional sequence of digits representing the fractional part of the number. For example:

```
18.46
38.
```

The exponent, if present, specifies the magnitude of the number as a power of 10, as shown in the following example:

```
18.46e0      // 18.46
18.46e1      // 184.6
```

The exponent may be specified using **e** or **E**, which have the same meaning, followed by an optional sign (+ or -) and a sequence of digits. If an exponent is present, the trailing decimal point is unnecessary in whole numbers such as `18E0`.

Floating-point constants default to type **double**. By using the suffixes **f** or **l** (or **F** or **L** — the suffix is not case sensitive), the constant can be specified as **float** or **long double**, respectively.

Although **long double** and **double** have the same representation, they are not the same type. For example, you can have overloaded functions like

```
void func( double );
```

and

```
void func( long double );
```

See Also

Reference

Literals (C++)

C++ String Literals

A string literal consists of zero or more characters from the source character set surrounded by double quotation marks (**"**). A string literal represents a sequence of characters that, taken together, form a null-terminated string.

String literals may contain any graphic character from the source character set except the double quotation mark ("), backslash (****), or newline character. They may contain the same escape sequences described in C++ Character Constants.

C++ strings have these types:

- Array of **char**[*n*], where *n* is the length of the string (in characters) plus 1 for the terminating '\0' that marks the end of the string
- Array of **wchar_t**, for wide-character strings

The result of modifying a string constant is undefined. For example:

```
char *szStr = "1234";
szStr[2] = 'A';      // Results undefined
```

Microsoft Specific

In some cases, identical string literals can be "pooled" to save space in the executable file. In string-literal pooling, the compiler causes all references to a particular string literal to point to the same location in memory, instead of having each reference point to a separate instance of the string literal. /GF enables string pooling.

END Microsoft Specific

When specifying string literals, adjacent strings are concatenated. Therefore, this declaration:

```
char szStr[] = "12" "34";
```

is identical to this declaration:

```
char szStr[] = "1234";
```

This concatenation of adjacent strings makes it easy to specify long strings across multiple lines:

```
cout << "Four score and seven years "
        "ago, our forefathers brought forth "
        "upon this continent a new nation.";
```

In the preceding example, the entire string `Four score and seven years ago, our forefathers brought forth upon this continent a new nation.` is spliced together. This string can also be specified using line splicing as follows:

```
cout << "Four score and seven years \
ago, our forefathers brought forth \
upon this continent a new nation.";
```

After all adjacent strings in the constant have been concatenated, the **NULL** character, `'\0'`, is appended to provide an end-of-string marker for C string-handling functions.

When the first string contains an escape character, string concatenation can yield surprising results. Consider the following two declarations:

```
char szStr1[] = "\01" "23";
char szStr2[] = "\0123";
```

Although it is natural to assume that `szStr1` and `szStr2` contain the same values, the values they actually contain are shown in the following figure.

Escapes and String Concatenation

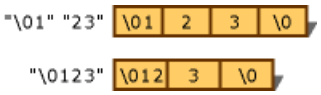

Microsoft Specific

The maximum length of a string literal is approximately 2,048 bytes. This limit applies to strings of type **char[]** and **wchar_t[]**. If a string literal consists of parts enclosed in double quotation marks, the preprocessor concatenates the parts into a single string, and for each line concatenated, it adds an extra byte to the total number of bytes.

For example, suppose a string consists of 40 lines with 50 characters per line (2,000 characters), and one line with 7 characters, and each line is surrounded by double quotation marks. This adds up to 2,007 bytes plus one byte for the terminating null character, for a total of 2,008 bytes. On concatenation, an extra character is added to the total number of bytes for each of the first 40 lines. This makes a total of 2,048 bytes. (The extra characters are not actually written to the string.) Note, however, that if line continuations (\) are used instead of double quotation marks, the preprocessor does not add an extra character for each line.

END Microsoft Specific

Determine the size of string objects by counting the number of characters and adding 1 for the terminating `'\0'` or 2 for type **wchar_t**.

Because the double quotation mark (`"`) encloses strings, use the escape sequence (`\"`) to represent enclosed double quotation marks. The single quotation mark (`'`) can be represented without an escape sequence. The backslash character (`\`) is a line-continuation character when placed at the end of a line. If you want a backslash character to appear within a string, you must type two backslashes (`\\`). (See Phases of Translation in the *Preprocessor Reference* for more information about line continuation.)

To specify a string of type wide-character (**wchar_t[]**), precede the opening double quotation mark with the character **L**. For example:

```
wchar_t wszStr[] = L"1a1g";
```

All normal escape codes listed in Character Constants are valid in string constants. For example:

```
cout << "First line\nSecond line";
cout << "Error! Take corrective action\a";
```

Because the escape code terminates at the first character that is not a hexadecimal digit, specification of string constants with embedded hexadecimal escape codes can cause unexpected results. The following example is intended to create a string literal containing ASCII 5, followed by the characters `five`:

```
\x05five"
```

The actual result is a hexadecimal 5F, which is the ASCII code for an underscore, followed by the characters `ive`. The following example produces the desired results:

```
"\005five"     // Use octal constant.
"\x05" "five"  // Use string splicing.
```

See Also

Reference

Literals (C++)

Not a Number (NAN) Items

The Visual C++ compiler supports comparisons of not a number (NAN) items in an IEEE-compliant manner. If *x* is NAN and *y* is not NAN:

- (*x* != *x*) == **true**
- (*x* == *x*) == **false**
- (*y* > *x*) == **false**
- (*y* < *x*) == **false**

NAN ordering tests always return false: NAN [<, <=, >, >=] [*any_number*] will be false.

The following code shows how NANs in Visual C++ cannot be compared successfully to a floating-point number:

```
#include <math.h>
#include <stdio.h>
#include <float.h>

int main( void ) {
unsigned long nan[2]={0xffffffff, 0x7fffffff};
double g = *( double* )nan;

if ( g <= 3.0 )
   printf( "g( %g ) <= 3.0\n", g );
else if ( g > 3.0)
   printf( "g( %g ) > 3.0\n", g );
else
   printf( "g( %g ) is NaN\n", g );
}
```

See Also

Reference

Lexical Conventions

Basic Concepts

This section explains concepts that are critical to understanding C++. C programmers will be familiar with many of these concepts, but there are some subtle differences that can cause unexpected program results. The following topics are included:

- Terms
- Declarations and definitions
- Scope of a C++ object or function
- Program definition and linkage rules
- Startup and termination
- Storage classes
- Types
- L-values and r-values
- Numerical limits

See Also

Other Resources

C++ Language Reference

Terms

C++ terms used in this book are defined in the following table:

C++ Terminology

Term	Meaning
Declaration	A declaration introduces names and their types into a program without necessarily defining an associated object or function. However, many declarations serve as definitions.
Definition	A definition provides information that allows the compiler to allocate memory for objects or generate code for functions.
Lifetime	The lifetime of an object is the period during which an object exists, including its creation and destruction.
Linkage	Names can have external linkage, internal linkage, or no linkage. Within a program (a set of translation units), only names with external linkage denote the same object or function. Within a translation unit, names with either internal or external linkage denote the same object or function (except when functions are overloaded). (For more information on translation units, see Phases of Translation), in the *Preprocessor Reference*.) Names with no linkage denote unique objects or functions.
Name	A name denotes an object, function, set of overloaded functions, enumerator, type, class member, template, value, or label. C++ programs use names to refer to their associated language element. Names can be type names or identifiers.
Object	An object is an instance (a data item) of a user-defined type (a class type). The difference between an object and a variable is that variables retain state information, whereas objects can also have behavior. This manual draws a distinction between objects and variables: "object" means instance of a user-defined type, whereas "variable" means instance of a fundamental type. In cases where either object or variable is applicable, the term "object" is used as the inclusive term, meaning "object or variable."
Scope	Names can be used only within specific regions of program text. These regions are called the scope of the name.
Storage class	The storage class of a named object determines its lifetime, initialization, and, in certain cases, its linkage.
Type	Names have associated types that determine the meaning of the value or values stored in an object or returned by a function.
Variable	A variable is a data item of a fundamental type (for example, **int**, **float**, or **double**). Variables store state information but define no behavior for how that information is handled. See the preceding list item "Object" for information about how the terms "variable" and "object" are used in this documentation.

See Also

Reference

Basic Concepts

C++ Declarations and Definitions

Declarations tell the compiler that a program element or name exists. Definitions specify what code or data the name describes. A name must be declared before it can be used.

See Also

Reference

Basic Concepts

C++ Declarations

A declaration introduces one or more names into a program. Declarations can occur more than once in a program. Therefore, classes, structures, enumerated types, and other user-defined types can be declared for each compilation unit. The constraint on this multiple declaration is that all declarations must be identical. Declarations also serve as definitions, except when the declaration:

- Is a function prototype (a function declaration with no function body).
- Contains the **extern** specifier but no initializer (objects and variables) or function body (functions). This signifies that the definition is not necessarily in the current translation unit and gives the name external linkage.
- Is of a static data member inside a class declaration.

  Because static class data members are discrete variables shared by all objects of the class, they must be defined and initialized outside the class declaration. (For more information about classes and class members, see Classes.)
- Is a class name declaration with no following definition, such as `class T;`.
- Is a **typedef** statement.

Examples of declarations that are also definitions are:

```
// Declare and define int variables i and j.
int i;
int j = 10;

// Declare enumeration suits.
enum suits { Spades = 1, Clubs, Hearts, Diamonds };

// Declare class CheckBox.
class CheckBox : public Control
{
public:
            Boolean IsChecked();
    virtual int     ChangeState() = 0;
};
```

Some declarations that are not definitions are:

```
extern int i;
char *strchr( const char *Str, const char Target );
```

See Also

Reference

C++ Declarations and Definitions

C++ Definitions

A definition is a unique specification of an object or variable, function, class, or enumerator. Because definitions must be unique, a program can contain only one definition for a given program element.

There can be a many-to-one correspondence between declarations and definitions. There are two cases in which a program element can be declared and not defined:

- A function is declared but never referenced with a function call or with an expression that takes the function's address.
- A class is used only in a way that does not require its definition be known. However, the class must be declared. The following code illustrates such a case:

```
// definitions.cpp
class WindowCounter;   // Forward reference; no definition

class Window
{
   // Definition of WindowCounter not required
   static WindowCounter windowCounter;
};

int main()
{
}
```

See Also

Reference

C++ Declarations and Definitions

Scope

C++ names can be used only in certain regions of a program. This area is called the "scope" of the name. Scope determines the "lifetime" of a name that does not denote an object of static extent. Scope also determines the visibility of a name, when class constructors and destructors are called, and when variables local to the scope are initialized. (For more information, see Constructors and Destructors.) There are five kinds of scope:

- **Local scope** A name declared within a block is accessible only within that block and blocks enclosed by it, and only after the point of declaration. The names of formal arguments to a function in the scope of the outermost block of the function have local scope, as if they had been declared inside the block enclosing the function body. Consider the following code fragment:

  ```
  {
      int i;
  }
  ```

  Because the declaration of `i` is in a block enclosed by curly braces, `i` has local scope and is never accessible because no code accesses it before the closing curly brace.

- **Function scope** Labels are the only names that have function scope. They can be used anywhere within a function, but are not accessible outside that function.

- **File scope** Any name declared outside all blocks or classes has file scope. It is accessible anywhere in the translation unit after its declaration. Names with file scope that do not declare static objects are often called global names.

  In C++, file scope is also known as namespace scope.

- **Class scope** Names of class members have class scope. Class member functions can be accessed only by using the member-selection operators (**.** or **->**) or pointer-to-member operators (**.*** or **->***) on an object or pointer to an object of that class; nonstatic class member data is considered local to the object of that class. Consider the following class declaration:

  ```
  class Point
  {
      int x;
      int y;
  };
  ```

  The class members `x` and `y` are considered to be in the scope of class `Point`.

- **Prototype scope** Names declared in a function prototype are visible only until the end of the prototype. The following prototype declares three names (`strDestination`, `numberOfElements`, and `strSource`); these names go out of scope at the end of the prototype:

  ```
  errno_t strcpy_s( char *strDestination, size_t numberOfElements, const char *strSource
  );
  ```

See Also

Reference

Basic Concepts

Point of Declaration

A name is considered to be declared immediately after its declarator but before its (optional) initializer. (For more information on declarators, see Declarators.) An enumerator is considered to be declared immediately after the identifier that names it but before its (optional) initializer.

Consider this example:

```
// point_of_declaration1.cpp
// compile with: /W1
double dVar = 7.0;
int main()
{
   double dVar = dVar;   // C4700
}
```

If the point of declaration were *after* the initialization, then the local `dVar` would be initialized to 7.0, the value of the global variable `dVar`. However, since that is not the case, `dVar` is initialized to an undefined value.

Enumerators follow the same rule. However, enumerators are exported to the enclosing scope of the enumeration. In the following example, the enumerators `Spades`, `Clubs`, `Hearts`, and `Diamonds` are declared. Because the enumerators are exported to the enclosing scope, they are considered to have global scope. The identifiers in the example are already defined in global scope.

Consider the following code:

```
const int Spades = 1, Clubs = 2, Hearts = 3, Diamonds = 4;
enum Suits
{
    Spades = Spades,     // error
    Clubs,               // error
    Hearts,              // error
    Diamonds             // error
};
```

Because the identifiers in the preceding code are already defined in global scope, an error message is generated.

Note

Using the same name to refer to more than one program element — for example, an enumerator and an object — is conside red poor programming practice and should be avoided. In the preceding example, this practice causes an error.

See Also

Reference

Scope

Hiding Names

You can hide a name by declaring it in an enclosed block. In the following figure, `i` is redeclared within the inner block, thereby hiding the variable associated with `i` in the outer block scope.

Block Scope and Name Hiding

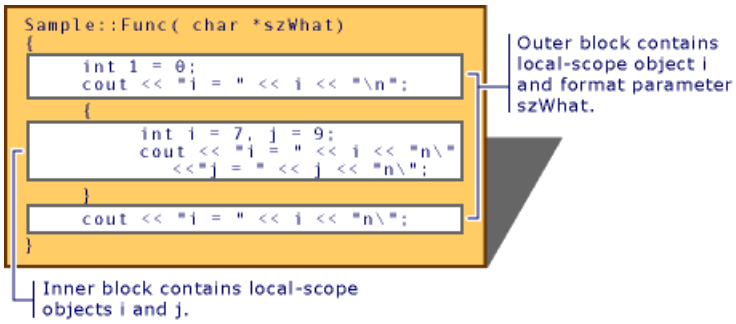

The output from the program shown in the figure is:

```
i = 0
i = 7
j = 9
i = 0
```

Note

The argument `szWhat` is considered to be in the scope of the function. Therefore, it is treated as if it had been declared in the outermost block of the function.

See Also

Reference

Scope

Hiding Names with File Scope

You can hide names with file scope by explicitly declaring the same name in block scope. However, file-scope names can be accessed using the scope-resolution operator (**::**).

Example

```
// file_scopes.cpp
// compile with: /EHsc
#include <iostream>

int i = 7;   // i has file scope, outside all blocks
using namespace std;

int main( int argc, char *argv[] ) {
   int i = 5;   // i has block scope, hides i at file scope
   cout << "Block-scoped i has the value: " << i << "\n";
   cout << "File-scoped i has the value: " << ::i << "\n";
}
```

Output

```
Block-scoped i has the value: 5
File-scoped i has the value: 7
```

See Also

Reference

Hiding Names

Hiding Class Names

You can hide class names by declaring a function, object or variable, or enumerator in the same scope. However, the class name can still be accessed when prefixed by the keyword **class**.

Example

```
// hiding_class_names.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

// Declare class Account at file scope.
class Account
{
public:
    Account( double InitialBalance )
        { balance = InitialBalance; }
    double GetBalance()
        { return balance; }
private:
    double balance;
};

double Account = 15.37;            // Hides class name Account

int main()
{
    class Account Checking( Account ); // Qualifies Account as
                                       //  class name

    cout << "Opening account with balance of: "
         << Checking.GetBalance() << "\n";
}
```

Output

```
Opening account with balance of: 15.37
```

Note

Any place the class name (`Account`) is called for, the keyword class must be used to differentiate it from the file-scoped variable Account. This rule does not apply when the class name occurs on the left side of the scope-resolution operator (::). Names on the left side of the scope-resolution operator are always considered class names.

The following example demonstrates how to declare a pointer to an object of type `Account` using the **class** keyword:

```
class Account *Checking = new class Account( Account );
```

The `Account` in the initializer (in parentheses) in the preceding statement has file scope; it is of type **double**.

Note

The reuse of identifier names as shown in this example is considered poor programming style.

For more information about pointers, see Derived Types. For information about declaration and initialization of class objects, see Classes, Structures, and Unions. For information about using the **new** and **delete** free-store operators, see Special Member Functions.

See Also

Reference

Hiding Names

Scope of Formal Arguments to Functions

Formal arguments (arguments specified in function definitions) to functions are considered to be in the scope of the outermost block of the function body.

See Also

Reference

Scope

Program and Linkage

A program consists of one or more translation units linked together. Execution (conceptually) begins in the translation unit that contains the function **main**. (For more information on translation units, see Phases of Translation, in the *Preprocessor Reference*.) For more information about the **main** function, see Program Startup: the main Function.)

See Also

Reference

Basic Concepts

Types of Linkage

The way the names of objects and functions are shared between translation units is called linkage. These names can have:

- Internal linkage, in which case they refer only to program elements inside their own translation units; they are not shared with other translation units.

  The same name in another translation unit may refer to a different object or a different class. Names with internal linkage are sometimes referred to as being local to their translation units.

  An example declaration of a name with internal linkage is:

  ```
  static int i;   // The static keyword ensures internal linkage.
  ```

- External linkage, in which case they can refer to program elements in any translation unit in the program — the program element is shared among the translation units.

  The same name in another translation unit is guaranteed to refer to the same object or class. Names with external linkage are sometimes referred to as being global.

  An example declaration of a name with external linkage is:

  ```
  extern int i;
  ```

- No linkage, in which case they refer to unique entities. The same name in another scope may not refer to the same object. An example is an enumeration. (Note, however, that you can pass a pointer to an object with no linkage. This makes the object accessible in other translation units.)

See Also

Reference

Program and Linkage

Linkage in Names with File Scope

The following linkage rules apply to names (other than **typedef** and enumerator names) with file scope:

- If a name is explicitly declared as **static**, it has internal linkage and identifies a program element inside its own translation unit.
- Enumerator names and **typedef** names have no linkage.
- All other names with file scope have external linkage.

Microsoft Specific

- If a function name with file scope is explicitly declared as **inline**, it has external linkage if it is instantiated or its address is referenced. Therefore, it is possible for a function with file scope to have either internal or external linkage.

END Microsoft Specific

A class has internal linkage if it:

- Uses no C++ functionality (for example, member-access control, member functions, constructors, destructors, and so on).
- Is not used in the declaration of another name that has external linkage. This constraint means that objects of class type that are passed to functions with external linkage cause the class to have external linkage.

See Also
Reference
Program and Linkage

Linkage in Names with Class Scope

The following linkage rules apply to names with class scope:

- Static class members have external linkage.
- Class member functions have external linkage.
- Enumerators and **typedef** names have no linkage.

Microsoft Specific

- Functions declared as **friend** functions must have external linkage. Declaring a static function as a **friend** generates an error.

END Microsoft Specific

See Also
Reference
Program and Linkage

Linkage in Names with Block Scope

The following linkage rules apply to names with block scope (local names):

- Names declared as **extern** have external linkage unless they were previously declared as **static**.
- All other names with block scope have no linkage.

See Also
Reference
Program and Linkage

Names with No Linkage

The only names that have no linkage are:

- Function parameters.
- Block-scoped names not declared as **extern** or **static**.
- Enumerators.
- Names declared in a **typedef** statement. An exception is when the **typedef** statement is used to provide a name for an unnamed class type. The name may then have external linkage if the class has external linkage. The following example shows a situation in which a **typedef** name has external linkage:

  ```
  // names_with_no_linkage.cpp
  typedef struct
  {
     short x;
     short y;
  } POINT;

  extern int MoveTo( POINT pt );

  int main()
  {
  }
  ```

  The **typedef** name, `POINT`, becomes the class name for the unnamed structure. It is then used to declare a function with external linkage.

Because **typedef** names have no linkage, their definitions can differ between translation units. Because the compilations take place discretely, there is no way for the compiler to detect these differences. As a result, errors of this kind are not detected until link time. Consider the following case:

```
// Translation unit 1
typedef int INT

INT myInt;
...

// Translation unit 2
typedef short INT

extern INT myInt;
...
```

The preceding code generates an "unresolved external" error at link time.

Example

C++ functions can be defined only in file or class scope. The following example illustrates how to define functions and shows an erroneous function definition:

```
// function_definitions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
void ShowChar( char ch );    // Declare function ShowChar.

void ShowChar( char ch )     // Define function ShowChar.
{                            // Function has file scope.
   cout << ch;
```

```
}

struct Char                  // Define class Char.
{
    char Show();             // Declare Show function.
    char Get();              // Declare Get function.
    char ch;
};

char Char::Show()            // Define Show function
{                            //  with class scope.
    cout << ch;
    return ch;
}

void GoodFuncDef( char ch )  // Define GoodFuncDef
{                            //  with file scope.
    int BadFuncDef( int i )  // C2601, Erroneous attempt to
    {                        //  nest functions.
        return i * 7;
    }
    for( int i = 0; i < BadFuncDef( 2 ); ++i )
        cout << ch;
    cout << "\n";
}
```

See Also

Reference

Program and Linkage

Using extern to Specify Linkage

```
extern string-literal { declaration-list }
extern string-literal declaration
```

Remarks

The **extern** keyword declares a variable or function and specifies that it has external linkage (its name is visible from files other than the one in which it's defined). When modifying a variable, **extern** specifies that the variable has static duration (it is allocated when the program begins and deallocated when the program ends). The variable or function may be defined in another source file, or later in the same file. Declarations of variables and functions at file scope are external by default.

Example

```
// specifying_linkage1.cpp
int i = 1;
void other();

int main() {
   // Reference to i, defined above:
   extern int i;
}

void other() {
   // Address of global i assigned to pointer variable:
   static int *external_i = &i;

   // i will be redefined; global i no longer visible:
   // int i = 16;
}
```

In C++, when used with a string, **extern** specifies that the linkage conventions of another language are being used for the declarator(s). C functions and data can be accessed only if they are previously declared as having C linkage. However, they must be defined in a separately compiled translation unit.

Microsoft C++ supports the strings **"C"** and **"C++"** in the *string-literal* field. All of the standard include files use the **extern** "C" syntax to allow the run-time library functions to be used in C++ programs.

The following example shows alternative ways to declare names that have C linkage:

```
// specifying_linkage2.cpp
// compile with: /c
// Declare printf with C linkage.
extern "C" int printf( const char *fmt, ... );

//  Cause everything in the specified header files
//   to have C linkage.
extern "C" {
   // add your #include statements here
   #include <stdio.h>
}

//  Declare the two functions ShowChar and GetChar
//   with C linkage.
extern "C" {
   char ShowChar( char ch );
   char GetChar( void );
}

//  Define the two functions ShowChar and GetChar
//   with C linkage.
extern "C" char ShowChar( char ch ) {
   putchar( ch );
```

```
   return ch;
}

extern "C" char GetChar( void ) {
   char ch;

   ch = getchar();
   return ch;
}

// Declare a global variable, errno, with C linkage.
extern "C" int errno;
```

See Also

Reference

C++ Keywords

Linkage Specifications

The extern Storage-Class Specifier

Concepts

Behavior of Identifiers

Linkage

Startup and Termination

Program startup and termination are facilitated by using two functions: main and exit. Other startup and termination code may be executed.

See Also

Reference

Basic Concepts

main: Program Startup

A special function called **main** is the starting point of execution for all C and C++ programs. If you are writing code that adheres to the Unicode programming model, you can use the wide-character version of **main**, **wmain**.

The **main** function is not predefined by the compiler; rather, it must be supplied in the program text.

The declaration syntax for **main** is:

```
int main( );
```

or, optionally:

```
int main( int argc[ , char *argv[ ] [, char *envp[ ] ] ] );
```

Microsoft Specific

The declaration syntax for **wmain** is as follows:

```
int wmain( );
```

or, optionally:

```
int wmain( int argc[ , wchar_t *argv[ ] [, wchar_t *envp[ ] ] ] );
```

You can also use **_tmain**, which is defined in TCHAR.h. **_tmain** will resolve to **main** unless _UNICODE is defined, in which case **_tmain** will resolve to **wmain**.

END Microsoft Specific

The types for *argc* and *argv* are defined by the language. The names *argc*, *argv*, and *envp* are traditional, but are not required by the compiler. See Argument Definitions for more information and for an example.

Alternately, the **main** and **wmain** functions can be declared as returning **void** (no return value). If you declare **main** or **wmain** as returning **void**, you cannot return an exit code to the parent process or operating system using a return statement; to return an exit code when **main** or **wmain** is declared as **void**, you must use the exit function.

See Also

Reference

C++ Keywords

Using wmain Instead of main

main Function Restrictions

Using wmain Instead of main

Microsoft Specific

In the Unicode programming model, you can define a wide-character version of the **main** function. Use **wmain** instead of **main** if you want to write portable code that adheres to the Unicode specification.

You declare formal parameters to **wmain** using a similar format to **main**. You can then pass wide-character arguments and, optionally, a wide-character environment pointer to the program. The *argv* and *envp* parameters to **wmain** are of type **wchar_t***.

If your program uses a **main** function, the multibyte-character environment is created by the operating system at program startup. A wide-character copy of the environment is created only when needed (for example, by a call to the _wgetenv or _wputenv functions). On the first call to **_wputenv**, or on the first call to **_wgetenv** if an MBCS environment already exists, a corresponding wide-character string environment is created and is then pointed to by the **_wenviron** global variable, which is a wide-character version of the **_environ** global variable. At this point, two copies of the environment (MBCS and Unicode) exist simultaneously and are maintained by the operating system throughout the life of the program.

Similarly, if your program uses a **wmain** function, an MBCS (ASCII) environment is created on the first call to **_putenv** or **getenv**, and is pointed to by the **_environ** global variable.

For more information on the MBCS environment, see Single-byte and Multibyte Character Sets in the *Run-Time Library Reference*.

END Microsoft Specific

See Also

Reference

main: Program Startup

Argument Definitions

The arguments in the prototype

```
      int main( int argc[ , char *argv[ ] [, char *envp[ ] ] ] );
int wmain( int argc[ , wchar_t *argv[ ] [, wchar_t *envp[ ] ] ] );
```

allow convenient command-line parsing of arguments and, optionally, access to environment variables. The argument definitions are as follows:

argc

An integer that contains the count of arguments that follow in *argv*. The *argc* parameter is always greater than or equal to 1.

argv

An array of null-terminated strings representing command-line arguments entered by the user of the program. By convention, *argv***[0]** is the command with which the program is invoked, *argv***[1]** is the first command-line argument, and so on, until *argv***[***argc***]**, which is always **NULL**. See Customizing Command Line Processing for information on suppressing command-line processing.

The first command-line argument is always *argv***[1]** and the last one is *argv***[***argc* – 1**]**.

Note
By convention, *argv***[0]** is the command with which the program is invoked. However, it is possible to spawn a process using CreateProcess and if you use both the first and second arguments (*lpApplicationName* and *lpCommandLine*), *argv***[0]** may not be the executable name; use GetModuleFileName to retrieve the executable name, and its fully-qualified path.

Microsoft Specific

envp

The *envp* array, which is a common extension in many UNIX® systems, is used in Microsoft C++. It is an array of strings representing the variables set in the user's environment. This array is terminated by a **NULL** entry. It can be declared as an array of pointers to **char (char** *envp[ ]**)** or as a pointer to pointers to **char (char** **envp**)**. If your program uses **wmain** instead of **main**, use the **wchar_t** data type instead of **char**. The environment block passed to **main** and **wmain** is a "frozen" copy of the current environment. If you subsequently change the environment via a call to **putenv** or **_wputenv**, the current environment (as returned by **getenv**/**_wgetenv** and the **_environ**/ **_wenviron** variable) will change, but the block pointed to by envp will not change. See Customizing Command Line Processing for information on suppressing environment processing. This argument is ANSI compatible in C, but not in C++.

END Microsoft Specific

Example

The following example shows how to use the *argc*, *argv*, and *envp* arguments to **main**:

```
// argument_definitions.cpp
// compile with: /EHsc
#include <iostream>
#include <string.h>

using namespace std;
int main( int argc, char *argv[], char *envp[] ) {
    int iNumberLines = 0;    // Default is no line numbers.

    // If /n is passed to the .exe, display numbered listing
    // of environment variables.

    if ( (argc == 2) && _stricmp( argv[1], "/n" ) == 0 )
         iNumberLines = 1;

    // Walk through list of strings until a NULL is encountered.
    for( int i = 0; envp[i] != NULL; ++i ) {
```

```
        if( iNumberLines )
            cout << i << ": " << envp[i] << "\n";
    }
}
```

See Also

Reference

main: Program Startup

Wildcard Expansion

Microsoft Specific

You can use wildcards — the question mark (?) and asterisk (*) — to specify filename and path arguments on the command-line.

Command-line arguments are handled by a routine called **_setargv** (or **_wsetargv** in the wide-character environment), which by default does not expand wildcards into separate strings in the *argv* string array. For more information on enabling wildcard expansion, refer to Expanding Wildcard Arguments.

END Microsoft Specific

See Also

Reference

main: Program Startup

Parsing C++ Command-Line Arguments

Microsoft Specific

Microsoft C/C++ startup code uses the following rules when interpreting arguments given on the operating system command line:

- Arguments are delimited by white space, which is either a space or a tab.
- The caret character (^) is not recognized as an escape character or delimiter. The character is handled completely by the command-line parser in the operating system before being passed to the `argv` array in the program.
- A string surrounded by double quotation marks ("*string*") is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument.
- A double quotation mark preceded by a backslash (\") is interpreted as a literal double quotation mark character (").
- Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
- If an even number of backslashes is followed by a double quotation mark, one backslash is placed in the `argv` array for every pair of backslashes, and the double quotation mark is interpreted as a string delimiter.
- If an odd number of backslashes is followed by a double quotation mark, one backslash is placed in the `argv` array for every pair of backslashes, and the double quotation mark is "escaped" by the remaining backslash, causing a literal double quotation mark (") to be placed in `argv`.

Example

The following program demonstrates how command-line arguments are passed:

```
// command_line_arguments.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
int main( int argc,      // Number of strings in array argv
          char *argv[],   // Array of command-line argument strings
          char *envp[] )  // Array of environment variable strings
{
    int count;

    // Display each command-line argument.
    cout << "\nCommand-line arguments:\n";
    for( count = 0; count < argc; count++ )
         cout << "  argv[" << count << "]   "
                << argv[count] << "\n";
}
```

The following table shows example input and expected output, demonstrating the rules in the preceding list.

Results of Parsing Command Lines

Command-Line Input	argv[1]	argv[2]	argv[3]
`"abc" d e`	`abc`	`d`	`E`
`a\\\b d"e f"g h`	`a\\\b`	`de fg`	`H`
`a\\\"b c d`	`a\"b`	`c`	`D`

a\\\\"b c" d e	a\\b c	d	E

END Microsoft Specific

See Also

Reference

main: Program Startup

Customizing C++ Command-Line Processing

Microsoft Specific

If your program does not take command-line arguments, you can save a small amount of space by suppressing use of the library routine that performs command-line processing. This routine is called **_setargv** and is described in Wildcard Expansion. To suppress its use, define a routine that does nothing in the file containing the **main** function, and name it **_setargv**. The call to **_setargv** is then satisfied by your definition of **_setargv**, and the library version is not loaded.

Similarly, if you never access the environment table through the *envp* argument, you can provide your own empty routine to be used in place of **_setenvp**, the environment-processing routine. Just as with the **_setargv** function, **_setenvp** must be declared as **extern "C"**.

Your program might make calls to the **spawn** or **exec** family of routines in the C run-time library. If this is the case, you should not suppress the environment-processing routine, since this routine is used to pass an environment from the parent process to the child process.

END Microsoft Specific

See Also

Reference

main: Program Startup

main Function Restrictions

Several restrictions apply to the **main** function that do not apply to any other C++ functions. The **main** function:

- Cannot be overloaded (see Overloading).
- Cannot be declared as **inline**.
- Cannot be declared as **static**.
- Cannot have its address taken.
- Cannot be called.

See Also

Reference

main: Program Startup

Program Termination

In C++, there are several ways to exit a program:

- Call the exit function.
- Call the abort function.
- Execute a return statement from **main**.

See Also

Reference

main: Program Startup

exit Function

The **exit** function, declared in the standard include file STDLIB.H, terminates a C++ program.

The value supplied as an argument to **exit** is returned to the operating system as the program's return code or exit code. By convention, a return code of zero means that the program completed successfully.

Note
You can use the constants **EXIT_FAILURE** and **EXIT_SUCCESS**, defined in STDLIB.H, to indicate success or failure of your program.

Issuing a **return** statement from the **main** function is equivalent to calling the **exit** function with the return value as its argument.

For more information, see exit in the *Run-Time Library Reference*.

See Also
Reference
Program Termination

abort Function

The **abort** function, also declared in the standard include file STDLIB.H, terminates a C++ program. The difference between **exit** and **abort** is that **exit** allows the C++ run-time termination processing to take place (global object destructors will be called), whereas **abort** terminates the program immediately. For more information, see abort in the *Run-Time Library Reference*.

See Also

Reference

Program Termination

return Statement (C++)

Issuing a **return** statement from **main** is functionally equivalent to calling the **exit** function. Consider the following example:

```
// return_statement.cpp
#include <stdlib.h>
int main()
{
    exit( 3 );
    return 3;
}
```

The **exit** and **return** statements in the preceding example are functionally identical. However, C++ requires that functions that have return types other than **void** return a value. The **return** statement allows you to return a value from **main**.

See Also

Reference

Program Termination

Additional Startup Considerations

In C++, object construction and destruction can involve executing user code. Therefore, it is important to understand which initializations happen before entry to **main** and which destructors are invoked after exit from **main**. (For detailed information about construction and destruction of objects, see Constructors and Destructors.)

The following initializations take place prior to entry to **main**:

- Default initialization of static data to zero. All static data without explicit initializers are set to zero prior to executing any other code, including run-time initialization. Static data members must still be explicitly defined.
- Initialization of global static objects in a translation unit. This may occur either before entry to **main** or before the first use of any function or object in the object's translation unit.

Microsoft Specific

In Microsoft C++, global static objects are initialized before entry to **main**.

END Microsoft Specific

Global static objects that are mutually interdependent but in different translation units may cause incorrect behavior.

See Also
Reference
Startup and Termination

Additional Termination Considerations

You can terminate a C++ program by using **exit**, **return**, or **abort**. You can add exit processing using the **atexit** function. These are discussed in the following sections.

See Also

Reference

Startup and Termination

Using exit or return

When you call **exit** or execute a **return** statement from **main**, static objects are destroyed in the reverse order of their initialization. The following example shows how such initialization and cleanup works.

Example

```
// using_exit_or_return1.cpp
#include <stdio.h>
class ShowData {
public:
   // Constructor opens a file.
   ShowData( const char *szDev ) {
   errno_t err;
      err = fopen_s(&OutputDev, szDev, "w" );
   }

   // Destructor closes the file.
   ~ShowData() { fclose( OutputDev ); }

   // Disp function shows a string on the output device.
   void Disp( char *szData ) {
      fputs( szData, OutputDev );
   }
private:
   FILE *OutputDev;
};

//  Define a static object of type ShowData. The output device
//   selected is "CON" -- the standard output device.
ShowData sd1 = "CON";

//  Define another static object of type ShowData. The output
//   is directed to a file called "HELLO.DAT"
ShowData sd2 = "hello.dat";

int main() {
   sd1.Disp( "hello to default device\n" );
   sd2.Disp( "hello to file hello.dat\n" );
}
```

In the preceding example, the static objects `sd1` and `sd2` are created and initialized before entry to `main`. After this program terminates using the `return` statement, first `sd2` is destroyed and then `sd1`. The destructor for the `ShowData` class closes the files associated with these static objects. (For more information about initialization, constructors, and destructors, see Special Member Functions.)

Another way to write this code is to declare the `ShowData` objects with block scope, allowing them to be destroyed when they go out of scope:

```
int main() {
   ShowData sd1, sd2( "hello.dat" );

   sd1.Disp( "hello to default device\n" );
   sd2.Disp( "hello to file hello.dat\n" );
}
```

See Also

Reference

Additional Termination Considerations

Using atexit

With the atexit function, you can specify an exit-processing function that executes prior to program termination. No global static objects initialized prior to the call to **atexit** are destroyed prior to execution of the exit-processing function.

See Also

Reference

Additional Termination Considerations

Using abort

Calling the abort function causes immediate termination. It bypasses the normal destruction process for initialized global static objects. It also bypasses any special processing that was specified using the **atexit** function.

See Also

Reference

Additional Termination Considerations

C++ Storage Classes

Storage classes govern the lifetime, linkage, and treatment of objects and variables in C++. A given object can have only one storage class. This section discusses the C++ storage classes for data:

- Automatic
- Static
- Register
- External

See Also

Reference

Storage-Class Specifiers

Automatic

Objects and variables with automatic storage are local to a given instance of a block. In recursive or multithreaded code, automatic objects and variables are guaranteed to have different storage in different instances of a block. Microsoft C++ stores automatic objects and variables on the program's stack.

Objects and variables defined within a block have **auto** storage unless otherwise specified using the **extern** or **static** keyword. Automatic objects and variables can be specified using the **auto** keyword, but explicit use of **auto** is unnecessary. Automatic objects and variables have no linkage.

Automatic objects and variables persist only until the end of the block in which they are declared.

See Also

Reference

C++ Storage Classes

auto Keyword

Static (C++)

When modifying a variable, the **static** keyword specifies that the variable has static duration (it is allocated when the program begins and deallocated when the program ends) and initializes it to 0 unless another value is specified. When modifying a variable or function at file scope, the **static** keyword specifies that the variable or function has internal linkage (its name is not visible from outside the file in which it is declared).

A variable declared **static** in a function retains its state between calls to that function.

When modifying a data member in a class declaration, the **static** keyword specifies that one copy of the member is shared by all instances of the class. When modifying a member function in a class declaration, the **static** keyword specifies that the function accesses only static members.

Static data members of classes must be initialized at file scope.

In recursive code, a static object or variable is guaranteed to have the same state in different instances of a block of code.

The members of a union cannot be declared as static. An anonymous union declared globally must be explicitly declared static.

Objects and variables defined outside all blocks have static lifetime and external linkage by default. A global object or variable that is explicitly declared as **static** has internal linkage.

For related information, see auto, extern, and register.

Example

The following example shows how a variable declared **static** in a function retains its state between calls to that function.

```
// static1.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
void showstat( int curr ) {
   static int nStatic;    // Value of nStatic is retained
                          // between each function call
   nStatic += curr;
   cout << "nStatic is " << nStatic << endl;
}

int main() {
   for ( int i = 0; i < 5; i++ )
      showstat( i );
}
```

Output

```
nStatic is 0
nStatic is 1
nStatic is 3
nStatic is 6
nStatic is 10
```

The following example shows the use of **static** in a class.

```
// static2.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class CMyClass {
public:
   static int m_i;
};

int CMyClass::m_i = 0;
```

```
int main() {
   cout << CMyClass::m_i << endl;
   cout << CMyClass::m_i << endl;

   CMyClass::m_i = 1;
   cout << CMyClass::m_i << endl;
   cout << CMyClass::m_i << endl;
}
```

Output

```
0
0
1
1
```

The following example shows a local variable declared **static** in a member function. The static variable is available to the whole program; all instances of the type share the same copy of the static variable.

Note
Assigning to a static local variable is not thread safe and is not recommended as a programming practice.

```
// static3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
struct C {
   void Test(int value) {
      static int var = 0;
      if (var == value)
         cout << "var == value" << endl;
      else
         cout << "var != value" << endl;

      var = value;
   }
};

int main() {
   C c1;
   C c2;
   c1.Test(100);
   c2.Test(100);
}
```

Output

```
var != value
var == value
```

See Also

Reference

C++ Keywords
C++ Storage Classes
Static Storage-Class Specifiers
Static Member Functions
Access Control and Static Members

Register

Only function arguments and local variables can be declared with the register storage class.

Like automatic variables, register variables persist only until the end of the block in which they are declared.

The compiler does not honor user requests for register variables; instead, it makes its own register choices when global optimizations are on. However, all other semantics associated with the register keyword are honored by the compiler.

See Also

Reference

C++ Storage Classes

External

Objects and variables declared as **extern** declare an object that is defined in another translation unit or in an enclosing scope as having external linkage.

Declaration of **const** variables with the **extern** storage class forces the variable to have external linkage. An initialization of an **extern const** variable is allowed in the defining translation unit. Initializations in translation units other than the defining translation unit produce undefined results.

The following code shows two **extern** declarations, `DefinedElsewhere` (which refers to a name defined in a different translation unit) and `DefinedHere` (which refers to a name defined in an enclosing scope):

```
// external.cpp
// defined in another translation unit
extern int DefinedElsewhere;
int main() {
   int DefinedHere;
   {
      // refers to DefinedHere in the enclosing scope
      extern int DefinedHere;
    }
}
```

See Also

Reference

C++ Storage Classes

Using extern to Specify Linkage

Initialization of Objects

A local automatic object or variable is initialized every time the flow of control reaches its definition. A local static object or variable is initialized the first time the flow of control reaches its definition.

Example

Consider the following example, which defines a class that logs initialization and destruction of objects and then defines three objects, `I1`, `I2`, and `I3`:

```
// initialization_of_objects.cpp
// compile with: /EHsc
#include <iostream>
#include <string.h>
using namespace std;

// Define a class that logs initializations and destructions.
class InitDemo {
public:
   InitDemo( const char *szWhat );
   ~InitDemo();

private:
   char *szObjName;
   size_t sizeofObjName;
};

// Constructor for class InitDemo
InitDemo::InitDemo( const char *szWhat ) :
   szObjName(NULL), sizeofObjName(0) {
   if( szWhat != 0 && strlen( szWhat ) > 0 ) {
      // Allocate storage for szObjName, then copy
      // initializer szWhat into szObjName, using
      // secured CRT functions.
      sizeofObjName = strlen( szWhat ) + 1;

      szObjName = new char[ sizeofObjName ];
      strcpy_s( szObjName, sizeofObjName, szWhat );

      cout << "Initializing: " << szObjName << "\n";
   }
   else
      szObjName = 0;
}

// Destructor for InitDemo
InitDemo::~InitDemo() {
   if( szObjName != 0 ) {
      cout << "Destroying: " << szObjName << "\n";
      delete szObjName;
   }
}

// Enter main function
int main() {
   InitDemo I1( "Auto I1" ); {
      cout << "In block.\n";
      InitDemo I2( "Auto I2" );
      static InitDemo I3( "Static I3" );
   }
   cout << "Exited block.\n";
}
```

Output

```
Initializing: Auto I1
In block.
Initializing: Auto I2
Initializing: Static I3
Destroying: Auto I2
Exited block.
Destroying: Auto I1
Destroying: Static I3
```

Comments

The preceding code demonstrates how and when the objects `I1`, `I2`, and `I3` are initialized and when they are destroyed.

There are several points to note about the program.

First, `I1` and `I2` are automatically destroyed when the flow of control exits the block in which they are defined.

Second, in C++, it is not necessary to declare objects or variables at the beginning of a block. Furthermore, these objects are initialized only when the flow of control reaches their definitions. (`I2` and `I3` are examples of such definitions.) The output shows exactly when they are initialized.

Finally, static local variables such as `I3` retain their values for the duration of the program, but are destroyed as the program terminates.

See Also

Reference

C++ Storage Classes

Types (C++)

C++ supports three kinds of object types:

- Fundamental types are built into the language (such as **int**, **float**, or **double**). Instances of these fundamental types are often called "variables."
- Derived types are new types derived from built-in types.
- Class types are new types created by combining existing types. These are discussed in Classes, Structures, and Unions.

See Also

Reference

Basic Concepts

Fundamental Types

Fundamental types in C++ are divided into three categories: integral, floating, and void. Integral types are capable of handling whole numbers. Floating types are capable of specifying values that may have fractional parts.

The void type describes an empty set of values. No variable of type **void** can be specified — it is used primarily to declare functions that return no values or to declare generic pointers to untyped or arbitrarily typed data. Any expression can be explicitly converted or cast to type **void**. However, such expressions are restricted to the following uses:

- An expression statement. (See Expressions, for more information.)
- The left operand of the comma operator. (See Comma Operator for more information.)
- The second or third operand of the conditional operator (**? :**). (See Expressions with the Conditional Operator for more information.)

The following table explains the restrictions on type sizes. These restrictions are independent of the Microsoft implementation.

Fundamental Types of the C++ Language

Category	Type	Contents
Integral	**char**	Type **char** is an integral type that usually contains members of the execution character set — in Microsoft C++, this is ASCII.
		The C++ compiler treats variables of type **char**, **signed char**, and **unsigned char** as having different types. Variables of type **char** are promoted to **int** as if they are type **signed char** by default, unless the /J compilation option is used. In this case they are treated as type **unsigned char** and are promoted to **int** without sign extension.
	bool	Type **bool** is an integral type that can have one of the two values **true** or **false**. Its size is unspecified.
	short	Type **short int** (or simply **short**) is an integral type that is larger than or equal to the size of type **char**, and shorter than or equal to the size of type **int**.
		Objects of type **short** can be declared as **signed short** or **unsigned short**. **Signed short** is a synonym for **short**.
	int	Type **int** is an integral type that is larger than or equal to the size of type **short int**, and shorter than or equal to the size of type **long**.
		Objects of type **int** can be declared as **signed int** or **unsigned int**. **Signed int** is a synonym for **int**.
	__int*n*	Sized integer, where *n* is the size, in bits, of the integer variable. The value of *n* can be 8, 16, 32, or 64. (**__int***n* is a Microsoft-specific keyword.)
	long	Type **long** (or **long int**) is an integral type that is larger than or equal to the size of type **int**.
		Objects of type **long** can be declared as **signed long** or **unsigned long**. **Signed long** is a synonym for **long**.
	long long	Larger than an unsigned **long**.
		Objects of type **long long** can be declared as **signed long long** or **unsigned long long**. **Signed long long** is a synonym for **long long**.

Floating	**float**	Type **float** is the smallest floating type.
	double	Type **double** is a floating type that is larger than or equal to type **float**, but shorter than or equal to the size of type **long double**.1
	long double1	Type **long double** is a floating type that is equal to type **double**.
Wide-character	**__wchar_t**	A variable of **__wchar_t** designates a wide-character or multibyte character type. By default, **wchar_t** is a native type but you can use /Zc:wchar_t- to make **wchar_t** a typedef for **unsigned short**. Use the L prefix before a character or string constant to designate the wide-character-type constant.

1 The representation of **long double** and **double** is identical. However, **long double** and **double** are separate types.

Microsoft Specific

The following table lists the amount of storage required for fundamental types in Microsoft C++.

Sizes of Fundamental Types

Type	Size
bool	1 byte
char, **unsigned char**, **signed char**	1 byte
short, **unsigned short**	2 bytes
int, **unsigned int**	4 bytes
__int *n*	8, 16, 32, 64, or 128 bits depending on the value of *n*. **__int***n* is Microsoft-specific.
long, **unsigned long**	4 bytes
float	4 bytes
double	8 bytes
long double1	8 bytes
long long	Equivalent to **__int64**.

1 The representation of **long double** and **double** is identical. However, **long double** and **double** are separate types.

END Microsoft Specific

See Data Type Ranges for a summary of the range of values of each type.

For more information about type conversion, see Standard Conversions.

See Also

Reference

Data Type Ranges

Types (C++)

Data Type Ranges

For 32-bit and 64-bit compilers, Microsoft Visual C++ recognizes the types shown in the table below. Note that the following type also have unsigned forms:

- **int** (**unsigned int**)
- **__int8** (**unsigned __int8**)
- **__int16** (**unsigned __int16**)
- **__int32** (**unsigned __int32**)
- **__int64** (**unsigned __int64**)
- **short** (**unsigned short**)
- **long** (**unsigned long**)
- **long long** (**unsigned long long**)

Type Name	Bytes	Other Names	Range of Values
int	4	signed	–2,147,483,648 to 2,147,483,647
unsigned int	4	unsigned	0 to 4,294,967,295
__int8	1	char	–128 to 127
unsigned __int8	1	unsigned char	0 to 255
__int16	2	short, short int, signed short int	–32,768 to 32,767
unsigned __int16	2	unsigned short, unsigned short int	0 to 65,535
__int32	4	signed, signed int, int	–2,147,483,648 to 2,147,483,647
unsigned __int32	4	unsigned, unsigned int	0 to 4,294,967,295
__int64	8	long long, signed long long	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned __int64	8	unsigned long long	0 to 18,446,744,073,709,551,615
bool	1	none	false or true
char	1	none	–128 to 127 by default 0 to 255 when compiled with /J
signed char	1	none	–128 to 127
unsigned char	1	none	0 to 255
short	2	short int, signed short int	–32,768 to 32,767
unsigned short	2	unsigned short int	0 to 65,535

long	4	long int, signed long int	–2,147,483,648 to 2,147,483,647
unsigned long	4	unsigned long int	0 to 4,294,967,295
long long	8	none (but equivalent to __int64)	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long	8	none (but equivalent to unsigned __int64)	0 to 18,446,744,073,709,551,615
enum	varies	none	See Remarks.
float	4	none	3.4E +/- 38 (7 digits)
double	8	none	1.7E +/- 308 (15 digits)
long double	same as double	none	same as double
wchar_t	2	__wchar_t	0 to 65,535

A variable of **__wchar_t** designates a wide-character or multibyte character type. By default **wchar_t** is a **typedef** for **unsigned short**. Use the **L** prefix before a character or string constant to designate the wide-character-type constant. When compiling with /Zc:wchar_t or /Za, the compiler can distinguish between an **unsigned short** and **wchar_t** for function overload purposes.

Signed and **unsigned** are modifiers that can be used with any integral type except **bool**. The **char** type behaves like **signed char** by default, but you can specify /J (compiler option) to make behave like **unsigned char** by default.

The **int** and **unsigned int** types have the size of the system word: four bytes. However, portable code should not depend on the size of **int**.

Microsoft C/C++ also features support for sized integer types. See __int8, __int16, __int32, __int64 for more information. Also see Integer Limits.

See Fundamental Types for more information on the restrictions of the sizes of each type.

The range of enumerated types varies depending on the language context and specified compiler flags. In C and C++ non-CLR enumerations, enumeration constants and values of enumerated types are expressed in terms of type **int**. C++ CLR enumerations may be backed by a specified scalar type. The range of an enumeration is defined as the smallest and largest values that can be stored in the bit-fields of the smallest and largest of its enumerators. However, the C++ compiler will allow any value within the range of the backing type of the enumeration to be specified (this may require a cast). For more information, refer to C Enumeration Declarations and C++ Enumeration Declarations.

See Also

Reference

C++ Keywords

Fundamental Types

void (C++)

When used as a function return type, the **void** keyword specifies that the function does not return a value. When used for a function's parameter list, void specifies that the function takes no parameters. When used in the declaration of a pointer, void specifies that the pointer is "universal."

If a pointer's type is **void ***, the pointer can point to any variable that is not declared with the **const** or **volatile** keyword. A void pointer cannot be dereferenced unless it is cast to another type. A void pointer can be converted into any other type of data pointer.

A void pointer can point to a function, but not to a class member in C++.

You cannot declare a variable of type void.

Example

```
// void.cpp
void vobject;   // C2182
void *pv;   // okay
int *pint; int i;
int main() {
   pv = &i;
   // Cast optional in C required in C++
   pint = (int *)pv;
}
```

See Also

Reference

C++ Keywords
Pointers to Type void
Fundamental Types

bool (C++)

This keyword is a built-in type. A variable of this type can have values true and false. Conditional expressions have the type **bool** and so have values of type **bool**. For example, `i!=0` now has **true** or **false** depending on the value of `i`.

The values **true** and **false** have the following relationship:

```
!false == true
!true == false
```

In the following statement:

```
if (condexpr1) statement1;
```

If `condexpr1` is **true**, `statement1` is always executed; if `condexpr1` is **false**, `statement1` is never executed.

When a postfix or prefix **++** operator is applied to a variable of type **bool**, the variable is set to **true**. The postfix or prefix **--** operator cannot be applied to a variable of this type.

The **bool** type participates in integral promotions. An r-value of type **bool** can be converted to an r-value of type **int**, with **false** becoming zero and **true** becoming one. As a distinct type, **bool** participates in overload resolution.

Microsoft Specific

In Visual C++4.2, the Standard C++ header files contained a **typedef** that equated **bool** with **int**. In Visual C++ 5.0 and later, **bool** is implemented as a built-in type with a size of 1 **byte**. That means that for Visual C++ 4.2, a call of **sizeof(bool)** yields 4, while in Visual C++ 5.0 and later, the same call yields 1. This can cause memory corruption problems if you have defined structure members of type **bool** in Visual C++ 4.2 and are mixing object files (OBJ) and/or DLLs built with the 4.2 and 5.0 or later compilers.

The **__BOOL_DEFINED** macro can be used to wrap code that is dependent on whether or not **bool** is supported.

Example

```
// bool.cpp
#include <stdio.h>

int main()
{
    #if !defined(__BOOL_DEFINED)
        printf_s("bool is not supported\n");
    #elif defined(__BOOL_DEFINED)
    printf_s("bool is supported\n");
    #endif
}
```

Output

```
bool is supported
```

END Microsoft Specific

See Also

Reference

C++ Keywords

Fundamental Types

false (C++)

The keyword is one of the two values for a variable of type bool or a conditional expression (a conditional expression is now a **true** Boolean expression). For example, if `i` is a variable of type **bool**, the `i = false;` statement assigns **false** to `i`.

Example

```
// bool_false.cpp
#include <stdio.h>

int main()
{
    bool bb = true;
    printf_s("%d\n", bb);
    bb = false;
    printf_s("%d\n", bb);
}
```

Output

```
1
0
```

See Also

Reference

C++ Keywords

true (C++)

```
bool-identifier = true ;
bool-expression logical-operator true ;
```

Remarks

This keyword is one of the two values for a variable of type bool or a conditional expression (a conditional expression is now a true boolean expression). If `i` is of type **bool**, then the statement `i = true;` assigns **true** to `i`.

Example

```
// bool_true.cpp
#include <stdio.h>
int main()
{
    bool bb = true;
    printf_s("%d\n", bb);
    bb = false;
    printf_s("%d\n", bb);
}
```

Output

```
1
0
```

See Also

Reference

C++ Keywords

__int8, __int16, __int32, __int64

Microsoft Specific

Microsoft C/C++ features support for sized integer types. You can declare 8-, 16-, 32-, or 64-bit integer variables by using the **__int***n* type specifier, where *n* is 8, 16, 32, or 64.

The following example declares one variable for each of these types of sized integers:

```
__int8 nSmall;      // Declares 8-bit integer
__int16 nMedium;    // Declares 16-bit integer
__int32 nLarge;     // Declares 32-bit integer
__int64 nHuge;      // Declares 64-bit integer
```

The types **__int8**, **__int16**, and **__int32** are synonyms for the ANSI types that have the same size, and are useful for writing portable code that behaves identically across multiple platforms. The **__int8** data type is synonymous with type **char**, **__int16** is synonymous with type **short**, and **__int32** is synonymous with type **int**. The **__int64** type has no ANSI equivalent.

Example

The following sample shows that an __int*xx* parameter will be promoted to **int**:

```
// sized_int_types.cpp

#include <stdio.h>

void func(int i) {
    printf_s("%s\n", __FUNCTION__);
}

int main()
{
    __int8 i8 = 100;
    func(i8);   // no void func(__int8 i8) function
                // __int8 will be promoted to int
}
```

Output

```
func
```

END Microsoft Specific

See Also

Reference

C++ Keywords

Fundamental Types

Data Type Ranges

__m64

Microsoft Specific

The **__m64** data type is for use with the MMX and 3DNow! intrinsics, and is defined in xmmintrin.h.

```
// data_types__m64.cpp
#include <xmmintrin.h>
int main()
{
   __m64 x;
}
```

Remarks

You should not access the **__m64** fields directly. You can, however, see these types in the debugger. A variable of type **__m64** maps to the MM[0-7] registers.

Variables of type **_m64** are automatically aligned on 8-byte boundaries.

The **__m64** data type is not supported on x64 processors. Applications that use __m64 as part of MMX intrinsics must be rewritten to use equivalent SSE and SSE2 intrinsics.

END Microsoft Specific

See Also

Reference

C++ Keywords

Fundamental Types

Data Type Ranges

__m128

Microsoft Specific

The **__m128 data** type, for use with the Streaming SIMD Extensions and Streaming SIMD Extensions 2 instructions intrinsics, is defined in xmmintrin.h.

```
// data_types__m128.cpp
#include <xmmintrin.h>
int main() {
   __m128 x;
}
```

Remarks

You should not access the **__m128** fields directly. You can, however, see these types in the debugger. A variable of type **__m128** maps to the XMM[0-7] registers.

Variables of type **_m128** are automatically aligned on 16-byte boundaries.

The **__m128** data type is not supported on Itanium Processor Family (IPF) processors.

END Microsoft Specific

See Also

Reference

C++ Keywords

Fundamental Types

Data Type Ranges

__m128d

Microsoft Specific

The **__m128d** data type, for use with the Streaming SIMD Extensions 2 instructions intrinsics, is defined in emmintrin.h.

```
// data_types__m128d.cpp
#include <emmintrin.h>
int main() {
   __m128d x;
}
```

Remarks

You should not access the **__m128d** fields directly. You can, however, see these types in the debugger. A variable of type **__m128** maps to the XMM[0-7] registers.

Variables of type **_m128d** are automatically aligned on 16-byte boundaries.

The **__m128d** data type is not supported on Itanium Processor Family (IPF) processors.

END Microsoft Specific

See Also

Reference

C++ Keywords

Fundamental Types

Data Type Ranges

__m128i

Microsoft Specific

The **__m128i** data type, for use with the Streaming SIMD Extensions 2 (SSE2) instructions intrinsics, is defined in emmintrin.h.

```
// data_types__m128i.cpp
#include <emmintrin.h>
int main() {
   __m128i x;
}
```

Remarks

You should not access the **__m128i** fields directly. You can, however, see these types in the debugger. A variable of type **__m128i** maps to the XMM[0-7] registers.

Variables of type **_m128i** are automatically aligned on 16-byte boundaries.

Note

Using variables of type **__m128i** will cause the compiler to generate the SSE2 **movdqa** instruction. This instruction does not cause a fault on Pentium III processors but will result in silent failure, with possible side effects caused by whatever instructio ns **movdqa** translates into on Pentium III processors.

The **__m128i** data type is not supported on Itanium Processor Family (IPF) processors.

END Microsoft Specific

See Also

Reference

C++ Keywords

Fundamental Types

Data Type Ranges

__ptr32, __ptr64

Microsoft Specific

__ptr32 represents a native pointer on a 32-bit system, while **__ptr64** represents a native pointer on a 64-bit system.

The following example shows how to declare each of these pointer types:

```
int * __ptr32 p32;
int * __ptr64 p64;
```

On a 32-bit system, a pointer declared with **__ptr64** is truncated to a 32-bit pointer. On a 64-bit system, a pointer declared with **__ptr32** is coerced to a 64-bit pointer.

Note
You cannot use **__ptr32** or **__ptr64** when compiling with **/clr:pure**. Otherwise, Compiler Error C2472 will be generated.

Example

The following example shows how to declare and allocate pointers with the **__ptr32** and **__ptr64** keywords.

```
#include <cstdlib>
#include <iostream>

int main()
{
    using namespace std;

    int * __ptr32 p32;
    int * __ptr64 p64;

    p32 = (int * __ptr32)malloc(4);
    *p32 = 32;
    cout << *p32 << endl;

    p64 = (int * __ptr64)malloc(4);
    *p64 = 64;
    cout << *p64 << endl;
}
```

Output

```
32
64
```

END Microsoft Specific

See Also

Reference

Fundamental Types

Derived Types

Derived types are new types that can be used in a program, and can include directly derived types and composed derivative types.

See Also

Reference

Types (C++)

Directly Derived Types

New types derived directly from existing types are types that point to, refer to, or (in the case of functions) transform type data to return a new type.

- Arrays of Variables or Objects
- Functions
- Pointers of a Given Type
- References to Objects
- Constants
- Pointers to Class Members

Arrays of Variables or Objects

Arrays of variables or objects can contain a specified number of a particular type. For example, an array derived from integers is an array of type **int**. The following code sample declares and defines an array of 10 **int** variables and an array of 5 objects of class `SampleClass`:

```
int         ArrayOfInt[10];
SampleClass aSampleClass[5];
```

Functions

Functions take zero or more arguments of given types and return objects of a specified type (or return nothing, if the function has a **void return** type).

Pointers of a Given Type

Pointers to variables or objects select an object in memory. The object can be global, local (or stack-frame), or dynamically allocated. Pointers to functions of a given type allow a program to defer selection of the function used on a particular object or objects until run time. The following example shows a definition of a pointer to a variable of type **char**:

```
char *szPathStr;
```

References to Objects

References to objects provide a convenient way to access objects by reference, but use the same syntax required to access objects by value. The following example demonstrates how to use references as arguments to functions and as return types of functions:

```
BigClassType &func( BigClassType &objname )
{
    objname.DoSomething();    // Note that member-of operator(.)
                              //  is used.
    objname.SomeData = 7;     // Data passed by non-const
                              //  reference is modifiable.
    return objname;
}
```

The important points about passing objects to a function by reference are:

- The syntax for accessing members of **class**, **struct**, and **union** objects is the same as if they were passed by value: the member-of operator (**.**).
- The objects are not copied prior to the function call; their addresses are passed. This can reduce the overhead of the function call.

Additionally, functions that return a reference need only accept the address of the object to which they refer, instead of a copy

of the whole object.

Although the preceding example describes references only in the context of communication with functions, references are not constrained to this use. Consider, for example, a case where a function needs to be an l-value — a common requirement for overloaded operators:

```
class Vector
{
public:
    Point &operator[]( int nSubscript ); // Function returns a
                                         //  reference type
};
```

The preceding declaration specifies a user-defined subscript operator for class `Vector`. In an assignment statement, two possible conditions occur:

```
Vector v1;
int    i;
Point p;
v1[7] = p;   // Vector used as an l-value
p = v1[7];   // Vector used as an r-value
```

The latter case, where `v1[7]` is used as an r-value, can be implemented without use of references. However, the former case, where `v1[7]` is used as an l-value, cannot be implemented easily without functions that are of reference type. Conceptually, the last two statements in the preceding example translate to the following code:

```
v1.operator[]( 7 ) = 3;    // Vector used as an l-value
i = v1.operator[]( 7 );    // Vector used as an r-value
```

When viewed in this way, it is easier to see that the first statement must be an l-value to be semantically correct on the left side of the assignment statement.

For more information about overloading, and about overloaded operators in particular, see Overloaded Operators.

You can also use references to declare a **const** reference to a variable or object. A reference declared as **const** retains the efficiency of passing an argument by reference, while preventing the called function from modifying the original object. Consider the following code:

```
// reference_to_objects2.cpp
// IntValue is a const reference.
#include <stdio.h>
void PrintInt( const int &IntValue )
{
    printf_s( "%d\n", IntValue );
}
int main()
{
}
```

Reference initialization is different from assignment to a variable of reference type. Consider the following code:

```
// reference_to_objects3.cpp
int main()
{
   int i = 7;
   int j = 5;

   // Reference initialization
   int &ri = i;  // Initialize ri to refer to i.
   int &rj = j;  // Initialize rj to refer to j.

   // Assignment
   ri = 3;       // i now equal to 3.
   rj = 12;      // j now equal to 12.
```

```
    ri = rj;      // i now equals j (12).
}
```

C++ Constants

See Literals for more information about the various kinds of constants allowed in C++.

Pointers to Class Members

These pointers define a type that points to a class member of a particular type. Such a pointer can be used by any object of the class type or any object of a type derived from the class type.

Use of pointers to class members enhances the type safety of the C++ language. Three new operators and constructs are used with pointers to members, as shown in the following table:

Operators and Constructs Used with Pointers to Members

Operator or Construct	Syntax	Use
::*	*type* **::*** *ptr-name*	Declaration of pointer to member. The *type* specifies the class name, and *ptr-name* specifies the name of the pointer to member. Pointers to members can be initialized. For example:
		`MyType::*pMyType = &MyType::i;`
.*	*obj-name* **.*** *ptr-name*	Dereference a pointer to a member using an object or object reference. For example:
		`int j = Object.*pMyType;`
->*	*obj-ptr* **->*** *ptr-name*	Dereference a pointer to a member using a pointer to an object. For example:
		`int j = pObject->*pMyType;`

Example

Consider this example that defines a class `AClass` and the derived type `pDAT`, which points to the member `I1`:

```
// pointers1.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

// Define class AClass.
class AClass
{
public:
    int I1;
    void Show() { cout << I1 << endl; }
};

// Define a derived type pDAT that points to I1 members of
// objects of type AClass.
int AClass::*pDAT = &AClass::I1;

int main()
{
    AClass aClass;             // Define an object of type AClass.
    AClass *paClass = &aClass;  // Define a pointer to that object.
```

```
    int i;

    aClass.*pDAT = 7777;     // Assign to aClass::I1 using .* operator.
    aClass.Show();

    i = paClass->*pDAT;       // Dereference a pointer
                              //  using ->* operator.
    cout << i << "\n";
}
```

Output

```
7777
7777
```

The pointer to member `pDAT` is a new type derived from class `AClass`. It is more strongly typed than a plain pointer to **int** because it points only to **int** members of class `AClass` (in this case, `I1`). Pointers to static members are plain pointers rather than pointers to class members. Consider the following example:

```
// pointers2.cpp
class HasStaticMember
{
public:
    static int SMember;
};
int HasStaticMember::SMember = 0;

int *pSMember = &HasStaticMember::SMember;

int main()
{
}
```

Note that the type of the pointer is "pointer to **int**," not "pointer to `HasStaticMember::int`."

Pointers to members can refer to member functions as well as member data.

Example

```
// pointers3.cpp
#include <stdio.h>

// Declare a base class, A, with a virtual function, Identify.
// (Note that in this context, struct is the same as class.)
struct A
{
    virtual void Identify() = 0; // No definition for class A.
};

// Declare a pointer to the Identify member function.
void (A::*pIdentify)() = &A::Identify;

// Declare class B derived from class A.
struct B : public A
{
    void Identify();
};

// Define Identify functions for classe B
void B::Identify()
{
    printf_s( "Identification is B::Identify\n" );
}

int main()
{
    B  BObject;          // Declare objects of type B
    A *pA;               // Declare pointer to type A.
```

```
    pA = &BObject;       // Make pA point to an object of type B.
    (pA->*pIdentify)();  // Call Identify function through pointer
                         //  to member pIdentify.
}
```

Output

```
Identification is B::Identify
```

The function is called through a pointer to type `A`. However, because the function is a virtual function, the correct function for the object to which `pA` refers is called.

See Also

Reference

Derived Types

Composed Derivative Types

This section describes the following composed derivative types:

- Classes
- Structures
- Unions

Information about aggregate types and initialization of aggregate types can be found in Initializing Aggregates.

C++ Classes

Classes are a composite group of member objects, functions to manipulate these members, and (optionally) access-control specifications to member objects and functions.

By grouping composite groups of objects and functions in classes, C++ enables programmers to create derivative types that define not only data but also the behavior of objects.

Class members default to private access and private inheritance. Classes are covered in Classes, Structures, and Unions. Access control is covered in Member-Access Control.

C++ Structures

C++ structures are the same as classes, except that all member data and functions default to public access, and inheritance defaults to public inheritance.

For more information about access control, see Member-Access Control.

C++ Unions

Unions enable programmers to define types capable of containing different kinds of variables in the same memory space. The following code shows how you can use a union to store several different types of variables:

Example

```
// unions.cpp
#include <stdio.h>
// compile with: /c
// Declare a union that can hold data of types char, int,
// or char *.
union ToPrint {
   char chVar;
   int iVar;
   char *szVar;
};

// Declare an enumerated type that describes what type to print.
enum PrintType {
   CHAR_T,
   INT_T,
   STRING_T
};

void Print( ToPrint Var, PrintType Type ) {
   switch( Type ) {
      case CHAR_T:
         printf_s( "%c", Var.chVar );
      break;
      case INT_T:
         printf_s( "%d", Var.iVar );
      break;
      case STRING_T:
         printf_s( "%s", Var.szVar );
      break;
    }
}
```

See Also

Reference

Derived Types

C++ Type Names

Synonyms for both fundamental and derived types can be defined using the typedef keyword. The following code illustrates the use of **typedef**:

```
typedef unsigned char BYTE;   // 8-bit unsigned entity.
typedef BYTE *        PBYTE;  // Pointer to BYTE.

BYTE Ch;                      // Declare a variable of type BYTE.
PBYTE pbCh;                   // Declare a pointer to a BYTE
                              //  variable.
```

The preceding example shows uniform declaration syntax for the fundamental type **unsigned char** and its derivative type **unsigned char ***. The **typedef** construct is also helpful in simplifying declarations. A **typedef** declaration defines a synonym, not a new, independent type. The following example declares a type name (`PVFN`) representing a pointer to a function that returns type **void**. The advantage of this declaration is that, later in the program, an array of these pointers is declared very simply.

```
// type_names.cpp
// Prototype two functions.
void func1(){};
void func2(){};

//  Define PVFN to represent a pointer to a function that
//   returns type void.
typedef void (*PVFN)();

int main()
{
   // Declare an array of pointers to functions.
   PVFN pvfn[] = { func1, func2 };

   // Invoke one of the functions.
   (*pvfn[1])();
}
```

See Also

Reference

Types (C++)

L-Values and R-Values

Expressions in C++ can evaluate to l-values or r-values. L-values are expressions that evaluate to a type other than **void** and that designate a variable.

L-values appear on the left side of an assignment statement (hence the "l" in l-value). Variables that would normally be l-values can be made nonmodifiable by using the **const** keyword; these cannot appear on the left of an assignment statement. Reference types are always l-values.

The term r-value is sometimes used to describe the value of an expression and to distinguish it from an l-value. All l-values are r-values but not all r-values are l-values.

Some examples of correct and incorrect usages are:

```
// lValues_rValues.cpp
int main() {
   int i, j, *p;
   i = 7;   // OK variable name is an l-value.
   7 = i;   // C2106 constant is an r-value.
   j * 4 = 7;   // C2106 expression j * 4 yields an r-value.
   *p = i;   // OK a dereferenced pointer is an l-value.

   const int ci = 7;
   ci = 9;   // C3892 ci is a nonmodifiable l-value
   ((i < 3) ? i : j) = 7;   // OK conditional operator returns l-value.
}
```

Note

The examples in this section illustrate correct and incorrect usage when operators are not overloaded. By overloading operat ors, you can make an expression such as `j * 4` an l-value.

See Also

Reference

Basic Concepts

Numerical Limits

The two standard include files, LIMITS.H and FLOAT.H, define the numerical limits, or minimum and maximum values that a variable of a given type can hold. These minimums and maximums are guaranteed to be portable to any C++ compiler that uses the same data representation as ANSI C. The LIMITS.H include file defines the numerical limits for integral types, and FLOAT.H defines the numerical limits for floating types.

See Also

Reference

Basic Concepts

Integer Limits

Microsoft Specific

The limits for integer types are listed in the following table. These limits are also defined in the standard header file LIMITS.H.

Limits on Integer Constants

Constant	Meaning	Value
CHAR_BIT	Number of bits in the smallest variable that is not a bit field.	8
SCHAR_MIN	Minimum value for a variable of type **signed char**.	–128
SCHAR_MAX	Maximum value for a variable of type **signed char**.	127
UCHAR_MAX	Maximum value for a variable of type **unsigned char**.	255 (0xff)
CHAR_MIN	Minimum value for a variable of type **char**.	–128; 0 if /J option used
CHAR_MAX	Maximum value for a variable of type **char**.	127; 255 if /J option used
MB_LEN_MAX	Maximum number of bytes in a multicharacter constant.	5
SHRT_MIN	Minimum value for a variable of type **short**.	–32768
SHRT_MAX	Maximum value for a variable of type **short**.	32767
USHRT_MAX	Maximum value for a variable of type **unsigned short**.	65535 (0xffff)
INT_MIN	Minimum value for a variable of type **int**.	–2147483648
INT_MAX	Maximum value for a variable of type **int**.	2147483647
UINT_MAX	Maximum value for a variable of type **unsigned int**.	4294967295 (0xffffffff)
LONG_MIN	Minimum value for a variable of type **long**.	–2147483648
LONG_MAX	Maximum value for a variable of type **long**.	2147483647
ULONG_MAX	Maximum value for a variable of type **unsigned long**.	4294967295 (0xffffffff)
_I64_MIN	Minimum value for a variable of type **__int64**	-9223372036854775808
_I64_MAX	Maximum value for a variable of type **__int64**	9223372036854775807
_UI64_MAX	Maximum value for a variable of type **unsigned __int64**	18446744073709551615 (0xffffffffffffffff)

If a value exceeds the largest integer representation, the Microsoft compiler generates an error.

END Microsoft Specific

See Also

Reference

Floating Limits

Floating Limits

Microsoft Specific

The following table lists the limits on the values of floating-point constants. These limits are also defined in the standard header file FLOAT.H.

Limits on Floating-Point Constants

Constant	Meaning	Value
FLT_DIG DBL_DIG LDBL_DIG	Number of digits, q, such that a floating-point number with q decimal digits can be rounded into a floating-point representation and back without loss of precision.	6 15 15
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	Smallest positive number x, such that x + 1.0 is not equal to 1.0.	1.192092896e–07F 2.2204460492503131e–016 2.2204460492503131e–016
FLT_GUARD		0
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	Number of digits in the radix specified by FLT_RADIX in the floating-point significand. The radix is 2; hence these values specify bits.	24 53 53
FLT_MAX DBL_MAX LDBL_MAX	Maximum representable floating-point number.	3.402823466e+38F 1.7976931348623158e+308 1.7976931348623158e+308
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	Maximum integer such that 10 raised to that number is a representable floating-point number.	38 308 308
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	Maximum integer such that FLT_RADIX raised to that number is a representable floating- point number.	128 1024 1024
FLT_MIN DBL_MIN LDBL_MIN	Minimum positive value.	1.175494351e–38F 2.2250738585072014e–308 2.2250738585072014e–308
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	Minimum negative integer such that 10 raised to that number is a representable floating- point number.	–37 –307 –307
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	Minimum negative integer such that FLT_RADIX raised to that number is a representable floating-point number.	–125 –1021 –1021
FLT_NORMALIZE		0
FLT_RADIX _DBL_RADIX _LDBL_RADIX	Radix of exponent representation.	2 2 2

FLT_ROUNDS _DBL_ROUNDS _LDBL_ROUNDS	Rounding mode for floating-point addition.	1 (near) 1 (near) 1 (near)

Note
The information in the table may differ in future versions of the product.

END Microsoft Specific

See Also

Reference

Integer Limits

Standard Conversions

The C++ language defines conversions between its fundamental types. It also defines conversions for pointer, reference, and pointer-to-member derived types. These conversions are called "standard conversions." (For more information about types, standard types, and derived types, see Types.)

This section discusses the following standard conversions:

- Integral promotions
- Integral conversions
- Floating conversions
- Floating and integral conversions
- Arithmetic conversions
- Pointer conversions
- Reference conversions
- Pointer-to-member conversions

Note

User-defined types can specify their own conversions. Conversion of user-defined types is covered in Constructors and Conversions.

The following code causes conversions (in this example, integral promotions):

```
long  lnum1, lnum2;
int   inum;

// inum promoted to type long prior to assignment.
lnum1 = inum;

// inum promoted to type long prior to multiplication.
lnum2 = inum * lnum2;
```

Note

The result of a conversion is an l-value only if it produces a reference type. For example, a user-defined conversion declared a s

```
operator int&()
```

Note

returns a reference and is an l-value. However, a conversion declared as

```
operator int()
```

Note

returns an object and is not an l-value.

See Also

Other Resources

C++ Language Reference

Integral Promotions

Objects of an integral type can be converted to another wider integral type (that is, a type that can represent a larger set of values). This widening type of conversion is called "integral promotion." With integral promotion, you can use the following in an expression wherever another integral type can be used:

- Objects, literals, and constants of type **char** and **short int**
- Enumeration types
- **int** bit fields
- Enumerators

C++ promotions are "value-preserving." That is, the value after the promotion is guaranteed to be the same as the value before the promotion. In value-preserving promotions, objects of shorter integral types (such as bit fields or objects of type **char**) are promoted to type **int** if **int** can represent the full range of the original type. If **int** cannot represent the full range of values, then the object is promoted to type **unsigned int**. Although this strategy is the same as that used by ANSI C, value-preserving conversions do not preserve the "signedness" of the object.

Value-preserving promotions and promotions that preserve signedness normally produce the same results. However, they can produce different results if the promoted object is one of the following:

- An operand of **/**, **%**, **/=**, **%=**, **<**, **<=**, **>**, or **>=**

  These operators rely on sign for determining the result. Therefore, value-preserving and sign-preserving promotions produce different results when applied to these operands.

- The left operand of **>>** or **>>=**

  These operators treat signed and unsigned quantities differently when performing a shift operation. For signed quantities, shifting a quantity right causes the sign bit to be propagated into the vacated bit positions. For unsigned quantities, the vacated bit positions are zero-filled.

- An argument to an overloaded function or operand of an overloaded operator that depends on the signedness of the type of that operand for argument matching. (See Overloaded Operators for more about defining overloaded operators.)

See Also
Reference
Standard Conversions

Integral Conversions

Integral conversions are performed between integral types. The integral types are **char**, **int**, and **long** (and the **short**, **signed**, and **unsigned** versions of these types).

This section describes the following types of integral conversions:

- Converting signed to unsigned
- Converting unsigned to signed
- Standard conversion

See Also

Reference

Standard Conversions

Converting Signed to Unsigned

Objects of signed integral types can be converted to corresponding unsigned types. When these conversions occur, the actual bit pattern does not change; however, the interpretation of the data changes. Consider this code:

Example

```
// conve__pluslang_Converting_Signed_to_Unsigned.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
int main()
{
    short  i = -3;
    unsigned short u;

    cout << (u = i) << "\n";
}
```

Output

```
65533
```

In the preceding example, a **signed short**, `i`, is defined and initialized to a negative number. The expression `(u = i)` causes `i` to be converted to an **unsigned short** prior to the assignment to `u`.

See Also

Reference

Integral Conversions

Converting Unsigned to Signed

Objects of unsigned integral types can be converted to corresponding signed types. However, such a conversion can cause misinterpretation of data if the value of the unsigned object is outside the range representable by the signed type, as demonstrated in the following example:

Example

```
// conve__pluslang_Converting_Unsigned_to_Signed.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
int main()
{
 short  i;
 unsigned short u = 65533;

 cout << (i = u) << "\n";
}
```

Output

```
-3
```

In the preceding example, `u` is an **unsigned short** integral object that must be converted to a signed quantity to evaluate the expression `(i = u)`. Because its value cannot be properly represented in a **signed short**, the data is misinterpreted as shown.

See Also

Reference

Integral Conversions

Standard Conversion

Objects of integral types can be converted to shorter signed or unsigned integral types. Such a conversion is called "standard conversion." It can result in loss of data if the value of the original object is outside the range representable by the shorter type.

Note
The compiler issues a high-level warning when a conversion to a shorter type takes place.

See Also

Reference

Integral Conversions

Floating Conversions

An object of a floating type can be safely converted to a more precise floating type — that is, the conversion causes no loss of significance. For example, conversions from **float** to **double** or from **double** to **long double** are safe, and the value is unchanged.

An object of a floating type can also be converted to a less precise type, if it is in a range representable by that type. (See Floating Limits for the ranges of floating types.) If the original value cannot be represented precisely, it can be converted to either the next higher or the next lower representable value. If no such value exists, the result is undefined. Consider the following example:

```
cout << (float)1E300 << endl;
```

The maximum value representable by type **float** is 3.402823466E38 — a much smaller number than 1E300. Therefore, the number is converted to infinity, and the result is 1.#INF.

See Also
Reference
Standard Conversions

Floating and Integral Conversions

Certain expressions can cause objects of floating type to be converted to integral types, or vice versa.

This section describes the following types of floating and integral conversions:

- Floating to integral
- Integral to floating

See Also
Reference
Standard Conversions

Floating to Integral

When an object of floating type is converted to an integral type, the fractional part is truncated. No rounding takes place in the conversion process. Truncation means that a number like 1.3 is converted to 1, and –1.3 is converted to –1.

See Also

Reference

Floating and Integral Conversions

Integral to Floating

When an object of integral type is converted to a floating type and the original value cannot be represented exactly, the result is either the next higher or the next lower representable value.

See Also

Reference

Floating and Integral Conversions

Arithmetic Conversions

Many binary operators (discussed in Expressions with Binary Operators) cause conversions of operands and yield results the same way. The way these operators cause conversions is called "usual arithmetic conversions." Arithmetic conversions of operands of different native types are performed as shown in the following table. Typedef types behave according to their underlying native types.

Conditions for Type Conversion

Conditions Met	Conversion
Either operand is of type **long double**.	Other operand is converted to type **long double**.
Preceding condition not met and either operand is of type **double**.	Other operand is converted to type **double**.
Preceding conditions not met and either operand is of type **float**.	Other operand is converted to type **float**.
Preceding conditions not met (none of the operands are of floating types).	Integral promotions are performed on the operands as follows: • If either operand is of type **unsigned long**, the other operand is converted to type **unsigned long**. • If preceding condition not met, and if either operand is of type **long** and the other of type **unsigned int**, both operands are converted to type **unsigned long**. • If the preceding two conditions are not met, and if either operand is of type **long**, the other operand is converted to type **long**. • If the preceding three conditions are not met, and if either operand is of type **unsigned int**, the other operand is converted to type **unsigned int**. • If none of the preceding conditions are met, both operands are converted to type **int**.

The following code illustrates the conversion rules described in the table:

```
// arithmetic_conversions.cpp
double dVal;
float fVal;
int iVal;
unsigned long ulVal;

int main() {
   // iVal converted to unsigned long
   // result of multiplication converted to double
   dVal = iVal * ulVal;

   // ulVal converted to float
   // result of addition converted to double
   dVal = ulVal + fVal;
}
```

The first statement in the preceding example shows multiplication of two integral types, `iVal` and `ulVal`. The condition met is that neither operand is of floating type and one operand is of type **unsigned int**. Therefore, the other operand, `iVal`, is converted to type **unsigned int**. The result is assigned to `dVal`. The condition met is that one operand is of type **double**; therefore, the **unsigned int** result of the multiplication is converted to type **double**.

The second statement in the preceding example shows addition of a **float** and an integral type, `fVal` and `ulVal`. The `ulVal` variable is converted to type **float** (third condition in the table). The result of the addition is converted to type **double** (second condition in the table) and assigned to `dVal`.

See Also

Reference

Standard Conversions

Pointer Conversions (C++)

Pointers can be converted during assignment, initialization, comparison, and other expressions. This section describes the following pointer conversion:

- Null pointers
- Pointers to type void
- Pointers to objects
- Pointers to functions
- Pointers to classes
- Expressions
- Pointers modified by const or volatile

See Also

Reference

Standard Conversions

Null Pointers

An integral constant expression that evaluates to zero, or such an expression cast to a pointer type, is converted to a pointer called the "null pointer." This pointer is guaranteed to compare unequal to a pointer to any valid object or function (except for pointers to based objects, which can have the same offset and still point to different objects).

See Also

Reference

Pointer Conversions (C++)

Pointers to Type void

Pointers to type **void** can be converted to pointers to any other type, but only with an explicit type cast (unlike in C). (See Expressions with Explicit Type Conversions for more information about type casts.) A pointer to any type can be converted implicitly to a pointer to type **void**.

A pointer to an incomplete object of a type can be converted to a pointer to **void** (implicitly) and back (explicitly). The result of such a conversion is equal to the value of the original pointer. An object is considered incomplete if it is declared, but there is insufficient information available to determine its size or base class.

See Also

Reference

Pointer Conversions (C++)

Pointers to Objects

A pointer to any object that is not **const** or **volatile** can be implicitly converted to a pointer of type **void ***.

See Also

Reference

Pointer Conversions (C++)

Pointers to Functions

A pointer to a function can be converted to type **void ***, if type **void *** is large enough to hold that pointer.

See Also

Reference

Pointer Conversions (C++)

Pointers to Classes

There are two cases in which a pointer to a class can be converted to a pointer to a base class.

Remarks

The first case is when the specified base class is accessible and the conversion is unambiguous. (See Multiple Base Classes for more information about ambiguous base-class references.)

Whether a base class is accessible depends on the kind of inheritance used in derivation. Consider the inheritance illustrated in the following figure.

Inheritance Graph for Illustration of Base-Class Accessibility

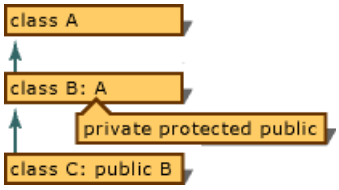

The following table shows the base-class accessibility for the situation illustrated in the figure.

Base-Class Accessibility

Type of Function	Derivation	Conversion from B* to A* Legal?
External (not class-scoped) function	Private	No
	Protected	No
	Public	Yes
B member function (in B scope)	Private	Yes
	Protected	Yes
	Public	Yes
C member function (in C scope)	Private	No
	Protected	Yes
	Public	Yes

The second case in which a pointer to a class can be converted to a pointer to a base class is when you use an explicit type conversion. (See Expressions with Explicit Type Conversions for more information about explicit type conversions.)

The result of such a conversion is a pointer to the "subobject," the portion of the object that is completely described by the base class.

The following code defines two classes, `A` and `B`, where `B` is derived from `A`. (For more information on inheritance, see Derived Classes.) It then defines `bObject`, an object of type `B`, and two pointers (`pA` and `pB`) that point to the object.

```
// conve__pluslang_Pointers_to_Classes.cpp
// C2039 expected
class A
{
public:
    int AComponent;
    int AMemberFunc();
};
```

```
class B : public A
{
public:
    int BComponent;
    int BMemberFunc();
};
int main()
{
   B bObject;
   A *pA = &bObject;
   B *pB = &bObject;

   pA->AMemberFunc();   // OK in class A
   pB->AMemberFunc();   // OK: inherited from class A
   pA->BMemberFunc();   // Error: not in class A
}
```

The pointer `pA` is of type `A *`, which can be interpreted as meaning "pointer to an object of type `A`." Members of `bObject` (such as `BComponent` and `BMemberFunc`) are unique to type `B` and are therefore inaccessible through `pA`. The `pA` pointer allows access only to those characteristics (member functions and data) of the object that are defined in class `A`.

See Also

Reference

Pointer Conversions (C++)

Pointer Expressions (C++)

Any expression with an array type can be converted to a pointer of the same type. The result of the conversion is a pointer to the first array element. The following example demonstrates such a conversion:

```
char szPath[_MAX_PATH]; // Array of type char.
char *pszPath = szPath; // Equals &szPath[0].
```

An expression that results in a function returning a particular type is converted to a pointer to a function returning that type, except when:

- The expression is used as an operand to the address-of operator (**&**).
- The expression is used as an operand to the function-call operator.

See Also

Reference

Pointer Conversions (C++)

Pointers Modified by const or volatile

C++ does not supply a standard conversion from a **const** or **volatile** type to a type that is not **const** or **volatile**. However, any sort of conversion can be specified using explicit type casts (including conversions that are unsafe).

Note
C++ pointers to members, except pointers to static members, are different from normal pointers and do not have the same standard conversions. Pointers to static members are normal pointers and have the same conversions as normal pointers. (See Directly Derived Types for more information.)

See Also

Reference

Pointer Conversions (C++)

Reference Conversions

A reference to a class can be converted to a reference to a base class in the following cases:

- The specified base class is accessible (as defined in Pointers to Classes).
- The conversion is unambiguous. (See Multiple Base Classes for more information about ambiguous base-class references.)

The result of the conversion is a pointer to the subobject that represents the base class.

See Also
Reference
Standard Conversions

Pointer-to-Member Conversions

Pointers to class members can be converted during assignment, initialization, comparison, and other expressions. This section describes the following pointer-to-member conversions:

- Integral constant expressions
- Pointers to base-class members

See Also

Reference

Standard Conversions

Integral Constant Expressions

An integral constant expression that evaluates to zero is converted to a pointer called the "null pointer." This pointer is guaranteed to compare unequal to a pointer to any valid object or function (except for pointers to based objects, which can have the same offset and still point to different objects).

The following code illustrates the definition of a pointer to member `i` in class `A`. The pointer, `pai`, is initialized to 0, which is the null pointer.

```
// conve__pluslang_Integral_Constant_Expressions.cpp
class A
{
public:
 int i;
};

int A::*pai = 0;

int main()
{
}
```

See Also

Reference

Pointer-to-Member Conversions

Pointers to Base-Class Members

A pointer to a member of a base class can be converted to a pointer to a member of a class derived from it, when the following conditions are met:

- The inverse conversion, from pointer to derived class to base-class pointer, is accessible.
- The derived class does not inherit virtually from the base class.

When the left operand is a pointer to member, the right operand must be of pointer-to-member type or be a constant expression that evaluates to 0. This assignment is valid only in the following cases:

- The right operand is a pointer to a member of the same class as the left operand.
- The left operand is a pointer to a member of a class derived publicly and unambiguously from the class of the right operand.

See Also

Reference

Pointer-to-Member Conversions

Expressions (C++)

This section describes C++ expressions. Expressions are sequences of operators and operands that are used for one or more of these purposes:

- Computing a value from the operands.
- Designating objects or functions.
- Generating "side effects." (Side effects are any actions other than the evaluation of the expression — for example, modifying the value of an object.)

In C++, operators can be overloaded and their meanings can be user-defined. However, their precedence and the number of operands they take cannot be modified. This section describes the syntax and semantics of operators as they are supplied with the language, not overloaded. In addition to types of expressions and semantics of expressions, the following topics are covered:

- Primary expressions
- Scope resolution operator
- Postfix expressions
- Expressions with unary operators
- Expressions with binary operators
- Conditional operator
- Constant expressions
- Expressions with explicit type conversions
- Casting operators
- Run-time type information

Topics on operators in other sections:

- Table of C++ operators
- Overloaded operators
- typeid

Note
Operators for built-in types cannot be overloaded; their behavior is predefined.

See Also

Other Resources

C++ Language Reference

Types of Expressions

C++ expressions are divided into several categories:

- Primary expressions. These are the building blocks from which all other expressions are formed.
- Postfix expressions. These are primary expressions followed by an operator — for example, the array subscript or postfix increment operator.
- Expressions formed with unary operators. Unary operators act on only one operand in an expression.
- Expressions formed with binary operators. Binary operators act on two operands in an expression.
- Expressions with the conditional operator. The conditional operator is a ternary operator — the only such operator in the C++ language — and takes three operands.
- Constant expressions. Constant expressions are formed entirely of constant data.
- Expressions with explicit type conversions. Explicit type conversions, or "casts," can be used in expressions.
- Expressions with pointer-to-member operators.
- Casting. Type-safe "casts" can be used in expressions.
- Run-Time Type Information. Determine the type of an object during program execution.

See Also

Reference

Expressions (C++)

Primary Expressions

Primary expressions are the building blocks of more complex expressions. They are literals, names, and names qualified by the scope-resolution operator (**::**). A primary expression may have any of the following forms:

```
        literal
this
:: name
name
( expression )
```

A *literal* is a constant primary expression. Its type depends on the form of its specification. See Literals for complete information about specifying literals.

The **this** keyword is a pointer to a class object. It is available within nonstatic member functions and points to the instance of the class for which the function was invoked. The **this** keyword cannot be used outside the body of a class-member function.

The type of the **this** pointer is *type* ***const** (where *type* is the class name) within functions not specifically modifying the **this** pointer. The following example shows member function declarations and the types of **this**:

```
// expre_Primary_Expressions.cpp
// compile with: /LD
class Example
{
public:
    void Func();          //  * const this
    void Func() const;    //  const * const this
    void Func() volatile; //  volatile * const this
};
```

See Type of this Pointer for more information about modifying the type of the **this** pointer.

The scope-resolution operator (**::**) followed by a name constitutes a primary expression. Such names must be names at global scope, not member names. The type of this expression is determined by the declaration of the name. It is an l-value (that is, it can appear on the left hand side of an assignment operator expression) if the declaring name is an l-value. The scope-resolution operator allows a global name to be referred to, even if that name is hidden in the current scope. See Scope for an example of how to use the scope-resolution operator.

An expression enclosed in parentheses is a primary expression whose type and value are identical to those of the unparenthesized expression. It is an l-value if the unparenthesized expression is an l-value.

In the context of the primary expression syntax given above, *name* means anything in the syntax described for Names, although when using the scope-resolution operator before the name, types of names that can only occur in a class are not allowed. This includes user-defined conversion function names, and destructor names.

Examples of primary expressions include:

```
100 // literal
'c' // literal
this // in a member function, a pointer to the class instance
::func // a global function
::operator + // an global operator function
::A::B // a global qualified name
( i + 1 ) // a parenthesized expression
```

The examples below are all considered *names*, and hence primary expressions, in various forms:

```
MyClass // a identifier
MyClass::f // a qualified name
operator = // an operator function name
operator char* // an converstion operator function name
~MyClass // a destructor name
```

```
A::B   // a qualified name
A<int> // a template id
```

See Also

Reference

Types of Expressions

Names

```
identifier
qualified-name
operator-function-name
conversion-function-name
~ class-name
template-identifier
template name
qualified-class-name :: template name
```

Remarks

Any *identifier* that has been declared is a *name*. A qualified name is a name composed with the scope resolution operator. See qualified names.

An *operator-function-name* is a name that is declared in the form

operator *operator-name***(** *argument1* [**,** *argument2*] **);**

See Overloaded Operators for more information about declaration of *operator-function-name*.

A *conversion-function-name* is a name that is declared in the form

*operator type-name***( )**

Note
You can supply a derivative type name such as **char *** in place of the *type-name* when declaring a conversion function.

Conversion functions supply conversions to and from user-defined types. For more information about user-supplied conversions, see Conversion Functions.

A name declared as **~** *class-name* is taken as the "destructor" for objects of a class type. Destructors typically perform cleanup operations at the end of an object's lifetime. For information on destructors, see Destructors.

There are also several forms of names used for template types and template functions.

Template identifiers include the template class name followed by angled brackets enclosing the template argument list, which may include types and expressions, depending on how the template was declared. Examples of *template-identifers* are:

```
A<int> // class A takes a type as a template argument
A<int, char> // class A takes two types as template arguments
A<4> // class A takes an int as a template argument
A<> // class A is a template with a default argument
```

Qualified names of template functions may include the template keyword. The template keyword is not allowed in a name unless preceded by the scope resolution operator, the **->** operator, the **.** operator, or a qualified name. The following are names of functions:

```
::template f<int> // f is a global template function
A::template g<100> //g is a template member function of class A
```

Similarly, the name of a class template can be specified using the template keyword if preceded by scope resolution or qualified name. The following are names of types:

```
::template A<int> // equivalent to A<int>
Outer::template Inner<int> // equivalent to Outer::Inner<int>
```

For more information on templates, see Templates.

See Also

Reference

Primary Expressions

Qualified Names

Specifies the scope.

```
qualified-class-name :: name
```

Remarks

If a *qualified-class-name* is followed by the scope-resolution operator (**::**) and then the name of a member of either that class or a base of that class, then the scope-resolution operator is considered a *qualified-name*. The type of a *qualified-name* is the same as the type of the member, and the result of a *qualified-name* expression is the member. If the member is an l-value, then the *qualified-name* is also an l-value. For information about declaring a *qualified-class-name*, see Type Specifiers or Class Names.

The *class-name* part of a *qualified-class-name* can be hidden by redeclaration of the same name in the current or enclosing scope; the *class-name* is still found and used. See Scope for an example of how to use a *qualified-class-name* to access a hidden *class-name*.

Note
Class constructors and destructors of the form *class-name* **::** *class-name* and *class-name* **::** **~** *class-name*, respectively, must refer to the same *class-name*.

A name with more than one qualification, such as the following, designates a member of a nested class:

class-name **::** *class-name* **::** *name*

See Also

Reference

Primary Expressions

Scope Resolution Operator: ::

You can tell the compiler to use the global identifier rather than the local identifier by prefixing the identifier with **::**, the scope resolution operator.

```
:: identifier
class-name :: identifier
namespace :: identifier
```

Remarks

The *identifier* can be a variable or a function.

If you have nested local scopes, the scope resolution operator does not provide access to identifiers in the next outermost scope. It provides access to only the global identifiers.

Example

This example has two variables named `amount`. The first is global and contains the value 123. The second is local to the main function. The scope resolution operator tells the compiler to use the global `amount` instead of the local one.

```
// expre_ScopeResolutionOperator.cpp
// compile with: /EHsc
// Demonstrate scope resolution operator
#include <iostream>

using namespace std;

int amount = 123;   // A global variable

int main() {
   int amount = 456;   // A local variable
   cout  << ::amount << endl   // Print the global variable
         << amount << endl;    // Print the local variable
}
```

See Also

Reference

C++ Operators

Operator Precedence and Associativity

Namespaces (C++)

Names and Qualified Names

Postfix Expressions

Postfix expressions consist of primary expressions or expressions in which postfix operators follow a primary expression. The postfix operators are listed in the following table.

Postfix Operators

Operator Name	Operator Notation
Subscript operator	**[ ]**
Function call operator	**( )**
Explicit type conversion operator	*type-name* **( )**
Member access operator	**.** or **–>**
Postfix increment operator	**++**
Postfix decrement operator	**—**

The following syntax describes possible postfix expressions:

```
        primary-expression
postfix-expression [ expression ]
postfix-expression ( expression-list<SUB>opt</SUB> )
simple-type-name ( expression-list<SUB>opt</SUB> )
postfix-expression . name
postfix-expression -> name
postfix-expression ++
postfix-expression --
cast-keyword < typename > (expression )
typeid ( typename )
```

The *postfix-expression* above may be a primary expression or another postfix expression. See primary expressions. Postfix expressions group left to right, thus allowing the expressions to be chained together as follows:

```
func(1)->GetValue()++
```

In the above expression, func is a primary expression, func(1) is a function postfix expression, func(1)->GetData is a postfix expression specifying a member of the class, func(1)->GetData() is another function postfix expression, and the entire expression is a postfix expression incrementing the return value of GetData. The meaning of the expression as a whole is "call func passing 1 as an argument and get a pointer to a class as a return value. Then call GetValue() on that class, then increment the value returned.

The expressions listed above are assignment expressions, meaning that the result of these expressions must be an r-value.

The postfix expression form

```
simple-type-name ( expression-list )
```

indicates the invocation of the constructor. If the simple-type-name is a fundamental type, the expression list must be a single expression, and this expression indicates a cast of the expression's value to the fundamental type. This type of cast expression mimics a constructor. Because this form allows fundamental types and classes to be constructed using the same syntax, this form is especially useful when defining template classes.

The *cast-keyword* is one of **dynamic_cast**, **static_cast** or **reinterpret_cast**. More information may be found in dynamic_cast, static_cast and reinterpet_cast.

The **typeid** operator is considered a postfix expression. See typeid operator.

See Also

Reference

Types of Expressions

Subscript Operator: []

```
postfix-expression [ expression ]
```

Remarks

A postfix expression (which can also be a primary expression) followed by the subscript operator, **[ ]**, specifies array indexing.

For information about managed arrays, see array (Visual C++).

Usually, the value represented by *postfix-expression* is a pointer value, such as an array identifier, and *expression* is an integral value (including enumerated types). However, all that is required syntactically is that one of the expressions be of pointer type and the other be of integral type. Thus the integral value could be in the *postfix-expression* position and the pointer value could be in the brackets in the *expression* or subscript position. Consider the following code fragment:

```
    int nArray[5] = { 0, 1, 2, 3, 4 };
    cout << nArray[2] << endl;            // prints "2"
    cout << 2[nArray] << endl;            // prints "2"
```

In the preceding example, the expression `nArray[2]` is identical to `2[nArray]`. The reason is that the result of a subscript expression *e1***[** *e2* **]** is given by:

(** **(** *e2* **)** + **(e1***)** **)**

The address yielded by the expression is not *e2* bytes from the address *e1*. Rather, the address is scaled to yield the next object in the array *e2*. For example:

```
double aDbl[2];
```

The addresses of `aDb[0]` and `aDb[1]` are 8 bytes apart — the size of an object of type **double**. This scaling according to object type is done automatically by the C++ language and is defined in Additive Operators where addition and subtraction of operands of pointer type is discussed.

A subscript expression can also have multiple subscripts, as follows:

expression1 **[***expression2***]** **[***expression3***]**...

Subscript expressions associate from left to right. The leftmost subscript expression, *expression1***[***expression2***]**, is evaluated first. The address that results from adding *expression1* and *expression2* forms a pointer expression; then *expression3* is added to this pointer expression to form a new pointer expression, and so on until the last subscript expression has been added. The indirection operator (*****) is applied after the last subscripted expression is evaluated, unless the final pointer value addresses an array type.

Expressions with multiple subscripts refer to elements of multidimensional arrays. A multidimensional array is an array whose elements are arrays. For example, the first element of a three-dimensional array is an array with two dimensions. The following example declares and initializes a simple two-dimensional array of characters:

```
// expre_Subscript_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
#define MAX_ROWS 2
#define MAX_COLS 2

int main() {
   char c[ MAX_ROWS ][ MAX_COLS ] = { { 'a', 'b' }, { 'c', 'd' } };
   for ( int i = 0; i < MAX_ROWS; i++ )
      for ( int j = 0; j < MAX_COLS; j++ )
         cout << c[ i ][ j ] << endl;
}
```

See Also

Reference

Postfix Expressions

C++ Operators

Operator Precedence and Associativity

Arrays (C++)

Concepts

One-Dimensional Arrays

Multidimensional Arrays (C)

Positive and Negative Subscripts

The first element of an array is element 0. The range of a C++ array is from *array***[**0**]** to *array***[***size* – 1**]**. However, C++ supports positive and negative subscripts. Negative subscripts must fall within array boundaries or results are unpredictable. The following code illustrates this concept:

```
// expre_Positive_and_Negative_Subscripts.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main() {
   int iNumberArray[1024];
   int *iNumberLine = &iNumberArray[512];

   cout << iNumberLine[-256] << "\n";   // OK

   // Unpredictable
   // cout << iNumberArray[-256] << "\n";}
}
```

The negative subscript in `iNumberArray` can produce a run-time error because it yields an address 256 bytes lower in memory than the origin of the array. The object `iNumberLine` is initialized to the middle of `iNumberArray`; it is therefore possible to use both positive and negative array indexes on i0.t. Array subscript errors do not generate compile-time errors, but they yield unpredictable results.

The subscript operator is commutative. Therefore, the expressions *array***[***index***]** and *index***[***array***]** are guaranteed to be equivalent as long as the subscript operator is not overloaded (see Overloaded Operators). The first form is the most common coding practice, but either works.

See Also

Reference

Postfix Expressions

Function Call Operator: ()

A postfix-expression followed by the function-call operator, **( )**, specifies a function call.

```
postfix-expression ( [argument-expression-list ] )
```

Remarks

The arguments to the function-call operator are zero or more expressions separated by commas — the actual arguments to the function.

The *postfix-expression* must evaluate to a function address (for example, a function identifier or the value of a function pointer), and *argument-expression-list* is a list of expressions (separated by commas) whose values (the arguments) are passed to the function. The *argument-expression-list* argument can be empty.

The *postfix-expression* must be of one of these types:

- Function returning type `T`. An example declaration is

  ```
  T func( int i )
  ```

- Pointer to a function returning type `T`. An example declaration is

  ```
  T (*func)( int i )
  ```

- Reference to a function returning type `T`. An example declaration is

  ```
  T (&func)(int i)
  ```

- Pointer-to-member function dereference returning type `T`. Example function calls are

  ```
  (pObject->*pmf)();
  (Object.*pmf)();
  ```

Example

The following example calls the standard library function `strcat` with two arguments:

```
// expre_Function_Call_Operator.cpp
// compile with: /EHsc

#include <iostream>
#include <string>

// STL name space
using namespace std;

int main()
{
    enum
    {
        sizeOfBuffer = 20
    };

    char s1[ sizeOfBuffer ] = "Welcome to ";
    char s2[ ] = "C++";

    strcat_s( s1, sizeOfBuffer, s2 );
```

```
    cout << s1 << endl;
}
```

Output

```
Welcome to C++
```

See Also

Reference

Postfix Expressions
C++ Operators
Operator Precedence and Associativity
Function Declarations

Concepts

Function Call (C)

Formal and Actual Arguments

Calling programs pass information to called functions in "actual arguments." The called functions access the information using corresponding "formal arguments."

When a function is called, the following tasks are performed:

- All actual arguments (those supplied by the caller) are evaluated. There is no implied order in which these arguments are evaluated, but all arguments are evaluated and all side effects completed prior to entry to the function.
- Each formal argument is initialized with its corresponding actual argument in the expression list. (A formal argument is an argument that is declared in the function header and used in the body of a function.) Conversions are done as if by initialization — both standard and user-defined conversions are performed in converting an actual argument to the correct type. The initialization performed is illustrated conceptually by the following code:

  ```
  void Func( int i ); // Function prototype
  ...
  Func( 7 );          // Execute function call
  ```

  The conceptual initializations prior to the call are:

  ```
  int Temp_i = 7;
  Func( Temp_i );
  ```

  Note that the initialization is performed as if using the equal-sign syntax instead of the parentheses syntax. A copy of `i` is made prior to passing the value to the function. (For more information, see Initializers and Conversions, Initialization Using Special Member Functions, and Explicit Initialization.

  Therefore, if the function prototype (declaration) calls for an argument of type **long**, and if the calling program supplies an actual argument of type **int**, the actual argument is promoted using a standard type conversion to type **long** (see Standard Conversions).

  It is an error to supply an actual argument for which there is no standard or user-defined conversion to the type of the formal argument.

  For actual arguments of class type, the formal argument is initialized by calling the class's constructor. (See Constructors for more about these special class member functions.)

- The function call is executed.

The following program fragment demonstrates a function call:

```
// expre_Formal_and_Actual_Arguments.cpp
void func( long param1, double param2 );

int main()
{
    long i = 1;
    double j = 2;

    // Call func with actual arguments i and j.
    func( i, j );
}

// Define func with formal parameters param1 and param2.
void func( long param1, double param2 )
{
}
```

When `func` is called from main, the formal parameter `param1` is initialized with the value of `i` (`i` is converted to type **long** to correspond to the correct type using a standard conversion), and the formal parameter `param2` is initialized with the value of `j`

(`j` is converted to type **double** using a standard conversion).

See Also

Reference

Postfix Expressions

Treatment of Argument Types

Formal arguments declared as const types cannot be changed within the body of a function. Functions can change any argument that is not of type **const**. However, the change is local to the function and does not affect the actual argument's value unless the actual argument was a reference to an object not of type **const**.

The following functions illustrate some of these concepts:

```
// expre_Treatment_of_Argument_Types.cpp
int func1( const int i, int j, char *c ) {
   i = 7;   // C3892 i is const.
   j = i;   // value of j is lost at return
   *c = 'a' + j;   // changes value of c in calling function
   return i;
}

double& func2( double& d, const char *c ) {
   d = 14.387;   // changes value of d in calling function.
   *c = 'a';   // C3892 c is a pointer to a const object.
    return d;
}
```

See Also

Reference

Postfix Expressions

Ellipses and Default Arguments

Functions can be declared to accept fewer arguments than specified in the function definition, using one of two methods: ellipsis (`...`) or default arguments.

Ellipses denote that arguments may be required but that the number and types are not specified in the declaration. This is normally poor C++ programming practice because it defeats one of the benefits of C++: type safety. Different conversions are applied to functions declared with ellipses than to those functions for which the formal and actual argument types are known:

- If the actual argument is of type **float**, it is promoted to type **double** prior to the function call.
- Any signed or unsigned **char**, **short**, enumerated type, or bit field is converted to either a signed or an unsigned **int** using integral promotion.
- Any argument of class type is passed by value as a data structure; the copy is created by binary copying instead of by invoking the class's copy constructor (if one exists).

Ellipses, if used, must be declared last in the argument list. For more information about passing a variable number of arguments, see the discussion of va_arg, va_start, and va_list in the *Run-Time Library Reference*.

Default arguments enable you to specify the value an argument should assume if none is supplied in the function call. The following code fragment shows how default arguments work. For more information about restrictions on specifying default arguments, see Default Arguments.

```
// expre_Ellipses_and_Default_Arguments.cpp
// compile with: /EHsc
#include <iostream>

// Declare the function print that prints a string,
// then a terminator.
void print( const char *string,
            const char *terminator = "\n" );

int main()
{
    print( "hello," );
    print( "world!" );

    print( "good morning", ", " );
    print( "sunshine." );
}

using namespace std;
// Define print.
void print( const char *string, const char *terminator )
{
    if( string != NULL )
        cout << string;

    if( terminator != NULL )
        cout << terminator;
}
```

The preceding program declares a function, `print`, that takes two arguments. However, the second argument, `terminator`, has a default value, `"\n"`. In **main**, the first two calls to `print` allow the default second argument to supply a new line to terminate the printed string. The third call specifies an explicit value for the second argument. The output from the program is

```
hello,
world!
good morning, sunshine.
```

See Also

Reference

Function-Call Results

A function call evaluates to an r-value unless the function is declared as a reference type. Functions with reference return type evaluate to l-values, and can be used on the left side of an assignment statement as follows:

```
// expre_Function_Call_Results.cpp
// compile with: /EHsc
#include <iostream>
class Point
{
public:
    // Define "accessor" functions as
    // reference types.
    unsigned& x() { return _x; }
    unsigned& y() { return _y; }
private:
    unsigned _x;
    unsigned _y;
};

using namespace std;
int main()
{
    Point ThePoint;

    ThePoint.x() = 7;           // Use x() as an l-value.
    unsigned y = ThePoint.y();  // Use y() as an r-value.

    // Use x() and y() as r-values.
    cout << "x = " << ThePoint.x() << "\n"
         << "y = " << ThePoint.y() << "\n";
}
```

The preceding code defines a class called `Point`, which contains private data objects that represent *x* and *y* coordinates. These data objects must be modified and their values retrieved. This program is only one of several designs for such a class; use of the `GetX` and `SetX` or `GetY` and `SetY` functions is another possible design.

Functions that return class types, pointers to class types, or references to class types can be used as the left operand to member-selection operators. Therefore, the following code is legal:

```
// expre_Function_Results2.cpp
class A {
public:
   A() {}
   A(int i) {}
   int SetA( int i ) {
      return (I = i);
   }

   int GetA() {
      return I;
   }

private:
   int I;
};

A func1() {
   A a = 0;
   return a;
}

A* func2() {
   A *a = new A();
```

```
   return a;
}

A& func3() {
   A *a = new A();
   A &b = *a;
   return b;
}

int main() {
   int iResult = func1().GetA();
   func2()->SetA( 3 );
   func3().SetA( 7 );
}
```

Functions can be called recursively. For more information about function declarations, see Function Specifiers and Member Functions. Related material is in Program and Linkage.

See Also

Reference

Postfix Expressions

Member Access Operators: . and ->

```
postfix-expression . name
postfix-expression -> name
```

Remarks

The member access operators **.** and **->** are used to refer to members of structures, unions, and classes. Member access expressions have the value and type of the selected member.

There are two forms of member access expressions:

1. In the first form, *postfix-expression* represents a value of struct, class, or union type, and *name* names a member of the specified structure, union, or class. The value of the operation is that of *name* and is an l-value if *postfix-expression* is an l-value.
2. In the second form, *postfix-expression* represents a pointer to a structure, union, or class, and *name* names a member of the specified structure, union, or class. The value is that of *name* and is an l-value. The **–>** operator dereferences the pointer. Therefore, the expressions *e***–>***member* and **(***e***)***.member* (where *e* represents a pointer) yield identical results (except when the operators **–>** or ***** are overloaded).

Example

The following example demonstrates both forms of the member access operator.

```
// expre_Selection_Operator.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

struct Date {
   Date(int i, int j, int k) : day(i), month(j), year(k){}
   int month;
   int day;
   int year;
};

int main() {
   Date mydate(1,1,1900);
   mydate.month = 2;
   cout  << mydate.month << "/" << mydate.day
         << "/" << mydate.year << endl;

   Date *mydate2 = new Date(1,1,2000);
   mydate2->month = 2;
   cout  << mydate2->month << "/" << mydate2->day
         << "/" << mydate2->year << endl;
   delete mydate2;
}
```

Output

```
2/1/1900
2/1/2000
```

See Also

Reference

Postfix Expressions
C++ Operators
Operator Precedence and Associativity
Classes, Structures, and Unions

Concepts

Postfix Increment and Decrement Operators: ++ and --

```
postfix-expression ++
postfix-expression --
```

Remarks

C++ provides prefix and postfix increment and decrement operators; this section describes only the postfix increment and decrement operators. (For more information, see Prefix Increment and Decrement Operators.) The difference between the two is that in the postfix notation, the operator appears after *postfix-expression*, whereas in the prefix notation, the operator appears before *expression*. The following example shows a postfix-increment operator:

```
i++;
```

The effect of applying the postfix increment operator (**++**) is that the operand's value is increased by one unit of the appropriate type. Similarly, the effect of applying the postfix decrement operator (**—**) is that the operand's value is decreased by one unit of the appropriate type.

It is important to note that a postfix increment or decrement expression evaluates to the value of the expression **prior to** application of the respective operator. The increment or decrement operation occurs **after** the operand is evaluated. This issue arises only when the postfix increment or decrement operation occurs in the context of a larger expression.

When a postfix operator is applied to a function argument, the value of the argument is not guaranteed to be incremented or decremented before it is passed to the function. See section 1.9.17 in the C++ standard for more information.

Applying the postfix increment operator to a pointer to an array of objects of type **long** actually adds four to the internal representation of the pointer. This behavior causes the pointer, which previously referred to the *n*th element of the array, to refer to the (*n*+1)th element.

The operands to postfix increment and postfix decrement operators must be modifiable (not **const**) l-values of arithmetic or pointer type. The type of the result is the same as that of the *postfix-expression*, but it is no longer an l-value.

The operand of a postfix increment operator may also be of type **bool**, in which case the operand is evaluated and then set to **true**. The operand of a postfix decrement operator cannot be of type **bool**.

The following code illustrates the postfix increment operator:

```
// expre_Postfix_Increment_and_Decrement_Operators.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main() {
   int i = 10;
   cout << i++ << endl;
   cout << i << endl;
}
```

Postincrement and postdecrement operations on enumerated types are not supported:

```
enum Compass { North, South, East, West );
Compass myCompass;
for( myCompass = North; myCompass != West; myCompass++ ) // Error
```

See Also

Reference

Postfix Expressions
C++ Operators
Operator Precedence and Associativity
C Postfix Increment and Decrement Operators

Expressions with Unary Operators

Unary operators act on only one operand in an expression. The unary operators are:

- Indirection operator (*)
- Address-of operator (&)
- Unary plus operator (+)
- Unary negation operator (–)
- Logical negation operator (!)
- One's complement operator (~)
- Prefix increment operator (++)
- Prefix decrement operator (—)
- Cast Operator ()
- sizeof operator
- __uuidof operator
- __alignof operator
- new operator
- delete operator

These operators have right-to-left associativity. Unary expressions generally involve syntax that precedes a postfix or primary expression.

The following are the possible forms of unary expressions:

```
        postfix-expression
++unary-expression
--unary-expression
unary-operator cast-expression
sizeof .unary-expression
sizeof ( type-name )
allocation-expression
deallocation-expression
```

The *unary-operator* may be one of

```
        * & + - ! ~
```

Any postfix expression is considered a unary expression, and since any primary expression is considered a postfix expression, all primary expressions are considered unary expressions as well. For more information, see Postfix Expressions and Primary Expressions.

The allocation-expression refers to the **new** operator. The deallocation-expression refers to the **delete** operator. See the links above.

The *cast-expression* is a unary expression with an optional cast to alter the type. For more information see Cast Operator: ().

See Also
Reference
Types of Expressions

Indirection Operator: *

```
* cast-expression
```

Remarks

The unary indirection operator (*****) dereferences a pointer; that is, it converts a pointer value to an l-value. The operand of the indirection operator must be a pointer to a type. The result of the indirection expression is the type from which the pointer type is derived. The use of the ***** operator in this context is different from its meaning as a binary operator, which is multiplication.

If the operand points to a function, the result is a function designator. If it points to a storage location, the result is an l-value designating the storage location.

The indirection operator may be used cumulatively to dereference pointers to pointers. For example:

```
// expre_Indirection_Operator.cpp
// compile with: /EHsc
// Demonstrate indirection operator
#include <iostream>
using namespace std;
int main() {
   int n = 5;
   int *pn = &n;
   int **ppn = &pn;

   cout  << "Value of n:\n"
         << "direct value: " << n << endl
         << "indirect value: " << *pn << endl
         << "doubly indirect value: " << **ppn << endl
         << "address of n: " << pn << endl
         << "address of n via indirection: " << *ppn << endl;
}
```

If the pointer value is invalid, the result is undefined. The following list includes some of the most common conditions that invalidate a pointer value.

- The pointer is a null pointer.
- The pointer specifies the address of a local item that is not visible at the time of the reference.
- The pointer specifies an address that is inappropriately aligned for the type of the object pointed to.
- The pointer specifies an address not used by the executing program.

See Also

Reference

Expressions with Unary Operators
C++ Operators
Operator Precedence and Associativity
Address-of Operator: &

Concepts

Indirection and Address-of Operators

Address-of Operator: &

```
& cast-expression
```

Remarks

The unary address-of operator (**&**) takes the address of its operand. The operand of the address-of operator can be either a function designator or an l-value that designates an object that is not a bit field and is not declared with the **register** storage-class specifier.

The address-of operator can only be applied to variables with fundamental, structure, class, or union types that are declared at the file-scope level, or to subscripted array references. In these expressions, a constant expression that does not include the address-of operator can be added to or subtracted from the address-of expression.

When applied to functions or l-values, the result of the expression is a pointer type (an r-value) derived from the type of the operand. For example, if the operand is of type **char**, the result of the expression is of type pointer to **char**. The address-of operator, applied to **const** or **volatile** objects, evaluates to **const** *type* ***** or **volatile** *type* *****, where *type* is the type of the original object.

When the address-of operator is applied to a qualified name, the result depends on whether the *qualified-name* specifies a static member. If so, the result is a pointer to the type specified in the declaration of the member. If the member is not static, the result is a pointer to the member *name* of the class indicated by *qualified-class-name*. (See Primary Expressions for more about *qualified-class-name*.) The following code fragment shows how the result differs, depending on whether the member is static:

```
// expre_Address_Of_Operator.cpp
// C2440 expected
class PTM {
public:
           int   iValue;
    static float fValue;
};

int main() {
   int   PTM::*piValue = &PTM::iValue;  // OK: non-static
   float PTM::*pfValue = &PTM::fValue;  // C2440 error: static
   float *spfValue     = &PTM::fValue;  // OK
}
```

In this example, the expression `&PTM::fValue` yields type `float *` instead of type `float PTM::*` because `fValue` is a static member.

The address of an overloaded function can be taken only when it is clear which version of the function is being referenced. See Address of Overloaded Functions for information about how to obtain the address of a particular overloaded function.

Applying the address-of operator to a reference type gives the same result as applying the operator to the object to which the reference is bound. For example:

Example

```
// expre_Address_Of_Operator2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main() {
   double d;        // Define an object of type double.
   double& rd = d;  // Define a reference to the object.

   // Obtain and compare their addresses
   if( &d == &rd )
      cout << "&d equals &rd" << endl;
}
```

Output

```
&d equals &rd
```

The following example uses the address-of operator to pass a pointer argument to a function:

```
// expre_Address_Of_Operator3.cpp
// compile with: /EHsc
// Demonstrate address-of operator &

#include <iostream>
using namespace std;

// Function argument is pointer to type int
int square( int *n ) {
   return (*n) * (*n);
}

int main() {
   int mynum = 5;
   cout << square( &mynum ) << endl;   // pass address of int
}
```

Output

```
25
```

See Also

Reference

Expressions with Unary Operators
C++ Operators
Operator Precedence and Associativity
Reference Operator: &

Concepts

Indirection and Address-of Operators

Reference Operator: &

Holds the address of an object but behaves syntactically like an object.

```
type-id & cast-expression
```

Remarks

A reference declaration consists of an (optional) list of specifiers followed by a reference declarator. A reference must be initialized and cannot be changed.

Any object whose address can be converted to a given pointer type can also be converted to the analogous reference type. For example, any object whose address can be converted to type **char *** can also be converted to type **char &**. No constructors or class conversion functions are called to make a conversion to a reference type.

Do not confuse reference declarations with use of the address-of operator. When **&** *identifier* is preceded by a type, such as **int** or **char**, then *identifier* is declared as a reference to the type. When **&** *identifier* is not preceded by a type, the usage is that of the address-of operator.

Example

```
// expre_ReferenceOperator.cpp
// compile with: /EHsc
// Demonstrate reference operator
#include <iostream>
using namespace std;
struct Person {
    char   *Name;
    short Age;
};

int main() {
   Person Friend;             // Declare the object.
   Person& rFriend = Friend;  // Declare the reference.

   Friend.Name = "Bill";
   rFriend.Age = 40;

   cout << rFriend.Name << " is " << Friend.Age << endl;
}
```

See Also

Reference

C++ Operators
Operator Precedence and Associativity
References (C++)
Reference-Type Function Arguments
Reference-Type Function Returns
References to Pointers

Unary Plus Operator: +

```
+ cast-expression
```

Remarks

The result of the unary plus operator (**+**) is the value of its operand. The operand to the unary plus operator must be of an arithmetic type.

Integral promotion is performed on integral operands. The resultant type is the type to which the operand is promoted. Thus, the expression `+ch`, where `ch` is of type **char**, results in type **int**; the value is unmodified. See Integral Promotions for more information about how the promotion is done.

See Also

Reference

Expressions with Unary Operators
C++ Operators
Operator Precedence and Associativity

Unary Negation Operator: -

```
- cast-expression
```

Remarks

The unary negation operator (**–**) produces the negative of its operand. The operand to the unary negation operator must be an arithmetic type.

Integral promotion is performed on integral operands, and the resultant type is the type to which the operand is promoted. See Integral Promotions for more information on how the promotion is done.

Microsoft Specific

Unary negation of unsigned quantities is performed by subtracting the value of the operand from 2n, where *n* is the number of bits in an object of the given unsigned type. (Microsoft C++ runs on processors that utilize two's-complement arithmetic. On other processors, the algorithm for negation can differ.)

END Microsoft Specific

See Also

Reference

Expressions with Unary Operators

C++ Operators

Operator Precedence and Associativity

Logical Negation Operator: !

```
! cast-expression
```

Remarks

The logical negation operator (**!**) reverses the meaning of its operand. The operand must be of arithmetic or pointer type (or an expression that evaluates to arithmetic or pointer type). The operand is implicitly converted to type **bool**. The result is **true** if the converted operand is **false**; the result is **false** if the converted operand is **true**. The result is of type **bool**.

For an expression *e*, the unary expression **!***e* is equivalent to the expression **(***e* **==** 0), except where overloaded operators are involved.

Operator Keyword for !

The **not** operator is the text equivalent of **!**. There are two ways to access the **not** operator in your programs: include the header file `iso646.h`, or compile with the /Za (Disable language extensions) compiler option.

Example

```
// expre_Logical_NOT_Operator.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main() {
   int i = 0;
   if (!i)
      cout << "i is zero" << endl;
}
```

See Also

Reference

Expressions with Unary Operators

C++ Operators

Operator Precedence and Associativity

Concepts

Unary Arithmetic Operators

One's Complement Operator: ~

```
~ cast-expression
```

Remarks

The one's complement operator (**~**), sometimes called the "bitwise complement" operator, yields a bitwise one's complement of its operand. That is, every bit that is 1 in the operand is 0 in the result. Conversely, every bit that is 0 in the operand is 1 in the result. The operand to the one's complement operator must be an integral type.

Operator Keyword for ~

The **compl** operator is the text equivalent of **~**. There are two ways to access the **compl** operator in your programs: include the header file `iso646.h`, or compile with /Za.

Example

```
// expre_One_Complement_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

int main () {
   unsigned short y = 0xFFFF;
   cout << hex << y << endl;
   y = ~y;   // Take one's complement
   cout << hex << y << endl;
}
```

In this example, the new value assigned to `y` is the one's complement of the unsigned value 0xFFFF, or 0x0000.

Integral promotion is performed on integral operands, and the resultant type is the type to which the operand is promoted. See Integral Promotions for more information on how the promotion is done.

See Also

Reference

Expressions with Unary Operators
C++ Operators
Operator Precedence and Associativity

Concepts

Unary Arithmetic Operators

Prefix Increment and Decrement Operators: ++ and --

```
++ unary-expression
-- unary-expression
```

Remarks

The prefix increment operator (**++**) adds one to its operand; this incremented value is the result of the expression. The operand must be an l-value not of type **const**. The result is an l-value of the same type as the operand.

The prefix decrement operator (**—**) is analogous to the prefix increment operator, except that the operand is decremented by one and the result is this decremented value.

When a prefix operator is applied to a function argument, the value of the argument is not guaranteed to be incremented or decremented before it is passed to the function. See section 1.9.17 in the C++ standard for more information.

Both the prefix and postfix increment and decrement operators affect their operands. The key difference between them is when the increment or decrement takes place in the evaluation of an expression. (For more information, see Postfix Increment and Decrement Operators.) In the prefix form, the increment or decrement takes place before the value is used in expression evaluation, so the value of the expression is different from the value of the operand. In the postfix form, the increment or decrement takes place after the value is used in expression evaluation, so the value of the expression is the same as the value of the operand. For example, the following program prints "`++i = 6`":

```
// expre_Increment_and_Decrement_Operators.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

int main() {
   int i = 5;
   cout << "++i = " << ++i << endl;
}
```

An operand of integral or floating type is incremented or decremented by the integer value 1. The type of the result is the same as the operand type. An operand of pointer type is incremented or decremented by the size of the object it addresses. An incremented pointer points to the next object; a decremented pointer points to the previous object.

Because increment and decrement operators have side effects, using expressions with increment or decrement operators in a preprocessor macro can have undesirable results. Consider this example:

```
// expre_Increment_and_Decrement_Operators2.cpp
#define max(a,b) ((a)<(b))?(b):(a)

int main()
{
   int i = 0, j = 0, k;
   k = max( ++i, j );
}
```

The macro expands to:

```
k = ((++i)<(j))?(j):(++i);
```

If `i` is greater than or equal to `j` or less than `j` by 1, it will be incremented twice.

Note

C++ inline functions are preferable to macros in many cases because they eliminate side effects such as those described her e, and allow the language to perform more complete type checking.

See Also

Reference

Expressions with Unary Operators

C++ Operators

Operator Precedence and Associativity

Concepts

Prefix Increment and Decrement Operators

Cast Operator: ()

A type cast provides a method for explicit conversion of the type of an object in a specific situation.

```
unary-expression
( type-name ) cast-expression
```

Remarks

Any unary expression is considered a cast expression.

The compiler treats *cast-expression* as type *type-name* after a type cast has been made. Casts can be used to convert objects of any scalar type to or from any other scalar type. Explicit type casts are constrained by the same rules that determine the effects of implicit conversions. Additional restraints on casts may result from the actual sizes or representation of specific types.

Example

```
// expre_CastOperator.cpp
// compile with: /EHsc
// Demonstrate cast operator
#include <iostream>

using namespace std;

int main()
{
    double x = 3.1;
    int i;
    cout << "x = " << x << endl;
    i = (int)x;   // assign i the integer part of x
    cout << "i = " << i << endl;
}
```

```
// expre_CastOperator2.cpp
// The following sample shows how to define and use a cast operator.
#include <string.h>
#include <stdio.h>

class CountedAnsiString
{
public:
    // Assume source is not null terminated
    CountedAnsiString(const char *pStr, size_t nSize) :
                      m_nSize(nSize)
    {
        m_pStr = new char[sizeOfBuffer];

        strncpy_s(m_pStr, sizeOfBuffer, pStr, m_nSize);
        memset(&m_pStr[m_nSize], '!', 9); // for demonstration purposes.
    }

    // Various string-like methods...

    const char *GetRawBytes() const
    {
        return(m_pStr);
    }

    //
    // operator to cast to a const char *
    //
    operator const char *()
    {
```

```
        m_pStr[m_nSize] = '\0';
        return(m_pStr);
    }

    enum
    {
        sizeOfBuffer = 20
    } size;

private:
    char *m_pStr;
    const size_t m_nSize;
};

int main()
{
    const char *kStr = "Excitinggg";
    CountedAnsiString myStr(kStr, 8);

    const char *pRaw = myStr.GetRawBytes();
    printf_s("RawBytes truncated to 10 chars:   %.10s\n", pRaw);

    const char *pCast = myStr; // or (const char *)myStr;
    printf_s("Casted Bytes:   %s\n", pCast);

    puts("Note that the cast changed the raw internal string");
    printf_s("Raw Bytes after cast:   %s\n", pRaw);
}
```

Output

```
RawBytes truncated to 10 chars:   Exciting!!
Casted Bytes:   Exciting
Note that the cast changed the raw internal string
Raw Bytes after cast:   Exciting
```

See Also

Reference

Expressions with Unary Operators
C++ Operators
Operator Precedence and Associativity
Explicit Type Conversion Operator: ()
Casting Operators

Concepts

Cast Operators

sizeof Operator

Yields the size of its operand with respect to the size of type **char**.

```
sizeof unary-expression
sizeof ( type-name )
```

Remarks

The result of the **sizeof** operator is of type **size_t**, an integral type defined in the include file STDDEF.H. This operator allows you to avoid specifying machine-dependent data sizes in your programs.

The operand to **sizeof** can be one of the following:

- A type name. To use **sizeof** with a type name, the name must be enclosed in parentheses.
- An expression. When used with an expression, **sizeof** can be specified with or without the parentheses. The expression is not evaluated.

When the **sizeof** operator is applied to an object of type **char**, it yields 1. When the **sizeof** operator is applied to an array, it yields the total number of bytes in that array, not the size of the pointer represented by the array identifier. To obtain the size of the pointer represented by the array identifier, pass it as a parameter to a function that uses **sizeof**. For example:

Example

```
// expre_sizeof_Operator.cpp
// compile with: /EHsc
#include <iostream>

size_t getPtrSize( char *ptr )
{
   return sizeof( ptr );
}

using namespace std;
int main()
{
   char szHello[] = "Hello, world!";

   cout  << "The size of a char is: "
         << sizeof( char )
         << "\nThe length of " << szHello << " is: "
         << sizeof szHello
         << "\nThe size of the pointer is "
         << getPtrSize( szHello ) << endl;
}
```

Sample Output

```
The size of a char is: 1
The length of Hello, world! is: 14
The size of the pointer is 4
```

When the **sizeof** operator is applied to a **class**, **struct**, or **union** type, the result is the number of bytes in an object of that type, plus any padding added to align members on word boundaries. The result does not necessarily correspond to the size calculated by adding the storage requirements of the individual members. The /Zp compiler option and the pack pragma affect alignment boundaries for members.

The **sizeof** operator never yields 0, even for an empty class.

The **sizeof** operator cannot be used with the following operands:

- Functions. (However, **sizeof** can be applied to pointers to functions.)

- Bit fields.
- Undefined classes.
- The type **void**.
- Dynamically allocated arrays.
- External arrays.
- Incomplete types.
- Parenthesized names of incomplete types.

When the **sizeof** operator is applied to a reference, the result is the same as if **sizeof** had been applied to the object itself.

If an unsized array is the last element of a structure, the **sizeof** operator returns the size of the structure without the array.

The **sizeof** operator is often used to calculate the number of elements in an array using an expression of the form:

```
sizeof array / sizeof array[0]
```

See Also

Reference

Expressions with Unary Operators

C++ Keywords

__uuidof Operator

Microsoft Specific

Retrieves the GUID attached to the expression.

```
__uuidof (
      expression
)
```

Remarks

The *expression* can be a type name, pointer, reference, or array of that type, a template specialized on these types, or a variable of these types. The argument is valid as long as the compiler can use it to find the attached GUID.

A special case of this intrinsic is when either **0** or **NULL** is supplied as the argument. In this case, **__uuidof** will return a GUID made up of zeros.

Use this keyword to extract the GUID attached to:

- An object by the uuid extended attribute.
- A library block created with the module attribute.

Note
In a debug build, **__uuidof** always initializes an object dynamically (at runtime). In a release build, **__uuidof** can statically (at compile time) initialize an object.

Example

The following code (compiled with ole32.lib) will display the uuid of a library block created with the module attribute:

```
// expre_uuidof.cpp
// compile with: ole32.lib
#include "stdio.h"
#include "windows.h"

[emitidl];
[module(name="MyLib")];
[export]
struct stuff {
   int i;
};

int main() {
   LPOLESTR lpolestr;
   StringFromCLSID(__uuidof(MyLib), &lpolestr);
   wprintf_s(L"%s", lpolestr);
   CoTaskMemFree(lpolestr);
}
```

Comments

In cases where the library name is no longer in scope, you can use __LIBID_ instead of **__uuidof**. For example:

```
StringFromCLSID(__LIBID_, &lpolestr);
```

END Microsoft Specific

See Also

Reference

Expressions with Unary Operators

__alignof Operator

Microsoft Specific

Returns a value, of type **size_t**, that is the alignment requirement of the type.

```
__alignof(
      type
)
```

Remarks

For example:

Expression	Value
__alignof(char)	1
__alignof(short)	2
__alignof(int)	4
__alignof(__int64)	8
__alignof(float)	4
__alignof(double)	8
__alignof(char*)	4

The **__alignof** value is the same as the value for **sizeof** for basic types. Consider, however, this example:

```
typedef struct { int a; double b; } S;
// __alignof(S) == 8
```

In this case, the **__alignof** value is the alignment requirement of the largest element in the structure.

Similarly, for

```
typedef __declspec(align(32)) struct { int a; } S;
```

`__alignof(S)` is equal to `32`.

One use for **__alignof** would be as a parameter to one of your own memory-allocation routines. For example, given the following defined structure `S`, you could call a memory-allocation routine named `aligned_malloc` to allocate memory on a particular alignment boundary.

```
typedef __declspec(align(32)) struct { int a; double b; } S;
int n = 50; // array size
S* p = (S*)aligned_malloc(n * sizeof(S), __alignof(S));
```

For more information on modifying alignment, see:

- pack
- align
- __unaligned

- /Zp (Struct Member Alignment)
- Structure Alignment Examples (x64 specific)

END Microsoft Specific

See Also

Reference

Expressions with Unary Operators

C++ Keywords

new Operator (C++)

Allocates memory for an object or array of objects of *type-name* from the free store and returns a suitably typed, nonzero pointer to the object.

```
[::] new [placement] new-type-name [new-initializer]
[::] new [placement] ( type-name ) [new-initializer]
```

Remarks

If unsuccessful, **new** returns zero or throws an exception; see The new and delete Operators for more information. You can change this default behavior by writing a custom exception-handling routine and calling the _set_new_handler run-time library function with your function name as its argument.

For information on how to create an object on the managed heap, see gcnew.

When **new** is used to allocate memory for a C++ class object, the object's constructor is called after the memory is allocated.

Use the delete operator to deallocate the memory allocated with the **new** operator.

The following example allocates and then frees a two-dimensional array of characters of size *dim* by 10. When allocating a multidimensional array, all dimensions except the first must be constant expressions that evaluate to positive values; the leftmost array dimension can be any expression that evaluates to a positive value. When allocating an array using the **new** operator, the first dimension can be zero — the **new** operator returns a unique pointer.

```
char (*pchar)[10] = new char[dim][10];
delete [] pchar;
```

The *type-name* cannot contain **const**, **volatile**, class declarations, or enumeration declarations. Therefore, the following expression is illegal:

```
volatile char *vch = new volatile char[20];
```

The **new** operator does not allocate reference types because they are not objects.

The **new** operator cannot be used to allocate a function, but it can be used to allocate pointers to functions. The following example allocates and then frees an array of seven pointers to functions that return integers.

```
int (**p) () = new (int (*[7]) ());
delete *p;
```

If you use the operator **new** without any extra arguments, and compile with the /GX, /EHa, or /EHs option, the compiler will generate code to call operator **delete** if the constructor throws an exception.

The following list describes the grammar elements of **new**:

placement

Provides a way of passing additional arguments if you overload **new**.

type-name

Specifies type to be allocated; it can be either a built-in or user-defined type. If the type specification is complicated, it can be surrounded by parentheses to force the order of binding.

initializer

Provides a value for the initialized object. Initializers cannot be specified for arrays. The **new** operator will create arrays of objects only if the class has a default constructor.

Example

The following code example allocates a character array and an object of class `CName` and then frees them.

```
// expre_new_Operator.cpp
// compile with: /EHsc
#include <string.h>

class CName {
public:
   enum {
      sizeOfBuffer = 256
   };

   char m_szFirst[sizeOfBuffer];
   char m_szLast[sizeOfBuffer];

public:
   void SetName(char* pszFirst, char* pszLast) {
     strcpy_s(m_szFirst, sizeOfBuffer, pszFirst);
     strcpy_s(m_szLast, sizeOfBuffer, pszLast);
   }

};

int main() {
   // Allocate memory for the array
   char* pCharArray = new char[CName::sizeOfBuffer];
   strcpy_s(pCharArray, CName::sizeOfBuffer, "Array of characters");

   // Deallocate memory for the array
   delete [] pCharArray;
   pCharArray = NULL;

   // Allocate memory for the object
   CName* pName = new CName;
   pName->SetName("Firstname", "Lastname");

   // Deallocate memory for the object
   delete pName;
   pName = NULL;
}
```

If you use the placement new form of the **new** operator, the form with arguments in addition to the size of the allocation, the compiler does not support a placement form of the **delete** operator if the constructor throws an exception. For example:

```
// expre_new_Operator2.cpp
// C2660 expected
class A {
public:
   A(int) { throw "Fail!"; }
};
void F(void) {
   try {
      // heap memory pointed to by pa1 will be deallocated
      // by calling ::operator delete(void*).
      A* pa1 = new A(10);
   } catch (...) {
   }
   try {
      // This will call ::operator new(size_t, char*, int).
      // When A::A(int) does a throw, we should call
      // ::operator delete(void*, char*, int) to deallocate
      // the memory pointed to by pa2.  Since
      // ::operator delete(void*, char*, int) has not been implemented,
      // memory will be leaked when the deallocation cannot occur.

      A* pa2 = new(__FILE__, __LINE__) A(20);
   } catch (...) {
   }
}
```

```
int main() {
   A a;
}
```

See Also

Reference

Expressions with Unary Operators
C++ Keywords
Lifetime of Objects Allocated with new
Initializing Objects Allocated with new
How new Works
The operator new Function

Lifetime of Objects Allocated with new

Objects allocated with the **new** operator are not destroyed when the scope in which they are defined is exited. Because the **new** operator returns a pointer to the objects it allocates, the program must define a pointer with suitable scope to access those objects. For example:

```
// expre_Lifetime_of_Objects_Allocated_with_new.cpp
// C2541 expected
int main()
{
    // Use new operator to allocate an array of 20 characters.
    char *AnArray = new char[20];

    for( int i = 0; i < 20; ++i )
    {
        // On the first iteration of the loop, allocate
        //  another array of 20 characters.
        if( i == 0 )
        {
            char *AnotherArray = new char[20];
        }
    }

    delete [] AnotherArray; // Error: pointer out of scope.
    delete [] AnArray;      // OK: pointer still in scope.
}
```

Once the pointer `AnotherArray` goes out of scope in the example, the object can no longer be deleted.

See Also

Reference

new Operator (C++)

Initializing Objects Allocated with new

An optional *initializer* field is included in the grammar for the **new** operator. This allows new objects to be initialized with user-defined constructors. For more information about how initialization is done, see Initializers. The following example illustrates how to use an initialization expression with the **new** operator:

```
// expre_Initializing_Objects_Allocated_with_new.cpp
class Acct
{
public:
    // Define default constructor and a constructor that accepts
    //  an initial balance.
    Acct() { balance = 0.0; }
    Acct( double init_balance ) { balance = init_balance; }
private:
    double balance;
};

int main()
{
    Acct *CheckingAcct = new Acct;
    Acct *SavingsAcct = new Acct ( 34.98 );
    double *HowMuch = new double ( 43.0 );
    // ...
}
```

In this example, the object `CheckingAcct` is allocated using the **new** operator, but no default initialization is specified. Therefore, the default constructor for the class, `Acct()`, is called. Then the object `SavingsAcct` is allocated the same way, except that it is explicitly initialized to 34.98. Because 34.98 is of type **double**, the constructor that takes an argument of that type is called to handle the initialization. Finally, the nonclass type `HowMuch` is initialized to 43.0.

If an object is of a class type and that class has constructors (as in the preceding example), the object can be initialized by the **new** operator only if one of these conditions is met:

- The arguments provided in the initializer agree with those of a constructor.
- The class has a default constructor (a constructor that can be called with no arguments).

Access control and ambiguity control are performed on **operator new** and on the constructors according to the rules set forth in Ambiguity and Initialization Using Special Member Functions.

No explicit per-element initialization can be done when allocating arrays using the **new** operator; only the default constructor, if present, is called. See Default Arguments for more information.

If the memory allocation fails (**operator new** returns a value of 0), no initialization is performed. This protects against attempts to initialize data that does not exist.

As with function calls, the order in which initialized expressions are evaluated is not defined. Furthermore, you should not rely on these expressions being completely evaluated before the memory allocation is performed. If the memory allocation fails and the **new** operator returns zero, some expressions in the initializer may not be completely evaluated.

See Also

Reference

new Operator (C++)

How new Works

The *allocation-expression* — the expression containing the **new** operator — does three things:

- Locates and reserves storage for the object or objects to be allocated. When this stage is complete, the correct amount of storage is allocated, but it is not yet an object.
- Initializes the object(s). Once initialization is complete, enough information is present for the allocated storage to be an object.
- Returns a pointer to the object(s) of a pointer type derived from *new-type-name* or *type-name*. The program uses this pointer to access the newly allocated object.

The **new** operator invokes the function **operator new**. For arrays of any type, and for objects that are not of **class**, **struct**, or **union** types, a global function, **::operator new**, is called to allocate storage. Class-type objects can define their own **operator new** static member function on a per-class basis.

When the compiler encounters the **new** operator to allocate an object of type *type*, it issues a call to *type***::operator new(sizeof(** *type* **))** or, if no user-defined **operator new** is defined, **::operator new(sizeof(** *type* **))**. Therefore, the **new** operator can allocate the correct amount of memory for the object.

Note
The argument to **operator new** is of type **size_t**. This type is defined in DIRECT.H, MALLOC.H, MEMORY.H, SEARCH.H, STDDEF.H, STDIO.H, STDLIB.H, STRING.H, and TIME.H.

An option in the grammar allows specification of *placement* (see the Grammar for new Operator). The *placement* parameter can be used only for user-defined implementations of **operator new**; it allows extra information to be passed to **operator new**. An expression with a *placement* field such as `T *TObject = new ( 0x0040 ) T;` is translated to `T *TObject = T::operator new( sizeof( T ), 0x0040 );` if class T has member operator new, otherwise to `T *TObject = ::operator new( sizeof( T ), 0x0040 );`.

The original intention of the *placement* field was to allow hardware-dependent objects to be allocated at user-specified addresses.

Note
Although the preceding example shows only one argument in the *placement* field, there is no restriction on how many extra arguments can be passed to **operator new** this way.

Even when **operator new** has been defined for a class type, the global operator can be used by using the form of this example:

```
T *TObject =::new TObject;
```

The scope-resolution operator (**::**) forces use of the global **new** operator.

See Also

Reference

new Operator (C++)

delete Operator (C++)

Deallocates a block of memory.

```
[::] delete cast-expression
[::] delete [ ] cast-expression
```

Remarks

The *cast-expression* argument must be a pointer to a block of memory previously allocated for an object created with the new operator. The **delete** operator has a result of type **void** and therefore does not return a value. For example:

```
CDialog* MyDialog = new CDialog;
// use MyDialog
delete MyDialog;
```

Using **delete** on a pointer to an object not allocated with **new** gives unpredictable results. You can, however, use **delete** on a pointer with the value 0. This provision means that, when **new** returns 0 on failure, deleting the result of a failed **new** operation is harmless. See The new and delete Operators for more information.

The **new** and **delete** operators can also be used for built-in types, including arrays. If *pointer* refers to an array, place empty brackets before *pointer*:

```
int* set = new int[100];
//use set[]
delete [] set;
```

Using the **delete** operator on an object deallocates its memory. A program that dereferences a pointer after the object is deleted can have unpredictable results or crash.

When **delete** is used to deallocate memory for a C++ class object, the object's destructor is called before the object's memory is deallocated (if the object has a destructor).

If the operand to the **delete** operator is a modifiable l-value, its value is undefined after the object is deleted.

Example

For examples of using **delete**, see new operator.

See Also

Reference

Expressions with Unary Operators
C++ Keywords
How delete Works
Using delete
The new and delete Operators
The operator delete Function

How delete Works

The delete operator invokes the function operator delete.

For objects not of class type (class, struct, or union), the global delete operator is invoked. For objects of class type, the name of the deallocation function is resolved in global scope if the delete expression begins with the unary scope resolution operator (::). Otherwise, the delete operator invokes the destructor for an object prior to deallocating memory (if the pointer is not null). The delete operator can be defined on a per-class basis; if there is no such definition for a given class, the global operator delete is invoked. If the delete expression is used to deallocate a class object whose static type has a virtual destructor, the deallocation function is resolved through the virtual destructor of the dynamic type of the object.

See Also

Reference

Expressions with Unary Operators

Using delete

There are two syntactic variants for the delete operator: one for single objects and the other for arrays of objects. The following code fragment shows how these differ:

```
// expre_Using_delete.cpp
struct UDType
{
};

int main()
{
   // Allocate a user-defined object, UDObject, and an object
   //  of type double on the free store using the
   //  new operator.
   UDType *UDObject = new UDType;
   double *dObject = new double;
   // Delete the two objects.
   delete UDObject;
   delete dObject;
   // Allocate an array of user-defined objects on the
   // free store using the new operator.
   UDType (*UDArr)[7] = new UDType[5][7];
   // Use the array syntax to delete the array of objects.
   delete [] UDArr;
}
```

The following two cases produce undefined results: using the array form of delete (delete [ ]) on an object and using the nonarray form of delete on an array.

See Also

Reference

Expressions with Unary Operators

Expressions with Binary Operators

Binary operators act on two operands in an expression. The binary operators are:

- Multiplicative operators
  - Multiplication (*)
  - Division (/)
  - Modulus (%)
- Additive operators
  - Addition (+)
  - Subtraction (–)
- Shift operators
  - Right shift (>>)
  - Left shift (<<)
- Relational and equality operators
  - Less than (<)
  - Greater than (>)
  - Less than or equal to (<=)
  - Greater than or equal to (>=)
  - Equal to (==)
  - Not equal to (!=)
- Bitwise operators
  - Bitwise AND (&)
  - Bitwise exclusive OR (^)
  - Bitwise inclusive OR (|)
- Logical operators
  - Logical AND (&&)
  - Logical OR (||)
- Assignment operators
  - Assignment (=)
  - Addition assignment (+=)
  - Subtraction assignment (–=)
  - Multiplication assignment (*=)
  - Division assignment (/=)
  - Modulus assignment (%=)
  - Left shift assignment (<<=)
  - Right shift assignment (>>=)
  - Bitwise AND assignment (&=)
  - Bitwise exclusive OR assignment (^=)

    - Bitwise inclusive OR assignment (|=)

- Comma Operator (,)

See Also

Reference

Types of Expressions

Multiplicative Operators: *, /, and %

```
expression * expression
expression / expression
expression % expression
```

Remarks

The multiplicative operators are:

- Multiplication (*****)
- Division (**/**)
- Modulus (remainder from division) (**%**)

These binary operators have left-to-right associativity.

The multiplicative operators take operands of arithmetic types. The modulus operator (**%**) has a stricter requirement in that its operands must be of integral type. (To get the remainder of a floating-point division, use the run-time function, fmod.) The conversions covered in Arithmetic Conversions are applied to the operands, and the result is of the converted type.

The multiplication operator yields the result of multiplying the first operand by the second.

The division operator yields the result of dividing the first operand by the second.

The modulus operator yields the remainder given by the following expression, where *e1* is the first operand and *e2* is the second: *e1* – (*e1* / *e2*) * *e2*, where both operands are of integral types.

Division by 0 in either a division or a modulus expression is undefined and causes a run-time error. Therefore, the following expressions generate undefined, erroneous results:

```
i % 0
f / 0.0
```

If both operands to a multiplication, division, or modulus expression have the same sign, the result is positive. Otherwise, the result is negative. The result of a modulus operation's sign is implementation-defined.

Note

Since the conversions performed by the multiplicative operators do not provide for overflow or underflow conditions, information may be lost if the result of a multiplicative operation cannot be represented in the type of the operands after conversion.

Microsoft Specific

In Microsoft C++, the result of a modulus expression is always the same as the sign of the first operand.

END Microsoft Specific

If the computed division of two integers is inexact and only one operand is negative, the result is the largest integer (in magnitude, disregarding the sign) that is less than the exact value the division operation would yield. For example, the computed value of –11 / 3 is –3.666666666. The result of that integral division is –3.

The relationship between the multiplicative operators is given by the identity (*e1* / *e2*) * *e2* + *e1* % *e2* == *e1*.

Example

The following program demonstrates the multiplicative operators. Note that either operand of `10 / 3` must be explicitly cast to type `float` to avoid truncation so that both operands are of type `float` before division.

```
// expre_Multiplicative_Operators.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
```

```
int main() {
   int x = 3, y = 6, z = 10;
   cout  << "3 * 6 is " << x * y << endl
         << "6 / 3 is " << y / x << endl
         << "10 % 3 is " << z % x << endl
         << "10 / 3 is " << (float) z / x << endl;
}
```

See Also

Reference

Expressions with Binary Operators

C++ Operators

Operator Precedence and Associativity

C Multiplicative Operators

Additive Operators: + and -

```
expression + expression
expression – expression
```

Remarks

The additive operators are:

- Addition (**+**)
- Subtraction (**–**)

These binary operators have left-to-right associativity.

The additive operators take operands of arithmetic or pointer types. The result of the addition (**+**) operator is the sum of the operands. The result of the subtraction (**–**) operator is the difference between the operands. If one or both of the operands are pointers, they must be pointers to objects, not to functions. If both operands are pointers, the results are not meaningful unless both are pointers to objects in the same array.

Additive operators take operands of *arithmetic*, *integral*, and *scalar* types. These are defined in the following table.

Types Used with Additive Operators

Type	Meaning
arithmetic	Integral and floating types are collectively called "arithmetic" types.
integral	Types char and int of all sizes (long, short) and enumerations are "integral" types.
scalar	Scalar operands are operands of either arithmetic or pointer type.

The legal combinations for these operators are:

arithmetic + *arithmetic*

scalar + *integral*

integral + *scalar*

arithmetic – *arithmetic*

scalar – *scalar*

Note that addition and subtraction are not equivalent operations.

If both operands are of arithmetic type, the conversions covered in Arithmetic Conversions are applied to the operands, and the result is of the converted type.

Example

```
// expre_Additive_Operators.cpp
// compile with: /EHsc
#include <iostream>
#define SIZE 5
using namespace std;
int main() {
   int i = 5, j = 10;
   int n[SIZE] = { 0, 1, 2, 3, 4 };
   cout  << "5 + 10 = " << i + j << endl
         << "5 - 10 = " << i - j << endl;

   // use pointer arithmetic on array

   cout << "n[3] = " << *( n + 3 ) << endl;
}
```

See Also

Reference

Expressions with Binary Operators
C++ Operators
Operator Precedence and Associativity
Addition of Pointer Types
Subtraction of Pointer Types
C Additive Operators

Addition of Pointer Types

If one of the operands in an addition operation is a pointer to an array of objects, the other must be of integral type. The result is a pointer that is of the same type as the original pointer and that points to another array element. The following code fragment illustrates this concept:

```
short IntArray[10]; // Objects of type short occupy 2 bytes
short *pIntArray = IntArray;

for( int i = 0; i < 10; ++i )
{
    *pIntArray = i;
    cout << *pIntArray << "\n";
    pIntArray = pIntArray + 1;
}
```

Although the integral value 1 is added to `pIntArray`, it does not mean "add 1 to the address"; rather it means "adjust the pointer to point to the next object in the array" that happens to be 2 bytes (or `sizeof( int )`) away.

Note

Code of the form `pIntArray = pIntArray + 1` is rarely found in C++ programs; to perform an increment, these forms are preferable: `pIntArray++` or `pIntArray += 1`.

See Also

Reference

Expressions with Binary Operators

Subtraction of Pointer Types

If both operands are pointers, the result of subtraction is the difference (in array elements) between the operands. The subtraction expression yields a signed integral result of type ptrdiff_t (defined in the standard include file STDDEF.H).

One of the operands can be of integral type, as long as it is the second operand. The result of the subtraction is of the same type as the original pointer. The value of the subtraction is a pointer to the ($n - i$)th array element, where n is the element pointed to by the original pointer and i is the integral value of the second operand.

See Also

Reference

Expressions with Binary Operators

Shift Operators: >> and <<

```
expression << expression
expression >> expression
```

Remarks

The bitwise shift operators are:

- Right shift (**>>**)
- Left shift (**<<**)

These binary operators have left-to-right associativity.

Both operands of the shift operators must be of integral types. Integral promotions are performed according to the rules described in Integral Promotions. The type of the result is the same as the type of the left operand. The value of a right-shift expression *e1* >> *e2* is *e1* / 2e2, and the value of a left-shift expression *e1* << *e2* is *e1* * 2e2.

The results are undefined if the right operand of a shift expression is negative or if the right operand is greater than or equal to the number of bits in the (promoted) left operand.

The left shift operator causes the bit pattern in the first operand to be shifted left the number of bits specified by the second operand. Bits vacated by the shift operation are zero-filled. This is a logical shift, as opposed to a shift-and-rotate operation.

The right shift operator causes the bit pattern in the first operand to be shifted right the number of bits specified by the second operand. Bits vacated by the shift operation are zero-filled for unsigned quantities. For signed quantities, the sign bit is propagated into the vacated bit positions. The shift is a logical shift if the left operand is an unsigned quantity; otherwise, it is an arithmetic shift.

Microsoft Specific

The result of a right shift of a signed negative quantity is implementation dependent. Although Microsoft C++ propagates the most-significant bit to fill vacated bit positions, there is no guarantee that other implementations will do likewise.

END Microsoft Specific

Example

```
// expre_Shift_Operators.cpp
// compile with: /EHsc
// Demonstrate shift operators
#include <iostream>
using namespace std;

int main() {
   cout   << "5 times 2 is " << (5 << 1) << endl
          << "20 divided by 4 is " << (20 >> 2) << endl;
}
```

See Also

Reference

Expressions with Binary Operators
C++ Operators
Operator Precedence and Associativity

Concepts

Bitwise Shift Operators

C++ Relational and Equality Operators

The relational and equality operators determine equality, inequality, or relative values of their operands. The relational operators are shown in the following table.

Relational and Equality Operators

Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

See Also

Reference

Expressions with Binary Operators
C++ Operators
Operator Precedence and Associativity

Relational Operators: <, >, <=, and >=

```
expression < expression
expression > expression
expression <= expression
expression >= expression
```

Remarks

The binary relational operators determine the following relationships:

- Less than (**<**)
- Greater than (**>**)
- Less than or equal to (**<=**)
- Greater than or equal to (**>=**)

The relational operators have left-to-right associativity. Both operands of relational operators must be of arithmetic or pointer type. They yield values of type **bool**. The value returned is **false** (0) if the relationship in the expression is false; otherwise, the value returned is **true** (1).

Example

```
// expre_Relational_Operators.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

int main() {
   cout  << "The true expression 3 > 2 yields: "
         << (3 > 2) << endl
         << "The false expression 20 < 10 yields: "
         << (20 < 10) << endl;
}
```

The expressions in the preceding example must be enclosed in parentheses because the stream insertion operator (**<<**) has higher precedence than the relational operators. Therefore, the first expression without the parentheses would be evaluated as:

```
(cout << "The true expression 3 > 2 yields: " << 3) < (2 << "\n");
```

The usual arithmetic conversions covered in Arithmetic Conversions are applied to operands of arithmetic types.

See Also

Reference

Expressions with Binary Operators
C++ Operators
Operator Precedence and Associativity
Comparing Pointers Using Relational Operators
C Relational and Equality Operators

Comparing Pointers Using Relational Operators

When two pointers to objects of the same type are compared, the result is determined by the location of the objects pointed to in the program's address space. Pointers can also be compared to a constant expression that evaluates to 0 or to a pointer of type void *. If a pointer comparison is made against a pointer of type void *, the other pointer is implicitly converted to type void *. Then the comparison is made.

Two pointers of different types cannot be compared unless:

- One type is a class type derived from the other type.
- At least one of the pointers is explicitly converted (cast) to type void *. (The other pointer is implicitly converted to type void * for the conversion.)

Two pointers of the same type that point to the same object are guaranteed to compare equal. If two pointers to nonstatic members of an object are compared, the following rules apply:

- If the class type is not a union, and if the two members are not separated by an *access-specifier*, such as public, protected, or private, the pointer to the member declared last will compare greater than the pointer to the member declared earlier. (For information on *access-specifier*, see the Syntax section in Access Specifiers.)
- If the two members are separated by an *access-specifier*, the results are undefined.
- If the class type is a union, pointers to different data members in that union compare equal.

If two pointers point to elements of the same array or to the element one beyond the end of the array, the pointer to the object with the higher subscript compares higher. Comparison of pointers is guaranteed valid only when the pointers refer to objects in the same array or to the location one past the end of the array.

See Also
Reference
Expressions with Binary Operators

Equality Operators: == and !=

```
expression == expression
expression != expression
```

Remarks

The binary equality operators compare their operands for strict equality or inequality.

The equality operators, equal to (**==**) and not equal to (**!=**), have lower precedence than the relational operators, but they behave similarly. The result type for these operators is **bool**.

The equal-to operator (**==**) returns **true** (1) if both operands have the same value; otherwise, it returns **false** (0). The not-equal-to operator (**!=**) returns **true** if the operands do not have the same value; otherwise, it returns **false**.

Operator Keyword for !=

The **not_eq** operator is the text equivalent of **!=**. There are two ways to access the **not_eq** operator in your programs: include the header file `iso646.h`, or compile with the /Za (Disable language extensions) compiler option.

Example

```
// expre_Equality_Operators.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

int main() {
   cout  << boolalpha
         << "The true expression 3 != 2 yields: "
         << (3 != 2) << endl
         << "The false expression 20 == 10 yields: "
         << (20 == 10) << endl;
}
```

Equality operators can compare pointers to members of the same type. In such a comparison, pointer-to-member conversions, as discussed in Pointer-to-Member Conversions are performed. Pointers to members can also be compared to a constant expression that evaluates to 0.

See Also

Reference

Expressions with Binary Operators
C++ Operators
Operator Precedence and Associativity
C Relational and Equality Operators

C++ Bitwise Operators

The bitwise operators are:

- Bitwise AND (&)
- Bitwise exclusive OR (^)
- Bitwise inclusive OR (|)

These operators return bitwise combinations of their operands.

See Also

Reference

Expressions with Binary Operators

Bitwise AND Operator: &

```
expression & expression
```

Remarks

The expressions may be other and-expressions, or (subject to the type restrictions mentioned below) equality expressions, relational expressions, additive expressions, multiplicative expressions, pointer to member expressions, cast expressions, unary expressions, postfix expressions, or primary expressions.

The bitwise AND operator (**&**) compares each bit of the first operand to the corresponding bit of the second operand. If both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

Both operands to the bitwise AND operator must be of integral types. The usual arithmetic conversions covered in Arithmetic Conversions, are applied to the operands.

Operator Keyword for &

The **bitand** operator is the text equivalent of **&**. There are two ways to access the **bitand** operator in your programs: include the header file `iso646.h`, or compile with the /Za (Disable language extensions) compiler option.

Example

```
// expre_Bitwise_AND_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise AND
#include <iostream>
using namespace std;
int main() {
   unsigned short a = 0xFFFF;      // pattern 1111 ...
   unsigned short b = 0xAAAA;      // pattern 1010 ...

   cout  << hex << ( a & b ) << endl;   // prints "aaaa", pattern 1010 ...
}
```

See Also

Reference

C++ Bitwise Operators
C++ Operators
Operator Precedence and Associativity
C Bitwise Operators

Bitwise Exclusive OR Operator: ^

```
expression ^ expression
```

Remarks

The bitwise exclusive OR operator (**^**) compares each bit of its first operand to the corresponding bit of its second operand. If one bit is 0 and the other bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

Both operands to the bitwise exclusive OR operator must be of integral types. The usual arithmetic conversions covered in Arithmetic Conversions are applied to the operands.

Operator Keyword for ^

The **xor** operator is the text equivalent of **^**. There are two ways to access the **xor** operator in your programs: include the header file `iso646.h`, or compile with the /Za (Disable language extensions) compiler option.

Example

```
// expre_Bitwise_Exclusive_OR_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise exclusive OR
#include <iostream>
using namespace std;
int main() {
   unsigned short a = 0x5555;      // pattern 0101 ...
   unsigned short b = 0xFFFF;      // pattern 1111 ...

   cout  << hex << ( a ^ b ) << endl;   // prints "aaaa" pattern 1010 ...
}
```

See Also

Reference

C++ Bitwise Operators
C++ Operators
Operator Precedence and Associativity
C Bitwise Operators

Bitwise Inclusive OR Operator: |

```
expression | expression
```

Remarks

The bitwise inclusive OR operator (**|**) compares each bit of its first operand to the corresponding bit of its second operand. If either bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

Both operands to the bitwise inclusive OR operator must be of integral types. The usual arithmetic conversions covered in Arithmetic Conversions are applied to the operands.

Operator Keyword for |

The **bitor** operator is the text equivalent of **|**. There are two ways to access the **bitor** operator in your programs: include the header file `iso646.h`, or compile with the /Za (Disable language extensions) compiler option.

Example

```
// expre_Bitwise_Inclusive_OR_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise inclusive OR
#include <iostream>
using namespace std;

int main() {
   unsigned short a = 0x5555;      // pattern 0101 ...
   unsigned short b = 0xAAAA;      // pattern 1010 ...

   cout  << hex << ( a | b ) << endl;   // prints "ffff" pattern 1111 ...
}
```

See Also

Reference

C++ Bitwise Operators
C++ Operators
Operator Precedence and Associativity
C Bitwise Operators

C++ Logical Operators

The logical operators, logical AND (&&) and logical OR (||), are used to combine multiple conditions formed using relational or equality expressions.

See Also

Reference

Expressions with Binary Operators

Logical AND Operator: &&

```
expression && expression
```

Remarks

The logical AND operator (**&&**) returns the boolean value **true** if both operands are **true** and returns **false** otherwise. The operands are implicitly converted to type **bool** prior to evaluation, and the result is of type **bool**. Logical AND has left-to-right associativity.

The operands to the logical AND operator need not be of the same type, but they must be of integral or pointer type. The operands are commonly relational or equality expressions.

The first operand is completely evaluated and all side effects are completed before continuing evaluation of the logical AND expression.

The second operand is evaluated only if the first operand evaluates to true (nonzero). This evaluation eliminates needless evaluation of the second operand when the logical AND expression is false. You can use this short-circuit evaluation to prevent null-pointer dereferencing, as shown in the following example:

```
char *pch = 0;
...
(pch) && (*pch = 'a');
```

If `pch` is null (0), the right side of the expression is never evaluated. Therefore, the assignment through a null pointer is impossible.

Operator Keyword for &&

The **and** operator is the text equivalent of **&&**. There are two ways to access the **and** operator in your programs: include the header file `iso646.h`, or compile with the /Za (Disable language extensions) compiler option.

Example

```
// expre_Logical_AND_Operator.cpp
// compile with: /EHsc
// Demonstrate logical AND
#include <iostream>

using namespace std;

int main() {
   int a = 5, b = 10, c = 15;
   cout  << boolalpha
         << "The true expression "
         << "a < b && b < c yields "
         << (a < b && b < c) << endl
         << "The false expression "
         << "a > b && b < c yields "
         << (a > b && b < c) << endl;
}
```

See Also

Reference

C++ Logical Operators
C++ Operators
Operator Precedence and Associativity
C Logical Operators

Logical OR Operator: ||

```
logical-or-expression || logical-and-expression
```

Remarks

The logical OR operator (**||**) returns the boolean value **true** if either or both operands is **true** and returns **false** otherwise. The operands are implicitly converted to type **bool** prior to evaluation, and the result is of type **bool**. Logical OR has left-to-right associativity.

The operands to the logical OR operator need not be of the same type, but they must be of integral or pointer type. The operands are commonly relational or equality expressions.

The first operand is completely evaluated and all side effects are completed before continuing evaluation of the logical OR expression.

The second operand is evaluated only if the first operand evaluates to false (0). This eliminates needless evaluation of the second operand when the logical OR expression is true.

```
printf( "%d" , (x == w || x == y || x == z) );
```

In the above example, if `x` is equal to either `w`, `y`, or `z`, the second argument to the `printf` function evaluates to true and the value 1 is printed. Otherwise, it evaluates to false and the value 0 is printed. As soon as one of the conditions evaluates to true, evaluation ceases.

Operator Keyword for ||

The **or** operator is the text equivalent of **||**. There are two ways to access the **or** operator in your programs: include the header file `iso646.h`, or compile with the /Za (Disable language extensions) compiler option.

Example

```
// expre_Logical_OR_Operator.cpp
// compile with: /EHsc
// Demonstrate logical OR
#include <iostream>
using namespace std;
int main() {
   int a = 5, b = 10, c = 15;
   cout  << boolalpha
         << "The true expression "
         << "a < b || b > c yields "
         << (a < b || b > c) << endl
         << "The false expression "
         << "a > b || b > c yields "
         << (a > b || b > c) << endl;
}
```

See Also

Reference

C++ Logical Operators
C++ Operators
Operator Precedence and Associativity
C Logical Operators

Assignment Operators: =, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, and |=

```
expression assignment-operator expression
assignment-operator : one of
      =   *=   /=   %=   +=   -=   <<=   >>=   &=   ^=   |=
```

Remarks

Assignment operators store a value in the object designated by the left operand. There are two kinds of assignment operations: simple assignment, in which the value of the second operand is stored in the object specified by the first operand, and compound assignment, in which an arithmetic, shift, or bitwise operation is performed prior to storing the result. All assignment operators in the following table except the = operator are compound assignment operators.

Assignment Operators

Operator	Meaning
=	Store the value of the second operand in the object specified by the first operand (simple assignment).
***=**	Multiply the value of the first operand by the value of the second operand; store the result in the object specified by the first operand.
/=	Divide the value of the first operand by the value of the second operand; store the result in the object specified by the first operand.
%=	Take modulus of the first operand specified by the value of the second operand; store the result in the object specified by the first operand.
+=	Add the value of the second operand to the value of the first operand; store the result in the object specified by the first operand.
–=	Subtract the value of the second operand from the value of the first operand; store the result in the object specified by the first operand.
<<=	Shift the value of the first operand left the number of bits specified by the value of the second operand; store the result in the object specified by the first operand.
>>=	Shift the value of the first operand right the number of bits specified by the value of the second operand; store the result in the object specified by the first operand.
&=	Obtain the bitwise AND of the first and second operands; store the result in the object specified by the first operand.
^=	Obtain the bitwise exclusive OR of the first and second operands; store the result in the object specified by the first operand.
|=	Obtain the bitwise inclusive OR of the first and second operands; store the result in the object specified by the first operand.

Operator Keywords

Three of the compound assignment operators have text equivalents. They are:

Operator	Equivalent

&=	**and_eq**
|=	**or_eq**
^=	**xor_eq**

There are two ways to access these operator keywords in your programs: include the header file `iso646.h`, or compile with the /Za (Disable language extensions) compiler option.

Example

```
// expre_Assignment_Operators.cpp
// compile with: /EHsc
// Demonstrate assignment operators
#include <iostream>
using namespace std;
int main() {
   int a = 3, b = 6, c = 10, d = 0xAAAA, e = 0x5555;

   a += b;      // a is 9
   b %= a;      // b is 6
   c >>= 1;      // c is 5
   d |= e;      // Bitwise--d is 0xFFFF

   cout  << "a = 3, b = 6, c = 10, d = 0xAAAA, e = 0x5555" << endl
         << "a += b yields " << a << endl
         << "b %= a yields " << b << endl
         << "c >>= 1 yields " << c << endl
         << "d |= e yields " << hex << d << endl;
}
```

See Also

Reference

Expressions with Binary Operators
C++ Operators
Operator Precedence and Associativity
Result of Assignment Operators
Simple Assignment (C++)
Compound Assignment
C Assignment Operators

Result of Assignment Operators

The assignment operators return the value of the object specified by the left operand after the assignment. The resultant type is the type of the left operand. The result of an assignment expression is always an l-value. These operators have right-to-left associativity. The left operand must be a modifiable l-value.

Note
In ANSI C, the result of an assignment expression is not an l-value. Therefore, the legal C++ expression `(a += b) += c` is illegal in C.

See Also

Reference

Expressions with Binary Operators

Simple Assignment (C++)

The simple assignment operator (=) causes the value of the second operand to be stored in the object specified by the first operand. If both objects are of arithmetic types, the right operand is converted to the type of the left, prior to storing the value.

Objects of const and volatile types can be assigned to l-values of types that are just volatile or that are neither const nor volatile.

Assignment to objects of class type (struct, union, and class types) is performed by a function named operator=. The default behavior of this operator function is to perform a bitwise copy; however, this behavior can be modified using overloaded operators. (See Overloaded Operators for more information.)

An object of any unambiguously derived class from a given base class can be assigned to an object of the base class. The reverse is not true because there is an implicit conversion from derived class to base class but not from base class to derived class. For example:

```
// expre_SimpleAssignment.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
class ABase
{
public:
    ABase() { cout << "constructing ABase\n"; }
};

class ADerived : public ABase
{
public:
    ADerived() { cout << "constructing ADerived\n"; }
};

int main()
{
    ABase aBase;
    ADerived aDerived;

    aBase = aDerived; // OK
    aDerived = aBase; // C2679
}
```

Assignments to reference types behave as if the assignment were being made to the object to which the reference points.

For class-type objects, assignment is different from initialization. To illustrate how different assignment and initialization can be, consider the code

```
UserType1 A;
UserType2 B = A;
```

The preceding code shows an initializer; it calls the constructor for `UserType2` that takes an argument of type `UserType1`. Given the code

```
UserType1 A;
UserType2 B;

B = A;
```

the assignment statement

```
B = A;
```

can have one of the following effects:

- Call the function operator= for `UserType2`, provided operator= is provided with a `UserType1` argument.
- Call the explicit conversion function `UserType1::operator UserType2`, if such a function exists.
- Call a constructor `UserType2::UserType2`, provided such a constructor exists, that takes a `UserType1` argument and copies the result.

See Also

Reference

Expressions with Binary Operators

Compound Assignment

The compound assignment operators, shown in the table in Assignment Operators, are specified in the form *e1 op= e2*, where *e1* is a modifiable l-value not of const type and *e2* is one of the following:

- An arithmetic type
- A pointer, if *op* is + or –

The *e1 op= e2* form behaves as *e1 = e1 op e2*, but *e1* is evaluated only once.

Compound assignment to an enumerated type generates an error message. If the left operand is of a pointer type, the right operand must be of a pointer type or it must be a constant expression that evaluates to 0. If the left operand is of an integral type, the right operand must not be of a pointer type.

See Also

Reference

Expressions with Binary Operators

Comma Operator: ,

Allows grouping two statements where one is expected.

```
expression , expression
```

Remarks

The comma operator has left-to-right associativity. Two expressions separated by a comma are evaluated left to right. The left operand is always evaluated, and all side effects are completed before the right operand is evaluated.

Commas can be used as separators in some contexts, such as function argument lists. Do not confuse the use of the comma as a separator with its use as an operator; the two uses are completely different.

Consider the expression

e1 , *e2*

The type and value of the expression are the type and value of *e2*; the result of evaluating *e1* is discarded. The result is an l-value if the right operand is an l-value.

Where the comma is normally used as a separator (for example in actual arguments to functions or aggregate initializers), the comma operator and its operands must be enclosed in parentheses. For example:

```
func_one( x, y + 2, z );
func_two( (x--, y + 2), z );
```

In the function call to `func_one` above, three arguments, separated by commas, are passed: `x`, `y + 2`, and `z`. In the function call to `func_two`, parentheses force the compiler to interpret the first comma as the sequential-evaluation operator. This function call passes two arguments to `func_two`. The first argument is the result of the sequential-evaluation operation `(x--, y + 2)`, which has the value and type of the expression `y + 2`; the second argument is `z`.

Example

```
// cpp_comma_operator.cpp
#include <stdio.h>
int main () {
   int i = 10, b = 20, c= 30;
   i = b, c;
   printf("%i\n", i);

   i = (b, c);
   printf("%i\n", i);
}
```

Output

```
20
30
```

See Also

Reference

Expressions with Binary Operators

C++ Operators

Operator Precedence and Associativity

Concepts

Sequential-Evaluation Operator

Conditional Operator: ? :

```
expression ? expression : expression
```

Remarks

The conditional operator (**? :**) is a ternary operator (it takes three operands). The conditional operator works as follows:

- The first operand is implicitly converted to **bool**. It is evaluated and all side effects are completed before continuing.
- If the first operand evaluates to **true** (1), the second operand is evaluated.
- If the first operand evaluates to **false** (0), the third operand is evaluated.

The result of the conditional operator is the result of whichever operand is evaluated — the second or the third. Only one of the last two operands is evaluated in a conditional expression.

Conditional expressions have right-to-left associativity. The first operand must be of integral or pointer type. The following rules apply to the second and third expressions:

- If both expressions are of the same type, the result is of that type.
- If both expressions are of arithmetic or enumeration types, the usual arithmetic conversions (covered in Arithmetic Conversions) are performed to convert them to a common type.
- If both expressions are of pointer types or if one is a pointer type and the other is a constant expression that evaluates to 0, pointer conversions are performed to convert them to a common type.
- If both expressions are of reference types, reference conversions are performed to convert them to a common type.
- If both expressions are of type void, the common type is type void.
- If both expressions are of a given class type, the common type is that class type.

Any combinations of second and third operands not in the preceding list are illegal. The type of the result is the common type, and it is an l-value if both the second and third operands are of the same type and both are l-values.

Example

```
// expre_Expressions_with_the_Conditional_Operator.cpp
// compile with: /EHsc
// Demonstrate conditional operator
#include <iostream>
using namespace std;
int main() {
   int i = 1, j = 2;
   cout << ( i > j ? i : j ) << " is greater." << endl;
}
```

See Also

Reference

C++ Operators

Operator Precedence and Associativity

Concepts

Conditional-Expression Operator

C++ Constant Expressions

C++ requires constant expressions — expressions that evaluate to a constant — for declarations of:

- Array bounds
- Selectors in case statements
- Bit-field length specification
- Enumeration initializers

The only operands that are legal in constant expressions are:

- Literals
- Enumeration constants
- Values declared as const that are initialized with constant expressions
- sizeof expressions

Nonintegral constants must be converted (either explicitly or implicitly) to integral types to be legal in a constant expression. Therefore, the following code is legal:

```
const double Size = 11.0;
char chArray[(int)Size];
```

Explicit conversions to integral types are legal in constant expressions; all other types and derived types are illegal except when used as operands to the sizeof operator.

The comma operator and assignment operators cannot be used in constant expressions.

See Also

Reference

Types of Expressions

Expressions with Explicit Type Conversions

C++ provides implicit type conversion, as described in Standard Conversions. You can also specify explicit type conversions when you need more precise control of the conversions applied.

See Also

Reference

Types of Expressions

Explicit Type Conversion Operator: ()

C++ allows explicit type conversion using syntax similar to the function-call syntax.

```
simple-type-name ( expression-list )
```

Remarks

A *simple-type-name* followed by an *expression-list* enclosed in parentheses constructs an object of the specified type using the specified expressions. The following example shows an explicit type conversion to type int:

```
int i = int( d );
```

The following example uses a modified version of the `Point` class defined in Function-Call Results.

Example

```
// expre_Explicit_Type_Conversion_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class Point
{
public:
    // Define default constructor.
    Point() { _x = _y = 0; }
    // Define another constructor.
    Point( int X, int Y ) { _x = X; _y = Y; }

    // Define "accessor" functions as
    // reference types.
    unsigned& x() { return _x; }
    unsigned& y() { return _y; }
    void Show()   { cout << "x = " << _x << ", "
                         << "y = " << _y << "\n"; }
private:
    unsigned _x;
    unsigned _y;
};

int main()
{
    Point Point1, Point2;

    // Assign Point1 the explicit conversion
    //  of ( 10, 10 ).
    Point1 = Point( 10, 10 );

    // Use x() as an l-value by assigning an explicit
    //  conversion of 20 to type unsigned.
    Point1.x() = unsigned( 20 );
    Point1.Show();

    // Assign Point2 the default Point object.
    Point2 = Point();
    Point2.Show();
}
```

Output

```
x = 20, y = 10
x = 0, y = 0
```

Although the preceding example demonstrates explicit type conversion using constants, the same technique works to perform these conversions on objects. The following code fragment demonstrates this:

```
int i = 7;
float d;

d = float( i );
```

Explicit type conversions can also be specified using the "cast" syntax. The previous example, rewritten using the cast syntax, is:

```
d = (float)i;
```

Both cast and function-style conversions have the same results when converting from single values. However, in the function-style syntax, you can specify more than one argument for conversion. This difference is important for user-defined types. Consider a `Point` class and its conversions:

```
struct Point
{
    Point( short x, short y ) { _x = x; _y = y; }
    ...
    short _x, _y;
};
...
Point pt = Point( 3, 10 );
```

The preceding example, which uses function-style conversion, shows how to convert two values (one for *x* and one for *y*) to the user-defined type `Point`.

Caution
Use the explicit type conversions with care, since they override the C++ compiler's built-in type checking.

The cast notation must be used for conversions to types that do not have a *simple-type-name* (pointer or reference types, for example). Conversion to types that can be expressed with a *simple-type-name* can be written in either form. See Type Specifiers for more information about what constitutes a *simple-type-name*.

Type definition within casts is illegal.

See Also

Reference

Postfix Expressions

C++ Operators

Operator Precedence and Associativity

Legal Conversions

You can do explicit conversions from a given type to another type if the conversion can be done using standard conversions. The results are the same. The conversions described in this section are legal; any other conversions not explicitly defined by the user (for a class type) are illegal.

A value of integral type can be explicitly converted to a pointer if the pointer is large enough to hold the integral value. A pointer that is converted to an integral value can be converted back to a pointer; its value is the same. This identity is given by the following (where *p* represents a pointer of any type):

p == (*type* *) *integral-conversion*(*p*)

With explicit conversions, the compiler does not check whether the converted value fits in the new type except when converting from pointer to integral type or vice versa.

This section describes the following conversions:

- Converting pointer types
- Converting the null pointer
- Converting to a forward reference class type
- Converting to reference types
- Converting among pointer to member types

See Also

Reference

Expressions with Explicit Type Conversions

Converting Pointer Types

A pointer to one object type can be explicitly converted to a pointer of another object type. A pointer declared as void * is considered a pointer to any object type.

A pointer to a base class can be explicitly converted to a pointer to a derived class as long as these conditions are met:

- There is an unambiguous conversion.
- The base class is not declared as virtual at any point.

Because conversion to type void * can change the representation of an object, there is no guarantee that the conversion *type1** void * *type2** is equivalent to the conversion *type1** *type2** (which is a change in value only).

When such a conversion is performed, the result is a pointer to the subobject of the original object representing the base class.

See Derived Classes, for more information about ambiguity and virtual base classes.

C++ allows explicit conversions of pointers to objects or functions to type void *.

Pointers to object types can be explicitly converted to pointers to functions if the function pointer type has enough bits to accommodate the pointer to object type.

A pointer to a const object can be explicitly converted to a pointer not of const type. The result of this conversion points to the original object. An object of const type, or a reference to an object of const type, can be cast to a reference to a type that is not const. The result is a reference to the original object. The original object was probably declared as const because it was to remain constant across the duration of the program. Therefore, an explicit conversion defeats this safety mechanism, allowing modification of such objects. The behavior in such cases is undefined.

A pointer to an object of volatile type can be cast to a pointer to a type that is not volatile. The result of this conversion refers to the original object. Similarly, an object of volatile type can be cast to a reference to a type that is not volatile.

See Also
Reference
Legal Conversions

Converting the Null Pointer

The null pointer (0) is converted into itself.

See Also

Reference

Legal Conversions

Converting to a Forward-Reference Class Type

A class that has been declared but not yet defined (a forward reference) can be used in a pointer cast. In this case, the compiler returns a pointer to the original object, not to a subobject as it might if the class's relationships were known.

See Also

Reference

Legal Conversions

Converting to Reference Types

Any object whose address can be converted to a given pointer type can also be converted to the analogous reference type. For example, any object whose address can be converted to type char * can also be converted to type char &. No constructors or class conversion functions are called to make a conversion to a reference type.

Objects or values can be converted to class-type objects only if a constructor or conversion operator has been provided specifically for this purpose. For more information about these user-defined functions, see Conversion Constructors.

Conversion of a reference to a base class, to a reference to a derived class (and vice versa) is done the same way as for pointers.

A cast to a reference type results in an l-value. The results of casts to other types are not l-values. Operations performed on the result of a pointer or reference cast are still performed on the original object.

See Also
Reference
Legal Conversions

Converting Among Pointer-to-Member Types

A pointer to a member can be converted to a different pointer-to-member type subject to these rules: Either the pointers must both be pointers to members in the same class or they must be pointers to members of classes, one of which is derived unambiguously from the other. When converting pointer-to-member functions, the return and argument types must match.

See Also

Reference

Legal Conversions

Pointer-to-Member Operators: .* and ->*

```
expression .* expression
expression ->* expression
```

Remarks

The pointer-to-member operators, .* and ->*, return the value of a specific class member for the object specified on the left side of the expression. The right side must specify a member of the class. The following example shows how to use these operators:

```
// expre_Expressions_with_Pointer_Member_Operators.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

class Testpm {
public:
   void m_func1() { cout << "m_func1\n"; }
   int m_num;
};

// Define derived types pmfn and pmd.
// These types are pointers to members m_func1() and
// m_num, respectively.
void (Testpm::*pmfn)() = &Testpm::m_func1;
int Testpm::*pmd = &Testpm::m_num;

int main() {
   Testpm ATestpm;
   Testpm *pTestpm = new Testpm;

// Access the member function
   (ATestpm.*pmfn)();
   (pTestpm->*pmfn)();   // Parentheses required since * binds
                        // less tightly than the function call.

// Access the member data
   ATestpm.*pmd = 1;
   pTestpm->*pmd = 2;

   cout  << ATestpm.*pmd << endl
         << pTestpm->*pmd << endl;
   delete pTestpm;
}
```

Output

```
m_func1
m_func1
1
2
```

In the preceding example, a pointer to a member, `pmfn`, is used to invoke the member function `m_func1`. Another pointer to a member, `pmd`, is used to access the `m_num` member.

The binary operator .* combines its first operand, which must be an object of class type, with its second operand, which must be a pointer-to-member type.

The binary operator ->* combines its first operand, which must be a pointer to an object of class type, with its second operand, which must be a pointer-to-member type.

In an expression containing the .* operator, the first operand must be of the class type of, and be accessible to, the pointer to member specified in the second operand or of an accessible type unambiguously derived from and accessible to that class.

In an expression containing the –>* operator, the first operand must be of the type "pointer to the class type" of the type specified in the second operand, or it must be of a type unambiguously derived from that class.

Example

Consider the following classes and program fragment:

```
// expre_Expressions_with_Pointer_Member_Operators2.cpp
// C2440 expected
class BaseClass {
public:
   BaseClass(); // Base class constructor.
   void Func1();
};

// Declare a pointer to member function Func1.
void (BaseClass::*pmfnFunc1)() = &BaseClass::Func1;

class Derived : public BaseClass {
public:
   Derived();  // Derived class constructor.
   void Func2();
};

// Declare a pointer to member function Func2.
void (Derived::*pmfnFunc2)() = &Derived::Func2;

int main() {
   BaseClass ABase;
   Derived ADerived;

   (ABase.*pmfnFunc1)();   // OK: defined for BaseClass.
   (ABase.*pmfnFunc2)();   // Error: cannot use base class to
                           // access pointers to members of
                           // derived classes.

   (ADerived.*pmfnFunc1)();   // OK: Derived is unambiguously
                              // derived from BaseClass.
   (ADerived.*pmfnFunc2)();   // OK: defined for Derived.
}
```

The result of the .* or –>* pointer-to-member operators is an object or function of the type specified in the declaration of the pointer to member. So, in the preceding example, the result of the expression `ADerived.*pmfnFunc1()` is a pointer to a function that returns void. This result is an l-value if the second operand is an l-value.

Note

If the result of one of the pointer-to-member operators is a function, then the result can be used only as an operand to the function call operator.

See Also

Reference

C++ Operators

Operator Precedence and Associativity

Semantics of Expressions

This section explains when, and in what order, expressions are evaluated.

The following topics are included:

- Order of evaluation
- Sequence points
- Ambiguous expressions
- Notation in expressions

See Also

Reference

Expressions (C++)

Order of Evaluation

This section discusses the order in which expressions are evaluated but does not explain the syntax or the semantics of the operators in these expressions. The earlier sections provide a complete reference for each of these operators.

Expressions are evaluated according to the precedence and grouping of their operators. (Operator Precedence and Associativity in Lexical Conventions, shows the relationships the C++ operators impose on expressions.) Consider this example:

Example

```
// expre_pluslang__pluslang_Order_of_Evaluation.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main()
{
    int a = 2, b = 4, c = 9;

    cout << a + b * c << "\n";
    cout << a + (b * c) << "\n";
    cout << (a + b) * c << "\n";
}
```

Output

```
38
38
54
```

Expression-Evaluation Order

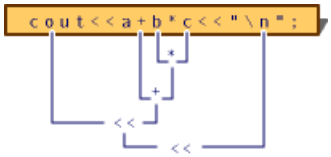

The order in which the expression shown in the above figure is evaluated is determined by the precedence and associativity of the operators:

1. Multiplication (*) has the highest precedence in this expression; hence the subexpression `b * c` is evaluated first.
2. Addition (+) has the next highest precedence, so `a` is added to the product of `b` and `c`.
3. Left shift (<<) has the lowest precedence in the expression, but there are two occurrences. Because the left-shift operator groups left-to-right, the left subexpression is evaluated first and then the right one.

When parentheses are used to group the subexpressions, they alter the precedence and also the order in which the expression is evaluated, as shown in the following figure.

Expression-Evaluation Order with Parentheses

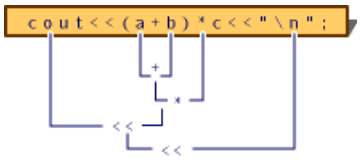

Expressions such as those in the above figure are evaluated purely for their side effects — in this case, to transfer information to the standard output device.

See Also

Reference

Semantics of Expressions

C++ Sequence Points

An expression can modify an object's value only once between consecutive "sequence points."

Microsoft Specific

The C++ language definition does not currently specify sequence points. Microsoft C++ uses the same sequence points as ANSI C for any expression involving C operators and not involving overloaded operators. When operators are overloaded, the semantics change from operator sequencing to function-call sequencing. Microsoft C++ uses the following sequence points:

- Left operand of the logical AND operator (&&). The left operand of the logical AND operator is completely evaluated and all side effects completed before continuing. There is no guarantee that the right operand of the logical AND operator will be evaluated.
- Left operand of the logical OR operator (||). The left operand of the logical OR operator is completely evaluated and all side effects completed before continuing. There is no guarantee that the right operand of the logical OR operator will be evaluated.
- Left operand of the comma operator. The left operand of the comma operator is completely evaluated and all side effects completed before continuing. Both operands of the comma operator are always evaluated.
- Function-call operator. The function-call expression and all arguments to a function, including default arguments, are evaluated and all side effects completed prior to entry to the function. There is no specified order of evaluation among the arguments or the function-call expression.
- First operand of the conditional operator. The first operand of the conditional operator is completely evaluated and all side effects completed before continuing.
- The end of a full initialization expression, such as the end of an initialization in a declaration statement.
- The expression in an expression statement. Expression statements consist of an optional expression followed by a semicolon (;). The expression is completely evaluated for its side effects.
- The controlling expression in a selection (if or switch) statement. The expression is completely evaluated and all side effects completed before the code dependent on the selection is executed.
- The controlling expression of a while or do statement. The expression is completely evaluated and all side effects completed before any statements in the next iteration of the while or do loop are executed.
- Each of the three expressions of a for statement. Each expression is completely evaluated and all side effects completed before moving to the next expression.
- The expression in a return statement. The expression is completely evaluated and all side effects completed before control returns to the calling function.

END Microsoft Specific

See Also

Reference

Semantics of Expressions

Ambiguous Expressions

Certain expressions are ambiguous in their meaning. These expressions occur most frequently when an object's value is modified more than once in the same expression. These expressions rely on a particular order of evaluation where the language does not define one. Consider the following example:

```
int i = 7;

func( i, ++i );
```

The C++ language does not guarantee the order in which arguments to a function call are evaluated. Therefore, in the preceding example, `func` could receive the values 7 and 8, or 8 and 8 for its parameters, depending on whether the parameters are evaluated from left to right or from right to left.

See Also

Reference

Semantics of Expressions

Notation in Expressions

The C++ language specifies certain compatibilities when specifying operands. The following table shows the types of operands acceptable to operators that require operands of type *type*.

Operand Types Acceptable to Operators

Type expected	Types allowed
type	**const** *type* **volatile** *type* *type*& **const** *type*& **volatile** *type*& **volatile const** *type* **volatile const** *type*&
*type**	*type** **const***type** **volatile***type** **volatile const**
const *type*	*type* **const** *type***const** *type*&
volatile *type*	*type* **volatile** *type***volatile** *type*&

Because the preceding rules can always be used in combination, a const pointer to a volatile object can be supplied where a pointer is expected.

See Also

Reference

Semantics of Expressions

Casting

The C++ language provides that if a class is derived from a base class containing virtual functions, a pointer to that base class type can be used to call the implementations of the virtual functions residing in the derived class object. A class containing virtual functions is sometimes called a "polymorphic class."

Since a derived class completely contains the definitions of all the base classes from which it is derived, it is safe to cast a pointer up the class hierarchy to any of these base classes. Given a pointer to a base class, it might be safe to cast the pointer down the hierarchy. It is safe if the object being pointed to is actually of a type derived from the base class. In this case, the actual object is said to be the "complete object." The pointer to the base class is said to point to a "subobject" of the complete object. For example, consider the class hierarchy shown in the following figure.

Class Hierarchy

An object of type `C` could be visualized as shown in the following figure.

Class C with B Subobject and A Subobject

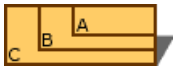

Given an instance of class `C`, there is a `B` subobject and an `A` subobject. The instance of `C`, including the `A` and `B` subobjects, is the "complete object."

Using run-time type information, it is possible to check whether a pointer actually points to a complete object and can be safely cast to point to another object in its hierarchy. The dynamic_cast operator can be used to make these types of casts. It also performs the run-time check necessary to make the operation safe.

For conversion of nonpolymorphic types, you can use the static_cast operator (this topic explains the difference between static and dynamic casting conversions, and when it is appropriate to use each).

This section covers the following topics:

- Casting operators
- Run-time type information

See Also
Reference
Expressions (C++)

Casting Operators

There are several casting operators specific to the C++ language. These operators are intended to remove some of the ambiguity and danger inherent in old style C language casts. These operators are:

- dynamic_cast Used for conversion of polymorphic types.
- static_cast Used for conversion of nonpolymorphic types.
- const_cast Used to remove the **const**, **volatile**, and **__unaligned** attributes.
- reinterpret_cast Used for simple reinterpretation of bits.
- safe_cast Used to produce verifiable MSIL.

Use **const_cast** and **reinterpret_cast** as a last resort, since these operators present the same dangers as old style casts. However, they are still necessary in order to completely replace old style casts.

See Also
Reference
Casting

dynamic_cast Operator

Converts the operand *expression* to an object of type *type-id*.

```
dynamic_cast < type-id > ( expression )
```

Remarks

The *type-id* must be a pointer or a reference to a previously defined class type or a "pointer to void". The type of *expression* must be a pointer if *type-id* is a pointer, or an l-value if *type-id* is a reference.

For more information about dynamic_cast, see Breaking Changes in dynamic_cast.

See static_cast for an explanation of the difference between static and dynamic casting conversions, and when it is appropriate to use each.

In Visual C++ 2005, there are two breaking changes in the behavior of **dynamic_cast** in managed code:

- **dynamic_cast** to a pointer to the underlying type of a boxed enum will fail at runtime, returning 0 instead of the converted pointer.
- **dynamic_cast** will no longer throw an exception when *type-id* is an interior pointer to a value type, with the cast failing at runtime. The cast will now return the 0 pointer value instead of throwing.

If *type-id* is a pointer to an unambiguous accessible direct or indirect base class of *expression*, a pointer to the unique subobject of type *type-id* is the result. For example:

```
// dynamic_cast_1.cpp
// compile with: /c
class B { };
class C : public B { };
class D : public C { };

void f(D* pd) {
   C* pc = dynamic_cast<C*>(pd);   // ok: C is a direct base class
                                   // pc points to C subobject of pd
   B* pb = dynamic_cast<B*>(pd);   // ok: B is an indirect base class
                                   // pb points to B subobject of pd
}
```

This type of conversion is called an "upcast" because it moves a pointer up a class hierarchy, from a derived class to a class it is derived from. An upcast is an implicit conversion.

If *type-id* is void*, a run-time check is made to determine the actual type of *expression*. The result is a pointer to the complete object pointed to by *expression*. For example:

```
// dynamic_cast_2.cpp
// compile with: /c /GR
class A {virtual void f();};
class B {virtual void f();};

void f() {
   A* pa = new A;
   B* pb = new B;
   void* pv = dynamic_cast<void*>(pa);
   // pv now points to an object of type A

   pv = dynamic_cast<void*>(pb);
   // pv now points to an object of type B
}
```

If *type-id* is not void*, a run-time check is made to see if the object pointed to by *expression* can be converted to the type

pointed to by *type-id*.

If the type of *expression* is a base class of the type of *type-id*, a run-time check is made to see if *expression* actually points to a complete object of the type of *type-id*. If this is true, the result is a pointer to a complete object of the type of *type-id*. For example:

```
// dynamic_cast_3.cpp
// compile with: /c /GR
class B {virtual void f();};
class D : public B {virtual void f();};

void f() {
   B* pb = new D;   // unclear but ok
   B* pb2 = new B;

   D* pd = dynamic_cast<D*>(pb);   // ok: pb actually points to a D
   D* pd2 = dynamic_cast<D*>(pb2);   // pb2 points to a B not a D
}
```

This type of conversion is called a "downcast" because it moves a pointer down a class hierarchy, from a given class to a class derived from it.

In cases of multiple inheritance, possibilities for ambiguity are introduced. Consider the class hierarchy shown in the following figure.

For CLR types, **dynamic_cast** results in either a no-op if the conversion can be performed implicitly, or an MSIL **isinst** instruction, which performs a dynamic check and returns **nullptr** if the conversion fails.

The following sample uses **dynamic_cast** to determine if a class is an instance of particular type:

```
// dynamic_cast_clr.cpp
// compile with: /clr
using namespace System;

void PrintObjectType( Object^o ) {
   if( dynamic_cast<String^>(o) )
      Console::WriteLine("Object is a String");
   else if( dynamic_cast<int^>(o) )
      Console::WriteLine("Object is an int");
}

int main() {
   Object^o1 = "hello";
   Object^o2 = 10;

   PrintObjectType(o1);
   PrintObjectType(o2);
}
```

Class Hierarchy Showing Multiple Inheritance

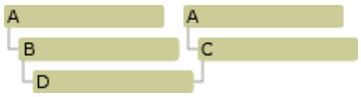

A pointer to an object of type `D` can be safely cast to `B` or `C`. However, if `D` is cast to point to an `A` object, which instance of `A` would result? This would result in an ambiguous casting error. To get around this problem, you can perform two unambiguous casts. For example:

```
// dynamic_cast_4.cpp
// compile with: /c /GR
class A {virtual void f();};
class B {virtual void f();};
class D : public B {virtual void f();};

void f() {
```

```
   D* pd = new D;
   B* pb = dynamic_cast<B*>(pd);   // first cast to B
   A* pa2 = dynamic_cast<A*>(pb);   // ok: unambiguous
}
```

Further ambiguities can be introduced when you use virtual base classes. Consider the class hierarchy shown in the following figure.

Class Hierarchy Showing Virtual Base Classes

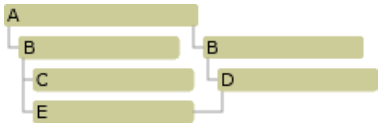

In this hierarchy, `A` is a virtual base class. See Virtual Base Classes for the definition of a virtual base class. Given an instance of class `E` and a pointer to the `A` subobject, a **dynamic_cast** to a pointer to `B` will fail due to ambiguity. You must first cast back to the complete `E` object, then work your way back up the hierarchy, in an unambiguous manner, to reach the correct `B` object.

Consider the class hierarchy shown in the following figure.

Class Hierarchy Showing Duplicate Base Classes

Given an object of type `E` and a pointer to the `D` subobject, to navigate from the `D` subobject to the left-most `A` subobject, three conversions can be made. You can perform a **dynamic_cast** conversion from the `D` pointer to an `E` pointer, then a conversion (either **dynamic_cast** or an implicit conversion) from `E` to `B`, and finally an implicit conversion from `B` to `A`. For example:

```
// dynamic_cast_5.cpp
// compile with: /c /GR
class A {virtual void f();};
class B : public A {virtual void f();};
class C : public A { };
class D {virtual void f();};
class E : public B, public C, public D {virtual void f();};

void f(D* pd) {
   E* pe = dynamic_cast<E*>(pd);
   B* pb = pe;   // upcast, implicit conversion
   A* pa = pb;   // upcast, implicit conversion
}
```

The **dynamic_cast** operator can also be used to perform a "cross cast." Using the same class hierarchy, it is possible to cast a pointer, for example, from the `B` subobject to the `D` subobject, as long as the complete object is of type `E`.

Considering cross casts, it is actually possible to do the conversion from a pointer to `D` to a pointer to the left-most `A` subobject in just two steps. You can perform a cross cast from `D` to `B`, then an implicit conversion from `B` to `A`. For example:

```
// dynamic_cast_6.cpp
// compile with: /c /GR
class A {virtual void f();};
class B : public A {virtual void f();};
class C : public A { };
class D {virtual void f();};
class E : public B, public C, public D {virtual void f();};

void f(D* pd) {
   B* pb = dynamic_cast<B*>(pd);   // cross cast
   A* pa = pb;   // upcast, implicit conversion
}
```

A null pointer value is converted to the null pointer value of the destination type by **dynamic_cast**.

When you use **dynamic_cast** < *type-id* > **(** *expression* **)**, if *expression* cannot be safely converted to type *type-id*, the run-time check causes the cast to fail. For example:

```
// dynamic_cast_7.cpp
// compile with: /c /GR
class A {virtual void f();};
class B {virtual void f();};

void f() {
   A* pa = new A;
   B* pb = dynamic_cast<B*>(pa);   // fails at runtime, not safe;
   // B not derived from A
}
```

The value of a failed cast to pointer type is the null pointer. A failed cast to reference type throws a bad_cast Exception. If *expression* does not point to or reference a valid object, a **__non_rtti_object** exception is thrown.

See typeid for an explanation of the **__non_rtti_object** exception.

Example

The following sample creates the base class (struct A) pointer, to an object (struct C). This, plus the fact there are virtual functions, enables runtime polymorphism.

The sample also calls a non-virtual function in the hierarchy.

```
// dynamic_cast_8.cpp
// compile with: /GR /EHsc
#include <stdio.h>
#include <iostream>

struct A {
    virtual void test() {
        printf_s("in A\n");
   }
};

struct B : A {
    virtual void test() {
        printf_s("in B\n");
    }

    void test2() {
        printf_s("test2 in B\n");
    }
};

struct C : B {
    virtual void test() {
        printf_s("in C\n");
    }

    void test2() {
        printf_s("test2 in C\n");
    }
};

void Globaltest(A& a) {
    try {
        C &c = dynamic_cast<C&>(a);
        printf_s("in GlobalTest\n");
    }
    catch(std::bad_cast) {
        printf_s("Can't cast to C\n");
    }
}

int main() {
```

```
    A *pa = new C;
    A *pa2 = new B;

    pa->test();

    B * pb = dynamic_cast<B *>(pa);
    if (pb)
        pb->test2();

    C * pc = dynamic_cast<C *>(pa2);
    if (pc)
        pc->test2();

    C ConStack;
    Globaltest(ConStack);

   // will fail because B knows nothing about C
    B BonStack;
    Globaltest(BonStack);
}
```

Output

```
in C
test2 in B
in GlobalTest
Can't cast to C
```

See Also

Reference

Casting Operators

C++ Keywords

Breaking Changes in dynamic_cast

In Microsoft Visual C++ 2005 as well as in Visual C++ .NET 2002, there were changes in the **dynamic_cast** operator, such that, the behavior of your application may change.

- The Visual C++ 6.0 compiler allowed a runtime check to occur, when a compile-time check was required by the standard. This was fixed in Visual C++ .NET 2002 (Visual C++ 7.0).
- The C runtime library now does a **dynamic_cast** runtime check to ensure the compile-time type of the expression being cast refers to a public base class sub-object of either the cast target type (for down-cast) or most-derived object type (for cross-cast).

For more information on compiler changes, see Breaking Changes in the Visual C++ Compiler.

Example

In this code sample, the variable pA2 is NULL in VC6; the compiler did not perform an identity check as specified in 5.2.7/3 in the ISO C++ Standard. However, this cast will succeed beginning in Visual C++ .NET 2002 (Visual C++ 7.0).

So, while this caused an exception to be thrown in Visual C++ 6.0, no exception in thrown now.

```
// dynamic_cast_breaking_change.cpp
struct A {
   virtual void F() {}
};

struct B {
   virtual void F() {}
};

void Test(A* pA) {
   A* pA2 = dynamic_cast<A*>(pA);
   A& rA = dynamic_cast<A&>(*pA);
}

int main() {
   B* pB = new B;
   Test(reinterpret_cast<A*>(pB));
}
```

Example

This sample shows that the C runtime library now does a **dynamic_cast** runtime check to ensure the compile-time type of the expression being cast refers to a public base class sub-object of either the cast target type (for down-cast) or most-derived object type (for cross-cast).

In previous Visual C++ releases, these dynamic casts succeeded.

```
// dynamic_cast_breaking_change_2.cpp
#include "stdio.h"

struct A {
   virtual void Test() {}
};

struct B : virtual private A {
   virtual void Test() {}
};

struct D : virtual private A {
   virtual void Test() {}
};

struct C : public B, public D {
```

```
   virtual void Test() {}
};

int main() {
   C c;
   printf("%p\n", dynamic_cast<B*>((A*)&c) );
   printf("%p\n", dynamic_cast<C*>((A*)&c) );
}
```

Sample Output

```
00000000
00000000
```

Example

This sample shows that if any sub-objects from the source to the target of the cast are not public, the **dynamic_cast** cast will fail.

```
// dynamic_cast_breaking_change_3.cpp
#include "stdio.h"
struct A {
   virtual void Test() {}
};

struct B : virtual public A {
   virtual void Test() {}
};

struct C : virtual private B {
   virtual void Test() {}
};

int main() {
   C c;
   printf("%p\n", dynamic_cast<B*>((A*)&c) );
   printf("%p\n", dynamic_cast<C*>((A*)&c) );
}
```

Sample Output

```
0012FF70
00000000
```

See Also

Reference

dynamic_cast Operator

bad_cast Exception

The **bad_cast** exception is thrown by the **dynamic_cast** operator as the result of a failed cast to a reference type.

```
catch (bad_cast)
      statement
```

Remarks

The interface for **bad_cast** is:

```
class bad_cast : public exception {
public:
   bad_cast(const char * _Message = "bad cast");
   bad_cast(const bad_cast &);
   virtual ~bad_cast();
};
```

The following code contains an example of a failed **dynamic_cast** that throws the **bad_cast** exception.

```
// expre_bad_cast_Exception.cpp
// compile with: /EHsc /GR
#include <typeinfo.h>
#include <iostream>

class Shape {
public:
   virtual void virtualfunc() const {}
};

class Circle: public Shape {
public:
   virtual void virtualfunc() const {};
};

using namespace std;
int main() {
   Shape shape_instance;
   Shape& ref_shape = shape_instance;
   try {
      Circle& ref_circle = dynamic_cast<Circle&>(ref_shape);
   }
   catch (bad_cast) {
      cout << "Caught: bad_cast exception. A Shape is not a Circle.\n";
   }
}
```

The exception is thrown because the object being cast (a Shape) is not derived from the specified cast type (Circle). To avoid the exception, add these declarations to `main`:

```
   Circle circle_instance;
   Circle& ref_circle = circle_instance;
```

Then reverse the sense of the cast in the `try` block as follows:

```
   Shape& ref_shape = dynamic_cast<Shape&>(ref_circle);
```

See Also

Reference

dynamic_cast Operator

C++ Keywords
C++ Exception Handling

static_cast Operator

Converts *expression* to the type of *type-id* based solely on the types present in the expression.

```
static_cast <type-id> ( expression )
```

Remarks

No run-time type check is made to help ensure the safety of the conversion.

The **static_cast** operator can be used for operations such as converting a pointer to a base class to a pointer to a derived class. Such conversions are not always safe.

In general you use **static_cast** when you want to convert numeric data types such as enums to ints or ints to floats, and you are certain of the data types involved in the conversion. **static_cast** conversions are not as safe as **dynamic_cast** conversions, because **static_cast** does no run-time type check, while **dynamic_cast** does. A **dynamic_cast** to an ambiguous pointer will fail, while a **static_cast** returns as if nothing were wrong; this can be dangerous. Although **dynamic_cast** conversions are safer, **dynamic_cast** only works on pointers or references, and the run-time type check is an overhead. See dynamic_cast for more details.

For example:

```
// static_cast_Operator.cpp
// compile with: /LD
class B {};

class D : public B {};

void f(B* pb, D* pd) {
   D* pd2 = static_cast<D*>(pb);   // not safe, pb may
                                   // point to just B

   B* pb2 = static_cast<B*>(pd);   // safe conversion
}
```

In contrast to dynamic_cast, no run-time check is made on the **static_cast** conversion of `pb`. The object pointed to by `pb` may not be an object of type `D`, in which case the use of `*pd2` could be disastrous. For instance, calling a function that is a member of the `D` class, but not the `B` class, could result in an access violation.

The **dynamic_cast** and **static_cast** operators move a pointer throughout a class hierarchy. However, **static_cast** relies exclusively on the information provided in the cast statement and can therefore be unsafe. For example:

```
// static_cast_Operator_2.cpp
// compile with: /LD /GR
class B {
public:
   virtual void Test(){}
};
class D : public B {};

void f(B* pb) {
   D* pd1 = dynamic_cast<D*>(pb);
   D* pd2 = static_cast<D*>(pb);
}
```

If `pb` really points to an object of type `D`, then `pd1` and `pd2` will get the same value. They will also get the same value if `pb == 0`.

If `pb` points to an object of type `B` and not to the complete `D` class, then **dynamic_cast** will know enough to return zero. However, **static_cast** relies on the programmer's assertion that `pb` points to an object of type `D` and simply returns a pointer to that supposed `D` object.

Consequently, **static_cast** can do the inverse of implicit conversions, in which case the results are undefined. It is left to the

programmer to verify that the results of a **static_cast** conversion are safe.

This behavior also applies to types other than class types. For instance, **static_cast** can be used to convert from an int to a **char**. However, the resulting **char** may not have enough bits to hold the entire **int** value. Again, it is left to the programmer to verify that the results of a **static_cast** conversion are safe.

The **static_cast** operator can also be used to perform any implicit conversion, including standard conversions and user-defined conversions. For example:

```
// static_cast_Operator_3.cpp
// compile with: /LD /GR
typedef unsigned char BYTE;

void f() {
   char ch;
   int i = 65;
   float f = 2.5;
   double dbl;

   ch = static_cast<char>(i);   // int to char
   dbl = static_cast<double>(f);   // float to double
   i = static_cast<BYTE>(ch);
}
```

The **static_cast** operator can explicitly convert an integral value to an enumeration type. If the value of the integral type does not fall within the range of enumeration values, the resulting enumeration value is undefined.

The **static_cast** operator converts a null pointer value to the null pointer value of the destination type.

Any expression can be explicitly converted to type void by the **static_cast** operator. The destination void type can optionally include the **const**, **volatile**, or **__unaligned** attribute.

The **static_cast** operator cannot cast away the **const**, **volatile**, or **__unaligned** attributes. See const_cast Operator for information on removing these attributes.

Due to the danger of performing unchecked casts on top of a relocating garbage collector, the use of **static_cast** should only be in performance-critical code when you are certain it will work correctly. If you must use **static_cast** in release mode, substitute it with safe_cast in your debug builds to ensure success.

See Also

Reference

Casting Operators

C++ Keywords

const_cast Operator

Removes the **const**, **volatile**, and **__unaligned** attribute(s) from a class.

```
const_cast < type-id > ( expression )
```

Remarks

A pointer to any object type or a pointer to a data member can be explicitly converted to a type that is identical except for the **const**, **volatile**, and **__unaligned** qualifiers. For pointers and references, the result will refer to the original object. For pointers to data members, the result will refer to the same member as the original (uncast) pointer to data member. Depending on the type of the referenced object, a write operation through the resulting pointer, reference, or pointer to data member might produce undefined behavior.

You cannot use the **const_cast** operator to directly override a constant variable's constant status.

The **const_cast** operator converts a null pointer value to the null pointer value of the destination type.

Example

```
// expre_const_cast_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class CCTest {
public:
   void setNumber( int );
   void printNumber() const;
private:
   int number;
};

void CCTest::setNumber( int num ) { number = num; }

void CCTest::printNumber() const {
   cout << "\nBefore: " << number;
   const_cast< CCTest * >( this )->number--;
   cout << "\nAfter: " << number;
}

int main() {
   CCTest X;
   X.setNumber( 8 );
   X.printNumber();
}
```

On the line containing the **const_cast**, the data type of the `this` pointer is `const CCTest *`. The **const_cast** operator changes the data type of the `this` pointer to `CCTest *`, allowing the member `number` to be modified. The cast lasts only for the remainder of the line on which it appears.

See Also

Reference

Casting Operators

C++ Keywords

reinterpret_cast Operator

Allows any pointer to be converted into any other pointer type.

```
reinterpret_cast < type-id > ( expression )
```

Remarks

The **reinterpret_cast** operator also allows any integral type to be converted into any pointer type and vice versa. Misuse of the **reinterpret_cast** operator can easily be unsafe. Unless the desired conversion is inherently low-level, you should use one of the other cast operators.

The **reinterpret_cast** operator can be used for conversions such as `char*` to `int*`, or `One_class*` to `Unrelated_class*`, which are inherently unsafe.

The result of a **reinterpret_cast** cannot safely be used for anything other than being cast back to its original type. Other uses are, at best, nonportable.

The **reinterpret_cast** operator cannot cast away the **const**, **volatile**, or **__unaligned** attributes. See const_cast Operator for information on removing these attributes.

The **reinterpret_cast** operator converts a null pointer value to the null pointer value of the destination type.

One practical use of **reinterpret_cast** is in a hash function, which maps a value to an index in such a way that two distinct values rarely end up with the same index.

```
// expre_reinterpret_cast_Operator.cpp
// compile with: /EHsc
#include <iostream>

// Returns a hash code based on an address
unsigned short Hash( void *p ) {
   unsigned int val = reinterpret_cast<unsigned int>( p );
   return ( unsigned short )( val ^ (val >> 16));
}

using namespace std;
int main() {
   int a[20];
   for ( int i = 0; i < 20; i++ )
      cout << Hash( a + i ) << endl;
}
```

The **reinterpret_cast** allows the pointer to be treated as an integral type. The result is then bit-shifted and XORed with itself to produce a unique index (unique to a high degree of probability). The index is then truncated by a standard C-style cast to the return type of the function.

See Also

Reference

Casting Operators

C++ Keywords

Run-Time Type Information

Run-time type information (RTTI) is a mechanism that allows the type of an object to be determined during program execution. RTTI was added to the C++ language because many vendors of class libraries were implementing this functionality themselves. This caused incompatibilities between libraries. Thus, it became obvious that support for run-time type information was needed at the language level.

For the sake of clarity, this discussion of RTTI is almost completely restricted to pointers. However, the concepts discussed also apply to references.

There are three main C++ language elements to run-time type information:

- The dynamic_cast operator.

  Used for conversion of polymorphic types.

- The typeid operator.

  Used for identifying the exact type of an object.

- The type_info class.

  Used to hold the type information returned by the **typeid** operator.

See Also
Reference
Casting

typeid Operator

```
typeid( type-id )
typeid( expression )
```

Remarks

The **typeid** operator allows the type of an object to be determined at run time.

The result of **typeid** is a **const type_info&**. The value is a reference to a **type_info** object that represents either the *type-id* or the type of the *expression*, depending on which form of **typeid** is used. See type_info Class for more information.

The **typeid** operator does not work with managed types (abstract declarators or instances), see typeid for information on getting the Type of a specified type.

The **typeid** operator does a run-time check when applied to an l-value of a polymorphic class type, where the true type of the object cannot be determined by the static information provided. Such cases are:

- A reference to a class
- A pointer, dereferenced with *
- A subscripted pointer (i.e. [ ]). (Note that it is generally not safe to use a subscript with a pointer to a polymorphic type.)

If the *expression* points to a base class type, yet the object is actually of a type derived from that base class, a **type_info** reference for the derived class is the result. The *expression* must point to a polymorphic type (a class with virtual functions). Otherwise, the result is the **type_info** for the static class referred to in the *expression*. Further, the pointer must be dereferenced so that the object it points to is used. Without dereferencing the pointer, the result will be the **type_info** for the pointer, not what it points to. For example:

```
// expre_typeid_Operator.cpp
// compile with: /GR /EHsc
#include <iostream>
#include <typeinfo.h>

class Base {
public:
   virtual void vvfunc() {}
};

class Derived : public Base {};

using namespace std;
int main() {
   Derived* pd = new Derived;
   Base* pb = pd;
   cout << typeid( pb ).name() << endl;   //prints "class Base *"
   cout << typeid( *pb ).name() << endl;   //prints "class Derived"
   cout << typeid( pd ).name() << endl;   //prints "class Derived *"
   cout << typeid( *pd ).name() << endl;   //prints "class Derived"
   delete pd;
}
```

If the *expression* is dereferencing a pointer, and that pointer's value is zero, **typeid** throws a bad_typeid exception. If the pointer does not point to a valid object, a **__non_rtti_object** exception is thrown, indicating an attempt to analyze the RTTI that triggered a fault (like access violation), because the object is somehow invalid (bad pointer or the code wasn't compiled with /GR).

If the *expression* is neither a pointer nor a reference to a base class of the object, the result is a **type_info** reference representing the static type of the *expression*. The *static type* of an expression refers to the type of an expression as it is known at compile time. Execution semantics are ignored when evaluating the static type of an expression. Furthermore, references are ignored when possible when determining the static type of an expression:

```
// expre_typeid_Operator_2.cpp
#include <typeinfo>

int main()
{
   typeid(int) == typeid(int&); // evaluates to true
}
```

typeid can also be used in templates to determine the type of a template parameter:

```
// expre_typeid_Operator_3.cpp
// compile with: /c
#include <typeinfo>
template < typename T >
T max( T arg1, T arg2 ) {
   cout << typeid( T ).name() << "s compared." << endl;
   return ( arg1 > arg2 ? arg1 : arg2 );
}
```

See Also

Reference

Run-Time Type Information

C++ Keywords

bad_typeid Exception

The **bad_typeid** exception is thrown by the typeid operator when the operand for **typeid** is a NULL pointer.

```
catch (bad_typeid)
      statement
```

Remarks

The interface for **bad_typeid** is:

```
class bad_typeid : public exception
{
public:
   bad_typeid(const char * _Message = "bad typeid");
   bad_typeid(const bad_typeid &);
   virtual ~bad_typeid();
};
```

The following example shows the **typeid** operator throwing a **bad_typeid** exception.

```
// expre_bad_typeid.cpp
// compile with: /EHsc /GR
#include <typeinfo.h>
#include <iostream>

class A{
public:
   // object for class needs vtable
   // for RTTI
   virtual ~A();
};

using namespace std;
int main() {
A* a = NULL;

try {
   cout << typeid(*a).name() << endl;  // Error condition
   }
catch (bad_typeid){
   cout << "Object is NULL" << endl;
   }
}
```

Output

```
Object is NULL
```

See Also

Reference

Run-Time Type Information

C++ Keywords

type_info Class

The **type_info** class describes type information generated within the program by the compiler. Objects of this class effectively store a pointer to a name for the type. The **type_info** class also stores an encoded value suitable for comparing two types for equality or collating order. The encoding rules and collating sequence for types are unspecified and may differ between programs.

The `typeinfo.h` header file must be included in order to use the **type_info** class. The interface for the **type_info** class is:

```
class type_info {
public:
    _CRTIMP virtual ~type_info();
    _CRTIMP bool operator==(const type_info& rhs) const;
    _CRTIMP bool operator!=(const type_info& rhs) const;
    _CRTIMP int before(const type_info& rhs) const;
    _CRTIMP const char* name() const;
    _CRTIMP const char* raw_name() const;
};
```

You cannot instantiate objects of the **type_info** class directly, because the class has only a private copy constructor. The only way to construct a (temporary) **type_info** object is to use the typeid operator. Since the assignment operator is also private, you cannot copy or assign objects of class **type_info**.

The operators **==** and **!=** can be used to compare for equality and inequality with other **type_info** objects, respectively.

There is no link between the collating order of types and inheritance relationships. Use the **type_info::before** member function to determine the collating sequence of types. There is no guarantee that **type_info::before** will yield the same result in different programs or even different runs of the same program. In this manner, **type_info::before** is similar to the address-of **(&)** operator.

The **type_info::name** member function returns a **const char*** to a null-terminated string representing the human-readable name of the type. The memory pointed to is cached and should never be directly deallocated.

The **type_info::raw_name** member function returns a **const char*** to a null-terminated string representing the decorated name of the object type. The name is actually stored in its decorated form to save space. Consequently, this function is faster than **type_info::name** because it doesn't need to undecorate the name. The string returned by the **type_info::raw_name** function is useful in comparison operations but is not readable. If you need a human-readable string, use the **type_info::name** function instead.

Type information is generated for polymorphic classes only if the /GR (Enable Run-Time Type Information) compiler option is specified.

See Also

Reference

Run-Time Type Information

Statements (C++)

C++ statements are the program elements that control how and in what order objects are manipulated. This section includes:

- Overview
- Labeled Statements
- Categories of Statements
  - Expression statements. These statements evaluate an expression for its side effects or for its return value.
  - Null statements. These statements can be provided where a statement is required by the C++ syntax but where no action is to be taken.
  - Compound statements. These statements are groups of statements enclosed in curly braces ({ }). They can be used wherever a single statement may be used.
  - Selection statements. These statements perform a test; they then execute one section of code if the test evaluates to true (nonzero). They may execute another section of code if the test evaluates to false.
  - Iteration statements. These statements provide for repeated execution of a block of code until a specified termination criterion is met.
  - Jump statements. These statements either transfer control immediately to another location in the function or return control from the function.
  - Declaration statements. Declarations introduce a name into a program. (Declarations provides more detailed information about declarations.)

For information on exception handling statements see Exception Handling.

See Also

Other Resources

C++ Language Reference

Overview of C++ Statements

C++ statements are executed sequentially, except when an expression statement, a selection statement, an iteration statement, or a jump statement specifically modifies that sequence.

Statements may be of the following types:

```
        labeled-statement
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement
declaration-statement
try-throw-catch
```

In most cases, the C++ statement syntax is identical to that of ANSI C. The primary difference between the two is that in C, declarations are allowed only at the start of a block; C++ adds the *declaration-statement*, which effectively removes this restriction. This enables you to introduce variables at a point in the program where a precomputed initialization value can be calculated.

Declaring variables inside blocks also allows you to exercise precise control over the scope and lifetime of those variables.

The topics on statements describe the following C++ keywords:

break	else	__if_exists	__try
case	__except	__if_not_exists	try
catch	for	__leave	while
continue	goto	return	
default	__finally	switch	
do	if	throw	

See Also

Reference

Statements (C++)

Labeled Statements

To transfer program control directly to a given statement, the statement must be labeled. See Using Labels with the goto Statement and Using Labels in the case Statement.

Remarks

There are three types of labeled statements. All use a colon to separate some type of label from the statement. The case and default labels are specific to case (or switch) statements:

```
        identifier
        :
        statement
case constant-expression : statement
default : statement
```

See Also

Reference

Overview of C++ Statements

The switch Statement

The switch Statement

Using Labels with the goto Statement

The appearance of an *identifier* label in the source program declares a label. Only a goto statement can transfer control to an *identifier* label. The following code fragment illustrates use of the **goto** statement and an *identifier* label:

Remarks

A label cannot appear by itself but must always be attached to a statement. If a label is needed by itself, place a null statement after the label.

The label has function scope and cannot be redeclared within the function. However, the same name can be used as a label in different functions.

Example

```
// labels_with_goto.cpp
// compile with: /EHsc
#include <iostream>
int main() {
   using namespace std;
   goto Test2;

   cout << "testing" << endl;

   Test2:
      cerr << "At Test2 label." << endl;
}
```

Output

```
At Test2 label.
```

See Also

Reference

Labeled Statements

Using Labels in the case Statement

Labels that appear after the **case** keyword cannot also appear outside a **switch** statement. (This restriction also applies to the **default** keyword.) The following code fragment shows the correct use of **case** labels:

```
// Sample Microsoft Windows message processing loop.
switch( msg )
{
   case WM_TIMER:    // Process timer event.
      SetClassWord( hWnd, GCW_HICON, ahIcon[nIcon++] );
      ShowWindow( hWnd, SW_SHOWNA );
      nIcon %= 14;
      Yield();
      break;

   case WM_PAINT:
      // Obtain a handle to the device context.
      // BeginPaint will send WM_ERASEBKGND if appropriate.

      memset( &ps, 0x00, sizeof(PAINTSTRUCT) );
      hDC = BeginPaint( hWnd, &ps );

      // Inform Windows that painting is complete.

      EndPaint( hWnd, &ps );
      break;

   case WM_CLOSE:
      // Close this window and all child windows.

      KillTimer( hWnd, TIMER1 );
      DestroyWindow( hWnd );
      if ( hWnd == hWndMain )
         PostQuitMessage( 0 );  // Quit the application.
      break;

   default:
      // This choice is taken for all messages not specifically
      //  covered by a case statement.

      return DefWindowProc( hWnd, Message, wParam, lParam );
      break;
}
```

See Also

Reference

Labeled Statements

Expression Statement

Expression statements cause expressions to be evaluated. No transfer of control or iteration takes place as a result of an expression statement.

The syntax for the expression statement is simply

```
[expression ] ;
```

Remarks

All expressions in an expression statement are evaluated and all side effects are completed before the next statement is executed. The most common expression statements are assignments and function calls. Since the expression is optional, a semicolon alone is considered an empty expression statement, referred to as the null statement.

See Also

Reference

Overview of C++ Statements

The Null Statement

The "null statement" is an expression statement with the *expression* missing. It is useful when the syntax of the language calls for a statement but no expression evaluation. It consists of a semicolon.

Null statements are commonly used as placeholders in iteration statements or as statements on which to place labels at the end of compound statements or functions.

The following code fragment shows how to copy one string to another and incorporates the null statement:

```
// null_statement.cpp
char *myStrCpy( char *Dest, const char *Source )
{
    char *DestStart = Dest;

    // Assign value pointed to by Source to
    // Dest until the end-of-string 0 is
    // encountered.
    while( *Dest++ = *Source++ )
        ;   // Null statement.

    return DestStart;
}

int main()
{
}
```

See Also

Reference

Expression Statement

Compound Statements (Blocks)

A compound statement consists of zero or more statements enclosed in curly braces (**{ }**). A compound statement can be used anywhere a statement is expected. Compound statements are commonly called "blocks."

```
{ [ statement-list ] }
```

Remarks

The following example uses a compound statement as the *statement* part of the **if** statement (see The if Statement for details about the syntax):

```
if( Amount > 100 )
{
    cout << "Amount was too large to handle\n";
    Alert();
}
else
    Balance -= Amount;
```

Note

Because a declaration is a statement, a declaration can be one of the statements in the *statement-list*. As a result, names declared inside a compound statement, but not explicitly declared as static, have local scope and (for objects) lifetime. See Scope for details about treatment of names with local scope.

See Also

Reference

Overview of C++ Statements

Selection Statements (C++)

The C++ selection statements, if and switch, provide a means to conditionally execute sections of code.

The __if_exists and __if_not_exists statements allow you to conditionally include code depending on the existence of a symbol.

See the individual topics for the syntax for each statement.

See Also

Reference

Overview of C++ Statements

The if-else Statement

Controls conditional branching.

```
if ( expression )
      statement1
[else
      statement2]
```

Remarks

If the value of *expression* is nonzero, *statement1* is executed. If the optional **else** is present, *statement2* is executed if the value of *expression* is zero. *expression* must be of arithmetic or pointer type, or it must be of a class type that defines an unambiguous conversion to an arithmetic or pointer type. (For information about conversions, see Standard Conversions.)

In both forms of the **if** statement, *expression*, which can have any value except a structure, is evaluated, including all side effects. Control passes from the **if** statement to the next statement in the program unless one of the *statements* contains a break, continue, or goto.

The **else** clause of an **if...else** statement is associated with the closest previous **if** statement in the same scope that does not have a corresponding **else** statement.

For this sample to be unambiguous about **if...else** pairing, uncomment the braces.

Example

```
// if_else_statement.cpp
#include <stdio.h>

int main() {
   int x = 0;
   if ( 1 )   // if statement #1
   // {
      if ( !x )   // if statement #2
         printf_s("!x\n");
      else   // paired with if statement #2
         printf_s("x\n");
   // }
}
```

Output

```
!x
```

See Also

Reference

Selection Statements (C++)
C++ Keywords
The switch Statement

The __if_exists Statement

__if_exists allows you to conditionally include code depending on whether the specified symbol exists.

```
__if_exists ( variable ) {
statements
}
```

where:

variable

The symbol whose existence you want to test for.

statements

One or more statements to execute if *variable* exists.

Remarks

__if_exists can be used to test for the existence of both members and non-members identifiers and can be used to test for the existence of an overloaded function but not for a specific form of the overload.

__if_exists should only be used within the body of a function. Outside of the body of a function **__if_exists** can only refer to fully defined types. The Visual C++ compiler must know what members a class template has before any instantiations.

__if_not_exists allows you to conditionally include code depending on whether the specified symbol does not exist.

Example

```
// the__if_exists_statement.cpp
// compile with: /EHsc
#include <iostream>

template<typename T>
class X : public T {
public:
   void Dump() {
      std::cout << "In X<T>::Dump()" << std::endl;

      __if_exists(T::Dump) {
         T::Dump();
      }

      __if_not_exists(T::Dump) {
         std::cout << "T::Dump does not exist" << std::endl;
      }
   }
};

class A {
public:
   void Dump() {
      std::cout << "In A::Dump()" << std::endl;
   }
};

class B {};

bool g_bFlag = true;

class C {
public:
   void f(int);
   void f(double);
};
```

```
int main() {
   X<A> x1;
   X<B> x2;

   x1.Dump();
   x2.Dump();

   __if_exists(::g_bFlag) {
      std::cout << "g_bFlag = " << g_bFlag << std::endl;
   }

   __if_exists(C::f) {
      std::cout << "C::f exists" << std::endl;
   }

   return 0;
}
```

Output

```
In X<T>::Dump()
In A::Dump()
In X<T>::Dump()
T::Dump does not exist
g_bFlag = 1
C::f exists
```

See Also

Reference

Selection Statements (C++)

C++ Keywords

The __if_not_exists Statement

__if_not_exists allows you to conditionally include code depending on whether the specified symbol does not exist.

```
        __if_not_exist ( variable ) {
statements
}
```

where:

variable

The symbol whose existence you want to test for.

statements

One or more statements to execute if *variable* does not exists.

Remarks

__if_not_exists can be applied to identifiers both inside of and outside of a class. When testing for overload functions, you cannot test for a specific form of the overload.

__if_not_exists should only be used within the body of a function. Outside of the body of a function **__if_not_exists** can only refer to fully defined types.

__if_exists allows you to conditionally include code depending on whether the specified symbol exists.

Example

See __if_exists for an example of how to use **__if_not_exists**.

See Also

Reference

Selection Statements (C++)

C++ Keywords

The switch Statement

Allows selection among multiple sections of code, depending on the value of an integral expression.

```
switch ( expression )
      case constant-expression : statement
   [default  : statement]
```

Remarks

The *expression* must be of an integral type or of a class type for which there is an unambiguous conversion to integral type. Integral promotion is performed as described in Integral Promotions.

The **switch** statement body consists of a series of **case** labels and an optional **default** label. No two constant expressions in **case** statements can evaluate to the same value. The **default** label can appear only once. The labeled statements are not syntactic requirements, but the **switch** statement is meaningless without them. The default statement need not come at the end; it can appear anywhere in the body of the switch statement. A case or default label can only appear inside a switch statement.

The *constant-expression* in each **case** label is converted to the type of *expression* and compared with *expression* for equality. Control passes to the statement whose **case** *constant-expression* matches the value of *expression*. The resulting behavior is shown in the following table.

Switch Statement Behavior

Condition	Action
Converted value matches that of the promoted controlling expression.	Control is transferred to the statement following that label.
None of the constants match the constants in the **case** labels; a **default** label is present.	Control is transferred to the **default** label.
None of the constants match the constants in the **case** labels; **default** label is not present.	Control is transferred to the statement after the **switch** statement.

If a matching expression is found, control is not impeded by subsequent **case** or **default** labels. The break statement is used to stop execution and transfer control to the statement after the **switch** statement. Without a **break** statement, every statement from the matched **case** label to the end of the **switch**, including the **default**, is executed. For example:

```
// switch_statement1.cpp
#include <stdio.h>

int main() {
   char *buffer = "Any character stream";
   int capa, lettera, nota;
   char c;
   capa = lettera = nota = 0;

   while ( c = *buffer++ )   // Walks buffer until NULL
   {
      switch ( c )
      {
         case 'A':
            capa++;
            break;
         case 'a':
            lettera++;
            break;
         default:
            nota++;
      }
```

```
    }
    printf_s( "\nUppercase a: %d\nLowercase a: %d\nTotal: %d\n",
       capa, lettera, (capa + lettera + nota) );
}
```

In the above example, `capa` is incremented if `c` is an uppercase `A`. The `break` statement after `capa++` terminates execution of the `switch` statement body and control passes to the `while` loop. Without the `break` statement, `lettera` and `nota` would also be incremented. A similar purpose is served by the `break` statement for `case 'a'`. If `c` is a lowercase `a`, `lettera` is incremented and the `break` statement terminates the `switch` statement body. If `c` is not an `a` or `A`, the `default` statement is executed.

An inner block of a **switch** statement can contain definitions with initializations as long as they are reachable — that is, not bypassed by all possible execution paths. Names introduced using these declarations have local scope. For example:

```
// switch_statement2.cpp
// C2360 expected
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
   switch( tolower( *argv[1] ) )
   {
       // Error. Unreachable declaration.
       char szChEntered[] = "Character entered was: ";

   case 'a' :
       {
       // Declaration of szChEntered OK. Local scope.
       char szChEntered[] = "Character entered was: ";
       cout << szChEntered << "a\n";
       }
       break;

   case 'b' :
       // Value of szChEntered undefined.
       cout << szChEntered << "b\n";
       break;

   default:
       // Value of szChEntered undefined.
       cout << szChEntered << "neither a nor b\n";
       break;
   }
}
```

A **switch** statement can be nested. In such cases, **case** or **default** labels associate with the closest **switch** statement that encloses them.

Microsoft Specific

Microsoft C does not limit the number of case values in a **switch** statement. The number is limited only by the available memory. ANSI C requires at least 257 case labels be allowed in a **switch** statement.

The default for Microsoft C is that the Microsoft extensions are enabled. Use the /Za compiler option to disable these extensions.

END Microsoft Specific

See Also

Reference

Selection Statements (C++)

C++ Keywords

Using Labels in the case Statement

Iteration Statements (C++)

Iteration statements cause statements (or compound statements) to be executed zero or more times, subject to some loop-termination criteria. When these statements are compound statements, they are executed in order, except when either the break statement or the continue statement is encountered.

C++ provides three iteration statements — while, do, and for. Each of these iterates until its termination expression evaluates to zero (false), or until loop termination is forced with a **break** statement. The following table summarizes these statements and their actions; each is discussed in detail in the sections that follow.

Iteration Statements

Statement	Evaluated At	Initialization	Increment
while	Top of loop	No	No
do	Bottom of loop	No	No
for	Top of loop	Yes	Yes

The statement part of an iteration statement cannot be a declaration. However, it can be a compound statement containing a declaration.

See Also

Reference

Overview of C++ Statements

The while Statement

Executes *statement* repeatedly until *expression* evaluates to zero.

```
while ( expression )
      statement
```

Remarks

The test of *expression* takes place before each execution of the loop; therefore, a **while** loop executes zero or more times. *expression* must be of an integral type, a pointer type, or a class type with an unambiguous conversion to an integral or pointer type.

A **while** loop can also terminate when a break, goto, or return within the statement body is executed. Use continue to terminate the current iteration without exiting the **while** loop. **continue** passes control to the next iteration of the **while** loop.

The following code uses a **while** loop to trim trailing underscores from a string:

```
// while_statement.cpp

#include <string.h>
#include <stdio.h>
char *trim( char *szSource )
{
    char *pszEOS = 0;

    //  Set pointer to character before terminating NULL
    pszEOS = szSource + strlen( szSource ) - 1;

    //  iterate backwards until non '_' is found
    while( (pszEOS >= szSource) && (*pszEOS == '_') )
        *pszEOS-- = '\0';

    return szSource;
}
int main()
{
    char szbuf[] = "12345_____";

    printf_s("\nBefore trim: %s", szbuf);
    printf_s("\nAfter trim: %s\n", trim(szbuf));
}
```

The termination condition is evaluated at the top of the loop. If there are no trailing underscores, the loop never executes.

See Also

Reference

Iteration Statements (C++)
C++ Keywords
The do-while Statement (C++)
The for Statement

The do-while Statement (C++)

Executes a *statement* repeatedly until the specified termination condition (the *expression*) evaluates to zero.

```
do
      statement
      while ( expression ) ;
```

Remarks

The test of the termination condition is made after each execution of the loop; therefore, a **do-while** loop executes one or more times, depending on the value of the termination expression. The **do-while** statement can also terminate when a break, goto, or return statement is executed within the statement body.

The *expression* must have arithmetic or pointer type. Execution proceeds as follows:

1. The statement body is executed.
2. Next, *expression* is evaluated. If *expression* is false, the **do-while** statement terminates and control passes to the next statement in the program. If *expression* is true (nonzero), the process is repeated, beginning with step 1.

Example

The following sample demonstrates the **do-while** statement:

```
// do_while_statement.cpp
#include <stdio.h>
int main()
{
    int i = 0;
    do
    {
        printf_s("\n%d",i++);
    } while (i < 3);
}
```

See Also

Reference

Iteration Statements (C++)
C++ Keywords
The while Statement

The for Statement

Executes *statement* and *loop-expression* repeatedly until *cond-expression* becomes false.

```
for ( init-expression ; cond-expression ; loop-expression )
      statement
```

Remarks

Use the **for** statement to construct loops that must execute a specified number of times.

The **for** statement consists of three optional parts, as shown in the following table.

for Loop Elements

Syntax Name	When Executed	Contents
init-expression	Before any other element of the **for** statement. *init-expression* is executed only once. Control then passes to *cond-expression*.	Often used to initialize loop indices. It can contain expressions or declarations (C++ only).
cond-expression	Before execution of each iteration of *statement*, including the first iteration. *statement* is executed only if *cond-expression* evaluates to true (nonzero).	An expression that evaluates to an integral type or a class type that has an unambiguous conversion to an integral type. Normally used to test for loop-termination criteria.
loop-expression	At the end of each iteration of *statement*. After *loop-expression* is executed, *cond-expression* is evaluated.	Normally used to increment loop indices.

For example:

```
// for_statement1.cpp
#include <stdio.h>
int main() {
int i;
for ( i = 0 ; i < 2 ; i++ )
   printf_s( "\n%d\n",i );
}
```

The preceding **for** loop is equivalent to the following **while** loop:

```
// for_statement2.cpp
#include <stdio.h>
int main()
{
   int i = 0;
   while ( i < 2 )
      printf_s( "%d\n", i++ );
}
```

init-expression and *loop-expression* can contain multiple statements separated by the comma operator. For example:

```
// for_statment3.cpp
#include <stdio.h>
int main()
{
   int i, j;
   for ( i = 5, j = 10 ; i + j < 20; i++, j++ )
```

```
        printf_s( "\n i + j = %d", (i + j) );
}
```

A **for** loop terminates when a break, return, or goto (to a labeled statement outside the **for** loop) within *statement* is executed. A continue statement in a **for** loop terminates only the current iteration.

If *cond-expression* is omitted, it is considered true and the **for** loop will not terminate without a **break**, **return**, or **goto** within *statement*.

A convenient way to specify an infinite loop using the **for** statement is:

```
for( ; ; )
{
    // Statements to be executed.
}
```

Although the three fields of the **for** statement are normally used for initialization, testing for termination, and incrementing, they are not restricted to these uses. For example, the following code prints the numbers 1 to 5. In this case, *statement* is the null statement:

```
// for_statement4.cpp
#include <stdio.h>
int main()
{
    int i;
    for( i = 0; i < 5; printf_s("%d\n", i), i++)
        ;
}
```

for Loops and the C++ Standard

The C++ standard says that a variable declared in a **for** loop shall go out of scope after the **for** loop ends. For example:

```
for (int i = 0 ; i < 5 ; i++) {
   // do something
}
// i is now out of scope under /Za or /Zc:forScope
```

By default, under /Ze, a variable declared in a **for** loop remains in scope until the **for** loop's enclosing scope ends.

/Zc:forScope enables standard behavior of variables declared in for loops without needing to specify /Za.

It is also possible to use the scoping differences of the **for** loop to redeclare variables under /Ze as follows:

```
// for_statement5.cpp
int main(){
   int i = 0;   // hidden by var with same name declared in for loop
   for ( int i = 0 ; i < 3; i++ ) {}

   for ( int i = 0 ; i < 3; i++ ) {}
}
```

This more closely mimics the standard behavior of a variable declared in a **for** loop, which requires variables declared in a **for** loop to go out of scope after the loop is done. When a variable is declared in a **for** loop, the compiler internally promotes it to a local variable in the **for** loop's enclosing scope even if there is already a local variable with the same name.

See Also

Reference

Iteration Statements (C++)
C++ Keywords
The while Statement
The do-while Statement (C++)

Jump Statements (C++)

The C++ jump statements perform an immediate local transfer of control.

```
<A HREF="_pluslang_the_c.2b2b_.break_statement.htm">break</A>   ;
<A HREF="_pluslang_the_c.2b2b_.continue_statement.htm">continue</A>   ;
<A HREF="_pluslang_the_c.2b2b_.return_statement.htm">return</A> [expression]<SUB> </SUB>;
<A HREF="_pluslang_the_goto_statement.htm">goto</A> identifier   ;
```

See Also

Reference

Overview of C++ Statements

The break Statement

The **break** statement terminates the execution of the nearest enclosing loop or conditional statement in which it appears. Control passes to the statement that follows the terminated statement, if any.

```
break;
```

Remarks

break is used with the conditional switch statement and with the do, for, and while loop statements.

In a **switch** statement, **break** causes the program to execute the next statement after the **switch**. Without a **break** statement, every statement from the matched **case** label to the end of the **switch**, including the **default**, is executed.

In loops, **break** terminates execution of the nearest enclosing **do**, **for**, or **while** statement. Control passes to the statement that follows the terminated statement, if any.

Within nested statements, the **break** statement terminates only the **do**, **for**, **switch**, or **while** statement that immediately encloses it. You can use a **return** or **goto** statement to transfer control from within more deeply nested structures.

Example

The following example illustrates the use of the **break** statement in a **for** loop.

```
// break_statement.cpp

#include <stdio.h>

int main()
{
    int i;

    for (i = 1; i < 10; i++)
    {
        printf_s("%d\n", i);

        if (i == 4)
            break;
    }
}  // Loop exits after printing 1 through 4
```

Output

```
1
2
3
4
```

See Also

Reference

Jump Statements (C++)
C++ Keywords
The continue Statement

The continue Statement

Forces transfer of control to the controlling expression of the smallest enclosing do, for, or while loop.

```
continue;
```

Remarks

Any remaining statements in the current iteration are not executed. The next iteration of the loop is determined as follows:

- In a **do** or **while** loop, the next iteration starts by reevaluating the controlling expression of the **do** or **while** statement.
- In a **for** loop (using the syntax **for**(*init-expr*; *cond-expr*; *loop-expr*)), **continue** causes *loop-expr* to be executed. Then *cond-expr* is reevaluated and, depending on the result, the loop either terminates or another iteration occurs.

The following example shows how the **continue** statement can be used to bypass sections of code and begin the next iteration of a loop.

Example

```
// continue_statement.cpp
#include <stdio.h>
int main()
{
    int i = 0;
    do
    {
        i++;
        printf_s("before the continue\n");
        continue;
        printf("after the continue, should never print\n");
     } while (i < 3);

     printf_s("after the do loop\n");
}
```

Output

```
before the continue
before the continue
before the continue
after the do loop
```

See Also

Reference

Jump Statements (C++)

C++ Keywords

The return Statement

Terminates the execution of a function and returns control to the calling function (or, in the case of the main function, transfers control back to the operating system). Execution resumes in the calling function at the point immediately following the call.

```
return [expression]
```

Remarks

The value of *expression*, if present, is returned to the calling function. If *expression* is omitted, the return value of the function is undefined. Functions of type **void**, constructors, and destructors cannot specify expressions in the **return** statement; functions of all other types must specify an expression in the **return** statement.

The expression, if specified, is converted to the type specified in the function declaration, as if an initialization were being performed. Conversion from the type of the expression to the **return** type of the function can cause temporary objects to be created. See Temporary Objects for more information about how and when temporaries are created.

When the flow of control exits the block enclosing the function definition, the result is the same as it would be if a **return** statement with no expression had been executed. This is illegal for functions that are declared as returning a value.

A function can have any number of **return** statements.

The following example uses an expression with **return** to obtain the largest of two integers.

Example

```
// return_statement2.cpp
#include <stdio.h>

int max ( int a, int b )
{
   return ( a > b ? a : b );
}

int main()
{
    int nOne = 5;
    int nTwo = 7;

    printf_s("\n%d is bigger\n", max( nOne, nTwo ));
}
```

See Also

Reference

Jump Statements (C++)

C++ Keywords

The goto Statement

Performs an unconditional transfer of control to the named label. The

```
goto identifier ;
```

Remarks

label must be in the current function.

A statement label is meaningful only to a **goto** statement; in any other context, a labeled statement is executed without regard to the label.

A *jump-statement* must reside in the same function and can appear before only one statement in the same function. The set of *identifier* names following a **goto** has its own name space so the names do not interfere with other identifiers. Labels cannot be redeclared.

It is good programming style to use the **break**, **continue**, and **return** statement in preference to **goto** whenever possible. Since the **break** statement only exits from one level of the loop, a **goto** may be necessary for exiting a loop from within a deeply nested loop.

For more information about labels and the **goto** statement, see Labeled Statements and Using Labels with the goto Statement.

Example

In this example, a **goto** statement transfers control to the point labeled `stop` when `i` equals 3.

```
// goto_statement.cpp
#include <stdio.h>
int main()
{
    int i, j;

    for ( i = 0; i < 10; i++ )
    {
        printf_s( "Outer loop executing. i = %d\n", i );
        for ( j = 0; j < 2; j++ )
        {
            printf_s( " Inner loop executing. j = %d\n", j );
            if ( i == 3 )
                goto stop;
        }
    }

    // This message does not print:
    printf_s( "Loop exited. i = %d\n", i );

    stop:
    printf_s( "Jumped to stop. i = %d\n", i );
}
```

Output

```
Outer loop executing. i = 0
 Inner loop executing. j = 0
 Inner loop executing. j = 1
Outer loop executing. i = 1
 Inner loop executing. j = 0
 Inner loop executing. j = 1
Outer loop executing. i = 2
 Inner loop executing. j = 0
 Inner loop executing. j = 1
Outer loop executing. i = 3
 Inner loop executing. j = 0
Jumped to stop. i = 3
```

See Also

Reference

Jump Statements (C++)

C++ Keywords

Declaration Statements (C++)

Declaration statements introduce new names into the current scope. These names can be:

- Type names (**class**, **struct**, **union**, **enum**, **typedef**, and pointer-to-member).
- Object names.
- Function names.

Declarations may be any of the following:

```
        block-declaration
function-definition
template-declaration
explicit-instantiation
explicit-specialization
linkage-specification
namespace-definition
```

If a declaration within a block introduces a name that is already declared outside the block, the previous declaration is hidden for the duration of the block. After termination of the block, the previous declaration is again visible.

Multiple declarations of the same name in the same block are illegal.

For more information about declarations and name hiding, see Declarations and Definitions and Scope.

What do you want to know more about?

- Declaration of Automatic Objects
- Declaration of Static Objects

See Also

Reference

Overview of C++ Statements

Declaration of Automatic Objects

In C++, objects can be declared with automatic storage class using the **auto** or **register** keyword. If no storage-class keyword is used for a local object (an object declared inside a function), **auto** is assumed. C++ initializes and declares these objects differently than objects declared with static storage classes.

See Also

Reference

Declaration Statements (C++)

Initialization of Automatic Objects

Each time declaration statements for objects of storage class **auto** or **register** are executed, initialization takes place. The following example, from The continue Statement, shows initialization of the automatic object `ch` inside the `do` loop.

```
// initialization_of_automated_objects.cpp
#include <conio.h>
#include <string.h>

// Get a character that is a member of the zero-terminated
//  string, szLegalString. Return the index of the character
//  entered.
int GetLegalChar( char *szLegalString )
{
    char *pch;

    do
    {
        // This declaration statement is executed once for each
        //  execution of the loop.
        char ch = _getch();

        if( (pch = strchr( szLegalString, ch )) == NULL )
            continue;

        // A character that was in the string szLegalString
        //  was entered. Return its index.
        return (pch - szLegalString);
    } while( 1 );
}
int main()
{
}
```

For each iteration of the loop (each time the declaration is encountered), the macro `_getch` is evaluated and `ch` is initialized with the results. When control is transferred outside the block using the `return` statement, `ch` is destroyed (in this case, the storage is deallocated).

See Storage Classes for another example of initialization.

See Also

Reference

Declaration of Automatic Objects

Destruction of Automatic Objects

Objects defined in a loop are destroyed once per iteration of the loop, on exit from the block, or when control transfers to a point prior to the declaration. Objects declared in a block that is not a loop are destroyed on exit from the block or when control transfers to a point prior to the declaration.

Note
Destruction can mean simply deallocating the object or, for class-type objects, invoking the object's destructor.

When a jump statement transfers control out of a loop or block, objects declared in the block transferred from are destroyed; objects in the block transferred to are not destroyed.

When control is transferred to a point prior to a declaration, the object is destroyed.

See Also

Reference

Declaration of Automatic Objects

Transfers of Control

You can use the **goto** statement or a **case** label in a **switch** statement to specify a program that branches past an initializer. Such code is illegal unless the declaration that contains the initializer is in a block enclosed by the block in which the jump statement occurs.

The following example shows a loop that declares and initializes the objects `total`, `ch`, and `i`. There is also an erroneous `goto` statement that transfers control past an initializer.

```
// transfers_of_control.cpp
// compile with: /W1
// Read input until a nonnumeric character is entered.
int main()
{
   char MyArray[5] = {'2','2','a','c'};
   int i = 0;
   while( 1 )
   {
      int total = 0;

      char ch = MyArray[i++];

      if ( ch >= '0' && ch <= '9' )
      {
         goto Label1;

         int i = ch - '0';
      Label1:
         total += i;   // C4700: transfers past initialization of i.
      } // i would be destroyed here if  goto error were not present
   else
      // Break statement transfers control out of loop,
      //  destroying total and ch.
      break;
   }
}
```

In the preceding example, the `goto` statement tries to transfer control past the initialization of `i`. However, if `i` were declared but not initialized, the transfer would be legal.

The objects `total` and `ch`, declared in the block that serves as the *statement* of the `while` statement, are destroyed when that block is exited using the `break` statement.

See Also

Reference

Declaration of Automatic Objects

Declaration of Static Objects

An object can be declared with static storage class using the **static** or **extern** keyword. Local objects must be explicitly declared as **static** or **extern** to have static storage class. All global objects (objects declared outside all functions) have static storage class. You cannot declare static instances in a tiny-model program.

See Also

Reference

Declaration Statements (C++)

Initialization of Static Objects

Global objects are initialized at program startup. (For more information about construction and destruction of global objects, see Additional Startup Considerations and Additional Termination Considerations.)

When a static local object is declared with an initializer that does not evaluate to a constant expression, the object is given the value 0 (converted to the appropriate type) at the point before execution enters the block for the first time. However, the object is not visible and no constructors are called until the actual point of declaration.

At the point of declaration, the object's constructor (if the object is of a class type) is called as expected. (Static local objects are only initialized the first time they are seen.)

Example

Local objects declared as **static** are initialized the first time their declarations are encountered in the program flow. The following class, introduced in Basic Concepts, shows how this works:

```
// initialization_of_static_objects.cpp
// compile with: /EHsc
// arguments: a
#include <iostream>
#include <string.h>

using namespace std;
// class that logs initializations and destructions.
class InitDemo {
public:
   InitDemo( char *szWhat );
   ~InitDemo();
private:
   char *szObjName;
};

InitDemo::InitDemo( char *szWhat ) {
   if ( szWhat != 0 && strlen( szWhat ) > 0 ) {
      szObjName = new char[ strlen( szWhat ) + 1 ];
      strcpy_s( szObjName, strlen( szWhat ) + 1, szWhat );
   }
   else
      szObjName = 0;

   clog << "Initializing: " << szObjName << "\n";
}

InitDemo::~InitDemo() {
   if ( szObjName != 0 ) {
      clog << "Destroying: " << szObjName << "\n";
      delete szObjName;
   }
}

int main( int argc, char *argv[] ) {
   if ( argc < 2 ) {
      cerr << "Supply a one-letter argument.\n";
      return -1;
   }

   if ( *argv[1] == 'a' ) {
      cout << "*argv[1] was an 'a'\n";
      // Declare static local object.
      static InitDemo I1( "static I1" );
   }
   else
      cout << "*argv[1] was not an 'a'\n";
}
```

Input

```
a
```

Output

```
*argv[1] was an 'a'
Initializing: static I1
Destroying: static I1
```

See Also

Reference

Declaration of Static Objects

Destruction of Static Objects

Local static objects are destroyed during termination specified by **atexit**.

If a static object was not constructed because the program's flow of control bypassed its declaration, no attempt is made to destroy that object.

See Also

Reference

Declaration of Static Objects

Declarations

Declarations introduce new names into a program. Topics covered in this section include the following uses for declarations:

- Specify storage-class, type, and linkage for an object or function.
- Declare a function as **inline** or **virtual**.
- Qualify a declaration as const or volatile.
- Associate a name with a constant (enumeration declaration).
- Declare a new type (**class**, **struct**, or **union** declaration).
- Specify a synonym for a type (typedef declaration).
- Specify a namespace.

In addition to introducing a new name, a declaration specifies how an identifier is to be interpreted by the compiler. Declarations do not automatically reserve storage associated with the identifier — reserving storage is done by definitions.

Note
Most declarations are also definitions. Declarations that are not definitions include class declarations without the member list , and function declarations without the function body.

A declaration may be one of:

```
[ decl-specifiers ] [ declarator-list ] ;
function-definition
linkage-specification
template-specification
explicit-template-instantiation
explicit-template-specialization
namespace-definition
namespace-alias-definition
using-declaration
using-directive
asm-definition
```

The declarators in *declarator-list* contain the names being declared. Although the *declarator-list* is shown as optional, it can be omitted only in declarations or definitions of a function.

Note
The declaration of a function is often called a "prototype." This declaration provides type information about arguments and t he function's return type that allows the compiler to perform correct conversions and to help provide type safety.

The *decl-specifiers* part of a declaration is also shown as optional; however, it can be omitted only in declarations of class types or enumerations.

Declarations occur in a scope. This controls the visibility of the name declared and the duration of the object defined (if any). For more information about how scope rules interact with declarations, see Scope.

An object declaration is also a definition unless it contains the **extern** storage-class specifier described in Storage-Class Specifiers. A function declaration is also a definition unless it is a prototype — a function header with no defining function body. An object's definition causes allocation of storage and appropriate initializations for that object.

For information on function definitions, see Function definitions.

For information on declaring linkage specifications, see linkage specifications.

For information on template specifications, explicit template instantiations and explicit template specializations, see Templates.

For information on namespace definitions and namespace alias definitions, see namespaces.

For information on asm definitions, see __asm.

See Also

Other Resources

C++ Language Reference

Specifiers

This section explains the *decl-specifiers* portion of declarations. (The syntax for declarations is given at the beginning of this section.)

The following are declaration specifiers:

```
        storage-class-specifier
type-specifier
function-specifier
friend
typedef
__declspec ( extended-decl-modifier-seq )
```

The Microsoft-specific keyword, **__declspec**, is discussed in Extended Attribute Syntax.

Remarks

The *decl-specifiers* portion of a declaration is the longest sequence of *decl-specifiers* that can be construed to be a type name, not including the pointer or reference modifiers. The remainder of the declaration is the declarator, including the name introduced. The examples in the following list illustrates this concept:

Declaration	***decl-specifiers***	***declarator***
`char *lpszAppName;`	`char`	`*lpszAppName`
`typedef char * LPSTR;`	`char`	`*LPSTR`
`const int func1();`	`const int`	`func1`
`volatile void *pvvObj;`	`volatile void`	`*pvvObj`

Because **signed**, **unsigned**, **long**, and **short** all imply **int**, a **typedef** name following one of these keywords is taken to be a member of *declarator-list,* not of *decl-specifiers*.

Note
Because a name can be redeclared, its interpretation is subject to the most recent declaration in the current scope. Redeclaration can affect how names are interpreted by the compiler, particularly **typedef** names.

See Also

Reference

Declarations

Storage-Class Specifiers

The C++ storage-class specifiers tell the compiler the duration and visibility of the object or function they declare, as well as where an object should be stored.

The following are storage class specifiers:

- auto
- register
- static
- extern

The mutable keyword may be considered a storage class specifier. However, it is only available in the member list of a class definition.

See Also
Reference
Specifiers

Automatic Storage-Class Specifiers

The **auto** and **register** storage-class specifiers can be used only to declare names used in blocks or to declare formal arguments to functions. The term "auto" comes from the fact that storage for these objects is automatically allocated at run time (normally on the program's stack).

See Also

Reference

Storage-Class Specifiers

auto Keyword

The **auto** storage-class specifier declares an automatic variable, a variable with a local lifetime. It is the default storage-class specifier for block-scoped variable declarations.

```
auto declarator ;
```

Remarks

An **auto** variable is visible only in the block in which it is declared. Declarations of **auto** variables can include initializers, as discussed in Initialization. Since variables with **auto** storage class are not initialized automatically, you should either explicitly initialize them when you declare them, or assign initial values to them in statements within the block. The values of uninitialized **auto** variables are undefined. (A local variable of **auto** or **register** storage class is initialized each time it comes in scope if an initializer is given.)

An internal **static** variable (a static variable with local or block scope) can be initialized with the address of any external or **static** item, but not with the address of another **auto** item, because the address of an **auto** item is not a constant.

Few programmers use the **auto** keyword in declarations because all block-scoped objects not explicitly declared with another storage class are implicitly automatic. Therefore, the following two declarations are equivalent:

```
// auto_keyword.cpp
int main()
{
   auto int i = 0;    // Explicitly declared as auto.
   int j = 0;    // Implicitly auto.
}
```

See Also

Reference

Storage-Class Specifiers
C++ Keywords
Using extern to Specify Linkage
register Keyword
Static (C++)

register Keyword

The **register** keyword specifies that the variable is to be stored in a machine register, if possible.

```
register declarator ;
```

Remarks

For example:

```
register int var;
```

Microsoft Specific

The compiler does not accept user requests for register variables; instead, it makes its own register choices when global register-allocation optimization (/Oe option) is on. However, all other semantics associated with the **register** keyword are honored.

END Microsoft Specific

ANSI C does not allow for taking the address of a register object; this restriction does not apply to C++. However, if the address-of operator (**&**) is used on an object, the compiler must put the object in a location for which an address can be represented. In practice, this means in memory instead of in a register.

See Also

Reference

Storage-Class Specifiers

C++ Keywords

Static Storage-Class Specifiers

The static storage-class specifiers, static and extern, can be applied to objects and functions. The following table shows where the keywords **static** and **extern** can and cannot be used.

Use of static and extern

Construct	Can static be Used?	Can extern be Used?
Function declarations within a block	No	Yes
Formal arguments to a function	No	No
Objects in a block	Yes	Yes
Objects outside a block	Yes	Yes
Functions	Yes	Yes
Class member functions	Yes	No
Class member data	Yes	No
typedef names	No	No

A name specified using the **static** keyword has internal linkage except for the static members of a class that have external linkage. That is, it is not visible outside the current translation unit. A name specified using the **extern** keyword has external linkage unless previously defined as having internal linkage. For more information about the visibility of names, see Scope and Program and Linkage.

Note
Functions that are declared as **inline** and that are not class member functions are given the same linkage characteristics as f unctions declared as **static**.

A class name whose declaration has not yet been encountered by the compiler can be used in an **extern** declaration. The name introduced with such a declaration cannot be used until the class declaration has been encountered.

See Also

Reference

Storage-Class Specifiers

Names Without Storage-Class Specifiers

File-scope names with no explicit storage-class specifiers have external linkage unless they are:

- Declared using the **const** keyword.
- Previously declared with internal linkage.

See Also

Reference

Storage-Class Specifiers

Function Specifiers

You can use the inline and virtual keywords as specifiers in function declarations. This use of **virtual** differs from its use in the base-class specifier of a class definition.

See Also

Reference

Specifiers

inline, __inline, __forceinline

The **inline** and **__inline** specifiers instruct the compiler to insert a copy of the function body into each place the function is called.

```
inline function_declarator;
__inline function_declarator;   // Microsoft Specific
__forceinline function_declarator;   // Microsoft Specific
```

Remarks

The insertion (called inline expansion or inlining) occurs only if the compiler's cost/benefit analysis show it to be profitable. Inline expansion alleviates the function-call overhead at the potential cost of larger code size.

The **__forceinline** keyword overrides the cost/benefit analysis and relies on the judgment of the programmer instead. Exercise caution when using **__forceinline**. Indiscriminate use of **__forceinline** can result in larger code with only marginal performance gains or, in some cases, even performance losses (due to increased paging of a larger executable, for example).

Using inline functions can make your program faster because they eliminate the overhead associated with function calls. Functions expanded inline are subject to code optimizations not available to normal functions.

The compiler treats the inline expansion options and keywords as suggestions. There is no guarantee that functions will be inlined. You cannot force the compiler to inline a particular function, even with the **__forceinline** keyword. When compiling with **/clr**, the compiler will not inline a function if there are security attributes applied to the function.

The **inline** keyword is available only in C++. The **__inline** and **__forceinline** keywords are available in both C and C++. For compatibility with previous versions, **_inline** is a synonym for **__inline**.

The **inline** keyword tells the compiler that inline expansion is preferred. However, the compiler can create a separate instance of the function (instantiate) and create standard calling linkages instead of inserting the code inline. Two cases where this can happen are:

- Recursive functions.
- Functions that are referred to through a pointer elsewhere in the translation unit.

These reasons may interfere with inlining, *as may others*, at the discretion of the compiler; you should not depend on the **inline** specifier to cause a function to be inlined.

Functions that are declared as **inline**, and that are not class member functions, have internal linkage unless otherwise specified.

As with normal functions, there is no defined order of evaluation of the arguments to an inline function. In fact, it could be different from the order in which the arguments are evaluated when passed using normal function call protocol.

The /Ob compiler optimization option helps to determine whether inline function expansion actually occurs.

/LTCG performs cross-module inlining regardless of whether it was requested in source code.

Example 1

```
// inline_keyword1.cpp
// compile with: /c
inline int max( int a , int b ) {
   if( a > b )
      return a;
   return b;
}
```

A class's member functions can be declared inline either by using the **inline** keyword or by placing the function definition within the class definition.

Example 2

```
// inline_keyword2.cpp
// compile with: /EHsc /c
```

```
#include <iostream>
using namespace std;

class MyClass {
public:
   void print() { cout << i << ' '; }   // Implicitly inline
private:
   int i;
};
```

Microsoft Specific

The **__inline** keyword is equivalent to **inline**.

Even with **__forceinline**, the compiler cannot inline code in all circumstances. The compiler cannot inline a function if:

- The function or its caller is compiled with /Ob0 (the default option for debug builds).
- The function and the caller use different types of exception handling (C++ exception handling in one, structured exception handling in the other).
- The function has a variable argument list.
- The function uses inline assembly, unless compiled with /Og, /Ox, /O1, or /O2.
- The function is recursive and not accompanied by **#pragma inline_recursion(on)**. With the pragma, recursive functions are inlined to a default depth of 16 calls. To reduce the inlining depth, use inline_depth pragma.
- The function is virtual and is called virtually. Direct calls to virtual functions can be inlined.
- The program takes the address of the function and the call is made via the pointer to the function. Direct calls to functions that have had their address taken can be inlined.
- The function is also marked with the naked __declspec modifier.

If the compiler cannot inline a function declared with **__forceinline**, it generates a level 1 warning.

Recursive functions can be substituted inline to a depth specified by the inline_depth pragma, up to a maximum of 16 calls. After that depth, recursive function calls are treated as calls to an instance of the function. The depth to which recursive functions are examined by the inline heuristic cannot exceed 16. The inline_recursion pragma controls the inline expansion of a function currently under expansion. See the Inline-Function Expansion (/Ob) compiler option for related information.

END Microsoft Specific

For more information on using the **inline** specifier, see:

- Inline Class Member Functions
- Inline Functions versus Macros
- When to Use Inline Functions
- Defining Inline C++ Functions with dllexport and dllimport

See Also
Reference
C++ Keywords
noinline
auto_inline

Inline Class Member Functions

A function defined in the body of a class declaration is an inline function.

Example

In the following class declaration, the `Account` constructor is an inline function. The member functions `GetBalance`, `Deposit`, and `Withdraw` are not specified as **inline** but can be implemented as inline functions.

```
// Inline_Member_Functions.cpp
class Account
{
public:
    Account(double initial_balance) { balance = initial_balance; }
    double GetBalance();
    double Deposit( double Amount );
    double Withdraw( double Amount );
private:
    double balance;
};

inline double Account::GetBalance()
{
    return balance;
}

inline double Account::Deposit( double Amount )
{
    return ( balance += Amount );
}

inline double Account::Withdraw( double Amount )
{
    return ( balance -= Amount );
}
int main()
{
}
```

Note

In the class declaration, the functions were declared without the **inline** keyword. The **inline** keyword can be specified in the class declaration; the result is the same.

A given inline member function must be declared the same way in every compilation unit. This constraint causes inline functions to behave as if they were instantiated functions. Additionally, there must be exactly one definition of an inline function.

A class member function defaults to external linkage unless a definition for that function contains the **inline** specifier. The preceding example shows that these functions need not be explicitly declared with the **inline** specifier; using **inline** in the function definition causes it to be an inline function. However, it is illegal to redeclare a function as **inline** after a call to that function.

See Also

Reference

inline, __inline, __forceinline

Inline Functions versus Macros

Although inline functions are similar to macros (because the function code is expanded at the point of the call at compile time), inline functions are parsed by the compiler, whereas macros are expanded by the preprocessor. As a result, there are several important differences:

- Inline functions follow all the protocols of type safety enforced on normal functions.
- Inline functions are specified using the same syntax as any other function except that they include the **inline** keyword in the function declaration.
- Expressions passed as arguments to inline functions are evaluated once. In some cases, expressions passed as arguments to macros can be evaluated more than once.

Example

The following example shows a macro that converts lowercase letters to uppercase:

```
// inline_functions_macro.c
#include <stdio.h>
#include <conio.h>

#define toupper(a) ((a) >= 'a' && ((a) <= 'z') ? ((a)-('a'-'A')):(a))

int main() {
   char ch;
   printf_s("Enter a character: ");
   ch = toupper( getc(stdin) );
   printf_s( "%c", ch );
}
```

Input

```
xyz
```

Inline functions remedy the problem previously described.

```
// inline_functions_inline.cpp
#include <stdio.h>
#include <conio.h>

inline char toupper( char a ) {
   return ((a >= 'a' && a <= 'z') ? a-('a'-'A') : a );
}

int main() {
   printf_s("Enter a character: ");
   char ch = toupper( getc(stdin) );
   printf_s( "%c", ch );
}
```

Input

```
a
```

Output

```
Enter a character: A
```

Sample Output

```
Z
```

The intent of the expression `toupper( getc(stdin)` is that a character should be read from the console device (**stdin**) and, if

necessary, converted to uppercase.

Because of the implementation of the macro, **getc** is executed once to determine whether the character is greater than or equal to "a," and once to determine whether it is less than or equal to "z." If it is in that range, **getc** is executed again to convert the character to uppercase. This means the program waits for two or three characters when, ideally, it should wait for only one.

See Also

Reference

inline, __inline, __forceinline

When to Use Inline Functions

Inline functions are best used for small functions such as accessing private data members. The main purpose of these one- or two-line "accessor" functions is to return state information about objects; short functions are sensitive to the overhead of function calls. Longer functions spend proportionately less time in the calling/returning sequence and benefit less from inlining.

Example

The `Point` class, introduced in Function-Call Results can be optimized as follows:

```
// when_to_use_inline_functions.cpp
class Point
{
public:
    // Define "accessor" functions as
    //  reference types.
    unsigned& x();
    unsigned& y();
private:
    unsigned _x;
    unsigned _y;
};

inline unsigned& Point::x()
{
    return _x;
}
inline unsigned& Point::y()
{
    return _y;
}
int main()
{
}
```

Assuming coordinate manipulation is a relatively common operation in a client of such a class, specifying the two accessor functions (`x` and `y` in the preceding example) as **inline** typically saves the overhead on:

- Function calls (including parameter passing and placing the object's address on the stack)
- Preservation of caller's stack frame
- New stack-frame setup
- Return-value communication
- Old stack-frame restore
- Return

See Also

Reference

inline, __inline, __forceinline

virtual Specifier

The virtual keyword can be applied only to nonstatic class member functions. It signifies that binding of calls to the function is deferred until run time. For more information, see Virtual Functions.

See Also

Reference

Function Specifiers

typedef Specifier

A **typedef** declaration introduces a name that, within its scope, becomes a synonym for the type given by the *type-declaration* portion of the declaration.

```
typedef type-declaration synonym;
```

Remarks

You can use typedef declarations to construct shorter or more meaningful names for types already defined by the language or for types that you have declared. Typedef names allow you to encapsulate implementation details that may change.

In contrast to the **class**, **struct**, **union**, and **enum** declarations, **typedef** declarations do not introduce new types — they introduce new names for existing types.

You cannot use the **typedef** specifier inside a function definition.

Typedef names share the name space with ordinary identifiers. Therefore, a program can have a typedef name and a local-scope identifier by the same name.

For more information, see:

- Redeclaration of typedef Names
- Use of typedef with Class Types
- Name Space of typedef Names

Example

```
// typedef_specifier1.cpp
typedef char FlagType;

int main()
{
}

void myproc( int )
{
    int FlagType;
}
```

When declaring a local-scope identifier by the same name as a typedef, or when declaring a member of a structure or union in the same scope or in an inner scope, the type specifier must be specified. For example:

```
typedef char FlagType;
const FlagType x;
```

To reuse the `FlagType` name for an identifier, a structure member, or a union member, the type must be provided:

```
const int FlagType;  // Type specifier required
```

It is not sufficient to say

```
const FlagType;      // Incomplete specification
```

because the `FlagType` is taken to be part of the type, not an identifier that is being redeclared. This declaration is taken to be an illegal declaration like

```
int;  // Illegal declaration
```

You can declare any type with typedef, including pointer, function, and array types. You can declare a typedef name for a pointer to a structure or union type before you define the structure or union type, as long as the definition has the same visibility as the declaration.

Examples

One use of **typedef** declarations is to make declarations more uniform and compact. For example:

```
typedef char CHAR;          // Character type.
typedef CHAR * PSTR;        // Pointer to a string (char *).
PSTR strchr( PSTR source, CHAR target );
typedef unsigned long ulong;
ulong ul;     // Equivalent to "unsigned long ul;"
```

To use **typedef** to specify fundamental and derived types in the same declaration, you can separate declarators with commas. For example:

```
typedef char CHAR, *PSTR;
```

The following example provides the type `DRAWF` for a function returning no value and taking two int arguments:

```
typedef void DRAWF( int, int );
```

After the above `typedef` statement, the declaration

```
DRAWF box;
```

would be equivalent to the declaration

```
void box( int, int );
```

typedef is often combined with **struct** to declare and name user-defined types:

```
// typedef_specifier2.cpp
#include <stdio.h>

typedef struct mystructtag
{
    int   i;
    double f;
} mystruct;

int main()
{
    mystruct ms;
    ms.i = 10;
    ms.f = 0.99;
    printf_s("%d   %f\n", ms.i, ms.f);
}
```

Output

```
10   0.990000
```

See Also

Reference

C++ Keywords

C++ Type Names

Redeclaration of typedef Names

The **typedef** declaration can be used to redeclare the same name to refer to the same type. For example:

```
// FILE1.H
typedef char CHAR;

// FILE2.H
typedef char CHAR;

// PROG.CPP
#include "file1.h"
#include "file2.h"   // OK
```

The program PROG.CPP includes two header files, both of which contain **typedef** declarations for the name `CHAR`. As long as both declarations refer to the same type, such redeclaration is acceptable.

A **typedef** cannot redefine a name that was previously declared as a different type. Therefore, if FILE2.H contains

```
// FILE2.H
typedef int CHAR;     // Error
```

the compiler issues an error because of the attempt to redeclare the name `CHAR` to refer to a different type. This extends to constructs such as:

```
typedef char CHAR;
typedef CHAR CHAR;      // OK: redeclared as same type

typedef union REGS      // OK: name REGS redeclared
{                       //  by typedef name with the
    struct wordregs x;  //  same meaning.
    struct byteregs h;
} REGS;
```

See Also

Reference

typedef Specifier

Use of typedef with Class Types

Use of the **typedef** specifier with class types is supported largely because of the ANSI C practice of declaring unnamed structures in **typedef** declarations. For example, many C programmers use the following:

```
// typedef_with_class_types1.cpp
// compile with: /c
typedef struct {   // Declare an unnamed structure and give it the
                   // typedef name POINT.
   unsigned x;
   unsigned y;
} POINT;
```

The advantage of such a declaration is that it enables declarations like:

```
POINT ptOrigin;
```

instead of:

```
struct point_t ptOrigin;
```

In C++, the difference between **typedef** names and real types (declared with the **class**, **struct**, **union**, and **enum** keywords) is more distinct. Although the C practice of declaring a nameless structure in a **typedef** statement still works, it provides no notational benefits as it does in C.

```
// typedef_with_class_types2.cpp
// compile with: /c /W1
typedef struct {
   int POINT();
   unsigned x;
   unsigned y;
} POINT;
```

The preceding example declares a class named `POINT` using the unnamed class **typedef** syntax. `POINT` is treated as a class name; however, the following restrictions apply to names introduced this way:

- The name (the synonym) cannot appear after a **class**, **struct**, or **union** prefix.
- The name cannot be used as constructor or destructor names within a class declaration.

In summary, this syntax does not provide any mechanism for inheritance, construction, or destruction.

See Also
Reference
typedef Specifier

Name Space of typedef Names

Names declared using **typedef** occupy the same name space as other identifiers (except statement labels). Therefore, they cannot use the same identifier as a previously declared name, except in a class-type declaration. Consider the following example:

```
// typedef_names1.cpp
// C2377 expected
typedef unsigned long UL;   // Declare a typedef name, UL.
int UL;                     // C2377: redefined.
```

The name-hiding rules that pertain to other identifiers also govern the visibility of names declared using **typedef**. Therefore, the following example is legal in C++:

```
// typedef_names2.cpp
typedef unsigned long UL;   // Declare a typedef name, UL
int main()
{
   unsigned int UL;   // Redeclaration hides typedef name
}

// typedef UL back in scope
```

See Also

Reference

typedef Specifier

friend Specifier

The **friend** specifier is used to designate functions or classes that have the same access privileges as class member functions. Friend functions and classes are covered in detail in friend.

See Also
Reference
Specifiers

C++ Type Specifiers

Type specifiers determine the type of the name being declared. The following are type specifiers:

```
simple-type-name
class-specifier
enum-specifier
elaborated-type-specifier
:: class-name
<A HREF="_langref_const.htm">const</A>
<A HREF="_langref_volatile.htm">volatile</A>
```

Remarks

The following sections discuss simple type names, elaborated type specifiers, and nested type names.

See Also

Reference

Specifiers

Simple Type Names

A simple type name is the name of a simple type. That is, a type that is not a pointer, reference, array or function pointer.

```
class-name
[ :: ] nested-name-specifier type-name
[ :: ] nested-name-specifier template template-id
char
wchar_t
bool
short
int
long
signed
unsigned
float
double
void
```

Remarks

A simple type name may be qualified by a nested-name-specifier, indicating the namespace or containing class.

```
int  // simple type name
unsigned int  // combination of simple type names
MyClass  // a class type
class MyClass  // class is optional when using the type name
struct MyStruct  // the keyword struct is optional in C++
enum MyEnum  // the keyword enum is optional in C++
::MyClass  // type name at global scope
Outer::Inner  // nested type name
::Outer::Inner  // nested type names with global scope operator
MyTemplate<int>  // a class template
Outer::Inner<int> // an inner class template
Outer<char>::Inner<int>  // an inner class template of a template class
::template MyTemplate<int>  // using the template keyword
typename MyClass  // the typename keyword (only in a template definition)
```

The following table shows how the simple type names can be used together.

Type Name Combinations

Type	Can Appear With	Comments
int	**long** or **short**, but not both	Type **int** implies type **long int**.
long	**int** or **double**	Type **long** implies type **long int**.
short	**int**	Type **short** implies type **short int**.
signed	**char**, **short**, **int**, or **long**	Type **signed** implies **signed int**. The most-significant bit of objects of type **signed char** and bit fields of signed integral types is taken to be the sign bit.
unsigned	**char**, **short**, **int**, or **long**	Type **unsigned** implies **unsigned int**. The most-significant bit of objects of type **unsigned char** and bit fields of unsigned integral types is not treated as the sign bit.

See Also

Reference

C++ Type Specifiers

Elaborated Type Specifiers

Elaborated type specifiers are used to declare user-defined types. These can be either class- or enumerated-types.

```
class identifier
struct identifier
union identifier
enum identifier
```

Remarks

If *identifier* is specified, it is taken to be a class name. For example:

```
class Window;
```

This statement declares the `Window` identifier as a class name. This syntax is used for forward declaration of classes. For more information about class names, see Class Names.

If a name is declared using the **union** keyword, it must also be defined using the **union** keyword. Names that are defined using the **class** keyword can be declared using the **struct** keyword (and vice versa). Therefore, the following code samples are legal:

Example

```
// elaborated_type_specifiers1.cpp
struct A;   // Forward declaration of A.

class A   // Define A.
{
public:
   int i;
};

int main()
{
}
```

```
// elaborated_type_specifiers2.cpp
class A;   // Forward declaration of A

struct A
{
private:
    int i;
};

int main()
{
}
```

```
// elaborated_type_specifiers3.cpp
union A;   // Forward declaration of A

union A
{
   int  i;
   char ch[2];
};

int main()
{
```

```
}
```

The following examples, however, are illegal:

```
// elaborated_type_specifiers4.cpp
union A;   // Forward declaration of A.

struct A
{   // C2011
   int i;
};
```

```
// elaborated_type_specifiers5.cpp
union A;   // Forward declaration of A.

class A
{   // C2011
public:
   int i;
};
```

```
// elaborated_type_specifiers6.cpp
struct A;   // Forward declaration of A.

union A
{   // C2011
   int  i;
   char ch[2];
};
```

See Also

Reference

C++ Type Specifiers

Nested Type Names

Microsoft C++ supports declaration of nested types — both named and anonymous. Anonymous structs and classes are a Microsoft specific extension to C++. Anonymous unions are standard.

```
[::] class-name :: class-name
[::] template-id :: class-name
[::] class-name :: template-id
[::] template-id :: template template-id
:: template template-id :: template template-id
```

Remarks

The first class-name above refers to the outer class; the second refers to the inner class. These names may be further nested. Thus,

```
Outer::Inner
::Outer::Inner
::Outer::Inner::Inner2
```

are all valid names of classes. In place of class names, template identifiers may be used, with the optional template keyword, as in

```
MyClass<int>::Inner
Outer<int>::template Inner<int>
```

In some programming situations, it makes sense to define nested types. These types are visible only to member functions of the class type in which they are defined. They can also be made visible by constructing a qualified type name using the scope-resolution operator (**::**).

Note

One commonly used class hierarchy that employs nested types is iostream. In the iostream header files, the definition of clas s **ios** includes a series of enumerated types, which are packaged for use only with the iostream library.

Example

The following example defines nested classes:

```
// nested_type_names1.cpp
class WinSystem
{
public:
   class Window
   {
   public:
      Window();   // Default constructor.
      ~Window();   // Destructor.
      int NumberOf();   // Number of objects of class.
      int Count();   // Count number of objects of class.
   private:
      static int CCount;
   };

   class CommPort
   {
   public:
      CommPort();   // Default constructor.
      ~CommPort();   // Destructor.
      int NumberOf();   // Number of objects of class.
      int Count();   // Count number of objects of class.
```

```
   private:
      static int CCount;
   };
};

// Initialize WinSystem static members.
int WinSystem::Window::CCount = 0;
int WinSystem::CommPort::CCount = 0;

int main()
{
}
```

To access a name defined in a nested class, use the scope-resolution operator (**::**) to construct a complete class name. Use of this operator is shown in the initializations of the **static** members in the preceding example. To use a nested class in your program, use code such as:

```
WinSystem::Window Desktop;
WinSystem::Window AppWindow;

cout << "Number of active windows: " << Desktop.Count() << "\n";
```

Microsoft Specific

Nested anonymous classes or structures can be defined as:

```
// nested_type_names2.cpp
class Ledger
{
   class
   {
   public:
      double PayableAmt;
      unsigned PayableDays;
   } Payables;

   class
   {
   public:
      double RecvableAmt;
      unsigned RecvableDays;
   } Receivables;
};

int main()
{
}
```

An anonymous class must be an aggregate that has no member functions and no static members.

END Microsoft specific

Note
Although an enumerated type can be defined inside a class declaration, the reverse is not true; class types cannot be defined inside enumeration declarations.

See Also

Reference

C++ Type Specifiers

const (C++)

When modifying a data declaration, the **const** keyword specifies that the object or variable is not modifiable.

```
const declaration ;
member-function const ;
```

Remarks

When following a member function's parameter list, the **const** keyword specifies that the function does not modify the object for which it is invoked.

For more information on **const**, see the following topics:

- Constant Values
- Constant Member Functions
- const and volatile Pointers
- Type Qualifiers (C Language Reference)
- volatile
- #define.

See Also

Reference

C++ Keywords

Constant Values

The **const** keyword specifies that a variable's value is constant and tells the compiler to prevent the programmer from modifying it.

```
// constant_values1.cpp
int main() {
   const int i = 5;
   i = 10;   // C3892
   i++;   // C2105
}
```

In C++, you can use the **const** keyword instead of the #define preprocessor directive to define constant values. Values defined with **const** are subject to type checking, and can be used in place of constant expressions. In C++, you can specify the size of an array with a **const** variable as follows:

```
// constant_values2.cpp
// compile with: /c
const int maxarray = 255;
char store_char[maxarray];  // allowed in C++; not allowed in C
```

In C, constant values default to external linkage, so they can appear only in source files. In C++, constant values default to internal linkage, which allows them to appear in header files.

The **const** keyword can also be used in pointer declarations.

```
// constant_values3.cpp
int main() {
   char *mybuf = 0, *yourbuf;
   char *const aptr = mybuf;
   *aptr = 'a';   // OK
   aptr = yourbuf;   // C3892
}
```

A pointer to a variable declared as **const** can be assigned only to a pointer that is also declared as **const**.

```
// constant_values4.cpp
#include <stdio.h>
int main() {
   const char *mybuf = "test";
   char *yourbuf = "test2";
   printf_s("%s\n", mybuf);

   const char *bptr = mybuf;   // Pointer to constant data
   printf_s("%s\n", bptr);

   // *bptr = 'a';   // Error
}
```

Output

```
test
test
```

You can use pointers to constant data as function parameters to prevent the function from modifying a parameter passed through a pointer.

For objects that are declared as **const**, you can only call constant member functions. This ensures that the constant object is never modified.

```
birthday.getMonth();     // Okay
birthday.setMonth( 4 ); // Error
```

You can call either constant or nonconstant member functions for a nonconstant object. You can also overload a member function using the **const** keyword; this allows a different version of the function to be called for constant and nonconstant objects.

You cannot declare constructors or destructors with the **const** keyword.

C and C++ const Differences

When you declare a variable as **const** in a C source code file, you do so as:

```
const int i = 2;
```

You can then use this variable in another module as follows:

```
extern const int i;
```

But to get the same behavior in C++, you must declare your **const** variable as:

```
extern const int i = 2;
```

If you wish to declare an `extern` variable in a C++ source code file for use in a C source code file, use:

```
extern "C" const int x=10;
```

to prevent name mangling by the C++ compiler.

See Also

Reference

const (C++)

Constant Member Functions

Constant Member Functions

Declaring a member function with the **const** keyword specifies that the function is a "read-only" function that does not modify the object for which it is called.

To declare a constant member function, place the **const** keyword after the closing parenthesis of the argument list. The **const** keyword is required in both the declaration and the definition. A constant member function cannot modify any data members or call any member functions that aren't constant.

Example

```
// constant_member_function.cpp
class Date
{
public:
   Date( int mn, int dy, int yr );
   int getMonth() const;     // A read-only function
   void setMonth( int mn );   // A write function; can't be const
private:
   int month;
};

int Date::getMonth() const
{
   return month;        // Doesn't modify anything
}
void Date::setMonth( int mn )
{
   month = mn;          // Modifies data member
}
int main()
{
   Date MyDate( 7, 4, 1998 );
   const Date BirthDate( 1, 18, 1953 );
   MyDate.setMonth( 4 );    // Okay
   BirthDate.getMonth();    // Okay
   BirthDate.setMonth( 4 ); // C2662 Error
}
```

See Also

Reference

const (C++)

Constant Values

volatile (C++)

The **volatile** keyword is a type qualifier used to declare that an object can be modified in the program by something such as the operating system, the hardware, or a concurrently executing thread.

```
volatile declarator ;
```

Remarks

The following example declares a volatile integer `nVint` whose value can be modified by external processes:

```
int volatile nVint;
```

Microsoft Specific

Objects declared as **volatile** are not used in certain optimizations because their values can change at any time. The system always reads the current value of a volatile object at the point it is requested, even if a previous instruction asked for a value from the same object. Also, the value of the object is written immediately on assignment.

Also, when optimizing, the compiler must maintain ordering among references to volatile objects as well as references to other global objects. In particular,

- A write to a volatile object (volatile write) has Release semantics; a reference to a global or static object that occurs before a write to a volatile object in the instruction sequence will occur before that volatile write in the compiled binary.
- A read of a volatile object (volatile read) has Acquire semantics; a reference to a global or static object that occurs after a read of volatile memory in the instruction sequence will occur after that volatile read in the compiled binary.

This allows volatile objects to be used for memory locks and releases in multithreaded applications.

Note
Although the processor will not reorder un-cacheable memory accesses, un-cacheable variables must be volatile to guarante e that the compiler will not change memory order.

End Microsoft Specific

One use of the **volatile** qualifier is to provide access to memory locations used by asynchronous processes such as interrupt handlers.

Note
When used on a variable that also has the __restrict keyword, **volatile** will take precedence.

Example

This program uses two features of volatile memory.

- It must honor all reads and writes from volatile memory; this allows it to be used for a mutex.
- It cannot move a reference to global data down across a volatile write; this allows you to use it to guard a critical section.
- It cannot move a reference to global data up across a volatile read; this ensures a critical section is exited before loading critical data.

```
// volatile.cpp
// compile with: /EHsc /O2
#include <iostream>
#include <windows.h>
using namespace std;

volatile bool Sentinel = true;
int CriticalData = 0;
```

```
unsigned ThreadFunc1( void* pArguments ) {
   while (Sentinel)
      Sleep(0);   // volatile spin lock

   // CriticalData load guaranteed after every load of Sentinel
   cout << "Critical Data = " << CriticalData << endl;
   return 0;
}

unsigned  ThreadFunc2( void* pArguments ) {
   Sleep(2000);
   CriticalData++;   // guaranteed to occur before write to Sentinel
   Sentinel = false; // exit critical section
   return 0;
}

int main() {
   HANDLE hThread1, hThread2;
   DWORD retCode;

   hThread1 = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)&ThreadFunc1,
      NULL, 0, NULL);
   hThread2 = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)&ThreadFunc2,
      NULL, 0, NULL);

   if (hThread1 == NULL || hThread2 == NULL)       {
      cout << "CreateThread failed." << endl;
      return 1;
   }

   retCode = WaitForSingleObject(hThread1,3000);

   CloseHandle(hThread1);
   CloseHandle(hThread2);

   if (retCode == WAIT_OBJECT_0 && CriticalData == 1 )
      cout << "Success" << endl;
   else
      cout << "Failure" << endl;
}
```

Output

```
Critical Data = 1
Success
```

See Also

Reference

C++ Keywords

const (C++)

const and volatile Pointers

C++ Enumeration Declarations

An enumeration is a user-defined type consisting of a set of named constants called enumerators.

For information on CLR enums, see enum class.

```
enum [tag] [: type] {enum-list} [declarator];   // for definition of enumerated type
enum tag declarator;   // for declaration of variable of type tag
```

Parameters

tag

The type name given to the enumeration.

type

The underlying type of the enumeration identifiers. See Remarks.

enum-list

List of the enumerators contained by the enumeration.

declarator

Declarator list specifying the names of the enumeration. See Overview of Declarators for more information.

Remarks

By default, the first enumerator has a value of 0, and each successive enumerator is one larger than the value of the previous one, unless you explicitly specify a value for a particular enumerator. Enumerators needn't have unique values within an enumeration. The name of each enumerator is treated as a constant and must be unique within the scope where the **enum** is defined. An enumerator can be promoted to an integer value. However, converting an integer to an enumerator requires an explicit cast, and the results are not defined if the integer value is outside the range of the defined enumeration.

type is the underlying type of the *identifiers*. This can be any scalar type, such as signed or unsigned versions of **int**, **short**, or **long**. **bool** or **char** is also allowed.

Enumerated types are valuable when an object can assume a known and reasonably limited set of values. Consider the example of the suits from a deck of cards:

```
// enumeration_declarations.cpp
// compile with: /c
class Card {
public:
   enum Suit {
      Diamonds,
      Hearts,
      Clubs,
      Spades
   };

   // Declare two constructors: a default constructor,
   //  and a constructor that sets the cardinal and
   //  suit value of the new card.
   Card();
   Card( int CardInit, Suit SuitInit );

   // Get and Set functions.
   int   GetCardinal();          // Get cardinal value of card.
   int   SetCardinal();          // Set cardinal value of card.
   Suit  GetSuit();              // Get suit of card.
   void  SetSuit(Suit new_suit); // Set suit of card.
   char *NameOf();               // Get string representation of card.

private:
   Suit  suit;
```

```
   int   cardinalValue;
};

// Define a postfix increment operator for Suit.
inline Card::Suit operator++( Card::Suit &rs, int ) {
   return rs = (Card::Suit)(rs + 1);
}
```

The preceding example defines a class, `Card`, that contains a nested enumerated type, `Suit`. To create a pack of cards in a program, use code such as:

```
Card *Deck[52];
int   j = 0;

for( Card::Suit curSuit = Card::Diamonds ; curSuit <= Card::Spades ; curSuit++ )
   for( int i = 1; i <= 13; ++i )
      Deck[j++] = new Card( i, curSuit );
```

In the preceding example, the type `Suit` is nested; therefore, the class name (`Card`) must be used explicitly in public references. In member functions, however, the class name can be omitted.

In the first segment of code, the postfix increment operator for `Card::Suit` is defined. Without a user-defined increment operator, `curSuit` could not be incremented. For more information about user-defined operators, see Overloaded Operators.

Consider the code for the `NameOf` member function (a better implementation is presented later):

```
#define SIZE_OF_CARD_NAMES 20
char* Card::NameOf() {   // Get the name of a card.
   static char szName[SIZE_OF_CARD_NAMES];
   static char *Numbers[] = {
      "1", "2", "3", "4", "5", "6", "7", "8", "9",
      "10", "Jack", "Queen", "King"
   };
   static char *Suits[] = {
      "Diamonds", "Hearts", "Clubs", "Spades"
   };

   if( GetCardinal() < 13)
      strcpy_s( szName, SIZE_OF_CARD_NAMES, Numbers[GetCardinal()] );

   strcat_s( szName, SIZE_OF_CARD_NAMES, " of " );

   switch( GetSuit() ) {
      // Diamonds, Hearts, Clubs, and Spades do not need explicit
      //  class qualifier.
      case Diamonds:
         strcat_s( szName, SIZE_OF_CARD_NAMES , "Diamonds" );
      break;
      case Hearts:
         strcat_s( szName, SIZE_OF_CARD_NAMES , "Hearts" );
      break;
      case Clubs:
         strcat_s( szName, SIZE_OF_CARD_NAMES , "Clubs" );
      break;
      case Spades:
         strcat_s( szName, SIZE_OF_CARD_NAMES , "Spades" );
      break;
   }
   return szName;
}
```

An enumerated type is an integral type. The identifiers introduced with the **enum** declaration can be used wherever constants appear. Normally, the first identifier's value is 0 (`Diamonds`, in the preceding example), and the values increase by one for each succeeding identifier. Therefore, the value of `Spades` is 3.

Any enumerator in the list, including the first one, can be initialized to a value other than its default value. Suppose the

declaration of `Suit` had been the following:

```
enum Suit {
   Diamonds = 5,
   Hearts,
   Clubs = 4,
   Spades
};
```

Then the values of `Diamonds`, `Hearts`, `Clubs`, and `Spades` would have been 5, 6, 4, and 5, respectively. Note that 5 is used more than once.

The default values for these enumerators simplify implementation of the `NameOf` function:

```
#define SIZE_OF_CARD_NAMES 20
char* Card::NameOf() {   // Get the name of a card.
   static char szName[SIZE_OF_CARD_NAMES];
   static char *Numbers[] = {
      "1", "2", "3", "4", "5", "6", "7", "8", "9",
      "10", "Jack", "Queen", "King"
   };
   static char *Suits[] = {
      "Diamonds", "Hearts", "Clubs", "Spades"
   };

   if( GetCardinal() < 13)
      strcpy_s( szName, SIZE_OF_CARD_NAMES, Numbers[GetCardinal()] );

   strcat_s( szName, SIZE_OF_CARD_NAMES, " of " );
   strcat_s( szName, SIZE_OF_CARD_NAMES, Suits[GetSuit()] );
   return szName;
}
```

The accessor function `GetSuit` returns type `Suit`, an enumerated type. Because enumerated types are integral types, they can be used as arguments to the array subscript operator. (For more information, see Subscript Operator.)

In C, the **enum** keyword is required to declare a variable of type *enumeration*. In C++, the **enum** keyword can be omitted. For example, given the `Days` enumeration from the code below:

```
Days tomorrow;          // Legal in C++ only
```

Example

```
// enumeration_declarations3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

enum Days {             // Declare enum type Days
   saturday,            // saturday = 0 by default
   sunday = 0,          // sunday = 0 as well
   monday,              // monday = 1
   tuesday,             // tuesday = 2
   wednesday,           // etc.
   thursday,
   friday
};

int main() {
   enum Days today = sunday;
   switch (today) {
      case 1:
         cout << "It's Monday" << endl;
         break;
      default:
         cout << "Not Monday" << endl;
```

```
    }
}
```

Output

```
Not Monday
```

See Also

Reference

C Enumeration Declarations
C++ Keywords
Enumerator Names
Definition of Enumerator Constants
Conversions and Enumerated Types

Enumerator Names

The names of enumerators must be different from any other enumerator or variable in the same scope. However, the values can be duplicated.

See Also

Reference

C++ Enumeration Declarations

Definition of Enumerator Constants

Enumerators are considered defined immediately after their initializers; therefore, they can be used to initialize succeeding enumerators. The following example defines an enumerated type that ensures that any two enumerators can be combined with the OR operator:

```
// enumerator_constants.cpp
enum FileOpenFlags
{
   OpenReadOnly  = 1,
   OpenReadWrite = OpenReadOnly  << 1,
   OpenBinary = OpenReadWrite << 1,
   OpenText = OpenBinary    << 1,
   OpenShareable = OpenText      << 1
};

int main()
{
}
```

In this example, the preceding enumerator initializes each succeeding enumerator.

See Also

Reference

C++ Enumeration Declarations

Conversions and Enumerated Types

Because enumerated types are integral types, any enumerator can be converted to another integral type by integral promotion. Consider this example:

```
// enumerated_types.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

enum Days
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};

int  i;
Days d = Thursday;
int main()
{
   i = d;    // Converted by integral promotion.
   cout << "i = " << i << "\n";
}
```

Output

```
i = 4
```

However, there is no implicit conversion from any integral type to an enumerated type. Therefore (continuing with the preceding example), the following statement is in error:

```
d = 6;    // Erroneous attempt to set d to Saturday.
```

Assignments such as this, where no implicit conversion exists, must use a cast to perform the conversion:

```
d = (Days)6;    // Explicit cast-style conversion to type Days.
d = Days( 4 );  // Explicit function-style conversion to type Days.
```

The preceding example shows conversions of values that coincide with the enumerators. There is no mechanism that protects you from converting a value that does not coincide with one of the enumerators. For example:

```
d = Days( 967 );
```

Some such conversions may work. However, there is no guarantee that the resultant value will be one of the enumerators. Additionally, if the size of the enumerator is too small to hold the value being converted, the value stored may not be what you expect.

See Also

Reference

C++ Enumeration Declarations

Linkage Specifications

The term "linkage specification" refers to the protocol for linking functions (or procedures) written in different languages. The following calling conventions are affected:

- Case sensitivity of names.
- Decoration of names. In C, the compiler prefixes names with an underscore. This is often called "decoration." In C++, name decoration is used to retain type information through the linkage phase. (See Decorated Names)
- Order in which arguments are expected on the stack.
- Responsibility for adjusting the stack on function return. Either the called function or the calling function is responsible.
- Passing of hidden arguments (whether any hidden arguments are passed).

```
        extern string-literal { declaration-list }
extern string-literal declaration
```

Linkage specification facilitates gradually porting C code to C++ by allowing the use of existing code.

Microsoft Specific

The only linkage specifications currently supported by Microsoft C++ are **"C"** and **"C++"**.

END Microsoft Specific

The following example declares the functions `atoi` and `atol` with C linkage:

```
extern "C"
{
    int  atoi( char *string );
    long atol( char *string );
}
```

Calls to these functions are made using C linkage. The same result could be achieved with these two declarations:

```
extern "C" int  atoi( char *string );
extern "C" long atol( char *string );
```

Microsoft Specific

All Microsoft C standard include files use conditional compilation directives to detect C++ compilation. When a C++ compilation is detected, the prototypes are enclosed in an **extern "C"** directive as follows:

```
// Sample.h
#if defined(__cplusplus)
extern "C"
{
#endif

// Function declarations

#if defined(__cplusplus)
}
#endif
```

END Microsoft Specific

You do not need to declare the functions in the standard include files as **extern "C"**.

If a function is overloaded, no more than one of the functions of the same name can have a linkage specifier. (For more information, see Function Overloading.)

The following table shows how various linkage specifications work.

Effects of Linkage Specifications

Specification	Effect
On an object	Affects linkage of that object only
On a function	Affects linkage of that function and all functions or objects declared within it
On a class	Affects linkage of all nonmember functions and objects declared within the class

If a function has more than one linkage specification, they must agree; it is an error to declare functions as having both C and C++ linkage. Furthermore, if two declarations for a function occur in a program — one with a linkage specification and one without — the declaration with the linkage specification must be first. Any redundant declarations of functions that already have linkage specification are given the linkage specified in the first declaration. For example:

```
extern "C" int CFunc1();
...
int CFunc1();            // Redeclaration is benign; C linkage is
                         //  retained.

int CFunc2();
...
extern "C" int CFunc2(); // Error: not the first declaration of
                         //  CFunc2;  cannot contain linkage
                         //  specifier.
```

Functions and objects explicitly declared as **static** within the body of a compound linkage specifier (**{ }**) are treated as static functions or objects; the linkage specifier is ignored. Other functions and objects behave as if declared using the **extern** keyword. (See Using extern to Specify Linkage for details about the **extern** keyword.)

See Also

Reference

Using extern to Specify Linkage

Namespaces (C++)

The C++ language provides a single global namespace. This can cause problems with global name clashes. For instance, consider these two C++ header files:

```
char func(char);
class String { ... };

// somelib.h
class String { ... };
```

With these definitions, it is impossible to use both header files in a single program; the `String` classes will clash.

A namespace is a declarative region that attaches an additional identifier to any names declared inside it. The additional identifier makes it less likely that a name will conflict with names declared elsewhere in the program. It is possible to use the same name in separate namespaces without conflict even if the names appear in the same translation unit. As long as they appear in separate namespaces, each name will be unique because of the addition of the namespace identifier. For example:

```
namespace one {
   char func(char);
   class String { ... };
}

// somelib.h
namespace SomeLib {
   class String { ... };
}
```

Now the class names will not clash because they become `one::String` and `SomeLib::String`, respectively.

C++ does not allow compound names for namespaces.

```
// pluslang_namespace.cpp
// compile with: /c
// OK
namespace a {
   namespace b {
      int i;
   }
}

// not allowed
namespace c::d {   // C2653
   int i;
}
```

Declarations in the file scope of a translation unit, outside all namespaces, are still members of the global namespace.

What do you want to know more about?

- namespace Declaration
- namespace Alias
- Unnamed Namespaces
- Defining Namespace Members
- using Declaration
- using Directive
- Explicit Qualification

See Also

Reference

Declarations

namespace Declaration

A **namespace** declaration identifies and assigns a unique name to a user-declared namespace.

```
namespace identifier
```

Remarks

Such namespaces are used to solve the problem of name collision in large programs and libraries. Programmers can use namespaces to develop new software components and libraries without causing naming conflicts with existing components.

For example:

```
// namespace_declaration1.cpp
namespace X
{
   int i;
   double j;
}
int main()
{
   X::i++;
}
```

The syntax for a namespace definition is:

```
namespace identifier
{
   [ declaration-list ]
}
```

A *namespace-definition* can be nested within another *namespace-definition*. Every *namespace-definition* must appear either at file scope or immediately within another *namespace-definition*.

For example:

```
// namespace_declaration2.cpp
// C2870 expected
namespace A
{
   int j = 3;
   int f(int k);
}

namespace Outer
{
   int n = 6;
   int func(int num);

   namespace Inner
   {
      float f = 9.993;
   }
}

int main()
{
   namespace local   // C2870: not at global scope
   {
   }
}
```

Unlike other declarative regions, the definition of a namespace can be split over several parts of a single translation unit.

```
// namespace_declaration3.cpp
namespace A
{
   // declare namespace A variables
   int i;
   int j;
}

namespace B
{
}

namespace A
{
   // declare namespace A functions
   void func(void);
   int int_func(int i);
}

int main()
{
}
```

When a namespace is continued in this manner, after its initial definition, the continuation is called an *extension namespace definition*. The original definition of that namespace is known as an *original namespace definition*.

Usage of this notation might be cumbersome with longer names or in large programs. The using declaration, using directive, and namespace aliases provide more straightforward ways to reference namespace members.

A **namespace** declaration, whether it involves a new namespace, an unnamed namespace, or an extended namespace definition, must be accompanied by a namespace body enclosed within curly braces. The statement

```
namespace X;
```

is a syntax error. The statement

```
namespace X{};
```

is not a syntax error, but is meaningless.

For background information, see Namespaces.

The identifier in an original namespace definition must be unique in the declarative region in which it is used. The identifier is the name of the namespace and is used to reference its members.

The declarative region of a namespace definition is its body. The body must be enclosed in curly braces (**{}**) and may contain declarations or definitions of variables, functions, objects, templates, and nested namespaces. The declarations in the *declaration-list* are said to be members of the namespace. The name of each namespace member is automatically qualified by the name of its namespace and the scope resolution operator.

See Also

Reference

Namespaces (C++)
C++ Keywords
Unnamed Namespaces

Unnamed Namespaces

You can declare an unnamed namespace as a superior alternative to the use of global static variable declarations.

```
namespace { declaration-list }
```

Remarks

An unnamed namespace definition having the syntax shown above behaves as if it were replaced by:

namespace *unique* **{** *declaration-list* **}**

using namespace *unique*;

Each unnamed namespace has an identifier, assigned and maintained by the program and represented here by *unique*, that differs from all other identifiers in the entire program. For example:

```
// unnamed_namespaces.cpp
// C2872 expected
namespace { int i; }          // unique::i
void f() { i++; }             // unique::i++

namespace A {
    namespace {
        int i;      // A::unique::i
        int j;      // A::unique::j
    }
}

using namespace A;

void h()
{
    i++;            // C2872: unique::i or A::unique::i
    A::i++;         // C2872: A::i undefined
    j++;            // A::unique::j++
}
```

Unnamed namespaces are a superior replacement for the static declaration of variables. They allow variables and functions to be visible within an entire translation unit, yet not visible externally. Although entities in an unnamed namespace might have external linkage, they are effectively qualified by a name unique to their translation unit and therefore can never be seen from any other translation unit.

See Also

Reference

Namespaces (C++)

Defining namespace Members

Members of a namespace may be defined within that namespace. For example:

```
namespace X { void f() { } }
```

Members of a named namespace can be defined outside the namespace in which they are declared by explicit qualification of the name being defined. However, the entity being defined must already be declared in the namespace. In addition, the definition must appear after the point of declaration in a namespace that encloses the declaration's namespace. For example:

```
// defining_namespace_members.cpp
// C2039 expected
namespace Q {
    namespace V {
        void f();
    }

    void V::f() { }        // ok
    void V::g() { }        // C2039, g() is not yet a member of V

    namespace V {
        void g();
    }
}
```

See Also

Reference

Namespaces (C++)

namespace Alias

A *namespace alias* is an alternative name for a **namespace**.

```
namespace identifier = [::] [nested-name-specifier] namespace-name;
```

Remarks

A namespace alias definition declares an alternate name for a namespace. The *identifier* is a synonym for the specified namespace and becomes a namespace alias. For example:

```
namespace a_very_long_namespace_name { ... }
namespace AVLNN = a_very_long_namespace_name;
// AVLNN is now a namespace-alias for a_very_long_namespace_name.
```

A namespace name or alias must be different from all other identifiers in its declarative region. A global namespace name or alias must be different from all other global entity names in the program.

See Also

Reference

Namespaces (C++)

using Declaration

The **using** declaration introduces a name into the declarative region in which the **using** declaration appears.

```
using [typename][::] nested-name-specifier unqualified-id
using :: unqualified-id
```

Remarks

The name becomes a synonym for an entity declared elsewhere. It allows an *individual* name from a specific namespace to be used without explicit qualification. This is in contrast to the **using** directive, which allows *all* the names in a namespace to be used without qualification. See using Directive for more information.

Example

A using declaration can be used in a class definition.

```
// using_declaration1.cpp
#include <stdio.h>
class B {
public:
   void f(char) {
      printf_s("In B::f()\n");
   }

   void g(char) {
      printf_s("In B::g()\n");
   }
};

class D : B {
public:
   using B::f;
   using B::g;
   void f(int) {
      printf_s("In D::f()\n");
      f('c');
   }

   void g(int) {
      printf_s("In D::g()\n");
      g('c');
   }
};

int main() {
   D myD;
   myD.f(1);
   myD.g('a');
}
```

Output

```
In D::f()
In B::f()
In B::g()
```

When used to declare a member, a using declaration must refer to a member of a base class.

```
// using_declaration2.cpp
#include <stdio.h>

class B {
```

```
public:
   void f(char) {
      printf_s("In B::f()\n");
   }

   void g(char) {
      printf_s("In B::g()\n");
   }
};

class C {
public:
   int g();
};

class D2 : public B {
public:
   using B::f;   // ok: B is a base of D2
   // using C::g;   // error: C isn't a base of D2
};

int main() {
   D2 MyD2;
   MyD2.f('a');
}
```

Output

```
In B::f()
```

Members declared with a using declaration can be referenced using explicit qualification. The **::** prefix refers to the global namespace.

```
// using_declaration3.cpp
#include <stdio.h>

void f() {
   printf_s("In f\n");
}

namespace A {
   void g() {
      printf_s("In A::g\n");
   }
}

namespace X {
   using ::f;   // global f
   using A::g;   // A's g
}

void h() {
   printf_s("In h\n");
   X::f();   // calls ::f
   X::g();   // calls A::g
}

int main() {
   h();
}
```

Output

```
In h
In f
In A::g
```

When a using declaration is made, the synonym created by the declaration refers only to definitions that are valid at the point of the using declaration. Definitions added to a namespace after the using declaration are not valid synonyms.

A name defined by a using declaration is an alias for its original name. It does not affect the type, linkage or other attributes of the original declaration.

```
// post_declaration_namespace_additions.cpp
// compile with: /c
namespace A {
   void f(int) {}
}

using A::f;   // f is a synonym for A::f(int) only

namespace A {
   void f(char) {}
}

void f() {
   f('a');   // refers to A::f(int), even though A::f(char) exists
}

void b() {
   using A::f;   // refers to A::f(int) AND A::f(char)
   f('a');   // calls A::f(char);
}
```

With respect to functions in namespaces, if a set of local declarations and using declarations for a single name are given in a declarative region, they must all refer to the same entity, or they must all refer to functions.

```
// functions_in_namespaces1.cpp
// C2874 expected
namespace B {
    int i;
    void f(int);
    void f(double);
}

void g() {
    int i;
    using B::i;   // error: i declared twice
    void f(char);
    using B::f;   // ok: each f is a function
}
```

In the example above, the `using B::i` statement causes a second `int i` to be declared in the `g()` function. The `using B::f` statement does not conflict with the `f(char)` function because the function names introduced by `B::f` have different parameter types.

A local function declaration cannot have the same name and type as a function introduced by using declaration. For example:

```
// functions_in_namespaces2.cpp
// C2668 expected
namespace B {
    void f(int);
    void f(double);
}

namespace C {
    void f(int);
    void f(double);
    void f(char);
}

void h() {
    using B::f;          // introduces B::f(int) and B::f(double)
```

```
    using C::f;          // C::f(int), C::f(double), and C::f(char)
    f('h');              // calls C::f(char)
    f(1);                // C2668 ambiguous: B::f(int) or C::f(int)?
    void f(int);         // C2883 conflicts with B::f(int) and C::f(int)
}
```

With respect to inheritance, when a using declaration introduces a name from a base class into a derived class scope, member functions in the derived class override virtual member functions with the same name and argument types in the base class.

```
// using_declaration_inheritance1.cpp
#include <stdio.h>
struct B {
   virtual void f(int) {
      printf_s("In B::f(int)\n");
   }

   virtual void f(char) {
      printf_s("In B::f(char)\n");
   }

   void g(int) {
      printf_s("In B::g\n");
   }

   void h(int);
};

struct D : B {
   using B::f;
   void f(int) {   // ok: D::f(int) overrides B::f(int)
      printf_s("In D::f(int)\n");
   }

   using B::g;
   void g(char) {   // ok: there is no B::g(char)
      printf_s("In D::g(char)\n");
   }

   using B::h;
   void h(int) {}   // Note: D::h(int) hides non-virtual B::h(int)
};

void f(D* pd) {
   pd->f(1);   // calls D::f(int)
   pd->f('a');   // calls B::f(char)
   pd->g(1);   // calls B::g(int)
   pd->g('a');   // calls D::g(char)
}

int main() {
   D * myd = new D();
   f(myd);
}
```

Output

```
In D::f(int)
In B::f(char)
In B::g
In D::g(char)
```

All instances of a name mentioned in a using declaration must be accessible. In particular, if a derived class uses a using declaration to access a member of a base class, the member name must be accessible. If the name is that of an overloaded member function, then all functions named must be accessible.

See Member-Access Control, for more information on accessibility of members.

```
// using_declaration_inheritance2.cpp
// C2876 expected
class A {
private:
   void f(char);
public:
   void f(int);
protected:
   void g();
};

class B : public A {
   using A::f;   // C2876: A::f(char) is inaccessible
public:
   using A::g;   // B::g is a public synonym for A::g
};
```

See Also

Reference

Namespaces (C++)

C++ Keywords

using Directive (C++)

The **using** directive allows the names in a **namespace** to be used without the *namespace-name* as an explicit qualifier. Of course, the complete, qualified name can still be used to improve readability.

```
using namespace [::] [ nested-name-specifier ] namespace-name
```

Remarks

Note the difference between the **using** directive and the using declaration : the **using** declaration allows an individual name to be used without qualification, the **using** directive allows all the names in a namespace to be used without qualification.

If a local variable has the same name as a namespace variable, the namespace variable is hidden. It is an error to have a namespace variable with the same name as a global variable.

Note
Putt your using directive at the beginning of the source code file to reduce the potential for unexpected behavior with IntelliSense.

The std namespace

The ANSI/ISO C++ standard requires you to explicitly declare the namespace in the standard library. For example, when using iostream, you must specify the namespace of **cout** in one of the following ways:

- `std::cout (explicitly)`
- `using std::cout` (**using** declaration)
- `using namespace std` (**using** directive)

/clr

The following sample shows how to allow names in a .NET Framework base class library **namespace** to be used without the *namespace-name* as an explicit qualifier.

```
// using_directive.cpp
// compile with: /c /clr
using namespace System::Reflection;
[assembly:AssemblyDescriptionAttribute("test")];
```

Example

```
// using_directive2.cpp
// compile with: /EHsc
#include <iostream>

int main() {
   std::cout << "Hello ";
   using namespace std;
   cout << "World." << endl;
}
```

Output

```
Hello World.
```

See Also

Reference

Namespaces (C++)

C++ Keywords

Explicit Qualification

Namespace members can be accessed using an explicit qualifier and the scope-resolution operator.

For example:

```
// explicit_qualification.cpp
int i;

namespace A
{
   int a, b, c;

   namespace B
   {
      int i, j, k;
   }
}

int main()
{
   A::a++;
   A::B::i++;   // B's i
   ::i++;   // the global I
}
```

The statement `::i++` accesses the `i` that is declared in the first statement of the example. Such usage of the scope-resolution operator without a preceding qualifier invokes the global namespace.

Usage of explicit qualification might be cumbersome with longer names or in large programs. The using declaration, using directive, and namespace aliases provide more straightforward ways to reference namespace members.

For more information, see Qualified Names.

See Also

Reference

Namespaces (C++)

Declarators

A declarator is the part of a declaration that gives an object, type, or function its name and indicates whether an object is a pointer, reference, or array. The base type itself is not considered part of the declarator, although it is part of the full declaration. Declarators appear in a declaration as one or more names separated by commas; each declarator may include an initializer.

This section includes the following topics:

- Overview
- Type names
- Abstract declarators
- Pointers
- References
- Arrays
- Function declarations
- Function definitions
- Initializers

See Also

Other Resources

C++ Language Reference

Overview of Declarators

Declarators are the components of a declaration that specify names of objects or functions. Declarators also specify whether or not the named object is an object, pointer, reference or array. While declarators do not specify the base type, they do modify the type information in the basic type to specify derived types such as pointers, references, and arrays. Applied to functions, the declarator works with the type specifier to fully specify the return type of a function to be an object, pointer, or reference. (Specifiers, discussed in Declarations, convey properties such as type and storage class. Modifiers, discussed in this section and in Microsoft-Specific Modifiers, modify declarators.) The following figure shows a complete declaration of `MyFunction`, and calls out the components of the declaration.

Specifiers, Modifiers, and Declarators

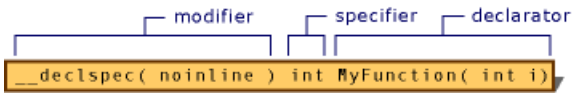

Microsoft Specific

Most Microsoft extended keywords can be used as modifiers to form derived types; they are not specifiers or declarators. (See Microsoft-Specific Modifiers.)

END Microsoft Specific

Declarators appear in the declaration syntax after an optional list of specifiers. These specifiers are discussed in Declarations. A declaration can contain more than one declarator, but each declarator declares only one name.

The following sample declaration shows how specifiers and declarators are combined to form a complete declaration:

```
const char *pch, ch;
```

In the preceding declaration, the keywords **const** and **char** make up the list of specifiers. Two declarators are listed: `*pch` and `ch`. A declaration that declares multiple entities consists of a type specifier followed by a comma-separated list of declarators, terminated with a semicolon.

Declarators for Simple Objects

The declarator of a simple object such as an int or double is simply its name, with optional parentheses.

```
int i; // declarator is i
```

```
int (i); // declarator is (i)
```

Declarators for Pointers, References and Arrays

Pointer operators inserted in front of the name cause the object to be a pointer or reference. The ***** operator declares the name as a pointer; the **&** operator declares it as a reference.

```
int *i; // declarator is *i
int &i; // declarator is &i
int **i; // declarator is **i;
```

Appending `const` or `volatile` give the pointer these special properties. The use of these specifiers in a declarator (as opposed to in the type specifier) modifies the properties of the pointer, not the object pointed to:

```
char *const cpc; // const pointer to char
const char *pcc; // pointer to const char
const char *const cpcc; // const pointer to const char
```

Further information may be found in const and volatile Pointers.

A pointer to a member of a class or struct is declared with the appropriate nested name specifier:

```
int X::* pIntMember;
int ::X::* pIntMember; // the initial :: specifies X is in global scope
```

```
char Outer::Inner::* pIntMember; // pointer to char in a nested class
```

Brackets enclosing an optional constant expression after the name cause the object to be an array. Successive brackets declare additional dimensions to the array.

```
int i[5]; // array with five elements of type int numbered from 0 to 4
int i[]; // array of unknown size
char *s[4]; // array of pointers to char
int i[2][2]; // two dimensional array
```

Declarators for Functions

Parentheses containing the argument list are used after the name to declare a function. The following declares a function of return type **int** and three arguments of type **int**.

```
int f(int a, int b, int c);
```

For information on argument lists, see Function Declarations.

Pointers and references to functions are declared by prepending the pointer or reference operator to the function name as shown below. Parentheses, normally optional, are required to distinguish a pointer to a function from a function that returns a pointer:

```
int (*pf)(int); // pointer to function returning int
int *f(int i); // function returning pointer to int
int (&pf)(int); // reference to function
```

Pointers to member functions are distinguished by nested name specifiers:

```
int (X::* pmf)(); // pointer to member function of X returning int
int* (X::* pmf)(); // pointer to member function returning pointer to int
```

See also Pointers to Members.

Functions and objects in the same declaration

Functions and objects may be declared in the same declaration as follows:

```
int i, *j, f(int k);  // int, pointer to int, function returning int
```

The syntax may be misleading in some circumstances. The following declaration

```
int* i, f(int k);  // pointer to int, function returning int (not int*)
```

may look like the declaration of an `int` pointer and a function returning `int*`, but it is not. That's because the * is part of the declarator for `i`, not part of the declarator for `f`.

Simplifying declarator syntax with typedef

A better technique, however, is to use a **typedef** or a combination of parentheses and the **typedef** keyword. Consider declaring an array of pointers to functions:

```
//  Function returning type int that takes one
//   argument of type char *.
typedef int (*PIFN)( char * );
//  Declare an array of 7 pointers to functions
//   returning int and taking one argument of type
//   char *.
PIFN pifnDispatchArray[7];
```

The equivalent declaration can be written without the **typedef** declaration, but it is so complicated that the potential for error

exceeds any benefits:

```
int ( *pifnDispatchArray[7] )( char * );
```

For more information on typedef, see typedef Specifier.

Pointers, references, arrays of a single base type can be combined in a single declaration (separated by commas) as

```
int i, *i, &i, i[5];
```

More complex declarator syntax

- Pointer, reference, array, and function declarators may be combined to specify such objects as arrays of pointers to functions, pointers to arrays, etc.
- The following recursive grammar describes pointer declarator syntax fully.
- A *declarator* is defined as one of:

```
1. identifier
2. qualified-name
3. declarator ( argument-list ) [cv-qualfiers] [exception-spec]
4. declarator [ [ constant-expression ] ]

5. pointer-operator declarator
6. ( declarator )
```

- and *pointer-operator* is one of:

```
      * [cv-qualifiers]
& [cv-qualifiers]
:: nested-name-specifier * [cv-qualfiers]
```

Because a declarator may contain declarators, it is possible to construct the more complex derived types such as arrays of pointers, functions returning arrays of function pointers, by using the above rules. To form each step of the construction, start with the identifier representing the base data type and apply the syntax rule above with the previous expression as the *declarator*. The order that you apply the syntax rules should be the reverse of the way the expression is stated in English. If applying the *pointer-operator* syntax rule to an array or function expression, use parentheses if you want a pointer to the array or function, as in the last row in the table below.

The following example shows the construction of "pointer to array of 10 pointers to int".

Verbal expression	Declarator	Syntax Rule Applied
	`i`	1
pointer(s) to	`*i`	5
array of 10	`(*i)[10]`	4
pointer to	`*((*i)[10])`	6 and then 5

When multiple pointer, reference, array or function modifiers are used, declarators may become quite complicated. The topic Interpreting More Complex Declarators describes how to read more complex declarator syntax. The topic is applicable to both C and C++, although in C++, anywhere the * is used to indicate a pointer, a qualified name such as MyClass::* may be used to specify a pointer to a member of a class.

See Also

Reference

Declarators

Type Names

Type names are used in some declarators in the following ways:

- In explicit conversions
- As arguments to the **sizeof** operator
- As arguments to the **new** operator
- In function prototypes
- In **typedef** statements

A type name consists of type specifiers such as **int** or the name of a class or struct, in combination with an abstract declarator, as described in Declarations and Abstract Declarators. The abstract declarator has the effect of modifying the base type to produce a pointer, reference or array type. It is not required if specifying a scalar type.

In the following example, the arguments to the strcpy_s function are supplied using their type names. In the case of the `strSource` argument, `const char` is the type specifier and `*` is the abstract declarator:

```
errno_t strcpy_s( char *strDestination, size_t numberOfElements, const char *strSource );
```

See Also
Reference
Declarators

C++ Abstract Declarators

An abstract declarator is a declarator in which the identifier is omitted. (For related information, see Type Names and Declarators) While non-abstract declarators are used in the declaration of objects or functions, stripping out the identifier leaves only the type information. Thus, abstract declarators have the effect of modifying a base type such as **char** to specify more complex type such as pointer to char or array of char. As such, they always are used in combination with a base type name and only used when a pointer, array or reference type is called for (including a pointer to function returning that base type). The abstract declarators corresponding to the following are discussed in this section:

- Ambiguity Resolution
- Pointers
- References
- Pointers to members
- Arrays
- Function pointers
- Default arguments

An abstract declarator is a declarator that does not declare a name — the identifier is left out. For example,

```
char *
```

specifies the type pointer to type **char**. The type name consists of the base type name **char** plus the abstract declarator ***.**

Similarly, the abstract declarator for a reference is the & operator, and an array type is specified using brackets. Any type may be specified by simply applying the syntax for the declarator and removing the identifier.

```
char &   // reference to char
char[5] // array of char
```

The following is a more complicated use of an abstract declarator that declares the type pointer to a function that takes two arguments, both of type **char ***, and returns type **char ***:

```
char * (*)( char *, char * )
```

In combination with the base type specifier, abstract declarators completely declare a type. The type names constructed from abstract declarators may be used anywhere a type is called for:

```
// Get the size of array of 10 pointers to type char.
size_t nSize = sizeof( char *[10] );
// cast fptr to pointer to function that takes two arguments, both of type
// pointer to char
 i = (char * (*)(char*, char*)) fptr;
```

The typedef Specifier may be used to avoid repeating the cumbersome syntax of abstract declarators.

See Also
Reference
Declarators

Ambiguity Resolution

To perform explicit conversions from one type to another, you must use casts, specifying the desired type name. Some type casts result in syntactic ambiguity. The following function-style type cast is ambiguous:

```
char *aName( String( s ) );
```

It is unclear whether it is a function declaration or an object declaration with a function-style cast as the initializer: It could declare a function returning type **char *** that takes one argument of type `String`, or it could declare the object `aName` and initialize it with the value of `s` cast to type `String`.

If a declaration can be considered a valid function declaration, it is treated as such. Only if it cannot possibly be a function declaration — that is, if it would be syntactically incorrect — is a statement examined to see if it is a function-style type cast. Therefore, the compiler considers the statement to be a declaration of a function and ignores the parentheses around the identifier `s`. On the other hand, the statements:

```
char *aName( (String)s );
```

and

```
char *aName = String( s );
```

are clearly declarations of objects, and a user-defined conversion from type `String` to type **char *** is invoked to perform the initialization of `aName`.

See Also

Reference

C++ Abstract Declarators

Pointers

Pointers are declared using the following sequence.

```
[storage-class-specifiers] [cv-qualifiers] type-specifiers
[ms-modifier] declarator ;
```

where any valid pointer declarator may be used for *declarator*. The syntax for a simple pointer declarator is as follows:

```
* [cv-qualifiers] identifier [= expression]
```

1. The declaration specifiers:

- An optional storage class specifier. For more information, see Specifiers.
- An optional **const** or **volatile** keyword applying to the type of the object to be pointed to.
- The type specifier: the name of a type representing the type of the object to be pointed to.

2. The declarator:

- An optional Microsoft specific modifier. For more information, see Microsoft-Specific Modifiers.
- The ***** operator.
- An optional **const** or **volatile** keyword applying to the pointer itself.
- The identifier.
- An optional initializer.

The declarator for a pointer to function looks like this:

```
(* [cv-qualifiers] identifier )( argument-list ) [cv-qualifers]
[exception specification] [= expression];
```

- For an array of pointers, the syntax looks like this:

```
* identifier [ [ constant-expression ] ]
```

- However, pointer declarators can be more complex. For more information,see Declarators.
- Multiple declarators and their initializers may appear together in a single declaration in a comma separated list following the declaration specifier.

A simple example of a pointer declaration is:

```
char *pch;
```

The preceding declaration specifies that `pch` points to an object of type **char**.

A more complex example is

```
static unsigned int * const ptr;
```

The preceding declaration specifies that `ptr` is a constant pointer to an object of type **unsigned int** with static storage duration.

The next example shows how multiple pointers are declared and initialized:

```
static int *p = &i, *q = &j;
```

In the preceding example, pointers p and q both point to objects of type **int** and are initialized to the addresses of i and j respectively. The storage class specifier **static** applies to both pointers.

Example

```
// pointer.cpp
// compile with: /EHsc
#include <iostream>
int main() {
   int i = 1, j = 2; // local variables on the stack
   int *p;

   // a pointer may be assigned to "point to" the value of
   // another variable using the & (address of) operator
   p = & j;

   // since j was on the stack, this address will be somewhere
   // on the stack.  Pointers are printed in hex format using
   // %p and conventionally marked with 0x.
   printf_s("0x%p\n",  p);

   // The * (indirection operator) can be read as "the value
   // pointed to by".
   // Since p is pointing to j, this should print "2"
   printf_s("0x%p %d\n",  p, *p);

   // changing j will change the result of the indirection
   // operator on p.
   j = 7;
   printf_s("0x%p %d\n",  p, *p );

   // The value of j can also be changed through the pointer
   // by making an assignment to the dereferenced pointer
   *p = 10;
   printf_s("j is %d\n", j); // j is now 10

   // allocate memory on the heap for an integer,
   // initialize to 5
   p = new int(5);

   // print the pointer and the object pointed to
   // the address will be somewhere on the heap
   printf_s("0x%p %d\n",  p, *p);

   // free the memory pointed to by p
   delete p;

   // At this point, dereferencing p with *p would trigger
   // a runtime access violation.

   // Pointer arithmetic may be done with an array declared
   // on the stack or allocated on the heap with new.
   // The increment operator takes into account the size
   // of the objects pointed to.
   p = new int[5];
   for (i = 0; i < 5; i++, p++) {
      *p = i * 10;
      printf_s("0x%p %d\n", p, *p);
   }

   // A common expression seen is dereferencing in combination
   // with increment or decrement operators, as shown here.
   // The indirection operator * takes precedence over the
   // increment operator ++.
   // These are particularly useful in manipulating char arrays.
```

```
   char s1[4] = "cat";
   char s2[4] = "dog";
   char* p1 = s1;
   char* p2 = s2;

   // the following is a string copy operation
   while (*p1++ = *p2++);

   // s2 was copied into s1, so now they are both equal to "dog"
   printf_s("%s %s", s1, s2);
}
```

Sample Output

```
0x0012FEC8
0x0012FEC8 2
0x0012FEC8 7
j is 10
0x00320850 5
0x00320850 0
0x00320854 10
0x00320858 20
0x0032085C 30
0x00320860 40
dog dog
```

Another example illustrates the use of pointers in data structures; in this case, a linked list.

```
// pointer_linkedlist.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

struct NewNode {
   NewNode() : node(0){}
   int i;
   NewNode * node;
};

void WalkList(NewNode * ptr) {
   if (ptr != 0) {
      int i = 1;
      while (ptr->node != 0 ) {
         cout << "node " << i++ << " = " << ptr->i << endl;
         ptr = ptr->node;
      }
      cout << "node " << i++ << " = " << ptr->i << endl;
   }
}

void AddNode(NewNode ** ptr) {
   NewNode * walker = 0;
   NewNode * MyNewNode = new NewNode;
   cout << "enter a number: " << endl;
   cin >> MyNewNode->i;

   if (*ptr == 0)
      *ptr = MyNewNode;
   else  {
      walker = *ptr;
      while (walker->node != 0)
         walker = walker->node;

      walker->node = MyNewNode;
   }
}

int main() {
```

```
    char ans = ' ';
    NewNode * ptr = 0;
    do {
       cout << "a (add node)  d (display list)  q (quit)" << endl;
       cin >> ans;
       switch (ans) {
       case 'a':
          AddNode(&ptr);
          break;
       case 'd':
          WalkList(ptr);
          break;
       }
    } while (ans != 'q');
 }
```

Input

```
a
45
d
a
789
d
q
```

Output

```
a (add node)  d (display list)  q (quit)
enter a number:
a (add node)  d (display list)  q (quit)
node 1 = 45
a (add node)  d (display list)  q (quit)
enter a number:
a (add node)  d (display list)  q (quit)
node 1 = 45
node 2 = 789
a (add node)  d (display list)  q (quit)
```

See Also

Reference

C++ Abstract Declarators

Addition of Pointer Types

Indirection Operator: *

Address-of Operator: &

const and volatile Pointers

The const and volatile keywords change how pointers are treated. The **const** keyword specifies that the pointer cannot be modified after initialization; the pointer is protected from modification thereafter.

The **volatile** keyword specifies that the value associated with the name that follows can be modified by actions other than those in the user application. Therefore, the **volatile** keyword is useful for declaring objects in shared memory that can be accessed by multiple processes or global data areas used for communication with interrupt service routines.

When a name is declared as **volatile**, the compiler reloads the value from memory each time it is accessed by the program. This dramatically reduces the possible optimizations. However, when the state of an object can change unexpectedly, it is the only way to ensure predictable program performance.

To declare the object pointed to by the pointer as **const** or **volatile**, use a declaration of the form:

```
const char *cpch;
volatile char *vpch;
```

To declare the value of the pointer — that is, the actual address stored in the pointer — as **const** or **volatile**, use a declaration of the form:

```
char * const pchc;
char * volatile pchv;
```

The C++ language prevents assignments that would allow modification of an object or pointer declared as **const**. Such assignments would remove the information that the object or pointer was declared with, thereby violating the intent of the original declaration. Consider the following declarations:

```
const char cch = 'A';
char ch = 'B';
```

Given the preceding declarations of two objects (`cch`, of type **const char**, and `ch`, of type **char)**, the following declaration/initializations are valid:

```
const char *pch1 = &cch;
const char *const pch4 = &cch;
const char *pch5 = &ch;
char *pch6 = &ch;
char *const pch7 = &ch;
const char *const pch8 = &ch;
```

The following declaration/initializations are erroneous.

```
char *pch2 = &cch;   // Error
char *const pch3 = &cch;   // Error
```

The declaration of `pch2` declares a pointer through which a constant object might be modified and is therefore disallowed. The declaration of `pch3` specifies that the *pointer* is constant, not the object; the declaration is disallowed for the same reason the `pch2` declaration is disallowed.

The following eight assignments show assigning through pointer and changing of pointer value for the preceding declarations; for now, assume that the initialization was correct for `pch1` through `pch8`.

```
*pch1 = 'A';  // Error: object declared const
pch1 = &ch;   // OK: pointer not declared const
*pch2 = 'A';  // OK: normal pointer
pch2 = &ch;   // OK: normal pointer
*pch3 = 'A';  // OK: object not declared const
pch3 = &ch;   // Error: pointer declared const
*pch4 = 'A';  // Error: object declared const
```

```
pch4 = &ch;   // Error: pointer declared const
```

Pointers declared as **volatile**, or as a mixture of **const** and **volatile**, obey the same rules.

Pointers to **const** objects are often used in function declarations as follows:

```
errno_t strcpy_s( char *strDestination, size_t numberOfElements, const char *strSource );
```

The preceding statement declares a function, strcpy_s, where two of the three arguments are of type pointer to **char**. Because the arguments are passed by reference and not by value, the function would be free to modify both `strDestination` and `strSource` if `strSource` were not declared as **const**. The declaration of `strSource` as **const** assures the caller that `strSource` cannot be changed by the called function.

Note

Because there is a standard conversion from *typename* ***** to **const** *typename* *****, it is legal to pass an argument of type **char *** to strcpy_s. However, the reverse is not true; no implicit conversion exists to remove the **const** attribute from an object or pointer.

A **const** pointer of a given type can be assigned to a pointer of the same type. However, a pointer that is not **const** cannot be assigned to a **const** pointer. The following code shows correct and incorrect assignments:

```
// const_pointer.cpp
int *const cpObject = 0;
int *pObject;

int main() {
pObject = cpObject;
cpObject = pObject;   // C3892
}
```

The following sample shows how to declare an object as const if you have a pointer to a pointer to an object.

```
// const_pointer2.cpp
struct X {
   X(int i) : m_i(i) { }
   int m_i;
};

int main() {
   // correct
   const X cx(10);
   const X * pcx = &cx;
   const X ** ppcx = &pcx;

   // also correct
   X const cx2(20);
   X const * pcx2 = &cx2;
   X const ** ppcx2 = &pcx2;
}
```

See Also

Reference

Pointers

References (C++)

References may be declared using the following syntax:

```
[storage-class-specifiers] [cv-qualifiers] type-specifiers
[ms-modifier] declarator [= expression];
```

Any valid declarator specifying a reference may be used. Unless the reference is a reference to function or array type, the following simplified syntax applies:

```
[storage-class-specifiers] [cv-qualifiers] type-specifiers &
[cv-qualifiers] identifier [= expression];
```

References are declared using the following sequence:

1. The declaration specifiers:

- An optional storage class specifier.
- Optional **const** and/or **volatile** qualifiers.
- The type specifier: the name of a type.
- 2. The declarator:
- An optional Microsoft specific modifier. For more information, see Microsoft-Specific Modifiers.
- The & operator.
- Optional **const** and/or **volatile** qualifers.
- The identifier.

3. An optional initializer.

The more complex declarator forms for pointers to arrays and functions also apply to references to arrays and functions, see pointers and declarators.

Multiple declarators and initializers may appear in a comma-separated list following a single declaration specifier. For example:

```
int &i;
int &i, &j;
```

References, pointers and objects may be declared together:

```
int &ref, *ptr, k;
```

A reference holds the address of an object, but behaves syntactically like an object.

In the following program, notice that the name of the object, `Today`, and the reference to the object, `TodayRef`, can be used identically in programs:

Example

```
// references.cpp
#include <stdio.h>
struct S {
   short i;
};

int main() {
   S  s;   // Declare the object.
   S& SRef = s;   // Declare the reference.
   s.i = 3;
```

```
    printf_s("%d\n", s.i);
    printf_s("%d\n", SRef.i);

    SRef.i = 4;
    printf_s("%d\n", s.i);
    printf_s("%d\n", SRef.i);
}
```

Output

```
3
3
4
4
```

Comment

Topics in this section:

- Reference-Type Function Arguments
- Reference-Type Function Returns
- References to Pointers

See Also
Reference
Initializing References

Reference-Type Function Arguments

It is often more efficient to pass references, rather than large objects, to functions. This allows the compiler to pass the address of the object while maintaining the syntax that would have been used to access the object. Consider the following example that uses the `Date` structure:

```
// reference_type_function_arguments.cpp
struct Date
{
short DayOfWeek;
short Month;
short Day;
short Year;
};

// Create a Julian date of the form DDDYYYY
// from a Gregorian date.
long JulianFromGregorian( Date& GDate )
{
static int cDaysInMonth[] = {
31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
   };
long JDate = 0;
// Add in days for months already elapsed.
for ( int i = 0; i < GDate.Month - 1; ++i )
JDate += cDaysInMonth[i];
// Add in days for this month.
JDate += GDate.Day;

// Check for leap year.
if ( GDate.Year % 100 != 0 && GDate.Year % 4 == 0 )
JDate++;
// Add in year.
JDate *= 10000;
JDate += GDate.Year;

return JDate;
}

int main()
{
}
```

The preceding code shows that members of a structure passed by reference are accessed using the member-selection operator (**.**) instead of the pointer member-selection operator (**->**).

Although arguments passed as reference types observe the syntax of non-pointer types, they retain one important characteristic of pointer types: they are modifiable unless declared as **const**. Because the intent of the preceding code is not to modify the object `GDate`, a more appropriate function prototype is:

```
long JulianFromGregorian( const Date& GDate );
```

This prototype guarantees that the function `JulianFromGregorian` will not change its argument.

Any function prototyped as taking a reference type can accept an object of the same type in its place because there is a standard conversion from *typename* to *typename***&**.

See Also

Reference

References (C++)

Reference-Type Function Returns

Functions can be declared to return a reference type. There are two reasons to make such a declaration:

- The information being returned is a large enough object that returning a reference is more efficient than returning a copy.
- The type of the function must be an l-value.

Just as it can be more efficient to pass large objects *to* functions by reference, it also can be more efficient to return large objects *from* functions by reference. Reference-return protocol eliminates the necessity of copying the object to a temporary location prior to returning.

Reference-return types can also be useful when the function must evaluate to an l-value. Most overloaded operators fall into this category, particularly the assignment operator. Overloaded operators are covered in Overloaded Operators.

Example

Consider the `Point` example:

```
// refType_function_returns.cpp
// compile with: /EHsc

#include <iostream>
using namespace std;

class Point
{
public:
// Define "accessor" functions as
//  reference types.
unsigned& x();
unsigned& y();
private:
unsigned obj_x;
unsigned obj_y;
};

unsigned& Point :: x()
{
return obj_x;
}
unsigned& Point :: y()
{
return obj_y;
}

int main()
{
Point ThePoint;
// Use x() and y() as l-values.
ThePoint.x() = 7;
ThePoint.y() = 9;

// Use x() and y() as r-values.
cout << "x = " << ThePoint.x() << "\n"
<< "y = " << ThePoint.y() << "\n";
}
```

Output

```
x = 7
y = 9
```

Notice that the functions `x` and `y` are declared as returning reference types. These functions can be used on either side of an assignment statement.

Declarations of reference types must contain initializers except in the following cases:

- Explicit **extern** declaration
- Declaration of a class member
- Declaration within a class
- Declaration of an argument to a function or the return type for a function

See Also

Reference

References (C++)

References to Pointers

References to pointers can be declared in much the same way as references to objects. Declaring a reference to a pointer yields a modifiable value that is used like a normal pointer.

Example

The following code samples illustrate the difference between using a pointer to a pointer and a reference to a pointer.

Functions `Add1` and `Add2` are functionally equivalent (although they are not called the same way). The difference is that `Add1` uses double indirection whereas `Add2` uses the convenience of a reference to a pointer.

```
// references_to_pointers.cpp
// compile with: /EHsc

#include <iostream>
#include <string>

// STL namespace
using namespace std;

enum {
   sizeOfBuffer = 132
};

// Define a binary tree structure.
struct BTree {
   char  *szText;
   BTree *Left;
   BTree *Right;
};

// Define a pointer to the root of the tree.
BTree *btRoot = 0;

int Add1( BTree **Root, char *szToAdd );
int Add2( BTree*& Root, char *szToAdd );
void PrintTree( BTree* btRoot );

int main( int argc, char *argv[] ) {
   // Usage message
   if( argc < 2 ) {
      cerr << "Usage: Refptr [1 | 2]" << "\n";
      cerr << "\nwhere:\n";
      cerr << "1 uses double indirection\n";
      cerr << "2 uses a reference to a pointer.\n";
      cerr << "\nInput is from stdin.\n";
      return 1;
   }

   char *szBuf = new char[sizeOfBuffer];
   if (szBuf == NULL) {
      cerr << "Out of memory!\n";
      return -1;
   }

   // Read a text file from the standard input device and
   //  build a binary tree.
   //while( !cin.eof() )
   {
      cin.get( szBuf, sizeOfBuffer, '\n' );
      cin.get();

      if ( strlen( szBuf ) ) {
         switch ( *argv[1] ) {
            // Method 1: Use double indirection.
```

```
            case '1':
               Add1( &btRoot, szBuf );
               break;
            // Method 2: Use reference to a pointer.
            case '2':
               Add2( btRoot, szBuf );
               break;
            default:
               cerr << "Illegal value '"
                  << *argv[1]
                  << "' supplied for add method.\n"
                     << "Choose 1 or 2.\n";
               return -1;
         }
      }
   }
   // Display the sorted list.
   PrintTree( btRoot );
}

// PrintTree: Display the binary tree in order.
void PrintTree( BTree* MybtRoot ) {
   // Traverse the left branch of the tree recursively.
   if ( btRoot->Left )
      PrintTree( btRoot->Left );

   // Print the current node.
   cout << btRoot->szText << "\n";

   // Traverse the right branch of the tree recursively.
   if ( btRoot->Right )
      PrintTree( btRoot->Right );
}

// Add1: Add a node to the binary tree.
//       Uses double indirection.
int Add1( BTree **Root, char *szToAdd ) {
   if ( (*Root) == 0 ) {
      (*Root) = new BTree;
      (*Root)->Left = 0;
      (*Root)->Right = 0;
      (*Root)->szText = new char[strlen( szToAdd ) + 1];
      strcpy_s((*Root)->szText, (strlen( szToAdd ) + 1), szToAdd );
      return 1;
   }
   else {
      if ( strcmp( (*Root)->szText, szToAdd ) > 0 )
         return Add1( &((*Root)->Left), szToAdd );
      else
         return Add1( &((*Root)->Right), szToAdd );
   }
}

// Add2: Add a node to the binary tree.
//       Uses reference to pointer
int Add2( BTree*& Root, char *szToAdd ) {
   if ( Root == 0 ) {
      Root = new BTree;
      Root->Left = 0;
      Root->Right = 0;
      Root->szText = new char[strlen( szToAdd ) + 1];
      strcpy_s( Root->szText, (strlen( szToAdd ) + 1), szToAdd );
      return 1;
   }
   else {
      if ( strcmp( Root->szText, szToAdd ) > 0 )
         return Add2( Root->Left, szToAdd );
      else
```

```
            return Add2( Root->Right, szToAdd );
        }
    }
```

Output

```
Usage: Refptr [1 | 2]

where:
1 uses double indirection
2 uses a reference to a pointer.

Input is from stdin.
```

See Also

Reference

References (C++)

Pointers to Members

Declarations of pointers to members are special cases of pointer declarations. They are declared using the following sequence:

```
[storage-class-specifiers] [cv-qualifiers] type-specifiers [ms-modifier]
qualified-name ::* [cv-qualifiers] identifier
[= & qualified-name :: member-name];
```

1. The declaration specifier:
   - An optional storage class specifier.
   - Optional **const** and/or **volatile** specifiers.
   - The type specifier: the name of a type. This is the type of the member to be pointed to, not the class.
2. The declarator:
   - An optional Microsoft specific modifier. For more information, see Microsoft-Specific Modifiers.
   - The qualified name of the class containing the members to be pointed to. See Names and Qualified Names.
   - The :: operator.
   - The ***** operator.
   - Optional **const** and/or **volatile** specifiers.
   - The identifier naming the pointer to member.
   - An optional initializer:

The **=** operator.

The **&** operator.

The qualified name of the class.

The **::** operator.

The name of a nonstatic member of the class of the appropriate type.

As always, multiple declarators (and any associated initializers) are allowed in a single declaration.

A pointer to a member of a class differs from a normal pointer because it has type information for the type of the member and for the class to which the member belongs. A normal pointer identifies (has the address of) only a single object in memory. A pointer to a member of a class identifies that member in any instance of the class. The following example declares a class, `Window`, and some pointers to member data.

```
// pointers_to_members1.cpp
class Window
{
public:
   Window();                              // Default constructor.
   Window( int x1, int y1,                // Constructor specifying
   int x2, int y2 );                      //  window size.
bool SetCaption( const char *szTitle ); // Set window caption.
   const char *GetCaption();              // Get window caption.
   char *szWinCaption;                    // Window caption.
};

// Declare a pointer to the data member szWinCaption.
char * Window::* pwCaption = &Window::szWinCaption;
int main()
{
}
```

In the preceding example, `pwCaption` is a pointer to any member of class `Window` that has type **char***. The type of `pwCaption` is `char * Window::*`. The next code fragment declares pointers to the `SetCaption` and `GetCaption` member functions.

```
const char * (Window::*pfnwGC)() = &Window::GetCaption;
bool (Window::*pfnwSC)( const char * ) = &Window::SetCaption;
```

The pointers `pfnwGC` and `pfnwSC` point to `GetCaption` and `SetCaption` of the `Window` class, respectively. The code copies information to the window caption directly using the pointer to member `pwCaption`:

```
Window wMainWindow;
Window *pwChildWindow = new Window;
char   *szUntitled    = "Untitled -  ";
int    cUntitledLen   = strlen( szUntitled );

strcpy_s( wMainWindow.*pwCaption, cUntitledLen, szUntitled );
(wMainWindow.*pwCaption)[cUntitledLen - 1] = '1';     //same as
//wMainWindow.SzWinCaption [cUntitledLen - 1] = '1';
strcpy_s( pwChildWindow->*pwCaption, cUntitledLen, szUntitled );
(pwChildWindow->*pwCaption)[cUntitledLen - 1] = '2'; //same as //pwChildWindow->szWinCaptio
n[cUntitledLen - 1] = '2';
```

The difference between the **.*** and **->*** operators (the pointer-to-member operators) is that the **.*** operator selects members given an object or object reference, while the **->*** operator selects members through a pointer. (For more about these operators, see Expressions with Pointer-to-Member Operators.)

The result of the pointer-to-member operators is the type of the member — in this case, **char ***.

The following code fragment invokes the member functions `GetCaption` and `SetCaption` using pointers to members:

```
// Allocate a buffer.
enum {
    sizeOfBuffer = 100
};
char szCaptionBase[sizeOfBuffer];

// Copy the main window caption into the buffer
//  and append " [View 1]".
strcpy_s( szCaptionBase, sizeOfBuffer, (wMainWindow.*pfnwGC)() );
strcat_s( szCaptionBase, sizeOfBuffer, " [View 1]" );
// Set the child window's caption.
(pwChildWindow->*pfnwSC)( szCaptionBase );
```

See Also

Reference

C++ Abstract Declarators

Restrictions on Pointers to Members

The address of a static member is not a pointer to a member. It is a regular pointer to the one instance of the static member. Because only one instance of a static member exists for all objects of a given class, the ordinary address-of **(&)** and dereference **(*)** operators can be used.

See Also

Reference

Pointers to Members

Pointers to Members and Virtual Functions

Invoking a virtual function through a pointer-to-member function works as if the function had been called directly; the correct function is looked up in the v-table and invoked.

Remarks

The key to virtual functions working, as always, is invoking them through a pointer to a base class. (For more information about virtual functions, see Virtual Functions.)

Example

The following code shows how to invoke a virtual function through a pointer-to-member function:

```
// virtual_functions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Base
{
public:
virtual void Print();
};
void (Base ::* bfnPrint)() = &Base :: Print;
void Base :: Print()
{
cout << "Print function for class Base\n";
}

class Derived : public Base
{
public:
void Print();  // Print is still a virtual function.
};

void Derived :: Print()
{
cout << "Print function for class Derived\n";
}

int main()
{
    Base   *bPtr;
    Base    bObject;
Derived dObject;
bPtr = &bObject;    // Set pointer to address of bObject.
(bPtr->*bfnPrint)();
bPtr = &dObject;    // Set pointer to address of dObject.
(bPtr->*bfnPrint)();
}
```

Output

```
Print function for class Base
Print function for class Derived
```

See Also

Reference

Pointers to Members

Inheritance Keywords

Microsoft Specific

```
      class [__single_inheritance] class-name;
class [__multiple_inheritance] class-name;
class [__virtual_inheritance] class-name;
```

where:

class-name

The name of the class being declared.

C++ allows you to declare a pointer to a class member prior to the definition of the class. For example:

```
class S;
int S::*p;
```

In the code above, `p` is declared to be a pointer to integer member of class S. However, `class S` has not yet been defined in this code; it has only been declared. When the compiler encounters such a pointer, it must make a generalized representation of the pointer. The size of the representation is dependent on the inheritance model specified. There are four ways to specify an inheritance model to the compiler:

- In the IDE under **Pointer-to-member representation**
- At the command line using the /vmg switch
- Using the pointers_to_members pragma
- Using the inheritance keywords **__single_inheritance**, **__multiple_inheritance**, and **__virtual_inheritance**. This technique controls the inheritance model on a per-class basis.

Note
If you always declare a pointer to a member of a class after defining the class, you don't need to use any of these optio ns.

Declaring a pointer to a member of a class prior to the class definition affects the size and speed of the resulting executable file. The more complex the inheritance used by a class, the greater the number of bytes required to represent a pointer to a member of the class and the larger the code required to interpret the pointer. Single inheritance is least complex, and virtual inheritance is most complex.

If the example above is changed to:

```
class __single_inheritance S;
int S::*p;
```

regardless of command-line options or pragmas, pointers to members of `class S` will use the smallest possible representation.

Note
The same forward declaration of a class pointer-to-member representation should occur in every translation unit that declar es pointers to members of that class, and the declaration should occur before the pointers to members are declared.

END Microsoft Specific

See Also

Reference

Arrays (C++)

An array is a collection of like objects. The simplest case of an array is a vector, which may be declared by the following sequence:

```
      decl-specifier identifier [ constant-expression ]
decl-specifier identifier []
decl-specifier identifer [][ constant-expression] . . .
decl-specifier identifier [ constant-expression ]
[ constant-expression ] . . .
```

1. The declaration specifier:

- An optional storage class specifier.
- Optional **const** and/or **volatile** specifiers.
- The type name of the elements of the array.

2. The declarator:

- The identifier.
- A constant expression of integral type enclosed in brackets, **[].** If multiple dimensions are declared using additional brackets, the constant expression may be omitted on the first set of brackets.
- Optional additional brackets enclosing constant expressions.

3. An optional initializer. See Initializers.

The number of elements in the array is given by the constant expression. The first element in the array is the 0th element, and the last element is the (*n*-1) element, where *n* is the number of elements the array can contain. The *constant-expression* must be of an integral type and must be greater than 0. A zero-sized array is legal only when the array is the last field in a **struct** or **union** and when the Microsoft extensions (/Ze) are enabled.

The following example shows how to define an array at run time:

```
// arrays.cpp
// compile with: /EHsc
#include <iostream>

int main() {
   using namespace std;
   int size = 3, i = 0;

   int* myarr = new int[size];

   for (i = 0 ; i < size ; i++)
      myarr[i] = 10;

   for (i = 0 ; i < size ; i++)
      printf_s("myarr[%d] = %d\n", i, myarr[i]);

   delete [] myarr;
}
```

Arrays are derived types and can therefore be constructed from any other derived or fundamental type except functions, references, and **void**.

Arrays constructed from other arrays are multidimensional arrays. These multidimensional arrays are specified by placing multiple bracketed constant expressions in sequence. For example, consider this declaration:

```
int i2[5][7];
```

It specifies an array of type **int**, conceptually arranged in a two-dimensional matrix of five rows and seven columns, as shown in the following figure:

Conceptual Layout of Multidimensional Array

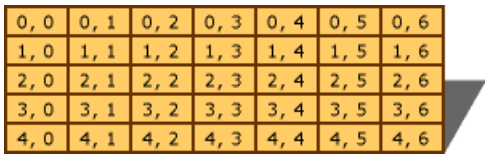

In declarations of multidimensioned arrays that have an initializer list (as described in Initializers), the constant expression that specifies the bounds for the first dimension can be omitted. For example:

```
// arrays2.cpp
// compile with: /c
const int cMarkets = 4;
// Declare a float that represents the transportation costs.
double TransportCosts[][cMarkets] = {
   { 32.19, 47.29, 31.99, 19.11 },
   { 11.29, 22.49, 33.47, 17.29 },
   { 41.97, 22.09,  9.76, 22.55 }
};
```

The preceding declaration defines an array that is three rows by four columns. The rows represent factories and the columns represent markets to which the factories ship. The values are the transportation costs from the factories to the markets. The first dimension of the array is left out, but the compiler fills it in by examining the initializer.

Topics in this section:

- Using Arrays
- Arrays in Expressions
- Interpretation of Subscript Operator
- Indirection on Array Types
- Ordering of C++ Arrays

Example

The technique of omitting the bounds specification for the first dimension of a multidimensional array can also be used in function declarations as follows:

```
// multidimensional_arrays.cpp
// compile with: /EHsc
// arguments: 3
#include <limits>   // Includes DBL_MAX
#include <iostream>

const int cMkts = 4, cFacts = 2;

// Declare a float that represents the transportation costs
double TransportCosts[][cMkts] = {
   { 32.19, 47.29, 31.99, 19.11 },
   { 11.29, 22.49, 33.47, 17.29 },
   { 41.97, 22.09,  9.76, 22.55 }
};

// Calculate size of unspecified dimension
const int cFactories = sizeof TransportCosts /
                  sizeof( double[cMkts] );

double FindMinToMkt( int Mkt, double myTransportCosts[][cMkts], int mycFacts);

using namespace std;
```

```
int main( int argc, char *argv[] ) {
   double MinCost;

   if (argv[1] == 0) {
      cout << "You must specify the number of markets." << endl;
      exit(0);
   }
   MinCost = FindMinToMkt( *argv[1] - '0', TransportCosts, cFacts);
   cout << "The minimum cost to Market " << argv[1] << " is: "
       << MinCost << "\n";
}

double FindMinToMkt(int Mkt, double myTransportCosts[][cMkts], int mycFacts) {
   double MinCost = DBL_MAX;

   for( int i = 0; i < cFacts; ++i )
      MinCost = (MinCost < TransportCosts[i][Mkt]) ?
         MinCost : TransportCosts[i][Mkt];

   return MinCost;
}
```

Output

```
The minimum cost to Market 3 is: 17.29
```

Comments

The function `FindMinToMkt` is written such that adding new factories does not require any code changes, just a recompilation.

See Also

Reference

C++ Abstract Declarators

Using Arrays (C++)

Individual elements of arrays are accessed using the array subscript operator (**[ ]**). If a singly dimensioned array is used in an expression with no subscript, the array name evaluates to a pointer to the first element in the array. For example:

```
// using_arrays.cpp
int main() {
   char chArray[10];
   char *pch = chArray;   // Pointer to first element.
   char   ch = chArray[0];   // Value of first element.
   ch = chArray[3];   // Value of fourth element.
}
```

When using multidimensional arrays, various combinations are acceptable in expressions. The following example illustrates this:

```
// using_arrays_2.cpp
// compile with: /EHsc /W1
#include <iostream>
using namespace std;
int main() {
   double multi[4][4][3];   // Declare the array.
   double (*p2multi)[3];
   double (*p1multi);

   cout << multi[3][2][2] << "\n";   // C4700 Use three subscripts.
   p2multi = multi[3];               // Make p2multi point to
                                     // fourth "plane" of multi.
   p1multi = multi[3][2];            // Make p1multi point to
                                     // fourth plane, third row
                                     // of multi.
}
```

In the preceding code, `multi` is a three-dimensional array of type **double**. The `p2multi` pointer points to an array of type **double** of size three. The array is used with one, two, and three subscripts in this example. Although it is more common to specify all the subscripts, as in the **cout** statement, it is sometimes useful to select a specific subset of array elements as shown in the succeeding statements.

See Also

Reference

Arrays (C++)

Arrays in Expressions

When an identifier of an array type appears in an expression other than **sizeof**, address-of (**&**), or initialization of a reference, it is converted to a pointer to the first array element. For example:

```
char szError1[] = "Error: Disk drive not ready.";
char *psz = szError1;
```

The pointer `psz` points to the first element of the array `szError1`. Note that arrays, unlike pointers, are not modifiable l-values. Therefore, the following assignment is illegal:

```
szError1 = psz;
```

See Also

Reference

Arrays (C++)

Interpretation of Subscript Operator

Like other operators, the subscript operator (**[ ]**) can be redefined by the user. The default behavior of the subscript operator, if not overloaded, is to combine the array name and the subscript using the following method:

*((*array-name*) + (*subscript*))

As in all addition that involves pointer types, scaling is performed automatically to adjust for the size of the type. Therefore, the resultant value is not *subscript* bytes from the origin of *array-name*; rather, it is the *subscript*th element of the array. (For more information about this conversion, see Additive Operators.)

Similarly, for multidimensional arrays, the address is derived using the following method:

((*array-name*) + (*subscript* 1 *max*2 * *max*3...*max*n) **+** *subscript*2 * *max*3...*max*n) . . . + *subscript*n))

See Also
Reference
Arrays (C++)

Indirection on Array Types

Use of the indirection operator (*****) on an *n*-dimensional array type yields an *n*–1 dimensional array. If *n* is 1, a scalar (or array element) is yielded.

See Also

Reference

Arrays (C++)

Ordering of C++ Arrays

C++ arrays are stored in row-major order. Row-major order means the last subscript varies the fastest.

See Also

Reference

Arrays (C++)

Function Declarations

This section includes the following topics:

- Function declaration syntax
    - Variable argument lists
    - Declaring functions that take no arguments
    - Function overloading
    - Restrictions on functions
    - The argument declaration list
    - Argument lists in function prototypes (nondefining declaration)
    - Argument lists in function definitions
- Default arguments
    - Default argument expressions
    - Other considerations

Function definition is covered in Function Definitions.

See Also
Reference
C++ Abstract Declarators

Function Declaration Syntax

Functions may be global or members of a class or struct. They are declared using the following sequence:

```
[friend] [storage-class-specifier] [function-specifier]
[[cv-qualifer] type-specifier] declarator [cv-qualifer]
[exception-spec]
[friend] [storage-class-specifier] [function-specifier]
[[cv-qualifer] type-specifier] [*|&] [ms-modifer] identifier
( argument-list ) [cv-qualifer] [exception-spec]
```

1. The declaration specifier. See Specifiers.
   - Within a class, an optional **friend** specifier.
   - An optional storage class specifier. For global functions, allowable storage class specifiers are **static** and **extern**. For class member functions, **extern** is not allowed.
   - An optional function specifier. For global functions, the only allowable function specifier is **inline.** For class member functions, **virtual** and **inline** are allowed and, for constructors, **explicit**.
   - A type specifier indicating the return type (subject to modification to a pointer or reference type by the use of * or &).
2. The declarator.
   - Optional pointer or reference operator(s) modifying the return type
   - An optional Microsoft specific modifier specifying the calling convention. See Microsoft Specific Modifiers.
   - The name of the function. If the function is a member of a class or struct, the name may be qualified using the scope resolution operator.
   - The argument declaration list enclosed in parentheses **()**.
   - An optional const or volatile qualifier. In this context, **const** may only be used for class members, and is used to indicate that the function will not modify data members of the class.

If the return type is a function pointer, the syntax of the declarator may be more complex, as explained below.

1. An optional exception specification indicating what exception(s) the function throws. See Exception Specifications.

The type of the identifier so declared may be constructed as follows: const or volatile (if specified) function, taking the specified argument list, returning the type indicated by the declaration specifier.

The following example shows two simple function declarations:

```
char *strchr( char *dest, char *src );
static int atoi( const char *ascnum ) const;
```

Declaring Functions Returning Pointers to Functions

It is recommended that **typedef** be used to declare an alias for the function pointer type if declaring a function that returns a function pointer type. For example

```
typedef int (*fp)(int);
fp myFunction(char* s); // function returning function pointer
```

If this is not done, the proper syntax for the function declaration may be deduced from the declarator syntax for the function pointer by replacing the identifier (`fp` in the above example) with the functions name and argument list, as follows:

```
int (*myFunction(char* s))(int);
```

The preceding declaration is equivalent to the declaration using typedef above.

See Also

Reference

Function Declarations

Variable Argument Lists

Function declarations in which the last member of *argument-declaration-list* is the ellipsis (...) can take a variable number of arguments. In these cases, C++ provides type checking only for the explicitly declared arguments. You can use variable argument lists when you need to make a function so general that even the number and types of arguments can vary. The **printf** family of functions is an example of functions that use variable argument lists.

To access arguments after those declared, use the macros contained in the standard include file STDARG.H as described in Functions with Variable Argument Lists.

Microsoft Specific

Microsoft C++ allows the ellipsis to be specified as an argument if the ellipsis is the last argument and the ellipsis is preceded by a comma. Therefore, the declaration `int Func( int i, ... );` is legal, but `int Func( int i ... );` is not.

END Microsoft Specific

Declaration of a function that takes a variable number of arguments requires at least one placeholder argument, even if it is not used. If this placeholder argument is not supplied, there is no way to access the remaining arguments.

When arguments of type **char** are passed as variable arguments, they are converted to type **int**. Similarly, when arguments of type **float** are passed as variable arguments, they are converted to type **double**. Arguments of other types are subject to the usual integral and floating-point promotions. See Integral Promotions for more information.

See Also
Reference
Function Declarations

Declaring Functions That Take No Arguments

A function declared with the single keyword **void** in the argument declaration list takes no arguments, as long as the keyword **void** is the first and only member of the argument declaration list. Arguments of type **void** elsewhere in the list produce errors. For example:

```
long GetTickCount( void );            // OK
long GetTickCount( int Reset, void ); // Error
long GetTickCount( void, int Reset ); // Error
```

In C++, explicitly specifying that a function requires no arguments is the same as declaring a function with an empty argument declaration list. Therefore, the following two statements are identical:

```
long GetTickCount();
long GetTickCount( void );
```

Note that, while it is illegal to specify a **void** argument except as outlined here, types derived from type **void** (such as pointers to **void** and arrays of **void**) can appear anywhere the argument declaration list.

See Also

Reference

Function Declarations

Function Overloading

C++ allows specification of more than one function of the same name in the same scope. These are called overloaded functions and are described in detail in Overloading. Overloaded functions enable programmers to supply different semantics for a function, depending on the types and number of arguments.

For example, a **print** function that takes a string (or **char ***) argument performs very different tasks than one that takes an argument of type **double**. Overloading permits uniform naming and prevents programmers from having to invent names such as `print_sz` or `print_d`. The following table shows what parts of a function declaration C++ uses to differentiate between groups of functions with the same name in the same scope.

Overloading Considerations

Function Declaration Element	Used for Overloading?
Function return type	No
Number of arguments	Yes
Type of arguments	Yes
Presence or absence of ellipsis	Yes
Use of **typedef** names	No
Unspecified array bounds	No
const or **volatile** (see below)	Yes

Although functions can be distinguished on the basis of return type, they cannot be overloaded on this basis. **Const** or **volatile** are only used as a basis for overloading if they are used in a class to apply to the **this** pointer for the class, not the function's return type. In other words, overloading applies only if the **const** or **volatile** keyword follows the function's argument list in the declaration.

Example

The following example illustrates how overloading can be used. Another way to solve the same problem is presented in Default Arguments.

```
// function_overloading.cpp
// compile with: /EHsc
#include <iostream>
#include <math.h>

// Prototype three print functions.
int print( char *s );                  // Print a string.
int print( double dvalue );            // Print a double.
int print( double dvalue, int prec );  // Print a double with a
//  given precision.
using namespace std;
int main( int argc, char *argv[] )
{
const double d = 893094.2987;
if( argc < 2 )
    {
// These calls to print invoke print( char *s ).
print( "This program requires one argument." );
print( "The argument specifies the number of" );
print( "digits precision for the second number" );
print( "printed." );
exit(0);
    }
```

```
// Invoke print( double dvalue ).
print( d );

// Invoke print( double dvalue, int prec ).
print( d, atoi( argv[1] ) );
}

// Print a string.
int print( char *s )
{
cout << s << endl;
return cout.good();
}

// Print a double in default precision.
int print( double dvalue )
{
cout << dvalue << endl;
return cout.good();
}

// Print a double in specified precision.
//  Positive numbers for precision indicate how many digits
//  precision after the decimal point to show. Negative
//  numbers for precision indicate where to round the number
//  to the left of the decimal point.
int print( double dvalue, int prec )
{
// Use table-lookup for rounding/truncation.
static const double rgPow10[] = {
10E-7, 10E-6, 10E-5, 10E-4, 10E-3, 10E-2, 10E-1, 10E0,
10E1,  10E2,  10E3,  10E4, 10E5,  10E6
    };
const int iPowZero = 6;
// If precision out of range, just print the number.
if( prec < -6 || prec > 7 )
return print( dvalue );
// Scale, truncate, then rescale.
dvalue = floor( dvalue / rgPow10[iPowZero - prec] ) *
rgPow10[iPowZero - prec];
cout << dvalue << endl;
return cout.good();
}
```

The preceding code shows overloading of the `print` function in file scope.

For restrictions on overloading and information on how overloading affects other elements of C++, see Overloading.

See Also

Reference

Function Declarations

Restrictions on Functions

Functions cannot return arrays or functions. They can, however, return references or pointers to arrays or functions. Another way to return an array is to declare a structure with only that array as a member:

```
struct Address
{ char szAddress[31]; };

Address GetAddress();
```

It is illegal to define a type in either the return-type portion of a function declaration or in the declaration of any argument to a function. The following legal C code is illegal in C++:

```
enum Weather { Cloudy, Rainy, Sunny } GetWeather( Date Today )
```

The preceding code is disallowed because the type `Weather` has function scope local to `GetWeather` and the return value cannot be properly used. Because arguments to functions have function scope, declarations made within the argument list would have the same problem if not allowed.

C++ does not support arrays of functions. However, arrays of pointers to functions can be useful. In parsing a Pascal-like language, the code is often separated into a lexical analyzer that parses tokens and a parser that attaches semantics to the tokens. If the analyzer returns a particular ordinal value for each token, code can be written to perform appropriate processing as shown in this example:

```
// restrictions_to_functions.cpp
// The following functions are user-defined
int Error( char *szText) {return 1;}
int ProcessFORToken( char *szText ) {return 1;}
int ProcessWHILEToken( char *szText ){return 1;}
int ProcessBEGINToken( char *szText ){return 1;}
int ProcessENDToken( char *szText ){return 1;}
int ProcessIFToken( char *szText ){return 1;}
int ProcessTHENToken( char *szText ){return 1;}
int ProcessELSEToken( char *szText ){return 1;}

int (*ProcessToken[])( char * ) = {
   ProcessFORToken, ProcessWHILEToken, ProcessBEGINToken,
   ProcessENDToken, ProcessIFToken, ProcessTHENToken,
   ProcessELSEToken
};

const int MaxTokenID = sizeof ProcessToken / sizeof( int (*)(char*) );

int DoProcessToken( int TokenID, char *szText ) {
   if( TokenID < MaxTokenID )
      return (*ProcessToken[TokenID])( szText );
   else
      return Error( szText );
}

int main()
{
}
```

See Also

Reference

Function Declarations

The Argument Declaration List

The argument declaration list portion of a function declaration:

- Allows the compiler to check type consistency among the arguments the function requires and the arguments supplied in the call.
- Enables conversions, either implicit or user-defined, to be performed from the supplied argument type to the required argument type.
- Checks initializations of, or assignments to, pointers to functions.
- Checks initializations of, or assignments to, references to functions.

Multiple argument declarations are listed separated by commas.

Functions that may take a variable number of arguments are specified by using the ellipsis operator () in place of the variable arguments. If the ellipsis is present, it must be the last element in the argument list. See Variable Argument Lists.

Each argument declaration consists of:

- Declaration specifiers, including the type specifier. Specifically, this means:
  - A storage class specifier **auto** or **register.**
  - **const** and/or **volatile.**
  - The type specifier.
- A declarator, which may be abstract. A non-abstract declarator like allows the argument to be named; an abstract declarator specifies an anonymous argument. Specifically, this means:
  - An optional ***** or **&** specifying a pointer or reference.
  - An optional identifier naming the argument.

  OR

  - Any complex declarator specifying any composition of pointer, reference, function pointer, and array to fully specify the type.
- An initializer specifying the default value of the argument. Specifically, argument initializers consist of
  - The **=** operator.
  - An expression. There are several restrictions on these expressions. See

    Default Argument Expressions.

Note that many of the Microsoft-specific keywords can appear in the declaration specifiers and in the declaration of the name.

See Also
Reference
Function Declarations

Argument Lists in Function Prototypes (Nondefining Declaration)

The form *argument-declaration-list* is a list of the type names of the arguments. Consider an *argument-declaration-list* for a function, `func`, that takes these three arguments: pointer to type **char ***, **char**, and **int**.

The code for such an *argument-declaration-list* can be written:

```
char *, char, int
```

The function declaration (the prototype) might therefore be written:

```
void func( char *, char, int );
```

Although the preceding declaration contains enough information for the compiler to perform type checking and conversions, it does not provide much information about what the arguments are. A good way to document function declarations is to include identifiers as they would appear in the function definition, as in the following:

```
void func( char *szTarget, char chSearchChar, int nStartAt );
```

These identifiers in prototypes are useful only for default arguments, because they go out of scope immediately. However, they provide meaningful program documentation.

See Also
Reference
Function Declarations

Argument Lists in Function Definitions

The argument list in a function definition differs from that of a prototype only in that the identifiers, if present, represent formal arguments to the function. The identifier names need not match those in the prototype (if there are any).

Note
It is possible to define functions with unnamed arguments. However, these arguments are inaccessible to the functions for which they are defined.

See Also

Reference

Function Declarations

Default Arguments

In many cases, functions have arguments that are used so infrequently that a default value would suffice. To address this, the default-argument facility allows for specifying only those arguments to a function that are meaningful in a given call. To illustrate this concept, consider the example presented in Function Overloading.

```
// Prototype three print functions.
int print( char *s );                  // Print a string.
int print( double dvalue );            // Print a double.
int print( double dvalue, int prec );  // Print a double with a
//  given precision.
```

In many applications, a reasonable default can be supplied for `prec`, eliminating the need for two functions:

```
// Prototype two print functions.
int print( char *s );                     // Print a string.
int print( double dvalue, int prec=2 );  // Print a double with a
//  given precision.
```

The implementation of the `print` function is changed slightly to reflect the fact that only one such function exists for type **double**:

```
// default_arguments.cpp
// compile with: /EHsc /c

// Print a double in specified precision.
//  Positive numbers for precision indicate how many digits
//  precision after the decimal point to show. Negative
//  numbers for precision indicate where to round the number
//  to the left of the decimal point.

#include <iostream>
#include <math.h>
using namespace std;

int print( double dvalue, int prec ) {
   // Use table-lookup for rounding/truncation.
   static const double rgPow10[] = {
      10E-7, 10E-6, 10E-5, 10E-4, 10E-3, 10E-2, 10E-1, 10E0,
         10E1,  10E2,  10E3,  10E4, 10E5,  10E6
   };
   const int iPowZero = 6;
   // If precision out of range, just print the number.
   if( prec >= -6 || prec <= 7 )
      // Scale, truncate, then rescale.
      dvalue = floor( dvalue / rgPow10[iPowZero - prec] ) *
      rgPow10[iPowZero - prec];
   cout << dvalue << endl;
   return cout.good();
}
```

To invoke the new `print` function, use code such as the following:

```
print( d );    // Precision of 2 supplied by default argument.
print( d, 0 ); // Override default argument to achieve other
//  results.
```

Note these points when using default arguments:

- Default arguments are used only in function calls where trailing arguments are omitted — they must be the last argument(s). Therefore, the following code is illegal:

```
int print( double dvalue = 0.0, int prec );
```

- A default argument cannot be redefined in later declarations even if the redefinition is identical to the original. Therefore, the following code produces an error:

```
// Prototype for print function.
int print( double dvalue, int prec = 2 );

...

// Definition for print function.
int print( double dvalue, int prec = 2 )
{
...
}
```

  The problem with this code is that the function declaration in the definition redefines the default argument for `prec`.

- Additional default arguments can be added by later declarations.
- Default arguments can be provided for pointers to functions. For example:

```
int (*pShowIntVal)( int i = 0 );
```

See Also

Reference

C++ Abstract Declarators

Default Argument Expressions

The expressions used for default arguments are often constant expressions, but this is not a requirement. The expression can combine functions that are visible in the current scope, constant expressions, and global variables. The expression cannot contain local variables or non-static class-member variables. The following code illustrates this:

```
BOOL CreateVScrollBar( HWND hWnd, short nWidth =
GetSystemMetrics( SM_CXVSCROLL ) );
```

The preceding declaration specifies a function that creates a vertical scroll bar of a given width for a window. If no width argument is supplied, the Windows API function, **GetSystemMetrics**, is called to find the default width for a scroll bar.

The default expression is evaluated after the function call, but the evaluation is completed before the function call actually takes place.

Because formal arguments to a function are in function scope, and because the evaluation of default arguments takes place prior to entry to this scope, you cannot use formal arguments, or local variables in default argument expressions.

Note that any formal argument declared before a default argument expression can hide a global name in the function scope, which can cause errors. The following code is illegal:

```
const int Categories = 9;
void EnumCategories( char *Categories[], int n = Categories );
```

In the preceding code, the global name `Categories` is hidden at function scope, making the default argument expression invalid.

See Also

Reference

Default Arguments

Other Considerations

The default argument is not considered part of the function type. Therefore, it is not used in selecting overloaded functions. Two functions that differ only in their default arguments are considered multiple definitions rather than overloaded functions.

Default arguments cannot be supplied for overloaded operators.

See Also

Reference

Default Arguments

C++ Function Definitions

Function definitions differ from function declarations in that they supply function bodies — the code that makes up the function. The form of a function definition is:

```
        decl-specifiers declarator [cv-qualifers] [exception-specification]
{
   // function body
}
```

The parts of the definition are:

- Declaration specifiers, as described in Function Declarations.
- The declarator. See below.
- An optional **const** or **volatile** qualifier. In this context, **const** may only be used for class members, and is used to indicate that the function will not modify data members of the class.
- Exception specification describing what exceptions the function may throw. See Exception Specifications.
- Function-body, consisting of statements enclosed in curly braces **{}**.

The form of the declarator is:

- Optional pointer or reference operators modifying the return type
- An optional Microsoft specific modifier. See Microsoft Specific Modifiers.
- The name of the function. If the function is a member of a class or struct, the name may be qualified using the scope resolution operator.
- The argument declaration list enclosed in parentheses **()**.
- For constructors, an optional constructor initializer (see below).

See the comments in Function Declarations on functions returning function pointers for information on the form of the declarator in such cases.

The formal arguments declared in the argument declaration list are in the scope of the function body.

The following figure shows the parts of a function definition. The shaded area is the function body.

Parts of a Function Definition

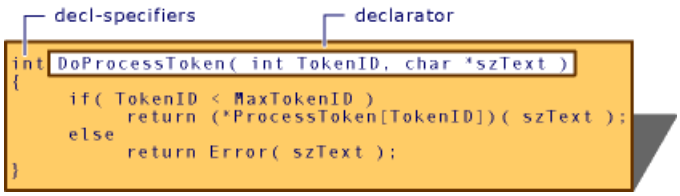

The constructor initializer element of the syntax is used only in constructors. Its purpose is to allow initialization of base classes and contained objects. (For more information about use of the constructor initializer, see Initializing Bases and Members.)

See Also
Reference
Declarators

Functions with Variable Argument Lists

Functions that require variable lists are declared using the ellipsis (...) in the argument list, as described in Variable Argument Lists. To access arguments passed to functions using this method, use the types and macros described in the STDARG.H standard include file in the CRT. For more information on these macros, see va_arg, va_end, va_start in the documentation for the C Run-Time Library.

Example

The following example shows how the **va_start**, **va_arg**, and **va_end** macros, along with the **va_list** type (declared in STDARG.H), work together:

```
// variable_argument_lists.cpp
#include <stdio.h>
#include <stdarg.h>

//  Declaration, but not definition, of ShowVar.
void ShowVar( char *szTypes, ... );
int main() {
   ShowVar( "fcsi", 32.4f, 'a', "Test string", 4 );
}

//  ShowVar takes a format string of the form
//   "ifcs", where each character specifies the
//   type of the argument in that position.
//
//  i = int
//  f = float
//  c = char
//  s = string (char *)
//
//  Following the format specification is a list
//  of n arguments, where n == strlen( szTypes ).
void ShowVar( char *szTypes, ... ) {
   va_list vl;
   int i;

   //  szTypes is the last argument specified; all
   //   others must be accessed using the variable-
   //   argument macros.
   va_start( vl, szTypes );

   // Step through the list.
   for( i = 0; szTypes[i] != '\0'; ++i ) {
      union Printable_t {
         int     i;
         float   f;
         char    c;
         char   *s;
      } Printable;

      switch( szTypes[i] ) {   // Type to expect.
         case 'i':
            Printable.i = va_arg( vl, int );
            printf_s( "%i\n", Printable.i );
         break;

         case 'f':
             Printable.f = va_arg( vl, double );
             printf_s( "%f\n", Printable.f );
         break;

         case 'c':
             Printable.c = va_arg( vl, char );
             printf_s( "%c\n", Printable.c );
```

```
            break;

            case 's':
                Printable.s = va_arg( vl, char * );
                printf_s( "%s\n", Printable.s );
            break;

            default:
            break;
        }
    }
    va_end( vl );
}
```

Output

```
32.400002
a
Test string
4
```

Comments

The preceding example illustrates these important concepts:

- A list marker must be established as a variable of type **va_list** before any variable arguments are accessed. In the preceding example, the marker is called `vl`.
- The individual arguments are accessed using the **va_arg** macro. The **va_arg** macro needs to be told the type of argument to retrieve so it can transfer the correct number of bytes from the stack. If an incorrect type of a size different than that supplied by the calling program is specified to **va_arg**, the results are unpredictable.
- The result obtained using the **va_arg** macro should be explicitly cast to the desired type.
- The **va_end** macro must be called to terminate variable-argument processing.

See Also

Reference

C++ Function Definitions

Initializers

Declarators can specify the initial value for objects. The only way to specify a value for objects of **const** type is in the declarator. The part of the declarator that specifies this initial value is called the initializer. There are two fundamental types of initializers:

- Initializers invoked using the equal-sign syntax, including aggregate initializers:

```
 = expression
= { expression-list }
= { { expression-list}, {expression-list}, . . . }
```

- Initializers invoked using function-style syntax:

```
              ( expression )
```

Only objects of classes with constructors can be initialized with the function-style syntax. The two syntax forms also differ in access control and in the potential use of temporary objects. Consider the following code, which illustrates some declarators with initializers:

```
int      i = 7;                  // Uses equal-sign syntax.
Customer Cust( "Taxpayer, Joe",  // Uses function-style
"14 Cherry Lane", //  syntax. Requires presence
"Manteca",        //  of a constructor.
"CA" );
```

Declarations of automatic, register, static, and external variables can contain initializers. However, declarations of external variables can contain initializers only if the variables are not declared as **extern**.

These initializers can contain expressions involving constants and variables in the current scope. The initializer expression is evaluated at the point the declaration is encountered in program flow, or, for global static objects and variables, at program startup. (For more information about initialization of global static objects, see Additional Startup Considerations.)

Topics in this section:

- Initializing Pointers to const Objects
- Uninitialized Objects
- Initializing Static Members
- Initializing Aggregates
- Initializing Character Arrays
- Initializing References

See Also
Reference
Declarators

Initializing Pointers to const Objects

A pointer to a **const** object can be initialized with a pointer to an object that is not **const**, but not vice versa. For example, the following initialization is legal:

```
Window StandardWindow;
const Window* pStandardWindow( &StandardWindow );
```

In the preceding code, the pointer `pStandardWindow` is declared as a pointer to a **const** object. Although `StandardWindow` is not declared as **const**, the declaration is acceptable because it does not allow an object not declared as **const** access to a **const** object. The reverse of this is as follows:

```
const Window StandardWindow;
Window* pStandardWindow( &StandardWindow );
```

The preceding code explicitly declares `StandardWindow` as a **const** object. Initializing the nonconstant pointer `pStandardWindow` with the address of `StandardWindow` generates an error because it allows access to the **const** object through the pointer. That is, it allows removal of the **const** attribute from the object.

See Also

Reference

Initializers

Uninitialized Objects

Objects and simple variables of storage class **static** that are declared with no initializer are guaranteed to be initialized to a bit pattern of zeros. No such special processing takes place for uninitialized objects of automatic or register storage classes. They have undefined values.

See Also

Reference

Initializers

Initializing Static Members

Static member initialization occurs in class scope. Therefore, they can access other member data or functions. For example:

```
// initializing_static_members.cpp
class DialogWindow
{
public:
static short  GetTextHeight()
    {
return 1;
    };
private:
static short nTextHeight;
};

short DialogWindow :: nTextHeight = GetTextHeight();
int main()
{
}
```

Note that in the preceding definition of the static member `nTextHeight`, `GetTextHeight` is implicitly known to be `DialogWindow :: GetTextHeight`.

See Also

Reference

Initializers

Initializing Aggregates

An aggregate type is an array, class, or structure type which:

- Has no constructors
- Has no nonpublic members
- Has no base classes
- Has no virtual functions

Initializers for aggregates can be specified as a comma-separated list of values enclosed in curly braces. For example, this code declares an **int** array of 10 and initializes it:

```
int rgiArray[10] = { 9, 8, 4, 6, 5, 6, 3, 5, 6, 11 };
```

The initializers are stored in the array elements in increasing subscript order. Therefore, `rgiArray[0]` is 9, `rgiArray[1]` is 8, and so on, until `rgiArray[9]`, which is 11. To initialize a structure, use code such as:

```
// initializing_aggregates.cpp
struct RCPrompt
{
short nRow;
short nCol;
char *szPrompt;
};
int main()
{
RCPrompt rcContinueYN = { 24, 0, "Continue (Y/N?)" };
}
```

See Also

Reference

Initializers

Length of Aggregate-Initializer Lists

If an aggregate initializer list is shorter than the array or class type that is being initialized, zeros are stored in the elements for which no initializer is specified. Therefore, the following two declarations are equivalent:

```
// Explicitly initialize all elements.
int rgiArray[5] = { 3, 2, 0, 0, 0 };

// Allow remaining elements to be zero-initialized.
int rgiArray[5] = { 3, 2 };
```

As this shows, initializer lists can be truncated, but supplying too many initializers generates an error.

See Also

Reference

Initializing Aggregates

Initializing Aggregates That Contain Aggregates

Some aggregates contain other aggregates — for example, arrays of arrays, arrays of structures, or structures that are composed of other structures. Initializers can be supplied for such constructs by initializing each one in the order it occurs with a brace-enclosed list. For example:

```
// Declare an array of type RCPrompt.
RCPrompt rgRCPrompt[4] =
{ { 4,  7, "Options Are:"  },
{ 6,  7, "1. Main Menu"  },
{ 8,  7, "2. Print Menu" },
{ 10, 7, "3. File Menu"  } };
```

Note that `rgRCPrompt` is initialized with a brace-enclosed list of brace-enclosed lists.

Example

The enclosed braces are not syntactically required, but they lend clarity to the declaration. The following example program shows how a two-dimensional array is filled by such an initializer:

```
// initializing_aggregates2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main()
{
int rgI[2][4] = { 1, 2, 3, 4, 5, 6, 7, 8 };
for( int i = 0; i < 2; ++i )
for( int j = 0; j < 4; ++j )
cout << "rgI[" << i << "][" << j << "] = "
<< rgI[i][j] << endl;
}
```

Output

```
rgI[0][0] = 1
rgI[0][1] = 2
rgI[0][2] = 3
rgI[0][3] = 4
rgI[1][0] = 5
rgI[1][1] = 6
rgI[1][2] = 7
rgI[1][3] = 8
```

Short initialization lists can be used only with explicit subaggregate initializers and enclosed in braces. If `rgI` had been declared as:

```
int rgI[2][4] = { { 1, 2 }, { 3, 4 } };
```

the program output would have been:

```
rgI[0][0] = 1
rgI[0][1] = 2
rgI[0][2] = 0
rgI[0][3] = 0
rgI[1][0] = 3
rgI[1][1] = 4
rgI[1][2] = 0
rgI[1][3] = 0
```

See Also

Reference

Initializing Aggregates

Initializing Incomplete Types

Incomplete types, such as unbounded array types, can be initialized as follows:

```
char HomeRow[] = { 'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l' };
```

The compiler computes the size of the array from the number of initializers provided.

Incomplete types, such as pointers to class types that are declared but not defined, are declared as follows:

```
class DefinedElsewhere;        // Class definition elsewhere.
class DefinedHere
{
    ...
friend class DefinedElsewhere;
};
```

See Also

Reference

Initializing Aggregates

Initializing Using Constructors

Objects of class type are initialized by calling the appropriate constructor for the class. For complete information about initializing class types, see Explicit Initialization.

See Also

Reference

Initializing Aggregates

Initializers and Unions

Objects of **union** type are initialized with a single value (if the union does not have a constructor). This is done in one of two ways:

- Initialize the union with another object of the same **union** type. For example:

```
// initializers_and_unions.cpp
// compile with: /W1
struct Point
{
unsigned x;
unsigned y;
};

union PtLong
{
long  l;
Point pt;
};

int main()
{
PtLong ptOrigin;
PtLong ptCurrent = ptOrigin;   // C4700
}
```

  In the preceding code, `ptCurrent` is initialized with the value of `ptOrigin` — an object of the same type.

- Initialize the union with a brace-enclosed initializer for the first member. For example:

```
PtLong ptCurrent = { 0x0a000aL };
```

See Also

Reference

Initializing Aggregates

Initializing Character Arrays

Character arrays can be initialized in one of two ways:

- Individually, as follows:

```
char chABCD[4] = { 'a', 'b', 'c', 'd' };
```

- With a string, as follows:

```
char chABCD[5] = "abcd";
```

In the second case, where the character array is initialized with a string, the compiler appends a trailing `'\0'` (end-of-string character). Therefore, the array must be at least one larger than the number of characters in the string.

Because most string handling uses the standard library functions or relies on the presence of the trailing end-of-string character, it is common to see unbounded array declarations initialized with strings:

```
char chABCD[] = "ABCD";
```

See Also

Reference

Initializers

Initializing References

Variables of reference type must be initialized with an object of the type from which the reference type is derived, or with an object of a type that can be converted to the type from which the reference type is derived. For example:

```
// initializing_references.cpp
int iVar;
long lVar;
int main() {
   long& LongRef1 = lVar;   // No conversion required.
   long& LongRef2 = iVar;   // C2440
   const long& LongRef3 = iVar;   // OK
   LongRef1 = 23L;   // Change lVar through a reference.
   LongRef2 = 11L;   // Change iVar through a reference.
   LongRef3 = 11L;   // C3892
}
```

The only way to initialize a reference with a temporary object is to initialize a constant temporary object. Once initialized, a reference-type variable always points to the same object; it cannot be modified to point to another object.

Although the syntax can be the same, initialization of reference-type variables and assignment to reference-type variables are semantically different. In the preceding example, the assignments that change `iVar` and `lVar` look similar to the initializations, but have different effects. The initialization specifies the object to which the reference-type variable points; the assignment assigns to the referred-to object through the reference.

Because both passing an argument of reference type to a function and returning a value of reference type from a function are initializations, the formal arguments to a function are initialized correctly, as are the references returned.

Reference-type variables can be declared without initializers only in the following:

- Function declarations (prototypes). For example:

  ```
  int func( int& );
  ```

- Function-return type declarations. For example:

  ```
  int& func( int& );
  ```

- Declaration of a reference-type class member. For example:

  ```
  class c {
  public:
     int& i;
  };
  ```

- Declaration of a variable explicitly specified as **extern**. For example:

  ```
  extern int& iVal;
  ```

When initializing a reference-type variable, the compiler uses the decision graph shown in the following figure to select between creating a reference to an object or creating a temporary object to which the reference points.

Decision Graph for Initialization of Reference Types

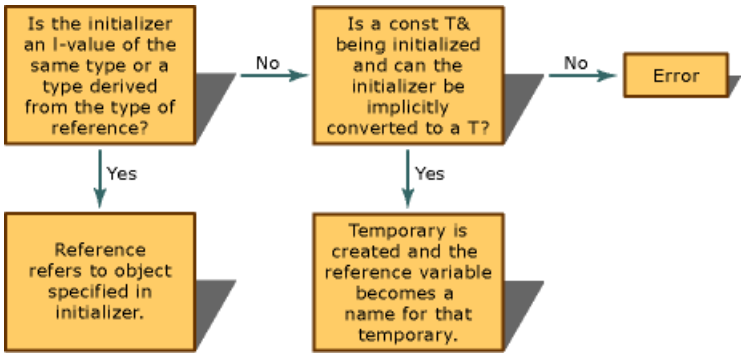

References to **volatile** types (declared as **volatile** *typename***&** *identifier*) can be initialized with **volatile** objects of the same type or with objects that have not been declared as **volatile**. They cannot, however, be initialized with **const** objects of that type. Similarly, references to **const** types (declared as **const** *typename***&** *identifier*) can be initialized with **const** objects of the same type (or anything that has a conversion to that type or with objects that have not been declared as **const**). They cannot, however, be initialized with **volatile** objects of that type.

References that are not qualified with either the **const** or **volatile** keyword can be initialized only with objects declared as neither **const** nor **volatile**.

See Also
Reference
Initializers

Classes, Structures, and Unions

This section introduces C++ classes. Classes, which can contain data and functions, introduce user-defined types into a program. User-defined types in traditional programming languages are collections of data which, taken together, describe an object's attributes and state. Class types in C++ enable you to describe attributes and state, and to define behavior.

The following topics are included:

- Overview
- Class names
- Class members
- Member functions
- Static data members
- Unions
- Bit fields
- Nested class declarations
- Type names in class scope

The three class types are structure, class, and union. They are declared using the struct, class, and union keywords (see Defining Class Types). The following table shows the differences among the three class types.

For information on managed classes and structs, see Classes and Structs.

Access Control and Constraints of Structures, Classes and Unions

Structures	**Classes**	**Unions**
class key is **struct**	class key is **class**	class key is **union**
Default access is public	Default access is private	Default access is public
No usage constraints	No usage constraints	Use only one member at a time

See Also

Other Resources

C++ Language Reference

Overview of Classes

Class types are defined using the class, struct, and union keywords. For simplicity, types defined with these keywords are called class types, except in discussions of language elements that behave differently depending on which keyword is used.

Names of classes defined within another class ("nested") have class scope of the enclosing class.

The name of a class may be either an identifier or a template identifier. See Templates.

The variables and functions of a class are called members. When defining a class, it is common practice to supply the following members (although all are optional):

- Class data members, which define the state and attributes of an object of the class type.
- One or more "constructor" functions, which initialize an object of the class type. Constructors are described in Constructors.
- A "destructor" function, which performs cleanup functions such as deallocating dynamically allocated memory or closing files. Destructors are described in Destructors.
- One or more member functions that define the object's behavior.

For more information, see:

- Defining Class Types
- Class-Type Objects

For information on managed classes and structs, see Classes and Structs.

See Also
Reference
Classes, Structures, and Unions

Defining Class Types

Class types are defined using class specifiers. Class types can be declared using elaborated type specifiers as shown in Type Specifiers. The class specifier consists of:

- The **class**, **struct** or **union** keyword.
- The class heading The class heading includes the class name and the applicable template heading.
- Optionally, a colon and the base list which identifies the base classes that the class being defined inherits from.
- Braces (**{}**) containing the class member declarations.

Class names are introduced as identifiers immediately after the compiler processes them (before entry into the class body); they can be used to declare class members. This allows declaration of self-referential data structures, such as the following:

```
// defining_class_types.cpp
// compile with: /c
class Tree
{
public:
   void *Data;
   Tree *Left;
   Tree *Right;
};
```

See Also

Reference

Overview of Classes

class (C++)

The **class** keyword declares a class type or defines an object of a class type.

```
[template-spec] class [ms-decl-spec] [tag [: base-list ]]
{
      member-list
} [declarators];
[ class ] tag declarators;
```

Parameters

template-spec

Optional template specifications. For more information, refer to Template Specifications.

class

The **class** keyword.

ms-decl-spec

Optional storage-class specification. For more information, refer to the __declspec keyword.

tag

The type name given to the class. The tag becomes a reserved word within the scope of the class. The tag is optional. If omitted, an anonymous class is defined. For more information, see Anonymous Class Types.

base-list

Optional list of classes or structures this class will derive its members from. See Base Classes for more information. Each base class or structure name can be preceded by an access specifier (public, private, protected) and the virtual keyword. See the member-access table in Controlling Access to Class Members for more information.

member-list

List of class members. Refer to Class Members for more information.

declarators

Declarator list specifying the names of the class. Declarator lists declare one or more instances of the class type. Declarators may include initializer lists if all data members of the class are **public**. This is more common in structures, whose data members are **public** by default, than in classes. See Overview of Declarators for more information.

Remarks

For more information on classes in general, refer to one of the following topics:

- struct
- union
- __multiple_inheritance
- __single_inheritance
- __virtual_inheritance

For information on managed classes and structs, see Classes and Structs

Example

```
// class.cpp
// compile with: /EHsc
// Example of the class keyword
// Exhibits polymorphism/virtual functions.

#include <iostream>
#include <string>
```

```
#define TRUE = 1
using namespace std;

class dog
{
public:
   dog()
   {
      _legs = 4;
      _bark = true;
   }

   void setDogSize(string dogSize)
   {
      _dogSize = dogSize;
   }
   virtual void setEars(string type)      // virtual function
   {
      _earType = type;
   }

private:
   string _dogSize, _earType;
   int _legs;
   bool _bark;

};

class breed : public dog
{
public:
   breed( string color, string size)
   {
      _color = color;
      setDogSize(size);
   }

   string getColor()
   {
      return _color;
   }

   // virtual function redefined
   void setEars(string length, string type)
   {
      _earLength = length;
      _earType = type;
   }

protected:
   string _color, _earLength, _earType;
};

int main()
{
   dog mongrel;
   breed labrador("yellow", "large");
   mongrel.setEars("pointy");
   labrador.setEars("long", "floppy");
   cout << "Cody is a " << labrador.getColor() << " labrador" << endl;
}
```

See Also

Reference

C++ Keywords

Classes, Structures, and Unions

struct (C++)

The **struct** keyword defines a structure type and/or a variable of a structure type.

```
[template-spec] struct [ms-decl-spec] [tag [: base-list ]]
{
      member-list
} [declarators];
[struct] tag declarators;
```

Parameters

template-spec

Optional template specifications. For more information, refer to Template Specifications.

struct

The **struct** keyword.

ms-decl-spec

Optional storage-class specification. For more information, refer to the __declspec keyword.

tag

The type name given to the structure. The tag becomes a reserved word within the scope of the structure. The tag is optional. If omitted, an anonymous structure is defined. For more information, see Anonymous Class Types.

base-list

Optional list of classes or structures this structure will derive its members from. See Base Classes for more information. Each base class or structure name can be preceded by an access specifier (public, private, protected) and the virtual keyword. See the member-access table in Controlling Access to Class Members for more information.

member-list

List of structure members. Refer to Class Members for more information. The only difference here is that **struct** is used in place of **class**.

declarators

Declarator list specifying the names of the class. Declarator lists declare one or more instances of the structure type. Declarators may include initializer lists if all data members of the class are **public**. Initializer lists are common in structures because data members are **public** by default. See Overview of Declarators for more information.

Remarks

A structure type is a user-defined composite type. It is composed of fields or members that can have different types.

In C++, a structure is the same as a class except that its members are **public** by default.

For information on managed classes and structs, see Classes and Structs.

Using a Structure

In C, you must explicitly use the **struct** keyword to declare a structure. In C++, this is unnecessary once the type has been defined.

You have the option of declaring variables when the structure type is defined by placing one or more comma-separated variable names between the closing brace and the semicolon.

Structure variables can be initialized. The initialization for each variable must be enclosed in braces.

For related information, see class, union, and enum.

Example

```
// struct1.cpp
struct PERSON {   // Declare PERSON struct type
```

```
   int age;   // Declare member types
   long ss;
   float weight;
   char name[25];
} family_member;   // Define object of type PERSON

int main() {
   struct PERSON sister;   // C style structure declaration
   PERSON brother;   // C++ style structure declaration
   sister.age = 13;   // assign values to members
   brother.age = 7;
}
```

```
// struct2.cpp
// compile with: /c
struct POINT {   // Declare POINT structure
   int x;   // Define members x and y
   int y;
} spot = { 20, 40 };    // Variable spot has
                        // values x = 20, y = 40

struct POINT there;     // Variable there has POINT type

struct CELL {   // Declare CELL bit field
   unsigned short character  : 8;  // 00000000 ????????
   unsigned short foreground : 3;  // 00000??? 00000000
   unsigned short intensity  : 1;  // 0000?000 00000000
   unsigned short background : 3;  // 0???0000 00000000
   unsigned short blink      : 1;  // ?0000000 00000000
} screen[25][80];       // Array of bit fields
```

See Also

Reference

Defining Class Types

Anonymous Structures

Microsoft Specific

A Microsoft C extension allows you to declare a structure variable within another structure without giving it a name. These nested structures are called anonymous structures. C++ does not allow anonymous structures.

You can access the members of an anonymous structure as if they were members in the containing structure.

Example

```
// anonymous_structures.c
#include <stdio.h>

struct phone
{
    int  areacode;
    long number;
};

struct person
{
    char   name[30];
    char   gender;
    int    age;
    int    weight;
    struct phone;    // Anonymous structure; no name needed
} Jim;

int main()
{
    Jim.number = 1234567;
    printf_s("%d\n", Jim.number);
}
```

Output

```
1234567
```

END Microsoft Specific

See Also

Reference

struct (C++)

Unsized Arrays in Structures

Microsoft Specific

A Microsoft extension allows the last member of a C or C++ structure or class to be a variable-sized array. These are called unsized arrays. The unsized array at the end of the structure allows you to append a variable-sized string or other array, thus avoiding the run-time execution cost of a pointer dereference.

```
// unsized_arrays_in_structures1.cpp
// compile with: /c
struct PERSON {
   unsigned number;
   char name[];   // Unsized array
};
```

If you apply the **sizeof** operator to this structure, the ending array size is considered to be 0. The size of this structure is 2 bytes, which is the size of the unsigned member. To get the true size of a variable of type `PERSON`, you would need to obtain the array size separately.

The size of the structure is added to the size of the array to get the total size to be allocated. After allocation, the array is copied to the array member of the structure, as shown below:

Example

```
// unsized_arrays_in_structures2.cpp
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>

enum {
   sizeOfBuffer = 40
};

struct PERSON {
   unsigned number;
   char name[];   // Unsized array
};

int main() {
   char szWho[sizeOfBuffer];
   PERSON *ptrPerson = NULL;

   printf_s( "Enter name: " );
   gets_s( szWho, sizeOfBuffer );

   // Allocate space for structure, name, and terminating null
   ptrPerson = (PERSON *)malloc( sizeof( struct PERSON )
                           + strlen( szWho ) + 1 );

   // Copy the string to the name member
   strcpy_s( ptrPerson->name, sizeOfBuffer, szWho );
}
```

Input

```
John
```

Comments

Even after the structure variable's array is initialized, the **sizeof** operator returns the size of the variable without the array.

Structures with unsized arrays can be initialized, but arrays of such structures cannot be initialized.

```
struct PERSON me  = { 6, "Me" };        // Legal
```

```
struct PERSON you = { 7, "You" };

struct PERSON us[2] = { { 8, "Them" },  // Error
                        { 9, "We" } };
```

An array of characters initialized with a string literal gets space for the terminating null; an array initialized with individual characters (for example, {'a', 'b', 'c'}) does not.

A structure with an unsized array can appear in other structures, as long as it is the last member declared in its enclosing structure. Classes or structures with unsized arrays cannot have direct or indirect virtual bases.

For related information, see volatile and #define.

END Microsoft Specific
See Also
Reference
struct (C++)

Anonymous Class Types

Classes can be anonymous — that is, they can be declared without an *identifier*. This is useful when you replace a class name with a **typedef** name, as in the following:

```
typedef struct
{
    unsigned x;
    unsigned y;
} POINT;
```

Note

The use of anonymous classes shown in the previous example is useful for preserving compatibility with existing C code. In some C code, the use of **typedef** in conjunction with anonymous structures is prevalent.

Anonymous classes are also useful when you want a reference to a class member to appear as though it were not contained in a separate class, as in the following:

```
struct PTValue
{
    POINT ptLoc;
    union
    {
        int  iValue;
        long lValue;
    };
};

PTValue ptv;
```

In the preceding code, `iValue` can be accessed using the object member-selection operator (**.**) as follows:

```
int i = ptv.iValue;
```

Anonymous classes are subject to certain restrictions. (For more information about anonymous unions, see Unions.) Anonymous classes:

- Cannot have a constructor or destructor.
- Cannot be passed as arguments to functions (unless type checking is defeated using ellipses).
- Cannot be returned as return values from functions.

See Also

Reference

Defining Class Types

Point of Class Definition

A class is defined at the end of its class specifier. Member functions need not be defined in order for the class to be considered defined. Consider the following:

```
// point_of_class_definition.cpp
class Point                     // Point class
{                               //  considered defined.
public:
    Point()
        { cx = cy = 0; }        // Constructor defined.
    Point( int x, int y )
        { cx = x, cy = y; }     // Constructor defined.
    unsigned &x( unsigned );    // Accessor declared.
    unsigned &y( unsigned );    // Accessor declared.
private:
    unsigned  cx, cy;
};
int main()
{
}
```

Even though the two accessor functions (`x` and `y`) are not defined, the class `Point` is considered defined. (Accessor functions are functions provided to help give safe access to member data.)

See Also

Reference

Defining Class Types

Class-Type Objects

An object is a typed region of storage in the execution environment; in addition to retaining state information, it also defines behavior. Class-type objects are defined using *class-name*. Consider the following code fragment:

```
// class_type_objects.cpp
class Account
{
public:
   Account()   // Default constructor
   {
   }
   Account( double );   // Construct from double.
   double& Deposit( double );
   double& Withdraw( double, int );
};

int main()
{
   Account CheckingAccount;   // Define object of class type.
}
```

The preceding code declares a class (a new type) called `Account`. It then uses this new type to define an object called `CheckingAccount`.

The following operations are defined by C++ for objects of class type:

- Assignment. One object can be assigned to another. The default behavior for this operation is a memberwise copy. This behavior can be modified by supplying a user-defined assignment operator.
- Initialization using copy constructors.

The following are examples of initialization using user-defined copy constructors:

- Explicit initialization of an object. For example:

  ```
  Point myPoint = thatPoint;
  ```

  declares `myPoint` as an object of type `Point` and initializes it to the value of `thatPoint`.

- Initialization caused by passing as an argument. Objects can be passed to functions by value or by reference. If they are passed by value, a copy of each object is passed to the function. The default method for creating the copy is memberwise copy; this can be modified by supplying a user-defined copy constructor (a constructor that takes a single argument of the type "reference to class").
- Initialization caused by the initialization of return values from functions. Objects can be returned from functions by value or by reference. The default method for returning an object by value is a memberwise copy; this can be modified by supplying a user-defined copy constructor. An object returned by reference (using pointer or reference types) should not be both automatic and local to the called function. If it is, the object referred to by the return value will have gone out of scope before it can be used.

Overloaded Operators explains how to redefine other operators on a class-by-class basis.

See Also
Reference
Overview of Classes

Empty Classes

You can declare empty classes, but objects of such types still have nonzero size.

Example

```
// empty_classes.cpp
// compile with: /EHsc
#include <iostream>

class NoMembers
{
};

using namespace std;
int main()
{
    NoMembers n;  // Object of type NoMembers.
    cout << "The size of an object of empty class is: "
         << sizeof n << endl;
}
```

Output

```
The size of an object of empty class is: 1
```

The memory allocated for such objects is of nonzero size; therefore, the objects have different addresses. Having different addresses makes it possible to compare pointers to objects for identity. Also, in arrays, each member array must have a distinct address.

Microsoft Specific

An empty base class typically contributes zero bytes to the size of a derived class.

END Microsoft Specific

See Also

Reference

Class-Type Objects

Class Names

Class declarations introduce new types, called class names or class types, into programs. Except for forward declarations, these class declarations also act as definitions of the class for a given translation unit. There may be only one definition for a given class type per translation unit. Using these new class types, you can declare objects, and the compiler can perform type checking to verify that no operations incompatible with the types are performed on the objects.

Remarks

An example of such type checking is:

```
// class_names.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
class Point {
public:
   unsigned x, y;
};

class Rect {
public:
   unsigned x1, y1, x2, y2;
};

// Prototype a function that takes two arguments, one of type
//  Point and the other of type pointer to Rect.
int PtInRect( Point, Rect & );

int main() {
   Point pt;
   Rect  rect;

   rect = pt;   // C2679 Types are incompatible.
   pt = rect;   // C2679 Types are incompatible.

  // Error. Arguments to PtInRect are reversed.
  // cout << "Point is " << PtInRect( rect, pt ) ? "" : "not"
  //   << " in rectangle" << endl;
}
```

As the preceding code illustrates, operations (such as assignment and argument passing) on class-type objects are subject to the same type checking as objects of built-in types.

Because the compiler distinguishes between class types, functions can be overloaded on the basis of class-type arguments as well as built-in type arguments. For more information about overloaded functions, see Function Overloading and Overloading.

See Also

Reference

Classes, Structures, and Unions

Declaring and Accessing Class Names

Class names can be declared in global or class scope. If they are declared in class scope, they are referred to as "nested" classes.

Microsoft Specific

Function definitions are not permitted in local class declarations in Microsoft C++.

END Microsoft Specific

Any class name introduced in class scope hides other elements of the same name in an enclosing scope. Names hidden by such a declaration can then be referred to only by using an *elaborated-type-specifier*. The following example shows an example of using an *elaborated-type-specifier* to refer to a hidden name:

```
// class_names2.cpp
struct A   // Global scope definition of A.
{
    int a;
};

int main()
{
    char A = 'a';   // Redefine the name A as an object.
    struct A AObject;
}
```

Because the name `A` that refers to the structure is hidden by the `A` that refers to the `char` object, **struct** (a *class-key*) must be used to declare `AObject` as type `A`.

You can use the *class-key* to declare a class without providing a definition. This nondefining declaration of a class introduces a class name for forward reference. This technique is useful when designing classes that refer to one another in **friend** declarations. It is also useful when class names must be present in header files but the definition is not required. For example:

```
// RECT.H
class Point;   // Nondefining declaration of class Point.
class Line
{
public:
    int Draw( Point &ptFrom, Point &ptTo );
    ...
};
```

In the preceding sample, the name `Point` must be present, but it need not be a defining declaration that introduces the name.

See Also

Reference

Class Names

typedef Statements and Classes

Using the **typedef** statement to name a class type causes the **typedef** name to become a *class-name*. For more information, see typedef Specifier.

See Also
Reference
Class Names

Class Members

Classes can have these kinds of members:

- Member functions.
- Data members.
- Classes, which include classes, structures, and unions. (See Nested Class Declarations and Unions.)
- Enumerations.
- Bit fields.
- Friends.
- Type names.

Note
Friends are included in the preceding list because they are contained in the class declaration. However, they are not tru e class members, because they are not in the scope of the class.

Remarks

The member list of a class may be divided into **private**, **protected** and **public** sections using keywords known as access specifiers. A colon **:** must follow the access specifier. These sections need not be contiguous, that is, any of these keywords may appear several times in the member list. The keyword designates the access of all members up until the next access specifier or the closing brace.

Member declarations in the member list are separated by semicolons **;**. See Class-Member Declaration Syntax.

The purpose of the member list is to:

- Declare the complete set of members for a given class.
- Specify the access (public, private, or protected) associated with various class members.

In the declaration of a member list, you can declare members only once; redeclaration of members produces an error message. Because a member list is a complete set of the members, you cannot add members to a given class with subsequent class declarations.

Member declarators cannot contain initializers. Supplying an initializer produces an error message as illustrated in the following code:

```
// class_members1.cpp
// C2864 expected
class CantInit
{
public:
    long l = 7;       // Error: attempt to initialize
                      //  class member.
    static int i = 9; // Error: must be defined and initialized
                      // outside of class declaration.
};
int main()
{
}
```

Because a separate instance of nonstatic member data is created for each object of a given class type, the correct way to initialize member data is to use the class's constructor. (Constructors are covered in Constructors,.)

There is only one shared copy of static data members for all objects of a given class type. Static data members must be defined and can be initialized at file scope. (For more information about static data members, see Static Data Members.) The following example shows how to perform these initializations:

```
// class_members2.cpp
class CanInit
{
public:
    CanInit() { l = 7; } // Initializes l when new objects of type
                         //  CanInit are created.
    long       l;
    static int i;
    static int j;
};

// i is defined at file scope and initialized to 15.
// The initializer is evaluated in the scope of CanInit.
int CanInit::i = 15;

// The right side of the initializer is in the scope
// of the object being initialized
int CanInit::j = i;

int main()
{
}
```

Note

The class name, `CanInit`, must precede `i` to specify that the `i` being defined is a member of class `CanInit`.

Microsoft Specific

Microsoft C++ allows static, const integral, and const enum data members to be initialized in the class definition. See Microsoft Extensions to C and C++.

END Microsoft Specific

See Also

Reference

Classes, Structures, and Unions

Class-Member Declaration Syntax

Class data members and function members are generally declared following the rules for declarations. They consist of, in this order:

- Declaration specifiers. In classes, the storage class specifiers **static** and **mutable** are allowed for data members. The cv-qualifiers **const** and **volatile** are allowed for data and function members. The declaration specifiers **friend**, **static**, **virtual** and **inline** are allowed for functions, although **friend** indicates a non-member function.
- Declarators without initializers. In the case of virtual functions, the pure specifier of the form **= 0** is used after the function declarator to specify a pure virtual function (a virtual function without an implementation in that class).
- Optionally, bit fields. See Bit Fields.

OR

- Function definitions. The definition of a function can be included in the class. If a member function is so defined, it is automatically considered inline.

OR

- Member templates (see the comments on C++ member templates in Members of Class Templates).

Member data cannot be declared as **auto**, **extern**, or **register** storage class. They can, however, be declared as having **static** storage class.

When you declare a **friend** class in a member list, you can omit the member declarator list. For more information on friends, see friend Specifier and Friends. Even if a class name has not been introduced, it can be used in a **friend** declaration. This **friend** declaration introduces the name. However, in member declarations for such classes, the elaborated type specifier syntax must be used, as shown in the following example:

```
// class_member_declaration_syntax.cpp
// compile with: /c
struct HasFriends {
   friend class NotDeclaredYet;
};
```

In the preceding example, there is no member declarator list after the class declaration. Because the declaration for `NotDeclaredYet` has not yet been processed, the elaborated type specifier form is used: `class NotDeclaredYet`. A type that has been declared can be specified in a **friend** member declaration using a normal type specifier:

```
// class_member_declaration_syntax2.cpp
// compile with: /c
struct AlreadyDeclared {};

struct HasFriends {
   friend AlreadyDeclared;
};
```

The pure specifier (shown in the following example) indicates that no implementation is supplied for the virtual function being declared. Therefore, the pure specifier can be specified only on virtual functions. Consider this example:

```
// class_member_declaration_syntax3.cpp
// compile with: /c
class StrBase {   // Base class for strings.
public:
   virtual int IsLessThan( StrBase& ) = 0;
   virtual int IsEqualTo( StrBase& ) = 0;
   virtual StrBase& CopyOf( StrBase& ) = 0;
};
```

The preceding code declares an abstract base class — that is, a class designed to be used only as the base class for more

specific classes. Such base classes can enforce a particular protocol, or set of functionality, by declaring one or more virtual functions as "pure" virtual functions, using the pure specifier.

Classes that inherit from the `StrBase` class must provide implementations for the pure virtual functions; otherwise, they, too, are considered abstract base classes.

Abstract base classes cannot be used to declare objects. For example, before an object of a type inherited from `StrBase` can be declared, the functions `IsLessThan`, `IsEqualTo`, and `CopyOf` must be implemented. (For more information about abstract base classes, see Abstract Classes.)

See Also

Reference

Class Members

Declaring Unsized Arrays in Member Lists

Microsoft Specific

Unsized arrays can be declared as the last data member in class member lists if the program is not compiled with the ANSI-compatibility option (/Za). Because this is a Microsoft extension, using unsized arrays in this way can make your code less portable. To declare an unsized array, omit the first dimension. For example:

```
// declaring_unsized_arrays.cpp
class Symbol
{
public:
   int  SymbolType;
   char SymbolText[];
};

int main()
{
}
```

END Microsoft Specific

See Also

Reference

Class Members

Restrictions

If a class contains an unsized array, it cannot be used as the base class for another class. In addition, a class containing an unsized array cannot be used to declare any member except the last member of another class. A class containing an unsized array cannot have a direct or indirect virtual base class.

The **sizeof** operator, when applied to a class containing an unsized array, returns the amount of storage required for all members except the unsized array. Implementors of classes that contain unsized arrays should provide alternate methods for obtaining the correct size of the class.

You cannot declare arrays of objects that have unsized array components. Also, performing pointer arithmetic on pointers to such objects generates an error message.

See Also

Reference

Declaring Unsized Arrays in Member Lists

Storage of Class-Member Data

Nonstatic class-member data is stored in such a way that items falling between access specifiers are stored at successively higher memory addresses. No ordering across access specifiers is guaranteed.

Microsoft Specific

Depending on the /Zp compiler option or the pack pragma, intervening space can be introduced to align member data on word or doubleword boundaries.

In Microsoft C++, class-member data is stored at successively higher memory addresses, even though the C++ language does not require it. Basing assumptions on this ordering can lead to nonportable code.

END Microsoft Specific

See Also
Reference
Class Members

Member Naming Restrictions

A function with the same name as the class in which it is declared is a constructor. A constructor is implicitly called when an object of this class type is created. (For more information about constructors, see Constructors.)

The following items cannot have the same name as the classes in whose scope they are declared: data members (both static and nonstatic), enclosed enumerators, members of anonymous unions, and nested class types.

See Also

Reference

Class Members

Member Functions (C++)

Classes can contain data and functions. These functions are referred to as "member functions." Any nonstatic function declared inside a class declaration is considered a member function and is called using the member-selection operators (**.** and **->**). When calling member functions from other member functions of the same class, the object and member-selection operator can be omitted. For example:

```
// member_functions1.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Point
{
public:
   short x()
   {
      return _x;
   }

   short y()
   {
      return _y;
   }

   void  Show()
   {
      cout << x() << ", " << y() << "\n";
   }
private:
   short _x, _y;
};

int main()
{
   Point pt;
   pt.Show();
}
```

Note that in the member function, `Show`, calls to the other member functions, `x` and `y`, are made without member-selection operators. These calls implicitly mean `this->x()` and `this->y()`. However, in **main**, the member function, `Show`, must be selected using the object `pt` and the member-selection operator (**.**).

Static functions declared inside a class can be called using the member-selection operators or by specifying the fully qualified function name (including the class name).

Note

A function declared using the **friend** keyword is not considered a member of the class in which it is declared a **friend** (although it can be a member of another class). A **friend** declaration controls the access a nonmember function has to class data.

The following class declaration shows how member functions are declared and called:

```
// member_functions2.cpp
class Point
{
public:
   unsigned GetX()
   {
      return ptX;
   }
   unsigned GetY()
   {
```

```
      return ptY;
   }

   void SetX( unsigned x )
   {
      ptX = x;
   }
   void SetY( unsigned y )
   {
      ptY = y;
   }

private:
    unsigned ptX, ptY;
};

int main()
{
   // Declare a new object of type Point.
   Point ptOrigin;

   // Member function calls use the . member-selection operator.
   ptOrigin.SetX( 0 );
   ptOrigin.SetY( 0 );

   // Declare a pointer to an object of type Point.
   Point *pptCurrent = new Point;
   // Member function calls use the -> member-selection operator.
   pptCurrent->SetX( ptOrigin.GetX() + 10 );
   pptCurrent->SetY( ptOrigin.GetY() + 10 );
}
```

In the preceding code, the member functions of the object `ptOrigin` are called using the member-selection operator (**.**). However, the member functions of the object pointed to by `pptCurrent` are called using the **–>** member-selection operator.

See Also

Reference

Classes, Structures, and Unions

Overview of Member Functions

Member functions are either static or nonstatic. The behavior of static member functions differs from other member functions because static member functions have no implicit **this** argument. Nonstatic member functions have a **this** pointer. Member functions, whether static or nonstatic, can be defined either in or outside the class declaration.

If a member function is defined inside a class declaration, it is treated as an inline function, and there is no need to qualify the function name with its class name. Although functions defined inside class declarations are already treated as inline functions, you can use the **inline** keyword to document code.

An example of declaring a function within a class declaration follows:

```
// overview_of_member_functions1.cpp
class Account
{
public:
    // Declare the member function Deposit within the declaration
    //  of class Account.
    double Deposit( double HowMuch )
    {
        balance += HowMuch;
        return balance;
    }
private:
    double balance;
};

int main()
{
}
```

If a member function's definition is outside the class declaration, it is treated as an inline function only if it is explicitly declared as **inline**. In addition, the function name in the definition must be qualified with its class name using the scope-resolution operator (**::**).

The following example is identical to the previous declaration of class `Account`, except that the `Deposit` function is defined outside the class declaration:

```
// overview_of_member_functions2.cpp
class Account
{
public:
    // Declare the member function Deposit but do not define it.
    double Deposit( double HowMuch );
private:
    double balance;
};

inline double Account::Deposit( double HowMuch )
{
    balance += HowMuch;
    return balance;
}

int main()
{
}
```

Note

Although member functions can be defined either inside a class declaration or separately, no member functions can be adde d to a class after the class is defined.

Classes containing member functions can have many declarations, but the member functions themselves must have only one definition in a program. Multiple definitions cause an error message at link time. If a class contains inline function definitions, the function definitions must be identical to observe this "one definition" rule.

See Also

Reference

Member Functions (C++)

Nonstatic Member Functions

Nonstatic member functions have an implied argument, **this**, that points to the object through which the function is invoked. The type of **this** is *type* *** const**. These functions are considered to have class scope and can use class data and other member functions in the same class scope directly. In the preceding example, the expression `balance += HowMuch` adds the value of `HowMuch` to the class member `balance`. Consider the following statements:

```
Account Checking;
Checking.Deposit( 57.00 );
```

The preceding code declares an object of type `Account` and then invokes the member function `Deposit` to add $57.00 to it. In the function `Account::Deposit`, balance is taken to mean `Checking.balance` (the balance member for this object).

Nonstatic member functions are intended to operate on objects of their class type. Calling such a function on objects of different types (using explicit type conversions) causes undefined behavior.

See Also

Reference

Overview of Member Functions

this Pointer

The **this** pointer is a pointer accessible only within the nonstatic member functions of a **class**, **struct**, or **union** type. It points to the object for which the member function is called. Static member functions do not have a **this** pointer.

```
this
this->member-identifier
```

Remarks

An object's **this** pointer is not part of the object itself; it is not reflected in the result of a **sizeof** statement on the object. Instead, when a nonstatic member function is called for an object, the address of the object is passed by the compiler as a hidden argument to the function. For example, the following function call:

```
myDate.setMonth( 3 );
```

can be interpreted this way:

```
setMonth( &myDate, 3 );
```

The object's address is available from within the member function as the **this** pointer. Most uses of **this** are implicit. It is legal, though unnecessary, to explicitly use **this** when referring to members of the class. For example:

```
void Date::setMonth( int mn )
{
   month = mn;            // These three statements
   this->month = mn;      // are equivalent
   (*this).month = mn;
}
```

The expression `*this` is commonly used to return the current object from a member function:

```
return *this;
```

The **this** pointer is also used to guard against self-reference:

```
if (&Object != this) {
// do not execute in cases of self-reference
```

Note

Because the **this** pointer is nonmodifiable, assignments to **this** are not allowed. Earlier implementations of C++ allowed assi gnments to **this**.

Occasionally, the **this** pointer is used directly — for example, to manipulate self-referential data structures, where the address of the current object is required.

Example

```
// this_pointer.cpp
// compile with: /EHsc

#include <iostream>
#include <string.h>

using namespace std;
```

```
class Buf
{
public:
    Buf( char* szBuffer, size_t sizeOfBuffer );
    Buf& operator=( const Buf & );
    void Display() { cout << buffer << endl; }

private:
    char*   buffer;
    size_t  sizeOfBuffer;
};

Buf::Buf( char* szBuffer, size_t sizeOfBuffer )
{
    sizeOfBuffer++; // account for a NULL terminator

    buffer = new char[ sizeOfBuffer ];
    if (buffer)
    {
        strcpy_s( buffer, sizeOfBuffer, szBuffer );
        sizeOfBuffer = sizeOfBuffer;
    }
}

Buf& Buf::operator=( const Buf &otherbuf )
{
    if( &otherbuf != this )
    {
        if (buffer)
            delete [] buffer;

        sizeOfBuffer =  strlen( otherbuf.buffer ) + 1;
        buffer = new char[sizeOfBuffer];
        strcpy_s( buffer, sizeOfBuffer, otherbuf.buffer );
    }
    return *this;
}

int main()
{
    Buf myBuf( "my buffer", 10 );
    Buf yourBuf( "your buffer", 12 );

    // Display 'my buffer'
    myBuf.Display();

    // assignment opperator
    myBuf = yourBuf;

    // Display 'your buffer'
    myBuf.Display();
}
```

Output

```
my buffer
your buffer
```

See Also

Reference

C++ Keywords

Type of this Pointer

Argument Matching and the this Pointer

Type of this Pointer

The **this** pointer's type can be modified in the function declaration by the **const** and **volatile** keywords. To declare a function as having the attributes of one or more of these keywords, add the keyword(s) after the function argument list.

Consider this example:

```
// type_of_this_pointer1.cpp
class Point
{
    unsigned X() const;
};
int main()
{
}
```

The preceding code declares a member function, `X`, in which the **this** pointer is treated as a **const** pointer to a **const** object. Combinations of *cv-mod-list* options can be used, but they always modify the object pointed to by **this**, not the **this** pointer itself. Therefore, the following declaration declares function `X`; the **this** pointer is a **const** pointer to a **const** object:

```
// type_of_this_pointer2.cpp
class Point
{
    unsigned X() const;
};
int main()
{
}
```

The type of **this** in a member function is described by the following syntax, where *cv-qualifier-list* is determined from the member functions declarator and can be **const** or **volatile** (or both), and *class-type* is the name of the class:

[cv-qualifier-list] class-type *** const this**

In other words, **this** is always a const pointer; it cannot be reassigned. The **const** or **volatile** qualifiers used in the member function declaration apply to the class instance pointed to by **this** in the scope of that function.

The following table explains more about how these modifiers work.

Semantics of this Modifiers

Modifier	Meaning
const	Cannot change member data; cannot invoke member functions that are not **const**.
volatile	Member data is loaded from memory each time it is accessed; disables certain optimizations.

It is an error to pass a **const** object to a member function that is not **const**. Similarly, it is an error to pass a **volatile** object to a member function that is not **volatile**.

Member functions declared as **const** cannot change member data — in such functions, the **this** pointer is a pointer to a **const** object.

Note
Constructors and destructors cannot be declared as **const** or **volatile**. They can, however, be invoked on **const** or **volatile** o bjects.

See Also

Reference

this Pointer

Static Member Functions

Static member functions are considered to have class scope. In contrast to nonstatic member functions, these functions have no implicit **this** argument; therefore, they can use only static data members, enumerators, or nested types directly. Static member functions can be accessed without using an object of the corresponding class type. Consider this example:

Example

```
// static_member_functions.cpp
#include <stdio.h>

class StaticTest
{
private:
    static int x;
public:
    static int count()
    {
        return x;
    }
};

int StaticTest::x = 9;

int main()
{
    printf_s("%d\n", StaticTest::count());
}
```

Output

```
9
```

Comments

In the preceding code, the class `StaticTest` contains the static member function `count`. This function returns the value of the private class member but is not necessarily associated with a given object of type `StaticTest`.

Static member functions have external linkage. These functions do not have this pointers. As a result, the following restrictions apply to such functions:

- They cannot access nonstatic class member data using the member-selection operators (**.** or **->**).
- They cannot be declared as **virtual**.
- They cannot have the same name as a nonstatic function that has the same argument types.

> **Note**
>
> The left side of a member-selection operator (**.** or **->**) that selects a static member function is not evaluated. This can be important if the function is used for its side effects. For example, the expression `SideEffects().CountOf()` does not cal l the function `SideEffects`.

See Also

Reference

this Pointer

Static Data Members (C++)

Classes can contain static member data and member functions. When a data member is declared as **static**, only one copy of the data is maintained for all objects of the class. (For more information, see Static Member Functions.)

Static data members are not part of objects of a given class type; they are separate objects. As a result, the declaration of a static data member is not considered a definition. The data member is declared in class scope, but definition is performed at file scope. These static members have external linkage. The following example illustrates this:

```
// static_data_members.cpp
class BufferedOutput
{
public:
   // Return number of bytes written by any object of this class.
   short BytesWritten()
   {
      return bytecount;
   }

   // Reset the counter.
   static void ResetCount()
   {
      bytecount = 0;
   }

   // Static member declaration.
   static long bytecount;
};

// Define bytecount in file scope.
long BufferedOutput::bytecount;

int main()
{
}
```

In the preceding code, the member `bytecount` is declared in class `BufferedOutput`, but it must be defined outside the class declaration.

Static data members can be referred to without referring to an object of class type. The number of bytes written using `BufferedOutput` objects can be obtained as follows:

```
long nBytes = BufferedOutput::bytecount;
```

For the static member to exist, it is not necessary that any objects of the class type exist. Static members can also be accessed using the member-selection (**.** and **->**) operators. For example:

```
BufferedOutput Console;

long nBytes = Console.bytecount;
```

In the preceding case, the reference to the object (`Console`) is not evaluated; the value returned is that of the static object `bytecount`.

Static data members are subject to class-member access rules, so private access to static data members is allowed only for class-member functions and friends. These rules are described in Member-Access Control. The exception is that static data members must be defined in file scope regardless of their access restrictions. If the data member is to be explicitly initialized, an initializer must be provided with the definition.

The type of a static member is not qualified by its class name. Therefore, the type of `BufferedOutput::bytecount` is `long`.

See Also

Reference

mutable

This keyword can only be applied to non-static and non-const data members of a class. If a data member is declared **mutable**, then it is legal to assign a value to this data member from a **const** member function.

```
mutable member-variable-declaration;
```

Remarks

For example, the following code will compile without error because `m_accessCount` has been declared to be `mutable`, and therefore can be modified by `GetFlag` even though `GetFlag` is a const member function.

```
// mutable.cpp
class X
{
public:
   bool GetFlag() const
   {
      m_accessCount++;
      return m_flag;
   }
private:
   bool m_flag;
   mutable int m_accessCount;
};

int main()
{
}
```

See Also

Reference

C++ Keywords

Unions

A **union** is a user-defined data or class type that, at any given time, contains only one object from its list of members (although that object can be an array or a class type).

```
union [tag] { member-list } [declarators];
[union] tag declarators;
```

Parameters

tag

The type name given to the union.

member-list

List of the types of data the union can contain. See Remarks.

declarators

Declarator list specifying the names of the union. See Overview of Declarators for more information.

Remarks

The *member-list* of a union represents the kinds of data the union can contain. A union requires enough storage to hold the largest member in its *member-list*. For more information, refer to Union Declarations (C Language Reference).

Declaring a Union

Begin the declaration of a union with the union keyword, and enclose the member list in curly braces:

```
// declaring_a_union.cpp
union DATATYPE    // Declare union type
{
    char   ch;
    int    i;
    long   l;
    float  f;
    double d;
} var1;          // Optional declaration of union variable

int main()
{
}
```

Using a Union

A C++ union is a limited form of the class type. It can contain access specifiers (public, protected, private), member data, and member functions, including constructors and destructors. It cannot contain virtual functions or static data members. It cannot be used as a base class, nor can it have base classes. Default access of members in a union is public.

A C union type can contain only data members.

In C, you must use the **union** keyword to declare a union variable. In C++, the **union** keyword is unnecessary:

```
union DATATYPE var2;   // C declaration of a union variable
DATATYPE var3;         // C++ declaration of a union variable
```

A variable of a union type can hold one value of any type declared in the union. Use the member-selection operator (**.**) to access a member of a union:

```
var1.i = 6;           // Use variable as integer
var2.d = 5.327;       // Use variable as double
```

You can declare and initialize a union in the same statement by assigning an expression enclosed in braces. The expression is evaluated and assigned to the first field of the union.

Example

```
// using_a_union.cpp
#include <stdio.h>

union NumericType
{
    int         iValue;
    long        lValue;
    double      dValue;
};

int main()
{
    union NumericType Values = { 10 };   // iValue = 10
    printf_s("%d\n", Values.iValue);
    Values.dValue = 3.1416;
    printf_s("%f\n", Values.dValue);
}
```

Output

```
10
3.141600
```

The `NumericType` union is arranged in memory (conceptually) as shown in the following figure.

Storage of Data in NumericType Union

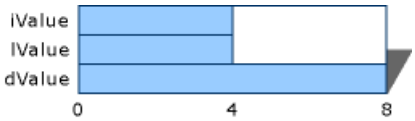

See Also

Reference

Classes, Structures, and Unions
C++ Keywords
Anonymous Unions
class (C++)
struct (C++)

Member Functions in Unions

In addition to member data, unions can have member functions, as described in Member Functions. Although unions can have special functions such as constructors and destructors, unions cannot contain virtual functions. (For more information, see Constructors and Destructors.)

See Also

Reference

Unions

Unions as Class Types

Unions cannot have base classes; that is, they cannot inherit the attributes of other unions, structures, or classes. Unions also cannot be used as base classes for further inheritance.

Inheritance is covered in detail in Derived Classes.

See Also

Reference

Unions

Union Member Data

Unions can contain most types in their member lists, except for the following:

- Class types that have constructors or destructors
- Class types that have user-defined assignment operators
- Static data members

See Also

Reference

Unions

Anonymous Unions

Anonymous unions are unions that are declared without a *class-name* or *declarator-list*.

```
union  {  member-list  }
```

Remarks

Such union declarations do not declare types — they declare objects. The names declared in an anonymous union cannot conflict with other names declared in the same scope.

In C, an anonymous union can have a tag; it cannot have declarators.

Names declared in an anonymous union are used directly, like nonmember variables.

In addition to the restrictions listed in Union Member Data, anonymous unions are subject to additional restrictions:

- They must also be declared as **static** if declared in file scope. If declared in local scope, they must be static or automatic.
- They can have only public members; private and protected members in anonymous unions generate errors.
- They cannot have function members.

Note

Simply omitting the *class-name* portion of the syntax does not make a union an anonymous union. For a union to qualify as an anonymous union, the declaration must not declare an object.

Example

```
// anonymous_unions.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
int main() {
   union {
      int d;
      char *f;
   };

   d = 4;
   cout << d << endl;

   f = "inside of union";
   cout << f << endl;
}
```

Output

```
4
inside of union
```

See Also

Reference

Unions

C++ Bit Fields

Classes and structures can contain members that occupy less storage than an integral type. These members are specified as bit fields. The syntax for bit-field *member-declarator* specification follows:

```
declarator  : constant-expression
```

Remarks

The (optional) *declarator* is the name by which the member is accessed in the program. It must be an integral type (including enumerated types). The *constant-expression* specifies the number of bits the member occupies in the structure. Anonymous bit fields — that is, bit-field members with no identifier — can be used for padding.

Note
An unnamed bit field of width 0 forces alignment of the next bit field to the next *type* boundary, where *type* is the type of the member.

The following example declares a structure that contains bit fields:

```
// bit_fields1.cpp
// compile with: /LD
struct Date {
   unsigned short nWeekDay  : 3;    // 0..7   (3 bits)
   unsigned short nMonthDay : 6;    // 0..31  (6 bits)
   unsigned short nMonth    : 5;    // 0..12  (5 bits)
   unsigned short nYear     : 8;    // 0..100 (8 bits)
};
```

The conceptual memory layout of an object of type `Date` is shown in the following figure.

Memory Layout of Date Object

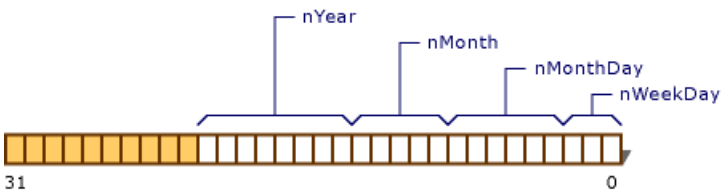

Note that `nYear` is 8 bits long and would overflow the word boundary of the declared type, **unsigned short**. Therefore, it is begun at the beginning of a new **unsigned short**. It is not necessary that all bit fields fit in one object of the underlying type; new units of storage are allocated, according to the number of bits requested in the declaration.

Microsoft Specific

The ordering of data declared as bit fields is from low to high bit, as shown in the figure above.

END Microsoft Specific

If the declaration of a structure includes an unnamed field of length 0, as shown in the following example,

```
// bit_fields2.cpp
// compile with: /LD
struct Date {
   unsigned nWeekDay  : 3;    // 0..7   (3 bits)
   unsigned nMonthDay : 6;    // 0..31  (6 bits)
   unsigned           : 0;    // Force alignment to next boundary.
   unsigned nMonth    : 5;    // 0..12  (5 bits)
   unsigned nYear     : 8;    // 0..100 (8 bits)
};
```

the memory layout is as shown in the following figure.

Layout of Date Object with Zero-Length Bit Field

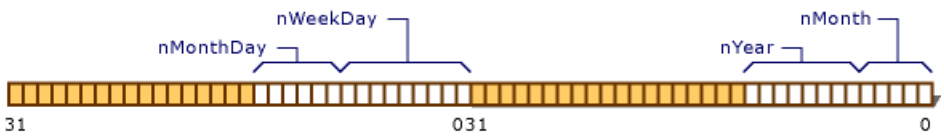

The underlying type of a bit field must be an integral type, as described in Fundamental Types.

See Also

Reference

Classes, Structures, and Unions

Restrictions on Use of Bit Fields

The following list details erroneous operations on bit fields:

- Taking the address of a bit field.
- Initializing a reference with a bit field.

See Also

Reference

C++ Bit Fields

Nested Class Declarations

A class can be declared within the scope of another class. Such a class is called a "nested class." Nested classes are considered to be within the scope of the enclosing class and are available for use within that scope. To refer to a nested class from a scope other than its immediate enclosing scope, you must use a fully qualified name.

The following example shows how to declare nested classes:

```
// nested_class_declarations.cpp
class BufferedIO
{
public:
   enum IOError { None, Access, General };

   // Declare nested class BufferedInput.
   class BufferedInput
   {
   public:
      int read();
      int good()
      {
         return _inputerror == None;
      }
   private:
       IOError _inputerror;
   };

   // Declare nested class BufferedOutput.
   class BufferedOutput
   {
      // Member list
   };
};

int main()
{
}
```

`BufferedIO::BufferedInput` and `BufferedIO::BufferedOutput` are declared within `BufferedIO`. These class names are not visible outside the scope of class `BufferedIO`. However, an object of type `BufferedIO` does not contain any objects of types `BufferedInput` or `BufferedOutput`.

Nested classes can directly use names, type names, names of static members, and enumerators only from the enclosing class. To use names of other class members, you must use pointers, references, or object names.

In the preceding `BufferedIO` example, the enumeration `IOError` can be accessed directly by member functions in the nested classes, `BufferedIO::BufferedInput` or `BufferedIO::BufferedOutput`, as shown in function `good`.

Note

Nested classes declare only types within class scope. They do not cause contained objects of the nested class to be created. T he preceding example declares two nested classes but does not declare any objects of these class types.

An exception to the scope visibility of a nested class declaration is when a type name is declared together with a forward declaration. In this case, the class name declared by the forward declaration is visible outside the enclosing class, with its scope defined to be the smallest enclosing non-class scope. For example:

```
// nested_class_declarations_2.cpp
class C
{
public:
    typedef class U u_t; // class U visible outside class C scope
    typedef class V {} v_t; // class V not visible outside class C
```

```
};

int main()
{
    // okay, forward declaration used above so file scope is used
    U* pu;

    // error, type name only exists in class C scope
    u_t* pu2; // C2065

    // error, class defined above so class C scope
    V* pv; // C2065

    // okay, fully qualified name
    C::V* pv2;
}
```

See Also

Reference

Classes, Structures, and Unions

Access Privileges and Nested Classes

Nesting a class within another class does not give special access privileges to member functions of the nested class. Similarly, member functions of the enclosing class have no special access to members of the nested class.

For more information about access privileges, see Member-Access Control.

See Also

Reference

Nested Class Declarations

Member Functions in Nested Classes

Member functions declared in nested classes can be defined in file scope. The preceding example could have been written:

```
// member_functions_in_nested_classes.cpp
class BufferedIO
{
public:
    enum IOError { None, Access, General };
    class BufferedInput
    {
    public:
        int read(); // Declare but do not define member
        int good(); //  functions read and good.
    private:
        IOError _inputerror;
    };

    class BufferedOutput
    {
        // Member list.
    };
};
// Define member functions read and good in
//  file scope.
int BufferedIO::BufferedInput::read()
{
   return(1);
}

int BufferedIO::BufferedInput::good()
{
    return _inputerror == None;
}
int main()
{
}
```

In the preceding example, the *qualified-type-name* syntax is used to declare the function name. The declaration:

```
BufferedIO::BufferedInput::read()
```

means "the `read` function that is a member of the `BufferedInput` class that is in the scope of the `BufferedIO` class." Because this declaration uses the *qualified-type-name* syntax, constructs of the following form are possible:

```
typedef BufferedIO::BufferedInput BIO_INPUT;

int BIO_INPUT::read()
```

The preceding declaration is equivalent to the previous one, but it uses a **typedef** name in place of the class names.

See Also

Reference

Nested Class Declarations

Friend Functions and Nested Classes

Friend functions declared in a nested class are considered to be in the scope of the nested class, not the enclosing class. Therefore, the friend functions gain no special access privileges to members or member functions of the enclosing class. If you want to use a name that is declared in a nested class in a friend function and the friend function is defined in file scope, use qualified type names as follows:

Example

```
// friend_functions_and_nested_classes.cpp

#include <string.h>

enum
{
    sizeOfMessage = 255
};

char *rgszMessage[sizeOfMessage];

class BufferedIO
{
public:
    class BufferedInput
    {
    public:
        friend int GetExtendedErrorStatus();
        static char *message;
        static int  messageSize;
        int iMsgNo;
   };
};

char *BufferedIO::BufferedInput::message;
int BufferedIO::BufferedInput::messageSize;

int GetExtendedErrorStatus()
{
    int iMsgNo = 1; // assign arbitrary value as message number

    strcpy_s( BufferedIO::BufferedInput::message,
              BufferedIO::BufferedInput::messageSize,
              rgszMessage[iMsgNo] );

    return iMsgNo;
}

int main()
{
}
```

Comment

The following code shows the function `GetExtendedErrorStatus` declared as a friend function. In the function, which is defined in file scope, a message is copied from a static array into a class member. Note that a better implementation of `GetExtendedErrorStatus` is to declare it as:

```
int GetExtendedErrorStatus( char *message )
```

With the preceding interface, several classes can use the services of this function by passing a memory location where they want the error message copied.

See Also

Reference

Type Names in Class Scope

Type names defined within class scope are considered local to their class. They cannot be used outside that class. The following example demonstrates this concept:

```
// type_names_in_class_scope.cpp
// C2146 expected
class Tree {
public:
   typedef Tree * PTREE;
   PTREE  Left;
   PTREE  Right;
   void  *vData;
};

PTREE pTree;  // not in class scope
```

See Also

Reference

Classes, Structures, and Unions

Derived Classes

This section explains how to use derived classes to produce extensible programs.

The following topics are included:

- Overview
- Single inheritance
- Multiple inheritance
- Multiple base classes
- Virtual base classes
- Virtual functions
- Explicit overrides
- Abstract classes
- Summary of scope rules

The __super and __interface keywords are documented in this section.

See Also

Other Resources

C++ Language Reference

Overview of Derived Classes

New classes can be derived from existing classes using a mechanism called "inheritance" (see the information beginning in Single Inheritance). Classes that are used for derivation are called "base classes" of a particular derived class. A derived class is declared using the following syntax:

```
 class Derived : [virtual] [access-specifier] Base
{
   // member list
};
 class Derived : [virtual] [access-specifier] Base1,
 [virtual] [access-specifier] Base2, . . .
{
   // member list
};
```

After the tag (name) for the class, a colon appears followed by a list of base specifications. The base classes so named must have been declared previously. The base specifications may contain an access specifier, which is one of the keywords **public**, **protected** or **private**. These access specifiers appear before the base class name and apply only to that base class. These specifiers control the derived class's permission to use to members of the base class. See Member-Access Control for information on access to base class members. If the access specifier is omitted, the access to that base is considered **private**. The base specifications may contain the keyword **virtual** to indicate virtual inheritance. This keyword may appear before or after the access specifier, if any. If virtual inheritance is used, the base class is referred to as a virtual base class. For more information, see Virtual Base Classes.

Multiple base classes can be specified, separated by commas. If a single base class is specified, the inheritance model is Single inheritance.If more than one base class is specified, the inheritance model is called Multiple inheritance,

See Also
Reference
Derived Classes

Single Inheritance

In "single inheritance," a common form of inheritance, classes have only one base class. Consider the relationship illustrated in the following figure.

Simple Single-Inheritance Graph

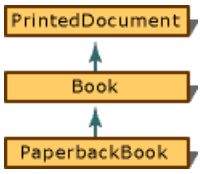

Note the progression from general to specific in the figure. Another common attribute found in the design of most class hierarchies is that the derived class has a "kind of" relationship with the base class. In the figure, a `Book` is a kind of a `PrintedDocument`, and a `PaperbackBook` is a kind of a `book`.

One other item of note in the figure: `Book` is both a derived class (from `PrintedDocument`) and a base class (`PaperbackBook` is derived from `Book`). A skeletal declaration of such a class hierarchy is shown in the following example:

```
// deriv_SingleInheritance.cpp
// compile with: /LD
class PrintedDocument {};

// Book is derived from PrintedDocument.
class Book : public PrintedDocument {};

// PaperbackBook is derived from Book.
class PaperbackBook : public Book {};
```

`PrintedDocument` is considered a "direct base" class to `Book`; it is an "indirect base" class to `PaperbackBook`. The difference is that a direct base class appears in the base list of a class declaration and an indirect base does not.

The base class from which each class is derived is declared before the declaration of the derived class. It is not sufficient to provide a forward-referencing declaration for a base class; it must be a complete declaration.

In the preceding example, the access specifier **public** is used. The meaning of public, protected, and private inheritance is described in Member-Access Control.

A class can serve as the base class for many specific classes, as illustrated in the following figure.

Sample of Directed Acyclic Graph

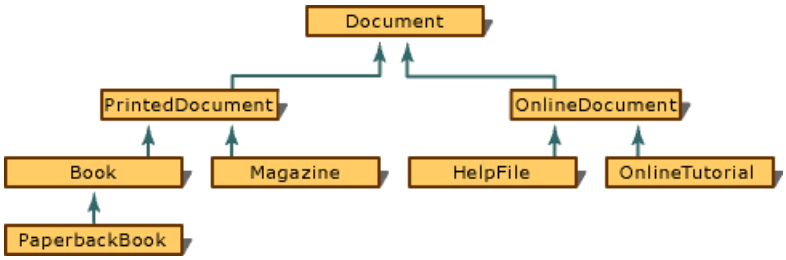

In the diagram shown above, called a "directed acyclic graph" (or "DAG"), some of the classes are base classes for more than one derived class. However, the reverse is not true: there is only one direct base class for any given derived class. The graph in the figure depicts a "single inheritance" structure.

Note
Directed acyclic graphs are not unique to single inheritance. They are also used to depict multiple-inheritance graphs. This topic is covered in Multiple Inheritance.

In inheritance, the derived class contains the members of the base class plus any new members you add. As a result, a derived class can refer to members of the base class (unless those members are redefined in the derived class). The scope-resolution

operator (**::**) can be used to refer to members of direct or indirect base classes when those members have been redefined in the derived class. Consider this example:

```
// deriv_SingleInheritance2.cpp
// compile with: /EHsc /c
#include <iostream>
using namespace std;
class Document {
public:
   char *Name;   // Document name.
   void PrintNameOf();   // Print name.
};

// Implementation of PrintNameOf function from class Document.
void Document::PrintNameOf() {
   cout << Name << endl;
}

class Book : public Document {
public:
   Book( char *name, long pagecount );
private:
   long  PageCount;
};

// Constructor from class Book.
Book::Book( char *name, long pagecount ) {
   Name = new char[ strlen( name ) + 1 ];
   strcpy_s( Name, sizeof(Name), name );
   PageCount = pagecount;
};
```

Note that the constructor for `Book`, (`Book::Book`), has access to the data member, `Name`. In a program, an object of type `Book` can be created and used as follows:

```
//  Create a new object of type Book. This invokes the
//   constructor Book::Book.
Book LibraryBook( "Programming Windows, 2nd Ed", 944 );

...

//  Use PrintNameOf function inherited from class Document.
LibraryBook.PrintNameOf();
```

As the preceding example demonstrates, class-member and inherited data and functions are used identically. If the implementation for class `Book` calls for a reimplementation of the `PrintNameOf` function, the function that belongs to the `Document` class can be called only by using the scope-resolution (**::**) operator:

```
// deriv_SingleInheritance3.cpp
// compile with: /EHsc /LD
#include <iostream>
using namespace std;

class Document {
public:
   char *Name;          // Document name.
   void  PrintNameOf() {}  // Print name.
};

class Book : public Document {
   Book( char *name, long pagecount );
   void PrintNameOf();
   long  PageCount;
};
```

```
void Book::PrintNameOf() {
   cout << "Name of book: ";
   Document::PrintNameOf();
}
```

Pointers and references to derived classes can be implicitly converted to pointers and references to their base classes if there is an accessible, unambiguous base class. The following code demonstrates this concept using pointers (the same principle applies to references):

```
// deriv_SingleInheritance4.cpp
// compile with: /W3
struct Document {
   char *Name;
   void PrintNameOf() {}
};

class PaperbackBook : public Document {};

int main() {
   Document * DocLib[10];   // Library of ten documents.
   for (int i = 0 ; i < 10 ; i++)
      DocLib[i] = new Document;
}
```

In the preceding example, different types are created. However, because these types are all derived from the `Document` class, there is an implicit conversion to `Document *`. As a result, `DocLib` is a "heterogeneous list" (a list in which not all objects are of the same type) containing different kinds of objects.

Because the `Document` class has a `PrintNameOf` function, it can print the name of each book in the library, although it may omit some of the information specific to the type of document (page count for `Book`, number of bytes for `HelpFile`, and so on).

Note

Forcing the base class to implement a function such as `PrintNameOf` is often not the best design. Virtual Functions offers other design alternatives.

See Also

Reference

Overview of Derived Classes

Multiple Inheritance

Multiple Inheritance

Later versions of C++ introduced a "multiple inheritance" model for inheritance. In a multiple-inheritance graph, the derived classes may have a number of direct base classes. Consider the graph in the following figure.

Simple Multiple-Inheritance Graph

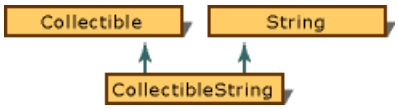

The diagram in the figure shows a class, `CollectibleString`. It is like a `Collectible` (something that can be contained in a collection), and it is like a `String`. Multiple inheritance is a good solution to this kind of problem (where a derived class has attributes of more than one base class) because it is easy to form a `CollectibleCustomer`, `CollectibleWindow`, and so on.

If the properties of either class are not required for a particular application, either class can be used alone or in combination with other classes. Therefore, given the hierarchy depicted in the above figure, you can form noncollectible strings and collectibles that are not strings. This flexibility is not possible using single inheritance.

See Also

Reference

Multiple Base Classes
Overview of Derived Classes
Multiple Inheritance

Virtual Base Class Hierarchies

Some class hierarchies are broad but have many things in common. The common code is implemented in a base class, whereas the specific code is in the derived classes.

It is important for the base classes to establish a protocol through which the derived classes can attain maximum functionality. These protocols are commonly implemented using virtual functions. Sometimes the base class provides a default implementation for such functions. In a class hierarchy such as the `Document` hierarchy in the figure Sample of Directed Acyclic Graph (see Single Inheritance), two useful functions are `Identify` and `WhereIs`.

When called, the `Identify` function returns a correct identification, appropriate to the kind of document: For a `Book`, a function call such as `doc->Identify()` must return the ISBN number; however, for a `HelpFile`, a product name and revision number are probably more appropriate. Similarly, `WhereIs` should return a row and shelf for a `Book`, but for a `HelpFile` it should return a disk location — perhaps a directory and filename.

It is important that all implementations of the `Identify` and `WhereIs` functions return the same kind of information. In this case, a character string is appropriate.

These functions are implemented as virtual functions and then invoked using a pointer to a base class. The binding to the actual code occurs at run time, selecting the correct `Identify` or `WhereIs` function.

See Also

Reference

Overview of Derived Classes

Class Protocol Implementation

Classes can be implemented to enforce a protocol. These classes are called "abstract classes" because no object of the class type can be created. They exist solely for derivation.

Classes are abstract classes if they contain pure virtual functions or if they inherit pure virtual functions and do not provide an implementation for them all. Pure virtual functions are virtual functions declared with the *pure-specifier* (`= 0`), as follows:

```
virtual char *Identify() = 0;
```

The base class, `Document`, might impose the following protocol on all derived classes:

- An appropriate `Identify` function must be implemented.
- An appropriate `WhereIs` function must be implemented.

By specifying such a protocol when designing the `Document` class, the class designer can be assured that no nonabstract class can be implemented without `Identify` and `WhereIs` functions. The `Document` class, therefore, contains these declarations:

```
// deriv_ClassProtocolImplementation.cpp
// compile with: /LD
class Document {
public:
    //  Requirements for derived classes: They must implement
    //   these functions.
    virtual char *Identify() = 0;
    virtual char *WhereIs() = 0;
};
```

See Also

Reference

Overview of Derived Classes

Base Classes

The inheritance process creates a new derived class that is made up of the members of the base class(es) plus any new members added by the derived class. In a multiple-inheritance, it is possible to construct an inheritance graph where the same base class is part of more than one of the derived classes. The following figure shows such a graph.

Multiple Instances of a Single Base Class

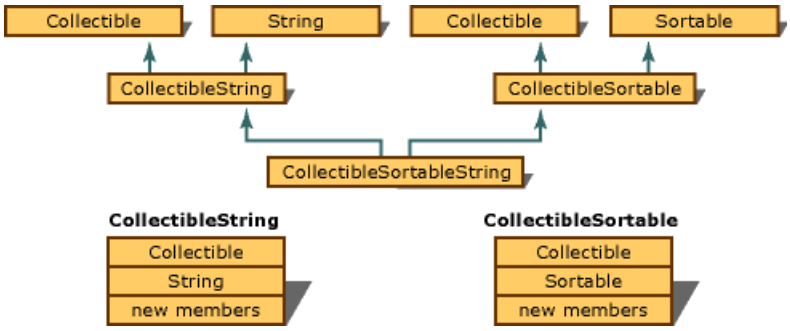

In the figure, pictorial representations of the components of `CollectibleString` and `CollectibleSortable` are shown. However, the base class, `Collectible`, is in `CollectibleSortableString` through the `CollectibleString` path and the `CollectibleSortable` path. To eliminate this redundancy, such classes can be declared as virtual base classes when they are inherited.

For information about declaring virtual base classes and how objects with virtual base classes are composed, see Virtual Base Classes.

See Also

Reference

Overview of Derived Classes

Multiple Base Classes

As described in Multiple Inheritance, a class can be derived from more than one base class. In a multiple-inheritance model (where classes are derived from more than one base class), the base classes are specified using the *base-list* grammar element (see the Grammar section in Overview). For example, the class declaration for `CollectionOfBook`, derived from `Collection` and `Book`, can be specified:

```
// deriv_MultipleBaseClasses.cpp
// compile with: /LD
class Collection {
};
class Book {};
class CollectionOfBook : public Book, public Collection {
    // New members
};
```

The order in which base classes are specified is not significant except in certain cases where constructors and destructors are invoked. In these cases, the order in which base classes are specified affects the following:

- The order in which initialization by constructor takes place. If your code relies on the `Book` portion of `CollectionOfBook` to be initialized before the `Collection` part, the order of specification is significant. Initialization takes place in the order the classes are specified in the *base-list*.
- The order in which destructors are invoked to clean up. Again, if a particular "part" of the class must be present when the other part is being destroyed, the order is significant. Destructors are called in the reverse order of the classes specified in the *base-list*.

Note
The order of specification of base classes can affect the memory layout of the class. Do not make any programming decisions based on the order of base members in memory.

When specifying the *base-list*, you cannot specify the same class name more than once. However, it is possible for a class to be an indirect base to a derived class more than once.

See Also

Reference

Derived Classes

Virtual Base Classes

Because a class can be an indirect base class to a derived class more than once, C++ provides a way to optimize the way such base classes work. Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritance.

Each nonvirtual object contains a copy of the data members defined in the base class. This duplication wastes space and requires you to specify which copy of the base class members you want whenever you access them.

When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use it as a virtual base.

When declaring a virtual base class, the **virtual** keyword appears in the base lists of the derived classes.

Consider the class hierarchy in the following figure, which illustrates a simulated lunch line.

Simulated Lunch-Line Graph

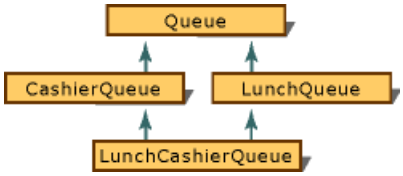

In the figure, `Queue` is the base class for both `CashierQueue` and `LunchQueue`. However, when both classes are combined to form `LunchCashierQueue`, the following problem arises: the new class contains two subobjects of type `Queue`, one from `CashierQueue` and the other from `LunchQueue`. The following figure shows the conceptual memory layout (the actual memory layout might be optimized).

Simulated Lunch-Line Object

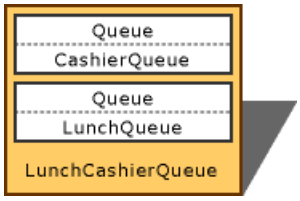

Note that there are two `Queue` subobjects in the `LunchCashierQueue` object. The following code declares `Queue` to be a virtual base class:

```
// deriv_VirtualBaseClasses.cpp
// compile with: /LD
class Queue {};
class CashierQueue : virtual public Queue {};
class LunchQueue : virtual public Queue {};
class LunchCashierQueue : public LunchQueue, public CashierQueue {};
```

The `virtual` keyword ensures that only one copy of the subobject `Queue` is included (see the following figure).

Simulated Lunch-Line Object with Virtual Base Classes

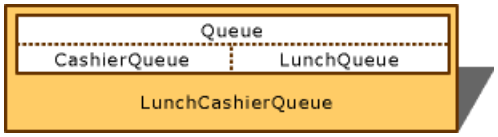

A class can have both a virtual component and a nonvirtual component of a given type. This happens in the conditions illustrated in the following figure.

Virtual and Nonvirtual Components of the Same Class

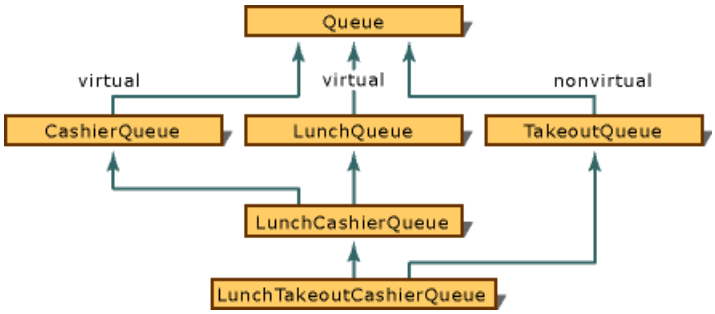

In the figure, `CashierQueue` and `LunchQueue` use `Queue` as a virtual base class. However, `TakeoutQueue` specifies `Queue` as a base class, not a virtual base class. Therefore, `LunchTakeoutCashierQueue` has two subobjects of type `Queue`: one from the inheritance path that includes `LunchCashierQueue` and one from the path that includes `TakeoutQueue`. This is illustrated in the following figure.

Object Layout with Virtual and Nonvirtual Inheritance

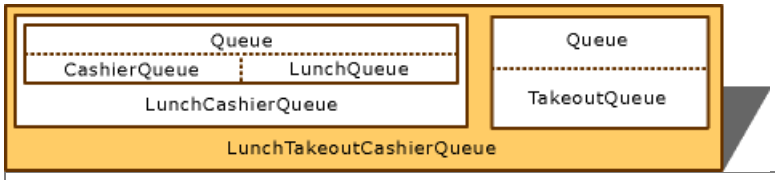

Note

Virtual inheritance provides significant size benefits when compared with nonvirtual inheritance. However, it can introduce e xtra processing overhead.

If a derived class overrides a virtual function that it inherits from a virtual base class, and if a constructor or a destructor for the derived base class calls that function using a pointer to the virtual base class, the compiler may introduce additional hidden "vtordisp" fields into the classes with virtual bases. The /vd0 compiler option suppresses the addition of the hidden vtordisp constructor/destructor displacement member. The /vd1 compiler option, the default, enables them where they are necessary. Turn off vtordisps only if you are sure that all class constructors and destructors call virtual functions virtually.

The /vd compiler option affects an entire compilation module. Use the **vtordisp** pragma to suppress and then reenable vtordisp fields on a class-by-class basis:

```
#pragma vtordisp( off )
class GetReal : virtual public { ... };
#pragma vtordisp( on )
```

See Also

Reference

Multiple Base Classes

Name Ambiguities

Multiple inheritance introduces the possibility for names to be inherited along more than one path. The class-member names along these paths are not necessarily unique. These name conflicts are called "ambiguities."

Any expression that refers to a class member must make an unambiguous reference. The following example shows how ambiguities develop:

```
// deriv_NameAmbiguities.cpp
// compile with: /LD
// Declare two base classes, A and B.
class A {
public:
    unsigned a;
    unsigned b();
};

class B {
public:
    unsigned a();  // Note that class A also has a member "a"
    int b();       //  and a member "b".
    char c;
};

// Define class C as derived from A and B.
class C : public A, public B {};
```

Given the preceding class declarations, code such as the following is ambiguous because it is unclear whether `b` refers to the `b` in `A` or in `B`:

```
C *pc = new C;

pc->b();
```

Consider the preceding example. Because the name `a` is a member of both class `A` and class `B`, the compiler cannot discern which `a` designates the function to be called. Access to a member is ambiguous if it can refer to more than one function, object, type, or enumerator.

The compiler detects ambiguities by performing tests in this order:

1. If access to the name is ambiguous (as just described), an error message is generated.
2. If overloaded functions are unambiguous, they are resolved. (For more information about function overloading ambiguity, see Argument Matching.)
3. If access to the name violates member-access permission, an error message is generated. (For more information, see Member-Access Control.)

When an expression produces an ambiguity through inheritance, you can manually resolve it by qualifying the name in question with its class name. To make the preceding example compile properly with no ambiguities, use code such as:

```
C *pc = new C;

pc->B::a();
```

Note

When `C` is declared, it has the potential to cause errors when `B` is referenced in the scope of `C`. No error is issued, however, until an unqualified reference to `B` is actually made in `C`'s scope.

See Also

Reference

Ambiguities and Virtual Base Classes

If virtual base classes are used, functions, objects, types, and enumerators can be reached through multiple-inheritance paths. Because there is only one instance of the base class, there is no ambiguity when accessing these names.

The following figure shows how objects are composed using virtual and nonvirtual inheritance.

Virtual vs. Nonvirtual Derivation

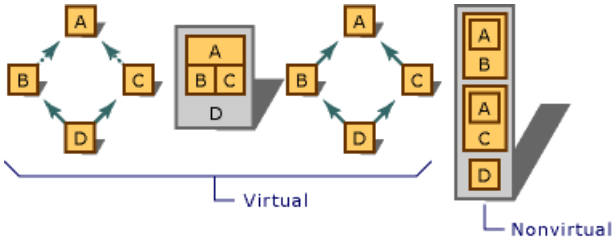

In the figure, accessing any member of class `A` through nonvirtual base classes causes an ambiguity; the compiler has no information that explains whether to use the subobject associated with `B` or the subobject associated with `C`. However, when `A` is specified as a virtual base class, there is no question which subobject is being accessed.

See Also

Reference

Name Ambiguities

Dominance

It is possible for more than one name (function, object, or enumerator) to be reached through an inheritance graph. Such cases are considered ambiguous with nonvirtual base classes. They are also ambiguous with virtual base classes, unless one of the names "dominates" the others.

A name dominates another name if it is defined in both classes and one class is derived from the other. The dominant name is the name in the derived class; this name is used when an ambiguity would otherwise have arisen, as shown in the following example:

```
// deriv_Dominance.cpp
// compile with: /LD
class A {
public:
    int a;
};

class B : public virtual A {
public:
    int a();
};

class C : public virtual A {};

class D : public B, public C {
public:
    D() { a(); } // Not ambiguous. B::a() dominates A::a.
};
```

See Also

Reference

Name Ambiguities

Ambiguous Conversions

Explicit and implicit conversions from pointers or references to class types can cause ambiguities. The next figure, Ambiguous Conversion of Pointers to Base Classes, shows the following:

- The declaration of an object of type `D`.
- The effect of applying the address-of operator (**&**) to that object. Note that the address-of operator always supplies the base address of the object.
- The effect of explicitly converting the pointer obtained using the address-of operator to the base-class type `A`. Note that coercing the address of the object to type `A*` does not always provide the compiler with enough information as to which subobject of type `A` to select; in this case, two subobjects exist.

Ambiguous Conversion of Pointers to Base Classes

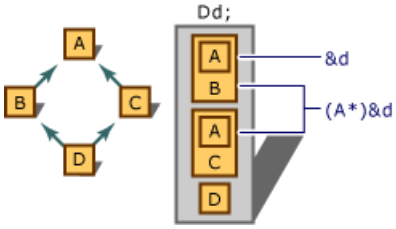

The conversion to type `A*` (pointer to `A`) is ambiguous because there is no way to discern which subobject of type `A` is the correct one. Note that you can avoid the ambiguity by explicitly specifying which subobject you mean to use, as follows:

```
(A *)(B *)&d       // Use B subobject.
(A *)(C *)&d       // Use C subobject.
```

See Also

Reference

Name Ambiguities

Virtual Functions

A virtual function is a member function that you expect to be redefined in derived classes. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

Virtual functions ensure that the correct function is called for an object, regardless of the expression used to make the function call.

Suppose a base class contains a function declared as virtual and a derived class defines the same function. The function from the derived class is invoked for objects of the derived class, even if it is called using a pointer or reference to the base class. The following example shows a base class that provides an implementation of the `PrintBalance` function and two derived classes

```
// deriv_VirtualFunctions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Account {
public:
   Account( double d ) { _balance = d; }
   virtual double GetBalance() { return _balance; }
   virtual void PrintBalance() { cerr << "Error. Balance not available for base type." << e
ndl; }
private:
    double _balance;
};

class CheckingAccount : public Account {
public:
   CheckingAccount(double d) : Account(d) {}
   void PrintBalance() { cout << "Checking account balance: " << GetBalance() << endl; }
};

class SavingsAccount : public Account {
public:
   SavingsAccount(double d) : Account(d) {}
   void PrintBalance() { cout << "Savings account balance: " << GetBalance(); }
};

int main() {
   // Create objects of type CheckingAccount and SavingsAccount.
   CheckingAccount *pChecking = new CheckingAccount( 100.00 ) ;
   SavingsAccount  *pSavings  = new SavingsAccount( 1000.00 );

   // Call PrintBalance using a pointer to Account.
   Account *pAccount = pChecking;
   pAccount->PrintBalance();

   // Call PrintBalance using a pointer to Account.
   pAccount = pSavings;
   pAccount->PrintBalance();
}
```

In the preceding code, the calls to `PrintBalance` are identical, except for the object `pAccount` points to. Because `PrintBalance` is virtual, the version of the function defined for each object is called. The `PrintBalance` function in the derived classes `CheckingAccount` and `SavingsAccount` "override" the function in the base class `Account`.

If a class is declared that does not provide an overriding implementation of the `PrintBalance` function, the default implementation from the base class `Account` is used.

Functions in derived classes override virtual functions in base classes only if their type is the same. A function in a derived class cannot differ from a virtual function in a base class in its return type only; the argument list must differ as well.

When calling a function using pointers or references, the following rules apply:

- A call to a virtual function is resolved according to the underlying type of object for which it is called.
- A call to a nonvirtual function is resolved according to the type of the pointer or reference.

The following example shows how virtual and nonvirtual functions behave when called through pointers:

```
// deriv_VirtualFunctions2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Base {
public:
   virtual void NameOf();   // Virtual function.
   void InvokingClass();   // Nonvirtual function.
};

// Implement the two functions.
void Base::NameOf() {
   cout << "Base::NameOf\n";
}

void Base::InvokingClass() {
   cout << "Invoked by Base\n";
}

class Derived : public Base {
public:
   void NameOf();   // Virtual function.
   void InvokingClass();   // Nonvirtual function.
};

// Implement the two functions.
void Derived::NameOf() {
   cout << "Derived::NameOf\n";
}

void Derived::InvokingClass() {
   cout << "Invoked by Derived\n";
}

int main() {
   // Declare an object of type Derived.
   Derived aDerived;

   // Declare two pointers, one of type Derived * and the other
   //  of type Base *, and initialize them to point to aDerived.
   Derived *pDerived = &aDerived;
   Base    *pBase    = &aDerived;

   // Call the functions.
   pBase->NameOf();           // Call virtual function.
   pBase->InvokingClass();    // Call nonvirtual function.
   pDerived->NameOf();        // Call virtual function.
   pDerived->InvokingClass(); // Call nonvirtual function.
}
```

Output

```
Derived::NameOf
Invoked by Base
Derived::NameOf
Invoked by Derived
```

Note that regardless of whether the `NameOf` function is invoked through a pointer to `Base` or a pointer to `Derived`, it calls the function for `Derived`. It calls the function for `Derived` because `NameOf` is a virtual function, and both `pBase` and `pDerived` point to an object of type `Derived`.

Because virtual functions are called only for objects of class types, you cannot declare global or static functions as **virtual**.

The **virtual** keyword can be used when declaring overriding functions in a derived class, but it is unnecessary; overrides of virtual functions are always virtual.

Virtual functions in a base class must be defined unless they are declared using the *pure-specifier*. (For more information about pure virtual functions, see Abstract Classes.)

The virtual function-call mechanism can be suppressed by explicitly qualifying the function name using the scope-resolution operator (**::**). Consider the earlier example involving the `Account` class. To call `PrintBalance` in the base class, use code such as the following:

```
CheckingAccount *pChecking = new CheckingAccount( 100.00 );

pChecking->Account::PrintBalance();  //  Explicit qualification.

Account *pAccount = pChecking;  // Call Account::PrintBalance

pAccount->Account::PrintBalance();   //  Explicit qualification.
```

Both calls to `PrintBalance` in the preceding example suppress the virtual function-call mechanism.

See Also

Reference

Access to Virtual Functions

virtual (C++)

The **virtual** keyword declares a virtual function or a virtual base class.

```
virtual [type-specifiers] member-function-declarator
virtual [access-specifier] base-class-name
```

Parameters

type-specifiers

Specifies the return type of the virtual member function.

member-function-declarator

Declares a member function.

access-specifier

Defines the level of access to the base class, **public**, **protected** or **private**. Can appear before or after the **virtual** keyword.

base-class-name

Identifies a previously declared class type.

Remarks

See Virtual Functions and Virtual Base Classes for more information.

Also see the following keywords: class, private, public, and protected.

See Also

Reference

C++ Keywords

Explicit Overrides (C++)

If the same virtual function is declared in two or more interfaces and if a class is derived from these interfaces, you can explicitly override each virtual function.

For information on explicit overrides in managed code using the new managed syntax, see Explicit Overrides.

Example

The following code example illustrates how to use explicit overrides:

```
// deriv_ExplicitOverrides.cpp
// compile with: /GR
extern "C" int printf_s(const char *, ...);

__interface IMyInt1 {
   void mf1();
   void mf1(int);
   void mf2();
   void mf2(int);
};

__interface IMyInt2 {
   void mf1();
   void mf1(int);
   void mf2();
   void mf2(int);
};

class CMyClass : public IMyInt1, public IMyInt2 {
public:
   void IMyInt1::mf1() {
      printf_s("In CMyClass::IMyInt1::mf1()\n");
   }

   void IMyInt1::mf1(int) {
      printf_s("In CMyClass::IMyInt1::mf1(int)\n");
   }

   void IMyInt1::mf2();
   void IMyInt1::mf2(int);

   void IMyInt2::mf1() {
      printf_s("In CMyClass::IMyInt2::mf1()\n");
   }

   void IMyInt2::mf1(int) {
         printf_s("In CMyClass::IMyInt2::mf1(int)\n");
   }

   void IMyInt2::mf2();
   void IMyInt2::mf2(int);
};

void CMyClass::IMyInt1::mf2() {
   printf_s("In CMyClass::IMyInt1::mf2()\n");
}

void CMyClass::IMyInt1::mf2(int) {
   printf_s("In CMyClass::IMyInt1::mf2(int)\n");
}

void CMyClass::IMyInt2::mf2() {
   printf_s("In CMyClass::IMyInt2::mf2()\n");
}
```

```
void CMyClass::IMyInt2::mf2(int) {
   printf_s("In CMyClass::IMyInt2::mf2(int)\n");
}

int main() {
   IMyInt1 *pIMyInt1 = new CMyClass();
   IMyInt2 *pIMyInt2 = dynamic_cast<IMyInt2 *>(pIMyInt1);

   pIMyInt1->mf1();
   pIMyInt1->mf1(1);
   pIMyInt1->mf2();
   pIMyInt1->mf2(2);
   pIMyInt2->mf1();
   pIMyInt2->mf1(3);
   pIMyInt2->mf2();
   pIMyInt2->mf2(4);

   delete pIMyInt1;
}
```

Output

```
In CMyClass::IMyInt1::mf1()
In CMyClass::IMyInt1::mf1(int)
In CMyClass::IMyInt1::mf2()
In CMyClass::IMyInt1::mf2(int)
In CMyClass::IMyInt2::mf1()
In CMyClass::IMyInt2::mf1(int)
In CMyClass::IMyInt2::mf2()
In CMyClass::IMyInt2::mf2(int)
```

See Also

Reference

Derived Classes

__interface

Microsoft Specific

A Visual C++ interface can be defined as follows:

- Can inherit from zero or more base interfaces.
- Cannot inherit from a base class.
- Can only contain public, pure virtual methods.
- Cannot contain constructors, destructors, or operators.
- Cannot contain static methods.
- Cannot contain data members; properties are allowed.

```
modifier __interface interface-name {interface-definition};
```

Remarks

A C++ class or struct could be implemented with these rules, but **__interface** enforces them.

For example, the following is a sample interface definition:

```
__interface IMyInterface {
   HRESULT CommitX();
   HRESULT get_X(BSTR* pbstrName);
};
```

For information on managed interfaces, see interface class.

Notice that you do not have to explicitly indicate that the `CommitX` and `get_X` functions are pure virtual. An equivalent declaration for the first function would be:

```
virtual HRESULT CommitX() = 0;
```

__interface implies the novtable **__declspec** modifier.

Example

The following sample shows how to use properties declared in an interface.

```
// deriv_interface.cpp
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include <string.h>
#include <comdef.h>
#include <stdio.h>

[module(name="test")];

[ object, uuid("00000000-0000-0000-0000-000000000001"), library_block ]
__interface IFace {
   [ id(0) ] int int_data;
   [ id(5) ] BSTR bstr_data;
};

[ coclass, uuid("00000000-0000-0000-0000-000000000002") ]
class MyClass : public IFace {
private:
    int m_i;
```

```
    BSTR m_bstr;

public:
    MyClass()
    {
        m_i = 0;
        m_bstr = 0;
    }

    ~MyClass()
    {
        if (m_bstr)
            ::SysFreeString(m_bstr);
    }

    int get_int_data()
    {
        return m_i;
    }

    void put_int_data(int _i)
    {
        m_i = _i;
    }

    BSTR get_bstr_data()
    {
        BSTR bstr = ::SysAllocString(m_bstr);
        return bstr;
    }

    void put_bstr_data(BSTR bstr)
    {
        if (m_bstr)
            ::SysFreeString(m_bstr);
        m_bstr = ::SysAllocString(bstr);
    }
};

int main()
{
    _bstr_t bstr("Testing");
    CoInitialize(NULL);
    CComObject<MyClass>* p;
    CComObject<MyClass>::CreateInstance(&p);
    p->int_data = 100;
    printf_s("p->int_data = %d\n", p->int_data);
    p->bstr_data = bstr;
    printf_s("bstr_data = %S\n", p->bstr_data);
}
```

Output

```
p->int_data = 100
bstr_data = Testing
```

END Microsoft Specific

See Also

Reference

C++ Keywords

Other Resources

Interface Attributes

__super

Microsoft Specific

Allows you to explicitly state that you are calling a base-class implementation for a function that you are overriding.

```
__super::member_function();
```

Remarks

All accessible base-class methods are considered during the overload resolution phase, and the function that provides the best match is the one that is called.

__super can only appear within the body of a member function.

__super cannot be used with a using declaration. See using Declaration for more information.

With the introduction of attributes that inject code, your code might contain one or more base classes whose names you may not know but that contain methods that you wish to call.

Example

```
// deriv_super.cpp
// compile with: /c
struct B1 {
   void mf(int) {}
};

struct B2 {
   void mf(short) {}

   void mf(char) {}
};

struct D : B1, B2 {
   void mf(short) {
      __super::mf(1);   // Calls B1::mf(int)
      __super::mf('s');   // Calls B2::mf(char)
   }
};
```

END Microsoft Specific

See Also

Reference

C++ Keywords

Abstract Classes (C++)

Abstract classes act as expressions of general concepts from which more specific classes can be derived. You cannot create an object of an abstract class type; however, you can use pointers and references to abstract class types.

A class that contains at least one pure virtual function is considered an abstract class. Classes derived from the abstract class must implement the pure virtual function or they, too, are abstract classes.

A virtual function is declared as "pure" by using the *pure-specifier* syntax (described in Class Protocol Implementation). Consider the example presented in Virtual Functions. The intent of class `Account` is to provide general functionality, but objects of type `Account` are too general to be useful. Therefore, `Account` is a good candidate for an abstract class:

```
// deriv_AbstractClasses.cpp
// compile with: /LD
class Account {
public:
   Account( double d );   // Constructor.
   virtual double GetBalance();   // Obtain balance.
   virtual void PrintBalance() = 0;   // Pure virtual function.
private:
    double _balance;
};
```

The only difference between this declaration and the previous one is that `PrintBalance` is declared with the pure specifier (`= 0`).

See Also

Reference

Derived Classes

Restrictions on Using Abstract Classes

Abstract classes cannot be used for:

- Variables or member data
- Argument types
- Function return types
- Types of explicit conversions

Another restriction is that if the constructor for an abstract class calls a pure virtual function, either directly or indirectly, the result is undefined. However, constructors and destructors for abstract classes can call other member functions.

Pure virtual functions can be defined for abstract classes, but they can be called directly only by using the syntax:

abstract-class-name **::** *function-name***( )**

This helps when designing class hierarchies whose base class(es) include pure virtual destructors, because base class destructors are always called in the process of destroying an object. Consider the following example:

```
// Declare an abstract base class with a pure virtual destructor.
// deriv_RestrictionsonUsingAbstractClasses.cpp
class base {
public:
    base() {}
    virtual ~base()=0;
};

// Provide a definition for destructor.
base::~base() {}

class derived:public base {
public:
    derived() {}
    ~derived(){}
};

int main() {
    derived *pDerived = new derived;
    delete pDerived;
}
```

When the object pointed to by `pDerived` is deleted, the destructor for class `derived` is called and then the destructor for class `base` is called. The empty implementation for the pure virtual function ensures that at least some implementation exists for the function.

Note

In the preceding example, the pure virtual function `base::~base` is called implicitly from `derived::~derived`. It is also possible to call pure virtual functions explicitly using a fully qualified member-function name.

See Also

Reference

Abstract Classes (C++)

Summary of Scope Rules

This section supplements Scope by adding the concepts pertaining to classes.

The following topics are included:

- Ambiguity
- Global names
- Names and qualified names
- Function argument names
- Constructor initializers

See Also

Reference

Derived Classes

Ambiguity

The use of a name must be unambiguous within its scope (up to the point where overloading is determined). If the name denotes a function, the function must be unambiguous with respect to number and type of arguments. If the name remains unambiguous, member-access rules are applied.

See Also

Reference

Summary of Scope Rules

Global Names

A name of an object, function, or enumerator is global if it is introduced outside any function or class or prefixed by the global unary scope operator (**::**), and if it is not used in conjunction with any of these binary operators:

- Scope-resolution (**::**)
- Member-selection for objects and references (**.**)
- Member-selection for pointers (**–>**)

See Also

Reference

Summary of Scope Rules

Names and Qualified Names

Names used with the binary scope-resolution operator (**::**) are called "qualified names." The name specified after the binary scope-resolution operator must be a member of the class specified on the left of the operator or a member of its base class(es).

Names specified after the member-selection operator (**.** or **–>**) must be members of the class type of the object specified on the left of the operator or members of its base class(es). Names specified on the right of the member-selection operator (**–>**) can also be objects of another class type, provided that the left-hand side of **–>** is a class object and that the class defines an overloaded member-selection operator (**–>**) that evaluates to a pointer to some other class type. (This provision is discussed in more detail in Class Member Access.)

The compiler searches for names in the following order, stopping when the name is found:

1. Current block scope if name is used inside a function; otherwise, global scope.
2. Outward through each enclosing block scope, including the outermost function scope (which includes function arguments).
3. If the name is used inside a member function, the class's scope is searched for the name.
4. The class's base classes are searched for the name.
5. The enclosing nested class scope (if any) and its bases are searched. The search continues until the outermost enclosing class scope is searched.
6. Global scope is searched.

However, you can make modifications to this search order as follows:

1. Names preceded by **::** force the search to begin at global scope.
2. Names preceded by the **class**, **struct**, and **union** keywords force the compiler to search only for **class**, **struct**, or **union** names.
3. Names on the left side of the scope-resolution operator (**::**) can be only **class**, **struct**, **namespace**, or **union** names.

If the name refers to a nonstatic member but is used in a static member function, an error message is generated. Similarly, if the name refers to any nonstatic member in an enclosing class, an error message is generated because enclosed classes do not have enclosing-class **this** pointers.

See Also
Reference
Summary of Scope Rules

Function Argument Names

Function argument names in function definitions are considered to be in the scope of the outermost block of the function. Therefore, they are local names and go out of scope when the function is exited.

Function argument names in function declarations (prototypes) are in local scope of the declaration and go out of scope at the end of the declaration.

Default arguments are in the scope of the argument for which they are the default, as described in the preceding two paragraphs. However, they cannot access local variables or nonstatic class members. Default arguments are evaluated at the point of the function call, but they are evaluated in the function declaration's original scope. Therefore, the default arguments for member functions are always evaluated in class scope.

See Also

Reference

Summary of Scope Rules

Constructor Initializers

Constructor initializers (described in Initializing Bases and Members) are evaluated in the scope of the outermost block of the constructor for which they are specified. Therefore, they can use the constructor's argument names.

See Also

Reference

Summary of Scope Rules

Member-Access Control

With C++, you can specify the level of access to member data and functions. There are three levels of access: public, protected, and private. This section explains how access control applies to objects of class type and to derived classes. This section includes the following topics:

- Controlling access to class members
- Access specifiers
- Access specifiers for base classes
- Friend classes and functions
- Access to virtual functions
- Multiple access

See Also

Other Resources

C++ Language Reference

Controlling Access to Class Members

You can increase the integrity of software built with C++ by helping control access to class member data and functions. Class members can be declared as having private, protected, or public access, as shown in the following table:

Member-Access Control

Type of Access	Meaning
private	Class members declared as **private** can be used only by member functions and friends (classes or functions) of the class.
protected	Class members declared as **protected** can be used by member functions and friends (classes or functions) of the class. Additionally, they can be used by classes derived from the class.
public	Class members declared as **public** can be used by any function.

Access control helps prevent you from using objects in ways they were not intended to be used. This protection is lost when explicit type conversions (casts) are performed.

Note
Access control is equally applicable to all names: member functions, member data, nested classes, and enumerators.

The default access to class members (members of a class type declared using the **class** keyword) is private; the default access to **struct** and **union** members is public. For either case, the current access level can be changed using the **public**, **private**, or **protected** keyword.

See Also

Reference

Access Specifiers for Base Classes

private (C++)

```
private:
   [member-list]
private base-class
```

Remarks

When preceding a list of class members, the **private** keyword specifies that those members are accessible only from member functions and friends of the class. This applies to all members declared up to the next access specifier or the end of the class.

When preceding the name of a base class, the **private** keyword specifies that the public and protected members of the base class are private members of the derived class.

Default access of members in a class is private. Default access of members in a structure or union is public.

Default access of a base class is private for classes and public for structures. Unions cannot have base classes.

For related information, see friend, public, protected, and the member-access table in Controlling Access to Class Members.

/clr Specific

In CLR types, the C++ access specifier keywords (**public**, **private**, and **protected**) can affect the visibility of types and methods with regard to assemblies. For more information, see Type and Member Visibility.

Note
Files compiled with /LN are not affected by this behavior. In this case, all managed classes (either public or private) will be visible.

END /clr Specific

Example

```
// keyword_private.cpp
class BaseClass {
public:
   // privMem accessible from member function
   int pubFunc() { return privMem; }
private:
   void privMem;
};

class DerivedClass : public BaseClass {
public:
   void usePrivate( int i )
      { privMem = i; }   // C2248: privMem not accessible
                         // from derived class
};

class DerivedClass2 : private BaseClass {
public:
   // pubFunc() accessible from derived class
   int usePublic() { return pubFunc(); }
};

int main() {
   BaseClass aBase;
   DerivedClass aDerived;
   DerivedClass2 aDerived2;
   aBase.privMem = 1;     // C2248: privMem not accessible
   aDerived.privMem = 1;  // C2248: privMem not accessible
                          //    in derived class
   aDerived2.pubFunc();   // C2247: pubFunc() is private in
                          //    derived class
}
```

See Also

Reference

Controlling Access to Class Members

C++ Keywords

protected (C++)

```
protected:
   [member-list]
protected base-class
```

Remarks

The **protected** keyword specifies access to class members in the *member-list* up to the next access specifier (**public** or **private**) or the end of the class definition. Class members declared as **protected** can be used only by the following:

- Member functions of the class that originally declared these members.
- Friends of the class that originally declared these members.
- Classes derived with public or protected access from the class that originally declared these members.
- Direct privately derived classes that also have private access to protected members.

When preceding the name of a base class, the **protected** keyword specifies that the public and protected members of the base class are protected members of its derived classes.

Protected members are not as private as **private** members, which are accessible only to members of the class in which they are declared, but they are not as public as **public** members, which are accessible in any function.

Protected members that are also declared as **static** are accessible to any friend or member function of a derived class. Protected members that are not declared as **static** are accessible to friends and member functions in a derived class only through a pointer to, reference to, or object of the derived class.

For related information, see friend, public, private, and the member-access table in Controlling Access to Class Members.

/clr Specific

In CLR types, the C++ access specifier keywords (**public**, **private**, and **protected**) can affect the visibility of types and methods with regard to assemblies. For more information, see Type and Member Visibility.

Note
Files compiled with /LN are not affected by this behavior. In this case, all managed classes (either public or private) will be visible.

END /clr Specific

Example

```
// keyword_protected.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class X {
public:
   void setProtMemb( int i ) { m_protMemb = i; }
   void Display() { cout << m_protMemb << endl; }
protected:
   int  m_protMemb;
   void Protfunc() { cout << "\nAccess allowed\n"; }
} x;

class Y : public X {
public:
   void useProtfunc() { Protfunc(); }
} y;

int main() {
   // x.m_protMemb;         error, m_protMemb is protected
```

```
    x.setProtMemb( 0 );   // OK, uses public access function
    x.Display();
    y.setProtMemb( 5 );   // OK, uses public access function
    y.Display();
    // x.Protfunc();         error, Protfunc() is protected
    y.useProtfunc();      // OK, uses public access function
                         // in derived class
}
```

See Also

Reference

Controlling Access to Class Members

C++ Keywords

public (C++)

```
public:
   [member-list]
public base-class
```

Remarks

When preceding a list of class members, the **public** keyword specifies that those members are accessible from any function. This applies to all members declared up to the next access specifier or the end of the class.

When preceding the name of a base class, the **public** keyword specifies that the public and protected members of the base class are public and protected members, respectively, of the derived class.

Default access of members in a class is private. Default access of members in a structure or union is public.

Default access of a base class is private for classes and public for structures. Unions cannot have base classes.

For more information, see private, protected, friend, and the member-access table in Controlling Access to Class Members.

/clr Specific

In CLR types, the C++ access specifier keywords (**public**, **private**, and **protected**) can affect the visibility of types and methods with regard to assemblies. For more information, see Type and Member Visibility.

Note

Files compiled with /LN are not affected by this behavior. In this case, all managed classes (either public or private) will be visible.

END /clr Specific

Example

```
// keyword_public.cpp
class BaseClass {
public:
   int pubFunc() { return 0; }
};

class DerivedClass : public BaseClass {};

int main() {
   BaseClass aBase;
   DerivedClass aDerived;
   aBase.pubFunc();       // pubFunc() is accessible
                          //    from any function
   aDerived.pubFunc();    // pubFunc() is still public in
                          //    derived class
}
```

See Also

Reference

Controlling Access to Class Members

C++ Keywords

Access Specifiers

In class declarations, members can have access specifiers.

Grammar

access-specifier **:** *member-list* $_{\text{opt}}$

access-specifier: one of

private

public

protected

The *access-specifier* determines the access to the names that follow it, up to the next *access-specifier* or the end of the class declaration. The following figure illustrates this concept.

Access Control in Classes

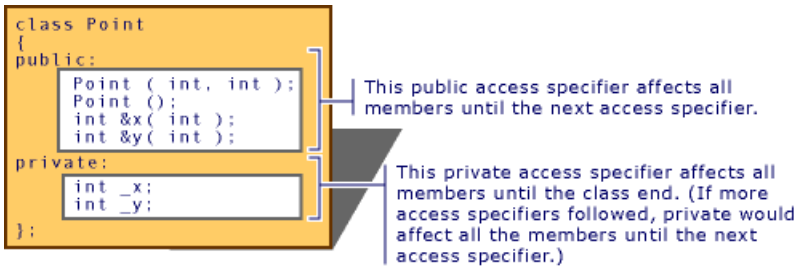

Although only two access specifiers are shown in the figure, there is no limit to the number of access specifiers in a given class declaration. For example, the Point class in the figure could just as easily be declared using multiple access specifiers as follows:

```
// access_specifiers1.cpp
class Point
{
public:                 // Declare public constructor.
    Point( int, int );
private:                // Declare private state variable.
    int _x;
public:                 // Declare public constructor.
    Point();
public:                 // Declare public accessor.
    int &x( int );
private:                // Declare private state variable.
    int _y;
public:                 // Declare public accessor.
    int &y( int );
};

int main()
{
}
```

Note that there is no specific order required for member access, as shown in the preceding example. The allocation of storage for objects of class types is implementation dependent, but members are guaranteed to be assigned successively higher memory addresses between access specifiers.

See Also

Reference

Member-Access Control

Access Specifiers for Base Classes

Two factors control which members of a base class are accessible in a derived class; these same factors control access to the inherited members in the derived class:

- Whether the derived class declares the base class using the **public** access specifier in the *class-head* (*class-head* is described in the Grammar section in Defining Class Types).
- What the access to the member is in the base class.

The following table shows the interaction between these factors and how to determine base-class member access.

Member Access in Base Class

private	protected	Public
Always inaccessible regardless of derivation access	Private in derived class if you use private derivation	Private in derived class if you use private derivation
	Protected in derived class if you use protected derivation	Protected in derived class if you use protected derivation
	Protected in derived class if you use public derivation	Public in derived class if you use public derivation

The following example illustrates this:

```
// access_specifiers_for_base_classes.cpp
class BaseClass
{
public:
    int PublicFunc();    // Declare a public member.
protected:
    int ProtectedFunc(); // Declare a protected member.
private:
    int PrivateFunc();   // Declare a private member.
};

// Declare two classes derived from BaseClass.
class DerivedClass1 : public BaseClass
{
};

class DerivedClass2 : private BaseClass
{
};

int main()
{
}
```

In `DerivedClass1`, the member function `PublicFunc` is a public member and `ProtectedFunc` is a protected member because `BaseClass` is a public base class. `PrivateFunc` is private to `BaseClass`, and it is inaccessible to any derived classes.

In `DerivedClass2`, the functions `PublicFunc` and `ProtectedFunc` are considered private members because `BaseClass` is a private base class. Again, `PrivateFunc` is private to `BaseClass`, and it is inaccessible to any derived classes.

You can declare a derived class without a base-class access specifier. In such a case, the derivation is considered private if the derived class declaration uses the **class** keyword. The derivation is considered public if the derived class declaration uses the **struct** keyword. For example, the following code:

```
class Derived : Base
...
```

is equivalent to:

```
class Derived : private Base
...
```

Similarly, the following code:

```
struct Derived : Base
...
```

is equivalent to:

```
struct Derived : public Base
...
```

Note that members declared as having private access are not accessible to functions or derived classes unless those functions or classes are declared using the **friend** declaration in the base class.

A **union** type cannot have a base class.

Note
When specifying a private base class, it is advisable to explicitly use the **private** keyword so users of the derived class unders tand the member access.

See Also

Reference

Member-Access Control

Access Control and Static Members

When you specify a base class as **private**, it affects only nonstatic members. Public static members are still accessible in the derived classes. However, accessing members of the base class using pointers, references, or objects can require a conversion, at which time access control is again applied. Consider the following example:

```
// access_control.cpp
class Base
{
public:
    int Print();             // Nonstatic member.
    static int CountOf();    // Static member.
};

// Derived1 declares Base as a private base class.
class Derived1 : private Base
{
};
// Derived2 declares Derived1 as a public base class.
class Derived2 : public Derived1
{
    int ShowCount();    // Nonstatic member.
};
// Define ShowCount function for Derived2.
int Derived2::ShowCount()
{
   // Call static member function CountOf explicitly.
    int cCount = Base::CountOf();     // OK.

   // Call static member function CountOf using pointer.
    cCount = this->CountOf();  // C2247. Conversion of
                               //  Derived2 * to Base * not
                               //  permitted.
    return cCount;
}
```

In the preceding code, access control prohibits conversion from a pointer to `Derived2` to a pointer to `Base`. The **this** pointer is implicitly of type `Derived2 *`. To select the `CountOf` function, **this** must be converted to type `Base *`. Such a conversion is not permitted because `Base` is a private indirect base class to `Derived2`. Conversion to a private base class type is acceptable only for pointers to immediate derived classes. Therefore, pointers of type `Derived1 *` can be converted to type `Base *`.

Note that calling the `CountOf` function explicitly, without using a pointer, reference, or object to select it, implies no conversion. Therefore, the call is allowed.

Members and friends of a derived class, *T*, can convert a pointer to *T* to a pointer to a private direct base class of *T*.

See Also

Reference

Access Specifiers for Base Classes

friend (C++)

In some circumstances, it is more convenient to grant member-level access to functions that are not members of a class or to all functions in a separate class. The **friend** keyword allows a function or class to gain access to the private and protected members of a class. You can declare friend functions or friend classes to access not only public members but also protected and private class members.

```
friend class-name;
friend function-declarator;
```

See Also

Reference

C++ Keywords

Friend Declarations

Defining Friend Functions in Class Declarations

Friend Functions

A **friend** function is a function that is not a member of a class but has access to the class's private and protected members. Friend functions are not considered class members; they are normal external functions that are given special access privileges. Friends are not in the class's scope, and they are not called using the member-selection operators (**.** and **->**) unless they are members of another class. A **friend** function is declared by the class that is granting access. The **friend** declaration can be placed anywhere in the class declaration. It is not affected by the access control keywords.

The following example shows a `Point` class and a friend function, `ChangePrivate`. The **friend** function has access to the private data member of the `Point` object it receives as a parameter.

Example

```
// friend_functions.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class Point
{
    friend void ChangePrivate( Point & );
public:
    Point( void ) : m_i(0) {}
    void PrintPrivate( void ){cout << m_i << endl; }

private:
    int m_i;
};

void ChangePrivate ( Point &i ) { i.m_i++; }

int main()
{
   Point sPoint;
   sPoint.PrintPrivate();
   ChangePrivate(sPoint);
   sPoint.PrintPrivate();
}
```

Output

```
0
1
```

See Also

Reference

friend (C++)

Class Member Functions and Classes as Friends

Class member functions can be declared as friends in other classes. Consider the following example:

```
// classes_as_friends1.cpp
// compile with: /c
class B;

class A {
public:
   int Func1( B& b );

private:
   int Func2( B& b );
};

class B {
private:
   int _b;

   // A::Func1 is a friend function to class B
   // so A::Func1 has access to all members of B
   friend int A::Func1( B& );
};

int A::Func1( B& b ) { return b._b; }   // OK
int A::Func2( B& b ) { return b._b; }   // C2248
```

In the preceding example, only the function `A::Func1( B& )` is granted friend access to class `B`. Therefore, access to the private member `_b` is correct in `Func1` of class `A` but not in `Func2`.

A **friend** class is a class all of whose member functions are friend functions of a class, that is, whose member functions have access to the other class's private and protected members. Suppose the **friend** declaration in class `B` had been:

```
friend class A;
```

In that case, all member functions in class `A` would have been granted friend access to class `B`. The following code is an example of a friend class:

```
// classes_as_friends2.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class YourClass {
friend class YourOtherClass;  // Declare a friend class
public:
   YourClass() : topSecret(0){}
   void printMember() { cout << topSecret << endl; }
private:
   int topSecret;
};

class YourOtherClass {
public:
   void change( YourClass& yc, int x ){yc.topSecret = x;}
};

int main() {
   YourClass yc1;
   YourOtherClass yoc1;
   yc1.printMember();
   yoc1.change( yc1, 5 );
```

```
    yc1.printMember();
}
```

Friendship is not mutual unless explicitly specified as such. In the above example, member functions of `YourClass` cannot access the private members of `YourOtherClass`.

A managed type cannot have any friend functions, friend classes, or friend interfaces.

Friendship is not inherited, meaning that classes derived from `YourOtherClass` cannot access `YourClass`'s private members. Friendship is not transitive, so classes that are friends of `YourOtherClass` cannot access `YourClass`'s private members.

The following figure shows four class declarations: `Base`, `Derived`, `aFriend`, and `anotherFriend`. Only class `aFriend` has direct access to the private members of `Base` (and to any members `Base` might have inherited).

Implications of friend Relationship

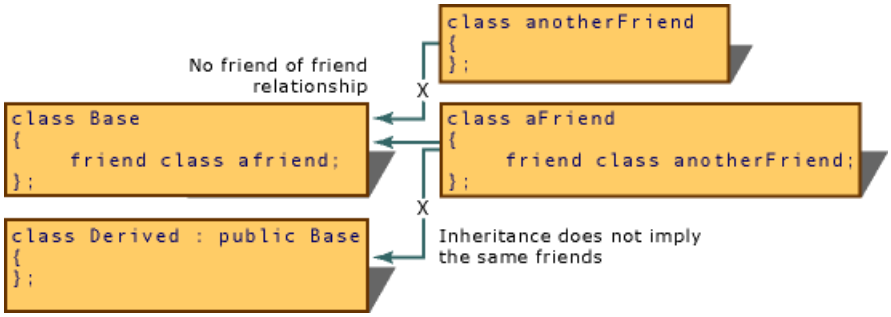

See Also

Reference

friend (C++)

Friend Declarations

If you declare a friend function that was not previously declared, that function is exported to the enclosing nonclass scope.

Functions declared in a friend declaration are treated as if they had been declared using the **extern** keyword. (For more information about **extern**, see Static Storage-Class Specifiers.)

Although functions with global scope can be declared as friends prior to their prototypes, member functions cannot be declared as friends before the appearance of their complete class declaration. The following code shows why this fails:

```
class ForwardDeclared;   // Class name is known.
class HasFriends
{
    friend int ForwardDeclared::IsAFriend();   // C2039 error expected
};
```

The preceding example enters the class name `ForwardDeclared` into scope, but the complete declaration — specifically, the portion that declares the function `IsAFriend` — is not known. Therefore, the **friend** declaration in class `HasFriends` generates an error.

To declare two classes that are friends of one another, the entire second class must be specified as a friend of the first class. The reason for this restriction is that the compiler has enough information to declare individual friend functions only at the point where the second class is declared.

Note
Although the entire second class must be a friend to the first class, you can select which functions in the first class will be friends of the second class.

See Also

Reference

friend (C++)

Defining Friend Functions in Class Declarations

Friend functions can be defined inside class declarations. These functions are inline functions, and like member inline functions they behave as though they were defined immediately after all class members have been seen but before the class scope is closed (the end of the class declaration).

Friend functions defined inside class declarations are not considered in the scope of the enclosing class; they are in file scope.

See Also

Reference

friend (C++)

Access to Virtual Functions

The access control applied to virtual functions is determined by the type used to make the function call. Overriding declarations of the function do not affect the access control for a given type. For example:

```
// access_to_virtual_functions.cpp
class VFuncBase
{
public:
    virtual int GetState() { return _state; }
protected:
    int _state;
};

class VFuncDerived : public VFuncBase
{
private:
    int GetState() { return _state; }
};

int main()
{
   VFuncDerived vfd;             // Object of derived type.
   VFuncBase *pvfb = &vfd;       // Pointer to base type.
   VFuncDerived *pvfd = &vfd;    // Pointer to derived type.
   int State;

   State = pvfb->GetState();     // GetState is public.
   State = pvfd->GetState();     // C2248 error expected; GetState is private;
}
```

In the preceding example, calling the virtual function `GetState` using a pointer to type `VFuncBase` calls `VFuncDerived::GetState`, and `GetState` is treated as public. However, calling `GetState` using a pointer to type `VFuncDerived` is an access-control violation because `GetState` is declared private in class `VFuncDerived`.

Caution

The virtual function `GetState` can be called using a pointer to the base class `VFuncBase`. This does not mean that the function called is the base-class version of that function.

See Also

Reference

Member-Access Control

Multiple Access

In multiple-inheritance lattices involving virtual base classes, a given name can be reached through more than one path. Because different access control can be applied along these different paths, the compiler chooses the path that gives the most access. See the following figure.

Access Along Paths of an Inheritance Graph

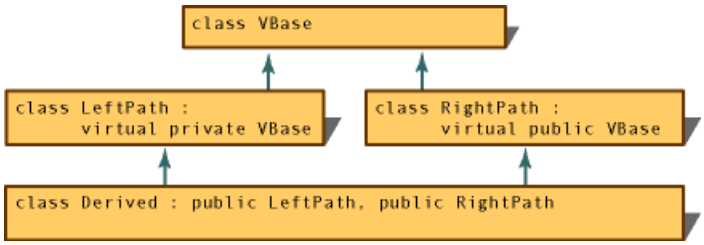

In the figure, a name declared in class `VBase` is always reached through class `RightPath`. The right path is more accessible because `RightPath` declares `VBase` as a public base class, whereas `LeftPath` declares `VBase` as private.

See Also

Reference

Member-Access Control

Special Member Functions

C++ defines several kinds of functions that can be declared only as class members — these are called "special member functions." These functions affect the way objects of a given class are created, destroyed, copied, and converted into objects of other types. Another important property of many of these functions is that they can be called implicitly (by the compiler).

The special member functions described in this section are as follows:

- Constructors. Enable automatic initialization of objects.
- Destructors. Perform cleanup after objects are explicitly or implicitly destroyed.
- Conversion functions. Convert between class types and other types.
- operator new function. Dynamically allocates storage.
- operator delete function. Releases storage allocated using the **new** operator.
- Assignment operator (operator=). Used when an assignment takes place.

The items in the preceding list can be user-defined for each class.

Special member functions obey the same access rules as other member functions. The access rules are described in Member-Access Control. The following table summarizes how member and friend functions behave.

Summary of Function Behavior

Function Type	Is Function Inherited from Base Class?	Can Function Be Virtual?	Can Function Return a Value?	Is Function a Member or Friend?	Will Compiler Generate Function if User Does Not?
Constructor	No	No	No	Member	Yes
Copy Constructor	No	No	No	Member	Yes
Destructor	No	Yes	No	Member	Yes
Conversion	Yes	Yes	No	Member	No
Assignment (**operator=**)	No	Yes	Yes	Member	Yes
new	Yes	No	**void***	Static member	No
delete	Yes	No	**void**	Static member	No
Other member functions	Yes	Yes	Yes	Member	No
Friend functions	No	No	Yes	Friend	No

See Also

Other Resources

C++ Language Reference

Constructors (C++)

A member function with the same name as its class is a constructor function. Constructors cannot return values. Specifying a constructor with a return type is an error, as is taking the address of a constructor.

If a class has a constructor, each object of that type is initialized with the constructor prior to use in a program. (For more information about initialization, see Initialization Using Special Member Functions.)

Constructors are called at the point an object is created. Objects are created as:

- Global (file-scoped or externally linked) objects.
- Local objects, within a function or smaller enclosing block.
- Dynamic objects, using the **new** operator. The **new** operator allocates an object on the program heap or "free store."
- Temporary objects created by explicitly calling a constructor. (For more information, see Temporary Objects.)
- Temporary objects created implicitly by the compiler. (For more information, see Temporary Objects.)
- Data members of another class. Creating objects of class type, where the class type is composed of other class-type variables, causes each object in the class to be created.
- Base class subobject of a class. Creating objects of derived class type causes the base class components to be created.

Example

```
// constructors.cpp
// compile with: /c
class MyClass {
public:
   MyClass(){}
   MyClass(int i) : m_i(i) {}

private:
   int m_i;
};
```

See Also

Reference

Special Member Functions

What a Constructor Does

A constructor performs various tasks that are not visible to you as the programmer, even if you write no code for the constructor. These tasks are all associated with building a complete and correct instance of class type.

In Microsoft C++ (and some other implementations of C++), a constructor:

- Initializes the object's virtual base pointer(s) (vbptr). This step is performed if the class is derived from virtual base classes.
- Calls base class and member constructors in the order of declaration.
- Initializes the object's virtual function pointers (vfptr). This step is performed if the class has or inherits virtual functions. Virtual function pointers point to the class's virtual function table (v-table) and allow correct binding of virtual function calls to code.
- Executes optional code in the body of the constructor function.

When the constructor is finished, the allocated memory is an object of a given class type. Because of the steps the constructor performs, "late binding" in the form of virtual functions can be resolved at the point of a virtual function call. The constructor has also constructed base classes and has constructed composed objects (objects included as data members). Late binding is the mechanism by which C++ implements polymorphic behavior for objects.

See Also

Reference

Constructors (C++)

Rules for Declaring Constructors

A constructor has the same name as its class. Any number of constructors can be declared, subject to the rules of overloaded functions. (For more information, see Overloading.)

```
class-name  (  argument-declaration-list  )
```

Remarks

The *argument-declaration-list* may be empty.

C++ defines two special kinds of constructors, default and copy constructors, described in the following table.

Default and Copy Constructors

Kind of Construction	Arguments	Purpose
Default constructor	Can be called with no arguments	Construct a default object of the class type
Copy constructor	Can accept a single argument of reference to same class type	Copy objects of the class type

Default constructors can be called with no arguments. However, you can declare a default constructor with an argument list, provided all arguments have defaults. Similarly, copy constructors must accept a single argument of reference to the same class type. More arguments can be supplied, provided all subsequent arguments have defaults.

If you do not supply any constructors, the compiler attempts to generate a default constructor. If you do not supply a copy constructor, the compiler attempts to generate one. These compiler-generated constructors are considered public member functions. An error is generated if you specify a copy constructor with a first argument that is an object and not a reference.

A compiler-generated default constructor sets up the object (initializes vftables and vbtables, as described previously), and it calls the default constructors for base classes and members, but it takes no other action. Base class and member constructors are called only if they exist, are accessible, and are unambiguous.

A compiler-generated copy constructor sets up a new object and performs a memberwise copy of the contents of the object to be copied. If base class or member constructors exist, they are called; otherwise, bitwise copying is performed.

If all base and member classes of a class *type* have copy constructors that accept a **const** argument, the compiler-generated copy constructor accepts a single argument of type **const** *type***&**. Otherwise, the compiler-generated copy constructor accepts a single argument of type *type***&**.

You can use a constructor to initialize a **const** or **volatile** object, but the constructor itself cannot be declared as **const** or **volatile**. The only legal storage class for a constructor is **inline**; use of any other storage-class modifier, including the **__declspec** keyword, with a constructor causes a compiler error.

The stdcall calling convention is used on static member functions and global functions declared with the **__stdcall** keyword, and that do not use a variable argument list. When you use the **__stdcall** keyword on a non-static member function, such as a constructor, the compiler will use the thiscall calling convention."

Constructors of base classes are not inherited by derived classes. When an object of derived class type is created, it is constructed starting with the base class components; then it moves to the derived class components. The compiler uses each base class's constructor as that part of the complete object is initialized (except in cases of virtual derivation, as described in Initializing Base Classes).

See Also

Reference

Constructors (C++)

Explicitly Called Constructors

Constructors can be explicitly called in a program to create objects of a given type. For example, to create two `Point` objects that describe the ends of a line, the following code can be written:

```
DrawLine( Point( 13, 22 ), Point( 87, 91 ) );
```

Two objects of type `Point` are created, passed to the function `DrawLine`, and destroyed at the end of the expression (the function call).

Another context in which a constructor is explicitly called is in an initialization:

```
Point pt = Point( 7, 11 );
```

An object of type `Point` is created and initialized using the constructor that accepts two arguments of type **int**.

Objects that are created by calling constructors explicitly, as in the preceding two examples, are unnamed and have a lifetime of the expression in which they are created. This is discussed in greater detail in Temporary Objects.

See Also

Reference

Rules for Declaring Constructors

Calling Member Functions and Virtual Functions from Within Constructors

It is usually safe to call any member function from within a constructor because the object has been completely set up (virtual tables have been initialized and so on) prior to the execution of the first line of user code. However, it is potentially unsafe for a member function to call a virtual member function for an abstract base class during construction or destruction.

Constructors can call virtual functions. When virtual functions are called, the function invoked is the function defined for the constructor's own class (or inherited from its bases). The following example shows what happens when a virtual function is called from within a constructor:

```
// specl_calling_virtual_functions.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class Base
{
public:
    Base();             // Default constructor.
    virtual void f();   // Virtual member function.
};

Base::Base()
{
    cout << "Constructing Base sub-object\n";
    f();                // Call virtual member function
}                       //  from inside constructor.

void Base::f()
{
    cout << "Called Base::f()\n";
}

class Derived : public Base
{
public:
    Derived();          // Default constructor.
    void f();           // Implementation of virtual
};                      //  function f for this class.

Derived::Derived()
{
    cout << "Constructing Derived object\n";
}

void Derived::f()
{
    cout << "Called Derived::f()\n";
}

int main()
{
    Derived d;
}
```

When the preceding program is run, the declaration `Derived d` causes the following sequence of events:

1. The constructor for class `Derived` (`Derived::Derived`) is called.
2. Prior to entering the body of the `Derived` class's constructor, the constructor for class `Base` (`Base::Base`) is called.
3. `Base::Base` calls the function `f`, which is a virtual function. Ordinarily, `Derived::f` would be called because the object `d` is

of type `Derived`. Because the `Base::Base` function is a constructor, the object is not yet of the `Derived` type, and `Base::f` is called.

See Also

Reference

Rules for Declaring Constructors

Constructors and Arrays

Arrays are constructed only using the default constructor. Default constructors are constructors that either accept no arguments or for which all arguments have a default. Arrays are always constructed in ascending order. The initialization for each member of the array is done using the same constructor.

See Also

Reference

Constructors (C++)

Order of Construction

For derived classes and classes that have class-type member data, the order in which construction occurs helps you understand what portions of the object you can use in any given constructor.

See Also

Reference

Constructors (C++)

Construction and Inheritance

An object of derived type is constructed from the base class to the derived class by calling the constructors for each class in order. Each class's constructor can rely on its base classes being completely constructed.

For a complete description of initialization, including the order of initialization, see Initializing Bases and Members.

See Also

Reference

Order of Construction

Construction and Composed Classes

Classes that contain class-type data members are called "composed classes." When an object of a composed class type is created, the constructors for the contained classes are called before the class's own constructor.

For a more information about this kind of initialization, see Initializing Bases and Members.

See Also

Reference

Order of Construction

Destructors (C++)

"Destructor" functions are the inverse of constructor functions. They are called when objects are destroyed (deallocated). Designate a function as a class's destructor by preceding the class name with a tilde (**~**). For example, the destructor for class `String` is declared: `~String()`.

In a /clr compilation, the destructor has a special role in releasing managed and unmanaged resources. See Destructors and Finalizers in Visual C++ for more information.

Example

The destructor is commonly used to "clean up" when an object is no longer necessary. Consider the following declaration of a `String` class:

```
// spec1_destructors.cpp
#include <string.h>

class String {
public:
   String( char *ch );  // Declare constructor
   ~String();           //  and destructor.
private:
   char    *_text;
   size_t  sizeOfText;
};

// Define the constructor.
String::String( char *ch ) {
   sizeOfText = strlen( ch ) + 1;

   // Dynamically allocate the correct amount of memory.
   _text = new char[ sizeOfText ];

   // If the allocation succeeds, copy the initialization string.
   if( _text )
      strcpy_s( _text, sizeOfText, ch );
}

// Define the destructor.
String::~String() {
   // Deallocate the memory that was previously reserved
   //  for this string.
   if (_text)
      delete[] _text;
}

int main() {
   String str("The piper in the glen...");
}
```

In the preceding example, the destructor `String::~String` uses the **delete** operator to deallocate the space dynamically allocated for text storage.

See Also

Reference

Special Member Functions

Declaring Destructors

Destructors are functions with the same name as the class but preceded by a tilde (**~**).

```
~class-name()
class-name::~class-name()
```

Remarks

The first form of the syntax is used for destructors declared or defined inside a class declaration; the second form is used for destructors defined outside a class declaration.

Several rules govern the declaration of destructors. Destructors:

- Do not accept arguments.
- Cannot specify any return type (including **void**).
- Cannot return a value using the **return** statement.
- Cannot be declared as **const**, **volatile**, or **static**. However, they can be invoked for the destruction of objects declared as **const**, **volatile**, or **static**.
- Can be declared as **virtual**. Using virtual destructors, you can destroy objects without knowing their type — the correct destructor for the object is invoked using the virtual function mechanism. Note that destructors can also be declared as pure virtual functions for abstract classes.

See Also

Reference

Destructors (C++)

Using Destructors

Destructors are called when one of the following events occurs:

- An object allocated using the **new** operator is explicitly deallocated using the **delete** operator. When objects are deallocated using the **delete** operator, memory is freed for the "most derived object," or the object that is a complete object and not a subobject representing a base class. This "most-derived object" deallocation is guaranteed to work only with virtual destructors. Deallocation may fail in multiple-inheritance situations where the type information does not correspond to the underlying type of the actual object.
- A local (automatic) object with block scope goes out of scope.
- The lifetime of a temporary object ends.
- A program ends and global or static objects exist.
- The destructor is explicitly called using the destructor function's fully qualified name. (For more information, see Explicit Destructor Calls.)

The cases described in the preceding list ensure that all objects can be destroyed with user-defined methods.

If a base class or data member has an accessible destructor, and if a derived class does not declare a destructor, the compiler generates one. This compiler-generated destructor calls the base class destructor and the destructors for members of the derived type. Default destructors are public. (For more information about accessibility, see Access Specifiers for Base Classes.)

Destructors can freely call class member functions and access class member data. When a virtual function is called from a destructor, the function called is the function for the class currently being destroyed. (For more information, see Order of Destruction.)

There are two restrictions on the use of destructors. The first restriction is that you cannot take the address of a destructor. The second is that derived classes do not inherit their base class's destructors. Instead, as previously explained, they always override the base class's destructors.

See Also

Reference

Destructors (C++)

Order of Destruction

When an object goes out of scope or is deleted, the sequence of events in its complete destruction is as follows:

1. The class's destructor is called, and the body of the destructor function is executed.
2. Destructors for nonstatic member objects are called in the reverse order in which they appear in the class declaration. The optional member initialization list used in construction of these members does not affect the order of (construction or) destruction. (For more information about initializing members, see Initializing Bases and Members.)
3. Destructors for nonvirtual base classes are called in the reverse order of declaration.
4. Destructors for virtual base classes are called in the reverse order of declaration.

Example

```
// order_of_destruction.cpp
#include <stdio.h>

struct A1      { virtual ~A1() { printf("A1 dtor\n"); } };
struct A2 : A1 { virtual ~A2() { printf("A2 dtor\n"); } };
struct A3 : A2 { virtual ~A3() { printf("A3 dtor\n"); } };

struct B1      { ~B1() { printf("B1 dtor\n"); } };
struct B2 : B1 { ~B2() { printf("B2 dtor\n"); } };
struct B3 : B2 { ~B3() { printf("B3 dtor\n"); } };

int main() {
   A1 * a = new A3;
   delete a;
   printf("\n");

   B1 * b = new B3;
   delete b;
   printf("\n");

   B3 * b2 = new B3;
   delete b2;
}
```

Output

```
A3 dtor
A2 dtor
A1 dtor

B1 dtor

B3 dtor
B2 dtor
B1 dtor
```

See Also

Reference

Destructors (C++)

Destructors for Nonvirtual Base Classes

The destructors for nonvirtual base classes are called in the reverse order in which the base class names are declared. Consider the following class declaration:

```
class MultInherit : public Base1, public Base2
...
```

In the preceding example, the destructor for `Base2` is called before the destructor for `Base1`.

See Also

Reference

Order of Destruction

Destructors for Virtual Base Classes

Destructors for virtual base classes are called in the reverse order of their appearance in a directed acyclic graph (depth-first, left-to-right, postorder traversal). the following figure depicts an inheritance graph.

Inheritance Graph Showing Virtual Base Classes

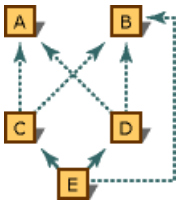

The following lists the class heads for the classes shown in the figure.

```
class A
class B
class C : virtual public A, virtual public B
class D : virtual public A, virtual public B
class E : public C, public D, virtual public B
```

To determine the order of destruction of the virtual base classes of an object of type `E`, the compiler builds a list by applying the following algorithm:

1. Traverse the graph left, starting at the deepest point in the graph (in this case, `E`).
2. Perform leftward traversals until all nodes have been visited. Note the name of the current node.
3. Revisit the previous node (down and to the right) to find out whether the node being remembered is a virtual base class.
4. If the remembered node is a virtual base class, scan the list to see whether it has already been entered. If it is not a virtual base class, ignore it.
5. If the remembered node is not yet in the list, add it to the bottom of the list.
6. Traverse the graph up and along the next path to the right.
7. Go to step 2.
8. When the last upward path is exhausted, note the name of the current node.
9. Go to step 3.
10. Continue this process until the bottom node is again the current node.

Therefore, for class `E`, the order of destruction is:

1. The nonvirtual base class `E`.
2. The nonvirtual base class `D`.
3. The nonvirtual base class `C`.
4. The virtual base class `B`.
5. The virtual base class `A`.

This process produces an ordered list of unique entries. No class name appears twice. Once the list is constructed, it is walked in reverse order, and the destructor for each of the classes in the list from the last to the first is called.

The order of construction or destruction is primarily important when constructors or destructors in one class rely on the other component being created first or persisting longer — for example, if the destructor for `A` (in the figure shown above) relied on `B` still being present when its code executed, or vice versa.

Such interdependencies between classes in an inheritance graph are inherently dangerous because classes derived later can

alter which is the leftmost path, thereby changing the order of construction and destruction.

See Also

Reference

Order of Destruction

Explicit Destructor Calls

Calling a destructor explicitly is seldom necessary. However, it can be useful to perform cleanup of objects placed at absolute addresses. These objects are commonly allocated using a user-defined **new** operator that takes a placement argument. The **delete** operator cannot deallocate this memory because it is not allocated from the free store (for more information, see The new and delete Operators). A call to the destructor, however, can perform appropriate cleanup. To explicitly call the destructor for an object, `s`, of class `String`, use one of the following statements:

```
s.String::~String();     // Nonvirtual call
ps->String::~String();   // Nonvirtual call

s.~String();       // Virtual call
ps->~String();     // Virtual call
```

The notation for explicit calls to destructors, shown in the preceding, can be used regardless of whether the type defines a destructor. This allows you to make such explicit calls without knowing if a destructor is defined for the type. An explicit call to a destructor where none is defined has no effect.

See Also

Reference

Destructors (C++)

Temporary Objects

In some cases, it is necessary for the compiler to create temporary objects. These temporary objects can be created for the following reasons:

- To initialize a **const** reference with an initializer of a type different from that of the underlying type of the reference being initialized.
- To store the return value of a function that returns a user-defined type. These temporaries are created only if your program does not copy the return value to an object. For example:

  ```
  UDT Func1();    //  Declare a function that returns a user-defined
                  //   type.

  ...

  Func1();        //  Call Func1, but discard return value.
                  //  A temporary object is created to store the return
                  //   value.
  ```

  Because the return value is not copied to another object, a temporary object is created. A more common case where temporaries are created is during the evaluation of an expression where overloaded operator functions must be called. These overloaded operator functions return a user-defined type that often is not copied to another object.

  Consider the expression `ComplexResult = Complex1 + Complex2 + Complex3`. The expression `Complex1 + Complex2` is evaluated, and the result is stored in a temporary object. Next, the expression *temporary* `+ Complex3` is evaluated, and the result is copied to `ComplexResult` (assuming the assignment operator is not overloaded).

- To store the result of a cast to a user-defined type. When an object of a given type is explicitly converted to a user-defined type, that new object is constructed as a temporary object.

Temporary objects have a lifetime that is defined by their point of creation and the point at which they are destroyed. Any expression that creates more than one temporary object eventually destroys them in the reverse order in which they were created. The points at which destruction occurs are shown in the following table.

Destruction Points for Temporary Objects

Reason Temporary Created	Destruction Point
Result of expression evaluation	All temporaries created as a result of expression evaluation are destroyed at the end of the expression statement (that is, at the semicolon), or at the end of the controlling expressions for **for**, **if**, **while**, **do**, and **switch** statements.
Result of expressions using the built-in (not overloaded) logical operators (**||** and **&&**)	Immediately after the right operand. At this destruction point, all temporary objects created by evaluation of the right operand are destroyed.
Initializing **const** references	If an initializer is not an l-value of the same type as the reference being initialized, a temporary of the underlying object type is created and initialized with the initialization expression. This temporary object is destroyed immediately after the reference object to which it is bound is destroyed.

See Also

Reference

Special Member Functions

Conversions

Objects of a given class type can be converted to objects of another type. This is done by constructing an object of the target class type from the source class type and copying the result to the target object. This process is called conversion by constructor. Objects can also be converted by user-supplied conversion functions.

When standard conversions (described in Standard Conversions) cannot completely convert from a given type to a class type, the compiler can select user-defined conversions to help complete the job. In addition to explicit type conversions, conversions take place when:

- An initializer expression is not the same type as the object being initialized.
- The type of argument used in a function call does not match the type of argument specified in the function declaration.
- The type of the object being returned from a function does not match the return type specified in the function declaration.
- Two expression operands must be of the same type.
- An expression controlling an iteration or selection statement requires a different type from the one supplied.

A user-defined conversion is applied only if it is unambiguous; otherwise, an error message is generated. Ambiguity is checked at the point of usage. Hence, if the features that cause ambiguity are not used, a class can be designated with potential ambiguities and not generate any errors. Although there are many situations in which ambiguities arise, these are two leading causes of ambiguities:

- A class type is derived using multiple inheritance, and it is unclear from which base class to select the conversion (see Ambiguity).
- An explicit type-conversion operator and a constructor for the same conversion exist (see Conversion Functions).

Both conversion by constructor and conversion by conversion functions obey access control rules, as described in Member-Access Control. Access control is tested only after the conversion is found to be unambiguous.

For additional information, see Conversion Constructors.

See Also
Reference
Special Member Functions

Conversion Constructors

A constructor that can be called with a single argument is used for conversions from the type of the argument to the class type. Such a constructor is called a conversion constructor. Consider the following example:

```
// spec1_conversion_constructors.cpp
class Point
{
public:
    Point();
    Point( int );
    //...
};

int main()
{
}
```

Sometimes a conversion is required but no conversion constructor exists in the class. These conversions cannot be performed by constructors. The compiler does not look for intermediate types through which to perform the conversion. For example, suppose a conversion exists from type `Point` to type `Rect` and a conversion exists from type **int** to type `Point`. The compiler does not supply a conversion from type **int** to type `Rect` by constructing an intermediate object of type `Point`.

See Also

Reference

Conversions

Conversions and Constants

Although constants for built-in types such as **int**, **long**, and **double** can appear in expressions, no constants of class types are allowed (this is partly because classes usually describe an object complicated enough to make notation inconvenient). However, if conversion constructors from built-in types are supplied, constants of these built-in types can be used in expressions, and the conversions cause correct behavior. For example, a `Money` class can have conversions from types **long** and **double**:

```
// spec1_conversions_and_constants.cpp
class Money
{
public:
    Money( long );
    Money( double );
   // ...
    Money operator+( const Money& );  // Overloaded addition operator.
};

int main()
{
}
```

Therefore, expressions such as the following can specify constant values:

```
Money AccountBalance = 37.89;
Money NewBalance = AccountBalance + 14L;
```

The second example involves the use of an overloaded addition operator, which is covered in the next section. Both examples cause the compiler to convert the constants to type `Money` before using them in the expressions.

See Also

Reference

Conversion Constructors

Drawbacks of Conversion Constructors

Because the compiler can select a conversion constructor implicitly, you relinquish control over what functions are called when. If it is essential to retain full control, do not declare any constructors that take a single argument; instead, define "helper" functions to perform conversions, as in the following example:

Example

```
// spec1_drawbacks_of_conversion_constructors.cpp
// compile with: /EHsc
#include <stdio.h>
#include <stdlib.h>

// Declare Money class.
class Money {
public:
   Money();

   // Define conversion functions that can only be called explicitly.
   static Money Convert( char * ch ) { return Money( ch ); }
   static Money Convert( double d )    { return Money( d ); };
   void  Print() { printf_s( "%f\n", _amount ); }

private:
   Money( char *ch ) { _amount = atof( ch ); }
   Money( double d ) { _amount = d; }
   double _amount;
};

int main() {
   // Perform a conversion from type char * to type Money.
   Money Acct = Money::Convert( "57.29" );
   Acct.Print();

   // Perform a conversion from type double to type Money.
   Acct = Money::Convert( 33.29 );
   Acct.Print();
}
```

Output

```
57.290000
33.290000
```

Comments

In the preceding code, the conversion constructors are private and cannot be used in type conversions. However, they can be invoked explicitly by calling the `Convert` functions. Because the `Convert` functions are static, they are accessible without referencing a particular object.

See Also

Reference

Conversion Constructors

explicit (C++)

This keyword is a declaration specifier that can only be applied to in-class constructor declarations. An explicit constructor cannot take part in implicit conversions. It can only be used to explicitly construct an object.

Example

The following program will fail to compile because of the **explicit** keyword. To resolve the error, remove the **explicit** keywords and adjust the code in `g`.

```
// spec1_explicit.cpp
// compile with: /EHsc
#include <stdio.h>

class C
{
public:
    int i;
    explicit C(const C&)   // an explicit copy constructor
    {
        printf_s("\nin the copy constructor");
    }
    explicit C(int i )   // an explicit constructor
    {
        printf_s("\nin the constructor");
    }

    C()
    {
        i = 0;
    }
};

class C2
{
public:
    int i;
    explicit C2(int i )   // an explicit constructor
    {
    }
};

C f(C c)
{   // C2558
    c.i = 2;
    return c;   // first call to copy constructor
}

void f2(C2)
{
}

void g(int i)
{
    f2(i);   // C2558
    // try the following line instead
    // f2(C2(i));
}

int main()
{
    C c, d;
    d = f(c);   // c is copied
}
```

Note
explicit on a constructor with multiple arguments has no effect, since such constructors cannot take part in implicit conversi ons. However, for the purpose of implicit conversion, explicit will have an effect if a constructor has multiple arguments and a ll but one of the arguments has a default value.

See Also

Reference

C++ Keywords

Conversions

Conversion Functions (C++)

In conversion by constructors, described in the previous section, objects of one type can be implicitly converted to a particular class type. This section describes a means by which you can provide explicit conversions from a given class type to another type. Conversion from a class type is often accomplished using conversion functions. Conversion functions use the following syntax:

Grammar
conversion-function-name:

operator *conversion-type-name* **()**

conversion-type-name:

type-specifier-list ptr-operator $_{\text{opt}}$

The following example specifies a conversion function that converts type `Money` to type `double`:

```
// spec1_conversion_functions1.cpp
struct Money {
   operator double() { return _amount; }

private:
    double _amount;
};

int main() {
   Money Account;
   double CashOnHand = Account;
}
```

The initialization of `CashOnHand` with `Account` causes a conversion from type `Account` to type `double`.

Conversion functions are often called "cast operators" because they (along with constructors) are the functions called when a cast is used. The following example uses a cast, or explicit conversion, to print the current value of an object of type `Money`:

```
cout << (double)Account << endl;
```

Conversion functions are inherited in derived classes. Conversion operators hide only base-class conversion operators that convert to exactly the same type. Therefore, a user-defined **operator int** function does not hide a user-defined **operator short** function in a base class.

Only one user-defined conversion function is applied when performing implicit conversions. If there is no explicitly defined conversion function, the compiler does not look for intermediate types into which an object can be converted.

If a conversion is required that causes an ambiguity, an error is generated. Ambiguities arise when more than one user-defined conversion is available or when a user-defined conversion and a built-in conversion exist.

Example

The following example illustrates a class declaration with a potential ambiguity:

```
// spec1_conversion_functions2.cpp
#include <string.h>
#include <stdio.h>

struct String {
   // Define constructor that converts from type char *.
   String( char *szBuffer, size_t sizeOfBuffer ) {
      if (sizeOfBuffer > sizeof(szBuffer) )
         return;
      strcpy_s( _text, sizeOfBuffer, szBuffer );
   }

   // Define conversion to type char *.
```

```
   operator char *() {
      return _text;
   }

   int operator==( const String &s )  {
      return !strcmp( _text, s._text );
   }

private:
   char _text[80];
};

int main() {
   String s( "abcd\0", 5 );
   char *ch = "efgh";

   // Cause the compiler to select a conversion.
   return s == ch;
}
```

In the expression `s == ch`, the compiler has two choices and no way of determining which is correct. It can convert `ch` to an object of type `String` using the constructor and then perform the comparison using the user-defined `operator==`. Or it can convert `s` to a pointer of type `char *` using the conversion function and then perform a comparison of the pointers.

Because neither choice is "more correct" than the other, the compiler cannot determine the meaning of the comparison expression, and it generates an error.

See Also

Reference

Conversions

Rules for Declaring Conversion Functions

The following four rules are used when declaring conversion functions (see Conversion Functions for syntax):

- Classes, enumerations, and **typedef** names cannot be declared in the *type-specifier-list*. Therefore, the following code generates an error:

  ```
  operator struct String{ char string_storage; }();
  ```

  Instead, declare the `String` structure prior to the conversion function.

- Conversion functions take no arguments. Specifying arguments generates an error.
- Conversion functions have the return type specified by the *conversion-type-name*; specifying any return type for a conversion function generates an error.
- Conversion functions can be declared as **virtual**.

See Also
Reference
Rules for Declaring Conversion Functions

The new and delete Operators

C++ supports dynamic allocation and deallocation of objects using the new and delete operators. These operators allocate memory for objects from a pool called the free store. The **new** operator calls the special function operator new, and the **delete** operator calls the special function operator delete.

In Visual C++ .NET 2002, the **new** function in the Standard C++ Library will support the behavior specified in the C++ standard, which is to throw a std::bad_alloc exception if the memory allocation fails.

In Visual C++ 2005, the C Runtime Library's **new** function will also throw a std::bad_alloc exception if the memory allocation fails.

If you still want the non-throwing version of **new** for the C Runtime Library, link your program with nothrownew.obj. However, when you link with nothrownew.obj, **new** in the Standard C++ Library will no longer function.

For a list of the library files that comprise the C Runtime Library and the Standard C++ Library, see C Run-Time Libraries.

See Also

Reference

Special Member Functions

The operator new Function

When a statement such as the following is encountered in a program, it translates into a call to the function **operator new**:

Remarks

```
char *pch = new char[BUFFER_SIZE];
```

If the request is for zero bytes of storage, **operator new** returns a pointer to a distinct object (that is, repeated calls to **operator new** return different pointers). If there is insufficient memory for the allocation request, **operator new** returns **NULL** or throws an exception (see The new and delete Operators for more information).

You can write a routine that attempts to free memory and retry the allocation; see _set_new_handler for more information. For more details on the recovery scheme, see the following topic, Handling Insufficient Memory Conditions.

The two scopes for **operator new** functions are described in the following table.

Scope for operator new Functions

Operator	Scope
::operator new	Global
class-name **::operator new**	Class

The first argument to **operator new** must be of type **size_t** (a type defined in STDDEF.H), and the return type is always **void ***.

The global **operator new** function is called when the **new** operator is used to allocate objects of built-in types, objects of class type that do not contain user-defined **operator new** functions, and arrays of any type. When the **new** operator is used to allocate objects of a class type where an **operator new** is defined, that class's **operator new** is called.

An **operator new** function defined for a class is a static member function (which cannot, therefore, be virtual) that hides the global **operator new** function for objects of that class type. Consider the case where **new** is used to allocate and set memory to a given value:

```
// spec1_the_operator_new_function1.cpp
#include <malloc.h>
#include <memory.h>

class Blanks
{
public:
    Blanks(){}
    void *operator new( size_t stAllocateBlock, char chInit );
};
void *Blanks::operator new( size_t stAllocateBlock, char chInit )
{
    void *pvTemp = malloc( stAllocateBlock );
    if( pvTemp != 0 )
        memset( pvTemp, chInit, stAllocateBlock );
    return pvTemp;
}
// For discrete objects of type Blanks, the global operator new function
// is hidden. Therefore, the following code allocates an object of type
// Blanks and initializes it to 0xa5
int main()
{
   Blanks *a5 = new(0xa5) Blanks;
   return a5 != 0;
}
```

The argument supplied in parentheses to **new** is passed to `Blanks::operator new` as the `chInit` argument. However, the global **operator new** function is hidden, causing code such as the following to generate an error:

```
Blanks *SomeBlanks = new Blanks;
```

In Visual C++ 5.0 and earlier, nonclass types and all arrays (regardless of whether they were of **class** type) allocated using the **new** operator always used the global **operator new** function.

Beginning with Visual C++ 5.0, the compiler supports member array **new** and **delete** operators in a class declaration. For example:

```
// spec1_the_operator_new_function2.cpp
class MyClass
{
public:
   void * operator new[] (size_t)
   {
      return 0;
   }
   void   operator delete[] (void*)
   {
   }
};

int main()
{
   MyClass *pMyClass = new MyClass[5];
   delete [] pMyClass;
}
```

See Also

Reference

The operator delete Function

new Operator (C++)

delete Operator (C++)

Handling Insufficient Memory Conditions

Testing for failed memory allocation can be done with code such as the following:

```
// insufficient_memory_conditions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
#define BIG_NUMBER 100000000
int main() {
   int *pI = new int[BIG_NUMBER];
   if( pI == 0x0 ) {
      cout << "Insufficient memory" << endl;
      return -1;
   }
}
```

There is another ways to handle failed memory allocation requests: write a custom recovery routine to handle such a failure, then register your function by calling the _set_new_handler run-time function.

See Also

Reference

The operator new Function

The operator delete Function

Memory that is dynamically allocated using the **new** operator can be freed using the **delete** operator. The delete operator calls the **operator delete** function, which frees memory back to the available pool. Using the **delete** operator also causes the class destructor (if there is one) to be called.

There are global and class-scoped **operator delete** functions. Only one **operator delete** function can be defined for a given class; if defined, it hides the global **operator delete** function. The global **operator delete** function is always called for arrays of any type.

The global **operator delete** function, if declared, takes a single argument of type **void ***, which contains a pointer to the object to deallocate. The return type is **void** (**operator delete** cannot return a value). Two forms exist for class-member **operator delete** functions:

```
void operator delete( void * );
void operator delete( void *, size_t );
```

Only one of the preceding two variants can be present for a given class. The first form works as described for global **operator delete**. The second form takes two arguments, the first of which is a pointer to the memory block to deallocate and the second of which is the number of bytes to deallocate. The second form is particularly useful when an **operator delete** function from a base class is used to delete an object of a derived class.

The **operator delete** function is static; therefore, it cannot be virtual. The **operator delete** function obeys access control, as described in Member-Access Control.

The following example shows user-defined **operator new** and **operator delete** functions designed to log allocations and deallocations of memory:

Example

```
// spec1_the_operator_delete_function1.cpp
// compile with: /EHsc
// arguments: 3
#include <iostream>
using namespace std;

int fLogMemory = 0;      // Perform logging (0=no; nonzero=yes)?
int cBlocksAllocated = 0;  // Count of blocks allocated.

// User-defined operator new.
void *operator new( size_t stAllocateBlock ) {
   static int fInOpNew = 0;   // Guard flag.

   if ( fLogMemory && !fInOpNew ) {
      fInOpNew = 1;
      clog << "Memory block " << ++cBlocksAllocated
          << " allocated for " << stAllocateBlock
          << " bytes\n";
      fInOpNew = 0;
   }
   return malloc( stAllocateBlock );
}

// User-defined operator delete.
void operator delete( void *pvMem ) {
   static int fInOpDelete = 0;   // Guard flag.
   if ( fLogMemory && !fInOpDelete ) {
      fInOpDelete = 1;
      clog << "Memory block " << cBlocksAllocated--
          << " deallocated\n";
      fInOpDelete = 0;
   }

   free( pvMem );
}
```

```
int main( int argc, char *argv[] ) {
   fLogMemory = 1;   // Turn logging on
   if( argc > 1 )
      for( int i = 0; i < atoi( argv[1] ); ++i ) {
         char *pMem = new char[10];
         delete[] pMem;
      }
   fLogMemory = 0;  // Turn logging off.
   return cBlocksAllocated;
}
```

Beginning with Visual C++ 5.0, the compiler supports member array **new** and **delete** operators in a class declaration. For example:

```
// spec1_the_operator_delete_function2.cpp
// compile with: /c
class X  {
public:
   void * operator new[] (size_t) {
      return 0;
   }
   void operator delete[] (void*) {}
};

void f() {
   X *pX = new X[5];
   delete [] pX;
}
```

Comment

The preceding code can be used to detect "memory leakage" — that is, memory that is allocated on the free store but never freed. To perform this detection, the global **new** and **delete** operators are redefined to count allocation and deallocation of memory.

See Also

Reference

The operator new Function
new Operator (C++)
delete Operator (C++)

Initialization Using Special Member Functions

This section describes initialization using special member functions. It expands on the following discussions of initialization:

- Initializing Aggregates which describes how to initialize arrays of nonclass types and objects of simple class types. These simple class types cannot have private or protected members, and they cannot have base classes.
- Constructors, which explains how to initialize class-type objects using special constructor functions.

The default method of initialization is to perform a bit-for-bit copy from the initializer into the object to be initialized. This technique is applicable only to:

- Objects of built-in types. For example:

```
int i = 100;
```

- Pointers. For example:

```
int i;
int *pi = &i;
```

- References. For example:

```
String sFileName( "FILE.DAT" );
String &rs = sFileName;
```

- Objects of class type, where the class has no private or protected members, no virtual functions, and no base classes. For example:

```
// spec1_special_member_functions.cpp
// compile with: /LD
struct Point
{
    int x, y;
};

Point pt = { 10, 20 };   // Static storage class only
```

Classes can specify more refined initialization by defining constructor functions. (For more information about declaring such functions, see Constructors.) If an object is of a class type that has a constructor, the object must be initialized, or there must be a default constructor. Objects that are not specifically initialized invoke the class's default constructor.

See Also

Reference

Special Member Functions

Explicit Initialization

C++ supports two forms of explicit initialization.

- Supplying an initializer list in parentheses:

  ```
  String sFileName( "FILE.DAT" );
  ```

  The items in the parenthesized list are considered arguments to the class constructor. This form of initialization enables initialization of an object with more than one value and can also be used in conjunction with the **new** operator. For example:

  ```
  Rect *pRect = new Rect( 10, 15, 24, 97 );
  ```

- Supplying a single initializer using the equal-sign initialization syntax. For example:

  ```
  String sFileName = "FILE.DAT";
  ```

  Although the preceding example works the same way as the example shown for `String` in the first list item, the syntax is not adaptable to use with objects allocated on the free store.

  The single expression on the right of the equal sign is taken as the argument to the class's copy constructor; therefore, it must be a type that can be converted to the class type.

  Note that because the equal sign (**=**) in the context of initialization is different from an assignment operator, overloading **operator=** has no effect on initialization.

The equal-sign initialization syntax is different from the function-style syntax, even though the generated code is identical in most cases. The difference is that when the equal-sign syntax is used, the compiler has to behave as if the following sequence of events were taking place:

- Creating a temporary object of the same type as the object being initialized.
- Copying the temporary object to the object.

The constructor must be accessible before the compiler can perform these steps. Even though the compiler can eliminate the temporary creation and copy steps in most cases, an inaccessible copy constructor causes equal-sign initialization to fail (under /Za, /Ze (Disable Language Extensions)). Consider the following example:

```
// spec1_explicit_initialization.cpp
// compile with: /Za
class anInt {
   anInt( const anInt &copy ) {}   // private copy constructor

public:
   anInt( int ) {}   // public constructor
};

int main() {
   // Access-control violation.
   // Attempt to reference private copy constructor.
   anInt myInt = 7;   // C2248

   anInt myInt2(7);   // Correct; no copy constructor called.
}
```

When a function is called, class-type arguments passed by value and objects returned by value are conceptually initialized using the form:

type-name name **=** *value*

For example:

```
String s = "C++";
```

Therefore, it follows that the argument type must be a type that can be converted to the class type being passed as an argument. The class's copy constructor, as well as user-defined conversion operators or constructors that accept the type of the actual argument, must be public.

In expressions that use the **new** operator, the objects allocated on the free store are conceptually initialized using the form:

type-name name **(** *initializer*1, *initializer*2, ... *initializer*n **)**

For example:

```
String *ps = new String( "C++" );
```

Initializers for base-class components and member objects of a class are also conceptually initialized this way. (For more information, see Initializing Bases and Members.)

See Also
Reference
Initialization Using Special Member Functions

Initializing Arrays

If a class has a constructor, arrays of that class are initialized by a constructor. If there are fewer items in the initializer list than elements in the array, the default constructor is used for the remaining elements. If no default constructor is defined for the class, the initializer list must be complete — that is, there must be one initializer for each element in the array.

Consider the `Point` class that defines two constructors:

```
// initializing_arrays1.cpp
class Point
{
public:
   Point()   // Default constructor.
   {
   }
   Point( int, int )   // Construct from two ints
   {
   }
};

// An array of Point objects can be declared as follows:
Point aPoint[3] = {
   Point( 3, 3 )     // Use int, int constructor.
};

int main()
{
}
```

The first element of `aPoint` is constructed using the constructor `Point( int, int )`; the remaining two elements are constructed using the default constructor.

Static member arrays (whether **const** or not) can be initialized in their definitions (outside the class declaration). For example:

```
// initializing_arrays2.cpp
class WindowColors
{
public:
    static const char *rgszWindowPartList[7];
};

const char *WindowColors::rgszWindowPartList[7] = {
    "Active Title Bar", "Inactive Title Bar", "Title Bar Text",
    "Menu Bar", "Menu Bar Text", "Window Background", "Frame"   };
int main()
{
}
```

See Also

Reference

Initialization Using Special Member Functions

Initializing Static Objects

Global static objects are initialized in the order they occur in the source. They are destroyed in the reverse order. Across translation units, however, the order of initialization is dependent on how the object files are arranged by the linker; the order of destruction still takes place in the reverse of that in which objects were constructed.

Local static objects are initialized when they are first encountered in the program flow, and they are destroyed in the reverse order at program termination. Destruction of local static objects occurs only if the object was encountered and initialized in the program flow.

See Also

Reference

Initialization Using Special Member Functions

Initializing Bases and Members

An object of a derived class is made up of a component that represents each base class and a component that is unique to the particular class. Objects of classes that have member objects may also contain instances of other classes. This section describes how these component objects are initialized when an object of the class type is created.

To perform the initialization, a list of initializers separated by commas is used. Each initializer consists of an identifier for a base class or a data member of the class being constructed, followed by an expression list in parentheses.

```
A() : MyBase(200, "A1"), value1(100), value2(100)
```

This syntax, used in constructors, is described more fully in the next section, Initializing Member Objects, and in Initializing Base Classes.

See Also

Reference

Initialization Using Special Member Functions

Initializing Member Objects

Classes can contain member objects of class type, but to ensure that initialization requirements for the member objects are met, one of the following conditions must be met:

- The contained object's class requires no constructor.
- The contained object's class has an accessible default constructor.
- The containing class's constructors all explicitly initialize the contained object.

The following example shows how to perform such an initialization:

```
// spec1_initializing_member_objects.cpp
// Declare a class Point.
class Point
{
public:
    Point( int x, int y ) { _x = x; _y = y; }
private:
    int _x, _y;
};

// Declare a rectangle class that contains objects of type Point.
class Rect
{
public:
    Rect( int x1, int y1, int x2, int y2 );
private:
    Point _topleft, _bottomright;
};

//  Define the constructor for class Rect. This constructor
//   explicitly initializes the objects of type Point.
Rect::Rect( int x1, int y1, int x2, int y2 ) :
_topleft( x1, y1 ), _bottomright( x2, y2 )
{
}

int main()
{
}
```

The `Rect` class, shown in the preceding example, contains two member objects of class `Point`. Its constructor explicitly initializes the objects `_topleft` and `_bottomright`. Note that a colon follows the closing parenthesis of the constructor (in the definition). The colon is followed by the member names and arguments with which to initialize the objects of type `Point`.

Note

The order in which the member initializers are specified in the constructor does not affect the order in which the members are constructed; the members are constructed in the order in which they are declared in the class.

Reference and **const** member objects must be initialized using the member initialization syntax shown in the Grammar section in Initializing Bases and Members. There is no other way to initialize these objects.

See Also

Reference

Initializing Bases and Members

Initializing Base Classes

Direct base classes are initialized in much the same way as member objects. Consider the following example:

```
// spec1_initializing_base_classes.cpp
// Declare class Window.
class MyClass
{
public:
   MyClass( int rSize )
   {
   }
};

//  Declare class DialogBox, derived from class MyClass
class DialogBox : public MyClass
{
public:
   DialogBox( int rSize );
};

//  Define the constructor for DialogBox. This constructor
//   explicitly initializes the MyClass subobject.
DialogBox::DialogBox( int rSize ) : MyClass( rSize )
{
}

int main()
{
}
```

Note that in the constructor for `DialogBox`, the `Window` base class is initialized using the argument `rSize`. This initialization consists of the name of the base class to initialize, followed by a parenthesized list of arguments to the class's constructor.

In initialization of base classes, the object that is not the subobject representing a base class's component is considered a "complete object." The complete object's class is considered the "most derived" class for the object.

The subobjects representing virtual base classes are initialized by the constructor for the most derived class. That means that where virtual derivation is specified, the most derived class must explicitly initialize the virtual base class, or the virtual base class must have a default constructor. Initializations for virtual base classes that appear in constructors for classes other than the most derived class are ignored.

Although initialization of base classes is usually restricted to direct base classes, a class constructor can initialize an indirect virtual base class.

See Also

Reference

Initializing Bases and Members

Initialization Order of Bases and Members

Base classes and member objects are initialized in the following order:

1. Virtual base classes are initialized in the order in which they appear in the directed acyclic graph. For information about using the directed acyclic graph to construct a list of unique subobjects, see Virtual Base Classes. (Note that these subobjects are destroyed by walking the same list in reverse.) For more information about how the directed acyclic graph is traversed, see Order of Destruction.
2. Nonvirtual base classes are initialized in the order in which they are declared in the class declaration.
3. Member objects are initialized in the order in which the objects are declared in the class.

The order in which base classes and member objects are initialized is not affected by the order in which the member initializers or base-class initializers appear in the *member-initializer-list* of the constructor.

See Also
Reference
Initializing Bases and Members

Scope of Initializers

Initializers for base classes and member objects are evaluated in the scope of the constructor with which they are declared. Therefore, they can refer implicitly to class-member data.

See Also

Reference

Initializing Bases and Members

Copying Class Objects

Two operations cause objects to be copied:

- Assignment. When one object's value is assigned to another object, the first object is copied to the second object. Therefore:

  ```
  Point a, b;
  ...
  a = b;
  ```

  causes the value of `b` to be copied to `a`.

- Initialization. Initialization occurs at the point of declaration of a new object, when arguments are passed to functions by value, and when values are returned from functions by value.

The programmer can define the semantics of "copy" for objects of class type. For example, consider the following code:

```
TextFile a, b;
a.Open( "FILE1.DAT" );
b.Open( "FILE2.DAT" );
b = a;
```

The preceding code could mean "copy the contents of FILE1.DAT to FILE2.DAT," or it could mean "ignore FILE2.DAT and make `b` a second handle to FILE1.DAT." The programmer is responsible for attaching appropriate copying semantics to each class.

Copying is done in one of two ways:

- Assignment (using the assignment operator, **operator=**).
- Initialization (using the copy constructor). (For more information about the copy constructor, see Rules for Declaring Constructors.)

Any given class can implement one or both copy methods. If neither method is implemented, assignment is handled as a member-by-member ("memberwise") assignment, and initialization is handled as a member-by-member initialization. Memberwise assignment is covered in more detail in Memberwise Assignment and Initialization.

The copy constructor takes a single argument of type *class-name***&**, where *class-name* is the name of the class for which the constructor is defined. For example:

```
// spec1_copying_class_objects.cpp
class Window
{
public:
    Window( const Window& ); // Declare copy constructor.
    // ...
};

int main()
{
}
```

Note

The type of the copy constructor's argument should be *const class-name***&** whenever possible. This prevents the copy constructor from accidentally changing the object from which it is copying. It also allows copying from **const** objects.

See Also

Reference

Special Member Functions

Compiler-Generated Copying

Compiler-generated copy constructors, like user-defined copy constructors, have a single argument of type "reference to *class-name*." An exception is when all base classes and member classes have copy constructors declared as taking a single argument of type **const** *class-name***&**. In such a case, the compiler-generated copy constructor's argument is also **const**.

When the argument type to the copy constructor is not **const**, initialization by copying a **const** object generates an error. The reverse is not true: If the argument is **const**, you can initialize by copying an object that is not **const**.

Compiler-generated assignment operators follow the same pattern with regard to **const.** They take a single argument of type *class-name***&** unless the assignment operators in all base and member classes take arguments of type **const** *class-name&*. In this case, the class's generated assignment operator takes a **const** argument.

Note
When virtual base classes are initialized by copy constructors, compiler-generated or user-defined, they are initialized only once: at the point when they are constructed.

The implications are similar to those of the copy constructor. When the argument type is not **const**, assignment from a **const** object generates an error. The reverse is not true: If a **const** value is assigned to a value that is not **const**, the assignment succeeds.

For more information about overloaded assignment operators, see Assignment.

See Also
Reference
Copying Class Objects

Memberwise Assignment and Initialization

The methods for default assignment and initialization are "memberwise assignment" and "memberwise initialization," respectively. Memberwise assignment consists of copying one object to the other, a member at a time, as if assigning each member individually. Memberwise initialization consists of copying one object to the other, a member at a time, as if initializing each member individually. The primary difference between the two is that memberwise assignment invokes each member's assignment operator (**operator=**), whereas memberwise initialization invokes each member's copy constructor.

Memberwise assignment is performed only by the assignment operator declared in the form:

type **&** *type* **:: operator=(** [**const** | **volatile**] *type***&)**

Default assignment operators for memberwise assignment cannot be generated if any of the following conditions exist:

- A member class has **const** members.
- A member class has reference members.
- A member class or its base class has a private assignment operator (**operator=**).
- A base class or member class has no assignment operator (**operator=**).

Default copy constructors for memberwise initialization cannot be generated if the class or one of its base classes has a private copy constructor or if any of the following conditions exist:

- A member class has **const** members.
- A member class has reference members.
- A member class or its base class has a private copy constructor.
- A base class or member class has no copy constructor.

The default assignment operators and copy constructors for a given class are always declared, but they are not defined unless both of the following conditions are met:

- The class does not provide a user-defined function for this copy.
- The program requires that the function be present. This requirement exists if an assignment or initialization is encountered that requires memberwise copying or if the address of the class's **operator=** function is taken.

If both of these conditions are not met, the compiler is not required to generate code for the default assignment operator and copy constructor functions (elimination of such code is an optimization performed by the Microsoft C++ compiler). Specifically, if the class declares a user-defined **operator=** that takes an argument of type "reference to *class-name*," no default assignment operator is generated. If the class declares a copy constructor, no default copy constructor is generated.

Therefore, for a given class `A`, the following declarations are always present:

```
//  Implicit declarations of copy constructor
//   and assignment operator.
A::A( const A& );
A& A::operator=( const A& );
```

The definitions are supplied only if required (according to the preceding criteria). The copy constructor functions shown in the preceding example are considered public member functions of the class.

Default assignment operators allow objects of a given class to be assigned to objects of a public base-class type. Consider the following code:

Example

```
// spec1_memberwise_assignment_and_initialization.cpp

#include<stdio.h>

class Account
{
```

```
protected:
    int _balance;
public:
   int getBalance()
   {
      return _balance;
   }
};

class Checking : public Account
{
private:
    int _fOverdraftProtect;
public:
    Checking(int balance, int fOverdraftProtect)
    {
        _balance = balance;
        _fOverdraftProtect = fOverdraftProtect;
    }
};

int main()
{
    Account account;
    Checking checking(1000, 1);
    account = checking;
    printf_s("Account balance = %d\n", account.getBalance());
}
```

Output

```
Account balance = 1000
```

Comment

In the preceding example, the assignment operator chosen is `Account::operator=`. Because the default `operator=` function takes an argument of type `Account&` (reference to `Account`), the `Account` subobject of `checking` is copied to `account`; `fOverdraftProtect` is not copied.

See Also

Reference

Copying Class Objects

Overloading

This section explains how to use C++ overloaded functions and overloaded operators. The following topics are included:

- Overview
- Restrictions on overloaded functions
- Declaration matching
- Argument matching
- Address of overloaded functions
- Overloaded operators
- General rules for operator overloading

See Also

Other Resources

C++ Language Reference

Overview of Overloading

With the C++ language, you can overload functions and operators. Overloading is the practice of supplying more than one definition for a given function name in the same scope. The compiler is left to pick the appropriate version of the function or operator based on the arguments with which it is called. For example, the function max is considered an overloaded function. It can be used in code such as the following:

```
// overview_overload.cpp
double max( double d1, double d2 )
{
   return ( d1 > d2 ) ? d1 : d2;
}

int max( int i1, int i2 )
{
   return ( i1 > i2 ) ? i1 : i2;
}
int main()
{
   int    i = max( 12, 8 );
   double d = max( 32.9, 17.4 );
}
```

In the first function call, where the maximum value of two variables of type `int` is being requested, the function `max( int, int )` is called. However, in the second function call, the arguments are of type `double`, so the function `max( double, double )` is called.

See Also

Reference

Overloading

Argument Type Differences

Overloaded functions differentiate between argument types that take different initializers. Therefore, an argument of a given type and a reference to that type are considered the same for the purposes of overloading. They are considered the same because they take the same initializers. For example, `max( double, double )` is considered the same as `max( double &, double & )`. Declaring two such functions causes an error.

Remarks

For the same reason, function arguments of a type modified by **const** or **volatile** are not treated differently than the base type for the purposes of overloading.

However, the function overloading mechanism can distinguish between references that are qualified by **const** and **volatile** and references to the base type. This makes code such as the following possible:

```
// argument_type_differences.cpp
// compile with: /EHsc /W3
// C4521 expected
#include <iostream>

using namespace std;
class Over {
public:
   Over() { cout << "Over default constructor\n"; }
   Over( Over &o ) { cout << "Over&\n"; }
   Over( const Over &co ) { cout << "const Over&\n"; }
   Over( volatile Over &vo ) { cout << "volatile Over&\n"; }
};

int main() {
   Over o1;            // Calls default constructor.
   Over o2( o1 );      // Calls Over( Over& ).
   const Over o3;      // Calls default constructor.
   Over o4( o3 );      // Calls Over( const Over& ).
   volatile Over o5;   // Calls default constructor.
   Over o6( o5 );      // Calls Over( volatile Over& ).
}
```

Output

```
Over default constructor
Over&
Over default constructor
const Over&
Over default constructor
volatile Over&
```

Pointers to **const** and **volatile** objects are also considered different from pointers to the base type for the purposes of overloading.

See Also

Reference

Overview of Overloading

Restrictions on Overloaded Functions

Several restrictions govern an acceptable set of overloaded functions:

- Any two functions in a set of overloaded functions must have different argument lists.
- Overloading functions with argument lists of the same types, based on return type alone, is an error.

  Microsoft Specific

You can overload **operator new** solely on the basis of return type — specifically, on the basis of the memory-model modifier specified.

END Microsoft Specific

- Member functions cannot be overloaded solely on the basis of one being static and the other nonstatic.
- **typedef** declarations do not define new types; they introduce synonyms for existing types. They do not affect the overloading mechanism. Consider the following code:

  ```
  typedef char * PSTR;

  void Print( char *szToPrint );
  void Print( PSTR szToPrint );
  ```

  The preceding two functions have identical argument lists. `PSTR` is a synonym for type **char ***. In member scope, this code generates an error.

- Enumerated types are distinct types and can be used to distinguish between overloaded functions.
- The types "array of " and "pointer to" are considered identical for the purposes of distinguishing between overloaded functions. This is true only for singly dimensioned arrays. Therefore, the following overloaded functions conflict and generate an error message:

  ```
  void Print( char *szToPrint );
  void Print( char szToPrint[] );
  ```

  For multiply dimensioned arrays, the second and all succeeding dimensions are considered part of the type. Therefore, they are used in distinguishing between overloaded functions:

  ```
  void Print( char szToPrint[] );
  void Print( char szToPrint[][7] );
  void Print( char szToPrint[][9][42] );
  ```

See Also
Reference
Overview of Overloading

Declaration Matching

Any two function declarations of the same name in the same scope can refer to the same function, or to two discrete functions that are overloaded. If the argument lists of the declarations contain arguments of equivalent types (as described in the previous section), the function declarations refer to the same function. Otherwise, they refer to two different functions that are selected using overloading.

Class scope is strictly observed; therefore, a function declared in a base class is not in the same scope as a function declared in a derived class. If a function in a derived class is declared with the same name as a function in the base class, the derived-class function hides the base-class function instead of causing overloading.

Block scope is strictly observed; therefore, a function declared in file scope is not in the same scope as a function declared locally. If a locally declared function has the same name as a function declared in file scope, the locally declared function hides the file-scoped function instead of causing overloading. For example:

```
// declaration_matching1.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
void func( int i )
{
    cout << "Called file-scoped func : " << i << endl;
}

void func( char *sz )
{
   cout << "Called locally declared func : " << sz << endl;
}

int main()
{
   // Declare func local to main.
   extern void func( char *sz );

   func( 3 );   // C2664 Error. func( int ) is hidden.
   func( "s" );
}
```

The preceding code shows two definitions from the function `func`. The definition that takes an argument of type `char *` is local to `main` because of the `extern` statement. Therefore, the definition that takes an argument of type `int` is hidden, and the first call to `func` is in error.

For overloaded member functions, different versions of the function can be given different access privileges. They are still considered to be in the scope of the enclosing class and thus are overloaded functions. Consider the following code, in which the member function `Deposit` is overloaded; one version is public, the other, private.

The intent of this sample is to provide an `Account` class in which a correct password is required to perform deposits. This is accomplished using overloading.

Note that the call to `Deposit` in `Account::Deposit` calls the private member function. This call is correct because `Account::Deposit` is a member function and therefore has access to the private members of the class.

```
// declaration_matching2.cpp
class Account
{
public:
   Account()
   {
   }
   double Deposit( double dAmount, char *szPassword );

private:
   double Deposit( double dAmount )
```

```
   {
      return 0.0;
   }
   int Validate( char *szPassword )
   {
      return 0;
   }

};

int main()
{
    // Allocate a new object of type Account.
    Account *pAcct = new Account;

    // Deposit $57.22. Error: calls a private function.
    // pAcct->Deposit( 57.22 );

    // Deposit $57.22 and supply a password. OK: calls a
    //  public function.
    pAcct->Deposit( 52.77, "pswd" );
}

double Account::Deposit( double dAmount, char *szPassword )
{
   if ( Validate( szPassword ) )
      return Deposit( dAmount );
   else
      return 0.0;
}
```

See Also

Reference

Overloading

Argument Matching

Overloaded functions are selected for the best match of function declarations in the current scope to the arguments supplied in the function call. If a suitable function is found, that function is called. "Suitable" in this context means one of the following:

- An exact match was found.
- A trivial conversion was performed.
- An integral promotion was performed.
- A standard conversion to the desired argument type exists.
- A user-defined conversion (either conversion operator or constructor) to the desired argument type exists.
- Arguments represented by an ellipsis were found.

The compiler creates a set of candidate functions for each argument. Candidate functions are functions in which the actual argument in that position can be converted to the type of the formal argument.

A set of "best matching functions" is built for each argument, and the selected function is the intersection of all the sets. If the intersection contains more than one function, the overloading is ambiguous and generates an error. The function that is eventually selected is always a better match than every other function in the group for at least one argument. If this is not the case (if there is no clear winner), the function call generates an error.

Consider the following declarations (the functions are marked `Variant 1`, `Variant 2`, and `Variant 3`, for identification in the following discussion):

```
Fraction &Add( Fraction &f, long l );       // Variant 1
Fraction &Add( long l, Fraction &f );       // Variant 2
Fraction &Add( Fraction &f, Fraction &f );  // Variant 3

Fraction F1, F2;
```

Consider the following statement:

```
F1 = Add( F2, 23 );
```

The preceding statement builds two sets:

Set 1: Candidate Functions That Have First Argument of Type Fraction	Set 2: Candidate Functions Whose Second Argument Can Be Converted to Type int
Variant 1	Variant 1 (**int** can be converted to **long** using a standard conversion)
Variant 3	

Functions in Set 2 are functions for which there are implicit conversions from actual parameter type to formal parameter type, and among such functions there is a function for which the "cost" of converting the actual parameter type to its formal parameter type is the smallest.

The intersection of these two sets is Variant 1. An example of an ambiguous function call is:

```
F1 = Add( 3, 6 );
```

The preceding function call builds the following sets:

Set 1: Candidate Functions That Have First Argument of Type int	Set 2: Candidate Functions That Have Second Argument of Type int

Variant 2 (**int** can be converted to **long** using a standard con version)	Variant 1 (**int** can be converted to **long** using a standard conver sion)

Note that the intersection between these two sets is empty. Therefore, the compiler generates an error message.

For argument matching, a function with *n* default arguments is treated as *n*+1 separate functions, each with a different number of arguments.

The ellipsis (...) acts as a wildcard; it matches any actual argument. This can lead to many ambiguous sets, if you do not design your overloaded function sets with extreme care.

Note
Ambiguity of overloaded functions cannot be determined until a function call is encountered. At that point, the sets are built f or each argument in the function call, and you can determine whether an unambiguous overload exists. This means that amb iguities can remain in your code until they are evoked by a particular function call.

See Also

Reference

Overloading

Argument Matching and the this Pointer

Class member functions are treated differently, depending on whether they are declared as **static**. Because nonstatic functions have an implicit argument that supplies the **this** pointer, nonstatic functions are considered to have one more argument than static functions; otherwise, they are declared identically.

These nonstatic member functions require that the implied **this** pointer match the object type through which the function is being called, or, for overloaded operators, they require that the first argument match the object on which the operator is being applied. (For more information about overloaded operators, see Overloaded Operators.)

Unlike other arguments in overloaded functions, no temporary objects are introduced and no conversions are attempted when trying to match the **this** pointer argument.

When the **– >** member-selection operator is used to access a member function, the **this** pointer argument has a type of *class-name* *** const**. If the members are declared as **const** or **volatile**, the types are **const** *class-name* *** const** and **volatile** *class-name* *** const**, respectively.

The **.** member-selection operator works exactly the same way, except that an implicit **&** (address-of) operator is prefixed to the object name. The following example shows how this works:

```
// Expression encountered in code
obj.name

// How the compiler treats it
(&obj)->name
```

The left operand of the **–>*** and **.*** (pointer to member) operators are treated the same way as the **.** and **–>** (member-selection) operators with respect to argument matching.

See Also
Reference
Argument Matching

Argument Matching and Conversions

When the compiler tries to match actual arguments against the arguments in function declarations, it can supply standard or user-defined conversions to obtain the correct type if no exact match can be found. The application of conversions is subject to these rules:

- Sequences of conversions that contain more than one user-defined conversion are not considered.
- Sequences of conversions that can be shortened by removing intermediate conversions are not considered.

The resultant sequence of conversions, if any, is called the best matching sequence. There are several ways to convert an object of type **int** to type **unsigned long** using standard conversions (described in Standard Conversions):

- Convert from **int** to **long** and then from **long** to **unsigned long**.
- Convert from **int** to **unsigned long**.

The first sequence, although it achieves the desired goal, is not the best matching sequence — a shorter sequence exists.

The following table shows a group of conversions, called trivial conversions, that have a limited effect on determining which sequence is the best matching. The instances in which trivial conversions affect choice of sequence are discussed in the list following the table.

Trivial Conversions

Convert from Type	Convert to Type
type-name	*type-name* **&**
type-name **&**	*type-name*
type-name **[ ]**	*type-name**
type-name **(** *argument-list* **)**	**(** **type-name* **) (** *argument-list* **)**
type-name	**const** *type-name*
type-name	**volatile** *type-name*
*type-name**	**const** *type-name**
*type-name**	**volatile** *type-name**

The sequence in which conversions are attempted is as follows:

1. Exact match. An exact match between the types with which the function is called and the types declared in the function prototype is always the best match. Sequences of trivial conversions are classified as exact matches. However, sequences that do not make any of these conversions are considered better than sequences that convert:
   - From pointer, to pointer to **const** (*type* ***** to **const** *type* *****).
   - From pointer, to pointer to **volatile** (*type* ***** to **volatile** *type* *****).
   - From reference, to reference to **const** (*type* **&** to **const** *type* **&**).
   - From reference, to reference to **volatile** (*type* **&** to **volatile** *type* **&**).
2. Match using promotions. Any sequence not classified as an exact match that contains only integral promotions, conversions from **float** to **double**, and trivial conversions is classified as a match using promotions. Although not as good a match as any exact match, a match using promotions is better than a match using standard conversions.
3. Match using standard conversions. Any sequence not classified as an exact match or a match using promotions that contains only standard conversions and trivial conversions is classified as a match using standard conversions. Within

this category, the following rules are applied:

- Conversion from a pointer to a derived class, to a pointer to a direct or indirect base class is preferable to converting to **void *** or **const void ***.
- Conversion from a pointer to a derived class, to a pointer to a base class produces a better match the closer the base class is to a direct base class. Suppose the class hierarchy is as shown in the following figure.

Graph Illustrating Preferred Conversions

Conversion from type `D*` to type `C*` is preferable to conversion from type `D*` to type `B*`. Similarly, conversion from type `D*` to type `B*` is preferable to conversion from type `D*` to type `A*`.

This same rule applies to reference conversions. Conversion from type `D&` to type `C&` is preferable to conversion from type `D&` to type `B&`, and so on.

This same rule applies to pointer-to-member conversions. Conversion from type `T D::*` to type `T C::*` is preferable to conversion from type `T D::*` to type `T B::*`, and so on (where `T` is the type of the member).

The preceding rule applies only along a given path of derivation. Consider the graph shown in the following figure.

Multiple-Inheritance Graph Illustrating Preferred Conversions

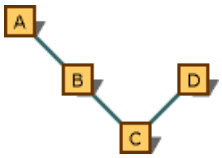

Conversion from type `C*` to type `B*` is preferable to conversion from type `C*` to type `A*`. The reason is that they are on the same path, and `B*` is closer. However, conversion from type `C*` to type `D*` is not preferable to conversion to type `A*`; there is no preference because the conversions follow different paths.

1. Match with user-defined conversions. This sequence cannot be classified as an exact match, a match using promotions, or a match using standard conversions. The sequence must contain only user-defined conversions, standard conversions, or trivial conversions to be classified as a match with user-defined conversions. A match with user-defined conversions is considered a better match than a match with an ellipsis but not as good a match as a match with standard conversions.
2. Match with an ellipsis. Any sequence that matches an ellipsis in the declaration is classified as a match with an ellipsis. This is considered the weakest match.

User-defined conversions are applied if no built-in promotion or conversion exists. These conversions are selected on the basis of the type of the argument being matched. Consider the following code:

```
// argument_matching1.cpp
class UDC
{
public:
   operator int()
   {
      return 0;
   }
   operator long();
};

void Print( int i )
{
};
```

```
UDC udc;

int main()
{
   Print( udc );
}
```

The available user-defined conversions for class `UDC` are from type **int** and type **long**. Therefore, the compiler considers conversions for the type of the object being matched: `UDC`. A conversion to **int** exists, and it is selected.

During the process of matching arguments, standard conversions can be applied to both the argument and the result of a user-defined conversion. Therefore, the following code works:

```
void LogToFile( long l );
...
UDC udc;
LogToFile( udc );
```

In the preceding example, the user-defined conversion, **operator long**, is invoked to convert `udc` to type **long**. If no user-defined conversion to type **long** had been defined, the conversion would have proceeded as follows: Type `UDC` would have been converted to type **int** using the user-defined conversion. Then the standard conversion from type **int** to type **long** would have been applied to match the argument in the declaration.

If any user-defined conversions are required to match an argument, the standard conversions are not used when evaluating the best match. This is true even if more than one candidate function requires a user-defined conversion; in such a case, the functions are considered equal. For example:

```
// argument_matching2.cpp
// C2668 expected
class UDC1
{
public:
   UDC1( int );  // User-defined conversion from int.
};

class UDC2
{
public:
   UDC2( long ); // User-defined conversion from long.
};

void Func( UDC1 );
void Func( UDC2 );

int main()
{
   Func( 1 );
}
```

Both versions of `Func` require a user-defined conversion to convert type **int** to the class type argument. The possible conversions are:

- Convert from type **int** to type `UDC1` (a user-defined conversion).
- Convert from type **int** to type **long**; then convert to type `UDC2` (a two-step conversion).

Even though the second of these requires a standard conversion, as well as the user-defined conversion, the two conversions are still considered equal.

Note
User-defined conversions are considered conversion by construction or conversion by initialization (conversion function). Both methods are considered equal when considering the best match.

See Also

Reference

Argument Matching

Address of Overloaded Functions

Use of a function name without arguments returns the address of that function. For example:

```
int Func( int i, int j );
int Func( long l );

...

int (*pFunc) ( int, int ) = Func;
```

In the preceding example, the first version of `Func` is selected, and its address is copied into `pFunc`.

The compiler determines which version of the function to select by finding a function with an argument list that exactly matches that of the target. The arguments in the overloaded function declarations are matched against one of the following:

- An object being initialized (as shown in the preceding example)
- The left side of an assignment statement
- A formal argument to a function
- A formal argument to a user-defined operator
- A function return type

If no exact match is found, the expression that takes the address of the function is ambiguous and an error is generated.

Note that although a nonmember function, `Func`, was used in the preceding example, the same rules are applied when taking the address of overloaded member functions.

See Also

Reference

Overloading

Operator Overloading

The **operator** keyword declares a function specifying what *operator-symbol* means when applied to instances of a class. This gives the operator more than one meaning, or "overloads" it. The compiler distinguishes between the different meanings of an operator by examining the types of its operands.

```
type operator operator-symbol ( parameter-list )
```

Remarks

You can redefine the function of most built-in operators globally or on a class-by-class basis. Overloaded operators are implemented as functions.

The name of an overloaded operator is **operator***x*, where *x* is the operator as it appears in the following table. For example, to overload the addition operator, you define a function called **operator+**. Similarly, to overload the addition/assignment operator, +=, define a function called **operator+=**.

Redefinable Operators

Operator	Name	Type
,	Comma	Binary
!	Logical NOT	Unary
!=	Inequality	Binary
%	Modulus	Binary
%=	Modulus assignment	Binary
&	Bitwise AND	Binary
&	Address-of	Unary
&&	Logical AND	Binary
&=	Bitwise AND assignment	Binary
( )	Function call	—
( )	Cast Operator	Unary
*****	Multiplication	Binary
*****	Pointer dereference	Unary
***=**	Multiplication assignment	Binary
+	Addition	Binary
+	Unary Plus	Unary
++	Increment1	Unary

+=	Addition assignment	Binary
–	Subtraction	Binary
–	Unary negation	Unary
––	Decrement1	Unary
–=	Subtraction assignment	Binary
–>	Member selection	Binary
–>*	Pointer-to-member selection	Binary
/	Division	Binary
/=	Division assignment	Binary
<	Less than	Binary
<<	Left shift	Binary
<<=	Left shift assignment	Binary
<=	Less than or equal to	Binary
=	Assignment	Binary
==	Equality	Binary
>	Greater than	Binary
>=	Greater than or equal to	Binary
>>	Right shift	Binary
>>=	Right shift assignment	Binary
[ ]	Array subscript	—
^	Exclusive OR	Binary
^=	Exclusive OR assignment	Binary
|	Bitwise inclusive OR	Binary
|=	Bitwise inclusive OR assignment	Binary
||	Logical OR	Binary
~	One's complement	Unary
delete	**Delete**	—

new	**New**	—
conversion operators	conversion operators	Unary

1 Two versions of the unary increment and decrement operators exist: preincrement and postincrement.

See General Rules for Operator Overloading for more information. The constraints on the various categories of overloaded operators are described in the following topics:

- Unary Operators
- Binary Operators
- Assignment
- Function Call
- Subscripting
- Class-Member Access
- Increment and Decrement.
- Conversion Functions

The operators shown in the following table cannot be overloaded. The table includes the preprocessor symbols **#** and **##**.

Nonredefinable Operators

Operator	**Name**
.	Member selection
.*	Pointer-to-member selection
::	Scope resolution
? :	Conditional
#	Preprocessor convert to string
##	Preprocessor concatenate

Although overloaded operators are usually called implicitly by the compiler when they are encountered in code, they can be invoked explicitly the same way as any member or nonmember function is called:

```
Point pt;
pt.operator+( 3 );  // Call addition operator to add 3 to pt.
```

Example

The following example overloads the **+** operator to add two complex numbers and returns the result.

```
// operator_overloading.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

struct Complex {
   Complex( double r, double i ) : re(r), im(i) {}
   Complex operator+( Complex &other );
   void Display( ) {   cout << re << ", " << im << endl; }
private:
   double re, im;
```

```
};

// Operator overloaded using a member function
Complex Complex::operator+( Complex &other ) {
   return Complex( re + other.re, im + other.im );
}

int main() {
   Complex a = Complex( 1.2, 3.4 );
   Complex b = Complex( 5.6, 7.8 );
   Complex c = Complex( 0.0, 0.0 );

   c = a + b;
   c.Display();
}
```

See Also

Reference

C++ Operators

C++ Keywords

General Rules for Operator Overloading

The following rules constrain how overloaded operators are implemented. However, they do not apply to the new and delete operators, which are covered separately.

- You cannot define new operators, such as **.
- You cannot redefine the meaning of operators when applied to built-in data types.
- Overloaded operators must either be a nonstatic class member function or a global function. A global function that needs access to private or protected class members must be declared as a friend of that class. A global function must take at least one argument that is of class or enumerated type or that is a reference to a class or enumerated type. For example:

  ```
  // rules_for_operator_overloading.cpp
  class Point
  {
  public:
      Point operator<( Point & );  // Declare a member operator
                                   //  overload.
      // Declare addition operators.
      friend Point operator+( Point&, int );
      friend Point operator+( int, Point& );
  };

  int main()
  {
  }
  ```

  The preceding code sample declares the less-than operator as a member function; however, the addition operators are declared as global functions that have friend access. Note that more than one implementation can be provided for a given operator. In the case of the preceding addition operator, the two implementations are provided to facilitate commutativity. It is just as likely that operators that add a `Point` to a `Point`, `int` to a `Point`, and so on, might be implemented.

- Operators obey the precedence, grouping, and number of operands dictated by their typical use with built-in types. Therefore, there is no way to express the concept "add 2 and 3 to an object of type `Point`," expecting 2 to be added to the *x* coordinate and 3 to be added to the *y* coordinate.
- Unary operators declared as member functions take no arguments; if declared as global functions, they take one argument.
- Binary operators declared as member functions take one argument; if declared as global functions, they take two arguments.
- If an operator can be used as either a unary or a binary operator (**&**, *****, **+**, and **-**), you can overload each use separately.
- Overloaded operators cannot have default arguments.
- All overloaded operators except assignment (**operator=**) are inherited by derived classes.
- The first argument for member-function overloaded operators is always of the class type of the object for which the operator is invoked (the class in which the operator is declared, or a class derived from that class). No conversions are supplied for the first argument.

Note that the meaning of any of the operators can be changed completely. That includes the meaning of the address-of (**&**), assignment (**=**), and function-call operators. Also, identities that can be relied upon for built-in types can be changed using operator overloading. For example, the following four statements are usually equivalent when completely evaluated:

```
var = var + 1;
var += 1;
var++;
++var;
```

This identity cannot be relied upon for class types that overload operators. Moreover, some of the requirements implicit in the use of these operators for basic types are relaxed for overloaded operators. For example, the addition/assignment operator, **+=**, requires the left operand to be an l-value when applied to basic types; there is no such requirement when the operator is overloaded.

Note
For consistency, it is often best to follow the model of the built-in types when defining overloaded operators. If the semantics of an overloaded operator differ significantly from its meaning in other contexts, it can be more confusing than useful.

See Also

Reference

Operator Overloading

C++ Unary Operators

The unary operators are shown in the following table.

Redefinable Unary Operators

Operator	Name
!	Logical NOT
&	Address-of
~	One's complement
*****	Pointer dereference
+	Unary plus
++	Increment
–	Unary negation
—	Decrement
conversion operators	conversion operators

Of the operators shown in preceding table, the postfix increment and decrement operators (**++** and **—**) are treated separately in Increment and Decrement.

Conversion operators are also discussed in a separate topic; see Conversion Functions.

To declare a unary operator function as a nonstatic member, you must declare it in the form:

ret-type **operator***op***()**

where *ret-type* is the return type and *op* is one of the operators listed in the preceding table.

To declare a unary operator function as a global function, you must declare it in the form:

ret-type **operator***op***(** *arg* **)**

where *ret-type* and *op* are as described for member operator functions and the *arg* is an argument of class type on which to operate.

Note
There is no restriction on the return types of the unary operators. For example, it makes sense for logical NOT (**!**) to return an integral value, but this is not enforced.

See Also

Reference

Operator Overloading

Increment and Decrement (C++)

The increment and decrement operators fall into a special category because there are two variants of each:

- Preincrement and postincrement
- Predecrement and postdecrement

When you write overloaded operator functions, it can be useful to implement separate versions for the prefix and postfix versions of these operators. To distinguish between the two, the following rule is observed: The prefix form of the operator is declared exactly the same way as any other unary operator; the postfix form accepts an additional argument of type **int**.

Note
When specifying an overloaded operator for the postfix form of the increment or decrement operator, the additional argument must be of type **int**; specifying any other type generates an error.

The following example shows how to define prefix and postfix increment and decrement operators for the `Point` class:

```
// increment_and_decrement1.cpp
class Point
{
public:
   // Declare prefix and postfix increment operators.
   Point& operator++();       // Prefix increment operator.
   Point operator++(int);     // Postfix increment operator.

   // Declare prefix and postfix decrement operators.
   Point& operator--();       // Prefix decrement operator.
   Point operator--(int);     // Postfix decrement operator.

   // Define default constructor.
   Point() { _x = _y = 0; }

   // Define accessor functions.
   int x() { return _x; }
   int y() { return _y; }
private:
   int _x, _y;
};

// Define prefix increment operator.
Point& Point::operator++()
{
   _x++;
   _y++;
   return *this;
}

// Define postfix increment operator.
Point Point::operator++(int)
{
   Point temp = *this;
   ++*this;
   return temp;
}

// Define prefix decrement operator.
Point& Point::operator--()
{
   _x--;
   _y--;
   return *this;
}
```

```
// Define postfix decrement operator.
Point Point::operator--(int)
{
   Point temp = *this;
   --*this;
   return temp;
}
int main()
{
}
```

The same operators can be defined in file scope (globally) using the following function heads:

```
friend Point& operator++( Point& )      // Prefix increment
friend Point& operator++( Point&, int ) // Postfix increment
friend Point& operator--( Point& )      // Prefix decrement
friend Point& operator--( Point&, int ) // Postfix decrement
```

The argument of type **int** that denotes the postfix form of the increment or decrement operator is not commonly used to pass arguments. It usually contains the value 0. However, it can be used as follows:

```
// increment_and_decrement2.cpp
class Int
{
public:
    Int &operator++( int n );
private:
    int _i;
};

Int& Int::operator++( int n )
{
    if( n != 0 )    // Handle case where an argument is passed.
        _i += n;
    else
        _i++;       // Handle case where no argument is passed.
    return *this;
}
int main()
{
   Int i;
   i.operator++( 25 ); // Increment by 25.
}
```

There is no syntax for using the increment or decrement operators to pass these values other than explicit invocation, as shown in the preceding code. A more straightforward way to implement this functionality is to overload the addition/assignment operator (**+=**).

See Also

Reference

Operator Overloading

Binary Operators

The following table shows a list of operators that can be overloaded.

Redefinable Binary Operators

Operator	Name
,	Comma
!=	Inequality
%	Modulus
%=	Modulus/assignment
&	Bitwise AND
&&	Logical AND
&=	Bitwise AND/assignment
*****	Multiplication
***=**	Multiplication/assignment
+	Addition
+=	Addition/assignment
–	Subtraction
–=	Subtraction/assignment
–>	Member selection
–>*	Pointer-to-member selection
/	Division
/=	Division/assignment
<	Less than
<<	Left shift
<<=	Left shift/assignment
<=	Less than or equal to
=	Assignment
==	Equality

>	Greater than
>=	Greater than or equal to
>>	Right shift
>>=	Right shift/assignment
^	Exclusive OR
^=	Exclusive OR/assignment
|	Bitwise inclusive OR
|=	Bitwise inclusive OR/assignment
||	Logical OR

To declare a binary operator function as a nonstatic member, you must declare it in the form:

ret-type **operator***op***(** *arg* **)**

where *ret-type* is the return type, *op* is one of the operators listed in the preceding table, and *arg* is an argument of any type.

To declare a binary operator function as a global function, you must declare it in the form:

ret-type **operator***op***(** *arg1***,** *arg2* **)**

where *ret-type* and *op* are as described for member operator functions and *arg1* and *arg2* are arguments. At least one of the arguments must be of class type.

Note
There is no restriction on the return types of the binary operators; however, most user-defined binary operators return either a class type or a reference to a class type.

See Also
Reference
Operator Overloading

Assignment

The assignment operator (**=**) is, strictly speaking, a binary operator. Its declaration is identical to any other binary operator, with the following exceptions:

- It must be a nonstatic member function. No **operator=** can be declared as a nonmember function.
- It is not inherited by derived classes.
- A default **operator=** function can be generated by the compiler for class types if none exists. (For more information about default **operator=** functions, see Memberwise Assignment and Initialization.)

The following example illustrates how to declare an assignment operator:

```
// assignment.cpp
class Point
{
public:
   Point &operator=( Point & );  // Right side is the argument.
   int _x, _y;
};

// Define assignment operator.
Point &Point::operator=( Point &ptRHS )
{
   _x = ptRHS._x;
   _y = ptRHS._y;

   return *this;  // Assignment operator returns left side.
}

int main()
{
}
```

Note that the supplied argument is the right side of the expression. The operator returns the object to preserve the behavior of the assignment operator, which returns the value of the left side after the assignment is complete. This allows writing statements such as:

```
pt1 = pt2 = pt3;
```

See Also

Reference

Operator Overloading

Function Call (C++)

The function-call operator, invoked using parentheses, is a binary operator.

```
primary-expression ( expression-list )
```

Remarks

In this context, *primary-expression* is the first operand, and *expression-list*, a possibly empty list of arguments, is the second operand. The function-call operator is used for operations that require a number of parameters. This works because *expression-list* is a list instead of a single operand. The function-call operator must be a nonstatic member function.

The function-call operator, when overloaded, does not modify how functions are called; rather, it modifies how the operator is to be interpreted when applied to objects of a given class type. For example, the following code would usually be meaningless:

```
Point pt;
pt( 3, 2 );
```

Given an appropriate overloaded function-call operator, however, this syntax can be used to offset the *x* coordinate 3 units and the *y* coordinate 2 units. The following code shows such a definition:

```
// function_call.cpp
class Point
{
public:
    Point() { _x = _y = 0; }
    Point &operator()( int dx, int dy )
        { _x += dx; _y += dy; return *this; }
private:
    int _x, _y;
};

int main()
{
   Point pt;
   pt( 3, 2 );
}
```

Note that the function-call operator is applied to the name of an object, not the name of a function.

See Also

Reference

Operator Overloading

Subscripting

The subscript operator (**[ ]**), like the function-call operator, is considered a binary operator. The subscript operator must be a nonstatic member function that takes a single argument. This argument can be of any type and designates the desired array subscript.

Example

The following example demonstrates how to create a vector of type **int** that implements bounds checking:

```
// subscripting.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class IntVector {
public:
   IntVector( int cElements );
   ~IntVector() { delete _iElements; }
   int& operator[]( int nSubscript );
private:
   int *_iElements;
   int _iUpperBound;
};

// Construct an IntVector.
IntVector::IntVector( int cElements ) {
   _iElements = new int[cElements];
   _iUpperBound = cElements;
}

// Subscript operator for IntVector.
int& IntVector::operator[]( int nSubscript ) {
   static int iErr = -1;

   if( nSubscript >= 0 && nSubscript < _iUpperBound )
      return _iElements[nSubscript];
   else {
      clog << "Array bounds violation." << endl;
      return iErr;
   }
}

// Test the IntVector class.
int main() {
   IntVector v( 10 );
   int i;

   for( i = 0; i <= 10; ++i )
      v[i] = i;

   v[3] = v[9];

   for ( i = 0; i <= 10; ++i )
      cout << "Element: [" << i << "] = " << v[i] << endl;
}
```

Output

```
Array bounds violation.
Element: [0] = 0
Element: [1] = 1
Element: [2] = 2
Element: [3] = 9
Element: [4] = 4
Element: [5] = 5
```

```
Element: [6] = 6
Element: [7] = 7
Element: [8] = 8
Element: [9] = 9
Array bounds violation.
Element: [10] = 10
```

Comments

When `i` reaches 10 in the preceding program, `operator[]` detects that an out-of-bounds subscript is being used and issues an error message.

Note that the function `operator[]` returns a reference type. This causes it to be an l-value, allowing you to use subscripted expressions on either side of assignment operators.

See Also

Reference

Operator Overloading

Member Access

Class member access can be controlled by overloading the member access operator (**->**). This operator is considered a unary operator in this usage, and the overloaded operator function must be a class member function. Therefore, the declaration for such a function is:

```
class-type *operator->()
```

Remarks

where *class-type* is the name of the class to which this operator belongs. The member access operator function must be a nonstatic member function.

This operator is used (often in conjunction with the pointer-dereference operator) to implement "smart pointers" that validate pointers prior to dereference or count usage.

The **.** member access operator cannot be overloaded.

See Also

Reference

Operator Overloading

Exception Handling in Visual C++

Robust code anticipates and handles exceptions. Exceptions occur when a program executes abnormally because of conditions outside the program's control. Certain operations, including object creation and file input/output, are subject to failures that go beyond errors. Out-of-memory conditions, for example, can occur even when your program is running correctly.

Abnormal situations should be handled by throwing and catching exceptions. Such situations are not the same as normal error conditions, such as a function executing correctly, but returning a result code indicating an error. A normal error condition, for example, would be a file status function indicating that a file does not exist. For normal error conditions, the program should examine the error code and respond appropriately.

Abnormal situations are also not the same as erroneous execution, in which, for example, the caller makes a mistake in passing arguments to a function or calls it in an inappropriate context. For erroneous execution, test your inputs and other assumptions with an assertion (see Using Assertions).

Visual C++ supports three kinds of exception handling:

- C++ exception handling

  Although structured exception handling works with C and C++ source files, it is not specifically designed for C++. For C++ programs, you should use C++ exception handling.

- Structured exception handling

  Windows supplies its own exception mechanism, called SEH. It is not recommended for C++ or MFC programming. Use SEH only in non-MFC C programs.

- MFC exceptions

  Since version 3.0, MFC has used C++ exceptions but still supports its older exception handling macros, which are similar to C++ exceptions in form. The older MFC exception handling macros have been supported since version 1.0. Although these macros are not recommended for new programming, they are still supported for backward compatibility. In programs that already use the macros, you can freely use C++ exceptions as well. During preprocessing, the macros evaluate to the exception handling keywords defined in the Visual C++ implementation of the C++ language as of Visual C++ version 2.0. You can leave existing exception macros in place while you begin to use C++ exceptions.

Do not mix the error handling mechanisms; for example, do not use C++ exceptions with SEH. For advice about mixing MFC macros and C++ exceptions, see Exceptions: Using MFC Macros and C++ Exceptions.

For information on handling exceptions in CLR applications, see Exception Handling under /clr.

For information about exception handling on x64 processors, see Exception Handling (x64).

See Also

Other Resources

C++ Language Reference

C++ Exception Handling

Exceptions are run-time anomalies, such as division by zero, that require immediate handling when encountered by your program. The C++ language provides built-in support for raising and handling exceptions. With C++ exception handling, your program can communicate unexpected events to a higher execution context that is better able to recover from such abnormal events. These exceptions are handled by code that is outside the normal flow of control.

This discussion on C++ exception handling includes:

- The try, catch, and throw Statements
- Exception Specifications
- Function-try Blocks
- Exception Handling: Default Synchronous Exception Model
- Type-Safe Exception Handling
- C++ Exception Examples
- Catchable Types
- Unhandled C++ Exceptions
- Order of Handlers
- Mixing C (Structured) and C++ Exceptions
- Exception Handling Overhead

Note
In these articles, the terms "structured exception handling" and "structured exception" (or "C exception") refer exclusively to the Win32 structured exception handling mechanism provided by Windows. All other references to exception handling (or "C++ exception") refer to the C++ exception handling mechanism.

Note
MFC now uses C++ exception handling. The older MFC exception macros, if you still use them, evaluate to C++ exception keywords. See Exceptions: Changes to Exception Macros in Version 3.0.

Unlike the Win32 structured exception handling mechanism, the language itself provides support for C++ exception handling. These articles describe the Microsoft implementation of C++ exception handling, which is based on the ANSI C++ Standard.

For C++ programs, you should use C++ exception handling rather than structured exception handling. While structured exception handling works in C++ programs, you can ensure that your code is more portable by using C++ exception handling. The C++ exception handling mechanism is more flexible, in that it can handle exceptions of any type. C exceptions are always of type **unsigned int**.

Using C++ Exceptions

In C++, the process of raising an exception is called "throwing" an exception. A designated exception handler then "catches" the thrown exception.

C++ exception handling is thread safe. Throwing on one thread (including rethrow) has no impact on any throw/catch in progress on another thread.

To enable C++ exception handling in your code, use /EHsc.

Note

As of version 4.0, MFC uses the C++ exception handling mechanism. Although you are encouraged to use C++ exception handling in new code, MFC version 4.0 and later retains the macros from previous versions of MFC so that old code will not be broken. The macros and the new mechanism can be combined as well. For information on mixing macros and C++ exception handling and on converting old code to use the new mechanism, see the articles
Exceptions: Using MFC Macros and C++ Exceptions and Exceptions: Converting from MFC Exception Macros.

See Also
Reference
Exception Handling in Visual C++

The try, catch, and throw Statements

The following syntax shows a try block and its handlers:

```
try {
   // code that could throw an exception
}
[ catch (exception-declaration) {
   // code that executes when exception-declaration is thrown
   // in the try block
}
[catch (exception-declaration) {
   // code that handles another exception type
} ] . . . ]
// The following syntax shows a throw expression:
throw [expression]
```

Remarks

The C++ language provides built-in support for handling anomalous situations, known as exceptions, which may occur during the execution of your program. The **try**, **throw**, and **catch** statements implement exception handling. With C++ exception handling, your program can communicate unexpected events to a higher execution context that is better able to recover from such abnormal events. These exceptions are handled by code which is outside the normal flow of control. The Microsoft C++ compiler implements the C++ exception handling model based on the ANSI C++ standard.

C++ also provides a way to explicitly specify whether or a function could throw exceptions. Exception specifications are used in function declarations to indicate that a function could throw an exception. For example, an exception specification **throw(...)** tells the compiler that a function could throw an exception, but doesn't specify the type. For example:

```
void MyFunc()  throw(...) {
   throw 1;
}
```

For more information, see Exception Specifications.

Note

The Win32 structured exception-handling mechanism works with both C and C++ source files. However, it is not specifically designed for C++. You can ensure that your code is more portable by using C++ exception handling. Also, C++ exception handling is more flexible, in that it can handle exceptions of any type. For C++ programs, it is recommended that you use the C++ exception-handling mechanism (**try**, **catch**, **throw**) described in this topic.

The code after the **try** clause is the guarded section of code. The throw expression throws (raises) an exception. The code block after the **catch** clause is the exception handler, and catches (handles) the exception thrown by the throw expression. The *exception-declaration* statement indicates the type of exception the clause handles. The type can be any valid data type, including a C++ class. If the *exception-declaration* statement is an ellipsis (**...**), the **catch** clause handles any type of exception, including C exceptions and system- or application-generated exceptions such as memory protection, divide by zero, and floating-point violations. Such a handler must be the last handler for its **try** block.

The operand of **throw** is syntactically similar to the operand of a **return** statement.

Execution proceeds as follows:

1. Control reaches the **try** statement by normal sequential execution. The guarded section within the **try** block is executed.
2. If no exception is thrown during execution of the guarded section, the **catch** clauses that follow the **try** block are not executed. Execution continues at the statement after the last **catch** clause following the **try** block in which the exception was thrown.
3. If an exception is thrown during execution of the guarded section or in any routine the guarded section calls (either directly or indirectly), an exception object is created from the object created by the **throw** operand. (This implies that a

copy constructor may be involved.) At this point, the compiler looks for a **catch** clause in a higher execution context that can handle an exception of the type thrown (or a **catch** handler that can handle any type of exception). The **catch** handlers are examined in order of their appearance following the **try** block. If no appropriate handler is found, the next dynamically enclosing **try** block is examined. This process continues until the outermost enclosing **try** block is examined.

4. If a matching handler is still not found, or if an exception occurs while unwinding, but before the handler gets control, the predefined run-time function **terminate** is called. If an exception occurs after throwing the exception, but before the unwind begins, **terminate** is called.

5. If a matching **catch** handler is found, and it catches by value, its formal parameter is initialized by copying the exception object. If it catches by reference, the parameter is initialized to refer to the exception object. After the formal parameter is initialized, the process of unwinding the stack begins. This involves the destruction of all automatic objects that were constructed (but not yet destructed) between the beginning of the **try** block associated with the **catch** handler and the exception's throw site. Destruction occurs in reverse order of construction. The **catch** handler is executed and the program resumes execution following the last handler (that is, the first statement or construct which is not a **catch** handler). Control can only enter a **catch** handler through a thrown exception, never via a **goto** statement or a **case** label in a **switch** statement.

The following is a simple example of a **try** block and its associated **catch** handler. This example detects failure of a memory allocation operation using the **new** operator. If **new** is successful, the **catch** handler is never executed:

```
// exceptions_trycatchandthrowstatements.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main() {
   char *buf;
   try {
      buf = new char[512];
      if( buf == 0 )
         throw "Memory allocation failure!";
   }
   catch( char * str ) {
      cout << "Exception raised: " << str << '\n';
   }
}
```

The operand of the **throw** expression specifies that an exception of type `char *` is being thrown. It is handled by a **catch** handler that expresses the ability to catch an exception of type `char *`. In the event of a memory allocation failure, this is the output from the preceding example:

```
Exception raised: Memory allocation failure!
```

The real power of C++ exception handling lies not only in its ability to deal with exceptions of varying types, but also in its ability to automatically call destructor functions during stack unwinding, for all local objects constructed before the exception was thrown.

The following example demonstrates C++ exception handling using classes with destructor semantics:

Example

```
// exceptions_trycatchandthrowstatements2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
void MyFunc( void );

class CTest {
public:
   CTest() {};
   ~CTest() {};
   const char *ShowReason() const {
      return "Exception in CTest class.";
   }
};
```

```
class CDtorDemo {
public:
   CDtorDemo();
   ~CDtorDemo();
};

CDtorDemo::CDtorDemo() {
   cout << "Constructing CDtorDemo.\n";
}

CDtorDemo::~CDtorDemo() {
   cout << "Destructing CDtorDemo.\n";
}

void MyFunc() {
   CDtorDemo D;
   cout<< "In MyFunc(). Throwing CTest exception.\n";
   throw CTest();
}

int main() {
   cout << "In main.\n";
   try {
       cout << "In try block, calling MyFunc().\n";
       MyFunc();
   }
   catch( CTest E ) {
       cout << "In catch handler.\n";
       cout << "Caught CTest exception type: ";
       cout << E.ShowReason() << "\n";
   }
   catch( char *str )    {
       cout << "Caught some other exception: " << str << "\n";
   }
   cout << "Back in main. Execution resumes here.\n";
}
```

Output

```
In main.
In try block, calling MyFunc().
Constructing CDtorDemo.
In MyFunc(). Throwing CTest exception.
Destructing CDtorDemo.
In catch handler.
Caught CTest exception type: Exception in CTest class.
Back in main. Execution resumes here.
```

Comments

Note that in this example, the exception parameter (the argument to the **catch** clause) is declared in both **catch** handlers:

```
catch( CTest E )
// ...
catch( char *str )
// ...
```

You do not need to declare this parameter; in many cases it may be sufficient to notify the handler that a particular type of exception has occurred. However, if you do not declare an exception object in the exception-declaration, you will not have access to that object in the **catch** handler clause.

A *throw-expression* with no operand re-throws the exception currently being handled. Such an expression should appear only in a **catch** handler or in a function called from within a **catch** handler. The re-thrown exception object is the original exception object (not a copy). For example:

```
try {
```

```
      throw CSomeOtherException();
   }
   catch(...) {  // Handle all exceptions
      // Respond (perhaps only partially) to exception
      throw;       // Pass exception to some other handler
   }
```

See Also

Reference

C++ Exception Handling

C++ Keywords

Function-try Blocks

Unhandled C++ Exceptions

Exception Specifications

Exception specifications are used to provide summary information about what exceptions can be thrown out of a function. For example:

```
void MyFunction(int i) throw(...);
```

An exception specification with an empty throw, as in

```
void MyFunction(int i) throw();
```

tells the compiler that the function does not throw any exceptions. It is the equivalent to using __declspec(nothrow). For example:

```
// exceptions_trycatchandthrowstatements3.cpp
// compile with: /EHsc
#include <stdio.h>

void empty() throw() {
   puts("In empty()");
}

void with_type() throw(int) {
   puts("Will throw an int");
   throw(1);
}

int main() {
   try {
      empty();
      with_type();
   }
   catch (int) {
      puts("Caught an int");
   }
}
```

Visual C++ departs from the ANSI Standard in its implementation of exception specifications. The following table summarizes the Visual C++ implementation of exception specifications:

Exception specification	Meaning
throw()	The function does not throw an exception. However, if an exception is thrown out of a function marked throw(), the Visual C++ compiler will not call unexpected (see unexpected (CRT) and unexpected (<exception>) for more information). If a function is marked with throw(), the Visual C++ compiler will assume that the function does not throw C++ exceptions and generated code accordingly. Due to code optimizations that maybe performed by the C++ compiler (based on the assumption that the function does not throw any C++ exceptions) if function does throw an exception, the program may not execute correctly.
throw(...)	The function can throw an exception.

throw(*type*)	The function can throw an exception of type *type*. However, in Visual C++ .NET, this is interpreted as throw(...). See Function Exception Specifiers.

If exception handling is used in an application, there must be one or more functions that handle thrown exceptions. Any functions called between the one that throws an exception and the one that handles the exception must be capable of throwing the exception.

The throw behavior of a function depends on the following factors:

- Whether you are compiling the function under C or C++.
- Which /EH compiler option you use.
- Whether you explicitly specify the exception specification.

Explicit exception specifications are not allowed on C functions.

The following table summarizes the throw behavior of a function:

Function	**/EHsc**	**/EHs**	**/EHa**	**/EHac**
C function	throw()	throw(...)	throw(...)	throw(...)
C++ function with no exception specification	throw(...)	throw(...)	throw(...)	throw(...)
C++ function with throw() exception specification	throw()	throw()	throw(...)	throw(...)
C++ function with throw(...) exception specification	throw(...)	throw(...)	throw(...)	throw(...)
C++ function with throw(*type*) exception specification	throw(...)	throw(...)	throw(...)	throw(...)

Example

```
// exception_specification.cpp
// compile with: /EHs
#include <stdio.h>

void handler() {
   printf_s("in handler\n");
}

void f1(void) throw(int) {
   printf_s("About to throw 1\n");
   if (1)
      throw 1;
}

void f5(void) throw() {
   try {
      f1();
   }
   catch(...) {
      handler();
    }
}

// invalid, doesn't handle the int exception thrown from f1()
// void f3(void) throw() {
//   f1();
// }

void __declspec(nothrow) f2(void) {
   try {
      f1();
   }
```

```
   catch(int) {
      handler();
    }
}

// only valid if compiled without /EHc
// /EHc means assume extern "C" functions don't throw exceptions
extern "C" void f4(void);
void f4(void) {
   f1();
}

int main() {
   f2();

   try {
      f4();
   }
   catch(...) {
      printf_s("Caught exception from f4\n");
   }
   f5();
}
```

Output

```
About to throw 1
in handler
About to throw 1
Caught exception from f4
About to throw 1
in handler
```

See Also

Reference

The try, catch, and throw Statements

C++ Exception Handling

Function-try Blocks

A function-try block catches exceptions generated from a call made in a constructor's initialization list.

Example

The following sample uses a function-try block:

```
// exceptions_Function_tryBlocks.cpp
// compile with: /EHsc
#include "stdio.h"

int f(int i) {
   throw "test";
   return 0;
}

class C {
   int i;

public:
   C(int);
};

C::C(int ii) {
   try {   // function-try block
      f(ii) ;
      // body of function goes in try block
   }
   catch (...) {
      // handles exceptions thrown from the constructor-initializer
      // and from the constructor function body
      printf_s("In the catch block\n");
   }
}

int main() {
   C *MyC = new C(0);
   delete MyC;
}
```

Output

```
In the catch block
```

See Also

Reference

C++ Exception Handling

The try, catch, and throw Statements

Exception Handling: Default Synchronous Exception Model

In previous versions of Visual C++, the C++ exception handling mechanism supported asynchronous (hardware) exceptions by default. Under the asynchronous model, the compiler assumes any instruction may generate an exception.

With the new synchronous exception model, now the default, exceptions can be thrown only with a **throw** statement. Therefore, the compiler can assume that exceptions happen only at a **throw** statement or at a function call. This model allows the compiler to eliminate the mechanics of tracking the lifetime of certain unwindable objects, and to significantly reduce the code size, if the objects' lifetimes do not overlap a function call or a **throw** statement. The two exception handling models, synchronous and asynchronous, are fully compatible and can be mixed in the same application.

See /EH for information on enabling exception handling.

See Also

Reference

C++ Exception Handling

Type-Safe Exception Handling

C++ exception handling supports type-safe exception handlers. C exceptions are always identified by an **unsigned int**. With C++ exception handling, you can specify that exceptions of a particular type (including C++ objects) are caught by a handler that matches the type of the exception being thrown.

See Also

Reference

C++ Exception Handling

C++ Exception Examples

The real power of C++ exception handling lies not only in its ability to deal with exceptions of varying types, but also in its ability to automatically call destructor functions during stack unwinding for all local objects constructed before the exception was thrown.

The context which exists between the **throw** site and the **catch** handler is referred to as the "exception stack frame." This frame may contain objects with destructor semantics. If an exception is thrown during execution of the guarded section or in any routine the guarded section calls (directly or indirectly), an exception object is created from the object created by the **throw** operand. (This implies that a copy constructor may be involved.) At this point, the compiler looks for a **catch** clause in a higher execution context that can handle an exception of the type thrown, or a **catch** handler that can handle any type of exception. The **catch** handlers are examined in order of their appearance following the **try** block. If no appropriate handler is found, the next dynamically enclosing **try** block is examined. This process continues until the outermost enclosing **try** block is examined.

If a matching handler is still not found, or if an exception occurs while unwinding but before the handler gets control, the predefined run-time function **terminate** is called. If an exception occurs after throwing the exception but before the unwind begins, the **terminate** function is called. You can install a custom termination function to handle such situations. See Unhandled C++ Exceptions for more information.

The following example demonstrates C++ exception handling using classes with destructor semantics. It declares two C++ classes; one (class `CTest`) for defining the exception object itself, and the second (class `CDtorDemo`) for demonstrating the destruction of a separate frame object during stack unwinding:

Example

```
// exceptions_Exception_Examples.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
void MyFunc( void );

class CTest
{
public:
    CTest(){};
    ~CTest(){};
    const char *ShowReason() const { return "Exception in CTest class."; }

};

class CDtorDemo
{
public:
    CDtorDemo();
    ~CDtorDemo();
};

CDtorDemo::CDtorDemo()
{
    cout << "Constructing CDtorDemo." << endl;
}

CDtorDemo::~CDtorDemo()
{
    cout << "Destructing CDtorDemo." << endl;
}

void MyFunc()
{

    CDtorDemo D;
    cout<< "In MyFunc(). Throwing CTest exception." << endl;
    throw CTest();
}
```

```
int main()
{
    cout << "In main." << endl;
    try
    {
        cout << "In try block, calling MyFunc()." << endl;
        MyFunc();
    }
    catch( CTest E )
    {
        cout << "In catch handler." << endl;
        cout << "Caught CTest exception type: ";
        cout << E.ShowReason() << endl;
    }
    catch( char *str )
    {
        cout << "Caught some other exception: " << str << endl;
    }
    cout << "Back in main. Execution resumes here." << endl;
    return 0;
}
```

If a matching **catch** handler is found, and it catches by value, its formal parameter is initialized by copying the exception object. If it catches by reference, the parameter is initialized to refer to the exception object. After the formal parameter is initialized, the process of "unwinding the stack" begins. This involves the destruction of all automatic objects that were constructed (but not yet destructed) between the beginning of the **try** block associated with the **catch** handler and the exception's throw site. Destruction occurs in reverse order of construction. The **catch** handler is executed and the program resumes execution following the last handler (that is, the first statement or construct that is not a **catch** handler).

Output

```
In main.
In try block, calling MyFunc().
Constructing CDtorDemo.
In MyFunc(). Throwing CTest exception.
Destructing CDtorDemo.
In catch handler.
Caught CTest exception type: Exception in CTest class.
Back in main. Execution resumes here.
```

Note the declaration of the exception parameter in both **catch** handlers:

```
catch( CTest E )
{ // ... }
catch( char *str )
{ // ... }
```

You do not need to declare this parameter; in many cases it may be sufficient to notify the handler that a particular type of exception has occurred. However, if you do not declare an exception object in the exception declaration, you will not have access to the object in the **catch** handler clause. For example:

```
catch( CTest )
{
    // No access to a CTest exception object in this handler.
}
```

A **throw** expression with no operand re-throws the exception currently being handled. Such an expression should appear only in a **catch** handler or in a function called from within a **catch** handler. The re-thrown exception object is the original exception object (not a copy). For example:

```
try
{
    throw CSomeOtherException();
}
catch(...)      // Handle all exceptions
{
    // Respond (perhaps only partially) to exception
    //...
```

```
    throw;      // Pass exception to some other handler
}
```

See Also

Reference

C++ Exception Handling

Catchable Types

Because C++ enables you to throw exceptions of any type, you need to determine which **catch** handlers can catch an exception of a specific class type. A C++ exception can be caught by a **catch** handler that specifies the same type as the thrown exception, or by a handler that can catch any type of exception. An exception can also be caught by a **catch** handler that uses a reference to the same type as the thrown exception.

If the type of thrown exception is a class, which also has a base class (or classes), it can be caught by handlers that accept base classes of the exception's type, as well as references to bases of the exception's type. Note that when an exception is caught by a reference, it is bound to the actual thrown exception object; otherwise, it is a copy (much the same as an argument to a function).

When an exception is thrown, it may be caught by the following types of **catch** handlers:

- A handler that can accept any type (using the ellipsis syntax).
- A handler that accepts the same type as the exception object; because it is a copy, **const** and **volatile** modifiers are ignored.
- A handler that accepts a reference to the same type as the exception object.
- A handler that accepts a reference to a **const** or **volatile** form of the same type as the exception object.
- A handler that accepts a base class of the same type as the exception object; since it is a copy, **const** and **volatile** modifiers are ignored. The **catch** handler for a base class must not precede the **catch** handler for the derived class.
- A handler that accepts a reference to a base class of the same type as the exception object.
- A handler that accepts a reference to a **const** or **volatile** form of a base class of the same type as the exception object.
- A handler that accepts a pointer to which a thrown pointer object can be converted via standard pointer conversion rules.

See Also

Reference

C++ Exception Handling

Unhandled C++ Exceptions

If a matching handler (or ellipsis **catch** handler) cannot be found for the current exception, the predefined `terminate` run-time function is called. (You can also explicitly call `terminate` in any of your handlers.) The default action of `terminate` is to call `abort`. If you want `terminate` to call some other function in your program before exiting the application, call the `set_terminate` function with the name of the function to be called as its single argument. You can call `set_terminate` at any point in your program. The `terminate` routine always calls the last function given as an argument to `set_terminate`.

Example

The following example throws a `char *` exception, but does not contain a handler designated to catch exceptions of type `char *`. The call to `set_terminate` instructs `terminate` to call `term_func`.

```
// exceptions_Unhandled_Exceptions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
void term_func() {
   cout << "term_func was called by terminate." << endl;
   exit( -1 );
}
int main() {
   try
   {
      set_terminate( term_func );
      throw "Out of memory!"; // No catch handler for this exception
   }
   catch( int )
   {
      cout << "Integer exception raised." << endl;
   }
   return 0;
}
```

Output

```
term_func was called by terminate.
```

The `term_func` function should terminate the program or current thread, ideally by calling `exit`. If it doesn't, and instead returns to its caller, `abort` is called.

See Also

Reference

C++ Exception Handling

Order of Handlers

The order in which **catch** handlers appear is significant, because handlers for a given **try** block are examined in order of their appearance. For example, it is an error to place the handler for a base class before the handler for a derived class. After a matching **catch** handler is found, subsequent handlers are not examined. As a result, an ellipsis **catch** handler must be the last handler for its **try** block. For example:

```
// ...
try
{
    // ...
}
catch( ... )
{
    // Handle exception here.
}
// Error: the next two handlers are never examined.
catch( const char * str )
{
    cout << "Caught exception: " << str << endl;
}
catch( CExcptClass E )
{
    // Handle CExcptClass exception here.
}
```

In this example, the ellipsis **catch** handler is the only handler that is examined.

See Also

Reference

C++ Exception Handling

Mixing C (Structured) and C++ Exceptions

If you want to write more portable code, using structured exception handling in a C++ program is not recommended. However, you may sometimes want to compile with **/EHa** and mix structured exceptions and C++ source code, and need some facility for handling both kinds of exceptions. Because a structured exception handler has no concept of objects or typed exceptions, it cannot handle exceptions thrown by C++ code; however, C++ **catch** handlers can handle structured exceptions. As such, C++ exception handling syntax (**try**, **throw**, **catch**) is not accepted by the C compiler, but structured exception handling syntax (**__try**, **__except**, **__finally**) is supported by the C++ compiler.

See _set_se_translator for information on handling structured exceptions as C++ exceptions.

If you mix structured and C++ exceptions, note the following:

1. C++ exceptions and structured exceptions cannot be mixed within the same function.
2. Termination handlers (**__finally** blocks) are always executed, even during unwinding after an exception is thrown.
3. C++ exception handling can catch and preserve unwind semantics in all modules compiled with the /EH compiler option (this option enables unwind semantics).
4. There may be some situations in which destructor functions are not called for all objects. For example, if a structured exception occurs while attempting to make a function call through an uninitialized function pointer, and that function takes as parameters objects that were constructed before the call, those objects will not have their destructors called during stack unwind.

What do you want to know more about?

- Using setjmp or longjmp in C++ programs
- Differences between SEH and C++ EH

See Also
Reference
C++ Exception Handling

Using setjmp/longjmp

Do not use setjmp and longjmp in C++ programs; these functions do not support C++ object semantics. Also, using these functions in C++ programs may degrade performance by preventing optimization on local variables. Use the C++ exception handling **try/catch** constructs instead.

If you must use **setjmp**/**longjmp** in a C++ program, the interaction between these functions and C++ exception handling requires that you include SETJMP.H or SETJMPEX.H. Destructors for local objects will be called during the stack unwind if you compile with /EH. If you compile with /EHs and one of your functions call a function that uses nothrow, and if the function that uses **nothrow** calls **longjmp**, the destructor unwind may not occur, depending on the optimizer.

Also, if you intend your code to be portable, do not rely on correct destruction of frame-based objects when executing a nonlocal goto using a call to **longjmp**.

See Also

Reference

Mixing C (Structured) and C++ Exceptions

Exception Handling Differences

The major difference between structured exception handling and C++ exception handling is that the C++ exception handling model deals in types, while the C structured exception handling model deals with exceptions of one type — specifically, **unsigned int**. That is, C exceptions are identified by an unsigned integer value, whereas C++ exceptions are identified by data type. When an exception is raised in C, each possible handler executes a filter that examines the C exception context and determines whether to accept the exception, pass it to some other handler, or ignore it. When an exception is thrown in C++, it may be of any type.

A second difference is that the C structured exception handling model is referred to as "asynchronous" in that exceptions occur secondary to the normal flow of control. The C++ exception handling mechanism is fully "synchronous," which means that exceptions occur only when they are thrown.

If a C exception is raised in a C++ program, it can be handled by a structured exception handler with its associated filter or by a C++ **catch** handler, whichever is dynamically nearer to the exception context. For example, the following C++ program raises a C exception inside a C++ **try** context:

Example

```
// exceptions_Exception_Handling_Differences.cpp
// compile with: /EHa
#include <iostream>

using namespace std;
void SEHFunc( void );

int main() {
   try {
      SEHFunc();
   }
   catch( ... ) {
      cout << "Caught a C exception."<< endl;
   }
}

void SEHFunc() {
   __try {
      int x, y = 0;
      x = 5 / y;
   }
   __finally {
      cout << "In finally." << endl;
   }
}
```

Output

```
In finally.
Caught a C exception.
```

For example, the following code installs a custom translation function, and then raises a C exception that is wrapped by the `SE_Exception` class:

```
// exceptions_Exception_Handling_Differences3.cpp
// compile with: /EHa
#include <stdio.h>
#include <eh.h>
#include <windows.h>

class SE_Exception {
private:
   SE_Exception() {}
   unsigned int nSE;

public:
```

```
   SE_Exception( SE_Exception& e) : nSE(e.nSE) {}
   SE_Exception(unsigned int n) : nSE(n) {}
   ~SE_Exception() {}
   unsigned int getSeNumber() { return nSE; }
};

void SEFunc() {
   __try {
      int x, y = 0;
      x = 5 / y;
    }
    __finally {
      printf_s( "In finally\n" );
   }
}

void trans_func( unsigned int u, _EXCEPTION_POINTERS* pExp ) {
   printf_s( "In trans_func.\n" );
   throw SE_Exception( u );
}

int main() {
   _set_se_translator( trans_func );
    try {
      SEFunc();
    }
    catch( SE_Exception e ) {
      printf_s( "Caught a __try exception with SE_Exception.\n" );
      printf_s( "nSE = 0x%x\n", e.getSeNumber() );
    }
}
```

Output

```
In trans_func.
In finally
Caught a __try exception with SE_Exception.
nSE = 0xc0000094
```

C Exception Wrapper Class

In a simple example like the above, the C exception can be caught only by an ellipsis (**...**) **catch** handler. No information about the type or nature of the exception is communicated to the handler. While this method works, in some cases you may need to define a transformation between the two exception handling models so that each C exception is associated with a specific class. To do this, you can define a C exception "wrapper" class, which can be used or derived from in order to attribute a specific class type to a C exception. By doing so, each C exception can be handled by a C++ **catch** handler more separately than in the preceding example.

Your wrapper class might have an interface consisting of some member functions that determine the value of the exception, and that access the extended exception context information provided by the C exception model. You might also want to define a default constructor and a constructor that accepts an **unsigned int** argument (to provide for the underlying C exception representation), and a bitwise copy constructor. The following is a possible implementation of a C exception wrapper class:

```
// exceptions_Exception_Handling_Differences2.cpp
// compile with: /c
class SE_Exception {
private:
   SE_Exception() {}
   SE_Exception( SE_Exception& ) {}
   unsigned int nSE;
public:
   SE_Exception( unsigned int n ) : nSE( n ) {}
   ~SE_Exception() {}
   unsigned int getSeNumber() {
      return nSE;
   }
};
```

To use this class, you install a custom C exception translation function that is called by the internal exception handling mechanism each time a C exception is thrown. Within your translation function, you can throw any typed exception (perhaps an `SE_Exception` type, or a class type derived from `SE_Exception`) that can be caught by an appropriate matching C++ **catch** handler. The translation function can simply return, which indicates that it did not handle the exception. If the translation function itself raises a C exception, terminate is called.

To specify a custom translation function, call the _set_se_translator function with the name of your translation function as its single argument. The translation function that you write is called once for each function invocation on the stack that has **try** blocks. There is no default translation function; if you do not specify one by calling **_set_se_translator**, the C exception can only be caught by an ellipsis **catch** handler.

See Also

Reference

Mixing C (Structured) and C++ Exceptions

Exception Handling Overhead

The extra overhead associated with the C++ exception handling mechanism may increase the size of executable files and slow program execution time. The /EHsc compiler option enables C++ exception handling and unwind semantics. If you are not using C++ exception handling in your program and you want to eliminate the associated overhead, you can use the /EHsc- compiler option to turn off exception handling and unwind semantics. Note that /EHsc- is the default.

Because of the nature of exception handling and the extra overhead involved, exceptions should be used only to signal the occurrence of unusual or unanticipated program events. Exception handlers should not be used to redirect the program's normal flow of control. For example, an exception should not be thrown in cases of potential logic or user input errors, such as the overflow of an array boundary. In these cases, simply returning an error code may be simpler and more concise. Judicious use of exception handling constructs makes your program easier to maintain and your code more readable.

For more information about C++ exception handling, see the *Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup.

See Also

Reference

C++ Exception Handling

Structured Exception Handling (C++)

Windows 95, Windows 98, and Windows 2000 (formerly Windows NT) support a robust approach to handling exceptions, called structured exception handling, which involves cooperation of the operating system but also has direct support in the programming language.

Microsoft Specific
Grammar

try-except-statement :

__try *compound-statement*

__except (*expression*) *compound-statement*

Remarks

An exception is an event that is unexpected or disrupts the ability of the process to proceed normally. Exceptions can be detected by both hardware and software. Hardware exceptions include dividing by zero and overflow of a numeric type. Software exceptions include those you detect and signal to the system by calling the **RaiseException** function and special situations detected by Windows.

You can write more reliable code with structured exception handling. You can ensure that resources, such as memory blocks and files, are properly closed in the event of unexpected termination. You can also handle specific problems, such as insufficient memory, with concise structured code that does not rely on **goto** statements or elaborate testing of return codes.

The try-except and try-finally statements are a Microsoft extension to the C language that enables applications to gain control of a program after events that would normally terminate execution.

Note
Structured exception handling works with C and C++ source files. However, it is not specifically designed for C++. Although destructors for local objects will be called if you use structured exception handling in a C++ program (if you use /EHsc), you can ensure that your code is more portable by using C++ exception handling. The C++ exception handling mechanism is mo re flexible, in that it can handle exceptions of any type.

There are two structured exception handling mechanisms:

- Exception handlers, which can respond to or dismiss the exception
- Termination handlers, which are called when an exception causes termination inside a block of code

These two types of handlers are distinct, yet they are closely related through a process called "unwinding the stack." When an exception occurs, Windows looks for the most recently installed exception handler that is currently active. The handler can do one of three things:

- Pass control to other handlers (fail to recognize the exception).
- Recognize but dismiss the exception.
- Recognize and handle the exception.

The exception handler that recognizes the exception may not be in the function that was running when the exception occurred. In some cases it may be in a function much higher on the stack. The currently running function, as well as all functions on the stack frame, are terminated. During this process, the stack is "unwound": local variables of terminated functions, unless they are **static**, are cleared from the stack.

As it unwinds the stack, the operating system calls any termination handlers you've written for each function. Use of a termination handler gives you a chance to clean up resources that otherwise would remain open due to abnormal termination. If you've entered a critical section, you can exit in the termination handler. If the program is going to shut down, you can perform other housekeeping tasks such as closing and removing temporary files.

If you have C modules that use structured exception handling, they can be mixed with C++ modules that use C++ exception handling. See Exception Handling Differences.

For more information, see:

- Writing an exception handler.
- Writing a termination handler.
- Using structured exception handling with C++.

END Microsoft Specific

See Also

Reference

Exception Handling in Visual C++

C++ Keywords

Writing an Exception Handler

Exception handlers are typically used to respond to specific errors. You can use the exception-handling syntax to filter out all exceptions other than those you know how to handle. Other exceptions should be passed to other handlers (possibly in the run-time library or the operating system) written to look for those specific exceptions.

Exception handlers use the try-except statement.

What do you want to know more about?

- The try-except statement
- Writing an exception filter
- Raising software exceptions
- Hardware exceptions
- Restrictions on exception handlers

See Also

Reference

Structured Exception Handling (C++)

try-except Statement

Microsoft Specific

The following syntax describes a try-except statement:

```
__try
{
   // guarded code
}
__except ( expression )
{
   // exception handler code
}
```

Remarks

The **try-except** statement is a Microsoft extension to the C and C++ languages that enables target applications to gain control when events that normally terminate program execution occur. Such events are called exceptions, and the mechanism that deals with exceptions is called structured exception handling.

For related information, see the try-finally statement.

Exceptions can be either hardware-based or software-based. Even when applications cannot completely recover from hardware or software exceptions, structured exception handling makes it possible to display error information and trap the internal state of the application to help diagnose the problem. This is especially useful for intermittent problems that cannot be reproduced easily.

Note
Structured exception handling works with Win32 for both C and C++ source files. However, it is not specifically designed for C++. You can ensure that your code is more portable by using C++ exception handling. Also, C++ exception handling is more flexible, in that it can handle exceptions of any type. For C++ programs, it is recommended that you use the C++ exception-handling mechanism (try, catch, and throw statements).

The compound statement after the **__try** clause is the body or guarded section. The compound statement after the **__except** clause is the exception handler. The handler specifies a set of actions to be taken if an exception is raised during execution of the body of the guarded section. Execution proceeds as follows:

1. The guarded section is executed.
2. If no exception occurs during execution of the guarded section, execution continues at the statement after the **__except** clause.
3. If an exception occurs during execution of the guarded section or in any routine the guarded section calls, the **__except** *expression* (called the *filter* expression) is evaluated and the value determines how the exception is handled. There are three values:

   EXCEPTION_CONTINUE_EXECUTION (–1) Exception is dismissed. Continue execution at the point where the exception occurred.

   EXCEPTION_CONTINUE_SEARCH (0) Exception is not recognized. Continue to search up the stack for a handler, first for containing **try-except** statements, then for handlers with the next highest precedence.

   EXCEPTION_EXECUTE_HANDLER (1) Exception is recognized. Transfer control to the exception handler by executing the **__except** compound statement, then continue execution after the **__except** block.

Because the **__except** expression is evaluated as a C expression, it is limited to a single value, the conditional-expression operator, or the comma operator. If more extensive processing is required, the expression can call a routine that returns one of the three values listed above.

Each application can have its own exception handler.

It is not valid to jump into a **__try** statement, but valid to jump out of one. The exception handler is not called if a process is terminated in the middle of executing a **try-except** statement.

For more information, see Knowledge Base article Q315937 : HOW TO: Trap Stack Overflow in a Visual C++ Application.

Structured Exception Handling Intrinsic Functions

Structured exception handling provides two intrinsic functions that are available to use with the **try-except** statement: **GetExceptionCode** and **GetExceptionInformation**.

GetExceptionCode returns the code (a 32-bit integer) of the exception.

The intrinsic function **GetExceptionInformation** returns a pointer to a structure containing additional information about the exception. Through this pointer, you can access the machine state that existed at the time of a hardware exception. The structure is as follows:

```
struct _EXCEPTION_POINTERS {
      EXCEPTION_RECORD *ExceptionRecord,
      CONTEXT *ContextRecord }
```

The pointer types **_EXCEPTION_RECORD** and **_CONTEXT** are defined in the include file EXCPT.H.

You can use **GetExceptionCode** within the exception handler. However, you can use **GetExceptionInformation** only within the exception filter expression. The information it points to is generally on the stack and is no longer available when control is transferred to the exception handler.

The intrinsic function **AbnormalTermination** is available within a termination handler. It returns 0 if the body of the **try-finally** statement terminates sequentially. In all other cases, it returns 1.

EXCPT.H defines some alternate names for these intrinsics:

GetExceptionCode is equivalent to _exception_code

GetExceptionInformation is equivalent to _exception_info

AbnormalTermination is equivalent to _abnormal_termination

Example

```
// exceptions_try_except_Statement.cpp
// Example of try-except and try-finally statements
#include <stdio.h>
#include <windows.h> // for EXCEPTION_ACCESS_VIOLATION
#include <excpt.h>
int filter(unsigned int code, struct _EXCEPTION_POINTERS *ep) {
   puts("in filter.");
   if (code == EXCEPTION_ACCESS_VIOLATION) {
      puts("caught AV as expected.");
      return EXCEPTION_EXECUTE_HANDLER;
   }
   else {
      puts("didn't catch AV, unexpected.");
      return EXCEPTION_CONTINUE_SEARCH;
   };
}
int main()
{
   int* p = 0x00000000;   // pointer to NULL
   puts("hello");
   __try{
```

```
      puts("in try");
      __try{
         puts("in try");
         *p = 13;    // causes an access violation exception;
      }__finally{
         puts("in finally. termination: ");
         puts(AbnormalTermination() ? "\tabnormal" : "\tnormal");
      }
   }__except(filter(GetExceptionCode(), GetExceptionInformation())){
      puts("in except");
   }
   puts("world");
}
```

Output

```
hello
in try
in try
in filter.
caught AV as expected.
in finally. termination:
        abnormal
in except
world
```

END Microsoft Specific

See Also

Reference

Writing an Exception Handler

Structured Exception Handling (C++)

C++ Keywords

Writing an Exception Filter

You can handle an exception either by jumping to the level of the exception handler or by continuing execution. Instead of using the exception handler code to handle the exception and falling through, you can use *filter* to clean up the problem and then, by returning –1, resume normal flow without clearing the stack.

Note

Some exceptions cannot be continued. If *filter* evaluates to –1 for such an exception, the system raises a new exception. When you call RaiseException, you determine whether the exception will continue.

For example, the following code uses a function call in the *filter* expression: this function handles the problem and then returns –1 to resume normal flow of control:

```
// exceptions_Writing_an_Exception_Filter.cpp
#include <windows.h>
int main() {
   int Eval_Exception( int );

   __try {}

   __except ( Eval_Exception( GetExceptionCode( ))) {
      ;
   }

}
void ResetVars( int ) {}
int Eval_Exception ( int n_except ) {
   if ( n_except != STATUS_INTEGER_OVERFLOW &&
      n_except != STATUS_FLOAT_OVERFLOW )   // Pass on most exceptions
   return EXCEPTION_CONTINUE_SEARCH;

   // Execute some code to clean up problem
   ResetVars( 0 );   // initializes data to 0
   return EXCEPTION_CONTINUE_EXECUTION;
}
```

It is a good idea to use a function call in the *filter* expression whenever *filter* needs to do anything complex. Evaluating the expression causes execution of the function, in this case, `Eval_Exception`.

Note the use of GetExceptionCode to determine the exception. You must call this function inside the filter itself. `Eval_Exception` cannot call **GetExceptionCode**, but it must have the exception code passed to it.

This handler passes control to another handler unless the exception is an integer or floating-point overflow. If it is, the handler calls a function (`ResetVars` is only an example, not an API function) to reset some global variables. *Statement-block-2*, which in this example is empty, can never be executed because `Eval_Exception` never returns EXCEPTION_EXECUTE_HANDLER (1).

Using a function call is a good general-purpose technique for dealing with complex filter expressions. Two other C language features that are useful are:

- The conditional operator
- The comma operator

The conditional operator is frequently useful, because it can be used to check for a specific return code and then return one of two different values. For example, the filter in the following code recognizes the exception only if the exception is `STATUS_INTEGER_OVERFLOW`:

```
__except( GetExceptionCode() == STATUS_INTEGER_OVERFLOW ? 1 : 0 ) {
```

The purpose of the conditional operator in this case is mainly to provide clarity, because the following code produces the same results:

```
__except( GetExceptionCode() == STATUS_INTEGER_OVERFLOW ) {
```

The conditional operator is more useful in situations where you might want the filter to evaluate to –1, EXCEPTION_CONTINUE_EXECUTION.

The comma operator enables you to perform multiple, independent operations inside a single expression. The effect is roughly that of executing multiple statements and then returning the value of the last expression. For example, the following code stores the exception code in a variable and then tests it:

```
__except( nCode = GetExceptionCode(), nCode == STATUS_INTEGER_OVERFLOW )
```

See Also

Reference

Writing an Exception Handler

Structured Exception Handling (C++)

Raising Software Exceptions

Some of the most common sources of program errors are not flagged as exceptions by the system. For example, if you attempt to allocate a memory block but there is insufficient memory, the run-time or API function does not raise an exception but returns an error code.

However, you can treat any condition as an exception by detecting that condition in your code and then reporting it by calling the RaiseException function. By flagging errors this way, you can bring the advantages of structured exception handling to any kind of run-time error.

To use structured exception handling with errors:

- Define your own exception code for the event.
- Call **RaiseException** when you detect a problem.
- Use exception-handling filters to test for the exception code you defined.

The WINERROR.H file shows the format for exception codes. To make sure that you do not define a code that conflicts with an existing exception code, set the third most significant bit to 1. The four most-significant bits should be set as shown in the following table.

Bits	Recommended binary setting	Description
31-30	11	These two bits describe the basic status of the code: 11 = error, 00 = success, 01 = informational, 10 = warning.
29	1	Client bit. Set to 1 for user-defined codes.
28	0	Reserved bit. (Leave set to 0.)

You can set the first two bits to a setting other than 11 binary if you want, although the "error" setting is appropriate for most exceptions. The important thing to remember is to set bits 29 and 28 as shown in the previous table.

The resulting error code should therefore have the highest four bits set to hexadecimal E. For example, the following definitions define exception codes that do not conflict with any Windows exception codes. (You may, however, need to check which codes are used by third-party DLLs.)

```
#define STATUS_INSUFFICIENT_MEM       0xE0000001
#define STATUS_FILE_BAD_FORMAT        0xE0000002
```

After you have defined an exception code, you can use it to raise an exception. For example, the following code raises the STATUS_INSUFFICIENT_MEM exception in response to a memory allocation problem:

```
lpstr = _malloc( nBufferSize );
if (lpstr == NULL)
    RaiseException( STATUS_INSUFFICIENT_MEM, 0, 0, 0);
```

If you want to simply raise an exception, you can set the last three parameters to 0. The three last parameters are useful for passing additional information and setting a flag that prevents handlers from continuing execution. See the RaiseException function in the *Platform SDK* for more information.

In your exception-handling filters, you can then test for the codes you've defined. For example:

```
__try {
    ...
}
__except (GetExceptionCode() == STATUS_INSUFFICIENT_MEM ||
        GetExceptionCode() == STATUS_FILE_BAD_FORMAT )
```

See Also

Reference

Writing an Exception Handler

Structured Exception Handling (C++)

Hardware Exceptions

Most of the standard exceptions recognized by the operating system are hardware-defined exceptions. Windows recognizes a few low-level software exceptions, but these are usually best handled by the operating system.

Windows maps the hardware errors of different processors to the exception codes in this section. In some cases, a processor may generate only a subset of these exceptions. Windows preprocesses information about the exception and issues the appropriate exception code.

The hardware exceptions recognized by Windows are summarized in the following table:

Exception code	Cause of exception
STATUS_ACCESS_VIOLATION	Reading or writing to an inaccessible memory location.
STATUS_BREAKPOINT	Encountering a hardware-defined breakpoint; used only by debuggers.
STATUS_DATATYPE_MISALIGNMENT	Reading or writing to data at an address that is not properly aligned; for example, 16-bit entities must be aligned on 2-byte boundaries. (Not applicable to Intel 80*x*86 processors.)
STATUS_FLOAT_DIVIDE_BY_ZERO	Dividing floating-point type by 0.0.
STATUS_FLOAT_OVERFLOW	Exceeding maximum positive exponent of floating-point type.
STATUS_FLOAT_UNDERFLOW	Exceeding magnitude of lowest negative exponent of floating-point type.
STATUS_FLOATING_RESEVERED_OPERAND	Using a reserved floating-point format (invalid use of format).
STATUS_ILLEGAL_INSTRUCTION	Attempting to execute an instruction code not defined by the processor.
STATUS_PRIVILEGED_INSTRUCTION	Executing an instruction not allowed in current machine mode.
STATUS_INTEGER_DIVIDE_BY_ZERO	Dividing an integer type by 0.
STATUS_INTEGER_OVERFLOW	Attempting an operation that exceeds the range of the integer.
STATUS_SINGLE_STEP	Executing one instruction in single-step mode; used only by debuggers.

Many of the exceptions listed in the previous table are intended to be handled by debuggers, the operating system, or other low-level code. With the exception of integer and floating-point errors, your code should not handle these errors. Thus, you should usually use the exception-handling filter to ignore exceptions (evaluate to 0). Otherwise, you may prevent lower-level mechanisms from responding appropriately. You can, however, take appropriate precautions against the potential effect of these low-level errors by writing termination handlers.

See Also

Reference

Writing an Exception Handler

Structured Exception Handling (C++)

Restrictions on Exception Handlers

The principal limitation to using exception handlers in code is that you cannot use a **goto** statement to jump into a **__try** statement block. Instead, you must enter the statement block through normal flow of control. You can jump out of a **__try** statement block and nest exception handlers as you choose.

See Also

Reference

Writing an Exception Handler

Structured Exception Handling (C++)

Writing a Termination Handler

Unlike an exception handler, a termination handler is always executed, regardless of whether the protected block of code terminated normally. The sole purpose of the termination handler should be to ensure that resources, such as memory, handles, and files, are properly closed regardless of how a section of code finishes executing.

Termination handlers use the try-finally statement.

What do you want to know more about?

- The try-finally statement
- Cleaning up resources
- Timing of actions in exception handling
- Restrictions on termination handlers

See Also

Reference

Structured Exception Handling (C++)

try-finally Statement

Microsoft Specific

The following syntax describes the **try-finally** statement:

```
__try {
   // guarded code
}
__finally ( expression ) {
   // termination code
}
```

Grammar
try-finally-statement:

__try *compound-statement*

__finally *compound-statement*

The **try-finally** statement is a Microsoft extension to the C and C++ languages that enables target applications to guarantee execution of cleanup code when execution of a block of code is interrupted. Cleanup consists of such tasks as deallocating memory, closing files, and releasing file handles. The **try-finally** statement is especially useful for routines that have several places where a check is made for an error that could cause premature return from the routine.

For related information and a code sample, see try-except Statement. For more information on structured exception handling in general, see Structured Exception Handling. For more information on handling exceptions in managed applications, see Exception Handling under /clr.

Note

Structured exception handling works with Win32 for both C and C++ source files. However, it is not specifically designed for C++. You can ensure that your code is more portable by using C++ exception handling. Also, C++ exception handling is more flexible, in that it can handle exceptions of any type. For C++ programs, it is recommended that you use the C++ exception-handling mechanism (try, catch, and throw statements).

The compound statement after the **__try** clause is the guarded section. The compound statement after the **__finally** clause is the termination handler. The handler specifies a set of actions that execute when the guarded section is exited, regardless of whether the guarded section is exited by an exception (abnormal termination), or by standard fall through (normal termination).

Control reaches a **__try** statement by simple sequential execution (fall through). When control enters the **__try**, its associated handler becomes active. If the flow of control reaches the end of the try block, execution proceeds as follows:

1. The termination handler is invoked.
2. When the termination handler completes, execution continues after the **__finally** statement. Regardless of how the guarded section ends (for example, via a **goto** out of the guarded body or a **return** statement), the termination handler is executed *before* the flow of control moves out of the guarded section.

   A **__finally** statement does not block searching for an appropriate exception handler.

If an exception occurs in the **__try** block, the operating system must find a handler for the exception or the program will fail. If a handler is found, any and all **__finally** blocks are executed and execution resumes in the handler.

For example, suppose a series of function calls links function A to function D, as shown in the following figure. Each function has one termination handler. If an exception is raised in function D and handled in A, the termination handlers are called in this order as the system unwinds the stack: D, C, B.

Order of Termination-Handler Execution

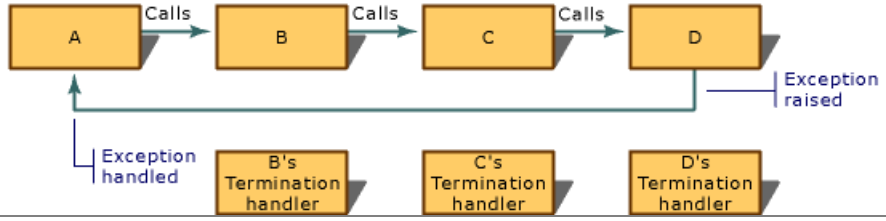

Note

The behavior of try-finally is different from some other languages that support the use of **finally**, such as C#. A single **__try** may have either, but not both, of **__finally** and **__except**. If both are to be used together, an outer try-except statement must enclose the inner try-finally statement. The rules specifying when each block executes are also different.

The __leave keyword

The **__leave** keyword is valid within a **try-finally** statement block. The effect of **__leave** is to jump to the end of the **try-finally** block. The termination handler is immediately executed. Although a **goto** statement can be used to accomplish the same result, a **goto** statement causes stack unwinding. The **__leave** statement is more efficient because it does not involve stack unwinding.

Abnormal Termination

Exiting a **try-finally** statement using the **longjmp** run-time function is considered abnormal termination. It is illegal to jump into a **__try** statement, but legal to jump out of one. All **__finally** statements that are active between the point of departure (normal termination of the **__try** block) and the destination (the **__except** block that handles the exception) must be run. This is called a local unwind.

If a **try** block is prematurely terminated for any reason, including a jump out of the block, the system executes the associated **finally** block as a part of the process of unwinding the stack. In such cases, the AbnormalTermination function returns TRUE if called from within the **finally** block; otherwise, it returns FALSE.

The termination handler is not called if a process is killed in the middle of executing a **try-finally** statement.

END Microsoft Specific

See Also

Reference

Writing a Termination Handler
Structured Exception Handling (C++)
C++ Keywords
Termination-Handler Syntax

Cleaning up Resources

During termination-handler execution, you may not know which resources are actually allocated before the termination handler was called. It is possible that the **__try** statement block was interrupted before all resources were allocated, so that not all resources were opened.

Therefore, to be safe, you should check to see which resources are actually open before proceeding with termination-handling cleanup. A recommended procedure is to:

1. Initialize handles to NULL.
2. In the **__try** statement block, allocate resources. Handles are set to positive values as the resource is allocated.
3. In the **__finally** statement block, release each resource whose corresponding handle or flag variable is nonzero or not NULL.

Example

For example, the following code uses a termination handler to close three files and a memory block that were allocated in the **__try** statement block. Before cleaning up a resource, the code first checks to see if the resource was allocated.

```
// exceptions_Cleaning_up_Resources.cpp
#include <stdlib.h>
#include <malloc.h>
#include <stdio.h>
#include <windows.h>

void fileOps() {
   FILE  *fp1 = NULL,
         *fp2 = NULL,
         *fp3 = NULL;
   LPVOID lpvoid = NULL;
   errno_t err;

   __try {
      lpvoid = malloc( BUFSIZ );

      err = fopen_s(&fp1, "ADDRESS.DAT", "w+" );
      err = fopen_s(&fp2, "NAMES.DAT", "w+" );
      err = fopen_s(&fp3, "CARS.DAT", "w+" );
   }
   __finally {
      if ( fp1 )
         fclose( fp1 );
      if ( fp2 )
         fclose( fp2 );
      if ( fp3 )
         fclose( fp3 );
      if ( lpvoid )
         free( lpvoid );
   }
}

int main() {
   fileOps();
}
```

See Also

Reference

Writing a Termination Handler

Structured Exception Handling (C++)

Timing of Exception Handling: A Summary

A termination handler is executed no matter how the **__try** statement block is terminated. Causes include jumping out of the **__try** block, a **longjmp** statement that transfers control out of the block, and unwinding the stack due to exception handling.

Note
Visual C++ supports two forms of the **setjmp** and **longjmp** statements. The fast version bypasses termination handling but is more efficient. To use this version, include the file SETJMP.H. The other version supports termination handling as described in the previous paragraph. To use this version, include the file SETJMPEX.H. The increase in performance of the fast version depends on hardware configuration.

The operating system executes all termination handlers in the proper order before any other code can be executed, including the body of an exception handler.

When the cause for interruption is an exception, the system must first execute the filter portion of one or more exception handlers before deciding what to terminate. The order of events is:

1. An exception is raised.
2. The system looks at the hierarchy of active exception handlers and executes the filter of the handler with highest precedence; this is the exception handler most recently installed and most deeply nested, in terms of blocks and function calls.
3. If this filter passes control (returns 0), the process continues until a filter is found that does not pass control.
4. If this filter returns –1, execution continues where the exception was raised, and no termination takes place.
5. If the filter returns 1, the following events occur:
   - The system unwinds the stack, clearing all stack frames between the currently executing code (where the exception was raised) and the stack frame that contains the exception handler gaining control.
   - As the stack is unwound, each termination handler on the stack is executed.
   - The exception handler itself is executed.
   - Control passes to the line of code after the end of this exception handler.

See Also

Reference

Writing a Termination Handler
Structured Exception Handling (C++)

Restrictions on Termination Handlers

You cannot use a **goto** statement to jump into a **__try** statement block or a **__finally** statement block. Instead, you must enter the statement block through normal flow of control. (You can, however, jump out of a **__try** statement block.) Also, you cannot nest an exception handler or termination handler inside a **__finally** block.

In addition, some kinds of code permitted in a termination handler produce questionable results, so you should use them with caution, if at all. One is a **goto** statement that jumps out of a **__finally** statement block. If the block is executing as part of normal termination, nothing unusual happens. But if the system is unwinding the stack, that unwinding stops, and the current function gains control as if there were no abnormal termination.

A **return** statement inside a **__finally** statement block presents roughly the same situation. Control returns to the immediate caller of the function containing the termination handler. If the system was unwinding the stack, this process is halted, and the program proceeds as if there had been no exception raised.

See Also

Reference

Writing a Termination Handler

Structured Exception Handling (C++)

Using Structured Exception Handling with C++

Structured exception handling described in these articles works with both C and C++ source files. However, it is not specifically designed for C++ and is not recommended. You can ensure that your code is more portable by using C++ exception handling. Also, the C++ exception handling mechanism is more flexible, in that it can handle exceptions of any type.

Microsoft C++ now supports the C++ exception handling model, based on the ANSI C++ Standard. This mechanism automatically handles destruction of local objects during stack unwind. If you are writing fault-tolerant C++ code, and you want to implement exception handling, it is strongly recommended that you use C++ exception handling, rather than structured exception handling. (Note that while the C++ compiler supports structured exception handling constructs as described in these articles, the standard C compiler does not support the C++ exception handling syntax.) For detailed information about C++ exception handling, see C++ Exception Handling and the *Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup.

See Also

Reference

Structured Exception Handling (C++)

Templates

C++ templates enable you to define a family of functions or classes that can operate on different types of information. The Microsoft implementation of C++ templates is based on the ISO/ANSI C++ Standard.

Use templates in situations that result in duplication of the same code for multiple types. For example, you can use function templates to create a set of functions that apply the same algorithm to different data types. You can also use class templates to develop a set of typesafe classes. Templates are sometimes a better solution than C macros and void pointers, and they are especially useful when working with collections (one of the main uses for templates in MFC) and smart pointers.

In This Section

Overview

Provides general information about templates, or parameterized types, in Visual C++.

Specifications

Provides a specification for a **template** declaration.

typename

Describes the **typename** keyword.

Referencing Templates [C++ Language]

Provides a task showing how to reference templates in your code.

Angle Bracket Placement

Describes how angle brackets are placed in templates.

Differences from Other Implementations

Discusses Microsoft-specific implementations of templates.

Name Resolution

Discusses the three types of names in template definitions.

Class Templates

Describes how to use and work with class templates.

Function Templates

Describes how to use and work with function templates.

Related Sections

ATL Reference

Provides reference material for the ATL Library, a set of template-based C++ classes that simplify the programming of COM objects.

C++ Language Reference

Describes the C++ programming language as implemented in Microsoft C++.

Generics and Templates

Discusses the differences between generics and templates.

Templates Overview

Templates, which are sometimes called parameterized types, are mechanisms for generating functions and classes based on type parameters. By using templates, you can design a single class or function that operates on data of many types, instead of having to create a separate class for each type.

Remarks

For example, to create a typesafe function that returns the minimum of two parameters without using templates, you would write a set of overloaded functions like this:

```
// what_are_templates1.cpp
// compile with: /c
// min for ints
int min( int a, int b ) {
   return ( a < b ) ? a : b;
}

// min for longs
long min( long a, long b ) {
   return ( a < b ) ? a : b;
}

// min for chars
char min( char a, char b ) {
   return ( a < b ) ? a : b;
}
```

By using templates, you can reduce this duplication to a single function template:

```
// what_are_templates2.cpp
// compile with: /c
template <class T> T min( T a, T b ) {
   return ( a < b ) ? a : b;
}
```

Templates can significantly reduce source code size and increase code flexibility without reducing type safety.

There are two main types of templates: function templates and class templates. In the previous example, `min` is a function template. A class template is a class with a parameter, such as:

```
// what_are_templates3.cpp
template <class T> class A {
   T m_t;
   public:
      A(T t): m_t(t) {}
      void f(T t);
};

int main() {
   A<int> a(10);
}
```

Templates are declared and defined somewhat like other functions and classes, with some major differences. A template declaration does not fully define a function or class; it only defines a syntactical skeleton for a class or function. An actual class or function is created from a template by a process called instantiation. The individual classes or functions created are referred to as instantiated. For example, a class template:

```
template <class T> struct A { . . . };
```

can be used to instantiate classes for `A<int>`, `A<char>`, `A<int*>`, `A<MyClass*>`, and so on.

Instantiation of classes or functions can be done explicitly or implicitly. Explicit instantiation is a way of calling out in code what versions of the template are to be generated. Implicit instantiation allows templates to be instantiated as needed at the point where they are first used.

Templates can also be parameterized by a value parameter, in which case the template parameter is declared like the parameter to a function. Floating-point types and class types are not allowed as value parameters.

```
// what_are_templates4.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <int i> class A {
   int array[i];
public:
   A() { memset(array, 0, i*sizeof(int)); }
};

int main() {
   A<10> a;
}
```

A common problem with templates is that they can be a one-size-fits-all solution, meaning that the same code applies to all types. If you need to customize the behavior of the template for a particular type, then you can use specialization. Using explicit specialization, a template can be specialized for a particular real type, not a generic type. A class template can also be partially specialized, which is useful if you have a template with multiple type parameters and you only want to customize the behavior with respect to some but not all parameters. A partial specialization is still generic and needs real template arguments to produce an actual instantiated class.

See Also

Other Resources

Templates

Template Specifications

The **template** declaration specifies a set of parameterized classes or functions.

```
template < template-parameter-list > declaration
```

Remarks

The *template-parameter-list* is a comma-separated list of template parameters, which may be types (in the form **class** *identifier*, **typename** *identifier*, or **template <** *template-parameter-list* **> class** *identifier*) or non-type parameters to be used in the template body. The syntax for a template parameter is one of the following:

```
parameter-declaration
class identifier [ = typename ]
typename identifier [ = typename ]
template < template-parameter-list > class [identifier][= name]
```

You can instantiate a class template much like you would instantiate a normal class, but you must include the template arguments within angle brackets (**<>**). These template arguments can be any type if the template argument list contains the class or **typename** keyword, or a value of the appropriate type if the argument is a non-type argument. No special syntax is required to call a function template, although the angle brackets and template arguments can be required if the template parameters cannot be deduced from the arguments to the function.

The *template-parameter-list* is a list of parameters used by the template function that specifies which parts of the following code will vary. For example:

```
template< class T, int i > class MyStack...
```

In this case, the template can receive a type (`class T`) and a constant parameter (`int i`). The template will use type `T` and the constant integer `i` upon instantiation. Within the body of the `MyStack` declaration, you must refer to the `T` identifier.

A template declaration itself does not generate code; it specifies a family of classes or functions, one or more of which will be generated when referenced by other code.

Template declarations have global, namespace, or class scope. They cannot be declared within a function.

The following example illustrates the declaration, definition, and instantiation of a class template with a type parameter `T` and a non-type template parameter `i`.

```
// template_specifications1.cpp
template <class T, int i> class TestClass
{
public:
   char buffer[i];
   T testFunc(T* p1 );
};

template <class T, int i>
T TestClass<T,i>::testFunc(T* p1)
{
    return *(p1++)
};

// To create an instance of TestClass
TestClass<char, 5> ClassInst;
int main()
{
}
```

Non-type template arguments

Non-type template parameters must be of integral, enumeration, pointer, reference, or pointer to member type, and must be constant at compile time. They can be qualified as const or volatile types. Floating point values are not allowed as template parameters. Objects of class, struct or union type are not allowed as non-type template parameters, although pointers to such objects are allowed. Arrays passed as non-type template parameters are converted into pointers. Functions passed as non-type parameters are treated as function pointers. String literals are not allowed as template parameters.

Using typename in a Template Declaration

The typename keyword can be used in the template parameter list. The following template declarations are identical:

```
template< class T1, class T2 > class X...
template< typename T1, typename T2 > class X...
```

Default Arguments for Template Parameters

Class templates can have default arguments specified using the **=** sign followed by the default type or value. Function templates cannot have default arguments. For more information, see Default Arguments for Class Templates .:

```
template<typename Type> class allocator {};
template<typename Type,
   typename Allocator = allocator<Type> > class stack
{
};
stack<int> MyStack;
```

Reuse of Template Parameters

Template parameters can be reused in the template parameter list. For example, the following code is allowed:

```
// template_specifications2.cpp

class Y
{
};
template<class T, T* pT> class X1
{
};
template<class T1, class T2 = T1> class X2
{
};

Y aY;

X1<Y, &aY> x1;
X2<int> x2;

int main()
{
}
```

Template template parameters

Template parameters can themselves be templates. This construct means that the argument must itself be a template, not a class constructed from template. In the following example, the name `A` of the template parameter for a template template parameter can be omitted, because there is no way that it can be used.

```
// template_specifications3.cpp
#include <stdio.h>

template <class T> struct str1
{
   T t;
};

template <template<class A> class T> struct str2
```

```
{
    T<int> t;
};

int main()
{
    str2<str1> mystr2;
    mystr2.t.t = 5;
    printf_s("%d\n", mystr2.t.t);
}
```

Output

```
5
```

References as Template Parameters

Visual Studio .NET 2003 introduced the ability to use references as non-type template parameters. This was not allowed in previous versions.

```
// references__supported_as_nontype_template_parameters.cpp
#include <stdio.h>

extern "C" int printf_s(const char*,...);
template <int & ri> struct S
{
   S()
   {
      printf_s("ri is %d\n", ri);
   }

   ~S()
   {
      printf_s("ri is %d\n", ri);
   }
};

int i = 1;

int main()
{
   S<i> s;
   i = 0;
}
```

Output

```
ri is 1
ri is 0
```

Nested Template Instances

Versions of Visual Studio prior to Visual Studio 2005 required that whitespace be inserted between template parameter lists when nested template instances were declared. The following syntax is now allowed in Visual Studio 2005:

```
// template_specifications4.cpp

template <typename T>
class A
{
};

template <int n>
class B
{
};
```

```
int main()
{
   A<B<22>>();
}
```

See Also

Reference

C++ Keywords

Other Resources

Templates

Advantages of Templates [C++]

You can use templates to:

- Create a typesafe collection class (for example, a stack) that can operate on data of any type.
- Add extra type checking for functions that would otherwise take **void** pointers.
- Encapsulate groups of operator overrides to modify type behavior (such as smart pointers).

Most of these uses can be implemented without templates; however, templates offer several advantages:

- Templates are easier to write. You create only one generic version of your class or function instead of manually creating specializations.
- Templates can be easier to understand, since they can provide a straightforward way of abstracting type information.
- Templates are typesafe. Because the types that templates act upon are known at compile time, the compiler can perform type checking before errors occur.

For more information, see:

- Templates and Macros
- Templates and Void Pointers
- Templates and Smart Pointers
- Templates and Collection Classes

Templates and Macros

In many ways, templates work like preprocessor macros, replacing the templated variable with the given type. However, there are many differences between a macro like this:

```
#define min(i, j) (((i) < (j)) ? (i) : (j))
```

and a template:

```
template<class T> T min (T i, T j) { return ((i < j) ? i : j) }
```

Here are some problems with the macro:

- There is no way for the compiler to verify that the macro parameters are of compatible types. The macro is expanded without any special type checking.
- The `i` and `j` parameters are evaluated twice. For example, if either parameter has a postincremented variable, the increment is performed two times.
- Because macros are expanded by the preprocessor, compiler error messages will refer to the expanded macro, rather than the macro definition itself. Also, the macro will show up in expanded form during debugging.

Templates and Void Pointers

Many functions that are now implemented with void pointers can be implemented with templates. Void pointers are often used to allow functions to operate on data of an unknown type. When using void pointers, the compiler cannot distinguish types, so it cannot perform type checking or type-specific behavior such as using type-specific operators, operator overloading, or constructors and destructors.

With templates, you can create functions and classes that operate on typed data. The type looks abstracted in the template definition. However, at compile time the compiler creates a separate version of the function for each specified type. This enables the compiler to treat class and function templates as if they acted on specific types. Templates can also improve coding clarity, because you do not need to create special cases for complex types such as structures.

Templates and Smart Pointers

C++ allows you to create smart pointer classes that encapsulate pointers and override pointer operators to add new functionality to pointer operations. Templates allow you to create generic wrappers to encapsulate pointers of almost any type.

The following code outlines a simple reference count garbage collector. The template class `Ptr<T>` implements a garbage collecting pointer to any class derived from `RefCount`.

Example

```
// templates_and_smart_pointers.cpp
#include <stdio.h>

#define TRACE printf

class RefCount {
   int crefs;
public:
   RefCount(void) {
      crefs = 0;
   }

   ~RefCount() {
      TRACE("goodbye(%d)\n", crefs);
   }

   void upcount(void) {
      ++crefs;
      TRACE("up to %d\n", crefs);
   }

   void downcount(void)
   {
      if (--crefs == 0)
         delete this;
   else
      TRACE("downto %d\n", crefs);
   }
};


class Sample : public RefCount {
public:
   void doSomething(void) {
      TRACE("Did something\n");
   }
};


template <class T> class Ptr {
   T* p;
public:
   Ptr(T* p_) : p(p_) {
      p->upcount();
   }

   ~Ptr(void) {
      p->downcount();
   }

   operator T*(void) {
      return p;
   }

   T& operator*(void) {
      return *p;
   }

   T* operator->(void) {
```

```
      return p;
   }

   Ptr& operator=(Ptr<T> &p_) {
      return operator=((T *) p_);
   }

   Ptr& operator=(T* p_) {
      p_->upcount();
      p->downcount();
      p = p_;
      return *this;
   }

   Ptr(const Ptr<T> &p_) {
      p = p_.p;
      p->upcount();
   };
};


int main() {
   Ptr<Sample> p  = new Sample; // sample #1
   Ptr<Sample> p2 = new Sample; // sample #2
   p = p2; // #1 will have 0 crefs, so it is destroyed;
         // #2 will have 2 crefs.
   p->doSomething();
   return 0;
   // As p2 and p go out of scope, their destructors call
   // downcount. The cref variable of #2 goes to 0, so #2 is
   // destroyed
}
```

Output

```
up to 1
up to 1
up to 2
goodbye(0)
Did something
downto 1
goodbye(0)
```

Classes `RefCount` and `Ptr<T>` together provide a simple garbage collection solution for any class that can afford the `int` per instance overhead to inherit from `RefCount`. Note that the primary benefit of using a parametric class like `Ptr<T>` instead of a more generic class like `Ptr` is that the former is completely typesafe. The preceding code ensures that a `Ptr<T>` can be used almost anywhere a `T*` is used; in contrast, a generic `Ptr` would only provide implicit conversions to `void*`.

For example, consider a class used to create and manipulate garbage collected files, symbols, strings, and so forth. From the class template `Ptr<T>`, the compiler will create template classes `Ptr<File>`, `Ptr<Symbol>`, `Ptr<String>`, and so on, and their member functions: `Ptr<File>::~Ptr()`, `Ptr<File>::operator File*()`, `Ptr<String>::~Ptr()`, `Ptr<String>::operator String*()`, and so on.

Templates and Collection Classes

Templates are a good way of implementing collection classes.

Example

```
// templates_and_collection_classes.cpp
template <class T, int i> class MyStack
{
   T StackBuffer[i];
   int cItems;
public:
   MyStack( void ) : cItems( i )
   {
   };
```

```
   void push( const T item );
   T pop( void );
};

template <class T, int i> void MyStack< T, i >::push( const T item )
{
   if( cItems > 0 )
      StackBuffer[--cItems] = item;
   else
      throw "Stack overflow error.";
}

template <class T, int i> T MyStack< T, i >::pop( void )
{
   if( cItems < i )
      return StackBuffer[cItems++]
   else
      throw "Stack underflow error.";
}

int main()
{
   MyStack<char , 1> v;
   v.push('a');
}
```

The `MyStack` collection is a simple implementation of a stack. The two template parameters, `T` and `i`, specify the type of elements in the stack and the maximum number of that item in the stack. The `push` and `pop` member functions add and remove items on the stack, with the stack growing from the bottom.

typename

Tells the compiler that an unknown identifier is a type.

```
typename identifier;
```

Remarks

Use this keyword only in template definitions.

This keyword must be used if the name is a qualified name dependent on a template argument; it is optional if the qualified name is not dependent. For more information, see Templates and Name Resolution.

typename can be used by any type anywhere in a template declaration or definition. It is not allowed in the base class list, unless as a template argument to a template base class.

```
template <class T>
class C1 : typename T::InnerType // Error - typename not allowed.
{};
template <class T>
class C2 : A<typename T::InnerType>  // typename OK.
{};
```

The **typename** keyword can also be used in place of **class** in template parameter lists. For example, the following statements are identical:

```
template<class T1, class T2>...
template<typename T1, typename T2>...
```

Example

```
// typename.cpp
template<class T> class X
{
   typename T::Y m_y;   // treat Y as a type
};

int main()
{
}
```

See Also

Reference

C++ Keywords

Other Resources

Templates

Angle Bracket Placement

The first opening bracket after a template name is interpreted as part of the template's argument list, not as a greater-than operator. The first > after the opening angle bracket is interpreted as the end of the template's argument list, unless it is enclosed in parentheses, is part of a nested template, or is part of a template-style cast expression such as `static_cast<int>`.

Bad placement of angle brackets (<>) causes many template syntax errors. Make sure that you use proper spacing and parentheses to distinguish angle brackets from operators such as **>**, **>>** and **->**. For example:

```
TempClass< float, a > b ? a : b > test1;
```

should be rewritten as:

```
TempClass< float, (a > b ? a : b) > test1;
```

Similarly, pay extra attention when using macros that use angle brackets as template arguments.

See Also

Other Resources

Templates

Differences from Other Implementations

Microsoft Specific

The following list shows some differences between Microsoft C++ and other compilers.

- The compiler cannot instantiate a template outside of the module in which it is defined. Visual C++.NET does not support the **export** keyword.
- Templates cannot be used with functions declared with **__declspec (dllimport)** or **__declspec (dllexport)**.
- All template arguments must be of an unambiguous type that exactly matches that of the template parameter list. For example:

  ```
  template< class T > T check( T );
  template< class S > void watch( int (*)(S) );
  watch( check );     //error
  ```

  The compiler should instantiate the `check` templated function in the form `int check( int )`, but the inference cannot be followed.
- When resolving names used in class templates or function templates, all names are treated as dependent names. See Name Resolution for Dependent Types
- In a class template, the template parameter can be redefined in the scope of the class definition. See Name Resolution for Locally Declared Names

END Microsoft Specific
See Also
Other Resources
Templates

Templates and Name Resolution

In template definitions, there are three types of names.

- Locally declared names, including the name of the template itself and any names declared inside the template definition.
- Names from the enclosing scope outside the template definition.
- Names that depend in some way on the template arguments, referred to as dependent names.

While the first two names also pertain to class and function scopes, special rules for name resolution are required in template definitions to deal with the added complexity of dependent names. This is because the compiler knows little about these names until the template is instantiated, because they could be totally different types depending on which template arguments are used. Nondependent names are looked up according to the usual rules and at the point of definition of the template. These names, being independent of the template arguments, are looked up once for all template specializations. Dependent names are not looked up until the template is instantiated and are looked up separately for each specialization.

A type is dependent if it depends on the template arguments. Specifically, a type is dependent if it is:

- The template argument itself:

```
T
```

- A qualified name with a qualification including a dependent type:

```
T::myType
```

- A qualified name if the unqualified part identifies a dependent type:

```
N::T
```

- A const or volatile type for which the base type is a dependent type:

```
const T
```

- A pointer, reference, array, or function pointer type based on a dependent type:

```
T *, T &, T [10], T (*)()
```

- An array whose size is based on a template parameter:

```
template <int arg> class X {
int x[arg] ; // dependent type
}
```

- a template type constructed from a template parameter:

```
T<int>, MyTemplate<T>
```

Type Dependence and Value Dependence

Names and expressions dependent on a template parameter are categorized as type dependent or value dependent, depending on whether the template parameter is a type parameter or a value parameter. Also, any identifiers declared in a template with a type dependent on the template argument are considered value dependent, as is a integral or enumeration type initialized with a value-dependent expression.

Type-dependent and value-dependent expressions are expressions that involve variables that are type dependent or value

dependent. These expressions can have semantics that differ, depending on the parameters used for the template.

See Also

Other Resources

Templates

Name Resolution for Dependent Types

The first rule applying to dependent names is the use of the **typename** keyword. **typename** is used for qualified names in template definitions to tell the compiler that the given qualified name identifies a type, as in the following example. For more information, see typename.

```
// template_name_resolution1.cpp
#include <stdio.h>
template <class T> class X
{
public:
   void f(typename T::myType* mt) {}
};

class Yarg
{
public:
   struct myType { };
};

int main()
{
   X<Yarg> x;
   x.f(new Yarg::myType());
   printf("Name resolved using typename keyword.");
}
```

Output

```
Name resolved using typename keyword.
```

Another rule is that name lookup for dependent names examines names from both the context of the template definition (in the following example, this context would find `myFunction(char)`) and the context of the template instantiation. In the following example, the template is instantiated in main, so the `MyNamespace::myFunction` is visible from the point of instantiation and is picked as the better match. If `MyNamespace::myFunction` were renamed, `myFunction(char)` would be called instead.

All names are resolved as if they were dependent names. Because this behavior could change in future versions of the compiler, it is recommended that such names be fully qualified if there is any possible conflict.

```
//template_name_resolution2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

void myFunction(char)
{
   cout << "Char myFunction" << endl;
}

template <class T> class Class1
{
public:
   Class1(T i)
   {
      // If replaced with myFunction(1), myFunction(char)
      // will be called
      myFunction(i);
}
};

namespace MyNamespace
{
   void myFunction(int)
   {
      cout << "Int MyNamespace::myFunction" << endl;
```

```
   }
};

using namespace MyNamespace;

int main()
{
   Class1<int>* c1 = new Class1<int>(100);
}
```

Output

```
Int MyNamespace::myFunction
```

See Also

Reference

Templates and Name Resolution

Name Resolution for Locally Declared Names

The template's name itself can be referred to with or without the template arguments. In the scope of a class template, the name itself refers to the template. In the scope of a template specialization or partial specialization, the name alone refers to the specialization or partial specialization. Other specializations or partial specializations of the template can also be referenced, with the appropriate template arguments.

Example

The following code shows that the class template's name A is interpreted differently in the scope of a specialization or partial specialization.

```
// template_name_resolution3.cpp
// compile with: /c
template <class T> class A {
   A* a1;   // A refers to A<T>
   A<int>* a2;  // A<int> refers to a specialization of A.
   A<T*>* a3;   // A<T*> refers to the partial specialization A<T*>.
};

template <class T> class A<T*> {
   A* a4; // A refers to A<T*>.
};

template<> class A<int> {
   A* a5; // A refers to A<int>.
};
```

In the case of a name conflict between a template parameter and another object, the template parameter can or cannot be hidden. The following rules will help determine precedence.

The template parameter is in scope from the point where it first appears until the end of the class or function template. If the name appears again in the template argument list or in the list of base classes, it refers to the same type. In standard C++, no other name that is identical to the template parameter can be declared in the same scope. A Microsoft extension allows the template parameter to be redefined in the scope of the template. The following example shows using the template parameter in the base specification of a class template.

```
// template_name_resolution4.cpp
// compile with: /EHsc
template <class T>
class Base1 {};

template <class T>
class Derived1 : Base1<T> {};

int main() {
   // Derived1<int> d;
}
```

When defining a template's member functions outside the class template, a different template parameter name can be used. If the template member function definition uses a different name for the template parameter than the declaration does, and the name used in the definition conflicts with another member of the declaration, the member in the template declaration takes precedence.

```
// template_name_resolution5.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T> class C {
public:
   struct Z {
      Z() { cout << "Z::Z()" << endl; }
```

```
   };
   void f();
};

template <class Z>
void C<Z>::f() {
   // Z refers to the struct Z, not to the template arg;
   // Therefore, the constructor for struct Z will be called.
   Z z;
}

int main() {
   C<int> c;
   c.f();
}
```

Output

```
Z::Z()
```

When defining a template function or member function outside the namespace in which the template was declared, the template argument takes precedence over the names of other members of the namespace.

```
// template_name_resolution6.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

namespace NS {
   void g() { cout << "NS::g" << endl; }

   template <class T> struct C {
      void f();
      void g() { cout << "C<T>::g" << endl; }
   };
};

template <class T>
void NS::C<T>::f() {
   g(); // C<T>::g, not NS::g
};

int main() {
   NS::C<int> c;
   c.f();
}
```

Output

```
C<T>::g
```

In definitions that are outside of the template class declaration, if a template class has a base class that does not depend on a template argument and if the base class or one of its members has the same name as a template argument, then the base class or member name hides the template argument.

```
// template_name_resolution7.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

struct B {
   int i;
   void print() { cout << "Base" << endl; }
};

template <class T, int i> struct C : public B {
   void f();
```

```
};

template <class B, int i>
void C<B, i>::f() {
   B b;   // Base class b, not template argument.
   b.print();
   i = 1; // Set base class's i to 1.
}

int main() {
   C<int, 1> c;
   c.f();
   cout << c.i << endl;
}
```

Output

```
Base
1
```

See Also

Reference

Templates and Name Resolution

Overload Resolution of Function Template Calls

A function template can overload nontemplate functions of the same name. In this scenario, function calls are resolved by first using template argument deduction to instantiate the function template with a unique specialization. If template argument deduction fails, the other function overloads are considered to resolve the call. These other overloads, also known as the candidate set, include nontemplate functions and other instantiated function templates. If template argument deduction succeeds, then the generated function is compared with the other functions to determine the best match, following the rules for overload resolution. For more information, see Overloading and Argument Matching.

Example

If a nontemplate function is an equally good match to a template function, the nontemplate function is chosen (unless the template arguments were explicitly specified), as in the call `f(1, 1)` in the following example.

```
// template_name_resolution9.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

void f(int, int) { cout << "f(int, int)" << endl; }
void f(char, char) { cout << "f(char, char)" << endl; }

template <class T1, class T2>
void f(T1, T2)
{
   cout << "void f(T1, T2)" << endl;
};

int main()
{
   f(1, 1);   // Equally good match; choose the nontemplate function.
   f('a', 1); // Chooses the template function.
   f<int, int>(2, 2);  // Template arguments explicitly specified.
}
```

Output

```
f(int, int)
void f(T1, T2)
void f(T1, T2)
```

The next example illustrates that the exactly matching template function is preferred if the nontemplate function requires a conversion.

```
// template_name_resolution10.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

void f(int, int) { cout << "f(int, int)" << endl; }

template <class T1, class T2>
void f(T1, T2)
{
   cout << "void f(T1, T2)" << endl;
};

int main()
{
   long l = 0;
   int i = 0;
   // Call the template function f(long, int) because f(int, int)
   // would require a conversion from long to int.
   f(l, i);
}
```

Output

```
void f(T1, T2)
```

See Also

Reference

Templates and Name Resolution
Overloading of Function Templates
typename
Template Argument Deduction

Template Argument Deduction

Template arguments are deduced when a call is made to a template function, but some or all template arguments are omitted. The compiler will attempt to deduce the intended template arguments. In most cases, this works as expected. If it does not, a compilation error occurs, in which case you should specify the template arguments explicitly.

The Visual C++ implementation of template argument deduction conforms to the C++ Standard. For the exact rules on template argument deduction, refer to the ISO Standard.

See Also

Reference

Templates and Name Resolution

Class Templates

You can use class templates to create a family of classes that operate on a type. Class templates are parameterized types. They imply that a separate class could be created for each conceivable value of the parameters (known as template arguments) passed in.

Template arguments can be types or constant values of a specified type. For example:

```
// class_templates.cpp
template <class T, int i> class TempClass
{
public:
    TempClass( void );
    ~TempClass( void );
    int MemberSet( T a, int b );
private:
    T Tarray[i];
    int arraysize;
};

int main()
{
}
```

In this example, the templated class uses two parameters, a type `T` and an int `i`. The `T` parameter can be passed any type, including structures and classes. The `i` parameter has to be passed an integer constant. Because `i` is a constant defined at compile time, you can define a member array of size `i` using a standard array declaration.

For more information, see:

- Members of Class Templates
- Templates for Class Members
- Member Functions of Template Classes
- Nested Class Templates
- Class Template Instantiation
- Explicit Specialization of Class Templates
- Partial Specialization of Class Templates
- Default Arguments for Class Templates
- Template Friends

See Also

Other Resources

Templates

Members of Class Templates

Members of class templates are just like members of any class. They can be static or nonstatic, data or function members, or even other templates. They can be defined either inside the template class or outside of it. The members of a template class can refer to the unknown types specified in the template argument list as if they were valid type names, and they can refer to the unknown object values specified in the template argument list as if they were constant expressions.

When members of templated classes are defined outside of the class declaration, they must be declared differently than those of nontemplated classes. The declaration must be preceded by syntax specifying the template class that the function is a member of.

```
template < template-argument-list > definition
```

Remarks

The declarator for a member function outside the template class must also specify the template arguments.

```
template-name < template-argument-list > :: identifier
```

Example

```
// members_of_class_templates1.cpp
// compile with: /c
template <class T, int i>
class TempClass {
   int MemberSet(T, int);
};

template <class T, int i>
int TempClass< T, i >::MemberSet( T a, int b ) {
   if( ( b >= 0 ) && (b < i) ) {
      Tarray[b++] = a;
      return sizeof( a );
   }
   else
      return -1;
}
```

C++ also allows nested templates, referred to as member templates. Member templates can be nested class templates or member function templates.

See Also

Reference

Class Templates

Templates for Class Members

When creating an out-of-line definition for a member of a template class, the template parameters should be specified on the type name and not on the member name.

Example

```
// templates_for_class_members.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
template <class T>
struct X {
   X();
   void Test();
   static const int i;
};

template <class T>
   X< T >::X() {
      cout << "X created." << endl;
}

template <class T>
   void X< T >::Test() {
      cout << "In Test." << endl;
}
template <class T>
   const int X<T>::i = 9;

int main() {
   X<int> x;
   x.Test();
   cout << X<int>::i << endl;
}
```

Output

```
X created.
In Test.
9
```

See Also

Reference

Class Templates

Member Functions of Template Classes

Member functions can be defined inside or outside of a class template. They are defined like function templates if defined outside the class template.

Example

```
// member_function_templates1.cpp
template<class T, int i> class MyStack
{
    T*  pStack;
    T StackBuffer[i];
    static const int cItems = i * sizeof(T);
public:
    MyStack( void );
    void push( const T item );
    T& pop( void );
};

template< class T, int i > MyStack< T, i >::MyStack( void )
{
};

template< class T, int i > void MyStack< T, i >::push( const T item )
{
};

template< class T, int i > T& MyStack< T, i >::pop( void )
{
};

int main()
{
}
```

Note that just as with any template class member function, the definition of the class's constructor member function includes the template argument list twice.

```
// member_templates.cpp
template<typename T>
class X
{
public:
   template<typename U>
   void mf(const U &u);
};

template<typename T> template <typename U>
void X<T>::mf(const U &u)
{
}

int main()
{
}
```

Description

Member functions can themselves be function templates, specifying additional parameters, as in the following example.

See Also

Reference

Class Templates

Nested Class Templates

Templates can be defined within classes or class templates, in which case they are referred to as member templates. Member templates that are classes are referred to as nested class templates. Member templates that are functions are discussed in Member Function Templates.

Nested class templates are declared as class templates inside the scope of the outer class. They can be defined inside or outside of the enclosing class.

Example

The following code demonstrates a nested class template inside an ordinary class.

```
// nested_class_template1.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class X
{

   template <class T>
   struct Y
   {
      T m_t;
      Y(T t): m_t(t) { }
   };

   Y<int> yInt;
   Y<char> yChar;

public:
   X(int i, char c) : yInt(i), yChar(c) { }
   void print()
   {
      cout << yInt.m_t << " " << yChar.m_t << endl;
   }
};

int main()
{
   X x(1, 'a');
   x.print();
}
```

When nested class templates are defined outside of their enclosing class, they must be prefaced by the template parameters for both the class template (if they are members of a class template) and template parameters for the member template.

```
// nested_class_template2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
class X
{
   template <class U> class Y
   {
      U* u;
   public:
      Y();
      U& Value();
      void print();
      ~Y();
   };
```

```
   Y<int> y;
public:
   X(T t) { y.Value() = t; }
   void print() { y.print(); }
};

template <class T>
template <class U>
X<T>::Y<U>::Y()
{
   cout << "X<T>::Y<U>::Y()" << endl;
   u = new U();
}

template <class T>
template <class U>
U& X<T>::Y<U>::Value()
{
   return *u;
}

template <class T>
template <class U>
void X<T>::Y<U>::print()
{
   cout << this->Value() << endl;
}

template <class T>
template <class U>
X<T>::Y<U>::~Y()
{
   cout << "X<T>::Y<U>::~Y()" << endl;
   delete u;
}

int main()
{
   X<int>* xi = new X<int>(10);
   X<char>* xc = new X<char>('c');
   xi->print();
   xc->print();
   delete xi;
   delete xc;
}
```

Output

```
1 a
```

Output

```
X<T>::Y<U>::Y()
X<T>::Y<U>::Y()
10
99
X<T>::Y<U>::~Y()
X<T>::Y<U>::~Y()
```

Local classes are not allowed to have member templates.

See Also

Reference

Class Templates

Member Function Templates

Class Template Instantiation

Unlike function templates, when instantiating a class template, you must explicitly instantiate the class by giving the arguments for the class templates. To create an instance of `TempClass`:

```
TempClass< float, 6 > test1;       // OK
TempClass< char, items++ > test2;  // Error, second parameter
                                   // must be constant.
```

No code is generated for a templated class or function until it is instantiated. Moreover, member functions are instantiated only if they are called. This can cause problems if you are building a library with templates for other users. For more information, see Explicit Instantiation.

A class template is first specialized and then instantiated by the compiler. The compiler does not instantiate the class template until a reference to a member of this template class is made, sizeof is used on the class, or an instance is created.

An explicit specialization of a template is a customized version of the template for a particular type.

```
// template_instantiation2.cpp
template<class T> class X
{
};

// Explicit specialization of X with 'int'
template<> class X<int>
{
};
int main()
{
}
```

Explicit specialization without `template<>` will produce an error. Versions of Visual C++ before Visual C++ .NET 2003 accepted this syntax:

```
// Explicit specialization of X with 'char'
class X<char> // C2906
{
};
```

If a template is explicitly instantiated or specialized once, it cannot be explicitly instantiated or specialized a second time in the same program for the same set of template arguments. Further specializations with different template arguments are allowed.

If there is not an explicit instantiation or specialization for a particular template argument, the template will be implicitly instantiated at the point where it is first used.

Because different instantiations of the same template are different types, static members for one specialization or instantiation of a template are separate from static members for a different specialization or instantiation of the same template.

See Also

Reference

Class Templates

Explicit Specialization of Class Templates

Class templates can be specialized for specific types or values of the template arguments. Specialization allows template code to be customized for a specific argument type or value. Without specialization, the same code is generated for each type used in a template instantiation. In a specialization, when the specific types are used, the definition for the specialization is used instead of the original template definition. A specialization has the same name as the template of which it is a specialization. However, a template specialization can be different in many ways from the original template. For example, it can have different data members and member functions.

Use specialization to customize a template for a specific type or value. Use partial specialization when the template has more than one template argument and you only need to specialize one of them, or when you want to specialize behavior for an entire set of types, such as all pointer types, reference types, or array types. For more information, see Partial Specialization of Class Templates.

Example

```
// explicit_specialization1.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

// Template class declaration and definition
template <class T> class Formatter
{
   T* m_t;
public:
   Formatter(T* t) : m_t(t) { }
   void print()
   {
      cout << *m_t << endl;
   }
};

// Specialization of template class for type char*
template<> class Formatter<char*>
{
   char** m_t;
public:
   Formatter(char** t) : m_t(t) { }
   void print()
   {
      cout << "Char value: " << **m_t << endl;
   }
};

int main()
{
   int i = 157;
   // Use the generic template with int as the argument.
   Formatter<int>* formatter1 = new Formatter<int>(&i);

   char str[10] = "string1";
   char* str1 = str;
   // Use the specialized template.
   Formatter<char*>* formatter2 = new Formatter<char*>(&str1);

   formatter1->print();
   formatter2->print();
}
```

Output

```
157
Char value: s
```

See Also

Reference

Class Template Instantiation

Partial Specialization of Class Templates (C++)

Partial Specialization of Class Templates (C++)

Class templates can be partially specialized, and the resulting class is still a template. Partial specialization allows template code to be partially customized for specific types in situations, such as:

- A template has multiple types and only some of them need to be specialized. The result is a template parameterized on the remaining types.
- A template has only one type, but a specialization is needed for pointer, reference, pointer to member, or function pointer types. The specialization itself is still a template on the type pointed to or referenced.

Example

```
// partial_specialization_of_class_templates.cpp
template <class T> struct PTS {
   enum {
      IsPointer = 0,
      IsPointerToDataMember = 0
   };
};

template <class T> struct PTS<T*> {
   enum {
      IsPointer = 1,
      IsPointerToDataMember = 0
   };
};

template <class T, class U> struct PTS<T U::*> {
   enum {
      IsPointer = 0,
      IsPointerToDataMember = 1
   };
};

struct S{};

extern "C" int printf_s(const char*,...);

int main() {
   S s, *pS;
   int S::*ptm;
   printf_s("PTS<S>::IsPointer == %d PTS<S>::IsPointerToDataMember == %d\n",
           PTS<S>::IsPointer, PTS<S>:: IsPointerToDataMember);
   printf_s("PTS<S*>::IsPointer == %d PTS<S*>::IsPointerToDataMember ==%d\n"
           , PTS<S*>::IsPointer, PTS<S*>:: IsPointerToDataMember);
   printf_s("PTS<int S::*>::IsPointer == %d PTS"
           "<int S::*>::IsPointerToDataMember == %d\n",
           PTS<int S::*>::IsPointer, PTS<int S::*>::
           IsPointerToDataMember);
}
```

Output

```
PTS<S>::IsPointer == 0 PTS<S>::IsPointerToDataMember == 0
PTS<S*>::IsPointer == 1 PTS<S*>::IsPointerToDataMember ==0
PTS<int S::*>::IsPointer == 0 PTS<int S::*>::IsPointerToDataMember == 1
```

If you have a template collection class that takes any type **T**, you can create a partial specialization that takes any pointer type **T***. The following code demonstrates a collection class template `Bag` and a partial specialization for pointer types in which the collection dereferences the pointer types before copying them to the array. The collection then stores the values that are pointed to. With the original template, only the pointers themselves would have been stored in the collection, leaving the data vulnerable to deletion or modification. In this special pointer version of the collection, code to check for a null pointer in the `add` method is added.

```
// partial_specialization_of_class_templates2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

// Original template collection class.
template <class T> class Bag {
   T* elem;
   int size;
   int max_size;

public:
   Bag() : elem(0), size(0), max_size(1) {}
   void add(T t) {
      T* tmp;
      if (size + 1 >= max_size) {
         max_size *= 2;
         tmp = new T [max_size];
         for (int i = 0; i < size; i++)
            tmp[i] = elem[i];
         tmp[size++] = t;
         delete[] elem;
         elem = tmp;
      }
      else
         elem[size++] = t;
   }

   void print() {
      for (int i = 0; i < size; i++)
         cout << elem[i] << " ";
      cout << endl;
   }
};

// Template partial specialization for pointer types.
// The collection has been modified to check for NULL
// and store types pointed to.
template <class T> class Bag<T*> {
   T* elem;
   int size;
   int max_size;

public:
   Bag() : elem(0), size(0), max_size(1) {}
   void add(T* t) {
      T* tmp;
      if (t == NULL) {   // Check for NULL
         cout << "Null pointer!" << endl;
         return;
      }

      if (size + 1 >= max_size) {
         max_size *= 2;
         tmp = new T [max_size];
         for (int i = 0; i < size; i++)
            tmp[i] = elem[i];
         tmp[size++] = *t;  // Dereference
         delete[] elem;
         elem = tmp;
      }
      else
         elem[size++] = *t; // Dereference
   }

   void print() {
      for (int i = 0; i < size; i++)
         cout << elem[i] << " ";
```

```
      cout << endl;
   }
};

int main() {
   Bag<int> xi;
   Bag<char> xc;
   Bag<int*> xp; // Uses partial specialization for pointer types.

   xi.add(10);
   xi.add(9);
   xi.add(8);
   xi.print();

   xc.add('a');
   xc.add('b');
   xc.add('c');
   xc.print();

   int i = 3, j = 87, *p = new int[2];
   *p = 8;
   *(p + 1) = 100;
   xp.add(&i);
   xp.add(&j);
   xp.add(p);
   xp.add(p + 1);
   p = NULL;
   xp.add(p);
   xp.print();
}
```

Output

```
10 9 8
a b c
Null pointer!
3 87 8 100
```

The following example defines a template class that takes pairs of any two types and then defines a partial specialization of that template class specialized so that one of the types is **int**. The specialization defines an additional sort method that implements a simple bubble sort based on the integer.

```
// partial_specialization_of_class_templates3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class Key, class Value> class Dictionary {
   Key* keys;
   Value* values;
   int size;
   int max_size;
public:
   Dictionary(int initial_size) :  size(0) {
      max_size = 1;
      while (initial_size >= max_size)
         max_size *= 2;
      keys = new Key[max_size];
      values = new Value[max_size];
   }
   void add(Key key, Value value) {
      Key* tmpKey;
      Value* tmpVal;
      if (size + 1 >= max_size) {
         max_size *= 2;
         tmpKey = new Key [max_size];
         tmpVal = new Value [max_size];
         for (int i = 0; i < size; i++) {
```

```
            tmpKey[i] = keys[i];
            tmpVal[i] = values[i];
         }
         tmpKey[size] = key;
         tmpVal[size] = value;
         delete[] keys;
         delete[] values;
         keys = tmpKey;
         values = tmpVal;
      }
      else {
         keys[size] = key;
         values[size] = value;
      }
      size++;
   }

   void print() {
      for (int i = 0; i < size; i++)
         cout << "{" << keys[i] << ", " << values[i] << "}" << endl;
   }
};

// Template partial specialization: Key is specified to be int.
template <class Value> class Dictionary<int, Value> {
   int* keys;
   Value* values;
   int size;
   int max_size;
public:
   Dictionary(int initial_size) :  size(0) {
      max_size = 1;
      while (initial_size >= max_size)
         max_size *= 2;
      keys = new int[max_size];
      values = new Value[max_size];
   }
   void add(int key, Value value) {
      int* tmpKey;
      Value* tmpVal;
      if (size + 1 >= max_size) {
         max_size *= 2;
         tmpKey = new int [max_size];
         tmpVal = new Value [max_size];
         for (int i = 0; i < size; i++) {
            tmpKey[i] = keys[i];
            tmpVal[i] = values[i];
         }
         tmpKey[size] = key;
         tmpVal[size] = value;
         delete[] keys;
         delete[] values;
         keys = tmpKey;
         values = tmpVal;
      }
      else {
         keys[size] = key;
         values[size] = value;
      }
      size++;
   }

   void sort() {
      // Sort method is defined.
      int smallest = 0;
      for (int i = 0; i < size - 1; i++) {
         for (int j = i; j < size; j++) {
            if (keys[j] < keys[smallest])
```

```
                smallest = j;
            }
            swap(keys[i], keys[smallest]);
            swap(values[i], values[smallest]);
        }
    }

    void print() {
        for (int i = 0; i < size; i++)
            cout << "{" << keys[i] << ", " << values[i] << "}" << endl;
    }
};

int main() {
    Dictionary<char*, char*>* dict = new Dictionary<char*, char*>(10);
    dict->print();
    dict->add("apple", "fruit");
    dict->add("banana", "fruit");
    dict->add("dog", "animal");
    dict->print();

    Dictionary<int, char*>* dict_specialized = new Dictionary<int, char*>(10);
    dict_specialized->print();
    dict_specialized->add(100, "apple");
    dict_specialized->add(101, "banana");
    dict_specialized->add(103, "dog");
    dict_specialized->add(89, "cat");
    dict_specialized->print();
    dict_specialized->sort();
    cout << endl << "Sorted list:" << endl;
    dict_specialized->print();
}
```

Output

```
{apple, fruit}
{banana, fruit}
{dog, animal}
{100, apple}
{101, banana}
{103, dog}
{89, cat}

Sorted list:
{89, cat}
{100, apple}
{101, banana}
{103, dog}
```

See Also

Reference

Explicit Specialization of Class Templates

Default Arguments for Class Templates

Class templates can have default arguments for type or value parameters. Specify default arguments with the equal (**=**) sign followed by the type name or value. For multiple template arguments, all arguments after the first default argument must have default arguments. When declaring a template class object with default arguments, omit the arguments to accept the default argument. If there are no nondefault arguments, do not omit the empty angle brackets.

A template that is multiply declared cannot specify a default argument more than once. The following code demonstrates an error:

```
template <class T = long> class A;
template <class T = long> class A { /* . . . */ }; // Generates C4348.
```

Example

In the following example, an array class template is defined with a default type **int** for the array element and a default value for the template parameter specifying the size.

```
// template_default_arg.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T = int, int size = 10> class Array
{
   T* array;
public:
   Array()
   {
      array = new T[size];
      memset(array, 0, size * sizeof(T));
   }
   T& operator[](int i)
   {
      return *(array + i);
   }
   const int Length() { return size; }
   void print()
   {
      for (int i = 0; i < size; i++)
      {
         cout << (*this)[i] << " ";
      }
      cout << endl;
   }
};

int main()
{
   // Explicitly specify the template arguments:
   Array<char, 26> ac;
   for (int i = 0; i < ac.Length(); i++)
   {
      ac[i] = 'A' + i;
   }
   ac.print();

   // Accept the default template arguments:
   Array<> a; // You must include the angle brackets.
   for (int i = 0; i < a.Length(); i++)
   {
      a[i] = i*10;
   }
   a.print();
```

```
}
```

Output

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 10 20 30 40 50 60 70 80 90
```

See Also

Reference

Default Arguments

Template Friends

Class templates can have friends. A class or class template, function, or function template can be a friend to a template class. Friends can also be specializations of a class template or function template, but not partial specializations.

Example

In the following example, a friend function is defined as a function template within the class template. This code produces a version of the friend function for every instantiation of the template. This construct is useful if your friend function depends on the same template parameters as the class does.

```
// template_friend1.cpp
// compile with: /EHsc

#include <iostream>
using namespace std;

template <class T> class Array {
   T* array;
   int size;

public:
   Array(int sz): size(sz) {
      array = new T[size];
      memset(array, 0, size * sizeof(T));
   }

   Array(const Array& a) {
      size = a.size;
      array = new T[size];
      memcpy_s(array, a.array, sizeof(T));
   }

   T& operator[](int i) {
      return *(array + i);
   }

   int Length() { return size; }

   void print() {
      for (int i = 0; i < size; i++)
         cout << *(array + i) << " ";

      cout << endl;
   }

   template<class T>
   friend Array<T>* combine(Array<T>& a1, Array<T>& a2);
};

template<class T>
Array<T>* combine(Array<T>& a1, Array<T>& a2) {
   Array<T>* a = new Array<T>(a1.size + a2.size);
   for (int i = 0; i < a1.size; i++)
      (*a)[i] = *(a1.array + i);

   for (int i = 0; i < a2.size; i++)
      (*a)[i + a1.size] = *(a2.array + i);

   return a;
}

int main() {
   Array<char> alpha1(26);
   for (int i = 0 ; i < alpha1.Length() ; i++)
      alpha1[i] = 'A' + i;
```

```
   alpha1.print();

   Array<char> alpha2(26);
   for (int i = 0 ; i < alpha2.Length() ; i++)
      alpha2[i] = 'a' + i;

   alpha2.print();
   Array<char>*alpha3 = combine(alpha1, alpha2);
   alpha3->print();
   delete alpha3;
}
```

Output

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r
s t u v w x y z
```

The next example involves a friend that has a template specialization. A function template specialization is automatically a friend if the original function template is a friend.

It is also possible to declare only the specialized version of the template as the friend, as the comment before the friend declaration in the following code indicates. If you do this, you must put the definition of the friend template specialization outside of the template class.

```
// template_friend2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
class Array;

template <class T>
void f(Array<T>& a);

template <class T> class Array
{
    T* array;
    int size;

public:
    Array(int sz): size(sz)
    {
        array = new T[size];
        memset(array, 0, size * sizeof(T));
    }
    Array(const Array& a)
    {
        size = a.size;
        array = new T[size];
        memcpy_s(array, a.array, sizeof(T));
    }
    T& operator[](int i)
    {
        return *(array + i);
    }
    int Length()
    {
        return size;
    }
    void print()
    {
        for (int i = 0; i < size; i++)
        {
            cout << *(array + i) << " ";
```

```
        }
        cout << endl;
    }
    // If you replace the friend declaration with the int-specific
    // version, only the int specialization will be a friend.
    // The code in the generic f will fail
    // with C2248: 'Array<T>::size' :
    // cannot access private member declared in class 'Array<T>'.
    //friend void f<int>(Array<int>& a);
    friend void f<>(Array<T>& a);
};

// f function template, friend of Array<T>
template <class T>
void f(Array<T>& a)
{
    cout << a.size << " generic" << endl;
}

// Specialization of f for int arrays
// will be a friend because the template f is a friend.
template<> void f(Array<int>& a)
{
    cout << a.size << " int" << endl;
}

int main()
{
    Array<char> ac(10);
    f(ac);

    Array<int> a(10);
    f(a);
}
```

Output

```
10 generic
10 int
```

The next example shows a friend class template declared within a class template. The class template is then used as the template argument for the friend class. Friend class templates must be defined outside of the class template in which they are declared. Any specializations or partial specializations of the friend template are also friends of the original class template.

```
// template_friend3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
class X
{
private:
   T* data;
   void InitData(int seed) { data = new T(seed); }
public:
   void print() { cout << *data << endl; }
   template <class U> friend class Factory;
};

template <class U>
class Factory
{
public:
   U* GetNewObject(int seed)
   {
      U* pu = new U;
      pu->InitData(seed);
```

```
        return pu;
    }
};

int main()
{
    Factory< X<int> > XintFactory;
    X<int>* x1 = XintFactory.GetNewObject(65);
    X<int>* x2 = XintFactory.GetNewObject(97);

    Factory< X<char> > XcharFactory;
    X<char>* x3 = XcharFactory.GetNewObject(65);
    X<char>* x4 = XcharFactory.GetNewObject(97);
    x1->print();
    x2->print();
    x3->print();
    x4->print();
}
```

Output

```
65
97
A
a
```

See Also

Reference

Default Arguments

Function Templates

Class templates define a family of related classes that are based on the type arguments passed to the class upon instantiation. Function templates are similar to class templates but define a family of functions. With function templates, you can specify a set of functions that are based on the same code but act on different types or classes. The following function template swaps two items:

```
// function_templates1.cpp
template< class T > void MySwap( T& a, T& b ) {
   T c(a);
   a = b;
   b = c;
}
int main() {
}
```

This code defines a family of functions that swap the values of the arguments. From this template, you can generate functions that will swap **int** and **long** types and also user-defined types. `MySwap` will even swap classes if the class's copy constructor and assignment operator are properly defined.

In addition, the function template will prevent you from swapping objects of different types, because the compiler knows the types of the `a` and `b` parameters at compile time.

Although this function could be performed by a nontemplated function, using void pointers, the template version is typesafe. Consider the following calls:

```
int j = 10;
int k = 18;
CString Hello = "Hello, Windows!";
MySwap( j, k );          //OK
MySwap( j, Hello );      //error
```

The second `MySwap` call triggers a compile-time error, because the compiler cannot generate a `MySwap` function with parameters of different types. If void pointers were used, both function calls would compile correctly, but the function would not work properly at run time.

Explicit specification of the template arguments for a function template is allowed. For example:

```
// function_templates2.cpp
template<class T> void f(T) {}
int main(int j) {
   f<char>(j);   // Generate the specialization f(char).
   // If not explicitly specified, f(int) would be deduced.
}
```

When the template argument is explicitly specified, normal implicit conversions are done to convert the function argument to the type of the corresponding function template parameters. In the above example, the compiler will convert (`char j`) to type `int`.

See Also

Reference

Function Template Instantiation

Explicit Instantiation

Explicit Specialization of Function Templates

Other Resources

Templates

Function Template Instantiation

When a function template is first called for each type, the compiler creates an instantiation. Each instantiation is a version of the templated function specialized for the type. This instantiation will be called every time the function is used for the type. If you have several identical instantiations, even in different modules, only one copy of the instantiation will end up in the executable file.

Conversion of function arguments is allowed in function templates for any argument and parameter pair where the parameter does not depend on a template argument. For example:

```
template<class T> void f(T, int){ };

int i; char c;
f(i, c);
```

In this example, a conversion of `i` to type `T` can cause an error. However, the conversion of (`char c`) to `int` is allowed.

Function templates can be explicitly instantiated by declaring the template with a particular type as an argument. For example, the following code is allowed:

```
// function_template_instantiation.cpp
template<class T> void f(T) { }

// Instantiate f with the explicitly specified template.
// argument 'int'
//
template void f<int> (int);

// Instantiate f with the deduced template argument 'char'.
template void f(char);
int main()
{
}
```

See Also

Reference

Function Templates

Explicit Instantiation

Explicit instantiation lets you create an instantiation of a templated class or function without actually using it in your code. Because this is useful when you are creating library (.lib) files that use templates for distribution, uninstantiated template definitions are not put into object (.obj) files.

The following explicitly instantiates `MyStack` for **int** variables and six items:

```
template class MyStack<int, 6>;
```

This statement creates an instantiation of `MyStack` without reserving any storage for an object; code is generated for all members.

The following explicitly instantiates only the constructor member function:

```
template MyStack<int, 6>::MyStack( void );
```

Function templates can be explicitly instantiated by redeclaring them with a specific type argument, as seen in the example in Function Template Instantiation.

Microsoft Specific

You can use the **extern** keyword to prevent the automatic instantiation of members. For example:

```
extern template class MyStack<int, 6>;
```

Similarly, you can mark specific members as being external and not instantiated as follows:

```
extern template MyStack<int, 6>::MyStack( void );
```

Note
The **extern** keyword in the specialization only applies to member functions defined outside of the body of the class. Functions defined inside the class declaration are considered inline functions and are always instantiated.

END Microsoft Specific

See Also

Reference

Function Templates

Explicit Specialization of Function Templates

With a function template, you can define special behavior for a specific type by providing an explicit specialization (override) of the function template for that type. For example:

```
template<> void MySwap(double a, double b);
```

This declaration enables you to define a different function for **double** variables. Like non-template functions, standard type conversions (such as promoting a variable of type **float** to **double**) are applied.

Example

```
// explicit_specialization.cpp
template<class T> void f(T t)
{
};

// Explicit specialization of f with 'char' with the
// template argument explicitly specified:
//
template<> void f<char>(char c)
{
}

// Explicit specialization of f with 'double' with the
// template argument deduced:
//
template<> void f(double d)
{
}
int main()
{
}
```

See Also

Reference

Function Templates

Partial Ordering of Function Templates (C++)

Multiple function templates that match the argument list of a function call can be available. C++ defines a partial ordering of function templates to specify which function should be called. The ordering is partial because there can be some templates that are considered equally specialized.

The compiler chooses the most specialized template function available from the possible matches. For example, if a function template takes a type **T**, and another function template taking **T*** is available, the **T*** version is said to be more specialized and is preferred over the generic **T** version whenever the argument is a pointer type, even though both would be allowable matches.

Use the following process to determine if one function template candidate is more specialized:

1. Consider two function templates, T1 and T2.
2. Replace the parameters in T1 with a hypothetical unique type X.
3. With the parameter list in T1, see if T2 is a valid template for that parameter list. Ignore any implicit conversions.
4. Repeat the same process with T1 and T2 reversed.
5. If one template is a valid template argument list for the other template, but the converse is not true, then that template is considered to be less specialized than the other template. If both templates using the previous step form valid arguments for each other, then they are considered to be equally specialized, and an ambiguous call will result when attempting to use them.
6. Using these rules:
   a. A template specialization for a specific type is more specialized than one taking a generic type argument.
   b. A template taking only **T*** is more specialized than one taking only **T**, because a hypothetical type **X*** is a valid argument for a **T** template argument, but **X** is not a valid argument for a **T*** template argument.
   c. **const T** is more specialized than **T**, because **const X** is a valid argument for a **T** template argument, but **X** is not a valid argument for a **const T** template argument.
   d. **const T*** is more specialized than **T***, because **const X*** is a valid argument for a **T*** template argument, but **X*** is not a valid argument for a **const T*** template argument.
7. The following sample works in Visual C++ .NET 2003 as specified in the standard:

```
// partial_ordering_of_function_templates.cpp
// compile with: /EHsc
#include <iostream>

extern "C" int printf(const char*,...);
template <class T> void f(T) {
   printf_s("Less specialized function called\n");
}

template <class T> void f(T*) {
   printf_s("More specialized function called\n");
}

template <class T> void f(const T*) {
   printf_s("Even more specialized function for const T*\n");
}

int main() {
   int i =0;
   const int j = 0;
   int *pi = &i;
   const int *cpi = &j;

   f(i);   // Calls less specialized function.
   f(pi);  // Calls more specialized function.
```

```
    f(cpi); // Calls even more specialized function.
    // Without partial ordering, these calls would be ambiguous.
  }
```

Output

```
Less specialized function called
More specialized function called
Even more specialized function for const T*
```

See Also

Reference

Function Templates

Member Function Templates

The term member template refers to both member function templates and nested class templates. Member function templates are template functions that are members of a class or class template. For more information, see Nested Class Templates.

Member functions can be function templates in several contexts. All functions of class templates are generic but are not referred to as member templates or member function templates. If these member functions take their own template arguments, they are considered to be member function templates. For details, see Member Functions of Template Classes.

Example

Member function templates of nontemplate or template classes are declared as function templates with their template parameters.

```
// member_function_templates.cpp
struct X
{
   template <class T> void mf(T* t) {}
};

int main()
{
   int i;
   X* x = new X();
   x->mf(&i);
}
```

The following example shows a member function template of a template class.

```
// member_function_templates2.cpp
template<typename T>
class X
{
public:
   template<typename U>
   void mf(const U &u)
   {
   }
};

int main()
{
}
```

Additionally, in Visual Studio .NET 2003 and later, member templates can also be defined outside of a class.

```
// defining_member_templates_outside_class.cpp
template<typename T>
class X
{
public:
   template<typename U>
   void mf(const U &u);
};

template<typename T> template <typename U>
void X<T>::mf(const U &u)
{
}

int main()
{
}
```

Local classes are not allowed to have member templates.

Member template functions cannot be virtual functions and cannot override virtual functions from a base class when they are declared with the same name as a base class virtual function.

Visual C++ .NET 2003 introduced support for templated user-defined conversions. The following sample works in Visual C++ .NET 2003 as specified in the standard.

```
// templated_user_defined_conversions.cpp
template <class T>
struct S
{
   template <class U> operator S<U>()
   {
      return S<U>();
   }
};

int main()
{
   S<int> s1;
   S<long> s2 = s1;  // Convert s1 using UDC and copy constructs S<long>.
}
```

See Also

Reference

Function Templates

Referencing Templates [C++ Language]

This topic shows how to use a template that has been previously declared.

```
template-name < template-arg-list >
```

Remarks

The *template-arg-list* should be a comma-separated list of:

```
        expression
        type-name
```

All *expression* arguments must be constant expressions. The compiler creates a new instance (called an instantiation) of the templated class or function if there is no exact match to a previously generated template. For example, to reference the `MyStack` class defined in Member Function Templates:

```
MyStack< unsigned long, 5 > stack1;
// Creates a stack of unsigned longs.
MyStack< DWORD, 5 >stack2;
// Uses code created above.
MyStack< char, 6 > stack3;
// Generates new code.
MyStack< MyClass, 6 > stack4;
// Generates stack of MyClass objects.
```

Each generated function template creates its own static variables and members.

All template arguments must be accessible at the point where they are used.

The exception to the above syntax rule is in identifying a member template specialization in an expression after the **::**, **.** or **->** operators. After these operators, the keyword **template** can be specified. Visual C++ departs from the standard in that the **template** keyword is always optional in this context, whereas the standard requires it in some circumstances. The template keyword cannot be used in the specialization unless following these operators.

[**::** | **->** | **.**] **template** *template-name* **<** *template-arg-list* **>**

For example, the following specifies a call to the `int` specialization of the member function template `f<T>(int)` which is a member of the class `X` and passes it the parameter `10`.

```
X::template f<int>(10);
```

See Also

Other Resources

Templates

Event Handling

Event handling is primarily supported for COM classes (C++ classes that implement COM objects, typically using ATL classes or the coclass attribute). For more information, see Event Handling in COM.

Event handling is also supported for native C++ classes (C++ classes that do not implement COM objects), however, that support is deprecated and will be removed in a future release. For more information, see Event Handling in Native C++.

Event handling supports single- and multithreaded usage and protects data from simultaneous multithread access. It also allows you to derive subclasses from event source or receiver classes and support extended event sourcing/receiving in the derived class.

Visual C++ includes attributes and keywords for declaring events and event handlers. The event attributes and keywords can be used in CLR programs and in native C++ programs.

Topic	Description
event_source	Creates an event source.
event_receiver	Creates an event receiver (sink).
__event	Declares an event.
__raise	Emphasizes the call site of an event.
__hook	Associates a handler method with an event.
__unhook	Dissociates a handler method from an event.

See Also

Reference

C++ Keywords

Other Resources

C++ Language Reference

Event Handling Samples

__event

Declares an event.

```
__event method-declarator;
__event __interface interface-specifier;
__event member-declarator;
```

Remarks

The keyword **__event** can be applied to a method declaration, an interface declaration, or a data member declaration.

Depending on whether your event source and receiver are native C++, COM, or managed (.NET Framework), you can use the following constructs as events:

Native C++	COM	Managed (.NET Framework)
Method	—	method
—	interface	—
—	—	data member

Use __hook in an event receiver to associate a handler method with an event method. Note that after you create an event with the **__event** keyword, all event handlers subsequently hooked to that event will be called when the event is called.

An **__event** method declaration cannot have a definition; a definition is implicitly generated, so the event method can be called as if it were any ordinary method.

Note
A templated class or struct cannot contain events.

Native Events

Native events are methods. The return type is typically **HRESULT** or **void**, but can be any integral type, including an **enum**. When an event uses an integral return type, an error condition is defined when an event handler returns a nonzero value, in which case the event being raised will call the other delegates.

```
// Examples of native C++ events:
__event void OnDblClick();
__event HRESULT OnClick(int* b, char* s);
```

See Event Handling in Native C++ for sample code.

COM Events

COM events are interfaces. The parameters of a method in an event source interface should be **in** parameters (but this is not rigorously enforced), because an **out** parameter is not useful when multicasting. A level 1 warning will be issued if you use an **out** parameter.

The return type is typically **HRESULT** or **void**, but can be any integral type, including **enum**. When an event uses an integral return type and an event handler returns a nonzero value, it is an error condition, in which case the event being raised aborts calls to the other delegates. Note that the compiler will automatically mark an event source interface as a source in the generated IDL.

The __interface keyword is always required after **__event** for a COM event source.

```
// Example of a COM event:
__event __interface IEvent1;
```

See Event Handling in COM for sample code.

Managed Events

For information on coding events in the new syntax, see event (Visual C++).

Managed events are data members or methods. When used with an event, the return type of a delegate must be compliant with the Common Language Specification. The return type of the event handler must match the return type of the delegate. For more information on delegates, see __delegate. If a managed event is a data member, its type must be a pointer to a delegate.

In the .NET Framework, you can treat a data member as if it were a method itself (that is, the **Invoke** method of its corresponding delegate). You must predefine the delegate type for declaring a managed event data member. In contrast, a managed event method implicitly defines the corresponding managed delegate if it is not already defined. For example, you can declare an event value such as `OnClick` as an event as follows:

```
// Examples of managed events:
__event ClickEventHandler* OnClick;  // data member as event
__event void OnClick(String* s);  // method as event
```

When implicitly declaring a managed event, you can specify add and remove accessors that will be called when event handlers are added or removed. You can also define the method that calls (raises) the event from outside the class.

Example: Native Events

```
// EventHandling_Native_Event.cpp
// compile with: /c
[event_source(native)]
class CSource {
public:
   __event void MyEvent(int nValue);
};
```

Example: COM Events

```
// EventHandling_COM_Event.cpp
// compile with: /c
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>

[ module(dll, name="EventSource", uuid="6E46B59E-89C3-4c15-A6D8-B8A1CEC98830") ];

[ dual, uuid("00000000-0000-0000-0000-000000000002") ]
__interface IEventSource {
   [id(1)] HRESULT MyEvent();
};
 [ coclass, uuid("00000000-0000-0000-0000-000000000003"),  event_source(com) ]
class CSource : public IEventSource {
public:
   __event __interface IEventSource;
   HRESULT FireEvent() {
      __raise MyEvent();
      return S_OK;
   }
};
```

Example: Managed Events

```
// EventHandling_Managed_Event.cpp
// compile with: /clr:oldSyntax /c
using namespace System;
[event_source(managed)]
public __gc class CPSource {

public:
   __event void MyEvent(Int16 nValue);
};
```

When applying an attribute to an event, you can specify that the attribute apply to either the generated methods or to the Invoke method of the generated delegate. The default (`event:`) is to apply the attribute to the event.

```
// EventHandling_Managed_Event_2.cpp
// compile with: /clr:oldSyntax /c
using namespace System;
[attribute(All, AllowMultiple=true)]
public __gc class Attr {};

public __delegate void D();

public __gc class X {
public:
   [method:Attr] __event D* E;
   [returnvalue:Attr] __event void noE();
};
```

See Also

Reference

C++ Keywords
Event Handling
event_source
event_receiver
__hook
__unhook
__raise

__hook

Associates a handler method with an event.

```
long __hook(
   &SourceClass::EventMethod,
      source,
   &ReceiverClass::HandlerMethod
   [, receiver = this]
);
long __hook(
      interface,
      source
);
```

Parameters

& *SourceClass* **::** *EventMethod*

A pointer to the event method to which you hook the event handler method:

- Native C++ events: *SourceClass* is the event source class and *EventMethod* is the event.
- COM events: *SourceClass* is the event source interface and *EventMethod* is one of its methods.
- Managed events: *SourceClass* is the event source class and *EventMethod* is the event.

interface

The interface name being hooked to *receiver*, only for COM event receivers in which the *layout_dependent* parameter of the event_receiver attribute is **true**.

source

A pointer to an instance of the event source. Depending on the code *type* specified in **event_receiver**, *source* can be one of the following:

- A native event source object pointer.
- An **IUnknown**-based pointer (COM source).
- A managed object pointer (for managed events).

& *ReceiverClass* **::** *HandlerMethod*

A pointer to the event handler method to be hooked to an event. The handler is specified as a method of a class or a reference to the same; if you do not specify the class name, **__hook** assumes the class to be that in which it is called.

- Native C++ events: *ReceiverClass* is the event receiver class and *HandlerMethod* is the handler.
- COM events: *ReceiverClass* is the event receiver interface and *HandlerMethod* is one of its handlers.
- Managed events: *ReceiverClass* is the event receiver class and *HandlerMethod* is the handler.

receiver(optional)

A pointer to an instance of the event receiver class. If you do not specify a receiver, the default is the receiver class or structure in which **__hook** is called.

Usage

Can be use in any function scope, including main, outside the event receiver class.

Remarks

Use the intrinsic function **__hook** in an event receiver to associate or hook a handler method with an event method. The specified handler is then called when the source raises the specified event. You can hook several handlers to a single event or hook several events to a single handler.

There are two forms of **__hook**. You can use the first (four-argument) form in most cases, specifically, for COM event receivers in which the *layout_dependent* parameter of the event_receiver attribute is **false**.

In these cases you do not need to hook all methods in an interface before firing an event on one of the methods; only the method handling the event needs to be hooked. You can use the second (two-argument) form of **__hook** only for a COM event receiver in which *layout_dependent*=**true**.

__hook returns a long value. A nonzero return value indicates that an error has occurred (managed events throw an exception).

The compiler checks for the existence of an event and that the event signature agrees with the delegate signature.

With the exception of COM events, **__hook** and **__unhook** can be called outside the event receiver.

An alternative to using **__hook** is to use the += operator.

For information on coding managed events in the new syntax, see event (Visual C++).

Note
A templated class or struct cannot contain events.

Example

See Event Handling in Native C++ and Event Handling in COM for samples.

See Also

Reference

C++ Keywords
Event Handling
event_source
event_receiver
__unhook
__raise

__raise

Emphasizes the call site of an event.

```
__raise method-declarator;
```

Remarks

From managed code, an event can only be raised from within the class where it is defined. See event (Visual C++) for more information.

The keyword **__raise** causes an error to be emitted if you call a non-event.

Note
A templated class or struct cannot contain events.

Example

```
// EventHandlingRef_raise.cpp
struct E {
   __event void func1();
   void func1(int) {}

   void func2() {}

   void b() {
      __raise func1();
      __raise func1(1);  // C3745: 'int Event::bar(int)':
                         // only an event can be 'raised'
      __raise func2();   // C3745
   }
};

int main() {
   E e;
   __raise e.func1();
   __raise e.func1(1);  // C3745
   __raise e.func2();   // C3745
}
```

See Also

Reference

C++ Keywords

Event Handling

Concepts

New Language Features in Visual C++

__unhook

Dissociates a handler method from an event.

```
long  __unhook(
   &SourceClass::EventMethod,
      source,
   &ReceiverClass::HandlerMethod
   [, receiver = this]
);
long  __unhook(
      interface,
      source
);
long  __unhook(
      source
);
```

Parameters

& *SourceClass* **::** *EventMethod*

A pointer to the event method from which you unhook the event handler method:

- Native C++ events: *SourceClass* is the event source class and *EventMethod* is the event.
- COM events: *SourceClass* is the event source interface and *EventMethod* is one of its methods.
- Managed events: *SourceClass* is the event source class and *EventMethod* is the event.

interface

The interface name being unhooked from *receiver*, only for COM event receivers in which the *layout_dependent* parameter of the event_receiver attribute is **true**.

source

A pointer to an instance of the event source. Depending on the code *type* specified in **event_receiver**, *source* can be one of the following:

- A native event source object pointer.
- An **IUnknown**-based pointer (COM source).
- A managed object pointer (for managed events).

& *ReceiverClass* **::** *HandlerMethod*

A pointer to the event handler method to be unhooked from an event. The handler is specified as a method of a class or a reference to the same; if you do not specify the class name, **__unhook** assumes the class to be that in which it is called.

- Native C++ events: *ReceiverClass* is the event receiver class and *HandlerMethod* is the handler.
- COM events: *ReceiverClass* is the event receiver interface and *HandlerMethod* is one of its handlers.
- Managed events: *ReceiverClass* is the event receiver class and *HandlerMethod* is the handler.

receiver(optional)

A pointer to an instance of the event receiver class. If you do not specify a receiver, the default is the receiver class or structure in which **__unhook** is called.

Usage

Can be use in any function scope, including main, outside the event receiver class.

Remarks

Use the intrinsic function **__unhook** in an event receiver to dissociate or "unhook" a handler method from an event method.

There are three forms of **__unhook**. You can use the first (four-argument) form in most cases. You can use the second (two-argument) form of **__unhook** only for a COM event receiver; this unhooks the entire event interface. You can use the third (one-argument) form to unhook all delegates from the specified source.

A nonzero return value indicates that an error has occurred (managed events will throw an exception).

If you call **__unhook** on an event and event handler that are not already hooked, it will have no effect.

At compile time, the compiler verifies that the event exists and does parameter type checking with the specified handler.

With the exception of COM events, **__hook** and **__unhook** can be called outside the event receiver.

An alternative to using **__unhook** is to use the -= operator.

For information on coding managed events in the new syntax, see event (Visual C++).

Note
A templated class or struct cannot contain events.

Example

See Event Handling in Native C++ and Event Handling in COM for samples.

See Also

Reference

C++ Keywords
event_source
event_receiver
__event
__hook
__raise

Event Handling in Native C++

In native C++ event handling, you set up an event source and event receiver using the event_source and event_receiver attributes, respectively, specifying *type*=**native**. These attributes allow the classes to which they are applied to fire events and handle events in a native, non-COM context.

Declaring Events

In an event source class, use the __event keyword on a method declaration to declare the method as an event. Make sure to declare the method, but do not define it; to do so will generate a compiler error, because the compiler defines the method implicitly when it is made into an event. Native events can be methods with zero or more parameters. The return type can be void or any integral type.

Defining Event Handlers

In an event receiver class, you define event handlers, which are methods with signatures (return types, calling conventions, and arguments) that match the event that they will handle.

Hooking Event Handlers to Events

Also in an event receiver class, you use the intrinsic function __hook to associate events with event handlers and __unhook to dissociate events from event handlers. You can hook several events to an event handler, or several event handlers to an event.

Firing Events

To fire an event, simply call the method declared as an event in the event source class. If handlers have been hooked to the event, the handlers will be called.

Native C++ Event Code

The following example shows how to fire an event in native C++. To compile and run the example, refer to the comments in the code.

Example

```
// evh_native.cpp
#include <stdio.h>

[event_source(native)]
class CSource {
public:
   __event void MyEvent(int nValue);
};

[event_receiver(native)]
class CReceiver {
public:
   void MyHandler1(int nValue) {
      printf_s("MyHandler1 was called with value %d.\n", nValue);
   }

   void MyHandler2(int nValue) {
      printf_s("MyHandler2 was called with value %d.\n", nValue);
   }

   void hookEvent(CSource* pSource) {
      __hook(&CSource::MyEvent, pSource, &CReceiver::MyHandler1);
      __hook(&CSource::MyEvent, pSource, &CReceiver::MyHandler2);
   }

   void unhookEvent(CSource* pSource) {
      __unhook(&CSource::MyEvent, pSource, &CReceiver::MyHandler1);
      __unhook(&CSource::MyEvent, pSource, &CReceiver::MyHandler2);
   }
};
```

```
int main() {
   CSource source;
   CReceiver receiver;

   receiver.hookEvent(&source);
   __raise source.MyEvent(123);
   receiver.unhookEvent(&source);
}
```

Output

```
MyHandler2 was called with value 123.
MyHandler1 was called with value 123.
```

See Also

Reference

Event Handling

Event Handling in COM

In COM event handling, you set up an event source and event receiver using the event_source and event_receiver attributes, respectively, specifying *type*=**com**. These attributes inject the appropriate code for custom, dispatch, and dual interfaces to allow the classes to which they are applied to fire events and handle events through COM connection points.

Declaring Events

In an event source class, use the __event keyword on an interface declaration to declare that interface's methods as events. The events of that interface are fired when you call them as interface methods. Methods on event interfaces can have zero or more parameters (which should all be **in** parameters). The return type can be void or any integral type.

Defining Event Handlers

In an event receiver class, you define event handlers, which are methods with signatures (return types, calling conventions, and arguments) that match the event that they will handle. For COM events, calling conventions do not have to match; see Layout Dependent COM Events below for details.

Hooking Event Handlers to Events

Also in an event receiver class, you use the intrinsic function __hook to associate events with event handlers and __unhook to dissociate events from event handlers. You can hook several events to an event handler, or several event handlers to an event.

Note
Typically, there are two techniques to allow a COM event receiver to access event source interface definitions. The first, as sh own below, is to share a common header file. The second is to use #import with the **embedded_idl** import qualifier, so that the event source type library is written to the .tlh file with the attribute-generated code preserved.

Firing Events

To fire an event, simply call a method in the interface declared with the **__event** keyword in the event source class. If handlers have been hooked to the event, the handlers will be called.

COM Event Code

The following example shows how to fire an event in a COM class. To compile and run the example, refer to the comments in the code.

```
// evh_server.h
#pragma once

[ dual, uuid("00000000-0000-0000-0000-000000000001") ]
__interface IEvents {
   [id(1)] HRESULT MyEvent([in] int value);
};

[ dual, uuid("00000000-0000-0000-0000-000000000002") ]
__interface IEventSource {
   [id(1)] HRESULT FireEvent();
};

class DECLSPEC_UUID("530DF3AD-6936-3214-A83B-27B63C7997C4") CSource;
```

And then the server:

```
// evh_server.cpp
// compile with: /LD
// post-build command: Regsvr32.exe /s evh_server.dll
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include "evh_server.h"
```

```
[ module(dll, name="EventSource", uuid="6E46B59E-89C3-4c15-A6D8-B8A1CEC98830") ];

[coclass, event_source(com), uuid("530DF3AD-6936-3214-A83B-27B63C7997C4")]
class CSource : public IEventSource {
public:
   __event __interface IEvents;

   HRESULT FireEvent() {
      __raise MyEvent(123);
      return S_OK;
   }
};
```

And then the client:

```
// evh_client.cpp
// compile with: /link /OPT:NOREF
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include <stdio.h>
#include "evh_server.h"

[ module(name="EventReceiver") ];

[ event_receiver(com) ]
class CReceiver {
public:
   HRESULT MyHandler1(int nValue) {
      printf_s("MyHandler1 was called with value %d.\n", nValue);
      return S_OK;
   }

   HRESULT MyHandler2(int nValue) {
      printf_s("MyHandler2 was called with value %d.\n", nValue);
      return S_OK;
   }

   void HookEvent(IEventSource* pSource) {
      __hook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler1);
      __hook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler2);
   }

   void UnhookEvent(IEventSource* pSource) {
      __unhook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler1);
      __unhook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler2);
   }
};

int main() {
   // Create COM object
   CoInitialize(NULL);
   {
      IEventSource* pSource = 0;
      HRESULT hr = CoCreateInstance(__uuidof(CSource), NULL,         CLSCTX_ALL, __uuidof(I
EventSource), (void **) &pSource);
      if (FAILED(hr)) {
         return -1;
      }

      // Create receiver and fire event
      CReceiver receiver;
      receiver.HookEvent(pSource);
      pSource->FireEvent();
      receiver.UnhookEvent(pSource);
   }
```

```
   CoUninitialize();
   return 0;
}
```

Output

```
MyHandler1 was called with value 123.
MyHandler2 was called with value 123.
```

Layout Dependent COM Events

Layout dependency is only an issue for COM programming. In native and managed event handling, the signatures (return type, calling convention, and arguments) of the handlers must match their events, but the handler names do not have to match their events.

However, in COM event handling, when you set the *layout_dependent* parameter of **event_receiver** to **true**, the name and signature matching is enforced. This means that the names and signatures of the handlers in the event receiver must exactly match the names and signatures of the events to which they are hooked.

When *layout_dependent* is set to **false**, the calling convention and storage class (virtual, static, and so on) can be mixed and matched between the firing event method and the hooking methods (its delegates). It is slightly more efficient to have *layout_dependent*=**true**.

For example, suppose `IEventSource` is defined to have the following methods:

```
[id(1)] HRESULT MyEvent1([in] int value);
[id(2)] HRESULT MyEvent2([in] int value);
```

Assume the event source has the following form:

```
[coclass, event_source(com)]
class CSource : public IEventSource {
public:
   __event __interface IEvents;

   HRESULT FireEvent() {
      MyEvent1(123);
      MyEvent2(123);
      return S_OK;
   }
};
```

Then, in the event receiver, any handler hooked to a method in `IEventSource` must match its name and signature, as follows:

```
[coclass, event_receiver(com, true)]
class CReceiver {
public:
   HRESULT MyEvent1(int nValue) {  // name and signature matches MyEvent1
      ...
   }
   HRESULT MyEvent2(E c, char* pc) {  // signature doesn't match MyEvent2
      ...
   }
   HRESULT MyHandler1(int nValue) {  // name doesn't match MyEvent1 (or 2)
      ...
   }
   void HookEvent(IEventSource* pSource) {
      __hook(IFace, pSource);  // Hooks up all name-matched events
                               // under layout_dependent = true
      __hook(&IFace::MyEvent1, pSource, &CReceive::MyEvent1);   // valid
      __hook(&IFace::MyEvent2, pSource, &CSink::MyEvent2);   // not valid
      __hook(&IFace::MyEvent1, pSource, &CSink:: MyHandler1); // not valid
   }
};
```

See Also

Reference

Event Handling

Microsoft-Specific Modifiers

This section describes Microsoft-specific extensions to C++ in the following areas:

- Based addressing, the practice of using a pointer as a base from which other pointers can be offset
- Function calling conventions
- Extended storage-class attributes declared with the __declspec keyword
- The __w64 keyword
- The __sptr and __uptr keywords used for pointer conversion on 64-bit platforms.

Many of the Microsoft-specific keywords can be used to modify declarators to form derived types. For more information about declarators, see Declarators.

Microsoft-Specific Keywords

Keyword	Meaning	Used to Form Derived Types?
__based	The name that follows declares a 32-bit offset to the 32-bit base contained in the declaration.	Yes
__cdecl	The name that follows uses the C naming and calling conventions.	Yes
__declspec	The name that follows specifies a Microsoft-specific storage-class attribute.	No
__fastcall	The name that follows declares a function that uses registers, when available, instead of the stack for argument passing.	Yes
__restrict	Similar to __declspec(restrict), but for use on variables.	No
__sptr	Specifies that pointer conversion extends the high-order bit of a 32-bit pointer throughout the high-order portion of a 64-bit pointer.	No
__stdcall	The name that follows specifies a function that observes the standard calling convention.	Yes
__uptr	Specifies that pointer conversion sets the high-order portion of a 64-bit pointer to zero.	No
__w64	Marks a date type as being larger on a 64-bit compiler.	No
__unaligned	Specifies that a pointer to a type or other data is not aligned.	No

See Also

Other Resources

C++ Language Reference

Based Addressing

This section includes the following topics:

- __based Grammar
- Based Pointers

See Also

Reference

Microsoft-Specific Modifiers

__based Grammar

Microsoft Specific

Based addressing is useful when you need precise control over the segment in which objects are allocated (static and dynamic based data).

The only form of based addressing acceptable in 32-bit and 64-bit compilations is "based on a pointer" that defines a type that contains a 32-bit or 64-bit displacement to a 32-bit or 64-bit base or based on **void**.

Grammar
based-range-modifier:

__based(*base-expression* **)**

base-expression:

based-variablebased-abstract-declaratorsegment-namesegment-cast

based-variable:

identifier

based-abstract-declarator:

abstract-declarator

base-type:

type-name

END Microsoft Specific
See Also
Reference
Based Pointers (C++)

Based Pointers (C++)

Microsoft Specific

The **__based** keyword allows you to declare pointers based on pointers (pointers that are offsets from existing pointers).

```
type __based( base ) declarator
```

Remarks

Pointers based on pointer addresses are the only form of the **__based** keyword valid in 32-bit or 64-bit compilations. For the Microsoft 32-bit C/C++ compiler, a based pointer is a 32-bit offset from a 32-bit pointer base. A similar restriction holds for 64-bit environments, where a based pointer is a 64-bit offset from the 64-bit base.

One use for pointers based on pointers is for persistent identifiers that contain pointers. A linked list that consists of pointers based on a pointer can be saved to disk, then reloaded to another place in memory, with the pointers remaining valid. For example:

```
// based_pointers1.cpp
// compile with: /c
void *vpBuffer;
struct llist_t {
   void __based( vpBuffer ) *vpData;
   struct llist_t __based( vpBuffer ) *llNext;
};
```

The pointer `vpBuffer` is assigned the address of memory allocated at some later point in the program. The linked list is relocated relative to the value of `vpBuffer`.

Note

Persisting identifiers containing pointers can also be accomplished by using memory-mapped files.

When dereferencing a based pointer, the base must be either explicitly specified or implicitly known through the declaration.

For compatibility with previous versions, **_based** is a synonym for **__based**.

Example

The following code demonstrates changing a based pointer by changing its base.

```
// based_pointers2.cpp
// compile with: /EHsc
#include <iostream>

int a1[] = { 1,2,3 };
int a2[] = { 10,11,12 };
int *pBased;

typedef int __based(pBased) * pBasedPtr;

using namespace std;
int main() {
   pBased = &a1[0];
   pBasedPtr pb = 0;

   cout << *pb << endl;
   cout << *(pb+1) << endl;

   pBased = &a2[0];

   cout << *pb << endl;
```

```
    cout << *(pb+1) << endl;
}
```

Output

```
1
2
10
11
```

See Also

Reference

C++ Keywords

alloc_text

Calling Conventions

The Visual C/C++ compiler provides several different conventions for calling internal and external functions. Understanding these different approaches can help you debug your program and link your code with assembly-language routines.

The topics on this subject explain the differences between the calling conventions, how arguments are passed, and how values are returned by functions. They also discuss naked function calls, an advanced feature that enables you to write your own prolog and epilog code.

For information on calling conventions for x64 processors, see Calling Convention.

Topics in this section

- Argument Passing and Naming Conventions (**__cdecl**, **__stdcall**, **__fastcall**, and others)
- Calling Example: Function Prototype and Call
- Using naked function calls to write custom prolog/epilog code
- Floating Point Coprocessor and Calling Conventions
- Obsolete calling conventions

See Also
Reference
Microsoft-Specific Modifiers

Argument Passing and Naming Conventions

Microsoft Specific

All arguments are widened to 32 bits when they are passed. Return values are also widened to 32 bits and returned in the EAX register, except for 8-byte structures, which are returned in the EDX:EAX register pair. Larger structures are returned in the EAX register as pointers to hidden return structures. Parameters are pushed onto the stack from right to left. Structures that are not PODs will not be returned in registers.

The compiler generates prolog and epilog code to save and restore the ESI, EDI, EBX, and EBP registers, if they are used in the function.

Note
When a struct, union, or class is returned from a function by value, all definitions of the type need to be the same, else the program may fail at runtime.

For information on how to define your own function prolog and epilog code, see Naked Function Calls.

The following calling conventions are supported by the Visual C/C++ compiler.

Keyword	Stack cleanup	Parameter passing
__cdecl	Caller	Pushes parameters on the stack, in reverse order (right to left)
__clrcall	n/a	Load parameters onto CLR expression stack in order (left to right).
__stdcall	Callee	Pushes parameters on the stack, in reverse order (right to left)
__fastcall	Callee	Stored in registers, then pushed on stack
__thiscall	Callee	Pushed on stack; **this** pointer stored in ECX

For related information, see Obsolete Calling Conventions.

END Specific

See Also

Reference

Calling Conventions

__cdecl

Microsoft Specific

This is the default calling convention for C and C++ programs. Because the stack is cleaned up by the caller, it can do **vararg** functions. The **__cdecl** calling convention creates larger executables than __stdcall, because it requires each function call to include stack cleanup code. The following list shows the implementation of this calling convention.

Element	Implementation
Argument-passing order	Right to left
Stack-maintenance responsibility	Calling function pops the arguments from the stack
Name-decoration convention	Underscore character (_) is prefixed to names, except when exporting __cdecl functions that use C linkage.
Case-translation convention	No case translation performed

Note

For related information, see Decorated Names.

On Itanium Processor Family (IPF) and x64 processors, **__cdecl** is accepted and ignored by the compiler; on IPF, by convention, parameters are passed in register.

Place the **__cdecl** modifier before a variable or a function name. Because the C naming and calling conventions are the default, the only time you need to use **__cdecl** is when you have specified the /Gz (stdcall) or /Gr (fastcall) compiler option. The /Gd compiler option forces the **__cdecl** calling convention.

For non-static class functions, if the function is defined out-of-line, the calling convention modifier does not have to be specified on the out-of-line definition. That is, for class non-static member methods, the calling convention specified during declaration is assumed at the point of definition. Given this class definition,

```
struct CMyClass {
   void __cdecl mymethod();
};
```

this

```
void CMyClass::mymethod() { return; }
```

is equivalent to this

```
void __cdecl CMyClass::mymethod() { return; }
```

Example

In the following example, the compiler is instructed to use C naming and calling conventions for the `system` function:

```
// Example of the __cdecl keyword on function
int __cdecl system(const char *);
// Example of the __cdecl keyword on function pointer
typedef BOOL (__cdecl *funcname_ptr)(void * arg1, const char * arg2, DWORD flags, ...);
```

See Also

Reference

Argument Passing and Naming Conventions

__clrcall

Microsoft Specific

Specifies that a function can only be called from managed code. Use **__clrcall** for all virtual functions that will only be called from managed code. However this calling convention cannot be used for functions that will be called from native code.

Use **__clrcall** to improve performance when calling from a managed function to a virtual managed function or from managed function to managed function through pointer.

Entry points are separate, compiler-generated functions. If a function has both native and managed entry points, one of them will be the actual function with the function implementation. The other function will be a separate function (a thunk) that calls into the actual function and lets the common language runtime perform PInvoke. When marking a function as **__clrcall**, you indicate the function implementation must be MSIL and that the native entry point function will not be generated.

When taking the address of a native function if **__clrcall** is not specified, the compiler uses the native entry point. **__clrcall** indicates that the function is managed and there is no need to go through the transition from managed to native. In that case the compiler uses the managed entry point.

When **/clr** (not **/clr:pure** or **/clr:safe**) is used and **__clrcall** is not used, taking the address of a function always returns the address of the native entry point function. When **__clrcall** is used, the native entry point function is not created, so you get the address of the managed function, not an entry point thunk function. For more information, see Double Thunking (C++).

/clr (Common Language Runtime Compilation) implies that all functions and function pointers are **__clrcall** and the compiler will not permit a function inside the compiland to be marked anything other than **__clrcall**. When **/clr:pure** is used, **__clrcall** can only be specified on function pointers and external declarations.

You can directly call **__clrcall** functions from legacy C++ code compiled with **/clr** as long as that function has an MSIL implementation. **__clrcall** functions cannot be called directly from functions that have inline asm and call CPU-specific intrinisics, for example, even if those functions are compiled with **/clr**.

__clrcall function pointers are only meant to be used in the application domain in which they were created. Instead of passing **__clrcall** function pointers across application domains, use CrossAppDomainDelegate. For more information, see Application Domains and Visual C++.

Example

```
// clrcall.cpp
// compile with: /clr:oldSyntax /LD
void __clrcall Test1( ) {}
void (__clrcall *fpTest1)( ) = &Test1;
```

Note that when a function is declared with **__clrcall**, code will be generated when needed; for example, when function is called.

```
// clrcall2.cpp
// compile with: /clr
using namespace System;
int __clrcall Func1() {
   Console::WriteLine("in Func1");
   return 0;
}

// Func1 hasn't been used at this point (code has not been generated),
// so runtime returns the adddress of a stub to the function
int (__clrcall *pf)() = &Func1;

// code calls the function, code generated at difference address
int i = pf();   // comment this line and comparison will pass

int main() {
   if (&Func1 == pf)
      Console::WriteLine("&Func1 == pf, comparison succeeds");
   else
      Console::WriteLine("&Func1 != pf, comparison fails");

   // even though comparison fails, stub and function call are correct
```

```
   pf();
   Func1();
}
```

Output

```
in Func1
&Func1 != pf, comparison fails
in Func1
in Func1
```

The following sample shows that you can define a function pointer, such that, you declare that the function pointer will only be invoked from managed code. This allows the compiler to directly call the managed function and avoid the native entry point (double thunk issue).

```
// clrcall3.cpp
// compile with: /clr
void Test() {
   System::Console::WriteLine("in Test");
}

int main() {
   void (*pTest)() = &Test;
   (*pTest)();

   void (__clrcall *pTest2)() = &Test;
   (*pTest2)();
}
```

See Also

Reference

Argument Passing and Naming Conventions

C++ Keywords

__stdcall

Microsoft Specific

The **__stdcall** calling convention is used to call Win32 API functions. The callee cleans the stack, so the compiler makes **vararg** functions **__cdecl**. Functions that use this calling convention require a function prototype.

```
return-type __stdcall function-name[(argument-list)]
```

Remarks

The following list shows the implementation of this calling convention.

Element	Implementation
Argument-passing order	Right to left.
Argument-passing convention	By value, unless a pointer or reference type is passed.
Stack-maintenance responsibility	Called function pops its own arguments from the stack.
Name-decoration convention	An underscore (_) is prefixed to the name. The name is followed by the at sign (@) followed by the number of bytes (in decimal) in the argument list. Therefore, the function declared as `int func( int a, double b )` is decorated as follows: `_func@12`
Case-translation convention	None

The /Gz compiler option specifies **__stdcall** for all functions not explicitly declared with a different calling convention.

Functions declared using the **__stdcall** modifier return values the same way as functions declared using __cdecl.

On Itanium Processor Family (IPF) and x64 processors, **__stdcall** is accepted and ignored by the compiler; on IPF, by convention, parameters are passed in register.

For non-static class functions, if the function is defined out-of-line, the calling convention modifier does not have to be specified on the out-of-line definition. That is, for class non-static member methods, the calling convention specified during declaration is assumed at the point of definition. Given this class definition,

```
struct CMyClass {
   void __stdcall mymethod();
};
```

this

```
void CMyClass::mymethod() { return; }
```

is equivalent to this

```
void __stdcall CMyClass::mymethod() { return; }
```

Example

In the following example, use of **__stdcall** results in all `WINAPI` function types being handled as a standard call:

```
// Example of the __stdcall keyword
#define WINAPI __stdcall
// Example of the __stdcall keyword on function pointer
typedef BOOL (__stdcall *funcname_ptr)(void * arg1, const char * arg2, DWORD flags, ...);
```

See Also

Reference

Argument Passing and Naming Conventions

C++ Keywords

__fastcall

Microsoft Specific

The **__fastcall** calling convention specifies that arguments to functions are to be passed in registers, when possible. The following list shows the implementation of this calling convention.

Element	Implementation
Argument-passing order	The first two DWORD or smaller arguments are passed in ECX and EDX registers; all other arguments are passed right to left.
Stack-maintenance responsibility	Called function pops the arguments from the stack.
Name-decoration convention	At sign (@) is prefixed to names; an at sign followed by the number of bytes (in decimal) in the parameter list is suffixed to names.
Case-translation convention	No case translation performed.

Note
Future compiler versions may use different registers to store parameters.

Using the /Gr compiler option causes each function in the module to compile as fastcall unless the function is declared with a conflicting attribute, or the name of the function is `main`.

On Itanium Processor Family (IPF) and AMD64 machines, **__fastcall** is accepted and ignored by the compiler; on an IPF chip, by convention, parameters are passed in register.

For non-static class functions, if the function is defined out-of-line, the calling convention modifier does not have to be specified on the out-of-line definition. That is, for class non-static member methods, the calling convention specified during declaration is assumed at the point of definition. Given this class definition,

```
struct CMyClass {
   void __fastcall mymethod();
};
```

this

```
void CMyClass::mymethod() { return; }
```

is equivalent to this

```
void __fastcall CMyClass::mymethod() { return; }
```

Example

In the following example, the function named `DeleteAggrWrapper` is passed arguments in registers:

```
// Example of the __fastcall keyword
#define FASTCALL    __fastcall

void FASTCALL DeleteAggrWrapper(void* pWrapper);
// Example of the __ fastcall keyword on function pointer
typedef BOOL (__fastcall *funcname_ptr)(void * arg1, const char * arg2, DWORD flags, ...);
```

END Microsoft Specific

See Also

Reference

Argument Passing and Naming Conventions

C++ Keywords

__thiscall

Microsoft Specific

The **__thiscall** calling convention is used on member functions and is the default calling convention used by C++ member functions that do not use variable arguments. Under **__thiscall**, the callee cleans the stack, which is impossible for **vararg** functions. Arguments are pushed on the stack from right to left, with the **this** pointer being passed via register ECX, and not on the stack, on the x86 architecture.

One reason to use **__thiscall** is in classes whose member functions use **__clrcall** by default. In that case, you can use **__thiscall** to make individual member functions callable from native code.

When compiling with /clr:pure, all functions and function pointers are **__clrcall** unless specified otherwise.

In releases before Visual C++ 2005, the thiscall calling convention could not be explicitly specified in a program, because **thiscall** was not a keyword.

vararg member functions use the **__cdecl** calling convention. All function arguments are pushed on the stack, with the **this** pointer placed on the stack last

Because this calling convention applies only to C++, there is no C name decoration scheme.

On Itanium Processor Family (IPF) and x64 machines, **__thiscall** is accepted and ignored by the compiler; on an IPF chip, by convention, parameters are passed in register.

For non-static class functions, if the function is defined out-of-line, the calling convention modifier does not have to be specified on the out-of-line definition. That is, for class non-static member methods, the calling convention specified during declaration is assumed at the point of definition.

Example

```
// thiscall_cc.cpp
// compile with: /c /clr:oldSyntax
struct CMyClass {
   void __thiscall mymethod();
   void __clrcall mymethod2();
};
```

END Microsoft Specific

See Also

Reference

Argument Passing and Naming Conventions

Calling Example: Function Prototype and Call

Microsoft Specific

The following example shows the results of making a function call using various calling conventions.

This example is based on the following function skeleton. Replace *calltype* with the appropriate calling convention.

```
void    calltype MyFunc( char c, short s, int i, double f );
.
.
.
void    MyFunc( char c, short s, int i, double f )
    {
    .
    .
    .
    }
.
.
.
MyFunc ('x', 12, 8192, 2.7183);
```

For more information, see Results of Calling Example.

END Microsoft Specific

See Also

Reference

Calling Conventions

Results of Calling Example

Microsoft Specific
__cdecl

The C decorated function name is "_MyFunc."

The __cdecl calling convention

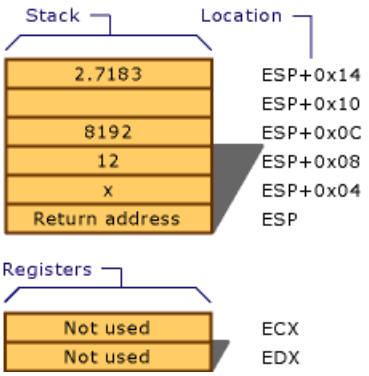

__stdcall and thiscall

The C decorated name (**__stdcall**) is "_MyFunc@20." The C++ decorated name is proprietary.

The __stdcall and thiscall calling conventions

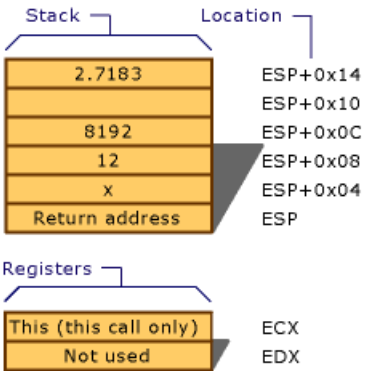

__fastcall

The C decorated name (**__fastcall**) is "@MyFunc@20." The C++ decorated name is proprietary.

The __fastcall calling convention

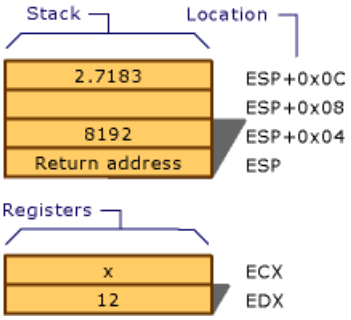

END Microsoft Specific

See Also

Reference

Calling Example: Function Prototype and Call

Naked Function Calls

Microsoft Specific

Functions declared with the **naked** attribute are emitted without prolog or epilog code, enabling you to write your own custom prolog/epilog sequences using the inline assembler. Naked functions are provided as an advanced feature. They enable you to declare a function that is being called from a context other than C/C++, and thus make different assumptions about where parameters are, or which registers are preserved. Examples include routines such as interrupt handlers. This feature is particularly useful for writers of virtual device drivers (VxDs).

What do you want to know more about?

- naked
- Rules and Limitations for Naked Functions
- Considerations for Writing Prolog/Epilog Code

END Microsoft Specific

See Also

Reference

Calling Conventions

Rules and Limitations for Naked Functions

Microsoft Specific

The following rules and limitations apply to naked functions:

- The **return** statement is not permitted.
- Structured Exception Handling and C++ Exception Handling constructs are not permitted because they must unwind across the stack frame.
- For the same reason, any form of **setjmp** is prohibited.
- Use of the **_alloca** function is prohibited.
- To ensure that no initialization code for local variables appears before the prolog sequence, initialized local variables are not permitted at function scope. In particular, the declaration of C++ objects is not permitted at function scope. There may, however, be initialized data in a nested scope.
- Frame pointer optimization (the /Oy compiler option) is not recommended, but it is automatically suppressed for a naked function.
- You cannot declare C++ class objects at the function lexical scope. You can, however, declare objects in a nested block.
- The **naked** keyword is ignored when compiling with /clr.
- For __fastcall naked functions, whenever there is a reference in C/C++ code to one of the register arguments, the prolog code should store the values of that register into the stack location for that variable. For example:

```
// nkdfastcl.cpp
// compile with: /c
// processor: x86
__declspec(naked) int __fastcall  power(int i, int j) {
   // calculates i^j, assumes that j >= 0

   // prolog
   __asm {
      push ebp
      mov ebp, esp
      sub esp, __LOCAL_SIZE
     // store ECX and EDX into stack locations allocated for i and j
     mov i, ecx
     mov j, edx
   }

   {
      int k = 1;   // return value
      while (j-- > 0)
         k *= i;
      __asm {
         mov eax, k
      };
   }

   // epilog
   __asm {
      mov esp, ebp
      pop ebp
      ret
   }
}
```

END Microsoft Specific

See Also

Reference

Naked Function Calls

Considerations for Writing Prolog/Epilog Code

Microsoft Specific

Before writing your own prolog and epilog code sequences, it is important to understand how the stack frame is laid out. It is also useful to know how to use the **__LOCAL_SIZE** symbol.

Stack Frame Layout

This example shows the standard prolog code that might appear in a 32-bit function:

```
push        ebp                 ; Save ebp
mov         ebp, esp            ; Set stack frame pointer
sub         esp, localbytes     ; Allocate space for locals
push        <registers>         ; Save registers
```

The `localbytes` variable represents the number of bytes needed on the stack for local variables, and the `<registers>` variable is a placeholder that represents the list of registers to be saved on the stack. After pushing the registers, you can place any other appropriate data on the stack. The following is the corresponding epilog code:

```
pop         <registers>   ; Restore registers
mov         esp, ebp      ; Restore stack pointer
pop         ebp           ; Restore ebp
ret                       ; Return from function
```

The stack always grows down (from high to low memory addresses). The base pointer (`ebp`) points to the pushed value of `ebp`. The locals area begins at `ebp-2`. To access local variables, calculate an offset from `ebp` by subtracting the appropriate value from `ebp`.

__LOCAL_SIZE

The compiler provides a symbol, **__LOCAL_SIZE**, for use in the inline assembler block of function prolog code. This symbol is used to allocate space for local variables on the stack frame in custom prolog code.

The compiler determines the value of **__LOCAL_SIZE**. Its value is the total number of bytes of all user-defined local variables and compiler-generated temporary variables. **__LOCAL_SIZE** can be used only as an immediate operand; it cannot be used in an expression. You must not change or redefine the value of this symbol. For example:

```
mov        eax, __LOCAL_SIZE           ;Immediate operand--Okay
mov        eax, [ebp - __LOCAL_SIZE]   ;Error
```

The following example of a naked function containing custom prolog and epilog sequences uses the **__LOCAL_SIZE** symbol in the prolog sequence:

```
// the__local_size_symbol.cpp
// processor: x86
__declspec ( naked ) int main() {
   int i;
   int j;

   __asm {      /* prolog */
      push   ebp
      mov      ebp, esp
      sub      esp, __LOCAL_SIZE
      }

   /* Function body */
   __asm {   /* epilog */
      mov      esp, ebp
      pop      ebp
      ret
      }
}
```

END Microsoft Specific

See Also

Reference

Naked Function Calls

Floating Point Coprocessor and Calling Conventions

If you are writing assembly routines for the floating point coprocessor, you must preserve the floating point control word and clean the coprocessor stack unless you are returning a **float** or **double** value (which your function should return in ST(0)).

See Also

Reference

Calling Conventions

Obsolete Calling Conventions

Microsoft Specific

The **__pascal**, **__fortran**, and **__syscall** calling conventions are no longer supported. You can emulate their functionality by using one of the supported calling conventions and appropriate linker options.

WINDOWS.H now supports the **WINAPI** macro, which translates to the appropriate calling convention for the target. Use **WINAPI** where you previously used **PASCAL** or **__far __pascal**.

END Microsoft Specific

See Also

Reference

Argument Passing and Naming Conventions

__declspec

Microsoft Specific

The extended attribute syntax for specifying storage-class information uses the **__declspec** keyword, which specifies that an instance of a given type is to be stored with a Microsoft-specific storage-class attribute listed below. Examples of other storage-class modifiers include the **static** and **extern** keywords. However, these keywords are part of the ANSI specification of the C and C++ languages, and as such are not covered by extended attribute syntax. The extended attribute syntax simplifies and standardizes Microsoft-specific extensions to the C and C++ languages.

Grammar
decl-specifier:

__declspec (*extended-decl-modifier-seq* **)**

extended-decl-modifier-seq:

extended-decl-modifier $_{\text{opt}}$

extended-decl-modifier extended-decl-modifier-seq

extended-decl-modifier:

align(# **)**

allocate(" *segname* **")**

appdomain

deprecated

dllimport

dllexport

jitintrinsic

naked

noalias

noinline

noreturn

nothrow

novtable

process

property({get=*get_func_name***|,put=***put_func_name***})**

restrict

selectany

thread

uuid(" *ComObjectGUID* **")**

White space separates the declaration modifier sequence. Examples appear in later sections.

Extended attribute grammar supports these Microsoft-specific storage-class attributes: align, allocate, appdomain, deprecated, dllexport, dllimport, jitintrinsic, naked, noalias, noinline, noreturn, nothrow, novtable, process, restrict, selectany, and thread. It also supports these COM-object attributes: property and uuid.

The **dllexport**, **dllimport**, **naked**, **noalias**, **nothrow**, **property**, **restrict**, **selectany**, **thread**, and **uuid** storage-class attributes are properties only of the declaration of the object or function to which they are applied. The **thread** attribute affects data and objects only. The **naked** attribute affects functions only. The **dllimport** and **dllexport** attributes affect functions, data, and objects. The **property**, **selectany**, and **uuid** attributes affect COM objects.

The **__declspec** keywords should be placed at the beginning of a simple declaration. The compiler ignores, without warning, any **__declspec** keywords placed after * or & and in front of the variable identifier in a declaration.

A **__declspec** attribute specified in the beginning of a user-defined type declaration applies to the variable of that type. For example:

```
__declspec(dllimport) class X {} varX;
```

In this case, the attribute applies to `varX`. A **__declspec** attribute placed after the **class** or **struct** keyword applies to the user-defined type. For example:

```
class __declspec(dllimport) X {};
```

In this case, the attribute applies to `X`.

The general guideline for using the **__declspec** attribute for simple declarations is as follows:

```
        decl-specifier-seq declarator-list;
```

The *decl-specifier-seq* should contain, among other things, a base type (e.g. **int**, **float**, a **typedef**, or a class name), a storage class (e.g. **static**, **extern**), or the **__declspec** extension. The *init-declarator-list* should contain, among other things, the pointer part of declarations. For example:

```
__declspec(selectany) int * pi1 = 0;   //OK, selectany & int both part of decl-specifier
int __declspec(selectany) * pi2 = 0;   //OK, selectany & int both part of decl-specifier
int * __declspec(selectany) pi3 = 0;   //ERROR, selectany is not part of a declarator
```

The following code declares an integer thread local variable and initializes it with a value:

```
// Example of the __declspec keyword
__declspec( thread ) int tls_i = 1;
```

END Microsoft Specific

See Also

Reference

C++ Keywords

C Extended Storage-Class Attributes

align (C++)

Microsoft Specific

Use **__declspec(align(#))** to precisely control the alignment of user-defined data (for example, static allocations or automatic data in a function).

For more information on alignment, see:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vcconwindowsdataalignmentonipfx86x86-64.asp?frame=true.

```
__declspec( align( # ) ) declarator
```

Remarks

Writing applications that use the latest processor instructions introduces some new constraints and issues. In particular, many new instructions require that data must be aligned to 16-byte boundaries. Additionally, by aligning frequently used data to the cache line size of a specific processor, you improve cache performance. For example, if you define a structure whose size is less than 32 bytes, you may want to align it to 32 bytes to ensure that objects of that structure type are efficiently cached.

is the alignment value. Valid entries are integer powers of two from 1 to 8192 (bytes), such as 2, 4, 8, 16, 32, or 64. *declarator* is the data that you are declaring as aligned.

See __alignof for information on how to return a value of type **size_t** that is the alignment requirement of the type. and __unaligned for information on how to declare unaligned pointers when targeting 64-bit processors.

You can use **__declspec(align(#))** when you define a **struct**, **union**, or **class**, or when you declare a variable.

Without **__declspec(align(#))**, Visual C++ aligns data on natural boundaries based on the size of the data, for example 4-byte integers on 4-byte boundaries and 8-byte doubles on 8-byte boundaries. Data in classes or structures is aligned within the class or structure at the minimum of its natural alignment and the current packing setting (from #pragma **pack** or the **/Zp** compiler option).

You cannot specify alignment for function parameters.

For example:

```
__declspec(align(32)) struct Str1{
   int a, b, c, d, e;
};
```

This type now has a 32-byte alignment attribute, meaning that all instances must start on a 32-byte boundary. Additional structure types declared with this type as a member preserve this type's alignment attribute, that is, any structure with `Str1` as an element will have an alignment attribute of at least 32.

To summarize:

- Unless overridden with **__declspec(align(#))**, the alignment of a scalar structure member is the minimum of its size and the current packing.
- Unless overridden with **__declspec(align(#))**, the alignment of a structure is the maximum of the individual alignments of its member(s).
- A structure member is placed at an offset from the beginning of its parent structure which is the smallest multiple of its alignment greater than or equal to the offset of the end of the previous member.
- The size of a structure is the smallest multiple of its alignment larger greater than or equal the offset of the end of its last member.

Note that `sizeof(struct Str1)` is equal to 32, such that, if an array of Str1 objects is created, and the base of the array is 32-byte aligned, then each member of the array will also be 32-byte aligned. To create an array whose base is properly aligned, use _aligned_malloc, or write your own allocator. Note that normal allocators, such as malloc, C++ operator new, and the Win32 allocators return memory that will most likely not be sufficiently aligned for **__declspec(align(#))** structures or arrays of structures.

The `sizeof` value for any structure is the offset of the final member, plus that member's size, rounded up to the nearest multiple of the largest member alignment value or the whole structure alignment value, whichever is greater.

__declspec(align(#)) can only increase alignment restrictions.

For more information, see:

- align Examples
- Defining New Types with __declspec(align(#))
- Aligning Data in Thread Local Storage
- How align Works with Data Packing
- Structure Alignment Examples (x64 specific)

align Examples

The following examples show how **__declspec(align(#))** affects the size and alignment of data structures. The examples assume the following definitions:

```
#define CACHE_LINE  32
#define CACHE_ALIGN __declspec(align(CACHE_LINE))
```

In the following example, the `S1` structure is defined with **__declspec(align(32))**. All uses of `S1`, whether for a variable definition or other type declarations, ensure that this structure data is 32-byte aligned. `sizeof(struct S1)` returns 32, and `S1` has 16 padding bytes following the 16 bytes required to hold the four integers. Each `int` member requires 4-byte alignment, but the alignment of the structure itself is declared to be 32, so the overall alignment is 32.

```
struct CACHE_ALIGN S1  // cache align all instances of S1{
   int a, b, c, d;
};
struct S1 s1;   // s1 is 32-byte cache aligned
```

In the following example, `sizeof(struct S2)` will return 16, which is exactly the sum of the member sizes, because that happens to be a multiple of the largest alignment requirement (a multiple of 8).

```
__declspec(align(8)) struct S2 {
   int a, b, c, d;
};
```

In the following example, `sizeof(struct S3)` returns 64.

```
struct S3 {
   struct S1 s1;   // S3 inherits cache alignment requirement
                   // from S1 declaration
   int a;          // a is now cache aligned because of s1
                   // 28 bytes of trailing padding
};
```

In the following example, note that `a` has only the alignment of natural type, in this case, 4 bytes. However, `S1` must be 32-byte aligned. Twenty-eight bytes of padding follow `a`, so that `s1` starts at offset 32. `S4` then inherits the alignment requirement of `S1`, because it is the largest alignment requirement in the structure. `sizeof(struct S4)` returns 64.

```
struct S4 {
   int a;
   // 28 bytes padding
    struct S1 s1;      // S4 inherits cache alignment requirement of S1
};
```

The following three variable declarations also use **__declspec(align(#))**. In each case, the variable must be 32-byte aligned. In the case of the array, the base address of the array, not each array member, is 32-byte aligned. The `sizeof` value for each array member is not affected by the use of **__declspec(align(#))**.

```
CACHE_ALIGN int i;
CACHE_ALIGN int array[128];
CACHE_ALIGN struct s2 s;
```

To align each individual member of an array, code such as the following should be used:

```
typedef CACHE_ALIGN struct { int a; } S5;
S5 array[10];
```

In the following example, note that aligning the structure itself and aligning the first element are identical:

```
CACHE_ALIGN struct S6 {
   int a;
   int b;
};

struct S7 {
   CACHE_ALIGN int a;
               int b;
};
```

`S6` and `S7` have identical alignment, allocation, and size characteristics.

In the following example, the alignment of the starting addresses of a, b, c, and d are 4, 1, 4, and 1, respectively.

```
void fn() {
   int a;
   char b;
   long c;
   char d[10]
}
```

The alignment if the memory were allocated on heap depends on which allocation function is called. For example, if you use **malloc**, the result depends on the operand size. If *arg* >= 8, alignment will be 8 byte aligned. If *arg* < 8, alignment will be the first power of 2 less than *arg*. For example, if you use malloc(7), alignment is 4 bytes.

Defining New Types with __declspec(align(#))

You can define a type with an alignment characteristic.

For example, you can define a `struct` with an alignment value as follows:

```
struct aType {int a; int b;};
typedef __declspec(align(32)) struct aType bType;
```

Now, `aType` and `bType` are the same size (8 bytes) but variables of type `bType` will be 32-byte aligned.

Aligning Data in Thread Local Storage

Static thread-local storage (TLS) created with the **__declspec(thread)** attribute and put in the TLS section in the image works for alignment exactly like normal static data. The operating system creates TLS data by allocating data the size of the TLS section and respecting the TLS section alignment attribute.

The following example shows various ways to place aligned data into thread local storage.

```
// put an aligned integer in TLS
__declspec(thread) __declspec(align(32)) int a;

// define an aligned structure and put a variable of the struct type
// into TLS
__declspec(thread) __declspec(align(32)) struct F1 { int a; int b; } a;

// create an aligned structure
struct CACHE_ALIGN S9 {
   int a;
   int b;
};
// put a variable of the structure type into TLS
__declspec(thread) struct S9 a;
```

How align Works with Data Packing

The **/Zp** compiler option and the **pack** pragma have the effect of packing data for structure and union members. This example shows how **/Zp** and **__declspec(align(** # **))** work together:

```
struct S {
   char a;
   short b;
   double c;
   CACHE_ALIGN double d;
   char e;
   double f;
};
```

The following table lists the offset of each member under a variety of **/Zp** (or #pragma **pack**) values, showing how the two interact.

Variable	**/Zp1**	**/Zp2**	**/Zp4**	**/Zp8**
A	0	0	0	0
B	1	2	2	2
C	3	4	4	8
D	32	32	32	32

E	40	40	40	40
F	41	42	44	48
sizeof(S)	64	64	64	64

For more information, see /Zp (Struct Member Alignment).

Thus, the offset of an object is based on the offset of the previous object and the current packing setting, unless the object has a **__declspec(align(#))** attribute, in which case the alignment is based on the offset of the previous object and the **__declspec(align(#))** value for the object.

END Microsoft Specific

See Also

Reference

__declspec

allocate

Microsoft Specific

The **allocate** declaration specifier names a data segment in which the data item will be allocated.

```
__declspec(allocate("segname")) declarator
```

Remarks

The name *segname* must be declared using one of the following pragmas:

- code_seg
- const_seg
- data_seg
- init_seg
- section

Example

```
// allocate.cpp
#pragma section("mycode", read)
__declspec(allocate("mycode"))  int i = 0;

int main() {
}
```

END Microsoft Specific

See Also

Reference

__declspec

C++ Keywords

appdomain

Specifies that each application domain of your managed application should have its own copy of a particular global variable or static member variable. See Application Domains and Application Domains and Visual C++ for more information.

Every application domain has its own copy of a per-appdomain variable. A constructor of an appdomain variable is executed when an assembly is loaded into an application domain, and the destructor is executed when the application domain is unloaded.

If you want all application domains within a process in the common language runtime to share a global variable, use the **__declspec(process)** modifier. **__declspec(process)** is in effect by default under /clr and **__declspec(appdomain)** is in effect by default under **/clr:pure**. **__declspec(appdomain)** is enforced under **/clr:safe**.

__declspec(appdomain) is only valid when one of the **/clr** compiler options is used. Only a global variable, static member variable, or a static local variable can be marked with **__declspec(appdomain)**. It is an error to apply **__declspec(appdomain)** to static members of managed types because they always have this behavior.

Using **__declspec(appdomain)** is similar to using Thread Local Storage (TLS). Threads have their own storage, as do application domains. Using **__declspec(appdomain)** ensures the global variable has its own storage in each application domain created for this application.

There are limitations to mixing the use of per process and per appdomain variables; see process for more information.

For example, at program start up, all per-process variables are initialized, then all per-appdomain variables are initialized. Therefore when a per-process variable is being initialized, it cannot depend on the value of any per-application domain variable. It is bad practice to mix the use (assignment) of per appdomain and per process variables.

For information on how to call a function in a specific application domain, see call_in_appdomain Function.

Example

```
// declspec_appdomain.cpp
// compile with: /clr
#include <stdio.h>
using namespace System;

class CGlobal {
public:
   CGlobal(bool bProcess) {
      Counter = 10;
      m_bProcess = bProcess;
      Console::WriteLine("__declspec({0}) CGlobal::CGlobal constructor", m_bProcess ? (Stri
ng^)"process" : (String^)"appdomain");
   }

   ~CGlobal() {
      Console::WriteLine("__declspec({0}) CGlobal::~CGlobal destructor", m_bProcess ? (Stri
ng^)"process" : (String^)"appdomain");
   }

   int Counter;

private:
   bool m_bProcess;
};

__declspec(process) CGlobal process_global = CGlobal(true);
__declspec(appdomain) CGlobal appdomain_global = CGlobal(false);

value class Functions {
public:
   static void change() {
      ++appdomain_global.Counter;
   }

   static void display() {
      Console::WriteLine("process_global value in appdomain '{0}': {1}",
```

```
            AppDomain::CurrentDomain->FriendlyName,
            process_global.Counter);

         Console::WriteLine("appdomain_global value in appdomain '{0}': {1}",
            AppDomain::CurrentDomain->FriendlyName,
            appdomain_global.Counter);
      }
   };

   int main() {
      AppDomain^ defaultDomain = AppDomain::CurrentDomain;
      AppDomain^ domain = AppDomain::CreateDomain("Domain 1");
      AppDomain^ domain2 = AppDomain::CreateDomain("Domain 2");
      CrossAppDomainDelegate^ changeDelegate = gcnew CrossAppDomainDelegate(&Functions::change
   );
      CrossAppDomainDelegate^ displayDelegate = gcnew CrossAppDomainDelegate(&Functions::displ
   ay);

      // Print the initial values of appdomain_global in all appdomains.
      Console::WriteLine("Initial value");
      defaultDomain->DoCallBack(displayDelegate);
      domain->DoCallBack(displayDelegate);
      domain2->DoCallBack(displayDelegate);

      // Changing the value of appdomain_global in the domain and domain2
      // appdomain_global value in "default" appdomain remain unchanged
      process_global.Counter = 20;
      domain->DoCallBack(changeDelegate);
      domain2->DoCallBack(changeDelegate);
      domain2->DoCallBack(changeDelegate);

      // Print values again
      Console::WriteLine("Changed value");
      defaultDomain->DoCallBack(displayDelegate);
      domain->DoCallBack(displayDelegate);
      domain2->DoCallBack(displayDelegate);

      AppDomain::Unload(domain);
      AppDomain::Unload(domain2);
   }
```

Output

```
__declspec(process) CGlobal::CGlobal constructor
__declspec(appdomain) CGlobal::CGlobal constructor
Initial value
process_global value in appdomain 'declspec_appdomain.exe': 10
appdomain_global value in appdomain 'declspec_appdomain.exe': 10
__declspec(appdomain) CGlobal::CGlobal constructor
process_global value in appdomain 'Domain 1': 10
appdomain_global value in appdomain 'Domain 1': 10
__declspec(appdomain) CGlobal::CGlobal constructor
process_global value in appdomain 'Domain 2': 10
appdomain_global value in appdomain 'Domain 2': 10
Changed value
process_global value in appdomain 'declspec_appdomain.exe': 20
appdomain_global value in appdomain 'declspec_appdomain.exe': 10
process_global value in appdomain 'Domain 1': 20
appdomain_global value in appdomain 'Domain 1': 11
process_global value in appdomain 'Domain 2': 20
appdomain_global value in appdomain 'Domain 2': 12
__declspec(appdomain) CGlobal::~CGlobal destructor
__declspec(appdomain) CGlobal::~CGlobal destructor
__declspec(appdomain) CGlobal::~CGlobal destructor
__declspec(process) CGlobal::~CGlobal destructor
```

See Also

Reference

__declspec

deprecated (C++)

(Microsoft specific) With the exceptions noted below, the **deprecated** declaration offers the same functionality as the deprecated pragma:

- The **deprecated** declaration lets you specify particular forms of function overloads as deprecated, whereas the pragma form applies to all overloaded forms of a function name.
- The **deprecated** declaration lets you specify a message that will display at compile time. The text of the message can be from a macro.
- Macros can only be marked as deprecated with the **deprecated** pragma.

If the compiler encounters the use of a deprecated identifier, a C4996 warning is thrown.

Example

The following sample shows how to mark functions as deprecated, and how to specify a message that will be displayed at compile time, when the deprecated function is used.

```
// deprecated.cpp
// compile with: /W1
#define MY_TEXT "function is deprecated"
void func1(void) {}
__declspec(deprecated) void func1(int) {}
__declspec(deprecated("** this is a deprecated function **")) void func2(int) {}
__declspec(deprecated(MY_TEXT)) void func3(int) {}

int main() {
   func1();
   func1(1);   // C4996
   func2(1);   // C4996
   func3(1);   // C4996
}
```

The following sample shows how to mark classes as deprecated, and how to specify a message that will be displayed at compile time, when the deprecated class is used.

```
// deprecate_class.cpp
// compile with: /W1
struct __declspec(deprecated) X {
   void f(){}
};

struct __declspec(deprecated("** X2 is deprecated **")) X2 {
   void f(){}
};

int main() {
   X x;   // C4996
   X2 x2;   // C4996
}
```

See Also

Reference

__declspec

C++ Keywords

dllexport, dllimport

Microsoft Specific

The **dllexport** and **dllimport** storage-class attributes are Microsoft-specific extensions to the C and C++ languages. They enable you to export and import functions, data, and objects to and from a DLL.

```
__declspec( dllimport ) declarator
__declspec( dllexport ) declarator
```

Remarks

These attributes explicitly define the DLL's interface to its client, which can be the executable file or another DLL. Declaring functions as **dllexport** eliminates the need for a module-definition (.DEF) file, at least with respect to the specification of exported functions. Note that **dllexport** replaces the **__export** keyword.

If a class is marked declspec(dllexport), any specializations of class templates in the class hierarchy are implicitly marked as declspec(dllexport). This means templates are explicitly instantiated and its members must be defined.

dllexport of a C++ function will expose the function with C++ name mangling. If C++ name mangling is not desired, either use a .def file (EXPORTS keyword) or declare the function as extern "C".

The declaration of **dllexport** and **dllimport** must use extended attribute syntax and the **__declspec** keyword.

Example

```
// Example of the dllimport and dllexport class attributes
__declspec( dllimport ) int i;
__declspec( dllexport ) void func();
```

Alternately, to make your code more readable, you can use macro definitions:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllExport void func();
DllExport int i = 10;
DllImport int j;
DllExport int n;
```

For additional information, see:

- Definitions and Declarations
- Defining Inline C++ Functions with dllexport and dllimport
- General Rules and Limitations
- Using dllimport and dllexport in C++ Classes

END Microsoft Specific

See Also

Reference

__declspec

C++ Keywords

Definitions and Declarations (C++)

Microsoft Specific

The DLL interface refers to all items (functions and data) that are known to be exported by some program in the system; that is, all items that are declared as **dllimport** or **dllexport**. All declarations included in the DLL interface must specify either the **dllimport** or **dllexport** attribute. However, the definition must specify only the **dllexport** attribute. For example, the following function definition generates a compiler error:

```
__declspec( dllimport ) int func() {   // Error; dllimport
                                    // prohibited on definition.
   return 1;
}
```

This code also generates an error:

```
#define DllImport   __declspec( dllimport )

__declspec( dllimport ) int i = 10;  // Error; this is a
                                     // definition.
```

However, this is correct syntax:

```
__declspec( dllexport ) int i = 10;     // Okay--export definition
```

The use of **dllexport** implies a definition, while **dllimport** implies a declaration. You must use the **extern** keyword with **dllexport** to force a declaration; otherwise, a definition is implied. Thus, the following examples are correct:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

extern DllImport int k; // These are both correct and imply a
DllImport int j;        // declaration.
```

The following examples clarify the preceding:

```
static __declspec( dllimport ) int l; // Error; not declared extern.

void func() {
    static __declspec( dllimport ) int s;  // Error; not declared
                                           // extern.
    __declspec( dllimport ) int m;         // Okay; this is a
                                           // declaration.
    __declspec( dllexport ) int n;         // Error; implies external
                                           // definition in local scope.
    extern __declspec( dllimport ) int i;  // Okay; this is a
                                           // declaration.
    extern __declspec( dllexport ) int k;  // Okay; extern implies
                                           // declaration.
    __declspec( dllexport ) int x = 5;     // Error; implies external
                                           // definition in local scope.
}
```

END Microsoft Specific

See Also

Reference

dllexport, dllimport

Defining Inline C++ Functions with dllexport and dllimport

Microsoft Specific

You can define as inline a function with the **dllexport** attribute. In this case, the function is always instantiated and exported, whether or not any module in the program references the function. The function is presumed to be imported by another program.

You can also define as inline a function declared with the **dllimport** attribute. In this case, the function can be expanded (subject to /Ob specifications), but never instantiated. In particular, if the address of an inline imported function is taken, the address of the function residing in the DLL is returned. This behavior is the same as taking the address of a non-inline imported function.

These rules apply to inline functions whose definitions appear within a class definition. In addition, static local data and strings in inline functions maintain the same identities between the DLL and client as they would in a single program (that is, an executable file without a DLL interface).

Exercise care when providing imported inline functions. For example, if you update the DLL, don't assume that the client will use the changed version of the DLL. To ensure that you are loading the proper version of the DLL, rebuild the DLL's client as well.

END Microsoft Specific

See Also

Reference

dllexport, dllimport

General Rules and Limitations

Microsoft Specific

- If you declare a function or object without the **dllimport** or **dllexport** attribute, the function or object is not considered part of the DLL interface. Therefore, the definition of the function or object must be present in that module or in another module of the same program. To make the function or object part of the DLL interface, you must declare the definition of the function or object in the other module as **dllexport**. Otherwise, a linker error is generated.

  If you declare a function or object with the **dllexport** attribute, its definition must appear in some module of the same program. Otherwise, a linker error is generated.

- If a single module in your program contains both **dllimport** and **dllexport** declarations for the same function or object, the **dllexport** attribute takes precedence over the **dllimport** attribute. However, a compiler warning is generated. For example:

```
__declspec( dllimport ) int i;
__declspec( dllexport ) int i;   // Warning; inconsistent;
                                 // dllexport takes precedence.
```

- In C++, you can initialize a globally declared or static local data pointer or with the address of a data object declared with the **dllimport** attribute, which generates an error in C. In addition, you can initialize a static local function pointer with the address of a function declared with the **dllimport** attribute. In C, such an assignment sets the pointer to the address of the DLL import thunk (a code stub that transfers control to the function) rather than the address of the function. In C++, it sets the pointer to the address of the function. For example:

```
__declspec( dllimport ) void func1( void );
__declspec( dllimport ) int i;

int *pi = &i;                             // Error in C
static void ( *pf )( void ) = &func1;     // Address of thunk in C,
                                          // function in C++

void func2()
{
   static int *pi = &i;                  // Error in C
   static void ( *pf )( void ) = &func1; // Address of thunk in C,
                                         // function in C++
}
```

  However, because a program that includes the **dllexport** attribute in the declaration of an object must provide the definition for that object somewhere in the program, you can initialize a global or local static function pointer with the address of a **dllexport** function. Similarly, you can initialize a global or local static data pointer with the address of a **dllexport** data object. For example, the following code does not generate errors in C or C++:

```
__declspec( dllexport ) void func1( void );
__declspec( dllexport ) int i;

int *pi = &i;                             // Okay
static void ( *pf )( void ) = &func1;     // Okay

void func2()
{
    static int *pi = &i;                  // Okay
    static void ( *pf )( void ) = &func1; // Okay
}
```

- Because of a change in behavior introduce in Visual C++ .NET to make the application of **dllexport** more consistent between regular classes and specializations of class templates, if you apply **dllexport** to a regular class that has a base class that is not marked as **dllexport**, the compiler will generate C4275.

  The compiler generates the same warning if the base class is a specialization of a class template. To work around this, mark the base-class with **dllexport**. The problem with a specialization of a class template is where to place the **__declspec(dllexport)**; you are not allowed to mark the class template. Instead, explicitly instantiate the class template and mark this explicit instantiation with **dllexport**. For example:

  ```
  template class __declspec(dllexport) B<int>;
  class __declspec(dllexport) D : public B<int> {
  // ...
  ```

  This workaround fails if the template argument is the deriving class. For example:

  ```
  class __declspec(dllexport) D : public B<D> {
  // ...
  ```

  Because this is common pattern with templates, the compiler changed the semantics of **dllexport** when it is applied to a class that has one or more base-classes and when one or more of the base classes is a specialization of a class template. In this case, the compiler implicitly applies **dllexport** to the specializations of class templates. In Visual C++ .NET, a user can do the following and not get a warning:

  ```
  class __declspec(dllexport) D : public B<D> {
  // ...
  ```

END Microsoft Specific

See Also

Reference

dllexport, dllimport

Using dllimport and dllexport in C++ Classes

Microsoft Specific

You can declare C++ classes with the **dllimport** or **dllexport** attribute. These forms imply that the entire class is imported or exported. Classes exported this way are called exportable classes.

The following example defines an exportable class. All its member functions and static data are exported:

```
#define DllExport   __declspec( dllexport )

class DllExport C {
   int i;
   virtual int func( void ) { return 1; }
};
```

Note that explicit use of the **dllimport** and **dllexport** attributes on members of an exportable class is prohibited.

dllexport Classes

When you declare a class **dllexport**, all its member functions and static data members are exported. You must provide the definitions of all such members in the same program. Otherwise, a linker error is generated. The one exception to this rule applies to pure virtual functions, for which you need not provide explicit definitions. However, because a destructor for an abstract class is always called by the destructor for the base class, pure virtual destructors must always provide a definition. Note that these rules are the same for nonexportable classes.

If you export data of class type or functions that return classes, be sure to export the class.

dllimport Classes

When you declare a class **dllimport**, all its member functions and static data members are imported. Unlike the behavior of **dllimport** and **dllexport** on nonclass types, static data members cannot specify a definition in the same program in which a **dllimport** class is defined.

Inheritance and Exportable Classes

All base classes of an exportable class must be exportable. If not, a compiler warning is generated. Moreover, all accessible members that are also classes must be exportable. This rule permits a **dllexport** class to inherit from a **dllimport** class, and a **dllimport** class to inherit from a **dllexport** class (though the latter is not recommended). As a rule, everything that is accessible to the DLL's client (according to C++ access rules) should be part of the exportable interface. This includes private data members referenced in inline functions.

Selective Member Import/Export

Because member functions and static data within a class implicitly have external linkage, you can declare them with the **dllimport** or **dllexport** attribute, unless the entire class is exported. If the entire class is imported or exported, the explicit declaration of member functions and data as **dllimport** or **dllexport** is prohibited. If you declare a static data member within a class definition as **dllexport**, a definition must occur somewhere within the same program (as with nonclass external linkage).

Similarly, you can declare member functions with the **dllimport** or **dllexport** attributes. In this case, you must provide a **dllexport** definition somewhere within the same program.

It is worthwhile to note several important points regarding selective member import and export:

- Selective member import/export is best used for providing a version of the exported class interface that is more restrictive; that is, one for which you can design a DLL that exposes fewer public and private features than the language would otherwise allow. It is also useful for fine-tuning the exportable interface: when you know that the client, by definition, is unable to access some private data, you need not export the entire class.
- If you export one virtual function in a class, you must export all of them, or at least provide versions that the client can use directly.
- If you have a class in which you are using selective member import/export with virtual functions, the functions must be in the exportable interface or defined inline (visible to the client).

- If you define a member as **dllexport** but do not include it in the class definition, a compiler error is generated. You must define the member in the class header.
- Although the definition of class members as **dllimport** or **dllexport** is permitted, you cannot override the interface specified in the class definition.
- If you define a member function in a place other than the body of the class definition in which you declared it, a warning is generated if the function is defined as **dllexport** or **dllimport** (if this definition differs from that specified in the class declaration).

END Microsoft Specific

See Also

Reference

dllexport, dllimport

jitintrinsic

Marks the function as significant to the 64-bit common language runtime. This is used on certain functions in Microsoft-provided libraries.

```
__declspec(jitintrinsic)
```

Remarks

jitintrinsic adds a MODOPT (IsJitIntrinsic) to a function signature.

Users are discouraged from using this **__declspec** modifier, as unexpected results can occur.

See Also

Reference

__declspec

C++ Keywords

naked (C++)

Microsoft Specific

For functions declared with the **naked** attribute, the compiler generates code without prolog and epilog code. You can use this feature to write your own prolog/epilog code sequences using inline assembler code. Naked functions are particularly useful in writing virtual device drivers. Note that the **naked** attribute is only valid on x86, and is not available on x64 or Itanium.

```
__declspec( naked ) declarator
```

Remarks

Because the **naked** attribute is only relevant to the definition of a function and is not a type modifier, naked functions must use extended attribute syntax and the __declspec keyword.

The compiler cannot generate an inline function for a function marked with the naked attribute, even if the function is also marked with the __forceinline keyword.

Examples

This code defines a function with the **naked** attribute:

```
__declspec( naked ) int func( formal_parameters ) {}
```

Or, alternately:

```
#define Naked __declspec( naked )
Naked int func( formal_parameters ) {}
```

The **naked** attribute affects only the nature of the compiler's code generation for the function's prolog and epilog sequences. It does not affect the code that is generated for calling such functions. Thus, the **naked** attribute is not considered part of the function's type, and function pointers cannot have the **naked** attribute. Furthermore, the **naked** attribute cannot be applied to a data definition. For example, this code sample generates an error:

```
__declspec( naked ) int i;       // Error--naked attribute not
                                 // permitted on data declarations.
```

The **naked** attribute is relevant only to the definition of the function and cannot be specified in the function's prototype. For example, this declaration generates a compiler error:

```
__declspec( naked ) int func();  // Error--naked attribute not
                                 // permitted on function declarations
```

END Microsoft Specific

See Also

Reference

__declspec

C++ Keywords

Naked Function Calls

noalias

Microsoft Specific

noalias means that a function call does not modify or reference visible global state and only modifies the memory pointed to **directly** by pointer parameters (first-level indirections).

If a function is annotated as **noalias**, the optimizer can assume that, in addition to the parameters themselves, only first-level indirections of pointer parameters are referenced or modified inside the function. The visible global state is the set of all data that is not defined or referenced outside of the compilation scope, and their address is not taken. The compilation scope is all source files (/LTCG (Link-time Code Generation) builds) or a single source file (non-**/LTCG** build).

Example

The following sample demonstrates using **__declspec(restrict)** and **__declspec(noalias)**. Normally, memory returned from **malloc** is **restrict** and **noalias** because the CRT headers are decorated appropriately.

However, in this example, the pointers `mempool` and `memptr` are global so the compiler has no assurance that the memory is not subject to aliasing. Decorating the functions that return pointers with **__declspec(restrict)** tells the compiler that the memory pointed to by the return value is not aliased.

Decorating the function in the example that accesses memory with **__declspec(noalias)** tells the compiler that this function does not interfere with the global state except through the pointers in its parameter list.

```
// declspec_noalias.c
#include <stdio.h>
#include <stdlib.h>

#define M 800
#define N 600
#define P 700

float * mempool, * memptr;

__declspec(restrict) float * ma(int size)
{
    float * retval;
    retval = memptr;
    memptr += size;
    return retval;
}

__declspec(restrict) float * init(int m, int n)
{
    float * a;
    int i, j;
    int k=1;

    a = ma(m * n);
    if (!a) exit(1);
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            a[i*n+j] = 0.1/k++;
    return a;
}

__declspec(noalias) void multiply(float * a, float * b, float * c)
{
    int i, j, k;

    for (j=0; j<P; j++)
        for (i=0; i<M; i++)
            for (k=0; k<N; k++)
                c[i * P + j] =
                          a[i * N + k] *
                          b[k * P + j];
```

```
}

int main()
{
    float * a, * b, * c;

    mempool = (float *) malloc(sizeof(float) * (M*N + N*P + M*P));

    if (!mempool)
        puts("ERROR: Malloc returned null"); exit(1);

    memptr = mempool;
    a = init(M, N);
    b = init(N, P);
    c = init(M, P);

    multiply(a, b, c);
}
```

See Also

Reference

__declspec

C++ Keywords

noinline

Microsoft Specific

__declspec(noinline) tells the compiler to never inline a particular member function (function in a class).

It may be worthwhile to not inline a function if it is small and not critical to the performance of your code. That is, if the function is small and not likely to be called often, such as a function that handles an error condition.

Keep in mind that if a function is marked **noinline**, the calling function will be smaller and thus, itself a candidate for compiler inlining.

```
class X {
   __declspec(noinline) int mbrfunc() {
      return 0;
   }   // will not inline
};
```

END Microsoft Specific

See Also

Reference

__declspec

C++ Keywords

inline, __inline, __forceinline

noreturn

Microsoft Specific

This **__declspec** attribute tells the compiler that a function does not return. As a consequence, the compiler knows that the code following a call to a **__declspec(noreturn)** function is unreachable.

If the compiler finds a function with a control path that does not return a value, it generates a warning (C4715) or error message (C2202). If the control path cannot be reached due to a function that never returns, you can use **__declspec(noreturn)** to prevent this warning or error.

Note
Adding **__declspec(noreturn)** to a function that is expected to return can result in undefined behavior.

Example

In the following sample,the **else** clause does not contain a return statement. Declaring `fatal` as **__declspec(noreturn)** avoids an error or warning message.

```
// noreturn2.cpp
__declspec(noreturn) extern void fatal () {}

int main() {
   if(1)
     return 1;
   else if(0)
     return 0;
   else
     fatal();
}
```

See Also

Reference

__declspec

C++ Keywords

nothrow (C++)

Microsoft Specific

A **__declspec** extended attribute which can be used in the declaration of functions.

```
return-type __declspec(nothrow) [call-convention] function-name ([argument-list])
```

Remarks

This attribute tells the compiler that the declared function and the functions it calls never throw an exception. With the synchronous exception handling model, now the default, the compiler can eliminate the mechanics of tracking the lifetime of certain unwindable objects in such a function, and significantly reduce the code size. Given the following preprocessor directive, the three function declarations below are equivalent:

```
#define WINAPI __declspec(nothrow) __stdcall

void WINAPI f1();
void __declspec(nothrow) __stdcall f2();
void __stdcall f3() throw();
```

Using `void __declspec(nothrow) __stdcall f2();` has the advantage that you can use an API definition, such as that illustrated by the `#define` statement, to easily specify **nothrow** on a set of functions. The third declaration, `void __stdcall f3() throw();` is the syntax defined by the C++ standard.

See Synchronous Exception Handling for more information.

END Microsoft Specific

See Also

Reference

__declspec

C++ Keywords

novtable

Microsoft Specific

This is a **__declspec** extended attribute.

This form of **__declspec** can be applied to any class declaration, but should only be applied to pure interface classes, that is, classes that will never be instantiated on their own. The **__declspec** stops the compiler from generating code to initialize the vfptr in the constructor(s) and destructor of the class. In many cases, this removes the only references to the vtable that are associated with the class and, thus, the linker will remove it. Using this form of **__declspec** can result in a significant reduction in code size.

If you attempt to instantiate a class marked with **novtable** and then access a class member, you will receive an access violation (AV).

Example

```
// novtable.cpp
#include <stdio.h>

struct __declspec(novtable) X {
   virtual void mf();
};

struct Y : public X {
   void mf() {
      printf_s("In Y\n");
   }
};

int main() {
   // X *pX = new X();
   // pX->mf();   // AV at runtime

   Y *pY = new Y();
   pY->mf();
}
```

Output

```
In Y
```

END Microsoft Specific

See Also

Reference

__declspec

C++ Keywords

process

Specifies that your managed application process should have a single copy of a particular global variable, static member variable, or static local variable shared across all application domains in the process. This is primarily intended to be used when compiling with **/clr:pure**, because under **/clr:pure** global and static variables are per application domain, by default. When compiling with **/clr**, global and static variables are per process by default (do not need to use **__declspec(process)**.

Only a global variable, a static member variable, or a static local variable of native type can be marked with **__declspec(process)**.

When compiling with **/clr:pure**, variables marked as per process must also be declared as **const**. This ensures that per process variables are not changed in one application domain, and giving unexpected results in another application domain. The primary intended use of **__declspec(process)** is to enable compile time initialization of a global variable, static member variable, or static local variable under **/clr:pure**.

process is only valid when compiling with /clr or **/clr:pure** and is not valid when compiling with **/clr:safe**.

If you want each application domain to have its own copy of a global variable, use appdomain.

See Application Domains and Visual C++ for more information.

See Also

Reference

__declspec

C++ Keywords

property (C++)

Microsoft Specific

This attribute can be applied to non-static "virtual data members" in a class or structure definition. The compiler treats these "virtual data members" as data members by changing their references into function calls.

```
__declspec( property( get=get_func_name ) ) declarator
__declspec( property( put=put_func_name ) ) declarator
__declspec( property( get=get_func_name, put=put_func_name ) ) declarator
```

Remarks

When the compiler sees a data member declared with this attribute on the right of a member-selection operator ("**.**" or "**->**"), it converts the operation to a **get** or **put** function, depending on whether such an expression is an l-value or an r-value. In more complicated contexts, such as "**+=**", a rewrite is performed by doing both **get** and **put**.

This attribute can also be used in the declaration of an empty array in a class or structure definition. For example:

```
__declspec(property(get=GetX, put=PutX)) int x[];
```

The above statement indicates that `x[]` can be used with one or more array indices. In this case, `i=p->x[a][b]` will be turned into `i=p->GetX(a, b)`, and `p->x[a][b] = i` will be turned into `p->PutX(a, b, i);`

END Microsoft Specific

Example

```
// declspec_property.cpp
struct S {
   int i;
   void putprop(int j) {
      i = j;
   }

   int getprop() {
      return i;
   }

   __declspec(property(get = getprop, put = putprop)) int the_prop;
};

int main() {
   S s;
   s.the_prop = 5;
   return s.the_prop;
}
```

See Also

Reference

__declspec

C++ Keywords

restrict

Microsoft Specific

Applied to a function declaration or definition that returns a pointer type and tells the compiler that the function returns an object that will not be aliased with any other pointers.

```
__declspec(restrict) return_type f();
```

Remarks

The compiler will propagate **__declspec(restrict)**. For example, the CRT **malloc** function is decorated with **__declspec(restrict)** and therefore, pointers initialized to memory locations with **malloc** are also implied to not be aliased.

The compiler does not check that the pointer is actually not aliased. It is the developer's responsibility to ensure the program does not alias a pointer marked with the **restrict __declspec** modifier.

For similar semantics on variables, see __restrict.

Example

See noalias for an example using **restrict**.

END Microsoft Specific

See Also
Reference
__declspec
C++ Keywords

selectany

Microsoft Specific

Tells the compiler that the declared global data item (variable or object) is a pick-any COMDAT (a packaged function).

```
__declspec( selectany ) declarator
```

Remarks

At link time, if multiple definitions of a COMDAT are seen, the linker picks one and discards the rest. If the linker option /OPT:REF (Optimizations) is selected, then COMDAT elimination will occur to remove all the unreferenced data items in the linker output.

Constructors and assignment by global function or static methods in the declaration do not create a reference and will not prevent /OPT:REF elimination. Side effects from such code should not be depended on when no other references to the data exist.

For dynamically initialized, global objects, **selectany** will discard an unreferenced object's initialization code, as well.

A global data item can normally be initialized only once in an EXE or DLL project. **selectany** can be used in initializing global data defined by headers, when the same header appears in more than one source file. **selectany** is available in both the C and C++ compilers.

Note
selectany can only be applied to the actual initialization of global data items that are externally visible.

Example

This code shows how to use the **selectany** attribute:

```
//Correct - x1 is initialized and externally visible
__declspec(selectany) int x1=1;

//Incorrect - const is by default static in C++, so
//x2 is not visible externally (This is OK in C, since
//const is not by default static in C)
const __declspec(selectany) int x2 =2;

//Correct - x3 is extern const, so externally visible
extern const __declspec(selectany) int x3=3;

//Correct - x4 is extern const, so it is externally visible
extern const int x4;
const __declspec(selectany) int x4=4;

//Incorrect - __declspec(selectany) is applied to the uninitialized
//declaration of x5
extern __declspec(selectany) int x5;

// OK: dynamic initialization of global object
class X {
public:
X(int i){i++;};
int i;
};

__declspec(selectany) X x(1);
```

This code shows how to use the **selectany** attribute to ensure data COMDAT folding when you also use the /OPT:ICF linker option. Note that data must be marked with **selectany** and placed in a **const** (readonly) section. You must explicitly specify the read-only section.

```
// selectany2.cpp
// in the following lines, const marks the variables as read only
__declspec(selectany) extern const int ix = 5;
__declspec(selectany) extern const int jx = 5;
int main() {
   int ij;
   ij = ix + jx;
}
```

END Microsoft Specific

See Also

Reference

__declspec

C++ Keywords

thread

Microsoft Specific

The **thread** extended storage-class modifier is used to declare a thread local variable.

```
__declspec( thread ) declarator
```

Remarks

Thread Local Storage (TLS) is the mechanism by which each thread in a multithreaded process allocates storage for thread-specific data. In standard multithreaded programs, data is shared among all threads of a given process, whereas thread local storage is the mechanism for allocating per-thread data. For a complete discussion of threads, see Multithreading.

Declarations of thread local variables must use extended attribute syntax and the **__declspec** keyword with the **thread** keyword. For example, the following code declares an integer thread local variable and initializes it with a value:

```
__declspec( thread ) int tls_i = 1;
```

You must observe these guidelines when declaring thread local objects and variables:

- You can apply the **thread** attribute only to data declarations and definitions, and classes that do not have member functions; **thread** cannot be used on function declarations or definitions.
- The use of the **thread** attribute may interfere with delay loading of DLL imports**.**
- You can specify the **thread** attribute only on data items with static storage duration. This includes global data objects (both **static** and **extern**), local static objects, and static data members of classes. You cannot declare automatic data objects with the **thread** attribute.
- You must use the **thread** attribute for the declaration and the definition of a thread local object, whether the declaration and definition occur in the same file or separate files.
- You cannot use the **thread** attribute as a type modifier.
- Only POD classes may be instantiated using **__declspec(thread)**. The thread attribute is ignored if no object is declared as part of the class declaration. For example:

  ```
  // declspec_thread.cpp
  // compile with: /LD
  __declspec(thread) class X
  {
  public:
     int I;
  } x;   // x is a thread object

  X y;   // y is not a thread object
  ```

- Because the declaration of objects that use the **thread** attribute is permitted, these two examples are semantically equivalent:

  ```
  // declspec_thread_2.cpp
  // compile with: /LD
  __declspec( thread ) class B {
  public:
     int data;
  } BObject;   // BObject declared thread local.

  class B2 {
  ```

```
public:
   int data;
};
__declspec( thread ) B2 BObject2;   // BObject2 declared thread local.
```

- Standard C permits initialization of an object or variable with an expression involving a reference to itself, but only for objects of nonstatic extent. Although C++ normally permits such dynamic initialization of an object with an expression involving a reference to itself, this type of initialization is not permitted with thread local objects. For example:

```
// declspec_thread_3.cpp
// compile with: /LD
#define Thread __declspec( thread )
int j = j;   // Okay in C++; C error
Thread int tls_i = sizeof( tls_i );   // Okay in C and C++
```

  Note that a **sizeof** expression that includes the object being initialized does not constitute a reference to itself and is allowed in C and C++.

END Microsoft Specific

See Also

Reference

__declspec

C++ Keywords

Concepts

Thread Local Storage (TLS)

uuid (C++)

Microsoft Specific

The compiler attaches a GUID to a class or structure declared or defined (full COM object definitions only) with the **uuid** attribute.

```
__declspec( uuid("ComObjectGUID") ) declarator
```

Remarks

The **uuid** attribute takes a string as its argument. This string names a GUID in normal registry format with or without the **{ }** delimiters. For example:

```
struct __declspec(uuid("00000000-0000-0000-c000-000000000046")) IUnknown;
struct __declspec(uuid("{00020400-0000-0000-c000-000000000046}")) IDispatch;
```

This attribute can be applied in a redeclaration. This allows the system headers to supply the definitions of interfaces such as **IUnknown**, and the redeclaration in some other header (such as COMDEF.H) to supply the GUID.

The keyword __uuidof can be applied to retrieve the constant GUID attached to a user-defined type.

END Microsoft Specific

See Also

Reference

__declspec

C++ Keywords

__restrict

Like the restrict **__declspec** modifier, the **__restrict** keyword indicates that a symbol is not aliased in the current scope. The **__restrict** keyword differs from the **restrict __declspec** modifier in the following ways:

- The **__restrict** keyword is valid only on variables, and **__declspec(restrict)** is only valid on function declarations and definitions.
- When **__restrict** is used, the compiler will not propagate the no-alias property of a variable. That is, if you assign a **__restrict** variable to a non-**__restrict** variable, the compiler will not imply that the non-**__restrict** variable is not aliased.

Generally, if you affect the behavior of an entire function, it is better to use the **__declspec** than the keyword.

__restrict is similar to **restrict** from the C99 spec, but **__restrict** can be used in C++ or C programs.

There is no support for **__restrict** on C++ references.

Note
When used on a variable that also has the volatile (C++) keyword, **volatile** will take precedence.

Example

```
// __restrict_keyword.c
// compile with: /LD
// In the following function, declare a and b as disjoint arrays
// but do not have same assurance for c and d.
void sum2(int n, int * __restrict a, int * __restrict b,
          int * c, int * d) {
   int i;
   for (i = 0; i < n; i++) {
      a[i] = b[i] + c[i];
      c[i] = b[i] + d[i];
    }
}

// By marking union members as __restrict, tell compiler that
// only z.x or z.y will be accessed in any given scope.
union z {
   int * __restrict x;
   double * __restrict y;
};
```

See Also

Reference

C++ Keywords

__unaligned

When a pointer is declared as **__unaligned**, the compiler assumes that the type or data pointed to is not aligned. **__unaligned** is only valid in compilers for x64 and the Itanium Processor Family (IPF).

Remarks

When declaring a pointer to a structure that may not have aligned data, you can use the **__unaligned** keyword to tell the compiler that the type must be read one byte at a time.

By default, pointers are aligned.

For more information about alignment, see,

- align (C++)
- __alignof Operator
- pack
- /Zp (Struct Member Alignment)
- Structure Alignment Examples (x64 specific)

Example

```
// unaligned_keyword.cpp
// compile with: /c
// processor: x64 IPF
#include <stdio.h>
int main() {
   char buf[100];

   int __unaligned *p1 = (int*)(&buf[37]);
   int *p2 = (int *)p1;

   *p1 = 0;   // ok

   __try {
      *p2 = 0;  // throws an exception
   }
   __except(1) {
      puts("exception");
   }
}
```

See Also

Reference

C++ Keywords

__w64

(Microsoft Specific) Lets you mark variables, such that when you compile with /Wp64 the compiler will report any warnings that would be reported if you were compiling with a 64-bit compiler.

```
type __w64 identifier
```

Parameters

type

One of the three types that might cause problems in code being ported from a 32-bit to a 64-bit compiler: **int**, **long**, or a pointer.

identifier

The identifier for the variable you are creating.

Remarks

Any typedef that has **__w64** on it must be 32 bits on x86 and 64 bits on Itanium Processor Family (IPF).

The **__w64** modifier should be specified on any typedefs that change size between 32 bit and 64 bit platforms. For any such type, **__w64** should appear only on the 32-bit definition of the typedef.

__w64 is ignored if the compilation does not use /Wp64.

For more information about porting to 64-bit, see

- Compiler Options
- /Wp64
- Porting 32-Bit Code to 64-Bit Code

Example

```
// __w64.cpp
// compile with: /W3 /Wp64
typedef int Int_32;
#ifdef _WIN64
typedef __int64 Int_Native;
#else
typedef int __w64 Int_Native;
#endif

int main() {
   Int_32 i0 = 5;
   Int_Native i1 = 10;
   i0 = i1;   // C4244 64-bit int assigned to 32-bit int

   // char __w64 c;  error, cannot use __w64 on char
}
```

See Also

Reference

C++ Keywords

__sptr, __uptr

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Use the **__sptr** or **__uptr** modifier on a 32-bit pointer declaration to specify how the compiler converts a 32-bit pointer to a 64-bit pointer. A 32-bit pointer is converted, for example, when it is assigned to a 64-bit pointer variable or is dereferenced on a 64-bit platform.

Microsoft documentation for support of 64-bit platforms sometimes refers to the most significant bit of a 32-bit pointer as the sign bit. By default, the compiler uses sign extension to convert a 32-bit pointer to a 64-bit pointer. That is, the least significant 32 bits of the 64-bit pointer are set to the value of the 32-bit pointer and the most significant 32 bits are set to the value of the sign bit of the 32-bit pointer. This conversion yields correct results if the sign bit is 0, but not if the sign bit is 1. For example, the 32-bit address 0x7FFFFFFF yields the equivalent 64-bit address 0x000000007FFFFFFF, but the 32-bit address 0x80000000 is incorrectly changed to 0xFFFFFFFF80000000.

The **__sptr**, or signed pointer, modifier specifies that a pointer conversion set the most significant bits of a 64-bit pointer to the sign bit of the 32-bit pointer. The **__uptr**, or unsigned pointer, modifier specifies that a conversion set the most significant bits to zero. The following declarations show the **__sptr** and **__uptr** modifiers used with two unqualified pointers, two pointers qualified with the __ptr32 type, and a function parameter.

```
int * __sptr psp;
int * __uptr pup;
int * __ptr32 __sptr psp32;
int * __ptr32 __uptr pup32;
void MyFunction(char * __uptr __ptr32 myValue);
```

Use the **__sptr** and **__uptr** modifiers with pointer declarations. Use the modifiers in the position of a pointer type qualifier, which means the modifier must follow the asterisk. You cannot use the modifiers with pointers to members. The modifiers do not affect non-pointer declarations.

If you do not use the **__sptr** or **__uptr** modifier, and you enable Compiler Warning (level 2) C4826, the compiler issues a warning when a 32-bit pointer is converted to 64 bits.

Example

The following example declares 32-bit pointers that use the **__sptr** and **__uptr** modifiers, assigns each 32-bit pointer to a 64-bit pointer variable, and then displays the hexadecimal value of each 64-bit pointer. The example is compiled with the native 64-bit compiler and is executed on a 64-bit platform.

```
// sptr_uptr.cpp
// processor: x64
#include "stdio.h"

// Warning C4826 is off by default.

int main()
{
    void *        __ptr64 p64;
    void *        __ptr32 p32d; //default signed pointer
    void * __sptr __ptr32 p32s; //explicit signed pointer
    void * __uptr __ptr32 p32u; //explicit unsigned pointer

// Set the 32-bit pointers to a value whose sign bit is 1.
    p32d = reinterpret_cast<void *>(0x87654321);
    p32s = p32d;
    p32u = p32d;

// The printf() function automatically displays leading zeroes with each 32-bit pointer. Th
ese are unrelated
// to the __sptr and __uptr modifiers.
    printf("Display each 32-bit pointer (as an unsigned 64-bit pointer):\n");
    printf("p32d:       %p\n", p32d);
    printf("p32s:       %p\n", p32s);
```

```
    printf("p32u:       %p\n", p32u);

    printf("\nDisplay the 64-bit pointer created from each 32-bit pointer:\n");
    p64 = p32d;
    printf("p32d: p64 = %p\n", p64);
    p64 = p32s;
    printf("p32s: p64 = %p\n", p64);
    p64 = p32u;
    printf("p32u: p64 = %p\n", p64);
    return 0;
}
```

Output

```
Display each 32-bit pointer (as an unsigned 64-bit pointer):
p32d:       0000000087654321
p32s:       0000000087654321
p32u:       0000000087654321

Display the 64-bit pointer created from each 32-bit pointer:
p32d: p64 = FFFFFFFF87654321
p32s: p64 = FFFFFFFF87654321
p32u: p64 = 0000000087654321
```

END Microsoft Specific

See Also

Reference

Microsoft-Specific Modifiers

Inline Assembler

Microsoft Specific

Assembly language serves many purposes, such as improving program speed, reducing memory needs, and controlling hardware. You can use the inline assembler to embed assembly-language instructions directly in your C and C++ source programs without extra assembly and link steps. The inline assembler is built into the compiler, so you don't need a separate assembler such as the Microsoft Macro Assembler (MASM).

Note
Programs with inline assembler code are not fully portable to other hardware platforms. If you are designing for portability, avoid using inline assembler.

Inline assembly is not supported on the Itanium and x64 processors. The following topics explain how to use the Visual C/C++ inline assembler with x86 processors:

- Inline Assembler Overview
- Advantages of Inline Assembly
- __asm
- Using Assembly Language in __asm Blocks
- Using C or C++ in __asm Blocks
- Using and Preserving Registers in Inline Assembly
- Jumping to Labels in Inline Assembly
- Calling C Functions in Inline Assembly
- Calling C++ Functions in Inline Assembly
- Defining __asm Blocks as C Macros
- Optimizing Inline Assembly

END Microsoft Specific

See Also

Other Resources

C++ Language Reference

Inline Assembler Overview

Microsoft Specific

The inline assembler lets you embed assembly-language instructions in your C and C++ source programs without extra assembly and link steps. The inline assembler is built into the compiler — you don't need a separate assembler such as the Microsoft Macro Assembler (MASM).

Because the inline assembler doesn't require separate assembly and link steps, it is more convenient than a separate assembler. Inline assembly code can use any C or C++ variable or function name that is in scope, so it is easy to integrate it with your program's C and C++ code. And because the assembly code can be mixed with C and C++ statements, it can do tasks that are cumbersome or impossible in C or C++ alone.

The __asm keyword invokes the inline assembler and can appear wherever a C or C++ statement is legal. It cannot appear by itself. It must be followed by an assembly instruction, a group of instructions enclosed in braces, or, at the very least, an empty pair of braces. The term "**__asm** block" here refers to any instruction or group of instructions, whether or not in braces.

The following code is a simple **__asm** block enclosed in braces. (The code is a custom function prolog sequence.)

```
// asm_overview.cpp
// processor: x86
void __declspec(naked) main()
{
    // Naked functions must provide their own prolog...
    __asm {
        push ebp
        mov ebp, esp
        sub esp, __LOCAL_SIZE
    }

    // ... and epilog
    __asm {
        pop ebp
        ret
    }
}
```

Alternatively, you can put **__asm** in front of each assembly instruction:

```
__asm push ebp
__asm mov  ebp, esp
__asm sub  esp, __LOCAL_SIZE
```

Since the **__asm** keyword is a statement separator, you can also put assembly instructions on the same line:

```
__asm push ebp   __asm mov  ebp, esp   __asm sub  esp, __LOCAL_SIZE
```

END Microsoft Specific

See Also
Reference
Inline Assembler

Advantages of Inline Assembly

Microsoft Specific

Because the inline assembler doesn't require separate assembly and link steps, it is more convenient than a separate assembler. Inline assembly code can use any C variable or function name that is in scope, so it is easy to integrate it with your program's C code. Because the assembly code can be mixed inline with C or C++ statements, it can do tasks that are cumbersome or impossible in C or C++.

The uses of inline assembly include:

- Writing functions in assembly language.
- Spot-optimizing speed-critical sections of code.
- Making direct hardware access for device drivers.
- Writing prolog and epilog code for "naked" calls.

Inline assembly is a special-purpose tool. If you plan to port an application to different machines, you'll probably want to place machine-specific code in a separate module. Because the inline assembler doesn't support all of Microsoft Macro Assembler's (MASM) macro and data directives, you may find it more convenient to use MASM for such modules.

END Microsoft Specific

See Also
Reference
Inline Assembler

__asm

Microsoft Specific

The **__asm** keyword invokes the inline assembler and can appear wherever a C or C++ statement is legal. It cannot appear by itself. It must be followed by an assembly instruction, a group of instructions enclosed in braces, or, at the very least, an empty pair of braces. The term "**__asm** block" here refers to any instruction or group of instructions, whether or not in braces.

Note
Visual C++ support for the Standard C++ **asm** keyword is limited to the fact that the compiler will not generate an error on the keyword. However, an **asm** block will not generate any meaningful code. Use **__asm** instead of **asm**.

Syntax:

__asm *assembly-instruction* [;]

__asm { *assembly-instruction-list* } [;]

Grammar
asm-statement:

__asm *assembly-instruction* **;** $_{\text{opt}}$

__asm { *assembly-instruction-list* **};** $_{\text{opt}}$

assembly-instruction-list:

assembly-instruction **;** $_{\text{opt}}$

assembly-instruction **;** *assembly-instruction-list* **;** $_{\text{opt}}$

If used without braces, the **__asm** keyword means that the rest of the line is an assembly-language statement. If used with braces, it means that each line between the braces is an assembly-language statement. For compatibility with previous versions, **_asm** is a synonym for **__asm**.

Since the **__asm** keyword is a statement separator, you can put assembly instructions on the same line.

Before Visual C++ 2005, the instruction,

```
__asm int 3
```

did not cause native code to be generated when compiled with **/clr**; the compiler translated the instruction to a CLR break instruction. Beginning in Visual C++ 2005, `__asm int 3` now results in native code generation for the function. If you want a function to cause a break point in your code and if you want that function compiled to MSIL, use __debugbreak.

Example

The following code fragment is a simple **__asm** block enclosed in braces:

```
__asm {
   mov al, 2
   mov dx, 0xD007
   out dx, al
}
```

Alternatively, you can put **__asm** in front of each assembly instruction:

```
__asm mov al, 2
__asm mov dx, 0xD007
__asm out dx, al
```

Because the **__asm** keyword is a statement separator, you can also put assembly instructions on the same line:

```
__asm mov al, 2   __asm mov dx, 0xD007   __asm out dx, al
```

All three examples generate the same code, but the first style (enclosing the **__asm** block in braces) has some advantages. The braces clearly separate assembly code from C or C++ code and avoid needless repetition of the **__asm** keyword. Braces can also prevent ambiguities. If you want to put a C or C++ statement on the same line as an **__asm** block, you must enclose the block in braces. Without the braces, the compiler cannot tell where assembly code stops and C or C++ statements begin. Finally, because the text in braces has the same format as ordinary MASM text, you can easily cut and paste text from existing MASM source files.

Unlike braces in C and C++, the braces enclosing an **__asm** block don't affect variable scope. You can also nest **__asm** blocks; nesting does not affect variable scope.

END Microsoft Specific

See Also
Reference
C++ Keywords
Inline Assembler

Using Assembly Language in __asm Blocks

Microsoft Specific

The inline assembler has much in common with other assemblers. For example, it accepts any expression that is legal in MASM. This section describes the use of assembly-language features in **__asm** blocks.

What do you want to know more about?

- Instruction Set for Inline Assembly
- MASM Expressions in Inline Assembly
- Data Directives and Operators in Inline Assembly
- EVEN and ALIGN Directives
- MASM Macro Directives in Inline Assembly
- Segment References in Inline Assembly
- Type and Variable Sizes in Inline Assembly
- Assembly-Language Comments
- The _emit Pseudoinstruction
- Debugging and Listings for Inline Assembly
- Intel's MMX Instruction Set

END Microsoft Specific

See Also
Reference
Inline Assembler

Instruction Set for Inline Assembly

Microsoft Specific

The Visual C++ compiler supports all opcodes through the Pentium 4 and AMD Athlon. Additional instructions supported by the target processor can be created with the _emit Pseudoinstruction.

END Microsoft Specific

See Also
Reference
Using Assembly Language in __asm Blocks

MASM Expressions in Inline Assembly

Microsoft Specific

Inline assembly code can use any MASM expression, which is any combination of operands and operators that evaluates to a single value or address.

END Microsoft Specific

See Also

Reference

Using Assembly Language in __asm Blocks

Data Directives and Operators in Inline Assembly

Microsoft Specific

Although an **__asm** block can reference C or C++ data types and objects, it cannot define data objects with MASM directives or operators. Specifically, you cannot use the definition directives **DB**, **DW**, **DD**, **DQ**, **DT**, and **DF**, or the operators **DUP** or **THIS**. MASM structures and records are also unavailable. The inline assembler doesn't accept the directives **STRUC**, **RECORD**, **WIDTH**, or **MASK**.

END Microsoft Specific

See Also

Reference

Using Assembly Language in __asm Blocks

EVEN and ALIGN Directives

Microsoft Specific

Although the inline assembler doesn't support most MASM directives, it does support **EVEN** and **ALIGN**. These directives put **NOP** (no operation) instructions in the assembly code as needed to align labels to specific boundaries. This makes instruction-fetch operations more efficient for some processors.

END Microsoft Specific

See Also
Reference
Using Assembly Language in __asm Blocks

MASM Macro Directives in Inline Assembly

Microsoft Specific

The inline assembler is not a macro assembler. You cannot use MASM macro directives (**MACRO**, **REPT**, **IRC**, **IRP**, and **ENDM**) or macro operators (**<>**, **!**, **&**, **%**, and **.TYPE**). An **__asm** block can use C preprocessor directives, however. See Using C or C++ in __asm Blocks for more information.

END Microsoft Specific

See Also
Reference
Using Assembly Language in __asm Blocks

Segment References in Inline Assembly

Microsoft Specific

You must refer to segments by register rather than by name (the segment name **_TEXT** is invalid, for instance). Segment overrides must use the register explicitly, as in ES:[BX].

END Microsoft Specific

See Also

Reference

Using Assembly Language in __asm Blocks

Type and Variable Sizes in Inline Assembly

Microsoft Specific

The **LENGTH**, **SIZE**, and **TYPE** operators have a limited meaning in inline assembly. They cannot be used at all with the **DUP** operator (because you cannot define data with MASM directives or operators). But you can use them to find the size of C or C++ variables or types:

- The **LENGTH** operator can return the number of elements in an array. It returns the value 1 for non-array variables.
- The **SIZE** operator can return the size of a C or C++ variable. A variable's size is the product of its **LENGTH** and **TYPE**.
- The **TYPE** operator can return the size of a C or C++ type or variable. If the variable is an array, **TYPE** returns the size of a single element of the array.

For example, if your program has an 8-element **int** array,

```
int arr[8];
```

the following C and assembly expressions yield the size of `arr` and its elements.

__asm	C	Size
LENGTH arr	**sizeof**(arr)/**sizeof**(arr[0])	8
SIZE arr	**sizeof**(arr)	32
TYPE arr	**sizeof**(arr[0])	4

END Microsoft Specific

See Also
Reference
Using Assembly Language in __asm Blocks

Assembly-Language Comments

Microsoft Specific

Instructions in an **__asm** block can use assembly-language comments:

```
__asm mov ax, offset buff ; Load address of buff
```

Because C macros expand into a single logical line, avoid using assembly-language comments in macros. (See Defining __asm Blocks as C Macros.) An **__asm** block can also contain C-style comments; for more information, see Using C or C++ in __asm Blocks.

END Microsoft Specific

See Also

Reference

Using Assembly Language in __asm Blocks

The _emit Pseudoinstruction

Microsoft Specific

The **_emit** pseudoinstruction is similar to the **DB** directive of MASM. You use **_emit** to define a single immediate byte at the current location in the current text segment. However, **_emit** can define only one byte at a time, and it can only define bytes in the text segment. It uses the same syntax as the **INT** instruction.

The following fragment places the given bytes into the code:

```
#define randasm __asm _emit 0x4A __asm _emit 0x43 __asm _emit 0x4B
 .
 .
 .
__asm {
     randasm
     }
```

END Microsoft Specific

See Also
Reference
Using Assembly Language in __asm Blocks

Debugging and Listings for Inline Assembly

Microsoft Specific

Programs containing inline assembly code can be debugged with a source-level debugger if you compile with the /Zi option.

Within the debugger, you can set breakpoints on both C or C++ and assembly-language lines. If you enable mixed assembly and source mode, you can display both the source and disassembled form of the assembly code.

Note that putting multiple assembly instructions or source language statements on one line can hamper debugging. In source mode, you can use the debugger to set breakpoints on a single line but not on individual statements on the same line. The same principle applies to an **__asm** block defined as a C macro, which expands to a single logical line.

If you create a mixed source and assembly listing with the /FAs compiler option, the listing contains both the source and assembly forms of each assembly-language line. Macros are not expanded in listings, but they are expanded during compilation.

END Microsoft Specific

See Also
Reference
Using Assembly Language in __asm Blocks

Intel's MMX Instruction Set

Microsoft Specific

The Visual C++ compiler allows you to use Intel's MMX (multimedia extension) instruction set in the inline assembler. The MMX instructions are also supported by the debugger disassembly. The compiler generates a warning message if the function contains MMX instructions but does not contain an EMMS instruction to empty the multimedia state. For more information, see the Intel Web site.

END Microsoft Specific

See Also

Reference

Using Assembly Language in __asm Blocks

Using C or C++ in __asm Blocks

Microsoft Specific

Because inline assembly instructions can be mixed with C or C++ statements, they can refer to C or C++ variables by name and use many other elements of those languages.

An **__asm** block can use the following language elements:

- Symbols, including labels and variable and function names
- Constants, including symbolic constants and **enum** members
- Macros and preprocessor directives
- Comments (both **/* */** and **//**)
- Type names (wherever a MASM type would be legal)
- **typedef** names, generally used with operators such as **PTR** and **TYPE** or to specify structure or union members

Within an **__asm** block, you can specify integer constants with either C notation or assembler radix notation (0x100 and 100h are equivalent, for example). This allows you to define (using **#define**) a constant in C and then use it in both C or C++ and assembly portions of the program. You can also specify constants in octal by preceding them with a 0. For example, 0777 specifies an octal constant.

What do you want to know more about?

- Using Operators in __asm Blocks
- Using C or C++ Symbols_in __asm Blocks
- Accessing C or C++ Data in __asm Blocks
- Writing Functions with Inline Assembly

END Microsoft Specific

See Also
Reference
Inline Assembler

Using Operators in __asm Blocks

Microsoft Specific

An **__asm** block cannot use C or C++ specific operators, such as the **<<** operator. However, operators shared by C and MASM, such as the * operator, are interpreted as assembly-language operators. For instance, outside an **__asm** block, square brackets (**[ ]**) are interpreted as enclosing array subscripts, which C automatically scales to the size of an element in the array. Inside an **__asm** block, they are seen as the MASM index operator, which yields an unscaled byte offset from any data object or label (not just an array). The following code illustrates the difference:

```
int array[10];

__asm mov array[6], bx ;  Store BX at array+6 (not scaled)

array[6] = 0;         /* Store 0 at array+24 (scaled) */
```

The first reference to `array` is not scaled, but the second is. Note that you can use the **TYPE** operator to achieve scaling based on a constant. For example, the following statements are equivalent:

```
__asm mov array[6 * TYPE int], 0 ; Store 0 at array + 24

array[6] = 0;                   /* Store 0 at array + 24 */
```

END Microsoft Specific

See Also
Reference
Using C or C++ in __asm Blocks

Using C or C++ Symbols in __asm Blocks

Microsoft Specific

An **__asm** block can refer to any C or C++ symbol in scope where the block appears. (C and C++ symbols are variable names, function names, and labels; that is, names that aren't symbolic constants or **enum** members. You cannot call C++ member functions.)

A few restrictions apply to the use of C and C++ symbols:

- Each assembly-language statement can contain only one C or C++ symbol. Multiple symbols can appear in the same assembly instruction only with **LENGTH**, **TYPE**, and **SIZE** expressions.
- Functions referenced in an **__asm** block must be declared (prototyped) earlier in the program. Otherwise, the compiler cannot distinguish between function names and labels in the **__asm** block.
- An **__asm** block cannot use any C or C++ symbols with the same spelling as MASM reserved words (regardless of case). MASM reserved words include instruction names such as **PUSH** and register names such as SI.
- Structure and union tags are not recognized in **__asm** blocks.

END Microsoft Specific

See Also
Reference
Using C or C++ in __asm Blocks

Accessing C or C++ Data in __asm Blocks

Microsoft Specific

A great convenience of inline assembly is the ability to refer to C or C++ variables by name. An **__asm** block can refer to any symbols, including variable names, that are in scope where the block appears. For instance, if the C variable `var` is in scope, the instruction

```
__asm mov eax, var
```

stores the value of `var` in EAX.

If a class, structure, or union member has a unique name, an **__asm** block can refer to it using only the member name, without specifying the variable or **typedef** name before the period (**.**) operator. If the member name is not unique, however, you must place a variable or **typedef** name immediately before the period operator. For example, the structure types in the following sample share `same_name` as their member name:.

If you declare variables with the types

```
struct first_type hal;
struct second_type oat;
```

all references to the member `same_name` must use the variable name because `same_name` is not unique. But the member `weasel` has a unique name, so you can refer to it using only its member name:

```
// InlineAssembler_Accessing_C_asm_Blocks.cpp
// processor: x86
#include <stdio.h>
struct first_type
{
   char *weasel;
   int same_name;
};

struct second_type
{
   int wonton;
   long same_name;
};

int main()
{
   struct first_type hal;
   struct second_type oat;

   __asm
   {
      lea ebx, hal
      mov ecx, [ebx]hal.same_name ; Must use 'hal'
      mov esi, [ebx].weasel       ; Can omit 'hal'
   }
   return 0;
}
```

Note that omitting the variable name is merely a coding convenience. The same assembly instructions are generated whether or not the variable name is present.

You can access data members in C++ without regard to access restrictions. However, you cannot call member functions.

END Microsoft Specific

See Also
Reference

Writing Functions with Inline Assembly

Microsoft Specific

If you write a function with inline assembly code, it's easy to pass arguments to the function and return a value from it. The following examples compare a function first written for a separate assembler and then rewritten for the inline assembler. The function, called `power2`, receives two parameters, multiplying the first parameter by 2 to the power of the second parameter. Written for a separate assembler, the function might look like this:

```
; POWER.ASM
; Compute the power of an integer
;
       PUBLIC _power2
_TEXT SEGMENT WORD PUBLIC 'CODE'
_power2 PROC

        push ebp        ; Save EBP
        mov ebp, esp    ; Move ESP into EBP so we can refer
                        ;   to arguments on the stack
        mov eax, [ebp+4] ; Get first argument
        mov ecx, [ebp+6] ; Get second argument
        shl eax, cl     ; EAX = EAX * ( 2 ^ CL )
        pop ebp         ; Restore EBP
        ret             ; Return with sum in EAX

_power2 ENDP
_TEXT   ENDS
        END
```

Since it's written for a separate assembler, the function requires a separate source file and assembly and link steps. C and C++ function arguments are usually passed on the stack, so this version of the `power2` function accesses its arguments by their positions on the stack. (Note that the **MODEL** directive, available in MASM and some other assemblers, also allows you to access stack arguments and local stack variables by name.)

Example

This program writes the `power2` function with inline assembly code:

```
// Power2_inline_asm.c
// compile with: /EHsc
// processor: x86

#include <stdio.h>

int power2( int num, int power );

int main( void )
{
    printf_s( "3 times 2 to the power of 5 is %d\n", \
              power2( 3, 5) );
}
int power2( int num, int power )
{
   __asm
   {
      mov eax, num    ; Get first argument
      mov ecx, power  ; Get second argument
      shl eax, cl     ; EAX = EAX * ( 2 to the power of CL )
   }
   // Return with result in EAX
}
```

The inline version of the `power2` function refers to its arguments by name and appears in the same source file as the rest of the program. This version also requires fewer assembly instructions.

Because the inline version of `power2` doesn't execute a C **return** statement, it causes a harmless warning if you compile at warning level 2 or higher. The function does return a value, but the compiler cannot tell that in the absence of a **return** statement. You can use #pragma warning to disable the generation of this warning.

END Microsoft Specific

See Also

Reference

Using C or C++ in __asm Blocks

Using and Preserving Registers in Inline Assembly

Microsoft Specific

In general, you should not assume that a register will have a given value when an **__asm** block begins. Register values are not guaranteed to be preserved across separate **__asm** blocks. If you end a block of inline code and begin another, you cannot rely on the registers in the second block to retain their values from the first block. An **__asm** block inherits whatever register values result from the normal flow of control.

If you use the **__fastcall** calling convention, the compiler passes function arguments in registers instead of on the stack. This can create problems in functions with **__asm** blocks because a function has no way to tell which parameter is in which register. If the function happens to receive a parameter in EAX and immediately stores something else in EAX, the original parameter is lost. In addition, you must preserve the ECX register in any function declared with **__fastcall**.

To avoid such register conflicts, don't use the **__fastcall** convention for functions that contain an **__asm** block. If you specify the **__fastcall** convention globally with the /Gr compiler option, declare every function containing an **__asm** block with **__cdecl** or **__stdcall**. (The **__cdecl** attribute tells the compiler to use the C calling convention for that function.) If you are not compiling with /Gr, avoid declaring the function with the **__fastcall** attribute.

When using **__asm** to write assembly language in C/C++ functions, you don't need to preserve the EAX, EBX, ECX, EDX, ESI, or EDI registers. For example, in the POWER2.C example in Writing Functions with Inline Assembly, the `power2` function doesn't preserve the value in the EAX register. However, using these registers will affect code quality because the register allocator cannot use them to store values across **__asm** blocks. In addition, by using EBX, ESI or EDI in inline assembly code, you force the compiler to save and restore those registers in the function prologue and epilogue.

You should preserve other registers you use (such as DS, SS, SP, BP, and flags registers) for the scope of the **__asm** block. You should preserve the ESP and EBP registers unless you have some reason to change them (to switch stacks, for example). Also see Optimizing Inline Assembly.

Some SSE types require eight-byte stack alignment, forcing the compiler to emit dynamic stack-alignment code. To be able to access both the local variables and the function parameters after the alignment, the compiler maintains two frame pointers. If the compiler performs frame pointer omission (FPO), it will use EBP and ESP. If the compiler does not perform FPO, it will use EBX and EBP. To ensure code runs correctly, do not modify EBX in asm code if the function requires dynamic stack alignment as it could modify the frame pointer. Either move the eight-byte aligned types out of the function, or avoid using EBX.

Note
If your inline assembly code changes the direction flag using the STD or CLD instructions, you must restore the flag to its original value.

END Microsoft Specific

See Also

Reference

Inline Assembler

Jumping to Labels in Inline Assembly

Microsoft Specific

Like an ordinary C or C++ label, a label in an **__asm** block has scope throughout the function in which it is defined (not only in the block). Both assembly instructions and **goto** statements can jump to labels inside or outside the **__asm** block.

Labels defined in **__asm** blocks are not case sensitive; both **goto** statements and assembly instructions can refer to those labels without regard to case. C and C++ labels are case sensitive only when used by **goto** statements. Assembly instructions can jump to a C or C++ label without regard to case.

The following code shows all the permutations:

```
void func( void )
{
   goto C_Dest;  /* Legal: correct case   */
   goto c_dest;  /* Error: incorrect case */

   goto A_Dest;  /* Legal: correct case   */
   goto a_dest;  /* Legal: incorrect case */

   __asm
   {
      jmp C_Dest ; Legal: correct case
      jmp c_dest ; Legal: incorrect case

      jmp A_Dest ; Legal: correct case
      jmp a_dest ; Legal: incorrect case

      a_dest:    ; __asm label
   }

   C_Dest:       /* C label */
   return;
}
int main()
{
}
```

Don't use C library function names as labels in **__asm** blocks. For instance, you might be tempted to use `exit` as a label, as follows:

```
; BAD TECHNIQUE: using library function name as label
jne exit
   .
   .
   .
exit:
   ; More __asm code follows
```

Because **exit** is the name of a C library function, this code might cause a jump to the **exit** function instead of to the desired location.

As in MASM programs, the dollar symbol (**$**) serves as the current location counter. It is a label for the instruction currently being assembled. In **__asm** blocks, its main use is to make long conditional jumps:

```
jne $+5 ; next instruction is 5 bytes long
jmp farlabel
; $+5
   .
   .
   .
farlabel:
```

END Microsoft Specific

See Also

Reference

Inline Assembler

Calling C Functions in Inline Assembly

Microsoft Specific

An **__asm** block can call C functions, including C library routines. The following example calls the **printf** library routine:

```
// InlineAssembler_Calling_C_Functions_in_Inline_Assembly.cpp
// processor: x86
#include <stdio.h>

char format[] = "%s %s\n";
char hello[] = "Hello";
char world[] = "world";
int main( void )
{
   __asm
   {
      mov  eax, offset world
      push eax
      mov  eax, offset hello
      push eax
      mov  eax, offset format
      push eax
      call printf
      //clean up the stack so that main can exit cleanly
      //use the unused register ebx to do the cleanup
      pop  ebx
      pop  ebx
      pop  ebx
   }
}
```

Because function arguments are passed on the stack, you simply push the needed arguments — string pointers, in the previous example — before calling the function. The arguments are pushed in reverse order, so they come off the stack in the desired order. To emulate the C statement

```
printf( format, hello, world );
```

the example pushes pointers to `world`, `hello`, and `format`, in that order, and then calls **printf**.

END Microsoft Specific

See Also
Reference
Inline Assembler

Calling C++ Functions in Inline Assembly

Microsoft Specific

An **__asm** block can call only global C++ functions that are not overloaded. If you call an overloaded global C++ function or a C++ member function, the compiler issues an error.

You can also call any functions declared with **extern "C"** linkage. This allows an **__asm** block within a C++ program to call the C library functions, because all the standard header files declare the library functions to have **extern "C"** linkage.

END Microsoft Specific

See Also

Reference

Inline Assembler

Defining __asm Blocks as C Macros

Microsoft Specific

C macros offer a convenient way to insert assembly code into your source code, but they demand extra care because a macro expands into a single logical line. To create trouble-free macros, follow these rules:

- Enclose the **__asm** block in braces.
- Put the **__asm** keyword in front of each assembly instruction.
- Use old-style C comments (`/* comment */`) instead of assembly-style comments (`; comment`) or single-line C comments (`// comment`).

To illustrate, the following example defines a simple macro:

```
#define PORTIO __asm      \
/* Port output */         \
{                         \
   __asm mov al, 2        \
   __asm mov dx, 0xD007   \
   __asm out dx, al       \
}
```

At first glance, the last three **__asm** keywords seem superfluous. They are needed, however, because the macro expands into a single line:

```
__asm /* Port output */ { __asm mov al, 2  __asm mov dx, 0xD007 __asm out dx, al }
```

The third and fourth **__asm** keywords are needed as statement separators. The only statement separators recognized in **__asm** blocks are the newline character and **__asm** keyword. Because a block defined as a macro is one logical line, you must separate each instruction with **__asm**.

The braces are essential as well. If you omit them, the compiler can be confused by C or C++ statements on the same line to the right of the macro invocation. Without the closing brace, the compiler cannot tell where assembly code stops, and it sees C or C++ statements after the **__asm** block as assembly instructions.

Assembly-style comments that start with a semicolon (**;**) continue to the end of the line. This causes problems in macros because the compiler ignores everything after the comment, all the way to the end of the logical line. The same is true of single-line C or C++ comments (`// comment`). To prevent errors, use old-style C comments (`/* comment */`) in **__asm** blocks defined as macros.

An **__asm** block written as a C macro can take arguments. Unlike an ordinary C macro, however, an **__asm** macro cannot return a value. So you cannot use such macros in C or C++ expressions.

Be careful not to invoke macros of this type indiscriminately. For instance, invoking an assembly-language macro in a function declared with the **__fastcall** convention may cause unexpected results. (See Using and Preserving Registers in Inline Assembly.)

END Microsoft Specific

See Also
Reference
Inline Assembler

Optimizing Inline Assembly

Microsoft Specific

The presence of an **__asm** block in a function affects optimization in several ways. First, the compiler doesn't try to optimize the **__asm** block itself. What you write in assembly language is exactly what you get. Second, the presence of an **__asm** block affects register variable storage. The compiler avoids enregistering variables across an **__asm** block if the register's contents would be changed by the **__asm** block. Finally, some other function-wide optimizations will be affected by the inclusion of assembly language in a function.

END Microsoft Specific

See Also

Reference

Inline Assembler

Compiler COM Support

Microsoft Specific

The Visual C++ compiler can directly read component object model (COM) type libraries and translate the contents into C++ source code that can be included in the compilation. Language extensions are available to facilitate COM programming on the client side.

By using the #import preprocessor directive, the compiler can read a type library and convert it into a C++ header file that describes the COM interfaces as classes. A set of **#import** attributes is available for user control of the content for the resulting type library header files.

You can use the __declspec extended attribute uuid to assign a globally unique identifier (GUID) to a COM object. The keyword __uuidof can be used to extract the GUID associated with a COM object. Another **__declspec** attribute, property, can be used to specify the **get** and **set** methods for a data member of a COM object.

A set of COM support global functions and classes is provided to support the **VARIANT** and **BSTR** types, implement smart pointers, and encapsulate the error object thrown by **_com_raise_error**:

- Compiler COM Global Functions
- _bstr_t
- _com_error
- _com_ptr_t
- _variant_t

END Microsoft Specific

See Also

Reference

Compiler COM Support Classes

Compiler COM Global Functions

Compiler COM Global Functions

Microsoft Specific

The following routines are available:

Function	Description
_com_raise_error	Throws a _com_error in response to a failure.
ConvertBSTRToString	Converts a **BSTR** value to a **char ***.
ConvertStringToBSTR	Converts a **char *** value to a **BSTR**.

END Microsoft Specific

See Also

Reference

Compiler COM Support Classes

Compiler COM Support

_com_raise_error

Microsoft Specific

Throws a _com_error in response to a failure.

```
void __stdcall _com_raise_error(
   HRESULT hr,
   IErrorInfo* perrinfo = 0
);
```

Parameters

hr

HRESULT information.

perrinfo

IErrorInfo object.

Remarks

_com_raise_error, which is defined in comdef.h, can be replaced by a user-written version of the same name and prototype. This could be done if you want to use **#import** but do not want to use C++ exception handling. In that case, a user version of **_com_raise_error** might decide to do a **longjmp** or display a message box and halt. The user version should not return, though, because the compiler COM support code does not expect it to return.

By default, **_com_raise_error** is defined as follows:

```
void __stdcall _com_raise_error(HRESULT hr, IErrorInfo* perrinfo) {
   throw _com_error(hr, perrinfo);
}
```

END Microsoft Specific

Requirements

Header: comdef.h

Lib: comsuppw.lib or comsuppwd.lib (see /Zc:wchar_t (wchar_t Is Native Type) for more information)

See Also

Reference

Compiler COM Global Functions

ConvertStringToBSTR

Microsoft Specific

Converts a **char *** value to a **BSTR**.

```
BSTR __stdcall ConvertStringToBSTR(
   const char* pSrc
)
```

Parameters

pSrc

A **char *** variable.

Example

```
// ConvertStringToBSTR.cpp
#include <comutil.h>
#include <stdio.h>

#pragma comment(lib, "comsuppw.lib")
#pragma comment(lib, "kernel32.lib")

int main() {
   char* lpszText = "Test";
   printf_s("char * text: %s\n", lpszText);

   BSTR bstrText = _com_util::ConvertStringToBSTR(lpszText);
   wprintf_s(L"BSTR text: %s\n", bstrText);

   SysFreeString(bstrText);
}
```

Output

```
char * text: Test
BSTR text: Test
```

END Microsoft Specific

Requirements

Header: comutil.h

Lib: comsuppw.lib or comsuppwd.lib (see /Zc:wchar_t (wchar_t Is Native Type) for more information)

See Also

Reference

Compiler COM Global Functions

ConvertBSTRToString

Microsoft Specific

Converts a **BSTR** value to a **char ***.

```
char* __stdcall ConvertBSTRToString(
   BSTR pSrc
);
```

Parameters

pSrc

A BSTR variable.

Remarks

ConvertBSTRToString allocates a string you must delete.

Example

```
// ConvertBSTRToString.cpp
#include <comutil.h>
#include <stdio.h>

#pragma comment(lib, "comsuppw.lib")

int main() {
   BSTR bstrText = ::SysAllocString(L"Test");
   wprintf_s(L"BSTR text: %s\n", bstrText);

   char* lpszText2 = _com_util::ConvertBSTRToString(bstrText);
   printf_s("char * text: %s\n", lpszText2);

   SysFreeString(bstrText);
   delete[] lpszText2;
}
```

Output

```
BSTR text: Test
char * text: Test
```

END Microsoft Specific

Requirements

Header: comutil.h.

Lib: comsuppw.lib or comsuppwd.lib (see /Zc:wchar_t (wchar_t Is Native Type) for more information)

See Also

Reference

Compiler COM Global Functions

Compiler COM Support Classes

Microsoft Specific

Standard classes are used to support some of the COM types. The classes are defined in comdef.h and the header files generated from the type library.

Class	Purpose
_bstr_t	Wraps the **BSTR** type to provide useful operators and methods.
_com_error	Defines the error object thrown by _com_raise_error in most failures.
_com_ptr_t	Encapsulates COM interface pointers, and automates the required calls to **AddRef**, **Release**, and **QueryInterface**.
_variant_t	Wraps the **VARIANT** type to provide useful operators and methods.

END Microsoft Specific

See Also

Reference

Compiler COM Support

Compiler COM Global Functions

Other Resources

C++ Language Reference

_bstr_t Class

Microsoft Specific

A **_bstr_t** object encapsulates the **BSTR** data type. The class manages resource allocation and deallocation through function calls to **SysAllocString** and **SysFreeString** and other **BSTR** APIs when appropriate. The **_bstr_t** class uses reference counting to avoid excessive overhead.

Construction

_bstr_t	Constructs a **_bstr_t** object.

Operations

Assign	Copies a **BSTR** into the **BSTR** wrapped by a **_bstr_t**.
Attach	Links a **_bstr_t** wrapper to a **BSTR**.
copy	Constructs a copy of the encapsulated **BSTR**.
Detach	Returns the **BSTR** wrapped by a **_bstr_t** and detaches the **BSTR** from the **_bstr_t**.
GetAddress	Points to the **BSTR** wrapped by a **_bstr_t**.
GetBSTR	Points to the beginning of the **BSTR** wrapped by the **_bstr_t**.
length	Returns the number of characters in the **_bstr_t**.

Operators

operator =	Assigns a new value to an existing **_bstr_t** object.
operator +=	Appends characters to the end of the **_bstr_t** object.
operator +	Concatenates two strings.
operator !	Checks if the encapsulated **BSTR** is a **NULL** string.
operator ==, !=, <, >, <=, >=	Compares two **_bstr_t** objects.
operator wchar_t* | char*	Extract the pointers to the encapsulated Unicode or multibyte **BSTR** object.

END Microsoft Specific

Requirements

Header: comutil.h

Lib: comsuppw.lib or comsuppwd.lib (see /Zc:wchar_t (wchar_t Is Native Type) for more information)

See Also

Reference

Compiler COM Support Classes

_bstr_t Member Functions

For information about **_bstr_t** member functions, see _bstr_t Class.

See Also

Reference

_bstr_t Class

_bstr_t::Assign

Microsoft Specific

Copies a **BSTR** into the **BSTR** wrapped by a **_bstr_t**.

```
void Assign(
   BSTR s
);
```

Parameters

s

A **BSTR** to be copied into the **BSTR** wrapped by a **_bstr_t**.

Remarks

Assign does a binary copy, which means the entire length of the **BSTR** is copied, regardless of content.

Example

```
// _bstr_t_Assign.cpp

#include <comdef.h>
#include <stdio.h>

int main()
{
    // creates a _bstr_t wrapper
    _bstr_t bstrWrapper;

    // creates BSTR and attaches to it
    bstrWrapper = "some text";
    wprintf_s(L"bstrWrapper = %s\n",
              static_cast<wchar_t*>(bstrWrapper));

    // bstrWrapper releases its BSTR
    BSTR bstr = bstrWrapper.Detach();
    wprintf_s(L"bstrWrapper = %s\n",
              static_cast<wchar_t*>(bstrWrapper));
    // "some text"
    wprintf_s(L"bstr = %s\n", bstr);

    bstrWrapper.Attach(SysAllocString(OLESTR("SysAllocedString")));
    wprintf_s(L"bstrWrapper = %s\n",
              static_cast<wchar_t*>(bstrWrapper));

    // assign a BSTR to our _bstr_t
    bstrWrapper.Assign(bstr);
    wprintf_s(L"bstrWrapper = %s\n",
              static_cast<wchar_t*>(bstrWrapper));

    // done with BSTR, do manual cleanup
    SysFreeString(bstr);

    // resuse bstr
    bstr= SysAllocString(OLESTR("Yet another string"));
    // two wrappers, one BSTR
    _bstr_t bstrWrapper2 = bstrWrapper;

    *bstrWrapper.GetAddress() = bstr;

    // bstrWrapper and bstrWrapper2 do still point to BSTR
    bstr = 0;
    wprintf_s(L"bstrWrapper = %s\n",
```

```
                static_cast<wchar_t*>(bstrWrapper));
    wprintf_s(L"bstrWrapper2 = %s\n",
                static_cast<wchar_t*>(bstrWrapper2));

    // new value into BSTR
    _snwprintf_s(bstrWrapper.GetBSTR(), 100, bstrWrapper.length(),
                 L"changing BSTR");
    wprintf_s(L"bstrWrapper = %s\n",
                static_cast<wchar_t*>(bstrWrapper));
    wprintf_s(L"bstrWrapper2 = %s\n",
                static_cast<wchar_t*>(bstrWrapper2));
}
```

Output

```
bstrWrapper = some text
bstrWrapper = (null)
bstr = some text
bstrWrapper = SysAllocedString
bstrWrapper = some text
bstrWrapper = Yet another string
bstrWrapper2 = some text
bstrWrapper = changing BSTR
bstrWrapper2 = some text
```

END Microsoft Specific

See Also

Reference

_bstr_t Class

_bstr_t::Attach

Microsoft Specific

Links a **_bstr_t** wrapper to a **BSTR**.

```
void Attach(
   BSTR s
);
```

Parameters

s

A **BSTR** to be associated with, or assigned to, the **_bstr_t** variable.

Remarks

If the **_bstr_t** was previously attached to another **BSTR**, the **_bstr_t** will clean up the **BSTR** resource, if no other **_bstr_t** variables are using the **BSTR**.

Example

See _bstr_t::Assign for an example using **Attach**.

END Microsoft Specific

See Also

Reference

_bstr_t Class

_bstr_t::_bstr_t

Microsoft Specific

Constructs a **_bstr_t** object.

```
_bstr_t( ) throw( );
_bstr_t(
   const _bstr_t& s1
) throw( );
_bstr_t(
   const char* s2
);
_bstr_t(
   const wchar_t* s3
);
_bstr_t(
   const _variant_t& var
);
_bstr_t(
   BSTR bstr,
   bool fCopy
);
```

Parameters

s1

A **_bstr_t** object to be copied.

s2

A multibyte string.

s3

A Unicode string

var

A **_variant_t** object.

bstr

An existing **BSTR** object.

fCopy

If false, the *bstr* argument is attached to the new object without making a copy by calling **SysAllocString**.

Remarks

- **_bstr_t( )** Constructs a default **_bstr_t** object that encapsulates a **NULL BSTR** object.
- **_bstr_t(_bstr_t&** *s1* **)** Constructs a **_bstr_t** object as a copy of another. This is a "shallow" copy, which increments the reference count of the encapsulated **BSTR** object instead of creating a new one.
- **_bstr_t(char*** *s2* **)** Constructs a **_bstr_t** object by calling **SysAllocString** to create a new **BSTR** object and encapsulate it. This constructor first performs a multibyte to Unicode conversion.
- **_bstr_t(wchar_t*** *s3* **)** Constructs a **_bstr_t** object by calling **SysAllocString** to create a new **BSTR** object and encapsulates it.
- **_bstr_t(_variant_t&** *var* **)** Constructs a **_bstr_t** object from a _variant_t object by first retrieving a **BSTR** object from the encapsulated **VARIANT** object.
- **_bstr_t(BSTR** *bstr* **| bool** *fCopy* **)** Constructs a **_bstr_t** object from an existing **BSTR** (as opposed to a **wchar_t*** string). If *fCopy* is **false**, the supplied **BSTR** is attached to the new object without making a new copy with **SysAllocString**. This

is the method used by the wrapper functions in the type library headers to encapsulate and take ownership of a **BSTR**, returned by an interface method, in a **_bstr_t** object.

END Microsoft Specific

See Also
Reference
_bstr_t Class

_bstr_t::copy

Microsoft Specific

Constructs a copy of the encapsulated **BSTR**.

```
BSTR copy(
  bool fCopy = true
) const;
```

Parameters

fCopy

If **true**, **copy** returns a copy of the contained **BSTR**, otherwise **copy** returns the actual BSTR.

Remarks

Returns a newly allocated copy of the encapsulated **BSTR** object.

Example

```
STDMETHODIMP CAlertMsg::get_ConnectionStr(BSTR *pVal){ //  m_bsConStr is _bstr_t
   *pVal = m_bsConStr.copy();
}
```

END Microsoft Specific

See Also

Reference

_bstr_t Class

_bstr_t::Detach

Microsoft Specific

Returns the **BSTR** wrapped by a **_bstr_t** and detaches the **BSTR** from the **_bstr_t**.

```
BSTR Detach( ) throw;
```

Return Value

The **BSTR** wrapped by the **_bstr_t**.

Example

See _bstr_t::Assign for a example using **Detach**.

END Microsoft Specific

See Also

Reference

_bstr_t Class

_bstr_t::GetAddress

Microsoft Specific

Points to the **BSTR** wrapped by a **_bstr_t**.

```
BSTR* GetAddress( );
```

Return Value

A pointer to the **BSTR** wrapped by the **_bstr_t**.

Remarks

GetAddress affects all **_bstr_t** objects that share a **BSTR**. More than one **_bstr_t** can share a **BSTR** through the use of the copy constructor and and **operator=**.

Example

See _bstr_t::Assign for a example using **GetAddress**.

END Microsoft Specific

See Also
Reference
_bstr_t Class

_bstr_t::GetBSTR

Microsoft Specific

Points to the beginning of the **BSTR** wrapped by the **_bstr_t**.

```
BSTR& GetBSTR( );
```

Return Value

The beginning of the **BSTR** wrapped by the **_bstr_t**.

Remarks

GetBSTR affects all **_bstr_t** objects that share a **BSTR**. More than one **_bstr_t** can share a **BSTR** through the use of the copy constructor and and **operator=**.

Example

See _bstr_t::Assign for an example using **GetBSTR**.

END Microsoft Specific

See Also
Reference
_bstr_t Class

_bstr_t::length

Microsoft Specific

Returns the number of characters in the **_bstr_t**, not including the terminating null, of the encapsulated **BSTR**.

```
unsigned int length ( ) const throw( );
```

Remarks

END Microsoft Specific

See Also
Reference
_bstr_t Class

_bstr_t Operators

For information about the **_bstr_t** operators, see _bstr_t Class.

See Also

Reference

_bstr_t Class

_bstr_t::operator =

Microsoft Specific

Assigns a new value to an existing **_bstr_t** object.

```
_bstr_t& operator=(
   const _bstr_t& s1
) throw ( );
_bstr_t& operator=(
   const char* s2
);
_bstr_t& operator=(
   const wchar_t* s3
);
_bstr_t& operator=(
   const _variant_t& var
);
```

Parameters

s1

A **_bstr_t** object to be assigned to an existing **_bstr_t** object.

s2

A multibyte string to be assigned to an existing **_bstr_t** object.

s3

A Unicode string to be assigned to an existing **_bstr_t** object.

var

A **_variant_t** object to be assigned to an existing **_bstr_t** object.

END Microsoft Specific

Example

See _bstr_t::Assign for an example of using **operator=**.

See Also

Reference

_bstr_t Class

_bstr_t::operator +=, +

Microsoft Specific

Appends characters to the end of the **_bstr_t** object or concatenates two strings.

```
_bstr_t& operator+=(
   const _bstr_t& s1
);
_bstr_t operator+(
   const _bstr_t& s1
);
friend _bstr_t operator+(
   const char* s2,
   const _bstr_t& s1
);
friend _bstr_t operator+(
   const wchar_t* s3,
   const _bstr_t& s1
);
```

Parameters

s1

A **_bstr_t** object.

s2

A multibyte string.

s3

A Unicode string.

Remarks

These operators perform string concatenation:

- **operator+=(** *s1* **)** Appends the characters in the encapsulated **BSTR** of *s1* to the end of this object's encapsulated **BSTR**.
- **operator+(** *s1* **)** Returns the new **_bstr_t** that is formed by concatenating this object's **BSTR** with that of *s1*.
- **operator+(** *s2* **|** *s1* **)** Returns a new **_bstr_t** that is formed by concatenating a multibyte string *s2*, converted to Unicode, with the **BSTR** encapsulated in *s1*.
- **operator+(** *s3* **,** *s1* **)** Returns a new **_bstr_t** that is formed by concatenating a Unicode string *s3* with the **BSTR** encapsulated in *s1*.

END Microsoft Specific

See Also

Reference

_bstr_t Class

_bstr_t::operator !

Microsoft Specific

Checks if the encapsulated **BSTR** is a **NULL** string.

```
bool operator!( ) const throw( );
```

Return Value

It returns **true** if yes, **false** if not.

END Microsoft Specific

See Also
Reference
_bstr_t Class

_bstr_t Relational Operators

Microsoft Specific

Compares two **_bstr_t** objects.

```
bool operator!( ) const throw( );
bool operator==(
   const _bstr_t& str
) const throw( );
bool operator!=(
   const _bstr_t& str
) const throw( );
bool operator<(
   const _bstr_t& str
) const throw( );
bool operator>(
   const _bstr_t& str
) const throw( );
bool operator<=(
   const _bstr_t& str
) const throw( );
bool operator>=(
   const _bstr_t& str
) const throw( );
```

Remarks

These operators compare two **_bstr_t** objects lexicographically. The operators return **true** if the comparisons hold, otherwise return **false**.

END Microsoft Specific

See Also

Reference

_bstr_t Class

_bstr_t::wchar_t*, _bstr_t::char*

Microsoft Specific

Extract the pointers to the encapsulated Unicode or multibyte **BSTR** object.

```
operator const wchar_t*( ) const throw( );
operator wchar_t*( ) const throw( );
operator const char*( ) const;
operator char*( ) const;
```

Remarks

These operators can be used to extract raw pointers to the encapsulated Unicode or multibyte **BSTR** object. The operators return the pointer to the actual internal buffer, so the resulting string cannot be modified.

END Microsoft Specific

See Also

Reference

_bstr_t Class

_com_error Class

Microsoft Specific

A **_com_error** object represents an exception condition detected by the error-handling wrapper functions in the header files generated from the type library or by one of the COM support classes. The **_com_error** class encapsulates the **HRESULT** error code and any associated **IErrorInfo** object.

Construction

_com_error	Constructs a **_com_error** object.

Operators

operator =	Assigns an existing **_com_error** object to another.

Extractor Functions

Error	Retrieves the **HRESULT** passed to the constructor.
ErrorInfo	Retrieves the **IErrorInfo** object passed to the constructor.
WCode	Retrieves the 16-bit error code mapped into the encapsulated **HRESULT**.

IErrorInfo Functions

Description	Calls **IErrorInfo::GetDescription** function.
HelpContext	Calls **IErrorInfo::GetHelpContext** function.
HelpFile	Calls **IErrorInfo::GetHelpFile** function
Source	Calls **IErrorInfo::GetSource** function.
GUID	Calls **IErrorInfo::GetGUID** function.

Format Message Extractor

ErrorMessage	Retrieves the string message for HRESULT stored in the **_com_error** object.

ExepInfo.wCode to HRESULT Mappers

HRESULTToWCode	Maps 32-bit **HRESULT** to 16-bit **wCode**.
WCodeToHRESULT	Maps 16-bit **wCode** to 32-bit **HRESULT**.

END Microsoft Specific

Requirements

Header: comdef.h

Lib: comsuppw.lib or comsuppwd.lib (see /Zc:wchar_t (wchar_t Is Native Type) for more information)

See Also

Reference

Compiler COM Support Classes

_com_error Member Functions

For information about the **_com_error** member functions, see _com_error Class.

See Also

Reference

_com_error Class

_com_error::_com_error

Microsoft Specific

Constructs a **_com_error** object.

```
_com_error(
   HRESULT hr,
   IErrorInfo* perrinfo = NULL,
   bool fAddRef=false
) throw( );
_com_error(
   const _com_error& that
) throw( );
```

Parameters

hr

HRESULT information.

perrinfo

IErrorInfo object.

bool fAddRef=false

Causes the constructor to call AddRef on a non-null **IErrorInfo** interface. This provides for correct reference counting in the common case where ownership of the interface is passed into the **_com_error** object, such as:

```
throw _com_error(hr, perrinfo);
```

If you do not want your code to transfer ownership to the **_com_error** object, and the **AddRef** is required to offset the **Release** in the **_com_error** destructor, construct the object as follows:

```
_com_error err(hr, perrinfo, true);
```

that

An existing **_com_error** object.

Remarks

The first constructor creates a new object given an **HRESULT** and optional **IErrorInfo** object. The second creates a copy of an existing **_com_error** object.

END Microsoft Specific

See Also

Reference

_com_error Class

_com_error::Description

Microsoft Specific

Calls **IErrorInfo::GetDescription** function.

```
_bstr_t Description( ) const;
```

Return Value

Returns the result of **IErrorInfo::GetDescription** for the **IErrorInfo** object recorded within the **_com_error** object. The resulting **BSTR** is encapsulated in a **_bstr_t** object. If no **IErrorInfo** is recorded, it returns an empty **_bstr_t**.

Remarks

Calls the **IErrorInfo::GetDescription** function and retrieves **IErrorInfo** recorded within the **_com_error** object. Any failure while calling the **IErrorInfo::GetDescription** method is ignored.

END Microsoft Specific

See Also

Reference

_com_error Class

_com_error::Error

Microsoft Specific

Retrieves the **HRESULT** passed to the constructor.

```
HRESULT Error( ) const throw( );
```

Return Value

Raw **HRESULT** item passed into the constructor.

Remarks

Retrieves the encapsulated **HRESULT** item in a **_com_error** object.

END Microsoft Specific

See Also

Reference

_com_error Class

_com_error::ErrorInfo

Microsoft Specific

Retrieves the **IErrorInfo** object passed to the constructor.

```
IErrorInfo * ErrorInfo( ) const throw( );
```

Return Value

Raw **IErrorInfo** item passed into the constructor.

Remarks

Retrieves the encapsulated **IErrorInfo** item in a **_com_error** object, or **NULL** if no **IErrorInfo** item is recorded. The caller must call **Release** on the returned object when finished using it.

END Microsoft Specific

See Also
Reference
_com_error Class

_com_error::ErrorMessage

Microsoft Specific

Retrieves the string message for **HRESULT** stored in the **_com_error** object.

```
const TCHAR * ErrorMessage( ) const throw( );
```

Return Value

Returns the string message for the **HRESULT** recorded within the **_com_error** object. If the **HRESULT** is a mapped 16-bit wCode, then a generic message "`IDispatch error #<wCode>`" is returned. If no message is found, then a generic message "`Unknown error #<hresult>`" is returned. The returned string is either a Unicode or multibyte string, depending on the state of the **_UNICODE** macro.

Remarks

Retrieves the appropriate system message text for **HRESULT** recorded within the **_com_error** object. The system message text is obtained by calling the Win32 FormatMessage function. The string returned is allocated by the **FormatMessage** API, and it is released when the **_com_error** object is destroyed.

END Microsoft Specific

See Also
Reference
_com_error Class

_com_error::GUID

Microsoft Specific

Calls **IErrorInfo::GetGUID** function.

```
GUID GUID( ) const throw( );
```

Return Value

Returns the result of **IErrorInfo::GetGUID** for the **IErrorInfo** object recorded within the **_com_error** object. If no **IErrorInfo** object is recorded, it returns **GUID_NULL**.

Remarks

Any failure while calling the **IErrorInfo::GetGUID** method is ignored.

END Microsoft Specific

See Also
Reference
_com_error Class

_com_error::HelpContext

Microsoft Specific

Calls **IErrorInfo::GetHelpContext** function.

```
DWORD HelpContext( ) const throw( );
```

Return Value

Returns the result of **IErrorInfo::GetHelpContext** for the **IErrorInfo** object recorded within the **_com_error** object. If no **IErrorInfo** object is recorded, it returns a zero.

Remarks

Any failure while calling the **IErrorInfo::GetHelpContext** method is ignored.

END Microsoft Specific

See Also
Reference
_com_error Class

_com_error::HelpFile

Microsoft Specific

Calls **IErrorInfo::GetHelpFile** function.

```
_bstr_t HelpFile() const;
```

Return Value

Returns the result of **IErrorInfo::GetHelpFile** for the **IErrorInfo** object recorded within the **_com_error** object. The resulting BSTR is encapsulated in a **_bstr_t** object. If no **IErrorInfo** is recorded, it returns an empty **_bstr_t**.

Remarks

Any failure while calling the **IErrorInfo::GetHelpFile** method is ignored.

END Microsoft Specific

See Also

Reference

_com_error Class

_com_error::HRESULTToWCode

Microsoft Specific

Maps 32-bit **HRESULT** to 16-bit **wCode**.

```
static WORD HRESULTToWCode(
   HRESULT hr
) throw( );
```

Parameters

hr

The 32-bit **HRESULT** to be mapped to 16-bit **wCode**.

Return Value

16-bit **wCode** mapped from the 32-bit **HRESULT**.

Remarks

See _com_error::WCode for more information.

END Microsoft Specific

See Also

Reference

_com_error::WCode
_com_error::WCodeToHRESULT
_com_error Class

_com_error::Source

Microsoft Specific

Calls **IErrorInfo::GetSource** function.

```
_bstr_t Source() const;
```

Return Value

Returns the result of **IErrorInfo::GetSource** for the **IErrorInfo** object recorded within the **_com_error** object. The resulting BSTR is encapsulated in a **_bstr_t** object. If no **IErrorInfo** is recorded, it returns an empty **_bstr_t**.

Remarks

Any failure while calling the **IErrorInfo::GetSource** method is ignored.

END Microsoft Specific

See Also

Reference

_com_error Class

_com_error::WCode

Microsoft Specific

Retrieves the 16-bit error code mapped into the encapsulated **HRESULT**.

```
WORD WCode ( ) const throw( );
```

Return Value

If the **HRESULT** is within the range 0x80040200 to 0x8004FFFF, the **WCode** method returns the **HRESULT** minus 0x80040200; otherwise, it returns zero.

Remarks

The **WCode** method is used to undo a mapping that happens in the COM support code. The wrapper for a **dispinterface** property or method calls a support routine that packages the arguments and calls **IDispatch::Invoke**. Upon return, if a failure **HRESULT** of **DISP_E_EXCEPTION** is returned, the error information is retrieved from the **EXCEPINFO** structure passed to **IDispatch::Invoke**. The error code can either be a 16-bit value stored in the **wCode** member of the **EXCEPINFO** structure or a full 32-bit value in the **scode** member of the **EXCEPINFO** structure. If a 16-bit **wCode** is returned, it must first be mapped to a 32-bit failure **HRESULT**.

END Microsoft Specific

See Also

Reference

_com_error::HRESULTToWCode
_com_error::WCodeToHRESULT
_com_error Class

_com_error::WCodeToHRESULT

Microsoft Specific

Maps 16-bit **wCode** to 32-bit **HRESULT**.

```
static HRESULT WCodeToHRESULT(
   WORD wCode
) throw( );
```

Parameters

wCode

The 16-bit **wCode** to be mapped to 32-bit **HRESULT**.

Return Value

32-bit **HRESULT** mapped from the 16-bit **wCode**.

Remarks

See the WCode member function.

END Microsoft Specific

See Also

Reference

_com_error::WCode
_com_error::HRESULTToWCode
_com_error Class

_com_error Operators

For information about the **_com_error** operators, see _com_error Class.

See Also

Reference

_com_error Class

_com_error::operator =

Microsoft Specific

Assigns an existing **_com_error** object to another.

```
_com_error& operator = (
   const _com_error& that
) throw ( );
```

Parameters

that

A **_com_error** object.

END Microsoft Specific

See Also

Reference

_com_error Class

_com_ptr_t Class

Microsoft Specific

A **_com_ptr_t** object encapsulates a COM interface pointer and is called a "smart" pointer. This template class manages resource allocation and deallocation through function calls to the **IUnknown** member functions: **QueryInterface**, **AddRef**, and **Release**.

A smart pointer is usually referenced by the typedef definition provided by the **_COM_SMARTPTR_TYPEDEF** macro. This macro takes an interface name and the IID and declares a specialization of **_com_ptr_t** with the name of the interface plus a suffix of **Ptr**. For example:

```
_COM_SMARTPTR_TYPEDEF(IMyInterface, __uuidof(IMyInterface));
```

declares the **_com_ptr_t** specialization **IMyInterfacePtr**.

A set of function templates, not members of this template class, support comparisons with a smart pointer on the right side of the comparison operator.

Construction

_com_ptr_t	Constructs a **_com_ptr_t** object.

Low-Level Operations

AddRef	Calls the **AddRef** member function of **IUnknown** on the encapsulated interface pointer.
Attach	Encapsulates a raw interface pointer of this smart pointer's type.
CreateInstance	Creates a new instance of an object given a **CLSID** or **ProgID**.
Detach	Extracts and returns the encapsulated interface pointer.
GetActiveObject	Attaches to an existing instance of an object given a **CLSID** or **ProgID**.
GetInterfacePtr	Returns the encapsulated interface pointer.
QueryInterface	Calls the **QueryInterface** member function of **IUnknown** on the encapsulated interface pointer.
Release	Calls the **Release** member function of **IUnknown** on the encapsulated interface pointer.

Operators

operator =	Assigns a new value to an existing **_com_ptr_t** object.
operators ==, !=, <, >, <=, >=	Compare the smart pointer object to another smart pointer, raw interface pointer, or **NULL**.
Extractors	Extract the encapsulated COM interface pointer.

END Microsoft Specific

Requirements

Header: comip.h

Lib: comsuppw.lib or comsuppwd.lib (see /Zc:wchar_t (wchar_t Is Native Type) for more information)

See Also

Reference

Compiler COM Support Classes

_com_ptr_t Member Functions

For information about the **_com_ptr_t** member functions, see _com_ptr_t Class.

See Also

Reference

_com_ptr_t Class

_com_ptr_t::_com_ptr_t

Microsoft Specific

Constructs a **_com_ptr_t** object.

```
// Default constructor.
// Constructs a NULL smart pointer.
_com_ptr_t() throw();

// Constructs a NULL smart pointer. The NULL argument must be zero.
_com_ptr_t(
   int null
);

// Constructs a smart pointer as a copy of another instance of the
// same smart pointer. AddRef is called to increment the reference
// count for the encapsulated interface pointer.
_com_ptr_t(
   const _com_ptr_t& cp
) throw();

// Constructs a smart pointer from a raw interface pointer of this
// smart pointer's type. If fAddRef is true, AddRef is called
// to increment the reference count for the encapsulated
// interface pointer. If fAddRef is false, this constructor
// takes ownership of the raw interface pointer without calling AddRef.
_com_ptr_t(
   Interface* pInterface,
   bool fAddRef
) throw();

// Construct pointer for a _variant_t object.
// Constructs a smart pointer from a _variant_t object. The
// encapsulated VARIANT must be of type VT_DISPATCH or VT_UNKNOWN, or
// it can be converted into one of these two types. If QueryInterface
// fails with an E_NOINTERFACE error, a NULL smart pointer is
// constructed.
_com_ptr_t(
   const _variant_t& varSrc
);

// Constructs a smart pointer given the CLSID of a coclass. This
// function calls CoCreateInstance, by the member function
//  CreateInstance, to create a new COM object and then queries for
// this smart pointer's interface type. If QueryInterface fails with
// an E_NOINTERFACE error, a NULL smart pointer is constructed.
explicit _com_ptr_t(
   const CLSID& clsid,
   IUnknown* pOuter = NULL,
   DWORD dwClsContext = CLSCTX_ALL
);

// Calls CoCreateClass with provided CLSID retrieved from string.
explicit _com_ptr_t(
   LPCWSTR str,
   IUnknown* pOuter = NULL,
   DWORD dwClsContext = CLSCTX_ALL
);

// Constructs a smart pointer given a multibyte character string that
// holds either a CLSID (starting with "{") or a ProgID. This function
// calls CoCreateInstance, by the member function CreateInstance, to
// create a new COM object and then queries for this smart pointer's
// interface type. If QueryInterface fails with an E_NOINTERFACE error,
```

```
// a NULL smart pointer is constructed.
explicit _com_ptr_t(
   LPCSTR str,
   IUnknown* pOuter = NULL,
   DWORD dwClsContext = CLSCTX_ALL
);

// Saves the interface.
template<>
_com_ptr_t(
   Interface* pInterface
) throw();

// Make sure correct ctor is called
template<>
_com_ptr_t(
   LPSTR str
);

// Make sure correct ctor is called
template<>
_com_ptr_t(
   LPWSTR str
);

// Constructs a smart pointer from a different smart pointer type or
// from a different raw interface pointer. QueryInterface is called to
// find an interface pointer of this smart pointer's type. If
// QueryInterface fails with an E_NOINTERFACE error, a NULL smart
// pointer is constructed.
template<typename _OtherIID>
_com_ptr_t(
   const _com_ptr_t<_OtherIID>& p
);

// Constructs a smart-pointer from any IUnknown-based interface pointer.
template<typename _InterfaceType>
_com_ptr_t(
   _InterfaceType* p
);

// Disable conversion using _com_ptr_t* specialization of
// template<typename _InterfaceType> _com_ptr_t(_InterfaceType* p)
template<>
explicit _com_ptr_t(
   _com_ptr_t* p
);
```

Parameters

pInterface

A raw interface pointer.

fAddRef

If **true**, **AddRef** is called to increment the reference count of the encapsulated interface pointer.

cp

A **_com_ptr_t** object.

p

A raw interface pointer, its type being different from the smart pointer type of this **_com_ptr_t** object.

varSrc

A **_variant_t** object.

clsid

The **CLSID** of a coclass.

dwClsContext

Context for running executable code.

lpcStr

A multibyte string that holds either a **CLSID** (starting with "**{**") or a **ProgID**.

pOuter

The outer unknown for aggregation.

See Also

Reference

_com_ptr_t Class

_com_ptr_t::AddRef

Microsoft Specific

Calls the **AddRef** member function of **IUnknown** on the encapsulated interface pointer.

```
void AddRef( );
```

Remarks

Calls **IUnknown::AddRef** on the encapsulated interface pointer, raising an **E_POINTER** error if the pointer is **NULL**.

END Microsoft Specific

See Also

Reference

_com_ptr_t Class

_com_ptr_t::Attach

Microsoft Specific

Encapsulates a raw interface pointer of this smart pointer's type.

```
void Attach(
   Interface* pInterface
) throw( );
void Attach(
   Interface* pInterface,
   bool fAddRef
) throw( );
```

Parameters

pInterface

A raw interface pointer.

fAddRef

If it is **true**, then **AddRef** is called. If it is **false**, the **_com_ptr_t** object takes ownership of the raw interface pointer without calling **AddRef**.

Remarks

- **Attach(** *pInterface* **)** **AddRef** is not called. The ownership of the interface is passed to this **_com_ptr_t** object. **Release** is called to decrement the reference count for the previously encapsulated pointer.
- **Attach(** *pInterface* **,** *fAddRef* **)** If *fAddRef* is **true**, **AddRef** is called to increment the reference count for the encapsulated interface pointer. If *fAddRef* is **false**, this **_com_ptr_t** object takes ownership of the raw interface pointer without calling **AddRef**. **Release** is called to decrement the reference count for the previously encapsulated pointer.

END Microsoft Specific

See Also

Reference

_com_ptr_t Class

_com_ptr_t::CreateInstance

Microsoft Specific

Creates a new instance of an object given a **CLSID** or **ProgID**.

```
HRESULT CreateInstance(
   const CLSID& rclsid,
   IUnknown* pOuter=NULL,
   DWORD dwClsContext = CLSCTX_ALL
) throw( );
HRESULT CreateInstance(
   LPCWSTR clsidString,
   IUnknown* pOuter=NULL,
   DWORD dwClsContext = CLSCTX_ALL
) throw( );
HRESULT CreateInstance(
   LPCSTR clsidStringA,
   IUnknown* pOuter=NULL,
   DWORD dwClsContext = CLSCTX_ALL
) throw( );
```

Parameters

rclsid

The **CLSID** of an object.

clsidString

A Unicode string that holds either a **CLSID** (starting with "**{**") or a **ProgID**.

clsidStringA

A multibyte string, using the ANSI code page, that holds either a **CLSID** (starting with "**{**") or a **ProgID**.

dwClsContext

Context for running executable code.

pOuter

The outer unknown for aggregation.

Remarks

These member functions call **CoCreateInstance** to create a new COM object and then queries for this smart pointer's interface type. The resulting pointer is then encapsulated within this **_com_ptr_t** object. **Release** is called to decrement the reference count for the previously encapsulated pointer. This routine returns the **HRESULT** to indicate success or failure.

- **CreateInstance(** *rclsid* **,** *dwClsContext* **)** Creates a new running instance of an object given a **CLSID**.
- **CreateInstance(** *clsidString* **,** *dwClsContext* **)** Creates a new running instance of an object given a Unicode string that holds either a **CLSID** (starting with "**{**") or a **ProgID**.
- **CreateInstance(** *clsidStringA* **,** *dwClsContext* **)** Creates a new running instance of an object given a multibyte character string that holds either a **CLSID** (starting with "**{**") or a **ProgID**. Calls MultiByteToWideChar, which assumes that the string is in the ANSI code page rather than an OEM code page.

END Microsoft Specific

See Also

Reference

_com_ptr_t Class

_com_ptr_t::Detach

Microsoft Specific

Extracts and returns the encapsulated interface pointer.

```
Interface* Detach( ) throw( );
```

Remarks

Extracts and returns the encapsulated interface pointer, and then clears the encapsulated pointer storage to **NULL**. This removes the interface pointer from encapsulation. It is up to you to call **Release** on the returned interface pointer.

END Microsoft Specific

See Also
Reference
_com_ptr_t Class

_com_ptr_t::GetActiveObject

Microsoft Specific

Attaches to an existing instance of an object given a **CLSID** or **ProgID**.

```
HRESULT GetActiveObject(
   const CLSID& rclsid
) throw( );
HRESULT GetActiveObject(
   LPCWSTR clsidString
) throw( );
HRESULT GetActiveObject(
   LPCSTR clsidStringA
) throw( );
```

Parameters

rclsid

The **CLSID** of an object.

clsidString

A Unicode string that holds either a **CLSID** (starting with "**{**") or a **ProgID**.

clsidStringA

A multibyte string, using the ANSI code page, that holds either a **CLSID** (starting with "**{**") or a **ProgID**.

Remarks

These member functions call GetActiveObject to retrieve a pointer to a running object that has been registered with OLE and then queries for this smart pointer's interface type. The resulting pointer is then encapsulated within this **_com_ptr_t** object. **Release** is called to decrement the reference count for the previously encapsulated pointer. This routine returns the **HRESULT** to indicate success or failure.

- **GetActiveObject(** *rclsid* **)** Attaches to an existing instance of an object given a **CLSID**.
- **GetActiveObject(** *clsidString* **)** Attaches to an existing instance of an object given a Unicode string that holds either a **CLSID** (starting with "**{**") or a **ProgID**.
- **GetActiveObject(** *clsidStringA* **)** Attaches to an existing instance of an object given a multibyte character string that holds either a **CLSID** (starting with "**{**") or a **ProgID**. Calls MultiByteToWideChar, which assumes that the string is in the ANSI code page rather than an OEM code page.

END Microsoft Specific

See Also

Reference

_com_ptr_t Class

_com_ptr_t::GetInterfacePtr

Microsoft Specific

Returns the encapsulated interface pointer.

```
Interface* GetInterfacePtr( ) const throw( );
Interface*& GetInterfacePtr() throw();
```

Remarks

Returns the encapsulated interface pointer, which may be **NULL**.

END Microsoft Specific

See Also
Reference
_com_ptr_t Class

_com_ptr_t::QueryInterface

Microsoft Specific

Calls the **QueryInterface** member function of **IUnknown** on the encapsulated interface pointer.

```
template<typename _InterfaceType> HRESULT QueryInterface (
   const IID& iid,
   _InterfaceType*& p
) throw ( );
template<typename _InterfaceType> HRESULT QueryInterface (
   const IID& iid,
   _InterfaceType** p
) throw( );
```

Parameters

iid

IID of an interface pointer.

p

Raw interface pointer.

Remarks

Calls **IUnknown::QueryInterface** on the encapsulated interface pointer with the specified **IID** and returns the resulting raw interface pointer in *p*. This routine returns the **HRESULT** to indicate success or failure.

END Microsoft Specific

See Also

Reference

_com_ptr_t Class

_com_ptr_t::Release

Microsoft Specific

Calls the **Release** member function of **IUnknown** on the encapsulated interface pointer.

```
void Release( );
```

Remarks

Calls **IUnknown::Release** on the encapsulated interface pointer, raising an **E_POINTER** error if this interface pointer is **NULL**.

END Microsoft Specific

See Also

Reference

_com_ptr_t Class

_com_ptr_t Operators

For information about the **_com_ptr_t** operators, see _com_ptr_t Class.

See Also

Reference

_com_ptr_t Class

_com_ptr_t::operator =

Microsoft Specific

Assigns a new value to an existing **_com_ptr_t** object.

```
template<typename _OtherIID>
_com_ptr_t& operator=(
   const _com_ptr_t<_OtherIID>& p
);

// Sets a smart pointer to be a different smart pointer of a different
// type or a different raw interface pointer. QueryInterface is called
// to find an interface pointer of this smart pointer's type, and
// Release is called to decrement the reference count for the previously
// encapsulated pointer. If QueryInterface fails with an E_NOINTERFACE,
// a NULL smart pointer results.
template<typename _InterfaceType>
_com_ptr_t& operator=(
   _InterfaceType* p
);

// Encapsulates a raw interface pointer of this smart pointer's type.
// AddRef is called to increment the reference count for the encapsulated
// interface pointer, and Release is called to decrement the reference
// count for the previously encapsulated pointer.
template<> _com_ptr_t&
operator=(
   Interface* pInterface
) throw();

// Sets a smart pointer to be a copy of another instance of the same
// smart pointer of the same type. AddRef is called to increment the
// reference count for the encapsulated interface pointer, and Release
// is called to decrement the reference count for the previously
// encapsulated pointer.
_com_ptr_t& operator=(
   const _com_ptr_t& cp
) throw();

// Sets a smart pointer to NULL. The NULL argument must be a zero.
_com_ptr_t& operator=(
   int null
);
// Sets a smart pointer to be a _variant_t object. The encapsulated
// VARIANT must be of type VT_DISPATCH or VT_UNKNOWN, or it can be
// converted to one of these two types. If QueryInterface fails with an
// E_NOINTERFACE error, a NULL smart pointer results.
_com_ptr_t& operator=(
   const _variant_t& varSrc
);
```

Remarks

Assigns an interface pointer to this **_com_ptr_t** object.

END Microsoft Specific

See Also

Reference

_com_ptr_t Class

_com_ptr_t Relational Operators

Microsoft Specific

Compare the smart pointer object to another smart pointer, raw interface pointer, or **NULL**.

```
template<typename _OtherIID>
bool operator==(
   const _com_ptr_t<_OtherIID>& p
);

template<typename _OtherIID>
bool operator==(
   _com_ptr_t<_OtherIID>& p
);

template<typename _InterfaceType>
bool operator==(
   _InterfaceType* p
);

template<>
bool operator==(
   Interface* p
);

template<>
bool operator==(
   const _com_ptr_t& p
) throw();

template<>
bool operator==(
   _com_ptr_t& p
) throw();

bool operator==(
   int null
);
```

```
template<typename _OtherIID>
bool operator!=(
   const _com_ptr_t<_OtherIID>& p
);

template<typename _OtherIID>
bool operator!=(
   _com_ptr_t<_OtherIID>& p
);

template<typename _InterfaceType>
bool operator!=(
   _InterfaceType* p
);

bool operator!=(
   int null
);
```

```
template<typename _OtherIID>
```

```
bool operator<(
   const _com_ptr_t<_OtherIID>& p
);

template<typename _OtherIID>
bool operator<(
   _com_ptr_t<_OtherIID>& p
);

template<typename _InterfaceType>
bool operator<(
   _InterfaceType* p
);
```

```
template<typename _OtherIID>
bool operator>(
   const _com_ptr_t<_OtherIID>& p
);

template<typename _OtherIID>
bool operator>(_com_ptr_t<
   _OtherIID>& p
);

template<typename _InterfaceType>
bool operator>(
   _InterfaceType* p
);
```

```
template<typename _OtherIID>
bool operator<=(
   const _com_ptr_t<_OtherIID>& p
);

template<typename _OtherIID>
bool operator<=(
   _com_ptr_t<_OtherIID>& p
);

template<typename _InterfaceType>
bool operator<=(
   _InterfaceType* p
);
```

```
template<typename _OtherIID>
bool operator>=(
   const _com_ptr_t<_OtherIID>& p
);

template<typename _OtherIID>
bool operator>=(
   _com_ptr_t<_OtherIID>& p
);

template<typename _InterfaceType>
bool operator>=(
   _InterfaceType* p
);
```

Remarks

Compares a smart pointer object to another smart pointer, raw interface pointer, or **NULL**. Except for the **NULL** pointer tests, these operators first query both pointers for **IUnknown**, and compare the results.

END Microsoft Specific

See Also
Reference
_com_ptr_t Class

_com_ptr_t Extractors

Microsoft Specific

Extract the encapsulated COM interface pointer.

```
operator Interface*( ) const throw( );
operator Interface&( ) const;
Interface& operator*( ) const;
Interface* operator->( ) const;
Interface** operator&( ) throw( );
operator bool( ) const throw( );
```

Remarks

- **operator Interface*** Returns the encapsulated interface pointer, which may be **NULL**.
- **operator Interface&** Returns a reference to the encapsulated interface pointer, and issues an error if the pointer is **NULL**.
- **operator*** Allows a smart pointer object to act as though it were the actual encapsulated interface when dereferenced.
- **operator->** Allows a smart pointer object to act as though it were the actual encapsulated interface when dereferenced.
- **operator&** Releases any encapsulated interface pointer, replacing it with **NULL**, and returns the address of the encapsulated pointer. This allows the smart pointer to be passed by address to a function that has an **out** parameter through which it returns an interface pointer.
- **operator bool** Allows a smart pointer object to be used in a conditional expression. This operator returns **true** if the pointer is not **NULL**.

END Microsoft Specific

See Also
Reference
_com_ptr_t Class

Relational Function Templates

Microsoft Specific

```
template<typename _InterfaceType> bool operator==(
   int NULL,
   _com_ptr_t<_InterfaceType>& p
);
template<typename _Interface,
   typename _InterfacePtr> bool operator==(
   _Interface* i,
   _com_ptr_t<_InterfacePtr>& p
);
template<typename _Interface> bool operator!=(
   int NULL,
   _com_ptr_t<_Interface>& p
);
template<typename _Interface,
   typename _InterfacePtr> bool operator!=(
   _Interface* i,
   _com_ptr_t<_InterfacePtr>& p
);
template<typename _Interface> bool operator<(
   int NULL,
   _com_ptr_t<_Interface>& p
);
template<typename _Interface,
   typename _InterfacePtr> bool operator<(
   _Interface* i,
   _com_ptr_t<_InterfacePtr>& p
);
template<typename _Interface> bool operator>(
   int NULL,
   _com_ptr_t<_Interface>& p
);
template<typename _Interface,
   typename _InterfacePtr> bool operator>(
   _Interface* i,
   _com_ptr_t<_InterfacePtr>& p
);
template<typename _Interface> bool operator<=(
   int NULL,
   _com_ptr_t<_Interface>& p
);
template<typename _Interface,
   typename _InterfacePtr> bool operator<=(
   _Interface* i,
   _com_ptr_t<_InterfacePtr>& p
);
template<typename _Interface> bool operator>=(
   int NULL,
   _com_ptr_t<_Interface>& p
);
template<typename _Interface,
   typename _InterfacePtr> bool operator>=(
   _Interface* i,
   _com_ptr_t<_InterfacePtr>& p
);
```

Parameters

i

A raw interface pointer.

p

A smart pointer.

Remarks

These function templates allow comparison with a smart pointer on the right side of the comparison operator. These are not member functions of **_com_ptr_t**.

END Microsoft Specific

See Also

Reference

_com_ptr_t Class

_variant_t Class

Microsoft Specific

A **_variant_t** object encapsulates the VARIANT data type. The class manages resource allocation and deallocation and makes function calls to **VariantInit** and **VariantClear** as appropriate.

Construction

_variant_t	Constructs a **_variant_t** object.

Operations

Attach	Attaches a **VARIANT** object into the **_variant_t** object.
Clear	Clears the encapsulated **VARIANT** object.
ChangeType	Changes the type of the **_variant_t** object to the indicated **VARTYPE**.
Detach	Detaches the encapsulated **VARIANT** object from this **_variant_t** object.
SetString	Assigns a string to this **_variant_t** object.

Operators

Operator =	Assigns a new value to an existing **_variant_t** object.
operator ==, !=	Compare two **_variant_t** objects for equality or inequality.
Extractors	Extract data from the encapsulated **VARIANT** object.

END Microsoft Specific

Requirements

Header: comutil.h

Lib: comsuppw.lib or comsuppwd.lib (see /Zc:wchar_t (wchar_t Is Native Type) for more information)

See Also

Reference

Compiler COM Support Classes

_variant_t Member Functions

For information about the **_variant_t** member functions, see _variant_t Class.

See Also

Reference

_variant_t Class

_variant_t::_variant_t

Microsoft Specific

Constructs a **_variant_t** object.

```
_variant_t( ) throw( );

_variant_t(
   const VARIANT& varSrc
);

_variant_t(
   const VARIANT* pVarSrc
);

_variant_t(
   const _variant_t& var_t_Src
);

_variant_t(
   VARIANT& varSrc,
   bool fCopy
);

_variant_t(
   short sSrc,
   VARTYPE vtSrc = VT_I2
);

_variant_t(
   long lSrc,
   VARTYPE vtSrc = VT_I4
);

_variant_t(
   float fltSrc
) throw( );

_variant_t(
   double dblSrc,
   VARTYPE vtSrc = VT_R8
);

_variant_t(
   const CY& cySrc
) throw( );

_variant_t(
   const _bstr_t& bstrSrc
);

_variant_t(
   const wchar_t *wstrSrc
);

_variant_t(
   const char* strSrc
);

_variant_t(
   IDispatch* pDispSrc,
   bool fAddRef = true
) throw( );
```

```
_variant_t(
   bool bSrc
) throw( );

_variant_t(
   IUnknown* pIUknownSrc,
   bool fAddRef = true
) throw( );

_variant_t(
   const DECIMAL& decSrc
) throw( );

_variant_t(
   BYTE bSrc
) throw( );

variant_t(
   char cSrc
) throw();

_variant_t(
   unsigned short usSrc
) throw();

_variant_t(
   unsigned long ulSrc
) throw();

_variant_t(
   int iSrc
) throw();

_variant_t(
   unsigned int uiSrc
) throw();

_variant_t(
   __int64 i8Src
) throw();

_variant_t(
   unsigned __int64 ui8Src
) throw();
```

Parameters

varSrc

A **VARIANT** object to be copied into the new **_variant_t** object.

pVarSrc

Pointer to a **VARIANT** object to be copied into the new **_variant_t** object.

var_t_Src

A **_variant_t** object to be copied into the new **_variant_t** object.

fCopy

If false, the supplied **VARIANT** object is attached to the new **_variant_t** object without making a new copy by **VariantCopy**.

lSrc, sSrc

An integer value to be copied into the new **_variant_t** object.

vtSrc

The **VARTYPE** for the new **_variant_t** object.

fltSrc, dblSrc

A numerical value to be copied into the new **_variant_t** object.

cySrc

A **CY** object to be copied into the new **_variant_t** object.

bstrSrc

A **_bstr_t** object to be copied into the new **_variant_t** object.

strSrc, wstrSrc

A string to be copied into the new **_variant_t** object.

bSrc

A **bool** value to be copied into the new **_variant_t** object.

pIUknownSrc

COM interface pointer to a **VT_UNKNOWN** object to be encapsulated into the new **_variant_t** object.

pDispSrc

COM interface pointer to a **VT_DISPATCH** object to be encapsulated into the new **_variant_t** object.

decSrc

A **DECIMAL** value to be copied into the new **_variant_t** object.

bSrc

A **BYTE** value to be copied into the new **_variant_t** object.

cSrc

A **char** value to be copied into the new **_variant_t** object.

usSrc

A **unsigned short** value to be copied into the new **_variant_t** object.

ulSrc

A **unsigned long** value to be copied into the new **_variant_t** object.

iSrc

An **int** value to be copied into the new **_variant_t** object.

uiSrc

An **unsigned int** value to be copied into the new **_variant_t** object.

i8Src

An **__int64** value to be copied into the new **_variant_t** object.

ui8Src

An **unsigned __int64** value to be copied into the new **_variant_t** object.

Remarks

- **_variant_t( )** Constructs an empty **_variant_t** object, **VT_EMPTY**.
- **_variant_t(VARIANT&** *varSrc* **)** Constructs a **_variant_t** object from a copy of the **VARIANT** object. The variant type is retained.
- **_variant_t(VARIANT*** *pVarSrc* **)** Constructs a **_variant_t** object from a copy of the **VARIANT** object. The variant type is retained.
- **_variant_t(_variant_t&** *var_t_Src* **)** Constructs a **_variant_t** object from another **_variant_t** object. The variant type is retained.
- **_variant_t(VARIANT&** *varSrc* **, bool** *fCopy* **)** Constructs a **_variant_t** object from an existing **VARIANT** object. If *fCopy* is **false**, the **VARIANT** object is attached to the new object without making a copy.

- **_variant_t(short** *sSrc* **, VARTYPE** *vtSrc* **= VT_I2)** Constructs a **_variant_t** object of type **VT_I2** or **VT_BOOL** from a **short** integer value. Any other **VARTYPE** results in an **E_INVALIDARG** error.
- **_variant_t(long** *lSrc* **, VARTYPE** *vtSrc* **= VT_I4)** Constructs a **_variant_t** object of type **VT_I4**, **VT_BOOL**, or **VT_ERROR** from a **long** integer value. Any other **VARTYPE** results in an **E_INVALIDARG** error.
- **_variant_t(float** *fltSrc* **)** Constructs a **_variant_t** object of type **VT_R4** from a **float** numerical value.
- **_variant_t(double** *dblSrc* **, VARTYPE** *vtSrc* **= VT_R8)** Constructs a **_variant_t** object of type **VT_R8** or **VT_DATE** from a **double** numerical value. Any other **VARTYPE** results in an **E_INVALIDARG** error.
- **_variant_t(CY&** *cySrc* **)** Constructs a **_variant_t** object of type **VT_CY** from a **CY** object.
- **_variant_t(_bstr_t&** *bstrSrc* **)** Constructs a **_variant_t** object of type **VT_BSTR** from a **_bstr_t** object. A new **BSTR** is allocated.
- **_variant_t(wchar_t *** *wstrSrc* **)** Constructs a **_variant_t** object of type **VT_BSTR** from a Unicode string. A new **BSTR** is allocated.
- **_variant_t(char*** *strSrc* **)** Constructs a **_variant_t** object of type **VT_BSTR** from a string. A new **BSTR** is allocated.
- **_variant_t(bool** *bSrc* **)** Constructs a **_variant_t** object of type **VT_BOOL** from a **bool** value.
- **_variant_t(IUnknown*** *pIUknownSrc* **, bool** *fAddRef* **= true)** Constructs a **_variant_t** object of type **VT_UNKNOWN** from a COM interface pointer. If *fAddRef* is **true**, then **AddRef** is called on the supplied interface pointer to match the call to **Release** that will occur when the **_variant_t** object is destroyed. It is up to you to call **Release** on the supplied interface pointer. If *fAddRef* is **false**, this constructor takes ownership of the supplied interface pointer; do not call **Release** on the supplied interface pointer.
- **_variant_t(IDispatch*** *pDispSrc* **, bool** *fAddRef* **= true)** Constructs a **_variant_t** object of type **VT_DISPATCH** from a COM interface pointer. If *fAddRef* is **true**, then **AddRef** is called on the supplied interface pointer to match the call to **Release** that will occur when the **_variant_t** object is destroyed. It is up to you to call **Release** on the supplied interface pointer. If **fAddRef** is false, this constructor takes ownership of the supplied interface pointer; do not call **Release** on the supplied interface pointer.
- **_variant_t(DECIMAL&** *decSrc* **)** Constructs a **_variant_t** object of type **VT_DECIMAL** from a **DECIMAL** value.
- **_variant_t(BYTE** *bSrc* **)** Constructs a **_variant_t** object of type **VT_UI1** from a **BYTE** value.

END Microsoft Specific

See Also
Reference
_variant_t Class

_variant_t::Attach

Microsoft Specific

Attaches a **VARIANT** object into the **_variant_t** object.

```
void Attach(
   VARIANT& varSrc
);
```

Parameters

varSrc

A **VARIANT** object to be attached to this **_variant_t** object.

Remarks

Takes ownership of the **VARIANT** by encapsulating it. This member function releases any existing encapsulated **VARIANT**, then copies the supplied **VARIANT**, and sets its **VARTYPE** to **VT_EMPTY** to make sure its resources can only be released by the **_variant_t** destructor.

END Microsoft Specific

See Also

Reference

_variant_t Class

_variant_t::Clear

Microsoft Specific

Clears the encapsulated **VARIANT** object.

```
void Clear( );
```

Remarks

Calls **VariantClear** on the encapsulated **VARIANT** object.

END Microsoft Specific

See Also

Reference

_variant_t Class

_variant_t::ChangeType

Microsoft Specific

Changes the type of the **_variant_t** object to the indicated **VARTYPE**.

```
void ChangeType(
   VARTYPE vartype,
   const _variant_t* pSrc = NULL
);
```

Parameters

vartype

The **VARTYPE** for this **_variant_t** object.

pSrc

A pointer to the **_variant_t** object to be converted. If this value is **NULL**, conversion is done in place.

Remarks

This member function converts a **_variant_t** object into the indicated **VARTYPE**. If *pSrc* is **NULL**, the conversion is done in place, otherwise this **_variant_t** object is copied from *pSrc* and then converted.

END Microsoft Specific

See Also

Reference

_variant_t Class

_variant_t::Detach

Microsoft Specific

Detaches the encapsulated **VARIANT** object from this **_variant_t** object.

```
VARIANT Detach( );
```

Return Value

The encapsulated **VARIANT**.

Remarks

Extracts and returns the encapsulated **VARIANT**, then clears this **_variant_t** object without destroying it. This member function removes the **VARIANT** from encapsulation and sets the **VARTYPE** of this **_variant_t** object to **VT_EMPTY**. It is up to you to release the returned **VARIANT** by calling the VariantClear function.

END Microsoft Specific

See Also

Reference

_variant_t Class

_variant_t::SetString

Microsoft Specific

Assigns a string to this **_variant_t** object.

```
void SetString(
   const char* pSrc
);
```

Parameters

pSrc

Pointer to the character string.

Remarks

Converts an ANSI character string to a Unicode **BSTR** string and assigns it to this **_variant_t** object.

END Microsoft Specific

See Also

Reference

_variant_t Class

_variant_t Operators

For information about the **_variant_t** operators, see _variant_t Class.

See Also

Reference

_variant_t Class

_variant_t::operator =

Microsoft Specific

```
_variant_t& operator=(
   const VARIANT& varSrc
);

_variant_t& operator=(
   const VARIANT* pVarSrc
);

_variant_t& operator=(
   const _variant_t& var_t_Src
);

_variant_t& operator=(
   short sSrc
);

_variant_t& operator=(
   long lSrc
);

_variant_t& operator=(
   float fltSrc
);

_variant_t& operator=(
   double dblSrc
);

_variant_t& operator=(
   const CY& cySrc
);

_variant_t& operator=(
   const _bstr_t& bstrSrc
);

_variant_t& operator=(
   const wchar_t* wstrSrc
);

_variant_t& operator=(
   const char* strSrc
);

_variant_t& operator=(
   IDispatch* pDispSrc
);

_variant_t& operator=(
   bool bSrc
);

_variant_t& operator=(
   IUnknown* pSrc
);

_variant_t& operator=(
   const DECIMAL& decSrc
);
```

```
_variant_t& operator=(
   BYTE bSrc
);

_variant_t& operator=(
   char cSrc
);

_variant_t& operator=(
   unsigned short usSrc
);

_variant_t& operator=(
   unsigned long ulSrc
);

_variant_t& operator=(
   int iSrc
);

_variant_t& operator=(
   unsigned int uiSrc
);

_variant_t& operator=(
   __int64 i8Src
);

_variant_t& operator=(
   unsigned __int64 ui8Src
);
```

Remarks

The operator assigns a new value to the **_variant_t** object:

- **operator=(** *varSrc* **)** Assigns an existing **VARIANT** to a **_variant_t** object.
- **operator=(** *pVarSrc* **)** Assigns an existing **VARIANT** to a **_variant_t** object.
- **operator=(** *var_t_Src* **)** Assigns an existing **_variant_t** object to a **_variant_t** object.
- **operator=(** *sSrc* **)** Assigns a **short** integer value to a **_variant_t** object.
- **operator=(** *lSrc* **)** Assigns a **long** integer value to a **_variant_t** object.
- **operator=(** *fltSrc* **)** Assigns a **float** numerical value to a **_variant_t** object.
- **operator=(** *dblSrc* **)** Assigns a **double** numerical value to a **_variant_t** object.
- **operator=(** *cySrc* **)** Assigns a **CY** object to a **_variant_t** object.
- **operator=(** *bstrSrc* **)** Assigns a **BSTR** object to a **_variant_t** object.
- **operator=(** *wstrSrc* **)** Assigns a Unicode string to a **_variant_t** object.
- **operator=(** *strSrc* **)** Assigns a multibyte string to a **_variant_t** object.
- **operator=(** *bSrc* **)** Assigns a **bool** value to a **_variant_t** object.
- **operator=(** *pDispSrc* **)** Assigns a **VT_DISPATCH** object to a **_variant_t** object.
- **operator=(** *pIUnknownSrc* **)** Assigns a **VT_UNKNOWN** object to a **_variant_t** object.
- **operator=(** *decSrc* **)** Assigns a **DECIMAL** value to a **_variant_t** object.
- **operator=(** *bSrc* **)** Assigns a **BYTE** value to a **_variant_t** object.

END Microsoft Specific

See Also

Reference

_variant_t Relational Operators

Microsoft Specific

Compare two **_variant_t** objects for equality or inequality.

```
bool operator==(
   const VARIANT& varSrc
) const;
bool operator==(
   const VARIANT* pSrc
) const;
bool operator!=(
   const VARIANT& varSrc
) const;
bool operator!=(
   const VARIANT* pSrc
) const;
```

Parameters

varSrc

A **VARIANT** to be compared with the **_variant_t** object.

pSrc

Pointer to the **VARIANT** to be compared with the **_variant_t** object.

Return Value

Returns **true** if comparison holds, **false** if not.

Remarks

Compares a **_variant_t** object with a **VARIANT**, testing for equality or inequality.

END Microsoft Specific

See Also

Reference

_variant_t Class

_variant_t Extractors

Microsoft Specific

Extract data from the encapsulated **VARIANT** object.

```
operator short( ) const;
operator long( ) const;
operator float( ) const;
operator double( ) const;
operator CY( ) const;
operator _bstr_t( ) const;
operator IDispatch*( ) const;
operator bool( ) const;
operator IUnknown*( ) const;
operator DECIMAL( ) const;
operator BYTE( ) const;
operator VARIANT() const throw();
operator char() const;
operator unsigned short() const;
operator unsigned long() const;
operator int() const;
operator unsigned int() const;
operator __int64() const;
operator unsigned __int64() const;
```

Remarks

Extracts raw data from an encapsulated **VARIANT**. If the **VARIANT** is not already the proper type, **VariantChangeType** is used to attempt a conversion, and an error is generated upon failure:

- **operator short( )** Extracts a **short** integer value.
- **operator long( )** Extracts a **long** integer value.
- **operator float( )** Extracts a **float** numerical value.
- **operator double( )** Extracts a **double** integer value.
- **operator CY( )** Extracts a **CY** object.
- **operator bool( )** Extracts a **bool** value.
- **operator DECIMAL( )** Extracts a **DECIMAL** value.
- **operator BYTE( )** Extracts a **BYTE** value.
- **operator _bstr_t( )** Extracts a string, which is encapsulated in a **_bstr_t** object.
- **operator IDispatch*( )** Extracts a dispinterface pointer from an encapsulated **VARIANT**. **AddRef** is called on the resulting pointer, so it is up to you to call **Release** to free it.
- **operator IUnknown*( )** Extracts a COM interface pointer from an encapsulated **VARIANT**. **AddRef** is called on the resulting pointer, so it is up to you to call **Release** to free it.

END Microsoft Specific

See Also

Reference

_variant_t Class

Grammar Summary (C++)

This section includes the grammar for Microsoft extensions to C++:

- The Microsoft Extensions section covers the syntax of features unique to Microsoft C++. Many of these features are covered in Microsoft-Specific Modifiers.

See Also

Other Resources

C++ Language Reference

Microsoft Extensions

asm-statement:

__asm *assembly-instruction* **;** $_{\text{opt}}$

__asm { *assembly-instruction-list* **};** $_{\text{opt}}$

assembly-instruction-list:

assembly-instruction **;** $_{\text{opt}}$

assembly-instruction **;** *assembly-instruction-list* **;** $_{\text{opt}}$

ms-modifier-list:

ms-modifier ms-modifier-list $_{\text{opt}}$

ms-modifier:

__cdecl

__fastcall

__stdcall

__syscall (reserved for future implementations)

__oldcall (reserved for future implementations)

__unaligned (reserved for future implementations)

based-modifier

based-modifier:

__based (*based-type* **)**

based-type:

name

See Also

Reference

Grammar Summary (C++)

C/C++ Preprocessor Reference

The *C/C++ Preprocessor Reference* explains the preprocessor as it is implemented in Microsoft C/C++. The preprocessor performs preliminary operations on C and C++ files before they are passed to the compiler. You can use the preprocessor to conditionally compile code, insert files, specify compile-time error messages, and apply machine-specific rules to sections of code.

In This Section

Preprocessor Directives

Describes directives, typically used to make source programs easy to change and easy to compile in different execution environments.

Preprocessor Operators

Discusses the four preprocessor-specific operators used in the context of the **#define** directive.

Predefined Macros

Discusses predefined macros as specified by ANSI and Microsoft C++.

Pragmas

Discusses pragmas, which offer a way for each compiler to offer machine- and operating system-specific features while retaining overall compatibility with the C and C++ languages.

Related Sections

C++ Language Reference

Provides reference material for the Microsoft implementation of the C++ language.

C Language Reference

Provides reference material for the Microsoft implementation of the C language.

Building a C/C++ Program

Provides links to topics discussing compiler and linker options.

Visual C++ Libraries

Provides links to the various libraries provided with Visual C++, including ATL, ATL Server, MFC, OLE DB, the C run-time library, and the Standard C++ Library.

VC++ Directories, Projects and Solutions, Options Dialog Box

Describes the user interface in Visual Studio that enables you to specify the directories that the project system will search to locate files for your C++ project.

The Preprocessor

The preprocessor is a text processor that manipulates the text of a source file as part of the first phase of translation. The preprocessor does not parse the source text, but it does break it up into tokens for the purpose of locating macro calls. Although the compiler ordinarily invokes the preprocessor in its first pass, the preprocessor can also be invoked separately to process text without compiling.

The reference material on the preprocessor includes the following sections:

- Preprocessor directives
- Preprocessor operators
- Predefined macros
- Pragmas

Microsoft Specific

You can obtain a listing of your source code after preprocessing by using the /E or /EP compiler option. Both options invoke the preprocessor and output the resulting text to the standard output device, which, in most cases, is the console. The difference between the two options is that /E includes **#line** directives and /EP strips these directives out.

END Microsoft Specific

Special Terminology

In the preprocessor documentation, the term "argument" refers to the entity that is passed to a function. In some cases, it is modified by "actual" or "formal," which describes the argument expression specified in the function call and the argument declaration specified in the function definition, respectively.

The term "variable" refers to a simple C-type data object. The term "object" refers to both C++ objects and variables; it is an inclusive term.

See Also

Reference

Phases of Translation

Other Resources

C/C++ Preprocessor Reference

Phases of Translation

C and C++ programs consist of one or more source files, each of which contains some of the text of the program. A source file, together with its include files (files that are included using the **#include** preprocessor directive) but not including sections of code removed by conditional-compilation directives such as **#if**, is called a "translation unit."

Source files can be translated at different times — in fact, it is common to translate only out-of-date files. The translated translation units can be processed into separate object files or object-code libraries. These separate, translated translation units are then linked to form an executable program or a dynamic-link library (DLL). For more information about files that can be used as input to the linker, see LINK Input Files.

Translation units can communicate using:

- Calls to functions that have external linkage.
- Calls to class member functions that have external linkage.
- Direct modification of objects that have external linkage.
- Direct modification of files.
- Interprocess communication (for Microsoft Windows-based applications only).

The following list describes the phases in which the compiler translates files:

Character mapping

Characters in the source file are mapped to the internal source representation. Trigraph sequences are converted to single-character internal representation in this phase.

Line splicing

All lines ending in a backslash (****) and immediately followed by a newline character are joined with the next line in the source file forming logical lines from the physical lines. Unless it is empty, a source file must end in a newline character that is not preceded by a backslash.

Tokenization

The source file is broken into preprocessing tokens and white-space characters. Comments in the source file are replaced with one space character each. Newline characters are retained.

Preprocessing

Preprocessing directives are executed and macros are expanded into the source file. The **#include** statement invokes translation starting with the preceding three translation steps on any included text.

Character-set mapping

All source character set members and escape sequences are converted to their equivalents in the execution character set. For Microsoft C and C++, both the source and the execution character sets are ASCII.

String concatenation

All adjacent string and wide-string literals are concatenated. For example, `"String " "concatenation"` becomes `"String concatenation"`.

Translation

All tokens are analyzed syntactically and semantically; these tokens are converted into object code.

Linkage

All external references are resolved to create an executable program or a dynamic-link library.

The compiler issues warnings or errors during phases of translation in which it encounters syntax errors.

The linker resolves all external references and creates an executable program or DLL by combining one or more separately processed translation units along with standard libraries.

See Also

Reference

Preprocessor Directives

Preprocessor directives, such as **#define** and **#ifdef**, are typically used to make source programs easy to change and easy to compile in different execution environments. Directives in the source file tell the preprocessor to perform specific actions. For example, the preprocessor can replace tokens in the text, insert the contents of other files into the source file, or suppress compilation of part of the file by removing sections of text. Preprocessor lines are recognized and carried out before macro expansion. Therefore, if a macro expands into something that looks like a preprocessor command, that command is not recognized by the preprocessor.

Preprocessor statements use the same character set as source file statements, with the exception that escape sequences are not supported. The character set used in preprocessor statements is the same as the execution character set. The preprocessor also recognizes negative character values.

The preprocessor recognizes the following directives:

#define	#error	#import	#undef
#elif	#if	#include	#using
#else	#ifdef	#line	
#endif	#ifndef	#pragma	

The number sign (**#**) must be the first nonwhite-space character on the line containing the directive; white-space characters can appear between the number sign and the first letter of the directive. Some directives include arguments or values. Any text that follows a directive (except an argument or value that is part of the directive) must be preceded by the single-line comment delimiter (**//**) or enclosed in comment delimiters (**/* */**). Lines containing preprocessor directives can be continued by immediately preceding the end-of-line marker with a backslash (****).

Preprocessor directives can appear anywhere in a source file, but they apply only to the remainder of the source file.

See Also

Reference

Preprocessor Operators

Predefined Macros

Other Resources

C/C++ Preprocessor Reference

The #define Directive

You can use the **#define** directive to give a meaningful name to a constant in your program. The two forms of the syntax are:

```
#define identifier token-string<SUB>opt</SUB>
#define identifier[( identifier<SUB>opt</SUB>,  ... , identifier<SUB>opt</SUB> )] token-str
ing<SUB>opt</SUB>
```

Remarks

The **#define** directive substitutes *token-string* for all subsequent occurrences of an *identifier* in the source file. The *identifier* is replaced only when it forms a token. (See C++ Tokens in the *C++ Language Reference*.) For instance, *identifier* is not replaced if it appears in a comment, within a string, or as part of a longer identifier.

A **#define** without a *token-string* removes occurrences of *identifier* from the source file. The *identifier* remains defined and can be tested using the **#if defined** and **#ifdef** directives.

The *token-string* argument consists of a series of tokens, such as keywords, constants, or complete statements. One or more white-space characters must separate *token-string* from *identifier*. This white space is not considered part of the substituted text, nor is any white space following the last token of the text.

Formal parameter names appear in *token-string* to mark the places where actual values are substituted. Each parameter name can appear more than once in *token-string*, and the names can appear in any order. The number of arguments in the call must match the number of parameters in the macro definition. Liberal use of parentheses ensures that complicated actual arguments are interpreted correctly.

The second syntax form allows the creation of function-like macros. This form accepts an optional list of parameters that must appear in parentheses. References to the *identifier* after the original definition replace each occurrence of *identifier*(*identifier*opt, ..., *identifier*opt) with a version of the *token-string* argument that has actual arguments substituted for formal parameters.

The formal parameters in the list are separated by commas. Each name in the list must be unique, and the list must be enclosed in parentheses. No spaces can separate *identifier* and the opening parenthesis. Use line concatenation — place a backslash (****) immediately before the newline character — for long directives on multiple source lines. The scope of a formal parameter name extends to the new line that ends *token-string*.

When a macro has been defined in the second syntax form, subsequent textual instances followed by an argument list constitute a macro call. The actual arguments following an instance of *identifier* in the source file are matched to the corresponding formal parameters in the macro definition. Each formal parameter in *token-string* that is not preceded by a stringizing (**#**), charizing (**#@**), or token-pasting (**##**) operator, or not followed by a **##** operator, is replaced by the corresponding actual argument. Any macros in the actual argument are expanded before the directive replaces the formal parameter. (The operators are described in Preprocessor Operators.)

The following examples of macros with arguments illustrate the second form of the **#define** syntax:

```
// Macro to define cursor lines
#define CURSOR(top, bottom) (((top) << 8) | (bottom))

// Macro to get a random integer with a specified range
#define getrandom(min, max) \
    ((rand()%(int)(((max) + 1)-(min)))+ (min))
```

Arguments with side effects sometimes cause macros to produce unexpected results. A given formal parameter may appear more than once in *token-string*. If that formal parameter is replaced by an expression with side effects, the expression, with its side effects, may be evaluated more than once. (See the examples under Token-Pasting Operator (##).)

The **#undef** directive causes an identifier's preprocessor definition to be forgotten. See The #undef Directive for more information.

If the name of the macro being defined occurs in *token-string* (even as a result of another macro expansion), it is not expanded.

A second **#define** for a macro with the same name generates an error unless the second token sequence is identical to the first.

Microsoft Specific

Microsoft C/C++ allows the redefinition of a macro, but generates a warning, provided the new definition is lexically identical to a previous definition. ANSI C considers macro redefinition an error. For example, these macros are equivalent for C/C++ but generate warnings:

```
#define test( f1, f2 ) ( f1 * f2 )
#define test( a1, a2 ) ( a1 * a2 )
```

END Microsoft Specific

This example illustrates the **#define** directive:

```
#define WIDTH       80
#define LENGTH      ( WIDTH + 10 )
```

The first statement defines the identifier `WIDTH` as the integer constant 80 and defines `LENGTH` in terms of `WIDTH` and the integer constant 10. Each occurrence of `LENGTH` is replaced by (`WIDTH + 10`). In turn, each occurrence of `WIDTH + 10` is replaced by the expression (`80 + 10`). The parentheses around `WIDTH + 10` are important because they control the interpretation in statements such as the following:

```
var = LENGTH * 20;
```

After the preprocessing stage the statement becomes:

```
var = ( 80 + 10 ) * 20;
```

which evaluates to 1800. Without parentheses, the result is:

```
var = 80 + 10 * 20;
```

which evaluates to 280.

Microsoft Specific

Defining macros and constants with the /D compiler option has the same effect as using a **#define** preprocessing directive at the beginning of your file. Up to 30 macros can be defined with the /D option.

END Microsoft Specific

See Also
Reference
Preprocessor Directives

The #error Directive

Error directives produce compiler-time error messages.

```
#error token-string
```

Remarks

The error messages include the argument *token-string* and are currently not subject to macro expansion. These directives are most useful for detecting programmer inconsistencies and violation of constraints during preprocessing. The following example demonstrates error processing during preprocessing:

```
#if !defined(__cplusplus)
#error C++ compiler required.
#endif
```

When **#error** directives are encountered, compilation terminates.

See Also

Reference

Preprocessor Directives

The #if, #elif, #else, and #endif Directives

The **#if** directive, with the **#elif**, **#else**, and **#endif** directives, controls compilation of portions of a source file. If the expression you write (after the **#if**) has a nonzero value, the line group immediately following the **#if** directive is retained in the translation unit.

Grammar

conditional :

if-part elif-parts $_{\text{opt}}$ *else-part*$_{\text{opt}}$ *endif-line*

if-part :

if-line text

if-line :

#if *constant-expression*

#ifdef *identifier*

#ifndef *identifier*

elif-parts :

elif-line text

elif-parts elif-line text

elif-line :

#elif *constant-expression*

else-part :

else-line text

else-line :

#else

endif-line :

#endif

Each **#if** directive in a source file must be matched by a closing **#endif** directive. Any number of **#elif** directives can appear between the **#if** and **#endif** directives, but at most one **#else** directive is allowed. The **#else** directive, if present, must be the last directive before **#endif**.

The **#if**, **#elif**, **#else**, and **#endif** directives can nest in the text portions of other **#if** directives. Each nested **#else**, **#elif**, or **#endif** directive belongs to the closest preceding **#if** directive.

All conditional-compilation directives, such as **#if** and **#ifdef**, must be matched with closing **#endif** directives prior to the end of file; otherwise, an error message is generated. When conditional-compilation directives are contained in include files, they must satisfy the same conditions: There must be no unmatched conditional-compilation directives at the end of the include file.

Macro replacement is performed within the part of the command line that follows an **#elif** command, so a macro call can be used in the *constant-expression*.

The preprocessor selects one of the given occurrences of *text* for further processing. A block specified in *text* can be any sequence of text. It can occupy more than one line. Usually *text* is program text that has meaning to the compiler or the preprocessor.

The preprocessor processes the selected *text* and passes it to the compiler. If *text* contains preprocessor directives, the preprocessor carries out those directives. Only text blocks selected by the preprocessor are compiled.

The preprocessor selects a single *text* item by evaluating the constant expression following each **#if** or **#elif** directive until it finds a true (nonzero) constant expression. It selects all text (including other preprocessor directives beginning with **#**) up to its associated **#elif**, **#else**, or **#endif**.

If all occurrences of *constant-expression* are false, or if no **#elif** directives appear, the preprocessor selects the text block after

the **#else** clause. If the **#else** clause is omitted and all instances of *constant-expression* in the **#if** block are false, no text block is selected.

The *constant-expression* is an integer constant expression with these additional restrictions:

- Expressions must have integral type and can include only integer constants, character constants, and the **defined** operator.
- The expression cannot use **sizeof** or a type-cast operator.
- The target environment may not be able to represent all ranges of integers.
- The translation represents type **int** the same as type **long**, and **unsigned int** the same as **unsigned long**.
- The translator can translate character constants to a set of code values different from the set for the target environment. To determine the properties of the target environment, check values of macros from LIMITS.H in an application built for the target environment.
- The expression must not perform any environmental inquiries and must remain insulated from implementation details on the target computer.

The preprocessor operator **defined** can be used in special constant expressions, as shown by the following syntax:

defined(*identifier*)

defined *identifier*

This constant expression is considered true (nonzero) if the *identifier* is currently defined; otherwise, the condition is false (0). An identifier defined as empty text is considered defined. The **defined** directive can be used in an **#if** and an **#elif** directive, but nowhere else.

In the following example, the **#if** and **#endif** directives control compilation of one of three function calls:

```
#if defined(CREDIT)
    credit();
#elif defined(DEBIT)
    debit();
#else
    printerror();
#endif
```

The function call to `credit` is compiled if the identifier `CREDIT` is defined. If the identifier `DEBIT` is defined, the function call to `debit` is compiled. If neither identifier is defined, the call to `printerror` is compiled. Note that `CREDIT` and `credit` are distinct identifiers in C and C++ because their cases are different.

The conditional compilation statements in the following example assume a previously defined symbolic constant named `DLEVEL`.

```
#if DLEVEL > 5
    #define SIGNAL  1
    #if STACKUSE == 1
        #define STACK   200
    #else
        #define STACK   100
    #endif
#else
    #define SIGNAL  0
    #if STACKUSE == 1
        #define STACK   100
    #else
        #define STACK   50
    #endif
#endif
#if DLEVEL == 0
    #define STACK 0
#elif DLEVEL == 1
    #define STACK 100
#elif DLEVEL > 5
```

```
    display( debugptr );
#else
    #define STACK 200
#endif
```

The first **#if** block shows two sets of nested **#if**, **#else**, and **#endif** directives. The first set of directives is processed only if `DLEVEL > 5` is true. Otherwise, the statements after #**else** are processed.

The **#elif** and **#else** directives in the second example are used to make one of four choices, based on the value of `DLEVEL`. The constant `STACK` is set to 0, 100, or 200, depending on the definition of `DLEVEL`. If `DLEVEL` is greater than 5, then the statement

```
#elif DLEVEL > 5
display(debugptr);
```

is compiled and `STACK` is not defined.

A common use for conditional compilation is to prevent multiple inclusions of the same header file. In C++, where classes are often defined in header files, constructs like the following can be used to prevent multiple definitions:

```
/*  EXAMPLE.H - Example header file  */
#if !defined( EXAMPLE_H )
#define EXAMPLE_H

class Example
{
...
};

#endif // !defined( EXAMPLE_H )
```

The preceding code checks to see if the symbolic constant `EXAMPLE_H` is defined. If so, the file has already been included and need not be reprocessed. If not, the constant `EXAMPLE_H` is defined to mark EXAMPLE.H as already processed.

Microsoft Specific

Conditional compilation expressions are treated as **signed long** values, and these expressions are evaluated using the same rules as expressions in C++. For example, this expression:

```
#if 0xFFFFFFFFL > 1UL
```

is true.

END Microsoft Specific

See Also
Reference
Preprocessor Directives

The #ifdef and #ifndef Directives

The **#ifdef** and **#ifndef** directives perform the same task as the **#if** directive when it is used with **defined**(*identifier*).

```
#ifdef identifier
#ifndef identifier

// equivalent to
#if defined identifier
#if !defined identifier
```

Remarks

You can use the **#ifdef** and **#ifndef** directives anywhere **#if** can be used. The **#ifdef** *identifier* statement is equivalent to `#if 1` when *identifier* has been defined, and it is equivalent to `#if 0` when *identifier* has not been defined or has been undefined with the **#undef** directive. These directives check only for the presence or absence of identifiers defined with **#define**, not for identifiers declared in the C or C++ source code.

These directives are provided only for compatibility with previous versions of the language. The **defined(** *identifier* **)** constant expression used with the **#if** directive is preferred.

The **#ifndef** directive checks for the opposite of the condition checked by **#ifdef**. If the identifier has not been defined (or its definition has been removed with **#undef**), the condition is true (nonzero). Otherwise, the condition is false (0).

Microsoft Specific

The *identifier* can be passed from the command line using the /D option. Up to 30 macros can be specified with /D.

This is useful for checking whether a definition exists, because a definition can be passed from the command line. For example:

```
// ifdef_ifndef.CPP
// compile with: /Dtest /c
#ifndef test
#define final
#endif
```

END Microsoft Specific

See Also

Reference

Preprocessor Directives

The #import Directive

C++ Specific

Used to incorporate information from a type library. The content of the type library is converted into C++ classes, mostly describing the COM interfaces.

```
#import "filename" [attributes]
#import <filename> [attributes]
```

Parameters

filename

Allows you to specify which type library you want to import. *filename* can be one of the following:

- A type library (.tlb or .odl). **file:**, which indicates type library, can precede each filename.
- The progid of a control in the type library. Note that **progid:** can precede each progid. For example:

  ```
  #import "progid:my.prog.id.1.5"
  ```

  For more on progids, see Specifying the Localization ID and Version Number.

  Note that when compiling with a cross compiler on a 64-bit operating system, the compiler will only be able to read the 32-bit registry hive, so you may want to use the native 64-bit compiler to build and register a 64-bit type library.
- The library ID of the type library. Note that **libid:** can precede each library ID. For example:

  ```
  #import "libid:12341234-1234-1234-1234-123412341234" version("4.0") lcid("9")
  ```

  If you do not specify version or lcid, the rules that are applied to **progid:** are also applied to **libid:**.
- An executable (.exe) file.
- A library (.dll) file containing a type library resource (such as .ocx).
- A compound document holding a type library.
- Any other file format that can be understood by the **LoadTypeLib** API.

attributes

One or more #import attributes. Separate attributes with either a space or comma. For example:

```
#import "..\drawctl\drawctl.tlb" no_namespace, raw_interfaces_only
```

-or-

```
#import "..\drawctl\drawctl.tlb" no_namespace raw_interfaces_only
```

Remarks

Search Order for filename

filename is optionally preceded by a directory specification. The file name must name an existing file. The difference between the two syntax forms is the order in which the preprocessor searches for the type library files when the path is incompletely specified.

Syntax form	Action

Quoted form	Instructs the preprocessor to look for type library files first in the directory of the file that contains the **#import** statement, and then in the directories of whatever files that include (**#include**) that file. The preprocessor then searches along the paths shown below.
Angle-bracket form	Instructs the preprocessor to search for type library files along the following paths: 1. The **PATH** environment variable path list 2. The **LIB** environment variable path list 3. The path specified by the /I (additional include directories) compiler option, except it the compiler is searching for a type library that was referenced from another type library with the no_registry attribute.

Specifying the Localization ID and Version Number

When you specify a progid, you can also specify the localization ID and version number of the progid. For example:

```
#import "progid:my.prog.id" lcid("0") version("4.0)
```

If you do not specify a localization ID, a progid is chosen according to the following rules:

- If there is only one localization ID, that one is used.
- If there is more than one localization ID, the first one with version number 0, 9, or 409 is used.
- If there is more than one localization ID and none of them are 0, 9, or 409, the last one is used.
- If you do not specify a version number, the most recent version is used.

Header Files Created by Import

#import creates two header files that reconstruct the type library contents in C++ source code. The primary header file is similar to that produced by the Microsoft Interface Definition Language (MIDL) compiler, but with additional compiler-generated code and data. The primary header file has the same base name as the type library, plus a .TLH extension. The secondary header file has the same base name as the type library, with a .TLI extension. It contains the implementations for compiler-generated member functions, and is included (**#include**) in the primary header file.

If importing a dispinterface property that uses byref parameters, #import will not generate __declspec(property) statement for the function.

Both header files are placed in the output directory specified by the /Fo (name object file) option. They are then read and compiled by the compiler as if the primary header file was named by a **#include** directive.

The following compiler optimizations come with the **#import** directive:

- The header file, when created, is given the same timestamp as the type library.
- When **#import** is processed, the compiler first checks if the header exists and is up to date. If yes, then it does not need to be re-created.

The **#import** directive also participates in minimal rebuild and can be placed in a precompiled header file. See Creating Precompiled Header Files for more information.

Primary Type Library Header File

The primary type library header file consists of seven sections:

- Heading boilerplate: Consists of comments, **#include** statement for COMDEF.H (which defines some standard macros used in the header), and other miscellaneous setup information.
- Forward references and typedefs: Consists of structure declarations such as `struct IMyInterface` and typedefs.
- Smart pointer declarations: The template class **_com_ptr_t** is a smart-pointer implementation that encapsulates interface pointers and eliminates the need to call **AddRef**, **Release**, **QueryInterface** functions. In addition, it hides the **CoCreateInstance** call in creating a new COM object. This section uses macro statement **_COM_SMARTPTR_TYPEDEF** to establish typedefs of COM interfaces to be template specializations of the _com_ptr_t template class. For example, for interface **IMyInterface**, the .TLH file will contain:

```
_COM_SMARTPTR_TYPEDEF(IMyInterface, __uuidof(IMyInterface));
```

which the compiler will expand to:

```
typedef _com_ptr_t<_com_IIID<IMyInterface, __uuidof(IMyInterface)> > IMyInterfacePtr;
```

Type `IMyInterfacePtr` can then be used in place of the raw interface pointer `IMyInterface*`. Consequently, there is no need to call the various **IUnknown** member functions

- Typeinfo declarations: Primarily consists of class definitions and other items exposing the individual typeinfo items returned by **ITypeLib:GetTypeInfo**. In this section, each typeinfo from the type library is reflected in the header in a form dependent on the TYPEKIND information.
- Optional old-style GUID definition: Contains initializations of the named GUID constants. These are names of the form **CLSID_CoClass** and **IID_Interface**, similar to those generated by the MIDL compiler.
- **#include** statement for the secondary type library header.
- Footer boilerplate: Currently includes `#pragma pack(pop)`.

All sections, except the heading boilerplate and footer boilerplate section, are enclosed in a namespace with its name specified by the **library** statement in the original IDL file. You can use the names from the type library header either by an explicit qualification with the namespace name or by including the following statement:

```
using namespace MyLib;
```

immediately after the **#import** statement in the source code.

The namespace can be suppressed by using the no_namespace attribute of the **#import** directive. However, suppressing the namespace may lead to name collisions. The namespace can also be renamed by the rename_namespace attribute.

The compiler provides the full path to any type library dependency required by the type library it is currently processing. The path is written, in the form of comments, into the type library header (.TLH) that the compiler generates for each processed type library.

If a type library includes references to types defined in other type libraries, then the .TLH file will include comments of the following sort:

```
//
// Cross-referenced type libraries:
//
//  #import "c:\path\typelib0.tlb"
//
```

The actual filename in the **#import** comment is the full path of the cross-referenced type library, as stored in the registry. If you encounter errors that are due to missing type definitions, check the comments at the head of the .TLH to see which dependent type libraries may need to be imported first. Likely errors are syntax errors (for example, C2143, C2146, C2321), C2501 (missing decl-specifiers), or C2433 ('inline' not permitted on data declaration) while compiling the .TLI file.

You must determine which of the dependency comments are not otherwise provided for by system headers and then provide an **#import** directive at some point before the **#import** directive of the dependent type library to resolve the errors.

For more information, see the Knowledge Base article "#import Wrapper Methods May Cause Access Violation" (Q242527) or "Compiler Errors When You Use #import with XML" (Q269194). You can find Knowledge Base articles on the MSDN Library media or at http://support.microsoft.com/support/.

#import Attributes

#import can optionally include one or more attributes. These attributes tell the compiler to modify the contents of the type-library headers. A backslash (****) symbol can be used to include additional lines in a single **#import** statement. For example:

```
#import "test.lib" no_namespace \
   rename("OldName", "NewName")
```

For more information, see #import Attributes.

END C++ Specific

See Also

Reference

Preprocessor Directives

#import Attributes

Provides links to attributes used with the #import directive.

Microsoft Specific

The following attributes are available to the #import directive.

Attribute	Description
auto_rename	Renames C++ reserved words by appending two underscores (__) to the variable name to resolve potential name conflicts.
auto_search	Specifies that, when a type library is referenced with #import and itself references another type library, the compiler can do an implicit #import for the other type library.
embedded_idl	Specifies that the type library is written to the .tlh file with the attribute-generated code preserved.
exclude (#import)	Excludes items from the type library header files being generated.
high_method_prefix	Specifies a prefix to be used in naming high-level properties and methods.
high_property_prefixes	Specifies alternate prefixes for three property methods.
implementation_only	Suppresses the generation of the .tlh header file (the primary header file).
include()	Disables automatic exclusion.
inject_statement	Inserts its argument as source text into the type-library header.
named_guids	Tells the compiler to define and initialize GUID variables in old style, of the form **LIBID_MyLib**, **CLSID_MyCoClass**, **IID_MyInterface**, and **DIID_MyDispInterface**.
no_auto_exclude	Disables automatic exclusion.
no_dual_interfaces	Changes the way the compiler generates wrapper functions for dual interface methods.
no_implementation	Suppresses the generation of the .tli header, which contains the implementations of the wrapper member functions.
no_namespace	Specifies that the namespace name is not generated by the compiler.
no_registry	Tells the compiler not to search the registry for type libraries.
no_search_namespace	Has the same functionality as the no_namespace attribute but is used on type libraries that you use the #import directive with the auto_search attribute.
no_smart_pointers	Suppresses the creation of smart pointers for all interfaces in the type library.
raw_dispinterfaces	Tells the compiler to generate low-level wrapper functions for dispinterface methods and properties that call **IDispatch::Invoke** and return the **HRESULT** error code.
raw_interfaces_only	Suppresses the generation of error-handling wrapper functions and property (C++) declarations that use those wrapper functions.

raw_method_prefix	Specifies a different prefix to avoid name collisions.
raw_native_types	Disables the use of COM support classes in the high-level wrapper functions and forces the use of low-level data types instead.
raw_property_prefixes	Specifies alternate prefixes for three property methods.
rename (#import)	Works around name collision problems.
rename_namespace	Renames the namespace that contains the contents of the type library.
rename_search_namespace	Has the same functionality as the rename_namespace attribute but is used on type libraries that y ou use the #import directive with the auto_search attribute.
tlbid	Allows for loading libraries other than the primary type library.

END Microsoft Specific

See Also
Reference
The #import Directive

auto_rename

C++ Specific

Renames C++ reserved words by appending two underscores (__) to the variable name to resolve potential name conflicts.

```
auto_rename
```

Remarks

This attribute is used when importing a type library that uses one or more C++ reserved words (keywords or macros) as variable names.

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

auto_search

C++ Specific

Specifies that, when a type library is referenced with #import and itself references another type library, the compiler can do an implicit #import for the other type library.

```
auto_search
```

Remarks

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

embedded_idl

C++ Specific

Specifies that the type library is written to the .tlh file with the attribute-generated code preserved.

```
embedded_idl[("param")]
```

Parameters

param

Can be one of two values:

- emitidl: Type information imported from the typelib will be present in the IDL generated for the attributed project. This is the default and will be in effect if you do not specify a parameter to **embedded_idl**.
- no_emitidl: Type information imported from the typelib will not be present in the IDL generated for the attributed project.

Example

```
// import_embedded_idl.cpp
// compile with: /LD
[module(name="MyLib2")];
#import "\school\bin\importlib.tlb" embedded_idl("no_emitidl")
```

Remarks

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

exclude (#import)

C++ Specific

Excludes items from the type library header files being generated.

```
exclude("Name1"[, "Name2",...])
```

Parameters

Name1

First item to be excluded.

Name2

Second item to be excluded (if necessary).

Remarks

Type libraries may include definitions of items defined in system headers or other type libraries. This attribute can take any number of arguments, each being a top-level type library item to be excluded.

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

high_method_prefix

C++ Specific

Specifies a prefix to be used in naming high-level properties and methods.

```
high_method_prefix("Prefix")
```

Parameters

Prefix

Prefix to be used.

Remarks

By default, high-level error-handling properties and methods are exposed by member functions named without a prefix. The names are from the type library.

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

high_property_prefixes

C++ Specific

Specifies alternate prefixes for three property methods.

```
high_property_prefixes("GetPrefix","PutPrefix","PutRefPrefix")
```

Parameters

GetPrefix

Prefix to be used for the **propget** methods.

PutPrefix

Prefix to be used for the **propput** methods.

PutRefPrefix

Prefix to be used for the **propputref** methods.

Remarks

By default, high-level error-handling **propget**, **propput**, and **propputref** methods are exposed by member functions named with prefixes **Get**, **Put**, and **PutRef**, respectively.

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

implementation_only

C++ Specific

Suppresses the generation of the .tlh header file (the primary header file).

```
implementation_only
```

Remarks

This file contains all the declarations used to expose the type-library contents. The .tli header file, with the implementations of the wrapper member functions, will be generated and included in the compilation.

When this attribute is specified, the content of the .tli header is in the same namespace as the one normally used in the .tlh header. In addition, the member functions are not declared as inline.

The **implementation_only** attribute is intended for use in conjunction with the no_implementation attribute as a way of keeping the implementations out of the precompiled header (PCH) file. An **#import** statement with the **no_implementation** attribute is placed in the source region used to create the PCH. The resulting PCH is used by a number of source files. An **#import** statement with the **implementation_only** attribute is then used outside the PCH region. You are required to use this statement only once in one of the source files. This will generate all the required wrapper member functions without additional recompilation for each source file.

Note

The **implementation_only** attribute in one **#import** statement must be use in conjunction with another **#import** statement, of the same type library, with the **no_implementation** attribute. Otherwise, compiler errors will be generated. This is because wrapper class definitions generated by the **#import** statement with the **no_implementation** attribute are required to compile the implementations generated by the **implementation_only** attribute.

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

include()

C++ Specific

Disables automatic exclusion.

```
include("Name1"[,"Name2", ...])
```

Parameters

Name1

First item to be forcibly included.

Name2

Second item to be forcibly included (if necessary).

Remarks

Type libraries may include definitions of items defined in system headers or other type libraries. **#import** attempts to avoid multiple definition errors by automatically excluding such items. If items have been excluded, as indicated by Compiler Warning (level 3) C4192, and they should not have been, this attribute can be used to disable the automatic exclusion. This attribute can take any number of arguments, each being the name of the type-library item to be included.

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

inject_statement

C++ Specific

Inserts its argument as source text into the type-library header.

```
inject_statement("source_text")
```

Parameters

source_text

Source text to be inserted into the type library header file.

Remarks

The text is placed at the beginning of the namespace declaration that wraps the type-library contents in the header file.

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

named_guids

C++ Specific

Tells the compiler to define and initialize GUID variables in old style, of the form **LIBID_MyLib**, **CLSID_MyCoClass**, **IID_MyInterface**, and **DIID_MyDispInterface**.

```
named_guids
```

Remarks

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

no_auto_exclude

C++ Specific

Disables automatic exclusion.

```
no_auto_exclude
```

Remarks

Type libraries may include definitions of items defined in system headers or other type libraries. **#import** attempts to avoid multiple definition errors by automatically excluding such items. When this is done, Compiler Warning (level 3) C4192 will be issued for each item to be excluded. You can disable this automatic exclusion by using this attribute.

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

no_dual_interfaces

C++ Specific

Changes the way the compiler generates wrapper functions for dual interface methods.

```
no_dual_interfaces
```

Remarks

Normally, the wrapper will call the method through the virtual function table for the interface. With **no_dual_interfaces**, the wrapper instead calls **IDispatch::Invoke** to invoke the method.

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

no_implementation

C++ Specific

Suppresses the generation of the .tli header, which contains the implementations of the wrapper member functions.

```
no_implementation
```

Remarks

If this attribute is specified, the .tlh header, with the declarations to expose type-library items, will be generated without an **#include** statement to include the .tli header file.

This attribute is used in conjunction with implementation_only.

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

no_namespace

C++ Specific

Specifies that the namespace name is not generated by the compiler.

```
no_namespace
```

Remarks

The type-library contents in the **#import** header file are normally defined in a namespace. The namespace name is specified in the **library** statement of the original IDL file. If the **no_namespace** attribute is specified, then this namespace is not generated by the compiler.

If you want to use a different namespace name, then use the rename_namespace attribute instead.

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

no_registry

no_registry tells the compiler not to search the registry for type libraries imported with **#import**.

```
#import filename no_registry
```

Parameters
filename

A type library.

Remarks

If a referenced type library is not found in the include directories, the compilation will fail even if the type library is in the registry. **no_registry** propagates to other type libraries implicitly imported with **auto_search**.

The compiler will never search the registry for type libraries that are specified by file name and passed directly to **#import**.

When **auto_search** is specified, the additional **#import**s will be generated with the **no_registry** setting of the initial **#import** (if the initial **#import** directive was **no_registry**, an **auto_search**-generated **#import** is also **no_registry**.)

no_registry is useful if you want to import cross referenced type libraries without the risk of the compiler finding an older version of the file in the registry. **no_registry** is also useful if the type library is not registered.

See Also
Reference
#import Attributes
The #import Directive

no_search_namespace

C++ Specific

Has the same functionality as the no_namespace attribute but is used on type libraries that you use the #import directive with the auto_search attribute.

```
no_search_namespace
```

Remarks

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

no_smart_pointers

C++ Specific

Suppresses the creation of smart pointers for all interfaces in the type library.

```
no_smart_pointers
```

Remarks

By default, when you use **#import**, you get a smart pointer declaration for all interfaces in the type library. These smart pointers are of type _com_ptr_t Class.

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

raw_dispinterfaces

C++ Specific

Tells the compiler to generate low-level wrapper functions for dispinterface methods and properties that call **IDispatch::Invoke** and return the **HRESULT** error code.

```
raw_dispinterfaces
```

Remarks

If this attribute is not specified, only high-level wrappers are generated, which throw C++ exceptions in case of failure.

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

raw_interfaces_only

C++ Specific

Suppresses the generation of error-handling wrapper functions and property (C++) declarations that use those wrapper functions.

```
raw_interfaces_only
```

Remarks

The **raw_interfaces_only** attribute also causes the default prefix used in naming the non-property functions to be removed. Normally, the prefix is **raw_**. If this attribute is specified, the function names are directly from the type library.

This attribute allows you to expose only the low-level contents of the type library.

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

raw_method_prefix

C++ Specific

Specifies a different prefix to avoid name collisions.

```
raw_method_prefix("Prefix")
```

Parameters

Prefix

The prefix to be used.

Remarks

Low-level properties and methods are exposed by member functions named with a default prefix of **raw_** to avoid name collisions with the high-level error-handling member functions.

Note
The effects of the **raw_method_prefix** attribute will not be changed by the presence of the raw_interfaces_only attribute. The **raw_method_prefix** always takes precedence over **raw_interfaces_only** in specifying a prefix. If both attributes are used in the same **#import** statement, then the prefix specified by the **raw_method_prefix** attribute is used.

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

raw_native_types

C++ Specific

Disables the use of COM support classes in the high-level wrapper functions and forces the use of low-level data types instead.

```
raw_native_types
```

Remarks

By default, the high-level error-handling methods use the COM support classes _bstr_t and _variant_t in place of the **BSTR** and **VARIANT** data types and raw COM interface pointers. These classes encapsulate the details of allocating and deallocating memory storage for these data types, and greatly simplify type casting and conversion operations.

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

raw_property_prefixes

C++ Specific

Specifies alternate prefixes for three property methods.

```
raw_property_prefixes("GetPrefix","PutPrefix","PutRefPrefix")
```

Parameters

GetPrefix

Prefix to be used for the **propget** methods.

PutPrefix

Prefix to be used for the **propput** methods.

PutRefPrefix

Prefix to be used for the **propputref** methods.

Remarks

By default, low-level **propget**, **propput**, and **propputref** methods are exposed by member functions named with prefixes of **get_**, **put_**, and **putref_** respectively. These prefixes are compatible with the names used in the header files generated by MIDL.

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

rename (#import)

C++ Specific

Works around name collision problems.

```
rename("OldName","NewName")
```

Parameters

OldName

Old name in the type library.

NewName

Name to be used instead of the old name.

Remarks

If this attribute is specified, the compiler replaces all occurrences of *OldName* in a type library with the user-supplied *NewName* in the resulting header files.

This attribute can be used when a name in the type library coincides with a macro definition in the system header files. If this situation is not resolved, then various syntax errors will be generated, such as Compiler Error C2059 and Compiler Error C2061.

Note
The replacement is for a name used in the type library, not for a name used in the resulting header file.

For example, suppose a property named `MyParent` exists in a type library, and a macro `GetMyParent` is defined in a header file and used before **#import**. Since `GetMyParent` is the default name of a wrapper function for the error-handling **get** property, a name collision will occur. To work around the problem, use the following attribute in the **#import** statement:

```
rename("MyParent","MyParentX")
```

which renames the name `MyParent` in the type library. An attempt to rename the `GetMyParent` wrapper name will fail:

```
rename("GetMyParent","GetMyParentX")
```

This is because the name `GetMyParent` only occurs in the resulting type library header file.

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

rename_namespace

C++ Specific

Renames the namespace that contains the contents of the type library.

```
rename_namespace("NewName")
```

Parameters

NewName

The new name of the namespace.

Remarks

It takes a single argument, *NewName*, which specifies the new name for the namespace.

To remove the namespace, use the no_namespace attribute instead.

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

rename_search_namespace

C++ Specific

Has the same functionality as the rename_namespace attribute but is used on type libraries that you use the #import directive with the auto_search attribute.

```
rename_search_namespace("NewName")
```

Parameters

NewName

The new name of the namespace.

Remarks

END C++ Specific

See Also

Reference

#import Attributes

The #import Directive

tlbid

C++ Specific

Allows for loading libraries other than the primary type library.

```
tlbid(number)
```

Parameters
number

The number of the type library in *filename*.

Remarks

If multiple type libraries are built into a single DLL, it possible to load libraries other than the primary type library by using **tlbid**.

For example:

```
#import <MyResource.dll> tlbid(2)
```

is equivalent to:

```
LoadTypeLib("MyResource.dll\\2");
```

END C++ Specific

See Also
Reference
#import Attributes
The #import Directive

The #include Directive

The **#include** directive tells the preprocessor to treat the contents of a specified file as if those contents had appeared in the source program at the point where the directive appears.

```
#include  "path-spec"
#include  <path-spec>
```

Remarks

You can organize constant and macro definitions into include files and then use **#include** directives to add these definitions to any source file. Include files are also useful for incorporating declarations of external variables and complex data types. You need to define and name the types only once in an include file created for that purpose.

The *path-spec* is a filename optionally preceded by a directory specification. The filename must name an existing file. The syntax of the *path-spec* depends on the operating system on which the program is compiled.

For information on how to reference assemblies in a C++ application compiled with /clr, see #using.

Both syntax forms cause replacement of that directive by the entire contents of the specified include file. The difference between the two forms is the order in which the preprocessor searches for header files when the path is incompletely specified.

Syntax Form	Action
Quoted form	This form instructs the preprocessor to look for include files in the same directory of the file that contains the **#include** statement, and then in the directories of any files that include (**#include**) that file. The preprocessor then searches along the path specified by the /I compiler option, then along paths specified by the INCLUDE environment variable.
Angle-bracket form	This form instructs the preprocessor to search for include files first along the path specified by the /I compiler option, then, when compiling from the command line, along the path specified by the INCLUDE environment variable.

The preprocessor stops searching as soon as it finds a file with the given name. If you specify a complete, unambiguous path specification for the include file between double quotation marks (**" "**), the preprocessor searches only that path specification and ignores the standard directories.

If the filename enclosed in double quotation marks is an incomplete path specification, the preprocessor first searches the "parent" file's directory. A parent file is the file containing the **#include** directive. For example, if you include a file named `file2` within a file named `file1`, `file1` is the parent file.

Include files can be "nested"; that is, an **#include** directive can appear in a file named by another **#include** directive. For example, `file2`, above, could include `file3`. In this case, `file1` would still be the parent of `file2` but would be the "grandparent" of `file3`.

When include files are nested and when compiling from the command line, directory searching begins with the directories of the parent file and then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source currently being processed. If the file is not found, the search moves to directories specified by the /I compiler option. Finally, the directories specified by the INCLUDE environment variable are searched.

From within the development environment, the INCLUDE environment variable is ignored. To set the directories searched for include files (this information also applies to the LIB environment variable.), see VC++ Directories, Projects, Options Dialog Box.

The following example shows file inclusion using angle brackets:

```
#include <stdio.h>
```

This example adds the contents of the file named STDIO.H to the source program. The angle brackets cause the preprocessor to search the directories specified by the INCLUDE environment variable for STDIO.H, after searching directories specified by the

/I compiler option.

The following example shows file inclusion using the quoted form:

```
#include "defs.h"
```

This example adds the contents of the file specified by DEFS.H to the source program. The double quotation marks mean that the preprocessor searches the directory containing the parent source file first.

Nesting of include files can continue up to 10 levels. Once the nested **#include** is processed, the preprocessor continues to insert the enclosing include file into the original source file.

Microsoft Specific

To locate includable source files, the preprocessor first searches the directories specified by the /I compiler option. If the /I option is not present or fails, the preprocessor uses the INCLUDE environment variable to find any include files within angle brackets. The INCLUDE environment variable and /I compiler option can contain multiple paths separated by semicolons (**;**). If more than one directory appears as part of the /I option or within the INCLUDE environment variable, the preprocessor searches them in the order in which they appear.

For example, the command

```
CL /ID:\MSVC\INCLUDE MYPROG.C
```

causes the preprocessor to search the directory D:\MSVC\INCLUDE for include files such as STDIO.H. The commands

```
SET INCLUDE=D:\MSVC\INCLUDE
CL MYPROG.C
```

have the same effect. If both sets of searches fail, a fatal compiler error is generated.

If the filename is fully specified for an include file with a path that includes a colon (for example, F:\MSVC\SPECIAL\INCL\TEST.H), the preprocessor follows the path.

For include files specified as **#include "***path-spec***"**, directory searching begins with the directory of the parent file and then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file containing the **#include** directive being processed. If there is no grandparent file and the file has not been found, the search continues as if the filename were enclosed in angle brackets.

END Microsoft Specific

See Also
Reference
Preprocessor Directives

The #line Directive

The **#line** directive tells the preprocessor to change the compiler's internally stored line number and filename to a given line number and filename.

```
#line digit-sequence ["filename"]
```

Remarks

The compiler uses the line number and optional filename to refer to errors that it finds during compilation. The line number usually refers to the current input line, and the filename refers to the current input file. The line number is incremented after each line is processed.

The *digit-sequence* value can be any integer constant. Macro replacement can be performed on the preprocessing tokens, but the result must evaluate to the correct syntax. The *filename* can be any combination of characters and must be enclosed in double quotation marks (**" "**). If *filename* is omitted, the previous filename remains unchanged.

You can alter the source line number and filename by writing a **#line** directive. The translator uses the line number and filename to determine the values of the predefined macros **__FILE__** and **__LINE__**. You can use these macros to insert self-descriptive error messages into the program text. For more information on these predefined macros, see Predefined Macros.

The **__FILE__** macro expands to a string whose contents are the filename, surrounded by double quotation marks (**" "**).

If you change the line number and filename, the compiler ignores the previous values and continues processing with the new values. The **#line** directive is typically used by program generators to cause error messages to refer to the original source file instead of to the generated program.

The following examples illustrate **#line** and the **__LINE__** and **__FILE__** macros.

In this statement, the internally stored line number is set to 151 and the filename is changed to `copy.c`.

```
#line 151 "copy.c"
```

In this example, the macro `ASSERT` uses the predefined macros **__LINE__** and **__FILE__** to print an error message about the source file if a given "assertion" is not true.

```
#define ASSERT(cond)

if( !(cond) )\
{printf( "assertion error line %d, file(%s)\n", \
__LINE__, __FILE__ );}
```

See Also

Reference

Preprocessor Directives

The Null Directive

The null preprocessor directive is a single number sign (**#**) alone on a line. It has no effect.

```
#
```

See Also

Reference

Preprocessor Directives

The #undef Directive

Removes (undefines) a name previously created with **#define**.

```
#undef identifier
```

Remarks

The **#undef** directive removes the current definition of *identifier*. Consequently, subsequent occurrences of *identifier* are ignored by the preprocessor. To remove a macro definition using **#undef**, give only the macro *identifier* ; do not give a parameter list.

You can also apply the **#undef** directive to an identifier that has no previous definition. This ensures that the identifier is undefined. Macro replacement is not performed within **#undef** statements.

The **#undef** directive is typically paired with a **#define** directive to create a region in a source program in which an identifier has a special meaning. For example, a specific function of the source program can use manifest constants to define environment-specific values that do not affect the rest of the program. The **#undef** directive also works with the **#if** directive to control conditional compilation of the source program. See The #if, #elif, #else, and #endif Directives for more information.

In the following example, the **#undef** directive removes definitions of a symbolic constant and a macro. Note that only the identifier of the macro is given.

```
#define WIDTH 80
#define ADD( X, Y ) ((X) + (Y))
.
.
.
#undef WIDTH
#undef ADD
```

Microsoft Specific

Macros can be undefined from the command line using the /U option, followed by the macro names to be undefined. The effect of issuing this command is equivalent to a sequence of **#undef** *macro-name* statements at the beginning of the file.

END Microsoft Specific

See Also

Reference

Preprocessor Directives

The #using Directive

Imports metadata into a program compiled with /clr.

```
#using file [as_friend]
```

Parameters

file

An MSIL .dll, .exe, .netmodule, or .obj. For example,

```
#using <MyComponent.dll>
```

as_friend

Specifies that all types in *file* are accessible. For more information, see Friend Assemblies.

Remarks

file can be a Microsoft intermediate language (MSIL) file that you import for its managed data and managed constructs. If a .dll file contains an assembly manifest, then all the .dlls referenced in the manifest are imported and the assembly you are building will list *file* in the metadata as an assembly reference.

If *file* does not contain an assembly (if *file* is a module) and if you do not intend to use type information from the module in the current (assembly) application, you have the option of just indicating that the module is part the assembly; use /ASSEMBLYMODULE. The types in the module would then be available to any application that referenced the assembly.

An alternative to use **#using** is the /FU compiler option.

.exe assemblies passed to **#using** should be compiled with **/clr:safe** or **/clr:pure**, or with any of the other Visual Studio compilers (Visual Basic, Visual J#, Visual C#, for example). Attempting to import metadata from an .exe assembly compiled with **/clr** will result in a file load exception.

Note
A component that is referenced with **#using** can be run with a different version of the file imported at compile time, causing a client application to give unexpected results.

In order for the compiler to recognize a type in an assembly (not a module), it needs to be forced to resolve the type, which you can do, for example, by defining an instance of the type. There are other ways to resolve type names in an assembly for the compiler, for example, if you inherit from a type in an assembly, the type name will then become known to the compiler.

When importing metadata built from source code that used __declspec(thread), the thread semantics are not persisted in metadata. For example, a variable declared with **__declspec(thread)**, compiled in a program that is build for the .NET Framework common language runtime, and then imported via **#using**, will no longer have **__declspec(thread)** semantics on the variable.

All imported types (both managed and native) in a file referenced by **#using** are available, but the compiler treats native types as declarations not definitions.

mscorlib.dll is automatically referenced when compiling with **/clr**.

The LIBPATH environment variable specifies the directories that will be searched when the compiler tries to resolve file names passed to **#using**.

The compiler will search for references along the following path:

- A path specified in the **#using** statement.
- The current directory.
- The .NET Framework system directory.
- Directories added with the /AI compiler option.
- Directories on LIBPATH environment variable.

Example

If you build an assembly (C) and reference an assembly (B) that itself references another assembly (A), you will not have to explicitly reference assembly A unless you explicitly use one of A's types in C.

```
// using_assembly_A.cpp
// compile with: /clr /LD
public ref class A {};
```

```
// using_assembly_B.cpp
// compile with: /clr /LD
#using "using_assembly_A.dll"
public ref class B {
public:
   void Test(A a) {}
   void Test() {}
};
```

In the following sample, there is no compiler error for not referencing using_assembly_A.dll because the program does not use any of the types defined in using_assembly_A.cpp.

```
// using_assembly_C.cpp
// compile with: /clr
#using "using_assembly_B.dll"
int main() {
   B b;
   b.Test();
}
```

See Also

Reference

Preprocessor Directives

Preprocessor Operators

Four preprocessor-specific operators are used in the context of the **#define** directive (see the following list for a summary of each). The stringizing, charizing, and token-pasting operators are discussed in the next three sections. For information on the **defined** operator, see The #if, #elif, #else, and #endif Directives.

Operator	Action
Stringizing operator (#)	Causes the corresponding actual argument to be enclosed in double quotation marks
Charizing operator (#@)	Causes the corresponding argument to be enclosed in single quotation marks and to be treated a s a character (Microsoft Specific)
Token-pasting operator (##)	Allows tokens used as actual arguments to be concatenated to form other tokens
defined operator	Simplifies the writing of compound expressions in certain macro directives

See Also

Reference

Preprocessor Directives

Predefined Macros

Other Resources

C/C++ Preprocessor Reference

Stringizing Operator (#)

The number-sign or "stringizing" operator (**#**) converts macro parameters to string literals without expanding the parameter definition. It is used only with macros that take arguments. If it precedes a formal parameter in the macro definition, the actual argument passed by the macro invocation is enclosed in quotation marks and treated as a string literal. The string literal then replaces each occurrence of a combination of the stringizing operator and formal parameter within the macro definition.

Note

The Microsoft C (versions 6.0 and earlier) extension to the ANSI C standard that previously expanded macro formal argumen ts appearing inside string literals and character constants is no longer supported. Code that relied on this extension should b e rewritten using the stringizing (**#**) operator.

White space preceding the first token of the actual argument and following the last token of the actual argument is ignored. Any white space between the tokens in the actual argument is reduced to a single white space in the resulting string literal. Thus, if a comment occurs between two tokens in the actual argument, it is reduced to a single white space. The resulting string literal is automatically concatenated with any adjacent string literals from which it is separated only by white space.

Further, if a character contained in the argument usually requires an escape sequence when used in a string literal (for example, the quotation mark (**"**) or backslash (****) character), the necessary escape backslash is automatically inserted before the character.

The Visual C++ stringizing operator may not behave as expected in all situations; see 16.3.2 The # Operator for more information.

Example

The following example shows a macro definition that includes the stringizing operator and a main function that invokes the macro:

Such invocations would be expanded during preprocessing, producing the following code:

```
int main() {
   printf_s( "In quotes in the printf function call\n" "\n" );
   printf_s( "\"In quotes when printed to the screen\"\n" "\n" );
   printf_s( "\"This: \\\" prints an escaped double quote\"" "\n" );
}
```

```
// stringizer.cpp
#include <stdio.h>
#define stringer( x ) printf_s( #x "\n" )
int main() {
   stringer( In quotes in the printf function call );
   stringer( "In quotes when printed to the screen" );
   stringer( "This: \"  prints an escaped double quote" );
}
```

Output

```
In quotes in the printf function call
"In quotes when printed to the screen"
"This: \"  prints an escaped double quote"
```

The following sample shows how you can expand a macro parameter:

```
// stringizer_2.cpp
// compile with: /E
#define F abc
#define B def
#define FB(arg) #arg
#define FB1(arg) FB(arg)
FB(F B)
FB1(F B)
```

See Also

Reference

Preprocessor Operators

Charizing Operator (#@)

Microsoft Specific

The charizing operator can be used only with arguments of macros. If **#@** precedes a formal parameter in the definition of the macro, the actual argument is enclosed in single quotation marks and treated as a character when the macro is expanded. For example:

```
#define makechar(x)  #@x
```

causes the statement

```
a = makechar(b);
```

to be expanded to

```
a = 'b';
```

The single-quotation character cannot be used with the charizing operator.

END Microsoft Specific

See Also
Reference
Preprocessor Operators

Token-Pasting Operator (##)

The double-number-sign or "token-pasting" operator (**##**), which is sometimes called the "merging" operator, is used in both object-like and function-like macros. It permits separate tokens to be joined into a single token and therefore cannot be the first or last token in the macro definition.

If a formal parameter in a macro definition is preceded or followed by the token-pasting operator, the formal parameter is immediately replaced by the unexpanded actual argument. Macro expansion is not performed on the argument prior to replacement.

Then, each occurrence of the token-pasting operator in *token-string* is removed, and the tokens preceding and following it are concatenated. The resulting token must be a valid token. If it is, the token is scanned for possible replacement if it represents a macro name. The identifier represents the name by which the concatenated tokens will be known in the program before replacement. Each token represents a token defined elsewhere, either within the program or on the compiler command line. White space preceding or following the operator is optional.

This example illustrates use of both the stringizing and token-pasting operators in specifying program output:

```
#define paster( n ) printf_s( "token" #n " = %d", token##n )
int token9 = 9;
```

If a macro is called with a numeric argument like

```
paster( 9 );
```

the macro yields

```
printf_s( "token" "9" " = %d", token9 );
```

which becomes

```
printf_s( "token9 = %d", token9 );
```

Example

```
// preprocessor_token_pasting.cpp
#include <stdio.h>
#define paster( n ) printf_s( "token" #n " = %d", token##n )
int token9 = 9;

int main()
{
   paster(9);
}
```

Output

```
token9 = 9
```

See Also

Reference

Preprocessor Operators

Macros (C/C++)

Preprocessing expands macros in all lines that are not preprocessor directives (lines that do not have a **#** as the first non-white-space character) and in parts of some directives that are not skipped as part of a conditional compilation. "Conditional compilation" directives allow you to suppress compilation of parts of a source file by testing a constant expression or identifier to determine which text blocks are passed on to the compiler and which text blocks are removed from the source file during preprocessing.

The **#define** directive is typically used to associate meaningful identifiers with constants, keywords, and commonly used statements or expressions. Identifiers that represent constants are sometimes called "symbolic constants" or "manifest constants." Identifiers that represent statements or expressions are called "macros." In this preprocessor documentation, only the term "macro" is used.

When the name of the macro is recognized in the program source text or in the arguments of certain other preprocessor commands, it is treated as a call to that macro. The macro name is replaced by a copy of the macro body. If the macro accepts arguments, the actual arguments following the macro name are substituted for formal parameters in the macro body. The process of replacing a macro call with the processed copy of the body is called "expansion" of the macro call.

In practical terms, there are two types of macros. "Object-like" macros take no arguments, whereas "function-like" macros can be defined to accept arguments so that they look and act like function calls. Because macros do not generate actual function calls, you can sometimes make programs run faster by replacing function calls with macros. (In C++, inline functions are often a preferred method.) However, macros can create problems if you do not define and use them with care. You may have to use parentheses in macro definitions with arguments to preserve the proper precedence in an expression. Also, macros may not correctly handle expressions with side effects. See the `getrandom` example in The #define Directive for more information.

Once you have defined a macro, you cannot redefine it to a different value without first removing the original definition. However, you can redefine the macro with exactly the same definition. Thus, the same definition can appear more than once in a program.

The #**undef** directive removes the definition of a macro. Once you have removed the definition, you can redefine the macro to a different value. The #define Directive and The #undef Directive discuss the **#define** and **#undef** directives, respectively.

For more information, see,

- Macros and C++
- Variadic Macros
- Predefined Macros

See Also

Other Resources

C/C++ Preprocessor Reference

Macros and C++

C++ offers new capabilities, some of which supplant those offered by the ANSI C preprocessor. These new capabilities enhance the type safety and predictability of the language:

- In C++, objects declared as **const** can be used in constant expressions. This allows programs to declare constants that have type and value information, and enumerations that can be viewed symbolically with the debugger. Using the preprocessor **#define** directive to define constants is not as precise. No storage is allocated for a **const** object unless an expression that takes its address is found in the program.
- The C++ inline function capability supplants function-type macros. The advantages of using inline functions over macros are:
    - Type safety. Inline functions are subject to the same type checking as normal functions. Macros are not type safe.
    - Correct handling of arguments that have side effects. Inline functions evaluate the expressions supplied as arguments prior to entering the function body. Therefore, there is no chance that an expression with side effects will be unsafe.

For more information on inline functions, see inline, __inline, __forceinline.

For backward compatibility, all preprocessor facilities that existed in ANSI C and in earlier C++ specifications are preserved for Microsoft C++.

See Also
Reference
Predefined Macros
Macros (C/C++)

Variadic Macros

Variadic macros are function-like macros that contain a variable number of arguments.

Remarks

To use variadic macros, the ellipsis may be specified as the final formal argument in a macro definition, and the replacement identifier **__VA_ARGS__** may be used in the definition to insert the extra arguments. **__VA_ARGS__** is replaced by all of the arguments that match the ellipsis, including commas between them.

The C Standard specifies that at least one argument must be passed to the ellipsis, to ensure that the macro does not resolve to an expression with a trailing comma. The Visual C++ implementation will suppress a trailing comma if no arguments are passed to the ellipsis.

Support for variadic macros was introduced in Visual C++ 2005.

Example

```
// variadic_macros.cpp
#include <stdio.h>
#define EMPTY

#define CHECK1(x, ...) if (!(x)) { printf(__VA_ARGS__); }
#define CHECK2(x, ...) if ((x)) { printf(__VA_ARGS__); }
#define CHECK3(...) { printf(__VA_ARGS__); }
#define MACRO(s, ...) printf(s, __VA_ARGS__)

int main() {
   CHECK1(0, "here %s %s %s", "are", "some", "varargs1(1)\n");
   CHECK1(1, "here %s %s %s", "are", "some", "varargs1(2)\n");   // won't print

   CHECK2(0, "here %s %s %s", "are", "some", "varargs2(3)\n");   // won't print
   CHECK2(1, "here %s %s %s", "are", "some", "varargs2(4)\n");

   // always invokes printf in the macro
   CHECK3("here %s %s %s", "are", "some", "varargs3(5)\n");

   MACRO("hello, world\n");
   // MACRO("error\n", EMPTY);   would cause C2059
}
```

Output

```
here are some varargs1(1)
here are some varargs2(4)
here are some varargs3(5)
hello, world
```

See Also

Reference

Macros (C/C++)

Predefined Macros

Names the predefined ANSI C and Microsoft C++ implementation macros.

The compiler recognizes predefined ANSI C macros and the Microsoft C++ implementation provides several more. These macros take no arguments and cannot be redefined. Some of the predefined macros listed below are defined with multiple values. See the following tables for more information.

ANSI-Compliant Predefined Macros

Macro	Description
__DATE__	The compilation date of the current source file. The date is a string literal of the form *Mmm dd yyyy*. The month name *Mmm* is the same as for dates generated by the library function **asctime** declared in TIME.H.

__FILE__	The name of the current source file. **__FILE__** expands to a string surrounded by double quotation marks. To ensure that the full path to the file is displayed, use /FC (Full Path of Source Code File in Diagnostics).
	You can create your own wide string version of **__FILE__** as follows:
	`#include <stdio.h>`
	`#define WIDEN2(x) L ## x`
	`#define WIDEN(x) WIDEN2(x)`
	`#define __WFILE__ WIDEN(__FILE__)`
	`wchar_t *pwsz = __WFILE__;`
	
	`int main() {}`

__LINE__	The line number in the current source file. The line number is a decimal integer constant. It can be altered with a **#line** directive.
__STDC__	Indicates full conformance with the ANSI C standard. Defined as the integer constant 1 only if the /Za compiler option is given and you are not compiling C++ code; otherwise is undefined.
__TIME__	The most recent compilation time of the current source file. The time is a string literal of the form *hh:mm:ss*.
__TIMESTAMP__	The date and time of the last modification of the current source file, expressed as a string literal in the form *Ddd Mmm Date hh:mm:ss yyyy*, where *Ddd* is the abbreviated day of the week and *Date* is an integer from 1 to 31.

Microsoft-Specific Predefined Macros

Macro	Description
_ATL_VER	Defines the ATL version.
_CHAR_UNSIGNED	Default **char** type is unsigned. Defined when /J is specified.

__CLR_VER	Defines the version of the common language runtime used when the application was compiled. The value returned will be in the following format:
	
	Mmmbbbbb
	
	where,
	
	M is the major version of the runtime.
	
	mm is the minor version of the runtime.
	
	bbbbb is the build number.
	
	`// clr_ver.cpp`
	`// compile with: /clr`
	`using namespace System;`
	`int main() {`
	`  Console::WriteLine(__CLR_VER);`
	`}`

__cplusplus_cli	Defined when compiling with **/clr**, **/clr:pure**, or **/clr:safe**. Value of __cplusplus_cli is 200406. __cplusplus_cli is in effect throughout the translation unit.
	
	`// cplusplus_cli.cpp`
	`// compile with: /clr`
	`#include "stdio.h"`
	`int main() {`
	`  #ifdef __cplusplus_cli`
	`  printf("%d\n", __cplusplus_cli);`
	`  #else`
	`  printf("not defined\n");`
	`  #endif`
	`}`

__COUNTER__	Expands to an integer starting with 0 and incrementing by 1 every time it is used in a compiland. **__COUNTER__** remembers its state when using precompiled headers. If the last **__COUNTER__** value was 4 after building a precompiled header (PCH), it will start with 5 on each PCH use.
	
	__COUNTER__ lets you generate unique variable names. You can use token pasting with a prefix to make a unique name. For example:
	
	`// pre_mac_counter.cpp`
	`#include <stdio.h>`
	`#define FUNC2(x,y) x##y`
	`#define FUNC1(x,y) FUNC2(x,y)`
	`#define FUNC(x) FUNC1(x,__COUNTER__)`
	
	`int FUNC(my_unique_prefix);`
	`int FUNC(my_unique_prefix);`
	
	`int main() {`
	`  my_unique_prefix0 = 0;`
	`  printf_s("\n%d",my_unique_prefix0);`
	`  my_unique_prefix0++;`
	`  printf_s("\n%d",my_unique_prefix0);`
	`}`

__cplusplus	Defined for C++ programs only.
_CPPLIB_VER	Defined if you include any of the C++ Standard Library headers; reports which version of the Dinkumware header files are present.

_CPPRTTI	Defined for code compiled with /GR (Enable Run-Time Type Information).
_CPPUNWIND	Defined for code compiled with /GX (Enable Exception Handling).
_DEBUG	Defined when compiling with /LDd, /MDd, and /MTd.
_DLL	Defined when /MD or /MDd (Multithread DLL) is specified.
__FUNCDNAME__	Valid only within a function and returns the decorated name of the enclosing function (as a string). **__FUNCDNAME__** is not expanded if you use the /EP or /P compiler option.
__FUNCSIG__	Valid only within a function and returns the signature of the enclosing function (as a string). **__FUNCSIG__** is not expanded if you use the /EP or /P compiler option. On a 64-bit operating system, the calling convention is **__cdecl** by default.
__FUNCTION__	Valid only within a function and returns the undecorated name of the enclosing function (as a string). **__FUNCTION__** is not expanded if you use the /EP or /P compiler option.

_INTEGRAL_MAX_BITS	Reports the maximum size (in bits) for an integral type.
	`// integral_max_bits.cpp`
	`#include <stdio.h>`
	`int main() {`
	`  printf("%d\n", _INTEGRAL_MAX_BITS);`
	`}`

_M_ALPHA	Defined for DEC ALPHA platforms (no longer supported).
_M_CEE	Defined for a compilation that uses any form of **/clr** (**/clr:oldSyntax**, **/clr:safe**, for example).
_M_CEE_PURE	Defined for a compilation that uses **/clr:pure**.
_M_CEE_SAFE	Defined for a compilation that uses **/clr:safe**.
_M_IX86	Defined for x86 processors. See Values for _M_IX86 for more details.
_M_IA64	Defined for Itanium Processor Family 64-bit processors.
_M_IX86_FP	Expands to a value indicating which **/arch** compiler option was used: • 0 if **/arch** was not used. • 1 if **/arch:SSE** was used. • 2 if **/arch:SSE2** was used. • See /arch (Minimum CPU Architecture) for more information.
_M_MPPC	Defined for Power Macintosh platforms (no longer supported).
_M_MRX000	Defined for MIPS platforms (no longer supported).

_M_PPC	Defined for PowerPC platforms (no longer supported).
_M_X64	Defined for x64 processors.
_MANAGED	Defined to be 1 when /clr is specified.
_MFC_VER	Defines the MFC version. For example, 0x0700 represents MFC version 7.
_MSC_EXTENSIONS	This macro is defined when compiling with the /Ze compiler option (the default). Its value, when defined, is 1.
_MSC_VER	Reports the major and minor versions of the compiler. For example, 1310 for Microsoft Visual C++ .NET 2003. 1310 represents version 13 and a 1.0 point release. The Visual C++ 2005 compiler version is 1400. Type **cl /?** at the command line to see the major and minor versions of your compiler along with the build number.
__MSVC_RUNTIME_CHECKS	Defined when one of the /RTC compiler options is specified.
_MT	Defined when /MD or /MDd (Multithreaded DLL) or /MT or /MTd (Multithreaded) is specified.
_NATIVE_WCHAR_T_DEFINED	Defined when /Zc:wchar_t is used.

_OPENMP	Defined when compiling with /openmp, returns an integer representing the date of the OpenMP specification implemented by Visual C++.
	
	`// _OPENMP_dir.cpp`
	`// compile with: /openmp`
	`#include <stdio.h>`
	`int main() {`
	`  printf("%d\n", _OPENMP);`
	`}`

_VC_NODEFAULTLIB	Defined when **/Zl** is used; see /Zl (Omit Default Library Name) for more information.
_WCHAR_T_DEFINED	Defined when /Zc:wchar_t is used or if wchar_t is defined in a system header file included in your project.
_WIN32	Defined for applications for Win32 and Win64. Always defined.
_WIN64	Defined for applications for Win64.
_Wp64	Defined when specifying /Wp64.

As shown in following table, the compiler generates a value for the preprocessor identifiers that reflect the processor option specified.

Values for _M_IX86

Option in Development Environment	Command-Line Option	Resulting Value

Blend	/GB	**_M_IX86 = 600** (Default. Future compilers will emit a different value to reflect the dominant processor.)
Pentium	/G5	**_M_IX86 = 500**
Pentium Pro, Pentium II, and Pentium III	/G6	**_M_IX86 = 600**
80386	/G3	**_M_IX86 = 300**
80486	/G4	**_M_IX86 = 400**

See Also

Reference

Macros (C/C++)

Preprocessor Operators

Preprocessor Directives

Pragma Directives

```
#pragma token-string
```

Remarks

Each implementation of C and C++ supports some features unique to its host machine or operating system. Some programs, for instance, need to exercise precise control over the memory areas where data is placed or to control the way certain functions receive parameters. The **#pragma** directives offer a way for each compiler to offer machine- and operating system-specific features while retaining overall compatibility with the C and C++ languages. Pragmas are machine- or operating system-specific by definition, and are usually different for every compiler.

Pragmas can be used in conditional statements, to provide new preprocessor functionality, or to provide implementation-defined information to the compiler. The Microsoft C and C++ compilers recognize the following pragmas:

alloc_text	auto_inline	bss_seg
check_stack	code_seg	comment
component	conform[1]	const_seg
data_seg	deprecated	fenv_access
float_control	fp_contract	function
hdrstop	include_alias	init_seg[1]
inline_depth	inline_recursion	intrinsic
make_public	managed	message
omp	once	optimize
pack	pointers_to_members[1]	pop_macro
push_macro	region, endregion	runtime_checks
section	setlocale	strict_gs_check
unmanaged	vtordisp[1]	warning

1. Supported only by the C++ compiler.

The *token-string* is a series of characters that gives a specific compiler instruction and arguments, if any. The number sign (**#**) must be the first non-white-space character on the line containing the pragma; white-space characters can separate the number sign and the word **pragma**. Following **#pragma**, write any text that the translator can parse as preprocessing tokens. The argument to **#pragma** is subject to macro expansion.

If the compiler finds a pragma it does not recognize, it issues a warning, but compilation continues.

Some pragmas provide the same functionality as compiler options. When a pragma is encountered in source code, it overrides the behavior specified by the compiler option. For example, if you specified /Zp8, you can override this compiler setting for specific portions of the code with pack:

```
cl /Zp8 ...
```

```
<file> - packing is 8
// ...
#pragma pack(push, 1) - packing is now 1
// ...
#pragma pack(pop) - packing is 8
</file>
```

See Also

Other Resources

C/C++ Preprocessor Reference

alloc_text

Names the code section where the specified function definitions are to reside. The pragma must occur between a function declarator and the function definition for the named functions.

```
#pragma alloc_text( "textsection", function1, ... )
```

Remarks

The **alloc_text** pragma does not handle C++ member functions or overloaded functions. It is applicable only to functions declared with C linkage — that is, functions declared with the **extern "C"** linkage specification. If you attempt to use this pragma on a function with C++ linkage, a compiler error is generated.

Since function addressing using **__based** is not supported, specifying section locations requires the use of the **alloc_text** pragma. The name specified by *textsection* should be enclosed in double quotation marks.

The **alloc_text** pragma must appear after the declarations of any of the specified functions and before the definitions of these functions.

Functions referenced in an **alloc_text** pragma should be defined in the same module as the pragma. If this is not done and an undefined function is later compiled into a different text section, the error may or may not be caught. Although the program will usually run correctly, the function will not be allocated in the intended sections.

Other limitations on **alloc_text** are as follows:

- It cannot be used inside a function.
- It must be used after the function has been declared, but before the function has been defined.

See Also

Reference

Pragma Directives

auto_inline

Excludes any functions defined within the range where **off** is specified from being considered as candidates for automatic inline expansion.

```
#pragma auto_inline( [{on | off}] )
```

Remarks

To use the **auto_inline** pragma, place it before and immediately after (not in) a function definition. The pragma takes effect at the first function definition after the pragma is seen.

See Also

Reference

Pragma Directives

bss_seg

Specifies the segment where uninitialized variables are stored in the .obj file.

```
#pragma bss_seg( [ [ { push | pop }, ] [ identifier, ] ] [ "segment-name" [, "segment-class
" ] ] )
```

Remarks

Obj files can be viewed with the dumpbin application. The default segment in the .obj file for uninitialized data is .bss. In some cases use of **bss_seg** can speed load times by grouping uninitialized data into one section.

bss_seg with no parameters resets the segment to .bss.

push(optional)

Puts a record on the internal compiler stack. A **push** can have an *identifier* and *segment-name*.

pop (optional)

Removes a record from the top of the internal compiler stack.

identifier (optional)

When used with **push**, assigns a name to the record on the internal compiler stack. When used with **pop**, pops records off the internal stack until *identifier* is removed; if *identifier* is not found on the internal stack, nothing is popped.

identifier enables multiple records to be popped with a single **pop** command.

"*segment-name*"(optional)

The name of a segment. When used with **pop**, the stack is popped and *segment-name* becomes the active segment name.

"*segment-class*" (optional)

Included for compatibility with C++ prior to version 2.0. It is ignored.

Example

```
// pragma_directive_bss_seg.cpp
int i;                      // stored in .bss
#pragma bss_seg(".my_data1")
int j;                      // stored in "my_data1"

#pragma bss_seg(push, stack1, ".my_data2")
int l;                      // stored in "my_data2"

#pragma bss_seg(pop, stack1)   // pop stack1 from stack
int m;                      // stored in "stack_data1"

int main() {
}
```

You can also specify sections for initialized data (data_seg), functions (code_seg), and const variables (const_seg).

Data allocated using the **bss_seg** pragma does not retain any information about its location.

See /SECTION for a list of names you should not use when creating a section.

See Also

Reference

Pragma Directives

check_stack

Instructs the compiler to turn off stack probes if **off** (or **–**) is specified, or to turn on stack probes if **on** (or **+**) is specified.

```
#pragma check_stack([ {on | off}] )
#pragma check_stack{+ | -}
```

Remarks

If no argument is given, stack probes are treated according to the default. This pragma takes effect at the first function defined after the pragma is seen. Stack probes are neither a part of macros nor of functions that are generated inline.

If you don't give an argument for the **check_stack** pragma, stack checking reverts to the behavior specified on the command line. For more information, see Compiler Reference. The interaction of the **#pragma check_stack** and the /Gs option is summarized in the following table.

Using the check_stack Pragma

Syntax	Compiled with /Gs option?	Action
#pragma check_stack( ) or **#pragma check_stack**	Yes	Turns off stack checking for functions that follow
#pragma check_stack( ) or **#pragma check_stack**	No	Turns on stack checking for functions that follow
#pragma check_stack(on) or **#pragma check_stack +**	Yes or No	Turns on stack checking for functions that follow
#pragma check_stack(off) or **#pragma check_stack –**	Yes or No	Turns off stack checking for functions that follow

See Also

Reference

Pragma Directives

code_seg

Specifies the segment where functions are stored in the .obj file.

```
#pragma code_seg( [ [ { push | pop}, ] [ identifier, ] ] [ "segment-name" [, "segment-class
" ] )
```

Remarks

OBJ files can be viewed with the dumpbin application. The default segment in the .obj file for functions is .text.

code_seg with no parameters resets the segment to .text.

push(optional)

Puts a record on the internal compiler stack. A **push** can have an *identifier* and *segment-name*.

pop (optional)

Removes a record from the top of the internal compiler stack.

identifier (optional)

When used with **push**, assigns a name to the record on the internal compiler stack. When used with **pop**, pops records off the internal stack until *identifier* is removed; if *identifier* is not found on the internal stack, nothing is popped.

identifier enables multiple records to be popped with a single **pop** command.

"*segment-name*"(optional)

The name of a segment. When used with **pop**, the stack is popped and *segment-name* becomes the active segment name.

"*segment-class*" (optional)

Included for compatibility with C++ prior to version 2.0. It is ignored.

Example

```
// pragma_directive_code_seg.cpp
void func1() {                  // stored in .text
}

#pragma code_seg(".my_data1")
void func2() {                  // stored in my_data1
}

#pragma code_seg(push, r1, ".my_data2")
void func3() {                  // stored in my_data2
}

#pragma code_seg(pop, r1)      // stored in my_data1
void func4() {
}

int main() {
}
```

See /SECTION for a list of names you should not use when creating a section.

You can also specify sections for initialized data (data_seg), uninitialized data (bss_seg), and const variables (const_seg).

See Also

Reference

Pragma Directives

comment (C/C++)

Places a comment record into an object file or executable file.

```
#pragma comment( comment-type [,"commentstring"] )
```

Remarks

The *comment-type* is one of the predefined identifiers, described below, that specifies the type of comment record. The optional *commentstring* is a string literal that provides additional information for some comment types. Because *commentstring* is a string literal, it obeys all the rules for string literals with respect to escape characters, embedded quotation marks (**"**), and concatenation.

compiler

Places the name and version number of the compiler in the object file. This comment record is ignored by the linker. If you supply a *commentstring* parameter for this record type, the compiler generates a warning.

exestr

Places *commentstring* in the object file. At link time this string is placed in the executable file. The string is not loaded into memory when the executable file is loaded; however, it can be found with a program that finds printable strings in files. One use for this comment-record type is to embed a version number or similar information in an executable file.

exestr is deprecated and will be removed in a future release; the linker does not process the comment record.

lib

Places a library-search record in the object file. This comment type must be accompanied by a *commentstring* parameter containing the name (and possibly the path) of the library that you want the linker to search. The library name follows the default library-search records in the object file; the linker searches for this library just as if you had named it on the command line provided that the library is not specified with /nodefaultlib. You can place multiple library-search records in the same source file; each record appears in the object file in the same order in which it is encountered in the source file.

If the order of the default library and an added library is important, compiling with the /Zl switch will prevent the default library name from being placed in the object module. A second comment pragma then can be used to insert the name of the default library after the added library. The libraries listed with these pragmas will appear in the object module in the same order they are found in the source code.

linker

Places a linker option in the object file. You can use this comment-type to specify a linker option instead of passing it to the command line or specifying it in the development environment. For example, you can specify the /include option to force the inclusion of a symbol:

```
#pragma comment(linker, "/include:__mySymbol")
```

Only the following (*comment-type*) linker options are available to be passed to the linker identifier:

- /DEFAULTLIB
- /EXPORT
- /INCLUDE
- /MANIFESTDEPENDENCY
- /MERGE
- /SECTION

user

Places a general comment in the object file. The *commentstring* parameter contains the text of the comment. This comment record is ignored by the linker.

The following pragma causes the linker to search for the EMAPI.LIB library while linking. The linker searches first in the current working directory and then in the path specified in the LIB environment variable.

```
#pragma comment( lib, "emapi" )
```

The following pragma causes the compiler to place the name and version number of the compiler in the object file:

```
#pragma comment( compiler )
```

Note

For comments that take a *commentstring* parameter, you can use a macro in any place where you would use a string literal, provided that the macro expands to a string literal. You can also concatenate any combination of string literals and macros th at expand to string literals. For example, the following statement is acceptable:

```
#pragma comment( user, "Compiled on " __DATE__ " at " __TIME__ )
```

See Also

Reference

Pragma Directives

component

Controls the collecting of browse information or dependency information from within source files.

```
#pragma component( browser, { on | off }[, references [, name ]] )
#pragma component( minrebuild, on | off )
#pragma component( mintypeinfo, on | off )
```

Remarks

Browser

You can turn collecting on or off, and you can specify particular names to be ignored as information is collected.

Using on or off controls the collection of browse information from the pragma forward. For example:

```
#pragma component(browser, off)
```

stops the compiler from collecting browse information.

Note
To turn on the collecting of browse information with this pragma, browse information must first be enabled.

The **references** option can be used with or without the *name* argument. Using **references** without *name* turns on or off the collecting of references (other browse information continues to be collected, however). For example:

```
#pragma component(browser, off, references)
```

stops the compiler from collecting reference information.

Using **references** with *name* and **off** prevents references to *name* from appearing in the browse information window. Use this syntax to ignore names and types you are not interested in and to reduce the size of browse information files. For example:

```
#pragma component(browser, off, references, DWORD)
```

ignores references to **DWORD** from that point forward. You can turn collecting of references to **DWORD** back on by using **on**:

```
#pragma component(browser, on, references, DWORD)
```

This is the only way to resume collecting references to *name*; you must explicitly turn on any *name* that you have turned off.

To prevent the preprocessor from expanding *name* (such as expanding **NULL** to **0**), put quotes around it:

```
#pragma component(browser, off, references, "NULL")
```

Minimal Rebuild

The Visual C++ minimal rebuild feature requires that the compiler create and store C++ class dependency information, which takes disk space. To save disk space, you can use `#pragma component( minrebuild, off )` whenever you don't need to collect dependency information, for instance, in unchanging header files. Insert `#pragma component(minrebuild, on)` after unchanging classes to turn dependency collection back on.

Reduce Type Information

The **mintypeinfo** option reduces the debugging information for the region specified. The volume of this information is considerable, impacting .pdb and .obj files. You cannot debug classes and structures in the mintypeinfo region. Use of the mintypeinfo option can be helpful to avoid the following warning:

```
LINK : warning LNK4018: too many type indexes in PDB "filename", discarding subsequent type
information
```

For more information, see the Enable Minimal Rebuild (/Gm) compiler option.

See Also

Reference

Pragma Directives

conform

C++ Specific

Specifies the run-time behavior of the /Zc:forScope compiler option.

```
#pragma conform(name [, show ] [, on | off ] [ [, push | pop ] [, identifier ] ] )
```

Parameters

name

Specifies the name of the compiler option to be modified. The only valid *name* is `forScope`.

show (optional)

Causes the current setting of *name* (true or false) to be displayed by means of a warning message during compilation. For example, `#pragma conform(forScope, show)`.

on, off(optional)

Setting *name* to **on** enables the /Zc:forScope compiler option. The default is **off**.

push (optional)

Pushes the current value of *name* onto the internal compiler stack. If you specify *identifier*, you can specify the **on** or **off** value for *name* to be pushed onto the stack. For example, `#pragma conform(forScope, push, myname, on)`.

pop (optional)

Sets the value of *name* to the value at the top of the internal compiler stack and then pops the stack. If identifier is specified with **pop**, the stack will be popped back until it finds the record with *identifier*, which will also be popped; the current value for *name* in the next record on the stack becomes the new value for *name*. If you specify pop with an *identifier* that is not in a record on the stack, the **pop** is ignored.

identifier(optional)

Can be included with a **push** or **pop** command. If *identifier* is used, then an **on** or **off** specifier can also be used.

Example

```
// pragma_directive_conform.cpp
// compile with: /W1
// C4811 expected
#pragma conform(forScope, show)
#pragma conform(forScope, push, x, on)
#pragma conform(forScope, push, x1, off)
#pragma conform(forScope, push, x2, off)
#pragma conform(forScope, push, x3, off)
#pragma conform(forScope, show)
#pragma conform(forScope, pop, x1)
#pragma conform(forScope, show)

int main() {}
```

See Also

Reference

Pragma Directives

const_seg

Specifies the segment where const variables are stored in the .obj file.

```
#pragma const_seg( [ [ { push | pop}, ] [ identifier, ] ] [ "segment-name" [, "segment-clas
s" ] )
```

Remarks

OBJ files can be viewed with the dumpbin application. The default segment in the .obj file for **const** variables is .rdata. Some **const** variables, such as scalars, are automatically inlined into the code stream. Inlined code will not appear in .rdata.

const_seg with no parameters resets the segment to .rdata.

push(optional)

Puts a record on the internal compiler stack. A **push** can have an *identifier* and *segment-name*.

pop (optional)

Removes a record from the top of the internal compiler stack.

identifier (optional)

When used with **push**, assigns a name to the record on the internal compiler stack. When used with **pop**, pops records off the internal stack until *identifier* is removed; if *identifier* is not found on the internal stack, nothing is popped.

Using *identifier* enables multiple records to be popped with a single **pop** command.

"*segment-name*"(optional)

The name of a segment. When used with **pop**, the stack is popped and *segment-name* becomes the active segment name.

"*segment-class*" (optional)

Included for compatibility with C++ prior to version 2.0. It is ignored.

Example

```
// pragma_directive_const_seg.cpp
// compile with: /EHsc
#include <iostream>

const int i = 7;               // inlined, not stored in .rdata
const char sz1[]= "test1";     // stored in .rdata

#pragma const_seg(".my_data1")
const char sz2[]= "test2";     // stored in .my_data1

#pragma const_seg(push, stack1, ".my_data2")
const char sz3[]= "test3";     // stored in .my_data2

#pragma const_seg(pop, stack1) // pop stack1 from stack
const char sz4[]= "test4";     // stored in .my_data1

int main() {
    using namespace std;
   // const data must be referenced to be put in .obj
   cout << sz1 << endl;
   cout << sz2 << endl;
   cout << sz3 << endl;
   cout << sz4 << endl;
}
```

Output

```
test1
```

```
test2
test3
test4
```

Comments

See /SECTION for a list of names you should not use when creating a section.

You can also specify sections for initialized data (data_seg), uninitialized data (bss_seg), and functions (code_seg).

See Also

Reference

Pragma Directives

data_seg

Specifies the data segment where initialized variables are stored in the .obj file.

```
#pragma data_seg( [ [ { push | pop }, ] [ identifier, ] ] [ "segment-name" [, "segment-clas
s" ] )
```

Remarks

OBJ files can be viewed with the dumpbin application. The default segment in the .obj file for initialized variables is .data. Variables initialized to zero are considered uninitialized and are stored in .bss.

data_seg with no parameters resets the segment to .data.

push(optional)

Puts a record on the internal compiler stack. A **push** can have an *identifier* and *segment-name*.

pop (optional)

Removes a record from the top of the internal compiler stack.

identifier (optional)

When used with **push**, assigns a name to the record on the internal compiler stack. When used with **pop**, pops records off the internal stack until *identifier* is removed; if *identifier* is not found on the internal stack, nothing is popped.

identifier enables multiple records to be popped with a single **pop** command.

"*segment-name*"(optional)

The name of a segment. When used with **pop**, the stack is popped and *segment-name* becomes the active segment name.

"*segment-class*" (optional)

Included for compatibility with C++ prior to version 2.0. It is ignored.

Example

```
// pragma_directive_data_seg.cpp
int h = 1;                    // stored in .data
int i = 0;                    // stored in .bss
#pragma data_seg(".my_data1")
int j = 1;                    // stored in "my_data1"

#pragma data_seg(push, stack1, ".my_data2")
int l = 2;                    // stored in "my_data2"

#pragma data_seg(pop, stack1)   // pop stack1 off the stack
int m = 3;                    // stored in "stack_data1"

int main() {
}
```

Data allocated using **data_seg** does not retain any information about its location.

See /SECTION for a list of names you should not use when creating a section.

You can also specify sections for const variables (const_seg), uninitialized data (bss_seg), and functions (code_seg).

See Also

Reference

Pragma Directives

deprecated (C/C++)

The **deprecated** pragma lets you indicate that a function, type, or any other identifier may no longer be supported in a future release or should no longer be used.

```
#pragma deprecated( identifier1 [,identifier2, ...] )
```

Remarks

When the compiler encounters a deprecated symbol, it issues C4995.

You can deprecate macro names. Place the macro name in quotes or else macro expansion will occur.

The deprecated **__declspec** modifier allows you to specify deprecated status for particular forms of overloaded functions.

Example

```
// pragma_directive_deprecated.cpp
// compile with: /W1
#include <stdio.h>
void func1(void) {
}

void func2(void) {
}

int main() {
   func1();
   func2();
   #pragma deprecated(func1, func2)
   func1();   // C4995
   func2();   // C4995
}
```

The following sample shows how to deprecate a class:

```
// pragma_directive_deprecated2.cpp
// compile with: /W1
#pragma deprecated(X)
class X {  // C4995
public:
   void f(){}
};

int main() {
   X x;   // C4995
}
```

See Also

Reference

Pragma Directives

fenv_access

Disables (ON) or enables (OFF) optimizations that could change flag tests and mode changes.

```
#pragma fenv_access [ON | OFF]
```

Remarks

By default, **fenv_access** is OFF.

For more information on floating-point behavior, see /fp (Specify Floating-Point Behavior).

The kinds of optimizations that are subject to **fenv_access** are:

- Global common subexpression elimination
- Code motion
- Constant folding

Other floating-point pragmas include:

- float_control
- fp_contract

Example

```
// pragma_directive_fenv_access_x86.cpp
// compile with: /O2
// processor: x86
#include <stdio.h>
#include <float.h>
#include <errno.h>
#pragma fenv_access (on)

int main() {
   double z, b = 0.1, t = 0.1;
   unsigned int currentControl;
   errno_t err;

   err = _controlfp_s(&currentControl, _PC_24, _MCW_PC);
   if (err != 0) {
      printf_s("The function _controlfp_s failed!\n");
      return -1;
   }
   z = b * t;
   printf_s ("out=%.15e\n",z);
}
```

Output

```
out=9.999999776482582e-003
```

The following sample is for compiler producing output files for Itanium processors. **/fp:precise** keeps the intermediate results in extended precision where values greater than FLT_MAX (3.402823466e+38F) can be calculated and as a result of that sum will have 1.0 result, as it should if manually calculated. **/fp:strict** keeps intermediate results in their source precision (float) so the first addition produces infinity, which is kept throughout the expression.

```
// pragma_directive_fenv_access_IPF.cpp
// compile with: /O2 /fp:precise
// processor: IPF
// compiling with /fp:precise prints 1.0F
// compile with /fp:strict to print infinity
```

```
#include <stdio.h>
float arr[5] = {3.402823465e+38F,
               3.402823462e+38F,
               3.402823464e+38F,
               3.402823463e+38F,
               1.0F};

int main() {
   float sum = 0;
   sum = arr[0] + arr[1] - arr[2] - arr[3] + arr[4];
   printf_s("%f\n", sum);
}
```

Output

```
1.000000
```

When commenting out `#pragma fenv_access (on)` from the previous sample, note that the output is different because the compiler does compile-time evaluation, which does not use the control mode.

```
// pragma_directive_fenv_access_2.cpp
// compile with: /O2
#include <stdio.h>
#include <float.h>

int main() {
   double z, b = 0.1, t = 0.1;
   unsigned int currentControl;
   errno_t err;

   err = _controlfp_s(&currentControl, _PC_24, _MCW_PC);
   if (err != 0) {
      printf_s("The function _controlfp_s failed!\n");
      return -1;
   }
   z = b * t;
   printf_s ("out=%.15e\n",z);
}
```

Output

```
out=1.000000000000000e-002
```

See Also

Reference

Pragma Directives

float_control

Specifies floating-point behavior for a function.

```
float_control( value,setting [push] | push | pop )
```

Flags

value , *setting* **[push]**

Specifies floating-point behavior. *value* can be **precise** or **except**. For more information, see /fp (Specify Floating-Point Behavior). *setting* can either be **on** or **off**.

If *value* is **precise**, the settings for **precise** and **except** are being specified. **except** can only be set to **on** when **precise** is also set to **on**.

If the optional **push** token is added, the current setting for *value* is pushed on to the internal compiler stack.

push

Push the current **float_control** setting on to the internal compiler stack

pop

Removes the **float_control** setting from the top of the internal compiler stack and makes that the new **float_control** setting.

Remarks

You cannot turn **float_control precise** off when **except** is on. Similarly, **precise** cannot be turned off when **fenv_access** is on. To go from strict model to a fast model with the **float_control** pragma, use the following code:

```
#pragma float_control(except, off)
#pragma fenv_access(off)
#pragma float_control(precise, off)
// The following line is needed on Itanium processors
#pragma fp_contract(on)
```

To go from fast model to a strict model with the **float_control** pragma, use the following code:

```
#pragma float_control(precise, on)
#pragma fenv_access(on)
#pragma float_control(except, on)
// The following line is needed on Itanium processors.
#pragma fp_contract(off)
```

Other floating-point pragmas include:

- fenv_access
- fp_contract

Example

The following sample shows how to catch an overflow floating-point exception by using pragma **float_control**.

```
// pragma_directive_float_control.cpp
// compile with: /EHa
#include <stdio.h>
#include <float.h>

double func( ) {
   return 1.1e75;
}
```

```
#pragma float_control (except,on)

int main( ) {
   float u[1];
   unsigned int currentControl;
   errno_t err;

   err = _controlfp_s(&currentControl, ~_EM_OVERFLOW, _MCW_EM);
   if (err != 0)
      printf_s("_controlfp_s failed!\n");

   try  {
      u[0] = func();
      printf_s ("Fail");
      return(1);
   }

   catch (...)  {
      printf_s ("Pass");
      return(0);
   }
}
```

Output

```
Pass
```

See Also

Reference

Pragma Directives

fp_contract

Determines whether floating-point contraction will take place.

```
#pragma fp_contract [ON | OFF]
```

Remarks

By default, **fp_contract** is ON.

For more information on floating-point behavior, see /fp (Specify Floating-Point Behavior).

Other floating-point pragmas include:

- fenv_access
- float_control

Example

The code generated from this sample does not use the Fused Multiply Add (**fma**) instruction on Itanium processors. If you comment out `#pragma fp_contract (off)`, the generated code will use the **fma** instruction.

```
// pragma_directive_fp_contract.cpp
// compile with: /O2
#include <stdio.h>
#include <float.h>

#pragma fp_contract (off)

int main() {
   double z, b, t;

   for (int i = 0; i < 10; i++) {
      b = i * 5.5;
      t = i * 56.025;
      _set_controlfp(_PC_24, _MCW_PC);

      z = t * i + b;
      printf_s ("out=%.15e\n", z);
   }
}
```

Sample Output

```
out=0.000000000000000e+000
out=6.152500152587891e+001
out=2.351000061035156e+002
out=5.207249755859375e+002
out=9.184000244140625e+002
out=1.428125000000000e+003
out=2.049899902343750e+003
out=2.783724853515625e+003
out=3.629600097656250e+003
out=4.587524902343750e+003
```

See Also

Reference

Pragma Directives

function (C/C++)

Specifies that calls to functions specified in the pragma's argument list be generated.

```
#pragma function( function1 [, function2, ...] )
```

Remarks

If you use the **intrinsic** pragma (or /Oi) to tell the compiler to generate intrinsic functions (intrinsic functions are generated as inline code, not as function calls), you can use the **function** pragma to explicitly force a function call. Once a function pragma is seen, it takes effect at the first function definition containing a specified intrinsic function. The effect continues to the end of the source file or to the appearance of an **intrinsic** pragma specifying the same intrinsic function. The **function** pragma can be used only outside of a function — at the global level.

For lists of the functions that have intrinsic forms, see #pragma intrinsic.

Example

```
// pragma_directive_function.cpp
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// use intrinsic forms of memset and strlen
#pragma intrinsic(memset, strlen)

// Find first word break in string, and set remaining
// chars in string to specified char value.
char *set_str_after_word(char *string, char ch) {
   int i;
   int len = strlen(string);  /* NOTE: uses intrinsic for strlen */

   for(i = 0; i < len; i++) {
      if (isspace(*(string + i)))
         break;
   }

   for(; i < len; i++)
      *(string + i) = ch;

   return string;
}

// do not use strlen intrinsic
#pragma function(strlen)

// Set all chars in string to specified char value.
char *set_str(char *string, char ch) {
   // Uses intrinsic for memset, but calls strlen library function
   return (char *) memset(string, ch, strlen(string));
}

int main() {
   char *str = (char *) malloc(20 * sizeof(char));

   strcpy_s(str, sizeof("Now is the time"), "Now is the time");
   printf("str is '%s'\n", set_str_after_word(str, '*'));
   printf("str is '%s'\n", set_str(str, '!'));
}
```

Output

```
str is 'Now************'
```

```
str is '!!!!!!!!!!!!!!!'
```

See Also

Reference

Pragma Directives

hdrstop

Gives you additional control over precompilation file names and over the location at which the compilation state is saved.

```
#pragma hdrstop [( "filename" )]
```

Remarks

The *filename* is the name of the precompiled header file to use or create (depending on whether /Yu or /Yc is specified). If *filename* does not contain a path specification, the precompiled header file is assumed to be in the same directory as the source file.

If a C or C++ file contains a **hdrstop** pragma when compiled with /Yc, the compiler saves the state of the compilation up to the location of the pragma. The compiled state of any code that follows the pragma is not saved.

Use *filename* to name the precompiled header file in which the compiled state is saved. A space between **hdrstop** and *filename* is optional. The file name specified in the **hdrstop** pragma is a string and is therefore subject to the constraints of any C or C++ string. In particular, you must enclose it in quotation marks and use the escape character (backslash) to specify directory names. For example:

```
#pragma hdrstop( "c:\\projects\\include\\myinc.pch" )
```

The name of the precompiled header file is determined according to the following rules, in order of precedence:

1. The argument to the /Fp compiler option
2. The *filename* argument to #**pragma hdrstop**
3. The base name of the source file with a .PCH extension

For the /Yc and /Yu options, if neither of the two compilation options nor the **hdrstop** pragma specifies a file name, the base name of the source file is used as the base name of the precompiled header file.

You can also use preprocessing commands to perform macro replacement as follows:

```
#define INCLUDE_PATH "c:\\progra~`1\\devstsu~1\\vc\\include\\"
#define PCH_FNAME "PROG.PCH"
.
.
.
#pragma hdrstop( INCLUDE_PATH PCH_FNAME )
```

The following rules govern where the **hdrstop** pragma can be placed:

- It must appear outside any data or function declaration or definition.
- It must be specified in the source file, not within a header file.

Example

```
#include <windows.h>                // Include several files
#include "myhdr.h"

__inline Disp( char *szToDisplay )   // Define an inline function
{
    ...                             // Some code to display string
}
#pragma hdrstop
```

In this example, the **hdrstop** pragma appears after two files have been included and an inline function has been defined. This might, at first, seem to be an odd placement for the pragma. Consider, however, that using the manual precompilation options, /Yc and /Yu, with the **hdrstop** pragma makes it possible for you to precompile entire source files — even inline code. The

Microsoft compiler does not limit you to precompiling only data declarations.

See Also

Reference

Pragma Directives

include_alias

Specifies that *short_filename* is to be used as an alias for *long_filename*.

```
#pragma include_alias( "long_filename", "short_filename" )
#pragma include_alias( <long_filename>, <short_filename> )
```

Remarks

Some file systems allow longer header filenames than the 8.3 FAT file system limit. The compiler cannot simply truncate the longer names to 8.3, because the first eight characters of the longer header filenames may not be unique. Whenever the compiler encounters the *long_filename* string, it substitutes *short_filename*, and looks for the header file *short_filename* instead. This pragma must appear before the corresponding **#include** directives. For example:

```
// First eight characters of these two files not unique.
#pragma include_alias( "AppleSystemHeaderQuickdraw.h", "quickdra.h" )
#pragma include_alias( "AppleSystemHeaderFruit.h", "fruit.h" )

#pragma include_alias( "GraphicsMenu.h", "gramenu.h" )

#include "AppleSystemHeaderQuickdraw.h"
#include "AppleSystemHeaderFruit.h"
#include "GraphicsMenu.h"
```

The alias being searched for must match the specification exactly, in case as well as in spelling and in use of double quotation marks or angle brackets. The **include_alias** pragma performs simple string matching on the filenames; no other filename validation is performed. For example, given the following directives,

```
#pragma include_alias("mymath.h", "math.h")
#include "./mymath.h"
#include "sys/mymath.h"
```

no aliasing (substitution) is performed, since the header file strings do not match exactly. Also, header filenames used as arguments to the /Yu and /Yc compiler options, or the **hdrstop** pragma, are not substituted. For example, if your source file contains the following directive,

```
#include <AppleSystemHeaderStop.h>
```

the corresponding compiler option should be

```
/YcAppleSystemHeaderStop.h
```

You can use the **include_alias** pragma to map any header filename to another. For example:

```
#pragma include_alias( "api.h", "c:\version1.0\api.h" )
#pragma include_alias( <stdio.h>, <newstdio.h> )
#include "api.h"
#include <stdio.h>
```

Do not mix filenames enclosed in double quotation marks with filenames enclosed in angle brackets. For example, given the above two **#pragma include_alias** directives, the compiler performs no substitution on the following **#include** directives:

```
#include <api.h>
#include "stdio.h"
```

Furthermore, the following directive generates an error:

```
#pragma include_alias(<header.h>, "header.h")  // Error
```

Note that the filename reported in error messages, or as the value of the predefined **__FILE__** macro, is the name of the file after the substitution has been performed. For example, after the following directives,

```
#pragma include_alias( "VeryLongFileName.H", "myfile.h" )
#include "VeryLongFileName.H"
```

an error in VERYLONGFILENAME.H produces the following error message:

```
myfile.h(15) : error C2059 : syntax error
```

Also note that transitivity is not supported. Given the following directives,

```
#pragma include_alias( "one.h", "two.h" )
#pragma include_alias( "two.h", "three.h" )
#include "one.h"
```

the compiler searches for the file TWO.H rather than THREE.H.

See Also
Reference
Pragma Directives

init_seg

C++ Specific

Specifies a keyword or code section that affects the order in which startup code is executed.

```
#pragma init_seg({ compiler | lib | user | "section-name" [, func-name]} )
```

Remarks

Because initialization of global static objects can involve executing code, you must specify a keyword that defines when the objects are to be constructed. It is particularly important to use the **init_seg** pragma in dynamic-link libraries (DLLs) or libraries requiring initialization.

The options to the **init_seg** pragma are:

compiler

Reserved for Microsoft C run-time library initialization. Objects in this group are constructed first.

lib

Available for third-party class-library vendors' initializations. Objects in this group are constructed after those marked as **compiler** but before any others.

user

Available to any user. Objects in this group are constructed last.

section-name

Allows explicit specification of the initialization section. Objects in a user-specified *section-name* are not implicitly constructed; however, their addresses are placed in the section named by *section-name*.

The section name you give will contain pointers to helper functions that will construct the global objects declared in that module after the pragma.

For a list of names you should not use when creating a section, see /SECTION.

func-name

Specifies a function to be called in place of **atexit** when the program exits. This helper function also calls atexit with a pointer to the destructor for the global object. If you specify a function identifier in the pragma of the form,

```
int __cdecl myexit (void (__cdecl *pf)(void))
```

then your function will be called instead of the C run-time library's **atexit**. This allows you to build a list of the destructors that will need to be called when you are ready to destroy the objects.

If you need to defer initialization (for example, in a DLL) you may choose to specify the section name explicitly. You must then call the constructors for each static object.

There are no quotes around the identifier for the **atexit** replacement.

Your objects will still be placed in the sections defined by the other XXX_seg pragmas.

The objects that are declared in the module will not be automatically initialized by the C run-time. You will need to do that yourself.

By default, **init_seg** sections are read only. If the section name is .CRT, the compiler will silently change the attribute to read only, even if it is marked as read, write.

You cannot specify **init_seg** more than once in a translation unit.

Even if your object does not have a user-defined constructor, a constructor not explicitly defined in code, the compiler may generate one (for example to bind v-table pointers). Therefore, your code will have to call the compiler-generated constructor.

Example

```
// pragma_directive_init_seg.cpp
#include <stdio.h>
#pragma warning(disable : 4075)

typedef void (__cdecl *PF)(void);
int cxpf = 0;   // number of destructors we need to call
PF pfx[200];    // pointers to destructors.

int myexit (PF pf) {
   pfx[cxpf++] = pf;
   return 0;
}

struct A {
   A() { puts("A()"); }
   ~A() { puts("~A()"); }
};

// ctor & dtor called by CRT startup code
// because this is before the pragma init_seg
A aaaa;

// The order here is important.
// Section names must be 8 characters or less.
// The sections with the same name before the $
// are merged into one section. The order that
// they are merged is determined by sorting
// the characters after the $.
// InitSegStart and InitSegEnd are used to set
// boundaries so we can find the real functions
// that we need to call for initialization.

#pragma section(".mine$a", read)
__declspec(allocate(".mine$a")) const PF InitSegStart = (PF)1;

#pragma section(".mine$z",read)
__declspec(allocate(".mine$z")) const PF InitSegEnd = (PF)1;

// The comparison for 0 is important.
// For now, each section is 256 bytes. When they
// are merged, they are padded with zeros. You
// can't depend on the section being 256 bytes, but
// you can depend on it being padded with zeros.

void InitializeObjects () {
   const PF *x = &InitSegStart;
   for (++x ; x < &InitSegEnd ; ++x)
      if (*x) (*x)();
}

void DestroyObjects () {
   while (cxpf>0) {
      --cxpf;
      (pfx[cxpf])();
   }
}

// by default, goes into a read only section
#pragma init_seg(".mine$m", myexit)

A bbbb;
A cccc;

int main () {
   InitializeObjects();
   DestroyObjects();
}
```

Output

```
A()
A()
A()
~A()
~A()
~A()
```

See Also

Reference

Pragma Directives

inline_depth

Specifies the inline heuristic search depth, such that, no function will be inlined if it is at a depth (in the call graph) greater than *n*.

```
#pragma inline_depth( [n] )
```

Remarks

This pragma controls the inlining of functions marked inline and __inline or inlined automatically under the /Ob2 option.

n can be a value between 0 and 255, where 255 means unlimited depth in the call graph, and zero inhibits inline expansion. When *n* is not specified, the default (254) is used.

The **inline_depth** pragma controls the number of times a series of function calls can be expanded. For example, if the inline depth is four, and if A calls B and B then calls C, all three calls will be expanded inline. However, if the closest inline expansion is two, only A and B are expanded, and C remains as a function call.

To use this pragma, you must set the /Ob compiler option to 1 or 2. The depth set using this pragma takes effect at the first function call after the pragma.

The inline depth can be decreased during expansion but not increased. If the inline depth is six and during expansion the preprocessor encounters an **inline_depth** pragma with a value of eight, the depth remains six.

The **inline_depth** pragma has no effect on functions marked with **__forceinline**.

Note
Recursive functions can be substituted inline to a maximum depth of 16 calls.

See Also

Reference

Pragma Directives

inline_recursion

inline_recursion

Controls the inline expansion of direct or mutually recursive function calls.

```
#pragma inline_recursion( [{on | off}] )
```

Remarks

Use this pragma to control functions marked as inline and __inline or functions that the compiler automatically expands under the /Ob2 option. Use of this pragma requires an /Ob compiler option setting of either 1 or 2. The default state for **inline_recursion** is off. This pragma takes effect at the first function call after the pragma is seen and does not affect the definition of the function.

The **inline_recursion** pragma controls how recursive functions are expanded. If **inline_recursion** is off, and if an inline function calls itself (either directly or indirectly), the function is expanded only once. If **inline_recursion** is on, the function is expanded multiple times until it reaches the value set by inline_depth, the default value of 8, or a capacity limit.

See Also

Reference

Pragma Directives

intrinsic

Specifies that calls to functions specified in the pragma's argument list are intrinsic.

```
#pragma intrinsic( function1 [, function2, ...] )
```

Remarks

The **intrinsic** pragma tells the compiler that a function has known behavior. The compiler may call the function and not replace the function call with inline instructions, if it will result in better performance.

The library functions with intrinsic forms are listed below. Once an **intrinsic** pragma is seen, it takes effect at the first function definition containing a specified intrinsic function. The effect continues to the end of the source file or to the appearance of a **function** pragma specifying the same intrinsic function. The **intrinsic** pragma can be used only outside of a function definition — at the global level.

The following functions have intrinsic forms and the intrinsic forms are used when you specify /Oi:

_disable	_outp	fabs	strcmp
_enable	_outpw	labs	strcpy
_inp	_rotl	memcmp	strlen
_inpw	_rotr	memcpy	
_lrotl	_strset	memset	
_lrotr	abs	strcat	

Programs that use intrinsic functions are faster because they do not have the overhead of function calls but may be larger due to the additional code generated.

x86 Specific

The _disable and _enable intrinsics generate kernel-mode instructions to disable/enable interrupts and could be useful in kernel-mode drivers.

Example

Compile the following code from the command line with "cl -c -FAs sample.c" and look at sample.asm to see that they turn into x86 instructions CLI and STI:

```
// pragma_directive_intrinsic.cpp
// processor: x86
#include <dos.h>   // definitions for _disable, _enable
#pragma intrinsic(_disable)
#pragma intrinsic(_enable)
void f1(void) {
   _disable();
   // do some work here that should not be interrupted
   _enable();
}
int main() {
}
```

End x86 Specific

The floating-point functions listed below do not have true intrinsic forms. Instead they have versions that pass arguments directly to the floating-point chip rather than pushing them onto the program stack:

acos	cosh	pow	tanh
asin	fmod	sinh	

The floating-point functions listed below have true intrinsic forms when you specify /Oi, /Og, and /fp:fast (or any option that includes /Og: /Ox, /O1, and /O2):

atan	exp	log10	sqrt
atan2	log	sin	tan
cos			

You can use /fp:strict or /Za to override generation of true intrinsic floating-point options. In this case, the functions are generated as library routines that pass arguments directly to the floating-point chip instead of pushing them onto the program stack.

See # pragma function for information and an example on how to enable/disable intrinsics for a block of source text.

See Also

Reference

Pragma Directives

Compiler Intrinsics

make_public

Indicates that a native type should have public assembly accessibility.

```
#pragma make_public(type)
```

Parameters

type is the name of the type you want to have public assembly accessibility.

Remarks

make_public is useful for when the native type you want to reference is from a .h file that you cannot change. If you want to use the native type in the signature of a public function in a type with public assembly visibility, the native type must also have public assembly accessibility or the compiler will issue a warning.

make_public must be specified at global scope and is only in effect from the point at which it is declared through to the end of the source code file.

The native type may be implicitly or explicitly private; see Type Visibility for more information.

Example

The following sample is the contents of a .h file that contains the definitions for two native structs.

```
// make_public_pragma.h
struct Native_Struct_1 { int i; };
struct Native_Struct_2 { int i; };
```

The following code sample consumes the header file and shows that unless you explicitly mark the native structs as public, using **make_public**, the compiler will generate a warning when you attempt to use the native structs in the signature of public function in a public managed type.

```
// make_public_pragma.cpp
// compile with: /c /clr /W1
#pragma warning (default : 4692)
#include "make_public_pragma.h"
#pragma make_public(Native_Struct_1)

public ref struct A {
   void Test(Native_Struct_1 u) {u.i = 0;}   // OK
   void Test(Native_Struct_2 u) {u.i = 0;}   // C4692
};
```

See Also

Reference

Pragma Directives

managed, unmanaged

Enable function-level control for compiling functions as managed or unmanaged.

```
#pragma managed
#pragma unmanaged
#pragma managed([push,] on | off)
#pragma managed(pop)
```

Remarks

The /clr compiler option provides module-level control for compiling functions either as managed or unmanaged.

An unmanaged function will be compiled for the native platform, and execution of that portion of the program will be passed to the native platform by the common language runtime.

Functions are compiled as managed by default when **/clr** is used.

Use the following guidelines when applying these pragmas:

- Add the pragma preceding a function but not within a function body.
- Add the pragma after **#include** statements (do not use these pragmas before **#include** statements).

The compiler ignores the **managed** and **unmanaged** pragmas if **/clr** is not used in the compilation.

When a template function is instantiated, the pragma state at the time of definition for the template determines if it is managed or unmanaged.

For more information, see Initialization of Mixed Assemblies.

Example

```
// pragma_directives_managed_unmanaged.cpp
// compile with: /clr
#include <stdio.h>

// func1 is managed
void func1() {
   System::Console::WriteLine("In managed function.");
}

// #pragma unmanaged
// push managed state on to stack and set unmanaged state
#pragma managed(push, off)

// func2 is unmanaged
void func2() {
   printf("In unmanaged function.\n");
}

// #pragma managed
#pragma managed(pop)

// main is managed
int main() {
   func1();
   func2();
}
```

Output

```
In managed function.
In unmanaged function.
```

See Also

Reference

Pragma Directives

message

Sends a string literal to the standard output without terminating the compilation.

```
#pragma message( messagestring )
```

Remarks

A typical use of the **message** pragma is to display informational messages at compile time.

The *messagestring* parameter can be a macro that expands to a string literal, and you can concatenate such macros with string literals in any combination.

If you use a predefined macro in the **message** pragma, the macro should return a string, else you will have to convert the output of the macro to a string.

The following code fragment uses the **message** pragma to display messages during compilation:

```
// pragma_directives_message1.cpp
// compile with: /LD
#if _M_IX86 >= 500
#pragma message("_M_IX86 >= 500")
#endif

#pragma message("")

#pragma message( "Compiling " __FILE__ )
#pragma message( "Last modified on " __TIMESTAMP__ )

#pragma message("")

// with line number
#define STRING2(x) #x
#define STRING(x) STRING2(x)

#pragma message (__FILE__ "[" STRING(__LINE__) "]: test")

#pragma message("")
```

See Also

Reference

Pragma Directives

omp

Takes one or more OpenMP directives, along with any optional directive clauses.

```
#pragma omp directive
```

Remarks

See OpenMP Directives for more information.

See Also

Reference

Pragma Directives

once

Specifies that the file will be included (opened) only once by the compiler when compiling a source code file.

```
#pragma once
```

Remarks

This can reduce build times as the compiler will not open and read the file after the first #include of the module.

For example:

```
// header.h
#pragma once
....
```

See Also

Reference

Pragma Directives

optimize

Specifies optimizations to be performed on a function-by-function basis.

```
#pragma optimize( "[optimization-list]", {on | off} )
```

Remarks

The **optimize** pragma must appear outside a function and takes effect at the first function defined after the pragma is seen. The **on** and **off** arguments turn options specified in the *optimization-list* on or off.

The *optimization-list* can be zero or more of the parameters shown in the following table.

Parameters of the optimize Pragma

Parameter(s)	Type of optimization
g	Enable global optimizations.
p	Improve floating-point consistency.
s or **t**	Specify short or fast sequences of machine code.
y	Generate frame pointers on the program stack.

These are the same letters used with the /O compiler options. For example:

```
#pragma optimize( "tp", on )
```

Using the **optimize** pragma with the empty string (**""**) is a special form of the directive:

When you use the **off** parameter, it turns the optimizations, listed in the table above, off.

When you use the **on** parameter, it resets the optimizations to those that you specified with the /O compiler option.

```
#pragma optimize( "", off )
.
.
.
#pragma optimize( "", on )
```

See Also

Reference

Pragma Directives

pack

Specifies packing alignment for structure, union, and class members.

```
#pragma pack( [ show ] | [ push | pop ] [, identifier ] , n  )
```

Remarks

pack gives control at the data-declaration level. This differs from compiler option /Zp, which only provides module-level control. **pack** takes effect at the first **struct**, **union**, or **class** declaration after the pragma is seen; **pack** has no effect on definitions. Calling **pack** with no arguments sets *n* to its default value. This is equivalent to compiler option /Zp8.

Note that if you change the alignment of a structure, the structure will not use as much space in memory, but you may see a decrease in performance or even get a hardware-generated exception for unaligned access. It is possible to modify this exception behavior with SetErrorMode.

show (optional)

Displays the current byte value for packing alignment. The value is displayed by means of a warning message.

push (optional)

Puts a specified packing alignment value, *n*, on the internal compiler stack, and sets the current packing alignment value to *n*. If *n* is not specified, the current packing alignment value is pushed.

pop (optional)

Removes the record from the top of the internal compiler stack. If *n* is not specified with **pop**, then the packing value associated with the resulting record on the top of the stack is the new packing alignment value. If *n* is specified, for example, `#pragma pack(pop, 16)`, *n* becomes the new packing alignment value. If you pop with *identifier*, for example, `#pragma pack(pop, r1)`, then all records on the stack are popped until the record with *identifier* is found, and that record is popped and the packing value associated with the resulting record on the top of is the stack the new packing alignment value. If you pop with an *identifier* that is not found in any record on the stack, then the **pop** is ignored.

identifier (optional)

When used with **push**, assigns a name to the record on the internal compiler stack. When used with **pop**, pops records off the internal stack until *identifier* is removed; if *identifier* is not found on the internal stack, nothing is popped.

n(optional)

Specifies the value, in bytes, to be used for packing. The default value for *n* is 8. Valid values are 1, 2, 4, 8, and 16. The alignment of a member will be on a boundary that is either a multiple of *n* or a multiple of the size of the member, whichever is smaller.

n can be used with **push** or **pop** for setting a particular stack value, or alone for setting the current value used by the compiler.

#pragma pack(pop, *identifier***,** *n***)** is undefined.

For more information on modifying alignment, see:

- __alignof
- align
- __unaligned
- Structure Alignment Examples (x64 specific)

Example

The following sample shows how to the **pack** pragma to change the alignment of a structure.

```
// pragma_directives_pack.cpp
#include <stddef.h>
#include <stdio.h>
```

```
struct S {
   int i;   // size 4
   short j;   // size 2
   double k;   // size 8
};

#pragma pack(2)
struct T {
   int i;
   short j;
   double k;
};

int main() {
   printf("%d ", offsetof(S, i));
   printf("%d ", offsetof(S, j));
   printf("%d\n", offsetof(S, k));

   T tt;
   printf("%d ", offsetof(T, i));
   printf("%d ", offsetof(T, j));
   printf("%d\n", offsetof(T, k));
}
```

```
0 4 8
0 4 6
```

The following sample shows how to use the push, pop, and show syntax.

```
// pragma_directives_pack_2.cpp
// compile with: /W1 /c
#pragma pack()   // n defaults to 8; equivalent to /Zp8
#pragma pack(show)   // C4810
#pragma pack(4)   // n = 4
#pragma pack(show)   // C4810
#pragma pack(push, r1, 16)   // n = 16, pushed to stack
#pragma pack(show)   // C4810
#pragma pack(pop, r1, 2)   // n = 2 , stack popped
#pragma pack(show)   // C4810
```

See Also

Reference

Pragma Directives

pointers_to_members

C++ Specific

Specifies whether a pointer to a class member can be declared before its associated class definition and is used to control the pointer size and the code required to interpret the pointer.

```
#pragma pointers_to_members( pointer-declaration, [most-general-representation] )
```

Remarks

You can place a **pointers_to_members** pragma in your source file as an alternative to using the /vmx compiler options or the inheritance keywords.

The *pointer-declaration* argument specifies whether you have declared a pointer to a member before or after the associated function definition. The *pointer-declaration* argument is one of the following two symbols:

Argument	Comments
full_generality	Generates safe, sometimes nonoptimal code. You use **full_generality** if any pointer to a member is declared before the associated class definition. This argument always uses the pointer representation specified by the *most-general-representation* argument. Equivalent to /vmg.
best_case	Generates safe, optimal code using best-case representation for all pointers to members. Requires defining the class before declaring a pointer to a member of the class. The default is **best_case**.

The *most-general-representation* argument specifies the smallest pointer representation that the compiler can safely use to reference any pointer to a member of a class in a translation unit. The argument can be one of the following:

Argument	Comments
single_inheritance	The most general representation is single-inheritance, pointer to a member function. Causes an error if the inheritance model of a class definition for which a pointer to a member is declared is ever either multiple or virtual.
multiple_inheritance	The most general representation is multiple-inheritance, pointer to a member function. Causes an error if the inheritance model of a class definition for which a pointer to a member is declared is virtual.
virtual_inheritance	The most general representation is virtual-inheritance, pointer to a member function. Never causes an error. This is the default argument when **#pragma pointers_to_members(full_generality)** is used.

Example

```
//   Specify single-inheritance only
#pragma pointers_to_members( full_generality, single_inheritance )
```

END C++ Specific

See Also

Reference

Pragma Directives

pop_macro

Sets the value of the *macro_name* macro to the value on the top of the stack for this macro.

```
#pragma pop_macro("macro_name")
```

Remarks

You must first issue a push_macro for *macro_name* before you can do a **pop_macro**.

Example

```
// pragma_directives_pop_macro.cpp
// compile with: /W1
#include <stdio.h>
#define X 1
#define Y 2

int main() {
   printf("%d",X);
   printf("\n%d",Y);
   #define Y 3   // C4005
   #pragma push_macro("Y")
   #pragma push_macro("X")
   printf("\n%d",X);
   #define X 2   // C4005
   printf("\n%d",X);
   #pragma pop_macro("X")
   printf("\n%d",X);
   #pragma pop_macro("Y")
   printf("\n%d",Y);
}
```

Output

```
1
2
1
2
1
3
```

See Also

Reference

Pragma Directives

push_macro

Saves the value of the *macro_name* macro on the top of the stack for this macro.

```
#pragma push_macro("macro_name")
```

Remarks

You can retrieve the value for *macro_name* with **pop_macro**.

See pop_macro for a sample.

See Also

Reference

Pragma Directives

region, endregion

#pragma region lets you specify a block of code that you can expand or collapse when using the outlining feature of the Visual Studio Code Editor.

```
#pragma region name
#pragma endregion comment
```

Parameters

comment(optional)

A comment that will display in the code editor.

name(optional)

The name of the region. This name will display in the code editor.

Remarks

#pragma endregion marks the end of a **#pragma region** block.

A **#region** block must be terminated with **#pragma endregion**.

Example

```
// pragma_directives_region.cpp
#pragma region Region_1
void Test() {}
void Test2() {}
void Test3() {}
#pragma endregion Region_1

int main() {}
```

See Also

Reference

Pragma Directives

runtime_checks

Disables or restores the /RTC settings.

```
#pragma runtime_checks( "[runtime_checks]", {restore | off} )
```

Remarks

You cannot enable a run-time check that was not enabled with a compiler option. For example, if you do not specify /RTCs, specifying `#pragma runtime_checks( "s", restore)` will not enable stack frame verification.

The **runtime_checks** pragma must appear outside a function and takes effect at the first function defined after the pragma is seen. The **restore** and **off** arguments turn options specified in the *runtime_checks* on or off.

The *runtime_checks* can be zero or more of the parameters shown in the following table.

Parameters of the runtime_checks Pragma

Parameter(s)	Type of run-time check
s	Enables stack (frame) verification.
c	Reports when a value is assigned to a smaller data type that results in a data loss.
u	Reports when a variable is used before it is defined.

These are the same letters used with the /RTC compiler option. For example:

```
#pragma runtime_checks( "sc", restore )
```

Using the **runtime_checks** pragma with the empty string (**""**) is a special form of the directive:

- When you use the **off** parameter, it turns the run-time error checks, listed in the table above, off.
- When you use the **restore** parameter, it resets the run-time error checks to those that you specified with the /RTC compiler option.

```
#pragma runtime_checks( "", off )
.
.
.
#pragma runtime_checks( "", restore )
```

See Also

Tasks

RTC Sample: Run-Time Error Checks

Reference

Pragma Directives

section

Creates a section in an .obj file.

```
#pragma section( "section-name" [, attributes] )
```

Remarks

Once a section is defined, it remains valid for the remainder of the compilation. However, you must use __declspec(allocate) or nothing will be placed in the section.

section-name is a required parameter that will be the name of the section. The name must not conflict with any standard section names. See /SECTION for a list of names you should not use when creating a section.

attributes is an optional parameter consisting of one or more comma-separated attributes that you want to assign to the section. Possible *attributes* are:

read

Allows read operations on data.

write

Allows write operations on data.

execute

Allows code to be executed.

shared

Shares the section among all processes that load the image.

nopage

Marks the section as not pageable; useful for Win32 device drivers.

nocache

Marks the section as not cacheable; useful for Win32 device drivers.

discard

Marks the section as discardable; useful for Win32 device drivers.

remove

Marks the section as not memory-resident; virtual device drivers (V*x*D) only.

If you do not specify attributes, the section will have read and write attributes.

Example

In the following example, the first instruction identifies the section and its attributes. The integer `j` is not put into `mysec` because it was not declared with `__declspec(allocate)`; `j` goes into the data section. The integer `i` does go into `mysec` as a result of its `__declspec(allocate)` storage-class attribute.

```
// pragma_section.cpp
#pragma section("mysec",read,write)
int j = 0;

__declspec(allocate("mysec"))
int i = 0;

int main(){}
```

See Also

Reference

setlocale

Defines the locale (Country/Region and language) to be used when translating wide-character constants and string literals.

```
#pragma setlocale( "[locale-string]" )
```

Remarks

Because the algorithm for converting multibyte characters to wide characters may vary by locale or the compilation may take place in a different locale from where an executable file will be run, this pragma provides a way to specify the target locale at compile time. This guarantees that the wide-character strings will be stored in the correct format.

The default *locale-string* is "".

The "C" locale maps each character in the string to its value as a **wchar_t** (unsigned short). Other values that are valid for **setlocale** are those entries that are found in the Language Strings list. For example, you could issue:

```
#pragma setlocale("dutch")
```

The ability to issue a language string depends on the code page and language ID support on your computer.

See Also

Reference

Pragma Directives

strict_gs_check

This pragma provides enhanced security checking.

```
#pragma strict_gs_check([push,] on )
#pragma strict_gs_check([push,] off )
#pragma strict_gs_check(pop)
```

Remarks

Instructs the compiler to insert a random cookie in the function stack to help detect some categories of stack-based buffer overrun. By default, the /GS (Buffer Security Check) compiler option does not insert a cookie for all functions. For more information, see /GS (Buffer Security Check).

You must compile with /GS (Buffer Security Check) to enable strict_gs_check.

Use this pragma sparingly, and only in code modules that are exposed to potentially harmful data. This pragma is very aggressive, and is applied to functions that might not need this defense, which can decrease the performance of the resulting application.

Even if you use this pragma, you should strive to write secure code. That is, make sure that your code has no buffer overruns. strict_gs_check might protect your application from buffer overruns that do remain in your code.

Example

In the following code a buffer overrun occurs when we copy an array to a local array. When you compile this code with /GS, no cookie is inserted in the stack, because the array data type is an unsigned integer. Adding the strict_gs_check pragma forces the stack cookie into the function stack.

```
// pragma_strict_gs_check.cpp
#pragma strict_gs_check(on)

unsigned int * ReverseArray(__in_ecount(cData) unsigned int *pdwData,
                            size_t cData)
{
   // *** This buffer is subject to being overrun!! ***
   unsigned int dwReversed[20];
   // Reverse the array into a temporary buffer
   for (size_t j=0, i = cData; i ; --i, ++j)
      // *** Possible buffer overrun!! ***
      dwReversed[j] = pdwData[i];

   // Copy temporary buffer back into input/output buffer
   for (size_t i = 0; i < cData ; ++i)
       pdwData[i] = dwReversed[i];

   return pdwData;
}
```

See Also

Reference

Pragma Directives

/GS (Buffer Security Check)

vtordisp

C++ Specific

Enables the addition of the hidden vtordisp construction/destruction displacement member.

```
#pragma vtordisp({on | off} )
```

Remarks

Note
vtordisp is now deprecated and will be removed in a future release of Visual C++.

The **vtordisp** pragma is applicable only to code that uses virtual bases. If a derived class overrides a virtual function that it inherits from a virtual base class, and if a constructor or destructor for the derived class calls that function using a pointer to the virtual base class, the compiler may introduce additional hidden "vtordisp" fields into classes with virtual bases.

The **vtordisp** pragma affects the layout of classes that follow it. The /vd0 and /vd1 options specify the same behavior for complete modules. Specifying **off** suppresses the hidden **vtordisp** members. Specifying **on**, the default, enables them where they are necessary. Turn off **vtordisp** only if there is no possibility that the class's constructors and destructors call virtual functions on the object pointed to by the **this** pointer.

```
#pragma vtordisp( off )
class GetReal : virtual public { ... };
#pragma vtordisp( on )
```

END C++ Specific

See Also

Reference

Pragma Directives

warning

Allows selective modification of the behavior of compiler warning messages.

```
#pragma warning( warning-specifier : warning-number-list [; warning-specifier : warning-num
ber-list...] )
#pragma warning( push[ ,n ] )
#pragma warning( pop )
```

Remarks

The *warning-specifier* can be one of the following.

Warning-specifier	**Meaning**
1, **2**, **3**, **4**	Apply the given level to the specified warning(s). This also has the effect of turning a specified warning on that is off by default.
default	Reset warning behavior to its default value. This also has the effect of turning a specified warning on that is off by default. The warning will be generated at its default, documented, level. See Compiler Warnings That Are Off by Default for more information.
disable	Do not issue the specified warning message(s).
error	Report the specified warnings as errors.
once	Display the specified message(s) only once.
suppress	Pushes the current state of the pragma on the stack, disables the specified warning for the next line, and then pops the warning stack, resetting the pragma state. You can only specify one warning for each **suppress** specifier, but multiple **warning** pragmas can operate on one line of code. **suppress** is only supported for C6000 warnings (code analysis warnings), which are enabled with the /analyze (Enterprise Code Analysis) compiler option.

The *warning-number-list* can contain any warning numbers. Multiple options can be specified in the same pragma directive as follows:

```
#pragma warning( disable : 4507 34; once : 4385; error : 164 )
```

This is functionally equivalent to:

```
// Disable warning messages 4507 and 4034.
#pragma warning( disable : 4507 34 )

// Issue warning 4385 only once.
#pragma warning( once : 4385 )

// Report warning 4164 as an error.
#pragma warning( error : 164 )
```

The compiler will add 4000 to any warning number that is between 0 and 999.

The compiler only supports up to 56 `#pragma warning` statements in a compiland.

For warning numbers in the range of 4700-4999, those associated with code generation, the state of the warning in effect when the compiler encounters the open curly brace of a function will be in effect for the rest of the function. Using the **warning** pragma inside the function to change the state of a warning greater than 4699 will only take effect after the end of the function. The following example shows the correct placement of **warning** pragmas to disable, and then restore, a code-generation warning message:

```
// pragma_warning.cpp
// compile with: /W1
#pragma warning(disable:4700)
void Test() {
   int x;
   int y = x;   // no C4700 here
   #pragma warning(default:4700)   // C4700 enabled after Test ends
}

int main() {
   int x;
   int y = x;   // C4700
}
```

Note that within a function body, the last setting of the **warning** pragma will be in effect for the entire function.

The **warning** pragma also supports the following syntax:

#pragma warning(push [**,***n*] **)**

#pragma warning(pop)

Where *n* represents a warning level (1 through 4).

The pragma **warning(push)** stores the current warning state for all warnings. The pragma **warning(push,** *n***)** stores the current state for all warnings and sets the global warning level to *n*.

The pragma **warning(pop)** pops the last warning state pushed onto the stack. Any changes made to the warning state between **push** and **pop** are undone. Consider this example:

```
#pragma warning( push )
#pragma warning( disable : 4705 )
#pragma warning( disable : 4706 )
#pragma warning( disable : 4707 )
// Some code
#pragma warning( pop )
```

At the end of this code, **pop** restores the state of all warnings (including 4705, 4706, and 4707) to what it was at the beginning of the code.

When you write header files, you can use **push** and **pop** to ensure that changes to warning states made by the user do not prevent your headers from compiling properly. Use **push** at the beginning of the header and **pop** at the end. Suppose, for example, you have a header that does not compile cleanly at warning level 4. The following code changes the warning level to 3 then restores the original warning level at the end of the header:

```
#pragma warning( push, 3 )
// Declarations/ definitions
#pragma warning( pop )
```

See /FI and /w for compiler options that help you suppress warnings.

See Also
Reference
Pragma Directives

Compiler Warnings That Are Off by Default

The compiler includes warnings that are off by default. If a warning is off by default, most users would not want to see it.

A warning that is off by default can be enabled in one of several ways:

#pragma warning(default : *warning_number***)**

The warning is enabled at its default level. Documentation for the warning contains the default level of the warning.

#pragma warning(*warning_level* **:** *warning_number***)**

The warning is enabled at the specified level (*warning_level*).

/Wall

/Wall enables all warnings that are off by default.

The following warnings are off by default:

- C4061
- C4062
- C4191
- C4242
- C4254
- C4255
- C4263
- C4264
- C4265
- C4266
- C4287
- C4289
- C4296
- C4302
- C4365
- C4431
- C4339
- C4342
- C4347
- C4350
- C4412
- C4514
- C4545
- C4546
- C4547
- C4548
- C4549

- C4536
- C4555
- C4557
- C4571
- C4619
- C4623
- C4625
- C4626
- C4628
- C4640
- C4641
- C4668
- C4682
- C4686
- C4692
- C4710
- C4711
- C4738
- C4820
- C4826
- C4836
- C4905
- C4906
- C4917
- C4928
- C4931
- C4946
- C4962

See Also

Reference

warning

Grammar Summary (C/C++)

This section describes the formal grammar of the preprocessor. It covers the syntax of preprocessing directives and operators discussed in The Preprocessor and in Pragma Directives.

The following topics are included:

- Definitions
- Conventions
- Preprocessor Grammar

See Also

Other Resources

C/C++ Preprocessor Reference

Definitions for the Grammar Summary

Terminals are endpoints in a syntax definition. No other resolution is possible. Terminals include the set of reserved words and user-defined identifiers.

Nonterminals are placeholders in the syntax. Most are defined elsewhere in this syntax summary. Definitions can be recursive. The following nonterminals are defined in the Grammar Summary of the *C++ Language Reference*:

constant, *constant-expression*, *identifier*, *keyword*, *operator*, *punctuator*

An optional component is indicated by the subscripted opt. For example, the following indicates an optional expression enclosed in curly braces:

{ *expression*opt **}**

See Also
Reference
Grammar Summary (C/C++)

Conventions

The conventions use different font attributes for different components of the syntax. The symbols and fonts are as follows:

Attribute	Description
nonterminal	Italic type indicates nonterminals.
#include	Terminals in bold type are literal reserved words and symbols that must be entered as shown. Characters in this context are always case sensitive.
opt	Nonterminals followed by opt are always optional.
default typeface	Characters in the set described or listed in this typeface can be used as terminals in statements.

A colon (:) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines.

See Also

Reference

Grammar Summary (C/C++)

Preprocessor Grammar

#define *identifier token-string*$_{opt}$

define *identifier*[**(** *identifier*$_{opt}$**,** ... **,** *identifier*$_{opt}$ **)**] *token-string*$_{opt}$

defined(*identifier* **)**

defined *identifier*

#include "*path-spec***"**

#include <*path-spec***>**

#line *digit-sequence* **"** *filename* **"** $_{opt}$

undef *identifier*

#error *token-string*

#pragma *token-string*

conditional :

if-part elif-parts $_{opt}$ *else-part*$_{opt}$ *endif-line*

if-part :

if-linetext

if-line :

#if *constant-expression*

#ifdef *identifier*

#ifndef *identifier*

elif-parts :

elif-line text

elif-parts elif-line text

elif-line :

#elif *constant-expression*

else-part :

else-linetext

else-line :

#else

endif-line :

#endif

digit-sequence :

digit

digit-sequence digit

digit : one of

0 1 2 3 4 5 6 7 8 9

token-string :

String of tokens

token :

keyword

identifier

constant

operator

punctuator

filename :

Legal operating system filename

path-spec :

Legal file path

text :

Any sequence of text

Note
The following nonterminals are expanded in Appendix A, Grammar Summary, of the *C++ Language Reference*: *constant*, *constant-expression*, *identifier*, *keyword*, *operator*, and *punctuator*.

See Also

Reference

Grammar Summary (C/C++)

Compiler Intrinsics

Most functions are contained in libraries, but some functions are built in (that is, intrinsic) to the compiler. These are referred to as intrinsic functions or intrinsics.

Remarks

If a function is an intrinsic, the code for that function is usually inserted inline, avoiding the overhead of a function call and allowing highly efficient machine instructions to be emitted for that function. An intrinsic is often faster than the equivalent inline assembly, because the optimizer has a built-in knowledge of how many intrinsics behave, so some optimizations can be available that are not available when inline assembly is used. Also, the optimizer can expand the intrinsic differently, align buffers differently, or make other adjustments depending on the context and arguments of the call.

The use of intrinsics affects the portability of code, because intrinsics that are available in Visual C++ might not be available if the code is compiled with other compilers and some intrinsics that might be available for some target architectures are not available for all architectures. However, intrinsics are usually more portable than inline assembly. The intrinsics are required on 64-bit architectures where inline assembly is not supported.

Some intrinsics, such as **__assume** and **__ReadWriteBarrier**, provide information to the compiler, which affects the behavior of the optimizer.

Some intrinsics are available only as intrinsics, and some are available both in function and intrinsic implementations. You can instruct the compiler to use the intrinsic implementation in one of two ways, depending on whether you want to enable only specific functions or you want to enable all intrinsics. The first way is to use `#pragma intrinsic(`*intrinsic-function-name-list*`)`. The pragma can be used to specify a single intrinsic or multiple intrinsics separated by commas. The second is to use the /Oi (Generate Intrinsic Functions) compiler option, which makes all intrinsics on a given platform available. Under **/Oi**, use `#pragma function(`*intrinsic-function-name-list*`)` to force a function call to be used instead of an intrinsic. If the documentation for a specific intrinsic notes that the routine is only available as an intrinsic, then the intrinsic implementation is used regardless of whether **/Oi** or `#pragma intrinsic` is specified. In all cases, **/Oi** or `#pragma intrinsic` allows, but does not force, the optimizer to use the intrinsic. The optimizer can still call the function.

Some standard C/C++ library functions are available in intrinsic implementations on some architectures. When calling a CRT function, the intrinsic implementation is used if **/Oi** is specified on the command line.

A header file, Intrin.h, is available that declares prototypes for the intrinsic functions. Additionally, certain Windows headers declare functions that map onto a compiler intrinsic.

The following sections describe intrinsics that are available on various architectures:

- Intrinsics Available on All Architectures
- x86 Intrinsics
- Itanium Intrinsics
- x64 Intrinsics

In addition, intrinsics are available for targeting multimedia (MMX) instruction sets as well as SSE, SSE2, and 3DNow! features of some chips. Many of these intrinsics are also available on the 64-bit architectures. For more information, see the following topics:

- MMX, SSE, and SSE2 Intrinsics
- AMD 3DNow! Technology Overview and Intrinsics

See Also

Reference

C++ Keywords

Other Resources

Run-Time Library Reference

Intrinsics Available on All Architectures

The following intrinsics are available with the x86, Intel 64-bit, and AMD 64-bit architectures:

__assume	_BitScanForward	_BitScanReverse
_bittest	_bittestandcomplement	_bittestandreset
_bittestandset		
__debugbreak	_disable	_enable
_InterlockedAnd	_InterlockedCompareExchange	_InterlockedCompareExchange16
_InterlockedCompareExchange64	_InterlockedDecrement	_InterlockedDecrement16
_InterlockedExchange	_InterlockedExchangeAdd	_InterlockedIncrement
_InterlockedIncrement16	_InterlockedOr	_InterlockedXor
__noop	_ReadBarrier	_ReadWriteBarrier
_ReturnAddress	_rotl8	_rotl16
_rotr8	_rotr16	_WriteBarrier

The following CRT functions have intrinsic forms on all architectures:

abs	_abs64	_alloca
_byteswap_uint64	_byteswap_ulong	_byteswap_ushort
labs	_lrotl	_lrotr
memcmp	memcpy	memset
_rotl	_rotl64	_rotr
_rotr64	_setjmp	strcat
strcmp	strcpy	strlen
strset	wcscat	wcscmp
wcscpy	wcslen	_wcsset

The following math library functions have intrinsic forms on all architectures:

acos	acosf	acosl
asin	asinf	asinl

atan	atanf	atanl
atan2	atan2f	atan2l
ceil	ceilf	ceill
cosh	coshf	coshl
cos	cosf	cosl
exp	expf	expl
floor	floorf	floorl
fmod	fmodf	fmodl
log	logf	logl
log10	log10f	log10l
pow	powf	powl
sin	sinf	sinl
sinh	sinhf	sinhl
sqrt	sqrtf	sqrtl
tan	tanf	tanl
tanh	tanhf	tanhl

See Also

Reference

x86 Intrinsics
Itanium Intrinsics
x64 Intrinsics

x86 Intrinsics

The Visual C++ compiler makes the following intrinsics available on the x86 architecture.

In addition, the Visual C++ compiler supports intrinsics for MMX, Streaming SIMD Extensions (SSE), and Streaming SIMD Extensions 2 (SSE2). For more information, see MMX, SSE, and SSE2 Intrinsics. Also, many functions in the CRT are implemented as intrinsics on the x86 platform.

Index

A - H I - Q R - Z Language Extensions

A - H

__addfsbyte

__addfsdword

__addfsword

__assume

_AddressOfReturnAddress

_BitScanForward

_BitScanReverse

_bittest

_bittestandcomplement

_bittestandreset

_bittestandset

__cpuid

__debugbreak

_disable

__emul

__emulu

_enable

__getcallerseflags

__halt

[Go back to top]

I - Q

__incfsbyte

__incfsdword

__incfsword

__int2c

_InterlockedAddLargeStatistic

_InterlockedAnd

_interlockedbittestandreset

_interlockedbittestandset

_InterlockedCompareExchange16

_InterlockedCompareExchange64

_InterlockedCompareExchange

_InterlockedDecrement16

_InterlockedDecrement

_InterlockedExchangeAdd

_InterlockedExchange

_InterlockedIncrement16

_InterlockedIncrement

_InterlockedOr

_InterlockedXor

__lidt

__ll_lshift

__ll_rshift

__noop

__nop

[Go back to top]

R - Z

__readcr8

__readdr

__readeflags

__readfsbyte

__readfsdword

__readfsqword

__readfsword

_ReadBarrier

_ReadWriteBarrier

_ReturnAddress

_rotl16

_rotl8

_rotr16

_rotr8

__sidt

__svm_clgi

__svm_invlpga

__svm_skinit

__svm_stgi

__svm_vmload

__svm_vmrun

__svm_vmsave

__ud2

__ull_rshift

__vmx_off

__vmx_vmptrst

__writecr8

__writedr

__writeeflags

__writefsbyte

__writefsdword

__writefsqword

__writefsword

_WriteBarrier

[Go back to top]

Language Extensions

The following functions, with or without the preceding underscore, are available only when you specify the /Ze compiler option to enable Microsoft extensions:

_inp

inp

_inpd

inpd

_inpw

inpw

_outp

outp

_outpd

outpd

_outpw

outpw

[Go back to top]

See Also

Reference

Compiler Intrinsics

Other Resources

C++ Language Reference

Itanium Intrinsics

The Visual C++ compiler includes the following intrinsics that are available on the Intel Itanium Processor Family (IPF) architecture:

_AcquireSpinLock	__assume	_BitScanForward
_BitScanForward64	_BitScanReverse	_BitScanReverse64
_bittest	_bittest64	_bittestandcomplement
_bittestandcomplement64	_bittestandreset	
_bittestandreset64	_bittestandset	_bittestandset64
__break	__debugbreak	_disable
__dsrlz	_enable	__fc
__fci	__fclrf	__fsetc
__fwb	__getCFS	__getPSP
__getReg	_InterlockedAdd	_InterlockedAdd_acq
_InterlockedAdd_rel	_InterlockedAdd64	_InterlockedAdd64_acq
_InterlockedAdd64_rel	_InterlockedAnd	_InterlockedAnd64
_InterlockedCompareExchange	_InterlockedCompareExchange_acq	_InterlockedCompareExchange_rel
_InterlockedCompareExchange16	_InterlockedCompareExchange16_acq	_InterlockedCompareExchange16_rel
_InterlockedCompareExchange64	_InterlockedCompareExchange64_acq	_InterlockedCompareExchange64_rel
_InterlockedCompareExchangePointer	_InterlockedCompareExchangePointer_acq	_InterlockedCompareExchangePointer_rel
_InterlockedCompare64Exchange128	_InterlockedCompare64Exchange128_acq	_InterlockedCompare64Exchange128_rel
_InterlockedDecrement	_InterlockedDecrement_acq	_InterlockedDecrement_rel
_InterlockedDecrement16	_InterlockedDecrement16_acq	_InterlockedDecrement16_rel
_InterlockedDecrement64	_InterlockedDecrement64_acq	_InterlockedDecrement64_rel
_InterlockedExchange	_InterlockedExchange_acq	_InterlockedExchange64
_InterlockedExchange64_acq	_InterlockedExchangeAdd	_InterlockedExchangeAdd_acq
_InterlockedExchangeAdd_rel	_InterlockedExchangeAdd64	_InterlockedExchangeAdd64_acq
_InterlockedExchangeAdd64_rel	_InterlockedExchangePointer	_InterlockedExchangePointer_acq

_InterlockedIncrement	_InterlockedIncrement_acq	_InterlockedIncrement_rel
_InterlockedIncrement16	_InterlockedIncrement16_acq	_InterlockedIncrement16_rel
_InterlockedIncrement64	_InterlockedIncrement64_acq	_InterlockedIncrement64_rel
_InterlockedOr	_InterlockedOr64	_InterlockedXor
_InterlockedXor64	__invalat	__isNat
__isrlz	__lfetch	__lfetch_excl
__lfetchfault	__lfetchfault_excl	__load128
__load128_acq	__m64_czx1l	
__m64_czx1r	__m64_czx2l	__m64_czx2r
__m64_dep_mi	__m64_dep_mr	__m64_dep_zi
__m64_dep_zr	__m64_extr	__m64_extru
__m64_mix1l	__m64_mix1r	__m64_mix2l
__m64_mix2r	__m64_mix4l	__m64_mix4r
__m64_muladd64hi	__m64_muladd64hi_u	__m64_muladd64lo
__m64_mux1	__m64_mux2	__m64_padd1uus
__m64_padd2uus	__m64_pavg1_nraz	__m64_pavg2_nraz
__m64_pavgsub1	__m64_pavgsub2	__m64_pmpy2l
__m64_pmpy2r	__m64_pmpyshr2	__m64_pmpyshr2u
__m64_popcnt	__m64_pshladd2	__m64_pshradd2
__m64_psub1uus	__m64_psub2uus	__m64_shladd
__m64_shrp	__mf	__mfa
__mul128	__ptcg	__ptcga
__ptcl	__ptrd	__ptri
_rdteb	_ReadBarrier	_ReadWriteBarrier
_ReleaseSpinLock	_ReturnAddress	__rsm
__rum	__setReg	__shiftleft128

__shiftright128	__ssm	__store128
__store128_rel	__sum	
__synci	__thash	__ttag
__umul128	__UMULH	_WriteBarrier
__yield		

In addition, all CRT functions in Intrinsics Available on All Architectures are enabled as intrinsics on the 64-bit Intel architecture, as well as the following additional CRT function:

_setjmpex		

Also, many MMX, Streaming SIMD Extensions (SSE), and Streaming SIMD Extensions 2 (SSE2) intrinsics are also available on the Intel 64-bit architecture.

See Also

Reference

Intrinsics Available on All Architectures
x86 Intrinsics
Intrinsics Available on Intel 64-bit Architectures
MMX, SSE, and SSE2 Intrinsics
Compiler Intrinsics

x64 Intrinsics

The Visual C++ compiler includes the following intrinsics available on the AMD 64-bit architecture:

Index

A – H I – K L – Q R - Z Other Intrinsics

A – H

__addgsbyte

__addgsdword

__addgsqword

__addgsword

__assume

_AddressOfReturnAddress

_BitScanForward

_BitScanForward64

_BitScanReverse

_BitScanReverse64

_bittest

_bittest64

_bittestandcomplement

_bittestandcomplement64

_bittestandreset64

_bittestandreset

_bittestandset64

_bittestandset

__debugbreak

_disable

__emul

__emulu

_enable

__faststorefence

__getcallerseflags

__halt

[Go back to top]

I – K

__inbyte

__inbytestring

__incgsbyte

__incgsdword

__incgsqword

__incgsword

__indword

__indwordstring

__int2c

__invlpg

__inword

__inwordstring

_InterlockedAnd64

_InterlockedAnd

_interlockedbittestandreset64

_interlockedbittestandreset

_interlockedbittestandset64

_interlockedbittestandset

_InterlockedCompare64Exchange128_acq

_InterlockedCompare64Exchange128_rel

_InterlockedCompare64Exchange128

_InterlockedCompareExchange64

_InterlockedCompareExchange

_InterlockedCompareExchangePointer

_InterlockedDecrement

_InterlockedDecrement64

_InterlockedExchange64

_InterlockedExchangeAdd

_InterlockedExchangeAdd64

_InterlockedExchange

_InterlockedExchangePointer

_InterlockedIncrement

_InterlockedIncrement64

_InterlockedOr

_InterlockedOr64

_InterlockedXor64

_InterlockedXor

[Go back to top]

L – Q

__lidt

__ll_lshift

__ll_rshift

__load128

__movsb

__movsd

__movsq

__movsw

__mul128

__mulh

_m_prefetch

_m_prefetchw

_mm_cvtsd_si64x

_mm_cvtsi128_si64x

_mm_cvtsi64x_sd

_mm_cvtsi64x_si128

_mm_cvtsi64x_ss

_mm_cvtss_si64x

_mm_cvttsd_si64x

_mm_cvttss_si64x

_mm_set_epi64x

_mm_set1_epi64x

_mm_setl_epi64

_mm_stream_si64x

__nop

__outbyte

__outbytestring

__outdword

__outdwordstring

[Go back to top]

R - Z

__rdtsc

__readcr0

__readcr2

__readcr3

__readcr4

__readcr8

__readcr8

__readdr

__readeflags

__readfsbyte

__readfsdword

__readfsqword

__readfsword

__readgsbyte

__readgsdword

__readgsqword

__readgsword

__readmsr

__readpmc

_ReadBarrier

_ReadWriteBarrier

_ReturnAddress

__segmentlimit

__shiftleft128

__shiftright128

__sidt

__store128

__stosb

__stosd

__stosq

__stosw

__svm_clgi

__svm_invlpga

__svm_skinit

__svm_stgi

__svm_vmload

__svm_vmrun

__svm_vmsave

__ud2

__ull_rshift

__umulh

_umul128

__vmx_off

__vmx_on

__vmx_vmclear

__vmx_vmlaunch

__vmx_vmptrld

__vmx_vmptrst

__vmx_vmread

__vmx_vmresume

__vmx_vmwrite

__wbinvd

__writecr0

__writecr3

__writecr4

__writecr8

__writecr8

__writedr

__writeeflags

__writefsbyte

__writefsdword

__writefsqword

__writefsword

__writegsbyte

__writegsdword

__writegsqword

__writegsword

__writemsr

_WriteBarrier

[Go back to top]

Other Intrinsics

All CRT functions in Intrinsics Available on All Architectures are enabled as intrinsics on the 64-bit AMD architecture. In addition, the following CRT functions are available:

_setjmpex

inp

_inp

inpd

_inpd

inpw

_inpw

outp

_outp

outpd

_outpd

outpw

_outpw

Finally, note that many MMX, Streaming SIMD Extensions (SSE), and Streaming SIMD Extensions 2 (SSE2) intrinsics are available on the AMD 64-bit platform.

[Go back to top]

See Also

Reference

Intrinsics Available on All Architectures
x86 Intrinsics
Itanium Intrinsics
MMX, SSE, and SSE2 Intrinsics
Compiler Intrinsics

Instruction Reference

This section enables you to look up an intrinsic by instruction. The following table lists instructions, followed by the intrinsic and the platforms for which that intrinsic is available. In some cases, more than one intrinsic can generate a single instruction; for example, if an instruction takes arguments of different type or size.

Instruction	Intrinsic	X86	X64	IPF
addps	_mm_add_ps	SSE	Yes	No
addpd	_mm_add_pd	SSE2	Yes	No
addsd	_mm_add_sd	SSE2	Yes	No
addss	_mm_add_ss	SSE	Yes	No
addsubpd	_mm_addsub_pd	SSE3	Yes	No
addsubps	_mm_addsub_ps	SSE3	Yes	No
andpd	_mm_and_pd	SSE2	Yes	No
andps	_mm_and_ps	SSE	Yes	No
andnpd	_mm_andnot_pd	SSE2	Yes	No
andnps	_mm_andnot_ps	SSE	Yes	No
break	__break	No	No	Yes
bsf	_BitScanForward	Yes	Yes	Yes
bsr	_BitScanReverse	Yes	Yes	Yes
bt	_bittest	Yes	Yes	Yes
btc	_bittestandcomplement	Yes	Yes	Yes
btr	_bittestandreset	Yes	Yes	Yes
bts	_bittestandset	Yes	Yes	Yes

cmpeqpd	_mm_cmpeq_pd	SSE 2	Yes	No
cmpeqps	_mm_cmpeq_ps	SSE	Yes	No
cmpeqsd	_mm_cmpeq_sd	SSE 2	Yes	No
cmpeqss	_mm_cmpeq_ss	SSE	Yes	No
cmplepd	_mm_cmple_pd	SSE 2	Yes	No
cmpleps	_mm_cmple_ps	SSE	Yes	No
cmplesd	_mm_cmple_sd	SSE 2	Yes	No
cmpless	_mm_cmple_ss	SSE	Yes	No
cmpltpd	_mm_cmplt_pd	SSE 2	Yes	No
cmpltps	_mm_cmplt_ps	SSE	Yes	No
cmpltsd	_mm_cmplt_sd	SSE 2	Yes	No
cmpltss	_mm_cmplt_ss	SSE	Yes	No
cmpneqpd	_mm_cmpneq_pd	SSE 2	Yes	No
cmpneqps	_mm_cmpneq_ps	SSE	Yes	No
cmpneqsd	_mm_cmpneq_sd	SSE 2	Yes	No
cmpneqss	_mm_cmpneq_ss	SSE	Yes	No
cmpnlepd	_mm_cmpnle_pd	SSE 2	Yes	No
cmpnleps	_mm_cmpnle_ps	SSE	Yes	No
cmpnlesd	_mm_cmpnle_sd	SSE 2	Yes	No
cmpnless	_mm_cmpnle_ss	SSE	Yes	No
cmpnltpd	_mm_cmpnlt_pd	SSE 2	Yes	No
cmpnltps	_mm_cmpnlt_ps	SSE	Yes	No

cmpnltsd	_mm_cmpnlt_sd	SSE 2	Yes	No
cmpnltss	_mm_cmpnlt_ss	SSE	Yes	No
cmpordpd	_mm_cmpord_pd	SSE 2	Yes	No
cmpordps	_mm_cmpord_ps	SSE	Yes	No
cmpordsd	_mm_cmpord_sd	SSE 2	Yes	No
cmpordss	_mm_cmpord_ss	SSE	Yes	No
cmpunordpd	_mm_cmpunord_pd	SSE 2	Yes	No
cmpunordps	_mm_cmpunord_ps	SSE	Yes	No
cmpunordsd	_mm_cmpunord_sd	SSE 2	Yes	No
cmpunordss	_mm_cmpunord_ss	SSE	Yes	No
cmpxchg4	_InterlockedCompareExchange	Yes	Yes	Yes
cmpxchg8	_InterlockedCompareExchange64	Yes	Yes	Yes
comisd	_mm_comieq_sd, _mm_comilt_sd, _mm_comile_sd, _mm_comigt_sd, _mm_comige_sd, _mm_comineq_sd	SSE 2	Yes	No
comiss	_mm_comieq_ss, _mm_comilt_ss, _mm_comile_ss, _mm_comigt_ss, _mm_comige_ss, _mm_comineq_ss	SSE	Yes	No
cpuid	__cpuid	Yes	Yes	No
cvtdq2pd	_mm_cvtepi32_pd	SSE 2	Yes	No
cvtdq2ps	_mm_cvtepi32_ps	SSE 2	Yes	No
cvtpd2dq	_mm_cvtpd_epi32	SSE 2	Yes	No
cvtpd2pi	_mm_cvtpd_pi32	SSE 2	No	No
cvtpd2ps	_mm_cvtpd_ps	SSE 2	Yes	No

cvtpi2pd	_mm_cvtpi32_pd	SSE 2	No	No
cvtpi2ps	_mm_cvtpi32_ps	SSE	No	No
cvtpi2ps	_mm_cvt_pi2ps	SSE 2	No	No
cvtps2dq	_mm_cvtpd_epi32	SSE 2	Yes	No
cvtps2pd	_mm_cvtps_pd	SSE 2	Yes	No
cvtps2pi	_mm_cvtps_pi32	SSE	No	No
cvtsd2si	_mm_cvtsd_si32	SSE 2	Yes	No
cvtsd2si	_mm_cvtsd_si64x	No	Yes	No
cvtsd2ss	_mm_cvtsd_ss	SSE 2	Yes	No
cvtsi2sd	_mm_cvtsi32_sd	SSE 2	Yes	No
cvtsi2sd	_mm_cvtsi64x_sd	No	Yes	No
cvtsi2ss	_mm_cvtsi32_ss	SSE	Yes	No
cvtsi2ss	_mm_cvt_si2ss	SSE 2	Yes	No
cvtsi2ss	_mm_cvtsi64x_ss	No	Yes	No
cvtss2sd	_mm_cvtss_sd	SSE 2	Yes	No
cvtss2si	_mm_cvtss_si32	SSE	Yes	No
cvtss2si	_mm_cvtss2si, _mm_cvtss_si64x	No	Yes	No
cvttpd2dq	_mm_cvtpd_epi32	SSE 2	Yes	No
cvttpd2pi	_mm_cvtpd_pi32	SSE 2	No	No
cvttps2dq	_mm_cvttps_epi32	SSE 2	Yes	No
cvttps2pi	_mm_cvttps_pi32	SSE	No	No

cvttps2pi	_mm_cvtt_ps2pi	SSE2	No	No
cvttsd2si	_mm_cvttsd_si64x	No	Yes	No
cvttss2si	_mm_cvttss_si32	SSE	Yes	No
cvttss2si	_mm_cvtt_ss2si	SSE2	Yes	No
cvttss2si	_mm_cvttss_si64x	No	Yes	No
czx1.l	__m64_czx1l	No	No	Yes
czx1.r	__m64_czx1r	No	No	Yes
czx2.l	__m64_czx2l	No	No	Yes
czx2.r	__m64_czx2r	No	No	Yes
dep.mi	__m64_dep_mi	No	No	Yes
dep.mr	__m64_dep_mr	No	No	Yes
dep.zi	__m64_dep_zi	No	No	Yes
dep.zr	__m64_dep_zr	No	No	Yes
Divps	_mm_div_ps	SSE	Yes	No
divpd	_mm_div_pd	SSE2	Yes	No
divsd	_mm_div_sd	SSE2	Yes	No
divss	_mm_div_ss	SSE	Yes	No
emms	_mm_empty	MMX	No	Yes
extr	__m64_extr	No	No	Yes
extr.u	__m64_extru	No	No	Yes

fc	__fc	No	No	Yes
fc.i	__fci	No	No	Yes
fclrf.s0	__fclrf	No	No	Yes
flushrs	__flushrs	No	No	Yes
fsetc	__fsetc	No	No	Yes
fwb	__fwb	No	No	Yes
haddpd	_mm_hadd_pd	SSE3	Yes	No
haddps	_mm_hadd_ps	SSE3	Yes	No
hsubpd	_mm_hsub_pd	SSE3	Yes	No
hsubps	_mm_hsub_ps	SSE3	Yes	No
in	__inbyte, __indword, __inword	Yes	Yes	No
int 2c	__int2c	Yes	Yes	No
invala	__invalat	No	No	Yes
invlpg	__invlpg	Yes	Yes	No
ldmxcsr	_mm_setcsr	SSE	Yes	No
lfetch	__lfetch	No	No	Yes
lfetch.excl	__lfetch_excl	No	No	Yes
lfetch.fault	__lfetchfault	No	No	Yes
lfetch.fault.excl	__lfetchfault_excl	No	No	Yes
lock_btr	_interlockedbittestandreset	Yes	Yes	No

lock_bts	_interlockedbittestandset	Yes	Yes	No
maskmovq	_mm_maskmove_si64	SSE	No	Yes
maskmovdqu	_mm_maskmoveu_si128	SSE2	Yes	No
maxsd	_mm_max_sd	SSE2	Yes	No
Maxpd	_mm_max_pd	SSE2	Yes	No
maxps	_mm_max_ps	SSE	Yes	No
maxss	_mm_max_ss	SSE	Yes	No
mf	__mf	No	No	Yes
mf.a	__mfa	No	No	Yes
minps	_mm_min_ps	SSE	Yes	No
minss	_mm_min_ss	SSE	Yes	No
mix1.l	__m64_mix1l	No	No	Yes
mix1.r	__m64_mix1r	No	No	Yes
mix2.l	__m64_mix2l	No	No	Yes
mix2.r	__m64_mix2r	No	No	Yes
mix4.l	__m64_mix4l	No	No	Yes
mix4.r	__m64_mix4r	No	No	Yes
monitor	_mm_monitor	SSE3	Yes	No
movapd	_mm_load_pd, _mm_store_pd	SSE2	Yes	No
movaps	_mm_load_ps, _mm_store_ps	SSE	Yes	No

movd	_mm_cvtsi32_si64, _mm_cvtsi64_si32	MMX	No	Yes
movd	_mm_cvtsi32_si128, _mm_cvtsi128_si32	SSE2	Yes	No
movd	_mm_cvtsi128_si64x, _mm_cvtsi64x_si128	No	Yes	No
movddup	_mm_loaddup_pd, _mm_movedup_pd	SSE3	No	Yes
movdq2q	_mm_movepi64_pi64	SSE2	No	No
movdqa	_mm_load_si128, _mm_store_si128	SSE2	Yes	No
movdqu	_mm_loadu_si128, _mm_storeu_si128	SSE2	Yes	No
movhlps	_mm_movehl_ps	SSE	Yes	No
movhpd	_mm_loadh_pd, _mm_storeh_pd	SSE2	Yes	No
movhps	_mm_loadh_pi, _mm_storeh_pi	SSE	Yes	No
movlhps	_mm_movelh_ps	SSE	Yes	No
movlpd	_mm_loadl_pd, _mm_storel_pd, _mm_stream_pd	SSE2	Yes	No
movlps	_mm_loadl_pi, _mm_storel_pi	SSE	Yes	No
movmskpd	_mm_movemask_pd	SSE2	Yes	No
movmskps	_mm_movemask_ps	SSE	Yes	No
movntps	_mm_stream_ps	SSE	Yes	No
movntq	_mm_stream_pi	SSE	No	No
movq	_mm_move_epi64, _mm_loadl_epi64	SSE2	Yes	No
movq	_mm_storel_epi64	SSE2	No	No
movq2dq	_mm_movpi64_pi64	SSE2	No	No
movsd	_mm_load_sd, _mm_move_sd, _mm_store_sd	SSE2	Yes	No

movshdup	_mm_movehdup_ps	SSE 3	Yes	No
movsldup	_mm_moveldup_ps	SSE 3	Yes	No
movss	_mm_load_ss, _mm_store_ss, _mm_move_ss	SSE	Yes	No
movupd	_mm_loadu_pd, _mm_storeu_pd	SSE 2	Yes	No
movups	_mm_loadu_ps, _mm_storeu_ps	SSE	Yes	No
mulpd	_mm_mul_pd	SSE 2	Yes	No
mulps	_mm_mul_ps	SSE	Yes	No
mulsd	_mm_mul_sd	SSE 2	Yes	No
mulss	_mm_mul_ss	SSE	Yes	No
mux1	__m64_mux1	No	No	Yes
mux2	__m64_mux2	No	No	Yes
mwait	_mm_mwait	SSE 3	Yes	No
orpd	_mm_or_pd	SSE 2	Yes	No
orps	_mm_or_ps	SSE	Yes	No
out	__outbyte, __outdword, __outword	Yes	Yes	No
padd1.uus	__m64_padd1uus	No	No	Yes
padd2.uus	__m64_padd2uus	No	No	Yes
packssdw	_mm_packs_pi32	MMX	No	Yes
packssdw	_mm_packs_epi32	SSE 2	Yes	No
packsswb	_mm_packs_pi16	MMX	No	Yes

packsswb	_mm_packs_epi16	SSE2	Yes	No
packuswb	_mm_packs_pu16	MMX	No	Yes
packuswb	_mm_packus_epi16	SSE2	Yes	No
paddb	_mm_add_pi8	MMX	No	Yes
paddb	_mm_add_epi8	SSE2	Yes	No
paddd	_mm_add_pi32	MMX	No	Yes
paddd	_mm_add_epi32	SSE2	Yes	No
paddq	_mm_add_si64	SSE2	No	Yes
paddq	_mm_add_epi64	SSE2	No	No
paddsb	_mm_add_pi8	MMX	No	Yes
paddsb	_mm_adds_epi8	SSE2	Yes	No
paddsw	_mm_adds_pi16	MMX	No	Yes
paddsw	_mm_adds_epi16	SSE2	Yes	No
paddusb	_mm_adds_pu8	MMX	No	Yes
paddusb	_mm_adds_epu8	SSE2	Yes	No
paddusw	_mm_adds_pu16	MMX	No	Yes
paddusw	_mm_adds_epu16	SSE2	Yes	No
paddw	_mm_adds_pi16	MMX	No	Yes
paddw	_mm_add_epi16	SSE2	Yes	No

pand	_mm_and_si64	MMX	No	Yes
pand	_mm_and_si128	SSE2	Yes	No
pandn	_mm_andnot_si64	MMX	No	Yes
pandn	_mm_andnot_si128	SSE2	Yes	No
pavg1.nraz	__m64_pavg1_nraz	No	No	Yes
pavg2.nraz	__m64_pavg2_nraz	No	No	Yes
pavgb	_mm_avg_pu8	SSE	No	Yes
pavgb	_mm_avg_epu8	SSE2	Yes	No
pavgw	_mm_avg_pu16	SSE	No	Yes
pavgw	_mm_avg_epu16	SSE2	Yes	No
pavgsub1	__m64_pavgsub1	No	No	Yes
pavgsub2	__m64_pavgsub2	No	No	Yes
pcmpeqb	_mm_cmpeq_pi8	MMX	No	Yes
pcmpeqb	_mm_cmpeq_epi8	SSE2	Yes	No
pcmpeqd	_mm_cmpeq_pi32	MMX	No	Yes
pcmpeqd	_mm_cmpeq_epi32	SSE2	Yes	No
pcmpeqw	_mm_cmpeq_pi16	MMX	No	Yes
pcmpeqw	_mm_cmpeq_epi16	SSE2	Yes	No

pcmpgtb	_mm_cmpgt_pi8	MMX	No	Yes
pcmpgtb	_mm_cmpgt_epi8	SSE2	Yes	No
pcmpgtd	_mm_cmpgt_pi32	MMX	No	Yes
pcmpgtd	_mm_cmpgt_epi32	SSE2	Yes	No
pcmpgtw	_mm_cmpgt_pi16	MMX	No	Yes
pcmpgtw	_mm_cmpgt_epi16	SSE2	Yes	No
pextrw	_mm_extract_pi16	SSE	No	Yes
pextrw	_mm_extract_epi16	SSE2	Yes	No
pinsrw	_mm_insert_pi16	SSE	No	Yes
pinsrw	_mm_insert_epi16	SSE2	Yes	No
pmaddwd	_mm_madd_pi16	MMX	No	Yes
pmaddwd	_mm_madd_epi16	SSE2	Yes	No
pmaxsw	_mm_max_pi16	SSE	No	Yes
pmaxsw	_mm_max_epi16	SSE2	Yes	No
pmaxub	_mm_max_pu8	MMX	No	Yes
pmaxub	_mm_max_epu8	SSE2	No	Yes
pminsw	_mm_min_pi16	SSE	No	Yes
pminsw	_mm_min_epi16	SSE2	Yes	No

pminub	_mm_min_pu8	SSE	No	Yes
pminub	_mm_min_epu8	SSE2	Yes	No
pmovmskb	_mm_movemask_pi8	SSE	No	Yes
pmovmskb	_mm_movemask_epi8	SSE2	Yes	No
pmpy2.l	__m64_pmpy2l	No	No	Yes
pmpy2.r	__m64_pmpy2r	No	No	Yes
pmpyshr2	__m64_pmpyshr2	No	No	Yes
pmpyshr2.u	__m64_pmpyshr2u	No	No	Yes
pmulhw	_mm_mulhi_pi16	MMX	No	Yes
pmulhw	_mm_mulhi_epi16	SSE2	Yes	No
pmulhuw	_mm_mulhi_pu16	SSE	No	Yes
pmulhuw	_mm_mulhi_epu16	SSE2	Yes	No
pmullw	_mm_mullo_pi16	MMX	No	Yes
pmullw	_mm_mullo_epi16	SSE2	Yes	No
pmuludq	_mm_mul_su32	SSE2	No	No
pmuludq	_mm_mul_epu32	SSE2	Yes	No
popcnt	__m64_popcnt	No	No	Yes
por	_mm_or_si64	MMX	No	Yes

por	_mm_or_si128	SSE2	Yes	No
prefetch	_mm_prefetch	SSE	Yes	No
psadbw	_mm_sad_pu8	SSE	No	Yes
psadbw	_mm_sad_epu8	SSE2	Yes	No
pshladd2	__m64_pshladd2	No	No	Yes
pshradd2	__m64_pshradd2	No	No	Yes
pshufd	_mm_shuffle_epi32	SSE2	Yes	No
pshufhw	_mm_shufflehi_epi16	SSE2	Yes	No
pshuflw	_mm_shufflelo_epi16	SSE2	Yes	No
pshufw	_mm_shuffle_pi16	SSE	No	Yes
pslld	_mm_sll_pi32	MMX	No	Yes
pslld	_mm_sll_epi32, _mm_slli_epi32	SSE2	Yes	No
pslldi	_mm_slli_pi32	MMX	No	Yes
pslldq	_mm_slli_si128	SSE2	Yes	No
psllq	_mm_sll_si64	MMX	No	Yes
psllq	_mm_sll_epi64, _mm_slli_epi64	SSE2	Yes	No
psllqi	_mm_slli_si64	MMX	No	Yes
psllw	_mm_sll_pi16	MMX	No	Yes
psllw	_mm_sll_epi16, _mm_slli_epi16	SSE2	Yes	No

psllwi	_mm_slli_pi16	MMX	No	Yes
psrad	_mm_sra_pi32	MMX	No	Yes
psrad	_mm_sra_epi32, _mm_srai_epi32	SSE2	Yes	No
psradi	_mm_srai_pi32	MMX	No	Yes
psraw	_mm_sra_pi16	MMX	No	Yes
psraw	_mm_sra_epi16, _mm_srai_epi16	SSE2	Yes	No
psrawi	_mm_srai_pi16	MMX	No	Yes
psrld	_mm_srl_pi32	MMX	No	Yes
psrld	_mm_srl_epi32, _mm_srli_epi32	SSE2	Yes	No
psrldi	_mm_srli_pi32	MMX	No	Yes
psrldq	_mm_srli_si128	SSE2	Yes	No
psrlq	_mm_srl_si64	MMX	No	Yes
psrlq	_mm_srl_epi64, _mm_srli_epi64	SSE2	Yes	No
psrlqi	_mm_srli_si64	MMX	No	Yes
psrlw	_mm_srl_pi16	MMX	No	Yes
psrlw	_mm_srl_epi16, _mm_srli_epi16	SSE2	Yes	No
psrlwi	_mm_srli_pi16	MMX	No	Yes
psub1.uus	__m64_psub1uus	No	No	Yes
psub2.uus	__m64_psub2uus	No	No	Yes

psubb	_mm_sub_pi8	MMX	No	Yes
psubb	_mm_sub_epi8	SSE2	Yes	No
psubd	_mm_sub_pi32	MMX	No	Yes
psubd	_mm_sub_epi32	SSE2	Yes	No
psubq	_mm_sub_si64	SSE2	No	No
psubq	_mm_sub_epi64	SSE2	Yes	No
psubsb	_mm_subs_pi8	MMX	No	Yes
psubsb	_mm_subs_epi8	SSE2	Yes	No
psubsw	_mm_subs_pi16	MMX	No	Yes
psubsw	_mm_subs_epi16	SSE2	Yes	No
psubusb	_mm_subs_pu8	MMX	No	Yes
psubusb	_mm_subs_epu8	SSE2	Yes	No
psubusw	_mm_subs_pu16	MMX	No	Yes
psubusw	_mm_subs_epu16	SSE2	Yes	No
psubw	_mm_sub_pi16	MMX	No	Yes
psubw	_mm_sub_epi16	SSE2	Yes	No
punpckhbw	_mm_unpackhi_pi8	MMX	No	Yes
punpckhbw	_mm_unpackhi_epi8	SSE2	Yes	No
punpckhdq	_mm_unpackhi_pi32	MMX	No	Yes

punpckhdq	_mm_unpackhi_epi32	SSE2	Yes	No
punpckhqdq	_mm_unpackhi_epi64	SSE2	Yes	No
punpckhwd	_mm_unpackhi_pi16	MMX	No	Yes
punpckhwd	_mm_unpackhi_epi16	SSE2	Yes	No
punpcklbw	_mm_unpacklo_pi8	MMX	No	Yes
punpckldq	_mm_unpacklo_pi32	MMX	No	Yes
punpcklqdq	_mm_unpacklo_epi64	SSE2	Yes	No
punpcklwd	_mm_unpacklo_pi16	MMX	No	Yes
ptcg	__ptcg	No	No	Yes
ptcg.a	__ptcga	No	No	Yes
ptcl	__ptcl	No	No	Yes
ptr.d	__ptrd	No	No	Yes
ptr.i	__ptri	No	No	Yes
pxor	_mm_xor_si64	MMX	No	Yes
pxor	_mm_xor_si128	SSE2	Yes	No
rcpps	_mm_xor_ps	SSE	Yes	No
rcpss	_mm_rcp_ss	SSE	Yes	No
rdmsr	__readmsr	Yes	Yes	No
rdpmc	__readpmc	Yes	Yes	No
rdteb	__rdteb	No	No	Yes

rdtsc	__rdtsc	Yes	Yes	No
rep insb	__inbytestring	Yes	Yes	No
rep insd	__indwordstring	Yes	Yes	No
rep insw	__inwordstring	Yes	Yes	No
rep movsb	__movsb	Yes	Yes	No
rep movsd	__movsd	Yes	Yes	No
rep movsq	__movsq	No	Yes	No
rep movsw	__movsw	Yes	Yes	No
rep outsb	__outbytestring	Yes	Yes	No
rep outsd	__outdwordstring	Yes	Yes	No
rep outsw	__outwordstring	Yes	Yes	No
rep stosb	__stosb	Yes	Yes	No
rep stosd	__stosd	Yes	Yes	No
rep stosq	__stosq	No	Yes	No
rep stosw	__stosw	Yes	Yes	No
Rsm	__rsm	No	No	Yes
rsqrtps	_mm_rsqrt_ps	SSE	Yes	No
rsprtss	_mm_rsqrt_ss	SSE	Yes	No
rum	__rum	No	No	Yes
sfence	_mm_sfence	SSE	Yes	No
shladd	__m64_shladd	No	No	Yes
shrp	__m64_shrp	No	No	Yes
shufpd	_mm_shuffle_pd	SSE2	Yes	No
shufps	_mm_shuffle_ps	SSE	Yes	No

sqrtpd	_mm_sqrt_pd	SSE2	Yes	No
sqrtps	_mm_sqrt_ps	SSE	Yes	No
sqrtsd	_mm_sqrt_sd	SSE2	Yes	No
sqrtss	_mm_sqrt_ss	SSE	Yes	No
srlz.d	__dsrlz	No	No	Yes
srlz.i	__isrlz	No	No	Yes
ssm	__ssm	No	No	Yes
sti	_enable	Yes	Yes	Yes
stmxcsr	_mm_getcsr	SSE	Yes	No
subpd	_mm_sub_pd	SSE2	Yes	No
subps	_mm_sub_ps	SSE	Yes	No
subsd	_mm_sub_sd	SSE2	Yes	No
subss	_mm_sub_ss	SSE	Yes	No
sum	__sum	No	No	Yes
sync.i	__synci	No	No	Yes
thash	__thash	No	No	Yes
tnat.nz	__isNat	No	No	Yes
ttag	__ttag	No	No	yes
ucomisd	_mm_ucomieq_sd, _mm_ucomilt_sd, _mm_ucomile_sd, _mm_ucomigt_sd, _mm_ucomige_sd, _mm_ucomineq_sd	SSE2	Yes	No
ucomiss	_mm_ucomieq_ss, _mm_ucomilt_ss, _mm_ucomile_ss, _mm_ucomigt_ss, _mm_ucomige_ss, _mm_ucomineq_ss	SSE	Yes	No

unpckhpd	_mm_unpackhi_pd	SSE 2	Yes	No
unpckhps	_mm_unpackhi_ps	SSE	Yes	No
unpckhpd	_mm_unpacklo_pd	SSE 2	Yes	No
unpcklps	_mm_unpacklo_ps	SSE	Yes	No
wbinvd	__wbinvd	Yes	Yes	No
wrmsr	__writemsr	Yes	Yes	No
xorpd	_mm_xor_pd	SSE 2	Yes	No
xorps	_mm_xor_ps	SSE	Yes	No
yield	__yield	No	No	Yes

See Also

Reference

Compiler Intrinsics

Alphabetical Listing of Intrinsic Functions

The following sections describe the intrinsic functions available on all architectures. CRT functions that have intrinsics implemented for them are not documented here; these functions are documented in the CRT reference.

See Also

Reference

Compiler Intrinsics

_AcquireSpinLock

Microsoft Specific

Used to enter a critical section, effectively ensuring that no other thread is able to execute the code path between the call to **_AcquireSpinLock** and the call to **_ReleaseSpinLock**.

```
void _AcquireSpinLock(
   unsigned __int64 * Lock
)
```

Parameters

[out] *Lock*

The lock variable: set to 1 when a lock is acquired.

Remarks

Other threads attempting to execute **_AcquireSpinLock** block until the thread with the lock calls **_ReleaseSpinLock**.

This routine is only available as an intrinsic.

Requirements

Intrinsic	Architecture
_AcquireSpinLock	IPF

Header file <intrin.h>

Example

```
// spinlock.cpp
// compile with: /MT
// processor: IPF

#include <cstdlib>
#include <cstdio>
#include <process.h>
#include <windows.h>
#include <intrin.h>

#pragma intrinsic(_AcquireSpinLock, _ReleaseSpinLock)

// Data to protect.

volatile LONG data = 0;
unsigned __int64 lock = 0;
void __cdecl SimpleThread(void* pParam);
const int THREAD_COUNT = 6;

int main()
{
    DWORD num;
    HANDLE threads[THREAD_COUNT];
    int args[THREAD_COUNT];
    int i;

    for (i = 0; i < THREAD_COUNT; i++)
    {
        args[i] = i+1;
        threads[i] =
             reinterpret_cast<HANDLE>(_beginthread(SimpleThread,
                                       0, args + i));
        if (threads[i] == reinterpret_cast<HANDLE>(-1))
        {
```

```
            // Error creating threads
            break;
        }
    }

    WaitForMultipleObjects(i, threads, true, INFINITE);
}

// Code for the simple thread
void __cdecl SimpleThread(void* pParam)
{
    int thread = *( (int*) pParam);
    int k = 1;

    _AcquireSpinLock(&lock);

    int x1 = data++;
    for (int i = 0; i < 10000; i++)
    {
        // Perform some calculations to create a delay
        k += i;
        k %= 10000;
    }
    int x2 = data++;

    printf_s("%I64d %d %d %d\n", lock, thread, x1, k);
    printf_s("%I64d %d %d %d\n", lock, thread, x2, k);

    _ReleaseSpinLock(&lock);

    if (x2 - x1 != 1)
        printf_s("Error!\n");
}
```

Sample Output

```
1 1 0 5001
1 1 1 5001
1 3 2 5001
1 3 3 5001
1 2 4 5001
1 2 5 5001
1 4 6 5001
1 4 7 5001
1 5 8 5001
1 5 9 5001
1 6 10 5001
1 6 11 5001
```

END Microsoft Specific

See Also

Reference

_ReleaseSpinLock

Compiler Intrinsics

__addfsbyte, __addfsword, __addfsdword

Microsoft Specific

Add a value to a memory location specified by an offset relative to the beginning of the **FS** segment.

```
void __addfsbyte(
   unsigned long Offset,
   unsigned char Data
);
void __addfsword(
   unsigned long Offset,
   unsigned short Data
);
void __addfsdword(
   unsigned long Offset,
   unsigned int Data
);
```

Parameters

[in] *Offset*

The offset from the beginning of **FS**.

[in] *Data*

The value to add to the memory location.

Requirements

Intrinsic	Architecture
__addfsbyte	x86
__addfsword	x86
__addfsdword	x86

Remarks

These intrinsics are available in kernel mode only, and these routines are only available as intrinsics.

END Microsoft Specific

See Also

Reference

__incfsbyte, __incfsword, __incfsdword
__readfsbyte, __readfsdword, __readfsqword, __readfsword
__writefsbyte, __writefsdword, __writefsqword, __writefsword
Compiler Intrinsics

__addgsbyte, __addgsword, __addgsdword, __addgsqword

Microsoft Specific

Add a value to a memory location specified by an offset relative to the beginning of the **GS** segment.

```
void __addgsbyte(
   unsigned long Offset,
   unsigned char Data
);
void __addgsword(
   unsigned long Offset,
   unsigned short Data
);
void __addgsdword(
   unsigned long Offset,
   unsigned int Data
);
void __addgsqword(
   unsigned long Offset,
   unsigned __int64 Data
);
```

Parameters

[in] *Offset*

The offset from the beginning of **GS**.

[in] *Data*

The value to add to the memory location.

Requirements

Intrinsic	Architecture
__addgsbyte	x64
__addgsword	x64
__addgsdword	x64
__addgsqword	x64

Remarks

These intrinsics are available in kernel mode only, and these routines are only available as intrinsics.

END Microsoft Specific

See Also

Reference

__incgsbyte, __incgsword, __incgsdword, __incgsqword
__readgsbyte, __readgsdword, __readgsqword, __readgsword
__writegsbyte, __writegsdword, __writegsqword, __writegsword
Compiler Intrinsics

_AddressOfReturnAddress

Microsoft Specific

Provides the address of the memory location that holds the return address of the current function. This address may not be used to access other memory locations (for example, the function's arguments).

```
void * _AddressOfReturnAddress();
```

Requirements

Intrinsic	Architecture
_AddressOfReturnAddress	x86, x64

Header file <intrin.h>

Remarks

When **_AddressOfReturnAddress** is used in a program compiled with /clr, the function containing the **_AddressOfReturnAddress** call is compiled as a native function. When a function compiled as managed calls into the function containing **_AddressOfReturnAddress**, **_AddressOfReturnAddress** might not behave as expected.

This routine is only available as an intrinsic.

Example

```
// compiler_intrinsics_AddressOfReturnAddress.cpp
// processor: x86, x64
#include <stdio.h>
#include <intrin.h>

// This function will print three values:
//   (1) The address retrieved from _AddressOfReturnAdress
//   (2) The return address stored at the location returned in (1)
//   (3) The return address retrieved the _ReturnAddress* intrinsic
// Note that (2) and (3) should be the same address.
__declspec(noinline)
void func() {
   void* pvAddressOfReturnAddress = _AddressOfReturnAddress();
   printf_s("%p\n", pvAddressOfReturnAddress);
   printf_s("%p\n", *((void**) pvAddressOfReturnAddress));
   printf_s("%p\n", _ReturnAddress());
}

int main() {
   func();
}
```

Sample Output

```
0012FF78
00401058
00401058
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

C++ Keywords

__assume

Microsoft Specific

Passes a hint to the optimizer.

```
__assume(
      expression
)
```

Parameters

expression

Any expression that is assumed to evaluate to true.

Remarks

The optimizer assumes that the condition represented by *expression* is true at the point where the keyword appears and remains true until *expression* is altered (for example, by assignment to a variable). Selective use of hints passed to the optimizer by **__assume** can improve optimization.

__assume is not a true intrinsic. It does not need to be declared as a function and cannot be used in a **#pragma intrinsic** directive. No code is generated, although it affects the code generated by the optimizer.

Use **__assume** in an ASSERT only when the assert is not recoverable. **__assume** should not be used in an assert for which you have subsequent error recovery code; the compiler might optimize away the error handling code.

The most common use of **__assume** is with the default case of a switch statement, as shown in the following example.

Requirements

Intrinsic	Architecture
__assume	x86, IPF, x64

Example

```
// compiler_intrinsics__assume.cpp
#ifdef DEBUG
# define ASSERT(e)    ( ((e) || assert(__FILE__, __LINE__) )
#else
# define ASSERT(e)    ( __assume(e) )
#endif

void func1(int i)
{
}

int main(int p)
{
   switch(p){
      case 1:
         func1(1);
         break;
      case 2:
         func1(-1);
         break;
      default:
         __assume(0);
            // This tells the optimizer that the default
            // cannot be reached. As so, it does not have to generate
            // the extra code to check that 'p' has a value
            // not represented by a case arm. This makes the switch
            // run faster.
   }
}
```

Example Remarks

The use of `assume(0)` tells the optimizer that the default case cannot be reached. As a result, the compiler does not generate code to test whether `p` has a value not represented in a case statement. Note that `assume(0)` must be the first statement in the body of the default case for this to work.

Because the compiler generates code based on **__assume**, that code might not run correctly if the expression inside the **__assume** statement is false at run time. If you are not sure that the expression will always be true at run time, you can use the `assert` function to protect the code:

```
# define ASSERT(e)    ( ((e) || assert(__FILE__, __LINE__)), __assume(e) )
```

Unfortunately, this use of `assert` prevents the compiler from performing the default-case optimization previously shown. Therefore, you might want to use a separate macro instead:

```
#ifdef DEBUG
# define NODEFAULT   ASSERT(0)
#else
# define NODEFAULT   __assume(0)
#endif

   default:
      NODEFAULT;
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

C++ Keywords

_BitScanForward, _BitScanForward64

Microsoft Specific

Search the mask data from least significant bit (LSB) to the most significant bit (MSB) for a set bit (1).

```
unsigned char _BitScanForward(
   unsigned long * Index,
   unsigned long Mask
);
unsigned char_BitScanForward64(
   unsigned long * Index,
   unsigned __int64 Mask
);
```

Parameters

[out] *Index*

Loaded with the bit position of the first set bit (1) found.

[in] *Mask*

The 32-bit or 64-bit value to search.

Return Value

0 if the mask is zero; nonzero otherwise.

Remarks

If a set bit is found, the bit position of the first set bit found is returned in the first parameter. If no set bit is found, 0 is returned; otherwise, 1 is returned.

Requirements

Intrinsic	**Architecture**
_BitScanForward	x86, IPF, x64
_BitScanForward64	IPF, x64

Header file <intrin.h>

Example

```
// BitScanForward.cpp
// compile with: /EHsc
#include <iostream>
#include <intrin.h>
using namespace std;

#pragma intrinsic(_BitScanForward)

int main()
{
   unsigned long mask = 0x1000;
   unsigned long index;
   unsigned char isNonzero;

   cout << "Enter a positive integer as the mask: " << flush;
   cin >> mask;
   isNonzero = _BitScanForward(&index, mask);
   if (isNonzero)
   {
      cout << "Mask: " << mask << " Index: " << index << endl;
   }
```

```
    else
    {
       cout << "No set bits found.  Mask is zero." << endl;
    }
 }
```

Input

```
12
```

Sample Output

```
 Enter a positive integer as the mask:
 Mask: 12 Index: 2
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_BitScanReverse, _BitScanReverse64

Microsoft Specific

Search the mask data from most significant bit (MSB) to least significant bit (LSB) for a set bit (1).

```
unsigned char _BitScanReverse(
   unsigned long * Index,
   unsigned long Mask
);
unsigned char _BitScanReverse64(
   unsigned long * Index,
   unsigned __int64 Mask
);
```

Parameters

[out] *Index*

Loaded with the bit position of the first set bit (1) found.

[in] *Mask*

The 32-bit or 64-bit value to search.

Return Value

0 if the mask is zero; nonzero otherwise.

Remarks

If a set bit is found, the bit position of the first set bit found is returned in the first parameter. If no set bit is found, 0 is returned; otherwise, 1 is returned.

Requirements

Intrinsic	**Architecture**
_BitScanReverse	x86, IPF, x64
_BitScanReverse64	IPF, x64

Header file <intrin.h>

Example

```
// BitScanReverse.cpp
// compile with: /EHsc
#include <iostream>
#include <intrin.h>
using namespace std;

#pragma intrinsic(_BitScanReverse)



int main()
{
   unsigned long mask = 0x1000;
   unsigned long index;
   unsigned char isNonzero;

   cout << "Enter a positive integer as the mask: " << flush;
   cin >> mask;
   isNonzero = _BitScanReverse(&index, mask);
   if (isNonzero)
   {
```

```
        cout << "Mask: " << mask << " Index: " << index << endl;
    }
    else
    {
        cout << "No set bits found.  Mask is zero." << endl;
    }
}
```

Input

```
12
```

Sample Output

```
Enter a positive integer as the mask:
Mask: 12 Index: 3
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_bittest, _bittest64

Microsoft Specific

Generates the **bt** instruction, which examines the bit in position *b* of address *a*, and returns the value of that bit.

```
unsigned char _bittest(
   long *a,
   long b
);
unsigned char _bittest64(
   __int64 *a,
   __int64 b
);
```

Parameters

[in] *a*

A pointer to the memory to examine.

[in] *b*

The bit position to test.

Return Value

The bit at the position specified.

Requirements

Intrinsic	Architecture
_bittest	x86, IPF, x64
_bittest64	IPF, x64

Header file <intrin.h>

Remarks

On the IPF architecture, the **bt** instruction is not available, so this intrinsic is a custom function that imitates the behavior of **bt**. This custom function might be slower than a hand-written inline function because it includes overhead, such as handling the case where *b* is negative, that might be unnecessary in specific cases.

This routine is only available as an intrinsic.

Example

```
// bittest.cpp
// processor: x86, IPF, x64

#include <stdio.h>
#include <intrin.h>

long num = 78002;


int main()
{
    unsigned char bits[32];
    long nBit;

    printf_s("Number: %d\n", num);

    for (nBit = 0; nBit < 31; nBit++)
    {
```

```
        bits[nBit] = _bittest(&num, nBit);
    }

    printf_s("Binary representation:\n");
    while (nBit--)
    {
        if (bits[nBit])
            printf_s("1");
        else
            printf_s("0");
    }
}
```

Output

```
Number: 78002
Binary representation:
00000000000000100110000010110010
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_bittestandcomplement, _bittestandcomplement64

Microsoft Specific

Generate the **btc** instruction, which examines bit *b* of the address *a*, returns its current value, and sets the bit to its complement.

```
unsigned char _bittestandcomplement(
   long *a,
   long b
);
unsigned char _bittestandcomplement64(
   __int64 *a,
   __int64 b
);
```

Parameters

[in, out] *a*

A pointer to the memory to examine.

[in] *b*

The bit position to test.

Return Value

The bit at the position specified.

Requirements

Intrinsic	Architecture
_bittestandcomplement	x86, IPF, x64
_bittestandcomplement64	IPF, x64

Header file <intrin.h>

Remarks

On the IPF architecture, the **btc** instruction is not available, so this intrinsic is a custom function that imitates the behavior of **btc**. This custom function might be slower than a hand-written inline function because it includes overhead, such as handling the case where *b* is negative, that might be unnecessary in specific cases.

This routine is only available as an intrinsic.

Example

```
// bittestandcomplement.cpp
// processor: x86, IPF, x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_bittestandcomplement)
#ifdef _M_AMD64
#pragma intrinsic(_bittestandcomplement64)
#endif

int main()
{
   long i = 1;
   __int64 i64 = 0x1I64;
   unsigned char result;
   printf("Initial value: %d\n", i);
   printf("Testing bit 1\n");
```

```
   result = _bittestandcomplement(&i, 1);
   printf("Value changed to %d, Result: %d\n", i, result);
#ifdef _M_AMD64
   printf("Testing bit 0\n");
   result = _bittestandcomplement64(&i64, 0);
   printf("Value changed to %I64d, Result: %d\n", i64, result);
#endif
}
```

Sample Output

```
Initial value: 1
Testing bit 1
Value changed to 3, Result: 0
Testing bit 0
Value changed to 0, Result: 1
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_bittestandreset, _bittestandreset64

Microsoft Specific

Generate the **btr** instruction, which examines bit *b* of the address *a*, returns its current value, and resets the bit to 0.

```
unsigned char _bittestandreset(
   long *a,
   long b
);
unsigned char _bittestandreset64(
   __int64 *a,
   __int64 b
);
```

Parameters

[in, out] *a*

A pointer to the memory to examine.

[in] *b*

The bit position to test.

Return Value

The bit at the position specified.

Requirements

Intrinsic	Architecture
_bittestandreset	x86, IPF, x64
_bittestandreset64	IPF, x64

Header file <intrin.h>

Remarks

On the IPF architecture, the **btr** instruction is not available, so this intrinsic is a custom function that imitates the behavior of **btr**. This custom function might be slower than a hand-written inline function because it includes overhead, such as handling the case where *b* is negative, that might be unnecessary in specific cases.

This routine is only available as an intrinsic.

Example

```
// bittestandreset.cpp
// processor: x86, IPF, x64
#include <stdio.h>
#include <limits.h>
#include <intrin.h>

#pragma intrinsic(_bittestandreset)

// Check the sign bit and reset to 0 (taking the absolute value)
// Returns 0 if the number is positive or zero
// Returns 1 if the number is negative
unsigned char absolute_value(long* p)
{
   const int SIGN_BIT = 31;
   return _bittestandreset(p, SIGN_BIT);
}

int main()
```

```
{
    long i = -112;
    unsigned char result;

    // Check the sign bit and reset to 0 (taking the absolute value)

    result = absolute_value(&i);
    if (result == 1)
        printf_s("The number was negative.\n");
}
```

Output

```
The number was negative.
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_bittestandset, _bittestandset64

Microsoft Specific

Generate the **bts** instruction, which examines bit *b* of the address *a*, returns its current value, and sets the bit to 1.

```
unsigned char _bittestandset(
   long *a,
   long b
);
unsigned char _bittestandset64(
   __int64 *a,
   __int64 b
);
```

Parameters

[in, out] *a*

A pointer to the memory to examine.

[in] *b*

The bit position to test.

Return Value

The bit at the position specified.

Requirements

Intrinsic	Architecture
_bittestandset	x86, IPF, x64
_bittestandset64	IPF, x64

Header file <intrin.h>

Remarks

On the IPF architecture, the **bts** instruction is not available, so this intrinsic is a custom function that imitates the behavior of **bts**. This custom function might be slower than a hand-written inline function because it includes overhead, such as handling the case where *b* is negative, that might be unnecessary in specific cases.

This routine is only available as an intrinsic.

Example

```
// bittestandset.cpp
// processor: x86, IPF, x64
// This example uses several of the _bittest family of intrinsics
// to implement a Flags class that allows bit level access to an
// integer field.
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_bittestandset, _bittestandreset,\
                  _bittestandcomplement, _bittest)

class Flags
{
private:
    long flags;
    long* oldValues;

public:
```

```
    Flags() : flags(0)
    {
        oldValues = new long[32];
    }

    ~Flags()
    {
        delete oldValues;
    }

    void SetFlagBit(long nBit)
    {
        // We omit range checks on the argument
        oldValues[nBit] = _bittestandset(&flags, nBit);
        printf_s("Flags: 0x%x\n", flags);
    }
    void ClearFlagBit(long nBit)
    {
        oldValues[nBit] = _bittestandreset(&flags, nBit);
        printf_s("Flags: 0x%x\n", flags);
    }
    unsigned char GetFlagBit(long nBit)
    {
        unsigned char result = _bittest(&flags, nBit);
        printf_s("Flags: 0x%x\n", flags);
        return result;
    }
    void RestoreFlagBit(long nBit)
    {
        if (oldValues[nBit])
            oldValues[nBit] = _bittestandset(&flags, nBit);
        else
            oldValues[nBit] = _bittestandreset(&flags, nBit);
        printf_s("Flags: 0x%x\n", flags);
    }
    unsigned char ToggleBit(long nBit)
    {
        unsigned char result = _bittestandcomplement(&flags, nBit);
        printf_s("Flags: 0x%x\n", flags);
        return result;
    }
};

int main()
{
    Flags f;
    f.SetFlagBit(1);
    f.SetFlagBit(2);
    f.SetFlagBit(3);
    f.ClearFlagBit(3);
    f.ToggleBit(1);
    f.RestoreFlagBit(2);
}
```

Output

```
Flags: 0x2
Flags: 0x6
Flags: 0xe
Flags: 0x6
Flags: 0x4
Flags: 0x0
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__break

Microsoft Specific

Generates the IPF **break** instruction, which sends a Break Instruction fault.

```
void __break(
   int Mask
);
```

Parameters

[in] *Mask*

The 21-bit value to write to the Interruption Immediate control register.

Requirements

Intrinsic	Architecture
__break	IPF

Header file <intrin.h>

Remarks

For more information about the **break** instruction, see the Intel IPF documentation.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__cpuid

Microsoft Specific

Generates the **cpuid** instruction available on x86 and x64, which queries the processor for information about the supported features and CPU type.

```
void __cpuid(
   int CPUInfo[4],
   int InfoType
);
```

Parameters

[out] *CPUInfo*

The information returned in an array of four integers.

[in] *InfoType*

A code indicating the type of information to be retrieved.

Requirements

Intrinsic	Architecture
__cpuid	x86, x64

Header file <intrin.h>

Remarks

The supported features and CPU type information is returned in *CPUInfo*, an array of four 32-bit integers that is filled with the values of the EAX, EBX, ECX, and EDX registers (in that order) after the **cpuid** instruction is called. The information returned has a different meaning depending on the value passed as the *InfoType* parameter. The information returned with various values of *InfoType* is processor-dependent.

When the *InfoType* argument is 0, the following table describes the output.

Information Returned

Array index	Bit range	Description
0	0-31	Maximum meaningful value for the *InfoType* parameter.
1	0-31	Identification String (part 1)
2	0-31	Identification String (part 3)
3	0-31	Identification String (part 2)

When the *InfoType* argument is 1, the following table describes the output.

CPU Information Returned

Array index	Bit range	Description
0	0-3	Stepping ID
0	4-7	Model
0	8-11	Family
0	12-13	Processor Type

0	14-15	Reserved
0	16-19	Extended model
0	20-27	Extended family
0	28-31	Reserved
1	0-7	Brand Index
1	8-15	CLFLUSH cache line size / 8
1	16-23	Reserved
1	24-31	APIC Physical ID
2	0	SSE3 New Instructions
2	1-2	Reserved
2	3	MONITOR/MWAIT
2	4	CPL Qualified Debug Store
2	5-7	Reserved
2	8	Thermal Monitor 2
2	9	Reserved
2	10	L1 Context ID
2	11-31	Reserved
3	0-31	Feature Information (see below)

The following table shows the meaning of the Feature Information value, the value of EDX which is written to CPUInfo[3], when the *InfoType* argument is 1.

Feature Information Returned

Bit	Mnemonic	Description
0	FPU	x87 FPU on Chip
1	VME	Virtual-8086 Mode Enhancement
2	DE	Debugging Extensions
3	PSE	Page Size Extensions
4	TSC	Time Stamp Counter
5	MSR	RDMSR and WRMSR Support

6	PAE	Physical Address Extensions
7	MCE	Machine Check Exception
8	CX8	CMPXCHG8B Inst.
9	APIC	APIC on Chip
10	n/a	Reserved
11	SEP	SYSENTER and SYSEXIT
12	MTRR	Memory Type Range Registers
13	PGE	PTE Global Bit
14	MCA	Machine Check Architecture
15	CMOV	Conditional Move/Compare Instruction
16	PAT	Page Attribute Table
17	PSE	Page Size Extension
18	PSN	Processor Serial Number
19	CLFSH	CFLUSH Instruction
20	n/a	Reserved
21	DS	Debug Store
22	ACPI	Thermal Monitor and Clock Ctrl
23	MMX	MMX Technology
24	FXSR	FXSAVE/FXRSTOR
25	SSE	SSE Extensions
26	SSE2	SSE2 Extensions
27	SS	Self Snoop
28	HTT	Hyper-threading technology
29	TM	Thermal Monitor
30	n/a	Reserved
31	PBE	Pend. Brk. En.

Some processors, such as the SSE3-enabled processors, support additional *InfoType* values. When the *InfoType* argument is 3 or 4, these processors might return cache and TLB information. When *InfoType* is 5, these processors return information

relating to the monitor feature (see _mm_monitor).

The SSE3-enabled processors support Extended Function CPUID information. If this is supported, *InfoType* values from 0x8000000 might be used to return information. To determine the maximum meaningful value allowed, set *InfoType* to 0x8000000. The maximum value of *InfoType* supported will be written to *CPUInfo[0]*. The following table shows values returned using the extended function CPUID information.

Extended Function CPUID Information Returned

InfoType	Array index	Information
0x80000000	0	Maximum meaningful value of InfoType for extended function CPUID information.
0x80000000	1-3	Reserved.
0x80000001	0-4	Reserved
0x80000002	0-4	Processor Brand String
0x80000003	0-4	Processor Brand String, continued
0x80000004	0-4	Processor Brand String, continued
0x80000005	0-4	Reserved
0x80000006	0-1	Reserved
0x80000006	2	Bits 0-7: Cache Line Size Bits 12-15: L2 Associativity Bits 16-31: Cache size in 1K units
0x80000006	3	Reserved
0x80000007	0-4	Reserved

Example

```
// cpuid.cpp
// processor: x86, x64
// Use the __cpuid intrinsic to get information about a CPU

#include <stdio.h>
#include <string.h>
#include <intrin.h>

const char* szFeatures[] =
{
    "x87 FPU On Chip",
    "Virtual-8086 Mode Enhancement",
    "Debugging Extensions",
    "Page Size Extensions",
    "Time Stamp Counter",
    "RDMSR and WRMSR Support",
    "Physical Address Extensions",
    "Machine Check Exception",
    "CMPXCHG8B Instruction",
    "APIC On Chip",
    "Unknown1",
    "SYSENTER and SYSEXIT",
    "Memory Type Range Registers",
    "PTE Global Bit",
    "Machine Check Architecture",
    "Conditional Move/Compare Instruction",
```

```
    "Page Attribute Table",
    "Page Size Extension",
    "Processor Serial Number",
    "CFLUSH Extension",
    "Unknown2",
    "Debug Store",
    "Thermal Monitor and Clock Ctrl",
    "MMX Technology",
    "FXSAVE/FXRSTOR",
    "SSE Extensions",
    "SSE2 Extensions",
    "Self Snoop",
    "Hyper-threading Technology",
    "Thermal Monitor",
    "Unknown4",
    "Pend. Brk. EN."
};

int main(int argc, char* argv[])
{
    char CPUString[0x20];
    char CPUBrandString[0x40];
    int CPUInfo[4] = {-1};
    int nSteppingID = 0;
    int nModel = 0;
    int nFamily = 0;
    int nProcessorType = 0;
    int nExtendedmodel = 0;
    int nExtendedfamily = 0;
    int nBrandIndex = 0;
    int nCLFLUSHcachelinesize = 0;
    int nAPICPhysicalID = 0;
    int nFeatureInfo = 0;
    int nCacheLineSize = 0;
    int nL2Associativity = 0;
    int nCacheSizeK = 0;
    int nRet = 0;
    unsigned    nIds, nExIds, i;
    bool    bSSE3NewInstructions = false;
    bool    bMONITOR_MWAIT = false;
    bool    bCPLQualifiedDebugStore = false;
    bool    bThermalMonitor2 = false;


    // __cpuid with an InfoType argument of 0 returns the number of
    // valid Ids in CPUInfo[0] and the CPU identification string in
    // the other three array elements. The CPU identification string is
    // not in linear order. The code below arranges the information
    // in a human readable form.
    __cpuid(CPUInfo, 0);
    nIds = CPUInfo[0];
    memset(CPUString, 0, sizeof(CPUString));
    *((int*)CPUString) = CPUInfo[1];
    *((int*)(CPUString+4)) = CPUInfo[3];
    *((int*)(CPUString+8)) = CPUInfo[2];

    // Get the information associated with each valid Id
    for (i=0; i<=nIds; ++i)
    {
        __cpuid(CPUInfo, i);
        printf_s("\nFor InfoType %d\n", i);
        printf_s("CPUInfo[0] = 0x%x\n", CPUInfo[0]);
        printf_s("CPUInfo[1] = 0x%x\n", CPUInfo[1]);
        printf_s("CPUInfo[2] = 0x%x\n", CPUInfo[2]);
        printf_s("CPUInfo[3] = 0x%x\n", CPUInfo[3]);

        // Interpret CPU feature information.
        if  (i == 1)
```

```
    {
        nSteppingID = CPUInfo[0] & 0xf;
        nModel = (CPUInfo[0] >> 4) & 0xf;
        nFamily = (CPUInfo[0] >> 8) & 0xf;
        nProcessorType = (CPUInfo[0] >> 12) & 0x3;
        nExtendedmodel = (CPUInfo[0] >> 16) & 0xf;
        nExtendedfamily = (CPUInfo[0] >> 20) & 0xff;
        nBrandIndex = CPUInfo[1] & 0xff;
        nCLFLUSHcachelinesize = ((CPUInfo[1] >> 8) & 0xff) * 8;
        nAPICPhysicalID = (CPUInfo[1] >> 24) & 0xff;
        bSSE3NewInstructions = (CPUInfo[2] & 0x1) || false;
        bMONITOR_MWAIT = (CPUInfo[2] & 0x8) || false;
        bCPLQualifiedDebugStore = (CPUInfo[2] & 0x10) || false;
        bThermalMonitor2 = (CPUInfo[2] & 0x100) || false;
        nFeatureInfo = CPUInfo[3];
    }
}

// Calling __cpuid with 0x80000000 as the InfoType argument
// gets the number of valid extended IDs.
__cpuid(CPUInfo, 0x80000000);
nExIds = CPUInfo[0];
memset(CPUBrandString, 0, sizeof(CPUBrandString));

// Get the information associated with each extended ID.
for (i=0x80000000; i<=nExIds; ++i)
{
    __cpuid(CPUInfo, i);
    printf_s("\nFor InfoType %x\n", i);
    printf_s("CPUInfo[0] = 0x%x\n", CPUInfo[0]);
    printf_s("CPUInfo[1] = 0x%x\n", CPUInfo[1]);
    printf_s("CPUInfo[2] = 0x%x\n", CPUInfo[2]);
    printf_s("CPUInfo[3] = 0x%x\n", CPUInfo[3]);

    // Interpret CPU brand string and cache information.
    if  (i == 0x80000002)
        memcpy(CPUBrandString, CPUInfo, sizeof(CPUInfo));
    else if  (i == 0x80000003)
        memcpy(CPUBrandString + 16, CPUInfo, sizeof(CPUInfo));
    else if  (i == 0x80000004)
        memcpy(CPUBrandString + 32, CPUInfo, sizeof(CPUInfo));
    else if  (i == 0x80000006)
    {
        nCacheLineSize = CPUInfo[2] & 0xff;
        nL2Associativity = (CPUInfo[2] >> 12) & 0xf;
        nCacheSizeK = (CPUInfo[2] >> 16) & 0xffff;
    }
}

// Display all the information in user-friendly format.

printf_s("\n\nCPU String: %s\n", CPUString);

if  (nIds >= 1)
{
    if  (nSteppingID)
        printf_s("Stepping ID = %d\n", nSteppingID);
    if  (nModel)
        printf_s("Model = %d\n", nModel);
    if  (nFamily)
        printf_s("Family = %d\n", nFamily);
    if  (nProcessorType)
        printf_s("Processor Type = %d\n", nProcessorType);
    if  (nExtendedmodel)
        printf_s("Extended model = %d\n", nExtendedmodel);
    if  (nExtendedfamily)
        printf_s("Extended family = %d\n", nExtendedfamily);
    if  (nBrandIndex)
```

```
            printf_s("Brand Index = %d\n", nBrandIndex);
        if  (nCLFLUSHcachelinesize)
            printf_s("CLFLUSH cache line size = %d\n",
                     nCLFLUSHcachelinesize);
        if  (nAPICPhysicalID)
            printf_s("APIC Physical ID = %d\n", nAPICPhysicalID);

  if  (nFeatureInfo || bSSE3NewInstructions ||
             bMONITOR_MWAIT || bCPLQualifiedDebugStore ||
             bThermalMonitor2)
        {
            printf_s("\nThe following features are supported:\n");

  if  (bSSE3NewInstructions)
  printf_s("\tSSE3 New Instructions\n");
  if  (bMONITOR_MWAIT)
  printf_s("\tMONITOR/MWAIT\n");
  if  (bCPLQualifiedDebugStore)
  printf_s("\tCPL Qualified Debug Store\n");
  if  (bThermalMonitor2)
  printf_s("\tThermal Monitor 2\n");

            i = 0;
            nIds = 1;
            while (i < (sizeof(szFeatures)/sizeof(const char*)))
            {
                if  (nFeatureInfo & nIds)
                {
                    printf_s("\t");
                    printf_s(szFeatures[i]);
                    printf_s("\n");
                }

                nIds <<= 1;
                ++i;
            }
        }
    }

    if  (nExIds >= 0x80000004)
        printf_s("\nCPU Brand String: %s\n", CPUBrandString);

    if  (nExIds >= 0x80000006)
    {
        printf_s("Cache Line Size = %d\n", nCacheLineSize);
        printf_s("L2 Associativity = %d\n", nL2Associativity);
        printf_s("Cache Size = %dK\n", nCacheSizeK);
    }

  return  nRet;
  }
```

Sample Output

```
For InfoType 0
CPUInfo[0] = 0x5
CPUInfo[1] = 0x756e6547
CPUInfo[2] = 0x6c65746e
CPUInfo[3] = 0x49656e69

For InfoType 1
CPUInfo[0] = 0xf31
CPUInfo[1] = 0x20800
CPUInfo[2] = 0x41d
CPUInfo[3] = 0xbfebfbff

For InfoType 2
CPUInfo[0] = 0x605b5001
CPUInfo[1] = 0x0
```

```
CPUInfo[2] = 0x0
CPUInfo[3] = 0x7c7040

For InfoType 3
CPUInfo[0] = 0x0
CPUInfo[1] = 0x0
CPUInfo[2] = 0x0
CPUInfo[3] = 0x0

For InfoType 4
CPUInfo[0] = 0x0
CPUInfo[1] = 0x0
CPUInfo[2] = 0x0
CPUInfo[3] = 0x0

For InfoType 5
CPUInfo[0] = 0x40
CPUInfo[1] = 0x40
CPUInfo[2] = 0x0
CPUInfo[3] = 0x0

For InfoType 80000000
CPUInfo[0] = 0x80000008
CPUInfo[1] = 0x0
CPUInfo[2] = 0x0
CPUInfo[3] = 0x0

For InfoType 80000001
CPUInfo[0] = 0x0
CPUInfo[1] = 0x0
CPUInfo[2] = 0x0
CPUInfo[3] = 0x0

For InfoType 80000002
CPUInfo[0] = 0x20202020
CPUInfo[1] = 0x20202020
CPUInfo[2] = 0x20202020
CPUInfo[3] = 0x20202020

For InfoType 80000003
CPUInfo[0] = 0x47202020
CPUInfo[1] = 0x69756e65
CPUInfo[2] = 0x4920656e
CPUInfo[3] = 0x6c65746e

For InfoType 80000004
CPUInfo[0] = 0x20295228
CPUInfo[1] = 0x20555043
CPUInfo[2] = 0x30382e32
CPUInfo[3] = 0x7a4847

For InfoType 80000005
CPUInfo[0] = 0x0
CPUInfo[1] = 0x0
CPUInfo[2] = 0x0
CPUInfo[3] = 0x0

For InfoType 80000006
CPUInfo[0] = 0x0
CPUInfo[1] = 0x0
CPUInfo[2] = 0x4008040
CPUInfo[3] = 0x0

For InfoType 80000007
CPUInfo[0] = 0x0
CPUInfo[1] = 0x0
CPUInfo[2] = 0x0
CPUInfo[3] = 0x0

For InfoType 80000008
CPUInfo[0] = 0x2028
CPUInfo[1] = 0x0
```

```
CPUInfo[2] = 0x0
CPUInfo[3] = 0x0


CPU String: GenuineIntel
Stepping ID = 1
Model = 3
Family = 15
CLFLUSH cache line size = 64

The following features are supported:
        SSE3 New Instructions
        MONITOR/MWAIT
        CPL Qualified Debug Store
        x87 FPU On Chip
        Virtual-8086 Mode Enhancement
        Debugging Extensions
        Page Size Extensions
        Time Stamp Counter
        RDMSR and WRMSR Support
        Physical Address Extensions
        Machine Check Exception
        CMPXCHG8B Instruction
        APIC On Chip
        SYSENTER and SYSEXIT
        Memory Type Range Registers
        PTE Global Bit
        Machine Check Architecture
        Conditional Move/Compare Instruction
        Page Attribute Table
        Page Size Extension
        CFLUSH Extension
        Debug Store
        Thermal Monitor and Clock Ctrl
        MMX Technology
        FXSAVE/FXRSTOR
        SSE Extensions
        SSE2 Extensions
        Self Snoop
        Hyper-threading Technology
        Thermal Monitor
        Pend. Brk. EN.

CPU Brand String:                    Genuine Intel(R) CPU 2.80GHz
Cache Line Size = 64
L2 Associativity = 8
Cache Size = 1024K
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__debugbreak

Microsoft Specific

Causes a breakpoint in your code, where the user will be prompted to run the debugger.

```
void __debugbreak();
```

Requirements

Intrinsic	Architecture
__debugbreak	x86, IPF, x64

Header file <intrin.h>

Remarks

The **__debugbreak** compiler intrinsic, similar to DebugBreak, is a portable Win32 way to cause a breakpoint.

Note
When compiling with **/clr**, a function containing **__debugbreak** will be compiled to MSIL. `asm int 3` causes a function to be compiled to native. For more information, see __asm.

For example:

```
main() {
   __debugbreak();
}
```

is similar to:

```
main() {
   __asm {
      int 3
   }
}
```

on an x86 computer.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

C++ Keywords

_disable

Microsoft Specific

Disables interrupts.

```
void _disable(void);
```

Requirements

Intrinsic	Architecture
_disable	x86, IPF, x64

Header file <intrin.h>

Remarks

_disable instructs the processor to clear the interrupt flag. On x86 systems, this function generates the Clear Interrupt Flag (**cli**) instruction. On IPF systems, this function generates the Reset System Mask (**rsm**) instruction with an argument that clears the interrupt control flag.

This function is only available in kernel mode. If used in user mode, a Privileged Instruction exception is thrown at run time.

On IPF platforms, this routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__dsrlz

Microsoft Specific

Generates the Serialize Data (**srlz.d**) instruction.

```
void __dsrlz(void);
```

Requirements

Intrinsic	Architecture
__dsrlz	IPF

Header file <intrin.h>

Remarks

For more information about the **srlz** instruction, see the Intel IPF documentation.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

__isrlz

Compiler Intrinsics

__emul, __emulu

Microsoft Specific

Performs multiplications that overflow what a 32-bit integer can hold.

```
__int64 __emul(
   int a,
   int b
);
unsigned __int64 __emulu(
   unsigned int a,
   unsigned int b
);
```

Parameters

[in] *a*

The first integer operand of the multiplication.

[in] *b*

The second integer operand of the multiplication.

Return Value

The result of the multiplication.

Requirements

Intrinsic	Architecture
__emul	x86, x64
__emulu	x86, x64

Header file <intrin.h>

Remarks

__emul takes two 32-bit signed values and returns the result of the multiplication as a 64-bit signed integer value.

__emulu takes two 32-bit unsigned integer values and returns the result of the multiplication as a 64-bit unsigned integer value.

Example

```
// emul.cpp
// compile with: /EHsc
// processor: x86, x64
#include <iostream>
#include <intrin.h>
using namespace std;

#pragma intrinsic(__emul)
#pragma intrinsic(__emulu)

int main()
{
   int a = -268435456;
   int b = 2;

   __int64 result = __emul(a, b);

   cout << a << " * " << b << " = " << result << endl;

   unsigned int ua = 0xFFFFFFFF; // Dec value: 4294967295
```

```
    unsigned int ub = 0xF000000;  // Dec value: 251658240

    unsigned __int64 uresult = __emulu(ua, ub);

    cout << ua << " * " << ub << " = " << uresult << endl;

  }
```

Output

```
-268435456 * 2 = -536870912
4294967295 * 251658240 = 1080863910317260800
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_enable

Microsoft Specific

Enables interrupts.

```
void _enable(void);
```

Requirements

Intrinsic	Architecture
_enable	x86, IPF, x64

Header file <intrin.h>

Remarks

_enable instructs the processor to set the interrupt flag. On x86 systems, this function generates the Set Interrupt Flag (**sti**) instruction. On an IPF system, this instruction generates a Set System Mask (**ssm**) instruction with an argument that sets the interrupt control flag.

This function is only available in kernel mode. If used in user mode, a Privileged Instruction exception is thrown.

On IPF platforms, this routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__faststorefence

Microsoft Specific

Guarantees that every preceding store is globally visible before any subsequent store.

```
void __faststorefence();
```

Requirements

Intrinsic	Architecture
__faststorefence	x64

Header file <intrin.h>

Remarks

On the x64 platform, this routine generates an instruction that is a faster store fence than the **sfence** instruction. Use this intrinsic instead of _mm_sfence on the x64 platform.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__fc

Microsoft Specific

Generates the IPF Flush Cache (**fc**) instruction.

```
void __fc(
   __int64 location
);
```

Parameters

[in] *location*

The location to flush.

Requirements

Intrinsic	Architecture
__fc	IPF

Header file <intrin.h>

Remarks

This operation causes the cache to be flushed at the location specified.

The location must be 32-bit aligned.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__fci

Microsoft Specific

Emits the IPF **fc.i** instruction (Flush cache with instruction cache coherent).

```
void __fci(
   void * Address
);
```

Parameters
[in] *Address*

A pointer to the address to flush.

Requirements

Intrinsic	Architecture
__fci	IPF

Header file <intrin.h>

Remarks

See the Intel IPF documentation for more information on the **fc.i** instruction.

This routine is only available as an intrinsic.

END Microsoft Specific
See Also
Reference
Compiler Intrinsics

__fclrf

Microsoft Specific

Generates the **fclrf.s0** instruction, which clears the floating-point flags in the main FP status field.

```
void __fclrf(void);
```

Requirements

Intrinsic	Architecture
__fclrf	IPF

Header file <intrin.h>

Remarks

For more information about the **fclrf** instruction, see the Intel IPF documentation.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

__fsetc

Compiler Intrinsics

__flushrs

Microsoft Specific

Generates the IPF Register Stack (**flushrs**) instruction.

```
void __flushrs(
   __int64 location
);
```

Parameters

[in] *location*

The location to flush.

Requirements

Intrinsic	Architecture
__flushrs	IPF

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__fsetc

Microsoft Specific

Generates the **fsetc.s0** instruction, which changes the control flags in the main status field.

```
void __fsetc(
   int amask,
   int omask
)
```

Parameters

[in] *amask*

The value to perform an **AND** operation with the existing control set.

[in] *omask*

The value to perform an **OR** operation with the existing control set.

Requirements

Intrinsic	Architecture
__fsetc	IPF

Header file <intrin.h>

Remarks

The new value of the status field will be the existing value AND the *amask* argument OR the *omask* argument.

For more information about the **fsetc** instruction, see the Intel documentation.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

__fsetc

Compiler Intrinsics

__fwb

Microsoft Specific

Generates the **fwb** instruction, which is a hint to the processor to flush the processor's write buffers.

```
void __fwb(void);
```

Requirements

Intrinsic	Architecture
__fwb	IPF

Header file <intrin.h>

Remarks

For more information about the **fwb** instruction, see the Intel IPF documentation.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

__fsetc

Compiler Intrinsics

__getcallerseflags

Microsoft Specific

Returns the EFLAGS value from the caller's context.

```
unsigned int __getcallerseflags(void);
```

Return Value

EFLAGS value from the caller's context.

Requirements

Intrinsic	Architecture
__getcallerseflags	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

Example

```
// getcallerseflags.cpp
// processor: x86, x64

#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__getcallerseflags)

unsigned int g()
{
    unsigned int EFLAGS = __getcallerseflags();
    printf_s("EFLAGS 0x%x\n", EFLAGS);
    return EFLAGS;
}
unsigned int f()
{
    return g();
}

int main()
{
    unsigned int i;
    i = f();
    i = g();
    return 0;
}
```

Sample Output

```
EFLAGS 0x202
EFLAGS 0x206
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__getPSP

Microsoft Specific

Returns the value of the stack pointer for the parent stack frame at the point of the call to the function associated with the current stack frame.

```
unsigned __int64 __getPSP(void);
```

Return Value

The value of the stack pointer when the calling function was first entered.

Requirements

Intrinsic	Architecture
__getPSP	IPF

Header file <intrin.h>

Remarks

If called within a function but outside of a catch or finally block, it returns the parent's stack pointer at the point of the parent's call to the function.

If called within a catch or finally block, it returns the original stack pointer for the function in which the catch or finally block resides.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__getCFS

Microsoft Specific

Gets the current frame state.

```
unsigned __int64 __getCFS(void);
```

Return Value

Returns the first 18 bits of the current frame marker (CFM).

Requirements

Intrinsic	Architecture
__getCFS	IPF

Header file <intrin.h>

Remarks

Bits 0 to 6 give the size of the current stack frame. Bits 7 to 13 give the size of the locals portion of the stack frame. Bits 14 to 17 give the size of the rotating portion of the stack frame.

This routine is only available as an intrinsic.

Example

```
// getCFS.c
// processor: IPF
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic (__getCFS)

#define SIZE_OF_FRAME_MASK 127
#define SIZE_OF_LOCALS_MASK 16256
#define SIZE_OF_ROT_MASK 245760

int main()
{
    int size_of_frame, size_of_locals, size_of_rot;
    unsigned __int64 cfs;

    cfs = __getCFS();
    printf_s("Current Frame State: 0x%x\n", cfs);

    size_of_frame = cfs & SIZE_OF_FRAME_MASK;
    size_of_locals = (cfs & SIZE_OF_LOCALS_MASK) >> 7;
    size_of_rot = (cfs & SIZE_OF_ROT_MASK) >> 13;
    printf_s("Size of current stack frame: %d\n",
             size_of_frame);
    printf_s("Size of locals portion of current stack frame: %d\n",
             size_of_locals);
    printf_s("Size of rotating portion of current stack frame: %d\n",
             size_of_rot);
   return 0;
}
```

Output

```
Current Frame State: 0x185
Size of current stack frame: 5
Size of locals portion of current stack frame: 3
Size of rotating portion of current stack frame: 0
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__getReg

Microsoft Specific

Gets the contents of a register specified with a constant that are the same values as declared in the header file `intrin.h`.

```
unsigned __int64 __getReg(
   int Register
);
```

Parameters

[in] *Register*

A constant specifying the desired processor register.

Return Value

The contents of the register indicated.

Requirements

Intrinsic	Architecture
__getReg	IPF

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic. Not all registers are readable. If you call this with an invalid register, a compile time error results.

Example

```
// getReg.cpp
// processor: IPF
#include <cstdio>
#include <stdio.h>
#include <intrin.h>
#define __REG_IA64_IntR1 1025

#pragma intrinsic(__getReg)

int main()
{
    unsigned __int64 reg;

    reg = __getReg(__REG_IA64_IntR1);
    printf_s("%I64d\n", reg);

    return 0;
}
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_InterlockedAdd Intrinsic Functions

Microsoft Specific

Perform an atomic addition, which ensures that the operation completes successfully when multiple threads have access to a shared variable.

```
long _InterlockedAdd(
   long volatile * Addend,
   long Value
);
long _InterlockedAdd_acq(
   long volatile * Addend,
   long Value
);
long _InterlockedAdd_rel(
   long volatile * Addend,
   long Value
);
__int64 _InterlockedAdd64(
   __int64 volatile * Addend,
   __int64 Value
);
__int64 _InterlockedAdd64_acq(
   __int64 volatile * Addend,
   __int64 Value
);
__int64 _InterlockedAdd64_rel(
   __int64 volatile * Addend,
   __int64 Value
);
```

Parameters

[in, out] *Addend*

Pointer to the integer to be added to; replaced by the result of the addition.

[in] *Value*

The value to add.

Return Value

Both functions return the result of the addition.

Requirements

Intrinsic	Architecture
_InterlockedAdd	IPF
_InterlockedAdd_acq	IPF
_InterlockedAdd_rel	IPF
_InterlockedAdd64	IPF
_InterlockedAdd64_acq	IPF
_InterlockedAdd64_rel	IPF

Header file <intrin.h>

Remarks

The versions of these functions with the `_acq` or `_rel` suffixes perform an interlocked addition following acquire or release semantics. Acquire semantics means that the result of the operation are made visible to all threads and processors before any subsequent memory reads and writes. Acquire is useful when entering a critical section. Release semantics means that all memory reads and writes are forced to be made visible to all threads and processors before the result of the operation is made visible itself. Release is useful when leaving a critical section.

In Visual C++ 2005, these functions behave as read-write memory barriers. For more information, see _ReadWriteBarrier.

These routines are only available as intrinsics.

Example

```
// interlockedadd.cpp
// Compile with: /Oi /EHsc
// processor: IPF
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_InterlockedAdd)

int main()
{
        long data1 = 0xFF00FF00;
        long data2 = 0x00FF0000;
        long retval;
        retval = _InterlockedAdd(&data1, data2);
        printf("0x%x 0x%x 0x%x", data1, data2, retval);
}
```

```
// interlockedadd64.cpp
// compile with: /Oi /EHsc
// processor: IPF
#include <iostream>
#include <intrin.h>
using namespace std;

#pragma intrinsic(_InterlockedAdd64)

int main()
{
        __int64 data1 = 0x0000FF0000000000;
        __int64 data2 = 0x00FF0000FFFFFFFF;
        __int64 retval;
        cout << hex << data1 << " + " << data2 << " = " ;
        retval = _InterlockedAdd64(&data1, data2);
        cout << data1 << endl;
        cout << "Return value: " << retval << endl;
}
```

Output

```
0xffffff00 0xff0000 0xffffff00
```

Output

```
ff0000000000 + ff0000ffffffff = ffff00ffffffff
Return value: ffff00ffffffff
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_InterlockedAddLargeStatistic

Microsoft Specific

Performs an interlocked addition in which the first operand is a 64-bit value.

```
long _InterlockedAddLargeStatistic(
   __int64 volatile * Addend,
   long Value
);
```

Parameters

[in, out] *Addend*

A pointer to the first operand to the add operation. The value pointed to is replaced by the result of the addition.

[in] *Value*

The second operand; value to add to the first operand.

Return Value

The value of the second operand.

Requirements

Intrinsic	Architecture
_InterlockedAddLargeStatistic	x86

Header file <intrin.h>

Remarks

This intrinsic is not atomic because it is implemented as two separate locked instructions. An atomic 64-bit read that occurs on another thread during the execution of this intrinsic could result in an inconsistent value being read.

In Visual C++ 2005, this function behaves as a read-write barrier. For more information, see _ReadWriteBarrier.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_InterlockedAnd, _InterlockedAnd64

Microsoft Specific

Used to perform an atomic AND operation on a variable shared by multiple threads.

```
long _InterlockedAnd(
   long volatile * value,
   long mask
);
long _InterlockedAnd_acq(
   long volatile * value,
   long mask
);
long _InterlockedAnd_rel(
   long volatile * value,
   long mask
);
char _InterlockedAnd8(
   char volatile * value,
   char mask
);
char _InterlockedAnd8_acq(
   char volatile * value,
   char mask
);
char _InterlockedAnd8_rel(
   char volatile * value,
   char mask
);
short _InterlockedAnd16(
   short volatile * value,
   short mask
);
short _InterlockedAnd16_acq(
   short volatile * value,
   short mask
);
short _InterlockedAnd16_rel(
   short volatile * value,
   short mask
);
__int64 _InterlockedAnd64(
   __int64 volatile* value,
   __int64 mask
);
__int64 _InterlockedAnd64_acq(
   __int64 volatile* value,
   __int64 mask
);
__int64 _InterlockedAnd64_rel(
   __int64 volatile* value,
   __int64 mask
);
```

Parameters

[in, out] *value*

A pointer to the first operand, to be replaced by the result.

[in] *mask*

The second operand.

Return Value

The original value of the first operand.

Requirements

Intrinsic	Architecture
_InterlockedAnd	x86, IPF, x64
_InterlockedAnd_acq	IPF
_InterlockedAnd_rel	IPF
_InterlockedAnd8	x86, IPF, x64
_InterlockedAnd8_acq	IPF
_InterlockedAnd8_rel	IPF
_InterlockedAnd16	x86, IPF, x64
_InterlockedAnd16_acq	IPF
_InterlockedAnd16_rel	IPF
_InterlockedAnd64	IPF, x64
_InterlockedAnd64_acq	IPF
_InterlockedAnd64_rel	IPF

Header file <intrin.h>

Remarks

The number in the name of each function specifies the bit size of the arguments.

In Visual C++ 2005, these functions behave as read-write memory barriers. For more information, see _ReadWriteBarrier.

The IPF-specific **_InterlockedAnd_acq**, **_InterlockedAnd8_acq**, **_InterlockedAnd16_acq**, and **_InterlockedAnd64_acq** intrinsic functions are the same as the corresponding functions without the **acq** suffix except that the operation is performed with acquire semantics, which is useful when entering a critical section.

The **_InterlockedAnd_rel**, **_InterlockedAnd8_rel**, **_InterlockedAnd16_rel**, and **_InterlockedAnd64_rel** intrinsic functions are the same as the corresponding functions without the **rel** suffix except that the operation is performed with release semantics, which is useful when leaving a critical section.

Example

```
// InterlockedAnd.cpp
// Compile with: /Oi
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_InterlockedAnd)

int main()
{
        long data1 = 0xFF00FF00;
        long data2 = 0x00FFFF00;
        long retval;
        retval = _InterlockedAnd(&data1, data2);
        printf_s("0x%x 0x%x 0x%x", data1, data2, retval);
}
```

Output

```
0xff00 0xffff00 0xff00ff00
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_interlockedbittestandreset, _interlockedbittestandreset64

Microsoft Specific

Generate the **lock_btr** instruction, which examines bit *b* of the address *a* and returns its current value.

```
unsigned char _interlockedbittestandreset(
   long *a,
   long b
);
unsigned char _interlockedbittestandreset64(
   __int64 *a,
   __int64 b
);
```

Parameters

[in] *a*

A pointer to the memory to examine.

[in] *b*

The bit position to test.

Return Value

The bit at the position specified.

Requirements

Intrinsic	Architecture
_interlockedbittestandreset	x86, x64
_interlockedbittestandreset64	x64

Header file <intrin.h>

Remarks

As a result of the **lock_btr** instruction, the bit is set to zero. The operation is atomic.

In Visual C++ 2005, these functions behave as read-write memory barriers. For more information, see _ReadWriteBarrier.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_interlockedbittestandset, _interlockedbittestandset64

Microsoft Specific

Generate the **lock_bts** instruction, which examines bit *b* of the address *a* and returns its current value.

```
unsigned char _interlockedbittestandset(
   long *a,
   long b
);
unsigned char _interlockedbittestandset64(
   __int64 *a,
   __int64 b
);
```

Parameters

[in] *a*

A pointer to the memory to examine.

[in] *b*

The bit position to test.

Return Value

The bit at the position specified.

Requirements

Intrinsic	Architecture
_interlockedbittestandset	x86, x64
_interlockedbittestandset64	x64

Header file <intrin.h>

Remarks

As a result of the **lock_bts** instruction, the bit is set to 1. The operation is atomic.

In Visual C++ 2005, these functions behave as read-write memory barriers. For more information, see _ReadWriteBarrier.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_InterlockedCompareExchange Intrinsic Functions

Microsoft Specific

Provides compiler intrinsic support for the Win32 Platform SDK InterlockedCompareExchange function.

```
long _InterlockedCompareExchange(
   long volatile * Destination,
   long Exchange,
   long Comperand
);
long _InterlockedCompareExchange_acq(
   long volatile * Destination,
   long Exchange,
   long Comperand
);
long _InterlockedCompareExchange_rel(
   long volatile * Destination,
   long Exchange,
   long Comperand
);
short _InterlockedCompareExchange16(
   short volatile * Destination,
   short Exchange,
   short Comperand
);
short _InterlockedCompareExchange16_acq(
   short volatile * Destination,
   short Exchange,
   short Comperand
);
short _InterlockedCompareExchange16_rel(
   short volatile * Destination,
   short Exchange,
   short Comperand
);
__int64 _InterlockedCompareExchange64(
   __int64 volatile * Destination,
   __int64 Exchange,
   __int64 Comperand
);
__int64 _InterlockedCompareExchange64_acq(
   __int64 volatile * Destination,
   __int64 Exchange,
   __int64 Comperand
);
__int64 _InterlockedCompareExchange64_rel(
   __int64 volatile * Destination,
   __int64 Exchange,
   __int64 Comperand
);
```

Parameters

[in, out] *Destination*

Pointer to the destination value. The sign is ignored.

[in] *Exchange*

Exchange value. The sign is ignored.

[in] *Comperand*

Value to compare to destination. The sign is ignored.

Return Value

The return value is the initial value of the *Destination* pointer.

Requirements

Intrinsic	Architecture
_InterlockedCompareExchange	x86, IPF, x64
_InterlockedCompareExchange_acq	IPF
_InterlockedCompareExchange_rel	IPF
_InterlockedCompareExchange16	x86, IPF, x64
_InterlockedCompareExchange16_acq	IPF
_InterlockedCompareExchange16_rel	IPF
_InterlockedCompareExchange64	x86 (Pentium)*, IPF, x64
_InterlockedCompareExchange64_acq	IPF
_InterlockedCompareExchange64_rel	IPF

Note: _InterlockedCompareExchange64 is available on x86 systems running on any Pentium architecture; it is not available on 386 or 486 architectures.

Header file <intrin.h>

Remarks

Note that to generate the intrinsic expansion, you need to use /Oi. **/Oi** is implied with **/O2**.

To declare one of the interlocked functions for use as an intrinsic, the function must be declared with the leading underscore and the new function must appear in a **#pragma** intrinsic statement. For convenience, the intrinsic versions of the functions can be declared in a **#define** statement to appear in the source code without the leading underscore.

_InterlockedCompareExchange performs an atomic comparison of the *Destination* value with the *Comperand* value. If the *Destination* value is equal to the *Comperand* value, the *Exchange* value is stored in the address specified by *Destination*. Otherwise, no operation is performed.

For a sample of how to use **_InterlockedCompareExchange**, see _InterlockedDecrement.

There are several variations on **_InterlockedCompareExchange** that vary based on the data types they involve and whether processor-specific acquire or release semantics is used.

While the **_InterlockedCompareExchange** function operates on long integer values, **_InterlockedCompareExchange16** operates on short integer values and **_InterlockedCompareExchange64** operates on 64-bit integer values. Because **_InterlockedCompareExchange64** uses the **cmpxchg8b** instruction, it is not available on pre-Pentium processors, such as the 486.

The IPF-specific **_InterlockedCompareExchange_acq**, **_InterlockedCompareExchange16_acq**, and **_InterlockedCompareExchange64_acq** intrinsic functions are the same as the corresponding functions without the **acq** suffix except that the operation is performed with acquire semantics, which is useful when entering a critical section.

The **_InterlockedCompareExchange_rel**, **_InterlockedCompareExchange16_rel**, and **_InterlockedCompareExchange64_rel** intrinsic functions are the same as the corresponding functions without the **rel** suffix except that the operation is performed with release semantics, which is useful when leaving a critical section.

In Visual C++ 2005, these functions behave as read-write memory barriers. For more information, see _ReadWriteBarrier.

These routines are only available as intrinsics.

Example

In the following example, **_InterlockedCompareExchange** is used for simple low-level thread synchronization. The approach has its limitations as a basis for multithreaded programming; it is presented to illustrate the typical use of the interlocked

intrinsics. For best results, use the Windows API. For further information about multithreaded programming, see Writing a Multithreaded Win32 Program.

```
// intrinExample.cpp
// compile with: /EHsc /O2
// Simple example of using _Interlocked* intrinsics to
// do manual synchronization
//
// Add [-DSKIP_LOCKING] to the command line to disable
// the locking. This will cause the threads to execute out
// of sequence.

#include "windows.h"

#include <iostream>
#include <queue>
#include <intrin.h>

using namespace std;

// ---------------------------------------------------------------------

// if defined, will not do any locking on shared data
//#define SKIP_LOCKING

// A common way of locking using _InterlockedCompareExchange.
// Please refer to other sources for a discussion of the many issues
// involved. For example, this particular locking scheme performs well
// when lock contention is low, as the while loop overhead is small and
// locks are acquired very quickly, but degrades as many callers want
// the lock and most threads are doing a lot of interlocked spinning.
// There are also no guarantees that a caller will ever acquire the
// lock.
namespace MyInterlockedIntrinsicLock
{
    typedef unsigned LOCK, *PLOCK;

#pragma intrinsic(_InterlockedCompareExchange, _InterlockedExchange)

    enum {LOCK_IS_FREE = 0, LOCK_IS_TAKEN = 1};

    void Lock(PLOCK pl)
    {
#if !defined(SKIP_LOCKING)
        // If *pl == LOCK_IS_FREE, it is set to LOCK_IS_TAKEN
        // atomically, so only 1 caller gets the lock.
        // If *pl == LOCK_IS_TAKEN,
        // the result is LOCK_IS_TAKEN, and the while loop keeps spinning.
        while (_InterlockedCompareExchange((long *)pl,
                                           LOCK_IS_TAKEN, // exchange
                                           LOCK_IS_FREE)  // comparand
               == LOCK_IS_TAKEN)
        {
            // spin!
            // call __yield() here on the IPF architecture to improve
            // performance.
        }
        // This will also work.
        //while (_InterlockedExchange(pl, LOCK_IS_TAKEN) ==
        //                            LOCK_IS_TAKEN)
        //{
        //    // spin!
        //}

        // At this point, the lock is acquired.
#endif
    }
```

```
    void Unlock(PLOCK pl) {
#if !defined(SKIP_LOCKING)
        _InterlockedExchange((long *)pl, LOCK_IS_FREE);
#endif
    }
}

// ----------------------------------------------------------------

// Data shared by threads

queue<int> SharedQueue;
MyInterlockedIntrinsicLock::LOCK SharedLock;
int TicketNumber;

// ----------------------------------------------------------------

DWORD WINAPI
ProducerThread(
    LPVOID unused
    )
{
    while (1) {
        // Acquire shared data. Enter critical section.
        MyInterlockedIntrinsicLock::Lock(&SharedLock);

        //cout << ">" << TicketNumber << endl;
        SharedQueue.push(TicketNumber++);

        // Release shared data. Leave critical section.
        MyInterlockedIntrinsicLock::Unlock(&SharedLock);

        Sleep(rand() % 20);
    }

    return 0;
}

DWORD WINAPI
ConsumerThread(
    LPVOID unused
    )
{
    while (1) {
        // Acquire shared data. Enter critical section
        MyInterlockedIntrinsicLock::Lock(&SharedLock);

        if (!SharedQueue.empty()) {
            int x = SharedQueue.front();
            cout << "<" << x << endl;
            SharedQueue.pop();
        }

        // Release shared data. Leave critical section
        MyInterlockedIntrinsicLock::Unlock(&SharedLock);

        Sleep(rand()%20);
    }
    return 0;
}


int main(
    void
    )
{
    const int timeoutTime = 500;
    int unused1, unused2;
```

```
    HANDLE threads[4];

    // The program creates 4 threads:
    // two producer threads adding to the queue
    // and two consumers taking data out and printing it.
    threads[0] = CreateThread(NULL,
                              0,
                              ProducerThread,
                              &unused1,
                              0,
                              (LPDWORD)&unused2);

    threads[1] = CreateThread(NULL,
                              0,
                              ConsumerThread,
                              &unused1,
                              0,
                              (LPDWORD)&unused2);

    threads[2] = CreateThread(NULL,
                              0,
                              ProducerThread,
                              &unused1,
                              0,
                              (LPDWORD)&unused2);

    threads[3] = CreateThread(NULL,
                              0,
                              ConsumerThread,
                              &unused1,
                              0,
                              (LPDWORD)&unused2);

    WaitForMultipleObjects(4, threads, TRUE, timeoutTime);

    return 0;
}
```

Output

```
<0
<1
<2
<3
<4
<5
<6
<7
<8
<9
<10
<11
<12
<13
<14
<15
<16
<17
<18
<19
<20
<21
<22
<23
<24
<25
<26
<27
<28
<29
```

END Microsoft Specific

See Also

Reference

_InterlockedCompare64Exchange128, _InterlockedCompare64Exchange128_acq, _InterlockedCompare64Exchange128_rel

_InterlockedCompareExchangePointer Intrinsic Functions

Compiler Intrinsics

C++ Keywords

_InterlockedCompare64Exchange128, _InterlockedCompare64Exchange128_acq, _InterlockedCompare64Exchange128_rel

Microsoft Specific

Provides atomic 128-bit compiler intrinsic support for the Win32 Platform SDK InterlockedCompareExchange function.

```
__int64 _InterlockedCompare64Exchange128(
   __int64 volatile * Destination,
   __int64 ExchangeHigh,
   __int64 ExchangeLow,
   __int64 Comperand
);
__int64 _InterlockedCompare64Exchange128_acq(
   __int64 volatile * Destination,
   __int64 ExchangeHigh,
   __int64 ExchangeLow,
   __int64 Comperand
);
__int64 _InterlockedCompare64Exchange128_rel(
   __int64 volatile * Destination,
   __int64 ExchangeHigh,
   __int64 ExchangeLow,
   __int64 Comperand
);
```

Parameters

[in, out] *Destination*

Pointer to the 128-bit Destination value.

[in] *ExchangeHigh*

The most significant 64 bit of the Exchange value.

[in] *ExchangeLow*

The least significant 64 bit of the Exchange value.

[in] *Comperand*

Value to compare to destination.

Return Value

The return value is the initial least significant 64-bit value of the destination.

Requirements

_InterlockedCompare64Exchange128

IPF, x64

_InterlockedCompare64Exchange128_acq

IPF, x64

_InterlockedCompare64Exchange128_rel

IPF, x64

Header file <intrin.h>

Remarks

Note that the intrinsic form of this function has a leading underscore in the name. Also note that these functions are always

expanded as intrinsics. That means you can use /Oi, but it is not required. **/Oi** implied with **/O2**.

Optionally, you can declare these functions for use as an intrinsic. You must declare them with the leading underscore, and the function name might also appear in a **#pragma** intrinsic statement. It is strongly recommended that you declare these functions to let the compiler catch parameter type mismatches.

These functions are not general-purpose 128-bit intrinsics. These functions should only be used if you really need 128-bit atomic compare-exchange operations; otherwise use the corresponding 64-bit versions for better performance.

The **_InterlockedCompare64Exchange128** function performs an atomic comparison of the least significant 64 bits of the *Destination* value with the *Comperand* value. If the *Destination* value is equal to the *Comperand* value, the 128-bit *Exchange* value is stored in the address specified by *Destination*. Otherwise, no operation is performed. The Destination pointer should point to a 16-byte aligned memory location.

The **_InterlockedCompare64Exchange128_acq** intrinsic function is the same as the corresponding function without the **_acq** suffix except that the operation is performed with acquire semantics.

The **_InterlockedCompare64Exchange128_rel** intrinsic function is the same as the corresponding function without the **_rel** suffix except that the operation is performed with release semantics.

On the x64 platform, the **_acq** and **_rel** forms are the same as the corresponding function without the **_acq** or **_rel** suffixes. These forms are supported for source code compatibility with the IPF platform.

The x64 platform does not have hardware support for the above described 128-bit atomic operations. The intrinsic function generates code that emulates the IPF semantics.

In Visual C++ 2005, these functions behave as read-write barriers. For more information, see _ReadWriteBarrier.

See Also
Reference
Compiler Intrinsics
_InterlockedCompareExchange Intrinsic Functions

_InterlockedCompareExchangePointer Intrinsic Functions

Microsoft Specific

Provides compiler intrinsic support for the Win32 Platform SDK InterlockedCompareExchangePointer function.

```
void * _InterlockedCompareExchangePointer (
   void * volatile * Destination,
   void * Exchange,
   void * Comperand
);
void * _InterlockedCompareExchangePointer_acq (
   void * volatile * Destination,
   void * Exchange,
   void * Comperand
);
long _InterlockedCompareExchangePointer_rel (
   void * volatile * Destination,
   void * Exchange,
   void * Comperand
);
```

Parameters

[in, out] *Destination*

Pointer to a pointer to the destination value. The sign is ignored.

[in] *Exchange*

Exchange pointer. The sign is ignored.

[in] *Comperand*

Pointer to compare to destination. The sign is ignored.

Return Value

The return value is the initial value of the destination.

Requirements

Intrinsic	Architecture
_InterlockedCompareExchangePointer	x86, IPF, x64
_InterlockedCompareExchangePointer_acq	IPF
_InterlockedCompareExchangePointer_rel	IPF

Note On the x86 architecture, **_InterlockedCompareExchangePointer** is a macro that calls **_InterlockedCompareExchange**.

Header file <intrin.h>

Remarks

Note that to generate the intrinsic expansion, you need to use /Oi. **/Oi** is implied with **/O2**.

To declare one of the interlocked functions for use as an intrinsic, the function must be declared with the leading underscore and the new function must appear in a **#pragma** intrinsic statement. For convenience, the intrinsic versions of the functions can be declared in a **#define** statement to appear in the source code without the leading underscore.

_InterlockedCompareExchangePointer performs an atomic comparison of the *Destination* address with the *Comperand* address. If the *Destination* address is equal to the *Comperand* address, the *Exchange* address is stored in the address specified by *Destination*. Otherwise, no operation is performed.

For a example of how to use **_InterlockedCompareExchangePointer**, see _InterlockedDecrement.

The **_InterlockedCompareExchangePointer_acq** intrinsic function is the same as the corresponding function without the **acq** suffix except that the operation is performed with acquire semantics, which is useful when entering a critical section.

The **_InterlockedCompareExchangePointer_rel** intrinsic function are the same as the corresponding function without the **rel** suffix except that the operation is performed with release semantics, which is useful when leaving a critical section.

In Visual C++ 2005, these functions behave as read-write memory barriers. For more information, see _ReadWriteBarrier.

These routines are only available as intrinsics.

END Microsoft Specific

See Also

Reference

_InterlockedCompareExchangePointer

Compiler Intrinsics

C++ Keywords

_InterlockedDecrement Intrinsic Functions

Microsoft Specific

Provides compiler intrinsic support for the Win32 Platform SDK InterlockedDecrement function.

```
long _InterlockedDecrement(
   long * lpAddend
);
long _InterlockedDecrement_acq(
   long * lpAddend
);
long _InterlockedDecrement_acq(
   long * lpAddend
);
short _InterlockedDecrement16(
   short * lpAddend
);
short _InterlockedDecrement16_acq(
   short * lpAddend
);
short _InterlockedDecrement16_rel(
   short * lpAddend
);
__int64 _InterlockedDecrement64(
   __int64 * lpAddend
);
__int64 _InterlockedDecrement64_acq(
   __int64 * lpAddend
);
__int64 _InterlockedDecrement64_rel(
   __int64 * lpAddend
);
```

Parameters

[in, out] *lpAddend*

Pointer to the variable to be decremented.

Return Value

Windows 98, Windows NT 4.0, and later: The return value is the resulting decremented value.

Windows 95, Windows NT 3.51, and earlier: If the result of the operation is zero, the return value is zero. If the result of the operation is less than zero, the return value is negative, but it is not necessarily equal to the result. If the result of the operation is greater than zero, the return value is positive, but it is not necessarily equal to the result.

Requirements

Intrinsic	Architecture
_InterlockedDecrement	x86, IPF, AMD64
_InterlockedDecrement_acq	IPF
_InterlockedDecrement_rel	IPF
_InterlockedDecrement16	x86, IPF, AMD64
_InterlockedDecrement16_acq	IPF
_InterlockedDecrement16_rel	IPF

_InterlockedDecrement64	IPF, AMD64
_InterlockedDecrement64_acq	IPF
_InterlockedDecrement64_rel	IPF

Header file <intrin.h>

Remarks

There are several variations on **_InterlockedDecrement** that vary based on the data types they involve and whether processor-specific acquire or release semantics is used.

While the **_InterlockedDecrement** function operates on 32-bit integer values, **_InterlockedDecrement16** operates on 16-bit integer values and **_InterlockedDecrement64** operates on 64-bit integer values.

The **_InterlockedDecrement_acq**, **_InterlockedDecrement16_acq** and **_InterlockedDecrement64_acq** intrinsic functions are the same as the corresponding functions without the **_acq** suffix except that the operation is performed with acquire semantics, which is useful when entering a critical section.

The **_InterlockedDecrement_rel**, **_InterlockedDecrement16_rel** and **_InterlockedDecrement64_rel** intrinsic function are the same as the corresponding functions without the **_rel** suffix except that the operation is performed with release semantics, which is useful when leaving a critical section.

The variable pointed to by the *lpAddend* parameter must be aligned on a 32-bit boundary; otherwise, this function fails on multiprocessor x86 systems and any non-x86 systems.

In Visual C++ 2005, these functions behave as read-write memory barriers. For more information, see _ReadWriteBarrier.

These routines are only available as intrinsics.

Example

```
// compiler_intrinsics_interlocked.cpp
// compile with: /Oi
#include <cstdlib>
#include <cstdio>
#include <process.h>
#include <windows.h>

// To declare an interlocked function for use as an intrinsic,
// include intrin.h and put the function in a #pragma intrinsic
// statement.
#include <intrin.h>

#pragma intrinsic (_InterlockedIncrement)

// Data to protect with the interlocked functions.
volatile LONG data = 1;

void __cdecl SimpleThread(void* pParam);

const int THREAD_COUNT = 6;

int main() {
   DWORD num;
   HANDLE threads[THREAD_COUNT];
   int args[THREAD_COUNT];
   int i;

   for (i = 0; i < THREAD_COUNT; i++) {
     args[i] = i + 1;
     threads[i] = reinterpret_cast<HANDLE>(_beginthread(SimpleThread, 0,
                           args + i));
      if (threads[i] == reinterpret_cast<HANDLE>(-1))
         // error creating threads
         break;
```

```
    }

    WaitForMultipleObjects(i, threads, true, INFINITE);
}

// Code for our simple thread
void __cdecl SimpleThread(void* pParam) {
    int threadNum = *((int*)pParam);
    int counter;
    int time = rand() % 500;

    while (data < 100) {
        if (data < 100) {
            _InterlockedIncrement(&data);
            printf_s("Thread %d: %d\n", threadNum, data);
        }

        Sleep(time);   // wait up to half of a second
    }

    printf_s("Thread %d complete: %d\n", threadNum, data);
}
```

See Also

Reference

Compiler Intrinsics

C++ Keywords

_InterlockedExchange Intrinsic Functions

Microsoft Specific

Provide compiler intrinsic support for the Win32 Platform SDK InterlockedExchange function.

```
long _InterlockedExchange(
   long * Target,
   long Value
);
long _InterlockedExchange_acq(
   long * Target,
   long Value
);
__int64 _InterlockedExchange64(
   __int64 * Target,
   __int64 Value
);
__int64 _InterlockedExchange64_acq(
   __int64 * Target,
   __int64 Value
);
```

Parameters

[in, out] *Target*

Pointer to the value to be exchanged. The function sets this variable to *Value* and returns its prior value.

[in] *Value*

Value to be exchanged with the value pointed to by *Target*.

Return Value

Returns the initial value pointed to by *Target*.

Requirements

Intrinsic	Architecture
_InterlockedExchange	x86, IPF, x64
_InterlockedExchange_acq	IPF
_InterlockedExchange64	IPF, x64
_InterlockedExchange64_acq	IPF

Header file <intrin.h>

Remarks

There are several variations on **_InterlockedExchange** that vary based on the data types they involve and whether processor-specific acquire or release semantics is used.

While the **_InterlockedExchange** function operates on 32-bit integer values, **_InterlockedExchange64** operates on 64-bit integer values.

The **_InterlockedExchange_acq** and **_InterlockedExchange64_acq** intrinsic functions are the same as the corresponding functions without the **_acq** suffix except that the operation is performed with acquire semantics, which is useful when entering a critical section.

There is no version of this function that uses release semantics.

In Visual C++ 2005, these functions behave as read-write memory barriers. For more information, see _ReadWriteBarrier.

These routines are only available as intrinsics.

Example

For a sample of how to use **_InterlockedExchange**, see _InterlockedDecrement.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

C++ Keywords

_InterlockedExchangeAdd Intrinsic Functions

Microsoft Specific

Provide compiler intrinsic support for the Win32 Platform SDK _InterlockedExchangeAdd Intrinsic Functions function.

```
long _InterlockedExchangeAdd(
   long volatile * Addend,
   long Value
);
long _InterlockedExchangeAdd_acq(
   long volatile * Addend,
   long Value
);
long _InterlockedExchangeAdd_rel(
   long volatile * Addend,
   long Value
);
__int64 _InterlockedExchangeAdd64(
   __int64 volatile * Addend,
   __int64 Value
);
__int64 _InterlockedExchangeAdd64_acq(
   __int64 volatile * Addend,
   __int64 Value
);
__int64 _InterlockedExchangeAdd64_rel(
   __int64 volatile * Addend,
   __int64 Value
);
```

Parameters

[in, out] *Addend*

The value to be added to; replaced by the result of the addition.

[in] *Value*

The value to add.

Return Value

The return value is the initial value of the variable pointed to by the *Addend* parameter.

Requirements

Intrinsic	Architecture
_InterlockedExchangeAdd	x86, IPF, x64
_InterlockedExchangeAdd_acq	IPF
_InterlockedExchangeAdd_rel	IPF
_InterlockedExchangeAdd64	IPF, x64
_InterlockedExchangeAdd64_acq	IPF
_InterlockedExchangeAdd64_rel	IPF

Header file <intrin.h>

Remarks

There are several variations on **_InterlockedExchangeAdd** that vary based on the data types they involve and whether processor-specific acquire or release semantics is used.

While the **_InterlockedExchangeAdd** function operates on 32-bit integer values, **_InterlockedExchangeAdd64** operates on 64-bit integer values.

The **_InterlockedExchangeAdd_acq** and **_InterlockedExchangeAdd64_acq** intrinsic functions are the same as the corresponding functions without the **_acq** suffix except that the operation is performed with acquire semantics, which is useful when entering a critical section.

There is no version of this function that uses release semantics.

In Visual C++ 2005, these functions behave as read-write memory barriers. For more information, see _ReadWriteBarrier.

These routines are only available as intrinsics. Thus, they are intrinsic whether or not /Oi or #pragma intrinsic is used. It is not possible to use #pragma function on these intrinsics.

Example

For a sample of how to use **_InterlockedExchangeAdd**, see _InterlockedDecrement.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

C++ Keywords

_InterlockedExchangePointer Intrinsic Functions

Microsoft Specific

Perform an atomic exchange operation, which copies the address passed in as the second argument to the first and returns the original address of the first.

```
void * _InterlockedExchangePointer(
   void * volatile * Target,
   void * Value
);
void * _InterlockedExchangePointer_acq(
   void * volatile * Target,
   void * Value
);
```

Parameters

[in, out] *Target*

Pointer to the pointer to the value to exchange. The function sets the value to *Value* and returns its previous value.

[in] *Value*

Value to be exchanged with the value pointed to by *Target*.

Return Value

The function returns the initial value pointed to by *Target*.

Requirements

Intrinsic	Architecture
_InterlockedExchangePointer	IPF, x64
_InterlockedExchangePointer_acq	IPF

Note On the x86 architecture, **_InterlockedExchangePointer** is a macro that calls **_InterlockedExchange**.

Header file <intrin.h>

Remarks

On a 64-bit system, the parameters are 64 bits and must be aligned on 64-bit boundaries; otherwise, the function fails. On a 32-bit system, the parameters are 32 bits and must be aligned on 32-bit boundaries.

Use **_InterlockedExchangePointer_acq** if you need acquire semantics, such as at the beginning of a critical section.

There is no version of this function that uses release semantics.

In Visual C++ 2005, these functions behave as read-write memory barriers. For more information, see _ReadWriteBarrier.

These routines are only available as intrinsics.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_InterlockedIncrement Intrinsic Functions

Microsoft Specific

Provide compiler intrinsic support for the Win32 Platform SDK InterlockedIncrement function.

```
long _InterlockedIncrement(
   long * lpAddend
);
long _InterlockedIncrement_acq(
   long * lpAddend
);
long _InterlockedIncrement_rel(
   long * lpAddend
);
short _InterlockedIncrement16(
   short * lpAddend
);
short _InterlockedIncrement16_acq(
   short * lpAddend
);
short _InterlockedIncrement16_rel(
   short * lpAddend
);
__int64 _InterlockedIncrement64(
   __int64 * lpAddend
);
__int64 _InterlockedIncrement64_acq(
   __int64 * lpAddend
);
__int64 _InterlockedIncrement64_rel(
   __int64 * lpAddend
);
```

Parameters

[in, out] *lpAddend*

Pointer to the variable to be incremented.

Return Value

Windows 98, Windows NT 4.0 and later: The return value is the resulting incremented value.

Windows 95, Windows NT 3.51 and earlier: If the result of the operation is zero, the return value is zero. If the result of the operation is less than zero, the return value is negative, but it is not necessarily equal to the result. If the result of the operation is greater than zero, the return value is positive, but it is not necessarily equal to the result.

Requirements

Intrinsic	Architecture
_InterlockedIncrement	x86, IPF, x64
_InterlockedIncrement_acq	IPF
_InterlockedIncrement_rel	IPF
_InterlockedIncrement16	x86, IPF, x64
_InterlockedIncrement16_acq	IPF
_InterlockedIncrement16_rel	IPF

_InterlockedIncrement64	IPF, x64
_InterlockedIncrement64_acq	IPF
_InterlockedIncrement64_rel	IPF

Header file <intrin.h>

Remarks

There are several variations on **_InterlockedIncrement** that vary based on the data types they involve and whether processor-specific acquire or release semantics is used.

While the **_InterlockedIncrement** function operates on 32-bit integer values, **_InterlockedIncrement16** operates on 16-bit integer values and **_InterlockedIncrement64** operates on 64-bit integer values.

The **_InterlockedIncrement_acq**, **_InterlockedIncrement16_acq**, and **_InterlockedIncrement64_acq** intrinsic functions are the same as the corresponding functions without the **_acq** suffix except that the operation is performed with acquire semantics, which is useful when entering a critical section.

The **_InterlockedIncrement_rel**, **_InterlockedIncrement16_rel**, and **_InterlockedIncrement64_rel** intrinsic function are the same as the corresponding functions without the **_rel** suffix except that the operation is performed with release semantics, which is useful when leaving a critical section.

The variable pointed to by the *lpAddend* parameter must be aligned on a 32-bit boundary; otherwise, this function fails on multiprocessor x86 systems and any non-x86 systems.

In Visual C++ 2005, these functions behave as read-write memory barriers. For more information, see _ReadWriteBarrier.

The Win32 function is declared in `Wdm.h` or `Ntddk.h`.

These routines are only available as intrinsics.

Example

For a sample of how to use **_InterlockedIncrement**, see _InterlockedDecrement.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

C++ Keywords

_InterlockedOr, _InterlockedOr64

Microsoft Specific

Perform an atomic operation (in this case, the **OR** operation) on a variable shared by multiple threads.

```
long _InterlockedOr(
   long volatile * Value,
   long Mask
);
long _InterlockedOr_acq(
   long volatile * Value,
   long Mask
);
long _InterlockedOr_rel(
   long volatile * Value,
   long Mask
);
char _InterlockedOr8(
   char volatile * Value,
   long Mask
);
char _InterlockedOr8_acq(
   char volatile * Value,
   char Mask
);
char _InterlockedOr8_rel(
   char volatile * Value,
   char Mask
);
short _InterlockedOr16(
   short volatile * Value,
   short Mask
);
short _InterlockedOr16_acq(
   short volatile * Value,
   short Mask
);
short _InterlockedOr16_rel(
   short volatile * Value,
   short Mask
);
__int64 _InterlockedOr64(
   __int64 volatile * Value,
   __int64 Mask
);
__int64 _InterlockedOr64_acq(
   __int64 volatile * Value,
   __int64 Mask
);
__int64 _InterlockedOr64_rel(
   __int64 volatile * Value,
   __int64 Mask
);
```

Parameters

[in, out] *Value*

A pointer to the first operand, to be replaced by the result.

[in] *Mask*

The second operand.

Return Value

The original value pointed to by the first parameter.

Requirements

Intrinsic	Architecture
_InterlockedOr	x86, IPF, x64
_InterlockedOr_acq	IPF
_InterlockedOr_rel	IPF
_InterlockedOr8	x86, IPF, x64
_InterlockedOr8_acq	IPF
_InterlockedOr8_rel	IPF
_InterlockedOr16	x86, IPF, x64
_InterlockedOr16_acq	IPF
_InterlockedOr16_rel	IPF
_InterlockedOr64	IPF, x64
_InterlockedOr64_acq	IPF
_InterlockedOr64_rel	IPF

Header file <intrin.h>

Remarks

The number in the name of each function specifies the bit size of the arguments.

In Visual C++ 2005, these functions behave as read-write memory barriers. For more information, see _ReadWriteBarrier.

The IPF-specific **_InterlockedOr_acq**, **_InterlockedOr8_acq**, **_InterlockedOr16_acq**, and **_InterlockedOr64_acq** intrinsic functions are the same as the corresponding functions without the **acq** suffix except that the operation is performed with acquire semantics, which is useful when entering a critical section.

The **_InterlockedOr_rel**, **_InterlockedOr8_rel**, **_InterlockedOr16_rel**, and **_InterlockedOr64_rel** intrinsic functions are the same as the corresponding functions without the **rel** suffix except that the operation is performed with release semantics, which is useful when leaving a critical section.

Example

```
// _InterlockedOr.cpp
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_InterlockedOr)

int main()
{
        long data1 = 0xFF00FF00;
        long data2 = 0x00FFFF00;
        long retval;
        retval = _InterlockedOr(&data1, data2);
        printf_s("0x%x 0x%x 0x%x", data1, data2, retval);
}
```

Output

```
0xffffff00 0xffff00 0xff00ff00
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_InterlockedXor, _InterlockedXor64

Microsoft Specific

Used to perform an atomic operation (in this case, the exclusive or **XOR** operation) on a variable shared by multiple threads.

```
long _InterlockedXor(
   long volatile * Value,
   long Mask
);
long _InterlockedXor_acq(
   long volatile * Value,
   long Mask
);
long _InterlockedXor_rel(
   long volatile * Value,
   long Mask
);
char _InterlockedXor8(
   char volatile * Value,
   char Mask
);
char _InterlockedXor8_acq(
   char volatile * Value,
   char Mask
);
char _InterlockedXor8_rel(
   char volatile * Value,
   char Mask
);
short _InterlockedXor16(
   short volatile * Value,
   short Mask
);
short _InterlockedXor16_acq(
   short volatile * Value,
   short Mask
);
short _InterlockedXor16_rel(
   short volatile * Value,
   short Mask
);
__int64 _InterlockedXor64(
   __int64 volatile * Value,
   __int64 Mask
);
__int64 _InterlockedXor64_acq(
   __int64 volatile * Value,
   __int64 Mask
);
__int64 _InterlockedXor64_rel(
   __int64 volatile * Value,
   __int64 Mask
);
```

Parameters

[in, out] *Value*

A pointer to the first operand, to be replaced by the result.

[in] *Mask*

The second operand.

Return Value

The original value of the first operand.

Requirements

Intrinsic	Architecture
_InterlockedXor	x86, IPF, x64
_InterlockedXor_acq	IPF
_InterlockedXor_rel	IPF
_InterlockedXor8	x86, IPF, x64
_InterlockedXor8_acq	IPF
_InterlockedXor8_rel	IPF
_InterlockedXor16	x86, IPF, x64
_InterlockedXor16_acq	IPF
_InterlockedXor16_rel	IPF
_InterlockedXor64	IPF, x64
_InterlockedXor64_acq	IPF
_InterlockedXor64_rel	IPF

Header file <intrin.h>

Remarks

The number in the name of each function specifies the bit size of the arguments.

In Visual C++ 2005, these functions behave as read-write memory barriers. For more information, see _ReadWriteBarrier.

The IPF-specific **_InterlockedXor_acq**, **_InterlockedXor8_acq**, **_InterlockedXor16_acq**, and **_InterlockedXor64_acq** intrinsic functions are the same as the corresponding functions without the **acq** suffix except that the operation is performed with acquire semantics, which is useful when entering a critical section.

The **_InterlockedXor_rel**, **_InterlockedXor8_rel**, **_InterlockedXor16_rel**, and **_InterlockedXor64_rel** intrinsic functions are the same as the corresponding functions without the **rel** suffix except that the operation is performed with release semantics, which is useful when leaving a critical section.

Example

```
// _InterLockedXor.cpp
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_InterlockedXor)

int main()
{
        long data1 = 0xFF00FF00;
        long data2 = 0x00FFFF00;
        long retval;
        retval = _InterlockedXor(&data1, data2);
        printf_s("0x%x 0x%x 0x%x", data1, data2, retval);
}
```

Output

```
0xffff0000 0xffff00 0xff00ff00
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__inbyte

Microsoft Specific

Generates the **in** instruction, returning one byte read from the port specified by *Port*.

```
unsigned char __inbyte(
   unsigned short Port
);
```

Parameters

[in] *Port*

The port to read from.

Return Value

The byte read from the specified port.

Requirements

Intrinsic	Architecture
__inbyte	x86, x64

Header file <intrin.h>

END Microsoft Specific

Remarks

This routine is only available as an intrinsic.

See Also

Reference

Compiler Intrinsics

__inbytestring

Microsoft Specific

Reads data from the specified port using the **rep insb** instruction.

```
void __inbytestring(
   unsigned short Port,
   unsigned char* Buffer,
   unsigned long Count
);
```

Parameters

[in] *Port*

The port to read from.

[out] *Buffer*

The data read from the port is written here.

[in] *Count*

The number of bytes of data to read.

Requirements

Intrinsic	Architecture
__inbytestring	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__incfsbyte, __incfsword, __incfsdword

Microsoft Specific

Add one to the value at a memory location specified by an offset relative to the beginning of the **FS** segment.

```
void __incfsbyte(
   unsigned long Offset
);
void __incfsword(
   unsigned long Offset
);
void __incfsdword(
   unsigned long Offset
);
```

Parameters

[in] *Offset*

The offset from the beginning of **FS**.

Requirements

Intrinsic	Architecture
__incfsbyte	x86
__incfsword	x86
__incfsdword	x86

Remarks

These intrinsics are only available in kernel mode, and the routines are only available as intrinsics.

END Microsoft Specific

See Also

Reference

__addfsbyte, __addfsword, __addfsdword
__readfsbyte, __readfsdword, __readfsqword, __readfsword
__writefsbyte, __writefsdword, __writefsqword, __writefsword
Compiler Intrinsics

__incgsbyte, __incgsword, __incgsdword, __incgsqword

Microsoft Specific

Add one to the value at a memory location specified by an offset relative to the beginning of the **GS** segment.

```
unsigned char __incgsbyte(
   unsigned long Offset
);
unsigned short __incgsword(
   unsigned long Offset
);
unsigned long __incgsdword(
   unsigned long Offset
);
unsigned __int64 __incgsqword(
   unsigned long Offset
);
```

Parameters

[in] *Offset*

The offset from the beginning of **GS**.

Requirements

Intrinsic	Architecture
__incgsbyte	x64
__incgsword	x64
__incgsdword	x64
__incgsqword	x64

Remarks

These intrinsics are only available in kernel mode, and the routines are only available as intrinsics.

END Microsoft Specific

See Also

Reference

__addgsbyte, __addgsword, __addgsdword, __addgsqword
__readgsbyte, __readgsdword, __readgsqword, __readgsword
__writegsbyte, __writegsdword, __writegsqword, __writegsword
Compiler Intrinsics

__indword

Microsoft Specific

Reads one double word of data from the specified port using the **in** instruction.

```
unsigned long __indword(
   unsigned short Port
);
```

Parameters

[in] *Port*

The port to read from.

Return Value

The word read from the port.

Requirements

Intrinsic	Architecture
__indword	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__indwordstring

Microsoft Specific

Reads data from the specified port using the **rep insd** instruction.

```
void __indwordstring(
   unsigned short Port,
   unsigned long* Buffer,
   unsigned long Count
);
```

Parameters

[in] *Port*

The port to read from.

[out] *Buffer*

The data read from the port is written here.

[in] *Count*

The number of bytes of data to read.

Requirements

Intrinsic	**Architecture**
__indwordstring	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__int2c

Microsoft Specific

Generates the **int 2c** instruction, which triggers the **2c** interrupt.

```
void __int2c(void);
```

Requirements

Intrinsic	Architecture
__int2c	x86, x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__invalat

Microsoft Specific

Emits the **invala** instruction, which invalidates the IPF processor's advanced load address table (ALAT).

```
void __invalat(void);
```

Requirements

Intrinsic	Architecture
__invalat	IPF

Header file <intrin.h>

Remarks

For more information about the **invala** instruction, see the Intel IPF documentation.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__invlpg

Microsoft Specific

Generates the x86 **invlpg** instruction, which invalidates the translation lookaside buffer (TLB) for the page associated with memory pointed to by *Address*.

```
void __invlpg(
   void* Address
);
```

Parameters

[in] *Address*

A 64-bit address.

Requirements

Intrinsic	Architecture
__invlpg	x86, x64

Header file <intrin.h>

Remarks

The intrinsic `__invlpg` emits a privileged instruction and is only available in kernel mode with a privilege level (CPL) of 0.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__inword

Microsoft Specific

Reads data from the specified port using the **in** instruction.

```
unsigned short __inword(
   unsigned short Port
);
```

Parameters

[in] *Port*

The port to read from.

Return Value

The word of data read.

Requirements

Intrinsic	Architecture
__inword	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__inwordstring

Microsoft Specific

Reads data from the specified port using the **rep insw** instruction.

```
void __inwordstring(
   unsigned short Port,
   unsigned short* Buffer,
   unsigned long Count
);
```

Parameters

[in] *Port*

The port to read from.

[out] *Buffer*

The data read from the port is written here.

[in] *Count*

The number of words of data to read.

Requirements

Intrinsic	Architecture
__inwordstring	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__isNat

Microsoft Specific

Checks the Not a Thing (NaT) bit in the register specified, indicating that the register does not contain a valid number.

```
int isNat(
   int Register
);
```

Parameters

[in] *Register*

The register to examine.

Return Value

0, if the register's NaT bit is 1.

Nonzero, if the register NaT bit is zero.

Requirements

Intrinsic	**Architecture**
__isNat	IPF

Header file <intrin.h>

Remarks

The IPF instruction generated is **tnat.nz**.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__isrlz

Microsoft Specific

Generates the IPF Serialize Instruction (**srlz.i**) instruction.

```
void __isrlz(void);
```

Requirements

Intrinsic	Architecture
__isrlz	IPF

Header file <intrin.h>

Remarks

For more information about the **srlz** instruction, see the Intel IPF documentation.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

__dsrlz

Compiler Intrinsics

__lfetch, __lfetch_excl, __lfetchfault, __lfetchfault_excl

Microsoft Specific

Generate various forms of the Line Prefetch (**lfetch**) instruction, which is used to tell the processor that an address is to be used frequently.

```
void __lfetch(
   unsigned int Hint,
   const void * Address
);
void __lfetch_excl(
   unsigned int Hint,
   const void * Address
);
void __lfetchfault(
   unsigned int Hint,
   const void * Address
);
void __lfetchfault_excl(
   unsigned int Hint,
   const void * Address
);
```

Parameters

[in] *Hint*

A hint to the processor about the cache level to use.

[in] *Address*

The address to reposition in the cache.

Requirements

Intrinsic	**Architecture**
__lfetch	IPF
__lfetch_excl	IPF
__lfetchfault	IPF
__lfetchfault_excl	IPF

Header file <intrin.h>

Remarks

The valid *Hint* values are constants from 0 through 3, which specify the locality hint. A *Hint* value of 0 indicates temporal locality (level 1), a 1 indicates no temporal locality (level 1), 2 indicates no temporal locality (level 2), and 3 represents no temporal locality (all levels). Any other value produces a compile time error. For more information, see the Intel IPF documentation.

The following table shows the instructions generated by **__lfetch**, **__lfetch_excl**, **__lfetchfault**, and **__lfetchfault_excl** with various values of the *Hint* parameter.

Hint	**__lfetch**	**__lfetch_excl**	**__lfetchfault**	**__lfetchfault_excl**
0	**Lfetch**	**lfetch.excl**	**lfetch.fault**	**lfetch.fault.excl**
1	**lfetch.nt1**	**lfetch.excl.nt1**	**lfetch.fault.nt1**	**lfetch.fault.excl.nt1**

2	**lfetch.nt2**	**lfetch.excl.nt2**	**lfetch.fault.nt2**	**lfetch.fault.excl.nt2**
3	**lfetch.nta**	**lfetch.excl.nta**	**lfetch.fault.nta**	**lfetch.fault.excl.nta**

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__ll_lshift

Microsoft Specific

Shifts a 64-bit value specified by the first parameter to the left by a number of bits specified by the second parameter.

```
unsigned __int64 __ll_lshift(
   unsigned __int64 Mask,
   int nBit
);
```

Parameters

[in] *Mask*

The 64-bit integer value to shift left.

[in] *nBit*

The number of bits to shift, modulo 64.

Return Value

The mask shifted left by *nBit* bits.

Requirements

Intrinsic	Architecture
__ll_lshift	x86, x64

Header file <intrin.h>

Remarks

If the second parameter is greater than 64, that number is taken modulo 64 to determine the number of bits to shift. The **ll** in the name indicates that this is an operation on **long long** (**__int64**).

Example

```
// ll_lshift.cpp
// compile with: /EHsc
// processor: x86, x64
#include <iostream>
#include <intrin.h>
using namespace std;

#pragma intrinsic(__ll_lshift)

int main()
{
   unsigned __int64 Mask = 0x100;
   int nBit = 8;
   Mask = __ll_lshift(Mask, nBit);
   cout << hex << Mask << endl;
}
```

Output

```
10000
```

Note There is no unsigned version of the left shift operation. Unlike the right shift, there is no sign dependence for the left shift, because the least significant bit in the result is always set to zero regardless of the sign of the value shifted.

END Microsoft Specific

See Also

Reference

__ll_rshift

__ll_rshift

Microsoft Specific

Shifts a 64-bit value specified by the first parameter to the right by a number of bits specified by the second parameter.

```
__int64 __ll_rshift(
   __int64 Mask,
   int nBit
);
```

Parameters

[in] *Mask*

The 64-bit integer value to shift right.

[in] *nBit*

The number of bits to shift, modulo 64 on x64, and modulo 32 on x86.

Return Value

The mask shifted by *nBit* bits.

Requirements

Intrinsic	Architecture
__ll_rshift	x86, x64

Header file <intrin.h>

Remarks

If the second parameter is greater than 64 on x64 (32 on x86), that number is taken modulo 64 (32 on x86) to determine the number of bits to shift. The **ll** prefix indicates that this is an operation on **long long**, another name for **__int64**, the 64-bit signed integral type.

Example

```
// ll_rshift.cpp
// compile with: /EHsc
// processor: x86, x64
#include <iostream>
#include <intrin.h>
using namespace std;

#pragma intrinsic(__ll_rshift)

int main()
{
   __int64 Mask = - 0x100;
   int nBit = 4;
   cout << hex << Mask << endl;
   cout << " - " << (- Mask) << endl;
   Mask = __ll_rshift(Mask, nBit);
   cout << hex << Mask << endl;
   cout << " - " << (- Mask) << endl;
}
```

Output

```
ffffffffffffff00
 - 100
fffffffffffffff0
 - 10
```

Note If **_ull_rshift** has been used, the MSB of the right-shifted value would have been zero, so the desired result would not have been obtained in the case of a negative value.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__ll_lshift

__ull_rshift

__load128, __load128_acq

Microsoft Specific

Loads a 128-bit value atomically.

```
__int64 __load128(
   __int64 volatile * Source,
   __int64 *DestinationHigh
);
__int64 __load128_acq(
   __int64 volatile * Source,
   __int64 *DestinationHigh
);
```

Parameters

[in] *Source*

Pointer to the 128-bit source value.

[out] *DestinationHigh*

Pointer to a memory location where the most significant 64-bit quadword of the 128-bit loaded value is stored.

Return Value

The return value is the least significant quadword of the 128-bit loaded value.

Platforms

Function	Architecture
__load128	IA64
__load128_acq	IA64

Requirements

Header file <intrin.h>

Remarks

The **__load128** function performs an atomic 128-bit read from the source memory location. The *Source* pointer should point to a 16-byte aligned memory location.

These functions are not general purpose 128-bit intrinsics. These functions should only be used if you really need 128-bit atomic loads. You get better performance with regular 64-bit operations.

Note that the intrinsic form of this function has a leading underscore in the name. Also note that these functions are always expanded as intrinsics. That means you might use /Oi or #pragma intrinsic, but it is not required.

The **__load128_acq** intrinsic function is the same as the corresponding function without the **_acq** suffix except that the operation is performed with acquire semantics.

In Visual C++ 2005, these functions behave as read-write barriers. For more information, see _ReadWriteBarrier.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__m64_czx1l, __m64_czx1r, __m64_czx2l, __m64_czx2r

Microsoft Specific

Emit various forms of the IPF Compute Zero Index (**czx**) instruction.

```
__m64 __m64_czx1l(
   __m64 value
);
__m64 __m64_czx1r(
   __m64 value
);
__m64 __m64_czx2l(
   __m64 value
);
__m64 __m64_czx2r(
   __m64 value
);
```

Parameters

[in] *value*

The 64-bit value to scan.

Return Value

The 1-byte form looks at each aligned 8-bit block in turn, either from right to left or left to right (depending on the intrinsic used), identifying the first aligned 8-bit section that is all zeroes and returning a number from 0 to 7 indicating the number of bytes scanned before the first zero byte was found. If no zero byte is found, the return value is 8. The 2-byte form looks at each aligned 16-bit block in turn, either from right to left or left to right, identifying the first aligned 16-bit section found that is all zeroes. A return value from 0 to 3 indicates the number of 16-bit blocks scanned before finding the first 16-bit block that was all zeroes. If no such section is found, the return value is 4.

Requirements

Intrinsic	Architecture
__m64_czx1l	IPF
__m64_czx1r	IPF
__m64_czx2l	IPF
__m64_czx2r	IPF

Header file <intrin.h>

Remarks

__m64_czx1l and **__m64_czx1r** emit the left and right forms of the 1-byte form of the instruction (**czx1.l** and **czx1.r**, respectively). **__m64_czx2l** and **__m64_czx2r** emit the left and right forms of the 2-byte form of the instruction (**czx2.l** and **czx2.r**, respectively). These intrinsics scan the value passed in for an aligned section of 1 or 2 bytes in length (depending on the intrinsic used) that is all zeroes. The left forms of these instructions scan from the most significant bit (MSB) down, and the right forms scan from the least significant bit (LSB) up.

Example

```
// czx.cpp
// compile with: /EHsc
// processor: IPF
#include <iostream>
#include <intrin.h>
using namespace std;
```

```
#pragma intrinsic(__m64_czx1l, __m64_czx1r, __m64_czx2l, __m64_czx2r)

int main()
{
   __m64 i, j, k, m;
   __m64 result_czx1l, result_czx1r, result_czx2l, result_czx2r;

   i.m64_u64 = 0xff00ff00ff00ff00;
   j.m64_u64 = 0xffffffffffffffff;
   k.m64_u64 = 0x00000000ffffffff;
   m.m64_u64 = 0x1101f0000230f10e;

   cout << hex;

   result_czx1l = __m64_czx1l(i);
   result_czx1r = __m64_czx1r(i);
   result_czx2l = __m64_czx2l(i);
   result_czx2r = __m64_czx2r(i);

   cout << "__m64_czx1l(" << i.m64_u64 << ") returns " <<
        result_czx1l.m64_u64 << " " << endl;
   cout << "__m64_czx1r(" << i.m64_u64 << ") returns " <<
        result_czx1r.m64_u64 << " " << endl;
   cout << "__m64_czx2l(" << i.m64_u64 << ") returns " <<
        result_czx2l.m64_u64 << " " << endl;
   cout << "__m64_czx2r(" << i.m64_u64 << ") returns " <<
        result_czx2r.m64_u64 << " " << endl;

   result_czx1l = __m64_czx1l(j);
   result_czx1r = __m64_czx1r(j);
   result_czx2l = __m64_czx2l(j);
   result_czx2r = __m64_czx2r(j);

   cout << "__m64_czx1l(" << j.m64_u64 << ") returns " <<
        result_czx1l.m64_u64 << " " << endl;
   cout << "__m64_czx1r(" << j.m64_u64 << ") returns " <<
        result_czx1r.m64_u64 << " " << endl;
   cout << "__m64_czx2l(" << j.m64_u64 << ") returns " <<
        result_czx2l.m64_u64 << " " << endl;
   cout << "__m64_czx2r(" << j.m64_u64 << ") returns " <<
        result_czx2r.m64_u64 << " " << endl;

   result_czx1l = __m64_czx1l(k);
   result_czx1r = __m64_czx1r(k);
   result_czx2l = __m64_czx2l(k);
   result_czx2r = __m64_czx2r(k);

   cout << "__m64_czx1l(" << k.m64_u64 << ") returns " <<
        result_czx1l.m64_u64 << " " << endl;
   cout << "__m64_czx1r(" << k.m64_u64 << ") returns " <<
        result_czx1r.m64_u64 << " " << endl;
   cout << "__m64_czx2l(" << k.m64_u64 << ") returns " <<
        result_czx2l.m64_u64 << " " << endl;
   cout << "__m64_czx2r(" << k.m64_u64 << ") returns " <<
        result_czx2r.m64_u64 << " " << endl;

   result_czx1l = __m64_czx1l(m);
   result_czx1r = __m64_czx1r(m);
   result_czx2l = __m64_czx2l(m);
   result_czx2r = __m64_czx2r(m);

   // Even though m has a 16-bit section of zeroes, it is not aligned
   // on a 16-byte boundary, so won't be a match.

   cout << "__m64_czx1l(" << m.m64_u64 << ") returns " <<
        result_czx1l.m64_u64 << " " << endl;
   cout << "__m64_czx1r(" << m.m64_u64 << ") returns " <<
        result_czx1r.m64_u64 << " " << endl;
```

```
    cout << "__m64_czx2l(" << m.m64_u64 << ") returns " <<
         result_czx2l.m64_u64 << " " << endl;
    cout << "__m64_czx2r(" << m.m64_u64 << ") returns " <<
         result_czx2r.m64_u64 << " " << endl;

  }
```

Output

```
__m64_czx1l(ff00ff00ff00ff00) returns 1
__m64_czx1r(ff00ff00ff00ff00) returns 0
__m64_czx2l(ff00ff00ff00ff00) returns 4
__m64_czx2r(ff00ff00ff00ff00) returns 4
__m64_czx1l(ffffffffffffffff) returns 8
__m64_czx1r(ffffffffffffffff) returns 8
__m64_czx2l(ffffffffffffffff) returns 4
__m64_czx2r(ffffffffffffffff) returns 4
__m64_czx1l(ffffffff) returns 0
__m64_czx1r(ffffffff) returns 4
__m64_czx2l(ffffffff) returns 0
__m64_czx2r(ffffffff) returns 2
__m64_czx1l(1101f0000230f10e) returns 3
__m64_czx1r(1101f0000230f10e) returns 4
__m64_czx2l(1101f0000230f10e) returns 4
__m64_czx2r(1101f0000230f10e) returns 4
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_dep_mi

Microsoft Specific

Emits the merge immediate form of the IPF Deposit (**dep**) instruction, which is used for copying a number of bits specified by *len* with the value *bit* into a value *value* in a register at the bit position specified by *pos*.

```
__m64 __m64_dep_mi(
   const int bit,
   __m64 value,
   const int pos,
   const int len
);
```

Parameters

[in] *bit*

Either 0 or 1. The value to set the merged bits to.

[in] *value*

The value to be merged into.

[in] *pos*

The bit position in the value to merge the bits. Valid values are 0 to 63.

[in] *len*

len is the number of bits to merge. Valid values are 1 to 64.

Requirements

Intrinsic	Architecture
__m64_dep_mi	IPF

Header file <intrin.h>

Example

```
// dep.cpp
// compile with: /EHsc
// processor: IPF
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__m64_dep_mi)

int main()
{
    __m64 m;
    // Merge a "1" bit at bit 1 in 0x200
    m = __m64_dep_mi(1, _m_from_int(0x200), 1, 1);

    printf_s("%I64x", m.m64_u64);
}
```

Output

```
202
```

END Microsoft Specific

See Also

Reference

__m64_dep_mr

__m64_dep_zi
__m64_dep_zr
__m64
Compiler Intrinsics

__m64_dep_mr

Microsoft Specific

Emits the merge register form of the IPF Deposit (**dep**) instruction, which is used for copying a number of bits specified by *len* from a value *source* into a value *value* in a register at the bit position specified by *pos*.

```
__m64 __m64_dep_mr(
   __m64 source,
   __m64 value,
   const int pos,
   const int len
);
```

Parameters

[in] *source*

The value from which to extract the bits for the merge.

[in] *value*

The value into which to merge.

[in] *pos*

The destination bit position for the merge into *value*. Valid values from 0 to 63.

[in] *len*

The number of bits to merge. Valid values from 1 to 16.

Requirements

Intrinsic	Architecture
__m64_dep_mr	IPF

Header file <intrin.h>

Example

```
// dep_mr.cpp
// compile with: /EHsc
// processor: IPF
#include <iostream>
#include <intrin.h>
using namespace std;

#pragma intrinsic(__m64_dep_mr)

int main()
{
  __m64 m, n;
  m.m64_u64 = 0xE00010;
  n.m64_u64 = 0x8888;

  __m64 result = __m64_dep_mr(n, m, 4, 16);

  cout << hex << "0x" << result.m64_u64;
}
```

Output

```
0xe88880
```

END Microsoft Specific

See Also

Reference

__m64_dep_mi
__m64_dep_zi
__m64_dep_zr
__m64
Compiler Intrinsics

__m64_dep_zi

Microsoft Specific

Emits the zero immediate form of the IPF Deposit (**dep.z**) instruction, which is used to copy a number of bits specified by *len* from a constant value *source* into a bit position specified by *pos* of the resulting value.

```
__m64 __m64_dep_zi(
   const int source,
   const int pos,
   const int len
);
```

Parameters

[in] *source*

An 8-bit constant value to act as the source of bits to be copied into the result.

[in] *pos*

The bit position in the result to place the copied bits.

[in] *len*

The number of bits to copy.

Requirements

Intrinsic	Architecture
__m64_dep_zi	IPF

Header file <intrin.h>

Remarks

The bits are copied sign extended. The rest of the bits in the result are cleared to zero.

Example

```
// dep_zi.cpp
// compile with: /EHsc
// processor: IPF
#include <iostream>
#include <intrin.h>
using namespace std;

#pragma intrinsic(__m64_dep_zi)

int main()
{
  const int imm = 0x12;

  __m64 result = __m64_dep_zi(imm, 4, 16);

  cout << hex << "0x" << result.m64_i64 << endl;

  // When copying a number of bits larger than an int,
  // the copied bits are sign extended:
  const int imm2 = -1;
  result = __m64_dep_zi(imm2, 0, 56);

  cout << hex << "0x" << result.m64_i64;
}
```

Output

```
0x120
0xffffffffffffff
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_dep_zr

Microsoft Specific

Emits the zero register form of the IPF Deposit (**dep.z**) instruction, which is used to copy a number of bits specified by *len* from a variable *source* into a bit position specified by *pos* of the resulting value.

```
__m64 __m64_dep_zr(
   __m64 source,
   const int pos,
   const int len
);
```

Parameters

[in] *source*

The value from which to copy the bits.

[in] *pos*

The destination bit position for the copied bits in the result. Valid values from 0 to 63.

[in] *len*

The number of bits to copy. Valid values from 1 to 64.

Requirements

Intrinsic	Architecture
__m64_dep_zr	IPF

Header file <intrin.h>

Remarks

The rest of the bits in the result are cleared to zero.

Example

```
// dep_zr.cpp
// compile with: /EHsc
// processor: IPF
#include <iostream>
#include <intrin.h>
using namespace std;

#pragma intrinsic(__m64_dep_zr)

int main()
{
  __m64 n;
  n.m64_u64 = 0x8888;

  __m64 result = __m64_dep_zr(n, 4, 16);

  cout << hex << "0x" << result.m64_u64;
}
```

Output

```
0x88880
```

END Microsoft Specific

See Also

Reference

__m64_extr, __m64_extru

Microsoft Specific

Emit the IPF Extract (**extr**) instruction.

```
__m64 __m64_extr(
   __m64 value,
   const int pos,
   const int len
);
__m64 __m64_extru(
   __m64 value,
   const int pos,
   const int len
);
```

Parameters

[in] *value*

The variable to extract bits from.

[in] *pos*

The position of the first bit to extract. Valid values from 0 to 63.

[in] *len*

The number of bits to extract. Valid values from 1 to 64.

Requirements

Intrinsic	Architecture
__m64_extr	IPF
__m64_extru	IPF

Header file <intrin.h>

Remarks

__m64_extr emits the signed form (**extr**) and **__m64_extru** emits the unsigned form (**extr.u**). Both instructions extract *len* bits starting at bit *pos* (counted from the least significant bit). These bits are placed in the least significant *len* bits of the result. The result is sign extended in the signed form of the instruction, and zero extended in the unsigned form. The highest bit extracted determines the sign.

Example

```
// extr.cpp
// processor: IPF
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__m64_extr, __m64_extru)

int main()
{
    __m64 m, n;
    m.m64_i64 = 0x8800ff;

    // The signed version does a sign extension based on the MSB
    // of the extracted portion.
    n = __m64_extr(m, 16, 8);
    printf_s("0x%I64x\n", n.m64_i64);
```

```
    // The unsigned version does a zero extension.
    n = __m64_extru(m, 16, 8);
    printf_s("0x%I64x\n", n.m64_i64);
    return 1;
}
```

Output

```
0xffffffffffffff88
0x88
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_mix1l, __m64_mix1r, __m64_mix2l, __m64_mix2r, __m64_mix4l, __m64_mix4r

Microsoft Specific

Emit various forms of the IPF Mix (**mix**) instruction, which generates a result based on a certain mix of the bytes from the two operands.

```
__m64 __m64_mix1l(
   __m64 a,
   __m64 b
);
__m64 __m64_mix1r(
   __m64 a,
   __m64 b
);
__m64 __m64_mix2l(
   __m64 a,
   __m64 b
);
__m64 __m64_mix2r(
   __m64 a,
   __m64 b
);
__m64 __m64_mix4l(
   __m64 a,
   __m64 b
);
__m64 __m64_mix4r(
   __m64 a,
   __m64 b
);
```

Parameters

[in] *a*

The first source value.

[in] *b*

The second source value.

Requirements

Intrinsic	Architecture
__m64_mix1l	IPF
__m64_mix1r	IPF
__m64_mix2l	IPF
__m64_mix2r	IPF
__m64_mix4l	IPF
__m64_mix4r	IPF

Header file <intrin.h>

Remarks

The various forms of the instruction specify the way in which the bytes are arranged to form the result. The instructions vary according to the number of bytes that are copied in blocks and whether the left or right block of each pair of blocks is chosen for the result.

The source parameters are considered to be divided into blocks of size 1, 2, or 4 bytes as indicated by the name of the intrinsic. Adjacent source blocks are grouped into pairs with a left (higher order) and right (lower order) member of each pair.

For the left-handed (l) versions of these intrinsics, the result is built up from left to right in the following way: each successive pair of blocks in the result is formed by choosing the left block of a pair from *a* followed by the left block of the corresponding pair from *b*, always choosing pairs from left to right.

For the right-handed (r) versions of these intrinsics, the result is built up from left to right in the following way: each successive pair of blocks in the result is formed by choosing the right block of a pair from *a* followed by the right block of the corresponding pair from *b*, always choosing pairs from left to right.

Example

```
// mix.cpp
// processor: IPF
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__m64_mix1l, __m64_mix1r, __m64_mix2l)
#pragma intrinsic(__m64_mix2r, __m64_mix4l, __m64_mix4r)

void f(__m64 a, __m64 b)
{
    __m64 t;
    t = __m64_mix1l(a, b);
    printf_s("mix1l(0x%I64x, 0x%I64x) returns 0x%I64x\n",
             a.m64_u64, b.m64_u64, t.m64_u64);
    t = __m64_mix1r(a, b);
    printf_s("mix1r(0x%I64x, 0x%I64x) returns 0x%I64x\n",
             a.m64_u64, b.m64_u64, t.m64_u64);
    t = __m64_mix2l(a, b);
    printf_s("mix2l(0x%I64x, 0x%I64x) returns 0x%I64x\n",
             a.m64_u64, b.m64_u64, t.m64_u64);
    t = __m64_mix2r(a, b);
    printf_s("mix2r(0x%I64x, 0x%I64x) returns 0x%I64x\n",
             a.m64_u64, b.m64_u64, t.m64_u64);
    t = __m64_mix4l(a, b);
    printf_s("mix4l(0x%I64x, 0x%I64x) returns 0x%I64x\n",
             a.m64_u64, b.m64_u64, t.m64_u64);
    t = __m64_mix4r(a, b);
    printf_s("mix4r(0x%I64x, 0x%I64x) returns 0x%I64x\n",
             a.m64_u64, b.m64_u64, t.m64_u64);
}

int main()
{
    __m64 m, n;
    m.m64_u64 = -1;
    n.m64_u64 = 0;

    f(m, n);

    m.m64_u64 = 0x7766554433221100;
    n.m64_u64 = 0xFFEEDDCCBBAA9988;

    f(m, n);
}
```

Output

```
mix1l(0xffffffffffffffff, 0x0) returns 0xff00ff00ff00ff00
mix1r(0xffffffffffffffff, 0x0) returns 0xff00ff00ff00ff00
mix2l(0xffffffffffffffff, 0x0) returns 0xffff0000ffff0000
mix2r(0xffffffffffffffff, 0x0) returns 0xffff0000ffff0000
mix4l(0xffffffffffffffff, 0x0) returns 0xffffffff00000000
```

```
mix4r(0xffffffffffffffff, 0x0) returns 0xffffffff00000000
mix1l(0x7766554433221100, 0xffeeddccbbaa9988) returns 0x77ff55dd33bb1199
mix1r(0x7766554433221100, 0xffeeddccbbaa9988) returns 0x66ee44cc22aa0088
mix2l(0x7766554433221100, 0xffeeddccbbaa9988) returns 0x7766ffee3322bbaa
mix2r(0x7766554433221100, 0xffeeddccbbaa9988) returns 0x5544ddcc11009988
mix4l(0x7766554433221100, 0xffeeddccbbaa9988) returns 0x77665544ffeeddcc
mix4r(0x7766554433221100, 0xffeeddccbbaa9988) returns 0x33221100bbaa9988
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_muladd64hi, __m64_muladd64hi_u

Microsoft Specific

Compute the high 64 bits of the expression *a***b*+*c*.

```
__m64 __m64_muladd64hi(
   __m64 a,
   __m64 b,
   __m64 c
);
__m64 __m64_muladd64hi_u(
   __m64 a,
   __m64 b,
   __m64 c
);
```

Parameters

[in] *a*

The first factor.

[in] *b*

The second factor.

[in] *c*

The number to be added to the product of the first two parameters.

Requirements

Intrinsic	Architecture
__m64_muladd64hi	IPF
__m64_muladd64hi_u	IPF

Header file <intrin.h>

Remarks

__m64_muladd64hi treats the arguments as signed 64-bit integers. **__m64_muladd64hi_u** treats the arguments as unsigned 64-bit integers.

Example

```
// muladd64hi.c
// processor: IPF

#include <stdio.h>
#include <intrin.h>
#include <limits.h>

#pragma intrinsic(__m64_muladd64hi, __m64_muladd64hi_u, __m64_muladd64lo)

int main()
{
    __m64 a, b, c, result;
    a.m64_i64 = 2I64;
    b.m64_i64 = _I64_MAX;
    c.m64_i64 = _I64_MAX;

    result = __m64_muladd64hi(a, b, c);
    printf_s("0x%I64x", result.m64_i64);
```

```
    result = __m64_muladd64lo(a, b, c);
    printf_s("%I64x\n", result.m64_i64);

    b.m64_u64 = _UI64_MAX;
    c.m64_u64 = _UI64_MAX;
    result = __m64_muladd64hi_u(a, b, c);
    printf_s("0x%I64x", result.m64_u64);

    result = __m64_muladd64lo(a, b, c);
    printf_s("%I64x", result.m64_u64);

    return 0;
}
```

Output

```
0x17fffffffffffffd
0x2fffffffffffffd
```

END Microsoft Specific

See Also

Reference

__m64_muladd64lo

__m64

Compiler Intrinsics

__m64_muladd64lo

Microsoft Specific

Computes the low 64 bits of the expression *a***b* + *c*.

```
__m64 __m64_muladd64lo(
   __m64 a,
   __m64 b,
   __m64 c
);
```

Parameters

[in] *a*

The first number to multiply.

[in] *b*

The second number to multiply.

[in] *c*

The number to add.

Requirements

Intrinsic	Architecture
__m64_muladd64lo	IPF

Header file <intrin.h>

Example

See __m64_muladd64hi, __m64_muladd64hi_u.

END Microsoft Specific

See Also

Reference

__m64

__m64_muladd64hi, __m64_muladd64hi_u

Compiler Intrinsics

__m64_mux1

Microsoft Specific

Emits the IPF instruction **mux1**, the 1-byte form of the **mux** instruction, which is used to arrange bytes in a particular pattern specified by the second parameter.

```
__m64 __m64_mux1(
   __m64 source,
   const int permutation
);
```

Parameters

[in] *source*

The source data for the permutation operation.

[in] *permutation*

Specifies the permutation of the source bytes in the result.

Requirements

Intrinsic	Architecture
__m64_mux1	IPF

Header file <intrin.h>

Remarks

The available patterns are specified in the following table.

Permutation	Operation	Description
0x0	@brcst	Copy the least significant byte to all bytes of the result.
0x8	@mix	Perform a Mix operation on the two halves of the source.
0x9	@shuf	Perform a Shuffle operation on the two halves of the source.
0xA	@alt	Perform an Alternate operation on the two halves of the source.
0xB	@rev	Reverse the order of the bytes.

Example

```
// mux1.cpp
// processor: IPF
#include <stdio.h>
#include <intrin.h>
#include <memory.h>

#pragma intrinsic(__m64_mux1)

void m64tochararray(__m64 m64, char* c)
{
    for (int j = 0; j < 8; j++)
        c[j] = m64.m64_i8[j];
}

void chararraytom64(const char* c, __m64* pm64)
{
    for (int j = 0; j < 8; j++)
```

```
        pm64->m64_i8[j] = c[j];
}

int main()
{
    char s1[9] = "01234567";
    char s2[9];
    __m64 m, result;

    printf_s("Source string:         %s\n", s1);

    memset(s2, 0, 9);
    chararraytom64(s1, &m);
    result = __m64_mux1(m, 0); // @brcst form
    m64tochararray(result, s2);
    printf_s("mux1 @brcst (0x0) form: %s\n", s2);

    memset(s2, 0, 9);
    chararraytom64(s1, &m);
    result = __m64_mux1(m, 8); // @mix form
    m64tochararray(result, s2);
    printf_s("mux1 @mix   (0x8) form: %s\n", s2);

    memset(s2, 0, 9);
    chararraytom64(s1, &m);
    result = __m64_mux1(m, 9); // @shuf form
    m64tochararray(result, s2);
    printf_s("mux1 @shuf  (0x9) form: %s\n", s2);

    memset(s2, 0, 9);
    chararraytom64(s1, &m);
    result = __m64_mux1(m, 0xA); // @alt form
    m64tochararray(result, s2);
    printf_s("mux1 @alt   (0xA) form: %s\n", s2);

    memset(s2, 0, 9);
    chararraytom64(s1, &m);
    result = __m64_mux1(m, 0xB); // @rev form
    m64tochararray(result, s2);
    printf_s("mux1 @rev   (0xB) form: %s\n", s2);
}
```

Output

```
Source string:          01234567
mux1 @brcst (0x0) form: 00000000
mux1 @mix   (0x8) form: 04261537
mux1 @shuf  (0x9) form: 04152637
mux1 @alt   (0xA) form: 02461357
mux1 @rev   (0xB) form: 76543210
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_mux2

Microsoft Specific

Emits the IPF **mux2** instruction, the 2-byte form of the **mux** instruction, which is used to arrange bytes in a particular pattern specified by the second parameter.

```
__m64 __m64_mux2(
   __m64 source,
   const int permutation
);
```

Parameters

[in] *source*

The source data for the permutation operation.

[in] *permutation*

Specifies the permutation of the source bytes in the result.

Requirements

Intrinsic	Architecture
__m64_mux2	IPF

Header file <intrin.h>

Remarks

The second parameter is 8 bits. The first pair of bits (moving from left to right, or most significant bit to least significant bit) tells which of the four 16-bit sections of the source to choose for the first 16-bit section of the result, and so on for the second, third, and fourth pairs of bits in the *permutation* parameter. For example, 0x1b (00011011) reverses the bits, and 0xe4 (11100100) puts the bits in the same position as the source. The source bits might be chosen more than once; for example, 0x0 (00000000) broadcasts the first 16-bits into every 16-bit slot of the result and 0xFF (11111111) broadcasts the last 16-bits into every 16-bit slot of the result.

Example

```
// mux2.cpp
// processor: IPF
#include <stdio.h>
#include <intrin.h>
#include <memory.h>

#pragma intrinsic(__m64_mux2)

void m64tochararray(__m64 m64, char* c)
{
    for (int j = 0; j < 8; j++)
        c[j] = m64.m64_i8[j];
}

void chararraytom64(const char* c, __m64* pm64)
{
    for (int j = 0; j < 8; j++)
        pm64->m64_i8[j] = c[j];
}

int main()
{
    char s1[9] = "00112233";
    char s2[9];
    __m64 m, result;
```

```
    printf_s("Source string:    %s\n", s1);

    memset(s2, 0, 9);
    chararraytom64(s1, &m);
    result = __m64_mux2(m, 0x1b);  // reverse: 00011011
    m64tochararray(result, s2);
    printf_s("mux2 (0x1b) form: %s\n", s2);

    memset(s2, 0, 9);
    chararraytom64(s1, &m);
    result = __m64_mux2(m, 0xe4); // identity: 11100100
    m64tochararray(result, s2);
    printf_s("mux2 (0xe4) form: %s\n", s2);

    memset(s2, 0, 9);
    chararraytom64(s1, &m);
    result = __m64_mux2(m, 0);     // broadcast the first 16-bits
    m64tochararray(result, s2);
    printf_s("mux2 (0x00) form: %s\n", s2);

    memset(s2, 0, 9);
    chararraytom64(s1, &m);
    result = __m64_mux2(m, 0xff);  // broadcast the last 16-bits
    m64tochararray(result, s2);
    printf_s("mux2 (0xff) form: %s\n", s2);

    memset(s2, 0, 9);
    chararraytom64(s1, &m);
    result = __m64_mux2(m, 0x8d);  // shuffle: 10001101
    m64tochararray(result, s2);
    printf_s("mux2 (0x8d) form: %s\n", s2);
}
```

Output

```
Source string:    00112233
mux2 (0x1b) form: 33221100
mux2 (0xe4) form: 00112233
mux2 (0x00) form: 00000000
mux2 (0xff) form: 33333333
mux2 (0x8d) form: 11330022
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_padd1uus

Microsoft Specific

Emits the IPF **padd1.uus** instruction (the 1-byte, unsigned, unsigned, signed-saturation form of Parallel Add).

```
__m64 __m64_padd1uus(
   __m64 a,
   __m64 b
);
```

Parameters

[in] *a*

The first 64-bit field containing bytes to be added.

[in] *b*

The second 64-bit field containing bytes to be added.

Requirements

Intrinsic	Architecture
__m64_padd1uus	IPF

Header file <intrin.h>

Remarks

Parallel Add adds each byte of one operand to the corresponding byte of the second operand and puts the result in the corresponding byte of the result. The bytes from the first operand are considered unsigned, and the bytes from the second operand are considered signed. The resulting bytes are considered unsigned.

If the result of the addition overflows 1 byte, the value saturates the byte and the byte is set to 0xFF. If the second (signed) parameter represents a negative integer greater in absolute value than the first parameter, the byte is said to underflow and is set to zero.

Example

```
// padd1uus.cpp
// processor: IPF
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__m64_padd1uus)

int main()
{
    __m64 m, n, result;
    m.m64_i64 = 0xee0100;
    n.m64_i64 = 0x00ff22;

    // As the second parameter, 0xff = -1,
    // so the second byte will add to zero.
    result = __m64_padd1uus(m, n);
    printf_s("__m64_padd1uus(0x%I64x, 0x%I64x) returns 0x%I64x\n",
             m, n, result);

    // This operation is not commutative, as this example shows.
    // The byte containing 0xff will be considered unsigned and when
    // 0x01 is added, will be saturated at the limit value of 0xff.
    // The byte containing ee will be a negative integer, which
    // when added to 0x00 will underflow, so that byte will be set
    // to zero.
    result = __m64_padd1uus(n, m);
```

```
    printf_s("__m64_padd1uus(0x%I64x, 0x%I64x) returns 0x%I64x\n",
             n, m, result);
}
```

Output

```
__m64_padd1uus(0xee0100, 0xff22) returns 0xee0022
__m64_padd1uus(0xff22, 0xee0100) returns 0xff22
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_padd2uus

Microsoft Specific

Emits the IPF **padd2.uus** instruction (the 2-byte, unsigned, unsigned, signed saturation form of Parallel Add).

```
__m64 __m64_padd2uus(
   __m64 a,
   __m64 b
);
```

Parameters

[in] *a*

The first 64-bit field containing 2-byte sections to be added.

[in] *b*

The second 64-bit field containing 2-byte sections to be added.

Requirements

Intrinsic	Architecture
__m64_padd1uus	IPF

Header file <intrin.h>

Remarks

__m64_padd2uus adds each 2-byte section of one operand with the corresponding 2-byte section of the second operand and puts the result in the corresponding 2-byte section of the result. The values from the first operand are considered unsigned, and the values from the second operand are considered signed. The resulting values are considered unsigned.

If the result of the addition overflows 2 bytes, the value saturates and the 16-bit section is set to 0xFFFF. If the result of the addition underflows, the 16 bits are set to zero.

Example

```
// padd2uus.cpp
// processor: IPF
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__m64_padd2uus)

int main()
{
    __m64 m, n, result;
    m.m64_i64 = 0x110022003300ff00;
    n.m64_i64 = 0x01100220033000ff;
    result = __m64_padd2uus(m, n);
    printf_s("__m64_padd2uus(%#18I64x, %#18I64x) ==\n %#18I64x\n",
             m, n, result);

    // This operation is not commutative!  The second parameter
    // is signed, so 0xff00 will be interpreted as a negative
    // integer (-256). When added to 0x00ff (256), the result is 0.
    result = __m64_padd2uus(n, m);
    printf_s("__m64_padd2uus(%#18I64x, %#18I64x) ==\n %#18I64x\n",
             n, m, result);
}
```

Output

```
__m64_padd2uus(0x110022003300ff00,  0x1100220033000ff) ==
```

```
 0x121024203630ffff
__m64_padd2uus( 0x1100220033000ff, 0x110022003300ff00) ==
 0x1210242036300000
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_pavg1_nraz

Microsoft Specific

Emits the 1-byte form of the IPF Parallel Average (**pavg1**) instruction, which computes the average of the two unsigned 8-bit integers in each byte of the input arguments and puts that average in the corresponding byte of the result.

```
__m64 __m64_pavg1_nraz(
   __m64 a,
   __m64 b
);
```

Parameters

[in] *a*

An **__m64** union containing an array of eight unsigned 8-bit integers.

[in] *b*

An **__m64** union containing an array of eight unsigned 8-bit integers.

Return Value

An **__m64** union containing the array of eight unsigned 8-bit integers, each position of which is the average of the corresponding integers in the source arrays.

Requirements

Intrinsic	Architecture
__m64_pavg1_nraz	IPF

Header file <intrin.h>

Remarks

If fractional, the result in each byte is rounded off to the nearest odd number. The suffix **nraz** means "not rounding away from zero."

Example

```
// pavg1_nraz.cpp
// processor: IPF
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__m64_pavg1_nraz)

void print8u(unsigned __int8* ia)
{
    printf_s("{ %4u, %4u, %4u, %4u, %4u, %4u, %4u, %4u }\n",
             ia[0], ia[1], ia[2], ia[3], ia[4], ia[5], ia[6], ia[7] );
}


int main()
{
    __m64 m, n, result;
    unsigned __int8 ra[8], i;
    unsigned __int8 ma[8] = { 10, 10, 10, 10, 10, 10, 10, 10 };
    unsigned __int8 na[8] = { 1, 3, 5,   7,   9, 11, 15, 1 };

    for (i = 0; i < 8; i++)
    {
       m.m64_u8[i] = ma[i];
       n.m64_u8[i] = na[i];
    }
```

```
    printf_s("a: \n");
    print8u(ma);
    printf_s("b: \n");
    print8u(na);
    printf_s("\n");

    result = __m64_pavg1_nraz(m, n);
    printf_s("Parallel Average (a[]+b[])/2 : \n");
    print8u(result.m64_u8);
    printf_s("\n");

    // This operation is commutative.
    result = __m64_pavg1_nraz(n, m);
    printf_s("Parallel Average (b[]+a[])/2 : \n");
    print8u(result.m64_u8);
    printf_s("\n");

}
```

Output

```
a:
{   10,   10,   10,   10,   10,   10,   10,   10 }
b:
{    1,    3,    5,    7,    9,   11,   15,    1 }

Parallel Average (a[]+b[])/2 :
{    5,    7,    7,    9,    9,   11,   13,    5 }

Parallel Average (b[]+a[])/2 :
{    5,    7,    7,    9,    9,   11,   13,    5 }
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_pavg2_nraz

Microsoft Specific

Emits the 2-byte form of the IPF Parallel Average (**pavg2**) instruction, which computes the average of the two unsigned short integers in each 16-bit block of the input arguments and puts that average in the corresponding 16-bit block of the result.

```
__m64 __m64_pavg2_nraz(
   __m64 a,
   __m64 b
);
```

Parameters

[in] *a*

An **__m64** union containing an array of four unsigned 16-bit integers.

[in] *b*

An **__m64** union containing an array of four unsigned 16-bit integers.

Return Value

An **__m64** union containing the array of four unsigned 16-bit integers, each element of which is the average of the corresponding integers in the source arrays.

Requirements

Intrinsic	Architecture
__m64_pavg2_nraz	IPF

Header file <intrin.h>

Remarks

If fractional, the result of each average is rounded to the nearest odd integer. The suffix **nraz** means "not rounding away from zero."

Example

```
// pavg2_nraz.cpp
// processor: IPF
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__m64_pavg2_nraz)

void print4u(unsigned __int16* ia)
{
    printf_s("{ %8u, %8u, %8u, %8u }\n", ia[0], ia[1], ia[2], ia[3] );
}


int main()
{
    __m64 m, n, result;
    unsigned __int16 ra[4], i;
    unsigned __int16 ma[4] = { 10, 100, 70, 10000 };
    unsigned __int16 na[4] = { 1, 95, 50,   5000 };

    for (i = 0; i < 4; i++)
    {
       m.m64_u16[i] = ma[i];
       n.m64_u16[i] = na[i];
    }
```

```
    printf_s("a: \n");
    print4u(ma);
    printf_s("b: \n");
    print4u(na);
    printf_s("\n");

    result = __m64_pavg2_nraz(m, n);

    printf_s("Parallel Average (a[]+b[])/2 : \n");
    print4u(result.m64_u16);
    printf_s("\n");

    // This operation is commutative.
    result = __m64_pavg2_nraz(n, m);

    printf_s("Parallel Average (b[]+a[])/2 : \n");
    print4u(result.m64_u16);
    printf_s("\n");
}
```

Output

```
a:
{       10,      100,       70,    10000 }
b:
{        1,       95,       50,     5000 }

Parallel Average (a[]+b[])/2 :
{        5,       97,       60,     7500 }

Parallel Average (b[]+a[])/2 :
{        5,       97,       60,     7500 }
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_pavgsub1

Microsoft Specific

Emits the 1-byte form of the Parallel Average Subtract (**pavgsub1**) instruction.

```
__m64 __m64_pavgsub1(
   __m64 a,
   __m64 b
);
```

Parameters

[in] *a*

An array of 8 bytes interpreted as unsigned 1-byte integers to be subtracted from.

[in] *b*

An array of 8 bytes interpreted as unsigned 1-byte integers to subtract.

Return Value

An **__m64** union containing the array of results of (*a*[*i*]-*b*[*i*])/2 interpreted as signed 8-bit integers.

Requirements

Intrinsic	Architecture
__m64_pavgsub1	IPF

Header file <intrin.h>

Remarks

The instruction computes (*a*[*i*]-*b*[*i*])/2 for each byte *i* of *a* and *b*. Each byte in the array is treated as an 8-bit unsigned integer. If fractional, the result in each byte is rounded to the nearest odd integer.

Example

```
// pavgsub1.cpp
// processor: IPF
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__m64_pavgsub1)

void print8(__int8* ia)
{
    printf_s("{ %4d, %4d, %4d, %4d, %4d, %4d, %4d, %4d}\n",
             ia[0], ia[1], ia[2], ia[3], ia[4], ia[5], ia[6], ia[7] );
}

void print8u(unsigned __int8* ia)
{
    printf_s("{ %4u, %4u, %4u, %4u, %4u, %4u, %4u, %4u }\n",
             ia[0], ia[1], ia[2], ia[3], ia[4], ia[5], ia[6], ia[7] );
}


int main()
{
    __m64 m, n, result;
    __int8 ra[8], i;
    unsigned __int8 ma[8] = { 1, 5, 255, 255, 8, 8, 8, 8 };
    unsigned __int8 na[8] = { 5, 1, 1,   0,   2, 1, 0, 8 };

    for (i = 0; i < 8; i++)
```

```
    {
       m.m64_u8[i] = ma[i];
       n.m64_u8[i] = na[i];
    }

    printf_s("a: \n");
    print8u(ma);
    printf_s("b: \n");
    print8u(na);
    printf_s("\n");

    result = __m64_pavgsub1(m, n);

    printf_s("Parallel Average Subtract (a[]-b[])/2 : \n");
    print8(result.m64_i8);
    printf_s("\n");

    result = __m64_pavgsub1(n, m);

    printf_s("Parallel Average Subtract (b[]-a[])/2 : \n");
    print8(result.m64_i8);
    printf_s("\n");

}
```

Output

```
a:
{    1,    5,  255,  255,    8,    8,    8,    8 }
b:
{    5,    1,    1,    0,    2,    1,    0,    8 }

Parallel Average Subtract (a[]-b[])/2 :
{   -2,    2,  127,  127,    3,    3,    4,    0}

Parallel Average Subtract (b[]-a[])/2 :
{    2,   -2, -127, -127,   -3,   -3,   -4,    0}
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_pavgsub2

Microsoft Specific

Emits the 2-byte form of the Parallel Average Subtract (**pavgsub2**) instruction.

```
__m64 __m64_pavgsub2(
   __m64 a,
   __m64 b
);
```

Parameters

[in] *a*

An **__m64** union containing an array of four unsigned short integers, which are subtracted from.

[in] *b*

An **__m64** union containing an array of four unsigned short integers, which are subtracted.

Return Value

An **__m64** union containing the array of results of (*a*[*i*]-*b*[*i*])/2 interpreted as 16-bit signed integers.

Requirements

Intrinsic	Architecture
__m64_pavgsub2	IPF

Header file <intrin.h>

Remarks

The numbers in the input array *b* are subtracted from the numbers in the input array *a*. The result is an **__m64** union containing an array of unsigned short integers with the result of (*a*[*i*] – *b*[*i*])/2, where *i* numbers the four unsigned integers in the source arrays. If fractional, the result of each average is rounded to the nearest odd integer.

Example

```
// pavgsub2.cpp
// processor: IPF
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__m64_pavgsub2)

void print4u(unsigned __int16* ia)
{
    printf_s("{ %8u, %8u, %8u, %8u }\n", ia[0], ia[1], ia[2], ia[3] );
}

void print4(__int16* ia)
{
    printf_s("{ %8d, %8d, %8d, %8d }\n", ia[0], ia[1], ia[2], ia[3] );
}


int main()
{
    __m64 m, n, result;
    unsigned __int16 ra[4], i;
    unsigned __int16 ma[4] = { 10, 100, 70, 10000 };
    unsigned __int16 na[4] = { 1, 95, 50,   5000 };

    for (i = 0; i < 4; i++)
    {
```

```
        m.m64_u16[i] = ma[i];
        n.m64_u16[i] = na[i];
    }

    printf_s("a: \n");
    print4u(ma);
    printf_s("b: \n");
    print4u(na);
    printf_s("\n");

    result = __m64_pavgsub2(m, n);

    printf_s("Parallel Average Subtract (a[]-b[])/2 : \n");
    print4(result.m64_i16);
    printf_s("\n");

    result = __m64_pavgsub2(n, m);

    printf_s("Parallel Average Subtract (b[]-a[])/2 : \n");
    print4(result.m64_i16);
    printf_s("\n");

}
```

Output

```
a:
{       10,      100,       70,    10000 }
b:
{        1,       95,       50,     5000 }

Parallel Average Subtract (a[]-b[])/2 :
{        5,        3,       10,     2500 }

Parallel Average Subtract (b[]-a[])/2 :
{       -5,       -3,      -10,    -2500 }
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_pmpy2l, __m64_pmpy2r

Microsoft Specific

Emit the two forms of the IPF Parallel Multiply (**pmpy2**) instruction, which performs multiplication in parallel on multiple integers.

```
__m64 __m64_pmpy2l(
   __m64 a,
    __m64 b
);
__m64 __m64_pmpy2r(
   __m64 a,
   __m64 b
);
```

Parameters

[in] *a*

An **__m64** union containing an array of four 16-bit signed integers.

[in] *b*

An **__m64** union containing an array of four 16-bit signed integers.

Return Value

An **__m64** union containing two 32-bit signed integers representing the results of the multiplication.

Requirements

Intrinsic	Architecture
__m64_pmpy2l	IPF
__m64_pmpy2r	IPF

Header file <intrin.h>

Remarks

The **__m64_pmpy2l** (left) intrinsic multiplies *a*[1]*b*[1] and *a*[3]*b*[3]. The **__m64_pmpy2r** (right) intrinsic multiplies *a*[0]*b*[0] and *a*[2]*b*[2]. In both cases, the result is two 32-bit signed integers, which are placed in elements 0 and 1 of the result.

Example

```
// pmpy2.cpp
// processor: IPF
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__m64_pmpy2l, __m64_pmpy2r)

void print16(__int16* ia)
{
    printf_s("{ %8d, %8d, %8d, %8d }\n", ia[0], ia[1], ia[2], ia[3]);
}

void print32(__int32* ia)
{
    printf_s("{ %8d, %8d }\n", ia[0], ia[1]);
}

int main()
{
    __int16 a[4] = { -1, 1, -2, 6000 };
```

```
    __int16 b[4] = { -1, 2, 10, 15000 };
    __int32 cl[2], cr[2], i;
    __m64 m, n, result;

    printf_s("a: \n");
    print16(a);
    printf_s("b: \n");
    print16(b);

    for (int i = 0; i < 4; i++)
    {
       m.m64_i16[i] = a[i];
       n.m64_i16[i] = b[i];
    }

    printf_s("__m64_pmpy2r computes a[0]*b[0] and a[2]*b[2] : ");
    result = __m64_pmpy2r(m, n);
    print32(result.m64_i32);

    result = __m64_pmpy2l(m, n);
    printf_s("__m64_pmpy2l computes a[1]*b[1] and a[3]*b[3] : ");
    print32(result.m64_i32);

 }
```

Output

```
a:
{       -1,         1,        -2,      6000 }
b:
{       -1,         2,        10,     15000 }
__m64_pmpy2r computes a[0]*b[0] and a[2]*b[2] : {         1,       -20 }
__m64_pmpy2l computes a[1]*b[1] and a[3]*b[3] : {         2, 90000000 }
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_pmpyshr2, __m64_pmpyshr2u

Microsoft Specific

Emit the Parallel Multiply and Shift Right (**pmpyshr**) instruction.

```
__m64 __m64_pmpyshr2(
   __m64 a,
   __m64 b,
   const int nBit
);
__m64 __m64_pmpyshr2u(
   __m64 a,
   __m64 b,
   const int nBit
);
```

Parameters

[in] *a*

An **__m64** union containing an array of four 16-bit integers.

[in] *b*

An **__m64** union containing an array of four 16-bit integers.

[in] *nBit*

The number of bits to shift the results of the multiplications, which are truncated to four 16-bit values. Allowed values are 0, 7, 15, or 16.

Requirements

Intrinsic	Architecture
__m64_pmpyshr2	IPF
__m64_pmpyshr2u	IPF

Header file <intrin.h>

Remarks

__m64_pmpyshr2 performs a signed multiplication of the integers in *a* with the corresponding integers in *b* and then a right shift of *nBit* bits; for example, `a[i]*b[i] >> nBit` for elements numbered by *i*; **__m64_pmpyshr2u** is the same but performs an unsigned multiplication.

Example

```
// pmpyshr2.cpp
// processor: IPF
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__m64_pmpyshr2, __m64_pmpyshr2u)

void print16(__int16* ia)
{
    printf_s("{ %#x, %#x, %#x, %#x }\n", ia[0], ia[1], ia[2], ia[3]);
}

void printresult(__int16* hi, __int16* lo)
{
    printf_s("{ 0x%x%x, 0x%x%x 0x%x%x 0x%x%x }\n",
             hi[0], lo[0], hi[1], lo[1],
             hi[2], lo[2], hi[3], lo[3]);
```

```
}

int main()
{
    __int16 a[4] = { 0xffff, 1, 8, 0x5000 };
    __int16 b[4] = { 0xffff, 2, 0x10, 0x5000 };
    int i;
    __m64 m, n, result_lo, result_hi;

    printf_s("a: \n");
    print16(a);
    printf_s("b: \n");
    print16(b);

    for (int i = 0; i < 4; i++)
    {
       m.m64_i16[i] = a[i];
       n.m64_i16[i] = b[i];
    }

    printf_s("__m64_pmpyshr2 computes :\n");
    result_lo = __m64_pmpyshr2(m, n, 0);
    result_hi = __m64_pmpyshr2(m, n, 16);
    printresult(result_hi.m64_i16, result_lo.m64_i16);

    result_lo = __m64_pmpyshr2u(m, n, 0);
    result_hi = __m64_pmpyshr2u(m, n, 16);
    printf_s("__m64_pmpyshr2u computes  :\n");
    printresult(result_hi.m64_i16, result_lo.m64_i16);

}
```

Output

```
a:
{ 0xffffffff, 0x1, 0x8, 0x5000 }
b:
{ 0xffffffff, 0x2, 0x10, 0x5000 }
__m64_pmpyshr2 computes :
{ 0x01, 0x02 0x080 0x19000 }
__m64_pmpyshr2u computes  :
{ 0xfffffffe1, 0x02 0x080 0x19000 }
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_popcnt

Microsoft Specific

Emits the IPF Population Count instruction (**popcnt**), which determines the number of set bits (bits with the value 1) in the input 64-bit *value*.

```
__m64 __m64_popcnt(
   __m64 value
);
```

Parameters

[in] *value*

An **__m64** union whose set bits are to be counted.

Return Value

The number of set (1) bits in the input.

Requirements

Intrinsic	Architecture
__m64_popcnt	IPF

Header file <intrin.h>

Example

```
// popcnt.cpp
// processor: IPF

#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__m64_popcnt)

int main()
{
    __m64 m, result;
    m.m64_i64 = -1;
    result = __m64_popcnt(m);
    printf_s("Population Count of %d (0x%I64x): %I64u\n",
              m.m64_i64, m.m64_i64, result.m64_i64);

    m.m64_u64 = 0xaaaaaaaaaaaaaaaaI64;
    result = __m64_popcnt(m);
    printf_s("Population Count of %u (0x%I64x): %I64u\n",
              m.m64_u64, m.m64_u64, result.m64_u64);
}
```

Output

```
Population Count of -1 (0xffffffffffffffff): 64
Population Count of 2863311530 (0xaaaaaaaaaaaaaaaa): 32
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_pshladd2

Microsoft Specific

Emits the IPF Parallel Shift Left and Add (**pshladd2**) instruction.

```
__m64 __m64_pshladd2(
   __m64 a,
   const int nBit,
   __m64 b
);
```

Parameters

[in] *a*

An **__m64** union containing an array of four 16-bit signed integers to be shifted left and added to.

[in] *nBit*

The number of bits to left shift the integers in the first array. Valid values from 0 to 3.

[in] *b*

An **__m64** union containing an array of four 16-bit signed integers to add.

Return Value

An **__m64** union containing an array of four 16-bit signed integers, the elements of which are computed according to the expression described here.

Requirements

Intrinsic	**Architecture**
__m64_pshladd2	IPF

Header file <intrin.h>

Remarks

This instruction computes:

```
c[i] = a[i] << nBit + b[i]
```

for each of the four elements *i* of *a* and *b* and returns the array *c* as an **__m64** union.

Example

```
// m64_pshladd2.c
// processor: IPF
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__m64_pshladd2)

int main()
{
    __m64 a, b, result;

    a.m64_i16[0] = 0;
    a.m64_i16[1] = 1;
    a.m64_i16[2] = 2;
    a.m64_i16[3] = 100;

    b.m64_i16[0] = 1;
    b.m64_i16[1] = 2;
```

```
    b.m64_i16[2] = 3;
    b.m64_i16[3] = 4;

    printf_s("Input a: %d %d %d %d\n", a.m64_i16[0],
             a.m64_i16[1], a.m64_i16[2],
             a.m64_i16[3]);

    printf_s("Input b: %d %d %d %d\n", b.m64_i16[0],
             b.m64_i16[1], b.m64_i16[2],
             b.m64_i16[3]);

    result = __m64_pshladd2(a, 1, b);

    printf_s("Results of pshladd2(a, 1, b): %d %d %d %d\n",
             result.m64_i16[0],
             result.m64_i16[1], result.m64_i16[2],
             result.m64_i16[3]);
}
```

Output

```
Input a: 0 1 2 100
Input b: 1 2 3 4
Results of pshladd2(a, 1, b): 1 4 7 204
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_pshradd2

Microsoft Specific

Emits the IPF Parallel Shift Right and Add (**pshradd2**) instruction.

```
__m64 __m64_pshradd2(
   __m64 a,
   const int nBit,
   __m64 b
);
```

Parameters

[in] *a*

An **__m64** union containing an array of four 16-bit signed integers to be shifted right and added to.

[in] *nBit*

The number of bits to right shift the integers in the first array. Valid values from 0 to 3.

[in] *b*

An **__m64** union containing an array of four 16-bit signed integers to add.

Return Value

An **__m64** union containing an array of four 16-bit signed integers, the elements of which are computed according to the expression described here.

Requirements

Intrinsic	**Architecture**
__m64_pshradd2	IPF

Header file <intrin.h>

Remarks

This instruction computes:

```
c[i] = a[i] >> nBit + b[i]
```

for each of the four elements *i* of *a* and *b* and returns the array *c* as an **__m64** union.

Example

```
// m64_pshradd2.c
// processor: IPF
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__m64_pshradd2)

int main()
{
    __m64 a, b, result;

    a.m64_i16[0] = 0;
    a.m64_i16[1] = 2;
    a.m64_i16[2] = 16;
    a.m64_i16[3] = 100;

    b.m64_i16[0] = 1;
    b.m64_i16[1] = 2;
```

```
    b.m64_i16[2] = 3;
    b.m64_i16[3] = 4;

    printf_s("Input a: %d %d %d %d\n",
             a.m64_i16[0], a.m64_i16[1], a.m64_i16[2],
             a.m64_i16[3]);

    printf_s("Input b: %d %d %d %d\n",
             b.m64_i16[0], b.m64_i16[1], b.m64_i16[2],
             b.m64_i16[3]);

    result = __m64_pshradd2(a, 1, b);

    printf_s("Results of pshradd2(a, 1, b): %d %d %d %d\n",
             result.m64_i16[0], result.m64_i16[1], result.m64_i16[2],
             result.m64_i16[3]);
}
```

Output

```
Input a: 0 2 16 100
Input b: 1 2 3 4
Results of pshradd2(a, 1, b): 1 3 11 54
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_psub1uus

Microsoft Specific

Emits the 1-byte uus (unsigned, unsigned, signed) saturation form of the IPF Parallel Subtract (**psub1.uus**) instruction.

```
__m64 __m64_psub1uus(
   __m64 a,
   __m64 b
);
```

Parameters

[in] *a*

An **__m64** union containing an array of eight 8-bit unsigned integers.

[in] *b*

An **__m64** union containing an array of eight 8-bit signed integers.

Return Value

An **__m64** union containing an array of eight 8-bit unsigned integers, which are the results of the subtraction of the integers in *b* from the integers in *a*.

Requirements

Intrinsic	**Architecture**
__m64_pshradd2	IPF

Header file <intrin.h>

Remarks

The uus-saturation form means that the first parameter is unsigned, the second parameter is signed, and the return value is unsigned, and if an overflow or underflow occurs, the value saturates at the limit value. The 1-byte form operates on each byte as a separate integer.

Example

```
// psub1uus.cpp
// processor: IPF
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__m64_psub1uus)

int main()
{
    __m64 m, n, result;
    m.m64_i64 = 0xee0100;
    n.m64_i64 = 0x00ff22;  // as the second parameter, 0xff = -1
    result = __m64_psub1uus(m, n);
    printf("__m64_psub1uus(0x%I64x, 0x%I64x) returns 0x%I64x\n",
        m, n, result);

    result = __m64_psub1uus(n, m);
    printf("__m64_psub1uus(0x%I64x, 0x%I64x) returns 0x%I64x\n",
        n, m, result);
}
```

Output

```
__m64_psub1uus(0xee0100, 0xff22) returns 0xee0200
__m64_psub1uus(0xff22, 0xee0100) returns 0x12fe22
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_psub2uus

Microsoft Specific

Emits the 2-byte uus (unsigned, unsigned, signed) saturation form of the IPF Parallel Subtract (**psub2.uus**) instruction.

```
__m64 __m64_psub2uus(
   __m64 a,
   __m64 b
);
```

Parameters

[in] *a*

An **__m64** union containing an array of four 16-bit unsigned integers.

[in] *b*

An **__m64** union containing an array of four 16-bit signed integers.

Return Value

An **__m64** union containing an array of eight 8-bit unsigned integers, which are the results of the subtraction of the integers in *b* from the integers in *a*.

Requirements

Intrinsic	Architecture
__m64_psub2uus	IPF

Header file <intrin.h>

Remarks

The uus-saturation form means that the first parameter is unsigned, the second parameter is signed, and the return value is unsigned, and if an overflow or underflow occurs, the value saturates at the limit value. The 2-byte form operates on each 16-bit block as a separate integer.

Example

```
// psub2uus.cpp
// processor: IPF

#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__m64_psub2uus)

int main()
{
    __m64 m, n, result;
    m.m64_i64 = 0x110022003300FF00;
    n.m64_i64 = 0x01100220033000FF;

    result = __m64_psub2uus(m, n);
    printf_s( "__m64_psub2uus(%#18I64x, %#18I64x) == %#18I64x\n",
              m, n, result);

    // due to underflow (negative result) many of the bytes here
    // will be zero.
    result = __m64_psub2uus(n, m);
    printf_s( "__m64_psub2uus(%#18I64x, %#18I64x) == %#18I64x\n",
              n, m, result);
}
```

Output

```
__m64_psub2uus(0x110022003300ff00,  0x1100220033000ff) ==  0xff01fe02fd0fe01
__m64_psub2uus( 0x1100220033000ff, 0x110022003300ff00) ==              0x1ff
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_shladd

Microsoft Specific

Emits the IPF Shift Left and Add (**shladd**) instruction.

```
__m64 __m64_shladd(
   __m64 a,
   const int nBit,
   __m64 b
);
```

Parameters

[in] *a*

An **__m64** union containing a 64-bit integer.

[in] *nBit*

The number of bits shifted left. Valid values are 1, 2, 3, or 4.

[in] *b*

An **__m64** union containing a 64-bit integer.

Return Value

The result of the left shift of *a* by *nBit* bytes, followed by addition of *b*.

Requirements

Intrinsic	Architecture
__m64_shladd	IPF

Header file <intrin.h>

Remarks

The result computed and returned is *a* * 2^*nBit* + *b*.

Example

```
// shladd.cpp
// processor: IPF
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__m64_shladd)

void print(__int64 a, int nBit, __int64 b, __int64 c)
{
    printf_s("__m64_shladd(%I64d, %d, %I64d) returns %I64d\n",
             a, nBit, b, c);
}

int main()
{
    __m64 m, n, result;
    m.m64_i64 = 15;
    n.m64_i64 = 7;
    result = __m64_shladd(m, 1, n);
    print(m.m64_i64, 1, n.m64_i64, result.m64_i64);

    result = __m64_shladd(m, 2, n);
    print(m.m64_i64, 2, n.m64_i64, result.m64_i64);
```

```
    result = __m64_shladd(m, 3, n);
    print(m.m64_i64, 3, n.m64_i64, result.m64_i64);

    result = __m64_shladd(m, 4, n);
    print(m.m64_i64, 4, n.m64_i64, result.m64_i64);

}
```

Output

```
__m64_shladd(15, 1, 7) returns 37
__m64_shladd(15, 2, 7) returns 67
__m64_shladd(15, 3, 7) returns 127
__m64_shladd(15, 4, 7) returns 247
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__m64_shrp

Microsoft Specific

Emits the IPF Shift Right Pair (**shrp**) instruction.

```
__m64 __m64_shrp(
   __m64 a,
   __m64 b,
   const int nBit
);
```

Parameters

[in] *a*

An **__m64** union containing a 64-bit data field.

[in] *b*

An **__m64** union containing a 64-bit data field.

[in] *nBit*

The number of bits to right shift the concatenated pair. Valid values from 0 to 63.

Return Value

The least significant 64 bits of the concatenated, right-shifted pair.

Requirements

Intrinsic	Architecture
__m64_shrp	IPF

Header file <intrin.h>

Remarks

Two 64-bit data fields are concatenated, *a* being placed in the upper 64 bits and *b* in the lower 64 bits; the resulting 128-bit value is right shifted by nBit bits and the lower 64 bits are returned.

Example

```
// shrp.cpp
// processor: IPF
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__m64_shrp)

int main()
{
    __m64 m, n, result;
    m.m64_u64 = 0x1111111111111111;
    n.m64_u64 = 0x2222222222222222;
    result = __m64_shrp(m, n, 32);
    printf_s("__m64_shrp(0x%I64x, 0x%I64x, %d) returns\n0x%I64x\n",
             m, n, 32, result);

    result = __m64_shrp(m, n, 17);
    printf_s("__m64_shrp(0x%I64x, 0x%I64x, %d) returns\n0x%I64x\n",
             m, n, 17, result);
}
```

Output

```
__m64_shrp(0x1111111111111111, 0x2222222222222222, 32) returns
0x1111111122222222
__m64_shrp(0x1111111111111111, 0x2222222222222222, 17) returns
0x8888911111111111
```

END Microsoft Specific

See Also

Reference

__m64

Compiler Intrinsics

__mf

Microsoft Specific

Emits the IPF Memory Fence (**mf**) instruction, which sets up a hardware memory fence.

```
void __mf(void);
```

Requirements

Intrinsic	Architecture
__mf	IPF

Header file <intrin.h>

Remarks

A hardware memory fence ensures that all memory reads and writes are made visible to all processors prior to any subsequent memory reads and writes. The **__mf** intrinsic does not ensure that these memory reads and writes have been accepted by the platform. If that is a requirement, use __mfa.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

__mfa
_ReadWriteBarrier
Compiler Intrinsics

__mfa

Microsoft Specific

Emits the acceptance form of the IPF Memory Fence (**mfa.a**) instruction, which sets up a hardware memory fence in which all reads and writes must be accepted by the operating system.

```
void __mfa(void);
```

Requirements

Intrinsic	Architecture
__mfa	IPF

Header file <intrin.h>

Remarks

A hardware memory fence ensures that all memory reads and writes are made visible to all processors prior to any subsequent memory reads and writes. For more information, see the Intel IPF documentation.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

__mf

_ReadWriteBarrier

Compiler Intrinsics

_mm_addsub_pd

Microsoft Specific

Generates the **addsubpd** instruction.

```
__m128d _mm_addsub_pd(
   __m128d a,
   __m128d b,
);
```

Parameters

[in] *a*

The first operand.

[in] *b*

The second operand.

Requirements

Intrinsic	Architecture
_mm_addsub_pd	Intel SSE3

Header file <intrin.h>

Remarks

The **addsubpd** instruction performs a packed operation involving double-precision addition and subtraction on 128-bit data. Each 128-bit operand consists of two 64-bit data elements, numbered from 0 to 1, with 1 being the high-order element. The operation subtracts the numbers in the low-order data elements and adds the numbers in the high-order elements. Thus, if the input is *{ A0, A1 }* and *{ B0, B1 }*, then the output is *{ A0–B0, A1+B1 }*.

This routine is only available as an intrinsic.

Example

```
// processor: x86 with SSE3
// Execute the addsub_pd instruction using the intrinsic
// _mm_addsub_pd

#include <stdio.h>
#include <intrin.h>

#pragma intrinsic ( _mm_addsub_pd )

int main( )
{
    __m128d u, v, w;
    __declspec(align(16)) double a[2] = { 0.1, 0.2 };
    __declspec(align(16)) double b[2] = { 0.001, 0.002 };

    printf_s("Loading double values %e %e into XMM register.\n",
             a[0], a[1] );
    u = _mm_load_pd(a);
    printf_s("Loading double values %e %e into XMM register.\n",
             b[0], b[1] );
    v = _mm_load_pd(b);

    printf_s("Calling _mm_addsub_pd to modify these values.\n");
    w = _mm_addsub_pd ( u , v);

    printf_s("Result: %e %e \n", w.m128d_f64[0], w.m128d_f64[1]);
}
```

Output

```
Loading double values 1.000000e-001 2.000000e-001 into XMM register.
Loading double values 1.000000e-003 2.000000e-003 into XMM register.
Calling _mm_addsub_pd to modify these values.
Result: 9.900000e-002 2.020000e-001
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_mm_addsub_ps

Microsoft Specific

Generates the **addsubps** instruction.

```
__m128 _mm_addsub_ps(
   __m128 a,
   __m128 b,
);
```

Parameters

[in] *a*

The first operand.

[in] *b*

The second operand.

Requirements

Intrinsic	Architecture
_mm_addsub_ps	Intel SSE3

Header file <intrin.h>

Remarks

The **addsubps** instruction performs a packed operation involving single-precision addition and subtraction on 128-bit data. Each 128-bit operand consists of four 32-bit data elements, numbered from 0 to 3, with 3 being the high-order element. The operation subtracts the numbers in the even data elements and adds the numbers in the odd data elements. Thus, if the input is *{ A0, A1, A2, A3 }* and *{ B0, B1, B2, B3 }*, then the output is *{ A0-B0, A1+B1, A2-B2, A3+B3 }*.

This routine is only available as an intrinsic.

Example

```
// processor: x86 with SSE3
// Execute the addsub_ps instruction using the PNI intrinsic
// _mm_addsub_ps

#include <stdio.h>
#include <intrin.h>

#pragma intrinsic ( _mm_addsub_ps )

int main( )
{
    __m128 u, v, w;
    __declspec(align(16)) float a[4] = { 0.1, 0.2, 0.3, 0.4 };
    __declspec(align(16)) float b[4] = { 0.0001, 0.002, 0.003, 0.004 };

    printf_s("Loading floating-point values\n"
             "%5.3f %5.3f %5.3f %5.3f into XMM register.\n",
             a[0], a[1], a[2], a[3] );
    u = _mm_load_ps(a);
    printf_s("Loading floating-point values\n "
             "%5.3f %5.3f %5.3f %5.3f into XMM register.\n",
             b[0], b[1], b[2], b[3] );
    v = _mm_load_ps(b);

    printf_s("Calling _mm_addsub_ps to modify these values.\n");
    w = _mm_addsub_ps ( u , v);
```

```
    printf_s("Result: %5.3f %5.3f %5.3f %5.3f\n",
             w.m128_f32[0], w.m128_f32[1],
             w.m128_f32[2], w.m128_f32[3] );
}
```

Output

```
Loading floating-point values
0.100 0.200 0.300 0.400 into XMM register.
Loading floating-point values
0.000 0.002 0.003 0.004 into XMM register.
Calling _mm_addsub_ps to modify these values.
Result: 0.100 0.202 0.297 0.404
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_mm_cvt_pi2ps

Microsoft Specific

Generates the Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values (**cvtpi2ps**) instruction, which converts the two packed 32-bit integers in *b* to single-precision floating-point values.

```
__m128 _mm_cvt_pi2ps(
   __m128 a,
   __m64 b
);
```

Parameters

[in] *a*

An **__m128** structure containing four packed floating-point values.

[in] *b*

An **__m64** structure containing two packed 32-bit integers.

Return Value

An **__m128** structure containing the result of the conversions as the first two elements; the third and fourth elements are copied from *a*.

Requirements

Intrinsic	Architecture
__mm_cvt_pi2ps	x86 with SSE2

Header file <intrin.h>

Remarks

The results are stored in the first two elements of the **__m128** structure returned; the other two elements of the return value are copied from the high quadword of *a*. The **__m128** structure represents an XMM register and the **__m64** structure represents an MMX register, so the instruction generated does not involve system memory.

This routine is only available as an intrinsic.

Example

```
// _mm_cvt_pi2ps.cpp
// processor: x86
#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_cvt_pi2ps)

int main()
{
    __m128 a;
    __m64 b;

    // Aligned because you are using _mm_load_ps to load these
    // numbers into the XMM registers.
    __declspec(align(16)) float af[4] = { 100.5, 200.5,300.5, 400.5 };

    // Load a, which maps to the XMM registers, with the floating point
    //  values. _mm_load_ps requires a 16-byte aligned address.
    a = _mm_load_ps(af);

    // Load b with the two 32-bit signed integers.
    b = _mm_set_pi32(32, 33);
```

```
    // Copy the ints in a to the lower floating point slots of b.
    a = _mm_cvt_pi2ps(a, b);

    // Use the EMMS instruction to signal the transition from XMM
    // registers to floating point registers (used to access the
    // floating point values for printing to stdout).
    _mm_empty();

    // Print out the values in a.
    printf_s("%lf %lf %lf %lf\n",
             a.m128_f32[0], a.m128_f32[1],
             a.m128_f32[2], a.m128_f32[3]);
}
```

Output

```
33.000000 32.000000 300.500000 400.500000
```

END Microsoft Specific

See Also

Reference

__m64

__m128

Compiler Intrinsics

_mm_cvt_ps2pi

Microsoft Specific

Generates the Convert Packed Single Precision Floating Point Values to Packed Doubleword Integers (**cvtps2pi**) instruction, which converts the floating point values in the first two elements in the **__m128** structure *value* to 32-bit integers and returns an **__m64** structure containing the two 32-bit integers.

```
__m64 _mm_cvt_ps2pi(
    __m128 value
);
```

Parameters

[in] *value*

An **__m128** structure containing four single-precision floating point values.

Return Value

An **__m64** structure containing the two 32-bit integers resulting from the conversion.

Requirements

Intrinsic	Architecture
__mm_cvt_ps2pi	x86 with SSE2

Header file <intrin.h>

Remarks

Rounding occurs according the rounding control bits of the MXCSR register. The default rounding mode is round-to-nearest, rounding to the even number if the decimal part is 0.5. If an overflow occurs, 0x80000000 is placed in the appropriate elements of the return value. The **__m128** structure represents an XMM register and the **__m64** structure represents an MMX register, so the instruction generated does not involve system memory.

This routine is only available as an intrinsic.

Example

```
// _mm_cvt_ps2pi.cpp
// processor: x86
#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_cvt_ps2pi)

int main()
{
    __m128 a;
    __m64 b;

    float af[4] = { 101.5, 200.5,300.5, 400.5 };

    // Load a with the floating point values
    // _mm_loadu_ps allowed unaligned addresses
    a = _mm_loadu_ps(af);

    //
    b = _mm_cvt_ps2pi(a);

   _mm_empty();

    // Print out the values in b
    printf_s("%d %d\n", b.m64_i32[0], b.m64_i32[1]);
}
```

Output

```
102 200
```

END Microsoft Specific

See Also

Reference

__m64

__m128d

Compiler Intrinsics

_mm_cvt_si2ss

Microsoft Specific

Generates the Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value (**cvtsi2ss**) instruction, which converts the 32-bit integer *b* to a single-precision floating-point value using the rounding behavior specified by the rounding control bits in MXCSR.

```
__m128 _mm_cvt_si2ss(
   __m128 a,
   int b
);
```

Parameters

[in] *a*

An **__m128** structure containing four single-precision floating point values.

[in] *b*

A 32-bit integer to be converted.

Return Value

An **__m128** structure with the result of the conversion in the first element and the remaining elements copied from *a*.

Requirements

Intrinsic	Architecture
__mm_cvt_si2ss	x86 with SSE2, x64

Header file <intrin.h>

Remarks

The default rounding mode is round to nearest, rounding to the even number if the decimal part is 0.5. The result is placed in the first doubleword of the result; the other three doublewords of the result are copied from *a*. The **__m128** structure represents an XMM register, so this intrinsic allows a value from system memory to be moved into the register.

This routine is only available as an intrinsic.

Example

```
// _mm_cvt_si2ss.cpp
// processor: x86, x64
#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_cvt_si2ss)
// _mm_empty is only available and only required on x86
#ifdef _M_IX86
#pragma intrinsic( _mm_empty)
#endif

int main()
{
    __m128 a;
    int b = 100;

    float af[4] = { 101.25, 200.75,300.5, 400.5 };

    // Load a with the floating point values.
    // The values will be copied to the XMM registers.
    a = _mm_loadu_ps(af);

    // Copy b into the first element of a
```

```
    a = _mm_cvt_si2ss(a, b);

    // Switch back to floating point (x86 only).
#ifdef _M_IX86
    _mm_empty();
#endif

    // Print out the values in a
    printf_s("%lf %lf %lf %lf\n", a.m128_f32[0], a.m128_f32[1],
             a.m128_f32[2], a.m128_f32[3] );
}
```

Output

```
100.000000 200.750000 300.500000 400.500000
```

END Microsoft Specific

See Also

Reference

__m64

__m128d

Compiler Intrinsics

_mm_cvt_ss2si

Microsoft Specific

Generates the Convert Single-Precision Floating-Point Value to Doubleword Integer (**cvtss2si**) instruction.

```
int _mm_cvt_ss2si(
   __m128 value
);
```

Parameters

[in] *value*

An **__m128** structure containing four single-precision floating-point values.

Return Value

An integer representing the result of the conversion.

Requirements

Intrinsic	Architecture
_mm_cvt_ss2si	x86 with SSE2, x64

Header file <intrin.h>

Remarks

The integer is rounded according to the rounding control bits in MXCSR. The default rounding mode is round-to-nearest, rounding to the even number if the decimal part is 0.5. If an overflow occurs, 0x80000000 is returned. The **__m128** structure represents an XMM register, so this intrinsic allows a value from the XMM register to be moved into system memory.

This routine is only available as an intrinsic.

Example

```
// _mm_cvt_ss2si.cpp
// processor: x86, x64
#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_cvt_ss2si)

int main()
{
    __m128 a;
    int b;

    float af[4] = { 101.25, 200.75,300.5, 400.5 };

    // Load a with the floating point values.
    // The values will be copied to the XMM registers.
    a = _mm_loadu_ps(af);

    // Extract the first element of a
    b = _mm_cvt_ss2si(a);

    printf_s("%d\n", b );
}
```

Output

```
101
```

END Microsoft Specific

See Also

Reference

__m64

__m128d

Compiler Intrinsics

_mm_cvtsd_si64x

Microsoft Specific

Generates the x64 extended form of the Convert Scalar Double-Precision Floating-Point Value to 64-Bit Integer (**cvtsd2si**) instruction, which takes the double in the first element of *value* and converts it to a 64-bit integer.

```
__int64 _mm_cvtsd_si64x(
   __m128d value
);
```

Parameters

[in] *value*

An **__m128d** structure containing two double-precision floating-point values.

Return Value

An integer representing the result of the conversion.

Requirements

Intrinsic	Architecture
_mm_cvtsd_si64x	x64

Header file <intrin.h>

Remarks

The rounding control bits in MXCSR are used to determine the rounding behavior. The default rounding mode is round to nearest, rounding to the even number if the decimal part is 0.5. If an overflow occurs, 0x8000000000000000 (9223372036854775808) is returned. The **__m128** structure represents an XMM register, so this intrinsic allows a value from the XMM register to be moved into system memory.

This routine is only available as an intrinsic.

Example

```
// _mm_cvtsd_si64x.cpp
// processor: x64
#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_cvtsd_si64x)

int main()
{
    __m128d d;
    __int64 b;

    double adbl[2] = { 1.8E6, 200.5 };

    // Loads the double values into d
    // (moves data into the XMM registers)
    d = _mm_loadu_pd (adbl);

    // Extract the first element of a
    b = _mm_cvtsd_si64x(d);

    printf_s("%I64d\n", b );
}
```

Output

```
1800000
```

END Microsoft Specific

See Also

Reference

__m128d

Compiler Intrinsics

_mm_cvtsi128_si64x

Microsoft Specific

Generates the x64 extended form of the **movd** instruction, which extracts the low 64-bit integer from an **__m128i** structure.

```
__int64 _mm_cvtsi128_si64x(
   __m128i value
);
```

Parameters

[in] *value*

An **__m128i** structure containing two 64-bit integer values.

Return Value

The low doubleword of the input structure as a 64-bit integer.

Requirements

Intrinsic	Architecture
_mm_cvtsi128_si64x	x64

Header file <intrin.h>

Remarks

The value is copied from the XMM register represented by *value* into system memory. The **__m128** structure represents an XMM register, so this intrinsic allows a value from the XMM register to be moved into system memory.

This routine is only available as an intrinsic.

Example

```
// _mm_cvtsi128_si64x.cpp
// processor: x64
#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_cvtsi128_si64x)

int main()
{
    __declspec(align(16)) __m128i c;
    __int64 b;

    // The following loads into system memory
    c.m128i_i64[0] = 180;
    c.m128i_i64[1] = 210;

    // Load c into the XMM Register
    c = _mm_load_si128 (&c);

    // Perform some operations
    // ...

    // Extract the first element of c back into
    // system memory
    b = _mm_cvtsi128_si64x(c);

   printf_s("%I64d\n", b );
}
```

Output

END Microsoft Specific

See Also

Reference

__m128i

Compiler Intrinsics

_mm_cvtsi64x_sd

Microsoft Specific

Generates the Convert Double Word Integer to Scalar Double-Precision Floating-Point Value (**cvtsi2sd**) instruction.

```
__m128d _mm_cvtsi64x_sd(
   __m128d a
   __int64 b
);
```

Parameters

[in] *a*

An **__m128d** structure containing two double-precision floating-point values.

[in] *b*

A 64-bit integer.

Return Value

An **__m128d** structure with the result of the conversion in the first position, and the second position copied from *a*.

Requirements

Intrinsic	Architecture
_mm_cvtsi64x_sd	AMD64

Header file <intrin.h>

Remarks

The result of the conversion is placed in the first element of the **__m128d** structure passed in as *a*, and the result is returned. The **__m128d** structure represents an XMM register, so this intrinsic allows the value *b* from system memory to be placed in an XMM register.

This routine is only available as an intrinsic.

Example

```
// _mm_cvtsi64x_sd.cpp
// processor: x64

#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_cvtsi64x_sd)

int main()
{
    __m128d a;
    __int64 b = 117;

    double adbl[2] = { 1.8E6, 200.5 };

    // loads the double values into d
    // (moves data into the XMM registers)
    a = _mm_loadu_pd (adbl);

    // Copy b into the first element of a
    a = _mm_cvtsi64x_sd(a, b);

    printf_s("%lf %lf\n", a.m128d_f64[0], a.m128d_f64[1] );
}
```

Output

```
117.000000 200.500000
```

END Microsoft Specific

See Also

Reference

__m128d

Compiler Intrinsics

_mm_cvtsi64x_si128

Microsoft Specific

Generates the x64 extended form of the **movd** instruction, which copies a 64-bit value to a **__m128i** structure, which represents an XMM register.

```
__m128i _mm_cvtsi64x_si128(
   __int64 value
);
```

Parameters

[in] *value*

A 64-bit integer value.

Return Value

An **__m128i** structure containing as the first 64-bit element the input *value*.

Requirements

Intrinsic	Architecture
_mm_cvtsi64x_si128	x64

Header file <intrin.h>

Remarks

The **__m128i** structure represents an XMM register, so this intrinsic moves *value* from system memory into an XMM register.

This routine is only available as an intrinsic.

Example

```
// _mm_cvtsi64x_si128.cpp
// processor: x64
#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_cvtsi64x_si128)

int main()
{
    __m128i a;
    __int64 b = 54;

    a = _mm_cvtsi64x_si128(b);

    printf_s("%I64d\n", a.m128i_i64[0] );
}
```

Output

```
54
```

END Microsoft Specific

See Also

Reference

__m128i

Compiler Intrinsics

_mm_cvtsi64x_ss

Microsoft Specific

Generates the x64 extended version of the Convert 64-Bit Integer to Scalar Single-Precision Floating-Point Value (**cvtsi2ss**) instruction.

```
__m128 _mm_cvtsi64x_ss(
   __m128 a,
   __int64 b
);
```

Parameters

[in] *a*

An **__m128** structure containing four single-precision floating-point values.

[in] *b*

A 64-bit integer to be converted into a floating-point value.

Return Value

An **__m128** structure whose first floating-point value is the result of the conversion. The other three values are copied unchanged from *a*.

Requirements

Intrinsic	Architecture
_mm_cvtsi64x_ss	x64

Header file <intrin.h>

Remarks

The **__m128** structure represents an XMM register, so this intrinsic allows the value *b* from system memory to be moved into an XMM register.

This routine is only available as an intrinsic.

Example

```
// _mm_cvtsi64x_ss.cpp
// processor: x64

#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_cvtsi64x_ss)

int main()
{
    __m128 a;
    __int64 b = 54;

    a.m128_f32[0] = 0;
    a.m128_f32[1] = 0;
    a.m128_f32[2] = 0;
    a.m128_f32[3] = 0;
    a = _mm_cvtsi64x_ss(a, b);

    printf_s( "%lf %lf %lf %lf\n",
              a.m128_f32[0], a.m128_f32[1],
              a.m128_f32[2], a.m128_f32[3] );
}
```

Output

```
54.000000 0.000000 0.000000 0.000000
```

END Microsoft Specific

See Also

Reference

__m128

Compiler Intrinsics

_mm_cvtss_si64x

Microsoft Specific

Generates the x64 extended version of the Convert Scalar Single Precision Floating Point Number to 64-bit Integer (**cvtss2si**) instruction.

```
__int64 _mm_cvtss_si64x(
   __m128 value
);
```

Parameters

[in] *value*

An **__m128** structure containing floating point-values.

Return Value

A 64-bit integer, the result of the conversion of the first floating-point value to an integer.

Requirements

Intrinsic	Architecture
_mm_cvtss_si64x	x64

Header file <intrin.h>

Remarks

The first element of the structure value is converted to an integer and returned. The rounding control bits in MXCSR are used to determine the rounding behavior. The default rounding mode is round to nearest, rounding to the even number if the decimal part is 0.5. Because the **__m128** structure represents an XMM register, this intrinsic takes a value from the XMM register and writes it to system memory.

This routine is only available as an intrinsic.

Example

```
// _mm_cvtss_si64x.cpp
// processor: x64
#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_cvtss_si64x)

int main()
{
    __m128 a;
    __int64 b = 54;

    // _mm_load_ps requires an aligned buffer.
    __declspec(align(16)) float af[4] =
                          { 101.25, 200.75, 300.5, 400.5 };

    // Load a with the floating point values.
    // The values will be copied to the XMM registers.
    a = _mm_load_ps(af);

    // Extract the first element of a and convert to an integer
    b = _mm_cvtss_si64x(a);

    printf_s("%I64d\n", b);
}
```

Output

END Microsoft Specific

See Also

Reference

__m128d

Compiler Intrinsics

_mm_cvtt_ps2pi

Microsoft Specific

Generates the Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers (**cvtps2pi**) instruction, which converts the floating-point values in the first two elements in the **__m128** structure *value* to 32-bit integers and returns an **__m64** structure containing the two 32-bit integers.

```
__m64 _mm_cvtt_ps2pi(
    __m128 value
);
```

Parameters

[in] *value*

An **__m128** structure containing four single-precision floating-point values.

Return Value

An **__m64** structure containing the two 32-bit integers resulting from the conversion.

Requirements

Intrinsic	Architecture
_mm_cvtt_ps2pi	x86 with SSE2

Header file <intrin.h>

Remarks

This intrinsic differs from **_mm_cvt_ps2pi** only in that the floating-point values are truncated rather than rounded according to the setting of the MXCSR register.

Because **__m128** represents an XMM register and **__m64** represents an MMX register, this intrinsic generates an instruction that operates on these registers directly and does not involve system memory.

This routine is only available as an intrinsic.

Example

```
// _mm_cvtt_ps2pi.cpp
// processor: x86

#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_cvtt_ps2pi)

int main()
{
    __m128 a;
    __m64 b;

    float af[4] = { 101.5, 200.5,300.5, 400.5 };

    // Load a with the floating point values
    // _mm_loadu_ps allowed unaligned addresses
    a = _mm_loadu_ps(af);

    //
    b = _mm_cvtt_ps2pi(a);

    // Return to floating point
    _mm_empty();

    // Print out the values in b
```

```
    printf_s("%d %d\n", b.m64_i32[0], b.m64_i32[1]);
}
```

Output

```
101 200
```

END Microsoft Specific

See Also

Reference

__m64

__m128d

Compiler Intrinsics

_mm_cvtt_ss2si

Microsoft Specific

Generates the Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer (**cvttss2si**) instruction.

```
int _mm_cvtt_ss2si(
   __m128 value
);
```

Parameters

[in] *value*

An **__m128** structure containing two single-precision floating-point values.

Return Value

An integer representing the result of the conversion.

Requirements

Intrinsic	Architecture
_mm_cvtt_ss2si	x86 with SSE2, x64

Header file <intrin.h>

Remarks

The first element of the input structure value is converted to an integer and returned. Because the **__m128** structure represents an XMM register, the instruction generated moves data from an XMM register into system memory.

This routine is only available as an intrinsic.

Example

```
// _mm_cvtt_ss2si.cpp
// processor: x86, x64
#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_cvtt_ss2si)

int main()
{
    __m128 a;
    int b;

    float af[4] = { 101.25, 200.75,300.5, 400.5 };

    // Load a with the floating point values.
    // The values will be copied to the XMM registers.
    a = _mm_loadu_ps(af);

    // Extract the first element of a
    b = _mm_cvtt_ss2si(a);

    printf_s("%d\n", b );
}
```

Output

```
101
```

END Microsoft Specific

See Also

Reference

__m128d

Compiler Intrinsics

_mm_cvttsd_si64x

Microsoft Specific

Generates the x64 extended version of the Convert with Truncation Scalar Double-Precision Floating-Point Value to 64-Bit Integer (**cvttsd2si**) instruction, which takes the first double in the input structure of packed doubles, converts it to a 64-bit integer, and returns the result.

```
__int64 _mm_cvttsd_si64x(
   __m128d value
);
```

Parameters

[in] *value*

An **__m128d** structure containing double-precision floating-point values.

Return Value

The result of the conversion of the first double in the input to a 64-bit integer.

Requirements

Intrinsic	Architecture
_mm_cvttsd_si64x	AMD64

Header file <intrin.h>

Remarks

The function differs from **_mm_cvtsd_si64x** only in that inexact conversions are truncated toward zero. If the result overflows, the value 0x8000000000000000 (-9223372036854775808) is returned. Because the **__m128d** structure represents an XMM register, the instruction generated moves data from an XMM register into system memory.

This routine is only available as an intrinsic.

Example

```
// _mm_cvttsd_si64x.cpp
// processor: x64
#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_cvttsd_si64x)

int main()
{
    __m128d d;
    __int64 b;

    double adbl[2] = { 101.5, 200.5 };

    // store the double values into d
    // (moves data into the XMM registers)
    d = _mm_loadu_pd (adbl);

    // Extract the first element of d
    b = _mm_cvttsd_si64x(d);

    printf_s("%I64d\n", b );
}
```

Output

END Microsoft Specific

See Also

Reference

__m128d

Compiler Intrinsics

_mm_cvttss_si64x

Microsoft Specific

Emits the x64 extended version of the Convert with Truncation Single-Precision Floating-Point Number to 64-Bit Integer (**cvttss2si**) instruction.

```
__int64 _mm_cvttss_si64x(
   __m128 value
);
```

Parameters

[in] *value*

An **__m128** structure containing single-precision floating-point values.

Return Value

The result of the conversion of the first floating-point value to a 64-bit integer.

Requirements

Intrinsic	Architecture
_mm_cvttss_si64x	x64

Header file <intrin.h>

Remarks

The intrinsic differs from **_mm_cvtss_si64x** only in that inexact conversions are truncated toward zero. Because the **__m128** structure represents an XMM register, the instruction generated moves data from an XMM register into system memory.

This routine is only available as an intrinsic.

Example

```
// _mm_cvttss_si64x.cpp
// processor: x64
#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_cvttss_si64x)

int main()
{
    __m128 a;
    __int64 b = 54;

    // _mm_load_ps requires an aligned buffer.
    __declspec(align(16)) float af[4] = { 101.5, 200.75,
                                          300.5, 400.5 };

    // Load a with the floating point values.
    // The values will be copied to the XMM registers.
    a = _mm_load_ps(af);

    // Extract the first element of a and convert to an integer
    b = _mm_cvttss_si64x(a);

    printf_s("%I64d\n", b);
}
```

Output

END Microsoft Specific

See Also

Reference

__m128

Compiler Intrinsics

_mm_hadd_pd

Microsoft Specific

Generates the **haddpd** instruction.

```
__m128d _mm_hadd_pd(
   __m128d a,
   __m128d b,
);
```

Parameters

[in] *a*

The first operand.

[in] *b*

The second operand.

Requirements

Intrinsic	Architecture
_mm_hadd_pd	Intel SSE3

Header file <intrin.h>

Remarks

The **haddpd** instruction performs a horizontal add, meaning that adjacent elements in the same operand are added together. Each 128-bit argument is considered as two 64-bit floating-point elements, numbered from 0 to 1, with 1 being the high-order element. The result of the operation on operand *a (A1, A0)* and operand *b (B1, B0)* is *(B1 + B0, A1 + A0)*.

This routine is only available as an intrinsic.

Example

```
// processor: x86 with SSE3
// Execute the hadd_pd instruction using the intrinsic
// _mm_hadd_pd

#include <stdio.h>
#include <intrin.h>

#pragma intrinsic ( _mm_hadd_pd )

int main( )
{
    __m128d u, v, w;
    __declspec(align(16)) double a[2] = { 0.1, 0.2 };
    __declspec(align(16)) double b[2] = { 0.001, 0.002 };

    printf_s("Loading double values %e %e into XMM register.\n",
             a[0], a[1] );
    u = _mm_load_pd(a);
    printf_s("Loading double values %e %e into XMM register.\n",
             b[0], b[1] );
    v = _mm_load_pd(b);

    printf_s("Calling _mm_hadd_pd to modify these values.\n");
    w = _mm_hadd_pd ( u , v);

    printf_s("Result: %e %e \n", w.m128d_f64[0], w.m128d_f64[1]);
}
```

Output

```
Loading double values 1.000000e-001 2.000000e-001 into XMM register.
Loading double values 1.000000e-003 2.000000e-003 into XMM register.
Calling _mm_hadd_pd to modify these values.
Result: 3.000000e-001 3.000000e-003
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_mm_hadd_ps

Microsoft Specific

Generates the **haddps** instruction.

```
__m128 _mm_hadd_ps(
   __m128 a,
   __m128 b,
);
```

Parameters

[in] *a*

The first operand.

[in] *b*

The second operand.

Requirements

Intrinsic	Architecture
_mm_hadd_ps	Intel SSE3

Header file <intrin.h>

Remarks

The **haddps** instruction performs a horizontal add, meaning that adjacent elements in the same operand are added together. Each 128-bit argument is considered as four 32-bit floating-point elements, numbered from 0 to 3, with 3 being the high-order element. The result of the operation on operand *a (A3, A2, A1, A0)* and operand *b (B3, B2, B1, B0)* is *(B3 + B2, B1 + B0, A3 + A2, A1 + A0)*.

This routine is only available as an intrinsic.

Example

```
// processor: x86 with SSE3
// Execute the hadd_ps instruction using the intrinsic
// _mm_hadd_ps

#include <stdio.h>
#include <intrin.h>

#pragma intrinsic ( _mm_hadd_ps )

int main( )
{
    __m128 u, v, w;
    __declspec(align(16)) float a[4] = { 0.1, 0.2, 0.3, 0.4 };
    __declspec(align(16)) float b[4] = { 0.0001, 0.002, 0.003, 0.004 };

    printf_s("Loading floating-point values\n"
             "%5.3f %5.3f %5.3f %5.3f into XMM register.\n ",
             a[0], a[1], a[2], a[3] );
    u = _mm_load_ps(a);
    printf_s("Loading floating-point values\n"
             "%5.3f %5.3f %5.3f %5.3f into XMM register.\n",
             b[0], b[1], b[2], b[3] );
    v = _mm_load_ps(b);

    printf_s("Calling _mm_hadd_ps to modify these values.\n");
    w = _mm_hadd_ps ( u , v);
```

```
    printf_s("Result: %5.3f %5.3f %5.3f %5.3f\n", w.m128_f32[0],
             w.m128_f32[1], w.m128_f32[2], w.m128_f32[3] );
}
```

Output

```
Loading floating-point values
0.100 0.200 0.300 0.400 into XMM register.
Loading floating-point values
0.000 0.002 0.003 0.004 into XMM register.
Calling _mm_hadd_ps to modify these values.
Result: 0.300 0.700 0.002 0.007
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_mm_hsub_pd

Microsoft Specific

Generates the **hsubpd** instruction.

```
__m128d _mm_hsub_pd(
   __m128d a,
   __m128d b,
);
```

Parameters

[in] *a*

The first operand.

[in] *b*

The second operand.

Requirements

Intrinsic	Architecture
_mm_hsub_pd	Intel SSE3

Header file <intrin.h>

Remarks

The **hsubpd** instruction performs a horizontal subtract, meaning that adjacent elements in the same operand are subtracted. Each 128-bit argument is considered as two 64-bit floating-point elements, numbered from 0 to 1, with 1 being the high-order element. The result of the operation on operand *a {A0, A1}* and operand *b {B0, B1}* is *{ A0 - A1, B0 - B1 }*.

This routine is only available as an intrinsic.

Example

```
// processor: x86 with SSE3
// Execute the hsub_pd instruction using the intrinsic
// _mm_hsub_pd

#include <stdio.h>
#include <intrin.h>

#pragma intrinsic ( _mm_hsub_pd )

int main( )
{
    __m128d u, v, w;
    __declspec(align(16)) double a[2] = { 0.5, 0.3 };
    __declspec(align(16)) double b[2] = { 0.005, 0.003 };

    printf_s("Loading double values %e %e into XMM register.\n",
             a[0], a[1] );
    u = _mm_load_pd(a);
    printf_s("Loading double values %e %e into XMM register.\n",
             b[0], b[1] );
    v = _mm_load_pd(b);

    printf_s("Calling _mm_hsub_pd to modify these values.\n");
    w = _mm_hsub_pd ( u , v);

    printf_s("Result: %e %e \n", w.m128d_f64[0], w.m128d_f64[1]);
}
```

Output

```
Loading double values 5.000000e-001 3.000000e-001 into XMM register.
Loading double values 5.000000e-003 3.000000e-003 into XMM register.
Calling _mm_hsub_pd to modify these values.
Result: 2.000000e-001 2.000000e-003
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_mm_hsub_ps

Microsoft Specific

Generates the **hsubps** instruction.

```
__m128 _mm_hsub_ps(
   __m128 a,
   __m128 b,
);
```

Parameters

[in] *a*

The first operand.

[in] *b*

The second operand.

Requirements

Intrinsic	Architecture
_mm_hsub_ps	Intel SSE3

Header file <intrin.h>

Remarks

The **hsubps** instruction performs a horizontal subtract, meaning that adjacent elements in the same operand are subtracted. Each 128-bit argument is considered as four 32-bit floating point elements, numbered from 0 to 3, with 3 being the high-order element. The result of the operation on operand *a { A0, A1, A2, A3 }* and operand *b {B0, B1, B2, B3 }* is *{ A0-A1, A2-A3, B0-B1, B2-B3 }*.

This routine is only available as an intrinsic.

Example

```
// processor: x86 with SSE3
// Execute the hsub_ps instruction using the intrinsic
// _mm_hsub_ps

#include <stdio.h>
#include <intrin.h>

#pragma intrinsic ( _mm_hsub_ps )

int main( )
{
    __m128 u, v, w;
    __declspec(align(16)) float a[4] = { 0.4, 0.3, 0.2, 0.05 };
    __declspec(align(16)) float b[4] = { 0.004, 0.003, 0.002, 0.0005 };

    printf_s("Loading floating-point values\n"
             "%6.4f %6.4f %6.4f %6.4f into XMM register.\n",
             a[0], a[1], a[2], a[3] );
    u = _mm_load_ps(a);
    printf_s("Loading floating-point values\n"
             "%6.4f %6.4f %6.4f %6.4f into XMM register.\n",
             b[0], b[1], b[2], b[3] );
    v = _mm_load_ps(b);

    printf_s("Calling _mm_hsub_ps to modify these values.\n");
    w = _mm_hsub_ps ( u , v);
```

```
    printf_s("Result: %6.4f %6.4f %6.4f %6.4f\n", w.m128_f32[0],
             w.m128_f32[1], w.m128_f32[2], w.m128_f32[3] );
}
```

Output

```
Loading floating-point values
0.4000 0.3000 0.2000 0.0500 into XMM register.
Loading floating-point values
0.0040 0.0030 0.0020 0.0005 into XMM register.
Calling _mm_hsub_ps to modify these values.
Result: 0.1000 0.1500 0.0010 0.0015
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_mm_lddqu_si128

Microsoft Specific

Generates the **lddqu** instruction.

```
__m128i _mm_lddqu_si128(
   __m128i const* Data
);
```

Parameters

[in] *Data*

The data to load.

Requirements

Intrinsic	Architecture
_mm_lddqu_si128	Intel SSE3

Header file <intrin.h>

Return Value

The return value represents a register containing the loaded data.

Remarks

The **lddqu** instruction loads 128-bit data but does some special alignment to prevent cache line splits. If the address of the data is 16-byte aligned, the address is loaded as is. If the address is not 16-byte aligned, the load begins at the highest 16-byte-aligned address less than the address of *Data*. A 32-byte block is loaded, and the requested bytes are extracted. This intrinsic is recommended for improved performance over 128-bit unaligned loads.

This routine is only available as an intrinsic.

Example

```
// processor: x86 with SSE3
// Execute the lddqu instruction using the intrinsic
// _mm_lddqu_si128

#include <stdio.h>
#include <intrin.h>

#pragma intrinsic( _mm_lddqu_si128 )

int main( )
{
    unsigned char pBuf[32];
    __m128i i;
    __m128i j;

    i.m128i_i32[0] = 1;
    i.m128i_i32[1] = 2;
    i.m128i_i32[2] = 3;
    i.m128i_i32[3] = 4;
    memcpy(pBuf+7, &i, 16);

    printf_s("load unaligned data using _mm_lddqu_si128 for best performance");
    j = _mm_lddqu_si128( (const __m128i*)pBuf );
}
```

Output

```
load unaligned data using _mm_lddqu_si128 for best performance
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_mm_loaddup_pd

Microsoft Specific

Generates the **movddup** instruction.

```
__m128d _mm_loaddup_pd(
   const double* value
);
```

Parameters

[in] *value*

The source data.

Requirements

Intrinsic	Architecture
_mm_loaddup_pd	Intel SSE3

Header file <intrin.h>

Return Value

The return value represents the XMM1 register, which will contain copies of the source data in both the upper and the lower halves of the register.

Remarks

The **_mm_loaddup_pd** intrinsic loads the double-precision floating-point number *value* into the lower and upper halves of the XMM1 register, duplicating the value in both halves of the destination register.

Example

```
// processor: x86 with SSE3
// Execute the loaddup_pd instruction using the intrinsic
// _mm_loaddup_pd

#include <stdio.h>
#include <intrin.h>

#pragma intrinsic ( _mm_loaddup_pd )

int main( )
{
    __m128d w;
    double a = 314.1592653589793;

    printf_s("Calling _mm_loaddup_pd to load the value.\n");
    w = _mm_loaddup_pd ( &a );

    printf_s("Result: %e %e \n", w.m128d_f64[0], w.m128d_f64[1]);
}
```

Output

```
Calling _mm_loaddup_pd to load the value.
Result: 3.141593e+002 3.141593e+002
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_mm_monitor

Microsoft Specific

Generates the **monitor** instruction.

```
void _mm_monitor(
   const void* Address,
   unsigned int a,
   unsigned int b
);
```

Parameters

[in] *Address*

Address to monitor.

[in] *a*

Reserved.

[in] *b*

Reserved.

Requirements

Intrinsic	Architecture
_mm_monitor	Intel SSE3

Header file <intrin.h>

Remarks

The **monitor** instruction sets up an address range for hardware monitoring. The values of *a* and *b* correspond to the values in ECX and EDX used by the monitor instruction. They are reserved for future use and should be zero for the SSE3-enabled processor. For more information, see the Intel documentation.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

_mm_mwait

Compiler Intrinsics

_mm_movedup_pd

Microsoft Specific

Generates the **movddup** instruction.

```
__m128d _mm_movedup_pd(
   __m128d value
);
```

Parameters

[in] *value*

The source data.

Requirements

Intrinsic	Architecture
_mm_movedup_pd	Intel SSE3

Header file <intrin.h>

Remarks

The **_mm_movedup_pd** intrinsic moves the double-precision 64-bit value in the lower half of the source operand (memory or XMM register) value into the lower and upper halves of an XMM register, duplicating the value in both halves of the destination register.

Example

```
// processor: x86 with SSE3
// Execute the moveduppd instruction using the intrinsic
// _mm_movedup_pd

#include <stdio.h>
#include <intrin.h>

#pragma intrinsic ( _mm_movedup_pd )

int main( )
{
    __m128d u, w;
    __declspec(align(16)) double a[2] = { 0.1, 0.2 };

    // Load the array a into an XMM register.
    u = _mm_load_pd( a );

    printf_s("Calling _mm_movedup_pd to load the value.\n");
    w = _mm_movedup_pd ( u );

    printf_s("Result: %e %e \n", w.m128d_f64[0], w.m128d_f64[1]);
}
```

Output

```
Calling _mm_movedup_pd to load the value.
Result: 1.000000e-001 1.000000e-001
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_mm_movehdup_ps

Microsoft Specific

Generates the **movshdup** instruction.

```
__m128 _mm_movehdup_ps(
   __m128 source
);
```

Parameters

[in] *source*

The source data.

Requirements

Intrinsic	Architecture
_mm_movehdup_ps	Intel SSE3

Header file <intrin.h>

Return Value

The return value represents an XMM register containing the duplicated elements as described in the Remarks section.

Remarks

The **_mm_movehdup_ps** intrinsic duplicates the second and fourth 32-bit data elements in the *source* data and places the second into the first and second elements, and the fourth element into the third and fourth elements of the return value. Thus, if *source* is *{ A0, A1, A2, A3 }*, the return value is *{ A1, A1, A3, A3 }*.

Example

```
// processor: x86 with SSE3
// Execute the movshdup instruction using the intrinsic
// _mm_movehdup_ps

#include <stdio.h>
#include <intrin.h>

#pragma intrinsic ( _mm_movehdup_ps )

int main( )
{
    __m128 u, v;
    __declspec(align(16)) float f[4] = { 0.1, 0.2, 0.3, 0.4 };

    printf_s("Source data: %f %f %f %f\n", f[0], f[1], f[2], f[3]);

    // Load the array a into an XMM register.
    u = _mm_load_ps( f );

    printf_s("Calling _mm_movedup_ps to duplicate the values.\n");
    v = _mm_movehdup_ps ( u );

    printf_s("Result: %f %f %f %f\n", v.m128_f32[0], v.m128_f32[1],
           v.m128_f32[2], v.m128_f32[3] );
}
```

Output

```
Source data: 0.100000 0.200000 0.300000 0.400000
Calling _mm_movedup_ps to duplicate the values.
Result: 0.200000 0.200000 0.400000 0.400000
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_mm_moveldup_ps

Microsoft Specific

Generates the **movsldup** instruction.

```
__m128 _mm_moveldup_ps(
   __m128 source
);
```

Parameters

[in] *source*

The source data.

Requirements

Intrinsic	Architecture
_mm_moveldup_ps	Intel SSE3

Header file <intrin.h>

Return Value

The return value represents an XMM register containing the duplicated elements as described in the Remarks section.

Remarks

The **_mm_moveldup_ps** intrinsic duplicates the first and third 32-bit data elements in the *source* data and places the first element into the first and second elements, and the third element into the third and fourth elements of the return value. Thus, if *source* is *{ A0, A1, A2, A3 }*, the return value is *{ A0, A0, A2, A2 }*.

Example

```
// processor: x86 with SSE3
// Execute the movsldup instruction using the intrinsic
// _mm_moveldup_ps

#include <stdio.h>
#include <intrin.h>

#pragma intrinsic ( _mm_moveldup_ps )

int main( )
{
    __m128 u, v;
    __declspec(align(16)) float f[4] = { 0.1, 0.2, 0.3, 0.4 };

    printf_s("Source data: %f %f %f %f\n", f[0], f[1], f[2], f[3]);

    // Load the array a into an XMM register
    u = _mm_load_ps( f );

    printf_s("Calling _mm_moveldup_ps to load the values.\n");
    v = _mm_moveldup_ps ( u );

    printf_s("Result: %f %f %f %f\n", v.m128_f32[0], v.m128_f32[1],
             v.m128_f32[2], v.m128_f32[3] );
}
```

Output

```
Source data: 0.100000 0.200000 0.300000 0.400000
Calling _mm_moveldup_ps to load the values.
Result: 0.100000 0.100000 0.300000 0.300000
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_mm_set_epi64x

Microsoft Specific

Returns the **__m128i** structure with its two 64-bit integer values initialized to the values of the two 64-bit integers passed in.

```
__m128i _mm_set_epi64x(
   __int64 i1
...__int64 i2
);
```

Parameters

[in] *i1*

A 64-bit integer to place in the second position of the result.

[in] *i2*

A 64-bit integer to place in the first position of the result.

Return Value

An **__m128i** structure containing two 64-bit integers composed of the input parameters.

Requirements

Intrinsic	Architecture
_mm_set_epi64x	x64

Header file <intrin.h>

Remarks

Because the **__m128i** structure represents an XMM register, this intrinsic provides a way of writing values from system memory to the XMM registers.

This routine is only available as an intrinsic.

Example

```
// _mm_set_epi64x.cpp
// processor: x64
#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_set_epi64x)

int main()
{
    __m128i c;
    __int64 a = 2, b = 54;

    c = _mm_set_epi64x(a, b);

    printf_s("%I64d %I64d\n", c.m128i_i64[0], c.m128i_i64[1]);
}
```

Output

```
54 2
```

END Microsoft Specific

See Also

Reference

__m128d

Compiler Intrinsics

_mm_set1_epi64x

Microsoft Specific

Provides a way to initialize the two 64-bit elements of the **__m128i** structure with two identical integers.

```
__m128i _mm_set1_epi64x(
   __int64 i
);
```

Parameters

[in] *i*

A 64-bit integer to use to initialize the result.

Return Value

An **__m128i** structure whose two 64-bit integer members are initialized with the input parameter.

Requirements

Intrinsic	Architecture
_mm_set1_epi64x	x64

Header file <intrin.h>

Remarks

Because the **__m128i** structure represents an XMM register, this intrinsic provides a way of writing values from system memory to the XMM registers.

This routine is only available as an intrinsic.

Example

```
// _mm_set1_epi64x.cpp
// processor: x64
#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_set1_epi64x)

int main()
{
    __m128i c;
    __int64 a = 2;

    c = _mm_set1_epi64x(a);

    printf_s("%I64d %I64d\n", c.m128i_i64[0], c.m128i_i64[1]);
}
```

Output

```
2 2
```

END Microsoft Specific

See Also

Reference

__m128d

Compiler Intrinsics

_mm_setl_epi64

Microsoft Specific

Returns the lower 64 bits of source argument in the lower 64 bits of the result.

```
__m128i _mm_setl_epi64(
   __m128i Value
);
```

Parameters

[in] *Value*

An **__m128i** structure to use to initialize the result.

Return Value

An **__m128i** structure containing the result.

Requirements

Intrinsic	Architecture
_mm_setl_epi64	x86 with SSE2 support, x64

Header file <intrin.h>

Remarks

The high 64 bits of the result are undefined.

This routine is only available as an intrinsic.

Example

```
// _mm_setl_epi64.cpp
// processor: x86, x64
#include <intrin.h>
#include <stdio.h>

#pragma intrinsic(_mm_setl_epi64)

int main()
{
    __m128i m;

    m.m128i_i64[0] = 100;
    m.m128i_i64[1] = 200;

    m = _mm_setl_epi64(m);
    // Return to floating point (x86 only)
#ifdef _M_IX86
    _mm_empty();
#endif
    printf_s("%I64d \n", m.m128i_i64[0]);
}
```

Output

```
100
```

END Microsoft Specific

See Also

Reference

__m128i

Compiler Intrinsics

_mm_stream_si64x

Microsoft Specific

Writes the data in *Source* to a memory location specified by *Dest*, without polluting the caches.

```
void _mm_stream_si64x(
   __int64 * Dest,
   __int64 Source
);
```

Parameters

[out] *Dest*

A pointer to the location to write the source data to.

[in] *Source*

The data to write.

Requirements

Intrinsic	Architecture
_mm_stream_si64x	x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

Example

```
// _mm_stream_si64x.c
// processor: x64

#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_mm_stream_si64x)

int main()
{
    __int64 val = 0xFFFFFFFFFFFFI64;
    __int64 a[10];

    memset(a, 0, sizeof(a));
    _mm_stream_si64x(a+1, val);
    printf_s( "%I64x %I64x %I64x %I64x", a[0], a[1], a[2], a[3]);
}
```

Output

```
0 fffffffffffff 0 0
```

END Microsoft Specific

See Also

Reference

Cache Support for Streaming SIMD Extensions 2 Integer Operations

Compiler Intrinsics

_mm_mwait

Microsoft Specific

Generates the **mwait** instruction.

```
void _mm_mwait(
   unsigned int a,
   unsigned int b,
);
```

Parameters

[in] *a*

The first operand.

[in] *b*

The second operand.

Requirements

Intrinsic	**Architecture**
_mm_mwait	Intel SSE3

Header file <intrin.h>

Remarks

The **mwait** instruction instructs the processor to enter a wait state in which the processor is instructed to monitor a certain address range waiting for an event or a store to that address range. The values of *a* and *b* are loaded into the ECX and EAX registers. For more information, see the Intel documentation.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

_mm_monitor

Compiler Intrinsics

__movsb

Microsoft Specific

Generates a Move String (**rep movsb**) instruction.

```
void __movsb(
   unsigned char* Destination,
   unsigned const char* Source,
   size_t Count
);
```

Parameters

[out] *Destination*

A pointer to the destination of the copy.

[in] *Source*

A pointer to the source of the copy.

[in] *Count*

The number of bytes to copy.

Requirements

Intrinsic	Architecture
__movsb	x86, x64

Header file <intrin.h>

Remarks

The result is that the first *Count* bytes pointed to by *Source* are copied to the *Destination* string.

This routine is only available as an intrinsic.

Example

```
// movsb.cpp
// processor: x86, x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__movsb)

int main()
{
    unsigned char s1[100];
    unsigned char s2[100] = "A big black dog.";
    __movsb(s1, s2, 100);

    printf_s("%s %s", s1, s2);
}
```

Output

```
A big black dog. A big black dog.
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__movsd

Microsoft Specific

Generates a Move String (**rep movsd**) instruction.

```
void __movsd(
   unsigned long* Dest,
   unsigned long* Source,
   size_t Count
);
```

Parameters

[out] *Dest*

The destination of the operation.

[in] *Source*

The source of the operation.

[in] *Count*

The number of doublewords to copy.

Requirements

Intrinsic	Architecture
__movsd	x86, x64

Header file <intrin.h>

Remarks

The result is that the first *Count* doublewords pointed to by *Source* are copied to the *Dest* string.

This routine is only available as an intrinsic.

Example

```
// movsd.cpp
// processor: x86, x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__movsd)

int main()
{
    unsigned long a1[10];
    unsigned long a2[10] = {950, 850, 750, 650, 550, 450, 350,
                            250, 150, 50};
    __movsd(a1, a2, 10);

    for (int i = 0; i < 10; i++)
        printf_s("%d ", a1[i]);
    printf_s("\n");
}
```

Output

```
950 850 750 650 550 450 350 250 150 50
```

END Microsoft Specific

See Also

Reference

__movsq

Microsoft Specific

Generates a repeated Move String (**rep movsq**) instruction.

```
void __movsq(
   unsigned char* Dest,
   unsigned char* Source,
   size_t Count
);
```

Parameters

[out] *Dest*

The destination of the operation.

[in] *Source*

The source of the operation.

[in] *Count*

The number of quadwords to copy.

Requirements

Intrinsic	Architecture
__movsq	x64

Header file <intrin.h>

Remarks

The result is that the first *Count* quadwords pointed to by *Source* are copied to the *Dest* string.

This routine is only available as an intrinsic.

Example

```
// movsq.cpp
// processor: x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__movsq)

int main()
{
    unsigned __int64 a1[10];
    unsigned __int64 a2[10] = {950, 850, 750, 650, 550, 450, 350, 250,
                               150, 50};
    __movsq(a1, a2, 10);

    for (int i = 0; i < 10; i++)
       printf_s("%d ", a1[i]);
    printf_s("\n");
}
```

Output

```
950 850 750 650 550 450 350 250 150 50
```

END Microsoft Specific

See Also

Reference

__movsw

Microsoft Specific

Generates a Move String (**rep movsw**) instruction.

```
void __movsw(
   unsigned short* Dest,
   unsigned short* Source,
   size_t Count
);
```

Parameters

[out] *Dest*

The destination of the operation.

[in] *Source*

The source of the operation.

[in] *Count*

The number of words to copy.

Requirements

Intrinsic	Architecture
__movsw	x86, x64

Header file <intrin.h>

Remarks

The result is that the first *Count* words pointed to by *Source* are copied to the *Dest* string.

This routine is only available as an intrinsic.

Example

```
// movsw.cpp
// processor: x86, x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__movsw)

int main()
{
    unsigned short s1[10];
    unsigned short s2[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    __movsw(s1, s2, 10);

    for (int i = 0; i < 10; i++)
        printf_s("%d ", s1[i]);
    printf_s("\n");
}
```

Output

```
0 1 2 3 4 5 6 7 8 9
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__mul128

Microsoft Specific

Multiplies two 64-bit integers passed in as the first two arguments and puts the high 64 bits of the product in the 64-bit integer pointed to by *HighProduct* and returns the low 64 bits of the product.

```
__int64 _mul128(
   __int64 Multiplier,
   __int64 Multiplicand,
   __int64 *HighProduct
);
```

Parameters

[in] *Multiplier*

The first 64-bit integer to multiply.

[in] *Multiplicand*

The second 64-bit integer to multiply.

[out] *HighProduct*

The high 64 bits of the product.

Return Value

The low 64 bits of the product.

Requirements

Intrinsic	Architecture
__mul128	IPF, x64

Header file <intrin.h>

Example

```
// mul128.c
// processor: IPF, x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_mul128)

int main()
{
    __int64 a = 0x0fffffffffffffffI64;
    __int64 b = 0xf0000000I64;
    __int64 c, d;

    d = _mul128(a, b, &c);

    printf_s("%#I64x * %#I64x = %#I64x%I64x\n", a, b, c, d);
}
```

Output

```
0xfffffffffffffff * 0xf0000000 = 0xeffffffffffffff10000000
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__mulh

Microsoft Specific

Returns the high 64 bits of the product of two 64-bit signed integers.

```
__int64 __mulh(
   __int64 a,
   __int64 b
);
```

Parameters

[in] *a*

The first number to multiply.

[in] *b*

The second number to multiply.

Return Value

The high 64 bits of the 128-bit result of the multiplication.

Requirements

Intrinsic	Architecture
__mulh	x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

Example

```
// mulh.cpp
// processor: x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic (__mulh)

int main()
{
    __int64 a = 0x0fffffffffffffffI64;
    __int64 b = 0xf0000000I64;

    __int64 result = __mulh(a, b); // the high 64 bits
    __int64 result2 = a * b; // the low 64 bits

    printf_s(" %#I64x * %#I64x = %#I64x%I64x\n",
             a, b, result, result2);
}
```

Output

```
0xffffffffffffffff * 0xf0000000 = 0xeffffffffffffff10000000
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__noop

Microsoft Specific

The **__noop** intrinsic specifies that a function should be ignored and the argument list be parsed but no code be generated for the arguments. It is intended for use in global debug functions that take a variable number of arguments.

Example

The following code shows how you could use **__noop**.

```
// compiler_intrinsics__noop.cpp
// compile with or without /DDEBUG
#include <stdio.h>

#if DEBUG
   #define PRINT   printf_s
#else
   #define PRINT   __noop
#endif

int main() {
   PRINT("\nhello\n");
}
```

See Also

Reference

Compiler Intrinsics

C++ Keywords

__outbyte

Microsoft Specific

Generates the **out** instruction, which sends 1 byte specified by *Data* out the I/O port specified by *Port*.

```
void __outbyte(
   unsigned short Port,
   unsigned char Data
);
```

Parameters

[in] *Port*

The port to send the data to.

[in] *Data*

The byte to be sent out the specified port.

Requirements

Intrinsic	Architecture
__outbyte	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__outbytestring

Microsoft Specific

Generates the **rep outsb** instruction,which sends the first *Count* bytes of data pointed to by *Buffer* to the port specified by *Port*.

```
void __outbytestring(
   unsigned short Port,
   unsigned char* Buffer,
   unsigned long Count
);
```

Parameters

[in] *Port*

The port to send the data to.

[in] *Buffer*

The data to be sent out the specified port.

[in] *Count*

The number of bytes of data to be sent.

Requirements

Intrinsic	**Architecture**
__outbytestring	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__outdword

Microsoft Specific

Generates the **out** instruction to send a doubleword *Data* out the port *Port*.

```
void __outdword(
   unsigned short Port,
   unsigned long Data
);
```

Parameters

[in] *Port*

The port to send the data to.

[in] *Data*

The doubleword to be sent.

Requirements

Intrinsic	Architecture
__outdword	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__outdwordstring

Microsoft Specific

Generates the **rep outsd** instruction, which sends *Count* doublewords starting at *Buffer* out the I/O port specified by *Port*.

```
void __outdwordstring(
   unsigned short Port,
   unsigned long* Buffer,
   unsigned long Count
);
```

Parameters

[in] *Port*

The port to send the data to.

[in] *Buffer*

A pointer to the data to be sent out the specified port.

[in] *Count*

The number of doublewords to send.

Requirements

Intrinsic	**Architecture**
__outdwordstring	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__outword

Microsoft Specific

Generates the **out** instruction, which sends the word *Data* out the I/O port specified by *Port*.

```
void __outword(
   unsigned short Port,
   unsigned short Data
);
```

Parameters

[in] *Port*

The port to send the data to.

[in] *Data*

The data to be sent.

Requirements

Intrinsic	Architecture
__outword	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__outwordstring

Microsoft Specific

Generates the **rep outsw** instruction, which sends *Count* words starting at *Buffer* out the I/O port specified by *Port*.

```
void __outwordstring(
   unsigned short Port,
   unsigned short* Buffer,
   unsigned long Count
);
```

Parameters

[in] *Port*

The port to send the data to.

[in] *Buffer*

A pointer to the data to be sent out the specified port.

[in] *Count*

The number of words to send.

Requirements

Intrinsic	**Architecture**
__outwordstring	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__ptcg, __ptcga

Microsoft Specific

Emits Purge Global Translation Cache (**ptc.g**), which is used to purge a range of addresses caused by a page teardown.

```
void __ptcg(
   __int64 Address,
      __int64 Range
);
void __ptcga(
   __int64 Address,
      __int64 Range
);
```

Parameters

[in] *Address*

Loaded into the first general register supplied to the instruction. Bits 0 to 60 specify the virtual address to purge. Bits 61 to 63 specify the region identifier.

[in] *Range*

Loaded into the second general register supplied to the instruction. Bits 2 to 7 represent the page size; other bits are ignored.

Requirements

Intrinsic	Architecture
__ptcg	IPF
__ptcga	IPF

Header file <intrin.h>

Remarks

The **__ptcga** intrinsic emits the **ptc.ga** instruction, which is used if the purge is needed because of a page remapping to ensure that other processors update their advanced load address table (ALAT).

For appropriate use of these instructions, see the Intel IPF documentation.

These functions are only available in kernel mode and are only available as intrinsics.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__ptcl

Microsoft Specific

Emits Purge Local Translation Cache (**ptc.l**), which is used to purge a range of addresses caused by a page teardown.

```
void __ptcl(
   __int64 Address,
      __int64 Range
);
```

Parameters

[in] *Address*

Loaded into the first general register supplied to the instruction. Bits 0 to 60 specify the virtual address to purge. Bits 61 to 63 specify the region identifier.

[in] *Range*

Loaded into the second general register supplied to the instruction. Bits 2 to 7 represent the page size; other bits are ignored.

Requirements

Intrinsic	Architecture
__ptcl	IPF

Header file <intrin.h>

Remarks

The difference between **__ptcl** and **__ptcg** is that **__ptcl** affects the local processor, where **__ptcg** affects all processors in the coherence domain.

For appropriate use of these instructions, see the Intel IPF documentation.

These functions are only available in kernel mode and are only available as intrinsics.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__ptrd, __ptri

Microsoft Specific

Emit the IPF Purge Translation Register (**ptr**) instruction.

```
void __ptrd(
   __int64 Address,
      __int64 Range
);
void __ptri(
   __int64 Address,
      __int64 Range
);
```

Parameters

[in] *Address*

Loaded into the first general register supplied to the instruction. Bits 0 to 60 specify the virtual address to purge. Bits 61 to 63 specify the region identifier.

[in] *Range*

Loaded into the second general register supplied to the instruction. Bits 2 to 7 represent the page size; other bits are ignored.

Requirements

Intrinsic	Architecture
__ptrd	IPF
__ptri	IPF

Header file <intrin.h>

Remarks

The instruction has two forms: **ptr.d** (data) and **ptr.i** (instruction). The intrinsics emit the similarly named form of the instruction. Unlike **__ptcg** and **__ptcga**, the operation is local to the processor.

For more information, see the Intel IPF documentation. These routines are only available as intrinsics.

END Microsoft Specific

See Also

Reference

__ptcg, __ptcga

Compiler Intrinsics

_rdteb

Microsoft Specific

Reads the Thread Pointer register (**IntR13**) and returns its value, which is the base address of the current thread environment block (TEB).

```
void* _rdteb(void);
```

Return Value

The thread pointer.

Requirements

Intrinsic	Architecture
_rdteb	IPF

Remarks

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__rdtsc

Microsoft Specific

Generates the **rdtsc** instruction, which returns the processor time stamp. The processor time stamp records the number of clock cycles since the last reset.

```
unsigned __int64 __rdtsc(void);
```

Return Value

A 64-bit unsigned integer representing a tick count.

Requirements

Intrinsic	Architecture
__rdtsc	x86, x64

Header file <intrin.h>

Remarks

This routine is only available as an intrinsic.

Example

```
// rdtsc.cpp
// processor: x86, x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__rdtsc)

int main()
{
    unsigned __int64 i;
    i = __rdtsc();
    printf_s("%I64d ticks\n", i);
}
```

Sample Output

```
3363423610155519 ticks
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__readcr0

Microsoft Specific

Reads the CR0 register and returns its value.

```
unsigned __int64 __readcr0(void);
```

Return Value

The value in the CR0 register.

Requirements

Intrinsic	Architecture
__readcr0	x86, x64

Header file <intrin.h>

Remarks

This intrinsic is only available in kernel mode, and the routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__readcr2

Microsoft Specific

Reads the CR2 register and returns its value.

```
unsigned __int64 __readcr2(void);
```

Return Value

The value in the CR2 register.

Requirements

Intrinsic	Architecture
__readcr2	x86, x64

Header file <intrin.h>

Remarks

This intrinsic is only available in kernel mode, and the routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__readcr3

Microsoft Specific

Reads the CR3 register and returns its value.

```
unsigned __int64 __readcr3(void);
```

Return Value

The value in the CR3 register.

Requirements

Intrinsic	Architecture
__readcr3	x86, x64

Header file <intrin.h>

Remarks

This intrinsic is only available in kernel mode, and the routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__readcr4

Microsoft Specific

Reads the CR4 register and returns its value.

```
unsigned __int64 __readcr4(void);
```

Return Value

The value in the CR4 register.

Requirements

Intrinsic	Architecture
__readcr4	x86, x64

Header file <intrin.h>

Remarks

This intrinsic is only available in kernel mode, and the routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__readcr8

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Reads the CR8 register and returns its contents.

```
unsigned __int64 __readcr8(void);
```

Return Value

The contents of the CR8 register.

Requirements

Intrinsic	Architecture
__readcr8	x86, x64

Header file <intrin.h>

Remarks

This intrinsic is available only in kernel mode, and the routine is available only as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__readfsbyte, __readfsdword, __readfsqword, __readfsword

Microsoft Specific

Read memory from a location specified by an offset relative to the beginning of the FS segment.

```
unsigned char __readfsbyte(
   unsigned long Offset
);
unsigned short __readfsword(
   unsigned long Offset
);
unsigned long __readfsdword(
   unsigned long Offset
);
unsigned __int64 __readfsqword(
   unsigned long Offset
);
```

Parameters

[in] *Offset*

The offset from the beginning of **FS** to read from.

Return Value

The memory contents of the byte, word, doubleword, or quadword (as indicated by the name of the function called) at the location **FS:[***Offset***]**.

Requirements

Intrinsic	Architecture
__readfsbyte	x86
__readfsdword	x86
__readfsqword	x86
__readfsword	x86

Header file <intrin.h>

Remarks

These intrinsics are only available in kernel mode, and the routines are only available as intrinsics.

END Microsoft Specific

See Also

Reference

__writefsbyte, __writefsdword, __writefsqword, __writefsword

Compiler Intrinsics

__readgsbyte, __readgsdword, __readgsqword, __readgsword

Microsoft Specific

Read memory from a location specified by an offset relative to the beginning of the GS segment.

```
unsigned char __readgsbyte(
   unsigned long Offset
);
unsigned short __readgsword(
   unsigned long Offset
);
unsigned long __readgsdword(
   unsigned long Offset
);
unsigned __int64 __readgsqword(
   unsigned long Offset
);
```

Parameters

[in] *Offset*

The offset from the beginning of **GS** to read from.

Return Value

The memory contents of the byte, word, double word, or quadword (as indicated by the name of the function called) at the location **GS:[***Offset***]**.

Requirements

Intrinsic	Architecture
__readgsbyte	x64
__readgsdword	x64
__readgsqword	x64
__readgsword	x64

Header file <intrin.h>

Remarks

These intrinsics are only available in kernel mode, and the routines are only available as intrinsics.

END Microsoft Specific

See Also

Reference

__writegsbyte, __writegsdword, __writegsqword, __writegsword

Compiler Intrinsics

_ReadBarrier

Microsoft Specific

Forces reads from memory to complete at the point of the call.

```
void _ReadBarrier(void);
```

Requirements

Intrinsic	Architecture
_ReadBarrier	x86, IPF, x64

Header file <intrin.h>

Remarks

The _ReadBarrier, _WriteBarrier, and _ReadWriteBarrier functions help ensure the proper operation of multithreaded programs that are optimized by the Visual C++ compiler. A correctly optimized program yields the same results when it executes on multiple threads as when it executes on a single thread.

To help ensure that the optimized program operates correctly, the _ReadBarrier function forces reads from memory to complete at the point of the call. After the call, other threads can access the memory without fear that the thread that made the call might have a pending read from the memory. For more information, see the remarks in _ReadWriteBarrier.

END Microsoft Specific

See Also

Reference

_ReadWriteBarrier
_WriteBarrier
Compiler Intrinsics
C++ Keywords

__readmsr

Microsoft Specific

Generates the **rdmsr** instruction, which reads the model-specific register specified by *register* and returns its value.

```
__int64 __readmsr(
   int register
);
```

Parameters

[in] *register*

The model specific register to read.

Return Value

The value in the specified register.

Requirements

Intrinsic	Architecture
__readmsr	x86, x64

Header file <intrin.h>

Remarks

This function is only available in kernel mode, and the routine is only available as an intrinsic.

For more information, see the AMD documentation.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__readpmc

Microsoft Specific

Generates the **rdpmc** instruction, which reads the performance monitoring counter specified by *counter*.

```
unsigned __int64 __readpmc(
   unsigned long counter
);
```

Parameters

[in] *counter*

The performance counter to read.

Return Value

The value of the specified performance counter.

Requirements

Intrinsic	Architecture
__readpmc	x86, x64

Header file <intrin.h>

Remarks

This intrinsic is available in kernel mode only, and the routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_ReadWriteBarrier

Microsoft Specific

Forces reads and writes to memory to complete at the point of the call.

```
void _ReadWriteBarrier(void);
```

Requirements

Intrinsic	Architecture
_ReadWriteBarrier	x86, IPF, x64

Header file <intrin.h>

Remarks

The _ReadBarrier, _WriteBarrier, and _ReadWriteBarrier functions help ensure the correct operation of multithreaded programs that are optimized by the Visual C++ compiler. A correctly optimized program yields the same results when it executes on multiple threads as when it executes on a single thread.

The point in an application where a _ReadBarrier, _WriteBarrier, or _ReadWriteBarrier function executes is called a memory barrier. A memory barrier can be for reads, writes, or both.

An instruction that accesses a variable in memory might be deleted or moved across a memory barrier as part of an optimization. Consequently, a thread might read an old value from a global variable before another thread completes writing a new value to the variable, or write a new value before another thread completes reading an old value from the variable.

To help ensure that the optimized program operates correctly, the _ReadWriteBarrier function forces reads and writes to memory to complete at the point of the call. After the call, other threads can access the memory without fear that the thread that made the call might have a pending read or write to the memory. A memory barrier prevents the compiler from optimizing memory accesses across the barrier, but enables the compiler to still optimize instructions between barriers.

Marking memory with a memory barrier is similar to marking memory with the volatile (C++) keyword. However, a memory barrier is more efficient because reads and writes are forced to complete at specific points in the program rather than globally. The optimizations that can occur if a memory barrier is used cannot occur if the variable is declared volatile.

Note In past versions of the Visual C++ compiler, the _ReadWriteBarrier and _WriteBarrier functions were enforced only locally and did not affect functions up the call tree. In Visual C++ 2005 and later, these functions are enforced all the way up the call tree.

Affected Memory

Global variables are affected by memory barriers, but typically local variables are not. Most local variables are not accessible to other threads and therefore do not need protection. A variable is affected by a memory barrier if it satisfies one of the following conditions:

- The variable is a global variable.
- The variable is a local variable used in a **__try**, **__except**, or **__finally** block if structured exception handling is used, or a catch block if C++ exception handling is used. For more information, see the /EHa compiler option.
- The variable is a local variable that is declared **volatile**.
- The variable is a local variable whose address escapes the current function in some way. For example, the variable is passed by reference to another function or its address is assigned to a global variable.
- The variable is accessed indirectly through a pointer and the dereferenced pointer satisfies one of the previous conditions. The most typical case is `*p` where `p` is a global variable or parameter.

Example

```
// intrinsics_readwritebarrier.c
// compile with: /O2 -DNO_BARRIER
// This code contains an error--dereferencing a null pointer--
```

```
// which will be optimized away as a useless assignment.
// Omit the NO_BARRIER command line to activate the Write Barrier.
// With the barrier activated, the assignment is not optimized away
// and causes an access violation.

#include <windows.h> // for EXCEPTION_ACCESS_VIOLATION
#include <excpt.h>
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_ReadWriteBarrier)

int x = 0;

__declspec(noinline) int f(int* p)
{
    x = *p;
#ifndef NO_BARRIER
    _ReadWriteBarrier();
#endif
    x = 7;
    return x;
}


// If code is EXCEPTION_ACCESS_VIOLATION it means an
// attempt to read from the NULL pointer we passed in, so
// we handle the exception.
int filter(unsigned int code, struct _EXCEPTION_POINTERS *ep)
{
    if (code == EXCEPTION_ACCESS_VIOLATION)
    {
        printf_s("AV\n");
        return EXCEPTION_EXECUTE_HANDLER;
    }

    // If not what we were looking for, we don't want to handle it.
    return EXCEPTION_CONTINUE_SEARCH;
}

int main()
{
    int nRet = 0;

    __try
    {
        // Should return only if the first assignment is
        // optimized away.
        nRet = f(NULL);
        printf_s("Assignment was optimized away!\n");
    }
    __except(filter(GetExceptionCode(), GetExceptionInformation()))
    {
        // We get here if an Access violation occurred.
        printf_s("Access Violation: assignment was not optimized away.\n");
    }
}
```

Output

```
Assignment was optimized away!
```

END Microsoft Specific

See Also

Reference

_ReadBarrier

_ReadWriteBarrier

Compiler Intrinsics

_ReleaseSpinLock

Microsoft Specific

Used to leave a critical section that was entered using **_AcquireSpinLock**, which signals that it is now safe to allow another thread to execute the critical section.

```
void _ReleaseSpinLock(
   unsigned __int64 * Lock
);
```

Parameters
[out] *Lock*

The lock variable, which is the same variable used in the **_AcquireSpinLock** function.

Requirements

Intrinsic	Architecture
_ReleaseSpinLock	IPF

Header file <intrin.h>

Remarks

Other threads attempting to execute **_AcquireSpinLock** with the same variable see zero in the Lock variable until the thread that has the lock calls **_ReleaseSpinLock** on that variable. **_AcquireSpinLock** and **_ReleaseSpinLock** must occur in pairs that use the same address.

This routine is only available as an intrinsic.

Example

For more information, see _AcquireSpinLock.

END Microsoft Specific
See Also
Reference
Compiler Intrinsics

_ReturnAddress

Microsoft Specific

The **_ReturnAddress** intrinsic provides the address of the instruction in the calling function that will be executed after control returns to the caller.

Build the following program and step through it in the debugger. As you step through the program, note the address that is returned from **_ReturnAddress**. Then, immediately after returning from the function where **_ReturnAddress** was used, open the How to: Use the Disassembly Window and note that the address of the next instruction to be executed matches the address returned from **_ReturnAddress**.

Optimizations such as inlining may affect the return address. For example, if the sample program below is compiled with /Ob1, `inline_func` will be inlined into the calling function, `main`. Therefore, the calls to **_ReturnAddress** from `inline_func` and `main` will each produce the same value.

When **_ReturnAddress** is used in a program compiled with /clr, the function containing the **_ReturnAddress** call will be compiled as a native function. When a function compiled as managed calls into the function containing **_ReturnAddress**, **_ReturnAddress** may not behave as expected.

Requirements

Header file <intrin.h>

Example

```
// compiler_intrinsics__ReturnAddress.cpp
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_ReturnAddress)

__declspec(noinline)
void noinline_func(void)
{
   printf("Return address from %s: %p\n", __FUNCTION__, _ReturnAddress());
}

__forceinline
void inline_func(void)
{
   printf("Return address from %s: %p\n", __FUNCTION__, _ReturnAddress());
}

int main(void)
{
   noinline_func();
   inline_func();
   printf("Return address from %s: %p\n", __FUNCTION__, _ReturnAddress());

   return 0;
}
```

END Microsoft Specific

See Also

Reference

_AddressOfReturnAddress

Compiler Intrinsics

C++ Keywords

_rotl8, _rotl16

Microsoft Specific

Rotate the input values to the left to the most significant bit (MSB) by a specified number of bit positions.

```
unsigned char _rotl8(
   unsigned char value,
   unsigned char shift
);
unsigned short _rotl16(
   unsigned short value,
   unsigned char shift
);
```

Parameters

[in] *value*

The value to rotate.

[in] *shift*

The number of bits to rotate.

Return Value

The rotated value.

Requirements

Intrinsic	Architecture
_rotl8	x86, IPF, x64
_rotl16	x86, IPF, x64

Header file <intrin.h>

Remarks

Unlike a left-shift operation, when executing a left rotation, the high order bits that fall off the high end are moved into the least significant bit positions.

Example

```
// rotl.cpp
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_rotl8, _rotl16)

int main()
{
    unsigned char c = 'A', c1, c2;

    for (int i = 0; i < 8; i++)
    {
       printf_s("Rotating 0x%x left by %d bits gives 0x%x\n", c,
               i, _rotl8(c, i));
    }

    unsigned short s = 0x12;
    int nBit = 10;

    printf_s("Rotating unsigned short 0x%x left by %d bits gives 0x%x\n",
            s, nBit, _rotl16(s, nBit));
```

```
}
```

Output

```
Rotating 0x41 left by 0 bits gives 0x41
Rotating 0x41 left by 1 bits gives 0x82
Rotating 0x41 left by 2 bits gives 0x5
Rotating 0x41 left by 3 bits gives 0xa
Rotating 0x41 left by 4 bits gives 0x14
Rotating 0x41 left by 5 bits gives 0x28
Rotating 0x41 left by 6 bits gives 0x50
Rotating 0x41 left by 7 bits gives 0xa0
Rotating unsigned short 0x12 left by 10 bits gives 0x4800
```

END Microsoft Specific

See Also

Reference

_rotr8, _rotr16

Compiler Intrinsics

_rotr8, _rotr16

Microsoft Specific

Rotate the input values to the right to the least significant bit (LSB) by a specified number of bit positions.

```
unsigned char _rotr8(
   unsigned char value,
   unsigned char shift
);
unsigned short _rotr16(
   unsigned short value,
   unsigned char shift
);
```

Parameters

[in] *value*

The value to rotate.

[in] *shift*

The number of bits to rotate.

Return Value

The rotated value.

Requirements

Intrinsic	Architecture
_rotr8	x86, IPF, x64
_rotr16	x86, IPF, x64

Header file <intrin.h>

Remarks

Unlike a right-shift operation, when executing a right rotation, the low order bits that fall off the low end are moved into the high order bit positions.

Example

```
// rotr.cpp
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_rotr8, _rotr16)

int main()
{
    unsigned char c = 'A', c1, c2;

    for (int i = 0; i < 8; i++)
    {
       printf_s("Rotating 0x%x right by %d bits gives 0x%x\n", c,
                i, _rotr8(c, i));
    }

    unsigned short s = 0x12;
    int nBit = 10;

    printf_s("Rotating unsigned short 0x%x right by %d bits "
             "gives 0x%x\n",
```

```
            s, nBit, _rotr16(s, nBit));
}
```

Output

```
Rotating 0x41 right by 0 bits gives 0x41
Rotating 0x41 right by 1 bits gives 0xa0
Rotating 0x41 right by 2 bits gives 0x50
Rotating 0x41 right by 3 bits gives 0x28
Rotating 0x41 right by 4 bits gives 0x14
Rotating 0x41 right by 5 bits gives 0xa
Rotating 0x41 right by 6 bits gives 0x5
Rotating 0x41 right by 7 bits gives 0x82
Rotating unsigned short 0x12 right by 10 bits gives 0x480
```

END Microsoft Specific

See Also

Reference

_rotl8, _rotl16

Compiler Intrinsics

__rsm

Microsoft Specific

Emits the IPF Reset System Mask (**rsm**) instruction.

```
void __rsm(
   int Mask
);
```

Parameters

[in] *Mask*

The value whose complement is ANDed with the system mask. Must be a compile time constant.

Requirements

Intrinsic	Architecture
__rsm	IPF

Header file <intrin.h>

Remarks

The complement of the *Mask* parameter is used with an **AND** operation with the system mask to determine the new system mask value.

This intrinsic is available in kernel mode only, and the routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

__ssm

Compiler Intrinsics

__rum

Microsoft Specific

Emits the IPF Reset User Mask (**rum**) instruction.

```
void __rum(
   int Mask
);
```

Parameters
[in] *Mask*

The 6-bit value whose complement is ANDed with the user mask. Must be a compile time constant.

Requirements

Intrinsic	Architecture
__rum	IPF

Header file <intrin.h>

Remarks

The complement of the *Mask* parameter is used with an **AND** operation with the user mask to determine the new user mask value.

The following table shows the bits that can be set in the user mask.

Bit	Mnemonic	Description	Meaning when clear	Meaning when set
0	--	Reserved	--	--
1	be	Big Endian Data Memory References	Little endian.	Big endian.
2	up	User Performance Monitoring	Disabled.	Enabled.
3	ac	Alignment Check (of data memory references)	Check alignment when needed.	Always check alignment.
4	mfl	Lower Floating-Point Registers (f2 to f31) Written	A write to f2 to f31 occurred.	No writes to f2 to f31 have occurred.
5	mfh	Higher Floating-Point Registers (f32 to f127) Written	A write to f32 to f127 occurred.	No writes to f32 to f127 have occurred.

For specific information about each setting, see the Intel IPF documentation.

This routine is only available as an intrinsic.

END Microsoft Specific
See Also
Reference
__sum
Compiler Intrinsics

__setReg

Microsoft Specific

Sets the value of a register specified by the first parameter, which are the same values as declared in the header file intrin.h.

```
void __setReg(
   int Register,
   unsigned __int64 Value
);
```

Parameters

[in] *Register*

A constant specifying the desired processor register.

[in] *Value*

The value to set in the specified register.

Requirements

Intrinsic	Architecture
__setReg	IPF

Header file <intrin.h>

Remarks

Attempting to write directly to processor registers can produce unexpected results. Some registers are not writeable in user mode and might throw a hardware exception or ignore attempts to write to them. For more information, see the processor documentation.

This routine is only available as an intrinsic.

Example

```
// setReg.cpp
// processor: IPF
#include <cstdio>
#include <intrin.h>
#define __CV_IPF_IntR10 1034

#pragma intrinsic(__setReg)
#pragma intrinsic(__getReg)

int main()
{
    __setReg(__CV_IPF_IntR10, 167);
    printf_s("%I64d\n", __getReg(__CV_IPF_IntR10));

    return 0;
}
```

Output

```
167
```

END Microsoft Specific

See Also

Reference

__getReg

Compiler Intrinsics

__segmentlimit

Microsoft Specific

Generates the **lsl** (Load Segment Limit) instruction.

```
unsigned long __segmentlimit(
   unsigned long a
);
```

Parameters

[in] *a*

A constant specifying the segment selector.

Return Value

The segment limit of the segment selector specified by *a*, provided that the selector is valid and visible at the current privilege level.

Requirements

Intrinsic	Architecture
__segmentlimit	x86, x64

Header file <intrin.h>

Remarks

If successful, the zero flag (ZF) is set to 1. On failure, ZF is set to zero.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__shiftleft128

Microsoft Specific

Shifts a 128-bit quantity, represented as two 64-bit quantities *LowPart* and *HighPart*, to the left by a number of bits specified by *Shift* and returns the high 64 bits of the result.

```
unsigned __int64 __shiftleft128(
   unsigned __int64 LowPart,
   unsigned __int64 HighPart,
   unsigned char Shift
);
```

Parameters

[in] *LowPart*

The low 64 bits of the 128-bit quantity to shift.

[in] *HighPart*

The high 64 bits of the 128-bit quantity to shift.

[in] *Shift*

The number of bits to shift.

Return Value

The high 64 bits of the result.

Requirements

Intrinsic	Architecture
__shiftleft128	IPF, x64

Header file <intrin.h>

Remarks

The *Shift* value is always modulo 64 so that, for example, if you call `__shiftleft128(1, 0, 64)`, the function will shift the low part `0` bits left and return a high part of `0` and not `1` as might otherwise be expected.

Example

```
// shiftleft128.c
// processor: IPF, x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic (__shiftleft128, __shiftright128)


int main()
{
    __int64 i = 0x1I64;
    __int64 j = 0x10I64;
    __int64 ResultLowPart;
    __int64 ResultHighPart;

    ResultLowPart = i << 1;
    ResultHighPart = __shiftleft128(i, j, 1);

    // concatenate the low and high parts padded with 0's
    // to display correct hexadecimal 128 bit values
    printf_s("0x%02I64x%016I64x << 1 = 0x%02I64x%016I64x\n",
             j, i, ResultHighPart, ResultLowPart);
```

```
    ResultHighPart = j >> 1;
    ResultLowPart = __shiftright128(i, j, 1);

    printf_s("0x%02I64x%016I64x >> 1 = 0x%02I64x%016I64x\n",
             j, i, ResultHighPart, ResultLowPart);
}
```

Output

```
0x100000000000000001 << 1 = 0x200000000000000002
0x100000000000000001 >> 1 = 0x080000000000000000
```

END Microsoft Specific

See Also

Reference

__shiftright128

Compiler Intrinsics

__shiftright128

Microsoft Specific

Shifts a 128-bit quantity, represented as two 64-bit quantities *LowPart* and *HighPart*, to the right by a number of bits specified by *Shift* and returns the low 64 bits of the result.

```
unsigned __int64 __shiftright128(
   unsigned __int64 LowPart,
   unsigned __int64 HighPart,
   unsigned char Shift
);
```

Parameters

[in] *LowPart*

The low 64 bits of the 128-bit quantity to shift.

[in] *HighPart*

The high 64 bits of the 128-bit quantity to shift.

[in] *Shift*

The number of bits to shift.

Return Value

The low 64 bits of the result.

Requirements

Intrinsic	Architecture
__shiftright128	IPF, x64

Header file <intrin.h>

Remarks

The *Shift* value is always modulo 64 so that, for example, if you call `__shiftright128(0, 1, 64)`, the function will shift the high part `0` bits right and return a low part of `0` and not `1` as might otherwise be expected.

Example

For an example, see __shiftleft128.

END Microsoft Specific

See Also

Reference

__shiftleft128

Compiler Intrinsics

__ssm

Microsoft Specific

Emits the IPF Set System Mask (**ssm**) instruction.

```
void __ssm(
   int Mask
);
```

Parameters
[in] *Mask*

The value to perform an **OR** operation with the system mask. Must be a compile time constant.

Requirements

Intrinsic	Architecture
__ssm	IPF

Header file <intrin.h>

Remarks

The *Mask* argument is used with an **OR** operation with the system mask to determine the new system mask value.

This intrinsic is available in kernel mode only, and the routine is only available as an intrinsic.

END Microsoft Specific
See Also
Reference
__rsm
Compiler Intrinsics

__store128, __store128_rel

Microsoft Specific

Stores a 128-bit value atomically.

```
void __store128(
   __int64 volatile * Destination,
   __int64 SourceHigh
   __int64 SourceLow
);
void __store128_rel(
   __int64 volatile * Destination,
   __int64 SourceHigh
   __int64 SourceLow
);
```

Parameters

[in] *Destination*

Pointer to the 128-bit *Destination* value.

[in] *SourceHigh*

The most significant 64 bits of the 128-bit value to be stored.

[in] *SourceLow*

The least significant 64 bits of the 128-bit value to be stored.

Requirements

Function	Platforms
__store128	Itanium
__store128_rel	Itanium

Header file <intrin.h>

Remarks

The **__store128** function performs an atomic 128-bit store to the *Destination* memory location. The *Destination* pointer should point to a 16-byte aligned memory location.

These functions are not general-purpose 128-bit intrinsics. These functions should only be used if you really need 128-bit atomic stores. You get better performance with regular 64-bit operations.

Note that the intrinsic form of this function has a leading underscore in the name. Also note that these functions are always expanded as intrinsics. That means you can use /Oi (Generate Intrinsic Functions) or **#pragma intrinsic**, but it is not required.

The **__store128_rel** intrinsic function is the same as the corresponding function without the **_rel** suffix except that the operation is performed with release semantics.

In Visual C++ 2005, these functions behave as read-write barriers. For more information, see _ReadWriteBarrier.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__load128, __load128_acq

__stosb

Microsoft Specific

Generates a store string instruction (**rep stosb**).

```
void __stosb(
   unsigned char* Dest,
   unsigned char Data,
   size_t Count
);
```

Parameters

[out] *Dest*

The destination of the operation.

[in] *Data*

The data to store.

[in] *Count*

The length of the block of bytes to write.

Requirements

Intrinsic	Architecture
__stosb	x86, x64

Header file <intrin.h>

Remarks

The result is that the character *Data* is written into a block of *Count* bytes in the *Dest* string.

This routine is only available as an intrinsic.

Example

```
// stosb.c
// processor: x86, x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__stosb)

int main()
{
    unsigned char c = 0x40; /* '@' character */
    unsigned char s[] = "********************************";

    printf_s("%s\n", s);
    __stosb((unsigned char*)s+1, c, 6);
    printf_s("%s\n", s);

}
```

Output

```
********************************
*@@@@@@*************************
```

END Microsoft Specific

See Also

Reference

__stosd

Microsoft Specific

Generates a store string instruction (**rep stosd**).

```
void __stosd(
   unsigned long* Dest,
   unsigned long Data,
   size_t Count
);
```

Parameters

[out] *Dest*

The destination of the operation.

[in] *Data*

The data to store.

[in] *Count*

The length of the block of doublewords to write.

Requirements

Intrinsic	Architecture
__stosd	x86, x64

Header file <intrin.h>

Remarks

The result is that the doubleword *Data* is written into a block of *Count* doublewords at the memory location pointed to by *Dest*.

This routine is only available as an intrinsic.

Example

```
// stosd.c
// processor: x86, x64

#include <stdio.h>
#include <memory.h>
#include <intrin.h>

#pragma intrinsic(__stosd)

int main()
{
    unsigned long val = 99999;
    unsigned long a[10];

    memset(a, 0, sizeof(a));
    __stosd(a+1, val, 2);

printf_s( "%u %u %u %u",
              a[0], a[1], a[2], a[3]);
}
```

Output

```
0 99999 99999 0
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__stosq

Microsoft Specific

Generates a store string instruction (**rep stosq**).

```
void __stosb(
   unsigned __int64* Dest,
   unsigned __int64 Data,
   size_t Count
);
```

Parameters

[out] *Dest*

The destination of the operation.

[in] *Data*

The data to store.

[in] *Count*

The length of the block of quadwords to write.

Requirements

Intrinsic	Architecture
__stosq	AMD64

Header file <intrin.h>

Remarks

The result is that the quadword *Data* is written into a block of *Count* quadwords in the *Dest* string.

This routine is only available as an intrinsic.

Example

```
// stosq.c
// processor: x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__stosq)

int main()
{
   unsigned __int64 val = 0xFFFFFFFFFFFFI64;
   unsigned __int64 a[10];
   memset(a, 0, sizeof(a));
   __stosq(a+1, val, 2);
   printf("%I64x %I64x %I64x %I64x", a[0], a[1], a[2], a[3]);
}
```

Output

```
0 ffffffffffff ffffffffffff 0
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__stosw

Microsoft Specific

Generates a store string instruction (**rep stosw**).

```
void __stosw(
   unsigned short* Dest,
   unsigned short Data,
   size_t Count
);
```

Parameters

[out] *Dest*

The destination of the operation.

[in] *Data*

The data to store.

[in] *Count*

The length of the block of words to write.

Requirements

Intrinsic	Architecture
__stosw	x86, x64

Header file <intrin.h>

Remarks

The result is that the word *Data* is written into a block of *Count* words in the *Dest* string.

This routine is only available as an intrinsic.

Example

```
// stosw.c
// processor: x86, x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__stosw)

int main()
{
    unsigned short val = 128;
    unsigned short a[100];
    memset(a, 0, sizeof(a));
    __stosw(a+10, val, 2);
    printf_s("%u %u %u %u", a[9], a[10], a[11], a[12]);
}
```

Output

```
0 128 128 0
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__sum

Microsoft Specific

Emits the IPF Set User Mask (**sum**) instruction.

```
void __sum(
   int Mask
);
```

Parameters

[in] *Mask*

The 6-bit value to perform an **OR** operation with the user mask. Must be a compile time constant.

Requirements

Intrinsic	Architecture
__sum	IPF

Header file <intrin.h>

Remarks

The *Mask* parameter is used with an **OR** operation with the user mask to determine the new user mask value. For a table of the meaning of each of the 6 bits, see __rum.

This routine is only available as an intrinsic.

Example

```
// usermask.cpp
// processor: IPF

#include <cstdio>
#include <intrin.h>

#pragma intrinsic(__sum, __rum)

#define BIG_ENDIAN 2

int main()
{
    int p[1];

    p[0] = 0x12345678;
    printf_s("0x%x\n", p[0]);

    // Switch to big-endian memory references.
    __sum(BIG_ENDIAN);

    // write the value in big-endian format
    p[0] = 0x12345678;

    // Switch back to little-endian memory references.
    __rum(BIG_ENDIAN);

    // The same statement in code has produced a different result!
    printf_s("0x%x\n", p[0]);
}
```

Output

```
0x12345678
0x78563412
```

END Microsoft Specific

See Also

Reference

__rum

Compiler Intrinsics

__synci

Microsoft Specific

Emits the IPF Memory Synchronization (**sync.i**) instruction.

```
void __synci(void);
```

Requirements

Intrinsic	Architecture
__synci	IPF

Header file <intrin.h>

Remarks

For more information, see the Intel IPF documentation.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__thash

Microsoft Specific

Generates the IPF **thash** instruction, which takes a virtual address and returns the corresponding virtual hashed page table (VHPT) entry address.

```
__int64 __thash(
   __int64 Address
);
```

Parameters
[in] *Address*

The virtual address to hash.

Return Value

The hashed form of the virtual address passed in.

Requirements

Intrinsic	Architecture
__thash	IPF

Header file <intrin.h>

Remarks

For more information, see the Intel IPF documentation.

This routine is only available as an intrinsic.

END Microsoft Specific
See Also
Reference
Compiler Intrinsics

__ttag

Microsoft Specific

Emits the IPF **ttag** instruction.

```
__int64 __ttag(
   __int64 Address
);
```

Parameters

[in] *Address*

The virtual address.

Return Value

The tag for the given address.

Requirements

Intrinsic	Architecture
__ttag	IPF

Header file <intrin.h>

Remarks

For more information, see the Intel IPF documentation.

This routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__ull_rshift

Microsoft Specific

on x64, shifts a 64-bit value specified by the first parameter to the right by a number of bits specified by the second parameter.

```
unsigned __int64 __ull_rshift(
   unsigned __int64 mask,
   int nBit
);
```

Parameters

[in] *mask*

The 64-bit integer value to shift right.

[in] *nBit*

The number of bits to shift, modulo 32 on x86, and modulo 64 on x64.

Return Value

The mask shifted by *nBit* bits.

Requirements

Intrinsic	**Architecture**
__ull_rshift	x86, x64

Header file <intrin.h>

Remarks

If the second parameter is greater than 31 on x86 (63 on x64), that number is taken modulo 32 (64 on x64) to determine the number of bits to shift. The **ull** in the name indicates **unsigned long long (unsigned __int64)**.

Example

```
// ull_rshift.cpp
// compile with: /EHsc
// processor: x86, x64
#include <iostream>
#include <intrin.h>
using namespace std;

#pragma intrinsic(__ull_rshift)

int main()
{
   unsigned __int64 mask = 0x100;
   int nBit = 8;
   mask = __ull_rshift(mask, nBit);
   cout << hex << mask << endl;
}
```

Output

```
1
```

END Microsoft Specific

See Also

Reference

__ll_lshift

__ll_rshift

Compiler Intrinsics

_umul128

Microsoft Specific

Multiplies two 64-bit unsigned integers passed in as the first two arguments and puts the high 64 bits of the product in the 64-bit unsigned integer pointed to by *HighProduct* and returns the low 64 bits of the product.

```
unsigned __int64 _umul128(
   unsigned __int64 Multiplier,
   unsigned __int64 Multiplicand,
   unsigned __int64 *HighProduct
);
```

Parameters

[in] *Multiplier*

The first 64-bit integer to multiply.

[in] *Multiplicand*

The second 64-bit integer to multiply.

[out] *HighProduct*

The high 64 bits of the product.

Return Value

The low 64 bits of the product.

Requirements

Intrinsic	**Architecture**
_umul128	IPF, x64

Header file <intrin.h>

Example

```
// umul128.c
// processor: IPF, x64

#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_umul128)

int main()
{
    unsigned __int64 a = 0x0fffffffffffffffI64;
    unsigned __int64 b = 0xf0000000I64;
    unsigned __int64 c, d;

    d = _umul128(a, b, &c);

    printf_s("%#I64x * %#I64x = %#I64x%I64x\n", a, b, c, d);
}
```

Output

```
0xfffffffffffffff * 0xf0000000 = 0xeffffffffffffff10000000
```

END Microsoft Specific

See Also

Reference

__UMULH, __umulh

Microsoft Specific

Return the high 64 bits of the product of two 64-bit unsigned integers.

```
unsigned __int64 __UMULH(
   unsigned __int64 a,
   unsigned __int64 b
);
unsigned __int64 __umulh(
   unsigned __int64 a,
   unsigned __int64 b
);
```

Parameters

[in] *a*

The first number to multiply.

[in] *b*

The second number to multiply.

Return Value

The high 64 bits of the 128-bit result of the multiplication.

Requirements

Intrinsic	Architecture
__UMULH	IPF
__umulh	x64

Header file <intrin.h>

Remarks

These routines are only available as intrinsics.

Example

```
// umulh.cpp
// processor: IPF
#include <cstdio>
#include <intrin.h>

#pragma intrinsic (__UMULH)

int main()
{
    unsigned __int64 i = 0x10;
    unsigned __int64 j = 0xFEDCBA9876543210;
    unsigned __int64 k = i * j; // k has the low 64 bits
                                // of the product.
    unsigned __int64 result;
    result = __UMULH(i, j); // result has the high 64 bits
                            // of the product.
    printf_s("0x%I64x * 0x%I64x = 0x%I64x%I64x \n", i, j, result, k);
    return 0;
}
```

Output

```
0x10 * 0xfedcba9876543210 = 0xfedcba98765432100
```

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__wbinvd

Microsoft Specific

Generates the Write Back and Invalidate Cache (**wbinvd**) instruction.

```
void __wbinvd(void);
```

Requirements

Intrinsic	Architecture
__wbinvd	x86, x64

Header file <intrin.h>

Remarks

This function is only available in kernel mode with a privilege level (CPL) of 0, and the routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

_WriteBarrier

Microsoft Specific

Forces writes to memory to complete at the point of the call.

```
void _WriteBarrier(void);
```

Requirements

Intrinsic	Architecture
_WriteBarrier	x86, IPF, x64

Header file <intrin.h>

Remarks

The _ReadBarrier, _WriteBarrier, and _ReadWriteBarrier functions help ensure the proper operation of multithreaded programs that are optimized by the Visual C++ compiler. A correctly optimized program yields the same results when it executes on multiple threads as when it executes on a single thread.

To help ensure that the optimized program operates correctly, the _WriteBarrier function forces writes to memory to complete at the point of the call. After the call, other threads can access the memory without fear that the thread that made the call might have a pending write to the memory. For more information, see the remarks in _ReadWriteBarrier.

Note In past versions of the Visual C++ compiler, the _ReadWriteBarrier and _WriteBarrier functions were enforced only locally and did not affect functions up the call tree. In Visual C++ 2005 and later, these functions are enforced all the way up the call tree.

Example

The following example is similar to that shown in _ReadWriteBarrier. The function `f` makes two assignments to the local variable `x`. Without the **_WriteBarrier** the first assignment is optimized away as a useless assignment. With the **_WriteBarrier** in place both assignments are retained and an access violation occurs when the first assignment attempts to dereference a NULL pointer.

```
// intrinsics_writebarrier.c
// compile with: /O2 -DNO_BARRIER
// This code contains an error--dereferencing a null pointer--
// which will be optimized away as a useless assignment.
// Omit the NO_BARRIER command line to activate the Write Barrier.
// With the barrier activated, the assignment is not optimized away
// and causes an access violation.

#include <windows.h> // for EXCEPTION_ACCESS_VIOLATION
#include <excpt.h>
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(_WriteBarrier)

int x = 0;

__declspec(noinline) int f(int* p)
{
    x = *p;
#ifndef NO_BARRIER
    _WriteBarrier();
#endif
    x = 7;
    return x;
}
```

```
// If code is EXCEPTION_ACCESS_VIOLATION it should be the
// attempt to read from the NULL pointer we passed in, so
// we handle the exception.
int filter(unsigned int code, struct _EXCEPTION_POINTERS *ep)
{
    if (code == EXCEPTION_ACCESS_VIOLATION)
    {
        printf_s("AV\n");
        return EXCEPTION_EXECUTE_HANDLER;
    }

    // If not what we were looking for, we don't want to handle it.
    return EXCEPTION_CONTINUE_SEARCH;
}

int main()
{
    __try
    {
        // Should return only if the first assignment is
        // optimized away.
        f(NULL);
        printf_s("Assignment was optimized away!\n");
    }
    __except(filter(GetExceptionCode(), GetExceptionInformation()))
    {
        // We get here if an Access violation occurred.
        printf_s("Access Violation: assignment was not optimized away.\n");
    }
}
```

Sample Output

Without /DNO_BARRIER:

```
AV
Access Violation: assignment was not optimized away.
```

With /DNO_BARRIER:

```
Assignment was optimized away!
```

END Microsoft Specific

See Also

Reference

_ReadBarrier

_ReadWriteBarrier

Compiler Intrinsics

__writecr0

Microsoft Specific

Writes the value *Data* to the CR0 register.

```
void writecr0(
   unsigned __int64 Data
);
```

Parameters

[in] *Data*

The value to write to the CR0 register.

Requirements

Intrinsic	Architecture
__writecr0	x86, x64

Header file <intrin.h>

Remarks

This intrinsic is only available in kernel mode, and the routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__writecr3

Microsoft Specific

Writes the value *Data* to the CR3 register.

```
void writecr3(
   unsigned __int64 Data
);
```

Parameters

[in] *Data*

The value to write to the CR3 register.

Requirements

Intrinsic	Architecture
__writecr3	x86, x64

Header file <intrin.h>

Remarks

This intrinsic is only available in kernel mode, and the routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__writecr4

Microsoft Specific

Writes the value *Data* to the CR4 register.

```
void writecr4(
   unsigned __int64 Data
);
```

Parameters

[in] *Data*

The value to write to the CR4 register.

Requirements

Intrinsic	Architecture
__writecr4	x86, x64

Header file <intrin.h>

Remarks

This intrinsic is only available in kernel mode, and the routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__writecr8

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Writes the value *Data* to the CR8 register.

```
void writecr8(
   unsigned __int64 Data
);
```

Parameters

[in] *Data*

The value to write to the CR8 register.

Requirements

Intrinsic	Architecture
__writecr8	x86, x64

Header file <intrin.h>

Remarks

This intrinsic is available only in kernel mode, and the routine is available only as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__writefsbyte, __writefsdword, __writefsqword, __writefsword

Microsoft Specific

Write memory to a location specified by an offset relative to the beginning of the FS segment.

```
void __writefsbyte(
   unsigned long Offset,
   unsigned char Data
);
void __writefsword(
   unsigned long Offset,
   unsigned short Data
);
void __writefsdword(
   unsigned long Offset,
   unsigned long Data
);
void __writefsqword(
   unsigned long Offset,
   unsigned __int64 Data
);
```

Parameters

[in] *Offset*

The offset from the beginning of FS to write to.

[in] *Data*

The value to write.

Requirements

Intrinsic	Architecture
__writefsbyte	x86
__writefsword	x86
__writefsdword	x86
__writefsqword	x86

Header file <intrin.h>

Remarks

These intrinsics are available in kernel mode only, and these routines are only available as intrinsics.

END Microsoft Specific

See Also

Reference

__readfsbyte, __readfsdword, __readfsqword, __readfsword

Compiler Intrinsics

__writegsbyte, __writegsdword, __writegsqword, __writegsword

Microsoft Specific

Write memory to a location specified by an offset relative to the beginning of the GS segment.

```
void __writegsbyte(
   unsigned long Offset,
   unsigned char Data
);
void __writegsword(
   unsigned long Offset,
   unsigned short Data
);
void __writegsdword(
   unsigned long Offset,
   unsigned long Data
);
void __writegsqword(
   unsigned long Offset,
   unsigned __int64 Data
);
```

Parameters

[in] *Offset*

The offset from the beginning of GS to write to.

[in] *Data*

The value to write.

Requirements

Intrinsic	Architecture
__writegsbyte	x64
__writegsdword	x64
__writegsqword	x64
__writegsword	x64

Header file <intrin.h>

Remarks

These intrinsics are available in kernel mode only, and these routines are only available as intrinsics.

END Microsoft Specific

See Also

Reference

__readgsbyte, __readgsdword, __readgsqword, __readgsword

Compiler Intrinsics

__writemsr

Microsoft Specific

Generates the Write to Model Specific Register (**wrmsr**) instruction.

```
void __writemsr(
   unsigned long Register,
   unsigned __int64 Value
);
```

Parameters

[in] *Register*

The model specific register.

[in] *Value*

The value to write.

Requirements

Intrinsic	Architecture
__writemsr	x86, x64

Header file <intrin.h>

Remarks

This function may only be used in kernel mode, and this routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__yield

Microsoft Specific

Generates the Performance Hint 0 (**hint 0**) instruction.

```
void __yield(void);
```

Requirements

Intrinsic	Architecture
__yield	IPF

Header file <intrin.h>

Remarks

The hint tells the processor that this thread is not doing anything useful, so it can freely yield processing time to another thread. For example, it can be used in a loop attempting to acquire a spin lock to reduce processor cycles wasted while waiting to acquire the lock.

This intrinsic is similar to the __mm_pause intrinsic. The routine is only available as an intrinsic.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__writedr

This topic is new for Visual Studio 2005 SP1.

Writes the specified value to the specified debug register.

```
void __writedr(unsigned DebugRegister, unsigned DebugValue);
void __writedr(unsigned DebugRegister, unsigned __int64 DebugValue);
```

Parameters

[in] *DebugRegister*

A number from 0 through 7 that identifies the debug register.

[in] *DebugValue*

A value to write to the debug register.

Remarks

These intrinsics are available only in kernel mode, and the routines are available only as intrinsics.

Requirements

Intrinsic	Architecture
__writedr	x86, x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__readdr

__readdr

This topic is new for Visual Studio 2005 SP1.

Reads the value of the specified debug register.

```
unsigned         __readdr(unsigned int DebugRegister);
unsigned __int64 __readdr(unsigned int DebugRegister);
```

Parameters

[in] *DebugRegister*

A number from 0 through 7 that identifies the debug register.

Return Value

The value of the specified debug register.

Remarks

These intrinsics are available only in kernel mode, and the routines are available only as intrinsics.

Requirements

Intrinsic	**Architecture**
__readdr	x86, x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__readeflags

__writeeflags

This topic is new for Visual Studio 2005 SP1.

Writes the specified value to the program status and control (EFLAGS) register.

```
void __writeeflags(unsigned Value);
void __writeeflags(unsigned __int64 Value);
```

Parameters

Parameter	Description
[in] *Value*	The value to write to the EFLAGS register. The *Value* parameter is 32 bits long for a 32-bit platform and 64 bits long for a 64-bit platform.

Remarks

These routines are available only as intrinsics.

Requirements

Intrinsic	Architecture
__writeeflags	x86, x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__readeflags

__readeflags

This topic is new for Visual Studio 2005 SP1.

Reads the program status and control (EFLAGS) register.

```
unsigned     int __readeflags(void);
unsigned __int64 __readeflags(void);
```

Return Value

The value of the EFLAGS register. The return value is 32 bits long on a 32-bit platform and 64 bits long on a 64-bit platform.

Remarks

These routines are available only as intrinsics.

Requirements

Intrinsic	Architecture
__readeflags	x86, x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__writeeflags

__svm_vmrun

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Starts execution of the virtual machine guest code that corresponds to the specified virtual machine control block (VMCB).

```
void __svm_vmrun(
   size_t VmcbPhysicalAddress
);
```

Parameters

Parameter	Description
[in] *VmcbPhysicalAddress*	The physical address of the VMCB.

Remarks

The **__svm_vmrun** function uses a minimal amount of information in the VMCB to begin executing the virtual machine guest code. Use the __svm_vmsave or __svm_vmload function if you require more information to handle a complex interrupt or to switch to another guest.

The **__svm_vmrun** function is equivalent to the **VMRUN** machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "AMD64 Architecture Programmer's Manual Volume 2: System Programming," document number 24593, revision 3.11 or later, at the AMD corporation site.

Requirements

Intrinsic	Architecture
__svm_vmrun	x86, x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__svm_vmsave

__svm_vmload

__svm_vmsave

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Stores a subset of processor state in the specified virtual machine control block (VMCB).

```
void __svm_smsave(
   size_t VmcbPhysicalAddress
);
```

Parameters

Parameter	Description
[in] *VmcbPhysicalAddress*	The physical address of the VMCB.

Remarks

The **__svm_vmsave** function is equivalent to the **VMSAVE** machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "AMD64 Architecture Programmer's Manual Volume 2: System Programming," document number 24593, revision 3.11 or later, at the AMD Corporation site.

Requirements

Intrinsic	Architecture
__svm_vmsave	x86, x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__svm_vmrun

__svm_vmload

__svm_vmload

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Loads a subset of processor state from the specified virtual machine control block (VMCB).

```
void __svm_vmload(
   size_t VmcbPhysicalAddress
);
```

Parameters

Parameter	Description
[in] *VmcbPhysicalAddress*	The physical address of the VMCB.

Remarks

The **__svm_vmload** function is equivalent to the **VMLOAD** machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "AMD64 Architecture Programmer's Manual Volume 2: System Programming," document number 24593, revision 3.11, at the AMD corporation site.

Requirements

Intrinsic	Architecture
__svm_vmload	x86, x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__svm_vmrun

__svm_vmsave

__svm_stgi

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Sets the global interrupt flag.

```
void __svm_stgi(void);
```

Remarks

The **__svm_stgi** function is equivalent to the **STGI** machine instruction. The global interrupt flag determines whether the microprocessor ignores, postpones, or handles interrupts due to events such as an I/O completion, a hardware temperature alert, or a debug exception.

This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "AMD64 Architecture Programmer's Manual Volume 2: System Programming," document number 24593, revision 3.11, at the AMD corporation site.

Requirements

Intrinsic	Architecture
__svm_stgi	x86, x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__svm_clgi

__svm_skinit

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Initiates the loading of verifiably secure software, such as a virtual machine monitor.

```
void __svm_skinit(
   int SLB
);
```

Parameters

Parameter	Description
SLB	The 32-bit physical address of a 64K byte Secure Loader Block (SLB).

Remarks

The **__svm_skinit** function is equivalent to the **SKINIT** machine instruction. This function is part of a security system that uses the processor and a Trusted Platform Module (TPM) to verify and load trusted software called a security kernel (SK). A virtual machine monitor is an example of a security kernel. The security system verifies program components loaded during the initialization process, and protects components from tampering by interrupts, device access, or another program if the computer is a multiprocessor.

The *SLB* parameter specifies the physical address of a 64K block of memory called the *Secure Loader Block* (SLB). The SLB contains a program called the secure loader that establishes the operating environment for the computer, and subsequently loads the security kernel.

This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "AMD64 Architecture Programmer's Manual Volume 2: System Programming," document number 24593, revision 3.11, at the AMD corporation site.

Requirements

Intrinsic	Architecture
__svm_skinit	x86, x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__svm_invlpga

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Invalidates the address mapping entry in the computer's translation look-aside buffer. Parameters specify the virtual address and address space identifier of the page to invalidate.

```
void __svm_invlpga(
        void *Va,
        int ASID);
```

Parameters

Parameter	Description
[in] *Va*	The virtual address of the page to invalidate.
[in] *ASID*	The address space identifier (ASID) of the page to invalidate.

Remarks

The **__svm_invlpga** function is equivalent to the **INVLPGA** machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "AMD64 Architecture Programmer's Manual Volume 2: System Programming," document number 24593, revision 3.11, at the AMD corporation site.

Requirements

Intrinsic	Architecture
__svm_invlpga	x86, x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__svm_clgi

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Clears the global interrupt flag.

```
void __svm_clgi( void );
```

Remarks

The **__svm_clgi** function is equivalent to the **CLGI** machine instruction. The global interrupt flag determines whether the microprocessor ignores, postpones, or handles interrupts due to events such as an I/O completion, a hardware temperature alert, or a debug exception.

This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "AMD64 Architecture Programmer's Manual Volume 2: System Programming," document number 24593, revision 3.11, at the AMD corporation site.

Requirements

Intrinsic	Architecture
__svm_clgi	x86, x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__svm_stgi

__nop

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Generates platform-specific machine code that performs no operation.

```
void __nop();
```

Requirements

Intrinsic	Architecture
__nop	x86, x64

Header file <intrin.h>

END Microsoft Specific

Remarks

The **__nop** function is equivalent to the **NOP** machine instruction. For more information, search for the document, "Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference," at the Intel Corporation site.

See Also

Reference

Compiler Intrinsics

__noop

__ud2

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Generates an undefined instruction.

```
void __ud2();
```

Remarks

The processor raises an invalid opcode exception if you execute an undefined instruction.

The **__ud2** function is equivalent to the **UD2** machine instruction, and is available only in kernel mode. For more information, search for the document, "Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference," at the Intel Corporation site.

Requirements

Intrinsic	Architecture
__ud2	x86, x64

Header file <intrin.h>

END Microsoft Specific

Example

The following example executes an undefined instruction, which raises an exception. The exception handler then changes the return code from zero to one.

```
// __ud2_intrinsic.cpp
#include <stdio.h>
#include <intrin.h>
#include <excpt.h>
// compile with /EHa

int main() {

// Initialize the return code to 0.
 int ret = 0;

// Attempt to execute an undefined instruction.
  printf("Before __ud2(). Return code = %d.\n", ret);
  __try {
  __ud2();
  }

// Catch any exceptions and set the return code to 1.
  __except(EXCEPTION_EXECUTE_HANDLER){
  printf("  In the exception handler.\n");
  ret = 1;
  }

// Report the value of the return code.
  printf("After __ud2().  Return code = %d.\n", ret);
  return ret;
}
```

Output

```
Before __ud2(). Return code = 0.
  In the exception handler.
After __ud2().  Return code = 1.
```

See Also

Reference

Compiler Intrinsics

__lidt

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Loads the interrupt descriptor table register (IDTR) with the value in the specified memory location.

```
void __lidt(
        void *Source);
```

Parameters

Parameter	Description
[in] *Source*	Pointer to the value to be copied to the IDTR.

Requirements

Intrinsic	Architecture
__lidt	x86, x64

Header file <intrin.h>

Remarks

The **__lidt** function is equivalent to the **LIDT** machine instruction, and is available only in kernel mode. For more information, search for the document, "Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference," at the Intel Corporation site.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__sidt

__sidt

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Stores the value of the interrupt descriptor table register (IDTR) in the specified memory location.

```
void __sidt(
        void *Destination);
```

Parameters

Parameter	Description
[in] *Destination*	A pointer to the memory location where the IDTR is stored.

Requirements

Intrinsic	Architecture
__sidt	x86, x64

Header file <intrin.h>

Remarks

The **__sidt** function is equivalent to the **SIDT** machine instruction. For more information, search for the document, "Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference," at the Intel Corporation site.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__lidt

__halt

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Halts the microprocessor until an enabled interrupt, a nonmaskable interrupt (NMI), or a reset occurs.

```
void __halt( void );
```

Requirements

Intrinsic	Architecture
__halt	x86, x64

Header file <intrin.h>

Remarks

The **__halt** function is equivalent to the **HLT** machine instruction, and is available only in kernel mode. For more information, search for the document, "Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference," at the Intel Corporation site.

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__vmx_vmptrst

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Stores the pointer to the current virtual-machine control structure (VMCS) at the specified address.

```
void __vmx_vmptrst(
   unsigned __int64 *VmcsPhysicalAddress
);
```

Parameters
[in] **VmcsPhysicalAddress*

The address where the current VMCS pointer is stored.

Remarks

The VMCS pointer is a 64-bit physical address.

The **__vmx_vmptrst** function is equivalent to the **VMPTRST** machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "Intel Virtualization Technical Specification for the IA-32 Intel Architecture," document number C97063-002, at the Intel Corporation site.

Requirements

Intrinsic	Architecture
__vmx_vmptrst	x86, x64

Header file <intrin.h>

END Microsoft Specific
See Also
Reference
Compiler Intrinsics
__vmx_vmptrld

__vmx_vmptrld

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Loads the pointer to the current virtual-machine control structure (VMCS) from the specified address.

```
int __vmx_vmptrld(
   unsigned __int64 *VmcsPhysicalAddress
);
```

Parameters

[in] **VmcsPhysicalAddress*

The address where the VMCS pointer is stored.

Return Value

0

The operation succeeded.

1

The operation failed with extended status available in the **VM-instruction error field** of the current VMCS.

2

The operation failed without status available.

Remarks

The VMCS pointer is a 64-bit physical address.

The **__vmx_vmptrld** function is equivalent to the **VMPTRLD** machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "Intel Virtualization Technical Specification for the IA-32 Intel Architecture," document number C97063-002, at the Intel Corporation site.

Requirements

Intrinsic	Architecture
__vmx_vmptrld	x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__vmx_vmptrst

__vmx_vmlaunch

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Places the calling application in VMX non-root operation state (VM enter) by using the current virtual-machine control structure (VMCS).

```
unsigned char __vmx_vmlaunch(
   void);
```

Return Value

Value	Meaning
0	The operation succeeded.
1	The operation failed with extended status available in the **VM-instruction error field** of the current VMCS.
2	The operation failed without status available.

Remarks

An application can perform a VM-enter operation by using either the __vmx_vmlaunch or __vmx_vmresume function. The __vmx_vmlaunch function can be used only with a VMCS whose launch state is **Clear**, and the __vmx_vmresume function can be used only with a VMCS whose launch state is **Launched**. Consequently, use the __vmx_vmclear function to set the launch state of a VMCS to **Clear**, and then use the __vmx_vmlaunch function for your first VM-enter operation and the __vmx_vmresume function for subsequent VM-enter operations.

The **__vmx_vmlaunch** function is equivalent to the **VMLAUNCH** machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "Intel Virtualization Technical Specification for the IA-32 Intel Architecture," document number C97063-002, at the Intel Corporation site.

Requirements

Intrinsic	Architecture
__vmx_vmlaunch	x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__vmx_vmresume

__vmx_vmclear

__vmx_vmwrite

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Writes the specified value to the specified field in the current virtual machine control structure (VMCS).

```
unsigned char __vmx_vmwrite(
   size_t Field,
   size_t FieldValue
);
```

Parameters

Parameter	Description
[in] *Field*	The VMCS field to write.
[in] *FieldValue*	The value to write to the VMCS field.

Return Value

0

The operation succeeded.

1

The operation failed with extended status available in the **VM-instruction error field** of the current VMCS.

2

The operation failed without status available.

Remarks

The **__vmx_vmwrite** function is equivalent to the **VMWRITE** machine instruction. The value of the *Field* parameter is an encoded field index that is described in Intel documentation. For more information, search for the document, "Intel Virtualization Technical Specification for the IA-32 Intel Architecture," document number C97063-002, at the Intel Corporation site, and then consult Appendix C of that document.

Requirements

Intrinsic	Architecture
__vmx_vmwrite	x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__vmx_vmread

__vmx_off

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Deactivates virtual machine extensions (VMX) operation in the processor.

```
void __vmx_off();
```

Remarks

The **__vmx_off** function is equivalent to the **VMXOFF** machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "Intel Virtualization Technical Specification for the IA-32 Intel Architecture," document number C97063-002, at the Intel Corporation site.

Requirements

Intrinsic	Architecture
__vmx_off	x86, x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__vmx_on

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Activates virtual machine extensions (VMX) operation in the processor.

```
unsigned char __vmx_on(
      unsigned __int64 *VmsSupportPhysicalAddress
);
```

Parameters

[in] *VmsSupportPhysicalAddress*

A pointer to a 64-bit physical address that points to a virtual-machine control structure (VMCS).

Return Value

Value	Meaning
0	The operation succeeded.
1	The operation failed with extended status available in the **VM-instruction error field** of the current VMCS.
2	The operation failed without status available.

Remarks

The **__vmx_on** function corresponds to the **VMXON** machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "Intel Virtualization Technical Specification for the IA-32 Intel Architecture," document number C97063-002, at the Intel Corporation site.

Requirements

Intrinsic	Architecture
__vmx_on	x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__vmx_vmread

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Reads a specified field from the current virtual machine control structure (VMCS) and places it in the specified location.

```
unsigned char __vmx_vmread(
   size_t Field,
   size_t *FieldValue
);
```

Parameters

Parameter	Description
[in] *Field*	The VMCS field to read.
[in] *FieldValue*	A pointer to the location to store the value read from the VMCS field specified by the *Field* parameter.

Return Value

Value	Meaning
0	The operation succeeded.
1	The operation failed with extended status available in the **VM-instruction error field** of the current VMCS.
2	The operation failed without status available.

Remarks

The **__vmx_vmread** function is equivalent to the **VMREAD** machine instruction. The value of the *Field* parameter is an encoded field index that is described in Intel documentation. For more information, search for the document, "Intel Virtualization Technical Specification for the IA-32 Intel Architecture," document number C97063-002, at the Intel Corporation site, then consult Appendix C of that document.

Requirements

Intrinsic	Architecture
__vmx_vmread	x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__vmx_vmwrite

__vmx_vmresume

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Resumes VMX non-root operation by using the current virtual machine control structure (VMCS).

```
unsigned char __vmx_vmresume(
   void);
```

Return Value

Value	Meaning
0	The operation succeeded.
1	The operation failed with extended status available in the **VM-instruction error field** of the current VMCS.
2	The operation failed without status available.

Remarks

An application can perform a VM-enter operation by using either the __vmx_vmlaunch or __vmx_vmresume function. The __vmx_vmlaunch function can be used only with a VMCS whose launch state is **Clear**, and the __vmx_vmresume function can be used only with a VMCS whose launch state is **Launched**. Consequently, use the __vmx_vmclear function to set the launch state of a VMCS to **Clear**, and then use the __vmx_vmlaunch function for your first VM-enter operation and the __vmx_vmresume function for subsequent VM-enter operations.

The **__vmx_vmresume** function is equivalent to the **VMRESUME** machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the PDF document, "Intel Virtualization Technical Specification for the IA-32 Intel Architecture," document number C97063-002, at the Intel Corporation site.

Requirements

Intrinsic	Architecture
__vmx_vmresume	x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__vmx_vmlaunch

__vmx_vmclear

__vmx_vmclear

Microsoft Specific

This topic is new for Visual Studio 2005 SP1.

Initializes the specified virtual machine control structure (VMCS) and sets its launch state to **Clear**.

```
unsigned char __vmx_vmclear(
      unsigned __int64 *VmcsPhysicalAddress
);
```

Parameters

Parameter	Description
[in] *VmcsPhysicalAddress*	A pointer to a 64-bit memory location that contains the physical address of the VMCS to clear.

Return Value

Value	Meaning
0	The operation succeeded.
1	The operation failed with extended status available in the **VM-instruction error field** of the current VMCS.
2	The operation failed without status available.

Remarks

An application can perform a VM-enter operation by using either the __vmx_vmlaunch or __vmx_vmresume function. The __vmx_vmlaunch function can be used only with a VMCS whose launch state is **Clear**, and the __vmx_vmresume function can be used only with a VMCS whose launch state is **Launched**. Consequently, use the __vmx_vmclear function to set the launch state of a VMCS to **Clear**. Use the __vmx_vmlaunch function for your first VM-enter operation and the __vmx_vmresume function for subsequent VM-enter operations.

The **__vmx_vmclear** function is equivalent to the **VMCLEAR** machine instruction. This function supports the interaction of a host's virtual machine monitor with a guest operating system and its applications. For more information, search for the document, "Intel Virtualization Technical Specification for the IA-32 Intel Architecture," document number C97063-002, at the Intel Corporation site.

Requirements

Intrinsic	Architecture
__vmx_vmclear	x64

Header file <intrin.h>

END Microsoft Specific

See Also

Reference

Compiler Intrinsics

__vmx_vmlaunch

__vmx_vmresume

MMX, SSE, and SSE2 Intrinsics

This section discusses intrinsic support for the enhanced instruction sets supported by Intel and Advanced Micro Devices (AMD) processors.

Microsoft Specific

- Compiler Support for the MMX, SSE, and SSE2 Intrinsics
- Intel Technology Overview of New Instructions and Extensions
- AMD 3DNow! Technology Overview and Intrinsics
- MMX Technology
- Streaming SIMD Extensions (SSE)
- Streaming SIMD Extensions 2 (SSE2) Instructions

An *intrinsic* is a function known by the compiler that directly maps to a sequence of one or more assembly language instructions. Intrinsic functions are inherently more efficient than called functions because no calling linkage is required.

Intrinsics make the use of processor-specific enhancements easier because they provide a C/C++ language interface to assembly instructions. In doing so, the compiler manages things that the user would normally have to be concerned with, such as register names, register allocations, and memory locations of data.

For information on how to detect the capabilities of a CPU, see CPUID Sample: Determines CPU Capabilities.

All the MMX, SSE and SSE2 intrinsics are only available as intrinsics, thus, they are not affected by the setting of **/Oi**, and `#pragma function` may not be used on them.

END Microsoft Specific
See Also
Reference
Compiler Intrinsics

Compiler Support for the MMX, SSE, and SSE2 Intrinsics

Microsoft Specific

To support the use of MMX, SSE, and SSE2 intrinsics, the compiler includes the following features:

- Data alignment
- Inline assembly

Data Alignment

Previously, alignment issues in programs were addressed either by the compiler or directly in hardware. Also, any alignment changes needed for a program to run correctly were automatically enabled. However, with the advent of intrinsic support, the user must take a more active role to guarantee that alignment issues are appropriately addressed.

Many of the new intrinsics have data alignment requirements. If these intrinsics are used and data is not appropriately aligned, the program will throw an exception that must be handled by the program; otherwise, the program will fault.

The new intrinsics require aligned data to allow better performance. With the size of new registers implemented to support the new, enhanced instruction sets, new alignment requirements were defined to make the best use of recent cache architectures. Specific alignment requirements for each intrinsic can be found in the documentation for the intrinsic.

There are different tools to specify appropriate rules for the alignment of data. For alignment of user declared variables, for example, static or automatic data, refer to the align section documentation. For data dynamically allocated from the heap, refer to the data alignment functions.

Note
The **__m64**, **__m128**, **__m128i** and **__m128d** new data types already have an alignment value.

- align
- __alignof

Inline Assembly

The compiler supports use of intrinsic assembly instructions in inline assembly (`__asm`) blocks. The compiler also accepts the new syntax `MMWORD PTR` and `XMMWORD PTR` to refer to 64- and 128-bit data.

END Microsoft Specific

For information on how to detect the capabilities of a CPU, see CPUID sample.

See Also

Reference

MMX, SSE, and SSE2 Intrinsics

Intel Overview of New Instructions and Extensions

Microsoft Specific

The Pentium III Processor and other processors such as the Pentium processor with MMX technology and Pentium II processor have instructions to enable development of optimized multimedia applications. The instructions are implemented through extensions to previously implemented instructions. This technology uses the single-instruction, multiple-data (SIMD) technique. By processing data elements in parallel, applications with media-rich bitstreams can significantly improve performance by using SIMD instructions.

You can access the Intel performance libraries at http://developer.intel.com/.

The most direct way to use these instructions is to inline the assembly language instructions into your source code. However, this can be time consuming and tedious. Instead, Intel provides easy implementation by using API extension sets, referred to as *intrinsics*.

Intrinsics Availability on Intel Processors

Processors	MMX technology intrinsics	Streaming SIMD Extensions (SSE)	Streaming SIMD Extensions 2 (SSE2) instructions
Processors that support SSE2	Yes	Yes	Yes
Pentium III	Yes	Yes	Not available
Pentium II	Yes	Not available	Not available
Pentium with MMX technology	Yes	Not available	Not available
Pentium Pro	Not available	Not available	Not available
Pentium	Not available	Not available	Not available

The following topics are covered:

- Benefits of Using Intrinsics
- Intrinsic Conventions
- Intrinsic Categories and Supporting Extensions

END Microsoft Specific
See Also
Reference
MMX, SSE, and SSE2 Intrinsics

Benefits of Using Intrinsics

Microsoft Specific

The major benefit of using intrinsics is that you now have access to key features that are not available using conventional coding practices:

- New Registers Enable packed data of up to 128 bits in length for optimal SIMD processing.
- New Data Types Enable packing of up to 16 elements of data in one register.

New Registers

New registers sets, a key feature, are provided by the architecture of the processors.

MMX Technology Register

The MMX technology intrinsics provide eight new registers (MM0 to MM7) that are 64 bits long (0 to 63).

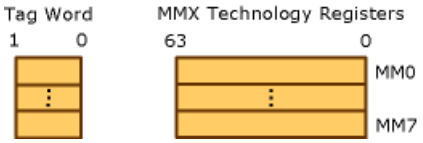

Streaming SIMD Extensions (SSE) and Streaming SIMD Extensions 2 (SSE2) Instructions Registers

The SSE and the SSE2 instructions make use of yet another eight registers (XMM0 to XMM7) that are 128 bits in length.

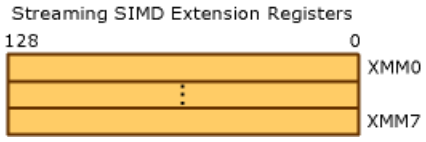

These new data registers enable the processing of data elements in parallel. Because each register can hold more than one data element, the processor can process more than one data element simultaneously. This processing capability is also known as SIMD processing. To enable SIMD processing with the C/C++ compiler, new data types are defined to exploit the expanded size of the new registers.

Using intrinsics allows you to code with the syntax of C function calls and variables instead of with the assembly language. For each computational and data manipulation instruction in the new extension sets, there is a corresponding C intrinsic that directly implements that instruction. This frees you from managing registers and assembly programming. Further, the compiler optimizes the instruction scheduling so that your executable runs faster.

New Data Types

New C data types, representing the new registers are used as the operands to these intrinsic functions. These data types are listed in the New Data Types Available for Intrinsic Extensions table.

New Data Types Available for Intrinsic Extensions

New data type	MMX technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2 instructions
__m64	Yes	Yes	Yes
__m128	Not available	Yes	Yes
__m128d	Not available	Not available	Yes
__m128i	Not available	Not available	Yes

The __m64 data type

The **__m64** data type is used to represent the contents of an MMX register, which is the register used by the MMX technology intrinsics. The **__m64** data type can hold eight 8-bit values, four 16-bit values, two 32-bit values, or one 64-bit value.

The __m128 data types

The compiler aligns **__m128** local data to 16 bytes boundaries on the stack. Global data of these types is also 16-byte aligned.

To align integer, float, or double arrays, you can use the *declspec* alignment.

Because the new instruction set treats the SSE registers in the same way whether you are using packed or scalar data, there is no **__m32** data type to represent scalar data as you might expect. For scalar operations, you should use the **__m128** objects and the scalar forms of the intrinsics; the compiler and the processor implement these operations with 32-bit memory references.

New Data Types Usage Guidelines

The new data types listed in the New Data Types Available for Intrinsic Extensions table are not basic ANSI C data types, and therefore you must observe the following usage restrictions:

- Use new data types only on the left side of an assignment as a return value or as a parameter. You cannot use it with other arithmetic expressions (" + ", " ", and so on).
- Use new data types as objects in aggregates, such as unions, to access the byte elements and structures. The address of an **__m64** or **__m128** object may be taken.
- Use new data types only with the respective intrinsics described in this guide.

For complete details of the hardware instructions, see the *Intel Architecture MMX Technology Programmer's Reference Manual*. For descriptions of data types, see the *Intel Architecture Software Developer's Manual*, volume 2: *Instruction Set Reference Manual*.

END Microsoft Specific

See Also

Reference

Intel Overview of New Instructions and Extensions

Intrinsic Conventions

Microsoft Specific

The intrinsics use the following syntax notation convention:

```
_mm_<intrin_op>_<suffix>
```

where `<intrin_op>` indicates the intrinsic's basic operation (for example, `add` for addition and `sub` for subtraction) and `<suffix>` denotes the type of data operated on by the instruction. The first one or two letters of each suffix denotes whether the data is packed (**p**), extended packed (**ep**), or scalar (**s**).

The remaining letters denote the type:

s

Single-precision floating point

d

Double-precision floating point

i128

Signed 128-bit integer

i64

Signed 64-bit integer

u64

Unsigned 64-bit integer

i32

Signed 32-bit integer

u32

Unsigned 32-bit integer

i16

Signed 16-bit integer

u16

Unsigned 16-bit integer

i8

Signed 8-bit integer

u8

Unsigned 8-bit integer

The packed values are represented in right-to-left order, with the lowest value being used for scalar operations. Consider the following example operation:

```
double a[2] = {1.0, 2.0};
__m128d t = _mm_load_pd(a);
```

The result is the same as either of the following:

```
__m128d t = _mm_set_pd(2.0, 1.0);
__m128d t = _mm_setr_pd(1.0, 2.0);
```

In other words, the `xmm` register that holds the value `t` will look as follows:

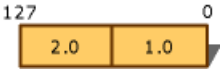

The scalar element is `1.0`. Because of the nature of the instruction, some intrinsics require their arguments to be immediates (constant integer literals).

Intrinsic Syntax

The following is the syntax for each intrinsic:

```
data_type intrinsic_name (parameters)
```

where `data_type` is the return data type (which can be **void**, **int**, **__m64**, **__m128**, **__m128d**, or **__m128i**, and only the **_mm_empty** intrinsic returns void) and `intrinsic_name` is the name of the intrinsic (which behaves like a function that you can use in your C/C++ code instead of inlining the actual instruction).

A table in each section provides the intrinsic names and the corresponding instruction. A detailed description of the intrinsic with the intrinsic's prototype and its corresponding instruction follow each table.

END Microsoft Specific

See Also
Reference
Intel Overview of New Instructions and Extensions

Intrinsic Categories and Supporting Extensions

Microsoft Specific

The intrinsics are grouped into the following categories and can be applied if you have the processors specified in the extension sets as shown in the Intrinsic Categories and Supporting Extensions table.

Intrinsic Categories and Supporting Extensions

Intrinsic category	MMX technology	Streaming SIMD Extensions	Streaming SIMD Extensions 2 instructions
General support	Yes	Not available	Not available
Integer arithmetic	Yes	Not available	Yes
Integer shift	Yes	Not available	Yes
Integer logical	Yes	Not available	Yes
Integer compare	Yes	Not available	Yes
Integer conversions	Yes	Not available	Yes
Integer set	Yes	Not available	Yes
Integer load	Not available	Not available	Yes
Integer store	Not available	Not available	Yes
Floating-point arithmetic	Not available	Yes	Yes
Floating-point logical	Not available	Yes	Yes
Floating-point compare	Not available	Yes	Yes
Floating-point conversion	Not available	Yes	Yes
Floating-point load	Not available	Yes	Yes
Floating-point set	Not available	Yes	Yes
Floating-point store	Not available	Yes	Yes
Floating-point miscellaneous	Not available	Yes	Yes
Integer miscellaneous	Not available	Yes	Yes
Cache intrinsics	Not available	Yes	Not available

END Microsoft Specific

See Also

Reference

Intel Overview of New Instructions and Extensions

AMD 3DNow! Technology Overview and Intrinsics

Microsoft Specific

The AMD 3DNow! technology is a group of instructions that opens the traditional processing bottlenecks for multimedia and floating-point-intensive applications. The 3DNow! technology enables faster frame rates on high-resolution scenes, much better physical modeling of real-world environments, sharper and more detailed 3D imaging, smoother video playback, and near-theater–quality audio.

The 3DNow! technology is compatible with today's existing x86 software and requires no operating system support, allowing 3DNow! applications to work with all existing operating systems. This technology is implemented by processors from AMD beginning with AMD-K6-2, AMD-K6-III, and AMD Athlon processors.

Beginning with the AMD Athlon processor, 3DNow! technology has been enhanced to add five new 3DNow! digital signal processing (DSP) instructions and 19 MMX Extensions, including streaming functionality.

This overview of AMD 3DNow! technology contains the following sections:

- Key Functionality
- Feature Detection
- Register Set
- Data Types
- 3DNow! Instruction Formats
- Intrinsics Overview
- Task Switching
- Exceptions
- Prefixes
- 3DNow! Intrinsics

END Microsoft Specific

See Also

Reference

MMX, SSE, and SSE2 Intrinsics

Key Functionality

Microsoft Specific

The 3DNow! technology instructions are intended to open a major processing bottleneck in a 3D graphics application: floating-point operations. Today's 3D applications are facing limitations because only one floating-point execution unit exists in the most advanced x86 processors. The front end of a typical 3D graphics software pipeline performs object physics, geometry transformations, clipping, and lighting calculations. These computations are very floating-point intensive and often limit the features and functionality of a 3D application.

The source of performance for the 3DNow! instructions originates from the single-instruction, multiple data (SIMD) implementation. With SIMD, each instruction not only operates on two single-precision, floating-point operands, but the microarchitecture within the processor can execute up to two 3DNow! instructions per clock through two register execution pipelines, which allows for a total of four floating-point operations per clock. In addition, because the 3DNow! instructions use the same floating-point registers as the MMX technology instructions, task switching between MMX and 3DNow! operations is eliminated.

The 3DNow! technology instruction set contains up to 26 instructions that support SIMD floating-point operations and includes SIMD integer operations, data prefetching, and faster MMX-to-floating-point switching. To improve MPEG decoding, the 3DNow! instructions include a specific SIMD integer instruction created to facilitate pixel-motion compensation. Because media-based software typically operates on large data sets, the processor often needs to wait for this data to be transferred from main memory. The extra time involved with retrieving this data can be avoided by using the 3DNow! instruction called PREFETCH. This instruction can ensure that data is in the level 1 cache when it is needed.

To improve the time it takes to switch between MMX and x87 code, the 3DNow! instructions include the fast entry/exit multimedia state (FEMMS) instruction, which eliminates much of the overhead involved with the switch. The addition of 3DNow! technology expands the capabilities of the AMD family of processors and enables a new generation of enriched user applications.

Enhanced 3DNow! technology instructions, first implemented on the AMD Athlon processor, provides 24 new 3DNow! Instructions. The Enhanced 3DNow! Technology consists of 5 new 3DNow! DSP instructions and 19 new MMX Extensions. Along with the new instructions, the AMD Athlon processor implements additional microarchitecture enhancements that enable more efficient operation of all these instructions, and programming can be simplified because there are fewer coding restrictions.

END Microsoft Specific

See Also

Reference

AMD 3DNow! Technology Overview and Intrinsics

Feature Detection

Microsoft Specific

To properly identify and use 3DNow! and Enhanced 3DNow! instructions, the application program must determine if the processor supports them. The CPUID instruction gives programmers the ability to determine the instructions supported by a processor. The following provides a summary of the steps needed to determine instruction set support. For a detailed description of the CPUID instruction, see the AMD Processor Recognition Application Note, order number 20734. See http://www.amd.com/devconn/3dsdk/msvc.html for more details.

1. Establish that the processor has support for CPUID.
2. Execute CPUID function 0, which returns the processor vendor string and the highest standard function supported. Save the vendor string for a later comparison. (See step 9.)
3. If step 2 indicates that the highest standard function is at least 1, execute CPUID function 1, which returns the standard feature flags in the EDX register.
4. If bit 23 of the standard feature flags is set to 1, MMX technology is supported. MMX instruction support is the basic minimum processor feature required to support other instruction extensions.
5. Optionally, if bit 25 of the standard feature flags is set, the processor has streaming SIMD extensions (SSE) capabilities. Further qualification of SSE is done by checking for OS support. SSE support might be present in the processor, but not usable due to a lack of OS support for the additional architected registers.
6. Execute CPUID extended function 8000_0000h. This function returns the highest extended function supported in EAX. If EAX=0, there is no support for extended functions.
7. If the highest extended function supported is at least 8000_0001h, execute CPUID function 8000_0001h. This function returns the extended feature flags in EDX.
8. If bit 31 of the extended feature flags is set to 1, the 3DNow! instructions are supported.
9. If the previously saved vendor string (see step 2) contains "AuthenticAMD", continue on to the next step.
10. If bit 30 of the extended feature flags is set to 1, the additions to the 3DNow! instruction set are supported.
11. If bit 22 of the extended feature flags is set to 1, the new multimedia enhancement instructions that augment the MMX instruction set are supported.

A much more comprehensive code example is available on the AMD Web site at http://www.amd.com/devconn/3dsdk/msvc.html.

END Microsoft Specific

See Also

Reference

AMD 3DNow! Technology Overview and Intrinsics

Register Set

Microsoft Specific

The multimedia units in the processor combine the existing MMX instructions with the new 3DNow! instructions. In addition, by merging 3DNow! with MMX, it becomes possible to write x86 programs containing integer, MMX, and floating-point graphics instructions with no performance penalty for switching between the multimedia (integer) and 3DNow! (floating-point) units.

The processor implements eight 64-bit 3DNow!/MMX registers. These registers are mapped onto the floating-point registers. As shown in the following figure, the 3DNow! and MMX instructions refer to these registers as MM0 to MM7. Mapping the new 3DNow!/MMX registers onto the floating-point register stack enables backwards compatibility for the register saving that must occur as a result of task switching.

3DNow!/MMX Registers

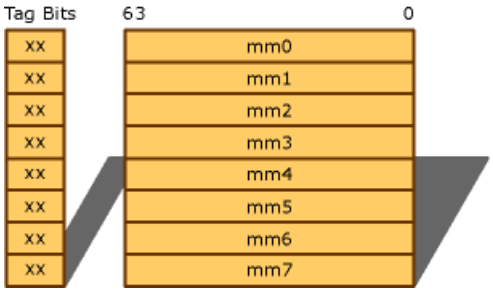

Aliasing the 3DNow!/MMX registers onto the floating-point register stack helps provide an excellent method to introduce 3DNow! and MMX technology, because it does not require modification to existing operating systems. Instead of requiring operating system modifications, new 3DNow! and MMX technology applications are supported through device drivers, 3DNow! and MMX libraries, or dynamic-link library (DLL) files.

Current operating systems have support for floating-point operations and the floating-point register state. Using the floating-point registers for 3DNow! and MMX code is a convenient way of implementing nonintrusive support for 3DNow! and MMX instructions. Every time the processor executes a 3DNow! or MMX instruction, all floating-point register tag bits are set to zero (00b=valid), except for the FEMMS and EMMS instructions, which set all tag bits to one (11b=empty).

Executing the PREFETCH instruction does not change the tag bits.

END Microsoft Specific

See Also

Reference

AMD 3DNow! Technology Overview and Intrinsics

Data Types (Compiler Intrinsics)

Microsoft Specific

3DNow! technology uses a packed data format where the data is packed in a single, 64-bit 3DNow!/MMX register or a quadword memory operand, comprising two IEEE 32-bit single-precision, floating-point doublewords.

The following figure shows the 3DNow! floating-point data type. D0 and D1 each hold an IEEE 32-bit single-precision, floating-point doubleword.

3DNow! Data Type

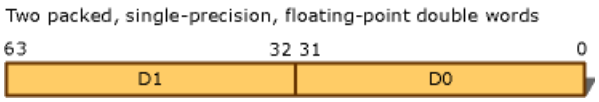

The following figure shows the IEEE 32-bit single-precision, floating-point format.

Single-Precision, Floating-Point Data Format

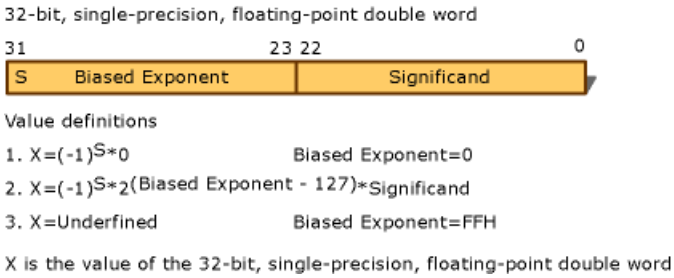

The final figure shows the formats for the integer data types.

Integer Data Types

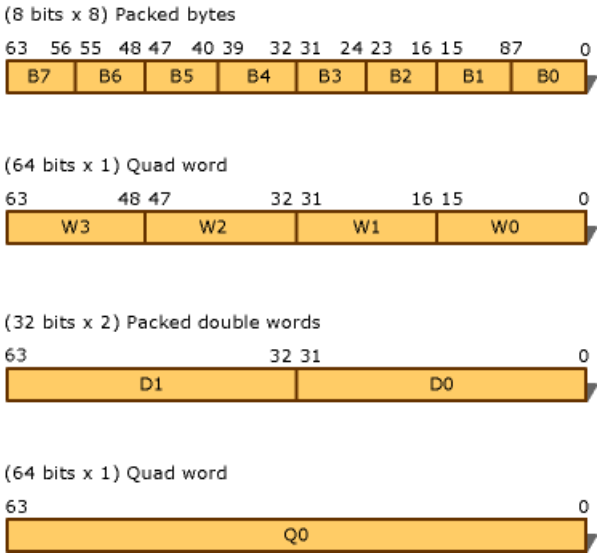

See __m64 for more information.

END Microsoft Specific

See Also

Reference

AMD 3DNow! Technology Overview and Intrinsics

3DNow! Instruction Formats

Microsoft Specific

The format of 3DNow! instruction encodings is based on the conventional x86 modR/M instruction format and is similar to the format used by MMX instructions. The assembly language syntax used for the 3DNow! instructions is as follows:

```
3DNow! Mnemonic   mmreg1, mmreg2/mem64
```

The destination and source1 operand (`mmreg1`) must be an MMX register (MM0–MM7). The source2 operand (`mmreg2/mem64`) can be either an MMX register or a 64-bit memory value.

The encoding uses the opcode prefix 0Fh followed by a second opcode byte of 0Fh. To differentiate the various 3DNow! instructions, a third instruction suffix byte is used. This suffix byte occupies the same position at the end of a 3DNow! instruction as would an imm8 byte. The opcode format is as follows:

```
0Fh 0Fh modR/M [sib] [displacement] 3DNow!_suffix
```

The specific operands (`mmreg1` and `mmreg2/mem64`) determine the values used in `modR/M [sib] [displacement]` and follow conventional x86 encodings. The 3DNow! suffix is determined by the actual 3DNow! instruction.

As an example, the 3DNow! PFMUL instruction can produce the following opcodes, depending on its use:

Opcode	Instruction
0F 0F CA B4	PFMUL mm1, mm2
0F 0F 0B B4	PFMUL mm1, [ebx]
0F 0F 4B 0A B4	PFMUL mm1, [ebx+10]
26 0F 0F 0B B4	PFMUL mm1, es:[ebx]
0F 0F 4C 83 0A B4	PFMUL mm1, [ebx+eax*4+10]

The encoding of the two performance enhancement instructions (FEMMS and PREFETCH) uses a single opcode prefix 0Fh.

END Microsoft Specific
See Also
Reference
AMD 3DNow! Technology Overview and Intrinsics

Intrinsics Overview

Microsoft Specific

3DNow! technology provides up to 26 additional instructions to support high-performance 3D graphics and audio processing. 3DNow! instructions are vector instructions that operate on 64-bit registers. 3DNow! instructions are SIMD; each instruction operates on pairs of 32-bit values. See 3DNow! Intrinsics for the reference documentation for the AMD intrinsics.

Vector instructions operate in parallel on two sets of 32-bit single-precision, floating-point words. Scalar instructions operate on a single set of 32-bit operands (from the low halves of the two 64-bit operands).

The 3DNow! single-precision, floating-point format is compatible with the IEEE 754 single-precision format. This format comprises a 1-bit sign, an 8-bit biased exponent, and a 23-bit significand with one hidden integer bit for a total of 24 bits in the significand. The bias of the exponent is 127, consistent with the IEEE single-precision standard. The significands are normalized to be within the range of [1,2).

In contrast to the IEEE standard that dictates four rounding modes, 3DNow! technology supports one rounding mode, as either round-to-nearest or round-to-zero (truncation). The hardware implementation of 3DNow! technology determines the rounding mode. The AMD processors implement round-to-nearest mode. Regardless of the rounding mode used, the floating-point-to-integer and integer-to-floating-point conversion instructions, PF2ID and PI2FD, always use the round-to-zero (truncation) mode.

The largest representable normal number in magnitude for this precision in hexadecimal has an exponent of FEh and a significand of 7FFFFFh, with a numerical value of 2127 (2 – 2–23). All results that overflow above the maximum representable positive value are saturated to either this maximum representable normal number or to positive infinity. Similarly, all results that overflow below the minimum representable negative value are saturated to either this minimum representable normal number or to negative infinity.

The implementation of 3DNow! technology determines how arithmetic overflow is handled, as either properly signed maximum or minimum representable normal numbers or properly signed infinities. The processor generates properly signed maximum or minimum representable normal numbers.

Infinities and NANs are not supported as operands to 3DNow! instructions.

The smallest representable normal number in magnitude for this precision in hexadecimal has an exponent of 01h and a significand of 000000h, with a numerical value of 2–126. Accordingly, all results below this minimum representable value in magnitude are held to zero. The following table shows the exponent ranges supported by the 3DNow! technology.

3DNow! Technology Exponent Ranges

Biased exponent	Description
FFh	Unsupported. Unsupported numbers can be used as operands. The results of operations with unsupported numbers are undefined.
00h	Zero.
00h<x<FFh	Normal.
01h	2 (1–127) lowest possible exponent.
FEh	2 (254–127) largest possible exponent.

Like MMX instructions, 3DNow! instructions do not generate numeric exceptions or set any status flags. It is the user's responsibility to ensure that in-range data is provided to 3DNow! instructions and that all computations remain within valid ranges (or are held as expected).

The register operations of all 3DNow! floating-point instructions are executed by either the register X unit or the register Y unit. One operation can be issued to each register unit at each clock cycle for a maximum issue and execution rate of two 3DNow! operations per cycle.

Normally, in high-performance 3DNow! code, all 3DNow! instructions are properly scheduled apart from each other to avoid delays caused by execution resource contentions (as well as taking into account dependencies and execution latencies).

For further information regarding code optimization on the AMD-K6 processor, see the *AMD-K6 Processor Code Optimization Application Note*, order number 21924. This document provides in-depth discussions of code optimization techniques for the processor.

For execution resources information on the AMD Athlon processor, refer to the *AMD Athlon Processor x86 Code Optimization Guide*, order number 22007.

The 3DNow! performance enhancement instructions for AMD processors are summarized in the following tables.

AMD 3DNow! Floating-Point Instructions

Operation	Function	Opcode
PAVGUSB	Packed 8-bit unsigned integer averaging	BFh
PFADD	Packed floating-point addition	9Eh
PFSUB	Packed floating-point subtraction	9Ah
PFSUBR	Packed floating-point reverse subtraction	Aah
PFACC	Packed floating-point accumulate	Aeh
PFCMPGE	Packed floating-point comparison, greater or equal	90h
PFCMPGT	Packed floating-point comparison, greater	A0h
PFCMPEQ	Packed floating-point comparison, equal	B0h
PFMIN	Packed floating-point minimum	94h
PFMAX	Packed floating-point maximum	A4h
PI2FD	Packed 32-bit integer to floating-point conversion	0Dh
PF2ID	Packed floating-point to 32-bit integer	1Dh
PFRCP	Packed floating-point reciprocal approximation	96h
PFRSQRT	Packed floating-point reciprocal square root approximation	97h
PFMUL	Packed floating-point multiplication	B4h
PFRCPIT1	Packed floating-point reciprocal first iteration step	A6h
PFRSQIT1	Packed floating-point reciprocal square root first iteration step	A7h
PFRCPIT2	Packed floating-point reciprocal/reciprocal square root second iteration step	B6h
PMULHRW	Packed 16-bit integer multiply with rounding	B7h

AMD 3DNow! Performance Enhancement Instructions

Operation	Function	Opcode second byte
FEMMS	Faster entry/exit of the MMX or floating-point state.	0Eh

PREFETCH/PREFETCHW	• The function prefetches at least a 32-byte line into L1 data cache (Dcache). • The AMD-K6-2 and AMD-K6-III processors execute the PREFETCHW instruction identically to the PREFETCH instruction. • On the AMD Athlon processor, PREFETCHW can increase performance by providing a hint to the processor of an intent to modify the cache line.	0Dh

AMD Athlon Processor 3DNow! Technology DSP Extensions

Operation	Function	Opcode / imm8
PF2IW	Packed floating-point to integer word conversion with sign extend	0Fh 0Fh / 1Ch
PFNACC	Packed floating-point negative accumulate	0Fh 0Fh / 8Ah
PFPNACC	Packed floating-point mixed positive-negative accumulate	0Fh 0Fh / 8Eh
PI2FW	Packed integer word to floating-point conversion	0Fh 0Fh / 0Ch
PSWAPD	Packed swap doubleword	0Fh 0Fh / BBh

MMX Instruction set extensions starting with AMD Athlon Processor

Operation	Function	Opcode / imm8
MASKMOVQ	Streaming (cache bypass) store using byte mask	0Fh F7h
MOVNTQ	Streaming (cache bypass) store	0Fh E7h
PAVGB	Packed average of unsigned byte	0Fh E0h
PAVGW	Packed average of unsigned word	0Fh E3h
PEXTRW	Extract word into integer register	0Fh C5h
PINSRW	Insert word from integer register	0Fh C4h
PMAXSW	Packed maximum signed word	0Fh Eeh
PMAXUB	Packed maximum unsigned byte	0Fh Deh
PMINSW	Packed minimum signed word	0Fh Eah
PMINUB	Packed minimum unsigned byte	0Fh Dah
PMOVMSKB	Move byte mask to integer register	0Fh D7h
PMULHUW	Packed multiply high unsigned word	0Fh E4h
PREFETCHNTA	Move data closer to the processor using the NTA reference	0Fh 18h 0*
PREFETCHT0	Move data closer to the processor using the T0 reference	0Fh 18h 1*
PREFETCHT1	Move data closer to the processor using the T1 reference	0Fh 18h 2*
PREFETCHT2	Move data closer to the processor using the T2 reference	0Fh 18h 3*

PSADBW	Packed sum of absolute byte differences	0Fh F6h
PSHUFW	Packed shuffle word	0Fh 70h
SFENCE	Store fence	0Fh AEh / 7h

*The number after the opcode indicates the different prefetch modes in the modR/M byte.

For further information regarding code optimization on the AMD-K6-2 processor, see the AMD-K6-2 Processor Code Optimization Application Note, order number 21924. This document provides in-depth discussions of code optimization techniques for the AMD-K6 family processor.

For execution resources information on the AMD Athlon processor, refer to the AMD Athlon Processor x86 Code Optimization Guide, order number 22007. This document provides in-depth discussions of code optimization techniques for the AMD Athlon processor.

See http://www.amd.com/devconn/3dsdk/msvc.html for the online versions of these documents.

END Microsoft Specific
See Also
Reference
AMD 3DNow! Technology Overview and Intrinsics

Task Switching

Microsoft Specific

With respect to task switching, treat the 3DNow! instructions exactly the same as MMX instructions. Operating system design must be taken into account when writing a 3DNow! program.

The programmer must know whether the operating system automatically saves the current states when task switching, or if the 3DNow! program has to provide the code to save states. All of the Microsoft 32-bit and 64-bit operating systems save the current state when task switching.

If a task switch occurs, the control register (CR0) task switch (TS) bit is set to 1. The processor then generates an interrupt 7 (int 7 — Device Not Available) when it encounters the next floating-point, 3DNow!, or MMX instruction, allowing the operating system to save the state of the 3DNow!, MMX, and FP registers.

In a multitasking operating system, if there is a task switch when 3DNow!/MMX applications are running with older applications that do not include MMX instructions, the MMX/FP register state is still saved automatically through the int 7 handler.

END Microsoft Specific

See Also

Reference

AMD 3DNow! Technology Overview and Intrinsics

Exceptions (AMD Intrinsics)

Microsoft Specific

The following table contains a list of exceptions that 3DNow! and MMX instructions can generate.

3DNow! and MMX Instruction Exceptions

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	Yes	Yes	Yes	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	Yes	Yes	Yes	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)	Yes	Yes	Yes	During instruction execution, the stack segment limit was exceeded.
General protection (13)	No	No	Yes	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	Yes	Yes	No	One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)	No	Yes	Yes	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	Yes	Yes	Yes	An exception is pending because of the floating-point execution unit.
Alignment check (17)	No	Yes	Yes	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The rules for exceptions are the same for both MMX and 3DNow! instructions. In addition, exception detection and handling is identical for MMX and 3DNow! instructions. Exception handlers do not need modification.

An invalid opcode exception (interrupt 6) occurs if a 3DNow! instruction is executed on a processor that does not support 3DNow! instructions.

If a floating-point exception is pending and the processor encounters a 3DNow! instruction, FERR# is asserted and, if CR0.NE = 1, an interrupt 16 is generated. This is the same for MMX instructions.

END Microsoft Specific

See Also

Reference

AMD 3DNow! Technology Overview and Intrinsics

Prefixes

Microsoft Specific

The following prefixes can be used with 3DNow! instructions:

- The segment override prefixes (2Eh/CS, 36h/SS, 3Eh/DS, 26h/ES, 64h/FS, and 65h/GS) affect 3DNow! instructions that contain a memory operand.
- The address-size override prefix (67h) affects 3DNow! instructions that contain a memory operand.
- The operand-size override prefix (66h) is ignored.
- The LOCK prefix (F0h) triggers an invalid opcode exception (interrupt 6).
- The REP prefixes (F3h/ REP/ REPE/ REPZ, F2h/ REPNE/ REPNZ) are ignored.

END Microsoft Specific

See Also

Reference

AMD 3DNow! Technology Overview and Intrinsics

3DNow! Intrinsics

Microsoft Specific

This topic contains the 3DNow! intrinsics. For each intrinsic, the header file mm3dnow.h is required.

The following table lists the 3DNow! intrinsics alphabetically.

Intrinsic	Use
_m_femms	Clears the architectural state when switching between MMX and floating-point instructions.
_m_from_float	Returns a 64-bit MMX value where the lower half is set to the floating-point, single-precision value from the source operand and the upper half is zero. There is no error return.
_m_pavgusb	Calculates the rounded averages of eight unsigned 8-bit integer values.
_m_pf2id	Converts packed floating-point, single-precision values to packed 32-bit integer values.
_m_pf2iw	Converts packed floating-point, single-precision values to packed 16-bit signed integer values using truncation.
_m_pfacc	Performs packed floating-point, single-precision accumulation.
_m_pfadd	Performs packed floating-point, single-precision addition.
_m_pfcmpeq	Compares packed floating-point, single-precision values to be equal and sets the corresponding return value to ones or zeros based on the result of the comparison.
_m_pfcmpge	Compares the first packed floating-point, single-precision value to be greater than or equal to the second one and sets the corresponding return value to ones or zeros based on the result of the comparison.
_m_pfcmpgt	Compares the first packed floating-point, single-precision value to be greater than the second one and sets the corresponding return value to ones or zeros based on the result of the comparison.
_m_pfmax	Returns the larger of the two packed floating-point, single-precision values.
_m_pfmin	Returns the smaller of the two packed floating-point, single-precision values.
_m_pfmul	Performs packed floating-point, single-precision multiplication.
_m_pfnacc	Performs packed floating-point, single-precision negative accumulation.
_m_pfpnacc	Performs packed floating-point, single-precision positive-negative accumulation.
_m_pfrcp	Performs scalar floating-point, low-precision reciprocal approximation.
_m_pfrcpit1	Performs the first intermediate step in the Newton-Raphson iteration to refine the reciprocal approximation produced by the **_m_pfrcp** intrinsic function.
_m_pfrcpit2	Performs the second and final step in the Newton-Raphson iteration to refine the reciprocal or reciprocal square root approximation produced by the **_m_pfrcp** or **_m_pfsqrt** intrinsic functions, respectively.
_m_pfrsqit1	Performs the first intermediate step in the Newton-Raphson iteration to refine the reciprocal square root approximation produced by **_m_pfsqrt** intrinsic function.

_m_pfrsqrt	Performs scalar floating-point, low-precision reciprocal square root approximation.
_m_pfsub	Performs packed floating-point, single-precision subtraction.
_m_pfsubr	Performs packed floating-point, single-precision reverse subtraction.
_m_pi2fd	Converts packed 32-bit integer values to packed floating-point, single-precision values.
_m_pi2fw	Converts packed 16-bit signed integer values to packed floating-point, single-precision values.
_m_pmulhrw	Multiplies four signed 16-bit integer values in the source operand by four signed 16-bit integer values in the destination operand.
_m_prefetch	Loads a 32-byte cache line into L1 data cache and sets the cache line state to **exclusive**.
_m_prefetchw	Loads a 32-byte cache line into L1 data cache and sets the cache line state to **modified**.
_m_pswapd	Swaps upper and lower halves of the source operand.
_m_to_float	Returns the floating-point, single-precision value from the lower half of the 64-bit MMX value in the source operand. There is no error return. The compiler correctly ensures that an implict FEMMS is issued before any attempt to use the result of _m_to_float( ) operation.

END Microsoft Specific

See Also

Reference

AMD 3DNow! Technology Overview and Intrinsics

_m_femms

Clears the architectural state when switching between MMX and floating-point instructions.

```
void _m_femms( void );
```

Remarks

The contents of MMX registers and floating-point stack are undefined after the function is executed.

See Also

Reference

3DNow! Intrinsics

_m_from_float

Microsoft Specific

Returns a 64-bit MMX value where the lower half is set to the floating-point, single-precision value from the source operand and the upper half is zero. There is no error return.

```
__m64 _m_from_float(float f);
```

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also

Reference

3DNow! Intrinsics

_m_pavgusb

Microsoft Specific

Calculates the rounded averages of eight unsigned 8-bit integer values.

```
__m64 _m_pavgusb( __m64 m1, __m64 m2 );
```

Remarks

The **_m_pavgusb** function takes two 64-bit MMX values and returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also
Reference
3DNow! Intrinsics

_m_pf2id

Microsoft Specific

Converts packed floating-point, single-precision values to packed 32-bit integer values.

```
__m64 _m_pf2id( __m64 m );
```

Remarks

The **_m_pf2id** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also
Reference
3DNow! Intrinsics

_m_pf2iw

Microsoft Specific

Converts packed floating-point, single-precision values to packed 16-bit signed integer values using truncation.

```
__m64 _m_pf2iw( __m64 m );
```

Remarks

The **_m_pf2iw** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also
Reference
3DNow! Intrinsics

_m_pfacc

Microsoft Specific

Performs packed floating-point, single-precision accumulation.

```
__m64 _m_pfacc( __m64 m1, __m64 m2 );
```

Remarks

The **_m_pfacc** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also

Reference

3DNow! Intrinsics

_m_pfadd

Microsoft Specific

Performs packed floating-point, single-precision addition.

```
__m64 _m_pfadd( __m64 m1, __m64 m2 );
```

Remarks

The **_m_pfadd** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also

Reference

3DNow! Intrinsics

_m_pfcmpeq

Microsoft Specific

Compares packed floating-point, single-precision values to be equal and sets the corresponding return value to ones or zeros based on the result of the comparison.

```
__m64 _m_pfcmpeq( __m64 m1, __m64 m2 );
```

Remarks

The **_m_pfcmpeq** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also
Reference
3DNow! Intrinsics

_m_pfcmpge

Microsoft Specific

Compares the first packed floating-point, single-precision value to be greater than or equal to the second one and sets the corresponding return value to ones or zeros based on the result of the comparison.

```
__m64 _m_pfcmpge( __m64 m1, __m64 m2 );
```

Remarks

The **_m_pfcmpge** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also
Reference
3DNow! Intrinsics

_m_pfcmpgt

Microsoft Specific

Compares the first packed floating-point, single-precision value to be greater than the second one and sets the corresponding return value to ones or zeros based on the result of the comparison.

```
__m64 _m_pfcmpgt( __m64 m1, __m64 m2 );
```

Remarks

The **_m_pfcmpgt** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also
Reference
3DNow! Intrinsics

_m_pfmax

Microsoft Specific

Returns the larger of the two packed floating-point, single-precision values.

```
__m64 _m_pfmax( __m64 m1, __m64 m2 );
```

Remarks

The **_m_pfmax** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also

Reference

3DNow! Intrinsics

_m_pfmin

Microsoft Specific

Returns the smaller of the two packed floating-point, single-precision values.

```
__m64 _m_pfmin( __m64 m1, __m64 m2 );
```

Remarks

The **_m_pfmin** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also

Reference

3DNow! Intrinsics

_m_pfmul

Microsoft Specific

Performs packed floating-point, single-precision multiplication.

```
__m64 _m_pfmul( __m64 m1, __m64 m2 );
```

Remarks

The **_m_pfmul** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also

Reference

3DNow! Intrinsics

_m_pfnacc

Microsoft Specific

Performs packed floating-point, single-precision negative accumulation.

```
__m64 _m_pfnacc( __m64 m1, __m64 m2 );
```

Remarks

The **_m_pfnacc** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also

Reference

3DNow! Intrinsics

_m_pfpnacc

Microsoft Specific

Performs packed floating-point, single-precision positive-negative accumulation.

```
__m64 _m_pfpnacc( __m64 m1, __m64 m2 );
```

Remarks

The **_m_pfpnacc** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also

Reference

3DNow! Intrinsics

_m_pfrcp

Microsoft Specific

Performs scalar floating-point, low-precision reciprocal approximation.

```
__m64 _m_pfrcp( __m64 m );
```

Remarks

The 14-bit accurate result is duplicated in both high and low halves of the return value. The **_m_pfrcp** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also

Reference

3DNow! Intrinsics

_m_pfrcpit1

Microsoft Specific

Performs the first intermediate step in the Newton-Raphson iteration to refine the reciprocal approximation produced by the **_m_pfrcp** intrinsic function.

```
__m64 _m_pfrcpit1( __m64 m1, __m64 m2 );
```

Remarks

The **_m_pfrcpit1** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also

Reference

3DNow! Intrinsics

_m_pfrcpit2

Microsoft Specific

Performs the second and final step in the Newton-Raphson iteration to refine the reciprocal or reciprocal square root approximation produced by the **_m_pfrcp** or **_m_pfsqrt** intrinsic functions, respectively.

```
__m64 _m_pfrcpit2( __m64 m1, __m64 m2 );
```

Remarks

The **_m_pfrcpit2** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also
Reference
3DNow! Intrinsics

_m_pfrsqit1

Microsoft Specific

Performs the first intermediate step in the Newton-Raphson iteration to refine the reciprocal square root approximation produced by **_m_pfsqrt** intrinsic function.

```
__m64 _m_pfrsqit1( __m64 m1, __m64 m2 );
```

Remarks

The **_m_pfrsqit1** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also
Reference
3DNow! Intrinsics

_m_pfrsqrt

Microsoft Specific

Performs scalar floating-point, low-precision reciprocal square root approximation.

```
__m64 _m_pfrsqrt( __m64 m );
```

Remarks

The 15-bit accurate result is duplicated in both high and low halves of the return value. The **_m_pfrsqrt** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also
Reference
3DNow! Intrinsics

_m_pfsub

Microsoft Specific

Performs packed floating-point, single-precision subtraction.

```
__m64 _m_pfsub( __m64 m1, __m64 m2 );
```

Remarks

The **_m_pfsub** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also

Reference

3DNow! Intrinsics

_m_pfsubr

Microsoft Specific

Performs packed floating-point, single-precision reverse subtraction.

```
__m64 _m_pfsubr( __m64 m1, __m64 m2 );
```

Remarks

The **_m_pfsubr** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also

Reference

3DNow! Intrinsics

_m_pi2fd

Microsoft Specific

Converts packed 32-bit integer values to packed floating-point, single-precision values.

```
__m64 _m_pi2fd( __m64 m );
```

Remarks

The **_m_pi2fd** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also

Reference

3DNow! Intrinsics

_m_pi2fw

Microsoft Specific

Converts packed 16-bit signed integer values to packed floating-point, single-precision values.

```
__m64 _m_pi2fw( __m64 m );
```

Remarks

The **_m_pi2fw** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also
Reference
3DNow! Intrinsics

_m_pmulhrw

Microsoft Specific

Multiplies four signed 16-bit integer values in the source operand by four signed 16-bit integer values in the destination operand.

```
__m64 _m_pmulhrw( __m64 m1, __m64 m2 );
```

Remarks

Each high-order 16-bit result is rounded. The **_m_pmulhrw** function returns a 64-bit MMX value. There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also
Reference
3DNow! Intrinsics

_m_prefetch

Microsoft Specific

Loads a 32-byte cache line into L1 data cache and sets the cache line state to **exclusive**.

```
void _m_prefetch( void* p );
```

Remarks

If the line is already in the cache or if a memory fault is detected, then the intrinsic function has no effect. The variable `p` specifies the address of the cache line to be loaded.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also

Reference

3DNow! Intrinsics

_m_prefetchw

Microsoft Specific

Loads a 32-byte cache line into L1 data cache and sets the cache line state to **modified**.

```
void _m_prefetchw( volatile const void* p );
```

Remarks

If the line is already in the cache or if a memory fault is detected, then the intrinsic function has no effect. The variable `p` specifies the address of the cache line to be loaded.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also

Reference

3DNow! Intrinsics

_m_pswapd

Microsoft Specific

Swaps upper and lower halves of the source operand and returns a 64-bit MMX value.

```
__m64 _m_pswapd( __m64 m );
```

Remarks

There is no error return.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also
Reference
3DNow! Intrinsics

_m_to_float

Microsoft Specific

Returns the floating-point, single-precision value from the lower half of the 64-bit MMX value in the source operand.

```
float _m_to_float( __m64 m );
```

Remarks

There is no error return.

The compiler correctly ensures that an implict FEMMS is issued before any attempt to use the result of _m_to_float( ) operation.

Requirements

Header: mm3dnow.h

END Microsoft Specific

See Also

Reference

3DNow! Intrinsics

MMX Technology

Microsoft Specific

Intel's MMX technology is an extension of the Intel architecture (IA) instruction set. The technology uses a single-instruction, multiple-data (SIMD) technique to speed up multimedia and communications software by processing data elements in parallel.

The MMX instruction set adds 57 opcodes and a 64-bit quadword data type. In addition, there are eight 64-bit MMX technology registers, each of which can be directly addressed using the register names MM0 to MM7. The figure in Understanding the EMMS Instruction shows the layout of the eight MMX technology registers.

The MMX technology is operating-system transparent and 100 percent compatible with all existing Intel architecture software. Therefore, all applications will continue to run on processors with MMX technology. Additional information and details about the MMX instructions, data types, and registers can be found in the *Intel Architecture MMX Technology Programmer's Reference Manual*, order number 243007.

MMX intrinsics use the __m64 data type, which is not supported on x64 processors.

- MMX Technology Intrinsic Groups
- Understanding the EMMS Instruction
- Guidelines for When to Use EMMS

The header file mmintrin.h contains the declarations for the MMX intrinsics. The file ivec.h contains operator overloads for some of the MMX intrinsics, which are available for use in C++ programs.

END Microsoft Specific

See Also

Tasks

MMXSwarm Sample: Demonstrates CImage and Visual C++ MMX Support

Reference

MMX, SSE, and SSE2 Intrinsics

MMX Technology Intrinsic Groups

Microsoft Specific

The MMX technology intrinsics are grouped into the following sets:

- MMX Technology General Support intrinsics
- MMX Technology Packed Arithmetic Intrinsics
- MMX Technology Shift Intrinsics
- MMX Technology Logical Intrinsics
- MMX Technology Compare Intrinsics
- MMX Technology Set Intrinsics

END Microsoft Specific
See Also
Reference
MMX Technology

MMX Technology General Support Intrinsics

Microsoft Specific

The intrinsics listed in the following table are followed by a description of each intrinsic.

MMX intrinsics use the __m64 data type, which is not supported on x64 processors.

The header file mmintrin.h contains the declarations for the MMX intrinsics.

General Support Intrinsics

Intrinsic name	Operation	Signed	Saturation	Assembly instruction
_mm_empty	Empties MM state	Not applicable	Not applicable	EMMS
_mm_cvtsi32_si64	Converts from int	Not applicable	Not applicable	MOVD
_mm_cvtsi64_si32	Converts from int	Not applicable	Not applicable	MOVD
_mm_packs_pi16	Packs	Yes	Yes	PACKSSWB
_mm_packs_pi32	Packs	Yes	Yes	PACKSSDW
_mm_packs_pu16	Packs	No	Yes	PACKUSWB
_mm_unpackhi_pi8	Interleaves	Not applicable	Not applicable	PUNPCKHBW
_mm_unpackhi_pi16	Interleaves	Not applicable	Not applicable	PUNPCKHWD
_mm_unpackhi_pi32	Interleaves	Not applicable	Not applicable	PUNPCKHDQ
_mm_unpacklo_pi8	Interleaves	Not applicable	Not applicable	PUNPCKLBW
_mm_unpacklo_pi16	Interleaves	Not applicable	Not applicable	PUNPCKLWD
_mm_unpacklo_pi32	Interleaves	Not applicable	Not applicable	PUNPCKLDQ

END Microsoft Specific

See Also

Reference

MMX Technology Intrinsic Groups

_mm_empty

Microsoft Specific

Empties the multimedia state.

```
void _mm_empty (void);
EMMS
```

Remarks

See the figure in Understanding the EMMS Instruction for details. Also see Guidelines for When to Use EMMS. A synonym for **_mm_empty** is **_m_empty**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology General Support Intrinsics

_mm_cvtsi32_si64

Microsoft Specific

Converts the integer object `i` to a 64-bit `__m64` object.

```
__m64 _mm_cvtsi32_si64 (int i );
MOVD
```

Remarks

The integer value is zero extended to 64 bits. A synonym for **_mm_cvtsi32_si64** is **_m_from_int**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology General Support Intrinsics

_mm_cvtsi64_si32

Microsoft Specific

Converts the lower 32 bits of the `__m64` object `m` to an integer.

```
int _mm_cvtsi64_si32 ( __m64 m );
MOVD
```

Remarks

A synonym for **_mm_cvtsi64_si32** is **_m_to_int**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology General Support Intrinsics

_mm_packs_pi16

Microsoft Specific

Packs the four 16-bit values from `m1` into the lower four 8-bit values of the result with signed saturation, and packs the four 16-bit values from `m2` into the upper four 8-bit values of the result with signed saturation.

```
__m64 _mm_packs_pi16 (__m64 m1, __m64 m2);
PACKSSWB
```

Remarks

A synonym for **_mm_packs_pi16** is **_m_packsswb**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology General Support Intrinsics

_mm_packs_pi32

Microsoft Specific

Packs the two 32-bit values from `m1` into the lower two 16-bit values of the result with signed saturation, and packs the two 32-bit values from `m2` into the upper two 16-bit values of the result with signed saturation.

```
__m64 _mm_packs_pi32 (__m64 m1, __m64 m2);
PACKSSDW
```

Remarks

A synonym for **_mm_packs_pi32** is **_m_packssdw**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology General Support Intrinsics

_mm_packs_pu16

Microsoft Specific

Packs the four 16-bit values from `m1` into the lower four 8-bit values of the result with unsigned saturation, and packs the four 16-bit values from `m2` into the upper four 8-bit values of the result with unsigned saturation.

```
__m64 _mm_packs_pu16 (__m64 m1, __m64 m2);
PACKUSWB
```

Remarks

A synonym for **_mm_packs_pu16** is **_m_packuswb**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology General Support Intrinsics

_mm_unpackhi_pi8

Microsoft Specific

Interleaves the four 8-bit values from the high half of `m1` with the four values from the high half of `m2` and takes the least significant element from `m1`.

```
__m64 _mm_unpackhi_pi8 (__m64 m1 , __m64 m2);
PUNPCKHBW
```

Remarks

A synonym for **_mm_unpackhi_pi8** is **_m_punpckhbw**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology General Support Intrinsics

_mm_unpackhi_pi16

Microsoft Specific

Interleaves the two 16-bit values from the high half of `m1` with the two values from the high half of `m2` and takes the least significant element from `m1`.

```
__m64 _mm_unpackhi_pi16 (__m64 m1 , __m64 m2);
PUNPCKHWD
```

Remarks

A synonym for **_mm_unpackhi_pi16** is **_m_punpckhwd**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology General Support Intrinsics

_mm_unpackhi_pi32

Microsoft Specific

Interleaves the 32-bit value from the high half of `m1` with the 32-bit value from the high half of `m2` and takes the least significant element from `m1`.

```
__m64 _mm_unpackhi_pi32 (__m64 m1 , __m64 m2);
PUNPCKHDQ
```

Remarks

A synonym for **_mm_unpackhi_pi32** is **_m_punpckhdq**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology General Support Intrinsics

_mm_unpacklo_pi8

Microsoft Specific

Interleaves the four 8-bit values from the low half of `m1` with the four values from the low half of `m2` and takes the least significant element from `m1`.

```
__m64 _mm_unpacklo_pi8 (__m64 m1 , __m64 m2);
PUNPCKLBW
```

Remarks

A synonym for **_mm_unpacklo_pi8** is **_p_punpcklbw**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology General Support Intrinsics

_mm_unpacklo_pi16

Microsoft Specific

Interleaves the two 16-bit values from the low half of `m1` with the two values from the low half of `m2` and takes the least significant element from `m1`.

```
__m64 _mm_unpacklo_pi16 (__m64 m1 , __m64 m2);
PUNPCKLWD
```

Remarks

A synonym for **_mm_unpacklo_pi16** is **_m_punpcklwd**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology General Support Intrinsics

_mm_unpacklo_pi32

Microsoft Specific

Interleaves the 32-bit value from the low half of `m1` with the 32-bit value from the low half of `m2` and takes the least significant element from `m1`.

```
__m64 _mm_unpacklo_pi32 (__m64 m1 , __m64 m2);
PUNPCKLDQ
```

Remarks

A synonym for **_mm_unpacklo_pi32** is **_m_punpckldq**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology General Support Intrinsics

MMX Technology Packed Arithmetic Intrinsics

Microsoft Specific

The intrinsics listed in the following table are followed by a description of each intrinsic.

MMX intrinsics use the __m64 data type, which is not supported on x64 processors.

The header file mmintrin.h contains the declarations for the MMX intrinsics.

Packed Arithmetic Intrinsics

Intrinsic name	Operation	Signed	Argument and result values/bits	Corresponding instruction
_mm_add_pi8	Adds	Not applicable	8/8, 8/8	PADDB
_mm_add_pi16	Adds	Not applicable	4/16, 4/16	PADDW
_mm_add_pi32	Adds	Not applicable	2/32, 2/32	PADDD
_mm_adds_pi8	Adds	Yes	8/8, 8/8	PADDSB
_mm_adds_pi16	Adds	Yes	4/16, 4/16	PADDSW
_mm_adds_pu8	Adds	No	8/8, 8/8	PADDUSB
_mm_adds_pu16	Adds	No	4/16, 4/16	PADDUSW
_mm_sub_pi8	Subtracts	Not applicable	8/8, 8/8	PSUBB
_mm_sub_pi16	Subtracts	Not applicable	4/16, 4/16	PSUBW
_mm_sub_pi32	Subtracts	Not applicable	2/32, 2/32	PSUBD
_mm_subs_pi8	Subtracts	Yes	8/8, 8/8	PSUBSB
_mm_subs_pi16	Subtracts	Yes	4/16, 4/16	PSUBSW
_mm_subs_pu8	Subtracts	No	8/8, 8/8	PSUBUSB
_mm_subs_pu16	Subtracts	No	4/16, 4/16	PSUBUSW
_mm_madd_pi16	Multiplies	Not applicable	4/16, 2/32	PMADDWD
_mm_mulhi_pi16	Multiplies	Yes	4/16, 4/16 (high)	PMULHW
_mm_mullo_pi16	Multiplies	Not applicable	4/16, 4/16 (low)	PMULLW

END Microsoft Specific

See Also

Reference

MMX Technology Intrinsic Groups

_mm_add_pi8

Microsoft Specific

Adds the eight 8-bit values in `m1` to the eight 8-bit values in `m2`.

```
__m64 _mm_add_pi8 (__m64 m1 , __m64 m2);
PADDB
```

Remarks

A synonym for **_mm_add_pi8** is **_m_paddb**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Packed Arithmetic Intrinsics

_mm_add_pi16

Microsoft Specific

Adds the four 16-bit values in `m1` to the four 16-bit values in `m2`.

```
__m64 _mm_add_pi16 (__m64 m1 , __m64 m2);
PADDW
```

Remarks

A synonym for **_mm_add_pi16** is **_m_paddw**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Packed Arithmetic Intrinsics

_mm_add_pi32

Microsoft Specific

Adds the two 32-bit values in `m1` to the two 32-bit values in `m2`.

```
__m64 _mm_add_pi32 (__m64 m1 , __m64 m2);
PADDD
```

Remarks

A synonym for **_mm_add_pi32** is **_m_paddd**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Packed Arithmetic Intrinsics

_mm_adds_pi8

Microsoft Specific

Adds the eight signed 8-bit values in `m1` to the eight signed 8-bit values in `m2` and saturates.

```
__m64 _mm_adds_pi8 (__m64 m1 , __m64 m2);
PADDSB
```

Remarks

A synonym for **_mm_adds_pi8** is **_m_paddsb**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Packed Arithmetic Intrinsics

_mm_adds_pi16

Microsoft Specific

Adds the four signed 16-bit values in `m1` to the four signed 16-bit values in `m2` and saturates.

```
__m64 _mm_adds_pi16 (__m64 m1 , __m64 m2);
PADDSW
```

Remarks

A synonym for **_mm_adds_pi16** is **_m_paddsw**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Packed Arithmetic Intrinsics

_mm_adds_pu8

Microsoft Specific

Adds the eight unsigned 8-bit values in `m1` to the eight unsigned 8-bit values in `m2` and saturates.

```
__m64 _mm_adds_pu8 (__m64 m1 , __m64 m2);
PADDUSB
```

Remarks

A synonym for **_mm_adds_pu8** is **_m_paddusb**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Packed Arithmetic Intrinsics

_mm_adds_pu16

Microsoft Specific

Add the four unsigned 16-bit values in `m1` to the four unsigned 16-bit values in `m2` and saturates.

```
__m64 _mm_adds_pu16 (__m64 m1 , __m64 m2);
PADDUSW
```

Remarks

A synonym for **_mm_adds_pu16** is **_m_paddusw**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Packed Arithmetic Intrinsics

_mm_sub_pi8

Microsoft Specific

Subtracts the eight 8-bit values in `m2` from the eight 8-bit values in `m1`.

```
__m64 _mm_sub_pi8 (__m64 m1 , __m64 m2);
PSUBB
```

Remarks

A synonym for **_mm_sub_pi18** is **_m_psubb**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Packed Arithmetic Intrinsics

_mm_sub_pi16

Microsoft Specific

Subtracts the four 16-bit values in `m2` from the four 16-bit values in `m1`.

```
__m64 _mm_sub_pi16 (__m64 m1 , __m64 m2);
PSUBW
```

Remarks

A synonym for **_mm_sub_pi16** is **_m_psubw**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Packed Arithmetic Intrinsics

_mm_sub_pi32

Microsoft Specific

Subtracts the two 32-bit values in `m2` from the two 32-bit values in `m1`.

```
__m64 _mm_sub_pi32 (__m64 m1 , __m64 m2);
PSUBD
```

Remarks

A synonym for **_mm_sub_pi32** is **_m_psubd**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Packed Arithmetic Intrinsics

_mm_subs_pi8

Microsoft Specific

Subtracts the eight signed 8-bit values in `m2` from the eight signed 8-bit values in `m1` and saturates.

```
__m64 _mm_subs_pi8 (__m64 m1 , __m64 m2);
PSUBSB
```

Remarks

A synonym for **_mm_subs_pi8** is **_m_psubsb**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Packed Arithmetic Intrinsics

_mm_subs_pi16

Microsoft Specific

Subtracts the four signed 16-bit values in `m2` from the four signed 16-bit values in `m1` and saturates.

```
__m64 _mm_subs_pi16 (__m64 m1 , __m64 m2);
PSUBSW
```

Remarks

A synonym for **_mm_subs_pi16** is **_m_psubsw**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Packed Arithmetic Intrinsics

_mm_subs_pu8

Microsoft Specific

Subtracts the eight unsigned 8-bit values in `m2` from the eight unsigned 8-bit values in `m1` and saturates.

```
__m64 _mm_subs_pu8 (__m64 m1 , __m64 m2);
PSUBUSB
```

Remarks

A synonym for **_mm_subs_pu8** is **_m_psubusb**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Packed Arithmetic Intrinsics

_mm_subs_pu16

Microsoft Specific

Subtracts the four unsigned 16-bit values in `m2` from the four unsigned 16-bit values in `m1` and saturates.

```
__m64 _mm_subs_pu16 (__m64 m1 , __m64 m2);
PSUBUSW
```

Remarks

A synonym for **_mm_subs_pu16** is **_m_psubusw**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Packed Arithmetic Intrinsics

_mm_madd_pi16

Microsoft Specific

Multiplies four 16-bit values in `m1` by four 16-bit values in `m2` to produce four 32-bit intermediate results, which are then summed by pairs to produce two 32-bit results.

```
__m64 _mm_madd_pi16 (__m64 m1 , __m64 m2);
PMADDWD
```

Remarks

A synonym for **_mm_madd_pi16** is **_m_pmaddwd**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Packed Arithmetic Intrinsics

_mm_mulhi_pi16

Microsoft Specific

Multiplies four signed 16-bit values in `m1` by four signed 16-bit values in `m2` and produces the high 16 bits of the four results.

```
__m64 _mm_mulhi_pi16 (__m64 m1 , __m64 m2);
PMULHW
```

Remarks

A synonym for **_mm_mulhi_pi16** is **_m_pmulhw**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Packed Arithmetic Intrinsics

_mm_mullo_pi16

Microsoft Specific

Multiplies four 16-bit values in `m1` by four 16-bit values in `m2` and produces the low 16 bits of the four results.

```
__m64 _mm_mullo_pi16 (__m64 m1 , __m64 m2);
PMULLW
```

Remarks

A synonym for **_mm_mullo_pi16** is **_pmullw**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Packed Arithmetic Intrinsics

MMX Technology Shift Intrinsics

Microsoft Specific

The intrinsics listed in the following table are followed by a description of each intrinsic.

MMX intrinsics use the __m64 data type, which is not supported on x64 processors.

The header file mmintrin.h contains the declarations for the MMX intrinsics.

Shift Intrinsics

Intrinsic name	Shift direction	Shift type	Corresponding instruction
_mm_sll_pi16	Left	Logical	PSLLW
_mm_slli_pi16	Left	Logical	PSLLWI
_mm_sll_pi32	Left	Logical	PSLLD
_mm_slli_pi32	Left	Logical	PSLLDI
_mm_sll_si64	Left	Logical	PSLLQ
_mm_slli_si64	Left	Logical	PSLLQI
_mm_sra_pi16	Right	Arithmetic	PSRAW
_mm_srai_pi16	Right	Arithmetic	PSRAWI
_mm_sra_pi32	Right	Arithmetic	PSRAD
_mm_srai_pi32	Right	Arithmetic	PSRADI
_mm_srl_pi16	Right	Logical	PSRLW
_mm_srli_pi16	Right	Logical	PSRLWI
_mm_srl_pi32	Right	Logical	PSRLD
_mm_srli_pi32	Right	Logical	PSRLDI
_mm_srl_si64	Right	Logical	PSRLQ
_mm_srli_si64	Right	Logical	PSRLQI

END Microsoft Specific

See Also

Reference

MMX Technology Intrinsic Groups

_mm_sll_pi16

Microsoft Specific

Shifts four 16-bit values in `m` left the amount specified by `count` while shifting in zeros.

```
__m64 _mm_sll_pi16 (__m64 m , __m64 count);
PSLLW
```

Remarks

A synonym for **_mm_sll_pi16** is **_m_psllw**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Shift Intrinsics

_mm_slli_pi16

Microsoft Specific

Shifts four 16-bit values in `m` left the amount specified by `count` while shifting in zeros.

```
__m64 _mm_slli_pi16 (__m64 m , int count);
PSLLWI
```

Remarks

For the best performance, `count` should be a constant. A synonym for **_mm_slli_pi16** is **_m_psllwi**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Shift Intrinsics

_mm_sll_pi32

Microsoft Specific

Shifts two 32-bit values in `m` left the amount specified by `count` while shifting in zeros.

```
__m64 _mm_sll_pi32 (__m64 m , __m64 count);
PSLLD
```

Remarks

A synonym for **_mm_sll_pi32** is **_m_pslld**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Shift Intrinsics

_mm_slli_pi32

Microsoft Specific

Shifts two 32-bit values in `m` left the amount specified by `count` while shifting in zeros.

```
__m64 _mm_slli_pi32 (__m64 m , int count);
PSLLDI
```

Remarks

For the best performance, `count` should be a constant. A synonym for **_mm_slli_pi32** is **_m_pslldi**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Shift Intrinsics

_mm_sll_si64

Microsoft Specific

Shifts the 64-bit value in `m` left the amount specified by `count` while shifting in zeros.

```
__m64 _mm_sll_si64 (__m64 m , __m64 count);
PSLLQ
```

Remarks

A synonym for **_mm_sll_si64** is **_m_psllq**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Shift Intrinsics

_mm_slli_si64

Microsoft Specific

Shifts the 64-bit value in `m` left the amount specified by `count` while shifting in zeros.

```
__m64 _mm_slli_si64 (__m64 m , int count);
PSLLQI
```

Remarks

For the best performance, `count` should be a constant. A synonym for **_mm_slli_si64** is **_m_psllqi.**

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Shift Intrinsics

_mm_sra_pi16

Microsoft Specific

Shifts four 16-bit values in `m` right the amount specified by `count` while shifting in the sign bit.

```
__m64 _mm_sra_pi16 (__m64 m , __m64 count);
PSRAW
```

Remarks

A synonym for **_mm_sra_pi16** is **_m_psraw**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Shift Intrinsics

_mm_srai_pi16

Microsoft Specific

Shifts four 16-bit values in `m` right the amount specified by `count` while shifting in the sign bit.

```
__m64 _mm_srai_pi16 (__m64 m , int count);
PSRAWI
```

Remarks

For the best performance, `count` should be a constant. A synonym for **_mm_srai_pi16** is **_m_psrawi**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Shift Intrinsics

_mm_sra_pi32

Microsoft Specific

Shifts two 32-bit values in `m` right the amount specified by `count` while shifting in the sign bit.

```
__m64 _mm_sra_pi32 (__m64 m , __m64 count);
PSRAD
```

Remarks

A synonym for **_mm_sra_pi32** is **_m_psrad**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Shift Intrinsics

_mm_srai_pi32

Microsoft Specific

Shifts two 32-bit values in `m` right the amount specified by count while shifting in the sign bit.

```
__m64 _mm_srai_pi32 (__m64 m , int count);
PSRADI
```

Remarks

For the best performance, `count` should be a constant. A synonym for **_mm_srai_pi32** is **_m_psradi**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Shift Intrinsics

_mm_srl_pi16

Microsoft Specific

Shifts four 16-bit values in `m` right the amount specified by `count` while shifting in zeros.

```
__m64 _mm_srl_pi16 (__m64 m , __m64 count);
PSRLW
```

Remarks

A synonym for **_mm_srl_pi16** is **_m_psrlw**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Shift Intrinsics

_mm_srli_pi16

Microsoft Specific

Shifts four 16-bit values in `m` right the amount specified by `count` while shifting in zeros.

```
__m64 _mm_srli_pi16 (__m64 m , int count);
PSRLWI
```

Remarks

For the best performance, `count` should be a constant. A synonym for **_mm_srli_pi16** is **_m_psrlwi**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Shift Intrinsics

_mm_srl_pi32

Microsoft Specific

Shifts two 32-bit values in `m` right the amount specified by `count` while shifting in zeros.

```
__m64 _mm_srl_pi32 (__m64 m , __m64 count);
PSRLD
```

Remarks

A synonym for **_mm_srl_pi32** is **_psrld**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Shift Intrinsics

_mm_srli_pi32

Microsoft Specific

Shifts two 32-bit values in `m` right the amount specified by `count` while shifting in zeros.

```
__m64 _mm_srli_pi32 (__m64 m , int count);
PSRLDI
```

Remarks

For the best performance, `count` should be a constant. A synonym for **_mm_srli_pi32** is **_m_psrldi**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Shift Intrinsics

_mm_srl_si64

Microsoft Specific

Shifts the 64-bit value in `m` right the amount specified by `count` while shifting in zeros.

```
__m64 _mm_srl_si64 (__m64 m , __m64 count);
PSRLQ
```

Remarks

A synonym for **_mm_srl_si64** is **_m_psrlq**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Shift Intrinsics

_mm_srli_si64

Microsoft Specific

Shifts the 64-bit value in `m` right the amount specified by `count` while shifting in zeros.

```
__m64 _mm_srli_si64 (__m64 m , int count);
PSRLQI
```

Remarks

For the best performance, `count` should be a constant. A synonym for **_mm_srli_si64** is **_m_psrlqi**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Shift Intrinsics

MMX Technology Logical Intrinsics

Microsoft Specific

The intrinsics listed in the following table are followed by a description of each intrinsic.

MMX intrinsics use the __m64 data type, which is not supported on x64 processors.

The header file mmintrin.h contains the declarations for the MMX intrinsics.

Logical Intrinsics

Intrinsic name	Operation	Corresponding instruction
_mm_and_si64	Bitwise AND	PAND
_mm_andnot_si64	Logical NOT	PANDN
_mm_or_si64	Bitwise OR	POR
_mm_xor_si64	Bitwise exclusive OR	PXOR

END Microsoft Specific

See Also

Reference

MMX Technology Intrinsic Groups

_mm_and_si64

Microsoft Specific

Performs a bitwise `AND` of the 64-bit value in `m1` with the 64-bit value in `m2`.

```
__m64 _mm_and_si64 (__m64 m1 , __m64 m2);
PAND
```

Remarks

A synonym for **_mm_and_si64** is **_m_pand**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Logical Intrinsics

_mm_andnot_si64

Microsoft Specific

Performs a logical `NOT` on the 64-bit value in `m1` and use the result in a bitwise `AND` with the 64-bit value in `m2`.

```
__m64 _mm_andnot_si64 (__m64 m1 , __m64 m2);
PANDN
```

Remarks

A synonym for **_mm_andnot_si64** is **_m_pandn**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Logical Intrinsics

_mm_or_si64

Microsoft Specific

Performs a bitwise `OR` of the 64-bit value in `m1` with the 64-bit value in `m2`.

```
__m64 _mm_or_si64 (__m64 m1 , __m64 m2);
POR
```

Remarks

A synonym for **_mm_or_si64** is **_m_por**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Logical Intrinsics

_mm_xor_si64

Microsoft Specific

Performs a bitwise `XOR` of the 64-bit value in `m1` with the 64-bit value in `m2`.

```
__m64 _mm_xor_si64 (__m64 m1 , __m64 m2);
PXOR
```

Remarks

A synonym for **_mm_xor_si64** is **_m_pxor**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Logical Intrinsics

MMX Technology Compare Intrinsics

Microsoft Specific

The intrinsics listed in the following table are followed by a description of each intrinsic.

MMX intrinsics use the __m64 data type, which is not supported on x64 processors.

The header file mmintrin.h contains the declarations for the MMX intrinsics.

Compare Intrinsics

Intrinsic name	Comparison	Number of elements	Element bit size	Corresponding instruction
_mm_cmpeq_pi8	Equals	8	8	PCMPEQB
_mm_cmpeq_pi16	Equals	4	16	PCMPEQW
_mm_cmpeq_pi32	Equals	2	32	PCMPEQD
_mm_cmpgt_pi8	Greater than	8	8	PCMPGTB
_mm_cmpgt_pi16	Greater than	4	16	PCMPGTW
_mm_cmpgt_pi32	Greater than	2	32	PCMPGTD

END Microsoft Specific

See Also

Reference

MMX Technology Intrinsic Groups

_mm_cmpeq_pi8

Microsoft Specific

If the respective 8-bit values in `m1` are equal to the respective 8-bit values in `m2`, sets the respective 8-bit resulting values to all ones; otherwise, sets them to all zeros.

```
__m64 _mm_cmpeq_pi8 (__m64 m1 , __m64 m2);
PCMPEQB
```

Remarks

A synonym for **_mm_cmpeq_pi8** is **_m_pcmpeqb**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Compare Intrinsics

_mm_cmpeq_pi16

Microsoft Specific

If the respective 16-bit values in `m1` are equal to the respective 16-bit values in `m2`, sets the respective 16-bit resulting values to all ones; otherwise, sets them to all zeros.

```
__m64 _mm_cmpeq_pi16 (__m64 m1 , __m64 m2);
PCMPEQW
```

Remarks

A synonym for **_mm_cmpeq_pi16** is **_m_pcmpeqw**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Compare Intrinsics

_mm_cmpeq_pi32

Microsoft Specific

If the respective 32-bit values in `m1` are equal to the respective 32-bit values in `m2`, sets the respective 32-bit resulting values to all ones; otherwise, sets them to all zeros.

```
__m64 _mm_cmpeq_pi32 (__m64 m1 , __m64 m2);
PCMPEQD
```

Remarks

A synonym for **_mm_cmpeq_pi32** is **_m_pcmpeqd**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Compare Intrinsics

_mm_cmpgt_pi8

Microsoft Specific

If the respective 8-bit values in `m1` are greater than the respective 8-bit values in `m2`, sets the respective 8-bit resulting values to all ones; otherwise, sets them to all zeros.

```
__m64 _mm_cmpgt_pi8 (__m64 m1 , __m64 m2);
PCMPGTB
```

Remarks

A synonym for **_mm_cmpgt_pi8** is **_m_pcmpgtb**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Compare Intrinsics

_mm_cmpgt_pi16

Microsoft Specific

If the respective 16-bit values in `m1` are greater than the respective 16-bit values in `m2`, sets the respective 16-bit resulting values to all ones; otherwise, sets them to all zeros.

```
__m64 _mm_cmpgt_pi16 (__m64 m1 , __m64 m2);
PCMPGTW
```

Remarks

A synonym for **_mm_cmpgt_pi16** is **_m_pcmpgtw**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also
Reference
MMX Technology Compare Intrinsics

_mm_cmpgt_pi32

Microsoft Specific

If the respective 32-bit values in `m1` are greater than the respective 32-bit values in `m2`, sets the respective 32-bit resulting values to all ones; otherwise, sets them all to zeros.

```
__m64 _mm_cmpgt_pi32 (__m64 m1 , __m64__m64 m2);
PCMPGTD
```

Remarks

A synonym for **_mm_cmpgt_pi32** is **_m_pcmpgtd**.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Compare Intrinsics

MMX Technology Set Intrinsics

Microsoft Specific

The intrinsics listed in the following are followed by a description of each intrinsic. All set intrinsics are made up of composite instructions.

MMX intrinsics use the __m64 data type, which is not supported on x64 processors.

The header file mmintrin.h contains the declarations for the MMX intrinsics.

Set Intrinsics

Intrinsic name	Operation	Number of elements	Element bit size	Signed	Reverse order
_mm_setzero_si64	Sets to zero	1	64	No	No
_mm_set_pi32	Sets integer values	2	32	No	No
_mm_set_pi16	Sets integer values	4	16	No	No
_mm_set_pi8	Sets integer values	8	8	No	No
_mm_set1_pi32	Sets integer values	2	32	Yes	No
_mm_set1_pi16	Sets integer values	4	16	Yes	No
_mm_set1_pi8	Sets integer values	8	8	Yes	No
_mm_setr_pi32	Sets integer values	2	32	No	Yes
_mm_setr_pi16	Sets integer values	4	16	No	Yes
_mm_setr_pi8	Sets integer values	8	8	No	Yes

END Microsoft Specific

See Also

Reference

MMX Technology Intrinsic Groups

_mm_setzero_si64

Microsoft Specific

Sets the 64-bit value to zero.

```
__m64 _mm_setzero_si64 ();
PXOR
```

Return Value

```
r := 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Set Intrinsics

_mm_set_pi32

Microsoft Specific

Sets the two signed 32-bit integer values.

```
__m64 _mm_set_pi32 (int i1, int i0)
(composite)
```

Return Value

```
r0 := i0
r1 := i1
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Set Intrinsics

_mm_set_pi16

Microsoft Specific

Sets the four signed 16-bit integer values.

```
__m64 _mm_set_pi16 (short w3, short w2,    short w1, short w0)
(composite)
```

Return Value

```
r0 := w0
r1 := w1
r2 := w2
r3 := w3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Set Intrinsics

_mm_set_pi8

Microsoft Specific

Sets the eight signed 8-bit integer values.

```
__m64 _mm_set_pi8 (char b7, char b6,    char b5, char b4,   char b3, char b2,   char b1, ch
ar b0)
(composite)
```

Return Value

```
r0 := b0
r1 := b1
...
r7 := b7
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Set Intrinsics

_mm_set1_pi32

Microsoft Specific

Sets the two signed 32-bit integer values to `i`.

```
__m64 _mm_set1_pi32 (int i)
(composite)
```

Return Value

```
r0 := i
r1 := i
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Set Intrinsics

_mm_set1_pi16

Microsoft Specific

Sets the four signed 16-bit integer values to `w`.

```
__m64 _mm_set1_pi16 (short w)
(composite)
```

Return Value

```
r0 := w
r1 := w
r2 := w
r3 := w
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Set Intrinsics

_mm_set1_pi8

Microsoft Specific

Sets the eight signed 8-bit integer values to `b`.

```
__m64 _mm_set1_pi8 (char b)
(composite)
```

Return Value

```
r0 := b
r1 := b
...
r7 := b
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Set Intrinsics

_mm_setr_pi32

Microsoft Specific

Sets the two signed 32-bit integer values in reverse order.

```
__m64 _mm_setr_pi32 (int i0, int i1)
(composite)
```

Return Value

```
r0 := i0
r1 := i1
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Set Intrinsics

_mm_setr_pi16

Microsoft Specific

Sets the four signed 16-bit integer values in reverse order.

```
__m64 _mm_setr_pi16 (short w0, short w1,    short w2, short w3)
(composite)
```

Return Value

```
r0 := w0
r1 := w1
r2 := w2
r3 := w3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Set Intrinsics

_mm_setr_pi8

Microsoft Specific

Sets the eight signed 8-bit integer values in reverse order.

```
__m64 _mm_setr_pi8 (char b0, char b1,    char b2, char b3,   char b4, char b5,   char b6, c
har b7)
(composite)
```

Return Value

```
r0 := b0
r1 := b1
...
r7 := b7
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

MMX Technology Set Intrinsics

Understanding the EMMS Instruction

Microsoft Specific

Using the EMMS instruction is like emptying a container to accommodate new content. For instance, MMX instructions automatically enable a floating-point (FP) tag word in the register to enable use of the **__m64** data type. This resets the FP register set to alias it as the MMX register set. To enable the FP register set again, reset the register state with the EMMS instruction or with the **_mm_empty** intrinsic.

Why You Need EMMS to Reset After an MMX Instruction

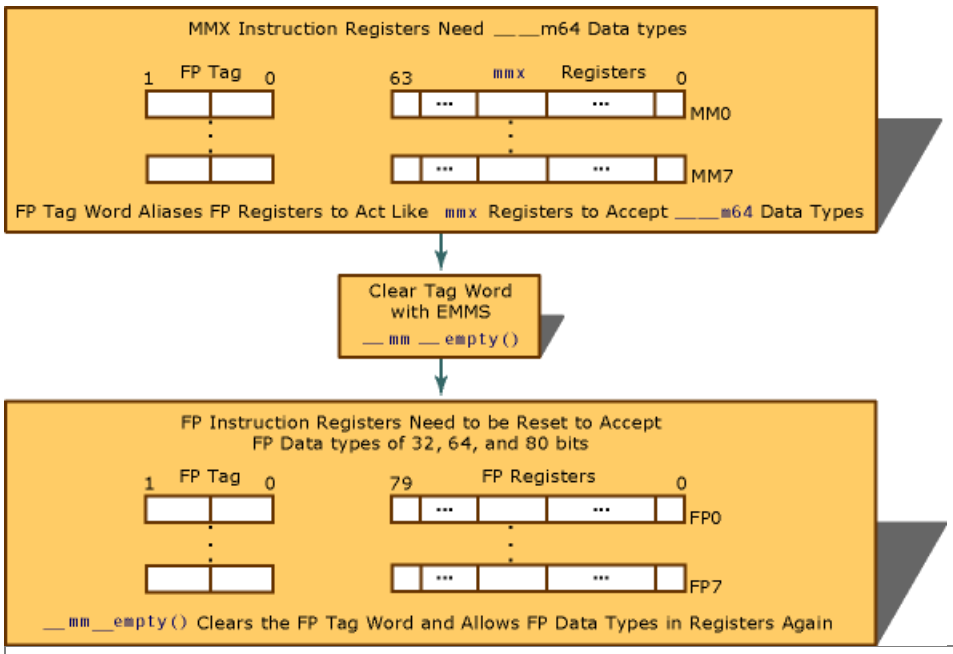

Caution
Failure to reset the tag word for floating-point instructions after using an MMX instruction can result in unexpected execution or poor performance.

For more information on EMMS, see Guidelines for When to Use EMMS.

END Microsoft Specific

See Also

Reference

MMX Technology

MMX Technology General Support Intrinsics

Guidelines for When to Use EMMS

Microsoft Specific

These guidelines help you determine when to use the EMMS instruction:

- If the next instruction is a floating-point instruction, use **_mm_empty** after an MMX instruction (for example, before doing calculations on floats, doubles, or long doubles).
- Don't empty when already empty. If the next instruction uses an MMX register, **_mm_empty** incurs an operation with no benefit (no-op).
- Use different functions for regions that use floating-point instructions and those that use MMX instructions. This eliminates needing an EMMS instruction within the body of a critical loop.
- Use **_mm_empty** during run-time initialization of **__m64** and **FP** data types. This ensures resetting the register between data type transitions. See usage coding in the following example.

Correct EMMS Usage in Initialization Code

Incorrect usage	Correct usage
`__m64 x = _m_paddd(y, z);` `float f = init();`	`__m64 x = _m_paddd(y, z);` `float f = (_mm_empty(), init());`

Further, you must be aware of all situations when your code generates an MMX instruction:

- When using an MMX intrinsic.
- When using the Streaming SIMD Extensions (for those intrinsics that use MMX data).
- When using an MMX instruction through inline assembly.
- When referencing an **__m64** data type variable.

For more documentation on EMMS, see http://developer.intel.com.

MMX intrinsics use the __m64 data type, which is not supported on x64 processors.

END Microsoft Specific
See Also
Reference
MMX Technology
MMX Technology General Support Intrinsics

Streaming SIMD Extensions (SSE)

Microsoft Specific

This section describes the C/C++ language-level features supporting SSE. The following features of the intrinsics are explained:

- Streaming SIMD Extensions Supported by 3DNow!
- Floating-Point Intrinsics Using Streaming SIMD Extensions
- Miscellaneous Intrinsics Using Streaming SIMD Extensions
- Memory and Initialization Using Streaming SIMD Extensions
- Integer Intrinsics Using Streaming SIMD Extensions
- Cache Support Using Streaming SIMD Extensions

In addition, the following macro functions are described:

- Macro Function for Shuffle Using Streaming SIMD Extensions
- Macro Functions to Read and Write the Control Registers
- Macro Function for Matrix Transposition

The header file xmmintrin.h contains the declarations for the SSE intrinsics. The file fvec.h contains operator overloads for some of the SSE intrinsics, which are available for use in C++ programs.

SSE intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE intrinsics that use the __m64 data type are not supported on x64 processors.

END Microsoft Specific

See Also

Reference

MMX, SSE, and SSE2 Intrinsics

Streaming SIMD Extensions Supported by 3DNow!

Microsoft Specific

Currently, the following streaming SIMD extension (SSE) intrinsics are supported by the AMD 3DNow! processor:

- **_mm_maskmove_si64**
- **_mm_avg_pu8**
- **_mm_avg_pu16**
- **_mm_extract_pi16**
- **_mm_insert_pi16**
- **_mm_max_pi16**
- **_mm_max_pu8**
- **_mm_min_pi16**
- **_mm_min_pu8**
- **_mm_movemask_pi8**
- **_mm_mulhi_pu16**
- **_mm_prefetch**
- **_mm_sad_pu8**
- **_mm_shuffle_pi16**
- **_mm_sfence**

The header file xmmintrin.h contains the declarations for the SSE intrinsics.

END Microsoft Specific

See Also
Reference
Streaming SIMD Extensions (SSE)

Floating-Point Intrinsics Using Streaming SIMD Extensions

Microsoft Specific

Each intrinsic entry is presented with its informal pseudo-code and it is followed with a corresponding instruction name in uppercase letters; for example, **ADDSS** is the name of the first instruction listed in this section, which corresponds to the intrinsic for the following:

```
__m128 _mm_add_ss(__m128 a , __m128 b );
ADDSS
```

The variable `r` is generally used for the intrinsic's return value. A number appended to a variable name indicates the element of a packed object. For example, `r0` is the lowest word of `r`. Some intrinsics are "composites" because they require more than one instruction to implement them.

You should be familiar with the hardware features provided by the Streaming SIMD Extensions (SSE) when writing programs with the intrinsics. The following are four important issues to keep in mind:

Certain intrinsics, such as **_mm_loadr_ps** and **_mm_cmpgt_ss**, are not directly supported by the instruction set. While these intrinsics are convenient programming aids, be mindful that they might consist of more than one machine-language instruction.

- Floating-point data loaded stored as **__m128** objects must be generally 16-byte aligned.
- Some intrinsics require that their argument be immediates, that is, constant integers (literals), because of the nature of the instruction.
- The result of arithmetic operations acting on two not a number (NAN) arguments is undefined. Therefore, floating-point operations using NAN arguments will not match the expected behavior of the corresponding assembly instructions.

The following floating-point operations are discussed:

- Arithmetic Operations
- Logical Operations
- Comparison Intrinsics
- Conversion Operations

END Microsoft Specific
See Also
Reference
Streaming SIMD Extensions (SSE)

Arithmetic Operations (Floating-Point SSE2 Intrinsics)

Microsoft Specific

The intrinsics listed in the following table are followed by a description of each intrinsic.

SSE intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE intrinsics that use the __m64 data type are not supported on x64 processors.

The header file xmmintrin.h contains the declarations for the SSE intrinsics.

Packed Arithmetic Intrinsics

Intrinsic	Instruction	Operation	R0	R1	R2	R3
_mm_add_ss	ADDSS	Adds	a0 [op] b0	a1	a2	a3
_mm_add_ps	ADDPS	Adds	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
_mm_sub_ss	SUBSS	Subtracts	a0 [op] b0	a1	a2	a3
_mm_sub_ps	SUBPS	Subtracts	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
_mm_mul_ss	MULSS	Multiplies	a0 [op] b0	a1	a2	a3
_mm_mul_ps	MULPS	Multiplies	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
_mm_div_ss	DIVSS	Divides	a0 [op] b0	a1	a2	a3
_mm_div_ps	DIVPS	Divides	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
_mm_sqrt_ss	SQRTSS	Computes squared root	[op] a0	a1	a2	a3
_mm_sqrt_ps	SQRTPS	Computes squared root	[op] a0	[op] b1	[op] b2	[op] b3
_mm_rcp_ss	RCPSS	Computes reciprocal	[op] a0	a1	a2	a3
_mm_rcp_ps	RCPPS	Computes reciprocal	[op] a0	[op] b1	[op] b2	[op] b3

_mm_rsqrt_ss	RSQRTSS	Computes reciprocal square root	`[op] a0`	`a1`	`a2`	`a3`
_mm_rsqrt_ps	RSQRTPS	Computes reciprocal squared root	`[op] a0`	`[op] b1`	`[op] b2`	`[op] b3`
_mm_min_ss	MINSS	Computes minimum	`[op]( a0,b0)`	`a1`	`a2`	`a3`
_mm_min_ps	MINPS	Computes minimum	`[op]( a0,b0)`	`[op] (a1, b1)`	`[op] (a2, b2)`	`[op] (a3, b3)`
_mm_max_ss	MAXSS	Computes maximum	`[op]( a0,b0)`	`a1`	`a2`	`a3`
_mm_max_ps	MAXPS	Computes maximum	`[op]( a0,b0)`	`[op] (a1, b1)`	`[op] (a2, b2)`	`[op] (a3, b3)`

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

END Microsoft Specific

See Also

Reference

Floating-Point Intrinsics Using Streaming SIMD Extensions

_mm_add_ss

Microsoft Specific

Adds the lower single-precision, floating-point values of `a` and `b`; the upper three single-precision, floating-point values are passed through from `a`.

```
__m128 _mm_add_ss(__m128 a , __m128 b )
ADDSS
```

Return Value

```
r0 := a0 + b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating-Point SSE2 Intrinsics)

_mm_add_ps

Microsoft Specific

Adds the four single-precision, floating-point values of `a` and `b`.

```
__m128 _mm_add_ps(__m128 a , __m128 b );
ADDPS
```

Return Value

```
r0 := a0 + b0
r1 := a1 + b1
r2 := a2 + b2
r3 := a3 + b3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating-Point SSE2 Intrinsics)

_mm_sub_ss

Microsoft Specific

Subtracts the lower single-precision, floating-point values of `a` and `b`.

```
__m128 _mm_sub_ss(__m128 a , __m128 b );
SUBSS
```

Return Value

The upper three single-precision, floating-point values are passed through from `a`.

```
r0 := a0 - b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating-Point SSE2 Intrinsics)

_mm_sub_ps

Microsoft Specific

Subtracts the four single-precision, floating-point values of `a` and `b`.

```
__m128 _mm_sub_ps(__m128 a , __m128 b );
SUBPS
```

Return Value

```
r0 := a0 - b0
r1 := a1 - b1
r2 := a2 - b2
r3 := a3 - b3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating-Point SSE2 Intrinsics)

_mm_mul_ss

Microsoft Specific

Multiplies the lower single-precision, floating-point values of `a` and `b`; the upper three single-precision, floating-point values are passed through from `a`.

```
__m128 _mm_mul_ss(__m128 a , __m128 b );
MULSS
```

Return Value

```
r0 := a0 * b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating-Point SSE2 Intrinsics)

_mm_mul_ps

Microsoft Specific

Multiplies the four single-precision, floating-point values of `a` and `b`.

```
__m128 _mm_mul_ps(__m128 a , __m128 b );
MULPS
```

Return Value

```
r0 := a0 * b0
r1 := a1 * b1
r2 := a2 * b2
r3 := a3 * b3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating-Point SSE2 Intrinsics)

_mm_div_ss

Microsoft Specific

Divides the lower single-precision, floating-point values of `a` and `b`; the upper three single-precision, floating-point values are passed through from `a`.

```
__m128 _mm_div_ss(__m128 a , __m128 b );
DIVSS
```

Return Value

```
r0 := a0 / b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating-Point SSE2 Intrinsics)

_mm_div_ps

Microsoft Specific

Divides the four single-precision, floating-point values of `a` and `b`.

```
__m128 _mm_div_ps(__m128 a, __m128 b );
DIVPS
```

Return Value

```
r0 := a0 / b0
r1 := a1 / b1
r2 := a2 / b2
r3 := a3 / b3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating-Point SSE2 Intrinsics)

_mm_sqrt_ss

Microsoft Specific

Computes the square root of the lower single-precision, floating-point value of `a`; the upper three single-precision, floating-point values are passed through.

```
__m128 _mm_sqrt_ss(__m128 a );
SQRTSS
```

Return Value

```
r0 := sqrt(a0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating-Point SSE2 Intrinsics)

_mm_sqrt_ps

Microsoft Specific

Computes the square roots of the four single-precision, floating-point values of a.

```
__m128 _mm_sqrt_ps(__m128 a );
SQRTPS
```

Return Value

```
r0 := sqrt(a0)
r1 := sqrt(a1)
r2 := sqrt(a2)
r3 := sqrt(a3)
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating-Point SSE2 Intrinsics)

_mm_rcp_ss

Microsoft Specific

Computes the approximation of the reciprocal of the lower single-precision, floating-point value of `a`; the upper three single-precision, floating-point values are passed through.

```
__m128 _mm_rcp_ss(__m128 a );
RCPSS
```

Return Value

```
r0 := recip(a0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating-Point SSE2 Intrinsics)

_mm_rcp_ps

Microsoft Specific

Computes the approximations of reciprocals of the four single-precision, floating-point values of `a`.

```
__m128 _mm_rcp_ps(__m128 a );
RCPPS
```

Return Value

```
r0 := recip(a0)
r1 := recip(a1)
r2 := recip(a2)
r3 := recip(a3)
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating-Point SSE2 Intrinsics)

_mm_rsqrt_ss

Microsoft Specific

Computes the approximation of the reciprocal of the square root of the lower single-precision, floating-point value of `a`; the upper three single-precision, floating-point values are passed through.

```
__m128 _mm_rsqrt_ss(__m128 a );
RSQRTSS
```

Return Value

```
r0 := recip(sqrt(a0))
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating-Point SSE2 Intrinsics)

_mm_rsqrt_ps

Microsoft Specific

Computes the approximations of the reciprocals of the square roots of the four single-precision, floating-point values of `a`.

```
__mm128 _mm_rsqrt_ps(__m128 a );
RSQRTPS
```

Return Value

```
r0 := recip(sqrt(a0))
r1 := recip(sqrt(a1))
r2 := recip(sqrt(a2))
r3 := recip(sqrt(a3))
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating-Point SSE2 Intrinsics)

_mm_min_ss

Microsoft Specific

Computes the minimum of the lower single-precision, floating-point values of `a` and `b`; the upper three single-precision, floating-point values are passed through from `a`.

```
__m128 _mm_min_ss(__m128 a , __m128 b );
MINSS
```

Return Value

```
r0 := min(a0, b0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating-Point SSE2 Intrinsics)

_mm_min_ps

Microsoft Specific

Computes the minima of the four single-precision, floating-point values of `a` and `b`.

```
__m128 _mm_min_ps(__m128 a , __m128 b );
MINPS
```

Return Value

```
r0 := min(a0, b0)
r1 := min(a1, b1)
r2 := min(a2, b2)
r3 := min(a3, b3)
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating-Point SSE2 Intrinsics)

_mm_max_ss

Microsoft Specific

Computes the maximum of the lower single-precision, floating-point values of `a` and `b`; the upper three single-precision, floating-point values are passed through from `a`.

```
__m128 _mm_max_ss(__m128 a , __m128 b );
MAXSS
```

Return Value

```
r0 := max(a0, b0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating-Point SSE2 Intrinsics)

_mm_max_ps

Microsoft Specific

Computes the maximums of the four single-precision, floating-point values of `a` and `b`.

```
__m128 _mm_max_ps(__m128 a , __m128 b );
MAXPS
```

Return Value

```
r0 := max(a0, b0)
r1 := max(a1, b1)
r2 := max(a2, b2)
r3 := max(a3, b3)
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating-Point SSE2 Intrinsics)

Logical Operations (SSE)

Microsoft Specific

The intrinsics listed in the following table are followed by a description of each intrinsic.

SSE intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE intrinsics that use the __m64 data type are not supported on x64 processors.

The header file xmmintrin.h contains the declarations for the SSE intrinsics.

Logical Intrinsics

Intrinsic name	Operation	Corresponding instruction
_mm_and_ps	Bitwise AND	ANDPS
_mm_andnot_ps	Logical NOT	ANDNPS
_mm_or_ps	Bitwise OR	ORPS
_mm_xor_ps	Bitwise Exclusive OR	XORPS

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

END Microsoft Specific

See Also

Reference

Floating-Point Intrinsics Using Streaming SIMD Extensions

_mm_and_ps

Microsoft Specific

Computes the bitwise AND of the four single-precision, floating-point values of a and b.

```
__m128 _mm_and_ps(__m128 a , __m128 b );
ANDPS
```

Return Value

```
r0 := a0 & b0
r1 := a1 & b1
r2 := a2 & b2
r3 := a3 & b3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Logical Operations (SSE)

_mm_andnot_ps

Microsoft Specific

Computes the bitwise `AND-NOT` of the four single-precision, floating-point values of `a` and `b`.

```
__m128 _mm_andnot_ps(__m128 a , __m128 b );
ANDNPS
```

Return Value

```
r0 := ~a0 & b0
r1 := ~a1 & b1
r2 := ~a2 & b2
r3 := ~a3 & b3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Logical Operations (SSE)

_mm_or_ps

Microsoft Specific

Computes the bitwise OR of the four single-precision, floating-point values of a and b.

```
__m128 _mm_or_ps(__m128 a , __m128 b );
ORPS
```

Return Value

```
r0 := a0 | b0
r1 := a1 | b1
r2 := a2 | b2
r3 := a3 | b3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Logical Operations (SSE)

_mm_xor_ps

Microsoft Specific

Computes bitwise `EXOR` (exclusive-or) of the four single-precision, floating-point values of `a` and `b`.

```
__m128 _mm_xor_ps(__m128 a , __m128 b );
XORPS
```

Return Value

```
r0 := a0 ^ b0
r1 := a1 ^ b1
r2 := a2 ^ b2
r3 := a3 ^ b3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Logical Operations (SSE)

Comparison Intrinsics (SSE)

Microsoft Specific

Each comparison intrinsic performs a comparison of `a` and `b`. For the packed form, the four single-precision, floating-point values of `a` and `b` are compared, and a 128-bit mask is returned. For the scalar form, the lower single-precision, floating-point values of `a` and `b` are compared, and a 32-bit mask is returned; the upper three single-precision, floating-point values are passed through from `a`. The mask is set to `0xffffffff` for each element where the comparison is true and `0x0` where the comparison is false.

The superscript '`r`' on the instruction indicates that the operands are reversed in the instruction implementation. The compare intrinsics listed in the following table are followed by a description of each intrinsic.

SSE intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE intrinsics that use the __m64 data type are not supported on x64 processors.

The header file xmmintrin.h contains the declarations for the SSE intrinsics.

Compare Intrinsics

Intrinsic name	Comparison	Corresponding instruction
_mm_cmpeq_ss	Equal	CMPEQSS
_mm_cmpeq_ps	Equal	CMPEQPS
_mm_cmplt_ss	Less than	CMPLTSS
_mm_cmplt_ps	Less than	CMPLTPS
_mm_cmple_ss	Less than or equal	CMPLESS
_mm_cmple_ps	Less than or equal	CMPLEPS
_mm_cmpgt_ss	Greater than	CMPLTSS
_mm_cmpgt_ps	Greater than	CMPLTPS
_mm_cmpge_ss	Greater than or equal	CMPLESS
_mm_cmpge_ps	Greater than or equal	CMPLEPS
_mm_cmpneq_ss	Not equal	CMPNEQSS
_mm_cmpneq_ps	Not equal	CMPNEQPS
_mm_cmpnlt_ss	Not less than	CMPNLTSS
_mm_cmpnlt_ps	Not less than	CMPNLTPS
_mm_cmpnle_ss	Not less than or equal	CMPNLESS
_mm_cmpnle_ps	Not less than or equal	CMPNLEPS
_mm_cmpngt_ss	Not greater than	CMPNLTSS
_mm_cmpngt_ps	Not greater than	CMPNLTPS

_mm_cmpnge_ss	Not greater than or equal	CMPNLESS
_mm_cmpnge_ps	Not greater than or equal	CMPNLEPS
_mm_cmpord_ss	Ordered	CMPORDSS
_mm_cmpord_ps	Ordered	CMPORDPS
_mm_cmpunord_ss	Unordered	CMPUNORDSS
_mm_cmpunord_ps	Unordered	CMPUNORDPS
_mm_comieq_ss	Equal	COMISS
_mm_comilt_ss	Less than	COMISS
_mm_comile_ss	Less than or equal	COMISS
_mm_comigt_ss	Greater than	COMISS
_mm_comige_ss	Greater than or equal	COMISS
_mm_comineq_ss	Not equal	COMISS
_mm_ucomieq_ss	Equal	UCOMISS
_mm_ucomilt_ss	Less than	UCOMISS
_mm_ucomile_ss	Less than or equal	UCOMISS
_mm_ucomigt_ss	Greater than	UCOMISS
_mm_ucomige_ss	Greater than or equal	UCOMISS
_mm_ucomineq_ss	Not equal	UCOMISS

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

END Microsoft Specific

See Also

Reference

Floating-Point Intrinsics Using Streaming SIMD Extensions

_mm_cmpeq_ss

Microsoft Specific

Compares for equality.

```
__m128 _mm_cmpeq_ss(__m128 a , __m128 b );
CMPEQSS
```

Return Value

```
r0 := (a0 == b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmpeq_ps

Microsoft Specific

Compares for equality.

```
__m128 _mm_cmpeq_ps(__m128 a , __m128 b );
CMPEQPS
```

Return Value

```
r0 := (a0 == b0) ? 0xffffffff : 0x0
r1 := (a1 == b1) ? 0xffffffff : 0x0
r2 := (a2 == b2) ? 0xffffffff : 0x0
r3 := (a3 == b3) ? 0xffffffff : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmplt_ss

Microsoft Specific

Compares for less than.

```
__m128 _mm_cmplt_ss(__m128 a , __m128 b );
CMPLTSS
```

Return Value

```
r0 := (a0 < b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmplt_ps

Microsoft Specific

Compares for less than.

```
__m128 _mm_cmplt_ps(__m128 a, __m128 b );
CMPLTPS
```

Return Value

```
r0 := (a0 < b0) ? 0xffffffff : 0x0
r1 := (a1 < b1) ? 0xffffffff : 0x0
r2 := (a2 < b2) ? 0xffffffff : 0x0
r3 := (a3 < b3) ? 0xffffffff : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmple_ss

Microsoft Specific

Compares for less than or equal.

```
__m128 _mm_cmple_ss(__m128 a , __m128 b );
CMPLESS
```

Return Value

```
r0 := (a0 <= b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmple_ps

Microsoft Specific

Compares for less than or equal.

```
__m128 _mm_cmple_ps(__m128 a , __m128 b );
CMPLEPS
```

Return Value

```
r0 := (a0 <= b0) ? 0xffffffff : 0x0
r1 := (a1 <= b1) ? 0xffffffff : 0x0
r2 := (a2 <= b2) ? 0xffffffff : 0x0
r3 := (a3 <= b3) ? 0xffffffff : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmpgt_ss

Microsoft Specific

Compares for greater than.

```
__m128 _mm_cmpgt_ss(__m128 a , __m128 b );
CMPLTSSr
```

Return Value

```
r0 := (a0 > b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmpgt_ps

Microsoft Specific

Compares for greater than.

```
__m128 _mm_cmpgt_ps(__m128 a, __m128 b );
CMPLTPSr
```

Return Value

```
r0 := (a0 > b0) ? 0xffffffff : 0x0
r1 := (a1 > b1) ? 0xffffffff : 0x0
r2 := (a2 > b2) ? 0xffffffff : 0x0
r3 := (a3 > b3) ? 0xffffffff : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmpge_ss

Microsoft Specific

Compares for greater than or equal.

```
__m128 _mm_cmpge_ss(__m128 a , __m128 b );
CMPLESSr
```

Return Value

```
r0 := (a0 >= b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmpge_ps

Microsoft Specific

Compares for greater than or equal.

```
__m128 _mm_cmpge_ps(__m128 a, __m128 b );
CMPLEPSr
```

Return Value

```
r0 := (a0 >= b0) ? 0xffffffff : 0x0
r1 := (a1 >= b1) ? 0xffffffff : 0x0
r2 := (a2 >= b2) ? 0xffffffff : 0x0
r3 := (a3 >= b3) ? 0xffffffff : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmpneq_ss

Microsoft Specific

Compares for inequality.

```
__m128 _mm_cmpneq_ss(__m128 a , __m128 b );
CMPNEQSS
```

Return Value

```
r0 := (a0 != b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmpneq_ps

Microsoft Specific

Compares for inequality.

```
__m128 _mm_cmpneq_ps(__m128 a , __m128 b );
CMPNEQPS
```

Return Value

```
r0 := (a0 != b0) ? 0xffffffff : 0x0
r1 := (a1 != b1) ? 0xffffffff : 0x0
r2 := (a2 != b2) ? 0xffffffff : 0x0
r3 := (a3 != b3) ? 0xffffffff : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmpnlt_ss

Microsoft Specific

Compares for not less than.

```
__m128 _mm_cmpnlt_ss(__m128 a , __m128 b );
CMPNLTSS
```

Return Value

```
r0 := !(a0 < b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmpnlt_ps

Microsoft Specific

Compares for not less than.

```
__m128 _mm_cmpnlt_ps(__m128 a, __m128 b );
CMPNLTPS
```

Return Value

```
r0 := !(a0 < b0) ? 0xffffffff : 0x0
r1 := !(a1 < b1) ? 0xffffffff : 0x0
r2 := !(a2 < b2) ? 0xffffffff : 0x0
r3 := !(a3 < b3) ? 0xffffffff : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmpnle_ss

Microsoft Specific

Compares for not less than or equal.

```
__m128 _mm_cmpnle_ss(__m128 a , __m128 b );
CMPNLESS
```

Return Value

```
r0 := !(a0 <= b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmpnle_ps

Microsoft Specific

Compares for not less than or equal.

```
__m128 _mm_cmpnle_ps(__m128 a , __m128 b );
CMPNLEPS
```

Return Value

```
r0 := !(a0 <= b0) ? 0xffffffff : 0x0
r1 := !(a1 <= b1) ? 0xffffffff : 0x0
r2 := !(a2 <= b2) ? 0xffffffff : 0x0
r3 := !(a3 <= b3) ? 0xffffffff : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmpngt_ss

Microsoft Specific

Compares for not greater than.

```
__m128 _mm_cmpngt_ss(__m128 a , __m128 b );
CMPNLTSSr
```

Return Value

```
r0 := !(a0 > b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmpngt_ps

Microsoft Specific

Compares for not greater than.

```
__m128 _mm_cmpngt_ps(__m128 a , __m128 b );
CMPNLTPSr
```

Return Value

```
r0 := !(a0 > b0) ? 0xffffffff : 0x0
r1 := !(a1 > b1) ? 0xffffffff : 0x0
r2 := !(a2 > b2) ? 0xffffffff : 0x0
r3 := !(a3 > b3) ? 0xffffffff : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmpnge_ss

Microsoft Specific

Compares for not greater than or equal.

```
__m128 _mm_cmpnge_ss(__m128 a , __m128 b );
CMPNLESSr
```

Return Value

```
r0 := !(a0 >= b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmpnge_ps

Microsoft Specific

Compares for not greater than or equal.

```
__m128 _mm_cmpnge_ps(__m128 a , __m128 b );
MPNLEPSr
```

Return Value

```
r0 := !(a0 >= b0) ? 0xffffffff : 0x0
r1 := !(a1 >= b1) ? 0xffffffff : 0x0
r2 := !(a2 >= b2) ? 0xffffffff : 0x0
r3 := !(a3 >= b3) ? 0xffffffff : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmpord_ss

Microsoft Specific

Compares for ordered.

```
__m128 _mm_cmpord_ss(__m128 a , __m128 b );
CMPORDSS
```

Return Value

```
r0 := (a0 ord? b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmpord_ps

Microsoft Specific

Compares for ordered.

```
__m128 _mm_cmpord_ps(__m128 a , __m128 b );
CMPORDPS
```

Return Value

```
r0 := (a0 ord? b0) ? 0xffffffff : 0x0
r1 := (a1 ord? b1) ? 0xffffffff : 0x0
r2 := (a2 ord? b2) ? 0xffffffff : 0x0
r3 := (a3 ord? b3) ? 0xffffffff : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmpunord_ss

Microsoft Specific

Compares for unordered.

```
__m128 _mm_cmpunord_ss(__m128 a , __m128 b );
CMPUNORDSS
```

Return Value

```
r0 := (a0 unord? b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_cmpunord_ps

Microsoft Specific

Compares for unordered.

```
__m128 _mm_cmpunord_ps(__m128 a , __m128 b );
CMPUNORDPS
```

Return Value

```
r0 := (a0 unord? b0) ? 0xffffffff : 0x0
r1 := (a1 unord? b1) ? 0xffffffff : 0x0
r2 := (a2 unord? b2) ? 0xffffffff : 0x0
r3 := (a3 unord? b3) ? 0xffffffff : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_comieq_ss

Microsoft Specific

Compares the lower single-precision, floating-point value of `a` and `b` for `a` equal to `b`.

```
int _mm_comieq_ss(__m128 a,__m128 b );
COMISS
```

Return Value

If `a` and `b` are equal, `1` is returned. Otherwise, `0` is returned. If `a` or `b` is a NaN, `1` is returned.

```
r := (a0 == b0) ? 0x1 : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also
Reference
Comparison Intrinsics (SSE)

_mm_comilt_ss

Microsoft Specific

Compares the lower single-precision, floating-point value of `a` and `b` for `a` less than `b`.

```
int _mm_comilt_ss(__m128 a,__m128 b );
COMISS
```

Return Value

If `a` is less than `b`, `1` is returned. Otherwise, `0` is returned. If `a` or `b` is a NaN, `1` is returned.

```
r := (a0 < b0) ? 0x1 : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_comile_ss

Microsoft Specific

Compares the lower single-precision, floating-point value of `a` and `b` for `a` less than or equal to `b`.

```
int _mm_comile_ss(__m128 a,__m128 b );
COMISS
```

Return Value

If `a` is less than or equal to `b`, `1` is returned. Otherwise, `0` is returned. If `a` or `b` is a NaN, `1` is returned.

```
r := (a0 <= b0) ? 0x1 : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also
Reference
Comparison Intrinsics (SSE)

_mm_comigt_ss

Microsoft Specific

Compares the lower single-precision, floating-point value of `a` and `b` for `a` greater than `b`.

```
int _mm_comigt_ss(__m128 a,__m128 b );
COMISS
```

Return Value

If `a` is greater than `b` are equal, `1` is returned. Otherwise, `0` is returned. If `a` or `b` is a NaN, `1` is returned.

```
r := (a0 > b0) ? 0x1 : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also
Reference
Comparison Intrinsics (SSE)

_mm_comige_ss

Microsoft Specific

Compares the lower single-precision, floating-point value of `a` and `b` for `a` greater than or equal to `b`.

```
int _mm_comige_ss(__m128 a,__m128 b );
COMISS
```

Return Value

If `a` is greater than or equal to `b`, `1` is returned. Otherwise, `0` is returned. If `a` or `b` is a NaN, `1` is returned.

```
r := (a0 >= b0) ? 0x1 : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also
Reference
Comparison Intrinsics (SSE)

_mm_comineq_ss

Microsoft Specific

Compares the lower single-precision, floating-point value of `a` and `b` for `a` not equal to `b`.

```
int _mm_comineq_ss(__m128 a,__m128 b );
COMISS
```

Return Value

If `a` and `b` are not equal, `1` is returned. Otherwise, `0` is returned. If `a` or `b` is a NaN, `1` is returned.

```
r := (a0 != b0) ? 0x1 : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_ucomieq_ss

Microsoft Specific

Compares the lower single-precision, floating-point value of `a` and `b` for `a` equal to `b`.

```
int _mm_ucomieq_ss(__m128 a,__m128 b );
UCOMISS
```

Return Value

If `a` and `b` are equal, `1` is returned. Otherwise, `0` is returned. If `a` or `b` is a NaN, `1` is returned.

```
r := (a0 == b0) ? 0x1 : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also
Reference
Comparison Intrinsics (SSE)

_mm_ucomilt_ss

Microsoft Specific

Compares the lower single-precision, floating-point value of `a` and `b` for `a` less than `b`.

```
int _mm_ucomilt_ss(__m128 a,__m128 b);
UCOMISS
```

Return Value

If `a` is less than `b`, `1` is returned. Otherwise, `0` is returned. If `a` or `b` is a NaN, `1` is returned.

```
r := (a0 < b0) ? 0x1 : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also
Reference
Comparison Intrinsics (SSE)

_mm_ucomile_ss

Microsoft Specific

Compares the lower single-precision, floating-point value of `a` and `b` for `a` less than or equal to `b`.

```
int _mm_ucomile_ss(__m128 a,__m128 b );
UCOMISS
```

Return Value

If `a` is less than or equal to `b`, `1` is returned. Otherwise, `0` is returned. If `a` or `b` is a NaN, `1` is returned.

```
r := (a0 <= b0) ? 0x1 : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_ucomigt_ss

Microsoft Specific

Compares the lower single-precision, floating-point value of `a` and `b` for `a` greater than `b`.

```
int _mm_ucomigt_ss(__m128 a,__m128 b );
UCOMISS
```

Return Value

If `a` is greater than `b` are equal, `1` is returned. Otherwise, `0` is returned. If `a` or `b` is a NaN, `1` is returned. If `a` or `b` is a NaN, `1` is returned.

```
r := (a0 > b0) ? 0x1 : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also
Reference
Comparison Intrinsics (SSE)

_mm_ucomige_ss

Microsoft Specific

Compares the lower single-precision, floating-point value of `a` and `b` for `a` greater than or equal to `b`.

```
int _mm_ucomige_ss(__m128 a,__m128 b );
UCOMISS
```

Return Value

If `a` is greater than or equal to `b`, `1` is returned. Otherwise, `0` is returned.

```
r := (a0 >= b0) ? 0x1 : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

_mm_ucomineq_ss

Microsoft Specific

Compares the lower single-precision, floating-point value of `a` and `b` for `a` not equal to `b`.

```
int _mm_ucomineq_ss(__m128 a,__m128 b );
UCOMISS
```

Return Value

If `a` and `b` are not equal, `1` is returned. Otherwise, `0` is returned. If `a` or `b` is a NaN, `1` is returned.

```
r := (a0 != b0) ? 0x1 : 0x0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE)

Conversion Operations (SSE)

Microsoft Specific

The conversions operations are listed in the Conversion Operations table followed by a description of each intrinsic with the most recent mnemonic naming convention. The alternate name is provided if you have used these intrinsics before.

SSE intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE intrinsics that use the __m64 data type are not supported on x64 processors.

The header file xmmintrin.h contains the declarations for the SSE intrinsics.

Conversion Operations

Intrinsic name	Corresponding instruction
_mm_cvtss_si32	CVTSS2SI
_mm_cvtps_pi32	CVTPS2PI
_mm_cvttss_si32	CVTTSS2SI
_mm_cvttps_pi32	CVTTPS2PI
_mm_cvtsi32_ss	CVTSI2SS
_mm_cvtpi32_ps	CVTTPS2PI
_mm_cvtpi16_ps	Composite
_mm_cvtpu16_ps	Composite
_mm_cvtpi8_ps	Composite
_mm_cvtpu8_ps	Composite
_mm_cvtpi32x2_ps	Composite
_mm_cvtps_pi16	Composite
_mm_cvtps_pi8	Composite

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

END Microsoft Specific

See Also

Reference

Floating-Point Intrinsics Using Streaming SIMD Extensions

_mm_cvtss_si32

Microsoft Specific

Converts the lower single-precision, floating-point value of `a` to a 32-bit integer according to the current rounding mode.

```
int _mm_cvtss_si32(__m128 a );
CVTSS2SI
```

Return Value

```
r := (int)a0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE)

_mm_cvtps_pi32

Microsoft Specific

Converts the two lower single-precision, floating-point values of `a` to two 32-bit integers according to the current rounding mode, returning the integers in packed form.

```
__m64 _mm_cvtps_pi32(__m128 a );
CVTPS2PI
```

Return Value

```
r0 := (int)a0
r1 := (int)a1
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE)

_mm_cvttss_si32

Microsoft Specific

Converts the lower single-precision, floating-point value of `a` to a 32-bit integer with truncation.

```
int _mm_cvttss_si32(__m128 a );
CVTTSS2SI
```

Return Value

```
r := (int)a0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE)

_mm_cvttps_pi32

Microsoft Specific

Converts the two lower single-precision, floating-point values of `a` to two 32-bit integer with truncation, returning the integers in packed form.

```
__m64 _mm_cvttps_pi32(__m128 a );
CVTTPS2PI
```

Return Value

```
r0 := (int)a0
r1 := (int)a1
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE)

_mm_cvtsi32_ss

Microsoft Specific

Converts the 32-bit integer value `b` to an single-precision, floating-point value; the upper three single-precision, floating-point values are passed through from `a`.

```
__m128 _mm_cvtsi32_ss(__m128 a , int b );
CVTSI2SS
```

Return Value

```
r0 := (float)b
r1 := a1 ; r2 := a2 ; r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE)

_mm_cvtpi32_ps

Microsoft Specific

Converts the two 32-bit integer values in packed form in `b` to two single-precision, floating-point values; the upper two single-precision, floating-point values are passed through from `a`.

```
__m128 _mm_cvtpi32_ps(__m128 a , __m64 b );
CVTPI2PS
```

Return Value

```
r0 := (float)b0
r1 := (float)b1
r2 := a2
r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE)

_mm_cvtpi16_ps

Microsoft Specific

Converts the four 16-bit signed integer values in `a` to four single-precision, floating-point values.

```
__m128 _mm_cvtpi16_ps(__m64 a );
(composite)
```

Return Value

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE)

_mm_cvtpu16_ps

Microsoft Specific

Converts the four 16-bit unsigned integer values in `a` to four single-precision, floating-point values.

```
__m128 _mm_cvtpu16_ps(__m64 a );
(composite)
```

Return Value

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE)

_mm_cvtpi8_ps

Microsoft Specific

Converts the lower four 8-bit signed integer values in `a` to four single-precision, floating-point values.

```
__m128 _mm_cvtpi8_ps(__m64 a );
(composite)
```

Return Value

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE)

_mm_cvtpu8_ps

Microsoft Specific

Converts the lower four 8-bit unsigned integer values in `a` to four single-precision, floating-point values.

```
__m128 _mm_cvtpu8_ps(__m64 a );
(composite)
```

Return Value

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE)

_mm_cvtpi32x2_ps

Microsoft Specific

Converts the two 32-bit signed integer values in `a` and the two 32-bit signed integer values in `b` to four single-precision, floating-point values.

```
__m128 _mm_cvtpi32x2_ps(__m64 a, __m64 b );
(composite)
```

Return Value

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)b0
r3 := (float)b1
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE)

_mm_cvtps_pi16

Microsoft Specific

Converts the four single-precision, floating-point values in `a` to four signed 16-bit integer values.

```
__m64 _mm_cvtps_pi16( __m128 a );
(composite)
```

Return Value

```
r0 := (short)a0
r1 := (short)a1
r2 := (short)a2
r3 := (short)a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE)

_mm_cvtps_pi8

Microsoft Specific

Converts the four single-precision, floating-point values in `a` to the lower four signed 8-bit integer values of the result.

```
__m64 _mm_cvtps_pi8( __m128 a );
(composite)
```

Return Value

```
r0 := (char)a0
r1 := (char)a1
r2 := (char)a2
r3 := (char)a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE)

Miscellaneous Intrinsics Using Streaming SIMD Extensions

Microsoft Specific

The miscellaneous intrinsics listed in the following table are followed by a description of each intrinsic.

SSE intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE intrinsics that use the __m64 data type are not supported on x64 processors.

The header file xmmintrin.h contains the declarations for the SSE intrinsics.

Miscellaneous Intrinsics

Intrinsic name	Operation	Corresponding instruction
_mm_shuffle_ps	Shuffles	SHUFPS
_mm_shuffle_pi16	Shuffles	PSHUFW
_mm_unpackhi_ps	Unpacks high	UNPCKHPS
_mm_unpacklo_ps	Unpacks low	UNPCKLPS
_mm_loadh_pi	Loads high	MOVHPS reg, mem
_mm_storeh_pi	Stores high	MOVHPS mem, reg
_mm_movehl_ps	Moves high to low	MOVHLPS
_mm_movelh_ps	Moves low to high	MOVLHPS
_mm_loadl_pi	Loads low	MOVLPS reg, mem
_mm_storel_pi	Stores low	MOVLPS mem, reg
_mm_movemask_ps	Creates four-bit mask	MOVMSKPS
_mm_getcsr	Returns register contents	STMXCSR
_mm_setcsr	Sets control register	LDMXCSR

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

END Microsoft Specific

See Also

Reference

Streaming SIMD Extensions (SSE)

_mm_shuffle_ps

Microsoft Specific

Selects four specific single-precision, floating-point values from `a` and `b`, based on the mask `i`.

```
__m128 _mm_shuffle_ps(__m128 a , __m128 b , int i );
SHUFPS
```

Return Value

The mask must be an immediate. See Macro Function for Shuffle Using Streaming SIMD Extensions for a description of the shuffle semantics.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Intrinsics Using Streaming SIMD Extensions

_mm_shuffle_pi16

Microsoft Specific

Shuffles the four signed or unsigned 16-bit integers in `a` as specified by `imm`. The shuffle value `imm` must be an immediate.

```
__m64 _mm_shuffle_pi16(__m64 a, int imm);
PSHUFW
```

Return Value

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Intrinsics Using Streaming SIMD Extensions

_mm_unpackhi_ps

Microsoft Specific

Selects and interleaves the upper two single-precision, floating-point values from `a` and `b`.

```
__m128 _mm_unpackhi_ps(__m128 a , __m128 b );
UNPCKHPS
```

Return Value

```
r0 := a2
r1 := b2
r2 := a3
r3 := b3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Intrinsics Using Streaming SIMD Extensions

_mm_unpacklo_ps

Microsoft Specific

Selects and interleaves the lower two single-precision, floating-point values from `a` and `b`.

```
__m128 _mm_unpacklo_ps(__m128 a , __m128 b );
UNPCKLPS
```

Return Value

```
r0 := a0
r1 := b0
r2 := a1
r3 := b1
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Intrinsics Using Streaming SIMD Extensions

_mm_loadh_pi

Microsoft Specific

Sets the upper two single-precision, floating-point values with 64 bits of data loaded from the address `p`; the lower two values are passed through from `a`.

```
__m128 _mm_loadh_pi( __m128 a , __m64 * p );
MOVHPS reg, mem
```

Return Value

```
r0 := a0
r1 := a1
r2 := *p0
r3 := *p1
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Intrinsics Using Streaming SIMD Extensions

_mm_storeh_pi

Microsoft Specific

Stores the upper two single-precision, floating-point values of `a` to the address `p`.

```
void _mm_storeh_pi( __m64 * p ,__m128 a );
MOVHPS mem, reg
```

Return Value

```
*p0 := a2
*p1 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Intrinsics Using Streaming SIMD Extensions

_mm_movehl_ps

Microsoft Specific

Moves the upper two single-precision, floating-point values of `b` to the lower two single-precision, floating-point values of the result.

```
__m128 _mm_movehl_ps( __m128 a, __m128 b );
MOVHLPS
```

Return Value

The upper two single-precision, floating-point values of `a` are passed through to the result.

```
r3 := a3
r2 := a2
r1 := b3
r0 := b2
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Intrinsics Using Streaming SIMD Extensions

_mm_movelh_ps

Microsoft Specific

Moves the lower two single-precision, floating-point values of `b` to the upper two single-precision, floating-point values of the result.

```
__m128 _mm_movelh_ps( __m128 a, __m128 b );
MOVLHPS
```

Return Value

The lower two single-precision, floating-point values of `a` are passed through to the result.

```
r3 := b1
r2 := b0
r1 := a1
r0 := a0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Intrinsics Using Streaming SIMD Extensions

_mm_loadl_pi

Microsoft Specific

Sets the lower two single-precision, floating-point values with 64 bits of data loaded from the address `p`; the upper two values are passed through from `a`.

```
__m128 _mm_loadl_pi( __m128 a , __m64 * p );
MOVLPS reg, mem
```

Return Value

```
r0 := *p0
r1 := *p1
r2 := a2
r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Intrinsics Using Streaming SIMD Extensions

_mm_storel_pi

Microsoft Specific

Stores the lower two single-precision, floating-point values of `a` to the address `p`.

```
void _mm_storel_pi( __m64 * p , __m128 a );
MOVLPS mem, reg
```

Return Value

```
*p0 := b0
*p1 := b1
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Intrinsics Using Streaming SIMD Extensions

_mm_movemask_ps

Microsoft Specific

Creates a 4-bit mask from the most significant bits of the four single-precision, floating-point values.

```
int _mm_movemask_ps( __m128 a );
MOVMSKPS
```

Return Value

```
r := sign(a3)<<3 | sign(a2)<<2 | sign(a1)<<1 | sign(a0)
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Intrinsics Using Streaming SIMD Extensions

_mm_getcsr

Microsoft Specific

Returns the contents of the control register.

```
unsigned int _mm_getcsr(void);
STMXCSR
```

Return Value

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Intrinsics Using Streaming SIMD Extensions

_mm_setcsr

Microsoft Specific

Sets the control register to the value specified.

```
void _mm_setcsr(unsigned int i );
LDMXCSR
```

Return Value

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Intrinsics Using Streaming SIMD Extensions

Memory and Initialization Using Streaming SIMD Extensions

Microsoft Specific

This section describes the load, set, and store operations, which let you load and store data into memory. The load and set operations are similar in that both initialize `__m128` data. However, the set operation takes a float argument and is intended for initialization with constants, whereas the load operation takes a floating-point argument and is intended to mimic the instructions for loading data from memory. The store operation assigns the initialized data to the address.

- Load Operations
- Set Operations
- Store Operations

END Microsoft Specific
See Also
Reference
Streaming SIMD Extensions (SSE)

Load Operations (SSE)

Microsoft Specific

The intrinsics listed in the following table are followed by a description of each intrinsic.

SSE intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE intrinsics that use the __m64 data type are not supported on x64 processors.

The header file xmmintrin.h contains the declarations for the SSE intrinsics.

Memory and Initialization Load Operations

Intrinsic name	Operation	Corresponding instruction
_mm_load_ss	Loads the low value and clears the three high values	MOVSS
_mm_load1_ps, _mm_load_ps1	Loads one value into all four words	MOVSS + Shuffling
_mm_load_ps	Loads four values, address aligned	MOVAPS
_mm_loadu_ps	Loads four values, address unaligned	MOVUPS
_mm_loadr_ps	Loads four values, in reverse order	MOVAPS + Shuffling

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions

_mm_load_ss

Microsoft Specific

Loads an single-precision, floating-point value into the low word and clears the upper three words.

```
__m128 _mm_load_ss(float * p );
MOVSS
```

Return Value

```
r0 := *p
r1 := 0.0 ; r2 := 0.0 ; r3 := 0.0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Load Operations (SSE)

_mm_load1_ps, _mm_load_ps1

Microsoft Specific

Loads a single single-precision, floating-point value, copying it into all four words.

```
__m128 _mm_load1_ps(float * p );
MOVSS + shuffling
__m128 _mm_load_ps1(float * p );
MOVSS + shuffling
```

Return Value

```
r0 := *p
r1 := *p
r2 := *p
r3 := *p
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Load Operations (SSE)

_mm_load_ps

Microsoft Specific

Loads four single-precision, floating-point values.

```
__m128 _mm_load_ps(float * p );
MOVAPS
```

Return Value

The address must be 16-byte aligned.

```
r0 := p[0]
r1 := p[1]
r2 := p[2]
r3 := p[3]
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also
Reference
Load Operations (SSE)

_mm_loadu_ps

Microsoft Specific

Loads four single-precision, floating-point values.

```
__m128 _mm_loadu_ps(float * p);
MOVUPS
```

Return Value

The address does not need to be 16-byte aligned.

```
r0 := p[0]
r1 := p[1]
r2 := p[2]
r3 := p[3]
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also
Reference
Load Operations (SSE)

_mm_loadr_ps

Microsoft Specific

Loads four single-precision, floating-point values in reverse order.

```
__m128 _mm_loadr_ps(float * p );
MOVAPS + shuffling
```

Return Value

The address must be 16-byte aligned.

```
r0 := p[3]
r1 := p[2]
r2 := p[1]
r3 := p[0]
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also
Reference
Load Operations (SSE)

Set Operations (SSE)

Microsoft Specific

The operation intrinsics listed in the following table are followed by a description of each intrinsic.

SSE intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE intrinsics that use the __m64 data type are not supported on x64 processors.

The header file xmmintrin.h contains the declarations for the SSE intrinsics.

Memory and Initialization Set Operations

Intrinsic name	Operation	Corresponding instruction
_mm_set_ss	Sets the low value and clears the three high values	Composite
_mm_set1_ps, _mm_set_ps1	Sets all four words with the same value	Composite
_mm_set_ps	Sets four values, address aligned	Composite
_mm_setr_ps	Sets four values, in reverse order	Composite
_mm_setzero_ps	Clears all four values	Composite

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions

_mm_set_ss

Microsoft Specific

Sets the low word of an single-precision, floating-point value to `w` and clears the upper three words.

```
__m128 _mm_set_ss(float w );
(composite)
```

Return Value

```
r0 := w
r1 := r2 := r3 := 0.0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (SSE)

_mm_set1_ps, _mm_set_ps1

Microsoft Specific

Sets the four single-precision, floating-point values to `w`.

```
__m128 _mm_set1_ps(float w );
(composite)
__m128 _mm_set_ps1(float w );
(composite)
```

Return Value

```
r0 := r1 := r2 := r3 := w
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (SSE)

_mm_set_ps

Microsoft Specific

Sets the four single-precision, floating-point values to the four inputs.

```
__m128 _mm_set_ps(float z , float y , float x , float w );
(composite)
```

Return Value

```
r0 := w
r1 := x
r2 := y
r3 := z
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (SSE)

_mm_setr_ps

Microsoft Specific

Sets the four single-precision, floating-point values to the four inputs in reverse order.

```
__m128 _mm_setr_ps(float z , float y , float x , float w );
(composite)
```

Return Value

```
r0 := z
r1 := y
r2 := x
r3 := w
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (SSE)

_mm_setzero_ps

Microsoft Specific

Clears the four single-precision, floating-point values.

```
__m128 _mm_setzero_ps(void);
(composite)
```

Return Value

```
r0 := r1 := r2 := r3 := 0.0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (SSE)

Store Operations (SSE)

Microsoft Specific

The operation intrinsics listed in the following table are followed by a description of each intrinsic.

SSE intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE intrinsics that use the __m64 data type are not supported on x64 processors.

The header file xmmintrin.h contains the declarations for the SSE intrinsics.

Memory and Initialization Store Operations

Intrinsic name	Operation	Corresponding instruction
_mm_store_ss	Stores the low value	MOVSS
_mm_store1_ps, _mm_store_ps1	Stores the low value across all four words	MOVSS + Shuffling
_mm_store_ps	Stores four values, address aligned	MOVAPS
_mm_storeu_ps	Stores four values, address unaligned	MOVUPS
_mm_storer_ps	Stores four values, in reverse order	MOVAPS + Shuffling
_mm_move_ss	Sets the low word, and passes in three high values	MOVSS

For an explanation of the syntax used in code samples in this topic, see
Floating-Point Intrinsics Using Streaming SIMD Extensions.

END Microsoft Specific
See Also
Reference
Integer Intrinsics Using Streaming SIMD Extensions

_mm_store_ss

Microsoft Specific

Stores the lower single-precision, floating-point value.

```
void _mm_store_ss(float * p , __m128 a );
MOVSS
```

Return Value

```
*p := a0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Store Operations (SSE)

_mm_store1_ps, _mm_store_ps1

Microsoft Specific

Stores the lower single-precision, floating-point value across four words.

```
void _mm_store1_ps(float * p , __m128 a );
MOVSS+ shuffling
void _mm_store_ps1(float * p , __m128 a );
MOVSS+ shuffling
```

Return Value

```
p[0] := a0
p[1] := a0
p[2] := a0
p[3] := a0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Store Operations (SSE)

_mm_store_ps

Microsoft Specific

Stores four single-precision, floating-point values.

```
void _mm_store_ps(float *p, __m128 a );
MOVAPS
```

Return Value

The address must be 16-byte aligned.

```
p[0] := a0
p[1] := a1
p[2] := a2
p[3] := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also
Reference
Store Operations (SSE)

_mm_storeu_ps

Microsoft Specific

Stores four single-precision, floating-point values.

```
void _mm_storeu_ps(float *p, __m128 a);
MOVUPS
```

Return Value

The address does not need to be 16-byte aligned.

```
p[0] := a0
p[1] := a1
p[2] := a2
p[3] := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Store Operations (SSE)

_mm_storer_ps

Microsoft Specific

Stores four single-precision, floating-point values in reverse order.

```
void _mm_storer_ps(float * p , __m128 a );
MOVAPS + shuffling
```

Return Value

The address must be 16-byte aligned.

```
p[0] := a3
p[1] := a2
p[2] := a1
p[3] := a0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Store Operations (SSE)

_mm_move_ss

Microsoft Specific

Sets the low word to the single-precision, floating-point value of `b`.

```
__m128 _mm_move_ss( __m128 a , __m128 b );
MOVSS
```

Return Value

The upper 3 single-precision, floating-point values are passed through from a.

```
r0 := b0
r1 := a1
r2 := a2
r3 := a3
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also
Reference
Store Operations (SSE)

Integer Intrinsics Using Streaming SIMD Extensions

Microsoft Specific

The intrinsics listed in the table are followed by a description of each intrinsic with the most recent mnemonic naming convention.

SSE intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE intrinsics that use the __m64 data type are not supported on x64 processors.

The header file xmmintrin.h contains the declarations for the SSE intrinsics.

Integer Intrinsics

Intrinsic name	Operation	Corresponding instruction
_mm_extract_pi16	Extracts one of four words	PEXTRW
_mm_insert_pi16	Inserts a word	PINSRW
_mm_max_pi16	Computes the maximum	PMAXSW
_mm_max_pu8	Computes the maximum, unsigned	PMAXUB
_mm_min_pi16	Computes the minimum	PMINSW
_mm_min_pu8	Computes the minimum, unsigned	PMINUB
_mm_movemask_pi8	Creates an 8-bit mask	PMOVMSKB
_mm_mulhi_pu16	Multiplies, returning high bits	PMULHUW
_mm_maskmove_si64	Returns a combination of four words	PSHUFW
_mm_avg_pu8	Computes conditional store	MASKMOVQ
_mm_avg_pu16	Computes rounded average	PAVGB
_mm_sad_pu8	Computes rounded average	PAVGW
_mm_shuffle_pi16 (Streaming SIMD Extensions)	Computes sum of absolute differences	PSADBW

For this section you need to empty the multimedia state for the MMX register. See the Understanding the EMMS Instruction section.

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

END Microsoft Specific

See Also

Reference

Streaming SIMD Extensions (SSE)

_mm_extract_pi16

Microsoft Specific

Extracts one of the four words of `a`.

```
int _mm_extract_pi16(__m64 a , int n );
PEXTRW
```

Return Value

The selector `n` must be an immediate.

```
r := (n==0) ? a0 : ( (n==1) ? a1 : ( (n==2) ? a2 : a3 ) )
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions

_mm_insert_pi16

Microsoft Specific

Inserts word `d` into one of four words of `a`.

```
__m64 _mm_insert_pi16(__m64 a , int d , int n );
PINSRW
```

Return Value

The selector `n` must be an immediate.

```
r0 := (n==0) ? d : a0;
r1 := (n==1) ? d : a1;
r2 := (n==2) ? d : a2;
r3 := (n==3) ? d : a3;
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions

_mm_max_pi16

Microsoft Specific

Computes the element-wise maximum of the words in `a` and `b`.

```
__m64 _mm_max_pi16(__m64 a , __m64 b );
PMAXSW
```

Return Value

```
r0 := min(a0, b0)
r1 := min(a1, b1)
r2 := min(a2, b2)
r3 := min(a3, b3)
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions

_mm_max_pu8

Microsoft Specific

Computes the element-wise maximum of the unsigned bytes in `a` and `b`.

```
__m64 _mm_max_pu8(__m64 a , __m64 b );
PMAXUB
```

Return Value

```
r0 := min(a0, b0)
r1 := min(a1, b1)
...
r7 := min(a7, b7)
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions

_mm_min_pi16

Microsoft Specific

Computes the element-wise minimum of the words in `a` and `b`.

```
__m64 _mm_min_pi16(__m64 a , __m64 b );
PMINSW
```

Return Value

```
r0 := min(a0, b0)
r1 := min(a1, b1)
r2 := min(a2, b2)
r3 := min(a3, b3)
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions

_mm_min_pu8

Microsoft Specific

Computes the element-wise minimum of the unsigned bytes in `a` and `b`.

```
__m64 _mm_min_pu8(__m64 a , __m64 b );
PMINUB
```

Return Value

```
r0 := min(a0, b0)
r1 := min(a1, b1)
...
r7 := min(a7, b7)
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions

_mm_movemask_pi8

Microsoft Specific

Creates an 8-bit mask from the most significant bits of the bytes in `a`.

```
int _mm_movemask_pi8(__m64 a );
PMOVMSKB
```

Return Value

```
r := sign(a7)<<7 | sign(a6)<<6 | ... | sign(a0)
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions

_mm_mulhi_pu16

Microsoft Specific

Multiplies the unsigned words in `a` and `b`, returning the upper 16 bits of the 32-bit intermediate results.

```
__m64 _mm_mulhi_pu16(__m64 a , __m64 b );
PMULHUW
```

Return Value

```
r0 := hiword(a0 * b0)
r1 := hiword(a1 * b1)
r2 := hiword(a2 * b2)
r3 := hiword(a3 * b3)
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions

_mm_maskmove_si64

Microsoft Specific

Conditionally stores byte elements of `d` to address `p`.

```
void _mm_maskmove_si64(__m64 d , __m64 n , char * p);
MASKMOVQ
```

Return Value

The high bit of each byte in the selector `n` determines whether the corresponding byte in `d` will be stored.

```
if (sign(n0)) p[0] := d0
if (sign(n1)) p[1] := d1
...
if (sign(n7)) p[7] := d7
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions

_mm_avg_pu8

Microsoft Specific

Computes the (rounded) averages of the unsigned bytes in `a` and `b`.

```
__m64 _mm_avg_pu8(__m64 a, __m64 b);
PAVGB
```

Return Value

```
t = (unsigned short)a0 + (unsigned short)b0
r0 = (t >> 1) | (t & 0x01)
...
t = (unsigned short)a7 + (unsigned short)b7
r7 = (unsigned char)((t >> 1) | (t & 0x01))
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions

_mm_avg_pu16

Microsoft Specific

Computes the (rounded) averages of the unsigned words in a and b.

```
__m64 _mm_avg_pu16(__m64 a, __m64 b);
PAVGW
```

Return Value

```
t = (unsigned int)a0 + (unsigned int)b0
r0 = (t >> 1) | (t & 0x01)
...
t = (unsigned word)a7 + (unsigned word)b7
r7 = (unsigned short)((t >> 1) | (t & 0x01))
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions

_mm_sad_pu8

Microsoft Specific

Computes the sum of the absolute differences of the unsigned bytes in `a` and `b`, returning the value in the lower word.

```
__m64 _mm_sad_pu8(__m64 a, __m64 b);
PSADBW
```

Return Value

The upper three words are cleared.

```
r0 = abs(a0-b0) + ... + abs(a7-b7)
r1 = r2 = r3 = 0
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions

_mm_shuffle_pi16 (Streaming SIMD Extensions)

Microsoft Specific

Returns a combination of the four words of a.

```
__m64 _mm_shuffle_pi16(__m64 a , int n );
PSHUFW
```

Return Value

The selector n must be an immediate.

```
r0 := word (n&0x3) of a
r1 := word ((n>>2)&0x3) of a
r2 := word ((n>>4)&0x3) of a
r3 := word ((n>>6)&0x3) of a
```

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions

Cache Support Using Streaming SIMD Extensions

Microsoft Specific

The following intrinsics provide ways to make efficient use of the cache. For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

Intrinsic	Description
_mm_prefetch	Loads one cache line of data from address `p` to a location closer to the processor.
_mm_stream_pi	Stores the data in `a` to the address `p` without polluting the caches.
_mm_stream_ps	Stores the data in `a` to the address `p` without polluting the caches.
_mm_sfence	Guarantees that every preceding store is globally visible before any subsequent store.

Requirements

Header: xmmintrin.h

END Microsoft Specific

See Also

Reference

Streaming SIMD Extensions (SSE)

_mm_prefetch

Microsoft Specific

Loads one cache line of data from address `p` to a location closer to the processor.

```
void _mm_prefetch(char * p , int i );
PREFETCH
```

Return Value

The value `i` specifies the type of prefetch operation: the constants `_MM_HINT_T0`, `_MM_HINT_T1`, `_MM_HINT_T2`, and `_MM_HINT_NTA`, corresponding to the type of prefetch instruction, should be used.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Cache Support Using Streaming SIMD Extensions

_mm_stream_pi

Microsoft Specific

Stores the data in `a` to the address `p` without polluting the caches.

```
void _mm_stream_pi(__m64 * p , __m64 a );
MOVNTQ
```

Return Value

This intrinsic requires you to empty the multimedia state for the MMX register. See Understanding the EMMS Instruction section.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Cache Support Using Streaming SIMD Extensions

_mm_stream_ps

Microsoft Specific

Stores the data in `a` to the address `p` without polluting the caches.

```
void _mm_stream_ps(float * p , __m128 a );
MOVNTPS
```

Return Value

The address must be 16-byte aligned.

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Cache Support Using Streaming SIMD Extensions

_mm_sfence

Microsoft Specific

Guarantees that every preceding store is globally visible before any subsequent store.

```
void _mm_sfence(void);
SFENCE
```

Return Value

Requirements

Header: mmintrin.h

END Microsoft Specific

See Also

Reference

Cache Support Using Streaming SIMD Extensions

Macro Function for Shuffle Using Streaming SIMD Extensions

Microsoft Specific

SSE provides a macro function to help create constants that describe shuffle operations. The macro takes four small integers (in the range of 0 to 3) and combines them into an 8-bit immediate value used by the `SHUFPS` instruction. See the following example.

Shuffle Function Macro

```
_MM_SHUFFLE(z, y, x, w)
/* expands to the following value */
(z<<6) | (y<<4) | (x<<2) | w
```

You can view the four integers as selectors for choosing which two words from the first input operand and which two words from the second are to be put into the result word.

View of Original and Result Words with Shuffle Function Macro

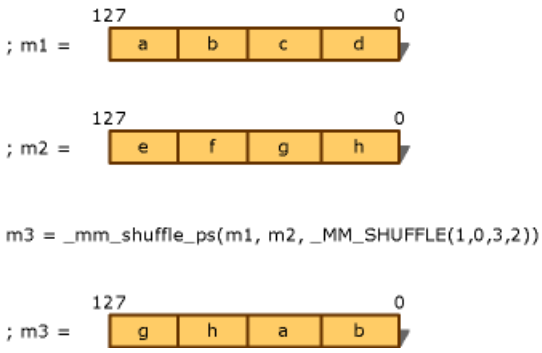

END Microsoft Specific

See Also

Reference

Streaming SIMD Extensions (SSE)

Miscellaneous Intrinsics Using Streaming SIMD Extensions

Macro Functions to Read and Write the Control Registers

Microsoft Specific

The following macro functions enable you to read and write bits to and from the control register. For more details, see Set Operations.

- _MM_SET_EXCEPTION_STATE
- _MM_GET_EXCEPTION_STATE
- _MM_SET_EXCEPTION_MASK
- _MM_GET_EXCEPTION_MASK
- _MM_SET_ROUNDING_MODE
- _MM_GET_ROUNDING_MODE
- _MM_SET_FLUSH_ZERO_MODE
- _MM_GET_FLUSH_ZERO_MODE

END Microsoft Specific

See Also
Reference
Streaming SIMD Extensions (SSE)

_MM_SET_EXCEPTION_STATE

Microsoft Specific

Writes to the sixth-least significant control register bit.

```
_MM_SET_EXCEPTION_STATE(
      x
)
```

Parameters

x

One of the following values:

- **_MM_EXCEPT_INVALID**
- **_MM_EXCEPT_DIV_ZERO**
- **_MM_EXCEPT_DENORM**
- **_MM_EXCEPT_OVERFLOW**
- **_MM_EXCEPT_UNDERFLOW**
- **_MM_EXCEPT_INEXACT**

Remarks

The header file xmmintrin.h contains the declarations for the SSE intrinsics.

END Microsoft Specific

See Also

Reference

Macro Functions to Read and Write the Control Registers

_MM_GET_EXCEPTION_STATE

_MM_GET_EXCEPTION_STATE

Microsoft Specific

Reads from the sixth-least significant control register bit.

```
_MM_GET_EXCEPTION_STATE( )
```

Example

The following example tests for a divide by zero exception.

```
if (_MM_GET_EXCEPTION_STATE(x) & _MM_EXCEPT_DIV_ZERO) {
      /* Exception has occurred */
}
```

The header file xmmintrin.h contains the declarations for the SSE intrinsics.

END Microsoft Specific

See Also

Reference

Macro Functions to Read and Write the Control Registers

_MM_SET_EXCEPTION_STATE

_MM_SET_EXCEPTION_MASK

Microsoft Specific

Writes to the seventh through twelfth control register bits.

```
_MM_SET_EXCEPTION_MASK(
      x
)
```

Parameters

x

One of the following values:

- **_MM_MASK_INVALID**
- **_MM_MASK_DIV_ZERO**
- **_MM_MASK_DENORM**
- **_MM_MASK_OVERFLOW**
- **_MM_MASK_UNDERFLOW**
- **_MM_MASK_INEXACT**

Remarks

The header file xmmintrin.h contains the declarations for the SSE intrinsics.

Example

The following example masks the overflow and underflow exceptions and unmasks all other exceptions.

```
_MM_SET_EXCEPTION_MASK( _MM_MASK_OVERFLOW | _MM_MASK_UNDERFLOW)
```

END Microsoft Specific

See Also

Reference

Macro Functions to Read and Write the Control Registers
_MM_GET_EXCEPTION_MASK

_MM_GET_EXCEPTION_MASK

Microsoft Specific

Reads from the seventh through twelfth control register bits.

```
_MM_SET_EXCEPTION_MASK( )
```

Remarks

The header file xmmintrin.h contains the declarations for the SSE intrinsics.

END Microsoft Specific

See Also

Reference

Macro Functions to Read and Write the Control Registers

_MM_SET_EXCEPTION_MASK

_MM_SET_ROUNDING_MODE

Microsoft Specific

Writes to bits thirteen and fourteen of the control register.

```
_MM_SET_ROUNDING_MODE(
      x
)
```

Parameters

x

One of the following values:

- **_MM_ROUND_NEAREST**
- **_MM_ROUND_DOWN**
- **_MM_ROUND_UP**
- **_MM_ROUND_TOWARD_ZERO**

Remarks

The header file xmmintrin.h contains the declarations for the SSE intrinsics.

END Microsoft Specific

See Also

Reference

Macro Functions to Read and Write the Control Registers

_MM_GET_ROUNDING_MODE

_MM_GET_ROUNDING_MODE

Microsoft Specific

Reads from bits thirteen and fourteen of the control register.

```
_MM_GET_ROUNDING_MODE( )
```

Remarks

The header file xmmintrin.h contains the declarations for the SSE intrinsics.

Example

The following example tests the rounding mode for round toward zero.

```
if (_MM_GET_ROUNDING_MODE() == _MM_ROUND_TOWARD_ZERO)
{
   /* Rounding mode is round toward zero */
}
```

END Microsoft Specific

See Also

Reference

Macro Functions to Read and Write the Control Registers

_MM_SET_ROUNDING_MODE

_MM_SET_FLUSH_ZERO_MODE

Microsoft Specific

Writes to bit fifteen of the control register.

```
_MM_SET_FLUSH_ZERO_MODE(
      x
)
```

Parameters

x

One of the following values:

- **_MM_FLUSH_ZERO_ON**
- **_MM_FLUSH_ZERO_OFF**

Remarks

The header file xmmintrin.h contains the declarations for the SSE intrinsics.

Example

The following example disables flush to zero mode.

```
_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_OFF)
```

END Microsoft Specific

See Also

Reference

Macro Functions to Read and Write the Control Registers

_MM_GET_FLUSH_ZERO_MODE

_MM_GET_FLUSH_ZERO_MODE

Microsoft Specific

Reads from bit fifteen of the control register.

```
_MM_GET_FLUSH_ZERO_MODE( )
```

Remarks

The header file xmmintrin.h contains the declarations for the SSE intrinsics.

END Microsoft Specific

See Also

Reference

Macro Functions to Read and Write the Control Registers

_MM_SET_FLUSH_ZERO_MODE

Macro Function for Matrix Transposition

Microsoft Specific

Transposes a 4-by-4 matrix of single-precision, floating-point values.

```
_MM_TRANSPOSE4_PS(row0, row1, row2, row3)
```

Remarks

The arguments `row0`, `row1`, `row2`, and `row3` are **__m128** values whose elements form the corresponding rows of a 4-by-4 matrix. The matrix transposition is returned in arguments `row0`, `row1`, `row2`, and `row3` where `row0` now holds `column 0` of the original matrix, `row1` now holds `column 1` of the original matrix, and so on.

The transposition function of this macro is illustrated in the following figure.

The header file xmmintrin.h contains the declarations for the SSE intrinsics.

Matrix Transposition Using the _MM_TRANSPOSE4_PS Macro

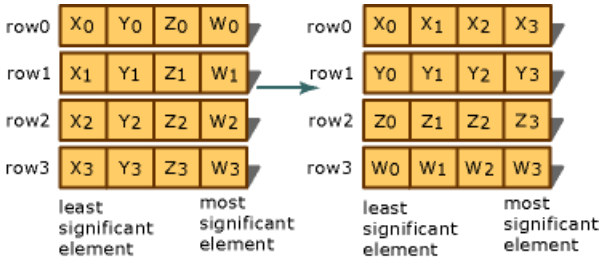

END Microsoft Specific

See Also

Reference

Streaming SIMD Extensions (SSE)

Streaming SIMD Extensions 2 Instructions

Microsoft Specific

This section describes the C/C++ language-level features supporting the Streaming SIMD Extensions 2 (SSE2) instructions:

- Floating-Point Intrinsics Using Streaming SIMD Extensions 2 Instructions that describe the intrinsic operations for the double-precision, floating-point data type (**__m128d**).
- Integer Intrinsics Using Streaming SIMD Extensions 2 that describe the intrinsics for the extended-precision integer data type (**__m128i**).

Other topics discussed in this section include:

- Floating-Point Memory and Initialization Operations Using Streaming SIMD Extensions 2
- Cache Support for Streaming SIMD Extensions 2 Floating-Point Operations
- Integer Memory and Initialization Using Streaming SIMD Extensions 2
- Cache Support for Streaming SIMD Extensions 2 Integer Operations
- Macro Function for Shuffle Using Streaming SIMD Extensions 2

The emmintrin.h header file contains the declarations for the SSE2 instructions intrinsics. The file dvec.h contains operator overloads for some of the SSE2 instructions intrinsics, which are available for use in C++ programs.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

END Microsoft Specific

See Also

Reference

MMX, SSE, and SSE2 Intrinsics

Floating-Point Intrinsics Using Streaming SIMD Extensions 2 Instructions

Microsoft Specific

The following topics list the floating-point and integer intrinsics broken into groups by the nature of the operation. Each intrinsic entry has an informal pseudo-code, and it is followed with a corresponding instruction name in uppercase letters; for example, `ADDSD` is the name of the first instruction listed in this section. The variable **r** is generally used for the intrinsic's return value. A number appended to a variable name indicates the element of a packed object. For example, **r0** is the lowest double of **r**. Some intrinsics are *composites* because they require more than one instruction to implement them. For more details, refer to the Streaming SIMD Extensions 2 (SSE2) instructions external architecture specification (EAS). You should be familiar with the hardware features provided by the SSE2 instructions when writing programs with the intrinsics. The following are three important issues to keep in mind:

- Certain intrinsics, such as **_mm_loadr_pd** and **_mm_cmpgt_sd**, are not directly supported by the instruction set. While these intrinsics are convenient programming aids, be mindful of their implementation cost.
- Data loaded or stored as **__m128d** objects must be generally 16-byte aligned.
- Some intrinsics require that their argument be immediates, that is, constant integers (literals), because of the nature of the instruction.

This section contains the following topics:

- Arithmetic Operations
- Logical Operations
- Comparisons
- Conversion Operations
- Miscellaneous Operations

END Microsoft Specific
See Also
Reference
Streaming SIMD Extensions 2 Instructions

Arithmetic Operations (Floating Point)

Microsoft Specific

The operations listed in the following table are followed by descriptions of each intrinsic.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

The emmintrin.h header file contains the declarations for the SSE2 instructions intrinsics.

Arithmetic Operation Intrinsics

Intrinsic name	Corresponding instruction	Operation	R0 value	R1 value
_mm_add_sd	ADDSD	Adds	`a0 [op] b0`	`a1`
_mm_add_pd	ADDPD	Adds	`a0 [op] b0`	`a1 [op] b1`
_mm_div_sd	DIVSD	Divides	`a0 [op] b0`	`a1`
_mm_div_pd	DIVPD	Divides	`a0 [op] b0`	`a1 [op] b1`
_mm_max_sd	MAXSD	Computes maximum	`a0 [op] b0`	`a1`
_mm_max_pd	MAXPD	Computes maximum	`a0 [op] b0`	`a1 [op] b1`
_mm_min_sd	MINSD	Computes minimum	`a0 [op] b0`	`a1`
_mm_min_pd	MINPD	Computes minimum	`a0 [op] b0`	`a1 [op] b1`
_mm_mul_sd	MULSD	Multiplies	`a0 [op] b0`	`a1`
_mm_mul_pd	MULPD	Multiplies	`a0 [op] b0`	`a1 [op] b1`
_mm_sqrt_sd	SQRTSD	Computes square root	`a0 [op] b0`	`a1`
_mm_sqrt_pd	SQRTPD	Computes square root	`a0 [op] b0`	`a1 [op] b1`
_mm_sub_sd	SUBSD	Subtracts	`a0 [op] b0`	`a1`
_mm_sub_pd	SUBPD	Subtracts	`a0 [op] b0`	`a1 [op] b1`

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

END Microsoft Specific

See Also

Reference

Floating-Point Intrinsics Using Streaming SIMD Extensions 2 Instructions

_mm_add_sd

Microsoft Specific

Adds the lower double-precision, floating-point values of `a` and `b`; the upper double-precision, floating-point value is passed through from `a`.

```
__m128d _mm_add_sd(__m128d a, __m128d b);
ADDSD
```

Return Value

```
r0 := a0 + b0
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating Point)

_mm_add_pd

Microsoft Specific

Adds the two double-precision, floating-point values of `a` and `b`.

```
__m128d _mm_add_pd(__m128d a, __m128d b);
ADDPD
```

Return Value

```
r0 := a0 + b0
r1 := a1 + b1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating Point)

_mm_div_sd

Microsoft Specific

Divides the lower double-precision, floating-point values of `a` and `b`.

```
__m128d _mm_div_sd(__m128d a, __m128d b);
DIVSD
```

Return Value

The upper double-precision, floating-point value is passed through from `a`.

```
r0 := a0 / b0
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Arithmetic Operations (Floating Point)

_mm_div_pd

Microsoft Specific

Divides the two double-precision, floating-point values of `a` and `b`.

```
__m128d _mm_div_pd(__m128d a, __m128d b);
DIVPD
```

Return Value

```
r0 := a0 / b0
r1 := a1 / b1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating Point)

_mm_max_sd

Microsoft Specific

Computes the maximum of the lower double-precision, floating-point values of `a` and `b`.

```
__m128d _mm_max_sd(__m128d a, __m128d b);
MAXSD
```

Return Value

The upper double-precision, floating-point value is passed through from `a`.

```
r0 := max (a0, b0)
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating Point)

_mm_max_pd

Microsoft Specific

Computes the maxima of the two double-precision, floating-point values of `a` and `b`.

```
__m128d _mm_max_pd(__m128d a, __m128d b);
MAXPD
```

Return Value

```
r0 := max(a0, b0)
r1 := max(a1, b1)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating Point)

_mm_min_sd

Microsoft Specific

Computes the minimum of the lower double-precision, floating-point values of `a` and `b`.

```
__m128d _mm_min_sd(__m128d a, __m128d b);
MINSD
```

Return Value

The upper double-precision, floating-point value is passed through from `a`.

```
r0 := min (a0, b0)
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating Point)

_mm_min_pd

Microsoft Specific

Computes the minima of the two double-precision, floating-point values of `a` and `b`.

```
__m128d _mm_min_pd(__m128d a, __m128d b);
MINPD
```

Return Value

```
r0 := min(a0, b0)
r1 := min(a1, b1)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating Point)

_mm_mul_sd

Microsoft Specific

Multiplies the lower double-precision, floating-point values of `a` and `b`.

```
__m128d _mm_mul_sd(__m128d a, __m128d b);
MULSD
```

Return Value

The upper double-precision, floating-point is passed through from `a`.

```
r0 := a0 * b0
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating Point)

_mm_mul_pd

Microsoft Specific

Multiplies the two double-precision, floating-point values of `a` and `b`.

```
__m128d _mm_mul_pd(__m128d a, __m128d b);
MULPD
```

Return Value

```
r0 := a0 * b0
r1 := a1 * b1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating Point)

_mm_sqrt_sd

Microsoft Specific

Computes the square root of the lower double-precision, floating-point value of `b`.

```
__m128d _mm_sqrt_sd(__m128d a, __m128d b);
SQRTSD
```

Return Value

The upper double-precision, floating-point value is passed through from `a`.

```
r0 := sqrt(b0)
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating Point)

_mm_sqrt_pd

Microsoft Specific

Computes the square roots of the two double-precision, floating-point values of `a`.

```
__m128d _mm_sqrt_pd(__m128d a);
SQRTPD
```

Return Value

```
r0 := sqrt(a0)
r1 := sqrt(a1)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating Point)

_mm_sub_sd

Microsoft Specific

Subtracts the lower double-precision, floating-point value of `b` from `a`.

```
__m128d _mm_sub_sd(__m128d a, __m128d b);
SUBSD
```

Return Value

The upper double-precision, floating-point value is passed through from `a`.

```
r0 := a0 - b0
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating Point)

_mm_sub_pd

Microsoft Specific

Subtracts the two double-precision, floating-point values of `b` from `a`.

```
__m128d _mm_sub_pd(__m128d a, __m128d b);
SUBPD
```

Return Value

```
r0 := a0 - b0
r1 := a1 - b1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Floating Point)

Logical Operations (Floating-Point SSE2 Intrinsics)

Microsoft Specific

For an explanation of the syntax used in code samples in this topic, see
Floating-Point Intrinsics Using Streaming SIMD Extensions.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

The emmintrin.h header file contains the declarations for the SSE2 instructions intrinsics.

```
__m128d _mm_andnot_pd (__m128d a, __m128d b);
```

```
ANDNPD
```

Computes the bitwise AND of the 128-bit value in b and the bitwise NOT of the 128-bit value in a.

```
r0 := (~a0) & b0
r1 := (~a1) & b1
```

```
__m128d _mm_and_pd (__m128d a, __m128d b);
```

```
ANDPD
```

Computes the bitwise AND of the two double-precision, floating-point values of a and b.

```
r0 := a0 & b0
r1 := a1 & b1
```

```
__m128d _mm_or_pd (__m128d a, __m128d b);
```

```
ORPD
```

Computes the bitwise OR of the two double-precision, floating-point values of a and b.

```
r0 := a0 | b0
r1 := a1 | b1
```

```
__m128d _mm_xor_pd (__m128d a, __m128d b);
```

```
XORPD
```

Computes the bitwise XOR of the two double-precision, floating-point values of a and b .

```
r0 := a0 ^ b0
r1 := a1 ^ b1
```

END Microsoft Specific

See Also

Reference

_mm_andnot_pd

Microsoft Specific

Computes the bitwise `AND` of the 128-bit value in `b` and the bitwise `NOT` of the 128-bit value in `a`.

```
__m128d _mm_andnot_pd (__m128d a, __m128d b);
ANDNPD
```

Return Value

```
r0 := (~a0) & b0
r1 := (~a1) & b1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Logical Operations (Floating-Point SSE2 Intrinsics)

_mm_and_pd

Microsoft Specific

Computes the bitwise AND of the two double-precision, floating-point values of a and b.

```
__m128d _mm_and_pd (__m128d a, __m128d b);
ANDPD
```

Return Value

```
r0 := a0 & b0
r1 := a1 & b1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Logical Operations (Floating-Point SSE2 Intrinsics)

_mm_or_pd

Microsoft Specific

Computes the bitwise `OR` of the two double-precision, floating-point values of `a` and `b`.

```
__m128d _mm_or_pd (__m128d a, __m128d b);
ORPD
```

Return Value

```
r0 := a0 | b0
r1 := a1 | b1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Logical Operations (Floating-Point SSE2 Intrinsics)

_mm_xor_pd

Microsoft Specific

Computes the bitwise `XOR` of the two double-precision, floating-point values of `a` and `b` .

```
__m128d _mm_xor_pd (__m128d a, __m128d b);
XORPD
```

Return Value

```
r0 := a0 ^ b0
r1 := a1 ^ b1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Logical Operations (Floating-Point SSE2 Intrinsics)

Comparisons

Microsoft Specific

Each comparison intrinsic performs a comparison of `a` and `b`. For the packed form, the two double-precision, floating-point values of `a` and `b` are compared, and a 128-bit mask is returned. For the scalar form, the lower double-precision, floating-point values of `a` and `b` are compared, and a 64-bit mask is returned; the upper double-precision, floating-point value is passed through from `a`. The mask is set to `0xffffffffffffffff` for each element where the comparison is true and `0x0` where the comparison is false. The `r` following the instruction name indicates that the operands to the instruction are reversed in the actual implementation.

The comparison intrinsics are listed in the following table, followed by detailed descriptions.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

The emmintrin.h header file contains the declarations for the SSE2 instructions intrinsics.

Comparison Intrinsics

Intrinsic name	Corresponding instruction	Compare for
_mm_cmpeq_pd	CMPEQPD	Equality
_mm_cmplt_pd	CMPLTPD	Less than
_mm_cmple_pd	CMPLEPD	Less than or equal
_mm_cmpgt_pd	CMPLTPDr	Greater than
_mm_cmpge_pd	CMPLEPDr	Greater than or equal
_mm_cmpord_pd	CMPORDPD	Ordered
_mm_cmpunord_pd	CMPUNORDPD	Unordered
_mm_cmpneq_pd	CMPNEQPD	Inequality
_mm_cmpnlt_pd	CMPNLTPD	Not less than
_mm_cmpnle_pd	CMPNLEPD	Not less than or equal
_mm_cmpngt_pd	CMPNLTPDr	Not greater than
_mm_cmpnge_pd	CMPLEPDr	Not greater than or equal
_mm_cmpeq_sd	CMPEQSD	Equality
_mm_cmplt_sd	CMPLTSD	Less than
_mm_cmple_sd	CMPLESD	Less than or equal
_mm_cmpgt_sd	CMPLTSDr	Greater than
_mm_cmpge_sd	CMPLESDr	Greater than or equal
_mm_cmpord_sd	CMPORDSD	Ordered

_mm_cmpunord_sd	CMPUNORDSD	Unordered
_mm_cmpneq_sd	CMPNEQSD	Inequality
_mm_cmpnlt_sd	CMPNLTSD	Not less than
_mm_cmpnle_sd	CMPNLESD	Not less than or equal
_mm_cmpngt_sd	CMPNLTSDr	Not greater than
_mm_cmpnge_sd	CMPNLESDR	Not greater than or equal
_mm_comieq_sd	COMISD	Equality
_mm_comilt_sd	COMISD	Less than
_mm_comile_sd	COMISD	Less than or equal
_mm_comigt_sd	COMISD	Greater than
_mm_comige_sd	COMISD	Greater than or equal
_mm_comineq_sd	COMISD	Not equal
_mm_ucomieq_sd	UCOMISD	Equality
_mm_ucomilt_sd	UCOMISD	Less than
_mm_ucomile_sd	UCOMISD	Less than or equal
_mm_ucomigt_sd	UCOMISD	Greater than
_mm_ucomige_sd	UCOMISD	Greater than or equal
_mm_ucomineq_sd	UCOMISD	Not equal

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

END Microsoft Specific

See Also

Reference

Floating-Point Intrinsics Using Streaming SIMD Extensions 2 Instructions

_mm_cmpeq_pd

Microsoft Specific

Compares the two double-precision, floating-point values of `a` and `b` for equality.

```
__m128d _mm_cmpeq_pd (__m128d a, __m128d b);
CMPEQPD
```

Return Value

```
r0 := (a0 == b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 == b1) ? 0xffffffffffffffff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparisons

_mm_cmplt_pd

Microsoft Specific

Compares the two double-precision, floating-point values of `a` and `b` for `a` less than `b`.

```
__m128d _mm_cmplt_pd (__m128d a, __m128d b);
CMPLTPD
```

Return Value

```
r0 := (a0 < b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 < b1) ? 0xffffffffffffffff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparisons

_mm_cmple_pd

Microsoft Specific

Compares the two double-precision, floating-point values of `a` and `b` for `a` less than or equal to `b`.

```
__m128d _mm_cmple_pd (__m128d a, __m128d b);
CMPLEPD
```

Return Value

```
r0 := (a0 <= b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 <= b1) ? 0xffffffffffffffff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparisons

_mm_cmpgt_pd

Microsoft Specific

Compares the two double-precision, floating-point values of `a` and `b` for `a` greater than `b`.

```
__m128d _mm_cmpgt_pd (__m128d a, __m128d b);
CMPLTPDr
```

Return Value

```
r0 := (a0 > b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 > b1) ? 0xffffffffffffffff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparisons

_mm_cmpge_pd

Microsoft Specific

Compares the two double-precision, floating-point values of `a` and `b` for `a` greater than or equal to `b`.

```
__m128d _mm_cmpge_pd (__m128d a, __m128d b);
CMPLEPDr
```

Return Value

```
r0 := (a0 >= b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 >= b1) ? 0xffffffffffffffff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparisons

_mm_cmpord_pd

Microsoft Specific

Compares the two double-precision, floating-point values of `a` and `b` for ordered.

```
__m128d _mm_cmpord_pd (__m128d a, __m128d b);
CMPORDPD
```

Return Value

```
r0 := (a0 ord b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 ord b1) ? 0xffffffffffffffff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparisons

_mm_cmpunord_pd

Microsoft Specific

Compares the two double-precision, floating-point values of `a` and `b` for unordered.

```
__m128d _mm_cmpunord_pd (__m128d a, __m128d b);
CMPUNORDPD
```

Return Value

```
r0 := (a0 unord b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 unord b1) ? 0xffffffffffffffff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparisons

_mm_cmpneq_pd

Microsoft Specific

Compares the two double-precision, floating-point values of `a` and `b` for inequality.

```
__m128d _mm_cmpneq_pd (__m128d a, __m128d b);
CMPNEQPD
```

Return Value

```
r0 := (a0 != b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 != b1) ? 0xffffffffffffffff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparisons

_mm_cmpnlt_pd

Microsoft Specific

Compares the two double-precision, floating-point values of `a` and `b` for `a` not less than `b`.

```
__m128d _mm_cmpnlt_pd (__m128d a, __m128d b);
CMPNLTPD
```

Return Value

```
r0 := !(a0 < b0) ? 0xffffffffffffffff : 0x0
r1 := !(a1 < b1) ? 0xffffffffffffffff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparisons

_mm_cmpnle_pd

Microsoft Specific

Compares the two double-precision, floating-point values of `a` and `b` for `a` not less than or equal to `b`.

```
__m128d _mm_cmpnle_pd (__m128d a, __m128d b);
CMPNLEPD
```

Return Value

```
r0 := !(a0 <= b0) ? 0xffffffffffffffff : 0x0
r1 := !(a1 <= b1) ? 0xffffffffffffffff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparisons

_mm_cmpngt_pd

Microsoft Specific

Compares the two double-precision, floating-point values of `a` and `b` for `a` not greater than `b`.

```
__m128d _mm_cmpngt_pd (__m128d a, __m128d b);
CMPNLTPDr
```

Return Value

```
r0 := !(a0 > b0) ? 0xffffffffffffffff : 0x0
r1 := !(a1 > b1) ? 0xffffffffffffffff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparisons

_mm_cmpnge_pd

Microsoft Specific

Compares the two double-precision, floating-point values of `a` and `b` for `a` not greater than or equal to `b`.

```
__m128d _mm_cmpnge_pd (__m128d a, __m128d b);
CMPNLEPDr
```

Return Value

```
r0 := !(a0 >= b0) ? 0xffffffffffffffff : 0x0
r1 := !(a1 >= b1) ? 0xffffffffffffffff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparisons

_mm_cmpeq_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for equality.

```
__m128d _mm_cmpeq_sd (__m128d a, __m128d b);
CMPEQSD
```

Return Value

The upper double-precision, floating-point value is passed through from `a`.

```
r0 := (a0 == b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Comparisons

_mm_cmplt_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` less than `b`.

```
__m128d _mm_cmplt_sd (__m128d a, __m128d b);
CMPLTSD
```

Return Value

The upper double-precision, floating-point value is passed through from `a`.

```
r0 := (a0 < b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Comparisons

_mm_cmple_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` less than or equal to `b`.

```
__m128d _mm_cmple_sd (__m128d a, __m128d b);
CMPLESD
```

Return Value

The upper double-precision, floating-point value is passed through from `a`.

```
r0 := (a0 <= b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Comparisons

_mm_cmpgt_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` greater than `b`.

```
__m128d _mm_cmpgt_sd (__m128d a, __m128d b);
CMPLTSDr
```

Return Value

The upper double-precision, floating-point value is passed through from `a`.

```
r0 := (a0 > b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Comparisons

_mm_cmpge_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` greater than or equal to `b`.

```
__m128d _mm_cmpge_sd (__m128d a, __m128d b);
CMPLESDr
```

Return Value

The upper double-precision, floating-point value is passed through from `a`.

```
r0 := (a0 >= b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Comparisons

_mm_cmpord_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for ordered.

```
__m128d _mm_cmpord_sd (__m128d a, __m128d b);
CMPORDSD
```

Return Value

The upper double-precision, floating-point value is passed through from `a`.

```
r0 := (a0 ord b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Comparisons

_mm_cmpunord_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for unordered.

```
__m128d _mm_cmpunord_sd (__m128d a, __m128d b);
CMPUNORDSD
```

Return Value

The upper double-precision, floating-point value is passed through from `a`.

```
r0 := (a0 unord b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Comparisons

_mm_cmpneq_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for inequality.

```
__m128d _mm_cmpneq_sd (__m128d a, __m128d b);
CMPNEQSD
```

Return Value

The upper double-precision, floating-point value is passed through from `a`.

```
r0 := (a0 != b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparisons

_mm_cmpnlt_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` not less than `b`.

```
__m128d _mm_cmpnlt_sd (__m128d a, __m128d b);
CMPNLTSD
```

Return Value

The upper double-precision, floating-point value is passed through from `a`.

```
r0 := !(a0 < b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Comparisons

_mm_cmpnle_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` not less than or equal to `b`.

```
__m128d _mm_cmpnle_sd (__m128d a, __m128d b);
CMPNLESD
```

Return Value

The upper double-precision, floating-point value is passed through from `a`.

```
r0 := !(a0 <= b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparisons

_mm_cmpngt_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` not greater than `b`.

```
__m128d _mm_cmpngt_sd (__m128d a, __m128d b);
CMPNLTSDr
```

Return Value

The upper double-precision, floating-point value is passed through from `a`.

```
r0 := !(a0 > b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Comparisons

_mm_cmpnge_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` not greater than or equal to `b`.

```
__m128d _mm_cmpnge_sd (__m128d a, __m128d b);
CMPNLESDr
```

Return Value

The upper double-precision, floating-point value is passed through from `a`.

```
r0 := !(a0 >= b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparisons

_mm_comieq_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` equal to `b`.

```
int _mm_comieq_sd (__m128d a, __m128d b);
COMISD
```

Return Value

If `a` and `b` are equal, `1` is returned. Otherwise, `0` is returned. If `a` and `b` is a NaN, `1` is returned.

```
r := (a0 == b0) ? 0x1 : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Comparisons

_mm_comilt_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` less than `b`.

```
int _mm_comilt_sd (__m128d a, __m128d b);
COMISD
```

Return Value

If `a` is less than `b`, `1` is returned. Otherwise, `0` is returned. If `a` and `b` is a NaN, `1` is returned.

```
r := (a0 < b0) ? 0x1 : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparisons

_mm_comile_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` less than or equal to `b`.

```
int _mm_comile_sd (__m128d a, __m128d b);
COMISD
```

Return Value

If `a` is less than or equal to `b`, `1` is returned. Otherwise, `0` is returned. If `a` and `b` is a NaN, `1` is returned.

```
r := (a0 <= b0) ? 0x1 : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Comparisons

_mm_comigt_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` greater than `b`.

```
int _mm_comigt_sd (__m128d a, __m128d b);
COMISD
```

Return Value

If `a` is greater than `b` are equal, `1` is returned. Otherwise, `0` is returned. If `a` and `b` is a NaN, `0` is returned.

```
r := (a0 > b0) ? 0x1 : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Comparisons

_mm_comige_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` greater than or equal to `b`.

```
int _mm_comige_sd (__m128d a, __m128d b);
COMISD
```

Return Value

If `a` is greater than or equal to `b`, `1` is returned. Otherwise, `0` is returned. If `a` and `b` is a NaN, `0` is returned.

```
r := (a0 >= b0) ? 0x1 : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Comparisons

_mm_comineq_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` not equal to `b`.

```
int _mm_comineq_sd (__m128d a, __m128d b);
COMISD
```

Return Value

If `a` and `b` are not equal, `1` is returned. Otherwise, `0` is returned. If `a` and `b` is a NaN, `0` is returned.

```
r := (a0 != b0) ? 0x1 : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparisons

_mm_ucomieq_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` equal to `b`.

```
int _mm_ucomieq_sd ( __m128d a, __m128d b);
UCOMISD
```

Return Value

If `a` and `b` are equal, `1` is returned. Otherwise, `0` is returned. If `a` and `b` is a NaN, `1` is returned.

```
r := (a0 == b0) ? 0x1 : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Comparisons

_mm_ucomilt_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` less than `b`.

```
int _mm_ucomilt_sd (__m128d a, __m128d b);
UCOMISD
```

Return Value

If `a` is less than `b`, `1` is returned. Otherwise, `0` is returned. If `a` and `b` is a NaN, `1` is returned.

```
r := (a0 < b0) ? 0x1 : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparisons

_mm_ucomile_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` less than or equal to `b`.

```
int _mm_ucomile_sd (__m128d a, __m128d b);
UCOMISD
```

Return Value

If `a` is less than or equal to `b`, `1` is returned. Otherwise, `0` is returned. If `a` and `b` is a NaN, `1` is returned.

```
r := (a0 <= b0) ? 0x1 : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Comparisons

_mm_ucomigt_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` greater than `b`.

```
int _mm_ucomigt_sd (__m128d a, __m128d b);
UCOMISD
```

Return Value

If `a` is greater than `b` are equal, `1` is returned. Otherwise, `0` is returned. If `a` and `b` is a NaN, `0` is returned.

```
r := (a0 > b0) ? 0x1 : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Comparisons

_mm_ucomige_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` greater than or equal to `b`.

```
int _mm_ucomige_sd (__m128d a, __m128d b);
UCOMISD
```

Return Value

If `a` is greater than or equal to `b`, `1` is returned. Otherwise, `0` is returned. If `a` and `b` is a NaN, `0` is returned.

```
r := (a0 >= b0) ? 0x1 : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Comparisons

_mm_ucomineq_sd

Microsoft Specific

Compares the lower double-precision, floating-point value of `a` and `b` for `a` not equal to `b`.

```
int _mm_ucomineq_sd (__m128d a, __m128d b);
UCOMISD
```

Return Value

If `a` and `b` are not equal, `1` is returned. Otherwise, `0` is returned. If `a` and `b` is a NaN, `0` is returned.

```
r := (a0 != b0) ? 0x1 : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparisons

Conversion Operations (SSE2)

Microsoft Specific

Each conversion intrinsic takes one data type and performs a conversion to a different type. Some conversions such as **_mm_cvtpd_ps** result in a loss of precision. The rounding mode used in such cases is determined by the value in the MXCSR register. The default rounding mode is round-to-nearest. Note that the rounding mode used by the C and C++ languages when performing a type conversion is to truncate. The **_mm_cvttpd_epi32**, **_mm_cvttsd_si32**, and **_mm_cvttps_epi32** intrinsics use the truncate rounding mode regardless of the mode specified by the `MXCSR` register.

The conversion-operation intrinsics are listed in the following table, followed by detailed descriptions.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

The emmintrin.h header file contains the declarations for the SSE2 instructions intrinsics.

Conversion Operations

Intrinsic name	Corresponding instruction	Return type	Parameters
_mm_cvtpd_ps	CVTPD2PS	__m128	(__m128d a)
_mm_cvtps_pd	CVTPS2PD	__m128d	(__m128 a)
_mm_cvtepi32_pd	CVTDQ2PD	__m128d	(__m128i a)
_mm_cvtpd_epi32	CVTPD2DQ	__m128i	(__m128d a)
_mm_cvtsd_si32	CVTSD2SI	int	(__m128d a)
_mm_cvtsd_ss	CVTSD2SS	__m128	(__m128 a, __m128d b)
_mm_cvtsi32_sd	CVTSI2SD	__m128d	(__m128d a, int b)
_mm_cvtss_sd	CVTSS2SD	__m128d	(__m128d a, __m128 b)
_mm_cvttpd_epi32	CVTTPD2DQ	__m128i	(__m128d a)
_mm_cvttsd_si32	CVTTSD2SI	int	(__m128d a)
_mm_cvtepi32_ps	CVTDQ2PS	__m128	(__m128i a)
_mm_cvtps_epi32	CVTPS2DQ	__m128i	(__m128 a)
_mm_cvttps_epi32	CVTTPS2DQ	__m128i	(__m128 a)
_mm_cvtpd_pi32	CVTPD2PI	__m64	(__m128d a)
_mm_cvttpd_pi32	CVTTPD2PI	__m64	(__m128d a)
_mm_cvtpi32_pd	CVTPI2PD	__m128d	(__m64 a)

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

END Microsoft Specific

See Also

Reference

_mm_cvtpd_ps

Microsoft Specific

Converts the two double-precision, floating-point values of `a` to single-precision, floating-point values.

```
__m128 _mm_cvtpd_ps (__m128d a);
CVTPD2PS
```

Return Value

```
r0 := (float) a0
r1 := (float) a1
r2 := 0.0 ; r3 := 0.0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE2)

_mm_cvtps_pd

Microsoft Specific

Converts the lower two single-precision, floating-point values of `a` to double-precision, floating-point values.

```
__m128d _mm_cvtps_pd (__m128 a);
CVTPS2PD
```

Return Value

```
r0 := (double) a0
r1 := (double) a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE2)

_mm_cvtepi32_pd

Microsoft Specific

Converts the lower two signed 32-bit integer values of `a` to double-precision, floating-point values.

```
__m128d _mm_cvtepi32_pd (__m128i a);
CVTDQ2PD
```

Return Value

```
r0 := (double) a0
r1 := (double) a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE2)

_mm_cvtpd_epi32

Microsoft Specific

Converts the two double-precision, floating-point values of `a` to 32-bit signed integer values.

```
__m128i _mm_cvtpd_epi32 (__m128d a);
CVTPD2DQ
```

Return Value

```
r0 := (int) a0
r1 := (int) a1
r2 := 0x0 ; r3 := 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE2)

_mm_cvtsd_si32

Microsoft Specific

Converts the lower double-precision, floating-point value of `a` to a 32-bit signed integer value.

```
int _mm_cvtsd_si32 (__m128d a);
CVTSD2SI
```

Return Value

```
r := (int) a0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE2)

_mm_cvtsd_ss

Microsoft Specific

Converts the lower double-precision, floating-point value of `b` to an single-precision, floating-point value.

```
__m128 _mm_cvtsd_ss (__m128 a, __m128d b);
CVTSD2SS
```

Return Value

The upper single-precision, floating-point values in a are passed through.

```
r0 := (float) b0
r1 := a1; r2 := a2 ; r3 := a3
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE2)

_mm_cvtsi32_sd

Microsoft Specific

Converts the signed integer value in `b` to a double-precision, floating-point value.

```
__m128d _mm_cvtsi32_sd (__m128d a, int b);
CVTSI2SD
```

Return Value

The upper double-precision, floating-point value in a is passed through.

```
r0 := (double) b
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE2)

_mm_cvtss_sd

Microsoft Specific

Converts the lower single-precision, floating-point value of `b` to a double-precision, floating-point value.

```
__m128d _mm_cvtss_sd (__m128d a, __m128 b);
CVTSS2SD
```

Return Value

The upper value double-precision, floating-point value in `a` is passed through.

```
r0 := (double) b0
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Conversion Operations (SSE2)

_mm_cvttpd_epi32

Microsoft Specific

Converts the two double-precision, floating-point values of `a` to 32-bit signed integers using truncate.

```
__m128i _mm_cvttpd_epi32 (__m128d a);
CVTTPD2DQ
```

Return Value

```
r0 := (int) a0
r1 := (int) a1
r2 := 0x0 ; r3 := 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE2)

_mm_cvttsd_si32

Microsoft Specific

Converts the lower double-precision, floating-point value of `a` to a 32-bit signed integer using truncate.

```
int _mm_cvttsd_si32 (__m128d a);
CVTTSD2SI
```

Return Value

```
r := (int) a0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE2)

_mm_cvtepi32_ps

Microsoft Specific

Converts the four signed 32-bit integer values of a to single-precision, floating-point values.

```
__m128 _mm_cvtepi32_ps (__m128i a);
CVTDQ2PS
```

Return Value

```
r0 := (float) a0
r1 := (float) a1
r2 := (float) a2
r3 := (float) a3
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE2)

_mm_cvtps_epi32

Microsoft Specific

Converts the four single-precision, floating-point values of `a` to signed 32-bit integer values.

```
__m128i _mm_cvtps_epi32 (__m128 a);
CVTPS2DQ
```

Return Value

```
r0 := (int) a0
r1 := (int) a1
r2 := (int) a2
r3 := (int) a3
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE2)

_mm_cvttps_epi32

Microsoft Specific

Converts the four single-precision, floating-point values of `a` to signed 32-bit integer values using truncate.

```
__m128i _mm_cvttps_epi32 (__m128 a);
CVTTPS2DQ
```

Return Value

```
r0 := (int) a0
r1 := (int) a1
r2 := (int) a2
r3 := (int) a3
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE2)

_mm_cvtpd_pi32

Microsoft Specific

Converts the two double-precision, floating-point values of `a` to 32-bit signed integer values.

```
__m64 _mm_cvtpd_pi32 (__m128d a);
CVTPD2PI
```

Return Value

```
r0 := (int) a0
r1 := (int) a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE2)

_mm_cvttpd_pi32

Microsoft Specific

Converts the two double-precision, floating-point values of `a` to 32-bit signed integer values using truncate.

```
__m64 _mm_cvttpd_pi32 (__m128d a);
CVTTPD2PI
```

Return Value

```
r0 := (int) a0
r1 := (int) a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE2)

_mm_cvtpi32_pd

Microsoft Specific

Converts the two 32-bit signed integer values of `a` to double-precision, floating-point values.

```
__m128d _mm_cvtpi32_pd (__m64 a);
CVTPI2PD
```

Return Value

```
r0 := (double) a0
r1 := (double) a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Operations (SSE2)

Miscellaneous Operations (Floating-Point SSE2 Intrinsics)

Microsoft Specific

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

The emmintrin.h header file contains the declarations for the SSE2 instructions intrinsics.

```
__m128d _mm_unpackhi_pd (__m128d a, __m128d b);
```

```
UNPCKHPD
```

Interleaves the upper double-precision, floating-point values of `a` and `b`.

```
r0 := a1
r1 := b1
```

```
__m128d _mm_unpacklo_pd (__m128d a, __m128d b);
```

```
UNPCKLPD
```

Interleaves the lower double-precision, floating-point values of `a` and `b`.

```
r0 := a0
2 := b0
```

```
int _mm_movemask_pd (__m128d a);
```

```
MOVMSKPD
```

Creates a two-bit mask from the sign bits of the two double-precision, floating-point values of `a`.

```
r := sign(a1) << 1 | sign(a0)
```

```
__m128d _mm_shuffle_pd (__m128d a, __m128d b, int i);
```

```
SHUFPD
```

Selects two specific double-precision, floating-point values from `a` and `b`, based on the mask `i`. The mask must be an immediate. See Macro Function for Shuffle Using Streaming SIMD Extensions 2 Instructions section for a description of the shuffle semantics.

END Microsoft Specific

See Also

Reference

Floating-Point Intrinsics Using Streaming SIMD Extensions 2 Instructions

_mm_unpackhi_pd

Microsoft Specific

Interleaves the upper double-precision, floating-point values of `a` and `b`.

```
__m128d _mm_unpackhi_pd (__m128d a, __m128d b);
UNPCKHPD
```

Return Value

```
r0 := a1
r1 := b1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Floating-Point SSE2 Intrinsics)

_mm_unpacklo_pd

Microsoft Specific

Interleaves the lower double-precision, floating-point values of `a` and `b`.

```
__m128d _mm_unpacklo_pd (__m128d a, __m128d b);
UNPCKLPD
```

Return Value

```
r0 := a0
2 := b0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Floating-Point SSE2 Intrinsics)

_mm_movemask_pd

Microsoft Specific

Creates a two-bit mask from the sign bits of the two double-precision, floating-point values of `a`.

```
int _mm_movemask_pd (__m128d a);
MOVMSKPD
```

Return Value

```
r := sign(a1) << 1 | sign(a0)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Floating-Point SSE2 Intrinsics)

_mm_shuffle_pd

Microsoft Specific

Selects two specific double-precision, floating-point values from `a` and `b`, based on the mask `i`.

```
__m128d _mm_shuffle_pd (__m128d a, __m128d b, int i);
SHUFPD
```

Return Value

The mask must be an immediate. See Macro Function for Shuffle Using Streaming SIMD Extensions 2 Instructions section for a description of the shuffle semantics.

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Floating-Point SSE2 Intrinsics)

Floating-Point Memory and Initialization Operations Using Streaming SIMD Extensions 2

Microsoft Specific

This topic describes the load, set, and store operations, which let you load and store data into memory. The load and set operations are similar in that both initialize **__m128d** data. However, the set operations take a double argument and are intended for initialization with constants, while the load operations take a double pointer argument and are intended to mimic the instructions for loading data from memory. The store operation assigns the initialized data to the address.

- Load Operations
- Set Operations
- Store Operations

END Microsoft Specific
See Also
Reference
Streaming SIMD Extensions 2 Instructions

Load Operations (Floating-Point SSE2 Intrinsics)

Microsoft Specific

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

The emmintrin.h header file contains the declarations for the SSE2 instructions intrinsics.

```
__m128d _mm_load_pd (double *p);
```

```
MOVAPD
```

Loads two double-precision, floating-point values. The address `p` must be 16-byte aligned.

```
r0 := p[0]
r1 := p[1]
```

```
__m128d _mm_load1_pd (double *p);
```

```
(MOVSD + shuffling)
```

Loads a single double-precision, floating-point value, copying to both elements. The address `p` does not need to be 16-byte aligned.

```
r0 := *p
r1 := *p
```

```
__m128d _mm_loadr_pd (double *p);
```

```
(MOVAPD + shuffling)
```

Loads two double-precision, floating-point values in reverse order. The address `p` must be 16-byte aligned.

```
r0 := p[1]
r1 := p[0]
```

```
__m128d _mm_loadu_pd (double *p);
```

```
MOVUPD
```

Loads two double-precision, floating-point values. The address `p` does not need to be 16-byte aligned.

```
r0 := p[0]
r1 := p[1]
```

```
__m128d _mm_load_sd (double *p);
```

```
MOVSD
```

Loads a double-precision, floating-point value. The upper double-precision, floating-point is set to zero. The address `p` does not need to be 16-byte aligned.

```
r0 := *p
r1 := 0.0
```

`__m128d _mm_loadh_pd (__m128d a, double *p);`

```
MOVHPD
```

Loads a double-precision, floating-point value as the upper double-precision, floating-point value of the result. The lower double-precision, floating-point value is passed through from `a`. The address `p` does not need to be 16-byte aligned.

```
r0 := a0
r1 := *p
```

`__m128d _mm_loadl_pd (__m128d a, double *p);`

```
MOVLPD
```

Loads a double-precision, floating-point value as the lower double-precision, floating-point value of the result. The upper double-precision, floating-point value is passed through from `a`. The address `p` does not need to be 16-byte aligned.

```
r0 := *p
r1 := a1
```

END Microsoft Specific

See Also

Reference

Floating-Point Memory and Initialization Operations Using Streaming SIMD Extensions 2

_mm_load_pd

Microsoft Specific

Loads two double-precision, floating-point values.

```
__m128d _mm_load_pd (double *p);
MOVAPD
```

Return Value

The address `p` must be 16-byte aligned.

```
r0 := p[0]
r1 := p[1]
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Load Operations (Floating-Point SSE2 Intrinsics)

_mm_load1_pd

Microsoft Specific

Loads a single double-precision, floating-point value, copying to both elements.

```
__m128d _mm_load1_pd (double *p);
(MOVSD + shuffling)
```

Return Value

The address `p` does not need to be 16-byte aligned.

```
r0 := *p
r1 := *p
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Load Operations (Floating-Point SSE2 Intrinsics)

_mm_loadr_pd

Microsoft Specific

Loads two double-precision, floating-point values in reverse order.

```
__m128d _mm_loadr_pd (double *p);
(MOVAPD + shuffling)
```

Return Value

The address `p` must be 16-byte aligned.

```
r0 := p[1]
r1 := p[0]
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Load Operations (Floating-Point SSE2 Intrinsics)

_mm_loadu_pd

Microsoft Specific

Loads two double-precision, floating-point values.

```
__m128d _mm_loadu_pd (double *p);
MOVUPD
```

Return Value

The address `p` does not need to be 16-byte aligned.

```
r0 := p[0]
r1 := p[1]
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Load Operations (Floating-Point SSE2 Intrinsics)

_mm_load_sd

Microsoft Specific

Loads a double-precision, floating-point value.

```
__m128d _mm_load_sd (double *p);
MOVSD
```

Return Value

The upper double-precision, floating-point is set to zero. The address `p` does not need to be 16-byte aligned.

```
r0 := *p
r1 := 0.0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Load Operations (Floating-Point SSE2 Intrinsics)

_mm_loadh_pd

Microsoft Specific

Loads a double-precision, floating-point value as the upper double-precision, floating-point value of the result.

```
__m128d _mm_loadh_pd (__m128d a, double *p);
MOVHPD
```

Return Value

The lower double-precision, floating-point value is passed through from `a`. The address `p` does not need to be 16-byte aligned.

```
r0 := a0
r1 := *p
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Load Operations (Floating-Point SSE2 Intrinsics)

_mm_loadl_pd

Microsoft Specific

Loads a double-precision, floating-point value as the lower double-precision, floating-point value of the result.

```
__m128d _mm_loadl_pd (__m128d a, double *p);
MOVLPD
```

Return Value

The upper double-precision, floating-point value is passed through from `a`. The address `p` does not need to be 16-byte aligned.

```
r0 := *p
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Load Operations (Floating-Point SSE2 Intrinsics)

Set Operations (Floating-Point SSE2 Intrinsics)

Microsoft Specific

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

Intrinsic	Description
_mm_set_sd	Sets the lower double-precision, floating-point value to `w` and sets the upper double-precision, floating-point value to zero.
_mm_set1_pd	Sets the 2 double-precision, floating-point values to `w`.
_mm_set_pd	Sets the lower double-precision, floating-point value to `x` and sets the upper double-precision, floating-point value to `w`.
_mm_setr_pd	Sets the lower double-precision, floating-point value to `w` and sets the upper double-precision, floating-point value to `x`.
_mm_setzero_pd	Sets the 2 double-precision, floating-point values to zero.
_mm_move_sd	Sets the lower double-precision, floating-point value to the lower double-precision, floating-point value of `b`. The upper double-precision, floating-point value is passed through from `a`.

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Floating-Point Memory and Initialization Operations Using Streaming SIMD Extensions 2

_mm_set_sd

Microsoft Specific

Sets the lower double-precision, floating-point value to `w` and sets the upper double-precision, floating-point value to zero.

```
__m128d _mm_set_sd (double w);
(composite)
```

Return Value

```
r0 := w
r1 := 0.0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (Floating-Point SSE2 Intrinsics)

_mm_set1_pd

Microsoft Specific

Sets the 2 double-precision, floating-point values to `w`.

```
__m128d _mm_set1_pd (double w);
(composite)
```

Return Value

```
r0 := w
r1 := w
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (Floating-Point SSE2 Intrinsics)

_mm_set_pd

Microsoft Specific

Sets the lower double-precision, floating-point value to `x` and sets the upper double-precision, floating-point value to `w`.

```
__m128d _mm_set_pd (double w, double x);
(composite)
```

Return Value

```
r0 := x
r1 := w
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (Floating-Point SSE2 Intrinsics)

_mm_setr_pd

Microsoft Specific

Sets the lower double-precision, floating-point value to `w` and sets the upper double-precision, floating-point value to `x`.

```
__m128d _mm_setr_pd (double w, double x);
(composite)
```

Return Value

```
r0 := w
r1 := x
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (Floating-Point SSE2 Intrinsics)

_mm_setzero_pd

Microsoft Specific

Sets the 2 double-precision, floating-point values to zero.

```
__m128d _mm_setzero_pd ( );
XORPD
```

Return Value

```
r0 := 0.0
r1 := 0.0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (Floating-Point SSE2 Intrinsics)

_mm_move_sd

Microsoft Specific

Sets the lower double-precision, floating-point value to the lower double-precision, floating-point value of `b`.

```
__m128d _mm_move_sd (__m128d a, __m128d b);
MOVSD
```

Return Value

The upper double-precision, floating-point value is passed through from `a`.

```
r0 := b0
r1 := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (Floating-Point SSE2 Intrinsics)

Store Operations (Floating-Point SSE2 Intrinsics)

Microsoft Specific

For an explanation of the syntax used in code samples in this topic, see
Floating-Point Intrinsics Using Streaming SIMD Extensions.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

The emmintrin.h header file contains the declarations for the SSE2 instructions intrinsics.

`void _mm_store_sd (double *p, __m128d a);`

```
MOVSD
```

Stores the lower double-precision, floating-point value of `a`. The address `p` does not need to be 16-byte aligned.

```
*p := a0
<A NAME="vcref_mm_store1_pd"></A>void _mm_store1_pd (double *p, __m128d a);
(MOVAPD + shuffling)
```

Stores the lower double-precision, floating-point value of `a` twice. The address `p` must be 16-byte aligned.

```
p[0] := a0
p[1] := a0
```

`void _mm_store_pd (double *p, __m128d a);`

```
MOVAPD
```

Stores two double-precision, floating-point values. The address `p` must be 16-byte aligned.

```
p[0] := a0
p[1] := a1
```

`void _mm_storeu_pd (double *p, __m128d a);`

```
MOVUPD
```

Stores two double-precision, floating-point values. The address `p` does not need to be 16-byte aligned.

```
p[0] := a0
p[1] := a1
```

`void _mm_storer_pd (double *p, __m128d a);`

```
(MOVAPD + shuffling)
```

Stores two double-precision, floating-point values in reverse order. The address `p` must be 16-byte aligned.

```
p[0] := a1
p[1] := a0
```

`void _mm_storeh_pd (double *p, __m128d a);`

```
MOVHPD
```

Stores the upper double-precision, floating-point value of `a`.

```
*p := a1
```

`void _mm_storel_pd (double *p, __m128d a);`

```
MOVLPD
```

Stores the lower double-precision, floating-point value of `a`.

```
*p := a0
```

END Microsoft Specific

See Also

Reference

Floating-Point Memory and Initialization Operations Using Streaming SIMD Extensions 2

_mm_store_sd

Microsoft Specific

Stores the lower double-precision, floating-point value of `a`.

```
void _mm_store_sd (double *p, __m128d a);
MOVSD
```

Return Value

The address `p` does not need to be 16-byte aligned.

```
*p := a0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Store Operations (Floating-Point SSE2 Intrinsics)

_mm_store1_pd

Microsoft Specific

Stores the lower double-precision, floating-point value of `a` twice.

```
void _mm_store1_pd (double *p, __m128d a);
(MOVAPD + shuffling)
```

Return Value

The address `p` must be 16-byte aligned.

```
p[0] := a0
p[1] := a0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Store Operations (Floating-Point SSE2 Intrinsics)

_mm_store_pd

Microsoft Specific

Stores two double-precision, floating-point values.

```
void _mm_store_pd (double *p, __m128d a);
MOVAPD
```

Return Value

The address `p` must be 16-byte aligned.

```
p[0] := a0
p[1] := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Store Operations (Floating-Point SSE2 Intrinsics)

_mm_storeu_pd

Microsoft Specific

Stores two double-precision, floating-point values.

```
void _mm_storeu_pd (double *p, __m128d a);
MOVUPD
```

Return Value

The address `p` does not need to be 16-byte aligned.

```
p[0] := a0
p[1] := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Store Operations (Floating-Point SSE2 Intrinsics)

_mm_storer_pd

Microsoft Specific

Stores two double-precision, floating-point values in reverse order.

```
void _mm_storer_pd (double *p, __m128d a);
(MOVAPD + shuffling)
```

Return Value

The address `p` must be 16-byte aligned.

```
p[0] := a1
p[1] := a0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Store Operations (Floating-Point SSE2 Intrinsics)

_mm_storeh_pd

Microsoft Specific

Stores the upper double-precision, floating-point value of `a`.

```
void _mm_storeh_pd (double *p, __m128d a);
MOVHPD
```

Return Value

```
*p := a1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Store Operations (Floating-Point SSE2 Intrinsics)

_mm_storel_pd

Microsoft Specific

Stores the lower double-precision, floating-point value of `a`.

```
void _mm_storel_pd (double *p, __m128d a);
MOVLPD
```

Return Value

```
*p := a0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Store Operations (Floating-Point SSE2 Intrinsics)

Cache Support for Streaming SIMD Extensions 2 Floating-Point Operations

Microsoft Specific

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

The emmintrin.h header file contains the declarations for the SSE2 instructions intrinsics.

```
void _mm_stream_pd (double *p, __m128d a);
```

```
MOVLPD
```

Stores the data in `a` to the address p without polluting caches. The address `p` must be 16-byte aligned. If the cache line containing address `p` is already in the cache, the cache will be updated.

```
p[0] := a0
p[1] := a1
```

END Microsoft Specific

See Also

Reference

Streaming SIMD Extensions 2 Instructions

Integer Intrinsics Using Streaming SIMD Extensions 2

Microsoft Specific

The packed arithmetic intrinsics supporting the 128-bit integer MMX technology enhancements provided by the Streaming SIMD Extensions 2 (SSE2) instructions process are listed in the following tables: Arithmetic Operations

- Logical Operations
- Shift Operations
- Conversion Intrinsics
- Comparisons
- Miscellaneous Operations

END Microsoft Specific
See Also
Reference
Streaming SIMD Extensions 2 Instructions

Arithmetic Operations (Integer SSE2 Intrinsics)

Microsoft Specific

The operations are listed in the following table, followed by their descriptions. The packed arithmetic intrinsics for the 64-bit integer MMX technology are listed in the Packed Arithmetic Operations table.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

The emmintrin.h header file contains the declarations for the SSE2 instructions intrinsics.

Integer Arithmetic Operations

Intrinsic	Instruction	Operation
_mm_add_epi8	PADDB	Addition
_mm_add_epi16	PADDW	Addition
_mm_add_epi32	PADDD	Addition
_mm_add_si64	PADDQ	Addition
_mm_add_epi64	PADDQ	Addition
_mm_adds_epi8	PADDSB	Addition
_mm_adds_epi16	PADDSW	Addition
_mm_adds_epu8	PADDUSB	Addition
_mm_adds_epu16	PADDUSW	Addition
_mm_avg_epu8	PAVGB	Computes average
_mm_avg_epu16	PAVGW	Computes average
_mm_madd_epi16	PMADDWD	Multiplication/addition
_mm_max_epi16	PMAXSW	Computes maxima
_mm_max_epu8	PMAXUB	Computes maxima
_mm_min_epi16	PMINSW	Computes minima
_mm_min_epu8	PMINUB	Computes minima
_mm_mulhi_epi16	PMULHW	Multiplication
_mm_mulhi_epu16	PMULHUW	Multiplication
_mm_mullo_epi16	PMULLW	Multiplication
_mm_mul_su32	PMULUDQ	Multiplication

_mm_mul_epu32	PMULUDQ	Multiplication
_mm_sad_epu8	PSADBW	Computes difference/adds
_mm_sub_epi8	PSUBB	Subtraction
_mm_sub_epi16	PSUBW	Subtraction
_mm_sub_epi32	PSUBD	Subtraction
_mm_sub_si64	PSUBQ	Subtraction
_mm_sub_epi64	PSUBQ	Subtraction
_mm_subs_epi8	PSUBSB	Subtraction
_mm_subs_epi16	PSUBSW	Subtraction
_mm_subs_epu8	PSUBUSB	Subtraction
_mm_subs_epu16	PSUBUSW	Subtraction

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions 2

_mm_add_epi8

Microsoft Specific

Adds the 16 signed or unsigned 8-bit integers in `a` to the 16 signed or unsigned 8-bit integers in `b`.

```
__m128i _mm_add_epi8 (__m128i a ,__m128i b);
PADDB
```

Return Value

```
r0 := a0 + b0
r1 := a1 + b1
...
r15 := a15 + b15
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_add_epi16

Microsoft Specific

Adds the 8 signed or unsigned 16-bit integers in `a` to the 8 signed or unsigned 16-bit integers in `b`.

```
__m128i _mm_add_epi16 (__m128i a, __m128i b);
PADDW
```

Return Value

```
r0 := a0 + b0
r1 := a1 + b1
...
r7 := a7 + b7
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_add_epi32

Microsoft Specific

Adds the 4 signed or unsigned 32-bit integers in `a` to the 4 signed or unsigned 32-bit integers in `b`.

```
__m128i _mm_add_epi32 (__m128i a, __m128i b);
PADDD
```

Return Value

```
r0 := a0 + b0
r1 := a1 + b1
r2 := a2 + b2
r3 := a3 + b3
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_add_si64

Microsoft Specific

Adds the signed or unsigned 64-bit integer `a` to the signed or unsigned 64-bit integer `b`.

```
__m64 _mm_add_si64 (__m64 a, __m64 b);
PADDQ
```

Return Value

```
r := a + b
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_add_epi64

Microsoft Specific

Adds the 2 signed or unsigned 64-bit integers in `a` to the 2 signed or unsigned 64-bit integers in `b`.

```
__m128i _mm_add_epi64 (__m128i a, __m128i b);
PADDQ
```

Return Value

```
r0 := a0 + b0
r1 := a1 + b1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_adds_epi8

Microsoft Specific

Adds the 16 signed 8-bit integers in `a` to the 16 signed 8-bit integers in `b` and saturates.

```
__m128i _mm_adds_epi8 (__m128i a, __m128i b);
PADDSB
```

Return Value

```
r0 := SignedSaturate(a0 + b0)
r1 := SignedSaturate(a1 + b1)
...
r15 := SignedSaturate(a15 + b15)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_adds_epi16

Microsoft Specific

Adds the 8 signed 16-bit integers in `a` to the 8 signed 16-bit integers in `b` and saturates.

```
__m128i _mm_adds_epi16 (__m128i a, __m128i b);
PADDSW
```

Return Value

```
r0 := SignedSaturate(a0 + b0)
r1 := SignedSaturate(a1 + b1)
...
r7 := SignedSaturate(a7 + b7)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_adds_epu8

Microsoft Specific

Adds the 16 unsigned 8-bit integers in `a` to the 16 unsigned 8-bit integers in `b` and saturates.

```
__m128i _mm_adds_epu8 (__m128i a, __m128i b);
PADDUSB
```

Return Value

```
r0 := UnsignedSaturate(a0 + b0)
r1 := UnsignedSaturate(a1 + b1)
...
r15 := UnsignedSaturate(a15 + b15)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_adds_epu16

Microsoft Specific

Adds the 8 unsigned 16-bit integers in `a` to the 8 unsigned 16-bit integers in `b` and saturates.

```
__m128i _mm_adds_epu16 (__m128i a, __m128i b);
PADDUSW
```

Return Value

```
r0 := UnsignedSaturate(a0 + b0)
r1 := UnsignedSaturate(a1 + b1)
...
r15 := UnsignedSaturate(a7 + b7)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_avg_epu8

Microsoft Specific

Computes the average of the 16 unsigned 8-bit integers in `a` and the 16 unsigned 8-bit integers in `b` and rounds.

```
__m128i _mm_avg_epu8 (__m128i a, __m128i b);
PAVGB
```

Return Value

```
r0 := (a0 + b0) / 2
r1 := (a1 + b1) / 2
...
r15 := (a15 + b15) / 2
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_avg_epu16

Microsoft Specific

Computes the average of the 8 unsigned 16-bit integers in `a` and the 8 unsigned 16-bit integers in `b` and rounds.

```
__m128i _mm_avg_epu16 (__m128i a, __m128i b);
PAVGW
```

Return Value

```
r0 := (a0 + b0) / 2
r1 := (a1 + b1) / 2
...
r7 := (a7 + b7) / 2
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_madd_epi16

Microsoft Specific

Multiplies the 8 signed 16-bit integers from `a` by the 8 signed 16-bit integers from `b`.

```
__m128i _mm_madd_epi16 (__m128i a, __m128i b);
PMADDWD
```

Return Value

Adds the signed 32-bit integer results pairwise and packs the 4 signed 32-bit integer results.

```
r0 := (a0 * b0) + (a1 * b1)
r1 := (a2 * b2) + (a3 * b3)
r2 := (a4 * b4) + (a5 * b5)
r3 := (a6 * b6) + (a7 * b7)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_max_epi16

Microsoft Specific

Computes the pairwise maxima of the 8 signed 16-bit integers from `a` and the 8 signed 16-bit integers from `b`.

```
__m128i _mm_max_epi16 (__m128i a, __m128i b);
PMAXSW
```

Return Value

```
r0 := max(a0, b0)
r1 := max(a1, b1)
...
r7 := max(a7, b7)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_max_epu8

Microsoft Specific

Computes the pairwise maxima of the 16 unsigned 8-bit integers from `a` and the 16 unsigned 8-bit integers from `b`.

```
__m128i _mm_max_epu8 (__m128i a, __m128i b);
PMAXUB
```

Return Value

```
r0 := max(a0, b0)
r1 := max(a1, b1)
...
r15 := max(a15, b15)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_min_epi16

Microsoft Specific

Computes the pairwise minima of the 8 signed 16-bit integers from `a` and the 8 signed 16-bit integers from `b`.

```
__m128i _mm_min_epi16 (__m128i a, __m128i b);
PMINSW
```

Return Value

```
r0 := min(a0, b0)
r1 := min(a1, b1)
...
r7 := min(a7, b7)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_min_epu8

Microsoft Specific

Computes the pairwise minima of the 16 unsigned 8-bit integers from `a` and the 16 unsigned 8-bit integers from `b`.

```
__m128i _mm_min_epu8 (__m128i a, __m128i b);
PMINUB
```

Return Value

```
r0 := min(a0, b0)
r1 := min(a1, b1)
...
r15 := min(a15, b15)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_mulhi_epi16

Microsoft Specific

Multiplies the 8 signed 16-bit integers from `a` by the 8 signed 16-bit integers from `b`.

```
__m128i _mm_mulhi_epi16 (__m128i a, __m128i b);
PMULHW
```

Return Value

Packs the upper 16 bits of the 8 signed 32-bit results.

```
r0 := (a0 * b0)[31:16]
r1 := (a1 * b1)[31:16]
...
r7 := (a7 * b7)[31:16]
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_mulhi_epu16

Microsoft Specific

Multiplies the 8 unsigned 16-bit integers from `a` by the 8 unsigned 16-bit integers from `b`.

```
__m128i _mm_mulhi_epu16 (__m128i a, __m128i b);
PMULHUW
```

Return Value

Packs the upper 16 bits of the 8 unsigned 32-bit results.

```
r0 := (a0 * b0)[31:16]
r1 := (a1 * b1)[31:16]
...
r7 := (a7 * b7)[31:16]
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_mullo_epi16

Microsoft Specific

Multiplies the 8 signed or unsigned 16-bit integers from `a` by the 8 signed or unsigned 16-bit integers from `b`.

```
__m128i _mm_mullo_epi16 (__m128i a, __m128i b);
PMULLW
```

Return Value

Packs the lower 16 bits of the 8 signed or unsigned 32-bit results.

```
r0 := (a0 * b0)[15:0]
r1 := (a1 * b1)[15:0]
...
r7 := (a7 * b7)[15:0]
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_mul_su32

Microsoft Specific

Multiplies the lower 32-bit integer from `a` by the lower 32-bit integer from `b`, and returns the 64-bit integer result.

```
__m64 _mm_mul_su32 (__m64 a, __m64 b);
PMULUDQ
```

Return Value

```
r := a0 * b0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_mul_epu32

Microsoft Specific

Multiplies the lower 32-bit integer from `a` by the lower 32-bit integer from `b`, and returns the 64-bit integer result.

```
__m64 _mm_mul_su32 (__m64 a, __m64 b);
PMULUDQ
```

Return Value

```
r := a0 * b0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_sad_epu8

Microsoft Specific

Computes the absolute difference of the 16 unsigned 8-bit integers from `a` and the 16 unsigned 8-bit integers from `b`.

```
__m128i _mm_sad_epu8 (__m128i a, __m128i b);
PSADBW
```

Return Value

Sums the upper 8 differences and lower 8 differences and packs the resulting 2 unsigned 16-bit integers into the upper and lower 64-bit elements.

```
r0 := abs(a0 - b0) + abs(a1 - b1) +...+ abs(a7 - b7)
r1 := 0x0 ; r2 := 0x0 ; r3 := 0x0
r4 := abs(a8 - b8) + abs(a9 - b9) +...+ abs(a15 - b15)
r5 := 0x0 ; r6 := 0x0 ; r7 := 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_sub_epi8

Microsoft Specific

Subtracts the 16 signed or unsigned 8-bit integers of `b` from the 16 signed or unsigned 8-bit integers of `a`.

```
__m128i _mm_sub_epi8 (__m128i a, __m128i b);
PSUBB
```

Return Value

```
r0 := a0 - b0
r1 := a1 - b1
...
r15 := a15 - b15
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_sub_epi16

Microsoft Specific

Subtracts the 8 signed or unsigned 16-bit integers of `b` from the 8 signed or unsigned 16-bit integers of `a`.

```
__m128i _mm_sub_epi16 (__m128i a, __m128i b);
PSUBW
```

Return Value

```
r0 := a0 - b0
r1 := a1 - b1
...
r7 := a7 - b7
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_sub_epi32

Microsoft Specific

Subtracts the 4 signed or unsigned 32-bit integers of `b` from the 4 signed or unsigned 32-bit integers of `a`.

```
__m128i _mm_sub_epi32 (__m128i a, __m128i b);
PSUBD
```

Return Value

```
r0 := a0 - b0
r1 := a1 - b1
r2 := a2 - b2
r3 := a3 - b3
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_sub_si64

Microsoft Specific

Subtracts the signed or unsigned 64-bit integer `b` from the signed or unsigned 64-bit integer `a`.

```
__m64 _mm_sub_si64 (__m64 a, __m64 b);
PSUBQ
```

Return Value

```
r := a - b
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_sub_epi64

Microsoft Specific

Subtracts the 2 signed or unsigned 64-bit integers in `b` from the 2 signed or unsigned 64-bit integers in `a`.

```
__m128i _mm_sub_epi64 (__m128i a, __m128i b);
PSUBQ
```

Return Value

```
r0 := a0 - b0
r1 := a1 - b1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_subs_epi8

Microsoft Specific

Subtracts the 16 signed 8-bit integers of `b` from the 16 signed 8-bit integers of `a` and saturates.

```
__m128i _mm_subs_epi8 (__m128i a, __m128i b);
PSUBSB
```

Return Value

```
r0 := SignedSaturate(a0 - b0)
r1 := SignedSaturate(a1 - b1)
...
r15 := SignedSaturate(a15 - b15)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_subs_epi16

Microsoft Specific

Subtracts the 8 signed 16-bit integers of `b` from the 8 signed 16-bit integers of `a` and saturates.

```
__m128i _mm_subs_epi16 (__m128i a, __m128i b);
PSUBSW
```

Return Value

```
r0 := SignedSaturate(a0 - b0)
r1 := SignedSaturate(a1 - b1)
...
r7 := SignedSaturate(a7 - b7)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_subs_epu8

Microsoft Specific

Subtracts the 16 unsigned 8-bit integers of `b` from the 16 unsigned 8-bit integers of `a` and saturates.

```
__m128i _mm_subs_epu8 (__m128i a, __m128i b);
PSUBUSB
```

Return Value

```
r0 := UnsignedSaturate(a0 - b0)
r1 := UnsignedSaturate(a1 - b1)
...
r15 := UnsignedSaturate(a15 - b15)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

_mm_subs_epu16

Microsoft Specific

Subtracts the 8 unsigned 16-bit integers of `b` from the 8 unsigned 16-bit integers of `a` and saturates.

```
__m128i _mm_subs_epu16 (__m128i a, __m128i b);
PSUBUSW
```

Return Value

```
r0 := UnsignedSaturate(a0 - b0)
r1 := UnsignedSaturate(a1 - b1)
...
r7 := UnsignedSaturate(a7 - b7)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Arithmetic Operations (Integer SSE2 Intrinsics)

Logical Operations (Integer SSE2 Intrinsics)

Microsoft Specific

The following four logical operation intrinsics and their respective instructions are functional on Intel processors that support Streaming SIMD Extensions 2 (SSE2) instructions.

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

The emmintrin.h header file contains the declarations for the SSE2 instructions intrinsics.

```
__m128i _mm_and_si128 (__m128i a, __m128i b);
```

```
PAND
```

Computes the bitwise AND of the 128-bit value in a and the 128-bit value in b.

```
r := a & b
```

```
__m128i _mm_andnot_si128 (__m128i a, __m128i b);
```

```
PANDN
```

Computes the bitwise AND of the 128-bit value in b and the bitwise NOT of the 128-bit value in a.

```
r := (~a) & b
```

```
__m128i _mm_or_si128 (__m128i a, __m128i b);
```

```
POR
```

Computes the bitwise OR of the 128-bit value in a and the 128-bit value in b.

```
r := a | b
```

```
__m128i _mm_xor_si128 ( __m128i a, __m128i b);
```

```
PXOR
```

Computes the bitwise XOR of the 128-bit value in a and the 128-bit value in b.

```
r := a ^ b
```

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions 2

_mm_and_si128

Microsoft Specific

Computes the bitwise AND of the 128-bit value in a and the 128-bit value in b.

```
__m128i _mm_and_si128 (__m128i a, __m128i b);
PAND
```

Return Value

```
r := a & b
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Logical Operations (Integer SSE2 Intrinsics)

_mm_andnot_si128

Microsoft Specific

Computes the bitwise AND of the 128-bit value in b and the bitwise NOT of the 128-bit value in a.

```
__m128i _mm_andnot_si128 (__m128i a, __m128i b);
PANDN
```

Return Value

```
r := (~a) & b
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Logical Operations (Integer SSE2 Intrinsics)

_mm_or_si128

Microsoft Specific

Computes the bitwise OR of the 128-bit value in a and the 128-bit value in b.

```
__m128i _mm_or_si128 (__m128i a, __m128i b);
POR
```

Return Value

```
r := a | b
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Logical Operations (Integer SSE2 Intrinsics)

_mm_xor_si128

Microsoft Specific

Computes the bitwise XOR of the 128-bit value in a and the 128-bit value in b.

```
__m128i _mm_xor_si128 ( __m128i a, __m128i b);
PXOR
```

Return Value

```
r := a ^ b
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Logical Operations (Integer SSE2 Intrinsics)

Shift Operations

Microsoft Specific

The intrinsics listed in the following table are followed by their descriptions.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

The emmintrin.h header file contains the declarations for the SSE2 instructions intrinsics.

Shift Operation Intrinsics

Intrinsic shift	Direction shift	Type	Corresponding instruction
_mm_slli_si128	Left	Logical	PSLLDQ
_mm_slli_epi16	Left	Logical	PSLLW
_mm_sll_epi16	Left	Logical	PSLLW
_mm_slli_epi32	Left	Logical	PSLLD
_mm_sll_epi32	Left	Logical	PSLLD
_mm_slli_epi64	Left	Logical	PSLLQ
_mm_sll_epi64	Left	Logical	PSLLQ
_mm_srai_epi16	Right	Arithmetic	PSRAW
_mm_sra_epi16	Right	Arithmetic	PSRAW
_mm_srai_epi32	Right	Arithmetic	PSRAD
_mm_sra_epi32	Right	Arithmetic	PSRAD
_mm_srli_si128	Right	Logical	PSRLDQ
_mm_srli_epi16	Right	Logical	PSRLW
_mm_srl_epi16	Right	Logical	PSRLW
_mm_srli_epi32	Right	Logical	PSRLD
_mm_srl_epi32	Right	Logical	PSRLD
_mm_srli_epi64	Right	Logical	PSRLQ
_mm_srl_epi64	Right	Logical	PSRLQ

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions 2

_mm_slli_si128

Microsoft Specific

Shifts the 128-bit value in `a` left by `imm` bytes while shifting in zeros. `imm` must be an immediate.

```
__m128i _mm_slli_si128 (__m128i a, int imm);
PSLLDQ
```

Return Value

```
r := a << (imm * 8)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Shift Operations

_mm_slli_epi16

Microsoft Specific

Shifts the 8 signed or unsigned 16-bit integers in `a` left by `count` bits while shifting in zeros.

```
__m128i _mm_slli_epi16 (__m128i a, int count);
PSLLW
```

Return Value

```
r0 := a0 << count
r1 := a1 << count
...
r7 := a7 << count
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Shift Operations

_mm_sll_epi16

Microsoft Specific

Shifts the 8 signed or unsigned 16-bit integers in `a` left by `count` bits while shifting in zeros.

```
__m128i _mm_sll_epi16 (__m128i a, __m128i count);
PSLLW
```

Return Value

```
r0 := a0 << count
r1 := a1 << count
...
r7 := a7 << count
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Shift Operations

_mm_slli_epi32

Microsoft Specific

Shifts the 4 signed or unsigned 32-bit integers in `a` left by `count` bits while shifting in zeros.

```
__m128i _mm_slli_epi32 (__m128i a, int count);
PSLLD
```

Return Value

```
r0 := a0 << count
r1 := a1 << count
r2 := a2 << count
r3 := a3 << count
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Shift Operations

_mm_sll_epi32

Microsoft Specific

Shifts the 4 signed or unsigned 32-bit integers in `a` left by `count` bits while shifting in zeros.

```
__m128i _mm_sll_epi32 (__m128i a, __m128i count);
PSLLD
```

Return Value

```
r0 := a0 << count
r1 := a1 << count
r2 := a2 << count
r3 := a3 << count
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Shift Operations

_mm_slli_epi64

Microsoft Specific

Shifts the 2 signed or unsigned 64-bit integers in `a` left by `count` bits while shifting in zeros.

```
__m128i _mm_slli_epi64 (__m128i a, int count);
PSLLQ
```

Return Value

```
r0 := a0 << count
r1 := a1 << count
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Shift Operations

_mm_sll_epi64

Microsoft Specific

Shifts the 2 signed or unsigned 64-bit integers in `a` left by `count` bits while shifting in zeros.

```
__m128i _mm_sll_epi64 (__m128i a, __m128i count);
PSLLQ
```

Return Value

```
r0 := a0 << count
r1 := a1 << count
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Shift Operations

_mm_srai_epi16

Microsoft Specific

Shifts the 8 signed 16-bit integers in `a` right by `count` bits while shifting in the sign bit.

```
__m128i _mm_srai_epi16 (__m128i a, int count);
PSRAW
```

Return Value

```
r0 := a0 >> count
r1 := a1 >> count
...
r7 := a7 >> count
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Shift Operations

_mm_sra_epi16

Microsoft Specific

Shifts the 8 signed 16-bit integers in `a` right by `count` bits while shifting in the sign bit.

```
__m128i _mm_sra_epi16 (__m128i a, __m128i count);
PSRAW
```

Return Value

```
r0 := a0 >> count
r1 := a1 >> count
...
r7 := a7 >> count
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Shift Operations

_mm_srai_epi32

Microsoft Specific

Shifts the 4 signed 32-bit integers in `a` right by `count` bits while shifting in the sign bit.

```
__m128i _mm_srai_epi32 (__m128i a, int count);
PSRAD
```

Return Value

```
r0 := a0 >> count
r1 := a1 >> count
r2 := a2 >> count
r3 := a3 >> count
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Shift Operations

_mm_sra_epi32

Microsoft Specific

Shifts the 4 signed 32-bit integers in `a` right by `count` bits while shifting in the sign bit.

```
__m128i _mm_sra_epi32 (__m128i a, __m128i count);
PSRAD
```

Return Value

```
r0 := a0 >> count
r1 := a1 >> count
r2 := a2 >> count
r3 := a3 >> count
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Shift Operations

_mm_srli_si128

Microsoft Specific

Shifts the 128-bit value in `a` right by `imm` bytes while shifting in zeros. `imm` must be an immediate.

```
__m128i _mm_srli_si128 (__m128i a, int imm);
PSRLDQ
```

Return Value

```
r := srl(a, imm*8)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Shift Operations

_mm_srli_epi16

Microsoft Specific

Shifts the 8 signed or unsigned 16-bit integers in `a` right by `count` bits while shifting in zeros.

```
__m128i _mm_srli_epi16 (__m128i a, int count);
PSRLW
```

Return Value

```
r0 := srl(a0, count)
r1 := srl(a1, count)
...
r7 := srl(a7, count)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Shift Operations

_mm_srl_epi16

Microsoft Specific

Shifts the 8 signed or unsigned 16-bit integers in `a` right by `count` bits while shifting in zeros.

```
__m128i _mm_srl_epi16 (__m128i a, __m128i count);
PSRLW
```

Return Value

```
r0 := srl(a0, count)
r1 := srl(a1, count)
...
r7 := srl(a7, count)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Shift Operations

_mm_srli_epi32

Microsoft Specific

Shifts the 4 signed or unsigned 32-bit integers in `a` right by `count` bits while shifting in zeros.

```
__m128i _mm_srli_epi32 (__m128i a, int count);
PSRLD
```

Return Value

```
r0 := srl(a0, count)
r1 := srl(a1, count)
r2 := srl(a2, count)
r3 := srl(a3, count)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Shift Operations

_mm_srl_epi32

Microsoft Specific

Shifts the 4 signed or unsigned 32-bit integers in `a` right by `count` bits while shifting in zeros.

```
__m128i _mm_srl_epi32 (__m128i a, __m128i count);
PSRLD
```

Return Value

```
r0 := srl(a0, count)
r1 := srl(a1, count)
r2 := srl(a2, count)
r3 := srl(a3, count)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Shift Operations

_mm_srli_epi64

Microsoft Specific

Shifts the 2 signed or unsigned 64-bit integers in `a` right by `count` bits while shifting in zeros.

```
__m128i _mm_srli_epi64 (__m128i a, int count);
PSRLQ
```

Return Value

```
r0 := srl(a0, count)
r1 := srl(a1, count)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Shift Operations

_mm_srl_epi64

Microsoft Specific

Shifts the 2 signed or unsigned 64-bit integers in `a` right by `count` bits while shifting in zeros.

```
__m128i _mm_srl_epi64 (__m128i a, __m128i count);
PSRLQ
```

Return Value

```
r0 := srl(a0, count)
r1 := srl(a1, count)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Shift Operations

Conversion Intrinsics

Microsoft Specific

The following two conversion intrinsics and their respective instructions are functional on Intel processors supporting Streaming SIMD Extensions 2 (SSE2) instructions.

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

The emmintrin.h header file contains the declarations for the SSE2 instructions intrinsics.

```
__m128i _mm_cvtsi32_si128 (int a);
```

```
MOVD
```

Moves 32-bit integer `a` to the least significant 32 bits of an `__m128` object one extending the upper bits.

```
r0 := a
r1 := 0x0 ; r2 := 0x0 ; r3 := 0x0
```

```
int _mm_cvtsi128_si32 (__m128i a);
```

```
MOVD
```

Moves the least significant 32 bits of `a` to a 32-bit integer.

```
r := a0
```

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions 2

_mm_cvtsi32_si128

Microsoft Specific

Moves 32-bit integer `a` to the least significant 32 bits of an `__m128` object one extending the upper bits.

```
__m128i _mm_cvtsi32_si128 (int a);
MOVD
```

Return Value

```
r0 := a
r1 := 0x0 ; r2 := 0x0 ; r3 := 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Intrinsics

_mm_cvtsi128_si32

Microsoft Specific

Moves the least significant 32 bits of `a` to a 32-bit integer.

```
int _mm_cvtsi128_si32 (__m128i a);
MOVD
```

Return Value

```
r := a0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Conversion Intrinsics

Comparison Intrinsics (SSE2)

Microsoft Specific

The intrinsics listed in the following table are followed by their descriptions.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

The emmintrin.h header file contains the declarations for the SSE2 instructions intrinsics.

Comparison Intrinsics

Intrinsic name	Instruction	Comparison	Elements	Size of elements
_mm_cmpeq_epi8	PCMPEQB	Equality	16	8
_mm_cmpeq_epi16	PCMPEQW	Equality	8	16
_mm_cmpeq_epi32	PCMPEQD	Equality	4	32
_mm_cmpgt_epi8	PCMPGTB	Greater than	16	8
_mm_cmpgt_epi16	PCMPGTW	Greater than	8	16
_mm_cmpgt_epi32	PCMPGTD	Greater than	4	32
_mm_cmplt_epi8	PCMPGTBr	Less than	16	8
_mm_cmplt_epi16	PCMPGTWr	Less than	8	16
_mm_cmplt_epi32	PCMPGTDr	Less than	4	32

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions 2

_mm_cmpeq_epi8

Microsoft Specific

Compares the 16 signed or unsigned 8-bit integers in `a` and the 16 signed or unsigned 8-bit integers in `b` for equality.

```
__m128i _mm_cmpeq_epi8 (__m128i a, __m128i b);
PCMPEQB
```

Return Value

```
r0 := (a0 == b0) ? 0xff : 0x0
r1 := (a1 == b1) ? 0xff : 0x0
...
r15 := (a15 == b15) ? 0xff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE2)

_mm_cmpeq_epi16

Microsoft Specific

Compares the 8 signed or unsigned 16-bit integers in `a` and the 8 signed or unsigned 16-bit integers in `b` for equality.

```
__m128i _mm_cmpeq_epi16 (__m128i a, __m128i b);
PCMPEQW
```

Return Value

```
r0 := (a0 == b0) ? 0xffff : 0x0
r1 := (a1 == b1) ? 0xffff : 0x0
...
r7 := (a7 == b7) ? 0xffff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE2)

_mm_cmpeq_epi32

Microsoft Specific

Compares the 4 signed or unsigned 32-bit integers in `a` and the 4 signed or unsigned 32-bit integers in `b` for equality.

```
__m128i _mm_cmpeq_epi32 (__m128i a, __m128i b);
PCMPEQD
```

Return Value

```
r0 := (a0 == b0) ? 0xffffffff : 0x0
r1 := (a1 == b1) ? 0xffffffff : 0x0
r2 := (a2 == b2) ? 0xffffffff : 0x0
r3 := (a3 == b3) ? 0xffffffff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE2)

_mm_cmpgt_epi8

Microsoft Specific

Compares the 16 signed 8-bit integers in `a` and the 16 signed 8-bit integers in `b` for greater than.

```
__m128i _mm_cmpgt_epi8 (__m128i a, __m128i b);
PCMPGTB
```

Return Value

```
r0 := (a0 > b0) ? 0xff : 0x0
r1 := (a1 > b1) ? 0xff : 0x0
...
r15 := (a15 > b15) ? 0xff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE2)

_mm_cmpgt_epi16

Microsoft Specific

Compares the 8 signed 16-bit integers in `a` and the 8 signed 16-bit integers in `b` for greater than.

```
__m128i _mm_cmpgt_epi16 (__m128i a, __m128i b);
PCMPGTW
```

Return Value

```
r0 := (a0 > b0) ? 0xffff : 0x0
r1 := (a1 > b1) ? 0xffff : 0x0
...
r7 := (a7 > b7) ? 0xffff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE2)

_mm_cmpgt_epi32

Microsoft Specific

Compares the 4 signed 32-bit integers in `a` and the 4 signed 32-bit integers in `b` for greater than.

```
__m128i _mm_cmpgt_epi32 (__m128i a, __m128i b);
PCMPGTD
```

Return Value

```
r0 := (a0 > b0) ? 0xffff : 0x0
r1 := (a1 > b1) ? 0xffff : 0x0
r2 := (a2 > b2) ? 0xffff : 0x0
r3 := (a3 > b3) ? 0xffff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE2)

_mm_cmplt_epi8

Microsoft Specific

Compares the 16 signed 8-bit integers in `a` and the 16 signed 8-bit integers in `b` for less than.

```
__m128i _mm_cmplt_epi8 (__m128i a, __m128i b);
PCMPGTBr
```

Return Value

```
r0 := (a0 < b0) ? 0xff : 0x0
r1 := (a1 < b1) ? 0xff : 0x0
...
r15 := (a15 < b15) ? 0xff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE2)

_mm_cmplt_epi16

Microsoft Specific

Compares the 8 signed 16-bit integers in `a` and the 8 signed 16-bit integers in `b` for less than.

```
__m128i _mm_cmplt_epi16 (__m128i a, __m128i b);
PCMPGTWr
```

Return Value

```
r0 := (a0 < b0) ? 0xffff : 0x0
r1 := (a1 < b1) ? 0xffff : 0x0
...
r7 := (a7 < b7) ? 0xffff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE2)

_mm_cmplt_epi32

Microsoft Specific

Compares the 4 signed 32-bit integers in `a` and the 4 signed 32-bit integers in `b` for less than.

```
__m128i _mm_cmplt_epi32 (__m128i a, __m128i b);
PCMPGTDr
```

Return Value

```
r0 := (a0 < b0) ? 0xffff : 0x0
r1 := (a1 < b1) ? 0xffff : 0x0
r2 := (a2 < b2) ? 0xffff : 0x0
r3 := (a3 < b3) ? 0xffff : 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Comparison Intrinsics (SSE2)

Miscellaneous Operations (Integer SSE2 Intrinsics)

Microsoft Specific

The intrinsics listed in the following table are followed by their descriptions.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

The emmintrin.h header file contains the declarations for the SSE2 instructions intrinsics.

Comparison Intrinsics

Intrinsic	Corresponding instruction	Operation
_mm_packs_epi16	PACKSSWB	Packed saturation
_mm_packs_epi32	PACKSSDW	Packed saturation
_mm_packus_epi16	PACKUSWB	Packed saturation
_mm_extract_epi16	PEXTRW	Extraction
_mm_insert_epi16	PINSRW	Insertion
_mm_movemask_epi8	PMOVMSKB	Mask creation
_mm_shuffle_epi32	PSHUFD	Shuffle
_mm_shufflehi_epi16	PSHUFHW	Shuffle
_mm_shufflelo_epi16	PSHUFLW	Shuffle
_mm_unpackhi_epi8	PUNPCKHBW	Interleave
_mm_unpackhi_epi16	PUNPCKHWD	Interleave
_mm_unpackhi_epi32	PUNPCKHDQ	Interleave
_mm_unpackhi_epi64	PUNPCKHQDQ	Interleave
_mm_unpacklo_epi8	PUNPCKLBW	Interleave
_mm_unpacklo_epi16	PUNPCKLWD	Interleave
_mm_unpacklo_epi32	PUNPCKLDQ	Interleave
_mm_unpacklo_epi64	PUNPCKLQDQ	Interleave
_mm_movepi64_pi64	MOVDQ2Q	Move
_mm_movpi64_epi64	MOVQ2DQ	Move
_mm_move_epi64	MOVQ	Move

For an explanation of the syntax used in code samples in this topic, see
Floating-Point Intrinsics Using Streaming SIMD Extensions.

END Microsoft Specific

See Also

Reference

Integer Intrinsics Using Streaming SIMD Extensions 2

_mm_packs_epi16

Microsoft Specific

Packs the 16 signed 16-bit integers from `a` and `b` into 8-bit integers and saturates.

```
__m128i _mm_packs_epi16 (__m128i a, __m128i b);
PACKSSWB
```

Return Value

```
r0 := SignedSaturate(a0)
r1 := SignedSaturate(a1)
...
r7 := SignedSaturate(a7)
r8 := SignedSaturate(b0)
r9 := SignedSaturate(b1)
...
r15 := SignedSaturate(b7)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Integer SSE2 Intrinsics)

_mm_packs_epi32

Microsoft Specific

Packs the 8 signed 32-bit integers from `a` and `b` into signed 16-bit integers and saturates.

```
__m128i _mm_packs_epi32 (__m128i a, __m128i b);
PACKSSDW
```

Return Value

```
r0 := SignedSaturate(a0)
r1 := SignedSaturate(a1)
r2 := SignedSaturate(a2)
r3 := SignedSaturate(a3)
r4 := SignedSaturate(b0)
r5 := SignedSaturate(b1)
r6 := SignedSaturate(b2)
r7 := SignedSaturate(b3)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Integer SSE2 Intrinsics)

_mm_packus_epi16

Microsoft Specific

Packs the 16 signed 16-bit integers from `a` and `b` into 8-bit unsigned integers and saturates.

```
__m128i _mm_packus_epi16 (__m128i a, __m128i b);
PACKUSWB
```

Return Value

```
r0 := UnsignedSaturate(a0)
r1 := UnsignedSaturate(a1)
...
r7 := UnsignedSaturate(a7)
r8 := UnsignedSaturate(b0)
r9 := UnsignedSaturate(b1)
...
r15 := UnsignedSaturate(b7)
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Integer SSE2 Intrinsics)

_mm_extract_epi16

Microsoft Specific

Extracts the selected signed or unsigned 16-bit integer from `a` and zero extends.

```
int _mm_extract_epi16 (__m128i a, int imm);
PEXTRW
```

Return Value

The selector `imm` must be an immediate.

```
r := (imm == 0) ? a0 :
( (imm == 1) ? a1 :
...
(imm == 7) ? a7 )
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Integer SSE2 Intrinsics)

_mm_insert_epi16

Microsoft Specific

Inserts the least significant 16 bits of `b` into the selected 16-bit integer of `a`

```
__m128i _mm_insert_epi16 (__m128i a, int b, int imm);
PINSRW
```

Return Value

The selector `imm` must be an immediate.

```
r0 := (imm == 0) ? b : a0;
r1 := (imm == 1) ? b : a1;
...
r7 := (imm == 7) ? b : a7;
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also
Reference
Miscellaneous Operations (Integer SSE2 Intrinsics)

_mm_movemask_epi8

Microsoft Specific

Creates a 16-bit mask from the most significant bits of the 16 signed or unsigned 8-bit integers in `a` and zero extends the upper bits.

```
int _mm_movemask_epi8 (__m128i a);
PMOVMSKB
```

Return Value

```
r := a15[7] << 15 |
a14[7] << 14 |
...
a1[7] << 1 |
a0[7]
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Integer SSE2 Intrinsics)

_mm_shuffle_epi32

Microsoft Specific

Shuffles the 4 signed or unsigned 32-bit integers in `a` as specified by `imm`.

```
__m128i _mm_shuffle_epi32 (__m128i a, int imm);
PSHUFD
```

Return Value

The shuffle value, `imm` , must be an immediate. See Macro Function for Shuffle Using Streaming SIMD Extensions 2 for a description of shuffle semantics.

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Integer SSE2 Intrinsics)

_mm_shufflehi_epi16

Microsoft Specific

Shuffles the upper 4 signed or unsigned 16-bit integers in `a` as specified by `imm`.

```
__m128i _mm_shufflehi_epi16 (__m128i a, int imm);
PSHUFHW
```

Return Value

The shuffle value, `imm`, must be an immediate. See Macro Function for Shuffle Using Streaming SIMD Extensions 2 for a description of shuffle semantics.

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Integer SSE2 Intrinsics)

_mm_shufflelo_epi16

Microsoft Specific

Shuffles the lower 4 signed or unsigned 16-bit integers in `a` as specified by `imm`.

```
__m128i _mm_shufflelo_epi16 (__m128i a, int imm);
PSHUFLW
```

Return Value

The shuffle value, `imm`, must be an immediate. See Macro Function for Shuffle Using Streaming SIMD Extensions 2 for a description of shuffle semantics.

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Integer SSE2 Intrinsics)

_mm_unpackhi_epi8

Microsoft Specific

Interleaves the upper 8 signed or unsigned 8-bit integers in `a` with the upper 8 signed or unsigned 8-bit integers in `b`.

```
__m128i _mm_unpackhi_epi8 (__m128i a, __m128i b);
PUNPCKHBW
```

Return Value

```
r0 := a8 ; r1 := b8
r2 := a9 ; r3 := b9
...
r14 := a15 ; r15 := b15
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Integer SSE2 Intrinsics)

_mm_unpackhi_epi16

Microsoft Specific

Interleaves the upper 4 signed or unsigned 16-bit integers in `a` with the upper 4 signed or unsigned 16-bit integers in `b`.

```
__m128i _mm_unpackhi_epi16 (__m128i a, __m128i b);
PUNPCKHWD
```

Return Value

```
r0 := a4 ; r1 := b4
r2 := a5 ; r3 := b5
r4 := a6 ; r5 := b6
r6 := a7 ; r7 := b7
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Integer SSE2 Intrinsics)

_mm_unpackhi_epi32

Microsoft Specific

Interleaves the upper 2 signed or unsigned 32-bit integers in `a` with the upper 2 signed or unsigned 32-bit integers in `b`.

```
__m128i _mm_unpackhi_epi32 (__m128i a, __m128i b);
PUNPCKHDQ
```

Return Value

```
r0 := a2 ; r1 := b2
r2 := a3 ; r3 := b3
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Integer SSE2 Intrinsics)

_mm_unpackhi_epi64

Microsoft Specific

Interleaves the upper signed or unsigned 64-bit integer in `a` with the upper signed or unsigned 64-bit integer in `b`.

```
__m128i _mm_unpackhi_epi64 (__m128i a, __m128i b);
PUNPCKHQDQ
```

Return Value

```
r0 := a1 ; r1 := b1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Integer SSE2 Intrinsics)

_mm_unpacklo_epi8

Microsoft Specific

Interleaves the lower 8 signed or unsigned 8-bit integers in `a` with the lower 8 signed or unsigned 8-bit integers in `b`.

```
__m128i _mm_unpacklo_epi8 (__m128i a, __m128i b);
PUNPCKLBW
```

Return Value

```
r0 := a0 ; r1 := b0
r2 := a1 ; r3 := b1
...
r14 := a7 ; r15 := b7
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Integer SSE2 Intrinsics)

_mm_unpacklo_epi16

Microsoft Specific

Interleaves the lower 4 signed or unsigned 16-bit integers in `a` with the lower 4 signed or unsigned 16-bit integers in `b`.

```
__m128i _mm_unpacklo_epi16 (__m128i a, __m128i b);
PUNPCKLWD
```

Return Value

```
r0 := a0 ; r1 := b0
r2 := a1 ; r3 := b1
r4 := a2 ; r5 := b2
r6 := a3 ; r7 := b3
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Integer SSE2 Intrinsics)

_mm_unpacklo_epi32

Microsoft Specific

Interleaves the lower 2 signed or unsigned 32-bit integers in `a` with the lower 2 signed or unsigned 32-bit integers in `b`.

```
__m128i _mm_unpacklo_epi32 (__m128i a, __m128i b);
PUNPCKLDQ
```

Return Value

```
r0 := a0 ; r1 := b0
r2 := a1 ; r3 := b1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Integer SSE2 Intrinsics)

_mm_unpacklo_epi64

Microsoft Specific

Interleaves the lower signed or unsigned 64-bit integer in `a` with the lower signed or unsigned 64-bit integer in `b`.

```
__m128i _mm_unpacklo_epi64 (__m128i a, __m128i b);
PUNPCKLQDQ
```

Return Value

```
r0 := a0 ; r1 := b0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Integer SSE2 Intrinsics)

_mm_movepi64_pi64

Microsoft Specific

Returns the lower 64 bits of `a` as an `__m64` type.

```
__m64 _mm_movepi64_pi64 (__m128i a);
MOVDQ2Q
```

Return Value

```
r0 := a0 ;
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Integer SSE2 Intrinsics)

_mm_movpi64_epi64

Microsoft Specific

Moves the 64 bits of `a` to the lower 64 bits of the result, zeroing the upper bits.

```
__m128i _mm_movpi64_epi64 (__m64 a);
MOVDQ2Q
```

Return Value

```
r0 := a0 ; r1 := 0X0 ;
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Integer SSE2 Intrinsics)

_mm_move_epi64

Microsoft Specific

Moves the lower 64 bits of the lower 64 bits of the result, zeroing the upper bits.

```
__m128i _mm_move_epi64 (__m128i a);
MOVQ
```

Return Value

r0 := a0 ; r1 := 0X0 ;

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Miscellaneous Operations (Integer SSE2 Intrinsics)

Integer Memory and Initialization Using Streaming SIMD Extensions 2

Microsoft Specific

This topic describes the load, set, and store operations, which let you load and store data to memory. The load and set operations are similar in that both initialize **__m128i** data. However, the set operations take various sizes of integer as arguments and are intended for initialization with constants, and the load operations take pointers as arguments to aligned or unaligned **__m128i** data and are intended for loading data from memory. The store operations also take pointers as arguments and are intended for storing data to memory.

- Load Operations
- Set Operations
- Store Operations

END Microsoft Specific
See Also
Reference
Streaming SIMD Extensions 2 Instructions

Load Operations (Integer SSE2 Intrinsics)

Microsoft Specific

The load operation intrinsics and their respective instructions are functional on Intel processors supporting Streaming SIMD Extensions 2 (SSE2) instructions.

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

The emmintrin.h header file contains the declarations for the SSE2 instructions intrinsics.

```
__m128i _mm_load_si128 (__m128i *p);
```

```
MOVDQA
```

Loads 128-bit value. Address `p` must be 16-byte aligned.

```
r := *p
```

```
__m128i _mm_loadu_si128 (__m128i *p);
```

```
MOVDQU
```

Loads 128-bit value. Address `p` does not need be 16-byte aligned.

```
r := *p
```

```
__m128i _mm_loadl_epi64(__m128i const*p);
```

```
MOVQ
```

Load the lower 64 bits of the value pointed to by p into the lower 64 bits of the result, zeroing the upper 64 bits of the result.

```
r0:= *p[63:0]
r1:=0x0
```

END Microsoft Specific

See Also

Reference

Integer Memory and Initialization Using Streaming SIMD Extensions 2

_mm_load_si128

Microsoft Specific

Loads 128-bit value.

```
__m128i _mm_load_si128 (__m128i *p);
MOVDQA
```

Return Value

Returns the value loaded in a variable representing a register.

Remarks

Address `p` must be 16-byte aligned. For the unaligned version of this intrinsic, see _mm_loadu_si128.

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Load Operations (Integer SSE2 Intrinsics)

_mm_loadu_si128

Microsoft Specific

Loads 128-bit value.

```
__m128i _mm_loadu_si128 (__m128i *p);
MOVDQU
```

Return Value

The same value in a variable representing a register.

Remarks

Address `p` does not need be 16-byte aligned. The aligned version of this intrinsic is _mm_load_si128.

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Load Operations (Integer SSE2 Intrinsics)

_mm_loadl_epi64

Microsoft Specific

Load the lower 64 bits of the value pointed to by p into the lower 64 bits of the result, zeroing the upper 64 bits of the result.

```
__m128i _mm_loadl_epi64(__m128i const*p);
MOVQ
```

Return Value

```
r0:= *p[63:0]
r1:=0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Load Operations (Integer SSE2 Intrinsics)

Set Operations (Integer SSE2 Intrinsics)

Microsoft Specific

The intrinsics are listed in the following table, followed by their descriptions.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

The emmintrin.h header file contains the declarations for the SSE2 instructions intrinsics.

Set Operation Intrinsics

Intrinsic	Corresponding instruction
_mm_set_epi64	Composite
_mm_set_epi32	Composite
_mm_set_epi16	Composite
_mm_set_epi8	Composite
_mm_set1_epi64	Composite
_mm_set1_epi32	Composite
_mm_set1_epi16	Composite
_mm_set1_epi8	Composite
_mm_setr_epi64	Composite
_mm_setr_epi32	Composite
_mm_setr_epi16	Composite
_mm_setr_epi8	Composite
_mm_setzero_si128	PXOR

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

END Microsoft Specific

See Also

Reference

Integer Memory and Initialization Using Streaming SIMD Extensions 2

_mm_set_epi64

Microsoft Specific

Sets the 2 64-bit integer values.

```
__m128i _mm_set_epi64 (__m64 q1, __m64 q0);
(composite)
```

Return Value

```
r0 := q0
r1 := q1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (Integer SSE2 Intrinsics)

_mm_set_epi32

Microsoft Specific

Sets the 4 signed 32-bit integer values.

```
__m128i _mm_set_epi32 (int i3, int i2,    int i1, int i0);
(composite)
```

Return Value

```
r0 := i0
r1 := i1
r2 := i2
r3 := i3
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (Integer SSE2 Intrinsics)

_mm_set_epi16

Microsoft Specific

Sets the 8 signed 16-bit integer values.

```
__m128i _mm_set_epi16 (short w7, short w6,    short w5, short w4,   short w3, short w2,   s
hort w1, short w0);
(composite)
```

Return Value

```
r0 := w0
r1 := w1
...
r7 := w7
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (Integer SSE2 Intrinsics)

_mm_set_epi8

Microsoft Specific

Sets the 16 signed 8-bit integer values.

```
__m128i _mm_set_epi8 (char b15, char b14,    char b13, char b12,   char b11, char b10,   ch
ar b9, char b8,   char b7, char b6,   char b5, char b4,   char b3, char b2,   char b1, char
b0);
(composite)
```

Return Value

```
r0 := b0
r1 := b1
...
r15 := b15
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (Integer SSE2 Intrinsics)

_mm_set1_epi64

Microsoft Specific

Sets the 2 64-bit integer values to `q`.

```
__m128i _mm_set1_epi64 (__m64 q);
(composite)
```

Return Value

```
r0 := q
r1 := q
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (Integer SSE2 Intrinsics)

_mm_set1_epi32

Microsoft Specific

Sets the 4 signed 32-bit integer values to `i`.

```
__m128i _mm_set1_epi32 (int i);
(composite)
```

Return Value

```
r0 := i
r1 := i
r2 := i
r3 := I
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (Integer SSE2 Intrinsics)

_mm_set1_epi16

Microsoft Specific

Sets the 8 signed 16-bit integer values to `w`.

```
__m128i _mm_set1_epi16 (short w);
(composite)
```

Return Value

```
r0 := w
r1 := w
...
r7 := w
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (Integer SSE2 Intrinsics)

_mm_set1_epi8

Microsoft Specific

Sets the 16 signed 8-bit integer values to `b`.

```
__m128i _mm_set1_epi8 (char b);
(composite)
```

Return Value

```
r0 := b
r1 := b
...
r15 := b
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (Integer SSE2 Intrinsics)

_mm_setr_epi64

Microsoft Specific

Sets the 2 64-bit integer values in reverse order.

```
__m128i _mm_setr_epi64 (__m64 q0, __m64 q1);
(composite)
```

Return Value

```
r0 := q0
r1 := q1
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (Integer SSE2 Intrinsics)

_mm_setr_epi32

Microsoft Specific

Sets the 4 signed 32-bit integer values in reverse order.

```
__m128i _mm_setr_epi32 (int i0, int i1,    int i2, int i3);
(composite)
```

Return Value

```
r0 := i0
r1 := i1
r2 := i2
r3 := i3
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (Integer SSE2 Intrinsics)

_mm_setr_epi16

Microsoft Specific

Sets the 8 signed 16-bit integer values in reverse order.

```
__m128i _mm_setr_epi16 (short w0, short w1,    short w2, short w3,   short w4, short w5,
short w6, short w7);
(composite)
```

Return Value

```
r0 := w0
r1 := w1
...
r7 := w7
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (Integer SSE2 Intrinsics)

_mm_setr_epi8

Microsoft Specific

Sets the 16 signed 8-bit integer values in reverse order.

```
__m128i _mm_setr_epi8 (char b0, char b1,    char b2, char b3,   char b4, char b5,   char b6
, char b7,   char b8, char b9,   char b10, char b11,   char b12, char b13,   char b14, char
b15);
(composite)
```

Return Value

```
r0 := b0
r1 := b1
...
r15 := b15
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (Integer SSE2 Intrinsics)

_mm_setzero_si128

Microsoft Specific

Sets the 128-bit value to zero.

```
__m128i _mm_setzero_si128 ();
PXOR
```

Return Value

```
r := 0x0
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Set Operations (Integer SSE2 Intrinsics)

Store Operations (Integer SSE2 Intrinsics)

Microsoft Specific

The following store operation intrinsics and their respective instructions are functional on Intel processors supporting Streaming SIMD Extensions 2 (SSE2) instructions.

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

The emmintrin.h header file contains the declarations for the SSE2 instructions intrinsics.

```
void _mm_store_si128 (__m128i *p, __m128i a);
```

MOVDQA

Stores 128-bit value. Address `p` must be 16-byte aligned.

```
*p := a
```

For more information, see _mm_store_si128.

```
void _mm_storeu_si128 (__m128i *p, __m128i a);
```

MOVDQU

Stores 128-bit value. Address `p` does not need to be 16-byte aligned.

```
*p := a
```

For more information, see _mm_storeu_si128.

```
void _mm_maskmoveu_si128(__m128i d, __m128i n, char *p);
```

MASKMOVDQU

Conditionally store byte elements of `d` to address `p`. The high bit of each byte in the selector `n` determines whether the corresponding byte in `d` will be stored. Address `p` does not need to be 16-byte aligned.

```
if (n0[7]) p[0] := d0
if (n1[7]) p[1] := d1
...
if (n15[7]) p[15] := d15
```

For more information, see _mm_maskmoveu_si128.

```
void _mm_storel_epi64(__m128i *p, __m128i a);
```

MOVQ

Stores the lower 64 bits of the value pointed to by `p`.

```
*p[63:0]:=a0
```

END Microsoft Specific

See Also

Reference

Integer Memory and Initialization Using Streaming SIMD Extensions 2

_mm_store_si128

Microsoft Specific

Stores 128-bit value.

```
void _mm_store_si128 (__m128i *p, __m128i a);
MOVDQA
```

Return Value

Address `p` must be 16-byte aligned.

```
*p := a
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Store Operations (Integer SSE2 Intrinsics)

_mm_storeu_si128

Microsoft Specific

Stores 128-bit value.

```
void _mm_storeu_si128 (__m128i *p, __m128i a);
MOVDQU
```

Return Value

Address `p` does not need to be 16-byte aligned.

```
*p := a
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Store Operations (Integer SSE2 Intrinsics)

_mm_maskmoveu_si128

Microsoft Specific

Conditionally store byte elements of `d` to address `p`.

```
void _mm_maskmoveu_si128(__m128i d, __m128i n, char *p);
MASKMOVDQU
```

Return Value

The high bit of each byte in the selector `n` determines whether the corresponding byte in `d` will be stored. Address `p` does not need to be 16-byte aligned.

```
if (n0[7]) p[0] := d0
if (n1[7]) p[1] := d1
...
if (n15[7]) p[15] := d15
```

Requirements

Header: emmintrin.h

END Microsoft Specific

See Also

Reference

Store Operations (Integer SSE2 Intrinsics)

_mm_storel_epi64

Microsoft Specific

Reads the lower 64 bits of *b* and stores them into the lower 64 bits of *a*.

```
void _mm_storel_epi64(
   __m128i* a
   __m128i b
);
```

Parameters

[in] a

An **__m128i** structure in memory to write to.

[in] b

An **__m128i** structure (mapping to an XMM hardware register) to read from.

Requirements

Intrinsic	Architecture
_mm_storel_epi64	x86 with SSE2 support, x64

Remarks

This intrinsic is useful for moving data from an MMX register into system memory.

This routine is only available as an intrinsic.

Example

```
// _mm_storel_epi64.cpp
// processor: x86
#include <emmintrin.h>
#include <stdio.h>
extern "C"
{
   void _mm_storel_epi64(__m128i* a, __m128i b);
}
#pragma intrinsic(_mm_storel_epi64, _mm_empty, _mm_set1_epi64)

int main()
{

   __m128i m, n;
   __m64 q;

   // q is in memory
   q.m64_i64 = 100;

   // m is stored in the XMM register
   m = _mm_set1_epi64(q);

   // write to n in memory.
   _mm_storel_epi64(&n, m);

   // Return to floating point (x86 only)
#ifdef _M_IX86
   _mm_empty();
#endif

   // The first element is 100
   // The second parameter is garbage
   printf("%I64d %I64d\n", n.m128i_i64[0], n.m128i_i64[1]);
```

```
}
```

Sample Output

```
100 8639981687544439014
```

END Microsoft Specific

See Also

Reference

__m128i

Compiler Intrinsics

Store Operations (Integer SSE2 Intrinsics)

Cache Support for Streaming SIMD Extensions 2 Integer Operations

Microsoft Specific

For an explanation of the syntax used in code samples in this topic, see Floating-Point Intrinsics Using Streaming SIMD Extensions.

SSE2 intrinsics use the __m128, __m128i, and __m128d data types, which are not supported on Itanium Processor Family (IPF) processors. Any SSE2 intrinsics that use the __m64 data type are not supported on x64 processors.

The emmintrin.h header file contains the declarations for the SSE2 instructions intrinsics.

`void _mm_stream_si128(__m128i *p, __m128i a)`

Stores the data in `a` to the address `p` without polluting the caches. If the cache line containing address `p` is already in the cache, the cache will be updated. Address `p` must be 16-byte aligned.

```
*p := a
```

`void _mm_stream_si32(int *p, int a)`

Stores the data in `a` to the address `p` without polluting the caches. If the cache line containing address `p` is already in the cache, the cache will be updated.

```
*p := a
```

`void _mm_clflush(void const*p)`

Cache line containing `p` is flushed and invalidated from all caches in the coherency domain.

`void _mm_lfence(void)`

Guarantees that every load instruction that precedes, in program order, the load fence instruction is globally visible before any load instruction that follows the fence in program order.

`void _mm_mfence(void)`

Guarantees that every memory access that precedes, in program order, the memory fence instruction is globally visible before any memory instruction that follows the fence in program order.

`void _mm_pause(void)`

The execution of the next instruction is delayed an implementation specific amount of time. The instruction does not modify the architectural state. This intrinsic provides especially significant performance gain and described in more detail in the following section.

END Microsoft Specific

See Also

Reference

Streaming SIMD Extensions 2 Instructions

Macro Function for Shuffle Using Streaming SIMD Extensions 2

Microsoft Specific

The Streaming SIMD Extensions 2 (SSE2) instructions provide a macro function to help create constants that describe shuffle operations. The macro takes two small integers (in the range of 0 to 1) and combines them into a 2-bit immediate value used by the **SHUFPD** instruction. See the following example.

Shuffle Function Macro

```
_MM_SHUFFLE2(x, y)
/* expands to the value of */
(x<<1) | y
```

You can view the two integers as selectors for choosing which two words from the first input operand and which two words from the second are to be put into the result word.

View of Original and Result Words with Shuffle Function Macro

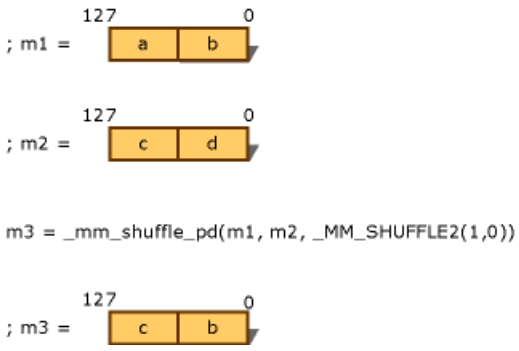

END Microsoft Specific

See Also

Reference

Streaming SIMD Extensions 2 Instructions

Microsoft Macro Assembler Reference

The Microsoft Macro Assembler (MASM) provides you with several advantages over inline assembly. MASM contains a macro language with looping, arithmetic, text string processing, and so on, and MASM supports the instruction sets of the 386, 486, and Pentium processors, providing you with greater direct control over the hardware. You also can avoid extra time and memory overhead when using MASM.

In This Section

ML and ML64 Command-Line Option

Describes the ML and ML64 command-line options.

ML Error Messages

Describes ML fatal and nonfatal error messages and warnings.

Directives Reference

Provides links to topics discussing the use of directives in MASM.

Symbols Reference

Provides links to topics discussing the use of symbols in MASM.

Operators Reference

Provides links to topics discussing the use of operators in MASM.

MASM for x64 (ml64.exe)

Information on creating output files for x64.

Related Sections

MASM Samples

Provides links to samples that demonstrate support for MASM.

Visual C++

Provides links to different areas of the Visual Studio and Visual C++ documentation set.

Reference

Provides links to topics describing the C and C++ language references, the libraries provided with Visual C++, and the Visual C++ Extensibility Object Model.

MASM for x64 (ml64.exe)

ml64.exe is the assembler that accepts x64 assembly language. For information on ml64.exe compiler options, see ML and ML64 Command-Line Reference.

Inline ASM is not supported for x64. Use MASM or compiler intrinsics (x64 Intrinsics).

The two workarounds are separate assembly with MASM (which supports x64 fully) and compiler intrinsics. We've added a lot of intrinsics to allow customers to make use of special-function instructions (e.g. privileged, bit scan/test, interlocked, etc...) in as close to cross-platform a manner as possible.

ml64-Specific Directives

Use the following directives with ml64.exe:

- .ALLOCSTACK
- .ENDPROLOG
- .PUSHFRAME
- .PUSHREG
- .SAVEREG
- .SAVEXMM128
- .SETFRAME

In addition, the PROC directive was updated for use with ml64.exe.

32-Bit Address Mode (Address Size Override)

MASM will emit the 0x67 address size override if a memory operand includes 32-bit registers. For example, the following examples cause the address size override to be emitted:

```
mov rax, QWORD PTR [ecx]
mov eax, DWORD PTR [ecx*2+r10d]
mov eax, DWORD PTR [ecx*2+r10d+0100h]
prefetch [eax]
movnti rax, QWORD PTR [r8d]
```

MASM assumes that if a 32-bit displacement appears alone as a memory operand, 64-bit addressing is intended. There is currently no support for 32-bit addressing with such operands.

Finally, mixing register sizes within a memory operand, as demonstrated in the following code, will generate an error.

```
mov eax, DWORD PTR [rcx*2+r10d]
mov eax, DWORD PTR [ecx*2+r10+0100h]
```

See Also

Other Resources

Microsoft Macro Assembler Reference

ML and ML64 Command-Line Reference

Assembles and links one or more assembly-language source files. The command-line options are case sensitive.

For more information on ml64.exe, see MASM for x64 (ml64.exe).

```
ML [[options]] filename [[ [[options]]  filename]]
ML64 [[options]] filename [[ [[options]]  filename]]
...
[[/link linkoptions]]
```

Parameters

options

The options listed in the following table.

Option	Action
/AT	Enables tiny-memory-model support. Enables error messages for code constructs that violate the requirements for .com format files. Note that this is not equivalent to the .MODEL **TINY** directive. Not available in ml64.exe.
/Bl *filename*	Selects an alternate linker.
/c	Assembles only. Does not link.
/coff	Generates common object file format (COFF) type of object module. Generally required for Win32 assembly language development. Not available in ml64.exe.
/Cp	Preserves case of all user identifiers.
/Cu	Maps all identifiers to upper case (default). Not available in ml64.exe.
/Cx	Preserves case in public and extern symbols.
/D *symbol*[[=*value*]]	Defines a text macro with the given name. If *value* is missing, it is blank. Multiple tokens separated by spaces must be enclosed in quotation marks.
/EP	Generates a preprocessed source listing (sent to STDOUT). See **/Sf**.
/ERRORREPORT [**NONE** | **PROMPT** | **QUEUE** | **SEND**]	If ml.exe or ml64.exe fails at runtime, you can use **/ERRORREPORT** to send information to Microsoft about these internal errors. For more information about /ERRORREPORT, see /errorReport (Report Internal Compiler Errors).
/F *hexnum*	Sets stack size to *hexnum* bytes (this is the same as **/link /STACK**:*number*). The value must be expressed in hexadecimal notation. There must be a space between **/F** and *hexnum*.
/Fe *filename*	Names the executable file.
/Fl[[*filename*]]	Generates an assembled code listing. See **/Sf**.

/Fm[[*filename*]]	Creates a linker map file.
/Fo *filename*	Names an object file. See Remarks section for more information.
/FPi	Generates emulator fix-ups for floating-point arithmetic (mixed language only). Not available in ml64.exe.
/Fr[[*filename*]]	Generates a source browser .sbr file.
/FR[[*filename*]]	Generates an extended form of a source browser .sbr file.
/Gc	Specifies use of FORTRAN- or Pascal-style function calling and naming conventions. Same as **OPTION LANGUAGE:PASCAL**. Not available in ml64.exe.
/Gd	Specifies use of C-style function calling and naming conventions. Same as **OPTION LANGUAGE:C**. Not available in ml64.exe.
/GZ	Specifies use of __stdcall function calling and naming conventions. Same as **OPTION LANGUAGE:STCALL**. Not available in ml64.exe.
/H *number*	Restricts external names to number significant characters. The default is 31 characters. Not available in ml64.exe.
/help	Calls QuickHelp for help on ML.
/I *pathname*	Sets path for include file. A maximum of 10 **/I** options is allowed.
/nologo	Suppresses messages for successful assembly.
/omf	Generates object module file format (OMF) type of object module. **/omf** implies **/c**; ML.exe does not support linking OMF objects. Not available in ml64.exe.
/Sa	Turns on listing of all available information.
/safeseh	Marks the object as either containing no exception handlers or containing exception handlers that are all declared with .SAFESEH. Not available in ml64.exe.
/Sf	Adds first-pass listing to listing file.
/Sl *width*	Sets the line width of source listing in characters per line. Range is 60 to 255 or 0. Default is 0. Same as PAGE width.
/Sn	Turns off symbol table when producing a listing.
/Sp *length*	Sets the page length of source listing in lines per page. Range is 10 to 255 or 0. Default is 0. Same as PAGE length.

/Ss *text*	Specifies text for source listing. Same as SUBTITLE text.
/St *text*	Specifies title for source listing. Same as TITLE text.
/Sx	Turns on false conditionals in listing.
/Ta *filename*	Assembles source file whose name does not end with the .asm extension.
/w	Same as **/W0 /WX**.
/W *level*	Sets the warning level, where *level* = 0, 1, 2, or 3.
/WX	Returns an error code if warnings are generated.
/X	Ignore INCLUDE environment path.
/Zd	Generates line-number information in object file.
/Zf	Makes all symbols public.
/Zi	Generates CodeView information in object file.
/Zm	Enables **M510** option for maximum compatibility with MASM 5.1. Not available in ml64.exe.
/Zp[[*alignment*]]	Packs structures on the specified byte boundary. The *alignment* can be 1, 2, or 4.
/Zs	Performs a syntax check only.
/?	Displays a summary of ML command-line syntax.

filename

The name of the file.

linkoptions

The link options. See Linker Options for more information.

Remarks

Some command-line options to ML and ML64 are placement-sensitive. For example, because ML and ML64 can accept several **/c** options, any corresponding **/Fo** options must be specified before **/c**. The following command-line example illustrates an object file specification for each assembly file specification:

ml.exe /Fo a1.obj /c a.asm /Fo b1.obj /c b.asm

Environment Variables

Variable	Description
INCLUDE	Specifies search path for include files.
ML	Specifies default command-line options.
TMP	Specifies path for temporary files.

See Also

Reference

ML Error Messages

Other Resources

Directives Reference

x64

.ALLOCSTACK	.ENDPROLOG	PROC
.PUSHFRAME	.PUSHREG	.SAVEREG
.SAVEXMM128	.SETFRAME	

Code Labels

ALIGN	EVEN
LABEL	ORG

Conditional Assembly

ELSE	ELSEIF	ELSEIF2
IF	IF2	IFB/IFNB
IFDEF/IFNDEF	IFDIF/IFDIF[[I]]	IFE
IFIDN/IFIDN[[I]]		

Conditional Control Flow

.BREAK	.CONTINUE	.ELSE
.ELSEIF	.ENDIF	.ENDW
.IF	.REPEAT	.UNTIL
.UNTILCXZ	.WHILE	

Conditional Error

.ERR	.ERR2	.ERRB
.ERRDEF	.ERRDIF/.ERRDIF[[I]]]	.ERRE
.ERRIDN/.ERRIDN[[I]]	.ERRNB	.ERRNDEF
.ERRNZ		

Data Allocation

ALIGN	BYTE/SBYTE	DWORD/SDWORD
EVEN	FWORD	LABEL
ORG	QWORD	REAL4
REAL8	REAL10	TBYTE
WORD/SWORD		

Equates

=		
EQU		
TEXTEQU		

Listing Control

.CREF	.LIST	.LISTALL
.LISTIF	.LISTMACRO	.LISTMACROALL
.NOCREF	.NOLIST	.NOLISTIF
.NOLISTMACRO	PAGE	SUBTITLE
.TFCOND	TITLE	

Macros

ENDM	EXITM	GOTO
LOCAL	MACRO	PURGE

Miscellaneous

ALIAS (MASM)	ASSUME	COMMENT
ECHO	END	.FPO
INCLUDE	INCLUDELIB	MMWORD
OPTION	POPCONTEXT	PUSHCONTEXT
.RADIX	.SAFESEH	XMMWORD

Procedures

ENDP	INVOKE	PROC
PROTO		

Processor

.186	.286	.286P
.287	.386	.386P
.387	.486	.486P
.586	.586P	.686
.686P	.K3D	.MMX
.XMM	.8086	.8087
.NO87		

Repeat Blocks

ENDM	FOR	FORC
GOTO	REPEAT	WHILE

Scope

COMM	EXTERN	EXTERNDEF
INCLUDELIB	PUBLIC	

Segment

.ALPHA	ASSUME	.DOSSEG
END	ENDS	GROUP
SEGMENT	.SEQ	

Simplified Segment

.CODE	.CONST	.DATA
.DATA?	.DOSSEG	.EXIT
.FARDATA	.FARDATA?	.MODEL
.STACK	.STARTUP	

String

CATSTR	INSTR
SIZESTR	SUBSTR

Structure and Record

ENDS	RECORD	STRUCT
TYPEDEF	UNION	

See Also

Other Resources

Microsoft Macro Assembler Reference

=

Assigns the numeric value of *expression* to *name*.

```
name = expression
```

Remarks

The symbol can be redefined later.

See Also

Other Resources

Directives Reference

.186

Enables assembly of instructions for the 80186 processor; disables assembly of instructions introduced with later processors.

```
.186
```

Remarks

Also enables 8087 instructions.

See Also

Other Resources

Directives Reference

.286

Enables assembly of nonprivileged instructions for the 80286 processor; disables assembly of instructions introduced with later processors.

```
.286
```

Remarks

Also enables 80287 instructions.

See Also

Other Resources

Directives Reference

.286P

Enables assembly of all instructions (including privileged) for the 80286 processor; disables assembly of instructions introduced with later processors.

```
.286P
```

Remarks

Also enables 80287 instructions.

See Also

Other Resources

Directives Reference

.287

Enables assembly of instructions for the 80287 coprocessor; disables assembly of instructions introduced with later coprocessors.

```
.287
```

See Also

Other Resources

Directives Reference

.386

Enables assembly of nonprivileged instructions for the 80386 processor; disables assembly of instructions introduced with later processors.

```
.386
```

Remarks

Also enables 80387 instructions.

See Also

Other Resources

Directives Reference

.386P

Enables assembly of all instructions (including privileged) for the 80386 processor; disables assembly of instructions introduced with later processors.

```
.386P
```

Remarks

Also enables 80387 instructions.

See Also

Other Resources

Directives Reference

.387

Enables assembly of instructions for the 80387 coprocessor.

```
.387
```

See Also

Other Resources

Directives Reference

.486

Enables assembly of nonprivileged instructions for the 80486 processor.

```
.486
```

See Also

Other Resources

Directives Reference

.486P

Enables assembly of all instructions (including privileged) for the 80486 processor.

```
.486P
```

See Also

Other Resources

Directives Reference

.586

Enables assembly of nonprivileged instructions for the Pentium processor.

```
.586P
```

See Also

Other Resources

Directives Reference

.586P

Enables assembly of all instructions (including privileged) for the Pentium processor.

```
.586P
```

See Also

Other Resources

Directives Reference

.686

Enables assembly of nonprivileged instructions for the Pentium Pro processor.

```
.686
```

See Also

Other Resources

Directives Reference

.686P

Enables assembly of all instructions (including privileged) for the Pentium Pro processor.

```
.686P
```

See Also

Other Resources

Directives Reference

.8086

Enables assembly of 8086 instructions (and the identical 8088 instructions); disables assembly of instructions introduced with later processors.

```
.8086
```

Remarks

Also enables 8087 instructions. This is the default mode for processors.

See Also

Other Resources

Directives Reference

.8087

Enables assembly of 8087 instructions; disables assembly of instructions introduced with later coprocessors.

```
.8087
```

Remarks

This is the default mode for coprocessors.

See Also

Other Resources

Directives Reference

ALIAS (MASM)

The **ALIAS** directive creates an alternate name for a function. This lets you create multiple names for a function, or create libraries that allow the linker (LINK.exe) to map an old function to a new function.

```
ALIAS  <alias> = <actual-name>
```

Parameters

actual-name

The actual name of the function or procedure. The angle brackets are required.

alias

The alternate or alias name. The angle brackets are required.

See Also

Other Resources

Directives Reference

ALIGN (MASM)

Aligns the next variable or instruction on a byte that is a multiple of *number*.

```
ALIGN [[number]]
```

See Also

Other Resources

Directives Reference

.ALLOCSTACK

Generates a **UWOP_ALLOC_SMALL** or a **UWOP_ALLOC_LARGE** with the specified size for the current offset in the prologue.

```
.ALLOCSTACK size
```

Remarks

MASM will choose the most efficient encoding for a given size.

.ALLOCSTACK allows ml64.exe users to specify how a frame function unwinds and is only allowed within the prologue, which extends from the PROC FRAME declaration to the .ENDPROLOG directive. These directives do not generate code; they only generate `.xdata` and `.pdata`. .ALLOCSTACK should be preceded by instructions that actually implement the actions to be unwound. It is a good practice to wrap both the unwind directives and the code they are meant to unwind in a macro to ensure agreement.

The *size* operand must be a multiple of 8.

Fore more information, see MASM for x64 (ml64.exe).

Sample

The following sample shows how to specify an unwind/exception handler:

```
; ml64 ex3.asm /link /entry:Example1  /SUBSYSTEM:Console
text SEGMENT
PUBLIC Example3
PUBLIC Example3_UW
Example3_UW PROC NEAR
   ; exception/unwind handler body

   ret 0

Example3_UW ENDP

Example3 PROC FRAME : Example3_UW

   sub rsp, 16
.allocstack 16

.endprolog

   ; function body
    add rsp, 16
   ret 0

Example3 ENDP
text ENDS
END
```

See Also

Other Resources

Directives Reference

.ALPHA

Orders segments alphabetically.

```
.ALPHA
```

See Also

Other Resources

Directives Reference

ASSUME

Enables error checking for register values.

```
ASSUME segregister:name [[, segregister:name]]...
ASSUME dataregister:type [[, dataregister:type]]...
ASSUME register:ERROR [[, register:ERROR]]...
ASSUME [[register:]] NOTHING [[, register:NOTHING]]...
```

Remarks

After an **ASSUME** is put into effect, the assembler watches for changes to the values of the given registers. **ERROR** generates an error if the register is used. **NOTHING** removes register error checking. You can combine different kinds of assumptions in one statement.

See Also

Other Resources

Directives Reference

.BREAK

Generates code to terminate a .WHILE or .REPEAT block if *condition* is true.

```
.BREAK [[.IF condition]]
```

See Also

Other Resources

Directives Reference

BYTE (MASM)

Allocates and optionally initializes a byte of storage for each *initializer*.

```
[[name]] BYTE initializer [[, initializer]] ...
```

Remarks

Can also be used as a type specifier anywhere a type is legal.

See Also

Other Resources

Directives Reference

CATSTR

Concatenates text items.

```
name CATSTR [[textitem1 [[, textitem2]] ...]]
```

Remarks

Each text item can be a literal string, a constant preceded by a **%**, or the string returned by a macro function.

See Also

Other Resources

Directives Reference

.CODE

When used with .MODEL, indicates the start of a code segment called *name* (the default segment name is _TEXT for tiny, small, compact, and flat models, or *module*_TEXT for other models).

```
.CODE [[name]]
```

See Also

Other Resources

Directives Reference

COMM

Creates a communal variable with the attributes specified in *definition*.

```
COMM definition [[, definition]] ...
```

Remarks

Each *definition* has the following form:

[[*langtype*]] [[**NEAR** | **FAR**]] *label***:***type*[[**:***count*]]

The *label* is the name of the variable. The *type* can be any type specifier (BYTE, WORD, and so on) or an integer specifying the number of bytes. The *count* specifies the number of data objects (one is the default).

See Also

Other Resources

Directives Reference

COMMENT (MASM)

Treats all *text* between or on the same line as the delimiters as a comment.

```
COMMENT delimiter [[text]]
[[text]]
[[text]] delimiter [[text]]
```

See Also

Other Resources

Directives Reference

.CONST

When used with .MODEL, starts a constant data segment (with segment name CONST).

```
.CONST
```

Remarks

This segment has the read-only attribute.

See Also

Other Resources

Directives Reference

.CONTINUE

Generates code to jump to the top of a .WHILE or .REPEAT block if *condition* is true.

```
.CONTINUE [[.IF condition]]
```

See Also

Other Resources

Directives Reference

.CREF

Enables listing of symbols in the symbol portion of the symbol table and browser file.

```
.CREF
```

See Also

Other Resources

Directives Reference

.DATA

When used with .MODEL, starts a near data segment for initialized data (segment name _DATA).

```
.DATA
```

See Also

Other Resources

Directives Reference

.DATA?

When used with .MODEL, starts a near data segment for uninitialized data (segment name _BSS).

```
.DATA?
```

See Also

Other Resources

Directives Reference

DB

Can be used to define data such as BYTE.

```
DB
```

See Also

Other Resources

Directives Reference

DD

Can be used to define data such as DWORD.

```
DD
```

See Also

Other Resources

Directives Reference

DF

Can be used to define data such as FWORD.

```
DF
```

See Also

Other Resources

Directives Reference

.DOSSEG

Orders the segments according to the MS-DOS segment convention: CODE first, then segments not in DGROUP, and then segments in DGROUP.

```
.DOSSEG
```

Remarks

The segments in DGROUP follow this order: segments not in BSS or STACK, then BSS segments, and finally STACK segments. Primarily used for ensuring CodeView support in MASM stand-alone programs. Same as DOSSEG.

See Also

Other Resources

Directives Reference

DOSSEG

Identical to .DOSSEG, which is the preferred form.

```
DOSSEG
```

See Also

Other Resources

Directives Reference

DQ

Can be used to define data such as QWORD.

```
DQ
```

See Also

Other Resources

Directives Reference

DT

Can be used to define data such as TBYTE.

```
DT
```

See Also

Other Resources

Directives Reference

DW

Can be used to define data such as WORD.

```
DW
```

See Also

Other Resources

Directives Reference

DWORD

Allocates and optionally initializes a double word (4 bytes) of storage for each *initializer*.

```
[[name]] DWORD initializer [[, initializer]]...
```

Remarks

Can also be used as a type specifier anywhere a type is legal.

See Also

Other Resources

Directives Reference

ECHO

Displays *message* to the standard output device (by default, the screen).

```
ECHO message
```

Remarks

Same as %OUT.

See Also

Other Resources

Directives Reference

.ELSE

```
.ELSE
```

Remarks

See .IF.

See Also

Other Resources

Directives Reference

ELSE (MASM)

Marks the beginning of an alternate block within a conditional block.

```
ELSE
```

Remarks

See IF.

See Also

Other Resources

Directives Reference

ELSEIF (MASM)

Combines ELSE and IF into one statement.

```
ELSEIF
```

Remarks

See IF.

See Also

Other Resources

Directives Reference

ELSEIF2

ELSEIF block evaluated on every assembly pass if **OPTION:SETIF2** is **TRUE**.

```
ELSEIF2
```

See Also

Other Resources

Directives Reference

END (MASM)

Marks the end of a module and, optionally, sets the program entry point to *address*.

```
END [[address]]
```

See Also

Other Resources

Directives Reference

.ENDIF

```
.ENDIF
```

Remarks

See .IF.

See Also

Other Resources

Directives Reference

ENDM

Terminates a macro or repeat block.

```
ENDM
```

Remarks

See MACRO, FOR, FORC, REPEAT, or **WHILE**.

See Also

Other Resources

Directives Reference

ENDP

Marks the end of procedure *name* previously begun with **PROC**.

```
name ENDP
```

Remarks

See PROC.

See Also

Other Resources

Directives Reference

.ENDPROLOG

Signals the end of the prologue declarations.

```
.ENDPROLOG
```

Remarks

It is an error to use any of the prologue declarations outside of the region between PROC FRAME and .ENDPROLOG.

For more information, see MASM for x64 (ml64.exe).

See Also

Other Resources

Directives Reference

ENDS (MASM)

Marks the end of segment, structure, or union *name* previously begun with SEGMENT, STRUCT, UNION, or a simplified segment directive.

```
name ENDS
```

See Also

Other Resources

Directives Reference

.ENDW

```
.ENDW
```

Remarks

See .WHILE.

See Also

Other Resources

Directives Reference

EQU

The first directive assigns numeric value of *expression* to *name*.

```
name EQU expression
name EQU <text>
```

Remarks

The *name* cannot be redefined later.

The second directive assigns specified *text* to *name*. The *name* can be assigned a different *text* later. See TEXTEQU.

See Also

Other Resources

Directives Reference

.ERR

Generates an error.

```
.ERR [[message]]
```

See Also

Other Resources

Directives Reference

.ERR2

.ERR block evaluated on every assembly pass if **OPTION:SETIF2** is **TRUE.**

```
.ERR2 [[message]]
```

See Also

Other Resources

Directives Reference

.ERRB

Generates an error if *textitem* is blank.

```
.ERRB <textitem> [[, message]]
```

See Also

Other Resources

Directives Reference

.ERRDEF

Generates an error if *name* is a previously defined label, variable, or symbol.

```
.ERRDEF name [[, message]]
```

See Also

Other Resources

Directives Reference

.ERRDIF[[I]]]

Generates an error if the text items are different.

```
.ERRDIF[[I]] <textitem1>, <textitem2> [[, message]]
```

Remarks

If **I** is given, the comparison is case insensitive.

See Also

Other Resources

Directives Reference

.ERRE

Generates an error if *expression* is false (0).

```
.ERRE expression [[, message]]
```

See Also

Other Resources

Directives Reference

.ERRIDN[[I]]

Generates an error if the text items are identical.

```
.ERRIDN[[I]] <textitem1>, <textitem2> [[, message]]
```

Remarks

If **I** is given, the comparison is case insensitive.

See Also

Other Resources

Directives Reference

.ERRNB

Generates an error if *textitem* is not blank.

```
.ERRNB <textitem> [[, message]]
```

See Also

Other Resources

Directives Reference

.ERRNDEF

Generates an error if *name* has not been defined.

```
.ERRNDEF name [[, message]]
```

See Also

Other Resources

Directives Reference

.ERRNZ

Generates an error if *expression* is true (nonzero).

```
.ERRNZ expression [[, message]]
```

See Also

Other Resources

Directives Reference

EVEN

Aligns the next variable or instruction on an even byte.

```
EVEN
```

See Also

Other Resources

Directives Reference

.EXIT

Generates termination code.

```
.EXIT [[expression]]
```

Remarks

Returns optional *expression* to shell.

See Also

Other Resources

Directives Reference

EXITM

Terminates expansion of the current repeat or macro block and begins assembly of the next statement outside the block.

```
EXITM [[textitem]]
```

Remarks

In a macro function, *textitem* is the value returned.

See Also

Other Resources

Directives Reference

EXTERN (MASM)

Defines one or more external variables, labels, or symbols called *name* whose type is *type*.

```
EXTERN [[langtype]] name [[(altid)]] :
      type [[, [[langtype]] name [[(altid)]] :type]]...
```

Remarks

The *type* can be ABS, which imports *name* as a constant. Same as EXTRN.

See Also

Other Resources

Directives Reference

EXTERNDEF

Defines one or more external variables, labels, or symbols called *name* whose type is *type*.

```
EXTERNDEF [[langtype]] name:type [[, [[langtype]] name:type]]...
```

Remarks

If *name* is defined in the module, it is treated as PUBLIC. If *name* is referenced in the module, it is treated as EXTERN. If *name* is not referenced, it is ignored. The *type* can be ABS, which imports *name* as a constant. Normally used in include files.

See Also

Other Resources

Directives Reference

EXTRN

```
EXTRN
```

Remarks

See EXTERN.

See Also

Other Resources

Directives Reference

.FARDATA

When used with .MODEL, starts a far data segment for initialized data (segment name FAR_DATA or *name*).

```
.FARDATA [[name]]
```

See Also

Other Resources

Directives Reference

.FARDATA?

When used with .MODEL, starts a far data segment for uninitialized data (segment name FAR_BSS or *name*).

```
.FARDATA? [[name]]
```

See Also

Other Resources

Directives Reference

FOR (MASM)

Marks a block that will be repeated once for each *argument*, with the current *argument* replacing *parameter* on each repetition.

```
FOR parameter [[:REQ | :=default]] , <argument [[, argument]]...>
      statements
      ENDM
```

Remarks

Same as IRP.

See Also

Other Resources

Directives Reference

FORC

Marks a block that will be repeated once for each character in *string*, with the current character replacing *parameter* on each repetition.

```
FORC
      parameter, <string> statements
      ENDM
```

Remarks

Same as IRPC.

See Also

Other Resources

Directives Reference

.FPO

The .FPO directive controls the emission of debug records to the .debug$F segment or section.

```
FPO (cdwLocals, cdwParams, cbProlog, cbRegs, fUseBP, cbFrame)
```

Parameters

cdwLocals

Number of local variables, an unsigned 32 bit value.

cdwParams

Size of the parameters, an unsigned 16 bit value.

cbProlog

Number of bytes in the function prolog code, an unsigned 8 bit value.

cbRegs

Number of bytes in the function prolog code, an unsigned 8 bit value.

fUseBP

Indicates whether the EBP register has been allocated. either 0 or 1.

cbFrame

Indicates the frame type. See FPO_DATA for more information.

See Also

Other Resources

Directives Reference

FWORD

Allocates and optionally initializes 6 bytes of storage for each *initializer*.

```
[[name]] FWORD initializer [[, initializer]]...
```

Remarks

Also can be used as a type specifier anywhere a type is legal.

See Also

Other Resources

Directives Reference

GOTO (MASM)

Transfers assembly to the line marked **:***macrolabel*.

```
GOTO macrolabel
```

Remarks

GOTO is permitted only inside MACRO, FOR, FORC, REPEAT, and **WHILE** blocks. The label must be the only directive on the line and must be preceded by a leading colon.

See Also

Other Resources

Directives Reference

GROUP

Add the specified *segments* to the group called *name*.

```
name GROUP segment [[, segment]]...
```

See Also

Other Resources

Directives Reference

.IF

Generates code that tests *condition1* (for example, AX > 7) and executes the *statements* if that condition is true.

```
.IF condition1
      statements
   [[.ELSEIF condition2
            statements]]
   [[.ELSE
            statements]]
      .ENDIF
```

Remarks

If a .ELSE follows, its statements are executed if the original condition was false. Note that the conditions are evaluated at run time.

See Also

Other Resources

Directives Reference

IF (MASM)

Grants assembly of *ifstatements* if *expression1* is true (nonzero) or *elseifstatements* if *expression1* is false (0) and *expression2* is true.

```
IF expression1
      ifstatements
   [[ELSEIF expression2
            elseifstatements]]
   [[ELSE
            elsestatements]]
   ENDIF
```

Remarks

The following directives may be substituted for ELSEIF: **ELSEIFB**, **ELSEIFDEF**, **ELSEIFDIF**, **ELSEIFDIFI**, **ELSEIFE**, **ELSEIFIDN**, **ELSEIFIDNI**, **ELSEIFNB**, and **ELSEIFNDEF**. Optionally, assembles *elsestatements* if the previous expression is false. Note that the expressions are evaluated at assembly time.

See Also

Other Resources

Directives Reference

IF2

IF block is evaluated on every assembly pass if **OPTION:SETIF2** is **TRUE**.

```
IF2 expression
```

Remarks

See IF for complete syntax.

See Also

Other Resources

Directives Reference

IFB

Grants assembly if *textitem* is blank.

```
IFB textitem
```

Remarks

See IF for complete syntax.

See Also

Other Resources

Directives Reference

IFDEF

Grants assembly if *name* is a previously defined label, variable, or symbol.

```
IFDEF name
```

Remarks

See IF for complete syntax.

See Also

Other Resources

Directives Reference

IFDIF[[I]]

Grants assembly if the text items are different.

```
IFDIF[[I]] textitem1, textitem2
```

Remarks

If **I** is given, the comparison is case insensitive. See IF for complete syntax.

See Also

Other Resources

Directives Reference

IFE

Grants assembly if *expression* is false (0).

```
IFE expression
```

Remarks

See IF for complete syntax.

See Also

Other Resources

Directives Reference

IFIDN[[I]]

Grants assembly if the text items are identical.

```
IFIDN[[I]] textitem1, textitem2
```

Remarks

If **I** is given, the comparison is case insensitive. See IF for complete syntax.

See Also

Other Resources

Directives Reference

IFNB

Grants assembly if *textitem* is not blank.

```
IFNB textitem
```

Remarks

See IF for complete syntax.

See Also

Other Resources

Directives Reference

IFNDEF

Grants assembly if *name* has not been defined.

```
IFNDEF name
```

Remarks

See IF for complete syntax.

See Also

Other Resources

Directives Reference

INCLUDE (MASM)

Inserts source code from the source file given by *filename* into the current source file during assembly.

```
INCLUDE filename
```

Remarks

The *filename* must be enclosed in angle brackets if it includes a backslash, semicolon, greater-than symbol, less-than symbol, single quotation mark, or double quotation mark.

See Also

Other Resources

Directives Reference

INCLUDELIB (MASM)

Informs the linker that the current module should be linked with *libraryname*.

```
INCLUDELIB libraryname
```

Remarks

The *libraryname* must be enclosed in angle brackets if it includes a backslash, semicolon, greater-than symbol, less-than symbol, single quotation mark, or double quotation mark.

See Also

Other Resources

Directives Reference

INSTR

Finds the first occurrence of *textitem2* in *textitem1*.

```
name INSTR [[position,]] textitem1, textitem2
```

Remarks

The starting *position* is optional. Each text item can be a literal string, a constant preceded by a **%**, or the string returned by a macro function.

See Also

Other Resources

Directives Reference

INVOKE

Calls the procedure at the address given by *expression*, passing the arguments on the stack or in registers according to the standard calling conventions of the language type.

```
INVOKE expression [[, arguments]]
```

Remarks

Each argument passed to the procedure may be an expression, a register pair, or an address expression (an expression preceded by **ADDR**).

See Also

Other Resources

Directives Reference

IRP

```
IRP
```

Remarks

See FOR.

See Also

Other Resources

Directives Reference

IRPC

```
IRPC
```

Remarks

See FORC.

See Also

Other Resources

Directives Reference

.K3D

Enables assembly of K3D instructions.

```
.K3D
```

See Also

Other Resources

Directives Reference

LABEL (MASM)

Creates a new label by assigning the current location-counter value and the given *type* to *name*.

```
name LABEL type
name LABEL [[NEAR | FAR | PROC]] PTR [[type]]
```

See Also

Other Resources

Directives Reference

.LALL

```
.LALL
```

Remarks

See .LISTMACROALL.

See Also

Other Resources

Directives Reference

.LFCOND

```
.LFCOND
```

Remarks

See .LISTIF.

See Also

Other Resources

Directives Reference

.LIST

Starts listing of statements.

```
.LIST
```

Remarks

This is the default.

See Also

Other Resources

Directives Reference

.LISTALL

Starts listing of all statements.

```
.LISTALL
```

Remarks

Equivalent to the combination of .LIST, .LISTIF, and .LISTMACROALL.

See Also

Other Resources

Directives Reference

.LISTIF

Starts listing of statements in false conditional blocks.

```
.LISTIF
```

Remarks

Same as .LFCOND.

See Also

Other Resources

Directives Reference

.LISTMACRO

Starts listing of macro expansion statements that generate code or data.

```
.LISTMACRO
```

Remarks

This is the default. Same as .XALL.

See Also

Other Resources

Directives Reference

.LISTMACROALL

Starts listing of all statements in macros.

```
.LISTMACROALL
```

Remarks

Same as .LALL.

See Also

Other Resources

Directives Reference

LOCAL (MASM)

In the first directive, within a macro, **LOCAL** defines labels that are unique to each instance of the macro.

```
LOCAL localname [[, localname]]...
LOCAL label [[ [count ] ]] [[:type]] [[, label [[ [count] ]] [[type]]]]...
```

Remarks

In the second directive, within a procedure definition (**PROC**), **LOCAL** creates stack-based variables that exist for the duration of the procedure. The *label* may be a simple variable or an array containing *count* elements.

See Also

Other Resources

Directives Reference

MACRO

Marks a macro block called *name* and establishes *parameter* placeholders for arguments passed when the macro is called.

```
name MACRO [[parameter [[:REQ | :=default | :VARARG]]]]...
      statements
   ENDM [[value]]
```

Remarks

A macro function returns *value* to the calling statement.

See Also

Other Resources

Directives Reference

MMWORD

Used for 64-bit multimedia operands with MMX and SSE (XMM) instructions.

```
MMWORD
```

Remarks

MMWORD was added to MASM in Visual C++ 2005.

MMWORD is a type. Prior to MMWORD being added to MASM, equivalent functionality could have been achieved with:

```
mov mm0, qword ptr [ebx]
```

While both instructions work on 64-bit operands, **QWORD** is the type for 64-bit unsigned integers and **MMWORD** is the type for a 64-bit multimedia value.

MMWORD is intended to represent the same type as __m64.

Example

```
movq     mm0, mmword ptr [ebx]
```

.MMX

Enables assembly of MMX or single-instruction, multiple data (SIMD) instructions.

```
.MMX
```

See Also

Other Resources

Directives Reference

.MODEL

Initializes the program memory model.

```
.MODEL memorymodel [[, langtype]] [[, stackoption]]
```

Remarks

The *memorymodel* can be **TINY**, **SMALL**, **COMPACT**, **MEDIUM**, **LARGE**, **HUGE**, or **FLAT**. The *langtype* can be **C**, **BASIC**, **FORTRAN**, **PASCAL**, **SYSCALL**, or **STDCALL**. The *stackoption* can be **NEARSTACK** or **FARSTACK**. *stackoption* is not allowed if *memorymodel* is **FLAT**.

See Also

Other Resources

Directives Reference

NAME (MASM)

Ignored.

```
NAME
```

See Also

Other Resources

Directives Reference

.NO87

Disallows assembly of all floating-point instructions.

```
.NO87
```

See Also

Other Resources

Directives Reference

.NOCREF

Suppresses listing of symbols in the symbol table and browser file.

```
.NOCREF [[name[[, name]]...]]
```

Remarks

If names are specified, then only the given names are suppressed. Same as .XCREF.

See Also

Other Resources

Directives Reference

.NOLIST

Suppresses program listing.

```
.NOLIST
```

Remarks

Same as .XLIST.

See Also

Other Resources

Directives Reference

.NOLISTIF

Suppresses listing of conditional blocks whose condition evaluates to false (0).

```
.NOLISTIF
```

Remarks

This is the default. Same as .SFCOND.

See Also

Other Resources

Directives Reference

.NOLISTMACRO

Suppresses listing of macro expansions.

```
.NOLISTMACRO
```

Remarks

Same as .SALL.

See Also

Other Resources

Directives Reference

OPTION (MASM)

Enables and disables features of the assembler.

```
OPTION optionlist
```

Remarks

Available options include:

CASEMAP	**DOTNAME**	**NODOTNAME**	**EMULATOR**
NOEMULATOR	**EPILOGUE**	**EXPR16**	**EXPR32**
LANGUAGE	**LJMP**	**NOLJMP**	**M510**
NOM510	**NOKEYWORD**	**NOSIGNEXTEND**	**OFFSET**
OLDMACROS	**NOOLDMACROS**	**OLDSTRUCTS**	**NOOLDSTRUCTS**
PROC	**PROLOGUE**	**READONLY**	**NOREADONLY**
SCOPED	**NOSCOPED**	**SEGMENT**	**SETIF2**.

The syntax for LANGUAGE is **OPTION LANGUAGE:***x*, where *x* is one of C, SYSCALL, STDCALL, PASCAL, FORTRAN, or BASIC. SYSCALL, PASCAL, FORTRAN, and BASIC are not supported with used with .MODEL FLAT.

See Also

Other Resources

Directives Reference

ORG

Sets the location counter to *expression*.

```
ORG expression
```

See Also

Other Resources

Directives Reference

%OUT

See ECHO.

```
%OUT
```

See Also

Other Resources

Directives Reference

OWORD

Used as a type specifier when an 16-byte data type is required.

```
OWORD
```

See Also

Other Resources

Directives Reference

PAGE

The first directive sets line *length* and character *width* of the program listing. If no arguments are given, generates a page break. The second directive increments the section number and resets the page number to 1.

```
PAGE [[[[length]], width]]
PAGE +
```

See Also

Other Resources

Directives Reference

POPCONTEXT

Restores part or all of the current *context* (saved by the PUSHCONTEXT directive). The *context* can be **ASSUMES**, **RADIX**, **LISTING**, **CPU**, or **ALL**.

```
POPCONTEXT context
```

See Also

Other Resources

Directives Reference

PROC

Marks start and end of a procedure block called *label*. The statements in the block can be called with the **CALL** instruction or INVOKE directive.

```
label PROC [[distance]] [[langtype]] [[visibility]] [[<prologuearg>]]
   [[USES reglist]] [[, parameter [[:tag]]]]...
   [FRAME [:ehandler-address] ]
      statements
      label ENDP
```

Remarks

[FRAME [:*ehandler-address*]] is only valid with ml64.exe, and causes MASM to generate a function table entry in .pdata and unwind information in .xdata for a function's structured exception handling unwind behavior.

When the **FRAME** attribute is used, it must be followed by an .ENDPROLOG directive.

See MASM for x64 (ml64.exe) for more information on using ml64.exe.

Example

```
; ml64 ex1.asm /link /entry:Example1 /SUBSYSTEM:CONSOLE
_text SEGMENT
Example1 PROC FRAME
   push r10
.pushreg r10
   push r15
.pushreg r15
   push rbx
.pushreg rbx
   push rsi
.pushreg rsi
.endprolog
   ; rest of function ...
   ret
Example1 ENDP
_text ENDS
END
```

The above code will emit the following function table and unwind information:

```
FileHeader->Machine 34404
Dumping Unwind Information for file ex2.exe

.pdata entry 1 0x00001000 0x00001023

  Unwind data: 0x00002000

    Unwind version: 1
    Unwind Flags: None
    Size of prologue: 0x08
    Count of codes: 3
    Frame register: rbp
    Frame offset: 0x0
    Unwind codes:

      Code offset: 0x08, SET_FPREG, register=rbp, offset=0x00
      Code offset: 0x05, ALLOC_SMALL, size=0x10
      Code offset: 0x01, PUSH_NONVOL, register=rbp
```

See Also

Other Resources

PROTO

Prototypes a function.

```
label PROTO [[distance]] [[langtype]] [[, [[parameter]]:tag]]...
```

See Also

Other Resources

Directives Reference

PUBLIC (MASM)

Makes each variable, label, or absolute symbol specified as *name* available to all other modules in the program.

```
PUBLIC [[langtype]] name [[, [[langtype]] name]]...
```

See Also

Other Resources

Directives Reference

PURGE

Deletes the specified macros from memory.

```
PURGE macroname [[, macroname]]...
```

See Also

Other Resources

Directives Reference

PUSHCONTEXT

Saves part or all of the current *context*: segment register assumes, radix value, listing and cref flags, or processor/coprocessor values. The *context* can be **ASSUMES**, **RADIX**, **LISTING**, **CPU**, or **ALL**.

```
PUSHCONTEXT context
```

See Also

Other Resources

Directives Reference

.PUSHFRAME

Generates a **UWOP_PUSH_MACHFRAME** unwind code entry. If the optional *code* is specified, the unwind code entry is given a modifier of 1. Otherwise the modifier is 0.

```
.PUSHFRAME [code]
```

Remarks

.PUSHFRAME allows ml64.exe users to specify how a frame function unwinds and is only allowed within the prologue, which extends from the PROC FRAME declaration to the .ENDPROLOG directive. These directives do not generate code; they only generate `.xdata` and `.pdata`. .PUSHFRAME should be preceded by instructions that actually implement the actions to be unwound. It is a good practice to wrap both the unwind directives and the code they are meant to unwind in a macro to ensure agreement.

For more information, see MASM for x64 (ml64.exe).

See Also

Other Resources

Directives Reference

.PUSHREG

Generates a **UWOP_PUSH_NONVOL** unwind code entry for the specified register number using the current offset in the prologue.

```
.PUSHREG register
```

Remarks

.PUSHREG allows ml64.exe users to specify how a frame function unwinds, and is only allowed within the prologue, which extends from the PROC FRAME declaration to the .ENDPROLOG directive. These directives do not generate code; they only generate `.xdata` and `.pdata`. .PUSHREG should be preceded by instructions that actually implement the actions to be unwound. It is a good practice to wrap both the unwind directives and the code they are meant to unwind in a macro to ensure agreement.

For more information, see MASM for x64 (ml64.exe).

Sample

The following sample shows how to push non-volatile tegisters.

```
; ml64 ex1.asm /link /entry:Example1 /SUBSYSTEM:CONSOLE
_text SEGMENT
Example1 PROC FRAME
   push r10
.pushreg r10
   push r15
.pushreg r15
   push rbx
.pushreg rbx
   push rsi
.pushreg rsi
.endprolog
   ; rest of function ...
   ret
Example1 ENDP
_text ENDS
END
```

See Also

Other Resources

Directives Reference

QWORD

Allocates and optionally initializes 8 bytes of storage for each *initializer*. Also can be used as a type specifier anywhere a type is legal.

```
[[name]] QWORD initializer [[, initializer]]...
```

See Also

Other Resources

Directives Reference

.RADIX

Sets the default radix, in the range 2 to 16, to the value of *expression*.

```
.RADIX expression
```

See Also

Other Resources

Directives Reference

REAL10

Allocates and optionally initializes a 10-byte floating-point number for each *initializer*.

```
name REAL10 initializer [[, initializer]]...
```

See Also

Other Resources

Directives Reference

REAL4

Allocates and optionally initializes a single-precision (4-byte) floating-point number for each *initializer*.

```
name REAL4 initializer [[, initializer]]...
```

See Also

Other Resources

Directives Reference

REAL8

Allocates and optionally initializes a double-precision (8-byte) floating-point number for each *initializer*.

```
name REAL8 initializer [[, initializer]]...
```

See Also

Other Resources

Directives Reference

RECORD (MASM)

Declares a record type consisting of the specified fields. *fieldname* names the field, *width* specifies the number of bits, and *expression* gives its initial value.

```
recordname RECORD fieldname:width [[= expression]]
   [[, fieldname:width [[= expression]]]]...
```

See Also

Other Resources

Directives Reference

.REPEAT

Generates code that repeats execution of the block of *statements* until *condition* becomes true. .UNTILCXZ, which becomes true when CX is zero, may be substituted for .UNTIL. The *condition* is optional with **.UNTILCXZ**.

```
.REPEAT
      statements
   .UNTIL condition
```

See Also

Other Resources

Directives Reference

REPEAT

Marks a block that is to be repeated *expression* times. Same as REPT.

```
REPEAT expression
      statements
      ENDM
```

See Also

Other Resources

Directives Reference

REPT

See REPEAT.

```
REPT
```

See Also

Other Resources

Directives Reference

.SAFESEH

Registers a function as a structured exception handler.

```
.SAFESEH identifier
```

Remarks

identifier must be the ID for a locally defined PROC or EXTRN PROC. A LABEL is not allowed. The .SAFESEH directive requires the /safeseh ml.exe command-line option.

For more information about structured exception handlers, see /SAFESEH.

For example, to register a safe exception handler, create a new MASM file (as follows), assemble with /safeseh, and add it to the linked objects.

```
.386
.model  flat
MyHandler   proto
.safeseh    MyHandler
end
```

See Also

Other Resources

Directives Reference

.SALL

See .NOLISTMACRO.

```
.SALL
```

See Also

Other Resources

Directives Reference

.SAVEREG

Generates either a **UWOP_SAVE_NONVOL** or a **UWOP_SAVE_NONVOL_FAR** unwind code entry for the specified register (*reg*) and offset (*offset*) using the current prologue offset. MASM will choose the most efficient encoding.

```
.SAVEREG reg, offset
```

Remarks

.SAVEREG allows ml64.exe users to specify how a frame function unwinds and is only allowed within the prologue, which extends from the PROC FRAME declaration to the .ENDPROLOG directive. These directives do not generate code; they only generate `.xdata` and `.pdata`. .SAVEREG should be preceded by instructions that actually implement the actions to be unwound. It is a good practice to wrap both the unwind directives and the code they are meant to unwind in a macro to ensure agreement.

For more information, see MASM for x64 (ml64.exe).

See Also

Other Resources

Directives Reference

.SAVEXMM128

Generates either a **UWOP_SAVE_XMM128** or a **UWOP_SAVE_XMM128_FAR** unwind code entry for the specified XMM register and offset using the current prologue offset. MASM will choose the most efficient encoding.

```
.savexmm128 xmmreg , offset
```

Remarks

.SAVEXMM128 allows ml64.exe users to specify how a frame function unwinds, and is only allowed within the prologue, which extends from the PROC FRAME declaration to the .ENDPROLOG directive. These directives do not generate code; they only generate `.xdata` and `.pdata`. .SAVEXMM128 should be preceded by instructions that actually implement the actions to be unwound. It is a good practice to wrap both the unwind directives and the code they are meant to unwind in a macro to ensure agreement.

offset must be a multiple of 16.

For more information, see MASM for x64 (ml64.exe).

See Also

Other Resources

Directives Reference

SBYTE (MASM)

Allocates and optionally initializes a signed byte of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal.

```
name SBYTE initializer [[, initializer]]...
```

See Also

Other Resources

Directives Reference

SDWORD

Allocates and optionally initializes a signed double word (4 bytes) of storage for each *initializer*. Also can be used as a type specifier anywhere a type is legal.

```
name SDWORD initializer [[, initializer]]...
```

See Also

Other Resources

Directives Reference

SEGMENT

Defines a program segment called *name* having segment attributes

```
name SEGMENT [[READONLY]] [[align]] [[combine]] [[use]] [[characteristics]] ALIAS(string) [
['class']]
      statements
      name ENDS
```

Parameters

align

The range of memory addresses from which a starting address for the segment can be selected. The alignment type can be any one of the following:

Align Type	Starting Address
BYTE	Next available byte address.
WORD	Next available word address (2 bytes per word).
DWORD	Next available double word address (4 bytes per double word).
PARA	Next available paragraph address (16 bytes per paragraph).
PAGE	Next available page address (256 bytes per page).
ALIGN(*n*)	Next available *n*th byte address. See Remarks section for more information.

If this parameter is not specified, **PARA** is used by default.

combine

PUBLIC, **STACK**, **COMMON**, **MEMORY**, **AT** *address*, **PRIVATE**

use

USE16, **USE32**, **FLAT**

characteristics

INFO, **READ**, **WRITE**, **EXECUTE**, **SHARED**, **NOPAGE**, **NOCACHE**, and **DISCARD**

These are supported for COFF only and correspond to the COFF section characteristics of similar name (for example, **SHARED** corresponds to IMAGE_SCN_MEM_SHARED). READ sets the IMAGE_SCN_MEM_READ flag. The obsolete READONLY flag caused the section to clear the IMG_SCN_MEM_WRITE flag. If any *characteristics* are set, the default characteristics are not used and only the programmer-specified flags are in effect.

ALIAS(*string* **)**

This string is used as the section name in the emitted COFF object. Creates multiple sections with the same external name, with distinct MASM segment names.

Not supported with **/omf**.

Remarks

For **ALIGN(***n***)**, *n* may be any power of 2 from 1 to 8192; not supported with **/omf**.

See Also

Other Resources

Directives Reference

.SEQ

Orders segments sequentially (the default order).

```
.SEQ
```

See Also

Other Resources

Directives Reference

.SETFRAME

Fills in the frame register field and offset in the unwind information using the specified register (*reg*) and offset (*offset*). The offset must be a multiple of 16 and less than or equal to 240. This directive also generates a **UWOP_SET_FPREG** unwind code entry for the specified register using the current prologue offset.

```
.SETFRAME reg, offset
```

Remarks

.SETFRAME allows ml64.exe users to specify how a frame function unwinds, and is only allowed within the prologue, which extends from the PROC FRAME declaration to the .ENDPROLOG directive. These directives do not generate code; they only generate `.xdata` and `.pdata`. .SETFRAME should be preceded by instructions that actually implement the actions to be unwound. It is a good practice to wrap both the unwind directives and the code they are meant to unwind in a macro to ensure agreement.

For more information, see MASM for x64 (ml64.exe).

Sample

The following sample shows how to use a frame pointer:

```
; ml64 frmex2.asm /link /entry:frmex2 /SUBSYSTEM:CONSOLE
_text SEGMENT
frmex2 PROC FRAME
   push rbp
.pushreg rbp
   sub rsp, 010h
.allocstack 010h
   mov rbp, rsp
.setframe rbp, 0
.endprolog
   ; modify the stack pointer outside of the prologue (similar to alloca)
   sub rsp, 060h

   ; we can unwind from the following AV because of the frame pointer
   mov rax, 0
   mov rax, [rax] ; AV!

   add rsp, 060h
   add rsp, 010h
   pop rbp
   ret
frmex2 ENDP
_text ENDS
END
```

See Also

Other Resources

Directives Reference

.SFCOND

See .NOLISTIF.

```
.SFCOND
```

See Also

Other Resources

Directives Reference

SIZESTR

Finds the size of a text item.

```
name SIZESTR textitem
```

See Also

Other Resources

Directives Reference

SQWORD

Allocates and optionally initializes 8 unsigned bytes of storage for each *initializer*. Also can be used as a type specifier anywhere a type is legal.

```
name SQWORD initializer [[, initializer]]...
```

See Also

Other Resources

Directives Reference

.STACK

When used with .MODEL, defines a stack segment (with segment name STACK). The optional *size* specifies the number of bytes for the stack (default 1,024). The **.STACK** directive automatically closes the stack statement.

```
.STACK [[size]]
```

See Also

Other Resources

Directives Reference

.STARTUP

Generates program start-up code.

```
.STARTUP
```

See Also

Other Resources

Directives Reference

STRUC

See STRUCT.

```
STRUC
```

See Also

Other Resources

Directives Reference

STRUCT (MASM)

Declares a structure type having the specified *fielddeclarations*. Each field must be a valid data definition. Same as STRUC.

```
name STRUCT [[alignment]] [[, NONUNIQUE]]
      fielddeclarations
      name ENDS
```

See Also

Other Resources

Directives Reference

SUBSTR

Returns a substring of *textitem*, starting at *position*. The *textitem* can be a literal string, a constant preceded by a **%**, or the string returned by a macro function.

```
name SUBSTR textitem, position [[, length]]
```

See Also

Other Resources

Directives Reference

SUBTITLE

Defines the listing subtitle. Same as SUBTTL.

```
SUBTITLE text
```

See Also

Other Resources

Directives Reference

SUBTTL

See SUBTITLE.

```
SUBTTL
```

See Also

Other Resources

Directives Reference

SWORD

Allocates and optionally initializes a signed word (2 bytes) of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal.

```
name SWORD initializer [[, initializer]]...
```

See Also

Other Resources

Directives Reference

TBYTE

Allocates and optionally initializes 10 bytes of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal.

```
[[name]] TBYTE initializer [[, initializer]]...
```

See Also

Other Resources

Directives Reference

TEXTEQU

Assigns *textitem* to *name*. The *textitem* can be a literal string, a constant preceded by a **%**, or the string returned by a macro function.

```
name TEXTEQU [[textitem]]
```

See Also

Other Resources

Directives Reference

.TFCOND

Toggles listing of false conditional blocks.

```
.TFCOND
```

See Also

Other Resources

Directives Reference

TITLE

Defines the program listing title.

```
TITLE text
```

See Also

Other Resources

Directives Reference

TYPEDEF (MASM)

Defines a new type called *name*, which is equivalent to *type*.

```
name TYPEDEF type
```

See Also

Other Resources

Directives Reference

UNION

Declares a union of one or more data types. The *fielddeclarations* must be valid data definitions. Omit the ENDS *name* label on nested **UNION** definitions.

```
name UNION [[alignment]] [[, NONUNIQUE]]
      fielddeclarations
[[name]] ENDS
```

See Also

Other Resources

Directives Reference

.UNTIL

See .REPEAT.

```
.UNTIL
```

See Also

Other Resources

Directives Reference

.UNTILCXZ

See .REPEAT.

See Also

Other Resources

Directives Reference

.WHILE

Generates code that executes the block of *statements* while *condition* remains true.

```
.WHILE condition
      statements
   .ENDW
```

See Also

Other Resources

Directives Reference

WHILE (MASM)

Repeats assembly of block *statements* as long as *expression* remains true.

```
WHILE expression
      statements
   ENDM
```

See Also

Other Resources

Directives Reference

WORD

Allocates and optionally initializes a word (2 bytes) of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal.

```
[[name]] WORD initializer [[, initializer]]...
```

See Also

Other Resources

Directives Reference

.XALL

See .LISTMACRO.

```
.XALL
```

See Also

Other Resources

Directives Reference

.XCREF

See .NOCREF.

```
.XCREF
```

See Also

Other Resources

Directives Reference

.XLIST

See .NOLIST.

```
.XLIST
```

See Also

Other Resources

Directives Reference

.XMM

Enables assembly of Internet Streaming SIMD Extension instructions.

```
.XMM
```

See Also

Other Resources

Directives Reference

XMMWORD

Used for 128-bit multimedia operands with MMX and SSE (XMM) instructions.

```
XMMWORD
```

Remarks

XMMWORD was added to MASM in Visual C++ 2005.

XMMWORD is intended to represent the same type as __m128.

Example

```
movdqa   xmm0, xmmword ptr [ebx]
```

Symbols Reference

Date and Time Information

@Date	@Time

Environment Information

@Cpu	@Environ
@Interface	@Version

File Information

@FileCur	@FileName
@Line	

Macro Functions

@CatStr	@InStr
@SizeStr	@SubStr

Miscellaneous

$	?	@@:
@B	@F	

Segment Information

@code	@CodeSize	@CurSeg
@data	@DataSize	@fardata
@fardata?	@Model	@stack
@WordSize		

See Also

Other Resources

Microsoft Macro Assembler Reference

$

The current value of the location counter.

```
$
```

See Also

Other Resources

Symbols Reference

?

In data declarations, a value that the assembler allocates but does not initialize.

```
?
```

See Also

Other Resources

Symbols Reference

@@:

Defines a code label recognizable only between *label1* and *label2*, where *label1* is either start of code or the previous **@@:** label, and *label2* is either end of code or the next **@@:** label. See @B and @F.

```
@@:
```

See Also

Other Resources

Symbols Reference

@B

The location of the previous @@: label.

```
@B
```

See Also

Other Resources

Symbols Reference

@CatStr

Macro function that concatenates one or more strings. Returns a string.

```
@CatStr( string1 [[, string2...]] )
```

See Also

Other Resources

Symbols Reference

@code

The name of the code segment (text macro).

```
@code
```

See Also

Other Resources

Symbols Reference

@CodeSize

0 for **TINY**, **SMALL**, **COMPACT**, and **FLAT** models, and 1 for **MEDIUM**, **LARGE**, and **HUGE** models (numeric equate).

```
@CodeSize
```

See Also

Other Resources

Symbols Reference

@Cpu

A bit mask specifying the processor mode (numeric equate).

```
@Cpu
```

See Also

Other Resources

Symbols Reference

@CurSeg

The name of the current segment (text macro).

```
@CurSeg
```

See Also

Other Resources

Symbols Reference

@data

The name of the default data group. Evaluates to DGROUP for all models except **FLAT**. Evaluates to **FLAT** under the **FLAT** memory model (text macro).

```
@data
```

See Also

Other Resources

Symbols Reference

@DataSize

0 for **TINY**, **SMALL**, **MEDIUM**, and **FLAT** models, 1 for **COMPACT** and **LARGE** models, and 2 for **HUGE** model (numeric equate).

```
@DataSize
```

See Also

Other Resources

Symbols Reference

@Date

The system date in the format mm/dd/yy (text macro).

```
@Date
```

See Also

Other Resources

Symbols Reference

@Environ

Value of environment variable *envvar* (macro function).

```
@Environ( envvar )
```

See Also

Other Resources

Symbols Reference

@F

The location of the next @@: label.

```
@F
```

See Also

Other Resources

Symbols Reference

@fardata

The name of the segment defined by the .FARDATA directive (text macro).

```
@fardata
```

See Also

Other Resources

Symbols Reference

@fardata?

The name of the segment defined by the .FARDATA? directive (text macro).

```
@fardata?
```

See Also

Other Resources

Symbols Reference

@FileCur

The name of the current file (text macro).

```
@FileCur
```

See Also

Other Resources

Symbols Reference

@FileName

The base name of the main file being assembled (text macro).

```
@FileName
```

See Also

Other Resources

Symbols Reference

@InStr

Macro function that finds the first occurrence of *string2* in *string1*, beginning at *position* within *string1*. If *position* does not appear, search begins at start of *string1*. Returns a position integer or 0 if *string2* is not found.

```
@InStr( [[position]], string1, string2 )
```

See Also

Other Resources

Symbols Reference

@Interface

Information about the language parameters (numeric equate).

```
@Interface
```

See Also

Other Resources

Symbols Reference

@Line

The source line number in the current file (numeric equate).

```
@Line
```

See Also

Other Resources

Symbols Reference

@Model

1 for **TINY** model, 2 for **SMALL** model, 3 for **COMPACT** model, 4 for **MEDIUM** model, 5 for **LARGE** model, 6 for **HUGE** model, and 7 for **FLAT** model (numeric equate).

```
@Model
```

See Also

Other Resources

Symbols Reference

@SizeStr

A macro function that returns the length of the given string. Returns an integer.

```
@SizeStr( string )
```

See Also

Other Resources

Symbols Reference

@stack

DGROUP for near stacks or STACK for far stacks (text macro).

```
@stack
```

See Also

Other Resources

Symbols Reference

@SubStr

A macro function that returns a substring starting at *position*.

```
@SubStr( string, position [[, length]] )
```

See Also

Other Resources

Symbols Reference

@Time

The system time in 24-hour hh:mm:ss format (text macro).

```
@Time
```

See Also

Other Resources

Symbols Reference

@Version

615 in MASM 6.15 (text macro).

```
@Version
```

See Also

Other Resources

Symbols Reference

@WordSize

Two for a 16-bit segment or four for a 32-bit segment (numeric equate).

```
@WordSize
```

See Also

Other Resources

Symbols Reference

Operators Reference

Arithmetic

*	+	-
.	/	[]
MOD		

Control Flow

!	!=	&
&&	<	<=
==	>	>=
||		

Logical and Shift

AND	NOT	OR
SHL	SHR	XOR

Macro

!	%	&
;;	<>	

Miscellaneous

' '	" "	:
::	;	CARRY?
DUP	OVERFLOW?	PARITY?
SIGN?	ZERO?	

Record

MASK	WIDTH

Relational

EQ	GE	GT
LE	LT	NE

Segment

:	IMAGEREL
LROFFSET	OFFSET
SECTIONREL	SEG

Type

HIGH	HIGH32	HIGHWORD
LENGTH	LENGTHOF	LOW
LOW32	LOWWORD	OPATTR
PTR	SHORT	SIZE
SIZEOF	THIS	TYPE

See Also

Other Resources

Microsoft Macro Assembler Reference

Operators (MASM)

For more information about operators in the Microsoft Macro Assembler Language, see Operators Reference.

See Also

Other Resources

Operators Reference

operator +

The first operator returns *expression1* plus *expression2*. The second operator reverses the sign of *expression*.

```
expression1 + expression2
-expression
```

See Also

Other Resources

Operators Reference

operator -

Returns *expression1* minus *expression2*.

```
expression1 - expression2
```

See Also

Other Resources

Operators Reference

operator *

Returns *expression1* times *expression2*.

```
expression1 * expression2
```

See Also

Other Resources

Operators Reference

operator /

Returns *expression1* divided by *expression2*.

```
expression1 / expression2
```

See Also

Other Resources

Operators Reference

operator []

Returns *expression1* plus **[***expression2***]**.

```
expression1 [expression2]
```

See Also

Other Resources

Operators Reference

operator :

Overrides the default segment of *expression* with *segment*. The *segment* can be a segment register, group name, segment name, or segment expression. The *expression* must be a constant.

```
segment: expression
```

See Also

Other Resources

Operators Reference

operator .

The first operator returns *expression* plus the offset of *field* within its structure or union. The second operator returns value at the location pointed to by *register* plus the offset of *field* within its structure or union.

```
expression. field [[. field]]...
[register]. field [[. field]]...
```

See Also

Other Resources

Operators Reference

operator <>

Treats *text* as a single literal element.

```
<text>
```

See Also

Other Resources

Operators Reference

operator " "

Treats "*text*" as a string.

```
"text"
```

See Also

Other Resources

Operators Reference

operator ' '

Treats '*text*' as a string.

```
'text'
```

See Also

Other Resources

Operators Reference

operator ! (MASM)

Treats *character* as a literal character rather than as an operator or symbol.

```
!character
```

See Also

Other Resources

Operators Reference

operator ;

Treats *text* as a comment.

```
;text
```

See Also

Other Resources

Operators Reference

operator ;;

Treats *text* as a comment in a macro that appears only in the macro definition. The listing does not show *text* where the macro is expanded.

```
;;text
```

See Also

Other Resources

Operators Reference

operator %

Treats the value of *expression* in a macro argument as text.

```
%expression
```

See Also

Other Resources

Operators Reference

operator && (MASM)

Replaces *parameter* with its corresponding argument value.

```
&parameter&
```

See Also

Other Resources

Operators Reference

operator ABS

See the EXTERNDEF directive.

```
ABS
```

See Also

Other Resources

Operators Reference

operator ADDR

See the INVOKE directive.

```
ADDR
```

See Also

Other Resources

Operators Reference

operator AND

Returns the result of a bitwise AND operation for *expression1* and *expression2*.

```
expression1 AND expression2
```

See Also

Other Resources

Operators Reference

operator DUP

Specifies *count* number of declarations of *initialvalue*.

```
count DUP (initialvalue [[, initialvalue]]...)
```

See Also

Other Resources

Operators Reference

operator EQ

Returns true (–1) if *expression1* equals *expression2*, or returns false (0) if it does not.

```
expression1 EQ expression2
```

See Also

Other Resources

Operators Reference

operator GE

Returns true (–1) if *expression1* is greater than or equal to *expression2*, or returns false (0) if it is not.

```
expression1 GE expression2
```

See Also

Other Resources

Operators Reference

operator GT

Returns true (–1) if *expression1* is greater than *expression2*, or returns false (0) if it is not.

```
expression1 GT expression2
```

See Also

Other Resources

Operators Reference

operator HIGH

Returns the high 8 bits of the low 16 bits of *expression*. MASM expressions are 64-bit values.

```
HIGH expression
```

See Also

Other Resources

Operators Reference

operator HIGH32

Returns the high 32 bits of *expression*. MASM expressions are 64-bit values.

```
HIGH32 expression
```

See Also

Other Resources

Operators Reference

operator HIGHWORD

Returns the high 16 bits of the low 32 bits of *expression*. MASM expressions are 64-bit values.

```
HIGHWORD expression
```

See Also

Other Resources

Operators Reference

operator IMAGEREL

Returns the image relative offset of *expression*.

```
IMAGEREL expression
```

Remarks

The resulting value is often referred to as an RVA or Relative Virtual Address.

IMAGEREL is available only with COFF object emission.

See Also

Other Resources

Operators Reference

operator LE

Returns true (–1) if *expression1* is less than or equal to *expression2*, or returns false (0) if it is not.

```
expression1 LE expression2
```

See Also

Other Resources

Operators Reference

operator LENGTH

Returns the number of data items in *variable* created by the first initializer.

```
LENGTH variable
```

See Also

Other Resources

Operators Reference

operator LENGTHOF

Returns the number of data objects in *variable*.

```
LENGTHOF variable
```

See Also

Other Resources

Operators Reference

operator LOW

Returns the low 8 bits of *expression*. MASM expressions are 64-bit values.

```
LOW expression
```

See Also

Other Resources

Operators Reference

operator LOW32

Returns the low 32 bits of *expression*. MASM expressions are 64-bit values.

```
LOW32 expression
```

See Also

Other Resources

Operators Reference

operator LOWWORD

Returns the low 16 bits of *expression*. MASM expressions are 64-bit values.

```
LOWWORD expression
```

See Also

Other Resources

Operators Reference

operator LROFFSET

Returns the offset of *expression*. Same as **OFFSET**, but it generates a loader resolved offset, which allows Windows to relocate code segments.

```
LROFFSET expression
```

See Also

Other Resources

Operators Reference

operator LT

Returns true (–1) if *expression1* is less than *expression2*, or returns false (0) if it is not.

```
expression1 LT expression2
```

See Also

Other Resources

Operators Reference

operator MASK

Returns a bit mask in which the bits in *recordfieldname* or *record* are set and all other bits are cleared.

```
MASK {recordfieldname | record}
```

See Also

Other Resources

Operators Reference

operator MOD

Returns the integer value of the remainder (modulo) when dividing *expression1* by *expression2*.

```
expression1 MOD expression2
```

See Also

Other Resources

Operators Reference

operator NE

Returns true (–1) if *expression1* does not equal *expression2*, or returns false (0) if it does.

```
expression1 NE expression2
```

See Also

Other Resources

Operators Reference

operator NOT

Returns *expression* with all bits reversed.

```
NOT expression
```

See Also

Other Resources

Operators Reference

operator OFFSET

Returns the offset of *expression*.

```
OFFSET expression
```

See Also

Other Resources

Operators Reference

operator OPATTR

Returns a word defining the mode and scope of *expression*. The low byte is identical to the byte returned by .TYPE. The high byte contains additional information.

```
OPATTR expression
```

See Also

Other Resources

Operators Reference

operator OR

Returns the result of a bitwise **OR** operation for *expression1* and *expression2*.

```
expression1 OR expression2
```

See Also

Other Resources

Operators Reference

operator PTR

The first operator forces the *expression* to be treated as having the specified *type*. The second operator specifies a pointer to *type*.

```
type PTR expression
[[distance]] PTR type
```

See Also

Other Resources

Operators Reference

operator SEG

Returns the segment of *expression*.

```
SEG expression
```

See Also

Other Resources

Operators Reference

operator SHL

Returns the result of shifting the bits of *expression* left *count* number of bits.

```
expression SHL count
```

See Also

Other Resources

Operators Reference

operator .TYPE

See OPATTR.

```
.TYPE expression
```

See Also

Other Resources

Operators Reference

operator SECTIONREL

Returns the section relative offset of expression relative to the section containing the target in the final executable.

```
SECTIONREL expression
```

Remarks

SECTIONREL is available only with COFF object emission.

See Also

Other Resources

Operators Reference

operator SHORT

Sets the type of *label* to short. All jumps to *label* must be short (within the range –128 to +127 bytes from the jump instruction to *label*).

```
SHORT label
```

See Also

Other Resources

Operators Reference

operator SHR

Returns the result of shifting the bits of *expression* right *count* number of bits.

```
expression SHR count
```

See Also

Other Resources

Operators Reference

operator SIZE

Returns the number of bytes in *variable* allocated by the first initializer.

```
SIZE variable
```

See Also

Other Resources

Operators Reference

operator SIZEOF

Returns the number of bytes in *variable* or *type*.

```
SIZEOF {variable | type}
```

See Also

Other Resources

Operators Reference

operator THIS

Returns an operand of specified *type* whose offset and segment values are equal to the current location counter value.

```
THIS type
```

See Also

Other Resources

Operators Reference

operator TYPE

Returns the type of *expression*.

```
TYPE expression
```

See Also

Other Resources

Operators Reference

operator WIDTH

Returns the width in bits of the current *recordfieldname* or *record*.

```
WIDTH {recordfieldname | record}
```

See Also

Other Resources

Operators Reference

operator XOR

Returns the result of a bitwise **XOR** operation for *expression1* and *expression2*.

```
expression1 XOR expression2
```

See Also

Other Resources

Operators Reference

Run-Time Operators

For more information about the run-time operators in the Microsoft Macro Assembler Language, see Operators Reference.

See Also

Other Resources

Operators Reference

operator == (MASM Run Time)

Is equal to. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

```
expression1 == expression2
```

See Also

Other Resources

Operators Reference

operator != (MASM)

Is not equal to. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

```
expression1 != expression2
```

See Also

Other Resources

Operators Reference

operator > (MASM Run Time)

Is greater than. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

```
expression1 > expression2
```

See Also

Other Resources

Operators Reference

operator >= (MASM Run Time)

Is greater than or equal to. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

```
expression1 >= expression2
```

See Also

Other Resources

Operators Reference

operator < (MASM Run Time)

Is less than. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

```
expression1 < expression2
```

See Also

Other Resources

Operators Reference

operator <= (MASM Run Time)

Is less than or equal to. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

```
expression1 <= expression2
```

See Also

Other Resources

Operators Reference

operator ||

Logical **OR**. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

```
expression1 || expression2
```

See Also

Other Resources

Operators Reference

operator && (MASM Run Time)

Logical **AND**. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

```
expression1 && expression2
```

See Also

Other Resources

Operators Reference

operator &

Bitwise **AND**. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

```
expression1 & expression2
```

See Also

Other Resources

Operators Reference

operator ! (MASM Run Time)

Logical negation. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

```
!expression
```

See Also

Other Resources

Operators Reference

operator CARRY?

Status of carry flag. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

```
CARRY?
```

See Also

Other Resources

Operators Reference

operator OVERFLOW?

Status of overflow flag. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

```
OVERFLOW?
```

See Also

Other Resources

Operators Reference

operator PARITY?

Status of parity flag. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

```
PARITY?
```

See Also

Other Resources

Operators Reference

operator SIGN?

Status of sign flag. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

```
SIGN?
```

See Also

Other Resources

Operators Reference

operator ZERO?

Status of zero flag. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

```
ZERO?
```

See Also

Other Resources

Operators Reference

ML Error Messages

The error messages generated by MASM components fall into three categories:

- **Fatal errors.** These indicate a severe problem that prevents the utility from completing its normal process.
- **Nonfatal errors.** The utility may complete its process. If it does, its result is not likely to be the one you want.
- **Warnings.** These messages indicate conditions that may prevent you from getting the results you want.

All error messages take the following form:

```
Utility: Filename (Line) : [Error_type} (Code): Message_text
```

where:

Utility

The program that sent the error message.

Filename

The file that contains the error-generating condition.

Line

The approximate line where the error condition exists.

Error_type

Fatal Error, Error, or Warning.

Code

The unique 5- or 6-digit error code.

Message_text

A short and general description of the error condition.

See Also

Other Resources

Microsoft Macro Assembler Reference

ML Fatal Errors

For more information about ML fatal errors, see ML Error Messages.

See Also

Reference

ML Error Messages

ML Fatal Error A1000

cannot open file: filename

The assembler was unable to open a source, include, or output file.

One of the following may be a cause:

- The file does not exist.
- The file is in use by another process.
- The filename is not valid.
- A read-only file with the output filename already exists.
- The current drive is full.
- The current directory is the root and is full.
- The device cannot be written to.
- The drive is not ready.

See Also

Reference

ML Error Messages

ML Fatal Error A1001

I/O error closing file

The operating system returned an error when the assembler attempted to close a file.

This error can be caused by having a corrupt file system or by removing a disk before the file could be closed.

See Also

Reference

ML Error Messages

ML Fatal Error A1002

I/O error writing file

The assembler was unable to write to an output file.

One of the following may be a cause:

- The current drive is full.
- The current directory is the root and is full.
- The device cannot be written to.
- The drive is not ready.

See Also

Reference

ML Error Messages

ML Fatal Error A1003

I/O error reading file

The assembler encountered an error when trying to read a file.

One of the following may be a cause:

- The disk has a bad sector.
- The file-access attribute is set to prevent reading.
- The drive is not ready.

See Also

Reference

ML Error Messages

ML Fatal Error A1004

out of memory

The operating system was unable to fill a request for memory.

To resolve this error, add more memory or increase the page file size. You might also try to split the current file into smaller files.

See Also

Reference

ML Error Messages

ML Fatal Error A1005

assembler limit : macro parameter name table full

Too many parameters, locals, or macro labels were defined for a macro. There was no more room in the macro name table.

Define shorter or fewer names, or remove unnecessary macros.

See Also

Reference

ML Error Messages

ML Fatal Error A1006

invalid command-line option: option

ML did not recognize the given parameter as an option.

This error is generally caused when there is a syntax error on the command line.

See Also

Reference

ML Error Messages

ML Fatal Error A1007

nesting level too deep

The assembler reached its nesting limit. The limit is 20 levels except where noted otherwise.

One of the following was nested too deeply:

- A high-level directive such as .IF, .REPEAT, or .WHILE.
- A structure definition.
- A conditional-assembly directive.
- A procedure definition.
- A PUSHCONTEXT directive (the limit is 10).
- A segment definition.
- An include file.
- A macro.

See Also

Reference

ML Error Messages

ML Fatal Error A1008

unmatched macro nesting

Either a macro was not terminated before the end of the file or the terminating directive ENDM was found outside of a macro block.

One cause of this error is omission of the dot before .REPEAT or .WHILE.

See Also

Reference

ML Error Messages

ML Fatal Error A1009

line too long

A line in a source file exceeded the limit of 512 characters.

If multiple physical lines are concatenated with the line-continuation character (\ ), then the resulting logical line is still limited to 512 characters.

See Also

Reference

ML Error Messages

ML Fatal Error A1010

unmatched block nesting :

A block beginning did not have a matching end, or a block end did not have a matching beginning. One of the following may be involved:

- A high-level directive such as .IF, .REPEAT, or .WHILE.
- A conditional-assembly directive such as IF, REPEAT, or **WHILE**.
- A structure or union definition.
- A procedure definition.
- A segment definition.
- A POPCONTEXT directive.
- A conditional-assembly directive, such as an ELSE, ELSEIF, or **ENDIF** without a matching IF.

See Also

Reference

ML Error Messages

ML Fatal Error A1011

directive must be in control block

The assembler found a high-level directive where one was not expected. One of the following directives was found:

- .ELSE without .IF
- .ENDIF without .IF
- .ENDW without .WHILE
- .UNTILCXZ without .REPEAT
- .CONTINUE without .WHILE or .REPEAT
- .BREAK without .WHILE or .REPEAT
- .ELSE following **.ELSE**

See Also

Reference

ML Error Messages

ML Fatal Error A1012

error count exceeds 100; stopping assembly

The number of nonfatal errors exceeded the assembler limit of 100.

Nonfatal errors are in the range A2xxx. When warnings are treated as errors, they are included in the count. Warnings are considered errors if you use the **/WX** command-line option.

See Also

Reference

ML Error Messages

ML Fatal Error A1013

invalid numerical command-line argument : number

The argument specified with an option was not a number or was an invalid number.

See Also

Reference

ML Error Messages

ML Fatal Error A1014

too many arguments

There was insufficient memory to hold all of the command-line arguments.

This error usually occurs while expanding input filename wildcards (* and ?). To eliminate this error, assemble multiple source files separately.

See Also

Reference

ML Error Messages

ML Fatal Error A1015

statement too complex

The assembler ran out of stack space while trying to parse the specified statement.

One or more of the following changes may eliminate this error:

- Break the statement into several shorter statements.
- Reorganize the statement to reduce the amount of parenthetical nesting.
- If the statement is part of a macro, break the macro into several shorter macros.

See Also

Reference

ML Error Messages

ML Fatal Error A1016

Internal Assembler Error

The MASM driver, called ML.exe, generated a system error.

Note the circumstances of the error and notify Microsoft Corporation. Product Support Services is available at http://support.microsoft.com/.

See Also

Reference

ML Error Messages

ML Fatal Error A1017

missing source filename

ML could not find a file to assemble or pass to the linker.

This error is generated when you give ML command-line options without specifying a filename to act upon. To assemble files that do not have an .asm extension, use the **/Ta** command-line option.

This error can also be generated by invoking ML with no parameters if the ML environment variable contains command-line options.

See Also

Reference

ML Error Messages

ML Fatal Error A1018

COFF error writing file

The MASM driver, called ML.exe, generated a system error.

Note the circumstances of the error and notify Microsoft Corporation. Product Support Services is available at http://support.microsoft.com/.

See Also

Reference

ML Error Messages

ML Fatal Error A1019

file exceeds browser line limit

When compiling with **/Fr** or **/FR**, your file exceeded browser line limit.

See ML and ML64 Command-Line Reference for more information.

To correct this error

- Limit source files to 32,767 lines or do not use the **/Fr** or **/FR** options.

See Also

Reference

ML Error Messages

ML Fatal Error A1020

cannot find LINK.EXE

ml.exe attempted to invoke the Visual C++ linker, but unable.

Make sure that link.exe is on your disk, and in a directory on the path.

See Also

Reference

ML Error Messages

ML Fatal Error A1022

Error Message
command-line option requires /omf

A1022 will occur when you compile with **/FPi** and **/coff**. x87 emulation is not supported with **/coff.** Since **/coff** is on by default, using **/FPi** without **/omf** will cause A1022.

For information on MASM compiler options, see ML and ML64 Command-Line Reference.

ML Nonfatal Errors

For more information about ML nonfatal errors, see ML Error Messages.

See Also

Reference

ML Error Messages

ML Nonfatal Errors A2000-A2049

For more information about ML nonfatal errors, see ML Error Messages.

See Also

Other Resources

ML Nonfatal Errors

ML Nonfatal Error A2000

memory operand not allowed in context

A memory operand was given to an instruction that cannot take a memory operand.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2001

immediate operand not allowed

A constant or memory offset was given to an instruction that cannot take an immediate operand.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2002

cannot have more than one ELSE clause per IF block

The assembler found an ELSE directive after an existing **ELSE** directive in a conditional-assembly block (IF block).

Only one **ELSE** can be used in an **IF** block. An **IF** block begins with an **IF**, IFE, IFB, IFNB, IFDEF, IFNDEF, IFDIF, or IFIDN directive. There can be several ELSEIF statements in an **IF** block.

One cause of this error is omission of an **ENDIF** statement from a nested **IF** block.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2003

extra characters after statement

A directive was followed by unexpected characters.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2004

symbol type conflict : identifier

The EXTERNDEF or LABEL directive was used on a variable, symbol, data structure, or label that was defined in the same module but with a different type.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2005

symbol redefinition : identifier

The given nonredefinable symbol was defined in two places.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2006

undefined symbol : identifier

An attempt was made to use a symbol that was not defined.

One of the following may have occurred:

- A symbol was not defined.
- A field was not a member of the specified structure.
- A symbol was defined in an include file that was not included.
- An external symbol was used without an EXTERN or EXTERNDEF directive.
- A symbol name was misspelled.
- A local code label was referenced outside of its scope.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2007

non-benign record redefinition

A RECORD definition conflicted with a previous definition.

One of the following occurred:

- There were different numbers of fields.
- There were different numbers of bits in a field.
- There was a different label.
- There were different initializers.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2008

syntax error :

A token at the current location caused a syntax error.

One of the following may have occurred:

- A dot prefix was added to or omitted from a directive.
- A reserved word (such as **C** or **SIZE**) was used as an identifier.
- An instruction was used that was not available with the current processor or coprocessor selection.
- A comparison run-time operator (such as **==**) was used in a conditional assembly statement instead of a relational operator (such as EQ).
- An instruction or directive was given too few operands.
- An obsolete directive was used.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2009

syntax error in expression

An expression on the current line contained a syntax error. This error message may also be a side effect of a preceding program error.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2010

invalid type expression

The operand to THIS or PTR was not a valid type expression.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2011

distance invalid for word size of current segment

A procedure definition or a code label defined with LABEL specified an address size that was incompatible with the current segment size.

One of the following occurred:

- A **NEAR16** or **FAR16** procedure was defined in a 32-bit segment.
- A **NEAR32** or **FAR32** procedure was defined in a 16-bit segment.
- A code label defined with **LABEL** specified **FAR16** or **NEAR16** in a 32-bit segment.
- A code label defined with **LABEL** specified **FAR32** or **NEAR32** in a 16-bit segment.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2012

PROC, MACRO, or macro repeat directive must precede LOCAL

A LOCAL directive must be immediately preceded by a MACRO, PROC, macro repeat directive (such as REPEAT, **WHILE**, or FOR), or another LOCAL directive.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2013

.MODEL must precede this directive

A simplified segment directive or a .STARTUP or .EXIT directive was not preceded by a .MODEL directive.

A .MODEL directive must specify the model defaults before a simplified segment directive, or a .STARTUP or .EXIT directive may be used.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2014

cannot define as public or external : identifier

Only labels, procedures, and numeric equates can be made public or external using PUBLIC, EXTERN, or EXTERNDEF. Local code labels cannot be made public.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2015

segment attributes cannot change : attribute

A segment was reopened with different attributes than it was opened with originally.

When a SEGMENT directive opens a previously defined segment, the newly opened segment inherits the attributes with which the segment was defined.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2016

expression expected

The assembler expected an expression at the current location but found one of the following:

- A unary operator without an operand
- A binary operator without two operands
- An empty pair of parentheses, ( ), or brackets, [ ]

See Also

Reference

ML Error Messages

ML Nonfatal Error A2017

operator expected

An expression operator was expected at the current location.

One possible cause of this error is a missing comma between expressions in an expression list.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2018

invalid use of external symbol : identifier

An attempt was made to compare the given external symbol using a relational operator.

The comparison cannot be made because the value or address of an external symbol is not known at assembly time.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2019

operand must be RECORD type or field

The operand following the WIDTH or MASK operator was not valid.

The WIDTH operator takes an operand that is the name of a field or a record. The MASK operator takes an operand that is the name of a field or a record type.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2020

identifier not a record : identifier

A record type was expected at the current location.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2021

record constants cannot span line breaks

A record constant must be defined on one physical line. A line ended in the middle of the definition of a record constant.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2022

instruction operands must be the same size

The operands to an instruction did not have the same size.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2023

instruction operand must have size

At least one of the operands to an instruction must have a known size.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2024

invalid operand size for instruction

The size of an operand was not valid.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2025

operands must be in same segment

Relocatable operands used with a relational or minus operator were not located in the same segment.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2026

constant expected

The assembler expected a constant expression at the current location. A constant expression is a numeric expression that can be resolved at assembly time.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2027

operand must be a memory expression

The right operand of a PTR expression was not a memory expression.

When the left operand of the **PTR** operator is a structure or union type, the right operand must be a memory expression.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2028

expression must be a code address

An expression evaluating to a code address was expected.

One of the following occurred:

- SHORT was not followed by a code address.
- **NEAR** PTR or **FAR PTR** was applied to something that was not a code address.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2029

multiple base registers not allowed

An attempt was made to combine two base registers in a memory expression.

For example, the following expressions cause this error:

```
[bx+bp]
[bx][bp]
```

In another example, given the following definition:

```
id1 proc arg1:byte
```

Either line causes this error:

```
mov al, [bx].arg1
lea ax, arg1[bx]
```

See Also

Reference

ML Error Messages

ML Nonfatal Error A2030

multiple index registers not allowed

An attempt was made to combine two index registers in a memory expression.

For example, the following expressions cause this error:

```
[si+di]
[di][si]
```

See Also

Reference

ML Error Messages

ML Nonfatal Error A2031

must be index or base register

An attempt was made to use a register that was not a base or index register in a memory expression.

For example, the following expressions cause this error:

```
[ax]
[bl]
```

See Also

Reference

ML Error Messages

ML Nonfatal Error A2032

invalid use of register

An attempt was made to use a register that was not valid for the intended use.

One of the following occurred:

- OFFSET was applied to a register. (**OFFSET** can be applied to a register under the **M510** option.)
- A special 386 register was used in an invalid context.
- A register was cast with PTR to a type of invalid size.
- A register was specified as the right operand of a segment override operator (:).
- A register was specified as the right operand of a binary minus operator (–).
- An attempt was made to multiply registers using the ***** operator.
- Brackets (**[ ]**) were missing around a register that was added to something.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2033

invalid INVOKE argument : argument number

The INVOKE directive was passed a special 386 register, or a register pair containing a byte register or special 386 register. These registers are illegal with **INVOKE**.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2034

must be in segment block

One of the following was found outside of a segment block:

- An instruction
- A label definition
- A THIS operator
- A $ operator
- A procedure definition
- An ALIGN directive
- An ORG directive

See Also

Reference

ML Error Messages

ML Nonfatal Error A2035

DUP too complex

A declaration using the DUP operator resulted in a data structure with an internal representation that was too large.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2036

too many initial values for structure: structure

The given structure was defined with more initializers than the number of fields in the type declaration of the structure.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2037

statement not allowed inside structure definition

A structure definition contained an invalid statement.

A structure cannot contain instructions, labels, procedures, control-flow directives, .STARTUP, or .EXIT.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2038

missing operand for macro operator

The assembler found the end of a macro's parameter list immediately after the ! or % operator.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2039

line too long

A source-file line exceeded the limit of 512 characters.

If multiple physical lines are concatenated with the line-continuation character (\ ), then the resulting logical line is still limited to 512 characters.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2040

segment register not allowed in context

A segment register was specified for an instruction that cannot take a segment register.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2041

string or text literal too long

A string or text literal, or a macro function return value, exceeded the limit of 255 characters.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2042

statement too complex

A statement was too complex for the assembler to parse.

Reduce either the number of tokens or the number of forward-referenced identifiers.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2043

identifier too long

An identifier exceeded the limit of 247 characters.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2044

invalid character in file

The source file contained a character outside a comment, string, or literal that was not recognized as an operator or other legal character.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2045

missing angle bracket or brace in literal

An unmatched angle bracket (either < or >) or brace (either { or }) was found in a literal constant or an initializer.

One of the following occurred:

- A pair of angle brackets or braces was not complete.
- An angle bracket was intended to be literal, but it was not preceded by an exclamation point (**!**) to indicate a literal character.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2046

missing single or double quotation mark in string

An unmatched quotation mark (either ' or ") was found in a string.

One of the following may have occurred:

- A pair of quotation marks around a string was not complete.
- A pair of quotation marks around a string was formed of one single and one double quotation mark.
- A single or double quotation mark was intended to be literal, but the surrounding quotation marks were the same kind as the literal one.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2047

empty (null) string

A string consisted of a delimiting pair of quotation marks and no characters within.

For a string to be valid, it must contain 1–255 characters.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2048

nondigit in number

A number contained a character that was not in the set of characters used by the current radix (base).

This error can occur if a B or D radix specifier is used when the default radix is one that includes that letter as a valid digit.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2049

syntax error in floating-point constant

A floating-point constant contained an invalid character.

See Also

Reference

ML Error Messages

ML Nonfatal Errors A2050-A2099

For more information about ML nonfatal errors, see ML Error Messages.

See Also

Other Resources

ML Nonfatal Errors

ML Nonfatal Error A2050

real or BCD number not allowed

A floating-point (real) number or binary coded decimal (BCD) constant was used other than as a data initializer.

One of the following occurred:

- A real number or a BCD was used in an expression.
- A real number was used to initialize a directive other than DWORD, QWORD, or TBYTE.
- A BCD was used to initialize a directive other than **TBYTE**.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2051

text item required

A literal constant or text macro was expected.

One of the following was expected:

- A literal constant, which is text enclosed in **<** **>**.
- A text macro name.
- A macro function call.
- A % followed by a constant expression.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2052

forced error

The conditional-error directive .ERR or **.ERR1** was used to generate this error.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2053

forced error : value equal to 0

The conditional-error directive .ERRE was used to generate this error.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2054

forced error : value not equal to 0

The conditional-error directive .ERRNZ was used to generate this error.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2055

forced error : symbol not defined

The conditional-error directive .ERRNDEF was used to generate this error.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2056

forced error : symbol defined

The conditional-error directive .ERRDEF was used to generate this error.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2057

forced error : string blank

The conditional-error directive .ERRB was used to generate this error.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2058

forced error : string not blank

The conditional-error directive .ERRNB was used to generate this error.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2059

forced error : strings equal

The conditional-error directive .ERRIDN or **.ERRIDNI** was used to generate this error.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2060

forced error : strings not equal

The conditional-error directive .ERRDIF or **.ERRDIFI** was used to generate this error.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2061

[[ELSE]]IF2/.ERR2 not allowed : single-pass assembler

A directive for a two-pass assembler was found.

The Microsoft Macro Assembler (MASM) is a one-pass assembler. MASM does not accept the IF2, ELSEIF2, and .ERR2 directives.

This error also occurs if an ELSE directive follows an **IF1** directive.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2062

expression too complex for .UNTILCXZ

An expression used in the condition that follows .UNTILCXZ was too complex.

The **.UNTILCXZ** directive can take only one expression, which can contain only **==** or **!=**. It cannot take other comparison operators or more complex expressions using operators like **||**.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2063

can ALIGN only to power of 2 : expression

The expression specified with the ALIGN directive was invalid.

The **ALIGN** expression must be a power of 2 between 2 and 256, and must be less than or equal to the alignment of the current segment, structure, or union.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2064

structure alignment must be 1, 2, 4, 8, or 16

The alignment specified in a structure definition was invalid.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2065

expected : token

The assembler expected the given token.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2066

incompatible CPU mode and segment size

An attempt was made to open a segment with a **USE16**, **USE32**, or **FLAT** attribute that was not compatible with the specified CPU, or to change to a 16-bit CPU while in a 32-bit segment.

The **USE32** and **FLAT** attributes must be preceded by the .386 or greater processor directive.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2067

LOCK must be followed by a memory operation

The **LOCK** prefix preceded an invalid instruction. No instruction can take the **LOCK** prefix unless one of its operands is a memory expression.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2068

instruction prefix not allowed

One of the prefixes **REP**, **REPE**, **REPNE**, or **LOCK** preceded an instruction for which it was not valid.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2069

no operands allowed for this instruction

One or more operands were specified with an instruction that takes no operands.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2070

invalid instruction operands

One or more operands were not valid for the instruction with which they were specified.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2071

initializer magnitude too large for specified size

An initializer value was too large for the data area it was initializing.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2073

operands have different frames

Two operands in an expression were in different frames.

Subtraction of pointers requires the pointers to be in the same frame. Subtraction of two expressions that have different effective frames is not allowed. An effective frame is calculated from the segment, group, or segment register.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2074

cannot access label through segment registers

An attempt was made to access a label through a segment register that was not assumed to its segment or group.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2075

jump destination too far [: by 'n' bytes]

The destination specified with a jump instruction was too far from the instruction.

One of the following may be a solution:

- Enable the **LJMP** option.
- Remove the SHORT operator. If **SHORT** has forced a jump that is too far, *n* is the number of bytes out of range.
- Rearrange code so that the jump is no longer out of range.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2076

jump destination must specify a label

A direct jump's destination must be relative to a code label.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2077

instruction does not allow NEAR indirect addressing

A conditional jump or loop cannot take a memory operand. It must be given a relative address or label.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2078

instruction does not allow FAR indirect addressing

A conditional jump or loop cannot take a memory operand. It must be given a relative address or label.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2079

instruction does not allow FAR direct addressing

A conditional jump or loop cannot be to a different segment or group.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2080

jump distance not possible in current CPU mode

A distance was specified with a jump instruction that was incompatible with the current processor mode.

For example, 48-bit jumps require .386 or above.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2081

missing operand after unary operator

An operator required an operand, but no operand followed.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2082

cannot mix 16- and 32-bit registers

An address expression contained both 16- and 32-bit registers.

For example, the following expression causes this error:

```
[bx+edi]
```

See Also

Reference

ML Error Messages

ML Nonfatal Error A2083

invalid scale value

A register scale was specified that was not 1, 2, 4, or 8.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2084

constant value too large

A constant was specified that was too big for the context in which it was used.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2085

instruction or register not accepted in current CPU mode

An attempt was made to use an instruction, register, or keyword that was not valid for the current processor mode.

For example, 32-bit registers require .386 or above. Control registers such as CR0 require privileged mode .386P or above. This error will also be generated for the **NEAR32**, **FAR32**, and **FLAT** keywords, which require .**386** or above.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2087

instruction form requires 80386/486

An instruction was used that was not compatible with the current processor mode.

The .386 or greater processor directive must precede the instruction.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2088

END directive required at end of file

The assembler reached the end of the main source file and did not find an END directive.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2089

too many bits in RECORD : identifier

One of the following occurred:

- Too many bits were defined for the given record field.
- Too many total bits were defined for the given record.

The size limit for a record or a field in a record is 16 bits when doing 16-bit arithmetic or 32 bits when doing 32-bit arithmetic.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2090

positive value expected

A positive value was not found in one of the following situations:

- The starting position specified for SUBSTR or @SubStr.
- The number of data objects specified for COMM.
- The element size specified for **COMM**.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2091

index value past end of string

An index value exceeded the length of the string it referred to when used with INSTR, SUBSTR, @InStr, or @SubStr.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2092

count must be positive or zero

The operand specified to the SUBSTR directive, @SubStr macro function, SHL operator, SHR operator, or DUP operator was negative.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2093

count value too large

The length argument specified for SUBSTR or @SubStr exceeded the length of the specified string.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2094

operand must be relocatable

An operand was not relative to a label.

One of the following occurred:

- An operand specified with the END directive was not relative to a label.
- An operand to the SEG operator was not relative to a label.
- The right operand to the minus operator was relative to a label, but the left operand was not.
- The operands to a relational operator were either not both integer constants or not both memory operands. Relational operators can take operands that are both addresses or both nonaddresses but not one of each.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2095

constant or relocatable label expected

The operand specified must be a constant expression or a memory offset.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2096

segment, group, or segment register expected

A segment or group was expected but was not found.

One of the following occurred:

- The left operand specified with the segment override operator (**:**) was not a segment register (CS, DS, SS, ES, FS, or GS), group name, segment name, or segment expression.
- The ASSUME directive was given a segment register without a valid segment address, segment register, group, or the special **FLAT** group.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2097

segment expected : identifier

The GROUP directive was given an identifier that was not a defined segment.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2098

invalid operand for OFFSET

The expression following the OFFSET operator must be a memory expression or an immediate expression.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2099

invalid use of external absolute

An attempt was made to subtract a constant defined in another module from an expression.

You can avoid this error by placing constants in include files rather than making them external.

See Also

Reference

ML Error Messages

ML Nonfatal Errors A2100-A2149

For more information about ML nonfatal errors, see ML Error Messages.

See Also

Other Resources

ML Nonfatal Errors

ML Nonfatal Error A2100

segment or group not allowed

An attempt was made to use a segment or group in a way that was not valid. Segments or groups cannot be added.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2101

cannot add two relocatable labels

An attempt was made to add two expressions that were both relative to a label.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2102

cannot add memory expression and code label

An attempt was made to add a code label to a memory expression.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2103

segment exceeds 64K limit

A 16-bit segment exceeded the size limit of 64K.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2104

invalid type for data declaration : type

The given type was not valid for a data declaration.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2105

HIGH and LOW require immediate operands

The operand specified with either the HIGH or the LOW operator was not an immediate expression.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2107

cannot have implicit far jump or call to near label

An attempt was made to make an implicit far jump or call to a near label in another segment.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2108

use of register assumed to ERROR

An attempt was made to use a register that had been assumed to ERROR with the ASSUME directive.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2109

only white space or comment can follow backslash

A character other than a semicolon (;) or a white-space character (spaces or TAB characters) was found after a line-continuation character (\ ).

See Also

Reference

ML Error Messages

ML Nonfatal Error A2110

COMMENT delimiter expected

A delimiter character was not specified for a COMMENT directive.

The delimiter character is specified by the first character that is not white space (spaces or TAB characters) after the **COMMENT** directive. The comment consists of all text following the delimiter until the end of the line containing the next appearance of the delimiter.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2111

conflicting parameter definition

A procedure defined with the PROC directive did not match its prototype as defined with the PROTO directive.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2112

PROC and prototype calling conventions conflict

A procedure was defined in a prototype (using the PROTO, EXTERNDEF, or EXTERN directive), but the calling convention did not match the corresponding PROC directive.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2113

invalid radix tag

The specified radix was not a number in the range 2–16.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2114

INVOKE argument type mismatch : argument number

The type of the arguments passed using the INVOKE directive did not match the type of the parameters in the prototype of the procedure being invoked.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2115

invalid coprocessor register

The coprocessor index specified was negative or greater than 7.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2117

/AT switch requires TINY memory model

/AT was specified on the assembler command line, but the program being assembled did not specify the **TINY** memory model with the .MODEL directive.

This error is only generated for modules that specify a start address or use the .STARTUP directive.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2118

cannot have segment address references with TINY model

An attempt was made to reference a segment in a **TINY** model program.

All **TINY** model code and data must be accessed with **NEAR** addresses.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2119

language type must be specified

A procedure definition or prototype was not given a language type.

A language type must be declared in each procedure definition or prototype if a default language type is not specified. A default language type is set using either the .MODEL directive, **OPTION LANG**, or the ML command-line options **/Gc** or **/Gd**.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2120

PROLOGUE must be macro function

The identifier specified with the **OPTION PROLOGUE** directive was not recognized as a defined macro function.

The user-defined prologue must be a macro function that returns the number of bytes needed for local variables and any extra space needed for the macro function.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2121

EPILOGUE must be macro procedure

The identifier specified with the **OPTION EPILOGUE** directive was not recognized as a defined macro procedure.

The user-defined epilogue macro cannot return a value.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2122

alternate identifier not allowed with EXTERNDEF

An attempt was made to specify an alternate identifier with an EXTERNDEF directive.

You can specify an optional alternate identifier with the EXTERN directive but not with **EXTERNDEF**.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2123

text macro nesting level too deep

A text macro was nested too deeply. The nesting limit for text macros is 40.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2125

missing macro argument

A required argument to @InStr, @SubStr, or a user-defined macro was not specified.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2126

EXITM used inconsistently

The EXITM directive was used both with and without a return value in the same macro.

A macro procedure returns a value; a macro function does not.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2127

macro function argument list too long

There were too many characters in a macro function's argument list. This error applies also to a prologue macro function called implicitly by the PROC directive.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2129

VARARG parameter must be last parameter

A parameter other than the last one was given the **VARARG** attribute.

The **:VARARG** specification can be applied only to the last parameter in a parameter list for macro and procedure definitions and prototypes. You cannot use multiple **:VARARG** specifications in a macro.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2130

VARARG parameter not allowed with LOCAL

An attempt was made to specify **:VARARG** as the type in a procedure's **LOCAL** declaration.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2131

VARARG parameter requires C calling convention

A **VARARG** parameter was specified in a procedure definition or prototype, but the **C**, **SYSCALL**, or **STDCALL** calling convention was not specified.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2132

ORG needs a constant or local offset

The expression specified with the ORG directive was not valid.

ORG requires an immediate expression with no reference to an external label or to a label outside the current segment.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2133

register value overwritten by INVOKE

A register was passed as an argument to a procedure, but the code generated by INVOKE to pass other arguments destroyed the contents of the register.

The AX, AL, AH, EAX, DX, DL, DH, and EDX registers may be used by the assembler to perform data conversion.

Use a different register.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2134

structure too large to pass with INVOKE : argument number

An attempt was made with INVOKE to pass a structure that exceeded 255 bytes.

Pass structures by reference if they are larger than 255 bytes.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2136

too many arguments to INVOKE

The number of arguments passed using the INVOKE directive exceeded the number of parameters in the prototype for the procedure being invoked.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2137

too few arguments to INVOKE

The number of arguments passed using the INVOKE directive was fewer than the number of required parameters specified in the prototype for the procedure being invoked.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2138

invalid data initializer

The initializer list for a data definition was invalid.

This error can be caused by using the R radix override with too few digits.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2140

RET operand too large

The operand specified to **RET**, **RETN**, or **RETF** exceeded two bytes.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2141

too many operands to instruction

Too many operands were specified with a string control instruction.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2142

cannot have more than one .ELSE clause per .IF block

The assembler found more than one .ELSE clause within the current .IF block.

Use **.ELSEIF** for all but the last block.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2143

expected data label

The LENGTHOF, SIZEOF, LENGTH, or SIZE operator was applied to a nondata label, or the **SIZEOF** or **SIZE** operator was applied to a type.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2144

cannot nest procedures

An attempt was made to nest a procedure containing a parameter, local variable, **USES** clause, or a statement that generated a new segment or group.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2145

EXPORT must be FAR : procedure

The given procedure was given **EXPORT** visibility and **NEAR** distance.

All **EXPORT** procedures must be **FAR**. The default visibility may have been set with the **OPTION PROC:EXPORT** statement or the **SMALL** or **COMPACT** memory models.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2146

procedure declared with two visibility attributes : procedure

The given procedure was given conflicting visibilities.

A procedure was declared with two different visibilities (**PUBLIC**, **PRIVATE**, or **EXPORT**). The PROC and PROTO statements for a procedure must have the same visibility.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2147

macro label not defined : macrolabel

The given macro label was not found.

A macro label is defined with **:***macrolabel*.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2148

invalid symbol type in expression : identifier

The given identifier was used in an expression in which it was not valid.

For example, a macro procedure name is not allowed in an expression.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2149

byte register cannot be first operand

A byte register was specified to an instruction that cannot take it as the first operand.

See Also

Reference

ML Error Messages

ML Nonfatal Errors A2150-A2199

For more information about ML nonfatal errors, see ML Error Messages.

See Also

Other Resources

ML Nonfatal Errors

ML Nonfatal Error A2150

word register cannot be first operand

A word register was specified to an instruction that cannot take it as the first operand.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2151

special register cannot be first operand

A special register was specified to an instruction that cannot take it as the first operand.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2152

coprocessor register cannot be first operand

A coprocessor (stack) register was specified to an instruction that cannot take it as the first operand.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2153

cannot change size of expression computations

An attempt was made to set the expression word size when the size had been already set using the **EXPR16**, **EXPR32**, **SEGMENT:USE32**, or **SEGMENT:FLAT** option or the .386 or higher processor selection directive.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2154

syntax error in control-flow directive

The condition for a control-flow directive (such as .IF or .WHILE) contained a syntax error.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2155

cannot use 16-bit register with a 32-bit address

An attempt was made to mix 16-bit and 32-bit offsets in an expression.

Use a 32-bit register with a symbol defined in a 32-bit segment.

For example, if `id1` is defined in a 32-bit segment, the following causes this error:

```
id1[bx]
```

See Also

Reference

ML Error Messages

ML Nonfatal Error A2156

constant value out of range

An invalid value was specified for the PAGE directive.

The first parameter of the **PAGE** directive can be either 0 or a value in the range 10–255. The second parameter of the **PAGE** directive can be either 0 or a value in the range 60–255.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2157

missing right parenthesis

A right parenthesis,), was missing from a macro function call.

Be sure that parentheses are in pairs if nested.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2158

type is wrong size for register

An attempt was made to assume a general-purpose register to a type with a different size than the register.

For example, the following pair of statements causes this error:

```
ASSUME bx:far ptr byte ; far pointer is 4 or 6 bytes
ASSUME al:word         ; al is a byte reg, cannot hold word
```

See Also

Reference

ML Error Messages

ML Nonfatal Error A2159

structure cannot be instanced

An attempt was made to create an instance of a structure when there were no fields or data defined in the structure definition or when **ORG** was used in the structure definition.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2160

non-benign structure redefinition : label incorrect

A label given in a structure redefinition either did not exist in the original definition or was out of order in the redefinition.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2161

non-benign structure redefinition : too few labels

Not enough members were defined in a structure redefinition.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2162

OLDSTRUCT/NOOLDSTRUCT state cannot be changed

Once the **OLDSTRUCTS** or **NOOLDSTRUCTS** option has been specified and a structure has been defined, the structure scoping cannot be altered or respecified in the same module.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2163

non-benign structure redefinition : incorrect initializers

A **STRUCT** or **UNION** was redefined with a different initializer value.

When structures and unions are defined more than once, the definitions must be identical. This error can be caused by using a variable as an initializer and having the value of the variable change between definitions.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2164

non-benign structure redefinition : too few initializers

A **STRUCT** or **UNION** was redefined with too few initializers.

When structures and unions are defined more than once, the definitions must be identical.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2165

non-benign structure redefinition : label has incorrect offset

The offset of a label in a redefined **STRUCT** or **UNION** differs from the original definition.

When structures and unions are defined more than once, the definitions must be identical. This error can be caused by a missing member or by a member that has a different size than in its original definition.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2166

structure field expected

The right side of a dot operator (.) is not a structure field.

This error may occur with some code acceptable to previous versions of the assembler. To enable the old behavior, use **OPTION OLDSTRUCTS**, which is automatically enabled by **OPTION M510** or the /Zm command-line option.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2167

unexpected literal found in expression

A literal was found where an expression was expected.

One of the following may have occurred:

- A literal was used as an initializer
- A record tag was omitted from a record constant

See Also

Reference

ML Error Messages

ML Nonfatal Error A2169

divide by zero in expression

An expression contains a divisor whose value is equal to zero.

Check that the syntax of the expression is correct and that the divisor (whether constant or variable) is correctly initialized.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2170

directive must appear inside a macro

A **GOTO** or **EXITM** directive was found outside the body of a macro.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2171

cannot expand macro function

A syntax error prevented the assembler from expanding the macro function.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2172

too few bits in RECORD

There was an attempt to define a record field of 0 bits.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2173

macro function cannot redefine itself

There was an attempt to define a macro function inside the body of a macro function with the same name. This error can also occur when a member of a chain of macros attempts to redefine a previous member of the chain.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2175

invalid qualified type

An identifier was encountered in a qualified type that was not a type, structure, record, union, or prototype.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2176

floating point initializer on an integer variable

An attempt was made to use a floating-point initializer with **DWORD**, **QWORD**, or **TBYTE**. Only integer initializers are allowed.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2177

nested structure improperly initialized

The nested structure initialization could not be resolved.

This error can be caused by using different beginning and ending delimiters in a nested structure initialization.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2178

invalid use of FLAT

There was an ambiguous reference to **FLAT** as a group.

This error is generated when there is a reference to **FLAT** instead of a **FLAT** subgroup. For example,

```
mov    ax, FLAT              ; Generates A2178
mov    ax, SEG FLAT:_data    ; Correct
```

See Also

Reference

ML Error Messages

ML Nonfatal Error A2179

structure improperly initialized

There was an error in a structure initializer.

One of the following occurred:

- The initializer is not a valid expression.
- The initializer is an invalid **DUP** statement.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2180

improper list initialization

In a structure, there was an attempt to initialize a list of items with a value or list of values of the wrong size.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2181

initializer must be a string or single item

There was an attempt to initialize a structure element with something other than a single item or string.

This error can be caused by omitting braces (**{ }**) around an initializer.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2182

initializer must be a single item

There was an attempt to initialize a structure element with something other than a single item.

This error can be caused by omitting braces (**{ }**) around an initializer.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2183

initializer must be a single byte

There was an attempt to initialize a structure element of byte size with something other than a single byte.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2184

improper use of list initializer

The assembler did not expect an opening brace (**{**) at this point.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2185

improper literal initialization

A literal structure initializer was not properly delimited.

This error can be caused by missing angle brackets (**< >**) or braces (**{ }**) around an initializer or by extra characters after the end of an initializer.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2186

extra characters in literal initialization

A literal structure initializer was not properly delimited.

One of the following may have occurred:

- There were missing or mismatched angle brackets (**< >**) or braces (**{ }**) around an initializer.
- There were extra characters after the end of an initializer.
- There was a syntax error in the structure initialization.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2187

must use floating point initializer

A variable declared with the **REAL4**, **REAL8**, and **REAL10** directives must be initialized with a floating-point number or a question mark (?).

This error can be caused by giving an initializer in integer form (such as 18) instead of in floating-point form (18.0).

See Also

Reference

ML Error Messages

ML Nonfatal Error A2188

cannot use .EXIT for OS_OS2 with .8086

The **INVOKE** generated by the **.EXIT** statement under **OS_OS2** requires the **.186** (or higher) directive, since it must be able to use the **PUSH** instruction to push immediates directly.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2189

invalid combination with segment alignment

The alignment specified by the **ALIGN** or **EVEN** directive was greater than the current segment alignment as specified by the **SEGMENT** directive.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2190

INVOKE requires prototype for procedure

The **INVOKE** directive must be preceded by a **PROTO** statement for the procedure being called.

When using **INVOKE** with an address rather than an explicit procedure name, you must precede the address with a pointer to the prototype.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2191

cannot include structure in self

You cannot reference a structure recursively (inside its own definition).

See Also

Reference

ML Error Messages

ML Nonfatal Error A2192

symbol language attribute conflict

Two declarations for the same symbol have conflicting language attributes (such as **C** and **PASCAL**). The attributes should be identical or compatible.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2193

non-benign COMM redefinition

A variable was redefined with the **COMM** directive to a different language type, distance, size, or instance count.

Multiple **COMM** definitions of a variable must be identical.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2194

COMM variable exceeds 64K

A variable declared with the **COMM** directive in a 16-bit segment was greater than 64K.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2195

parameter or local cannot have void type

The assembler attempted to create an argument or create a local without a type.

This error can be caused by declaring or passing a symbol followed by a colon without specifying a type or by using a user-defined type defined as void.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2196

cannot use TINY model with OS_OS2

A **.MODEL** statement specified the **TINY** memory model and the **OS_OS2** operating system. The tiny memory model is not allowed under OS/2.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2197

expression size must be 32-bits

There was an attempt to use the 16-bit expression evaluator in a 32-bit segment. In a 32-bit segment (**USE32** or **FLAT**), you cannot use the default 16-bit expression evaluator (**OPTION EXPR16**).

See Also

Reference

ML Error Messages

ML Nonfatal Error A2198

.EXIT does not work with 32-bit segments

The **.EXIT** directive cannot be used in a 32-bit segment; it is valid only when generating 16-bit code.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2199

.STARTUP does not work with 32-bit segments

The **.STARTUP** directive cannot be used in a 32-bit segment; it is valid only when generating 16-bit code.

See Also

Reference

ML Error Messages

ML Nonfatal Errors A2200-A2901

For more information about ML nonfatal errors, see ML Error Messages.

See Also

Other Resources

ML Nonfatal Errors

ML Nonfatal Error A2200

ORG directive not allowed in unions

The **ORG** directive is not valid inside a **UNION** definition.

You can use the **ORG** directive inside **STRUCT** definitions, but it is meaningless inside a **UNION**.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2202

illegal use of segment register

You cannot use segment overrides for the FS or GS segment registers when generating floating-point emulation instructions with the /FPi command-line option or **OPTION EMULATOR**.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2203

cannot declare scoped code label as PUBLIC

A code label defined with the "label:" syntax was declared **PUBLIC**. Use the "label::" syntax, the **LABEL** directive, or **OPTION NOSCOPED** to eliminate this error.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2204

.MSFLOAT directive is obsolete : ignored

The Microsoft Binary Format is no longer supported. You should convert your code to the IEEE numeric standard, which is used in the 80x87-series coprocessors.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2205

ESC instruction is obsolete : ignored

The **ESC** (Escape) instruction is no longer supported. All numeric coprocessor instructions are now supported directly by the assembler.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2206

missing operator in expression

An expression cannot be evaluated because it is missing an operator. This error message may also be a side-effect of a preceding program error.

The following line will generate this error:

```
value1 = ( 1 + 2 ) 3
```

See Also

Reference

ML Error Messages

ML Nonfatal Error A2207

missing right parenthesis in expression

An expression cannot be evaluated because it is missing a right (closing) parenthesis. This error message may also be a side-effect of a preceding program error.

The following line will generate this error:

```
value1 = ( ( 1 + 2 ) * 3
```

See Also

Reference

ML Error Messages

ML Nonfatal Error A2208

missing left parenthesis in expression

An expression cannot be evaluated because it is missing a left (opening) parenthesis. This error message may also be a side-effect of a preceding program error.

The following line will generate this error:

```
value1 = ( ( 1 + 2 ) * 3 ) )
```

See Also

Reference

ML Error Messages

ML Nonfatal Error A2209

reference to forward macro redefinition

A macro cannot be accessed because it has not been yet defined.

Move the macro definition ahead of all references to the macro.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2211

invalid .model parameter for flat model

The *stackoption* or the *langtype* parameter used was not compatible with the FLAT memory model. See .MODEL for more information.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2212

invalid .model parameter for flat model

NEARSTACK or FARSTACK cannot be used as *stackoption* parameters when the *memorymodel* is FLAT. See .MODEL for more information.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2213

ALIAS name is empty

The ALIAS directive was not specified correctly.

See ALIAS (MASM) for more information.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2214

GROUP directive not allowed with /coff option

The GROUP directive is not permitted when compiling with **/coff**.

See ML and ML64 Command-Line Reference for more information.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2215

.FPO is not compatible with nested procedures

An .FPO directive can not be used in a procedure block that is nested within another procedure block or in a procedure block that contains a nested procedure block.

See PROC for information on defining a procedure block.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2216

.SAFESEH argument must be a PROC in an executable segment

The .SAFESEH directive must be in an executable segment in a procedure block (PROC).

Example

The following sample shows a how to declare a safe exception handler in a procedure block.

```
.686
.model  flat
.code
.safeseh   MyHandler
MyHandler   proc

;handler
ret

MyHandler   endp
end
```

See Also

Reference

ML Error Messages

ML Nonfatal Error A2217

must be public or external

An alternate symbol was not public or external.

Example

The code sample below would generate A2217 if the line:

```
        public  alternate
```

were not present.

```
        .686
        .model  flat

        extrn   primary(alternate):near

        .code

        public  alternate
alternate:
        ret

        end
```

See Also

Reference

ML Error Messages

ML Nonfatal Error A2218

AH, BH, CH, or DH may not be used with R8-R15, BPL, DIL, SIL, or SPL

A2218 is for ml64.exe only.

When an instruction encoding uses any of the R8-15, BPL, DIL, SIL, or SPL registers, it is not possible to use AH, BH, CH, or DH.

ML Nonfatal Error A2219

Bad alignment for offset in unwind code

The operand for .ALLOCSTACK and .SAVEREG must be a multiple of 8. The operand for .SAVEXMM128 and .SETFRAME must be a multiple of 16.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2220

Missing .ENDPROLOG

When the **FRAME** attribute is used in a procedure block, it must be followed by an .ENDPROLOG directive.

See PROC for more information.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2221

Missing Frame in proc, no unwind code will be generated.

A2221 occurs when .ALLOCSTACK, .ENDPROLOG, .PUSHFRAME, .PUSHREG, .SAVEREG, .SAVEXMM128, or .SETFRAME are used in a procedure block (PROC) that does not have a **FRAME** attribute.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2229

cannot use labels and 32-bit addressing in the same expression

When compiling with ml64.exe, 32-bit addressing can not be used with a relocatable symbol.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2247

size of prolog too big, must be < 256 bytes

When compiling with ml64.exe and using **PROC** with **FRAME**, the .ENDPROLOG must be within the first 255 bytes of the function.

See Also

Reference

ML Error Messages

ML Nonfatal Error A2249

Error Message
not supported with /coff

A2249 occurs when you use **/coff** with **EMULATOR** or **NOEMULATOR**. **/coff** is on by default. Use **/omf** if you want to use **EMULATOR** or **NOEMULATOR**.

For information on MASM compiler options, see ML and ML64 Command-Line Reference. **EMULATOR** or **NOEMULATOR** are used with the OPTION (MASM) directive.

ML Warnings

For more information about ML warnings, see ML Error Messages.

See Also

Reference

ML Error Messages

ML Warning A4000

cannot modify READONLY segment

An attempt was made to modify an operand in a segment marked with the **READONLY** attribute.

See Also

Reference

ML Error Messages

ML Warning A4002

non-unique STRUCT/UNION field used without qualification

A **STRUCT** or **UNION** field can be referenced without qualification only if it has a unique identifier.

This conflict can be resolved either by renaming one of the structure fields to make it unique or by fully specifying both field references.

The **NONUNIQUE** keyword requires that all references to the elements of a **STRUCT** or **UNION** be fully specified.

See Also

Reference

ML Error Messages

ML Warning A4003

start address on END directive ignored with .STARTUP

Both **.STARTUP** and a program load address (optional with the **END** directive) were specified. The address specification with the **END** directive was ignored.

See Also

Reference

ML Error Messages

ML Warning A4004

cannot ASSUME CS

An attempt was made to assume a value for the CS register. CS is always set to the current segment or group.

See Also

Reference

ML Error Messages

ML Warning A4005

unknown default prologue argument

An unknown argument was passed to the default prologue.

The default prologue understands only the **FORCEFRAME** and **LOADDS** arguments.

See Also

Reference

ML Error Messages

ML Warning A4006

too many arguments in macro call

There were more arguments given in the macro call than there were parameters in the macro definition.

See Also

Reference

ML Error Messages

ML Warning A4008

invalid command-line option value, default is used : option

The value specified with the given option was not valid. The option was ignored, and the default was assumed.

See Also

Reference

ML Error Messages

ML Warning A4009

insufficient memory for /EP : /EP ignored

There is not enough memory to generate a first-pass listing.

See Also

Reference

ML Error Messages

ML Warning A4010

expected '>' on text literal

A macro was called with a text literal argument that was missing a closing angle bracket.

See Also

Reference

ML Error Messages

ML Warning A4011

multiple .MODEL directives found : .MODEL ignored

More than one **.MODEL** directive was found in the current module. Only the first **.MODEL** statement is used.

See Also

Reference

ML Error Messages

ML Warning A4012

line number information for segment without class 'CODE'

There were instructions in a segment that did not have a class name that ends with "CODE." The assembler did not generate CodeView information for these instructions.

CodeView cannot process modules with code in segments with class names that do not end with "CODE."

See Also

Reference

ML Error Messages

ML Warning A4013

instructions and initialized data not supported in AT segments

An instruction or initialized data was found in a segment defined with the AT attribute. The code or data will not be loaded at run time.

Data in AT segments must be declared with the **?** initializer.

See Also

Reference

ML Error Messages

ML Warning A4014

instructions and initialized data not supported in BSS segments

An attempt was made to define initialized data within a BSS section. A BSS section is defined as a class whose name is BSS. This includes the simplified segment **.data?**.

See Also

Reference

ML Error Messages

ML Warning A4022

debug information too complex for

A STRUCT (MASM) definition is so complex that its description exceeds the size limit in debug information.

See Also

Reference

ML Error Messages

ML Warning A4026

.ENDPROLOG found before EH directives

A4026 occurs when .ALLOCSTACK, .ENDPROLOG, .PUSHFRAME, .PUSHREG, .SAVEREG, .SAVEXMM128, or .SETFRAME is used after the end of the prolog in a procedure block (PROC).

See Also

Reference

ML Error Messages

ML Warning A4027

reserved word expected

One or more items in the list specified with a **NOKEYWORD** option were not recognized as reserved words.

See Also

Reference

ML Error Messages

ML Warning A4028

must read and agree to the Data Collection Policy before using /errorReport:send

Before you can use **/errorReport:send** to send information on an internal error in ml.exe or ml64.exe you must agree to Microsoft's policy on data collection.

The text of the A4028 diagnostic will refer you to a website where you can agree to Microsoft's policy on data collection.

For more information on error reporting, see /errorReport (Report Internal Compiler Errors).

See Also

Reference

ML Error Messages

ML Warning A5000

@@: label defined but not referenced

A jump target was defined with the **@@**: label, but the target was not used by a jump instruction.

One common cause of this error is insertion of an extra **@@:** label between the jump and the **@@:** label that the jump originally referred to.

See Also

Reference

ML Error Messages

ML Warning A5001

expression expected, assume value 0

There was an **IF**, **ELSEIF**, **IFE**, **IFNE**, **ELSEIFE**, or **ELSEIFNE** directive without an expression to evaluate. The assembler assumes a 0 for the comparison expression.

See Also

Reference

ML Error Messages

ML Warning A5002

externdef previously assumed to be external

The **OPATTR** or **.TYPE** operator was applied to a symbol after the symbol was used in an **EXTERNDEF** statement but before it was declared. These operators were used on a line where the assembler assumed that the symbol was external.

See Also

Reference

ML Error Messages

ML Warning A5003

length of symbol previously assumed to be different

The **LENGTHOF**, **LENGTH**, **SIZEOF**, or **SIZE** operator was applied to a symbol after the symbol was used in an **EXTERNDEF** statement but before it was declared. These operators were used on a line where the assembler assumed that the symbol had a different length and size.

See Also

Reference

ML Error Messages

ML Warning A5004

symbol previously assumed to not be in a group

A symbol was used in an **EXTERNDEF** statement outside of a segment and then was declared inside a segment.

See Also

Reference

ML Error Messages

ML Warning A5005

types are different

The type given by an **INVOKE** statement differed from that given in the procedure prototype. The assembler performed the appropriate type conversion.

See Also

Reference

ML Error Messages

ML Warning A6001

no return from procedure

A **PROC** statement generated a prologue, but there was no **RET** or **IRET** instruction found inside the procedure block.

See Also

Reference

ML Error Messages

ML Warning A6003

conditional jump lengthened

A conditional jump was encoded as a reverse conditional jump around a near unconditional jump.

You may be able to rearrange code to avoid the longer form.

See Also

Reference

ML Error Messages

ML Warning A6004

procedure argument or local not referenced

You passed a procedure argument or created a variable with the **LOCAL** directive that was not used in the procedure body.

Unnecessary parameters and locals waste code and stack space.

See Also

Reference

ML Error Messages

ML Warning A6005

expression condition may be pass-dependent

Under the **/Zm** command-line option or the **OPTION M510** directive, the value of an expression changed between passes.

This error message may indicate that the code is pass-dependent and must be rewritten.

See Also

Reference

ML Error Messages

Visual C++ Samples

Visual C++ provides walkthroughs, which give step-by-step instructions for common scenarios, and sample applications, which are complete applications that use many of the technologies in Visual C++.

In This Section

Visual C++ Walkthroughs

Provides links to walkthroughs that discuss the development of a specific application type or major application feature using a series of steps.

Visual C++ Sample Applications

Provides links to sample code showing the capabilities of Visual C++ and the libraries and technologies it supports.

Visual C++ Walkthroughs

Walkthroughs give step-by-step instructions for common scenarios, which makes them a good place to start learning about the product or a particular feature area.

Getting Started

Walkthrough: Using the Visual Studio IDE

Creates a new standard C++ program and tests it functionality using Visual Studio.

Walkthrough: Creating a Windows Forms Control

Creates a Windows Forms control and an application to test it.

Walkthrough: Creating and Using a Managed Assembly

Creates a managed assembly and an application that references the assembly.

Walkthrough: Creating and Using a Dynamic Link Library

Creates a dynamic link library and an application that references the DLL.

Walkthrough: Creating and Using a Static Library

Creates a static library and an application that references it.

Attributes Walkthroughs

Walkthrough: Creating a COM Server Using a Text Editor

Demonstrates how to quickly develop a simple COM server using various attributes with a text editor.

Walkthrough: Creating a COM Server Using Wizards

Demonstrates how to quickly develop a simple COM server using various attributes with Visual C++ wizards.

Walkthrough: Creating an ActiveX Control with Attributes

Uses attributes and wizards to create a simple ActiveX control.

Deployment Walkthroughs

Walkthrough: Deploying a Windows-based Application

Guides you through deploying an application to another computer.

Walkthrough: Installing Shared Components Using Merge Modules

Guides you through packaging a component in a merge module and then including the merge module in an installer.

Walkthrough: Creating a Custom Action

Guides you through creating a custom action to send a user to a Web site following installation.

Walkthrough: Using a Custom Action to Display a Message at Installation

Demonstrates passing data to a dynamic property during installation using a custom action and the **CustomActionData** property.

Walkthrough: Using a Custom Action to Pre-Compile an Assembly at Installation

Guides you through creating a custom action to precompile an assembly after installation.

Walkthrough: Using a Custom Action to Create a Database at Installation

Guides you through creating a custom action to create a database during installation.

Redirecting an Application to Target a Different XML Web Service During Installation

Demonstrates how to create a Web application that can be redirected to target a different XML Web service by using the **URL Behavior** property, an Installer class, and a Web Setup project.

Libraries Walkthroughs

ATL Tutorial

Leads you through the creation of a control and demonstrates some ATL fundamentals in the process.

Attributes Tutorial

Leads you through the creation of a client and a server application using attributes and events.

ATL Server Tutorial

Shows how to create a simple online store that accesses a database. Covers some basic and advanced features of ATL Server including handling forms, validating user input using regular expressions, creating and using cookies, exposing statistics as performance counters, creating dynamic services, using cryptography, and more.

Walkthrough: Creating a Rich Client Application with MFC

Demonstrates how to create a rich-client interface using the Microsoft Foundation Class (MFC) Library, a well-tested class library that supports a broad range of features common to all Windows applications, and consume an XML Web service created with Visual Basic or Visual C#.

Windows Forms Walkthroughs

Walkthrough: Creating a Simple Windows Form

Demonstrates how to make a simple "Hello, World" application.

Walkthrough: Retrieving Dialog Box Information Collectively Using Objects

Gives directions for using an object to expose a group of related data from a dialog box.

Walkthrough: Updating Status Bar Information at Run Time

Explains how to programmatically control the data within status-bar panels.

Walkthrough: Debugging a Windows Form

Gives step-by-step instructions for creating a Windows Form and debugging that form.

XML Web Services Walkthroughs

Walkthrough: Creating an XML Web Service Using ATL Server

Describes the process for creating an XML Web service that converts temperatures measured in Fahrenheit to Celsius using Visual C++ and ATL Server.

Walkthrough: Creating an XML Web Service Using C++ and the CLR

Describes creating an XML Web service using C++ and the ASP.NET Web Service project template, implementing the XML Web service, and deploying the XML Web service.

Walkthrough: Accessing an XML Web Service Using C++

Describes the process for accessing an XML Web service from an application created using Visual C++.

Walkthrough: Accessing an XML Web Service Using C++ and the CLR

Describes the process for accessing an XML Web service from an application created using C++.

Related Sections

Visual C++ Sample Applications

Provides links to sample code showing the capabilities of Visual C++ and the libraries and technologies that it supports.

See Also

Other Resources

Visual C++ Samples

Visual C++ Sample Applications

This section lists the sample applications included with Visual C++. For more information on how to download the samples, see How to: Download Sample Files in Help Topics.

In This Section

General Samples

New Walkthroughs in Visual C++ 2005

STL Samples

ATL Samples

ATL Server Samples

Compiler COM Support Samples

Custom Wizard Samples

Event Handling Samples

Extensibility

International Samples

Interoperability Samples

MASM Samples

MFC Samples

Platform SDK Samples

See Also

Other Resources

Visual C++ Samples

General Samples

This page contains links to samples that work with all versions of Visual C++, including Visual C++ 2005 Express Edition.

In This Section

ScribbleDotNET Sample: Extending Scribble with CLR Features

A collection of samples based on the classic Scribble sample, updated and extended to include new .NET functionality.

YukonCLR Sample: Upgrading Stored Procedures to Work with SQL Server 2005

This sample demonstrates how to leverage an existing extended stored procedure and how to use C++ for developing CLR code.

SecureCRT Sample: Converting Deprecated CRT Functions to their _s Counterparts

This sample demonstrates how to upgrade code that used deprecated CRT functions to increase code security.

CppLangFeat Sample: Language Features Overview

This sample demonstrates key concepts of new features in C++.

COMEXCEL Sample: Demonstrates an Automation Client Application with Compiler COM Support

A standalone Automation client that creates a new Microsoft Excel spreadsheet and generates a pie chart of the data stored on that spreadsheet.

COMMAIL Sample: Demonstrates an Automation Client Application with Compiler COM Support

Demonstrates an Automation client application with MAPI for Microsoft Exchange (or Outlook) and compiler COM support.

FREETHRD Sample: Multithreaded Client and Free-Threaded Server with Compiler COM Support

Demonstrates a multithreaded client and free-threaded server with compiler COM support.

ccWrapper Sample: Demonstrates How to Map C/C++ Compiler Flags from Other Compilers to Visual C++

Demonstrates how to map C/C++ compiler flags from other compilers to Visual C++.

CPUID Sample: Determines CPU Capabilities

Determines the capabilities of the CPU being run.

crt_dbg1 Sample: C Run-Time Basic Debugging Features

Illustrates the basic debugging features of the C run-time libraries.

crt_dbg2 Sample: C Run-Time Debugging Hook Functions

Demonstrates the C run-time debugging hook functions.

dfacobjs Sample: C Run-Time _CrtDoForAllClientObjects Function

Shows how to use the **_CrtDoForAllClientObjects** C run-time function to iterate through a linked list of client objects.

report Sample: C Run-Time Debugging Report Functions

Illustrates the C run-time debugging report functions.

RTC Sample: Run-Time Error Checks

Demonstrates the run-time error checks feature.

EEAddIn Sample: Debugging Expression Evaluator Add-In

Uses the Expression Evaluator Add-In API to extend the native debugger expression evaluator.

SatDLL Sample: Implements Multilingual Resources in a Win32 Application

Demonstrates a recommended way to implement multilingual resources in a Win32 application.

Unires Sample: Demonstrates the Use of Unicode Resource Files

Demonstrates the use of Unicode resource files.

Data Sample: Demonstrates Simple Access to a SQL Database

Demonstrates a simple query from an existing SQL database using the console.

BirthdayPicker Sample: Demonstrates .NET Framework Resources with Windows Forms

Shows how the .NET Framework resource mechanism can be used in C++ applications. It also demonstrates some common Window Forms components.

Calculator Sample: Windows Forms Pocket Calculator

Implements a simple pocket calculator using C++ and the .NET Framework Windows Forms classes.

Scribble Sample: MDI Drawing Application

Demonstrates how to develop a Windows Forms MDI application using C++ and the .NET Framework classes.

See Also

Other Resources

Visual C++ Sample Applications

ScribbleDotNET Sample: Extending Scribble with CLR Features

Download sample

In this collection of samples the classic Scribble sample is updated and extended to include new .NET functionality. The sample files include two solutions, one with an MFC implementation of the Scribble sample and the other with a Window Forms implementation of the Scribble sample. Each of the feature areas that are demonstrated in the sample solutions are described in the following sample descriptions:

Walkthrough: Exposing a .NET Plug-in Model

The MFC Scribble solution is extended to expose a .NET object model. Furthermore, the sample also exposes a plug-in architecture that enables .NET plug-ins to interact seamlessly with the MFC Scribble solution.

Walkthrough: Using /clr:pure Features

Windows Forms Scribble solution is extended to include some native classes that contain static data. The Windows Forms Scribble solution also exposes Windows Forms controls that must be usable from the Windows Forms designer. Building the sample with the /clr:pure option addresses potential issues with those scenarios.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

To build this sample

- See the above walkthroughs for the correct procedure to build the application.

To run this sample

- See the above walkthroughs for the correct procedure to run the application.

See Also

Other Resources

General Samples

YukonCLR Sample: Upgrading Stored Procedures to Work with SQL Server 2005

Download sample

This sample demonstrates how to leverage an existing extended stored procedure and how to use C++ for developing CLR code. It comprises two solutions:

- CppXP2Clr, which contains two projects:
  - xp_cheksum, an extended stored procedure with C++ functions
  - clr_checksum, a port using CLR pure
- CppUdt is a CLR safe SQL Server 2005 project that contains user-defined data type

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

To build the sample

- On the **Build** menu, select the **Build Solution** menu item.

To run the sample

- On the **Debug** menu, select the **Start** menu item.

Requirements

This sample requires the AdventureWorks sample database on a SQL Server 2005 server. After you load a SQL Server Class Library application in the Visual Studio integrated development environment (IDE), a message box will prompt you to specify connection information to a SQL Server. For this sample application, specify the AdventureWorks database.

Demonstrates

How to:

- Use an existing extended stored procedure
- Move existing stored procedure functions to SQL Server 2005
- Use new user-defined data types in SQL Server 2005

How it Works

Yukon/CLR consists of two solutions, CppXP2Clr and CppUdt.

Solution CppXP2Clr consists of two projects, an extended stored procedure and a SQL Server Class Library project. The SQL/CLR project demonstrates the simplicity of moving native C++ function to SQL/CLR. The function checksum() will be used in both projects. The SQL/CLR project will be built using /clr:pure to leverage existing C++ functions.

Solution CppUdt is a SQL/CLR project built with /clr:safe and creates a user-defined data type (UDT). This project demonstrates using C++ to build SQL/CLR project with UDT.

See Also

Other Resources

General Samples

SecureCRT Sample: Converting Deprecated CRT Functions to their _s Counterparts

Download sample

This sample demonstrates how to convert CRT functions which have been deprecated for security reasons to their _s counterparts. It covers the conversion of the following functions: **strcpy**, **strncpy**, **fopen**, **_splitpath**, **sprintf**, **getenv**, **strcat**, **_creat**, and **itoa**. It also demonstrates the use of **_set_invalid_parameter_handler**, another Secure CRT addition.

This sample implements the Boyer-Moore string matching algorithm using standard CRT functions.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

To build and run this sample

1. Open the solution file, SecureCRT.sln, in the Visual Studio development environment.
2. From the **Build** menu, click **Build Solution**.
3. From the **Projects** menu, click **Properties**.
4. Under **Configuration Properties**, select **Debugging**.
5. Specify the arguments to pass to the program in the **Command Arguments** text box ("if find.cpp" is a good first choice).
6. From the **Debug** menu, click **Start Without Debugging**. The application will search each line of the file (find.cpp, in this case) for the text you specified ("if") in the Command Arguments and write all matches to a file (find.cpp.match, in this case) in the working directory.
7. This application can also be run from the command line: SecureCRT.exe if find.cpp

Keywords

strcpy_s, **strncpy_s**, **fopen_s**, **_splitpath_s**, **sprintf_s**, **getenv_s**, **strcat_s**, **itoa_s**, **_set_invalid_parameter_handler**

See Also

Reference

strcpy_s, wcscpy_s, _mbscpy_s
strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l
fopen_s, _wfopen_s
_splitpath_s, _wsplitpath_s
sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l
getenv_s, _wgetenv_s
strcat_s, wcscat_s, _mbscat_s
_itoa_s, _i64toa_s, _ui64toa_s, _itow_s, _i64tow_s, _ui64tow_s
_set_invalid_parameter_handler

Other Resources

General Samples

CppLangFeat Sample: Language Features Overview

Download sample

This sample demonstrates key concepts of new features in C++. These features are primarily added to support .NET and expose capabilities of the CLR. The sample presents short snippets of code for each language feature, which shows common ways to use the C++ syntax. The sample is meant to be built and run. For best results, using debugger breakpoints to walk through the code will help with understanding how new C++ features work.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

To build and run this sample

1. Open the solution file, C++ Language Samples.sln, in the Visual Studio development environment.
2. From the **Build** menu, click **Build Solution**.
3. From the **Debug** menu, click **Start**.

Demonstrates

Generics, finalization, and partial template specialization.

How it Works

Each file in the sample shows off a new feature of the C++ language. When the sample is run, each feature is used in turn, and output from that file is displayed.

See Also

Other Resources

General Samples

COMEXCEL Sample: Demonstrates an Automation Client Application with Compiler COM Support

Download sample

COMEXCEL is a standalone Automation client. It creates a new Microsoft Excel spreadsheet and generates a pie chart of the data stored on that spreadsheet. This sample works only with Microsoft Excel, which exposes the dispinterfaces.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Make sure that the #import reference to COMEXCEL.exe in the sample's source code is correct for your computer.

To build this sample within Visual C++, determine which version of Excel you have on your computer and then look in the appropriate COMEXCEL sample directory.

If you get an access violation while running the COMEXCEL sample under Windows NT 4.0, install the Windows NT Service Pack 3.

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution comexcel.sln.
2. If you are using Excel 2000, modify the following line in comexcel.cpp:

   ```
   #define OFFICE_VER OFFICEXP
   ```

   to instead be:

   ```
   #define OFFICE_VER OFFICE2000
   ```

3. On the **Build** menu, click **Build Solution**.
4. On the **Debug** menu, click **Start Without Debugging**.

Keywords

This sample demonstrates the following keywords:

dispinterface; #import; _com_ptr_t; _variant_t; _bstr_t; _com_error

See Also

Reference

Compiler COM Support

Other Resources

Compiler COM Support Samples

General Samples

COMMAIL Sample: Demonstrates an Automation Client Application with Compiler COM Support

Download sample

COMMAIL is a standalone Automation client. It automates MAPI for Microsoft Exchange 4.0 (or later) or Microsoft Outlook. This sample sends an e-mail message which contains its source code to a specified e-mail recipient.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build this sample with Microsoft Office XP, be sure that the macro **OfficeXP** is defined in your code as follows: `#define OFFICEXP 1`.

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution commail.sln.
2. Modify paths and file names in the source code by following the instructions given in the `//To DO` lines in commail.cpp.
3. Change the recipient e-mail name and other e-mail parameters. Search the source code for `#pragma` for instructions on what you need to modify.
4. Add the directory path to olemsg32.dll (Office 97) or cdo.dll (Office 2000 and Office XP). You can add this file to your Include files path in the VC++ Directories dialog box.
5. On the **Build** menu, click **Build**.
6. On the **Debug** menu, click **Start Without Debugging**.

Look at the type library header files (.tlh and .tli). Compare the COM interfaces in the header files with interfaces of the same type library displayed by OLEVIEW.

Keywords

This sample demonstrates the following keywords:

dispinterface; #import; _com_ptr_t; _variant_t; _bstr_t; _com_error

See Also

Reference

Compiler COM Support

Other Resources

Compiler COM Support Samples

General Samples

FREETHRD Sample: Multithreaded Client and Free-Threaded Server with Compiler COM Support

Download sample

This sample demonstrates a multithreaded client and free-threaded server with compiler COM support.

This sample consists of the following parts:

- Freclien, a multithreaded client
- Freserve, a free-threaded in-process server

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution freethrd.sln.
2. Build the server project and then build the client project.
3. Make the client project the startup project (right-click the project node and click **Set as StartUp Project**) and run the sample.

How the Sample Works

The Freclien sample spawns multiple threads to create and use the `COBall` COM object provided by the Freserve free-threaded server. The `COBall` object itself spawns no threads; instead, it passively responds to `IBall` interface requests from many possible client threads. The Freclien client creates and controls one `COBall` object through the `IBall` interface that the object exposes. As three of Freclien's threads move the ball through calls to `IBall::Move`, the remaining main thread uses a system timer to obtain timed updates of the `COBall` object's position, size, and color. This main thread uses that data, obtained by calling `IBall::GetBall`, to display graphical snapshot images of the ball in the client's main window.

In the Freserve sample, the `COBall` object internally updates its color property to reflect the last thread that called the object's `Move` method. The display thread uses this data for each ball image it displays. As the ball moves, it changes color to reflect each thread that moves the ball. As the ball moves, it also leaves a trail that provides a striking visual history of these passing threads. This trail demonstrates that, with COM's free-threading model, every thread that makes interface requests to the same object accesses the object on the calling thread. Each different color of the single ball object represents a different calling thread.

This sample uses native compiler COM support. It gives an example of a custom COM interface that marshals the RECT and POINT structures with the free-threaded model. It demonstrates the use of the **exclude** attribute of the **#import** directive.

See Also

Reference

Compiler COM Support

Other Resources

Compiler COM Support Samples

General Samples

ccWrapper Sample: Demonstrates How to Map C/C++ Compiler Flags from Other Compilers to Visual C++

Download sample

Download Sample

ccWrapper shows how to programmatically map flags from other compilers to the Visual C++ compiler. The included configuration file is used only as an example, not as an official way of showing how the flags map from one architecture to another, and is configurable so that you can change the configuration. For more information, see the README.htm file.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To run this sample

1. Open the solution GccWrapper.sln.
2. On the **Build** menu, click **Build Solution**.
3. At the command prompt, copy the GccWrapper and config files to the TinyXML folder:

   ```
   copy Release\GccWrapper.exe Examples\TinyXML\gcc
   copy ccFile.cgf Examples\TinyXML
   ```

4. At the command prompt, move to the TinyXML folder:

   ```
   cd Examples\TinyXML
   ```

5. Run Make.exe by entering `make` at the command prompt.

   Note that you will receive multiple errors because **/Wall** is set, but xmltest.exe is being built.

6. Run xmltest.exe by entering `xmltest.exe` at the command prompt.

See Also

Other Resources

General Samples

CPUID Sample: Determines CPU Capabilities

Download sample

The CPUID sample provides a routine that uses the CPUID instruction to determine the capabilities of the CPU being run.

The sample provides the function `int _cpuid(_p_info *pinfo)`, which returns data about the CPU. The `int` return value is a bitmask of flags for major processor features. The bits that might be set are:

- `#define _CPU_FEATURE_MMX 0x0001`
- `#define _CPU_FEATURE_SSE 0x0002`
- `#define _CPU_FEATURE_SSE2 0x0004`
- `#define _CPU_FEATURE_3DNOW 0x0008`

Security Note

This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Note

If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution cpuid.sln.
2. From the **Build** menu, click **Build**.
3. From the **Debug** menu, select **Start Without Debugging**.

Example Program Output

The sample includes a test.cpp file that trivially calls `_cpuid` and outputs the values in the resulting `_p_info` struct. For example, on a Pentium III computer that supports MMX and SSE, the program output would look something like this:

```
C:\work\cpuid>test
v_name:         GenuineIntel
model:          INTEL Pentium-III
family:         6
model:          8
stepping:       3
feature:        00000003
        yes     _CPU_FEATURE_MMX
        yes     _CPU_FEATURE_SSE
        no      _CPU_FEATURE_SSE2
        no      _CPU_FEATURE_3DNOW
os_support:     00000003
        yes     _CPU_FEATURE_MMX
        yes     _CPU_FEATURE_SSE
        no      _CPU_FEATURE_SSE2
        no      _CPU_FEATURE_3DNOW
checks:         0000000f
```

See Also

Other Resources

crt_dbg1 Sample: C Run-Time Basic Debugging Features

Download sample

The crt_dbg1 sample illustrates the basic debugging features of the C run-time libraries and the kind of debug output that these features generate.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution crt_dbg1.sln.
2. From the **Build** menu, click **Build**.
3. From the **Debug** menu, select **Start Without Debugging**.
4. To see the output of the program as it progresses, set breakpoints in the code and then view the output in the program's command window. Or, run Debug\crt_dbg1.exe from the command line.

Keywords

This sample uses the following keywords:

_asserte; _crtcheckmemory; _crtdumpmemoryleaks; _crtmemcheckpoint; _crtmemdifference; _crtmemdumpallobjectssince; _crtmemdumpstatistics; _crtsetdbgflag; _crtsetreportfile; _crtsetreportmode; _malloc_dbg; _rpt1; _rpt2; aboutbox; clear_crt_debug_field; createinstance; displaystring; free; get_size; helpstring; id; iid_is; malloc; module; outputheading; pointer_default; put_size; set_crt_debug_field; strcmp; strcpy; uuid

See Also

Other Resources

General Samples

crt_dbg2 Sample: C Run-Time Debugging Hook Functions

Download sample

The crt_dbg2 sample illustrates several ways to use debugging hook functions with the debug versions of the C run-time library. To add some realism, it has a few elements of an actual application, including two bugs.

Note
This sample is not supported on Intel's Itanium processors.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution crt_dbg2.sln.
2. From the **Build** menu, click **Build**.
3. From the **Debug** menu, select **Start Without Debugging**.

How the Sample Works

The program stores birthdate information in a linked list of Client blocks. A Client-dump hook function validates the birthday data and reports the contents of the Client blocks. An allocation hook function logs heap operations to a text file, and the report hook function logs selected reports to the same text file.

Note that the allocation hook function explicitly excludes CRT blocks (the memory allocated internally by the C run-time library) from its log. The hook function uses fprintf to write to the log file, and **fprintf** allocates a CRT block. If CRT blocks were not excluded in this case, an endless loop would overflow the stack: **fprintf** would cause the hook function to be called, the hook would in turn call **fprintf**, which would in turn cause the hook to be called again, and so forth.

To be able to report CRT-type blocks in your allocation hook, Windows API functions could be used instead of C run-time functions. Because the Windows APIs do not use the CRT heap, they would not trap the hook in an endless loop.

The debug heap catches two bugs and a data error in the second example. One bug is that the birthday name field is not large enough to hold several of the test names. The field should be larger, and strncpy should be used instead of strcpy. The second bug is that the while loop in the `printRecords` function should not end until the `HeadPtr` itself is equal to null. This bug results not only in an incomplete display of birthdays, but also in a memory leak. Finally, Gauss's birthday should be April 30, not April 32.

Keywords

This sample uses the following keywords:

_assert; _asserte; _crtcheckmemory; _crtmemcheckpoint; _crtmemdumpallobjectssince; _crtmemdumpstatistics; _crtsetallochook; _crtsetdbgflag; _crtsetdumpclient; _crtsetreportfile; _crtsetreporthook; _crtsetreportmode; _free_dbg; _malloc_dbg; _rpt4; _strdate; _strtime; aboutbox; clear_crt_debug_field; createinstance; createrecord; displaystring; donttouch; exit; fatalerror; fclose; fflush; fopen; fprintf; fputs; get_size; helpstring; id; iid_is; module; myallochook; mydumpclienthook; myreporthook; pointer_default; printf; printrecords; put_size; set_crt_debug_field; strcpy; strstr; uuid

See Also

Other Resources

General Samples

dfacobjs Sample: C Run-Time _CrtDoForAllClientObjects Function

Download sample

The dfacobjs sample shows how to use the **_CrtDoForAllClientObjects** function to iterate through a linked list of client objects.

Note
This sample is not supported on the Intel Itanium processors.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution dfacobjs.sln.
2. From the **Build** menu, click **Build**.
3. From the **Debug** menu, select **Start Without Debugging**.

Keywords

This sample uses the following keywords:

_crtdoforallclientobjects; _crtismemoryblock; _crtsetdbgflag; _free_dbg; _malloc_dbg; aboutbox; createinstance; creatememoryblock; displaystring; div; donttouch; exit; get_size; helpstring; id; iid_is; module; myclientobjecthook; pointer_default; printf; put_size; restorememorytoheap; uuid

See Also

Other Resources

General Samples

report Sample: C Run-Time Debugging Report Functions

Download sample

The report sample illustrates several ways to use the debug report functions of the C run-time libraries.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution report.sln.
2. From the **Build** menu, click **Build**.
3. From the **Debug** menu, select **Start Without Debugging**.
4. To see the output of the program as it progresses, set breakpoints in the code and then view the output in the program's command window. Or, run Debug\report.exe from the command line.

Keywords

This sample uses the following keywords:

_assert; _asserte; _crtsetreportfile; _crtsetreporthook; _crtsetreportmode; _rpt0; _rpt2; _rptf2; aboutbox; createinstance; displaystring; fflush; fprintf; free; get_size; helpstring; id; iid_is; malloc; module; ourreportingfunction; pointer_default; put_size; strcpy; uuid

See Also

Other Resources

General Samples

RTC Sample: Run-Time Error Checks

Download sample

The RTC sample shows how to use the C run-time library's run-time error check feature.

When you load this project into the development environment, you will notice two custom build configurations:

No CRT

Does not use the C run-time library and, therefore, uses customized error reporting.

Normal

Uses the C run-time library and its error reporting mechanism.

Notice that the rtcsamp.cpp file contains the code that causes the bugs that are reported.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution rtcsample.sln.
2. From the **Build** menu, click **Build**.
3. From the **Debug** menu, select **Start Without Debugging**.

If you debug this project, you will see the Visual Studio debugger support for run-time error checks. That is, you will see the debugger's error reporting mechanism. The debugger's support for reporting run-time error checks is independent of whether you use the C run-time library.

Additional Information

For more information about run-time error checks, see the following topics:

- /RTC
- Using Run-Time Error Checks
- runtime_checks pragma
- Run-Time Error Checking

Keywords

This sample uses the following keywords:

_RTC_error_fn; _RTC_ErrorNumber; _crt_rtc_init; _rtc_geterrdesc; _rtc_initialize; _rtc_seterrorfunc; _rtc_terminate; catch_rtc_failure; defined; interlockedexchange; intrinsic; messagebox; sleep; sprintf; strcat; strcpy; va_arg; va_end; va_start; vsprintf; winmain

See Also

Other Resources

General Samples

EEAddIn Sample: Debugging Expression Evaluator Add-In

Download sample

The EEAddin sample shows how to extend the native debugger expression evaluator using the Expression Evaluator Add-In API.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

EE Add-In API

The expression evaluator is the part of the debugger that interprets (evaluates) expressions. When you set a breakpoint on an expression or type an expression in a debugger window, the expression evaluator interprets the input. For details, see Expressions in the Debugger. With the Expression Evaluator Add-In API, you can extend the expression evaluator to handle new types.

To extend the expression evaluator for a new type, you need to write a function as part of a Win32 DLL (in the same directory as autoexp.dat) and export it by name. You also need to add a line to the autoexp.dat file. You can extend the expression evaluator for more than one type by exporting multiple functions from the DLL.

Building and Running the Sample

The steps required to build and run this sample fall into three parts.

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run the sample

1. Build an expression evaluator Add-In DLL (eeaddin.dll).
2. Edit autoexp.dat to use the expression evaluator Add-In DLL.
3. Test the Add-In by creating a project that uses the custom data type evaluated by autoexp.dat.

The following procedures explain these steps in detail.

To build the expression evaluator Add-In DLL

1. In Visual Studio, open the solution eeaddin.sln.
2. From the **Build** menu, click **Build**.
3. Copy the resulting eeaddin.dll to your common7\ide directory (the same directory that contains devenv.exe).
4. From the **File** menu, click **Close Solution**.

To edit autoexp.dat

1. From the **File** menu, point to **Open** and click **File**.
2. In the **Open File** dialog box, find the file autoexp.dat (in the common7\packages\debugger directory) and click **Open**.
3. Edit autoexp.dat to add the following lines:

```
_SYSTEMTIME=$ADDIN(eeaddin.dll,AddIn_SystemTime@28)
_FILETIME=$ADDIN(eeaddin.dll,AddIn_FileTime@28)
```

Save autoexp.dat.

To create a project that uses the custom data types

1. From the **File** menu, point to **New** and click **Project**.
2. In the **New Project** dialog box, highlight **Visual C++ Projects**, click **MFC Application**, enter a name for the project, and click **OK**.
3. In the **MFC Application Wizard**, click **Finish**. The project must be an MFC application because in the next step you will add MFC functions.
4. In the MFC application, add a SYSTEMTIME or FILETIME object.

```
SYSTEMTIME *s = new SYSTEMTIME();
FILETIME *f = new FILETIME();
GetSystemTime(s);
SystemTimeToFileTime(s,f);
```

5. From the **Build** menu, click **Build**.
6. Start debugging and examine your **SYSTEMTIME** or **FILETIME** objects in the watch window.

How the Sample Works

To extend the expression evaluator for a custom data type, you write a custom viewer function in the expression evaluator Add-In DLL. The function uses a pointer to an object in the memory space of the program being debugged (not the memory space of the expression evaluator you are extending). You cannot use normal casts with this pointer. You must read it and the data it points to using a callback function. A callback pointer of type `DEBUGHELPER*` points to an object with various methods.

The syntax looks like this:

```
HRESULT WINAPI CustomViewer(
   DWORD dwAddress,       // low 32-bits of address
   DEBUGHELPER *pHelper,  // callback pointer to access helper functions
   int nBase,             // decimal or hex
   BOOL bIgnore,          // not used
   char *pResult,         // where the result needs to go
   size_t max,            // how large the above buffer is
   DWORD dwReserved       // always pass zero
)
```

The sample has two implementations of this type of function, `AddIn_SystemTime` and `AddIn_FileTime` in timeaddin.cpp. The `DEBUGHELPER` struct (defined in custview.h) consists of function pointers that can assist you in writing your extension. This pointer is passed to your `CustomViewer` function, and you can use it to call the helper functions.

You can get the processor type with `pHelper->GetProcessorType`. There are two methods for reading memory, `pHelper->ReadDebuggeeMemory` and `pHelper->ReadDebuggeeMemoryEx`. `ReadDebuggeeMemoryEx` handles 64-bit addresses and is supported by the Visual Studio .NET debugger. `ReadDebuggeeMemory` does not handle 64-bit addresses and is supported by the Visual Studio .NET and Visual C++ 6.0 debuggers. If your Add-In is designed for the Visual Studio .NET debugger only, you can use `ReadDebuggeeMemoryEx`. If your Add-In needs to work with Visual C++ 6.0 also, you must check the `dwVersion` field and avoid calling `ReadDebuggeeMemoryEx` for Visual C++ 6.0.

The following code works with both debuggers and reads the contents of a *localobject* (whose type is `MyType`) from the program being debugged:

```
DWORDLONG qwRealAddress;
DWORD dwGot;
MyType localobject;
if (pHelper->dwVersion<0x20000)
{
   // Visual C++ 6.0 version
   qwRealAddress = dwAddress;
   pHelper->ReadDebuggeeMemory( pHelper, dwAddress,
      sizeof(localobject), &localobject, &dwGot );
```

```
}
else
{
   qwRealAddress = pHelper->GetRealAddress(pHelper);
   pHelper->ReadDebuggeeMemoryEx( pHelper, qwRealAddress,
      sizeof(localobject), &localobject, &dwGot );
}
// TODO: display localobject here
```

Editing autoexp.dat

In the `[AutoExpand]` section of autoexp.dat, the lines you will add have the following syntax:

```
type=$ADDIN(dllname.dll,exportname)
```

For example:

```
_SYSTEMTIME=$ADDIN(eeaddin.dll,AddIn_SystemTime)
```

or:

```
_FILETIME=$ADDIN(eeaddin.dll,AddIn_FileTime)
```

If the DLL is not in the directory containing devenv.exe or on the PATH, you must use a full path name for the DLL. The *exportname* argument is case sensitive and must exactly match the exported name you got when you ran `dumpbin -exports` on your DLL.

To test the debugger with the new Add-In, first stop debugging any program you were debugging when you installed the new DLL, and then start a new debugger session.

Note The Add-In runs within the debugger, so if your code crashes, you will crash the IDE.

See Also

Other Resources

General Samples

SatDLL Sample: Implements Multilingual Resources in a Win32 Application

Download sample

The SatDLL sample demonstrates a number of things related to localization and globalization:

- How to set up a solution that builds one main EXE file and individual satellite DLLs containing different language versions of the user interface.
- The recommended way to implement a satellite DLL-loading mechanism with language fallback in case the selected language isn't available.
- Code to detect the preferred language for the user interface on any version of Windows.
- How to dynamically switch the user interface language upon a change request from a user (what is not demonstrated is how the user choice for the user interface language can be persisted).
- How to use the generic character encoding mapping functions to be able to build ANSI and Unicode versions of an application from the same source code.

Security Note

This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Note

If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution file satdll.sln.
2. From the Build menu, click **Build Solution**.
3. From the **Debug** menu, click **Start Without Debugging**.

In the default configuration, this solution builds a Unicode application which requires Windows 2000 or later with language support for Western European languages and Japanese installed to display all characters in the user interface correctly.

By changing the Character Set in the general project settings to "Use Multi-Byte Character Set" the sample can be compiled for earlier Windows platforms. You will only be able to switch between languages which are supported by the ANSI codepage for that earlier Windows version though (for example, you will not be able to display Japanese on a English version of Windows ME).

Note

When debugging this sample, the current directory is the solution directory and not the Debug directory. For this reason, the directories for the satellite DLLs (1031, 1033, and 1041) are created in the solution directory. When you deliver your application to an end-user, these satellite DLL directories should, of course, be created in the directory where the main executable file is located.

Keywords

This sample demonstrates the following keywords:

LoadString; LoadMenu; LoadAccelerators; LoadIcon; LoadCursor; DialogBox; CreateFontIndirect; DrawText; InvalidateRect;

UpdateWindow; DestroyMenu; SetMenu; DrawMenuBar; GetLocaleInfo; GetCurrentDirectory; FindFirstFile; FindNextFile; LoadLibrary; EnumResourceLanguages; GetVersionInfoEx; RegOpenKeyEx; RegQueryValueEx; RegCloseKey; GetUserDefaultUILanguage; GetUserDefaultLangID; _tWinMain; _tcscpy; _tcsncpy; _tcslen; _tcsclen; _tcscat; _ttoi; _itot

See Also

Concepts

Localized Resources in MFC Applications: Satellite DLLs

Other Resources

International Samples

General Samples

Unires Sample: Demonstrates the Use of Unicode Resource Files

Download sample

The Unires sample demonstrates how to create a Win32 application with only Unicode string resources and how to integrate a Unicode .rc file into a project. As the resource editor only supports ANSI in Visual Studio .NET, modifications to the .rc file have to be made in the source editor.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

Running this application requires an operating system that allows the display of Unicode windows — that is, Windows 2000 or later. It also requires that you have language support for all possible Windows languages installed (check in the Control Panel under Regional Options). The application can be built on any Windows version supported by Visual Studio .NET.

Building and Running the Sample

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution file unires.sln.
2. From the **Build** menu, click **Build**.
3. From the **Debug** menu, click **Start Without Debugging**. You will get a window with the sentence "When the world wants to talk, it speaks Unicode." displayed in 41 languages.

Keywords

This sample demonstrates the following keywords:

LoadString; CreateFontIndirect; GetTextMetrics; ScrollWindowEx; SetScrollInfo

See Also

Other Resources

International Samples
General Samples

Data Sample: Demonstrates Simple Access to a SQL Database

Download sample

The Data sample demonstrates a simple query from an existing SQL database using the console. The result of the query is displayed and the program exits.

Note
The sample assumes an existing SQL server and table. To run this sample to completion, you must provide a SQL server, a database, and a table.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run Data using Visual Studio

1. In the Visual Studio IDE, load the solution file data.sln.
2. In Solution Explorer, right-click the **Data** solution.
3. On the shortcut menu, click **Build**.
4. On the **Debug** menu, click **Start**.

Classes and Keywords

This sample demonstrates the following classes:

SQLDataReader; DataSet

This sample demonstrates the following keywords:

SQLDataReader; DataSet; SQL; Data

See Also

Other Resources

Interoperability Samples
General Samples

BirthdayPicker Sample: Demonstrates .NET Framework Resources with Windows Forms

Download sample

The BirthdayPicker sample shows how the .NET Framework resource mechanism can be used in C++ applications. It also demonstrates some common Window Forms components.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To take advantage of .NET Framework resources, C++ applications can use the .NET Framework tool WinRes.exe. This tool is specialized in creating ".resx" files (the equivalent of ".rc" files in Win32 C++ applications). Birthdaypicker.resx contains the `imageList1` resource along with the collection of bitmaps it handles. The resultant .resx file is then compiled by resgen.exe into the file birthdaypicker.resources, which is then linked into the resulting assembly of the application.

To build and run BirthdayPicker using Visual Studio

1. In the Visual Studio IDE, load the solution file BirthdayPicker.sln.
2. In Solution Explorer, right-click the **BirthdayPicker** solution.
3. On the **Debug** menu, click **Start**.

   Observe the Custom build step, running resgen.exe on birthdaypicker.resx and generating the resource file for the application. In addition, note the linker option **/assemblyresource**, used to link the resulting birthdaypicker.resources to the application.
4. On the **Debug** menu, click **Start**.

Once the sample is running, two DateTimePicker controls are displayed. Select your birthday using the first control and an arbitrary date (such as today) with the second control. Once the date range has been selected, the linked tree control is propagated with multiple nodes interpreting the timespan in different ways. For example, the number of years or days elapsed, the equivalent in dog years or in fruit fly generations, and how many full moons have passed in this time interval.

Classes and Keywords

This sample demonstrates the following classes:

DateTimePicker; ImageList; ResourceManager; TreeView

This sample demonstrates the following keywords:

ResourceManager::GetObject; DateTimePicker::Format; TimeSpan; KeyPressEventHandler; ToolTip::SetToolTip; ContextMenu; TreeView::Nodes; TreeView::SelectedNode; TreeNode::GetNodeCount; TreeNode::Remove; TreeNodeCollection::Insert; TreeNodeCollection::Clear; String::Format; Panel; dynamic_cast

See Also

Other Resources

General Samples

Calculator Sample: Windows Forms Pocket Calculator

Download sample

The Calculator sample implements a simple pocket calculator using C++ and the .NET Framework Windows Forms classes. It demonstrates using .NET Framework classes to write a user-interface without the support of the resource editors, and illustrates the following concepts:

- Working with forms, menus, edit fields, buttons, and icons for forms
- Manipulating managed arrays
- Using delegates/event handlers
- Exception handling
- Using enums
- Conversions between the .NET Framework **String**, **Double**, and **Char** classes

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run Calculator using Visual Studio

1. In the Visual Studio IDE, load the solution file calc.sln.
2. In Solution Explorer, right-click the Calc solution.
3. On the shortcut menu, click **Build**.

Note
Since this program does not compile the icon file (calc.ico) into a XML-based resource file (*.resx), you must copy the icon file into the same directory as the .exe (built from the sample) before continuing.

4. Run the resulting application and try out various operations with the calculator.

Classes and Keywords

This sample demonstrates the following classes:

Form; TextBox; Button; Icon; MainMenu; MenuItem; MessageBox; Exception; FileNotFoundException; Size structure; Font

This sample demonstrates the following keywords:

__gc; __value; enum; add_KeyPress; KeyPressEventHandler; EventHandler; __try_cast; EventArgs; KeyPressEventArgs; FormStartPosition; FormBorderStyle; SizeGripStyle; BorderStyle; SystemColors; HorizontalAlignment; FlatStyle; add_Click

See Also

Other Resources

General Samples

Scribble Sample: MDI Drawing Application

Download sample

The Scribble sample is based on the original MFC Scribble. The sample demonstrates how to develop a Windows Forms MDI application using C++ and the .NET Framework classes. Scribble is a small drawing application that lets you draw free-hand drawings using the mouse and save those images in a file.

The sample contains three objects of interest:

- **Document object** Manages the user-created documents and performs basic opening and saving operations.
- **View object** Provides the view for the document object. The user can have different views of the same document.
- **Main form object** The equivalent of the standard MFC frame object. This object is the parent for all of the views.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run Scribble using Visual Studio

1. In the Visual Studio IDE, load the solution file scribble.sln.
2. In Solution Explorer, right-click the scribble solution.
3. On the shortcut menu, click **Build**.
4. On the **Debug** menu, click **Start**.

Run the resulting application and try out the basic features of the application.

Other Versions of Scribble

Scribble is also available as an MFC sample and as a ScribbleDotNET sample:

- SCRIBBLE Sample: MFC MDI Drawing Application
- ScribbleDotNET Sample: Extending Scribble with CLR Features

Classes and Keywords

This sample demonstrates the following classes:

Form; ArrayList; PrintDocument; PrintPageEventHandler; BinaryFormatter; ImageList; HelpProvider; ToolBar; StatusBar; OpenFileDialog; SaveFileDialog; Menu; MenuItem; PrintPreviewDialog; Help

This sample demonstrates the following keywords:

Form::MDIParent; Form::MDIChildren; Graphics::DrawLine

See Also

Other Resources

Visual C++ Samples
General Samples

New Walkthroughs in Visual C++ 2005

This page contains links to walkthroughs that highlight features new to Visual C++ 2005.

In This Section

Walkthrough: Exposing a .NET Plug-in Model

Demonstrates extending the classic Scribble MFC sample to support a plug-in model using .NET technologies.

Walkthrough: Using /clr:pure Features

Demonstrates the use of reflection in a pure MSIL application.

Walkthrough: Using Profile-Guided Optimizations

Demonstrates how to use profile-guided optimization (PGO) for varied scenarios that range from simple implementations using PGO build menus in the Visual Studio Integrated Development Environment (IDE), to more advanced scenarios that require custom configurations.

See Also

Concepts

Visual C++ Walkthroughs

Other Resources

Visual C++ Sample Applications

Walkthrough: Exposing a .NET Plug-in Model

This walkthrough demonstrates extending the classic Scribble MFC sample to support a plug-in model using .NET technologies.

The MFC Scribble solution is extended to expose a .NET object model. Furthermore, the sample also exposes a plug-in architecture that enables .NET plug-ins to interact seamlessly with the MFC Scribble solution.

Prerequisites

ScribbleDotNET Sample: Extending Scribble with CLR Features

To build and run this sample

1. Open the solution file, Scribble.sln, in the Visual Studio development environment.
2. Build the solution using either the "CLR Debug" or "CLR Release" configurations.
3. Under the Solution Explorer open Solution Items plugin.xml
4. In the plugins.xml change the first part of the path in the Assembly element to correspond to the configuration you built under, e.g. change it to "CLR Debug" for "CLR Debug"
5. Run the built executable.
6. Select **Plugins** from the menu bar
    a. Select the **Draw Shapes** menu item. This will run the VBShapes plug-in which will draw three outlines of different colored squares.
7. Select **Plugins** from the menu bar
    a. Select the **Change Pen Color** menu item. This will run CSPenColor plug-in which will allow you to select the pen color through a .Net ColorDialog

How it Works

This sample is implemented in two major parts:

1. Exposing an Object Model for the plug-in writers to use

   Two main interfaces are exposed to implement the plug-in model (see the ScribbleApp Project for definitions of these interfaces):

   - IScribbleApp – this interface exposes a set of features that will allow the plug-in writer to interact with a running Scribble application. The interface is implemented by Scribble (see ScribbleApp.cpp in the Scribble Project) and an instance of the implemented interface is passed to the Run() member function of IScribblePlugin, discussed next
   - IScribblePlugin – this interface is implemented by the plug-in writer. The plug-in writer will use the features exposed by IScribbleApp to implement their plug-in.
2. Extending the base Scribble application to find, load, and execute the plug-ins
   - First an XML file containing a list of plug-ins to be loaded is parsed. Each plug-in found is loaded and verified to be of the correct type by using System.Reflection. More specifically, we use reflection to search through the types exposed by the plug-in module and ensure that one of them implements the IPlugin interface that we have exposed. The implementation for this can be found in PluginBld.cpp in the Scribble Project.
   - CScribbleApp then uses the routines implemented in PluginBld.cpp to obtain a list of all the discovered plugins. For each plugin in this list a menu item is dynamically created under the menu Plugins. When the user selects a menu item from Plugins the appropriate plugin will be run.

Technologies Used

- Using XML to specify the location of the plug-ins
- Using reflection to verify the plug-in specified in the XML file is of the appropriate type

- Defining an object model for creating plug-ins
- Loading and running the plug-in in a C++ application

See Also

Tasks

SCRIBBLE Sample: MFC MDI Drawing Application

Walkthrough: Using /clr:pure Features

This walkthrough demonstrates the use of reflection in pure MSIL application. The Windows Forms Scribble solution is extended to include some native classes that contain static data as well as exposing Windows Forms controls that must be usable from the Windows Forms designer. Building the sample with the /clr:pure option addresses potential issues with this scenario. See How to: Migrate to /clr:pure for more information.

Prerequisites

ScribbleDotNET Sample: Extending Scribble with CLR Features

To build/run this sample application

1. Download the ScribbleDotNET sample.
2. Open the solution file, Scribble.sln, in the Visual Studio development environment.
3. Build the solution using either the "CLR-Pure Debug" or "CLR-Pure Release" configurations.
4. Open the StringDialog.h component in Design mode
5. The design window which appears will contain an instance of the "ScribbleCustomControl".

Technologies

- New C++ Syntax for .NET
- Reflection
- marshal_as<T> for access to native types from managed classes

This sample creates an unmanaged class and wraps it in a managed Control class. When compiled into a pure MSIL application, the Visual Studio Windows Forms designer is able to reflect over the resulting EXE file and load the managed control for use in the Windows Forms designer. This behavior is only possible when the current configuration is to create a pure MSIL application, and not in a unmanaged or mixed mode.

Within our Control, we instantiate an instance of a native class and use marshal_as<T> to allow our managed control access to a native data type returned by a call to a method of our native class. Without using marshal_as<T>, we would be unable to access the non-managed data type from a CLR-enabled application.

See Also

Concepts

New Language Features in Visual C++

Other Resources

New Walkthroughs in Visual C++ 2005

Walkthrough: Using Profile-Guided Optimizations

This walkthrough demonstrates how to use profile-guided optimization (PGO) for varied scenarios that range from simple implementations using PGO build menus in the Visual Studio Integrated Development Environment (IDE), to more advanced scenarios that require custom configurations. This walkthrough also discusses advanced techniques that enable you to use PGO command line tools to customize the process for collecting and merging the profiling data. For overview information, see Profile-Guided Optimizations.

Building an Application with PGO

This example scenario illustrates how to build a simple application using PGO.

Building with PGO in the IDE

Profile guided optimizations can be applied on any native Visual C++ project. The following procedures illustrate using PGO with a Win32 application.

To create a Win32 project

1. On the **File** menu, select **New**, and then select **Project**. The New Project dialog box appears.
2. In the **Project Types** pane, select the **Visual C++ Projects** node, and then in the **Templates** pane, select **Win32 Project**.
3. In the **Name** box, enter **PGODemo**.
4. Click the **OK** button. The Win32 Application Wizard appears.
5. Keep all default settings. Click the **Finish** button.

To build the PGODemo project with PGO

1. In the Standard toolbar, click the **Solution Configuration** combo box and select the **Release** configuration.
2. In **Solution Explorer**, right-click the PGODemo project node. In the project context menu, select **Profile Guided Optimization**, and then select the **Instrument** menu item.
3. In **Solution Explorer**, right-click the PGODemo project node. In the project context menu, select **Profile Guided Optimization**, then select **Run Instrumented Application** to launch your application.

Note
You can also launch the instrumented application by selecting **Start** or **Start Without Debugging** from the **Debug** menu.

4. Repeat step 3, to train the instrumented application using different scenarios. When an instrumented application shuts down, a .PGC file is created in the project output directory, which contains the profile data for that run.
5. In **Solution Explorer**, right-click the PGODemo project node. In the project context menu, select **Profile Guided Optimization**, then select **Optimize**.

Building with PGO at the Command Line

Assume you want to build an application myapp.exe from the source files S1,…,Sn, using PGO optimization. First step is to make sure you compile your sources S1,…,Sn using the /GL compiler option.

You need also to specify the optimization options that matches your needs, let's say you choose the /O2 option. It's to be mentioned that building your application with PGO but without any optimization options, i.e. using /Od, will fire a warning and the compiler will switch automatically to build for LTCG. From the command line your first step would be as follows:

To compile source files with /GL

1. `cl.exe /c /GL /O2 S1.cpp …Sn.cpp`
2. `link /ltcg:pgi S1.obj … Sn.obj /out:myapp.exe`

Step 2 creates a PGO instrumented image myapp.exe and also creates myapp.pgd which is the profile database file that will be used by the compiler to build the final optimized myapp.exe. The default name of the .pgd file is the name of the application and the default path is the output path.

You can use linker option /pgd in the previous step to change the default name of the .pgd file generated for your application. You can use /pgd as follows:

To create an instrumented image with /LTCG:PGI and /PGD

- `link /ltcg:pgi /pgd:mypgd.pgd S1.obj … Sn.obj /out:myapp.exe`

Now you are ready to train the instrumented myapp.exe using your profiling scenarios. After you are done running the scenarios you will find a .pgc file created per run.

Note
If you run more than one image of myapp.exe from the same path simultaneously only one .pgc file will be created for all of t hem and this file will be written to the disk only after all instances are shut down.

Note
The default path for the .pgc files is the executable path. You can change this by setting the environment variable VCPROFILE _PATH before running your profiling scenarios. i.e. set VCPROFILE_PATH=<some path> then run your instrumented image myapp.exe

You are ready now to build the PGO optimized myapp.exe.

To create an optimized image with /LTCG:PGO and /PGD

- `link /ltcg:pgo /pgd:mypgd.pgd S1.obj … Sn.obj /out:myapp.exe`

Note
Before step 1, you need to have all .PGC files created during the training scenario in the same folder as the profile database fi le (.PGD)

Custom Profile Data Merge

Let's assume you have two main customer scenarios for your application. The first scenario is twice as important - or frequently done by customers - as the second one.

In the following steps you will learn how to use the Visual Studio development environment and the command line tools to give some profile data higher weight while merging it into the .PGD file.

Merging Profile Data in the IDE

To create a Win32 project

1. On the **File** menu, select **New**, and then select **Project**. The New Project dialog box appears.
2. In the **Project Types** pane, select the **Visual C++ Projects** node, and then in the **Templates** pane, select **Win32 Project**.
3. In the **Name** box, enter **PGODemo**.
4. Click the **OK** button. The Win32 Application Wizard appears.
5. Keep all default settings. Click the **Finish** button.

To build PGODemo project with merged profile data in the IDE

1. In the standard toolbar, click the Solution Configuration combo box and choose the Release configuration.
2. In the **Solution Explorer**, right click on the PGODemo project. Click Profile Guided Optimization form the shortcut menu, and then click Instrument.

3. In the **Solution Explorer**, right click on the PGODemo project. Click Profile Guided Optimization from the shortcut menu, and then click Run Instrumented Application. This will launch your application. Note that you can also launch the instrumented application using the Start or Start Without Debugging from the Debug menu as normal.
4. Repeat step 3 twice, once for each of your customer scenarios. This step will create two .PGC files in the output directory: PGODemo!1.pgc and PGODemo!2.pgc.
5. In the **Solution Explorer**, right click on the PGODemo project. Click Properties, the PGODemo property pages dialog appears.
6. In **Configuration properties** click **Build Events** then click **Pre-Link Event**. In the **Command Line**, type the following command:

   ```
   "$(VCInstallDir)bin\pgomgr.exe" -merge:2 $(OutDir)\$(ProjectName)!1.pgc $(OutDir)\$(ProjectName).pgd
   ```

Note
This step will use pgomgr.exe to merge PGODemo!1.pgc in a separate step before launching the linker to build the final optimized application. The linker will call pgomgr.exe again during the next step to merge PGODemo!2.pgc file but with the default weight of 1.

7. In the **Solution Explorer**, right click on the PGODemo project. Click **Profile Guided Optimization** from the shortcut menu, and then click **Optimize**.

Merging Profile Data from the Command Line

You can use the command line tools to perform custom profile data merge as follows.

To build merged profile dated from the command line

1. `cl.exe /c /GL /O2 S1.cpp …Sn.cpp`
2. `link /ltcg:pgi S1.obj … Sn.obj /out:myapp.exe`

   The previous two steps will create the instrumented myapp.exe.
3. Train the instrumented myapp.exe using two different scenarios will create two different .PGC files myapp!1.pgc and myapp!2.pgc.

   To give the first scenario higher weight use pgomgr.exe as follows:

   ```
   Pgomgr –merge:2 myapp!1.pgc myapp.pgd
   ```

4. Run the link command to create the final optimized application as follows

   ```
   link /ltcg:pgo /pgd:myapp.pgd /out:myapp.exe
   ```

   This will merge myapp!2.pgc with the default weight of 1.

PGO Update

In the previous two scenarios we used the -ltcg:pgo linker option to build the final optimized application after building and training the instrumented. When using –ltcg:pgo option, the linker will perform some checks to make sure that no changes happened to the input files (object files, libraries, etc…) used to build the instrumented application. In other words, to use –ltcg:pgo, all input files passed to the linker must be the same as the files passed in the –ltcg:pgi step.

This is because accidental changes to the sources after building the instrumented application and the .PGD file can dramatically affect PGO code generation and optimization decisions.

Assume that after you built the instrumented application, created the profile data and built the PGO optimized application, you discovered that you forgot to update the version number of your application, or, you discovered a trivial bug which needs a very tiny fix that does not dramatically affect the flow in the application. For this scenario we have the –ltcg:pgu option. Using this option the linker will not perform the checks it does when using –ltcg:pgo option. You can edit and recompile files or even add new files to the application between PGI and PGU builds.

In the following steps you will learn how to use the Visual Studio development environment and the command line tools to use the –ltcg:pgu option.

Update in the IDE

To build PGODemo with /LTCG:PGU in the IDE

1. In the standard toolbar, click the Solution Configuration combo box and choose the Release configuration.
2. In the **Solution Explorer**, right click on the PGODemo project. Click Profile Guided Optimization form the shortcut menu, and then click Instrument.
3. In the **Solution Explorer**, right click on the PGODemo project. Click Profile Guided Optimization from the shortcut menu, and then click Run Instrumented Application. This will launch your application. Note that you can also launch the instrumented application using the Start or Start Without Debugging from the Debug menu as normal.
4. In this step you can apply any changes to any source file. As mention above, changes that dramatically affect the behavior of your application can cause severe performance regressions. In general you would make very tiny changes, like small bug fixes or resource files changes.
5. In the **Solution Explorer**, right click on the PGODemo project. Click Profile Guided Optimization from the shortcut menu, and then click Update. This will recompile the changed files only and then launch the linker with –ltcg:pgu option instead of –ltcg:pgo, allowing edited files.

Note
You will get warnings for each new function that does not have profile data and each function edited in a way that invalidates the profile data collected during the training step. In the standard toolbar, click the Solution Configuration combo box and choose the Release configuration.

Note
In step 5 if you right click on the changes source file in the **Solution Explorer** and click **Compile**, this will delete all PGO related files in the output folder because the project system performs a clean build when you move from PGO build to a normal build.

Update at the Command Line

You can use the command line tools to perform the PGO update as follows.

To build with /LTCG:PGU at the command line

1. `cl.exe /c /GL /O2 S1.cpp …Sn.cpp`
2. `link /ltcg:pgi S1.obj … Sn.obj /out:myapp.exe`

   The previous two steps will create the instrumented myapp.exe.
3. Train the instrumented myapp.exe
4. `link /ltcg:pgo /pgd:myapp.pgd /out:myapp.exe`

   Step 4 will create the optimized myapp.exe. Assume that you found a small bug in Sm.cpp, you can apply the fixes and compile only Sm.cpp as follows
5. `cl /c /GL /O2 Sm.cpp`

   Then use –ltcg:pgu to build the optimized application using the old profile data created in step 3.
6. `link /ltcg:pgu /pgd:myapp.pgd /out:myapp.exe`

Custom PGO Configurations

In this scenario you will learn how to create custom PGO configurations to build solutions with multiple projects. In this scenario we will add a DLL to the solution and link it to PGODemo application. We will create two configurations: PGIRelease and PGORelease and use them to build the whole solution instead of using the Profile Guided Optimization menu items to build each product at a time.

To create custom PGO configurations

1. On the **File** menu, select **New**, and then select **Project**. The New Project dialog box appears.

2. In the **Project Types** pane, select the **Visual C++ Projects** node, and then in the **Templates** pane, select **Win32 Project**.
3. In the **Name** box, enter **PGODemo**.
4. Click the **OK** button. The Win32 Application Wizard appears.
5. Keep all default settings. Click the **Finish** button.

   Now you have a solution and a project named PGODemo. Next, create a DLL project.
6. In the **Solution Explorer**, right-click on the solution, and select **Add**. And then click **New Project**. The New Project dialog box appears.
7. In the **Project Types** pane, select the **Visual C++ Projects** node, and then in the **Templates** pane, select **Win32 Project**.
8. In the **Name** box, enter **PGODemoDLL**.
9. Click the **OK** button. The Win32 Application Wizard appears.
10. In the **Application Settings** page, choose **DLL Application** type and click **Finish**.

    You will now have a DLL project named PGODemoDLL. Next, add dependency on PGODemoDLL to PGODemo
11. On the **Solution Explorer**, double click on PGODemoDLL project, double click on PGODemoDLL.cpp and add the following code:

    ```
    __declspec(dllexport) void HelloMessage(void)
    {
       MessageBox(NULL, "hello", "PGODemoDLL", MB_OK);
    }
    ```
12. On the **Solution Explorer**, double click on PGODemo project, double click on PGODemo.cpp and add the following declaration before the definition of the function _tWinMain:

    ```
    __declspec(dllimport) void HelloMessage(void);
    ```
13. In –tWinMain before the main message loop add the following code:

    ```
    HelloMessage();
    ```
14. In the **Solution Explorer**, right click on the PDODemo solution and click **Project Dependencies**. The **Project Dependencies** dialog will appear.
15. In the Dependencies page click chose PGODemo in the **Projects** combo box. Check the PGODemoDLL in the **Depends on** list.

    Next, you will create the PGIRelease and PGORelease configurations.
16. In the **Solution Explorer**, right click on the solution, click **Configuration Manager**. Click on the **Configuration** combo box for PGODemo project and click **<New ...>**. The New Project Configuration dialog box will appear.

    In the Project Configuration Name edit box type PGIRelease and choose **Release** in the **Copy Settings From** combo box. Make sure the **Also create new solution configuration** check box is checked.

    Repeat the same step on the PGODemoDLL project and to add the PGIRelease configuration to this project. Make sure that Also create new solution configuration check box is not checked this time.
17. Repeat step 16 for both projects to create PGORelease configuration. Make sure it also copies the settings from the **Release** configuration. Close the **Configuration Manager** dialog box.

    Now we created two configurations, PGIRelease and PGORelease.
18. In the **Solution Explorer**, right click on the PGODemo Project and click **Properties**. In the **Configuration** combo box choose the PGIRelease configuration. In **Configuration** properties click **General**. Click the **Whole Program**

Optimization combo box and choose the **Profile Guided Optimization – Instrument** option. Click **Apply** to save your changes.

19. In the **Configuration** combo box choose the PGORelease configuration. In **Configuration** properties click **General**. Click the **Whole Program Optimization** combo box and choose the **Profile Guided Optimization - Optimize**. Click **Apply** to save your changes.

20. Repeat steps 18 and 19 on the PGIRelease and PGORelease configurations for the PGODemoDLL project.

    Now we will change the **Intermediate Directory** for the PGORelease configuration to be the same as the **Output** directory for the PGIRelease configuration for both projects.

21. In the **Solution Explorer**, right-click on the PGODemo Project and click **Properties**. In the configuration combo box choose the PGORelease configuration. In **Configuration** properties click General. In **the Intermediate Directory**, type PGIRelease. Click **Apply** and close the dialog.

22. Repeat step 21 for the PGODemoDLL project.

    Next, we will add the path of PGODemoDLL.dll to the path environment variable of the PGODemo application.

23. In the **Solution Explorer**, right click on the PGODemo Project and click **Properties**. In the configuration combo box choose the PGIRelease configuration. In **Configuration** properties click **Debugging**. In the **Environment**, type the following and then click **Apply**:

    ```
    path=%path%;$(SolutionDir)PGODemoDLL\PGIRelease
    ```

    In the configuration combo box choose the PGORelease configuration. In **Configuration** properties click **Debugging**. In the **Environment**, type the following:

    ```
    path=%path%;$(SolutionDir)PGODemoDLL\PGORelease
    ```

    Then click **Apply** and then click **OK** to close the dialog.

24. In the **Solution Explorer**, right click on the PGODemo Project and click **Properties**. In the **Configuration** combo box choose the PGORelease configuration. In **Configuration** properties click **Linker** and then click **Optimization**. In **the Profile Guided Database** edit box, replace $(TargetDir) with $(IntDir).

    Repeat this step for PGODemoDLL project.

25. In the **Solution Explorer**, right click on the PGODemoDLL Project and click **Properties**. In the configuration combo box choose the PGORelease configuration. In **Configuration** properties click **Linker** and then click **Advanced**. In the **Import Library** edit box, replace $(OutDir) with $(IntDir).

26. In the standard toolbar click the **Solution Configurations** combo and choose the PGIRelease configuration. In the **Solution Explorer**, right click on the solution and click **Build Solution**.

    This will build the instrumented bits for both projects.

27. Click the **Debug** menu and click **Start** or **Start without Debugging**. This will start PGODemo. After you shutdown the application, two .PGC files will be created, one on each of the two otput folders PGODemo\PGIRelease and PGODemoDLL\PGIRelease.

28. In the standard toolbar click the Solution Configurations combo and choose the PGORelease configuration. In the **Solution Explorer**, right click on the solution and click Build Solution.

    Now you have the PGO optimized images in the PGORelease configuration folders for both projects

Custom Profile Data Collection

The default behavior for profile data collection is to save all the data that describe the behavior of the instrumented application since its startup until it shuts down. PGO comes with a tool called PGOSWEEP that helps you customizing the profile data collection. You can use PGOSWEEP to collect profile data for some specific time frame during the profile scenario run.

In the following steps you will learn how to use the Visual Studio development environment and the command line tools to control the profile data collection.

To create a Win32 project

1. On the **File** menu, select **New**, and then select **Project**. The **New Project** dialog box appears.

2. In the **Project Types** pane, select the **Visual C++ Projects** node, and then in the **Templates** pane, select **Win32 Project**.
3. In the Name box, enter PGODemo.
4. Click the **OK** button. The Win32 Application Wizard appears.
5. Keep all default settings. Click the **Finish** button.

To build instrumented PGODemo project in the IDE

1. In the standard toolbar, click the **Solution Configuration** combo box and choose the **Release** configuration.
2. In the **Solution Explorer**, right click on the PGODemo project. Click **Profile Guided Optimization** from the shortcut menu, and then click **Instrument**.
3. In the **Solution Explorer**, right click on the PGODemo project. Click **Profile Guided Optimization** from the shortcut menu, and then click **Run Instrumented Application**. This will launch your application. Note that you can also launch the instrumented application using the **Start** or **Start Without Debugging** from the **Debug** menu as normal.
4. Click the **Tools** menu, click **External Tools**. The **External Tools** dialog appears. Click **Add**. In the **Title** edit box, type Pgosweep. Click the **Browse** button next to the **Command** edit box and locate the path to the PGOSWEEP tool. PGOSWEEP should be installed in the ...\VC\bin folder where Visual Studio is installed. Check the **Prompt for arguments** check box and then click **OK** to close the **External Tools** dialog.
5. From the **Tools** menu click **Pgosweep**. A dialog box appears. In the **Arguments** edit box, type:

   ```
   $(TargetPath) $(TargetDir)$(TargetName)_custom!1.pgc
   ```

   and click **OK**.
6. In the PGODemo main window click **File** and click **Exit** to end the profile data collection run.
7. Now, if you look at the $(OutDir), you will find two .PGC files. PGODemo_custom!1.pgc is the one that was created by PGOSWEEP and has the profile data from the beginning of the profile run until the time we called PGOSWEEP. The second one will have the default naming scheme PGODemo!1.pgc and contains profile data from the point after the sweeping until the we shut down the instrumented application.
8. It's very important to note that only the .PGC files that obey the standard naming $(ProjectName)!n.PGC where n is a number, will be merged automatically when you build the Optimized image. To merge PGODemo_custom!1.pgc, we created in the previous step, you will have to add a custom merge step (Please see the Custom Profile Data Merge scenario described earlier in this walkthrough).

Using these steps, you can control profile data collection and--in turn--optimize your application based on the most important parts of the application at run time.

Collecting Profile Data from the Command Line

You can use the command line tools to perform custom profile data collection as follows.

To build merged profile dated from the command line

1. Create the instrumented executable:

   ```
   cl.exe /c /GL /O2 S1.cpp Sn.cpp
   link /ltcg:pgi S1.obj Sn.obj /out:myapp.exe
   ```

2. Run the instrumented myapp.exe and, at the desired point while it's running, use the pgosweep tool to collect the profile data:

   ```
   Pgosweep myapp.exe mycustomdata.pgc
   ```

   Note PGC files that do not obey the standard format will have to be merged manually using the Pgomgr tool.

3. After myapp.exe shuts down, another .PGC file (myapp!1.pgc) will be created automatically. Make sure to delete this one or move it outside the linker working directory before you run the linker to build the optimized myapp.exe, if you do not want it to be used as a part of the profile data.

4. Merge mycustomdata.pgc using Pgomgr:

```
Pgomgr –merge mycustomdata.pgc myapp.pgd
```

5. Run the link command to create the final optimized application:

```
link /ltcg:pgo /pgd:myapp
```

Custom profile data collection with PgoAutoSweep

You can call PgoAutoSweep from your application to save and reset the profile data at any point at runtime. The example below shows how this works

The example below will create two .PGC files. The first contains data that describes the runtime behavior until count is equal to 3, and the second contains the data collected after this point until application termination.

```
#include <stdio.h>
#include <windows.h>
#include <pgobootrun.h>

int count = 10;
int g = 0;

void func2(void)
{
    printf("hello from func2 %d\n", count);
    Sleep(2000);
}

void func1(void)
{
    printf("hello from func1 %d\n", count);
    Sleep(2000);
}
void main(void)
{
    while (count--)
    {
        if(g)
            func2();
        else
            func1();
        if (count == 3)
        {
            PgoAutoSweep("func1");
            g = 1;
        }
    }
    PgoAutoSweep("func2");
}
```

See Also

Other Resources

New Walkthroughs in Visual C++ 2005

STL Samples

The Standard Template Library (STL) documentation includes numerous sample programs.

See Also

Other Resources

Visual C++ Samples

Standard C++ Library Reference

ATL Samples

The following table is an alphabetical list of the ATL samples in Visual C++.

In This Section

ACTIVEDOC Sample: Implements an Active Document Server

Demonstrates how to implement an Active Document Server.

ASYNC Sample: Downloads Data Asynchronously

Downloads data asynchronously from a URL.

ASYNC Attributes Sample: Downloads Data Asynchronously

Downloads data asynchronously from a URL with attributes.

ATLAgent Attributes Sample: Uses CCommand and CAccessor to Read Information from a Database

Demonstrates using **CCommand** and **CAccessor** to read information from a database, and demonstrates using the compiler COM support to control the Microsoft Agent control.

ATLCollections Sample: Demonstrates ICollectionOnSTLImpl, CComEnumOnSTL, and Custom Copy Policy Classes

Demonstrates the use of **ICollectionOnSTLImpl** and **CComEnumOnSTL**, and the implementation of custom copy policy classes.

ATLBUTTON Attributes Sample: Demonstrates a Button with Bitmaps for Different States

Creates a button that displays itself with three different bitmaps depending on its state.

ATLCON Sample: Demonstrates Creating a Simple Container

Demonstrates a simple control container.

ATLCON Attributes Sample: Demonstrates Creating a Simple Container

Demonstrates a simple control container with attributes.

ATLDuck Sample: Uses Connection Points with ATL

Demonstrates using connection points with ATL controls.

ATLDuck Attributes Sample: Uses Connection Points with ATL

Demonstrates using connection points with ATL controls and attributes.

ATLEventHandling Sample: Handles Word Events Using IDispEventImpl and IDispEventSimpleImpl

Demonstrates the use of **IDispEventImpl** and **IDispEventSimpleImpl** to handle events fired by Microsoft Word.

ATLFire Sample: Demonstrates Building a Windowed Control

Demonstrates how to build a windowed control using ATL.

ATLFire Attributes Sample: Demonstrates Building a Windowed Control

Demonstrates how to build a windowed control using ATL and attributes.

ATLMovie Attributes Sample: Uses Compiler COM Support and Active Movie Interfaces in an ATL Control

Demonstrates using compiler COM support and the Active Movie interfaces to play a movie in an ATL control.

ATLPages Sample: Implements a Property Page Using IPropertyPageImpl

Demonstrates the implementation of a property page using **IPropertyPageImpl**.

ATLSafeArray Sample: Demonstrates CComSafeArray and Passing SAFEARRAYs to Script

Shows how to create and maintain SAFEARRAYs using **CComSafeArray**; also how to pass SAFEARRAYs from a component to script.

ATLSecurity Sample: Demonstrates ATL Security Classes

Shows how to use the ATL security classes to examine security settings.

ATLTangram Attributes Sample: Demonstrates Managing Large Projects That Use ATL, MFC, and COM

Demonstrates managing a large ATL project with multiple project dependencies in the IDE and with attributes. Also demonstrates some basic COM concepts.

ATLTraceTool Sample: Displays Output of ATLTRACE2

Displays the output generated by the ATLTRACE2 macro.

AutoThread Sample: Demonstrates the Use of CAtlAutoThreadModule

Demonstrates using CComAutoThreadModule Class.

AutoThread Attributes Sample: Demonstrates the Use of CAtlAutoThreadModule

Demonstrates using CComAutoThreadModule Class with attributes.

BEEPER Sample: Demonstrates a Tearoff Interface

Implements a tearoff interface — a collection/enumeration of **BSTR**s.

BEEPER Attributes Sample: Demonstrates a Tearoff Interface

Implements a tearoff interface — a collection/enumeration of **BSTR**s with attributes.

CatDB Sample: Data Source Schema Browser

Displays the schema information, such as tables and columns, of OLE DB providers.

CDINFO Attributes Sample: Plays and Displays Information About CD Audio Tracks

Plays CD audio tracks and displays information about the tracks in tooltips and a piechart display.

CIRC Sample: Demonstrates Using Property Pages

Creates a control that demonstrates property pages and draws a circle.

CIRC Attributes Sample: Demonstrates Using Property Pages

Creates a control with attributes that demonstrates property pages and draws a circle.

CIRCCOLL Attributes Sample: Demonstrates a Collection/Enumeration

Implements a collection/enumeration of objects using ATL with attributes and Standard C++ Library Reference (C++ Standard Template Library).

COMMAP Attributes Sample: Demonstrates COM Interface Map Entry Macros

Shows how different COM interface map entry macros are used.

CThreadPool Sample: Demonstrates How to Use a Thread Pool

Shows how to use a thread pool in an application and how implementing a thread pool can improve the application's performance.

CustomString Sample: Demonstrates Custom Memory Allocators for CStringT

Shows how to use a custom memory allocator for **CStringT** to improve performance in a multithreaded application.

DBViewer Sample: Database Browser

Demonstrates a mid-level application that relies on the **CManualAccessor** class to take full control of data bindings for your applications.

DCOM Sample: Demonstrates Remotely Calling a COM Object

Demonstrates how to call a COM object implemented in a Windows NT service from multiple clients running on different machines.

DIRECT3D Attributes Sample: Demonstrates Using Direct3D

Creates a control that draws a spinning triangle using the Direct3D graphics library and attributes.

DispSink Sample: Handles Events Fired from a Singleton COM Server Through a Dispatch Interface

Demonstrates using a connection point on dispatch interfaces.

DispSink Attributes Sample: Handles Events Fired from a Singleton COM Server Through a Dispatch Interface

Demonstrates using a connection point on dispatch interfaces with attributes.

DynamicConsumer Sample: Uses Dynamic Accessor and Schema Rowset Classes to Read Metadata from a Database

Demonstrates using dynamic accessor and schema rowset classes to read metadata from a database.

LABRADOR Attributes Sample: Implements a Server with No User Interface

Implements an EXE server that does not have any user interface.

Marquee Sample: Provides Performance Monitoring Data Through Scripting

Shows how to use performance monitor objects through scripting.

MFCATL Sample: Uses ATL COM Objects in an MFC Server

Illustrates how ATL COM objects can be used in an MFC server EXE.

MultiRead Sample: Reads Database Table Using Multiple Threads

Reads through a table in a database using multiple threads.

MultiRead Attributes Sample: Reads Database Table Using Multiple Threads

Reads through a table in a database using multiple threads.

OPENGL Attributes Sample: Demonstrates Using OpenGL

Creates a control that draws a spinning cube using the OpenGL graphics library and attributes.

POLYGON Sample: The ATL Tutorial

The project files for the ATL Tutorial. Builds a control that implements custom properties, events, property pages, and object safety.

POLYGON Attributes Sample: Demonstrates Custom Properties, Events, a Property Page, and Object Safety

The project files for the ATL Tutorial. Builds a control that implements custom properties, events, property pages, and object safety with ATL and attributes.

SUBEDIT Sample: Superclasses a Standard Windows Control

Creates a superclassed Windows control.

SUBEDIT Attributes Sample: Superclasses a Standard Windows Control

Creates a superclassed Windows control with attributes.

UpdatePV Sample: Implements an Updatable OLE DB Provider

Implements an updateable (read/write) OLE DB provider.

See Also

Other Resources

Visual C++ Samples
ATL Server Samples
ATL Attributes Samples
OLE DB Templates
ATL Concepts

ATL Advanced Samples

The following topics are the abstracts for the ATL advanced samples.

See Also

Other Resources

ATL Samples

ACTIVEDOC Sample: Implements an Active Document Server

Download sample

The ACTIVEDOC sample demonstrates how to implement an Active Document Server. The sample demonstrates the following:

- How to implement the interfaces **IOleDocument** and **IOleDocumentView**.
- How to perform menu merging with the container.
- How to implement a toolbar in your object.
- How to override the **IPersistStreamInit** methods **Load** and **Save** to perform custom saving and loading.
- How to use **CDialogImpl** to implement an About dialog box.
- How to subclass the Rich Text Edit Control.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build the sample

1. Open the solution ActiveDoc.sln.
2. From the **Build** menu, click **Build Solution**.

To run the sample using Internet Explorer

1. Open ActiveDoc.htm or the ActiveDoc.AAA generated file in Internet Explorer (to open ActiveDoc.AAA, click the **Open** button when prompted to open this file).
2. Use the Format menu that has been merged into Internet Explorer's menu bar, or just click the toolbar colors to change the color of the typed text.

To run the sample using Microsoft Office Binder

1. Run Microsoft Office Binder and from the **Section** menu click **Add**.
2. Select the icon labeled "ActiveDoc Class" and click **OK**.

   You should see the object embedded in Office Binder. Notice that the menus have been merged and the ActiveDoc's toolbar is shown. You can enter text and use the menu and toolbar buttons to change the color of the text that is typed.

Sample Files

The sample includes the following main files:

- ActiveCtl.h
- ActiveCtl.cpp — implements **CActiveDoc**, which represents the overall **ActiveDoc** object.
- OleDocument.h — contains the implementation of the **IOleDocument** and **IOleDocumentView** interfaces.
- Toolbar.h — contains **CToolbar**, which helps with the implementation of the toolbar.

Keywords

This sample uses the following keywords:

ActiveXDocActivate; ALT_MSG_MAP; ATLASSERT; ATLTRACE; BEGIN_COM_MAP; BEGIN_MSG_MAP; BEGIN_OBJECT_MAP;

BEGIN_PROPERTY_MAP; BEGIN_TOOLBAR_MAP; CAboutDlg::DoModal; CanInPlaceActivate; CComCoClass; CComControl; CComModule::GetClassObject; CComModule::GetLockCount; CComModule::Init; CComModule::RegisterServer; CComModule::Term; CComModule::UnregisterServer; CComObjectRoot; CComObjectRootEx::InternalQueryInterface; CComPtr; CHARFORMAT; CMenu; COM_INTERFACE_ENTRY; COM_INTERFACE_ENTRY_IMPL; COMMAND_ID_HANDLER; COMMAND_RANGE_HANDLER; CreateMenu; CreateRTFWindow; CreateToolbar; CreateWindowEx; CToolbar; CWindow::DestroyWindow; CWindow::ModifyStyle; CWindow::SetFocus; CWindow::SetParent; CWindow::SetWindowPos; DECLARE_REGISTRY_RESOURCEID; DestroyMenu; DestroyToolbar; DisableThreadLibraryCalls; DllMain; DoesVerbUIActivate; EDITSTREAM::dwCookie; EDITSTREAM::dwError; EDITSTREAM::pfnCallback; Ellipse; END_COM_MAP; END_MSG_MAP; END_OBJECT_MAP; END_PROPERTY_MAP; END_TOOLBAR_MAP; EndDialog; ExtTextOut; FreeLibrary; GetMenuItemCount; GetMenuItemID; GetMenuState; GetMenuString; GetResourceInstance; GetStockObject; GetSubMenu; GetTextMetrics; GetWindow; GetWindowContext; GetWindowRect; HDC; hdcDraw;; IDataObjectImpl; IDispatchImpl; InitCommonControls; InlineIsEqualGUID; InPlaceDeactivate; InPlaceMenuCreate; InPlaceMenuDestroy; InsertMenu; InterfaceSupportsErrorInfo; InternalQueryInterface; IOleControlImpl; IOleDocument::GetDocMiscStatus; IOleDocumentImpl; IOleDocumentView::ApplyViewState; IOleDocumentView::SetInPlaceSite; IOleDocumentViewImpl; IOleInPlaceActiveObjectImpl; IOleInPlaceFrame* pFrame;; IOleInPlaceFrame::InsertMenus; IOleInPlaceFrame::Release; IOleInPlaceFrame::SetMenu; IOleInPlaceObjectWindowlessImpl; IOLEInPlaceSite::GetWindowContext; IOleObjectImpl; IPersistStorageImpl; IPersistStreamInitImpl; IProvideClassInfo2Impl; IQuickActivateImpl; ISupportErrorImpl; IViewObjectExImpl; LoadLibrary; LoadMenu; LoadString; MAKEINTRESOURCE; MergeMenus; MESSAGE_HANDLER; NOTIFY_CODE_HANDLER; OBJECT_ENTRY; OleCreateMenuDescriptor; OLEINPLACEFRAMEINFO; OnInPlaceActivate; OnUIActivate; prcBounds; RECT rcPos, rcClip;; RemoveMenu; SelectObject; SendMessage; SetActiveObject; SetBkMode; SetBorderSpace; SetControlFocus; SetFocus; SetInPlaceSite; SetMenu; SetObjectRects; SetTextColor; ShowObject; ShowWindow; UIActivate; UIDeactivate; UnmergeMenus; ZeroMemory

See Also

Other Resources

ATL Samples

ASYNC Sample: Downloads Data Asynchronously

Download sample

The ASYNC sample creates a control that downloads data asynchronously from a URL. The control implements the **IBindStatusCallback** interface. Typically, you asynchronously download large binary objects or properties. This allows the control's user interface to remain unblocked during potentially lengthy network operations. The use of asynchronous downloading also gives the user a chance to abort the download. ATL uses WinInet functions internally to implement asynchronous downloading.

The ASYNC attributes sample is the attributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file async.sln.
2. From the **Build** menu, click **Build Solution**.
3. After the sample builds, open ATLAsync.htm in your Web browser and follow the instructions. This sets the ASYNC control's **URL** property and starts the download. As data is downloaded, you will see it displayed in the ASYNC control.

You can test the control in the ActiveX Control Test Container. For details on accessing Test Container and using it to test a control, see Testing Properties and Events with Test Container.

How the Sample Works

ASYNC creates a subclassed edit control with one property called **URL**. The **URL** property is a **BSTR** that represents a URL that points to data. The ASYNC sample uses the ATL **CBindStatusCallback** class to implement asynchronous downloading. When the control user sets the URL property, ASYNC creates a **CBindStatusCallback** object. The **CBindStatusCallback::StartAsyncDownload** method is then called and passed both the URL and a pointer to a callback function. This function, **CAtlAsync::OnData**, is called by the **CBindStatusCallback** object and is passed the binary data from the URL as it is received. **CAtlAsync::OnData** simply sends the received data to the subclassed edit control, where it is displayed.

For an example of how to superclass Windows controls using ATL, see the ATL SubEdit sample.

Keywords

This sample uses the following keywords:

ALT_MSG_MAP; ATLTRACE ; BEGIN_COM_MAP; BEGIN_MSG_MAP; BEGIN_OBJECT_MAP; BEGIN_PROPERTY_MAP; CBindStatusCallback::Download; CComBSTR::Append; CComCoClass; CComControl; CComModule::GetClassObject; CComModule::GetLockCount; CComModule::Init; CComModule::RegisterServer; CComModule::Term; CComModule::UnregisterServer; CComObjectRoot; COM_INTERFACE_ENTRY; COM_INTERFACE_ENTRY_IMPL; DECLARE_REGISTRY_RESOURCEID; DisableThreadLibraryCalls; DLL_PROCESS_ATTACH; DLL_PROCESS_DETACH; DllMain; END_COM_MAP; END_MSG_MAP; END_OBJECT_MAP; END_PROPERTY_MAP; IDataObjectImpl; IDispatchImpl; IObjectSafetyImpl; IOleControlImpl; IOleInPlaceActiveObjectImpl; IOleInPlaceObjectWindowlessImpl; IOleInPlaceObjectWindowlessImpl::SetObjectRects; IOleObjectImpl; IPerPropertyBrowsingImpl; IPersistPropertyBagImpl; IPersistStorageImpl; IPersistStreamInitImpl; IProvideClassInfo2Impl; IQuickActivateImpl; IsWindow; IViewObjectExImpl; MESSAGE_HANDLER; OBJECT_ENTRY; PROP_ENTRY; SendMessage; USES_CONVERSION

See Also

Other Resources

ATL Samples

ATLDuck Sample: Uses Connection Points with ATL

Download sample

The ATLDuck sample consists of four projects: atlduck, duck and their respective proxy/stub projects atlduckPS and duckPS. The duck project creates an instance of an object that implements the IDuckInt interface. This interface includes four member functions: Flap, Paddle, Quack, and Walk.

The second project, atlduck, has a connection point for the IDuckInt interface (that is, it knows how to use the interface but does not implement it). Only a single instance of this object will be created. Once the connection is established between the two applications, atlduck will call the functions in the IDuckInt interface for sinks that have called **IConnectionPoint::Advise**.

This sample also makes use of the marshaling code (through atlduckPS and duckPS) needed to connect interfaces in different processes.

The ATLDuck attributes sample is the attributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build this sample

1. Open the atlduck.sln solution file.
2. From the **Build** menu, click **Build Solution**. This will build all four projects and do the necessary registration. The following files will be created:
   - atlduck.exe, an EXE server
   - atlduckPS.dll, a marshalling DLL for the atlduck.exe server
   - duck.exe, an EXE client
   - duckPS.dll, a marshalling DLL for the duck.exe client

To run this sample

1. Start one or more instances of duck.exe.

Note
Opening several instances of duck.exe demonstrates connection points most effectively.

2. A dialog box appears. In the dialog box, click the button **Create DoDuck Object**. When you click this button, the application creates an instance of an object with class `ID CLSID_DuckDoer`, running atlduck.
3. A new dialog box, issued by atlduck.exe, appears. This dialog box shows a button for each function in the IDuckInt interface, as well as a list box with the connections to active sinks, and their cookies. From the duck dialog boxes, you can either **Advise** or **Unadvise** the connection point. Depending on your choice, you will receive or not receive notifications from the sources. The notification, when you receive it, will show in the status edit field. Additionally, ATLDuck demonstrates the value of the cookie supplied by the connection point when the connection is advised.

Classes and Keywords

The sample use the following classes:

CDialog (MFC), CComObject (ATL), IConnectionPointImpl<CDuckDoer> (ATL)

This sample uses the following keywords:

_ASSERTE; _tcslen; _VERIFY; AddRef; Advise; BEGIN_COM_MAP; BEGIN_CONNECTION_POINT_MAP; BEGIN_MSG_MAP; BEGIN_OBJECT_MAP; CComCoClass; CComModule::Unlock; CComObjectRootEx; CDialogImpl; CenterWindow; CoInitialize; COM_INTERFACE_ENTRY; COM_INTERFACE_ENTRY_IMPL; COMMAND_HANDLER; COMMAND_ID_HANDLER; CONNECTION_POINT_ENTRY; CoSuspendClassObjects; CoUninitialize; Create; DECLARE_CLASSFACTORY_SINGLETON; DECLARE_NOT_AGGREGATABLE; DECLARE_REGISTRY_RESOURCEID; DestroyWindow; DispatchMessage; EnableWindow; END_COM_MAP; END_CONNECTION_POINT_MAP; END_MSG_MAP; END_OBJECT_MAP; ExitProcess; FindConnectionPoint; GetCommandLine; GetDC; GetDlgItem; GetMessage; GetTextExtentPoint32; IConnectionPointContainerImpl; IConnectionPointImpl; Init; IsWindowVisible; MESSAGE_HANDLER; MessageBox; OBJECT_ENTRY; OleInitializeCoCreateInstance; OnCancel; OnFlap; OnInitDialog; OnOK; OnPaddle; OnWalk OnQuack; PostThreadMessage; QueryInterface; RecalcListboxExtent; RegisterClassObjects; RegisterServer; reinterpret_cast; ReleaseDC; RevokeClassObjects; SendMessage; SetOwner; SetWindowText; ShowStatus; ShowWindow; Unadvise; UnregisterServer; UNUSED_ALWAYS; UpdateRegistryFromResource; UpdateWindow; USES_CONVERSION

See Also

Other Resources

ATL Samples

ATLSecurity Sample: Demonstrates ATL Security Classes

Download sample

The ATLSecurity sample shows how to use the ATL security classes to examine security settings.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build this sample

1. Open the solution file ATLSecurity.sln.
2. From the **Build** menu, click **Build Solution**.

You can run the ATLSecurity sample from a command prompt or the debugger. Passing -? as a parameter displays a help message. To get examples of the format of the names of various objects ATLSecurity can examine, see the documentation for the SE_OBJECT_TYPE enumeration.

The Event subproject creates an event with the name `MyEvent` and a DACL that gives everyone read access to the event and gives the Administrator and Localsystem full access to the event. The Event subproject demonstrates how to use the ATL security classes to build a DACL and gives a target for the ATLSecurity sample.

Event.exe will create an event and wait for a keypress. To use it with ATLSecurity.exe, run Event.exe from one command prompt, then run the following from another command prompt:

```
ATLSecurity -kernel MyEvent
```

Keywords

This sample shows how to use the following classes:

CSid, CAcl, CDacl, CSacl, CSecurityDesc, CSecurityAttributes, CTokenGroups, CTokenPrivileges, CAccessToken

See Also

Other Resources

ATL Samples

ATLTraceTool Sample: Displays Output of ATLTRACE2

Download sample

The ATLTraceTool sample builds an application that specifies which ATLTRACE2 messages appear in the output window. This application, the ATL Trace Tool, displays debug trace messages in the ATL and MFC sources. You can control the type and amount of messages displayed.

AtlTraceTool.exe ships in Visual Studio .NET and can be found in the Microsoft Visual Studio .NET 2003\Common7\Tools directory. AtlTraceTool.exe is also available from the **Tools** menu in the development environment.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file tracetool.sln.
2. From the **Build** menu, click **Build**.
3. From the **Debug** menu, click **Start Without Debugging**.

Note
This procedure is optional. You can use the prebuilt AtlTraceTool.exe instead if you want.

To use ATLTraceTool.exe

1. Debug an MFC or ATL project by clicking **Start** from the **Debug** menu.
2. From the **Tools** menu, click **MFC/ATL Trace Tool** if it is not already running.
3. Expand the tree control list in the **Trace List** window. The window shows the running application, any modules within that application, and the trace categories for each module.
4. Customize, for each process, module, and category, which information is displayed in the output window. The **Trace level** control in the **Process** group is related to the **ATLTRACE2** level; only those **ATLTRACE2** messages with a level equal to or greater than the setting in the **Trace level** control will be displayed in the output window.
5. Click **Apply** to put your settings into effect.

You can save your settings and load them the next time you debug the application; use the **Save** and **Load** buttons.

ATL Trace Tool User Interface

Trace List

A tree control with a list of processes that use debug ATL/MFC sources. Under each process, modules are listed and under each module, trace categories are listed.

Refresh

Updates the list of processes and modules under **Trace List**.

Process Group

Process-wide settings:

Trace level – Equates to the trace level parameter in calls to ATLTRACE2. Currently the ATL/MFC sources use a trace level of 0–4, where 0 is the most critical level.

Enabled – Enables tracing for the process.

Category & Function Names – Includes the trace category and function name in trace messages.

File Name & Line No. – Includes the file name and line number in trace messages.

Module Group

You must first select a module in the **Trace List** tree control before this group is enabled. These settings affect the module.

Trace level – Equates to the trace level parameter in calls to **ATLTRACE2**. Currently the ATL/MFC sources use a trace level of 0–4, where 0 is the most critical.

Inherit from Process – Allows trace settings in the **Process** group to also be used for this module.

Enabled – Enables tracing for this module.

Disabled – Disables tracing for this module.

Category Group

You must first select a category in the **Trace List** tree control before this group is enabled. These settings affect the category within the module.

Trace level – Equates to the trace level parameter in calls to ATLTRACE2. Currently the ATL/MFC sources use a trace level of 0–4, where 0 is the most critical.

Inherit from Module – Allows trace settings in the **Module** group to also be used for this category.

Enabled – Enables tracing for this category.

Disabled – Disables tracing for this category.

Save

Saves settings for the current process and all modules under it to a .trc file. The .trc file can be loaded by using the **Load** button or by calling AtlTraceLoadSettings from the project. If the .trc file is located in the same directory as the EXE/DLL, you can call **AtlTraceLoadSettings** and pass a NULL for the file name.

Load

Loads a .trc file.

Apply

Applies the current settings to the loaded process.

Close

Closes the **ATL/MFC Trace Tool** dialog box. Changes will not take effect unless **Apply** was selected.

Help

Displays help.

See Also

Other Resources

ATL Samples

CThreadPool Sample: Demonstrates How to Use a Thread Pool

Download sample

The CThreadPool sample shows how to use a thread pool in an application and how implementing a thread pool can improve the application's performance.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build the sample

1. Open the solution ThrdPool.sln.
2. Select the desired configuration (Debug or Release).
3. From the **Build** menu, click **Rebuild All**.

To run the sample within Visual Studio

- From the **Debug** menu, click **Start Without Debugging**.

To run the sample from a command window

1. Change to the directory where the chosen configuration was built (for example, ..\CThreadPool\Debug).
2. Execute ThrdPool.exe.

Keywords

This sample demonstrates the following keywords:

CThreadPool; CSimpleArray; InterlockedIncrement; GetCurrentThreadId

See Also

Other Resources

ATL Samples

DCOM Sample: Demonstrates Remotely Calling a COM Object

Download sample

The DCOM sample demonstrates how to call a COM object implemented in a Windows service from multiple clients running on different machines. It is composed of three parts:

- DrawServ — The Windows service that implements the COM object.
- ATLDraw — The client that connects to the DrawServ COM object.
- DrawCtl — A control version of ATLDraw.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

For this sample to work, all computers must be running Windows NT 4.0 or later.

To build and run the sample

1. Open the solution DCOM.sln.
2. From the **Build** menu, click **Build Solution**.
3. Copy DrawServ.exe and ATLDraw.exe or DrawCtl.dll to each computer on which you want to run the sample. Register the server on each computer by running DrawServ with the command-line argument `/RegServer` or `-RegServer` (this is case-insensitive). For example:

   ```
   C:\ATL> DrawServ /RegServer
   ```

   (ATL.DLL must be registered for this to work.) You must copy the server to each client to register the CLSID for the server and the server type library on each client.
4. Start the service on the server by using the Services icon in Control Panel.
5. Using the DCOMCNFG utility on each client (run it from a command line), select the Properties of the **DrawServ** Class. From the Location tab, select **Run application on the following computer**. Enter the name of the computer on which you are running the server object.
6. Run ATLDraw and select Server Connect from the menu on each client. Draw on the client window by holding the left mouse button down and dragging a line. The drawn line should appear on each client that is connected to the same server. You can also use the **View/Color** menu option to change the color for each client.

The DrawServ sample will not register correctly when its path contains spaces. One solution to this problem is to edit the corresponding .rgs file and add single quotes around all occurrences of `%MODULE%`. For example, change a line containing the following:

```
InprocServer32 = s %MODULE%
```

to the following:

```
InprocServer32 = s '%MODULE%'
```

Keywords

This sample uses the following keywords:

AfxMessageBox; ASSERT_VALID; ATLASSERT; AtlUnadvise; BEGIN_COM_MAP; BEGIN_CONNECTION_POINT_MAP; BEGIN_INTERFACE_MAP; BEGIN_MSG_MAP; BEGIN_OBJECT_MAP; CComCoClass; CComControl; CComModule::GetClassObject; CComModule::GetLockCount; CComModule::Init; CComModule::RegisterServer; CComModule::Term; CComModule::Unlock; CComModule::UnregisterServer; CComModule::UpdateRegistryFromResource; CComObjectRoot; CDocument::OnNewDocument; CFrameWnd::AssertValid; CFrameWnd::DockControlBar; CFrameWnd::Dump; CFrameWnd::EnableDocking; CFrameWnd::OnCreate; CFrameWnd::PreCreateWindow; CloseServiceHandle; CoCreateInstance; CoCreateInstanceEx; CoInitialize; CoInitializeSecurity; COM_INTERFACE_ENTRY; COM_INTERFACE_ENTRY_IMPL; COM_INTERFACE_ENTRY_IMPL_IID; COM_INTERFACE_ENTRY2; CONNECTION_POINT_ENTRY; ConnectSink; ControlService; CoUninitialize; CProxyIDrawServ; CreatePen; CreateService; CStatusBar::SetIndicators; CToolBar::EnableDocking; CToolBar::LoadToolBar; CView::DoPreparePrinting; CView::OnLButtonUp(nFlags, point);; CView::PreCreateWindow; CWinApp::AddDocTemplate; CWinApp::LoadStdProfileSettings; CWinApp::ParseCommandLine; CWinApp::ProcessShellCommand; CWindow::SetCapture; DECLARE_GET_CONTROLLING_UNKNOWN; DECLARE_REGISTRY_RESOURCEID; DeleteObject; DeregisterEventSource; DisableThreadLibraryCalls; Disconnect; DisconnectSink; END_COM_MAP; END_CONNECTION_POINT_MAP; END_INTERFACE_MAP; END_OBJECT_MAP; GetModuleFileName; IConnectionPointContainerImpl; IDataObjectImpl; IDispatchImpl; IMPLEMENT_DYNCREATE; Install; INTERFACE_PART; IObjectSafetyImpl; IOleControlImpl; IOleInPlaceActiveObjectImpl; IOleInPlaceObjectWindowlessImpl; IOleObjectImpl; IPersistStorageImpl; IPersistStreamInitImpl; IProvideClassInfo2Impl; IQuickActivateImpl; ISupportErrorInfo; IViewObjectExImpl; IViewObjectExImpl::Draw; LineTo; LoadString; LogEvent; MESSAGE_HANDLER; MessageBox; MoveToEx; OBJECT_ENTRY; OffsetRect; OpenSCManager; OpenService; PostThreadMessage; PtInRect; QueryInterface; RegisterEventSource; ReleaseDC; ReportEvent; SelectObject; SetCapture; SetServiceStatus; SetWindowOrgEx; StartServiceCtrlDispatcher; Uninstall

See Also

Other Resources

ATL Samples

Marquee Sample: Provides Performance Monitoring Data Through Scripting

Download sample

The Marquee sample shows how to use performance monitor objects through scripting.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file Marquee.sln.
2. From the **Build** menu, click **Build Solution**.

   This will build and register the PerfDisp project and will build the Canvas project.
3. From the **Debug** menu, click **Start Without Debugging**.

   This will run Canvas.exe, which will display a window with a scrolling marquee.

How the Sample Works

The project PerfMonDisp is a DLL that wraps the functionality of **CPerfMon** and exposes it through an automation interface. These interfaces are defined using attributes. See PerfDisp\PerfMonDisp.h for this code.

PerfMonDisp.dll also handles the DLL entry points that the performance monitoring system requires to collect performance monitoring data.

Canvas is a dialog-based MFC project that hosts JScript and PerfMonDisp. It does the following:

- Loads JScript and parses script.js.
- Exposes a drawing area to the script by giving the script an **IDispatch** interface implemented by the dialog box.
- Gives the script an instance of PerfMonDisp.
- Calls methods implemented in the loaded script to update the drawing area.

The management of the scripting engine is done by the code in Canvas\Script.h.

Canvas\CanvasDlg.cpp calls the script management code in Canvas\Script.h.

Classes

This sample demonstrates the following classes:

CPerfMon, IActiveScript, IActiveScriptSite

See Also

Reference

Performance Monitoring Samples

Other Resources

ATL Samples

MFCATL Sample: Uses ATL COM Objects in an MFC Server

Download sample

The MFCATL sample illustrates how ATL COM objects can be used in an MFC server EXE.

The server allows the creation of two objects: **ObjectOne** (implemented in MFC and supporting a **dispinterface**) and **ObjectTwo** (implemented in ATL and supporting a dual interface).

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the sample

1. Open the solution file mfcatl.sln.
2. From the **Build** menu, click **Build Solution**.
3. From the **Debug** menu, click **Start Without Debugging**. This will run the mfcatl.exe server standalone and will register it.
4. Open mfcatl.htm file in your Web browser and click the buttons to call into appropriate objects. You can call each object individually or both at the same time.

Conversion Remarks

Originally, both MFCATL objects were implemented in MFC. Both were derived from **CCmdTarget**. **ObjectTwo** was reimplemented using ATL by following these steps:

1. Include the ATL header files (Atlbase.h and Atlcom.h) in Premfcat.h.
2. Include Atlimpl.cpp in Premfcat.cpp.
3. Add a CComModule-derived class to Prefcat.h (similar to the BEEPER EXE sample). The derived class implements the **Lock** and **Unlock** methods to forward lock counts to MFC by calling **AfxOleLockApp** and **AfxOleUnlockApp**.
4. Add an object map macro pair (**BEGIN_OBJECT_MAP**/**END_OBJECT_MAP**) in Mfcatl.cpp and add a static instance of the module class called **_Module**.
5. Call **_Module.Init** and **Term** from **InitInstance** and **ExitInstance**.
6. Add the typelib as a resource.
7. Call **_Module.RegisterServer(TRUE)** from **InitInstance** to match the MFC call to **COleObjectFactory::UpdateRegistryAll**.
8. Call **_Module.RegisterClassObjects(CLSCTX_LOCAL_SERVER**, **REGCLS_MULTIPLEUSE)** to match the **REGCLS_MULTIPLEUSE** of MFC's class factories, as implicitly done by the **IMPLEMENT_OLECREATE** macro.
9. Reimplement **ObjectTwo** in ATL by following these steps:
   - Change the ODL file from **dispinterface** to dual interface.
   - Have the ODL compilation generate a header file (Interf.h) needed by ATL.
   - Rewrite .h and .cpp files (the quickest way is to run the ATL wizards and copy, paste, and rename the automatically generated code in place of the original MFC code).
   - Add an **OBJECT_ENTRY(CLSID_ObjectTwo, CObjectTwo)** to the ATL object map.

Additional conversion steps (not covered here) might include:

- Porting the ODL file to IDL format.
- Adding support for **-RegServer** and **-UnregServer** command-line arguments.

Keywords

This sample uses the following keywords:

AfxMessageBox; AfxOleInit; AfxOleLockApp; AfxOleUnlockApp; ASSERT; BEGIN_COM_MAP; CCmdTarget; CCmdTarget::OnFinalRelease; CComCoClass; CComModule; CComModule::GetLockCount; CComModule::Lock; CComModule::Unlock; CComObjectRoot; CDialog; CMenu::AppendMenu; COleObjectFactory::RegisterAll; COleObjectFactory::UpdateRegistryAll; COM_INTERFACE_ENTRY; CString::IsEmpty; CString::LoadString; CWindow::GetSystemMenu; DECLARE_DYNCREATE; DECLARE_NOT_AGGREGATABLE; DECLARE_REGISTRY; DestroyWindow; EnableAutomation; GetClientRect; GetSystemMetrics; IDispatchImpl; PostMessage; SendMessage; SetIcon; ShowWindow; SysAllocString

See Also

Other Resources

ATL Samples

ATL Attributes Samples

The following topics are the abstracts for the ATL attributes samples.

See Also

Other Resources

ATL Samples

ASYNC Attributes Sample: Downloads Data Asynchronously

Download sample

The ASYNC attributes sample creates a control that downloads data asynchronously from a URL. The control implements the **IBindStatusCallback** interface. Typically, you asynchronously download large binary objects or properties. This allows the control's user interface to remain unblocked during potentially lengthy network operations. The use of asynchronous downloading also gives the user a chance to abort the download. ATL uses WinInet functions internally to implement asynchronous downloading.

The ASYNC sample is the nonattributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file async.sln.
2. From the **Build** menu, click **Build Solution**.
3. After the sample builds, open ATLAsync.htm in your Web browser and follow the instructions. This sets the ASYNC control's **URL** property and starts the download. As data is downloaded, you will see it displayed in the ASYNC control.

You can test the control in the ActiveX Control Test Container. For details on accessing Test Container and using it to test a control, see Testing Properties and Events with Test Container.

How the Sample Works

ASYNC creates a subclassed edit control with one property called **URL**. The **URL** property is a **BSTR** that represents a URL that points to data. The ASYNC sample uses the ATL **CBindStatusCallback** class to implement asynchronous downloading. When the control user sets the URL property, ASYNC creates a **CBindStatusCallback** object. The **CBindStatusCallback::StartAsyncDownload** method is then called and passed both the URL and a pointer to a callback function. This function, **CAtlAsync::OnData**, is called by the **CBindStatusCallback** object and is passed the binary data from the URL as it is received. **CAtlAsync::OnData** simply sends the received data to the subclassed edit control, where it is displayed.

For an example of how to superclass Windows controls using ATL, see the ATL SubEdit sample.

Attributes

This sample uses the following attributes:

coclass, default, dual, helpstring, id, implements_category, in, module, object, out, pointer_default, progid, propget, propput, registration_script, retval, threading, uuid, version, vi_progid

Keywords

This sample uses the following keywords:

ALT_MSG_MAP; ATLTRACE ; BEGIN_COM_MAP; BEGIN_MSG_MAP; BEGIN_OBJECT_MAP; BEGIN_PROPERTY_MAP; CBindStatusCallback::Download; CComBSTR::Append; CComCoClass; CComControl; CComModule::GetClassObject; CComModule::GetLockCount; CComModule::Init; CComModule::RegisterServer; CComModule::Term; CComModule::UnregisterServer; CComObjectRoot; COM_INTERFACE_ENTRY; COM_INTERFACE_ENTRY_IMPL; DECLARE_REGISTRY_RESOURCEID; DisableThreadLibraryCalls; DLL_PROCESS_ATTACH; DLL_PROCESS_DETACH; DllMain; END_COM_MAP; END_MSG_MAP; END_OBJECT_MAP; END_PROPERTY_MAP; IDataObjectImpl; IDispatchImpl; IObjectSafetyImpl; IOleControlImpl; IOleInPlaceActiveObjectImpl; IOleInPlaceObjectWindowlessImpl; IOleInPlaceObjectWindowlessImpl::SetObjectRects; IOleObjectImpl; IPerPropertyBrowsingImpl; IPersistPropertyBagImpl; IPersistStorageImpl; IPersistStreamInitImpl; IProvideClassInfo2Impl; IQuickActivateImpl; IsWindow; IViewObjectExImpl;

MESSAGE_HANDLER; OBJECT_ENTRY; PROP_ENTRY; SendMessage; USES_CONVERSION

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

ATLAgent Attributes Sample: Uses CCommand and CAccessor to Read Information from a Database

Download sample

The ATLAgent attributes sample reads instructions from a database using the OLE DB Consumer Template classes. These instructions are used to command the Microsoft Agent control. The sample demonstrates how to use the **CCommand** and **CAccessor** classes to read information from a database and how to use the compiler COM support to control the Microsoft Agent.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To install the Microsoft Agent control

1. Download the Microsoft Agent control from http://go.microsoft.com/fwlink/?linkid=558.
2. Optionally, download the text-to-speech engine available from the same page to allow the agent to speak.
3. Choose the link to download the Microsoft Agent character animation files and download one or more of the character .acs files. By default the ATLAgent control uses the Merlin-with-sound-effects character (merlinsfx.acs). Save the files to the directory where you have installed Microsoft Agent, typically C:\Program Files\Microsoft Agent\.

To use the Microsoft Agent control

1. Set up a Microsoft Access data source called Agent and point the data source to the Agent.mdb file contained in the sample's directory as follows:
   - In Control Panel, select **Administrative Tools**, then select **Data Sources (ODBC)**; the ODBC Data Source Administrator dialog box appears.
   - In the ODBC Data Source Administrator dialog box, go to the **System DSN** tab, then click **Add**; the Create New Data Source dialog box appears.
   - In the Create New Data Source dialog box, from the list of data sources, choose "Microsoft Access Driver (*.mdb)" then click **Finish**; the ODBC Microsoft Access Setup dialog box appears.
   - In the ODBC Microsoft Access Setup dialog box, in **Data Source Name**, type "Agent" then click **Advanced**; the Set Advanced Options dialog box appears.
   - In **Set Advanced Options** dialog box, under **Options**, select DefaultDir, and enter the path to the Agent.mdb file as the DefaultDir property.
2. Modify the directory locations at the beginning of AgentCtl.h if you have installed Microsoft Agent in a different directory.
3. Copy the ATLAgent project files and build the ATLAgent project.
4. Open ATLAgent.htm from the sample and click the **Play** button. The Agent should appear and should follow the instructions contained in the Instructions table in the Agent.mdb database.

Attributes

This sample uses the following attributes:

coclass, db_column, db_source, db_table, default, dll, dual, helpstring, id, in, module, name, object, out, pointer_default, progid, propget, propput, retval, uuid, vi_progid

Note

Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

ATLBUTTON Attributes Sample: Demonstrates a Button with Bitmaps for Different States

Download sample

The ATLBUTTON attributes sample creates a button that displays itself with one of three different bitmaps, depending on its state. The button has a bitmap for the unpushed state, the hover state (when the mouse moves over the button), and the pushed state. Default bitmaps are included with the sample, but you can override these properties to use your own bitmaps.

This sample also demonstrates the use of several IDL, COM, and compiler attributes. The dispinterface attribute is used to place the dispatch interface _ATLButton in the atlbutn.idl file. The propputref attribute is used to allow references to be used instead of values in the PictureStatic, PictureHover, and PicturePush functions. The version attribute is demonstrated, among other commonly used attributes.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file atlbutn.sln.
2. From the **Build** menu, click **Build Solution**.
3. Open the ActiveX Control Test Container and insert the control (CAtlButton Object). For details on accessing Test Container and using it to test a control, see Testing Properties and Events with Test Container.
4. Open the Properties dialog box for the control, then associate an image with each of the listed properties, by browsing to the sample folder, selecting a BMP file, and clicking **Apply**.
5. Note how the image displayed in the control changes if the pointer hovers over it, or if the control is clicked.

Attributes

This sample uses the following attributes:

coclass, default, dispinterface, dll, dual, event_source, helpstring, id, implements_category, in, module, name, object, out, pointer_default, progid, propget, propput, propputref, registration_script, retval, support_error_info, uuid, version, vi_progid

Keywords

This sample uses the following keywords:

Apartment; ATLTRACE; AtlWaitWithMessageLoop; BEGIN_COM_MAP; BEGIN_CONNECTION_POINT_MAP; BEGIN_MSG_MAP; BEGIN_OBJECT_MAP; BEGIN_PROPERTY_MAP; CComCoClass; CComControl; CComControl::FireViewChange; CComControl::InPlaceActivate; CComModule::GetClassObject; CComModule::GetLockCount; CComModule::Init; CComModule::RegisterServer; CComModule::Term; CComModule::UnregisterServer; CComObjectRoot; CloseHandle; CoGetInterfaceAndReleaseStream; CoInitialize; COleControl::OnClick; COM_INTERFACE_ENTRY; COM_INTERFACE_ENTRY_IMPL_IID; CoMarshalInterThreadInterfaceInStream; CONNECTION_POINT_ENTRY; CProxy_ATLButton; CreateDIBPalette; CreatePalette; CreateThread; CTimer; DECLARE_HANDLE; DECLARE_REGISTRY_RESOURCEID; DIBNumColors; DisableThreadLibraryCalls; DLL_PROCESS_DETACH; END_COM_MAP; END_CONNECTION_POINT_MAP; END_MSG_MAP; END_OBJECT_MAP; END_PROPERTY_MAP; GetCursorPos; GetFileSize; GetWindow; GlobalAlloc; GlobalFree; GlobalLock; GlobalSize; GlobalUnlock; IConnectionPointContainerImpl; IConnectionPointImpl; IDispatchImpl; IObjectSafetyImpl; IOleControlImpl; IOleInPlaceActiveObjectImpl; IOleInPlaceObjectWindowlessImpl; IOleInPlaceObjectWindowlessImpl::InPlaceDeactivate; IOleObjectImpl; IPersistPropertyBagImpl; IPersistStreamInitImpl; IProvideClassInfo2Impl; IViewObjectExImpl; LPLOGPALETTE; MESSAGE_HANDLER; OBJECT_ENTRY; PaintDIB; PaletteSize; PROP_ENTRY; PtInRect; PutImage; ReadDIBFile; ReadFile; RealizePalette; ScreenToClient; SelectPalette; SetDIBitsToDevice; SetStretchBltMode; Sleep; StretchDIBits; TimerOff; TimerOn; Unlock; USES_CONVERSION

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

ATLCON Attributes Sample: Demonstrates Creating a Simple Container

Download sample

The ATLCON attribute sample demonstrates creating a simple container. The container implements the **IOleClientSite**, **IOleWindow**, and **IOleInPlaceSite** interfaces.

The ATLCON sample is the nonattributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution ATLCon.sln.
2. From the **Build** menu, click **Build Solution**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. Once the sample is running, use its **File** menu to insert a control into the container.

Attributes

This sample uses the following attributes:

coclass, default, dual, exe, helpstring, id, in, module, name, object, out, pointer_default, propget, propput, retval, uuid

Keywords

This sample uses the following keywords:

TLASSERT; ATLTRACENOTIMPL; BEGIN_COM_MAP; BEGIN_MSG_MAP; BEGIN_OBJECT_MAP; CComCoClass; CComModule::Unlock; CComObjectRoot; CComQIPtr; CExeModule::Init; CExeModule::RegisterClassObjects; CExeModule::RegisterServer; CExeModule::RevokeClassObjects; CExeModule::Unlock; CExeModule::UnregisterServer; CExeModule::UpdateRegistryFromResource; Close; CoCreateInstance; CoInitialize; COM_INTERFACE_ENTRY; COMMAND_ID_HANDLER; CoUninitialize; CWindowImpl; DECLARE_REGISTRY_RESOURCEID; DispatchMessage; DoVerb; END_COM_MAP; END_MSG_MAP; END_OBJECT_MAP; GetClientRect; GetCommandLine; GetCurrentThreadId; GetDesktopWindow; GetMessage; GetResourceInstance; IAtlCon; IOleClientSite; IOleInPlaceSite; LoadMenu; LPOLEINPLACEFRAMEINFO; LPRECT; MESSAGE_HANDLER; MessageBox; OBJECT_ENTRY; PostQuitMessage; PostThreadMessage; SetClientSite; SetObjectRects; ShowWindow; TranslateMessage; ZeroMemory

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

ATLDuck Attributes Sample: Uses Connection Points with ATL

Download sample

The ATLDuck attributes sample consists of two projects: atlduck and duck. The duck project creates an instance of an object that implements the `IDuckInt` interface. This interface includes four member functions: `Flap`, `Paddle`, `Quack`, and `Walk`.

The second project, atlduck, has a connection point for the `IDuckInt` interface (that is, it knows how to use the interface but does not implement it). Only a single instance of this object will be created. Once the connection is established between the two applications, atlduck will call the functions in the `IDuckInt` interface for sinks that have called **IConnectionPoint::Advise**.

Various IDL, COM, and compiler attributes are demonstrated in ATLDuck, including aggregatable, event_source, and event_receiver. The `DuckInt` class is made unaggregratable by specifying the aggregratable("never") attribute. The `DuckDoer` class is set up as a com event_source, and the `DuckInt` class as a com event_receiver, without layout dependent=true.

The ATLDuck sample is the nonattributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build this sample

1. Open the solution file atlduck.sln.
2. From the **Build** menu, click **Build Solution**. This will build both projects and perform the necessary registration. The following files will be created:
   - atlduck.exe, an EXE server
   - duck.exe, an EXE client

To run this sample

1. From the **Debug** menu, click **Start Without Debugging**. This will start the atlduck server and one instance of the duck client. To run more than one client, navigate to the directory where duck.exe is located and run it manually from the command prompt.
2. A dialog box appears. In the dialog box, click the button **Create DoDuck Object**. When you click this button, the application creates an instance of an object with class `ID CLSID_DuckDoer`, running atlduck.
3. A new dialog box, issued by atlduck.exe, appears. This dialog box shows a button for each function in the `IDuckInt` interface, as well as a list box with the connections to active sinks and their cookies. From the duck dialog boxes, you can either **Advise** or **Unadvise** the connection point. Depending on your choice, you will receive or not receive notifications from the sources. The notification, when you receive it, will show in the status edit field. Additionally, ATLDuck demonstrates the value of the cookie supplied by the connection point when the connection is advised.

Attributes

This sample uses the following attributes:

- **ATLDuck** aggregatable, coclass, dispinterface, event_source, exe, helpstring, id, module, name, progid, uuid, vi_progid
- **ATLDuck/Duck** aggregatable, coclass, default, dispinterface, event_receiver, exe, helpstring, id, iid_is, in, module, name, object, out, progid, unique, uuid, version, vi_progid

Classes and Keywords

The sample use the following classes:

CDialog (MFC), CComObject (ATL), IConnectionPointImpl<CDuckDoer> (ATL)

This sample uses the following keywords:

_ASSERTE; _tcslen; _VERIFY; AddRef; Advise; BEGIN_COM_MAP; BEGIN_CONNECTION_POINT_MAP; BEGIN_MSG_MAP; BEGIN_OBJECT_MAP; CComCoClass; CComModule::Unlock; CComObjectRootEx; CDialogImpl; CenterWindow; CoInitialize; COM_INTERFACE_ENTRY; COM_INTERFACE_ENTRY_IMPL; COMMAND_HANDLER; COMMAND_ID_HANDLER; CONNECTION_POINT_ENTRY; CoSuspendClassObjects; CoUninitialize; Create; DECLARE_CLASSFACTORY_SINGLETON; DECLARE_NOT_AGGREGATABLE; DECLARE_REGISTRY_RESOURCEID; DestroyWindow; DispatchMessage; EnableWindow; END_COM_MAP; END_CONNECTION_POINT_MAP; END_MSG_MAP; END_OBJECT_MAP; ExitProcess; FindConnectionPoint; GetCommandLine; GetDC; GetDlgItem; GetMessage; GetTextExtentPoint32; IConnectionPointContainerImpl; IConnectionPointImpl; Init; IsWindowVisible; MESSAGE_HANDLER; MessageBox; OBJECT_ENTRY; OleInitializeCoCreateInstance; OnCancel; OnFlap; OnInitDialog; OnOK; OnPaddle; OnWalk OnQuack; PostThreadMessage; QueryInterface; RecalcListboxExtent; RegisterClassObjects; RegisterServer; reinterpret_cast; ReleaseDC; RevokeClassObjects; SendMessage; SetOwner; SetWindowText; ShowStatus; ShowWindow; Unadvise; UnregisterServer; UNUSED_ALWAYS; UpdateRegistryFromResource; UpdateWindow; USES_CONVERSION

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

ATLFire Attributes Sample: Demonstrates Building a Windowed Control

Download sample

The ATLFire attributes sample is an ActiveX control that demonstrates how to build a windowed control using ATL by setting the **m_bWindowedOnly** flag to **TRUE**. It also shows what needs to be done to the MFC drawing code to convert it into straight Win32 code. The sample uses the ATL support for Win32 dialog boxes and property sheets and also demonstrates how to use a Win32 tab control in an ActiveX control. The sample also demonstrates several of the ATL macros.

The ATLFire sample is the nonattributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file ATLFire.sln.
2. From the **Build** menu, click **Build**.
3. After the sample builds, open FireTabCtrl.htm in your Web browser and try out the various types of fire that the sample simulates.

You can test the ATLFire control in the ActiveX Control Test Container. For details on accessing Test Container and using it to test a control, see Testing Properties and Events with Test Container.

Attributes

This sample uses the following attributes:

coclass, dual, emitidl, helpstring, id, in, module, object, out, pointer_default, progid, propget, propput, registration_script, retval, support_error_info, threading, uuid, version

Classes and Keywords

This sample uses the following ATL classes:

CComObjectRootEx, CComCoClass, CComControl, IDispatchImpl, IProvideClassInfo2Impl, IPersistStreamInitImpl, IPersistStorageImpl, IPersistPropertyBagImpl, IPerPropertyBrowsingImpl, IQuickActivateImpl, IObjectSafetyImpl, IOleControlImpl, IOleObjectImpl, IOleInPlaceActiveObjectImpl, IViewObjectExImpl, IOleInPlaceObjectWindowlessImpl, IDataObjectImpl, ISupportErrorInfo, ISpecifyPropertyPagesImpl, IConnectionPointContainerImpl, IPropertyNotifySinkCP, CDialogImpl

This sample uses the following keywords:

_ASSERTE; _itot; _tcstol; ALT_MSG_MAP; Apply; ATLTRACE; BEGIN_COM_MAP; BEGIN_CONNECTION_POINT_MAP; BEGIN_MSG_MAP; BEGIN_OBJECT_MAP; BEGIN_PROPERTY_MAP; CComCoClass; CComModule::GetClassObject; CComModule::GetLockCount; CComModule::Init; CComModule::RegisterServer; CComModule::Term; CComModule::UnregisterServer; CComObjectRootEx; CDialogImpl; ClientToScreen; COM_INTERFACE_ENTRY; COM_INTERFACE_ENTRY_IMPL; COM_INTERFACE_ENTRY_IMPL_IID; COMMAND_HANDLER; COMMAND_ID_HANDLER; CONNECTION_POINT_ENTRY; CPropDlg::DoModal; CreateBitmap; CreatePalette; DECLARE_REGISTRY_RESOURCEID; DeleteDC; DeleteObject; DestroyMenu; DisableThreadLibraryCalls; EnableWindow; END_COM_MAP; END_CONNECTION_POINT_MAP; END_MSG_MAP; END_OBJECT_MAP; END_PROPERTY_MAP; EndDialog; FillRect; FireOnChanged; GetActiveWindow; GetClientRect; GetDlgItem; GetDlgItemText; GetModuleInstance; GetWindowRect; IConnectionPointContainerImpl; IDataObjectImpl; IDispatchImpl; InitFire; InlineIsEqualGUID; IObjectSafetyImpl; IOleControlImpl; IOleInPlaceActiveObjectImpl; IOleInPlaceObjectWindowlessImpl; IOleObjectImpl; IPersistPropertyBagImpl; IPersistStorageImpl; IPersistStreamInitImpl; IPropertyNotifySinkCP; IProvideClassInfo2Impl; IQuickActivateImpl; ISpecifyPropertyPagesImpl; ISupportErrorInfo;

IViewObjectExImpl; memcpy; MESSAGE_HANDLER; MessageBox; NOTIFY_CODE_HANDLER; OBJECT_ENTRY; OnActivate; OnApply; OnCancel; OnCreate; OnDestroy; OnEraseBackground; OnInitDialog; OnOK; OnPaint; OnPaletteChanged; OnProperties; OnPropertyChanged; OnQueryNewPalette; OnRButtonDown; OnSelChanged; OnSelChanging; OnSize; OnTimer; PAINTSTRUCT ; PeekMessage; PROP_ENTRY; RealizePalette; SelectObject; SendMessage; SetTimer; SetWindowPos; TrackPopupMenuEx; UnregisterClass

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

ATLMovie Attributes Sample: Uses Compiler COM Support and Active Movie Interfaces in an ATL Control

Download sample

The ATLMovie attributes sample is an ATL control that demonstrates how to use compiler COM support and the Active Movie interfaces to play a movie. It implements several COM properties and methods on its `IMovieCtl` interface.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file ATLMovie.sln.
2. From the **Build** menu, click **Build**.
3. After the sample builds, open MovieCtl.htm in your Web browser. This will play the clock.avi file, which ships with Windows.

   You may need to edit MovieCtl.htm and update the line:

   ```
   MovieCtl.FileName = "C:\Winnt\Clock.AVI"
   ```

   to point to the location of the Clock.AVI file on your particular system.

Properties and Methods

Properties

Name	Description
FileName	The file name of the movie to play.

Methods

Name	Description
Pause	Pause the currently playing movie.
Play	Play the movie pointed to by the **FileName** property.
Reset	Reset the playing position to the beginning of the movie.
Stop	Stop the movie.

Attributes

This sample uses the following attributes:

coclass, dual, emitidl, helpstring, id, in, module, object, oleautomation, pointer_default, progid, propput, registration_script, threading, uuid

Keywords

This sample uses the following keywords:

BEGIN_COM_MAP; BEGIN_MSG_MAP; BEGIN_OBJECT_MAP; BEGIN_PROPERTY_MAP; CComCoClass; CComControl; CComModule::GetClassObject; CComModule::GetLockCount; CComModule::Init; CComModule::RegisterServer; CComModule::Term; CComModule::UnregisterServer; CComObjectRootEx; COM_INTERFACE_ENTRY;

COM_INTERFACE_ENTRY_IMPL; COM_INTERFACE_ENTRY_IMPL_IID; CreateBrushIndirect; CreateFilterGraph; CWindow::GetDC; DECLARE_REGISTRY_RESOURCEID; DeleteObject; DisableThreadLibraryCalls; END_COM_MAP; END_MSG_MAP; END_OBJECT_MAP; END_PROPERTY_MAP; FillRect; GetProperty; IDispatchImpl; IMediaControlPtr::Pause; IMediaControlPtr::Release; IMediaControlPtr::RenderFile; IMediaControlPtr::Run; IMediaControlPtr::Stop; IMediaPositionPtr::CurrentPosition; IMediaPositionPtr::Duration; IObjectSafetyImpl; IOleControlImpl; IOleInPlaceActiveObjectImpl; IOleInPlaceObjectWindowlessImpl; IOleInPlaceObjectWindowlessImpl::SetObjectRects; IOleObjectImpl; IPersistStorageImpl; IPersistStreamInitImpl; IProvideClassInfo2Impl; IVideoWindowPtr ::Owner IVideoWindowPtr::.CreateInstance; IVideoWindowPtr::Owner ; IVideoWindowPtr::SetWindowPosition; IVideoWindowPtr::Visible ; IVideoWindowPtr::WindowStyle ; IViewObjectExImpl; MESSAGE_HANDLER; OBJECT_ENTRY; OffsetRect; OleTranslateColor; Reset; SetWindowPosition; SysFreeString

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

ATLTangram Attributes Sample: Demonstrates Managing Large Projects That Use ATL, MFC, and COM

Download sample

ATLTangram is a port of the Tangram example found in the last chapter of Dale Rogerson's *Inside COM* (http://mspress.microsoft.com). Many thanks to Dale for doing all the hard work and allowing us to use the code in an ATL sample. This sample will help you convert a legacy COM application to one that uses ATL for its infrastructure.

ATLTangram is a large project consisting of the ATLTangram solution, which is the master controller for six subprojects: MFCTangram, ATLModel, ATLGdiWorld, ATLGLWorld, ATLModelExe, and ATLTangramCanvas. The sample demonstrates several features of the integrated development environment (IDE) and several COM concepts. The sample also demonstrates using MFC as the client of ATL COM servers.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file atltangram.sln.
2. From the **Build** menu, click **Build Solution**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. A dialog box will appear: select either option. The Tangram program will begin.

Managing a Large Project

The solution has established the interdependencies in the IDE. These interdependencies work with the default directory structure for the sample and are path-relative.

The dependency hierarchy looks roughly like this:

```
MFCTangram
   ATLGLWorld
      ATLGdiWorld
         ATLTangramCanvas
            ATLModel
   ATLModelExe
      ATLModel
      ATLModel
```

All of the project settings are already established for the sample. Follow these steps to examine the project interdependencies.

1. From the **Project** menu, click **Properties**.
2. Select the Atltangram solution in Solution Explorer.
3. In the **Property Pages** dialog box, expand the **Common Properties** folder and select **Debug Source Files**. Examine the **Search these paths for source files** box.
4. Click **Project Dependencies** and examine the dependent project names.

The ATLModel and ATLModelExe projects demonstrate how to set up a COM server so you can build it as either an in-proc server or a local server using the same set of files and two project files. Using two project files allows dependencies to exist on both the DLL and the EXE.

COM/ATL Features

This sample is a COM system that consists of several COM servers and an MFC application that uses the servers. The sample exhibits the intermodule communication via the connection point mechanism and demonstrates local and in-proc servers.

Other Demonstrated Features

- The ATL servers use the Standard Template Library for collections.
- The MFC driver uses MFC template classes.
- The MFC driver, an example of a non-document/view application, uses a class derived from **CFrameWnd** as the output window for the drawing the Tangram pieces.

Attributes

This sample uses the following attributes:

- **ATLTANGRAM** export, helpstring, object, pointer_default, uuid
- **ATLTANGRAM/atlgdiworld** coclass, com_interface_entry, default, dll, event_receiver, helpstring, iid_is, implements_category, in, module, name, object, out, pointer_default, progid, registration_script, size_is, unique, uuid, version, vi_progid
- **ATLTANGRAM/atlglworld** coclass, com_interface_entry, default, dll, event_receiver, helpstring, implements_category, in, module, name, object, out, pointer_default, progid, registration_script, uuid, vi_progid
- **ATLTANGRAM/atlmodel** coclass, default, dll, event_source, exe, helpstring, in, module, name, object, out, pointer_default, progid, size_is, uuid
- **ATLTANGRAM/atltangramcanvas** coclass, default, dll, helpstring, in, module, name, object, out, pointer_default, progid, registration_script, uuid, vi_progid

Classes and Keywords

The sample uses the following ATL classes:

CComObjectRootEx, CComCoClass, CComControl, IDispatchImpl, IProvideClassInfo2Impl, IPersistStreamInitImpl, IPersistStorageImpl, IPersistPropertyBagImpl, IPerPropertyBrowsingImpl, IQuickActivateImpl, IObjectSafetyImpl, IOleControlImpl, IOleObjectImpl, IOleInPlaceActiveObjectImpl, IViewObjectExImpl, IOleInPlaceObjectWindowlessImpl, IDataObjectImpl, ISupportErrorInfo, ISpecifyPropertyPagesImpl, IConnectionPointContainerImpl, IPropertyNotifySinkCP, CDialogImpl

The sample uses the following MFC classes:

CFrameWnd, CTypedPtrList<>, CDialog, CWinApp, and additional supporting classes

This sample uses the following keywords:

_ASSERTE; AddRef; AddUpdateRect; Advise; assert; ASSERT; ATLTRACE; auxSolidSphere; BEGIN_COM_MAP; BEGIN_CONNECTION_POINT_MAP; BEGIN_MESSAGE_MAP; BEGIN_OBJECT_MAP; BitBlt; CATEGORYINFO ; CComCoClass; CComModule::GetClassObject; CComModule::GetLockCount; CComModule::Init; CComModule::RegisterServer; CComModule::Term; CComModule::UnregisterServer; CComObjectRootEx; CExeModule::Init; CExeModule::RegisterClassObjects; CExeModule::RegisterServer; CExeModule::RevokeClassObjects; CExeModule::UnregisterServer; CExeModule::UpdateRegistryFromResource; CFrameWnd::AssertValid; CFrameWnd::Dump; CFrameWnd::PreCreateWindow; ChoosePixelFormat; CModelList; CoCreateInstance; CoInitializeEx; COM_INTERFACE_ENTRY; COM_INTERFACE_ENTRY_AGGREGATE; COM_INTERFACE_ENTRY_IMPL; CONNECTION_POINT_ENTRY; CopyRect; CoTaskMemFree; CProxyIATLTangramModelEvent; CreateCompatibleDC; CreatePalette; CWnd::CreateEx; DECLARE_GET_CONTROLLING_UNKNOWN; DECLARE_MESSAGE_MAP; DECLARE_ONLY_AGGREGATABLE; DECLARE_REGISTRY_RESOURCEID; DeleteObject; DescribePixelFormat; DisableThreadLibraryCalls; DispatchMessage; DoButtonDown; DoModal; END_COM_MAP; END_CONNECTION_POINT_MAP; END_MESSAGE_MAP; END_OBJECT_MAP; EqualRect; ErrorMessage; GdiFlush; GetBoundingRect; GetClientRect; GetCommandLine; GetControllingUnknown; GetDC; GetMessage; GetModuleFileName; GetObject; GetPalette; GetPaletteEntries; GetPixelFormat; GetRotation; GetVertices; glBegin; glClearColor; GLdouble CoCreateInstance; glEnable; glEnd; glFlush; glGetIntegerv; glInitNames; glLightfv; glLightModelfv; glLoadIdentity; glMatrixMode; glNormal3d; glPolygonMode; glPopMatrix; glPopName; glPushMatrix; glPushName; GLRender; glRenderMode; GLResize; glRotated; glSelectBuffer; GLSetup; glTranslated; glTranslatef; gluPerspective; gluPickMatrix; gluUnProject; glVertex2d; glVertex3d; glViewport; HPALETTE; ICatInformation::EnumClassesOfCategories; ICatRegister::QueryInterface; ICatRegister::RegisterCategories; ICatRegister::RegisterClassImplCategories;

ICatRegister::UnRegisterCategories; ICatRegister::UnRegisterClassImplCategories; IConnectionPointContainer::FindConnectionPoint; IConnectionPointContainer::Release; IConnectionPointContainerImpl; InitInstance; InvalidateRect; IsCurrent; IsValidAddress; IUnknown::Release; LoadIcon; LoadStandardCursor; LocalFree; MakeCurrent; OBJECT_ENTRY; ON_COMMAND; ON_WM_DESTROY ; OnCancel; OnDestroy; OnInitDialog; OnOK; OnQueryNewPalette; OutputDebugString; OutputGlError; Polygon; PreCreateWindow; PtInRegion; QueryInterface; RealizePalette; Release; ReleaseConnectionPoint; ReleaseDC; Rotate; SelectObject; SelectPalette; SetPixelFormat; SetRectEmpty; specifyMaterial; StringFromCLSID; SubkeyExists; va_end; wcscpy; wglCreateContext; wglGetCurrentContext; wglMakeCurrent

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

AutoThread Attributes Sample: Demonstrates the Use of CAtlAutoThreadModule

Download sample

The AutoThread sample demonstrates using **CAtlAutoThreadModule**. The server is implemented in the Server.exe file. The module of the EXE is derived from **CAtlAutoThreadModule** instead of **CAtlModule**.

The AutoThread sample is the nonattributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

To build and run this sample

1. Open the solution file AutoThread.sln.
2. From the **Build** menu, click **Build Solution**. This will build and register the client and the server.
3. Start two instances of the ActiveX Control Test Container. See Testing Properties and Events with Test Container for information on how to access the test container.
4. Insert one of the client controls (CAutoCtl Class) into each of the test containers.
5. Click one of the controls and notice that it takes one second for the server to return.
6. Position and resize the test containers so that both of them are visible at the same time.
7. Click one of the controls and then quickly click the other control. Notice that they finish waiting at approximately the same time. (If **CComAutoThreadModule** were not used, the first control would finish after one second, but the second control would not finish until a full second after the first control finished. The second call to sleep would not occur until the first had finished.) You can use the **Delay(PropGet)** and **Delay(PropPut)** methods to adjust the number of milliseconds the server sleeps for. If set properly, the second call to sleep may return before the first call to sleep.

How the Sample Works

The server interface has a single method: **Sleep**. This method puts the server thread to sleep for a given amount of time. The client portion of the sample is an ActiveX control that invokes the server's sleep method when the user clicks the control. The client also has a property named **Delay** that represents how long the server thread will sleep. The control displays the text "Ready" when it is waiting for a user click. The text "Waiting" is displayed when the control is waiting for the server to finish sleeping.

Attributes

This sample uses the following attributes:

- **AutoThread/AutoClient** coclass, dual, emitidl, helpstring, id, in, module, object, out, pointer_default, progid, propget, propput, registration_script, retval, threading, uuid, version, vi_progid
- **AutoThread/AutoServer** coclass, dual, emitidl, helpstring, id, module, object, pointer_default, progid, threading, uuid, version, vi_progid

Keywords

This sample uses the following keywords:

AtlGetObjectSourceInterface; BEGIN_SINK_MAP; CoCreateInstance; DECLARE_CLASSFACTORY_SINGLETON; IDispEventSimpleImpl::DispEventAdvise; END_SINK_MAP; IConnectionPointContainerImpl; IConnectionPointImpl; IDispEventImpl; OLE2CT; SINK_ENTRY_EX; SysAllocString; USES_CONVERSION; VARIANT; VariantClear; VariantCopy

Note

Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

BEEPER Attributes Sample: Demonstrates a Tearoff Interface

Download sample

The BEEPER attributes sample illustrates the implementation of a tearoff interface — a collection/enumeration of **BSTR** values. It can be built as a DLL and as an EXE.

The BEEPER sample is the nonattributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file beeper.sln.
2. From the **Build** menu, click **Build Solution**.
3. After the sample builds, open beeper.htm (an HTML file that uses VBScript) in your Web browser and follow its instructions.

Attributes

This sample uses the following attributes:

aggregatable, coclass, dual, emitidl, helpstring, id, in, module, object, out, pointer_default, progid, propget, restricted, retval, support_error_info, threading, uuid, version

Keywords

This sample uses the following keywords:

BEGIN_COM_MAP; CComCoClass; CComModule::Unlock; CComObjectRoot; COM_INTERFACE_ENTRY; COM_INTERFACE_ENTRY_TEAR_OFF; DECLARE_GET_CONTROLLING_UNKNOWN; DECLARE_NOT_AGGREGATABLE; DECLARE_REGISTRY; END_COM_MAP; FinalConstruct; IDispatchImpl; ISupportErrorInfo; PostThreadMessage; return Error; SysAllocString; THREADFLAGS_BOTH

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

CDINFO Attributes Sample: Plays and Displays Information About CD Audio Tracks

Download sample

The CDINFO attributes sample uses the Media Control Interface (MCI) APIs to retrieve the length of the audio tracks on the CD currently in your CD-ROM drive. CDINFO then draws a piechart to display track length. CDINFO also demonstrates how to use the ToolTip common control to implement a tool tip. This tool tip displays the audio track length corresponding to the cursor's location in the piechart.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file CDInfo.sln.
2. From the **Build** menu, click **Build**.
3. After the sample builds, open CDInfo.htm in your Web browser and follow its instructions.

Properties and Methods

The control implements the following methods:

Read Reads the track length information from the CD currently in the drive and displays the information in the form of a piechart.

Redraw Redraws the piechart for the current CD.

Play Starts playing the CD at the specified track number. The control also rotates the piechart so that the track playing is at the top.

The following properties are available:

Tracks The number of tracks on the CD

Length The length of the specified track number in seconds

TotalLength The total length of the CD in seconds

Attributes

This sample uses the following attributes:

coclass, default, dispinterface, dual, emitidl, event_source, helpstring, id, implements_category, in, module, object, out, pointer_default, progid, propget, propput, registration_script, retval, threading, uuid, version, vi_progid

Keywords

This sample uses the following keywords:

BEGIN_COM_MAP; BEGIN_CONNECTION_POINT_MAP; BEGIN_MSG_MAP; BEGIN_OBJECT_MAP; BEGIN_PROPERTY_MAP; CComCoClass; CComControl; CComObjectRoot; COM_INTERFACE_ENTRY; COM_INTERFACE_ENTRY_IMPL; CONNECTION_POINT_ENTRY; CProxyCDEvents; CreateBrushIndirect; CreateRectRgn; DECLARE_REGISTRY_RESOURCEID; DeleteDC; DeleteObject; DrawCD; Ellipse; END_COM_MAP; END_CONNECTION_POINT_MAP; END_MSG_MAP; END_OBJECT_MAP; END_PROPERTY_MAP; IConnectionPointContainerImpl; IConnectionPointImpl; IDataObjectImpl; IDispatchImpl; IOleControlImpl; IOleInPlaceActiveObjectImpl; IOleInPlaceObjectWindowlessImpl; IOleLinkImpl; IOleObjectImpl; IPerPropertyBrowsingImpl; IPersistStorageImpl; IPersistStreamInitImpl; IPropertyNotifySinkCP; IProvideClassInfo2Impl; IQuickActivateImpl; IRunnableObjectImpl; IViewObjectExImpl; LineTo; mciSendCommand; MESSAGE_HANDLER; MoveToEx;

OBJECT_ENTRY; ReduceRect; RelayEvent; SelectClipRgn; SelectObject; VariantInit; ZeroMemory

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

CIRC Attributes Sample: Demonstrates Using Property Pages

Download sample

The CIRC attributes sample shows how to use ATL to make a simple ActiveX control. The sample creates a control that draws a circle and demonstrates property pages, control painting, use of color and caption properties, Click and KeyPress events, and a Color property page.

The CIRC sample is the nonattributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file circ.sln.
2. From the **Build** menu, click **Build Solution**.
3. Insert the control into a control container like ActiveX Control Test Container and test its properties and events. For details on accessing Test Container and using it to test a control, see Testing Properties and Events with Test Container.

Attributes

This sample uses the following attributes:

coclass, dispinterface, dll, dual, event_source, helpstring, id, implements_category, in, module, name, object, out, pointer_default, propget, propput, propputref, retval, support_error_info, uuid

Keywords

This sample uses the following keywords:

AtlCreateTargetDC; ATLTRACE; BEGIN_COM_MAP; BEGIN_CONNECTION_POINT_MAP; BEGIN_MSG_MAP; BEGIN_OBJECT_MAP; BEGIN_PROPERTY_MAP; CCircPropertyPageDlg ; CComCoClass; CComControl; CComControl::SetDirty; CComObjectRoot; CDialogImpl; CHAIN_MSG_MAP; COM_INTERFACE_ENTRY; COM_INTERFACE_ENTRY_IID; COM_INTERFACE_ENTRY_IMPL; COMMAND_ID_HANDLER; CONNECTION_POINT_ENTRY; CProxy_CircEvents; CreateEllipticRgn; CreatePen; CreateSolidBrush; CStockPropImpl; DECLARE_GET_CONTROLLING_UNKNOWN; DECLARE_REGISTRY_RESOURCEID; Ellipse; END_COM_MAP; END_CONNECTION_POINT_MAP; END_MSG_MAP; END_OBJECT_MAP; END_PROPERTY_MAP; GetClientRect; GetDialogBaseUnits; GetStockObject; GetTextMetrics; IConnectionPointContainerImpl; IConnectionPointImpl; IDataObjectImpl; IOleControlImpl; IOleInPlaceActiveObjectImpl; IOleInPlaceObjectWindowlessImpl; IOleLinkImpl; IOleObjectImpl; IOleObjectImpl::DoVerb; IPerPropertyBrowsingImpl; IPersistPropertyBagImpl; IPersistStorageImpl; IPersistStreamInitImpl; IPropertyNotifySinkCP; IPropertyPageImpl; IPropertyPageImpl::SetObjects; IProvideClassInfo2Impl; IQuickActivateImpl; IQuickActivateImpl::QuickActivate; IRunnableObjectImpl; ISpecifyPropertyPagesImpl; IViewObjectExImpl; MESSAGE_HANDLER; OBJECT_ENTRY; OleTranslateColor; PROP_ENTRY; SelectObject; SendDlgItemMessage; SetBkMode; SetTextColor; SetWindowRgn; USES_CONVERSION

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

CIRCCOLL Attributes Sample: Demonstrates a Collection/Enumeration

Download sample

The CIRCCOLL attributes sample shows how to implement a collection/enumeration of objects using ATL.

The CIRCCOLL sample is the nonattributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file circoll.sln.
2. From the **Build** menu, click **Build Solution**.
3. In Solution Explorer, right-click the CircCollClient project and select **Set as StartUp Project**.
4. From the **Debug** menu, click **Start**.
5. The Visual Basic form, **Form1**, will open. Click the button to activate the application.

How the Sample Works

Three object classes are implemented: the collection creator, the collection, and the object inside the collection. Only the collection creator has a **coclass** associated with it (see Circcoll.idl for the **coclass** declaration). The collection creator has a method to return a collection of circle objects. The collection object implements the **Item**, **Count**, and **_NewItem** methods, so that the object can be manipulated from Visual Basic using both the **For...Next** and **For Each...** syntax.

Attributes

This sample uses the following attributes:

aggregatable, coclass, dual, emitidl, helpstring, id, in, module, noncreatable, object, out, pointer_default, progid, propget, propput, restricted, retval, support_error_info, threading, uuid, version

Keywords

This sample uses the following keywords:

AddRef; ATLASSERT; ATLTRACE; BEGIN_COM_MAP; BEGIN_OBJECT_MAP; CComCoClass; CComObject::CreateInstance; CComObjectRoot; CComVariant; COM_INTERFACE_ENTRY; DECLARE_NOT_AGGREGATABLE; DECLARE_REGISTRY; END_COM_MAP; END_OBJECT_MAP; glBegin; glEnd; glNormal3d; glPolygonMode; glVertex2d; glVertex3d; IConnectionPointContainerImpl::FindConnectionPoint; IConnectionPointImpl::Advise; IDispatchImpl; ISupportErrorInfo; OBJECT_ENTRY; QueryInterface; Release; USES_CONVERSION; VariantCopy; VariantInit

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

COMMAP Attributes Sample: Demonstrates COM Interface Map Entry Macros

Download sample

The COMMAP attributes sample shows how different COM interface map entry macros are used.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file COMMap.sln.
2. From the **Build** menu, click **Build Solution**.
3. After the sample builds, open commap.htm in your Web browser. Commap.htm includes comments about each type of map entry.

Attributes

This sample uses the following attributes:

- **COMMap** coclass, com_interface_entry, default, dual, emitidl, helpstring, id, implements_category, in, module, object, out, pointer_default, progid, propget, retval, threading, uuid, version, vi_progid
- **COMMap/Aggreg** coclass, default, dual, emitidl, helpstring, id, module, object, out, pointer_default, progid, propget, retval, support_error_info, threading, uuid, version, vi_progid

Keywords

This sample uses the following keywords:

BEGIN_COM_MAP; CComCoClass; CComModule::GetClassObject; CComModule::GetLockCount; CComModule::Init; CComModule::RegisterServer; CComModule::Term; CComModule::UnregisterServer; CComObjectRoot; CComTearOffObjectBase; COM_INTERFACE_ENTRY; COM_INTERFACE_ENTRY_AGGREGATE; COM_INTERFACE_ENTRY_AGGREGATE_BLIND; COM_INTERFACE_ENTRY_AUTOAGGREGATE; COM_INTERFACE_ENTRY_AUTOAGGREGATE_BLIND; COM_INTERFACE_ENTRY_CACHED_TEAR_OFF; COM_INTERFACE_ENTRY_TEAR_OFF; DECLARE_GET_CONTROLLING_UNKNOWN; DECLARE_REGISTRY_RESOURCEID; DisableThreadLibraryCalls; END_COM_MAP; IDispatchImpl; ISupportErrorInfo; SysAllocString

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

CONNECT Attributes Sample: Demonstrates Implementation and Use of Connection Points

Download sample

The CONNECT attributes sample illustrates the implementation and use of connection points (the **IConnectionPointContainer** and **IConnectionPoint** interfaces) in a multithreaded environment.

The sample demonstrates several commonly used IDL, COM, and compiler attributes, including support_errorinfo, event_source, and event_receiver.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file connect.sln.
2. From the **Build** menu, click **Build Solution**.
3. Select which client you want to run, Drive or MDrive, and make it the startup project (right-click the project folder and click **Set as StartUp Project**). More information about the clients can be found in the How the Sample Works section.
4. On the **Debug** menu, click **Start Without Debugging**.

How the Sample Works

The server is implemented in Connect.dll. This DLL allows the creation of a `CoRandom` COM object, implemented by the `CRandom` C++ class. The COM object supports `IRandom` (a dual interface) and **IConnectionPointContainer**, and it accepts connections for the `IRandomEvent` interface.

The `IRandom` interface supports the following methods:

- `Start` — starts a thread inside the object
- `Stop` — stops a thread inside the object
- `StopAll` — stops all of the running threads

When running, the secondary threads inside the object keep firing events through the connection point.

Two clients are provided: Drive and MDrive. They can be found in the Drive and MDrive subdirectories.

- Drive.exe is a simple console application that provides a single object implementing the `IRandomEvent` interface. It creates a `CoRandom` object, calls `Advise` and `Unadvise` on the connection point, and makes the `CoRandom` object fire events into the drive's object.
- Mdrive.exe is an MFC dialog-based application, able to create multiple advise sinks and control the number of threads the server creates. When you run Mdrive.exe, click the **Start** button at least once, then click the **Advise** button several times. Each click of the **Advise** button adds a connection point, which makes the display wider. If you do not click the **Advise** button, you will not see any activity in the display.

Attributes

This sample uses the following attributes:

- **Connect** coclass, default, dll, dual, event_source, helpstring, id, in, module, name, object, out, pointer_default, progid, support_error_info, uuid, vi_progid

- **Connect/drive** event_receiver, module

Keywords

This sample uses the following keywords:

AfxGetApp; AfxMessageBox; AtlAdvise; ATLASSERT; AtlUnadvise; BEGIN_COM_MAP; BEGIN_CONNECTION_POINT_MAP; BEGIN_MESSAGE_MAP; BEGIN_OBJECT_MAP; CComCoClass; CComModule::GetClassObject; CComModule::GetLockCount; CComModule::Init; CComModule::RegisterServer; CComModule::Term; CComModule::UnregisterServer; CComObject::CreateInstance; CComObjectRoot; CDialog::OnCancel; CDialog::OnOK; CloseHandle; CoCreateInstance; COleTemplateServer::RegisterAll; COM_INTERFACE_ENTRY; COM_INTERFACE_ENTRY_IMPL; COM_INTERFACE_ENTRY2; CONNECTION_POINT_ENTRY; CoUninitialize; CreateEvent; CreateThread; DECLARE_REGISTRY_RESOURCEID; DisableThreadLibraryCalls; DoModal; DrawIcon; END_COM_MAP; END_CONNECTION_POINT_MAP; END_MESSAGE_MAP; END_OBJECT_MAP; GdiFlush; GetClientRect; GetDlgItem; GetTickCount; GetUnknown; IConnectionPointContainerImpl; IConnectionPointImpl; IDispatchImpl; ISupportErrorInfo; LoadIcon; Lock; memset; OBJECT_ENTRY; ON_COMMAND; puts; ReleaseDC; SendMessage; SetEvent; SetIcon; SetPixel; Sleep; Unadvise; Unlock; WaitForSingleObject

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

DIRECT3D Attributes Sample: Demonstrates Using Direct3D

Download sample

The DIRECT3D attributes sample creates a control that draws a spinning triangle using the Direct3D graphics library.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file Direct3D.sln.
2. From the **Build** menu, click **Build Solution**.
3. After the sample builds, open Direct3D.htm in your Web browser.

You can test the control in the ActiveX Control Test Container. For details on accessing Test Container and using it to test a control, see Testing Properties and Events with Test Container.

Attributes

This sample uses the following attributes:

coclass, default, dll, dual, helpstring, implements_category, module, name, object, pointer_default, progid, registration_script, support_error_info, uuid, vi_progid

Keywords

This sample uses the following keywords:

Apartment; ATLASSERT; ATLTRACE; AtlWaitWithMessageLoop; BEGIN_COM_MAP; BEGIN_MSG_MAP; BEGIN_OBJECT_MAP; BEGIN_PROPERTY_MAP; CComCoClass; CComControl; CComControlBase::InPlaceActivate; CComModule::GetClassObject; CComModule::GetLockCount; CComModule::Init; CComModule::RegisterServer; CComModule::Term; CComModule::UnregisterServer; CComObjectRoot; CException::ReportError; CoGetInterfaceAndReleaseStream; CoInitialize; COM_INTERFACE_ENTRY; COM_INTERFACE_ENTRY_IMPL; COM_INTERFACE_ENTRY_IMPL_IID; CoMarshalInterThreadInterfaceInStream; CopyMemory; CoUninitialize; CreatePalette; CreateRectRgn; CreateThread; D3DEXECUTEBUFFERDESC; DDSURFACEDESC; DDSURFACEDESC; DECLARE_GET_CONTROLLING_UNKNOWN; DECLARE_REGISTRY_RESOURCEID; DirectDrawCreate; DirectDrawCreateClipper; DisableThreadLibraryCalls; END_OBJECT_MAP; GetClientRect; GetOurWindow; GetSystemPaletteEntries; IDataObjectImpl; IDirect3D::CreateLight; IDirect3D::CreateMaterial; IDirect3D::CreateViewport; IDirect3DDevice::AddViewport; IDirect3DDevice::BeginScene; IDirect3DDevice::CreateMatrix; IDirect3DDevice::DeleteMatrix; IDirect3DDevice::EndScene; IDirect3DDevice::Execute; IDirect3DViewport::AddLight; IDirect3DViewport::CreateMatrix; IDirect3DViewport::SetBackground; IDirect3DViewport::SetMatrix; IDirectDraw2::EnumDevices; IDirectDraw2::SetCooperativeLevel; IDirectDrawClipper::SetHWnd; IDirectDrawSurface2::AddAttachedSurface; IDirectDrawSurface2::GetSurfaceDesc; IDirectDrawSurface2::Restore; IDirectDrawSurface2::SetClipper; IDispatchImpl; IObjectSafetyImpl; IOleControlImpl; IOleInPlaceActiveObjectImpl; IOleInPlaceObjectWindowlessImpl; IOleInPlaceObjectWindowlessImpl::InPlaceDeactivate; IOleInPlaceObjectWindowlessImpl::SetObjectRects; IOleObjectImpl; IPersistStorageImpl; IPersistStreamInitImpl; IProvideClassInfo2Impl; IQuickActivateImpl; ISupportErrorInfo; IViewObjectExImpl; LPD3DDEVICEDESC; LPD3DINSTRUCTION; LPD3DPROCESSVERTICES; LPD3DSTATE; LPD3DTRIANGLE; LPD3DVERTEX; LPDIRECTDRAWCLIPPER ; MessageBox; OBJECT_ENTRY; OffsetRect; OnEraseBackground; PALETTEENTRY; QueryInterface; RenderScene; SelectClipRgn; SetPalette; WindowFromDC; ZeroMemory

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

DispSink Attributes Sample: Handles Events Fired from a Singleton COM Server Through a Dispatch Interface

Download sample

The DispSink attributes sample demonstrates a singleton server object (an object that can have only one instance) that has its own dual interface and a dispatch interface used for firing off events.

The Attributes Tutorial includes step-by-step procedures for creating the DispSink attributes sample.

The DispSink sample is the nonattributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file DispSink.sln.
2. From the **Build** menu, click **Build Solution**.
3. Open two or more instances of the ActiveX Control Test Container and insert the client control, DispCtl, into each instance. See Testing Properties and Events with Test Container for information on how to access the test container.
4. Invoke the **Connect** method on all of the controls.
5. Invoke the **Send** method on one of the controls. Change the **Parameter Type** field of the **Invoke Methods** dialog box to VT_BSTR and then type any string into the **Parameter Value** box. Click the **Invoke** button. The string will be displayed in the center of all connected controls.
6. Invoke the **Disconnect** method on all controls prior to deleting them.

How the Sample Works

The server is a singleton object that has its own dual interface as well as a dispatch interface used for firing off events. The dispatch interface is placed in the .idl file using the **dispinterface** attribute. The server object receives data through its dual interface **Send** method and transmits it to all connected components through the **Transfer** event on its dispatch interface. The Dispserver uses the **event_source** attribute, and the Dispclient uses the **event_receiver** attribute.

The client is an ActiveX control that contains a server object. The control responds to the **Transfer** event fired by the server object. It has a dual interface that has **Connect**, **Send**, and **Disconnect** methods. If the **Transfer** event is fired with a variant containing a BSTR, the string is displayed in the center of the control.

Attributes

This sample uses the following attributes:

- **DISPSINK/DispClient** coclass, dual, event_receiver, helpstring, id, module, object, pointer_default, registration_script, threading, uuid, version, vi_progid
- **DISPSINK/DispServer** coclass, default, dispinterface, dual, event_source, helpstring, id, module, object, pointer_default, threading, uuid, vi_progid

Keywords

This sample uses the following keywords:

#import; CComAutoThreadModule; CComSimpleThreadAllocator; CoCreateInstance; DECLARE_CLASSFACTORY_AUTO_THREAD; FireViewChange

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

LABRADOR Attributes Sample: Implements a Server with No User Interface

Download sample

The LABRADOR attributes sample shows how to use ATL to implement an EXE server without any user interface. The server allows creation of an object that supports two custom interfaces, defined in Labrador.idl.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

This sample uses three components: the server, the marshaling DLL, and the driver.

To build and register the components

1. Open the solution file Labrador.sln
2. From the **Build** menu, click **Build Solution**.

   The server, the marshaling DLL, and the driver will be built and registered.

To run the driver

1. In Solution Explorer, right-click the labdriv project and select **Set as StartUp Project**.
2. From the **Debug** menu, click **Start Without Debugging**.

   The driver will create an object, make a few calls into it, and then release it.

Attributes

This sample uses the following attributes:

aggregatable, coclass, default, emitidl, helpstring, in, module, object, out, progid, string, threading, uuid, version, vi_progid

Keywords

This sample uses the following keywords:

_CrtDumpMemoryLeaks; _tcsicmp; _tcstok; _tprintf; _vstprintf; ATLASSERT; BEGIN_COM_MAP; BEGIN_OBJECT_MAP; CComModule::Init; CComModule::RegisterClassObjects; CComModule::RevokeClassObjects; CComModule::Unlock; CComModule::UnregisterServer; CComObjectRoot; CoCreateInstance; COM_INTERFACE_ENTRY; CoUninitialize; DECLARE_NOT_AGGREGATABLE; DECLARE_REGISTRY; DispatchMessage; END_OBJECT_MAP; GetCurrentThreadId; GetMessage; OBJECT_ENTRY; OutputDebugString; PostThreadMessage; Trace; va_end; va_list; va_start; wcscpy

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

MultiRead Attributes Sample: Reads Database Table Using Multiple Threads

Download sample

The MultiRead attributes sample demonstrates how to use the OLE DB consumer template classes to read through a table in a database using multiple threads.

The MultiRead sample is the nonattributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

1. Open the solution file MultiRead.sln.
2. From the **Build** menu, click **Build**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. A Multi-Threaded Read dialog box will appear, prompting you to specify the number of threads to use to read through the table. Click **Run**.

   The result will appear as text in the Multi-Threaded Read dialog box, for example, `15 records in 7 ms`.

How the Sample Works

The sample contains the `CMultiDlg` class, which is used to show a dialog box. With this dialog box, the user enters the number of threads to use to read through the table. When the user clicks the **Run** button, **ReadRecords** in DBRead.h is called to open the database, session, and table, and to create the necessary number of threads. As it opens the table, the function sets the **DBPROP_CANHOLDROWS** property to true so that the provider allows the user to retrieve new rows without releasing the previously retrieved rows. This capability is required, because multiple threads retrieve new rows as other threads are still processing their current threads.

The sample also shows how to extend the standard **CRowset** class by creating a new `CMyRowset` class, which derives from **CRowset**, and adds the `MoveAndProcess` member function. The start routine of each thread is the `ReadTable` function and the table class is passed to this function. The function calls the routine `MoveAndProcess` to read through each record. Note that the accessor class `CProduct` is defined so that the data will not be retrieved automatically on the `MoveNext` call. This avoids buffer conflicts with the other threads, and there is no need to protect `MoveNext` with a critical section. The `MoveAndProcess` function calls `MoveNext`, and then calls `GetDataHere` to place the data directly into a local variable of that function. `ProcessRecord` is called for each record retrieved and the function simply traces out the values of the record by default.

Each thread counts the number of records it reads, and those are traced out at the end, along with a total and time taken, which is displayed on the dialog box.

Note
The MultiRead sample reads the MultiRead.mdb database file. The sample code assumes that this file is located in the current directory.

Attributes

This sample uses the following attributes:

db_column, db_source, db_table, exe, helpstring, module, name, threading, uuid, version

Keywords

The sample demonstrates the following classes:

CAccessor, CDataSource, CDBPropSet, CRowset, CSession, CTable

The sample demonstrates the following macros:

BEGIN_ACCESSOR_MAP, BEGIN_ACCESSOR, COLUMN_ENTRY, END_ACCESSOR, END_ACCESSOR_MAP, DEFINE_COMMAND

The sample demonstrates the following functions:

CreateThread, GetCurrentThreadId, GetExitCodeThread, WaitForMultipleObjects

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

OPENGL Attributes Sample: Demonstrates Using OpenGL

Download sample

The OPENGL attributes sample creates a control that draws a spinning cube using the OpenGL graphics library.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file OpenGl.sln.
2. From the **Build** menu, click **Build Solution**.
3. After the sample builds, open OpenGl.htm in your Web browser.

You can test the control in the ActiveX Control Test Container. For details on accessing Test Container and using it to test a control, see Testing Properties and Events with Test Container.

Attributes

This sample uses the following attributes:

coclass, default, dll, dual, helpstring, id, implements_category, in, module, name, object, out, pointer_default, progid, propget, propput, registration_script, retval, uuid, vi_progid

Keywords

This sample uses the following keywords:

ATLASSERT; ATLTRACE; auxWireSphere; BEGIN_COM_MAP; BEGIN_CONNECTION_POINT_MAP; BEGIN_MSG_MAP; BEGIN_OBJECT_MAP; BEGIN_PROPERTY_MAP; CComCoClass; CComControl; CComModule::GetClassObject; CComModule::GetLockCount; CComModule::Init; CComModule::RegisterServer; CComModule::Term; CComModule::UnregisterServer; CComObjectRoot; ChoosePixelFormat; CoTaskMemAlloc; CreateContext; CreateRGBPalette; DECLARE_GET_CONTROLLING_UNKNOWN; DECLARE_REGISTRY_RESOURCEID; DeleteObject; DescribePixelFormat; DisableThreadLibraryCalls; DrawText; END_CONNECTION_POINT_MAP; END_OBJECT_MAP; FireViewChange; GetClientRect; GetPixelFormat; glBegin; glClear; glClearColor; glClearDepth; glColor3f; glEnable; glEnd; glFinish; glLoadIdentity; glMatrixMode; glPopMatrix; glPushMatrix; glRotatef; glTranslatef; gluPerspective; glVertex3f; IDispatchImpl; IObjectSafetyImpl; IOleControlImpl; IOleInPlaceActiveObjectImpl; IOleInPlaceObjectWindowlessImpl ; IOleObjectImpl; IPersistStorageImpl; IPersistStreamInitImpl; IViewObjectExImpl; joyReleaseCapture; joySetThreshold; memcpy; OBJECT_ENTRY; PIXELFORMATDESCRIPTOR; RealizePalette; ReleaseCapture; SelectPalette; SetBkMode; SetCapture; SetTextColor; SwapBuffers; USES_CONVERSION; wglCreateContext; wglDeleteContext; wglGetCurrentDC; wglMakeCurrent

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

POLYGON Attributes Sample: Demonstrates Custom Properties, Events, a Property Page, and Object Safety

Download sample

The POLYGON attributes sample shows how to implement custom properties, events, a property page, and object safety for an ATL control. The POLYGON sample is the nonattributed version of this sample and is used as the basis of the ATL Tutorial.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file Polygon.sln.
2. From the **Build** menu, click **Build**.
3. After the sample builds, open PolyCtl.htm in your Web browser and try out the Polygon control. If you click inside the polygon, the number of sides increases. If you click outside the polygon, the number of sides decreases.

You can test the Polygon control in the ActiveX Control Test Container. For details on accessing Test Container and using it to test a control, see Testing Properties and Events with Test Container.

Attributes

This sample uses the following attributes:

coclass, default, dispinterface, dual, event_source, helpstring, id, implements_category, in, module, object, out, pointer_default, progid, propget, propput, registration_script, retval, support_error_info, threading, uuid, version, vi_progid

Keywords

This sample uses the following keywords:

ATLASSERT; ATLTRACE; auxWireSphere; BEGIN_COM_MAP; BEGIN_CONNECTION_POINT_MAP; BEGIN_MSG_MAP; BEGIN_OBJECT_MAP; BEGIN_PROPERTY_MAP; CComCoClass; CComControl; CComModule::GetClassObject; CComModule::GetLockCount; CComModule::Init; CComModule::RegisterServer; CComModule::Term; CComModule::UnregisterServer; CComObjectRoot; ChoosePixelFormat; CoTaskMemAlloc; CreateContext; CreateRGBPalette; DECLARE_GET_CONTROLLING_UNKNOWN; DECLARE_REGISTRY_RESOURCEID; DeleteObject; DescribePixelFormat; DisableThreadLibraryCalls; DrawText; END_CONNECTION_POINT_MAP; END_OBJECT_MAP; FireViewChange; GetClientRect; GetPixelFormat; glBegin; glClear; glClearColor; glClearDepth; glColor3f; glEnable; glEnd; glFinish; glLoadIdentity; glMatrixMode; glPopMatrix; glPushMatrix; glRotatef; glTranslatef; gluPerspective; glVertex3f; IDispatchImpl; IObjectSafetyImpl; IOleControlImpl; IOleInPlaceActiveObjectImpl; IOleInPlaceObjectWindowlessImpl ; IOleObjectImpl; IPersistStorageImpl; IPersistStreamInitImpl; IViewObjectExImpl; joyReleaseCapture; joySetThreshold; memcpy; OBJECT_ENTRY; PIXELFORMATDESCRIPTOR; RealizePalette; ReleaseCapture; SelectPalette; SetBkMode; SetCapture; SetTextColor; SwapBuffers; USES_CONVERSION; wglCreateContext; wglDeleteContext; wglGetCurrentDC; wglMakeCurrent

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

SUBEDIT Attributes Sample: Superclasses a Standard Windows Control

Download sample

The SUBEDIT attributes sample demonstrates how to create an ATL control that superclasses the standard Windows Edit control.

The SUBEDIT sample is the nonattributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file SubEdit.sln.
2. From the **Build** menu, click **Build**.
3. After the sample builds, open AtlEdit.htm in your Web browser and try out the control.

You can test out the control in the ActiveX Control Test Container by opening the ATLEdit Class. For details on accessing Test Container and using it to test a control, see Testing Properties and Events with Test Container.

Superclassing a Windows Control

ATL provides the ability to create a control that superclasses a standard Windows control. Superclassing allows you to create a window class that is based on an existing class but uses a different window procedure. You then create a window based on this new window class. When you superclass a control, messages are first processed by an ATL message map before being sent to the control's original window procedure. This allows you to modify the default behavior of standard Windows controls.

When you use the ATL Controls Wizard to create an ActiveX control, you can choose to add a control based on a standard window class. In this case, the wizard adds a member variable of type **CContainedWindow** to your ActiveX control's class. **CContainedWindow::Create** then creates a window that superclasses the window class you specified. This window uses **CContainedWindow::WindowProc** to route its messages through a message map. If a message needs further processing, it is sent to the original window procedure of the window class.

Examining the Sample Code (AtlEdit.h)

The constructor for the **CAtlEdit** class sets the **m_bWindowOnly** member variable to **TRUE**. This ensures the control will never activate as a windowless control.

The **CContainedWindow** member variable, **m_EditCtrl**, is initialized by the **CAtlEdit** constructor. The **CContainedWindow** constructor takes three parameters: the name of the window class to be superclassed (in this case, "EDIT"); a pointer to the **CAtlEdit** class, which contains the message map; and the identifier of the message map that will process **m_EditCtrl**'s messages. By default, **m_EditCtrl** uses an alternate message map, declared with the **ALT_MSG_MAP** macro.

The default message map declares the names of the handler functions for **WM_CREATE** and **WM_CTLCOLOREDIT** messages sent to the **CAtlEdit** control. The **OnCreate** handler calls **CContainedWindow::Create** to create **m_EditCtrl**'s window. The **OnCtlColorEdit** handler specifies a new background and text color for **m_EditCtrl**.

The alternate message map declares a handler function for **WM_CHAR** messages sent to **m_EditCtrl**. This handler only accepts characters, not symbols or numbers, and then passes the **WM_CHAR** message to the original window procedure defined by the Windows Edit class.

Attributes

This sample uses the following attributes:

coclass, com_interface_entry, default, dual, emitidl, helpstring, implements_category, module, object, pointer_default, progid, registration_script, threading, uuid, version, vi_progid

Keywords

This sample uses the following keywords:

CComCoClass; CComControl; CComModule::GetClassObject; CComModule::GetLockCount; CComModule::Init; CComModule::RegisterServer; CComModule::Term; CComModule::UnregisterServer; CComObjectRoot; CContainedWindow::DefWindowProc; DisableThreadLibraryCalls; GetStockObject; GetWindowRect; IDataObjectImpl; IDispatchImpl; IOleControlImpl; IOleInPlaceActiveObjectImpl; IOleInPlaceObjectWindowlessImpl; IOleInPlaceObjectWindowlessImpl::SetObjectRects; IOleObjectImpl; IPersistStorageImpl; IPersistStreamInitImpl; IProvideClassInfo2Impl; IQuickActivateImpl; ISpecifyPropertyPagesImpl; IViewObjectExImpl; SetBkColor; SetTextColor; SetWindowPos

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Attributes Samples

ATL Controls Samples

The following topics are the abstracts for the ATL controls samples.

See Also

Other Resources

ATL Samples

ATLFire Sample: Demonstrates Building a Windowed Control

Download sample

The ATLFire sample is an ActiveX control that demonstrates how to build a windowed control using ATL by setting the **m_bWindowedOnly** flag to **TRUE**. It also shows what needs to be done to the MFC drawing code to convert it into straight Win32 code. The sample uses the ATL support for Win32 dialog boxes and property sheets and also demonstrates how to use a Win32 tab control in an ActiveX control. The sample also demonstrates several of the ATL macros.

The ATLFire attributes sample is the attributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file ATLFire.sln.
2. From the **Build** menu, click **Build**.
3. After the sample builds, open FireTabCtrl.htm in your Web browser and try out the various types of fire that the sample simulates.

You can test the ATLFire control in the ActiveX Control Test Container. For details on accessing Test Container and using it to test a control, see Testing Properties and Events with Test Container.

Classes and Keywords

This sample uses the following ATL classes:

CComObjectRootEx, CComCoClass, CComControl, IDispatchImpl, IProvideClassInfo2Impl, IPersistStreamInitImpl, IPersistStorageImpl, IPersistPropertyBagImpl, IPerPropertyBrowsingImpl, IQuickActivateImpl, IObjectSafetyImpl, IOleControlImpl, IOleObjectImpl, IOleInPlaceActiveObjectImpl, IViewObjectExImpl, IOleInPlaceObjectWindowlessImpl, IDataObjectImpl, ISupportErrorInfo, ISpecifyPropertyPagesImpl, IConnectionPointContainerImpl, IPropertyNotifySinkCP, CDialogImpl

This sample uses the following keywords:

_ASSERTE; _itot; _tcstol; ALT_MSG_MAP; Apply; ATLTRACE; BEGIN_COM_MAP; BEGIN_CONNECTION_POINT_MAP; BEGIN_MSG_MAP; BEGIN_OBJECT_MAP; BEGIN_PROPERTY_MAP; CComCoClass; CComModule::GetClassObject; CComModule::GetLockCount; CComModule::Init; CComModule::RegisterServer; CComModule::Term; CComModule::UnregisterServer; CComObjectRootEx; CDialogImpl; ClientToScreen; COM_INTERFACE_ENTRY; COM_INTERFACE_ENTRY_IMPL; COM_INTERFACE_ENTRY_IMPL_IID; COMMAND_HANDLER; COMMAND_ID_HANDLER; CONNECTION_POINT_ENTRY; CPropDlg::DoModal; CreateBitmap; CreatePalette; DECLARE_REGISTRY_RESOURCEID; DeleteDC; DeleteObject; DestroyMenu; DisableThreadLibraryCalls; EnableWindow; END_COM_MAP; END_CONNECTION_POINT_MAP; END_MSG_MAP; END_OBJECT_MAP; END_PROPERTY_MAP; EndDialog; FillRect; FireOnChanged; GetActiveWindow; GetClientRect; GetDlgItem; GetDlgItemText; GetModuleInstance; GetWindowRect; IConnectionPointContainerImpl; IDataObjectImpl; IDispatchImpl; InitFire; InlineIsEqualGUID; IObjectSafetyImpl; IOleControlImpl; IOleInPlaceActiveObjectImpl; IOleInPlaceObjectWindowlessImpl; IOleObjectImpl; IPersistPropertyBagImpl; IPersistStorageImpl; IPersistStreamInitImpl; IPropertyNotifySinkCP; IProvideClassInfo2Impl; IQuickActivateImpl; ISpecifyPropertyPagesImpl; ISupportErrorInfo; IViewObjectExImpl; memcpy; MESSAGE_HANDLER; MessageBox; NOTIFY_CODE_HANDLER; OBJECT_ENTRY; OnActivate; OnApply; OnCancel; OnCreate; OnDestroy; OnEraseBackground; OnInitDialog; OnOK; OnPaint; OnPaletteChanged; OnProperties; OnPropertyChanged; OnQueryNewPalette; OnRButtonDown; OnSelChanged; OnSelChanging; OnSize; OnTimer; PAINTSTRUCT ; PeekMessage; PROP_ENTRY; RealizePalette; SelectObject; SendMessage; SetTimer; SetWindowPos; TrackPopupMenuEx; UnregisterClass

See Also

Other Resources

CIRC Sample: Demonstrates Using Property Pages

Download sample

The CIRC sample shows how to use ATL to make a simple ActiveX control. The sample creates a control that draws a circle and demonstrates property pages, control painting, use of color and caption properties, Click and KeyPress events, and a Color property page.

The CIRC attributes sample is the attributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file circ.sln.
2. From the **Build** menu, click **Build Solution**.
3. Insert the control into a control container like ActiveX Control Test Container and test its properties and events. For details on accessing Test Container and using it to test a control, see Testing Properties and Events with Test Container.

Keywords

This sample uses the following keywords:

AtlCreateTargetDC; ATLTRACE; BEGIN_COM_MAP; BEGIN_CONNECTION_POINT_MAP; BEGIN_MSG_MAP; BEGIN_OBJECT_MAP; BEGIN_PROPERTY_MAP; CCircPropertyPageDlg ; CComCoClass; CComControl; CComControl::SetDirty; CComObjectRoot; CDialogImpl; CHAIN_MSG_MAP; COM_INTERFACE_ENTRY; COM_INTERFACE_ENTRY_IID; COM_INTERFACE_ENTRY_IMPL; COMMAND_ID_HANDLER; CONNECTION_POINT_ENTRY; CProxy_CircEvents; CreateEllipticRgn; CreatePen; CreateSolidBrush; CStockPropImpl; DECLARE_GET_CONTROLLING_UNKNOWN; DECLARE_REGISTRY_RESOURCEID; Ellipse; END_COM_MAP; END_CONNECTION_POINT_MAP; END_MSG_MAP; END_OBJECT_MAP; END_PROPERTY_MAP; GetClientRect; GetDialogBaseUnits; GetStockObject; GetTextMetrics; IConnectionPointContainerImpl; IConnectionPointImpl; IDataObjectImpl; IOleControlImpl; IOleInPlaceActiveObjectImpl; IOleInPlaceObjectWindowlessImpl; IOleLinkImpl; IOleObjectImpl; IOleObjectImpl::DoVerb; IPerPropertyBrowsingImpl; IPersistPropertyBagImpl; IPersistStorageImpl; IPersistStreamInitImpl; IPropertyNotifySinkCP; IPropertyPageImpl; IPropertyPageImpl::SetObjects; IProvideClassInfo2Impl; IQuickActivateImpl; IQuickActivateImpl::QuickActivate; IRunnableObjectImpl; ISpecifyPropertyPagesImpl; IViewObjectExImpl; MESSAGE_HANDLER; OBJECT_ENTRY; OleTranslateColor; PROP_ENTRY; SelectObject; SendDlgItemMessage; SetBkMode; SetTextColor; SetWindowRgn; USES_CONVERSION

See Also

Other Resources

ATL Samples

POLYGON Sample: The ATL Tutorial

Download sample

The POLYGON sample shows how to implement custom properties, events, a property page, and object safety for an ATL control. Go through the ATL Tutorial to create this control step by step.

The POLYGON attributes sample is the attributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file Polygon.sln.
2. From the **Build** menu, click **Build**.
3. After the sample builds, open PolyCtl.htm in your Web browser and try out the Polygon control. If you click inside the polygon, the number of sides increases. If you click outside the polygon, the number of sides decreases.

You can test the Polygon control in the ActiveX Control Test Container. For details on accessing Test Container and using it to test a control, see Testing Properties and Events with Test Container.

Keywords

This sample uses the following keywords:

ATLASSERT; ATLTRACE; auxWireSphere; BEGIN_COM_MAP; BEGIN_CONNECTION_POINT_MAP; BEGIN_MSG_MAP; BEGIN_OBJECT_MAP; BEGIN_PROPERTY_MAP; CComCoClass; CComControl; CComModule::GetClassObject; CComModule::GetLockCount; CComModule::Init; CComModule::RegisterServer; CComModule::Term; CComModule::UnregisterServer; CComObjectRoot; ChoosePixelFormat; CoTaskMemAlloc; CreateContext; CreateRGBPalette; DECLARE_GET_CONTROLLING_UNKNOWN; DECLARE_REGISTRY_RESOURCEID; DeleteObject; DescribePixelFormat; DisableThreadLibraryCalls; DrawText; END_CONNECTION_POINT_MAP; END_OBJECT_MAP; FireViewChange; GetClientRect; GetPixelFormat; glBegin; glClear; glClearColor; glClearDepth; glColor3f; glEnable; glEnd; glFinish; glLoadIdentity; glMatrixMode; glPopMatrix; glPushMatrix; glRotatef; glTranslatef; gluPerspective; glVertex3f; IDispatchImpl; IObjectSafetyImpl; IOleControlImpl; IOleInPlaceActiveObjectImpl; IOleInPlaceObjectWindowlessImpl ; IOleObjectImpl; IPersistStorageImpl; IPersistStreamInitImpl; IViewObjectExImpl; joyReleaseCapture; joySetThreshold; memcpy; OBJECT_ENTRY; PIXELFORMATDESCRIPTOR; RealizePalette; ReleaseCapture; SelectPalette; SetBkMode; SetCapture; SetTextColor; SwapBuffers; USES_CONVERSION; wglCreateContext; wglDeleteContext; wglGetCurrentDC; wglMakeCurrent

See Also

Other Resources

ATL Samples

SUBEDIT Sample: Superclasses a Standard Windows Control

Download sample

The SUBEDIT sample demonstrates how to create an ATL control that superclasses the standard Windows Edit control.

The SUBEDIT attributes sample is the attributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file SubEdit.sln.
2. From the **Build** menu, click **Build**.
3. After the sample builds, open AtlEdit.htm in your Web browser and try out the control.

You can test the control in the ActiveX Control Test Container by opening the ATLEdit Class. For details on accessing Test Container and using it to test a control, see Testing Properties and Events with Test Container.

Superclassing a Windows Control

ATL provides the ability to create a control that superclasses a standard Windows control. Superclassing allows you to create a window class that is based on an existing class but uses a different window procedure. You then create a window based on this new window class. When you superclass a control, messages are first processed by an ATL message map before being sent to the control's original window procedure. This allows you to modify the default behavior of standard Windows controls.

When you use the ATL Control Wizard to create an ActiveX control, you can choose to add a control based on a standard window class. In this case, the wizard adds a member variable of type **CContainedWindow** to your ActiveX control's class. **CContainedWindow::Create** then creates a window that superclasses the window class you specified. This window uses **CContainedWindow::WindowProc** to route its messages through a message map. If a message needs further processing, it is sent to the original window procedure of the window class.

Examining the SUBEDIT Sample Code (AtlEdit.h)

The constructor for the **CAtlEdit** class sets the **m_bWindowOnly** member variable to **TRUE**. This ensures the control will never activate as a windowless control.

The **CContainedWindow** member variable, **m_EditCtrl**, is initialized by the **CAtlEdit** constructor. The **CContainedWindow** constructor takes three parameters: the name of the window class to be superclassed (in this case, "EDIT"); a pointer to the **CAtlEdit** class, which contains the message map; and the identifier of the message map that will process **m_EditCtrl**'s messages. By default, **m_EditCtrl** uses an alternate message map, declared with the **ALT_MSG_MAP** macro.

The default message map declares the names of the handler functions for **WM_CREATE** and **WM_CTLCOLOREDIT** messages sent to the **CAtlEdit** control. The **OnCreate** handler calls **CContainedWindow::Create** to create **m_EditCtrl**'s window. The **OnCtlColorEdit** handler specifies a new background and text color for **m_EditCtrl**.

The alternate message map declares a handler function for **WM_CHAR** messages sent to **m_EditCtrl**. This handler only accepts characters, not symbols or numbers, and then passes the **WM_CHAR** message to the original window procedure defined by the Windows Edit class.

Keywords

This sample uses the following keywords:

CComCoClass; CComControl; CComModule::GetClassObject; CComModule::GetLockCount; CComModule::Init; CComModule::RegisterServer; CComModule::Term; CComModule::UnregisterServer; CComObjectRoot; CContainedWindow::DefWindowProc; DisableThreadLibraryCalls; GetStockObject; GetWindowRect; IDataObjectImpl;

IDispatchImpl; IOleControlImpl; IOleInPlaceActiveObjectImpl; IOleInPlaceObjectWindowlessImpl; IOleInPlaceObjectWindowlessImpl::SetObjectRects; IOleObjectImpl; IPersistStorageImpl; IPersistStreamInitImpl; IProvideClassInfo2Impl; IQuickActivateImpl; ISpecifyPropertyPagesImpl; IViewObjectExImpl; SetBkColor; SetTextColor; SetWindowPos

See Also

Other Resources

ATL Samples

ATL General Samples

The following topics are the abstracts for the ATL general samples.

See Also

Other Resources

ATL Samples

ATLCollections Sample: Demonstrates ICollectionOnSTLImpl, CComEnumOnSTL, and Custom Copy Policy Classes

Download sample

The ATLCollections sample demonstrates the use of **ICollectionOnSTLImpl** and **CComEnumOnSTL**, and the implementation of custom copy policy classes.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file ATLCollections.sln.
2. From the Build menu, click **Build Solution**.
3. From the **Debug** menu, click **Start Without Debugging**. This will test the collections using the supplied C++ client. This client outputs the content of the collections to a command window by looping through using the `Count` and `Item` properties and by enumerating the items using the `_NewEnum` property. The `Add`, `Remove`, and `Clear` methods of the `IItems` interface are also tested.

About the Sample

This project implements two COM collections based on data stored in STL containers:

- The simpler of the two classes, `CWords`, is a read-only **BSTR** collection based on data stored as **std::string**s in a **std::vector**. This class uses the custom copy policy class `VCUE::GenericCopy`, defined in VCUE_Copy.h and VCUE_CopyString.h, to convert the data from **std::string**s to the type appropriate for the collection or enumerator interface.
- The second class, `CItems`, is a read-write **VARIANT** collection based on data stored in a **std::map**. The **CComVariant** data in the map uses **CComBSTR**s (actually **CAdapt<CComBSTR>**s) as the keys. This class uses the custom copy policy class `VCUE::MapCopy`, defined in VCUE_Copy.h, to convert the stored data to **VARIANT**s that can be passed back to COM clients. `CItems` derives from class `VCUE::ICollectionOnSTLCopyImpl`, defined in VCUE_Collections.h. The latter class derives from **ICollectionOnSTLImpl** and overrides **get__NewEnum** so that each enumerator has its own copy of the collection data. `CItems` implements the `Add`, `Remove`, and `Clear` methods in the collection interface.

Keywords

This sample uses the following keywords:

_Copy, CComEnumOnSTL, ICollectionOnSTLImpl, CAdapt, IEnumVARIANT, std::map, std::string, std::vector

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Samples

ATLCON Sample: Demonstrates Creating a Simple Container

Download sample

The ATLCON sample demonstrates creating a simple container. The container implements the **IOleClientSite**, **IOleWindow**, and **IOleInPlaceSite** interfaces.

Container name and value columns are populated with the properties of the control which is inserted in the container.

The ATLCON attributes sample is the attributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file ATLCon.sln.
2. From the **Build** menu, click **Build Solution**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. Once the sample is running, use its **File** menu to insert a control into the container.

Keywords

This sample uses the following keywords:

ATLASSERT; ATLTRACENOTIMPL; BEGIN_COM_MAP; BEGIN_MSG_MAP; CComCoClass; CComModule::Unlock; CComObjectRoot; CComQIPtr; CExeModule::Init; CExeModule::RegisterClassObjects; CExeModule::RegisterServer; CExeModule::RevokeClassObjects; CExeModule::Unlock; CExeModule::UnregisterServer; CExeModule::UpdateRegistryFromResource; Close; CoCreateInstance; CoInitialize; COM_INTERFACE_ENTRY; COMMAND_ID_HANDLER; CoUninitialize; CWindowImpl; DECLARE_REGISTRY_RESOURCEID; DispatchMessage; DoVerb; END_COM_MAP; END_MSG_MAP; GetClientRect; GetCommandLine; GetCurrentThreadId; GetDesktopWindow; GetMessage; GetResourceInstance; IAtlCon; IOleClientSite; IOleInPlaceSite; LoadMenu; LPOLEINPLACEFRAMEINFO; LPRECT; MESSAGE_HANDLER; MessageBox; OBJECT_ENTRY; PostQuitMessage; PostThreadMessage; SetClientSite; SetObjectRects; ShowWindow; TranslateMessage; ZeroMemory

See Also

Other Resources

ATL Samples

ATLEventHandling Sample: Handles Word Events Using IDispEventImpl and IDispEventSimpleImpl

Download sample

The ATLEventHandling sample demonstrates the use of **IDispEventImpl** and **IDispEventSimpleImpl** to handle events fired by Microsoft Word.

The sample consists of an ATL project that includes two COM classes: `CSimple` and `CNotSoSimple`. These classes represent COM objects that display a message box in response to document change events fired by Microsoft Word. `CSimple` shows how to handle events using ATL's **IDispEventSimpleImpl** class template, and `CNotSoSimple` shows how to handle events using ATL's **IDispEventImpl** class template. The code in the two classes is identical except for the differences caused by the use of the different ATL templates.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Note
This sample requires that Microsoft Office 2000 be installed. If you want to use this sample with other versions of Microsoft Office, you must specify the path to the mso dll file referenced in the StdAfx.h file under the ATLEventHandling directory. For example, Office 2000 makes use of mso9.dll and Office XP makes use of mso.dll.

To build and run this sample

1. Open the solution file ATLEventHandling.sln.
2. If you are using Excel 2000, modify the following line in StdAfx.h:

   ```
   #define OFFICE_VER OFFICEXP
   ```

   to instead be:

   ```
   #define OFFICE_VER OFFICE2000
   ```

3. From the **Build** menu, click **Build Solution**.
4. Run MFCClient.exe.

You'll see a dialog box with a drop-down list allowing you to run the test using **IDispEventImpl** or **IDispEventSimpleImpl**. Select one of these items, and then click the **Start** button. If Microsoft Word is installed on your computer, a new instance will be loaded and the test component will start to receive events sent by Word whenever the active document changes.

You can trigger events by creating new documents, loading documents, or switching to a different document using Word's **Window** menu. Each time the document changes, a message box will be displayed indicating which component is receiving the events and the name of the newly active document. You can disconnect the event handler from the instance of Word by using the **Stop** button on the **MFCClient** dialog box, by changing the selection of the drop-down list, or by closing Word.

Close the sample application using the **OK** or **Cancel** command button or the dialog box's **Close** button.

Keywords

This sample uses the following keywords:

_ATL_FUNC_INFO, BEGIN_SINK_MAP, END_SINK_MAP, SINK_ENTRY_EX, SINK_ENTRY_INFO, IDispEventImpl,

IDispEventSimpleImpl, IDispEventSimpleImpl::DispEventAdvise, IDispEventSimpleImpl::DispEventUnadvise

See Also

Other Resources

ATL Samples

ATLPages Sample: Implements a Property Page Using IPropertyPageImpl

Download sample

The ATLPages sample demonstrates the implementation of a property page using **IPropertyPageImpl**.

This sample consists of:

- A property page class, `CDocumentProperties`, that uses the **EnvDTE::Document** interface to display (and allow changes to) the properties of a text document.
- A helper component, `CHelper`, that exposes a simplified wrapper to the **OleCreatePropertyFrame** API to scripting languages.
- A simple test macro, Test in the ATLPages.vsmacros project, that uses the helper to display the property page for the active document within the Visual C++ editor.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To run the sample, you need to build the solution and then run the test macro.

To build this sample

1. Open the solution file ATLPages7.sln.
2. From the **Build** menu, click **Build Solution**.

To run the macro

1. Open a text document in the editor. For example, open one of the source files for the ATLPages sample.
2. From the **Tools** menu, point to **Macros** and then click **Macro Explorer**. This will open the Macro Explorer window.
3. From the **Tools** menu, point to **Macros** and then click **Load Macro Project**. This will open the **Add Macro Project** dialog box.
4. Browse to the location of the ATLPages.vsmacros file (it's in the same folder as the ATLPages sample) and click the **Open** button. This action will load the macros file and it will appear in the Macro Explorer.
5. Expand the ATLPages node and double-click the Test macro.

   If no text document is open when you run the macro, it does nothing. If a document is open, a property page for that document will be displayed indicating the name of the file and its read-only status. You can change either of these items. Your changes will be applied when you click the **Apply** or **OK** button. Changing the name of the file will cause the file to be saved to disk under the new name. Changing the read-only status of the file affects whether the file can be changed within the Visual C++ editor (it doesn't affect the read-only attribute of the file on disk).

Note
Changing the read-only property of files under source code control is not possible using this property page.

Keywords

This sample uses the following keywords:

IPropertyPage, IPropertyPageImpl, OleCreatePropertyFrame, IPropertyPage::SetObjects, IPropertyPage::Activate,

IPropertyPage::Apply, EnvDTE

See Also

Other Resources

ATL Samples

ATLSafeArray Sample: Demonstrates CComSafeArray and Passing SAFEARRAYs to Script

Download sample

The ATLSafeArray sample shows how to create and maintain SAFEARRAYs using **CComSafeArray**. It also shows how to pass SAFEARRAYs from a component to script: when the user clicks the buttons on the HTML page, various methods in the control are executed.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file ATLSafeArray.sln.
2. From the **Build** menu, click **Build Solution**.
3. When the build finishes, open ATLSafeArray.htm in your Web browser.
4. Fill in the text box and click the appropriate buttons to Add, Change, or Remove an item from the list box.

Classes and Keywords

This sample demonstrates the following classes:

CComSafeArray; CComVariant

This sample demonstrates the following keywords:

CComSafeArray::Add; CComSafeArray::SetAt; CComSafeArray::operator=; CComSafeArray::operator[]; CComSafeArray::GetCount; CComSafeArray::Resize; CComVariant::Detach

See Also

Other Resources

ATL Samples

AutoThread Sample: Demonstrates the Use of CAtlAutoThreadModule

Download sample

The AutoThread sample demonstrates using **CAtlAutoThreadModule**. The server is implemented in the Server.exe file. The module of the EXE is derived from **CAtlAutoThreadModule** instead of **CAtlModule**.

The AutoThread attributes sample is the attributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file AutoThread.sln.
2. From the **Build** menu, click **Build Solution**. This will build and register the client and the server.
3. Start two instances of the ActiveX Control Test Container. See Testing Properties and Events with Test Container for information on how to access the test container.
4. Insert one of the client controls (AutoCtl Class) into each of the test containers.
5. Click one of the controls and notice that it takes one second for the server to return.
6. Position and resize the test containers so that both of them are visible at the same time.
7. Click one of the controls and then quickly click the other control. Notice that they finish waiting at approximately the same time. (If **CAtlAutoThreadModule** were not used, the first control would finish after one second, but the second control would not finish until a full second after the first control finished. The second call to sleep would not occur until the first had finished.) You can use the **Delay(PropGet)** and **Delay(PropPut)** methods to adjust the number of milliseconds the server sleeps for. If set properly, the second call to sleep may return before the first call to sleep.

How the Sample Works

The server interface has a single method: **Sleep**. This method puts the server thread to sleep for a given amount of time. The client portion of the sample is an ActiveX control that invokes the server's sleep method when the user clicks the control. The client also has a property named **Delay** that represents how long the server thread will sleep. The control displays the text "Ready" when it is waiting for a user click. The text "Waiting" is displayed when the control is waiting for the server to finish sleeping.

Keywords

This sample uses the following keywords:

AtlGetObjectSourceInterface; BEGIN_SINK_MAP; CoCreateInstance; DECLARE_CLASSFACTORY_SINGLETON; IDispEventSimpleImpl::DispEventAdvise; END_SINK_MAP; IConnectionPointContainerImpl; IConnectionPointImpl; IDispEventImpl; OLE2CT; SINK_ENTRY_EX; SysAllocString; USES_CONVERSION; VARIANT; VariantClear; VariantCopy

See Also

Other Resources

ATL Samples

BEEPER Sample: Demonstrates a Tearoff Interface

Download sample

The BEEPER sample illustrates the implementation of a tearoff interface — a collection/enumeration of **BSTR** values. It can be built as a DLL and as an EXE.

The BEEPER attributes sample is the attributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file beeper.sln.
2. From the **Build** menu, click **Build Solution**.
3. After the sample builds, open beeper.htm (an HTML file that uses VBScript) in your Web browser and follow its instructions.

Keywords

This sample uses the following keywords:

BEGIN_COM_MAP; CComCoClass; CComModule::Unlock; CComObjectRoot; COM_INTERFACE_ENTRY; COM_INTERFACE_ENTRY_TEAR_OFF; DECLARE_GET_CONTROLLING_UNKNOWN; DECLARE_NOT_AGGREGATABLE; DECLARE_REGISTRY; END_COM_MAP; FinalConstruct; IDispatchImpl; ISupportErrorInfo; PostThreadMessage; return Error; SysAllocString; THREADFLAGS_BOTH

See Also

Other Resources

ATL Attributes Samples

CIRCCOLL Sample: Demonstrates a Collection/Enumeration

Download sample

The CIRCOLL sample shows how to implement a collection/enumeration of objects using ATL.

The CIRCCOLL attributes sample is the attributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file circoll.sln.
2. From the **Build** menu, click **Build Solution**.
3. From the **Debug** menu, click **Start**.
4. The Visual Basic form, **Form1**, will open. Click the button to activate the application.

How the Sample Works

Three object classes are implemented: the collection creator, the collection, and the object inside the collection. Only the collection creator has a **coclass** associated with it (see Circcoll.idl for the **coclass** declaration). The collection creator has a method to return a collection of circle objects. The collection object implements the **Item**, **Count**, and **_NewItem** methods, so that the object can be manipulated from Visual Basic using both the **For...Next** and **For Each...** syntax.

Keywords

This sample uses the following keywords:

AddRef; ATLASSERT; ATLTRACE; BEGIN_COM_MAP; BEGIN_OBJECT_MAP; CComCoClass; CComObject::CreateInstance; CComObjectRoot; CComVariant; COM_INTERFACE_ENTRY; DECLARE_NOT_AGGREGATABLE; DECLARE_REGISTRY; END_COM_MAP; END_OBJECT_MAP; glBegin; glEnd; glNormal3d; glPolygonMode; glVertex2d; glVertex3d; IConnectionPointContainerImpl::FindConnectionPoint; IConnectionPointImpl::Advise; IDispatchImpl; ISupportErrorInfo; OBJECT_ENTRY; QueryInterface; Release; USES_CONVERSION; VariantCopy; VariantInit

See Also

Other Resources

ATL Samples

CustomString Sample: Demonstrates Custom Memory Allocators for CStringT

Download sample

The CustomString sample shows how to use a custom memory allocator for **CStringT** to improve performance in a multithreaded application. The sample application replaces carriage return/line feed pairs in a set of text files with a single carriage return, processing multiple files simultaneously on different threads.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file CustomString.sln.
2. From the **Build** menu, click **Build Solution**.
3. To run the sample from a command prompt, type CustomString *.txt. The sample will find all files in the current directory with ".txt" as the file extension and will create a copy of each file with ".utxt" as the file extension. The new files will have all CR/LF pairs replaced with a single CR.
4. To run the sample from within Visual Studio, right-click the solution and click **Properties** on the shortcut menu. Under **Configuration Properties**. select **Debugging**, and set the command-line argument property to "*.txt" for the correct configuration.

Classes and Keywords

This sample uses the following classes:

IAtlStringMgr; CWin32Heap; CAtlStringMgr; CStringT

This sample uses the following keywords:

CBitmap::LoadBitmap; CEdit::Clear; CEdit::GetLineCount; CEdit::LineIndex; CEdit::LineLength; CEdit::ReplaceSel; CEdit::SetSel; CFileDialog::DoModal; CFileDialog::GetPathName; CGdiObject::DeleteObject; CSpinButtonCtrl::GetBuddy; CString::GetBufferSetLength; CString::GetLength; CString::Left; CString::LoadString; CString::ReleaseBuffer; CWinApp::LoadIcon; CWnd::GetClientRect; CWnd::GetWindowRect; CWnd::SetWindowPos; CWnd::SetWindowText; DeleteItem; CWnd::DestroyWindow; CWnd::EnableWindow; GetCursorPos; GetDlgItem; GetParent; GetWindowLong; GetWindowRect; InvalidateRect; LoadIcon; MAKELONG; MessageBeep; ReleaseCapture; ScreenToClient; SetCapture; SetWindowLong; SetWindowPos; UpdateWindow; WindowProc; mbstowcs; rand; srand; time; wsprintf

See Also

Other Resources

ATL Samples

DispSink Sample: Handles Events Fired from a Singleton COM Server Through a Dispatch Interface

Download sample

The DispSink sample demonstrates a singleton server object (an object that can have only one instance) that has its own dual interface and a dispatch interface used for firing off events.

The DispSink attributes sample is the attributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file DispSink.sln.
2. From the **Build** menu, click **Build Solution**.
3. Open two or more instances of the ActiveX Control Test Container and insert the client control, DispCtl, into each instance. See Testing Properties and Events with Test Container for information on how to access the test container.
4. Invoke the **Connect** method on all of the controls.
5. Invoke the **Send** method on one of the controls. Change the **Parameter Type** field of the **Invoke Methods** dialog box to VT_BSTR and then type any string into the **Parameter Value** box. Click the **Invoke** button. The string will be displayed in the center of all connected controls.
6. Invoke the **Disconnect** method on all controls prior to deleting them.

How the Sample Works

The server is a singleton object that has its own dual interface as well as a dispatch interface used for firing off events. The server object receives data through its dual interface **Send** method and transmits it to all connected components through the **Transfer** event on its dispatch interface.

The client is an ActiveX control that contains a server object. The control responds to the **Transfer** event fired by the server object. It has a dual interface that has **Connect**, **Send**, and **Disconnect** methods. If the **Transfer** event is fired with a variant containing a BSTR, the string is displayed in the center of the control.

Keywords

This sample uses the following keywords:

#import; CComAutoThreadModule; CComSimpleThreadAllocator; CoCreateInstance; DECLARE_CLASSFACTORY_AUTO_THREAD(); FireViewChange

See Also

Other Resources

ATL Samples

LABRADOR Sample: Implements a Server with No User Interface

Download sample

The LABRADOR sample shows how to use ATL to implement an EXE server without any user interface. The server allows creation of an object that supports two custom interfaces, defined in Labrador.idl.

The LABRADOR attributes sample is the attributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

This sample uses three components: the server, the marshaling DLL, and the driver.

To build and register the components

1. Open the solution file Labrador.sln
2. From the **Build** menu, click **Build Solution**.

   The server, the marshaling DLL, and the driver will be built and registered.

To run the driver

- From the **Debug** menu, click **Start Without Debugging**.

  The driver will create an object, make a few calls into it, and then release it.

Keywords

This sample uses the following keywords:

_CrtDumpMemoryLeaks; _tcsicmp; _tcstok; _tprintf; _vstprintf; ATLASSERT; BEGIN_COM_MAP; BEGIN_OBJECT_MAP; CComModule::Init; CComModule::RegisterClassObjects; CComModule::RevokeClassObjects; CComModule::Unlock; CComModule::UnregisterServer; CComObjectRoot; CoCreateInstance; COM_INTERFACE_ENTRY; CoUninitialize; DECLARE_NOT_AGGREGATABLE; DECLARE_REGISTRY; DispatchMessage; END_OBJECT_MAP; GetCurrentThreadId; GetMessage; OBJECT_ENTRY; OutputDebugString; PostThreadMessage; Trace; va_end; va_list; va_start; wcscpy

See Also

Other Resources

ATL Samples

OLE DB Templates Samples

The following topics are the abstracts for the ATL OLE DB Templates samples.

In This Section

AdvancedPV Sample: Demonstrates Advanced Provider Techniques

CatDB Sample: Data Source Schema Browser

DBViewer Sample: Database Browser

DynamicConsumer Sample: Uses Dynamic Accessor and Schema Rowset Classes to Read Metadata from a Database

MultiRead Sample: Reads Database Table Using Multiple Threads

UpdatePV Sample: Implements an Updatable OLE DB Provider

See Also

Other Resources

ATL Samples

AdvancedPV Sample: Demonstrates Advanced Provider Techniques

Download sample

The AdvancedPV sample is very similar to UpdatePV, but it demonstrates some advanced techniques.

Normally providers written using OLE DB Templates use **CAtlArray** for data storage. The OLE DB Provider Templates call a user-provided `Execute` method to populate the array (for example, to load all of the rows from the data file into the array). Another user-provided method, `FlushData`, is used to save the contents of the array (for example, to write the array contents back into the data file).The problem with this approach is that in `Execute` you have to load all of the rows in the rowset, and in `FlushData` you have to save all of the rows at the same time. If there is a large number of rows in the rowset, all of the data needs to be stored in memory (in the **CAtlArray** object).

AdvancedPV demonstrates how to use a special array class in place of the default **CAtlArray** array to make the provider load and save rows as necessary. The rows will be loaded from the data file only when they are actually requested (through a specially implemented `operator[]`) and the changes will be written back to the file as soon as the array contents change.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file AdvancedPV.sln.
2. From the **Build** menu, click **Build**.
3. Create a Win32 Console application with the Win32 Project wizard. Give it ATL support.
4. Add an OLE DB consumer to the project (from **Add Class**, select **ATL OLE DB Consumer**).
5. In the ATL OLE DB Consumer wizard, click the **Data Source** button and in **Data Link Properties**, select **AdvancedProv Provider**. (The **AdvancedProv** provider should be registered automatically when you build AdvancedPV, but if you don't see it listed here, run regsvr32.exe on AdvancedPV.dll.)
6. Click **Next** to go to the **Connection** tab, then under **Enter the initial catalog to use**, specify the initial catalog to use (the path to DataFile.dat).
7. Under **Select Database Object**, open **Tables**; there is only one item (the path to DataFile.dat). Select it and click **OK**. When you return to the ATL OLE DB Consumer wizard, select **Table**, rename the class to something shorter (if necessary) such as `CMyCons`, and click **Finish**. Build the consumer project.
8. Add the following to your project's main code:

```
                #include "MyCons.h"

int main( int argc, char* argv[] )
{
   // Add this code
   HRESULT hr = CoInitialize(NULL);
   CMyCons rs;
   hr = rs.OpenAll();
   ATLASSERT( SUCCEEDED(hr));

   hr = rs.MoveFirst();
```

```
    while( SUCCEEDED(hr) && hr != DB_S_ENDOFROWSET )
    {
       printf( "%d %s %s %s %s\n", rs.m_Fixed, rs.m_Command, rs.m_Text,
       rs.m_Command2, rs.m_Text2 );
       hr = rs.MoveNext();
    }

    rs.CloseAll();
    CoUninitialize();
    return 0;
 }
```

9. Put a breakpoint on the **CoUninitialize** function; this will make the console stay open so you can view the results. Run the application by clicking the **Start** button (or click **Start Without Debugging** from the **Debug** menu). You should see five columns of text printed out (one index and four text columns).

Keywords

This sample uses the following interfaces:

IRowsetLocateImpl, IRowsetScroll, IRowsetScrollImpl, IRowsetUpdateImpl, IConnectionPointContainerImpl, IRowsetNotifyCP, IDBCreateSessionImpl, IDBInitializeImpl, IDBPropertiesImpl, IPersistImpl, IInternalConnectionImpl, IGetDataSourceImpl, IOpenRowsetImpl, ISessionPropertiesImpl, IObjectWithSiteSessionImpl, IDBSchemaRowsetImpl, IDBCreateCommandImpl, IAccessorImpl, ICommandTextImpl, ICommandPropertiesImpl, IObjectWithSiteImpl, IConvertTypeImpl, IColumnsInfoImpl, IInternalCommandConnectionImpl

The sample demonstrates the following classes:

CSchemaRowsetImpl, CComObjectRootEx, CComObjectRootEx, CRowsetImpl, CFileArray, CSimpleRow

The sample demonstrates the following macros:

COM_INTERFACE_ENTRY, PROPERTY_INFO_ENTRY

See Also

Other Resources

ATL Samples

CatDB Sample: Data Source Schema Browser

Download sample

CatDB is ported from the MFC ODBC Catalog application. The application displays the schema information, such as tables and columns, of OLE DB providers. It is easy for you to track the differences between this sample and the MFC ODBC Catalog sample, because all of the changes from the MFC sample are surgical. This sample makes use of the CEnumerator, CDataSource, CSession, and schema rowset classes.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

You can use this sample with the ODBC provider and with the Microsoft Access 97 and Microsoft SQL Server 6.5 databases (or later versions). To run this sample, you must have the MDAC SDK installed.

To build and run this sample

1. Open the solution file CatDB.sln.
2. From the **Build** menu, click **Build**.
3. From the **Debug** menu, click **Start Without Debugging**.

   A dialog box will appear, titled "CatDB - [No data source selected]."
4. Click **Open** from the **File** menu. The **Data Link Properties** dialog box will appear. On the **Provider** tab, select Microsoft OLE DB Provider for SQL Server (or Microsoft Jet 4.0 OLE DB Provider). On the **Connection** tab, select the Northwind database.

   The table information will appear in the dialog box's window. The dialog box title will change to the database name.

How the Sample Works

The code to enumerate the providers, connect to a data source, and create a session is in the `CCatDBDoc::OnOpenDocument` function. The code to open the schema rowsets is in `CCatDBDoc::FetchTableInfo` and `CCatDBDoc::FetchColumnInfo`. Both `FetchTableInfo` and `FetchColumnInfo` use the OLE DB Templates schema rowset classes, CTables and CColumns. The code to display the schema information is in `CCatDBView::OnUpdate`. The `OnUpdate` function moves through the records in the database and accesses the **CTables** and **CColumns** classes for data.

This sample demonstrates how to enumerate the OLE DB providers. `CEnumSourcesDlg` provides support to recursively display and connect to the data sources. The dialog box uses the OLE DB Templates **CEnumerator** class. To connect to the data source, `CEnumSourcesDlg` returns an **LPMONIKER** value for the selected data source. **CDataSource::Open** then takes this **LPMONIKER** value and performs the connection.

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

Keywords

The sample demonstrates the following classes:

CColumns, CDataSource, CEnumerator, CSession, CTables

The sample demonstrates the following functions:

CDataSource::Open

See Also

Other Resources

ATL Samples

DBViewer Sample: Database Browser

Download sample

The DBViewer sample is an enhancement of the MFC DAOVIEW sample. It demonstrates a mid-level application that relies on the OLE DB Templates **CManualAccessor** class. This sample illustrates how to take full control of the bindings and use them for your applications.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

You can use DBViewer with the ODBC provider, and with the Microsoft Access 97 and Microsoft SQL Server 6.5 databases (or later versions). However, DBViewer cannot handle SQL Server stored procedures with integer parameters, or with names consisting of more than one word, for example, `My stored procedure`. You will get errors in these cases.

To build and run this sample

1. Open the solution file DBViewer.sln.
2. From the **Build** menu, click **Build**.
3. From the **Debug** menu, click **Start Without Debugging**.

   A DBViewer dialog box will appear, with two panes.
4. Click **Open** from the **File** menu. The **Data Link Properties** dialog box will appear. On the **Provider** tab, select Microsoft OLE DB Provider for SQL Server (or Microsoft Jet 4.0 OLE DB Provider). On the **Connection** tab, select a database such as Northwind.

   The specified database's tables and stored procedures will appear in the tree view pane on the left.
5. Once you have connected to a data source, you can manipulate the data, call the stored procedures, and view the schema information. The tree view on the left side of the application window displays the tables and the stored procedures. To view or modify data, right-click a table or a stored procedure, and a shortcut menu will appear with additional operations.

How the Sample Works

Once DBViewer connects to a data source via the enumerator support, you can manipulate the data, call the stored procedures, and view the schema information. The tree view on the left side of the application window displays the tables and the stored procedures. To view or modify data, right-click a table or a stored procedure, and a shortcut menu will appear with additional operations.

This sample contains code to support stored procedures. However, you will find that not all stored procedures will run under this sample. This is because the underlying ODBC drivers do not provide support for some OLE DB methods. DBViewer does support multiple-result sets. If you click the list view where the data is located, you may see the **multiple result sets** button highlighted on the toolbar. You can click it and obtain the next result set.

DBViewer demonstrates how to use the error information in your application. The `CErrorsDialog` class handles the **IErrorRecords** interface and displays the error information returned from a particular call.

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

Keywords

The sample demonstrates the following classes:

CManualAccessor, CErrorsDialog

The sample demonstrates the following interfaces:

IErrorRecords

See Also

Other Resources

ATL Samples

DynamicConsumer Sample: Uses Dynamic Accessor and Schema Rowset Classes to Read Metadata from a Database

Download sample

The DynamicConsumer sample is a simple console application that allows you to connect to any data source and select a table or stored procedure. If you select a stored procedure, the sample application will prompt you to enter the values for parameters. It then executes a command to display the data in the result rowset.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file DynamicConsumer.sln.
2. From the **Build** menu, click **Build**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. A console window will appear, prompting you to specify which kind of accessor to use. Enter a number to specify an accessor type. Other specification menus will follow, depending on your choice.
5. When you are finished with the menu selections, the **Data Link Properties** dialog box will appear. On the **Provider** tab, select Microsoft OLE DB Provider for SQL Server. On the **Connection** tab, select a database such as Northwind.
6. When you have selected a data source, the console prompts you to specify the table from which you want to print out data. When you do so, the table's data displays in the console.

How the Sample Works

The sample uses schema rowset classes to read metadata from the data source. Metadata is any information about a data source other than the data stored in the database, such as column names, column data types, table names, or stored procedures. DynamicConsumer uses the schema rowset classes **CTables**, **CProcedures**, and **CProcedureParameters** to select the table or procedure and to build a SQL command string dynamically. The sample also uses the following accessor classes:

- **CDynamicAccessor** (demonstrates the various ways to handle BLOB data)
- **CDynamicStringAccessor**
- **CXMLAccessor**
- **CDynamicParameterAccessor**

Keywords

The sample demonstrates the following classes:

CDynamicAccessor, CDynamicParameterAccessor, CDynamicStringAccessor, CProcedureParameters, CProcedures, CTables, CXMLAccessor

See Also

Other Resources

ATL Samples

MultiRead Sample: Reads Database Table Using Multiple Threads

Download sample

The MultiRead sample demonstrates how to use the OLE DB Consumer Templates classes to read through a table in a database using multiple threads.

The MultiRead attributes sample is the attributed version of this sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file MultiRead.sln.
2. From the **Build** menu, click **Build**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. A Multi-Threaded Read dialog box will appear, prompting you to specify the number of threads to use to read through the table. Click **Run**.
5. The result will appear as text in the Multi-Threaded Read dialog box, for example, `15 records in 7 ms`.

How the Sample Works

The sample contains the `CMultiDlg` class, which is used to show a dialog box. With this dialog box, the user enters the number of threads to use to read through the table. When the user clicks the **Run** button, `ReadRecords` in DBRead.h is called to open the database, the session, and the table, and to create the necessary number of threads. As it opens the table, the function sets the **DBPROP_CANHOLDROWS** property to true so that the provider allows the user to retrieve new rows without releasing the previously retrieved rows. This capability is required, because multiple threads retrieve new rows as other threads are still processing their current threads.

The sample also shows how to extend the standard **CRowset** class by creating a new `CMyRowset` class, which derives from **CRowset**, and adds the `MoveAndProcess` member function. The start routine of each thread is the `ReadTable` function, and the table class is passed to this function. The function calls the routine `MoveAndProcess` to read through each record. Note that the accessor class `CProduct` is defined so that the data will not be retrieved automatically on the `MoveNext` call. This avoids buffer conflicts with the other threads, and there is no need to protect `MoveNext` with a critical section. The `MoveAndProcess` function calls `MoveNext`, and then calls `GetDataHere` to place the data directly into a local variable of that function. `ProcessRecord` is called for each record retrieved, and the function simply traces out the values of the record by default.

Each thread counts the number of records it reads, and those are traced out at the end, along with a total and time taken, which is displayed on the dialog box.

Note
The MultiRead sample reads the MultiRead.mdb database file. The sample code assumes that this file is located in the current directory.

Keywords

The sample demonstrates the following classes:

CAccessor, CDataSource, CDBPropSet, CRowset, CSession, CTable

The sample demonstrates the following macros:

BEGIN_ACCESSOR_MAP, BEGIN_ACCESSOR, COLUMN_ENTRY, END_ACCESSOR, END_ACCESSOR_MAP, DEFINE_COMMAND

The sample demonstrates the following functions:

CreateThread, GetCurrentThreadId, GetExitCodeThread, WaitForMultipleObjects

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

ATL Samples

UpdatePV Sample: Implements an Updatable OLE DB Provider

Download sample

The UpdatePV sample is an OLE DB Provider Templates sample that demonstrates how to implement an updatable (read/write) provider. Specifically, it demonstrates how to perform immediate and deferred inserts/updates/deletes. UpdatePV also demonstrates how to use schema rowsets (which make it easier for some wizards to interact with a provider). UpdatePV also demonstrates the **IRowsetLocateImpl** class, as does MyProv.

See the AdvancedPV sample for illustrations of additional techniques on loading and saving data with a provider.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To demonstrate the sample's intended features, build the sample, create a consumer project with an accessor for the provider, and create a console application to access and output the data.

To build and run this sample

1. Open the solution file UpdatePV.sln.
2. From the **Build** menu, click **Build Solution**.
3. Create a consumer project with the ATL Project wizard (make it an attributed .dll).
4. Add an OLE DB consumer to the consumer project (from **Add Class**, select **ATL OLE DB Consumer**).
5. In the ATL OLE DB Consumer wizard, click the **Data Source** button and in **Data Link Properties**, select **UpProv OLE DB Provider**. (The **UpProv** provider should be registered automatically when you build UpdatePV, but if you don't see it listed here, run regsvr32.exe on UpdatePV.dll.)
6. Click **Next** to go to the **Connection** tab, then under **Enter the initial catalog to use**, make sure the path name to MyData.txt is correct.
7. Click **Ok**. The **Select Database Object** dialog will appear, then open **Tables**; there is only one item: the path name for MyData.txt. Select it and click **OK**. You return to the ATL OLE DB Consumer wizard.
8. In the ATL OLE DB Consumer wizard, select **Table**, rename the class to something shorter (if necessary) such as `CMyCons`, and click **Finish**.
9. Dismiss the security warning dialog that pops up by pressing **OK**.
10. Remove this line from CMyCons.h:

    ```
    #error Security Issue: The connection string may contain a password
    ```

    This line will prevent compilation, but is inserted to alert you to the fact that (in real-world scenarios) sensitive information should be protected.
11. Build the consumer project by right-clicking on the project's node in the **Solution Explorer** and clicking **Build**.
12. Create a new console application. In the .cpp file, include the consumer header and modify the code as follows:

    ```
    #include "stdafx.h"
    #include <atldbcli.h>
    #include "..\consumer\cmycons.h"

    int _tmain(int argc, _TCHAR* argv[])
    ```

```
{
   HRESULT hr = CoInitialize(NULL);

   CMyCons c;

   hr = c.OpenAll();
   ATLASSERT( SUCCEEDED( hr ) );

   hr = c.MoveFirst();
   while( SUCCEEDED(hr) && hr != DB_S_ENDOFROWSET )
   {
      printf( "%d %s %s %s %s\n", c.m_Fixed, c.m_Command, c.m_Text,
         c.m_Command2, c.m_Text2 );
      hr = c.MoveNext();
   }

   c.CloseAll();
   CoUninitialize();
   return 0;
}
```

13. Place a breakpoint on the **CoUninitialize** function; this will make the console stay open so you can view the results. Run the console application from the development environment by clicking the **Start** button. You should see five columns of text printed out (one index column and four text columns).

How the Sample Works

UpdatePV is built on top of the C run-time file I/O functions. This represents a data store. Specifically, the sample takes a text file consisting of a pair of data elements and turns it into a rowset. The sample comes with a text file, MyData.txt, that contains pairs of data elements. However, you can run it against any text file (it will just parse everything into two-word tuples).

UpdatePV performs its read operations with **RUpdateRowset::Execute** (rowset.h). The write operations are handled with **RUpdateRowset::FlushData** (rowset.h). These are functions called by the OLE DB Provider Templates as part of the normal provider operation. The sample uses the OLE DB Provider Templates **IRowsetChangeImpl** and **IRowsetUpdateImpl** classes. The **IRowsetChangeImpl** class provides support for immediate inserts/updates/deletes. The **IRowsetUpdateImpl** class support deferred inserts/updates/deletes. The **IRowsetUpdateImpl** class inherits from **IRowsetChangeImpl**. For more information on getting/setting data, read Creating an Updatable Provider in the Visual C++ documentation, and Updating Data in Rowsets in the *OLE DB Programmer's Reference* in the Platform SDK documentation.

UpdatePV also provides support for schema rowsets. These schema rowsets allow consumers to find out information about a provider without opening a rowset or executing a command. The Visual C++ wizards use schema rowsets to generate client side accessors. The primary functions are **CUpdateSessionTRSchemaRowset::Execute**, **CUpdateSessionColSchemaRowset::Execute**, and **CUpdateSessionPTSchemaRowset::Execute**. All three functions return information on what tables the provider supports, columns on the tables, and the data types on the tables. For further information on schema rowsets, see the **IDBSchemaRowset** interface in the *OLE DB Programmer's Reference*.

Keywords

The sample demonstrates the following interfaces:

IRowsetChange, IRowsetUpdateImpl

The sample demonstrates the following properties:

DBPROP_IRowsetChange, DBPROP_IRowsetUpdate

See Also

Other Resources

ATL Samples

ATL Server Samples

The following table is an alphabetical list of the ATL Server samples in Visual C++.

ATLRT Sample: Demonstrates Custom SRF Tags

Demonstrates extending ATL Server's stencil-parsing code to support custom tags. The custom tags supplied in this sample provide support for variables and generic database access from within a SRF file.

BlobCache Sample: Generic Memory Caching

Demonstrates the use of the **CBlobCache** class from within an ATL Server application.

CLStencil Sample: Runs Request Handler DLLs or SRF Files from the Command Line

Demonstrates how to provide custom implementations of IHttpServerContext Interface and IIsapiExtension to create a command line tool for generating responses from SRF files and Web application DLLs without requiring IIS.

DataSetConsumer Sample: Consumes Multiple Table Datasets

Demonstrates how to consume multiple table DataSets in a SPROXY.EXE: XML Web Service Proxy Generator-generated SOAP client.

Hotswap Sample: Replacing ATL Server Application DLLs at Run Time

Demonstrates how to replace ATL Server Web application DLLs at run time without losing requests.

HttpClient Sample: HTTP Client Services Demonstration

Demonstrates making HTTP Web requests from within an MFC application using ATL Server's HTTP client support.

HttpPing Sample: HTTP Client Tool

Demonstrates making HTTP Web requests from within a console application using ATL Server's HTTP client support.

Input Sample: Demonstrates User Input Validation On Client And Server

Demonstrates validation of user input on both client and server.

ISAPIFilter Sample: Maps URLs to Parameterized Queries

Demonstrates the creation of an ISAPI filter using ATL Server's support classes. The filter maps URL requests to new addresses using regular expression replacements provided in a configuration file.

Mailer Sample: Demonstrates SMTP Mail Classes

Demonstrates the SMTP mail classes provided with ATL Server.

MantaWeb Sample: Online Personal Information Management

Demonstrates how to use ATL Server to build a personal information management Web site that includes e-mail, scheduling and task lists, and a file store.

OnlineAddressBook Sample: Stores an Address Book in an XML Web Service

Demonstrates the use of session state to store an address book in an XML Web service.

PerformanceCounter Sample: Performance Monitoring in an ATL Server Application

Demonstrates how to expose performance information using ATL Server's performance monitoring support within a Web application.

PerformancePersist Sample: Provides Custom Performance Monitoring Data

Demonstrates how to expose performance information from an ATL Server Web application without losing that data when IIS is stopped and restarted.

PerformanceScribble Sample: Performance Monitoring in an MFC Application

Demonstrates how to expose performance information using ATL Server's performance monitoring support within an MFC application.

PooledAsync Sample: Increases Performance with an Additional Thread Pool

Demonstrates how to increase Web application and SOAP service availability and performance with an additional thread pool.

RegExp Sample: Demonstrates ATL Regular Expression Classes

Demonstrates how to use CAtlRegExp Class and CAtlREMatchContext to search and replace regular expressions inside a file uploaded to a Web server.

SessionSettings Sample: Session State Demonstration

Demonstrates how to use session state to maintain user preferences across multiple client requests.

SRFSyntax Sample: Server Response File Demonstration

Demonstrates how to use attributed code to implement an ATL Server request handler that illustrates advanced features of server response files.

SecureSOAP Sample: Implements a Secure SOAP Communication (HTTPS)

Demonstrates how to implement a secure SOAP communication.

SOAPDataTypes Sample: Demonstrates Basic Data Types and Aggregated Types in a SOAP Server

Demonstrates the use of basic data types and aggregated types in a SOAP server.

SOAPDebugApp Sample: Debugging XML Web Services in the Client Memory Space

Demonstrates how to debug an XML Web service in the memory space of the client application.

SOAPState Sample: Uses ATL Server to Implement Persistent SOAP Servers

Demonstrates how to implement a persistent SOAP server.

SOAPTransport Sample: Communicates SOAP Messages Over Sockets, MSMQ, the File System, and HTTP Listener

Demonstrates the creation of SOAP servers and clients that communicate using different transports: sockets, Microsoft Message Queue, the file system, and HTTP listener.

StencilCache Sample: Uses the ATL Server Stencil Cache

Demonstrates how to use the stencil cache service in an ATL Server Web application.

Tutorial Sample

The code for the ATL Server Tutorial.

WeatherService Sample: An XML Web Service Created with ATL Server

Demonstrates implementing a simple XML Web service using ATL Server attributes, and accessing an XML Web service from an MFC-based client application. This sample also shows Windows 2000 layered window support.

WebDbg Sample: A Client for Debugging Server Applications Using CDebugReportHook

Demonstrates client code for CDebugReportHook Class in the form of an MFC application providing a user interface for debugging server applications.

WebFeatures Sample: Demonstrates Various ATL Server Tasks

Demonstrates a variety of tasks related to writing ATL Server Web applications, including setting and viewing cookies, using session state, handling uploaded files, pulling images from a database, recognizing users' localization preferences, protecting resources, registering users with a Web site, and more.

See Also

Other Resources

Visual C++ Samples
ATL Samples
ATL Server Concepts
ATL Server Reference

Applications Samples

The following topics are the abstracts for the ATL Server applications samples. These samples are complete applications that cover a range of ATL Server functionality.

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

MantaWeb Sample: Online Personal Information Management

Download sample

Demonstrates how to use ATL Server to build a personal information management Web site that includes e-mail, scheduling and task lists, and a file store.

The MantaWeb sample includes the following features:

- Use of attributed and nonattributed programming for easy comparison
- Use of persistent and session cookies
- Heavy use of stenciled pages to separate dynamic and static content
- Use of ATL Server validation and exchange support
- Use of SMTP classes for e-mail support
- Mail XML Web service with native client for stand-alone e-mail access

Note
Because JET components are not supported on the 64-bit versions of Microsoft Windows, this sample only works with the 32-bit versions.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- IIS

Note
On Windows Server 2003, your machine must be configured as an Application Server.

Building and Running the Sample

To build the sample and set up the Web site

1. Create the following folders on your machine.

   ```
   C:\MantaWebSample
   C:\MantaWebSample\Database
   C:\MantaWebSample\FileStore
   ```

   Note that if you do not use the suggested paths, you will have to change the hardcoded strings in \Common\MWS_Common.h and \FileStore\FilesHandler.h.

2. Set permission for the IUSR_*machinename* user for C:\MantaWebSample\Database and C:\MantaWebSample\FileStore to allow read and write operations.

   To do this, right-click the MantaWebSample folder and click **Properties**. The **Properties** dialog box appears. Click the **Security** tab, and then click **Add**. In the **Select Users, Computers, or Groups** dialog box, ensure that **Object Types** includes **Users**, click **Locations** and select the local machine, and then enter IUSR_*machinename* in **Enter the object names to select** box. Click **Check Names**. An entry for **Internet Guest Account** will appear in the **Properties** dialog box; in the lower pane, select **Read** and **Write**. Click **OK**. This sets the permissions to allow read and write access for the folder and both subfolders.

3. Copy manta\MantaWeb\MantaWeb.mdb to the C:\MantaWebSample\Database directory. Ensure that the database file is not read-only.
4. Open the solution file, MantaWeb.sln, in the Visual Studio integrated development environment.
5. Build the solution. This will also deploy the solution to the local Web server.
6. Use the Internet Information Services Manager to add login.srf as a default document for the MantaWeb virtual directory. Right-click **My Computer**. On the shortcut menu, click **Manage**. In the **Computer Management** dialog box, open **Internet Information Services**, then **Web Sites**, and then **Default Web Site**.
7. Use Internet Explorer to view http://localhost/mantaweb/.

   If you change the virtual directory that the MantaWeb solution deploys to, you will need to take the following steps to update the MFC mail client:

   a. Open the mailservice.h file in the MailClientMFC project.
   b. On line 75, change the string passed to the **TClient** base constructor to reflect the changes made to the path on the Web server.
   c. Build the MailClientMFC project.

To use the Web site

1. Use a Web browser to view http://localhost/mantaweb/login.srf.
2. Sign up for an account. Provide values for all requested information.
3. Log in to the site.
4. Use the site to send mail and manage your personal information.

Notes

By default, the MantaWeb sample will not send mail through the localhost SMTP server unless the following preprocessor defines are set: **SEND_CONFIRMATION_MAIL**, **SEND_FORGOTPASS_MAIL**, and **SEND_OUTGOING_MAIL**. You can uncomment the preprocessor **#define**s in \Common\MWS_Common.h.

This sample was written so that developers can learn about ATL Server without having to have SQL Server installed. It is strongly recommended that you use SQL Server for your Web-based applications because of its enhanced security features. It is also strongly recommended that you use SQL Server stored procedures wherever possible for your database logic.

Project Layout

The project layout of MantaWeb is as follows:

Folder	Description
Calendar	Nonattributed ATL Server application DLL project that provides a task list and basic calendar and scheduling support.
Common	Folder holding header files used by all application DLL projects.
FileStore	Nonattributed ATL Server application DLL project that provides an online file management system.
Login	Attributed ATL Server application DLL project that provides for user signup and login.
Mailbox	Attributed ATL Server application DLL project that provides the mailbox functionality to the Web site.
MailClientMFC	XML Web service MFC client that uses the MantaWeb's XML Web service to check e-mail.
MailService	Attributed ATL Server application DLL project that provides the mail XML Web service and discovery of the X ML Web service.

Main	Attributed ATL Server application DLL project that provides session checking and profile management.
MantaWeb	Nonattributed ATL Server ISAPI DLL project.
MantaWeb\Images	Folder holding all images used on the Web site.

Demonstrates

BEGIN_HANDLER_MAP | BEGIN_REPLACEMENT_METHOD_MAP | CAccessorRowset::Close | CAtlFile::Create | CAtlFile::Read | CCookie::AddValue | CCookie::SetDiscard | CCookie::SetExpires | CCookie::SetName | CCookie::SetPath | CCryptHash::AddData | CCryptHash::GetSize | CCryptHash::GetValue | CCryptProv::GenRandom | CCryptProv::Initialize | CDataConnection | CHttpRequest::Cookies | CHttpRequest::GetFirstFile | CHttpRequest::GetFirstFormVar | CHttpRequest::GetFormVars | CHttpRequest::GetNextFile | CHttpRequest::GetNextFormVar | CHttpRequest::GetQueryParams | CHttpRequest::GetServerContext | CHttpRequestParams::Lookup | CHttpResponse::AppendCookie | CHttpResponse::AppendHeader | CHttpResponse::ClearResponse | CHttpResponse::Flush | CHttpResponse::Redirect | CHttpResponse::SendHeadersInternal | CHttpResponse::SetContentType | CIsapiExtension::GetExtensionVersion | CIsapiExtension::TerminateExtension | CIsapiWorker::GetWorkerData | CMimeHeader::AddRecipient | CMimeHeader::SetSender | CMimeHeader::SetSenderName | CMimeHeader::SetSubject | CMimeMessage::AddText | COleDateTime::GetCurrentTime | COleDateTime::GetDay | COleDateTime::GetDayOfWeek | COleDateTime::GetHour | COleDateTime::GetMinute | COleDateTime::GetMonth | COleDateTime::GetStatus | COleDateTime::GetYear | COleDateTime::ParseDateTime | COleDateTime::SetDate | COleDateTime::SetStatus | COleDateTimeSpan | CRequestHandlerT | CRequestHandlerT::m_HttpRequest | CRequestHandlerT::m_HttpResponse | CRequestHandlerT::ValidateAndExchange | CSMTPConnection::Connect | CSMTPConnection::SendMessage | CStringT::Format | CValidateObject::Exchange | CValidateObject::Validate | CWriteStreamHelper::operator << | db_column | db_command | db_param | EscapeToCString | export | GetDataSource | HANDLER_ENTRY | HTTP_CODE | IHttpServerContext::WriteClient | IRequestHandlerImpl::m_spServiceProvider | REPLACEMENT_METHOD_ENTRY | REPLACEMENT_METHOD_ENTRY_EX | request_handler | soap_handler | soap_method | tag_name

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

Client Samples

The following topics are the abstracts for the ATL Server client samples. These samples demonstrate the use of ATL Server's HTTP client classes to make requests of a Web server.

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

HttpClient Sample: HTTP Client Services Demonstration

Download sample

Demonstrates making HTTP Web requests from within an MFC application using ATL Server's HTTP client support.

The HttpClient sample is an MFC dialog box application that allows users to retrieve and examine an HTTP response from a given URL. The HttpClient sample uses the CAtlHttpClientT class, which simplifies HTTP client operations by providing a simple interface and handling redirections and protection schemes in an automated fashion. After a successful HTTP transaction, the sample displays the response headers and body. If the URL indicates an image, the sample displays the image in a separate dialog box.

This sample demonstrates:

- Using the CAtlHttpClientT class to launch HTTP client requests and receive responses.
- Using the CAtlNavigateData class to modify the default CAtlHttpClientT behavior, including installing a callback that is called periodically during large downloads. The sample uses this callback to update a progress bar.
- Using the CAtlHttpClientT::SetProxy function to direct HTTP transactions to be performed through a proxy server.
- Optional use of the CBasicAuthObject and CNTLMAuthObject classes to allow access to protected URLs.
- Using the ATL CImage class to display image data, if the URL provided by the user points to an image file.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- IIS

Building and Running the Sample

To build and run this sample

1. Open the solution file, HttpClient.sln, in the Visual Studio development environment.
2. Build the solution.
3. From the **Debug** menu, click **Start**.
4. Enter a URL in the **URL** text box, specify proxy server settings if necessary, and click the **Navigate** button. The sample will make an HTTP GET request to the specified URL and display the resulting response data or an error message. If the server returns an image, the sample will display the image in a separate window.

Demonstrates

CAtlHttpClientT | CAtlHttpClientT::AddAuthObj | CAtlHttpClientT::GetBody | CAtlHttpClientT::GetBodyLength | CAtlHttpClientT::GetHeaderValue | CAtlHttpClientT::GetRawResponseHeader | CAtlHttpClientT::GetStatus | CAtlHttpClientT::Navigate | CAtlHttpClientT::SetProxy | CAtlNavigateData::SetReadStatusCallback | CBasicAuthObject::Authenticate | CBasicAuthObject::Init | CImage::BitBlt | CImage::GetHeight | CImage::GetWidth | CImage::Load | CNTLMAuthObject | IAuthInfo::GetDomain | IAuthInfo::GetPassword | IAuthInfo::GetUserName

See Also

Concepts

HTTP Client Services

Other Resources

ATL Server Samples
ATL Server Concepts
HTTP Client Reference
ATL Server Reference

HttpPing Sample: HTTP Client Tool

Download sample

Demonstrates making HTTP Web requests from within a console application using ATL Server's HTTP client support.

The HttpPing sample allows HTTP requests to be launched from the command line, given a URL, and displays the server response headers and response time. The sample also supports optional settings such as proxy server and authorization settings, which can be provided either on the command line, or in an .ini file.

This sample demonstrates:

- Directing HTTP transactions to use a proxy server using the CAtlHttpClientT::SetProxy function.
- Launching HTTP client requests using the ATL Server CAtlHttpClientT class.
- Retrieving the resulting HTTP response headers using the CAtlHttpClientT::GetRawResponseHeader function.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- IIS

Building and Running the Sample

To build and run this sample

1. Open the solution file, HttpPing.sln, in the Visual Studio development environment.
2. Build the solution.
3. Open a command prompt window from the Windows Start menu.
4. Change the current directory to where the HttpPing.exe file is located.
5. If your Internet connection requires the use of a proxy server, edit the HttpPing.ini file. Specify a proxy server name and port number, and set **use_proxy** to **true**.
6. Run the sample using this syntax:

   HttpPing < *URL* >

Demonstrates

CAtlHttpClientT::GetRawResponseHeader | CAtlHttpClientT::GetStatus | CAtlHttpClientT::Navigate | CAtlHttpClientT::SetProxy

See Also

Concepts

HTTP Client Services

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference
HTTP Client Reference

Performance Monitoring Samples

The following topics are the abstracts for the ATL Server performance monitoring samples. These samples demonstrate ATL Server's support for providing performance counters that can be used to allow access to statistical information about a running application or service.

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

PerformanceCounter Sample: Performance Monitoring in an ATL Server Application

Download sample

Demonstrates how to expose performance information using ATL Server's performance monitoring support within a Web application.

The PerformanceCounter sample presents a Web page with three buttons, each of which is represented by an instance of a performance object. When one of the buttons is clicked, the sample updates the corresponding performance object's counter values.

The results can be viewed using the Windows Performance Console.

This sample demonstrates:

- Defining performance objects and counters within an ATL Server application.
- Using the nonattributed variety of the ATL performance monitoring API.
- Creating multiple instances of a performance object.
- Accessing performance objects from an ATL Server request handler.
- The ATL Server classes CPerfMon and CPerfObject.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- IIS

Building and Running the Sample

To build and run this sample

1. Open the solution file, PerformanceCounter.sln, in the Visual Studio development environment.
2. Build the solution. This will also deploy the solution to the local Web server.
3. Use a Web browser to view http://localhost/performancecounter/performancecounter.srf.
4. Run the Windows Performance Monitor. (Click the **Start** button, click **Run**, type **perfmon** in the **Open** box, and click **OK**.)
5. Within the Performance Console, click the **Add** button on the toolbar (represented by a plus sign) to open the **Add Counters** dialog box.
6. In the **Performance objects** box, select **PerformanceCounterSample**.
7. Select the **All instances** and **All counters** radio buttons and click **Add**.
8. Position the Performance Console and Web browser windows so that they do not overlap.
9. Click the buttons on the PerformanceCounter Web page and watch as the Performance Monitor charts the counter values.

Demonstrates

CPerfMon | CPerfObject | DECLARE_PERF_OBJECT | BEGIN_COUNTER_MAP | DEFINE_COUNTER | END_COUNTER_MAP | BEGIN_PERF_MAP | CHAIN_PERF_OBJECT | END_PERF_MAP | CPerfMon::CreateInstanceByName | CPerfMon::Initialize | CPerfMon::UnInitialize | CPerfLock | CPerfLock::GetStatus

See Also

Concepts

Performance Monitoring

Other Resources

ATL Server Samples
ATL Server Concepts
Performance Monitoring Reference
ATL Server Reference

PerformancePersist Sample: Provides Custom Performance Monitoring Data

Download sample

Demonstrates how to expose performance information from an ATL Server Web application without losing that data when IIS is stopped and restarted.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- IIS

Building and Running the Sample

To build and run this sample

1. Open the solution file, PerfPersist.sln, in the Visual Studio development environment.
2. From the **Build** menu, click **Build Solution**.
3. Register PerfServ by running the following command from the command line while in the directory with PerfServ's build output:

   ```
   PerfServ /Service
   ```

4. Start PerfServ by running the following command from the command line:

   ```
   net start PerfServ
   ```

5. Register the performance monitor data in the application DLL by running the following from the virtual directory created in step 2:

   ```
   regsvr32 PerfPersist.dll
   ```

   This directory will probably be C:\inetpub\wwwroot\PerfPersist\bin, but may be different if the IIS root is different
6. From the **Debug** menu, click **Start Without Debugging**.
7. Run the Windows Performance Monitor and add counters for the object "Fibonacci Stats." These are the counters that are provided by the sample ATL Server application.
8. Submit a few requests and verify that the data is changing in the performance monitor.
9. Run the following command from the command line to unload the application DLL:

   ```
   iisreset
   ```

   No performance monitoring data should be lost even though the application providing the data is no longer running. The service PerfServ is keeping this data persistent in shared memory. When the ATL Server application is started again by navigating to the original page, it will reattach to the same shared memory and continue where it left off.

How the Sample Persists the Performance Data

ATL Server request handlers are created for each request and their DLL is only loaded as long as it is needed. If an ATL Server application wants to provide custom performance data using the performance monitoring subsystem, this data must be persisted between request handler invocations. This sample shows two methods for persisting the data:

- Use a COM object to maintain a global instance of a CPerfMon-derived object that manages custom performance counters.

  The COM object is maintained by the ATL Server ISAPI DLL, which is loaded persistently by IIS. When a request handler starts, it requests this service from the ISAPI extension, and gets the CPerfMon object from the service. Alternately, this service could be added directly to a custom ISAPI DLL, but the method used by this sample does not require modification of the wizard-generated ISAPI DLL and is thus suitable for an environment with one generic ISAPI DLL servicing several application DLLs.

- Use an external service process to maintain performance data.

  The service process executable file exposes the **IPerfCache** interface, which the application DLL uses when it creates its CPerfMon-derived object. This ensures that if IIS is stopped or restarted for any reason that the performance data will not be lost even though the ISAPI DLL is no longer maintaining it.

Demonstrates

perfmon | perf_object | perf_counter | CPerfMon | CPerfMon::CreateInstance | IIsapiExtension::AddService | IServiceProvider

See Also

Concepts

Performance Monitoring

Other Resources

ATL Server Samples
ATL Server Concepts
Performance Monitoring Reference
ATL Server Reference

PerformanceScribble Sample: Performance Monitoring in an MFC Application

Download sample

Demonstrates how to expose performance information using ATL Server's performance monitoring support within an MFC application.

The PerformanceScribble sample is identical to the MFC Scribble sample, except that performance data support has been added.

This sample demonstrates:

- Defining performance objects and counters within an MFC application.
- Using the nonattributed variety of the ATL performance monitoring API.
- Performance monitoring support used outside of an ATL Server application.
- The ATL Server classes CPerfMon and CPerfObject.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file, PerformanceScribble.sln, in the Visual Studio development environment.
2. From the **Build** menu, click **Build Solution**.
3. From the **Debug** menu, click **Start Without Debugging**. The PerformanceScribble application will open displaying a window where you can draw with the mouse.
4. Run the Windows Performance Monitor. (Click the **Start** button, click **Run**, type **perfmon** in the **Open** box, and click **OK**.)
5. Within the Performance Console, click the **Add** button on the toolbar (represented by a plus sign) to open the **Add Counters** dialog box.
6. In the **Performance objects** box, select **ScribbleMouse**.
7. Select the **All instances** and **All counters** radio buttons and click **Add**.
8. Position the Performance Console and PerformanceScribble windows so that they do not overlap.
9. Move the mouse around within the drawing area of the PerformanceScribble application and observe the changes in the Performance Console.

How the Sample Works

ATL Server's performance monitoring support must reside within a DLL, so this sample uses two subprojects: one to generate a performance-enabled DLL, and the other to generate an EXE. Together they form the PerformanceScribble sample and expose three counters:

- Horizontal mouse movement
- Vertical mouse movement
- Mouse events per second

Note that without a performance monitoring application such as the Windows Performance Console to monitor performance

output, the PerformanceScribble demonstration looks exactly like the MFC Scribble sample. There is no external evidence that performance-monitoring support has been added.

Demonstrates

CPerfMon | CPerfObject | DECLARE_PERF_OBJECT | BEGIN_COUNTER_MAP | DEFINE_COUNTER | END_COUNTER_MAP | BEGIN_PERF_MAP | CHAIN_PERF_OBJECT | END_PERF_MAP | CPerfMon::CreateInstanceByName | CPerfMon::Initialize | CPerfMon::UnInitialize | CPerfLock | CPerfLock::GetStatus

See Also

Concepts

Performance Monitoring

Other Resources

ATL Server Samples
ATL Server Concepts
Performance Monitoring Reference
ATL Server Reference
SCRIBBLE Sample: MFC MDI Drawing Application

Server Samples

The server samples demonstrate the use of ATL Server to create Web applications and other server-based applications. The ATL Server server samples are in two categories:

General

Illustrate some typical uses of ATL Server to create Web applications.

Advanced

Demonstrate some more advanced features and show how to customize ATL Server code to suit your needs.

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

Server General Samples

The following topics are the abstracts for the ATL Server general server samples. For additional server samples, see Server Samples.

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

Input Sample: Demonstrates User Input Validation On Client And Server

Download sample

Demonstrates validation of user input on both client and server.

The Input sample shows how to use the following classes to validate user input: CAtlRegExp, CAtlREMatchContext, CValidateContext, CHttpRequestParams.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

IIS 4 or later running on Windows NT 4.0 or later

Building and Running the Sample

To build and run this sample

1. Open the solution file, Input.sln, in the Visual Studio development environment.
2. Build the solution. This will also deploy the solution to the local Web server.
3. Use a Web browser to view http://localhost/input/input.srf.

Concepts Presented in the Sample

Three key concepts are presented in this sample:

- Separation of developer and designer tasks
- User-input validation on client and server
- Reuse of regular expression validation routines

Separation of Developer and Designer Tasks

The ATL Server architecture was designed so that the developer needs only to pass a list of the stencil tags to be used to the HTML designer. This makes it possible for the HTML designer to focus on the presentation requirements of the Web page, without having to worry about the implementation details of validation or data retrieval. The developer can then more productively spend time writing C++ code. This sample shows how to achieve a good separation of form and function using some generic stencil tags.

Validation on Client and Server

Modern browsers that support scripting can be used to enhance the user experience by validating input before it is even submitted to the server. Validation on the client provides a convenient way for the user to correct input mistakes and prevents unnecessary round trips to the server.

However, validation on the client cannot be trusted as a security measure, because it is trivial for malicious users to send an HTTP query that bypasses the validation code contained in script, so server-side validation is still essential. ATL Server request handlers should typically add validation code to the **ValidateAndExchange** method of the request handler.

This sample demonstrates a class that provides methods to output client validation code as well as perform server validation inside the **ValidateAndExchange** method of the request handler.

Regular Expression Validation Routines

The validation routines in this sample are based on the principle that it is safer to verify that input data conforms to a known good pattern than to try to catch all the bad patterns. Equivalent regular expressions are provided for CAtlRegExp on the server and JScript on the client.

Demonstrates

CAtlRECharTraitsA | CAtlRegExp | CAtlRegExp::Match | CAtlRegExp::Parse | CAtlREMatchContext | CHttpRequest::GetFormVars | CHttpRequestParams | CHttpRequestParams::Lookup | CRequestHandlerT::ValidateAndExchange | CValidateContext | CValidateContext::ParamsOK | CValidateObject::Validate | request_handler | tag_name

See Also

Other Resources

ATL Server Samples
ATL Server Concepts

Mailer Sample: Demonstrates SMTP Mail Classes

Download sample

Demonstrates the SMTP mail classes provided with ATL Server.

The Mailer sample allows you to fill in a form, attach some files, and send the resulting message through SMTP e-mail.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- IIS

Building and Running the Sample

To build and run this sample

1. Open the solution file, Mailer.sln, in the Visual Studio development environment.
2. Build the solution. This will also deploy the solution to the local Web server.
3. Use a Web browser to view http://localhost/mailer/mailsamp.htm.
4. Compose your message and click Send Message to send the message. The table below describes the fields that allow you to configure the settings for your message:

Field	Description	Example	Optional
Server Name	The SMTP server you are sending to.	smtp.microsoft.com	No
Sender Name	The friendly name of the sender.	John Doe	Yes
From	The sender's e-mail address.	johndoe@microsoft.com	No
Recipient Name	The friendly name of the recipient.	Jane Doe	Yes
To	The recipient's e-mail address.	janedoe@microsoft.com	No
CC	The e-mail address(es) of the recipient(s) to get copies of the message. Multiple addresses should be separated by commas, semicolons, or spaces.	ellen@microsoft.com, frank@microsoft.com	Yes
Subject	The subject of the message.	Greetings!	Yes
Message	The message body.	Hi Jane!	Yes
Priority	The priority of your message.	Normal	Yes

File fields	These fields allow you to attach up to 3 files to your message and choose the encoding scheme.	file1.txt, file2.txt, file3.txt	Yes

Demonstrates

CHttpMap::GetNext | CHttpMap::GetStartPosition | CHttpMap::GetValueAt | CHttpRequest::DeleteFiles | CMimeHeader::AddCc | CMimeHeader::AddRecipient | CMimeHeader::SetPriority | CMimeHeader::SetSender | CMimeHeader::SetSenderName | CMimeHeader::SetSubject | CMimeMessage | CMimeMessage::AddText | CMimeMessage::AttachFile | CSMTPConnection | CSMTPConnection::Connect | CSMTPConnection::SendMessage | CValidateContext | CValidateContext::GetResultAt | CValidateContext::GetResultCount | CValidateContext::ParamsOK | CWriteStreamHelper::Write | IHttpFile::GetFileName | IHttpFile::GetTempFileName

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

RegExp Sample: Demonstrates ATL Regular Expression Classes

Download sample

Demonstrates how to use CAtlRegExp and CAtlREMatchContext to search and replace regular expressions inside a file uploaded to a Web server.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- IIS

Building and Running the Sample

To build and run this sample

1. Open the solution file, rxReplacer.sln, in the Visual Studio development environment.
2. Build the solution. This will also deploy the solution to the local Web server.
3. Use a Web browser to view http://localhost/rxreplacer/rxreplacer.srf.
4. Use the **Browse** button to choose the file in which you want to find/replace a regular expression.
5. Type in the regular expression and the new expression you want to substitute it with.
6. Use the check boxes to set options for your search, including case sensitivity, global substitution, and use of variables.
7. Click **Get Modified File** to obtain the new file.

Demonstrates

CAtlFile::Create | CAtlFile::Read | CAtlRegExp::Match | CAtlRegExp::Parse | CAtlREMatchContext::GetMatch | CAtlREMatchContext::m_Match | CAtlREMatchContext::m_uNumGroups | CAtlREMatchContext::m_uNumGroups | CAtlREMatchContext::MatchGroup | CAtlREMatchContext::MatchGroup::szEnd | CAtlREMatchContext::MatchGroup::szStart | CHttpMap::GetCount | CHttpMap::GetNextValue | CHttpMap::GetStartPosition | CHttpRequest::DeleteFiles | CHttpRequest::m_Files | CRequestHandlerT::FormFlags | CValidateObject::Exchange | IHttpFile::GetFileSize | IHttpFile::GetTempFileName | IHttpFile::GetTempFileName

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

SessionSettings Sample: Session State Demonstration

Download sample

Demonstrates how to use session state to maintain user preferences across multiple client requests.

The SessionSettings sample takes the form of a simple photo gallery, with images that can be displayed either in actual size or scaled, and in high, medium, and low resolutions. Visitors to the photo gallery can specify the desired settings through an HTML page, and submit these preferences to the server. The settings are stored using ATL Server's session state support, where they remain available for use in responding to subsequent HTML requests from the gallery pages.

This sample shows how to:

- Maintain user preferences across multiple HTML requests.
- Add session state support to an ATL Server extension DLL.
- Use either memory or database support to store session state data.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- IIS

Building and Running the Sample

To build and run this sample

1. Open the solution file, SessionSettings.sln, in the Visual Studio development environment.
2. Build the solution. This will also deploy the solution to the local Web server.
3. Use a Web browser to view http://localhost/sessionsettings/gallery.srf.

Demonstrates

ISession | ISessionStateService | ISessionStateControl | CSessionStateService | ISession::SetVariable | ISession::GetVariable | ISessionStateService::CloseSession | ISessionStateService::CreateNewSession | ISessionStateService::GetSession | HttpResponse::Redirect | CMemSessionServiceImpl | CDBSessionServiceImplT

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

SRFSyntax Sample: Server Response File Demonstration

Download sample

Demonstrates how to use attributed code to implement an ATL Server request handler that illustrates advanced features of server response files.

The SRFSyntax sample illustrates how to:

- Associate replacement methods with replacement tags using the tag_name attribute.
- Parse replacement tag parameters and handle them in a replacement method.
- Control the flow of execution within a SRF file using if and while statements.
- Include one SRF file in another using the include tag.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- IIS

Building and Running the Sample

To build and run this sample

1. Open the solution file, SRFSyntaxSample.sln, in the Visual Studio development environment.
2. Build the solution. This will also deploy the solution to the local Web server.
3. Use a Web browser to view http://localhost/srfsyntaxsample/srfsyntax.srf.

Demonstrates

Comment Tag | Replacement Tag | Handler Tag | Include Tag | Subhandler Tag | tag_name | request_handler | CRequestHandlerT::m_HttpResponse | CHttpResponse::SetContentType | IAtlMemMgr | IAtlMemMgr::Allocate | HTTP_S_FALSE

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference
ATL Server Response File Reference

WebFeatures Sample: Demonstrates Various ATL Server Tasks

Download sample

Demonstrates a variety of tasks related to writing ATL Server Web applications, including setting and viewing cookies, using session state, handling uploaded files, pulling images from a database, recognizing users' localization preferences, protecting resources, registering users with a Web site, and more.

The WebFeatures sample consists of a number of small, focused Web application DLLs that each show one particular aspect of using ATL Server to write Web applications. All the Web application DLLs use the same ISAPI extension DLL and can be viewed from the links on the default Web page. The code for each feature being demonstrated is in a separate subfolder of WebFeatures.

Note

In the file ShowCookies.cpp, the code that allows you to view cookies echoes cookies to the client. This is done for demonstration only. In production code, you should never echo cookies to the client. In addition, the following line of code echoes an error message back to the client, a practice that you should also avoid:

```
m_HttpResponse << "<i>" << GetErrorString() << "</i>";
```

Security Note

This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file, WebFeatures.sln, in the Visual Studio development environment.
2. Build the solution. This will also deploy the solution to the local Web server.
3. Use a Web browser to view http://localhost/webfeatures/default.htm.
4. See ConfirmUser and ShowImage for the database installation steps for those parts of the sample.

For more information, see the following links:

Feature	Description
ConfirmUser	Demonstrates how to register a user with a Web site and confirm the validity of the user's e-mail address.
Cookies	Demonstrates how to set cookie data in the response and get cookie data from a request.
ForceLogin	Demonstrates custom user validation and the use of a cookie to identify a previously validated user. Also shows simple use of the CryptoAPI.
HelloWorld	Demonstrates how to implement the simplest possible ATL Server request handler.
NotModified	Demonstrates how to send the Last-Modified response header and check for the If-Modified-Since request header to avoid generating a response body and transmitting data to the client unnecessarily.
ShowAccept	Demonstrates how to read the HTTP_ACCEPT, HTTP_ACCEPT_ENCODING, HTTP_ACCEPT_LANGUAGE, and HTTP_ACCEPT_CHARSET request headers and order the elements according to the user's preference.
ShowBrowser	Demonstrates how to get information about the user's browser from the browser capabilities service.

ShowErrors	Demonstrates techniques for generating error responses to send to the user. Shows the use of ATL Server's default error response, custom static responses, and custom dynamic responses.
ShowFiles	Demonstrates the use of the request object's files collection to get information about files uploaded to the server.
ShowForm	Demonstrates the use of the request object's form variables collection to get information about forms sent to the server.
ShowImage	Demonstrates how to convert, store, and retrieve images. Allows the user to upload images that the server will attempt to convert to PNG format. If successful, the PNG format data is added to a database. Images in the database can be downloaded using a URL that contains the ID of the image as a parameter in the query string.
ShowLocalized	Demonstrates how to match the user's language settings against languages supported by the server and display content specific to the user's locale. The server attempts to use the language setting to display a localized phrase loaded from a resource and the appropriately formatted system time.
ShowQueryString	Demonstrates the use of the request object's query parameters collection to get information about the query string sent to the server.
ShowRequest	Demonstrates how to gather information about the current request by retrieving the properties of the **CHttpRequest** object.
ShowSession	Demonstrates how to get and set session information.
ShowUser	Demonstrates the server variables that are affected by the user and the security settings used on the page, shows how to require authentication by returning a 401 (Unauthorized) status code if the LOGON_USER server variable is empty, and shows how to get the impersonation token for the client.
ShowVariables	Demonstrates how to retrieve the values of the server variables that IIS makes available to your ISAPI extension and request handlers.

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

ConfirmUser

Demonstrates how to register a user with a Web site and confirm the validity of the user's e-mail address.

Requirements

- IIS
- SQL Server

Extra Installation Steps

1. Create a SQL server database and run the Signup.sql SQL script to create the necessary tables and stored procedures.
2. Before building and deploying the application, double-click ConfirmUser.udl and configure the data link properties to point to the database you just set up.

   If the database connection uses Windows authentication, make sure that the **IUSR**_*Machine* account has read-write access to the database. See ATL Server Security for information about the accounts used by a typical ATL Server application running in IIS.

   If the database connection uses SQL authentication, make sure that the security information is persisted.
3. Configure an SMTP server and edit the code in ConfirmUser.cpp to use that server.

Note
A warning advising you to change the server name will appear during the compilation and will guide you to the code t hat needs changing.

4. After building and deploying the application, change the NTFS security settings on the UDL file that was deployed to the same location as ConfirmUser.dll to allow read access by the **IUSR**_*Machine* account.

Instructions

Use a browser to view ConfirmUser.srf. Choose a user name and password, and enter a valid e-mail address. If your user name is not available, you will have a chance to enter another. If your user name is available, the Web application will send you an e-mail message asking you to visit a unique Web page and enter your user name and password. By visiting the page and entering your name and password, the application can be sure that your e-mail address is valid and that you remember your user name and password.

How It Works

The server looks up the user name that you enter to confirm that it is not already taken. If it is free, your details are added to the database and marked as unconfirmed. The application generates a random GUID at this time and stores that in the database along with the other details.

The application generates a message and sends e-mail to the address provided using the CMimeMessage and CSMTPConnection classes. The e-mail contains a URL that includes the GUID generated for you. When you navigate to that URL, the application generates a login page that allows you to enter your user name and password and also adds the GUID to a hidden form field.

When the page is posted back to the server, if the user name, password, and GUID all match an unconfirmed user in the database, that user is marked as confirmed.

Demonstrates

BEGIN_HANDLER_MAP | BEGIN_PARAM_MAP | BEGIN_REPLACEMENT_METHOD_MAP | CCryptHash::AddData | CCryptHash::GetSize | CCryptHash::GetValue | CCryptProv::GenRandom | CCryptProv::GetHandle | CCryptProv::Initialize | CDataSource::OpenFromFileName | CHttpRequest::GetFormVars | CHttpRequest::GetQueryParams | CHttpRequest::GetScriptName | CHttpRequest::GetServerVariable | CHttpRequestParams::Lookup | CHttpResponse::AppendHeader | CHttpResponse::ClearResponse | CHttpResponse::Flush | CHttpResponse::SetCacheControl | CHttpResponse::SetContentType | CHttpResponse::SetExpires | CHttpResponse::SetStatusCode | CMimeHeader | CMimeHeader::AddRecipient | CMimeHeader::SetSender | CMimeHeader::SetSubject | CMimeMessage | CMimeMessage::AddText | COLUMN_ENTRY | CRequestHandlerT | CRequestHandlerT::FormFlags | CRequestHandlerT::m_HttpResponse | CRequestHandlerT::ValidateAndExchange | CSession::Open | CSMTPConnection |

CSMTPConnection::Connect | CSMTPConnection::SendMessage | END_HANDLER_MAP | END_PARAM_MAP | END_REPLACEMENT_METHOD_MAP | HANDLER_ENTRY | OLE DB Consumer Templates | REPLACEMENT_METHOD_ENTRY | SET_PARAM_TYPE

See Also

Other Resources

WebFeatures Sample: Demonstrates Various ATL Server Tasks

Cookies

Demonstrates how to set cookie data in the response and get cookie data from a request.

Requirements

- IIS

Instructions

Use a browser to view SetCookies.srf, then navigate to ShowCookies.srf. The first page sends cookies to the browser. The second page displays the cookies sent by the browser.

How It Works

The SetCookies page sends the following types of cookie to the browser:

- A single-valued session cookie
- A single-valued cookie
- A cookie without a value
- A multivalued cookie
- A single-valued persistent cookie

The ShowCookies page displays the cookies sent by the browser in three different ways:

1. Displays the cookie data available from the CHttpRequest::m_requestCookies collection.
2. Displays the cookie data available when CHttpRequest::GetServerVariable is called for the HTTP_COOKIE server variable.
3. Displays the cookie data available to client script from the cookie property of the document object.

Demonstrates

CCookie | CCookie::AddValue | CCookie::CCookie | CCookie::GetFirstValuePos | CCookie::GetName | CCookie::GetNextValueAssoc | CCookie::GetValue | CCookie::SetExpires | CCookie::SetPath | CHttpMap::GetCount | CHttpMap::GetNextValue | CHttpMap::GetStartPosition | CHttpRequest::GetServerVariable | CHttpRequest::m_requestCookies | CHttpResponse::AppendCookie | CSessionCookie | CSessionCookie::CSessionCookie | SystemTimeToHttpDate

See Also

Other Resources

WebFeatures Sample: Demonstrates Various ATL Server Tasks

ForceLogin

Demonstrates custom user validation and the use of a cookie to identify a previously validated user. Also shows simple use of the CryptoAPI.

Requirements

- IIS

Instructions

Use a browser to view ProtectedResource.srf. Each time you try this, you will be redirected to LoginPage.srf until you have successfully logged in. You will be allowed to log in if the user name and password are identical. Once logged in, you will be redirected to ProtectedResource.srf, and future navigations to that page will succeed until the browser is closed.

How It Works

When the server receives a request for ProtectedResource.srf, the request handler for that page looks for a special cookie. If the cookie is not present or its contents are invalid, the server redirects the user to the login page.

When submitted, the login page returns the user's name and password to the server along with the URL that the user originally requested (the URL is hidden from the user in a hidden form field).

If the login attempt is successful, the user is supplied with the special cookie, then redirected back to the original resource. This time the server allows access to the resource, and will continue to allow access for as long as the user retains the cookie.

Demonstrates

AtlHexEncode | AtlHexEncodeGetRequiredLength | CCookie::GetValue | CCryptHash | CCryptHash::AddData | CCryptHash::Attach | CCryptHash::GetSize | CCryptHash::GetValue | CCryptProv::GenRandom | CCryptProv::GetHandle | CCryptProv::Initialize | CHttpRequest::Cookies | CHttpRequest::GetCookies | CHttpResponse::AppendCookie | CHttpResponse::Redirect | CHttpResponse::SetStatusCode | IHttpServerContext::GetTotalBytes | IHttpServerContext::GetTotalBytes | IRequestHandlerImpl::m_spServerContext

See Also

Other Resources

WebFeatures Sample: Demonstrates Various ATL Server Tasks

HelloWorld

Demonstrates how to implement the simplest possible ATL Server request handler.

Requirements

- IIS

Instructions

Use a browser to view HelloWorld.srf. If you see a simple page containing the text `Hello World!`, you have confirmed that the ISAPI DLL is correctly installed and that other ATL Server request handlers can use it.

Demonstrates

CRequestHandlerT

See Also

Other Resources

WebFeatures Sample: Demonstrates Various ATL Server Tasks

NotModified

Demonstrates how to send the Last-Modified response header and check for the If-Modified-Since request header to avoid generating a response body and transmitting data to the client unnecessarily.

Requirements

- IIS

Instructions

Use a browser to view NotModified.srf and note the time at which the page was created (displayed in the page itself). Refresh the page and see if the creation date changes. If client-side caching is enabled in the browser, the page will not be updated until at least a minute has passed since the last version of the page was cached.

How It Works

The server compares the value of the HTTP_IF_MODIFIED_SINCE request header with the current time. If less than 1 minute has passed, the server returns a 304 (Not Modified) status code without generating a response body. If more than 1 minute has passed, or the HTTP_IF_MODIFIED_SINCE header was not present in the request, the server generates the response in the standard way and sets the Last-Modified response header with the current server time.

Details

Use the 304 (Not Modified) HTTP status code when:

- The client has performed a conditional GET request.
- Access to the resource is allowed.
- The content cached by the client is up to date.

The benefit of this status code is that it allows the server to avoid generating a response body and transmitting that data to the client.

To make use of this status code:

- Append a Last-Modified header and/or an ETag header to a successful (200 OK) response for this resource. This supplies the client with the information it needs to make conditional requests in the future.
- Check the HTTP_IF_MODIFIED_SINCE and/or HTTP_IF_NONE_MATCH request headers. You should receive a request containing a HTTP_IF_MODIFIED_SINCE header if the client cached a response that contained a Last-Modified header. You should receive a request containing a HTTP_IF_NONE_MATCH header if the client cached a response that contained an ETag header.
- Use HTTP_IF_MODIFIED_SINCE for time-based validation of the cached content.
- Use HTTP_IF_NONE_MATCH for other validation of the cached content (the meaning of the ETag is decided by the server).

Demonstrates

CHttpResponse::AppendHeader | CHttpResponse::SetContentType | COleDateTime | IHttpServerContext::GetServerVariable | HTTP_NOT_MODIFIED

See Also

Other Resources

WebFeatures Sample: Demonstrates Various ATL Server Tasks

ShowAccept

Demonstrates how to read the HTTP_ACCEPT, HTTP_ACCEPT_ENCODING, HTTP_ACCEPT_LANGUAGE, and HTTP_ACCEPT_CHARSET request headers and order the elements according to the user's preference.

Requirements

- IIS

Instructions

View ShowAccept.srf in a browser to see the MIME types, languages, character sets, and encodings that your browser indicates are acceptable in the request that it sent to the server. Change your browser's language settings to see that the server recognizes the priority you give to each language. In Internet Explorer, you can change your language settings. On the **Tools** menu, click **Internet Options**. On the **General** tab, click **Languages** and select the appropriate language settings.

How It Works

The code parses each of the HTTP_ACCEPT_* headers into its separate components. The elements of the request header are added to a collection ordered by their priority value. The collections are then displayed in a table.

Demonstrates

CHttpRequest::GetUserLanguages | CHttpRequest::GetAcceptEncodings | CHttpRequest::GetAcceptTypes | CHttpRequest::GetServerVariable

See Also

Other Resources

WebFeatures Sample: Demonstrates Various ATL Server Tasks

ShowBrowser

Demonstrates how to get information about the user's browser from the browser capabilities service.

Requirements

- IIS

Instructions

View ShowBrowser.srf with various browsers or different versions of the same browser to see how the table of browser properties changes from browser to browser. The first line of the table shows the HTTP_USER_AGENT request header. The remaining lines show properties that are calculated by the browser capabilities object.

How It Works

The request handler queries the service provider for the browser capabilities service and uses that to provide the properties displayed on the page. The browser capabilities service compares the HTTP_USER_AGENT request header against information stored in an XML file.

Demonstrates

CHttpRequest::GetServerVariable | IBrowserCaps | IBrowserCaps::GetBrowserName | IBrowserCaps::GetMajorVer | IBrowserCaps::GetMinorVer | IBrowserCaps::GetPlatform | IBrowserCaps::GetVersion | IBrowserCaps::IsAK | IBrowserCaps::IsAOL | IBrowserCaps::IsBeta | IBrowserCaps::IsCrawler | IBrowserCaps::IsSK | IBrowserCaps::IsUpdate | IBrowserCaps::IsWin16 | IBrowserCaps::SupportsActiveXControls | IBrowserCaps::SupportsAuthenticodeUpdate | IBrowserCaps::SupportsBackgroundSounds | IBrowserCaps::SupportsCDF | IBrowserCaps::SupportsCookies | IBrowserCaps::SupportsFrames | IBrowserCaps::SupportsJavaApplets | IBrowserCaps::SupportsJavaScript | IBrowserCaps::SupportsTables | IBrowserCaps::SupportsVBScript | IBrowserCapsSvc | IBrowserCapsSvc::GetCaps | IRequestHandlerImpl::m_spServiceProvider

See Also

Other Resources

WebFeatures Sample: Demonstrates Various ATL Server Tasks

ShowErrors

Demonstrates techniques for generating error responses to send to the user. Shows the use of ATL Server's default error response, custom static responses, and custom dynamic responses.

Requirements

- IIS

Instructions

Use a browser to view ShowErrors.srf and use the form to choose the error that the server will send back when you click Submit. Select an HTTP status code using the drop-down list and optionally type a numeric subcode into the text box. To view the response generated by the default error provider, leave the Return Custom Error box unchecked. To view a custom error response, select that box. If you select 404 with a subcode of zero, you should see the custom error file (404.htm) that was deployed to the local errors folder.

To ensure that you are seeing the errors sent by the server, you must disable Internet Explorer friendly error messages. On the **Tools** menu, click **Internet Options** and select the **Advanced** tab. Under the **Browsing** heading, make sure that the **Show friendly HTTP error messages** is not selected. Note that the dynamically generated custom errors look similar to Internet Explorer friendly error messages but can be viewed in any browser because they are generated by the server.

How It Works

If Return Custom Error is cleared, the request handler builds an error code from the selected HTTP status code and subcode using the HTTP_ERROR macro. This value is returned from **ValidateAndExchange** and the ATL Server infrastructure uses CDefaultErrorProvider to build a response.

If Return Custom Error is selected, the request handler does the following:

1. Clears the response object of any headers and body that may have been set.
2. Prevents the error response from being cached by setting the Cache-Control and Expires response headers.
3. Applies the HTTP status code to the response object.
4. Tries to transmit a file that corresponds to the selected status code and subcode from the local errors folder.
5. If the previous step fails, tries to transmit a file from the errors folder for the site.
6. If the previous step fails, generates a generic response that features instructions for the user to try to work around the problem as well as technical details (time, date, URL, status code, and subcode) that can be used by technical support to track down the cause of the problem.

Demonstrates

CDefaultErrorProvider::GetErrorText | CHttpRequest::GetMethod | CHttpRequest::HTTP_METHOD_HEAD | CHttpResponse::ClearResponse | CHttpResponse::Flush | CHttpResponse::GetServerContext | CHttpResponse::SetCacheControl | CHttpResponse::SetExpires | CHttpResponse::SetStatusCode | HTTP_ERROR_CODE | HTTP_SUBERROR_CODE | IHttpServerContext::GetScriptPathTranslated | IHttpServerContext::SendResponseHeader | IHttpServerContext::WriteClient

See Also

Other Resources

WebFeatures Sample: Demonstrates Various ATL Server Tasks

ShowFiles

Demonstrates the use of the request object's files collection to get information about files uploaded to the server.

Requirements

- IIS

Instructions

Use a browser to view ShowFiles.srf and use the form linked to from that page to send a request that contains one or more files. The information about each file will be displayed in a table. The files will be accepted at the start of the request, saved in the temporary folder on the server, and deleted at the end of the request.

How It Works

The request handler loops through the files in CHttpRequest::m_Files and displays the properties obtained from the IHttpFile interface. Finally the code calls CHttpRequest::DeleteFiles to ensure that any files sent to the server are deleted.

Demonstrates

CHttpMap::GetNextValue | CHttpRequest::DeleteFiles | CHttpRequest::FileMap | CHttpRequest::GetUrlReferer | CHttpRequest::m_Files | CRequestHandlerT::FormFlags | IHttpFile | IHttpFile::GetContentType | IHttpFile::GetFileName | IHttpFile::GetFileSize | IHttpFile::GetFullFileName | IHttpFile::GetParamName | IHttpFile::GetTempFileName

See Also

Other Resources

WebFeatures Sample: Demonstrates Various ATL Server Tasks

ShowForm

Demonstrates the use of the request object's form variables collection to get information about forms sent to the server.

Requirements

- IIS

Instructions

Use a browser to view ShowForm.srf and use the form linked to from that page to send a request back to that page. The information about each element of the form will be displayed in a table. Try changing the submit method or submit encoding to see whether that affects the information displayed by the page.

How It Works

The request handler loops through the form variables in CHttpRequest::GetFormVars and displays the information for each variable.

Demonstrates

CHttpRequest::GetFormVars | CRequestHandlerT::FormFlags | CHttpRequest::DeleteFiles | CHttpRequestParams | CHttpMap::GetCount | CHttpMap::GetStartPosition | CHttpMap::GetNextAssoc

See Also
Other Resources
WebFeatures Sample: Demonstrates Various ATL Server Tasks

ShowImage

Demonstrates how to convert, store, and retrieve images. Allows the user to upload images that the server will attempt to convert to PNG format. If successful, the PNG format data is added to a database. Images in the database can be downloaded using a URL that contains the ID of the image as a parameter in the query string.

Requirements

- IIS
- SQL Server

Extra Installation Steps

1. Create a SQL server database and run the ShowImage.sql SQL script to create the necessary tables and stored procedures.
2. Before building and deploying the application, double-click ShowImage.udl and configure the data link properties to point to the database you just set up.

   If the database connection uses Windows authentication, make sure that the **IUSR**_*Machine* account has read-write access to the database. See ATL Server Security for information about the accounts used by a typical ATL Server application running in IIS.

   If the database connection uses SQL authentication, make sure that the security information is persisted.
3. After building and deploying the application, change the NTFS security settings on the UDL file that was deployed to the same location as ShowImage.dll to allow read access by the **IUSR**_*Machine* account.

Instructions

Use a browser to view ShowImage.srf and use the form linked to from that page to send some image files to the server. The server will attempt to convert the files to PNG format and add the image data to the database. The response will indicate which files were received by the server, which files were images, which could be converted to PNG format, and which of those could be added to the database. Any images added to the database will be displayed as part of the page.

How It Works

The server loops through the files sent to the server and loads each of them into a CImage object. If the file could be loaded successfully, the server generates a temporary file name and tries to save the image to that file name in PNG format. If that is successful, the server loads the file into an object that exposes the data through the IWriteStream interface and hooks that up to an OLE DB accessor. The server then calls a stored procedure using that accessor to add the image to the database along with information such as its width, height, and bits per pixel.

Each image added to the database causes an IMG element to be added to the response. The SRC attribute of the IMG is a URL to this page. The presence of a query string indicates that the server should send back image data retrieved from the database rather than an HTML page. In this example, the query string just contains the ID of the database record holding the image. This ID is used as a parameter to a stored procedure that returns the image data that is then written to the client.

Demonstrates

AtlHresultFromLastError | BEGIN_HANDLER_MAP | BEGIN_REPLACEMENT_METHOD_MAP | CAtlFile::Create | CAtlFile::GetSize | CCommand::Open | CDataSource::OpenFromFileName | CHttpMap::GetCount | CHttpMap::GetNextValue | CHttpMap::GetStartPosition | CHttpMap::Lookup | CHttpRequest::FileMap | CHttpRequest::GetFormVars | CHttpRequest::GetMethod | CHttpRequest::HTTP_METHOD_HEAD | CHttpRequestParams::BaseMap | CHttpResponse::AppendHeader | CHttpResponse::ClearResponse | CHttpResponse::Flush | CHttpResponse::GetServerContext | CHttpResponse::SetCacheControl | CHttpResponse::SetContentType | CHttpResponse::SetExpires | CHttpResponse::WriteLen | CImage | CImage::GetBPP | CImage::GetHeight | CImage::GetWidth | CImage::Load | CImage::Save | COleDateTime | COleDateTime::COleDateTime | COleDateTimeSpan | CRequestHandlerT::FormFlags | CRequestHandlerT::MaxFormSize | CRowset::MoveFirst | CSession::Open | END_HANDLER_MAP | END_REPLACEMENT_METHOD_MAP | HANDLER_ENTRY | IHttpFile | IHttpFile::GetContentType | IHttpFile::GetFileName | IHttpFile::GetFileSize | IHttpFile::GetFileSize | IHttpFile::GetFullFileName | IHttpFile::GetParamName | IHttpFile::GetTempFileName | IHttpServerContext | IHttpServerContext::GetServerVariable | OLE DB Consumer Templates | REPLACEMENT_METHOD_ENTRY

See Also

Other Resources

WebFeatures Sample: Demonstrates Various ATL Server Tasks

ShowLocalized

Demonstrates how to match the user's language settings against languages supported by the server and display content specific to the user's locale. The server attempts to use the language setting to display a localized phrase loaded from a resource and the appropriately formatted system time.

Requirements

- IIS

Instructions

Use a browser to view ShowLocalized.srf. The server will attempt to use your language settings to display a localized phrase and the current system time formatted according to the language supported by the server that most closely matches your language settings. Change your browser's language settings to see that the server recognizes the priority you give to each language. Try changing your favored language to one that is not directly supported by the server. In Internet Explorer, you can change your language settings. On the **Tools** menu, click **Internet Options**. On the **General** tab, click **Languages** and select the appropriate language settings.

How It Works

The request handler parses the HTTP_ACCEPT_LANGUAGES request header to get a prioritized list of languages supported by the client. The server then compares the languages accepted by the client with the list of languages supported by the server. The best match is used to display the message and system time.

Demonstrates

AtlUnicodeToUTF8 | CStringT::LoadString | IMultiLanguage

See Also
Other Resources
WebFeatures Sample: Demonstrates Various ATL Server Tasks

ShowQueryString

Demonstrates the use of the request object's query parameters collection to get information about the query string sent to the server.

Requirements

- IIS

Instructions

Use a browser to view ShowQueryString.srf. Use the link on that page to send a request containing a query string to the page and view the results.

How It Works

The page displays the query string obtained from the CHttpRequest::GetQueryParams collection and the raw text from the QUERY_STRING variable, which are both available on the server, along with the location.search property of the document object, which is available through client-side script.

Demonstrates

CHttpRequest::GetQueryParams

See Also

Other Resources

WebFeatures Sample: Demonstrates Various ATL Server Tasks

ShowRequest

Demonstrates how to gather information about the current request by retrieving the properties of the **CHttpRequest** object.

Requirements

- IIS

Instructions

View ShowRequest.srf in a browser to see the properties that ATL Server makes available through the CHttpRequest object.

Demonstrates

CHttpRequest | CHttpRequest::GetAcceptEncodings | CHttpRequest::GetAcceptTypes | CHttpRequest::GetAuthenticated | CHttpRequest::GetAuthenticationType | CHttpRequest::GetAuthUserName | CHttpRequest::GetAuthUserPassword | CHttpRequest::GetAvailableBytes | CHttpRequest::GetAvailableData | CHttpRequest::GetContentType | CHttpRequest::GetMethodString | CHttpRequest::GetPathInfo | CHttpRequest::GetPathTranslated | CHttpRequest::GetQueryString | CHttpRequest::GetScriptName | CHttpRequest::GetScriptPathTranslated | CHttpRequest::GetTotalBytes | CHttpRequest::GetUrl | CHttpRequest::GetUrlReferer | CHttpRequest::GetUserAgent | CHttpRequest::GetUserHostAddress | CHttpRequest::GetUserHostName | CHttpRequest::GetUserLanguages | CHttpRequest::GetUserName | CRequestHandlerT::m_HttpRequest

See Also

Other Resources

WebFeatures Sample: Demonstrates Various ATL Server Tasks

ShowSession

Demonstrates how to get and set session information.

Requirements

- IIS

Instructions

Use a browser to view ShowSession.srf and use the form to set or get the value of a session variable. The number of sessions and session variables will update on each request to the server.

Demonstrates

ISessionStateService | ISessionStateService::CreateNewSession | ISessionStateService::GetSession | ISession | ISession::GetVariable | ISession::SetVariable | ISession::GetCount | ISessionStateControl::GetSessionCount

See Also
Other Resources
WebFeatures Sample: Demonstrates Various ATL Server Tasks

ShowUser

Demonstrates the server variables that are affected by the user and the security settings used on the page, shows how to require authentication by returning a 401 (Unauthorized) status code if the LOGON_USER server variable is empty, and shows how to get the impersonation token for the client.

Requirements

- IIS

Instructions

Use a browser to view ShowUser.srf. Try changing the security settings for this page in Internet Services Manager to see how the displayed values change.

How It Works

The request handler gets the value of the LOGON_USER server variable and checks its length. If the length is zero, the request handler returns HTTP_UNAUTHORIZED as the code. As long as one of the Authenticated access methods is enabled for the document, the browser will allow users to authenticate themselves; otherwise users will just see that access has been denied.

When authentication occurs, the page displays the following server variables:

- AUTH_PASSWORD
- AUTH_TYPE
- AUTH_USER
- HTTP_AUTHORIZATION
- LOGON_USER
- REMOTE_USER

The request handler also gets the impersonation token for the user by calling IHttpServerContext::GetImpersonationToken.

Demonstrates

HTTP_UNAUTHORIZED | IHttpServerContext::GetImpersonationToken

See Also

Other Resources

WebFeatures Sample: Demonstrates Various ATL Server Tasks

ShowVariables

Demonstrates how to retrieve the values of the server variables that IIS makes available to your ISAPI extension and request handlers.

Requirements

- IIS

Instructions

View ShowVariables.srf in a browser to see the values of the server variables made available to the request handler. Try refreshing the page, posting a form to the page from Form.srf, or changing other settings related to the page or your browser to see how the server variables change.

How It Works

The request handler calls CHttpRequest::GetServerVariable on a succession of known server and environment variables and displays the results.

Demonstrates

CHttpRequest::GetServerVariable

See Also

Other Resources

WebFeatures Sample: Demonstrates Various ATL Server Tasks

ATL Server Advanced Samples

The following topics are the abstracts for the ATL Server advanced samples. For additional server samples, see Server Samples.

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

ATLRT Sample: Demonstrates Custom SRF Tags

Download sample

Demonstrates extending ATL Server's stencil-parsing code to support custom tags. The custom tags supplied in this sample provide support for variables and generic database access from within a SRF file.

ATLRT is an ATL Server request handler that exposes a set of custom tags that can be used to build ATL Server applications. This sample illustrates database access using OLE DB consumer templates as well as how ATL Server tags can be used to create a virtual scripting language.

The tags used in this sample are described in ATLRT Tag Reference.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- SQL Server
- IIS

Building and Running the Sample

To build and run this sample

1. Ensure that you have a SQL Server database containing a stored procedure that returns a few results. If necessary, use the pubs database sample distributed with SQL Server and execute the SQL script in atlrt.sql to add a stored procedure to return all of the authors.
2. Edit atlrt.srf and change the values in the {{SetConnection}} and {{Execute}} tags to provide the connection string to your database and the name of a stored procedure.
3. Open the solution file, AtlRt.sln, in the Visual Studio development environment.
4. Build the solution. This will also deploy the solution to the local Web server.
5. Use a Web browser to view http://localhost/atlrt/atlrt.srf.

See Also

Other Resources

ATL Server Samples

ATLRT Tag Reference

These custom ATLRT tags can be used to build ATL Server applications:

- CloseResults
- CompareValue
- ContainsValue
- CopyValue
- Execute
- GetColumnNumber
- GetColumnValue
- GetRowNumber
- GetValue
- MaintainValue
- MoveNextColumn
- MoveNextRow
- ResetResults
- RowCount
- SaveColumnValue
- SetConnection

See Also

Other Resources

ATLRT Sample: Demonstrates Custom SRF Tags
ATL Server Samples
ATL Server Concepts
ATL Server Reference

CloseResults Tag

Used to close a result set.

```
{{ CloseResults([result_name]) }}
```

Remarks

The Execute tag must be used before **CloseResults**. *result_name* can be used to specify the result set to close. If this parameter is not specified, the default result set is closed.

See Also

Reference

ATLRT Tag Reference

Other Resources

ATLRT Sample: Demonstrates Custom SRF Tags

ATL Server Samples

ATL Server Concepts

ATL Server Reference

CompareValue Tag

Compares the value of the variable specified by **name** with the literal value specified by **value**.

```
{{ CompareValue( name=variable_to_compare; value=literal_value ) }}
```

Remarks

See GetValue tag for information about variables. **CompareValue** returns **true** or **false** depending on whether the values are equal. **CompareValue** does not generate any output and should be used in an {{**if**}} statement.

See Also

Reference

ATLRT Tag Reference

Other Resources

ATLRT Sample: Demonstrates Custom SRF Tags

ATL Server Samples

ATL Server Concepts

ATL Server Reference

ContainsValue Tag

Similar to the GetValue tag, except that **ContainsValue** only checks for the existence of the variable specified by *name*.

```
{{ ContainsValue( name ) }}
```

Remarks

ContainsValue generates no output, but returns **true** or **false**, depending on whether *name* exists or not. **ContainsValue** should be used in an {{**if**}} statement.

See Also

Reference

ATLRT Tag Reference

Other Resources

ATLRT Sample: Demonstrates Custom SRF Tags

ATL Server Samples

ATL Server Concepts

ATL Server Reference

CopyValue Tag

Copies the value of the variable specified by *source_variable* into the variable map entry specified by *new_variable*.

```
{{ CopyValue( name=source_variable; dest=new_variable ) }}
```

Remarks

Values are obtained for the variable specified by *source_variable* in the same manner as GetValue. This tag generates no output.

See Also

Reference

ATLRT Tag Reference

Other Resources

ATLRT Sample: Demonstrates Custom SRF Tags

ATL Server Samples

ATL Server Concepts

ATL Server Reference

Execute Tag

Executes a stored procedure.

```
{{ Execute(
   cmd=stored_procedure_name
   [;conn=connection_name];
   [;results=result1, result2, result3, ...]
   [;params=variable_param1, 'literal_param2', ...]
   [;output_params=param1, param2, ...]
) }}
```

Remarks

The SetConnection tag must be used before the **Execute** tag. The following is a description of the parameters that the **Execute** tag accepts:

- **cmd** =*stored_procedure_name* is the only required parameter. Used to specify the stored procedure name.
- **conn** is used to specify the name of the connection string to use. If **conn** is not specified, then the default connection string is used. Whether named or not, the connection string must have been set by a previous use of the SetConnection tag.
- **results** is optional. If a name is specified, the result set returned by the stored procedure is given that name so that it can be referred to from other tags. If **results** is not specified, then any results of the stored procedure will be stored using the default name. Specifying more than one value for **results** implies that the stored procedure will return multiple results.
- **params** is used to pass parameters to the stored procedure. Parameters can be variables, in which case the value of the variable is passed to the stored procedure, or they can be literals, in which case the parameter is passed directly. Literals must be surrounded by single quotation (' ') marks to distinguish them from variables. The value of a variable is obtained from the same places as the GetValue tag: the variable map, the request parameters, or server variables.
- **output_params** is used to name any output parameters returned by the stored procedure. The values of the output parameters will be stored in the variable map with the specified names.

See Also

Reference

ATLRT Tag Reference

Other Resources

ATLRT Sample: Demonstrates Custom SRF Tags

ATL Server Samples

ATL Server Concepts

ATL Server Reference

GetColumnNumber Tag

Writes the number of the current column to the output stream.

```
{{ GetColumnNumber( [result_name] ) }}
```

Remarks

The Execute tag must be used before **GetColumnNumber**. *result_name* is optional and can be used to specify the result set from which to get the column number. The default result set is used if this parameter is not specified.

See Also

Reference

ATLRT Tag Reference

Other Resources

ATLRT Sample: Demonstrates Custom SRF Tags

ATL Server Samples

ATL Server Concepts

ATL Server Reference

GetColumnValue Tag

Used to get the value of a specific column from a set of results.

```
{{ GetColumnValue( [column_index] ) }}
{{ GetColumnValue( name=result_name[; column=column_index] ) }}
```

Remarks

The Execute tag must be used before **GetColumnValue**. The **GetColumnValue** parameters determine which column is used. The following are the possible ways to call **GetColumnValue**:

- **{{ GetColumnValue() }}**

  Use the default result set and the current column value stored for that result.

- **{{ GetColumnValue(name=** *result_name* **) }}**

  Use the specified result set and the current value stored for that result.

- **{{ GetColumnValue(name=** *result_name* **;column=** *column_index* **) }}**

  Use the specified result set and the specified column value.

- **{{ GetColumnValue(** *column_index* **) }}**

  Use the default result set and the specified column value.

See Also

Reference

ATLRT Tag Reference

Other Resources

ATLRT Sample: Demonstrates Custom SRF Tags

ATL Server Samples

ATL Server Concepts

ATL Server Reference

GetRowNumber Tag

Writes the number of the current row to the output stream.

```
{{ GetRowNumber( [result_name] ) }}
```

Remarks

The Execute tag must be used before **GetRowNumber**. *result_name* is optional and can be used to specify the result set from which to get the row number. The default result set is used if this parameter is not specified.

See Also

Reference

ATLRT Tag Reference

Other Resources

ATLRT Sample: Demonstrates Custom SRF Tags

ATL Server Samples

ATL Server Concepts

ATL Server Reference

GetValue Tag

Writes the value of the variable specified by *name* to the output stream.

```
{{ GetValue( name ) }}
```

Remarks

GetValue looks for variables in the following order:

- Variable map
- Request variables (POST and GET)
- Server variables

Variables can be stored in the variable map by using the database operations. Request variables are set by form submissions. Server variables are set on every request of a .srf file.

See Also

Reference

ATLRT Tag Reference

Other Resources

ATLRT Sample: Demonstrates Custom SRF Tags

ATL Server Samples

ATL Server Concepts

ATL Server Reference

MaintainValue Tag

Stores the specified variable into a hidden input field with the same name as the variable.

```
{{ MaintainValue( variable_name ) }}
{{ MaintainValue( name=variable_name; save_as=new_name ) }}
```

Remarks

For example, given a variable named `testing` whose value is `"Hello World"`:

```
{{ MaintainValue(testing) }}
```

would be replaced by:

```
<input type="hidden" name="testing" value="Hello World">
```

You can also name the hidden field by using the **save_as=***new_name* parameter. For example:

```
{{ MaintainValue(name=testing; save_as=testing2) }}
```

would be replaced by:

```
<input type="hidden" name="testing2" value="Hello World">
```

See Also

Reference

ATLRT Tag Reference

Other Resources

ATLRT Sample: Demonstrates Custom SRF Tags

ATL Server Samples

ATL Server Concepts

ATL Server Reference

MoveNextColumn Tag

Moves to the next column in the specified result set.

```
{{ MoveNextColumn( [result_name] ) }}
```

Remarks

The Execute tag must be used before **MoveNextColumn**. **MoveNextColumn** generates no output, but it will return **false** if there are no more columns in the current row. **MoveNextColumn** should be used in a {{**while**}} or {{**if**}} statement.

See Also

Reference

ATLRT Tag Reference

Other Resources

ATLRT Sample: Demonstrates Custom SRF Tags

ATL Server Samples

ATL Server Concepts

ATL Server Reference

MoveNextRow Tag

Moves to the next row in the specified result set.

```
{{ MoveNextRow( [result_name] ) }}
```

Remarks

The Execute tag must be used before **MoveNextRow**. **MoveNextRow** generates no output, but it will return **false** if there are no more rows in the result set. **MoveNextRow** should be used in a {{**while**}} or {{**if**}} statement.

See Also

Reference

ATLRT Tag Reference

Other Resources

ATLRT Sample: Demonstrates Custom SRF Tags

ATL Server Samples

ATL Server Concepts

ATL Server Reference

ResetResults Tag

Used to reset the current row and column number of the specified result set.

```
{{ ResetResults( [result_name] ) }}
```

Remarks

The Execute tag must be used before **ResetResults**. The default result set is used if no name is specified.

See Also

Reference

ATLRT Tag Reference

Other Resources

ATLRT Sample: Demonstrates Custom SRF Tags

ATL Server Samples

ATL Server Concepts

ATL Server Reference

RowCount Tag

Writes the number of rows in the specified result set to the output stream.

```
{{ RowCount( [result_name] ) }}
```

Remarks

The Execute tag must be used before **RowCount**. The default result set is used if no name is specified.

See Also

Reference

ATLRT Tag Reference

Other Resources

ATLRT Sample: Demonstrates Custom SRF Tags

ATL Server Samples

ATL Server Concepts

ATL Server Reference

SaveColumnValue Tag

Similar to the GetColumnValue tag except that it stores the specified column value into the variable map instead of writing it to the output stream.

```
{{ SaveColumnValue(
   [name=result_name;column=column_index;save_as=save_as_name] |
   [name=result_name;column=column_index] |
   [save_as=save_as_name] |
   [column=column_index;save_as=save_as_name]
) }}
```

Remarks

The Execute tag must be used before **SaveColumnValue**. The following are possible ways to use **SaveColumnValue**:

- **{{ SaveColumnValue() }}**

  Save the value of the current column of the default result set to the variable map under the default variable name.

- **{{SaveColumnValue(name=** *result_name* **;column=** *column_index* **;save_as=** *save_as_name* **) }}**

  Save the value of the column specified by *column_index* of the result set specified by *result_name* in the variable map under the name *save_as_name*.

- **{{ SaveColumnValue(name=** *result_name* **;column=** *column_index* **) }}**

  Save the value of the column specified by *column_index* of the result set specified by *result_name* in the variable map under the name *result_name*.

- **{{ SaveColumnValue(save_as=** *save_as_name* **) }}**

  Save the value of the current column of the default result set in the variable map under the name *save_as_name*.

- **{{ SaveColumnValue(column=** *column_index* **;save_as=** *save_as_name* **) }}**

  Save the value of the column specified by *column_index* of the default result set in the variable map under the name *save_as_name*.

See Also

Reference

ATLRT Tag Reference

Other Resources

ATLRT Sample: Demonstrates Custom SRF Tags

ATL Server Samples

ATL Server Concepts

ATL Server Reference

SetConnection Tag

Used to store a connection string.

```
{{ SetConnection( [name=connection_name; conn=]connection_string ) }}
```

Remarks

The Execute tag, used to connect to data sources, will use these connection strings. You can use multiple connection strings on a page by naming them with the **name=***connection_name* parameter. If you do not specify a **name** parameter, then the connection string is stored using a default name.

See Also

Reference

ATLRT Tag Reference

Other Resources

ATLRT Sample: Demonstrates Custom SRF Tags

ATL Server Samples

ATL Server Concepts

ATL Server Reference

BlobCache Sample: Generic Memory Caching

Download sample

Demonstrates the use of the **CBlobCache** class from within an ATL Server application.

The BlobCache sample presents a Web page featuring a text input field that accepts numbers from 1 to 100. Entering a number and pressing the Lookup button causes the page to submit the value to the application DLL, which then uses the number as the key for a data item in an in-memory cache. If the cache contains an entry to the given number, the cached value corresponding to the key is returned; otherwise the number is used as an index into a data file containing strings. Regardless of how it was retrieved, the resulting value is displayed in the HTTP response, along with a message indicating whether the retrieved string had been cached.

This sample demonstrates:

- Using ATL Server's CBlobCache class to cache generic blocks of memory.
- Using the IMemoryCache, IMemoryCacheControl, and IMemoryCacheStats interfaces to manipulate caching entities.
- Caching server-side data to improve response times.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- IIS

Building and Running the Sample

To build and run this sample

1. Open the solution file, BlobCache.sln, in the Visual Studio development environment.
2. Build the solution. This will also deploy the solution to the local Web server.
3. Use a Web browser to view http://localhost/blobcache/blobcache.srf.
4. Enter a numeric value between 1 and 100 and click the Lookup button. See how varying or repeating requests for items affects the cache statistics displayed on the page.

Demonstrates

CBlobCache | CBlobCache::Initialize | CBlobCache::QueryInterface | CBlobCache::Uninitialize | CHttpRequest::GetPhysicalPath | IMemoryCache | IMemoryCache::Add | IMemoryCache::Flush | IMemoryCache::GetData | IMemoryCache::LookupEntry | IMemoryCache::ReleaseEntry | IMemoryCacheClient | IMemoryCacheClient::Free | IMemoryCacheControl | IMemoryCacheControl::GetMaxAllowedEntries | IMemoryCacheControl::ResetCache | IMemoryCacheControl::SetMaxAllowedEntries | IMemoryCacheStats | IMemoryCacheStats::ClearStats | IMemoryCacheStats::GetCurrentEntryCount | IMemoryCacheStats::GetHitCount | IMemoryCacheStats::GetMaxEntryCount | IMemoryCacheStats::GetMissCount

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
HTTP Client Reference
ATL Server Reference

Hotswap Sample: Replacing ATL Server Application DLLs at Run Time

Download sample

Demonstrates how to replace ATL Server Web application DLLs at run time without losing requests.

When running this sample, use your favorite Web stress tool to verify that no requests are lost. The most that should happen is a small delay while the swap actually occurs.

Note that this mechanism only works for simple application DLLs. It relies on the fact that the DLL can be unloaded if no threads from the ATL Server thread pool are within a call to **DispatchStencilCall**. If your application DLL increments its reference count by registering a service with the ISAPI extension (through **IServiceProvider**), creates a secondary thread pool, or uses asynchronous processing, extra work will be required in `CSwapExtension::StopRequests` to negotiate the proper shutdown of the application DLL.

Additionally, this sample does not restrict who is allowed to request a swap. When using this mechanism, it is highly recommended that some form of role-based access check be performed.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- IIS

Building and Running the Sample

To build and run this sample

1. Open the solution file, HotSwap.sln, in the Visual Studio development environment.
2. Build the solution. This will also deploy the solution to the local Web server.
3. Use a Web browser to view http://localhost/hotswap/hotswaptest.srf. You should see a simple greeting.
4. Edit the code in `OnHello` in hotswaptest.h to change the greeting.
5. Change the link settings for the HotswapTest project to output the DLL as hotswaptest.dll.new and rebuild that project. This will also deploy the new DLL to the local Web server.
6. Use a Web browser to view http://localhost/hotswap/hotswapadmin.srf.
7. Enter the full paths of the original and new hotswaptest.dll files. For example, they might be:
    - Target: C:\Inetpub\wwwroot\hotswap\hotswaptest.dll
    - Source: C:\Inetpub\wwwroot\hotswap\hotswaptest.dll.new
8. Click the **swap files** button on the page.
9. Use a Web browser to view http://localhost/hotswap/hotswaptest.srf. You should see the greeting modified in Step 4.

Note
You will probably need to modify the security settings of the DLLs in the file system or of the virtual directory to enable copying to take place.

Demonstrates

CDllCache::Flush | CIsapiExtension | CIsapiExtension::DispatchStencilCall | CIsapiExtension::GetExtensionVersion |

CIsapiExtension::m_DllCache | CIsapiExtension::m_StencilCache | CIsapiExtension::TerminateExtension | CStencilCache::RemoveAllStencils

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

ISAPIFilter Sample: Maps URLs to Parameterized Queries

Download sample

Demonstrates the creation of an ISAPI filter using ATL Server's support classes. The filter maps URL requests to new addresses using regular expression replacements provided in a configuration file.

ATL Server does not provide direct support for ISAPI filters; it only provides support for ISAPI extensions. There are many utility classes provided by ATL and ATL Server that make building a site-enhancing filter easier.

This sample is a name-mapping filter that translates one URL into another using regular expressions. The mappings are read from a file named friendly.cfg that must be in the same directory as the filter DLL. Each pair of lines is a regular expression to match against followed by a replacement to use if the regular expression is matched. Blank lines are ignored. The regular expressions are checked in the order that they appear in the file.

Note
The configuration file is loaded once when the filter is first loaded. Subsequent changes to the configuration file will not be a pparent in the filter until it is reloaded.

The configuration file provided with this example includes a single replacement that maps URLs of the form /hello/{\a*} to the following URL: /isapifilter/hello.dll?user=$0. This causes the part of the path following /hello/ to be passed to the request handler DLL in the query string.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustra te the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file, IsapiFilter.sln, in the Visual Studio development environment.
2. Build the solution. This will also deploy the solution to the local Web server.
3. Open Internet Services Manager. Click the **Start** button, point to **Programs**, point to **Administrative Tools**, and click **Internet Services Manager**.
4. Right-click the active Web site for port 80 (typically **Default Web Site**) and click **Properties** from the shortcut menu.
5. Click the **ISAPI Filters** property page and add \INetpub\wwwroot\ISAPIFilter\friendly.dll to the list of installed filters. Click **OK** to close the dialog box.

Note
This adds friendly.dll from the location where it is deployed.

6. Use a Web browser to view http://localhost/hello/*YourName* (replace *YourName* with your name). You should see a message greeting you. Observe that your Web site does not have a document or directory that corresponds to the URL you entered, so you can be sure that the filter has redirected the request to the request handler DLL.

Demonstrates

CAtlFile | CAtlFile::Create | CAtlFile::GetSize | CAtlFile::Read | CAtlRegExp | CAtlRegExp::Match | CAtlRegExp::Parse | CAtlREMatchContext | CAtlREMatchContext::GetMatch | CAtlREMatchContext::MatchGroup | CAutoPtr | CAutoPtr::Attach | CAutoPtrArray | CHeapPtr | CHeapPtr::Allocate | CPath | CPathA | CPathT::RemoveFileSpec | CStrBufT | CStringT | CStringT::Find

See Also

Other Resources

PooledAsync Sample: Increases Performance with an Additional Thread Pool

Download sample

Demonstrates how to increase Web application and SOAP service availability and performance with an additional thread pool.

The PooledAsync sample contains two closely related samples demonstrating how to increase Web application and SOAP service availability and performance by incorporating an additional thread pool.

By default, the ATL Server framework provides very high performance by using a thread pool in the ISAPI Extension to handle incoming requests. If any of these requests takes a long time to complete, however, the possibility arises that the application or service will become unresponsive to additional requests. This situation may occur when a high percentage of the threads in the primary pool (in the ISAPI Extension) are busy with long-running tasks (validating credit cards, calling another SOAP server, and so on).

The solution demonstrated in this sample is to implement an additional thread pool and use ATL Server's asynchronous features to offload these long-running tasks.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file, PooledAsync.sln, in the Visual Studio development environment.
2. Build the solution. This will also deploy the solution to the local Web server.
3. Use a Web browser to view http://localhost/pooledasync/index.htm. Use the links on the page to test the different ways that the Web application DLL provides for returning data to the client.
4. Open a command prompt in the folder that holds the SOAP client application, PooledAsyncSoapClient.exe, and run the application to test the ways that the XML Web service DLL provides for returning data to the client.

Demonstrates

AtlServerRequest::dwRequestState | CAtlMap | CComCriticalSection | CHttpResponse::AsyncPrep | CIsapiExtension | CSoapHandler::SoapFault | CSoapSocketClientT | CStreamOnServerContext | CThreadPool | CThreadPool::Initialize | CThreadPool::QueryInterface | CThreadPool::QueueRequest | CThreadPool::Shutdown | DECLARE_ASYNC_HANDLER | HTTP_SUCCESS_ASYNC | HTTP_SUCCESS_ASYNC_DONE | Worker Archetype | WorkerArchetype::Execute | WorkerArchetype::Initialize | WorkerArchetype::Terminate

See Also

Other Resources

ATL Server Samples

StencilCache Sample: Uses the ATL Server Stencil Cache

Download sample

Demonstrates how to use the stencil cache service in an ATL Server Web application.

The StencilCache sample illustrates how to:

- Get and set the lifetime of the cache entries.
- Use the statistics provided by the cache service.

The sample demonstrates the behavior of the stencil cache (also called the server response file cache) by writing a server response file to disk and changing it whenever the Touch button is clicked. This file is returned as part of the frameset displayed in the sample so that the cache is exercised. The statistics related to the cache are displayed in another page in the frameset.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- IIS

Building and Running the Sample

To build and run this sample

1. Open the solution file, SCSample.sln, in the Visual Studio development environment.
2. Build the solution. This will also deploy the solution to the local Web server.
3. Use a Web browser to view http://localhost/scsample/scsample.htm. Use the buttons on the page to test server response file caching.

Demonstrates

CHttpResponse::SetContentType | CHttpResponse::AppendCookie | CHttpRequest::Cookies | CCookie::GetValue | IServiceProvider::QueryService | IMemoryCacheStats Class | IMemoryCacheStats::GetMissCount | IMemoryCacheStats::GetHitCount | IMemoryCacheStats::GetCurrentAllocSize | IMemoryCacheStats::GetMaxAllocSize | IMemoryCacheStats::GetCurrentEntryCount | IMemoryCacheStats::GetMaxEntryCount | IStencilCache | IStencilCacheControl | IStencilCacheControl::GetDefaultLifeSpan | IStencilCacheControl::SetDefaultLifeSpan

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

SOAP Samples

The following topics are the abstracts for the ATL Server SOAP samples. These samples demonstrate XML Web services and clients accessible with SOAP and created using the support provided by ATL Server.

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

DataSetConsumer Sample: Consumes Multiple Table Datasets

Download sample

Demonstrates how to consume multiple table datasets from an unmanaged Visual C++ client using an XML Web service proxy.

The DataSetConsumer sample illustrates how an unmanaged Web application can consume data from a managed XML Web service. DataSetConsumer is an ATL Server application that consumes a multiple table dataset exposed through an XML Web service, WebService1, which you create using ASP.NET.

DataSetConsumer also shows how to use Web references to support the consumption of multiple table datasets. Web references convert single-table datasets into C++ structures but treat multiple-table datasets as XML streams. Therefore, the corresponding XML Web service method in the proxy class returns the full XML serialization of the dataset as string data. It must then be parsed to be useful in most applications.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- IIS must be installed and running.
- At least one SQL Server database must be installed.

Demonstrates

Adding and Removing Web References | Creating XML Web Service Clients | DataSet Class | request_handler | soap_handler | soap_method

Building and Running the Sample

To use this sample, follow the steps in this section. You will create an XML Web service project `WebService1` that defines a Web method to access data from a dataset. You then add a Web reference, which creates a proxy class for the XML Web service. DataSetConsumer uses an instance of this proxy to call the XML Web service method in `WebService1`. `WebService1` returns the dataset as type **BSTR**, and DataSetConsumer loads it as XML, using the DOM and the MSXML parser to parse the string and print the data as HTML.

To create an ASP.NET Web service project to access a dataset

1. Create a new XML Web service using ASP.NET.

   On the **File** menu, click **New**, and then click **Project**. The **New Project** dialog box appears. Click **Visual C# Projects** in the Project Types pane, and then click **ASP.NET Web Service** in the Templates pane. In the **Name** box, enter **WebService1**. Click **OK**.

Note
Note The location should be the default location (`http://localhost/WebService1`).

   When the project has been created, you will see Service1.asmx.cs in the Design pane.

2. Create a data connection.

   In Server Explorer, right-click **Data Connection**. On the shortcut menu, click **Add Connection**. The **Data Link Properties** dialog box appears. Complete it to create a data connection to a database, for example, the Northwind database.

3. Add tables and data adapters.

   Open the **Tables** node of your new data connection and drag two tables from Server Explorer to the Design pane. Two

DataAdapters and one data connection will appear in the Design pane.

4. Generate a dataset.

   On the **Data** menu, click **Generate Dataset**. The **Generate Dataset** dialog box appears. Choose a **New Dataset** named `DataSet1` (the default), and add both tables to the dataset. Also select the **Add this dataset to the designer** box (optional but recommended). Click **OK** to create the dataset (named `dataSet1` by default).

5. Implement a Web method.

   Right-click on the Service1.asmx.cs design surface in the Design pane. On the shortcut menu, click **View Code**. The code for Service1.asmx.cs appears in the Edit pane. Scroll to the bottom of the file to the `HelloWorld` example Web method:

```
//      [WebMethod]
//      public string HelloWorld()
//      {
//         return "Hello World";
//      }
```

   In place of the `HelloWorld` example code, implement a new Web method named `GetDataSet` as follows:

```
        [WebMethod]
        public DataSet1 GetDataSet()
        {
           this.sqlDataAdapter1.Fill (this.dataSet11);
           return dataSet11;
        }
```

Note
This code assumes that you use a SQL Server database.

6. Build the `WebService1` project. On the **Build** menu, click **Build Solution**.

To consume data from DataSetConsumer

1. Open the `DataSetConsumer` project.

   Open the solution file, DataSetConsumer.sln, in the Visual Studio development environment.

   Make sure that DataSetConsumer.h contains this line:

```
#include "Service1.h"
```

2. Add a Web reference to the project.

   Adding a Web reference will generate a proxy class for the XML Web service. An XML Web service client can call a method on the proxy, which in turn does the necessary work to remotely invoke the call over the network to call the corresponding XML Web service method. By default, the proxy class uses SOAP to access the corresponding method, because SOAP supports the richest set of data types of the three supported protocols (SOAP, HTTP-GET, and HTTP-POST). Adding a Web reference can generate proxy classes for the latter two protocols as well.

   In Solution Explorer, right-click the **DataSetConsumer** project node. On the shortcut menu, click **Add Web Reference**.

   The **Add Web Reference** dialog box appears.

   In the left pane of the **Add Web Reference** dialog box, click **Web References on Local Machine** and wait for the wizard to download the XML Web service files. When the files are finished downloading, a link and URL for each registered XML Web service will appear in the left pane. Click the link for the `Service1` Web service with the URL:

```
http://localhost/WebService1/Service1.asmx
```

The contents of the XML Web service's description file appears in the left pane. The Web service description is an XML document written in an XML grammar called Web Service Description Language (WSDL) that defines the format of messages for the XML Web service.

In the **Web reference name** box, enter **Service1**.

Click **Add Reference**. This adds the proxy class `Service1` to the `DataSetConsumer` project. The header file Service1.h appears in Solution Explorer.

Note
In Service1.h, the `CService1T` constructor is hardcoded to use the URL specified when you added the Web reference. If you want to use DataSetConsumer with another XML Web service, change the URL to match the location of that XML Web service's .asmx file.

3. Build the `DataSetConsumer` project. On the **Build** menu, click **Build Solution**.

4. Run the client application.

   On the **Debug** menu, click **Start Without Debugging**.

   When prompted for a URL, enter the following URL:

   ```
   http://localhost/DataSetConsumer/DataSetConsumer.srf
   ```

   Click **OK**. The Web browser appears, displaying the Northwind data accessed through the XML Web service.

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

OnlineAddressBook Sample: Stores an Address Book in an XML Web Service

Download sample

Demonstrates the use of session state to store an address book in an XML Web service created with ATL Server.

The OnlineAddressBook sample uses OLE DB client template classes to store a user's address book and lets the user set and retrieve this information using an XML Web service created with ATL Server.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- IIS
- Microsoft Office XP

Building and Running the Sample

Note
This sample requires Microsoft Office XP.

To build and run this sample

1. Open the solution file, OnlineAddressBook.sln, in the Visual Studio development environment.
2. Modify the OnlineAddressBookWS.disco file and the deployment project target computer name to point to the computer you are planning to deploy to. Alternately, you can run all of the projects without any modifications if you deploy and run everything to localhost.
3. Copy the included Microsoft Access database, OnlineAddressBookWS.mdb, to your C:\ drive. Alternatively, you can copy it to a different location as long as you modify `MYDATASOURCE`, which is defined in the file OnlineAddressBookWS\OnlineAddressBookWS.h.
4. Build OnlineAddressBookWSIsapi and OnlineAddressBookWS. This step will also create a virtual directory on your computer and deploy the relevant files there.
5. Build the included C# project, OnlineAddressBookCSharpClient. Set it as a startup project and run it so that you can use this client to import your current personal address book from Microsoft Outlook XP and upload it to your server or any server running this XML Web service. You can also create your own contacts and upload them to the XML Web service and later download them from anywhere.
6. Build OnlineAddressBookSRF. This will deploy an ATL Server project with a built-in XML Web service client. Once you have created an account and added some records to your address book with it using the C# client, you can use this project to view your contacts using a browser. This page should be in http://localhost/OnlineAddressBookWS/OnlineAddressBookSRF.srf.

Demonstrates

ISession::SetVariable | ISession::GetVariable | ISessionStateService::CloseSession | ISessionStateService::CreateNewSession | ISessionStateService::GetSession | CAccessorRowset::Close | GetDataSource | ISession | ISessionStateService | IDataSourceCache | soap_method | soap_handler | db_command

See Also

Other Resources

ATL Server Samples

ATL Server Concepts
ATL Server Reference

SecureSOAP Sample: Implements a Secure SOAP Communication (HTTPS)

Download sample

Demonstrates how to implement a secure SOAP communication.

The SecureSOAP sample shows how to implement a secure SOAP communication using the following:

- A custom SSL-enabled socket class as a template parameter for **CSoapSocketClientT**, based on the security support provided by Windows 2000
- The MSXML HTTP client (**CSoapMSXMLInetClient**)
- The WinInet-based HTTP client (**CSoapWininetClient**)

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- IIS
- SSL Certificate
- Windows 2000 or later

Building and Running the Sample

To build the sample

1. Open the solution file, SecureSOAP.sln, in the Visual Studio development environment.
2. At line 57 in SecureValidationClient.cpp, replace `<your_machine_name>` with the name of your Web server and remove the `#pragma message` directive at the line above.
3. Build the solution. This will build SecureValidation.dll, which is a combined DLL implementing the ISAPI interface, a request handler, and a SOAP handler (XML Web service).
4. Deploy the solution manually:
    a. Create a virtual folder within IIS.
    b. Map a directory to this virtual folder.
    c. Copy SecureValidation.dll and SecureValidation.srf to that directory.
    d. Configure the "dll" and "srf" extensions to be associated with the SecureValidation.dll ISAPI extension DLL.

To set up SSL on your Web server

1. If you have IIS 5, see http://localhost/iisHelp/iis/misc/default.asp and navigate in the Contents window to Administration->Server Administration->Security->Certificates->Setting Up SSL on Your Server.
2. See certReadme.txt in the SecureSOAP sample folder. This file contains a step-by-step tutorial on how to generate and install digital certificates.

To run the sample

1. Run SecureValidationClient.exe.

2. To check the Web page, use the following URL:

```
https://<your_web_server>/<your_virtual_path>/SecureValidation.srf
```

The Web page will display output showing whether communication succeeded or failed.

Demonstrates

CSoapSocketClientT | CSoapMSXMLInetClient | CSOAPWininetClient | CSoapRootHandler::Cleanup | CSoapSocketClientT::SetUrl

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

SOAPDataTypes Sample: Demonstrates Basic Data Types and Aggregated Types in a SOAP Server

Download sample

Demonstrates the use of basic data types and aggregated types in a SOAP server.

The SOAPDataTypes sample transmits the following types through the ATL Server SOAP implementation:

- Basic types
- Structures
- Fixed-size and dynamic-size arrays, both single dimensional and multidimensional
- Arrays of structures
- Embedded structures
- Embedded arrays

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- IIS

Building and Running the Sample

To build and run this sample

1. Open the solution file, SOAPDataTypes.sln, in the Visual Studio development environment.
2. Build the solution. This will deploy the XML Web service to IIS.
3. Run the SoapDataTypesClient executable file.

The SoapDataTypesClient executable file displays the results of each SOAP data type as it is passed from the client to the server and back.

Demonstrates

ATLSOAP_BLOB | request_handler | soap_handler | soap_method | ATL Server XML Web Service Supported Types

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference
XML Web Services Created with ATL Server

SOAPDebugApp Sample: Debugging XML Web Services in the Client Memory Space

Download sample

Demonstrates how to debug an XML Web service created with ATL Server in the memory space of the client application.

The SOAPDebugApp sample illustrates how to use the following:

- TClient template parameter of the Sproxy-generated client class
- **IHttpServerContext** interface

The sample shows a simple way of testing/debugging XML Web services in the memory space of the client application, that is, without involving the Web server.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution file, SoapDebugApp.sln, in the Visual Studio development environment.
2. Build the solution.

Note
This will build SimpleSoapApp.dll, an XML Web service application DLL implementing a simple HelloWorld SOAP method, and demoApp.exe, a small client application using **CSoapDebugClient**. No Web deployment is needed to run this sample.

3. Set a breakpoint at the start of the HelloWorld method of the XML Web service DLL (the simpleSoapApp.h file of the SimpleSoapApp project).
4. Make sure demoApp is selected as the startup project, and then press F5 to start the client application in debug mode. The breakpoint should be hit.

How the Sample Works

The sample provides a class (`CSoapDebugClient`) to act as the TClient parameter for the Sproxy-generated client class. The application DLL (the SOAP server) uses an **IHttpServerContext** interface to read the request and to write the response. Instead of sending the request over HTTP, `CSoapDebugClient` loads the application DLL (the SOAP server) and then emulates the behavior of a real HTTP server by implementing the **IHttpServerContext** interface.

Demonstrates

IHttpServerContext | CAtlFile

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference
XML Web Services Created with ATL Server

SOAPState Sample: Uses ATL Server to Implement Persistent SOAP Servers

Download sample

Demonstrates how to implement a persistent SOAP server.

The SOAPState sample shows how to use the following:

- **IServiceProvider** interface implemented by **CIsapiExtension**
- SOAP headers

The sample shows a method of loading the state of a SOAP server before a method invocation and of saving the state immediately after the invocation. The state of the SOAP server is preserved in memory. The code can be easily modified to store the state in a database or a file. A specific service is implemented in the **CIsapiExtension**-based class of the ISAPI DLL project to allow serialization of the SOAP servers.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- IIS

Building and Running the Sample

To build and run this sample

1. Open the solution file, SoapState.sln, in the Visual Studio development environment.
2. Build the solution.

Note
This will build SStateIsapi.dll, an ISAPI extension containing the ISOAPSrvStorageService, and SState.dll, an example of a persistent SOAP server using the ISOAPSrvStorageService. It will also build demoApp.exe, a small client application showing how to access a persistent SOAP server.

3. Run the console application demoApp.exe. You will see a series of output messages relating the communication with the server.

If you map the virtual folder under a different name or on a different computer, you will need either to regenerate the demoApp proxy file (simpleSoapApProxy.h) or add the following to the demoApp.cpp file, before executing any SOAP invocation:

```
srv.SetUrl(
   "http://< machine_name>/<folder_name>/SState.dll?Handler=Default");
```

How the Sample Works

Each persistent SOAP server has to implement a specific interface. This interface (`IPersistSoapServer`) contains two sets of methods. (Implementation of these methods should not be different from the implementation provided with the sample.)

Methods Exposed Through SOAP

The first set of methods is exposed through SOAP:

```
[id(1)]HRESULT initPersistSoapServer([in]BSTR bstrUser, [in]BSTR bstrPwd, [out, retval]eSta
te* state);
[id(2)]HRESULT destroyPersistSoapServer([in]BSTR bstrUser, [in]BSTR bstrPwd);
[id(3)]HRESULT setPersistSoapServerTimeout([in]DWORD dwTimeoutSecs);
```

Only the first method, `initPersistSoapServer`, must be implemented and called to have persistence. If the client application does not call `destroyPersistSoapServer`, the persistent object will be destroyed at timeout. If the client application does not call `setPersistSoapServerTimeout`, a default timeout value will be assigned.

The `initPersistSoapServer` method generates a unique value for a SOAP header defined as:

```
[soap_header(value="m_bstrStorageKey")]
```

It takes the default parameters, `in="true"` and `out="true"`, that will be applied to each SOAP method, so it will be sent both ways in each method invocation. Whenever a client application will invoke a SOAP method, the `m_bstrStorageKey` header will be part of the invocation. If, during the lifetime of one persistent SOAP server, two client applications use the same `bstrUser` and `bstrPwd` parameters for the `initPersistentSoapServer` method, the two applications will be connected to the same instance of the persistent object. If the parameters are different, different instances are created on the server side. If one client application invokes the initialization method with NULL as `bstrUser` and `bstrPwd`, then the persistent object created cannot be identified by the other application. It will be accessible only for the client that created it.

Once a SOAP invocation occurs, the following should appear in any SOAP method to ensure the persistent state is loaded:

```
CPersistentHandler persistHandler(static_cast<IPersistSoapServer*>(this), m_spServiceProvid
er);
```

The `CPersistentHandler` object resides on the stack, so it will be destroyed when the method ends. On destruction, it will take care of saving the state of the server.

Methods Not Exposed Through SOAP

In the constructor/destructor of the `CPersistentHandler` object, the second set of methods in `IPersistSoapServer`, not exposed through SOAP, are used:

```
HRESULT getObjToken(storageKey& token);
HRESULT setObjToken(storageKey& token);
HRESULT persist_load(const ATLSOAP_BLOB &data);
HRESULT persist_dump(ATLSOAP_BLOB &data);
```

- `getObjToken` is called while attempting to restore the state of the object, and it is supposed to return the value of the `m_bstrStorageKey` header to act as a storage key for the state of this server.
- `persist_load` is also called while attempting to restore the state of the object, to load the state of the server from the saved binary BLOB.
- `persist_dump` is called while attempting to save the state of the object, to serialize the object in the provided binary BLOB.
- `setObjToken` is called only during `initPersistSoapServer`, to set the value of the `m_bstrStorageKey` for the communication.

Internally, the `CPersistentHandler` object connects to `ISOAPSrvStorageService` (defined in the persist.h file of the sample) provided by the ISAPI extension and simply sends binary BLOBs for storage under a specific key or retrieves the BLOBs based on the same key.

Although this sample stores the BLOBs in memory by modifying the implementation of `ISOAPSrvStorageService`, the SOAP server state can also be preserved in a database or a file.

Demonstrates

CIsapiExtension::QueryService | soap_header | soap_method

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

SOAPTransport Sample: Communicates SOAP Messages Over Sockets, MSMQ, the File System, and HTTP Listener

Download sample

Demonstrates the creation of SOAP servers and clients that communicate using different transports: sockets, Microsoft Message Queue, the file system, and a custom HTTP listener.

The SOAPTransport sample consists of eight projects: a SOAP server and a client for each of the four supported transports.

The sample shows how to separate the SOAP support offered by ATL Server from the ISAPI framework, allowing use of ATL Server to build SOAP servers and clients over virtually any communication channel.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- MSMQ

Building and Running the Sample

To build and run this sample

1. Open the solution file, SoapTransport.sln, in the Visual Studio development environment.
2. Build the solution.

Note
This will build all the client and server applications.

3. Run FloppyTransport:
   a. Start the FloppyClient application. It will ask for the name of the file to save the request to. Enter a file name.
   b. Start the FloppyServer application. It will ask for the file containing the request. Enter the file where the request was saved.
   c. The server application will ask for a file to save the response to. Enter a file name.
   d. Return to the FloppyClient running instance and press any key. It will ask for the file containing the response. Enter the file where the server saved the SOAP response.

      The result of the SOAP invocation is displayed.
4. Run TCPIPTransport:
   a. Start the TCPServer application. It will launch the TCP/IP listener.
   b. Start the TCPClient application. It will send the request and then display the result of the invocation.
5. Run MSMQTransport (you will need the Message Queuing Services installed on your computer):
   a. Start the MQServer application. It will create a message queue and wait for a message containing a SOAP request to be posted.
   b. Start the MQClient application. It will post the request and then display the result of the invocation.
6. Run HTTPListenerTransport:
   a. Start the HTTPListenerServer application. It will wait for a message containing a SOAP request to be posted.

b. Start the HTTPListenerCSharpClient application. It will send the request and then display the result of the invocation.

How the Sample Works

The following discussion covers the server, client, and transports.

Server Side

The ATL Server Web Service application wizard will generate the code for an XML Web service exposed through HTTP. The XML Web service is implemented as a class with the following declaration:

```
[
request_handler(name="Default", sdl="GenSimpleSoapAppServiceWSDL"),
soap_handler(
      name="SimpleSoapAppService",
      namespace="urn:SimpleSoapAppService",
      protocol="soap"
   )
]
class CSimpleSoapAppService :
   public ISimpleSoapAppService
```

In this code sequence, the attributes before the class declaration do most of the work. The `soap_handler` attribute will make the class a `CSoapHandler<CSimpleSoapAppService>` derivative (that is, an XML Web service) able to handle SOAP messages and map them to internal method calls, and also to wrap the result of internal method invocations to SOAP responses. The `request_handler` attribute will make it able to handle HTTP requests.

If the transport layer is not going to be HTTP, the `request_handler` attribute is no longer needed, so it can be commented out.

The `CSoapHandler<>` class in atlsoap.h is designed to handle SOAP requests through HTTP. The entry point for handling an HTTP SOAP request is:

```
HTTP_CODE HandleRequest(AtlServerRequest *, IServiceProvider *)
```

The request body and the SOAP-related HTTP header ("SOAPAction") are retrieved from the ISAPI extension through the `pServerRequest` (an ECB wrapper) member of the `AtlServerRequest*` parameter.

The `pServerRequest` object provides methods for reading the request body, reading the SOAPAction HTTP header, and writing the SOAP response back to the client. As this sample intends to replace the HTTP transport, it needs to launch the SOAP request processing and to provide it with a way of performing the same operations.

This is solved by creating a `CSoapHandler` derivative, `CSoapTransportHandler`, that implements the following entry point (in file SoapTransportSrv.h of the sample):

```
HTTP_CODE InvokeSoapMethod(stSoapTransportDescription   *pTransInfo)
```

The `stSoapTransportDescription` is defined as:

```
struct  stSoapTransportDescription
{
// the stream to write the SOAP response to
   IWriteStream   *pWriteStream;
// the stream to read the SOAP request from
   IStream      *pReadStream;
// the SOAP Action
CStringA         strSOAPAction;
// the service provider
   IServiceProvider   *pServiceProvider;
};
```

Perhaps the only member that needs explanation is `pServiceProvider`. It allows the SOAP servers to share services exactly as those in an ISAPI DLL. It can be NULL in an implementation, as long as the SOAP servers will not attempt to use it. One of the services commonly exposed through the service provider is the MSXML Reader.

Now, the SOAP server class generated by the wizard has to be modified to use the new `CSoapTransportHandler` functionality. The new code would look like this:

```
[
      // request_handler(name="Default", sdl= "GenSimpleSoapAppWSDL"),
   soap_handler(
      name="SimpleSoapAppService",
      namespace="urn:SimpleSoapAppService",
      protocol="soap"
   )
]
class CSimpleSoapAppService :
   public CSoapTransportHandler<CSimpleSoapAppService>,
   public ISimpleSoapAppService
```

No other modifications need to be done in the server code.

At this point, the SOAP server can be instantiated and the SOAP methods can be invoked through `CSoapTransportHandler::InvokeSoapMethod`. Using custom read and write streams, there is no longer a dependency on ATL Server HTTP support.

The instantiation of the server or servers is handled by another class, `CSoapDispatcher`, implemented in the soapDispatch.h file of the sample.

At this point, a server application only has to:

- Get the SOAP request and the SOAP Action header through any transport channel.
- Wrap the request in an `IStream` interface.
- Provide a stream to store the response.
- Invoke `CSoapDispatcher::DispatchCall`.
- Send the content of the write stream (the SOAP response) over any transport channel to the client.

Client Side

The template-based code generated by the sproxy.exe tool for an ATL Server SOAP client makes for easy overriding of the transport on the client side. The SOAP proxy class generated by sproxy.exe has the following prototype:

```
template <typename TClient = CSoapSocketClientT<> >
class CSimpleSoapAppServiceT
```

The methods that have to be implemented by the `TClient` template parameter are well documented in the ATL Server Reference. The method that is really of importance here is `SendRequest`. That method is supposed to send the request from a stream to the SOAP server and to fill a different stream with the server response.

With a custom class that contains all of the methods required to be a `TClient`, the actual channel used to send the request to the SOAP server only affects the implementation of the `SendRequest` method.

Transport Channels

The sample implements the following transport channels:

- TCP/IP communication
- MSMQ communication
- Communication through files

Each transport channel implementation contains a client application and a server application. The server uses the SOAP server included in the "Include" folder (which is built as described above). The code in the server application only takes care of the transport. The client application only implements the transport-specific class to act as a TClient parameter for the Sproxy.exe-generated class.

A limitation of the sample is that the WSDL used in generating the proxy has to be generated outside of the sample. The WSDL used in the applications specified above is saved as simpleSoapSrv.wsdl in the "Include" folder. The solution for this would be

to create the SOAP server through the wizard, complete the interface and then save the WSDL through the default HTTP way provided by ATL Server.

An Extension: HTTPListenerTransport

This folder contains an enhanced version of the TCPIPTransport server. This server listens on a specified TCPIP port, but it accepts HTTP requests. It provides a way of responding to .disco requests, to generate the WSDL for the implemented SOAP servers and also to accept SOAP invocations. This server can provide a very lightweight, complete, SOAP implementation without requiring an HTTP server to be installed on the computer.

To create a client for this lightweight server, use the Add Web Reference wizard from the Visual Studio IDE and point the wizard to the following URL:

http://<your_machine>:<ListenPort>/disco

The client provided for this server (**HttpListenerCSharpClient**) is generated this way.

Demonstrates

CSoapHandler | CSoapHandler::SoapFault | CSoapRootHandler::GenerateResponse | CSoapHandler::SetHttpError | CSoapRootHandler::InitializeSOAP | CSoapRootHandler::BeginParse | CSoapRootHandler::CallFunctionInternal | CSoapRootHandler::UninitializeSOAP | soap_handler | request_handler | CAtlIsapiBuffer

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

WeatherService Sample: An XML Web Service Created with ATL Server

Download sample

Demonstrates implementing a simple XML Web service using ATL Server attributes and accessing an XML Web service from an MFC-based client application. This sample also shows layered window support available in Windows 2000 and later.

The WeatherService sample allows access to weather conditions in a variety of cities. The client application demonstrates layered window support, so it requires Windows 2000 or later to run. For best results, set your color depth to 16-bit or better.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

- IIS (client and server)
- Windows 2000 or later (MFC client only)

Building and Running the Sample

To build and run this sample

1. Open the web service solution WeatherService.sln in the Visual Studio development environment.
2. Build the solution. This will deploy the solution to the local Web server.
3. Open the client solution WeatherServiceClient\wcli.sln.
4. Add a web reference from the client to the web service:
    - Right-click on the **WeatherServiceClient** project in the **Solution Explorer** and select **Add Web Reference...**.
    - Click the **Web services on the local machine** link. A list of web services on your machine should appear.
    - Click on the **WeatherService** link.
    - Change the web reference name from **localhost** to **WeatherService** and click **Add Reference**.
5. Build the client solution.
6. Run the client executable file and use the UI to access the weather service.

Demonstrates

soap_handler | request_handler | soap_method | export | Layered Windows

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

Tools Samples

The following topics are the abstracts for the ATL Server tools samples. These tools are useful applications that can help you debug and work with your own ATL Server projects. These tools are shipped as binaries with Visual Studio. They can also be built from the source code provided.

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

CLStencil Sample: Runs Request Handler DLLs or SRF Files from the Command Line

Download sample

Demonstrates how to provide custom implementations of IHttpServerContext and IIsapiExtension to create a command-line tool for generating responses from SRF files and Web application DLLs without requiring IIS.

Note that CLStencil only provides a partial implementation of **IHttpServerContext** and **CIsapiExtension**. Items that are specific to a Web server, such as IHttpServerContext::MapUrlToPathEx, are not supported.

CLStencil tries to locate server variables in the current environment, so if the request handler needs a specific server variable, be sure to set it in the environment before running the application. For example, you can run the following command at a command prompt to make the SERVER_PROTOCOL variable available to CLStencil and the request handlers it loads:

SET SERVER_PROTOCOL=HTTP/1.1.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

It is not necessary to build this sample to see the tool in action. The executable file (clstencil.exe) is shipped with Visual Studio in the Vc7\bin directory. Instructions for use can be found below:

To use CLStencil

- Run clstencil.exe from the command line using the following command:

  clstencil i *SrfFile* **o** *OutputFile* **e** *ErrorFile*

  Where:

  SrfFile The name of the SRF file to be processed.

  OutputFile The name of the file to which the generated content will be written.

  ErrorFile The name of the file to which warning and error messages will be written.

  Run clstencil.exe with no arguments for more information on its command line arguments.

Demonstrates

IHttpServerContext | IIsapiExtension | AtlServerRequest

See Also

Other Resources

ATL Server Samples
ATL Server Concepts
ATL Server Reference

WebDbg Sample: A Client for Debugging Server Applications Using CDebugReportHook

Download sample

Demonstrates client code for CDebugReportHook in the form of an MFC application providing a user interface for debugging server applications.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

It is not necessary to build this sample to see the tool in action. The executable file (WebDbg.exe) is shipped with Visual Studio in the Common7\Tools directory. Instructions for use can be found in the topics on Viewing Trace Messages and Handling Asserts and CDebugReportHook.

See Also

Reference

CDebugReportHook Class

Concepts

Viewing Trace Messages And Handling Asserts

Other Resources

ATL Server Samples

ATL Server Concepts

ATL Server Reference

Tutorial Sample

Download sample

This sample provides the code for the ATL Server Tutorial.

Demonstrates Web access to a database for a simple online store. When you run the application, a Web page prompts you for a username and password. You can either enter a username and password for an existing customer in the database or create a new account, in which case the application will write the new username and password to the database.

Building and Running the Sample

To build and run this sample

1. Open the solution file, Tutorial.sln, in the Visual Studio development environment.
2. Build the solution. This will also deploy the solution to the local Web server.
3. Configure the data link to connect to the Tutorial.mdb database. In the sample's Tutorial\Tutorial\bin directory, double-click Tutorial.udl.

   The **Data Link Properties** dialog box appears.
4. On the **Provider** tab, select **Microsoft Jet 4.0 OLE DB Provider**.
5. On the **Connection** tab, click the Ellipsis (...) button to specify the Tutorial.mdb file in the Tutorial\Tutorial\bin directory. Leave the username and password blank.
6. Ensure that Tutorial.mdb is not read-only. Right-click the file and select **Properties**. On the **General** tab, ensure that the **Read-only** check box is cleared.
7. Change the NTFS security settings on the Tutorial.mdb file to allow read-write access by the Internet Guest Account, **IUSR**_*Machine*. To do this:

   a. Right-click the Tutorial.mdb file and select **Properties** on the shortcut menu.

   b. In the **Properties** dialog box, select the **Security** tab and click **Add**.

   c. The **Select Users or Groups** dialog box appears; in that dialog box, set the following:

      Object Types to **Users**.

      Locations to the local machine.

      Check Names to **IUSR**_*Machine*.

   d. Select the Internet Guest Account that you just created. In the **Permissions** box, select the **Read** and **Write** boxes if they are not already selected.

Note
See ATL Server Security for information about the accounts used by a typical ATL Server application running in IIS.

8. Change the NTFS security settings on the folder containing Tutorial.mdb to allow only write access by **IUSR**_*Machine*, using the procedure described in Step 7.
9. Change the NTFS security settings on the Tutorial.udl file to allow read access by the **IUSR**_*Machine* account, using the procedure described in Step 7.
10. Use a Web browser to view http://localhost/tutorial/login.srf.

For more information, see ATL Server Tutorial.

See Also

Other Resources

ATL Server Samples

ATL Server Concepts
ATL Server Reference

Compiler COM Support Samples

The samples in this section demonstrate the Visual C++ compiler's built-in support for COM. For more information on the COM support classes and compiler features included with Visual C++, see Compiler COM Support.

In This Section

ACDUAL Sample: Adds Dual Interfaces to an Automation Application (C++ Samples)

Adds dual interfaces to an Automation application.

ADOSAMP Sample: Implements a Three-Tier Client/Server Application

Implements a three-tier client/server application.

ALLINONE Sample: Implements a Server Using ATL, Exposing STL Collections, Controlled by Compiler COM Support in an MFC Application

Implements a server using ATL, exposing STL collections, and controlled by compiler COM support in an MFC application.

COMMAP Sample: Demonstrates COM Interface Map Entry Macros with Compiler COM Support

Demonstrates COM interface map entry macros with compiler COM support.

CONNECT Sample: Demonstrates Implementation and Use of Connection Points (C++ Samples)

Illustrates the use and implementation of connection points (the **IConnectionPointContainer** and **IConnectionPoint** interfaces) in a multithreaded environment.

DCOM Sample: Calls COM Object in Windows NT Service from Multiple Clients

Demonstrates how to call a COM object implemented in a Windows service from multiple clients running on different computers.

INPROC Sample: Demonstrates an In-Process Automation Server Application (C++ Samples)

Demonstrates an in-process Automation server application with compiler COM support.

LABRADOR Sample: Implements a Server with an Out-of-Proc Custom Interface and Compiler COM Support

Implements an EXE server that does not have any user interface.

MFCCALC Sample: Demonstrates an Automation Server Application (C++ Samples)

Demonstrates an Automation server application with compiler COM support.

See Also

Other Resources

Visual C++ Samples

ACDUAL Sample: Adds Dual Interfaces to an Automation Application (C++ Samples)

Download sample

ACDUAL demonstrates how to add dual-interface support to an MFC-based Automation application with the native compiler COM support. See the MFC ACDUAL sample for more information.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution acdual.sln.
2. On the **Build** menu, click **Build Solution**.
3. Set acdual as the startup project (right-click the project node and click **Set as StartUp Project**) and click the **Start** button. Acdual will register itself (otherwise the client project couldn't run). Close the acdual application before continuing.
4. Set autodriv as the startup project and run it. Use the **AutoClik Test Drive** dialog box to modify the output in the **ACDual AClick** window.

Comparison with the MFC Version

The differences between this sample and the MFC sample with the same name are:

- The **COleDispatchDriver** class is no longer necessary. Instead, **#import** is used to import a type library.
- The **_bstr_t** and **_variant_t** wrapper classes are used to simplify operations of **BSTR** and **VARIANT** types.
- **__declspec(property)** is used to simplify assignment operations for properties of COM objects.
- COM error handling is now performed by using the **_com_error** class.

With the Visual C++ native compiler COM support, the resulting sample code is shorter and more efficient. The MFC-based ACDUAL sample uses both dual interfaces and dispinterfaces only. Compare this sample with the MFC version to see the differences in source code.

Keywords

This sample demonstrates the following keywords:

dispinterface; #import; _com_ptr_t; _variant_t; _bstr_t; _com_error

See Also

Reference

Compiler COM Support

Other Resources

Compiler COM Support Samples

ACDUAL Sample: Adds Dual Interfaces to an Automation Application

ADOSAMP Sample: Implements a Three-Tier Client/Server Application

Download sample

This sample implements a three-tier client/server application using ADO, ATL, and compiler COM support.

The sample demonstrates how you can use compiler COM support with ADO to create a client/server database application. The application creates a small database of information about motorcycles and their top speeds. The application inserts rows, deletes rows, adds a new column, and searches on top speeds greater than a specified value.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. In the Control Panel, open the Administrative Tools folder and then double-click Data Sources (ODBC). If you are using either Microsoft Access or SQLServer with this sample, add a (User) DSN with a **Data Source Name** "HotBikes." If you are using Microsoft Access, create a database called Test.
2. For SQLServer, provide the server name and under **Options** specify a **Database Name**, for example, HotBikesDB.
3. Open the solution ADOSamp.sln.
4. Add the directory containing MsADO15.dll to the Include files path in the VC++ Directories dialog box.
5. If either OLE DB or ADO has not been installed, install it from the Visual Studio CD or DVD. It will most likely be installed in:

   \Program Files\Common Files\System\ADO\MsADO15.dll
6. If you created a SQL Server data source for this sample in step 1, uncomment the following line in RunADO.cpp:

   ```
   //#define SQLSERVER
   ```

   This disables a SQL command that Microsoft Access can understand but SQLServer cannot:

   `"ALTER TABLE HotBikes DROP COLUMN ID"`
7. On the **Build** menu, click **Build Solution**.
8. On the Debug menu, click **Start Without Debugging**. A table will be created (deleting the existing one), and several rows will be added, deleted, and updated. A new column will be added and populated with data. In the case of Microsoft Access, a column will be deleted. The contents of the table will be displayed after each modification.

See Also

Reference

Compiler COM Support

Other Resources

Compiler COM Support Samples

ALLINONE Sample: Implements a Server Using ATL, Exposing STL Collections, Controlled by Compiler COM Support in an MFC Application

Download sample

This sample demonstrates the ability of ATL, STL, Visual C++ compiler COM support, and MFC to coexist in a single application. Building on the COLLECT MFC sample, nine collections are implemented in STL by using map, multimap, list, vector, deque, and set. They are exposed as custom interfaces by using ATL with multiple types of aggregation. The MFC controller uses compiler COM support.

Compare this sample with the COLLECT MFC sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution AllInOne.sln.
2. On the **Build** menu, click **Build Solution**.
3. Press F5 to run the sample.
4. In the sample's user interface, select and use a collection class from the **Example** menu.

Keywords

This sample demonstrates the following keywords:

map, vector, set, multimap, list, deque, _com_ptr_t, _bstr_t, _com_error

See Also

Reference

Compiler COM Support

Other Resources

Compiler COM Support Samples

COMMAP Sample: Demonstrates COM Interface Map Entry Macros with Compiler COM Support

Download sample

This basic sample shows how different COM interface map entry macros are used. This sample uses the smart pointer mechanism (**_com_ptr_t**) of Visual C++ native compiler COM support. The **#import** mechanism is used instead of the MIDL-generated .h and .c files to provide marshaling information.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution commap.sln.
2. On the **Build** menu, click **Build Solution**.
3. Set ctlcomm as the startup project (right-click the project node and click **Set as StartUp Project**) and press F5.
4. Open Commap.htm in your Web browser. It includes comments about each type of map entry.

CTLCOMM is an MFC-wrapped Visual C++ compiler COM support controller of the COMMAP ATL server.

Keywords

This sample demonstrates the following keywords:

dispinterface; #import; _com_ptr_t; _variant_t; _bstr_t; _com_error

See Also

Reference

Compiler COM Support

Other Resources

Compiler COM Support Samples

COMMAP Sample: Demonstrates COM Interface Map Entry Macros with Compiler COM Support

CONNECT Sample: Demonstrates Implementation and Use of Connection Points (C++ Samples)

Download sample

This sample illustrates the use and implementation of connection points (the **IConnectionPointContainer** and **IConnectionPoint** interfaces) in a multithreaded environment.

Note
This sample is a modification of the ATL sample Connect to demonstrate compiler COM support for connection point sinks. The Drive client is a native implementation, whereas the MDrive client slightly belabors COM-correctness in using a class factory to create sink objects. Neither Drive nor MDrive depend on ATL or MFC for any COM support.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution connect.sln.
2. On the **Build** menu, click **Build Solution**.
3. Select which client you want to run, Drive or MDrive, and make it the startup project (right-click the project folder and click **Set as StartUp Project**). More information about the clients can be found in the How the Sample Works section.
4. On the **Debug** menu, click **Start Without Debugging**.

How the Sample Works

The server is implemented in Connect.dll. This DLL allows the creation of a `CoRandom` COM object, implemented by the `CRandom` C++ class. The COM object supports `IRandom` (a dual interface) and **IConnectionPointContainer**, and it accepts connections for the `IRandomEvent` interface.

The `IRandom` interface supports the following methods.

- `Start` Starts a thread inside the object.
- `Stop` Stops a thread inside the object.
- `StopAll` Stops all running threads.

When running, the secondary threads inside the object keep firing events through the connection point.

Two clients are provided: Drive and MDrive. They can be found in the Drive and MDrive subdirectories.

- Drive.exe is a simple console application that provides a single object implementing the `IRandomEvent` interface. It creates a `CoRandom` object, calls `Advise` and `Unadvise` on the connection point, and has the `CoRandom` object fire events into the drive's object.
- Mdrive.exe is an MFC dialog-based application, able to create multiple advise sinks and to control the number of threads the server creates. When you run Mdrive.exe, click the **Start** button at least once, then click the **Advise** button several

times. Each click of the **Advise** button adds a connection point, which makes the display wider. If you do not click the **Advise** button, you will not see any activity in the display.

See Also

Reference

Compiler COM Support

Other Resources

Compiler COM Support Samples

DCOM Sample: Calls COM Object in Windows NT Service from Multiple Clients

Download sample

The DCOM sample demonstrates how to call a COM object implemented in a Windows service from multiple clients running on different computers. It is composed of several parts:

- **DrawServ** The Windows service that implements the COM object.
- **ATLDraw** The client that connects to the DrawServ COM object.
- **DrawCtl** A control version of ATLDraw. It has a Connect method (that takes a machine name string parameter), a Disconnect method, and a Clear method.
- **MFCCont** An MFC container for DrawCtl. Enter the name of the (possibly remote) server without quotes or slashes.
- **ATLCont** An ATL container for DrawCtl. Enter the name of the (possibly remote) server without quotes or slashes.

All computers must be running Windows NT 4.0 or later for this sample to work.

Compare this sample with the ATL DCOM sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution dcom.sln.
2. Build the DrawServ and ATLDraw samples.
3. Copy Drawserv.exe and Atldraw.exe to each computer that you want to run the sample from. Register the server on each computer by running DrawServ with the command-line argument /RegServer or -RegServer (case-insensitive). For example:

   ```
   DrawServ /RegServer
   ```

   (Register.dll must be registered for this to work.) The reason you need to copy the server to each client is that the CLSID for the server and the server type library need to be registered on the client.
4. Start the service on the server by using the Services icon in Control Panel.
5. Using DCOMCNFG on each client, select the Properties of the DrawServ Class. From the **Location** tab, select **Run application on the following computer**, then enter the name of the computer on which you are running the server object. Clear the **Run application on this computer** check box.
6. Run ATLDraw and select Server. Connect from the menu on each client. Draw on the client window by holding the left mouse button down and dragging a line. The drawn line should appear on each client that is also connected to the same server. You can also use the **Color** command from the **View** menu to change the color for each client.
7. If there is any problem connecting to a remote server (for example, if Tstcon32.exe does not call CoInitializeSecurity), run

DCOMCNFG on the client and open the tab **Default Properties**. Set **Default Authentication Level** to None and **Default Impersonation Level** to Anonymous.

See Also

Reference

Compiler COM Support

Other Resources

Compiler COM Support Samples

DCOM Sample: Demonstrates Remotely Calling a COM Object

INPROC Sample: Demonstrates an In-Process Automation Server Application (C++ Samples)

Download sample

INPROC is an in-process Automation server. Unlike the other Automation server samples provided, INPROC can be loaded as a dynamic-link library (DLL) in the client's address space. This sample uses the native compiler COM support. See the MFC INPROC sample for more information.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution inproc.sln.
2. On the **Build** menu, click **Build Solution**.
3. Register the server:
   a. Change the path in server\inproc.reg to the path of the inproc.dll you just built.
   b. Run `regedit server\inproc.reg`.
4. Set the ipdrive project as the startup project (right-click the project node and click **Set as StartUp Project**) and run this application.

Comparison with the MFC Version

The differences between this sample and the MFC sample with the same name are:

- The **COleDispatchDriver** class is no longer necessary. Instead, **#import** is used to import a type library.
- The **_bstr_t** and **_variant_t** wrapper classes are used to simplify operations of **BSTR** and **VARIANT** types.
- **__declspec(property)** is used to simplify assignment operations for properties of COM objects.
- COM error handling is now performed by using the **_com_error** class.

With the Visual C++ native compiler COM support, the resulting sample code is shorter and more efficient. The MFC-based INPROC sample uses dispinterfaces only. Compare this sample with the MFC version to see the differences in source code.

Keywords

This sample demonstrates the following keywords:

dispinterface; #import; _com_ptr_t; _variant_t; _bstr_t; _com_error

See Also

Reference

Compiler COM Support

Other Resources

Compiler COM Support Samples

INPROC Sample: Demonstrates an In-Process Automation Server Application

LABRADOR Sample: Implements a Server with an Out-of-Proc Custom Interface and Compiler COM Support

Download sample

This sample shows how to use ATL to implement an EXE server. The server allows creation of an object that supports two custom interfaces, defined in Labrador.idl. Compiler COM support is used to implement custom interfaces that perform nontrivial marshaling of a C-style array.

Compare this sample with the LABRADOR sample in the ATL directory.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution labrador.sln.
2. On the **Build** menu, click **Build Solution**. Note that a compiler warning will occur because this sample uses the obsolete ATL class **CComModule**. Refer to the ATL version of this sample to see how to use CAtlExeModule.
3. Set labdriv as the startup project (right-click the project node and click **Set as StartUp Project**).
4. On the **Debug** menu, click **Start Without Debugging**.

Keywords

This sample demonstrates the following keywords:

#import; _com_ptr_t; _com_error

See Also

Reference

Compiler COM Support

Other Resources

Compiler COM Support Samples

LABRADOR Sample: Implements a Server with No User Interface

MFCCALC Sample: Demonstrates an Automation Server Application (C++ Samples)

Download sample

MFCCALC is an Automation server. It implements a simple calculator similar to the CALC application in Windows. It can be driven via Automation by running its CalcDriv project, or it can be run standalone by running its MfcCalc project.

This sample is similar to the MFC samples MFCCALC and CALCDRIV except that it uses the native compiler COM support and it combines the calculator and driver in a single sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution mfccalc.sln.
2. On the **Build** menu, click **Build Solution**.
3. Set MfcCalc as the startup project (right-click the project node and click **Set as StartUp Project**).
4. On the **Debug** menu, click **Start Without Debugging**. This will run the calculator standalone. Close the MfcCalc application before continuing.
5. Now run the calculator using Automation. Set CalcDriv as the startup project and click **Start Without Debugging**. Type an expression in the **MFC Calc Driver** dialog box and click the **Go** button to run the calculator.

Comparison with the MFC Version

For more information, see the corresponding MFC samples MFCCALC and CALCDRIV.

The differences between this sample and the MFC samples:

- The **COleDispatchDriver** class is no longer necessary. Instead, **#import** is used to import a type library.
- **_bstr_t** and **_variant_t** wrapper classes are used to simplify operations of **BSTR** and **VARIANT** types.
- **__declspec(property)** is used to simplify assignment operations for properties of COM objects.
- COM error handling is now performed by using the **_com_error** class.

With the Visual C++ native compiler COM support, the resulting sample code is shorter and more efficient. The MFC-based MFCCALC sample uses dispinterfaces only. Compare this sample with the MFC version to see the differences in source code.

Keywords

This sample demonstrates the following keywords:

dispinterface; #import; _com_ptr_t; _variant_t; _bstr_t; _com_error

See Also

Reference

Compiler COM Support

Other Resources

Compiler COM Support Samples
MFCCALC Sample: Demonstrates an Automation Server Application
CALCDRIV Sample: Demonstrates an Automation Client Application

Custom Wizard Samples

The samples in this section show how to create your own wizards to streamline tasks related to creating custom applications or adding code.

ConsoleApp

Application wizard that creates a Win32 Console Application project.

For details on deleting a custom wizard and removing the wizard from the New Project dialog box, see Files Created for Your Wizard.

See Also

Other Resources

Visual C++ Samples

ConsoleApp Sample: Demonstrates How to Create a Custom Application Wizard

Download sample

The ConsoleApp sample illustrates how to create a custom application wizard and add it to the list of projects. ConsoleApp generates a C++ Win32 Console Application project depending on the options set by the user.

This wizard uses the VS wizard control to display an HTML interface to the user. The page's JScript code calls the control to store symbol information that is later used to parse the application code templates. After the application has been generated, the Project Model is used to set the project settings.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Installing and Running the Sample

To install the wizard

1. Copy all of the sample files to your hard disk.
2. Copy the contents of the VCProjects folder to the Vc7\VCProjects folder in the directory where Visual Studio is installed (for example, c:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\VCProjects\).
3. Copy the contents of the VCwizards folder to the Vc7\VCWizards folder in the directory where Visual Studio is installed (for example, c:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\VCWizards\).

To use the wizard

1. Start Visual Studio.
2. From the **File** menu, point to **New** and then click **Project**.
3. In the **New Project** dialog box, select the **Visual C++ Projects** folder.
4. The new **Console Appwizard** appears in the **Templates** pane. Double-click the icon to display the wizard.
5. Click the **Application Settings** tab and select any options you want your project to have.
6. Click **Finish** to generate the project.

Keywords

This sample demonstrates the following keywords:

IVCCLCompilerTool::Defines; IVCCLCompilerTool::InlineFunctionExpansion; IVCCLCompilerTool::RuntimeLibrary; IVCCLCompilerTool::UsePrecompiledHeader; IVCCLCompilerTool::UsePrecompiledHeader; IVCLinkerTool::AdditionalInputs; IVCLinkerTool::GenerateDebug; IVCLinkerTool::LinkIncremental; IVCLinkerTool::OutputFile; IVCLinkerTool::ProgramDatabaseFile; IVCLinkerTool::SubSystem; IVCWizCtlUI::AddSymbol; IVCWizCtlUI::Alert; IVCWizCtlUI::CreateGuid; IVCWizCtlUI::FindSymbol; IVCWizCtlUI::FormatGuid; IVCWizCtlUI::Load; IVCWizCtlUI::Next; IVCWizCtlUI::SetDefaults

See Also

Other Resources

Custom Wizard Samples

Event Handling Samples

The samples in this section demonstrate event handling in Visual C++.

In This Section

COMEvents Sample: Demonstrates Creating and Using COM Events

Demonstrates event handling using COM.

See Also

Other Resources

Visual C++ Samples

MEDriver Sample: Demonstrates COM Eventing in the .NET Framework

COMEvents Sample: Demonstrates Creating and Using COM Events

Download sample

The COMEvents sample demonstrates event handling using COM.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run COMEvents

1. Open the solution file COMEvents.sln.
2. From the **Build** menu, click **Build Solution**.
3. Right-click the EventReceiver node in **Solution Explorer** and select **Set as Startup Project**.
4. From the **Debug** menu, click **Start Without Debugging**.

How the Sample Works

The COMEvents sample creates an event source `CSource` and an event receiver `CSink` using the **event_source** and **event_receiver** attributes, respectively.

The event source declares the methods in the interfaces `IEvent`, `IEvent2`, and `IJW` as events (see Ifaces.h for the interface definitions).

The event receiver declares handler methods `f1` through `f6`, and hooks each of these handlers to an event.

The main code instantiates `CSource` and `CSink`. The `CSink` constructor performs the hooking. The main code then fires the events, unhooks the events, then fires the events again.

See Also

Other Resources

Event Handling Samples

Extensibility

This section includes the abstracts for the Visual C++ extensibility samples.

In This Section

Code Model Samples

Provides links to samples demonstrating various features of the Visual C++ Code Model.

Custom Wizard Samples

Provides links to samples demonstrating how to create your own wizards.

Project Model Samples

Provides links to samples showing how to programmatically use objects in the project model.

Related Sections

Visual C++ Sample Applications

Provides links to sample code showing the capabilities of Visual C++ and the libraries and technologies it supports.

Visual C++ Walkthroughs

Provides links to walkthroughs that discuss the development of a specific application type or major application feature using a serious of steps.

Code Model Samples

The samples in this section demonstrate various features of the Visual C++ Code Model.

CodeModelMacros

A collection of macros that use the Code Model to perform useful tasks.

See Also

Concepts

Visual C++ Code Model

Other Resources

Visual C++ Samples

CodeModelMacros Sample: Demonstrates How to Use the Code Model Functions

Download sample

This sample is a collection of macros that use the Code Model to perform useful tasks. The macros included are:

- **HierarchyGenerator** Generates an HTML file with an inheritance tree for the classes, structs, and interfaces in a project.

  This macro first assembles a collection of the terminal classes, structs, or interfaces (those that don't have any derived types). It does this by verifying that no other code element has the class, struct, or interface in question in its bases collection. Then it generates an HTML report listing the terminal objects with all of its bases. The macro adds this file to the first project in the solution.

- **InterfaceDocGenerator** Generates HTML documentation for the implemented interfaces in a project. It gathers the comments from the class functions that implement the interface methods.

  This macro enumerates the interfaces contained in the CodeModel.Interfaces collection, and then it finds the classes that implement each interface. For each class, it looks up the interface's methods implementation and stores the comments in an HTML file.

- **UserTypeGenerator** Generates a file named usertype.dat containing the names of the classes in your project. If you place this file in the same location where devenv.exe resides (for example, c:\Program Files\Microsoft Visual Studio .NET 2003\Common7\IDE), the source code editor will colorize your class names.

  This macro writes to a file (usertype.dat) the names of the classes contained in the CodeModel.Classes collection.

- **VirtualFunctionFinder** Adds a TODO comment to the virtual functions found in the project.

  The VirtualFunctionFinder finds the virtual functions declared in the project by recursively verifying the functions of each class or struct. Then it adds a comment to each one by changing the CodeFunction.Comment property.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Installing and Running the Sample

To install the macros

1. Copy the file CodeModelMacros.vsmacros to your hard drive.
2. Start Visual Studio.
3. From the **Tools** menu, point to **Macros** and then click **Macro Explorer**. This will open the Macro Explorer window.
4. From the **Tools** menu, point to **Macros** and then click **Load Macro Project**. This will open the **Add Macro Project** dialog box.
5. Browse to the CodeModelMacros.vsmacros file and click the **Open** button. This action will load the macros file and the macros will appear in the Macro Explorer.
6. Expand the **CodeModelMacros** node. You will see one module for each macro.

To execute HierarchyGenerator

1. Expand the **HierarchyGenerator** node in the Macro Explorer.
2. Double-click the **HierarchyGenerator** macro. This will execute the macro and add an HTML file to the first project in your solution.

To execute InterfaceDocGenerator

1. Expand the **InterfaceDocGenerator** node in the Macro Explorer.
2. Double-click the **InterfaceDocGenerator** macro. This will execute the macro and add an HTML file to the first project in your solution.

To execute UserTypeGenerator

1. Expand the **UserTypeGenerator** node in the Macro Explorer.
2. Double-click the **UsertypeC** macro. This will execute the macro and will create a usertype.dat file in the root directory of your C: drive.

To execute VirtualFunctionFinder

1. Expand the **VirtualFunctionFinder** node in the Macro Explorer.
2. Double-click the **MainVirtualFunctionFinder** macro. This will execute the macro and will add a TODO comment to the virtual functions declared in your project.

Keywords

This sample demonstrates the following keywords:

CodeClass::Members; CodeElement.CodeTypeFromFullName; CodeElement::Children; CodeElement::FullName; CodeElement::Kind; CodeFunction::CanOverride; CodeFunction::Comment; CodeFunction::InfoLocation; CodeFunction::Name; CodeInterface::Name; CodeModel::CodeElements; CodeModel::Language; CodeNamespace::Members; CodeStruct::Members; CodeType::Bases; Project::CodeModel; Solution::Count; VCCodeBase::FullName; VCCodeBase::Name; VCCodeClass::Functions; VCCodeClass::ImplementedInterfaces; VCCodeClass::IsSelf; VCCodeClass::Name; VCCodeFunction::Attributes; VCCodeFunction::Comment; VCCodeFunction::IsVirtual; VCCodeInterface::Functions; VCCodeModel::Classes; VCCodeModel::Interfaces

See Also

Other Resources

Code Model Samples

Project Model Samples

The samples in this section show how to programmatically use objects in the project model.

VCProjectEngineObject Samples

A series of five samples that illustrate how to programmatically modify a Visual C++ project. The samples are written in the following languages: Visual C++, Visual C#, JScript, Visual Basic .NET, and Visual Basic 6.

See Also

Other Resources

Visual C++ Samples

VCProjectEngineObject Samples

The VCProjectEngineObject samples illustrate how to programmatically modify a Visual C++ project. The sample shows how to do the following:

- Create a new instance of the **VCProjectEngine** object and use it to load an existing Visual C++ project for modification.
- Use the **VCProject** object to change the project name and add a new configuration to the project.
- Use the **VCConfiguration** object to get the debug configuration from the project and change the project type from a dynamic-link library (.dll) to an application (.exe).
- Get the linker tool from the configuration and use the **VCLinkerTool** object to change the Show Progress property from the default Not Set to Display All Progress Messages.
- Use the **AddFile** method to add a .cpp file to the project.
- Use the **IVCCollection** object to enumerate to the existing .cpp file, access the file level release configuration, get the compiler tool associated with the file, and change the Optimization property to Full Optimization.
- Save the project to a different .vcproj file.

Versions of the Sample

The sample is available for the following languages:

- C++
- C#
- JScript
- Visual Basic .NET

Building and Running the Sample

The target project file (Myproject.vcproj) will be modified by the sample and saved to a new file (MyProjectNew.vcproj). To facilitate an easy return to the starting state for further exploration and subsequent runs, make copies of the target project files before running the sample.

Also, prior to running the sample, open MyProject.vcproj and open the **Property Pages** dialog box to examine its properties.

To build and run the Visual C++, Visual C#, or Visual Basic .NET sample

1. Open the solution file for the desired language project.
2. From the **Build** menu, click **Build**.
3. Copy the Sample.exe file in the build output directory to the MyProject subdirectory. For the Visual C# and Visual Basic samples, you also need to copy Interop.VCProjectEngineLibrary_1_0.dll to the MyProject subdirectory.
4. From the command prompt in the MyProject subdirectory, type `Sample`.

To build and run the sample in JScript

1. Copy the project sample.js file to the MyProject subdirectory.
2. From the command prompt in the MyProject subdirectory, type `cscript sample.js`.

After running the sample, observe and compare the changes made and saved to MyNewProject.vcproj. The changes are that the project name has been changed to Voila, a new configuration has been added, and the New.cpp file has been added to the project.

In addition, the following properties have been modified in the **Property Pages** dialog box:

- **Show Progress** property (**Linker** folder, **General** page)
- **Optimization** property (**C/C++** folder, **Optimization** page)

Sample Files

The files for each language are:

Language	Files
Visual C++	Sample.vcproj and Sample.cpp
Visual C#	Sample.csproj, Sample.csproj.user, Assemblyinfo.cs, and Class1.cs
JScript	Sample.js
Visual Basic .NET	Sample.vbproj, Sample.vbproj.user, and Module1.vb

The target project contains the following files:

- MyProject.vcproj – The project file.
- New.cpp – The file to be added.
- Existing.cpp – The file whose property is to be modified.

Keywords

This sample demonstrates the following keywords:

VCProjectEngineObject; VCProject; VCConfiguration; VCLinkerTool; AddFile; IVCCollection

See Also

Other Resources

Project Model Samples

VCProjEngineObject_CPP Sample: Programmatically Modifies a Visual C++ Project Using C++

Download sample

This is the Visual C++ version of the VCProjectEngineObject sample.

Use this topic to copy the sample source files to your hard disk.

See VCProjectEngineObject Samples for details on building and running the sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

See Also

Other Resources

VCProjectEngineObject Samples

VCProjEngineObject_CS Sample: Programmatically Modifies a Visual C++ Project Using C#

Download sample

This is the Visual C# version of the VCProjectEngineObject sample.

Use this topic to copy the sample source files to your hard disk.

See VCProjectEngineObject Samples for details on building and running the sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

See Also

Other Resources

VCProjectEngineObject Samples

VCProjEngineObject_JS Sample: Programmatically Modifies a Visual C++ Project Using JScript

Download sample

This is the JScript version of the VCProjectEngineObject sample.

Use this topic to copy the sample source files to your hard disk.

See VCProjectEngineObject Samples for details on building and running the sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

See Also

Other Resources

VCProjectEngineObject Samples

VCProjEngineObject_VB7 Sample: Programmatically Modifies a Visual C++ Project Using Visual Basic 2005

Download sample

This is the Visual Basic .NET version of the VCProjectEngineObject sample.

Use this topic to copy the sample source files to your hard disk.

See VCProjectEngineObject Samples for details on building and running the sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

See Also

Other Resources

VCProjectEngineObject Samples

International Samples

The samples in this section are related to writing code for international markets.

In This Section

IME Sample: Demonstrates How to Control IME Mode and Implement IME Level 3

Demonstrates how to control the Input Method Editor mode and how to implement IME level 3.

See Also

Reference

ShowLocalized

Other Resources

Visual C++ Samples

IME Sample: Demonstrates How to Control IME Mode and Implement IME Level 3

Download sample

The IME sample uses `CIMEEdit`, a subclass of **CEdit**, to create an edit control. `CIMEEdit` handles all input characters, and it composes DBCS strings. The sample has the following features:

- It shows how to control IME mode using C/C++. You can set the initial IME mode of `CIMEEdit` as to input language, shape, and so on. If you want your edit control initially set to get Korean characters, you can add code as in this sample. Also, if you want only English, you can disable IME.
- It also shows how to implement IME level 3 in your source code. IME level 3 is more convenient and powerful for East Asian users. You can create a custom control or window that supports IME level 3 using this sample.
- This sample determines the font and code page used for Ansi-Unicode conversion by the keyboard layout so that you can try four different IMEs for all East Asian languages without changing either font or code page. This is a feature of Windows 2000 and is not a requirement for support of IME level 3.
- The function `GetCombinedCharLength` supports only surrogate characters in this sample. If you need to support combined characters, add your code to the method.

To use IME functions, your program must include IMM.H and link with IMM32.LIB.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

Note
If you are running Visual C++ Express Edition, you might need to install the Platform SDK before running this sample. For information on how to do this, see How to: Use Visual C++ Express Edition with the Microsoft Platform SDK.

To build and run this sample

1. Open the solution IME.sln.
2. From the **Build** menu, click **Build**.
3. From the **Debug** menu, click **Start Without Debugging**.

Run the sample and try out the following features.

Control IME Mode

- You can change IME mode using check boxes. If you change options using check boxes, the status of the IME window will be changed.
- You can change IME mode using the IME window. If you change options using the IME window, check boxes in the sample will be changed.
- You can enable or disable IME.

Implement IME Level 3

- If you input double-byte characters, you can see how to composite strings.
- If you have the proper font on Windows 2000, you can see surrogate characters.

Classes and Keywords

This sample demonstrates the following classes:

CEdit

This sample demonstrates the following keywords:

CBitmap::OnPaint; CEdit::OnChar; CEdit::OnSetFocus; CEdit::OnKillFocus; CEdit::OnKeyDown; CEdit::WindowProc; ImmGetContext; ImmReleaseContext; ImmGetConversionStatus; ImmSetConversionStatus; ImmGetOpenStatus; ImmSetOpenStatus; ImmAssociateContext; ImmNotifyIME; ImmGetCompositionString; ImmGetProperty; ImmSetCandidateWIndow; ImmSetCompositionWIndow

See Also

Other Resources

International Samples

Interoperability Samples

The following topics are the abstracts for the Interoperability samples written in C++.

In This Section

MEDriver Sample: Demonstrates COM Eventing in the .NET Framework

Illustrates the use of COM events (fired from an unmanaged COM server) through a .NET Framework wrapper, automatically generated from the COM server's type library.

QStat Sample: Wraps a COM Object to Expose Business Logic

Shows how to create a DLL that wraps access to a COM object and exposes its functionality to .NET Framework clients.

TilePuzzle Sample: Demonstrates Interoperability Between C# and C++

Demonstrates interoperability between managed components (written with C++ and C#) and native components (written with C++ using COM attributes).

Related Sections

Using C++ Interop (Implicit PInvoke)

Describes and demonstrates using C++ to combine managed and unmanaged functionality into single applications.

Mixed (Native and Managed) Assemblies

Provides an overview of mixed assemblies, which allow managed and unmanaged C++ components to interoperate.

See Also

Other Resources

Visual C++ Sample Applications

MEDriver Sample: Demonstrates COM Eventing in the .NET Framework

Download sample

The MEDriver sample illustrates the use of COM events (fired from an unmanaged COM server) through a .NET Framework wrapper, automatically generated from the COM server's type library.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run MEDriver using Visual Studio

1. In the Visual Studio IDE, load the solution file MEDriver.sln.
2. In Solution Explorer, right-click the **MEDriver** solution.
3. On the shortcut menu, click **Build Solution**.
4. On the **Debug** menu, click **Start**.

When you run MEDriver.exe, click the **Start** button at least once, and then click the **Advise** button several times. Each click of the **Advise** button adds a sink to the **Fire** event, which makes the display wider. If you do not click the **Advise** button, you will not see any activity in the display.

Keywords

This sample demonstrates the following keywords:

Control::CreateGraphics; Control::ClientRectangle; Graphics::DrawLine; Color::FromArgb; Mutex::WaitOne; Mutex::ReleaseMutex; Hashtable::Item; Application::Run; Button::Location; Button::Size; Button::Text; ControlCollection::AddRange

See Also

Other Resources

Interoperability Samples

QStat Sample: Wraps a COM Object to Expose Business Logic

Download sample

The QStat sample shows how to create a DLL that wraps access to a COM object and exposes its functionality to .NET Framework clients using C++.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build the sample

1. In the Visual Studio IDE, load the solution file QStat.sln.
2. In Solution Explorer, right-click the **QStat** solution.
3. On the shortcut menu, click **Build Solution**.

To run the sample

1. In Solution Explorer, right-click the **netClient** project.
2. On the **Debug** menu, click **Start**.

Once the clients have started, a series of messages are printed to the console, notifying you of calls into the business logic.

Note
To get more information on the interaction between components, step through the code by either setting breakpoints or using the **Debug** menu to step into or over the sample code.

Project Description

The focus of the sample is on accessing the business logic from both COM and .NET Framework clients and not on implementation of the business logic itself, so the business logic is relatively uninteresting. In fact, the business logic in this case does simple arithmetic (geometric, harmonic, and quadratic mean calculations), returning the result to the client.

It is a better programming practice to separate business logic from the scaffolding that exposes that business logic (COM in this case). However, this sample assumes that the business logic has been implemented directly within the COM object that will be consumed and that there is no access to the underlying business logic except by going through this COM object.

The sample creates a wrapper DLL that consumes the COM object via **#import** and exposes a .NET Framework class to be consumed by .NET Framework clients. This is done by defining a native C++ class, `nativeMeanCalc`, which contains a COM smart pointer data member brought in through **#import**. A .NET Framework class is defined that contains a pointer to the native C++ class. The .NET Framework class, `netMeanCalc`, contains a pointer to a `nativeMeanCalc` object, which completes the pathway to the destination COM object. `netMeanCalc` is the public class that the .NET Framework clients will consume.

There are two client projects:

- `nativeClient` is a COM client and is implemented in native C++.
- `netClient` is written using the new syntax for C++.

Classes and Keywords

This sample demonstrates the following keywords:

__gc; #using; #import; try; catch; _com_error; Console::WriteLine

See Also

Other Resources

Interoperability Samples

TilePuzzle Sample: Demonstrates Interoperability Between C# and C++

Download sample

The TilePuzzle sample demonstrates several major Visual C++ features:

- The sample is comprised of components implemented with several new technologies: C++, C#, and the new COM-related attributes.
- The sample demonstrates interoperability between managed components (written with C++ and C#) and native components (written with C++ using COM attributes).

The sample implements a basic puzzle game called the Tile Puzzle. The sample loads a bitmap, splits the bitmap into any number of tiles (determined by the user), and randomizes the individual tiles. The user then solves the puzzle by sliding individual tiles around until the original picture is displayed. In addition to these features, the sample has the ability to solve the puzzle using heuristic search algorithms written in C++ and the .NET Framework classes.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run TilePuzzle using Visual Studio

1. In the Visual Studio IDE, load the solution file PUZZLE.sln.
2. In Solution Explorer, right-click the **PUZZLE** solution.
3. On the shortcut menu, click **Build Solution**.
4. On the **Debug** menu, click **Start**.

After the project builds successfully, try to solve the puzzle yourself.

Architecture of the Sample

Here is the basic architecture:

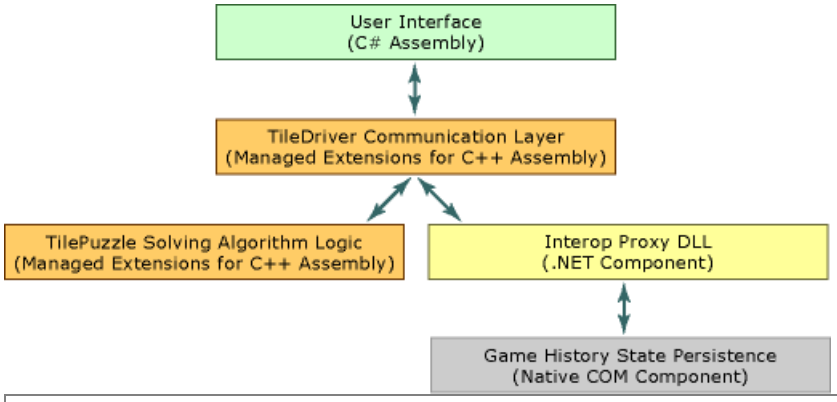

Note
The save and load game features are not implemented.

To access the native COM component from within the managed .NET Framework objects, the sample uses TLBIMP.EXE to generate a .NET Framework proxy dll.

Classes

The sample demonstrates the following classes:

- System.Windows.Forms.Form – Implements the `AboutForm` object found in the PUZZLE project.
- System.Object – Implements the `GameLevelEnum` object found in the PUZZLE project.
- System.Delegate – Implements the `SolveThreadProcDlg` object found in the PUZZLE project.

See Also

Reference

gcnew
Classes and Structs (Managed)
String
Array
Assert
Int32

Other Resources

Interoperability Samples

MASM Samples

The samples in this section demonstrate support for Microsoft Macro Assembler (MASM) source files in Visual C++. They are not meant to teach assembly language programming or to present an optimal coding solution.

For more information on MASM, see Microsoft Macro Assembler Reference.

In This Section

EUCLIDSTEP1

A pure C project that demonstrates Euclid's algorithm for finding the greatest common divisor.

EUCLIDSTEP2

An extension of EUCLIDSTEP1 that is a mixed C and ASM project. The core of Euclid's algorithm is moved from the .c file to an .asm file, with the .c file calling into the .asm file.

PRIMESSTEP1

A pure C project that demonstrates the sieve of Eratosthenes to find prime numbers.

PRIMESSTEP2

An extension of PRIMESSTEP1 that is a mixed C and ASM project that moves the core algorithm to the .asm file.

PRIMESSTEP3

An extension of PRIMESSTEP2 that adds a separate C header file and an .asm include file to declare the extern function and global data structure.

See Also

Other Resources

Visual C++ Samples

EUCLIDSTEP1 Sample: Provides a Baseline for the EUCLIDSTEP2 Sample

Download sample

The EUCLIDSTEP1 sample is a pure C project that demonstrates Euclid's algorithm for finding the greatest common divisor. No .asm files are included in this project. This sample provides a baseline from which to expand to including an .asm file, as demonstrated in the EUCLIDSTEP2 sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution EuclidStep1.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

See Also

Other Resources

EUCLIDSTEP2 Sample: Demonstrates a Mixed C and ASM Project
MASM Samples
Microsoft Macro Assembler Reference

EUCLIDSTEP2 Sample: Demonstrates a Mixed C and ASM Project

Download sample

The EUCLIDSTEP2 sample is an extension of EUCLIDSTEP1 as a mixed C and ASM project. In this sample, the core of Euclid's algorithm is moved from the .c file to an .asm file, with the .c file calling into the .asm file.

In the .asm source file, the settings for the **Command Line** and **Ouputs** boxes of the **Custom Build Step** tab are included in comments. To view these settings, right-click the .asm file and click **Properties**. These settings, which support including the .asm file in the build, have already been entered for you.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution EuclidStep2.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

You can use this sample to explore Visual C++ support for mixed C/C++ and ASM projects, such as using the **Custom Build Step** tab in the **Properties** dialog box and debugging .asm files at a source level. You can use the .asm file or project file as a template to incorporate .asm files into a Visual C++ project.

See Also

Other Resources

EUCLIDSTEP1 Sample: Provides a Baseline for the EUCLIDSTEP2 Sample
MASM Samples
Microsoft Macro Assembler Reference

PRIMESSTEP1 Sample: Provides a Baseline for the PRIMESSTEP Sample Series

Download sample

The PRIMESSTEP1 sample is a pure C project that demonstrates the sieve of Eratosthenes to find prime numbers. No .asm files are included in this project. This sample provides a baseline from which to expand to including an .asm file, as demonstrated in the PRIMESSTEP2 sample.

This implementation uses a global array as a principal data structure.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution PrimesStep1.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

See Also

Other Resources

PRIMESSTEP2 Sample: Demonstrates a Mixed C and ASM Project
PRIMESSTEP3 Sample: Demonstrates a Mixed C and ASM Project with a C Header File
MASM Samples
Microsoft Macro Assembler Reference

PRIMESSTEP2 Sample: Demonstrates a Mixed C and ASM Project

Download sample

The PRIMESSTEP2 sample is a mixed C and ASM project that moves the core algorithm to the .asm file. The extern function and global data structure are separately declared in the .c file and the .asm source file.

In the .asm source file, the settings for the **Command Line** and **Ouputs** boxes of the **Custom Build Step** tab are included in comments. To view these settings, right-click the .asm file and click **Properties**. These settings, which support including the .asm file in the build, have already been entered for you.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution PrimesStep2.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

You can use this sample to explore Visual C++ support for mixed C/C++ and ASM projects, such as using the **Custom Build Step** tab in the **Properties** dialog box and debugging .asm files at a source level. You can use the .asm file or project file as a template to incorporate .asm files into a Visual C++ project.

See Also

Other Resources

PRIMESSTEP1 Sample: Provides a Baseline for the PRIMESSTEP Sample Series
PRIMESSTEP3 Sample: Demonstrates a Mixed C and ASM Project with a C Header File
MASM Samples
Microsoft Macro Assembler Reference

PRIMESSTEP3 Sample: Demonstrates a Mixed C and ASM Project with a C Header File

Download sample

The PRIMESSTEP3 sample is a mixed C and ASM project that moves the core algorithm to the .asm file and adds a separate C header and .asm include file to declare the extern function and global data structure.

In the .asm source file, the settings for the **Command Line** and **Ouputs** boxes of the **Custom Build Step** tab are included in comments. In addition, the settings for the **Additional Dependencies** box lists the .asm include file. This assembles the .asm file if either the .asm file or the .asm include file is new than the object file generated by **Custom Build Step**.

To view these settings, right-click the .asm file and click **Properties**. These settings, which support including the .asm file in the build, have already been entered for you.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run this sample

1. Open the solution PrimesStep3.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

You can use this sample to explore Visual C++ support for mixed C/C++ and ASM projects, such as using the **Custom Build Step** tab in the **Properties** dialog box and debugging .asm files at a source level. You can use the .asm file or project file as a template to incorporate .asm files into a Visual C++ project.

See Also

Other Resources

PRIMESSTEP1 Sample: Provides a Baseline for the PRIMESSTEP Sample Series
PRIMESSTEP2 Sample: Demonstrates a Mixed C and ASM Project
MASM Samples
Microsoft Macro Assembler Reference

MFC Samples

The following table is an alphabetical list of the MFC samples included with Visual C++.

In This Section

ACDUAL Sample: Adds Dual Interfaces to an Automation Application

Demonstrates how to add dual interface support to an MFC-based Automation server.

AUTOCLIK Sample: Automation Server Application

Illustrates Automation features. Includes AUTODRIV, a simple Automation client application that drives the AUTOCLIK sample application.

BUTTON Sample: Demonstrates a Menu and a Property Page

Demonstrates use of an in-place active menu, a stock property page, and the About box control option.

CALCDRIV Sample: Demonstrates an Automation Client Application

Automation client.

CIRC Sample: ActiveX Control

Demonstrates ActiveX control basics, including control painting, stock and custom properties, stock and custom events, use of colors and fonts, the stock Font property page, the default property page, and versioning

CMNCTRL1 Sample: Demonstrates Common Control MFC Classes, Part 1

Demonstrates how to create and change the styles of Windows Common Controls using MFC classes (Part 1).

CMNCTRL2 Sample: Demonstrates Common Control MFC Classes, Part 2

Demonstrates how to create and change the styles of Windows Common Controls using MFC classes (Part 2).

COLLECT Sample: Illustrates MFC Collection Classes

Demonstrates MFC C++ template-based collection classes and standard prebuilt collection classes.

CONTAINER Sample: Visual Editing Container Application

Demonstrates a Visual Editing Container Application.

CTRLBARS Sample: Illustrates Custom Control Bars

Custom toolbar and status bar, dialog bar, and floating palette.

CTRLTEST Sample: Implements Custom Controls

Owner-draw list box and menu, custom control, bitmap button, spin control.

CUBE Sample: Demonstrates an OpenGL Application

OpenGL application using MFC device contexts along with OpenGL's resource contexts.

DHtmlExplore Sample: Demonstrates Using MFC DHtml Classes

Demonstrates handling DHTML events and using DHTML DDX.

DIBLOOK Sample: Illustrates the Use of DIBs and Color Palettes

Demonstrates the Use of DIBs and Color Palettes.

DLGCBR32 Sample: Demonstrates Adding a Status Bar and Toolbar to Dialog Boxes

Adding a toolbar and a status bar to a dialog-based application.

DLGTEMPL Sample: Creates Dialog Templates Dynamically

Demonstrates the dynamic creation of dialog templates.

DLLHUSK Sample: Dynamically Links the MFC Library

Sharing the DLL version of the Foundation class library with an application and custom DLL.

DLLScreenCap Sample: Demonstrates a Regular DLL That Statically or Dynamically Links to MFC

A regular DLL that can be statically or dynamically linked to the Microsoft Foundation Class Library.

DOCKTOOL Sample: Demonstrates Dockable Toolbars

Dragging and floating toolbars that are "dockable".

DRAWCLI Sample: Illustrates Integrating Active Container Support with Application-Specific Features

Full-featured object-oriented drawing application that is also an ActiveX Visual Editing container.

DYNAMENU Sample: Dynamically Updates Menus

Dynamically modifying list of items in menus; handling commands not known at compile time; and updating the status bar command prompt for such commands.

HIERSVR Sample: Demonstrates a Server Application with OLE Drag and Drop

Demonstrates a Server Application with OLE Drag and Drop.

HTMLEdit Sample: Wraps the Internet Explorer MSHTML Editing Control

Wraps the Internet Explorer MSHTML editing control.

INPROC Sample: Demonstrates an In-Process Automation Server Application

An in-process Automation server that can be loaded as a DLL in the client's address space.

IPDRIVE Sample: Demonstrates an Automation Client Application

A simple Automation client application that drives the INPROC sample application.

LICENSED Sample: Enforces Use of Design-Time and Run-Time Licenses

A control that enforces use of a design-time and run-time license.

LOCALIZE Sample: Control with a Localized User Interface

A control with a localized user interface that demonstrates use of separate type libraries and resource dynamic-link libraries (DLLs) for localization.

MDI Sample: Demonstrates MDI Without Using Doc/View Architecture

MDI application that does not use documents and views.

MDIDOCVW Sample: Demonstrates MDI Using Doc/View Architecture

New version of the MDI sample that uses the document/view architecture.

MFCBIND Sample: Active Document Container

Shows how to create an Active document (formerly known as a DocObject) container.

MFCIE Sample: Demonstrates the MFC CHtmlView and CReBar Classes

Demonstrates the MFC CHtmlView and CReBar Classes.

MFCCALC Sample: Demonstrates an Automation Server Application

An Automation server that implements a simple calculator.

MMXSwarm Sample: Demonstrates CImage and Visual C++ MMX Support

Demonstrates how to use **CImage**, the **__m64** data type, and device-independent bitmaps (DIBs).

MODELESS Sample: Uses a CDialog Object as a Modeless Dialog Box

Demonstrates the use of an MFC **CDialog** object as a modeless dialog.

MTMDI Sample: Demonstrates an MFC User Interface Thread

Multithread illustration, where user-interface events are processed in a separate user-interface thread.

MTRECALC Sample: Supports Multithread Applications

Multithread illustration, where recalculations are done in a worker thread.

NPP Sample: Demonstrates the Windows Messaging API (MAPI)

Demonstrates the Windows Messaging API (MAPI)

OCLIENT Sample: Illustrates a Visual Editing Container Application

ActiveX Visual Editing container application, with drag and drop.

OLEVIEW Sample: ActiveX Object Viewer

Implementing an OLE object browser through custom OLE interfaces.

PROPDLG Sample: Demonstrates Property Sheet Support

Property sheets (dialogs).

REGSVR Sample: Invokes Self-Registration Code

Demonstrates the invocation of Self-Registration Code.

ROWLIST Sample: Demonstrates Selecting Full Rows in List Views

Illustrates full row selection in a list-view common control.

SCRIBBLE Sample: MFC MDI Drawing Application

SCRIBBLE is drawing application that provides simple illustrations of a wide breadth of MFC features.

SimpleImage Sample: Loads, Resizes, Converts, and Saves Images

Demonstrates loading, resizing, conversion, and saving images.

SNAPVW Sample: Uses Property Pages in a Form View Application

Shows how to use property pages in a MDI child frame window.

SPEAKN Sample: Demonstrates Multimedia Sound Using User-Defined Resources

Demonstrates Multimedia Sound Using User-Defined Resources.

SPINDIAL Sample: Demonstrates Property Page Validation

A control with the visual appearance of a spin-dial that demonstrates property page data validation.

Spiro Sample: Animated Drawing Game

A game that shows to use **CImageList** and how to use memory display contexts in applications requiring animation effects.

StockTicker Sample: Displays Web Information in an ATL Control

An MFC and ATL application that retrieves stock quotes from the Web and displays them in an ATL control. The user can set the display's properties, including rate of data refresh.

SUPERPAD Sample: Demonstrates a Visual Editing Server That Edits Text Using CEditView

Demonstrates a Visual Editing Server That Edits Text Using CEditView.

TIME Sample: Fires a Timer at Set Intervals

A control that is invisible at run time and fires a timer event at set intervals. Demonstrates notification functions and ambient properties.

TSTCON Sample: ActiveX Control Test Container

Implements an ActiveX control container using MFC's support for OLE embedding. You can use TSTCON to test ActiveX controls, change their properties, and invoke their methods.

TESTHELP Sample: ActiveX Control with Tooltips and Help

An ActiveX control that has its own help file and tooltips.

TRACKER Sample: Illustrates Various CRectTracker Styles and Options

Demonstrates various CRectTracker styles and options.

VariantUse Sample: Demonstrates the Use of Variants

Demonstrates the use of the variant data type.

VIEWEX Sample: Demonstrates Multiple Views, Scroll Views, and Splitter Windows

Multiple views, scroll view, splitter windows.

WinFormIntegration Sample: Demonstrates Hosting a WinForms User Control in an MFC Application

Demonstrates how to host a WinForm user control as a dialog or a view in an MFC application.

WORDPAD Sample: The Windows Application

Uses MFC's support for rich edit controls to create a basic word processor.

XLIST Sample: Displays Text or Bitmaps

A control, subclassed from a Windows list box, that displays text or bitmap items.

See Also

Other Resources

Visual C++ Samples

MFC Advanced Samples

The following topics are the abstracts for the MFC advanced samples. For a list of all MFC samples, see MFC Samples.

See Also

Other Resources

MFC Samples

COLLECT Sample: Illustrates MFC Collection Classes

Download sample

The COLLECT sample illustrates a wide variety of collection classes offered by the Microsoft Foundation Class Library (MFC).

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the COLLECT sample

1. Open the solution collect.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

The **Example** menu lists nine collection class illustrations organized according to the following table. Each example presents a form view that exercises many common collection operations:

- Adding a new element.
- Inserting a new element in the middle of a list.
- Searching an element (by clicking its entry in the dialog box's list box).
- Searching an entry in the map by entering the Key and clicking Find.
- Updating the value of an element.
- Removing an element.
- Removing all the elements of the collection.

You can save changes to all the examples by clicking **Save** on the **File** menu and read them back by clicking **Open** on the **File** menu.

MFC Collection Classes

The MFC collection classes can be classified according to shape and design. MFC offers classes for three types of collection shapes:

- Lists
- Arrays
- Maps

MFC also offers three types of designs:

- No use of C++ templates.
- Use of a C++ template–based collection of simple elements.
- Use of a C++ template–based collection of type-safe pointers.

The following table identifies all the MFC collection classes sorted according to shape and design. The COLLECT sample illustrates nine of these collections, one from each cell in the table. Each represents a distinct combination of shape and design. Also, because usage of the **CTypedPtrArray** and **CTypedPtrList** template-based collections differs according to whether they are used to hold pointers to **CObjects** or to non-**CObject**s, the COLLECT sample illustrates both cases. The nine collections illustrated by COLLECT are indicated with an asterisk.

Collection shapes	Non-template based	Template-based simple elements	Template-based typeSafe pointers
List	**CStringList***	**CList***	**CTypedPtrList**
	CObList		(of pointers)*
	CPtrList		**CTypedPtrList**
			(of **CObject**s)
Array	**CByteArray**	**CArray***	**CTypedPtrArray**
	CUIntArray		(of pointers)
	CWordArray*		
	CDWordArray		**CTypedPtrArray**
	CStringArray		(of **CObject**s)*
	CPtrArray		
	CObArray		
Map	**CMapWordToPtr**	**CMap***	**CTypedPtrMap***
	CMapWordToOb		
	CMapStringToString*		
	CMapStringToPtr		
	CMapStringToOb		
	CMapPtrToWord		
	CMapPtrToPtr		

Additional COLLECT Features

In addition to illustrating the MFC collection classes, COLLECT also illustrates:

- Coordinating the addition and removal of entries in a list box with the addition and removal of entries in a collection.
- Switching between different views of a document, with only one shown at a time. That is, although COLLECT has many different views, it has a single document interface (SDI) instead of a multiple document interface (MDI).

Keywords

This sample demonstrates the following keywords:

AfxMessageBox; CCmdUI::SetCheck; CControlBar::EnableDocking; CDialog::DoModal; CDocument::DeleteContents; CDocument::OnNewDocument; CFrameWnd::DockControlBar; CFrameWnd::EnableDocking; CFrameWnd::GetActiveDocument; CFrameWnd::GetActiveView; CFrameWnd::RecalcLayout; CFrameWnd::SetActiveView; CListBox::AddString; CListBox::DeleteString; CListBox::FindString; CListBox::GetCurSel; CListBox::GetItemData; CListBox::GetItemDataPtr; CListBox::GetText; CListBox::InsertString; CListBox::ResetContent; CListBox::SetItemData; CListBox::SetItemDataPtr; CObject::AssertValid; CObject::Dump; CObject::Serialize; CStatusBar::Create; CStatusBar::SetIndicators; CString::Empty; CString::Format; CString::LoadString; CToolBar::Create; CToolBar::LoadBitmap; CToolBar::SetButtons; CView::DoPreparePrinting; CView::GetDocument; CView::OnBeginPrinting; CView::OnDraw; CView::OnEndPrinting; CView::OnInitialUpdate;

CView::OnPreparePrinting; CView::OnUpdate; CWinApp::AddDocTemplate; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWinApp::OnFileNew; CWnd::DestroyWindow; CWnd::DoDataExchange; CWnd::OnCreate; CWnd::ShowWindow; CWnd::UpdateData; LoadString; SetWindowLong

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

CUBE Sample: Demonstrates an OpenGL Application

Download sample

The CUBE sample is a simple OpenGL application. It demonstrates how to integrate OpenGL with the MFC single document interface (SDI), and how OpenGL's resource contexts are used in conjunction with device contexts.

For a more detailed explanation of OpenGL, refer to the *Platform SDK*.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the CUBE sample

1. Open the solution cube.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

When you start the sample, a three-dimensional colorful cube is drawn. To animate the cube, either click **Play** on the **Edit** menu or click the **Play** toolbar button.

Keywords

This sample demonstrates the following keywords:

CCmdUI::SetCheck; CDC::GetSafeHdc; CDC::RealizePalette; CDC::SelectPalette; CDialog::DoModal; CDocument::OnNewDocument; CFrameWnd::Create; CFrameWnd::DockControlBar; CFrameWnd::EnableDocking; CFrameWnd::GetActiveView; CObject::AssertValid; CObject::Dump; CObject::Serialize; CPalette::CreatePalette; CView::GetDocument; CView::OnDraw; CWinApp::AddDocTemplate; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWinApp::OnFileNew; CWnd::DoDataExchange; CWnd::GetClientRect; CWnd::InvalidateRect; CWnd::KillTimer; CWnd::MessageBox; CWnd::OnCreate; CWnd::OnDestroy; CWnd::OnEraseBkgnd; CWnd::OnPaletteChanged; CWnd::OnQueryNewPalette; CWnd::OnSize; CWnd::OnTimer; CWnd::PreCreateWindow; CWnd::RedrawWindow; CWnd::SetTimer; ChoosePixelFormat; DescribePixelFormat; GetPixelFormat; LoadBitmap; PIXELFORMATDESCRIPTOR; PeekMessage; SetPixelFormat; SwapBuffers; glBegin; glClear; glClearColor; glClearDepth; glEnd; glFinish; glLoadIdentity; glMatrixMode; glPopMatrix; glPushMatrix; glRotatef; glTranslatef; glViewport; gluPerspective; wglCreateContext; wglDeleteContext; wglGetCurrentContext; wglGetCurrentDC; wglMakeCurrent

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

DLLHUSK Sample: Dynamically Links the MFC Library

Download sample

The DLLHUSK sample dynamically links the MFC library to applications and custom dynamic-link libraries (DLL) sharing the same class library code, thus reducing the total memory required by running multiple applications.

Dynamically linking to MFC also offers other possible application architectures in which part of the application is implemented in a custom DLL, and both the application and the custom DLL share the MFC DLL (Mfcxx.dll). A custom DLL that shares framework functionality with an application is called an MFC extension DLL.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the DLLHUSK sample

1. Open the solution dllhusk.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

The DLLHUSK solution builds both the Dllhusk.exe application and the two DLLs (TESTDLL1.DLL and TESTDLL2.DLL) to which the application is dynamically linked. DLLHUSK requires MFCxx.DLL or MFCxxD.DLL at run time. These DLLs are installed by default in the Windows system directory.

DLLHUSK MFC Extension DLLs

DLLHUSK demonstrates MFC extension DLLs with class exports. C++ classes in the DLLs (Testdll1.dll and Testdll2.dll) are exported using the AFX_EXT_CLASS macro. The first MFC Extension DLL (TESTDLL1) is one in which all C++ class interfaces of the custom DLL are accessed only by the framework, not directly by the application. The custom DLL exports only **extern** "C" functions to the application. The custom DLL does not need to export the functions of classes derived from framework classes. All calls from the framework to derived classes in the custom DLL are resolved via the C++ virtual function mechanism.

The second MFC Extension DLL (TESTDLL2) is one in which some C++ class interfaces of the custom DLL are exported to and accessed directly by the application.

Testdll1.dll

Testdll1.dll provides the implementation for DLLHUSK's document and view classes for both document types — the TEXT document type and the HELLO document type. The Dllhusk.exe implements the MDI frame window class, and the framework implements the multiple document interface (MDI) child window class (**CMDIChildWnd**). Two document template objects establish the associations among `CTextDoc`, **CMDIChildWnd**, and **CEditView** and among `CDummyDoc`, **CMDIChildWnd**, and `CHelloView`. DLLHUSK therefore illustrates that the framework can coordinate the relationships among framework-defined objects, even though the classes for those objects are implemented in the application, the custom (MFC extension) DLL, and the framework's Mfcxx.dll.

TESTDLL1 actually calls the application object's **AddDocTemplate** member function twice to register the two document template objects. It does so in the implementation of TESTDLL1's `InitTestDLL1` function, which is the only function that TESTDLL1 exports. This function is declared with **extern** "C" so that the DLLHUSK application can call it as a stand-alone C function.

The Testdll1.h header file (added as a `#include` by Dllhusk.cpp) includes not only the declaration of `InitTestDLL1`, but also the declarations of TESTDLL1's classes. Dllhusk.cpp refers directly to only the `InitTestDLL1` function. Indirectly, however, DLLHUSK uses the document and view classes implemented in Testdll1.dll.

Testdll2.dll

Testdll2.dll provides the implementation for DLLHUSK's **CListOutputFrame** class. When the user chooses a diagnostic command by using the shortcut menu, the application creates a **CListOutputFrame** object and then sends diagnostic

messages to the List Output window by calling **CListOutputFrame::AddString**.

All public member functions of **CListOutputFrame** are exported in the Testdll2.def file. The exports include not only `AddString` but also the public **CListOutputFrame** constructor and destructor.

It is more difficult to implement an MFC extension DLL that exports class member functions than one that exports only C functions. This is true particularly because you must manually add the C++ name-decorated function exports to the DLL's .def file. A technique for doing so is explained in Technical Note 33.

Additional DLLHUSK Features

DLLHUSK also illustrates:

- Splitting resources for a single application into multiple .rc files, which can be edited in the Visual C++ resource editor.
- Using two global diagnostic functions, **AfxDoForAllClasses** and **AfxDoForAllObjects**.
- Enumerating **CDynLinkLibrary** objects.

Keywords

This sample demonstrates the following keywords:

AfxDoForAllClasses; AfxDoForAllObjects; AfxGetApp; AfxGetResourceHandle; AfxMessageBox; AfxSetResourceHandle; AfxThrowMemoryException; CCmdUI::SetCheck; CColorDialog::DoModal; CColorDialog::GetColor; CDC::DrawText; CDC::SetBkColor; CDC::SetTextColor; CDialogBar::Create; CDocTemplate::GetDocString; CEditView::SerializeRaw; CFrameWnd::LoadFrame; CListBox::AddString; CListBox::Create; CListBox::GetCount; CListBox::GetText; CListBox::GetTextLen; CListBox::ResetContent; CListBox::SetCurSel; CMDIChildWnd::Create; CMenu::GetSubMenu; CMenu::LoadMenu; CMenu::TrackPopupMenu; CObject::AssertValid; CObject::Dump; CObject::Serialize; CStatusBar::Create; CStatusBar::SetIndicators; CToolBar::Create; CToolBar::LoadBitmap; CToolBar::SetButtons; CView::OnDraw; CWinApp::AddDocTemplate; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWinApp::OnFileNew; CWinApp::OpenDocumentFile; CWnd::GetClientRect; CWnd::GetCurrentMessage; CWnd::GetFont; CWnd::Invalidate; CWnd::OnCreate; CWnd::OnNcRButtonDown; CWnd::OpenClipboard; CWnd::SetFont; CWnd::ShowWindow; CWnd::UpdateWindow; CloseClipboard; EmptyClipboard; GetModuleFileName; GetSysColor; GlobalAlloc; GlobalLock; LOWORD; RGB; SetClipboardData; lstrlen; wsprintf

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

DLLScreenCap Sample: Demonstrates a Regular DLL That Statically or Dynamically Links to MFC

Download sample

The DLLScreenCap sample illustrates a dynamic-link library (DLL) version of a screen capture tool. DLLScreenCap supercedes the DLLTRACE sample, which is obsolete. DLLTRACE was introduced in MFC version 1.0 to illustrate how to write a DLL that is itself statically linked to the MFC library. DLLScreenCap provides a C-based program interface to the Microsoft Windows–based application to which it is dynamically linked. A DLL that is statically linked to the MFC library cannot successfully export member functions of any classes that are derived from MFC classes.

The technique of statically linking a DLL to the MFC library is discussed in Technical Note 11: Using MFC as Part of a DLL. The DLLScreenCap sample can also be linked to MFC dynamically, without being an extension DLL. Before you decide to implement a custom DLL by statically linking to MFC, consider implementing it as an MFC extension DLL, as explained in Technical Note 33: DLL Version of MFC and as illustrated by the DLLHUSK sample.

A non-extension DLL that is linked to the MFC library needs to have a **CWinApp**-derived class and a single object of that application class, as does an executable MFC application. The **CWinApp** object of the DLL, however, does not have a main message pump, as does the **CWinApp** object of an application. If the DLL opens modeless dialog boxes or has a main frame window of its own, the application's main message pump must call a routine exported by the DLL, which in turn calls the **CWinApp::PreTranslateMessage** member function of the DLL's application object. This is illustrated by the `FilterDllMsg` function exported by DLLScreenCap.dll.

ScreenCapApi.h shows that a way to provide a DLL interface to client applications is to declare functions with **extern** "C". The use of **extern** "C" has several advantages. First, it makes your DLL usable by non-C++ client applications. Second, it reduces DLL overhead, because C++ name decoration will not be applied to the exported name. Third, it makes it easier to explicitly add to a .def file (for exporting by ordinal), without having to worry about C++ name decoration.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

The DLLScreenCap sample consists of two projects: DLLScreenCap, a DLL project, and ScreenCap, an EXE project that calls into the DLL.

Building ScreenCap from the provided solution will automatically build DLLScreenCap and copy the DLL into ScreenCap's output directory.

Note Because this sample uses a custom build step to copy a file, you should not open the solution using a UNC path; put the sample files on a directory with a drive letter.

To build and run the DLLScreenCap sample

1. Open the solution DllScreenCap.sln.
2. In Solution Explorer, right-click the ScreenCap project folder and click **Set as Startup Project** on the shortcut menu.
3. On the **Build** menu, click **Build**.
4. On the **Debug** menu, click **Start Without Debugging**.

The ScreenCap window displays the last screen captured after being scaled to the window's size. Click **Configure Screen Capture** on the **File** menu. This opens a dialog box to specify whether to capture the screen or the active window as well as a path to where the captured file should be saved. Click **Screen Capture** to create a captured file and update the client window display.

Converting DLLScreenCap to Dynamically Link with the MFC DLL

DLLScreenCap demonstrates how to create a regular DLL with an exported function that can be called to display a modal

dialog box. In older versions of Visual C++, this was the only option available for regular DLLs. These kinds of DLLs were formerly known as _USRDLLs.

Now, it is possible for a regular DLL to use MFC from the shared MFCx0 DLL. Because of the potential to reduce the size of the build, you might want to build the DLLScreenCap sample using the shared MFC DLL. Supporting the shared library in DLLScreenCap.dll will reduce the size of a release-build DLL from over 100K to about 16K and will reduce the size of the debug build from over 1 megabyte to 100K. To support linking dynamically in DLLScreenCap, use the macro AFX_MANAGE_STATE to switch the global MFC module state correctly.

To verify that DLLScreenCap can be used in a shared library

1. In Solution Explorer, right-click the ScreenCap project node and click **Properties** on the shortcut menu.

   The **Property Pages** dialog box appears.

2. From the **Configuration** drop-down menu, select **Multiple Configurations**. See Setting Visual C++ Project Properties for more information about the **Property Pages** dialog box.

3. Select both the **Release** and **Debug** builds to change their settings.

4. In the General Property Page for the project, verify that the **Use of MFC** property to use MFC in a shared DLL property is checked.

5. Verify that the following line of code is in the beginning of every function exported from the DLL (see Managing the State Data of MFC Modules for a discussion):

   ```
   AFX_MANAGE_STATE(AfxGetStaticModuleState())
   ```

For example, DllScreenCap.dll exports four functions:

- `CaptureScreen`
- `ConfigureCapture`
- `ProcessDLLIdle`
- `FilterDLLMsg`

`FilterDLLMsg` should look like this in the converted form:

```
BOOL WINAPI FilterDllMsg(LPMSG lpMsg)
{
   AFX_MANAGE_STATE(AfxGetStaticModuleState())
   TRY
   {
       return AfxGetApp()->PreTranslateMessage(lpMsg);
   }
   END_TRY
return FALSE;
}
```

Keywords

This sample demonstrates the following keywords:

CDialog::DoModal, CWinApp::InitInstance, CWinApp::OnIdle, CWinApp::PreTranslateMessage, CWnd::DoDataExchange, CWnd::GetClientRect, CWnd::OnPaint, ShowWindow, UpdateWindow, CImage, AFX_MANAGE_STATE, AfxGetStaticModuleState, CWnd::GetDesktopWindow, CWnd::GetActiveWindow, CImage::Create, CImageDC, CImage::Save, SHBrowseForFolder, CImage::Load, CDC::SetStretchBltMode, CImage::StretchBlt, CWnd::OnEraseBkgnd, CWindowDC

See Also

Other Resources

MFC Samples

MTMDI Sample: Demonstrates an MFC User Interface Thread

Download sample

The MTMDI sample illustrates an MFC user interface thread where user interface events are processed in a separate thread from the main application thread. This sample is a modified version of the single-thread MDI sample.

MTMDI does not claim a strong rationale for putting the bouncing ball window in a separate thread. A user would not be able to detect the difference between the MDI and MTMDI samples on a single-processor computer. Even on a multiprocessor computer, the user would not be able to detect the difference, given that the ball movement is based on a window timer.

Nevertheless, MTMDI does illustrate techniques for implementing an MFC user interface thread. Compare the sources for the MDI and MTMDI samples to study the programming overhead associated with using MFC user interface threads.

The programming overhead in MTMDI should be a warning that you should have a good reason for using a user interface thread to justify the programming cost. The much more common type of thread in MFC is the worker thread, illustrated by the MTRECALC sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the MTMDI sample

1. Open the solution MtMdi.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

You can use the **File** menu to create new windows of two different types. Once the windows are created, the application will let you change the attributes of the items in the window by using commands in one of these menus: **Color**, **Speed**, **Window**, and **Help**. Note that the "Hello!" windows do not have a **Speed** menu.

MTMDI makes use of its own **CWinThread**-derived class, called `CBounceThread`. `CBounceThread` is implemented in the Mtbounce.cpp file. The thread contains all the painting and timing code that the bouncing ball window needs to animate the ball in the window.

The application creates each thread just as the MDI child window is created. This happens in the `CBounceMDIChildWnd::Create` function, found in Bounce.cpp. This function creates both the window and the thread, associating the thread to the window.

Whenever you close a bounce window, the associated thread is automatically destroyed. The "Hello!" windows do not have a thread of their own; their messages are handled by the application's primary thread.

Keywords

This sample demonstrates the following keywords:

AfxGetInstanceHandle; AfxMessageBox; AfxRegisterWndClass; CBitmap::CreateCompatibleBitmap; CCmdTarget::OnCmdMsg; CCmdUI::SetCheck; CColorDialog::DoModal; CColorDialog::GetColor; CDC::BitBlt; CDC::CreateCompatibleDC; CDC::DeleteDC; CDC::DrawText; CDC::Ellipse; CDC::FillRect; CDC::GetDeviceCaps; CDC::SelectObject; CDC::SetBkColor; CDC::SetTextColor; CFrameWnd::LoadFrame; CFrameWnd::rectDefault; CGdiObject::DeleteObject; CMDIChildWnd::Create; CMenu::LoadMenu; CRect::Height; CRect::Width; CWinApp::ExitInstance; CWinApp::InitInstance; CWnd::Create; CWnd::DestroyWindow; CWnd::GetClientRect; CWnd::GetCurrentMessage; CWnd::GetDC; CWnd::GetDlgItem; CWnd::GetWindow; CWnd::Invalidate; CWnd::KillTimer; CWnd::MessageBox; CWnd::OnCreate; CWnd::OnDestroy; CWnd::OnLButtonDown; CWnd::OnPaint; CWnd::OnSize; CWnd::OnTimer; CWnd::ReleaseDC; CWnd::SendMessage; CWnd::SetTimer; CWnd::SetWindowPos; CWnd::ShowWindow; CWnd::UpdateWindow; CloseHandle; CreateEvent; CreateThread; EnumChildWindows; GetSysColor; GetVersion; LOWORD; LoadCursor; LoadIcon; LoadMenu; MAKEINTRESOURCE; RGB; SetEvent; Sleep; WaitForSingleObject; max; min

Note

Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

MTRECALC Sample: Supports Multithread Applications

Download sample

The MTRECALC sample illustrates MFC support for multithread applications.

MFC distinguishes two types of threads: user interface threads and worker threads. A user interface thread has its own message pump for handling user interface events separately from other threads. A worker thread does not have its own message pump. MTRECALC illustrates a worker thread. The worker thread simulates a lengthy calculation by waiting for a timer before completing the calculation of adding two numbers.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the MTRECALC sample

1. Open the solution MtRecalc.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

MTRECALC displays a simple form for adding two integers, simulating a spreadsheet that recalculates whenever you update a cell or click **Recalculate Now**. The result field displays "Recalculating..." during the time MTRECALC is simulating a lengthy calculation. You can change the recalculation time by clicking **Speed of Recalculation** on the **Demo** menu.

Click **Worker Thread** or **Single Thread** on the **Demo** menu to compare the "feel" of recalculation in a separate worker thread versus recalculation in the same thread as the main application user interface thread. For example, in the Single Thread demonstration, you cannot access the menus of MTRECALC while it is recalculating, but in the Worker Thread demonstration you can. In the Single Thread demonstration, you cannot update either of the integer fields while MTRECALC is recalculating, but in the Worker Thread demonstration you can. If you update a field while the worker thread is already recalculating, the active recalculation will be abandoned and a new one started.

The **Kill Worker Thread** command on the **Demo** menu is enabled only while the worker thread is recalculating. If you use the **Kill Worker Thread** command to interrupt the recalculation before it is done, then the result field displays a question mark ("?"). You can restart the calculation with the **Recalculate Now** command.

If you attempt to save the MTRECALC document before recalculation has completed, you will be prompted with "Do you wish to wait while recalculation finishes?" If you respond with yes, MTRECALC displays an hourglass while it waits for the recalculation to complete before saving the document. To exercise this demonstration, you will probably need to use the **Speed of Recalculation** command to increase the recalculation time from the default 5 seconds to 10 or 15 seconds. This will give you enough time to navigate through the **File** menu and **Save** dialog box.

Keywords

This sample demonstrates the following keywords:

AfxBeginThread; AfxGetMainWnd; AfxMessageBox; CArchive::IsStoring; CCmdTarget::BeginWaitCursor; CCmdTarget::EndWaitCursor; CCmdUI::Enable; CCmdUI::SetCheck; CDialog::DoModal; CDocument::GetFirstViewPosition; CDocument::GetNextView; CDocument::OnNewDocument; CDocument::OnSaveDocument; CDocument::SetModifiedFlag; CDocument::UpdateAllViews; CFrameWnd::Create; CFrameWnd::DockControlBar; CFrameWnd::EnableDocking; CFrameWnd::LoadFrame; CObject::AssertValid; CObject::Dump; CObject::Serialize; CStatusBar::CommandToIndex; CStatusBar::SetPaneText; CString::Format; CString::LoadString; CView::DoPreparePrinting; CView::GetDocument; CView::OnBeginPrinting; CView::OnEndPrinting; CView::OnPreparePrinting; CView::OnPrint; CView::OnUpdate; CWinApp::AddDocTemplate; CWinApp::EnableShellOpen; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWinApp::RegisterShellFileTypes; CWnd::DoDataExchange; CWnd::OnCreate; CWnd::OnKillFocus; CWnd::PostMessage; CWnd::SendMessage; CWnd::SetWindowText; CWnd::ShowWindow; CWnd::UpdateData; CWnd::UpdateWindow; CreateEvent; DragAcceptFiles; GetExitCodeThread; GetVersion; LoadBitmap; PostMessage; ResetEvent; SetEvent; Sleep; WaitForSingleObject

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

SPEAKN Sample: Demonstrates Multimedia Sound Using User-Defined Resources

Download sample

The SPEAKN sample brings together graphics output and sound output. The SPEAKN sample illustrates multimedia extensions with the following programming techniques:

- Implementing sound output using the multimedia APIs defined in the Platform SDK MMSYSTEM.H header file.
- Using user-defined resources to store multimedia data like sounds and bitmaps. See Technical Note 35: Using Multiple Resource Files and Header Files with Visual C++ for a discussion of maintaining user-defined resources in a separate .rc file not maintained directly by Microsoft Visual C++.
- Using bitmap buttons. (The CTRLTEST sample application provides a more exhaustive illustration of using **CBitmapButton**.)

A sound card is required if you want to hear the sound output, but you can run the application without a sound card.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the SPEAKN sample

1. Open the solution Speakn.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

When you first run SPEAKN, it plays a "welcome" sound and displays a **What is this** dialog box. When the sound play completes, SPEAKN displays the first of a series of images — a picture of a dog. Type "dog" into the edit box. When you enter the correct word, SPEAKN rewards you with a sound and moves on to the next image.

The happy-face bitmap initially has no smile or frown. If the first character you type is correct, the face turns to a smile; if it is incorrect, the face turns to a frown.

Keywords

This sample demonstrates the following keywords:

AfxGetResourceHandle; AfxMessageBox; CBitmapButton::AutoLoad; CBitmapButton::LoadBitmaps; CDialog::DoModal; CDialog::EndDialog; CDialog::OnInitDialog; CDialog::OnOK; CFont::CreateFontIndirect; CString::GetLength; CString::IsEmpty; CString::LoadString; CString::MakeUpper; CWinApp::InitInstance; CWnd::DoDataExchange; CWnd::EnableWindow; CWnd::GetDlgItem; CWnd::GetWindowText; CWnd::Invalidate; CWnd::SetFocus; CWnd::SetFont; CWnd::SetWindowText; CWnd::ShowWindow; CWnd::SubclassDlgItem; CWnd::UpdateData; CWnd::UpdateWindow; DestroyIcon; FindResource; FreeResource; LoadIcon; LoadResource; LockResource; MAKEINTRESOURCE; PlaySound; PostQuitMessage; lstrcpy; mbstowcs; memset; sndPlaySound; strlen

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

Controls Samples (MFC)

The topics in this section are the abstracts for the MFC ActiveX controls samples. For a list of all MFC samples, see MFC Samples.

An ActiveX control (formerly OLE control) is an ActiveX object with an extended interface that lets it behave like a control for Microsoft Windows.

ActiveX Controls

The following samples demonstrate ActiveX controls:

- BUTTON A control subclassed from a Windows button control. Demonstrates use of an in-place active menu, a stock property page, and the **About** box control option.
- CIRC A simple circle control that illustrates painting, stock properties, custom properties, mouse events, custom events, adding text and fonts, property pages, and data binding.
- CONTAINER A simple example of a Visual Editing container application.
- IMAGE A control that is capable of downloading data asynchronously.
- LICENSED A control that enforces use of a design-time and run-time license.
- LOCALIZE A control with a localized user interface. Demonstrates use of separate type libraries and resource dynamic-link libraries (DLL) for localization.
- REGSVR This utility is used to invoke the self-registration code built into ActiveX Controls to add or remove a control's information in the registry.
- SPINDIAL A control with the visual appearance of a spin-dial. Demonstrates property page data validation.
- TESTHELP A control that has its own help file to display help for the user.
- TIME A control that is invisible at run time and fires a timer event at set intervals. Demonstrates notification functions and ambient properties.
- XLIST A control, subclassed from a Windows list box, that displays text or bitmap items. Demonstrates methods, ambient properties, picture holders, and font holders.

Containers

This section includes the following container sample:

- CONTAINER A simple example of a Visual Editing container application.

Adding ActiveX Controls to the Toolbox

To place ActiveX controls or objects in the **Toolbox** window, see Customize Toolbox Dialog Box. Once inserted, the ActiveX controls can be dragged to the dialog box that you are constructing.

Tip
To quickly add registered ActiveX controls to a dialog box template, see Viewing and Adding ActiveX Controls to a Dialog Box .

Manual Registration of ActiveX Controls

For the following procedure to work, the ActiveX development tools must be installed and the development environment must be open.

1. Open the ActiveX Control Test Container. See Testing Properties and Events with Test Container for information on how to access Test Container.
2. On the **File** menu, click **Register Controls**.

   The **Register Controls** dialog box appears.

3. Click **Register** and browse to the directory in which the ActiveX control is installed.
4. In the **File name** list, double-click the file of the control. Test Container will register the control and add it to the list under **Registered Controls** in the **Register Controls** dialog box.
5. Click **Close** to close the **Register Controls** dialog box. You can now insert your control in Test Container or in any dialog box or form you want.

Testing Controls in Test Container

The ActiveX Control Test Container is a tool for testing and debugging ActiveX controls. You can use it to test a control's functionality by changing the control's properties, invoking its methods, and firing its events. See Testing Properties and Events with Test Container for information on using Test Container.

See Also

Other Resources

MFC Samples

BUTTON Sample: Demonstrates a Menu and a Property Page

Download sample

The BUTTON sample is a control subclassed from a Windows button control. It demonstrates use of an in-place active menu, a stock property page, and the About box control option.

See Knowledge Base article Q199431 for information on enabling menu mnemonics in an MFC ActiveX control.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the BUTTON sample

1. Open the solution button.sln.
2. On the **Build** menu, click **Build**.

See Controls Samples for details on adding a control to the Toolbox, manually registering a control, and using the ActiveX Control Test Container.

Keywords

The BUTTON sample demonstrates the following keywords:

AfxGetInstanceHandle; CDialog::DoModal; CMenu::LoadMenu; HIWORD; MAKELONG

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

CIRC Sample: ActiveX Control

Download sample

The CIRC sample is a simple example of an ActiveX Control.

CIRC illustrates the following:

- Changing the painting behavior of an ActiveX control
- Adding stock properties
- Adding custom properties
- Responding to mouse events
- Adding custom events
- Using text and fonts
- Implementing ActiveX control property pages
- Using simple data binding for control properties

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the CIRC sample

1. Open the solution Circ.sln.
2. On the **Build** menu, click **Build**.

See Controls Samples for details on adding a control to the Toolbox, manually registering a control, and using the ActiveX Control Test Container.

Keywords

The CIRC sample demonstrates the following keywords:

AfxGetInstanceHandle; CDC::Ellipse; CDC::SelectObject; CDC::SelectStockObject; CDialog::DoModal; CString::AllocSysString; CString::GetLength; Ellipse; ExtTextOut; FillRect; GetClientRect; GetDC; GetVersion; MAKELONG; RGB; ReleaseDC; SelectObject; SetBkMode; SetTextAlign; SetTextColor

Note
Some of the samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

CONTAINER Sample: Visual Editing Container Application

Download sample

The CONTAINER sample is a simple example of a Visual Editing container application.

CONTAINER illustrates the following:

- Container starter code created by the application wizard.
- The application-specific classes **CContainerItem** and **CContainerView** and the MFC classes **COleClientItem** and **COleDocument**.
- Selection and hit detection of embedded objects.
- Activation of embedded objects.
- Tracker rectangles using the **CRectTracker** class.

For examples of more advanced Visual Editing container features, see the DRAWCLI and OCLIENT sample applications.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the CONTAINER sample

1. Open the solution container.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

Keywords

The CONTAINER sample demonstrates the following keywords:

AfxMessageBox; AfxOleInit; AfxThrowMemoryException; CArchive::IsStoring; CCmdTarget::BeginWaitCursor; CCmdTarget::EndWaitCursor; CDC::HIMETRICtoDP; CDialog::DoModal; CDocTemplate::SetContainerInfo; CDocument::OnNewDocument; CFrameWnd::Create; CFrameWnd::DockControlBar; CFrameWnd::EnableDocking; CFrameWnd::LoadFrame; CObject::AssertValid; CObject::Dump; CObject::Serialize; COleClientItem::Close; COleClientItem::CreateFromClipboard; COleClientItem::Deactivate; COleClientItem::Delete; COleClientItem::DoVerb; COleClientItem::Draw; COleClientItem::GetActiveView; COleClientItem::GetDocument; COleClientItem::GetInPlaceWindow; COleClientItem::GetItemState; COleClientItem::IsInPlaceActive; COleClientItem::OnChange; COleClientItem::OnChangeItemPosition; COleClientItem::OnDeactivateUI; COleClientItem::OnGetItemPosition; COleClientItem::SetItemRects; COleClientItem::UpdateLink; COleDocument::EnableCompoundFile; COleDocument::GetNextItem; COleDocument::GetStartPosition; COleInsertDialog::CreateItem; COleInsertDialog::DoModal; COleInsertDialog::GetSelectionType; CRect::PtInRect; CRect::SetRect; CRect::Size; CRectTracker::Draw; CRectTracker::GetTrueRect; CRectTracker::SetCursor; CRectTracker::Track; CView::DoPreparePrinting; CView::GetDocument; CView::IsSelected; CView::OnBeginPrinting; CView::OnDraw; CView::OnEndPrinting; CView::OnInitialUpdate; CView::OnPreparePrinting; CView::OnUpdate; CWinApp::AddDocTemplate; CWinApp::EnableShellOpen; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWinApp::RegisterShellFileTypes; CWnd::DoDataExchange; CWnd::Invalidate; CWnd::InvalidateRect; CWnd::OnCreate; CWnd::OnLButtonDblClk; CWnd::OnLButtonDown; CWnd::OnSetCursor; CWnd::OnSetFocus; CWnd::OnSize; CWnd::PreCreateWindow; CWnd::SetFocus; CWnd::ShowWindow; CWnd::UpdateWindow; DragAcceptFiles; GetKeyState

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

IMAGE Sample: Creates an ActiveX Control That Downloads Asynchronously

Download sample

The IMAGE sample demonstrates how to use MFC to build an ActiveX control that is capable of downloading data asynchronously. The control has a property based on the CDataPathProperty class.

The IMAGE control can display a bitmap image from a BMP file (but does not support BMP files with greater than 8-bit color depth, so files containing more than 256 colors cannot be displayed.) The `ImagePath` property of the control provides a path to the image. If the image is large and the download takes place over a slow link, the asynchronous support built into the **CDataPathProperty** object managing the path becomes active and renders the data to the control's painting code without blocking other processing.

The control features a simple property page that gives access to the `ImagePath` property as well as the `AutoSize` property. The `AutoSize` property, if set to **TRUE**, automatically snaps the size of the control to the size of the image it contains. If the `AutoSize` property is **FALSE**, the automatic sizing is disabled.

The image control also implements a read-only Boolean property named `ReadyState`. `ReadyState` is **TRUE** if the control has successfully read and presented all the data in the image stream. `ReadyState` is **FALSE** until the control is fully initialized.

See Controls Samples for details on adding a control to the Toolbox, manually registering a control, and using the ActiveX Control Test Container.

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

See Also

Other Resources

MFC Samples

LICENSED Sample: Enforces Use of Design-Time and Run-Time Licenses

Download sample

The LICENSED sample is a control that enforces use of a design-time and run-time license.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the LICENSED sample

1. Open the solution Licensed.sln.
2. On the **Build** menu, click **Build**.

See Controls Samples for details on adding a control to the Toolbox, manually registering a control, and using the ActiveX Control Test Container.

Keywords

The LICENSED sample demonstrates the following keywords:

AfxGetInstanceHandle; CDC::ExtTextOut; CDC::SelectObject; CDC::SetTextAlign; CDialog::DoModal; CString::GetLength; MAKELONG; MessageBox

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

LOCALIZE Sample: Control with a Localized User Interface

Download sample

The LOCALIZE sample is a control with a localized user interface. It demonstrates use of separate type libraries and resource dynamic-link libraries (DLL) for localization.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the LOCALIZE sample

1. Open the solution Localize.sln.
2. On the **Build** menu, click **Build**.

See Controls Samples for details on adding a control to the Toolbox, manually registering a control, and using the ActiveX Control Test Container.

Keywords

The LOCALIZE sample demonstrates the following keywords:

AfxGetInstanceHandle; AfxGetResourceHandle; AfxSetResourceHandle; CDC::InvertRect; CDialog::DoModal; CString::GetLength; CString::LoadString; ExtTextOut; FreeLibrary; GetClientRect; GetDC; LibMain; LoadLibrary; MAKELONG; ReleaseCapture; ReleaseDC; SelectObject; SetCapture; SetTextAlign

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

REGSVR Sample: Invokes Self-Registration Code

Download sample

The REGSVR sample is the source code to the existing REGSVR32.EXE utility. This utility is used to invoke the self-registration code built into ActiveX controls that adds or removes a control's information in the registry. Developers can use this sample to modify the behavior of the REGSVR32 utility.

REGSVR32.EXE can be found in the \Microsoft Visual Studio .NET 2003\Common7\Tools\Bin directory. For more information on using REGSVR32, see MFC ActiveX Controls: Distributing ActiveX Controls.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building the Sample

To build and run the REGSVR sample

1. Open the solution regsvr.sln.
2. On the **Build** menu, click **Build**.

See Controls Samples for details on adding a control to the Toolbox, manually registering a control, and using the ActiveX Control Test Container.

Keywords

The REGSVR sample demonstrates the following keywords:

FreeLibrary; GetLastError; GetProcAddress; LoadLibrary; LoadString; MessageBox; OleInitialize; OleUninitialize; WinMain; lstrcpy; lstrlen; wsprintf

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

SPINDIAL Sample: Demonstrates Property Page Validation

Download sample

The SPINDIAL sample is a control with the visual appearance of a spin dial. It demonstrates property page data validation.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the SPINDIAL sample

1. Open the solution spindial.sln.
2. On the **Build** menu, click **Build**.

See Controls Samples for details on adding a control to the Toolbox, manually registering a control, and using the ActiveX Control Test Container.

Keywords

The SPINDIAL sample demonstrates the following keywords:

AfxGetInstanceHandle; CBrush::CreateSolidBrush; CDC::Ellipse; CDC::FillRect; CDC::LineTo; CDC::MoveTo; CDC::SelectObject; CDC::SelectStockObject; CDialog::DoModal; MAKELONG

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

TESTHELP Sample: ActiveX Control with Tooltips and Help

Download sample

The TESTHELP sample demonstrates how to use MFC to build an ActiveX control that has its own help file to display help for the user. Testhelp.ocx also shows how to add tooltips to an MFC ActiveX control.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the TESTHELP sample

1. Open the solution Testhelp.sln.
2. On the **Build** menu, click **Build**.

After you have built the TESTHELP sample, you will be able to use it in any ActiveX container application. See Testing Properties and Events with Test Container for information on how to access the test container. Include the TESTHELP control in your container's project and run the project. You can set the properties of the TESTHELP control by opening the property dialog of your container.

The Help project file and all Help project's source files are also included in this sample.

To rebuild and run the TESTHELP help file

1. From the **Start** menu, click **Help Workshop**.
2. Open Testhelp.hpj. Help Workshop displays a wizard to convert the file into an .hhp file. Run the wizard, naming the file and accepting the changes.
3. Click **Compile** on the **File** menu. In the **Create a compiled file dialog box**, indicate where you want to store the compiled file, and then click **Compile**.
4. To view the compiled file, click **Compiled Help File** on the **View** menu.

TESTHELP Features

TESTHELP demonstrates how to:

- Display a tooltip for your ActiveX control.
- Toggle the tooltip on and off using the control's property page.
- Change the tooltip's text using the control's property page.
- Enable the ActiveX control container's property frame Help button to display the control's help file when the Help button is clicked.
- Enable the "?" button in Visual C++'s Component and Controls Gallery so that the control's help file will be displayed when this control is chosen from the Gallery and the "?" button is clicked.
- Display help at run time when the control has the keyboard focus and F1 is pressed.
- Use the Font, BackColor, and ForeColor stock properties of an ActiveX control.

General Property Page

Type of Help to Use

This property determines how help will be used when the user presses F1 while the TESTHELP control has the keyboard focus. Help in a pop-up window means that when the user presses F1 while the TESTHELP control is running in a container

and has the keyboard focus, a pop-up window with the "What is this thing?" topic from the Testhelp.hlp helpfile is displayed. Show the Whole Help File means that when the user presses F1 while the TESTHELP control is running in a container and has the keyboard focus, the entire Windows help program will come up displaying the contents topic of the Testhelp.hlp file.

Use Control's Help File

When this box is checked, help will be displayed in the format determined by the Type of Help to Use property when the user presses F1 while the control is running in a container and has the keyboard focus.

Use ToolTip for this Control

If this box is checked, a tooltip will be displayed over the control's window if the control is running in the container and the mouse cursor moves into and stops in the control's window. The text to be displayed is determined by the Tooltip Caption property.

Tooltip Caption

This is the text to be displayed in a tooltip if the user chooses to show tooltips.

- Name1 The name displayed at the top of the control.
- Name2 The name displayed in the middle of the control.
- Name3 The name displayed at the bottom of the control.

Color Property Page

Lets the user select colors to be used in painting this control.

Font Property Page

Lets the user specify the font used for the text on the control.

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

Keywords

The TESTHELP sample demonstrates the following classes and keywords:

CToolTipCtrl; COleControl::OnDraw; COleControl::OnResetState; COlePropertyPage::OnHelp

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

TIME Sample: Fires a Timer at Set Intervals

Download sample

The TIME sample is a control that is invisible at run time and fires a timer event at set intervals. It demonstrates notification functions and ambient properties.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the TIME sample

1. Open the solution Time.sln.
2. On the **Build** menu, click **Build**.

See Controls Samples for details on adding a control to the Toolbox, manually registering a control, and using the ActiveX Control Test Container.

Keywords

The TIME sample demonstrates the following keywords:

AfxGetInstanceHandle; CBitmap::LoadBitmap; CDialog::DoModal; CGdiObject::GetObject; KillTimer; MAKELONG; SetTimer

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

XLIST Sample: Displays Text or Bitmaps

Download sample

The XLIST sample is a control, subclassed from a Windows list box, that displays text or bitmap items. It demonstrates methods, ambient properties, picture holders, and font holders.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the XLIST sample

1. Open the solution Xlist.sln.
2. On the **Build** menu, click **Build**.

See Controls Samples for details on adding a control to the Toolbox, manually registering a control, and using the ActiveX Control Test Container.

Keywords

The XLIST sample demonstrates the following keywords:

AddItem; AfxGetInstanceHandle; CBrush::CreateSolidBrush; CDC::DrawFocusRect; CDC::ExtTextOut; CDC::FillRect; CDC::InvertRect; CDC::SelectObject; CDC::SelectStockObject; CDC::SetBkMode; CDC::SetTextColor; CDialog::DoModal; CObArray::GetAt; CObArray::GetSize; CObArray::InsertAt; CObArray::RemoveAll; CObject::IsKindOf; COleDispatchDriver::AttachDispatch; COleDispatchDriver::DetachDispatch; COleDispatchDriver::GetProperty; CString::AllocSysString; GetDC; GetDeviceCaps; GetSystemMetrics; HIWORD; LOWORD; MAKELONG; ReleaseDC; SendMessage

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

MFC Samples General Samples

The following topics are the abstracts for the MFC general samples. For a list of all MFC samples, see MFC Samples.

See Also

Other Resources

MFC Samples

Sample CLIPART: Common Resources

Download sample

The CLIPART directory contains sample resources that you can use to customize the appearance of your application. Among the resources included are:

- Dialog resources for the common dialogs.
- A bitmap containing a variety of toolbar buttons.
- Cursor resources for splitting, moving, resizing, and other cursor operations.
- Icon resources for books, clocks, cardfiles, disks, file cabinets, folders, mail, network connections, pencils, phones, traffic signs, and other images.
- String table resources for common status bar prompts and indicators.

These resources are located in a file called common.res. You can use these resource files as is or with your own modifications.

To copy resources from common.res to your own resource script file

1. On the **File** menu, click **Open**, and then click **File**.
2. Navigate to and open an .rc file in an existing or new project.
3. Navigate to and open common.res.
4. Tile the two windows, displaying each file so you can drag and drop between them. On the **Window** menu, click **New Horizontal Tab Group** or **New Vertical Tab Group**.
5. Drag the resources you want from the common.res window to the project .rc window.

For more information about browsing and editing resources with Visual C++, see Resource Editors.

See Also

Other Resources

MFC Samples

CMNCTRL1 Sample: Demonstrates Common Control MFC Classes, Part 1

Download sample

The CMNCTRL1 sample illustrates how to use the following common control MFC classes: **CTreeCtrl**, **CToolBarCtrl**, **CAnimateCtrl**, **CDateTimeCtrl**, and **CMonthCalCtrl**. It shows how the controls are created and how to change the controls using many of the different available styles. Some controls cover topics that often yield questions from the user; for example, the drag and drop implementation for **CTreeCtrl**.

CMNCTRL1 is the result of replacing the original CMNCTRLS sample with two smaller projects: CMNCTRL1 (tree, toolbar, date and time picker, month calendar, and animate controls) and CMNCTRL2 (progress, up/down, and slider controls). This was done to reduce the footprint of the sample code, making it easier to trace specific portions of the application.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the CMNCTRL1 sample

1. Open the solution cmnctrl1.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

When you run CMNCTRL1, select the tab for the control you want to activate. Select the styles and views from the check boxes, radio buttons, and combo boxes available in the different property pages. Note how each option changes the behavior or appearance of the control being displayed. For **CTreeCtrl**, you will be able to see the different notification messages sent by the control as you experiment with it.

Note
Because of a display bug in the Win32 API, the tree view control will not update the node labels when the **TVS_HASLINES** and **TVS_LINESATROOT** styles are dynamically applied. To restore the labels, open and close one of the child nodes.

Keywords

This sample demonstrates the following keywords:

CBitmap::LoadBitmap; CEdit::Clear; CEdit::GetLineCount; CEdit::LineIndex; CEdit::LineLength; CEdit::ReplaceSel; CEdit::SetSel; CFileDialog::GetPathName; CGdiObject::DeleteObject; CImageList::DragShowNolock; CString::GetBufferSetLength; CString::GetLength; CString::Left; CString::LoadString; CString::ReleaseBuffer; CTreeCtrl::SelectDropTarget; CWnd::GetInvalidateRect; CWnd::GetClientRect; CWnd::GetWindowRect; CWnd::SetWindowPos; CWnd::SetWindowText; DeleteItem; GetCursorPos; GetDlgItem; GetParent; GetWindowLong; InvalidateRect; CWinApp::LoadIcon; MAKELONG; MessageBeep; ReleaseCapture; ScreenToClient; SetCapture; SetWindowLong; SetWindowPos; UpdateWindow; CWnd::WindowProc; mbstowcs; rand; srand; time; wsprintf

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

CMNCTRL2 Sample: Demonstrates Common Control MFC Classes, Part 2

Download sample

The CMNCTRL2 sample illustrates how to use the following common control MFC classes: **CProgressCtrl**, **CSliderCtrl**, and **CSpinButtonCtrl**. It shows how the controls are created and how to change the controls using many of the different available styles.

CMNCTRL2 is the result of replacing the original CMNCTRLS sample with two smaller projects: CMNCTRL1 (tree, toolbar, date and time picker, month calendar, and animate controls) and CMNCTRL2 (progress, up/down, and slider controls). This was done to reduce the footprint of the sample code, making it easier to trace specific portions of the application.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the CMNCTRL2 sample

1. Open the solution cmnctrl2.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

When you run CMNCTRL2, select the tab for the control you want to activate. Select the styles and views from the check boxes, radio buttons, and combo boxes available in the different property pages. Notice how each option changes the behavior or appearance of the control being displayed.

Keywords

This sample demonstrates the following keywords:

CBitmap::LoadBitmap; CEdit::Clear; CEdit::GetLineCount; CEdit::LineIndex; CEdit::LineLength; CEdit::ReplaceSel; CEdit::SetSel; CFileDialog::DoModal; CFileDialog::GetPathName; CGdiObject::DeleteObject; CSpinButtonCtrl::GetBuddy; CString::GetBufferSetLength; CString::GetLength; CString::Left; CString::LoadString; CString::ReleaseBuffer; CWinApp::LoadIcon; CWnd::GetClientRect; CWnd::GetWindowRect; CWnd::SetWindowPos; CWnd::SetWindowText; DeleteItem; CWnd::DestroyWindow; CWnd::EnableWindow; GetCursorPos; GetDlgItem; GetParent; GetWindowLong; GetWindowRect; InvalidateRect; LoadIcon; MAKELONG; MessageBeep; ReleaseCapture; ScreenToClient; SetCapture; SetWindowLong; SetWindowPos; UpdateWindow; WindowProc; mbstowcs; rand; srand; time; wsprintf

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

CTRLBARS Sample: Illustrates Custom Control Bars

Download sample

The CTRLBARS sample illustrates a wide variety of customization options for control bars:

- Multiple control bars in a frame window, selective hiding and showing of control bars, and dynamic rearrangement of controls along the border of the frame window. The control bars are allocated space in the frame window according to their Z-order, which initially is the order in which they are created (see **CMainFrame::OnCreate**). CTRLBARS changes the Z-order of the dialog bar with the **CWnd::SetWindowPos** function. It hides or shows a control bar using **CWnd::ShowWindow**. Whenever CTRLBARS changes the Z-order or hides or shows a control bar, it calls **CFrameWnd::RecalcLayout** afterward so that the window real estate is reallocated to the remaining visible control bars.
- Custom toolbars, dynamic rearrangement of buttons on the toolbar, and adding controls (such as a combo box) to a toolbar. CTRLBARS demonstrates two ways of customizing a toolbar. The first toolbar, the Tool Bar, changes the arrangement of buttons between short (5 buttons) and long (10 buttons). CTRLBARS calls **CToolBar::SetButtonInfo** for each button to map it to a tile position on the toolbar's bitmap and to a command identification. The second toolbar, the Style Bar, illustrates replacing a toolbar button (or separator) with a control — a combo box in this example. **CMainFrame::CreateStyleBar** creates a 100-pixel-wide toolbar separator. It then creates the combo box (**IDW_COMBO**) as a child of the toolbar, and sets the position of the combo box to take the space it just allocated for the separator.
- Custom status bar, custom indicators, and sending text to the message line. For status bars, the framework automatically updates a CAP LOCK, NUM LOCK, or SCROLL LOCK indicator if you specify, for example, **ID_INDICATOR_CAPS** as one of the identifications passed in the *indicators*[ ] array to **CStatusBar::SetIndicators**. CTRLBARS illustrates extending the standard status bar indicators with the overstrike (OVR) state, which the user toggles with the INSERT key. This requires a message handler for the **ID_TOGGLE_INSERT** command (mapped to the **VK_INSERT** key), and a resource string (**ID_INDICATOR_OVR**) for the text "OVR" to be displayed when the Overstrike mode is on. The resource identification **ID_INDICATOR_OVR** is a predefined MFC constant.
- Dialog bar, which is a control bar whose layout is defined by a dialog resource template. As with any control bar, notifications from controls in the dialog bar are routed to the owner of the dialog bar, namely the main frame window. For example, **CMainFrame::OnSelChangePalette** is the handler for the **CBN_SELCHANGE** notification from the combo box control in a dialog bar. **CMainFrame** also has handlers for two of the three Hide/Show check boxes (**Styles** and **Palette**). No handler needs to be written for the third check box, **Hide/Show Toolbar**, since the framework provides the standard handler for **ID_VIEW_TOOLBAR**.
- Floating tool palette, which behaves like a toolbar but displays a two-dimensional array of tool buttons and floats as a modeless window above the owner frame window. The floating tool palette is implemented in a reusable class, **CPaletteBar**, derived from **CToolBar**. The derivation from **CToolBar** provides **CPaletteBar** with toolbar button behavior. The palette-specific behavior implemented in **CPaletteBar** includes:
    - Floating (**WS_POPUP**) window style.
    - Two-dimensional arrangement of the tool buttons into rows and columns: creation, painting, and button hit-testing.
    - Thin (no text) title bar.
    - Movable palette using a tracking rectangle.
- CTRLBARS also illustrates using **ON_COMMAND_EX** and **ON_UPDATE_COMMAND_UI_RANGE**. Many of the control bar customization options illustrated by CTRLBARS are discussed in Technical Note 31.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the CTRLBARS sample

1. Open the solution Ctrlbars.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

When you first run CTRLBARS, all the various controls are visible.

- A toolbar with 5 buttons lies immediately below the menu bar. The first button (thick up-arrow) toggles the toolbar arrangement between "Short" (5 buttons) and "Long" (10 buttons). The buttons are always inactive except this first **Short/Long** button and the **Help** button, which opens the **About** box.
- A second toolbar lies immediately below the first toolbar. This toolbar, called the Style Bar, is where you specify one of the following text alignment styles — Left, Centered, Right, or Justified. Selecting one of the styles has no effect other than to change the state of the Style Bar.
- A status bar lies at the bottom of the window.
- A floating palette, with a 3-by-4 array of tool buttons, lies on top of the window.
- A dialog bar lies on the left border of the window. It is a dialog bar because the layout of this control bar is defined in a dialog template resource (**IDD_VIEW_SELECT**).

The **View** menu lets you hide or show any of the first four of the controls bars. The dialog bar is always visible. The hide/show state of the **Tools**, **Styles**, and **Palette** control bars is immediately reflected in the **Hide/Show** check boxes in the dialog bar. You can also hide or show one of the other control bars by toggling its check box.

With the **Dlg Bar Top** command on the **View** menu, you can rearrange the control bars so that the dialog bar is at the top of the Z-order of the control bars. When the dialog bar is at the top, it extends along the entire left border of the window except for the menu and title bars. The left ends of the two toolbars touch the dialog bar. When the dialog bar is returned to its original Z-order position (after all of the other control bars), the top of the dialog bar touches the lower edge of the second toolbar bar, and the bottom of the dialog bar touches the upper edge of the status bar. This reflects the basic algorithm that control bars are allocated window real estate on a first-come, first-serve basis.

The **Style** menu lets you select one of the four text alignment styles — Left, Centered, Right, or Justified. Any selection you make is immediately reflected in the Style Bar's combo box and in the corresponding button. Similarly, you can make a selection either by choosing one of the styles in the Style Bar combo box or by pressing one of the four buttons. The new selection is immediately reflected in the states of the other controls and in the **Style** menu.

The **Palette** menu lets you change the tool arrangement of the palette from 3-by-4 to 2-by-6.

When you select a tool from the **Palette**, the status bar message line shows "You have selected the <*type*> tool," in which <*type*> indicates which of the 12 tools you selected. This selection is reflected in the combo box in the dialog bar. You can also select a tool using this combo box.

The status bar, in addition to showing the most recently selected tool, also shows the status of three keys — INS, CAPS LOCK, and NUM LOCK.

Keywords

This sample demonstrates the following keywords:

AfxFormatString1; AfxGetApp; AfxIsValidAddress; AfxRegisterWndClass; AfxThrowResourceException; BitBlt; CBrush::CreateSolidBrush; CCmdUI::ContinueRouting; CCmdUI::Enable; CCmdUI::SetCheck; CCmdUI::SetText; CComboBox::AddString; CComboBox::Create; CComboBox::GetCurSel; CComboBox::GetLBText; CComboBox::SetCurSel; CControlBar::GetBarStyle; CControlBar::SetBarStyle; CDC::Attach; CDC::Detach; CDC::PatBlt; CDC::RectVisible; CDialogBar::Create; CDumpContext::GetDepth; CFont::CreateFontIndirect; CFrameWnd::LoadFrame; CFrameWnd::RecalcLayout; CFrameWnd::SetMessageText; CGdiObject::Attach; CObject::AssertValid; CObject::Dump; CRect::Height; CRect::InflateRect; CRect::SetRectEmpty; CRect::Width; CStatusBar::Create; CStatusBar::GetPaneInfo; CStatusBar::SetIndicators; CStatusBar::SetPaneInfo; CString::LoadString; CToolBar::CToolBar; CToolBar::CommandToIndex; CToolBar::Create; CToolBar::GetButtonInfo; CToolBar::GetItemID; CToolBar::GetItemRect; CToolBar::LoadBitmap; CToolBar::SetButtonInfo; CToolBar::SetButtons; CToolBar::SetHeight; CToolBar::SetSizes; CWinApp::InitInstance; CWinApp::OnIdle; CWnd::ClientToScreen; CWnd::CreateEx; CWnd::GetCapture; CWnd::GetClientRect; CWnd::GetDC; CWnd::GetDlgItem; CWnd::GetOwner; CWnd::GetParentFrame; CWnd::GetSafeHwnd; CWnd::GetStyle; CWnd::GetWindowRect; CWnd::Invalidate; CWnd::InvalidateRect; CWnd::OnCancelMode; CWnd::OnCreate; CWnd::OnLButtonDown; CWnd::OnMouseActivate; CWnd::OnMouseMove; CWnd::OnSysColorChange; CWnd::PreCreateWindow; CWnd::ReleaseDC; CWnd::SendMessage; CWnd::SetCapture;

CWnd::SetFont; CWnd::SetWindowPos; CWnd::ShowWindow; CWnd::UpdateDialogControls; CWnd::UpdateWindow; CreateBitmap; CreateCompatibleBitmap; CreateCompatibleDC; CreateDIBitmap; CreatePatternBrush; CreatePen; CreateSolidBrush; DeleteDC; DeleteObject; FillRect; FindResource; FrameRect; FreeResource; GetActiveWindow; GetBValue; GetCapture; GetDC; GetDeviceCaps; GetGValue; GetNextWindow; GetObjectType; GetParent; GetRValue; GetStockObject; GetSysColor; GetSystemMetrics; GetVersion; GetWindow; HIBYTE; InvertRect; LOBYTE; LoadCursor; LoadIcon; LoadResource; LockResource; MAKEINTRESOURCE; OffsetRect; PatBlt; RGB; ReleaseCapture; ReleaseDC; SelectObject; SendMessage; SetActiveWindow; SetBkColor; SetCapture; SetROP2; SetRect; SetTextColor; StretchDIBits; UpdateWindow; free; lstrcpy; malloc; memcpy; memset

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

CTRLTEST Sample: Implements Custom Controls

Download sample

The CTRLTEST sample illustrates several techniques for implementing and using custom controls:

- Implementing **CParsedEdit**, a specialized edit control deriving functionality from its library control class, and three methods of using custom controls.
- Using the spin control. The spin control has small up-arrow and down-arrow buttons for incrementing or decrementing a value.
- Bitmap button implementation of **Custom** menu commands using **CBitmapButton**.
- Owner (parent window) drawing of menus and list boxes. Corresponding control classes deriving from the **CMenu** class and **CListBox** provide this feature in an object-oriented way.
- Using resource files not maintainable by the Microsoft Visual C++ resource editors. This illustrates the benefits and drawbacks of using an .rc2 file in a dialog box that has a custom control with styles defined by constants in a header file.

All the illustrations in CTRLTEST are initiated through menu commands.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the CTRLTEST sample

1. Open the solution Ctrltest.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

Example: Implementing and Using Custom Controls

You can implement a custom control by deriving from **CWnd**, but it is much easier if you can borrow functionality from a standard Windows controls by deriving from its control class in the library. CTRLTEST does this to implement a specialized edit control, **CParsedEdit**. This edit control accepts only specified sets of characters as user input: numeric, alphabetic, or noncontrol characters. **CParsedEdit** is derived from **CEdit**. It has an **OnChar** message handler to filter the characters.

The implementation of the commands in the **Simple** menu illustrates three methods of using a custom control. The methods are distinguished according to how the application associates instances of the control in the dialog box with the **CParsedEdit** class. Each **Simple** menu command displays a dialog box with four instances of the **CParsedEdit** control. Data entered in the dialog box is sent to the debug port as **TRACE** output. The three **Simple** menu commands are:

Test C++ Derived Class

The **CParsedEdit** controls are data members of the dialog class. The controls are explicitly created in the dialog's `OnInitDialog` function by calling **CParsedEdit::CreateSet**. See Dertest.cpp.

Test WNDCLASS Registered

The **CParsedEdit** controls are laid out in a dialog template resource (IDD_WNDCLASS_EDIT) as custom controls with a WNDCLASS identified as "paredit." It is instructive to examine the properties of these custom controls using the Visual C++ dialog editor.

- Caption blank. This is the initial value shown in the **CParsedEdit** control.
- Class:paredit. This is the WNDCLASS name that is registered by **CParsedEdit::RegisterControlClass** in PAREDIT2.CPP, before the dialog is called.

- Visible:checked. The control is visible.
- Tabstop:checked. The user can tab to this control.
- Style:0x5081002, 0x5081001, 0x5081003, 0x5081ffff for the four parsed edit controls. The style 0x500000 is for WS_CHILD and WS_VISIBLE, and 0x1000 is for WS_TABSTOP. All custom controls have the WS_CHILD style. The WS_VISIBLE and WS_TABSTOP styles are automatically set by the dialog editor when you check the Visible and Tabstop styles. 0x80000 is for WS_BORDER. Since the dialog editor property page for a custom control does offer all of the window styles such as WS_BORDER, you must look up the constant in \Microsoft Visual Studio .NET 2003\Vc7\PlatformSDK\Include\WINUSER.H. The styles 0x0001, 0x0002, 0x0004, and 0x0ffff are defined in PAREDIT.H as, respectively, PES_NUMBERS, PES_LETTERS, PER_OTHERCHARS, and PES_ALL.
- The hexadecimal styles in the custom control property page are not self-documenting. If it is important to you to use the symbolic styles, such as PES_NUMBERS and PES_LETTERS, you can alternatively manually edit a separate resource file, such as RES\Ctrltest.rc2, that is included by the resource compiler at compile time, but is not read by Visual C++ at editing time. For a discussion of the pros and cons of manually editing a custom control dialog in an .rc2 file, see Using resource files not maintainable by the Visual C++ resource editors.

Test Dynamic Subclassed

The controls are laid out in a dialog template resource (IDD_SUB_EDIT in Ctrltest.rc) as standard edit controls. The controls are declared as **CParsedEdit** data members in the dialog class. The dialog's **OnInitDialog** function calls **CParsedEdit::SubClassEdit**, which in turn calls **CWnd::SubclassDlgItem**, to associate each specific instance of the edit control with the **CParsedEdit** class. See Paredit.cpp.

Example: Spin Control

The CTRLTEST sample includes an implementation of a spin control. The spin control has a small up-arrow and a small down-arrow button for incrementing or decrementing a value.

The **Spin Control** command calls a dialog box that has four **CParsedEdit** controls, each associated with a spin control. Data entered in the **CParsedEdit** controls in this dialog box is filtered to accept only nonnegative integers. The user can enter numeric data by either typing the value into the **CParsedEdit** control or using the associated spin control.

Example: Bitmap Button

The implementation of the following **Custom** menu commands illustrates ways to use **CBitmapButton**:

- **Bitmap Button 1** - The dialog constructor explicitly loads the bitmap resources for each of the three states (up, down, and focus) of the button by calling **CBitmapButton::LoadBitmaps**.
- **Bitmap Button 2** - The dialog's **OnInitDialog** function calls **CBitmapButton::Autoload** to load the bitmap resources based on the following naming convention. The window text of the control serves as the base resource name, and the letters U, D, and F are appended to create the names of the resources for each of the three bitmap images for, respectively, up, down, and focus. For example, the three bitmap resources for the **OK** button are named OKU, OKD, and OKF.
- **Bitmap Button 3** - The dialog box is an extension of the second dialog box above, using a fourth possible button state, disabled. To use this dialog, click the left- or right-arrow bitmap button until the displayed number reaches 1, the lowest number, or 10, the highest number. When the limit is reached, the button is disabled and a fourth bitmap image is displayed. The naming convention for the bitmap resource for the disabled state is an X suffix, as is reflected in the resource names PREVX and NEXTX.

Example: Owner Drawing (Menu and List Box)

Various Windows controls and menus have an owner draw feature that lets the parent (or owner) window draw whatever it wants in the client area of the control instead of the standard control behavior. The corresponding control classes and **CMenu** class provide this feature in a more convenient, object-oriented way: The control or menu class handles the drawing. This is called "self drawing."

CTRLTEST illustrates the general technique of owner drawing in implementing the following commands in the **Custom** menu:

- **Custom Menu** -This menu item calls a **CColorMenu** pop-up menu, derived from **CMenu**. Each submenu item displays one of eight colors using the self-drawing feature. A message box confirms the color you select from the submenu.
- **Custom List Box** - This menu item calls a dialog box that displays a **CColorListBox**, derived from **CListBox**. The list box has eight entries, each drawn in one of eight colors using the self-drawing feature. **TRACE** output confirms your selection

from the list box.

Example: Using Resource Files Not Maintainable by Visual C++ Resource Editors

The resource file CTRLTEST\RES\Ctrltest.rc2 is an example of a resource file not maintainable by the Visual C++ resource editors in a human-readable form. If you were to open Ctrltest.rc2 in Visual C++ and then save it, you would lose useful human-readable information, even though the resource compiler would still be able to compile the .rc2 file and produce an equivalent binary .res file. Thus, RES\Ctrltest.rc2 has been added as a `#include` in Ctrltest.rc with a compile-time directive specified with the **Resource File Set Includes** command.

The following are three categories of human-readable information that are not maintainable by the Visual C++ resource editors. Two of these are demonstrated in Ctrltest.rc2:

- **Custom control styles symbols** - For example, "msctls_updown32" is a style defined for the spin control. Although Visual C++ can interpret this symbol as it reads in the .rc2 file, Visual C++ would write it back out to the .rc2 file as a hexadecimal value.
- **Standard Windows WS_ or control style symbols used in a control from a standard Windows control-derived class** - For example, **ES_AUTOHSCROLL** is defined for the spin control in the IDD_SPIN_EDIT dialog. Although Visual C++ can interpret these symbols as it reads in the .rc2 file, Visual C++ would write it back out to the .rc2 files as a hexadecimal value.
- **Arithmetic in the .rc file** - Expressions such as "IDC_EDIT1+2" to identify controls in the IDD_SPIN_EDIT dialog would be written back out to the .rc2 file as a single hexadecimal value by Visual C++.

The CTRLTEST sample illustrates the pros and cons of using an .rc2 file in the case of a dialog that has a custom control with styles defined with constants in a header file. Both dialogs **IDD_WNDCLASS_EDIT** and **IDD_SPIN_EDIT** have custom controls with symbolically defined styles; but **IDD_WNDCLASS** is specified in a .rc file editable by the Visual C++ dialog editor, whereas **IDD_SPIN_EDIT** is specified in a .rc2 file that is only manually editable.

The differences between using the .rc file and the .rc2 file can be summarized as follows.

For the **IDD_WNDCLASS_EDIT** dialog, the resource script is defined in Ctrltest.rc. For the **IDD_SPIN_EDIT** dialog, the resource script is defined in RES\Ctrltest.rc2. For the **IDD_WNDCLASS_EDIT** dialog, the WNDCLASS custom control is "paredit", the style constants are defined in PAREDIT.H and an example style constant is PES_NUMBER. **IDD_WNDCLASS_EDIT** is editable by Visual C++ but cannot use `#define` styles. **IDD_SPIN_EDIT** is not editable by Visual C++ but can use `#define` styles.

The tradeoff is that if you use the .rc2 file, you can use human-readable symbolic styles defined in the header file for the custom control, but you cannot edit the .rc2 file with the Visual C++ dialog editor. It is easier to lay out the dialog using Visual C++ than it is to manually write resource script; and writing resource script is more error prone. On the other hand, the styles are not self documenting when displayed in hexadecimal in the custom control property page by the Visual C++ dialog editor.

Keywords

This sample demonstrates the following keywords:

AfxGetInstanceHandle; AfxMessageBox; AfxThrowResourceException; CBitmapButton::AutoLoad; CDC::FillRect; CDC::FrameRect; CDialog::DoModal; CDialog::EndDialog; CDialog::OnInitDialog; CDialog::OnOK; CDialog::OnSetFont; CEdit::Create; CEdit::SetSel; CFrameWnd::Create; CListBox::AddString; CListBox::CompareItem; CListBox::DrawItem; CListBox::GetItemData; CListBox::MeasureItem; CMenu::AppendMenu; CMenu::CreateMenu; CMenu::Detach; CMenu::DrawItem; CMenu::EnableMenuItem; CMenu::FromHandle; CMenu::GetMenuString; CMenu::MeasureItem; CRect::Width; CStatic::Create; CString::Format; CString::LoadString; CWinApp::InitInstance; CWnd::Attach; CWnd::EnableWindow; CWnd::FromHandle; CWnd::GetDlgCtrlID; CWnd::GetDlgItem; CWnd::GetDlgItemInt; CWnd::GetMenu; CWnd::GetParent; CWnd::GetWindowRect; CWnd::GetWindowText; CWnd::IsWindowEnabled; CWnd::MessageBox; CWnd::OnChar; CWnd::OnCommand; CWnd::OnVScroll; CWnd::PostNcDestroy; CWnd::SendMessage; CWnd::SetDlgItemInt; CWnd::SetFocus; CWnd::SetFont; CWnd::SetWindowPos; CWnd::ShowWindow; CWnd::SubclassDlgItem; CallWindowProc; GetBValue; GetClassInfo; GetGValue; GetRValue; GetSystemMetrics; HIWORD; IsCharAlpha; IsCharAlphaNumeric; LOWORD; MAKEINTRESOURCE; MAKELONG; MessageBeep; ModifyMenu; RGB; RegisterClass; SetWindowLong

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

DIBLOOK Sample: Illustrates the Use of DIBs and Color Palettes

Download sample

The DIBLOOK sample illustrates the use of device-independent bitmaps (DIB) and the closely related use of color palettes.

DIBLOOK also illustrates a document that has an externally defined file format (in this case, the DIB file format). This is in contrast to an internally defined file format, which is otherwise implied when the framework automatically calls the document's **Serialize** function to store the contents of the document on disk. DIBLOOK further illustrates use of the Clipboard, **CFile**, and scroll views.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the DIBLOOK sample

1. Open the solution DibLook.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

DIBLOOK is a multiple document interface (MDI) application that lets you view multiple bitmaps at the same time. Use **File Open** to open an existing device-independent bitmap (.dib) file or device-dependent bitmap (.bmp) file. Alternately, you can create a new bitmap document by copying a bitmap from another application, such as Paint, using the Clipboard, as follows:

- From the other application, copy a bitmap to the Clipboard.
- Use the DIBLOOK **New** command on the **File** menu to create a new bitmap document.
- Use the **Paste** command on the **Edit** menu to copy the bitmap from the Clipboard into the new document.

Although you cannot edit the image in DIBLOOK, you can save the bitmap to another file by using the **Save As** command on the **File** menu. The bitmap is saved in device-independent bitmap format, even if its original format was device dependent.

Externally Defined Document Format

DIBLOOK reads and stores bitmaps in the standard Windows device-independent bitmap format. While in memory, the bitmap is managed by Windows using an opaque **HDIB** handle. The internal format of the bitmap is visible to the application. Thus, DIBLOOK's document does not itself store the bitmap bits or the color table. Instead, **CDibDoc** holds a handle to the DIB (`HDIB m_hDIB`). DIBLOOK is an application whose document format is defined externally (typically some standard file format standard such as DIB), in contrast to an application whose document format is implicitly defined according to the sequence in which it serializes the document items in the **CDocument::Serialize** override.

DIBLOOK does not override **CDocument::Serialize**. Instead, DIBLOOK overrides **CDocument::OnOpenDocument** and **OnSaveDocument**. Both overrides use the *pszPathName*, passed in by the framework, to open a **CFile** object and to read or save the DIB. The actual code for reading and saving the DIB file is provided in Myfile.cpp; this code is reusable by any application that needs to read and save DIB files.

Using DIBs and Color Palettes

DIBLOOK illustrates how to display a DIB in a window and how to prepare the color palette for the window displaying the DIB.

After DIBLOOK reads the DIB from a file, it prepares a **CPalette** object based on the color table of the DIB and stores the palette as `m_palhDIB` in the **CDibDoc** object. When DIBLOOK displays the DIB in its **CDibView::OnDraw**, it calls a Windows **::PaintDIB** routine implemented in DIBLOOK's Dibapi.cpp file. **PaintDIB** in turn calls the Windows function **::SetDIBitsToDevice** or **::StretchDIBits**, using the color table of the DIB as the color palette. The DIB-displaying routines in Dibapi.cpp are reusable by any application that displays DIBs.

DIBLOOK selects a color palette that is optimal for the currently active window. DIBLOOK selects a color palette matching the

color table of the DIB displayed in the currently active MDI child window. When an application is about to receive the input focus (shifting away from another application), its top-level window receives the **WM_QUERYNEWPALETTE** message. DIBLOOK's **CMainFrame** window handles this message by sending an application-defined message, **WM_DOREALIZE,** to each of the descendant windows. The list of descendant windows includes all the views of the possible multiple opened documents. In turn, each view selects into its display context as a foreground or background palette, depending on whether the view is the active one or not.

If another application changes the system palette, DIBLOOK receives the **WM_PALETTECHANGED** message. In this case, the `CMainFrame` window again sends the application-defined message, **WM_DOREALIZE,** to each view. Each view selects its palette into the display context as a background palette, however, yielding the foreground palette to the other application.

When the focus shifts within the application from one view to another, DIBLOOK selects and realizes the palette for the currently active view (see **CDibView::OnActivateView**). When the new palette is realized, Windows sends the **WM_PALETTECHANGED** message to all application top-level windows, including those of DIBLOOK. DIBLOOK handles the message by realizing as a background palette the color tables associated with the other DIBs in the other views.

Keywords

This sample demonstrates the following keywords:

AfxGetApp; CCmdTarget::BeginWaitCursor; CCmdTarget::EndWaitCursor; CCmdUI::Enable; CDC::RealizePalette; CDC::SelectPalette; CDialog::DoModal; CDocument::DeleteContents; CDocument::OnNewDocument; CDocument::OnOpenDocument; CDocument::OnSaveDocument; CDocument::ReportSaveLoadException; CDocument::SetModifiedFlag; CDocument::SetPathName; CDocument::UpdateAllViews; CFile::Abort; CFile::Close; CFile::GetLength; CFile::Open; CFile::Read; CFile::ReadHuge; CFile::Write; CFile::WriteHuge; CFrameWnd::GetActiveView; CFrameWnd::LoadFrame; CMDIFrameWnd::MDIGetActive; CObject::AssertValid; CObject::Dump; CPalette::CreatePalette; CScrollView::SetScrollSizes; CStatusBar::Create; CStatusBar::SetIndicators; CString::LoadString; CToolBar::Create; CToolBar::LoadBitmap; CToolBar::SetButtons; CView::DoPreparePrinting; CView::GetDocument; CView::OnActivateView; CView::OnDraw; CView::OnInitialUpdate; CView::OnPreparePrinting; CWinApp::AddDocTemplate; CWinApp::EnableShellOpen; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWinApp::RegisterShellFileTypes; CWnd::DoDataExchange; CWnd::OnCreate; CWnd::OnPaletteChanged; CWnd::OnQueryNewPalette; CWnd::OpenClipboard; CWnd::SendMessage; CWnd::SendMessageToDescendants; CWnd::ShowWindow; CWnd::UpdateWindow; CloseClipboard; DragAcceptFiles; EmptyClipboard; GetClipboardData; GetDeviceCaps; GlobalAlloc; GlobalFree; GlobalLock; GlobalSize; GlobalUnlock; IsClipboardFormatAvailable; MessageBox; SelectPalette; SetClipboardData; SetDIBitsToDevice; SetStretchBltMode; StretchDIBits; memcpy

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

DLGCBR32 Sample: Demonstrates Adding a Status Bar and Toolbar to Dialog Boxes

Download sample

The DLGCBR32 sample demonstrates how to add a status bar and a toolbar to a dialog box. In addition, it shows a number of techniques related to using a modeless dialog box as the main window of an MFC application.

In an MFC application, you can attach control bars such as status bars and toolbars to a frame window. However, for many applications a simple dialog box-based user interface is sufficient. MFC does not provide built-in support for adding control bars to dialog boxes.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the DLGCBR32 sample

1. Open the solution Dlgcbr32.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

Adding a Control Bar to a Dialog Box

To add a control bar to a dialog box, create the control bar as usual, and then make room for the control bar within the client area of the dialog box. For the control bar to function properly, the dialog box must duplicate some of the functionality of a frame window. If you want **ON_UPDATE_COMMAND_UI** handlers to work for the control bars, you also need to derive new control bar classes and handle the **WM_IDLEUPDATECMDUI** message. If your dialog box is not the main window of your application, you will also need to modify its parent frame window to pass the **WM_IDLEUPDATECMDUI** message on to the dialog box's control bars.

To make room for a control bar within the client area of the dialog box, follow these steps in your dialog box's **OnInitDialog** function:

1. Create the control bars. Figure out how much room the control bars will take by using the `reposQuery` option of `RepositionBars`.

   ```
   CRect rcClientStart;
   CRect rcClientNow;
   GetClientRect(rcClientStart);
   RepositionBars(AFX_IDW_CONTROLBAR_FIRST,
                  AFX_IDW_CONTROLBAR_LAST,
                  0, reposQuery, rcClientNow);
   ```

2. Move the controls in your dialog box to account for the space used by control bars at the top or left of the client area. If your dialog box contains a menu, you also need to account for the space used by the menu.

   ```
   CPoint ptOffset(rcClientNow.left - rcClientStart.left,
                   rcClientNow.top - rcClientStart.top);
   CRect rcChild;
   CWnd* pwndChild = GetWindow(GW_CHILD);
   while (pwndChild)
       {
   ```

```
            pwndChild->GetWindowRect(rcChild);
            ScreenToClient(rcChild);
            rcChild.OffsetRect(ptOffset);
            pwndChild->MoveWindow(rcChild, FALSE);
            pwndChild = pwndChild->GetNextWindow();
        }
```

3. Increase the dialog box window dimensions by the amount of space used by the control bars.

```
CRect rcWindow;
GetWindowRect(rcWindow);
rcWindow.right += rcClientStart.Width() - rcClientNow.Width();
rcWindow.bottom += rcClientStart.Height() - rcClientNow.Height();
MoveWindow(rcWindow, FALSE);
```

4. Position the control bars using `RepositionBars`.

To update the first pane of a status bar with menu item text, you must handle **WM_MENUSELECT**, **WM_ENTERIDLE**, **WM_SETMESSAGESTRING**, and **WM_POPMESSAGESTRING** in your dialog box class. You need to duplicate the functionality of the **CFrameWnd** handlers for these messages. See the **CModelessMain** class in the sample program for examples of these message handlers.

To display tooltips for the toolbar buttons, it is necessary to handle the **TTN_NEEDTEXTW** and **TTN_NEEDTEXTA** notifications.

To allow **ON_UPDATE_COMMAND_UI** handlers to work for other status bar panes and for toolbar buttons, you must derive new control bar classes and implement a message handler for **WM_IDLEUPDATECMDUI**. This is necessary because the default control bar implementations of **OnUpdateCmdUI** assume the parent window is a frame window. However, **OnUpdateCmdUI** does not do anything but pass the parent-window pointer to a function that requires only a **CCmdTarget** pointer. Therefore, you can temporarily tell **OnUpdateCmdUI** that the parent-window pointer you are giving it is a **CFrameWnd** pointer to meet the compiler requirements. For example:

```
LRESULT CDlgToolBar::OnIdleUpdateCmdUI(WPARAM wParam, LPARAM lParam)
{
   if (IsWindowVisible())
   {
      CFrameWnd* pParent = (CFrameWnd*)GetParent();
      if (pParent)
         OnUpdateCmdUI(pParent, (BOOL)wParam);
   }
   return 0L;
}
```

To pass **WM_IDLEUPDATECMDUI** messages on to dialog boxes other than the main window, save dialog pointers in your frame window class and create a **WM_IDLEUPDATECMDUI** handler in that class. The handler should send the **WM_IDLEUPDATECMDUI** message on to the dialog child windows by using **CWnd::SendMessageToDescendants**. Then perform default processing for the message within the frame window.

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

DLGTEMPL Sample: Creates Dialog Templates Dynamically

Download sample

The DLGTEMPL sample shows how to create a dialog template dynamically and use the template with CDialog::InitModalIndirect. The sample displays a dialog box for selecting the height, width, and other attributes of a button, static text, and multiline edit control, then displays the resulting dialog box using templates in memory.

The bulk of the code for dynamically creating a dialog is found in the **CMyDialogTemplate::DemoIt** member function. DLGTEMPL does not prevent you from entering values that lay one control over another, nor does it prevent you from positioning a control beyond the boundary of the dialog. You can experiment with DLGTEMPL to see the effects of resizing and overlaying controls.

For more information on dialog templates, see the class CDialog and the Windows structures DLGTEMPLATE and DLGITEMTEMPLATE in the *Platform SDK*.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the DLGTEMPL sample

1. Open the solution dlgtempl.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

When you run DLGTEMPL, it displays a dialog box in which you select **Button**, **Edit**, or **Text**. Enter the desired values for height, width, and X and Y coordinates for placement on the resulting dialog box, and a caption to be displayed. Click **Show Dialog** to display a dynamically created dialog box based on the values you enter. Press ESCAPE to close the dialog box.

See Also

Other Resources

MFC Samples

DOCKTOOL Sample: Demonstrates Dockable Toolbars

Download sample

The DOCKTOOL sample demonstrates support for dockable toolbars. A dockable toolbar can be attached, or docked, to any side of its parent window, or it can be detached, or floated, in its own miniframe window using **CMiniFrameWnd**.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the DOCKTOOL sample

1. Open the solution docktool.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

Supporting Dockable Toolbars

The following three steps are necessary to place a dockable toolbar in your application:

1. Enable docking for the frame window or destination using the **CFrameWnd::EnableDocking** function. One **DWORD** parameter indicates which side of the frame window accepts docking. To dock control bars anywhere, pass **CBRS_ALIGN_ANY** to **EnableDocking**.
2. Enable docking for the toolbar or source by calling **CControlBar::EnableDocking** for each toolbar. Specify the destination sides to which the toolbar should dock. If none of the sides specified match the sides enabled for docking in the frame window, the toolbar cannot dock; it will float. Once it has been floated, it remains a floating toolbar, unable to dock to the frame window.
3. Dock the toolbar to the frame window by calling **CFrameWnd::DockControlBar**. Conversely, call **CFrameWnd::FloatControlBar** to detach a dockable toolbar from the frame window.

If you do not complete all three steps, your application will display a standard toolbar. The last two steps must be performed for each dockable toolbar in your application.

To retain the state of dockable toolbars (whether docked or floating) between invocations of your application, use the **CControlBar::SetBarStyle** and **CControlBar::GetBarStyle** functions to retrieve and restore the settings of any particular control bar. Normally, this information is stored in the application's .ini file using **CFrameWnd::SaveBarState** and retrieved using **CFrameWnd::LoadBarState**.

Keywords

This sample demonstrates the following keywords:

AfxGetApp; CArchive::IsStoring; CComboBox::Create; CComboBox::DeleteString; CComboBox::GetCount; CComboBox::InsertString; CComboBox::SetCurSel; CControlBar::GetBarStyle; CControlBar::SetBarStyle; CDialog::DoModal; CDocument::OnNewDocument; CFrameWnd::Create; CFrameWnd::DockControlBar; CFrameWnd::EnableDocking; CFrameWnd::LoadBarState; CFrameWnd::RecalcLayout; CFrameWnd::SaveBarState; CMenu::GetSubMenu; CMenu::LoadMenu; CMenu::TrackPopupMenu; CObject::AssertValid; CObject::Dump; CObject::Serialize; CRect::OffsetRect; CString::IsEmpty; CToolBar::Create; CToolBar::GetItemRect; CToolBar::LoadBitmap; CToolBar::SetButtonInfo; CToolBar::SetButtons; CView::GetDocument; CView::OnDraw; CWinApp::AddDocTemplate; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWinApp::OnFileNew; CWnd::ClientToScreen; CWnd::DoDataExchange; CWnd::FromHandlePermanent; CWnd::GetParentFrame; CWnd::GetStyle; CWnd::GetWindowPlacement; CWnd::GetWindowRect; CWnd::GetWindowText; CWnd::Invalidate; CWnd::IsZoomed; CWnd::OnClose; CWnd::OnCreate; CWnd::PreTranslateMessage; CWnd::SendMessage; CWnd::SetWindowPlacement; CWnd::SetWindowPos; CWnd::SetWindowText; CWnd::ShowWindow; GetProfileInt; GetProfileString; GetStockObject; HIWORD; LOWORD; LoadBitmap; WriteProfileString; wsprintf

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

DYNAMENU Sample: Dynamically Updates Menus

Download sample

The DYNAMENU sample illustrates dynamic modification of menus and status bars regardless of whether handling commands are known at compile time. DYNAMENU illustrates the following capabilities:

- Dynamically updating the list of items in a menu.
- Implementing the equivalent of **ON_COMMAND** and **ON_UPDATE_COMMAND_UI** handlers for menu commands whose IDs are not known at compile time. This illustration can be applied to more complex cases, such as user-configurable menus.
- Updating the status bar command prompt for commands whose IDs are not known at compile time.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the DYNAMENU sample

1. Open the solution dynamenu.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

DYNAMENU initially displays a window with the text: "This text is displayed in the current color." You can change the color of the displayed text by selecting one of four items initially offered in the **Color** menu: Black, Red, Purple, or Blue.

To exercise the dynamic menu updating feature of DYNAMENU, click **Change Options** on the **Color** menu, which opens a **Change Color Options** dialog box. Check boxes for Black, Red, Purple, and Blue allow you to choose which colors are dynamically offered in the **Color** menu. For example, if you clear Red and Purple and return to the **Color** menu, only Black and Blue items will be offered in the menu.

Note how the status bar displays, for example, "Set current color text to Black," when the focus is on the Black item in the **Color** menu.

Dynamically Updating the List of Items in a Menu

Class **CDynaMDIChildWnd** in Mdichild.cpp implements dynamic updating of items in the **Color** menu. When the list of available colors is updated, or when then MDI child window is activated, the **CDynaMDIChildWnd::RefreshColorMenu** function calls **CMenu::DeleteMenu** to delete each color item from the menu, and then adds the currently available colors to the menu using **CMenu::AppendMenu**.

Implementing Command Handlers for Dynamic Menu Items

DYNAMENU could have been implemented by reserving a fixed list of command IDs for the colors: **ID_COLOR_BLACK**, **ID_COLOR_RED**, and so on. In such a case, **ON_COMMAND** and **ON_UPDATE_COMMAND_UI** handlers could have been implemented for the color commands as usual. This would be the most straightforward way to implement DYNAMENU.

However, for sake of illustration, DYNAMENU does not use fixed command IDs. Instead, DYNAMENU dynamically assigns command IDs not known or associated with the menu items at compile time. This illustration can be applied to more complex cases, such as user-configurable menus.

The equivalent of **ON_COMMAND** and **ON_UPDATE_COMMAND** message map handling is implemented in the document's override of **CCmdTarget::OnCmdMsg**. If the **OnCmdMsg** function is called with a NULL pointer for the **AFX_CMDHANDLERINFO*** parameter, this means that no message map entry has been found for the command. In this case, the override of **OnCmdMsg** checks whether the command ID, passed as the first parameter, is one of the dynamically assigned command IDs for the color menu items. If so, the override calls either a command handler (**DoSelectColor**) or command user

interface handler (**DoUpdateSelectColor**), depending on whether the second parameter passed to **OnCmdMsg** is the MFC-defined **CN_COMMAND** or **CN_UPDATE_COMMAND_UI**.

Updating Status Bar Command Prompt for Dynamic Menu Items

In DYNAMENU, the MDI child window (**CDynaMDIChildWnd**) owns the status bar. The default implementation of **CFrameWnd::GetMessageString** uses the currently shown command ID (for the item that currently has the focus in the menu) to get the corresponding string resource for the command and display it in the first pane of the status bar. DYNAMENU overrides **GetMessageString** to display a command prompt for the dynamically defined commands.

Keywords

This sample demonstrates the following keywords:

AfxFormatString1; AfxGetMainWnd; CCmdTarget::OnCmdMsg; CDialog::DoModal; CDialog::OnInitDialog; CDocument::GetFirstViewPosition; CDocument::GetNextView; CDocument::OnNewDocument; CDocument::UpdateAllViews; CFrameWnd::Create; CFrameWnd::GetActiveDocument; CFrameWnd::GetMessageString; CFrameWnd::LoadFrame; CMenu::AppendMenu; CMenu::DeleteMenu; CMenu::GetMenuItemCount; CMenu::GetMenuItemID; CMenu::GetSubMenu; CObject::AssertValid; CObject::Dump; CObject::Serialize; CString::LoadString; CView::DoPreparePrinting; CView::GetDocument; CView::OnBeginPrinting; CView::OnDraw; CView::OnEndPrinting; CView::OnPreparePrinting; CWinApp::AddDocTemplate; CWinApp::EnableShellOpen; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWinApp::RegisterShellFileTypes; CWnd::CenterWindow; CWnd::DoDataExchange; CWnd::GetClientRect; CWnd::GetDlgItem; CWnd::GetMenu; CWnd::GetParentFrame; CWnd::OnCreate; CWnd::SetWindowText; CWnd::ShowWindow; CWnd::UpdateWindow; DragAcceptFiles; DrawText; LoadBitmap; RGB; SetBkMode; SetTextColor

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

GUIDGEN Sample: Generates Globally Unique Identifiers (GUIDs)

Download sample

The GUIDGEN sample is a simple dialog-based MFC application that can help you as you code ActiveX applications. GUIDGEN can be used to generate globally unique identifiers, or GUIDs, that identify classes, objects, and interfaces.

Aside from being a dialog-box based application, GUIDGEN also shows how MFC applications can paste text data to the Clipboard. The **CGuidGenDlg::OnOK** function in the GUIDGDLG.CPP file shows how **OpenClipboard** and **SetClipboardData** can be called to accomplish this task. The application also demonstrates calling the **CoCreateGuid** API to request a new GUID from the operating system. Look for this code in **CGuidGenDlg::OnNewguid**, which is also in the GUIDGDLG.CPP file.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Running the Sample

If you have installed the Microsoft Visual C++ product, the GUIDGEN utility (GUIDGEN.EXE) is installed by default in \\Microsoft Visual Studio .NET 2003\Common7\Tools.

The radio buttons in the **GUID Format** group box determine the format of the GUID the program will generate.

- The first format is designed to be used in invocations of MFC's IMPLEMENT_OLECREATE macro.
- The second format generates a reference to the DEFINE_GUID macro.

The remaining formats are appropriate for statically allocated GUIDs and GUIDs included in registry entries or registry editor scripts.

Once you have selected the format for your GUID, you can copy it to the Clipboard with the **Copy** button. You can return to your editor and paste the GUID code into your application's source. When you need to generate a new GUID, click the **New Guid** button.

Keywords

This sample demonstrates the following keywords:

AfxGetApp; AfxMessageBox; CDC::DrawIcon; CDC::GetSafeHdc; CDialog::DoModal; CDialog::EndDialog; CDialog::OnInitDialog; CDialog::OnOK; CMenu::AppendMenu; CRect::Height; CRect::Width; CString::Format; CString::GetLength; CString::IsEmpty; CString::LoadString; CWinApp::ExitInstance; CWinApp::InitInstance; CWinApp::SetRegistryKey; CWnd::CenterWindow; CWnd::DoDataExchange; CWnd::GetClientRect; CWnd::GetParent; CWnd::GetSystemMenu; CWnd::IsIconic; CWnd::OnPaint; CWnd::OnQueryDragIcon; CWnd::OnSysCommand; CWnd::OpenClipboard; CWnd::SendMessage; CWnd::SetDlgItemText; CWnd::UpdateData; CloseClipboard; CoInitialize; CoUninitialize; EmptyClipboard; GetForegroundWindow; GetProfileInt; GetSystemMetrics; GlobalAlloc; GlobalLock; GlobalUnlock; LoadIcon; SetClipboardData; memcpy

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

MAKEHM Sample: Utility for Associating Resources with Help Contexts

Download sample

The MAKEHM sample is a console application that produces a mapping between resource identifications and Help contexts. The sources are provided for MAKEHM to serve as an illustration of a console application that uses MFC and to enable you to modify the tool.

A console application such as MAKEHM is linked with the same variant of the MFC library as graphical user interface (GUI) applications.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Running the Sample

If you have installed the Microsoft Visual C++ product, the MAKEHM utility (MAKEHM.EXE) is installed by default in \\Microsoft Visual Studio .NET 2003\Common7\Tools.

The command line syntax to run MAKEHM directly is:

```
makehm <from>,<to>,<add> [<from>, <to>,<add> [...]] <resource.h> >> [output.hm]
```

`<from>`

Specifies the symbol prefix to be read, such as a command ID.

```
<to>
```

Specifies the symbol prefix of the Help context identification to be produced in the .hm file.

```
<add>
```

The Help ID base number to be added to the value of the `<from>` identification to produce the Help context number.

You can specify one or more sets of `<from>`,`<to>`,`<add>` in the command line.

```
<resource.h>
```

The header file that provides `#define` definitions for the symbols to be read.

```
<output.hm>
```

The .hm file to be produced by MAKEHM.

Look at the custom build steps for the resource.h file to learn what command line values to use.

Keywords

This sample demonstrates the following keywords:

AfxThrowMemoryException; CFile::Close; CFile::GetPosition; CFile::Open; CFile::Seek; CStdioFile::ReadString; CStdioFile::WriteString; CString::GetBuffer; CString::GetLength; CString::IsEmpty; CString::ReleaseBuffer; CString::Right; __iscsym; __iscsymf; _strdup; exit; fprintf; free; isdigit; isxdigit; sprintf; strchr; strcmp; strlen; strncmp; strstr; strtok

Note

Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

MDI Sample: Demonstrates MDI Without Using Doc/View Architecture

Download sample

The MDI sample uses MFC's multiple-document interface (MDI) support without using the document/view architecture.

For an additional MDI sample, see MDIDOCVW.

MDI also illustrates the Microsoft Windows timer, **CColorDialog**, **CBitmap**, and changing the default cursor of a window.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the MDI sample

1. Open the solution mdi.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

The MDI sample application provides a parent window with two distinct types of MDI child windows: a Bounce window, in which a ball bounces around; and a Hello window, which displays the text "Hello, World!" You can create multiple instances of each type of window.

Click **New Bounce** or **New Hello** on the **File** menu to create a new MDI child window. Both MDI child windows determine what menus are shown when they are active. Both windows have a **Color** menu. When you choose a color from this menu, the color of the bouncing ball or of the Hello text is updated. Choose the Custom item to call a color dialog box, from which you can select a special color.

The Bounce window also has a **Speed** menu, from which you can select the speed at which the ball moves around the screen.

If you minimize the Bounce window, notice that there is not a static icon. Rather, a small ball bounces around in the icon. In contrast, the Hello window has a normal icon.

MDI Without Documents and Views

Unlike an MDI application produced by the application wizard, this sample application does not use the framework's support for documents and views. Thus, it does not use document templates; the application's **InitInstance** does not call **AddDocTemplate**.

This application nevertheless takes full advantage of the framework's MDI support. **CMainFrame** is derived from **CMDIFrameWnd. CBounceWnd** and **CHelloWnd** are derived from **CMDIChildWnd**. Commands on the **Window** menu, such as **Tile**, are handled by the framework's default implementation in **CMDIFrameWnd**.

Although **CBounceWnd** and **CHelloWnd** both override **Create**, the override is not necessary to take advantage of the framework's MDI support. The override of **Create** illustrates how to change the default cursor and icon of a window. Changing the default cursor or icon of a window requires registering a new **WNDCLASS** by calling **AFXRegisterWndClass** and passing the name of the **WNDCLASS** in the call to **Create**.

Keywords

This sample demonstrates the following keywords:

AfxGetInstanceHandle; AfxRegisterWndClass; CBitmap::CreateCompatibleBitmap; CCmdUI::SetCheck; CColorDialog::DoModal; CColorDialog::GetColor; CDC::BitBlt; CDC::CreateCompatibleDC; CDC::DeleteDC; CDC::DrawText; CDC::Ellipse; CDC::FillRect; CDC::GetDeviceCaps; CDC::SelectObject; CDC::SetBkColor; CDC::SetTextColor; CFrameWnd::LoadFrame; CFrameWnd::rectDefault; CGdiObject::DeleteObject; CMDIChildWnd::Create; CMenu::LoadMenu; CWinApp::InitInstance;

CWnd::DestroyWindow; CWnd::GetClientRect; CWnd::GetCurrentMessage; CWnd::GetDC; CWnd::Invalidate; CWnd::KillTimer; CWnd::MessageBox; CWnd::OnCreate; CWnd::OnPaint; CWnd::OnSize; CWnd::OnTimer; CWnd::ReleaseDC; CWnd::SetTimer; CWnd::ShowWindow; CWnd::UpdateWindow; GetSysColor; LOWORD; LoadCursor; LoadIcon; MAKEINTRESOURCE; RGB; max; min

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

MDIDOCVW Sample: Demonstrates MDI Using Doc/View Architecture

Download sample

The MDIDOCVW sample uses MFC's multiple-document interface (MDI) support and the document/view architecture. The sample contains two document types: Hello (which prints out a string in a view) and Bounce (which displays a bouncing ball of color in a view). MDIDOCVW also illustrates the Microsoft Windows timer, **CColorDialog**, **CBitmap**, and changing the default cursor of a window.

The MDIDOCVW sample provides a parent window with two distinct types of MDI child windows: a Bounce window, in which a ball bounces around; and a Hello window, which simply displays the text "Hello, World!"

For a sample that uses MDI support without the document/view architecture, see MDI.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the MDIDOCVW sample

1. Open the solution mdi.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

Click **New Bounce** or **New Hello** on the **File** menu to create a new MDI child window. Both types of MDI child windows determine what menus are shown when they are active. Both types of windows have a **Color** menu. When you choose a color from this menu, the color of the bouncing ball or of the Hello text is updated. Click **Custom** to call a **Color** dialog box, from which you can select a custom color.

The Bounce window also has a **Speed** menu, from which you can select the speed the ball moves around the screen.

A toolbar implements most of the menu commands. The toolbar also implements check box buttons for the five basic colors.

Changing the Default Cursor

Although **CBounceView** and **CHelloView** both override **Create**, the override is not necessary to take advantage of the framework's MDI support. The override of **Create**, found in the **CBounceView** class, illustrates how to change the default cursor. Changing the default cursor of a window requires registering a new **WNDCLASS** by calling **AfxRegisterWndClass** and passing the name of the **WNDCLASS** in the call to **Create**.

Keywords

This sample demonstrates the following keywords:

AfxRegisterWndClass; CBitmap::CreateCompatibleBitmap; CCmdUI::SetCheck; CColorDialog::DoModal; CColorDialog::GetColor; CDC::BitBlt; CDC::CreateCompatibleDC; CDC::DeleteDC; CDC::DrawText; CDC::Ellipse; CDC::FillRect; CDC::GetDeviceCaps; CDC::SelectObject; CDC::SetBkColor; CDC::SetTextColor; CFrameWnd::LoadFrame; CFrameWnd::rectDefault; CGdiObject::DeleteObject; CMDIChildWnd::Create; CWinApp::InitInstance; CWnd::DestroyWindow; CWnd::GetClientRect; CWnd::GetCurrentMessage; CWnd::GetDC; CWnd::KillTimer; CWnd::MessageBox; CWnd::OnCreate; CWnd::OnSize; CWnd::OnTimer; CWnd::ReleaseDC; CWnd::SetTimer; CWnd::ShowWindow; CWnd::UpdateWindow; GetSysColor; LOWORD; RGB; max; min

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

MMXSwarm Sample: Demonstrates CImage and Visual C++ MMX Support

Download sample

MMXSwarm is an MFC sample that demonstrates how to use the MFC CImage class, the __m64 and __m128i data types, and device-independent bitmaps (DIB).

The sample demonstrates the implementation of **CImage** support into an application for basic loading and saving of data in many image formats. In addition, the sample demonstrates direct manipulation of a DIB surface, as well as using higher-level compiler support for MMX and SSE2 integer instructions to optimize bitmap manipulation without writing assembler code.

In the project, the Surface.* files contain the C++ base class for the DIB surface manipulation:

- The MMXSurface*.* files contain derived classes with MMX optimizations. MMXWrapper.h contains a simple class that encapsulates the compiler's **__m64** data type and operations into a C++ friendly class for 16-bit saturated MMX operations.
- The SSE2Surface*.* files contain derived classes with SSE2 optimizations. SSE2Wrapper.h contains a simple class that encapsulates the compiler's **__m128i** data type and operations into a C++ friendly class for 16-bit saturated SSE2 integer operations.

These wrapper classes are not generally useful "as is", containing only enough functionality for the sample. The sample does not use the document/view archictecture. The `CChildView` class is derived from CWnd and bound to the frame window.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the MMXSwarm sample

1. Open the solution MMXSwarm.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

In both Debug and Release mode, assembly listings of the MMXSurface*.cpp and SSE2Surface*.cpp files are generated in the output directory. Examine these files to see the code generated by the MMX and SSE2 portions of the project.

Build the sample in Release mode to see the speed improvements gained from using the intrinsic MMX and SSE2 integer support. In addition, debug information is generated for the release build to ease stepping through the MMX and SSE2 generated code.

For best results, run the sample in 32-bit color mode.

The sample draws a simple swarm on the screen and uses a custom fade routine to blur the image over time. The **File** menu has the options for saving the current image or loading an existing image. The **View** menu has options to disable/enable the blurring and the swarm. The **View** menu also contains a 'Pause Blit' command that will disable the BitBlt to the screen that enables better frame rate distinctions in the blur implementations. In addition, several keystrokes are asynchronously checked for other DIB manipulation (SHIFT, SPACE, UP ARROW, and DELETE). Finally, the **CImage** menu lets you select the bit depth of the DIB being manipulated. Typically, it is best if it matches the desktop bit depth. Each bit depth has an option for blurring with a generic C++ routine, or one optimized with the MMX or SSE2 integer intrinsics. The status bar contains a frame-per-second measurement to illustrate the performance differences.

Keywords

This sample demonstrates the following keywords:

CImage::GetExporterFilterString, CImage::Load, CImage::SaveAdvanced, CImage::Create, CImage::GetPitch, CImage::GetDC,

CImage::ReleaseDC, CImage::Destroy, CImage::BitBlt, CImage::GetBits, CImage::GetHeight, CImage::GetPixelAddress, GetAsyncKeyState, CDC::GetDeviceCaps, ON_UPDATE_COMMAND_UI_RANGE, CWnd::OnSizing, __m64, __m128i, CWinApp::OnIdle, _m_from_int, _mm_setzero_si64, _mm_adds_pu16, _mm_subs_pu16, _mm_srli_pi16, _mm_slli_pi16, _mm_and_si64, _mm_or_si64, _mm_andnot_si64, _m_to_int, _mm_packs_pu16, _mm_unpacklo_pi8, _mm_unpackhi_pi8, _mm_loadl_epi64, _mm_setzero_si128, _mm_set1_epi64, _mm_adds_epu16, _mm_subs_epu16, _mm_srli_epi16, _mm_slli_epi16, _mm_and_si128, _mm_or_si128, _mm_andnot_si128, _mm_packus_epi16, _mm_store_si128, _mm_load_si128, _mm_packus_epi16, _mm_unpacklo_epi8, _mm_unpackhi_epi8

See Also

Other Resources

MFC Samples

MODELESS Sample: Uses a CDialog Object as a Modeless Dialog Box

Download sample

The MODELESS sample demonstrates the use of an MFC **CDialog** object as a modeless dialog box. MODELESS is a simple dialog-based application that manages a list box in its main dialog box while providing a modeless dialog box that allows you to add strings to the list box in the main window.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the MODELESS sample

1. Open the solution modeless.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

When the sample starts, it presents an empty list box. You can open the modeless dialog box by clicking **Add**. Even while the **Modeless Adder** dialog box is open, you can bring focus back to the main dialog box. The **Add** button on the main dialog box is unavailable when the modeless dialog box is open so that the user cannot create more than one instance of the modeless box.

The main dialog box's **CMainDlg** class manages a pointer to the modeless dialog box. It does this just for convenience; once created, the modeless dialog box requires no further management. In your application, you might choose to offer the modeless box information — that pointer would provide access to the C++ object managing the dialog box and therefore would be a great place to start.

The code for the **Add** button in the main dialog box creates the modeless dialog box using the **Create** function, instead of calling **DoModal**. This is what makes the box modeless; Windows treats messages for the box differently. When the box is destroyed, **EndDialog** is not used; instead, **DestroyWindow** is called. Because the normal **OnOk** and **OnCancel** member functions of a **CDialog** object would call **EndDialog**, make sure your modeless dialog box does not call those functions and instead overrides them to call **DestroyWindow**.

Usually, when you create a modal dialog box, you destroy it manually after **DoModal** returns. Because you cannot wait for **Create** to return while displaying your modeless dialog box, you need to have some other mechanism for destroying the C++ object associated with the window. This sample uses a very simple mechanism: It performs `delete this` in **PostNcDestroy** — a function that is called after the nonclient area of the box has been destroyed.

Note that the modeless dialog box communicates with its parent dialog in two different ways. First, when the user presses **OK**, the string in the edit control in the modeless dialog box is added to the content of the list box in the modal dialog box. Second, when the user destroys the window by whatever means, the modeless box calls the **BoxDone** function in the modal window. This function simply resets the pointer to the modal dialog box and re-enables the Add button.

Keywords

This sample demonstrates the following keywords:

AfxGetApp; CDC::DrawIcon; CDC::GetSafeHdc; CDialog::Create; CDialog::DoModal; CDialog::OnCancel; CDialog::OnOK; CListBox::AddString; CMenu::AppendMenu; CMenu::ModifyMenu; CRect::Height; CRect::Width; CString::IsEmpty; CString::LoadString; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWnd::DestroyWindow; CWnd::DoDataExchange; CWnd::EnableWindow; CWnd::GetClientRect; CWnd::GetDlgItem; CWnd::GetWindowText; CWnd::IsIconic; CWnd::OnPaint; CWnd::OnQueryDragIcon; CWnd::OnSysCommand; CWnd::PostNcDestroy; CWnd::SendMessage; CWnd::SetActiveWindow; GetSystemMenu; GetSystemMetrics; LoadIcon

Note

Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

NPP Sample: Demonstrates the Windows Messaging API (MAPI)

Download sample

The NPP sample is a single-document interface (SDI) application similar to Notepad that allows you to edit text messages and send them to other users or other systems through the Windows messaging API, or MAPI.

If you are running Windows NT 4.0, make sure that MAPI is installed on your machine before running the sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the NPP sample

1. Open the solution NPP.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

The **File** menu has a **Send** command that can attach the content of the edit window to a MAPI message. Selecting **Send** will prompt you for a user name and password to start your MAPI session if you are not already logged on.

The two important lines of code that enable this feature (aside from adding the menu choice itself to the resource file) are entries in the message map for the **CNotepadDoc** instance, which the application maintains. The entries are:

```
ON_COMMAND(ID_FILE_SEND_MAIL, OnFileSendMail)
ON_UPDATE_COMMAND_UI(ID_FILE_SEND_MAIL, OnUpdateFileSendMail)
```

CDocument::OnFileSendMail and CDocument::OnUpdateFileSendMail are built into MFC; nothing extra is needed to gain this level of MAPI support in your applications.

Keywords

This sample demonstrates the following keywords:

AfxGetMainWnd; AfxMessageBox; CComboBox::Copy; CComboBox::Create; CComboBox::GetDroppedState; CComboBox::Paste; CComboBox::SetEditSel; CComboBox::ShowDropDown; CDC::GetTextExtent; CDC::GetTextMetrics; CDC::SelectObject; CDialog::DoModal; CDialog::OnCancel; CDialog::OnInitDialog; CDocument::OnNewDocument; CEdit::GetLineCount; CEdit::LineFromChar; CEdit::LineIndex; CEdit::LineScroll; CEdit::SetSel; CEditView::GetEditCtrl; CEditView::GetSelectedText; CEditView::OnFindNext; CFrameWnd::Create; CFrameWnd::DockControlBar; CFrameWnd::EnableDocking; CFrameWnd::GetActiveView; CFrameWnd::RecalcLayout; CObject::AssertValid; CObject::Dump; CObject::Serialize; CRect::Height; CStatusBar::CommandToIndex; CStatusBar::SetPaneText; CString::Format; CString::GetLength; CString::LoadString; CToolBar::GetItemRect; CToolBar::SetButtonInfo; CView::GetDocument; CView::OnBeginPrinting; CView::OnDraw; CView::OnEndPrinting; CView::OnPreparePrinting; CWinApp::AddDocTemplate; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWinApp::OnFileNew; CWinApp::OnIdle; CWinApp::OpenDocumentFile; CWnd::Create; CWnd::DoDataExchange; CWnd::FromHandle; CWnd::GetDescendantWindow; CWnd::GetDlgItem; CWnd::GetFocus; CWnd::GetStyle; CWnd::GetWindowRect; CWnd::GetWindowText; CWnd::GetWindowTextLength; CWnd::KillTimer; CWnd::OnClose; CWnd::OnCreate; CWnd::PostMessage; CWnd::PreTranslateMessage; CWnd::ScreenToClient; CWnd::SendMessage; CWnd::SetFocus; CWnd::SetFont; CWnd::SetTimer; CWnd::SetWindowPos; CWnd::SetWindowText; CWnd::ShowWindow; CreateFontIndirect; DragAcceptFiles; EndDialog; EnumFontFamilies; GetDlgItem; GetKeyState; GetLocalTime; GetStockObject; LoadBitmap; MessageBeep; SendMessage; SetWindowText; memcpy

Note

Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

PROPDLG Sample: Demonstrates Property Sheet Support

Download sample

The PROPDLG sample illustrates MFC support for property sheets or tabbed dialog boxes. PROPDLG also illustrates a modeless miniframe window.

PROPDLG is a simple object drawing program that uses property sheets for entering the shape and color attributes of a currently selected object. For an example of a more fully featured object drawing program, see the DRAWCLI sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the PROPDLG sample

1. Open the solution propdlg.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

When you run the sample, click anywhere in the view to add a new shape, which is initially a fixed-size rectangle. Click elsewhere to add more shapes. To select a shape, click that shape. The **Object** menu offers three types of property sheets for updating the attributes of the currently selected object. A real application would typically only offer one of these types of property sheets:

Simple Property Sheet

A pop-up dialog box with tabs for **Style** and **Shape**. Click **OK** to apply the properties to the currently selected object and exit the dialog box. Click **Cancel** to exit the dialog box without applying the properties. The **Apply Now** and **Help** buttons are always disabled in these illustrations.

Property Sheet with Preview

Also a pop-up dialog box with two tabs. This dialog box also has a preview child window. The preview window shows you what the object would look like if the current values in the property sheet were applied. This illustrates how you can customize the layout of a property sheet. This example also implements the **Apply Now** button, which is enabled whenever you change any property.

Miniframe Property Sheet

A modeless property sheet dialog box within a mini-frame window. The properties in this modeless dialog box always reflect the currently selected object. Changes to values in the property sheet are applied immediately to the currently selected object.

Keywords

This sample demonstrates the following keywords:

AfxGetMainWnd; CCmdUI::Enable; CDC::GetClipBox; CDialog::DoModal; CDocument::OnNewDocument; CDocument::SetModifiedFlag; CDocument::UpdateAllViews; CFrameWnd::Create; CFrameWnd::GetActiveFrame; CFrameWnd::GetActiveView; CFrameWnd::LoadFrame; CMDIFrameWnd::MDIGetActive; CObject::AssertValid; CObject::Dump; CObject::Serialize; CPen::CreatePen; CRect::Height; CRect::IntersectRect; CRect::IsRectNull; CRect::PtInRect; CRect::Width; CStatusBar::Create; CStatusBar::SetIndicators; CString::LoadString; CToolBar::Create; CToolBar::LoadBitmap; CToolBar::SetButtons; CView::DoPreparePrinting; CView::GetDocument; CView::OnBeginPrinting; CView::OnDraw; CView::OnEndPrinting; CView::OnPreparePrinting; CWinApp::AddDocTemplate; CWinApp::EnableShellOpen; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWinApp::RegisterShellFileTypes; CWnd::CenterWindow; CWnd::Create; CWnd::DoDataExchange; CWnd::FromHandle; CWnd::GetClientRect; CWnd::GetParent; CWnd::InvalidateRect; CWnd::IsWindowVisible; CWnd::OnCreate; CWnd::OnEraseBkgnd; CWnd::OnKeyDown; CWnd::OnLButtonDblClk; CWnd::OnLButtonDown; CWnd::OnPaint; CWnd::SendMessage; CWnd::ShowWindow; CWnd::UpdateWindow; DragAcceptFiles;

Ellipse; FillRect; GetParent; GetWindowRect; RGB; Rectangle; RoundRect; SelectObject; SendMessage; SetFocus; SetWindowPos; UpdateWindow

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

ROWLIST Sample: Demonstrates Selecting Full Rows in List Views

Download sample

The ROWLIST sample illustrates selecting full rows in a report mode of the **CListView** MFC common control class. The sample implements a reusable class, **CListViewEx**, that provides full row selection as an additional view mode of a list view control. **CListViewEx** uses the owner-draw mode to do all the painting for the control. You can use the class in your own projects, either directly or as a base class. ROWLIST also demonstrates how to use state and overlay images with a list view control.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the ROWLIST sample

1. Open the solution rowlist.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

When you run ROWLIST, it presents a single-document interface (SDI) application whose view is a **CListView**-derived class. The view has items representing different colors, with additional data about them. The list view control comes up initially in the report mode, with full row selection enabled. The **View** menu and the toolbar allow you to select other view modes.

State images are initially set to an empty square on the left side of each item. If you click a state image, the item's image becomes an icon displayed on the caption bar. If state images are not displayed, you can double-click an item to change its state.

CListViewEx Class

The class **CListViewEx**, derived from **CListView**, is a reusable class that implements the full row selection mode. The class provides complete support for the owner-draw mode of a list view control, and provides full row selection by implementing painting code for all images and labels. **CListViewEx** displays a control's images and labels exactly as in the report mode, with the addition of the full row selection. The class has only two additional member functions (as compared to **CListView**): **SetFullRowSel** and **GetFullRowSel**. These functions are used to set or query for the full row selection mode.

You can reuse the class in your own projects, either by deriving a class from it, or by using it directly. The simplest way to use the class is to use the application wizard to create an application that has a **CListView**-derived class, and then to change its base class to **CListViewEx**.

Using State and Overlay Images with a CListView

ROWLIST demonstrates use of state and overlay images. State images are initially set to an empty square on the left side of each item. If you click a state image, the item's image becomes an icon displayed on the caption bar. In response, the item's state icon changes to a square with a check mark. **CRowListView::OnLButtonDown** contains the code that inserts the check mark if a state icon is selected, and that changes a state icon for an item.

If state images are not displayed, then the image currently in use will be marked with an overlay image on top of it (a gray square with a check mark). You can use a double-click to change the state of an item, which also toggles an overlay image on or off. The implementation code for the double-click is found in **CRowListView::OnLButtonDblClk**.

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

SCRIBBLE Sample: MFC MDI Drawing Application

Download sample

SCRIBBLE is a small drawing application that lets you draw freehand drawings using the mouse and save the images in a file. The sample provides simple illustrations of a wide breadth of MFC features:

- Application objects
- Documents, views, and document templates
- Commands, message maps, and command user interface updating
- MDI (multiple document interface) frame and child windows
- Toolbars and status bars
- Update hints from documents to views
- Dialog boxes, data exchange, and validation
- Scroll view
- Splitter window
- Printing and print preview

Refer to other MFC samples for illustrations of advanced use of these features.

SCRIBBLE also includes Visual Editing server functionality.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the SCRIBBLE sample

1. Open the solution Scribble.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

Other Versions of Scribble

Scribble is also available in versions that target the CLR:

- Scribble Sample: MDI Drawing Application
- ScribbleDotNET Sample: Extending Scribble with CLR Features

Keywords

The SCRIBBLE sample demonstrates the following keywords:

AfxMessageBox; AfxOleInit; CArchive::IsStoring; CCmdUI::Enable; CCmdUI::SetCheck; CControlBar::EnableDocking; CControlBar::GetBarStyle; CControlBar::SetBarStyle; CDC::DPtoLP; CDC::GetDeviceCaps; CDC::GetTextMetrics; CDC::LPtoDP; CDC::LPtoHIMETRIC; CDC::LineTo; CDC::MoveTo; CDC::SelectObject; CDC::SetMapMode; CDC::SetTextAlign; CDC::SetViewportExt; CDC::SetWindowExt; CDC::SetWindowOrg; CDC::TextOut; CDialog::DoModal; CDocTemplate::SetServerInfo; CDocument::DeleteContents; CDocument::GetFirstViewPosition; CDocument::GetNextView; CDocument::OnNewDocument; CDocument::OnOpenDocument; CDocument::SetModifiedFlag; CDocument::UpdateAllViews; CFont::CreateFontIndirect; CFrameWnd::Create; CFrameWnd::DockControlBar; CFrameWnd::EnableDocking; CFrameWnd::LoadFrame; CFrameWnd::OnCreateClient; CObList::GetHeadPosition; CObList::GetNext; CObject::AssertValid; CObject::Dump;

CObject::IsKindOf; CObject::Serialize; COleDocument::EnableCompoundFile; COleIPFrameWnd::OnCreateControlBars; COleServerDoc::GetEmbeddedItem; COleServerDoc::GetZoomFactor; COleServerDoc::NotifyChanged; COleServerDoc::OnGetEmbeddedItem; COleServerDoc::OnSetItemRects; COleServerItem::CopyToClipboard; COleServerItem::GetDocument; COleServerItem::IsLinkedItem; COleServerItem::OnDraw; COleServerItem::OnGetExtent; COleTemplateServer::ConnectTemplate; COleTemplateServer::UpdateRegistry; CPen::CreatePen; CRect::InflateRect; CRect::IntersectRect; CRect::SetRectEmpty; CScrollView::SetScrollSizes; CSplitterWnd::Create; CStatusBar::Create; CStatusBar::SetIndicators; CToolBar::Create; CView::DoPreparePrinting; CView::GetDocument; CView::OnBeginPrinting; CView::OnDraw; CView::OnEndPrinting; CView::OnInitialUpdate; CView::OnPrepareDC; CView::OnPreparePrinting; CView::OnPrint; CView::OnUpdate; CWinApp::AddDocTemplate; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWnd::DoDataExchange; CWnd::GetCapture; CWnd::Invalidate; CWnd::InvalidateRect; CWnd::OnCreate; CWnd::OnLButtonDown; CWnd::OnLButtonUp; CWnd::OnMouseMove; CWnd::OnSize; CWnd::PreCreateWindow; CWnd::SetCapture; CWnd::SetOwner; CWnd::ShowWindow; CWnd::UpdateWindow; CreatePen; DeleteObject; DragAcceptFiles; GetClipBox; LPtoDP; RGB; ReleaseCapture; SetRectEmpty; max; memset; min

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

SimpleImage Sample: Loads, Resizes, Converts, and Saves Images

Download sample

The SimpleImage sample shows how to use the CImage class to load, display, and save a variety of different image formats, including .bmp, .gif, .jpg, and .png. It shows how **CImage** can be used to manipulate the image in various ways, and how to extract information about the image. SimpleImage uses the CFileDialog class to present the user with the standard Windows dialog boxes for loading and saving files of various formats. The sample creates a docking toolbar using the CToolBar class. The toolbar displays a set of buttons for resizing the image.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the SimpleImage sample

1. Open the solution SimpleImage.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

The sample is a simple application that allows you to do the following:

- Load an image from disk in a variety of formats.
- Resize the image and convert it from color to grayscale.
- Save the image in several image formats.

Classes and Keywords

- This sample demonstrates the following classes:

  CImage, CString, CPaintDC, CDialog, CFileDialog, CToolBar, CStatic

- This sample demonstrates the following keywords:

  CToolBar::SetButtonInfo, CToolBar::SetButtons, CToolBar::LoadToolBar, CToolBar::EnableDocking, CImage::GetWidth, CImage::GetHeight, CImage::StretchBlt, CImage::GetExporterFilterSting, CImage::Load, CImage::Save, CImage::IsIndexed, CImage::GetPixel, CImage::SetPixelRGB, CImage::GetMaxColorTableEntries, CImage::GetColorTable, CImage::IsNull, CImage::GetBPP, CImage::IsDIBSection, CImage::GetPitch, CCmdUI::Enable, CCmdUI::SetCheck, CStatic::SendMessage, CWnd::GetClientRect, CWnd::GetDlgItem, sprintf

See Also

Other Resources

MFC Samples

SNAPVW Sample: Uses Property Pages in a Form View Application

Download sample

The SNAPVW sample demonstrates how to use property pages in a form view. The frame of the multiple-document interface (MDI) child window is used in place of a property sheet that might otherwise contain the **OK**, **Cancel**, and **Apply** buttons.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the SNAPVW sample

1. Open the solution Snap.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

When you run SNAPVW, a set of property pages is displayed within the MDI child frame. **CSnapView::OnSize** overrides the CFormView base class implementation and calculates the size of the frame window so that it "snaps" to the size of the property pages. **CSnapView::Create** sets up the window, adds the property pages, and sets the appropriate styles. The **Minimize** and **Maximize** buttons have been removed from the child frame window so that the frame remains closely aligned with the edges of the property pages. You can navigate between the pages by clicking the individual pages and choosing the menu items, or by using the mnemonics and arrow keys.

Keywords

This sample demonstrates the following keywords:

CCmdUI::Enable; CDocument::OnNewDocument; CFormView::Create; CFormView::OnEraseBkgnd; CFormView::OnSize; CFrameWnd::DockControlBar; CFrameWnd::EnableDocking; CObject::AssertValid; CObject::Dump; CObject::GetRuntimeClass; CObject::IsKindOf; CPropertyPage::Create; CPropertySheet::AddPage; CScrollView::SetScrollSizes; ,CWinApp::AddDocTemplate; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWinApp::OnFileNew; CWnd::Create; CWnd::DoDataExchange; CWnd::GetDlgItem; CWnd::GetWindowText; CWnd::OnCreate; CWnd::PreCreateWindow; CWnd::SetDlgCtrlID; CWnd::SetWindowPos; CWnd::ShowWindow

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

Spiro Sample: Animated Drawing Game

Download sample

The Spiro sample is a drawing game that shows how to work with the following features:

- Memory DCs in applications requiring animation effects.
- **MM_LOENGLISH** mapping mode.
- Image lists (created using **CImageList**), including transparency mode and bitmap dragging and dropping.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the Spiro sample

1. Open the solution spiro.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

The toolbar contains four ring types and seven wheel types. The rings and wheels can be combined (one and one) to create different "spirograph" drawings.

Note
The wheels can also be used as rings. In this case, a wheel is set outside another wheel.

Once the ring and the wheel are set on the drawing area, you can click the wheel's "pen-spot" and drag it to another position on the wheel. By changing the pen-spot location, different drawings are generated. In addition, you can select which colors to use from the toolbar. Finally, you can select the width of the pen by clicking the "width setting" button on the toolbar.

Once these settings have been selected, start drawing by clicking the "Play" button. This triggers the drawing action, which can be paused with the "Pause" button. It can be "stepped" with the "step" button.

Other Features

- **Drawing Speed** The drawing speed is controlled using the "Fast Drawing" and "Slow Drawing" buttons on the toolbar.
- **Clearing or Repositioning the Rings** Once the drawing is complete, you can remove the rings or wheel from the drawing surface by clicking the "cancel-ring" or "cancel-wheel" button on the toolbar.

Note
Clicking the "cancel-ring" button removes the drawing wheel as well if it is present.

  To reposition the wheel or ring, remove the ring and place it again.
- **Copying and Dragging Spirographs** Completed spirographs can be dragged to different locations on the drawing area using the mouse. To copy a drawing, hold down the Ctrl key and drag it to a new location.
- **Removing Spirographs** To remove a drawing from the drawing area, drag and drop it off the drawing surface. To accomplish this, you may need to zoom out one or more times until the limits of the drawing area are displayed on the screen.
- **Zooming** There are two toolbar buttons to perform zoom-in and zoom-out. These buttons work even while a drawing

is being created. You can zoom out several times until the limits of the drawing area are visible.

- **Printing** You can see the drawings prior to printing using the Print Preview option on the File menu. Very nice patterns can be printed with a color printer.

Classes and Keywords

This sample demonstrates the following classes:

CImageList, CToolbar, CPoint, CPen, CBrush, CDC, CArchive, CPaintDC, CRgn, CBitmap, CCmdUI, CToolbarCtrl, CSize

This sample demonstrates the following keywords:

CBitmap::CreateCompatibleBitmap, CBitmap::DeleteObject, CImageList::DragEnter, CImageList::DragLeave, CImageList::DragMove, CImageList::EndDrag, CDC::DPtoLP, CDC::LPtoDP, CDC::IsPrinting, CDC::SetWindowOrg, CDC::SetViewportOrg, CDC::FillRect, CCmdUI::Enable, CView::OnPrepareDC, CRect::SetRect, CRect::InflateRect, CToolbarCtrl::IsButtonChecked

See Also

Other Resources

MFC Samples

TRACKER Sample: Illustrates Various CRectTracker Styles and Options

Download sample

The TRACKER sample is a test application that provides an exhaustive illustration of **CRectTracker** member functions, styles, and options. For a real-world example that uses **CRectTracker**, see DRAWCLI, the object-oriented drawing sample application.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the TRACKER sample

1. Open the solution tracker.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

TRACKER initially displays a square with four colored and numbered quadrants. The quadrants are displayed to help you see when the square is inverted horizontally and/or vertically. Initially, the square has no **CRectTracker** adornments. Try the various toolbar commands, or **Edit** menu commands, to turn on and off **CRectTracker** styles, including dotted or solid lines; hatched border, inside or outside the rectangle; and resize handles, inside or outside the rectangle.

Note how the cursor changes shape over parts of the rectangle to indicate what happens if you drag the mouse. Try moving and resizing the rectangle.

Keywords

This sample demonstrates the following keywords:

CBrush::CreateSolidBrush; CDialog::DoModal; CDocument::OnNewDocument; CDocument::SetModifiedFlag; CDocument::UpdateAllViews; CFrameWnd::Create; CFrameWnd::LoadFrame; CGdiObject::DeleteObject; CObject::AssertValid; CObject::Dump; CObject::Serialize; CRect::Height; CRect::IntersectRect; CRect::Width; CRectTracker::Draw; CRectTracker::GetTrueRect; CRectTracker::HitTest; CRectTracker::SetCursor; CRectTracker::Track; CRectTracker::TrackRubberBand; CView::DoPreparePrinting; CView::GetDocument; CView::OnBeginPrinting; CView::OnDraw; CView::OnEndPrinting; CView::OnPreparePrinting; CView::OnUpdate; CWinApp::AddDocTemplate; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWinApp::OnFileNew; CWnd::DoDataExchange; CWnd::InvalidateRect; CWnd::OnCreate; CWnd::OnLButtonDown; CWnd::OnSetCursor; CWnd::ShowWindow; CWnd::UpdateWindow; ExtTextOut; GetTextMetrics; LoadBitmap; MessageBeep; PatBlt; RGB; SelectObject; SetBkMode; SetTextAlign; abs

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

VariantUse Sample: Demonstrates the Use of Variants

Download sample

The VariantUse sample demonstrates changing existing data into a variant, and changing a variant into other types of data. Many COM objects accept variants as function parameters. The goal of this sample is to help you change standard C data types into variants.

This sample covers the use of currency, dates, SAFEARRAYs, multidimensional arrays, strings, chars, shorts, and longs.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the VariantUse sample

1. Open the solution VariantUse.sln.
2. On the **Build** menu, click **Build**.

Run VariantUse.exe in the debugger. Place breakpoints in the code dealing with the type of data you are trying to understand. For example, if you want to see the use of variants with strings, place breakpoints in the **OnString** function. Run the sample in the debugger and click the **Strings** button. Clicking the buttons on the main dialog box will have no apparent effect unless there are breakpoints in the associated code. Once a breakpoint is reached, use the debugger's single-step feature to step through the code that transforms data into and out of a variant.

Keywords

This sample uses the following keywords:

_bstr_t; CComBSTR; CComBSTR::Append; COleSafeArray; COeSafeArray::AccessData; COleSafeArray::CreateOneDim; COleSafeArray::UnaccessData; CString; CURRENCY; SAFEARRAY; SafeArrayAccessData; SAFEARRAYBOUND; SafeArrayCreate; SafeArrayCreateVector; SafeArrayDestroy; SafeArrayGetElement; SafeArrayUnaccessData; SysAllocString; SysFreeString; T2COLE; USES_CONVERSION; VARIANT; VariantChangeType; VariantCopy

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

VIEWEX Sample: Demonstrates Multiple Views, Scroll Views, and Splitter Windows

Download sample

The VIEWEX sample illustrates the following.

- A static splitter window in which the order and number of panes never change and the panes are usually of different view classes.
- A form view (**CFormView**) that stays synchronized with other views on the same document.
- Dialog data exchange (DDX), specifically of radio group button data.

The document in VIEWEX is simple. Its data consists of a string and its color.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the VIEWEX sample

1. Open the solution viewex.sln.
2. On the **Build** menu, click **Build**.
3. On the **Debug** menu, click **Start Without Debugging**.

When you first run VIEWEX, it presents a **New** dialog box in which you select the type of document window to be illustrated:

- **Simple Text** displays the string, using the current color, centered in the window.
- **Input Form View** provides a form (a **CFormView**) in which you can change the string in an edit control, and change the color by selecting from a radio button group.
- **Splitter Frame** with both Simple Text and Input Form displays in two side-by-side panes of a splitter window. When you change the string (character by character) or color in the Input Form, the Simple Text view automatically reflects the change.
- **Three-way Splitter Frame** displays the Input Form view in one splitter window pane. The other pane is divided into two subpanes by a splitter bar: The upper subpane displays a Simple Text view; the lower subpane displays a Color view. The Color view simply paints itself entirely in the current color.

You can change the string by using the edit control in the input form and by clicking **Change Data** on the **Edit** menu.

You can also view multiple documents at a time. For the sake of simplicity, however, you cannot save or reopen documents, nor can you view the same document in more than one of the above windows.

Static Splitter Window

In VIEWEX, the MDI child window creates a 1-by-2 splitter window within its client area by overriding **OnCreateClient**. In the case of the three-way splitter, VIEWEX embeds a second 2-by-1 splitter window within the right-side pane of the 1-by-2 splitter window.

VIEWEX calls **CSplitterWnd::CreateStatic** instead of **CSplitterWnd::Create** to create static splitter windows instead of dynamic splitter windows. Each pane of VIEWEX's splitter window has a different view class. VIEWEX calls **CSplitterWnd::CreateView** to specify the **CView** derived class for each pane. In the case of the first pane, VIEWEX uses the view class specific in the document template object, which is passed to **OnCreateClient** in the **m_pNewViewClass** member of the **CCreateContext**. In the case of other panes, VIEWEX specifies the **RUNTIME_CLASS** of the view.

Form View

CInputView illustrates synchronizing a form view with other views, so when the user enters data in one of the fields, the change is immediately reflected in other views. **CInputView** accomplishes this by mapping **ON_EN_CHANGE** and **ON_BN_CLICKED** for all of its controls to a central **OnDataChange** message handler. **OnDataChange** calls **CView::UpdateData** to transfer data from the screen to the member variables of **CInputView** and then calls the document's **UpdateAllViews** to inform the other views about the change.

Dialog Data Exchange (DDX)

VIEWEX's **CInputView** illustrates dialog data exchange (DDX) of radio group button data. In Iputvw.h, note that only the first button in the radio group is mapped to the member variable `m_iColor` in **CInputView::DoDataExchange**. `m_iColor` is a zero-based ordinal value (**int**) representing which radio button is selected.

Keywords

This sample demonstrates the following keywords:

CDC::FillRect; CDialog::DoModal; CDocument::OnNewDocument; CDocument::UpdateAllViews; CFrameWnd::LoadFrame; CFrameWnd::OnCreateClient; CFrameWnd::SetActiveView; CObject::Serialize; CRect::Height; CRect::Width; CSplitterWnd::CreateStatic; CSplitterWnd::CreateView; CSplitterWnd::GetPane; CSplitterWnd::IdFromRowCol; CString::GetLength; CView::GetDocument; CView::OnActivateView; CView::OnDraw; CView::OnUpdate; CWinApp::AddDocTemplate; CWinApp::InitInstance; CWinApp::OnFileNew; CWnd::DoDataExchange; CWnd::GetClientRect; CWnd::OnMouseActivate; CWnd::ShowWindow; CWnd::UpdateData; CWnd::UpdateWindow; RGB; SetBkMode; SetTextAlign; SetTextColor; TextOut; max

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

MFC Internet Samples

The following topics are the abstracts for the MFC Internet samples. For a list of all MFC samples, see MFC Samples.

See Also

Other Resources

MFC Samples

DHtmlExplore Sample: Demonstrates Using MFC DHtml Classes

Download sample

The DHtmlExplore sample shows how to use the **CDHtmlDialog** class with HTML resources. It also demonstrates how to handle dynamic HTML (DHTML) events and how to use DHTML dynamic data exchange (DDX).

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the DHtmlExplore sample

1. Open the solution DHtmlExplore.sln.
2. On the **Build** menu, click **Build**.
3. Run the application DHtmlExplore.exe.
4. Click the **Browse** button to select a directory. The dialog box is updated with the contents of the directory.
5. Click a folder or a file to open it. Clicking the folder named ".." moves up one directory. Clicking on the folder named "." refreshes the current directory.
6. Right-click an element to display the shortcut menu. Selecting **Browse** from the shortcut menu has the same effect as clicking the **Browse** button. Selecting **Properties** from the shortcut menu after right-clicking a folder or file displays a DHTML dialog box that presents the properties of the selected folder or file.

Classes and Keywords

- This sample demonstrates the following MFC class:

  CDHtmlDialog

- This sample demonstrates the following keywords:

  CDHtmlDialog::OnDocumentComplete; CDHtmlDialog::LoadFromResource; CDHtmlHostHandler::SetHostFlags; CDHtmlHostHandler::ShowContextMenu; CDialog::DoDataExchange; DECLARE_DHTML_EVENT_MAP; BEGIN_DHTML_EVENT_MAP; CDialog::DoModal; CDialog::OnInitDialog; CDHtmlDialog::OnInitDialog; DDX_DHtml_ElementInnerText; DDX_DHtml_CheckBox; DHTML_EVENT_ONCLICK; DHTML_EVENT_CLASS; CDHtmlEventHandler::GetElementInterface

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

HTMLEdit Sample: Wraps the Internet Explorer MSHTML Editing Control

Download sample

The HTMLEdit sample shows how to use the MFC classes that wrap the Internet Explorer WebBrowser ActiveX control. In editing mode, the WebBrowser ActiveX control exposes what you see is what you get (WYSIWYG) editing functionality. The MFC library provide classes (CHtmlEditView and CHtmlEditDoc) that host the MSHTML editing control in a window class and provide convenient wrapper classes for the control's properties, methods, and events.

For more information on the WebBrowser ActiveX control, see WebBrowser Control and Reusing MSHTML in MSDN Web Workshop.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

You must install Internet Explorer version 5 or later to get the proper editing functionality in the WebBrowser ActiveX control.

Building and Running the Sample

To build and run the HTMLEdit sample

1. Open the solution HTMLEdit.sln.
2. On the **Build** menu, click **Build**.
3. Run the HTMLEdit application.

HTMLEdit opens a simple SDI application similar in appearance to Notepad. The user can enter text in the view and then apply different HTML formats using either the toolbar or by hand.

Keywords

This sample demonstrates the following keywords:

CHtmlEditView, CHtmlEditDoc

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

MFCIE Sample: Demonstrates the MFC CHtmlView and CReBar Classes

Download sample

The MFCIE sample uses MFC's **CHtmlView** and **CReBar** classes to implement a subset of the functionality provided by Microsoft Internet Explorer.

The MFCIE sample provides a view very much like Microsoft Internet Explorer. You can browse to any location on the Web, enter data in forms, print Web pages, download files, or open files on your local hard drive. New addresses can be entered manually in the Address bar of the **CReBar** control, or hypertext links can be clicked on in the main view.

The sample demonstrates the ease in which a Web browsing capability can be inserted into any application. Most of the functionality of **CHtmlView** is trivial to implement (for example, **GoHome** or **GoBack**), although some of the less obvious features such as printing and font sizing have been implemented for your convenience. The MFCIE sample also demonstrates the proper method of retrieving and reading the Favorites selections and displays them in their own menu.

One thing to note is that the Web browser control used by **CHtmlView** does its own printing internally, so MFC's printing and Print Preview methods are bypassed entirely.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the MFCIE sample

1. Open the solution mfcie.sln.
2. On the **Build** menu, click **Build**.
3. Run the MFCIE application.

MFCIE assumes that you are already connected to the Internet. When it first starts, it will open your home page and you can browse from there just as you would with Microsoft Internet Explorer, either by using the Address bar, the Favorites menu, or by following links on the current page.

Keywords

This sample demonstrates the following keywords:

CToolBarCtrl::SetButtonWidth; CToolBarCtrl::SetHotImageList; CToolBarCtrl::SetImageList; CReToolBar::SetButtons; CReToolBar::SetButtonInfo; CReToolBar::SetButtonText; CReBar::AddBar; CHtmlView::Navigate2; CAnimateCtrl::Play; CAnimateCtrl::Stop; CAnimateCtrl::Seek; CComboBoxEx::InsertItem; CHtmlView::GoHome; CHtmlView::GoBack; CHtmlView::GoForward; CHtmlView::GoSearch; CHtmlView::Stop; CHtmlView::Refresh; CHtmlView::ExecWB

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

StockTicker Sample: Displays Web Information in an ATL Control

Download sample

The StockTicker sample downloads stock information from the Web and displays it in an ATL control that scrolls the data from right to left. The control updates the data at set intervals, which the user sets.

StockTicker contains four parts:

- Shared ATL component (`stockquotes`).
- ATL control (`stocktickeratl`).
- ISAPI Extension DLL (`stocksourcemfc`).
- MFC container app (`containermfc`). You can rename this .exe file.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the StockTicker sample

1. Open the solution StockTicker.sln.
2. On the **Build** menu, click **Build**.
3. Use `regsvr32` to register two DLLs: `stockquotes.dll` and `stocktickeratl.dll`. You may need to register `ATL.dll`, also.

   The ISAPI Extension DLL mimics a stock data source. It should be installed on a computer where Web server software (such as Microsoft Internet Information Services [IIS]) has been installed. You must tell StockTicker where this DLL is located. For instructions on how to use the ISAPI Extension DLL, see below.
4. Run `containermfc.exe`. Right-click in the client area of the application to display a shortcut menu. Use this menu to change the behavior and properties of StockTicker, as described in a following section.

How to Add Stock Source Web Sites

Follow these steps to add your own stock source Web site.

1. Edit `stockquotes\stockquotes_.h`. Examine the `CSQSISAPIDll` class. You can use the class as an example for creating new stock quote classes.
2. Create a new class, derived from `CStockQuoteSource`.
3. In the new class constructor, set `m_strURL` to the full URL of the Web site from where to retrieve data. Be sure to include the command to retrieve the stock data.
4. The format of the HTML returned from the Web site may require that you override `SetStockInfo` and/or `ParseStockInfo`.
5. Rebuild the component and register it.

If your selected site returns data resembling `CONAME: <NA>`, where `CONAME` is the stock symbol, either the site could not be accessed, or the source class you created could not find the stock information from the HTML.

Note

The format, either decimal or fractional, of the stock data may differ among the stocks you choose to monitor. StockTicker m akes no attempt to alter the data to make it consistent.

Behavior and Properties of StockTicker

Stocks

Displays a dialog box through which you can add or remove the stock symbols to monitor. The dialog box is initialized with the current list of stock symbols.

Update Now

Updates the stock information from the Web.

Properties

The Properties dialog box has three property pages: color, font, and custom. If the stock color and font property pages do not display in the Properties dialog box, make sure you have registered `msstkprp.dll`.

You can change the following properties.

- **Ticker Speed** Move the slider to increase or decrease the rate at which the stock information scrolls from right to left.
- **Update Interval** Time in minutes between updates.
- **ISAPI Extension DLL URL** Enter the full URL to the ISAPI Extension DLL. Append the following text to the end of the URL: `?QueryForStock?Symbol=`.

  When specifying the URL for the ISAPI Extension DLL, you need to include the component that is responsible for returning the HTML that contains the stock information.

About Stock Ticker

Opens the About dialog box.

Stay On Top

StockTicker will stay on top of all other applications; otherwise, it can be hidden by other applications.

Features Used by StockTicker

- ATL COM object
- ATL Dialog object
- ATL control
- MFC Internet (WinInet) classes
- MFC ISAPI Extension DLL
- Connection points
- Persistence
- Custom enumerators
- Standard C++ Library

StockTicker Components

This section provides more detail on the StockTicker components.

Stock Quotes Component

The shared component, the ATL object, is created with the ATL Project Application Wizard. It uses MFC WinInet support and other utility classes. Stock Quotes keeps track of the stocks to be monitored and updates the stock information from the Internet. An application can use this component for either synchronous or asynchronous updates. If the `StartUpdating` method is called, a child thread is created and the function returns immediately. The child thread creates a timer and waits until the update interval has elapsed. Then it requests an update from the Stock Quotes component. Subsequent calls to `Update` post a thread message to the child thread indicating that an update should occur, and `Update` returns immediately. In

synchronous mode, `Update` blocks until the download is complete.

The Stock Quotes component keeps track of stock data source objects that represent the Web sites from which the stock information is downloaded. When StockTicker updates the stock information, the Stock Quotes component goes through the list of stock data sources and attempts to download the information. If the attempt fails, the component tries to download stock data from the next source. If all of the stock data sources fail to download the information, `<NA>` will appear in the ticker window after the stock symbol.

Stock data comes from the Web site in HTML form. The data must be parsed to find the current price and the price change since the last update. The Stock Quotes component handles this parsing by searching for two keywords: for example, `Current:` or `Change:`. The component reads the data following the keywords if that data is either a fractional or decimal number (fractional or decimal).

Because stock Web sites may change the way they deliver data, you may have to change the algorithm that Stock Quotes uses to get the stock information.

See below for instructions on how to add your own stock data sources.

ISAPI extension DLL

The default stock data source is the local ISAPI extension DLL. The ISAPI extension DLL does not return real stock data. Instead, when you request data for a stock, the DLL checks whether the stock symbol is in its list of known stocks. If the stock symbol is on the list, the DLL retrieves the data and randomly adjusts the current stock price up or down. If the stock symbol is new, it is added to the list and an initial stock price is set.

Stock Ticker Control

This control was created with the ATL Object Wizard. It uses MFC to make some things easier.

You can use this control outside of StockTicker.

The Stock Ticker control uses the Stock Quotes component to handle the stock information and updates. When it comes time to save itself to a stream, the control will also ask the Stock Quotes component to save itself.

On startup, the Stock Ticker control creates the Stock Quotes component and asks it to start updating stock information. The control then starts a timer that it uses for the draw/render loop. When stock data has been updated, the Stock Quotes component notifies the control (through a connection point). The control then gets all the stock information from the Stock Quotes component and creates a string out of it. This string is drawn in the control and scrolls from right to left.

Stock Ticker Container

The Stock Ticker container is an MFC Active Control container, created with the MFC Application Wizard. It saves its size and position upon quitting, and it restores its previous state when it is restarted.

The container also handles the StockTicker menus.

Keywords

This sample demonstrates the following keywords:

IEnumXXX, CWinThread, std::vector, IConnectionPointContainer, CInternetSession, CInternetException, AfxParseURL, CHttpConnection, CHttpFile, COleClientItem, COleDocument, COleDispatchDriver, CreateILockBytesOnHGlobal, StgCreateDocfileOnILockBytes, OleSave, StgIsStorageILockBytes, GetHGlobalFromILockBytes, COleFont, std::basic_istream, std::basic_ostream, CHttpServer, CComPtr, CDialogImpl, std::list, IPersistStreamInit, IFontDisp, IViewObjectEx, CComControl, IOleControl, IPropertyPageImpl, CBitmap, CDC, IPersistStreamInit_Load, IPersistStreamInit_Save, IConnectionPointImpl

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

MFC WinForms Samples

The following samples demonstrate MFC WinForms Integration. For a list of all MFC samples, see MFC Samples.

In This Section

WinFormIntegration Sample: Demonstrates Hosting a WinForms User Control in an MFC Application

Demonstrates how to host a WinForm user control as a dialog or a view in an MFC application.

See Also

Other Resources

MFC Samples

Visual C++ Samples

WinFormIntegration Sample: Demonstrates Hosting a WinForms User Control in an MFC Application

Download sample

This sample demonstrates how to host a WinForms user control as a dialog or a view in an MFC application.

The sample is comprised of three MFC extension DLLs, two WinForms user controls, and one main MFC MDI app:

EXTDLL1 – Uses CWinFormsView to host the WinFormUserControlView1 user control as a CView window inside the MFC MDI app. Also uses command routing to send messages to the WinForms user control.

EXTDLL2 – Shows how to host a WinForms user control as a static control inside an existing **CView** window.

EXTDLL3 – Encapsulates the different ways a WinForms user control can be hosted in an MFC dialog, including using CDialog and CWinFormsDialog.

WinFormUserControl1 – A simple WinForms user control with some common controls added.

WinFormUserControlView1 – A WinForms user control that implements the **Microsoft::VisualC::MFC::IView** and **Microsoft::VisualC::MFC::ICommandTarget** interfaces to participate in MFC command routing and View notifications. This is a basic text editor (using a multiline .NET TextBox control) that uses MFC to save and open files.

MFC04 – This is the main MFC MDI app that loads the MFC extension DLLs, adding them as different MFC doc templates.

To build and run the sample:

1. To build the projects, simply open MFC04.sln and select **Build Solution** from the **Build** menu.
2. To run the different scenarios, launch the app and from the **File** menu, select **New**. Then select the CView, CFormView, or Dialogs template:
    a. The CView template opens a **CWinFormsView** with a .NET multiline editbox that you can edit. You also have the ability to save this text into a file and load it again via the MFC **Save** and **Open** menu items. You can invoke the **Save** functionality from within the WinForms by selecting the **Save Changes** drop-down button in the WinForms StatusStrip. Selecting the MFC **CViewCmds**->**Command1** menu item will update the text in the **StatusStrip** demonstrating MFC command routing to the hosted WinForms UserControl.
    b. The CFormView template is the same WinForms UserControl as above except it is hosted inside of a CFormView resource. This template only demonstrates the hosting code, the **Save** and **Open** functionality is not implemented.
    c. The Dialogs template is a View demonstrating a WinForms UserControl hosted in the following scenarios: inside of a **CDialog** invoked modal, inside of a **CDialog** invoked modeless, as a **CWinFormsDialog** invoked modal, and as a **CWinFormsDialog** invoked modeless. The **CWinFormsDialog** examples demonstrate hooking MFC functions up to WinForms control events. These examples also show how to pass strings from managed back to native via the CString to **System::String^** conversion operators.

Requirements

Supported Operating Systems: Windows Server 2003, Windows XP

See Also

Reference

MFC WinForms Samples

MFC OLE Samples

The following topics are the abstracts for the MFC OLE samples. For a list of all MFC samples, see MFC Samples.

See Also

Other Resources

MFC Samples

ACDUAL Sample: Adds Dual Interfaces to an Automation Application

Download sample

ACDual demonstrates how to add dual-interface support to an MFC-based Automation (formerly OLE Automation) application. The solution consists of the following projects:

- ACDualDriv, containing a version of the automation client AUTODRIV project that lets you select whether to control the server application using the dispatch interface or using vtable binding.
- ACDual, containing an actual version of the automation server AUTOCLIK project with dual-interface support.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building the Sample

To build the ACDual sample

1. Open the solution acdual.sln, located in the acdual directory.
2. On the **Build** menu, click **Build**.
3. Run ACDual once as a stand-alone application so it can register itself with the system.

Running the Sample

After registering the ACDual application, you are ready to run the ACDualDriv application. Building the solution from the IDE will perform the ACDualDriv registration automatically before ACDualDriv is run. ACDualDriv launches the ACDual application and creates a Document object, which you can then manipulate using Automation via the ACDualDriv user interface. It also contains an additional check box that lets you select whether to use VTBL binding to communicate with the ACDual Document object.

The ACDual server uses AUTOCLIK as a starting point. New globally unique identifiers (GUID) were generated to prevent confusion with the original AUTOCLIK sample, and some resources strings were changed to clarify whether the ACDual server is running. All other changes to the sources are marked with comment blocks like this:

```
// DUAL_SUPPORT_START
... modified code goes here
// DUAL_SUPPORT_END
```

For more information about dual interfaces, object description language (ODL) scripts, and Automation error interfaces, see Technical Note 65.

Dual Interfaces

A dual interface allows you to implement an **IDispatch** interface or a VTBL interface. A dual interface is strongly recommended for all exposed Automation objects. Issues to consider when implementing a dual interface are discussed in Technical Note 65 and include:

- Implementing dual-interface support for **CCmdTarget**-based classes
- Passing dual-interface pointers
- Enabling typesafe binding, including:
    - Registering the application's library type
    - Modifying project build settings to accommodate type library changes

- Specifying the correct object class name in a type library
- Handling exceptions and the automation error interface

For more information, see the ActiveX topics Overview of Automation, Dual Interfaces, Type Description Interfaces, and the ODL reference entry on the dual attribute.

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

Keywords

The ACDual sample demonstrates the following keywords:

AfxMessageBox; AfxOleInit; AfxOleLockApp; AfxOleUnlockApp; CCmdTarget::EnableAutomation; CCmdTarget::FromIDispatch; CCmdTarget::GetIDispatch; CCmdTarget::OnFinalRelease; CControlBar::EnableDocking; CControlBar::GetBarStyle; CControlBar::SetBarStyle; CDialog::DoModal; CDocument::GetFirstViewPosition; CDocument::GetNextView; CDocument::OnNewDocument; CDocument::SetModifiedFlag; CDocument::UpdateAllViews; CFrameWnd::ActivateFrame; CFrameWnd::DockControlBar; CFrameWnd::EnableDocking; CFrameWnd::LoadFrame; CObject::AssertValid; CObject::Dump; CObject::IsKindOf; CObject::Serialize; COleTemplateServer::ConnectTemplate; COleTemplateServer::UpdateRegistry; CStatusBar::Create; CStatusBar::SetIndicators; CToolBar::Create; CView::DoPreparePrinting; CView::GetDocument; CView::OnBeginPrinting; CView::OnDraw; CView::OnEndPrinting; CView::OnPreparePrinting; CWinApp::AddDocTemplate; CWinApp::EnableShellOpen; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWinApp::RegisterShellFileTypes; CWnd::DoDataExchange; CWnd::GetParentFrame; CWnd::OnCreate; CWnd::OnLButtonDown; CWnd::PreCreateWindow; CWnd::ShowWindow; CWnd::UpdateWindow; DragAcceptFiles; ShowWindow; TextOut

The ACDualDriv demonstrates the following keywords:

AfxGetApp; AfxMessageBox; AfxOleInit; CDC::DrawIcon; CDC::GetSafeHdc; CDialog::DoModal; CDialog::EndDialog; COleDispatchDriver::AttachDispatch; COleDispatchDriver::GetProperty; COleDispatchDriver::InvokeHelper; COleDispatchDriver::SetProperty; CRect::Height; CRect::Width; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWnd::DoDataExchange; CWnd::GetClientRect; CWnd::IsIconic; CWnd::OnClose; CWnd::OnCreate; CWnd::OnPaint; CWnd::OnQueryDragIcon; CWnd::SendMessage; CWnd::ShowWindow; CWnd::UpdateData; GetSystemMetrics; LoadIcon; ShowWindow

See Also

Other Resources

MFC Samples

AUTOCLIK Sample: Automation Server Application

Download sample

The AUTOCLIK sample is a simple Automation (formerly OLE Automation) server application. The autoclik solution consists of two distinct projects: the autoclik project, representing the implementation of the automation server, and the autodriv project, representing the client side that drives the sample.

AUTOCLIK illustrates the following:

- Automation starter code created by the application wizard.
- Option for creating automation-enabled **CCmdTarget**-derived classes and for adding automation properties and methods.
- Exposing existing class members and functions to other applications using Automation.
- Multiple dispatch interfaces in a single Microsoft Foundation Class Library (MFC) application.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building the Sample

To build the AUTOCLIK sample

1. Open the solution autoclik.sln.
2. On the **Build** menu, click **Build**.

Running the Sample

After registering the AUTOCLIK application, you are ready to run the AUTODRIV application. Building the solution from the IDE will perform the AUTODRIV registration automatically before AUTODRIV is run. Run the AUTODRIV application. It automatically launches the AUTOCLIK application and creates a document. To test the capabilities of this sample, manipulate the document using the interface provided by the AUTODRIV application.

The AUTODRIV sample is a simple Automation client application that drives the AUTOCLIK sample. The AUTODRIV sample is a good demonstration of writing an automation driver application. It uses the COleDispatchDriver::SetProperty and COleDispatchDriver::GetProperty to manipulate the controls through automation.

For details on how to use AUTOCLIK and AUTODRIV together using Remote Automation, see Running Remote Automation Using AUTOCLIK and AUTODRIV.

Keywords

The AUTOCLIK automation server sample demonstrates the following keywords:

AfxMessageBox; AfxOleInit; AfxOleLockApp; AfxOleUnlockApp; CCmdTarget::EnableAutomation; CCmdTarget::FromIDispatch; CCmdTarget::GetIDispatch; CCmdTarget::OnFinalRelease; CControlBar::EnableDocking; CControlBar::GetBarStyle; CControlBar::SetBarStyle; CDialog::DoModal; CDocument::GetFirstViewPosition; CDocument::GetNextView; CDocument::OnNewDocument; CDocument::SetModifiedFlag; CDocument::UpdateAllViews; CFrameWnd::ActivateFrame; CFrameWnd::DockControlBar; CFrameWnd::EnableDocking; CFrameWnd::LoadFrame; CObject::AssertValid; CObject::Dump; CObject::IsKindOf; CObject::Serialize; COleTemplateServer::ConnectTemplate; COleTemplateServer::UpdateRegistry; CStatusBar::Create; CStatusBar::SetIndicators; CToolBar::Create; CView::DoPreparePrinting; CView::GetDocument; CView::OnBeginPrinting; CView::OnDraw; CView::OnEndPrinting; CView::OnPreparePrinting; CWinApp::AddDocTemplate; CWinApp::EnableShellOpen; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWinApp::RegisterShellFileTypes; CWnd::DoDataExchange; CWnd::GetParentFrame; CWnd::OnCreate; CWnd::OnLButtonDown; CWnd::PreCreateWindow; CWnd::ShowWindow; CWnd::UpdateWindow; DragAcceptFiles; ShowWindow; TextOut

The AUTODRIV Automation Client sample demonstrates the following keywords:

AfxGetApp; AfxMessageBox; AfxOleInit; CDC::DrawIcon; CDC::GetSafeHdc; CDialog::DoModal; CDialog::EndDialog; COleDispatchDriver::AttachDispatch; COleDispatchDriver::GetProperty; COleDispatchDriver::InvokeHelper; COleDispatchDriver::SetProperty; CRect::Height; CRect::Width; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWnd::DoDataExchange; CWnd::GetClientRect; CWnd::IsIconic; CWnd::OnClose; CWnd::OnCreate; CWnd::OnPaint; CWnd::OnQueryDragIcon; CWnd::SendMessage; CWnd::ShowWindow; CWnd::UpdateData; GetSystemMetrics; LoadIcon; ShowWindow

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

CALCDRIV Sample: Demonstrates an Automation Client Application

Download sample

The CALCDRIV sample is a simple Automation (formerly OLE Automation) client application. CALCDRIV drives the MFCCALC sample application, which is an Automation server that provides basic calculator functions. MFCCALC has a simple calculator interface that looks like the Calculator application that comes with Microsoft Windows.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the CALCDRIV sample

1. Follow the instructions in MFCCALC to build and run that project.
2. Open the solution calcdriv.sln
3. On the **Build** menu, click **Build**.

Note
If you do not build and register MFCCALC before building CALCDRIV, you get an "Unable to create 'MFCCALC.Application' object" message.

When you run CALCDRIV, the Microsoft Foundation Class Library (MFC) **Calc Driver** dialog box appears and the MFCCALC application is launched. You can use MFCCALC's calculator user interface directly at this point, or you can drive MFCCALC from CALCDRIV as follows:

1. In MFC Calc Driver's **Expression** box, provide two or more numbers separated by the operator +, -, *, or /.

Note
MFCCALC does not interpret parentheses and does not implement operator precedence.

2. Click **Go** to have MFCCALC evaluate the expression in one step.
3. Click **Single Step** to step through the expression one number or operator at a time.
4. Click **Refresh** to request state information for MFCCALC (**Last Accum** and **Last Operand**).

Using a Dispatch Class

CALCDRIV uses the **CRemoteCalcDlg** class as follows:

- The **CRemoteCalcDlg** class of CALCDRIV represents the dispatch interface of MFCCALC. Class **CRemoteCalcDlg** is derived from **CCmdTarget**, which has a few automation-specific member functions, such as **CreateDispatch**.
- **CDriverDlg** embeds a **CRemoteCalcDlg** object, `m_calc`. The **CRemoteCalcDlg** object, like the `CDriverDlg` object in which it is embedded, is alive for most of the duration of CALCDRIV. The **CRemoteCalcDlg** is constructed when the dialog object is constructed.
- **CDriverDlg::OnInitDialog** calls **CCmdTarget::CreateDispatch** for the **CRemoteCalcDlg** object. **CreateDispatch** requires the dispatch name as the first parameter. Typically, the developer of an automation server application provides documentation describing the names of the dispatch interfaces and the properties and methods of the interfaces. Another way to find the names of the dispatch interfaces of an automation server application is to look at the server's

Windows registration, using REGEDIT /v (the verbose option).

- **CDriverDlg** implements CALCDRIV's expression evaluator by calling the `Button` method of MFCCALC, which is an emulator for the various buttons in the calculator's dialog box.
- **CDriverDlg** implements CALCDRIV's `Refresh` function by calling the `GetOpnd` and `GetAccum` methods of MFCCALC.
- The **CDriverDlg** destructor calls the `Quit` method exposed by MFCCALC to shut down MFCCALC when CALCDRIV closes.

Keywords

This sample demonstrates the following keywords:

AfxMessageBox; AfxOleInit; CDialog::DoModal; CDialog::EndDialog; CDialog::OnInitDialog; CEdit::GetSel; CEdit::SetSel; COleDispatchDriver::AttachDispatch; COleDispatchDriver::CreateDispatch; COleDispatchDriver::GetProperty; COleDispatchDriver::InvokeHelper; COleDispatchDriver::SetProperty; CString::GetLength; CWinApp::InitInstance; CWnd::DoDataExchange; CWnd::GetWindowText; CWnd::SetWindowText; GetWindowText; afxMemDF; min; wsprintf

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

DRAWCLI Sample: Illustrates Integrating Active Container Support with Application-Specific Features

Download sample

The DRAWCLI sample is an object-oriented drawing application with Visual Editing container support. Among the MFC Active container samples — CONTAINER, OCLIENT, and DRAWCLI — this sample provides the best illustration of integrating Active container support with application-specific features (in this case, drawing features). In addition, DRAWCLI demonstrates effective use of C++ polymorphism in the design of its "shape" and "drawing tool" classes (**CDrawObj** and **CDrawTool**).

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the DRAWCLI sample

1. Open the solution drawcli.sln.
2. On the **Build** menu, click **Build**.
3. From the project's **Debug** directory, open and run the DRAWCLI application.

DRAWCLI's Windows Logo Features

DRAWCLI also illustrates Windows logo compliance. All MFC applications meet some of the requirements for the Windows logo: a Win32 executable, support for long file names, support for UNC path names, and use of system colors and metrics. DRAWCLI meets the remaining requirements for the Windows logo by including the following features.

- ActiveX support. DRAWCLI is an Active container that stores its files in the compound file format, supports in-place activation, and acts as a drop target for drag-and-drop operations.
- MAPI support. DRAWCLI provides a "Send as Mail" message on its File menu, allowing the user to send a document as a mail attachment.
- Compliance with shell guidelines, including registration of large and small icons, use of the system registry instead of an .ini file, and having a setup and an uninstall program. For the latter, DRAWCLI includes a script compatible with InstallSHIELD, Stirling Software's toolkit for creating setup and uninstall programs.

DRAWCLI also meets the following recommendations for Windows applications.

- Uses tabbed property pages.
- Uses Windows common controls.
- Displays a shortcut menu in response to a right-button mouse click.
- Stores Summary Information with its documents.

DRAWCLI's user interface is similar to those of other object-oriented drawing programs.

Integrating Active Container Support with Application-Specific Features

The DRAWCLI sample was originally a stand-alone drawing application developed using the MFC classes. The stand-alone version of DRAWCLI was then integrated with a second skeleton version of DRAWCLI created using the application wizard's ActiveX Container feature. This process is similar to how the ActiveX Visual Editing server adds server support to SCRIBBLE.

The design of an MFC ActiveX container application should look essentially the same, regardless of whether you are adding ActiveX functionality to an existing stand-alone MFC (doc/view) application, or whether you're starting with an application wizard–generated ActiveX container application. The following is a brief description of how DRAWCLI is separated into application-specific code and ActiveX container-specific code.

- Class **CDrawObj**, implemented in Drawobj.cpp, is a base class for derived "shape" classes. This base class handles hit testing of shapes, moving of shapes, and resizing of shapes. Using polymorphism, DRAWCLI can interact with objects of different classes through **CDrawObj's** interface.
- Classes **CDrawRect** and **CDrawPoly** are derived from **CDrawObj**. **CDrawRect** is used to draw rectangles, rounded rectangles, ellipses, and lines. **CDrawPoly** is used to draw polygons. These two classes are independent of DRAWCLI's ActiveX container functionality.
- Class **CDrawOleObj** is also derived from **CDrawObj**, and is used to represent embedded objects. **CDrawOleObj** delegates any ActiveX-specific operation to a contained **CDrawItem** object (described below). For generic shape operations, embedded objects are treated like other shape objects in DRAWCLI because **CDrawOleObj** is derived from **CDrawObj**.
- Class **CDrawItem**, derived from **COleClientItem**, handles all the ActiveX-specific behavior for the embedded object. The implementation of **CDrawItem** is similar to the implementation of the **COleClientItem**-derived classes in the CONTAINER and OCLIENT samples.
- Class **CDrawDoc** is derived from **COleDocument**. The **COleDocument** object maintains a **CObList** of **CDrawObj** objects. **CDrawDoc** delegates several ActiveX container-specific menu commands, such as **Edit Paste**, **Paste Link**, and **Links**, to the base class **COleDocument**.
- Class **CDrawView** is derived from **CScrollView**. The ActiveX-specific implementation of **CDrawView** is similar to the implementation of the view classes in the CONTAIN and OCLIENT samples. The bulk of DRAWCLI's drawing-specific user interface is also implemented in **CDrawView**.

Keywords

This sample demonstrates the following keywords:

AfxGetApp; AfxGetMainWnd; AfxMessageBox; AfxOleInit; AfxRegisterWndClass; AfxThrowMemoryException; CArchive::Close; CArchive::IsStoring; CBitmap::CreateCompatibleBitmap; CBrush::CreateBrushIndirect; CBrush::CreateSolidBrush; CCmdTarget::BeginWaitCursor; CCmdTarget::EndWaitCursor; CCmdUI::Enable; CCmdUI::SetCheck; CCmdUI::SetRadio; CColorDialog::DoModal; CColorDialog::GetColor; CControlBar::EnableDocking; CControlBar::GetBarStyle; CControlBar::SetBarStyle; CDC::Attach; CDC::BitBlt; CDC::CreateCompatibleDC; CDC::DPtoLP; CDC::DrawFocusRect; CDC::FillRect; CDC::GetClipBox; CDC::GetDeviceCaps; CDC::HIMETRICtoDP; CDC::IntersectClipRect; CDC::IsPrinting; CDC::LPtoDP; CDC::LineTo; CDC::MoveTo; CDC::OffsetViewportOrg; CDC::OffsetWindowOrg; CDC::PatBlt; CDC::SelectObject; CDC::SetBkColor; CDC::SetBrushOrg; CDC::SetMapMode; CDC::SetViewportExt; CDC::SetViewportOrg; CDC::SetWindowExt; CDC::SetWindowOrg; CDialog::DoModal; CDocTemplate::SetContainerInfo; CDocument::GetFirstViewPosition; CDocument::GetNextView; CDocument::GetTitle; CDocument::OnNewDocument; CDocument::OnOpenDocument; CDocument::OnSaveDocument; CDocument::SetModifiedFlag; CDocument::SetTitle; CDocument::UpdateAllViews; CFrameWnd::DockControlBar; CFrameWnd::EnableDocking; CFrameWnd::LoadFrame; CFrameWnd::OnCreateClient; CGdiObject::UnrealizeObject; CMDIChildWnd::Create; CMenu::GetSubMenu; CMenu::LoadMenu; CMenu::TrackPopupMenu; CObList::AddTail; CObList::GetCount; CObList::GetHeadPosition; CObList::GetNext; CObList::IsEmpty; CObList::RemoveAll; CObList::RemoveAt; CObject::AssertValid; CObject::Dump; CObject::IsKindOf; CObject::Serialize; COleClientItem::Close; COleClientItem::CreateCloneFrom; COleClientItem::CreateFromData; COleClientItem::CreateStaticFromData; COleClientItem::Deactivate; COleClientItem::Delete; COleClientItem::DoVerb; COleClientItem::Draw; COleClientItem::GetActiveView; COleClientItem::GetClipboardData; COleClientItem::GetDocument; COleClientItem::GetExtent; COleClientItem::GetInPlaceWindow; COleClientItem::GetItemState; COleClientItem::GetType; COleClientItem::IsInPlaceActive; COleClientItem::OnChange; COleClientItem::OnChangeItemPosition; COleClientItem::OnGetItemPosition; COleClientItem::Release; COleClientItem::SetItemRects; COleClientItem::UpdateLink; COleDataObject::AttachClipboard; COleDataObject::GetFileData; COleDataObject::IsDataAvailable; COleDataSource::CacheGlobalData; COleDataSource::SetClipboard; COleInsertDialog::CreateItem; COleInsertDialog::DoModal; COleInsertDialog::GetSelectionType; CPen::CreatePen; CPen::CreatePenIndirect; CPrintDialog::CreatePrinterDC; CRect::BottomRight; CRect::Height; CRect::InflateRect; CRect::IntersectRect; CRect::IsRectEmpty; CRect::NormalizeRect; CRect::OffsetRect; CRect::SetRect; CRect::TopLeft; CRect::Width; CRectTracker::Draw; CRgn::CreateEllipticRgnIndirect; CRgn::CreatePolygonRgn; CRgn::CreateRoundRectRgn; CRgn::RectInRegion; CScrollView::GetDeviceScrollPosition; CScrollView::SetScrollSizes; CStatusBar::Create; CStatusBar::SetIndicators; CString::MakeLower; CToolBar::Create; CView::DoPreparePrinting; CView::GetDocument; CView::IsSelected; CView::OnActivateView; CView::OnBeginPrinting; CView::OnDragEnter; CView::OnDragLeave; CView::OnDragOver; CView::OnDraw; CView::OnDrop; CView::OnEndPrinting; CView::OnInitialUpdate; CView::OnPrepareDC; CView::OnPreparePrinting; CView::OnPrint; CView::OnScrollBy; CView::OnUpdate; CWinApp::AddDocTemplate; CWinApp::EnableShellOpen; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWinApp::RegisterShellFileTypes; CWinApp::SetRegistryKey; CWnd::DoDataExchange; CWnd::GetCapture; CWnd::GetParentFrame; CWnd::Invalidate; CWnd::InvalidateRect; CWnd::OnCreate; CWnd::OnDestroy; CWnd::OnEraseBkgnd; CWnd::OnLButtonDblClk; CWnd::OnLButtonDown; CWnd::OnLButtonUp; CWnd::OnMouseMove; CWnd::OnSetFocus; CWnd::OnSize;

CWnd::PreCreateWindow; CWnd::ScreenToClient; CWnd::SetCapture; CWnd::SetFocus; CWnd::ShowWindow; CWnd::UpdateWindow; DragAcceptFiles; Ellipse; GetACP; GetKeyState; GetMapMode; GetVersion; GlobalFree; GlobalLock; GlobalUnlock; LOWORD; LineTo; LoadCursor; MAKELONG; MoveTo; MulDiv; Polygon; RGB; Rectangle; RegisterClipboardFormat; ReleaseCapture; RoundRect; SelectObject; SetCursor; free; malloc; memcpy; min; realloc; wcstombs

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

HIERSVR Sample: Demonstrates a Server Application with OLE Drag and Drop

Download sample

The HIERSVR sample is an example of a Visual Editing server application that supports linked objects as well as in-place edited objects. In addition, as a "full" server, this application also can directly open and save files. The HIERSVR object is a multiple-level hierarchy diagram, where each node in the hierarchy consists of text surrounded by a rectangle or other shapes. In addition to illustrating server functionality, HIERSVR also illustrates OLE drag and drop, and copying to the Clipboard.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the HIERSVR sample

1. Open the solution hiersvr.sln.
2. On the **Build** menu, click **Build**.
3. Run HIERSRV as a stand-alone application so it can register itself with the system.

HIERSRV Diagrams

The type of document or object HIERSVR edits and displays is a single inheritance hierarchy diagram. Each node in the diagram, except the root node, has only one parent node. Each parent can have multiple children. A company's organization chart is an example of a hierarchy diagram. So is the list of the MFC classes, because they do not use C++ multiple inheritance. Mfcclass.hie is included as an example HIERSVR document.

Every node in the hierarchy has a minimal set of attributes.

- Description — Text consisting of up to 30 characters.
- Shape — Style of rectangle that surrounds the text rectangle, rounded rectangle, or oval rectangle.
- Item Key Link (optional) — Name used to refer to a linked item. If not specified, the item key link defaults to the description string. This name is visible when you view the link using the Edit Links command in the client application.

Creating a HIERSVR Diagram

A new HIERSVR diagram begins as a single node with the text "Root node." To add a node, first select its parent node by clicking it. Then click **Add Node** on the **Edit** menu. In the **Add Node** dialog box, specify the three attributes described above. To change the attributes of a node, select it with a mouse click, and then click **Change Node**. To move a node, click it and drag to the new location.

HIERSVR will import hierarchy information from a text file formatted as follows: The text in each line (delimited by a line feed) is the node's Description. The number of tab characters preceding the text determines the level of the node in the hierarchy. In the HIERSVR sample directory, Mfcclass.txt is the text import file for Mfcclass.hie, the native HIERSVR file.

To link a hierarchy diagram to a client document, you must first save the file in HIERSVR. Then select the node you want to copy using **Edit Copy**. This will copy the link to the Clipboard. The text of the selected node will also be copied to the Clipboard in the **CF_TEXT** format, and the entire hierarchy of nodes will be copied in a HIERSVR-native clipboard format. Saving the document is required because the link format requires a document name. Finally, in the client application, click **Edit Paste Link** (or click **Paste Link** on the **Edit Paste Special** menu). The link will be displayed in the client document as the node that you selected before clicking **Edit Copy**. The rectangle will also be displayed.

To embed a hierarchy diagram in a client document, you can either use a clipboard procedure similar to that above for linking the object, or click **Insert New Object** in the client application. If you choose the clipboard procedure, use **Edit Paste** rather than **Edit Paste Link** in the client application. Only the subhierarchy starting at the selected node will be embedded in the

client document.

Whether you link or embed the diagram, only the top selected node will be displayed in the client application window. To see the rest of the diagram, you will need to launch HIERSVR from the client application by double-clicking the item.

Visual Editing Server

HIERSVR illustrates the class derivations necessary to implement a fully functional Visual Editing server application and illustrates some of the most common client application user interfaces that require additional code.

The three framework classes used to implement a Visual Editing server application are illustrated by the following.

- A **COleTemplateServer** object, stored as member variable *m_server* in HIERSVR's application class.
- The HIERSVR **CServerDoc** class, derived from **COleServerDoc**.
- The HIERSVR **CServerItem** class, derived from **COleServerItem**.
- The HIERSVR **CInPlaceFrame** class, derived from **COleIPFrameWnd**, originally created by the application wizard.

Keywords

This sample demonstrates the following keywords:

AfxGetMainWnd; AfxMessageBox; AfxOleInit; AfxThrowMemoryException; AfxThrowNotSupportedException; AfxThrowResourceException; CArchive::Close; CCmdTarget::BeginWaitCursor; CCmdTarget::EndWaitCursor; CCmdUI::Enable; CCmdUI::SetCheck; CControlBar::EnableDocking; CControlBar::GetBarStyle; CControlBar::SetBarStyle; CDC::GetDeviceCaps; CDC::LPtoDP; CDC::LPtoHIMETRIC; CDC::SelectObject; CDC::SetMapMode; CDC::SetViewportExt; CDC::SetWindowExt; CDialog::DoModal; CDialog::OnInitDialog; CDocTemplate::SetServerInfo; CDocument::DeleteContents; CDocument::GetFirstViewPosition; CDocument::GetNextView; CDocument::OnNewDocument; CDocument::SetModifiedFlag; CDocument::UpdateAllViews; CFile::Open; CFileDialog::DoModal; CFileDialog::GetPathName; CFontDialog::DoModal; CFontDialog::GetColor; CFrameWnd::Create; CFrameWnd::DockControlBar; CFrameWnd::EnableDocking; CFrameWnd::LoadFrame; CMenu::GetSubMenu; CMenu::LoadMenu; CMenu::TrackPopupMenu; CObList::AddHead; CObList::AddTail; CObList::GetCount; CObList::GetHeadPosition; CObList::GetNext; CObList::GetTail; CObList::RemoveAll; CObList::RemoveAt; CObject::AssertValid; CObject::Dump; CObject::Serialize; COleDataObject::AttachClipboard; COleDataObject::GetFileData; COleDataObject::IsDataAvailable; COleIPFrameWnd::OnCreateControlBars; COleLinkingDoc::OnGetLinkedItem; COleServerDoc::GetItemPosition; COleServerDoc::IsInPlaceActive; COleServerDoc::OnDeactivateUI; COleServerDoc::OnGetEmbeddedItem; COleServerDoc::OnSetItemRects; COleServerDoc::RequestPositionChange; COleServerDoc::UpdateAllItems; COleServerItem::CopyToClipboard; COleServerItem::DoDragDrop; COleServerItem::GetClipboardData; COleServerItem::GetDataSource; COleServerItem::GetDocument; COleServerItem::GetItemName; COleServerItem::IsLinkedItem; COleServerItem::OnDraw; COleServerItem::OnGetClipboardData; COleServerItem::OnGetExtent; COleServerItem::OnOpen; COleServerItem::OnRenderFileData; COleServerItem::SetItemName; COleTemplateServer::ConnectTemplate; COleTemplateServer::UpdateRegistry; CPen::CreatePen; CRect::BottomRight; CRect::Height; CRect::InflateRect; CRect::IntersectRect; CRect::OffsetRect; CRect::PtInRect; CRect::Size; CRect::TopLeft; CScrollView::GetDeviceScrollPosition; CScrollView::ScrollToPosition; CScrollView::SetScrollSizes; CString::Empty; CString::GetLength; CString::IsEmpty; CToolBar::Create; CToolBar::LoadBitmap; CToolBar::SetButtons; CView::GetDocument; CView::OnDragEnter; CView::OnDragLeave; CView::OnDragOver; CView::OnDraw; CView::OnDrop; CView::OnInitialUpdate; CView::OnPrepareDC; CView::OnUpdate; CWinApp::AddDocTemplate; CWinApp::EnableShellOpen; CWinApp::ExitInstance; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWinApp::RegisterShellFileTypes; CWnd::ClientToScreen; CWnd::DoDataExchange; CWnd::GetClientRect; CWnd::GetParentFrame; CWnd::Invalidate; CWnd::InvalidateRect; CWnd::OnCreate; CWnd::OnKeyDown; CWnd::OnLButtonDblClk; CWnd::OnLButtonDown; CWnd::OnRButtonDown; CWnd::OnSize; CWnd::SetOwner; CWnd::ShowWindow; CWnd::UpdateWindow; CreateFontIndirect; DragAcceptFiles; EnableWindow; ExtTextOut; GetDeviceCaps; GetSysColor; GetTextExtent; GetWindowTextLength; IsChild; LPtoDP; LineTo; LoadBitmap; MoveTo; MulDiv; RGB; RectVisible; Rectangle; RegisterClipboardFormat; RoundRect; SelectObject; SetTextColor; SetViewportExt; SetWindowExt; SetWindowOrg; _alloca; afxMemDF; lstrcpy; memset; min; strlen; wcstombs

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

INPROC Sample: Demonstrates an In-Process Automation Server Application

Download sample

The INPROC sample is an in-process Automation (formerly OLE Automation) server. Unlike the other MFC automation server samples, INPROC can be loaded as a dynamic-link library (DLL) in the client's address space. In-process servers are usually more efficient than servers implemented as separate EXEs because a remote procedure call (RPC) is not necessary to invoke methods on the objects implemented by the server.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the INPROC sample

1. Open the solution inproc.sln.
2. On the **Build** menu, click **Build**.
3. Register INPROC's DLL using the project's .reg file, that is, either run `regedit INPROC.REG`, or use REGSVR from the REGSVR sample.

   If you use REGEDIT, be sure that Inproc.dll is on the system path. Alternately, you can modify the .reg file to refer to the path of Inproc.dll explicitly.

   A DLL cannot be run stand alone, as an EXE can; therefore, to test INPROC, you must "drive" it from a client application. INPROC can be driven from Visual Basic or from Visual C++. See the IPDRIVE sample for an example of driving INPROC from Visual C++.

Note
You can build an EXE version or a DLL version of INPROC. Set the EXE variant in the sample's project settings. An EXE version of INPROC only registers itself in the Windows registry.

INPROC Classes

The **CVariantMap** class implements a **VARIANT** to **VARIANT** map. This allows any **VARIANT** to be mapped to any other **VARIANT**. Although this is probably not useful to Visual C++ programmers (who would probably use **CMap** directly), it does bring the power of MFC's collection classes to Visual Basic users. The **CVariantMap** class is accessed by the name `mfc.inproc.varmap`. This is how the object is registered in the Windows registry. **CVariantMap** implements the standard collection methods and properties as well as the **_NewEnum** method.

The **CStringCollect** class implements an array of strings. Its implementation is simpler than the **CVariantMap** class, even though it implements many of the same automation features as **CVariantMap**. The **CStringCollect** object is accessed by the name `mfc.inproc.strcoll`. It implements most of the standard collection methods and properties and is a good example for how to implement your own collections. Of particular interest is its implementation of the **_NewEnum** method, which allows Visual Basic users to use the `For Each... In` syntax when enumerating the contents of a collection. **CStringCollect** uses **CEnumVariant** to implement this functionality (it implements **IEnumVARIANT** using the MFC interface maps). You may find **CEnumVariant** useful in your own applications.

Finally, the sample implements a few dummy properties, which are used simply to compare the performance of in-process servers and LocalServer servers. These are the properties tested by IPDRIVE's Test1 and Test2 buttons.

Keywords

This sample demonstrates the following keywords:

AfxIsValidAddress; AfxMessageBox; AfxOleInit; AfxOleLockApp; AfxOleUnlockApp; AfxThrowMemoryException; AfxThrowOleException; CCmdTarget::EnableAutomation; CCmdTarget::GetIDispatch; CCmdTarget::OnFinalRelease; CString::AllocSysString; CWinApp::InitInstance; CWinApp::RunAutomated; CWinApp::RunEmbedded; DllCanUnloadNow; DllGetClassObject; min

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

IPDRIVE Sample: Demonstrates an Automation Client Application

Download sample

The IPDRIVE sample is a simple Automation (formerly OLE Automation) client application that drives the INPROC sample application. It tests the functionality provided by INPROC, an Automation in-process server. IPDRIVE can drive both the in-process version and the out-of-process or local server version of INPROC, demonstrating the performance advantages of in-process components. See the INPROC sample for more information.

Note
Before running IPDRIVE, you must run the INPROC sample application so that INPROC registers itself in the registry.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the IPDRIVE sample

1. Follow the instructions to run, build, and register the INPROC sample.
2. Open the solution ipdrive.sln.
3. On the **Build** menu, click **Build**.
4. Run IPDRIVE as a stand-alone application so it can register itself with the system.

   When you run IPDRIVE, it will load the INPROC sample and create a variant map object that you can manipulate using IPDRIVE's interface.

A variant map object is much like the MFC **CMap** collection. It is used to map one arbitrary value to another arbitrary value. Each value is a **VARIANT**, which consists of a type and a value. The IPDRIVE user interface allows you to access both the type and the value. Any type/value pair can be mapped to any other type/value pair.

IPDRIVE also provides two buttons, **Test1** and **Test2**, which can be used to test the performance of the INPROC server application. For the most part, these functions test the function call overhead of automation. Function call overhead is expected to be much greater with the local server versions of INPROC than with the in-process versions of INPROC. A local server runs in its own address space as a separate executable. As such, it requires a remote procedure call (RPC) to access the methods and properties of its objects. An in-process server, however, loads as a DLL in the client's address space and does not require an RPC. Both the Test1 and Test2 buttons execute for exactly five seconds, then display the number of calls. The more calls, the better the performance.

INPROC also contains a simpler collection (a string array), which is not exercised by IPDRIVE.

Keywords

This sample demonstrates the following keywords:

AfxMessageBox; AfxOleInit; AfxThrowUserException; CArchive::IsStoring; CComboBox::GetCurSel; CComboBox::SetCurSel; CControlBar::EnableDocking; CControlBar::GetBarStyle; CControlBar::SetBarStyle; CDialog::DoModal; CDocument::OnNewDocument; CFrameWnd::DockControlBar; CFrameWnd::EnableDocking; CObject::AssertValid; CObject::Dump; CObject::Serialize; COleDispatchDriver::GetProperty; COleDispatchDriver::InvokeHelper; COleDispatchDriver::SetProperty; CScrollView::ResizeParentToFit; CStatusBar::Create; CStatusBar::SetIndicators; CString::AllocSysString; CString::Format; CString::GetBuffer; CString::LoadString; CString::ReleaseBuffer; CToolBar::Create; CToolBar::LoadBitmap; CToolBar::SetButtons; CView::GetDocument; CView::OnInitialUpdate; CWinApp::AddDocTemplate; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWinApp::OnFileNew; CWnd::DoDataExchange; CWnd::GetDlgItem; CWnd::GetParentFrame; CWnd::GetWindowText; CWnd::OnCreate; CWnd::SetDlgItemText; CWnd::SetWindowText;

DragAcceptFiles; GetKeyState; GetTickCount; max; wsprintf

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

MFCBIND Sample: Active Document Container

Download sample

The MFCBIND sample shows how to create an Active document (formerly known as a DocObject) container using the Active document container support classes in MFC. The MFC Binder sample is an SDI application that uses the **COleDocObjectItem** class to represent an embedded Active document in an MFC document. MFCBIND works much like the Office Binder application, which ships with Microsoft Office.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the MFCBIND sample

1. Open the solution mfcbind.sln.
2. On the **Build** menu, click **Build**.
3. Start the application and click the **Add** menu item from the **Section** menu to add to the binder as many sections as you want.

   Image, Word, and Excel documents are examples of sections. When you add a section to the binder, you can edit it directly from within the MFC Binder application as though you were working in the section's native application.
4. When you finish editing the sections in the binder, save the binder by clicking **Save** on the **File** menu. **Save** saves the sections in the binder to a single file. When you reopen the binder later, the sections are available in their previously saved state.

Active Documents in MFC

In MFC, Active documents are handled much the same way regular in-place editable embeddings are handled. The **COleDocument**-derived class still maintains a list of the currently embedded items. The **COleClientItem** object, which is replaced by the **COleClientItem**-derived **COleDocObjectItem** class, represents the embedded item in the **COleDocument**. It is these **COleDocObjectItem**-derived items that are maintained in a list by the **COleDocument**-derived class. Active documents, however, take up the entire client area of the view when they are in-place active. An Active document container also has full control of the Help menu, unlike the older in-place embeddings. The Help menu contains menu items for both the Active document container and server. Because the Active document container owns the Help menu, it is responsible for forwarding messages for the server's part of the Help menu to the server. The help menu merging and message forwarding functionality is completely handled by the MFC framework.

Keywords

This sample demonstrates the following keywords:

COleClientItem::FinishCreate, COleClientItem::GetIconFromRegistry, IOleObject::GetUserClassID, COleClientItem::OnChange, COleClientItem::OnChangeItemPosition, COleClientItem::OnGetItemPosition, COleClientItem::OnActivate, COleDocument::GetInPlaceActiveItem, COleClientItem::Deactivate, COleClientItem::CreateNewItem, COleClientItem::DoVerb, COleDocObjectItem::COleDocObjectItem, COleClientItem::CreateFromFile, COleClientItem::Activate, COleDocObjectItem::OnPreparePrintin, COleDocObjectItem::OnPrint, SystemParametersInfo, MeasureItem, DrawItem, GetSystemMetrics, SelectObject

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

MFCCALC Sample: Demonstrates an Automation Server Application

Download sample

The MFCCALC sample is an Automation (formerly OLE Automation) server. It implements a simple calculator similar to the CALC application in Windows. It can be driven with Automation with the CALCDRIV sample or run stand alone.

For more information, see the CALCDRIV sample.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the MFCCALC sample

1. Open the solution mfccalc.sln.
2. On the **Build** menu, click **Build**.
3. Run MFCCALC as a stand-alone application so it can register itself with the system.

Keywords

This sample demonstrates the following keywords:

AfxGetApp; AfxGetInstanceHandle; AfxGetThread; AfxMessageBox; AfxOleInit; CCmdTarget::EnableAutomation; CDC::DrawIcon; CDC::GetSafeHdc; CDialog::Create; CDialog::DoModal; CDialog::OnCancel; CDialog::OnInitDialog; CDialog::OnOK; CMenu::AppendMenu; CMenu::ModifyMenu; CMenu::RemoveMenu; CRect::Height; CRect::Width; CString::Format; CString::IsEmpty; CString::LoadString; CWinApp::InitInstance; CWinApp::RunAutomated; CWinApp::RunEmbedded; CWnd::DestroyWindow; CWnd::DoDataExchange; CWnd::GetClientRect; CWnd::GetDlgItem; CWnd::GetSafeHwnd; CWnd::GetStyle; CWnd::IsIconic; CWnd::OnPaint; CWnd::OnQueryDragIcon; CWnd::OnSysCommand; CWnd::PostNcDestroy; CWnd::PreTranslateMessage; CWnd::SendMessage; CWnd::SetFocus; CWnd::SetWindowText; CWnd::ShowWindow; GetSystemMenu; GetSystemMetrics; LoadAccelerators; LoadIcon; MAKEINTRESOURCE; SetFocus; TranslateAccelerator

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

OCLIENT Sample: Illustrates a Visual Editing Container Application

Download sample

The OCLIENT sample is an example of a visual editing container application. It is essentially an extended version of the CONTAINER sample, although it is not strictly a derivation of CONTAINER.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the OCLIENT sample

1. Open the solution OCLIENT.sln.
2. On the **Build** menu, click **Build**.
3. Open and run the OCLIENT application.

OCLIENT illustrates these features:

- Drag and drop to and from another application.
- Cloning (duplication) of objects with Control-Drag within the same application window.
- Automatic scrolling of window during a drag-and-drop operation.
- Edit Paste of a native format object.
- Edit Paste Link.

OCLIENT, like CONTAINER, is a simple object-drawing program. The only type of object it draws is a linked or embedded OLE item.

To add a new object to the OCLIENT drawing

1. Click **Insert New Object** on the **Edit** menu.

   The **Insert New Object** dialog box appears.

2. Select the type of OLE item you want to add.

   A new object will appear in OCLIENT's window, and OCLIENT's menu and toolbar will be updated with pop-up menus and toolbar buttons supplied by the server application.

-or-

1. While running an Automation server, copy an OLE item to the Clipboard.
2. On the **Edit** menu in OCLIENT, click **Paste** to embed the OLE item, or click **Paste Link** to link the OLE item.

The newly added OLE item is always placed in the top left corner of the OCLIENT drawing. The new OLE item might partially or completely cover an older item. You can select one OLE item at a time using the mouse. A selected linked item is indicated with a dotted rectangle, and an embedded item is indicated with a solid rectangle. You can move an OLE item by dragging it; you can resize a selected OLE item by using the resize handles.

To delete an OLE item, select it with a mouse click, then press DELETE or use the **Clear** command on the **Edit** menu.

To edit the contents of an OLE item, double-click it or select it with the mouse and then click **Edit** <type> **Object** on the **Edit** menu. To complete the editing of an in-place editing item, click somewhere outside the rectangle of the item in OCLIENT's

window. To complete the editing of an item that has been fully opened in the server application, use the server's **File Update** command.

Keywords

This sample demonstrates the following keywords:

AfxGetMainWnd; AfxMessageBox; AfxOleInit; AfxThrowArchiveException; AfxThrowFileException; AfxThrowMemoryException; CArchive::Close; CArchive::IsStoring; CCmdTarget::BeginWaitCursor; CCmdTarget::EndWaitCursor; CCmdUI::Enable; CCmdUI::SetCheck; CControlBar::EnableDocking; CDC::DPtoLP; CDC::DrawFocusRect; CDC::GetDeviceCaps; CDC::HIMETRICtoDP; CDC::LPtoDP; CDC::RealizePalette; CDC::SelectPalette; CDocTemplate::SetContainerInfo; CDocument::SetModifiedFlag; CDocument::UpdateAllViews; CFrameWnd::DockControlBar; CFrameWnd::EnableDocking; CFrameWnd::LoadFrame; CFrameWnd::OnCreateClient; CGdiObject::UnrealizeObject; CMDIChildWnd::Create; CMenu::GetSubMenu; CMenu::LoadMenu; CMenu::TrackPopupMenu; CObject::AssertValid; CObject::Dump; CObject::IsKindOf; CObject::Serialize; COleClientItem::Close; COleClientItem::CreateCloneFrom; COleClientItem::Deactivate; COleClientItem::Delete; COleClientItem::DoDragDrop; COleClientItem::DoVerb; COleClientItem::Draw; COleClientItem::GetActiveView; COleClientItem::GetClipboardData; COleClientItem::GetDocument; COleClientItem::GetDrawAspect; COleClientItem::GetInPlaceWindow; COleClientItem::GetItemState; COleClientItem::GetType; COleClientItem::IsInPlaceActive; COleClientItem::OnActivate; COleClientItem::OnChange; COleClientItem::OnChangeItemPosition; COleClientItem::OnDeactivateUI; COleClientItem::OnGetClipboardData; COleClientItem::OnGetItemPosition; COleClientItem::SetDrawAspect; COleClientItem::SetItemRects; COleClientItem::UpdateLink; COleDataObject::Attach; COleDataObject::AttachClipboard; COleDataObject::IsDataAvailable; COleDataSource::CacheGlobalData; COleDocument::EnableCompoundFile; COleDocument::GetNextItem; COleDocument::GetStartPosition; COleDocument::HasBlankItems; COleDocument::OnShowViews; COleInsertDialog::CreateItem; COleInsertDialog::DoModal; COleInsertDialog::GetSelectionType; COlePasteSpecialDialog::AddFormat; COlePasteSpecialDialog::AddStandardFormats; COlePasteSpecialDialog::CreateItem; COlePasteSpecialDialog::DoModal; COlePasteSpecialDialog::GetSelectionType; COleTemplateServer::ConnectTemplate; COleTemplateServer::UpdateRegistry; COleUpdateDialog::DoModal; CPalette::CreateHalftonePalette; CRect::InflateRect; CRect::IntersectRect; CRect::IsRectEmpty; CRect::OffsetRect; CRect::Size; CRect::TopLeft; CRectTracker::Draw; CRectTracker::HitTest; CRectTracker::SetCursor; CRectTracker::Track; CScrollView::GetDeviceScrollPosition; CScrollView::SetScrollSizes; CStatusBar::Create; CStatusBar::SetIndicators; CToolBar::Create; CToolBar::LoadBitmap; CToolBar::SetButtons; CView::DoPreparePrinting; CView::GetDocument; CView::IsSelected; CView::OnDragEnter; CView::OnDragLeave; CView::OnDragOver; CView::OnDraw; CView::OnDrop; CView::OnInitialUpdate; CView::OnPrepareDC; CView::OnPreparePrinting; CView::OnScrollBy; CView::OnUpdate; CWinApp::AddDocTemplate; CWinApp::EnableShellOpen; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWinApp::RegisterShellFileTypes; CWinApp::RunEmbedded; CWnd::ClientToScreen; CWnd::GetClientRect; CWnd::GetDC; CWnd::GetParentFrame; CWnd::InvalidateRect; CWnd::OnChar; CWnd::OnCreate; CWnd::OnDestroy; CWnd::OnLButtonDblClk; CWnd::OnLButtonDown; CWnd::OnPaletteChanged; CWnd::OnQueryNewPalette; CWnd::OnRButtonDown; CWnd::OnSetCursor; CWnd::OnSetFocus; CWnd::OnSize; CWnd::ReleaseDC; CWnd::SetFocus; CWnd::ShowWindow; CWnd::UpdateWindow; CreateHatchBrush; DeleteItem; DragAcceptFiles; FORMATETC; FillRect; GetDeviceCaps; GetKeyState; GetSysColor; GlobalFree; GlobalLock; GlobalUnlock; LPtoDP; MessageBeep; MulDiv; RGB; RectVisible; RegisterClipboardFormat; ReleaseStgMedium; STGMEDIUM; SelectPalette; SetBkColor; SetBrushOrg; SetMapMode; SetRect; SetTextColor; SetViewportExt; SetWindowExt; abs; afxMemDF; max; memset; min

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

OLEVIEW Sample: ActiveX Object Viewer

Download sample

The OLEVIEW sample illustrates how to implement ActiveX Object viewers through custom ActiveX interfaces. These custom interfaces are implemented in Iviewers.dll.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the OLEVIEW sample

1. Open the solution oleview.sln.
2. On the **Build** menu, click **Build**.

The application has an SDI interface with two panes. The left pane allows you to browse the registry by means of a tree control. Once you select a node, the right pane will display the pertinent information.

Note that this sample application cannot display information for all the objects, but rather only for objects that support the `IViewer` interface.

Keywords

This sample demonstrates the following keywords:

AfxGetInstanceHandle; AfxMessageBox; AfxOleInit; AfxThrowMemoryException; AfxThrowOleException; BitBlt; CBitmap::CreateBitmap; CBitmap::LoadBitmap; CBrush::CreateSolidBrush; CCmdTarget::BeginWaitCursor; CCmdTarget::EndWaitCursor; CDC::Attach; CDC::BitBlt; CDC::CreateCompatibleDC; CDC::DeleteDC; CDC::Detach; CDC::GetPixel; CDC::GetTextMetrics; CDC::SelectObject; CDialog::DoModal; CDialog::OnInitDialog; CDocument::OnCloseDocument; CDocument::OnNewDocument; CDocument::UpdateAllViews; CFileDialog::DoModal; CFileDialog::GetPathName; CFont::CreateFontIndirect; CFrameWnd::Create; CFrameWnd::GetActiveDocument; CFrameWnd::LoadFrame; CFrameWnd::OnCreateClient; CFrameWnd::SetActiveView; CGdiObject::DeleteObject; CListBox::SetCurSel; CObject::AssertValid; CObject::Dump; CRect::Height; CRect::Width; CScrollView::SetScrollSizes; CSplitterWnd::CreateStatic; CSplitterWnd::CreateView; CSplitterWnd::GetColumnInfo; CSplitterWnd::GetPane; CSplitterWnd::SetColumnInfo; CStatic::SetIcon; CString::Format; CString::GetBuffer; CString::IsEmpty; CString::LoadString; CString::ReleaseBuffer; CView::GetDocument; CView::OnDragEnter; CView::OnDragLeave; CView::OnDragOver; CView::OnDrop; CView::OnInitialUpdate; CView::OnUpdate; CWinApp::AddDocTemplate; CWinApp::ExitInstance; CWinApp::InitInstance; CWinApp::OnFileNew; CWinApp::SetRegistryKey; CWnd::DoDataExchange; CWnd::DragAcceptFiles; CWnd::GetParent; CWnd::GetSafeHwnd; CWnd::GetStyle; CWnd::GetWindowPlacement; CWnd::GetWindowRect; CWnd::OnCreate; CWnd::OnDestroy; CWnd::OnSize; CWnd::OnSysColorChange; CWnd::ScreenToClient; CWnd::SetFocus; CWnd::SetFont; CWnd::SetRedraw; CWnd::SetWindowPlacement; CWnd::SetWindowPos; CWnd::SetWindowText; CWnd::ShowWindow; ClientToScreen; CoCreateInstance; CoFreeUnusedLibraries; CoGetClassObject; CoGetMalloc; CreateSolidBrush; DeleteItem; DeleteObject; EnumFontFamilies; ExtTextOut; ExtractIcon; FormatMessage; FreeLibrary; FreeProcInstance; GetBkColor; GetClientRect; GetDC; GetDeviceCaps; GetNearestColor; GetObject; GetParent; GetProcAddress; GetProfileInt; GetProfileString; GetStockObject; GetSysColor; GetSystemMetrics; GetTextExtentPoint; GetTextMetrics; GetUserDefaultLCID; GetVersionEx; GetWindowLong; GetWindowRect; IClassFactory::CreateInstance; IMalloc::Free; IUnknown::QueryInterface; IUnknown::Release; IsWindow; LoadBitmap; LoadIcon; LoadLibrary; LoadLibraryEx; LocalFree; MAKEINTRESOURCE; MakeProcInstance; MulDiv; OffsetRect; Polygon; RGB; RectVisible; RegCloseKey; RegEnumKey; RegOpenKey; RegQueryValue; ReleaseDC; SelectObject; SetBkColor; SetTextColor; SetWindowPos; TabbedTextOut; WinExec; WinHelp; WriteProfileString; _fstrnicmp; atoi; isdigit; isspace; isxdigit; lstrcat; lstrcmpi; lstrcpy; lstrlen; max; strrchr; toupper; wsprintf

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

SUPERPAD Sample: Demonstrates a Visual Editing Server That Edits Text Using CEditView

Download sample

The SUPERPAD sample illustrates how to use the **CEditView** class in conjunction with Visual Editing server support.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the SUPERPAD sample

1. Open the solution superpad.sln.
2. On the **Build** menu, click **Build**.
3. To use SUPERPAD as a Visual Editing server, run it once as a stand-alone application so it can register itself with the system.

   When you first run SUPERPAD, it displays a splash window that looks like an About dialog box. This dialog box stays on the screen briefly and then vanishes after a time-out. To close the splash window, press any key or click the mouse.

   SUPERPAD remembers its window position from the last time you ran it. It also remembers whether the document window was maximized in the MDI frame window.

Additional SUPERPAD Features

SUPERPAD illustrates advanced uses of **CEditView** through class derivation, including toggling the word wrap state of the edit control and changing the screen and printer font. SUPERPAD also illustrates a wide variety of MFC programming techniques.

- About dialog box showing system resource usage.
- Splash window that is displayed when the application first starts.
- Persistent window placement by storing window position information in a private .ini file.
- Idle loop processing.
- Using a wait cursor.
- Subclassing a window.
- Adding page headers and footers during printing.
- Page breaking during printing.
- Starting print preview at the current selection in the document.
- Returning from print preview to the same position in the document as last previewed.
- Standard font dialog box (**CFontDialog**).

SUPERPAD Menus

The **File** menu offers standard commands — **New**, **Open**, **Close**, **Save**, **Save As**, **Print**, **Print Preview**, **Print Setup**, the most recently used (MRU) file list, and **Exit**.

The **File** menu also offers a **Page Setup** option you can use to specify a per-page header and/or footer. Leave the header or footer blank if you want none. You can specify any text in the header and footer. You can also specify any of the following formatting codes.

- &f for the file name.
- &p for the page number.
- Any of the formatting codes defined for the C run-time strftime function. For example, %a for the abbreviated weekday name, %B for the full month name, %d for the day of the month, and %Y for the year with century can be specified. These formatting options are not available if you link SUPERPAD to the DLL version of MFC because **strftime** is not available in a DLL.

SUPERPAD offers the following **Edit** menu commands, which are handled by default by **CEditView**: **Cut**, **Copy**, **Page**, **Delete**, **Find**. **Find Next**, **Replace**, **Select All**, and **Undo**. The **Edit** menu also offers a **Word Wrap** command to turn word wrapping on or off.

The **View** menu offers the following commands, which exercise various features of **CEditView**:

- **Set Tab Stops** — enter a single integer in the dialog box for equally spaced tab stops.
- **Set Font** — specify the font to be used when displaying the file on the screen.
- **Set Printer Font** — specify the font to be used when printing.
- **Mirror Display Font** — turn this toggle state on if you want to temporarily deselect the currently specified printer font and instead use a printer font that most closely matches the screen font.

SUPERPAD Classes

SUPERPAD derives its view class **CPadView** from **CEditView**. **CPadView** accesses the attributes and operations of **CEditView** to add to the functionality of **CEditView**, as described here.

CPadView calls **CEditView::SetFont** and **SetPrinterFont** to initialize and change the screen and printer fonts.

To implement the **Edit Word Wrap** command, **CPadView** toggles the state of word wrapping by creating a new edit control (whose **WNDCLASS** is the standard Windows edit control). It toggles the **ES_AUTOHSCROLL** and **WS_HSCROLL** styles of the control, which together determine whether an edit control does word wrapping. **CPadView** saves the buffer of the old edit control and moves it into the new edit control. Finally, it destroys the old edit control and then associates the *hWnd* of the new edit control with the **CPadView** class by calling **CWnd::SubclassWindow**.

CPadView overrides **OnPrint** to add page headers and footers. It overrides **OnBeginPrinting** to set the current time, which is optionally displayed in the header, in the footer, or in both. It starts print preview at the current selection in the edit control.

CPadView overrides **CView::OnScrollTo** to scroll the view to the same position as last viewed during print preview.

Keywords

This sample demonstrates the following keywords:

AfxGetApp; AfxGetInstanceHandle; AfxMessageBox; AfxOleInit; CArchive::Flush; CArchive::GetFile; CArchive::IsStoring; CBitmap::CreateCompatibleBitmap; CButton::DrawItem; CCmdUI::SetCheck; CDC::Attach; CDC::CreateCompatibleDC; CDC::Detach; CDC::DrawIcon; CDC::FillRect; CDC::GetDeviceCaps; CDC::GetTextMetrics; CDC::IntersectClipRect; CDC::LPtoHIMETRIC; CDC::LineTo; CDC::MoveTo; CDC::Rectangle; CDC::RestoreDC; CDC::SaveDC; CDC::SelectObject; CDC::SetBkMode; CDC::SetWindowExt; CDC::SetWindowOrg; CDC::StretchBlt; CDC::TextOut; CDialog::Create; CDialog::DoModal; CDialog::OnInitDialog; CDocTemplate::SetServerInfo; CDocument::DeleteContents; CDocument::GetFirstViewPosition; CDocument::GetNextView; CEditView::GetEditCtrl; CEditView::GetPrinterFont; CEditView::PrintInsideRect; CEditView::SerializeRaw; CEditView::SetPrinterFont; CEditView::SetTabStops; CEditView::dwStyleDefault; CFile::Write; CFont::CreateFontIndirect; CFontDialog::DoModal; CFrameWnd::ActivateFrame; CFrameWnd::Create; CFrameWnd::LoadFrame; CGdiObject::CreateStockObject; CGdiObject::DeleteObject; CGdiObject::GetObject; CObject::AssertValid; CObject::Dump; CObject::IsKindOf; CObject::Serialize; COleIPFrameWnd::OnCreateControlBars; COleLinkingDoc::OnGetLinkedItem; COleResizeBar::Create; COleServerDoc::IsInPlaceActive; COleServerDoc::OnDeactivateUI; COleServerDoc::OnGetEmbeddedItem; COleServerDoc::UpdateAllItems; COleServerItem::CopyToClipboard; COleServerItem::GetDataSource; COleServerItem::GetDocument; COleServerItem::OnDraw; COleServerItem::OnGetExtent; COleServerItem::OnRenderFileData; COleServerItem::OnShow; COleServerItem::SetItemName; COleTemplateServer::ConnectTemplate; COleTemplateServer::UpdateRegistry; CPrintDialog::CreatePrinterDC; CPrintInfo::SetMaxPage; CRect::Height; CRect::InflateRect; CRect::IsRectEmpty; CRect::SetRect; CRect::Size; CRect::Width; CStatusBar::Create; CStatusBar::SetIndicators; CString::Find; CString::GetBuffer; CString::GetLength; CString::IsEmpty; CString::Left; CString::LoadString; CString::Mid; CString::ReleaseBuffer; CTime::Format; CToolBar::Create; CToolBar::LoadBitmap; CToolBar::SetButtons; CView::GetDocument; CView::OnBeginPrinting; CView::OnPrepareDC; CView::OnPrint; CWinApp::AddDocTemplate; CWinApp::EnableShellOpen; CWinApp::ExitInstance; CWinApp::GetProfileInt; CWinApp::GetProfileString; CWinApp::InitInstance; CWinApp::LoadStdProfileSettings; CWinApp::OnIdle; CWinApp::PreTranslateMessage; CWinApp::RegisterShellFileTypes; CWinApp::WriteProfileInt; CWinApp::WriteProfileString;

CWnd::BringWindowToTop; CWnd::CenterWindow; CWnd::DestroyWindow; CWnd::Detach; CWnd::DoDataExchange; CWnd::FromHandle; CWnd::GetClientRect; CWnd::GetDC; CWnd::GetDlgCtrlID; CWnd::GetFocus; CWnd::GetFont; CWnd::GetParent; CWnd::GetParentFrame; CWnd::GetStyle; CWnd::GetSuperWndProcAddr; CWnd::GetWindowPlacement; CWnd::GetWindowRect; CWnd::GetWindowText; CWnd::GetWindowTextLength; CWnd::IsZoomed; CWnd::KillTimer; CWnd::OnClose; CWnd::OnCreate; CWnd::OnEraseBkgnd; CWnd::OnRButtonDown; CWnd::OnSize; CWnd::OnTimer; CWnd::PreCreateWindow; CWnd::ReleaseDC; CWnd::ScreenToClient; CWnd::SendMessage; CWnd::SetDlgCtrlID; CWnd::SetDlgItemText; CWnd::SetFocus; CWnd::SetFont; CWnd::SetOwner; CWnd::SetTimer; CWnd::SetWindowPlacement; CWnd::SetWindowPos; CWnd::SetWindowText; CWnd::ShowWindow; CWnd::SubclassDlgItem; CWnd::SubclassWindow; CWnd::UpdateWindow; CreateWindow; DragAcceptFiles; GetCurrentTime; GetObject; GetProfileInt; GetProfileString; GetStockObject; GetSystemMetrics; GlobalMemoryStatus; IsIconic; IsWindowVisible; LOWORD; LoadIcon; MulDiv; SetWindowLong; SetWindowPos; UpdateWindow; WideCharToMultiByte; WriteProfileString; _alloca; _getdrive; lstrcpyn; lstrlen; min; wcstombs; wsprintf

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

TSTCON Sample: ActiveX Control Test Container

Download sample

The TSTCON sample implements an ActiveX control container using MFC support for OLE embedding. You can use TSTCON to test ActiveX controls, change their properties, and invoke their methods. You can write scripts using the VBScript language to automate the testing of the controls. TSTCON can keep a log of the events and property change notifications fired by a control.

Note
This sample requires the Microsoft Calendar Control, which is installed with Microsoft Access.

TSTCON also demonstrates several MFC programming topics, including the following topics:

- CCheckListBox: A list box with a check box next for each item.
- CDragListBox: A list box in which you can drag the items to rearrange their order.
- Implementing an Active Scripting Engine host (VBScript).
- Implementing context-sensitive help for dialog boxes.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the TSTCON sample

1. Open the solution tstcon.sln.
2. On the **Build** menu, click **Build Solution**.
3. Run the test container application as a stand-alone application so it can register itself with the system.

Alternately, you can open the sample test container application from the **Tools** menu in Visual Studio, as follows:

To launch a control in the container

1. In Visual Studio, from the **Tools** menu, launch the **ActiveX Control Test Container**.
2. In **ActiveX Control Test Container**, from the **Edit** menu, click **Insert New Control**.
3. In the **Insert New Control** dialog box, from the list box, select **Calendar Control**, and click **OK**.

   The Microsoft Calendar Control appears in the upper half of the splitter window. Note that this control is installed with Microsoft Access.
4. Manipulate the calendar by changing month, year, and date. In the lower pane, observe how the logs, events, and property change notifications are fired by the control as you manipulate it.

To change the control's properties

1. Select the calendar control by clicking the calendar's border.
2. From the **Edit** menu, click **Properties**. Test Container displays the **Properties** dialog box for the control.
3. Using this dialog box, you can edit the properties of the control. Change the control's properties and observe the changes.
4. Close the **Properties** dialog box.

To invoke methods on the control

1. From the **Control** menu, click **Invoke Methods**. The **Invoke Method** dialog box appears.
2. In the **Method Name** combo box, select **BackColor (PropPut)**.
3. From the **Parameter Type** combo box, select **VT_COLOR**.
4. Click **Choose Color**, choose a color from the color palette, and click **OK**.
5. In the **Invoke Method** dialog box, click **Invoke**. The background color of the calendar control changes to the color you selected in the color palette.

For more information on how to use the Test Container, from the **Help** menu, click **Help Topics**.

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

WORDPAD Sample: The Windows Application

Download sample

WORDPAD is the word processing application included with Windows. It is an example of a full-fledged word processing application written using MFC.

WORDPAD comes with a help file describing its user interface.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Building and Running the Sample

To build and run the WORDPAD sample

1. Open the solution wordpad.sln.
2. On the **Build** menu, click **Build**.
3. Run WordPad as a stand-alone application so it can register itself with the system.

WORDPAD Features

The SUPERPAD sample illustrates a basic word processor based on the **CEditView** class. In contrast, WORDPAD is based on the **CRichEditView**, **CRichEditDoc**, and **CRichEditCntrItem** classes, and — besides supporting the features that SUPERPAD supports — also demonstrates the following features.

- The ability to read, write, and convert between multiple file formats, including the Word file format, .rtf (Rich Text Format) format, and ASCII text format.
- Text formatting, including the use of fonts, character formatting (bold, italics, underlining), color, and point sizes. These attributes are available in a formatting toolbar.
- Per-paragraph formatting, including alignment (left, center, or right), tabs, margins, and first-line indentation. These attributes are also available in a ruler toolbar.
- Undo support.
- ActiveX component support, both container and server.

Keywords

This sample demonstrates the following keywords:

AfxBeginThread; AfxFormatString1; AfxFormatString2; AfxGetApp; AfxGetAppName; AfxGetInstanceHandle; AfxGetMainWnd; AfxMessageBox; AfxOleGetMessageFilter; AfxOleInit; AfxOleSetUserCtrl; AfxRegisterClass; AfxRegisterWndClass; AfxThrowFileException; AfxThrowNotSupportedException; AfxThrowResourceException; BitBlt; CArchive::Close; CArchive::Read; CArchive::Write; CBrush::CreateSolidBrush; CButton::DrawItem; CComboBox::AddString; CComboBox::CompareItem; CComboBox::Create; CComboBox::DrawItem; CComboBox::FindString; CComboBox::FindStringExact; CComboBox::GetCount; CComboBox::GetCurSel; CComboBox::GetDroppedState; CComboBox::GetItemData; CComboBox::GetItemHeight; CComboBox::GetLBText; CComboBox::InsertString; CComboBox::LimitText; CComboBox::MeasureItem; CComboBox::ResetContent; CComboBox::SetCurSel; CComboBox::SetItemData; CComboBox::SetItemHeight; CComboBox::ShowDropDown; CControlBar::EnableDocking; CDC::Attach; CDC::BitBlt; CDC::CreateCompatibleDC; CDC::CreateIC; CDC::Detach; CDC::DrawFocusRect; CDC::DrawText; CDC::ExtTextOut; CDC::FillRect; CDC::GetBkColor; CDC::GetDeviceCaps; CDC::GetTextColor; CDC::GetTextExtent; CDC::GetTextMetrics; CDC::LineTo; CDC::MoveTo; CDC::Rectangle; CDC::RestoreDC; CDC::SaveDC; CDC::SelectObject; CDC::SetBkMode; CDC::SetTextColor; CDC::SetViewportOrg; CDC::TextOut; CDialog::Create; CDialog::DoModal; CDialog::EndDialog; CDialog::GetDefID; CDialog::InitModalIndirect; CDialog::OnInitDialog; CDialog::OnOK; CDialog::SetDefID; CDocTemplate::GetDocString; CDocument::GetFirstViewPosition; CDocument::GetNextView; CFile::Abort; CFile::Close; CFile::Flush; CFile::GetLength; CFile::GetPosition; CFile::GetStatus; CFile::LockRange; CFile::Open; CFile::Read; CFile::Seek; CFile::SeekToBegin; CFile::SetLength; CFile::UnlockRange; CFile::Write; CFileDialog::DoModal;

CFont::CreateFontIndirect; CFrameWnd::ActivateFrame; CFrameWnd::Create; CFrameWnd::DockControlBar; CFrameWnd::EnableDocking; CFrameWnd::GetActiveDocument; CFrameWnd::GetActiveView; CFrameWnd::RecalcLayout; CFrameWnd::SetMessageText; CGdiObject::Attach; CGdiObject::CreateStockObject; CListBox::AddString; CListBox::GetCurSel; CListBox::SetCurSel; CMenu::AppendMenu; CMenu::CreatePopupMenu; CMenu::DeleteMenu; CMenu::Detach; CMenu::DrawItem; CMenu::FromHandle; CMenu::GetMenuItemCount; CMenu::GetSubMenu; CMenu::LoadMenu; CMenu::MeasureItem; CMenu::RemoveMenu; CMenu::TrackPopupMenu; CObject::AssertValid; CObject::Dump; CObject::Serialize; COleDataObject::Attach; COleDataObject::IsDataAvailable; COleDataSource::CacheData; COleDropTarget::Register; COleIPFrameWnd::OnCreateControlBars; COleIPFrameWnd::RepositionFrame; COleMessageFilter::EnableBusyDialog; COleResizeBar::Create; COleServerItem::GetClipboardData; COleServerItem::GetDocument; COleServerItem::OnDraw; COleServerItem::OnDrawEx; COleServerItem::OnGetExtent; COleStreamFile::OpenStream; COleTemplateServer::ConnectTemplate; CPalette::GetPaletteEntries; CPoint::Offset; CRect::BottomRight; CRect::CopyRect; CRect::Height; CRect::IntersectRect; CRect::OffsetRect; CRect::PtInRect; CRect::SetRect; CRect::SetRectEmpty; CRect::Size; CRect::TopLeft; CRect::Width; CStatusBar::Create; CStatusBar::SetIndicators; CString::Collate; CString::Empty; CString::FindOneOf; CString::GetBuffer; CString::GetBufferSetLength; CString::GetLength; CString::IsEmpty; CString::LoadString; CString::MakeUpper; CString::ReleaseBuffer; CString::Right; CToolBar::CommandToIndex; CToolBar::Create; CToolBar::GetItemRect; CToolBar::LoadBitmap; CToolBar::SetButtonInfo; CToolBar::SetButtons; CToolBar::SetSizes; CWinApp::ExitInstance; CWinApp::GetProfileInt; CWinApp::InitInstance; CWinApp::LoadIcon; CWinApp::LoadStdProfileSettings; CWinApp::OnDDECommand; CWinApp::OnFileNew; CWinApp::OnFileOpen; CWinApp::OpenDocumentFile; CWinApp::PreTranslateMessage; CWinApp::SetRegistryKey; CWinApp::WinHelp; CWinApp::WriteProfileInt; CWnd::BringWindowToTop; CWnd::CalcWindowRect; CWnd::CenterWindow; CWnd::ClientToScreen; CWnd::Create; CWnd::DestroyWindow; CWnd::DoDataExchange; CWnd::EnableWindow; CWnd::FromHandle; CWnd::GetCapture; CWnd::GetClientRect; CWnd::GetDlgCtrlID; CWnd::GetDlgItem; CWnd::GetFocus; CWnd::GetNextDlgTabItem; CWnd::GetOwner; CWnd::GetParent; CWnd::GetParentFrame; CWnd::GetSafeHwnd; CWnd::GetScrollPos; CWnd::GetStyle; CWnd::GetWindowPlacement; CWnd::GetWindowRect; CWnd::GetWindowTextLength; CWnd::Invalidate; CWnd::InvalidateRect; CWnd::IsChild; CWnd::IsWindowEnabled; CWnd::MoveWindow; CWnd::OnCommand; CWnd::OnCreate; CWnd::OnDestroy; CWnd::OnDevModeChange; CWnd::OnDropFiles; CWnd::OnEnable; CWnd::OnEraseBkgnd; CWnd::OnFontChange; CWnd::OnLButtonDown; CWnd::OnLButtonUp; CWnd::OnMouseMove; CWnd::OnMove; CWnd::OnPaletteChanged; CWnd::OnQueryNewPalette; CWnd::OnShowWindow; CWnd::OnSize; CWnd::OnSysColorChange; CWnd::OnWindowPosChanged; CWnd::OnWindowPosChanging; CWnd::PreCreateWindow; CWnd::PreTranslateMessage; CWnd::ScreenToClient; CWnd::SendMessage; CWnd::SetCapture; CWnd::SetDlgItemText; CWnd::SetFocus; CWnd::SetOwner; CWnd::SetWindowPos; CWnd::SetWindowText; CWnd::ShowWindow; CWnd::SubclassDlgItem; CWnd::UpdateWindow; CloseHandle; CommDlgExtendedError; CreateEvent; CreateFontIndirect; CreateILockBytesOnHGlobal; CreatePen; CreateSolidBrush; DeleteAtom; DeleteObject; DragAcceptFiles; DragFinish; DragQueryFile; EnumDateFormats; EnumFontFamilies; EnumTimeFormats; EnumWindows; FORMATETC; FindResource; FindWindow; FormatMessage; FreeLibrary; GetClassInfo; GetClassName; GetClientRect; GetClipboardData; GetDC; GetDateFormat; GetDeviceCaps; GetDlgItem; GetKeyState; GetLocalTime; GetLocaleInfo; GetModuleFileName; GetModuleHandle; GetObject; GetProcAddress; GetProfileInt; GetShortPathName; GetStockObject; GetSysColor; GetSystemMetrics; GetTextMetrics; GetTimeFormat; GetUserDefaultLCID; GetVersion; GetWindow; GetWindowText; GlobalAddAtom; GlobalAlloc; GlobalFree; GlobalGetAtomName; GlobalLock; GlobalSize; GlobalUnlock; HIWORD; InvalidateRect; KillTimer; LOWORD; LineTo; LoadBitmap; LoadCursor; LoadIcon; LoadLibrary; LoadString; MAKEINTRESOURCE; MAKELPARAM; MessageBeep; MoveTo; MsgWaitForMultipleObjects; MulDiv; OleInitialize; OleUninitialize; OpenFile; PeekMessage; PtInRect; RGB; ReadClassStg; RegCloseKey; RegCreateKey; RegOpenKey; RegQueryValueEx; RegSetValueEx; RegisterClipboardFormat; ReleaseCapture; ReleaseDC; ReleaseStgMedium; ResetEvent; RestoreDC; RoundRect; STGMEDIUM; SaveDC; SelectObject; SendMessage; SendMessageTimeout; SetEvent; SetFocus; SetForegroundWindow; SetROP2; SetRect; SetTimer; SetWindowPos; SetWindowText; ShellAbout; ShowWindow; StgCreateDocfileOnILockBytes; StgOpenStorage; WinHelp; div; isspace; lstrcat; lstrcmp; lstrcmpi; lstrcpy; lstrlen; max; memcmp; memcpy; memmove; memset; min; strcpy; wsprintf

Note
Some samples, such as this one, have not been modified to reflect the changes in the Visual C++ wizards, libraries, and compiler, but still demonstrate how to complete your desired task.

See Also

Other Resources

MFC Samples

Platform SDK Samples

The stand-alone Platform Software Development Kit includes working samples that demonstrate several key technologies, including DirectX, Internet Information Services, Windows Installer, and the core SDK. The stand-alone Platform SDK is available on the Web at http://www.microsoft.com/msdownload/platformsdk/sdkupdate/.

In This Section

GetImage Sample: Demonstrates the Windows Image Acquisition API

Demonstrates the Windows Image Acquisition (WIA) application programming interfaces (APIs).

See Also

Other Resources

Visual C++ Samples

GetImage Sample: Demonstrates the Windows Image Acquisition API

Download sample

The GetImage sample demonstrates the Windows Image Acquisition (WIA) application programming interfaces (APIs).

The WIA API set contains simple APIs for file transfers, but there is no simple API call for in-memory transfers. GetImage implements a wrapper function named WiaGetImage that combines the functionality of the SelectDeviceDlg, DeviceDlg, and idtGetBandedData APIs to completely encapsulate image acquisition within a single function call.

The main feature of the sample is the code itself. The files WiaWrap, EventCallback, DataCallback, BitmapUtil, and ProgressDlg show the correct way to implement the in-memory image transfers. The "Notes on Programming with the WIA API" section later in this abstract highlights the common pitfalls while writing a WIA application, and the sample shows how to handle these. You can read this sample as a reference or directly use the sample code as a library. Usage of each function is described in detail in the header file comments.

Security Note
This sample code is provided to illustrate a concept and should not be used in applications or Web sites, as it may not illustrate the safest coding practices. Microsoft assumes no liability for incidental or consequential damages should the sample code be used for purposes other than as intended.

Requirements

This sample can only be run on Windows XP and later.

The Microsoft Platform SDK must be installed.

Building and Running the Sample

To build and run this sample

1. In Visual Studio, load the solution GetImage.sln.
2. In Solution Explorer, right-click the project and click **Properties**. Set the following properties:
   - **C++** folder, **General** property page: modify the **Additional Include Directories** property to add the directory which contains the Platform SDK header files (for example, c:\Microsoft Platform SDK\include).
   - **Linker** folder, **General** property page: modify the **Additional Library Directories** property to add the directory which contains the Platform SDK library files (for example, c:\Microsoft Platform SDK\lib).
3. From the **Build** menu, click **Build**.
4. From the **Debug** menu, click **Start Without Debugging**.

The sample application has a single command on its **File** menu, named **From scanner or camera**. When a WIA device (or a device emulator) is attached, the menu item becomes enabled. When the user selects the menu command, the sample will sequentially display the WIA Device Selection dialog box, Image Selection dialog box, and Image Transfer dialog box. The device and image selection dialog boxes are the common dialog boxes supplied by the system, and the transfer dialog box is implemented in this sample. Finally, the sample will display the transferred image(s) in child window(s).

Files in the Sample

File	Description
WiaWrap.cpp	Contains wrapper functions for simpler WIA operations
WiaWrap.h	Header file for WiaWrap.cpp
EventCallback.cpp	Implements the callback function for WIA device events

EventCallback.h	Header file for EventCallback.cpp
DataCallback.cpp	Implements the callback function for data transfers
DataCallback.h	Header file for DataCallback.cpp
BitmapUtil.cpp	Contains device independent bitmap (DIB) functions
BitmapUtil.h	Header file for BitmapUtil.cpp
ProgressDlg.cpp	Implements a simple progress dialog
ProgressDlg.h	Header file for ProgressDlg.cpp
GetImage.cpp	Implements the main entry point for the application
GetImage.rc	Contains the project resources
resource.h	Header file for the resource identifiers
MainWnd.cpp	Implements the main frame window
MainWnd.h	Header file for MainWnd.cpp
BitmapWnd.cpp	Implements an image display window
BitmapWnd.h	Header file for BitmapWnd.cpp
StdAfx.cpp	Precompiled headers file
StdAfx.h	Includes frequently used standard system files

Notes on Programming with the WIA API

It is recommended that applications make device and image selection available through a menu item named "From Scanner or Camera..." on the "File" menu. This item could be grayed if there are no WIA devices on the system.

It is recommended that application developers test their applications with serial and USB cameras, and flatbed, scroll-fed, and ADF scanners. Also, software-only device emulators available on the Windows DDK can be used for testing.

Scroll-fed scanners generally do not know the image height when the data transfer starts, so they may return 0 for the image size and for the image height in the bitmap header. In this case, the callback function should be able to expand its buffer when new data arrives and should explicitly calculate the height when the transfer completes.

When using the DeviceDlg API on an automatic document feeder (ADF), the API will set the ADF to scan one page. If multiple pages are desired, the application should explicitly set WIA_DPS_PAGES to the number of pages it requests or to the ALL_PAGES value.

The SelectDeviceDlg, DeviceDlg, idtGetData, and idtGetBandedData APIs return S_FALSE if the user cancels, so it is not sufficient to check the return value with the SUCCEEDED macro. The programmer must explicitly check S_FALSE. Similarly, the ReadMultiple and WriteMultiple APIs return S_FALSE when the function arguments are correct, but the function cannot perform the requested operation, so the programmer must explicitly check the return value against S_FALSE.

If the application wants to display a progress dialog box that can be canceled, it should display the dialog box in a separate thread. The data transfer thread will be blocked until the transfer is complete, so it will not be able to process the window messages as soon as they arrive. If the progress dialog box is created in the same thread, the dialog box (and the cancel button) will be very sluggish to user input.

The data transfer can be canceled only when BandedDataCallback returns. Since BandedDataCallback may be called every few

seconds, the application should indicate to the user that the cancel operation is in progress. For example, it can disable the cancel button, or display a wait message.

Keywords

This sample demonstrates the following keywords:

IWiaDevMgr, IWiaDevMgr::SelectDeviceDlg, IWiaEventCallback, IWiaEventCallback::ImageEventCallback, IWiaItem, IWiaItem::DeviceDlg, IWiaPropertyStorage, IWiaPropertyStorage::ReadMultiple, IWiaPropertyStorage::WriteMultiple, IWiaDataTransfer, IWiaDataTransfer::idtGetBandedData, IWiaDataCallback, IWiaDataCallback::BandedDataCallback

See Also

Other Resources

Visual C++ Samples